Formations .NET C# – Delphi.NET / Win32
Audit – Développement

Les nouvelles transactions distribuées de .NET 2.0

Abstract

Les transactions distribuées sont un ajout essentiel du framework .NET 2.0. Elles peuvent être utilisées pour valider des transactions réparties sur plusieurs serveurs à la fois. Leur force est d'être orientées Objet et donc de pouvoir s'appliquer à toute sorte de ressources, bases de données, objets, queue de messages (MMQ par exemple), etc.

En simplifiant la programmation des transactions tout en offrant un champ d'application plus vaste que les transactions purement orientées données des bases SQL, les nouvelles transactions distribuées offrent cohérence et consistance aux applications. *Elles s'appliquent aux développements simples comme aux développements sophistiqués et concernent donc tous les développeurs.*

Sommaire

Présentation

La notion de transaction

La gestion des transactions est au cœur du processus de développement l'un des aspects les plus importants. Par transaction il faut entendre un ensemble d'opérations bornées et identifiées au sein d'un groupe (la transaction) garantissant que cet ensemble est validé ou rejeté en totalité.

L'application la plus courante de ce concept se retrouve dans les bases de données relationnelles qui intègrent ce concept au niveau même du serveur. Oracle, Interbase, SQL Server et d'autres proposent la notion de transaction.

La transaction ne garantit pas seulement l'annulation ("*Rollback*") ou la validation ("*Commit*") de la totalité des opérations qu'elle encadre, elle assure aussi la cohérence des lectures des informations fixant ainsi un champ de visibilité bien déterminé à l'application ouvrant une transaction. Interbase ou SQL server 2005 proposent par exemple la notion de transaction "*snapshot*" (cliché) que d'autres serveurs n'offrent pas et qui permet de voir la base exactement telle qu'elle était à l'ouverture de la transaction quels que soient les actions des autres utilisateurs. On retrouve aussi des concepts plus fréquent tel que le "*read committed*" assurant que durant la transaction seuls les informations validées dans les autres transactions seront visibles ou bien le "*dirty read*" permettant d'accéder à toute nouvelle information même non validée.

Certaines bases vont plus loin en offrant la notion de transaction dite "*two phase commit*" (validation en deux temps) autorisant d'étendre le contexte d'une même transaction à des opérations réalisées sur plusieurs bases de données différentes (généralement limité à des bases de même type du même éditeur).

Tout cela est très utile et très puissant, même si on peut constater que beaucoup de développeurs négligent cet aspect dans leurs réalisations (expérience faite de l'auteur qui pratique des audits pour auprès de clients très divers).

Au-delà des SGBD

Depuis de nombreuses années le développement professionnel s'est dirigé vers la POO. Dès lors les données ne sont plus manipulées uniquement selon le mode client / serveur, il suffit de voir comment sous .NET par exemple les `ObjectDataSource`, l'implémentation de DAL Objet (*Data Access Layer*) et d'autres techniques modifient radicalement l'approche de la gestion des données. Rien n'a changé dans l'esprit, il s'agit juste d'élargir la notion de « données » à toute « ressource » pouvant se présenter sous la forme d'un accès effectif à une base de données autant qu'à la modification des propriétés d'un objet ou d'une grappe d'objets en

mémoire, voire sous la forme de dialogues entre plusieurs serveurs et clients dans une architecture multi-tiers.

On comprend bien que dès lors la gestion des transactions offerte par les serveurs de bases de données, même en mode « two phase commit », n'est plus suffisante ni adaptée à toutes ces opérations qui ont lieu, le plus souvent, en dehors de toute liaison à un serveur SQL.

Ainsi on trouve naturellement des solutions nouvelles à ce besoin nouveau. Microsoft propose depuis longtemps déjà MTS (Microsoft Transaction Server) et les Enterprise Services. Sous Java on trouve les JTA/JTS (Java Transaction API/Java Transaction Service) qui offrent des possibilités de même type.

Il était donc tout aussi naturel que .NET apporte sa pierre à l'édifice en offrant son lot de nouveautés tout en rendant la gestion des transactions distribuées plus simples à programmer.

System.Transactions

Descendons sans transition de ces concepts éthérés aux mains dans le cambouis…

Avec l'espace de noms `System.Transactions`, .NET 2.0 apporte une réponse aux besoins les plus larges de transaction. Plus encore, les services ici proposés forment le socle d'une gestion transactionnelle intégralement managée.

Avec cette nouvelle couche transactionnelle, Microsoft ne nous propose pas seulement une gestion distribuée réservée aux développements multi serveurs puisque les transactions purement locales sont aussi prises en compte. ***Cette nouvelle approche intéresse donc tous les développeurs, de petites ou de grandes applications, locales ou distribuées***.

Les transactions sous .NET 1.1

Avant d'entrer dans le vif du sujet rappelons comment fonctionnaient les transactions sous .NET 1.1.

La première des choses à noter est que ce système transactionnel ne s'appliquait qu'aux données et faisait donc partie intégrante du bloc ADO.NET alors que les nouvelles transactions ne sont pas liées à ce dernier.

```
string connectionString = "…";
SqlConnection conn = new SqlConnection(connectionString);
conn.Open();
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;

SqlTransaction transaction;
transaction = connection.BeginTransaction(); // début
cmd.Transaction = transaction;
try
{
   /* travail sur les données */
   transaction.Commit(); // validation
}
catch
{
   transaction.Rollback(); // annulation
}
finally
{
   connection.Close();
}
```

Code 1 - Gestion transactionnelle sous .NET 1.1 avec ADO.NET

Le code[1] 1 ci-dessus montre le mécanisme de gestion des transactions applicable sous .NET 1.1 avec ADO.NET. L'exemple utilise les classes spécialisées pour SQL Serveur.

La séquence comprend : l'obtention d'un objet transaction qui marque le début de celle-ci, le travail sur les données, la validation de la transaction. La gestion des exceptions permet de gérer l'annulation de la transaction en cas de problème.

Tout ceci est purement orienté données « SQL » et dépend du support transactionnel du serveur considéré.

Les limitations du modèle classique

Comme pointé plus haut, la première limitation de ce modèle transactionnel est qu'il est intimement lié au concept de base de données ce qui interdit son utilisation dans d'autres contextes, notamment pour des objets en mémoire.

L'autre point le plus gênant, et même en restant dans le domaine des SGBD, concerne la structure de la transaction. Dans le modèle classique vu plus haut (code 1), nous pouvons identifier facilement l'objet qui est responsable du début et de la validation de la transaction qui ne porte, de plus, que sur une seule ressource externe (une seule base de données).

[1] Les exemples ici proposés sont tous en C#. N'étant que des squelettes de code invoquant des classes du framework ils sont facilement transposables sous d'autres langages.

Mais que ce passe-t-il si plusieurs objets participent en mémoire à la mise à jour des données dans la base ?

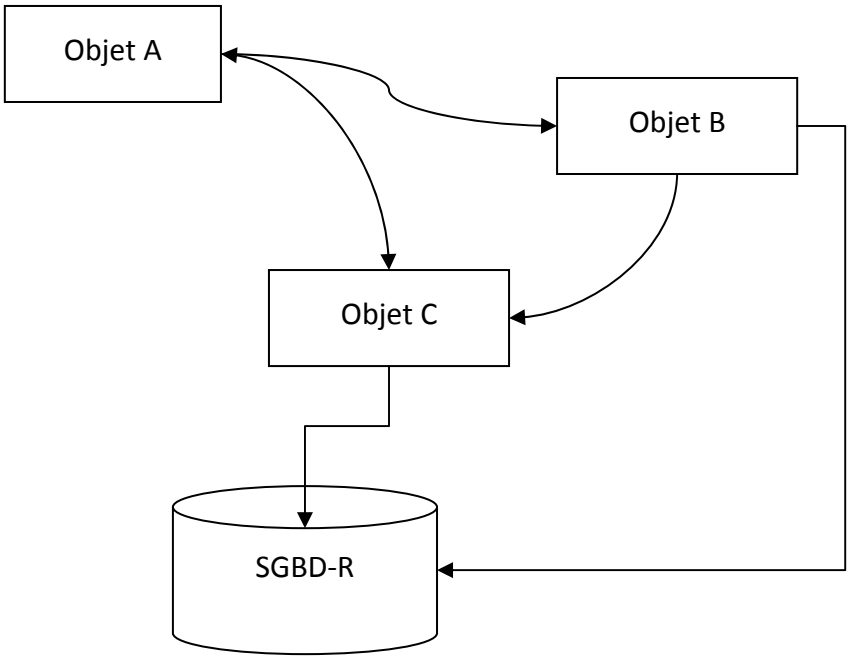

Figure 1 – Accès multi-objet à une ressource SGBD-R

La figure 1 montre un cas plus complexe où la modification et l'accès aux données sont réalisés par plusieurs instances d'objet différents (A, B et C sur le dessin).

Dans un tel contexte, lequel de ces objets sera responsable de l'appel à `BeginTransaction` ? Lequel aura pour responsabilité de valider ou d'annuler cette même transaction ?

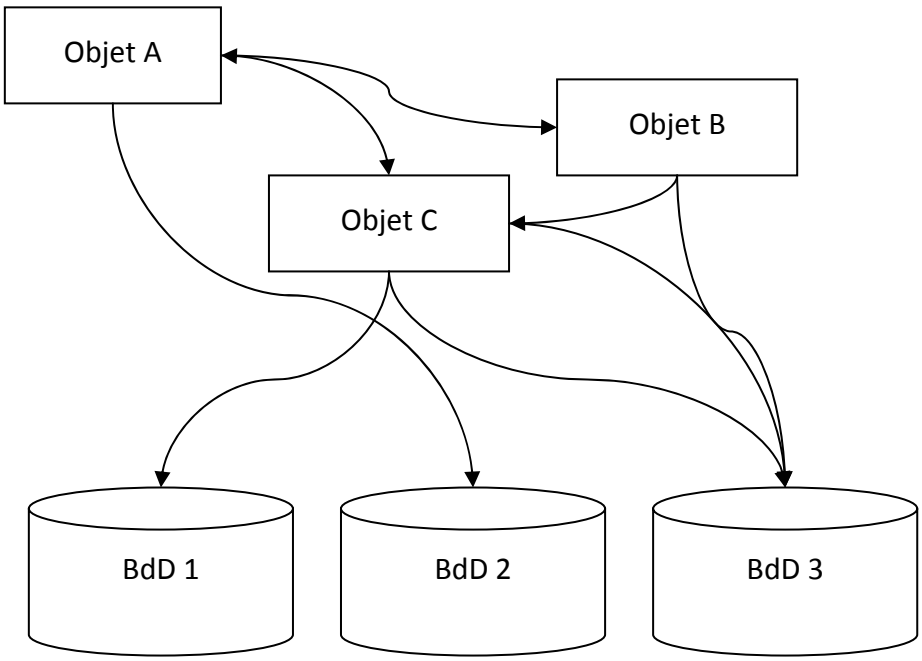

Figure 2 – Un cas encore plus complexe

La figure 2 nous montre un cas encore plus complexe où de nombreux objets interagissent avec de nombreux serveurs de données.

On comprend bien qu'arrivé à un tel niveau de sophistication, répondant pourtant à des besoins bien réels, les solutions classiques deviennent totalement inopérantes.

Les DTC et les Enterprise Services

Pour lever les limitations du modèle classique, Windows met à la disposition du développeur un service système appelé DTC pour *Distributed Transaction Coordinator* (coordinateur de transaction distribuée).

Son avantage est de porter le concept de transaction à un niveau plus élevé en gérant les transactions au travers des composants, des processus et des machines. DTC utilise le protocole OleTx (OLE Transactions).

S'il est tout à fait possible d'atteindre et de programmer DTC directement, sous .NET la façon la plus directe reste l'utilisation des services de l'espace de noms `System.EnterpriseServices`.

```
using System.EnterpriseServices;

[Transaction]
public class MonComposant : ServicedComponent
{
   [AutoComplete]
   public void MaMéthode()
   {
       /* interaction avec les ressources
          protégées par la transaction */
   }
}
```

Code 2 – Utilisation des Enterprise Services sous .NET

Comme le montre le code 2, les Enterprise Services fonctionnent sur un modèle de programmation déclaratif. La classe à protéger par une transaction descend de `ServicedComponent` et utilise l'attribut `Transaction` qui garantit que dès qu'une méthode sera appelée, elle sera exécutée dans le contexte d'une transaction. La seule chose que l'objet doit faire c'est de prévenir DTC, par le biais de la classe `ContextUtil`, que la transaction doit être validée ou annulée.

Plus simple encore, une méthode marquée par l'attribut `AutoComplete` effectuera ce travail automatiquement : en cas de succès et en fin de méthode la transaction sera validée, en cas d'erreur d'exécution (exception) la transaction sera annulée.

Les limitations des Enterprise Services

Comme on peut le constater, les Enterprise Services offrent un confort et une simplicité de programmation tout à fait appréciables, surtout lorsqu'on pense à la complexité des tâches effectuées automatiquement pour maintenir le contexte transactionnel entre toutes les ressources possibles.

Néanmoins il faut admettre que ce modèle n'est pas sans imperfections ou plutôt « lourdeurs ». Par exemple le fait d'obliger l'héritage depuis `ServicedComponent` est assez peu pratique, cet héritage prend la place de celui que l'application est en droit d'utiliser et fausse ainsi sa modélisation objet. Il devient alors impossible de supporter les Enterprises Services dans une arborescence de classes métiers sans contorsions stylistiques nuisibles.

D'autre part, l'utilisation des Enterprise Services implique l'utilisation sous jacente de transactions DTC et cela même si un seul objet et une seule ressource sont impliqués dans celles-ci. L'utilisation systématique du « two phase commit » impose son coût et grignote inutilement les performances de l'application et des serveurs.

Au-delà de ces inconvénients, les Enterprise Services reposent sur COM+ et ses travers que nombre de développeurs cherchent à fuir… On pourra aussi

noter que les stratégies de gestion des E.S. ne sont pas sans poser de souci dès lors qu'on n'opère plus sur des objets activés à la demande mais avec des pools d'objets. Enfin, ajoutons que les E.S. sont *thread-safe* et que cela implique que plusieurs threads ne peuvent participés à la même transaction. Il s'agit le plus souvent d'une garantie de bon fonctionnement et d'une simplification du code, mais cela peut s'avérer être une sévère limitation dans certains contextes.

Les transactions sous .NET 2.0

Devant la situation dépeinte plus haut il fallait trouver une solution si ce n'est plus élégante (DTC et les Enterprises Services forment une solution qui l'est plutôt assez), au moins plus souple et ne souffrant pas des défauts que nous avons listés.

Pour ce faire Microsoft a introduit deux nouveaux gestionnaires de transaction ainsi qu'un espace de nom dédié à leur gestion.

Les nouveaux gestionnaires sont le LTM (*Lightweight Transaction Manager*, gestionnaire léger de transaction) et OTM (*OleTx Transaction Manager*).

LTM & OTM

LTM a été conçu pour gérer les transactions à l'intérieur d'un seul *App Domain*[2] et n'utilisant qu'une seule ressource.

OTM s'occupe lui de gérer les transactions s'étendant sur plusieurs App Domains, voire sur plusieurs processus ou machines. Il est aussi activé dans le cadre de transactions vivant dans un seul App Domain mais invoquant plusieurs ressources.

Techniquement OTM utilise RPC (*Remote Procedure Call*, appel de procédure distante) pour les appels inter machines et se comporte finalement de façon assez proche de DTC dont nous avons parlé plus haut.

L'avantage de cette séparation entre LTM et OTM est que lorsque les transactions sont simples (un App Domain et une ressource) la totalité de la gestion réside dans l'App Domain considéré ce qui est beaucoup plus rapide que les invocations RPC. Or il se trouve que les transactions les plus courantes sont de ce type… du coup la nouvelle gestion de .NET 2.0 accélère la majorité des applications par l'utilisation de LTM, sans pour autant présenter de limite lorsqu'il s'agit de traverser les processus et les machines puisque là c'est OTM qui est utilisé.

S'occuper de tout cela pourrait rendre le développement plus complexe. Il n'en est rien puisque une couche de haut niveau a été implémentée par Microsoft, c'est justement `System.Transactions`.

Une des particularités de cette nouvelle gestion transactionnelle est appelée la *promotion* de transaction. Dans le cas le plus simple (un App

[2] espace d'exécution d'une application .NET.

Domain, une ressource) c'est une transaction de type LTM qui sera utilisée, mais si une ressource supplémentaire est utilisée, si un objet distant est invoqué, alors la transaction sera promue en type OTM, automatiquement. Par défaut, ce qui est modifiable par code, toute nouvelle transaction est ainsi de type LTM. Elle ne se voit promue en OTM que si le contexte l'impose.

Au final c'est donc une gestion simplifiée et unifiée, majoritairement automatique qui est proposée, ce qui rend le développement plus aisé.

Un cas simple

Pour de nombreux développeurs les transactions utilisées par leur code sont presque uniquement des transactions simples : elles vivent au sein de l'App Domain de leur application et ne font qu'invoquer une ressource unique, par exemple une base de données Oracle ou SQL Server.

Dans un tel cas c'est une transaction LTM qui sera allouée par .NET et comme la ressource utilisée est capable de gérer elle-même la transaction, celle créée par .NET se bornera à monitorer le travail effectué par la base de données.

Dans un tel cas de figure, utiliser les transactions de .NET au lieu de celle de la base de données peut sembler superflu. Ce n'est pas totalement faux, mais comme cette nouvelle gestion de transaction n'alourdit pas le travail à effectuer par la machine ou par le serveur de données, autant l'utiliser car elle garantit une grande cohérence au code (transaction SGBD ou transaction objet, plus aucune différence), permet de résoudre certains problèmes (voir la figure 1), et rend toute amélioration du code simple (par exemple utiliser deux bases de données dans la transaction sans pour autant avoir à en modifier le type et la gestion).

Les cas moins simples

Il y a deux événements qui déclenchent la promotion d'une transaction de type LTM en OTM : l'ouverture d'une connexion à autre base de données ou autre ressource dite « durable », et la transmission au-delà des limites de l'App Domain de la transaction sérialisée.

Par exemple en cas de remoting il suffit d'ajouter un paramètre de type `Transaction` à la procédure distante auquel on passera la transaction en cours (`Transaction.Current()` qui retourne `null` si aucune transaction n'est engagée).

D'autres mécanismes ou moyen déclaratifs permettent de contrôler le type de transaction (LTM ou OTM) et même dans les cas les plus ardus le nouveau système de transaction s'avère tout aussi puissant que simple.

Il y aurait encore beaucoup à dire sur la façon exacte dont tout cela fonctionne. Mais nous risquerions de lasser ceux d'entre vous plus intéressés par la pratique que par les considérations techniques. Pour ceux

que cela passionne tous les détails sur la gestion interne des transactions se trouvent sur les sites de Microsoft et dans certains articles sur le web.

Comment utiliser les transactions de .NET 2.0

Il est temps d'explorer concrètement comment utiliser cette nouvelle gestion de transaction.

Avant toute chose, rappelons que l'espace de noms à utiliser est `System.Transactions`. Pour y avoir accès dans un assemblage il suffit d'ajouter au projet la référence à la DLL .NET de même nom.

Le modèle déclaratif de programmation des transactions ne change pas et nous ne l'aborderons pas ici, il suffit d'utiliser les Enterprise Services, avec les avantages et défauts dont nous avons parlé en introduction. Comme ADO.NET a été modifié dans .NET 2.0 pour tirer profit des nouvelles transactions il n'y a rien à changer par rapport à du code .NET 1.1. Les LTM seront utilisées à chaque fois que cela est possible ce qui induira de meilleures performances. Ce mode déclaratif n'est pas forcément celui que le développeur devra aujourd'hui choisir.

TransactionScope

Microsoft appelle modèle explicite de programmation des transactions le fait d'utiliser volontairement dans le code les possibilités de l'espace de noms `System.Transactions`.

Le moyen le plus simple de tirer profit des nouveaux mécanismes passe par la classe `TransactionScope`, qui, comme son nom l'indique, permet de créer un espace transactionnel pour une section de code.

Lorsqu'une instance est créée une transaction est ouverte (en mode LTM par défaut) et cette instance est placée dans le champ de classe `Current` de la classe `Transaction` ce qui permet de pouvoir s'y référer à tout moment et depuis un point d'accès unique.

Comme `TransactionScope` est une classe qui supporte `IDisposable`, la transaction se terminera lorsque la méthode `Dispose()` sera appelée explicitement ou bien implicitement par le biais du mot clé `using`.

```
Using (TransactionScope scope = new TransactionScope())
{
   /* Manipulation de la ressource durable ici */
   scope.Complete(); // aucune erreur, validation
}
```

Code 3 - TransactionScope

Le code 3 ci-dessus montre la façon la plus simple d'utiliser les nouvelles transactions. La ressource durable sera le plus souvent une base de données dans les applications de gestion. Mais si le code utilise deux

connexions demain, il marchera quand même en mode transactionnel par la magie de la promotion automatique de la transaction LTM en OTM.

Dead-lock

Si le code de la transaction prend trop de temps pour s'exécuter cela peut signifier qu'il existe un dead lock, ou verrou mortel en français. Pour ne pas se laisser piéger par une telle situation la transaction sera automatiquement annulée passé un certain temps. Ce laps de temps dédié à la transaction est un timeout paramétrable qui est fixé à 60 secondes par défaut. On peut modifier ce temps par programmation ou par déclaration dans un fichier de configuration.

Réglage du timeout

Le timeout peut être spécifié lors de la création de l'instance de `TransactionScope` :

```
TimeSpan timeout = TimeSpan.FromSeconds(25);
using(TransactionScope scope = new ↵
   TransactionScope(TransactionScopeOption.Required,
   timeout))
{...}
```

Code 4 - Régler le timeout

Un timeout de zéro définit un timeout infini. Cela n'est à réserver qu'en mode de débogage lorsqu'il faut pouvoir inspecter le code avec des points d'arrêt sans que le timeout par défaut ne s'enclenche en raison des pauses effectuées dans le traitement.

On peut aussi être amené à poser un timeout très court, de l'ordre de quelques millisecondes, pour forcer la transaction à échouer ce qui est très pratique en débogage pour vérifier comment le code gère les exceptions en cas d'échec.

Si une transaction imbriquée rejoint l'AT[3] (voir plus bas) en fixant une valeur de timeout inférieure à celle de l'AT c'est cette valeur qui devient le timeout de l'AT. Si la valeur est plus grande, cela n'a pas d'effet sur l'AT.

La modification de la valeur du timeout peut avoir d'autres motivations (comme éviter des dead-locks) mais il faut savoir que cela ne doit pas être fait à la légère en raison des implications par toujours évidentes à cerner au premier coup d'œil…

Validation

L'objet transaction n'a en lui-même aucun moyen de savoir si la transaction doit être annulée ou validée. Dans certains cas comme le timeout l'annulation est choisie automatiquement mais cela reste un cas particulier.

[3] Ambient Transaction, transaction courante dans la portée.

Pour marquer le succès d'une transaction l'objet `TransactionScope` possède un champ booléen nommé `consistency` qui est à false à l'initialisation de la transaction. Lorsqu'il passe à true la transaction est validée (Commit). Plutôt que de manipuler `consistency` il est conseillé, comme dans le code exemple 3 plus haut d'appeler la méthode `Complete()`.

Bien entendu cet appel est unique sur une transaction donnée. L'appeler une seconde fois lèvera une exception `InvalidOperation`.

Mort d'une transaction

Lorsqu'un objet `TransactionScope` n'est pas utilisé dans un bloc using et si rien n'est fait volontairement l'objet va à un moment donné sortir de la portée et donc devenir éligible à sa destruction par le ramasse-miettes. Si GC qui finalise l'objet ou bien si le timeout intervient avant, la transaction sera tuée. L'état des données protégées par la transaction à ce moment dépendra du champ booléen `consistency`. S'il est à false, la transaction sera annulée, s'il avait été placé à true, notamment par un appel à `Complete()`, la transaction sera validée.

Il est bien entendu déconseillé de programmer les transactions de cette manière.

Exceptions

Le fait d'appeler `Complete()` pour valider une transaction ne garantit en rien que cette validation va s'effectuer correctement. Les ressources engagées dans la transaction peuvent refuser la validation pour des tas de raisons. S'il s'agit d'une base de données, le `Commit` qui sera envoyé au serveur peut échouer car les données insérées, modifiées ou détruites violent des contraintes par exemple.

Dans un tel cas l'objet transaction lèvera une exception de type `TransactionAbortedException`. Une programmation correcte des transactions se doit de détecter ce cas et d'en avertir l'utilisateur ou de prendre des décisions visant à corriger la situation.

Le code exemple 5 ci-dessous montre comment utiliser les exceptions dans une gestion de transaction :

```
try
{
   using(TransactionScope scope = new TransactionScope())
   {
       /* manipulation des ressources durables */
       scope.Complete(); // validation
   }
}
catch(TransactionAbortedException e)
{
   MessageBox.Show(e.Message);
}
catch //autres exceptions
{
   Trace.WriteLine("La transaction a été annulée.");
   throw;
}
```

Code 5 - Gestion des exceptions dans une transaction

Le niveau d'isolation

Le niveau d'isolation d'une transaction conditionne la visibilité des informations validées et non validées dans les autres transactions en cours.

Certains constructeurs de `TransactionScope` acceptent un paramètre permettant de fixer ce niveau d'isolation qui peut prendre les valeurs suivantes :

```
public enum IsolationLevel
{
   ReadUncommitted,
   ReadCommitted,
   RepeatableRead,
   Serializable,
   Unspecified,
   Chaos,
   Snapshot
}
```

Code 6 - Niveaux d'isolation d'une transaction

Les niveaux définis par .NET sont ici ceux qu'on a l'habitude de retrouver dans les bases de données. Leur signification est la même (même si les transactions .NET ici discutées peuvent concernées des ressources autres que les bases de données).

Le niveau `Chaos` définit un mode spécial dans lequel aucune isolation n'existe (d'où son nom).

Le niveau `SnapShot` est notamment implémenté par SQL Server 2005.

L'utilisation d'un mode autre que `Serializable` implique une excellente maîtrise des concepts transactionnels ainsi que du fonctionnement des ressources durables invoquées. Il faut aussi noter que tous les gestionnaires de ressources n'implémentent pas forcément tous les niveaux proposés par .NET, un autre niveau est alors substitué ce qui peut entraîner une inconsistance aux conséquences imprévisibles.

Les transactions imbriquées

Il est tout à fait possible d'imbriquer des transactions. Cela se fait le plus simplement comme le code exemple 7 ci-dessous le montre :

```
using(TransactionScope scope1 = new TransactionScope())
{
   /* code */
   using(TransactionScope scope2 = new TransactionScope())
   {
       /* code */
       scope2.Complete();
   }
   /* code */
   scope1.Complete();
}
```

Code 7 - Transactions imbriquées

Ce type d'imbrication est dit direct car l'imbrication est implémentée volontairement dans une même section de code.

Il existe aussi des imbrications dites indirectes. Cela arrive lorsqu'une méthode qui a ouvert une transaction en appelle une autre qui fait de même. Le code exemple 8 ci-dessous montre un tel cas :

```
void Methode_A()
{
   using(TransactionScope scope = new TransactionScope())
   {
       /* manipulation des ressources durables ici */
       Methode_B(); // imbrication de Trans. indirecte.
       scope.Complete();
   }
}

void Methode_B()
{
   using(TransactionScope scope = new TransactionScope())
   {
       /* manipulation des ressources durables ici */
       scope.Complete();
   }
}
```

Code 8 - Imbrication indirectes de transactions

Il peut exister autant de niveaux d'imbrication que nécessaire, directs et indirects. La transaction la plus externe est appelée la racine.

Lorsque des transactions sont imbriquées il se pose immédiatement la question de savoir comment le devenir de chacune des transactions internes va influer sur celui de la transaction racine. Comme cela peut dépendre du contexte, l'objet TransactionScope fournit plusieurs versions de son constructeur qui acceptent un paramètre de type TransactionScopeOption qui est une énumération définissant les choix suivants :

- `Required`
- `RequiresNew`
- `Suppress`

L'effet de chaque option dépend de l'éventuelle transaction en cours. Pour mieux comprendre nous avons regroupé ci-dessous les combinaisons dans un tableau. Ce qui est appelé *ambient transaction* (notée AT plus loin) est la transaction éventuellement en cours qui existe ou non lorsque la création d'un nouvel objet TransactionScope intervient. Le paramètre (son effet) s'applique à ce nouvel objet `TransactionScope`.

Option	Ambient transaction (AT)	
	Existe	N'existe pas
Required	‣ Rejoint AT	‣ Création d'une transaction. ‣ Devient la racine dans sa portée.
RequiresNew	‣ Création d'une nouvelle transaction. ‣ Devient la racine dans sa portée.	
Suppress	‣ Ne participera jamais à une transaction en cours. ‣ L'AT dans sa portée sera toujours null.	

Tableau 1 - Les différentes combinaisons des options d'une transaction

Lorsqu'une transaction rejoint une transaction en cours, elle ne met pas fin à l'AT à laquelle elle participe lorsque `Dispose()` est rencontrée. C'est à la transaction racine de décider.

Lors de la création d'un objet `TransactionScope`, et si l'option n'est pas précisée dans le constructeur, c'est la valeur `Required` qui est utilisée par défaut.

Les autres options prennent leur intérêt dans des cas particuliers. Par exemple l'option `Suppress` est adaptée à des sections de code qui ne doivent pas faire échouer l'éventuelle transaction de plus haut niveau si jamais elles échouent elles-mêmes. Il peut être parfois intéressant de créer une section non transactionnelle au sein d'une transaction et c'est dans ce cas précis qu'on utilisera `Suppress`.

Lorsqu'on utilise le mode `RequiresNew` il est important de veiller à ce que les deux transactions, l'AT et celle nouvellement créée, ne soient pas en interaction menant à une inconsistance des données protégées si l'une échoue et que l'autre est validée.

Pour terminer notons que l'appel à `Complete()` dans une transaction imbriquée, et même si celle-ci à rejoint l'AT, ne modifie pas l'état du booléen consistancy de l'AT elle-même. De fait il convient de valider chaque transaction quelle que soit l'imbrication.

Conclusion

La nouvelle gestion de transaction proposée par .NET 2.0 est un grand progrès. Plus simple à programmer que les anciennes techniques, plus

automatisé, plus économe en ressource, le nouveau modèle est pratique et puissant.

Il existe bien d'autres aspects non traités ici comme la gestion manuelle des transactions impliquant `CommittableTransaction`, ou bien les événements de `TransactionScope`, ou encore la gestion de la sécurité des transactions distribuées devant se protéger contre des attaques de type deny-of-service et bien d'autres choses encore. Chacun de ses sujets pour passionnant qu'il soit nous a semblé un peu trop pointu pour en justifier la présentation dans cet article qui se veut seulement être une découverte d'une nouveauté de .NET 2.0.

Si le sujet vous accroche, n'hésitez pas à consulter les sites de Microsoft et les articles qu'on trouve sur le Web, et surtout : n'hésitez pas à publier le résultat de vos tests personnels. C'est par le partage de la connaissance que nous évoluons tous.

En matière de technique il y a les mystificateurs qui tentent de faire croire qu'ils savent tout par leurs recherches personnelles, et il y a les autres, lucides et pas moins méritants, au contraire, qui l'avouent humblement « si je sais quelque chose, c'est que quelqu'un l'a écrit… ».

Ecrivons et partageons nos savoirs, nous avons tous à y gagner !

Liens

Vous pouvez approfondir votre compréhension des nouvelles transactions de .NET 2.0 en lisant l'information proposée aux liens qui suivent. Ils sont donnés à titre indicatif uniquement et n'engage pas l'auteur de ces lignes. Les liens sont réputés valides au jour et à l'heure où ces lignes ont été écrites.

"Introducing System.Transactions in the Microsoft .NET Framework version 2.0" – n'existe pas en français à l'heure actuelle. La trame a été partiellement utilisée pour construire l'article ici présenté.

http://www.microsoft.com/downloads/details.aspx?familyid=AAC3D722-444C-4E27-8B2E-C6157ED16B15&displaylang=en

« System.Transactions, espace de noms » - Microsoft, lien français.

http://msdn2.microsoft.com/fr-fr/library/system.transactions.aspx

« Mercredi du développement - System.Transactions - 7 juin 2006 » - Slide de Microsoft France assez complet.

http://download.microsoft.com/download/f/6/b/f6bf1900-db6f-4478-b451-56f091e678b1/Presentation.ppt