Implémenter ETW en .NET

par Franck SORIANO (Pages perso)

Date de publication :

Dernière mise à jour :

Cet article fait suite à celui sur ETW et Delphi.
Il présente comment implémenter un provider ETW en .NET.
L'accent est surtout mis sur les spécificités propres à la CLR et l'écriture de code non managé.

I - Introduction

I-A - Présentation

Dans l'article précédent (**http://fsoriano.developpez.com/articles/etw/delphi**), j'ai présenté les principes d'ETW. Ces principes ont été utilisés pour réaliser un profiler SQL en Delphi, ainsi qu'un outil générique (**ETWTraceViewer**) capable de contrôler et visualiser une trace ETW quelconque.

Ces principes peuvent être transposés tel quel à d'autres langages. Cependant, l'implémentation sur la plateforme .NET présente quelques spécificités. En effet, il faut appeler du code non managé et échanger des données entre la CLR et l'environnement natif.

Aussi, dans cet article nous allons voir comment développer un provider ETW pour .NET, en C#. On pourra alors l'utiliser pour instrumenter du code .NET, comme par exemple un site ASP.NET.

I-B - Télécharger les sources de l'article

ftp://ftp-developpez.com/fsoriano/archives/etw/dotnet/fichiers/dotnet-etw-src.zip

Ceux qui n'ont pas Delphi et ne peuvent pas compiler ETWTraceViewer peuvent le télécharger **ici**.

Vous pouvez télécharger l'article complet au format docx **ici**.

Le code source présenté compile avec **Microsoft Visual C# 2008 Express Edition**. Les exemples sont conçus pour tourner sous Windows XP 32 bits avec le Framework .NET 2.0.

On utilisera **ETWTraceViewer** pour démarrer et visualiser les traces.

II - Import de l'API ETW

La première chose à faire est d'écrire une classe pour appeler les fonctions de l'API ETW. Nous n'écrivons que le provider ETW, on peut se limiter à quelques fonctions. Il n'est pas nécessaire de convertir tout le fichier **EvnTrace.h**

II-A - Respect des structures physiques

La plupart des fonctions des API Win32 utilisent des structures en paramètre. Ces structures doivent être déclarées et initialisées en .NET. Lorsqu'elles sont transmises à une API native, elles ne sont pas converties. Aussi, il faut que le format de stockage en mémoire de la structure .NET corresponde strictement à ce qu'attend le code natif, à l'octet près.

Il faut respecter l'alignement de chaque champ et choisir des types de données CLR qui correspondent aux types de données natifs.

Il faut également être sûr que le compilateur traitera nos structures telles qu'elles sont déclarées, sans ajouter d'octets de calage pour aligner les données en mémoire. Pour cela, on marque chaque structure avec l'attribut **[StructLayout(LayoutKind.Sequential)]**.

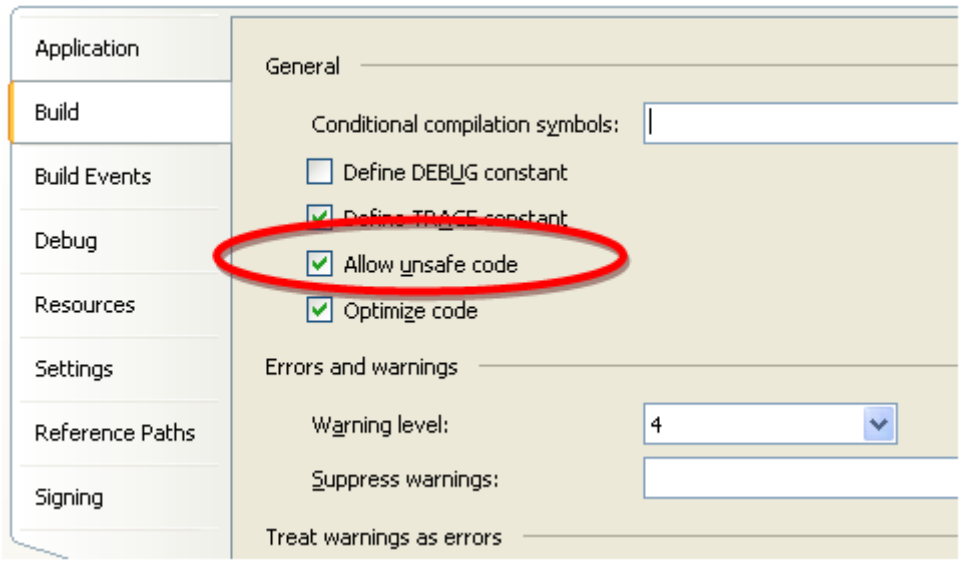
II-B - Utilisation de pointeurs : code unsafe

Certaines structures ou fonctions utilisent des pointeurs. C'est par exemple le cas de **TRACE_GUID_REGISTRATION**. Hors les pointeurs sont interdits lorsqu'on écrit du code managé. Chaque fois qu'on a besoin d'utiliser un pointeur, on doit utiliser du code dit ***unsafe***.

Le code *unsafe* est tout simplement du code que la CLR n'est pas en mesure de contrôler. Lorsqu'on utilise des pointeurs, la CLR ne peut pas savoir ce qu'on fait avec. Elle ne peut pas détecter les débordements par exemple. C'est pourquoi, le code *unsafe* est à éviter autant que possible en temps normal.

Par défaut, le compilateur interdit le code *unsafe* dans une assembly. Pour pouvoir compiler du code *unsafe*, il faut commencer par l'autoriser au niveau du projet.

Dans les options du projet, il faut cocher la checkbox ***Allow unsafe code*** sur l'onglet **Build** :



III - Le provider ETW

Pour écrire le provider, nous allons simplement porter la classe **TGenericLogger** de l'implémentation Delphi en C#.

Comme pour **TGenericLogger**, c'est le constructeur de la classe qui va s'occuper d'appeler **RegisterTraceGuids**.

```
/// <summary>
/// Crée et initialise une nouvelle instance de la classe GenericLogger.
/// Déclare le provider comme fournisseur d'événements ETW.
/// </summary>
/// <param name="pProviderGUID">GUID identifiant le provider ETW</param>
/// <param name="pEventClassGuid">GUID définissant l'EventClass des événements
/// qui seront générés.</param>
public GenericLogger(Guid pProviderGUID, Guid pEventClassGuid)
{
    // On commence par mémoriser la classe d'événement à générer.
    eventClass = pEventClassGuid;

    // On va devoir passer une référence à l'instance dans le Context de
    // la méthode RegistrerTraceGuids.
    // Il faut qu'on fournisse une référence sur un objet managé à du code non
    // managé.
    // Le GC est succeptible de déclencher entre l'appel à RegisterTraceGuids
    // et l'appel de la méthode ControlCallback. La référence fournie en retour
    // par le code non managé risque de ne plus être valide.
    // Pour éviter ce problème, on doit utiliser un GCHandle.
    loggerHandle = GCHandle.Alloc(this);

    unsafe
    {
        // Il faut définir un tableau de type TRACE_GUID_REGISTRATION pour
        // déclarer les eventClass qui seront générés. Ici on ne génère qu'un
        // seul eventClass. En guise de tableau, on peut se contenter d'une seule
        // structure TRACE_GUID_REGISTRATION.
        Etw.TRACE_GUID_REGISTRATION reg;

        // On utilise une référence sur le paramètre. Le paramètre ne peut pas
        // être collecté par le GC. Par conséquent, il ne peut pas être déplacé
        // pendant toute sa durée de vie. C'est pourquoi on peut utiliser son
        // adresse directement, sans être obligé d'utiliser un bloc fixed()
        reg.guid = new IntPtr(&pEventClassGuid);
        reg.RegHandle = 0;

        // De même, pour définir la méthode callback, il faut passer par un délégué.
        // Si on passe ControlCallback directement en paramètre à RegisterTraceGuids,
        // le compilateur va passer par un délégué anonyme et fournir ce délégué à
        // RegisterTraceGuids.
        // Hors comme on ne garde pas de référence sur ce dernier, il a toute les
        // chances d'être collecté par le GC et de ne plus être valide lorsque le code
        // non managé voudra appeler la méthode callback.
        // Pour éviter ce problème, il faut créer explicitement le délégué et
        // conserver une référence dessus pendant toute la durée de vie de l'instance.
        control = new Etw.EtwProc(ControlCallback);
        uint result = Etw.RegisterTraceGuids(control, GCHandle.ToIntPtr(loggerHandle),
            ref pProviderGUID, 1, ref reg, null, null, out hTraceReg);
        if (result != Etw.ERROR_SUCCESS)
        {
            throw new LoggerException("Erreur lors de l'appel à RegisterTraceGuids");
        }
    }
}
```

III-A - Le garbage collector n'est pas ton ami

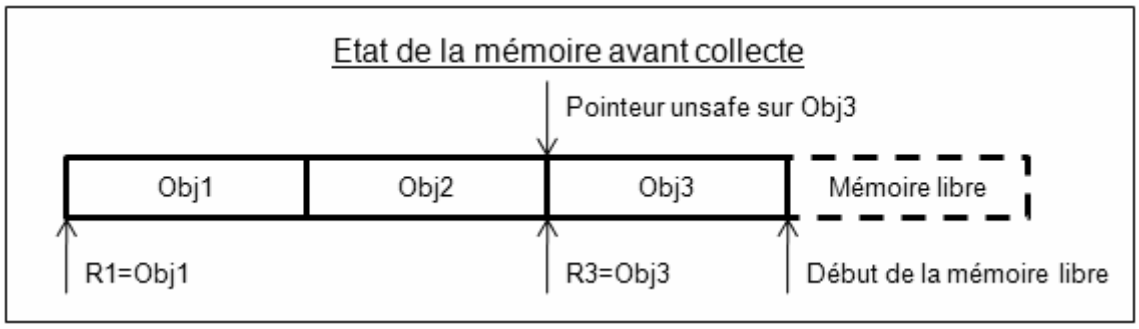
On doit faire face à trois particularités, toutes liées au GC et à la machine virtuelle.

Tout d'abord, on doit fournir le *Guid* de l'*EventClass* sous la forme d'un pointeur sur sa valeur, renseigné dans la structure **TRACE_GUID_REGISTRATION**.

Pour pouvoir définir cette valeur, on est obligé de passer par du code *unsafe*.

Le GC pose un problème particulier avec les pointeurs. En effet, à l'issu d'une collecte, le GC peut reloger les objets en mémoire afin que l'application n'utilise que des blocs mémoires continus. Du fait du mode de gestion de la mémoire dans la CLR, cette relocation est même indispensable pour libérer la mémoire inutilisée.

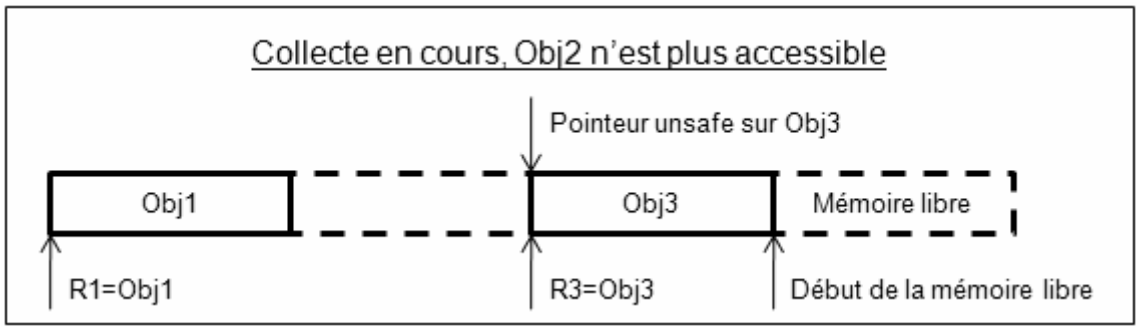
En temps normal, lorsque le GC déplace un objet en mémoire, il se charge également de mettre à jour toutes les références sur cet objet. Mais si on utilise des pointeurs et du code *unsafe*, le résultat est incertain comme le montre le schéma suivant :



Avant la collecte, la mémoire est utilisée par trois objets : **Obj1**, **Obj2** et **Obj3**. **Obj1** est référencé par **R1**. **Obj3** est référencé par **R3**. **Obj2** n'est plus référencé.

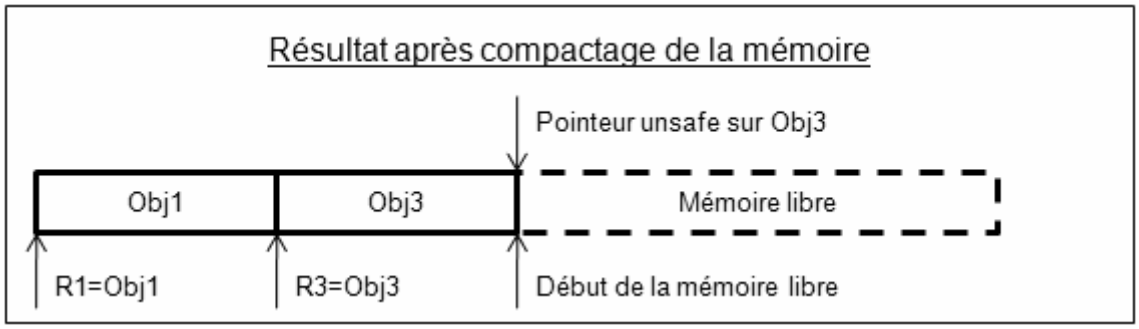
On a définit un pointeur *unsafe* qui pointe sur **Obj3**.

Lorsque la collecte déclenche, le GC va détecter que **Obj2** n'est plus accessible et va le détruire :



Sauf que s'il s'arrête là, le pointeur du début de la mémoire libre est toujours à la même place. Cela signifie que la mémoire libérée n'est pas utilisable.

Le GC va procéder à un compactage mémoire pour récupérer les trous dans la mémoire. **Obj3** est alors déplacé à la suite de **Obj1**. Les références **R1** et **R3** sont gérées par du code managé. Le GC va automatiquement les mettre à jour de sorte que la collecte est totalement transparente.



Sauf que les pointeurs *unsafe* ne peuvent pas être mis à jour automatiquement. Le pointeur *unsafe* devient invalide, ce qui risque d'entrainer un crash complet de la CLR.

Pour éviter ce problème, le compilateur interdit purement et simplement de définir un pointeur sur un élément susceptible d'être relogé.

Normalement, on ne peut définir un pointeur *unsafe* qu'à l'intérieur d'un bloc **fixed**. Dans ce cas, les objets pointés sont alors figés en mémoire, ce qui interdit au GC de les déplacer.

Dans l'implémentation Delphi, la valeur de l'*EventClass* est mémorisée dans un attribut de la classe et **TRACE_GUID_REGISTRATION** est initialisée avec une référence sur cet attribut.

En .NET on ne peut pas faire la même chose à cause du GC. La référence ne peut être définie que sur un objet **fixed**.

On peut cependant s'en sortir facilement avec une petite subtilité : Les valeurs passées en paramètre sont placées sur la pile et non pas dans le tas. De fait, elles ne sont pas gérées par le GC, elles ne risquent pas d'être relogées, elles sont automatiquement **fixed**. Aussi, au lieu de définir une variable intermédiaire **fixed** avec la valeur du guid pour ensuite la référencer dans **TRACE_GUID_REGISTRATION** on peut se contenter de référencer directement le paramètre.

III-B - Appel de la fonction callback

L'appel de la fonction callback pose également un problème. Cette dernière est écrite en C#. Il s'agit de code managé qui s'exécute dans la CLR. En revanche, elle doit être appelée en callback par du code natif : ETW.

Cet appel callback n'est pas totalement immédiat. Il s'effectue par l'intermédiaire d'un délégué (**delegate**). Ce délégué n'est pas seulement un type permettant de définir le prototype de la méthode appelée. Il s'agit bel et bien d'une classe intermédiaire utilisée pour effectuer l'appel. Elle est instanciée par le compilateur et se charge d'effectuer la transition code natif->code managé.

Si on indique directement la méthode callback lors de l'appel à **RegisterTraceGuids**, le compilateur va instancier une classe anonyme pour la donner en paramètre à **RegisterTraceGuids**. Par définition cette dernière n'est pas référencée du côté du code managé.

Seul ETW va conserver une référence qui servira au moment du démarrage ou de l'arrêt de la trace.

Il se peut que le GC déclenche une collecte entre l'appel à **RegisterTraceGuids** et l'utilisation du délégué par ETW. Si le cas se présente, le délégué aura toutes les chances d'être collecté, sans qu'ETW ne le sache. Au final ETW va essayer d'invoquer du code aléatoirement dans la CLR ce qui aura probablement pour effet de provoquer un crash de cette dernière, ou produira des effets indésirables.

Depuis le Framework 2.0, ce genre d'anomalie est automatiquement détecté grâce aux **Managed Debugging Assistants (MDAs)**.

Lorsque le GC collecte un délégué en débogage, il le remplace par une classe spéciale qui sait que la collecte a été faite, et que le délégué n'est plus valide. Si le délégué est ensuite invoqué depuis le code natif, cette classe va réagir en provoquant l'affichage d'un message d'erreur dans le débogueur. Ainsi on obtient le message suivant :

```
CallbackOnCollectedDelegate was detected
Message: A callback was made on a garbage collected delegate of type
'ETW!Fso.Diagnostic.Etw+EtwProc::Invoke'. This may cause application
crashes, corruption and data loss. When passing delegates to unmanaged
code, they must be kept alive by the managed application until it is
guaranteed that they will never be called.
```

Pour résoudre le problème, la solution est très simple : Il suffit d'instancier le délégué explicitement et de conserver la référence jusqu'à ce que **UnregisterTraceGuids** soit appelée.

III-C - Transmission de l'instance de GenericLogger à la méthode callback

La méthode callback est appelée par ETW pour notifier le provider du démarrage ou de l'arrêt de la trace. Cette méthode est une méthode statique. Cependant, elle doit transmettre cette notification à l'instance du provider. C'est ce qu'elle fait en appelant les méthodes **EnableEvents** et **DisableEvents**.

En Win32, on se servait du paramètre *Context* de **RegisterTraceGuids** pour transmettre le pointeur de l'instance à la méthode callback.

Avec du code managé, on rencontre à nouveau le même problème qu'avec tout pointeur. Ce dernier risque de devenir invalide si une collecte déclenche et que l'objet est relogé. Sauf que cette fois, il n'est pas possible de fixer l'instance en mémoire. En tout cas, pas avec un bloc **fixed**.

Pour résoudre le problème, il faut utiliser un **GCHandle**. Sur le principe, le **GCHandle** est un moyen que nous fournit le GC pour verrouiller un objet en mémoire et obtenir un pointeur qui restera valide tout au long de la vie du **GCHandle**. En fait, sans aller jusqu'à verrouiller l'objet en mémoire, le **GCHandle** permet d'associer un handle à un objet. Ce handle est un identifiant qui permettra de retrouver l'instance le moment voulu.

Pour s'en servir, on commence par l'allouer avec la ligne suivante :

```
loggerHandle = GCHandle.Alloc(this);
```

Puis on peut le convertir en pointeur **IntPtr** avec :

```
IntPtr context = GCHandle.ToIntPtr(loggerHandle);
```

Context peut alors être passé en paramètre pour appeler du code natif. La méthode callback recevra cette valeur en paramètre. Il ne reste plus qu'à faire l'opération inverse pour retrouver l'instance de **GenericLogger** :

```
GenericLogger log = (GenericLogger) GCHandle.FromIntPtr(context).Target;
```

Ce handle devra être libéré dans le destructeur avec :

```
loggerHandle.Free();
```

III-D - Ecriture d'un événement dans la trace

III-D-1 - Schéma de l'événement

Nous allons garder sensiblement le même format d'événement que pour l'implémentation Delphi. La seule particularité, c'est que .NET travaillant en unicode pour les chaînes de caractères, le message de l'événement sera également en unicode.

Le fichier MOF décrivant le provider et l'événement est le suivant :

```
#pragma namespace("\\\\.\\root\\wmi")

[dynamic: ToInstance, Description(".NET Generic Trace"),
 Guid("{2F5162CC-79F0-4181-B42F-B98B6CF8FABC}")]
class NetDefaultProvider : EventTrace
{
};

[dynamic: ToInstance, Description("MSGEVENT"): Amended,
 Guid("{9A327179-7EE6-4186-A579-2BE6BA7B95BE}"),
 DisplayName("MSG"),
 EventVersion(0)]
class NetDefaultEventClass : NetDefaultProvider
{
};

[dynamic: ToInstance, Description("MSG"): Amended,
 EventType{0, 1, 2, 3},
 EventTypeName{"INFO",
               "START",
               "END",
               "ERROR"}]
class NetDefaultEventClass_EventType1 : NetDefaultEventClass
{
    [WmiDataId(1), Extension("Noprint"),
     Description("OperationTime"): Amended, read]
    sint64 OperationTime;

    [WmiDataId(2), Description("TextData"): Amended, read,
     StringTermination("NullTerminated"), Format("w")]
    string TextData;
};
```

Comme pour l'implémentation Delphi, il faut installer le fichier MOF avec la commande :

```
Mofcomp nomDuFichier.mof
```

La structure qui sera fournie à **TraceEvent** est la suivante :

```
[StructLayout(LayoutKind.Sequential)]
public struct EVENT_GENERIC
{
    public Etw.EVENT_TRACE_HEADER Header;
    public Etw.MOF_FIELD OperationTime;
    public Etw.MOF_FIELD Message;
}
```

III-D-2 - Les chaînes de caractères

Comme précédemment, il se pose le problème du passage de pointeur sur des variables managées à du code non managé.

Nous avons vu qu'on pouvait définir un pointeur sur une valeur **fixed**. Nous avons vu qu'on pouvait utiliser un **GCHandle** pour suivre une instance d'un objet. Il reste une troisième méthode : La classe **Marshal**.

Cette dernière sert à lire et écrire dans la mémoire non managée de la CLR. On peut l'utiliser pour recopier des données dans la mémoire non managée, puis transmettre un pointeur sur cette mémoire au code natif. Elle permet également de traduire des données .NET en code natif et vice-versa. C'est ce que nous allons faire pour définir les champs de l'événement.

Nous devons initialiser une structure **MOF_FIELD** pour définir le message à écrire dans la trace. Pour cela, nous devons obtenir une référence **IntPtr** sur le message. La classe **Marshal** fournit des méthodes dédiées pour gérer les chaînes de caractères. Il suffit d'appeler la méthode **StringToBSTR** :

```
evnt.Message.DataPtr = Marshal.StringToBSTR(message);
```

La classe **Marshal** va alors allouer un bloc mémoire non managé pour stocker la chaîne de caractères, recopier *message* à cet emplacement et nous retourner un **IntPtr** référençant la nouvelle zone.

Attention : *La chaîne ainsi créée est définie en mémoire non managée ! Cela signifie que la mémoire ne sera pas libérée automatiquement par le GC. Il faut la libérer explicitement lorsqu'on n'en a plus besoin avec :*

```
Marshal.FreeBSTR(evnt.Message.DataPtr);
```

III-D-3 - Les autres données

Dans le cas général, il faut allouer un buffer dans la mémoire non managée. Nous allons utiliser cette technique pour écrire le champ **OperationTime**.

Dans un premier temps, on alloue un bloc mémoire de la taille d'un **Int64** (8 octets) dans la mémoire non managée avec :

```
evnt.OperationTime.DataPtr = Marshal.AllocHGlobal(8);
```

Ensuite, il ne reste plus qu'à écrire la valeur de **OperationTime** dans cette zone mémoire. Cette écriture s'effectue à l'aide de **Marshal** :

```
Marshal.WriteInt64(evnt.OperationTime.DataPtr, operationTime);
```

Après l'appel de **TraceEvent**, il ne faudra pas oublier de libérer la mémoire avec :

```
Marshal.FreeHGlobal(evnt.OperationTime.DataPtr);
```

Au final, l'écriture de l'événement dans la trace s'effectue de la façon suivante :

```
public void Trace(int eventType, string message, int level, long operationTime)
{
    if (IsMessageEnabled(level)) // Est-ce que la trace est active pour ce message ?
    {
        // On créer une structure pour l'événement vide.
        Etw.EVENT_GENERIC evnt = new Etw.EVENT_GENERIC();

        // On initialise l'entête de l'événement de façon standard.
        evnt.Header.Size = (ushort)Marshal.SizeOf(typeof(Etw.EVENT_GENERIC));
        evnt.Header.Flags = Etw.WNODE_FLAG_TRACED_GUID | Etw.WNODE_FLAG_USE_MOF_PTR;
        evnt.Header.Guid = eventClass; // Définition de l'eventClass
        evnt.Header.Version = 0;
        evnt.Header.Type = (byte)eventType;
        evnt.Header.Level = (byte)level;

        // Ecriture des champs utilisateurs.
        // Pour OperationTime, on alloue un bloc dans la mémoire non managée.
        evnt.OperationTime.DataPtr = Marshal.AllocHGlobal(8);
        // Puis on écrit la valeur dans ce bloc.
        Marshal.WriteInt64(evnt.OperationTime.DataPtr, operationTime);
        evnt.OperationTime.Length = 8;

        // Pour Message, on laisse la classe Marshal s'occuper de la traduction
        // de la chaîne de caractère.
        evnt.Message.DataPtr = Marshal.StringToBSTR(message);
        try
        {
            // On travaille en unicode. Donc chaque caractère fait 2 octets.
            evnt.Message.Length = message.Length * 2 + 2;

            // Ecriture du message dans la trace.
            Etw.TraceEvent(hTraceLog, ref evnt);
        }
        finally
        {
            // Il ne faut pas oublier de libérer la mémoire non managée.
            Marshal.FreeBSTR(evnt.Message.DataPtr);
            Marshal.FreeHGlobal(evnt.OperationTime.DataPtr);
        }
    }
}
```

IV - Intégration avec System.Diagnostics

System.Diagnostics propose déjà sa propre solution de tracing avec la classe **Trace**.

Il est probable que vous ayez déjà instrumenté vos applications à l'aide de cette dernière. Aussi, il serait intéressant de pouvoir rediriger les traces générées par **Trace** vers une trace **Etw**.

Pour cela, il suffit en fait d'écrire une classe **TraceListener** spécialisée et de lui demander de traiter chaque message en l'écrivant avec **GenericLogger**.

Le code de la classe est très simple :

```
public class EtwTraceListener : TraceListener
{
    // Petite optimisation pour éviter d'appeler GenericLogger.Logger() pour
    // chaque message.
    private GenericLogger log = GenericLogger.Logger();

    private StringBuilder currentLine = new StringBuilder();

    /// <summary>
    /// Ajoute un message à la suite de la ligne en cours dans la trace.
    /// </summary>
    /// <param name="message">message à écrire dans la trace</param>
    public override void Write(string message)
    {
        // On ajoute simplement le message au buffer. Il sera écrit
        // dans la trace par WriteLine.
        currentLine.Append(message);
    }

    /// <summary>
    /// Ajoute un message à la suite de la ligne encours et écrit
    /// cette ligne dans la trace.
    /// </summary>
    /// <param name="message">message à écrire.</param>
    public override void WriteLine(string message)
    {
        // On ajoute le message au buffer.
        currentLine.Append(message);

        // Ecriture de la ligne dans la trace Etw.
        log.Trace(currentLine.ToString());

        // remise à zéro du buffer puisque la ligne est écrite.
        currentLine.Length = 0;
    }

    /// <summary>
    /// Log un message d'erreur dans la trace.
    /// </summary>
    /// <param name="message">message d'erreur</param>
    public override void Fail(string message)
    {
        log.Trace(Etw.EVENT_ERROR, message);
    }

    /// <summary>
    /// Log un message détaillé dans la trace. Le message est écrit en deux
    /// lignes. Une ligne pour le message, une ligne pour le détail.
    /// </summary>
    /// <param name="message">Message d'erreur</param>
    /// <param name="detailMessage">Détail du message</param>
    public override void Fail(string message, string detailMessage)
    {
        log.Trace(Etw.EVENT_ERROR, message);
```

```
            log.Trace(Etw.EVENT_ERROR, detailMessage);
        }
    }
```

Il ne reste plus qu'à installer le listener dans le fichier de configuration de l'application :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <add name="myListener" type="Fso.Diagnostic.EtwTraceListener, Etw" />
        <remove name="Default" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

IV-A - Framework 3.5 et Windows Vista

Pour ceux qui ont Windows Vista et le Framework 3.5, vous pouvez oublier la quasi-totalité de cet article. En effet, le Framework 3.5 propose le namespace **System.Diagnostics.Eventing**. Ce dernier est tout simplement dédié à l'intégration d'Etw dans le Framework .NET.

La MSDN indique que les classes définies dans ce namespace ne fonctionnent que sous Vista. N'ayant pas Vista, je n'ai pas pu tester.

V - Evaluation des performances

A présent, comme pour l'implémentation Delphi, nous allons évaluer les performances du provider. Nous pourrons alors les comparer.

V-A - Test 1 : Ecriture de 10000 messages

Nous allons effectuer le test suivant :

```
long debut;
long fin;
long freq;
Etw.QueryPerformanceCounter(out debut);
for (int i = 0; i < 10000; i++)
{
    log.Trace("Message " + i.ToString());
}
Etw.QueryPerformanceCounter(out fin);
Etw.QueryPerformanceFrequency(out freq);

long temps = (fin - debut) * 1000 / freq;
```

Nous effectuons le test en démarrant **ETWTraceViewer** pour visualiser la trace.

Il s'exécute en **21 ms**. On peut l'effectuer avec une trace fichier sans perdre le moindre événement.

Avec l'implémentation équivalente en Delphi, le même test prenait 16 ms. On est donc environ 25% plus lent, ce qui était prévisible (pas de copie de données mémoire managé/non managée en Delphi). De plus en .NET on écrit des chaînes UNICODE deux fois plus volumineuses qu'une chaîne ANSI.

Cependant les performances restent excellentes.

V-B - Test 2 : Dégradation des performances liée à la trace

Nous effectuons à présent le test sans démarrer la trace :

```
long debut;
long fin;
long freq;
Etw.QueryPerformanceCounter(out debut);
for (int i = 0; i < 1000000; i++)
{
    log.Trace(0, "", 0, 0);
}
Etw.QueryPerformanceCounter(out fin);
Etw.QueryPerformanceFrequency(out freq);

long temps = (fin - debut) * 1000 / freq;
```

On génère 1 000 000 d'événements en **55 ms** (42 pour l'implémentation Delphi). Encore une fois, on est plus lent d'environ 25% que pour Delphi.

Cependant on peut à nouveau constater que la trace a un effet négligeable sur les performances lorsqu'elle n'est pas activée.

V-C - Test 3 : System.Diagnostics.Trace

A présent, évaluons les performances lorsqu'on utilise **System.Diagnostics.Trace** pour écrire dans la trace.

On utilise pour le cela le listener **EtwTraceListener** décrit précédemment.

On effectue le même test que précédemment, légèrement adapté :

```
long debut;
long fin;
long freq;
Etw.QueryPerformanceCounter(out debut);
for (int i = 0; i < 10000; i++)
{
    Trace.WriteLine("Message " + i.ToString());
}
Etw.QueryPerformanceCounter(out fin);
Etw.QueryPerformanceFrequency(out freq);

long temps = (fin - debut) * 1000 / freq;
```

Cette fois, les 10000 messages sont écrits dans la trace en **50 ms**. Lorsqu'on écrivait directement avec **GenericLogger** on était à 21 ms.

L'écriture des messages avec **System.Diagnostics.Trace** est donc deux fois plus lente que par la méthode directe.

Cependant, on garde un bon niveau de performances.

V-D - Test 4 : System.Diagnostics.Trace à vide

Il ne reste plus qu'à mesurer les performances si on ne démarre pas la trace.

On effectue le test suivant :

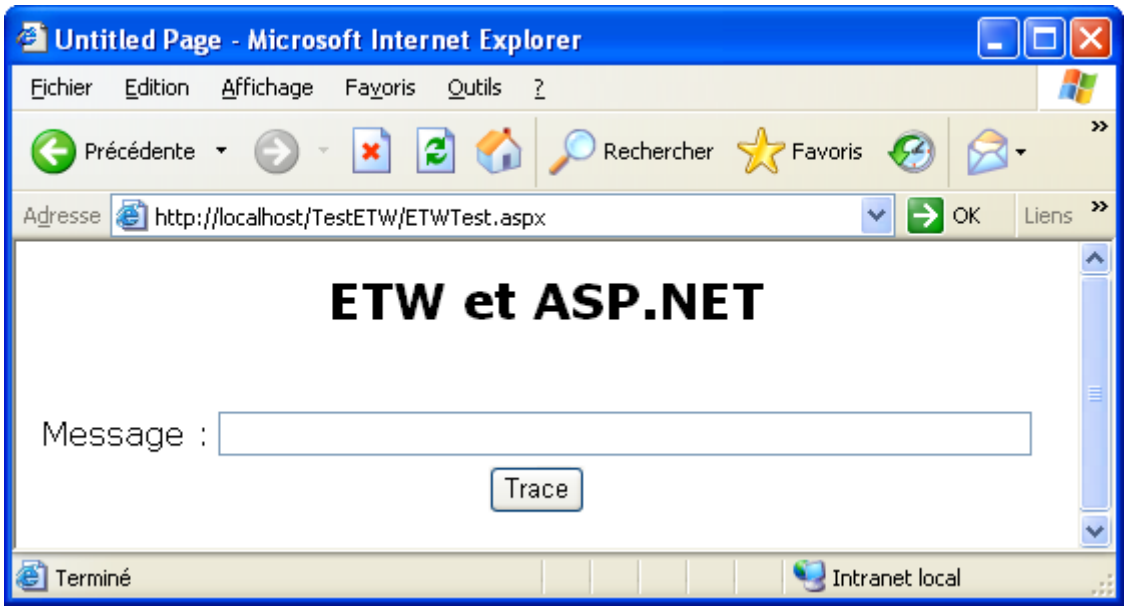
```
long debut;
long fin;
long freq;
Etw.QueryPerformanceCounter(out debut);
for (int i = 0; i < 1000000; i++)
{
    Trace.WriteLine("");
}
Etw.QueryPerformanceCounter(out fin);
Etw.QueryPerformanceFrequency(out freq);

long temps = (fin - debut) * 1000 / freq;
```

Si on laisse le listener **EtwTraceListener** en place, le test s'exécute en **285 ms** au lieu de 55 ms pour **GenericLogger**. **System.Diagnostics** est donc bien moins performant dans le cas de figure ou la trace n'est pas démarrée. Cependant 285 ms pour 1 000 000 d'événements, cela reste acceptable.

Si on fait le même test, cette fois ci en retirant **EtwTraceListener** de la configuration de l'application, le test s'effectue en **160 ms** (toujours contre 55 ms pour **GenericLogger**).

VI - Utilisation avec ASP.NET

Nous allons à présent utiliser notre provider ETW pour instrumenter une page ASP.NET très simple :



Elle se compose d'un simple **TextBox** permettant de saisir un message, et d'un bouton **Trace**. Lorsque l'utilisateur clique sur le bouton, le message saisi est écrit dans la trace.

Nous allons en plus tracer les événements **Load** et **Unload** de la page.

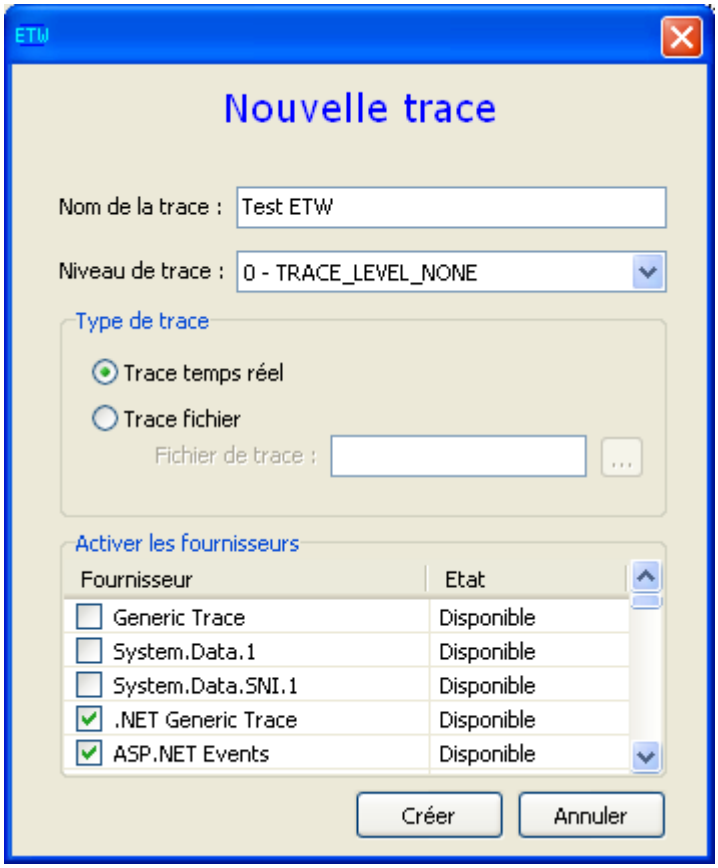
Le code behind est le suivant :

```
public partial class _Default : System.Web.UI.Page
{
    GenericLogger log = GenericLogger.Logger();

    protected void Page_Load(object sender, EventArgs e)
    {
        this.Unload += new EventHandler(_Default_Unload);
        log.Trace("Page_Load: " + this.Request.RawUrl);
    }

    void _Default_Unload(object sender, EventArgs e)
    {
        log.Trace("Unload: " + sender.ToString());
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        log.Trace(Etw.EVENT_ERROR, TextBox1.Text);
    }
}
```

On démarre une trace temps réel avec **ETWTraceViewer** en activant les providers *.NET Generic Trace* (le provider qu'on vient d'écrire) et *ASP.NET Events*.

ASP.NET Events fait parti du Framework .NET. Par contre, pour que *.NET Generic Trace* apparaisse dans la liste des providers disponibles, il ne faut pas oublier de publier le fichier MOF avec la commande :

```
Mofcomp NetLogger.mof
```



Lors de la première exécution après un redémarrage de IIS, on obtient la trace suivante :

Delta	Event	Type	User
0,0000	AspNetReq	Start	GET /TestETW/ETWTest.aspx
375,5148	.NET Generic Trace	INFO	La trace a démarré
34,3872	.NET Generic Trace	INFO	Page_Load: /TestETW/ETWTest.aspx
11,4155	.NET Generic Trace	INFO	Unload: ASP.etwtest_aspx
1,1439	AspNetReq	End	

On remarque les événements **AspNetReq** *Start* et *end*. Il s'agit d'événements générés par le moteur ASP.NET. Ils indiquent le début et la fin du traitement de la page. Si on avait exécuté le test sur un serveur IIS6 ou supérieur, on aurait également pu afficher le cheminement de la requête http sur le serveur, avant d'arriver à l'extension ISAPI qui traite les pages ASP.NET.

La première chose marquante, c'est qu'on a des temps de traitement assez importants. En tout le traitement complet de la trace page dure **422 ms**. On remarque également que l'essentiel du temps de traitement se passe avant même que la page ne soit chargée et que l'événement **Load** ne déclenche. Il s'agit en fait de l'initialisation du site et du moteur ASP.NET. Les autres temps de traitement correspondent probablement à la compilation just-in-time.

Pour s'en assurer, il suffit de rafraichir la page et de regarder la trace :

Delta	Event	Type	User
0,0000	AspNetReq	Start	GET /TestETW/ETWTest.aspx
2,078	.NET Generic Trace	INFO	Page_Load: /TestETW/ETWTest.aspx
0,2280	.NET Generic Trace	INFO	Unload: ASP.etwtest_aspx
0,4435	AspNetReq	End	

Cette fois, les temps n'ont rien à voir. La page est traitée en 2,8ms.

On remarque également que la ligne ***La trace a démarré*** n'est plus présente dans la trace. C'est parce que le provider est géré en singleton et est global au site. L'instance qui a été créée la première fois n'a pas besoin d'être recréée à chaque requête.

Maintenant si on saisit un message et qu'on clique sur le bouton :

Delta	Event	Type	User
0,0000	AspNetReq	Start	POST /TestETW/ETWTest.aspx
2,1450	.NET Generic Trace	INFO	Page_Load: /TestETW/ETWTest.aspx
0,0253	.NET Generic Trace	ERROR	Trace
0,2080	.NET Generic Trace	INFO	Unload: ASP.etwtest_aspx
0,3048	AspNetReq	End	

On voit qu'on a bien le message dans la trace.

Il n'y a plus qu'à le mettre en place dans une véritable application.

VII - Référence

Event Tracing for Windows : **http://msdn2.microsoft.com/en-us/library/bb968803(VS.85).aspx**

System.Diagnostics.Eventing : **http://msdn2.microsoft.com/fr-fr/library/system.diagnostics.eventing(en-us).aspx**

VIII - Conclusion

Dans cet article, nous avons vu comment implémenter un provider Etw en .NET tout en gardant d'excellentes performances. Ca a été l'occasion de voir comment utiliser une API native en .NET.