Intergiciel et Construction d'Applications Réparties

Chapitre 6

La plate-forme .NET

.NET[1] est en passe de devenir une technologie fréquemment utilisée pour la réalisation de projets informatiques. Dans ce chapitre nous abordons les principaux principes de cette plate-forme, diverse et parfois complexe intégrant les standards actuels. La section d'introduction met en évidence la complétude et la richesse de l'ensemble des bibliothèques de la plate-forme .NET. Elle nous permet de fixer le vocabulaire pour différencier les éléments de la plate-forme normalisés par Microsoft de ceux qui sont restés propriétaires afin de pouvoir très rapidement comparer les différentes plates-formes de développement. La section 6.2 insiste sur l'architecture générale de la plate-forme et sur ses principales caractéristiques. La section 6.2.4 présente le modèle de composants et la machine virtuelle permettant leur exécution. Les trois sections suivantes font respectivement le lien entre les composants .NET et les services techniques (6.4), l'étage d'accès aux données (6.4.1) et l'étage de présentation (6.3.2) présents dans toute architecture à N étages (N-*tier*). La section 6.3 est constituée d'un petit exemple permettant d'illustrer les principaux principes de la plate-forme. Le chapitre se termine par une évaluation qualitative et quantitative et par une présentation de l'évolution future de cette plate-forme.

6.1 Historique, définitions et principes

6.1.1 Généralité

Microsoft .NET est une architecture logicielle destinée à faciliter la création, le déploiement des applications en général, mais plus spécifiquement des applications Web ou des services Web. Cette architecture logicielle concerne aussi bien les clients que les serveurs qui vont dialoguer entre eux à l'aide d'XML. Elle se compose de quatre principaux éléments :

- un modèle de programmation qui prend en compte les problèmes liés aux déploiements des applications (gestion de version, sécurité) ;

[1]une partie de ce texte a été publiée dans les *Techniques de l'Ingénieur* - Traité "Technologies logicielles et architecture des systèmes" - vol. H3540 - février 2006

- des outils de développement dont le cœur est constitué de Visual Studio .NET ;
- un ensemble de systèmes serveurs représentés par Windows Server 2003, SQL Server ou Microsoft Biztalk Server qui intègrent, exécutent et gèrent les services Web et les applications ;
- un ensemble de systèmes clients représenté par Windows XP, Windows CE ou Microsoft Office 2003.

En fait, Microsoft .NET est essentiellement un environnement de développement et d'exécution reprenant, entre autres, les concepts de la machine virtuelle de Java, via le CLR (*Common Language Runtime*). Le principe, bien connu, est le suivant : la compilation du code source génère un objet intermédiaire dans le langage MSIL (*Microsoft Intermediate Language*) indépendant de toute architecture de processeur et de tout système d'exploitation hôte. Cet objet intermédiaire, est ensuite compilé à la volée, au sein du CLR, au moyen d'un compilateur JIT (*Just In Time*) qui le transforme en code machine lié au processeur sur lequel il réside. Il est alors exécuté.

Au contraire de Java, pour lequel le code intermédiaire (*byte-code*) est lié à un seul langage source, le code intermédiaire de la plate-forme .NET est commun à un ensemble de langages (C#, VB[2], C++, etc.). Le CLR en charge de son exécution contient un ensemble de classes de base liées à la gestion de la sécurité, gestion de la mémoire, des processus et des *threads*.

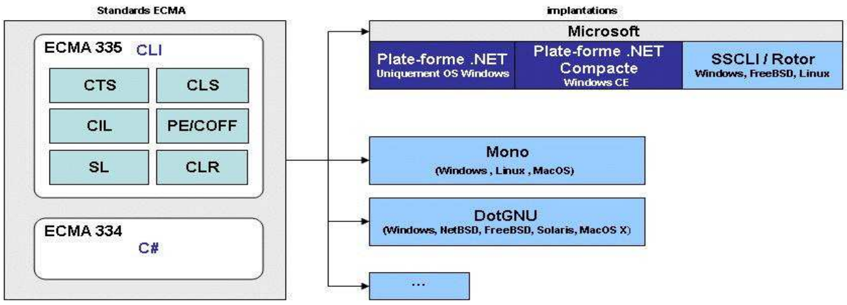

Figure 6.1 – Les différentes plates-formes .NET

La figure 6.1 résume les différents aspects de ces normes qui concernent le langage de programmation C# "inventé" pour l'occasion [3] et la plupart des interfaces internes [4] de la plate-forme. Cette même figure présente les principales implémentations de ces spécifications.

[2]VB.NET est la nouvelle version de Visual Basic, il intègre la notion d'objets qui existe dans .NET et les programmeurs de l'actuelle version 6.0 sont invités à faire migrer leurs codes
[3]ECMA 334 puis ISO/IEC 23270 : C# Langage Specification
[4]ECMA 335 puis ISO/IEC 23271 : Common Langage Interface (CLI)

6.1.2 Implantations

Rotor - SSCLI

Le code source partagé CLI (*Common Language Infrastructure*)[5] [Stutz et al. 2003] correspond à une implémentation fonctionnelle des spécifications du langage C# et de la CLI normalisées par l'ISO. Cette implémentation va au delà de ces spécifications en proposant un environnement d'exécution, des bibliothèques de classes, le compilateur C# et JScript, un débogueur, des outils, la couche PAL (Platform Adaptation Layer), une version portable du système et portée sur Windows XP, FreeBSD et MacOS X, ainsi qu'une suite complète de test. Elle est proposée à des fins non commerciales via la licence SSCLI dont les principales caractéristiques sont :

- le logiciel doit être utilisé à des fins non commerciales (enseignement, recherche universitaire ou expérimentations personnelles). Il peut toutefois être distribué au sein d'ouvrages ou autres supports de cours liés à l'enseignement, ou publié sur des sites Web dont le but est de former à son utilisation.
- toutes utilisations du logiciel à des fins commerciales sont prohibées.
- le logiciel peut être modifié et le résultat de ces modifications peut être distribué dans un but non commercial à la condition de ne pas accorder de droit supplémentaire.
- le logiciel est livré sans garanties.

Autres implémentations

Suite à l'adoption officielle de ces deux standards par l'ECMA en décembre 2001, et plus tard par l'ISO2 en avril 2003, la réaction ne se fit pas attendre. Différentes implantations du CLI basées sur la spécification ECMA ont vu le jour permettant ainsi l'exécution et le développement d'applications .NET sur des systèmes d'exploitation non Microsoft comme Linux, FreeBSD et MacOS. Nous pouvons par exemple citer deux initiatives logiciels libres (*open-source*), indépendantes de Microsoft, et ayant une réelle importance : Mono [6] [Edd Dumbill and M. Bornstein 2004] et DotGNU [7] [Weatherley and Gopal 2003].

6.2 Architecture générale et concepts

6.2.1 CLR

Le composant central de l'architecture .NET est le CLR (*Common Language Runtime*) [Stutz et al. 2003] qui est un environnement d'exécution pour des applications réparties écrites dans des langages différents. En effet, avec .NET on peut véritablement parler d'interopérabilité entre langages. Cette interopérabilité s'appuie sur le CLS (*Common Language Specification*) qui définit un ensemble de règles que tout compilateur de la plateforme doit respecter. Le CLS utilise entre autres un système de types unifié permettant d'avoir le même système de type entre les différents langages et le même système de type entre les types prédéfinis et les types définis par l'utilisateur.

[5]Rotor : `http ://msdn.microsoft.com/net/sscli/`
[6]Mono : `http ://www.go-mono.org`
[7]DotGNU : `http ://www.southern-storm.com.au/portable_net.html`

Un autre concept majeur, lié au CLR est celui de code géré, c'est à dire de code exécuté sous le contrôle de la machine virtuelle. Dans cet environnement, un ensemble de règles garantissent que les applications se comporteront d'une manière uniforme, et ce indépendamment du langage ayant servi à les écrire.

Pour répondre à ce souci d'interopérabilité entre les langages, la plate-forme .NET contient une bibliothèque de classes très complète, utilisable depuis tous les langages et permettant aux développeurs d'utiliser une même interface de programmation pour toutes les fonctions offertes.

Dans .NET, le langage lui-même est essentiellement une interface syntaxique des bibliothèques définies dans le CLR. Grâce au jeu commun de ces bibliothèques, tous les langages disposent théoriquement des mêmes capacités car ils doivent, exception faite de la déclaration des variables, passer par ces bibliothèques pour créer des *threads*, les synchroniser, sérialiser des données, accéder à internet, etc.

6.2.2 MSIL et JIT

Pour permettre la compilation de différents langages sur la plate-forme .NET, Microsoft a défini une sorte de langage d'assemblage, indépendant de tout processeur baptisé MSIL (*Microsoft Intermediate Language*). Pour compiler une application destinée à .NET, le compilateur concerné lit le code source dans un langage respectant les spécifications CLS, puis génère du code MSIL. Ce code est traduit en langage machine lorsque l'application est exécutée pour la première fois. Le code MSIL produit contient un en-tête au standard PE (Win-32 Portable Executable) qui permet de différencier les exécutables .NET des autres exécutables de l'OS Windows en chargeant au démarrage de l'application un ensemble de DLL spécifiques précisées dans cet en-tête.

Pour compiler à l'exécution le code MSIL, la plate-forme .NET utilise un compilateur à la volée (JIT Compiler - *just-in-time compiler*) qui place le code produit dans un cache. Sur la plate-forme .NET, trois JIT sont disponible :

- Génération de code à l'installation : le code MSIL est traduit en code machine lors de la phase de compilation comme le fait tout compilateur.
- JIT : les méthodes sont compilées une par une à la volée lors de leur première exécution, ou lorsqu'elles ne sont plus en cache.
- EconoJIT : jit permettant de minimiser la place mémoire utilisée pour exécuter un programme et qui sera utilisé pour les équipements portables.

Les compilateurs de la plate-forme .NET incorporent dans l'exécutable produit différentes métadonnées. Ces métadonnées décrivent l'exécutable produit et lui sont systématiquement incorporées pour pouvoir être atteintes lors de l'exécution. Il est ainsi possible dynamiquement de connaître les différents types déclarés par un exécutable, les méthodes qu'il implémente, etc. La lecture des métadonnées s'appelle réflexion (voir 2.4 et la plate-forme .NET fournit de nombreuses classes permettant de manipuler de manière réflexive les exécutables. Voici un exemple de code qui crée dans un premier temps un exécutable et regarde par la suite les métadonnées.

Fichier calculatrice.cs

```
using System;
```

```
class MaCalculatrice {
    public int Add(int a, int b) {
        return a+b;
    }
    public int Sub(int a, int b) {
        return a-b;
    }
}
```

Fichier application.cs

```
public class MonApplication {
    static void Main (string[] args) {
        MaCalculatrice calculette = new MaCalculatrice ();
        int i = 10, j = 20;
    Console.WriteLine ("{i} + {j} =", i, j, calculette.Add (i, j));
    }
}
```

La compilation de l'application, s'effectue par étape :
- génération de la bibliothèque calculatrice.dll
- génération de l'exécutable application.exe

```
% csc.exe /target:library /out:calculatrice.dll calculatrice.cs
% csc.exe /target:exe /reference:calculatrice.dll application.cs
```

```
using System;
using System.IO;
using System.Reflection;

public class Meta {
public static int Main () {
    // lire l'assembly
    Assembly a = Assembly.LoadFrom ("application.exe");

    // lire tous les modules de l'assembly
   Module[] modules = a.GetModules();

    // inspecter tous le premier module
    Module module = modules[0];

    // lire tous les types du premier module
    Type[] types = module.GetTypes();

   // inspecter tous les types
   foreach (Type type in types) {
       Console.WriteLine("Type {0} a cette méthode : ", type.Name);

      // inspecter toutes les méthodes du type
      MethodInfo[] mInfo = type.GetMethods();
```

```
      foreach (MethodInfo mi in mInfo) {
           Console.WriteLine (" {0}", mi);
      }
   }
   return 0;
 }
}
```

La compilation de l'application permettant de lire l'application "application.exe" puis son exécution, s'effectue de la manière suivante :

```
% csc.exe meta.cs
% meta.exe

Type MaCalculatrice a cette méthode :
 Int32 Add(Int32, Int32)
 System.Type GetType()
 System.String ToString()
 Boolean Equals(System.Object)
 Int32 GetHashCode()
Type MonApplication a cette méthode :
 System.Type GetType()
 System.String ToString()
 Boolean Equals(System.Object)
 Int32 GetHashCode()
```

L'outil ILDASM (*Intermediate Language Disassembler*) qui permet d'afficher de manière hiérarchique les informations sur une application utilise essentiellement la réflexion. La figure 6.1 montre à quoi ressemble le programme précédent sous ILDASM.

6.2.3 Sécurité et déploiement des composants

La sécurité est cruciale pour toutes les applications. Elle s'applique dès le chargement d'une classe par le CLR, via la vérification de différents éléments liés aux règles de sécurité définies dans la plate-forme : règles d'accès, contraintes de cohérence, localisation du fichier, etc. Des contrôles de sécurité garantissent que telle ou telle portion de code est habilitée à accéder à telle ou telle ressource. Pour cela, le code de sécurité vérifie rôles et données d'identification. Les contrôles de sécurité peuvent être effectués entre processus ou entre machines afin d'assurer la sécurité même dans un contexte distribué.

Windows nous a habitués à un déploiement hasardeux pour lequel il est nécessaire de gérer de multiples fichiers binaires, la base de registres, des composants COM, des bibliothèques comme ODBC. Fort heureusement, .NET change cette approche et le déploiement des applications repose sur la copie dans un répertoire spécifique, le GAC (*Global Assemblies Cache*), ou dans l'arborescence utilisateur, des fichiers exécutables produits. Ces fichiers exécutables (mais pas toujours) "assembly[8]" sont un paquet regroupant des éléments devant être déployé simultanément. Ce sont généralement des fichiers exécutable répartis sur un ou plusieurs fichiers physiques. Les problèmes antérieurs à .NET ont été éliminés et il n'est plus nécessaire d'enregistrer des composants dans la base de

[8]Un assembly est une unité de déploiement.

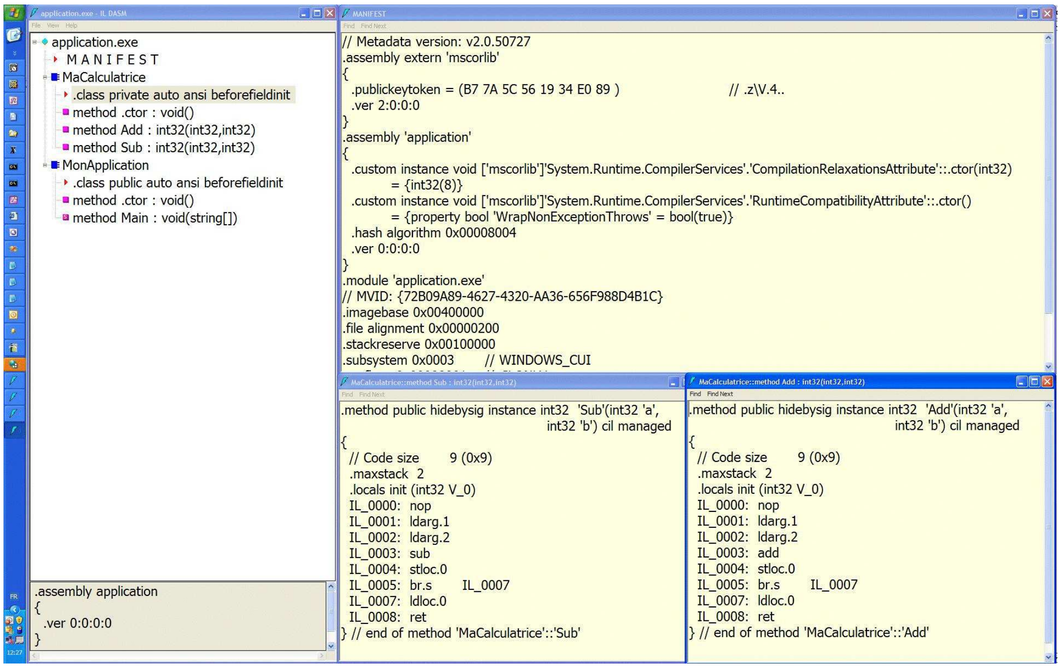

Figure 6.2 – Le programme de 6.2.2 analysé avec ILDASM

registre. .NET permet aussi de gérer dans le GAC plusieurs versions du même fichier et de rediriger à la demande les appels vers un assemblage vers une version plus récente si c'est souhaité par le concepteur de l'application. En effet, grâce aux métadonnées et à la réflexion, tous les composants sont désormais auto-descriptifs, y compris leur version. Cela à pour conséquence, que toute application installée est automatiquement associée aux fichiers faisant partie de son assemblage.

Une partie de la sécurité du code repose sur la possibilité de signer celui-ci à l'aide d'une clé privée afin de pouvoir identifier ultérieurement la personne ou l'organisation qui a écrit ce code à l'aide de la clé publique associée. Si l'exécution dans le répertoire courant ne nécessite pas la signature du code, toute publication dans le GAC ainsi que la gestion des versions dans celui-ci le nécessite.

6.2.4 Composants .NET

Composants

La brique de base d'une application .NET est le composant (littéralement assemblage ou *assembly*). Il peut s'agir d'un exécutable (`.exe`) ou d'une bibliothèque (`.dll`) contenant des ressources, un manifeste, des métadonnées, des types et du code exécutable sous la forme d'une liste d'instructions en langage intermédiaire. Ces métadonnées sont utilisées par l'environnement d'exécution (run-time). La définition des classes peut être obtenue directement à partir du composant, en examinant les métadonnées Au contraire de l'approche Java RMI et de Corba, aucun autre fichier n'est nécessaire pour utiliser un composant et par conséquent, aucun fichier IDL, aucune bibliothèque de type ou couple talon client/serveur (*stub/skeleton*) n'est nécessaire pour accéder à un composant depuis un autre langage ou un autre processus. Il n'est pas nécessaire de déployer des informations de configuration séparées pour identifier les attributs de services requis par le développeur. Plus important encore, les métadonnées étant générées à partir du code source durant le processus de compilation et stockées avec le code exécutable, elles ne sont jamais désynchronisées par rapport à celui-ci.

Le rôle du composant est de plusieurs natures.

- C'est une unité de déploiement unique. C'est une unité d'exécution unique. Chaque composant est chargé dans un domaine d'application différent lui permettant ainsi d'être isolé des autres composants de l'application.
- Il intègre un mécanisme de gestion de version spécifique.
- Il permet également de définir la visibilité des types qu'il contient afin d'exposer publiquement ou non certaines fonctionnalités vers d'autres domaines d'application.
- Il permet de sécuriser l'ensemble des ressources qu'il contient de manière homogène car c'est à la fois une unité de déploiement, d'exécution (au sein d'un domaine d'application) et une unité pour la gestion des versions.

Déploiement des composants

Le modèle de déploiement d'applications de la plate-forme .NET règle les problèmes liés à la complexité de l'installation d'applications, à la gestion des versions de DLL dans le monde Windows ou du positionnement de la variable `CLASSPATH` dans le monde Java. Ce modèle utilise la notion de composant qui permet de déployer de façon unitaire un groupe de ressources et de types auquel est associé un ensemble de métadonnées appelé manifeste (*assembly manifest*).

Le manifeste comprend entre autres la liste des types et des ressources visibles en dehors du composant et des informations sur les dépendances entre les différentes ressources qu'il utilise. Les développeurs peuvent aussi spécifier des stratégies de gestion des versions pour indiquer si l'environnement d'exécution doit charger la dernière version d'un composant, une version spécifique ou la version utilisée lors de la conception. Ils peuvent aussi préciser la stratégie, spécifique à chaque application, pour la gestion de la recherche et le chargement des composants.

La plate-forme .NET permet le déploiement côte à côte (*side by side*) des composants : plusieurs copies d'un composant, mais pas nécessairement de la même version, peuvent résider sur un même système. Les assemblages peuvent être privés ou partagés par plusieurs applications et il est possible de déployer plusieurs versions d'un composant simultanément sur une même station. Les informations de configuration d'applications définissent l'emplacement de recherche des composants et permettent ainsi à l'environnement d'exécution de charger des versions différentes du même composant pour deux applications différentes exécutées simultanément. Ceci élimine les problèmes relatifs à la compatibilité des versions de composants et améliore la stabilité globale du système.

Si nécessaire, les administrateurs peuvent ajouter aux composants lors du déploiement des informations de configuration telles que des stratégies de gestion des versions différentes. Cependant, les informations d'origine fournies au moment de la conception ne sont jamais perdues. En raison du caractère auto-descriptif des composants, aucun enregistrement explicite auprès du système d'exploitation n'est nécessaire. Le déploiement d'une application peut se limiter à la copie de fichiers dans une arborescence de répertoires. Les informations de configuration sont stockées dans des fichiers XML qui peuvent être modifiés avec tout éditeur de texte. La tâche se complique légèrement si des composants non-gérés (*unmanaged components*) sont nécessaires au fonctionnement de l'application.

Le terme *"unmanaged"* rencontré ci-dessus fait référence au processus de compilation des composants .NET.

– Soit la compilation du composant respecte toutes les spécifications de la plate-forme .NET et alors l'exécution de ce composant est complètement prise en charge par la machine virtuelle .NET et l'on parle alors de code *"managed"*. Le composant associé est lui aussi déclaré *"managed"*.
– Soit la compilation du composant ne respecte pas toutes les spécifications de la plate-forme .NET, et c'est particulièrement le cas si l'on souhaite reprendre sans modification du code existant (C ou C++ par exemple) manipulant directement des pointeurs. Alors la machine virtuelle .NET isole ce composant dans un espace spécifique, n'assure pas pour ce composant une gestion mémoire spécifique (pas de ramassage des miettes) et l'on parle alors de code "unmanaged" et de composants "unmanaged".

Gestion de version

L'installation d'une version de la plate-forme .NET sur une machine conduit à la création d'un cache global pour les composants .NET (GAC : Global Assembly Cache). Ce cache est un répertoire (e.g. `c:\Windows\assembly`) dans lequel se trouvent tous les composants .NET destinés à être partagés entres les différentes applications s'exécutant sur cette machine. Il contient les composants de base de la plate-forme .NET de même que les composants déposés par les différents programmeurs. Le fait d'être placé dans le GAC plutôt que dans le répertoire d'une application donnée permet à un composant d'être partagé et donc d'être localisé de manière unique sans ambiguïté sur la machine. Ce répertoire spécifique peut aussi contenir plusieurs versions d'un même composant ce qui permet de résoudre les problèmes classiques liés à l'utilisation des DLL Windows standard où chaque nouvelle version d'une bibliothèque écrase l'ancienne conduisant à une compatibilité aléatoire des applications.

L'installation d'un composant dans le GAC se fait en plusieurs étapes :

– affectation d'un numéro de version selon un format bien défini (128 bits) `Major.Minor.Build.Revision` (par exemple (2.0.142.20) ;
– signature du composant afin de lui donner un nom unique (*strong name*) ;
– installation du composant dans le GAC ;
– mise au point de la sécurité.

6.2.5 Assemblage de composants .NET

Après une présentation très rapide des systèmes de type de .NET, cette section, reprise du document d'habilitation d'Antoine Beugnard [Beugnard 2005] [Beugnard 2006] est une critique détaillée des aspects "multi-langages" du framework. On compare dans un premier temps le comportement des langages à objets vis-à-vis de la liaison dynamique et dans un second temps la capacité de la plate-forme à avoir un comportement neutre pour la programmation par composition.

Le système de type CTS

L'un des objectifs de la plate-forme logicielle .NET est d'unifier les processus de développement et en particulier de résoudre le problème de l'hétérogénéité des langages de programmation. Pour atteindre cet objectif, la plate-forme .NET s'appuie sur un système de types commun. Ce système de types est capable d'exprimer la sémantique de la plupart des langages de programmation actuels et peut être vu comme un sur-ensemble des constructions que l'on retrouve dans la plupart des langages comme Java et C++. Cette spécification impose que chaque langage ciblant la plate-forme .NET doit respecter ce système de types. Ceci permet par exemple à un développeur C# d'utiliser une classe écrite par un développeur VB.NET, celle-ci héritant d'une classe Java (ou plutôt J# l'équivalent pour .NET). Chaque langage utilise ce système commun de type qui est appliqué à son propre modèle. Par exemple, chaque langage va utiliser le type `System.Int` afin que tous les int (ceux de C, de C++, de VB, de J# ou de C#) aient la même taille. Le code source est compilé vers un langage intermédiaire unique (MSIL : Microsoft Intermediate Language) pour le moniteur d'exécution .NET (run-time).

A l'exécution, le langage d'origine importe peu, et par exemple, des exceptions peuvent être signalées à partir de code écrit dans un langage et interceptées dans du code écrit dans un autre langage. Les opérations telles que la mise au point ou le profilage fonctionnent indépendamment des langages utilisés pour écrire le code. Les fournisseurs de bibliothèques n'ont ainsi plus besoin de créer des versions différentes pour chaque langage de programmation ou compilateur et les développeurs d'applications ne sont plus limités aux bibliothèques développées pour le langage de programmation qu'ils utilisent.

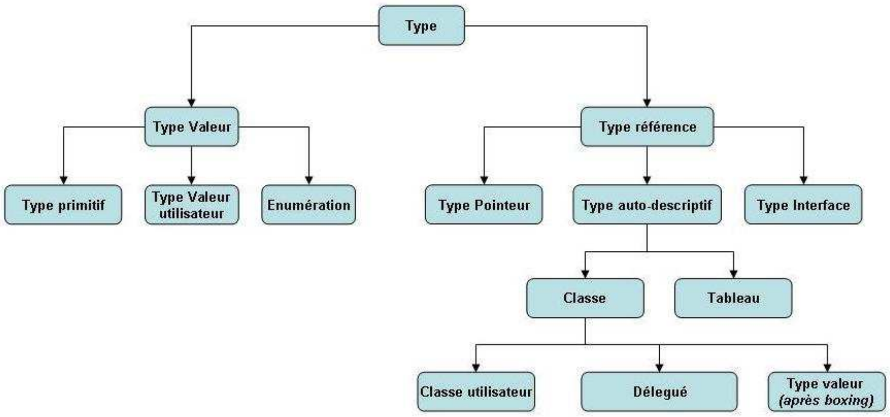

Figure 6.3 – Le système de types de la plate-forme .NET

Comme pour la plupart des langages de programmation à objets, le modèle de type de .NET distingue (voir figure 6.3) le type valeur (ce sont les types de base : `bool`, `char`, `int` et `float` mais aussi les types énumérés (`enum`) et les types structurés (`struct`)) et le type référence qui permet de désigner les objets (classes, interfaces, types délégués et tableaux). Les valeurs sont directement allouées dans la pile. La valeur d'un tel type contient directement les bits représentant la donnée valeur. Par exemple une valeur de type `int` correspondra à une case mémoire de 8 octets. Afin de garantir l'intégrité des types à l'exécution, le CLR garde la trace des types contenus dans la pile. Les références sont allouées dans le tas.

Un type valeur contient directement sa valeur tandis qu'une référence désigne l'objet qui contient la valeur. On retrouve ici la différence entre variable et pointeur dans certains langages de programmation. La conséquence de cette distinction est que deux variables sont nécessairement distinctes et que la modification de la valeur de l'une n'aura pas d'incidence sur l'autre tandis que deux références peuvent désigner le même objet et que la modification de la valeur de celui-ci sera aussi perçue par l'autre.

Par exemple au type `System.Boolean` du moniteur d'exécution .NET correspond le type C# `bool` et le type VB `Boolean`. Idem pour les types `System.Byte`, `System.Char`, `System.Object` qui sont respectivement couplés avec les types `byte`, `char` et `object` de C# et les types `Byte`, `Char` et `Object` de VB.

Dans .NET les deux types valeur et référence sont unifiés pour les différents types de base du langage. Il est ainsi possible de transformer un type valeur en un type de référence par le mécanisme appelé empaquetage ou objectification (*boxing*). Celui-ci transforme le type primitif en son homologue objet. L'opération inverse est appelée dépaquetage (*unboxing*) (voir figure 6.4)

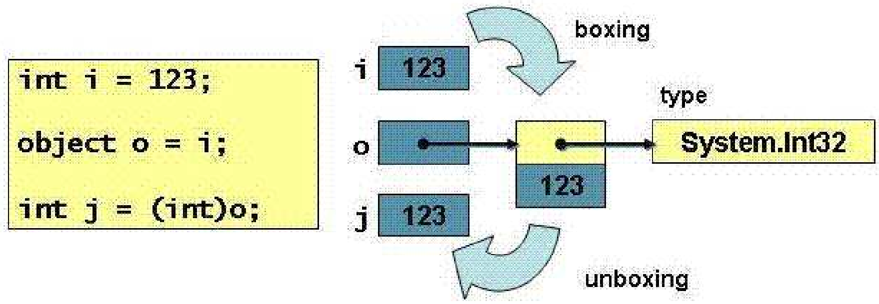

Figure 6.4 – Boxing et Unboxing

6.2.6 Le langage C#

Comme nous l'avons vu, la plate-forme .NET n'impose a priori aucun langage de programmation. En effet, toute compilation conduit à un ensemble de types suivant la spécification du système de types commun .NET (CTS) et le code se retrouve sous forme de langage intermédiaire (MSIL). Malgré cela un nouveau langage nommé C# [Gunnerson and Wienholt] [Ben Albahari et al. 2002] est apparu en même temps que la plate-forme. Ce langage a fait également l'objet d'une normalisation auprès de l'ECMA et de l'ISO. C'est le langage de référence de la plateforme .NET puisqu'il permet d'en manipuler tous les concepts : des instructions du MSIL au système de types .NET (classes, structures, énumérations, interfaces, délégués, tableaux, opérateurs, constructeurs, destructeurs, accesseurs (*setter*, *getter*), etc.). C'est pourquoi, en dépit de sa nouveauté, de nombreux développeurs apprennent ce langage pour programmer des applications .NET.

Pour un programmeur C++ ou Java, la lecture d'un code C# n'est pas déstabilisante au premier abord puisque le langage ressemble fortement à un mélange entre ces langages avec le meilleur de chacun et la simplification des fonctionnalités complexes comme l'héritage multiple en C++. Des exemples de code seront donnés dans les sections suivantes.

6.3 Applications distribuées

La plateforme .NET offre différents mécanismes permettant de distribuer les applications. Même si le framework est principalement orienté vers les services web, d'autres mécanismes sont disponibles pour le développeur. Par exemple .NET Remoting est une proposition beaucoup plus efficace que les services web pour la distribution d'application. Il est également possible d'utiliser la plateforme de service COM+ ou encore l'intergiciel orienté messages MSMQ. La sortie de la version 2.0 du framework a donné naissance à la nouvelle plateforme de distribution de Microsoft «Windows Communication Framework» (nom de code Indigo). Elle vise à unifier le développement et le déploiement des applications distribuées sur la plateforme .NET. Lorsque l'on développe un service distant, aucune technologie de distribution n'est à privilégier de manière systématique. Le choix entre les différentes technologies dépend des contraintes de chaque application (rapidité, standardisation, disponibilité, montée en charge) mais aussi des contraintes de fonctionnement (en intranet, volonté d'avoir des clients légers ou lourd). Dans ce chapitre nous présentons les principales technologies offertes dans .NET pour la communication.

6.3.1 Le canevas .NET Remoting

Equivalent de la technologie RMI pour la plateforme JAVA, le Remoting .NET permet le dialogue entre applications situées dans des domaines d'applications distincts, en particulier sur des stations différentes. L'architecture du Remoting .NET est conçue de manière extrêmement modulaire. Il est en effet possible d'insérer ses propres mécanismes à de nombreux endroits de la chaîne de traitement exécutée lors de l'envoi d'un message depuis le composant client vers le composant cible. Dans ce canevas, ont été séparés les aspects liés au transport de messages, à l'empaquetage des données et à l'appel à distance. Il est par exemple possible en implantant certaines classes du framework de faire communiquer un client .NET avec un composant RMI ou encore un composant CORBA. De même, le même code d'un composant pourra dialoguer sur un canal http en sérialisant ses données en XML ou alors dialoguer sur un canal TCP en sérialisant les données envoyées dans un format binaire plus compact.

En .NET Remoting, lors d'un dialogue client-serveur, des objets peuvent être passés en paramètres d'un appel de méthode. D'autres valeurs peuvent être retournées au client en réponse à sa requête. Il existe alors deux modes de passage de paramètres :

- **Passage par valeur** : dans ce mode c'est une copie intégrale de l'objet qui est envoyée au client. Si le client modifie cet objet une fois récupéré, les modifications ne seront pas propagées côté serveur. Ce type d'objet implémente l'interface `ISerializable`. On parle alors d'objet «sérialisable».

– **Passage par référence** : dans ce mode, c'est une référence vers un objet qui est envoyée au client. Une modification de l'objet soit du coté serveur, soit du coté client entraînera la modification de l'objet désigné dans le domaine d'application où réside l'objet. Ce type d'objet implémente la classe `MarshalByRefObject`. On parle alors de service .NET remoting ou encore de composant .NET Remoting.

Un composant .NET Remoting peut être activé de différentes manières. La création d'une instance du composant sur le serveur sera soit à la charge du serveur lui-même (on parle alors de service «activé par le serveur») soit par le client (on parle de service «activé par le client»). Le cycle de vie des services *Remoting* est également variable. On distingue ainsi trois types de service .NET Remoting :

– Les services **« activés par le serveur à requête unique »** (mode `SingleCall`) Pour chaque invocation d'une méthode du service par un client, une nouvelle instance spécifique du composant est créée pour répondre à la requête. C'est le serveur qui est en charge de l'instanciation du composant.
– Les services **« activés par le serveur et partagés»** (mode `Singleton`) La même instance du composant serveur est utilisée pour répondre aux différentes requêtes des différents clients. Ce mode permet un partage des données entre les clients. C'est le serveur qui est en charge de l'instanciation du composant.
– Les services **« activés par le client »** Dans ce mode c'est au client de demander explicitement la création d'une instance du composant dans l'application serveur par un appel explicite à l'opérateur `new`. Le composant serveur répondra alors aux différentes requêtes d'un même client

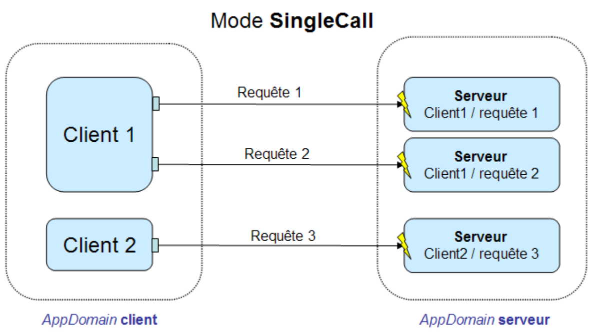

Figure 6.5 – Activation côté serveur - SingleCall

Les applications .NET Remoting dialoguent à travers un canal de communication sur lequel circulent des messages. Le framework offre la possibilité de définir par programmation le type de canal utilisé de même que la manière dont les messages sont encodés sur ce canal. Deux types de canaux sont fournis au développeur par le framework : le canal `HttpChannel` permettant d'établir un dialogue via le protocole HTTP et le canal `TCPChannel` permettant un dialogue sur une socket TCP. Au niveau de l'encodage, deux formateurs de messages sont également fournis. Le premier (`BinaryFormatter`) permet un encodage binaire des données sur le canal. L'autre (`SOAPFormatter`) permet un encodage des messages sous forme de document SOAP. L'utilisateur peut en outre implémenter ses propres canaux et ses propres formateurs en implémentant respectivement les interfaces

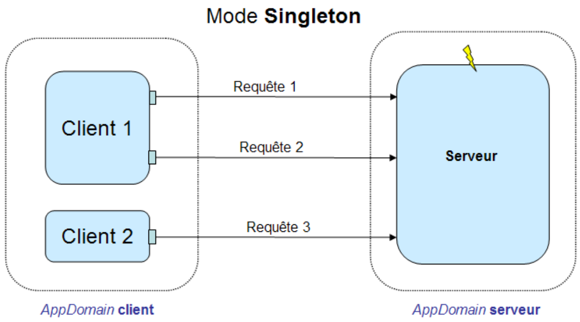

Figure 6.6 – Activation côté serveur - Singleton

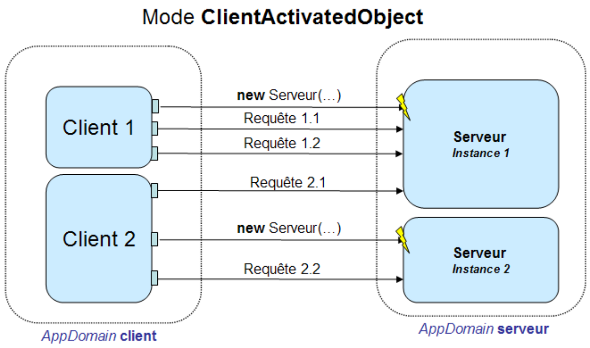

Figure 6.7 – Activation par le client

`IChannel` et `IRemotingFormatter`. Le type de dialogue entre un client et un serveur est défini par le couple (Canal, Formateur) qui est donné par le programmeur de l'application. Il est ainsi possible de faire circuler des messages SOAP sur un canal TCP ou encore encoder les messages sous un format binaire compact sur un canal HTTP.

Création d'un service .NET Remoting

Pour comprendre le processus de développement d'un service en .NET Remoting, prenons le cas d'un service très simple afin de s'abstraire des mécanismes spécifiques à l'implémentation du service. Considérons un service permettant d'effectuer deux opérations : la division réelle et la division entière. Dans un premier temps il s'agit de définir l'interface du service ainsi que les classes des données échangées. Ces classes de données exposées par le service (paramètres et valeur de retour) doivent pouvoir permettre à leurs instances d'être passées par valeur afin de pouvoir traverser les frontières du service. On parle alors d'objets « sérialisables ». En particulier les types de données simples fournis par le canevas (`int`, `double`, `string`) sont tous sérialisables. Dans le cas de la division entière nous désirons retourner au client un objet contenant le quotient et le reste de la division euclidienne de A par B. Pour cela nous marquons la classe `QuotientEtReste` par l'attribut `[Serializable]`.

Fichier EchoContract.cs

```
namespace Div.ServiceContract {
     [Serializable]
     public class QuotientEtReste {
          public int Quotient;
          public int Reste;
     }

     public interface IDivService {
          double Division(double A, double B);

          QuotientEtReste DivisionEntiere(int A, int B);
     }
}
```

Ces deux classes définissent la base du «contrat» entre le client et le service. Lorsque le service est activé côté serveur (mode `SingleCall` ou `Singleton`), seules ces classes seront partagées par les deux parties.

Pour créer le service lui même il faut ensuite coder une classe implémentant cette interface et la faire hériter de la classe du classe `System.MarshalByRefObjet` indiquant au runtime .NET que les instances de cette classe ne seront pas sérialisées par copie mais par référence.

```
namespace Div.ServiceImpl {
     using Div.ServiceContract;

     public class DivService: MarshalByRefObject,IDivService {
          string mName;
```

```
        static int mCounter=0;

        public DivService()     : this("Anonyme_"+mCounter++){}

        public DivService(string name) {
            this.mName=name;
            Console.WriteLine("new DivService("+mName+")");;
        }

        public double Division(double A, double B) {
            Console.WriteLine(mName+".Division("+A+","+B+")");
            return A/B;
        }

        public QuotientEtReste DivisionEntiere(int A, int B) {
            Console.WriteLine(mName+".DivisionEntiere("+A+","+B+")");
            QuotientEtReste RESULT=new QuotientEtReste();
            RESULT.Quotient=A/B;
            RESULT.Reste=A%B;
            return RESULT;
        }
    }
}
```

Notre service possède deux constructeurs. Le constructeur par défaut (sans paramètre) est obligatoire si l'on publie le service en mode « activé côté serveur ». En effet ce sera au runtime du *Remoting* d'instancier de manière transparente les instances de ce service. Nous définissons également un constructeur prenant une chaîne de caractère en paramètre et ce afin que le service soit identifié lorsque nous le testerons. Ce constructeur nous permettra de voir les différences à l'exécution entre les différents types de cycle de vie du service (lors de l'instanciation).

Il reste maintenant à créer l'application qui hébergera notre service. Nous allons créer deux applications console. L'une hébergera le service sous les deux modes « activé côté serveur » SingleCall et Singleton. La deuxième application hébergera les instances du service en mode «activé par le client». Il existe deux manières de spécifier au framework ces paramètres : soit par programmation soit à l'aide de fichier de configuration. L'utilisation de fichier de configuration est la méthode à privilégier puisqu'elle permet de modifier les paramètres de déploiement sans recompiler l'application. Nous présentons ici l'approche par programmation. La deuxième approche par fichier de configuration sera présentée dans le chapitre suivant .

Application hôte N ° 1 en mode « activé par le serveur »

```
namespace Div.Host.SAO {
    using Div.ServiceContract;
    using Div.ServiceImpl;

    class DivHost {
        [STAThread]
```

```
        static void Main(string[] args) {
           // création et enregistrement du canal HTTP 8081
           ChannelServices.RegisterChannel(new HttpChannel(8081));
           RemotingConfiguration.RegisterWellKnownServiceType(typeof(DivService),
                   "division/singleton", WellKnownObjectMode.Singleton);
           RemotingConfiguration.RegisterWellKnownServiceType(typeof(DivService),
                   "division/singlecall", WellKnownObjectMode.SingleCall);
           RemotingConfiguration.RegisterActivatedServiceType(
                   typeof(Div.ServiceImpl.DivService));
           Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
           Console.ReadLine();
        }
    }
}
```

Tout d'abord nous indiquons au *Remoting* le canal de transport à utiliser. Nous créons une instance de canal HTTP sur le port 8081 que nous enregistrons auprès du gestionnaire de canal `ChannelServices`. Dans un deuxième temps nous devons enregistrer notre service auprès du *Remoting.* C'est le rôle de la méthode `RegisterWellKnonwnServiceType` pour les services « activés par le serveur ». Cette méthode prend en paramètre la classe des futures instances du service, une adresse URI relative au canal de transport utilisé, ainsi que le mode de publication choisi pour le service. Nous appelons cette méthode une première fois pour publier le service en mode Singleton et une seconde fois pour publier le service en mode SingleCall.

Voyons à présent comment la publication du service s'effectue en mode « activé par le client »

Application hôte N ˚ 2 en mode « activé par le client »

```
namespace Div.CAO.Host {
    using Div.ServiceContract;
    using Div.ServiceImpl;

    class DivHost {
        [STAThread]
        static void Main(string[] args) {
            // création et enregistrement du canal HTTP 8082
            ChannelServices.RegisterChannel(new HttpChannel(8082));
            RemotingConfiguration.RegisterActivatedServiceType
                    (typeof(Div.ServiceImpl.DivService));
            Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
            Console.ReadLine();
        }
    }
}
```

La différence réside dans la manière de publier le service. Cette fois c'est la méthode `RegisterActivatedServiceType` qui est utilisée. Elle indique au *Remoting* qu'il s'agit d'une classe de service à enregistrer dans le mode « activé par le client » et ne prend en paramètre que la classe du service.

Création d'un client .NET Remoting

Voyons à présent comment dialoguer avec nos services dans les différents modes. Le premier exemple montre comment dialoguer avec les deux services publiés en mode Singleton et SingleCall. Afin d'obtenir une référence distante sur le service, le client utilise la méthode `GetObject` de la classe `Activator`, en lui passant en paramètres l'interface et l'adresse URI du service distant. Grâce à ces paramètres le *Remoting* est en mesure de localiser l'application hébergeant le service et de créer une référence transparente pour le client. Une fois la référence obtenue le client peut invoquer les méthodes du service comme ci celui-ci était présent localement. A chaque invocation de méthode l'appel sera transporté jusqu'au serveur qui déterminera l'instance qui devra répondre à l'appel client. S'il s'agit du composant `SingleCall` le *Remoting* créera une nouvelle instance du service sur le serveur à chaque appel client. Dans le cas du service `Singleton` le *Remoting* créera une instance du service lors du premier appel de méthode du client

Application cliente N ° 1 (dialoguant en mode SingleCall et Singleton)

```
class Client {
    [STAThread]
    static void Main(string[] args) {
        IDivService service;
        service=(IDivService)
                    Activator.GetObject(typeof(Div.ServiceContract.IDivService),
                                        "http://localhost:8081/division/singleton");
        Test("1)Mode Singleton",service);

        service=(IDivService)
                    Activator.GetObject(typeof(Div.ServiceContract.IDivService),
                                        "http://localhost:8081/division/singlecall");
        Test("2)MODE SingleCall",service);
    }

    static void Test(string testName, IDivService service) {
        Console.WriteLine(testName);

        double RESULT1=service.Division(100.0,70.0);
        Console.WriteLine("100.0 / 70.0="+ RESULT1);

        QuotientEtReste QR=service.DivisionEntiere(100,70);
        Console.WriteLine("100 / 70 -> Q="+ QR.Quotient+" R="+QR.Reste);

        Console.WriteLine("Appuyez sur [Entrée] pour continuer.");
        Console.ReadLine();
    }
}
```

En mode « activé par le client », l'obtention de la référence distante sur le service est assez différente. Cette fois c'est la méthode `CreateInstance` de la classe `Activator` qui est utilisée. Celle-ci prend en paramètre l'url (dans un tableau d'objet) de l'application qui héberge les instances du service. C'est à ce moment que le serveur crée l'instance qui servira le client. Le client peut alors appeler différentes méthodes sur cette même instance du service. Le *Remoting* utilise un système paramétrable de « bail » permettant de contrôler le cycle de vie de cette instance. Lorsque le bail expire après un certain temps, le serveur détruira l'instance. Chaque invocation du client sur le service renouvellera ce bail pour une durée donnée. Ce mécanisme de bail existe également dans les mode SingleCall et Singleton.

Application cliente N ° 21 (dialoguant en mode SingleCall et Singleton)

```
[STAThread]
static void Main(string[] args) {
    object[] url   = { new UrlAttribute("http://localhost:8082")};
    object[] param = { "Icar"};
    IDivService service=(DivService)
    Activator.CreateInstance(typeof(DivService),param,url);
    Test("1)MODE CAO",service);
}
```

Evolution future

Un aspect à prendre en compte lorsque l'on utilise le .NET Remoting est le couplage qu'il créé entre le client et le serveur. En effet ces deux entités doivent partager un même *assembly* .NET : celui contenant l'interface du service. Les changements de version du service impactant l'interface implique également la modification du client. Dans le nouveau framework Indigo, le découplage entre consommateur et fournisseur de service devient réel puisque les deux entités partageront non plus une interface, mais plutôt un contrat de service dans un format standard tel que WSDL.

6.3.2 Les services Web en .NET

Les services web ayant déjà été présenté dans un chapitre précédent (cf. chapitre **??**, cette section présente uniquement leur mise en œuvre dans le cadre de la plate-forme .NET à l'aide de ASP.NET (nouvelle version d'ASP - Active Server Pages). ASP.NET permet d'intégrer dans une page html du code .NET. Ce code est exécuté par le serveur Web lorsque la page est demandée. Il va de soit, que le serveur Web doit être, pour cela, capable d'exécuter du code .NET. Deux suffixes de fichiers sont utilisés :

- *"aspx"* pour les pages html qui contiennent du code .NET devant être interprétée par le serveur web. Les pages sont envoyées au client dans le format HTML.
- *"asmx"* pour des pages qui contiennent le code du service web. Ce service est exécuté par le serveur web. Les pages sont envoyées au client dans le format SOAP.

Cette section présente essentiellement la mise en œuvre des services web dans l'architecture .NET.

Un modèle de développement web unifié, ASP.NET

JSP (Java Server Pages) a déjà été une réponse à la proposition ASP de Microsoft. Aujourd'hui une nouvelle étape est franchie. Les applications web conçues sur la plate-forme .NET partagent avec toutes les autres applications .NET un même modèle d'application.

Dans ce modèle, une application web est un ensemble d'URL dont la racine est une URL de base. Il englobe donc les applications web générant des pages pour un navigateur et les services web. Ce modèle de programmation d'applications web est une évolution d'ASP (Active Server Pages). ASP.NET utilise les avantages du Common Language Runtime et de la plate-forme .NET pour fournir un environnement fiable, robuste et évolutif aux applications web. ASP.NET bénéficie aussi des avantages du modèle d'assemblage du Common Language Runtime pour simplifier le développement d'applications. Il fournit en outre, des services pour faciliter le développement d'applications (tels que les services de gestion d'état) et des modèles de programmation haut niveau (tels que ASP+ Web Forms et ASP.NET Services Web).

L'environnement d'exécution HTTP dans la plate-forme .NET est au cœur d'ASP.NET pour le traitement des requêtes HTTP. Cette machine virtuelle haute performance est basée sur une architecture de bas niveau similaire à l'architecture ISAPI traditionnellement fournie par Microsoft Internet Information Services (IIS). Cet environnement d'exécution HTTP est constitué de " managed code " exécuté dans un processus spécifique. Il est responsable du traitement de toutes les requêtes HTTP, de la résolution de l'URL de chaque requête vers une application et de la distribution de la requête à l'application pour traitement complémentaire. ASP.NET utilise le modèle de déploiement de la plate-forme .NET basé sur les assemblages et permet la mise à jour des applications à chaud. Les administrateurs n'ont pas besoin d'arrêter le serveur Web ou l'application pour mettre à jour les fichiers d'application. Ceux-ci ne sont jamais verrouillés de façon à pouvoir être remplacés même lors de l'exécution de l'application. Lorsque les fichiers sont mis à jour, le système bascule vers la nouvelle version. Le système détecte les modifications de fichiers, lance une nouvelle instance de l'application avec le nouveau code et commence à router les requêtes entrantes vers cette application. Lorsque toutes les requêtes en suspens traitées par l'instance de l'application existante ont été traitées, cette instance est arrêtée.

Le traitement des requêtes HTTP

Au sein d'une application, les requêtes HTTP sont routées via un pipeline de modules HTTP et en dernier lieu, par un gestionnaire de requêtes. Les modules et gestionnaires de requêtes HTTP sont simplement des classes gérées qui implantent des interfaces spécifiques définies par ASP.NET. L'architecture en pipeline facilite grandement l'ajout de services aux applications : il suffit de fournir un module HTTP. Par exemple, la sécurité, la gestion d'état et des traces sont implantées en tant que modules HTTP. Les modèles de programmation de haut niveau, tels que les services web et Web Forms, sont généralement implantés comme gestionnaires de requêtes. Une application peut être associée à plusieurs gestionnaires de requêtes (un par URL), mais toutes les requêtes HTTP sont routées au travers du même pipeline. Le web constitue un modèle fondamentalement sans état et sans corrélation entre les requêtes HTTP. Ceci peut rendre l'écriture d'applications web difficile, car les applications nécessitent habituellement de conserver leur état entre plusieurs requêtes. ASP.NET améliore les services de gestion d'état introduits par ASP pour fournir trois types d'état aux applications web : application, session et utilisateur. L'état " application ", tout comme pour ASP, est spécifique à une instance de l'application et n'est pas persistant. L'état " session " est spécifique à une session utilisateur de l'application. Contrairement à l'état session de ASP, l'état session de ASP.NET est stocké dans un processus distinct et peut même être configuré pour être stocké sur un ordinateur distinct. Ceci rend l'état session utilisable lorsqu'une application est déployée sur un ensemble de serveurs web. L'état utilisateur est similaire à l'état session mais en général, il n'expire jamais et est persistant. L'état utilisateur est donc utile pour stocker les préférences de l'utilisateur et d'autres informations de personnalisation. Tous les services de gestion d'état sont implantés en tant que modules HTTP et peuvent en conséquence être facilement ajoutés ou supprimés du pipeline d'une application. Si des services de gestion d'état autres que ceux fournis par ASP.NET sont nécessaires, ils peuvent être fournis par un module tiers.

ASP.NET fournit aussi des services de cache pour améliorer les performances. Un cache de sortie permet d'enregistrer les pages générées en entier et un cache partiel permet de stocker des fragments de pages. Des classes sont fournies pour permettre aux applications, aux modules HTTP et aux gestionnaires de requêtes de stocker à la demande des objets arbitraires dans le cache.

Création d'un Service Web XML en ASP.NET

Le framework .NET offre de nombreux mécanismes simplifiant au maximum la création de service web. En particulier tous les aspects liés à SOAP et WSDL peuvent devenir complètement transparents pour le développeur. De plus s'utilisation de l'environnement de développement de Microsoft Visual Studio pousse encore plus loin la simplicité de création et de déploiement de service web en quelques click. Nous montrons cependant ici comment se passent les étapes de développement sans l'aide de cet IDE afin de ne pas trop masquer les mécanismes impliqués.

Les étapes du processus de développement et de déploiement d'un service web en .NET sont les suivantes :

- Implémenter la classe du service en le faisant hériter de la classe System.Web.Services.WebService
- Décorer l'implémentation en faisant précéder chaque déclaration de méthode devant être exposée par l'attribut [WebMethod]
- Créer un répertoire virtuel sur le serveur (exemple IIS).
- Compiler le service (dans le répertoire virtuel)
- Déployer le service sur le serveur web

Tout d'abord il s'agit d'écrire le service lui même. Reprenons l'exemple de la section précédente sur le service de division. Voici le code C# permettant d'en faire un service web

DivWebService.cs Le service web

```
using System; using System.Web.Services;

namespace Div {
    public class DivWebService: WebService {
        [WebMethod]
        public double Division(double A, double B) {
            return A/B;
        }
```

```
        public void UneMethodeNonPubliee() {
        }
    }
}
```

Pour qu'une classe puisse être exposée via un service web, celle-ci doit hériter de la classe du framework `System.Web.WebServices.WebService`. De plus chaque méthode qui devra être publiée devra être précédée de l'attribut `[WebMethod]` Nous pouvons remarquer dans notre exemple que la méthode `UneMethodeNonPubliee` ne sera pas publiée dans l'interface du service web car sa signature n'est pas précédée de l'attribut [WebMethod]. Ainsi cette méthode ne sera pas accessible aux clients. Nous compilerons le code dans une bibliothèque `DivWS.dll` grace à la ligne de commande suivante :

```
csc.exe  /out:bin\DIVWS.dll /t:library DivService.cs
```

Pour pouvoir tester notre service une page `div.asmx` utilisant la notion de *"code behind"* doit être mise en œuvre. Nous spécifions dans son entête qu'il s'agit d'un web service ainsi que le nom du fichier source C# correspondant à cette classe à des fins de débogage.

Le fichier div.asmx

```
<%@ WebService Class="Div.DivWebService"%>
```

A présent, il nous faut déployer notre service au sein d'une application web hébergée par le serveur IIS de Microsoft. Pour cela nous pouvons utiliser la console d'administration de Windows afin de créer un nouvelle application. Nous la nommerons par exemple `icar`. Dès lors l'application aura comme racine distante l'url `http ://localhost/icar`. Créons un répertoire bin dans lequel nous plaçons la bibliothèque compilée précédemment. La page aspx du service sera placé à la racine de l'application. Le service est alors opérationnel. Il peut immédiatement être testé en utilisant un navigateur web quelconque et en allant à l'adresse `http ://localhost/icar/div.asmx ?WSDL`. Cette page permet de visualiser le contrat WSDL de notre service. Pour tester le service lui même il faut se rendre à l'adresse `http ://localhost/icar/div.asmx`. .NET génére automatiquement un formulaire web qui permet d'appeler les différentes WebMethod du service web.

Utilisation de Services Web

Nous allons voir a présent comment consommer un service web. Nous prendrons comme exemple un service Web assez pratique : celui offert par Google©. Avant toute chose il faut disposer de la description WSDL de ce service. Celle-ci se trouve à l'adresse `http ://api.google.com/GoogleSearch.wsdl`

Le framework .NET fournit un utilitaire `wsdl.exe` permettant à partir de cette description de générer les classes proxy permettant de consommer le service web. Voici la ligne de commande permettant d'obtenir le code C# du proxy :

```
wsdl.exe /o:google.cs http://api.google.com/GoogleSearch.wsdl
```

Le fichier généré `google.cs` contient différentes classes : celles du proxy vers le service Web et les classes de données manipulées par le service.

Voici à présent un programme client simple permettant d'effectuer une correction d'un mot mal orthographié.

```
class TestGoogle {
     static void Main(string[] args) {
          string GOOGLE_KEY="-xxxxx-";
          string phrase="Intergicielle";
          GoogleSearchService service=new GoogleSearchService();
          string rep=service.doSpellingSuggestion(GOOGLE_KEY,phrase);
          Console.WriteLine(‘‘Correction: ‘‘+rep);
          Console.ReadLine();
     }
}
```

Le programme une fois exécuté affichera le texte corrigé « Intergiciel » Dans notre exemple nous utilisons l'opération doSpellingSuggestion du Web Service en lui donnant comme entrée d'une part une clé d'authentification (fournie par Google© permettant d'accéder au service web après enregistrement en ligne) et en deuxième paramètre la phrase à corriger.

6.3.3 L'Asynchronisme

Il existe deux possibilités pour effectuer des appels asynchrones de méthode dans .NET. La première utilise les appels asynchrones de méthode disponible dans la plate-forme .NET et la seconde repose sur les files de messages disponible dans la plate-forme Windows. Nous détaillons ci-après ces deux possibilités.

Appel asynchrone

Pour pouvoir effectuer en .NET des appels asynchrones de méthode, il est nécessaire d'utiliser le principe des délégués qui fournit pour chacun d'entre eux deux méthodes : BeginInvoke () et EndInvoke (). BeginInvoke() permet d'effectuer l'appel asynchrone et EndInvoke () permet de récupérer les résultats de cet appel. La synchronisation entre l'appel et la lecture du résultat est à la charge du programmeur qui peut :

– effectuer une attente bloquante du résultat en appelant la méthode EndInvoke (),
– être prévenu de la disponibilité du résultat en transmettant lors de l'appel, le code à exécuter pour récupérer le résultat.

Voici un exemple illustrant ces deux possibilités :

```
class Exemple {
    public delegate int Prototype(int a, int b);

    static int PGCD(int a, int b) {
        // calcul du pgcd
        return c;
    }
```

```
    static void FinCalcul(IAsyncResult A) {
        // récupération du résultat
        Prototype p = (Prototype) (IAsyncResult A).AsyncDelegate;
        int c = p.EndInvoke(A);
    }

    static void Main () {
        // création de l'instance du délégué
        Prototype p = New Prototype (PGCD);
        int c;

        // appel synchrone du délégué
        c = p (10, 25);

        // appel asynchrone du délégué
        IAsyncResult A = p.BeginInvoke (10, 25, null, null);

        // lecture bloquante du résultat
        c = p.EndInvoke(A);

        // appel asynchrone du délégué en transmettant la procédure à
        // exécuter à la fin de l'appel (méthode de finalisation)
        IAsyncResult A = p.BeginInvoke (10, 25, new AsyncCallBack(FinCalcul), null);
    }
}
```

Le dernier paramètre, non utilisé dans l'exemple ci-dessus, de la procédure `BeginInvoke` permet de transmettre la référence d'un objet utilisable simultanément dans le *thread* qui déclenche l'appel asynchrone et dans celui qui exécute la méthode de finalisation.

Les Files de message

Contrairement à l'appel asynchrone de .NET, le client et le serveur ne sont plus obligés d'être connectés et les requêtes des clients sont stockées dans une file sous forme de messages. Il est par contre nécessaire d'installer sur les stations hôtes le mécanisme de gestion de file de messages (Microsoft Message Queue - MSMQ).

6.4 Composants et services techniques

Les services techniques, de plus en plus nombreux, sont destinésà simplifier la tâche des programmeurs en leur évitant d'avoir à développer tout ce qui peut petre mis en commun. Dans la plate-forme .Net, ces services repose sur Enterprise Services qui est la nouvelle appellation de COM+ et ils sont présent dans la plate-forme support (Windows) que .NET utilise. Les principaux services techniques permettent l'accès aux données, la gestion de la sécurité applicative, la mise en œuvre de transactions distribuées. Ce sont ces services qui nous allons détailler dans cette section.

6.4.1 Accès aux données

Pratiquement toutes les applications utilisent un système de persistance au sens large du terme. Dans la plate-forme .NET, cette persistance peut être réalisée de plusieurs manières :

– par sérialisation de l'état de l'objet sous forme de flots d'octets ou en XML ;
– par sauvegarde de la valeur de l'état de l'objet dans une base de données relationnelle.

L'appelation ADO (*ActiveX Data Object*) englobe à la fois l'ensemble des classes du framework .NET utilisées pour manipuler les données contenues dans les SGBD relationnels et la philosophie d'utilisation de ces classes.

ADO permet la connexion avec une base de données en mode :

– connecté : si l'application charge des données de la base au fur et à mesure de ces besoins et envoie les modifications à ces données dès qu'elles surviennent.
– déconnecté : si l'application charge des données de la base, les exploite hors ligne et les met à jour si nécessaire ultérieurement. Elle peut aussi récupérer ses données de manière externe à la base, par exemple depuis un fichier XML, les manipuler et les sauvegarder dans la base ; ou réciproquement.

La faiblesse du mode connecté réside dans les multiples accès qui sont effectués sur la base de données, générant pas là même de très nombreux accès réseaux. Le mode déconnecté n'a pas cet inconvénient mais est beaucoup plus consommateur de mémoire utilisée pour stocker de manière temporaire les données nécessaires.

Parmi toutes les classes qu'ADO regroupe, DataSet joue un rôle prépondérant. Les instances de cette classe représentent une partie de la base. Pour chacun de ces tables, l'instance contient des lignes extraites et éventuellement les contraintes d'intégrités associées à la table. Cette classe est à utiliser pour tout travail en mode déconnecté. Cette classe permet aussi le passage d'un DataSet à un document XML ou réciproquement.

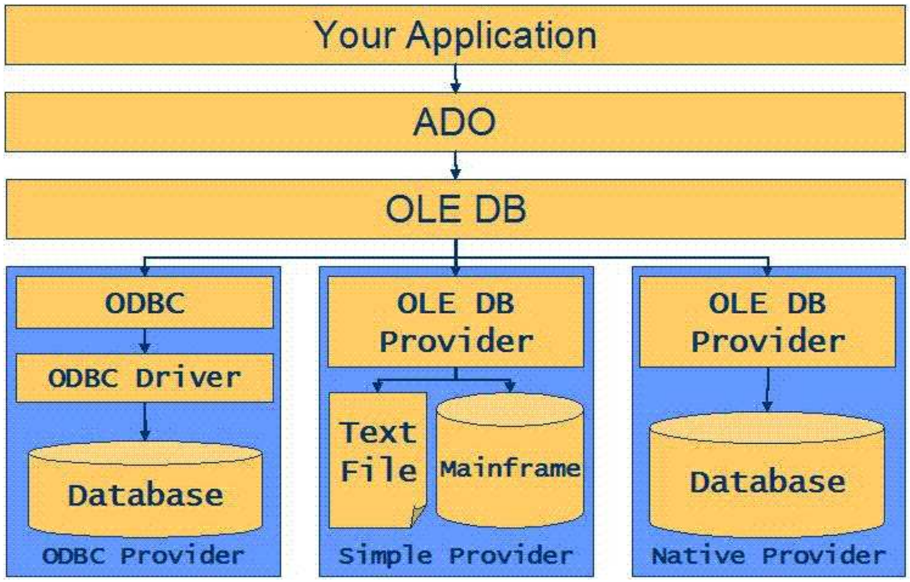

Figure 6.8 – Accès aux données

Exemple de manipulation d'un DataSet :

```
// créer le DataSet et définir les tables
DataSet dataset = new DataSet();
dataset.DataSetName = ‘‘BookAuthors’’;

DataTable authors = new DataTable(‘‘Author’’);
DataTable books = new DataTable(‘‘Book’’);

// définir les colonnes et les clés
DataColumn id = authors.Columns.Add("ID", typeof(Int32));
id.AutoIncrement = true;
authors.PrimaryKey = new DataColumn[] {id};

DataColumn name = new authors.Columns.Add("Name",typeof(String));

DataColumn isbn = books.Columns.Add("ISBN", typeof(String));
books.PrimaryKey = new DataColumn[] {isbn};

DataColumn title = books.Columns.Add("Title", typeof(String));
DataColumn authid = books.Columns.Add("AuthID",typeof(Int32));
DataColumn[] foreignkey = new DataColumn[] {authid};

// ajouter les tables au DataSet
dataset.Tables.Add (authors);
dataset.Tables.Add (books);

// Ajouter des données et sauvegarder le DataSet
DataRow shkspr = authors.NewRow();
shkspr["Name"] = "William Shakespeare";
authors.Rows.Add(shkspr);

DataRelation bookauth = new DataRelation("BookAuthors",
                authors.PrimaryKey, foreignkey);
dataset.Relations.Add (bookauth);

DataRow row = books.NewRow();
row["AuthID"] = shkspr["ID"];
row["ISBN"] = "1000-XYZ";
row["Title"] = "MacBeth";
books.Rows.Add(row);

dataset.AcceptChanges();
```

6.4.2 Sécurité

Les services de sécurité intégrés à la plate-forme permettent de s'assurer que seuls les utilisateurs autorisés accèdent aux ressources d'un ordinateur et seul le code ayant acquis les droits suffisants peut exécuter les actions nécessitant ces droits. Ceci améliore la sécurité globale du système ainsi que sa fiabilité. L'environnement d'exécution étant utilisé pour charger le code, créer les objets et effectuer les appels de méthodes, il peut contrôler la sécurité et assurer l'application des stratégies de sécurité lorsque du " managed code " est chargé et exécuté. La plate-forme .NET propose une sécurité d'accès au code, mais également une sécurité basée sur les rôles.

Grâce à la sécurité d'accès au code, les développeurs peuvent spécifier les autorisations requises par leur code pour effectuer une tâche. Le code peut par exemple nécessiter une autorisation d'écriture sur un fichier ou d'accès aux variables d'environnement. Ces informations, ainsi que celles concernant l'identité du code, sont stockées au niveau du composant. Lors du chargement et des appels de méthodes, l'environnement d'exécution vérifie que le code peut se voir accorder les autorisations qu'il a demandées. Si ce n'est pas le cas, une violation de sécurité est déclarée sous la forme du signalement d'une exception. Les stratégies d'attribution d'autorisations (connues sous le nom de trust policies) sont établies par les administrateurs des systèmes et sont basées sur des informations relatives au code telles que son origine ou les demandes d'autorisation de l'assemblage. Les développeurs peuvent également spécifier explicitement les autorisations qu'ils ne souhaitent pas, pour éviter toute utilisation malveillante du code. Il est possible de programmer des contrôles de sécurité dynamiques si les autorisations requises dépendent d'informations inconnues avant l'exécution.

Outre la sécurité d'accès au code, l'environnement d'exécution prend en charge la sécurité basée sur les rôles. La sécurité basée sur les rôles est conçue sur le même modèle d'autorisations que la sécurité d'accès au code, à ceci près que les autorisations sont basées sur l'identité de l'utilisateur et non sur l'identité du code. Les rôles représentent des catégories d'utilisateurs et sont définis lors du développement ou du déploiement. Des stratégies d'attribution d'autorisations sont assignées à chaque rôle. Lors de l'exécution, l'identité de l'utilisateur pour lequel le code est exécuté est déterminée. L'environnement d'exécution détermine les rôles auxquels l'utilisateur appartient et accorde ensuite les autorisations en fonction de ces rôles.

Sécurité d'accès utilisateur La sécurité d'accès utilisateur sert à déterminer les droits associés à l'identité en cours. Il est possible de vérifier ce qu'un appelant a le droit de faire de nombreuses façons. Dans les applications avec une interface utilisateur, il est possible d'emprunter l'identité de l'appelant, mais il est aussi possible que le code utilise un compte de " service " spécifique par modification de l'identité en cours d'exécution (modification de l'objet `Principal`).

Les droits de l'utilisateur sur l'environnement et la plate-forme sont généralement contrôlés à l'aide de listes de contrôle d'accès qui sont comparées à l'identité Windows du thread en cours.

La technologie .NET fournit une infrastructure complète et extensible pour la gestion de la sécurité d'accès utilisateur y compris des identités, des permissions et la notion d'objet Principal et de rôles. Pour garantir que les utilisateurs appartenant à un certain rôle appellent une méthode donnée, il suffit de définir un attribut sur la méthode de classe, comme indiqué dans le code suivant :

```
[PrincipalPermission(SecurityAction.Demand, role="Managers")]
public void managersReservedMethod () {
      // Ce code ne s'exécutera pas si le Principal associé au
      // thread n'a pas ''Managers'' lorsque IsInRole est appelé
```

```
}
```

Si le composant est basé sur un déploiement dans Enterprise Services et authentifie les utilisateurs via Windows, il est aussi possible d'utiliser la gestion des rôles Enterprise Services comme indiqué dans le code ci-dessous.

```
[SecurityRole("Managers")]
public void managersReservedMethod () {
     // Ce code ne s'exécutera pas si le Principal associé à
     // l'utilisateur n'est pas dans le groupe ''Managers''
}
```

Sécurité d'accès au code La sécurité d'accès au code permet de définir les droits de l'assembly, mais il est également possible de décider du lancement ou non du code, en fonction du code essayant d'y accéder. Par exemple, cela permet d'empêcher que des objets soient appelés à partir de scripts lancés de manière inopportune par une personne disposant de privilèges suffisants. Vous pouvez contrôler l'accès au code à l'aide des éléments suivants :

- le répertoire d'installation de l'application ;
- le hachage cryptographique de l'assembly ;
- la signature numérique de l'éditeur de l'assembly ;
- le site dont provient l'assembly ;
- le nom fort cryptographique de l'assembly ;
- l'URL dont provient l'assembly ;
- la zone dont provient l'assembly.

Les politiques de sécurité peuvent être appliquées à l'entreprise, l'ordinateur, l'utilisateur et l'application. Les zones définies par la technologie .NET sont les suivantes : Internet, intranet, Poste de travail, NoZone, de confiance et non fiable.

Implémentation de contrôles d'autorisation complexes Dans certains cas, une application doit effectuer des contrôles d'autorisation complexes reposant sur une politique d'autorisation qui requiert des combinaisons de permissions combinant simultanément la sécurité utilisateur, la sécurité d'accès au code mais aussi des permissions liées aux paramètres de l'appel. Ces types de contrôles d'autorisation sont à effectuer dans le code de l'application sous forme de programmes et requièrent généralement un développement conséquent.

Communication sécurisée Outre l'authentification des utilisateurs et des requêtes d'autorisation, il est aussi nécessaire de s'assurer que la communication entre les différents éléments d'une application soit sécurisée afin d'empêcher toute attaque de “reniflage” ou de falsification lors de la transmission ou du placement des données dans une file d'attente.

Une communication sécurisée implique la sécurisation des transferts de données entre les composants distants. Pour ce faire, .NET, comme la plupart des plates-formes, offre les options suivantes pour :

- **Sécuriser l'ensemble du canal :**

  - Protocole SSL (Secure Sockets Layer) recommandé pour les communications avec le protocole HTTP. Il s'agit d'un standard très répandu et il est généralement permis d'ouvrir des ports SSL sur les pare-feu.
  - IPSec. Ce mécanisme constitue un choix judicieux lorsque les deux points d'entrée de la communication sont bien connus et sous le contrôle de la même entité administrative.
  - Réseau privé virtuel (VPN) qui permet d'établir un transport IP point à point sur Internet (ou d'autres réseaux).
- **Sécurisation des données :**
  - Signature d'un message. Cela permet de détecter les messages falsifiés. Les signatures peuvent être utilisées pour l'authentification au sein d'un même processus.
  - Cryptage de l'ensemble d'un message. De cette façon, l'ensemble du message devient illisible en cas de compromission des paquets du réseau. Le cryptage d'un message à l'aide d'algorithmes appropriés permet également de détecter les messages falsifiés.
  - Cryptage des parties sensibles d'un message si seulement des parties du message doivent être protégées.

6.4.3 Transaction

Pour gérer des transactions distribuées avec plusieurs bases de données, il n'est pas possible dans la technologie .NET d'utiliser ADO.NET 6.4.1 et il faut, comme pour la sécurité utiliser le gestionnaire de transactions disponible dans Enterprises Services (DTC : Distributed Transaction Coordinator). Pour illustrer la mise en œuvre des transactions nous allons prendre l'exemple très classique du transfert bancaire entre deux comptes. L'architecture de l'application est décrite dans la figure 6.9.

Ci-dessous, la partie essentielle du source de l'application :

```
public class Programme {
    static void Main () {
        TdF transfert = new TdF ();
        tranfert.Transfert (``Michel'', ``David'', 1960, 1979, 1000) ;
        // transfert 1000 euros du compte de Michel au compte de David
    }
}

[Transaction(TransactionOption.RequiresNew)]
public class TdF : ServicedComponent {
    // hérite d'un objet Services Interprises
    public void Transfert (String ctx_client1, ctx_client2,
                           int ID_client1, ID_client2, montant) {
        Compte client1 = new Compte (); Compte client2 = new Compte ();
        // création de la connection à la base de données
        client1.Connect (ctx_client1); client2.Connect (ctx_client2);
        // connection à la base de données pour lecture du solde du compte
        int solde_client1 = client1.ConsulteCompte (ID_client1);
        int solde_client2 = client2.ConsulteCompte (ID_client2);
        // modification locale des montants des comptes
```

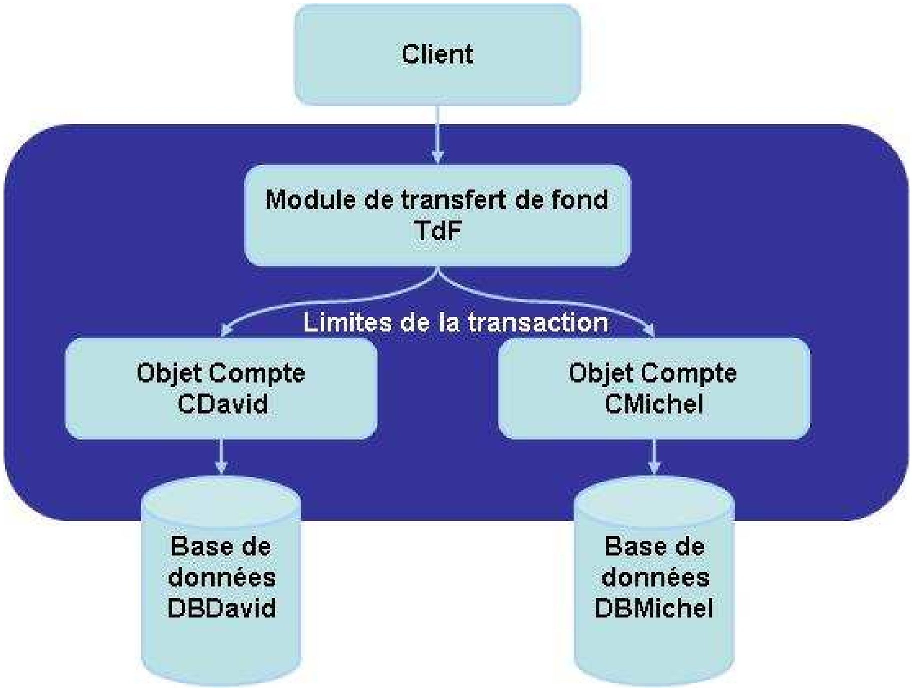

Figure 6.9 – Application de transfert entre comptes

```
        solde_client1 -= montant; solde_client2 += montant;
        // connection à la base de données pour mise à jour du compte
        client1.MetAJour (ID_client1, solde_client1);
        client2.MetAJour (ID_client2, solde_client2);
    }
}

[Transaction(TransactionOption.Required)
public classe Compte : ServiceComponent {
    public void Connect (String ctx_client) {
        // ouverture de la connexion à la base de donnée
    }
    public int ConsulteCompte (int ID_client) {
        // connexion à la base de données pour lire le solde du compte
    }
    [autocomplete(true)]
    public void MetAJour (int ID_Client, int montant) {
        // connexion à la base de données pour modification du solde du compte
        // fermeture de la connexion à la base de données
    }
}
```

Il n'est pas nécessaire d'insérer dans le code du programme une manipulation explicite des transactions, celle-ci se faisant par l'utilisation de l'attribut Transaction sur une classe (dans l'exemple sur les classe TdF et Compte). Un composant peut adopter cinq comportements différents face à la création ou à l'utilisation d'une transaction. Ils sont décrits par les cinq options de l'attribut Transaction suivant :

- Disabled : les instances de ce composant n'utilisent pas de transaction et ne doivent pas être appelées pendant une transaction.
- NotSupported : les instances de ce composant n'utilisent pas de transaction mais peuvent être utilisées pendant une transaction. Attention, la transaction n'est pas propagée aux instances de composants appelées par un composant en mode transactionnel NotSupported
- Supported : les instances de ce composant n'utilisent pas de transaction et peuvent être utilisées pendant une transaction. La transaction est propagée aux instances de composants appelées par un composant en mode transactionnel Supported.
- Required : les instances de ce composant doivent être exécutées en mode transactionnel. Soit l'appel de méthode véhiculait un contexte transactionnel et cette instance est enrôlée dans la transaction, soit l'appel n'était pas dans une transaction et une transaction est alors créée pour exécuter cette méthode. Toutes les connexions avec des bases de données utilisées pendant cet appel seront enrôlées dans la transaction courante.
- RequiredNew : un appel de méthode sur une instance de ce composant provoque la création d'une nouvelle transaction. Toutes les connexions avec des bases de données utilisées pendant cette transaction seront enrôlées dans celle-ci.

Il est aussi possible de modifier l'option transactionnelle de chaque instance par programme et de créer explicitement par programme des transactions.

Si la transaction a été créée par le moniteur transactionnel de COM+ suite aux attributs posés sur les composants, la fin de la transaction correspond à la fin de la méthode ayant provoqué sa création. Il est possible de signaler le mode de terminaison d'une méthode exécutée dans une transaction. L'attribut AutoComplete positionné sur la méthode MetAJour () permet de provoquer un arrêt de la transaction si cette méthode ne se termine pas normalement. En cas de terminaison normale la transaction se poursuit. Il est aussi possible de provoquer par programme un arrêt de la transaction :

```
// pas d'attribut pour cette méthode
public void MetAJour (int ID_Client, int montant) {
    ContextUtil.DeactivateOnReturn = true;
    // connexion à la base de données pour modification du solde du compte
    // fermeture de la connexion à la base de données
    // si (l'opération s'est bien déroulée) alors la transaction peut continuer
    ContextUtil.SetComplete ();
    // sinon la transaction doit être arrêtée
    ContextUtil.SetAbort ();
}
```

6.5 Exemple : Mise en œuvre d'une application complète

6.5.1 Architecture générale

Afin de décrire l'application d'exemple que nous allons mettre en œuvre, nous nous appuierons sur le cahier des charges suivant :

- Nous disposons (virtuellement) d'une bibliothèque de synthèse vocale (1) pour la plateforme Windows XP que nous appellerons TTS. Nous pouvons écrire une application (2) pour Windows XP qui inclut directement cette bibliothèque.
- Certaines autres applications (3) de l'Intranet sécurisé nécessitent également l'utilisation d'une bibliothèque de synthèse vocale, mais nous ne possédons pas de version de la bibliothèque pour ces plateformes (exemple d'un PDA). Nous créerons donc un service .NET Remoting (4) encapsulant les fonctionnalités de la bibliothèque TTS. Ce service sera distribué via un canal TCP Binaire permettant un accès rapide pour les clients de l'Intranet.
- Nous désirons offrir ce service à des applications tierces en dehors de la zone sécurisée de l'Intranet (5). Nous voulons mettre à disposition ce service de Synthèse vocale pour tout type de plate-forme (un client Java sous linux par exemple). Nous mettons alors en place un Service Web (6) constituant ainsi une passerelle standardisée vers le service .NET Remoting.

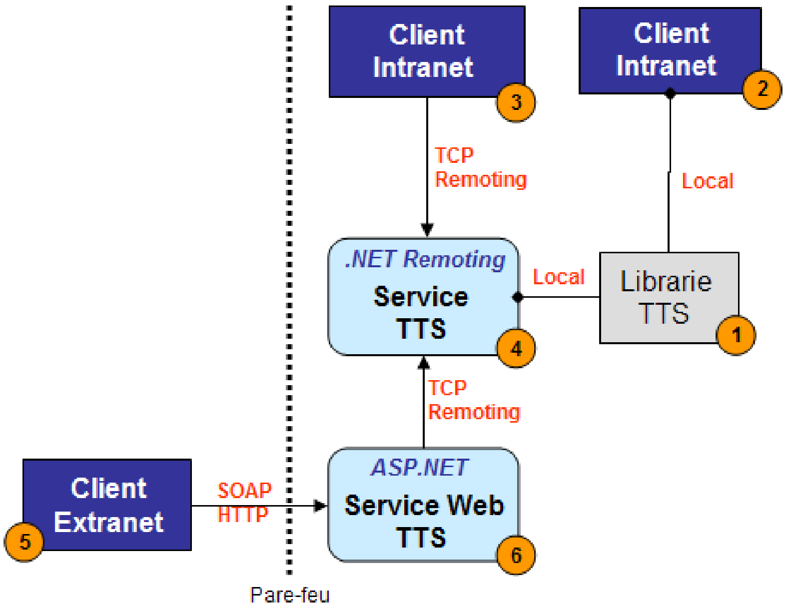

Figure 6.10 – Architecture générale de l'application

L'avantage de déporter le traitement d'un service réside souvent dans le fait que les clients de ce service ne disposent pas eux-mêmes des fonctionnalités leur permettant d'effectuer ce traitement sur le terminal sur lequel ils tournent.

6.5.2 Implémentation et déploiement des services

Etapes de la mise en œuvre

Avant de décrire en détails comment implémenter cette architecture, nous présentons ici le processus de développement utilisé. Nous développerons cette application en essayant de modulariser le plus possible les différentes parties de l'architecture. Les choix de conception sont donnés ici à titre indicatif mais peuvent être mis en œuvre de manière assez générale.

Nous commencerons tout d'abord par définir et implémenter les types de données échangées par les différentes entités. Dans un deuxième temps, nous définirons l'interface du service. Ces deux premières bibliothèques constitueront le sous ensemble minimal à partager entre client et serveur dans notre application. A' partir de ces deux briques, nous développerons et déploierons le service .NET Remoting. Cela consistera en une bibliothèque pour l'implémentation du service ainsi qu'une application pour héberger le service. Ensuite, nous écrirons un client pour ce service afin de le tester. Afin de dépasser les frontières de l'intranet, nous implémenterons ensuite le service web qui redirigera les appels clients vers le service .NET remoting. Enfin nous écrirons un client simple de ce service web.

Les types de données échangées

Le client enverra sa requête sous la forme d'un objet Sentence contenant un texte à convertir vocalement ainsi que la langue dans laquelle la synthèse vocale s'effectue. En réponse le service renverra un objet TTSResult dans lequel le flux audio sera représenté par un tableau d'octets.

```
using System;
using System.Xml;
using System.Xml.Serialization;

namespace TTS.Schema {
        [Serializable]
        XmlType("sentence")]
        public class Sentence {
                public Sentence(){ }
                public Sentence(string lang, string text) {
                        this.Language=lang;
                        this.Content=text;
                }

                XmlAttribute("lang")]
                public string Language;

                XmlElement("content")]
                public string Content;
        }

        [Serializable]
        [XmlType("result")]
        public class TTSResult {
                public TTSResult(){ }
                public TTSResult(byte[] stream) {
                        this.AudioStream=stream;
                }

                XmlElement("audio")]
                public byte[] AudioStream;
```

```
    }
}
```

Chaque instance de ces données devra traverser les frontières du service et donc pouvoir être sérialisé. Les deux classes sont donc marquées par l'attribut C# `Serializable`. Selon la configuration dans laquelle sera publié notre service, la sérialisation pourra prendre plusieurs formes. Si les données sont sérialisées via un formateur binaire, le résultat sera une série d'octets suivant une spécification propriétaire du Remoting .NET. Dans le cas ou le formateur de données sera un formateur SOAP, les données seront encodées sous forme de document XML. Le framework s'occupe par défaut de tout ce qui a trait à cette sérialisation. Par contre, si l'on veut pouvoir contrôler le format de ce document XML, nous devons marquer la classe ainsi que ses champs et méthodes par des attributs de sérialisation. Par exemple une instance de la classe sentence contenant "EN" dans le champ Language et "Bonjour" dans le champ Content sera transformée en XML comme suit :

```
<sentence lang="EN">
  <content>BONJOUR</content>
</sentence>
```

Si aucun attribut de contrôle de la sérialisation XML n'avait été employé, le framework aurait par défaut utilisé les noms de classe, champs et méthode, ce qui formaterait les instances comme ci-dessous :

```
<Sentence Language="EN">
  <Content>BONJOUR</Content>
</Sentence>
```

L'interface du service (partagée entre client et serveur)

```
using System;

namespace TTS.ServiceInterface {
     using TTS.Schema;

     public interface ISpeechService {
          TTSResult Speak(Sentence sentence);
     }
}
```

L'implémentation du service .NET Remoting (4)

Il s'agit ici d'implémenter l'interface du service sous la forme d'une classe héritant de la classe `MarshalByRefObject` afin qu'elle puisse être passée par référence.

```
using System;
using TTSLibrary;
using TTS.Schema;
using TTS.ServiceInterface;
```

```
namespace TTS.Service.Remoting {
     public class SpeechService: MarshalByRefObject,ISpeechService {
          public SpeechService() {
               Console.WriteLine("Instance de 'SpeechService' créée.");
          }

          public TTSResult Speak(Sentence sentence) {
               TTSObject speaker=new TTSObject(sentence.Language);
               return new TTSResult(speaker.DoSpeak(sentence.Content));
          }
     }
}
```

Déploiement du service .NET Remoting

Pour rendre le service accessible, nous devons choisir de le déployer dans un conteneur. Ce conteneur peut être au choix une application, un service Windows ou encore le serveur web IIS. Nous allons déployer notre service dans une application Console. Cette application une fois exécutée hébergera les instances du service.

Nous avons vu dans la section précédente comment paramétrer ce déploiement par programmation. Nous montrons ici comment effectuer ce paramétrage à l'aide d'un fichier de configuration. Cette méthode est beaucoup plus souple puisque le service n'a pas besoin d'être recompilé afin d'être déployé différemment. L'application hôte consistera uniquement à charger le fichier de configuration et à se bloquer (ici par l'appel à Console.ReadLine()) afin que l'application soit toujours disponible.

```
using System;
using System.Runtime.Remoting;

namespace TTS.Service.Remoting {
    public class MainClass {
        [STAThread]
        static void Main(string[] args) {
            RemotingConfiguration.Configure("server.config");
            Console.WriteLine("Appuyez sur [Entrée] pour quitter.");
            Console.ReadLine();
        }
    }
}
```

En .NET, chaque application dispose de son propre fichier de configuration. Ce fichier respecte une grammaire XML composée de différentes sections. En particulier nous nous intéressons ici à la section `<system.runtime.remoting>` permettant de paramétrer le mode d'exécution de notre service. La partie `<channels>` permet de spécifier l'ensemble des canaux de communication gérés par le framework .NET Remoting au sein de cette application. Nous utiliserons ici deux canaux différents permettant aux clients d'atteindre le service via deux ports différents : un canal ouvert sur le port 9000 et dialoguant via le protocole HTTP, un autre dialoguant sur le port 9001 via un canal

TCP. Vient ensuite la section `<services>` permettant de décrire le mode de publication de notre service. Nous le déployons dans notre exemple dans un mode Singleton en spécifiant l'assembly contenant l'implémentation du service ainsi que l'adresse relative à l'URL implicitement définie par les canaux de communications. Dans notre exemple le service sera accessible respectivement aux adresses `http ://localhost :9000/speech` et `tcp ://localhost :9001/speech`.

Fichier de configuration du serveur (server.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
  <application>
    <channels>
      <channel port="9000" ref="http" />
      <channel port="9001" ref="tcp" />
    </channels>
    <service>
      <wellknown
        mode="Singleton"
        type="TTS.Service.Remoting.SpeechService, TTS.Service.Remoting"
        objectUri="speech" />
    </service>
  </application>
  </system.runtime.remoting>
</configuration>
```

Via ce fichier de configuration, le composant en mode singleton sera accessible de deux manières différentes : soit via un canal HTTP sur port 9000, soit via un canal TCP sur le port 9001. Les deux URI correspondantes seront respectivement `http ://localhost :9000/speech` et `tcp ://localhost :9001/speech`.

6.5.3 Client du service .NET Remoting (3)

Le Code du client

```
ISpeechService service=(ISpeechService)Activator.GetObject(typeof(ISpeechService),url);
TTSResult result=service.Speak(new Sentence("fr",textBox1.Text));
```

Fichier de configuration du client (client.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
  <application>
    <client>
      <wellknown
        type="TTS.ServiceInterface.ISpeechService, TTS.ServiceInterface"
        url="http://localhost:9000/speech"/>
    </client>
```

```
  </application>
  </system.runtime.remoting>
</configuration>
```

6.5.4 Implementation du Service WEB (6)

Nous allons à présent créer le service web permettant d'accéder aux fonctionnalités de notre service en dehors de l'intranet de l'entreprise. Selon notre schéma, ce service web a pour rôle de rediriger ses requêtes vers le service .NET Remoting implémenté précédemment. Ainsi son code fonctionnel est très similaire au code d'un client .NET Remoting.

Le Code du service web

La classe du web service

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Web.Services;

using TTS.Schema;
using TTS.ServiceInterface;

public class TTSService : WebService {
    [WebMethod]
    public TTSResult Speak(Sentence sentence) {
        //RemotingConfiguration.Configure("client.config");
        ISpeechService remoting_service=
            (ISpeechService)
            Activator.GetObject(typeof(ISpeechService),
                                "tcp://localhost:9001/speech");
        return remoting_service.Speak(sentence);

    }
}
```

La page d'accès au webservice

```
<%@ WebService CodeBehind="TTSWebService.cs" Class="TTSService"%>
```

Compilation du service web

Notre service web dépend des différentes bibliothèques compilées précédemment. Il faut placer ces bibliothèques dans le répertoire `bin` de l'application web hébergement le service web.

Structure du répertoire de l'application

```
\www_icar
   \bin
      TTS.Schema.dll          --> Schémas de donnée
      TTSSpeechService.dll    --> Interface du service .NET Remoting
      TTSWebService.dll       --> Le web service compilé
      tts.asmx                --> La page d'accès
      TTSWebService.aspx      --> Le code c# du web service
```

La page d'accès au webservice

```
set PATH=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322;%PATH%

SET LIB_SCHEMA=bin\TTS.Schema.dll SET
LIB_INTERFACE=bin\TTS.SpeechService.dll

SET OUTPUT=bin/TTSWebService.dll SET SOURCES=TTSWebService.cs

csc.exe /reference:%LIB_SCHEMA% /reference:%LIB_INTERFACE%      \
        /reference:System.Runtime.Remoting.dll /out:%OUTPUT%    \
        /t:library %SOURCES%
pause
```

Déploiement du service web

Le service doit être hébergé dans une application ASP.NET sous le contrôle du serveur web IIS de Microsoft. La console d'administration des services Internet permet de créer ce répertoire que nous nommerons `icar`. L'arborescence de cette application doit contenir un sous-répertoire `bin` contenant tous les assemblages .NET nécessaires au fonctionnement du service web. Dans notre cas, ce répertoire devra contenir la bibliothèque contenant le web service `TTS.WebService.dll`, la bibliothèque `TTS.ServiceContract` ainsi que la bibliothèque `TTS.SpeechService` contenant l'interface du service. Dans un deuxième temps nous écrivons une page web avec l'extension `asmx` permettant de décrire le service web. Nous nommerons cette page `ttsservice.asmx`.

Le service est alors disponible à l'url `http ://hostname :80/icar/ttsservice.asmx`. Sa description WSDL peut être récupérée à l'adresse `http ://hostname :80/icar/ttsservice.asmx?WSDL`. Notre service web est désormais prêt à l'emploi.

6.5.5 Ecriture du client du Service web

Génération du proxy du service web via l'outil en ligne de commande `wsdl.exe`.

```
using System;

namespace TTS.ClientWebService {
    /// <summary>
    /// Description résumée de Class1.
    /// </summary>
```

```
    class Class1 {
        /// <summary>
        /// Point d'entrée principal de l'application.
        /// </summary>
        [STAThread]
        static void Main(string[] args) {
            TTS.WebService.sentence request = new TTS.WebService.sentence();
            request.content = "Salut";
            request.lang = "FR";
            TTS.WebService.result result =
                           new TTS.WebService.TTSService().Speak(request);
            Console.ReadLine();
        }
    }
}
```

6.6 Evaluation

6.6.1 .Net et les langages à objets

Comme on l'a vu dans les parties précédentes, .NET offre, grâce à une machine virtuelle, les mécanismes nécessaires à la représentation de données et aux échanges de données entre modules écrits dans des langages différents. Le but de cette première partie d'une évaluation de .NET est de s'assurer que les comportements des principaux langages (VB, C++, C# et J#) de la plate-forme .NET sont conformes à ceux attendus, en particulier pour l'un des points essentiels de la programmation objet : la redéfinition et la surcharge.

On parle de redéfinition lorsqu'une méthode possède plusieurs définitions qui peuvent participer à la liaison dynamique. Le compilateur ne peut pas choisir au moment de la compilation et met en place une procédure de *lookup* qui sera déclenchée lors de l'exécution du programme pour sélectionner la méthode appropriée ; en général, celle qui se trouve le plus bas dans le graphe d'héritage. Pour la surcharge il s'agit en fait de l'utilisation d'un même nom de méthode mais pour des méthodes que le compilateur considère comme différentes et qui ne sont donc pas éligibles en même temps lors de la liaison dynamique. Les langages à objets font des choix différents ; les redéfinitions peuvent être invariantes, covariantes ou contravariantes et la surcharge est parfois interdite. Pour mettre ces différences en évidence, Antoine Beugnard[9] propose une expérimentation simple.

Nous considérons deux classes : `Up`, et `Down` qui est une sous-classe de `Up`. Chacune des deux classes offre deux méthodes `cv` et `ctv` qui requièrent un paramètre unique. Les paramètres sont des instances de trois classes : `Top`, `Middle` qui est une sous-classe de `Top` et `Bottom` qui est une sous-classe de `Middle`. Pour tester tous les cas (covariants et contravariants) nous définissons les classes `Up` et `Down` selon la figure 6.11. La méthode `cv` parait covariante car les classes où elles sont définies et les classes des paramètres varient dans le même sens, tandis que la méthode `ctv` parait contravariante puisque les classes des paramètres varient dans le sens inverse des classes de définition de `ctv`.

Notre but n'est pas ici de discuter l'avantage de tel ou tel choix fait par les langages à objet, mais simplement de montrer qu'ils offrent de nombreuses sémantiques et que le

[9] L'ensemble de l'étude est disponible à l'adresse : `http ://perso-info.enst-bretagne.fr/ beugnard/papier/lb-sem.shtml`

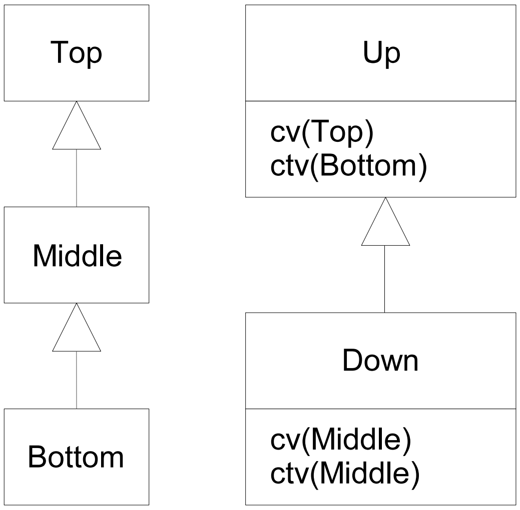

Figure 6.11 – Les classes utilisées pour l'expérimentation

framework .Net les respecte. Pour illustrer cela on crée, pour chacun des langages étudié, un programme client qui instancie les objets puis essaye tous les appels possibles : les deux méthodes (`cv` et `ctv`) étant appelées successivement avec les trois classes de paramètres possibles (`Top`, `Middle` et `Bottom`) pour les trois objets récepteurs possibles : u de type `Up` instancié avec un objet de classe `Up`, d de type `Down` instancié avec un objet de classe `Down` et ud de type `Up` mais instancié avec un objet de classe `Down`. Le dernier cas est autorisé car `Down` est une sous-classe de `Up`. Ce code peut parfois ne pas compiler ou ne pas s'exécuter lorsque le système de type l'interdit.

Les résultats sont présentés sous la forme d'un tableau de 6 lignes (une par appel possible) et 3 colonnes (une par objet récepteur possible). Chaque case contient la classe dans laquelle a été trouvée la méthode effectivement exécutée. Les sources et les résultats des exécutions peuvent être lus sur `http ://perso-info.enst-bretagne.fr/~beugnard/papier/lb-sem.shtml`. Ces tableaux sont appelés *signature du langage* (vis-à-vis de la redéfinition et de la surcharge).

Nous reproduisons dans les tables 6.1 à 6.4 les signatures pour C++, C#, J# et VB.Net. Ce sont les mêmes qu'avec d'autres compilateurs comme `gcc` ou `javac` (jusqu'à la version 1.3) par exemple.

C++	u	d	ud
cv(Top)	Up	**Compil. error**	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	**Down**	Up

TAB. 6.1 – signature de C++

On notera avec intérêt que bien que proches, ces langages ont tous une deuxième co-

C#	u	d	ud
cv(Top)	Up	**Up**	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	**Down**	Up

TAB. 6.2 – signature de C#

J#	u	d	ud
cv(Top)	Up	**Up**	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	**Compil. error**	Up

TAB. 6.3 – signature de Java before 1.3 (et J#)

lonne différente ! La différence entre ces 4 langages est due à des interprétations différentes de la surcharge. En effet, la troisième colonne montre que `cv` et `ctv` sont en fait des surcharges ; il aurait fallu voir apparaître des `Down` dans la troisième colonne pour avoir de la liaison dynamique et donc des redéfinitions. Ces 4 langages n'acceptent que des redéfinitions invariantes. Des expériences avec d'autres langages à objets (comme ADA95, Dylan, Eiffel, OCaml ou Smalltalk) montrent une plus grande variété encore.

Ce qui nous intéresse ici est de conclure que le framework .Net permet de définir plusieurs sémantiques et que les langages existants hors du framework ont la même sémantique qu'avec le framework (C++ et J#/Java version 1.3).

6.6.2 Compositions multi-langages en .Net

Comme nous venons de le voir, les interprétations de la redéfinition et de la surcharge dans les langages à objets sont nombreuses. Quelle conséquence cela peut-il avoir lorsqu'on assemble des morceaux de logiciel écrits dans des langages différents ? Le framework .Net met en avant la notion de composants. Ces composants peuvent être écrits avec plusieurs langages et leurs interfaces doivent assurer leur interconnexion. Mais, comme on va le voir avec .NET, les modèles à composants n'ont pas un comportement neutre vis-à-vis des langages d'implémentation de chacun des composants. En effet, la différence de comportement des langages à objets en ce qui concerne la redéfinition et la surcharge n'est pas prise en compte dans les interfaces des composants. Pour mettre cela en évidence, il suffit de construire la petite expérience suivante, toujours reprise du document d'habilitation d'Antoine Beugnard.

Imaginons le découpage de la figure 6.12. Les quatre classes `Up`, `Top`, `Middle` et `Bottom` de la partie précédente définissent un framework **C1** élaboré dans le langage *L1*. Par la suite, dans le cadre d'une évolution du framework, la classe `Up` est étendue par la classe

VB.Net	u	d	ud
cv(Top)	Up	**Up**	Up
cv(Middle)	Up	Down	Up
cv(Bottom)	Up	Down	Up
ctv(Top)	Compil. error	Compil. error	Compil. error
ctv(Middle)	Compil. error	Down	Compil. error
ctv(Bottom)	Up	**Up**	Up

TAB. 6.4 – signature de Visual Basic

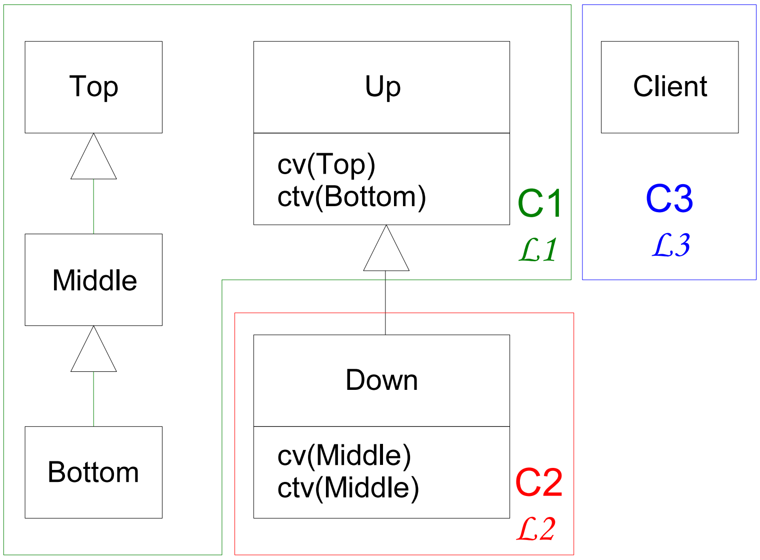

Figure 6.12 – Le découpage avec plusieurs langages

`Down` dans le langage *L2* pour former le composant **C2**. Cette extension, multi-langage est tout à fait permise par la plate-forme .Net. Enfin, pour compléter l'expérimentation un client **C3** est écrit dans le langage *L3*. Pour mettre en œuvre cette expérimentation nous avons utilisé les langages VB, C++ et C# qui forment le trio des langages de base de la plate-forme .Net. Nous obtenons donc un total de 27 (3*3*3) assemblages possibles. Pour présenter les résultats, nous construisons un tableau de 9 signatures pour chacun des 3 clients (**C3**). Les colonnes de chaque tableau représentent les langages d'implantation du framework de référence **C1** et les lignes, les langages d'implantation de l'extension **C2**. Chaque cellule du tableau contient la signature observée.

Dans le tableau suivant, le client est en C#.

Extension/Framework	C#	VB	C++
C#	*C#*	*C#*	*C++*
VB	*C#*	*C#*	*C++*
C++	*C#*	*C#*	*C++*

On constate que le comportement attendu par le client est majoritairement celui de C#, mais que dans le cas où le framework (**C1**) est écrit en C++, l'anomalie d'héritage de la signature de C++ (erreur de compilation ligne 1, colonne 2 de la table 6.1) est toujours présente.

Dans le tableau suivant, le client est en Visual Basic.

Extension/Framework	C#	VB	C++
C#	*VB*	*VB*	*C++*
VB	*VB*	*VB*	*C++*
C++	*VB*	*C#*	*C++*

On constate que le comportement attendu par le client est majoritairement celui de Visual Basic, mais que dans le cas où le framework (**C1**) est écrit en C++, l'anomalie d'héritage de la signature de C++ est encore présente. Un nouveau comportement apparaît lorsque le framework est écrit en Visual Basic et l'extension en C++, le client Visual Basic observe un comportement “ à la ” C#.

Dans le tableau suivant, le client est en C++.

Extension/Framework	C#	VB	C++
C#	*VB/C++*	*VB/C++*	*VB/C++*
VB	*C++*	*C++*	*C++*
C++	*C++*	*C++*	*C++*

On constate que le comportement attendu par le client est majoritairement celui de C++, mais que dans le cas où l'extension (**C2**) est écrit en C#, un nouveau comportement apparaît : la signature observée par le client est un mélange de C++ et de VB (erreur de compilation comme C++ ligne 1, colonne 2 table 6.1 et choix de Up comme VB ligne 6 colonne 2 table 6.4). Cette signature ne correspond à aucun des 13 langages à objet étudiés par Antoine Beugnard.

Au delà de la compréhension fine des résultats, cette rapide expérience montre que pour pouvoir prévoir le comportement d'un assemblage, le client (et donc l'utilisateur) doit avoir connaissance de la manière dont la surcharge est interprétée, et donc du langage d'implémentation de la partie serveur. Il pourrait être intéressant de pousser plus avant cette expérimentation avec les services web, pour vérifier là aussi la réelle neutralité de la plate-forme vis-à-vis des langages d'implémentation.

6.6.3 Quelques éléments de performances

L'évaluation d'un langage et d'une plate-forme est quelque chose de complexe. Cette courte section ne se donne pas comme ambition d'évaluer toutes les caractéristiques mais de donner quelques éléments de référence. Une étude publiée dans www.gotdotnet.com à propos de l'application PetStore qui servait d'exemple à la plate-forme J2EE titrait : “.NET, 10 fois plus rapide que J2EE”. N'ayant pas les capacités pour reproduire sur une autre application significative cette étude nous avons dans un premier temps mesuré le cout des appels de méthodes. La figure suivante donne quelques valeurs comparatives entre un appel de méthode local en Java et en .NET, puis entre un appel distant entre Java RMI et .NET Remoting, la taille des paquets d'appels évoluant.

Attention il s'agit d'une échelle logarithmique (cf. figure 6.14). Nous n'arrivons pas à expliquer les différentes discontinuités présentes dans les mesures .NET en local.

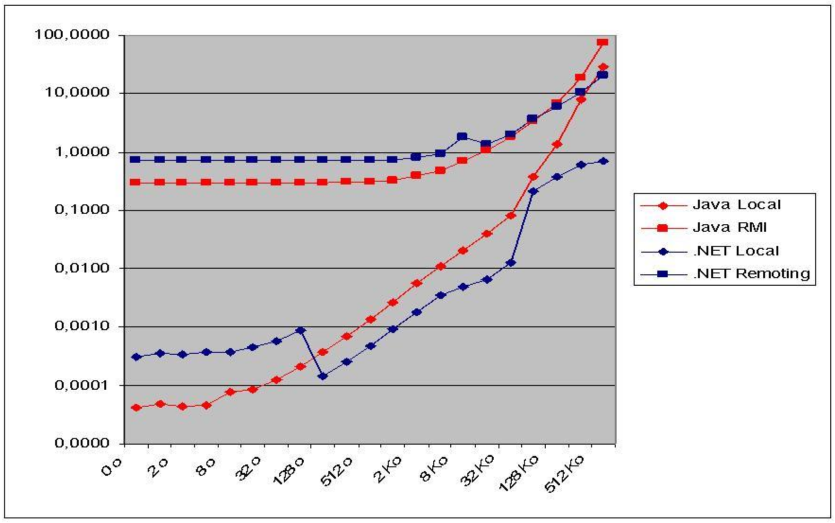

Figure 6.13 – Coût d'un appel de méthode

Voici deux autres éléments complémentaires incluant des utilisations plus complexes du langage et de la plate-forme. L'ensemble des résultats, y compris avec les principaux fichiers sources, est disponible à l'adresse : `http ://www.dotnetguru.org/articles/Comparatifs/benchJ2EEDotNET/J2EEvsNETBench.html`.

Etude du démarrage d'une application graphique

L'application utilisée pour ce test est simpliste par ses fonctionnalités mais permet de mesurer le temps de démarrage d'initialisation de la partie graphique d'un client lourd. Elle consiste à charger un formulaire permettant d'enregistrer un utilisateur, photos comprise dans une base. Sans équivoque la version C# avec Winforms a été nettement plus rapide que la version Java avec Swing.

Efficacité de la mise en œuvre d'un service web

Pour évaluer celle-ci, l'auteur du papier a choisi d'appeler une fonction permettant de trouver le minima d'un tableau d'entiers. Ce service web a été implanté en .NET et à l'aide de la plate-forme Java/axis puis en Java avec la plate-forme Glue (www.themindelectric.com). A été mesuré uniquement (cf. figure 6.15) le temps de réponse du service sans vouloir décomposer celui-ci entre les différentes activités : empaquetage/dépaquetage, transfert des données, activation du service, traitement du coté serveur.

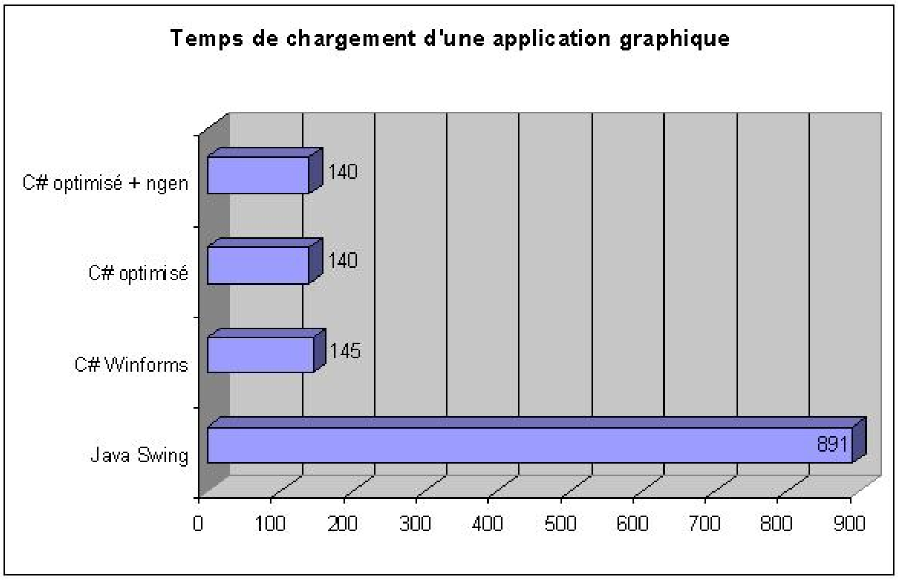

Figure 6.14 – Coût démarrage application graphique

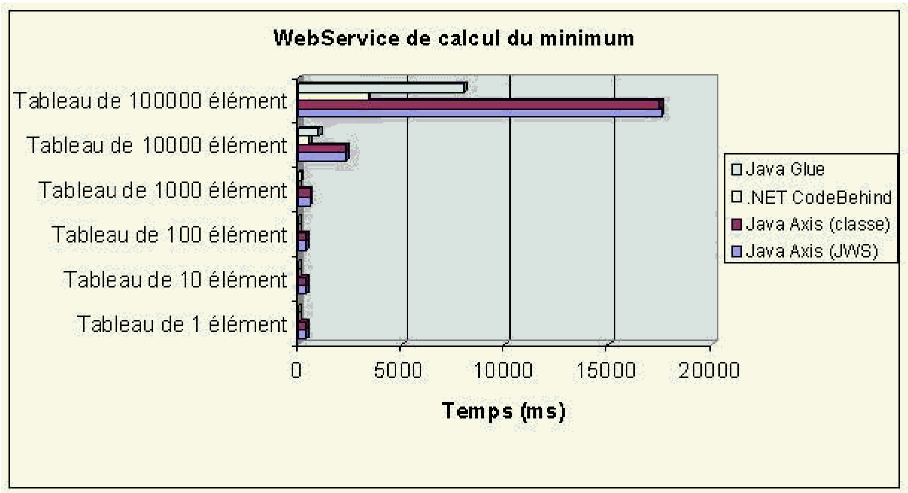

Figure 6.15 – Cout d'utilisation d'un service web

6.6.4 La plate-forme .NET 2.0 et Indigo : nouveautés à venir

Une nouvelle plate-forme de distribution

L'architecture distribuée orientée Services Web (S.O.A) repose sur quatre principes fondamentaux :
- un couplage faible entre services (fournis par SOAP et WSDL) ;
- un bus de communication synchrone (HTTP) ou asynchrone (Middleware Oriented Messages : MOM)
- une invocation des services de type échange de documents (XML, RPC)
- des processus métiers qui pilotent l'assemblage de services Web (Business Process Management : BPM).

Le futur de la plate-forme .NET (appelé Indigo) renforce encore plus le souhait de Microsoft de se positionner comme un acteur dans une approche de programmation orientée services. L'objectif d'Indigo est de choisir les meilleurs abstractions pour construire des logiciels orientés processus en intégrant de manière plus étroites les services nécessaires à ces applications (bus de messages, sécurité et transaction) en s'appuyant de manière explicite sur les travaux réalisés par le consortium Oasis. Indigo fait lui-même une partie de la nouvelle version de Windows nommée Longhorm.

Pour atteindre cet objectif, Indigo unifie l'ensemble des « attributs » auparavant disponibles pour construire un service. Par exemple, pour publier un web service, il était nécessaire de faire précéder chaque méthode exposée de l'attribut `[WebMethod]`, de même, un composant qui devait utiliser le service transactionnel devait être précédé de l'attribut `[Transaction]`. Indigo propose d'introduire la notion de contrat :

```
Using System.ServiceModel;

[ServiceContract]
public class AnnuaireWebService {
     private AnnuaireService service;
     public AnnuaireWebService() {
          service=new AnnuaireService();
     }
     [OperationContact]
     public CarteDeVisite Rechercher(string nom) {
          return service.Rechercher(nom); ;
     }
}
```

Les contrats peuvent porter sur les classes ou sur les méthodes, mais aussi sur les données échangées afin de permettre le transport de données structurées. Pour chaque points de publication d'un service, sans avoir à modifier le code de celui-ci, ni du client qui le consomme il sera possible de préciser dans un fichier de configuration : le protocole de transport (par exemple : HTTP), l'adresse du service mais aussi les services techniques associés aux appels (sécurité ou transaction) basés sur les spécifications du W3C pour les Web services.

La construction d'un client Indigo sera identique (et proche de la manière de procéder actuellement pour le Web services) : à partir d'un proxy du service généré à partir du fichier de publication du service, le client pourra accéder au service de la même manière quelque soit le protocole utilisé (.NET remoting ou SOAP).

Evolution du langage de référence : C# version 2.0

C# va, avec la version deux de la plate-forme .NET, être complété par l'introduction de très nombreuses nouveautés : la généricité, les itérateurs, les méthodes anonymes et les classes partielles pour ne citer que les améliorations les plus notables.

Généricité En C++ les templates sont extrêmement utilisés et le fait qu'ils n'ont pas été implanté sous C# a été un des reproches majeurs à sa sortie. Microsoft a écouté les utilisateurs et l'a mis en place en nommant dorénavant la technique Generic. Pour contourner l'absence de généricité, les programmeurs C# utilisaient actuellement des collections d'objets non typés et sollicitaient massivement l'usage coûteux du boxing/unboxing et de cast. L'introduction de la généricité devrait permettre d'augmenter le typage statique des programmes et d'augmenter simultanément l'efficacité de ceux-ci.

La généricité sera utilisée pour simplifier l'écriture des classes de collections qui puissent être énumerable (c'est à dire interrogeables dans une boucle foreach).

Méthodes génériques Les méthodes anonymes sont aussi une manière d'alléger le code en particulier dans la gestion des événements. Actuellement, il était nécessaire de déclarer un 'callback d'événement' par une méthode qui possédait un nom. Dorénavant, il suffira juste de donner le code à exécuter :

```
void Page_Load (object sender, System.EventArgs e) {
     this.Button1.Click += delegate(object dlgSender, EventArgs dlgE) {
          Label1.Text = " Vous avez cliqué sur le bouton 1 !";
     }
};
```

Classes partielles Les classes partielles permettront à une équipe de développeurs de travailler en commun : la même classe sera contenue dans plusieurs fichiers. En reprenant l'exemple de la section précédente, on trouvera par exemple dans le fichier annuaireService.cs le code suivant :

```
public partial class AnnuaireService : IAnnuaireService {
     // contructeur de la classe
     public AnnuaireService(){}
     // implementation de la méthode métier
     public CarteDeVisite Rechercher(string nom) {
          AnnuaireDAL dal=new AnnuaireDAL();
          return dal.ExtraireCarte(nom);
     }
}
```

Le fichier AnnuaireAdminService.cs, pourra lui contenir :

```
public partial class AnnuaireService : IAnnuaireAdminService {
     // implementation de la méthode métier
     public void Ajouter(CarteDeVisite carte) {
          AnnuaireDAL dal=new AnnuaireDAL();
          // on verifie que la carte existe pas déjà
          CarteDeVisite test=dal.ExtraireCarte(carte.Nom);
          if (test==null)
```

```
                dal.AjouterCarte(carte);
            else
                throw new Exception("La carte existe déjà");
        }
}
```

6.7 Conclusion

La terre est envahie de composants COM, et si l'application que vous voulez construire fait un usage intensif de cet univers là alors .NET est fait pour vous. Vous y trouverez des outils de développement sophistiqués, des langages de programmation adaptés à vos besoins, la possibilité de construire un composant en utilisant plusieurs langages de programmation grâce au langage de type commun. Mais même si vous souhaitez écrire une nouvelle application .NET peut aussi être fait pour vous, voici quelques apports :

- Point d'accès : c'est le point fort par excellence de l'approche .NET, n'importe quel client Web peut accéder à un service publié sur le Web. Il n'est pas nécessaire d'avoir une configuration spécifique, d'installer une pile de protocole, de configurer un pare-feu (firewall). L'intégration avec les standards du Web est totale, un client XML et la disponibilité prochaine du protocole SOAP permettront cet usage et rendront peutêtre inutile la programmation de script CGI, ceux-ci étant remplacés par des composants .NET.
- Simplicité d'utilisation : .NET a été conçu dès l'origine comme une plate-forme à composants distribués intégrant les standards de l'Internet XML et SOAP. Pour cela il n'est pas nécessaire de d'envelopper (wrapper) les composants afin de les rendre accessibles, il n'est pas non plus nécessaire d'ajouter du code lors du déploiement pour permettre l'exécution du composant car celui-ci est inclus dans la plate-forme ou directement associé avec les composants. Un composant est un objet simple qui contient tout ce qui est nécessaire pour le manipuler : la description de ses interfaces, la description de ses services, le code pour accéder aux services et les services eux mêmes. Le déploiement est immédiat. .NET propose aussi avec ADO un modèle très évolué pour l'accès aux données acceptant le mode déconnecté, permettant l'accès à différents formats de base de données, etc. Toutes les données peuvent être accéder immédiatement depuis n'importe quel client de la plate-forme qui se charge de les transporter et de les convertir.
- Les performances : selon que l'on s'intéresse à la manière dont le service est rendu ou l'accès au service, le point de vue est diamétralement opposé. En ce qui concerne l'exécution des services, les performances sont au rendez-vous : code compilé, multiprocesseur, équilibrage de charge, etc. Par contre, les appels inter-service seront lents par nature. En effet le choix d'utiliser le protocole SOAP afin de pouvoir être accédé depuis n'importe quel type de station cliente se paye au prix fort. La technologie proposée permet essentiellement de relier entre eux des composants ayant une grosse granularité.
- Multiplicité des langages de programmation : .NET ne fait pas le choix d'un langage de programmation à votre place, il vous laisse libre de choisir le langage que vous voulez utiliser en fonction de vos usages ou de votre humeur. COM permettait de faire

coopérer des composants écrits dans des langages différents, .NET permet d'intégrer les langages les uns avec les autres. Tous les langages sont a priori égaux, même si les langages de types procéduraux sont plus proches de la machine virtuelle proposée. Il par exemple est possible d'utiliser simultanément différents langages (un par classe par exemple). Le typage est complet et les différents types du Common Language Runtime, dont par exemple les threads sont disponibles dans les différents langages de programmation de la plate-forme. Des compilateurs C++, C#, VB et Jscript sont aujourd'hui disponibles et différentes compagnies ont déjà annoncé qu'elles allaient sous peu fournir des compilateurs. Il est imaginable que dans peu de temps, des compilateurs APL, ML, Cobol, Oz, Pascal, Perl, Python, Smalltalk et Java seront disponible sur la plate-forme .NET.

Bibliographie

[Ben Albahari et al. 2002] Ben Albahari, Peter Drayton, and Brad Merrill (2002). *C# Essentials.* O'Reilly. ISBN: 0-596-00315-3, 216 pages.

[Beugnard 2005] Beugnard, A. (2005). Contribution à l'étude de l'assemblage de logiciels. Habilitation à diriger des recherches, Université de Bretagne Sud.

[Beugnard 2006] Beugnard, A. (2006). Method overloading and overriding cause encapsulation flaw. In *Object-Oriented Programming Languages and Systems, 21st ACM Symposium on Applied Computing (SAC)*, Dijon, France.

[Edd Dumbill and M. Bornstein 2004] Edd Dumbill and M. Bornstein, N. (2004). *Mono: A Developer's Notebook.* O'Reilly. ISBN: 0-596-00792-2, 304 pages.

[Gunnerson and Wienholt] Gunnerson, E. and Wienholt, N. *A Programmer's Introduction to C# 2.0.* Apress. ISBN 1590595017, 568 pages.

[Stutz et al. 2003] Stutz, D., Ted Neward, and Geoff Shilling (2003). *Shared Source CLI Essentials.* O'Reilly. ISBN 0-596-00351-x, 378 pages.

[Weatherley and Gopal 2003] Weatherley, R. and Gopal, V. (2003). Design of the Portable.Net Interpreter DotGNU. In *Linux.conf.au, Perth (Australia), 22-25 January.*