UNIVERSITÉ DE SHERBROOKE
Faculté des sciences appliquées
Département de génie électrique et informatique

OBJETS DISTRIBUÉS POUR SYSTÈMES EMBARQUÉS

Mémoire de maîtrise ès sciences appliquées
Spécialité : génie électrique

Samuel Guénette, ing. stag.

Sherbrooke (Québec), CANADA Le 9 octobre 2002

Résumé

La distribution des objets constitue une technique fort intéressante dans la réalisation de certains systèmes informatiques faisant appel à la réseautique. Un objet distribué correspond à un objet localisé sur différents nœuds du réseau et dont les opérations peuvent être appelées à distance, et ce, de façon transparente pour l'utilisateur. Les technologies les plus populaires mettant en œuvre ce concept, sont certainement celles de l'OMG (CORBA) et de Microsoft (DCOM).

Avec l'avancement exponentiel des technologies de communication, de plus en plus de systèmes embarqués se retrouvent à la périphérie des réseaux IP. Ce genre de système, possédant généralement peu de ressource matérielle, pourrait certainement tirer avantage du concept d'objets distribués, mais malheureusement les standards et les technologies actuels sont parfois non propices et souvent inutilisables pour de tels systèmes. C'est donc dire que dans la majorité des cas, il n'est pas requis ou même il est carrément inadéquat d'adhérer au formalisme de CORBA ou de DCOM, car ces standards sont lourds et n'ont pas été conçus, à la base, pour des systèmes de moindres tailles. Face aux systèmes embarqués, les technologies actuelles n'offrent évidemment pas de réelle solution pour la distribution des objets.

Comme les standards actuels pour la distribution des objets ne répondent pas, à ce jour, à tous les besoins d'un système embarqué, il devient pertinent de développer une solution propriétaire offrant le support nécessaire à la création de systèmes à objets distribués. Le but du projet, présenté par l'intermédiaire de ce mémoire, est donc de concevoir un protocole de communication pour la distribution des objets ainsi qu'une librairie de classes C++ facilitant le développement et la programmation de systèmes embarqués à objets distribués. Le protocole développé convoite simplicité et légèreté afin de satisfaire aux contraintes des systèmes embarqués pour qui les ressources matérielles sont limitées. Quant à la librairie de classes, elle exploite le concept d'héritage pour permettre une distribution quasi transparente des objets. En fait, pour arriver à tirer avantage des services d'un objet distant, il suffit de créer un *Proxy* et un *Adaptor*, spécifique à cet objet, et de les intégrer au système. L'objet client peut alors faire appel aux différentes opérations de l'objet serveur, et ce, même s'ils ne sont pas physiquement localisés sur le même nœud. Le *Proxy* et l'*Adaptor*, aidés des classes internes à la librairie, font en sorte d'abstraire les contraintes et les limites physiques du système dans son ensemble.

Ce mémoire, dans un premier temps, aborde la notion d'objets distribués et présente le standard CORBA. Dans un deuxième temps, les besoins auxquels doit répondre le système sont définis, le protocole est spécifié et les classes de la librairie sont décrites. Pour terminer, les étapes de vérification du protocole par *Model Checking* ainsi que les tests d'intégration et de distribution d'une application de messagerie vocale sont exposés.

Remerciements

Tout d'abord, j'aimerais remercier Manon, Alyssia et Ismael qui ont accepté et supporté mon absence psychique plus souvent qu'autrement depuis trop longtemps déjà. J'ai apprécié grandement votre patience et je vous remercie affectueusement d'être près de moi!

Je voudrais aussi souligner la patience et le support manifestés par mon directeur, Monsieur Philippe Mabilleau. Merci Philippe d'avoir cru en mon potentiel et d'avoir su me laisser terminer mes « devoirs » avec un « peu » de retard.

Merci à mes collègues de travail faisant partie de la compagnie Médiatrix, sans qui je n'aurais pu avoir toutes ces discussions philosophiques au sujet des objets, des systèmes embarqués et de la réseautique.

Pour terminer, il m'est impossible de ne pas placer un mot agréable pour mes correcteurs de syntaxe et d'orthographe sans qui ce texte n'offrirait pas la même plaisance et douceur de lecture. Merci Yolande et Gilles!

Table des matières

Liste des figures

Liste des tableaux

Lexique

Bit	Valeur binaire : 0 ou 1.
BDER	Basic Date Encoding Rules
Blob	Binary Large Object
Byte	Voir octet
CDR	Common Data Representation
COM	Components Object Modeling
CORBA	Common Object Request Broker Architecture
DCOM	Distributed COM
DOES	Distributed Objects for Embedded System
DOOM	Distributed Object-Oriented Messaging
IDL	Interface Description Language
IOR	Interoperable Object Reference
IP	Internet Protocol
IPC	Inter-Process Communication
LAN	Local Area Network
LSB	Least Significant Byte
Marshaling	Action de sérialisation des paramètres d'une opération.
MSB	Most Significant Byte
MTU	Maximum Transfer Unit
OA	Object Adapter
Octet	Valeur entière représentée par 8-bits.
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
STA	Single Thread Apartment
TCP	Transport Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
WAN	Wide Area Network

1. Introduction

De tous les secteurs d'activité entourant l'informatique, il est probablement justifié de dire que, celui de la réseautique, a connu la plus grande croissance durant les dernières années. Bien entendu, les techniques de transport de données ont évolué rapidement pour atteindre, à ce jour, des débits difficilement imaginables il y a cinq ans à peine. Par exemple, un lien sur fibre optique peut transporter plusieurs milliards de bits de données par seconde.

L'évolution rapide des réseaux s'est fait sentir aussi du côté des applications logiciels. Depuis leur apparition, les idées, les recherches et le développement entourant les applications distribuées n'ont cessé de progresser. Le travail en collaboration, l'échange de documents, le partage d'informations au sein d'une même entreprise localisée sur plusieurs sites distants, constituent quelques exemples des nombreux avantages offerts par les plus récentes applications.

Avec la popularité toujours croissante de l'Internet ainsi que l'avancement exponentiel des technologies composant ce réseau, il est normal d'y voir de plus en plus d'applications de tous genres. Par exemple, les systèmes embarqués sont de plus en plus présents à la périphérie du réseau et ce phénomène ne cessera de croître. En fait, l'une des plus grandes réalisations de l'être humain est sûrement le réseau téléphonique mondial : ce réseau interconnecte des millions de téléphones, de fax et de modems autour du monde avec une fiabilité de l'ordre de 99.999 %. Le seul fait de décrocher le combiné d'un téléphone au Québec et d'effectuer un appel en Europe implique la coopération de dizaines de commutateurs, de liens à haut débit sur fibre optique, de satellites et de systèmes de facturation tous produits par autant de fournisseurs différents.

Présentement, chaque type d'application possède son propre protocole. Afin de pouvoir interagir entre elles, deux applications doivent nécessairement comprendre et parler le même langage. Les protocoles sont généralement standardisés par des organismes comme l'IETF (*Internet Engineering Task Force*) ou l'ITU (*International Telecommunication Union*) : cet effort est essentiel, puisque sans standard, aucune interopérabilité ne serait possible entre les différentes implémentations d'un même genre d'applications.

Il faut dire qu'habituellement, un protocole Internet se présente sous la forme d'états, de commandes et de paramètres. Ces commandes sont échangées entre le client et le serveur, au niveau du réseau, sous un format texte ou binaire[1]. Les caractères ainsi reçus sont décodés et analysés par l'application qui, par la suite, exécute une action. Pour en arriver à l'exécution de la commande proprement dite, il y a généralement plusieurs étapes de traitement. Par exemple, le client doit générer la commande sous un format texte et l'envoyer sur le réseau à l'aide de fonctions qui s'interfacent avec la couche transport (ex : BSD Sockets). Une fois que le serveur a reçu le paquet, il doit le décoder et exécuter le service approprié. Les applications doivent aussi s'assurer qu'elles respectent les différents états du protocole, en plus d'être solides et résistantes aux attaques potentielles en provenance du réseau.

[1] L'IETF préconise le format texte (US-ASCII), tandis que l'ITU préconise un format binaire (ASN.1)

Il est évident que certaines opérations de bas niveau, même si elles ne sont pas totalement identiques d'une application à l'autre, sont généralement les mêmes. Il paraît donc assez fastidieux et très peu productif de toujours réinventer la roue pour chaque type de protocole. Il serait beaucoup plus approprié de faire abstraction du réseau et de se concentrer sur le cœur de l'application : le fait d'appeler directement les fonctions d'interfaces des objets serveurs, indépendamment de leur localisation, serait un avantage énorme. D'ailleurs, c'est ce que des groupes comme OMG (CORBA) et Microsoft (DCOM) cherchent à faire. Avec une telle approche, la définition d'un protocole se résume à la description des capacités de l'interface d'un objet pouvant être utilisé de façon distribuée.

Comme cette technologie est déjà existante et fonctionnelle, il est pertinent de poser la question suivante : « Pourquoi ne pas utiliser une implémentation commerciale, ou même gratuite, d'une telle technologie pour la réalisation de futures applications distribuées? » Dans la majorité des cas, la réponse sera favorable à l'utilisation d'un standard, comme CORBA ou DCOM, et d'une implémentation commerciale. Par contre, le cas particulier traité à l'intérieur de ce mémoire concerne celui d'un système embarqué. Plusieurs facteurs contribuent à ce que les standards actuels ne soient pas propices pour de tels systèmes. Un système embarqué est généralement autonome et il possède peu de ressources matérielles : les unités de stockages sont très limitées et parfois même absentes sur de tels systèmes. Les ressources CPU ainsi que les ressources mémorielles sont aussi très limitées. Les systèmes plus légers doivent donc maximiser l'utilisation des ressources dont ils disposent. Le choix d'une technologie d'objets distribués devient, pour ce genre de système, une tâche plus complexe! Dans la majorité des cas, il est n'est pas requis ou il est même carrément inadéquat d'adhérer au formalisme de CORBA ou de DCOM car c'est standards sont lourds et ils n'ont pas été conçus, à la base, pour des systèmes de moindres tailles.

La performance de CORBA est pauvre et peu contrôlable, compte tenu de l'utilisation du protocole TCP. Même avec l'objectif de réaliser une version minimale et optimisée du standard CORBA, il est peu probable que son intégration soit possible, ou optimale, à l'intérieur d'une application embarquée. Pour être interopérable, l'ORB doit respecter le protocole IIOP, qui lui, est basé sur TCP. Pour certain système, TCP peut s'avérer être un handicap. Ce protocole ne tient pas compte des contraintes temps réel que nécessitent certains systèmes. Bien que CORBA ne contraigne pas les développeurs à utiliser un protocole de transport en particulier, c'est généralement le protocole TCP qui est de mise pour maximiser l'interopérabilité.

La technologie DCOM de Microsoft, quant à elle, n'est pas portable sur tous les systèmes embarqués. Les seuls systèmes pouvant utiliser DCOM sont ceux fonctionnant avec Windows comme système d'exploitation. Bien que WindowsCE repose sur une version allégée et orientée temps réel de Win32, il est trop contraignant de devoir s'en remettre uniquement à Microsoft pour tous les types d'applications à objets distribués. Puisque la technologie DCOM est intimement liée avec le système d'exploitation de Microsoft, cela la rend pratiquement inutilisable avec tout autre type d'environnement.

Comme les systèmes actuels ne répondent pas à tous les besoins d'un système embarqué, il devient pertinent de développer un système propriétaire offrant le support nécessaire à la création de systèmes à objets distribués. Le but de ce projet est donc de concevoir un protocole de communication réseau et une librairie de classes facilitant le développement et la programmation

de systèmes embarqués à objets distribués. Cette librairie de classes offrira les services nécessaires à la création de tels systèmes : elle sera simple, facile à utiliser, facile à maintenir, portable, efficace et fiable. Bien entendu, pour arriver à cela, certains compromis seront faits. Les limitations majeures sont les suivantes : aucun mécanisme de sécurité n'est présent dans la première version; seul le langage de programmation C++ est supporté; aucun compilateur IDL ne peut être utilisé puisque IDL n'est pas complètement respecté.

La réalisation de ce projet est constituée des étapes suivantes de recherche et de développement. Premièrement, une lecture et une analyse approfondie de différents ouvrages traitant du sujet ainsi que des technologies objets en général, ont été accomplies. Les articles et les ouvrages de références sont de bonne qualité : ils proviennent de revues comme Dr. Dobbs, C/C++ User Journal, Embedded Systems, etc. De plus, ces articles portent généralement sur des applications réellement fonctionnelles en industrie, ou en milieu universitaire. Deuxièmement, les besoins à combler ont été identifiés. Troisièmement, le protocole pour la distribution des objets a été conçu. Quatrièmement, les phases de design et de conception des classes qui implémentent le protocole ont été réalisées. Cinquièmement, le projet à été validé par une phase de tests appliqués aux différents concepts mis de l'avant.

Voici en résumé le contenu des autres chapitres. Le deuxième chapitre aborde la notion d'objets distribués en résumant les concepts et les approches de base entourant cette notion. Le troisième chapitre passe en revue le standard CORBA : il est très important de bien saisir l'architecture de CORBA, puisque le système actuel s'inspire grandement des concepts de ce standard. Le quatrième chapitre définit clairement tous les besoins auxquels doit répondre le système. Le cinquième chapitre explique le protocole de messagerie orienté objet, appelé DOOM. Par la suite, le sixième chapitre fait état de l'analyse et de la conception des classes, et pour terminer, le septième chapitre traite de l'étape de vérification du protocole et des classes.

Il est à noter que ce projet a un but réel : ses résultats seront utilisés à l'intérieur d'applications commerciales de Téléphonie Internet (VoIP). Ces produits sont développés par la compagnie Médiatrix Télécom, Inc. située à Sherbrooke.

2. Notion d'objets distribués

La conception et la programmation par objet constituent un sujet encore bien jeune, même si elles font partie de la culture informatique depuis déjà plus d'une décennie. Au fil des ans, différents concepts se sont greffés autour de l'objet et son utilisation s'est multipliée partout dans le monde. La notion retenant notre attention, à l'intérieur de ce mémoire, est celle du système composé d'objets distribués. Ce concept a vu le jour au cours des dernières années avec l'explosion des réseaux informatiques. De plus, la performance et la fiabilité des réseaux d'aujourd'hui permettent au concept d'objets distribués d'aborder efficacement le problème de la distribution des applications. Par le passé, cette approche s'avérait trop lourde lors de l'exécution, à cause de ses différents niveaux d'abstraction. Aujourd'hui, ce n'est plus le cas, le temps de développement et la capacité d'expansion d'une application constituée d'objets distribués étant bien supérieurs à celui et à celle d'une application développée à l'aide de méthodes traditionnelles.

Il est important de cerner correctement cette notion avant d'entreprendre une étude plus approfondie du sujet. La puissance de ce concept et de ses différentes tangentes doivent être bien comprises afin de saisir tout l'intérêt que cela suscite à l'intérieur de la communauté informatique. Afin de documenter cette notion, la présente section s'appuie en majeure partie sur la thèse du D^re^ Cherkaoui [2].

Plusieurs groupes de recherche de par le monde travaillent au développement de techniques concernant les diverses façons d'implémenter un système à objets distribués. Ces systèmes mettent généralement à profit les caractéristiques des applications distribuées qui sont orientées objets. Dans ce contexte, le marché informatique d'aujourd'hui assiste à l'apparition de nombreuses technologies aspirant à offrir des environnements de développement de systèmes d'objets distribués. Ces technologies se fixent comme objectif de dépasser les frontières jadis imposées tels que l'espace d'adressage, l'unité de traitement et même le langage de programmation.

La notion d'objets distribués désigne de manière générale un système fait d'objets hétérogènes accessibles à partir de nœuds distants. Par définition, un **objet** est une entité discrète possédant des frontières bien définies et une identité qui englobe un comportement et des états. Un **objet local** désigne un objet résidant sur le même nœud que celui de son client (l'application qui l'utilise). Un **objet distant** désigne un objet qui réside sur un nœud distant du nœud de son client. La Figure 2.1 illustre une application utilisant différents objets localisés sur des nœuds distants et interconnectés via un réseau.

Le niveau d'accès à ces objets peut varier d'une application à une autre et dépend de l'interface publique de l'objet en question. Dans certains cas, seule la fonctionnalité de l'objet est visée : la notion d'objets distribués se limite alors à un accès distant aux différents services offerts par l'objet. Dans d'autres cas, le souhait est d'interagir avec l'objet de façon homogène, depuis n'importe quel site du réseau. L'objet distribué doit alors se présenter comme un objet « partagé » entre les différents clients et il désigne exactement la même entité chez chacun d'eux. Cette notion est celle utilisée dans le cas des applications de travail coopératif.

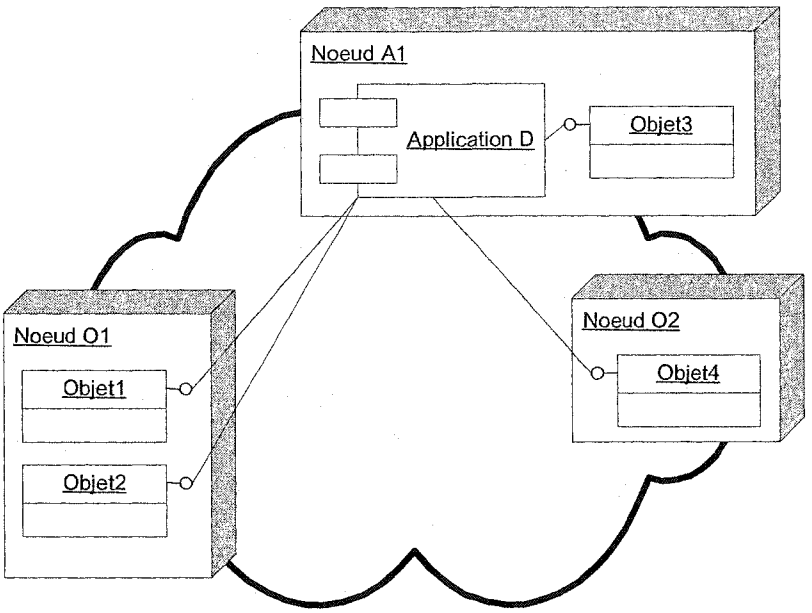

Figure 2.1 Système à objets distribués

La notion d'objet distribué choisie joue un rôle décisif dans l'architecture du système à objets distribués à concevoir. Dans la littérature, maints exemples d'architectures de système à objets distribués expriment des choix différents, relativement à des facteurs tels que le degré ou le type de distribution désirée, ainsi que le genre d'applications auxquelles ces systèmes serviront de base. Parmi les architectures les plus populaires, celles de l'architecture à invocations distantes, l'architecture à données centralisées et de l'architecture totalement distribuée seront citées. De manière assez grossière, presque tous les types de systèmes à objets distribués peuvent être rattachés à l'une ou l'autre de ces architectures.

De plus, les sections suivantes présentent quelques-unes des technologies existantes, tout en mettant en évidence, les points particuliers distinguant chacune d'entre elles.

2.1 Architecture à invocations distantes

Dans l'architecture à invocations distantes, les méthodes et les données associées à un objet sont localisées sur un même nœud, et l'accès aux services de l'objet se fait via des mécanismes d'appels distants. Ces mécanismes peuvent utiliser de simples appels de procédures distantes (RPCs) comme ils peuvent utiliser des outils plus performants tels que les ORBs (*Object Request Brokers*). Ceux-ci permettent de bénéficier des caractéristiques offertes par la programmation orientée objet tels l'encapsulation et le polymorphisme. La Figure 2.2 illustre l'architecture à invocations distantes.

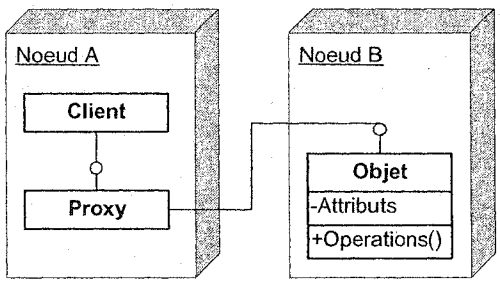

Figure 2.2 Architecture à invocations distantes

L'architecture à invocations distantes, quoique n'étant pas la seule, est souvent considérée comme étant l'architecture à objets distribués, puisque beaucoup de systèmes existants l'ont adoptée. Elle est surtout utilisée lorsqu'on désire accéder aux services d'objets dont la présence sur place serait inutile, coûteuse ou impossible. Dans l'architecture, les interactions mises en place sont des interactions Client/Serveur. Un exemple typique d'utilisation de cette architecture est celui d'une application permettant l'accès, depuis plusieurs sites, à des données distantes, à travers un objet gestionnaire de base de données. Un autre exemple est celui d'un système mettant en œuvre un ensemble d'objets accessibles à partir d'une plate-forme et d'un langage de programmation différent de l'environnement où ils ont été créés.

2.2 Architecture à données centralisées

Au niveau de l'architecture à données centralisées, les données associées à un objet sont situées sur un nœud central, alors que les méthodes associées à l'objet sont distribuées au sein des différents clients. La Figure 2.3 illustre cette architecture.

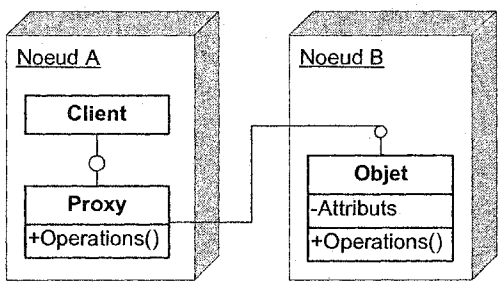

Figure 2.3 Architecture à données centralisées

Cette architecture a l'avantage de permettre un certain parallélisme au niveau de l'exécution des services d'un objet. Les méthodes de l'objet s'exécutent localement sur chacun des nœuds où une opération est appelée par un client. Cela a pour effet de diminuer la charge du nœud où se situe l'objet. Elle permet aussi d'assurer naturellement la consistance des données, puisqu'il n'y en a qu'une seule copie. Cependant, elle peut supposer la circulation d'une quantité importante de données à travers le réseau, du fait que l'exécution des services d'un objet doit se traduire irrémédiablement par un chargement des données, depuis le nœud central. D'autres architectures ont été dérivées de l'architecture à données centralisées afin de réduire le coût de chargement de données. Par exemple, l'architecture à relais de données, voir Figure 2.4, exploite plusieurs nœuds « relais » contenant des copies des données associées à un objet distribué, alors qu'un site

central s'occupe de garder les données réelles. Cette architecture requiert cependant une synchronisation parfaite des données sur chacun des nœuds.

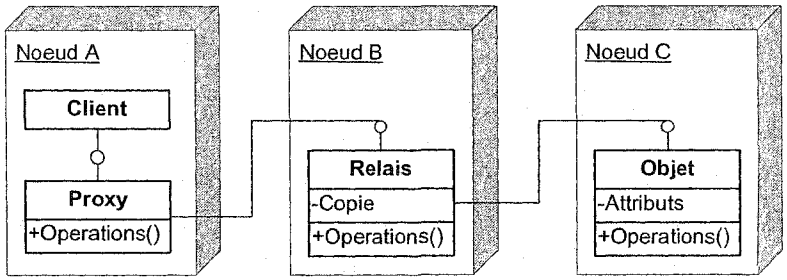

Figure 2.4 Architecture à relais de données

2.3 Architecture totalement distribuée

L'architecture totalement distribuée se distingue des autres architectures par le fait que les données, ainsi que les méthodes associées à un objet distribué, sont présentes au niveau de chaque instance d'objet, comme cela est montré dans la Figure 2.5. Chaque client d'un objet distribué a alors l'impression que l'objet distribué est local à sa machine. Les actions des usagers peuvent être réalisées de manière synchrone ou asynchrone.

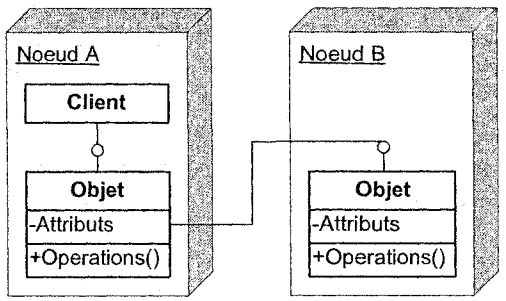

Figure 2.5 Architecture totalement distribuée

Il est certain que la mise en œuvre d'une telle architecture nécessite le développement de nombreux outils et mécanismes, afin d'assurer la coordination des actions dans les différentes copies d'un objet distribué, tout en sauvegardant la consistance de ses données. Néanmoins, elle présente l'avantage que toutes les ressources nécessaires à l'utilisation d'un objet sont locales à chacune de ses copies.

Plusieurs technologies existantes reposent sur la notion d'objets distribués. COM/DCOM, OLE et CORBA ne sont que quelques-unes des technologies qui ont vu le jour au cours des dernières années. Le chapitre suivant résume les concepts et l'architecture de la norme CORBA.

3. Notion de CORBA

Ce chapitre passe en revue l'architecture et les concepts d'une technologie de distribution d'objets, popularisée au niveau mondial durant les dernières années. Cette technologie se nomme *Common Object Request Broker Architecture* (CORBA) et elle a été développée par l'*Objet Management Group* (OMG) [13] [14] [15] [16] [17] [18] [19]. La présente section est réservée à la description de CORBA puisque plusieurs concepts du système DOES s'en inspirent et que l'architecture proposée par CORBA repose sur la notion d'invocation distante des objets : la compréhension générale de cette technologie devient donc un atout significatif.

L'OMG a été fondé en mai 1989 par huit compagnies. Le consortium, regroupant aujourd'hui plus de 800 membres, a comme rôle principal le développement du standard CORBA (*Common Object Request Broker Architecture*) par l'établissement de spécifications comme : CORBA/IIOP, *Object Services*, *Internet Facilities* et *Internet Domain*. En fait, l'OMG tente de résoudre les deux problèmes suivants avec l'aide de la norme CORBA : développement d'applications Client/Serveur basées sur des concepts orientés objets; intégration rapide de vieux systèmes avec de nouvelles applications.

L'approche objet est fondamentale pour l'OMG : tous les développements y sont orientés objets, ce qui favorise l'extension des fonctionnalités existantes et celles à venir. Cette structure a permis d'établir les lignes directrices communes et nécessaires au développement d'applications évoluant dans des environnements hétérogènes, constitués de différentes plates-formes hardwares et softwares. Les ouvrages de l'OMG fournissant en détail toutes les spécifications essentielles à la production de systèmes à objets distribués. Ces spécifications sont utilisées pour le développement et le déploiement d'applications distribuées destinées aux milieux de la finance, des télécoms, du commerce électronique, des systèmes temps réel et des systèmes de la santé.

Selon l'OMG, CORBA est la réponse au besoin d'interopérabilité créé par l'évolution, parfois trop rapide, des divers produits hardwares et softwares. Entre autre, CORBA permet à une application de communiquer avec une autre application, et ce, peu importe où elle se trouve, peu importe sur quel matériel et sur quel système d'exploitation elle fonctionne et peu importe avec quel langage elle a été écrite. CORBA 1.1 a été défini en 1991, mais c'est vraiment avec l'arrivée, en 1994, de la version 2.0 et du protocole *Internet Inter-ORB protocol* (IIOP) que la véritable promesse d'interopérabilité fut respectée.

À la base, l'OMG a développé l'*Object Management Architecture* (OMA) qui définit le plus haut niveau de spécifications. Les quatre éléments circonscrits par l'OMA et étant représentés à la Figure 3.1 sont :

1. **Object Request Broker (ORB) :** achemine les requêtes entre les différents éléments.
2. **CORBAservices :** fournit les services de base comme « Naming », « Persistance », « Event Notification ».
3. **CORBAdomains :** répond à certains domaines d'applications comme les télécommunications, la finance et la fabrication.

4. **CORBAfacilities :** couvre divers champs d'applications génériques par ses fonctions haut niveau.

Ces éléments seront définis plus en détail dans les prochaines sections.

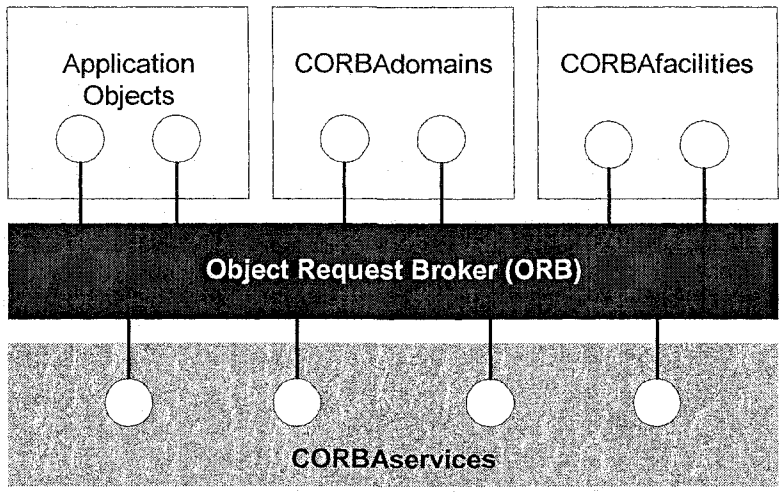

Figure 3.1 Objet Management Architecture

Le concept de composante logiciel est un rêve poursuivi sous plusieurs formes depuis très longtemps, et ce, par plusieurs groupes différents. Selon l'OMG, CORBA est le premier succès commercial d'une structure de base où les objets logiciels peuvent être branchés de façon transparente. Cette réussite résulte de caractéristiques inhérentes à CORBA tels qu'un accès uniforme aux services, une méthode de découvert uniforme des ressources et des objets, une gestion uniforme des erreurs, une politique uniforme de gestion de la sécurité. Les prochains paragraphes présentent un résumé des trois concepts importants de CORBA.

Modèle objet

CORBA repose solidement sur les concepts de l'orienté objet. Ces concepts de base sont ceux des objets, des classes, de l'encapsulation, de l'héritage et du polymorphisme. CORBA définit un modèle objet pour lequel les frontières et la signification des termes de base n'ont aucune équivoque. Le modèle est basé sur une approche complètement objet dans laquelle un client envoie un message à un objet serveur. Ce message identifie l'objet ainsi qu'un ou plusieurs paramètres. Le premier paramètre définit l'opération à exécuter qui sera appelée par l'ORB. Le modèle objet mis de l'avant par CORBA est constitué des concepts suivants :

- **Objet :** entité abstraite fournissant des services à un client. Chaque objet est identifié par une référence.
- **Requête :** action créée par un client et dirigée vers un objet cible. Cette action contient les informations sur l'opération à exécuter et un ou plusieurs paramètres.
- **Création et destruction d'objet :** requêtes par lesquelles un objet peut être créé et détruit.

- **Type :** entité définie avec des valeurs de base.
- **Interfaces :** ensemble des opérations propres à un objet dont un client peut faire la requête.
- **Opérations :** entité identifiable définissant ce qu'un client peut demander à un objet.

Environnement informatique ouvert et distribué

CORBA se veut un environnement fournissant un accès unifié à la communication multidirectionnelle entre les objets distribués d'un système, indépendamment de leur localisation réelle sur le réseau. Le modèle Client/Serveur est illustré par la Figure 3.2. À l'intérieur de ce modèle, une requête de service est faite à partir d'une composante logicielle vers une autre composante visible sur le réseau. CORBA ajoute une nouvelle dimension à ce modèle, en insérant un *broker* entre le client et le serveur dans le but de réduire la complexité de l'interaction entre les deux entités. Les rôles majeurs du *broker* sont de fournir des services communs de messagerie et de communication entre le client et le serveur, un service de répertoire, un service de description des métas données, un service pour la transparence de la localisation et un service de sécurité. De plus, il isole l'application des configurations particulières du matériel, du système d'exploitation, des protocoles réseaux et des langages de programmation.

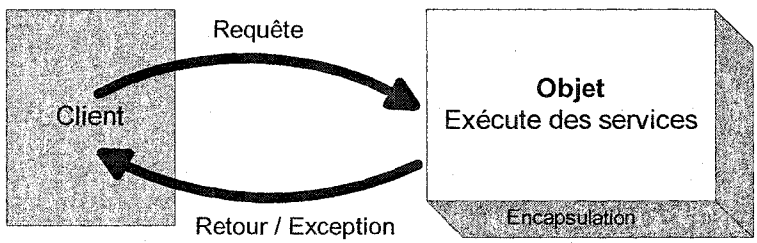

Figure 3.2 Modèle Client/Serveur

La communication dans le modèle CORBA est basée sur des liens point à point supportant les messages synchrones et sur une version limitée des messages asynchrones. L'interopérabilité entre les différentes implantations d'ORB est assurée par le protocole *General Inter-ORB Protocol* (GIOP). Quant aux communications via un réseau TCP/IP, elles s'appuient sur le protocole nommé *l'Internet Inter-ORB Protocol* (IIOP).

Intégration et réutilisation de composantes

Le concept clef pour l'intégration d'application repose sur la spécification d'interfaces à l'aide de l'*Interface Definition Language* (IDL). Définie de cette façon, l'application devient indépendante de sa localisation, du type de plate-forme, du protocole réseau et du langage de programmation utilisé. Le langage IDL ne peut cependant pas assurer, à lui seul, une réutilisation et une intégration facile des composantes. Sans de bons outils et de bonnes techniques d'intégration, la réutilisation est difficile. Par exemple, sans un *broker*, les interfaces doivent être redéfinies pour chacune des nouvelles interactions entre les composantes. Par contre, avec un *broker*, une interface n'est définie qu'une seule fois et chacune des interactions suivantes est gérée par l'ORB. Sans un ORB, le nombre de relations entre les interfaces croît d'une façon exponentielle,

tandis qu'avec l'ORB, ce nombre augmente de manière linéaire, comme l'illustre la Figure 3.3. Lors de l'extension d'un système complexe, les avantages du *broker* sont clairs.

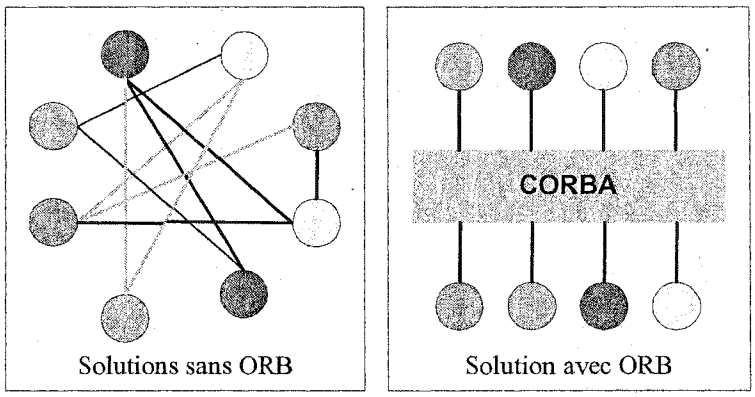

Figure 3.3 Intégration avec et sans ORB

3.1 Interface Definition Language

Toutes les interfaces dans CORBA sont spécifiées à l'aide du langage *Interface Description Language* (IDL). IDL est l'un des standards les plus importants de CORBA. Il est comparable au langage Backus-Naur Form. En fait, IDL définit une notation universelle pour les interfaces logiciels. OMG IDL a été standardisé en 1991 et jusqu'à ce jour, les spécifications n'ont pratiquement pas changées : cette grande stabilité du standard IDL est plus que souhaitable, car IDL constitue la base de toutes les spécifications adoptées par l'OGM. L'un des aspects importants d'IDL repose sur le fait qu'il soit totalement indépendant du langages de programmation. Par exemple, un client qui utilise une interface IDL n'aura jamais besoin d'être modifié ou recompilé, et ce, même si l'implémentation de l'interface est réalisée avec différents langages de programmation, s'exécutant sur différentes plates-formes. Avec un seul fichier IDL définissant une interface quelconque, il est possible de produire des fichiers de code source pour des langages comme Ada95, SmallTalk, C, C++ et ainsi qu'à peu près tous les autres langages commerciaux de moyenne importance. Il est à noter qu'IDL obéit aux mêmes règles lexicales que le C++. Cependant, certains mots clefs ont été ajoutés pour supporter les concepts de distribution.

Il faut cependant faire très attention, car IDL n'est utilisé que pour spécifier la forme d'une interface. L'implémentation de l'objet qui réalise cette interface est faite de façon indépendante. Il n'est pas approprié de représenter avec IDL les caractéristiques de l'implémentation comme la dynamique, l'héritage ou les relations. Cette approche permet d'obtenir un séparation stricte entre le client et l'objet, ce qui accroît encore plus la portabilité et la réutilisation du code. IDL utilise le concept de type : il y a les types d'objets et les types de données. Chaque nouvelle interface définit un nouveau type d'objets qui est unique de par ses attributs, ses opérations et ses exceptions. Quant aux types de données utilisés, ce sont des types simples comme *integer*, *boolean*, *double*, etc. Des types de données complexes peuvent aussi être définis comme des structures ou des séquences. Le TABLEAU 3.1 résume les divers concepts de la notation IDL proposés par l'OMG.

TABLEAU 3.1 IDL, notations et concepts

Concepts	Description
Module	Un module est utilisé pour créer un *name space* servant à éviter les collisions de noms entre deux domaines. Les modules IDL se comparent aux *packages* d'UML. Pour accéder à un nom faisant partie d'un autre module, l'opérateur de portée, défini par le symbole « :: », doit être utilisé.
Interface	Chaque interface définit un nouveau type d'objets avec ses opérations et ses attributs publiques. En fait, l'interface sert de délimiteur entre l'implémentation des services et de leurs clients.
Forward	Sert à déclarer le nom d'une interface avant que le corps de celle-ci soit défini.
Constante	Définition de valeurs constantes (*integer*, *character*, *boolean*, etc.).
Type	La déclaration de nouveaux types à partir de types intrinsèques d'IDL est permise. La fonction de validation sévère des types (*strong type checking*) peut ainsi avoir lieu en tout temps au niveau du compilateur IDL. C'est l'équivalent du mot clef *typedef* en C++.
Type *Any*	Le type *Any* est le seul type qui n'est pas validé à la compilation. Ce type peut prendre n'importe quelle forme au moment de l'exécution. Comme le type *Any* se décrit lui-même, des mécanismes d'extraction d'information peuvent être utilisés au moment de l'exécution pour déterminer de quel type est la variable.
Séquence	Une séquence est un tableau de longueur variable. En mémoire, la séquence est enregistrée dans un tableau continu d'éléments du type indiqué.
Attribut	Tous les attributs définis à l'intérieur d'une interface IDL sont publics. Les attributs privés ne doivent pas apparaître à l'intérieur d'une interface. Contrairement au C++, un attribut peut être défini en lecture seulement avec le mot clef *readonly*. Par défaut, tout attribut est accessible en lecture et en écriture.
Exception	Une exception définit la valeur passée par l'interface dans un cas d'erreur.
Opération	La signature d'une opération à l'intérieur d'une interface définit une façon possible d'accéder à l'objet. Par défaut, les opérations IDL sont exécutées de façon synchrone, mais elles peuvent être exécutées de façon asynchrone grâce au mot clef *oneway*.

3.2 Object Request Broker

Le rôle principal de l'*Object Request Broker* (ORB) est l'unification de l'accès aux services via un mécanisme orienté objet d'appel distant d'opérations. Il représente l'infrastructure de base utilisée par tous les objets communiquant entre eux. L'ORB procure une méthode de communication entre les différents modules d'un système localisés sur un même nœud, ou sur des nœuds distants du réseau. Par ce mécanisme, le client invoque les opérations d'un objet via

les services de l'ORB de façon totalement transparente. Comme le montre la Figure 3.4, l'ORB agit comme un intermédiaire entre le client et l'objet. Il procure ainsi une abstraction des mécanismes de communication. Il est responsable d'intercepter l'appel du client, de trouver l'objet correspondant, de *marshaler* les paramètres de la méthode, d'invoquer la méthode et de retourner le résultat. En plus de séparer les différentes entités d'un système, le *broker* les isole des différentes mécaniques sous-jacentes comme les piles de protocoles, les systèmes d'opération, les plates-formes matérielles, le langage de programmation, etc. Cette approche procure une grande flexibilité et simplifie grandement le modèle de distribution des objets. Le client n'a pas besoin de savoir où se trouve l'objet, ni de connaître les aspects ne faisant pas partie de l'interface de l'objet serveur. De cette façon, l'ORB fournit une interopérabilité certaine entre des applications fonctionnant dans un environnement distribué et hétérogène[2].

La Figure 3.5 représente la manière dont une application utilise les interfaces CORBA. Il est à remarquer que seuls le client et l'objet serveur se situent à l'extérieur de l'ORB, tous les autres éléments faisant partie de ce dernier. La couche du bas représente le cœur de CORBA. C'est à l'intérieur de l'ORB *Core* que réside l'implémentation des spécifications CORBA. La couche du centre représente les interfaces publiques qui sont exposées à l'extérieur de l'ORB et utilisées par les clients et par les objets serveurs de l'application. Ces différentes interfaces sont regroupées en trois catégories. La première catégorie correspond aux interfaces définies par CORBA et implémentées de façon standard : elle regroupe les *ORB Interface*, les *Dynamic Invocation Interface* et les *Dynamic Skeleton Interface.*

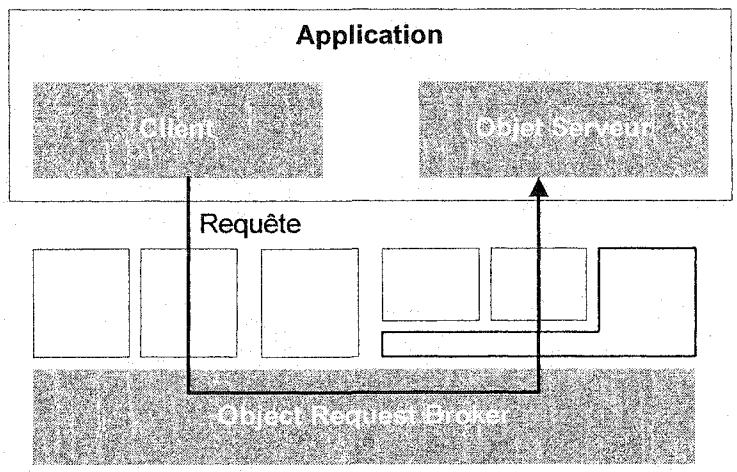

Figure 3.4 Infrastructure de communication

La seconde catégorie correspond aux adaptateurs d'objets (*Object Adapter*, OA). En plus d'être une interface standard définie par CORBA, l'OA peut aussi être défini par le créateur d'un ORB. Le seul OA défini à ce jour par CORBA est le *Basic Object Adapter* (BOA). Cet OA fournit les services essentiels de base. Les implémentations des serveurs accèdent à la plupart des services ORB via un OA. Ces services incluent : la génération et l'interprétation des références d'objets; l'authentification des clients; l'activation et la désactivation d'objets; l'invocation de méthodes. La

[2] Il faut noter que pour assurer l'interopérabilité, les objets client et serveur doivent respecter le même protocole de communication. L'avantage de CORBA est que, contrairement à l'approche typique du développement de systèmes Client/Serveur, le protocole entre les objets est défini via leurs interfaces décrites avec IDL.

troisième catégorie correspond aux interfaces définies par IDL (*Stubs* et *Static Skeleton*) : celles-ci peuvent être générées par tous les usagers d'un ORB.

L'architecture CORBA offre à un client trois façons d'interagir avec l'ORB :

1. En utilisant les interfaces statiques (*IDL-Stubs*) générées automatiquement à l'aide de la spécification IDL de l'interface au moment de la compilation : cette façon de faire est la méthode la plus naturelle pour l'invocation d'opérations. L'opération d'un objet CORBA est appelée avec la même syntaxe que celle utilisée pour un objet local.

2. En utilisant les interfaces dynamiques (*Dynamic Invocation Interface*) représentant un moyen de spécifier les requêtes au moment de l'exécution. Cette méthode est utilisée lorsque le client ne peut pas connaître le type de l'interface à utiliser au moment de la compilation. Le client peut découvrir quelle est la bonne interface à utiliser, grâce au dépôt d'interface (*Interface Repository*) auquel il peut accéder via un éventail d'opérations standards.

3. En utilisant directement les services ORB via l'*ORB Interface*. Par exemple, le client peut obtenir la référence à un objet, grâce à un service standard offert par l'ORB.

Du côté de l'objet serveur, l'implémentation reçoit les requêtes soit par la voie des *Static IDL Skeleton* générés par la même description IDL inhérente à l'interface, soit par la voie des *Dynamic Skeleton Interface*. Cette implémentation de l'objet serveur peut faire des appels à l'*ORB Interface* ou à l'OA, lors de l'exécution de la requête.

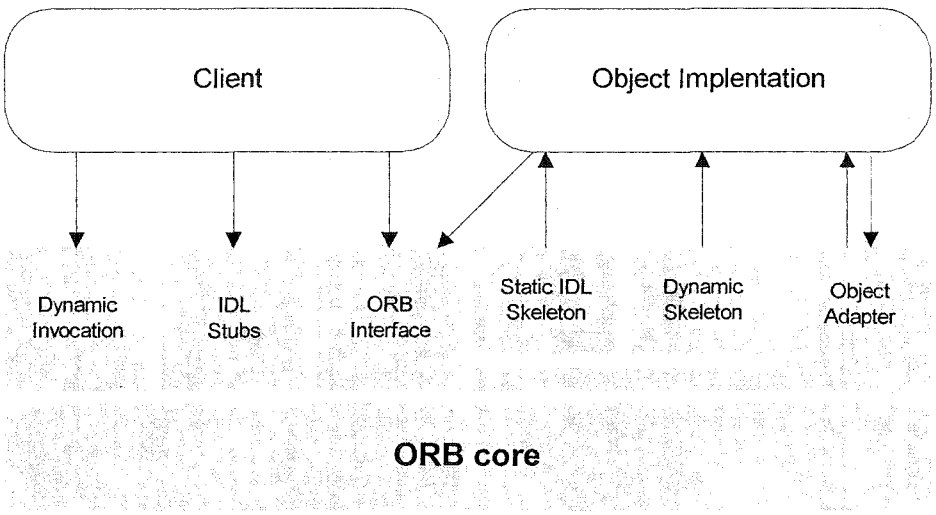

Figure 3.5 Les interfaces d'un *Object Request Broker*

3.3 ORB interopérabilité

Pour un système où les objets sont entièrement distribués sur le réseau, l'interopérabilité est un problème important qu'il ne faut surtout pas négliger. Dans le contexte d'une application CORBA, l'interopérabilité est indépendante des interfaces définies par les clients. Les problèmes

de communication entre ORB sont traités à un niveau inférieur à l'application et ils n'ont aucune relation avec celle-ci. Par exemple, à l'intérieur d'un même système, un objet client peut accéder à un objet serveur reposant sur un ORB différent du sien. Pour que le client puisse accéder au serveur de façon transparente, les fabricants des différents ORB doivent s'entendre sur un protocole de communication commun.

CORBA propose plusieurs solutions et différentes architectures pour résoudre les problèmes d'interopérabilité. L'une de ces architectures est le *General Inter-ORB Protocol* (GIOP) [15] à l'intérieur duquel se retrouvent les spécifications du protocole *Internet Inter-ORB Protocol* (IIOP). Ce protocole repose sur les technologies standards de l'Internet comme TCP/IP. A ce jour, IIOP est le seul protocole à avoir été standardisé par CORBA.

Il est intéressant de noter que CORBA peut être vu comme un gestionnaire intelligent de couches réseaux de bas niveau. Une fois que la connexion est établie, il se retire pour laisser les objets communiquer directement entre eux. Cette méthode permet d'atteindre des performances appréciables.

3.4 CORBAservices

Les CORBAservices [19] représentent un éventail d'interfaces fournissant certains services de base à des applications plus générales. Les services les plus utilisés varient beaucoup selon les domaines d'application, mais les deux services les plus répandus sont : Naming et Events. Habituellement, les ORB commerciaux sont vendus avec plusieurs services, ce qui évite souvent aux développeurs de réinventer la roue. Il est cependant surprenant de noter que parmi les quinze services définis par CORBA, très peu sont supportés commercialement. Dans certains cas, le développement de services complémentaires ne peut pas être évité. Le choix d'un ORB, en fonction des services qu'il offre et des besoins de l'application, est donc une tâche importante. Les différents services de CORBA peuvent être regroupés en quatre grandes catégories. Le TABLEAU 3.2 présente un résumé des diverses catégories et de leurs services.

TABLEAU 3.2 Catégorisation des CORBAservices

Catégories	Services	Description
Infrastructure	Security, Time	Regroupe les services fortement couplés avec les mécanismes de l'ORB.
Information Management	Property, Relationship, Query, Externalization, Persistent, Collection.	Regroupe les services de base utilisés pour la manipulation et l'extraction de données.
Task Management	Events, Concurrency, Transaction,	Regroupe les services permettant la gestion des événements et des transactions.
System Management	Naming, Life Cycle, Licensing, Trader	Regroupe les services de base permettant la gestion des méta-données, des licences et de la durée de vie des objets.

3.5 CORBAfacilities et CORBAdomains

CORBA définit aussi un éventail de services haut niveau, nommés CORBAfacilities, reposant sur les services de base offerts par les CORBAservices. Certains domaines d'application sont aussi standardisés par l'OMG à l'intérieur des CORBAdomains. Ainsi, tout ce qui ne peut pas être standardisé, à l'intérieur de l'architecture de CORBA, fait partie soit des CORBAfacilities, soit des CORBAdomains.

Les différents CORBAdomains sont développés avec une perspective verticale : le développement se fait en considérant uniquement les besoins d'interopérabilité du domaine. Les différents domaines regroupent, par exemple : les services financiers, les services de la santé, les services de télécommunication ainsi que plusieurs autres.

Les différents CORBAfacilities sont développés avec une perspective horizontale : les services partagés par les différents domaines d'application sont réalisés à l'intérieur des CORBAfacilities. Une attention particulière est portée aux problèmes d'interopérabilité, car ces services peuvent être utilisés à l'intérieur de différents contextes. Par exemple, les *Compound Documents* et les fonctions de gestion de système sont des services utilisés par plusieurs domaines d'application.

3.6 Produits logiciels

Il est important de spécifier que CORBA ne constitue pas un produit logiciel. CORBA est en fait un ensemble de spécifications et de modèles définis par un groupe de standardisation accrédité par la communauté informatique. Une fois les standards établis, certaines compagnies privées se chargent de développer et de supporter des applications CORBA commerciales communément appelées ORB. Il est à noter qu'aujourd'hui, un grand nombre d'entreprises respectent les spécifications de l'OMG et tentent de vendre leurs produits. À cet effet, plus de 100 compagnies

sont répertoriées à l'intérieur du document *CORBA Buyers Guide* publié par l'OMG en 1997. Les prochains paragraphes présentent quelques-uns des nombreux ORB commerciaux disponibles sur le marché :[3]

VisiBroker

VisiBroker est offert par la compagnie Inprise, autrefois connue sous le nom de Visigenic Software Inc. Ce produit est facile d'utilisation et il est aussi très flexible, ce qui lui permet de pour répondre aux besoins toujours grandissants des nouvelles applications. Inprise présente entre autre un ORB pour Java. Les particularités de cet ORB sont les suivantes : d'abord, il est le premier à avoir été implémenté complètement en Java, et de plus il est le premier à respecter CORBA 2.0. Plusieurs autres produits commerciaux sont offerts, mais les seuls CORBAservices disponibles et supportés par VisiBroker sont Naming et Events.

MICO

L'acronyme MICO signifie *MICO Is CORBA* [21]. Ce produit fait partie d'un développement *open source*, ce qui signifie que le code source est disponible sur le Web et que toute personne intéressée peut contribuer à l'avancement du projet. Le but du projet est de fournir une implémentation de la version 2.2 des spécifications de CORBA et cela, tout à fait gratuitement. Les points forts de MICO sont les suivants : l'implémentation est neuve et elle n'utilise que les appels UNIX standards; il n'y a que les besoins obligatoires qui sont réalisés; le langage utilisé est le C++; seuls les outils standards et utiles sont développés; l'architecture interne est propre et bien documentée. Pour obtenir plus de détails sur MICO, il faut se référer à la page suivante : http://www.icsi.berkeley.edu/_mico/.

ORBIX de IONA Technology Limited

IONA Technologies a été la première compagnie à offrir un produit commercial avec un compilateur IDL. La première version de Orbix fut livrée en 1993 et ce dernier est rapidement devenu le leader. Orbix supporte à ce jour les langages C++, Ada95 et Java sur plus de dix plates-formes différentes, dont Windows NT. Les premières versions de Orbix ont été implémentées au-dessus de ONC-RPC. Aujourd'hui, Orbix supporte le protocole IIOP, ce qui augmente son niveau d'interopérabilité.

[3] Une liste exhaustive d'ORB commerciaux se retrouve à l'intérieur du document [14].

4. Besoins et spécifications

Le présent chapitre définit les besoins et les spécifications devant être comblés par le système de Distribution d'Objets pour Système Embarqué (DOES). Le but premier de ce chapitre est de dresser un profil global de ce système DOES. Ce profil sert à saisir, éclaircir et faciliter la compréhension générale du système et de ses applications potentielles. L'élaboration des spécifications repose sur trois sections ayant chacune un rôle bien spécifique : la première section décrit et précise le modèle contextuel du système; la seconde section décrit les différents cas d'usages du système; la troisième section décrit les caractéristiques supplémentaires qui sont applicables de façon générale au système.

4.1 Modèle contextuel

Le modèle illustré par la Figure 4.1 expose les principaux objets du système DOES. Ces objets représentent les entités existant dans l'environnement de ce système. Ce modèle est basé sur le pattern *Proxy* [29]. Il s'appuie aussi sur le modèle de l'architecture CORBA présenté à l'intérieur de la troisième section et ainsi qu'à l'intérieur du modèle COM de Microsoft [25].

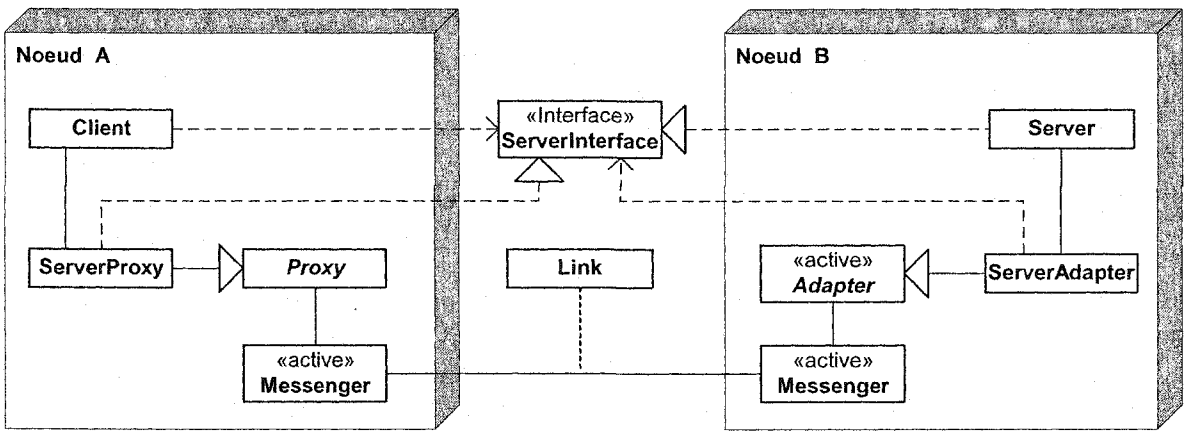

Figure 4.1 Modèle contextuel

Ce modèle est composé de deux nœuds interconnectés. Ces nœuds peuvent être vus comme des systèmes matériels indépendants et distribués dans l'espace d'un réseau. Ils sont les hôtes de composantes logiciels et ils servent de plate-forme d'exécution aux différents objets. Dans le contexte présent, ils sont reliés par un réseau physique et ils communiquent entre eux grâce au protocole IP. La Figure 4.1 présente deux nœuds d'un système distribué : le Nœud A accueille un objet client et le Nœud B accueille un objet serveur. Il est à noter que chacun des nœuds d'un système distribué peut contenir des objets clients, des objets serveurs ou des objets de n'importe quels autres types, mais le schéma ne représente pas ce dernier cas, afin de ne pas complexifier le modèle inutilement.

Le client est un objet qui émet des requêtes vers un autre objet, le serveur. La requête du client contient les informations sur l'opération à exécuter, ainsi que ses paramètres. Le client a une dépendance directe envers l'interface de l'objet serveur qu'il utilise. Cependant, lorsque l'objet

serveur est localisé sur un nœud distant, l'objet client ne peut pas posséder un lien direct avec l'instance du serveur : ce dernier possède alors un lien sur un objet Proxy d'intermédiaire. Le serveur est une entité abstraite fournissant des services à un client et devant répondre à toutes les requêtes qui respectent son interface publique. Ce serveur peut être localisé sur le même nœud que le client, ou sur un nœud distant. Dans le cas ou le serveur est créé sur un nœud distant, un Adapter gouvernera ce serveur tout au long du processus.

La classe *Proxy* est une classe abstraite responsable du *marshaling* et possédant une association avec le *Messenger*. Elle est responsable d'acheminer les requêtes d'un client vers le *Messenger* du processus, afin qu'elles soient transférées vers l'objet serveur. Cependant, cette classe ne peut pas être utilisée directement par un client, puisqu'elle ne contient aucune connaissance de l'interface réalisée par l'objet serveur. Les actions de marshaling doivent donc être réalisées par une classe dérivée de la classe *Proxy* qui héritera de la même interface que l'objet serveur. C'est donc dire qu'au moment de l'exécution, le client possède en fait un lien sur un objet *Proxy* spécialisé réalisant une interface et servant uniquement d'intermédiaire entre lui et le serveur.

L'*Adapter* se retrouve sur le même nœud que le serveur. Il sert principalement à recréer l'environnement et l'espace mémoire d'un client distant. La création et la destruction des objets serveurs passent aussi par lui. Lorsqu'une requête en provenance d'un client arrive via le réseau, les paramètres doivent être extraits et replacés en mémoire de façon à être utilisés par l'objet serveur. Cette action est communément appelée le *unmarshaling*. Comme le *Proxy*, l'*Adapter* se présente, lui aussi, sous la forme d'une classe abstraite de base. Cette classe possède une association bidirectionnelle avec le *Messenger* par lequel sont échangés les messages indépendamment qu'ils soient en provenance ou en direction d'un nœud distant. Les actions de *unmarshaling* doivent être réalisées par une classe dérivée de la classe *Adapter* possédant les connaissances nécessaires à l'extraction des paramètres de chaque opération. Une fois les paramètres extraits, l'*Adapter* appelle la méthode appropriée de l'objet serveur lui étant associée.

La classe *Messenger* se présente sous le format d'un singleton [29]. Cela signifie qu'une seule instance de la classe peut exister sur un même nœud. Toutes les requêtes de service dédiées à un objet serveur distant doivent passer par cette entité unique. Le *Messenger* est responsable de la gestion du protocole de communication inter-nœud.

En résumé, une fois qu'un *Proxy* a sérialisé les paramètres d'une opération, il fait appel aux services du *Messenger* pour transférer le message vers le nœud du réseau où réside l'objet serveur. A son arrivée, le message est validé par le *Messenger* et si sa destination est correcte, il est redirigé vers l'*Adapter* adéquat qui se chargera de faire l'appel de la méthode correspondante au serveur. Il est à noter que pour qu'un transfert du message ait lieu, l'*Adapter* doit s'être préalablement enregistré au Messenger et que ce dernier, doit lui-même être associé à un *Socket*.

Le système DOES sert donc à rendre le mécanisme d'interconnexion virtuel entre le client et le serveur. Ce mécanisme, se voulant aussi transparent que possible, possède des avantages indéniables pour le client. En effet, l'appel d'opérations d'un objet serveur localisé sur un nœud distant par rapport à son client se fait de façon traditionnelle. C'est donc dire, que pour le client, il n'y a aucune différence quant à l'utilisation de l'interface du serveur, qu'il soit local ou distant : tout le mécanisme de transfert des requêtes est pris en charge par DOES.

4.2 Cas d'usages

Le problème majeur dans l'élaboration des besoins du système est d'arriver à développer un modèle de celui-ci, pour mieux saisir ses limites et son contexte d'application. Cette section tente donc de saisir les frontières du système et ses besoins fonctionnels, en utilisant la méthode des *Use Cases* proposée par UML [26] [27] [28]. Le cœur de cette analyse porte sur les entités gravitant autour du système (les acteurs) ainsi que sur les actions pouvant être exécutées par le système (les cas d'usages). Les caractéristiques supplémentaires du système sont traitées à l'intérieur de la section 4.3.

La Figure 4.2 présente le modèle des *Use Cases* pour le système DOES. À l'intérieur des prochains paragraphes, les noms des acteurs et leurs rôles respectifs seront décrits. De plus, leurs besoins et leurs responsabilités y seront aussi définis. Il est important de bien comprendre la nature même de chacun des acteurs parce que cela permet, entre autre, d'être en mesure d'évaluer précisément ce qu'ils représentent pour le système. Suite à cette évaluation, chacun des Use *Cases* est passé en revue : ils sont définis en fonction de leurs responsabilités et de leurs actions potentielles. Cette description permet de connaître avec précision, ce que doit faire le système lorsqu'il interagit avec ses acteurs, en fonction de chacun de ses cas d'usages.

4.2.1 Description des acteurs

Un **Objet** est une entité discrète qui possède une frontière bien définie et une identité propre englobant ses différents états et son comportement. Cet acteur représente toutes les classes d'objets logiciels en relation avec le système DOES. Ses responsabilités sont indéterminées et illimitées. Tout objet utilise DOES pour s'isoler du système d'exploitation et pour gérer ses allocations de mémoire et ses flots d'exécutions.

Le **Client** est un élément qui requiert les services d'un autre élément. Cet acteur peut être défini comme un objet jouant un rôle de client face à un autre objet, et ce, en invoquant certains de ses services. Le client a la responsabilité de s'associer, de créer ou de détruire les objets serveurs qu'il utilise. Une fois le lien établi entre lui et le serveur, le client réclame certains services du serveur par l'intermédiaire du mécanisme d'appel d'opération du système DOES.

Le **Serveur** est un élément qui fournit des services pouvant être invoqués par d'autres objets. Les services du serveur sont indéterminés et illimités. Face au système DOES, sa seule responsabilité est de répondre aux appels d'opération en provenance d'un client, de façon synchrone ou asynchrone selon le cas. Comme le diagramme le démontre, l'entité serveur est représentée par deux acteurs différents : **Serveur Local** et **Serveur Distant**. Le Serveur Local se retrouve et s'exécute sur le même nœud que le client qui l'utilise, tandis que le Serveur Distant se retrouve sur un nœud différent du nœud sur lequel s'exécute son client.

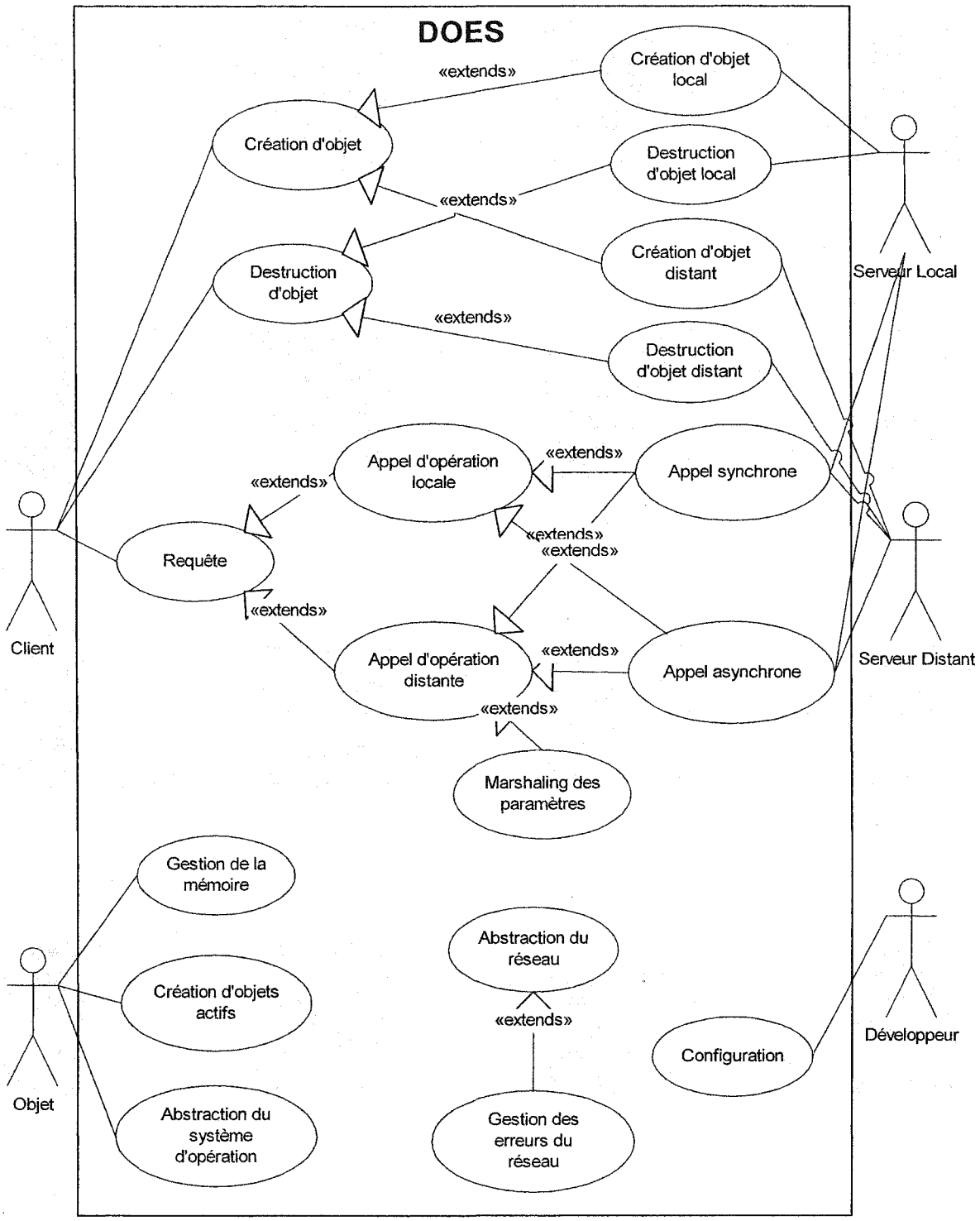

Figure 4.2 Diagramme des cas d'usages

Le **Développeur** représente l'être humain qui programme et intègre les différents objets gravitant autour du système DOES et formant l'application finale. Le développeur a la responsabilité de configurer proprement le système DOES en utilisant les différents paramètres disponibles à cet effet.

4.2.2 Description des cas d'usages

La **création d'objet** est utilisée par le client lors de la toute première étape. Avant de pouvoir faire appel aux services d'un objet quelconque, cet objet serveur doit être créé. C'est le client qui fait cette requête au système. L'objet est créé normalement avec l'interface standard du C++ (new), à la seule différence que le client peut spécifier le nœud cible qui sera l'hôte de l'instance. Ce nœud peut être le même que celui où le client réside, ou tout autre nœud qui est physiquement relié au nœud du client. Lorsque la création est **locale**, l'objet serveur réside et s'exécute sur le même nœud que le client. Il est important de spécifier qu'un objet local est défini comme résidant à l'intérieur du même processus que le client. Dans ce cas, DOES ne s'interpose pas entre les deux objets : le lien est direct, ce qui permet d'avoir une efficacité maximale lors des échanges. De plus, comme la mémoire est partagée entre les objets, qu'ils soient actifs ou non, les échanges de structures ou de paramètres de grandes dimensions sont très efficaces, puisque ces données n'ont pas besoin d'être copiées. Quant à la création d'objet **distant**, elle implique que l'objet serveur soit créé à l'extérieur du processus où réside le client ordonnant cette action. La cible peut être soit un nœud distant, soit un processus distinct. Les paramètres de création comprendront essentiellement l'adresse exacte du nœud. Cette adresse peut être obtenue statiquement ou dynamiquement par le client et elle doit être fournie au système DOES lors d'une requête de création. Cette recherche d'adresse n'est pas supportée par le *Messenger* : elle doit donc être entièrement assumée par le client.

L'objet client possédant un lien de composition ou un simple lien de possession sur un autre objet, qu'il soit local ou distant, est responsable de sa **destruction**. En effet, une fois la vie utile de l'objet terminée, le processus de destruction doit être engagé par l'objet maître. La syntaxe utilisée est la même que la syntaxe normale du C++ (delete), c'est à dire que lorsque l'objet est **local**, le mécanisme de destruction appelle le destructeur de l'objet et libère la mémoire associée, et que par contre, lorsque l'objet se situe sur un nœud **distant**, un message ordonnant la destruction doit transiter par le *Messenger*. De plus, une fois que la destruction de l'objet serveur est effectuée, le *Proxy* est aussi détruit et la mémoire est libérée.

Un client peut spécifier et envoyer un stimulus à une instance via un mécanisme d'**appel**. Cela se traduit par l'appel d'une opération ou l'envoi d'un signal à un objet serveur. Ce mécanisme permet à un client de faire appel à n'importe quelle opération publiée par l'interface de l'objet serveur. Le système supporte des appels d'opérations **locaux** ou **distants**. Lorsque l'objet est distant, le mécanisme veille à ce que la requête transite vers le nœud associé. Dans le cas d'un appel local, les mécanismes standards d'appel d'opération sont utilisés, ce qui permet une plus grande efficacité en évitant toutes les « indirections » inutiles.

L'appel d'une opération peut être **synchrone** ou **asynchrone**. Tous les compilateurs C++ supportent, par défaut, les appels d'opération synchrones : en fait, ce mode est le seul qui est supporté de façon standard. Le système DOES doit supporter les appels synchrones et asynchrones d'opération sur des objets locaux ou distants. Le retour d'une opération asynchrone est, par définition, instantané, tandis que le temps d'exécution d'une opération synchrone est indéfini. Lorsque le serveur est local, les appels synchrones reposent sur les mécanismes standards du C++. Par contre, lorsque le serveur est distant, le système doit s'assurer que le client est bloqué pendant toute la durée du traitement de l'opération : une opération asynchrone requiert donc un certain nombre de services évolués, puisque aucun mécanisme standard du C++ ne

supporte ce type d'appel. Ces services doivent permettrent à un objet d'être actif et réceptif aux requêtes en provenance de l'externe. Le client, qu'il soit local ou distant, et qui fait appel à une opération de type asynchrone n'est pas bloqué pendant le traitement de celle-ci.

Le mécanisme de ***marshaling*** entre en fonction lors de l'appel d'une opération impliquant le passage d'un ou de plusieurs paramètres. Lorsque l'objet serveur est local, les paramètres de l'opération peuvent être passés par référence ou par valeur, mais comme la mémoire est partagée à l'intérieur d'un même processus (les *threads* partagent aussi le même espace de mémoire), il est beaucoup plus efficace d'utiliser un passage par référence lorsque les objets client et serveur sont locaux. Par contre, lors d'un appel d'opération sur un objet distant, il devient impossible d'utiliser un passage par référence, puisque les espaces mémoires ne sont pas partagés entre les objets : le système doit donc permettre de simuler ce genre de passage entre des objets distants, c'est à dire que lorsque l'objet serveur est distant, les paramètres sont sérialisés et transportés vers le nœud distant ou vers un mécanisme approprié d'échanges de données inter-processus. Une fois les données arrivées à bon port, un mécanisme d'extraction des paramètres entre en jeux et l'espace mémoire du client est alors recréé pour leurrer l'objet serveur. Ainsi, l'invocation de l'opération peut avoir lieu sans aucun problème. La sérialisation des paramètres doit être entièrement supportée pour tous les types de base du C++ et ce service doit veiller à ce que les types échangés, entre deux objets résidant sur des nœuds différents, soient correctement interprétés de part et d'autre. Cela implique la gestion de l'ordre des octets qui sont transmis sur le réseau.

Le système offre un mécanisme de **gestion de la mémoire**. Ce mécanisme a pour but de réduire la fragmentation et d'optimiser la rapidité du processus d'allocation de mémoire dynamique. Par exemple, certaines applications sont extrêmement exigeantes quant à l'allocation de mémoire. Un tel comportement a pour effet la fragmentation du silo de mémoire. Cette fragmentation peut s'avérer un problème grave pour les systèmes ayant peu de ressources. Les différents mécanismes du système DOES, comme le mécanisme de *marshaling*, sont à eux seuls très exigeants, c'est pourquoi l'échange de paramètres entre deux objets actifs du système nécessite l'utilisation du silo (*heap*) de mémoire. De plus, la pile (*stack*) ne peut pas être utilisée sans risque, car il peut toujours y avoir une sortie de la portée (*scope*) d'une fonction asynchrone et il en résulterait une destruction prématurée des paramètres. En résumé, les paramètres doivent donc transiter via le silo de mémoire qui est partagé par tous, de façon permanente.

Un **objet actif** est défini comme étant un objet réceptif aux stimuli externes. Il peut ainsi recevoir des évènements en provenance de plusieurs objets clients. Malheureusement, cette classe d'objets n'existe pas au niveau des librairies standards du C++. Pour pallier à cette lacune, le système DOES fournit donc un mécanisme simple de création d'objets actifs.

Afin d'offrir une portabilité maximale, la librairie de classes formant le système DOES doit reposer sur une couche de classes de bas niveau ayant pour rôle l'**abstraction du système d'exploitation**. Ainsi, lorsqu'une application doit être portée vers un environnement différent, les changements au code source sont limités et aucun appel de fonction du système d'exploitation ne doit être fait directement par les classes de haut niveau. Il existe à ce jour plusieurs types de protocoles réseau au niveau de la couche transport. Tous ces protocoles reposent sur des concepts différents et ils offrent tous des interfaces différentes. Encore une fois, afin de maximiser la portabilité du code source d'une application, le système DOES doit offrir une couche de classes de bas niveau ayant pour rôle l'**abstraction du réseau**.

Le risque d'erreur de communication entre des applications de type réseau demeure toujours présent. Bien entendu, les autres types d'erreurs continuent d'exister et ils ne doivent pas être négligés. La **gestion des erreurs** et leur traitement doit faire partie intégrante du système. Les réseaux actuels n'offrent pas une fiabilité à 100 %. Il peut survenir plusieurs erreurs lors d'une communication entre deux nœuds, surtout sur le réseau Internet. Le système doit donc être prêt à traiter ces erreurs et à prendre des mesures correctives, lorsque cela est possible. Si par malheur l'erreur n'est pas évitée, celle-ci doit être rapportée à l'application de façon standard. Par exemple, lorsqu'un nœud serveur se voit débranché du réseau pendant une connexion, le client doit en être informé après un certain laps de temps. Le traitement des erreurs doit reposer sur des techniques simples, accessibles et qui demandent peu de ressource. Quant au mécanisme de gestion par les exceptions (*Exception Handling*), il est trop lourd et trop peu supporté par les environnements embarqués : ce mécanisme n'a pas à être supporté.

Le système doit être entièrement **configurable**. Le développeur d'application utilisant le système DOES doit pouvoir décider de la valeur de chacun des paramètres de configuration comme par exemple : le délai de retransmission des paquets, le nombre de retransmissions maximal de chaque message, le nombre de connexions maximal pouvant être traitées simultanément, etc. Cette latitude permet, entre autre, une plus grande portabilité et une meilleure adaptation du système aux différents types d'applications pouvant en faire usage.

4.3 Caractéristiques supplémentaires

Le système DOES doit être conçu en vue d'offrir des services de distribution d'objets sur l'Internet : le protocole de la couche réseau utilisé est le protocole **IP**. Au niveau de la couche transport, le protocole doit être performant, léger et facilement configurable : le protocole utilisé pour le transport est le protocole **UDP**. Contrairement au protocole TCP, ce protocole n'est pas orienté connexion et il n'offre pas de garantie de livraison des paquets. Un protocole de gestion de connexion léger, robuste et entièrement paramétrable doit donc être développé pour assurer une bonne qualité de services. Par exemple, chaque minuteur (*timer*) doit pouvoir être configuré. De plus, le système DOES doit supporter les connexions point à point de façon autonome, c'est donc dire qu'une entité tertiaire ne doit pas être requise pour que la connexion entre deux objets puisse avoir lieu. Une application doit pouvoir être distribuée entièrement sans nécessiter un support centralisé.

Un **mécanisme fiable** doit être utilisé pour assurer la connexion entre les objets qui sont distants. Comme les objets peuvent évoluer sur différents nœuds du réseau, il est important pour un client de pouvoir se connecter à un objet serveur en tout temps, et ce, peu importe sa localisation. Le protocole UDP, de par sa nature, contient certains risques d'erreurs dans l'échange des paquets de données entre les objets. Une attention particulière doit donc être portée à ce sujet dans l'élaboration du protocole de contrôle des connexions inter-objet. Par exemple, les paquets doivent être réorganisés en séquence et leur réception doit être notifiée. Attention, il ne s'agit pas de recréer le mécanisme de connexion TCP qui est complexe et lourd, mais de réussir à assurer un minimum de fiabilité au niveau des échanges entre objets.

La **sécurité** au niveau des applications réseaux peut s'avérer très importante dans certains secteurs d'activité. Par exemple, une compagnie peut désirer utiliser un canal de communication

téléphonique crypté pour discuter de certains sujets secrets avec ses clients. Bien que cette caractéristique soit très importante, la cryptographie ne sera pas utilisée dans la première version du système. Par contre, le design doit être pensé en fonction de l'ajout éventuel d'un module de cryptage pouvant être intégré facilement.

Les nœuds d'un réseau peuvent contenir plusieurs objets appartenant à la même classe, ou à des classes différentes d'objets. Par exemple, sur un même hôte, plusieurs objets de la même classe peuvent être créés par un client, et chacun d'eux doit pouvoir être adressé indépendamment. Un mécanisme fiable doit être mis en place pour assurer l'acheminement de l'information et des requêtes de service. Le système doit être en mesure de démultiplexer l'information et de la redistribuer aux objets serveurs correspondants. La **localisation** des objets doit être assez flexible pour supporter plusieurs milliers d'objets, évoluant sur des nœuds différents à l'intérieur d'un réseau de systèmes embarqués distribués à un niveau mondial.

La gestion d'une application possédant des caractéristiques **temps réel** n'est pas simple, surtout lorsque des tâches accèdent à des ressources elles-mêmes localisées sur des nœuds distants du réseau. Le système DOES doit permettre à une telle application de gérer efficacement ses ressources temps réel. Le principal outil de gestion des contraintes temps réel devant être disponible, dans la première version du système, est la gestion d'opérations asynchrones.

Le système DOES doit supporter l'invocation statique d'**interfaces**. Cette approche est la façon naturelle d'invoquer une opération, car le lien entre le langage de programmation et l'interface est automatiquement réalisé par le compilateur. Une validation forte des types (*strong type checking*) a lieu au moment de la compilation. De plus, les interfaces définies statiquement sont faciles d'utilisation et elles requièrent peu de code. Par contre, la découverte et l'invocation dynamique d'interfaces n'ont pas à être supportées, car ce type d'invocation n'est pas naturel et il nécessite qu'une seule interface soit utilisée, pour l'ensemble des interfaces existantes et futures. Cette technique d'invocation dynamique exige beaucoup plus de code pour réussir à découvrir les opérations et les paramètres des différentes interfaces et cela n'est pas souhaitable pour une application embarquée.

Dans la première phase de développement, le système supporte uniquement le **langage C++**. Le respect du C++ et de ses caractéristiques est nécessaire. Tous les types de classes doivent être distribuables à l'aide des classes de base de DOES. Il est à noter que l'utilisation du langage IDL pour la définition des interfaces n'est pas nécessaire, mais ce mécanisme de spécification étendu pourraît quand même être utilisé dans une version future du système afin de combler les problèmes et les limitations du C++ quant à la définition des interfaces d'objets distribués.

L'utilisation, du *Proxy* ou du serveur, doit être **transparente** pour le client, c'est-à-dire que le *Proxy* ou le serveur doivent être utilisés de la même façon, sans qu'aucun changement ne soit requis au niveau du client. Par exemple, la création d'un objet serveur doit se faire avec l'opérateur *new()*, qu'il soit local ou distant. Ainsi, l'implémentation des objets clients reste simple. Les objets serveurs doivent être implémentés indépendamment d'une éventuelle utilisation dans un environnement distribué : un objet serveur doit être utilisable par un client qu'il soit local ou distant.

Il convient de présenter le système DOES sous le format d'une **librairie** de classes. Comme DOES est le cœur de l'application distribuée, une attention particulière doit être portée à l'efficacité des algorithmes, dans le but de maximiser les performances. Généralement, les systèmes embarqués ne possèdent que très peu de ressources, tant du côté du processeur que du côté des périphériques. Cette librairie doit donc être optimisée pour minimiser l'espace mémoire requis.

5. Protocole de messagerie orienté-objet

Ce chapitre contient les spécifications du protocole orienté-objet de distribution de messages nommé *Distributed Object-Oriented Messaging* (DOOM). En effet, afin de permettre à deux objets existant sur des nœuds différents d'un réseau, de pouvoir communiquer entre eux, il est nécessaire d'établir les règles régissant leurs échanges : ces règles visent donc à définir les contraintes de transport des paquets d'informations. À un plus haut niveau, les objets sont libres d'exposer tous les genres d'interfaces. Cet aspect a l'avantage d'offrir une très grande polyvalence et peu de contraintes au niveau de la conception d'un système à objets distribués.

Le protocole DOOM tente principalement d'offrir des règles d'interopérabilité simples et efficaces. Ces règles de communication servent à créer des associations virtuelles (liens) entre plusieurs objets distribués sur les nœuds d'un réseau hétérogène. Ce protocole, inspiré des protocoles GIOP et IIOP qui ont été définis par l'OMG [15], possède les caractéristiques suivantes :

Disponibilité : DOOM est conçu pour fonctionner avec le protocole réseau IP qui est le plus répandu au monde.

Efficacité : DOOM nécessite seulement un protocole de transport minimal de type *datagram* sans connexion et très efficace comme UDP. En plus, il ajoute à ce protocole une mince couche protocolaire nécessaire à la transmission des messages entre deux objets.

Simplicité : DOOM se veut simple et léger, mais en étant le plus puissant et le plus versatile possible, car cela permet une implémentation efficace et peu coûteuse, et ce, peu importe le type de système visé. Le protocole demande peu d'efforts d'ingénierie, peu de ressources CPU et peu de ressources mémorielles : il peut donc être avantageusement utilisé sur n'importe quel type de système embarqué opérant dans un environnement distribué.

Extensibilité : DOOM supporte des réseaux d'objets de dimensions variables. Il tend à s'adapter à la grandeur de l'Internet d'aujourd'hui et de demain, tout en restant efficace.

Neutralité : DOOM ne fait aucune supposition quant à l'architecture du réseau, du système et des objets sous-jacents. Tous les nœuds sont traités comme des entités opaques avec des architectures inconnues. La seule condition requise est qu'au moins une entité par processus soit capable d'envoyer et de recevoir des messages de type DOOM.

Les spécifications de DOOM comprennent un ensemble de onze sections. La première section nomme et résume les différents concepts de DOOM; la deuxième, parle de ses éléments et de ses identificateurs; la troisième, explique le principe d'action-réaction; la quatrième, définit les sept services offerts (liaison, création, destruction, appel d'opération, envoi de signal, verrouillage, localisation); la cinquième, traite du mécanisme d'accusé réception; la sixième, de la gestion des exceptions; la septième, de la politique de sécurité; la huitième, très importantes, du *marshaling* d'opération; la neuvième, du protocol de transport des messages; la dixième, de l'algorithme de transmission fiable des messages et la onzième, du format de ces messages.

5.1 Concepts et définitions

Plusieurs concepts, termes et définitions sont utilisés à l'intérieur de la description du protocole DOOM. La Figure 5.1 représente les principales entités du protocole ainsi que leurs relations.

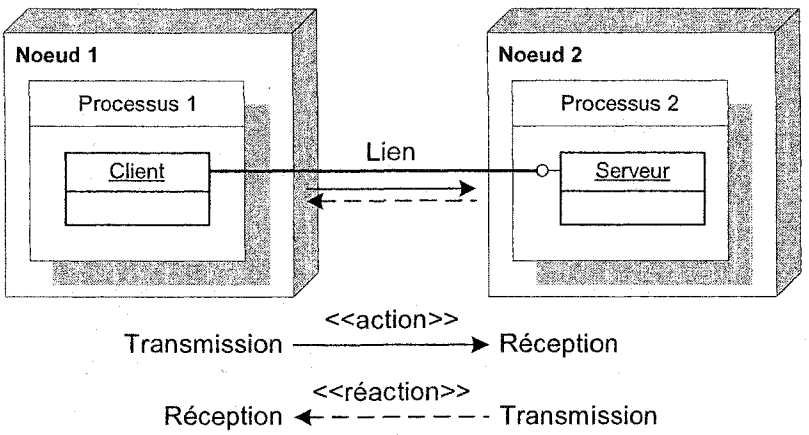

Figure 5.1 Les entités du protocole et leurs relations

Les paragraphes suivants définissent les termes et concepts utilisés tout au long de la définition du protocole.

Objet : entité discrète et identifiable. Il possède des frontières fixes, définies par ses interfaces ainsi qu'une identité encapsulant tous ses états et ses comportements. Il offre aussi un ou plusieurs services à tous ses clients. Un objet est l'instance d'une classe.

Interface : ensemble nommé d'opérations qui caractérise le comportement d'un objet.

Opération : entité identifiable qui dénote un service offert par un objet et qui peut être appelée par un autre objet. L'opération possède une signature décrite par son nom ainsi que par ses paramètres d'appels et son paramètre de retour.

Méthode : implémentation d'une opération. Elle spécifie un algorithme ou une procédure qui produit le résultat de l'opération.

Objet Client : type d'objet ayant pour rôle d'appeler des services d'une autre entité, le serveur. Il est celui qui initie le lien avec le serveur. Une fois le lien établi entre lui et un objet bien précis de type serveur, il peut transmettre ses requêtes à ce dernier.

Objet Serveur : type d'objet ayant pour rôle d'accepter les requêtes en provenance d'un ou de plusieurs clients. Il offre plusieurs services par l'entremise des opérations constituant son interface. Il est aussi créé et détruit sur ordre du client qui le possède (agrégation forte).

Lien : connexion individuelle entre deux objets par laquelle sont acheminés les messages. Le lien est l'instance d'une association entre deux classes.

Processus : flot de contrôle, connu du système d'exploitation, qui possède un espace mémoire privé et protégé. Tous les processus d'un nœud partagent les ressources disponibles et s'exécutent concurremment en fonction de la priorité qui leur est associée.

Nœud : élément physique qui représente une ressource avec des capacités de traitements et de mémoire. Un objet réside sur un nœud. Un nœud peut accueillir plusieurs objets pendant une même période de temps.

Message : information échangée entre deux objets; le message est composé d'un émetteur, d'un récepteur et d'une action. A chaque transmission et réception d'un message correspond un événement. Une action résulte de l'échange d'un message.

Action : abstraction d'un élément de calcul qui est uniquement exécutable de façon atomique. L'exécution de l'action doit être rapide pour ne pas modifier le comportement du système. L'action est habituellement l'exécution d'une opération primitive d'un objet. L'exécution d'une action résulte en un changement d'état au niveau de l'objet. Le comportement d'un système est bâti à partir d'actions primitives. Le système peut exécuter plusieurs actions simultanément, mais dans ce cas, leurs exécutions doivent être indépendantes. L'action primitive associée à un message est constituée d'un objet cible, d'une référence à une opération ou à un signal (la requête) et d'aucune, d'exactement une ou de plusieurs valeurs de paramètres.

Réaction : action qui résulte de l'exécution d'une action précédente.

Liaison : action de lier deux objets par la création et l'identification d'un lien. Cette action est implicitement réalisée lors de la construction. Elle possède uniquement la référence à un objet cible.

Création : action de créer et d'initialiser un objet. Cette action possède une référence à une classe ainsi qu'une référence à une opération et à la valeur de ses paramètres. L'opération doit se situer à l'intérieur de la portée de la classe. L'exécution de cette action crée une nouvelle instance de la classe. Si l'opération de création n'est pas explicite, l'objet est construit et ses attributs sont initialisés entièrement avec des valeurs par défaut.

Destruction : action de détruire un objet. Cette action possède uniquement une référence à un objet. Le résultat de l'exécution de cette action est la destruction totale de l'objet, de ses composantes et de ses liens.

Appel : action d'invoquer une opération d'un objet. L'action possède une référence à un objet cible, la référence à l'opération ainsi que les valeurs des paramètres de l'opération. Cette action est synchrone, c'est-à-dire que le client attend que l'exécution de l'opération se termine avant de reprendre le contrôle. La fin de l'exécution est signalée par le retour de l'opération ainsi qu'un résultat optionnel.

Retour : action de retourner le résultat d'une opération. Cette action transfère le contrôle au client qui a fait l'appel de l'opération. Elle est seulement permise à l'intérieur d'une opération invoquée précédemment par un appel. De plus, elle possède une liste optionnelle de valeurs qui est rendue disponible au moment où le client retrouve le contrôle.

Envoi : action d'envoyer un signal d'appel d'une opération à un objet. L'action possède une référence à un objet cible, la référence à l'opération ainsi que les valeurs des paramètres de l'opération. Cette action est asynchrone, c'est-à-dire que le client n'attend ni le commencement ni la fin de l'exécution de l'action pour continuer son exécution. Si un paramètre de retour est spécifié par l'opération, il est perdu.

Accuser Réception : action de confirmer la réception d'un message par un accusé de réception.

Verrouiller : action d'empêcher l'accès à un objet.

Publication : action de rendre public la référence à un objet ou à une classe d'objets.

Localisation : action de situer un objet, ou une classe d'objets, parmi les nœuds du réseau. Cette action possède des références à un objet et à une classe d'objets.

Transmission : action de faire parvenir un message. Le message transite de l'émetteur vers le récepteur.

Réception : action de recevoir un message. Le récepteur est l'objet qui reçoit le message en provenance de l'émetteur. La réception du message engendre l'action.

***Marshaling* :** sérialisation et extraction (*unmarshaling*) d'une opération à l'intérieur d'un flot d'octets contigus. Une fois sérialisées, les données respectent les règles d'encodage BDER. Ce flot d'octets peut alors être inséré à l'intérieur d'un message.

Règles d'encodage : règles d'encodage des types de données de bases. Ces règles standardisent le format de l'information contenue à l'intérieur des messages échangés par les objets d'un réseau hétérogène.

5.2 Identification des éléments

Afin de rendre efficace l'échange d'information entre les objets distribués d'un réseau de plusieurs nœuds, il est nécessaire de bien identifier chacun des éléments du protocole. Par exemple, pour identifier de façon unique, chacune des classes ainsi que chacun des objets qui existent à l'intérieur du système, le protocole fait usage de GUID. Mais comme la structure GUID est grande et complexe, il n'est pas possible de l'appliquer efficacement à tous les éléments. D'autres types d'identificateurs mieux adaptés sont donc nécessaires. Les paragraphes suivants décrivent chacun des éléments du protocole ainsi que leurs identificateurs.

L'*Action* ID (AID) identifie le type d'action ou de réaction transportée par un message. L'AID est un nombre entier non signé sur 16 bits. Il peut prendre les valeurs entières comprises entre 0 et 65 535. Les valeurs associées aux différentes actions sont décrites à la section 5.5.

L'*Operation* ID (OPID) identifie l'opération de l'interface du serveur qui doit être appelée. Chaque opération de l'interface d'un objet est représentée par un nombre unique à l'intérieur du contexte de la classe. L'OPID est un nombre entier non signé sur 32 bits. Il peut prendre les valeurs entières comprises entre 0 et 2^{32} - 1.

Le *Class* ID (CLSID) identifie une classe d'objets dont les opérations peuvent être invoquées à distance. Le CLSID est utilisé lors de la création, de la liaison ou de la localisation d'une classe d'objet. Le CLSID correspond à un GUID.

Le *Object* ID (OBJID) identifie un objet du réseau de façon unique. Le OBJID est utilisé lors de la liaison ou de la localisation d'un objet. Le OBJID correspond à un GUID.

Le *Node* ID (NODEID) identifie un nœud du réseau sur lequel s'exécute un certain nombre de processus. Le NODEID est constitué de l'adresse IPv4 de l'interface réseau du nœud en format numérique. Il est important de noter que les octets d'une adresse IP sont toujours disposés suivant la convention Big-Endian.

Le *Process* ID (PROCID) identifie le processus d'un nœud du réseau à l'intérieur duquel résident un certain nombre d'objets avec lesquels il est possible de communiquer. Le PROCID est constitué du numéro du port d'écoute UDP auquel le processus est lié. Le numéro de port est un nombre entier non signé de 16 bits. Il peut prendre les valeurs entières comprises entre 5000 et 65 535[4].

Le *Transmitter* ID (TxID) identifie, de façon unique pour tout le réseau, l'émetteur d'un message. Le TxID est formé des deux identificateurs NODEID et PROCID correspondant à ceux de l'émetteur.

Le *Receptor* ID (RxID) identifie, de façon unique pour tout le réseau, le récepteur d'un message. Le RxID est formé des deux identificateurs NODEID et PROCID correspondant à ceux du récepteur. **Le *Link* ID (LINKID)** identifie l'instance d'une association (lien) entre deux objets, de façon unique, pour tout le système. Donc, il est possible à partir du LINKID de déterminer exactement quel objet O_1 du nœud N_1 est lié avec quel objet O_2 du nœud N_2. Le LINKID est formé des trois identificateurs suivants : TxID, RxID et LKN. Comme il est possible qu'un ou plusieurs objets à l'intérieur du contexte d'un processus, communiquent avec un ou plusieurs objets d'un autre processus, les TxID et RxID ne sont pas suffisant pour identifier un lien de façon unique. C'est pour cette raison qu'un *Link Number* (LKN) est ajouté à ces deux identificateurs. Le LKN est un nombre entier non signé sur 32 bits : il peut prendre les valeurs entières comprises entre 0 et 2^{32} - 1. Comme la transmission d'un message se fait toujours en relation avec un lien, le LKN est transmis à l'intérieur de tous les messages afin que le nœud récepteur puisse bien identifier à quel objet sont destinés le message et son action. Autrement dit, le LINKID est utilisé pour le démultiplexage des messages. Lors de la liaison de deux objets, la valeur du LKN est générée par le nœud où réside le serveur. Ce nœud doit s'assurer que la valeur choisie est unique à l'intérieur du processus afin qu'il n'y ait aucune ambiguïté de créée. Bien qu'il soit possible de choisir une valeur aléatoirement, il est recommandé de procéder à une attribution incrémentale des numéros, pour éviter au maximum les erreurs de chevauchement. Une attention particulière est portée à la sélection du LKN au moment d'un *wraparound*. Une fois attribué, le LKN reste constant pendant toute la durée de l'association entre le client et le serveur. Il est important de noter qu'une fois deux objets liés, l'OBJID n'est pas requis à l'intérieur des messages suivants puisque les objets sont identifiés de façon unique par le LINKID.

[4] Le ports de 0 à 4999 sont généralement réservés par le système d'exploitation et les protocoles standards.

Le ***Message Sequence Number*** **(MSN)** établit la corrélation entre les messages d'action, de réaction et d'accusé de réception qui sont transmis sur un lien. Il indique aussi l'ordre dans lequel les messages sont transmis sur un lien. Mais il est de la responsabilité du transmetteur d'associer le MSN au message qui commande une nouvelle action. Un message de réaction ou d'accusé de réception copie toujours le MSN du message d'action qui les précède : c'est de cette façon que la corrélation est établie entre les messages circulant entre deux nœuds. Le serveur doit absolument exécuter les actions suivant l'ordre dicté par le MSN des messages, et non pas suivant l'ordre de réception[5]. Afin de pouvoir exécuter l'action d'un message (MSN = n), le serveur doit attendre d'avoir reçu et exécuté les actions de tous les messages précédents (MSN < n). Le MSN est un nombre entier non signé représenté sur 16 bits : il peut prendre les valeurs entières comprises entre 0 et 65 535. Lors de la liaison de deux objets, le MSN est égal à zéro (0). Chaque MSN suivant est incrémenté de 1 : il est attribué aux messages en fonction de l'ordre de leur transmission sur le lien. La réutilisation d'un ancien MSN sur un lien se fait lorsqu'il y a *wraparound*. Lorsque le MSN atteint son maximum, il est tout simplement remis à zéro lors de la transmission du message suivant.

Le message est identifié de façon unique par un TxID, un RxID, LINKID un MSN et un AID.

5.2.1 *Globaly Unique Identifier* (GUID)

Le problème d'assignation d'identificateur unique sur un réseau mondial comme l'Internet n'est pas trivial en soi. Comme DOES, les technologies de style OLE et RPC font aussi face à ce genre de problématique. C'est pour cette raison que l'*Open Software Foundation* (OSF) a créé l'*Universally Unique Identifier* (UUID) qui fut intégré à leur *Distributed Computing Environment* (DCE), standard sur lequel repose la technologie RPC de Microsoft Windows [41]. Un UUID correspond à un entier de 128 bits (16 octets) : il est virtuellement garanti que cet entier est unique dans le monde, et ce, indépendamment du temps et de l'espace. Cependant, revendiquer qu'un tel entier soit unique pour tout l'univers, est un peu présomptueux. Une unicité globale est plus réaliste. Voilà pourquoi plusieurs ont remplacé le U par un G.

Les GUID ne sont généralement pas manipulés directement. Ce sont des constantes qui servent à identifier une entité de façon unique. La représentation binaire d'un GUID est la suivante :

```
struct GUID
{
  unsigned long dw1;
  unsigned short w1;
  unsigned short w2;
  unsigned char b[8];
};
```

La morphologie du GUID permet l'existence de plusieurs techniques d'allocations différentes. La technique la plus utilisée incorpore les 48 bits qui identifient de façon unique l'interface Ethernet d'un ordinateur (*Ethernet MAC address*) avec le temps courant (UTC). De plus, cette technique incorpore des informations persistantes, pour éviter l'utilisation de données en provenance d'une

[5] Le protocole UDP n'offre aucune garantie quant à l'ordre d'arrivée des paquets [35]. Il est donc possible que le message 1 arrive après le message 2.

horloge défectueuse ou non synchronisée. En théorie, il est possible de générer des GUID différents à un rythme de dix millions à la seconde sur chacun des ordinateurs dans le monde, et ce, pour les prochaines 3 240 années[6].

5.3 Principe d'action-réaction

Le protocole DOOM est basé sur le principe d'**action-réaction** qui stipule que toute action engendre toujours une réaction. Deux rôles différents sont attribués aux objets d'un réseau : il y a le client et il y a le serveur. Le client est **proactif** et le serveur est **réactif**. D'abord, le client est celui qui initie le lien, en demandant la création du serveur ou sa liaison à un objet serveur déjà existant. Il transmet ensuite l'action via ce lien. Quant au serveur, il est celui qui reçoit l'action, l'exécute et retourne le résultat via une réaction : la réaction est en fait une action exécutée par le client.

Le protocole ne requiert qu'une négociation simple entre deux objets avant que le client puisse utiliser les ressources du serveur, car la création d'un nouvel objet, ou la liaison de deux objets existant, se fait par l'entremise d'une seule action. Puisqu'un lien est l'instance d'une association entre deux objets, un message est toujours véhiculé en corrélation avec un lien. À partir du moment où le lien est établi entre deux objets, le client peut transmettre des actions au serveur.

L'échange des messages entre les objets n'est pas symétrique, car l'action est transmise du client vers le serveur à l'intérieur d'un message, tandis que la réaction est transmise du serveur vers le client à l'intérieur d'un autre message. Seul le client peut transmettre une action et seul le serveur peut transmettre une réaction. Les autres types de messages peuvent être transmis par l'un ou l'autre des objets.

Le protocole ne définit aucun ordre quant à l'appel des opérations d'un objet. Il ne faut cependant pas confondre l'ordre d'exécution des opérations dicté par le client et l'ordre des messages transmis entre le client et le serveur. Les messages doivent toujours être traités en séquence par le serveur. Par exemple, si un client appelle l'opération A et ensuite l'opération B, alors le serveur doit absolument traiter A avant B. Par contre, rien n'empêche le client d'invoquer B avant A. Enfin, la dernière action à être exécutée par un serveur est toujours celle de destruction.

5.4 Définition des services

Au total, le protocole offre sept services. Chacun des services est réalisé à l'aide d'une action et de sa réaction associée. L'établissement du lien racine entre deux nœuds est toujours la première action à être exécutée. Il est ensuite possible de créer ou de lier les objets entre eux, suivant les demandes du client. Si le lien racine entre deux nœuds est brisé, ou recréé, tous les sous-liens seront considérés brisés et tous les objets seront détruits, à l'exception de ceux qui possèdent au moins un autre lien toujours actif. De la même façon, un serveur et son lien peuvent être détruits

[6] L'utilisation d'un logiciel spécialisé pour la génération de GUID est recommandée. Le logiciel guidgen.exe de Microsoft en est un exemple. Il est installé par défaut avec les plus récentes versions de MS Visual C++.

après un certain temps si le client n'acquiesce pas la réception de la réaction. Les sous-sections suivantes décrivent chacun des services.

5.4.1 Liaison

Le service de liaison sert à lier deux objets par la création et l'identification d'un lien. Le message *msgLink* qui transporte l'action *aLink* sert à invoquer la création d'un lien entre deux objets : le client et le serveur. Une fois les objets liés, le client peut transmettre ses requêtes au serveur. Le serveur, avec lequel le client veut être lié, est clairement identifié par son CLSID et son OBJID. Pour exécuter cette action, l'objet doit déjà exister sur le nœud cible et le CLSID et l'OBJID spécifiés à l'intérieur de l'action doivent correspondre à ceux de l'objet. Il est de la responsabilité du nœud de vérifier la concordance des deux valeurs avant d'accepter la requête du client qui souhaite le lien. Cette restriction vise à éviter l'utilisation par le client d'une interface qui n'est pas supportée par le serveur.

Dans plusieurs cas, il peut s'avérer utile pour un client de pouvoir créer un lien avec un objet déjà existant. Lorsqu'un objet migre vers un autre nœud du réseau, le client doit rétablir le lien avec celui-ci. Une fois localisé, le client transmet le message *msgLink* pour obtenir un nouveau LNKID.

Le message *msgLinked* transporte la réaction *rLinked.* C'est à l'intérieur de ce message que le LNKID est transmis au client. Autrement, un message d'erreur *msgNak* est retourné.

Un client peut vouloir briser le lien qu'il possède avec un objet sans pour autant commander sa destruction. Le message *msgUnlink* qui transporte l'action *aUnlink* est utilisé à cette fin. Il est probable que dans un cas d'associations multiples ou de persistance infinie, le client utilise ce message pour signaler qu'il ne souhaite plus les services du serveur. Une fois le message reçu et accepté par le serveur, le LNKID devient invalide et le client ne doit plus en faire usage.

Le message *msgUnlinked* transporte la réaction *rUnlinked* annonçant au client que le lien est maintenant rompu. Autrement, un message d'erreur *msgNak* est retourné.

Avant de pouvoir échanger des messages entre deux objets, il doit toujours y avoir une liaison initiale entre leurs nœuds respectifs. C'est sur ce lien que les messages des services de base du protocole sont échangés. Par exemple, le message de création d'un objet est transmis sur le lien établi entre le nœud du client et le nœud cible où doit être construit le serveur. Ce type de liaison est possible puisqu'un nœud est défini comme étant un objet de type serveur persistant qui ne peut être ni créé ni détruit. Lors de l'établissement du lien entre deux nœuds via le message *msgLink*, le numéro de séquence MSN du message est toujours égal à zéro (0). Il en va de même pour les identificateurs CLSID et OBJID qui doivent, eux aussi, être égaux à zéro (0).

5.4.2 Création

C'est par le service de création qu'un objet peut être créé sur un nœud différent du nœud où se situe le client qui en fait la demande. C'est le nœud cible qui exécute l'action. Pour que ce message puisse être accepté et traité, un lien doit avoir été préalablement établi, entre le nœud du client et le nœud cible où la création de l'objet doit avoir lieu.

Le message *msgCreate* qui transporte l'action *aCreate* sert à invoquer la création d'un objet sur le nœud cible. La classe de l'objet à créer, est précisée avec l'identificateur CLSID. De plus, le message contient l'opération de construction sous la forme d'un flot d'octets. Si l'objet ne peut pas être construit, alors un message d'erreur *msgNak* est retourné.

Le message *msgCreated* transporte la réaction *rCreated* annonçant au client que l'objet est maintenant construit et pleinement fonctionnel. Il transporte aussi l'identificateur OBJID de l'objet serveur nouvellement créé. A partir de ce moment, un lien entre le client et le serveur est activé. Ce lien, identifié par un LNKID, est généré par le nœud cible où le serveur est construit. En fait, l'action *aLink* est implicitement réalisée lors de la création d'un objet. En plus des identificateurs, le message *msgCreated* contient aussi les paramètres de retour (*out*) du constructeur.

La création d'un objet s'exécute de façon synchrone, c'est-à-dire que le client continuera son exécution seulement une fois le message *msgCreated* reçu. Après réception de ce message, le client peut continuer son exécution et faire appel aux services de l'objet. Par défaut, tous les objets créés sont publics. Cependant, il est possible au moment de la création d'un objet de le verrouiller. Dans ce cas, l'objet ne peut pas être partagé par plusieurs clients et seul le propriétaire peut lui transmettre des messages. Pour déverrouiller l'objet, le message *msgUnlock* doit être utilisé.

5.4.3 Destruction

Ce service commande la destruction du serveur. Le message *msgDestroy* transportant l'action *aDestroy* sert à invoquer l'opération de destruction du serveur. Le résultat de l'exécution de cette action est la destruction totale de l'objet, de ses composantes et de ses liens.

Une fois lié à un serveur par l'entremise des actions *aLink* ou *aCreate*, un client peut commander la destruction de l'objet à tout moment. Même si plusieurs clients sont toujours liés à un même serveur, la destruction doit être immédiate lorsqu'un message *msgDestroy* est reçu. Ce comportement est le même que celui obtenu à l'intérieur d'un programme C++, où plusieurs clients partagent une référence à un même objet et que l'un des clients décide d'appeler la fonction *delete* sur le pointeur. Cependant, aucun mécanisme de protection, comme le calcul du nombre de liens actifs, n'est défini par le protocole[7]. Si un tel mécanisme s'avère nécessaire, il peut toujours être implanté, mais sa définition ne fait pas partie de la porté du protocole DOOM, car il n'est pas désirable de spécifier un tel mécanisme de protection à ce niveau.

Une fois l'objet détruit, le client s'en voit informé par la réception du message *msgDestroyed* qui transporte la réaction *rDestroyed*. À partir de ce moment, le LNKID devient invalide. Si le client essai de transmettre un nouveau message sur le lien, il reçoit en retour un message d'erreur *msgNak*. Même chose pour les autres clients qui étaient liés à cet objet.

[7] Il est toujours possible d'utiliser le service de verrouillage pour éviter la destruction du serveur par un tiers.

5.4.4 Appel d'opération

L'appel d'opération est généralement le service le plus utilisé par un client. Une fois le serveur créé et lié au client, ce dernier utilise les services du serveur par l'entremise de l'appel des opérations disponibles via son interface.

Le message *msgCall*, transportant l'action *aCall*, sert à invoquer les opérations synchrones du serveur. Le message est aussi constitué du flot d'octets décrivant l'opération à être appelée. Une fois le message transmis, le client attend que l'exécution de l'opération appelée se termine, avant de reprendre le contrôle.

Le message *msgReturn* transporte la réaction *rReturn* annonçant au client que l'opération est terminée. C'est au moment de la réception de ce message que le client reprend le contrôle. Le retour d'opération ainsi que ses paramètres font partie du message. Ce message est transmis au client uniquement après l'exécution complète de l'opération. Cependant, lorsque l'opération désignée par l'action *aCall* ne peut pas être exécutée, un message d'erreur *msgNak* est retourné.

La réception du message *msgReturn* signifie que l'action a été exécutée avec succès par le serveur. Par contre, il est possible que l'opération ait échoué au niveau de l'objet, soit à cause de mauvais paramètres ou d'une séquence d'appels inappropriée ou, tout simplement, à cause de l'état de l'objet. Comme expliqué à la section 5.6, la gestion de ce type d'erreur n'est pas assumée par le protocole.

5.4.5 Envoi de signal

L'envoi d'un signal est un service particulier du protocole qui permet l'appel d'une opération asynchrone d'un objet. Il est possible pour un même client d'avoir plusieurs opérations asynchrones en cours d'exécution, sur un ou plusieurs serveurs, de différentes classes.

Le message *msgSend*, transportant l'action *aSend*, sert à invoquer les opérations asynchrones du serveur. Ce message est aussi constitué du flot d'octets qui décrit l'opération à être appelée. Donc, le client n'attend ni le commencement ni la fin de l'exécution de l'opération pour continuer son exécution. Il reprend le contrôle immédiatement après avoir transmis le message.

Le message *msgReceived* transporte la réaction *rReceived* annonçant au client que le signal est reçu. Il est transmis par le serveur aussitôt que l'action *aSend* est reçue. C'est au moment de la réception de ce message que le client reprend le contrôle. Aucun paramètre de retour ne fait partie du message. Si un paramètre de retour est spécifié par l'opération, ce dernier est perdu.

La réception du message *msgReceived* ne signifie pas que l'action sera exécutée avec succès par le serveur. Si un client veut connaître l'état du signal d'une opération, un mécanisme de notification doit être intégré à l'interface du serveur et les messages doivent être encapsulés à l'intérieur du protocole.

5.4.6 Verrouillage

Il se peut qu'un client veuille tirer profit des services d'un objet sans que d'autres n'y aient accès. Le verrouillage est un service étendu du protocole qui permet de restreindre l'accès à un objet.

Une fois verrouillé, l'objet ne peut pas être partagé : seul le propriétaire peut lui transmettre des requêtes. Pour déverrouiller l'objet, l'action *aUnlock* est utilisée. Cependant, un nœud peut accepter ou refuser de verrouiller l'accès à un objet selon ses propres paramètres. Il n'est donc pas garanti que l'action sera fructueuse et tout client doit être prêt à faire face à cette éventualité.

Le message *msgLock* transporte l'action *aLock* servant à verrouiller l'accès à un objet. Pour réussir à verrouiller un objet, le client doit préalablement créer un lien avec celui-ci. De plus, au moment de l'exécution de l'action de verrouillage, l'objet ne doit pas avoir été préalablement verrouillé : autrement, l'action est rejetée. Si deux clients font une demande de verrouillage d'un objet en même temps, c'est le premier message traité par le nœud qui l'emporte : il est donc possible que certaines implémentations offrent des conditions de course (*race conditions*). Pour éviter de telles conditions, l'objet peut aussi être verrouillé au moment de sa construction. Les liens déjà existants entre l'objet et d'autres clients ne sont pas invalidés par une action de verrouillage. Par contre, toutes les actions en provenance de ces derniers doivent être refusées ainsi que la création de nouveaux liens avec l'objet.

Le message *msgLocked* transporte la réaction *rLocked* annonçant au client que le verrouillage de l'objet est acquis. Dans le cas contraire, c'est un message d'erreur *msgNak* qui est retourné. Une fois verrouillé, le serveur accepte uniquement les messages en provenance du client dont l'adresse et le LNKID correspondent à ceux utilisés lors du verrouillage. De plus, l'objet ne peut plus migrer d'un nœud vers un autre. Le serveur doit attendre d'être déverrouillé avant de pouvoir se déplacer.

Lorsqu'un client veut partager l'accès à un objet qu'il a préalablement verrouillé, il le déverrouille grâce au message *msgUnlock* qui transporte l'action *aUnlock*. Pour que cette action puisse être traitée avec succès, le serveur doit être verrouillé par le client, autrement l'action n'a aucun effet Seule une action de déverrouillage en provenance du client possédant l'objet doit être acceptée par le serveur. Autrement, un message d'erreur *msgNak* est retourné.

Une fois le serveur déverrouillé, le message *msgUnlocked* transportant la réaction *rUnlocked* est retourné au client pour l'en informer. À partir de ce moment, tous les autres clients potentiels sont libres de faire appel aux services du serveur. Il devient ainsi possible pour un autre client d'acquérir un nouveau verrou sur ce serveur.

5.4.7 Localisation

La localisation est un service étendu, mais optionnel, qui permet la recherche de l'adresse d'un objet : le client qui ne connaît pas l'adresse d'un objet en particulier, peut transmettre un message *msgLocate* pour tenter de le localiser. Il est aussi intéressant, dans certains contextes, qu'un objet puisse être activé à différents endroits au cours de sa vie et qu'il puisse migrer de façon dynamique d'un nœud à l'autre. Cette capacité de migration des objets est réalisée grâce au service de localisation.

Un client ne doit jamais faire de présomption quant à la longévité de l'adresse d'un objet obtenue via le service de localisation. Une fois le lien brisé, le client peut tenter de réutiliser l'adresse précédente, mais si la tentative échoue, il doit être prêt à recommencer le processus de

localisation. Le client doit aussi se préparer à voir migrer le serveur vers un autre nœud du réseau, et ce, même après avoir utilisé une adresse avec succès.

L'action de localisation est transmise à l'aide du message *msgLocate*. Ce message identifie, entre autre, la classe et l'objet recherchés. L'action *aLocate* comporte plusieurs utilités. Premièrement, elle peut être utilisée par un client pour retrouver l'adresse d'un objet. Deuxièmement, elle peut servir à vérifier si la référence à un objet est toujours valide. Troisièmement, le client peut valider la capacité d'un nœud à créer une instance d'une certaine classe d'objets. Quatrièment, elle peut servir d'optimisation dans un contexte particulier où le client ne connaît pas la position exacte du serveur : il peut ainsi éviter toute retransmission d'un message de création qui contiendrait une grande quantité de données.

Pour qu'un objet soit localisé, il doit toujours y avoir une correspondance entre l'OBJID et le CLSID. Le CLSID doit toujours être valide. L'OBJID, quant à lui, n'est pas obligatoire et peut être égal à zéro : dans ce cas, le client ne recherche pas un objet en particulier, mais plutôt un nœud qui possède la capacité de créer une instance du type spécifié.

Le message *msgLocate* est particulier : il est le seul message à supporter plusieurs destinataires. Pour une localisation plus efficace au sein du réseau, il est donc possible de faire un *broadcast* du message. Dans cette situation, le client doit être prêt à recevoir plusieurs messages *msgLocated* en provenance de diverses origines. Le *broadcast* doit se faire sur le port **8 945**. Par contre, toutes les réactions doivent être transmises en *unicast*, seulement au PROCID d'origine.

L'information générée par une action de localisation est retournée via le message *msgLocated*. Le message *msgLocated* transporte la réaction *rLocated* à une action de localisation *aLocate*. Il n'est pas garanti que chaque localisation sera couronnée de succès : il se peut qu'un nœud soit dans l'impossibilité de localiser un objet. Par contre, lorsque le résultat de la recherche est positif, le message *msgLocated* peut être composé d'un IOR. Dans tous les cas, le client doit être informé de l'état final des recherches indiquant si la classe et l'objet ont été localisés ou non. Le résultat de l'action de localisation peut prendre six états différents tels que présentés ci-dessous :

ClassUnknown **:** la classe est inconnue du nœud cible et toutes les recherches initiées par ce nœud ont échoué. Cet état implique que l'objet recherché est aussi inconnu. Le message ne contient pas d'IOR.

ObjectUnknown **:** l'objet est inconnue du nœud cible et toutes les recherches initiées par ce nœud ont échoué. Par contre, la classe de l'objet spécifié n'est pas inconnue. Le message contient une référence IOR pointant sur un nœud capable de créer des instances de cette classe.

ClassHere **:** la classe est connue du nœud cible et il est capable d'en créer les instances. Cet état est retourné seulement lorsqu'aucun objet en particulier n'est recherché. De plus, il faut que le nœud cible possède lui-même une association avec la classe désirée. Le message *msgLocated* ne contient pas d'IOR puisque le client possède déjà l'information de localisation.

ObjectHere **:** l'objet existe sur le nœud cible et le message *msgLocated* ne contient pas d'IOR puisque le client possède déjà l'information de localisation.

ClassForward : la classe existe sur un nœud différent du nœud cible et le message contient une référence IOR.

ObjectForward : l'objet existe sur un nœud différent du nœud cible et le message contient une référence IOR.

Il est possible pour un nœud d'implémenter le service de localisation de façon partielle, c'est-à-dire qu'il peut être conçu de façon à accepter les actions dédiées aux classes et aux objets qu'il supporte, tout en retournant un état *ClassUnknown* ou *ObjectUnknown* pour les autres. Autrement dit, ce noeud n'effectue aucune recherche sur le réseau pour situer la classe ou l'objet. Dans ce cas, l'état du résultat de la recherche ne prendra jamais les valeurs *ClassForward* ou *ObjectForward.*

Le message *msgLocated* peut aussi être retourné à la suite d'une autre action. Un client peut recevoir en tout temps un message de type *msgLocated* lui indiquant le nouvel emplacement d'un objet. C'est par ce mécanisme que la migration des objets est assurée. En fait, l'action de localisation est implicite et son exécution peut remplacer chacune des autres actions : elle remplace donc l'action demandée lorsque la classe n'existe pas ou lorsque l'objet n'existe plus sur le nœud cible. Dans ce contexte, seuls les états de localisations suivants peuvent être obtenus : *ClassForward* et *ObjectForward.* Par exemple, si un appel d'opération est envoyé à un objet n'étant plus localisé sur le nœud ciblé par le message, le message *msgLocated*, avec l'état *ObjectForward*, est retourné au client pour lui indiquer la nouvelle position de l'objet dans le réseau.

Lors de la migration, il est primordial que les états des objets client et serveur soit conservés. Une fois la nouvelle position reçue, le client est responsable de retransmettre le message original au serveur nouvellement localisé. Cependant, comme le LNKID n'est plus valide, le lien doit être recréé avant de pouvoir transmettre l'action.

Lors de la localisation, c'est l'*Interoperable Object Reference* (IOR) qui est utilisé pour rapporter la référence exacte de l'objet au client en faisant la demande. La référence à un objet, lui-même accessible publiquement, doit aussi être publiée avec l'aide de l'IOR. La référence à un objet possède les caractéristiques suivantes :

Version du protocole : la version du protocole DOOM devant être utilisée pour communiquer avec l'objet défini par l'IOR.

Famille d'adresse : le type d'adressage utilisé pour référencer un objet sur le réseau. La version 1.0 du protocole ne supporte que la famille d'adresse IPv4 désignée par AF_INET.

Identificateur de classe : la classe de l'objet identifiée par un CLSID.

Identificateur d'objet : l'instance de l'objet identifiée par un OBJID.

Node ID : l'adresse IPv4 du nœud où réside l'objet.

Process ID : le port de réception des messages du processus où réside l'objet.

La structure exacte de l'IOR se retrouve à la section 5.11.18. Aucun mécanisme strict de publication des IOR n'est précisé par ce protocole. L'échange des IOR est assuré par les nœuds d'un système. Les différentes stratégies de publication ne sont pas traitées par ce protocole.

Il est à noter qu'il n'est pas nécessaire pour un nœud d'implémenter ce service, et le cas échéant, le message *msgNak* doit être retourné au client pour l'informer de l'incapacité du nœud à exécuter l'action *aLocate*.

5.5 Accusé de réception

Il est toujours possible que certains messages se perdent en cours de route[8], mais comme la réception de tous les messages du protocole est essentielle, il est nécessaire de retransmettre ceux qui n'arrivent pas à destination.

Pour le client, la réception du message de réaction confirme que le serveur a bien reçu le message d'action. En temps normal, le message de réaction sert d'accusé de réception au message d'action. Cependant, il est parfois utile d'envoyer un accusé de réception avant la fin de l'exécution de l'action, si celle-ci est trop longue, pour éviter la retransmission du message original par le client. C'est un message *msgAck* qui est utilisé dans ce cas.

Pour le serveur, la réception d'un nouveau message d'action confirme que le client a bien reçu le message de réaction. En temps normal, le message d'action sert d'accusé de réception au message de réaction. Cependant, il est parfois nécessaire pour le client d'envoyer un accusé de réception au serveur lorsqu'aucune autre action n'est transmise à l'intérieur du délai maximum défini à la section 5.10 du protocole. Comme dans le cas précédant, c'est le message *msgAck* qui est utilisé à cette fin.

Le message *msgAck* est identifié avec les mêmes LINKID et MSN que ceux identifiant le message d'origine et pour lequel un accusé de réception est transmis. Ce sont ces deux identificateurs qui permettent d'associer l'accusé de réception au message précédemment transmis. Cependant, lorsqu'un accusé de réception est reçu, toute retransmission du message original doit être stoppée.

5.6 Gestion des exceptions

Il est possible que des conditions exceptionnelles provoquent des erreurs. Les conditions suivantes en sont quelques exemples : l'en-tête du message est mal formé; l'action n'est pas supportée par le serveur; le contenu du message est erroné; etc. Lorsqu'un message d'erreur est reçu, cela signifie habituellement qu'une exception s'est produite au niveau du traitement du message précédent, ou de son contenu. Le message d'erreur peut provenir du serveur comme il peut tout aussi bien provenir du client.

---

[8] Ce fait est dû à la nature du protocole de transport UDP qui ne garantit pas la transmission des paquets.

Le message *msgNak* rapporte le niveau et le code d'erreur d'une exception bas niveau propre au protocole. Le niveau est un nombre indiquant la gravité de l'erreur. Pour la version 1.0 du protocole, seulement trois niveaux sur une possibilité de 256 sont définis : ces trois niveaux conviennent dans la majorité des cas. L'en-tête du message n'est cependant pas limité à trois niveaux et d'autres pourront être ajoutés aux besoins, dans une version future du protocole. Les différents **niveaux** sont :

- ***Critical*** : ce niveau indique qu'une erreur critique est survenue au niveau du système. Cette faute est généralement irrécupérable et le système est alors bloqué. Aucun autre traitement ne peut être effectué par le système avant que celui-ci ne soit réinitialisé. Par exemple, une fuite de mémoire (*memory leak*) sature les ressources du système, ce qui l'oblige à redémarrer.

- ***Error*** : ce niveau indique qu'un certain traitement a échoué sans pour autant obstruer le fonctionnement du système. Cette erreur est généralement isolée et irrécupérable pour le processus qui l'a rencontrée. Par exemple, l'exécution d'une opération synchrone est avortée à cause d'une erreur de traitement à l'intérieur de l'objet. Dans ce cas, la réponse ne peut pas être retournée au client, mais un *msgNak* l'informe de la situation.

- ***Warning*** : ce niveau indique qu'une erreur de moindre importance est survenue. Ce type d'erreur n'a pas d'impact réel sur le déroulement d'un processus. Le traitement peut continuer presque normalement après qu'il y ait eu récupération de l'erreur. Par exemple, lorsqu'un client demande la destruction d'un objet inexistant ou déjà détruit, un message d'erreur de niveau *warning* est retourné pour l'en aviser. Cette erreur n'a aucun impact et le client peut continuer normalement son exécution.

Chaque message d'erreur se voit associer un numéro de code. Tous les codes d'erreur du protocole sont préfixés et identifiés sur 32 bits. Les numéros de code et la signification des différentes erreurs se retrouvent à la section 5.6.1. Seules les erreurs de protocole, de communication et de *marshaling* peuvent être rapportées par ce mécanisme.

Le message *msgNak* ne doit pas être utilisé pour la signalisation des erreurs survenant au niveau des objets client et des objets serveur. Le mécanisme pour la notification des erreurs entre client et serveur doit être implémenté au niveau des interfaces des objets et encapsulé à l'intérieur des messages standards du protocole. Par exemple, si une opération retourne un booléen indiquant une exécution réussie ou non, ce booléen sera transmis via le mécanisme standard de retour d'opération avec le message *msgReturn*.

5.6.1 Codes et niveaux des erreurs

Un ensemble d'erreurs relativement complet est défini pour les besoins de DOOM. Un nom, un code, un niveau de sévérité et une définition sont associés à chaque erreur. Le format utilisé est :

nom (code, niveau) : définition

Les erreurs pouvant être signalées à l'intérieur d'un message *msgNak* sont :

UnknownCritical **(0, critical) :** une erreur inconnue de niveau *critique* est survenue.

***UnknownError* (1, error) :** une erreur inconnue de niveau *error* est survenue.

***UnknownWarning* (2, warning) :** une erreur inconnue de niveau *warning* est survenue.

***NodeShutdown* (3, critical) :** une erreur interne au nœud est survenue et le nœud doit redémarrer.

***ProcessShutdown* (4, critical) :** une erreur interne au processus est survenue et le processus doit redémarrer.

***OutOfMemory* (5, critical) :** il n'y a plus de mémoire disponible sur le nœud.

***OutOfResource* (6, critical) :** il n'y a plus de ressource disponible sur le nœud.

***ProcessUnknow* (7, error) :** le processus cible est inconnu du nœud.

***ClassUnknown* (8, error) :** la classe est inconnue du processus.

***ObjectUnknown* (9, error) :** l'objet est inconnu du processus; il a été détruit ou n'a jamais existé.

***LinkUnknown* (10, warning) :** le lien est inconnu.

***BrokenLink* (11, error) :** le lien est brisé et donc invalide.

***ActionUnknown* (12, warning) :** l'action primitive spécifiée n'est pas spécifié par le protocole.

***ActionUnsupported* (13, warning) :** l'action primitive spécifiée n'est pas supportée par le noeud.

***ActionRejected* (14, error) :** l'exécution de l'action primitive est refusée par le noeud.

***ActionAborted* (15, error) :** l'exécution de l'action primitive est annulée par le noeud.

***OperationUnknown* (16,warning) :** l'opération est inconnue de l'objet serveur.

***OperationUnsupported* (17, warning) :** l'opération n'est pas supportée par l'objet serveur.

***OperationRejected* (18, error) :** l'exécution de l'opération est refusée par l'objet serveur.

***OperationAborted* (19, error) :** l'exécution de l'opération est annulée par l'objet serveur.

***UnmarshalingFailed* (20, error) :** la reconstruction de l'opération a échoué.

***MessageUnknown* (21, warning) :** le type du message reçu est inconnu.

***MessageInvalid* (22, error) :** le format du message reçu n'est pas valide.

***ServerAborted* (23, critical) :** le serveur ne répond plus aux actions. Toutes ses opérations en cours d'exécution sont terminées et l'objet est détruit. Le lien est alors brisé.

5.7 Politique de sécurité

Aucun mécanisme d'authentification et de cryptage n'est spécifié par cette version du protocole. Cependant, le serveur peut toujours accepter ou refuser de traiter arbitrairement un message. De façon générale, le serveur doit accepter d'exécuter toutes les actions correctement formulées, mais il est libre d'établir n'importe quelle politique d'acceptation. Une décision peut être prise de façon gratuite. Par exemple, un mécanisme de sécurité peut vouloir rejeter tous les messages qui ne proviennent pas d'un nœud en particulier. Lorsqu'un message est rejeté, le serveur doit retourner un message d'erreur à l'initiateur du message : le client. Si le client persiste à transmettre des messages, le serveur peut les ignorer. De plus, il peut stopper le renvoi des messages d'erreur.

5.8 Marshaling

Un client communique avec un serveur uniquement via les opérations publiques de ce dernier. Autrement dit, l'état d'un objet est manipulé par l'invocation des différentes opérations constituant son interface. Dans le contexte d'un système d'objets distribués, le serveur peut exister à l'intérieur d'un processus différent de celui du client. Plus précisément, le serveur peut être soit local au processus du client, soit dans un processus différent, soit sur un autre nœud du réseau. L'appel d'une opération d'un objet situé sur un nœud distant, via un pointeur local au processus du client, n'est pas direct. Pour y arriver, les valeurs des paramètres caractérisant l'opération doivent être transportées d'un processus à un autre, sans qu'il y ait perte d'information. Arrivée à destination, l'opération doit être reconstruite à l'intérieur de l'espace mémoire du processus du serveur. La préparation de l'opération pour la transmission est désignée par le terme *marshaling*. Le terme *marshaling*, qui est générique, est utilisé dans l'industrie pour exprimer l'action de sérialiser n'importe quel type de données, à l'intérieur d'un flot d'octets, qui est prêt pour la transmission réseau via une infrastructure RPC.

Dans le contexte de DOOM, le *marshaling* correspond à la sérialisation d'une opération à l'intérieur d'un flot d'octets pour la transmission vers un processus distant. Le *unmarshaling* correspond à l'extraction de l'opération du flot d'octets et à sa reconstruction. Le *marshaling* possède les caractéristiques suivantes :

Ordre variable des octets : deux nœuds peuvent échanger des messages sans adapter l'ordre des octets. La convention d'ordre est encapsulée à l'intérieur du flot d'octets. Lorsque deux nœuds utilisent une convention d'ordre différente, le transmetteur détermine l'ordre et le récepteur est responsable de remettre les octets dans l'ordre correspondant à son ordre naturel. L'encodage des types de données est donc bi-canonique.

Alignement des octets : les types de base sont alignés selon leurs frontières naturelles à l'intérieur du flot d'octets. Cette contrainte permet une manipulation efficace des données sur les systèmes obligeant l'alignement des données en mémoire : par exemple, les processeurs MIPS.

Règles d'encodage des types de base (BDER) : ces règles décrivent et proposent un encodage pour chacun des types de données de base couvert par le protocole. Les types plus évolués, comme les structures ou les tableaux, doivent être construits à partir des types de base. Les types

de base peuvent tous être sérialisés de façon indépendante à l'intérieur d'un flot d'octets. Ce flot d'octets peut lui aussi être encapsulé à l'intérieur d'un autre flot d'octets.

Les prochaines sous-sections décrivent les règles et conventions régissant le *marshaling*.

5.8.1 Flot d'octets

Afin d'être transmise sur le réseau, une opération doit être sérialisée à l'intérieur d'un flot d'octets (*octet stream*). Un flot d'octets est une notion abstraite correspondant physiquement à un tampon de mémoire envoyé à un autre nœud situé sur le réseau, via un mécanisme de transport IPC (*Inter-Process Communication*) quelconque. Un flot d'octets est aussi comparable à une longue séquence (finie) d'octets commençant à une position bien déterminée. En termes C++, le flot d'octets correspond à un tableau d'octets de longueur finie. Les octets faisant partie du flot, sont numérotés de *0* à *n-1* (*n* représente la grandeur de la séquence). La position numérique d'un octet dans la séquence est appelée l'index.

Il est important de faire la distinction entre deux types de flot d'octets. Les en-têtes de message, qui sont des unités d'informations propres au protocole DOOM, sont représentés sous un format de flot d'octets. Les données sont, elles aussi, à l'intérieur d'un flot d'octets. Cependant, ce flot n'a aucun lien avec le flot précédent et il est utilisable indépendamment du type de message.

Il est à noter que l'octet est le plus petit élément et il doit toujours être traité de façon atomique. Il est constitué de 8 bits et ne peut pas être décomposé lors du *marshaling*. Son bit de plus fort poids doit toujours être sérialisé en premier.

5.8.2 Convention d'ordre des octets

Généralement, une attention particulière sera portée à l'ordre des octets véhiculés sur le fil d'un réseau hétérogène. La représentation en mémoire des types de base d'un nœud avec un processeur d'architecture Intel, n'est pas la même que celle d'un nœud avec un processeur d'architecture Motorola (ex : PowerPC). L'IETF propose la convention Big-Endian, appelée aussi *Network byte Order*, pour la transmission de données entre les nœuds d'un réseau. En temps normal, un processeur de type Intel respecte la convention Little-Endian et doit toujours effectuer une conversion de l'ordre des octets qu'il transmet ou qu'il reçoit. Cependant, il est évident que lorsque deux nœuds à processeur Intel échangent des données, beaucoup de temps est perdu à la réorganisation des octets : tout ce travail inutile peut être évité. Pour rendre le *marshaling* plus efficace, les deux parties indiquent l'ordre de transmission des octets.

Contrairement à la norme établie par l'IETF, les données de l'opération, qui constituent le flot d'octets transmis, ne sont pas nécessairement sérialisées suivant le *Network Byte Order*. Lors du *marshaling*, deux conventions d'ordre peuvent être utilisées. La convention **Big-Endian** spécifie que l'octet le plus significatif (MSB) de la donnée est toujours transmis en premier. La convention **Little-Endian,** quant à elle, spécifie que la transmission de l'octet le moins significatif (LSB) se fait en premier. Par exemple, l'octet de poids le moins significatif d'un *long*, encodé selon la convention Little-Endian, sera transmis sur le réseau en premier. Le dernier octet à être transmis sera le MSB (index 3).

Afin de permettre à deux nœuds, partageant la même convention, de communiquer sans surcharge inutile, le flot d'octets contient un drapeau indiquant quelle convention a été utilisée lors de la sérialisation. Le premier bit du flot représente un drapeau indiquant l'ordre d'encodage des données ayant été sérialisées : si la valeur du bit est 0, les données ont été encodées suivant l'ordre **Little-Endian**; si la valeur indique 1, les données ont été encodées suivant l'ordre **Big-Endian**. Cette valeur ne fait pas partie des données encapsulées, mais elle fait partie du flot d'octets contenant ces données. Basées sur la valeur du drapeau, les données sont ensuite sérialisées à l'intérieur du flot d'octets, selon les règles d'encodage BDER définies à la section 5.8.5. Les mêmes règles s'appliquent lorsque les données sont extraites du flot d'octets. À l'extraction des données, l'ordre des octets est inversé au besoin. Il est de la responsabilité du récepteur de changer l'ordre des octets si cela s'avère nécessaire.

Il est à noter que pour tout autre type de convention d'ordre qui serait en vigueur sur un nœud, ce dernier doit sérialiser les données suivant l'ordre Big-Endian ou Little-Endian, selon sa préférence.

5.8.3 Convention d'alignement des types de base en mémoire

De façon à permettre l'insertion et l'extraction des types de base du flot d'octets avec les instructions spécialisées d'un processeur, chaque donnée doit être alignée sur sa frontière naturelle. La frontière d'alignement naturelle d'un type de donné est identifiée comme étant les multiples de la grandeur du type (ou l'espace qu'il occupe en mémoire), cette grandeur elle-même calculé en nombre d'octets. Un type de grandeur *n* doit commencer à un index, à l'intérieur du flot de données, qui équivaut à un multiple de *n*. Pour tous les types couverts, *n* peut prendre l'une des valeurs suivantes : 1, 2, 4 ou 8. Par exemple, un type de 32 bits (4 octets) doit être aligné sur un index divisible par quatre. Lorsque nécessaire, un gap peut être inséré, avant un type de base, afin que l'index du premier octet soit aligné correctement (*padding*). La valeur des octets insérés est toujours 0. Le tableau ci-dessous résume les frontières d'alignement naturelles pour tous les types de base.

TABLEAU 5.1 Convention d'alignement en mémoire des types de bases

Type	Alignement en octets
bool	1
char	1
unsigned char	1
wchar_t	2
short	2
unsigned short	2
long	4
unsigned long	4
float	4
double	8
long double	8
enum	4

L'alignement est toujours relatif au commencement du message contenant le flot d'octets transmis sur le réseau. Le premier octet du flot se situe à l'index 0. Pour le flot d'octets résultant du *marshaling*, le premier octet correspond au drapeau indiquant l'ordre d'encodage des octets. Comme un flot d'octets peut être inséré à la suite d'un autre flot d'octets, celui qui précède doit avoir une longueur qui est un multiple de huit, sinon le *padding* du flot d'octets est nécessaire[9].

5.8.4 Marshaling d'une opération

Avant de transmettre une action, il est nécessaire d'y inclure la description de l'opération désirée. La signature d'une opération est constituée des éléments suivants : un nom (OPID); zéro ou un paramètre de retour; zéro ou des paramètres d'entrées (IN); zéro ou des paramètres de sorties; zéro ou des paramètres d'entrées-sorties. L'identificateur d'opération (OPID) spécifie le numéro de l'opération qui doit être appelée. Chaque opération de l'objet est représentée par un nombre unique à l'intérieur du contexte de son interface. C'est lors du *marshaling* que le bon numéro est attribué selon la méthode à appeler. Pour l'appel d'une opération, le flot d'octets décrivant l'opération doit contenir, dans l'ordre, les informations suivantes : l'identificateur de l'opération et la valeur de chaque paramètre d'entrée (IN). Chaque valeur doit être sérialisée suivant l'ordre de déclaration des paramètres de l'opération. Par exemple, pour l'appel de l'opération suivante :

```
int Exemple(IN short nP1, OUT short& rnP2, INOUT long* plP3);
```

[9] Tous les types actuels s'alignent naturellement sur un index qui est un multiple de 8.

Sont *marshaling* en Big-Endian correspond à :

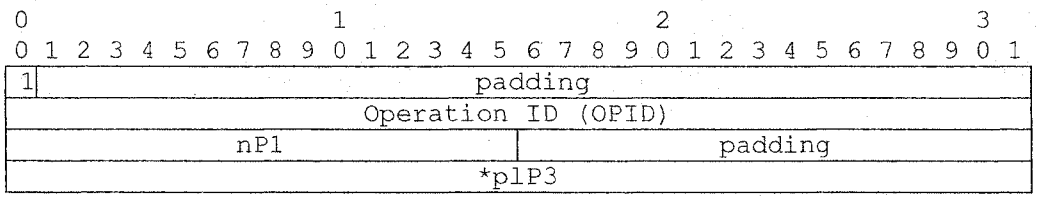

Figure 5.2 *Marshaling* des paramètres d'entrées

Il est à remarquer que seules les valeurs des paramètres IN et INOUT sont encapsulées à l'intérieur du flot d'octets.

Une fois exécutée, l'opération produit un résultat qui doit être transmis avec la réaction. Le flot d'octets décrivant le résultat doit contenir, dans l'ordre, les informations suivantes : la valeur du paramètre de retour et la valeur de chaque paramètre de sortie (OUT). Comme pour l'appel, chaque valeur doit être sérialisée suivant l'ordre de déclaration des paramètres de l'opération. Par exemple, au retour, le *marshaling* de l'opération décrite précédemment correspond à :

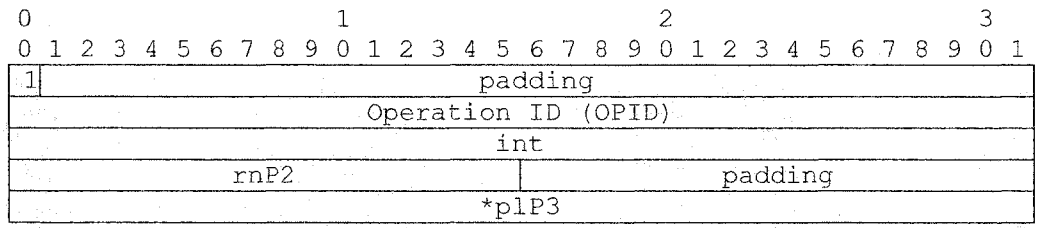

Figure 5.3 *Marshaling* des paramètres de sorties

Il est à remarquer que seules les valeurs du paramètre de retour ainsi que les valeurs des paramètres OUT et INOUT sont encapsulées à l'intérieur du flot d'octets.

5.8.5 Règles d'encodage

Pour l'échange des données, les règles d'encodage *Basic Data Encoding Rules* (BDER) sont définies. Ces règles sont utilisées pour le *marshaling* des données transférées entre des processus distincts. Dans cette version, tous les types de bases du C++ sont couverts, et ce, pour chacune des conventions. Les règles BDER offrent aussi une représentation pour chaque convention d'ordre des octets.

Nombres entiers : la Figure 5.5 illustre la représentation en mémoire des entiers. Les types d'entiers suivants sont représentés : *short* et *long*. L'ordre des octets et la grandeur du type sont aussi représentés par la figure. Les types signés sont représentés comme des nombres à double compléments (*two's-complement numbers*) [40] et la version non signée est représentée comme un nombre binaire non signé.

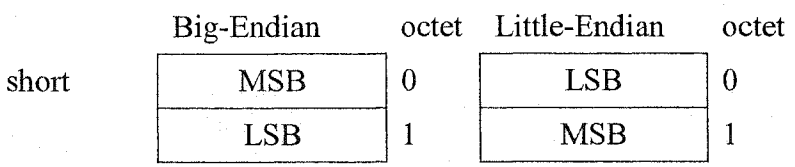

Figure 5.4 Encodage d'un *short*

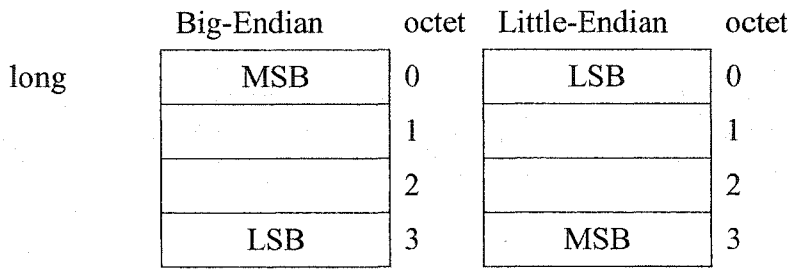

Figure 5.5 Encodage d'un *long*

Aucune règle n'est définie pour le type de base *int*. Comme ce type est de longueur variable en fonction de l'architecture du système, il devient impossible de l'encoder sans ambiguïté. Un entier de type *int* doit donc toujours être encodé comme un *long*, et ce, peu importe le processeur et le système d'exploitation. Cette contrainte limite le champ des valeurs d'un *int*, mais elle est nécessaire à l'interopérabilité entre les différents systèmes.

Nombres à point flottant : les Figure 5.6, Figure 5.7 et Figure 5.8 illustrent la représentation des nombres à point flottant. Cet encodage respecte entièrement le standard de l'IEEE[10] quant à la représentation des nombres à simple et à double précision. Par exemple, un nombre à simple précision (*float*), encodé selon la convention Big-Endian, est composé des éléments b, e et f dans l'ordre suivant : un bit pour le signe **s**, 8 bits pour l'exposant **e** (sur les figures : e = e1 + e2) et 23 bits pour la partie fractionnaire de la mantisse **f** (sur les figures : f = f1 + f2 + f3 + f_n). Cette représentation sur 4 octets donne un champ de valeurs allant approximativement de 3,4E-38 à 3,4E+38 pour le type *float*. Les types *float*, *double* et *long double* sont ainsi représentés :

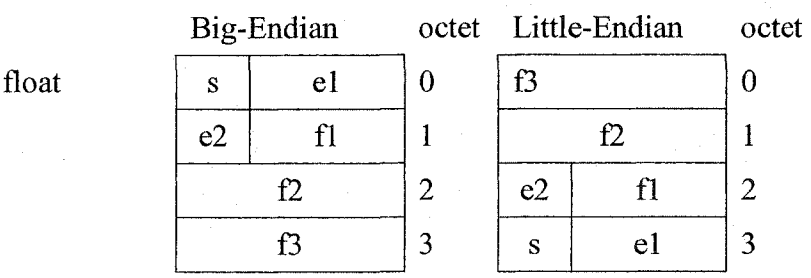

Figure 5.6 Encodage d'un *float*

[10] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

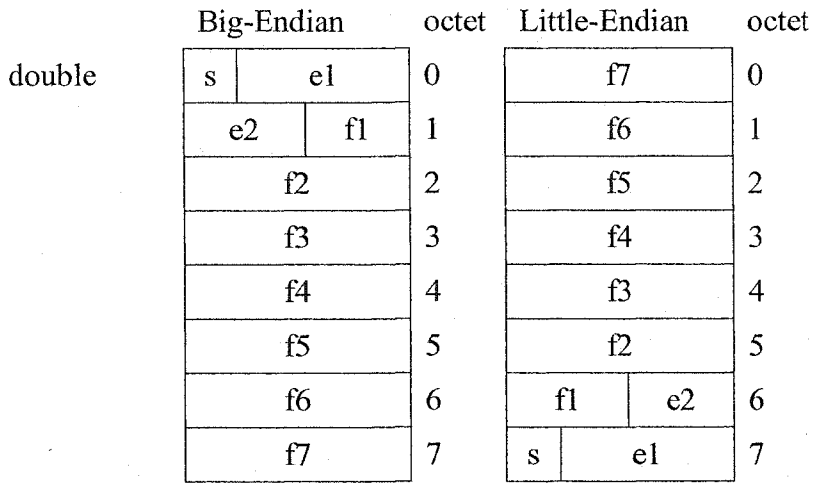

Figure 5.7 Encodage d'un *double*

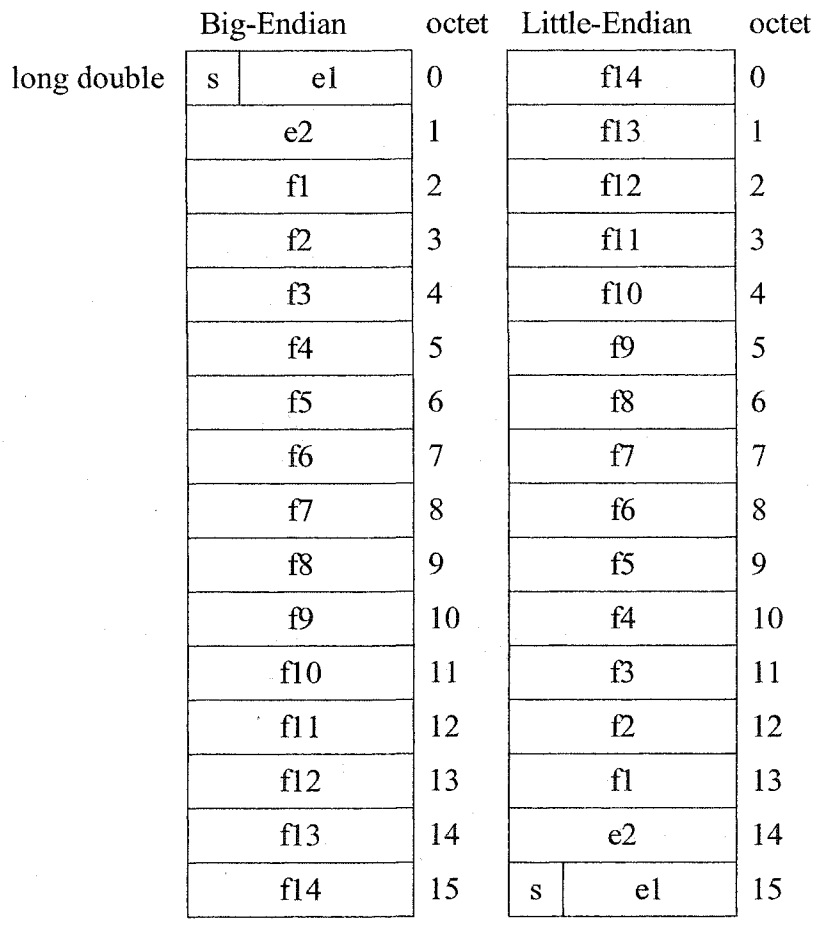

Figure 5.8 Encodage d'un *long double*

unsigned char[11] **:** correspond à un entier, non signé et non interprété, représenté par 8 bits. Son contenu doit être interprété comme un entier et ne doit pas subir de conversion durant une transmission. Il est à noter qu'un octet correspond à un *unsigned char* en C++.

bool : correspond à une valeur booléenne encodée sur 8 bit. Vrai correspond à la valeur 1 (0x01), tandis que faux correspond à la valeur 0.

char : correspond à un caractère alpha-numérique et encodé sur 8 bits. Chaque caractère correspond à un code d'affichage, défini soit à l'intérieur de la table des caractères standard US-ASCII (code 0-127), ou soit à l'intérieur de la table étendue (code 128-255) appelée *IBM Character Set*. Les objets qui échangent des caractères doivent respecter ce standard pour la transmission au niveau du réseau. Le récepteur est responsable de convertir la donnée en une autre représentation, si besoin il y a.

wchar_t : correspond à un caractère alpha-numérique étendu et encodé sur 16 bits. Chacun des caractères est encodé suivant la norme ISO/IEC 10646-1 (Universal Character Set). Les objets, échangeant des caractères à octets multiples, doivent respecter ce standard pour la transmission au niveau du réseau. Le récepteur est responsable de convertir les caractères en une autre représentation, si besoin il y a. La Figure 5.9 illustre la représentation en mémoire du type *wchar_t*.

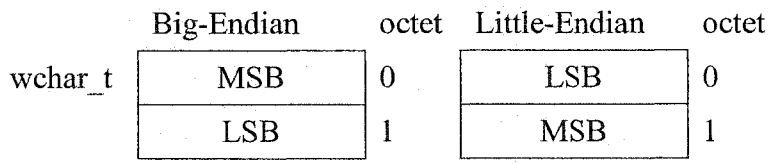

Figure 5.9 Encodage d'un *wchar_t*

Les types évolués, comme les structures, les énumérations ou encore les classes, sont construits à partir des types de base. Les types structurés n'ont aucune restriction quant à leur alignement. Cependant, l'alignement des types de base avec lesquels ils sont construits, doit être respecté.

struct : les éléments d'une structure sont encodés selon l'ordre de leur déclaration dans la structure. Chaque élément est encodé selon son type de base.

union : la valeur de l'*union* est encodée selon le type de son discriminant. Le discriminant correspond à l'élément de l'union ayant la plus grande dimension. Si plusieurs éléments possède la même dimension, le premier à être déclaré définie le format d'encodage.

tableau : chacun des éléments du tableau est encodé en séquence, selon le type du tableau. La longueur du tableau (nombre d'éléments valides) et la capacité (nombre maximal d'éléments pouvant être contenu) du tableau doivent précéder la valeur d'index 0[12]. La longueur et la capacité sont de type *unsigned long* et elles sont encodées dans l'ordre de déclaration présent.

---

[11] Le type unsigned char correspond à un octet (*byte*).

[12] L'utilisation des paramètres longueur et capacité permet de *marshaler* seulement les données utiles contenues par la tableau. Toutes les cellules situées après la dernière cellule valide n'ont pas à être *marshaler*.

Seuls les éléments valides doivent être sérialisés à l'intérieur du flot d'octets. La valeur de la capacité sert à recréer exactement le même espace mémoire, et ce, par l'agent qui extrait les données du flot d'octets.

Dans le cas d'un tableau de *char* représentant une chaîne de caractères, le caractère de terminaison (NULL) doit être encapsulé lui aussi. La longueur du tableau doit donc être égale à la longueur de la chaîne de caractères plus 1. La même chose s'applique à un tableau de *wchar_t*.

matrice : dans le cas d'un tableau multidimensionnel, les éléments de chacune des dimensions sont encodés en séquence selon le type de la matrice. Le nombre de dimensions, encodé sous la forme d'un *unsigned long*, doit être la première valeur à l'intérieur du flot d'octets. Ensuite, pour chacune des dimensions, la longueur et la capacité de celle-ci doivent être encodées de façon à précéder la première donnée. La longueur sert à définir le nombre d'éléments valides à l'intérieur de la dimension, tandis que la capacité de la dimension (nombre maximal d'éléments) sert à recréer le même espace mémoire (même taille) lors de l'extraction des données. La longueur et la capacité sont de type *unsigned long* et elles sont encodées dans cet ordre : longueur, capacité. Les éléments sont alors ordonnés en séquence, de façon à ce que l'index de la première dimension varie lentement, et de façon à ce que l'index de la dernière dimension varie rapidement. Seuls les éléments valides doivent être sérialisés à l'intérieur du flot d'octets.

L'exemple suivant illustre le flot d'octets résultant du *marshaling* de la matrice d'entiers à trois dimensions T :

```
long T[2][2][2] = { {{1, 2}, {3, 4}},
                    {{5, 6}, {7, 8}}, };
```

est *marshalé* en mémoire selon la convention Big-Endian de la façon suivante :

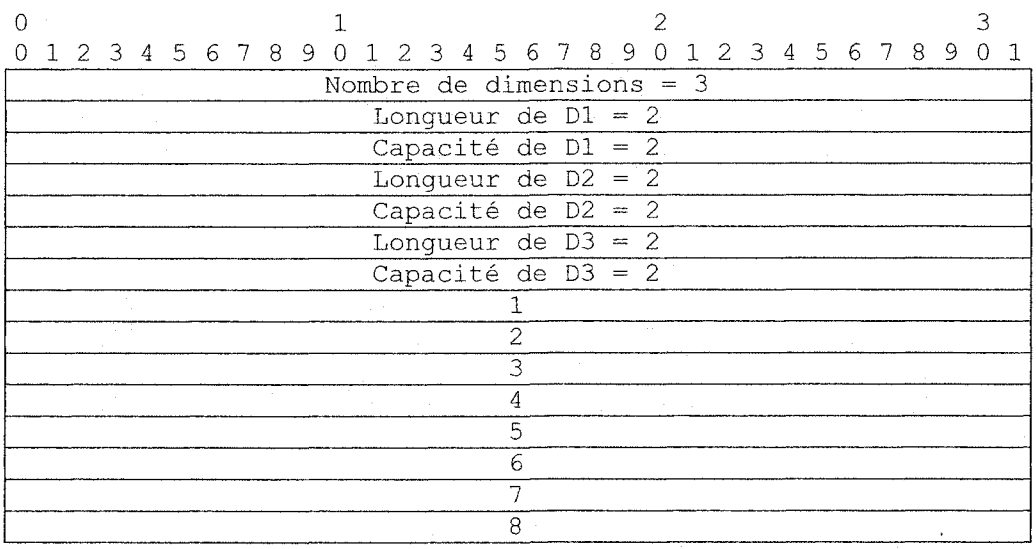

Figure 5.10 ***Marshaling*** **d'une matrice**

Dans le cas d'une matrice de *char* représentant des chaînes de caractères, le caractère de terminaison (NULL) doit être sérialisé lui aussi. La même chose s'applique à une matrice de *wchar_t*.

enum : les types énumérés sont encodés comme des *unsigned long*.

5.9 Transport des messages

Le protocole de transmission des messages se situe au niveau de la couche application du modèle TCP/IP. Le transport des messages entre les nœuds du réseau est réalisé par le protocole UDP basé sur le concept de *datagram*. Ce protocole est sans connexion et il est non fiable, car il y a possibilité de perte, de dédoublement et d'inversion de paquets. Bien que la transmission des messages soit conçue en fonction de UDP, il est possible d'utiliser tout autre protocole de transport offrant les même services. Les caractéristiques et les services du protocole de transport, requis par le protocole de transmission, sont les suivants :

- Le protocole de transport ne repose pas sur le concept de connexion. La duplication ou la perte de paquets est possible;
- Le transport des octets, situés à l'intérieur d'un paquet (datagram), doit être fiable. La couche de transport doit garantir que les octets d'un paquet, lorsque reçus, sont sans erreur et dans le même ordre qu'au moment de la transmission;
- Le protocole de transport doit pouvoir gérer des paquets d'une longueur d'au moins 512 octets. La longueur du paquet doit être indiquée;
- Le processus émetteur ainsi que le processus récepteur doivent être identifiables;
- Une notification doit être transmise lorsque le destinataire n'est pas en mesure de recevoir le message. Si un processus meurt, qu'un nœud ne répond plus ou que le lien réseau est brisé, un message de type *ICMP Port Unreachable* doit être transmis à l'émetteur pour l'en informer;

Un seul port UDP, par processus sur un nœud, sert à la fois pour la transmission et la réception des messages, et ce, pour les objets de type serveur et de type client existant à l'intérieur de ce processus. Le port par défaut est **22 500**. Le premier processus du nœud doit toujours se connecter à ce numéro. Les autres processus doivent utiliser un autre numéro qui, normalement, est le premier disponible suivant celui par défaut. Il va de soi que l'utilisation d'un numéro différent pour le port d'écoute par défaut doit être publiée et connue de tous les clients potentiels; le présent protocole ne régit pas le mécanisme de publication du port d'écoute, car différents types de mécanisme peuvent être utilisés selon les besoins. De plus, le numéro de port d'origine d'un message MA doit être utilisé comme port de destination pour transmettre le message de retour ACK, NAK, ou MR.

Il est à noter que les classes d'adresses IPv4, à l'intérieur de cette version du protocole, sont limitées aux classes A, B et C. La classe D (*multicast*) d'adresses n'est pas permise, ce type d'adressage étant réservé pour les versions futures de DOOM.

5.10 Transmission des messages

Comme il est spécifié à la section 5.9, le transport des messages entre les nœuds et les processus d'un réseau sont réalisés par un service de transport par paquets, sans connexion, avec possibilité de pertes et de duplications d'information. L'algorithme de transmission doit donc garantir que chaque message transmis est reçu par l'objet cible et que l'action ou la réaction associée est exécutée une et une seule fois. Autrement, la synchronisation des deux objets, client et serveur, ne serait pas garantie et l'état général du système serait invalide. L'algorithme de transmission est conçu de façon à permettre le chevauchement et l'entrelacement des messages échangés entre différents objets d'un réseau constitué de plusieurs nœuds et processus. les objets peuvant être situés sur des nœuds distincts ou non.

Pour ce faire, la transmission des messages est basée sur le concept de *three ways handshake*. C'est-à-dire que le transmetteur est informé de la réception d'un message par un accusé de réception. La troisième « poignée de main » vient du fait que pour chaque action, il y a toujours une réaction en provenance du serveur. La réception de ce message par le client génère alors un autre accusé de réception, ce qui termine l'échange de l'action et de la réaction entre les deux objets. Il est important de rappeler qu'un message contenant une action, transite toujours du client vers le serveur, tandis que la réaction associée est transmise du serveur au client.

La transmission d'une action et de sa réaction, entre deux objets, est atomique. Bien que l'information soit divisée entre plusieurs messages, il est impossible pour un client de transmettre une nouvelle action tant que l'échange en cours n'est pas terminé. Cependant, la séquence de transmission des messages doit toujours être complète entre les deux objets. De plus, le message de réaction ne peut pas provenir d'un serveur $\mathbf{S_B}$ lorsque l'action associée a été transmise du client $\mathbf{C_A}$ vers le serveur $\mathbf{S_A}$. Il faut bien comprendre que cette contrainte est appliquée uniquement à l'intérieur du contexte d'un lien entre deux objets. L'exécution des actions en provenance de plusieurs clients se fait de façon concurrente.

La transmission d'une action et de sa réaction se réalisent par l'échange d'un minimum de deux messages. En fonction du temps d'exécution de l'action, il peut aussi y avoir jusqu'à deux messages d'accusé de réception qui soient transmis, mais idéalement, ces messages ne sont pas utilisés. Comme la réaction est une action qui résulte de l'exécution d'une opération par le serveur, sa transmission est symétrique à celle d'une action, c'est-à-dire que le client et le serveur peuvent être considérés comme transmetteur et récepteur, et vice versa. C'est dans cette optique que l'algorithme de transmission est décrit. Par la suite, certains raffinements spécifiques aux types d'objets y sont apportés.

Afin de simplifier l'écriture, les symboles suivants sont utilisés :

Tx :	transmetteur;
Rx :	récepteur;
M :	message générique;
MACT :	message contenant une action (client $\Rightarrow$ serveur);
MRCT :	message contenant une réaction (client $\Leftarrow$ serveur);
MACK :	message d'accusé de réception (client $\Leftrightarrow$ serveur);

MNAK : message d'erreur ou de négation (client ⇔ serveur);
$\mathbf{C_{FT}}$: horloge du temps écoulé depuis la première transmission d'un message;
$\mathbf{C_{LT}}$: horloge du temps écoulé depuis la dernière retransmission d'un message;
$\mathbf{C_{LR}}$: horloge du temps écoulé depuis la réception du dernier message;
$\mathbf{D_{RET}}$: délai avant la retransmission d'un message resté sans accusé de réception;
$\mathbf{D_{MAX}}$: délai maximal pour la retransmission d'un message;
$\mathbf{D_{ACK}}$: délai maximal avant l'envoi d'un accusé de réception;
n : numéro de séquence d'un message (MSN);

Le transmetteur **Tx** est défini comme étant l'objet transmettant un message $\mathbf{M_n}$: ce message n'est pas un accusé de réception. Le récepteur **Rx** du message, est celui qui transmet l'accusé de réception **MACK** ou **MNAK**. La transmission d'un message entre ces deux objets est régie par les règles suivantes :

1. **MACT** est toujours transmis du client vers le serveur.
2. **MRCT est toujours transmis du serveur vers le client.**
3. **MACK peut être transmis du client vers le serveur et vice versa.**
4. **MNAK peut être transmis du client vers le serveur et vice versa.**
5. $\mathbf{C_{FT}}$ et $\mathbf{C_{LT}}$ doivent être remises à zéro au moment où $\mathbf{M_n}$ est transmis pour la première fois.
6. $\mathbf{C_{LT}}$ doit être remise à zéro après chaque retransmission de $\mathbf{M_n}$.
7. **Tx** se doit de retransmettre $\mathbf{M_n}$ après que $\mathbf{D_{RET}}$ se soit écoulé sans qu'aucun $\mathbf{MACK_n}$ ou qu'un message suivant $\mathbf{M_{n+1}}$ n'ait été reçu par Tx en provenance de Rx.
8. **Tx** doit garder en mémoire $\mathbf{M_n}$ aussi longtemps qu'il n'a pas eu acquiescement et que $\mathbf{D_{MAX}}$ n'est pas dépassé.
9. **Tx** doit stopper la retransmission de $\mathbf{M_n}$ et l'effacer de sa mémoire aussitôt qu'il reçoit un des messages $\mathbf{MACK_n}$, $\mathbf{MNAK_n}$ ou $\mathbf{M_{n+1}}$ en provenance de Rx.
10. $\mathbf{C_{LR}}$ doit être remise à zéro au moment de la réception de $\mathbf{M_{n+1}}$ par Rx et **Rx** doit accuser réception avant $\mathbf{D_{ACK}}$.
11. Si un duplicata du message $\mathbf{M_n}$ est reçu par **Rx**, un $\mathbf{MACK_n}$ doit être transmis sans délai.
12. Si **Rx** reçoit un message avec un **MSN** différent du **n** attendu, ce message doit être ignoré et effacé.
13. Si **Tx** reçoit un **MACK** pour un message dont l'accusé de réception fut déjà reçu, il doit ignorer le message.

14. Lorsque C_{FT} égale ou dépasse D_{MAX}, **Tx** peut considérer le lien brisé, arrêter la retransmission et effacer M_n.

Les règles suivantes s'appliquent à un objet de type **client** :

15. La réception par le client d'un message $MRCT_{n+1}$ en provenance de serveur équivaut à la réception d'un accusé de réception $MACK_n$ pour le message $MACT_n$; la règle numéro 9 doit s'appliquer immédiatement.

16. La réception du message $MRCT_{n+1}$ peut être signifiée par $MACT_{n+2}$. Dans certains cas, le client peut tirer avantage du fait que D_{ACK} est plus grand que zéro pour transmettre $MACT_{n+2}$ au lieu de $MACK_{n+1}$.

17. Le client doit toujours attendre la réception du $MRCT_{n+1}$ avant de transmettre le message $MACT_{n+2}$.

18. Si le serveur ne répond pas, le client doit présumer qu'il est détruit, ou encore, que le nœud cible est hors d'usage et que la dernière action n'a pas été exécutée.

Les règles suivantes s'appliquent à un objet de type **serveur** :

19. La réception par le serveur d'un message $MACT_{n+1}$ en provenance du client équivaut à la réception d'un accusé de réception $MACK_n$ pour le message $MRCT_n$; la règle numéro 9 doit s'appliquer immédiatement.

20. La réception du message $MACT_{n+1}$ peut être signifiée par $MRCT_{n+2}$. Dans certains cas, le serveur peut tirer avantage du fait que D_{ACK} est plus grand que le temps d'exécution de l'action pour transmettre $MRCT_{n+2}$ au lieu de $MACK_{n+1}$.

21. Si le client qui ne répond pas, le serveur doit présumer que le client est détruit, ou encore, que le nœud cible est hors d'usage. Si le lien était le dernier, l'autodestruction du serveur doit suivre immédiatement. Autrement, il doit continuer à servir ses autres clients normalement.

Pour le client, il est important de toujours pouvoir associer une réaction à une et une seule action. Pour le serveur, il est important de ne pas exécuter deux fois une même action dont le message aurait été dupliqué, par le réseau ou par une retransmission. Ce contrôle est possible grâce au numéro de séquence **MSN** du message. L'assignation et l'utilisation du numéro de séquence sont régie par les règles suivantes :

22. Tous les messages transmis sur un lien doivent être numérotés.

23. Le **MSN** du premier message à être échangé, entre deux objets nouvellement liés, doit toujours être égal à zéro (0).

24. Le **MSN** est attribué séquentiellement aux messages en fonction de l'ordre de leur transmission. Pour chaque nouveau message transmis par le client, le **MSN** doit être

incrémenté de 1 et pour chaque nouveau message transmis par le serveur, le **MSN** doit également être incrémenté de 1.

25. Le client et le serveur ne doivent jamais incrémenter le **MSN** lors de la transmission d'un **MACK** ou **MNAK**. Ils utilisent toujours le numéro du **MACT** ou celui du **MRCT** associé.

26. Lorsque la valeur du **MSN** atteint le maximum d'un entier encodé sur 32-bits, le **MSN** du $\mathbf{MA_{n+1}}$ suivant est égal à 0. Aucune gestion particulière du **MSN** n'est nécessaire dans ce cas, parce que les messages de numéros inférieurs et supérieurs au MSN courrantsont ignorés.

Bien que les paramètres de retransmission puissent être constants, il est préférable d'implémenter un algorithme d'ajustement adaptatif du délai, sensible aux types de réseaux sous-jacents.

Cette exigence provient du danger de non-application des valeurs par défaut à certaines topologies de réseaux comportant un délai de propagation plus court, plus long ou très variable. Les règles qui régissent l'ajustement du délai de retransmission d'un message sont les suivantes :

27. La précision des horloges $\mathbf{C_{FT}}$, $\mathbf{C_{LT}}$ et $\mathbf{C_{LR}}$ doit être au moins de l'ordre de la milliseconde.

28. La valeur minimale par défaut pour $\mathbf{D_{ACK}}$ et $\mathbf{D_{RET}}$ est de 250 msec[13].

29. La valeur par défaut suggérée du $\mathbf{D_{ACK}}$ pour le **Rx** est de 1000 msec : ce paramètre doit être constant et fixé à la même valeur sur tous les noeuds du réseau, autrement une retransmission inutile des messages MACT et MRCT pourraît survenir entre deux noeuds pour lesquels le D_{ACK} du premier est plus petit que le D_{RET} du second.

30. La valeur par défaut suggérée du $\mathbf{D_{RET}}$ pour le **Tx** est de 2000 msec et ce paramètre est adaptatif (voir plus bas) : la valeur de ce paramètre ne doit jamais être ajustée à moins de deux fois la valeur de D_{ACK}.

31. La valeur par défaut suggérée pour $\mathbf{D_{MAX}}$ est de 30 000 msec : ce paramètre doit être au moins quatre fois plus grand que la valeur par défaut de $\mathbf{D_{RET}}$. Si la valeur choisie pour D_{MAX} est trop proche de D_{RET}, il risque de n'y avoir aucune possibilité de retransmission.

32. La valeur de $\mathbf{D_{ACK}}$ doit toujours être au moins dix fois plus petite que $\mathbf{D_{MAX}}$ pour assurer la transmission d'un nombre acceptable d'accusés de réception du Rx vers le Tx. Cette règle permet d'éviter, lorsqu'il y a des pertes de paquets sur le réseau, que le Tx détecte faussement la perte du lien.

33. Pour chaque retransmission d'un message, à partir de la deuxième transmission, **Tx** doit ajuster $\mathbf{D_{RET}}$ à la hausse, suivant l'Équation 5.1.

---

[13] Il est important de noter que les valeurs par défaut suggérées sont ajustées en fonction du réseau Internet d'aujourd'hui où le délai de propagation peut atteindre facilement 500 msec.

34. Pour chaque nouveau message, $\mathbf{D_{RET}}$ est remis à sa valeur minimale.

35. Après cinq réajustements consécutifs de $\mathbf{D_{RET}}$, sa valeur minimale est doublée. Cette règle permet, par exemple, d'éviter des retransmissions inutiles sur un réseau pour lequel le délai de propagation est probablement plus grand que D_{RET}.

36. Après dix réceptions consécutives d'accusé de réception à l'intérieur d'un délai au moins deux fois inférieur à $\mathbf{D_{RET}}$, la valeur de ce dernier est divisée par deux, mais elle ne peut pas être inférieure à la valeur par défaut de $\mathbf{D_{RET}}$. Cette règle permet de réduire le temps de retransmission d'un paquet perdu sur le réseau et ainsi de diminuer le temps de réaction.

37. L'ajustement de $\mathbf{D_{RET}}$ doit se faire, pour chacun des liens, de façon indépendante. Il n'est pas dit que les chemins empruntés sur le réseau par les messages appartenant à deux liens différents d'un même nœud soient les mêmes et que les caractéristiques de propagation soient équivalentes. Par exemple, la communication entre les objets des nœuds A et B peut se faire sur un réseau local, tandis que la communication entre les objets des nœuds A et C peut se faire par l'Internet.

L'équation qui régit l'ajustement de $\mathbf{D_{RET}}$ est la suivante :

$$D_{RET} = D_{RET} * 2 + (char)\ rand()$$

Équation 5.1

Cette équation possède trois caractéristiques importantes : la composante exponentielle (D_{RET} * 2) ralentissant automatiquement le flot de messages en cas de congestion ou d'un long délai de propagation des messages; la composante aléatoire (résultat de la fonction *rand*) brisant toute synchronisation de transmission potentielle entre les nœuds d'un réseau; le *typecast* du retour de la fonction *rand()*[14], en un *char* signé, permettant d'ajouter ou de soustraire rapidement le facteur aléatoire compris entre [-128..127]. Cette technique évite l'utilisation de calcul à point flottant, de modulo et de division.

5.11 Format des messages

Cette section spécifie le format des messages transmis. Ces messages possèdent plusieurs caractéristiques, mais ils sont avant tout très simples. La structure d'un message est constituée d'au moins deux en-têtes : il y a toujours l'en-tête général suivi de l'en-tête spécifique à l'action. Tous les en-têtes qui forment un message sont alignés sur un multiple de huit octets (64-bits). Cette contrainte a pour but de s'assurer que les données de l'opération sérialisée soient alignées, elles aussi, sur huit octets.

Le format des messages est défini à l'aide de structures décrites en langage « C ». La convention d'alignement des octets *Big-Endian* est utilisée pour la description des structures. La position des éléments de chaque structures doit être adaptée pour les plates-formes ayant un processeur

[14] La fonction rand() doit être correctement initialisée avec la fonction srand(time()).

fonctionnant avec la convention *Little-Endian* : les processeurs x86 d'Intel en sont un exemple. De plus, chacun des champs de bits (*bit fields*) est aligné sur le bit le plus significatif. Une fois encore, leur position doit être adaptée pour les plates-formes ayant un processeur fonctionnant avec la convention *Little-Endian*.

La transmission des champs d'une structure à l'intérieur du paquet de transport doit se faire dans l'ordre de leur déclaration. La transmission d'un champ constitué de plus d'un octet (ex : *unsigned short* ou *unsigned long*) doit se faire suivant la convention définie par le drapeau **BYTEORDERING**. La transmission des octets (*unsigned char*) doit se faire suivant l'ordre de leur déclaration à l'intérieur de la structure. Pour ce qui est des bits formant un octet, l'ordre de leur transmission est géré par la couche de transport.

Comme le protocole de transport utilisé est UDP, il n'est pas nécessaire d'ajouter la grandeur du message à l'en-tête général. Cette valeur est extraite via un service de la couche de transport. Il est aussi important de noter que la taille du flot d'octets d'une opération est calculée à partir de la taille du paquet, moins la taille des en-têtes qui le précèdent.

Pour simplifier l'écriture des structures, les définitions suivantes sont établies :

```
typedef struct MSG;
typedef struct HDR;
typedef unsigned char[1] OCTETSTREAM;
```

Maintenant, la structure de l'en-tête général du protocole est :

```
HDR hdrProtocolID
{
  unsigned char ID[4];
  unsigned char VERSIONMAJOR;
  unsigned char VERSIONMINOR;
};
```

ID : cet élément identifie le type de protocole.

VERSIONMAJOR : cet élément représente la valeur la plus significative de la version du protocole.

VERSIONMINOR : cet élément représente la valeur la moins significative de la version du protocole.

```
HDR hdrRoot
{
  const hdrProtocolID PROTOCOLID = {'D', 'O', 'O', 'M', 0x01, 0x00 };
  unsigned short BYTEORDERING:1;
  unsigned short LOCK:1;
  const unsigned short PADDING:14 = {0};
  unsigned long  LKN;
  unsigned short MSN;
  unsinged short AID;
};
```

hdrProtocolID : les quatre premiers octets de l'en-tête identifient le protocole et ces octets doivent toujours contenir les quatre caractères "DOOM" encodés selon la table de caractères standards US-ASCII. Ils forment ainsi le nombre magique (0x4C494F50) qui doit toujours être le même pour tous les messages. Les deux octets suivants représentent la version du protocole et celle-ci s'applique à tous les éléments du protocole (marshaling, action, format des messages, etc.). VERSIONMAJOR, pour la présente spécification, vaut un (1) et VERSIONMINOR vaut zéro (0). Seuls les messages de type DOOM doivent être acceptés : tout autre type de message doit être rejeté, car un mauvais message doit générer le moins de traitement possible.

BYTEORDERING : ce drapeau définit l'ordre des octets pour les éléments du message à venir. La valeur zéro (0) indique que la convention *Big-Endian* est utilisée, tandis qu'une valeur un (1) indique que la convention *Little-Endian* est utilisée. Au niveau de l'en-tête général, les champs LINKID, MSN et ACTIONID dépendent de l'ordre des octets.

LOCK : ce drapeau, lorsqu'il est positionné à un (1), signifie que le client désire verrouiller l'objet. Il offre la même fonctionnalité que l'action *Lock*. Il peut être utilisé avec les actions *Link*, *Create*, *Call*, *Send* et *Lock*. Par exemple, le client peut désirer verrouiller l'objet au moment de sa création et positionner le bit LOCK à un (1), à l'intérieur du message de création. Pour déverrouiller l'objet, le message *Unlock* doit être utilisé.

PADDING : les drapeaux à l'intérieur de l'en-tête général sont représentés chacun par un bit, c'est-à-dire que les 16 bits suivants l'en-tête *hdrProtocol* sont réservés à cette fin, mais que seuls les deux premiers bits sont utilisés, car les 14 bits suivants sont réservés pour usage futur. Tous les bits inutiles doivent avoir une valeur égale à zéro (0).

LKN : cet élément est l'identificateur du lien entre le client et le serveur. Une fois alloué par le serveur, il doit rester le même pour tous les échanges jusqu'à ce qu'il y ait destruction de l'objet, ou rupture du lien.

MSN : cet élément est le numéro de séquence du message en relation avec le lien. À chaque action correspond une réaction dont le message possède le même MSN.

AID : cet élément est l'identificateur de l'action à exécuter. Le TABLEAU 5.2 et le TABLEAU 5.3 résument la liste des actions et des réactions ainsi que leurs identificateurs numériques. Les actions et les réactions, notées obligatoires (colonne Ob), doivent absolument être supportées.

TABLEAU 5.2 Actions ID

Action	AID	Ob
aLink	0x0001	Oui
aUnlink	0x0002	Oui
aCreate	0x0003	Oui
aDestroy	0x0004	Oui
aCall	0x0005	Oui
aSend	0x0006	Oui
aLock	0x0010	Non
aUnlock	0x0020	Non
aLocate	0x0030	Non

TABLEAU 5.3 Réactions ID

Réaction	AID	Ob
rLinked	0x8001	Oui
rUnlinked	0x8002	Oui
rCreated	0x8003	Oui
rDestroyed	0x8004	Oui
rReturn	0x8005	Oui
rReceived	0x8006	Oui
rLocked	0x8010	Non
rUnlocked	0x8020	Non
rLocated	0x8030	Non
rAck	0x0000	Oui
rNak	0xFFFF	Oui

Tous les messages doivent commencer avec l'en-tête *hdrRoot*. Cette structure identifie clairement tous les éléments communs d'un message. La version du protocole et le type d'action en sont des exemples.

Les sections suivantes spécifient le format des en-têtes de chacun des messages associés aux actions du TABLEAU 5.2 et du TABLEAU 5.3.

5.11.1 Message « *msgLink* »

L'en-tête de ce message est :

```
HDR hdrLink
{
  GUID CLSID;
  GUID OBJID;
};
```

CLSID : cet élément identifie de façon unique la classe d'objets avec lequel le client veut créer un lien.

OBJID : cet élément identifie de façon unique l'objet (l'instance de la classe CLSID) avec lequel le client veut créer un lien.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgLink
{
```

```
  hdrRoot;
  hdrLink;
};
```

Pour que la liaison soit acceptée, les deux identificateurs doivent correspondre et l'objet doit exister sur le nœud cible. Aucune opération n'est associée au message.

5.11.2 Message « *msgLinked* »

L'en-tête de ce message est :

```
HDR hdrLinked
{
  unsigned long LNK;
};
```

LINKID : c'est l'élément qui servira à identifier le lien entre le serveur et le client lors de l'échange des messages futurs.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgLinked
{
  hdrRoot;
  hdrLinked;
};
```

Aucune opération n'est associée au message.

5.11.3 Message « *msgUnlink* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgUnlink
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.4 Message « *msgUnlinked* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgUnlinked
{
```

```
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.5 Message « *msgCreate* »

L'en-tête de ce message est :

```
HDR hdrCreate
{
  GUID CLSID;
};
```

CLSID : cet élément spécifie exactement la classe de l'instance à créer.

Le message possède trois éléments encodés dans cet ordre :

```
MSG msgCreate
{
  hdrRoot;
  hdrCreate;
  OCTETSTREAM osOperation;
};
```

L'opération de construction choisie par le client, le constructeur, doit être sérialisée selon les règles de *marshaling* énoncées précédemment. Le flot d'octets résultant est inséré à la suite de l'en-tête *hdrCreate*.

5.11.6 Message « *msgCreated* »

L'en-tête de ce message est :

```
HDR hdrCreated
{
  GUID OBJID;
  unsigned long LNK;
  unsigned long PADDING;
};
```

OBJID : cet élément identifie de façon unique l'objet créé.

LINKID : c'est l'élément qui servira à identifier le lien entre le serveur et le client lors de l'échange des messages futurs.

PADDING : les 32 bits qui suivent le LINKID servent à l'alignement du flot d'octets de l'opération. Tous les bits de *padding* doivent avoir une valeur de zéro (0).

Le message possède trois éléments encodés dans cet ordre :

```
MSG msgCreated
{
```

```
    hdrRoot;
    hdrCreated;
    OCTETSTREAM osOperation;
};
```

Le retour de l'opération de construction, exécutée lors de la création de l'objet, doit être sérialisé selon les règles de *marshaling* énoncées précédemment. Le flot d'octets résultant est inséré à la suite de l'en-tête *hdrCreated.*

5.11.7 Message « *msgDestroy* »

Une fois que les services du serveur ne sont plus requis par le client, le serveur doit être détruit. Le message *msgDestroy* est utilisé à cette fin : l'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgDestroy
{
  hdrRoot;
};
```

Aucune opération n'est spécifiée à l'intérieur du message puisqu'il n'y a jamais d'échange de paramètres à la destruction.

5.11.8 Message « *msgDestroyed* »

La destruction complète de l'objet est annoncée au client, grâce au message *Destroyed.* L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgDestroyed
{
  hdrRoot;
};
```

Aucune opération n'est spécifiée à l'intérieur du message puisqu'il n'y a jamais d'échange de paramètres à la destruction.

5.11.9 Message « *msgCall* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgCall
{
  hdrRoot;
```

```
  OCTETSTREAM osOperation;
};
```

L'opération choisie par le client doit être sérialisée selon les règles de *marshaling*. Le flot d'octets résultant est inséré à la suite de l'en-tête *hdrRoot*. Cette opération ne peut pas impliquer une action de création ou de destruction.

5.11.10 Message « *msgReturn* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgReturn
{
  hdrRoot;
  OCTETSTREAM osOperation;
};
```

Le retour de l'opération doit être sérialisé selon les règles de *marshaling*. Le flot d'octets résultant est inséré à la suite de l'en-tête *hdrRoot*.

5.11.11 Message « *msgSend* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgSend
{
  hdrRoot;
  OCTETSTREAM osOperation;
};
```

L'opération choisie par le client doit être sérialisée selon les règles de *marshaling*. Le flot d'octets résultant est inséré à la suite de l'en-tête *hdrRoot*. Cette opération ne peut pas impliquer une action de création ou de destruction.

5.11.12 Message « *msgReceived* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgReceived
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.13 Message « *msgLock* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgLock
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.14 Message « *msgLocked* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgLocked
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.15 Message « *msgUnlock* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgUnlock
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.16 Message « *msgUnlocked* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgUnlocked
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.17 Message « *msgLocate* »

L'en-tête de ce message est :

```
HDR hdrLocate
{
  GUID CLSID;
  GUID OBJID;
};
```

CLSID : cet élément identifie de façon unique la classe d'objets recherchée. Cet identificateur ne peut pas être nul.

OBJID : cet élément identifie de façon unique l'objet recherché (l'instance de la classe CLSID). Cet identificateur peut être nul.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgLocate
{
  hdrRoot;
  hdrLocate;
};
```

Aucune opération n'est associée au message.

5.11.18 Message « *msgLocated* »

L'en-tête de ce message est :

```
enum eLocalizationStates
{
  ClassUnknown      = 0,
  ObjectUnknown     = 1,
  ClassHere         = 2,
  ObjectHere        = 3,
  ClassForward      = 4,
  ObjectForward     = 5
};

HDR hdrLocated
{
  eLocalizationStates LOCATIONSTATUS;
};
```

LOCATIONSTATUS : cet élément correspond au résultat de l'action de localisation. Les résultats *ClassForward* et *ObjectForward* indique que le message se poursuit avec un en-tête IOR.

```
HDR hdrInteroperableObjectReference
{
  hdrProtocolID PROTOCOLID;
  unsigned char BYTEORDERING:1;
  const unsigned char PADDING:7 = {0};
  const unsigned char ADDRESSFAMILY = {0};
  GUID CLSID;
  GUID OBJID;
};
```

PROTOCOLID : cet élément est l'identificateur du protocole qui est répété à l'intérieur de l'IOR pour permettre à un client de connaître, à l'avance, la version du protocole nécessaire pour communiquer avec le serveur. (Voir la description de ce champ au niveau de l'en-tête général à la section 5.5.)

BYTEORDERING : il est possible que l'IOR soit généré par un système différent de celui qui transmet la localisation. Dans ce cas, il est donc nécessaire d'ajuster l'ordre des octets différemment pour cette section du message. (Voir la description de ce champ au niveau de l'en-tête général à la section 5.5.)

PADDING : ce sont des bits de valeur égale à zéro pour fin d'alignement.

ADDRESSFAMILY : cet octet contient un identificateur représentant le type d'adressage qui est supporté par le nœud où vit l'objet. La valeur de ce membre indique le type d'adresse qui suivra. La version 1.0 du protocole ne supporte que la famille d'adresses IP, désignée par la valeur zéro (0), et décrite par l'en-tête hdrInternetObjectAddress (IOA).

CLSID : cet élément identifie, de façon unique, le type d'objet avec lequel le client veut créer un lien. Le CLSID doit être égal à la valeur spécifiée par le client.

OBJID : cet élément identifie, de façon unique, l'objet avec lequel le client veut créer un lien. L'OBJID doit être égal à la valeur spécifiée par le client, sauf dans le cas où la référence spécifiée n'indiquerait aucun objet en particulier. A ce moment, c'est le GUID de l'objet trouvé qui est retourné au client.

```
HDR hdrInternetObjectAddress
{
  unsigned short PROCID;
  const unsigned short PADDING = {0};
  unsigned long NODEID;
};
```

PROCID : cet élément correspond à l'adresse du serveur dans le contexte du nœud. Cette adresse, en format numérique, correspond au numéro de port UDP utilisé pour la réception des messages par le serveur. (Voir la description du PROCID à la section 5.2.)

PADDING : ce sont des bits de valeur égale à zéro pour fin d'alignement.

NODEID : cet élément correspond à l'adresse du nœud où se trouve l'objet. L'adresse est en format numérique. (Voir la description du NODEID à la section 5.2.)

Le message possède jusqu'à quatre éléments encodés dans cet ordre :

```
MSG msgLocated
{
  hdrRoot;
  hdrLocated;
  hdrInteroperableObjectReference;
  hdrInternetObjectAddress;
};
```

L'IOR est présent uniquement lorsque l'état de localisation est égal à *ClassForward* ou *ObjectForward*. Lorsque l'état de localisation est négatif, les en-têtes IOR et IOA ne sont pas présents. Un seul IOR est permis à l'intérieur du message.

Aucune opération n'est associée au message.

5.11.19 Message « *msgAck* »

L'en-tête de ce message est vide et cet en-tête ne possède pas de corps puisque toute l'information requise se situe à l'intérieur de l'en-tête général.

Le message possède un élément encodé dans cet ordre :

```
MSG msgAck
{
  hdrRoot;
};
```

Aucune opération n'est associée au message.

5.11.20 Message « *msgNak* »

L'en-tête de ce message est :

```
HDR hdrNak
{
  unsigned long  ERRORCODE;
  unsigned char  ERRORLEVEL;
};
```

ERRORLEVEL : cet élément correspond au niveau d'erreur. Plus le nombre est petit, plus l'erreur est grave. Les niveaux définis pour la version 1.0 du protocole sont : *Critical* (0), *Error* (1) et *Warning* (2).

ERRORCODE : cet élément correspond au code identifiant la cause de l'erreur. La section 5.6.1 décrit chacune des différentes erreurs.

Le message possède deux éléments encodés dans cet ordre :

```
MSG msgNak
{
  hdrRoot;
  hdrNak;
};
```

Aucune opération n'est associée au message.

6. Architecture de distribution d'objets

Dans un système à objets distribués, l'objet serveur par rapport au client peut résider soit à l'intérieur du même processus, soit à l'intérieur d'un autre processus, soit à l'intérieur d'un processus qui s'exécute sur un nœud distant. C'est alors que deux questions importantes se posent : premièrement, comment l'invocation d'une opération de construction peut-elle retourner un pointeur valide à un objet nouvellement construit sur un nœud distant; deuxièmement, comment ce pointeur peut-il ensuite être utilisé par un objet client, distant de l'objet serveur? Le chapitre précédent répond en partie à ces deux questions en établissant les règles qui régissent la communication entre les objets à travers les frontières des processus. Le présent chapitre complète l'information : il décrit les classes réalisant le protocole DOOM et permettant d'obtenir et d'utiliser un pointeur sur un objet serveur, résidant sur un nœud distant du nœud client.

L'implémentation du protocole DOOM est basée sur un ensemble de classes. Le nombre de classes formant cet ensemble est peu élevé. Certaines classes sont abstraites, tandis que d'autres sont passives ou actives. Afin d'éviter la redondance, la description des classes est plus ou moins détaillée selon leurs caractéristiques et leurs responsabilités, déjà définies ou non, à l'intérieur du cinquième chapitre. Il est aussi important de noter que les concepts et définitions, présentés à la section précédente, sont appliqués aux classes de l'architecture. Par exemple, le terme client désigne toujours un objet faisant usage de l'interface d'un autre objet, soit le serveur.

Bien que le protocole DOOM soit simple comparativement à d'autres protocoles du même genre, ce chapitre traite uniquement des classes essentielles à l'implémentation des services de bases pour la distribution des objets. Les services avancés, tels la localisation et le verrouillage sont omis. La conception complète et détaillée de tous les services offerts par DOOM dépasse le cadre de ce mémoire.

La Figure 6.1 présente les entités impliquées dans l'appel d'une opération d'un objet distant. Les objets client et les objets serveur utilisent les services du *Proxy* et de l'*Adapter* pour communiquer entre eux de façon transparente, via le réseau. Le rôle principal du *Messenger* est d'abstraire le protocole DOOM. Il est en charge de l'acheminement des messages du client au serveur et vice versa. Autant pour le client que pour le serveur, l'abstraction de la distribution est totale, c'est-à-dire qu'aucune dépendance n'est générée par l'architecture et que les objets client et les objets serveur peuvent être utilisés, dans un contexte distribué ou non, sans modifier leur code source. Du côté client, l'appel d'une opération passe à travers le *Proxy* pour ensuite être transmis sur le réseau par le *Messenger*. L'opération doit d'abord sérialisée avant d'être transmise : cela permet à l'appel de franchir les frontières des processus. Même si un seul objet *Messenger* est illustré, chacun des processus possède sa propre instance.

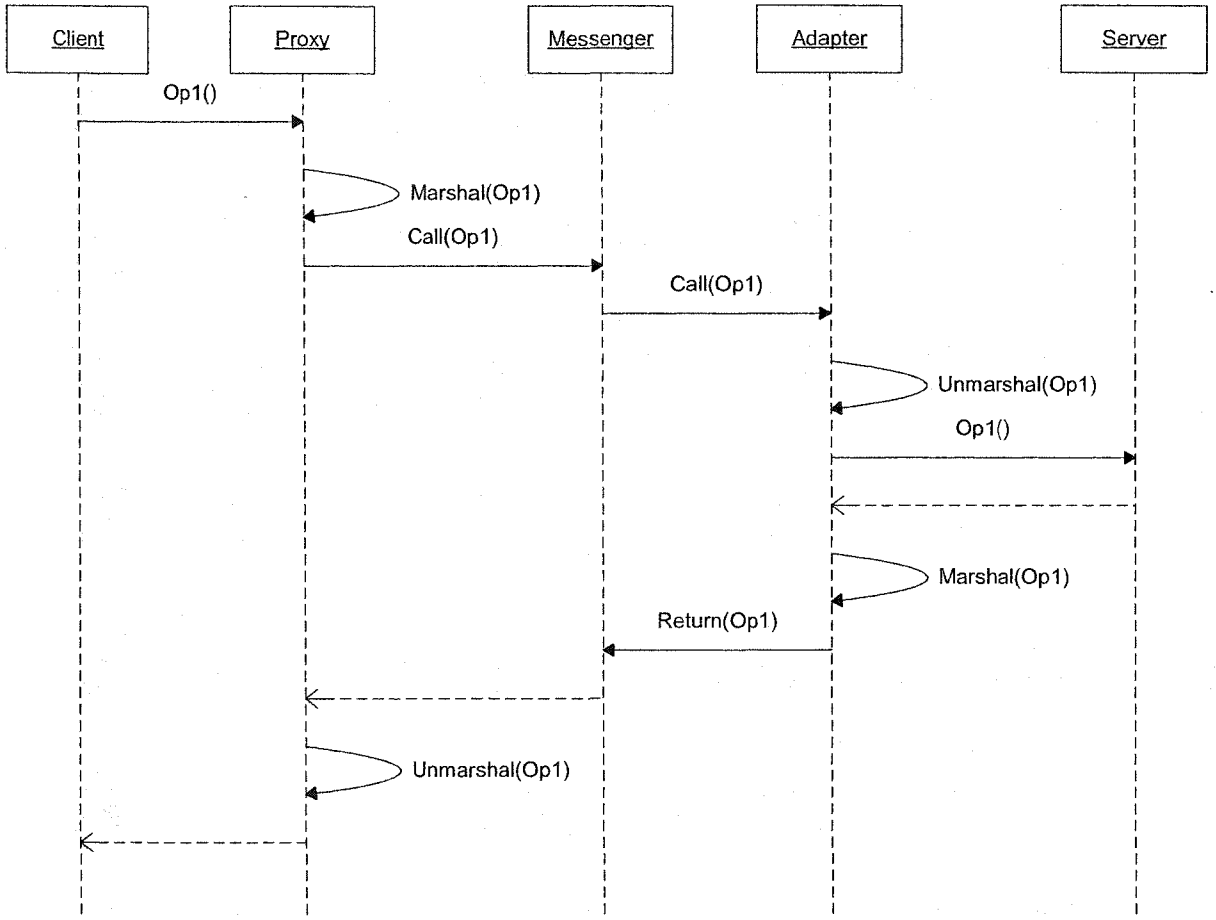

Figure 6.1 CMessenger -- diagramme de séquences externes

Il va de soi que la coopération de plusieurs classes et de plusieurs modules logiciels est nécessaire à la réalisation de l'appel d'une opération de l'interface d'un objet distant.

Les prochaines sections décriront les entités de base qui implémentent le protocole, leurs rôles, leurs responsabilités ainsi que leurs relations.

6.1 Messenger

Le *Messenger* (classe *CMessenger*) est l'entité responsable de la transmission et de la réception des messages DOOM. Il offre au *Proxy* et à l'*Adapter* une interface permettant l'échange d'opérations sérialisées à travers les frontières d'un réseau de nœuds distants. Il est comparable à l'ORB d'un système CORBA. La classe *CMessenger* est complexe et composée de plusieurs autres classes et méthodes. Elle partage cependant la grande majorité de ses responsabilités avec un ensemble restreint de quatre classes. Ce sont ces classes qui sont responsables de la transmission et de la réception des messages. Elles suivent les règles établies au niveau du

protocole DOOM. La présente description du *Messenger* se limite à la classe *CMessenger* elle-même, ainsi qu'à ses quatre classes principales.

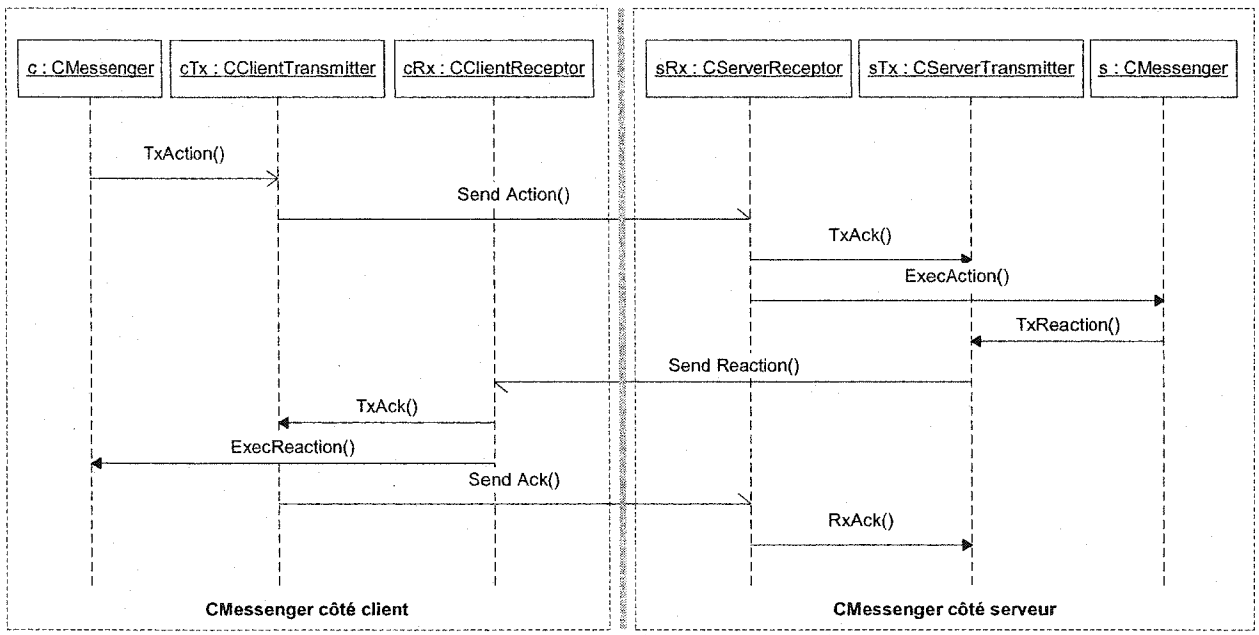

Figure 6.2 CMessenger -- diagramme de séquences internes

La figure précédente illustre la séquence des messages de base qui sont échangés entre les entités internes du *Messenger*. Les messages *Send* représentent la transmission d'un message entre les deux processus. Ce message est véhiculé à travers le réseau en partance du nœud émetteur vers le nœud récepteur où réside le serveur. Tous les autres messages sont formés d'appels d'opérations synchrones et locales au processus : les messages de type *TxAck* indiquent à l'objet de transmetteur qu'un message a été reçu et qu'il doit transmettre l'accusé de réception après le délai D_{ACK}; les messages de type *TxAction* et *TxReaction* demandent respectivement la transmission d'une action ou d'une réaction, tandis que les messages *ExecAction* et *ExecReaction* demande l'exécution d'opérations du serveur ou du client; finalement, les messages de type *RxAck* informent le transmetteur qu'un accusé de réception a été reçu.

La ligne grise au centre du diagramme représente la frontière délimitant l'espace adressable de chacun des processus. Pour qu'un message puisse traverser cette frontière, un *Socket* est requis. C'est à ce niveau que les règles de transmission du protocole DOOM interviennent. Le client se retrouve du côté gauche de la frontière, tandis que le serveur est du côté droit. Dans chacun des cas, le rôle de l'objet *Messenger* est qualifié soit avec le préfixe *Client*, soit avec le préfixe *Server*. Cela a pour unique but de mieux distinguer le comportement du *Messenger* en fonction du type d'objets qui utilise ses services. En réalité, la classe *CMessenger* n'est pas spécialisée en fonction du rôle joué, car elle supporte les deux rôles à la fois.

Lorsque le client fait appel à une opération du *Proxy*, ce dernier, après avoir sérialisé l'opération, fait appel à son tour au *Messenger* pour transmettre, le message et son action, vers le serveur qui réside sur un nœud distant. Dans ce cas, le *ClientTx* (*CClientTransmitter*) doit intervenir. À l'intérieur du processus opposé, le *ServerRx* (*CServerReceptor*) reçoit le message pour le

rediriger vers le *Messenger* qui se charge d'acheminer l'action vers l'*Adapter* associé. Une fois l'action exécutée, le message de réaction est transmis par le *ServerTx* (*CServerTransmitter*) vers le *ClientRx* (*CClientReceptor*). La réaction est ensuite acheminée jusqu'au client externe. C'est aussi le *ClientTx* qui transmet l'accusé de réception au *ServerRx* et qui conclut l'échange. Bien que la classe *CMessenger* soit la même, qu'elle soit du côté client ou du côté serveur, les classes de transmission et de réception sont différentes, car leur comportement est étroitement lié à leur rôle : c'est pourquoi chaque classe est distincte.

La Figure 6.3 présente les principales classes avec lesquelles la classe *CMessenger* interagit. Il est important de noter qu'il n'existe qu'un et un seul *Messenger* à l'intérieur d'un même processus. L'instance de *CMessenger* est un singleton auquel tous les *Proxy* et *Adapter* du processus se lient. La classe est aussi active[15], ce qui lui permet d'initier des flots d'exécutions basés sur des événements internes et externes tels les événements de temps. Cela lui permet d'offrir une exécution concurrente des appels en provenance d'objets différents, non synchronisés, et qui existent sur des nœuds distincts du réseau.

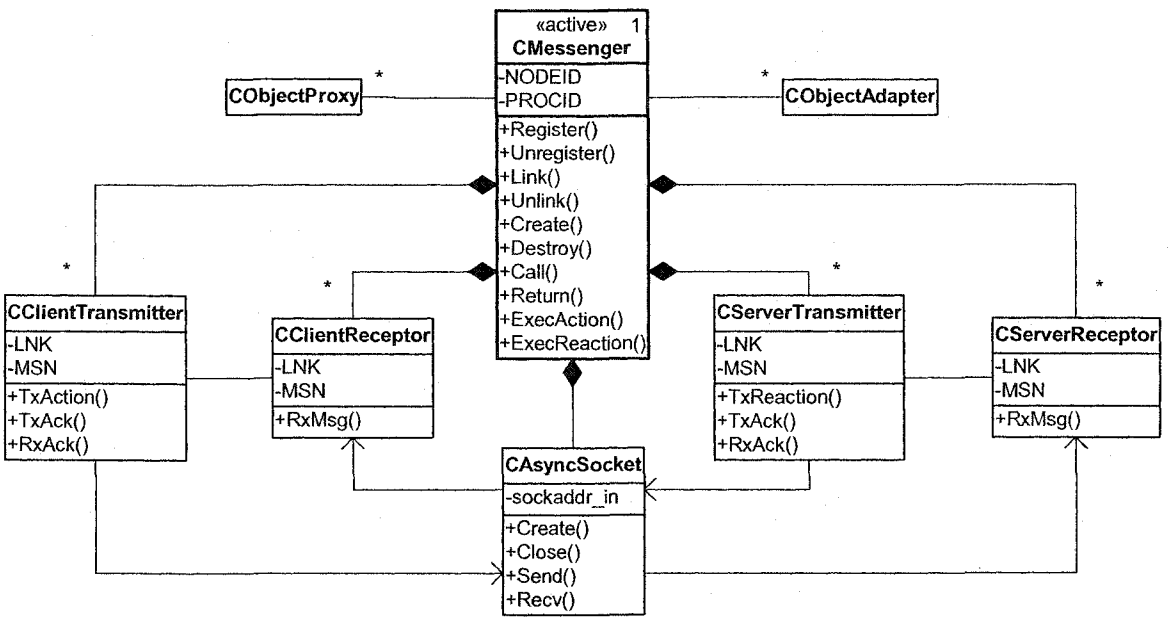

Figure 6.3 CMessenger -- diagramme statique

La classe mémorise les identificateurs de nœud et les identificateurs de processus. C'est ce qui permet au *Messenger* de se distinguer parmi les autres instances du réseau soit en créant un LinkID unique, pour chaque objet qu'il gère. Le *Messenger* est responsable de la localisation du serveur lorsqu'aucune adresse est spécifiée par le client.

Comme il est précisé au niveau du protocole DOOM, un seul *Socket* est assigné à la classe *CMessenger*. La classe doit alors assurer le démultiplexage des messages entrant vers le bon récepteur. Cette tâche est réalisée à l'aide du LNK inclus dans chacun des messages. Le nombre

[15] Voir la section 6.5 pour la description d'une classe active.

de *Proxy* ou d'*Adapter* lié au *Messenger* est limité par le protocole qui supporte jusqu'à un maximum de 2^{32} liens différents. Même aujourd'hui, la plupart des systèmes sont incapables de supporter un tel nombre d'instances. C'est pourquoi le nombre d'associations est généralement limité par l'implémentation. Pour chaque *Proxy* s'enregistrant auprès du *Messenger*, une paire d'instances des classes *CClientTransmitter* et *CClientReceptor* lui est associée. Le même principe s'applique à l'enregistrement d'un *Adapter*, mais cette fois les instances sont celles des classes *CServerTransmitter* et *CServerReceptor*. Chaque *Proxy* et *Adapter* possèdent donc leur propre instance de machine d'états, en rapport avec l'échange des messages, et ce, à l'intérieur du *Messenger*. Cette approche permet de simplifier grandement le code interne de la classe *CMessenger* et ne requiert qu'un léger surplus de mémoire puisque la structure de données, interne aux transmetteurs et aux récepteurs, est simple et légère. De plus, l'échange de messages entre deux objets est assurément indépendant des autres objets liés au *Messenger*.

Les opérations de l'interface de la classe *CMessenger* calquent les services du protocole DOOM. Par exemple, lorsqu'un client construit un objet distant, il construit en réalité le *Proxy*. Ce même *Proxy* fait alors appel à l'opération *Create()* du *Messenger* qui transmet aussitôt le message de création vers le nœud où réside le serveur réel. Par contre, avant de pouvoir appeler ces opérations, chaque *Proxy* doit s'enregistrer auprès du *Messenger* via l'opération *Register()*. La même règle s'applique à l'*Adapter* pour qu'il puisse être retracé par le *Messenger* lorsqu'un message lui est destiné. Les deux opérations *ExecAction()* et *ExecReaction()* sont particulières et ne peuvent être appelées que par les récepteurs. Par exemple, lorsqu'un *ServerReceptor* décode une nouvelle action devant être exécuter par l'objet auquel il est associé, il fait appel à l'opération *ExecAction()*. Pour ce qui est du *ClientReceptor*, celui-ci fait appel à l'opération *ExecReaction()* pour transmettre au client le retour de l'opération.

Les deux prochaines sections décrivent le comportement des classes de transmission et de réception formant le cœur de la classe *CMessenger*.

6.1.1 Transmitter

Le transmetteur est l'entité responsable d'émettre les messages en direction du récepteur se trouvant à l'autre bout du lien. Les responsabilités du transmetteur sont simples. Premièrement, il gère l'état de la transmission d'un message en corrélation avec le lien existant entre deux objets à travers un réseau non fiable. Il s'assure donc de retransmettre chaque message jusqu'à ce qu'il lui soit indiqué que le message a été reçu. Deuxièmement, il veille à ce que les messages soient émis selon les règles du protocole DOOM définies à la section 5.10. Il doit aussi former l'en-tête du message en y ajoutant les bonnes valeurs.

À l'intérieur de la classe *CMessenger*, il existe deux types de transmetteurs : la classe *CClientTransmitter* utilisée du côté client, et la classe *CServerTransmitter* utilisée du côté serveur. Ces deux classes de transmission sont différentes parce que les messages transmis, dépendamment du rôle joué, ne sont pas les mêmes. De plus, les états du protocole diffèrent eux aussi en fonction du rôle.

La classe *CClientTransmitter* possède une interface simple avec seulement trois opérations importantes à décrire. Tout d'abord, l'opération *TxAction()* est utilisée par le *Messenger* pour initier la transmission d'un message vers le serveur. Celle-ci prend en paramètre un *marshaler* et

retourne immédiatement la première transmission du message effectuée. Le *Messenger* peut alors continuer à exécuter d'autres tâches en attendant la réaction. Ensuite, l'opération *RxAck()* est appelée par le récepteur dès que l'accusé de réception est reçu. Pour le transmetteur, cela signifie qu'il doit cesser la retransmission du message et passer à l'état *Waiting*. Finalement, l'opération *TxAck()* est aussi appelée par le récepteur, et ce, lorsque le message de réaction est reçu. Si le transmetteur ne reçoit aucun autre message à émettre, à l'intérieur d'un délai de DACK millisecondes, il envoie l'accusé de réception. À partir du moment où l'opération *TxAck()* est appelée, il est possible que l'opération *TxAction()* soit appelée à nouveau par le *Messenger*.

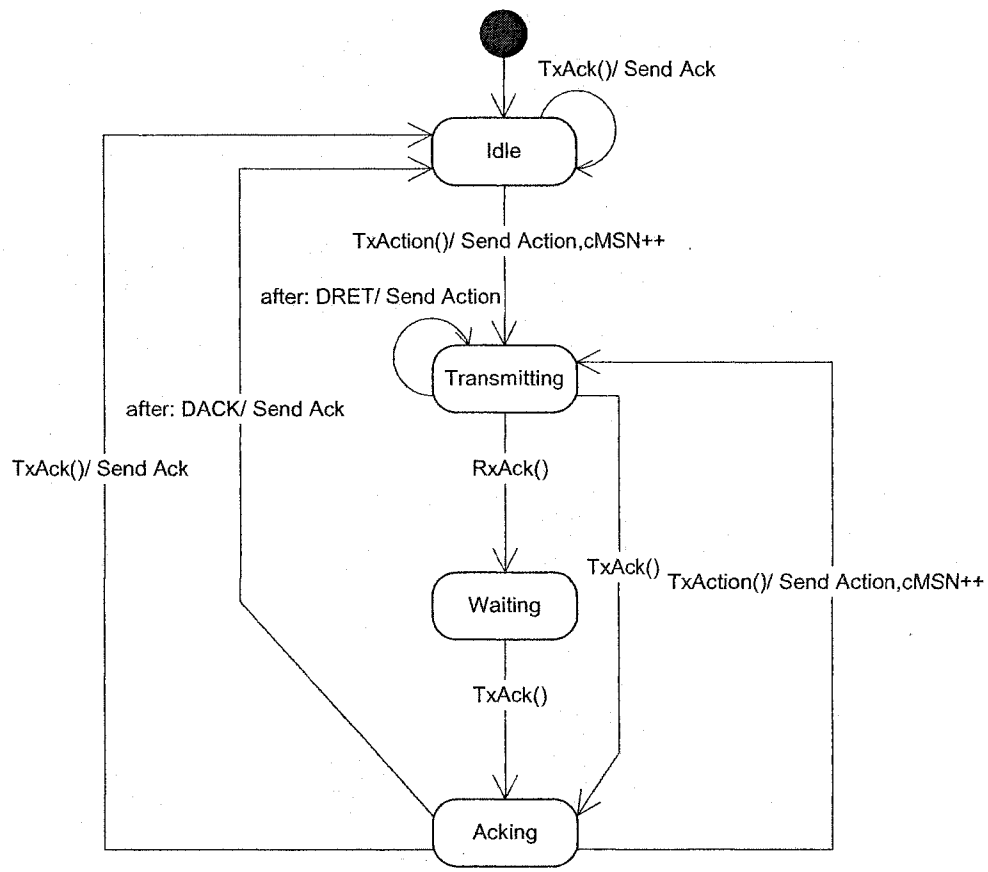

Figure 6.4 CClientTransmitter -- diagramme d'états

La Figure 6.4 décrit le comportement de la classe *CClientTransmitter*. Ce diagramme d'états illustre les règles émises à la section 5.10 se rapportant à la transmission d'un message contenant une action. Lorsque le client appelle une opération du serveur, les états du *CClientTransmitter* s'enchaînent généralement pour passer de *Idle* à *Transmitting*, de *Transmitting* à *Waiting*, de *Waiting* à *Acking* pour finalement retourner à *Idle*. Il se peut, qu'en fonction des délais, les états *Idle* et *Waiting* soient omis, mais pour chaque message, les états *Transmitting* et *Acking* sont atteints. Les transitions sont initiées par l'appel des opérations *TxAction()*, *RxAck()*, *TxAck()* ou par l'expiration d'un délai. La signification du délai de retransmission DRET et du délai d'accusé de réception DACK se retrouvent à l'intérieur du protocole.

Comme la classe *CClientTransmitter*, la classe *CServerTransmitter* possède, elle aussi, une interface simple avec trois opérations importantes. Tout d'abord, l'opération *TxAck()* est appelée par le récepteur, et ce, lorsqu'un message d'action est reçu. À partir du moment où l'opération *TxAck()* est appelée, il est possible que l'opération *TxReaction()* soit appelée par le *Messenger*. Lorsque le délai de DACK millisecondes est écoulé, un accusé de réception est émis. Ensuite, une fois l'action exécutée par le serveur, le *Messenger* indique au transmetteur d'émettre un message de réaction via l'opération *TxReaction()*. Cette dernière prend en paramètre un *marshaler* et retourne immédiatement la première transmission du message effectuée. Si l'appel à l'opération *TxReaction()* se fait à l'intérieur du délai DACK, aucun accusé de réception n'est envoyé. Finalement, l'opération *RxAck()* est appelée par le récepteur, et ce, lorsque l'accusé de réception en provenance du client est reçu. Pour le transmetteur, cela signifie de cesser la retransmission du message et de passer à l'état *Idle* en attente d'une autre action à exécuter.

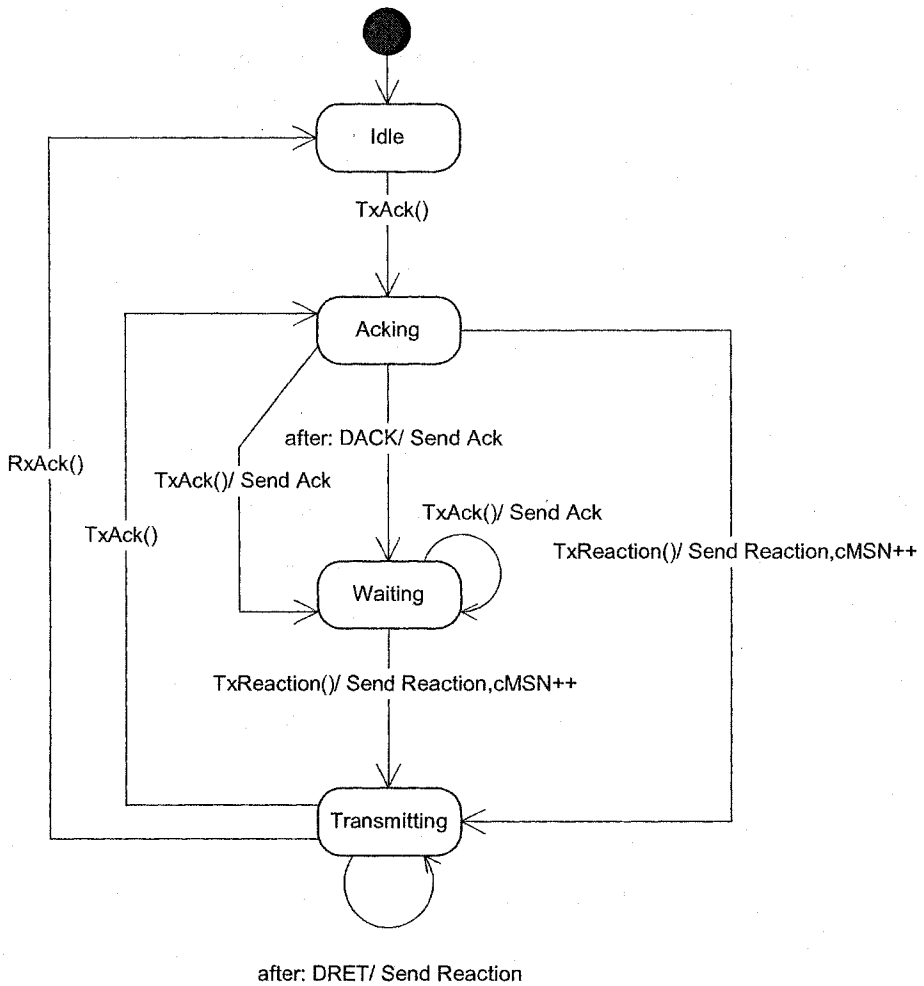

Figure 6.5 CServerTransmitter -- diagramme d'états

La Figure 6.5 décrit le comportement de la classe *CServerTransmitter*. Comme précédemment, ce diagramme d'états illustre les règles définies par le protocole DOOM se rapportant à la transmission d'un message contenant une réaction. Lorsque le client appelle une opération du serveur, cela génère l'exécution d'une action. Pour transmettre le retour de l'opération au client, à

l'intérieur d'un message de réaction, les états du *CServerTransmitter* s'enchaînent généralement pour passer de *Idle* à *Acking*, de *Acking* à *Waiting*, de *Waiting* à *Transmitting* pour finalement retourner à *Idle*. Il se peut qu'en fonction des délais, les états *Idle* et *Waiting* soient omis, mais pour chaque message les états *Transmitting* et *Acking* sont atteints. Ici aussi, les transitions sont initiées par l'appel des opérations *TxReaction()*, *RxAck()*, *TxAck()* ou par l'expiration d'un délai DRET ou DACK.

6.1.2 Receptor

Le récepteur est l'entité responsable de la réception des messages en provenance du transmetteur se trouvant à l'autre bout du lien. Les responsabilités du récepteur sont simples. Premièrement, il gère l'état de la réception d'un message, en corrélation avec le lien existant entre deux objets, et ce, à travers un réseau non fiable. Il s'assure donc de rapporter à un plus haut niveau chacun des messages, une seule fois : les messages dupliqués sont ignorés. Deuxièmement, il veille à ce que les messages reçus respectent les règles du protocole DOOM définies à la section 5.10 : il décortique l'en-tête des messages et s'assure que les valeurs sont correctes.

À l'intérieur de la classe *CMessenger*, il existe deux types de récepteurs : la classe *CClientReceptor* utilisée du côté client, et la classe *CServerReceptor* utilisée du côté serveur. Ces deux classes de réception sont différentes parce que les messages reçus, dépendamment du rôle joué, ne sont pas les mêmes. De plus, comme pour les transmetteurs, les états du protocole diffèrent en fonction du rôle.

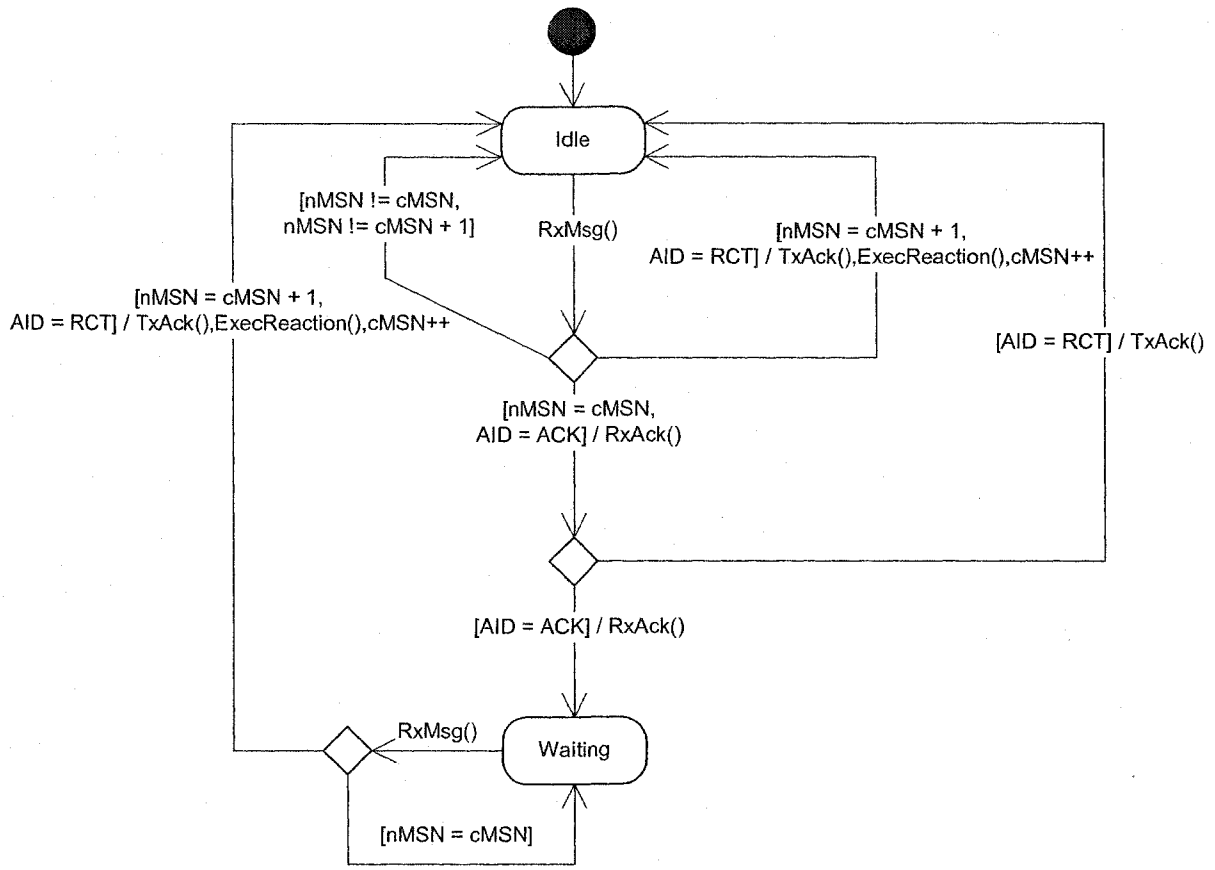

Figure 6.6 CClientReceptor -- diagramme d'états

Les deux classes de réception possèdent une seule opération à leur interface. L'opération *RxMsg()* est appelée par la couche de transport lorsqu'un message est reçu et qu'il est destiné à l'objet associé au récepteur. Si l'en-tête du message est valide et qu'il ne s'agit pas d'un duplicata, le récepteur passe l'action ou la réaction à un plus haut niveau, via le *Messenger*. Le récepteur possède le flot de contrôle uniquement lorsqu'un message lui est remis pour fin de traitement : il est totalement passif. Peu importe l'état dans lequel se trouve le récepteur, il est toujours possible que l'opération *RxMsg()* soit appelée.

La Figure 6.6 décrit le comportement de la classe *CClientReceptor*. Ce diagramme d'états illustre les règles du protocole pour la réception d'un message de réaction. Lorsque le serveur retourne le résultat de l'opération au client, les états du *CClientReceptor* s'enchaînent généralement pour passer de *Idle* à *Waiting* et de *Waiting* à *Idle*. Chaque transition est initiée par l'appel de l'opération *RxMsg()*. L'état *Waiting* signifie que le *CClientReceptor* est en attente d'une réaction. La transition vers cet état est toujours initiée par la réception d'un accusé de réception. Lors de cette transition, le récepteur indique au *CClientTransmitter*, via l'appel de l'opération *RxAck()*, que le message d'action est reçu du côté serveur. Au moment où le message de réaction est reçu, le récepteur en informe le transmetteur via un appel à *TxAck()*. Il appelle aussi l'opération *ExecReaction()* du *Messenger* pour que ce dernier transmette le retour de l'opération au client, via son *Proxy*. Il est aussi possible pour le *CClientReceptor* de recevoir, à partir de l'état *Idle*, un message de réaction; dans ce cas les opérations du transmetteur et du *Messenger* sont aussi

appelées par le récepteur, et ce, avant que ce dernier ne retourne à l'état *Idle*. Dans tous les cas, il est important de remarquer l'incrémentation du MSN lorsque le récepteur a bien reçu la réaction et qu'il passe à la suivante. À partir de ce moment, tous les messages dont le numéro de séquence est inférieur au dernier numéro plus un, sont ignorés.

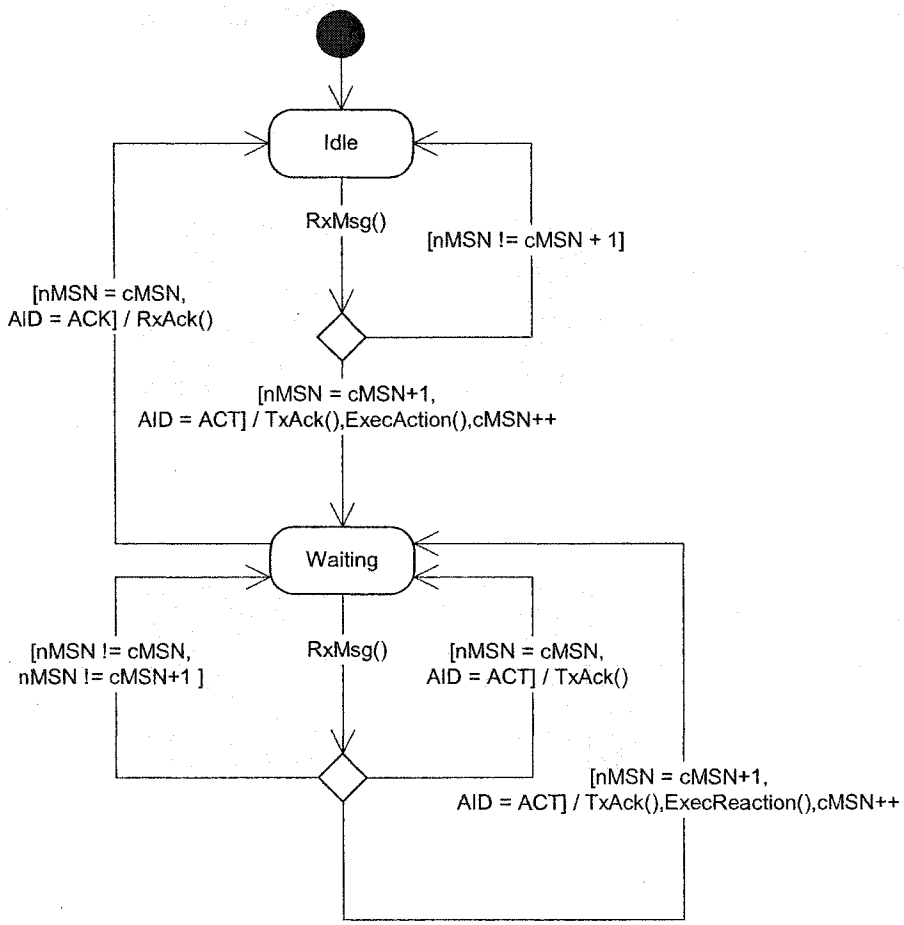

Figure 6.7 CServerReceptor -- diagramme d'états

La Figure 6.7 décrit le comportement de la classe *CServerReceptor*. Ce diagramme d'états illustre les règles du protocole pour la réception d'un message d'action. Suite à la réception d'une action en provenance du client, les états du *CServerReceptor* s'enchaînent généralement pour passer de *Idle* à *Waiting* et de *Waiting* à *Idle*. Chaque transition est initiée par l'appel de l'opération *RxMsg()*. L'état *Waiting* signifie que le *CServerReceptor* est en attente d'un accusé de réception faisant suite à la transmission de la réaction en provenance du serveur. La transition vers cet état est toujours initiée par la réception d'une action. La nouvelle action est signifiée à la fois au *CServerTransmitter* via l'opération *TxAck()* et au *Messenger* via l'opération *ExecAction()*. Si le message reçu dans l'état *Idle* ne correspond pas à une action, il n'est pas traité. Il est aussi possible, pour le récepteur, de recevoir une nouvelle action à l'intérieur de l'état *Waiting*. Dans ce cas, comme précédemment, il en informe ses collaborateurs et il retourne à l'état *Waiting*. Par contre, l'accusé de réception le fait transiter vers l'état *Idle*. Dans tous les cas, il est important de remarquer l'incrémentation du MSN lorsque le récepteur a bien reçu une nouvelle action et qu'il

se met en attente de la suivante. À partir de ce moment, tous les messages dont le numéro de séquence est inférieur au dernier numéro plus un, sont ignorés.

6.2 Proxy

Du point de vue du client, l'interface d'un objet est appelée par un pointeur. Ce pointeur doit être local au processus client pour des raisons évidentes : lorsque le serveur est à l'intérieur du même processus que le client, le pointeur référence l'objet natif et l'appel d'une de ses opérations est direct, par contre, lorsque le serveur est à l'extérieur du processus client, le pointeur référence un objet de type *Proxy* et l'appel d'une opération du serveur devient indirect. Autrement dit, le client accède toujours à l'espace mémoire local de son propre processus. Pour que le client puisse construire ou établir un lien avec le serveur, l'adresse de ce dernier doit être passée en paramètre au *Proxy*, ou bien, l'adresse de l'objet peut être déterminée automatiquement par le service de localisation.

Le *Proxy* présente l'interface du serveur au client à l'intérieur de l'espace adressable par ce dernier. Pour ce faire, il doit résider à l'intérieur du même processus que le client. Il a pour rôle d'abstraire la localisation du serveur. Il réalise la même interface que le serveur auquel le client veut accéder. Lorsque le client appelle une opération, le *Proxy* sérialise, à la l'aide d'un *Marshaler*, l'opération et ses paramètres. Cela s'effectue de façon à ce que celle-ci puisse être transmise et reconstruite à l'intérieur de l'espace mémoire du processus où réside le serveur.

L'entité générique *Proxy* est implémenté à l'intérieur de la classe abstraite *CObjectProxy*. Cette classe implémente les méthodes d'accès au *Messenger* nécessaires à tous les objets *Proxy* spécialisés. Elle est passive et ne fait intervenir aucun *thread* d'exécution. La synchronisation avec le serveur est assurée par le *Messenger*.

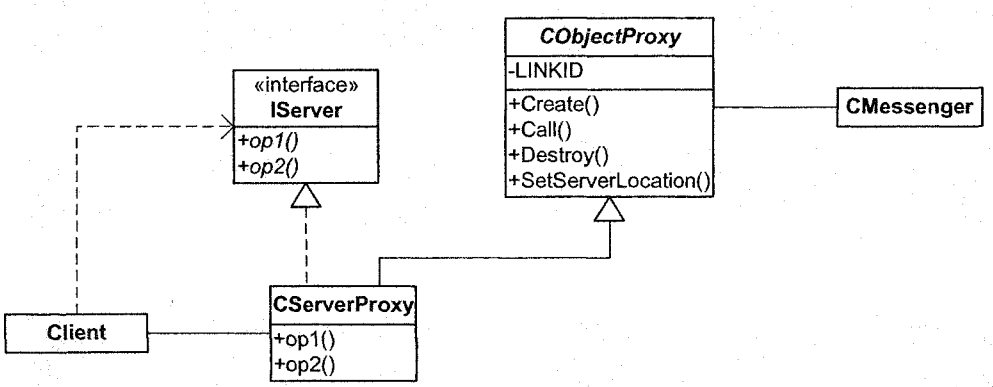

Figure 6.8 CObjectProxy -- diagramme statique[16]

La Figure 6.8 illustre les relations existant entre la classe *CObjectProxy* et un client. En fait, la classe *CObjectProxy* n'est jamais directement utilisée par le client. Pour que l'abstraction soit

[16] Les attributs et les opérations de la classe *CObjectProxy* présentés à l'intérieur de la Figure 6.8 ne sont pas complets. L'implémentation détaillée est disponible en annexe 1.

complète, le client connaît seulement l'interface du serveur. Il possède, via un pointeur d'interface, une association avec une classe spécialisée. Cette classe hérite de *CObjectProxy* et de l'interface spécifique au serveur. Cette relation est illustrée par l'interface *IServer* et par le *Proxy* spécialisé *CServerProxy*. C'est lors de l'appel à l'une des opérations de l'interface par le client, que le *Proxy* intervient. Une fois le *marshaling* de l'opération effectué, le message est transmis grâce aux méthodes de la classe *CObjectProxy* faisant appel aux services du *Messenger*.

6.3 Adapter

À cause de la grande variété d'objets existant, il est impossible pour le *Messenger* de s'adapter à toutes leurs interfaces. Pour arriver à contourner ce problème, il est obligatoire de recourir au principe d'adaptateur. L'adaptateur offre au *Messenger* l'accès aux opérations du serveur. Il est principalement responsable de l'adaptation de l'interface d'un objet au protocole DOOM. Via l'*Object Adapter*, il est possible pour le *Messenger* de communiquer avec tous les types d'objets. Il existe un *Object Adapter* pour chaque classe d'objets. Les services qui sont offerts par l'*Object Adapter* incluent : la création et la destruction de l'objet serveur; l'appel des opérations; la gestion de la sécurité des interractions; l'enregistrement de la classe d'objet auprès du *Messenger*. L'adaptateur réside à l'intérieur du processus serveur. Il accomplit le travail inverse du *Proxy*.

C'est la classe *CObjectAdapter* qui implémente les services de base de l'*Object Adapter*. L'un de ses rôles est de définir l'interface commune entre le *Messenger* et tous les adaptateurs. Elle est donc abstraite et purement virtuelle. Les opérations *Create()*, *Call()* et *Destroy()* doivent être implémentées par les classes enfants. Ces opérations génériques sont appelées par le *Messenger*, en réponse aux messages reçus du côté client. C'est à la classe spécialisée que revient la responsabilité d'appeler la bonne opération du serveur en lui passant les valeurs extraites du flot d'octets. La sérialisation des valeurs de retour de l'opération, pour sa transmission vers le client, est aussi sous sa responsabilité. Les opérations *Register()* et *Unregister()*, qui ne sont pas abstraites, servent à gérer l'enregistrement de la classe rendue disponible à travers le réseau. L'identificateur de la classe (CLSID) et la référence à l'adaptateur (classe spécialisée) sont donnés au *Messenger*. Ainsi, lorsqu'un message de construction arrive en provenance du réseau pour une classe d'objet X, c'est à l'adaptateur enregistré avec ce numéro que le *Messenger* s'adresse.

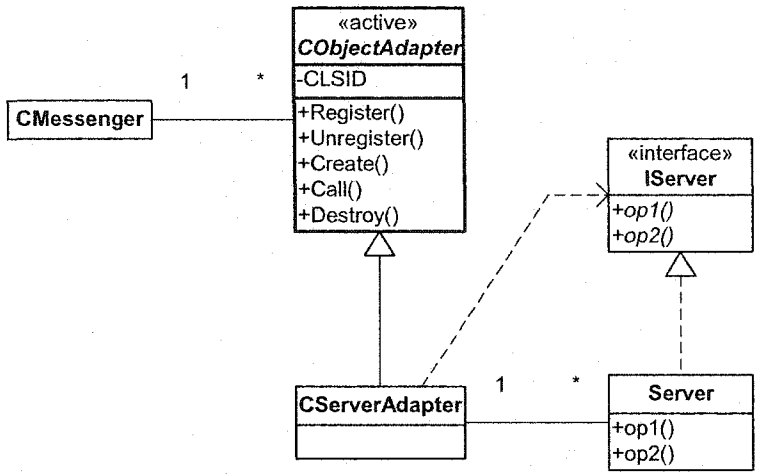

Figure 6.9 CObjectAdapter -- diagramme statique[17]

Comme le démontre la Figure 6.9, la classe *CObjectAdapter* est active, c'est-à-dire qu'elle possède son propre *thread* d'exécution. Bien que cette classe ne fasse que l'adaptation entre le *Messenger* et le serveur, il est primordial que le flot d'exécution du *Messenger* soit indépendant de celui de l'adaptateur.

Le *thread* principal d'exécution, celui du *Messenger*, ne peut pas se permettre de bloquer sur l'appel de l'opération d'un quelconque objet. Puisqu'il gère plusieurs liens entre plusieurs objets différents, son exécution n'est pas dédiée et il doit toujours être à l'écoute des messages en provenance du réseau, ou de l'interne. Par exemple, si le Messenger est bloqué sur l'appel d'une opération trop longtemps, il sera incapable de transmettre l'accusé de réception au client, ce qui pourrait causer la brisure du lien.

Selon le comportement du serveur, l'adaptateur peut être implémenté de deux façons. Premièrement, si le serveur est simple et qu'aucune de ses opérations ne risque de bloquer, il est possible d'utiliser un seul *thread* au niveau de l'adaptateur pour toutes les instances de serveur. Autrement, la deuxième solution s'impose et il est préférable d'avoir un *thread* pour chaque instance de serveur afin d'éviter que l'appel d'une opération d'une instance n'interfère avec les autres instances. Il est à noter que chaque solution possède des avantages et des inconvénients, et qu'il est important de choisir l'avenue qui répond le mieux aux besoins du système.

L'adaptateur offre une certaine forme de sécurité. Son implémentation s'assure que le format de chaque opération du serveur est respecté. Il appelle une opération uniquement lorsqu'aucun paramètre n'est erroné ou manquant : le respect de cette condition est primordial, car autrement, il devient possible d'endommager le système par erreur ou encore, d'attaquer volontairement le système.

[17] Les attributs et les opérations de la classe *CObjectAdapter* présentés à l'intérieur de la Figure 6.9 ne sont pas complets. L'implémentation détaillée est disponible en annexe 1.

6.4 Marshaler

L'application principale de DOES est de veiller à la distribution des objets sur un réseau hétérogène : il est donc avantageux d'utiliser une méthode de *marshaling* commune à toutes les implémentations. Même si les opérations peuvent être sérialisées suivant une convention propriétaire établie entre le client et l'objet serveur, il est avantageux de ne pas le faire ainsi, et ce, afin de permettre une plus grande homogénéité des systèmes et afin de rendre tous les objets interopérables dans n'importe quel type de configurations réseau. C'est pourquoi l'implémentation du *marshaling* respecte les règles d'encodages BDER établies à la section5.8.5, et ce, pour chacun des types de base du C++.

Les fonctions de *marshaling* sont réalisées à l'intérieur d'une hiérarchie simple de trois classes (*CInprocMarshaler*, *COutprocMarshaler* et *CRootMarshaler*) et d'une interface (*IMarshaler*). Le *marshaler* est l'objet responsable de la sérialisation et de l'extraction d'une opération à l'intérieur d'un flot d'octets. En fait, le terme *marshaler* désigne une instance de la classe *CInprocMarshaler* ou de la classe *COutprocMarshaler*. Quant à la classe *CRootMarshaler*, qui est abstraite, elle ne peut pas être utilisée directement : seules les opérations communes aux deux modes de *marshaling* (*inproc* et *outproc*) y sont implémentées.

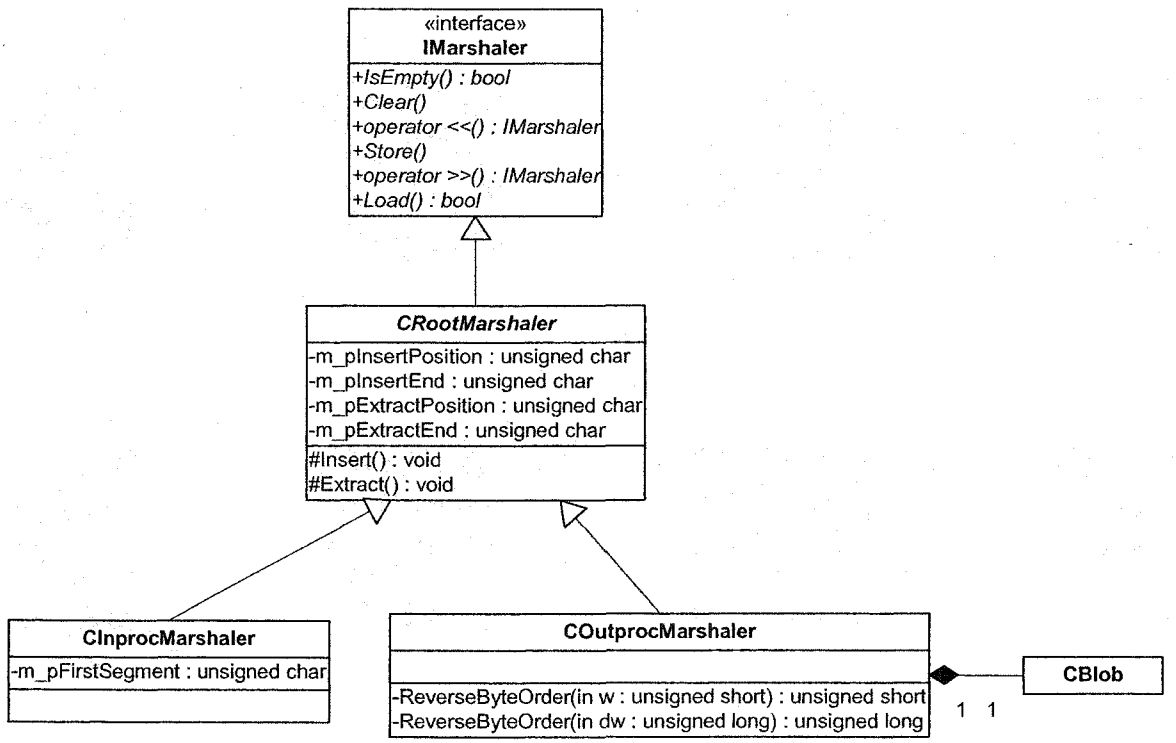

Figure 6.10 Marshaler -- diagramme statique

Pour ce qui est de l'interface *IMarshaler*, elle sert à abstraire le mode de *marshaling* ou encore, la véritable nature de l'instance. De cette façon, il est possible de réduire la complexité des opérateurs de *marshaling* devant être implémentés au sein des types construits, comme les classes et les structures.

Le *marshaler* permet le *marshaling* d'une opération selon deux modes. Le premier est appelé *inproc* et le second *outproc*. Ces deux modes de *marshaling* sont respectivement assumés par les classes *CInprocMarshaler* et *COutprocMarshaler*. Lorsque les objets client et serveur ne sont pas à l'intérieur du même processus, c'est le mode *outproc* qui est de mise afin d'assurer la communication inter-processus. Dans le cas contraire, le mode *inproc* qui est plus rapide, devrait être utilisé. Il n'est pas nécessaire, pour les objets qui résident à l'intérieur du même processus, de respecter toutes les règles d'encodages BDER pour réussir à communiquer entre eux. Par exemple, le *marshaler* en mode *inproc* ne gère pas l'ordre des octets.

Attention, un pointeur ou une référence à un objet ne doit jamais être sérialisé en mode *outproc*. Dans ce mode, c'est l'objet lui-même qui doit être sérialisé puisque l'objet distant n'a pas accès à l'espace mémoire du client, et vice versa. Cependant, cette contrainte n'est pas présente lorsque les objets partagent le même processus et le même espace mémoire.

La méthode d'insertion et d'extraction utilisée par le *marshaler* est de type FIFO. Les types de bases du C++ peuvent tous être sérialisés de façon indépendante à l'intérieur d'un flot d'octets et l'ordre dans lequel ils sont sérialisés n'est pas géré par le *marshaler*. Par contre, une fois l'ordre établi lors de la sérialisation, il devient très important de respecter ce même ordre lors de l'extraction, c'est-à-dire que le premier argument à être inséré doit être le premier à être extrait du *marshaler*. De plus, il est très important de respecter l'ordre de *marshaling* des paramètres d'une opération qui est prescrit par les règles d'encodages BDER, le *Proxy* et l'*Adapter* étant responsables d'assurer cet ordre. Le type de la donnée n'est pas inséré à l'intérieur du flot d'octets, sauf en mode de vérification : donc, il est impossible de rechercher une donnée de façon aléatoire (*randum access*) sans parcourir entièrement le flot d'octets ou encore de modifier les adresses d'insertion et d'extraction du *marshaler*. Le *marshaling* et le *unmarshaling* d'une opération doivent être déterminés, complets et séquentiels : ces contraintes permettent de maximiser les performances du *marshaler*.

La classe *CRootMarshaler* utilise quatre pointeurs : les deux premiers pointeurs, *m_pInsertPosition* et *m_pInsertEnd*, servant à la sérialisation; les deux autres pointeurs, *m_pExtractPosition* et *m_pExtractEnd*, servant à l'extraction. À la construction, les deux pointeurs de « position » sont initialisés avec l'adresse du début du segment mémoire, tandis que les deux autres pointeurs indiquent la « fin » du segment, moins l'espace nécessaire à la sauvegarde de l'adresse du segment suivant. La donnée sérialisée est donc insérée à l'endroit où pointe *m_pInsertPosition* et ensuite, la valeur du pointeur est augmentée en fonction de la grandeur de la donnée. Cependant, si le segment mémoire ne contient pas suffisamment d'espace libre pour insérer entièrement la donnée (*m_pInsertEnd* - *m_pInsertPosition* < ***sizeof***(*data*)), une partie de cette donnée sera insérée dans l'espace restant, tandis que l'autre partie le sera à l'intérieur du prochain segment mémoire automatiquement alloué. L'adresse de ce nouveau segment alloué est sauvegardée à la fin du segment présent et les pointeurs d'insertions sont repositionnés en vue de continuer la sérialisation, et ce, à l'intérieur de ce nouveau segment mémoire. Quant à l'extraction des données du *marshaler*, elle fonctionne suivant le même principe : chaque fois qu'une donnée est extraite, elle l'est à partir de l'endroit où pointe *m_pExtractPosition* et cet endroit est ensuite modifié pour pointer à l'adresse de la prochaine donnée. Lorsque l'adresse pointée par *m_pExtractEnd* est atteint, les pointeurs d'extractions sont repositionnés en fonction de l'adresse du début du segment suivant se trouvant à l'endroit pointé

par la valeur de **m_pExtractEnd*; autrement dit, la valeur obtenue par « déréférenciation » de m_*pExtractEnd* correspond à l'adresse du début du segment suivant.

Le flot d'octets est représenté à l'intérieur du *marshaler* par un ou des tableaux de *unsigned char*. En mode *inproc*, le flot peut être fragmenté sur plusieurs segments de mémoire, ce qui n'est pas le cas en mode *outproc*.

Afin d'optimiser la sérialisation en mode *inproc*, la classe *CInprocMarshaler* utilise une liste simplement chaînée de segments de mémoire de longueur fixe qu'elle alloue au besoin. Ces segments sont relâchés seulement lorsque le *marshaler* est détruit. Donc, il est possible d'utiliser plusieurs segments de mémoire en mode *inproc* parce que le même espace mémoire est partagé entre les différents objets d'un même processus, et parce que la liste de segments contenant le flot d'octets peut être échangée facilement à l'aide d'une adresse mémoire. Il est à souligner que cette structure permet une allocation maximale, sans fragmentation, basée sur l'algorithme de gestion mémoire décrite à la section 6.6.

La figure suivante illustre l'interaction qui existe entre les différents pointeurs de *CRootMarshaler* et des segments de mémoire alloués par *CInprocMarshaler*. Le pointeur *m_pFirstSegment*, quant à lui, fait partie de *CInprocMarshaler* et il marque le premier segment de la liste. Les derniers octets du segment servent à sauvegarder l'adresse du début du segment suivant. De cette façon, les données de la liste n'occupent pas d'espace mémoire superflu et l'accès en est très rapide.

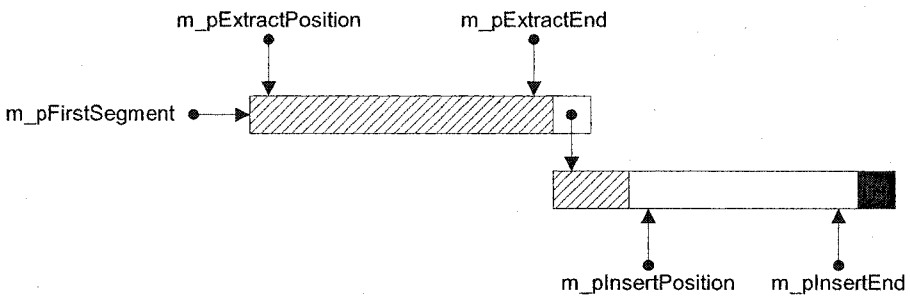

Figure 6.11 Exemple de segments de mémoire

Contrairement à *CInprocMarshaler*, la classe *COutprocMarshaler* utilise un seul segment de mémoire contiguë, appelé *blob* pour *Binary Large Object*, et qui est alloué à l'aide de la classe *CBlob*. Puisque le contenu du *marshaler* en mode *outproc* est destiné au transport réseau, il est impossible d'utiliser des segments de mémoire non contigus. Il est aussi plus rapide d'utiliser un *blob* qui peut être échangé directement avec le *Socket*. La sérialisation et l'extraction sont quand même réalisées par la classe mère *CRootMarshaler*. Dans ce mode, les pointeurs d'insertion et d'extraction pointent simplement sur le segment de mémoire qui est réservé par le *blob* lors de la construction d'une instance de *COutprocMarshaler*.

6.4.1 Opérateurs de *marshaling*

Des opérateurs d'insertion et d'extraction existent pour chaque type de donnée de base du C++. Ces opérateurs sont simples et efficaces. Ils sont tous implémentés au niveau de la classe *CRootMarshaler*. Par exemple, les opérateurs suivants se chargent du *marshaling* du type *bool* :

```
IMarshaler& operator<< (IN bool data);
IMarshaler& operator>> (OUT bool& data);
```

Les opérations *Store()* et *Load()* servent respectivement à sérialiser et à extraire une valeur de type non défini et de longueur variable :

```
void Store(IN const void* pData, IN WORD wSize);
bool Load(OUT void* pData, IN WORD wCapacity, OUT WORD& rwSize);
```

Il est aussi possible de sérialiser un pointeur à l'intérieur d'un *marshaler* de type *CInprocMarshaler*, grâce à des opérateurs génériques. Ces opérateurs acceptent toutes les sortes de pointeurs allant du type *void** à la référence d'une classe. Cela est rendu possible par les fonctions suivantes :

```
template <class _Type>
IMarshaler& operator<< (IN CInprocMarshaler& rM, IN const _Type* pPtr);

template <class _Type>
IMarshaler& operator>> (IN CInprocMarshaler& rM, OUT _Type*& pPtr);
```

Lorsqu'utilisée, seule la valeur du pointeur est sérialisée et non pas la valeur de l'objet vers lequel il pointe. Ces opérateurs ne doivent pas être utilisés en mode *outproc*. C'est pourquoi la classe *COutprocMarshaler* n'offre pas ces deux opérations. En mode *outproc*, une sérialisation complète de l'objet doit toujours avoir lieu.

À titre d'exemple, le *marshaling* de l'opération suivante :

```
void FooBar(IN bool b, IN int n);
```

s'exprime ainsi :

```
CMarshaler* pMarshaler = new CMarshaler;
*pMarshaler << b << n;
```

Le *unmarshaling* s'exprime ainsi :

```
bool b;
int n;

*pMarshaler >> b >> n;
```

Il est à remarquer que l'ordre d'extraction des paramètres respecte l'ordre d'insertion. Une « assertion », en mode de vérification, indique une erreur si le type du paramètre extrait ne correspond pas au type du paramètre qui fut inséré à cette position.

6.4.2 *Marshaling* de types construits

Le *marshaling* d'un type construit se fait par l'intermédiaire de deux opérations qui respectent les prototypes suivants : des opérateurs d'insertion et d'extraction sont créés pour chaque classe ou chaque structure devant être sérialisées par un *marshaler*; les classes, qui n'ont pas besoin d'être sérialisées, n'ont pas à définir les opérateurs de *marshaling*. Les formats des opérateurs sont :

```
IMarshaler& operator<< (IN IMarshaler& rMarshaler, IN const CGazou& rG);
IMarshaler& operator>> (IN IMarshaler& rMarshaler, OUT CGazou& rG);
```

Cela permet un *marshaling* homogène des types construits avec les types de base du C++. Par exemple, pour la classe *CGazou* suivante :

```
class CGazou
{
friend IMarshaler& operator<< (IN IMarshaler& rM, IN const CGazou& rG);
friend IMarshaler& operator>> (IN IMarshaler& rM, OUT CGazou& rG);

public:
   CGazou();
   virtual ~CGazou();

private:
    int m_nInt;
    long m_nlong;
};
```

Son *marshaling* est assuré par les fonctions suivantes :

```
IMarshaler& operator<< (IN IMarshaler& rM, IN const CGazou& rG)
{
   rM << rG.m_nInt << rG.m_nLong;
   return rM;
}

IMarshaler& operator>> (IN IMarshaler& rM, OUT CGazou& rG)
{
   rM >> rG.m_nInt >> rG.m_nLong;
   return rM;
}
```

Ainsi le *marshaling* de la classe *CGazou* peut s'effectuer facilement de cette façon :

```
CGazou gazou;
CMarshaler* pMarshaler = new CMarshaler;
*pMarshaler << gazou;
```

La même méthode peut être appliquée aux structures devant être sérialisées. Il est à noter que les opérateurs doivent être définis comme *friend* au niveau de la classe puisqu'ils doivent pouvoir accéder aux données membres qui sont privées. De plus, les opérateurs de *marshaling* des types de base doivent toujours être utilisés par les opérateurs de *marshaling* d'une structure ou d'une classe.

6.4.3 Ordre des octets

Comme défini par les règles d'encodage BDER établies à la section 5.8.5, le premier octet du flot (index 0) contient un booléen qui indique l'ordre d'encodage des octets des données ayant été sérialisées (voir 5.8.4). Ce drapeau ne fait pas partie des données sérialisées, mais il fait partie du flot d'octets contenant ces données. Basées sur la valeur du drapeau, les données à être encapsulées sont sérialisées par le *marshaler* selon les règles d'encodages définis par BDER. Les même règles s'appliquent lorsque les données sont extraites du *marshaler*. Par contre, ce drapeau est omis en mode *inproc* puisque dans ce mode, il n'y a pas d'ordonnancement des octets, car l'objet serveur utilise nécessairement le même ordre que le client, étant donné qu'ils sont à l'intérieur du même processus.

En mode *outproc*, l'ordre des octets est ajusté uniquement lorsque celui-ci ne correspond pas à la convention locale en vigueur. Lorsque le drapeau d'ordre indique qu'une convention différente fut utilisée lors de la sérialisation, l'ordre des octets est inversé au moment de l'extraction. Cette opération, qui est valide pour les types signés et non signés, est assurée par les méthodes suivantes :

```
unsigned short ReverseByteOrder(IN unsigned short w);
unsigned long ReverseByteOrder(IN unsigned long dw);
```

Lors de la sérialisation, l'ordre des octets n'est jamais modifié. L'ordre respecte la convention en vigueur au niveau du système d'exploitation et du CPU. Les opérateurs de sérialisations sont donc les mêmes, peu importe le mode de *marshaling*. Seuls les opérateurs d'extraction des types de base sensibles à l'ordre des octets sont surchargés par la classe *COutprocMarshaler*.

6.5 Threading

Le développement d'applications *mutlithread* est généralement complexe. En sus des problèmes de base que l'on retrouve à l'intérieur d'une application à exécution séquentielle, le *multithreading* et la concurrence introduisent d'autres facteurs de risque. Les problèmes les plus communs sont le *deadlock* et la corruption de données partagées. C'est pourquoi, la conception des flots d'exécution doit généralement être vérifiée méticuleusement afin d'éviter tous conflits entre les objets de l'application. Donc, le développement de ce type d'application est, plus souvent qu'autrement, coûteux en terme d'effort de vérification.

L'architecture interne de DOES repose sur des concepts de concurrence et de *multithreading*. De plus, l'échange de signaux et l'appel d'opérations synchrones et asynchrones doivent être supportés par DOES. Compte tenu des problèmes associés au développement de ce genre d'application, il est approprié de développer des mécanismes d'abstraction du *multithreading* et de la messagerie. Les buts principaux de ces mécanismes d'abstraction sont : d'éliminer la corruption de données due à une mauvaise synchronisation des accès; de fournir une hiérarchie de classes qui facilite la gestion et l'échange de messages asynchrones; de veiller à ce que la sémantique et l'intégrité de l'objet soient toujours protégées.

La solution adoptée repose sur une hiérarchie de classes abstraites. La structure proposée se rapproche du modèle *Single-Threaded Appartment* (STA) définie par COM [23] [24] [25] ainsi que sur les concepts d'objet actif, de message et de communication décrits par Grady Booch [26].

6.5.1 Objet actif

Un objet actif est un objet qui possède son propre *thread* d'exécution et qui peut initier une activité contrôlée. Une classe active est une classe dont les instances sont des objets actifs [27]. Plusieurs langages de programmation supportent le concept d'objet actif comme Java, Smalltalk, Ada, etc. En C++, ce concept est supporté à l'aide de différentes librairies de classes construites en fonction des services offerts par le système d'exploitation. Le concept d'objet actif est cependant peu répandu.

Dans un système purement séquentiel, il n'existe qu'un seul flot de contrôle. Cela signifie qu'une seule action peut s'exécuter à un moment donné. Lorsqu'un programme séquentiel débute, le flot de contrôle est démarré à la racine du programme et chacune des instructions est digérée dans l'ordre déterminé. Même si différents événements externes sont produits de façon aléatoire, ils seront traités séquentiellement. Par contre, dans un système où la concurrence est permise, il y a plus d'un flot de contrôle : cela signifie que plus d'une chose peut survenir au même instant, et que différents flots de contrôle sont donc démarrés à la racine de *threads* indépendants.

Une classe active représente un *thread* comme étant la racine d'un flot de contrôle indépendant et concurrent aux autres flots du système. Une classe active possède les même propriétés qu'une classe ordinaire, sauf son association à un flot de contrôle. Pour sa part, la classe ordinaire est qualifiée de passive.

6.5.2 Communication inter-objet

La communication entre objets se fait à l'aide de messages. Le message transporte de l'information, d'un objet vers un autre objet. La réception d'un message est considérée comme l'instance d'un événement et il en résulte généralement une activité du côté de l'objet récepteur. Le message se concrétise par l'envoi d'un signal en partance d'un objet (l'émetteur) vers un ou plusieurs objets (le ou les récepteurs). Il peut aussi s'agir de l'appel d'une opération d'un objet (le récepteur) par un autre objet (l'émetteur). Le message possède donc un émetteur, un récepteur et une action. L'émetteur a le rôle d'envoyer le message et le récepteur de le recevoir. Concrètement, l'action est un appel d'opération, ou un signal, ou une opération locale à l'émetteur ou encore, une action primitive comme la création ou la destruction. L'action inclue une liste d'arguments en plus d'une référence à l'opération ou au signal. L'émission d'un signal et l'appel d'une opération sont logiquement semblables : ils impliquent la communication d'informations, de l'émetteur vers le récepteur, et ce dernier l'utilise pour déterminer ce qu'il doit faire.

L'implémentation d'un message peut prendre plusieurs formes telles que l'appel d'une opération, la transmission d'un signal, une communication entre processus, le déclenchement d'événements, etc. L'appel d'une opération synchrone est composé de deux messages : le message d'appel et le message de retour. L'opération asynchrone, quant à elle, se compose d'un seul message d'appel :

donc, le signal est aussi composé d'un seul message qui déclenche généralement une transition à l'intérieur de la machine à états finis du ou des récepteurs.

Lorsqu'un objet collabore avec un autre objet, ils interagissent par l'échange de messages. Lorsqu'un message est passé d'un objet actif vers un autre objet actif, il en résulte une communication inter-*thread*[18]. Deux styles de communication sont alors possibles. Le premier style survient lorsqu'un objet actif appelle une opération synchrone d'un autre objet. Une opération synchrone est une opération pour laquelle l'émetteur attend le résultat avant de continuer son exécution. Ce style de communication a une sémantique de type rendez-vous. Cela signifie que l'émetteur appelle l'opération, qu'il attend ensuite que le récepteur accepte l'appel, que l'opération soit exécutée et que le résultat (optionnel) soit retourné. Ce n'est qu'après cela que les deux objets continuent leur chemin de façon indépendante. Pour toute la durée de l'appel, les deux flots de contrôle sont synchronisés et l'émetteur est bloqué. Quant à lui, le second style survient lorsqu'un objet actif envoie un signal ou appelle une opération asynchrone. Une opération asynchrone est une opération pour laquelle l'émetteur n'attend pas le résultat pour continuer son exécution : la sémantique y est de type boîte aux lettres. Cela signifie que l'émetteur envoie un signal ou appelle une opération et continue immédiatement son exécution, sans perdre le contrôle. Pendant ce temps, le récepteur accepte le signal ou l'appel lorsqu'il est prêt, exécute l'opération et continue son propre chemin, une fois l'action terminée. Les deux objets ne sont pas synchronisés et l'émetteur n'est pas bloqué.

À l'intérieur d'un système concurrent, il est possible que plusieurs flots de contrôle accèdent à une même opération ou à plusieurs opérations d'un même objet. Lorsque plusieurs flots de contrôle accèdent à un objet en même temps, un problème classique de **synchronisation** peut en résulter : il est possible qu'un flot de contrôle vienne gêner l'exécution d'un autre flot, avec comme résultat la corruption de l'objet. L'incapacité à gérer ce problème produit plusieurs conditions de course et d'interférence ayant pour conséquence de corrompre un système de façon mystérieuse et aléatoire.

La solution générique à ce problème de synchronisation pour les systèmes orientés objets est la suivante : il faut traiter l'objet comme une région critique. Les trois approches alternatives suivantes sont possibles, chacune impliquant la capacité d'ajouter une propriété de synchronisation aux opérations de la classe :

Sequential : il est de la responsabilité des émetteurs de synchroniser leurs appels à l'objet afin qu'à tout moment un seul flot de contrôle soit présent à l'intérieur de l'objet. La sémantique et l'intégrité de l'objet ne sont pas garanties lorsque cette condition n'est pas respectée.

Garded : plusieurs appels d'opérations en provenance de *threads* concurrents peuvent survenir simultanément sur l'objet, mais une seule opérations peut s'exécuter à la fois. Les autres sont bloqués jusqu'à ce que l'éxécution de la première soit terminée.

---

[18] Il est à noter que les objets passifs supportent uniquement les opérations synchrones, car ils ne possèdent pas de *thread* d'exécution leur permettant d'agir concurremment avec d'autres objets. Lorsqu'une opération d'un objet passif est appelée, elle commence et se termine à l'intérieur du contexte d'exécution de l'appelant.

Concurrent : plusieurs appels d'opérations en provenance de *threads* concurrents peuvent survenir simultanément sur l'objet et elles sont toutes exécutées de façon concurrente. Chaque opération doit pouvoir s'exécuter concurremment par rapport aux autres opérations de l'objet.

La deuxième approche, dite *Garded*, est celle retenue pour le système DOES, car son implémentation est simple et efficace par rapport aux besoins du système.

6.5.3 Classes de communication

Cette section décrit les classes qui servent à réaliser et à abstraire le concept d'objet actif ainsi que le mécanisme de messagerie expliquée précédemment. La hiérarchie de classes se compose principalement des classes *CThread*, *CAliveObj* et *CMsgDrivenObj*. La classe *CAliveObj* réalise de façon abstraite le stéréotype de classe active. Quant à la réalisation du mécanisme de messagerie, elle repose sur la classe *CMsgDrivenObj*. Cette dernière hérite de la classe *CAliveObj* et elle s'assure que les opérations synchrones et asynchrones s'exécutent de façon sécuritaire à l'intérieur du contexte de l'objet actif. La classe *CMsgDrivenObj* est principalement composée d'une queue de message et gère la boucle de messagerie.

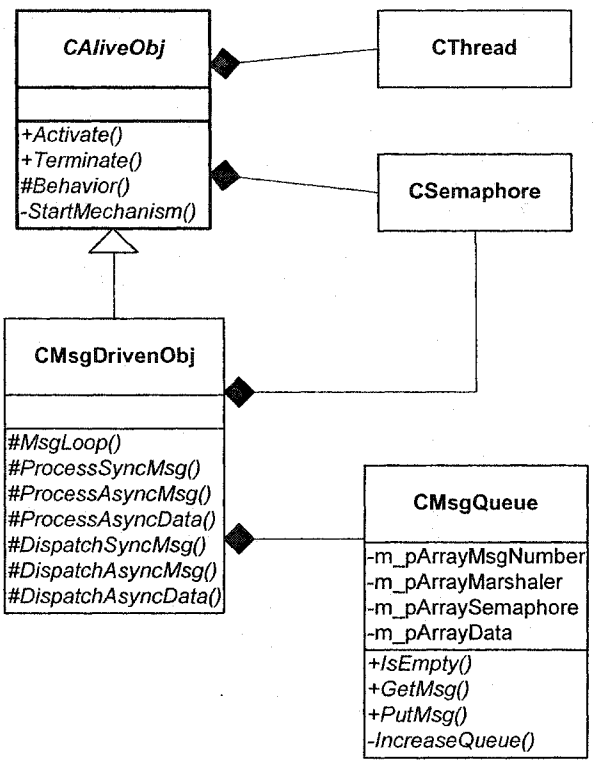

Figure 6.12 Diagramme des classes CAliveObj et CMsgDrivenObj

Les classes illustrées par la Figure 6.12 permettent à tous les objets qui héritent de *CMsgDrivenObj* d'être protégés de façon *Garded*. Même si plusieurs accès concurrents peuvent survenir, une seule opération peut s'exécuter à la fois, garantissant qu'aucun accès simultané aux données ne peut survenir. Pour arriver à protéger entièrement le contexte d'un objet actif, il est important que l'accès aux objets internes qui le composent se fasse par l'entremise des opérations

de son interface publique; l'utilisation par un objet externe d'une référence ou d'un pointeur à un objet interne est à proscrire. Ainsi, l'intégrité de l'objet actif est protégée, et ses accès synchronisés. C'est dans ce sens que le modèle présenté se rapproche du modèle STA de COM.

CAliveObj

La responsabilité de la classe *CAliveObj* est d'offrir des opérations de création et de gestion d'objets actifs. Cette classe, qui est purement abstraite et qui doit être spécialisée par une autre classe, permet de rendre un objet actif, sans pour autant gérer l'accès à ses opérations. *CAliveObj* est composée d'un *CThread* qui est définie à la section 6.7.

À sa création, l'objet est « *asleep* ». Cela signifie que le *thread* associé n'est pas encore démarré. L'objet devient réellement « *alive* » lorsque la méthode *Activate()* est appelée. A ce moment, le *thread* est créé, démarré et l'objet devient « *alive* ». Le système d'exploitation responsable de la gestion des *threads* nécessite généralement un point d'entrée pour débuter l'exécution du *thread* : cette fonction peut être comparée à la fonction *main()*. Dans le contexte de la classe *CAliveObj*, cette fonction est nommée *Behavior()*. La méthode *Behavior()* est une fonction purement virtuelle qui doit absolument être redéfinie par la classe héritant de *CAliveObj*.

Lors de l'activation d'un nouveau *thread*, il est possible de rencontrer une condition de course (*race condition*) critique. Cette condition survient lorsque l'activation du *thread*, par le système d'exploitation, n'est pas terminée même si la construction de l'objet est réussie et que le client appel une opération de l'objet[19]. C'est pourquoi un sémaphore est utilisé pour signaler au client que le démarrage du *thread* est réussi et que l'activation de l'objet est terminée. Le client est donc débloqué et il peut continuer son exécution dès le signal de ce sémaphore. Ce mécanisme est pris en charge par la méthode *Activate()* qui est donc synchrone.

La mort d'un *Alive Object* est définie comme étant la fin de l'exécution du *thread* associé. Il est de la responsabilité de l'objet de définir le mécanisme qui permet de terminer l'exécution de son flot de contrôle, mais il n'est généralement pas souhaitable de devoir invoquer une fonction de destruction du système d'exploitation. La fin du *thread* se traduit par la fin de la fonction utilisée pour son démarrage : dans ce cas-ci, c'est la méthode *Behavior()*. Pour s'assurer d'un mécanisme de terminaison commun à tous les objets actifs, la fonction purement virtuelle *Terminate()* est définie. Cette méthode doit assurer, d'une façon quelconque, la terminaison de la méthode *Behavior()* et par conséquent du *thread* associé.

L'exemple suivant démontre une réalisation simple de la méthode *Terminate()* :

```
class CChildrenOfAliveObj : public CAliveObj
{
public:
    CChildrenOfAliveObj() : m_bContinue(true) {}

    void Terminate()
```

[19] En fonction du système d'exploitation, de son mécanisme de *multithreading* et des différentes priorités, il est possible que la construction de l'objet se termine avant que le *thread* associé ne soit complètement créé et démarré.

```
    {
        m_bContinue = false;
    }

private:
    void Behavior()
    {
        while (m_bContinue == true)
        {
            ... Do something useful ...
        }
    }

private:
    bool m_bContinue;
};
```

Lorsqu'un objet actif meurt, il passe à l'état « *dead* ». Sa destruction en mémoire survient uniquement lorsque son destructeur est appelé. Il est cependant essentiel de tuer l'objet avant sa destruction, car si le *thread* associé est toujours actif après la destruction de l'objet, il est alors possible que des données, maintenant non disponibles en mémoire, soient accédées par le flot de contrôle. Il est donc de la responsabilité de la classe enfant, et de son client, de veiller à ce que l'exécution du *thread* soit stoppée avant la destruction complète de l'objet en mémoire. Lors de l'appel de son destructeur, la classe *CAliveObj* vérifie que l'état de l'objet correspond à « *dead* ». Rien de plus ne peut être fait par le destructeur de la classe *CAliveObj* puisqu'il est exécuté après celui de la classe enfant et que la façon normale de stopper le *thread* est implantée par l'enfant et n'est pas connue de *CAliveObj*.

Si l'implémentation de la méthode *Terminate()* est asynchrone, une mesure particulière doit être prise pour synchroniser la destruction de l'objet, car il est très important que le *thread* soit proprement terminé avant que l'appel au destructeur ne soit complété. La classe *CAliveObj* possède un mécanisme pour aider à la synchronisation. Lors de l'activation, il est possible de passer en paramètre un pointeur à un sémaphore. Le sémaphore passé en paramètre est automatiquement signalé lorsque la fonction *Behavior()* se termine. Le signal de ce sémaphore confirme que l'état « *dead* » est atteint.

Pour certains types d'objets, il peut être désirable d'automatiser l'activation et la terminaison du *thread*. L'exemple suivant démontre comment y arriver simplement :

```
class CChildrenOfAliveObj : public CAliveObj
{
public:
    CChildrenOfAliveObj()
    {
        ... Do the needed initialization ...
        Activate(&m_terminationSemaphore);
    };

    ~CChildrenOfAliveObj()
    {
        Terminate();
        m_terminationSemaphore.Wait();
        ... Do the needed memory desallocation ...
```

```
    };
private:
    CCountingSemaphore m_terminationSemaphore;
};
```

CMsgDrivenObj

La classe *CMsgDrivenObj* est une spécialisation de la classe *CAliveObj*. La responsabilité la plus importante de cette classe est de veiller à la gestion et à la propagation des messages du client vers l'instance de l'objet. Tous les messages en provenance de l'extérieur (externe au contexte du *thread* associé) transitent par les méthodes de cette classe. La queue des messages de type FIFO[20] est le point d'entrée commun pour tous les messages.

Lorsqu'un message est reçu, différentes informations faisant partie d'une structure sont ajoutées à la queue de messages *CMsgQueue*. Il y a premièrement un identificateur de message : cet identificateur représente de façon unique, à l'intérieur du contexte de l'interface de l'objet actif, la méthode appelée par le client. Deuxièmement, les valeurs des paramètres transportées par le message sont aussi incluses. Pour que le mécanisme puisse rester générique, les paramètres sont sérialisés à l'intérieur d'un objet *CMarshaler*[21] : il en résulte une sérialisation de la valeur des différents paramètres suivant les règles d'encodages BDER et pouvant ainsi être véhiculées via la queue de messages. Un objet de type *CMsgQueue* correspond à une région critique puisqu'il est sollicité par deux flots de contrôle, celui de l'émetteur et celui du récepteur. C'est pourquoi son accès doit être synchronisé à l'aide d'un sémaphore.

Le comportement de la classe *CMsgDrivenObj* est d'attendre qu'un message soit déposé dans sa queue de messages. Il est à noter que ce comportement est réalisé pour la méthode *MsgLoop()* et que cette même méthode est la seule à faire partie de la fonction virtuelle *Behavior()* héritée de *CAliveObj*. Lorsqu'un message arrive, il est retiré de la queue de messages et il est redirigé vers la méthode interne appropriée en fonction du membre associée au message. La distribution des messages est alors implémentée via les méthodes *DispatchSyncMsg()*, *DispatchAsyncMsg()*, *DispatchAsyncData()* : il est de la responsabilité de la classe spécialisée de réaliser ces fonctions purement virtuelles puisque son interface est inconnue de sa classe mère.

Les communications de types synchrones et asynchrones sont réalisées par *CMsgDrivenObj*. L'opération de réception d'un message est réalisée par les méthodes *ProcessSyncMsg()*, *ProcessAsyncMsg()* et *ProcessAsyncData()*. Le préfix *Process* est utilisé parce que le client, via l'interface de l'objet actif, demande à ce dernier de recevoir et d'exécuter l'action désignée par le message.

La méthode *ProcessAsyncMsg()* est réalisée de cette façon : si l'objet est actif, le message est inséré à l'intérieur de la queue et le sémaphore de *CMsgDrivenObj* signale à la méthode *MsgLoop()*, à l'intérieur du contexte de l'objet actif, qu'un nouveau message doit être traité. La synchronisation est basée sur le fait que seule la queue de messages est accédée par de multiples flots de contrôle. Il est important de comprendre que le message est déposé à l'intérieur de la

[20] First In First Out.
[21] Voir la section 6.4.

queue de l'objet actif par l'émetteur du message, via son propre flot de contrôle. L'exécution de la méthode *ProcessAsyncMsg()* se fait donc à l'intérieur du contexte de l'émetteur. Parce que l'opération est de type asynchrone, il est impossible pour l'émetteur de savoir quand le traitement du message a lieu du côté du récepteur : aucun retour de fonction n'est donc possible puisque le flot de contrôle de l'émetteur n'est pas synchronisé avec celui du récepteur. La méthode *ProcessAsyncData()* est une version optimisée de la méthode *ProcessAsyncMsg()*. Cette méthode offre moins de protection d'erreur, mais est plus efficace lorsqu'un message est envoyé de façon intensive.

La méthode *ProcessSyncMsg()* est utilisée lors de l'envoi d'un message synchrone. Le message est quand même inséré à l'intérieur de la queue et le sémaphore est signalé par la méthode *ProcessSyncMsg()* : l'émetteur bloque alors sur un sémaphore temporaire jusqu'à ce que l'exécution de la méthode invoquée se termine et que le sémaphore soit signalé à son tour. Dans ce cas, il est possible d'avoir un retour de fonction. La technique utilisée afin de pouvoir retourner une valeur est la suivante : un objet du même type que le type de retour est alloué sur la pile. Un pointeur sur cet objet est sérialisé et passé en paramètre avec les autres données. À la fin de son exécution, la méthode met à jour la valeur de l'objet pointé. Puisque l'expéditeur est bloqué par un sémaphore jusqu'à la fin de l'exécution de l'opération, l'accès à cet objet est sécuritaire et ne nécessite aucune synchronisation. Pour y arriver, un objet actif doit posséder son propre contexte d'exécution (*thread*) et il n'est permis de modifier l'objet qu'à l'intérieur de ce contexte.

6.5.4 Autres considérations

Il est important pour un objet actif de protéger son intégrité par l'utilisation de la classe *CMsgDrivenObj*. Il est aussi important de comprendre que la protection est relative au niveau d'encapsulation. Par exemple, si un objet O1 possède un lien sur un objet O2 qui compose un objet actif O3, il est possible pour l'objet O1 d'accéder à l'objet O2 et de le corrompre. Pour éviter cela, il est important d'encapsuler tous les objets passifs composant un objet actif et de ne pas permettre un accès direct à ces objets. Dans ces cas-ci, tous les messages de O1 doivent passer par l'interface de l'objet O3.

Une encapsulation sécuritaire requiert donc que toutes les méthodes publiques de la classe active redirigent le message vers la queue de messages de *CMsgDrivenObj*. De plus, toutes les données membres doivent être privées et accessibles uniquement par des méthodes de lectures publiques. Seule la lecture de données constantes peut être permise directement de l'externe et sans protection, puisque la valeur de ce type de donnée ne change jamais.

Il est à noter que les objets passifs, qui sont partagés et utilisés par au moins deux objets actifs, doivent généralement être protégés et leurs accès synchronisés. Une attention particulière doit être portée afin d'éviter tout risque de *deadlock*. Les appels d'opération synchrones sur un objet actif introduisent aussi un danger de *deadlock*.

6.6 Algorithme de gestion mémoire

En ce qui a trait aux systèmes évolués, la gestion de la mémoire est généralement prise pour acquise. Par contre, le ou les algorithmes fournis sont habituellement très génériques et plus ou moins efficaces. Dans certains cas, il peut s'avérer primordial d'y apporter des améliorations, car une gestion de la mémoire adaptée au système entraîne de meilleures performances ainsi qu'une robustesse accrue. Parfois, la vitesse d'exécution peut même doubler en changeant simplement l'algorithme d'allocation de la mémoire.[22] La présente section explique et illustre un algorithme de gestion de la mémoire adapté aux besoins de DOES. Cet algorithme est tiré de l'article « Efficient Memory Allocation » publié par Sasha Gontmakher and Ilan Horn. [12]

Il existe plusieurs types d'algorithmes d'allocation de mémoire permettant de rencontrer les besoins d'un système. Certains sont dits « on-line » : les requêtes sont consommées une à la fois sans aucune connaissance des requêtes futures. D'autres sont dits « off-line » : le flot de requêtes est connu entièrement, et ce, à l'avance : l'algorithme peut ainsi tirer avantage de ce fait et maximiser les résultats. Bien entendu, aucune stratégie d'allocation ne peut prétendre être optimale pour tous les types de système. Un algorithme efficace et réaliste pour un système évolué comme DOES réside quelque part entre les deux extrêmes.

Étant donné que DOES, de par son mode de fonctionnement asynchrone, effectue une allocation intensive d'objets, il est nécessaire de veiller à optimiser cette partie pour ne pas embourber les systèmes plus restreints en terme de ressources. La stratégie proposée repose sur les hypothèses suivantes :

- Des objets sont alloués, libérés et réalloués de façon fréquente;
- Les objets alloués sont majoritairement de petite taille (< 2 KOctets);
- Le nombre d'objets de tailles différentes est limité et déterminable;
- La taille moyenne instantanée des objets varie dans le temps.

Une façon simple et très efficace de gérer des allocations fréquentes de mémoire consiste à exploiter des listes chaînées de blocs de mémoire pré-alloués et de taille prédéterminée basée sur la grosseur des différents objets du système. Afin de réponde adéquatement aux besoins des différents types d'objet, plusieurs listes sont créées et chacune est composée de blocs de même taille; la taille des blocs variant entre les différentes listes. Pour retrouver rapidement une liste lorsqu'un bloc de taille t est nécessaire, un tableau de pointeurs de liste contient un pointeur pour chacune des listes. De plus, l'index de chacun des éléments du tableau est fonction de la taille des blocs de la liste qui y est référencée. Donc, pour allouer un bloc de taille t, l'élément en position de tête est retiré de la liste pointée par la valeur de l'élément situé à l'index t du tableau de listes. Cet algorithme est d'ordre constant et la recherche extrêmement rapide, car l'accès à seulement deux pointeurs est nécessaire pour arriver à trouver un bloc de mémoire disponible et de bonne taille.

[22] Des tests de performance effectués sur la plate-forme Windows NT 4.0 permirent de démontrer que l'algorithme proposé est de 2,5 à 3,5 fois plus performant que la fonction *malloc()* du système d'exploitation.

Si la liste est vide, d'autres objets de taille *n* doivent être alloués à partir des ressources du système, pour compléter l'opération avec succès. Lorsqu'un objet est libéré, celui-ci est simplement réintroduit à la tête de sa liste. Afin de rendre efficace l'allocation d'une page pour chacune des tailles d'objet, il ne doit y avoir que très peu d'objets de tailles différentes. Ainsi chaque page devrait tendre vers une utilisation maximale de sa capacité, ce qui permettrait une utilisation maximale de la mémoire du système. De plus, le phénomène de fragmentation de la mémoire, commun aux algorithmes traditionnels, est ainsi évité.

Cette première description simpliste sert de base à l'élaboration d'un module de gestion de mémoire, complet et flexible. Ceci répond mieux aux besoins des systèmes dont les séquences d'allocation d'objets sont difficilement prévisibles et très exigeantes. Les sections suivantes proposent une structure basée sur l'utilisation d'une méta-page et de plusieurs pages de mémoires fractionnées en blocs de taille identique. Les algorithmes de recherche, d'allocation et de libération des pages et des blocs de mémoire y sont décrits en détails.

6.6.1 Méta-page de mémoire

La méta-page est responsable de la gestion d'un super bloc de mémoire préalablement alloué. Ce super bloc de mémoire est généralement obtenu à partir du silo, par l'intermédiaire d'une fonction du système d'exploitation (*malloc*). Par la suite, chaque allocation dynamique de mémoire s'effectue à l'intérieur de ce bloc. Pour faciliter la gestion des allocations et maximiser l'utilisation de l'espace mémoire disponible, la méta-page est segmentée en un nombre fini de pages de même dimension. La figure suivante représente une méta-page.

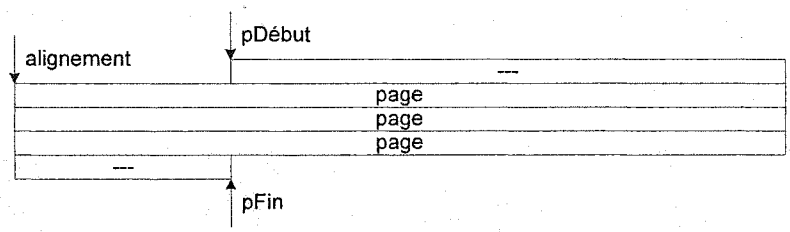

Figure 6.13 Méta-page

À sa création, la méta-page génère une liste de pages vides et prêtes à être utilisées. Elle est alors en mesure de rechercher, de sélectionner et d'initialiser des pages pour répondre aux besoins d'allocation de mémoire. La recherche et l'utilisation d'une page se font à deux occasions : la première survient lors de l'allocation d'un bloc et la seconde survient lorsqu'un bloc de mémoire est relâché.

Le relâchement d'un bloc de mémoire implique qu'il soit réintégré au sein de la liste de blocs libres de sa page. La difficulté de cette action réside dans le fait qu'une fois allouée, un bloc de mémoire contient uniquement les données utiles au client. Aucun en-tête d'information n'y est ajouté (voir ci-dessous). La page de mémoire doit alors être retrouvée, avec comme seule information, l'adresse mémoire du bloc (valeur du pointeur). Pour y arriver, le positionnement des pages doit se faire sur une frontière logique permettant de les retracer rapidement. Une façon portable de réaliser ceci, sans miser sur les capacités du système d'exploitation, est d'aligner la première page sur la plus petite adresse mémoire correspondant à une puissance de sa taille, à

l'intérieur de la méta-page. La page suivante est contiguë à la première et ainsi de suite. De plus, toutes les pages doivent être de même taille. Cet alignement des pages permet l'utilisation d'un algorithme de recherche rapide et constant. Pour retrouver l'adresse de la page, il suffit d'aligner à la baisse l'adresse du bloc en fonction de la taille d'une page. Le calcul d'alignement est réalisé à l'aide de l'Équation 6.6. Fait intéressant : si la taille de la page est une puissance de deux, l'arrondissement peut se faire encore plus rapidement avec une simple opération de masquage (comparaison & logique). La nouvelle adresse obtenue est alors interprétée comme étant l'adresse du début de la page. Le bloc de mémoire peut donc y être réintégré.

Comme illustré par la Figure 6.13, à cause de l'alignement, les sections situées aux frontières de la méta-page risquent d'être ignorées et perdues. Par contre, la perte mémoire encourue par l'alignement est négligeable si le nombre de pages à l'intérieur de la méta-page est suffisamment grand. L'équation suivante démontre cette affirmation :

MPsize = taille de la méta-page en nombre d'octets;
Psize = taille de chaque page en nombre d'octets;
MPlost = pourcentage de mémoire perdue à l'intérieur de la méta-page suite à l'alignement de la première page.

$$MPlost\% < \frac{MPsize - \left(\left(\left[\frac{MPsize}{Psize}\right] - 1\right) \bullet Psize\right)}{MPsize} \bullet 100$$

Équation 6.1

Selon l'équation précédente, si la taille de la méta-page est 100 fois supérieure à la taille d'une page, la perte de mémoire maximale résultant d'un alignement de la première page est inférieure à 1 % de la taille de la méta-page. Il est recommandé d'employer une taille de page égale à une puissance de deux, et idéalement, beaucoup plus grande que la taille de l'objet le plus souvent alloué par le système.

Toutes les pages vides font partie de la même liste. Chaque fois qu'une nouvelle page est nécessaire à l'allocation d'un bloc, elle est retirée de la tête de cette liste. Une fois tous ses blocs relâchés, la page est réinsérée en tête de la liste des pages vides. La page peut alors être configurée pour contenir des blocs d'une taille différente.

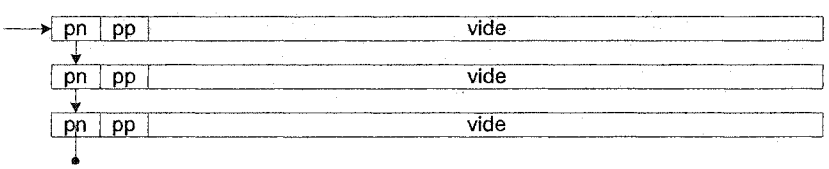

Figure 6.14 Méta-page -- liste de pages vides

L'allocation des blocs de mémoire doit se faire de façon à minimiser le nombre de pages impliquées. C'est pourquoi, les blocs de même taille doivent tous être alloués au sein de la même page jusqu'à sa pleine capacité, avant qu'une autre page soit réservée. La méta-page doit être en

mesure de trouver rapidement une page qui possède un bloc de grandeur adéquate. Pour ce faire, elle dispose d'un tableau de listes de pages partielles indexées en fonction de la taille des blocs. Les listes sont doublement chaînées afin de permettre une extraction rapide et constante d'une page située en milieu de liste lorsqu'elle passe à un autre état. Toutes les pages partiellement utilisées font partie de cette structure.

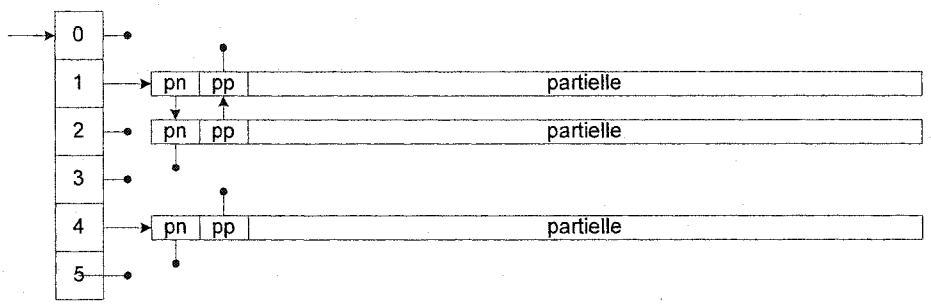

Figure 6.15 Méta-page -- tableau de listes de pages partielles

Pour déterminer l'index *i* de la liste d'une page contenant des blocs de taille *t*, il suffit de diviser *t* par le facteur d'alignement *fa*. Cette équation suppose qu'un bloc de taille zéro ne peut pas être alloué. Par exemple, avec un facteur d'alignement de quatre, la liste située à l'index 1 du tableau contient des pages dont les blocs sont de taille égale à huit. Pour déterminer la taille du tableau d'index, il suffit de calculer les différentes tailles de bloc qu'une page peut contenir.

$$i = \left\lceil \frac{t-1}{fa} \right\rceil$$

Équation 6.2

Lorsque tous les blocs d'une page sont alloués, cette dernière est dite complète. Comme l'illustre la figure suivante, une page complète ne fait partie d'aucune liste. De cette façon, la méta-page n'est pas encombrée par des pages dont elle ne peut pas faire usage.

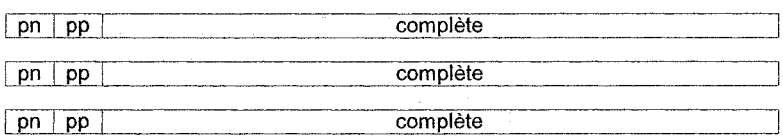

Figure 6.16 Méta-page -- pages complètes

Aussitôt qu'un objet libère un bloc de mémoire associé à une page complète, cette dernière est réintégrée au sein de la liste des pages partielles correspondantes. Si la page contient un seul bloc assez grand pour occuper tout son espace disponible, lorsque ce bloc de mémoire est libéré elle est alors directement réintégrée à la liste des pages vides.

6.6.2 Page de mémoire

La page est responsable d'une section mémoire plus ou moins grande et elle gère l'allocation de blocs de mémoire au sein de cette section. La page est structurée de façon à éviter certains problèmes, tout en maximisant l'efficacité des algorithmes d'allocation. Les paragraphes suivants expliquent les raisons qui ont contribué au choix de cette structure.

L'implémentation de la fonction *malloc()* ajoute généralement un en-tête descriptif à chaque bloc de mémoire alloué. Dans ce cas, lorsqu'un programme alloue plusieurs blocs de petite taille, le surplus de mémoire utilisé peut être significatif : la page de mémoire est donc sectionnée en plusieurs blocs de même dimension afin d'éviter ce problème. De cette façon, seule la page contient un en-tête descriptif du format des blocs. La charge utile d'une page de mémoire est ainsi maximisée, et ce, même pour des petits blocs d'à peine quatre octets.

Certains programmes peuvent allouer un nombre important d'objets de taille t_1, puis les relâcher au complet et allouer de nouveau un nombre important d'objets de taille t_2. Cependant, il est possible qu'un objet de taille t_2 ne puisse pas être alloué à l'intérieur d'un bloc de taille t_1. C'est pourquoi, il est primordial de pouvoir réutiliser une page précédemment initialisée. Pour y arriver, il est nécessaire de garder un compteur des blocs alloués pour chacune des pages. Alors, lorsque la page est vide (nombre de blocs alloués = 0), il est possible d'utiliser cette page à nouveau pour des blocs de taille différente. De ce compteur découle les trois états d'une page : complète, partielle et vide.

Le fait de garder une liste des pages vides au niveau de la méta-page permet la réutilisation rapide de celles-ci, uniquement en modifiant leurs paramètres. Cette méthode introduit par contre un problème de performance. En supposant qu'un programme alloue un objet, puis le libère, en alloue un autre de même taille, le libère et ainsi de suite, il en résulterait un mouvement d'insertion et d'extraction très intensif d'une même page à l'intérieur de la liste des pages vides. La création d'une liste de blocs au sein de cette même page, à chaque initialisation, serait alors désastreuse. Afin d'éviter ce problème et de permettre une initialisation rapide de la page, deux pointeurs sont exploités : le premier pointe sur le prochain bloc encore inutilisé et le second pointe sur une liste de blocs libres qui ont précédemment été utilisés. À l'initialisation, le premier pointeur indique le premier bloc de la page et le second est nul puisque aucun bloc n'a encore été utilisé.

La taille de l'en-tête ainsi que la taille des blocs d'une page, assurent l'alignement de tous les type de données. L'espace d'alignement doit correspondre à une section de la page qui n'est pas utilisée, ni pour l'en-tête ni pour les données : cette section garantit que l'adresse du bloc alloué est alignée en mémoire pour tous les types de données. Si l'alignement choisi correspond à l'alignement du système, le compilateur se charge automatiquement de créer l'en-tête de la page avec une taille qui assure l'alignement. Afin que l'alignement soit également respecté pour les blocs qui suivront, la taille des blocs d'une page doit toujours être égale à une puissance du facteur d'alignement. De plus, pour que les données de l'en-tête de la page puissent être alignées en mémoire, la taille des pages doit, elle aussi, correspondre à une puissance du facteur d'alignement du système. Pour toutes ces raisons, la page est organisée de la façon suivante :

Figure 6.17 Page

La figure suivante explique chacun des différents éléments qui forment la page :

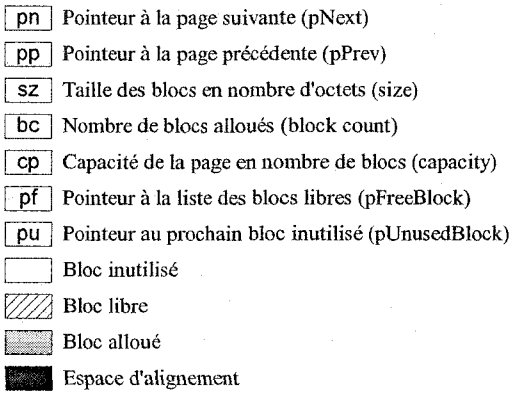

Figure 6.18 Légende de la page

La page vide fait partie d'une liste simplement chaînée où seul *pn* est utilisé. Dans cet état, la taille des blocs est indéterminée (sz = ?), le nombre de bloc alloué est nécessairement nul (bc = 0), la capacité inconnue (cp = ?) et les pointeurs aux listes de blocs sont nuls (pf = pu = 0).

Figure 6.19 Page vide

La page partielle fait partie d'une liste doublement chaînée avec *pn* et *pp* qui pointent sur les pages suivante et précédente. La page partielle possède une liste de blocs libres *pf* ainsi qu'un pointeur sur le prochain bloc à allouer *pu*. Lorsqu'une page passe de l'état vide à l'état partiel, son initialisation se fait rapidement et de façon constante : la taille des blocs est définie par *sz*; le compteur de blocs alloués est mis à zéro; la capacité est calculée en fonction de *sz* et de l'espace disponible à l'intérieur d'une page (taille - en-tête). De plus, le pointeur *pf* est nul puisqu'aucun bloc n'à été relâché et *pu* pointe sur le prochain bloc à être utilisé. De cette façon, l'initialisation est rapide et constante. Par la suite, les blocs alloués et relâchés contribuent à faire évoluer la liste de blocs libres *pf*. Lorsque la liste *pf* est vide, l'allocation d'un autre bloc se fait à partir de *pu*. Lorsque *bc* égale *cp*, la liste passe de l'état partiel à l'état complet.

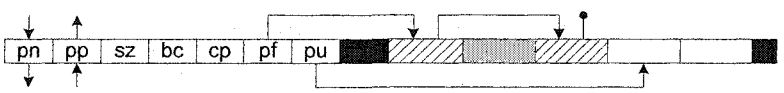

Figure 6.20 Page partielle

Une fois tous les blocs d'une page alloués (*bc* = *cp*), celle-ci est retirée de la méta-page. Tous ses pointeurs sont nuls et *pu* ne doit plus servir jusqu'à ce que la page soit réinitialisée. Pour être réintégrée à l'intérieur de la méta-page, au moins un des blocs de la page doit être libéré.

Figure 6.21 Page complète

6.6.3 Alignement en mémoire

La majorité des systèmes requiert que certains types, comme les entiers, soient alignés en mémoire. L'alignement se fait généralement sur 4 octets pour une architecture 32-bits et sur 8 octets pour une architecture 64-bits. Partant de ce fait, si un programme veut allouer 3 octets, alors un bloc de 4 octets peut lui être attribué puisque le dernier octet est perdu. Il est ainsi possible d'augmenter l'efficacité de l'algorithme pour qu'un bloc de taille *t* soit alloué à chaque requête dont la taille demandée se situe à l'intérieur de l'intervalle [*t* – *fa* + 1 .. *t*]. Par exemple, pour un système dont l'alignement se fait sur 4 octets, toutes les requêtes d'allocation de 1, 2, 3 et 4 octets se font à l'intérieur de blocs de 4 octets.

Les équations suivantes sont utilisées pour faire le calcul d'alignement :

t = taille de l'objet en nombre d'octets;
t_{up} = taille arrondie à la hausse;
t_{dn} = taille arrondie à la baisse;
fa = facteur d'alignement.

$$t_{up} = \left(\frac{t + fa - 1}{fa} \right) \times fa$$

Équation 6.3

$$t_{dn} = \left(\frac{t}{fa} \right) \times fa$$

Équation 6.4

Les équations suivantes représentent une version optimisée en langage C des équations précédentes :

$$t_{up} = (t + fa - 1) \,\&\, \sim(fa - 1)$$

Équation 6.5

$$t_{dn} = (t) \,\&\, \sim(fa - 1)$$

Équation 6.6

6.6.4 Objet de forte taille

L'algorithme suppose que la taille des objets les plus grands ne dépasse pas la taille d'une page. Dans le cas où le programme aurait besoin de l'allocation d'un objet de dimension plus importante, l'allocation de la mémoire nécessaire devra être relayée au système d'exploitation par l'appel d'une fonction tel que *malloc()*. C'est l'adresse de l'objet lors de son effacement qui indique si son bloc mémoire associé fait partie d'une page de la méta-page ou non. Si l'adresse pointe à l'intérieur des frontières de la méta-page, alors cet objet en fait partie; sinon la libération de son bloc de mémoire doit être prise en charge par le système d'exploitation - *free()*. Heureusement, les objets de forte taille (> 4 KOctets) sont rarement nécessaires, faisant en sorte que le nombre des allocations mémoires moins performantes, réalisées directement par le système d'exploitation, est négligeable.

6.6.5 Algorithmes d'allocations

L'allocation d'un bloc de mémoire pour un objet de taille *t* suit les étapes suivantes :

1. Demande d'allocation d'un bloc de mémoire de taille *t* à la méta-page;
2. Recherche ou création d'une page de blocs de taille correspondante;
3. Réservation d'un bloc de mémoire de la page sélectionnée;
4. Mise à jour de l'état de la page (complète, partielle, vide) et positionnement de celle-ci à l'intérieur de la liste des pages de même état.

Voici la description de l'algorithme en format pseudo-code :

```
Alignement à la hausse de la taille t : t_up = (t + fa - 1) & ~(fa - 1)
si t_up <= espace page alors
   calcul de l'index i des pages partielles : i = t / fa
   si une page partielle existe à l'index i alors
      allocation du bloc à partir de cette page
      si la page est complète alors
         la page est retirée de la liste des pages partielles
   sinon
      si un page vide est disponible alors
         extraction de la page de la liste des pages vides
         initialisation de la page avec t_up
         allocation du bloc à partir de cette page
         si la page est partielle alors
            ajout de la page à la liste des pages partielles à l'index i

t    = taille du bloc
t_up = taille du bloc alignée à la hausse
fa   = facteur d'alignement
i    = index de la liste des pages de blocs de taille t_up
```

La libération d'un bloc de mémoire pour un objet de taille *t* suit les étapes suivantes :

1. Recherche de la page correspondant au bloc libéré;

2. Libération du bloc;
3. Mise à jour de l'état de la page (complète, partielle, vide) et positionnement de celle-ci à l'intérieur de la liste des pages de même état.

Voici la description de l'algorithme en format pseudo-code :

```
si ptr pointe à l'intérieur des limites de la méta-page alors
   obtention du pointeur de page : pPage = (ptr + t_page - 1) & ~(t_page - 1)
   si la page est complète alors
      libération du bloc de mémoire
      si la page est vide alors
         ajout de la page à la liste des pages vides
      si la page est partielle alors
         calcul de l'index i des pages partielles : i = t / fa
         ajout de la page à la liste des pages partielles à l'index i
   sinon
      libération du bloc de mémoire
      si la page est vide alors
         ajout de la page à la liste des pages vides

pPage = adresse de la page
t_page = taille de la page
t      = taille du bloc
t_up   = taille du bloc alignée à la hausse
fa     = facteur d'alignement
i      = index de la liste des pages de blocs de taille t_up
```

Le code source qui réalise ces algorithmes se retrouve en annexe 2.

6.7 Abstraction de l'OS et des *Sockets*

L'un des critères de design du système DOES est d'offrir une portabilité rapide et facile vers différents systèmes d'opération[23]. La majorité des classes de la librairie sont entièrement portables, par contre certaines sections dépendent de services spécialisés offerts par le système d'exploitation (OS). Bien que les systèmes d'opération d'aujourd'hui offrent généralement tous les mêmes services de bases, il est rare que ces services soient offerts de la même façon et avec un seul et même API.

Afin de remédier à ce problème, la librairie de classes de DOES repose sur deux packages d'abstraction : *OsLayer* et *NetLayer*. Le premier, abstrait les services du système d'exploitation et fournit une interface orientée objet sur laquelle reposent les classes de niveau supérieur. Le second, abstrait les services des BDS *Sockets* et fournit une interface orientée objet avec traitement asynchrone des événements en provenances du réseau.

Ainsi, lorsque DOES doit être porté vers un environnement différent, les changements au code source sont limités à ces deux ensembles de classes bien précis. Il est évident que pour garantir la

[23] Le système d'exploitation doit comprendre un ensemble minimal de services nécessaires à la réalisation des classes de base de la couche d'abstraction *OsLayer*.

portabilité de DOES, aucun service du système d'exploitation ne peut être utilisé directement par les classes de haut niveau. Tous les appels de services de l'OS doivent se faire via les packages d'abstractions. La Figure 6.22 expose une vue en couches, selon le modèle OSI, des packages d'abstractions du système DOES et de leurs dépendances.

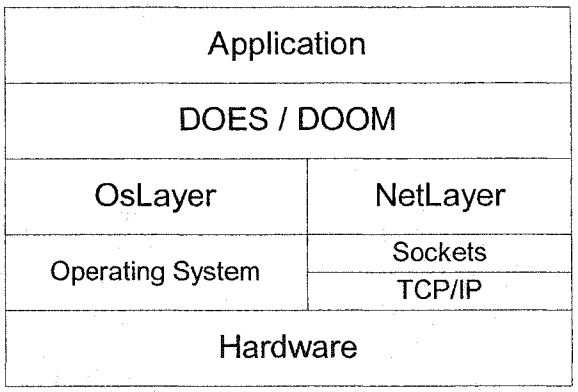

Figure 6.22 Vue en couches des packages d'abstractions

Les packages OsLayer et NetLayer sont constitués de classes avec des interfaces rigides et bien définies. Pour porter DOES vers un autre type de plate-forme, il suffit de réaliser chacune des opérations avec des méthodes qui tirent profit des services spécifiques de l'OS en plus de ceux de la pile TCP/IP visée.

Les prochains paragraphes décrivent ces classes. La description est cependant sommaire puisque ces classes sont bien connues et bien documentées par plusieurs ouvrages de référence [38] [39] [41]. Le but de la présente section est d'offrir une vue d'ensemble des classes d'abstractions qui sont disponibles.

6.7.1 Package OsLayer

Le package OsLayer offre une interface générique aux services de base disponibles sur la majorité des OS. Les classes de ce package réalisent principalement des services de *multithreading*, de synchronisation et de temporisation. La classe *CThread* est l'abstraction d'un *thread* (flot de contrôle) de l'OS. La classe *CTimer* gère l'aspect temps et rend possible l'implémentation d'opérations qui nécessitent des notifications temporelles, comme par exemple, la suspension d'un *thread* pendant un intervalle de temps précis. S'il est permis aux objets appartenant à plus d'un *thread* d'accéder à une même ressource, les classes *CBinarySemaphore* et *CCountingSemaphore* peuvent être utilisées pour assurer un accès synchronisé à cette ressource. La classe *CCriticalSection* permet aussi de synchroniser l'accès à une ressource partagée entre deux ou plusieurs *threads*.

CThread

Un processus est un flot de contrôle qui peut s'exécuter concurremment et indépendamment d'autres processus. Un *thread* est un flot de contrôle léger partageant l'espace mémoire et s'exécutant concurremment avec d'autres *threads* à l'intérieur du même processus. Les systèmes d'opération pour système embarqué offrent rarement le service de processus. Par contre, le

service de *thread* est généralement disponible. L'instance de la classe *CThread* représente un *thread* du système d'exploitation. Cet objet est chargé de gérer et de démarrer le flot de contrôle associé à l'aide des services de *threading* offerts par l'OS.

Les différentes opérations formant l'interface publique de cette classe et qui doivent être réalisées sont et signifient que :

```
CThread();
```

Le constructeur est responsable de créer l'instance et d'initialiser les différents membres de l'objet.

```
virtual ~CThread();
```

Le destructeur est responsable de relâcher les ressources réservées par l'objet lors de sa création. Si le *thread* associé au système d'exploitation n'a pas préalablement terminé son exécution, une destruction explicite réalisée par l'appel à la fonction de destruction du *thread* doit survenir avant de relâcher les ressources associées à l'objet *CThread.*

```
void StartThread(
    IN MxThreadFunction  pThreadFunc,
    IN void*             pParam,
    IN char*             pszName = "CThread",
    IN DWORD             dwStackSize = 0,
    IN ePriority         nPriority = eNORMAL);
```

Cette opération démarre l'exécution du flot de contrôle associé à l'objet via le pointeur de fonction *pThreadFunc.* Ce pointeur désigne le point de départ du *thread.* La fin de cette fonction correspond à la fin et à la destruction du *thread.* Attention, l'objet *CThread* n'est cependant pas détruit automatiquement : un appel explicite au destructeur doit toujours avoir lieu pour libérer toutes les ressources associées à l'objet. La valeur et la signification du premier paramètre *pParam* sont spécifiques au contexte du client. La fonction attachée au *thread*, via ce premier paramètre, n'est pas interprétée ou employée par la classe *CThread.* Un nom est associé au *thread* via le pointeur de caractère ASCII *pszName* : ce nom est généralement utilisé pour différencier les *threads* du système. Le paramètre *dwStackSize* spécifie la taille, en octets, de la pile associée au *thread.* Lorsque la valeur 0 est passée, la taille par défaut du système est utilisée. Le dernier paramètre de l'opération *nPriority* définit la priorité d'exécution du *thread.* L'exécution du *thread* affecté de la plus haute priorité est favorisée par l'OS. Les valeurs disponibles sont : eLOWEST, eLOW, eNORMAL, eHIGH, eHIGHEST.

CSemaphore

L'instance de la classe *CSemaphore* représente un sémaphore. Cet objet de synchronisation permet de gérer l'accès limité à une ressource. Il peut aussi servir à synchroniser deux *threads.* Le type d'accès, binaire ou multiple, est spécifié à la création de l'objet. Un accès binaire permet l'accès à une ressource par un seul *thread* à la fois, tandis qu'un accès multiple permet à plusieurs *threads* d'accéder à la même ressource au même instant.

Le nombre d'accès à un instant précis est toujours maintenu par le sémaphore. Le compte représente le nombre de permissions d'accès encore disponible. Lorsque la valeur du compte atteint zéro, cela signifie que tous les accès ont été distribués. Une nouvelle tentative d'accès résulte alors en une période d'attente pour le flot de contrôle. Les requêtes sont misent dans une file de type FIFO.

Les différentes opérations formant l'interface publique de cette classe et devant être réalisées sont et signifient que :

```
CSemaphore(
    IN int   nStartCount = 1,
    IN int   nMaxCount = 1);
```

Le constructeur est responsable de créer et d'initialiser l'objet sémaphore de l'OS. La valeur du paramètre *nStartCount* définit le nombre d'accès initialement disponible. La valeur de *nMaxCount* définit le nombre d'accès simultanés permis.

Pour construire un sémaphore binaire, il suffit de définir *nMaxCount* égal à 1.

```
~CSemaphore();
```

Le destructeur libère simplement l'objet système rattaché à l'instance de *CSemaphore*.

```
bool Wait(
    IN DWORD dwWaitTime = SEM_WAIT_INFINITE
    ) const;
```

L'appel de cette opération permet de gagner l'accès à la ressource. Cette opération est bloquante et retourne uniquement lorsqu'un accès à la ressource est disponible, ou que le temps en msec spécifié par le paramètre *dwWaitTime* est écoulé. Le retour est vrai, si le sémaphore est acquis, et faux, si le temps d'attente est expiré.

```
void Signal(
    IN bool bForceTaskSwitch = false
    ) const;
```

La disponibilité de la ressource est signalée à l'aide de cette opération. Lorsque le paramètre *bForceTaskSwitch* est vrai, un changement de contexte vers un autre *thread* a lieu avant le retour de l'opération.

CCriticalSection

L'instance de la classe *CCriticalSection* représente une section critique. Comme pour le sémaphore, cet objet de synchronisation permet de limiter l'accès d'une ressource à un seul *thread*. Par exemple, une section critique peut être utilisée lorsqu'un seul *thread* à la fois doit modifier les données membres d'un objet. La grande différence entre la section critique et le sémaphore réside dans le fait qu'une même section critique peut être pénétrée indéfiniment par le même *thread*, ce qui n'est pas le cas pour le sémaphore. De plus, pour être de nouveau accessible à un *thread* différent, la section critique doit être relâchée autant de fois qu'elle est acquise par le

même *thread*. Dans le cas d'un sémaphore, la méthode *Signal()* peut être appelée à partir de plusieurs *threads* différents.

Les différentes opérations formant l'interface publique de cette classe et devant être réalisées sont et signifient que :

```
CCriticalSection();
```

Le constructeur est responsable de créer et d'initialiser la section critique de l'OS.

```
~CCriticalSection();
```

Le destructeur libère la section critique.

```
void Enter() const;
```

L'accès à la section critique est gagné par l'appel de cette opération. L'appel de cette opération, par un deuxième *thread*, résulte en un blocage du flot de contrôle de ce *thread*.

```
void Exit() const;
```

La sortie du flot de contrôle de la section critique est signalée par l'appel à cette opération. Seul un *thread*, qui a déjà acquis la section, peut faire appel à cette opération.

Il est à remarquer que certains OS ne fournissent pas le service de section critique. Dans ce cas, il est toujours possible d'implémenter la classe *CCriticalSection* à l'aide de deux sémaphores binaires.

CTimer

La classe *CTimer* est utilisée pour réaliser les opérations nécessitant certaines notions de temps. Ainsi, l'exécution d'un *thread* peut être stoppée momentanément, ou encore de façon cyclique, avec une période bien précise. Deux opérations statiques, *Wait()* et *GetTime()*, peuvent être appelées sans création d'instance.

Les différentes opérations formant l'interface publique de cette classe et devant être réalisées sont et signifient que :

```
CTimer();
```

Le constructeur alloue les ressources nécessaires à l'objet. Il est obligatoire de créer un objet pour l'utilisation du service d'attente cyclique.

```
~CTimer();
```

Le destructeur relâche les ressources acquises par l'objet.

```
static void Wait(
    IN DWORD dwMs);
```

Cette opération suspend l'exécution du *thread* courant, pendant l'intervalle de temps en msec spécifié par *dwMs*. Le compte à rebours commence au moment de l'appel de l'opération. Une valeur de zéro provoque un changement de contexte vers un autre *thread* d'égale priorité. S'il n'y a aucun *thread* en attente, le *thread* courant regagne immédiatement le contrôle.

```
void CyclicWait();
```

Cette opération suspend l'exécution du *thread* courant jusqu'à ce que l'intervalle de temps, depuis le dernier appel à cette même opération, soit dépassé. L'intervalle est spécifié par l'appel à l'opération *StartCyclicWait()*. Le compte à rebours, pour chaque appel à *CyclicWait()*, est synchronisé avec l'appel de l'opération *StartCyclicWait()*. Si l'intervalle de temps spécifié par *dwMs* lors de l'appel à *StartCyclicWait()* est dépassé entre deux appels séquenciels à *CyclicWait()*, le deuxième appel de l'opération *CyclicWait()* ne bloque pas et retourne immédiatement, car la période du cycle est déja dépassée.

```
void StartCyclicWait(
    IN DWORD dwMs);
```

Cette opération initialise et démarre le service d'attente cyclique. La valeur du paramètre *dwMs* spécifie l'intervalle de temps en msec. Cette opération doit être appelée avant tout appel à l'opération *CyclicWait()*.

```
void StopCyclicWait();
```

Cette opératoin arrête le service d'attente cyclique.

```
static DWORD GetTime();
```

Cette opération retourne le temps système courant. Cette valeur représente le nombre de msec écoulées depuis le démarrage du système.

6.7.2 Package NetLayer

Il existe à ce jour plusieurs types de protocoles de transport qui reposent sur des concepts plus ou moins différents. Le concept de programmation réseau le plus répandu est probablement celui des BSD *Sockets*. Un *Socket* se définit comme un point de terminaison d'une communication réseau. Les librairies qui réalisent ces services de communication réseau (pile TCP/IP) offrent généralement une interface de programmation standard (API) basée sur les BSD *Sockets*. Malheureusement, les services les plus performants d'une librairie ne peuvent pas toujours être exploités via cet API. De plus, l'API est parfois mal adapté aux besoins propres à une application embarquée. Ces problèmes peuvent être contournés avec l'aide des services optimisés, possiblement offerts par la pile. Afin de maximiser la portabilité du code source, tout comme pour l'OS, le système DOES doit reposer sur un ensemble de classes ayant pour rôle d'abstraire les services de la pile TCP/IP. Ces classes doivent aussi offrir certains services adaptés aux besoins de DOES.

Deux classes et quelques fonctions réalisent une abstraction suffisante. Seules les classes *CSocketAddr* et *CAsyncSocket* sont décrites à l'intérieur de cette section puisqu'elles sont les deux classes principales du package *NetLayer*.

CAsyncSocket

La classe *CAsyncSocket* fournit une abstraction orientée objet de l'interface réseau, réalisée par la pile TCP/IP. Afin de faciliter sa compréhension, l'interface et l'implémentation de *CAsyncSocket* reposent sur les BSD *Sockets*. Un objet de la classe *CAsyncSocket* équivaut donc à un *Socket*. Pour sa réalisation, il est cependant possible de recourir aux services optimisés offerts par la pile.

L'implémentation de cette classe est plus complexe que les classes de l'*OsLayer* décrites précédemment puisqu'elle comporte des opérations asynchrones. De plus, les événements en provenance du réseau issus de la pile TCP/IP doivent être traités et notifiés de façon asynchrone. La classe *CAsyncSocket* est donc active. La Figure 6.23 illustre ses associations et dépendances. L'association d'héritage démontre que l'asynchronisme est réalisé à l'aide de la classe *CMsgDrivenObj* décrite à la section 6.5.

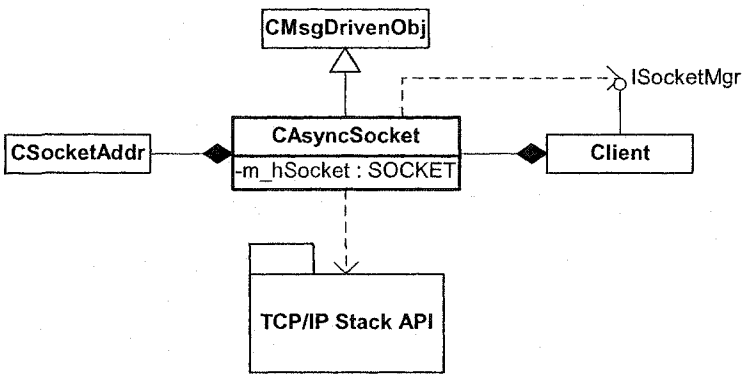

Figure 6.23 CAsyncSocket -- diagramme statique

Le client d'un objet de type *CAsyncSocket* doit hériter de l'interface *ISocketMgr*, car les événements asynchrones lui sont envoyés via cette interface. Cependant, il est d'abord nécessaire de configurer toutes les instances de *CAsycnSocket* avec un pointeur de gestionnaire valide avant d'appeler les autres opérations. Ensuite, l'opération *Create()* peut être appelée pour créer et initialiser le *Socket* au niveau de la pile TCP/IP. Pour que le *Socket* soit prêt à recevoir des paquets sur son port de communication, il faut appeler l'opération *Listen()*. L'opération *ConnectA()*, quant à elle, est appelée pour initier la connexion.

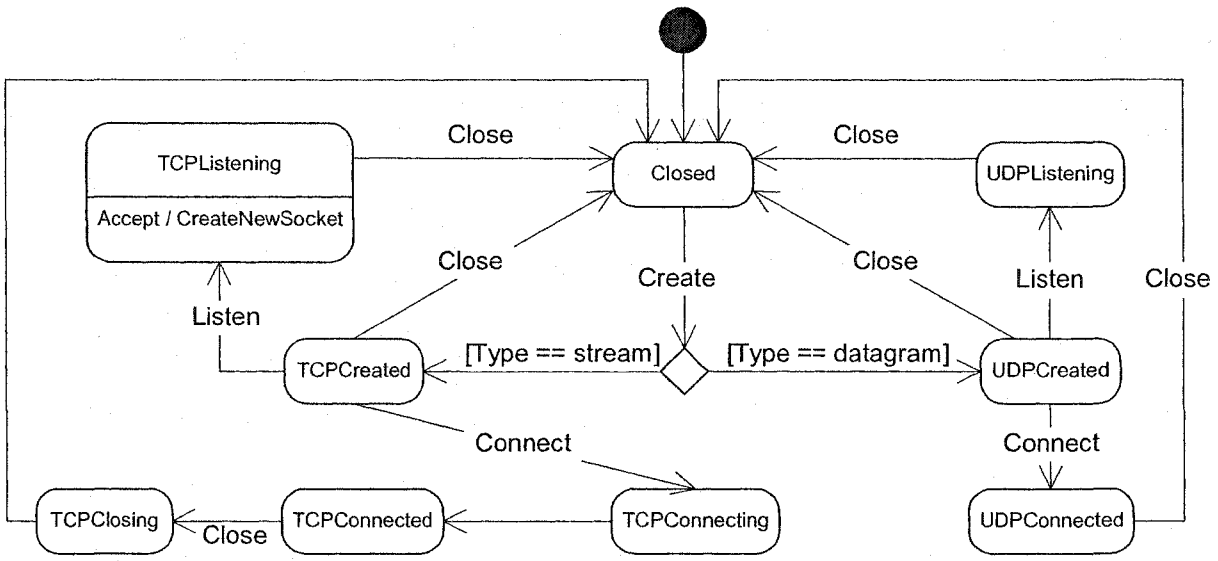

Figure 6.24 CAsyncSocket -- diagramme d'états

La figure précédente illustre les états de la classe *CAsyncSocket*. Cette machine d'états se rapproche de celle des protocoles TCP et UDP, mais avec un niveau d'abstraction plus élevé. Il est à noter que lorsque le *Socket* est dans l'état *eUDPConnected*, seuls les paquets en provenance du pair spécifié par *ConnectA()* sont acceptés. Les autres paquets sont automatiquement rejetés. Par contre, dans un état *UDPListening*, tous les paquets en provenance de n'importe quel nœud sont acceptés.

Plusieurs mécanismes internes peuvent être utilisés pour réaliser la classe *CAsyncSocket*. Cependant, le *polling* du *Socket* n'est pas acceptable dans un contexte de système embarqué. Lorsque l'API standard de BSD est le seul à être disponible, un second *thread* est généralement exploité : il est aussi possible d'en utiliser un seul, mais en ayant recours à la fonction *select()* afin d'éviter le *polling*. Ce deuxième *thread* est responsable de la notification asynchrone des événements réseaux via la queue de messages de l'objet *CAsyncSocket*. Malheureusement, les BSD *Sockets* n'offrent pas de fonctions simples pour la notification asynchrone. Pour maximiser la capacité de transport, il est donc préférable d'utiliser deux *threads* : l'un qui gère les opérations synchrones et asynchrones offertes au client et l'autre qui gère la notification des événements réseaux. Cette approche, bien qu'efficace, peut s'avérer lourde dans un contexte embarqué lorsque plusieurs *Sockets* sont requis par l'application. Il est aussi possible d'utiliser un seul *thread*, sans diminuer les performances, lorsque l'interface de la pile offre des services de gestion des événements asynchrones. La pile TCP/IP offerte avec la plate-forme *Windows* en est un bon exemple. Avec *Winsock2*, un objet de type *WSAEVENT* et la fonction *WSAEventSelect()* peuvent simplifier et optimiser la structure interne de *CAsyncSocket*. Pour y arriver, il est nécessaire de redéfinir la méthode virtuelle *MsgLoop()* de la classe *CMsgDrivenObj*. La notification des événements réseaux peut alors être intégrée à la boucle principale de consommation des événements en provenance de l'interface de *CAsyncSocket*. Malheureusement, cette technique

n'est pas réalisable sur tous les types d'OS puisqu'un service d'attente qui bloque simultanément sur plusieurs objets de synchronisation est requis[24].

Comme expliqué précédemment, les opérations de cette classe peuvent être implémentées de plusieurs façons. Puisque les mêmes opérations sont utilisées pour les *Sockets* de type *stream* et *datagram*, aucune méthode de l'interface ne peut être omise. La fonctionnalité *stream* de la classe n'est cependant pas requise par DOES. Seuls les services de *datagram* sont nécessaires. Il est donc possible d'omettre l'implémentation des sections responsables du protocole TCP[25].

Le TABLEAU 6.1 énumère les données membres de la classe *CAsyncSocket* dans l'optique de favoriser la compréhension de son interface.

TABLEAU 6.1 CAsyncSocket - données membres

Type	Nom	Description
ISocketMgr*	m_pSocketMgr	Pointeur sur l'objet client qui réalise l'interface ISocketMgr et qui recevra les différentes notifications asynchrones.
SOCKET	m_hSocket	Descripteur de *Socket* en provenance de la pile TCP/IP.
ESocketType	m_Type	Type du *Socket* (*stream* ou *datagram*).
UINT	m_unID	Identificateur (ID) qui sert à retracer la provenance des différents appels aux opérations de *ISocketMgr*. Ce cas survient lorsque plusieurs *Sockets* sont utilisés par le même client.
CSocketAddr	m_PeerAddr	Adresse du *Socket* distant.
ESocketState	m_State	État courant de l'objet.

Les différentes opérations formant l'interface de cette classe et devant être réalisées sont et signifient que :

Opérations de construction et de destruction

```
CAsyncSocket(
    IN UINT              unSockID = 0,
    IN ISocketMgr*       pSockMgr = NULL,
    CThread::ePriority   ePrio = CThread::eNORMAL);
```

L'appel du constructeur initialise les données membres de l'objet. La valeur de *m_pSocketMgr* est obtenue à partir de *pSockMgr*. Bien qu'une valeur nulle soit disponible par défaut, il est généralement préférable d'initialiser cette donnée membre au moment de

[24] Ce service est réalisé par la fonction *WaitForMultipleObjects()* sur la plate-forme Windows.
[25] La structure de la classe *CAsyncSocket* supporte les *Sockets* de type *stream* pour les besoins d'extension future.

sa construction puisque ce pointeur est nécessaire lors de la notification asynchrone des événements en provenance du réseau. Autrement, il est toujours possible de modifier l'adresse du client via l'opération *SetSocketMgr()*.

L'activation du *thread*, via la méthode *Activate()*, est aussi sous la responsabilité du constructeur. Le paramètre *ePrio* permet de changer la priorité du *thread* associé. Une fois que l'opération de construction est terminée, le *Socket* est activé et prêt à être utilisé.

Le constructeur initialise aussi l'état du *Socket* à *eClosed*.

```
virtual ~CAsyncSocket();
```

Le destructeur est responsable de fermer la connexion et de détruire le *Socket*. Il s'assure ainsi que l'état final du *Socket* sera *eClosed* avant de permettre la destruction. Il est aussi chargé d'appeler la méthode *Terminate()* et d'attendre que le *thread* de son parent *CAliveObj()* soit détruit avant de terminer son appel.

Opérations synchrones

Les opérations suivantes sont synchrones : leur exécution est généralement rapide et aucun événement en provenance du réseau n'est impliqué. Il est à noter que chaque opération synchrone retourne un objet *CError*[26] pour indiquer si l'exécution est réussie ou non.

```
CError Create(
    IN ESocketType          socketType = eDatagram,
    IN const CSocketAddr*   pLocalAddr = NULL);
```

L'appel de cette opération doit se faire lorsque le *Socket* est dans un état *eClosed*. Le type de *Socket* voulu (*stream* ou *datagram*) est passé en paramètre.

```
ESocketType
{
    eStream,
    eDatagram
};
```

Un descripteur de *Socket* est obtenu à l'aide de la fonction *socket()*. Il est ensuite sauvegardé au niveau de la donnée membre *m_hSocket*. La fonction *bind()* est aussi appelée pour attacher le *Socket* à un port local spécifique (définie par *pLocalAddr*) ou aléatoire.

Pour qu'il soit non-bloquant, il est préférable de modifier le *Socket* avec l'aide de la fonction *IOctl()*. Une fois toutes ces tâches exécutées avec succès, l'état du *Socket* doit être changé pour *eUDPCreated* ou *eTCPCreated* en fonction du type de *Socket* créé.

[26] Un objet de type *CError* contient un niveau et un code d'erreur.

```
CError Listen(
    IN WORD wMaxPendingConnection = 5);
```

Cette opération est appelée lorsque le *Socket* est en mode serveur. Cela signifie qu'il est possible avec ce *Socket* de recevoir et d'accepter une nouvelle connexion TCP (*stream*) ou des paquets UDP (*datagram*). L'appel de cette fonction doit se faire lorsque le *Socket* est dans un état *eUDPCreated* ou *eTCPCreated.*

Si le *Socket* est de type *stream*, la fonction *listen()* de la pile place ce dernier en mode d'écoute, avec une queue de connexions de longueur *wMaxPendingConnection* : ce paramètre ne sert qu'au *Socket* de type *stream*. L'état du *Socket* passe alors à *eTCPListening*. Lorsqu'une nouvelle connexion est demandée, l'opération *ISocketMgr::SocketAcceptedA()* est appelée pour en avertir le client.

Si le *Socket* est de type *datagram*, cette opération autorise la réception de paquets UDP sur le port d'écoute et permet aussi l'appel à l'opération *SendToA()*. L'état du *Socket* passe alors à *eUDPListening*.

```
CError Reset();
```

Cette opération peut être appelée dans tous les états. Elle termine la connexion et détruit le *Socket*, puis les données membres sont aussi remises à zéro. Avant le retour de la fonction, l'état du *Socket* passe à *eClosed*, mais attention, une connexion TCP est terminée abruptement par l'appel de cette opération.

Opérations asynchrones

Les opérations suivantes sont asynchrones : elles sont ainsi conçues pour éviter que le client bloque inutilement en attente d'un événement en provenance du réseau. Contrairement aux opérations synchrones, le retour de l'appel se fait avant que la méthode soit exécutée. Il est donc impossible de retourner le résultat de l'opération. Il faut prendre note que chaque opération asynchrone retourne quand même un objet *CError* pour indiquer si la transmission du message inter-*thread* est réussie ou non. Cependant, ce résultat ne correspond pas au résultat final de l'opération, mais indique au moins si l'opération sera exécutée ou non. En cas d'erreur pendant l'exécution de l'opération, *ISocketMgr::SocketError()* est appelée pour en informer le client.

```
CError ConnectA(
    IN const CSocketAddr& rPeerAddr);
```

Tout comme la fonction *connect()* de l'interface BSD *Sockets*, l'opération *ConnectA()* sert à établir une connexion avec un nœud distant. Le paramètre *nPeerAddr* indique l'adresse du serveur (IP + port). Cette opération doit être appelée uniquement lorsque l'état du *Socket* est *eUDPCreated* ou *eTCPCreated.*

Dans le cas du *Socket* de type *stream*, une fois l'appel effectué, l'état du *Socket* passe à *eTCPConnecting*. Par contre, dans le cas du *Socket* de type *datagram*, aucun mécanisme de connexion n'est requis par le protocole UDP : l'état du *Socket* passe donc directement à *eUDPConnected.* Lorsque le serveur accepte la connexion TCP, l'opération *ISocketMgr::SocketConnectedA()* est appelée pour signaler au client que l'opération est

réussie et que le *Socket* est maintenant dans un état *eTCPConnected*. Il est à souligner que l'opération est toujours réussie dans le cas d'un *Socket* de type *datagram* et que le client s'en trouve automatiquement notifié. À partir de ce moment, les opérations *ISocketMgr::SocketReadyToSendA()*, *ISocketMgr::ISocketReceivedA()* et *ISocketMgr::SocketReceivedOutOfBandA()* peuvent être appelées par l'objet *CAsyncSocket* afin de signaler au client que le *Socket* est prêt pour l'envoi de données ou encore, que des données ont été reçues. Advenant le cas où la connexion ne peut pas être établie, l'opération *ISocketMgr::SocketError()* est appelée pour signaler l'erreur.

```
void SendA(
    IN TO CBlob*   pData,
    IN EMsgFlags   flags = eNone);
```

Une fois la connexion établie, cette opération est utilisée pour envoyer un paquet de données au nœud distant. L'état du *Socket* doit être *eTCPConnected* ou *eUPDConnected*. Les octets à être transmis sont contenus à l'intérieur d'un *blob* à l'adresse pointée par *pData*. Le paramètre *flags*, qui peut prendre les valeurs suivantes, n'est utile que pour un *Socket* de type *stream*.

```
enum EMsgFlags
{
    eNone = 0,
    eOOB = MSG_OOB
};
```

Lorsque la transmission ne peut pas être effectuée correctement, le client en est informé via *ISocketMgr::SocketError()*.

```
void SendToA(
    IN TO CBlob*              pData,
    IN const CSocketAddr&     rPeerAddr);
```

Le comportement de l'opération *SendToA()* est similaire à l'opération *SendA()*; par contre, son utilisation est quelque peu différente. Premièrement, cette opération est utilisable uniquement avec un *Socket* de type *datagram*. Deuxièmement, le *Socket* doit être dans un état *eUDPListening*. Des paquets peuvent alors être envoyés vers de multiples destinations. C'est pour cette raison que l'opération comporte un paramètre indiquant la destination du paquet. Lorsque la transmission ne peut pas être effectuée correctement, le client en est informé via *ISocketMgr::SocketError()*.

```
CError CloseA();
```

L'opération *CloseA()* doit être appelée pour terminer proprement une connexion. Elle peut être invoquée dans n'importe quel état. Il en résulte toujours une transition vers l'état *eClosed* avec un appel à l'opération *ISocketMgr::SocketClosedA()*. Si le *Socket* est de type *stream*, celui-ci termine la connexion proprement (*shutdown*) et transite par l'état *eTCPClosing*, et ce, en attendant que tous les paquets soient transmis, pour ensuite transiter vers l'état *eClosed*. Lorsqu'une erreur survient, le client en est informé via *ISocketMgr::SocketError()*.

ISocketMgr

L'interface *ISocketMgr* doit être réalisée par le client du *Socket*. C'est par cette interface que le *Socket* avertit son client des différents événements qui surviennent. Par exemple, le *Socket* appelle l'opération *ISocketMgr::SocketReceivedA()* pour signaler la réception d'un paquet et le transmettre au client.

Il est bon de noter que chaque opération possède un paramètre *unSockID*. La valeur de ce paramètre est nécessaire au multiplexage des notifications en provenance des différents *Sockets* qu'un même client peut posséder.

Cette interface est constituée des opérations suivantes qui signifient que :

```
virtual void SocketConnectedA(
    IN UINT unSockID,
    IN const CSocketAddr& rPeerAddr) = 0;
```

Cette opération avertit le client que la connexion est établie avec le serveur situé à l'adresse *rPeerAddr*.

```
virtual void SocketAcceptedA(
    IN UINT unSockID,
    IN TO CAsyncSocket* pNewSock,
    IN const CSocketAddr& rPeerAddr) = 0;
```

Cette opération avertit le client qu'une nouvelle connexion est disponible. Le *Socket* associé à la nouvelle connexion est obtenue par le paramètre *pNewSock* et l'adresse du demandeur est spécifiée par l'adresse *rPeerAddr*.

```
virtual void SocketReadyToSendA(
    IN UINT unSockID) = 0;
```

Cette opération avertit le client que le *Socket* est prêt à envoyer des paquets sur le réseau. Cet appel survient après une connexion réussie. La notification peut aussi survenir après une erreur de transmission, une fois que le *Socket* est de nouveau fonctionnel.

```
virtual void SocketReceivedA(
    IN UINT unSockID,
    IN TO CBlob* pData,
    IN const CSocketAddr& rPeerAddr) = 0;
```

Cette opération avertit le client qu'un paquet a été reçu sur le port du *Socket*. Le paquet est transmis au client par le paramètre *pData*. Le client possède alors le *blob* et il se doit de le détruire une fois qu'il a terminé de l'utiliser.

```
virtual void SocketReceivedOutOfBandA(
    IN UINT unSockID,
    IN TO CBlob* pData,
    IN const CSocketAddr& rPeerAddr) = 0;
```

Cette opération est similaire à l'opération *SocketReceivedA()* à la différence que le paquet reçu a été transmis avec une priorité élevée (*out-of-band*). Elle n'est utilisée que par les *Sockets* de type *stream*.

```
virtual void SocketClosingA(
    IN UINT unSockID) = 0;
```

Cette opération avertit le client que le *Socket* distant amorce la fermeture de la connexion.

```
virtual void SocketClosedA(
    IN UINT unSockID) = 0;
```

Cette opération avertit le client que le *Socket* est maintenant fermé.

```
virtual void SocketErrorA(
    IN UINT unSockID,
    IN CError lastError) = 0;
```

Cette opération avertit le client chaque fois qu'une erreur survient pendant l'exécution d'une opération asynchrone.

CSocketAddr

Le format d'adressage du protocole IP est constitué d'un entier non signé de 32 bits. Cette adresse est généralement représentée en format ASCII par quatre nombres allant de 0 à 255 et qui sont séparés par un point. Par exemple : « 10.1.1.101 ». Chaque nœud est désigné par une adresse unique sur le réseau [36].

De plus, les protocoles de transport comme UDP et TCP ajoutent un deuxième identificateur unique à l'adresse. Cet identificateur est un nombre non signé de 16 bits appelé « port ». Il sert à identifier le *Socket* associé au message à l'intérieur du contexte du nœud désigné par l'adresse IP.

Le format le plus largement utilisé pour représenter une adresse complète (IP + port) est issu de l'interface BSD *Sockets*. Cette structure simple prend la forme suivante :

```
struct sockaddr {
    u_short    sa_family;
    char       sa_data[14];
};
```

Il est facile de constater que la structure *sockaddr* est très générique. Elle est ainsi faite pour s'adapter aux différents types de protocole pouvant être sélectionnés et contrôlés par l'API des BSD *Sockets* et qui ne sont pas nécessairement de nature IP.

La structure suivante, qui est adaptée aux besoins d'adressage des protocoles IP, vient définir le format de l'adresse *sockaddr* :

```
struct sockaddr_in {
    short     sin_family;
    u_short   sin_port;
  struct  in_addr {
    union {
      struct {
        unsigned char s_b1, s_b2, s_b3, s_b4;
      } S_un_b;
      struct {
        unsigned short s_w1, s_w2;
      } S_un_w;
      unsigned long S_addr;
    } S_un;
  } sin_addr;
    char sin_zero[8];
};
```

La manipulation de cette structure peut s'avérer fastidieuse et parfois complexe. Il est aussi peu recommandable d'utiliser cette structure comme donnée membre à cause de sa faible portabilité. Puisque DOES est bâti sur les protocoles réseaux IP, ses besoins d'adressage sont évidents. Il est donc important d'abstraire cette structure et de l'entourer de mécanismes de manipulations efficaces et adaptées.

Il est de la responsabilité de la classe *CSocketAddr* de voir à la manipulation efficace d'une adresse réseau dans l'environnement DOES. La donnée membre de cette classe est tout simplement un objet de type *socketaddr*. Les différentes opérations offertes par la classe encapsulent l'information et permettent de manipuler l'adresse. Son interface n'est pas décrite ici puisqu'elle est triviale.

7. Modélisation et vérification

Plusieurs méthodes existent pour la vérification des propriétés d'un système, mais à ce jour aucune n'est parfaite. Il faut habituellement avoir recours à plus d'une méthode pour couvrir l'ensemble des propriétés d'un système. Bien que simple, le protocole DOOM possède un espace d'états relativement vaste. De plus, à l'intérieur de cet espace, il faut inclure les états de transmission d'un réseau non fiable. Il est donc assez difficile d'explorer entièrement cet espace avec des techniques standards faisant appel à des tests unitaires ou encore à des tests de type boîte noire effectués sur un système complet et fonctionnel. Afin de s'assurer de la validité du protocole DOOM et de l'architecture des classes qui si rattachent, deux méthodes de vérification sont utilisées.

La première méthode retenue, appelée *Model-Checking*, a les caractéristiques nécessaires pour vérifier la validité de la spécification du protocole, plus particulièrement des règles de transmission des messages d'action et de réaction. La section 7.1 de ce chapitre explique cette méthode et les résultas obtenus. Malheureusement, il est pratiquement impossible de modéliser entièrement le protocole. Les outils de modélisation d'aujourd'hui manquent de maturité, ce qui limite la complexité du modèle pouvant être construit. C'est pour cette raison qu'une seconde méthode a aussi été retenue.

La seconde méthode retenue est la méthode classique de prototypage : elle est utilisée afin de pouvoir vérifier d'autres propriétés importantes du protocole. Le prototype qui est développé intègre certains services de base de DOOM qui sont à vérifier. Des tests d'intégration de type boîte noire peuvent alors être exécutés à l'intérieur d'un environnement réel et contrôlé. La section 7.2 de ce chapitre explique ce prototype ainsi que les propriétés qu'il vérifie.

7.1 Vérification de la spécification du protocole de transmission

Le *model-checking* est une méthode de vérification très exhaustive. Elle se base sur la spécification formelle d'un système et ignore l'implémentation : elle énonce ses propriétés et effectue l'exploration de son espace d'état. Cette méthode se prête bien aux systèmes critiques, distribués et réactifs. Comme l'exploration de l'espace d'état d'un système est généralement très complexe, cette technique doit être assistée par ordinateur. Plusieurs outils permettent de faire la vérification. Celui qui est utilisé pour vérifier DOOM se nomme Uppaal.

Avant de passer à la modélisation et à la vérification du protocole, la prochaine section résume les fonctions de l'outil ainsi que certaines notions du *model-checking*.

7.1.1 Uppaal et le *model-checking*

L'outil Uppaal se situe dans la lignée des outils de *model-checking* pour la vérification des systèmes temps réel, des contrôleurs temps réel et des protocoles de communication. Il est disponible gratuitement via Internet[27] et fonctionne sur un environnement Windows et SunOS.

Les modèles d'Uppaal sont des réseaux d'automates temporisés d'Alur et Dill [44] [45] étendus avec certains types de variables. Un automate est un modèle de machine évoluant d'état en état sous l'effet de transitions. Un automate **A** est défini formellement comme un quintuplé **A** = < **Q,E,T,q0, l** > où :

Q = ensemble fini d'états;
E = ensemble d'étiquettes de transition;
$\mathbf{T} \subseteq \mathbf{Q} \times \mathbf{E} \times \mathbf{Q}$ = ensemble des transitions;
$\mathbf{q_0}$ = état initial de l'automate;
l = application qui associe à tout état de **Q** l'ensemble fini de propriétés élémentaires vérifiées dans cet état.

Les automates temporisés sont des automates finis constitués d'un ensemble fini de variables d'horloges, elles-mêmes utilisées pour spécifier les contraintes quantitatives de temps. Les horloges sont en fait des variables réelles. Les automates temporisés d'Uppaal ajoutent aux variables d'horloge des variables entières, pour rendre le langage de modélisation plus expressif. Le domaine des valeurs pour Uppaal doit, par contre, être fini pour garantir la terminaison de la procédure de vérification. À l'intérieur d'Uppaal, le temps progresse d'une façon continue et les transitions sont instantanées. Toutes les horloges progressent avec la même vitesse. Les automates ont le choix d'exécuter la transition permise, ou de rester dans le même état, si aucun invariant d'état ne force la transition.

Au niveau d'une transition entre deux états, il est possible d'ajouter des gardes. Les gardes expriment les conditions ou les contraintes sur les horloges et sur les variables entières qui doivent être satisfaites pour décider de la transition à effectuer (conditions de franchissement). Par exemple, la garde qui veut que le temps indiqué par l'horloge x soit plus grand que 2 pour autoriser une transition s'exprime par : $x > 2$. Quant aux invariants, ils expriment des contraintes de temps à l'intérieur d'un état. Ils obligent une transition vers un autre état, une fois un des délais expirés. Un invariant prend la forme $x \sim n$ ou $\sim \in \{<, \leq\}$. Sur une transition, il est possible d'affecter les variables de données et les horloges en terme d'assignations ou ré-initialisations. Les assignations sont évaluées séquentiellement. Pour une transition synchronisée, les assignations sur « ! » sont évaluées avant « ? ». Les *commited locations* (état avec un C) n'allouent aucune transition de délai, seules les transitions d'action sont permises.

La communication entre les différents automates du système se fait par la déclaration de canaux. La communication est toujours réalisée à travers une fonction de synchronisation binaire, c'est-à-dire que les automates communiquent entre eux, deux par deux, selon un modèle d'envoi/réception de messages. L'action « a! » correspond à une émission sur ce canal, tandis que

[27] http://www.uppaal.com

l'action « a? » correspond à une réception sur ce canal. Le modèle défini par Uppaal ne permet pas l'exécution simultanée de plus de deux actions et aucun paramètre ne peut être échangé via le canal.

La description du système à vérifier avec Uppaal comprend un ensemble d'automates ainsi qu'une déclaration textuelle des variables globales, des canaux de communications et de la définition des processus. Avec Uppaal, les automates sont décrits en terme de *template* à l'aide d'une interface graphique. L'instance d'un automate est désignée par le terme processus et elle s'obtient à partir d'un *template* et de paramètres d'initialisation.

Le principe de base du *model-checking* consiste à vérifier certaines propriétés du système étudié qui sont d'ordre critique. L'outil, à partir d'une expression logique, explore l'espace d'états du modèle afin de vérifier si la propriété énoncée est valide ou non. La logique utilisée par Uppaal, pour énoncer formellement les propriétés portant sur l'exécution d'un système, est un fragment d'une logique temporelle plus complète appelée CTL* (Computation Tree Logic). La logique CTL*, qui a été introduite par Emerson et Halpern [42] [43], contient les éléments suivants :

- **Propositions atomiques :** elles portent sur les états de contrôle et les valeurs des variables; S.e exprime que l'automate S est dans l'état e possédant les variables V de valeur (V ~ n) et les horloges h de valeur (h ~ n); V est une variable entière, h est une horloge, n ∈ N et ~ ∈{ =, <,>,≤,≥}.

- **Combinateurs booléens :** Ils permettent la négation (¬), la conjonction (∧), la disjonction (∨); l' implication logique (⇒) et la double implication ou équivalence (⇔).

- **Formules propositionnelles :** Ils permettent la combinaison de propositions et de combinateurs logiques; e.g : froid ⇒ ¬ chaud.

- **Combinateurs temporels F et G :** ils permettent d'exprimer des états le long d'une exécution et non pas des états individuels; FΦ veut dire qu'un état futur vérifie Φ, sans préciser quel état; GΦ exprime que tous les états futurs vérifient Φ.

- **Quantificateurs de chemin A et E :** ils permettent d'exprimer le côté arborescent du comportement et aussi de quantifier l'ensemble des exécutions; AΦ veut dire que toutes les exécutions partant de l'état courant satisfont la propriété Φ; EΦ veut dire qu'il existe une exécution satisfaisant Φ.

Malheureusement, Uppaal ne permet pas d'utiliser les combinateurs individuellement et les seules combinaisons possibles sont AGΦ et EFΦ[28]. La logique temporelle d'Uppaal permet donc de vérifier des propriétés **d'atteignabilité** (EFΦ) et de **sûreté** (AGΦ) : EFΦ signifie qu'on peut atteindre une configuration vérifiant Φ (atteignabilité) et AGΦ signifie que toutes les configurations atteignables vérifient Φ (sûreté).

[28] Avec la syntaxe d'Uppaal, EFΦ s'écrit E<>Φ et AG Φ s'écrit A[]Φ.

De plus, la négation de ces types de propriétés peut être très intéressante. Par exemple, la propriété d'accès unique à une section critique CS par deux processus concurrents S1 et S2 peut s'énoncer de la façon suivante : $\neg EF(S1.CS \wedge S2.CS)$. Ainsi, la négation d'une propriété d'atteignabilité (non-atteignabilité) correspond à une propriété de sûreté, car s'il est impossible de trouver au moins un chemin d'exécution avec un état satisfaisant l'atteinte simultanée de la section critique CS par les deux processus S1 et S2, il est évident que la propriété $AG\neg(S1.CS \wedge S2.CS)$ est aussi vrai.

Le *model-checking* d'Uppaal est symbolique. C'est-à-dire qu'il s'intéresse à représenter de façon symbolique, par opposition à explicite, les états et les transitions de l'automate global. Cette technique permet de répondre au problème de l'explosion combinatoire des états du système se produisant, chaque fois qu'il est question de représenter explicitement tous les états de l'automate global examiné. L'idée, dans la représentation symbolique, est de considérer qu'un automate se comportera de manière identique dans un sous-ensemble des états atteignables dont le nombre tend vers l'infini. Ce sous-ensemble permet alors de définir un espace d'états symboliques fini et explorable.

7.1.2 Modélisation des automates

Afin de pouvoir vérifier les propriétés du protocole à l'aide du logiciel Uppaal, il est nécessaire de le modéliser. La modélisation du protocole à partir de sa spécification informelle est directe. Pour simplifier le modèle, l'hypothèse suivante est faite : le lien entre le client et le serveur est établi. De plus, puisque chaque lien est traité indépendamment par le protocole, le modèle est limité à l'échange de messages sur un lien unique entre deux objets. Cette contrainte simplifie grandement le modèle tout en permettant la vérification complète des règles de transmission.

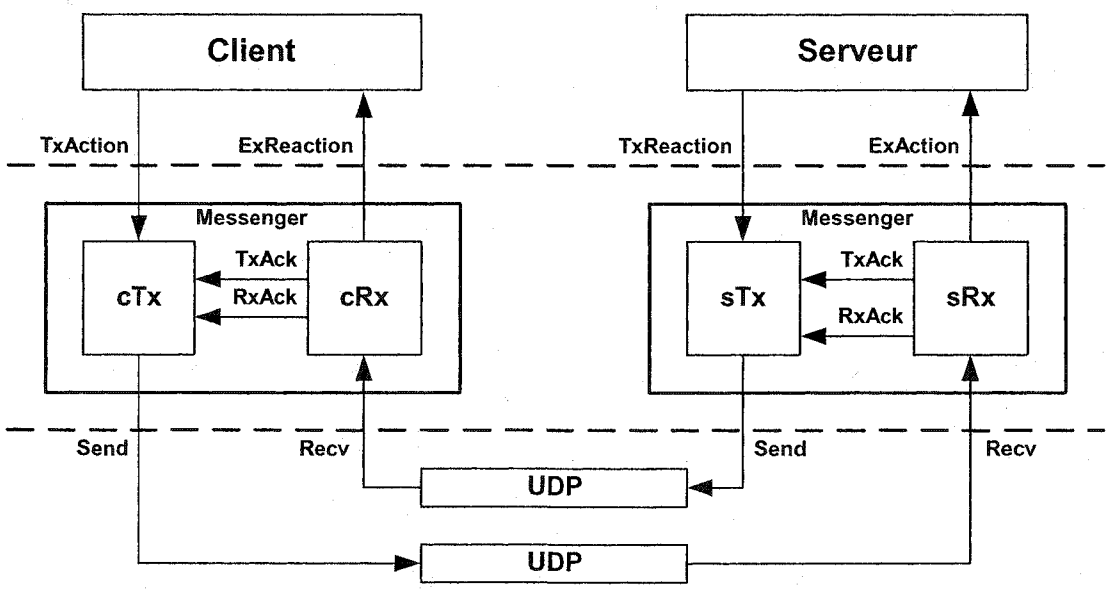

Figure 7.1 Schéma bloc du protocole DOOM

La Figure 7.1 illustre les éléments qui sont modélisés. Chaque bloc représente un processus spécifié par un automate temporisé. Le bloc **Client** correspond à un objet de type client et le bloc **Serveur** correspond à l'objet serveur avec lequel le client est lié. Les blocs **cTx** et **cRx**

représentent les automates de transmission et de réception pour un client, tandis que **sTx** et **sRx** représentent ceux du serveur. Les blocs **UDP** simulent les canaux de transport de *datagrams*. Les flèches, quant à elles, représentent les canaux de communication pour la synchronisation des automates. **TxAction** synchronise **cTx** et **Client** pour la transmission d'une action. **TxReaction** synchronise **sTx** et **Serveur** pour la transmission de la réaction. **ExAction** et **ExReaction** demandent l'exécution de l'action et de la réaction. Les canaux **TxAck** et **RxAck** servent à synchroniser le transmetteur avec le récepteur lors de la réception de nouveaux messages. Le canal **Send** indique au processus **UDP** quand transmettre un *datagram* et le canal **Recv** indique au récepteur quand un *datagram* est reçu. Les diagrammes suivants représentent chacun des automates temporisés du système. La signification des variables, constantes, gardes, canaux, etc. se retrouve à l'intérieur de la section 5.10 du protocole DOOM. Les noms utilisés à l'intérieur des automates du modèle sont les mêmes que ceux du protocole, ce qui facilite la compréhension et le repérage de l'information. Les seules variables ajoutées au niveau des transmetteurs sont les horloges *ct* et *cr*. L'horloge *ct* sert à calculer le délai depuis la dernière transmission, tandis que *cr* maintient le temps total depuis la première transmission.

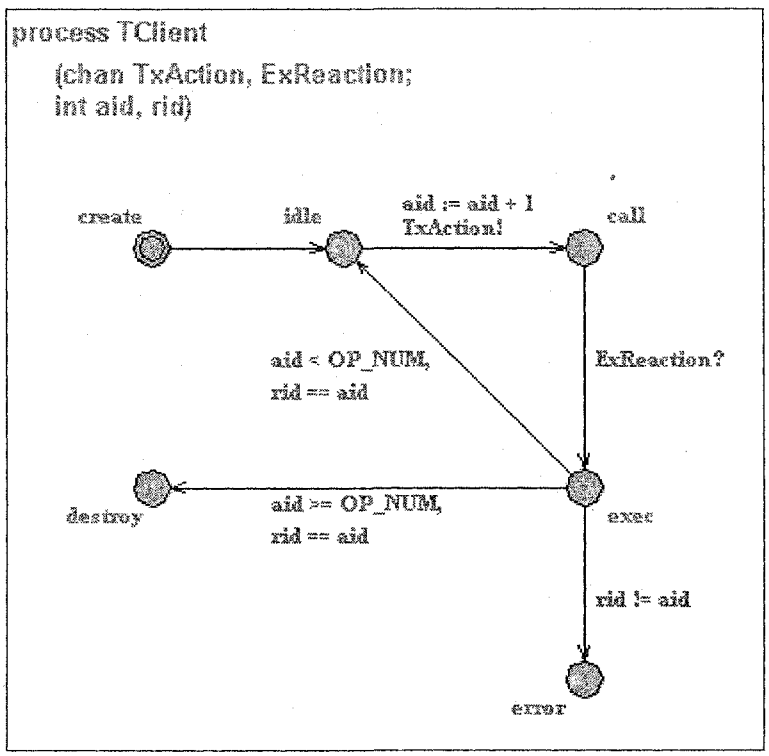

Figure 7.2 Processus TClient

L'automate du client demande la transmission d'une action vers le serveur. L'action est représentée par un identificateur qui s'incrémente de un pour chaque nouvelle action. Le client attend ensuite que la réaction lui parvienne. Au moment de l'exécution de la réaction, son identificateur est comparé à celui de l'action précédemment transmise. Si les deux identificateurs correspondent, le client passe à la prochaine action; autrement, l'état erreur est atteint.

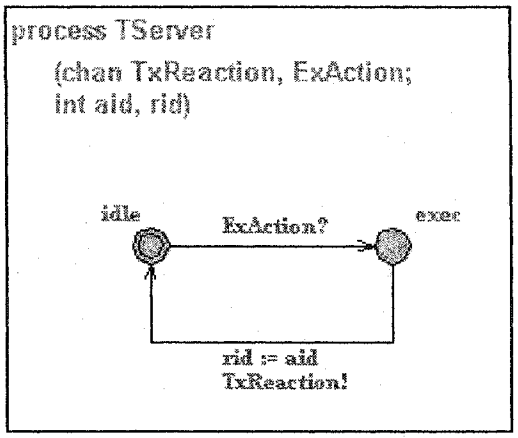

Figure 7.3 Processus TServer

L'automate du serveur est très simple et ne contient que deux états de contrôle. Pour chaque action exécutée, il demande la transmission d'une réaction. Le numéro d'identification de l'action reçue est assigné à la réaction.

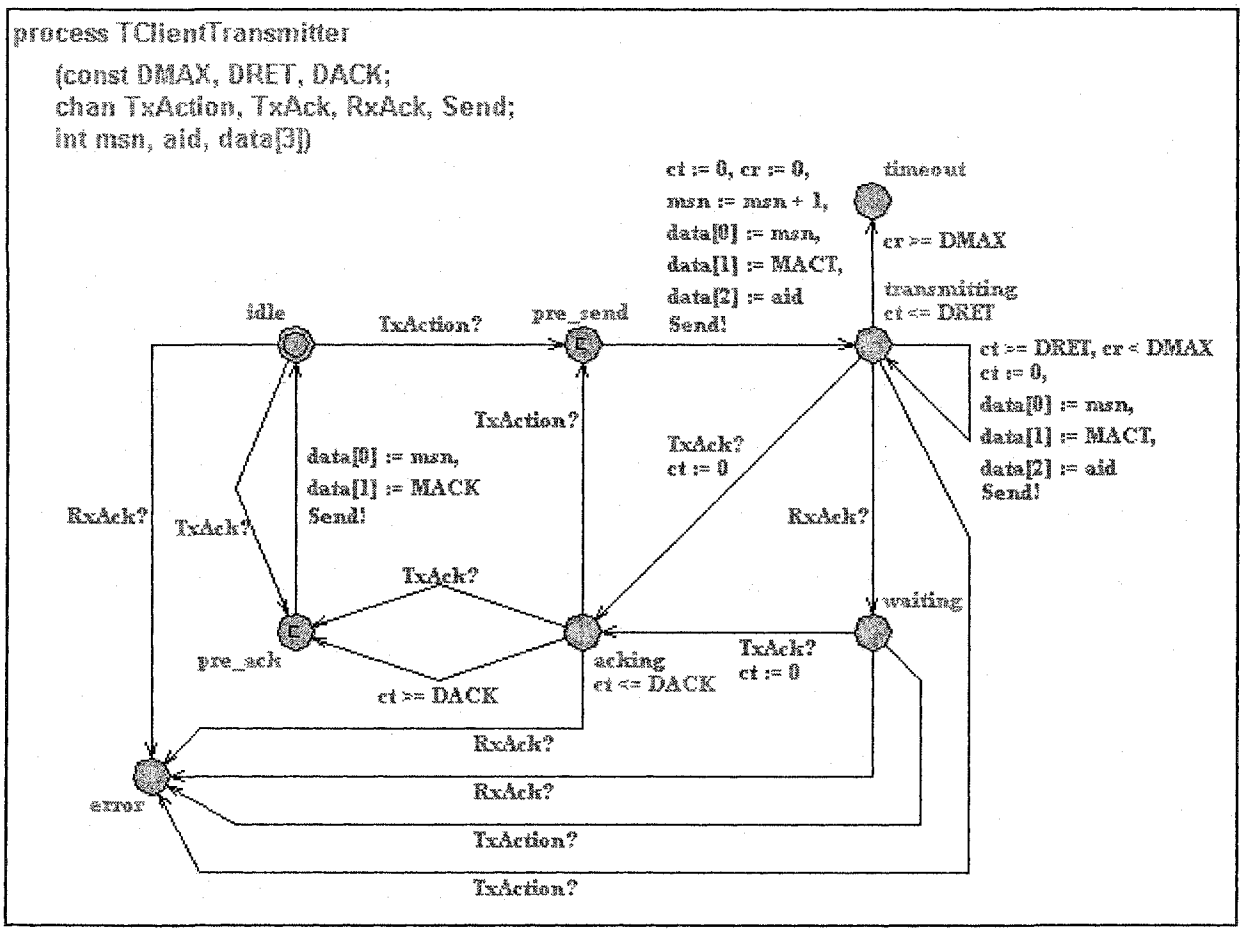

Figure 7.4 Processus TClientTransmitter

L'automate *TClientTransmitter*, bien que décrit avec la syntaxe graphique d'Uppaal, est pratiquement identique à l'automate présenté à la Figure 6.4 avec le langage UML. Les états *timeout* et *error* sont ajoutés pour permettre d'énoncer les propriétés de sûreté.

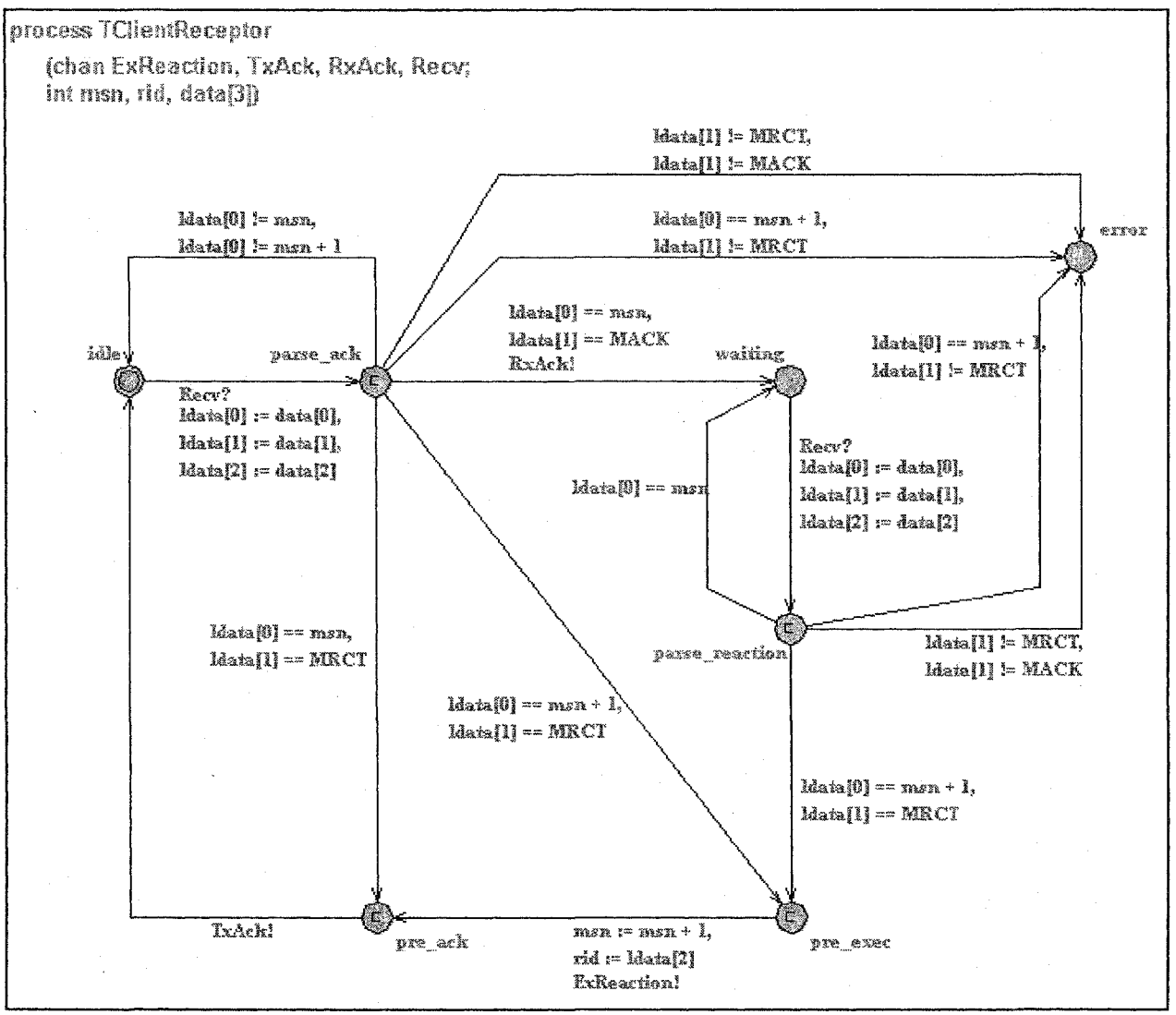

Figure 7.5 Processus TClientReceptor

L'automate *TClientReceptor* est pratiquement identique à l'automate présenté à la Figure 6.6 avec le langage UML. L'état *error* est ajouté pour permettre d'énoncer les propriétés de sûreté.

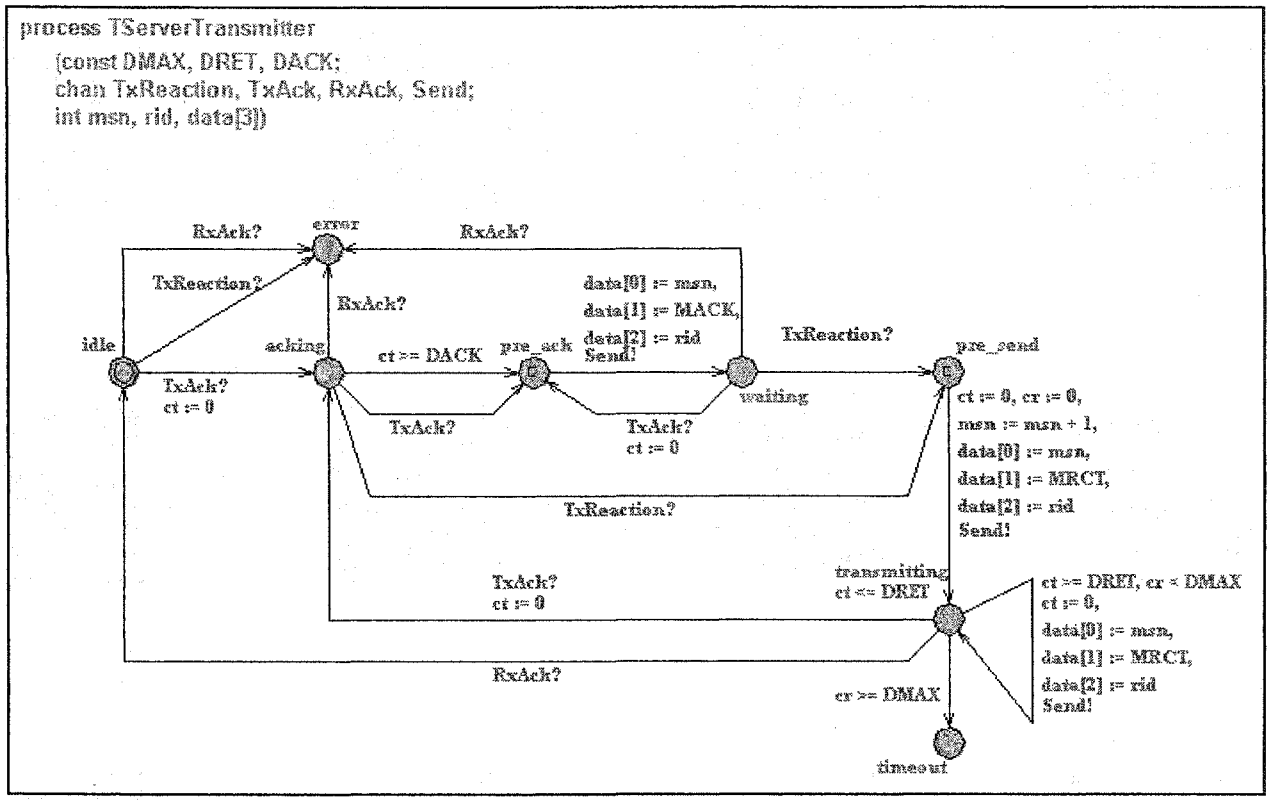

Figure 7.6 Processus TServerTransmitter

L'automate *TServerTransmitter* est pratiquement identique à l'automate présenté à la Figure 6.5 avec le langage UML. Les états *timeout* et *error* sont ajoutés pour permettre d'énoncer les propriétés de sûreté.

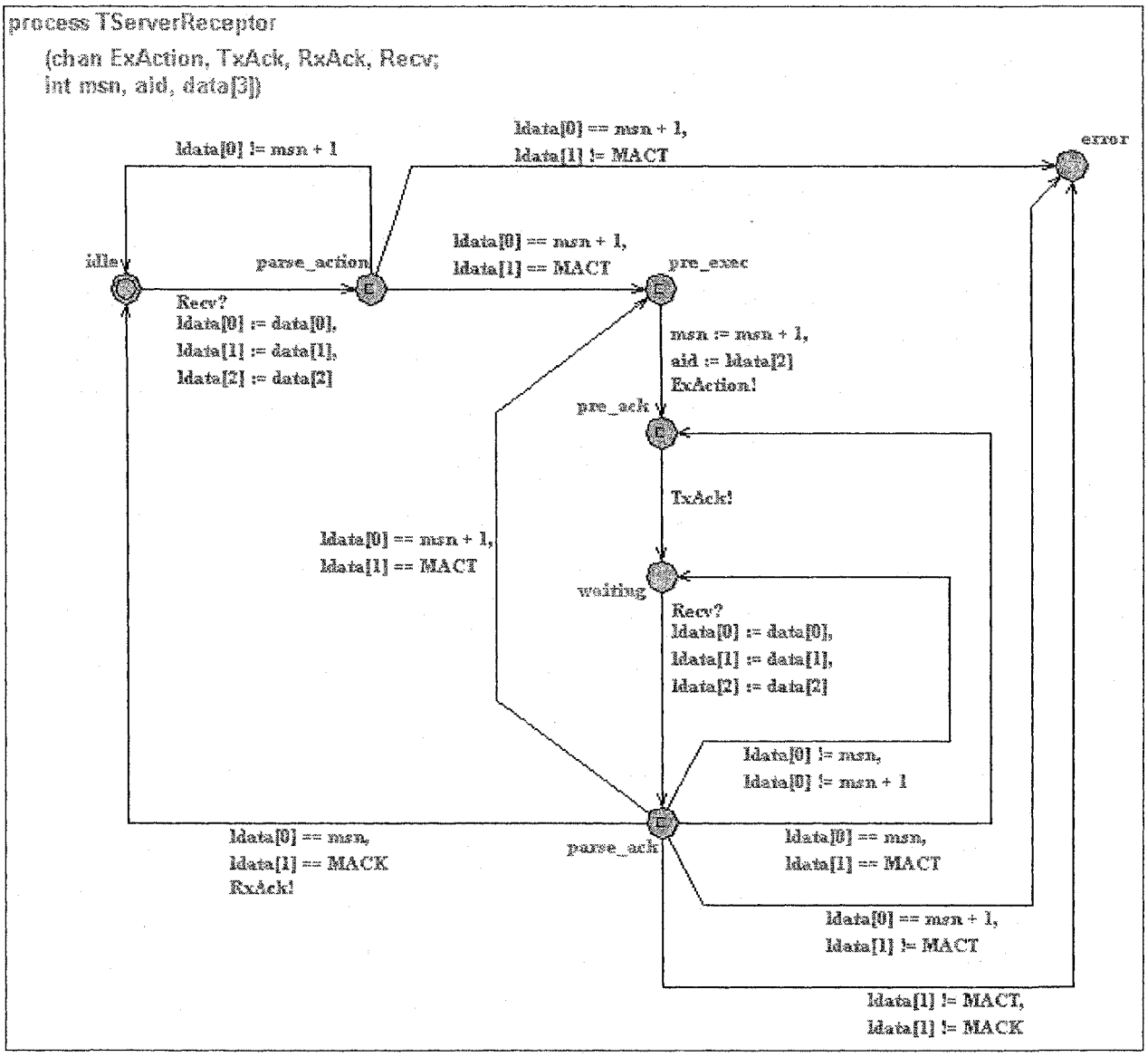

Figure 7.7 Processus TServerReceptor

L'automate *TServerReceptor* est pratiquement identique à l'automate présenté à la Figure 6.7 avec le langage UML. L'état *error* est ajouté pour permettre d'énoncer les propriétés de sûreté.

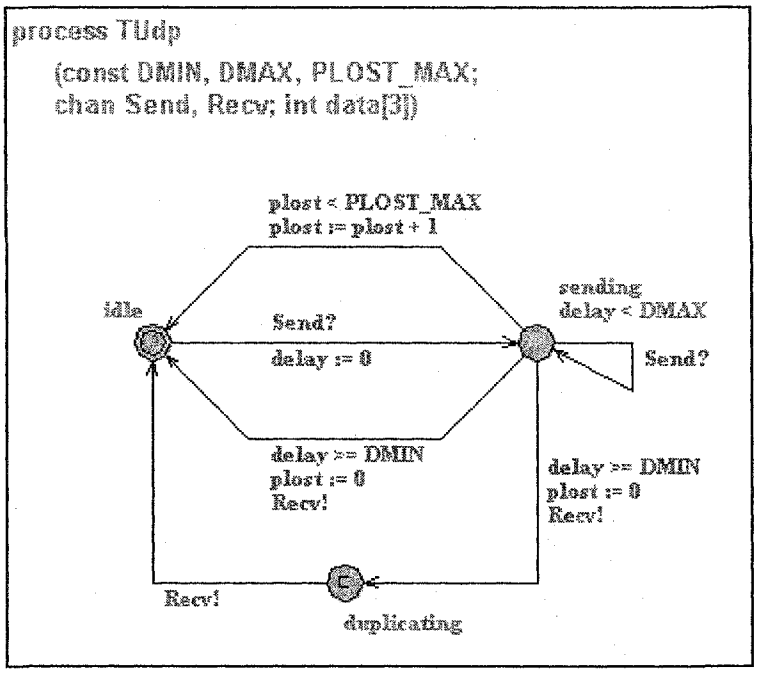

Figure 7.8 Processus TUdp

L'automate TUdp modélise la couche de transport. Chaque canal UDP est paramétrable en trois points : premièrement, il est possible de créer et de limiter les pertes des paquets; deuxièmement, les délais de transmission minimal et maximal peuvent être ajustés; troisièmement, une duplication aléatoire de paquets peut être introduite. Le canal permet aussi de transporter trois données entières. Ce sont ces données, à l'intérieur de la variable globale *data*, qui sont utilisées lors de la validation des messages à l'intérieur des automates de transmission et de réception. Les données qui s'y retrouvent sont dans l'ordre suivant : le numéro de séquence du message, le type d'action et l'identificateur d'action. Cet identificateur correspond à une opération sérialisée.

7.1.3 Vérification des automates

Afin de vérifier le protocole, des états spéciaux ont été utilisés. Ces états sont : pour le client, l'état **destroy** et l'état **error;** pour les transmetteurs, l'état **error** et **timeout**; pour les récepteurs, l'état **error**. Selon les conditions du réseau, ces états devaient ou ne devaient pas être atteignables.

Avec des conditions de transport acceptables, les propriétés suivantes sont vérifiées :

- **E<> client.destroy** : il existe au moins un chemin d'exécution pour lequel l'automate « client » peut atteindre l'état « destroy »;
- **A[] not (client.error or cTx.error or cRx.error or sTx.error or sRx.error)** : parmi tous les chemins d'exécution, il n'en existe aucun par lequel l'un ou l'autre des automates « client, cTx, cRx, sTx, sRx » puissent atteindre l'état « error »;
- **A[] not (cTx.timeout or sTx.timeout)** : parmi tous les chemins d'exécution, il n'en existe aucun par lequel les automates « cTx » ou « sTx » puissent atteindre l'état « timeout ».

Les conditions de transport testées allaient de parfaites, c'est-à-dire sans perte de paquets et sans aucun délai, jusqu'à pauvres, c'est-à-dire avec 75 % de perte de paquets et avec un délai de propagation maximal situé sous la constante DMAX. Dans tous les cas, les trois propriétés précédentes sont satisfaites.

Avec des conditions de transport extrêmement mauvaises, dépassant les pires conditions précédentes, les propriétés suivantes sont vérifiées :

- A[] not client.destroy : parmi tous les chemins d'exécution, il n'en existe aucun par lequel l'automate « client » puisse atteindre l'état « destroy »;
- A[] not (client.error or cTx.error or cRx.error or sTx.error or sRx.error) : parmi tous les chemins d'exécution, il n'en existe aucun par lequel l'un ou l'autre des automates « client, cTx, cRx, sTx, sRx » puisse atteindre l'état « error »;
- E[] (cTx.timeout or sTx.timeout) : il existe au moins un chemin d'exécution pour lequel l'un ou l'autre des automates « cTx » ou « sTx » puissent atteindre l'état « timeout ».

Les conditions de transport extrêmes correspondaient à 100 % de perte de paquets ou à un délai de propagation minimal situé au-dessus de la constante DMAX. Dans tous les cas, les trois propriétés précédentes sont satisfaites.

La vérification de ces six propriétés du modèle démontre donc que le protocole de transmission, défini par DOOM, permet une transmission fiable des messages d'action-réaction, et ce, via un protocole de transport non fiable, comme UDP.

7.2 Vérification par prototypage

Bien que plusieurs tests unitaires aient été réalisés pour s'assurer du bon fonctionnement des classes de DOES, cette étape n'est pas suffisante. Afin de pousser plus loin la vérification, il est nécessaire d'avoir recours à des tests d'intégration, effectués sur un prototype réel. L'idée, derrière le prototype, est de construire une application distribuée en se basant sur les services de DOES et de son protocole DOOM, et d'exécuter des tests de type « boîte noire » sur cette même application distribuée. L'application qui sert à réaliser cette vérification est un **répondeur téléphonique distribué :** la première partie du répondeur, utilise les ressources matérielles spécialisées d'un système embarqué, nommé APA, pour interfacer avec le PSTN (*Public Switched Telephone Network*); la seconde partie du répondeur, qui gère l'automate de réponses et l'archivage des messages, s'exécute sur un PC.

L'APA[29] est un appareil de télécommunications permettant d'établir jusqu'à quatre communications téléphoniques, voix ou fax, via un réseau IP (VoIP). Il est constitué principalement d'un processeur Motorola PowerPC MPC860T@50MHz, d'une interface Ethernet 10/100 Base-T, de 16 Mo RAM et de 2 Mo Flash ROM. Le système d'exploitation

[29] Le modèle exact de l'appareil est APA-III-4FXS/FXO.

temps réel est appelé eCos et il est fabriqué par la compagnie Red Hat[30]. Plus précisément, l'APA est un adaptateur capable de convertir les signaux d'appareils analogiques en données numériques, pour leur transport sur un réseau comme l'Internet. De plus, l'APA possède une option de configuration au niveau du port 4 : ce port peut agir soit comme port FXS (*Foreign eXchange Subscriber*), soit comme port FXO (*Foreign Exchange Office*). Aux ports de type FXS, il est possible de brancher un téléphone analogique ou un fax. Par contre, lorsque le port 4 est configuré en mode FXO, il faut le relier au PSTN : il est alors possible d'obtenir une connexion avec le CO (*Central Office*) et d'émettre ou de recevoir un appel téléphonique. Donc, afin de réaliser le répondeur téléphonique distribué, seul le port 4 en mode FXO est requis. Les autres ports ne sont pas activés. L'APA supporte la détection et la génération de DTMF, la détection de sonnerie, la génération de tonalités et la compression de la voix. Ces services sont assurés par un processeur spécialisé appelé DSP (*Digital Signal Processor*). Dans le cas du répondeur, c'est l'algorithme G.711 u-Law avec un débit de 64 Kbps qui est utilisé pour la compression de la voix et son transfert vers le PC.

Le PC est un ordinateur personnel constitué d'un processeur Intel Pentium-III@800MHz, de 256 Mo RAM et d'un disque dur de 32 Go. Le système d'exploitation est Windows 2000 de Microsoft. Contrairement au processeur PowerPC de l'APA, le processeur Intel du PC est basé sur la convention Big-Endian, pour l'ordre des octets en mémoire.

Le répondeur est principalement constitué des quatre classes suivantes : *CAnalogLine*, *CAnswerMachine*, *CVoiceMessage* et *CAnswerMenu*. La classe *CAnalogLine* abstrait les périphériques associés au port FXO, et offre tous les services de base qui sont disponibles sur une ligne analogique de type PSTN. Les opérations *Take()* et *Release()* permettent d'ouvrir et de fermer une connexion avec le CO, et à plus haut niveau, elles recréent les états *on-hook* et *off-hook* d'un téléphone qui serait branché à la ligne : de la même façon, l'opération *Flash()* génère un *flash-hook*[31] sur la ligne. Quant aux opérations *PlayTone()* et *StopTone()*, elles servent à la génération de tonalités. Finalement, l'opération *Dial()* génère une suite de DTMF. La classe *CAnswerMachine* recrée le comportement d'un répondeur. Elle gère les étapes de réponse, d'enregistrement et d'écoute. L'association qui existe entre *CAnalogLine* et *CAnswerMachine* est bidirectionnelle. C'est qu'en plus de commander la ligne, l'automate du répondeur reçoit les événements en provenance du port FXO. De plus, la classe *CAnswerMachine* est composée des classes *CVoiceMessage* et *CAnswerMenu* : ces deux classes sont responsables de l'archivage des messages vocaux à l'intérieur du système de fichiers. Il est certain que les classes d'un répondeur, telles que présentées à la Figure 7.9, peuvent résider à l'intérieur de l'APA. Par contre, comme l'APA est limité en capacité de stockage, il est impossible d'y archiver plusieurs messages, en plus du message d'accueil.

[30] eCos a été développé par Cygnus Solution qui appartient maintenant à Red Hat.

[31] L'événement *flash-hook* est généré par une transition de l'état *off-hook* à l'état *on-hook* suivie d'une transition vers l'état *off-hook* à l'intérieur d'un délai d'une seconde.

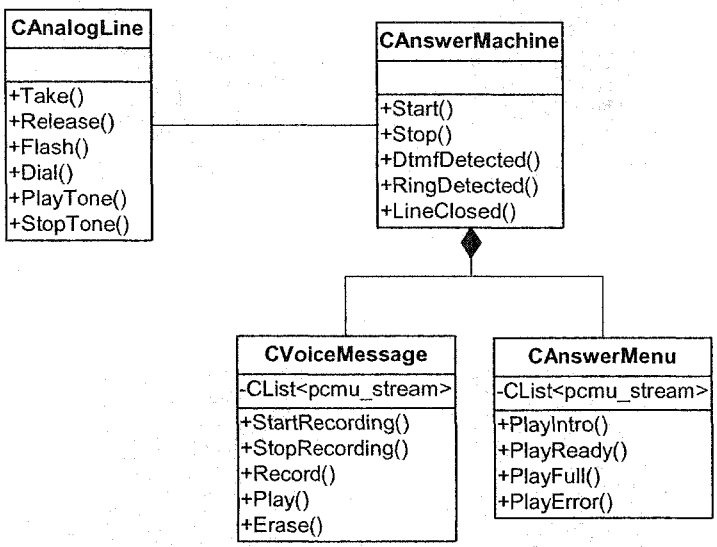

Figure 7.9 Répondeur téléphonique -- diagramme statique

Pour augmenter la capacité de stockage ainsi que pour faciliter la gestion de l'automate du répondeur, les classes *CAnalogLine* et *CAnswerMachine* sont distribuées selon la Figure 7.10. L'emplacement réel des objets est abstrait grâce aux *Proxy*, *Adapter* et *Messenger*.

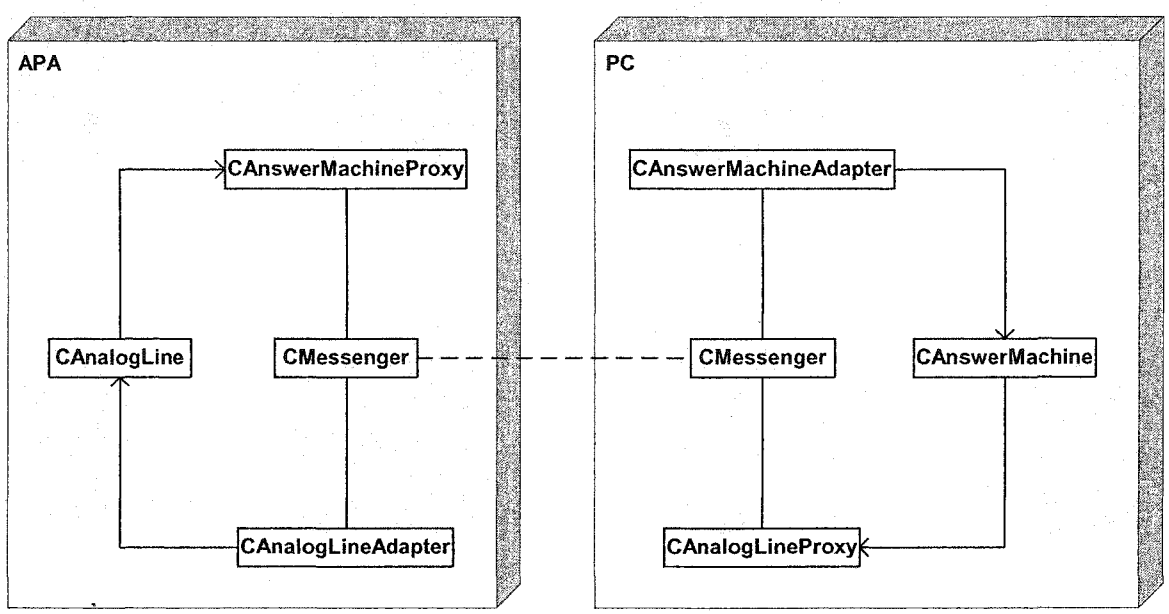

Figure 7.10 Répondeur téléphonique distribué -- diagramme statique

Puisque l'association entre les classes *CAnalogLine* et *CAnswerMachine* est bidirectionnelle, il est nécessaire de recourir à deux liens virtuels entre leurs objets : dans un cas, l'instance de *CAnswerMachine* joue le rôle de client et elle utilise les services de l'instance de *CAnalogLine*; dans l'autre cas, les événements du port FXO sont transmis au répondeur qui joue le rôle de serveur. C'est pourquoi il existe un *Proxy* et un *Adapter* pour chacune des classes; le code source de ces classes se retrouve en annexe 3.

Cette architecture de test, bien que simple, permet de vérifier la majorité des fonctionnalités de DOES. Le scénario le plus complet est, bien entendu, la réception d'un appel avec message d'accueil, suivi de l'enregistrement du message vocal. Pour ce faire, il faut tout d'abord que les liens soient établis entre l'APA et le PC : les services de liaison et de création du protocole DOOM sont requis pour y arriver. Ensuite, lorsqu'un appel est reçu, l'événement *RingDetected* est transmis au répondeur par l'appel d'une opération asynchrone. Par la suite, l'objet du répondeur qui reçoit l'événement, agit comme client et fait appel à l'opération synchrone *Take()* de la ligne analogique, pour accepter la communication. Finalement, cet objet du répondeur fait entendre le message d'accueil, et termine en enregistrant la voix de l'appelant, puis en relâchant la ligne.

Il est intéressant de noter que le système du répondeur téléphonique, tel que présenté, fonctionne à l'intérieur d'un environnement hétérogène. Cependant, les systèmes d'opération, les processeurs et l'ordre des octets en mémoire ne sont pas les mêmes à l'intérieur des deux nœuds : c'est pourquoi les règles de *marshaling* du protocole DOOM sont exploitées. De plus, il est démontré que les classes de DOES sont portables, grâce aux couches d'abstraction, et qu'elles peuvent s'intégrer à différentes plates-formes.

Une fois opérationnel, ce système de répondeur téléphonique distribué a été soumis à une série d'appels : tous les messages ont été enregistrés avec succès et il fut impossible de faire échouer le système! À partir du PC, il était possible d'écouter les messages enregistrés sur le disque dur. Les résultats de ce test sont donc très satisfaisants et ils démontrent la pleine fonctionnalité de DOES dans un environnement réel. Bien entendu, ces tests ne vérifient pas tous les aspects du protocole, mais ils sont suffisants pour offrir la preuve que les concepts de DOES sont valides et fonctionnels.

8. Conclusion

Malgré l'avancement des technologies pour la distribution des objets, leurs applications au domaine des systèmes embarqués restent limitées. Même si les besoins des systèmes embarqués sont généralement moins grands, les problématiques de base de la distribution restent les mêmes. Malheureusement, les contraintes matérielles de tels systèmes limitent l'utilisation des solutions existantes, ce qui est incommodant pour les systèmes embarqués désirant s'intégrer à un réseau d'objets distribués.

Le projet présenté à l'intérieur de ce document avait pour objectif premier, le développement d'un protocole de distribution d'objets respectant les contraintes des systèmes embarqués. Le second objectif était de concevoir une architecture de classes faisant abstraction du protocole et facilitant le développement d'applications distribuées. Le protocole DOOM et les classes comme *CMessenger*, développés à l'intérieur du projet DOES, offrent une infrastructure logicielle avec des solutions réelles aux problématiques des objets distribués pour systèmes embarqués.

Les architectures de classes permettent la construction rapide d'applications à objets distribués, grâce au concept d'héritage, et elles offrent une solution pratique d'utilisation, basée sur le langage de programmation C++. De plus, les services de l'OS et de la pile TCP/IP sont isolés par des couches d'abstraction, ce qui facilite grandement l'adaptation des classes de haut niveau : le développement de systèmes qui font intervenir des plates-formes hétérogènes s'en trouve grandement facilité. Par surcroît, la distribution des objets peut s'accomplir de façon transparente lors d'une seconde phase : par exemple, il est possible de distribuer les services d'une classe déjà existante, sans avoir à la modifier, c'est-à-dire qu'il faut simplement créer les versions spécialisées des classes *CObjectProxy* et *CObjectAdapter* pour y arriver.

Le protocole DOOM est, quant à lui, simple et efficace et il peut s'adapter à des systèmes qui possèdent un nombre élevé d'objets. Il offre tous les services de base nécessaires à la création et à la destruction d'objets. Ses services permettent aussi l'appel d'opérations synchrones et asynchrones. Les règles que ce protocole définies pour le *marshaling* couvrent tous les types de signatures d'opération. Enfin, sa logique pour le transport des messages est basée sur un concept d'action-réaction qui limite les différents états du protocole, ce qui simplifie son implémentation.

Les techniques de vérification employées ont permis de démontrer la validité du projet DOES. De plus, la modélisation du protocole par des automates et la vérification de leurs propriétés, ont su prouver la fiabilité de l'algorithme de transmission des messages. En supplément, le prototype du répondeur distribué a apporté une preuve tangible du potentiel de DOES à l'intérieur d'un environnement réel.

Il faut noter que l'application du répondeur téléphonique est un premier pas important qui démontre la faisabilité d'un système embarqué et distribué. Suite à cette expérimentation, le protocole DOOM et les classes de DOES pourraient être utilisés à l'intérieur de produits commerciaux de téléphonie IP : ces produits sont développés par la compagnie Médiatrix Télécom, Inc. située à Sherbrooke. Des tests plus poussés seront réalisés en laboratoire afin de s'assurer que tous les services de DOES soient sans faille et qu'ils répondent adéquatement aux

besoins des applications. Si tel est le cas, la fonctionnalité des APA pourra être étendue, grâce aux objets distribués!

Bibliographie

[1] Comité des études supérieures, (Janvier 1990) *Protocole de Rédaction et de Dépôt d'un Essai, d'un Mémoire ou d'une Thèse*, Canada, Université de Sherbrooke, Faculté des Sciences Appliquées, 30 p.

[2] Cherkaoui, S. (Décembre 1996) *DOC, Thèse de doctorat*, Canada, Université de Sherbrooke, Faculté des Sciences Appliquées, Département de Génie Électrique, 104 p.

[3] Anderson, M. (Avril 1999) *CORBA Load Balancing with VisiBroker*, C/C++ User Journal, United-States, vol. 17, n. 4, p. 55-64.

[4] Rousselle, P. (Avril 1998) *Dynamic Distributed Systems in Java : Using Agent Technology to Build Flexible Systems*, Dr. Dobb's, United-States, n. 284, p. 88-92.

[5] Swaine, M. (November 1998) *Sun Dreams of Jini*, Dr. Dobb's, United-States, n. 291, p. 113-117.

[6] Flanigan, P. Karim, J. (Novembre 1998) *NCSA Symera*, Dr. Dobb's, United-States, n. 291, p. 20-27.

[7] Houlding, D. (Novembre 1998) *A CORBA Bean Framework*, Dr. Dobb's, United-States, n. 291, p. 34-40.

[8] Morgan, B. (Avril 1998) *Building Distributed Applications with Java and CORBA : Eliminating Problems with Code Distribution, Versioning and more*, Dr. Dobb's, n. 284, p. 94-99.

[9] Murphy, N. (Octobre 1998) *Introduction to CORBA for Embedded Systems*, Embedded System Programming, United-States, vol. 11, n. 11, p. 60-73.

[10] Resendes, R. Laukien, M. (Avril 1998) *Introduction to CORBA Distributed Objects*, C/C++ User Journal, United-States, vol. 16, n. 4, p. 55-65.

[11] Heidel, G. (Avril 1999) *Using Asynchronous Calls in COM*, C/C++ User Journal, United-States, vol. 17, n. 4, p. 47-54.

[12] Gontmakher, S., Horn I., (Janvier 1999) *Efficient Memory Allocation*, Dr. Dobb's Journal, United-States, n. 295, p. 116-119.

[13] Mowbray, T., William, A. (1997) *Inside CORBA : Distributed Object Standards and Applications*, United-States, Addisson-Wesley Longman Inc., 376 p.

[14] Object Management Group (OMG). (1997) *CORBA Buyers Guide*, United-States, OMG, 132 p.

[15] Object Management Group (OMG). (Février 1998) *The Common Object Request Broker: Architecture and Specification*, United-States, Revision 2.2, OMG, 962 p.

[16] Object Management Group (OMG). (Janvier 1997) *A discussion of the Object Management Architecture*, United-States, OMG, 44 p.

[17] Object Management Group (OMG). (Juin 1998) *CORBAtelecoms : Telecommunications Domain Specifications*, United-States, Version 1.0, OMG, 92 p.

[18] Object Management Group (OMG). (Novembre 1995) *Commom Facilities Architecture*, United-States, Revision 4.0, OMG, 135 p.

[19] Object Management Group (OMG). (Novembre 1997) *CORBAservices : Common Object Services Specification*, United-States, OMG, 1075 p.

[20] Puder, A. (November 1998) *The MICO CORBA Compliant System*, Dr. Dobb's, United-States, n. 291, p. 44-51.

[21] R'omer, K. Puder, A. (Mars 1999) *MICO is CORBA : CORBA 2.2 Implementation*, Version 2.2.6, 104 p.

[22] Inprise (1998) VisiBroker for C++ : Programmer's Guide, United-States, Inprise Corporation, Version 3.3, 336 p.

[23] Rogerson, D. (1997) *Inside COM*, United-States, Microsoft Press, 432 p.

[24] Eddon, G. (1998) *Inside Distributed COM*, United-States, Microsoft Press, 552 p.

[25] Microsoft, DEC, (1995), The Component Object Model Specification, United-States, Microsoft, 266 p.

[26] Booch, G. Rumbaugh, J. Jacobson, I. (Octobre 1998) *The Unified Modeling Language User Guide*, United-States, Addison-Wesley, 382 p.

[27] Rumbaugh, J. Jacobson, I. Booch, G. (Décembre 1998) *The Unified Modeling Language Reference manual*, United-States, Addison-Wesley, 550 p.

[28] Jacobson, I. Rumbaugh, J. Booch, G. (Janvier 1999) *The Unified Development Process*, United-States, Addison-Wesley, 463 p.

[29] Gamma, E., Vlissides, J., Johnson, R., Helm, R. (1995) *Design Patterns : Elements of Reusable Object-Oriented Software*, United-States, Addison-Wesley, 416 p.

[30] Lippman, B. (1995) *C++ Primer, 2nd Edition*, United-States, Addison-Wesley, 614 p.

[31] Meyers, S. (1997) *Effective C++, 2nd Edition*, United-States, Addison-Wesley, 256 p.

[32] Meyers, S. (1996) *C++, More Effective*, United-States, Addison-Wesley, 336 p.

[33] Schulzrinne, Casner, Frederick, Jacobson, (Juin 1999) *RTP : A Transport Protocol for Real-Time Applications*, IETF, Internet-Draft : draft-ietf-avt-rtp-new-04.txt, 74 p.

[34] Mauricio, A., Dugan, A., Elliott, I., Huitema, C., Pickett, S., (Février 1999) *MGCP : Media Gateway Control Protocol*, IETF, Internet-Draft : draft-huitema-megaco-mgcp-v0r1-05.txt, 120 p.

[35] Yergeau, F. (Janvier 1998) *UTF-8, a transformation format of ISO 10646*, IETF, RFC-2279, 10 p.

[36] Stevens, R. (1994) *TCP/IP Illustrated Volume 1: The Protocols*, United-States, Addison-Wesley, 600 p.

[37] Wright, G. (1995) *TCP/IP Illustrated Volume 2: The Implementation*, United-States, Addison-Wesley, 1200 p.

[38] Quinn, B., Shute, D. (1996) *Windows Sockets Network Programming*, United-States, Addison-Wesley, 656 p.

[39] Beveridge, J. (1997), *Win32, Multithreading Applications In*, United-States, Addison-Wesley, 368 p.

[40] Wakerly, J. F. (1990) *Digital Design Principles and Practices*, United-States, Prentice Hall, 716 p.

[41] Microsoft, (2000) *Microsoft Development Network Library (MSDN)*, United-States, Microsoft.

[42] Schnoebelen, P. (1999) *La vérification de logiciels Techniques et outils du model-checking*, France, Jean-Philippe Moreux, 197 p.

[43] Clarke, E., Grumberg, O., Peled, D. (2001) *Model Checking*, United-States, MIT Press, 314 p.

[44] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W., Weise, C. (July 1998) *New Generation of UPPAAL*, In proceedings of the International Workshop on Software Tools for Technology Transfer, Aalborg, Denmark.

[45] Larsen, K., Pettersson, P., Yi, W. (1997) *UPPAAL in a Nutshell*, In Springer International Journal of Software Tools for Technology Transfer 1(1+2).

[46] Lönn, H., Pettersson, P. (December 1997) *Formal Verification of a TDMA Protocol Start-Up Mechanism*, In Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems, Taipei, Taiwan, p. 235-242.

[47] Kristoffersen, K., Larroussinie, F., Larsen, K., Pettersson, P., Yi, W. (April 1997) *A Compositional Proof of a Real-Time Mutual Exclusion Protocol*, In Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development, Lille, France,.

[48] D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., Tretmans, J. (April 1997) *The bounded retransmission protocol must be on time!*, In Proceedings of the 3*rd* International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Enschede, The Netherlands, LNCS 1217, p. 416-431.

[49] D'Argenio, P.R., Katoen, J.-P., Ruys, T.C., Tretmans, J. (June, 1996) *Modeling and Verifying a Bounded Retransmission Protocol*, In Proceedings of COST 247, International Workshop on Applied Formal Methods in System Design. Maribor, Slovenia.

[50] Bengtsson J., Larsson, F. (January 1996) *UPPAAL a Tool for Automatic Verification of Real-Time Systems*, DoCS Technical Report Nr 96/67, Uppsala University.

Annexe 1 : classes de distribution d'objets

```
//==SDOC========================================================================
//==============================================================================
//
//  File: DoomHeaders.h
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __DOOMHEADERS_H__
#define __DOOMHEADERS_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"
#include "Adt/Guid.h"

NAMESPACE_START(Does)

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

#if defined(_Type)
#error "_Type" is already defined as a MACRO. Cannot be used again.
#endif

#if defined(OS_ECOS)
    #error Please define OS_ECOS GNU compiler packer
#elif defined(OS_WIN32)
    #pragma pack(push)
    #pragma pack(1)
#else
    #error Please define OS type!
#endif

template <class _Type>
struct hdrBase
{
    hdrBase() {};
    ~hdrBase() {};

    void Swap() {};
    void SwapGuid(IN SGuid& rGuid)
    {
        rGuid.dw1 = SwapBytes(rGuid.dw1);
        rGuid.w1  = SwapBytes(rGuid.w1);
        rGuid.w2  = SwapBytes(rGuid.w2);
    }
    void SwapNodeID(IN NodeID& rNodeID)
    {
        rNodeID = SwapBytes(rNodeID.S_un.S_addr);
    }

    hdrBase(IN const hdrBase& rhs)
```

```
    {
        *this = rhs;
    }

    hdrBase& operator=(IN const hdrBase& rhs)
    {
        if (this != &rhs)
        {
            memcpy(this, &rhs, sizeof(_Type));
        }
        return *this;
    }

    bool operator!=(IN const hdrBase& rhs) const
    {
        return !(*this == rhs);
    }

    bool operator==(IN const hdrBase& rhs) const
    {
        if (&rhs == this)
        {
            return true;
        }
        else if (memcmp(this, &rhs, sizeof(_Type)) == 0)
        {
            return true;
        }
        return false;
    }
};

#define HDR(x) struct x : public hdrBase<x>

HDR(hdrOperation)
{
    union
    {
        UINT8 octetstream[1];
        UINT8 byteorder;
    };
    UINT8  padding1;
    UINT16 padding2;
    UINT32 oid;

    hdrOperation() :
        byteorder(SYSTEM_BYTE_ORDERING),
        padding1(PAD),
        padding2(PAD)
    {};

    void Swap()
    {
        if (byteorder != SYSTEM_BYTE_ORDERING)
        {
            oid = SwapBytes(oid);
        }
    };
};

HDR(hdrProtocolId)
{
    UINT32 id;
    UINT8  vmajor;
    UINT8  vminor;

    hdrProtocolId(UINT32 doomid = DOOM_ID,
                  UINT8 doomvmajor = DOOM_VMAJOR,
                  UINT8 doomvminor = DOOM_VMINOR) :
        id(doomid),
        vmajor(doomvmajor),
```

```
        vminor(doomvminor)
    {};

    void Swap() {}; // No need to swap the magic dword because
                    // it is hardcoded in the right order
};

const hdrProtocolId DOOM_VERSION_1_0;

HDR(hdrRoot)
{
    hdrProtocolId id;
#if SYSTEM_BYTE_ORDERING == BIG_ENDIAN_UINT8_ORDERING
    UINT8 byteorder:1;
    UINT8 lock:1;
    UINT8 padding1:6;
    UINT8 padding2;
#elif SYSTEM_BYTE_ORDERING == LITTLE_ENDIAN_UINT8_ORDERING
    UINT8 padding1:6;
    UINT8 lock:1;
    UINT8 byteorder:1;
    UINT8 padding2;
#else
    #error Please define system byte ordering!
#endif
    LinkNumber lkn;
    MsgSeqNum msn;
    ActionID aid;

    hdrRoot(hdrProtocolId& rId = hdrProtocolId(), UINT8 lock = LOCK_ON) :
        id(rId),
        byteorder(SYSTEM_BYTE_ORDERING),
        lock(lock),
        padding1(PAD),
        padding2(PAD)
    {};

    hdrRoot(LinkNumber linknum,
            ActionID actionid,
            MsgSeqNum msgseqnum = 0,
            hdrProtocolId& rId = hdrProtocolId(),
            UINT8 lock = LOCK_ON) :
        id(rId),
        byteorder(SYSTEM_BYTE_ORDERING),
        lock(lock),
        padding1(PAD),
        padding2(PAD),
        lkn(linknum),
        msn(msgseqnum),
        aid(actionid)
    {};

    void Swap()
    {
        if (byteorder != SYSTEM_BYTE_ORDERING)
        {
            id.Swap();
            lkn = SwapBytes(lkn);
            msn = SwapBytes(msn);
            aid = SwapBytes(aid);
        }
    };
};

HDR(hdrLink)
{
    hdrRoot root;
    ClassID clsid;
    ObjectID objid;

    void Swap()
```

```
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            SwapGuid(clsid);
            SwapGuid(objid);
        }
    };
};

HDR(hdrLinked)
{
    hdrRoot root;
    LinkNumber lkn;

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            lkn = SwapBytes(lkn);
        }
    };
};

HDR(hdrUnlink)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrUnlinked)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrCreate)
{
    hdrRoot root;
    ClassID clsid;
    hdrOperation op;

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            SwapGuid(clsid);
        }
    };
};

HDR(hdrCreated)
{
    hdrRoot root;
    ObjectID objid;
    LinkNumber lkn;
    UINT32 padding;
    hdrOperation op;

    hdrCreated() :
        padding(PAD)
    {};

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            SwapGuid(objid);
            lkn = SwapBytes(lkn);
        }
```

```
    };
};

HDR(hdrDestroy)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrDestroyed)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrCall)
{
    hdrRoot root;
    hdrOperation op;

    void Swap() {};
};

HDR(hdrReturn)
{
    hdrRoot root;
    hdrOperation op;

    void Swap() {};
};

HDR(hdrSend)
{
    hdrRoot root;
    hdrOperation op;

    void Swap() {};
};

HDR(hdrReceived)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrLock)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrLocked)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrUnlock)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrUnlocked)
{
```

```
    hdrRoot root;

    void Swap() {};
};

HDR(hdrLocate)
{
    hdrRoot root;
    ClassID clsid;
    ObjectID objid;

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            SwapGuid(clsid);
            SwapGuid(objid);
        }
    };
};

HDR(hdrInteroperableObjectReference)
{
    hdrProtocolId id;
#if SYSTEM_BYTE_ORDERING == BIG_ENDIAN_UINT8_ORDERING
    UINT8 byteorder:1;
    UINT8 padding:7;
#elif SYSTEM_BYTE_ORDERING == LITTLE_ENDIAN_UINT8_ORDERING
    UINT8 padding:7;
    UINT8 byteorder:1;
#else
    #error Please define system byte ordering!
#endif
    UINT8 addressfamily;
    ClassID clsid;
    ObjectID objid;

    hdrInteroperableObjectReference(hdrProtocolId &rId = hdrProtocolId()) :
        id(rId),
        byteorder(SYSTEM_BYTE_ORDERING),
        padding(PAD)
    {};

    void Swap()
    {
        if (byteorder != SYSTEM_BYTE_ORDERING)
        {
            id.Swap();
            SwapGuid(clsid);
            SwapGuid(objid);
        }
    };
};

HDR(hdrInternetObjectAddress)
{
    UINT8 addresstype;
    UINT8 padding;
    ProcessID procid;
    NodeID nodeid;

    hdrInternetObjectAddress() :
        padding(PAD)
    {};

    void Swap()
    {
        procid = SwapBytes(procid);
        SwapNodeID(nodeid);
    };
};
```

```
HDR(hdrLocated)
{
    hdrRoot root;
    UINT32 locationstatus;
    hdrInteroperableObjectReference ior;
    hdrInternetObjectAddress ioa;

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            locationstatus = SwapBytes(locationstatus);
            ior.Swap();
            ioa.Swap();
        }
    };
};

HDR(hdrAck)
{
    hdrRoot root;

    void Swap() {};
};

HDR(hdrNak)
{
    hdrRoot root;
    UINT8   errorlevel;
    UINT16  padding1;
    UINT8   padding2;
    UINT32  errorcode;

    hdrNak() :
        padding1(PAD),
        padding2(PAD)
    {};

    void Swap()
    {
        if (root.byteorder != SYSTEM_BYTE_ORDERING)
        {
            errorcode = SwapBytes(errorcode);
        }
    };
};

#if defined(OS_ECOS)
    #error Please define OS_ECOS GNU compiler packer
#elif defined(OS_WIN32)
    #pragma pack(pop)
#else
    #error Please define OS type!
#endif

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __DOOMHEADERS_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: Guid.h
//
//  Package: Adt
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __GUID_H__
#define __GUID_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

#include "Threading/CMarshaler.h"

NAMESPACE_START(Mx)

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

struct SGuid
{
    // << Data Members >>
    //--------------------
    UINT32 dw1;
    UINT16 w1;
    UINT16 w2;
    UINT8  b[8];

    // << Constructors/Destructors >>
    //--------------------------------
    SGuid();
    SGuid(IN UINT32 dword1, IN UINT16 word1, IN UINT16 word2,
          IN UINT8 byte0, IN UINT8 byte1, IN UINT8 byte2, IN UINT8 byte3,
          IN UINT8 byte4, IN UINT8 byte5, IN UINT8 byte6, IN UINT8 byte7);
    SGuid(IN const SGuid& rhs);
    ~SGuid() {};

    // << Operators >>
    //----------------------------------------
    SGuid& operator=(IN const SGuid& rhs);
    bool operator==(IN const SGuid& rhs) const;
    bool operator!=(IN const SGuid& rhs) const;
    bool operator<(IN const SGuid& rhs) const;
    bool operator>(IN const SGuid& rhs) const;
    bool operator<=(IN const SGuid& rhs) const;
    bool operator>=(IN const SGuid& rhs) const;
};

//==============================================================================
//====  MACROS  ================================================================
//==============================================================================
#ifdef DEFINE_GUID
#undef DEFINE_GUID
#endif
#define DEFINE_GUID(name, dw1, w1, w2, b0, b1, b2, b3, b4, b5, b6, b7) \
            SGuid name(dw1, w1, w2, b0, b1, b2, b3, b4, b5, b6, b7)

#ifdef DEFINE_CONST_GUID
#undef DEFINE_CONST_GUID
#endif
```

```
#define DEFINE_CONST_GUID(name, dw1, w1, w2, b0, b1, b2, b3, b4, b5, b6, b7) \
            const SGuid name(dw1, w1, w2, b0, b1, b2, b3, b4, b5, b6, b7)

//=============================================================================
//====  CONSTANTS + DEFINES  ==================================================
//=============================================================================

const SGuid GUID_NULL;

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

inline
SGuid::SGuid()
{
    memset(this, 0, sizeof(SGuid));
}

inline
SGuid::SGuid(IN UINT32 dword1, IN UINT16 word1, IN UINT16 word2,
             IN UINT8 byte0, IN UINT8 byte1, IN UINT8 byte2, IN UINT8 byte3,
             IN UINT8 byte4, IN UINT8 byte5, IN UINT8 byte6, IN UINT8 byte7)
{
    dw1  = dword1;
    w1   = word1;
    w2   = word2;
    b[0] = byte0;
    b[1] = byte1;
    b[2] = byte2;
    b[3] = byte3;
    b[4] = byte4;
    b[5] = byte5;
    b[6] = byte6;
    b[7] = byte7;
}

inline
SGuid::SGuid(IN const SGuid& rhs)
{
    *this = rhs;
}

inline
SGuid& SGuid::operator=(IN const SGuid& rhs)
{
    if (this != &rhs)
    {
        memcpy(this, &rhs, sizeof(SGuid));
    }

    return *this;
}

inline
bool SGuid::operator==(IN const SGuid& rhs) const
{
    return ((this == &rhs) ||
            (memcmp(this, &rhs, sizeof(SGuid)) == 0));
}

inline
bool SGuid::operator!=(IN const SGuid& rhs) const
{
    return !(*this == rhs);
}

inline
bool SGuid::operator<(IN const SGuid& rhs) const
{
    return ((this != &rhs) &&
```

```
            (memcmp(this, &rhs, sizeof(SGuid)) < 0));
}

inline
bool SGuid::operator>(IN const SGuid& rhs) const
{
    return (rhs < *this);
}

inline
bool SGuid::operator<=(IN const SGuid& rhs) const
{
    return !(rhs < *this);
}

inline
bool SGuid::operator>=(IN const SGuid& rhs) const
{
    return !(*this < rhs);
}

//=============================================================================
//====  MARSHALING FUNCTIONS  =================================================
//=============================================================================

inline
IMarshaler& operator<< (IN IMarshaler& rMarshaler, IN const SGuid& rData)
{
    rMarshaler.Store(&rData, sizeof(SGuid));
    return rMarshaler;
}

inline
IMarshaler& operator>> (IN IMarshaler& rMarshaler, IN SGuid& rData)
{
    UINT16 wDataLen;
    rMarshaler.Load(&rData, sizeof(SGuid), wDataLen);
    return rMarshaler;
}

NAMESPACE_END(Mx)

#endif // #ifndef __GUID_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMessage.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CMESSAGE_H__
#define __CMESSAGE_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"
#include "Does/DoomHeaders.h"
#include "Adt/CBlob.h"

// Data Member
#include "NetLayer/CSocketAddr.h"

// Interface Realized & Parent
#include "Memory/CObjectAllocator.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CMessage
//
//==EDOC========================================================================
class CMessage : public CObjectAllocator<CMessage> // Must be thread protected
{
// Published Interface
//--------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CMessage();
    CMessage(IN CBlob* pData);
    CMessage(IN const CMessage& rhs);
    virtual ~CMessage() {};

    // << accessors >>
    //----------------
    bool IsValidHdr() const;
    bool IsEmpty() const;
    bool IsAction() const;
    bool IsReaction() const;

    bool SwapHdrBytes();
```

```
    UINT32 GetSize() const;
    UINT32 GetCapacity() const;
    bool Resize(UINT32 nSize);

    const CSocketAddr& GetSourceAddr() const;
    const CSocketAddr& GetDestinationAddr() const;
    void SetSourceAddr(IN const CSocketAddr& rSource);
    void SetDestinationAddr(IN const CSocketAddr& rDest);

    UINT8* GetRawMsg();

    TransmitterID GetTxID() const;

    bool CopyToBlob(IN CBlob* pData) const;

    // << Operators >>
    //---------------------------------
    CMessage& operator=(IN const CMessage& rhs);
    bool operator==(IN const CMessage& rhs) const;
    bool operator!=(IN const CMessage& rhs) const;

    operator hdrProtocolId*();
    operator hdrRoot*();
    operator hdrLink*();
    operator hdrLinked*();
    operator hdrUnlink*();
    operator hdrUnlinked*();
    operator hdrCreate*();
    operator hdrCreated*();
    operator hdrDestroy*();
    operator hdrDestroyed*();
    operator hdrCall*();
    operator hdrReturn*();
    operator hdrSend*();
    operator hdrReceived*();
    operator hdrLock*();
    operator hdrLocked*();
    operator hdrUnlock*();
    operator hdrUnlocked*();
    operator hdrLocate*();
    operator hdrLocated*();
    operator hdrAck*();
    operator hdrNak*();

// Public Data Members
//---------------------
private:
    CSocketAddr m_addrSource;
    CSocketAddr m_addrDestination;

    static const UINT32 m_nCapacity;
    UINT32 m_nSize;

    union
    {
        hdrProtocolId* pProtocolId;
        hdrRoot* pRoot;
        hdrLink* pLink;
        hdrLinked* pLinked;
        hdrUnlink* pUnlink;
        hdrUnlinked* pUnlinked;
        hdrCreate* pCreate;
        hdrCreated* pCreated;
        hdrDestroy* pDestroy;
        hdrDestroyed* pDestroyed;
        hdrCall* pCall;
        hdrReturn* pReturn;
        hdrSend* pSend;
        hdrReceived* pReceived;
        hdrLock* pLock;
        hdrLocked* pLocked;
```

```
        hdrUnlock* pUnlock;
        hdrUnlocked* pUnlocked;
        hdrLocate* pLocate;
        hdrLocated* pLocated;
        hdrAck* pAck;
        hdrNak* pNak;
        UINT8 raw[MAX_MESSAGE_SIZE];
    } m_msg;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

inline
CMessage::CMessage()
: m_nSize(0)
{
}

inline
CMessage::CMessage(IN CBlob* pData)
{
    ASSERT(pData != NULL);
    m_nSize = pData->Size();
    ASSERT(m_nSize >= m_nCapacity);
    memcpy(m_msg.raw, pData->Begin(), m_nSize);
}

inline
CMessage::CMessage(IN const CMessage& rhs)
{
    *this = rhs;
}

inline
bool CMessage::IsValidHdr() const
{
    return (m_nSize > sizeof(hdrRoot) &&
            *(m_msg.pProtocolId) == DOOM_VERSION_1_0);
}

inline
bool CMessage::IsEmpty() const
{
    return (m_nSize == 0);
}

inline
bool CMessage::IsAction() const
{
    return !(m_msg.pRoot->aid & ACTIONID_BITMASK);
}

inline
bool CMessage::IsReaction() const
{
    return (m_msg.pRoot->aid & ACTIONID_BITMASK == 0);
}

inline
bool CMessage::SwapHdrBytes()
{
    bool bSwapped = true;

    m_msg.pRoot->Swap();
    switch (m_msg.pRoot->aid)
    {
    case aLINK:
        m_msg.pLink->Swap();
        break;
```

```
    case rLINKED:
        m_msg.pLinked->Swap();
        break;
    case aUNLINK:
        m_msg.pUnlink->Swap();
        break;
    case rUNLINKED:
        m_msg.pUnlinked->Swap();
        break;
    case aCREATE:
        m_msg.pCreate->Swap();
        break;
    case rCREATED:
        m_msg.pCreated->Swap();
        break;
    case aDESTROY:
        m_msg.pDestroy->Swap();
        break;
    case rDESTROYED:
        m_msg.pDestroyed->Swap();
        break;
    case aCALL:
        m_msg.pCall->Swap();
        break;
    case rRETURN:
        m_msg.pReturn->Swap();
        break;
    case aSEND:
        m_msg.pSend->Swap();
        break;
    case rRECEIVED:
        m_msg.pReceived->Swap();
        break;
    case aLOCK:
        m_msg.pLock->Swap();
        break;
    case rLOCKED:
        m_msg.pLocked->Swap();
        break;
    case aUNLOCK:
        m_msg.pUnlock->Swap();
        break;
    case rUNLOCKED:
        m_msg.pUnlocked->Swap();
        break;
    case aLOCATE:
        m_msg.pLocate->Swap();
        break;
    case rLOCATED:
        m_msg.pLocated->Swap();
        break;
    case aACK:
    case rACK:
        m_msg.pAck->Swap();
        break;
    case aNAK:
    case rNAK:
        m_msg.pNak->Swap();
        break;
    default:
        bSwapped = false;
    }
    return bSwapped;
}

inline
UINT32 CMessage::GetSize() const
{
    return m_nSize;
}
```

```
inline
UINT32 CMessage::GetCapacity() const
{
    return m_nCapacity;
}

inline
bool CMessage::Resize(UINT32 nSize)
{
    bool bSuccess = true;

    if (nSize <= m_nCapacity)
    {
        m_nSize = nSize;
    }
    else
    {
        bSuccess = false;
    }

    return bSuccess;
}

inline
const CSocketAddr& CMessage::GetSourceAddr() const
{
    return m_addrSource;
}

inline
const CSocketAddr& CMessage::GetDestinationAddr() const
{
    return m_addrDestination;
}

inline
void CMessage::SetSourceAddr(IN const CSocketAddr& rSource)
{
    m_addrSource = rSource;
}

inline
void CMessage::SetDestinationAddr(IN const CSocketAddr& rDest)
{
    m_addrDestination = rDest;
}

inline
TransmitterID CMessage::GetTxID() const
{
    TransmitterID txid;
    m_addrSource.GetIPAddr(txid.nid);
    txid.pid = m_addrSource.GetIPPort();
    return txid;
}

inline
UINT8* CMessage::GetRawMsg()
{
    return m_msg.raw;
}

inline
bool CMessage::CopyToBlob(IN CBlob* pData) const
{
    bool bCopied = false;

    if (GetSize() <= pData->Capacity())
    {
        memcpy(pData->Begin(), m_msg.raw, GetSize());
        pData->Resize(GetSize());
```

```
        bCopied = true;
    }
    else
    {
        ASSERT(0);
    }

    return bCopied;
}

inline
CMessage& CMessage::operator=(IN const CMessage& rhs)
{
    if (this != &rhs)
    {
        m_addrSource = rhs.m_addrSource;
        m_addrDestination = rhs.m_addrDestination;
        m_nSize = rhs.m_nSize;
        memcpy(m_msg.raw, rhs.m_msg.raw, rhs.m_nSize);
    }

    return *this;
}

inline
bool CMessage::operator==(IN const CMessage& rhs) const
{
    bool bEqual = false;

    if (&rhs == this)
    {
        bEqual = true;
    }
    else if (m_addrSource == rhs.m_addrSource &&
             m_addrDestination == rhs.m_addrDestination &&
             m_nSize == rhs.m_nSize &&
             memcmp(m_msg.raw, rhs.m_msg.raw, rhs.m_nSize) == 0)
    {
        bEqual = true;
    }

    return bEqual;
}

inline
bool CMessage::operator!=(IN const CMessage& rhs) const
{
    return !(*this == rhs);
}

inline
CMessage::operator hdrProtocolId*()
{
    return m_msg.pProtocolId;
}

inline
CMessage::operator hdrRoot*()
{
    return m_msg.pRoot;
}

inline
CMessage::operator hdrLink*()
{
    return m_msg.pLink;
}

inline
CMessage::operator hdrLinked*()
{
```

```
    return m_msg.pLinked;
}

inline
CMessage::operator hdrUnlink*()
{
    return m_msg.pUnlink;
}

inline
CMessage::operator hdrUnlinked*()
{
    return m_msg.pUnlinked;
}

inline
CMessage::operator hdrCreate*()
{
    return m_msg.pCreate;
}

inline
CMessage::operator hdrCreated*()
{
    return m_msg.pCreated;
}

inline
CMessage::operator hdrDestroy*()
{
    return m_msg.pDestroy;
}

inline
CMessage::operator hdrDestroyed*()
{
    return m_msg.pDestroyed;
}

inline
CMessage::operator hdrCall*()
{
    return m_msg.pCall;
}

inline
CMessage::operator hdrReturn*()
{
    return m_msg.pReturn;
}

inline
CMessage::operator hdrSend*()
{
    return m_msg.pSend;
}

inline
CMessage::operator hdrReceived*()
{
    return m_msg.pReceived;
}

inline
CMessage::operator hdrLock*()
{
    return m_msg.pLock;
}

inline
CMessage::operator hdrLocked*()
```

```
{
    return m_msg.pLocked;
}

inline
CMessage::operator hdrUnlock*()
{
    return m_msg.pUnlock;
}

inline
CMessage::operator hdrUnlocked*()
{
    return m_msg.pUnlocked;
}

inline
CMessage::operator hdrLocate*()
{
    return m_msg.pLocate;
}

inline
CMessage::operator hdrLocated*()
{
    return m_msg.pLocated;
}

inline
CMessage::operator hdrAck*()
{
    return m_msg.pAck;
}

inline
CMessage::operator hdrNak*()
{
    return m_msg.pNak;
}

NAMESPACE_END(Does)

#endif // #ifndef __CMESSAGE_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMarshaler.h
//
//  Package: Threading
//
//  OS: All
//  HW: All
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __MARSHALER_H__
#define __MARSHALER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member

// Interface Realized & Parent
#include "Std/IInterface.h"
#include "Memory/CObjectAllocator.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Mx)

// Forward Declarations Inside of the Namespace
class CRootMarshaler;
class CInprocMarshaler;
class COutprocMarshaler;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

// Buffer constants
const DWORD MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SIZE = 64;
const DWORD MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SPACE =
MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SIZE - sizeof(ptrdiff_t);
const DWORD MARSHALER_OUTPROC_MEMORY_SEGMENT_SIZE = 2048;

// Byte ordering constants
#if SYSTEM_BYTE_ORDERING == BIG_ENDIAN_BYTE_ORDERING
    const DWORD MARSHALER_BYTE_ORDERING = 0x01000000;
#elif SYSTEM_BYTE_ORDERING == LITTLE_ENDIAN_BYTE_ORDERING
    const DWORD MARSHALER_BYTE_ORDERING = 0x00000000;
#else
    #error Please define system byte ordering!
#endif

// Data Type Constants used for debug purpose
const BYTE MARSHALING_TYPE_BOOL        = 1;
const BYTE MARSHALING_TYPE_CHAR        = 2;
const BYTE MARSHALING_TYPE_UCHAR       = 3;
const BYTE MARSHALING_TYPE_SCHAR       = 4;
const BYTE MARSHALING_TYPE_USHORT      = 5;
const BYTE MARSHALING_TYPE_SSHORT      = 6;
const BYTE MARSHALING_TYPE_UINT        = 7;
const BYTE MARSHALING_TYPE_SINT        = 8;
const BYTE MARSHALING_TYPE_ULONG       = 9;
const BYTE MARSHALING_TYPE_SLONG       = 10;
const BYTE MARSHALING_TYPE_FLOAT       = 11;
const BYTE MARSHALING_TYPE_DOUBLE      = 12;
const BYTE MARSHALING_TYPE_LONGDOUBLE  = 13;
const BYTE MARSHALING_TYPE_POINTER     = 14;
```

```
const BYTE MARSHALING_TYPE_UNKNOWN     = 15;

//=============================================================================
//====  NEW TYPE DEFINITIONS  =================================================
//=============================================================================
#if defined(_Type)
#error "_Type" is already defined as a MACRO. Cannot be used again in template definition.
#endif

// Default marshaler, used for backward compatibility
typedef CInprocMarshaler CMarshaler;

//==SDOC=======================================================================
//
//  Interface IMarshaler
//
//==EDOC=======================================================================
interface(IMarshaler)
{
    virtual bool IsEmpty() const = 0;
    virtual void Clear() = 0;

    // Storing Methods
    //----------------------------------------
    virtual IMarshaler& operator<< (IN bool data) = 0;
    virtual IMarshaler& operator<< (IN char data) = 0;
    virtual IMarshaler& operator<< (IN unsigned char data) = 0;
    virtual IMarshaler& operator<< (IN signed char data) = 0;
    virtual IMarshaler& operator<< (IN unsigned short data) = 0;
    virtual IMarshaler& operator<< (IN signed short data) = 0;
    virtual IMarshaler& operator<< (IN unsigned int data) = 0;
    virtual IMarshaler& operator<< (IN signed int data) = 0;
    virtual IMarshaler& operator<< (IN unsigned long data) = 0;
    virtual IMarshaler& operator<< (IN signed long data) = 0;
    virtual IMarshaler& operator<< (IN const float& data) = 0;
    virtual IMarshaler& operator<< (IN const double& data) = 0;
    virtual IMarshaler& operator<< (IN const long double& data) = 0;

    // Variable length data
    virtual bool Store(IN const void* pData, IN UINT16 wSize) = 0;

    // Loading Methods
    //----------------------------------------
    virtual IMarshaler& operator>> (OUT bool& data) = 0;
    virtual IMarshaler& operator>> (OUT char& data) = 0;
    virtual IMarshaler& operator>> (OUT unsigned char& data) = 0;
    virtual IMarshaler& operator>> (OUT signed char& data) = 0;
    virtual IMarshaler& operator>> (OUT unsigned short& data) = 0;
    virtual IMarshaler& operator>> (OUT signed short& data) = 0;
    virtual IMarshaler& operator>> (OUT unsigned int& data) = 0;
    virtual IMarshaler& operator>> (OUT signed int& data) = 0;
    virtual IMarshaler& operator>> (OUT unsigned long& data) = 0;
    virtual IMarshaler& operator>> (OUT signed long& data) = 0;
    virtual IMarshaler& operator>> (OUT float& data) = 0;
    virtual IMarshaler& operator>> (OUT double& data) = 0;
    virtual IMarshaler& operator>> (OUT long double& data) = 0;

    // Variable Length Data
    virtual bool Load(OUT void* pData, IN UINT16 wCapacity, OUT UINT16& rwSize) = 0;
};

//==SDOC=======================================================================
//
//  class CRootMarshaler
//
//==EDOC=======================================================================
class CRootMarshaler : public IMarshaler
{
// Published Interface
//--------------------
public:
```

```
    // << Constructors / Destructors >>
    //--------------------------------------
    CRootMarshaler() {};
    virtual ~CRootMarshaler() { ASSERT(IsEmpty()); };

    // IMarshaler Interface
    //--------------------------------------
    virtual bool IsEmpty() const;
    virtual void Clear();

    // Storing Methods
    virtual IMarshaler& operator<< (IN bool data);
    virtual IMarshaler& operator<< (IN char data);
    virtual IMarshaler& operator<< (IN unsigned char data);
    virtual IMarshaler& operator<< (IN signed char data);
    virtual IMarshaler& operator<< (IN unsigned short data);
    virtual IMarshaler& operator<< (IN signed short data);
    virtual IMarshaler& operator<< (IN unsigned int data);
    virtual IMarshaler& operator<< (IN signed int data);
    virtual IMarshaler& operator<< (IN unsigned long data);
    virtual IMarshaler& operator<< (IN signed long data);
    virtual IMarshaler& operator<< (IN const float& data);
    virtual IMarshaler& operator<< (IN const double& data);
    virtual IMarshaler& operator<< (IN const long double& data);
    virtual bool Store(IN const void* pData, IN UINT16 wSize);

    // Loading Methods
    virtual IMarshaler& operator>> (OUT bool& data);
    virtual IMarshaler& operator>> (OUT char& data);
    virtual IMarshaler& operator>> (OUT unsigned char& data);
    virtual IMarshaler& operator>> (OUT signed char& data);
    virtual IMarshaler& operator>> (OUT unsigned short& data);
    virtual IMarshaler& operator>> (OUT signed short& data);
    virtual IMarshaler& operator>> (OUT unsigned int& data);
    virtual IMarshaler& operator>> (OUT signed int& data);
    virtual IMarshaler& operator>> (OUT unsigned long& data);
    virtual IMarshaler& operator>> (OUT signed long& data);
    virtual IMarshaler& operator>> (OUT float& data);
    virtual IMarshaler& operator>> (OUT double& data);
    virtual IMarshaler& operator>> (OUT long double& data);
    virtual bool Load(OUT void* pData, IN UINT16 wCapacity, OUT UINT16& rwSize);

// Hidden Methods
//---------------
protected:
    // Insertion and extraction
    //-------------------------
    virtual void Insert(IN const void* pData, IN UINT16 wSize) = 0;
    virtual void Extract(OUT void* pData, IN UINT16 wSize) = 0;

    virtual UINT16 AdaptByteOrder(UINT16 w) const = 0;
    virtual UINT32 AdaptByteOrder(UINT32 dw) const = 0;

    // Insertion and extraction of type info
    // NOTE: For debug purpose only
    //--------------------------------------
    virtual void InsertTypeInfo(IN BYTE btTypeInfo);
    virtual void ExtractTypeInfo(IN BYTE btTypeInfo);

private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CRootMarshaler(const CRootMarshaler& rhs);
    CRootMarshaler& operator=(const CRootMarshaler& rhs);

// Hidden Data Members
//--------------------
protected:
    // Extraction and Insertion Pointers
    //----------------------------------
    BYTE* m_pInsertPosition;
```

```
    BYTE* m_pInsertSegmentEnd;
    BYTE* m_pExtractPosition;
    BYTE* m_pExtractSegmentEnd;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

//============================================================================
//====  IMARSHALER METHODS  ==================================================
//============================================================================

inline
bool CRootMarshaler::IsEmpty() const
{
    return (m_pInsertPosition == m_pExtractPosition);
}

inline
void CRootMarshaler::Clear()
{
    m_pExtractPosition = m_pInsertPosition;
    m_pExtractSegmentEnd = m_pInsertSegmentEnd;
}

//============================================================================
//====  STORING METHODS  =====================================================
//============================================================================

inline
void CRootMarshaler::InsertTypeInfo(IN BYTE btTypeInfo)
{
#ifdef MX_DEBUG
    Insert(&btTypeInfo, sizeof(btTypeInfo));
#endif
}

inline
IMarshaler& CRootMarshaler::operator<< (IN bool data)
{
    InsertTypeInfo(MARSHALING_TYPE_BOOL);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN char data)
{
    InsertTypeInfo(MARSHALING_TYPE_CHAR);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN unsigned char data)
{
    InsertTypeInfo(MARSHALING_TYPE_UCHAR);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN signed char data)
{
    InsertTypeInfo(MARSHALING_TYPE_SCHAR);
    Insert(&data, sizeof(data));
    return *this;
}

inline
```

```
IMarshaler& CRootMarshaler::operator<< (IN unsigned short data)
{
    InsertTypeInfo(MARSHALING_TYPE_USHORT);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN signed short data)
{
    InsertTypeInfo(MARSHALING_TYPE_SSHORT);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN unsigned int data)
{
    InsertTypeInfo(MARSHALING_TYPE_UINT);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN signed int data)
{
    InsertTypeInfo(MARSHALING_TYPE_SINT);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN unsigned long data)
{
    InsertTypeInfo(MARSHALING_TYPE_ULONG);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN signed long data)
{
    InsertTypeInfo(MARSHALING_TYPE_SLONG);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN const float& data)
{
    InsertTypeInfo(MARSHALING_TYPE_FLOAT);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN const double& data)
{
    InsertTypeInfo(MARSHALING_TYPE_DOUBLE);
    Insert(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator<< (IN const long double& data)
{
    InsertTypeInfo(MARSHALING_TYPE_LONGDOUBLE);
    Insert(&data, sizeof(data));
    return *this;
}
```

```
//============================================================================
//====  LOADING METHODS  =====================================================
//============================================================================

inline
void CRootMarshaler::ExtractTypeInfo(IN BYTE btTypeInfo)
{
#ifdef MX_DEBUG
    BYTE btExtractedTypeInfo = 0;
    Extract(&btExtractedTypeInfo, sizeof(btTypeInfo));
    ASSERT(btTypeInfo == btExtractedTypeInfo);
#endif
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT bool& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_BOOL);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT char& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_CHAR);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT unsigned char& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_UCHAR);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT signed char& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_SCHAR);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT unsigned short& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_USHORT);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT16)data);
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT signed short& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_SSHORT);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT16)data);
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT unsigned int& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_UINT);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT32)data);
    return *this;
```

```
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT signed int& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_SINT);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT32)data);
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT unsigned long& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_ULONG);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT32)data);
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT signed long& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_SLONG);
    Extract(&data, sizeof(data));
    AdaptByteOrder((UINT32)data);
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT float& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_FLOAT);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT double& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_DOUBLE);
    Extract(&data, sizeof(data));
    return *this;
}

inline
IMarshaler& CRootMarshaler::operator>> (OUT long double& data)
{
    ExtractTypeInfo(MARSHALING_TYPE_LONGDOUBLE);
    Extract(&data, sizeof(data));
    return *this;
}

//==SDOC========================================================================
//
//  class CInprocMarshaler
//
//==EDOC========================================================================
class CInprocMarshaler : public CRootMarshaler, public CObjectAllocator<CInprocMarshaler>
{
// Friend Declaration
//--------------------
    template <class _Type> friend IMarshaler& operator<< (IN IMarshaler& rMarshaler, IN const
_Type* pPtr);
    template <class _Type> friend IMarshaler& operator>> (IN IMarshaler& rMarshaler, OUT _Type*&
pPtr);
    template <class _Type> friend IMarshaler& operator<< (IN CInprocMarshaler& rMarshaler, IN
const _Type* pPtr);
    template <class _Type> friend IMarshaler& operator>> (IN CInprocMarshaler& rMarshaler, OUT
_Type*& pPtr);
```

```
// Published Interface
//--------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------------
    CInprocMarshaler();
    virtual ~CInprocMarshaler();

// Hidden Methods
//---------------
protected:
    // Insertion and extraction
    //-------------------------
    virtual void Insert(IN const void* pData, IN UINT16 wSize);
    virtual void Extract(OUT void* pData, IN UINT16 wSize);

    virtual UINT16 AdaptByteOrder(UINT16 w) const { return w; };
    virtual UINT32 AdaptByteOrder(UINT32 dw) const { return dw; };

private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CInprocMarshaler(const CInprocMarshaler& rhs);
    CInprocMarshaler& operator=(const CInprocMarshaler& rhs);

// Hidden Data Members
//--------------------
private:
    BYTE m_pFirstSegment[MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SIZE];
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

template<class _Type>
inline
IMarshaler& operator<< (IN IMarshaler& rMarshaler, IN const _Type* pPtr)
{
    static_cast<CInprocMarshaler*>(&rMarshaler)->InsertTypeInfo(MARSHALING_TYPE_POINTER);
    static_cast<CInprocMarshaler*>(&rMarshaler)->Insert(&pPtr, sizeof(const _Type*));
    return rMarshaler;
}

template<class _Type>
inline
IMarshaler& operator>> (IN IMarshaler& rMarshaler, OUT _Type*& pPtr)
{
    static_cast<CInprocMarshaler*>(&rMarshaler)->ExtractTypeInfo(MARSHALING_TYPE_POINTER);
    static_cast<CInprocMarshaler*>(&rMarshaler)->Extract(&pPtr, sizeof(const _Type*));
    return rMarshaler;
}

template<class _Type>
inline
IMarshaler& operator<< (IN CInprocMarshaler& rMarshaler, IN const _Type* pPtr)
{
    rMarshaler.InsertTypeInfo(MARSHALING_TYPE_POINTER);
    rMarshaler.Insert(&pPtr, sizeof(const _Type*));
    return rMarshaler;
}

template<class _Type>
inline
IMarshaler& operator>> (IN CInprocMarshaler& rMarshaler, OUT _Type*& pPtr)
{
    rMarshaler.ExtractTypeInfo(MARSHALING_TYPE_POINTER);
    rMarshaler.Extract(&pPtr, sizeof(const _Type*));
    return rMarshaler;
}
```

```
//==SDOC========================================================================
//
//  class COutprocMarshaler
//
//==EDOC========================================================================
class COutprocMarshaler : public CRootMarshaler, public CObjectAllocator<COutprocMarshaler>
{
// Published Interface
//--------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    COutprocMarshaler(IN UINT16 wCapacity = MARSHALER_OUTPROC_MEMORY_SEGMENT_SIZE);
    COutprocMarshaler(IN BYTE* pOctetStream, IN UINT16 wSize, IN UINT16 wCapacity);
    virtual ~COutprocMarshaler();

    BYTE* ReleaseBuffer();

// Hidden Methods
//---------------
protected:
    // Insertion and extraction
    //-------------------------
    virtual void Insert(IN const void* pData, IN UINT16 wSize);
    virtual void Extract(OUT void* pData, IN UINT16 wSize);

    virtual UINT16 AdaptByteOrder(UINT16 w) const;
    virtual UINT32 AdaptByteOrder(UINT32 dw) const;

private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    COutprocMarshaler(const COutprocMarshaler& rhs);
    COutprocMarshaler& operator=(const COutprocMarshaler& rhs);

// Hidden Data Members
//--------------------
private:
    // Byte Ordering :
    //      0x00000000 = LITTLE_ENDIAN
    //      0x01000000 = BIG_ENDIAN
    //--------------------------------
    UINT32 m_btByteOrdering;
    BYTE* m_pBufferHead;
};

//==============================================================================
//====  INLINE FUNCTIONS  ======================================================
//==============================================================================

inline
UINT16 COutprocMarshaler::AdaptByteOrder(UINT16 w) const
{
    if (m_btByteOrdering != MARSHALER_BYTE_ORDERING)
    {
        return SwapBytes(w);
    }
    else
    {
        return w;
    }
}

inline
UINT32 COutprocMarshaler::AdaptByteOrder(UINT32 dw) const
{
    if (m_btByteOrdering != MARSHALER_BYTE_ORDERING)
    {
        return SwapBytes(dw);
    }
    else
```

```
    {
        return dw;
    }
}

NAMESPACE_END(Mx)

#endif // #ifndef __MARSHALER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMarshaler.h
//
//  Package: Threading
//
//  OS: All
//  HW: All
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Threading/CMarshaler.h"

extern "C"
{
}

NAMESPACE_START(Mx)

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
const DWORD MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SIZE = sizeof(CMarshaler);
const DWORD MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SPACE = MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SIZE
- sizeof(ptrdiff_t);

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//==============================================================================
//====  CROOTMARSHALER  ========================================================
//==============================================================================
//==============================================================================

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

bool CRootMarshaler::Store(IN const void* pData, IN WORD wSize)
{
    InsertTypeInfo(MARSHALING_TYPE_UNKNOWN);

    // Store the data length
    //----------------------
    *this << wSize;

    // Store the data
    //---------------
    Insert(pData, wSize);

    return true;
}

bool CRootMarshaler::Load(OUT void* pData, IN WORD wCapacity, OUT WORD& rwSize)
{
    ASSERT(!IsEmpty());

    bool bExtracted = true;

    // Check if the buffer in large enough and load the data
```

```
    //-------------------------------------------------------
    BYTE* pExtractPositionTemp = m_pExtractPosition;
    BYTE* pExtractSegmentEndTemp = m_pExtractSegmentEnd;

    ExtractTypeInfo(MARSHALING_TYPE_UNKNOWN);

    *this >> rwSize;

    if (wCapacity < rwSize)
    {
        // Restore the pointer to the position of the data lenght.
        // This is required for the next Load that may come with
        // a large enough buffer.
        m_pExtractPosition = pExtractPositionTemp;
        m_pExtractSegmentEnd = pExtractSegmentEndTemp;

        bExtracted = false;
    }
    else
    {
        Extract(pData, rwSize);
    }

    return bExtracted;
}

//=============================================================================
//=============================================================================
//====  CINPROCMARSHALER  =====================================================
//=============================================================================
//=============================================================================

//=============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ===============================================
//=============================================================================

CInprocMarshaler::CInprocMarshaler()
{
    m_pInsertPosition = m_pFirstSegment;
    m_pInsertSegmentEnd = m_pFirstSegment + MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SPACE;
    m_pExtractPosition = m_pFirstSegment;
    m_pExtractSegmentEnd = m_pFirstSegment + MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SPACE;

    // There is only one segment in the list
    *(reinterpret_cast<BYTE**>(m_pInsertSegmentEnd)) = NULL;
}

CInprocMarshaler::~CInprocMarshaler()
{
    ASSERT(IsEmpty());

    // Get pointer on second segment
    //------------------------------
    BYTE* pNextSegment = *reinterpret_cast<BYTE**>(m_pFirstSegment +
                         MARSHALER_INPROC_FIRST_MEMORY_SEGMENT_SPACE);
    BYTE* pNextSegmentTemp;

    // Release Subsequent Segments
    //----------------------------
    while (pNextSegment)
    {
        pNextSegmentTemp = *reinterpret_cast<BYTE**>(pNextSegment +
                           MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SPACE);
        ReleaseObject(pNextSegment);
        pNextSegment = pNextSegmentTemp;
    }
}

//=============================================================================
//====  PROTECTED FUNCTIONS  ==================================================
//=============================================================================
```

```
void CInprocMarshaler::Insert(IN const void* pData, IN WORD wSize)
{
    ASSERT_VALID_DATA_PTR(pData);

    // Calculate the size of the data that fit inside the actual segment
    //-------------------------------------------------------------------
    WORD wSegmentSpace = m_pInsertSegmentEnd - m_pInsertPosition;

    while (wSize > wSegmentSpace)
    {
        // Copy the data that fit in the actual segment
        //---------------------------------------------
        memcpy(m_pInsertPosition, pData, wSegmentSpace);

        // Set data pointer to point on the next data to be copied
        //--------------------------------------------------------
        pData = reinterpret_cast<void*>(reinterpret_cast<ptrdiff_t>(pData) + wSegmentSpace);
        wSize -= wSegmentSpace;

        // Allocate one more segment
        //--------------------------
        m_pInsertPosition = static_cast<BYTE*>(AllocateObject());

        // Link the new segment in the list
        //---------------------------------
        *(reinterpret_cast<BYTE**>(m_pInsertSegmentEnd)) = m_pInsertPosition;

        // Ajust the new segment space value
        //----------------------------------
        wSegmentSpace = MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SPACE;

        // Position the end pointer
        // NOTE: In the reserved space for linked list pointer
        //----------------------------------------------------
        m_pInsertSegmentEnd = m_pInsertPosition + wSegmentSpace;
        *(reinterpret_cast<BYTE**>(m_pInsertSegmentEnd)) = NULL;
    }

    // Copy the remaning data in the segment
    //--------------------------------------
    memcpy(m_pInsertPosition, pData, wSize);
    m_pInsertPosition += wSize;

    ASSERT(m_pInsertPosition <= m_pInsertSegmentEnd);
}

void CInprocMarshaler::Extract(OUT void* pData, IN WORD wSize)
{
    ASSERT_VALID_DATA_PTR(pData);
    ASSERT(wSize == 0 || !IsEmpty());

    // Calculate the size of the data that his inside the actual segment
    //------------------------------------------------------------------
    WORD wSegmentSpace = m_pExtractSegmentEnd - m_pExtractPosition;

    while (wSize > wSegmentSpace)
    {
        // Copy the data that his in the actual segment
        //---------------------------------------------
        memcpy(pData, m_pExtractPosition, wSegmentSpace);

        // Set data pointer on the next position where to copy data
        //---------------------------------------------------------
        pData = reinterpret_cast<void*>(reinterpret_cast<ptrdiff_t>(pData) + wSegmentSpace);
        wSize -= wSegmentSpace;

        // Locate next segment
        //--------------------
        m_pExtractPosition = *reinterpret_cast<BYTE**>(m_pExtractSegmentEnd);
```

```
        // Ajust the new segment space value
        //----------------------------------
        wSegmentSpace = MARSHALER_SUBSEQUENT_MEMORY_SEGMENT_SPACE;

        // Position the end pointer
        //-------------------------
        m_pExtractSegmentEnd = m_pExtractPosition + wSegmentSpace;
    }

    memcpy(pData, m_pExtractPosition, wSize);
    m_pExtractPosition += wSize;

    ASSERT(m_pExtractPosition <= m_pExtractSegmentEnd);
}

//============================================================================
//============================================================================
//====  COUTPROCMARSHALER  ===================================================
//============================================================================
//============================================================================

//============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ==============================================
//============================================================================
COutprocMarshaler::COutprocMarshaler(IN UINT16 wCapacity)
{
    wCapacity += sizeof(MARSHALER_BYTE_ORDERING);

    m_pBufferHead = new BYTE[wCapacity];
    m_pInsertPosition = m_pBufferHead + sizeof(MARSHALER_BYTE_ORDERING);
    m_pInsertSegmentEnd = m_pBufferHead + wCapacity;
    m_pExtractPosition = m_pBufferHead + sizeof(MARSHALER_BYTE_ORDERING);
    m_pExtractSegmentEnd = m_pInsertSegmentEnd;

    // Store the byte ordering to the octect stream
    m_btByteOrdering = MARSHALER_BYTE_ORDERING;
    *((UINT32*)m_pBufferHead) = MARSHALER_BYTE_ORDERING;
}

COutprocMarshaler::COutprocMarshaler(IN BYTE* pOctetStream, IN WORD wSize, IN WORD wCapacity)
{
    ASSERT_VALID_DATA_PTR(pOctetStream);
    ASSERT(wSize <= wCapacity);
    ASSERT(wCapacity > sizeof(MARSHALER_BYTE_ORDERING));

    m_pBufferHead = NULL;

    m_pInsertPosition = pOctetStream + Max(wSize, sizeof(MARSHALER_BYTE_ORDERING));
    m_pInsertSegmentEnd = pOctetStream + wCapacity;
    m_pExtractPosition = pOctetStream + sizeof(MARSHALER_BYTE_ORDERING);
    m_pExtractSegmentEnd = m_pInsertSegmentEnd;

    // Load the byte ordering from the octet stream
    if (wSize >= sizeof(MARSHALER_BYTE_ORDERING))
    {
        m_btByteOrdering = *((DWORD*)pOctetStream);
    }
    // Store the byte ordering to the octect stream
    else if (wCapacity > sizeof(MARSHALER_BYTE_ORDERING))
    {
        m_btByteOrdering = MARSHALER_BYTE_ORDERING;
        *((DWORD*)pOctetStream) = MARSHALER_BYTE_ORDERING;
    }
    else
    {
        ASSERT(0);
    }
}

COutprocMarshaler::~COutprocMarshaler()
{
```

```
    delete m_pBufferHead;
    m_pBufferHead = NULL;
}

BYTE* COutprocMarshaler::ReleaseBuffer()
{
    // TBD (sguenette) : This is not clean! Must always use external buffer...
    BYTE* m_pBufferHeadTemp = m_pBufferHead;
    m_pBufferHead = NULL;
    m_pInsertPosition = NULL;
    m_pInsertSegmentEnd = NULL;
    m_pExtractPosition = NULL;
    m_pExtractSegmentEnd = NULL;
    return m_pBufferHeadTemp;
}

//============================================================================
//====  PROTECTED FUNCTIONS  =================================================
//============================================================================

void COutprocMarshaler::Insert(IN const void* pData, IN WORD wSize)
{
    ASSERT_VALID_DATA_PTR(pData);
    ASSERT(wSize <= m_pInsertSegmentEnd - m_pInsertPosition);

    wSize = Min(m_pInsertSegmentEnd - m_pInsertPosition, wSize);
    memcpy(m_pInsertPosition, pData, wSize);
    m_pInsertPosition += wSize;

    ASSERT(m_pInsertPosition <= m_pInsertSegmentEnd);
}

void COutprocMarshaler::Extract(OUT void* pData, IN WORD wSize)
{
    ASSERT_VALID_DATA_PTR(pData);
    ASSERT(wSize == 0 || !IsEmpty());
    ASSERT(wSize <= m_pExtractSegmentEnd - m_pExtractPosition);

    wSize = Min(m_pExtractSegmentEnd - m_pExtractPosition, wSize);
    memcpy(pData, m_pExtractPosition, wSize);
    m_pExtractPosition += wSize;

    ASSERT(m_pExtractPosition <= m_pExtractSegmentEnd);
}

NAMESPACE_END(Mx)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectProxy.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __COBJECTPROXY_H__
#define __COBJECTPROXY_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

#include "Adt/CError.h"

// Data Member
#include "NetLayer/CSocketAddr.h"
#include "OsLayer/CSemaphore.h"

// Interface Realized & Parent

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CMessenger;
class COutprocMarshaler;
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CObjectProxy
//
//==EDOC========================================================================
class CObjectProxy : public CObject
{
// Child Methods
//---------------
protected:
    // << Constructors / Destructors >>
    //---------------------------------
    CObjectProxy(IN const ClassID& rClsid, IN const CSocketAddr* pProcessAddr = NULL);
    virtual ~CObjectProxy();

    CError Create(IN COutprocMarshaler* pInParams = NULL,
                  OUT COutprocMarshaler* pOutParams = NULL);
    CError Call(IN COutprocMarshaler* pInParams = NULL,
                OUT COutprocMarshaler* pOutParams = NULL);
    CError Destroy();

    const ClassID& GetClassID() const;
    CLink* GetLink();
```

```
    bool Wait();
    void Signal();

// Hidden Methods
//----------------
private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CObjectProxy(const CObjectProxy& rhs);
    CObjectProxy& operator=(const CObjectProxy& rhs);

// Hidden Data Members
//--------------------
protected:
    const ClassID m_clsid;

    CSocketAddr m_processAddr;
    CCountingSemaphore m_sem;

    CLink* m_pLink;

    CMessenger* m_pMessenger;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

inline
const ClassID& CObjectProxy::GetClassID() const
{
    return m_clsid;
}

inline
CLink* CObjectProxy::GetLink()
{
    return m_pLink;
}

inline
bool CObjectProxy::Wait()
{
    return m_sem.Wait();
}

inline
void CObjectProxy::Signal()
{
    m_sem.Signal();
}

NAMESPACE_END(Does)

#endif // #ifndef __COBJECTPROXY_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectProxy.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CObjectProxy.h"
#include "Does/CMessenger.h"

#include "Adt/CError.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CObjectProxy::CObjectProxy(IN const ClassID& rClsid, IN const CSocketAddr* pProcessAddr)
: m_clsid(rClsid),
  m_pLink(NULL)
{
    if (pProcessAddr != NULL)
    {
        m_processAddr = *pProcessAddr;
    }

    m_pMessenger = CMessenger::Instance();
    m_pMessenger->Register(this);
}

CObjectProxy::~CObjectProxy()
{
    if (m_pLink != NULL)
    {
        CError err;
        err = Destroy();
        ASSERT(!err());
    }
    m_pMessenger->Unregister(this);
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

CError CObjectProxy::Create(IN COutprocMarshaler* pInParams,
                            OUT COutprocMarshaler* pOutParams)
{
    ASSERT(m_processAddr.IsValid());
```

```
    ASSERT(m_pLink == NULL);

    CError err;

    if (m_pLink == false)
    {
        err << m_pMessenger->LinkProcess(m_processAddr);
        if (!err())
        {
            err << m_pMessenger->Create(this, m_processAddr, pInParams, pOutParams, &m_pLink);
        }
    }
    else
    {
        err.SetLevel(CError::eERROR);
        ASSERT(0);
    }

    return err;
}

CError CObjectProxy::Call(IN COutprocMarshaler* pInParams,
                          OUT COutprocMarshaler* pOutParams)
{
    ASSERT(m_pLink != NULL);

    CError err;

    if (m_pLink != NULL)
    {
        err << m_pMessenger->Call(m_pLink, pInParams, pOutParams);
    }
    else
    {
        err.SetLevel(CError::eERROR);
        ASSERT(0);
    }

    return err;
}

CError CObjectProxy::Destroy()
{
    ASSERT(m_pLink != NULL);

    CError err;

    if (m_pLink != NULL)
    {
        err << m_pMessenger->Destroy(m_pLink);

        if (!err())
        {
            m_pLink = NULL;
        }
    }
    else
    {
        err.SetLevel(CError::eERROR);
        ASSERT(0);
    }

    return err;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectAdapter.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __COBJECTADAPTER_H__
#define __COBJECTADAPTER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

#include "Adt/CError.h"

// Data Member
#include "OsLayer/CSemaphore.h"

// Interface Realized & Parent
#include "Threading/CMarshaler.h"
#include "Threading/CMsgDrivenObj.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CMessenger;
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CObjectAdapter
//
//==EDOC========================================================================
class CObjectAdapter : public CMsgDrivenObj
{
// Published Interface
//---------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CObjectAdapter(IN const ClassID& rClsid);
    virtual ~CObjectAdapter();

    CError Enable();
    CError Disable();

    void CreateA(IN CLink* pLink, IN COutprocMarshaler* pInParams);
    void CallA(IN CLink* pLink, IN void* pObject, IN COutprocMarshaler* pInParams);
    void DestroyA(IN CLink* pLink, IN void* pObject);

    const ClassID& GetClassID() const;
```

```
// Child Adapter Methods
//------------------------------
protected:
    virtual CError Create(IN OperationID opid,
                          IN COutprocMarshaler* pInParams,
                          OUT COutprocMarshaler** ppOutParams,
                          OUT void** ppObject) = 0;
    virtual CError Call(IN void* pObject,
                        IN OperationID opid,
                        IN COutprocMarshaler* pInParams,
                        OUT COutprocMarshaler** ppOutParams) = 0;
    virtual CError Destroy(IN void* pObject) = 0;

// Base Class methods overriding
//------------------------------
private:

    // CMsgDrivenObj Functions Overriding
    //-----------------------------------
    virtual void DispatchAsyncMsg(IN int nMsgNb, IN CMarshaler* pParameter);

// Internal Methods Definition
//----------------------------
private:
    // Message Number Definition
    enum EMsgNumber
    {
        MSG_CREATEA,
        MSG_CALLA,
        MSG_DESTROYA
    };

    void InternalCreateA(IN CMarshaler* pParams);
    void InternalCallA(IN CMarshaler* pParams);
    void InternalDestroyA(IN CMarshaler* pParams);

// Hidden Methods
//---------------
private:
    // Deactivation of the copy constructor and the assignment operator
    //-----------------------------------------------------------------
    CObjectAdapter(IN const CObjectAdapter& rhs);
    CObjectAdapter& operator=(IN const CObjectAdapter& rhs);

// Hidden Data Members
//--------------------
private:
    const ClassID m_clsid;

    CMessenger* m_pMessenger;

    CCountingSemaphore m_semAlive;
};

//============================================================================
//=====  INLINE FUNCTIONS  ===================================================
//============================================================================

inline
const ClassID& CObjectAdapter::GetClassID() const
{
    return m_clsid;
}

NAMESPACE_END(Does)

#endif // #ifndef __COBJECTADAPTER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectAdapter.cpp
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CObjectAdapter.h"
#include "Does/CMessenger.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CObjectAdapter::CObjectAdapter(IN const ClassID& rClsid)
: CMsgDrivenObj(0),
  m_clsid(rClsid),
  m_pMessenger(NULL)
{
}

CObjectAdapter::~CObjectAdapter()
{
    ASSERT(m_pMessenger == NULL);
}

//==============================================================================
//====  PROTECTED FUNCTIONS  ===================================================
//==============================================================================

CError CObjectAdapter::Enable()
{
    CError err;

    m_pMessenger = CMessenger::Instance();
    err = m_pMessenger->Register(this);

    if (!err())
    {
        Activate(&m_semAlive);
    }

    return err;
}

CError CObjectAdapter::Disable()
{
    CError err;
```

```
    err = m_pMessenger->Unregister(this);

    if (!err())
    {
        Terminate();
        m_semAlive.Wait();
        m_pMessenger = NULL;
    }

    return err;
}

//=============================================================================
//====  PRIVATE FUNCTIONS  ====================================================
//=============================================================================

void CObjectAdapter::CreateA(IN CLink* pLink, IN COutprocMarshaler* pInParams)
{
    CMarshaler* pMarshaler = new CMarshaler;
    *pMarshaler << pLink
                << pInParams;

    ProcessAsyncMsg(MSG_CREATEA, pMarshaler);
}

void CObjectAdapter::InternalCreateA(IN CMarshaler* pParams)
{
    CLink* pLink;
    COutprocMarshaler* pInParams;
    *pParams >> pLink
             >> pInParams;

    CError err;
    OperationID opid;
    COutprocMarshaler* pOutParams = NULL;
    void* pObject = NULL;

    *pInParams >> opid;

    err = Create(opid,
                 pInParams,
                 &pOutParams,
                 &pObject);

    if (!err())
    {
        m_pMessenger->CreatedA(pLink, pObject, pOutParams);
    }
    else
    {
        ASSERT(0);
        // TBD (sguenette) : return error msg
    }

    delete pInParams;
    pInParams = NULL;
}

void CObjectAdapter::CallA(IN CLink* pLink, IN void* pObject, IN COutprocMarshaler* pInParams)
{
    CMarshaler* pMarshaler = new CMarshaler;
    *pMarshaler << pLink
                << pObject
                << pInParams;

    ProcessAsyncMsg(MSG_CALLA, pMarshaler);
}

void CObjectAdapter::InternalCallA(IN CMarshaler* pParams)
{
```

```
    CLink* pLink;
    void* pObject;
    COutprocMarshaler* pInParams;
    *pParams >> pLink
             >> pObject
             >> pInParams;

    CError err;
    OperationID opid;
    COutprocMarshaler* pOutParams = NULL;

    *pInParams >> opid;

    err = Call(pObject,
               opid,
               pInParams,
               &pOutParams);

    if (!err())
    {
        m_pMessenger->ReturnA(pLink, pOutParams);
    }
    else
    {
        ASSERT(0);
        // TBD (sguenette) : return error msg
    }

    delete pInParams;
    pInParams = NULL;
}

void CObjectAdapter::DestroyA(IN CLink* pLink, IN void* pObject)
{
    CMarshaler* pMarshaler = new CMarshaler;
    *pMarshaler << pLink
                << pObject;

    ProcessAsyncMsg(MSG_DESTROYA, pMarshaler);
}

void CObjectAdapter::InternalDestroyA(IN CMarshaler* pParams)
{
    CLink* pLink;
    void* pObject;
    *pParams >> pLink
             >> pObject;

    CError err;

    err = Destroy(pObject);

    if (!err())
    {
        m_pMessenger->DestroyedA(pLink);
    }
    else
    {
        ASSERT(0);
        // TBD (sguenette) : return error msg
    }
}

//=============================================================================
//====  BASE CLASS OVERRIDING METHOD  =========================================
//=============================================================================

void CObjectAdapter::DispatchAsyncMsg(IN int nMsgNb, IN CMarshaler* pParameter)
{
    switch (nMsgNb)
    {
```

```
        case MSG_CREATEA:   InternalCreateA(pParameter); break;
        case MSG_CALLA:     InternalCallA(pParameter);  break;
        case MSG_DESTROYA:  InternalDestroyA(pParameter); break;
        case MSG_TERMINATE: InternalTerminate(pParameter); break;
        default: ASSERT(false);
    }
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CClientTransmitter.h
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CCLIENTTRANSMITTER_H__
#define __CCLIENTTRANSMITTER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

// Data Member

// Interface Realized & Parent
#include "Does/ITransmitter.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CAsyncSocket;
class CMessage;
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CClientTransmitter
//
//==EDOC========================================================================
class CClientTransmitter : public CObject, public ITransmitter
{
// Published Interface
//-------------------------
public:
    static void SetSocket(IN CAsyncSocket* pSocket);

    // << Constructors / Destructors >>
    //-----------------------------------------
    CClientTransmitter(IN CLink* pLink);
    virtual ~CClientTransmitter();

    virtual bool TxMsg(IN CMessage* pMsg);
    virtual bool TxAck();
    virtual bool RxAck();

// Hidden Methods
//-------------------------
private:

    // Deactivated Constructors / Destructors / Operators
```

```
    //-------------------------------------------------
    CClientTransmitter(IN const CClientTransmitter& rhs);
    CClientTransmitter& operator=(IN const CClientTransmitter& rhs);

// Hidden Data Members
//-------------------------
private:
    static CAsyncSocket* m_pSocket;

    enum EStates
    {
        eIdle,
        eTransmitting,
        eWaiting,
        eAcking
    } m_Status;

    CLink* m_pLink;
    CMessage* m_pMsg;
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CCLIENTTRANSMITTER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CClientTransmitter.cpp
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CClientTransmitter.h"
#include "Does/CMessage.h"
#include "Does/CLink.h"

#include "NetLayer/CAsyncSocket.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

CAsyncSocket* CClientTransmitter::m_pSocket = NULL;

void CClientTransmitter::SetSocket(IN CAsyncSocket* pSocket)
{
    m_pSocket = pSocket;
}

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CClientTransmitter::CClientTransmitter(IN CLink* pLink)
{
    m_Status = eIdle;
    m_pLink = pLink;
    m_pMsg = NULL;
}

CClientTransmitter::~CClientTransmitter()
{
    delete m_pMsg;
    m_pMsg = NULL;
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

bool CClientTransmitter::TxMsg(IN CMessage* pMsg)
{
    ASSERT(pMsg->IsAction());

    bool bTransmitted = true;

    if (m_Status == eIdle ||
```

```
        m_Status == eAcking)
    {
        m_Status = eTransmitting;
        m_pLink->m_msn++;
        ((hdrRoot*)pMsg)->msn = m_pLink->m_msn;

        // Copy msg to blob
        CBlob* pData = new CBlob();
        if (pMsg->CopyToBlob(pData))
        {
            ASSERT(m_pSocket != NULL);
            m_pSocket->SendToA(pData, m_pLink->m_destAddr);
        }
        else
        {
            ASSERT(0);
        }

        ASSERT(m_pMsg == NULL); // Check that old message was deleted
        delete m_pMsg; // For security...
        m_pMsg = pMsg; // Keep a copy for future transmission
    }
    else
    {
        // Must be an action only, no ACK or NAK.
        // Transmission can occure only from eIdle or eAcking state
        // There is a protocole error in this case.
        ASSERT(0);
        bTransmitted = false;
        delete pMsg;
        pMsg = NULL;
    }

    return bTransmitted;
}

bool CClientTransmitter::TxAck()
{
    bool bAcked = true;

    // WARNING: For now, we step over the Acking state. So all state Leeds to
    //          the eIdle state after acking immediately.
    // TBD (sguenette) : This method must be reworked when timers are ready.
    //-------------------------------------------------------------------------

    switch (m_Status)
    {
    case eTransmitting:
        // The message was transmitted successfully, so we can delete it.
        // TBD (sguenette) : stop timer!
        delete m_pMsg;
        m_pMsg = NULL;
    case eIdle:
    case eAcking: // Must implement timers for this state to be useful.
    case eWaiting:
        {
            m_Status = eIdle;

            CMessage* pMsg = new CMessage();
            new ((hdrRoot*)pMsg) hdrRoot(m_pLink->m_linkid.lkn, aACK, m_pLink->m_msn);
            pMsg->Resize(sizeof(hdrAck));

            // Copy msg to blob
            CBlob* pData = new CBlob();
            if (pMsg->CopyToBlob(pData))
            {
                ASSERT(m_pSocket != NULL);
                m_pSocket->SendToA(pData, m_pLink->m_destAddr);
            }
            else
            {
```

```
                ASSERT(0);
            }

            delete pMsg;
            pMsg = NULL;
        }
        break;
    default:
        // Unknown status ???
        ASSERT(0);
        bAcked = false;
    }

    return bAcked;
}

bool CClientTransmitter::RxAck()
{
    bool bSuccess = true;

    if (m_Status == eTransmitting)
    {
        m_Status = eWaiting;
        // TBD (sguenette) : stop timer!
        // The message was transmitted successfully, so we can delete it.
        delete m_pMsg;
        m_pMsg = NULL;
    }
    else
    {
        // Unknown state or protocol error!
        ASSERT(0);
        bSuccess = false;
    }

    return bSuccess;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CClientReceptor.h
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CCLIENTRECEPTOR_H__
#define __CCLIENTRECEPTOR_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

// Data Member

// Interface Realized & Parent
#include "Does/IReceptor.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CMessenger;
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CClientReceptor
//
//==EDOC========================================================================
class CClientReceptor : public CObject, public IReceptor
{
// Published Interface
//-------------------------
public:
    // << Constructors / Destructors >>
    //-----------------------------------------
    CClientReceptor(IN CLink* pLink);
    virtual ~CClientReceptor() {};

    bool RxMsg(IN CMessage* pMsg);

// Hidden Methods
//-------------------------
private:

    // Deactivated Constructors / Destructors / Operators
    //-------------------------------------------------
    CClientReceptor(IN const CClientReceptor& rhs);
    CClientReceptor& operator=(IN const CClientReceptor& rhs);

// Hidden Data Members
```

```
//-------------------------
private:
    enum EStates
    {
        eIdle,
        eWaiting
    } m_Status;

    CLink* m_pLink;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CRECEPTOR_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CClientReceptor.cpp
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CClientReceptor.h"
#include "Does/CLink.h"
#include "Does/CMessage.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CClientReceptor::CClientReceptor(IN CLink* pLink)
{
    ASSERT(pLink != NULL);

    m_Status = eIdle;
    m_pLink = pLink;
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

bool CClientReceptor:: RxMsg(IN CMessage* pMsg)
{
    ASSERT(pMsg != NULL);

    bool bReceived = true;

    // WARNING: At this point, the message must hold a reaction.
    //-----------------------------------------------------------
    if (pMsg->IsReaction())
    {
        if (m_Status == eIdle)
        {
            if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn)
            {
                if (((hdrRoot*)pMsg)->aid == rACK)
                {
                    m_Status = eWaiting;
                    m_pLink->RxAck();
                }
                else
                {
```

```
            //  This is the last message, so we ack it again.
            m_pLink->TxAck();
        }

        delete pMsg;
        pMsg = NULL;
    }
    else if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn + 1)
    {
        if (((hdrRoot*)pMsg)->aid != rACK &&
            ((hdrRoot*)pMsg)->aid != rNAK)
        {
            m_pLink->m_msn++;
            m_pLink->TxAck();

            if (!m_pLink->ExecReaction(pMsg))
            {
                // TBD (sguenette) : handle error and tx NAK
                ASSERT(0);
            }

            pMsg = NULL;
        }
        else
        {
            // Must be a reaction only, no ACK or NAK.
            // There is a protocole error in this case.
            ASSERT(0);
            bReceived = false;
            delete pMsg;
            pMsg = NULL;
        }
    }
    else
    {
        // Do nothing; this is an old message!
        // We just delete the message and we
        // return no error.
        delete pMsg;
        pMsg = NULL;
    }
}
else if (m_Status == eWaiting)
{
    if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn + 1)
    {
        if (((hdrRoot*)pMsg)->aid != rACK &&
            ((hdrRoot*)pMsg)->aid != rNAK)
        {
            m_Status = eIdle;
            m_pLink->m_msn++;
            m_pLink->TxAck();

            if (!m_pLink->ExecReaction(pMsg))
            {
                // TBD (sguenette) : handle error and tx NAK
                ASSERT(0);
            }

            pMsg = NULL;
        }
        else
        {
            // Must be a reaction only, no ACK or NAK.
            // There is a protocole error in this case.
            ASSERT(0);
            bReceived = false;
            delete pMsg;
            pMsg = NULL;
        }
    }
```

```
            else
            {
                // Do nothing; this is an old message!
                // We just delete the message and we
                // return no error.
                delete pMsg;
                pMsg = NULL;
            }
        }
        else
        {
            // Unknown status ???
            ASSERT(0);
            bReceived = false;
            delete pMsg;
            pMsg = NULL;
        }
    }
    else
    {
        // Not a Reaction ???
        ASSERT(0);
        bReceived = false;
        delete pMsg;
        pMsg = NULL;
    }

    return bReceived;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CServerTransmitter.h
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CSERVERTRANSMITTER_H__
#define __CSERVERTRANSMITTER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

// Data Member
#include "NetLayer/CAsyncSocket.h"

// Interface Realized & Parent
#include "Does/ITransmitter.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CMessage;
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CServerTransmitter
//
//==EDOC========================================================================
class CServerTransmitter : public CObject, public ITransmitter
{
// Published Interface
//------------------------
public:
    static void SetSocket(IN CAsyncSocket* pSocket);

    // << Constructors / Destructors >>
    //-----------------------------------------
    CServerTransmitter(IN CLink* pLink);
    virtual ~CServerTransmitter();

    virtual bool TxMsg(IN CMessage* pMsg);
    virtual bool TxAck();
    virtual bool RxAck();

// Hidden Methods
//------------------------
private:

    // Deactivated Constructors / Destructors / Operators
```

```
    //-------------------------------------------------
    CServerTransmitter(IN const CServerTransmitter& rhs);
    CServerTransmitter& operator=(IN const CServerTransmitter& rhs);

// Hidden Data Members
//-------------------------
private:
    static CAsyncSocket* m_pSocket;

    enum EStates
    {
        eIdle,
        eTransmitting,
        eWaiting,
        eAcking
    } m_Status;

    CLink* m_pLink;
    CMessage* m_pMsg;
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CSERVERTRANSMITTER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CServerTransmitter.cpp
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CServerTransmitter.h"
#include "Does/CMessage.h"
#include "Does/CLink.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

CAsyncSocket* CServerTransmitter::m_pSocket = NULL;

void CServerTransmitter::SetSocket(IN CAsyncSocket* pSocket)
{
    m_pSocket = pSocket;
}

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CServerTransmitter::CServerTransmitter(IN CLink* pLink)
{
    ASSERT(m_pSocket != NULL);

    m_Status = eIdle;
    m_pLink = pLink;
    m_pMsg = NULL;
}

CServerTransmitter::~CServerTransmitter()
{
    delete m_pMsg;
    m_pMsg = NULL;
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

bool CServerTransmitter::TxMsg(IN CMessage* pMsg)
{
    ASSERT(pMsg->IsReaction());

    bool bTransmitted = true;

    if (m_Status == eAcking ||
```

```
        m_Status == eWaiting)
    {
        m_Status = eTransmitting;
        m_pLink->m_msn++;
        ((hdrRoot*)pMsg)->msn = m_pLink->m_msn;

        // Copy msg to blob
        CBlob* pData = new CBlob();
        if (pMsg->CopyToBlob(pData))
        {
            ASSERT(m_pSocket != NULL);
            m_pSocket->SendToA(pData, m_pLink->m_destAddr);
        }
        else
        {
            ASSERT(0);
        }

        ASSERT(m_pMsg == NULL);
        delete m_pMsg; // For security...
        m_pMsg = pMsg; // Keep a copy for future transmission
    }
    else
    {
        // Must be an action only, no ACK or NAK.
        // There is a protocole error in this case.
        ASSERT(0);
        bTransmitted = false;
        delete pMsg;
        pMsg = NULL;
    }

    return bTransmitted;
}

bool CServerTransmitter::TxAck()
{
    bool bAcked = true;

    // WARNING: For now, we step over the Acking state. So all state Leeds to
    //          the eWaiting state after acking immediately.
    // TBD (sguenette) : This method must be reworked when timers are ready.
    //-------------------------------------------------------------------------

    switch (m_Status)
    {
    case eTransmitting:
        // The message was transmitted successfully, so we can delete it.
        // TBD (sguenette) : stop timer!
        delete m_pMsg;
        m_pMsg = NULL;
    case eIdle:
    case eAcking: // Must implement timers for this state to be useful.
    case eWaiting:
        {
            CMessage* pMsg = new CMessage();
            new ((hdrRoot*)pMsg) hdrRoot(m_pLink->m_linkid.lkn, rACK, m_pLink->m_msn);
            pMsg->Resize(sizeof(hdrAck));

            // Copy msg to blob
            CBlob* pData = new CBlob();
            if (pMsg->CopyToBlob(pData))
            {
                ASSERT(m_pSocket != NULL);
                m_pSocket->SendToA(pData, m_pLink->m_destAddr);
            }
            else
            {
                ASSERT(0);
            }
```

```
            delete pMsg;
            pMsg = NULL;
        }
        break;
    default:
        // Unknown status ???
        ASSERT(0);
        bAcked = false;
    }

    return bAcked;
}

bool CServerTransmitter::RxAck()
{
    bool bSuccess = true;

    if (m_Status == eTransmitting)
    {
        m_Status = eIdle;
        // TBD (sguenette) : stop timer!
        // The message was transmitted successfully, so we can delete it.
        delete m_pMsg;
        m_pMsg = NULL;
    }
    else
    {
        // Unknown state or protocol error!
        ASSERT(0);
        bSuccess = false;
    }

    return bSuccess;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CServerReceptor.h
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CSERVERRECEPTOR_H__
#define __CSERVERRECEPTOR_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

// Data Member

// Interface Realized & Parent
#include "Does/IReceptor.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CLink;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CServerReceptor
//
//==EDOC========================================================================
class CServerReceptor : public CObject, public IReceptor
{
// Published Interface
//-------------------------
public:
    // << Constructors / Destructors >>
    //-----------------------------------------
    CServerReceptor(IN CLink* pLink);
    virtual ~CServerReceptor() {};

    virtual bool RxMsg(IN CMessage* pMsg);

// Hidden Methods
//-------------------------
private:
    // Deactivated Constructors / Destructors / Operators
    //-----------------------------------------------------
    CServerReceptor(IN const CServerReceptor& rhs);
    CServerReceptor& operator=(IN const CServerReceptor& rhs);

// Hidden Data Members
//-------------------------
private:
```

```
    enum EStates
    {
        eIdle,
        eWaiting
    } m_Status;

    CLink* m_pLink;
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CSERVERRECEPTOR_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CServerReceptor.cpp
//
//  Package: Does
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Does/CServerReceptor.h"
#include "Does/CLink.h"
#include "Does/CMessage.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CServerReceptor::CServerReceptor(IN CLink* pLink)
{
    ASSERT(pLink != NULL);

    m_Status = eIdle;
    m_pLink = pLink;
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

bool CServerReceptor:: RxMsg(IN CMessage* pMsg)
{
    ASSERT(pMsg != NULL);

    bool bReceived = true;

    // WARNING: At this point, the message must hold an action.
    //-----------------------------------------------------------
    if (pMsg->IsAction())
    {
        if (m_Status == eIdle)
        {
            if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn + 1)
            {
                if (((hdrRoot*)pMsg)->aid != aACK &&
                    ((hdrRoot*)pMsg)->aid != aNAK)
                {
                    m_Status = eWaiting;
                    m_pLink->m_msn++;
                    m_pLink->TxAck();
```

```
                if (!m_pLink->ExecAction(pMsg))
                {
                    // TBD (sguenette) : handle error and tx NAK
                    ASSERT(0);
                }

                pMsg = NULL;
            }
            else
            {
                // Must be an action only, no ACK or NAK.
                // There is a protocole error in this case.
                ASSERT(0);
                bReceived = false;
                delete pMsg;
                pMsg = NULL;
            }
        }
        else
        {
            // Do nothing; this is an old message!
            // We just delete the message and we
            // return no error.
            delete pMsg;
            pMsg = NULL;
        }
    }
    else if (m_Status == eWaiting)
    {
        if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn + 1)
        {
            if (((hdrRoot*)pMsg)->aid != aACK &&
                ((hdrRoot*)pMsg)->aid != aNAK)
            {
                m_pLink->m_msn++;
                m_pLink->TxAck();

                if (!m_pLink->ExecAction(pMsg))
                {
                    // TBD (sguenette) : handle error and tx NAK
                    ASSERT(0);
                }

                pMsg = NULL;
            }
            else
            {
                // Must be an action only, no ACK or NAK.
                // There is a protocole error in this case.
                ASSERT(0);
                bReceived = false;
                delete pMsg;
                pMsg = NULL;
            }
        }
        else if (((hdrRoot*)pMsg)->msn == m_pLink->m_msn)
        {
            if (((hdrRoot*)pMsg)->aid == aACK ||
                ((hdrRoot*)pMsg)->aid == aNAK)
            {
                m_Status = eIdle;
                m_pLink->RxAck();
                delete pMsg;
                pMsg = NULL;
            }
            else
            {
                // Must be an action only, no ACK or NAK.
                // There is a protocole error in this case.
                ASSERT(0);
                bReceived = false;
```

```
                    delete pMsg;
                    pMsg = NULL;
                }
            }
            else
            {
                // Do nothing; this is an old message!
                // We just delete the message and we
                // return no error.
                delete pMsg;
                pMsg = NULL;
            }
        }
        else
        {
            // Unknown status ???
            ASSERT(0);
            bReceived = false;
            delete pMsg;
            pMsg = NULL;
        }
    }
    else
    {
        // Not an Action ???
        ASSERT(0);
        bReceived = false;
        delete pMsg;
        pMsg = NULL;
    }

    return bReceived;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMessenger.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CMESSENGER_H__
#define __CMESSENGER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================
#include "Does/Defines.h"

// Data Member
#include "Adt/CError.h"
#include "NetLayer/CAsyncSocket.h"
#include "NetLayer/CSocketAddr.h"

// Interface Realized & Parent
#include "Patterns/CSingleton.h"
#include "Threading/CMsgDrivenObj.h"
#include "NetLayer/ISocketMgr.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace
class CLink;
class CLinker;
class CObjectProxy;
class CObjectAdapter;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CMessenger
//
//==EDOC========================================================================
class CMessenger : public CSingleton<CMessenger>,
                   public CMsgDrivenObj,
                   public ISocketMgr
{
    friend class CSingleton<CMessenger>;

// Published Interface
//--------------------
public:
    CError Enable(IN ProcessID pid);
    CError Disable();

    // Proxy and Adapter registration
    //-------------------------------
    CError Register(IN CObjectProxy* pProxy);
```

```
    CError Register(IN CObjectAdapter* pAdapter);
    CError Unregister(IN CObjectProxy* pProxy);
    CError Unregister(IN CObjectAdapter* pAdapter);

    CError LinkProcess(IN const CSocketAddr& rProcessAddr);

    // Action by the Proxy
    CError Create(IN CObjectProxy* pProxy,
                  IN CSocketAddr& rProcessAddr,
                  IN COutprocMarshaler* pInParams,
                  OUT COutprocMarshaler* pOutParams,
                  OUT CLink** ppLink);
    CError Call(IN CLink* pLink,
                IN COutprocMarshaler* pInParams,
                OUT COutprocMarshaler* pOutParams);
    CError Destroy(IN CLink* pLink);

    // Reaction by the Adapter
    void CreatedA(IN CLink* pLink,
                  IN void* pObject,
                  IN COutprocMarshaler* pOutParams);
    void ReturnA(IN CLink* pLink,
                 IN COutprocMarshaler* pOutParams);
    void DestroyedA(IN CLink* pLink);

    // Interface ISocketMgr
    //---------------------
    void SocketReceivedA(IN UINT unSockID, IN TO CBlob* pData,
                         IN const CSocketAddr& rPeerAddr);
    void SocketConnectedA(IN UINT unSockID,
                          IN const CSocketAddr& rPeerAddr) {ASSERT(0);};
    void SocketAcceptedA(IN UINT unSockID,
                         IN TO CAsyncSocket* pNewSock,
                         IN const CSocketAddr& rPeerAddr) {ASSERT(0);};
    void SocketReadyToSendA(IN UINT unSockID) {ASSERT(0);};
    void SocketReceivedOutOfBandA(IN UINT unSockID,
                        IN TO CBlob* pData,
                        IN const CSocketAddr& rPeerAddr) {ASSERT(0);};
    void SocketClosingA(IN UINT unSockID) {ASSERT(0);};
    void SocketClosedA(IN UINT unSockID) {ASSERT(0);};
    void SocketErrorA(IN UINT unSockID,
                      IN CError lastError) {ASSERT(0);};

// Hidden Methods
//---------------
private:
    // Action execution
    CError ExecCreate(IN CLink* pLink);
    CError ExecCall(IN CLink* pLink);
    CError ExecDestroy(IN CLink* pLink);

    // Reaction execution
    CError ExecCreated(IN CLink* pLink);
    CError ExecReturn(IN CLink* pLink);
    CError ExecDestroyed(IN CLink* pLink);

    // Deactivation of the copy constructor and the assignment operator
    //-----------------------------------------------------------------
    CMessenger(IN const CMessenger& rhs);
    CMessenger& operator=(IN const CMessenger& rhs);

// Base Class methods overriding
//------------------------------
private:
    // << Constructors / Destructors >>
    //---------------------------------
    CMessenger();
    virtual ~CMessenger();

    // CMsgDrivenObj Functions Overriding
    //-----------------------------------
```

```
    virtual void DispatchAsyncMsg(IN int nMsgNb, IN CMarshaler* pParams);
    virtual void DispatchSyncMsg(IN int nMsgNb, IN CMarshaler* pParams, OUT CError& rError);

// Internal Methods Definition
//----------------------------
private:
    // Message Number Definition
    enum EMsgNumber
    {
        MSG_LINKPROC,
        MSG_REGISTERPROXY,
        MSG_REGISTERADAPTER,
        MSG_UNREGISTERPROXY,
        MSG_UNREGISTERADAPTER,
        MSG_CREATE,
        MSG_CREATEDA,
        MSG_CALL,
        MSG_RETURNA,
        MSG_DESTROY,
        MSG_DESTROYEDA,
        MSG_SOCKETRECEIVEDA
    };

    void InternalRegisterProxy(IN CMarshaler* pParams, OUT CError& rError);
    void InternalRegisterAdapter(IN CMarshaler* pParams, OUT CError& rError);
    void InternalUnregisterProxy(IN CMarshaler* pParams, OUT CError& rError);
    void InternalUnregisterAdapter(IN CMarshaler* pParams, OUT CError& rError);
    void InternalLinkProcess(IN CMarshaler* pParams);
    void InternalCreate(IN CMarshaler* pParams);
    void InternalCall(IN CMarshaler* pParams);
    void InternalDestroy(IN CMarshaler* pParams);
    void InternalCreatedA(IN CMarshaler* pParams);
    void InternalReturnA(IN CMarshaler* pParams);
    void InternalDestroyedA(CN IMarshaler* pParams);
    void InternalSocketReceivedA(IN CMarshaler* pParams);

// Hidden Data Members
//--------------------
private:
    CCountingSemaphore m_semAlive;
    CLinker* m_pLinker;
    CAsyncSocket* m_pSocket;
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CMESSENGER_H__
```

Annexe 2 : classes de gestion mémoire

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectAllocator.h
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __COBJECTALLOCATOR_H__
#define __COBJECTALLOCATOR_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member
#include "Memory/CObjectPool.h"
#include "OsLayer/CSemaphore.h"

// Interface Realized & Parent

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Mx)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================
#if defined(_Type)
#error "_Type" is already defined as a MACRO. Cannot be used again in template definition.
#endif

//==SDOC========================================================================
//
//  class CObjectAllocator
//
//==EDOC========================================================================
template<class _Type>
class CObjectAllocator : public CObject
{
// Published Interface
//--------------------
public:
    CObjectAllocator() {};
    virtual ~CObjectAllocator() {};

    static void InitAllocator(IN int nSizeOfPool, IN bool bEnableThreadProtection = true);

    void* operator new(size_t size);
    void operator delete(void* ptr);

    static void* AllocateObject();
    static void  ReleaseObject(void* ptr);
```

```
// Hidden Methods
//---------------
private:
    // Deactivated Constructors / Destructors / Operators
    //----------------------------------------------------
    CObjectAllocator(const CObjectAllocator& rhs);
    CObjectAllocator& operator=(const CObjectAllocator& rhs);

// Hidden Data Members
//--------------------
private:
    static CObjectPool<_Type>* ms_pObjectPool;
    static CCountingSemaphore* ms_pSem;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

template<class _Type>
void CObjectAllocator<_Type>::InitAllocator(IN int nSizeOfPool, IN bool bEnableThreadProtection)
{
    ASSERT(nSizeOfPool > 0);

    if (ms_pObjectPool == NULL)
    {
        ms_pObjectPool = new CObjectPool<_Type>(nSizeOfPool);
    }

    if (ms_pSem == NULL && bEnableThreadProtection == true)
    {
        ms_pSem = new CCountingSemaphore(1);
    }
}

template<class _Type>
inline
void* CObjectAllocator<_Type>::operator new(size_t size)
{
    ASSERT(size > 0);
    ASSERT(size <= sizeof(_Type));

    return AllocateObject();
}

template<class _Type>
inline
void CObjectAllocator<_Type>::operator delete(void* ptr)
{
    ReleaseObject(ptr);
}

template<class _Type>
inline
void* CObjectAllocator<_Type>::AllocateObject()
{
    ASSERT(ms_pObjectPool != NULL);

    if (ms_pSem == NULL)
    {
        return ms_pObjectPool->AllocateObject();
    }
    else
    {
        ms_pSem->Wait();
        void* pObject = ms_pObjectPool->AllocateObject();
        ms_pSem->Signal();
        return pObject;
    }
}
```

```
template<class _Type>
inline
void CObjectAllocator<_Type>::ReleaseObject(void* ptr)
{
    ASSERT(ms_pObjectPool != NULL);

    if (ms_pSem == NULL)
    {
        ms_pObjectPool->ReleaseObject(ptr);
    }
    else
    {
        ms_pSem->Wait();
        ms_pObjectPool->ReleaseObject(ptr);
        ms_pSem->Signal();
    }
}

//============================================================================
//====  STATIC MEMBERS INITIALIZATION  ======================================
//============================================================================

template<class _Type>
CObjectPool<_Type>* CObjectAllocator<_Type>::ms_pObjectPool = NULL;

template<class _Type>
CCountingSemaphore* CObjectAllocator<_Type>::ms_pSem = NULL;

NAMESPACE_END(Mx)

#endif // #ifndef __COBJECTALLOCATOR_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CObjectPool.h
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __COBJECTPOOL_H__
#define __COBJECTPOOL_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member
#include "Memory/CMemoryPage.h"

// Interface Realized & Parent

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Mx)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================
#if defined(_Type)
#error "_Type" is already defined as a MACRO. Cannot be used again in template definition.
#endif

//==SDOC========================================================================
//
//  class CObjectPool
//
//==EDOC========================================================================
template<class _Type>
class CObjectPool : public CObject
{
// Published Interface
//--------------------
public:
    CObjectPool(IN size_t nSize);
    virtual ~CObjectPool();

    void* AllocateObject();
    void ReleaseObject(void* pObject);

// Hidden Methods
//---------------
private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CObjectPool(const CObjectPool& rhs);
    CObjectPool& operator=(const CObjectPool& rhs);

// Hidden Data Members
//--------------------
```

```
private:
    CMemoryPage* m_pPage;
    CMemoryPage::EState m_PageState;
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

template<class _Type>
CObjectPool<_Type>::CObjectPool(IN size_t nSize)
{
    int nPageSize = nSize * sizeof(_Type) + sizeof(CMemoryPage);
    void* pBuffer = new BYTE[nPageSize];
    m_pPage = new (pBuffer) CMemoryPage(nPageSize, sizeof(_Type));
    m_PageState = CMemoryPage::eEmpty;
    ASSERT(pBuffer == m_pPage != NULL);
}

template<class _Type>
CObjectPool<_Type>::~CObjectPool()
{
    // Don't need to delete the memory page because
    // its is included inside the pool buffer
    //---------------------------------------------
    delete ((BYTE*)m_pPage);
}

template<class _Type>
inline
void* CObjectPool<_Type>::AllocateObject()
{
    void* pObject = NULL;

    if (m_PageState != CMemoryPage::eFull)
    {
        m_PageState = m_pPage->AllocateBlock(pObject);
    }
    else
    {
        T1("ERROR: An object pool is Empty!\n");
        ASSERT(0);
    }

    return pObject;
}

template<class _Type>
inline
void CObjectPool<_Type>::ReleaseObject(void* pObject)
{
    m_PageState = m_pPage->ReleaseBlock(pObject);
}

NAMESPACE_END(Mx)

#endif // #ifndef __COBJECTPOOL_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMetaMemoryPage.h
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CMETAMEMORYPAGE_H__
#define __CMETAMEMORYPAGE_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member

// Interface Realized & Parent

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Mx)

// Forward Declarations Inside of the Namespace
class CMemoryPage;

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CMetaMemoryPage
//
//==EDOC========================================================================
class CMetaMemoryPage : public CObject
{
// Published Interface
//---------------------
public:
    // Constructor allocate pages of memory
    //--------------------------------------
    CMetaMemoryPage(IN size_t nPageSize, IN DWORD dwNumberOfPageInMetaPage);
    ~CMetaMemoryPage();

    // Memory allocation
    //-------------------
    void* AllocateBlock(IN size_t nBlockSize);
    bool ReleaseBlock(IN void* ptr);

    // << allocator >>
    //-----------------
    void* operator new (size_t size);
    void  operator delete (void* ptr);

// Hidden Methods
//----------------
private:
    // Memory Page Allocation and initialization
    //-------------------------------------------
```

```
    void AllocatePages(IN size_t nPageSize, IN DWORD dwNumberOfPages);

    // Deactivated Constructors / Destructors / Operators
    //----------------------------------------------------
    CMetaMemoryPage(const CMetaMemoryPage& rhs);
    CMetaMemoryPage& operator=(const CMetaMemoryPage& rhs);
    void* operator new [] (size_t size);
    void  operator delete [] (void* ptr);

// Hidden Data Members
//--------------------
private:
    // Memory Meta-Page starting and ending address
    //---------------------------------------------
    void* m_pMetaPageBegin;
    void* m_pMetaPageEnd; // Point one byte after the meta-page buffer

    // Available memory inside a page and total size of a page
    //--------------------------------------------------------
    size_t m_nPageSize;
    size_t m_nPageFreeSpace;

    // Pages List Reference
    //---------------------
    CMemoryPage* m_pEmptyPageList;      // Single Linked List
    CMemoryPage** m_tpPartialPageList;  // Double Linked List, each element of
                                        // the array is a double linked list of
                                        // partial memory page.
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

inline
void* CMetaMemoryPage::operator new (size_t size)
{
    ASSERT(size > 0);
    if (size <= 0) return NULL;

    void* pMetaPage = malloc(size);
    ASSERT(pMetaPage != NULL);

    return pMetaPage;
}

inline
void CMetaMemoryPage::operator delete (void* ptr)
{
    if (ptr != NULL)
    {
        free(ptr);
    }
}

NAMESPACE_END(Mx)

#endif // #ifndef __CMETAMEMORYPAGE_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMetaMemoryPage.cpp
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "Memory/CMemoryPage.h"
#include "Memory/CMetaMemoryPage.h"
#include "Memory/MemoryAllocator.h"

extern "C"
{
}

NAMESPACE_START(Mx)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CMetaMemoryPage::CMetaMemoryPage(IN size_t nPageSize, IN DWORD dwNumberOfPageInMetaPage)
: m_pMetaPageBegin(NULL),
  m_pMetaPageEnd(NULL),
  m_nPageSize(nPageSize),
  m_pEmptyPageList(NULL)
{
    // WARNING: The page size must respect the memory alignment constraint!
    ASSERT(nPageSize == AlignUp(nPageSize, cMEM_ALIGNMENT));
    ASSERT(dwNumberOfPageInMetaPage > 0);

    m_nPageFreeSpace = m_nPageSize - AlignUp(sizeof(CMemoryPage), cMEM_ALIGNMENT);

    // Allocate pages of memory
    //-------------------------
    AllocatePages(m_nPageSize, dwNumberOfPageInMetaPage);

    // Initialize the partial page list. This list is empty at the beginning.
    //
    // NOTE: This array must be as large as the number of different block
    // sizes that may be allocated inside the free space of a page + 1
    //-----------------------------------------------------------------
    int nTableSize = sizeof(CMemoryPage*) * ((m_nPageFreeSpace / cMEM_ALIGNMENT) + 1);
    m_tpPartialPageList = static_cast<CMemoryPage**>(malloc(nTableSize));
    ASSERT(m_tpPartialPageList != NULL);
    memset(m_tpPartialPageList, NULL, nTableSize);
}

CMetaMemoryPage::~CMetaMemoryPage()
{
    if (m_pMetaPageBegin != NULL)
    {
        free(m_pMetaPageBegin);
    }
```

```
    if (m_tpPartialPageList != NULL)
    {
        free(m_tpPartialPageList);
    }
}

void CMetaMemoryPage::AllocatePages(IN size_t nPageSize, IN DWORD dwNumberOfPages)
{
    ASSERT(dwNumberOfPages > 0);

    CMemoryPage* pCurrentPage = NULL;
    CMemoryPage* pPageList = NULL;

    // Allocate memory from the Heap
    //------------------------------
    m_pMetaPageBegin = malloc(dwNumberOfPages * nPageSize);
    ASSERT(m_pMetaPageBegin != NULL);

    // Align the first page. Other pages will be aligned because of there size
    // that must respect the memory alignment constraint.
    //-------------------------------------------------------------------------
    pCurrentPage =
reinterpret_cast<CMemoryPage*>(AlignUp(reinterpret_cast<ptrdiff_t>(m_pMetaPageBegin),
nPageSize));
    if (pCurrentPage != m_pMetaPageBegin)
    {
        --dwNumberOfPages;
        ASSERT(dwNumberOfPages > 0);
    }

    m_pEmptyPageList = pCurrentPage;

    // Create the linked list of empty pages
    //--------------------------------------
    for (DWORD i = 0; i < (dwNumberOfPages - 1); ++i)
    {
        pCurrentPage-
>SetNext(reinterpret_cast<CMemoryPage*>(reinterpret_cast<ptrdiff_t>(pCurrentPage) + nPageSize));
        pCurrentPage = pCurrentPage->GetNext();
    }
    pCurrentPage->SetNext(0);

    m_pMetaPageEnd = reinterpret_cast<void*>(reinterpret_cast<ptrdiff_t>(pCurrentPage) +
nPageSize);
}

void* CMetaMemoryPage::AllocateBlock(IN size_t nBlockSize)
{
    ASSERT(nBlockSize > 0);

    void* pBlock = NULL;

    // Align the block size
    nBlockSize = AlignUp(nBlockSize, cMEM_ALIGNMENT);

    if (nBlockSize <= m_nPageFreeSpace)
    {
        CMemoryPage* pPage;

        UINT nIndex = nBlockSize / cMEM_ALIGNMENT;
        ASSERT(nIndex < sizeof(CMemoryPage*) * ((m_nPageFreeSpace / cMEM_ALIGNMENT) + 1));

        // Take a partial page
        //--------------------
        if ((pPage = m_tpPartialPageList[nIndex]) == NULL)
        {
            // If no partial page is available,
            // the method initialize an empty page.
            //
            // NOTE: If no more empty page are available,
```

```
            //       the method fail and return NULL.
            //-----------------------------------------
            if (m_pEmptyPageList != NULL)
            {
                // Unlink from the empty page list
                pPage = m_pEmptyPageList;
                m_pEmptyPageList = pPage->GetNext();

                // Initializing the page
                new (pPage) CMemoryPage(m_nPageSize, nBlockSize);

                // Allocate the block from the page
                if (pPage->AllocateBlock(pBlock) == CMemoryPage::ePartial)
                {
                    // Link to the partial page list
                    //
                    // OPTIMIZATION: This is the only page in the table index.
                    //               So we can just set the list pointer to the
                    //               new page.
                    //-------------------------------------------------------------
                    m_tpPartialPageList[nIndex] = pPage;
                }

                // NOTE: In case the page is full with only one block
                // allocated (the first one), we just don't link it to the
                // partial page. It will return to the empty page list as soon
                // as the block is released.
                //-------------------------------------------------------------------
            }
        }
        // Allocate the block from the page
        else if (pPage->AllocateBlock(pBlock) == CMemoryPage::eFull)
        {
            // Unlink from the partial page list
            //
            // OPTIMIZATION: Because this page is the first page in the list
            //               we can just make the m_tpPartialPageList point to
            //               the next page.
            //-------------------------------------------------------------------
            m_tpPartialPageList[nIndex] = pPage->GetNext();
        }
        else
        {
            ASSERT(pPage->GetState() == CMemoryPage::ePartial);
        }
    }

    return pBlock;
}

bool CMetaMemoryPage::ReleaseBlock(IN void* ptr)
{
    bool bReleased = true;

    if (ptr >= m_pMetaPageBegin && ptr < m_pMetaPageEnd)
    {
        // Get the page pointer
        CMemoryPage* pPage =
reinterpret_cast<CMemoryPage*>(AlignDown(reinterpret_cast<ptrdiff_t>(ptr), m_nPageSize));

        if (pPage->IsFull())
        {
            // Release memory block to page
            switch (pPage->ReleaseBlock(ptr))
            {
            case CMemoryPage::eEmpty:
                // The page was not in the partial page list. For this reason
                // we can just add it to the empty page list.
                pPage->SetNext(m_pEmptyPageList);
                m_pEmptyPageList = pPage;
                break;
```

```
            case CMemoryPage::ePartial:
                {
                    // Add the page to the partial page list. Add it at the
                    // beginning of the list.
                    //
                    // OPTIMIZATION: The Prev pointer is not set because it
                    //               should not be used when the page is the
                    //               first in the list.
                    //-------------------------------------------------------
                    int nIndex = pPage->GetBlockSize() / cMEM_ALIGNMENT;
                    pPage->SetNext(m_tpPartialPageList[nIndex]);
                    if (m_tpPartialPageList[nIndex] != NULL)
                    {
                        // Now this page is no longer the first page in
                        // the list. We must now set its previous pointer
                        //-----------------------------------------------
                        m_tpPartialPageList[nIndex]->SetPrev(pPage);
                    }
                    m_tpPartialPageList[nIndex] = pPage;
                }
                break;
            default:
                ASSERT(0);
            }
        }
        // This was a partial page
        else if (pPage->ReleaseBlock(ptr) == CMemoryPage::eEmpty)
        {
            // Now that we know that the page is empty, we must
            // remove the page from the partial page list
            //-------------------------------------------------
            int nIndex = pPage->GetBlockSize() / cMEM_ALIGNMENT;

            // Remove from the head of the list
            if (m_tpPartialPageList[nIndex] == pPage)
            {
                m_tpPartialPageList[nIndex] = pPage->GetNext();
            }
            // Remove from the end of the list
            else if (pPage->GetNext() == NULL)
            {
                ASSERT(pPage->GetPrev() != NULL);
                pPage->GetPrev()->SetNext(NULL);
            }
            // Remove from the middle of the list
            else
            {
                ASSERT(pPage->GetPrev() != NULL);
                ASSERT(pPage->GetNext() != NULL);
                pPage->GetPrev()->SetNext(pPage->GetNext());
                pPage->GetNext()->SetPrev(pPage->GetPrev());
            }

            // Add the page to the empty page list
            pPage->SetNext(m_pEmptyPageList);
            m_pEmptyPageList = pPage;
        }
    }
    else
    {
        bReleased = false;
    }

    return bReleased;
}

NAMESPACE_END(Mx)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CMemoryPage.h
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CMEMORYPAGE_H__
#define __CMEMORYPAGE_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member

// Interface Realized & Parent

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Mx)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
const DWORD MEMORY_PAGE_MAGIC_NUMBER = 0xAEAFAEAF;

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CMemoryPage
//
//  WARNINGS:
//       Always use the placement new for construction.
//       Never allocated this object on the stack
//
//==EDOC========================================================================
class CMemoryPage : public CObject
{
// Enum Declaration
//------------------
public:
    enum EState
    {
        eEmpty,
        ePartial,
        eFull
    };

// Published Interface
//---------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CMemoryPage(IN size_t nPageSize, IN size_t nBlockSize);
    ~CMemoryPage() {};

    // << Placement new >>
```

```
    //--------------------
    void* operator new(size_t sise, void*);
    void operator delete(void*, void*);
    void operator delete(void*);

    // << Allocators >>
    //-----------------
    EState AllocateBlock(OUT void*& rpBlock);
    EState ReleaseBlock(IN void* pBlock);

    // << Accessors >>
    //----------------
    size_t GetBlockSize() const;
    WORD GetMaxBlockCount() const;
    WORD GetAllocatedBlockCount() const;
    WORD GetFreeBlockCount() const;

    EState GetState() const;
    bool IsFull() const;
    bool IsPartial() const;
    bool IsEmpty() const;

    // << List manipulators >>
    //------------------------
    CMemoryPage* GetNext() const;
    CMemoryPage* GetPrev() const;
    void SetNext(CMemoryPage* pPage);
    void SetPrev(CMemoryPage* pPage);

// Hidden Methods
//---------------
private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CMemoryPage(const CMemoryPage& rhs);
    CMemoryPage& operator=(const CMemoryPage& rhs);
    void* operator new (size_t size);
    void* operator new [] (size_t size);
    void  operator delete [] (void* ptr);

// Hidden Data Members
//--------------------
private:
    // Double Linked List
    //-------------------
    CMemoryPage* m_pNext;
    CMemoryPage* m_pPrev;

    // Page Data
    //-----------
    size_t m_nBlockSize;
    WORD  m_wBlockCount;
    WORD  m_wMaxBlockCount;

    // Block Data
    //------------
    void* m_pFreeBlockList;
    void* m_pNextUnallocatedBlock;
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

inline
CMemoryPage* CMemoryPage::GetNext() const
{
    return m_pNext;
}

inline
```

```
CMemoryPage* CMemoryPage::GetPrev() const
{
    return m_pPrev;
}

inline
void CMemoryPage::SetNext(CMemoryPage* pPage)
{
    m_pNext = pPage;
}

inline
void CMemoryPage::SetPrev(CMemoryPage* pPage)
{
    m_pPrev = pPage;
}

inline
void* CMemoryPage::operator new(size_t sise, void* ptr)
{
#ifdef MX_DEBUG
    *reinterpret_cast<DWORD*>(reinterpret_cast<ptrdiff_t>(ptr) +
                              sizeof(CMemoryPage) -
                              sizeof(void*)) = MEMORY_PAGE_MAGIC_NUMBER;
#endif
    return ptr;
}

inline
void CMemoryPage::operator delete(void*, void*)
{
    // Placement delete, do nothing!
}

inline
void CMemoryPage::operator delete(void*)
{
    // Placement delete, do nothing!
}

inline
size_t CMemoryPage::GetBlockSize() const
{
    return m_nBlockSize;
}

inline
WORD CMemoryPage::GetMaxBlockCount() const
{
    return m_wMaxBlockCount;
}

inline
WORD CMemoryPage::GetAllocatedBlockCount() const
{
    return m_wBlockCount;
}

inline
WORD CMemoryPage::GetFreeBlockCount() const
{
    return (m_wMaxBlockCount - m_wBlockCount);
}

inline
bool CMemoryPage::IsFull() const
{
    return (m_wBlockCount == m_wMaxBlockCount);
}

inline
```

```
bool CMemoryPage::IsPartial() const
{
    return (m_wBlockCount > 0 && m_wBlockCount < m_wMaxBlockCount);
}

inline
bool CMemoryPage::IsEmpty() const
{
    return (m_wBlockCount == 0);
}

NAMESPACE_END(Mx)

#endif // #ifndef __CMEMORYPAGE_H__
```

```
//==SDOC=======================================================================
//=============================================================================
//
//  File: CMemoryPage.cpp
//
//  Package: Memory
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//=============================================================================
//==EDOC=======================================================================
#include "Std/StdIncludes.h"

//=============================================================================
//====  INCLUDE FILES  ========================================================
//=============================================================================
#include "Memory/MemoryAllocator.h"
#include "Memory/CMemoryPage.h"

extern "C"
{
}

NAMESPACE_START(Mx)

//=============================================================================
//====  STATIC MEMBERS INITIALIZATION  ========================================
//=============================================================================

//=============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ===============================================
//=============================================================================

CMemoryPage::CMemoryPage(IN size_t nPageSize, IN size_t nBlockSize)
: m_pNext(NULL),
  m_pPrev(NULL),
  m_nBlockSize(nBlockSize),
  m_wBlockCount(0),
  m_pFreeBlockList(NULL)
{
    ASSERT(nBlockSize >= sizeof(ptrdiff_t));
    ASSERT(nPageSize > sizeof(CMemoryPage));

    // WARNING: The block size must respect the memory alignment constraint!
    ASSERT(nBlockSize == AlignUp(nBlockSize, cMEM_ALIGNMENT));

    ptrdiff_t nLowerBound = AlignUp(reinterpret_cast<ptrdiff_t>(this) + sizeof(CMemoryPage),
cMEM_ALIGNMENT);
    ptrdiff_t nUpperBound = reinterpret_cast<ptrdiff_t>(this) + nPageSize;

    m_wMaxBlockCount = static_cast<WORD>((nUpperBound - nLowerBound) / nBlockSize);

    // If you hit this assert, it's because you've try to allocate a memory page on the stack.
    // WARNING: You must always use the placement new for CMemoryPage instance.
    ASSERT(*reinterpret_cast<DWORD*>(&m_pNextUnallocatedBlock) == MEMORY_PAGE_MAGIC_NUMBER);

    m_pNextUnallocatedBlock = reinterpret_cast<void*>(nLowerBound);
}

//=============================================================================
//====  PUBLIC FUNCTIONS  =====================================================
//=============================================================================

CMemoryPage::EState CMemoryPage::AllocateBlock(OUT void*& rpBlock)
{
    ASSERT(!IsFull());
```

```
    // Allocate first from the linked list of already used blocks
    //-----------------------------------------------------------
    if (m_pFreeBlockList)
    {
        rpBlock = m_pFreeBlockList;
        m_pFreeBlockList = *(reinterpret_cast<ptrdiff_t**>(m_pFreeBlockList));
    }

    // If the linked list is empty, allocate a new block from the free space
    //-----------------------------------------------------------------------
    else
    {
        ASSERT(m_pNextUnallocatedBlock != NULL);

        rpBlock = m_pNextUnallocatedBlock;
        m_pNextUnallocatedBlock =
reinterpret_cast<void*>(reinterpret_cast<ptrdiff_t>(m_pNextUnallocatedBlock) + m_nBlockSize);
    }

    ++m_wBlockCount;

    // Check if the page is full and return the corresponding state
    //-------------------------------------------------------------
    if (IsFull())
    {
        m_pNextUnallocatedBlock = NULL; // All blocks were allocated at least once
        return eFull;
    }
    else
    {
        ASSERT(IsPartial());
        return ePartial;
    }
}

CMemoryPage::EState CMemoryPage::ReleaseBlock(IN void* pBlock)
{
    ASSERT(!IsEmpty());
    ASSERT(reinterpret_cast<ptrdiff_t>(pBlock) >=
reinterpret_cast<ptrdiff_t>(&(m_pNextUnallocatedBlock)) + sizeof(ptrdiff_t) &&
           reinterpret_cast<ptrdiff_t>(pBlock) <=
reinterpret_cast<ptrdiff_t>(&(m_pNextUnallocatedBlock)  + sizeof(ptrdiff_t) + m_wMaxBlockCount *
m_nBlockSize));

    // Add the block in the linked list of already used blocks
    //--------------------------------------------------------
    *reinterpret_cast<ptrdiff_t*>(pBlock) = reinterpret_cast<ptrdiff_t>(m_pFreeBlockList);
    m_pFreeBlockList = pBlock;

    --m_wBlockCount;

    // Check if the page is empty and return the corresponding state
    //--------------------------------------------------------------
    if (IsEmpty())
    {
        return eEmpty;
    }
    else
    {
        ASSERT(IsPartial());
        return ePartial;
    }
}

CMemoryPage::EState CMemoryPage::GetState() const
{
    EState state;

    if (IsEmpty())
    {
        state = eEmpty;
```

```
    }
    else if (IsFull())
    {
        state = eFull;
    }
    else
    {
        ASSERT(IsPartial());
        state = ePartial;
    }

    return state;
}

NAMESPACE_END(Mx)
```

Annexe 3 : classes Proxy et Adaptor du répondeur téléphonique

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnalogLinePstnCtrlProxy.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CANALOGLINEPSTNCTRLPROXY_H__
#define __CANALOGLINEPSTNCTRLPROXY_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

#include "Adt/CError.h"

// Data Member

// Interface Realized & Parent
#include "Does/CObjectProxy.h"
#include "AnalogLine/IPstnCtrl.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
// {70161A74-A71B-11d5-923A-00A0CC7993F9}
DEFINE_CONST_GUID(GUID_CANALOGLINEPSTNCTRLPROXY, \
0x70161a74, 0xa71b, 0x11d5, 0x92, 0x3a, 0x0, 0xa0, 0xcc, 0x79, 0x93, 0xf9);

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CAnalogLinePstnCtrlProxy
//
//==EDOC========================================================================
class CAnalogLinePstnCtrlProxy : public CObjectProxy, public IPstnCtrl
{
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CAnalogLinePstnCtrlProxy(IN CSocketAddr& rProcessAddr);
    virtual ~CAnalogLinePstnCtrlProxy();

    // IPstnCtrl interface
    //--------------------
    virtual void PlayToneA(IN UINT EndpointNumber, IN EndPointTone tone);
    virtual void StopToneA(IN UINT EndpointNumber);
```

```
    virtual void TakeLineA(IN UINT EndpointNumber);
    virtual void ReleaseLineA(IN UINT EndpointNumber);
    virtual void DialDtmfStringA(IN UINT EndpointNumber,
                                 IN const char* pDTMF_or_MFstring,
                                 IN bool bMFRequested = false);
    virtual void StopDialDtmfStringA(IN UINT EndpointNumber);
    virtual void GenerateFlashA(IN UINT EndpointNumber);

// Hidden Methods
//----------------
private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CAnalogLinePstnCtrlProxy(const CAnalogLinePstnCtrlProxy& rhs);
    CAnalogLinePstnCtrlProxy& operator=(const CAnalogLinePstnCtrlProxy& rhs);

// Hidden Data Members
//--------------------
private:

    enum EOpID
    {
        opCAnalogLine,
        opPlayToneA,
        opStopToneA,
        opTakeLineA,
        opReleaseLineA,
        opDialDtmfStringA,
        opStopDialDtmfStringA,
        opGenerateFlashA,
    };
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CANALOGLINEPSTNCTRLPROXY_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnalogLinePstnCtrlProxy.cpp
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"


//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "AnalogLine/CAnalogLinePstnCtrlProxy.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

CAnalogLinePstnCtrlProxy::CAnalogLinePstnCtrlProxy(IN CSocketAddr& rProcessAddr)
: CObjectProxy(GUID_CANALOGLINEPSTNCTRLPROXY, &rProcessAddr)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID));

    *pInParams << opCAnalogLine;

    err = Create(pInParams, NULL);
    ASSERT(!err());
}

CAnalogLinePstnCtrlProxy::~CAnalogLinePstnCtrlProxy()
{
    CError err;
    err = Destroy();
    ASSERT(!err());
}

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

void CAnalogLinePstnCtrlProxy::PlayToneA(IN UINT EndpointNumber,
                                         IN EndPointTone tone)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(tone));

    *pInParams << opPlayToneA
               << EndpointNumber
               << tone;
```

```
    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::StopToneA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opStopToneA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::TakeLineA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opTakeLineA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::ReleaseLineA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opReleaseLineA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::DialDtmfStringA(IN UINT EndpointNumber,
                                               IN const char* pDTMF_or_MFstring,
                                               IN bool bMFRequested)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         strlen(pDTMF_or_MFstring) + 1 +
                                                         sizeof(bMFRequested));

    *pInParams << opDialDtmfStringA
               << EndpointNumber;
    pInParams->Store(pDTMF_or_MFstring, strlen(pDTMF_or_MFstring));
    *pInParams << bMFRequested;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::StopDialDtmfStringA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opStopDialDtmfStringA
               << EndpointNumber;
```

```
    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnalogLinePstnCtrlProxy::GenerateFlashA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opGenerateFlashA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnalogLinePstnCtrlAdapter.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CANALOGLINEPSTNCTRLADAPTER_H__
#define __CANALOGLINEPSTNCTRLADAPTER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member

// Interface Realized & Parent
#include "Does/CObjectAdapter.h"
#include "AnalogLine/IPstnCtrl.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
// {70161A74-A71B-11d5-923A-00A0CC7993F9}
DEFINE_CONST_GUID(GUID_CANALOGLINEPSTNCTRLADAPTER, \
0x70161a74, 0xa71b, 0x11d5, 0x92, 0x3a, 0x0, 0xa0, 0xcc, 0x79, 0x93, 0xf9);

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CAnalogLinePstnCtrlAdapter
//
//==EDOC========================================================================
class CAnalogLinePstnCtrlAdapter : public CObjectAdapter
{
// Published Interface
//--------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CAnalogLinePstnCtrlAdapter() : CObjectAdapter(GUID_CANALOGLINEPSTNCTRLADAPTER) {};
    virtual ~CAnalogLinePstnCtrlAdapter() {};

// Adaptor Methods Overriding
//---------------------------
protected:
    virtual CError Create(IN OperationID opid,
                          IN COutprocMarshaler* pInParams,
                          OUT COutprocMarshaler** ppOutParams,
                          OUT void** ppObject);
    virtual CError Call(IN void* pObject,
                        IN OperationID opid,
                        IN COutprocMarshaler* pInParams,
                        OUT COutprocMarshaler** ppOutParams);
```

```
    virtual CError Destroy(IN void* pObject);

// Hidden Methods
//---------------
private:
    // Deactivation of the copy constructor and the assignment operator
    //-----------------------------------------------------------------
    CAnalogLinePstnCtrlAdapter(IN const CAnalogLinePstnCtrlAdapter& rhs);
    CAnalogLinePstnCtrlAdapter& operator=(IN const CAnalogLinePstnCtrlAdapter& rhs);

// Hidden Data Members
//--------------------
private:
    enum EOpID
    {
        opCAnalogLine,
        opPlayToneA,
        opStopToneA,
        opTakeLineA,
        opReleaseLineA,
        opDialDtmfStringA,
        opStopDialDtmfStringA,
        opGenerateFlashA,
    };
};

//============================================================================
//====  INLINE FUNCTIONS  ====================================================
//============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CANALOGLINEPSTNCTRLADAPTER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnalogLinePstnCtrlAdapter.cpp
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "AnalogLine/CAnalogLinePstnCtrlAdapter.h"
#include "AnalogLine/CAnalogLine.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

CError CAnalogLinePstnCtrlAdapter::Create(IN OperationID opid,
                                          IN COutprocMarshaler* pInParams,
                                          OUT COutprocMarshaler** ppOutParams,
                                          OUT void** ppObject)
{
    ASSERT(opid == opCAnalogLine);

    CError err;

    if (opid == opCAnalogLine)
    {
        *ppObject = new CAnalogLine();
    }
    else
    {
        err.SetLevel(CError::eERROR);
    }

    return err;
}

CError CAnalogLinePstnCtrlAdapter::Call(IN void* pObject,
                                        IN OperationID opid,
                                        IN COutprocMarshaler* pInParams,
                                        OUT COutprocMarshaler** ppOutParams)
{
    ASSERT(pObject != NULL);

    CError err;
```

```
    IPstnCtrl* pPstnCtrl = reinterpret_cast<IPstnCtrl*>(pObject);
    *ppOutParams = NULL;

    UINT unEndpointNum;
    *pInParams >> unEndpointNum;

    switch (opid)
    {
    case opPlayToneA:
        {
            EndPointTone tone;
            *pInParams >> reinterpret_cast<int&>(tone);
            pPstnCtrl->PlayToneA(unEndpointNum, tone);
        }
        break;
    case opStopToneA:
        pPstnCtrl->StopToneA(unEndpointNum);
        break;
    case opTakeLineA:
        pPstnCtrl->TakeLineA(unEndpointNum);
        break;
    case opReleaseLineA:
        pPstnCtrl->ReleaseLineA(unEndpointNum);
        break;
    case opDialDtmfStringA:
        {
            char* pDtmf = new char[128];
            UINT16 unDtmfSize = 0;
            bool bMFRequested;
            pInParams->Load(pDtmf, 128, unDtmfSize);
            *pInParams >> bMFRequested;
            pPstnCtrl->DialDtmfStringA(unEndpointNum, pDtmf, bMFRequested);
            delete pDtmf;
        }
        break;
    case opStopDialDtmfStringA:
        pPstnCtrl->StopDialDtmfStringA(unEndpointNum);
        break;
    case opGenerateFlashA:
        pPstnCtrl->GenerateFlashA(unEndpointNum);
        break;
    default:
        err.SetLevel(CError::eERROR);
        ASSERT(0);
    }

    return err;
}

CError CAnalogLinePstnCtrlAdapter::Destroy(IN void* pObject)
{
    ASSERT(pObject != NULL);

    CError err;

    CAnalogLine* pAnalogLine = reinterpret_cast<CAnalogLine*>(pObject);
    delete pAnalogLine;

    return err;
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnswerMachineProxy.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CANSWERMACHINEPROXY_H__
#define __CANSWERMACHINEPROXY_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

#include "Adt/CError.h"

// Data Member

// Interface Realized & Parent
#include "Does/CObjectProxy.h"
#include "AnalogLine/IPstnMgr.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
// {70161A75-A71B-11d5-923A-00A0CC7993F9}
DEFINE_GUID(GUID_CANSWERMACHINEPROXY, \
0x70161a75, 0xa71b, 0x11d5, 0x92, 0x3a, 0x0, 0xa0, 0xcc, 0x79, 0x93, 0xf9);

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CAnswerMachineProxy
//
//==EDOC========================================================================
class CAnswerMachineProxy : public CObjectProxy, public IPstnMgr
{
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CAnswerMachineProxy(IN CSocketAddr& rProcessAddr);
    virtual ~CAnswerMachineProxy();

    // IPstnMgr interface
    //--------------------
    virtual void DtmfDetectA(IN UINT EndpointNumber, IN char Digit);
    virtual void DtmfTerminatedA(IN UINT EndpointNumber, IN char Digit);
    virtual void RingOnDetectA(IN UINT EndpointNumber);
    virtual void RingOnDetectA(IN UINT EndpointNumber,
                               IN UINT uTimeToWaitBeforeTakeLine_ms);
    virtual void RingOffDetectA(IN UINT EndpointNumber);
    virtual void CallerIdDetectedA(IN UINT EndpointNumber,
                                   IN const CCallerIdInfo& rCallId);
    virtual void FaxDetectedA(IN UINT EndpointNumber,
```

```
                                IN const CFaxSwitchoverAction& rFaxSwitchoverAction);
    virtual void T38FaxTerminatedA(IN UINT EndpointNumber);
    virtual void DialDtmfStringTerminatedA(IN UINT EndpointNumber);
    virtual void LineTakenA(IN UINT EndpointNumber);
    virtual void LineTakenA(IN UINT EndpointNumber,
                            IN UINT uTimeToWaitBeforeDialing_ms);
    virtual void LineTakenFailedA(IN UINT EndpointNumber,
                                  IN LINETAKENFAIL_MSG reason);
    virtual void RemotePartyHangupA(IN UINT EndpointNumber);
    virtual void FlashTerminatedA(IN UINT EndpointNumber);
    virtual void SilencePeriodDetectedA(IN UINT EndpointNumber,
                                        IN UINT32 uTimeSinceSilencePeriodBegin_ms);
    virtual void SilencePeriodDetectedA(IN UINT EndpointNumber,
                                        IN UINT32 uTimeSinceSilencePeriodBeginLocal_ms,
                                        IN UINT32 uTimeSinceSilencePeriodBeginNetwork_ms);

// Hidden Methods
//----------------
private:
    // Deactivated Constructors / Destructors / Operators
    //---------------------------------------------------
    CAnswerMachineProxy(const CAnswerMachineProxy& rhs);
    CAnswerMachineProxy& operator=(const CAnswerMachineProxy& rhs);

// Hidden Data Members
//--------------------
private:

    enum EOpID
    {
        opCAnswerMachineProxy,
        opDtmfDetectA,
        opDtmfTerminatedA,
        opRingOnDetectA0,
        opRingOnDetectA1,
        opRingOffDetectA,
        opCallerIdDetectedA,
        opFaxDetectedA,
        opT38FaxTerminatedA,
        opDialDtmfStringTerminatedA,
        opLineTakenA0,
        opLineTakenA1,
        opLineTakenFailedA,
        opRemotePartyHangupA,
        opFlashTerminatedA,
        opSilencePeriodDetectedA0,
        opSilencePeriodDetectedA1
    };
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CANSWERMACHINEPROXY_H__
```

```
//==SDOC=========================================================================
//===============================================================================
//
//  File: CAnswerMachineProxy.cpp
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//===============================================================================
//==EDOC=========================================================================
#include "Std/StdIncludes.h"

//===============================================================================
//====  INCLUDE FILES  ==========================================================
//===============================================================================
#include "AnalogLine/CAnswerMachineProxy.h"

extern "C"
{
}

NAMESPACE_START(Does)

//===============================================================================
//====  STATIC MEMBERS INITIALIZATION  ==========================================
//===============================================================================

//===============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  =================================================
//===============================================================================

CAnswerMachineProxy::CAnswerMachineProxy(IN CSocketAddr& rProcessAddr)
: CObjectProxy(GUID_CANSWERMACHINEPROXY, &rProcessAddr)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID));

    *pInParams << opCAnswerMachineProxy;

    err = Create(pInParams, NULL);
    ASSERT(!err());
}

CAnswerMachineProxy::~CAnswerMachineProxy()
{
    CError err;
    err = Destroy();
    ASSERT(!err());
}

//===============================================================================
//====  PUBLIC FUNCTIONS  =======================================================
//===============================================================================

void CAnswerMachineProxy::DtmfDetectA(IN UINT EndpointNumber, IN char Digit)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(Digit));

    *pInParams << opDtmfDetectA
               << EndpointNumber
               << Digit;

    err = Call(pInParams, NULL);
```

```
    ASSERT(!err());
}

void CAnswerMachineProxy::DtmfTerminatedA(IN UINT EndpointNumber, IN char Digit)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(Digit));

    *pInParams << opDtmfTerminatedA
               << EndpointNumber
               << Digit;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::RingOnDetectA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opRingOnDetectA0
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());

}

void CAnswerMachineProxy::RingOnDetectA(IN UINT EndpointNumber,
                                        IN UINT uTimeToWaitBeforeTakeLine_ms)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(uTimeToWaitBeforeTakeLine_ms));

    *pInParams << opRingOnDetectA1
               << EndpointNumber
               << uTimeToWaitBeforeTakeLine_ms;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::RingOffDetectA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opRingOffDetectA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::CallerIdDetectedA(IN UINT EndpointNumber,
                                            IN const CCallerIdInfo& rCallId)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(rCallId));

    *pInParams << opCallerIdDetectedA
               << EndpointNumber
```

```
                << rCallId;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::FaxDetectedA(IN UINT EndpointNumber,
                                       IN const CFaxSwitchoverAction& rFaxSwitchoverAction)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(rFaxSwitchoverAction));

    *pInParams << opFaxDetectedA
               << EndpointNumber
               << rFaxSwitchoverAction;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::T38FaxTerminatedA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opT38FaxTerminatedA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::DialDtmfStringTerminatedA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opDialDtmfStringTerminatedA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::LineTakenA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opLineTakenA0
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::LineTakenA(IN UINT EndpointNumber,
                                     IN UINT uTimeToWaitBeforeDialing_ms)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(uTimeToWaitBeforeDialing_ms));

    *pInParams << opLineTakenA1
               << EndpointNumber
```

```
                << uTimeToWaitBeforeDialing_ms;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::LineTakenFailedA(IN UINT EndpointNumber,
                                           IN LINETAKENFAIL_MSG reason)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +
                                                         sizeof(reason));

    *pInParams << opLineTakenFailedA
               << EndpointNumber
               << reason;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::RemotePartyHangupA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << opRemotePartyHangupA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::FlashTerminatedA(IN UINT EndpointNumber)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber));

    *pInParams << FlashTerminatedA
               << EndpointNumber;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::SilencePeriodDetectedA(IN UINT EndpointNumber,
                                                 IN UINT32 uTimeSinceSilencePeriodBegin_ms)
{
    CError err;
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +

sizeof(uTimeSinceSilencePeriodBegin_ms));

    *pInParams << opSilencePeriodDetectedA0
               << EndpointNumber
               << uTimeSinceSilencePeriodBegin_ms;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

void CAnswerMachineProxy::SilencePeriodDetectedA(IN UINT EndpointNumber,
                                                 IN UINT32 uTimeSinceSilencePeriodBeginLocal_ms,
                                                 IN UINT32
uTimeSinceSilencePeriodBeginNetwork_ms)
{
    CError err;
```

```
    COutprocMarshaler* pInParams = new COutprocMarshaler(sizeof(EOpID) +
                                                         sizeof(EndpointNumber) +

sizeof(uTimeSinceSilencePeriodBeginNetwork_ms));

    *pInParams << opSilencePeriodDetectedA1
               << EndpointNumber
               << uTimeSinceSilencePeriodBeginNetwork_ms;

    err = Call(pInParams, NULL);
    ASSERT(!err());
}

NAMESPACE_END(Does)
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnswerMachineAdapter.h
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#ifndef __CANSWERMACHINEADAPTER_H__
#define __CANSWERMACHINEADAPTER_H__

//==============================================================================
//====  INCLUDES + FORWARD DECLARATIONS  =======================================
//==============================================================================

// Data Member

// Interface Realized & Parent
#include "Does/CObjectAdapter.h"
#include "AnalogLine/IPstnMgr.h"

// Forward Declarations Outside of the Namespace

NAMESPACE_START(Does)

// Forward Declarations Inside of the Namespace

//==============================================================================
//====  CONSTANTS + DEFINES  ===================================================
//==============================================================================
// {70161A75-A71B-11d5-923A-00A0CC7993F9}
DEFINE_GUID(GUID_CANSWERMACHINEADAPTER, \
0x70161a75, 0xa71b, 0x11d5, 0x92, 0x3a, 0x0, 0xa0, 0xcc, 0x79, 0x93, 0xf9);

//==============================================================================
//====  NEW TYPE DEFINITIONS  ==================================================
//==============================================================================

//==SDOC========================================================================
//
//  class CAnswerMachineAdapter
//
//==EDOC========================================================================
class CAnswerMachineAdapter : public CObjectAdapter
{
// Published Interface
//--------------------
public:
    // << Constructors / Destructors >>
    //---------------------------------
    CAnswerMachineAdapter() : CObjectAdapter(GUID_CANSWERMACHINEADAPTER) {};
    virtual ~CAnswerMachineAdapter() {};

// Adaptor Methods Overriding
//-------------------------------
protected:
    virtual CError Create(IN OperationID opid,
                          IN COutprocMarshaler* pInParams,
                          OUT COutprocMarshaler** ppOutParams,
                          OUT void** ppObject);
    virtual CError Call(IN void* pObject,
                        IN OperationID opid,
                        IN COutprocMarshaler* pInParams,
                        OUT COutprocMarshaler** ppOutParams);
```

```
    virtual CError Destroy(IN void* pObject);

// Hidden Methods
//---------------
private:
    // Deactivation of the copy constructor and the assignment operator
    //-----------------------------------------------------------------
    CAnswerMachineAdapter(IN const CAnswerMachineAdapter& rhs);
    CAnswerMachineAdapter& operator=(IN const CAnswerMachineAdapter& rhs);

// Hidden Data Members
//--------------------
private:

    enum EOpID
    {
        opCAnswerMachineProxy,
        opDtmfDetectA,
        opDtmfTerminatedA,
        opRingOnDetectA0,
        opRingOnDetectA1,
        opRingOffDetectA,
        opCallerIdDetectedA,
        opFaxDetectedA,
        opT38FaxTerminatedA,
        opDialDtmfStringTerminatedA,
        opLineTakenA0,
        opLineTakenA1,
        opLineTakenFailedA,
        opRemotePartyHangupA,
        opFlashTerminatedA,
        opSilencePeriodDetectedA0,
        opSilencePeriodDetectedA1
    };
};

//=============================================================================
//====  INLINE FUNCTIONS  =====================================================
//=============================================================================

NAMESPACE_END(Does)

#endif // #ifndef __CANSWERMACHINEADAPTER_H__
```

```
//==SDOC========================================================================
//==============================================================================
//
//  File: CAnswerMachineAdapter.cpp
//
//  Package: DOES
//
//  OS: ALL
//  HW: ALL
//  Other Dependencies: -
//
//  Created by: Samuel Guenette
//
//==============================================================================
//==EDOC========================================================================
#include "Std/StdIncludes.h"

//==============================================================================
//====  INCLUDE FILES  =========================================================
//==============================================================================
#include "AnalogLine/CAnswerMachineAdapter.h"
#include "AnalogLine/CAnswerMachine.h"
#include "AnalogLine/CCallerIdInfo.h"
#include "AnalogLine/IPstnMgr.h"

extern "C"
{
}

NAMESPACE_START(Does)

//==============================================================================
//====  STATIC MEMBERS INITIALIZATION  =========================================
//==============================================================================

//==============================================================================
//====  CONSTRUCTOR/DESTRUCTOR  ================================================
//==============================================================================

//==============================================================================
//====  PUBLIC FUNCTIONS  ======================================================
//==============================================================================

CError CAnswerMachineAdapter::Create(IN OperationID opid,
                                     IN COutprocMarshaler* pInParams,
                                     OUT COutprocMarshaler** ppOutParams,
                                     OUT void** ppObject)
{
    ASSERT(opid == opCAnswerMachineProxy);

    CError err;

    if (opid == opCAnswerMachineProxy)
    {
        *ppObject = new CAnswerMachine();
    }
    else
    {
        err.SetLevel(CError::eERROR);
    }

    return err;
}

CError CAnswerMachineAdapter::Call(IN void* pObject,
                                   IN OperationID opid,
                                   IN COutprocMarshaler* pInParams,
                                   OUT COutprocMarshaler** ppOutParams)
{
    ASSERT(pObject != NULL);
```

```
CError err;

IPstnMgr* pPstnMgr = reinterpret_cast<IPstnMgr*>(pObject);
*ppOutParams = NULL;

UINT unEndpointNum;
*pInParams >> unEndpointNum;

switch (opid)
{
case opDtmfDetectA:
    {
        char digit;
        *pInParams >> digit;
        pPstnMgr->DtmfDetectA(unEndpointNum, digit);
    }
    break;
case opDtmfTerminatedA:
    {
        char digit;
        *pInParams >> digit;
        pPstnMgr->DtmfTerminatedA(unEndpointNum, digit);
    }
    break;
case opRingOnDetectA0:
    pPstnMgr->RingOnDetectA(unEndpointNum);
    break;
case opRingOnDetectA1:
    {
        UINT uTimeToWaitBeforeTakeLine_ms;
        *pInParams >> uTimeToWaitBeforeTakeLine_ms;
        pPstnMgr->RingOnDetectA(unEndpointNum, uTimeToWaitBeforeTakeLine_ms);
    }
    break;
case opRingOffDetectA:
    pPstnMgr->RingOffDetectA(unEndpointNum);
    break;
case opCallerIdDetectedA:
    {
        CCallerIdInfo callId;
        *pInParams >> callId;
        pPstnMgr->CallerIdDetectedA(unEndpointNum, callId);
    }
    break;
case opFaxDetectedA:
    {
        CFaxSwitchoverAction faxSwitchoverAction;
        *pInParams >> faxSwitchoverAction;
        pPstnMgr->FaxDetectedA(unEndpointNum, faxSwitchoverAction);
    }
    break;
case opT38FaxTerminatedA:
    pPstnMgr->T38FaxTerminatedA(unEndpointNum);
    break;
case opDialDtmfStringTerminatedA:
    pPstnMgr->DialDtmfStringTerminatedA(unEndpointNum);
    break;
case opLineTakenA0:
    pPstnMgr->LineTakenA(unEndpointNum);
    break;
case opLineTakenA1:
    {
        UINT uTimeToWaitBeforeDialing_ms;
        *pInParams >> uTimeToWaitBeforeDialing_ms;
        pPstnMgr->LineTakenA(unEndpointNum, uTimeToWaitBeforeDialing_ms);
    }
    break;
case opLineTakenFailedA:
    {
        IPstnMgr::LINETAKENFAIL_MSG reason;
        *pInParams >> reinterpret_cast<int&>(reason);
```

```
            pPstnMgr->LineTakenFailedA(unEndpointNum, reason);
        }
        break;
    case opRemotePartyHangupA:
        pPstnMgr->RemotePartyHangupA(unEndpointNum);
        break;
    case opFlashTerminatedA:
        pPstnMgr->FlashTerminatedA(unEndpointNum);
        break;
    case opSilencePeriodDetectedA0:
        {
            UINT32 uTimeSinceSilencePeriodBegin_ms;
            *pInParams >> uTimeSinceSilencePeriodBegin_ms;
            pPstnMgr->SilencePeriodDetectedA(unEndpointNum,
                                             uTimeSinceSilencePeriodBegin_ms);
        }
        break;
    case opSilencePeriodDetectedA1:
        {
            UINT32 uTimeSinceSilencePeriodBeginLocal_ms;
            UINT32 uTimeSinceSilencePeriodBeginNetwork_ms;
            *pInParams >> uTimeSinceSilencePeriodBeginLocal_ms
                       >> uTimeSinceSilencePeriodBeginNetwork_ms;
            pPstnMgr->SilencePeriodDetectedA(unEndpointNum,
                                             uTimeSinceSilencePeriodBeginLocal_ms,
                                             uTimeSinceSilencePeriodBeginNetwork_ms);
        }
        break;
    default:
        err.SetLevel(CError::eERROR);
        ASSERT(0);
    }

    return err;
}

CError CAnswerMachineAdapter::Destroy(IN void* pObject)
{
    ASSERT(pObject != NULL);

    CError err;

    CAnswerMachine* pAnswerMachine = reinterpret_cast<CAnswerMachine*>(pObject);
    delete pAnswerMachine;

    return err;
}

NAMESPACE_END(Does)
```