CORBA et CorbaScript

J-M. Geib & P. Merle & C. Gransart

Laboratoire d'Informatique Fondamentale de Lille,
USTL, Bât. M3, 59655 Villeneuve d'Ascq Cedex, France
Tél: 03 20 43 47 21, Fax: 03 20 43 65 66
{geib, merle, gransart}@lifl.fr

Résumé

Ce papier décrit les travaux du Laboratoire d'Informatique Fondamentale de Lille sur CorbaScript: un nouveau langage de scripts dédié à Corba. Il permet de manipuler simplement, interactivement et dynamiquement des objets distribués sur une infrastructure à la norme CORBA de l'OMG. Sa vocation concerne essentiellement l'administration, la configuration, le test, le prototypage, etc, c'est-à-dire les tâches de caractère dynamique qui sont le lot de tout utilisateur de telles infrastructures. L'environnement CorbaScript génère dynamiquement les talons des objets rencontrés lors de la navigation dans les serveurs d'objets. Il utilise pour cela les mécanismes dynamiques associés à l'Interface d'Invocation Dynamique (DII), et au dépôt d'interfaces (IR). De plus des interfaces IDL peuvent être dynamiquement implantées par des classes CorbaScript grâce à l'Interface de Squelette Dynamique (DSI). Après un rappel critique du cycle de développement CORBA, nous illustrons l'utilisation de CorbaScript par un petit exemple d'un serveur de grilles et l'utilisation du service de nommage de CORBA.

Mots Clés: CORBA, langage de scripts pour CORBA, CorbaScript.

1. Introduction

Plusieurs grosses compagnies comme Motorola, Boeing ou Alcatel, ont planifié la reconception de leurs systèmes d'information - qui s'étendent à l'échelle mondiale - sur la base d'infrastructures standardisées à objets distribués comme ODP [Lin94], CORBA [OMG96], ou TINA [Con96]. La raison de ces travaux est que ces infrastructures peuvent naturellement supporter de nombreux utilisateurs, répartis sur de nombreuses machines, et interagissant de façon complexe, entre-eux et sur l'ensemble des ressources et informations disponibles. Les infrastructures à objets distribués apportent ici un modèle unifié qui permet de cacher l'hétérogénéité des machines, systèmes, langages, informations et coopérations.

Mais ces systèmes doivent de plus pouvoir s'adapter à de multiples et rapides évolutions, en termes de fonctionnalités, d'environnements informatiques, mais aussi par exemple en fonction du marché et de la législation... Et l'un des gros problèmes liés à cette importante variabilité est la difficulté, pour les ingénieurs en charge du système, de répondre rapidement aux multiples demandes de nouveaux services. En effet dans la vision traditionnelle des environnements actuels, l'utilisateur final n'a accès qu'à des services développés par des ingénieurs spécialistes de l'infrastructure. Cette dichotomie est essentiellement due à la complexité intrinsèque de l'environnement, complexité qui ne permet pas à un utilisateur final de créer lui-même les services nouveaux dont il a besoin (même si ceux-ci sont simples, ou ne font que combiner des services existants).

Pour réduire ce goulot d'étranglement des services spécialisés, il faut donc rendre une part de pouvoir aux utilisateurs clients d'informations, afin que ceux-ci puissent participer au processus d'enrichissement des services offerts par l'infrastructure globale. A titre de comparaison, peut-on encore imaginer de nos jours un système de gestion de bases de données pour lequel chaque requête d'un utilisateur ne pourrait qu'être écrite par un spécialiste des routines du système. Les langages comme SQL [ISO92] or OQL [CK94] permettent dans ce cadre à l'utilisateur final de définir simplement ses propres requêtes sans devoir maîtriser toute la complexité de l'ensemble, mais uniquement en se focalisant sur la logique de son travail.

La même évolution doit intervenir dans les environnements à objets distribués, c'est la raison de l'intérêt de l'OMG pour des langages de scripts pour CORBA. C'est aussi celui du projet GOODE (Generic Object Oriented Dynamic Environment [MGG96a]) du LIFL avec le but de promouvoir l'utilisation des systèmes distribués de grande échelle [MGG97a]. Nous présentons ici CorbaScript [MGG96c, MGG97b], un langage simple et interprété, fonctionnant au dessus des bus CORBA.

La section 2 présente le cycle traditionnel de développement d'un service CORBA. La section 3 discute des bénéfices des langages de scripts. Les fonctionnalités de CorbaScript sont détaillées par l'exemple dans la section 4. La section 5 donne des informations sur l'implantation du langage.

2. L'utilisation traditionnelle de CORBA

2.1. Un rapide tour de CORBA

L'approche de OMG [Sol94] favorise une approche modulaire par objets pour les composants d'applications distribuées. Un bus (ORB: Object Request Broker) permet de déployer et de connecter ces composants. Les interfaces des composants sont spécifiés par un langage de définition d'interfaces (IDL). Les fichiers IDL sont compilés pour générer les talons clients et serveurs qui vont permettre d'utiliser l'ORB pour transporter les requêtes sur le réseau.

L'ORB et les talons générés par compilation forment un mécanisme de RPC orienté objet qui prend en charge les problèmes d'hétérogénéité aux niveaux matériel et logiciel, systèmes et langages. De plus le protocole IIOP (Internet Inter-Orb Protocol) offre l'interopérabilité sur Internet entre bus CORBA.

L'architecture de l'OMG et son bus CORBA rejoignent les besoins des concepteurs de services distribués: transparence des communications entre objets hétérogènes, associée à une ambitieuse liste de services de base et utilitaires [OHE95, GGM97] qui couvrent les classiques problèmes des environnements distribués: nommage, événements, messages, persistance, sécurité, transactions, etc.

2.2. Une petite application

Lorsqu'on implante une application en utilisant CORBA, la première étape consiste en la spécification en IDL de l'interface des objets qui vont interagir. Comme cet exemple est une simple illustration de CORBA et de CorbaScript, nous avons volontairement simplifié le modèle objet des ressources proposées par cette application (*cf.* figure 1).

L'application est composée d'une *fabrique* qui permet la création d'objets *grille*. Une grille est une matrice de valeurs et fournit des opérations pour lire (*get*) et écrire (*set*) les valeurs. Une exception (*InvalidCoord*) est définie pour prendre en charge les indices hors limites. Le module IDL *GridService* donne un espace de nommage pour tous les types IDL et définitions d'interfaces à l'intérieur de cette application et évite les conflits de noms avec d'autres spécifications IDL.

```
module GridService {
  struct Coord { unsigned short x, y; };
  exception InvalidCoord { Coord pos; };
  typedef double Value;

  interface Grid {
    readonly attribute Coord dimension;
    void set (in Coord pos, in Value val) raises (InvalidCoord);
    Value get (in Coord pos) raises (InvalidCoord);
    void destroy ();
  };

  interface Factory {
    Grid create_grid (in Coord dim, in Value init_value);
  };
};
```

Figure 1: Le service de grilles spécifié en IDL

2.3. Le processus de développement et de déploiement

Avec l'interface de service IDL décrite ci-dessus, il n'est pas nécessaire de projeter manuellement les spécifications dans un langage particulier pour permettre l'écriture puis l'installation des clients et serveurs. La figure 2 montre la procédure à suivre:

1. **Compilation des interfaces IDL**: Les interfaces sont décrites en IDL et stockées dans des fichiers texte. Le compilateur IDL prend de tels fichiers en entrée et les analyse. Il charge ensuite ces interfaces dans le dépôt d'interface (IR: Interface Repository). C'est le tout premier travail effectué par tout compilateur IDL.

2. **Projection dans un langage de programmation**: Le compilateur IDL génère ensuite en sortie les talons qui seront utilisés par les clients des types d'objets décrits, ainsi que ceux (squelettes) qui seront utilisés par les programmes serveurs qui les implanteront. Cette projection est spécifique pour chaque langage de programmation. Un environnement CORBA doit fournir un compilateur IDL pour chaque langage cible potentiel.

3. **Implantation des objets serveurs**: En complétant les squelettes générés, le développeur du service implante les objets dans le langage de son choix (C++[Str91], Java [GM95], SmallTalk [GR83], etc). Il doit suivre pour cela les règles de projection pour chaque langage, règles complexes et pas toujours naturelles...

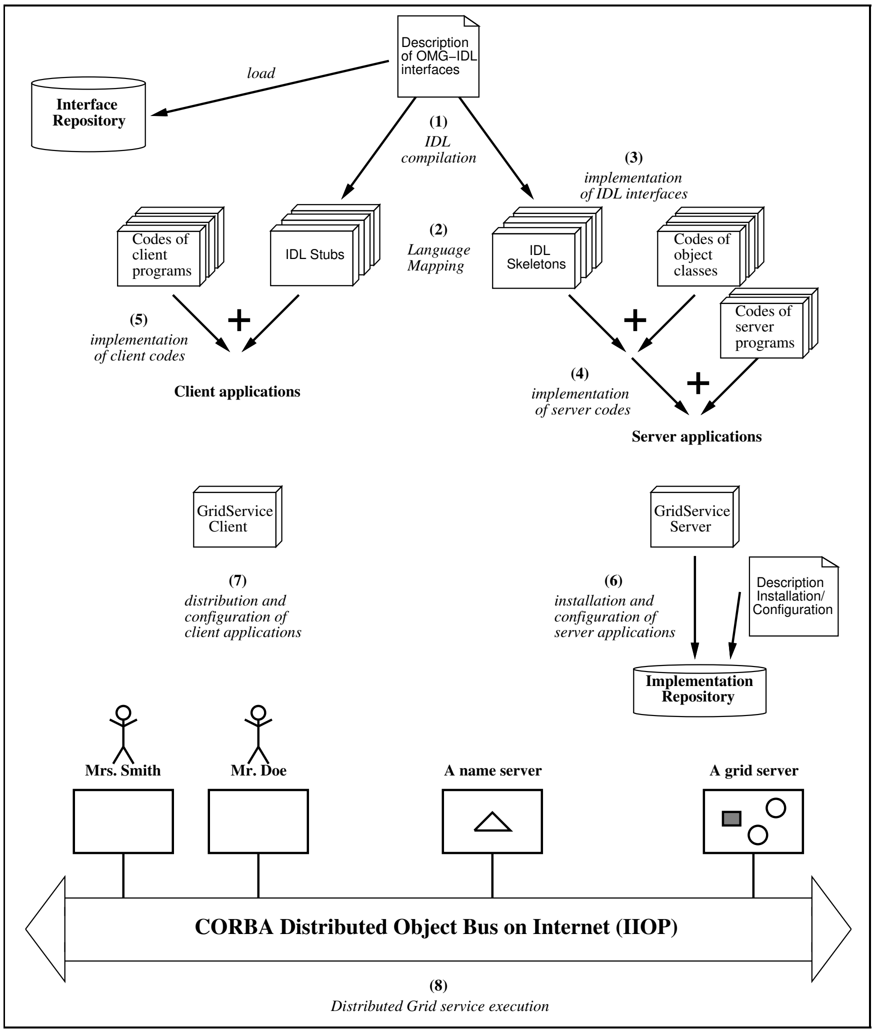

Figure 2: Le développement et le déploiement du service de grilles

4. **Création des serveurs**: Le développeur du service doit écrire le(s) programme(s) serveur(s) incluant l'implantation des objets et leurs squelettes. Dans notre exemple, il n'y a qu'un seul serveur: *GridService*. Il contient le code pour se connecter à l'ORB, créer la *fabrique*, et rendre public sa référence grâce à un service standard de CORBA comme le service de *Nommage* ou de *Trading* .

5. **Implantation des objets clients**: Ici le développeur écrit les programmes clients qui utilisent les objets du serveur. Ces programmes doivent inclure le talon généré, agrémenté de code spécifique comme une interface graphique. Un programme client obtient les références sur les objets serveurs, toujours grâce aux services de *Nommage* ou de *Trading* précédents.

6. **Installation et configuration des serveurs**: Cette phase comprend l'installation du ou des serveurs dans un Dépôt d'Implantations pour permettre leur activation automatique lorsque des requêtes seront envoyées à destination des objets qu'ils renferment.

7. **Distribution et configuration des programmes clients**: Une fois déverminés, les programmes clients peuvent être envoyés vers les sites d'utilisation. A ces endroits, ils doivent être configurés de telle manière de connaître le serveur de grilles.

8. **Exécution du service distribué**: Finalement le service de grilles peut être utilisé. L'ORB va se charger de réaliser les communications et l'activation des serveurs.

Le processus décrit ici est général. Il doit être suivi pour toute application utilisant un langage de programmation supportant un ORB. CORBA permet bien sur de mixer différents langages de programmation, par exemple un serveur en C++ et un client dans une applet Java. L'ensemble reste cependant plutôt complexe et réservé à des programmeurs expérimentés et non aux utilisateurs clients.

2.4. Vers une autre approche

Actuellement la plupart des développements CORBA se fondent sur l'approche "statique" précédente, dans laquelle les interactions entre clients et serveurs sont fixées à la compilation. Les clients ne contiennent donc que les talons pour les types d'objets identifiés dans une phase de spécification initiale, et ce seront les seuls qui pourront être utilisés pour faire des invocations lors de l'exécution.

Nous considérons que cette approche est inappropriée pour supporter des services très évolutifs comme indiqué dans l'introduction. Avec une approche statique, lorsqu'un service évolue, les programmes serveurs doivent être adaptés, et, plus gravement, la multitude de programmes clients dispersés doit être changée. Comme CORBA est une norme complexe (en particulier sur les règles de projection), ces changements demandent l'aide de personnes expertes, ce qui limite fortement l'utilisation de CORBA dans un univers en évolution constante.

Pour répondre à ce problème, il faut faire émerger un ensemble d'outils flexibles favorisant le développement sur ce type d'infrastructures. C'est l'objectif de notre projet GOODE [Mer97]. Nous proposons une approche "dynamique" qui découple les programmes clients et serveurs, en permettant l'adaptation dynamique des clients à l'exécution. Notre approche utilise les mécanismes dynamiques de CORBA, c'est-à-dire l'interface d'invocation dynamique (DII) et le dépôt d'interfaces (IR). Ces mécanismes sont eux-mêmes complexes, nos outils permettent de les introduire sur le poste de l'utilisateur final. Ces expérimentations ont été validées par la conception du langage de scripts CorbaScript.

3. Pourquoi utiliser des scripts?

Avant d'introduire CorbaScript, cette section analyse les bénéfices des scripts dans l'univers CORBA.

3.1. Pourquoi des scripts?

L'utilisation d'un langage de script apporte les bénéfices suivant:

- **Facilité d'utilisation**: Un script est plus conscis et facile à écrire que l'équivalent dans un langage de programmation traditionnel. Le langage de scripts apporte souvent des facilités comme le typage dynamique ou un ramasse-miettes, et il est de plus, souvent pragmatiquement orienté vers l'utilisateur final, même novice, qui veut écrire rapidement les petits programmes dont il a besoin.
- **Meilleure productivité**: La facilité d'utilisation permet une activité de prototypage rapide, en mode interactif, qui facilite l'émergence de solutions difficilement testables en mode statique. Un langage de scripts encourage de plus l'échange de scripts entre utilisateurs, favorisant la diffusion de solutions.

- **Coûts réduits**: La simplicité et la productivité réduisent les coûts de formation et de développement. Elles simplifient les structures conventionnelles de développement de logiciels.

3.2. Bénéfices pour les utilisateurs

Les qualités précédentes augmentent fortement les possibilités d'utilisation des objets distribués CORBA dans toutes les phases du développement et d'installation.

- **Prototypage**: Durant la phase de conception d'un service distribué, on est amené à faire évoluer très rapidement les interfaces IDL et l'implantation des objets. Seule une expérimentation permet de fixer les choix. Prototyper dans un mode statique, en C++ par exemple, est long et coûteux, un langage interprété qui raccourcit le cycle est un avantage important.

- **Test**: La phase de test d'un service CORBA nécessite de développer tout un ensemble de petits programmes clients, validant chacun une des fonctionnalités du service. Ces programmes de test sont ici eux-mêmes difficiles à écrire et à tester. De plus ils deviennent inutiles lorsque la phase de test se termine. La possibilité de faire du test via un interpréteur de commandes fait gagner beaucoup de temps et d'argent. Comme de plus les interfaces sont présentes dans le dépôt d'interfaces, certains tests peuvent être, en mode dynamique, générés quasi automatiquement.

- **Configuration**: Comme pour le test, l'administration et la configuration nécessitent de nombreux petits programmes utilitaires, qui sont souvent dépendants des interfaces IDL utilisées. Un langage de script est une bonne alternative à cela, en offrant la souplesse liée à sa dynamicité.

- **Composition**: Les services présents sur le bus peuvent souvent être combinés pour en fabriquer de plus spécifiques. Avec un langage de scripts un utilisateur peut simplement faire ces combinaisons et autres adaptations de surface, sans passer par le cycle lourd du développement d'un nouveau service. Le langage de scripts est ici vu comme une "colle logicielle" permettant de manipuler des composants. Notons que ces nouveaux composants peuvent eux-mêmes être accessibles comme des objest "ordinaires".

- **Evolution**: La dynamicité d'un langage de scripts permet de découvrir "à la volée" les nouveaux services qui peuvent apparaître sur le bus, et donc de les utiliser dès qu'ils sont disponibles. Dans un autres sens, de légères modifications des interfaces IDL peuvent être compatibles avec un langage de scripts dynamique, et donc ne pas nécessiter de ré-écriture chez les utilisateurs de ces scripts.

Comme nous venons de le voir, les avantages d'un langage de scripts sont importants. Les nombreuses utilisations impliquent qu'un tel langage contienne les mécanismes permettant de découvrir dynamiquement les objets et leurs types, de pouvoir invoquer ces objets sans phase de compilation, de pouvoir naviguer dans les habituels graphes d'objets connectés, et finalement de pouvoir aussi en créer. Tout cela implique de pouvoir acquérir dynamiquement les talons des types d'objets rencontrés, sans qu'il ait été prévu de les rencontrer.

3.3. Exemples de langages de scripts

Ce type d'approche a aussi été proposé par les environnements ILU [Jan96] et TclDii [ASG+94]:

- ILU est un bus compatible CORBA développé par Xerox Parc pour un grand nombre de langage (C, C++, Java, Common Lisp and Python). Python [vR96] est un langage de scripts du CWI Amsterdam. La liaison entre ILU et Python permet l'accès interactif aux objets CORBA, mais l'interpréteur doit ici être lié avec les talons prégénérés des types d'objets qu'il pourra manipuler. Cela limite son utilisation en univers très évolutif, puisqu'il ne peut découvrir de nouveaux types d'objets pendant l'exécution.

- TclDii est un outil de l'environnement Web* du CERC. Il s'agit d'un lien entre le langage interprété Tcl et le monde CORBA. Il utilise clairement une approche dynamique, mais, contrairement à CorbaScript, il n'utilise pas les possibilités du dépôt d'interfaces, se privant donc d'un typage dynamique et automatique. Il reste donc complexe à utiliser et ne permet pas la découverte d'interfaces IDL pendant l'exécution.

4. Le langage CorbaScript

4.1. Fonctionnalités de base

CorbaScript est un langage de haut niveau, modulaire et orienté-objet comprenant procédures, classes, héritage multiple et polymorphisme. C'est un langage interprété (*cf.* figure 3) accessible en ligne de commande ou par exécution de fichiers script. Un script est un ensemble structuré d'instructions, avec variables, structures de contrôle et exceptions.

```
unix_promt> cssh
CorbaScript 1.0 for OmniBroker for C++
Copyright 1996-97 LIFL, France
>>>

unix_promt> cssh a_script_file.cs
```

Figure 3: L'interpréteur CorbaScript

Un petit nombre de types de base sont cablés dans l'interpréteur: integer, string, array, associate table, basic IDL data types, etc. La conformité des expressions est testée dynamiquement à l'exécution par des mécanismes de typage dynamique. Toutes les valeurs sont des objets. La notation pointée traditionnelle est utilisée pour exprimer les invocations de méthodes.

4.2. Utilisation automatique de CORBA

Quand un utilisateur invoque une méthode sur un objet CORBA, l'interpréteur teste si l'invocation est conforme à la spécification IDL contenue dans le dépôt d'interfaces. Pour des raisons d'efficacité, un cache local conserve ces informations en provenance de l'IR pour accélérer les prochains tests et réduire les communications avec l'IR.

L'invocation est ensuite réalisée via l'interface DII qui permet de construire dynamiquement des requêtes compréhensibles par le bus.

De plus les classes CorbaScript permettent de créer dynamiquement des objets visibles sur le bus. L'interpréteur utilise l'interface DSI de Corba pour intercepter et décoder les requêtes qui pourraient être envoyées vers des objets créés de cette manière.

4.3. Accès aux définitions IDL

Par CorbaScript, un utilisateur peut accéder dynamiquement et interactivement aux spécifications IDL de l'IR. Cela lui permet de les découvrir. L'utilisateur doit uniquement donner le nom du module qu'il souhaite explorer, comme présenté dans la figure 4.

Comme on le voit, CorbaScript est connecté de façon transparente au dépôt d'interfaces, et à toutes les définitions IDL qui s'y trouvent. De plus, CorbaScript ajoute quelques fonctionnalités à ces définitions,

```
>>> GridService.Grid
< OMG-IDL interface GridService::Grid {
   attribute readonly struct Coord dim;
   void set (in struct Coord pos, in Value val)
      raises(GridService::InvalidCoord);
   Value get (in struct Coord pos)
      raises(GridService::InvalidCoord);
   void destroy ();

   build-in readonly attribute _type;
   build-in method _is_a(type);
   build-in readonly attribute _ior;
   ... other internal needed features ...
} >

>>> GridService.Coord
< OMG-IDL struct Coord { unsigned short x; unsigned short y; } >
```

Figure 4: Accès aux définitions IDL

comme par exemple la possibilité d'obtenir le type IDL d'un objet (*_type*), de tester le type d'un objet (*_is_a*), ou encore d'obtenir une référence sur un objet (*_ior*), etc.

4.4. Accès aux objets CORBA

Pour accéder à un objet CORBA, l'utilisateur doit en connaître une référence. CorbaScript propose différentes manières d'obtenir ces références. Un utilisateur peut donner une adresse d'objet qu'il connaît. Celle-ci doit être au format IOR de l'OMG ou au format URL (IP host, IP port, et une identification locale de l'objet). L'utilisateur peut aussi utiliser le service standard de nommage de CORBA. Il doit alors se connecter à ce service par l'opération *resolve_initial_references* directement accessible dans l'interface *CORBA::ORB* qui représente le bus.

```
>>> factory = GridService.Factory("IOR:000000000000001c4...")

>>> factory = GridService.Factory("iiop://an_IP_host_name:5000/factory")

>>> NS = CORBA.ORB.resolve_initial_references ("NameService")
>>> factory = NS.resolve ( [ ["aGridService", ""] ] )
```

Figure 5: Accès aux objets

On remarquera dans l'exemple que l'utilisateur n'a pas à spécifier le type de l'objet retourné, puisque CorbaScript utilise lorsqu'il le faut le dépôt d'interfaces pour acquérir les définitions de type. Ici donc quand le service de nommage retourne sa réponse, CorbaScript crée automatiquement une référence pour le type dynamique de l'objet retourné. S'il ne connaît pas ce type, il accède à l'IR pour obtenir sa définition. De cette manière, l'utilisateur peut dynamiquement naviguer dans le service de nommage en découvrant les types des objets.

4.5. Opérations, Attributs et Exceptions

Comme illustré dans l'exemple précédent, l'utilisateur n'a pas à indiquer les types de paramètres lors des invocations puisque CorbaScript fait automatiquement les conversions. La valeur *[["aGridService", ""]]* est un tableau contenant un tableau de deux chaînes de caractères. Cette valeur est automatiquement convertie en une structure de type *CosNaming::Name* [1].

```
>>> grid = factory.create_grid ([20,5], 2)
>>> # or more precisely, (GridService.Coord(20,5), GridService.Value(1))

>>> grid.dim
GridService::Coord(20,5)

>>> try {
      grid.set([100,100],10)
    } catch (GridService::InvalidCoord e) {
      print "GridService::InvalidCoord raises on ", e.pos, '\n'
    }
GridService::InvalidCoord raises on GridService::Coord(100,100)
```

Figure 6: Operations, Attributs et Exceptions

Cet exemple illustre la simplicité d'invoquer des opérations et d'accéder à des attributs d'objets CORBA distants. CorbaScript fournit de plus un mécanisme d'exception à la C++ qui permet de traiter les exceptions définies en IDL ou les exceptions provenant du système CORBA.

[1] c-à-d une séquence de *CosNaming::NameComponent* contenant deux champs *string*. Ce type est utilisé par le service de nommage pour gérer des associations entre noms symboliques et références d'objet.

4.6. Procédures et Modules

Ces exemples sont très simples. CorbaScript permet cependant de créer des scripts plus complexes en utilisant procédures et modules. Comme pour les invocations, les paramètres des procédures ne sont pas typés. Les procédures peuvent être regroupées en modules chargeables.

Le fragment suivant fait partie du module *grid_user*. Ce module contient une procédure (*DisplayGrid*) qui itère sur une grille pour obtenir ses valeurs (*get*) et les afficher. Un utilisateur peut charger ce module et exécuter ce type de procédure sur un objet grille pour lequel il aurait obtenu une référence.

```
proc DisplayGrid (grid)
{
  dim = grid.dimension
  h = dim.y
  w = dim.x
  print "The dimension of this grid is ", w, "*", h, "\n\n"

  # iterate to get each values of the grid
  for i in range (0, h-1) {
    for j in range (0, w-1) {
      print ' ', grid.get([i,j])
    }
    print '\n'
  }
}

>>> import grid_user
>>> grid_user.DisplayGrid(grid)
The dimension of this grid is 20*5

 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 7: Exemple de procédure

De cette manière, de nombreuses fonctionnalités peuvent être implantées sans être un expert CORBA. Il est juste nécessaire de connaître CorbaScript et les interfaces IDL que l'on veut utiliser.

4.7. Implantation d'interfaces IDL

Un script contient normalement des valeurs locales et des références sur des objets distants. il agit donc souvent comme un client CORBA.

```
class GRID {
  # GRID instance initialisation
  proc __GRID__ (self, dim, init_value)
  { # This GRID instance (self) is a GridService.Grid CORBA object
    CSOA.create (self, GridService.Grid)

    # set the GRID instance attributes
    self.dim = dim
    self.values = create_matrix (dim, init_value) # not detailed
  }

  # Implementation of the GridService::Grid IDL specification

  proc _get_dimension (self)  # readonly 'dimension' attribute
  {
    return self.dim
  }
  proc set (self, pos, val)   # 'set' operation
  {
    try {
      self.values[pos.y][pos.x] = val
    } catch (BadIndex excep) {
      throw GridService.InvalidCoord(pos)
    }
  }
  proc get (self, pos)        # 'get' operation
  {
    try {
      return self.values[pos.y][pos.x]
    } catch (BadIndex excep) {
      throw GridService.InvalidCoord(pos)
    }
  }
  proc destroy (self)         # 'destroy' operation
  {
    CSOA.dispose (self)
  }
}

class FACTORY
{
  proc __FACTORY__ (self)
  {
    CSOA.create (self, GridService.Factory)
  }

  proc create_grid (self, dim, init_value)
  {
    grid = GRID(dim, init_value)
    return grid._this
  }
}
```

Figure 8: L'implantation du service de grilles

Une autre fonctionnalité permet l'implantation de nouveaux types d'objets (locaux ou CORBA). Elle provient de l'orientation objet du langage qui offre classes, héritage et polymorphisme. Ces concepts sont projetés dans le monde CORBA.

Dans la version actuelle, toute méthode d'instance doit avoir un premier paramètre qui dénote l'instance courante. Cependant il n'y a pas de convention figée pour le nom de ce paramètre: l'utilisateur peut choisir au choix *self*, *this* ou autre chose. A travers cette référence, le code d'une méthode peut accéder aux attributs de l'instance. Ces attributs sont implicitement déclarés à leur première affectation.

La figure 8 présente une possible implantation du service de grilles. Les classes *GRID* et *FACTORY* implantent respectivement les interfaces *GridService::Grid* et *GridService::Factory*. CorbaScript force une convention de nommage pour les méthodes d'initialisation: (*__GRID__* et *__FACTORY__*). Les opération IDL sont implantées par des méthodes de même nom. Les attributs sont aussi implantés par les méthodes avec la convention de nommage suivante: un préfixe *_get_* pour un accès en lecture, un préfixe *_set_* pour un accès en écriture.

Le symbole *CSOA* dénote l'Adaptateur d'Objet de CorbaScript. Cet objet fournit les opérations pour connecter et déconnecter un objet à CORBA. L'opération *create* permet d'associer une instance CorbaScript à un nouvel objet CORBA: le premier paramètre denote l'instance, le deuxième denote l'interface IDL que l'instance implante. L'opération *dispose* coupe cette association, rendant l'objet CORBA invalide.

4.8. Création de serveurs CORBA

De cette façon un script peut devenir un serveur CORBA accessible depuis tout autre programme (compilé ou script) CORBA. La figure 9 montre l'implantation du serveur *Grid*. On notera l'importation du module *Grid* contenant les classes *GRID* et *FACTORY*. Une instance de *FACTORY* est créée et enregistrée dans le service de nommage standard de CORBA par l'opération *bind*. Une boucle est ensuite lancée pour se mettre en attente des requêtes en provenance de l'ORB. Finalement, la fabrique est désenregistrée du service de nommage (opération *unbind*), et le serveur est stoppé.

Nous avons fait un rapide tour des fonctionnalités de CorbaScript. CorbaScript offre de nombreuses possibilités pour développer rapidement des clients et serveurs CORBA. D'autres fonctionnalités n'ont pas été introduites ici, comme l'héritage, l'introspection, le chargement dynamique de librairies C, le lien avec les objets Java, etc.

```
# Load the GridService implementation 'grid_impl.cs' file
import grid_impl

# Create a FACTORY instance
factory = grid_impl.FACTORY()

# Obtain the Name Service reference
NS = CORBA.ORB.resolve_initial_references("NameService")

# Register the server object into the Name Service
NS.bind ( [["aGridService", ""]], factory._this)

# Start the main loop to wait for ORB requests
CSOA.run ()

# Unregister the server object from the Name Service
NS.unbind ( [["aGridService", ""]])
```

Figure 9: Le script principal du serveur de grilles

5. Implantation de CorbaScript

L'interpréteur du langage CorbaScript est écrit en C++. Il comprend 27000 lignes de code éclatées dans un centaine de classes regroupées en trois module (*cf.* figure 10). Le module *ObjectScript* prend en charge le noyau de l'interpréteur: analyse syntaxique, machine virtuelle, et implantation des concepts de base comme les valeurs, les types, les procédures, les modules, les classes et les instances. Le module *CorbaScript* gère le cache du dépôt d'interfaces, réalise les conversions entre les valeurs ObjectScript et les valeurs IDL, et utilise les interfaces DII et DSI pour envoyer et recevoir des requêtes IIOP. Enfin, le module *CorbaTools* encapsule tout ce qui peut être spécifique à l'ORB utilisé.

Grâce à cette architecture modulaire, le langage CorbaScript est totalement isolé de l'ORB sous-jacent. Notre langage peut donc évoluer indépendamment des mécanismes de CORBA (avec par exemple de nouvelles constructions syntaxiques ou de nouvelles valeurs CorbaScript). CorbaScript tourne actuellement sur ORBacus, un très bon bus CORBA, libre pour des utilisations non commerciales, utilisable avec C++ and Java [LS97]. Nous réalisons les portages du module CorbaTools sur MICO ORB [RP97] et VisiBroker for C++[Vis96].

En développant CorbaScript, un des gros problèmes que nous avons rencontré fut celui de construire des requêtes DII nécessitant des valeurs de type *CORBA::Any*. Un *Any* IDL peut contenir n'importe quelle valeur IDL. Il comprend un champ donnée (*void**) et un champ pour décrire le type de la donnée (*CORBA::TypeCode*). Corba 2.0 définit l'API *DynAny* permettant de créer un *Any* contenant une donnée complexe (structure, union, array, sequence), malheureusement, cette API n'est pas fournie avec tous les bus CORBA: par exemple le bus Orbix [ION95] ne la fournit pas. Nous avons implanté cette API sur ORBacus, mais aussi longtemps que cette API ne sera pas systématiquement implantée sur chaque bus, nous aurons à refaire ce travail.

De plus, tous les ORB ne fournissent pas un dépôt d'interfaces (IR), un élément essentiel pour le typage dynamique de CorbaScript. Les développeurs d'ORB n'ont pas encore tous compris l'importance des mécanismes dynamiques, se limitant au mode client/serveur statique. Des travaux comme les nôtres montrent que les apports de ces mécanismes peuvent être importants.

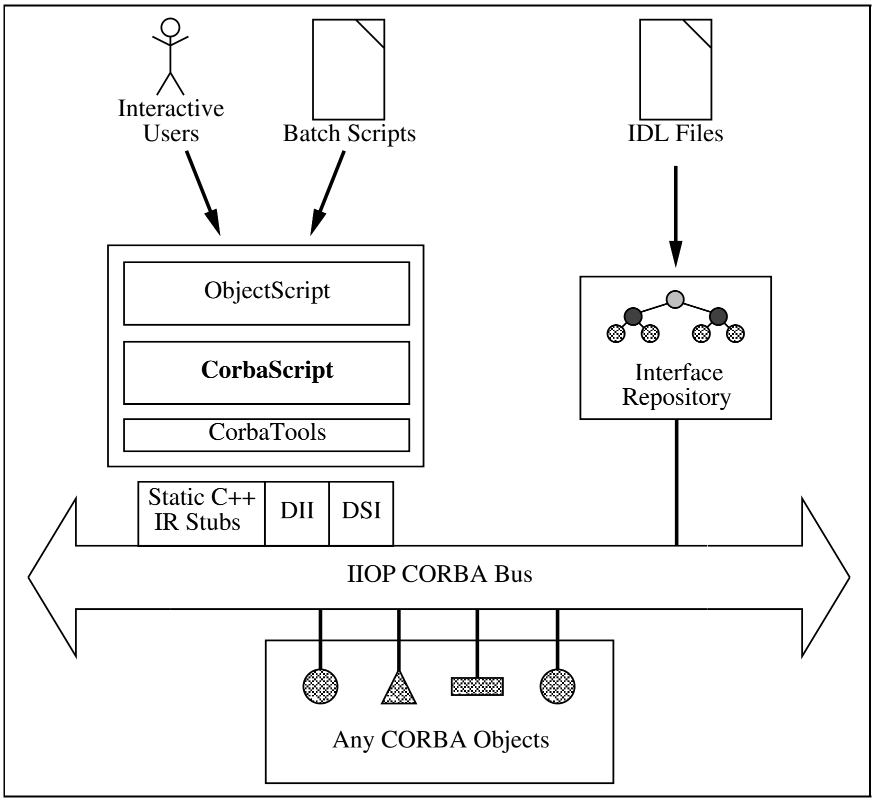

Figure 10: L'architecture de CorbaScript

6. Conclusion

Nous avons montré dans ce papier que le langage CorbaScript apporte les fonctionnalités nécessaires pour simplement et dynamiquement utiliser et implanter des services CORBA. La transparence qu'offre CorbaScript permet d'accéder simplement à tout objet CORBA visible. L'objectif de fournir un outil pour l'utilisateur final est donc atteint. CorbaScript est actuellement utilisé en formation sur les environnements CORBA. Il a été utilisé pour développer une passerelle générique d'intégration des objets CORBA dans le World Wide Web: l'environnement CorbaWeb [MGG96b]. La version 2.0 de CorbaScript a été proposée [OMG97] à l'OMG pour intégration dans les services communs de la norme CORBA.

Toute l'expérience issue de cette réflexion et de ces développements nous conduit à promouvoir une nouvelle vision du client/serveur à grande échelle sur la base d'outils s'adaptant dynamiquement à leurs environnements, ouverts, et utilisant les technologies normalisées que sont CORBA et le Web.

CorbaScript et CorbaWeb sont disponibles sur notre site Web `http://corbaweb.lifl.fr`.

Bibliographie

[ASG+94] George Almasi, Anca Suvaiala, Cristian Goina, Calin Cascaval, and V. ¡¡Juggy¿¿ Jagannathan. TclDii: A TCL Interface to the Orbix Dymanic Invocation Interface. Technical report, Concurrent Engineering Research Center (CERC), 1994.
<URL:http://www.cerc.wvu.edu/dice/iss/TclDii/TclDiiDoc.ps>.

[CK94] R. Cartell and M. Kaufmann. *The Object Database Standard: ODMG-93.* San Francisco, CA, U.S.A., 1994.

[Con96] TINA-C Consortium. Public TINA-C WWW server Home Page, 1996.
<URL:http://www.tinac.com>.

[GGM97] Jean-Marc Geib, Christophe Gransart, and Philippe Merle. *Corba : des concepts à la pratique.* Collection InterEditions. Masson, Paris, France, 1997.
ISBN : 2-225-83046-0.

[GM95] James Gosling and Henry McGilton. The Java Language Environment: a White Paper. Technical report, Sun Mi-

crosystems Inc., 7550 Garcia Avenue, Mountain View, CA 94043 U.S.A., October 1995.
<URL: http://java.sun.com/whitePaper/java-whitepaper-1.html>.

[GR83] Adele Goldberg and David Robson. *SMALLTALK-80. The Language and Its Implementation.* Addison-Wesley Publishing Company Inc., Massachusetts, U.S.A., 1983.

[ION95] IONA. *Orbix 2: Programming Guide and Reference Guide.* IONA Technologies Ltd., 8-34 Percy Place, Dublin 4, Ireland, 2.0 edition, November 1995.
<URL: http://www.iona.com/>.

[ISO92] ISO International Organization for Standardization, Genève, Suisse. *International Standard ISO/IEC 9075 : Information Technology — Database — languages — SQL*, third edition, November 1992.

[Jan96] Bill Janssen. The Inter-Language Unification – ILU. Technical report, Xerox Palo Alto Research Center (Xerox PARC), 1996.
<URL: ftp://ftp.parc.xerox.com/pub/ilu/ilu.html>.

[Lin94] P.F. Linington. Introduction to the Open Distributed Processing Reference Model. In J. De Meer, V. Heymer, and R. Roth, editors, *Proceedings of IFIP TC6/WF6.4 International Workshop on Open Distributed Processing, Berlin, Allemagne*, pages 3–13. Elsevier Science Publishers B.V. (North Holland), 8–11 Octobre 1994.

[LS97] Marc Laukien and Uwe Seimet. OmniBroker Version 2.0. User and Programmer guide, Object-Oriented Concepts, Inc., 44 Manning Road, Billerica, MA 01821, USA, August 1997.
<URL: http://www.ooc.com>.

[Mer97] Philippe Merle. *CorbaScript - CorbaWeb : propositions pour l'accès à des objets et services distribués.* Thèse de doctorat en informatique, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Université des Sciences et Technologies de Lille (USTL), France, January 1997.

[MGG96a] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. CorbaScript and CorbaWeb: A Generic Object-Oriented Dynamic Environment upon CORBA. In *Proceedings of TOOLS Europe'96, Palais des Congrès, Paris, France, February 26–29, 1996.* Prentice-Hall, February 1996.

[MGG96b] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. CorbaWeb: A Generic Object Navigator. In *The Fifth International World Wide Web Conference (WWW5), Paris, France, May 6–10, 1996*, volume 28 (7-11), pages 1269–1281, Paris, France, May 1996. Elsevier, Computer Networks and ISDN System.
Also published in Dr. Dobb's SOURCEBOOK, Volume 22, Issue 13, 'Distributed Objects'*, Miller Freeman Inc. Editor, January/February 1997.*

[MGG96c] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. How to Make Corba Objects User-Friendly with a Generic Object-Oriented Dynamic Environment? In *European Conference on Object-Oriented Programming, Workshop on Putting Distributed Objects to Work*, Linz, Austria, July 1996.

[MGG97a] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. CorbaScript et CorbaWeb : Concevoir, déployer et utiliser des services distribués. In *NOTERE'97, Colloque International sur les NOuvelles TEchnologies de la REpartition*, Pau, France, November 1997.

[MGG97b] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. Generic Tools: A New Way to Use CORBA. In *European Conference on Object-Oriented Programming, Workshop on CORBA: Implementation, Use, and Evaluation*, Jyväskyä, Finland, June 1997.

[OHE95] Robert Orfali, Dan Harkey, and Jeri Edwards. *Essential Distributed Objects Survival Guide.* John Wiley and Sons Inc., New York, U.S.A., 1995.

[OMG96] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG TC Document 96-7-X, Object Management Group, Headquaters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., July 1996.
<URL:http://www.omg.org/>.

[OMG97] Object Management Group. CORBA Scripting Language Request For Proposal. OMG TC Document orbos/97-06-13, Object Management Group, Headquaters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., June 1997.

[RP97] Kay Romer and Arno Puder. MICO: CORBA 2.0 Implementation. User and Programmer guide, Computer Science Department, University of Frankfurt, Germany, 1997.
<URL:http://www.vsb.cs.uni-frankfurt.de/~mico>.

[Sol94] Richard Mark Soley. Object Management Architecture Guide. OMG TC Document 94-11-1, Object Management Group, Headquaters, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., November 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company Inc., Massachusetts, U.S.A., 2nd edition, December 1991.

[Vis96] Visigenic. VisiBroker for C++: Reference Manuals, 1996.
<URL:http://www.visigenic.com/>.

[vR96] Guido van Rossum. Python Language Home Page, 1996.
<URL:http://www.python.org/>.