Lucrarea de laborator nr. 3
MSMQ

3. Cum folosim coada de mesaje în .NET

CLR oferă namespace-ul SystemMessaging, care conține clase ce înglobează funcţionalităţile oferite de MSMQ API. Acest namespace este conţinut de System.Messaging.dll, astfel încât, pentru a folosi clase pentru mesaje trebuie să stabilim o referinţă la acesta. Pentru a ne familiariza cu acesta clase vom prezenta in continuarea modul in care au fost realizate câteva aplicații pentru trimitere si primire.

3.1. Crearea aplicației care transmite mesajul

Clasa primară in namespace-ul System.Messaging este MessageQueue, aceasta având câteva metode statice care oferă posibilitatea creării si ștergerii cozilor ori oferă posibilitatea de a căuta cozi in directorul activ conform unor criterii specificate. Putem, de asemenea crea o instanță a obiectului MessageQueue care face referinţă la o coada existenta prin oferirea in constructor a caii cozii respective. Clasa are si membri pentru a realiza operațiuni cu mesaje, cum ar fi: trimiterea, primirea, eliminarea tuturor mesajelor existente sau pentru a obține diferite informații in legătura cu coada.
Exemplul următor scoate in evidenta modul in care putem folosi clasa MessageQueue pentru a verifica existenta unei cozi private, pentru a o crea si a o șterge.

```
using System;
using System.Messaging;
namespace Sender
{ class SenderMain
  { static void Main(string[] args)
    { MessageQueue mq;
      // Does the queue already exist?
      if(MessageQueue.Exists(@".\private$\NewPrivateQ"))
      {  // Yes, then create an object representing the queue
        mq = new MessageQueue(@".\private$\NewPrivateQ");
      }
      else
      {  // No, create the queue and cache the returned object
        mq = MessageQueue.Create(@".\private$\NewPrivateQ");
      }
      // Now use queue to send messages ...
      // Close and delete the queue
      mq.Close();
      MessageQueue.Delete(@".\private$\NewPrivateQ");
      Console.ReadLine();
    }
  }
}
```

Așa cum se vede din exemplul de mai sus, la o coada se face referința cu ajutorul unei cai. In cazul unei cozi private calea este de forma :

<machinename>\private$\<queuename>
unde private$ este un literal necesar in cazul cozilor private. De exemplu, următoarea linie de cod construiește un obiect MessageQueue care face referire la o coada privata numita NewPrivateQ de pe mașina locala.

```
mq = new MessageQueue(@".\private$\NewPrivateQ");
```

Pentru a specifica o coada publica eliminam literalul private$. De exemplu, următoarea linie de cod construiește un obiect MessageQueue care face referire la o coada publica numita NewPublicQ de pe mașina locală.

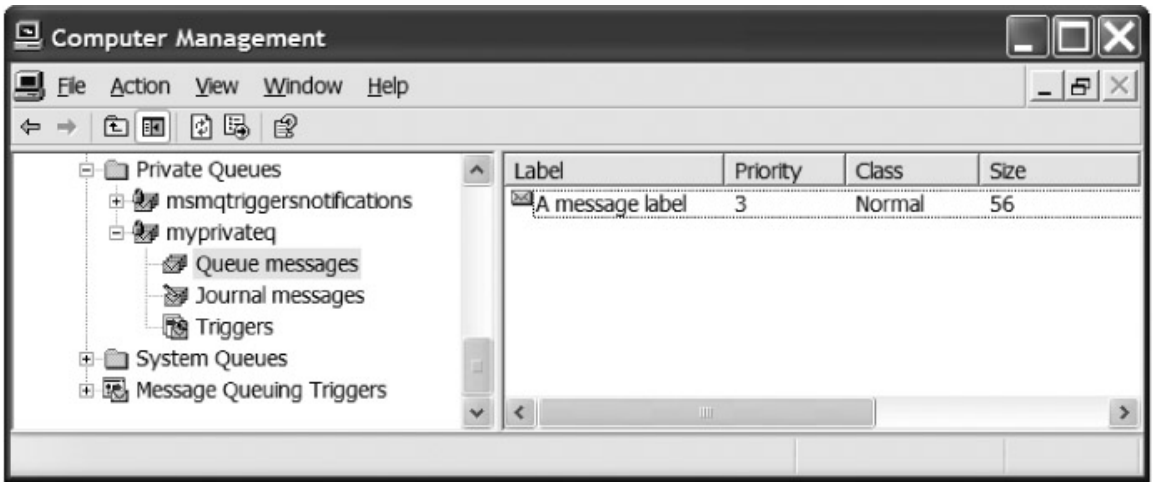
```
mq = new MessageQueue(@".\NewPublicQ");
```

3.2. Trimiterea unui mesaj simplu

Din momentul in care un obiect MessageQueue este instanţiat putem apela metoda send pentru a trimite un mesaj cozii. De exemplu:

```
static void Main(string[] args)
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Send a message - XmlMessageFormatter used by default.
   mq.Send("The body of the message", "A message label");
}
```

Acest exemplu trimite un mesaj unei cozi private numita MyPrivateQ. Deoarece primul parametru a funcției Send este de tip obiect putem trimite orice instanța de clasă în corpul mesajului. In cazul de fata este trimis un string simplu. Figura de mai jos prezinta continutul cozii MyPrivateQ dupa ce codul a fost executat.



Folosirea Computer Management (Computer Management (Start->Run-> compmgmt.msc)) pentru a confirma faptul ca mesajul a fost transmis.

3.3. Trimiterea mesajelor complexe

CLR ofera, de asemenea, o clasa Message care permite crearea unui mesaj si specificarea de proprietati ale acestuia. O data ce un obiect Message este construit si configurat acesta poate fi trimis unei versiuni supraincarcate a functiei MessageQueue.Send, asa cum se poate observa in exemplul de mai jos :

```
static void Main(string[] args)
```

```
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   Message msg = new Message();
   msg.Label = "A message label";
   msg.Body = "The message body";
   // This message waits on the queue for a max of 20 seconds.
   msg.TimeToBeReceived = TimeSpan.FromSeconds(20);
   // If the message times out, delete it from destination queue and
   // add and entry to the dead letter queue.
   msg.UseDeadLetterQueue = true;
   mq.Send(msg);
}
```

Acest exemplu construieste un obiect de tip message si stabileste cateva proprietati printre care Body si label. Prin setarea proprietatii TimeToBeReceivede acest mesaj va putea sa ramana in coda de mesaje pentru maxim 20 de secunde. Daca nu este citit din coada in 20 de secunde, coada il va sterge. Stabilirea valorii true pentru proprietatea UseDeadLetter face ca MSMQ sa copie mesajul intr-o coada a sistemului numita « Dead-letter messanges » inainte de a o scoate din coada destinatie. Aceasta caracteristica este utila atunci cand vrem sa vedem care mesaje nu au fost citite la timp si au fost sterse din cozile respective.

3.4. Referinta la cozi cu cai directe

Utilizarea cailor simple pentru a face referinta la cozi de mesaje publice din retea functioneaza doar daca acel calculator care trimite ruleaza MSMQ in modul domeniu. In acest caz, cererea de deschidere a cozii determina o consultare prealabila a Active Directory server pentru a certifica existenta cozii si pentru a se determina locatia acesteia in retea.

De asemenea putem sa ne referim la cozi publice sau private in modul workgroup sau chiar atunci cand calculatorul este deconectat de la retea cu ajutorul unei cai directe. Atunci cand se deschide o coada cu ajutorul unei cai directe, MSMQ nu consulta Active Directory server, trecand direct la coada specificata in cale.

Caile directe pot avea multe forme, insa, in toate cazurile, calea directa trebuie sa aiba prefixul "FORMATNAME:DIRECT=". De exemplu, codul urmator face referire la o coada private de pe un calculator denumit interlap1:

```
MessageQueue mq;
mq = new MessageQueue( @"FORMATNAME:DIRECT=OS:interlap1\private$\MyPrivateQ");
```

Alte exemple de cai:

```
string directPath;
// Refer to a private queue. Use the underlying OS network
// computer naming scheme
directPath = @"FORMATNAME:DIRECT=OS:interlap1\private$\MyPrivateQ";
// Refer to a public queue. Refer to machine using IP address
directPath = @"FORMATNAME:DIRECT=TCP:157.13.8.1\MyPublicQ";
// Refer to queue using a URL (Windows XP only)
directPath = @"FORMATNAME:DIRECT=HTTP://thewebserver/msmq/PublicQ";
```

Ultimul exemplu prezinta o particularitate: daca o coada este gazduita pe un calculator care ruleaza Windows XP ii putem trimite mesaje utilizand HTTP. Acest fapt poate fi folositor atunci cand vrem sa trimitem un mesaj printr-un firewall.

3.5. Construirea aplicatiei de receptionare

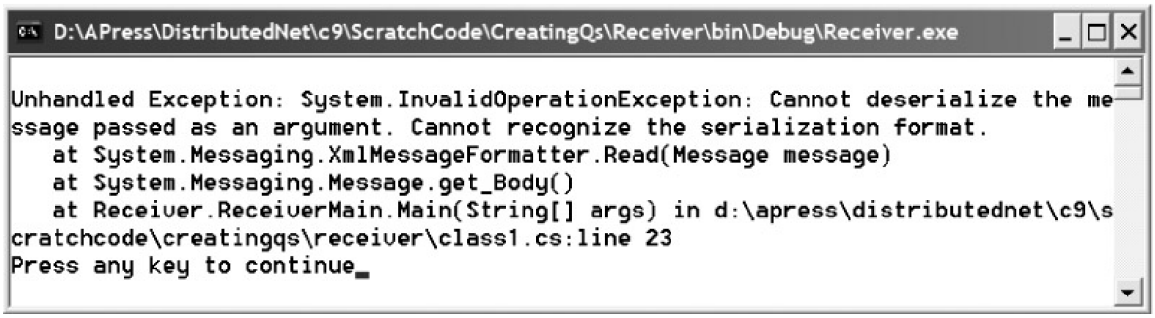
Crearea unei aplicatii care sa monitorizeze coada si sa citeasca mesajele pe masura ce acestea sosesc este de complexitate mai mare decat crearea aplicatiei de trimitere. Apar doua probleme: mai intai aplicatia care receptioneaza trebuie sa stie cum sa interpreteze corpul mesajului deoarece MSMQ nu impune structura corpului mesajului, ceea ce permite folosirea oricarui format atat timp cat aplicatiile care il trimit si il receptioneaza il inteleg.
In al doilea rand, aplicatia care receptioneaza trebuie sa aiba un sistem eficeint de monitorizare a cozii. Sunt disponibile mai multe optiuni pentru monitorizare, pornind de la blocarea la operatiunea de citire a cozii pana la sosirea mesajului si sfarsind cu raspunderea la un eveniment atunci cand mesajul soseste.

3.6. Folosirea XmlMessageFormatter

In ceea ce preiveste prima problema, si anume cea a interpretarii corpului mesajului, .NET messaging ofera trei formatatori de mesaje care serializaza tipurile CLR in corpul mesajului : XmlMessageFormatter, BinaryMessageFormatter, si ActiveXMessageFormatter. Atunci cand trimitem un mesaj foramtatorul implicit este XmlMessageFormatter, astfel incat mesajele trimise in exemplele anterioare l-au folosit pe acesta. Cu toate acestea, atunci cand se receptioneaza un mesaj, trebuie sa se specifice in mod explicit formater-ul, asa cum se observa mai jos:

```
class ReceiverMain
{
  static void Main(string[] args)
  {  // Open queue
     MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
     // Create an array of types expected in the message body
     Type[] expectedTypes = new Type[] {typeof(string), typeof(float)};
     // Construct formatter with expected types
     mq.Formatter = new XmlMessageFormatter(expectedTypes);
     // Loop forever reading messages from the queue
     while (true)
     {  Message msg = mq.Receive(); // <-- blocks until message arrives
        Console.WriteLine(msg.Body.ToString());
     }
  }
}
```

In acest exemplu liniile de cod evidentiate stabilesc formater-ul corect XmlMessageFormatter care este construit cu un tablou din tipurile asteptate. Atunci cand este receptionat, XmlMessageFormatter compara datele din mesaj cu tipurile din tablou. Daca se potrivesc atunci acesta deserializeaza corpul mesajului. In caz contrar se genereaza exceptia din figura de mai jos. De aceea implementarea receptorului poate sa deserializeze doar mesajele care contin in corp un sting sau un numar cu virgule mobile.



XmlMessageFormatter ridica aceasta exceptie daca nu recunoaste formatul corpului mesajului.
In mod alternativ, putem construi XmlMessageFormatter trimitandu-i un tablou de string numele tipurilor asteptate. De exemplu:

```
string[] expectedTypeNames;
expectedTypeNames = new String[] {"System.String", "System.Single"};
mq.Formatter = new XmlMessageFormatter(expectedTypeNames);
```

3.7. Verificarea continua (polling) a cozii

A doua problema este cea a monitorizarii cozii. In acest moment, aplicatia receptoare foloseste urmatorul cod pentru a citi coada:

```
// Loop forever reading messages from the queue
while (true)
{  Message msg = mq.Receive(); // <-- blocks waiting for a message to arrive
   Console.WriteLine(msg.Body.ToString());}
```

Asa cum indica si comnetariile, apelarea functiei MessageQueue.Receive blocheaza firul de executie pana cand un mesaj soseste in coada. In tot acest timp aplicatia nu face nimic. In cele mai multe cazuri, insa, dorim ca aplicatia sa realizeze alte operatiuni in asteptarea mesajului. Putem rezolva aceasta problema fie prin citirea periodica a cozii (un process denumit polling), fie folosind functia MessageQueue.BeginReceive pentru a realize o citire asincrona.
Clasa MessageQueue nu suporta in mod direct polling. Cu toate acestea este usor de implementat folosind alte clase cum ar fi clasa SystemThreading.Timer. Folosind aceasta clasa putem crea un fir de executie care apeleaza periodic o functie. In acest caz functia trebuie sa verifice daca sunt mesaje noi in coada. Codul urmator implementeaza acest mecanismul descris anterior:

```
static void Main(string[] args)
{  // Open queue
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Set up the formatter...
   // Construct timer to fire every 5 seconds. Note the message queue
   // reference is passed as the state object.
   Timer tm = new Timer(new TimerCallback(OnTimer), mq, 5000, 5000);
   // Simulate doing other work
   while (true)
   { Console.WriteLine("Doing other work on thread {0}",
      Thread.CurrentThread.GetHashCode());
      Thread.Sleep(1000);
   }
}
```

Se observa linia de cod care construieste un obiect de tip Timer.

```
Timer tm = new Timer(new TimerCallback(OnTimer), mq, 5000, 5000);
```

Este creat un fir de executie care apeleaza functia InTimer la fiecare 5 secunde. De asemenea este facuta si o referinta la obiectul de tip MessageQueue pe care functia OnTimer il primeste ca parametru. Pentru a implementa funcria OnTimer trebuie sa urmam semnatura definita de TimerCallback delegate. De exemplu:

```
static void OnTimer(object state)
{  // Show current thread id
  Console.WriteLine("Checking queue for messages on thread {0}",
     Thread.CurrentThread.GetHashCode());
  // Time to check the queue, first get the queue from the state param
  MessageQueue mq = (MessageQueue)state;
  // Read queue, but only block for 1 second
  try
  {  Message msg = mq.Receive(TimeSpan.FromSeconds(1));
     Console.WriteLine(msg.Body.ToString()); }
  catch
  { // No Messages, timeout occurred
    Console.WriteLine("No new messages"); }
}
```

Aceasta implementare OnTimer trimite parametrul de stare care soseste unui obiect de tip MessageQueue si il foloseste pentru a citi coada. Dar sa analizam apelarea functiei Receive :

```
Message msg = mq.Receive(TimeSpan.FromSeconds(1));
```

Daca coada contine un mesaj atunci functia il citeste si intoarece imediat. In caz contrar, asteapta pana la o secunda aparitia unui mesa. Chiar daca aceasta blocheaza firul de executie care a apelat-o deoarece functia OnTimer se executa pe un fir separat de cel principal, aplicatia in acest timp poate realiza alte operatii. Daca nu sosete nici un mesaj in perioada de timp specificata atunci functia ridica o exceptie MessageQueueException.

3.8. Citirea Asincrona a Mesajelor

Clasa MessageQueuing urmeaza modelul delegate callback pentru a oferi capabilitati de citire asincrona a mesajelor, adica ofera functiile BeginReceive si EndReceive care simuleaza BeginInvoke si EndInvoke ale delegatului. Functia BeginReceive porneste un nou fir de executie care monitorizeaza coada pentru a vedea daca sosec noi mesaje. Atunci cand soseste un mesaj nou, firul de executie fie ridica un eveniment fie apeleaza o functie callback specificata, in functie de parametrii functiei BeginReceive. Putem apela functia EndReceive in callback sau in event handler pentru a citi mesajul din coada.

Urmatorul cod apeleaza Begin Receive folosind o functie callback:

```
static void Main(string[] args)
{ // Open queue
  MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
  // Set up formatter ...
  IAsyncResult ar = mq.BeginReceive(TimeSpan.FromSeconds(5),     /* Timeout value*/
     mq,  /* State object, the message queue*/   new AsyncCallback(OnMessageArrival) // Callback);
  // Simulate doing other work
```

```
  while(true)
  { Console.WriteLine("Doing other work ...");
    System.Threading.Thread.Sleep(1000);
  }
}
```

In acest exemplu apelul functiei Beginreceive porneste un fir de executie care monitorizeaza coada pentru 5 secunde. Daca un mesaj soseste sau daca trece acest interval de timp atunci firul de executie apeleaza functia callback OnMessageArrival, trimitand referinta la MesasgeQueue.
Implemetarea functiei OnMessageArrival trebuie sa aiba in vedere ambele situatii: expirarea timpului si sosirea mesajului si este facil de realizat folosindu-se cod care trateaza exceptia asa cum este ilustrat in exemplul de mai jos:

```
static void OnMessageArrival(IAsyncResult ar)
{  // Cast the state object to MessageQueue
  MessageQueue mq = (MessageQueue)ar.AsyncState;
  try
  {  Message msg = mq.EndReceive(ar);
    Console.WriteLine(msg.Body.ToString());
  }
  catch
  { Console.WriteLine("Timeout!"); }
  finally
  {  mq.BeginReceive(TimeSpan.FromSeconds(5), mq,
    new AsyncCallback(OnMessageArrival)); }
}
```

3.9. Trimiterea instantelor claselor definite de utilizator in mesaje

Exemplele anterioare au uzat de tipuri simple pentru a prezenta aspectele fundamentale ale problematicii cozilor de mesaje. Adevaratele capabilitati ale acesteia sunt evidentiate insa atunci cand trimitem mesaje care contin date specifice aplicatiei cum ar fi date despre clienti, date despre comenzi, date despre angajati, s.a.m.d. Formatatorii de mesaje incorporate permit translarea facila a obiecteleor ce contin date ale aplicatiei in mesaje si vice-versa. Cozile de mesaje .NET ofera urmatorii formatatori:

1. XmlMessageFormatter. Acesta este formater-ul standad care a fost utilizat si in exemplele anterioare. Asa cum sugereaza si denumirea sa XmlMessageFormatter serializeaza tipurile individualizate intr-o reprezentare XML utilizand tipurile de date din schema XML. Acest formatator este incet si creaza mesaje relative mari. Cu toate acestea mesajele pot fi partajate si intelese de alte aplicatii care ruleaza pe diferite platforme.
2. BinaryMessageFormatter. Acest formatator serializaza tipul individualizat intr-un format binary proprietar, fiind mai rapid decat XmlMessageFormatter si generand mesaje compacte. Cu toate acestea doar un destiantar implementat in .NET poate traduce cu usurinta continutul mesajelor.
3. ActiveXMessageFormatter. Ca si BinaryMessageFormatter, acesta realizeaza serializarea intr-un format binary proprietar, acest format fiind acelasi cu cel folosit de de componentele MSMQ COM. Aceste componente COM ofera functionalitate bazata pe MSMQ limbajelor COM cum ar fi Visual Basic 6. De aceea putem folosi acest formatator pentru a trimite mesaje la sau a primi mesaje de la aplicatii MSMQ scrise in Visual Basic 6.

Suplimentar, namespaceul System.Messaging ofera o interfata IMessageFormatter care paote fi folosita pentru a cea un formatator particularizat.
In continuare vom explica cum poate fi folosit fiecare dintr acesti formatatori. Exemplele vor serializa urmatoarea clasa Customer. Vom presupune ca aceasta clasa este compilata intr-un assembly denumit CustomerLibrary.dll.

```
namespace CustomerLibrary
{
  public class Customer
  {  public string Name;        // Public field
     private string mCreditCard; // Private field
     private string mEmail;      // Private field with public property
     public string Email
     {  get
              {return mEmail;}
        set
              {mEmail = value;}
     }
     public Customer(string name, string email, string ccNum)
     {   Name = name; mEmail = email; mCreditCard = ccNum; }
     // Required for serialization
     public Customer() {}
  }
}
```

3.10. Folosirea XmlMessageFormatter

XmlMessageFormatter are un comportament asemnator cu clasa XmlSerializer asociata cu serviciile Web, serializand obiecte CLR in text XML. XmlMessageFormatter insa, este optimizat pentru a serializa mesaje MSMQ. Acesta poate serializa date publice sau private daca acestea din urma sunt expuse prin proprietati publice. In cazul din urma proprietatea trebuie sa suporte scriiere si citire, adica sa implementeze blocuri get si set.
Urmatoarele linii de cod prezinta partea de trimitere de mesaje care serializeaza clasa Customer intr-un corp de mesaj MSMQ.

```
static void Main(string[] args)
{  // Create the queue instance
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   Message msg = new Message();
   msg.Label = "A Customer Message";
   do
   { // Construct Customer and send to queue
      msg.Body = new Customer("Homer", "hsimpson@atomic.com", "5555");
      mq.Send(msg);
   } while(Console.ReadLine() != "q");
}
```

Asa cum se poate observa, acest cod atribuie proprietatii Message.Body o instanta de clasa Customer. Ca rezultat, XmlMessageFormatter serializeaza automat obiectul intr-un corp de mesaj. Datele clasei serializate arata in felul urmator:

```
<?xml version="1.0"?>
<Customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Name>Homer</Name>
  <Email>hsimpson@atomic.com</Email>
</Customer>
```

Campul privat mCreditCard nu este serializat.

Pentru a deserializa acest mesaj, codul care il receptioneaza trebuie sa construieasca XmlMessageFormatter astfel incat sa astepte mesaje caresa contina datele despre clienti serializate. Ca si in cazul tipurilor simple, putem sa specificam ca asteptam tipul Customer cu ajutorul operatorului typeof, asa cum reiese din exemplul urmator:

```
static void Main(string[] args)
{ // Open queue
   MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
   // Create an array of types expected in the message body
   Type[] expectedTypes = new Type[] {typeof(CustomerLibrary.Customer)};
   // Construct formatter with expected types
   mq.Formatter = new XmlMessageFormatter(expectedTypes);
   // Receive message and
   Message msg = mq.Receive();
   // Deserialized body into customer object
   Customer cust = (Customer)msg.Body;
   // Process customer data ...
}
```

Deoarece acest cod face referinta in mod direct la tipul Customer se compileaza doar daca proiectul face referinta la assembly-ul CustumerLibrary. In consecinta trebuie sa distribuim assembly-ul atat aplicatiei care trimite cat si celei care receptioneaza. Alternativ, putem specifica tipurile de mesaj asteptate sub forma unui tablou de string in care fiecare string contine numele complet al tipului:

```
// Create and array of expected type names
string[] expectedTypeNames =
   new String[] {"CustomerLibrary.Customer,CustomerLibrary"};
// Construct formatter with expected type names
mq.Formatter = new XmlMessageFormatter(expectedTypeNames);
```

Avantajul acestei tehnici este acela ca permite aplicatiei sa se conecteze la assembly dinamic in timpul rulari. De asemena aceasta perimte sa se determine tipurile asteptate prin programare si sa se construiasca formaterul XmlMessageFormatter necesar.

Chiar daca XmlMessageFormatter este relativ incet el are cateva avantaje incontestabile. Deoarece mesajul este XML el poate fi citit si interpretat de orice paser de XML, Cu alte cuvinte aplicatia care receptioneaza nu trebuie sa foloseasca XmlMessageFormatter pentru a deserailiza mesajul, aceasta putand sa citeasca datele brute in orice parser de XML. Deoarece proprietatea Messasge.Body incearca intotdeauna sa deserializeze continutul mesajului trebuie sa folosim proprietatea Message.BodyStream pentru a obtine continutul brut al mesajului. Aceasta proprietate returneaza

un obiect de tip System.IO.Stream care poate fi trimis unei varietati de parseri pentru procesare. De exemplu, liniile urmatoare folosesc System.Xml.XmlTextReader pentru a lista toate nodurile din mesaj:

```
// Receive message
Message msg = mq.Receive();
// Read the message body stream using the XML text reader.
XmlTextReader xtr = new XmlTextReader(msg.BodyStream);
xtr.WhitespaceHandling = WhitespaceHandling.None;
while(xtr.Read())
{   Console.WriteLine("{0} = {1}", xtr.Name, xtr.Value);}
```

Un alt avantaj al XmlMessageFormatter este acela ca nu este pretentios in ceea ce priveste tipul. Chiar daca trebuie sa ii dam o lista cu tipurile asteptate acesta doar verifica daca mesajul poate fi citit cu informatiile despre tip care i s-au oferit, nefacand validarea numelui assembly-ului, a numarului versiunii, s.a.m.d. De exemplu, sa presupunem ca exista urmatorul tip in assembly-ul receptor :

```
public struct Customer
{ public string Name;
  public string Email;
}
```

Acest tip Customer difera de cel initial in mai multe feluri, pornind chiar de la faptul ca acest tip este o structura in timp ce tipul initial era o clasa. In termenii schemei de date aceasta structura si clasa initiala sunt identice, de aceea structura poate fi folosita pentru a citit mesaje care contin date Custumer.

Folosind attribute in namespace-ul System.Xml.Serializatiom, putem defini un tip diferit pe care mai apoi il putem folosi pentru a deserializa mesajul Customer:

```
 [System.Xml.Serialization.XmlRoot("Customer")]
public struct FooBar
{  [XmlElement("Name")]
   public string Foo;
   [XmlElement("Email")]
   public string Bar;
}
```

Asa cum reiese din exemplul urmator, codul de primire nu mai trebuie sa faca referire la tipul initial Customer si, de aceea, nu mai trebuie sa se lege la assembly-ul CustumerLibrary.

```
// Create an array of types expected in the message body
Type[] expectedTypes = new Type[] {typeof(FooBar)};
// Construct formatter with expected type names
mq.Formatter = new XmlMessageFormatter(expectedTypes);
// Receive message
Message msg = mq.Receive();
FooBar foo = (FooBar)msg.Body;
Console.WriteLine(foo.Bar);
```

Chiar daca acest exemplu este putin exagerat, el demonstreaza flexibilitatea XmlMessageFormatter. Datorita acestei flexibilitati expeditorul si destinatarul trebuie doar sa se puna de acord asupra schemei de date. Atat timp cat aceasta ramane neschimbata, oricare aplicatie poate sa isi modifice versiunea proprie de tip Customer fara sa o afecteze pe cealalta aplicatie.

3.11. Utilizarea BinaryMessageFormatter

Spre deosebire de XmlMessageFormatter, formater-ul BinaryMessageFormatter foloseste un format binar compact pentru a serializa obiectul intr-un corp de mesaj. In fapt el foloseste acelasi mecanism de serializare la runtime cu .NET Remoting, ceea ce inseamna ca trebuie sa adaugam la tipuri atributul Serializable. Mai mult decat atat, fiecare camp dintr-o clase, chiar daca este privat, este serializat daca nu contine atributul NonSerializable.
De aceea formater-ul binar BinaryMessageFormatter poate serializa clasa Customer urmatoare, fiind inclus si campul privat mCreditCard:

```
[Serializable]
public class Customer
{
  public string Name;        // Public field
  private string mCreditCard; // Private field
  private string mEmail; // Private field with public property
  public string Email
  {  get {return mEmail;}
     set {mEmail = value;}
  }
  public Customer(string name, string email, string ccNum)
  {  Name = name; mEmail = email; mCreditCard = ccNum; }
  // Required for serialization
  public Customer(){}
}
```

Pentru a trimite un mesaj Custumer folosind acest formatator trebuie doar sa construim un BinaryMessageFormatter si sa il asociem fie lui MesageQueue fie fiecarui obiect de tip Message:

```
class BinarySenderMain
{
  static void Main(string[] args)
  {
    // Create the queue instance
    MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
    Message msg = new Message();
    msg.Label = "A Customer object";
    msg.Formatter = new BinaryMessageFormatter();
    do
    {
      // Construct Customer and send to queue
      msg.Body = new Customer("Homer", "hsimpson@atomic.com", "333-33-3333");
      mq.Send(msg);
    } while(Console.ReadLine() != "q");
  }
}
```

Acesta este codul de primire:

```
class BinaryReceiverMain
{
  static void Main(string[] args)
  {
    // Open queue
    MessageQueue mq = new MessageQueue(@".\private$\MyPrivateQ");
    // Construct formatter with expected type names
    mq.Formatter = new BinaryMessageFormatter();
    // Receive message
    Message msg = mq.Receive();
    // Deserialized body into customer object
    Customer cust = (Customer)msg.Body;
    // Use the object
    Console.WriteLine(cust.Email);
  }
}
```

Cu toate ca formater-ul BinaryMessageFormatter este mai rapid si creaza mesaje mai compacte decat formater-ul XmlMessageFormatter, este mai inflexibil. Atat emitatorul cat si primitorul trebuie sa aiba o copie a assembly-ului CustumerLibrary.

3.12. Folosirea ActiveXMessageFormatter

Inainte de .NET, multe aplicatii MSMQ erau construite folosind un set de obiecte com care impachetau API-ul MSMQ. Programatorii in Visual Basic, in special, se bazau pe aceste obiecte pentru a dezvolta apicatii care lucrau cu mesaje. Pentru a permite compatibilitatea cu aceste aplicatii, .NET ofera formater-ul ActiveXMessageFormatter. Acest formatator foloseste aceeasi schema de serializare cu obiectele COM MSMQ , astfel incat putem dezvolta in .NET o aplicatie care sa trimita mesaje unui receptor crealizat in Visual Basic 6 si viceversa.

Tema lab:

Construiti doua aplicatii una care sa transmita si una care sa primeasca mesaje. Aplicatia care receptionaza mesajele sa receptioneze mesajele fie prin intermediul unui callback delegate fie prin intermediul evenimentului ReceiveCompleted. Cele doua aplicatii sa vehiculeze date impachetate intr-o clasa Customer care sa contina diferite campuri. Încercați mai întâi sa folosiți XmlMessageFormatter si apoi BinaryMessageFormatter.

Tema acasa

Folosind MSMQ realizati un editor text colaborativ) ce vede-scrie-sterge unul vad toti si vice versa)