Debreceni Egyetem
Informatika Kar

Windows Communication Foundation

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Szeifert Péter
programtervező
matematikus

Debrecen
2009

Tartalomjegyzék

1 Bevezetés

A Windows Communication Foundation (WCF) egy egyesített, kiterjeszthető keretrendszer, amely biztonságos, megbízható, menedzselhető, más technológiákkal együttműködő alkalmazások írásában nyújt segítséget, a ma igényeinek eleget téve.
Míg más elosztott technológiák jól működnek egy bizonyos környezetben, addig a Windows Communication Foundation minden esetben a legjobb megoldást kínálja. Például egy komplex alkalmazásnál felmerülhet az igény egy J2EE alkalmazással való kommunikációra, amelyre az ASP.NET web service megoldást jelenthet, de ha már Windows alapú alkalmazásra történő garantált kézbesítésre van szükség, akkor a Microsoft Message Queuing megoldása elengedhetetlen. A WCF esetében egyesítve megtaláljuk az alábbi technológiákat:

- ASMX
- WSE
- Enterprise Services
- System.Messaging
- .NET Remoting
- MSMQ

A Windows Communication Foundation tervezése során az alábbi célokat tűzték ki:

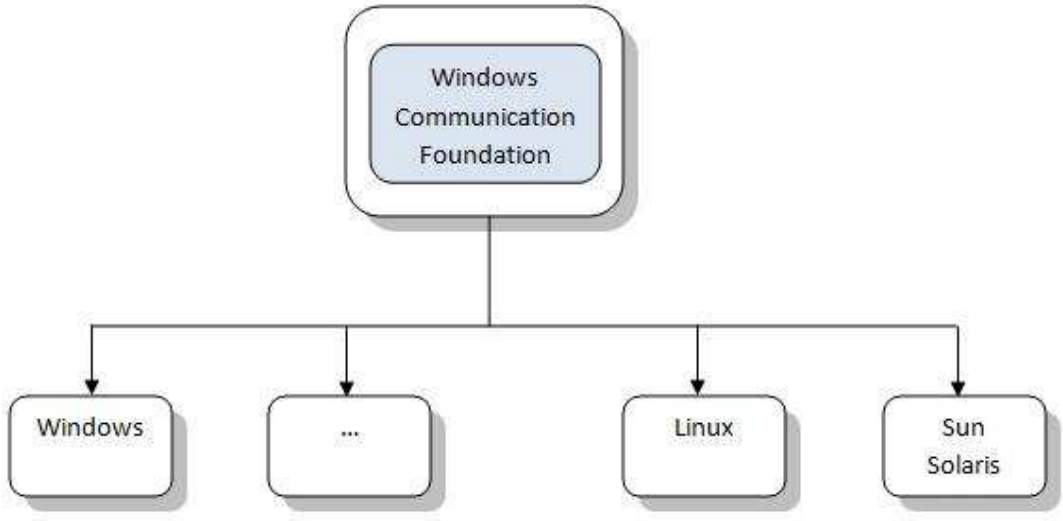
- A meglévő technológiák egyesítése
- Együttműködés más platformokkal
- Szolgáltatás orientált fejlesztés

A különböző platformokon futtatott alkalmazások közötti kommunikációt nehezíti, hogy az adott gyártó által használt protokollok és a platform szorosan összefügg. A modern elosztott enterprise rendszereknél sokszor elkerülhetetlen a különböző technológiák, platformok összekapcsolása, ezért megszületett egy egységesített webszolgáltatás specifikáció. Mivel a webszolgáltatások

nyílt szabványokat használnak, ezért lényegtelen, hogy az adott szolgáltatást milyen technológia felhasználásával írták. A WCF a SOAP üzenetküldési protokollt használja arra, hogy a kommunikációt megvalósítsa a heterogén rendszerekben.

Régebben az elosztott rendszereket olyan technológiák alkalmazásával fejlesztették, amelyek az objektum orientált paradigmára épültek, de szükség volt platformfüggetlen technológiákra. A technológia gyorsan változik, ezért a SOA (Service-oriented architecture) használatával egy alkalmazás képes lesz együttműködni más platformokkal, tűrni a változásokat és kevesebb karbantartásra szorul. Az objektum orientált szemlélet a homogén rendszereket célozza, interfészek és osztályok megosztásán alapszik, míg a SOA mind a homogén, mind a heterogén rendszereket segíti és séma, illetve contract megosztással egy absztraktabb, kényelmesebb utat mutat.

A webszolgáltatások olyan széles körben elfogadott és alkalmazott szabványokra épülnek, mint az SOAP, XML, HTTP. Az XML webszolgáltatások könnyedén elérhetőek nem Windows platformról is.



1. ábra – Együttműködés más operációs rendszerekkel

A szolgáltatás és a kliens független egymástól, azaz megállapodnak egy interfészben, amelyen keresztül kommunikálnak, de ezen felül mindegy a platform, az operációs rendszer.

A SOA szemszögéből egy szolgáltatás az alábbi alapelvekre épül:

- Explicit határok
  Olyan belépési pontként funkcionáló publikus interfészt kell definiálni, amelyen keresztül megvalósulhat a kommunikáció. Ez az interfész ne legyen nagy, illetve ne tegyen közzé belső implementációs információkat. A megvalósított algoritmus absztrakciójaként funkcionál.
- Autonomitás
  A szolgáltatások lazán csatoltak. Tudnak egymásról, de függetlenek maradnak.
- Séma és contract megosztása osztályok helyett.
  Ez elősegíti a különböző technológiák közötti kommunikációt.
- Rendszabályok segítségével szétválasztható a szerkezeti és szemantikai kompatibilitás.

A WCF-ben, mint a .NET keretrendszerben általában, a következő programozási módszerek használhatók:

- Imperatív módon, kódban
- Konfigurációs fájlokon keresztül
- Deklaratív módon, attribútumokkal

Egy szolgáltatás írásánál általában párhuzamosan használjuk ezeket a modelleket, szem előtt tartva az egyes modellek előnyeit.

A WCF magában foglal számos meglévő elosztott technológiát, de ki is terjeszti azokat. A WCF deklaratív és imperatív programozási modellje segítségével kevesebb kód írásával is elérhetjük ugyanazt az eredményt, mint annak előtte. Konfigurációs fájlok használatával a szolgáltatás

anélkül módosítható, hogy újra kellene azt fordítani, illetve a szerverre kihelyezni. Az attribútum alapú modellje pedig lehetőséget biztosít az üzleti logika és az üzenetküldést végző infrastruktúra szétválasztására.

Diplomamunkám célja a Windows Communication Foundation szolgáltatások írásához szükséges fogalmak és használatuk bemutatása, az általam fontosnak tartott technológiai lehetőségek (például aszinkron működés, tranzakciós tulajdonság) ismertetése.

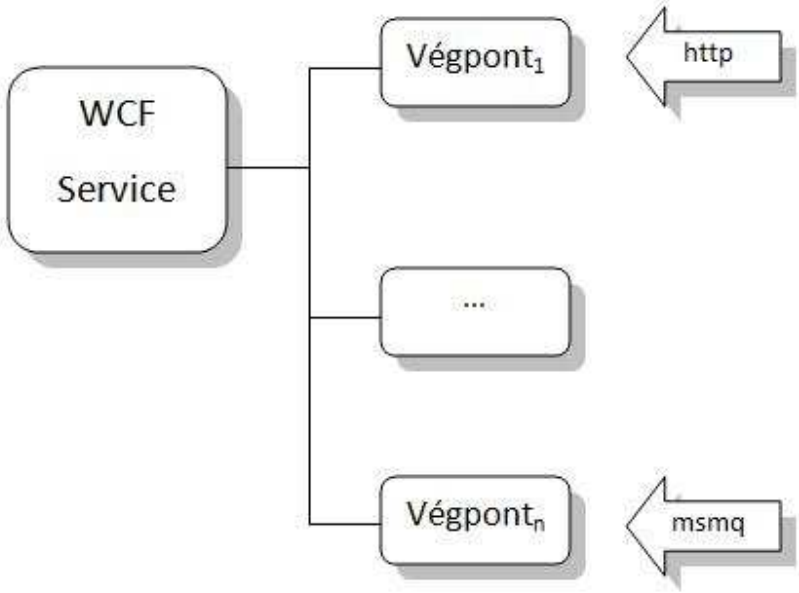
2 Végpontok

A WCF szolgáltatással történő mindennemű kommunikáció végpontokon keresztül történik, ezeken keresztül érheti el a kliens a szolgáltatás műveleteit. Egy végpontot a következő négy tulajdonságával adhatjuk meg:

- *Address*
  Cím, amely egyedileg azonosítja a végpontot és információt ad arról, hogy hol található a szolgáltatás.
- *Binding*
  Meghatározza, hogy a kliens és a szolgáltatás hogyan kommunikálnak egymással (milyen protokollon keresztül, milyen kódolással, stb.).
- *Contract*
  Meghatározza a szolgáltatás által publikált műveleteket, az adatok, üzenetek szerkezetét, az üzenetváltás irányát. A contract szó szerződést jelent, azaz egy megállapodás a kliens és a szolgáltatás között arról, hogy milyen szabályok szerint kommunikálnak egymással.
- *Behaviors*
  A végpont viselkedését írja elő.

Az első három tulajdonságot kötelező megadni. Ezeket a WCF ábécéjének is szokták nevezni. Ezt a három tulajdonságot ismernie kell a kliensnek, ha használni akarja a szolgáltatást. A *Web Service Description Language* (WSDL) egy standard eszköz arra, hogy ezen adatokat publikálni lehessen, és más platformok is megértsék. Azt adja meg, hogy a szolgáltatás hol található, hogyan férhetünk hozzá, illetve milyen műveleteket támogat.

A végpontokat megadhatjuk imperatív módon a kódban, illetve deklaratív módon, konfigurációs fájlok használatával. Egy WCF szolgáltatás több végponttal is rendelkezhet, ezáltal az egyes végpontok a fentebb említett tulajdonságaikban eltérhetnek. Például más protokollon keresztül kommunikálhat a klienssel.



2. ábra – WCF szolgáltatás több végponttal

2.1 A végpont címe (address)

A végpont címe egy *URI (Uniform Resource Identifier)*, amely megadja, hogy hol található az adott végpont. A végpont címének megadására a WCF a *System.ServiceModel.EndpointAddress* osztályt használja. A végpont címét megadhatjuk a kódban, illetve konfigurációs fájlokban. A végponttal kapcsolatos információkat célszerű a konfigurációs fájlokban beállítani, mert módosítás esetén nem kell az alkalmazásunkat újrafordítani vagy a webszerverre kihelyezni. Gondoljunk csak arra, hogy egy alkalmazás fejlesztési fázisában általában nem a végleges címet használjuk, hanem egy tesztszerveren vagy lokálisan futó szolgáltatás címét. Egy másik előnye ennek a módszernek, hogy a fejlesztés után a rendszergazda is módosítani tudja az adott beállítást anélkül, hogy értené a mögöttes implementációt.

A 3. és 4. ábra egy szolgáltatás végpontjának beállítását szemlélteti konfigurációs fájlban, illetve kódban:

```
...
<system.serviceModel>
<services>
      <service behaviorConfiguration="EmployeeServiceBehavior"
            name="Thesis.WCFServices.EmployeeService">
            <host>
                  <baseAddresses>
                        <add baseAddress="http://localhost:8080"/>
                  </baseAddresses>
            </host>
            <endpoint address="EmployeeService"
                  binding="wsHttpBinding"
                  bindingConfiguration="EmployeeBinding"
                  contract="Thesis.WCFServices.IEmployee" />
            <endpoint address="mex" binding="mexHttpBinding"
                  contract="IMetadataExchange" />
      </service>
</services>
<bindings>
      <wsHttpBinding>
            <binding name="EmployeeBinding"
                  openTimeout="00:00:30"
                  closeTimeout="00:00:30"
                  receiveTimeout="00:00:30"
                  sendTimeout="00:00:30">
            </binding>
      </wsHttpBinding>
</bindings>
<behaviors>
      <serviceBehaviors>
            <behavior name="EmployeeServiceBehavior">
                  <serviceMetadata httpGetEnabled="true" />
            </behavior>
      </serviceBehaviors>
</behaviors>
</system.serviceModel>
...
```

3. ábra – Szolgáltatás végpontjának beállítása konfigurációs fájlban

A végpont megadásánál használhatunk abszolút és relatív címet. Amennyiben relatív címet adunk meg, akkor definiálnunk kell egy báziscímet is, azonban ha *Internet Information Services (IIS)* segítségével hosztolunk, akkor ne adjunk meg báziscímet, csak relatív címet, mert az .svc fájl

címe lesz a báziscím (erről a későbbiekben még szó lesz). Ha egy végpontnak nem adunk meg relatív vagy abszolút címet, akkor a báziscím lesz az adott végpont címe. Itt azonban fontos megjegyezni, hogy a végpontok címeinek egyedinek kell lenni, tehát hibához vezet, ha minden végpontnál a báziscímet alkalmazzuk (és természetesen minden más esetben is, ahol sérül ez az egyediségre vonatkozó megszorítás).

```
…
Uri baseAddress = new Uri("http://localhost:8080");
using (ServiceHost host =
            new ServiceHost(typeof(EmployeeService), baseAddress))
{
      string relativeAddress = "EmployeeService";
      ServiceMetadataBehavior behavior =
                                    new ServiceMetadataBehavior();

      behavior.HttpGetEnabled = true;

      WSHttpBinding binding = new WSHttpBinding();

      binding.OpenTimeout = new TimeSpan(0, 0, 30);
      binding.CloseTimeout = new TimeSpan(0, 0, 30);
      binding.SendTimeout = new TimeSpan(0, 0, 30);
      binding.ReceiveTimeout = new TimeSpan(0, 0, 30);

      host.Description.Behaviors.Add(behavior);
      host.AddServiceEndpoint(typeof(IEmployee),
                              binding,
                              relativeAddress);

      try
      {
            host.Open();
            Console.WriteLine("A szerviz fut.");
            Console.ReadLine();
      }
      catch(Exception ex)
      {
            // Kivétel kezelése
      }
}
…
```

4. ábra - Szolgáltatás végpontjának beállítása kódban

Megjegyzés:
A kódban szereplő *IEmployee* interfész a *service contract*-ot reprezentálja, az *EmployeeService* osztály pedig ezen interfész implementációjával az üzleti logikát tartalmazza. Ezzel később részletesen foglalkozunk.

A *System.ServiceModel* névtér *ServiceHost* osztálya egy hosztot szolgáltat a szolgáltatásnak, melynek típusát megadtuk, báziscímet, viselkedést, illetve végpontot rendeltünk hozzá.

2.2 Binding

A *binding*-ok segítségével megadhatjuk a kommunikáció során használt protokollokat, üzenet-kódolási beállításokat, az adatok szállítására vonatkozó információkat. A *binding* binding elemek gyűjteménye, ahol minden elem a kommunikáció adott tulajdonságára vonatkozik. A WCF segítségével előre definiált *binding*-okat is használhatunk, amelyek a binding elemek megfelelő alapértelmezett értékeit képviselik, amelyeken változtathatunk, de lehetőség van saját *binding* megadására is.

A fenti ábrákon láthattuk a *binding* megadását konfigurációs fájlban, illetve kódban.

Az <*endpoint*> elem *binding* attribútumával megadtuk, hogy a WCF előre definiált *wsHttpBinding binding*-ját használja a végpont, illetve a *bindingConfiguration* attribútum a <*bindings*> elem megfelelő elemére mutat, ahol a megadott *binding* beállításain módosíthatunk.

Ha a kódban szeretnénk megadni a *binding*-ot, akkor azt a *ServiceHost* példányon hívott *AddServiceEndpoint* metódus segítségével tehetjük meg.

2.3 Service contract

A *contract*-ok közül ebben a fejezetben a *service contract*-ot ismertetem. A *Data*, *Message* és *Fault contract*-ok a későbbi fejezetekben kerülnek bemutatásra.
A szolgáltatások műveletek (operation) csoportjai, ahol a *service contract* határozza meg, hogy a szolgáltatás milyen műveleteket publikál, azaz egy publikus interfészt biztosít, amelyen keresztül igénybe lehet venni a szolgáltatást. A WCF alkalmazásoknál ezen műveletek megadása metódusok létrehozását jelenti, melyek el vannak látva *System.ServiceModel.OperationContractAttribute* attribútummal. *Service contract* létrehozásához megadhatjuk ezen metódusok specifikációját egy interfészben, amely el van látva *System.ServiceModel.ServiceContractAttribute* attribútummal vagy közvetlenül megadhatjuk implementációval együtt egy osztályban, amely szintén meg van jelölve az előbb említett attribútummal.
Célszerű az interfészek használata, mert így az interfész implementációját megváltoztatva a *service contract* változatlan marad és több *service contract* is kiterjeszthető. Azok a metódusok, melyek nincsenek megjelölve *OperationContractAttribute* attribútummal, nem szolgáltatás műveletek, nem hívhatók a kliens által, de használhatóak a műveletekben a láthatóságuknak megfelelően.

Az alábbi példában a *service contract*-ot egy interfész segítségével adjuk meg:

```
// Az interfész definiálja a szolgáltatás által
// hívható metódusokat
[ServiceContract]
public interface IEmployee
{
      [OperationContract]
      EmployeeInfo GetEmployeeInfo(string id);

      [OperationContract]
      void SetEmployeeInfo(EmployeeInfo info);
}
```

5. ábra – Service contract megadása interfészben

A contract szó jelentése szerződés, tehát a *service contract* egy megállapodás a szolgáltatás és a kliens között arról, hogy milyen metódusokat hívhat a kliens, a kommunikáció milyen irányú, stb. Ezekről szó lesz a későbbiekben.

A fenti interfész egy *service contract*-ot definiál, ahol a két *OperationContractAttribute* attribútummal ellátott metódus az adott szolgáltatás publikus interfészét jelenti majd, azaz a kliens számára elérhetővé válik.
Az alábbi ábra a fentebb definiált interfész egy implementációját mutatja, ahol implementáljuk az interfészt, azaz logikát adunk a specifikációkhoz.

```
// Az osztály ad implementációt a szolgáltatásnak
public class EmployeeService : IEmployee
{
	// Az kliens által hívható
	public EmployeeInfo GetEmployeeInfo(string id)
	{
		return GetInfo(id);
	}

	// Az kliens által hívható
	public void SetEmployeeInfo(EmployeeInfo info)
	{
		SetInfo(info);
	}

	// Az kliens által nem hívható
	private EmployeeInfo GetInfo(string id)
	{
		EmployeeInfo info = null;
		// üzleti logika
		return info;
	}

	// Az kliens által nem hívható
	private void SetInfo(EmployeeInfo info)
	{
		// üzleti logika
	}
}
```

6. ábra – Service contract implementációja

Azok a metódusok, amelyek a *service contract*-ban nincsenek ellátva *OperationContractAttribute* attribútummal, azok nem képezik a szolgáltatás és a kliens közötti megállapodás részét, ezáltal a kliens számára elérhetetlenek.
Az *OperationContractAttribute* attribútum a megjelölt metódust a *service contract* részévé teszi, és bizonyos viselkedéseket írhat elő a metódusra nézve. Például:

- hívása munkamenetet indíthat
- lezárhat egy munkamenetet
- aszinkron módon hívható
- megadja a kommunikáció irányát

A következő fejezet a szolgáltatás hosztolási lehetőségeit mutatja be.

3 Hosztolás

Ahhoz, hogy egy szolgáltatás működőképes legyen, szükség van futtató környezetre, amely kezeli a szolgáltatás életciklusát. A WCF többféle hosztolási lehetőséget támogat:

- Self-hosting (hosztolás menedzselt alkalmazásban)
- Windows Process Activation Service (WAS)
- Windows Service
- IIS

Ahhoz, hogy az adott szituációban a legmegfelelőbbet alkalmazzuk, meg kell vizsgálnunk a következő területeket:

- Elérhetőség
- Menedzselhetőség
- Verziókezelés
- Megbízhatóság

A WCF lehetővé teszi, hogy anélkül váltsunk a hosztolási módok között, hogy a szolgáltatás implementációján változtatnunk kellene. A fejezet további részében a *self-hosting* és a *WAS* hosztolási módot mutatom be.

3.1 Self-hosting

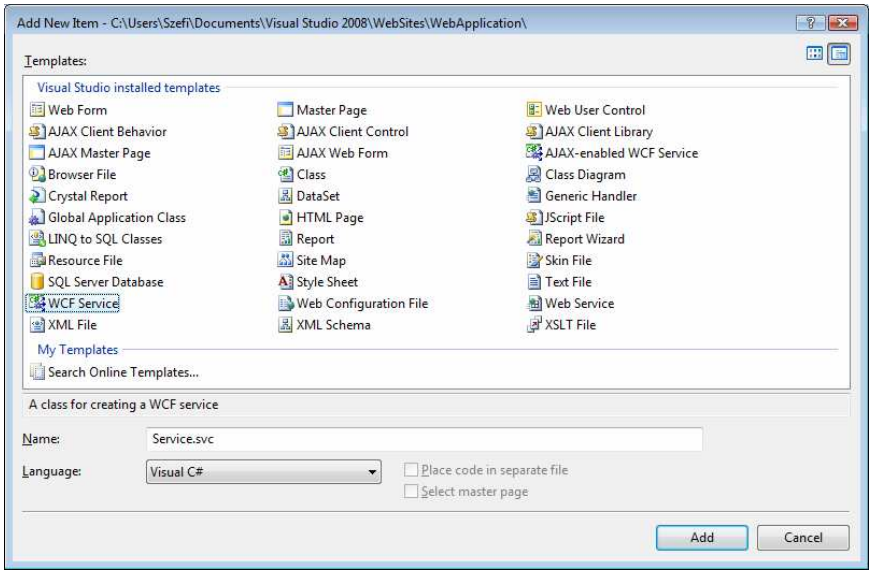
A WCF szolgáltatások szabadon hosztolhatók menedzselt alkalmazásokban. A szolgáltatás kódját beágyazzuk a menedzselt kódba, és ott gondoskodunk az életciklusának kezeléséről. Ezt a módot egyszerű alkalmazni, ugyanis elég néhány sor kódot írni hozzá, és máris futtathatjuk a szolgáltatást. Az életciklust is kezelhetjük a *ServiceHost* példány *Open* és *Close* metódusaival. A

további beállításokat (végpont, binding, stb.) is a kódban vagy konfigurációs fájlok segítségével adjuk meg. A hibakeresés leegyszerűsödik, ugyanis az adott alkalmazás hosztolja a szolgáltatást, azonban a hátrányai közé tartozik, hogy a szolgáltatás csak akkor elérhető, ha a hosztoló alkalmazás is, illetve nem támogat olyan menedzselési funkciókat, mint az *IIS* (például verziókezelést).

Ennek a hosztolási módszernek a tipikus esete, amikor konzol alkalmazás segítségével hosztoljuk a szolgáltatást. Ez a fejlesztési fázisban megkönnyíti a hibakeresést és megnöveli a kód mozgathatóságát. Az erre vonatkozó példát már bemutattam a *Végpontok* című fejezetben.

3.2 Windows Process Activation Service (WAS)

A *WAS* előnye az *IIS*-sel szemben, hogy nem függ a HTTP protokolltól. Miután a *service contract*-ot létrehoztuk és implementáltuk, nincs szükség másra a kódban. A szükséges beállításokat a konfigurációs fájlban kell elhelyezni. Létre kell hoznunk egy *.svc* fájlt és elhelyezni az *IIS* virtualis könyvtárában. A *Microsoft Visual Studio* segítségével létrehozhatjuk ezt a fájlt a *WCF Service* sablont választva.



7. ábra – WCF Service sablon

A *.svc* fájlban a következő sort helyezzük el, ahol megadjuk a szolgáltatás típusát:

```
<%@ ServiceHost Language="C#" Debug="true"
Service="Thesis.WCFServices.EmployeeService" %>
```

4 Üzenetváltási modellek

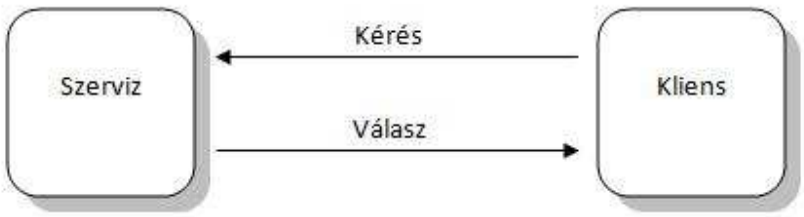
A WCF a következő üzenetváltási modelleket támogatja:

- *Request/Reply* (kérés/válasz)
- *One-way* (egyirányú)
- *Duplex*

A következő alfejezetek ezeknek a modelleknek a tulajdonságait, alkalmazásukat mutatják be.

4.1 Kérés/válasz alapú üzenetváltás

Ebben az üzenetváltási modellben a kliens kérést küld a szolgáltatásnak, majd erre választ kap. Ez az alapértelmezett minta. Ebben a modellben a service metódus hívásakor az input paramétereken és a visszatérési értéken kívül *out* és *ref* paraméter(ek) használatára is lehetőség van. Ha nincs visszatérési érték megadva, akkor is érkezik válasz a kliensnek egy üres üzenet formájában, amely jelzi, hogy a metódus visszatért (az egyetlen lehetőség a válasz üzenet elkerülésére az egyirányú kommunikációs minta használata).



8. ábra – Kérés/válasz alapú üzenetváltási modell

Ez a kommunikációs modell csökkentheti a kliens teljesítményét, ha a metódus hosszú ideig fut. Ezt a problémát elkerülhetjük, ha egyirányú kommunikációt folytatunk. Ennek azonban az a

hátránya, hogy nincs válasz üzenet (A későbbiekben látni fogjuk az aszinkron modell használatát, amely segítséget nyújt ezen problémák megoldásában.). Ha visszatérési típus helyett a *void* kulcsszót adjuk meg, akkor is van értelme, illetve előnye ennek a kommunikációs formának: a válasz üzenetből a hívás során keletkezett hibákról a kliens tudomást szerezhet.

Kérés/válasz alapú service metódus létrehozásakor beállítjuk az *OperationContractAttribute* attribútum *IsOneWay* property-jének értékét *false*-ra (ez az alapértelmezett érték).

```
// Az interfész definiálja a szolgáltatás által
// hívható metódusokat
 [ServiceContract]
public interface IEmployee
{
      // Kérés/válasz alapú üzenetváltás
      [OperationContract(IsOneWay=false)]
      EmployeeInfo GetEmployeeInfo(string id);

      // Kérés/válasz alapú üzenetváltás
      [OperationContract]
      void SetEmployeeInfo(EmployeeInfo info);
}
```

9. ábra – Kérés/válasz alapú üzenetküldés beállítása a service contract-ban.

4.2 Egyirányú üzenetküldés

Ezt a kommunikációs modellt akkor érdemes használni, ha a kliens nem akar vagy nem tud várni, amíg a service metódus lefut, vagy nincs szüksége válaszra (például az *Observer* tervezési minta esetén a feliratkozottak értesítése nem igényel választ). Ebben az esetben a kliens nem tudja kezelni a kommunikáció során felmerülő SOAP hibákat sem. Nincs visszatérési érték, nem használható *out* és *ref* paraméter, ugyanis nincs válasz üzenet, amely visszaadná az értéket.

10. ábra – Egyirányú üzenetküldési modell

Egyirányú service metódus létrehozásakor beállítjuk az *OperationContractAttribute* attribútum *IsOneWay* property-jének értékét *true*-ra (*false* az alapértelmezett érték).

```
[ServiceContract]
public interface IObserver
{
	// Egyirányú üzenetküldés
	[OperationContract(IsOneWay=true)]
	void Subscribe(string id);

	// Egyirányú üzenetküldés
	[OperationContract(IsOneWay=true)]
	void NotifyAll();
}
```

11. ábra – Egyirányú üzenetküldés beállítása a service contract-ban.

4.3 Duplex modell

A duplex kommunikációs modellben a szolgáltatás és a kliens egymástól függetlenül küldhet üzeneteket a másiknak egyirányú vagy kérés/válasz típusú üzenetküldéssel.

12. ábra – Duplex üzenetküldési modell

A duplex *service contract* mellé létre kell hozni egy másik interfészt is, amelyben megadjuk azon metódusok specifikációját, amelyeket a szolgáltatás hívhat a kliensen. Az implementáció természetesen a kliens oldalon kerül megadásra. Itt is használjuk az *OperationContractAttribute* attribútumot azon metódusokra, melyek publikusak lesznek a szolgáltatás felé.

A callback interfész típusát meg kell adnunk a *ServiceContractAttribute* attribútum *CallbackContract* property-jének.

```
[ServiceContract(CallbackContract=typeof(ICallback))]
public interface IDuplex
{
      [OperationContract(IsOneWay = true)]
      void DoTimeConsumingWork();
}

public interface ICallback
{
      [OperationContract(IsOneWay=true)]
      void Complete();
}
```

13. ábra – Service és callback contract

Ezután implementáljuk a *service contract*-ot definiáló interfészt. Ekkor az *OperationContext.Current.GetCallbackChannel<T>()* metódussal hozzáférhetünk a kliens publikus interfészéhez, azaz meghívhatjuk azokat a metódusokat, amelyeket a kliens publikált.

```
// Szolgáltatás implementációja
public class DuplexService : IDuplex
{
       public void DoTimeConsumingWork()
       {
              Console.WriteLine("A szolgáltatás elkezdte a műveletet: {0}",
                                  DateTime.Now);
              // Egy időigényes művelet szimulálása
              Thread.Sleep(5000);
              Console.WriteLine("A szolgáltatás elkezdte a műveletet: {0}",
                                  DateTime.Now);
             // Metódushívás a kliensen
       OperationContext.Current.GetCallbackChannel<ICallback>().Complete();
       }
}
```

14. ábra – A szolgáltatás implementációja

A kliens oldalon adjunk implementációt a callback interfészhez. Hozzunk létre egy *InstanceContext* objektumot, melynek konstruktorában adjuk át az implementált interfész egy példányát. Ezen a példányon hívja majd a szolgáltatás a publikált metódusokat. Az *InstanceContext* példányt pedig adjuk át a *service proxy* konstruktorának példányosításkor.

```
// Callback interfész implementációja
public class CallbackImplementation : IDuplexCallback
{
       public void Complete()
       {
              Console.WriteLine("A kliens értesítést kapott a " +
                                  "szolgáltatástól: {0}", DateTime.Now);
       }
}
...
InstanceContext callbackInstance =
                     new InstanceContext(new CallbackImplementation());
DuplexClient client = new DuplexClient(callbackInstance);

try
{
       client.DoTimeConsumingWork();
       client.Close();
}
...
```

15. ábra – Callback interfész implementálása és a service proxy példányosítása.

Megjegyzés:

A kliens oldal felépítésében elengedhetetlen eszköz a *ServiceModel Metadata Utility Tool* (svcutil.exe), melynek segítségével könnyedén generálhatjuk a szolgáltatás proxy infrastruktúráját, illetve a szükséges konfigurációs fájlt.

5 Szerializáció

A .NET keretrendszer kétféle szerializálást használ: bináris és XML szerializálást. Bináris esetben a küldő és fogadó félnek azonos platformmal és futtató rendszerrel kell rendelkeznie, de a szolgáltatás orientált alkalmazásoknál gyakori, hogy a kommunikáció végpontjain különböző nyelveken írt programok futnak, akár különböző operációs rendszert felhasználva. Ekkor is elvárható az a működés, hogy a két fél fel tudja dolgozni a megkapott adatokat. Ehhez nyújt segítséget az XML szerializálási technika, melynek során memóriabeli objektumból XML adat keletkezik, majd a deszerializálás során visszakapjuk az objektumot.

A WCF szolgáltatások esetén a kliens és a szerver a kommunikáció során adatot cserélnek. Ezt a műveletet paraméterek átadásával, illetve visszatérési érték formájában valósítják meg. Az üzenetben szereplő adatok reprezentációja XML, tehát szerializációra is szükség van, mielőtt az adat a csatornára kerül, és deszerializációra, mielőtt a másik oldal feldolgozná azt. A primitív típusokat (*int*, *float*, *byte*, stb.) módosítás nélkül használhatjuk, viszont a felhasználó által definiált típusokat szerializálhatóvá kell tennünk. Ehhez nyújt segítséget a következő két WCF által támogatott XML szerializáló mechanizmus.

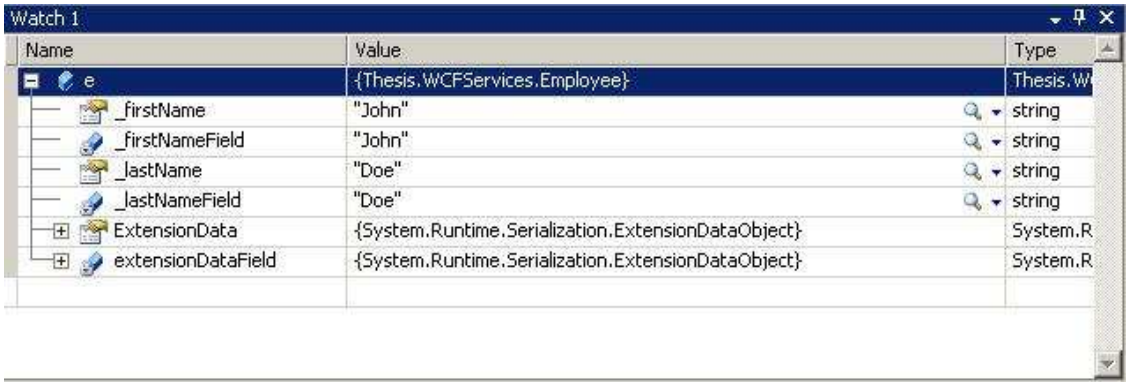
5.1 DataContractSerializer

Ez az alapértelmezett szerializálási technika. Ezen mechanizmus használatához az adott osztályra alkalmazni kell a *System.Runtime.Serialization.DataContractAttribute* attribútumot, illetve a szerializálandó adattagokra a *System.Runtime.Serialization.DataMemberAttribute* attribútumot. Csak azok az adattagok kerülnek szerializálásra, amelyek meg vannak jelölve ezzel az attribútummal. (A *DataContractAttribute* attribútum osztályokon kívül struktúrákra és felsorolásos típusokra is alkalmazható.) Példaként tekintsük az alábbi osztályt:

```
[DataContract]
public class Employee
{
	[DataMember]
	public string _firstName;
	[DataMember]
	public string _lastName;
	public string _address;
	public string _email;
	private string _division;
	private int _salary;
}
```

16. ábra – Szerializálás DataContractSerializer használatával

A *_firstname* és *_lastname public* láthatóságú változókra alkalmaztam a *DataMemberAttribute* attribútumot, melynek hatására ezek a változók részt vesznek a szerializálási folyamatban. A kliens oldalon visszakapott *Employee* példány a következő változókat mutatja:



17. ábra – A deszerializálás eredménye

A kliens és szerver kommunikációja során nem kötelező, hogy az üzenetváltásnál használt típusok a csatorna mindkét végén létezzenek. Elégséges a *data contract* egyenlősége. Két *data contract* akkor egyenlő, ha a következő feltételek mindegyike teljesül:

- Azonos a nevük.

- Az összes tartalmazott *DataMember* attribútummal ellátott adattag neve és sorrendje megegyezik.

Megjegyzés: A *DataContract*, illetve *DataMember* nevei alapértelmezésként a megjelölt osztályok, adattagok nevei, de az attribútum *Name* property-je segítségével felül lehet bírálni.

A sorrendnél fontos megjegyezni, hogy az alapértelmezett rendezés ábécé szerinti, de a *DataMember* attribútum *Order* property-je segítségével ez felülbírálható. Az alábbi ábrán két ekvivalens *data contract* szerepel:

```
[DataContract(Name="Circle")]
public class Circle1
{
	[DataMember]
	public int originX;
	[DataMember]
	public int originY;
	[DataMember]
	public int ray;
}

[DataContract(Name = "Circle")]
public class Circle2
{
	[DataMember(Order = 3)]
	public int ray;
	[DataMember(Order = 1)]
	public int originX;
	[DataMember(Order = 2)]
	public int originY;
}
```

18. ábra – Ekvivalens data contract-ok

Leszármazott típusok esetén a sorrendnél a szülőosztály adattagjainak sorrendjét, majd a fennmaradó adattagok sorrendjét vesszük alapul.

A kommunikáció során gyakran használunk olyan típusú paramétereket, visszatérési értékeket, melyek nem képezik a *data contract* részét, de kötődnek hozzá. Ilyen előfordulhat abban az esetben, ha a *data contract*-ban a konkrét típus helyén valamilyen interfész típusú, bázisosztály típusú (akár *object*, akár más) eszköz szerepel. Ekkor a szerializálás sikere ellenére a deszerializálás hibás lesz, mert a másik fél nem tudja azonosítani az adott típust, mivel nem szerepel a *data contract*-ban. E probléma megszüntetésére használhatjuk a *System.Runtime.Serialization.KnownTypeAttribute* attribútumot. Használatára nézzünk egy példát:

```
[DataContract]
public class Tire { }

[DataContract]
public class GoodYear : Tire { }

[DataContract]
public class Michelin : Tire { }

[DataContract]
[KnownType(typeof(GoodYear))]
[KnownType(typeof(Michelin))]
public class RaceCar
{
      [DataMember]
      public Tire _tire;
      …
}
```

19. ábra – KnownType attribútum használata

(A fent említett kapcsolódó típusokat a konfigurációs fájlokban is beállíthatjuk.)

Lehetőség van arra, hogy egy típust előre kompatibilissé tegyünk. Ehhez implementálnunk kell a *System.Runtime.Serialization.IExtensibleDataObject* interfészt, mely tartalmazza az *ExtensionData* property definícióját. Ha a WCF olyan adatot talál, amelyik nem része a *data contract*-nak, akkor eltárolja azt ebben a property-ben.

Előfordulhat, hogy a kommunikáció egyik oldalán egy *data contract* olyan újabb verziója szerepel, amelyben a régihez képest újabb adattagok is szerepelnek. Felmerülhet az igény arra, hogy a (de)szerializálás valamely szakaszában kezeljük a verzionális eltéréseket, például adjunk alapértelmezett értéket az adott adattagnak. Ehhez nyújt segítséget a *System.Runtime.Serialization* névtér alábbi 4 attribútuma:

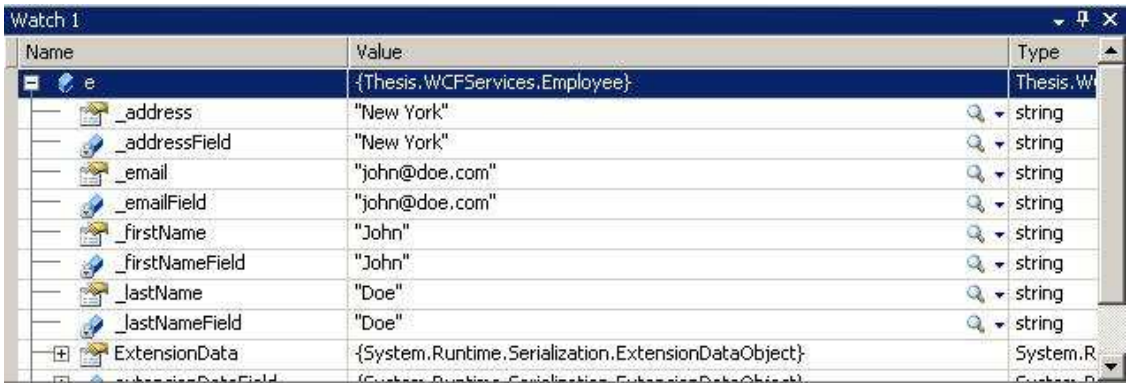
- *OnSerializingAttribute*
  A megjelölt metódus a szerializálás előtt hívódik meg.
- *OnSerializedAttribute*
  A megjelölt metódus a szerializálás után hívódik meg.
- *OnDeserializingAttribute*
  A megjelölt metódus a deszerializálás előtt hívódik meg.
- *OnDeserializedAttribute*
  A megjelölt metódus a deszerializálás után hívódik meg.

5.2 XmlSerializer

Az alapértelmezett szerializálás a *System.Runtime.Serialization.DataContractSerializer* osztály segítségével történik, ezért ha az *System.Xml.Serialization.XmlSerializer*-t szeretnénk használni, akkor a szolgáltatást meg kell jelölni az *XmlSerializerFormatAttribute* attribútummal. Az *XmlSerializer* a típusok szűkebb körét támogatja, de nagyobb kontrollt biztosít az XML szerkezetének felépítéséhez.

A szerializáló mechanizmusok váltogatása nagy körültekintést igényel, ugyanis előfordulhat, hogy nem ugyanaz lesz az eredmény. Kimaradhatnak vagy éppen megjelenhetnek olyan adatok, melyeknek nem kellene. Míg a *DataContractSerializer* esetén csak a *DataMemberAttribute* attribútummal megjelölt adattagok kerülnek szerializálásra, addig az *XmlSerializer* használatával minden publikus adattag.

Példaként tekintsük ismét a fentebb említett *Employee* osztályt. Szerializáljuk most az *XmlSerializer* osztály segítségével.



20. ábra – A deszerializálás eredménye

Láthatjuk, hogy ebben az esetben az *Employee* osztály minden publikus adattagja megjelenik a kliens oldalon.

5.3 Message Contract

Előfordulhat olyan szituáció, amikor nem csak az üzenetben küldött adatok szerkezetének meghatározása lényeges, hanem az üzenet szerkezete is. A típusok SOAP üzenetté történő leképezéséhez nyújt segítséget a *System.ServiceModel.MessageContractAttribute* attribútum, valamint a *System.Messaging.Message* osztály. Ekkor megadhatjuk, hogy mely adattagok képezzék az üzenet fejrészét, törzsét. Amennyiben használni szeretnénk ezt a funkciót, akkor olyan service metódusokat kell írnunk, melyek legfeljebb 1 paraméterrel rendelkeznek, illetve van visszatérési értékük. Mindkét esetben a típust meg kell jelölni a *MessageContractAttribute* attribútummal vagy *Message* típusúnak kell lennie.

Egy típust a *MessageContractAttribute* attribútummal megjelölve *data contract*-tá tehetünk. Ha a típus adattagjait a *System.ServiceModel.MessageHeaderAttribute* attribútummal jelöljük meg,

akkor azok a SOAP fejlécbe kerülnek, *System.ServiceModel.MessageBodyMemberAttribute* esetén pedig a törzsbe.

```
…
[OperationContract]
EmployeeInfo GetEmployeeInfo(Employee e);
…

[MessageContract]
public class Employee
{
      [MessageHeader] public string _firstName;
      [MessageHeader] public string _lastName;
      [MessageBodyMember] public string _id;
}

[MessageContract]
public class EmployeeInfo
{
      [MessageHeader(Name="FirstName")] public string _firstName;
      [MessageHeader(Name="LastName")] public string _lastName;
      [MessageBodyMember(Name="Identifier")] private string _id;
      [MessageBodyMember] private DateTime _dateOfBirth;
      [MessageBodyMember] private string _division;
      [MessageBodyMember] private string _salary;
}
```

21. ábra – Message contract használata

Az üzenetben szereplő adatokat védeni tudjuk, ha a *MessageHeaderAttribute* és *MessageBodyMemberAttribute* attribútumok *ProtectionLevel* property-jét beállítjuk.
Lehetséges értékei:

- *None* (nincs titkosítás és digitális aláírás)
- *Sign* (digitális aláírás)
- *EncryptAndSign* (titkosítás és digitális aláírás)

A fejrészre külön-külön is beállíthatjuk az értékeket, a törzsre viszont a legerősebb érték kerül alkalmazásra.

A WCF lehetőséget nyújt arra, hogy az üzenetet szűrjük, mellyel különböző értékekre különböző futásidejű viselkedést írhatunk elő. A szűrés az üzenet beérkezése után hajtódik végre (Például, amikor egy sor alapú rendszerbe üzenet érkezik, akkor dönthetünk arról, hogy egy magasabb prioritású üzenet a sorban előrébb jusson.). Szűrhetjük az *Action header*-t, a végpont címét vagy annak egy prefixét, illetve *XPath* kifejezés segítségével eldönthetjük, hogy az adott XML dokumentum tartalmaz-e egy adott elemet, attribútumot, stb. A WCF üzenet szűrői a következők:

- *MatchAllMessageFilter*
  Az összes üzenetre illeszkedik.
- *MatchNoneMessageFilter*
  Egyik üzenetre sem illeszkedik.
- *XPathMessageFilter*
  Egy XPath kifejezés segítségével dönt az illeszkedésről.
- *EndpointAddressMessageFilter*
  A végpont címére szűr.
- *PrefixEndpointAddressMessageFilter*
  A végpont címének prefixére szűr.

6 Aszinkron programozási modell (APM)

A .NET keretrendszer lehetővé teszi, hogy utasításokat futtassunk nemlineáris módon, ezáltal az alkalmazásunk teljesítménye megnő, a válaszidők lecsökkennek, bizonyos szűk keresztmetszetek eltűnnek és a rendszer erőforrásai magasabb hatásfokkal működnek.

Az aszinkron modell használatával meghatározott kódrészek külön szálon futnak, lehetővé téve, hogy egy hosszabb futás alatt (pl.: nyomtatás, IO műveletek) az alkalmazás más teendőket is végezzen. A keretrendszer jónéhány osztálya rendelkezik az aszinkron modellt megvalósító *BeginX* és *EndX* metódusokkal, például a *FileStream* osztályban megtalálhatjuk a *BeginRead* és *EndRead* metódusokat.
A *BeginX* metódusok sajátossága, hogy *System.IAsyncResult* objektummal térnek vissza, illetve két formális prarméterrel rendelkezniük kell: egy *System.AsyncCallback* és egy *object* típusú paraméterrel.

```
IAsyncResult BeginRead(byte[] array, int offset, int numBytes,
AsyncCallback userCallback, object stateObject);
```

Az *EndX* metódusoknak rendelkezniük kell egy *IAsyncResult* típusú formális paraméterrel.

```
int EndRead(IAsyncResult asyncResult);
```

Programunkat képessé kell tenni arra, hogy egy aszinkron futás végéről tudomást szerezzen, és eredményét fel tudja dolgozni. Erre 3 modellt ismertetnék: a *Wait-Until-Done* modellt, a *Polling* modellt, illetve a *Callback* modellt.

6.1 Wait-Until-Done modell

Ez tükörfordításban annyit jelent, hogy ”várj, amíg nincs kész”. Ez a modell is lehetővé teszi, hogy más utasítások fussanak a *BeginX* és *EndX* metódushívások között, de ha ezen utasítások hamarabb lefutottak, mint ahogy az aszinkron hívás visszatér, akkor a fő szál blokkolt állapotba kerül, megvárja a hívás eredményét.

```
byte[] buffer = new byte[1000];
FileStream stream = new FileStream("c:/data.txt", FileMode.Open);

// Aszinkron hívás
IAsyncResult result = stream.BeginRead(buffer, 0, buffer.Length,
                                       null, null);
// A szál nem blokkolt, mert az
// aszinkron metódus másikat használ
DoUsefulWork();

// Az EndRead blokkolja a szálat, amíg az
// aszinkron hívás be nem fejeződik
int numberOfBytes = stream.EndRead(result);
stream.Close();
```

22. ábra – Wait-Until-Done modell

Ez a modell nem nyújt megfelelő segítséget, szűk keresztmetszeteket, hosszú várakozást eredményezhet.

6.2 Polling modell

Ez a modell megvizsgálja az *IAsyncResult* objektum (az *IAsyncResult* interfészt implementáló osztály példánya) *IsComplete* property-je segítségével, hogy az adott aszinkron hívás visszatért-e. Ha nem, akkor addig értékes műveleteket lehet végezni.

```
byte[] buffer = new byte[1000];
FileStream stream = new FileStream("c:/data.txt", FileMode.Open);

// Aszinkron hívás
IAsyncResult result = stream.BeginRead(buffer, 0, buffer.Length,
                                       null, null);
// Amíg az aszinkron művelet nem fejeződött be,
// addig más műveleteket végezhetünk.
while(!result.IsCompleted)
{
      // Ide kerülnek a végzendő műveletek
}

// Itt már befejeződött az aszinkron művelet,
// a szál nem lesz blokkolt.
int numberOfBytes = stream.EndRead(result);
stream.Close();
```

23. ábra – Polling modell

Ez a modell feltételezi, hogy az alkalmazásnak az aszinkron művelet alatt van más teendője is. Ez a legtöbb esetben nem teljesül, ezért az igazi megoldást a *Callback* modell nyújtja.

6.3 Callback modell

Ez a modell nyújtja a legkényelmesebb szolgáltatást, ugyanis megadhatunk egy metódust, ami meghívódik, ha az aszinkron művelet befejeződött. A művelet által visszaadott objektumot feldolgozó logikát pedig elhelyezhetjük ebben a metódusban.

A *BeginX* metódus *AsyncCallback* típusú paraméterének (ami egy *delegate*) megadunk egy *AsyncCallback* példányt, amire felirattuk a meghívandó metódusunkat. A metódusnak egy *IAsyncResult* paraméterrel kell rendelkeznie, és nem lehet visszatérési értéke.

```
static void Main(string[] args)
{
      byte[] buffer = new byte[1000];
      FileStream stream = new FileStream("c:/data.txt",
                                         FileMode.Open);
      // Aszinkron hívás
      IAsyncResult result = stream.BeginRead(buffer, 0,
                            buffer.Length,
                            new AsyncCallback(WorkCompleted),
                            null);

      DoUsefulWork();

      stream.Close();
}

public static void WorkCompleted(IAsyncResult result)
{
      // eredmény feldolgozása
}
```

24. ábra – Callback modell

6.4 Kivételkezelés aszinkron esetben

Az aszinkron hívás esetében a kivétel nem a bekövetkezés időpontjában és helyén dobódik, hanem az *EndX* metódus hívásakor, tehát a kivételeket ekkor tudjuk elkapni és kezelni.

Az *EndX* metódust feltétlenül *try* blokkban kell elhelyezni, ugyanis kivételt dobhat. Tekintsük az előző példában szemléltetett *Callback* modellt. A kivételkezelést az aszinkron művelet befejezésekor meghívandó metódusban kell elhelyezni, ha ott szeretnénk az eredményt feldolgozni.

```
public static void WorkCompleted(IAsyncResult result)
{
      int numberOfBytes = 0;
      try
      {
            FileStream stream = (FileStream)result.AsyncState;
            numberOfBytes = stream.EndRead(result);
            stream.Close();
      }
      catch (Exception ex)
      {
            // kivétel kezelése
      }
      ...
}
```

25. ábra – Kivételkezelés

6.5 Szinkron és aszinkron WCF metódusok

A WCF service metódusoknál mindenképpen használjunk aszinkron modellt, ha az alkalmazásunk egyszálas vagy a metódus olyan műveleteket végez, melyek blokkolóan hatnak az adott szálra. Ilyenek például az IO műveletek.

A WCF kliens kétféle aszinkron hívási modellt támogat. Az egyik az előző fejezetekben ismertetett *System.IAsyncResult* objektumot használja, míg a másik esemény alapú.

Ahhoz, hogy egy service metódust aszinkronná tegyünk, a *BeginX* metódusra az *OperationContractAttribute* attribútumot kell alkalmaznunk és az *AsyncPattern* property-jét *true*-ra kell állítani:

```
...
[OperationContract(AsyncPattern=true)]
IAsyncResult BeginDoWork(string data,
                         ref string inout,
                         AsyncCallback callback,
                         object state);

int EndDoWork(ref string inout, out string outonly, IAsyncResult
result);
...
```

26. ábra – Aszinkron service metódus implementációja

Egy WCF alkalmazásban egy service metódus lehet szinkron és aszinkron is anélkül, hogy ez megkövetelné a klienstől, hogy hogyan hívja. Tehát egy szinkron metódust meg lehet hívni aszinkron módon és egy aszinkron metódust szinkron módon.

A kliens kód generálásánál az svcutil.exe programnak megadhatjuk, hogy generáljon aszinkron és szinkron infrastruktúrát a szolgáltatáshoz. Ekkor a szinkron metódusoknak aszinkron változata is lesz és fordítva, illetve az esemény alapú aszinkron hívást is megvalósíthatjuk. Példaként tekintsük a következő szolgáltatás implementációt:

```
class MathService : IMathContract
{
      public double Add(double number1, double number2)
      {
            // Hosszú futás szimulálása
            Thread.Sleep(1000);
            return number1 + number2;
      }
      …
}
```

27. ábra – Service metódus implementációja

Az implementációban egy szinkron metódust láthatunk, amelyben a hívó szálat blokkoljuk 1 másodpercig, ezzel szimulálva a hosszú futási időt. Szinkron hívás esetén programunk blokkolt állapotba kerül, ami hosszú válaszidőt eredményez, ezért hívjuk meg aszinkron módon.

Ehhez hívjuk segítségül az svcutil.exe programot és futtassuk a következő módon:

```
svcutil <wsdl_cím> /async /tcv:Version35
```

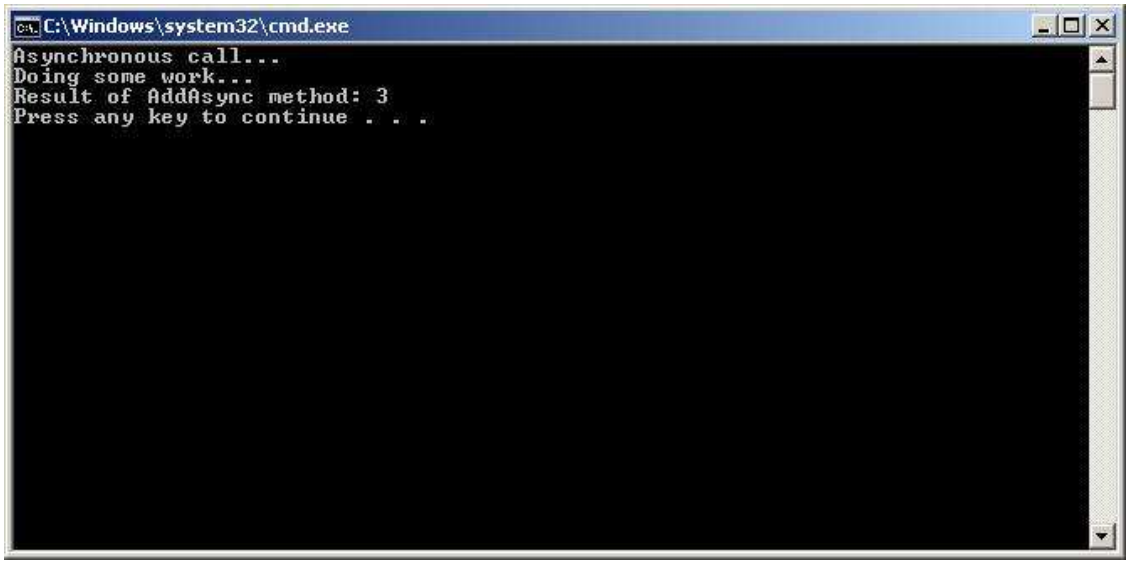
A /async kapcsoló hatására a program generálja nekünk a metódusok szinkron és aszinkron változatát, valamint a megfelelő *event*-eket. Nézzük meg a kliens oldalt:

```
class Client
{
      static void Main(string[] args)
      {
            MathClient client = new MathClient();
            client.AddCompleted +=
new EventHandler<AddCompletedEventArgs>(Client.Client_AddCompleted);
            Console.WriteLine("Asynchronous call...");
            client.AddAsync(1, 2);
            Console.WriteLine("Doing some work...");
            Thread.Sleep(10000);
      }

      static void Client_AddCompleted(object sender,
                                      AddCompletedEventArgs e)
      {
            Console.WriteLine("Result of AddAsync method: {0}",
                              e.Result);
      }
}
```

28. ábra – Aszinkron hívás a kliens oldalon

A kliens oldalon meghívtuk az *Add* metódus generált aszinkron változatát, majd kiírattunk a konzolra, ezzel is szimulálva egyéb utasítások végrehajtását, a szintén generált *AddCompleted* eseményhez megadott eseménykezelő metódus pedig megjeleníti az eredményt, amint az aszinkron hívás visszatért.

A konzolon a várt eredmény jelenik meg:



29. ábra – Aszinkron hívás eredménye a konzolon

7 Munkamenetek

A Windows Communication Foundation munkamenete üzenetek egy csoportját egy adott párbeszéd részévé teszi, azaz lehetőség van egy munkameneten belül állapotmenedzsmentre, adatok megosztására, illetve a munkamenetet inicializáló és lezáró műveletek megadására.

A WCF munkamenetek fő jellemzői:

- A hívó alkalmazás inicializálja és zárja le explicit módon.
- Nincs a WCF munkameneteknek fenntartott általános tárolóhely.
- A munkamenet alatt kézbesített üzenetek az érkezés sorrendjében kerülnek feldolgozásra.

A *ServiceContractAttribute* attribútum *SessionMode* property-je segítségével beállíthatjuk, hogy az adott service contract megköveteli, megengedi vagy tiltja a munkamenet alapú *binding*-okat. A lehetséges értékek a *System.ServiceModel.SessionMode* enumeráció értékei lehetnek:

- *Allowed*: A *service contract* támogatja a munkameneteket, ha a beérkező *binding* is támogatja.
- *Required*: A *service contract* megköveteli a munkameneteket támogató *binding*-ot. Ha nincs, akkor kivétel váltódik ki.
- *NotAllowed* : A *service contract* nem támogat olyan *binding*-ot, amely munkamenetet tud inicializálni.

Ez a beállítás csak a *binding* munkamenettel kapcsolatos tulajdonságára ad megszorítást, egyéb tulajdonságaira (például transport, security, stb.) nem. Ha a *SessionMode* property értéke és a *binding* között nincs összefüggés, akkor kivétel váltódik ki.

Ha a *service contract*-ot úgy állítottuk be, hogy megköveteljen (*Required*) vagy támogasson (*Allowed*) munkameneteket, akkor a service metódusokhoz megadhatjuk, hogy hívásukkal indítsanak-e, illetve lezárjanak-e munkamenetet.

Az inicializáló metódusokat meg kell hívni egy új munkamenet létrehozásához, a nem inicializáló metódusok hívását pedig meg kell, hogy előzze legalább egy inicializáló hívás.

Munkamenet indítását okozzák a következő hívások a kliens oldalon:

- A *System.ServiceModel.ChannelFactory.CreateChannel* metódus által visszaadott objektumon hívott *Open* metódus.
- *System.ServiceModel.ClientBase.Open* metódus hívása az Svcutil.exe program által generált kliens osztály példányán.
- Inicializáló service metódus (alapértelmezésként minden service metódus inicializáló) hívása a kliens objektumon.

Munkamenet lezárását okozzák a következő hívások a kliens oldalon:

- A *System.ServiceModel.ChannelFactory.CreateChannel* metódus által visszaadott objektumon hívott *Close* metódus.
- *System.ServiceModel.ClientBase.Close* metódus hívása az Svcutil.exe program által generált kliens osztály példányán.
- Terminláló service metódus hívása a kliens példányon.

A következő példában (forrás: http://msdn.microsoft.com/) egy olyan *service contract*-ot láthatunk, amely megköveteli a munkamenetek használatát. Láthatjuk, hogy az egyetlen inicializáló metódus a *Clear*, tehát az implementáció során az eredmény tárolására használt változót itt inicializálhatjuk (például nulla értéket adunk neki). Az *Equals* metódus terminláló, azaz lezárja a munkamenetet. A további metódusok nem indítanak és nem zárnak le

munkamenetet. Ez a szerkezet biztosítja, hogy egy munkamenet létrejöttekor és lezárásakor bizonyos feladatokat elvégezzen a szolgáltatás.

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface ICalculator
{
        [OperationContract(IsOneWay=true,
                           IsInitiating=true,
                           IsTerminating=false)]
        void Clear();

        [OperationContract(IsOneWay = true,
                           IsInitiating = false,
                           IsTerminating = false)]
        void AddTo(double n);

        [OperationContract(IsOneWay = true,
                           IsInitiating = false,
                           IsTerminating = false)]
        void SubtractFrom(double n);

        [OperationContract(IsOneWay = true,
                           IsInitiating = false,
                           IsTerminating = false)]
        void MultiplyBy(double n);

        [OperationContract(IsOneWay = true,
                           IsInitiating = false,
                           IsTerminating = false)]
        void DivideBy(double n);

        [OperationContract(IsInitiating = false,
                           IsTerminating = true)]
        double Equals();
}
```

30. ábra – Munkamenetes tulajdonságok beállítása

8 Szolgáltatás példányok

A *System.ServiceModel.ServiceBehaviorAttribute* attribútum *InstanceContextMode* property-je segítségével vezérelhetjük a szolgáltatás példányainak létrejöttét. A lehetséges értékeket a *System.ServiceModel.InstanceContextMode* enumeráció tartalmazza:

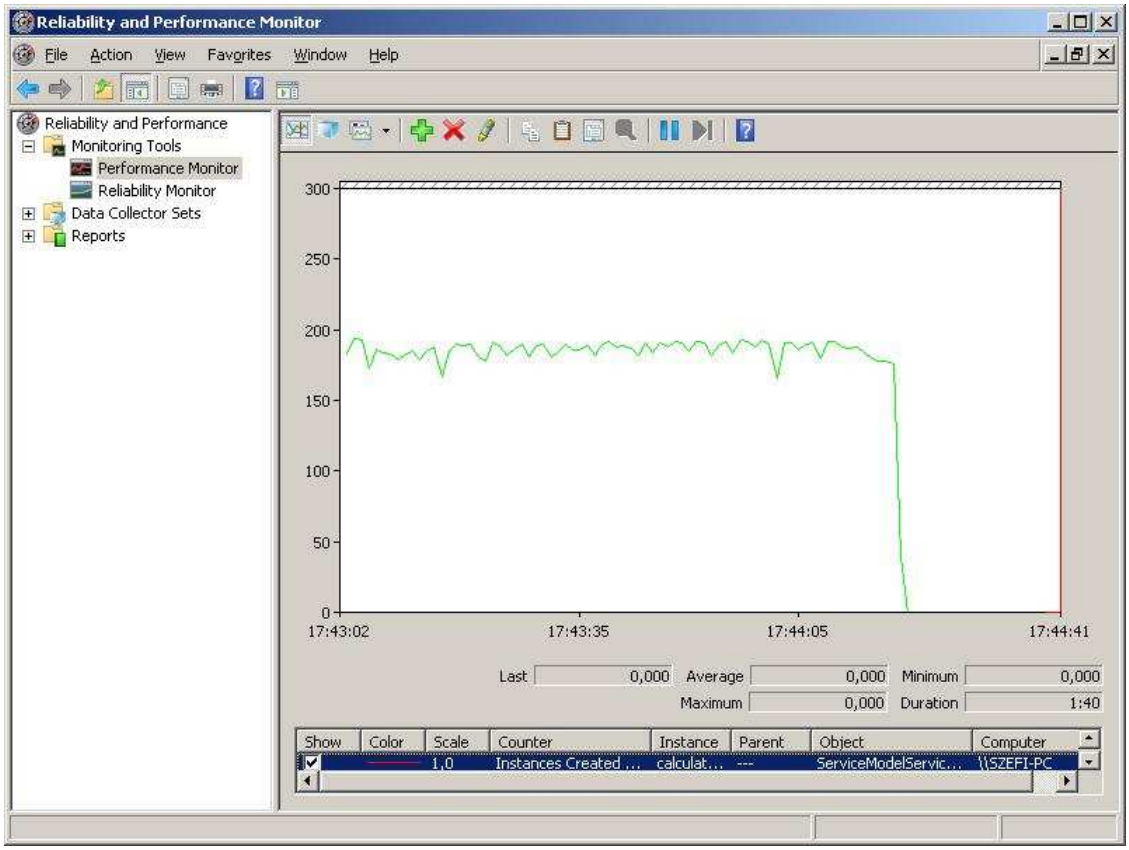
- *PerCall* : egy új szolgáltatás objektum jön létre minden egyes hívás előtt.
- *PerSession* : egy új szolgáltatás objektum jön létre minden egyes munkamenetre.
- *Single*: egy szolgáltatás objektum jön létre az összes hívás kezelésére.

A következő példa forrása a http://msdn.microsoft.com/:

```
[ServiceContract(SessionMode=SessionMode.Allowed)]
public interface ICalculator
{
      [OperationContract]
      double Add(double n1, double n2);
      [OperationContract]
      double Subtract(double n1, double n2);
      [OperationContract]
      double Multiply(double n1, double n2);
      [OperationContract]
      double Divide(double n1, double n2);
}

[ServiceBehaviorAttribute(InstanceContextMode =
InstanceContextMode.PerSession)]
public class CalculatorService : ICalculator
{
      ...
}
```

31. ábra – InstanceContextMode property beállítása

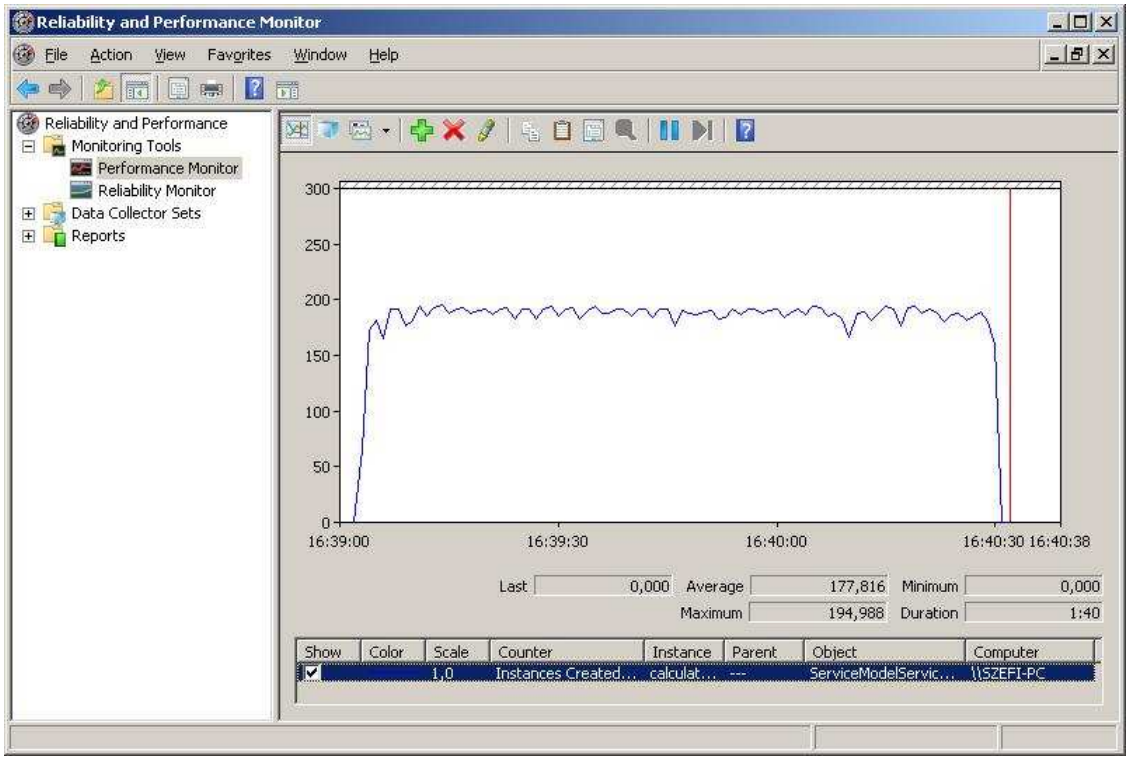
Ez a *service contract* támogatja a munkameneteket. A kliens oldal szimulálja a szolgáltatás metódusainak a folyamatos hívását. A végponthoz érkező hívások és a keletkezett példányok másodpercenkénti számát a következő ábra mutatja:



32. ábra – A végponthoz érkező hívások és a keletkezett példányok másodpercenkénti száma

Amint az ábrán láthatjuk, másodpercenként körülbelül 180-190 service metódust hív a kliens, de mivel ez ugyanahhoz a munkamenethez tartozik, és az *InstanceContextMode* property értéke *PerSession*, nem keletkezik új példány. (Az ábrán a zöld görbe jelöli a meghívott service metódusok másodpercenkénti számát, a másodpercenként létrejött példányok száma pedig nulla, ezért annak a görbéje nem látható.)

Ha az *InstanceContextMode* property-nek *PerCall* értéket adunk, akkor minden service metódus hívása esetén egy új szolgáltatás példány jön létre. A következő ábra az előző példában szereplő teljesítményszámlálókat mutatja (A kliens változatlan, csak a *service contract*-on, illetve a generált kódban történt módosítás).



33. ábra – A végponthoz érkező hívások és a keletkezett példányok másodpercenkénti száma

A hívott service metódusok számában természetesen nincs jelentős eltérés, de a képen kijelölt teljesítményszámláló (*Instances Created Per Second*) a 177,816 átlagot mutatja, ami jól szemlélteti, hogy minden hívás előtt új példány keletkezik.

9 Tranzakciók

A tranzakció utasítások, műveletek oszthatatlan végrehajtási egysége, amely rendelkezik az ún. *ACID* tulajdonságokkal:

- ***A**tomicity* (atomosság): A tranzakcióban szereplő utasítások vagy teljes egészében végrehajtódnak és tartóssá (lásd: *durability*) válnak, vagy visszagörgetésre kerülnek és hatásuk elvész.
- ***C**onsistency* (konzisztencia): Ez a tulajdonság garantálja, hogy egy tranzakció alatt bekövetkezett változások az egyik konzisztens állapotból egy másik konzisztens állapotba visznek.
- ***I**solation* (elkülönülés): A tranzakció nem vehet figyelembe egy be nem fejezett tranzakció által végrehajtott változtatást. Úgy kell viselkednie, mintha az adott pillanatban ő lenne az egyetlen tranzakció.
- ***D**urability* (tartósság): Amikor egy tranzakció befejeződött, akkor a kifejtett hatása rögzül, nem állítható vissza a megelőző állapot.

A WCF esetén a következő 3 attribútum segítségével meghatározhatjuk egy szolgáltatás tranzakciós viselkedését:

TransactionFlowAttribute attribútum

Meghatározza, hogy az adott metódus hogyan viszonyuljon a klienstől érkező tranzakciókhoz. Külső tényezőre ad megszorítást. Lehetséges értékek:

- *Mandatory* (kötelező)
- *Allowed* (engedélyezett)
- *NotAllowed* (nem engedélyezett)

Megjegyzés: Lehetőség van ezen előírás magasabb szinten (végpont) történő megadására is (akár konfigurációs fájlban, akár kódban)

ServiceBehaviorAttribute attribútum

Ez az attribútum a szolgáltatás belső működését határozza meg. A tranzakciós viselkedésre vonatkozó property-jei a következők:

- *TransactionAutoCompleteOnSessionClose*
  Ha a property értéke *true* (ebben az esetben természetesen a csatornának munkamenetesnek kell lennie) és a munkamenet szabályosan fejeződik be, akkor a folyamatban lévő tranzakciók befejeződnek. Ha a property értéke *false* vagy a munkamenet nem szabályosan fejeződik be (kezeletlen kivétel, hálózati probléma), akkor a nyitott tranzakciók visszagörgetésre kerülnek.

- *ReleaseServiceInstanceOnTransactionComplete*
  Ezzel a property-vel megadhatjuk, hogy egy tranzakció befejezésével a szolgáltatás adott példánya felszabaduljon-e és egy új példány jöjjön létre vagy sem. Amennyiben új példány keletkezik minden tranzakció befejeződésekor, akkor a tranzakciók által létrejött állapotok nem maradnak meg.

- *TransactionTimeout*
  Azt az intervallumot határozza meg, amely alatt egy tranzakciónak be kell fejeződnie, máskülönben meg lesz szakítva.

- *TransactionIsolationLevel*
  A szolgáltatás tranzakcióknál használt izolációs szintjének megadására szolgál.

OperationBehaviorAttribute attribútum

Ez az attribútum a service metódusok belső működését határozza meg. Nincs hatással a WSDL-re. A tranzakciós viselkedésre vonatkozó property-jei a következők:

- *TransactionScopeRequired*

  A property értéke megadja, hogy az adott metódust egy aktív tranzakció hatáskörében kell-e hívni. Amennyiben nincs külső tranzakciós hatáskör, akkor egy új tranzakció jön létre, közvetlenül a hívás előtt.

- *TransactionAutoComplete*

  Megadja, hogy egy tranzakció automatikusan befejeződik-e, ha a metódus nem kezelt kivétel dobása nélkül visszatér.

Megjegyzés: Bizonyos tranzakciós tulajdonságok a konfigurációs fájlokban is beállíthatók.

Az alábbi példában egy metódus néhány tranzakciós tulajdonsága van megadva:

```
[ServiceBehavior(
        TransactionTimeout = "00:00:30",
        ReleaseServiceInstanceOnTransactionComplete = false,
        TransactionAutoCompleteOnSessionClose = false)]
class TransactionalService : ITransactionalContract
{
      [OperationBehavior(TransactionScopeRequired = true,
                         TransactionAutoComplete = true)]
      public void TransactionalOperation()
      {
            // Tranzakciós műveletek szimulálása
            Thread.Sleep(10);
      }
}
```

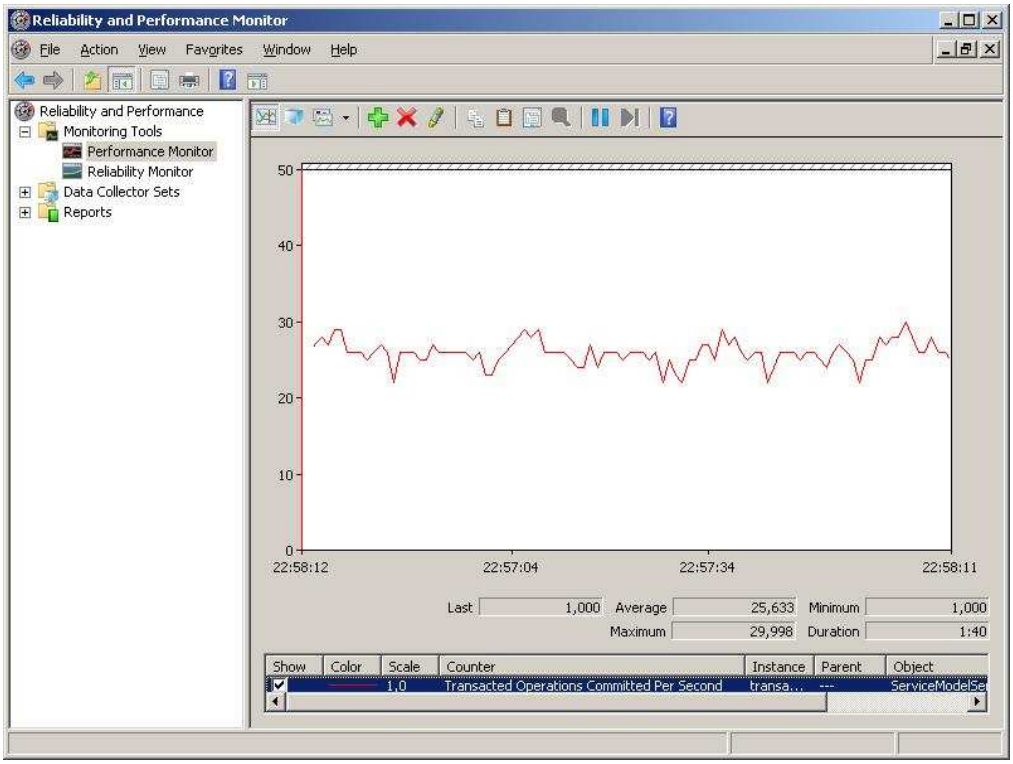
34. ábra – Szolgáltatás tranzakciós tulajdonságainak beállítása

A kliens oldalon a metódust egy tranzakción belül hívjuk:

```
using (TransactionScope tx = new TransactionScope())
{
      Console.WriteLine("Transaction started.");
      client.TransactionalOperation();
      // Complete transaction
      tx.Complete();
      Console.WriteLine("Transaction completed.");
}
```

35. ábra – Service metódus hívása tranzakció részeként

A teljesítményszámlálók alkalmazásával a tranzakciós szolgáltatások működéséről diagnosztikai információkat kaphatunk. A következő ábrán egy szimulált környezet másodpercenkénti befejezett tranzakcióit láthatjuk.



36. ábra – Másodpercenként befejezett tranzakciók számát mutató teljesítményszámláló

10 Kivételkezelés

Kivételek természetesen a WCF szolgáltatások esetén is kiváltódhatnak, tehát fel kell készíteni az alkalmazást a kivételek kezelésére.
A SOAP alapú alkalmazásoknál a hibák információi SOAP hibaüzenet formájában kerülnek továbbításra. A SOAP hibák olyan üzenettípusok, melyek részei a service metódusok metaadatainak, ezáltal a kliens kezelni tudja azokat, robosztussá téve az alkalmazást. Természetesen ezek az üzenetek is XML formájában kerülnek továbbításra, ezért minden SOAP platformot használó kliens képes az információ kinyerésére.

A szerver kivételei SOAP hibává alakítva jutnak el a klienshez, a kliens pedig a megfelelő kivétellé alakítja. *Duplex contract* esetén a kliens is küldhet SOAP hibákat.
A SOAP hibák esetén megkülönböztetünk deklarált és nem deklarált hibát. Deklarált esetben a service metódust meg kell jelölni a *System.ServiceModel.FaultContractAttribute* attribútummal, és fel kell tüntetni a hibát reprezentáló típust. Természetesen több hiba is előfordulhat, az attribútum ismételt használatával azokat is jelezhetjük. Nem deklarált esetben hiányzik az attribútum és a WSDL sem tartalmaz róla információt. Fontos, hogy minden valószínűsíthető hibát jelezzünk, így a kliens fel tud készülni a megfelelő kezelésükre.

Megjegyzés: Csak olyan információkat küldjünk a kliensnek, amelyek feltétlenül szükségesek a hiba kezeléséhez.

Az alábbi ábrán egy *fault contract* megadása látható:

```
[DataContract]
class EmployeeFault
{
	private string _errorMessage;

	public EmployeeFault(string message)
	{
		_errorMessage = message;
	}

	[DataMember]
	public string ErrorMessage
	{
		get
		{
			return this._errorMessage;
		}
	}
}
```

37. ábra – Fault contract megadása

A hiba információit tároló osztálynak szerializálhatónak kell lennie, hogy a csatornára kerülhessen, ezért meg van jelölve a *DataContractAttribute* attribútummal, szerializálandó adattagjai pedig a *DataMemberAttribute* attribútummal.

```
[OperationContract]
[FaultContract(typeof(EmployeeFault))]
public EmployeeInfo GetEmployeeInfo(Employee e)
{
	…
	throw new FaultException<EmployeeFault>(new
		EmployeeFault("Employee not found."));
	…
}
```

38. ábra – Kivétel kiváltása

A *GetEmployeeInfo* service metódusra alkalmazott *FaultContractAttribute* attribútum segítségével megadtuk, hogy a metódus az *EmployeeFault* deklarált hibát dobhatja.

Ahhoz, hogy nem deklarált SOAP hibát küldhessünk, a nem generikus *System.ServiceModel.FaultException* osztály egy példányát kell dobnunk.
Most nézzük, hogy a kliens hogy reagálhat a felmerülő kivételekre. Fontos megemlíteni, hogy a *FaultException*-ön kívül *System.ServiceModel.CommunicationException* és *System.TimeoutException* is előfordulhat. Ezeket is kezelni kell.

A *catch* ágak megadásánál lényeges a sorrend. A generikus *FaultException* osztály ősosztálya a *FaultException*, annak pedig a *CommunicationException*, ezért a leszármazott kivételeket kapjuk el először. Ha a legáltalánosabb kivétellel kezdenénk, akkor minden kivételt elkapnánk, és nem tudnánk reagálni a speciálisabb kivételekre.

```
try
{
      EmployeeInfo info =
      ((IMessageContract)client).GetEmployeeInfo(new Employee());
      client.Close();
}
catch (TimeoutException ex)
{
      // Kivétel kezelése
      client.Abort();
}
catch (FaultException<EmployeeFault> ex)
{
      // Kivétel kezelése
      client.Abort();
}
catch (FaultException ex)
{
      // Kivétel kezelése
      client.Abort();
}
catch (CommunicationException ex)
{
      // Kivétel kezelése
      client.Abort();
}
```

39. ábra – Catch ágak helyes sorrendje

A *service proxy* létrehozásához nem javasolt a *using* utasítás használata, ugyanis az alábbi nem várt működést eredményezheti:

- A *using* utasítás végén lefut az adott erőforrás *Dispose* metódusa, amely jelen esetben ekvivalens a *Close* metódus hívásával. A metódus hívása kivételt eredményezhet a kliens oldalon, tehát annak ellenére, hogy a *using* utasítás törzsében kivételkezelőt alkalmaztunk, kivétel váltódhat ki.
- Ha a *using* utasításban egy nem kezelt kivétel váltódik ki, akkor az elfedésre kerül, ha a *Dispose* metódus kivételt vált ki.

11 Diagnosztikai eszközök

11.1 Teljesítményszámlálók

A teljesítményszámlálók információt nyújtanak az operációs rendszer, szolgáltatások, driver-ek, alkalmazások teljesítményéről. A Windows Communication Foundation teljesítményszámlálók széles körét alkalmazza, melyek segítségével felmérhetjük alkalmazásunk teljesítményét, szűk keresztmetszeteket, rendelkezésre állást, sorbanállást, stb.

A teljesítményszámlálók alapértelmezésként nincsenek engedélyezve a WCF szolgáltatások számára. Engedélyezni az App.config alábbi bejegyzésével lehet:

```
…
<system.serviceModel>
…
    <diagnostics performanceCounters="All" />
…
</system.serviceModel>
…
```

40. ábra – Teljesítményszámlálók engedélyezése konfigurációs fájl segítségével

A *performanceCounters* attribútum lehetséges értékei és ezek jelentése:

- *All*: Az összes WCF számláló kategória (*ServiceModelService 3.0.0.0*, *ServiceModelEndpoint 3.0.0.0*, *ServiceModelOperation 3.0.0.0*) engedélyezett.
- *ServiceOnly*: A *ServiceModelService 3.0.0.0* kategória engedélyezett.
- *Off*: A három kategória egyike sem engedélyezett (alapértelmezés).

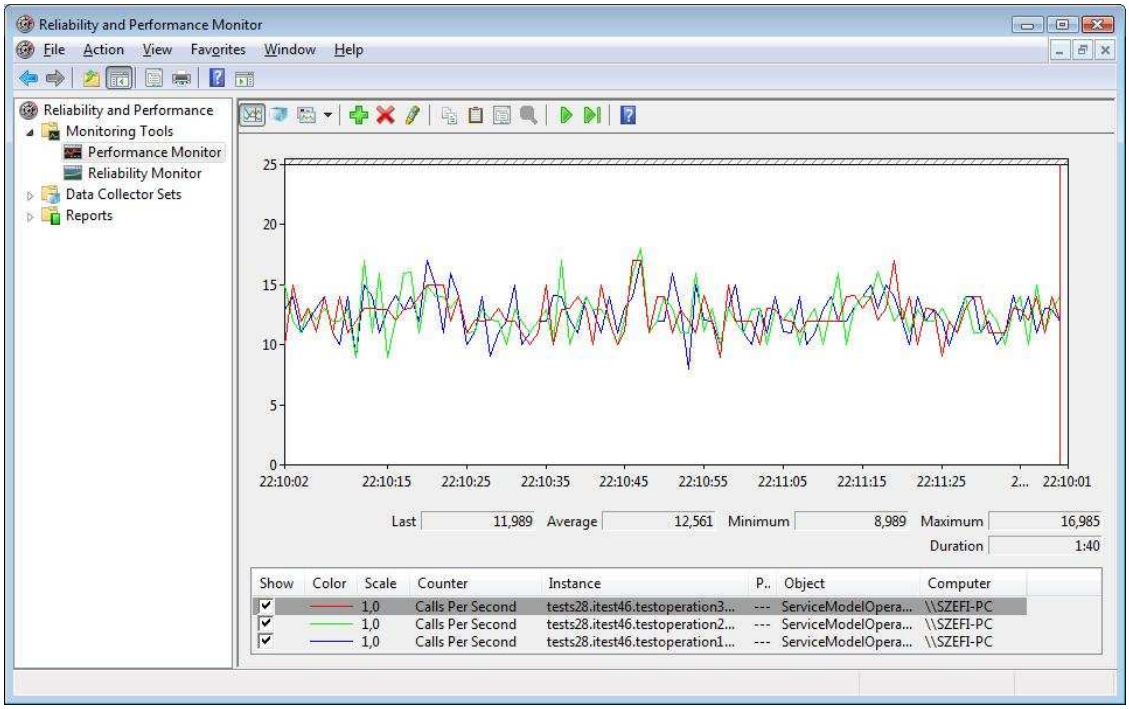
Ha az összes szolgáltatásra szeretnénk engedélyezni a teljesítményszámlálókat, akkor a Machine.config konfigurációs fájlban is elhelyezhetjük a fent említett bejegyzést. Mivel a

Machine.config fájl bejegyzéseiből örököl az összes konfigurációs fájl, illetve annak bejegyzéseit terheli, nem kell az egyes szolgáltatások fájljaiban engedélyezni a teljesítményszámlálókat.

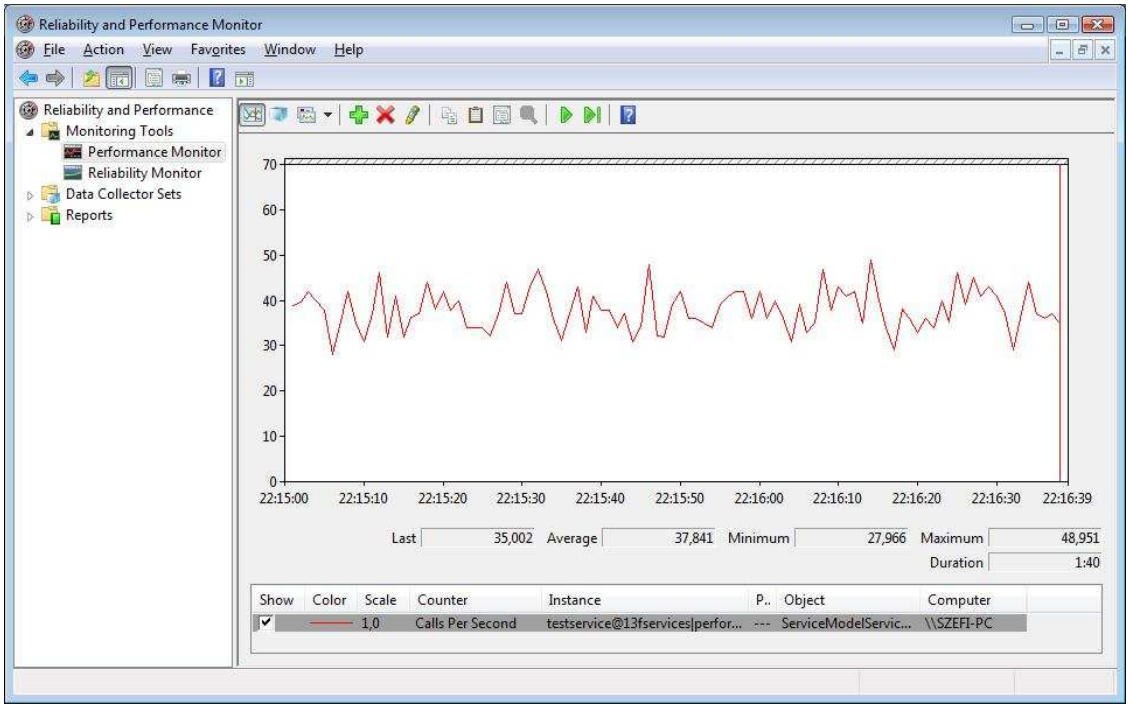
A számlálók által szolgáltatott adatok megtekinthetők a *Windows* operációs rendszer *Performance Monitor* (perfmon.exe) eszközével, melyet a vezérlőpult adminisztrációs eszközei között találunk meg.

A teljesítményszámlálók működésének szemléltetésére létrehoztam egy WCF szolgáltatást 3 metódussal, melyeket 10 kliens hív.

A következő ábrák a szolgáltatás egyes metódusain történő hívások, illetve a szolgáltatáshoz érkező kérések másodpercenkénti számát mutatják.



41. ábra – A szolgáltatás egyes metódusain történő hívások másodpercenkénti száma



42. ábra – A szolgáltatáshoz érkező kérések másodpercenkénti száma

11.2 Nyomkövetés

A nyomkövetés segítségével elemezhetjük, hogy miként működik az alkalmazásunk, milyen kivételeket dob, milyen életciklus események következtek be, milyen hívások történtek. Erre támogatást nyújt a WCF. Ahhoz, hogy engedélyezzük a nyomkövetést, definiálnunk kell a forrást (*trace source*), a nyomkövetés szintjét és egy *listener*-t. Ha több forráshoz is ugyanazt a *listener*-t szeretnénk használni, akkor közös (*shared*) *listener*-ként is megadhatjuk. A Windows Communication Foundation minden egyes *assembly*-jéhez definiál egy forrást. Ezek közül a *System.ServiceModel* a legáltalánosabb.
A nyomkövetés szintjét a *switchValue* attribútummal állíthatjuk be.

Lehetséges értékei:

- Off

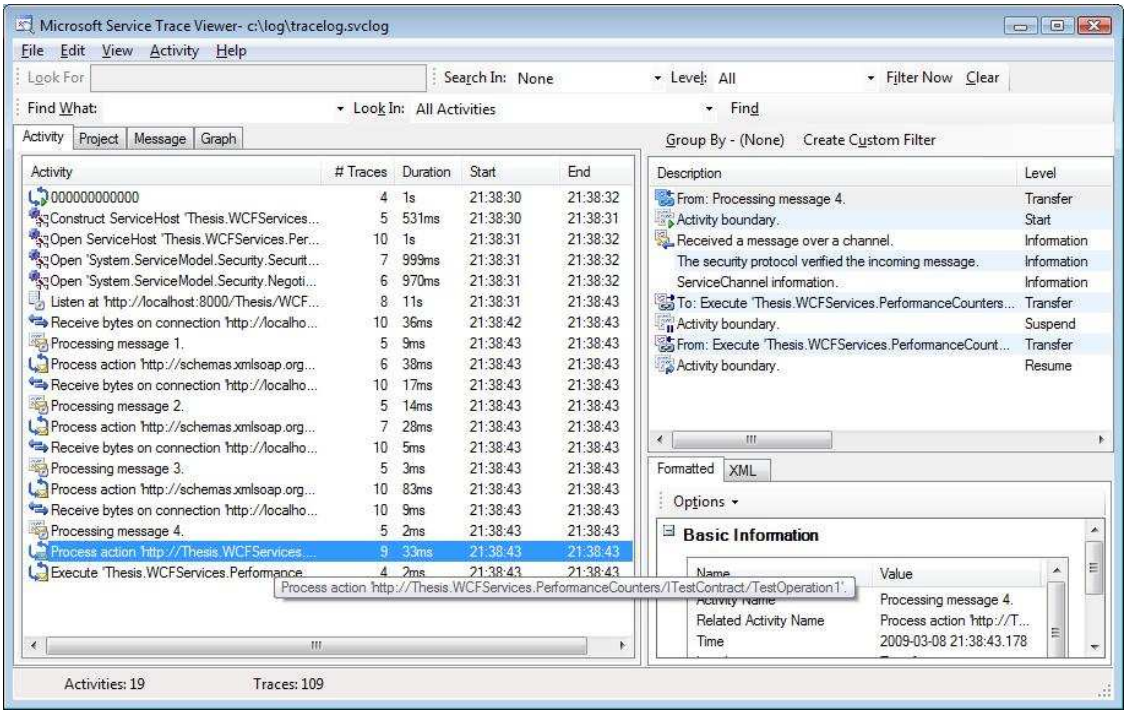
- Critical
- Error
- Warning
- Information
- Verbose
- ActivityTracing
- All

melyekből többet is megadhatunk vesszővel elválasztva. A nyomkövetést engedélyezhetjük a kódban, de célszerű konfigurációs fájlban (web.config, app.config) beállítani, mert annak módosítása nem igényli az alkalmazás újrafordítását és a szerverre történő ismételt kihelyezését (deployment).

```
<system.diagnostics>
    <sources>
      <source name="System.ServiceModel"
              switchValue="Information, ActivityTracing" >
        <listeners>
          <add name="traceListener"
               type="System.Diagnostics.XmlWriterTraceListener"
               initializeData= "c:\Log\TraceLog.svclog" />
        </listeners>
      </source>
    </sources>
</system.diagnostics>
```

43. ábra - Nyomkövetés beállítása konfigurációs fájlban

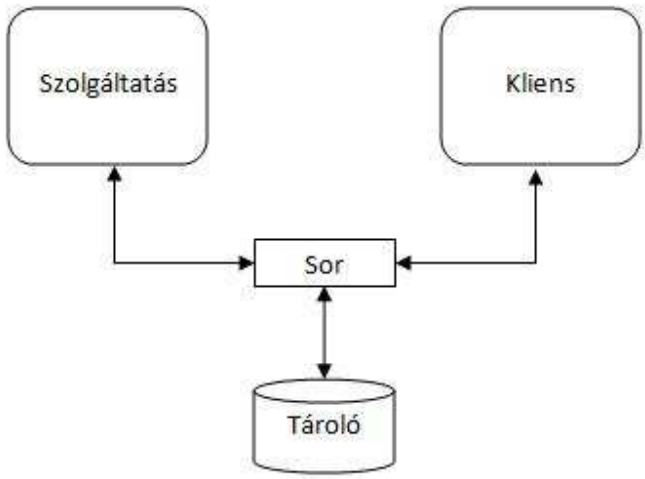
A fenti konfigurációs beállítás a *c:\Log\TraceLog.svclog* fájlba gyűjti a nyomkövetési információkat. Ezeket a bejegyzéseket megtekinthetjük a *Service Trace Viewer* programmal.



44. ábra – Tracing bejegyzések megtekintése

12 WCF és MSMQ

Az elosztott rendszerek szempontjából fontos kérdés, hogy a szolgáltatás és a kliens között a kommunikáció direkt vagy sem. Direkt esetben, ha a hálózat nem működik megfelelően (például egy kábel nincs csatlakozva), vagy csak nem indult el a szolgáltatás, akkor a kommunikáció nem lesz sikeres, fontos üzenetek veszhetnek el. Ezt a problémát megszüntethetjük, ha sor alapú kommunikációt valósítunk meg, azaz a szolgáltatás és a kliens(ek) között az üzenetek egy átmeneti tárolása jön létre. A sor természetesen független lehet a szolgáltatást és/vagy a klienst hosztoló géptől.



45. ábra – Sor alapú kommunikáció

Lehetőség van az üzeneteket egy tranzakció részeként a sorra küldeni, illetve onnan egy tranzakció részeként kiolvasni. Amennyiben a küldés során a tranzakció visszagörgetésre került, a sorban nem lesznek benne a tranzakció részeként küldött üzenetek. Ha a kiolvasás is tranzakciós és a tranzakció sikertelen, az üzenetek változatlanul a sorban maradnak.

A WCF támogatja a Microsoft által implementált sor alapú, operációs rendszer szintű transzportot, az *MSMQ*-t (*Microsoft Message Queuing*). Ekkor a küldő alkalmazástól érkező üzeneteket egy sor tárolja, majd továbbítja a fogadó félnek.

Az *MSMQ* használatának előnyei:

- Lazán csatolt alkalmazások jöhetnek létre, ugyanis a küldő és fogadó alkalmazások függetlenek attól, hogy a másik elérhető-e.
- A kommunikációt megbízhatóvá teszi még akkor is, ha a hálózat nem az. Ha a sorba bekerültek az üzenetek, akkor a fogadó alkalmazás fel tudja dolgozni azokat, ha elérhetővé válik.
- Egy szerver alkalmazás több példánya is olvashatja ugyanazt a sort, ezáltal a terhelés elosztható.

Ahhoz, hogy egy kliens alkalmazás üzenetet tudjon küldeni egy szolgáltatásnak, a célsort kell megcímeznie, és a szolgáltatásnak is az azon a címen található sort kell figyelnie.
A cím a következő mintára épül:

net.msmq://<hosztnév>/[private/]<sornév>

A kommunikáció során a kliens nem kap garanciát arra, hogy a szolgáltatás mikor dolgozta, dolgozza fel a küldött üzeneteket, ezért használjunk *one-way* metódusokat a *service contract*-ban.

Az *MSMQ* használatához engedélyeznünk kell a megfelelő Windows komponenst a vezérlőpultban, majd a *Computer Management* programban létrehozhatjuk a használni kívánt sort. Ezt természetesen a kódban is megtehetjük, illetve vizsgálnunk is kell, hogy létezik-e. Amennyiben nem, akkor hozzuk létre.

Tekintsük a következő szolgáltatást, amely *MSMQ*-t használ a garantált kézbesítéshez:

```
[ServiceContract]
interface IMSMQ
{
      [OperationContract(IsOneWay=true)]
      void SendMessage(string message);
}

public class MSMQService : IMSMQ
{
      public void SendMessage(string message)
      {
            Console.WriteLine("A klienstől a következő üzenet" +
            "érkezett: {0} ({1})", message, DateTime.Now);
      }
}
```

46. ábra – A service contract és implementációja

A *SendMessage* metódus egyirányú, mert a kliens nem várhatja meg, amíg a szolgáltatás megkapja az adott üzenetet (az is lehet, hogy nincs elindítva). A metódus hívása azt eredményezi, hogy a konzolon megjelenik a kliens által paraméterként átadott üzenet, illetve az aktuális időpont.

A hosztoláshoz szükséges kód és a hozzá tartozó konfigurációs fájl a következő ábrákon látható.

```
string queueName = ConfigurationManager.AppSettings["queueName"];

if (!MessageQueue.Exists(queueName))
{
      MessageQueue.Create(queueName, true);
}

Uri baseAddress = new Uri("http://localhost:8080/MSMQService");
using (ServiceHost host =
                  new ServiceHost(typeof(MSMQService), baseAddress))
{
      ...
      host.Open();
      Console.WriteLine("A szolgáltatás elérhető" +
                        "({0})", DateTime.Now);
      ...
```

47. ábra – A hoszt kódja

A sor nevét a konfigurációs fájlban célszerű tárolni, ugyanis ez az adat változhat, és ebben az esetben nem kell az alkalmazást újrafordítani.

```
<configuration>
  <appSettings>
    <add key="queueName" value=".\private$\MyQueue"/>
  </appSettings>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="MSMQBehavior"
               name="Thesis.WCFServices.MSMQService">
        <endpoint address="net.msmq://localhost/private/MyQueue"
                  binding="netMsmqBinding"
                  bindingConfiguration="MsmqConf"
                  contract="Thesis.WCFServices.IMSMQ"/>
      </service>
    </services>
    <bindings>
      <netMsmqBinding>
        <binding name="MsmqConf">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MSMQBehavior">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

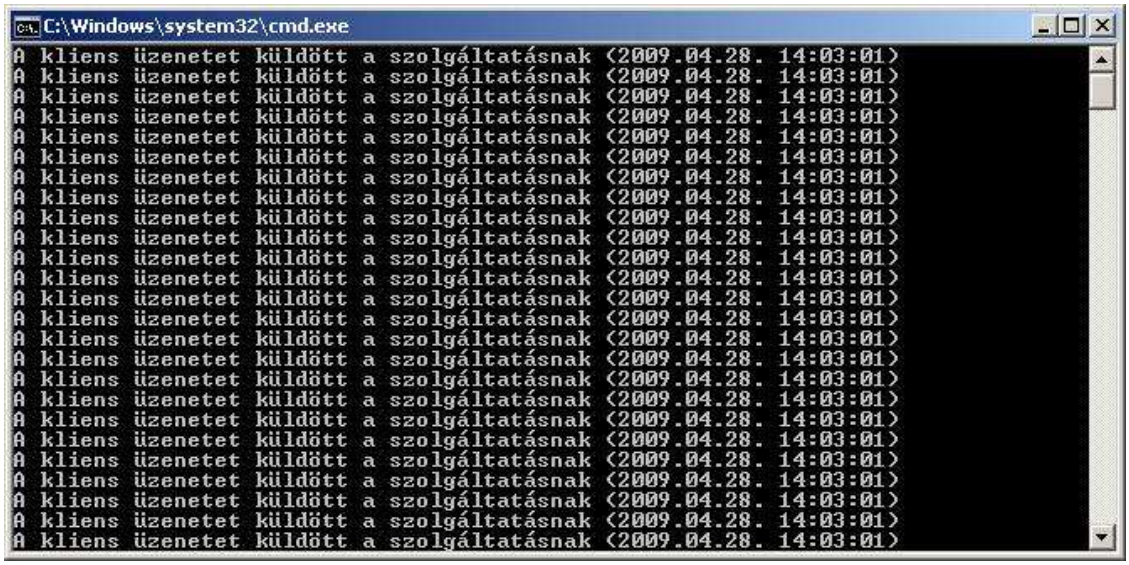
48. ábra – A konfigurációs fájl

A konfigurációs fájl *appSettings* bejegyzése tartalmazza a sor nevét. A *binding*-ot az *MSMQ*-t támogató *netMsmqBinding*-ra állítottam, illetve a fentebb megadott címzési sémát követtem.
A kliens számára a *service proxy*-t a *ServiceModel Metadata Utility Tool* segítségével lehet generálni.

```
...

MSMQClient client = new MSMQClient();

for (int i = 0; i < 100; i++)
{
      client.SendMessage("Hello!");
      Console.WriteLine("A kliens üzenetet küldött a " +
                        "szolgáltatásnak ({0})", DateTime.Now);
}

...
```

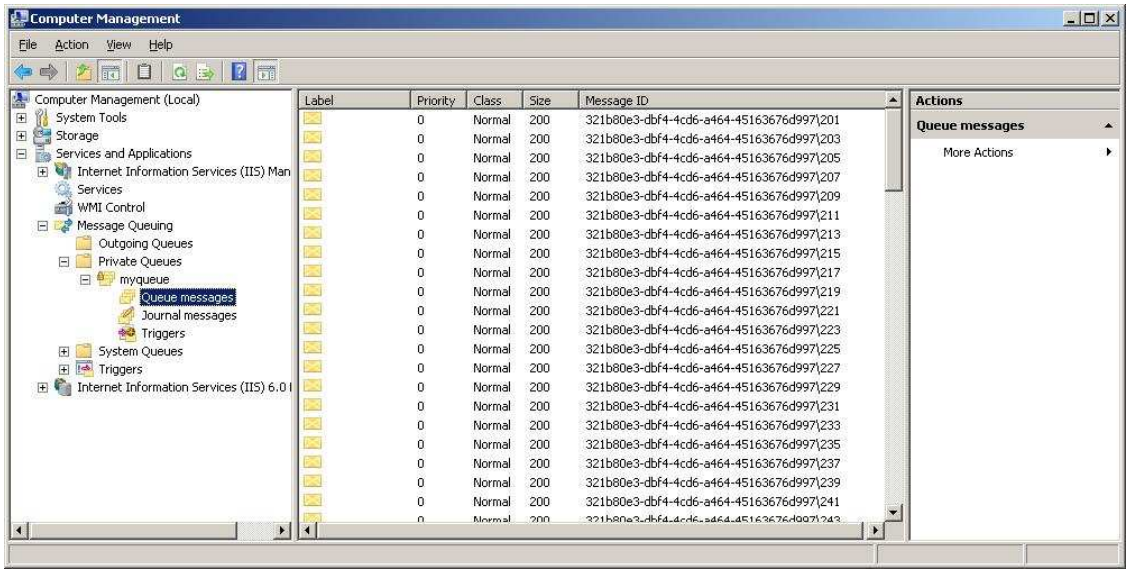
49. ábra – A kliens működése

Látható, hogy a kliens százszor hívja meg a *SendMessage* metódust. Azért, hogy szemléltessem a sor használatát, nem indítottam el a szolgáltatást, de a klienst igen. A kliens oldali kimenet a következő:



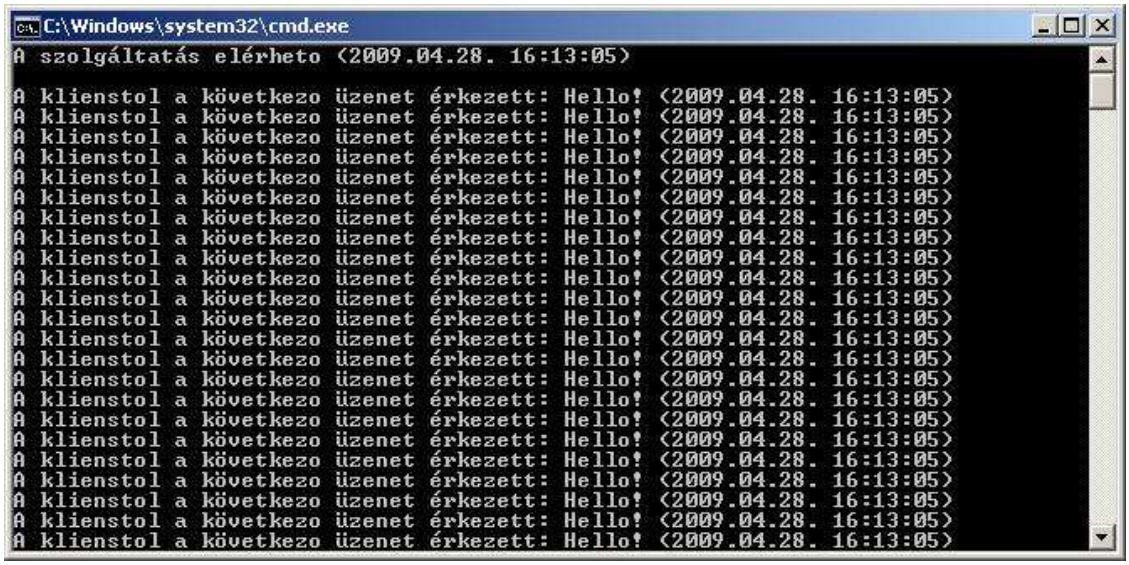
50. ábra – A kliens kimenete

A kliens hiba nélkül hívhatta a szolgáltatás által publikált metódusokat. A küldött üzeneteket a *Computer Management* program segítségével megtekinthetjük. Amíg a fogadó oldal nem kapja meg az üzenetet, addig az a sorban marad.



51. ábra – A sorban lévő üzenetek

Ha elindítjuk a szolgáltatást, akkor az feldolgozza a sorban lévő üzeneteket, és a következő kimenetet szolgáltatja:



52. ábra – A szolgáltatás kimenete

Látható, hogy a szolgáltatás körülbelül két órával az üzenetek küldése után indult el, de a kézbesítés sikeres volt.

13 Összefoglalás

Diplomamunkám célja az volt, hogy ismertessem a *Windows Communication Foundation* szolgáltatások írásához nélkülözhetetlen fogalmakat, példákon keresztül bemutassan az általam fontosnak tartott technológiai lehetőségeket. A munkám és tanulmányaim során szerzett ismeretek, tapasztalatok alapján fontosnak tartottam, hogy beszámoljak a szinkron és aszinkron programozási modellről, a tranzakciós működésről (erről egyetemi tanulmányaim során számos tárgy kapcsán tanultam), a munkamenetekről és nagy vonalakban az *MSMQ*-ról. Ezeket a célokat sikerült elérnem, azonban a diplomamunka készítése során szembesültem azzal, hogy a bevezetőben ismertetett tervezési célok megvalósításainak részletes tárgyalása már nem fér e dolgozat keretei közé, de a későbbiekben szeretnék ezzel komolyabban foglalkozni.

14 Köszönetnyilvánítás

Szeretnék köszönetet nyilvánítani témavezetőmnek, Dr. Fazekas Gábor egyetemi docensnek a diplomamunkámat érintő hasznos tanácsokért, valamint a Debreceni Egyetem oktatóinak, akik munkájukkal hozzájárultak szakmai fejlődésemhez és tanulmányaim alatt segítségemre voltak, továbbá köszönettel tartozom Szeifert Péternének és Bölkény Edinának, hogy diplomamunkám írása során mindenben támogattak.

15 Irodalomjegyzék

Chris Peiris, Dennis Mulder, Shawn Cicoria, Amit Bahree, Nishith Pathak: Pro WCF – Practical Microsoft SOA Implementation (2007)

Craig McMurtry, Marc Mercuri, Nigel Watling, Matt Winkler: Windows Communication Foundation Unleashed (2007)

Tony Northrup, Shawn Wildermuth, Bill Ryan : Microsoft .NET Framework 2.0 Application Development Foundation Training Kit (2006)

Glenn Johnson, Tony Northrup: Microsoft .NET Framework 2.0 Web-Based Client Development Training Kit (2007)

Andrew Troelsen: Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition (2007)

http://msdn.microsoft.com/hu-hu/library/ms735119(en-us).aspx

http://msdn.microsoft.com/hu-hu/library/bb400851(en-us).aspx

http://msdn.microsoft.com/hu-hu/library/aa139617%28en-us%29.aspx