Les EXE

ActiveX

NB : ce fascicule fait partie d'un travail de diplôme sur le sujet « Technologie ActiveX & Visual Basic 6 ».
Les autres fascicules peuvent être demandés à fcomte@caramail.com.

La reproduction – sous n'importe quelle forme que ce soit - est libre de droits. Veuillez tout de même en informer l'auteur.
Critiques, remarques, questions ? fcomte@caramail.com

1 - Caractéristiques

Un programme ActiveX EXE est similaire à une DLL ActiveX, bien qu'un tel programme ne s'exécute pas dans l'espace mémoire courant, mais dans son propre espace. De façon plus technique, on dira qu'un programme ActiveX EXE est un serveur « out-of-process », et qu'une DLL ActiveX, un serveur « in-process ».

Un ActiveX EXE, tout comme une DLL ActiveX, est un objet COM, construit avec Visual Basic en utilisant les modules de classes. Il est possible de déplacer ces modules de classe d'une DLL ActiveX vers un ActiveX EXE sans en changer une seule ligne de code.

Pourquoi alors développer un ActiveX EXE plutôt qu'une DLL ActiveX ? Le choix à faire entre ces deux méthodes lorsqu'on développe une application n'est pas forcément évident. Les communications entre un client et un serveur sont beaucoup plus efficaces avec un serveur in-process plutôt qu'avec un serveur out-of-process. Cependant, un serveur out-of-process a la possibilité d'être exécuté sur une autre machine, ce qui nous permettrait de dédier une machine dans le modèle de programme client-serveur.

Nous pouvons également employer cette technique pour créer des instances d'objets dans une DLL d'ActiveX distante. Puisque la DLL est un serveur in-process, elle ne peut pas fonctionner comme processus autonome. Windows contient un serveur de « substitution » qui peut être utilisé à cette fin, mais il faudra alors éditer la base de registre de Windows directement pour créer toutes les définitions appropriées pour ce travail. Pour cette raison, la plupart des auteurs recommandent d'éviter cette approche pour créer les objets distants et pour les lier avec des programmes ActiveX EXE.

2 - Intérêt des EXE ActiveX

2.1 Avantages

→ Les objets s'exécutent dans leur propre thread d'exécution.
→ Les objets peuvent être créés et utilisés par l'application cliente comme par l'exécution du serveur en temps qu'application autonome.

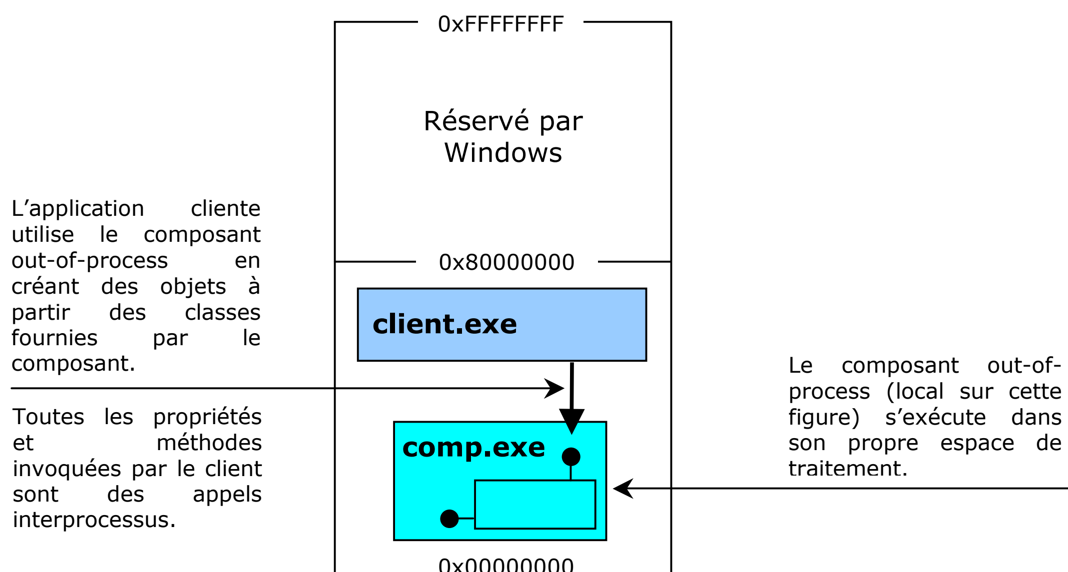
2.2 Inconvénients

→ Les performances sont considérablement plus mauvaises que celle d'une DLL ActiveX.
→ Le taux d'utilisation du système est élevé du fait du lancement d'une tâche séparée permettant le chargement de l'objet.
→ La complexité du déploiement de l'application est augmentée.
→ L'enregistrement, la vérification de version, ainsi que la vérification des composants sont requis pour une distribution sûre du produit.

En résumé, les ActiveX EXE sont idéaux pour exposer un modèle applicatif à d'autres programmes, ainsi que pour implémenter des objets qui s'exécutent en tâche de fond (les threads sont séparés) et de façon asynchrone à notre application.

3 - Un EXE ActiveX est un composant out-of-process

Un serveur local s'exécute dans un processus séparé sur la même machine que le client. Ce type de serveur est un EXE, ce qui lui permet de se qualifier comme processus séparé. Les serveurs locaux sont nettement plus lents d'accès que les serveurs in-process car le système d'exploitation doit basculer entre les processus et copier toute donnée qui doit être transférée entre client et applications serveurs. L'un des avantages des serveurs locaux est que, comme il s'agit de fichiers EXE, ils peuvent être exécutés par l'utilisateur en tant qu'applications autonomes sans client externe. Par exemple, une application telle que Microsoft Internet Explorer offre un bon exemple de serveur local, car il est possible de l'exécuter pour parcourir l'Internet ou pour appeler ses objets à partir d'une autre application, comme Visual Basic.



Le fait qu'un composant out-of-process s'exécute dans son propre processus signifie qu'un client peut lui demander d'accomplir une action puis reprendre ses activités pendant que le composant traite l'ordre reçu. Quand un tel système est correctement installé, le composant peut indiquer au client qu'il a terminé la tâche confiée par le biais d'une notification asynchrone[1].

---

[1] Lorsqu'un client effectue un appel de méthode, il est bloqué jusqu'au retour de l'appel. C'est-à-dire que le client ne peut pas exécuter de code pendant cette attente. Cela est qualifié de traitement synchrone. En utilisant un traitement asynchrone, il est possible de libérer le client pour lui permettre d'effectuer d'autres opérations pendant cette attente.
En traitement asynchrone, l'appel de la méthode qui démarre une tâche revient instantanément, sans fournir de résultats. Le client effectue le traitement qui lui est imparti, pendant que le composant exécute la tâche. Une fois la tâche terminée, le composant avertit le client que le résultat est prêt.
Le traitement asynchrone est également utile lorsque des clients doivent être avertis d'occurrences intéressantes, par exemple des modifications de valeurs de base de données, ou l'arrivée de messages. Un client indique à un composant qu'il souhaite être averti de l'occurrence de certains événements, et le composant envoie des notifications lorsque ces événements se produisent.
Ces deux scénarios dépendent de notifications asynchrones. L'application client exécute les tâches qui lui sont confiées puis une notification l'avertit que la requête asynchrone est terminée ou qu'un événement intéressant s'est produit. Visual Basic offre deux mécanismes de mise en œuvre des notifications asynchrones : par événements (le composant déclenche un événement, le client l'intercepte puis répond par une intervention quelconque), ou par méthodes de rappel (Le client met en œuvre une interface comportant une méthode de rappel, que le composant appelle lorsque des notifications sont requises).

4 Création d'un EXE ActiveX

Un EXE ActiveX se crée quasiment de la même manière qu'une DLL ActiveX, alors pourquoi deux modèles différents de composants ? Nous avons vu précédemment que si ces deux modèles sont semblables, il ne s'exécutent pas de la même manière, et ne sont pas aussi rapide l'un que l'autre (en raison du modèle de threading). De plus, il faut savoir que si on veut développer une solution avec MTS (Microsoft Transaction Server), il faudra travailler avec les DLL, alors que si c'est DCOM qui nous intéresse, il faudra pencher du côté des EXE ActiveX. Nous verrons dans le chapitre consacré à DCOM comment mettre en œuvre un EXE ActiveX distant, au sein d'une application client/serveur.

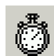
Nous allons maintenant regarder comment mettre à profit l'avantage des EXE ActiveX : le fait qu'ils soient des composants out-of-process.

Imaginons que nous avons besoin d'une application qui vérifie continuellement si une modification est effectuée sur un fichier (une base de données par exemple). Une application « conventionnelle », munie d'un mécanisme similaire à un timer, peut tout à fait remplir ce rôle. La problème qui risque d'apparaître est qu'à chaque opération de vérification, le programme entier sera gelé sur le timer, et qu'aucune autre opération ne sera possible à ce moment. Ennuyeux, non ? C'est ici qu'un EXE ActiveX peut donner un sérieux coup de main : en utilisant la notification asynchrone par événement (nous verrons plus loin de quoi il est précisément question), la tâche de vérification sera prise en charge par le composant pendant que notre application poursuivra son travail. Laquelle sera avertie dès que le composant aura terminé.

4.1 Création du projet Notification

Notre composant permettra de vérifier la présence d'un fichier à un endroit donné. Toutes les 60 secondes, il vérifiera l'existence du fichier en question, et enverra un message de notification à l'application cliente.

Commençons par créer un nouveau projet ActiveX EXE, et renommons-le « Fichier ». Nommons la classe « VerifFichier ».



Ajoutons une feuille, ainsi qu'un contrôle `Timer`. La feuille ne sera pas affichée, mais le contrôle `Timer` sera lui utilisé pour vérifier la présence du fichier toutes les 60 secondes.

Déclarons ensuite sur cette feuille la variable `NomFichier`, qui désignera le fichier à vérifier :

```
Public NomFichier As String
```

Ainsi que l'événement `FichierTrouve`, qui sera levé si le `Timer` trouve le fichier.

```
Public Event FichierTrouve()
```

Le `Timer` quand à lui aura ce code :

```
Private Sub Timer1_Timer()
      ' si le fichier est trouve
      If Dir(NomFichier) <> "" Then
            ' l'evenement est leve
RaiseEvent FichierTrouve
' stoppe les verifications suivantes
            Timer1.Interval = 0
      End If
End Sub
```

Ouvrons maintenant la classe `VerifFichier` et ajoutons (dans les déclarations générales) :

```
Dim WithEvents objVerifFichier As Form1
```

Le mot-clé `WithEvents` indique que la classe peut recevoir des événements transmis par la feuille, comme le code `FichierTrouve`.

Dans la liste déroulante Objet, sélectionnons Class, ainsi que la méthode Initialize, et ajoutons le code suivant :

```
Private Sub Class_Initialize()

' crée une nouvelle instance de Form1
Set objVerifFichier = New Form1

End Sub
```

A partir de là, on peut utiliser toutes les fonctionnalités écrites dans Form1.
Ecrivons à présent une procédure permettant de surveiller le fichier (au sein de la classe VerfiFichier) :

```
Public Sub Surveillance (NomFichier As String)

objVerifFichier. NomFichier = NomFichier
objVerifFichier.Timer1.Interval = 60000

End Sub
```

Lorsque le client appelle `Surveillance` – en lui passant un nom de fichier – la variable `NomFichier` de la feuille est remplacée par le paramètre. Ensuite le `Timer` démarre, pour une durée de 60'000 millisecondes (une minute).

Il faut maintenant avertir le programme utilisant l'EXE ActiveX (lorsque le fichier est trouvé), en levant l'événement `FichierTrouve`. Ajoutons donc la déclaration suivante (toujours dans le module de classe), définissant l'événement `FichierTrouve` :

```
Public Event FichierTrouve(NomFichier As String)
```

Et ajoutons son code :

```
Private Sub objVerifFichier_FichierTrouve()
RaiseEvent FichierTrouve(objVerifFichier.NomFichier)
End Sub
```

Le lancement de la méthode `Surveillance` envoie le `Timer`. Lorsque celui-ci trouve le fichier (passé en paramètre de `Surveillance`), il lève un événement dans la classe `VerifFichier`, qui lui, lève également un événement indiquant à l'utilisateur que son fichier a été trouvé.

Choisissons Fichier → Créer Fichier.exe… et testons maintenant notre petit EXE ActiveX.

4.2 Application de test

Créons un nouveau projet EXE standard, et ajoutons une référence à notre composant Fichier.exe. Comme on l'a vu avec les DLL, VB s'occupe tout seul d'inscrire le fichier que nous venons de créer dans la base de registres.

Ajoutons ensuite le code suivant sous les déclarations générales :

```
Dim WithEvents MonObjetFichier As VerifFichier
```

Et :

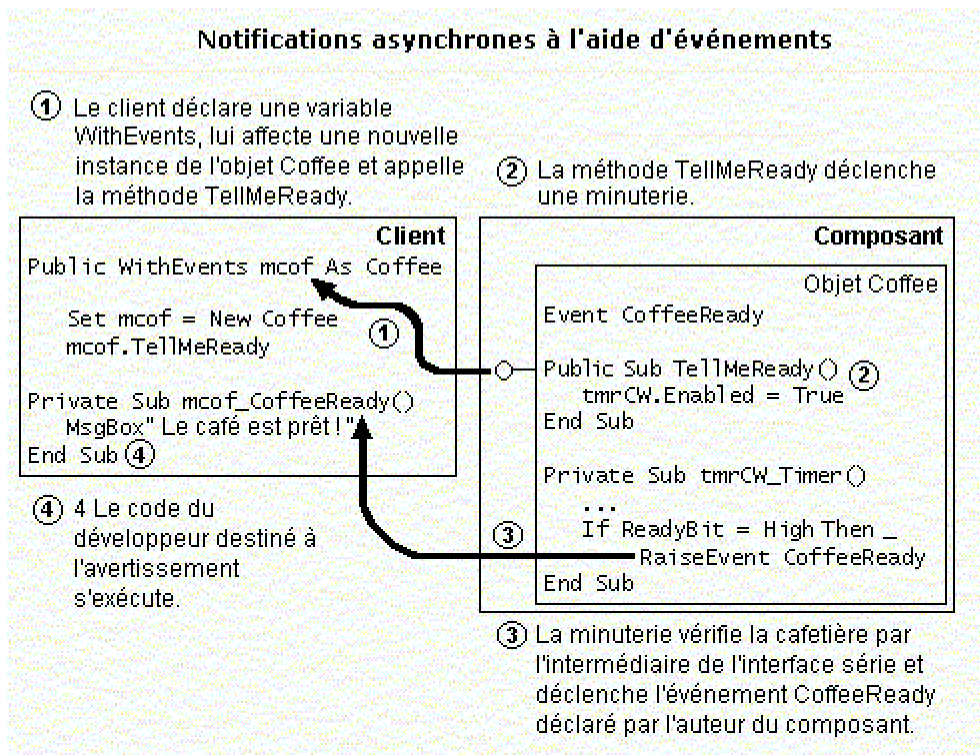
```
Private Sub MonObjetFichier_FichierTrouve(NomFichier As String)
MsgBox "Fichier trouvé : " & NomFichier
End Sub
```

Ajoutons un bouton sur la feuille et saisissons le code :

```
Private Sub Command1_Click()
Set MonObjetFichier = New VerifFichier
MonObjetFichier.Surveillance ("c:\test.txt")
End Sub
```

Exécutons le projet de test. Normalement, il ne devrait pas y avoir de fichier text.txt sur notre disque c:\. Donc, il ne va rien se passer. Mais si nous créons un simple fichier nommé text.txt et que nous le plaçons sur le disque c:\, dans les 60 secondes une boîte de dialogue va apparaître nous indiquant qu'elle a trouvé le fichier.

Ce petit exemple nous permet de mieux comprendre les notifications asynchrones à l'aide d'événements. Le schéma suivant – tiré de la MSDN – explique dans quel ordre les instructions sont parcourues dans un tel mécanisme.

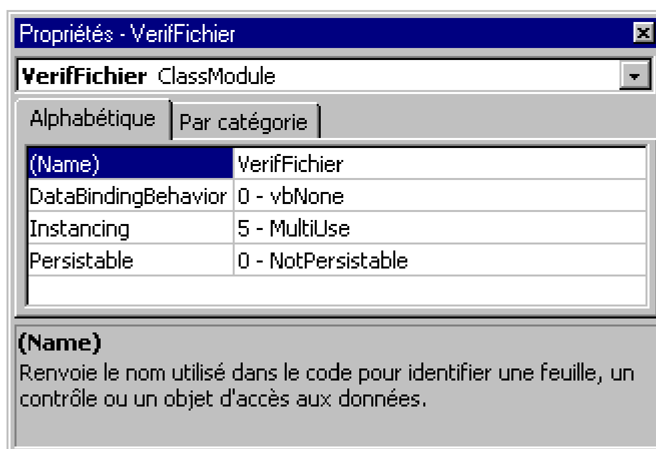


5 - Propriétés d'un projet EXE ActiveX

Elles sont sensiblement différentes de celles d'un projet DLL ActiveX. Premièrement la propriété MTSTransactionMode n'existe pas, et deuxièmement, la propriété Instancing offre les valeurs 3 et 4 suivantes (en plus de celle qu'une DLL ActiveX offre) :

3 – SingleUse : Permet aux autres applications de créer des objets à partir de la classe, mais chaque objet de cette classe créé par un client démarre une nouvelle instance du composant, exécuté dans son propre espace d'adressage.

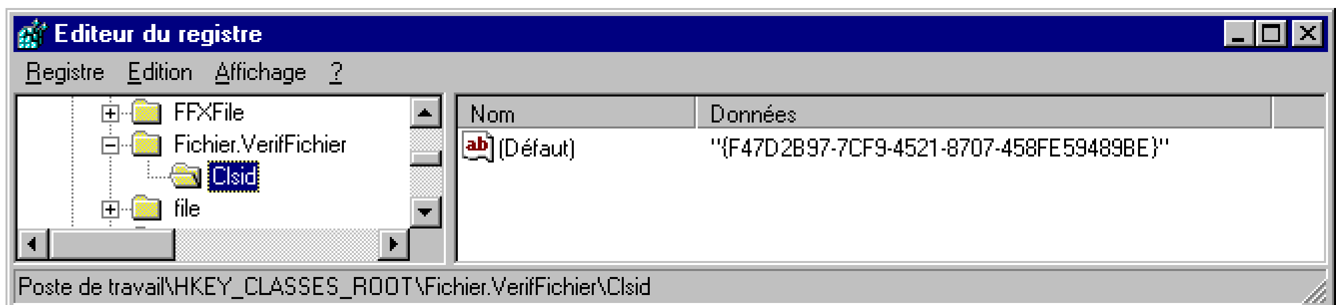
4 – GlobalSingleUse : Similaire à SingleUse, à l'exception que les propriétés et méthodes de la classe peuvent être invoqués comme de simples fonctions globales.



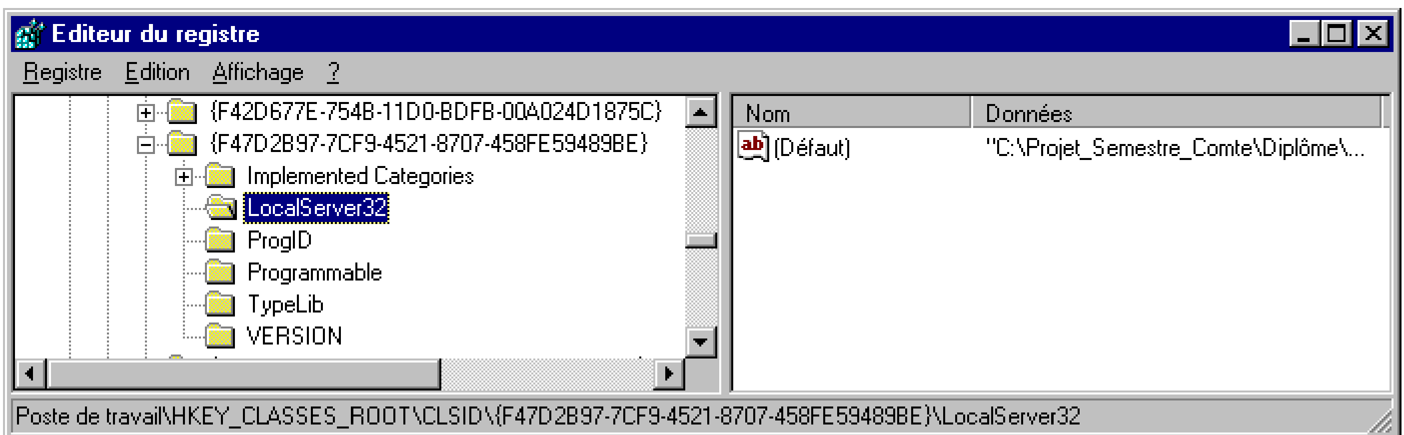
6 - Cycle de vie

Tout comme une DLL ActiveX, un objet dans un EXE ActiveX est inscrit (registered) automatiquement – sous VB – lors de la compilation, et peut être désinscrit (unregistered) avec le programme /UnRegServer (dans la ligne de commande).
Mais on peut également inscrire un fichier .VBR (créé à la demande lorsqu'on compile un composant, depuis la fenêtre des propriétés du projet) avec le programme CliReg32.exe (inclus dans la version Entreprise de VB), et ceci afin d'utiliser le composant de manière distante[2].

Les entrées créées par l'EXE ActiveX dans la base de registres sont les mêmes que celles d'une DLL ActiveX.



La grosse différence se situe dans l'emplacement : au lieu d'être située sous la clé \InProcServer32, elles sont sous \LocalServer32, ce qui indique par là que le serveur implémentant l'objet s'exécute sur le système local, et non au sein d'un serveur in-process.



De plus, le fichier .IDL est quasiment identique à celui d'une DLL, la librairie de types étant similaire.

[2] Le fichier .VBR contient des informations dans la librairie de types permettant d'utiliser le composant de manière distante.

6.1 Cycle de vie de l'EXE ActiveX

Au moment de l'enregistrement (Registration Time)

Tout ce que nous avons dit pour les DLL est vrai pour les EXE (registration automatique par VB), si ce n'est que cette registration peut également être faite lors de l'exécution de l'EXE.

Au moment de la conception (Design Time)

Une référence à l'objet Fichier.VerifFichier a été faite :

→ VB utilise l'identificateur du programme (Program ID) pour chercher dans le registre le CLSID de l'objet.

→ VB trouve dans le registre le serveur d'objets Fichier.exe.

→ VB lit la librairie de types de l'EXE et connaît à présent les interfaces de l'objet.

→ Tout comme pour une DLL, VB stocke ces informations dans le fichier de projet afin de charger les références via la librairie de types lorsqu'on recharge le projet.

Au moment de la compilation (Compilation Time)

Puisque l'information de l'interface est disponible au moment de la conception (Design Time), tous les appels aux variables définies dans l'objet `Fichier` peuvent être faits avec une liaison précoce (early binding). VB peut donc compiler les appels aux fonctions de l'interface au lieu de dépendre de l'interface IDDispatch de l'objet.

Au moment de l'exécution (Runtime)

Lorsqu'une application essaie de créer un objet Fichier, elle utilise le CLSID de celui-ci en cherchant le nom de l'EXE implémentant l'objet dans la base de registres. Ensuite :

→ VB charge Fichier.exe.

→ VB utilise les fonctionnalités d'OLE pour créer l'objet comme désiré. Quand OLE crée un objet out-of-process, il crée d'abord l'objet dans l'espace du processus de l'application, puis crée un objet proxy[3] dans l'espace de l'application cliente et retourne un pointeur sur l'interface `IUnknown` de l'objet proxy. L'objet proxy peut ensuite reproduire les interfaces de l'objet réel. VB utilise `QueryInterface` pour obtenir l'interface désirée (`_VerifFichier` par exemple) sur l'objet proxy, qui est assignée à une variable de l'objet. L'objet est ensuite créé avec le compteur de référence à 1.

---

[3] A partir d'un programme, on ne peut pas (sauf astuce du marshalling) appeler une fonction d'un autre programme et, a fortiori une fonction d'un programme d'une autre machine du réseau !
Pour résoudre le problème, un objet doit être inclus dans le composant tant chez l'émetteur (le programme client du composant serveur) que chez le récepteur (le serveur local ou distant en EXE). On parle d'objet proxy pour l'objet incorporé dans le client et d'objet stub pour l'objet incorporé dans le serveur. Le client appelle des méthodes de son proxy, celles-ci étant en tous points semblables aux méthodes du composant serveur. Ces objets proxy et stub mettent en paquet les arguments et les envoient au correspondant (éventuellement via le réseau) qui doit effectuer l'opération inverse. Cette opération de mise en paquet est appelée marshalling en anglais. Les proxy et stub sont automatiquement générés par le compilateur MIDL.

→ Les propriétés et les méthodes de l'interface de l'objet sont exécutées par appels des fonctions de l'interface de l'objet proxy. Ce dernier envoie les appels de fonctions et paramètres par marshalling, et OLE envoie les informations à l'objet réel dans l'espace d'adressage de Fichier.exe.

→ Lorsque le projet est fermé ou que la variable objet est mise à `Nothing`, le compteur de référence de l'objet passe à 0. Le programme Fichier.exe le remarque et libère la mémoire de l'objet. Quand tous les objets implémentés par le programme sont libérés, l'application se termine.

7 - Que choisir : une DLL ou un EXE ?

Nous avons vu que les DLL et les EXE, s'ils sont différents dans leur manière de s'exécuter, sont conçus quasiment de la même manière. Nous avons également noté que le code des serveurs in-process est plus rapide que celui des serveurs out-of-process. Pourquoi alors utiliser à tout prix ces derniers ? Parce qu'ils ont tout de même des avantages. Les domaines à considérer sont :

- les performances
- les opérations en arrière-plan
- l'habilité à partager des ressources sur plusieurs processus
- et enfin le multitâche (multithreading)

Performances

Ce que nous n'avons cessé de répéter jusque là – à savoir « les DLL ActiveX sont plus rapides que les EXE ActiveX » - est à nuancer. Il serait plus correct de dire que l'accès aux propriétés et méthodes de composants implémentés par des serveurs EXE est plus lent que l'accès aux propriétés et méthodes de composants implémentés par des serveurs DLL. Le code du composant lui-même s'exécute à la même vitesse. Et bien que ce temps d'accès soit plus long, que vaut-il face à un appel de fonction prenant plusieurs minutes pour s'exécuter (par exemple une grosse requête sur une base de données) ? Il est clair que dans ce cas, la différence entre quelques microsecondes et quelques millisecondes est négligeable par rapport au temps total passé dans la fonction.
Lorsqu'on estime l'impact de la performance sur le choix entre DLL et EXE, il faut également considérer le temps passé par l'application a accéder à l'objet. Ainsi, un objet qui est accédé continuellement par le programme devrait être implémenté dans une DLL, car le processus de marshalling va imposer une grosse charge sur le système (toutefois négligeable face aux performances globales du système).

Bien que pour certaines applications précises le choix entre DLL et EXE s'effectue sur les mesures de performances, il faut également considérer les domaines qui suivent pour prendre la bonne décision.

Opérations en arrière-plan

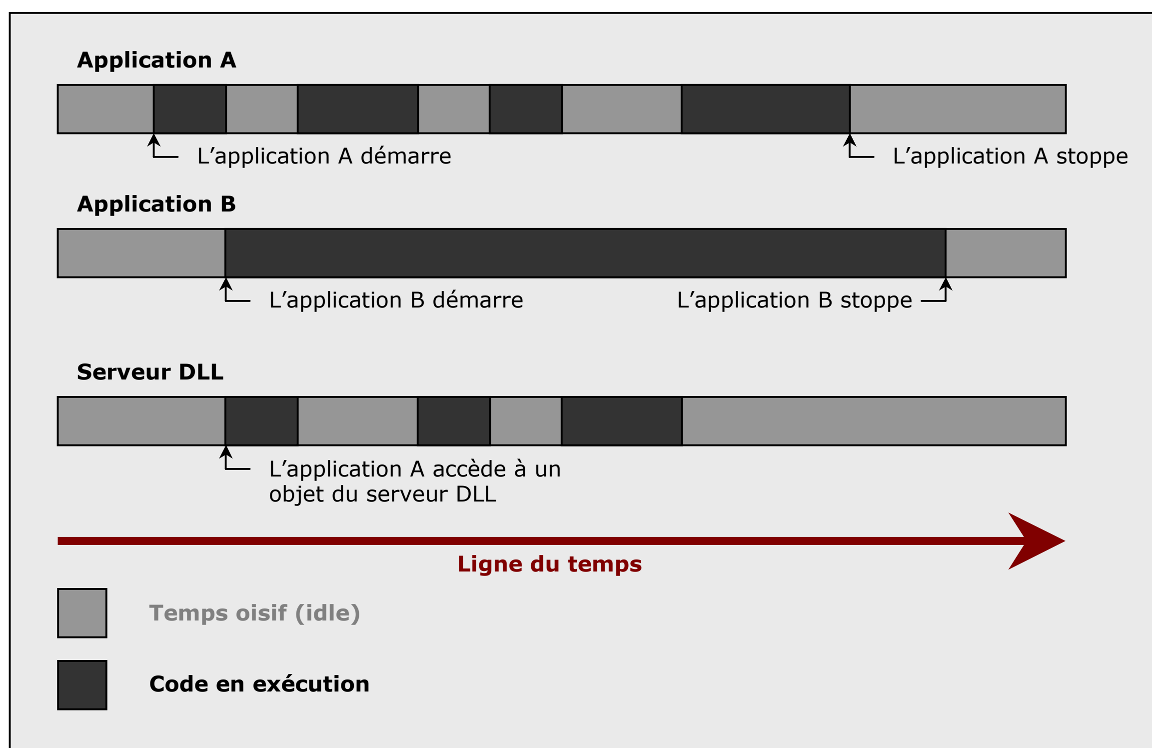
Avant de comprendre les différences entre une DLL et un EXE à ce sujet, quelques mots – brièvement, car nous n'allons pas nous étendre sur le sujet, qui prendrait plusieurs semaines d'études (le lecteur se référera à n'importe quel ouvrage sur le système d'exploitation Windows) - sur le système de threading de l'environnement Windows.

Même si on pourrait croire que Windows est un système multi-tâches (plusieurs programmes s'exécutent simultanément), il n'en est rien : au niveau le plus atomique,

seul un thread[4] à la fois est exécuté par le processeur. En fait, l'OS switche tellement rapidement entre les applications qu'il parvient à bluffer l'utilisateur. De la perspective de celui-ci, ou du programmeur, une application s'exécute comme un flot continu d'instructions traité par le processeur : c'est ce que l'on appelle un thread d'exécution. Chaque application s'exécute dans son processus, et chaque processus possède un thread (au moins). Un processus peut tout à fait avoir plusieurs threads, mais nous n'en parlerons pas dans ce rapport.

Voyons donc comment travaille le serveur DLL.

La figure suivante montre les flots de deux applications et d'un serveur DLL. L'application B s'exécute dans son propre thread et n'a aucun impact ni sur l'application A ni sur le serveur DLL. Mais, comme on peut le voir, chaque fois que l'application A exécute le code de la DLL, elle n'exécute pas son propre code. En d'autres termes, le code de la DLL s'exécute dans le même thread que le processus appelant.

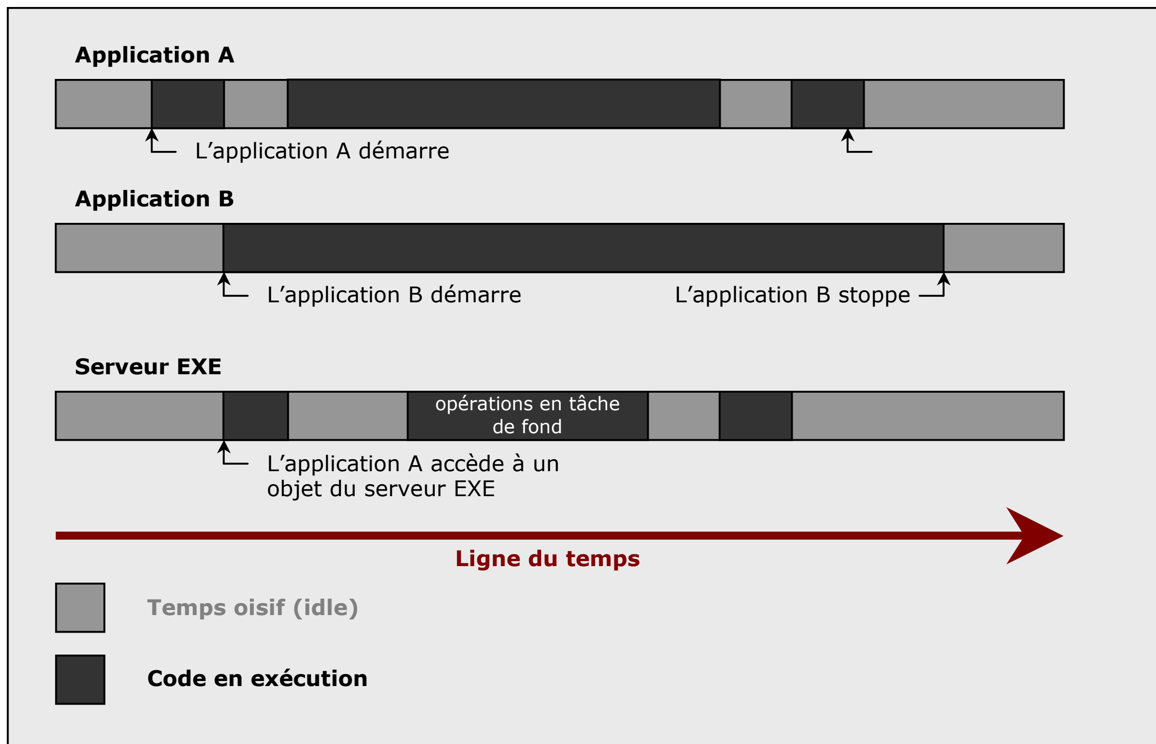


Séquence d'exécution avec un serveur DLL

[4] Difficile de donner une bonne traduction de ce terme. Néanmoins retenons « processus léger », correspondant à l'exécution d'un petit programme ou d'une routine d'un programme plus gros. En Ada, cela correspondrait au terme « tâche ». L'auteur préfère d'ailleurs utiliser ce terme dans ce rapport.

Un serveur EXE quant à lui s'exécute dans un processus séparé. Nous avons discuté de l'impact que ceci a sur les performances, du fait qu'il possède son propre espace de mémoire. Et maintenant nous savons qu'un serveur EXE s'exécute également dans son propre thread.

Ce qui a deux implications : premièrement, lorsqu'une application exécute une fonction d'un serveur EXE, les autres applications essayant d'accéder le serveur sont bloquées par défaut. Le serveur attend jusqu'à ce qu'un appel de méthode (ou de propriété) soit terminé avant que le suivant ne soit traité. Deuxièmement, une application peut lancer des opérations en tâche de fond qui seront effectuées par le serveur EXE pendant qu'elle continue d'exécuter son propre code.



Séquence d'exécution avec un serveur EXE

8 - Limitation des serveurs EXE ActiveX

Si l'on désire travailler avec un serveur DCOM et un composant EXE ActiveX, attention au nombre de clients qui accéderont au serveur ! En effet, le modèle EXE ActiveX – bien qu'il soit censé fonctionner dans un tel cas – n'est pas approprié pour de telles applications, qui créent et détruisent (rapidement) des objets[5]. Il est recommandé d'utiliser pour ce genre de conception les DLL ActiveX en relation avec un serveur MTS[6].

[5] Le développeur se trouverait confronté aux erreurs d'exécution typiques (7, 430, 424, …). Se référer au problème Q241896 dans la base de connaissances de la msdn.

[6] S'assurer que les cases Exécution autonome et Conservé en mémoire sont cochées (sous l'onglet Général des propriétés du projet), et ne pas utiliser la propriété d'instanciation `GlobalMultiUse` pour la classe (lorsque le composant ActiveX est utilisé sous MTS ou COM+).

Bibliographie critique et liens Internet

- Visual Basic developer's guide to COM and COM+, Wayne S. Freeze, Sybex 2000, 460 pages.
  L'ouvrage qui permet le mieux (parmi ceux cités dans cette bibliographie) de comprendre les technologies de composants de Microsoft, et met en relation simplement et efficacement les différents concepts. La partie consacrée à DCOM est malheureusement très courte, et n'offre qu'une petite introduction au protocole. L'auteur traite également COM+, MSMQ, et IMDB.

- Developping COM/ActiveX components with Visual Basic 6, Dan Appleman, Sams 1999, 860 pages.
  Cet ouvrage s'adresse avant tout aux développeurs professionnels sur VB, et montre les meilleures techniques pour développer des composants COM et DCOM. D'un niveau très nettement plus élevé que le titre précédent, cette « bible » (presque 900 pages !) passe en revue la conception d'objets COM mais aussi et surtout les mécanismes internes lors de l'exécution des composants. C'est à l'aide de cet ouvrage que j'ai réalisé une grande partie des explications techniques.

- http://msdn.microsoft.com, le lien INCONTOURNABLE pour développer sous plate-forme Windows.

- http://www.microsoft.com/com/tech/com.asp, la page de Microsoft pour tout ce qui concerne la technologie COM.