Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Nagy Ákos

HATÉKONY SOROSÍTÁS DINAMIKUS KÓDEMITTÁLÁSSAL .NET PLATFORMON

KONZULENS

Kővári Bence András

BUDAPEST, 2011

Tartalomjegyzék

Összefoglaló

A szolgáltatásorientált architektúra lehetővé teszi a fejlesztők számára, hogy lazán kapcsolódó és platformfüggetlen módon hozzanak létre komplex rendszereket, amelyek üzleti vagy más munkafolyamatokat támogatnak. Ezen rendszerek elemei elosztott módon működnek, így kulcsfontosságú a kommunikáció megbízhatósága és hatékonysága.

Az architektúrát a .NET keretrendszer a Windows Communication Foundation (WCF) osztálykönyvtárral támogatja. A csomag felel egyebek mellett azért is, hogy a küldendő és a vett adatokat megfelelő formátumra alakítsa küldés előtt ill. vétel után. Ez utóbbi a sorosítás folyamata.

Számos esetben azonban mind a szolgáltatások, mind pedig az azokat igénybe vevő szoftverek .NET platformon futnak – ez pedig lehetőséget ad arra, hogy optimalizáljuk platformspecifikus adatok felhasználásával vagy épp elhagyásával a kommunikációt.

Bár a WCF tartalmaz implementációt erre a speciális esetre is, méréseink során úgy találtuk, hogy ennek hatékonysága sokszor nemhogy jobb, hanem rosszabb, mint az általános esetre megírt variánsnak. Célunk tehát az volt, hogy a fenti megoldásoknál gyorsabb komponenst adjunk a fejlesztők kezébe.

Miután egyedi sorosítókkal sikerült olyan mérési adatokat produkálni, amelyek a már rendelkezésre álló eszközök mért adatainál jóval kedvezőbbek voltak, különböző, általánosabb, a lehető legtöbb adatszerkezetre működő sorosítókat dolgoztunk ki folyamatosan szem előtt tartva a hatékonyságot, végső célként azt kitűzve, hogy a lehető legjobban meg tudjuk közelíteni a kevésbé általános megoldások időigényét.

Eredményeink azt mutatják, hogy van gyorsabb alternatíva a „beépített” megoldásoknál, amelyek felhasználásával időt spórolhatunk meg a kommunikáció során – ez pedig mind az alkalmazások üzemeltetői, mind a felhasználói oldaláról mérhető teljesítményjavulást hozhat.

1 Bevezetés

Ahogy a technológia fejlődésével egyre több helyről és egyre gyorsabban lehetett kapcsolódni különböző hálózatokhoz, a szoftveriparban megjelent az igény ennek a lehetőségnek a kihasználására. Erre az igényre született meg válaszként az elosztott rendszerek koncepciója, azaz különböző számítógépeken futó szoftverek együttműködésének lehetősége valamilyen közös cél érdekében. Könnyen látható, hogy a hálózaton történő együttműködésre az egyetlen járható út az, ha a rendszer számítógépei - illetve az azon futtatott programok – valamilyen, előre lefektetett protokoll szerint üzeneteket cserélnek; ezekben nem csak „hasznos” adatokat, hanem a kommunikáció vezérléséhez szükséges segédadatokat is elhelyezhetnek.

Míg az alkalmazások alkotják a rendszer gerincét, addig az üzenetek felelnek azért, hogy ez a gerinc működőképes legyen és helyesen működjön. Fontos tehát, hogy ezek az üzenetek tömörek legyenek és a lehető leggyorsabban átérjenek a hálózat egyik végéből a másikba, valamint hogy könnyen fel lehessen dolgozni őket – így a lehető legkevesebb erőforrást veszünk el a tényleges feladat végrehajtásától.

Mindezekkel az igényekkel párhuzamosan továbbra is léteztek olyan kritériumok, mint a platformfüggetlenség, a lazán csatoltság vagy az üzleti folyamatok könnyű irányíthatósága, illetve voltak kidolgozott architektúrák bizonyos problémák hatékony megoldására elosztott környezetben (például a kliens-szerver architektúra), amelynek előnyei megkérdőjelezhetetlenek voltak már akkor - és természetesen megvoltak a maguk hátrányai. Ebből a háttérből született meg az az architektúra, ami a dolgozat egyik fő építőkockáját adja: a szolgáltatás-orientált architektúra (SOA).

A dolgozat másik nagy építőeleme, a .NET keretrendszer, a Microsoft saját programozási platformja saját virtuális géppel, osztálykönyvtárral, típusrendszerrel, fordítóval, programnyelvekkel és szabványokkal. Ez a keretrendszer lehetőséget ad a fejlesztőknek arra, hogy gyorsan, hatékonyan tudjanak fejleszteni – a hatalmas osztálykönyvtár és folyamatos bővítései az adatbáziseléréstől kezdve, grafikus felületek fejlesztésén át a webes vagy mobilos kliensek létrehozásáig mindenre adnak megoldást. Ezek után talán nem meglepő, hogy saját implementációja van a SOA architektúra elemeire is.

A dolgozat fő célja, hogy egy hatékony sorosítómotort adjon a fejlesztők kezébe, ha olyan alkalmazásokat kell elkészíteni, amelyek megfelelnek a SOA architektúrának, SOAP üzenetekkel kommunikálnak, de mind egységesen .NET platformra készültek.

1.1 A .NET keretrendszer

A .NET keretrendszer egy az iparban is széles körben használt programozási környezet, amivel a legkülönfélébb igényeknek megfelelni képes szoftvereket, szoftverkomponenseket vagy akár szoftverrendszerek készíthetünk. A platform sokszínűsége nem teszi lehetővé a dolgozat keretei között a rendelkezésre álló eszközök teljes listájának még vázlatos leírást sem, de a dolgozatom szempontjából fontosabb részeket röviden bemutatom a következő néhány szakaszban.

1.1.1 Dinamikus kódgenerálás

A keretrendszer lehetőséget ad arra is, hogy olyan programokat írjunk, amelyek programokat – vagy legalábbis valamilyen kódot – hoznak létre. Lényegében az az API áll a fejlesztők rendelkezésére explicit módon, amit maga a keretrendszer is használ bizonyos feladatok végrehajtásához (például a reguláris kifejezések használata esetén erősen támaszkodik a keretrendszer erre az API-ra) [1]. Természetesen ahhoz, hogy hatékonyan tudjuk használni ezt a kicsit nehézkesnek tűnő felületet, nem elég azt önmagában ismerni, tisztában kell lennünk a keretrendszer működésével is.

1.1.2 .NET IL kód és végrehajtása

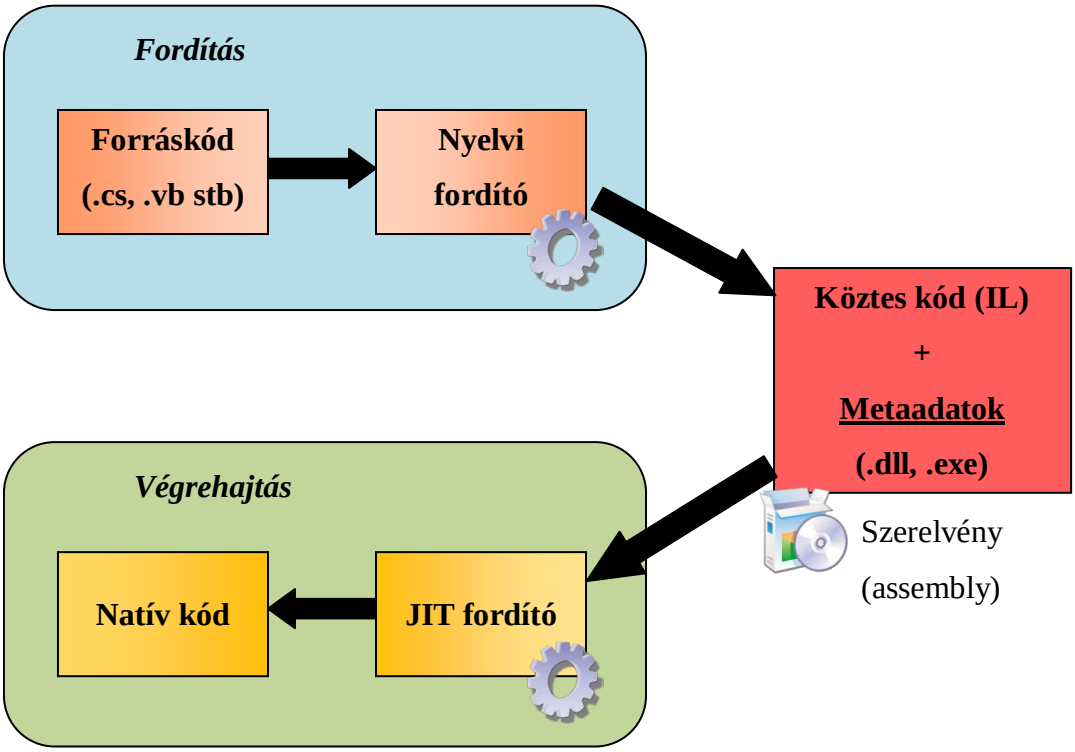
A .NET egyik nagy előnye már bevezetésétől kezdve, hogy különböző programnyelveken dolgozhatnak a fejlesztők úgy, hogy mind kihasználhatják a keretrendszer lehetőségeit és akár egymás, különböző programnyelveken megírt komponenseit is felhasználhatják. Mindemellett nem szükséges újabb programnyelveket elsajátítaniuk; ha pedig több programozási nyelvet is ismernek, adott feladathoz kiválaszthatják azt, amelyikkel a megoldás a legkönnyebben leírható.

Mindez úgy lehetséges, hogy van egy közös nyelv, amire a fenti programnyelvek bármelyikén megírt kód előzetesen lefordul – a .NET keretrendszer futtatókörnyezete pedig ezt a programkódot tudja majd értelmezni. Ez a köztes programnyelv az IL (Intermediate Language). A Microsoft szabványosította ezt a nyelvet Common Language Specificationként (CLI) az ECMA 335-ös szabványban. Röviden érdemes

átnézni, hogy hogyan is működik a rendszer illetve mely elemei fontosak a szabványnak a dolgozat szempontjából.

1.1.2.1 Fordítás és futtatás

A fordítás során elsőként ez a bizonyos IL nyelvű leírás áll elő. Ez az a kód, ami a futtatható állományokban és egyéb szerelvényekben „benne van” bájtok sorozataként. Ez a kód kerül majd feldolgozásra futtatáskor a JIT (Just-in-Time) fordító által, ami az első híváskor vagy telepítéskor natív kódra fordítja az IL kódot.



1. ábra - Fordítás és futtatás .NET platformon [2]

1.1.2.2 A kiértékelő verem

A kód futtatása kiértékelő verem alapú (független attól a veremtől, ahol a lokális változóink és hívási paramétereink vannak). Egy művelet meghívása során a következőképpen járunk el:

1. Betöltjük a verembe a művelet argumentumait a művelet specifikációjának megfelelő sorrendben.
2. Kiadjuk a parancsot a műveletre vonatkozóan.
3. A művelet az argumentumokat adott sorrendben kiveszi a veremből.

4. A művelet végrehajtódik.
5. Az eredmény (ha volt), a kiértékelő verem tetejére kerül.

Azt, hogy pontosan milyen argumentumokra és milyen sorrendben van szükség, azt a művelet leírásából kideríthetjük (lásd [3] vagy [4]).

1.1.2.3 Utasítások

Maga az ECMA 335 szabvány két különböző típusú utasítást ír le. Az egyik csoportba az ún. objektumkezelő utasítások (object model instructions) tartoznak; ezeket használhatjuk objektumok létrehozására, dobozolásra, kidobozolására, típuskényszerítésére, a kiértékelő verembe töltésre vagy onnan levételre, lokális változókból történő betöltésre, változókba történő tárolásra. Minden, ami nem sorolható az előző kategóriába (matematikai és logikai műveletek, a program futását vezérlő utasítások, metódushívások, konstansok betöltése, a verem közvetlen kezelése), az ún. alapvető utasítások (basic instructions) csoportjába tartozik.

A jobb áttekinthetőség szempontjából az utasításokat másképpen csoportosítva érdemes röviden bemutatni.

- **Matematikai és logikai műveletek:** Betöltjük a verembe a művelete operandusait, majd a művelet ezeket kiveszi onnan és az eredményt visszarakja a verem tetejére.
- **A program vezérlését szabályozó utasítások:** Ezek általában valamilyen értéket várnak a veremben, amelyről eldöntik, hogy hamis/null/0 és ez alapján ugranak egy meghatározott címkére. Kivételként érdemes 4 kategóriát említeni:
    - **Metódushívások:** A metódusaink nagy részét érdemes a virtuális metódustáblán keresztül irányítani, így nem csak az argumentumokat, hanem egy objektumpéldányt is a verembe kell tölteni, amelyen elvégezzük a hívást. Ez statikus metódusoknál természetesen nem lehetséges [5]. További elvárt paraméterek a meghívandó metódus argumentumai.
    - **Switch:** Ez az utasítás egy címkelistát vár, és a verem tetején lévő elem értékétől függően ugrik az adott címkék közül a megfelelőre.
    - **Ret:** A visszatérés művelete – minden metódus kötelezően ezzel zárul [4]. Ha a metódusnak specifikáltunk visszatérési értéket, akkor egy ilyen típusú elem kell, hogy legyen a verem tetején, a Ret ezt kiveszi és visszaadja; void visszatérésnél csupán kilép a metódusból. Fontos, hogy a kiértékelő vermet a metódus kilépése után olyan állapotban kell hagyni, amilyenben belépéskor volt.
    - **Kivételkezelő utasítások**
- **Utasítások a verem kezelésére:** Idetartoznak az utasítások, amelyekkel elemeket (akár konstansokat, akár változók értékeit) lehet a verembe betölteni illetve azok az utasítások, amelyekkel elemeket lehet a veremből kivenni.

- **Utasítások lokális változók, argumentumok és mezők kezelésére:** Idetartoznak a lokális változók elérésére, az argumentumok manipulálására és a mezők betöltésére illetve elérésére szolgáló utasítások.
- **Típuskezelő utasítások:** A típuskényszerítés, típuskonverzió, be- és kidobozolás utasításai.

1.1.2.4 Típusok

Fontos, hogy a különböző programnyeleken ne csak az utasítások, hanem a használt adattípusok is kompatibilisek legyenek. Ezt biztosítják a Common Type System definíciói, ami része az ECMA 335 szabványnak. Ebben többek között találunk leírást arra, hogy hogyan nézzenek ki az egyes típusok, hol kell őket tárolni, mikor tekinthetőek azonosnak, hogyan működik az öröklés, a kényszerítés, de még elnevezési szabályokat is ad a szabvány és leírja saját csoportosítási rendszerében a típusokat (lásd az A függelék: A CTS típusrendszere című részben).

A dolgozat szempontjából a következőket érdemes tudni:

- **Referenciatípus vagy értéktípus:** Egy adattípus értéktípus, ha az adatot a saját, allokált memóriaterületén tárolja. Egy referenciatípus egy mutatót tartalmaz egy másik memóriaterületre, ami tárolja az adatot [6]. A megkülönböztetés számunkra is fontos lesz, hiszen az értéktípusok példányait nem tudjuk olyan helyen közvetlenül használni, ahol referenciatípust várnak, dobozolást kell végrehajtani.
- **Generikus típus:** A szabvány szerint az a típus generikus, aminek van generikus argumentuma. Kicsit gyakorlatiasabban megközelítve: a List<T> osztály generikus, generikus argumentuma a T. T bármilyen típus lehet majd a felhasználás során. Ilyen formában a generikus típus nyitott, amint pontosan megmondjuk, milyen típust is takar T (List<int>), a típus zárt.
- **Primitív típus:** Primitívnek nevezzük azokat a típusokat, amikhez van közvetlen nyelvi támogatás. Ez természetesen programozás nyelvtől függően változik; a C# primitív típusok listáját a B függelék: A C# primitív típusai című rész mutatja be.
- **Atomi típus:** Ez saját definíciónk. Később, a sorosításkor látni fogjuk, hogy mivel minden típus csak ilyen primitív típusokból áll, ezért ezek képviselik az adott objektum állapotát, így valójában ezek kerülnek sorosításra. Lényegében a primitív típusokat tartalmazza az alábbi kivételekkel (a teljes listát lásd C függelék: Atomi típusok című részben).
    - **object:** Ez nem tekinthető atomi típusnak, hiszen minden osztály öröklődik a System.Object osztályból, tehát bármi lehet egy ilyen típusú mezőben.
    - **dynamic:** A dynamic típusú objektum lényegében egy speciális attribútummal ellátott object típusú példány, így a fentiek alapján ez sem tekinthető atomi típusnak.
    - A **DateTime** típus bekerült az atomi típusok közé – ennek technikai megfontolásai vannak.

- **Minden Enum** típus atomi, hiszen valójában csupán speciális szemantikával bíró egész számokról van szó.

1.1.2.5 Egy IL állomány vizsgálata

Példaként nézzük meg a List<T> osztályhoz tartozó Add metódus IL kódjának egy részletét:

```
.method public hidebysig newslot virtual
              instance void  Add(object key, object 'value')
              cil managed
{
  // Code size 103 (0x67)
  .maxstack 5
  .locals init       (int32 V_0, object[] V_1)
  IL_0000: ldarg.1
  IL_0001: brtrue.s  IL_0018
  IL_0003: ldstr     "key"
  IL_0008: ldstr     "ArgumentNull_Key"
  //...
  IL_003c: newarr    System.Object
  IL_0041: stloc.1
  IL_0042: ldloc.1
  IL_0043: ldc.i4.0
  IL_0044: ldarg.0
  IL_0045: ldloc.0
  //...
  IL_005c: ldarg.0
  IL_005d: ldloc.0
  IL_005e: not
  IL_005f: ldarg.1
  IL_0060: ldarg.2
  IL_0061: call      instance void
                     System.Collections.SortedList::Insert
                                        (int32,object,object)
  IL_0066: ret
} // end of method SortedList::Add
```

Minden utasítás külön sorba kerül. A sor elején látható a címke, ezután az utasítás szöveges azonosítója, majd az argumentumok, ahol lehet nevesítve. Fontos azonban, hogy ezek mind csak a könnyebb olvashatóság érdekében vannak így leírva. Valójában csupán egy bájtfolyamról van szó, amit az értelmezéshez használt program számunkra átalakít.

A címke egyáltalán nem része a kódnak. Generáláskor tudunk definiálni címkéket, és tudunk hivatkozni rájuk, de ezek is csak a gördülékenyebb fejlesztés miatt működnek így.

Az utasításokat természetesen szintén számok azonosítják – ez az azonosító az ún. műveletkód (operation code, opcode). Ez után pedig jönnek az argumentumok. Az

argumentumok lehetnek konstansok, ilyenkor a konstans megfelelő reprezentálása kerül ide, de – mint ahogy a call vagy a newarr esetében látjuk – lehetnek típusok, metódusok és egyéb CLR szerkezetek. Az IL kódban azonban ezek neve helyett szintén egy szám szerepel – ez az ún. token. A token minden ilyen szerkezetre egyedi az őt tartalmazó modulon belül – innen tudja az értelmező is, hogy melyik CLR szerkezet áll argumentumként az adott utasítás mögött. Az argumentumok száma és hossza utasításonként változik: lehet 0 bájt (operandus nélküli utasítás), 1 bájt (az utasítások ún. rövid formája esetén), 2 bájt, 4 bájt, 8 bájt vagy 4 bájt egyéb egész többszöröse (ez utóbbi csak a switch műveletkód esetén [7]).

1.1.3 A Reflection.Emit kódgenerálás-modell

A Reflection.Emit névtér olyan osztályokat tartalmaz, amelyek segítségével mi magunk tudunk futási időben CLR szerkezeteket létrehozni és ezeket akár azonnal, akár később felhasználni. Minden egyes ilyen szerkezethez (Assembly, Module, Type, Field, Property, Method, LocalVariable) tartozik egy-egy információs leíró osztály, amelyben megtaláljuk egy konkrét entitáshoz tartozó metaadatokat (például Method esetén a visszatérési érték típusa, maga a metódus IL bájtkódja, a tartalmazó modul stb.).

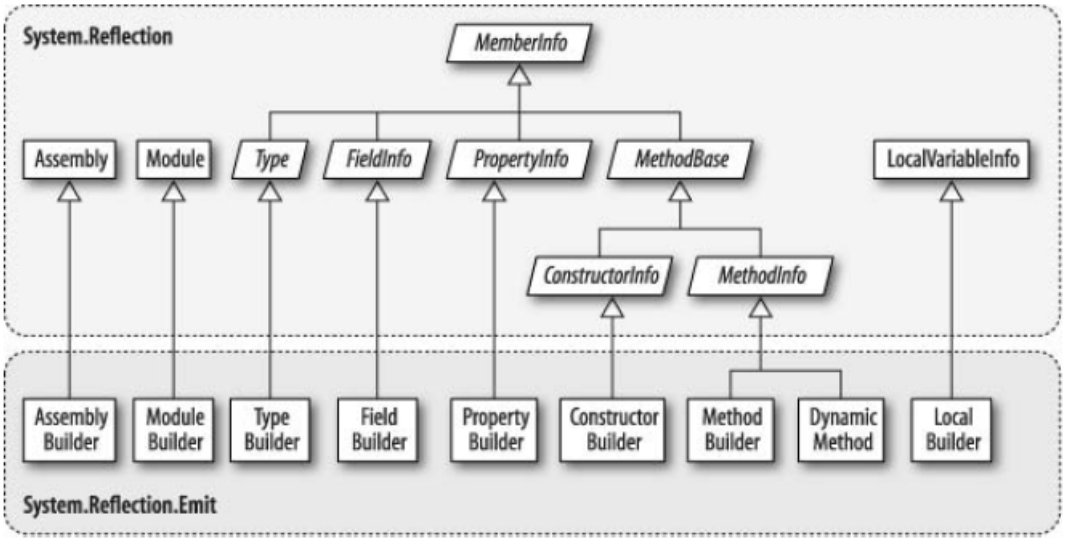
Minden egyes ilyen leíró osztályból származik egy olyan osztály, amellyel a megfelelő CLR szerkezetet mi magunk is elő tudjuk állítani – ezek az osztályok (AssemblyBuilder, ModuleBuilder, TypeBuilder, FieldBuilder, PropertyBuilder, MethodBuilder, LocalBuilder) alkotják az API magját.

Ezek az osztályok tehát egyrészt azért fontosak, mert segítségükkel tudunk futási időben, dinamikusan kódot előállítani. De ezen felül van még egy fontos felhasználási lehetőség, amit a későbbiek során ki is használunk majd: mivel minden Builder osztály valamilyen Info osztályból származik, így azok „helyett” is felhasználó az öröklés alapvető tulajdonságai miatt.

Hol hasznos ez? Az adott szerkezet kezdetben egy ún. „létrehozatlan” állapotban van, azaz őket magukat nem tudjuk a szokásos reflexiós (Reflection) hívásokkal feltérképezni – tehát például egy TypeBuilder-rel létrehozott típusnak nem tudjuk a metódusait lekérdezni (általában is igaz, hogy nem tudunk olyan hívásokat végrehajtani, amelyek bármilyen MemberInfo leszármazottat adnak vissza). Ahhoz, hogy ezt megtehessük, a típust „meg kell sütni” (bake), azaz valójában létre kell hozni. Egy CLR típus (és minden egyéb CLR szerkezet) azonban megváltoztathatatlan (immutable), azaz ha egyszer létrehoztuk, már nem tudunk rajta módosítani. Hogyan oldható meg tehát,

hogy egy ilyen módon definiált típusban két dinamikusan létrehozott metódus egymást tudja hívni?

A válasz a fentiek ismeretében már egyszerű – nem kell meghívni azt a függvényt, amivel a metódusinformációkat le tudjuk kérni egy MethodInfo objektumba, egész egyszerűen használható helyette a MethodBuilder objektum, amivel létrehoztuk a metódust. Valójában miután a típust „megsütöttük”, ezek a MethodBuilder objektumok átalakulnak speciális proxykká a valódi metódusok felé. Az IL kódban a hívott metódust egy speciális token azonosítja majd – ez pedig közös lesz a MethodBuilder-ben és a neki megfelelő metódusban.



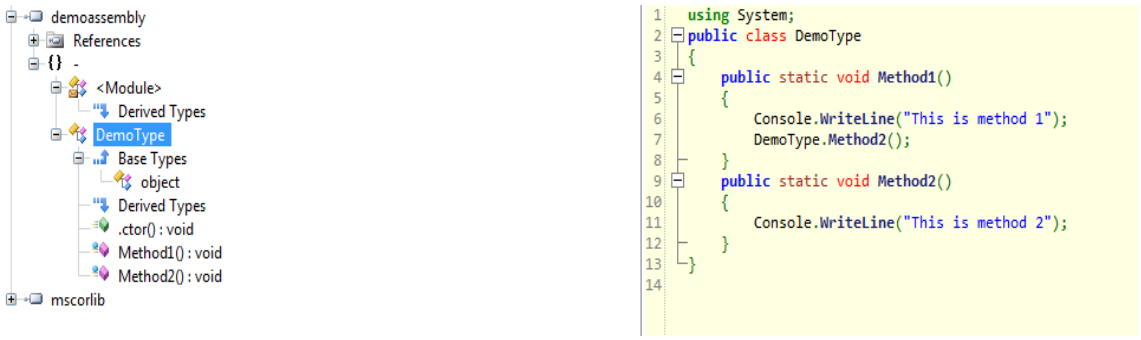
2. ábra - A Reflection.Emit kódgenerálás-modell [19]

Fontos még megemlíteni az ILGenerator osztályt – ennek példányai teszik lehetővé magát a kódemittálást.

Az alábbi példában egy egyszerű típust definiálunk két statikus metódussal az API alapjainak használatát bemutatva:

```
class Program
{
  static void Main(string[] args)
  {
   //Létrehozunk egy szerelvényt
   AssemblyBuilder assembly =
         AppDomain.CurrentDomain.DefineDynamicAssembly
                          (
                             new AssemblyName("DemoAssembly"),
                             AssemblyBuilderAccess.RunAndSave
                           );
    //...definiálunk egy modult...
    ModuleBuilder module =
      assembly.DefineDynamicModule
                               (
                           "DemoModule",
                           "demoassembly.dll"
                           );
    //...definiálunk egy típust...
    TypeBuilder type = module.DefineType
                         (
                            "DemoType",
                            TypeAttributes.Public
                         );
    //... és két metódust.
    MethodBuilder method1 = type.DefineMethod
                         (
                            "Method1",
                             MethodAttributes.Public |
                             MethodAttributes.Static,
                             typeof(void),null
                          );
    MethodBuilder method2 = type.DefineMethod
                         (
                             "Method2",
                             MethodAttributes.Public |
                             MethodAttributes.Static,
                             typeof(void),null
                          );
    //A metódus törzsét fel kell tölteni kóddal.
    ILGenerator mgen1 = method1.GetILGenerator();
    mgen1.EmitWriteLine("This is method 1");
    //Nem MethodIndo, hanem MethodBuilder az argumentum
    mgen1.Emit(OpCodes.Call, method2);
    //Minden metódus utolsó művelete a visszatérés
    mgen1.Emit(OpCodes.Ret);
    ILGenerator mgen2 = method2.GetILGenerator();
    mgen2.EmitWriteLine("This is method 2");
    mgen2.Emit(OpCodes.Ret);
    //A típus "megsütése", majd a metódus futtatása
    type.CreateType().GetMethod("Method1").Invoke(null,null);
    //Mentés a merevlemezre
    assembly.Save("demoassembly.dll");
    }
  }
```

A fenti kód lefuttatása után létrejön a demoassembly.dll nevű állomány, amelynek tartalmát egy reflectorral megvizsgálhatjuk. A következő szerelvény generálódott:



3. ábra - A generált szerelvény visszafejtése ILSpy segítségével

A fenti példa, bár nagyon egyszerű, mégis tartalmaz majdnem mindent a Reflection.Emit API-ból és jól mutatja az osztályok felhasználását.

1.2 A szolgáltatásorientált architektúra

Mivel dolgozatomban alapevően a webes szolgáltatások optimalizálásával foglalkozok, szeretném néhány gondolattal a szolgáltatás-orientált architektúrát, mint modellt bemutatni és pontosan behatárolni, hol is helyezkedik el a SOA a WCF implementációban, illetve a WCF a SOA alkalmazások fejlesztésének eszköztárában.

1.2.1 SOA, mint modell

A SOA egy speciális architekturális tervezési minta. A mintában az alkalmazásokat felépítő funkcionális egységek a szolgáltatások - ezek a funkcionális egységek pedig az elvárt előfeltételeikkel (például műveleti paraméterek) és a műveleti céljaikkal (például visszatérési érték) definiáltak. Fontos eleme az architektúrának, hogy ezek pontosan definiálják a műveletet; az, hogy a művelet hogyan van implementálva, csak másodlagos. Így tehát könnyen tudunk olyan szolgáltatásokat létrehozni, amelyek lazán csatoltak és platformfüggetlenek, de mégis elosztott módon működnek.

A dolgozat szempontjából fontos még a SOA modellből az, hogy a szolgáltatásokat futtató host és az azt igénybevevő kliens üzenetekkel kommunikál. Mivel a platformfüggetlenség fontos szempont volt az architektúra kialakításakor, ezért az üzenetek formátumát szabvány rögzíti – ez a SOAP (Simple Object Access Protocol). Ez egy XML alapú protokoll, ami pontosan meghatározza a kicserélhető

üzenetek formátumát. Például egy megrendelő alkalmazás a következő formátumban tudja a saját, memóriabeli reprezentációját SOAP üzenetbe csomagolni [8]:

```
<soap:Envelope>
  <soap:Header>
    <TickerSymbol>MSFT</TickerSymbol>
    <CompanyName>Microsoft</CompanyName>
  </soap:Header>
  <soap:Body>
    <LastTrade>29.24</LastTrade>
    <Change>0.02</Change>
    <PreviousClose>29.17</PreviousClose>
    <AverageVolume>59.31</AverageVolume>
    <MarketCapital>287.44</MarketCapital>
    <PriceEarningRatio>24.37</PriceEarningRatio>
    <EarningsPerShare>1.20</EarningsPerShare>
    <_52WkHigh>29.40</_52WkHigh>
    <_52WkLow>21.45</_52WkLow>
  </soap:Body>
</soap:Envelope>
```

1.2.2 A SOA és a WCF kapcsolata

A WCF (Windows Communication Foundation) egy osztálykönyvtár, ami a .NET keretrendszert egészíti ki kommunikációs funkciókkal. Fontos, hogy a kommunikáció a WCF esetében nem korlátozódik egyszerűen egy Microsoftos SOA implementációra; ezen kívül rengeteg egyéb kommunikációs forma programozását és konfigurálást megkönnyíti [9]. A dolgozat szempontjából azonban csak ez a része fontos, így ez kerül röviden bemutatásra.

A WCF-ben minden megvan, ami egy SOA alkalmazás elkészítéséhez kell – a megírt alkalmazásaink tudnak más platformokkal is kommunikálni és ehhez SOAP üzeneteket használnak. A memóriában élő objektumokat először azonban valahogy olyan formára kell hozni, hogy az átküldhető legyen egy SOAP üzenetben – ez az átalakítási folyamat a sorosítás (serialization), ennek az inverz művelete (tehát amikor az üzenet alapján rekonstruálunk egy, az adott platformon is használható, de az eredetivel tartalmában ekvivalens objektumot) a visszasorosítás (deserialization). Erre a feladatra a WCF az ún. DataContractSerializer osztály használja alapértelmezésben.

Előfordulhat az is, hogy minden kommunikáló alkalmazás .NET keretrendszerre lett írva, ilyenkor a platformfüggetlenség előnyeit sokszor már nem akarjuk kihasználni. Erre a speciális esetre a WCF a NetDataContractSerializer osztályt ajánlja, ez azonban – mint látni fogjuk – meglehetősen lassú.

1.3 A WCF saját sorosítási megoldásai

Ahhoz, hogy egy gyors sorosítót kidolgozzunk, érdemes megismerkedni a jelenlegi megoldások működési elveinek alapjaival és megvizsgálni a gyengepontjaikat.

1.3.1 A DataContractSerializer

A DataContractSerializer osztály a WCF osztálykönyvtár alapértelmezett megoldása a szolgáltatások és a kliensek közötti kommunikációban részt vevő adatok sorosításra [10]. Ahogyan láttuk, ebben a helyzetben fontos az interoperabilitás – ennek a követelménynek a DataContractSerializer teljesen meg is felel. Az adatokat XML formátumba sorosítja egy rögzített szabvány szerint, így tehát más platformokon sem lehet probléma feldolgozni az adatokat. A 0 fejezetben bemutatott példaosztály sorosított formájának egy részlete például így néz ki (a teljes adatfolyam az D függelék: A beépített sorosítók kimeneti formátuma részben látható):

```
<?xml version="1.0" encoding="utf-8"?>
<Employees xmlns:i=http://www.w3.org/2001/XMLSchema-instance
           xmlns="http://schemas.datacontract.org/
                                2004/07/EmployeeLib">
  <Address>507 - 20th Ave. E. Apt. 2A</Address>
  <BirthDate>1948-12-08T00:00:00</BirthDate>
  <City>Seattle</City>
  <Country>USA</Country>
  <EmployeeID>1</EmployeeID>
  <Extension>5467</Extension>
  <FavouriteNumbers xmlns:d2p1="http://schemas.microsoft.com/
                                  2003/10/Serialization/Arrays">
    <d2p1:int>1478526959</d2p1:int>
    <d2p1:int>125403918</d2p1:int>
    <d2p1:int>243117508</d2p1:int>
```

Látható, hogy sehol nem használja fel a .NET típusok metaadatait. Másrészről meglehetősen terjengős a formátum.

A DataContractSerializer nem tudja „magától” kideríteni, hogy milyen típust is kell sorosítania, ezt nekünk a konstruktorban specifikálni kell (természetesen egy szolgáltatás leírásakor vagy igénybevételekor ez a színfalak mögött történik) [11]. Másrészt pedig az öröklés kezelése is nehézkes. Mivel a típusrendszerről a sorosító nem rendelkezhet közvetlen információval a leszármazott típusokat explicit módon meg kell jelölnünk az őstípusban, vagy a szolgáltatásinterfészben [12].

A fentiek mind szükségesek a megfelelő interoperabilitás miatt. Ha azonban a SOA kommunikáció .NET-.NET alkalmazások között zajlik, hiányként vagy éppen

feleslegként jelentkeznek, kényelmetlen használatot és többletterhelést eredményeznek. Erre próbál megoldást adni a NetDataContractSerializer.

1.3.2 A NetDataContractSerializer

A NetDataContractSerializer szintén egy XML-alapú formátumot állít elő az objektumainkból, de ez már felhasználja a CLR típusinformációkat is [13]. Íme egy részlet a sorosított példaosztály kimeneti XML-folyamából (a teljes folyamot lásd D függelék: A beépített sorosítók kimeneti formátuma):

```
<?xml version="1.0" encoding="utf-8"?>
<Employees xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
     z:Id="1" z:Type="EmployeeLib.Employees"
           z:Assembly="ConsoleApplication1, Version=1.0.0.0,
                       Culture=neutral, PublicKeyToken=null"
    xmlns:z=http://schemas.microsoft.com/2003/10/Serialization/
    xmlns="http://schemas.datacontract.org/2004/07/EmployeeLib">
  <Address z:Id="2">507 - 20th Ave. E. Apt. 2A</Address>
  <BirthDate>1948-12-08T00:00:00</BirthDate>
  <City z:Id="3">Seattle</City>
  <Country z:Id="4">USA</Country>
  <EmployeeID>1</EmployeeID>
  <PostalCode z:Id="13">98122</PostalCode>
  <Region z:Id="14">WA</Region>
  <SensitiveData
           Culture=neutral,PublicKeyToken=b77a5c561934e089],
           [System.Int32, mscorlib, Version=2.0.0.0,
           Culture=neutral,PublicKeyToken=b77a5c561934e089]][]"
           z:Assembly="0" z:Size="2" xmlns="">
     <KeyValuePairOfstringint
                 xmlns="http://schemas.datacontract.org/
                            2004/07/System.Collections.Generic">
       <key z:Id="20">salary</key>
       <value>1531733115</value>
     </KeyValuePairOfstringint>
     <KeyValuePairOfstringint
                 xmlns="http://schemas.datacontract.org/
                            2004/07/System.Collections.Generic">
       <key z:Id="21">numOfChildren</key>
       <value>117442512</value>
     </KeyValuePairOfstringint>
    </KeyValuePairs>
  </SensitiveData>
  <Title z:Id="22">Sales Representative</Title>
  <TitleOfCourtesy z:Id="23">Ms.</TitleOfCourtesy>
</Employees>
```

Bár az örökléssel kapcsolatos problémák megszűnnek és bármilyen típust képes ugyanaz a példány sorosítani, a formátum még mindig meglehetősen terjengős: például a sorosítás sorrendjének jelölése még mindig megvan, és emellett verzióinformációink is vannak, amire szintén számos esetben nincs szükségünk. Ezen felül magának a

szerelvények a jelölését ugyan tömöríti, de a típus jelölését nem. Mindezeken felül ráadásul – ahogy látni fogjuk – maga a sorosítás nagyon lassú.

2 Egyéni sorosítási megoldások

A technikai-technológiai háttér leírása után a következő fejezetekben bemutatásra kerül egy saját sorosítómotor kidolgozása; elsőként definiálok egy mérési módszertant, amivel jellemezhetőek az egyes sorosítók, majd bemutatom, hogyan írhatóak le a beépített megoldások ezzel a metrikával. Ezek után pedig az egyeditől elindulva fokozatosan kidolgozok egy általános, de gyors sorosítót. Mindeközben folyamatosan bemutatom az egyes fázisokban kidolgozott sorosítók előnyeit és hátrányait.

2.1 A mérések módszertana

A mérések során végig a SOA architektúra keretein belül dolgoztam. Egy host (konzolos .NET alkalmazás) hostol egy szolgáltatást – ennek műveletei pedig semmit nem csinálnak, csak void értékkel visszatérnek. A kliensből (szintén konzolos .NET alkalmazás) ezeket szinkron módon hívom, és mérem egyrészt a teljes hívás idejét, másrészt az átküldött adat mennyiségét. Más adatra nincs szükségünk, hiszen ezekkel a sorosítók jól leírhatóak. Az átküldött adat mennyisége egy adott sorosítóra természetesen ugyanannyi minden híváskor, időben azonban jelentős ingadozás tapasztalható a WCF kommunikációs és architekturális megoldásai miatt [14], így nem az első, hanem a huszadik hívások idejét fogjuk összehasonlítási alapul használni.

A fentebb említett adatok könnyebb összehasonlításának kedvéért definiálok két mérőszámot; ezekre a továbbiakban „átviteli idő” és „kimenet mérete” néven hivatkozok.

Az „átviteli idő” mérőszámának értéke lényegében nem más, mint az átküldött objektumok (a 0 fejezetben bemutatott példaosztály példányai) számának függvényében mért hívásidő görbének a meredeksége úgy, hogy lineáris összefüggést feltételezünk a kettő között. Praktikusan tehát azt mérjük, hogy hány milliszekundum kell ahhoz, hogy egy objektumot át tudjunk küldeni a hálózaton. Mértékegysége is erre utal, ms/db. A lineáris összefüggés első körben csupán feltételezés, de a görbéken látni fogjuk, hogy jó közelítéssel valóban lineáris kapcsolatról van szó.

A „kimenet mérete” érték hasonlóan definiált, a sorosított objektumok számának a függvényében mért sorosított adathossz görbéjének meredeksége ugyanígy lineáris összefüggést feltételezve. Gyakorlatiasabb megfogalmazásban: hány kilobájt szükséges

ahhoz, hogy egy objektumot le tudjunk írni a sorosított adatfolyamunkban. A lineáris összefüggés itt is csupán feltételezésként született, de a mérések azt mutatták, hogy valóban lineáris a kapcsolat a két mért érték között, tehát jogosan használhatjuk ezt a mértéket is. Mértékegysége kbyte/db.

A görbéknél a tengelymetszettel nem foglalkozok külön – egyrészt, mert ez nagyjából ugyanannyi a mérések alapján minden módszerre, másrészt nagy objektumszámnál ez már nem meghatározó.

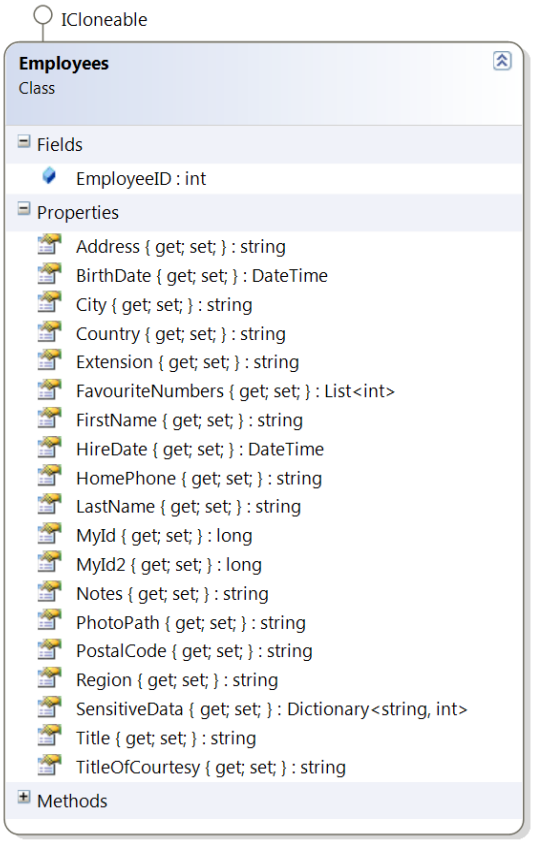
Ezek a teljesítménymérők nem csak azért jók, mert összehasonlíthatóvá teszik a módszereket (értelemszerűen mindkét esetben a minél kisebb érték a kedvező), hanem azért is, mert a linearitás miatt számszerűsíteni is tudjuk köztük a különbséget – ha egy adott módszernél az átviteli idő értéke 2 ms/db, egy másiknál 0,5 ms/db, akkor ez utóbbi módszer négyszer gyorsabb az előbbinél. Hasonló igaz a kimenet méretének értékére az adatfolyamok hosszának összehasonlítására vonatkozóan.

Minden egyes kidolgozott sorosítónál lemértem az átviteli időt és a kimenet méretét 1, 10, 50, 100, 500, 1000, 2000, 5000, 10000, 15000, 20000, 25000, 50000 objektumnál, majd kiszámoltam a fent definiált teljesítménymutatókat.

A szolgáltatáshívások során 10 különböző objektum klónjait küldöm át – ez azonban (különösen a nagy mérési tartományokban) nem okoz túlságosan nagy eltérést a lineáris modelltől, mivel a nagyobb tartományban 10 valamilyen többszöröse az átküldött objektumok száma.

2.2 A használt példaosztály

A tesztelések és mérések során használt példaosztály kialakításakor szempont volt, hogy egy, az éles alkalmazásokban is gyakran használt típushoz hasonló osztállyal dolgozzunk, másrészt elég nagy legyen ahhoz, hogy méréseink hiteles eredményeket adhassanak, harmadrészt viszonylag könnyen generálhatóak legyenek az osztály példányai. Ezeket szem előtt tartva a Northwind adatbázis Employees táblájának rekordszerkezete adta a mi példaosztályunk alapját – ennek mérete azonban nem volt kielégítően nagy, ezért különböző, azonosító jellegű tulajdonságokat, időpontokat reprezentáló tulajdonságokat és gyűjteményeket vettem még fel az osztályba – ez utóbbiakban mindig ugyanannyi elem szerepelt (3 a listában és 2 a szótárban). A könnyű generálhatóság miatt a példaadatbázisból Linq2SQL technológiával beolvastam az összes rekordot, majd ezeket klónoztam.



4. ábra - Az Employees példaosztály szerkezete

2.3 Mérések a beépített sorosítási megoldásokra

Elsőként érdemes számszerűsíteni azt, amit a korábbi fejezetekben mérési módszertan hiányában csak úgy emlegettünk, mint „lassú” vagy „terjengős” – miért is gondolom, hogy ezeknél a komponenseknél gyorsabbat is lehetne készíteni?

2.3.1 A DataContractSerializer

Az alapértelmezett megoldás teljesítményét a következő táblázat foglalja össze:

Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	514	1,071
10	514	9,774
50	529	48,290
100	546	96,435
500	561	481,591
1000	600	963,036
2000	639	1 925,927
5000	842	4 814,599
10000	1185	9 629,052
15000	1470	14 443,505
20000	1840	19 257,958
25000	2105	24 072,411
50000	3987	48 144,677

1. táblázat - A DataContractSerializer eredményei

Az objektumszám-hívásidő függvény jó közelítéssel tekinthető lineárisnak, így jól alkalmazható a sorosító leírására a bevezetett átviteli idő mérőszám; ennek értéke erre a módszerre 0,068 ms/db.

Az adathossz görbe teljesen lineáris, így a kimenet méretére vonatkozó mérőszámmal nagyon jól leírható a módszer adathossz-igénye; ennek értéke 0,963 kbyte/db.

Ezek tehát az alapértelmezett megoldás teljesítményindikátorai. Ezekkel fogjuk összehasonlítani a későbbi megoldásoknál számolt értékeket.

2.3.2 A NetDataContractSerializer

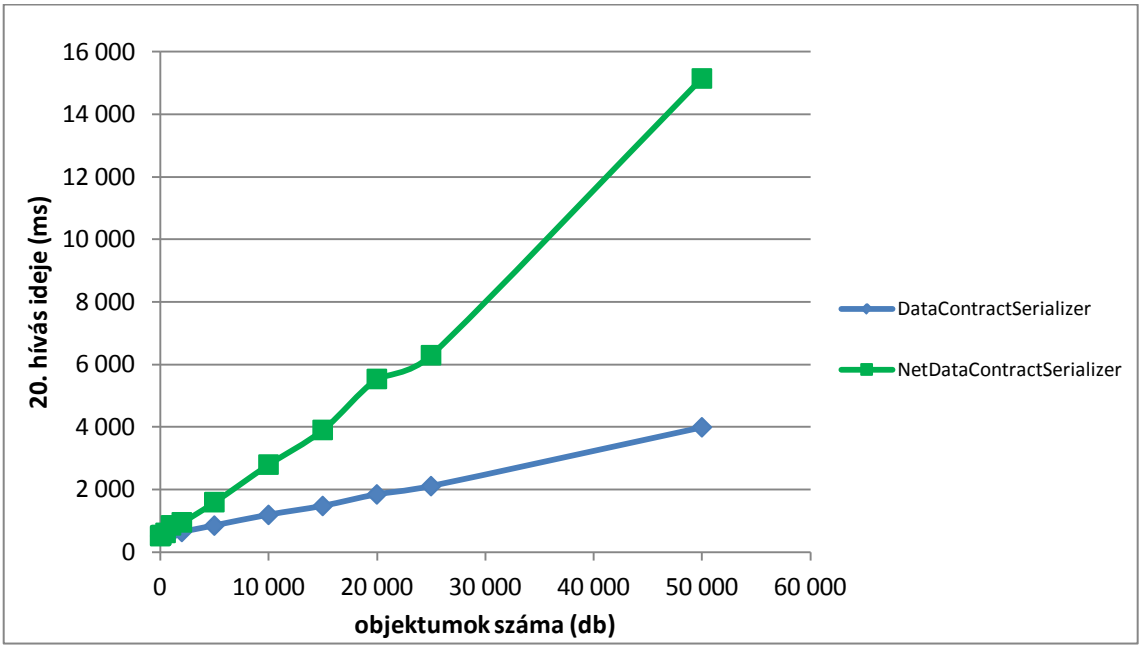
A fenti méréseket NetDataContractSerializer használatakor is elvégezve a következő eredményeket kapjuk:

Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	511	2,546
10	514	21,666
50	532	106,618
100	541	212,820
500	609	1 064,892
1000	850	2 130,322
2000	947	4 265,108
5000	1593	10 678,194
10000	2794	21 366,672
15000	3902	32 060,052
20000	5533	42 782,708
25000	6292	53 505,364
50000	15145	107 118,646

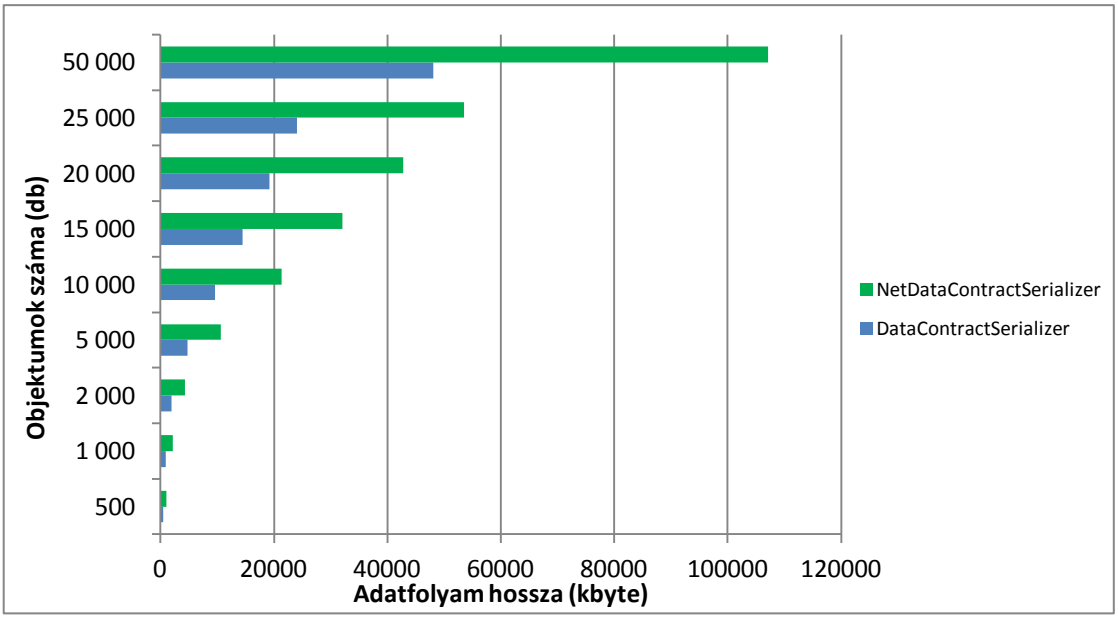
2. táblázat - A NetDataContractSerializer eredményei

Ennek a táblázatnak az értékeit a DataContractSerializer méréseit tartalmazó táblázat (1. táblázat - A DataContractSerializer eredményei) eredményeivel összehasonlítva látható, hogy mind az átviteli idő, mind a kimenet méretének értéke jóval nagyobb (0,279 ms/db illetve 2,141 kbyte/db; emlékeztetőül a DataContractSerializerre ezek az értékek 0,068 ms/db illetve 0,963 kbyte/db).

A görbéken a különbség szemmel láthatóan jelentős.



5. ábra – DataContractSerializer és NetDataContractSerializer átviteli idejei



6. ábra - DataContractSerializer és NetDataContractSerializer kimeneteinek mérete

A NetDataContractSerializer tehát egyértelműen nem jelent megoldást a problémára. Valójában nem is vártunk jobb teljesítményt – a NetDataContractSerializernek nem az az előnye, hogy gyorsabb vagy tömörebb, mint a DataContractSerializer, hanem az, hogy képes kezelni a CLR típusinformációkat [13].

Mivel a NetDataContractSerializer eredményei még a DataContractSerializer eredményeinél is rosszabbak, így ezt nem fogom referenciaként kezelni a későbbiek során. Ha a végcélként kifejleszteni kívánt sorosító gyorsabb lesz a DataContractSerializernél, akkor a NetDataContractSerializernél is hatékonyabban tud majd dolgozni. Mivel azonban kidolgozni kívánt sorosító .NET-.NET alkalmazások kommunikációjára lesz optimalizálva, így az is fog rendelkezni a NetDataContractSerializer egyetlen előnyével: a CLR típusinformációk kezelésének képességével.

2.4 Egy saját megoldás előkészítése és mérések

Adottak tehát az eredmények, amelyeknél jobbat szeretnénk elérni. A következőkben lépésről lépésre bemutatom az ehhez vezető megoldás kifejlesztését.

2.4.1 Irányelvek

Fontos lefektetni, hogy pontosan mi is kerül sorosításra egy adott típusból. Részben a különböző, már meglévő .NET sorosítók működését alapul véve, részben saját technikai-technológiai megfontolások következtében a következőkhöz tartom magam:

- Egy típus publikus, nem readonly mezői sorosításra kerülnek.
- Egy típus publikus tulajdonságai, amikhez van publikus setter, sorosításra kerülnek.
- A fenti kettőtől eltérő tagok nem kerülnek sorosításra.
- Először a tulajdonságokat dolgozom fel, aztán a mezőket (ez csupán megállapodás kérdése).
- Minden objektumnak jelzem a típusát.
- Mind a mezőket, mint a tulajdonságokat névsorrendben dolgozom fel (mivel minden tagnak egyedi névvel kell rendelkeznie, így bármely két alkalommal ugyanúgy történik a sorosítás egy adott típusra.
- A gyűjtemények feldolgozása során jelzem, hogy hány elem van a gyűjteményben.
- Csak olyan típusok kerülnek sorosításra, amikhez van alapértelmezett konstruktor; kivétel ez alól a tömb, az atomi típusok és néhány egyéb értéktípus (hiszen a konstruktorba nem tudnánk a megfelelő paramétereket megadni).
- Generikus típusokat zárt formában dolgozom fel; azaz például egy List<int> nem úgy kerül sorosításra, mint List<T> külön jelezve, hogy T=int.
- Fontos, hogy végig a SOA architektúra korlátain belül maradok – azaz ugyanúgy üzenetekkel segítségével történik a kommunikáció, mint eddig, az üzenetek ugyanúgy SOAP üzenetek, csak a tartalmuk más.
- Az üzenetek Body része egyetlen elemből áll, ebbe írom bele a sorosítómotorunk által előállított adatfolyamot Base64 kódolással [15], és innen olvasom ki a másik oldalon.
- A sorosítás első lépése a típusfeltérképezés. Ennek során feljegyezem az objektumgráfban szereplő összes típust. Később, amikor egy objektumot sorosítok, a típusát úgy jelezem, hogy megnézem, hányadik helyet foglalja el ebben a listában és ezzel hivatkozok rá. Az öröklés helyes kezelése miatt ezt muszáj megtenni.
- Az atomi típusok értékei kerülnek sorosításra – ha nem atomi a sorosítandó típus, akkor az adatfolyamban jelzem a típusát, majd attól függően, hogy valamilyen gyűjtemény vagy egyéb komplex típus, hasonló módon feldolgozom az elemeit vagy az előzőleg meghatározott kritériumnak megfelelő mezőit/tulajdonságait.

Fontos megjegyezni, hogy ezek csak az első megközelítés irányelvei – a későbbiek során ezek módosulhatnak (lásd a 2.4.4.2 és a 2.4.4.3 fejezetekben).

2.4.2 Sorosító motorok egyetlen típusra

Ezekben a megoldásokban kihasználtam azt, hogy tudom, milyen típust szeretnénk sorosítani és csak erre készítettem fel az adott sorosítót. Ipari környezetben természetesen ez a megoldás használhatatlan – a cél ezekkel egyrészt az volt, hogy lássam, lehet-e egyáltalán a beépített megoldások teljesítményigénye alá menni, másrészt hogy kapjak egy referenciaértéket arra vonatkozóan, hogy mi az az érték, aminél jobbat valószínűleg nem lehet produkálni.

Könnyen látható, hogy ezek a megoldások a lehető leggyorsabbak, hiszen ezek során semmilyen típusfeltérképezést nem használnak – egyetlen típust fogadnak el,

annak tulajdonságait és mezőit adott sorrendben sorosítják (ezekről persze szintén lehet tudni, milyen típusúak). A gyűjteményobjektumok hossza bekerül a sorosított adatfolyamba, de ezekről szintén lehet tudni, milyen típusú elemeket tárolnak. Így semmilyen reflexiós hívás nincs a kódban, ami a sorosítást végzi – egyszerű mező- és tulajdonságértékek lekérdezése illetve valamilyen formában történő kiírása, majd ennek megfelelően visszaolvasása és az értékek beállítása.

Az előző szakaszban felsorolt irányelvek közül azokat, amelyek értelmezhetőek voltak, szemmel tartottam az egyedi sorosítók esetében is.

2.4.2.1 FixedStringSerializer

Első megoldásként egyetlen karakterláncba fűztem össze a mezők és tulajdonságok megfelelő szöveges reprezentációit (azaz például a 23 esetén „23” kerül a sorosított karakterláncba), speciális határoló karaktereket használva az elválasztáshoz. A mért értékekből is látható, hogy ez a sorosító körülbelül a DataContractSerializer teljesítményét tudja nyújtani – időben legalábbis mindenképp:

Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	655	0,444
10	656	5,059
50	666	25,284
100	657	50,562
500	672	252,762
1000	713	505,563
2000	799	1 011,130
5000	962	2 527,697
10000	1269	5 055,379
15000	1690	7 583,148
20000	2029	10 110,929
25000	2345	12 638,543
50000	3957	25 277,364

3. táblázat - A FixedStringSerializer eredményei

A pontos teljesítménymutatók: átviteli idő: 0,066 ms/db valamint a kimenet mérete: 0,505 kybte/db (megint csak elmondható, hogy időre jól, hosszra pedig tökéletesen alkalmazhatóak a teljesítménymutatók). A DataContractSerializer esetében az átviteli időre 0,068 ms/db, a kimenet méretére 0,963 kbyte/db értékeket kaptam. Ez tehát azt jelenti, hogy az sorosított adatfolyam hosszát sikerült majdnem felére zsugorítani, mégis csak éppen, hogy gyorsabb módszer. A módszer lassúságát a

karakterláncok kezelése okozza, hiszen ez még StringBuilder használata esetén sem túl gyors. Így ez a feldolgozási stratégia nem megfelelően gyors referenciaértéknek.

Bár a tömörítés értéke már közelít a jóhoz, egyrészt a protokoll nehéz kezelhetősége és bővíthetősége, másrészt a fölösleges elválasztó karakterek miatt úgy döntöttem, hogy a kimeneti formátum nem lehet karakterlánc-alapú – át kell térni tehát valamilyen bináris formátumra.

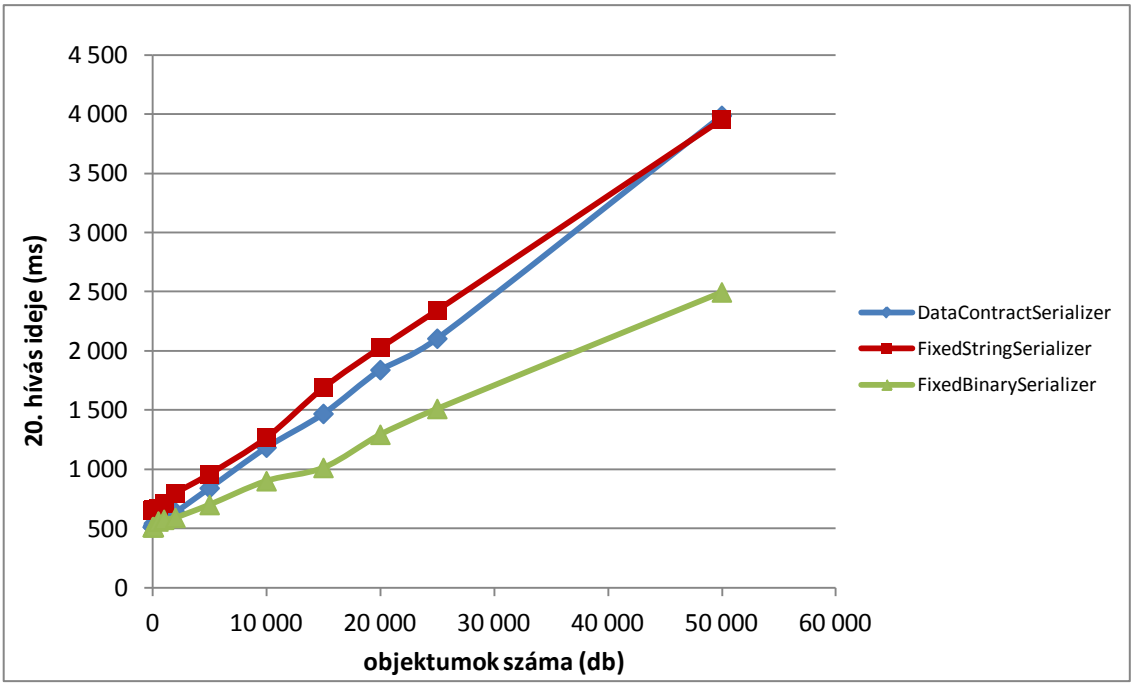
2.4.2.2 FixedBinarySerializer

A karakterlánc-alapú megoldás hibáinak kiküszöbölésére egy bináris sorosítót dolgoztam ki – ez az adott példatípus mezőit és tulajdonságait adott sorrendben binárisan egy folyamba írja, majd innen olvassa vissza. Ehhez a BinaryReader és a BinaryWriter osztályokat használtam. Most már egyrészt minden adat csak annyi helyet foglal a sorosított adatfolyamban, amennyivel leírható, másrészt az egyetlen járulékos adat a gyűjteményobjektumok hossza; hogy annyira azért ne legyen kötött a megoldás, hogy ezt meghagyjuk, hiszen csupán 4 bájt méretű adatról van szó.

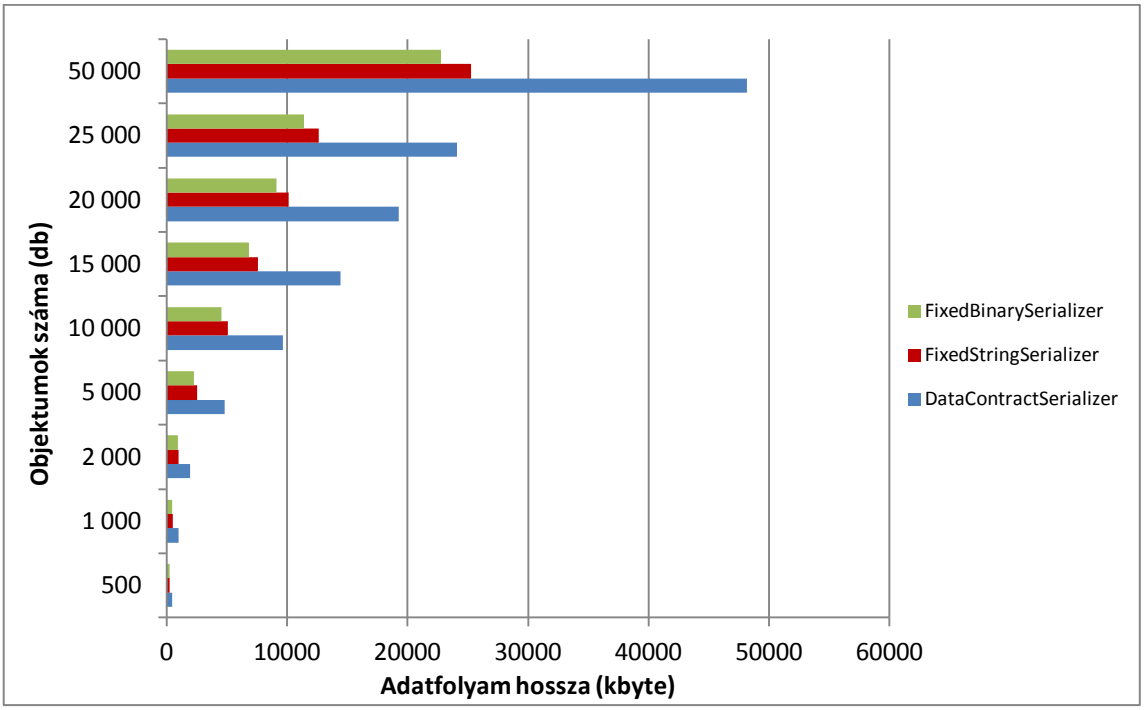
Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	514	0,397
10	514	4,555
50	515	22,758
100	514	45,512
500	562	227,543
1000	576	455,082
2000	592	910,160
5000	701	2 275,395
10000	902	4 550,785
15000	1014	6 826,176
20000	1294	9 101,566
25000	1512	11 376,957
50000	2496	22 753,910

4. táblázat - A FixedBinarySerializer eredményei

A FixedStringSerializernél jobb értékeket mutatnak a mérések – továbbá a DataContractSerializernél is gyorsabb és tömörebb a módszer. A fentiekkel összehasonlítva ezt a módszert a görbék:



7. ábra – FixedBinarySerializer, DataContractSerializer és FixedStringSerializer átviteli idejei



8. ábra – FixedBinarySerializer, DataContractSerializer és FixedStringSerializer kimeneteinek mérete

A kiszámolt teljesítménymutatók: átviteli idő: 0,039 ms/db és a kimenet mérete: 0,455 kbyte/db (lévén, hogy az időfüggvény közel teljesen, az adathossz függvény pedig teljesen lineáris). Ennél a módszernél mindkét érték sokkal jobb, mint a DataContractSerializer esetében, így ezeket az értékeket már használhatóak referenciaként.

	DataContract Serializer	NetDataContract Serializer	FixedString Serializer	FixedBinary Serializer
átivteli idő (ms/db)	0,068	0,279	0,066	0,039
kimenet mérete (kbyte/db)	0,963	2,161	0,505	0,455

5. táblázat - Az eddigi sorosítók összehasonlítása

Sikerült tehát azt is bebizonyítanom, hogy valóban egy ilyen jellegű bináris protokoll a legtömörebb. Magáról a sorosítási algoritmusról még sokat nem tudunk, csupán egy határértékünk van a futási idejére (valamint sejtjük, hogy egyáltalán létezik).

Természetesen – ahogy fentebb említettem – ez csak egyfajta határérték; még egy nagyon jó módszerrel is csak megközelíteni lehet ezt a módszert. Továbbá ha szükséges, akkor a sorosított adatfolyam hossza lehet akár hosszabb is, ha ezzel gyorsítani lehet a módszeren.

2.4.3 Általános megoldások kidolgozása

Miután meggyőződtem róla, hogy lehet gyorsabb megoldást használni a beépített megoldásoknál és meghatároztuk a referenciaértékeket, amiket el kell érni, nekiláttam az általános megoldások kidolgozásának.

2.4.3.1 BinarySerializer

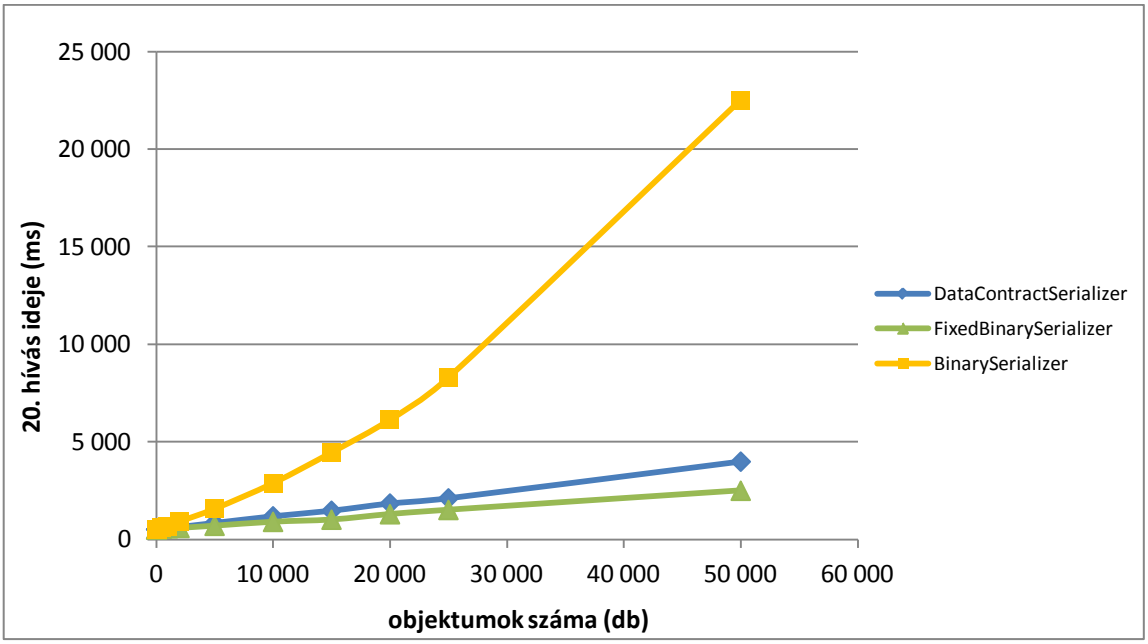
Első megoldás az általánosítás felé egy olyan sorosítómotor kidolgozása volt, ami a sorosítás feladatát a BinaryFormatter osztály egy példányához delegálja. A BinaryFormatter a 2.4.1 fejezetben felsoroltak közül elég sokat kielégít – legfontosabb, hogy a példaosztályunkból valóban kimenti az ott specifikált tagokat [16].

A megoldás implementációja természetesen nagyon egyszerű, és minden típust gond nélkül le tud kezelni. Egész egyszerűen csak kiírom egy adatfolyamra az objektumot egy BinaryFormatter példánnyal, majd ezt az adatfolyamot elhelyezem egy XML-tagban az üzenetben. Viszont ennek a kézenfekvőségnek ára van, ahogy az a teljesítmények mérőszámain látszik is:

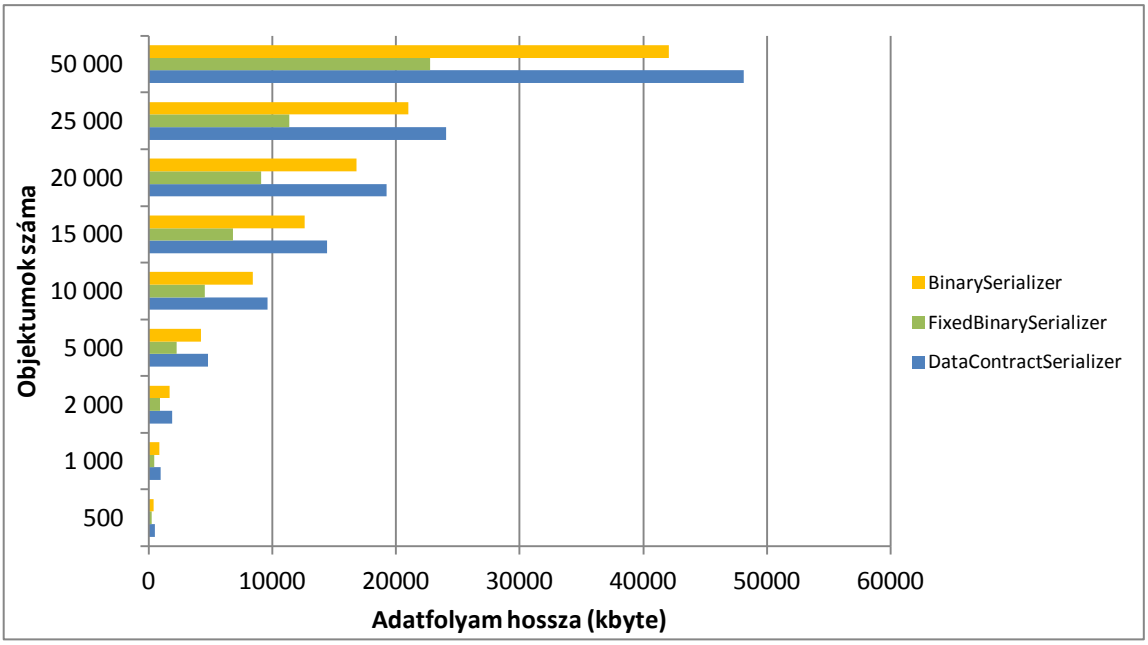
Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	514	2,959
10	530	10,629
50	530	44,262
100	530	86,303
500	608	422,631
1000	670	843,041
2000	919	1 683,861
5000	1575	4 206,322
10000	2888	8 410,424
15000	4460	12 614,525
20000	6141	16 818,627
25000	8299	21 022,729
50000	22518	42 043,236

6. táblázat - A BinarySerializer eredményei

Összehasonlítva a DataContractSerializer, valamint a referenciaként használt FixedBinarySerializer eredményeivel jól látszik, hogy ez a módszer még nem tudja a kívánt eredményeket produkálni:



9. ábra - BinarySerializer, DataContractSerializer és FixedBinarySerializer átviteli idejei



10. ábra - BinarySerializer, DataContractSerializer és FixedBinarySerializer kimeneteinek mérete

A hívásidőt tekintve messze nem éri el nemhogy a FixedBinarySerializer, de még a DataContractSerializer idejét sem. Bár az output mérete valamivel kisebb, összességében azonban ez a módszer nem használható elég hatékonyan. A számított teljesítménymutatók (az idő görbe nagyon jó közelítéssel lineáris, az adathossz görbe pedig ebben az esetben is teljesen) és összehasonlításuk a fenti két módszer mutatóival:

	DataContractSerializer	FixedBinarySerializer	BinarySerializer
átviteli idő (ms/db)	0,068	0,039	0,409
kimenet mérete (kbyte/db)	0,963	0,455	0,84

7. táblázat – a DataContractSerializer, a FixedBinarySerialize és a BinarySerializer teljesítménymutatói

Ez a megoldás tehát messze nem kielégítő, tanulságként leszűrhető, hogy muszáj saját típusfeltérképezést kidolgozni és sorosítási protokollt definiálni.

2.4.3.2 Az általános sorosítás protokollja; ReflectionSerializer

A saját sorosítási módszer kidolgozásának az algoritmus leírása mellett a kimeneti formátum definiálása is fontos része. A cél adott volt: egy közel olyan tömörségű protokollt kellett kidolgozni, mint amilyet a FixedBinarySerializer produkált, de úgy, hogy alkalmazható legyen tetszőleges típusra.

Mivel a munkának ebben a fázisában nem a sebességnövelés volt a cél, hanem kifejezetten a protokoll megalkotása, így maga az itt megalkotott sorosítómotor csupán prototípus. Mivel a sebesség nem volt szempont, ezért – hogy a kimeneti formátumra és az algoritmus vázára jobban tudjak koncentrálni – ebben a fázisban a System.Reflection névtér osztályaival és azok metódusaival dolgoztam. Ezek a hívások természetesen nagyon lassúak – lévén, hogy karakterláncokkal azonosítanak mindent illetve minden be- és kimeneti érték object típusban kerül kezelésre –, de az itt kidolgozott protokoll alkalmazása egy gyorsabb módszerrel már meghozhatja a várt eredményt.

Kidolgoztam egy adatstruktúrát, mely az általánosság elvesztése nélkül rendkívül tömören tud .NET-es objektumgráfokat leírni. Extended Backus-Naur formában [17] a következőképpen írható le a kimeneti adatfolyam általam definiált struktúrája:

```
<serialized_stream>::=<types><serialized_graph>
<types>::=<typecount>*<type_name>
<serialized_graph>::=<serialized_object>|{<serialized_object>}
<serialized_object::=<serialized_primitive>|<serialized_dict>|<serialized_list>|
                        <serialized_complex>
<serialized_primitive>::=<type_id><object_value>
<serialized_dict>::=<type_id><dictlength>*<serialized_keyvaluepair>
<serialized_keyvaluepair>::=<serialized_key><seralized_value>
<serialized_key>::=<serialized_object>
<serialized_value>::=<serialized_object>
<serialized_list>::=<type_id><listlength>*<serialized_object>
<serialized_complex>::=<type_id>{<serialized_property>}{<serialized_field>}
<serialized_property>::=<serialized_object>
<serialized_field>::=<serialized_object>
```

Az adatfolyam elején egy típuslista van (types), ami az objektumgráfban előforduló típusok neveit tartalmazza (természetesen mindegyiket csak egyszer). A lista elejére odakerül, hogy hány típusról is van szó (typecount), majd ezután minden, az objektumgráfban szereplő típus teljes neve (AssemblyQualifiedName), hiszen ez kell ahhoz, hogy magát a típust tényleg azonosítani tudjuk (type_name).

Teljesítmény-megfontolásból ennek a listának az első 15 elemét automatikusan feltöltöm az atomi típusok neveivel, hiszen ezeket gyakran használjuk egy éles

alkalmazás fejlesztésekor objektumainkban [18]. Ezeket így át sem kell küldeni a hálózaton, csupán a sorrendjüket kell rögzíteni; a másik oldalon ugyanúgy tárolható fixen ebben a listában (a lista tartalmát lásd C függelék: Atomi típusok című részben).

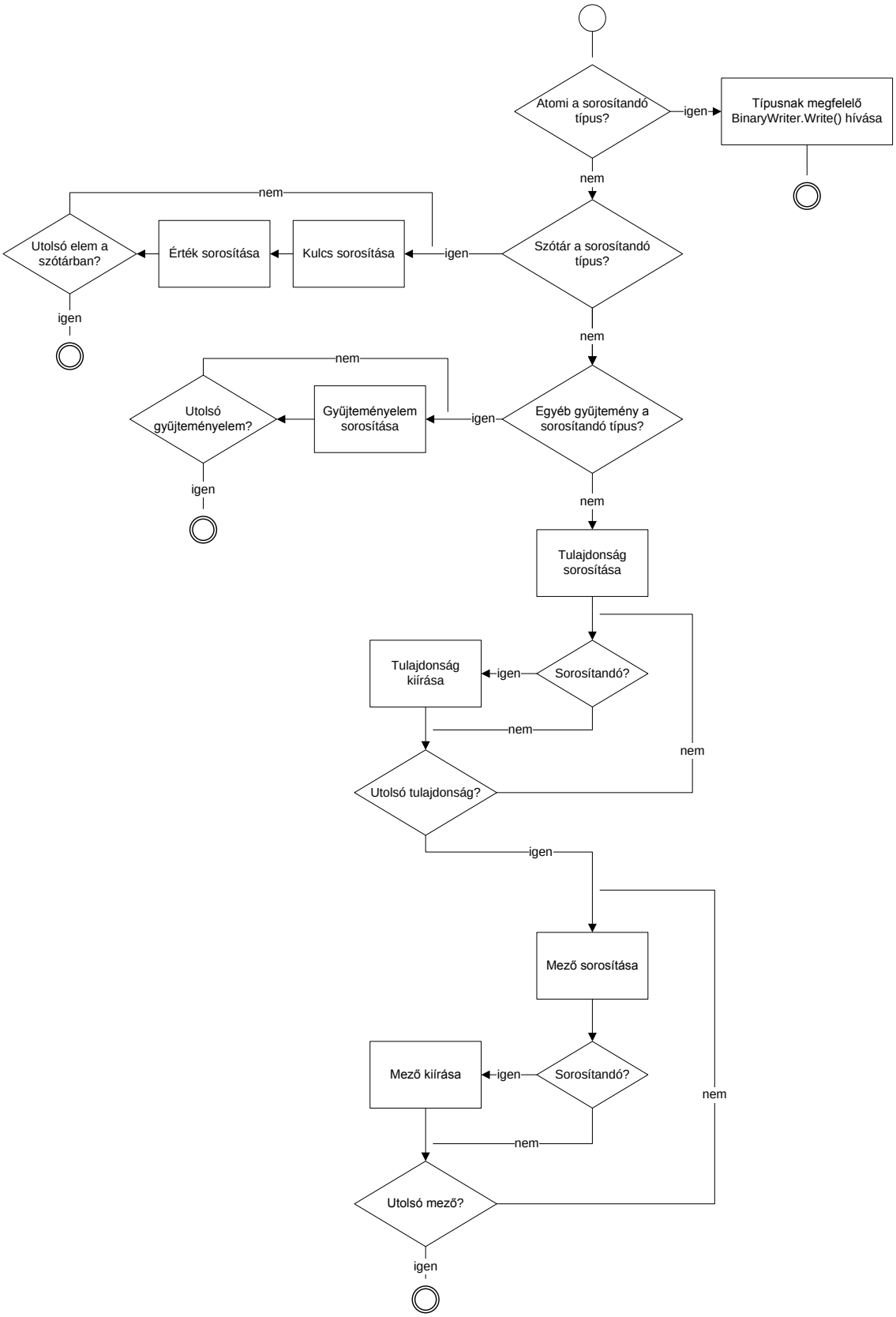
Ebben tehát (az enumerációkon kívül) benne van az összes primitív típus; az első, ezektől eltérő típus a listában a 16. helyre kerülhet.

Az adatfolyam második darabja maga a sorosított objektumgráf (serialized_graph); a protokollnak ez a része adja az algoritmusunk vázát is. Elsőként a teljes gráf kerül feldolgozásra; tegyük fel, hogy ebben van legalább egy objektum. Ez az objektum lehet primitív típusú, szótártípusú, egyéb gyűjtemény, vagy valamilyen egyéb komplex típus – ezek a típushalmazok diszjunktak, tehát a protokollban ezek szétválasztása jogos.

Ha ez az objektum primitív típusú, akkor kiírom a folyamra a típusazonosítót (type_id) (ez a típusnév indexe a típusok listájában), majd az értékét is (serialized_primitive). Ezzel készen van a sorosítás.

Ha nem primitív típus, akkor – ahogy a 2.4.1 részben írtam – nem sorosítom külön objektumként, hanem részekre bontom. Mindenképpen kiírom a típusindexét, ha szótár vagy egyéb gyűjtemény, akkor a hosszát is (dictlength ill. listlength). Ezek után, ha szótárról van szó, akkor a kulcs-érték párokat sorosítom (serialized_keyvaluepair), de nem önálló objektumként, hanem ezt is darabonként – azaz külön sorosítom a kulcs objektumot (serialized_key) és az érték objektumot (serialized_value). Egyéb gyűjtemény esetén a gyűjtemény elemeit sorosítom.

Ha egyéb komplex típusról van szó, akkor kiírom a típusazonosítót, majd a 2.4.1 részben meghatározott feltételeknek eleget tevő tagok kerülnek sorosításra, mint önálló objektumok. Ahogyan megállapodtunk, először a tulajdonságok, azután a mezők, ezek közül is csak azok, amik publikusak és publikusan elérhetőek. A szintaxis ezt nem tudja ugyan jelölni, de fontos, hogy a tulajdonságok és a mezők is névsorrendben vannak. Az alábbi folyamatábrán egyetlen objektum sorosítása követhető:



11. ábra - Egyetlen objektum sorosítása

Visszafelé hasonlóan működik az algoritmus – a folyamról beolvasom a típusokat, majd kezdődik a feldolgozás. Beolvasom a típusindexet, megnézem a típuslistában, hogy ez milyen típus volt. Ha primitív típus, akkor beolvasom a megfelelő hosszúságú értéket és ezzel kész a visszaolvasás. Ha szótár, akkor beolvasom a következő elemet, ami a hosszt mutatja, majd beolvasok ennyi kulcs objektumot és érték objektumot. Listánál hasonlóan, ott természetesen csak érték objektum van. Ha komplex típusról van szó, akkor pedig végiglépkedek névsorrendben először a sorosítható tulajdonságokon, aztán a mezőkön, minden egyes alkalommal beolvasva egy objektumot (referenciaként a sorosítás kiírási irányának vázlatos algoritmusát lásd E függelék: A sorosítás vázlatos algoritmusa kódban című részben).

Lássuk, hogy milyen eredményeket tud produkálni ez a megoldás! Elsősorban a kimenet mérete a fontos, de a teljesség kedvéért megmértük a sebességet is:

Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	530	0,932
10	561	5,385
50	635	24,587
100	702	48,591
500	1373	240,617
1000	2231	480,650
2000	3931	960,717
5000	8938	2 400,916
10000	17629	4 801,248
15000	26083	7 201,581
20000	34376	9 601,913
25000	44609	12 002,245
50000	86674	24 003,906

8. táblázat - A ReflectionSerializer eredményei

A számított teljesítménymutatók: átviteli idő: 1,726 ms/db valamint a kimenet mérete: 0,48 kbyte/db.

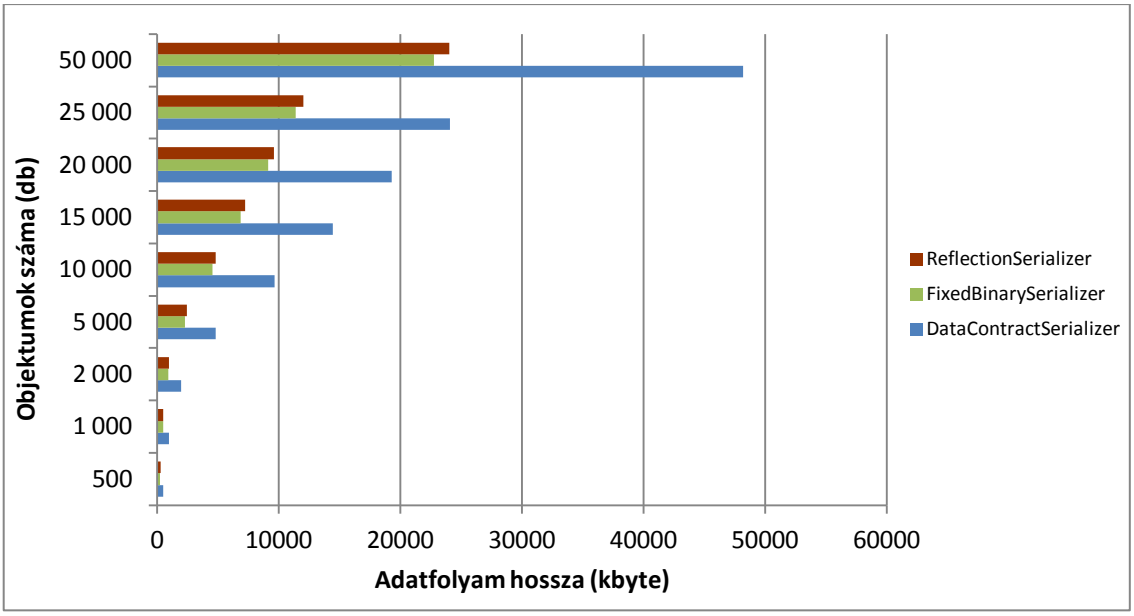
Az átviteli idő értékét sokáig nem érdemes vizsgálni, ahogyan vártuk, eddig ez a legmagasabb érték, ez pedig jórészt a System.Reflection névtér osztályainak használata miatt van így. Egy metódus hívása közvetlenül (a Call vagy a Callvirt operációval) nagyságrendekkel gyorsabb, mint egy Invoke() hívás az őt leíró MethodInfo objektumon [19]. Valamennyit ugyan tudnánk javítani a módszeren, ha a MethodInfot egy delegate-hez kötjük, és a delegate kerül meghívásra, de messze nem megfelelő mértékű a nyereség, így mindenképpen egy másik módszert kell majd kidolgozni.

A kimenet mérete viszont ígéretesnek látszik; emlékeztetőül és összehasonlításképpen a DataContractSerializer és a referenciaként használt FixedBinarySerializer értékei:

	DataContractSerializer	FixedBinarySerializer	ReflectionSerializer
átviteli idő (ms/db)	0,068	0,0392	1,726
kimenet mérete (kbyte/db)	0,963	0,455	0,48

9. táblázat - a DataContractSerializer, a FixedBinarySerializer és a ReflectionSerializer teljesítménymutatói

Látható, hogy a DataContractSerializernél jóval rövidebb az adatfolyam, sőt a FixedBinarySerializer értékét is egészen jól közelítjük (előbbinél kb. 53%-kal jobb az érték és az utóbbitól kb. 5,5%-kal marad csak el).



12. ábra – ReflectionSerializer, DataContractSerializer és FixedBinarySerializer kimeneteinek mérete

Ez a protokoll tehát megfelelő arra, hogy építsünk rá egy hatékonyabb komponenseket felhasználó sorosítómotort: ez lesz a dinamikus megoldás. Ebben a dinamikus megoldásban hasonló végrehajtási útvonalak lesznek, de a későbbiekben apróbb módosításokat végrehajtunk a sebesség növelésének érdekében (lásd a 2.4.4.3 fejezetben). A protokollon szintén módosítunk majd egy keveset a hatékonyabban kommunikáció érdekében (lásd a 2.4.4.2 fejezetben).

2.4.4 Az általános és gyors megoldás: DynamicSerializer

A munkámnak erre a fázisára tehát adottak voltak a referenciaértékek, amiket el kellett érni, adott volt a protokoll, ami megfelelőnek tűnt a sorosítás hatékony kivitelezéséhez és megvolt az algoritmus, ami minden esetet le tud kezelni. Egyedül az algoritmus implementációja során felhasznált technológiai apparátus (a System.Reflection névtér elemei) volt akadálya a gyors verzió megvalósításának – erre a feladatra tehát másik technológiai eszköztárat kellett választani. Ez az eszköztár az 1.1.3 részben is bemutatott Reflection.Emit kódgenerálás-modell; az így megvalósított sorosító pedig – mivel futási időben állítja elő tetszőleges típushoz a sorosító kódot – a dinamikus sorosító (DynamicSerializer).

2.4.4.1 Az első verzió

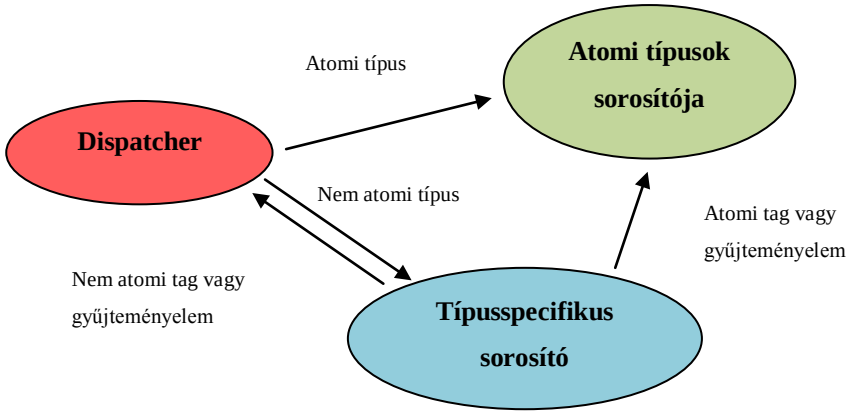
Az első megoldás lényegében adott az eddigi kódokból. A szerelvénygeneráláshoz majdnem egy az egyben a bevezető részben bemutatott kód megfelelő részét használjuk, annyi különbséggel, hogy nem mentem ki fájlba a szerelvényt, csupán „megsütöm” a sorosítótípust, és egy olyan szótárban tárolom, aminek kulcsa a sorosított objektumgráf gyökérelemének típusa (ez a megoldás nem tökéletes, de a méréseink számára kielégítő, lásd a 2.5.4 részt).

Az algoritmus pedig nagyjából adott a RelfectionSerializerből, működése majdnem teljesen azonos az ott bemutatott megoldáséval.

Technológiai kivételként érdemes megjegyezni, hogy a tömbök sorosítása külön lett választva teljesen a generikus gyűjteményekétől, mert ezek kezelése speciális műveleteket igényel. Másik fontos különbség, hogy a ReflectionSerializer mindenhol object típusú elemekkel dolgozik, így a fordító a generált IL kódba beleteszi a megfelelő helyekre a dobozlásra vonatkozó utasításokat, ha kell. A DynamicSerializer esetében azonban magam generálom a kódot, így nekem kellett eldönteni, hova kell és hova nem dobozolás, azaz praktikusan szét kell választani a referencia és értéktípusok esetleges object típusúra történő változtatását; előbbi esetben Castclass, utóbbi esetben Box_Any illetve Unbox_Any műveletek kerülnek végrehajtásra. A megkülönböztetés a tervezési időben deklarált típus alapján (ebből adódhatnak problémák, lásd a 2.5.4 részt) történik.

Technikai különbség a sorosítás mindkét irányában egy olyan metódus bevezetése, ami eldönti, hogy a sorosítandó típus atomi típus-e vagy sem. Ha igen, akkor meghívja az atomi típusok sorosítására generált metódust, ha nem, akkor pedig

attól függően, hogy milyen típusú a sorosítandó objektum, meghívja a típusra generált sorosítót. Ezt a dispatcher-jellegű metódust kell még legenerálni. Az atomi típusok sorosítására használt metódus csak kényelmi szempontokból született meg, a generálása teljesen statikus, hiszen mindig ugyanazokat a típusokat és ugyanúgy sorosítja (később el is vetjük a külön metódus ötletét, lásd a 2.4.4.3 részt).



13. ábra - A sorosításban részt vevő, generált metódusok

A ReflectionSerializer algoritmusa adja azonban azt az alapot, amivel a típusspecifikus sorosítókat generáljuk, csak a legtöbb esetben nem a System.Reflection névtérben található metódusokat hívom, hanem belegenerálom a készülő kódba a hívást (azaz például nem egy Invoke() hívás történik egy MethodInfo objektumon, hanem egy ILGenerator.EmitCall() hívás a MethodInfo objektummal, mint paraméter). Például ugyanúgy szét vannak választva a szótárak, egyéb gyűjtemények és egyéb komplex objektumok, vagy ugyanúgy végig kell lépkedni a gyűjteményeken illetve a komplex objektumok tagjain és így tovább.

Fontos volt, hogy nagyjából olyan kódot generáljak, mint a referenciaként használt FixedBinarySerializer kódja, hiszen azt már lefektettük, hogy annál gyorsabbat nem valószínű, hogy létre lehet hozni és azt érdemes használni, mint megközelítendő határérték.

2.4.4.2 A protokoll módosítása

A hatékonyság növelésének érdekében a protokollt is módosítottam egy kicsit. Az értéktípusok esetén nem léphet fel öröklés, az atomi típusaink pedig mind értéktípusok. Mivel a típusok jelölésére viszont pont az öröklés helyes kezelése miatt van szükség, ezért ebben az esetben nem kell explicit módon jelölni a típust – ahol a kódunkban tervezési időben valahol atomi típust használtunk (akár generikus

típusargumentum, akár tulajdonság vagy mező esetén), ott biztosak lehetünk benne, hogy futási időben is csak és kizárólag olyan típus állhat.

2.4.4.3 A sorosító motor módosításai

Bár az algoritmus nem változott, sok módosítást végeztem az objektumgráf teljes feldolgozása során, amelyek mind-mind nagyban hozzájárultak a hatékonyság növeléséhez:

- A gyűjtemény objektumok mérete benne van a sorosított adatfolyamban, így ezeket kezdőmérettel inicializálom. Ez a gyűjtemények belső, tömbalapú reprezentációja miatt fontos [20].
- A gyűjtemények bejárása típusos enumerátorokkal történik. Ezzel elkerülhető az esetleges bedobozolás és kidobozolás illetve típuskényszerítés, valamint az ezekből adódó többletterhelés.
- Azokat a tulajdonságokat és mezőket, amelyek atomi típusúak, nem a sorosító metódusokon keresztül írom bele a sorosított adatfolyamba vagy olvasom ki onnan őket, hanem közvetlenül a BinaryReader és BinaryWriter megfelelő metódusait hívom. Lényegében a .NET fordító által is alkalmazott inline kód optimalizációt hajtottam végre.
- Ugyanezt az optimalizációt megtetettem a gyűjteményeknél is, ha a típusargumentumok atomi típusúak.
- Inline lett az a dispatcher jellegű metódus is, ami eldönti, hogy melyik, nem atomi típusra írt sorosítót kell esetleg meghívni. Érdemes megjegyezni, hogy a generált kód önmagában meglehetősen rossz gyakorlatot mutat, hiszen a sok inline rész miatt egy-egy kódrészlet többször is előfordul. Mivel azonban ezek a kódok generáltak, így azt a kódot, ami generál csak néhány helyen kell módosítani, hogy a generált kódban több helyen is változtassunk.
- Mivel az atomi típusok elhelyezkedése a típuslistában fix, így nem csak átküldeni nem kell őket, lényegében bele sem kell őket írni ebbe a listába – ezt el is hagytam.
- A jelenlegi verzióban a visszaolvasáskor sehol nem a típusleíró Type objektumra, hanem a neki megfelelő type_id-re vonatkozóan teszek feltételeket, amik vezérlik a generált kódot.

Ezeket figyelembe véve például a kódrészlet, ami atomi generikus típusargumentumú gyűjteményeket tud sorosítani, a következőképpen néz ki:

```
//...
//A kiíró ciklus kezdetének megjelölése
gen.MarkLabel(beginwhile);
//Ha atomi típusok alkotják a gyűjteményt...
if (IsAtomic(componenttype))
{
  //...betöltjük a metódus első argumentumát; ez a BinaryWriter egy
  //példánya
  gen.Emit(OpCodes.Ldarg_1);
  //Betöltjük az előzőleg már lokális változóba eltárolt, típusos
  //enumerátort
  gen.Emit(OpCodes.Ldloc, enumerator);
  //Lekérjük az akutálisan mutatott elemet
  gen.Emit(OpCodes.Callvirt, enumeratortype.GetProperty("Current",
                                          BindingFlags.Public |
                                          BindingFlags.Instance)
                                          .GetGetMethod());
  //DateTime típus esetén a ticks mező kerül kiírásra
  if (componenttype == typeof(DateTime))
  {//...}
  if (componenttype.IsEnum)
  {
    //Az enumerációk egészként kerülnek kiírásra
    gen.Emit(OpCodes.Callvirt, writermethods[typeof(int)]);
  }
  else
  {
    //A többi atomi típus a neki megfelelő BinaryWriter metódussal
    //kerül kiírásra
    gen.Emit(OpCodes.Callvirt, writermethods[componenttype]);
  }
}
else
{
  //Nem atomi típusú komponensek sorosítása
}
//While ciklus végének jelzése
gen.MarkLabel(endwhile);
//Újra betöltjük az enumerátort
gen.Emit(OpCodes.Ldloc, enumerator);
//Léptetjük az enumerátort
gen.Emit(OpCodes.Callvirt,
typeof(IEnumerator).GetMethod("MoveNext",  BindingFlags.Public |
                                          BindingFlags.Instance));
//Ha van még elem, akkor ugrunk a ciklus elejére
gen.Emit(OpCodes.Brtrue, beginwhile);
}
//...
```

A kódrészletben látható, hogy a generálás technológiai kulcsa az ILGenerator osztály (gen példány). Ennek Emit() metódusával illeszthetünk be tetszőleges IL utasítást (és az argumentumait) az éppen generált kódba – így akár mi magunk is elő tudunk állítani tetszőleges vezérlési szerkezeteket (mint ahogyan a példában szerepel a while ciklus). Érdemes megjegyezni, hogy a generáló kód és a generált kód végrehajtási útjai teljesen különbözőek. A fenti kódrészlet statikus, azaz akárhányszor lefut, mindig

lesz például egy if-else páros, amivel eldöntjük, hogy a sorosítandó komponensek atomi típusúak-e vagy sem. A generált kódban azonban már semmi erre utaló jelet nem lehet találni – atomi típusok esetén csak a rájuk vonatkozó rész kerül legenerálásra, mint ahogy egyéb komplex típusok esetén is.

Az így generált sorosító teljes (visszafejtett) kódja és a FixedBinarySerializer kódja ezek után a módosítások után már nagyon hasonló (az összehasonlítás megtalálható az F függelék: A FixedBinarySerializer és a generált kód című részben); az sorosítómotor pedig ezek után a módosítások után kész arra, hogy teszteknek vessük alá.

2.4.4.4 Mérések a DynamicSerializer teljesítményére

A módosítások elvégzése után a sorosítómotor végleges állapotában a következő mérési eredményeket tudta produkálni:

Objektumok száma (db)	20. hívás ideje (ms)	Adatfolyam hossza (kbyte)
1	514	0,998
10	515	5,354
50	524	24,026
100	533	47,366
500	530	234,085
1000	577	467,483
2000	592	934,280
5000	717	2 334,671
10000	873	4 668,655
15000	1060	7 002,640
20000	1368	9 336,624
25000	1493	11 670,608
50000	2672	23 340,530

10. táblázat - A DynamicSerializer eredményei

Első ránézésre az eredmények biztatónak tűnnek. A számított teljesítménymutatók közül az átviteli idő értéke 0,042 ms/db, míg a kimenet méretének értéke 0,466 kbyte/db. Nézzük végig, hogy ezek a számok pontosan mit is jelentenek az előző megoldások teljesítményének tükrében – sikerült-e gyorsabb megoldást kidolgozni?

2.5 A saját megoldás eredményeinek értékelése

Most, hogy van egy megoldásunk, amiknek az eredményei biztatóak, érdemes megnézni, hogy valóban jobb-e ez, mint ami eddig rendelkezésünkre állt, és ha igen, akkor pontosan mennyit is nyerhetünk az alkalmazásával.

2.5.1 Összehasonlítás a FixedBinarySerializerrel

Hasonlítsuk össze először a DynamicSerializer teljesítménymutatóit a referenciaként használt FixedBinarySerializer értékeivel!

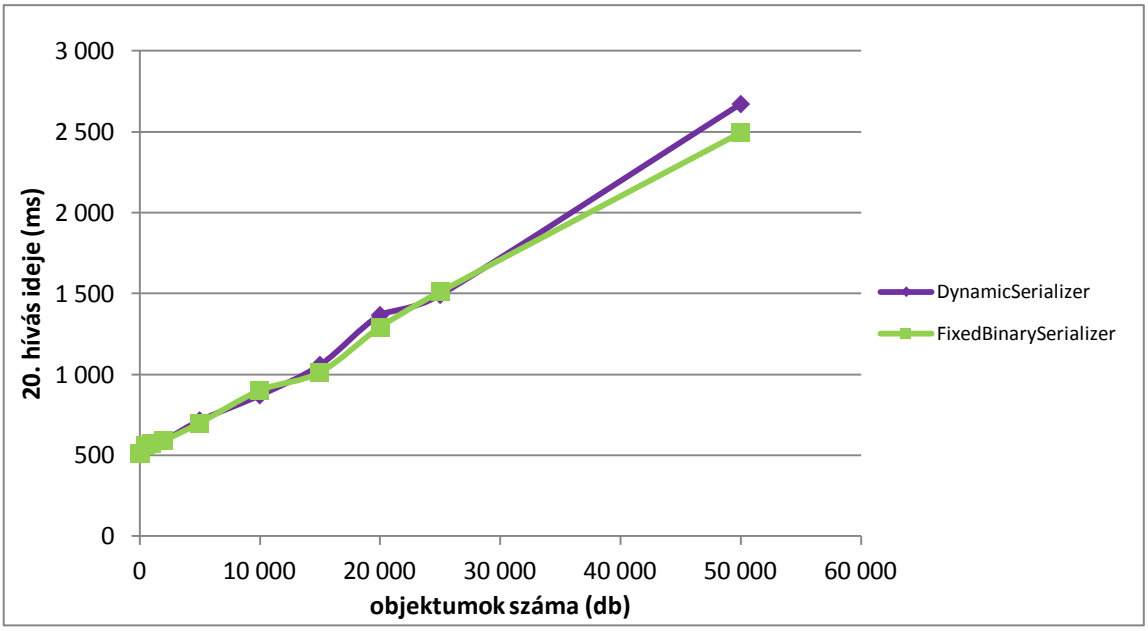
A FixedBinarySerializer értékei emlékeztetőül és mellette a DynamicSerializer értékei:

	FixedBinarySerializer	DynamicSerializer
átviteli idő (ms/db)	0,039	0,042
kimenet mérete (kbyte/db)	0,455	0,466

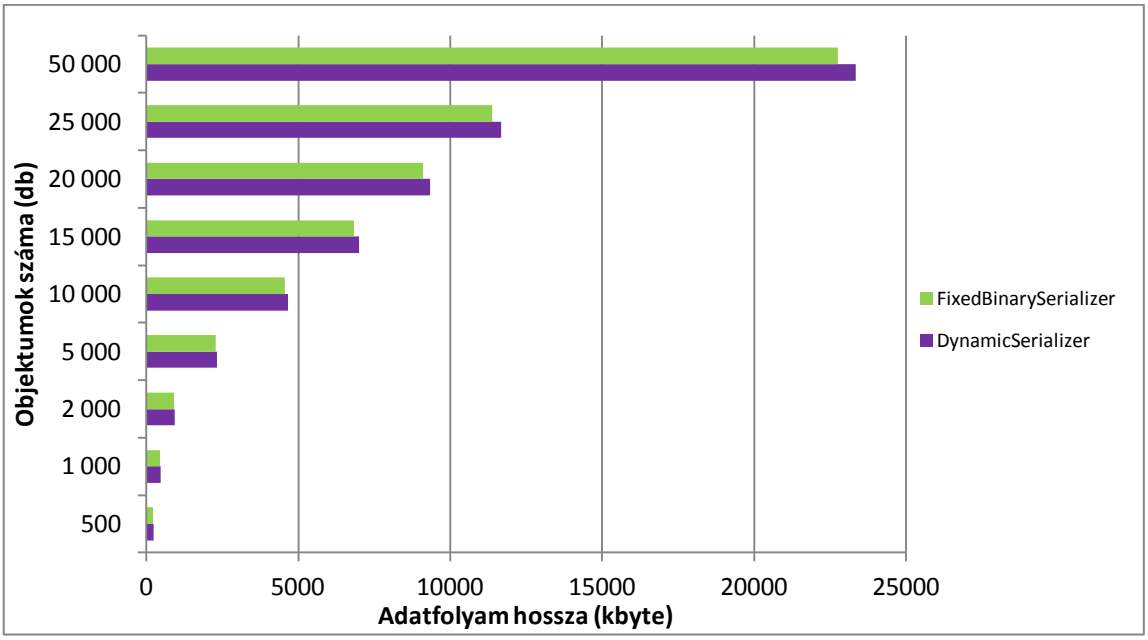
11. táblázat – a FixedBinarySerializer és a DynamicSerializer teljesítménymutatói

A kapott értékek alapján elmondható, hogy sikerült jól megközelíteni a referenciaként használt sorosítót. Az időt tekintve 7,78%-kal lassabb ez a módszer, az átvitt adat hossza pedig 2,57%-kal hosszabb. Ez utóbbi különbség – mindamellett, hogy elhanyagolhatóan kicsi – feltételezhetően már nem, vagy csak minimálisan csökkenthető, hiszen a típusok nevét csak egyszer küldtem át. Az időbeli különbség pedig elfogadható, tekintve hogy a DynamicSerializer egy általánosan használható megoldás (ez a különbség 50000 objektumnál 176 ms időt jelent mindössze).

A közelítés pontosságának szemléltetésére érdemes a görbéket is megnézni:



14. ábra - DynamicSerializer és FixedBinarySerializer átviteli idejei



15. ábra – a DynamicSerializer és a FixedBinarySerializer kimeneteinek mérete

Ez utóbbi ábra pedig magáért beszél – a kimenet mérete szinte egy az egyben a FixedBinarySerializerét adja.

Összességében tehát sikerült jól megközelíteni az előre meghatározott referenciaértékeket. Érdemes azonban megvizsgálni azt is, hogy mennyivel sikerült jobb megoldást előállítani az iparban már használtaknál.

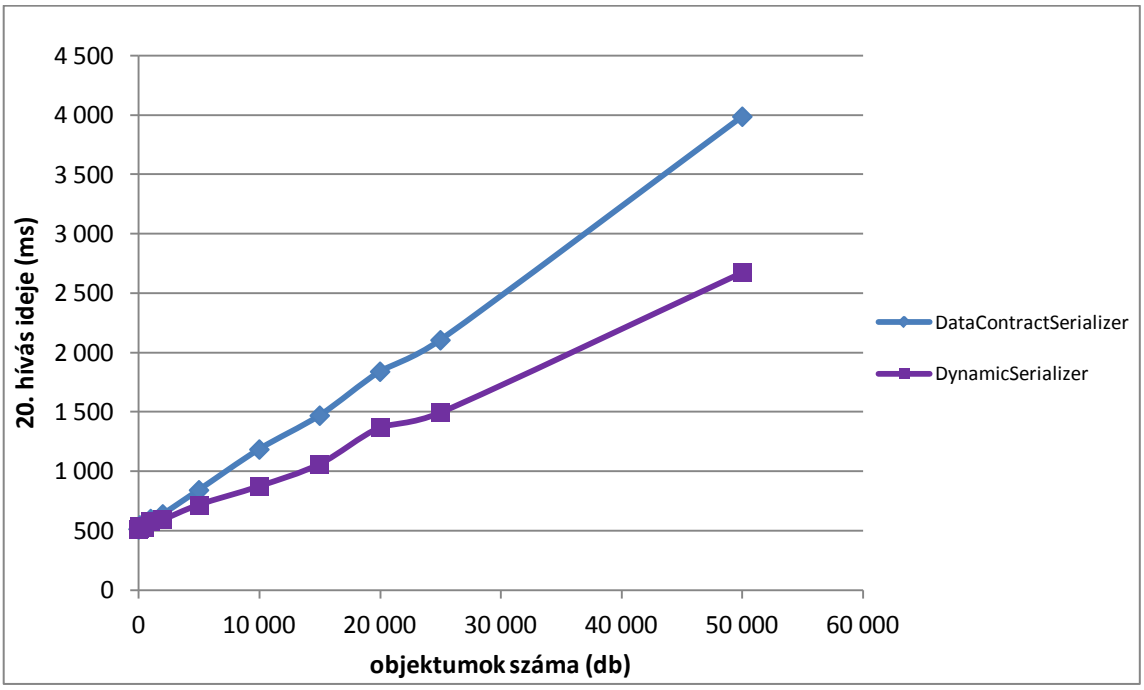
2.5.2 Összehasonlítás a DataContractSerializerrel

Nézzük meg, mennyivel teljesít jobban a DynamicSerializer a DataContractSerializernél! Mielőtt azonban az adatokat átnézzük, érdemes megjegyezni, hogy a DataContractSerializer előnyét, a platformfüggetlenséget a DynamicSerializer nem biztosítja – de ez nem is volt cél. A cél az volt, hogy .NET alkalmazások SOA kommunikációját megvizsgáljam és erre az esetre adjak egy gyors alternatívát. Ezek után nézzük meg az adatokat:

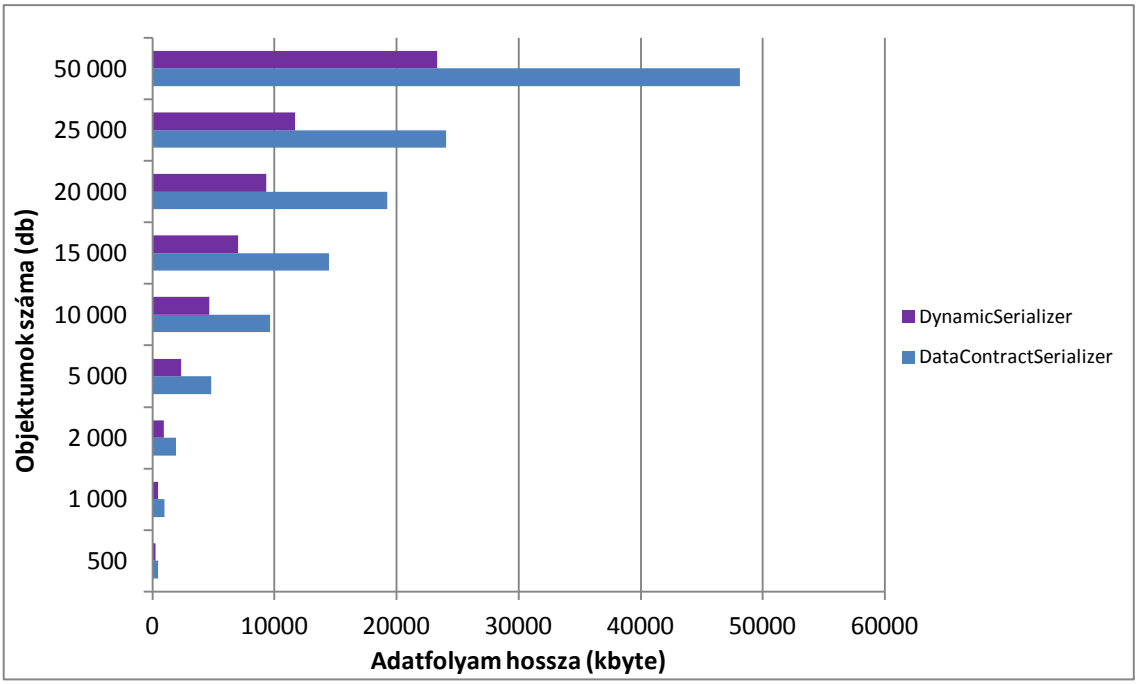
	DataContractSerializer	DynamicSerializer
átviteli idő (ms/db)	0,068	0,042
kimenet mérete (kbyte/db)	0,963	0,466

12. táblázat – a DataContractSerializer és a DynamicSerializer teljesítménymutatói

A teljesítménymutatók alapján sebességben 40%-kal, míg adathosszban 52%-kal hatékonyabb a DynamicSerializer. Bár az eredmények kicsit nehezen értelmezhetőek egyértelműen „jó”-nak, hiszen a platformfüggetlenség, mint előny ebben az esetben már nincs meg, de a ez nem is szerepelt a kezdeti célok között, az elért javulás pedig jelentősnek tekinthető, különösen sebességben.



16. ábra - DynamicSerializer és DataContractSerializer átviteli idejei



17. ábra – DynamicSerializer és DataContractSerializer kimeneteinek mérete

Az ábrákon a számszerűsített különbségek jól láthatóak.

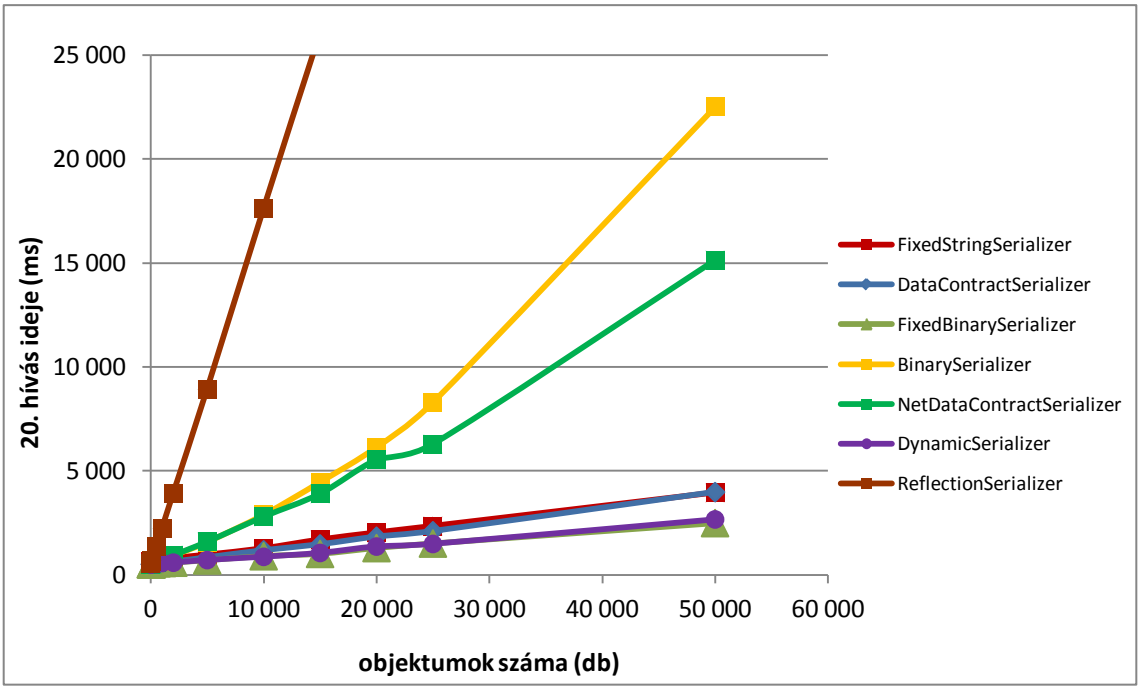
A sorosítók hasonló viselkedésmódját egyébként szépen demonstrálja az objektumszám-hívásidő görbe: mindkettőben körülbelül ugyanott találhatóak „töréspontok”, csupán más meredekségben – azaz nagyjából ugyanannyi objektumnál változik meg jelentősebben a sebességük.

2.5.3 Mindenki mindenki ellen

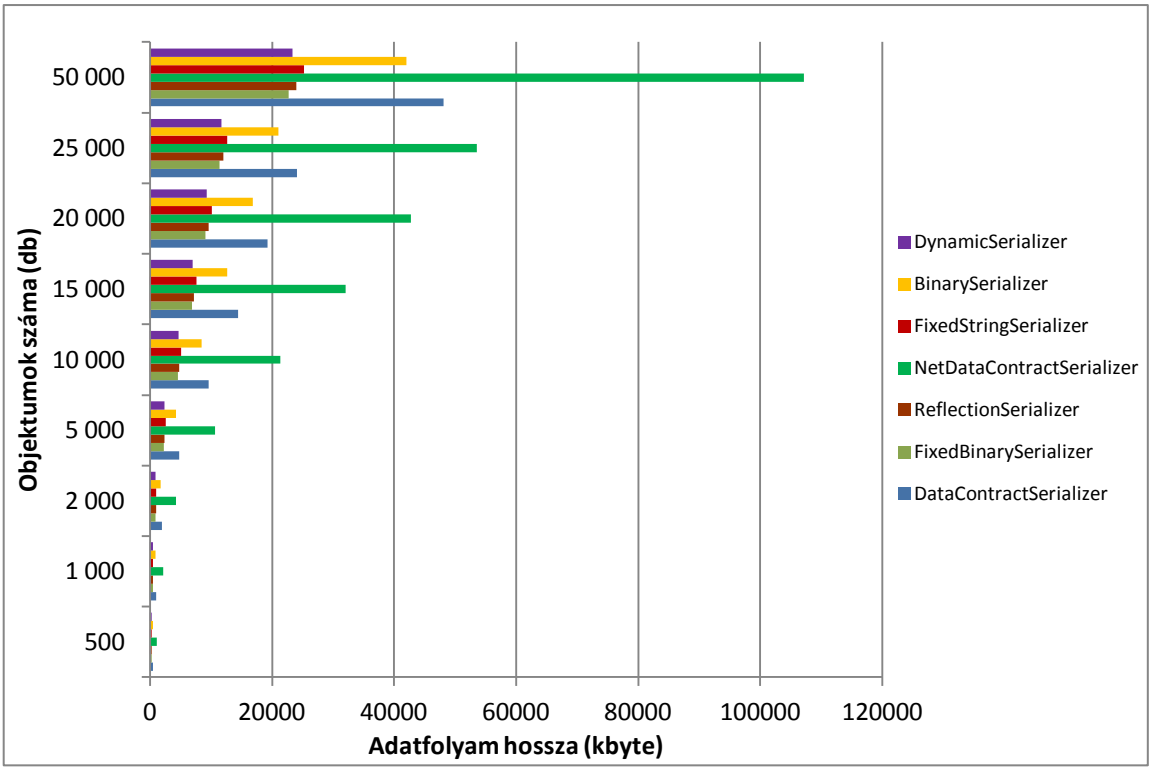
Utolsóként nézzük meg, hogy egymáshoz képest hogyan teljesítenek a sorosítók. Az ábrákról és a táblázatokból látszik a munka íve is. Elsőként megvizsgáltam a DataContractSerializert, majd a NetDataContractSerializert. Ezek a mérések nem adtak elég jó eredményeket, ezért nekiláttam gyorsabbat és tömörebbet keresni. Miután találtam egyet, általánosítottam ezt a megoldást. Az ábrák tanulsága az, hogy a dolgozat elején kitűzött célt sikerült elérni.

	DataContract Serializer	NetDataContract Serializer	FixedString Serializer	FixedBinary Serializer	Binary Serializer	Reflection Serializer	Dynamic Serializer
átviteli idő (ms/db)	0,068	0,279	0,066	0,039	0,409	1,726	0,042
kimenet mérete (kbyte/db)	0,963	2,161	0,505	0,455	0,84	0,480	0,466

13. táblázat - Mindenki mindenki ellen: teljesítménymutatók



18. ábra - Mindenki mindenki ellen; átviteli idők



19. ábra - Mindenki mindenki ellen; kimenetek mérete

Előzetesen tehát kijelenthetjük, hogy a dolgozat elején – valamint itt-ott közben és a végén is – leírt feltételek mellett (SOA architektúra, SOAP-alapú üzenetes kommunikáció) sikerült egy gyorsabb megoldást találni a sorosítás feladatára. A munka természetesen ezzel még azonban messze nem ér véget.

2.5.4 Fejlesztési lehetőségek

A jövőben az ipari felhasználhatóságot szem előtt tartva további módosítások, javítások elvégzése tervezett. Ezek közül a legfontosabb az egyetlen, valódi hibajelenség kijavítása. Ahogyan arra már kitértem, a protokoll és az algoritmus képes az öröklés megfelelő kezelésére, de – ahogyan már szintén megemlítettem – az alkalmazott módszer nem tudja helyesen lekezelni azt az esetet, hogy ha a sorosítandó elem tervezési időben object típusú, de futtatáskor valamilyen értéktípus az, ami valójában sorosításra kerül. Hasonló hibajelenség lép fel akkor is, ha nem object, hanem valamilyen interfészreferencia-típus a tervezési időbeli típus és valamilyen, ezt implementálandó struktúra kellene, hogy ténylegesen sorosításra kerüljön. Bár a „best practice" gyakorlatok gyakran felhívják a programozók figyelmét annak veszélyeire, ha egy értéktípust referenciatípus alól próbálnak kezelni [21], vannak helyzetek, mikor ez elkerülhetetlen, így ennek helyes kezelésére fel kell készíteni a sorosítót.

Ezt az egy hibát leszámítva azonban a módszer működőképesnek és gyorsnak bizonyult. A munka jelenlegi fázisában ennyi is volt a cél – egy gyors alternatívát adni a már oly sokszor emlegetett előfeltételek mellett a sorosításra. Ehhez természetesen méréseket kellett végezni, így néhány olyan funkció, ami a mérhetőséget és a hibakeresési hatékonyságot rontaná nem került még implementálásra. (Fontos megjegyezni, hogy ezek a módszerek a mért értékeket nem befolyásolhatják jelentős mértékben, ezért is hagyhatóak ki jogosan a munka jelenlegi fázisában.)

A legszembetűnőbb lehet, hogy az algoritmus (főként a javítások részben bemutatott változtatások egy része) csak azokon az ágakon van teljesen implementálva, amik a példaosztályunk esetében ténylegesen lefutnak, valamint a nem generikus gyűjtemények sorosítása egyelőre nem megoldott. Az előző hiányosság javítása evidens módon csak idő kérdése, míg utóbbi implementálása (a generikus verziók megléte miatt) nem nagy feladat.

Egy ehhez hasonló hiányosság, hogy az utólagos módosítások, amiket a protokollon végeztem (lásd a 2.4.4.2 részben), a jelenlegi formában nem teszik lehetővé egyetlen, atomi típusú elem sorosítását. Természetesen maga a módszer képes ezt az esetet is kezelni, csupán le kell kódolni.

Kevésbé feltűnő ugyan, de a típusfeltérképezés egyelőre nem dinamikusan történik, hanem a System.Reflection névtér alkalmazásával, emiatt egyelőre csak egyszer történik meg a mérések során. Erre viszont létezik nagyon egyszerű és gyors,

dinamikusan generált alternatíva – maga a generálás azonban kissé nehézkes. Mivel a mérési eredményeket szintén nem befolyásolhatja jelentős mértékben a típusfeltérképezés dinamikus verziója, így egyelőre ez nincs implementálva (kompenzációként a reflexiót használó verzió egyszeri lefutása benne van a mérésekben).

Szintén nem feltűnő, hogy egyelőre a generált sorosító azonosítása a sorosított gráf gyökérobjektumának típusával történik. Ez szintén csak a könnyebb hibakeresés miatt van így – a korrekt megoldás egy hash jellegű azonosító generálása, ami az összes, a gráfban lévő típustól függ.

Miután találtunk egy tömör módszert – és annak összes hibáját javítottuk – érdemes lehet további optimalizálási lehetőségeket megfontolni. Az egyik ilyen lehet az, hogy az atomi típusokra bevezetett protokollmódosítást (részletesen lásd a 2.4.4.2 fejezetben) bevezetjük minden értéktípusra és a zárt (sealed) osztályokra. Könnyen látható, hogy ezzel az algoritmust nem rontanánk el, hiszen a típusok jelölése csak az öröklés miatt fontos, értéktípusok között azonban ez nem jelenhet meg, a zárt osztályokból pedig nem lehet örökölni.

A szótár és egyéb gyűjtemény típusú elemek sorosítása a hibakeresés megkönnyítése miatt lett szétválasztva – maga az algoritmus és a protokoll azonban képes minden gyűjteményt (így egy szótárat is, ami valójában kulcs-érték párok gyűjteménye) egységesen kezelni. Az egységes implementáció a rövidebb kód és a könnyebb karbantarthatóság mellett valamennyi teljesítménynövekedéssel is járhat a generálás során.

Végül pedig, hogy a .NET-es, illetve WCF-es fejlesztési paradigmába beilleszthető legyen a DynamicSerializer, valamilyen módon meg kell oldani a könnyű alkalmazhatóságot (akár attribútumok, akár valamilyen egyszerű Behavior-konfigurálási lehetőség támogatásával [22]).

3 Összefoglalás

A dolgozatban a WCF sorosító komponenseit vizsgáltam sebesség és hatékonyság szempontjából. Bemutattam a fontosabb technológiai eszközöket, melyekre munkám során támaszkodtam. Ezután kidolgoztam egy megfelelő összehasonlító erővel rendelkező, de a gyakorlat szempontjából is egyszerűen és jól használható mérési metodikát, majd ezek alapján mérési eredményeket mutattam arra vonatkozólag, miért is gondolom úgy, hogy lehetne javítani ezeken a módszereken. Ezek után fokozatosan elkészítettem egy, az adott feltételek mellett (SOA architektúra, üzenetekkel kommunikáló .NET host és kliens) egy gyorsabb és tömörebb sorosítómotort.

Ennek első lépéseként megmutattam, hogy egyáltalán létezhet gyorsabb és tömörebb megoldás, még ha csak egyetlen esetre is. Ezt a megoldást referenciaként használva bemutattam az általános megoldás lépéseinek kidolgozását: egy hatékony protokoll definiálása, egy hatékony algoritmus kidolgozása és egy gyors eszközöket használó implementáció. Mindezek együtt alkotják a DynamicSerializer komponenst.

Végül megvizsgáltam ezt a komponenst, összehasonlítottam a már meglévő, beépített megoldásokkal illetve a referenciaértékekkel, és azt találtam, hogy valóban sikerült egy gyorsabb, de ugyanolyan általánosan működő sorosítót kidolgozni.

A feladat megfogalmazásakor nagy hangsúlyt kapott, hogy a munka végső eredménye egy olyan „termék” legyen, amit a későbbiek során ipari körülmények között is biztonsággal és hatékonyságot növelve lehet majd használni az adott feltételek mellett. Egyelőre még itt nem jár a projekt, de az első – és legnehezebb – lépések megtörténtek e cél felé: sikerült bebizonyítani, hogy a probléma létező és releváns, sikerült belátni, hogy van jobb módszer, sikerült ezt a jobb módszert kidolgozni és nagyon nagy részben implementálni, végül pedig ennek a jobb módszernek a hatékonyságát ellenőrző mérésekkel megmutatni. A következő lépések ennek fényében már kevésbé tűnnek nehéznek; legfőképpen idő és biztatás függvénye, hogy a még ránk váró javítások elkészüljenek.

A mérési eredmények mindenesetre ígéretesek. Bár egy nagy, elosztott szoftverrendszerben sok egyéb komponens is meghatározza az egész alkalmazás sebességét, mégis a bemutatott 40%-os sebességkülönbség önmagában is számottevő hatékonyságnövekedést jelenthet a sorosítót használó szolgáltatásokban.

Irodalomjegyzék

[1] S. Lamb and A. Oltean, "How Microsoft Uses Reflection," *MSDN Magazine*, 2004.

[2] Benedek Zoltán: Szoftvertechnikák előadás. Futtatókörnyezetek.

[3] MSDN Libarary: OpCodes Fields. (2011. 08. 31.) http://msdn.microsoft.com/en-us/library/812xyxy2.aspx

[4] (2010) Standard ECMA-335: Common Language Infrastructure.

[5] MSDN Library: OpCodes.Callvirt Field. (2011. 08. 31.) http://msdn.microsoft.com/enus/library/system.reflection.emit.opcodes.callvirt.aspx

[6] MSDN Library: Value Types and Reference Types. (2011. 10. 02.) http://msdn.microsoft.com/en-us/library/t63sy5hs%28v=vs.80%29.aspx

[7] MSDN Library: OpCodes.Swicth Field. (2011. 08. 31.) http://msdn.microsoft.com/enus/library/ system.reflection.emit.opcodes.switch%28v=vs.71%29.aspx

[8] N. Pathak, *Pro WCF 4: Practical Microsoft SOA Implementation, Second Edition*. Apress, 2011.

[9] A. Mackey, *A .NET 4.0 és a Visual Studio 2010*. SZAK Kiadó, 2010.

[10] P. Cibraro, K. Claeys, F. Cozzolino, and J. Grabner, *Professional WCF 4: Windows Communication Foundation with .NET 4*. Wiley Publishing Inc., 2010.

[11] MSDN Library: DataContractSerializer Class. (2011. 07. 28.) http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractserializer.aspx

[12] MSDN Library: KnownTypeAttribute Class. (2011. 08. 16.) http://msdn.microsoft.com/en-us/library/system.runtime.serialization.knowntypeattribute.aspx

[13] MSDN Library: NetDataContractSerializer Class. (2011. 08. 16.) http://msdn.microsoft.com/en-us/library/system.runtime.serialization.netdatacontractserializer.aspx

[14] MSDN Blogs: E. Osovetsky, Improving the start-up time of WCF clients. (2011. 09. 22.)

http://blogs.msdn.com/b/eugeneos/archive/2007/02/05/improving-the-start-up-time-of-wcf-clients.aspx

[15] RFC 3548: The Base16, Base32 and Base64 Data Encodings .

[16] MSDN Library: SerializableAttribute Class. (2011. 08. 16.) http://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx

[17] Extended Backus–Naur Form. (2011. 10. 02.) http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

[18] P. Kougiouris, "Use Reflection to Discover and Assess the Most Common Types in the .NET Framework," *MSDN Magazine*, 2002.

[19] J. Pobar, "Dodge Common Performance Pitfalls to Craft Speedy Applications," *MSDN Magazine*, 2005.

[20] H. Schildt, *C# 4.0: The Complete Reference*. McGraw-Hill Companies, 2010.

[21] MSDN Blogs: A. Basu, C#: structs and Interface. (2011. 10. 04.) http://blogs.msdn.com/b/abhinaba/archive/2005/10/05/477238.aspx

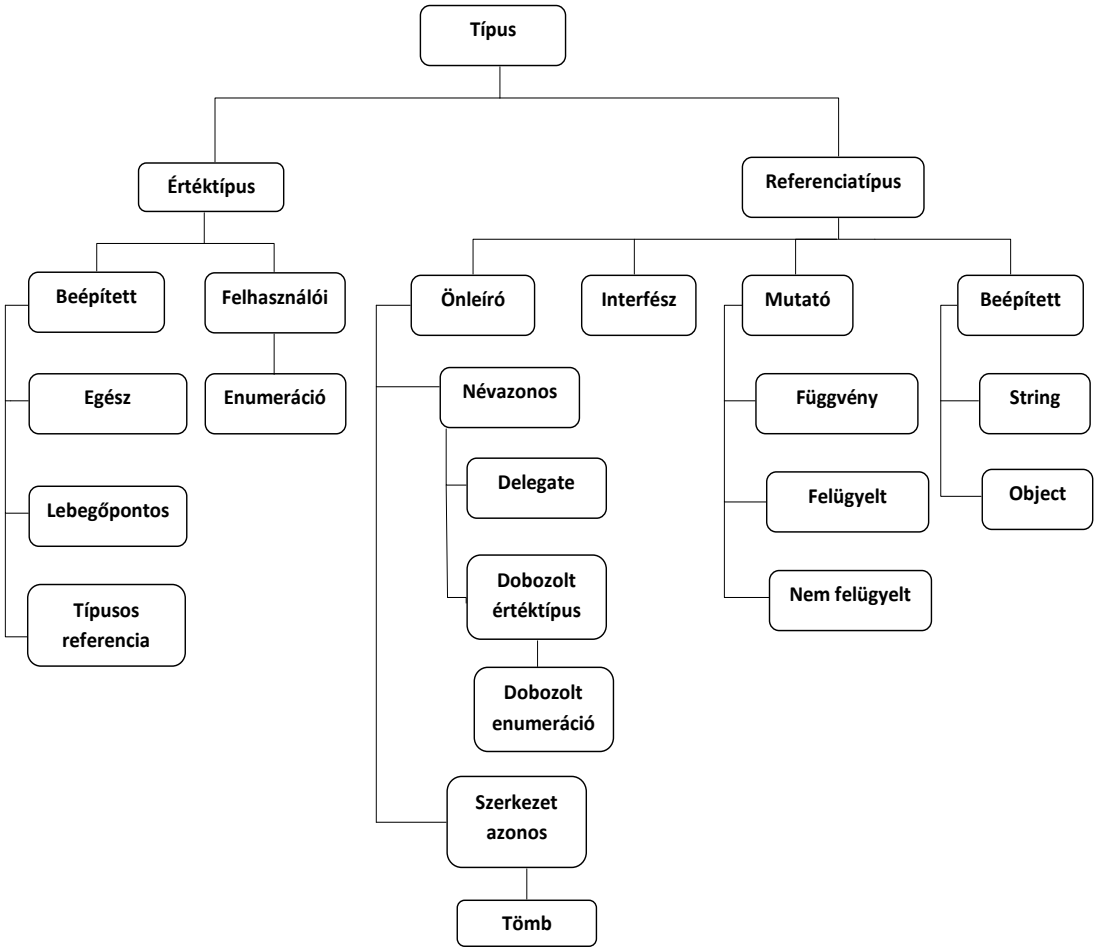
[22] MSDN Library: Configuring and Extending the Runtime with Behaviors. (2011. 08.28.) http://msdn.microsoft.com/en-us/library/ms730137.aspx

[23] J. Richter, *CLR via C#, Third Edition*. Microsoft Press, 2010.

[24] J. Albahari and B. Albahari, *C# 4.0 in a Nutshell, Fourth Edition*. O'Reilly Media, Inc., 2010.

A függelék: A CTS típusrendszere

Az alábbi ábra a Common Type System típuscsoportosítást szemlélteti [4]:



B függelék: A C# primitív típusai

A C# nyelv primitív típusainak listáját a következő táblázat tartalmazza [23]. A „primitív típus” oszlopban azt a nevet olvashatjuk, amivel a nyelv támogatja a könnyebb felhasználhatóságot (alias). A „BCL típus” oszlop mutatja meg, hogy a .NET osztálykönyvtár melyik típusára képződik le fordítás során a primitív típus. A „CLS” oszlopban azt jelezzük, hogy a primitív típus illeszkedik-e a szabvány ajánlásaihoz.

Primitív típus	BCL típus	CLS	Leírás
sbyte	**System.Sbyte**	✘	Előjeles, 8 bites érték
byte	**System.Byte**	✓	Előjel nélküli, 8 bites érték
short	**System.Int16**	✓	Előjeles, 16 bites érték
ushort	**System.UInt16**	✘	Előjel nélküli, 16 bites érték
int	**System.Int32**	✓	Előjeles, 32 bites érték
uint	**System.UInt32**	✘	Előjel nélküli, 32 bites érték
long	**System.Int64**	✓	Előjeles, 64 bites érték
ulong	**System.UInt64**	✘	Előjel nélküli, 64 bites érték
char	**System.Char**	✓	16 bites, Unicode karakter
float	**System.Single**	✓	IEEE 32 bites lebegőpontos érték
double	**System.Double**	✓	IEEE 64 bites lebegőpontos érték
bool	**System.Boolean**	✓	Logikai igaz/hamis (true/false)
decimal	**System.Decimal**	✓	128 bites, lebegőpontos érték
string	**System.String**	✓	Karaktertömb
object	**System.Object**	✓	Minden típus közös őse
dynamic	**System.Object**	✓	Speciális object

C függelék: Atomi típusok

A definiált atomi típusok és indexük a típustömbben:

0. System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
1. System.Int32, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
2. System.Int64, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
3. System.Int16, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
4. System.Boolean, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
5. System.Double, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
6. System.Byte, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
7. System.Char, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
8. System.Decimal, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
9. System.SByte, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
10. System.Single, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
11. System.UInt32, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
12. System.UInt64, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
13. System.UInt16, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
14. System.DateTime, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

D függelék: A beépített sorosítók kimeneti formátuma

A dolgozat során használt példaosztály teljes sorosított adatfolyama DataContractSerializer használta esetén:

```
<?xml version="1.0" encoding="utf-8"?>
<Employees xmlns:i=http://www.w3.org/2001/XMLSchema-instance
     xmlns="http://schemas.datacontract.org/
                    2004/07/EmployeeLib">
   <Address>507 - 20th Ave. E. Apt. 2A</Address>
   <BirthDate>1948-12-08T00:00:00</BirthDate>
   <City>Seattle</City>
   <Country>USA</Country>
   <EmployeeID>1</EmployeeID>
   <Extension>5467</Extension>
   <FavouriteNumbers  xmlns:d2p1="http://schemas.microsoft.com/
                    2003/10/Serialization/Arrays">
      <d2p1:int>1478526959</d2p1:int>
      <d2p1:int>125403918</d2p1:int>
      <d2p1:int>243117508</d2p1:int>
   </FavouriteNumbers>
   <FirstName>Nancy</FirstName>
   <HireDate>1992-05-01T00:00:00</HireDate>
   <HomePhone>(206) 555-9857</HomePhone>
   <LastName>Davolio</LastName>
   <MyId>0</MyId>
   <MyId2>0</MyId2>
   <Notes>Education includes a BA in psychology from Colorado State
    University in 1970.  She also completed "The Art of the Cold
    Call."  Nancy is a member of Toastmasters
    International.</Notes>
   <PhotoPath>http://accweb/emmployees/davolio.bmp</PhotoPath>
   <PostalCode>98122</PostalCode>
   <Region>WA</Region>
   <SensitiveData  xmlns:d2p1="http://schemas.microsoft.com/
                             2003/10/Serialization/Arrays">
      <d2p1:KeyValueOfstringint>
         <d2p1:Key>salary</d2p1:Key>
   <d2p1:Value>1531733115</d2p1:Value>
      </d2p1:KeyValueOfstringint>
      <d2p1:KeyValueOfstringint>
         <d2p1:Key>numOfChildren</d2p1:Key>
   <d2p1:Value>117442512</d2p1:Value>
      </d2p1:KeyValueOfstringint>
   </SensitiveData>
   <Title>Sales Representative</Title>
   <TitleOfCourtesy>Ms.</TitleOfCourtesy>
</Employees>
```

Ugyanez az objektumpéldány a NetDataContractSerializer sorosításában:

```
<?xml version="1.0" encoding="utf-8"?>
<Employees xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
           z:Id="1" z:Type="EmployeeLib.Employees"
           z:Assembly="ConsoleApplication1, Version=1.0.0.0,
                       Culture=neutral, PublicKeyToken=null"
          xmlns:z=http://schemas.microsoft.com/2003/10/Serialization/
          xmlns="http://schemas.datacontract.org/2004/07/EmployeeLib">
   <Address z:Id="2">507 - 20th Ave. E. Apt. 2A</Address>
   <BirthDate>1948-12-08T00:00:00</BirthDate>
   <City z:Id="3">Seattle</City>
   <Country z:Id="4">USA</Country>
   <EmployeeID>1</EmployeeID>
   <Extension z:Id="5">5467</Extension>
   <FavouriteNumbers xmlns:d2p1=http://schemas.microsoft.com/
                                        2003/10/Serialization/Arrays
                     z:Id="6">
      <d2p1:_items z:Id="7" z:Size="4">
         <d2p1:int>1478526959</d2p1:int>
         <d2p1:int>125403918</d2p1:int>
         <d2p1:int>243117508</d2p1:int>
         <d2p1:int>0</d2p1:int>
      </d2p1:_items>
      <d2p1:_size>3</d2p1:_size>
      <d2p1:_version>3</d2p1:_version>
   </FavouriteNumbers>
   <FirstName z:Id="8">Nancy</FirstName>
   <HireDate>1992-05-01T00:00:00</HireDate>
   <HomePhone z:Id="9">(206) 555-9857</HomePhone>
   <LastName z:Id="10">Davolio</LastName>
   <MyId>0</MyId>
   <MyId2>0</MyId2>
   <Notes z:Id="11"> Education includes a BA in psychology from
                     Colorado State University in 1970. She also
                     completed "The Art of the Cold Call."  Nancy is a
                     member of Toastmasters International.</Notes>
   <PhotoPathz:Id="12">http://accweb/emmployees/davolio.bmp
   </PhotoPath>
   <PostalCode z:Id="13">98122</PostalCode>
   <Region z:Id="14">WA</Region>
   <SensitiveData xmlns:d2p1=http://schemas.microsoft.com/
                                       2003/10/Serialization/Arrays
                  z:Id="15"
                  z:Type="System.Collections.Generic.Dictionary`2[
                          [System.String, mscorlib, Version=2.0.0.0,
                          Culture=neutral,
                          PublicKeyToken=b77a5c561934e089],
                          [System.Int32, mscorlib,Version=2.0.0.0,
                          Culture=neutral,
                          PublicKeyToken=b77a5c561934e089]]"
                  z:Assembly="0">
      <Version z:Id="16" z:Type="System.Int32" z:Assembly="0"
                          xmlns=""> 2 </Version>
      <Comparer z:Id="17"  z:Type="System.Collections.Generic
                                  .GenericEqualityComparer`1[
                                  [System.String,mscorlib,
                                  Version=2.0.0.0, Culture=neutral,
                                  PublicKeyToken=b77a5c561934e089]]"
                z:Assembly="0" xmlns="" />
      <HashSize z:Id="18" z:Type="System.Int32" z:Assembly="0"
```

```
                xmlns=""> 3 </HashSize>
        <KeyValuePairs z:Id="19"
                    z:Type="System.Collections.Generic.KeyValuePair`2
                    [[System.String, mscorlib, Version=2.0.0.0,
                    Culture=neutral,PublicKeyToken=b77a5c561934e089],
                    [System.Int32, mscorlib, Version=2.0.0.0,
                    Culture=neutral,
                     PublicKeyToken=b77a5c561934e089]][]"
                     z:Assembly="0" z:Size="2" xmlns="">
            <KeyValuePairOfstringint
                                xmlns="http://schemas.datacontract.org/
                                    2004/07/System.Collections.Generic">
                <key z:Id="20">salary</key>
                <value>1531733115</value>
            </KeyValuePairOfstringint>
            <KeyValuePairOfstringint
                                xmlns="http://schemas.datacontract.org/
                                    2004/07/System.Collections.Generic">
                <key z:Id="21">numOfChildren</key>
                <value>117442512</value>
            </KeyValuePairOfstringint>
        </KeyValuePairs>
    </SensitiveData>
    <Title z:Id="22">Sales Representative</Title>
    <TitleOfCourtesy z:Id="23">Ms.</TitleOfCourtesy>
</Employees>
```

E függelék: A sorosítás vázlatos algoritmusa kódban

A sorosításhoz használt algoritmus vázlatos kódja (System.Reflection névteres implementációval):

```
private static void innerSerialize(BinaryWriter bw, object graph)
{
  Type t = graph.GetType();
  //Kiírjuk a típusindexet
  bw.Write(serializetypes.IndexOf(t.AssemblyQualifiedName));
  //Ha atomi a típus...
  if (IsAtomic(t))
  {
   int typeindex = serializetypes.IndexOf(t.AssemblyQualifiedName);
   //...akkor közvetlenül sorosítunk a BinaryWriter megfelelő
   //metódusával.
   switch (typeindex)
   {
     case 0:
       bw.Write((string)graph);
        break;
     case 1:
       bw.Write((int) graph);
       break;
     //... hasonlóan a többi primitív típusra is
     case 14:
       bw.Write(((DateTime)graph).Ticks);
       break;
     default:
       bw.Write((int)graph); //(bármi egyéb csak enumeráció lehet)
       break;
    }
   }
   //Ha a típus generikus szótár...
   if (IsGenericDictionary(t))
   {
    //...akkor először meghatározzuk milyen típusú elemek alkotják,
    //hogy végig tudjunk lépkedni rajta
    Type genericKeyValue=typeof(KeyValuePair<,>)
                         .MakeGenericType(t.GetGenericArguments());
    //...kikeressük a metódust, ami megmondja hány elemünk van...
    MethodInfo counter =
            (from result in
              (from method in typeof(Enumerable).GetMethods()
               where method.Name == "Count"
               select method)
             where result.GetParameters().Count() == 1
             select result)
             .Single().MakeGenericMethod(genericKeyValue);
     int dcount = (int)counter.Invoke(null, new[] { graph });
     bw.Write(dcount);
```

```
      //...kikeressük a metódust, amivel tudunk indexelni...
      MethodInfo dindexer =
             (from method in (typeof(Enumerable).GetMethods())
              where method.Name == "ElementAt"
              select method)
              .Single().MakeGenericMethod(generickeyvalue);
      //...majd egyesével végiglépkedünk a szótáron...
      for (int i = 0; i < dcount; i++)
      {
         //... és sorosítjuk először a kulcsot, majd az értéket.
         object keyvaluepair = dindexer.
                                 Invoke(null, new[] { graph, i });
         object key=keyvaluepair.GetType().GetProperty("Key")
                                        .GetValue(keyvaluepair, null);
         object value = keyvaluepair.GetType().GetProperty("Value")
                                        .GetValue(keyvaluepair, null);
         innerSerialize(bw, key); innerSerialize(bw, value);
       }
       return;
      }
      if (IsGenericCollection(t))
      {
      //Az egyéb generikus gyűjtemények esetén ugyanígy játunk el...
      }
      //Egyébként pedif valamilyen komplex típusról van szó...
      PropertyInfo[] properties;
      //...lekérjük a tulajdonságokat (és cachelünk)
      if (!propertycache.ContainsKey(t.AssemblyQualifiedName))
      {
         properties = t.GetProperties(BindingFlags.Public |
                                        BindingFlags.Instance);
         //...rendezzük a tulajdonságokat név szerint...
         Array.Sort(properties, (a, b) => a.Name.CompareTo(b.Name));
         propertycache.Add(t.AssemblyQualifiedName, properties);
      }
      else
      {
          properties = propertycache[t.AssemblyQualifiedName];
      }
      FieldInfo[] fields;
      //...hasonlóan járunk el mezők esetében is...
      if (!fieldcache.ContainsKey(t.AssemblyQualifiedName))
      { //... }
      int propnum = properties.Length; int fieldnum = fields.Length;
      //...majd sorosítjuk az összes sorosítható tulajdonságot...
      for (int i = 0; i < propnum; i++)
      {
         if (IsPropertyWritable(properties[i]))
           innerSerialize(bw, properties[i].GetValue(graph, null));
       }
       //... és mezőt.
       for (int i = 0; i < fieldnum; i++)
       {
         if (IsFieldWritable(fields[i]))
           innerSerialize(bw, fields[i].GetValue(graph));
       }
    }
```

F függelék: A FixedBinarySerializer és a generált kód

A könnyebb összehasonlíthatóság kedvéért álljon itt a FixedBinarySerializer forráskódja illetve a példaosztályra dinamikusan generált sorosító kódja:

A FixedBinarySerializer kódja:

```
public static void Serialize(Stream s, object graph)
{
  BinaryWriter bw = new BinaryWriter(s);
  Employees e = (Employees)graph;
  bw.Write(e.Address);
  bw.Write(e.BirthDate.Ticks);
  bw.Write(e.City);
  bw.Write(e.Country);
  bw.Write(e.EmployeeID);
  bw.Write(e.Extension);
  bw.Write(e.FavouriteNumbers.Count);
  foreach (int k in e.FavouriteNumbers)
  {
    bw.Write(k);
  }
  bw.Write(e.FirstName);
  bw.Write(e.HireDate.Ticks);
  bw.Write(e.HomePhone);
  bw.Write(e.LastName);
  bw.Write(e.MyId);
  bw.Write(e.MyId2);
  bw.Write(e.Notes);
  bw.Write(e.PhotoPath);
  bw.Write(e.PostalCode);
  bw.Write(e.Region);
  bw.Write(e.SensitiveData.Count);
  foreach (var keyValuePair in e.SensitiveData)
  {
    bw.Write(keyValuePair.Key);
    bw.Write(keyValuePair.Value);
  }
  bw.Write(e.Title);
  bw.Write(e.TitleOfCourtesy);
  bw.Flush();
}
```

A dinamikusan generált kód pedig a következőképpen néz ki (a hívási viszonyok a 13. ábra - A sorosításban részt vevő, generált metódusok című képen láthatóak):

```
public static void SerializeTypeEmployees(
                                          Employees employees,
                                          BinaryWriter binaryWriter
                                          )
{
  binaryWriter.Write(16);
  binaryWriter.Write(employees.Address);
  binaryWriter.Write(employees.BirthDate);
  binaryWriter.Write(employees.City);
  binaryWriter.Write(employees.Country);
  binaryWriter.Write(employees.Extension);
  List<int> favouriteNumbers = employees.FavouriteNumbers;
  Type type = favouriteNumbers.GetType();
  if (type == typeof(Employees[]))
  {
    Employees[]Serializer.SerializeTypeEmployees[]
                                        (
                                        (Employees[])favouriteNumbers,
                                         binaryWriter
                                         );
  }
  else
  {
    if (type == typeof(Employees))
    {
      Employees[]Serializer.SerializeTypeEmployees
                                      (
                                      (Employees)favouriteNumbers,
                                      binaryWriter
                                      );
    }
    else
    {
      if (type == typeof(List<int>))
      {
        Employees[]Serializer.SerializeTypeList`1
                                        (
                                          (List<int>)favouriteNumbers,
                                          binaryWriter
                                         );
      }
      else
      {
        if (type == typeof(Dictionary<string, int>))
        {
          Employees[]Serializer.SerializeTypeDictionary`2
                              (
                                (Dictionary<string,int>)favouriteNumber,
                                 binaryWriter
                              );
        }
      }
    }
  }
```

```
  binaryWriter.Write(employees.FirstName);
  binaryWriter.Write(employees.HireDate);
  binaryWriter.Write(employees.HomePhone);
  binaryWriter.Write(employees.LastName);
  binaryWriter.Write(employees.MyId);
  binaryWriter.Write(employees.MyId2);
  binaryWriter.Write(employees.Notes);
  binaryWriter.Write(employees.PhotoPath);
  binaryWriter.Write(employees.PostalCode);
  binaryWriter.Write(employees.Region);
  Dictionary<string, int> sensitiveData = employees.SensitiveData;
  Type type2 = sensitiveData.GetType();
  if (type2 == typeof(Employees[]))
  {
      Employees[]Serializer.SerializeTypeEmployees[]
                                          (
                                            (Employees[])sensitiveData,
                                            binaryWriter
                                          );
  }
  else
  {
    if (type2 == typeof(Employees))
    {
      Employees[]Serializer.SerializeTypeEmployees
                                  (
                                      (Employees)sensitiveData,
                                       binaryWriter
                                   );
    }
    else
    {
      if (type2 == typeof(List<int>))
      {
        Employees[]Serializer.SerializeTypeList`1
                                      (
                                         (List<int>)sensitiveData,
                                         binaryWriter
                                      );
      }
      else
      {
        if (type2 == typeof(Dictionary<string, int>))
        {
          Employees[]Serializer.SerializeTypeDictionary`2
                               (
                                 (Dictionary<string, int>)sensitiveData,
                                  binaryWriter
                               );
        }
      }
    }
  }
  binaryWriter.Write(employees.Title);
  binaryWriter.Write(employees.TitleOfCourtesy);
  binaryWriter.Write(employees.EmployeeID);
}
```

```
public static void SerializeTypeList(List<int> list,
                                      BinaryWriter binaryWriter)
{
  binaryWriter.Write(17);
  binaryWriter.Write(((ICollection<int>)list).Count);
  IEnumerator<int> enumerator =((ICollection<int>)list)
                               .GetEnumerator();
  while (enumerator.MoveNext())
  {
    binaryWriter.Write(enumerator.Current);
  }
}

public static void SerializeTypeDictionary(
                                Dictionary<string, int> dictionary,
                                BinaryWriter binaryWriter
                                )
{
  binaryWriter.Write(18);
  binaryWriter.Write(
                 ((ICollection<KeyValuePair<string,int>>)dictionary
                  .Count);
IEnumerator<KeyValuePair<string, int>> enumerator =
               ((ICollection<KeyValuePair<string,int>>)dictionary)
                                                   .GetEnumerator();
  while (enumerator.MoveNext())
  {
    KeyValuePair<string, int> current = enumerator.Current;
    binaryWriter.Write(current.Key);
    current = enumerator.Current;
    binaryWriter.Write(current.Value);
  }
}
```

A generált kódban feltűnhet, hogy a metódusok nevei nem szabályosak. Ennek oka az, hogy az ilyen metódusok esetén szövegesen azonosítjuk a metódust mind létrehozáskor, mind híváskor, így a „nem megengedett karakter” fogalmát nem értelmezhetjük – bármi szerepelhet a névben. Mivel a típusokkal azonosítjuk a metódusokat, így a típusneveket a metódusokba elhelyezve egyértelművé válik az azonosítás; a típusnevek viszont tartalmazhatnak ilyen speciális karaktereket (különösen a szintén dinamikusan, a fordító által generált típusok, mint például a generikusok esetében); ezeket pedig a reflektor nem tudja pontosan értelmezni (és a Visual Studio szintaxis-kiemelése sem működik teljesen jól; sőt ez a kód így nem is fordítható).

A két kód hasonlósága azonban szembetűnik; egyedüli különbség, hogy ahol szükséges kiírjuk a típusokat, illetve például a gyűjtemények esetében van egy indirekciós lépés, amikor eldönti a sorosító, hogy milyen típust kell sorosítani és meghívja rá a megfelelő függvényt. Ezek a függvények viszont már nagyon hasonlóan működnek a FixedBinarySerializer megfelelő részletéhez.