1. Poglavlje: **Uvod**

Distribuirani proces bi mogli shvatiti kao više odvojenih fizičkih komponenti koje zajedno rade kao jedan sistem. Odvojenih komponenti možemo shvatiti da imamo više različitih CPU, ili češće da imamo više različitih računala na mreži.
O motivaciji i potrebama za distribuiranjem nećemo sada (nego možda na predavanjima), nego ćemo u ovom uvodnom poglavlju dati kratak povijesni razvoj trehnika distribuiranog programiranja pod Microsoft OS-om.
U početku, aplikacije su se gradile na cetralnom serveru oko kojeg su bili terminali. Sav posao je odrađivao server, pa je morao imati jak hadrware. Sav posao je odrađivala jedna (velika) aplikacija, pa ju je teško održavati. Povezivanje s drugim aplikacijama je bilo komplicirano.

2. Poglavlje: C# i .net framework

C# je case-sensitive programski jezik. Sintaksa mu je vrlo slična C++-u, pa ćemo samo spomenuti koje su razlike.
C# je objektno orjentirani programski jezik. To znači da svaka funkcija/varijabla mora biti unutar neke klase. Ne postoje globalne funkcije/objekti, pa ako želimo koristi takve stvari, moramo ih navesti kao statičke (i cijele klase možemo navesti kao statičke, što je zgodno kada želimo preopteretiti operatore na već postojećim tipovima). Programi su grupirani u Projekte. Projekti mogu biti različiti library ili izvršne datoteke. Ukoliko je program izvršna datoteka, treba imati statičku funkciju Main, koja označava početak programa.

```
class Program {
  static void Main(string[] args) { }
}
```

Iako C# možemo koristiti u različitim razvojnim alatima, mi ćemo koristiti isključivo Visual Studio (budući da je na fakultetu instalirana verzija 2005, raditi ćemo u njoj).
Da smo startali novi projekt, i odabrali C#, pa Console application, prethodni probram bi mogli kompajlirati sa Build->Build Solution, ili sa Ctrl-Shift-B, a pokrenuti sa Debug->Start Without Debugging (Ctrl-F5) ili u radu za otklanjanje grešaka sa Debug->Start Debugging (F5).
Rezultat kompajliranja programa je assembly, a on može biti .dll (za library-e) ili .exe (za izvršne datoteke).
C# je strogo tipizirani programski jezik, to znači da prilikom kreiranja/korištenja nekog objekta moramo znati kojeg je on tipa (od verzije 2010 to nije nužno, ali je bolje).
Osim osnovnih tipova (char, int, string, …) .net framework ima tisuće gotovih tipova, koji su na osnovu sličnosti grupirani u različite namespaceove. Ukoliko ne želimo svaki puta koristiti potpuno ime, npr.

```
System.Console.WriteLine("Pozdrav");
```

možemo na početku programa napisati using, npr.

```
using System;
```

pa bi onda prethodna naredba bila ekvivalentna ovoj kraćoj:

```
Console.WriteLine("Pozdrav");
```

Osim gotovih tipova, C# nam pruža da definiramo i vlastite tipove. To možemo pomoću ključnih riječi enum, struct i class. Svi tipovi se dijele na vrijednosne i referentne. Vrijednosni su ugrađeni brojevni tipovi, stringovi, te svi napravljeni sa enum i struct. Oni se kreiraju na stogu te se uništavaju kao i u C++-u (u C# struktura se ne može nasljeđivati, ali može zadovoljavati interface). Svi ostali C# tipovi su napravljeni korištenjem class riječi. Oni zapravo predstavljaju nešto slično smart-pointerima, kreiraju se na heapu, te ih možemo shvatiti slično kao pointere u C-u (dakle npr. izmjena vrijednosti u funkciji će rezultirati izmjenom i u glavnom programu koji ih je kreirao). O uništavanju class-a se brine Garbrage Collector (GC). On *pohvata* sve objekte koji više nikom ne trebaju, pa ih pouništava kada mu to odgovara (najčešće tek na završetku programa ili kada nestane memorije na računalu). Način i vrijeme uništavanja objekata možemo i mi kontrolirati, ali to nije svrha ovoga kolegija, pa o tome nećemo puno govoriti. Također, C# omogućuje i korištenje normalnih pointera, što nije ni često, niti se preporuča, pa to isto nećemo raditi.
Klase u C#-u možemo i nasljeđivati, ali ne postoji višestruko nasljeđivanje, međutim svaka klasa može nasljediti (zadovoljavati) više interfacea (čisto apstraktnih klasa).

.net pruža brojne gotove klase za rad sa spremnicima, a od verzije 2005 svi spremnici mogu raditi i u generičkom obliku, a to su:

- nizovi proizvoljnog tipa, npr. (ako odmah navodimo elemente spremnika, ne moramo navesti njegovu veličinu)

```
int[] a = new int[] { 1, 2, 3, 4, 5 };
```

  a imamo i mogućnost rada s višedimenzionalnim nizovima (nejednakih ili pravokutnih oblika)

```
int[][] b = new int[][] { { 1, 2, 3 }, { 4, 5, 6 } };
int[,] c = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };
```

- Queue<T> - predstavlja FIFO spremnik, slično kao queue u STL-u
- List<T>, LinkedList<T> - predstavlja dvostruko linkanu listu objekata tipa T, slično kao list u STL-u
- SortedDictionary<K,V> - predstavlja kolekciju parova key=value, slično kao map u STL-u
- Dictionary<K,V> slično kao SortedDictionary, ali se elementima pristupa na osnovu hash-vrijednosti ključa
- SortedList<K,V> slično kao SortedDictionary, ali zauzima manje prostora u memoriji, i sporije je ubacivanje elemenata

Kroz spremnike osim korištenjem `[]` i odgovarajućih `for`, `while` ili `do`, možemo i korištenjem `foreach` ključne riječi.

2.1 Funkcije

Funkcije u C# mogu vraćati void, ili bilo koji drugi tip. Funkcije možemo preopterećivati, dovoljno je da se razlikuju u barem jednom tipu parametra (od verzije 2010 imamo i opcione parametre). Za razliku od C++-a, svaki parametar može biti ulazni (ništa ne navedemo), ulazno/izlazni (ref) ili izlazni (out):

```
static void f1(int x) { ++x; }
static void f2(ref int x) { ++x; }
static void f3(out int x)
{
    // ++x; ovo bi bila greska
    x = 0; ++x;
}
static void Main(string[] args)
{
    int x = 10;
    f1(x); Console.WriteLine(x);      // 10
    f2(ref x); Console.WriteLine(x); // 11
    f3(out x); Console.WriteLine(x); // 1
}
```

U .net-u svaki element može imati i dodatne atribute koji govore kompajleru kako da ih shvati. Ti atributi se pišu u uglatim zagradama ispred elementa.

2.2 Kreiranje, pretvorbe tipova i uništavanje objekata

Svaki podatak mora biti inicijaliziran, u protivnom ćemo dobiti grešku prilikom njegovog korištenja. Svi referentni tipovi (napravljani sa class) su u stvari izvedeni iz tipa System.Object i oni po defaultu imaju vrijednost postavljenu na null. Da bismo ih koristili, moramo ga inicijalizirati:

```
tip t = new tip();
```

Svaki vrijednosni tip ima i tip na koji dodajemo znak ?, što znači da on može imati vrijednost null ili sve one vrijednosti koje je imao polazni tip:

```
int? x = null;
x = 10;
```

Implicitne pretvorbe brojčanih tipova rade kao i u C++-u; rade ako se ne gube podaci ili ako izvedeni tip pretvaramo u bazni

```
int x = 5;
double y = x;
```

a za obrnutu stvar moramo navesti da 'znamo što radimo':

```
double x=5;
int y = (int)x;
```

Kastanje iz izvedenog u bazni tip radi bez problema, ali kada želimo obratnu stvar, objekt moramo skastati u izvedeni tip. Ukoliko stvarni objekt nije izvedenog tipa (ili nekog izvedenog iz njeg), dobit ćemo Exception. Zbog toga su u C# uvedeni 2 operatora.

```
            bazni x = new bazni();
            izvedeni y = x as izvedeni;
```

rezultat će biti null ako nije pravog tipa (kao kod dynamic-casta u C++-u). Ukoliko nam ne treba stvarni tip, nego samo želimo provjeriti je li objekt toga tipa (što često može biti korisno), možemo umjesto `as` koristiti `is`, koji vraća true/false ovisno o tome je li toga tipa.

2.2.1 Uništavanje objekata

Ukoliko želimo kontrolirati kada nam objekt više ne treba (ako želimo osloboditi ne *skupe* resurse), potrebno je klasu izvesti iz IDisposable:

```
class Program
{
     class cl : IDisposable
    {
        public cl()
        {
            Console.WriteLine("Konstruktor");
        }
         public void Dispose()
         {
             Console.WriteLine("Dispose");
             GC.SuppressFinalize(this); // da GC više ne vodi brigu

         }
         ~cl()
         {
             Console.WriteLine("Destruktor");
         }

    }
    static void Main(string[] args)
    {
        {
            cl c = new cl();
            try
            {
                ...
            }
            finally
            {
                c.Dispose();
```

```
            }
        }
        Console.WriteLine("Program gotov");
    }
}
```

Skraćenica za kreiranje i ovakav oblik naredbe sa Dispose u finally dijelu je koristeći using:

```
            using(cl c = new cl())
            {
                ...
            }
```

2.3 Klase i strukture

Klase i strukture nam omogućuju da kreiramo vlastite tipove. One se sastoje od spiska proizvoljnih polja (varijable), property-a (to su funkcije koje koristimo slično kao varijable, ali pišemo kôd koji će očitavati/mijenjati vrijednost same varijable) i funkcija (i drugih stvari koje nam u ovom kolegiju neće trebati). Oblik pristupa svakom elemenatu može biti private (to je defaultno), protected, internal (vidljiv unutar jednog assemblya), te public.
U C# projektu možemo i koristiti objekte napravljene i u drugim projektima. Potrebno je samo u referencama projekta dodati željeni objekt (to može biti assembly bilo kojeg projekta u .net-u, pa i pisan u drugim programskim jezicima, bilo koja COM klasa i drugi elementi). Ukoliko odaberemo neki .net assemby, VS će automatski tu datoteku (ako je to privatni assembly, a takve ćemo koristiti u ovom kolegiju; postoje i dijeleni, koji se instaliraju u GAC, i koje onda ne treba kopirati, ali mi ih nećemo koristiti zbog nepotrebnih komplikacija prilikom njihovog kreiranja)) prekopirati u folder gdje je trenutni assembly (i ona se mora zajedno kopirati prilikom kopiranja programa). Tip objekta se određuje na osnovnu njegovog imena, a tako isto i pristup elementima, pa ako slučajno ili namjerno prekopiramo novi assemby, imena moraju biti ista, i sve će uredno raditi (nema potrebe da se drugi projekt ponovo kompajlira).
Što se konstruktora i destruktora tiče, sve je potpuno jednako kao u C++-u, jedino tšo u C#-u polja klase možemo odmah inicijalizirati prilikom deklaracije, a imamo i jedan specijalni konstruktor, kojeg označavamo sa static u kojem možemo inicijalizirati statičke elemente (i koji će biti automatski pozvan prije prvog korištenja klase).
Što se nasljeđivanja tiče, stvari su vrlo slične kao u C++-u, ali je svako nasljeđivanje javno, pa se to ne navodi. Ukoliko u konstruktoru u inicijalizacijskoj listi želimo pozvati konstruktor bazne klase, to radimo pomoću ključne riječi base, a ne imena bazne klase. Inače, svaki element klase je instanciran s defaultnim konstruktorom (brojevi imaju vrijednost 0, string je prazan string, a objekti imaju vrijednost null).

2.4 Serijalizacija

Pod serijalizacijom podrazumijevamo da objekt svoje trenutno stanje pretvori u niz byteova iz kojeg se onda deserijalizacijom omogućava iz tog niza kreiranje objekta sa istim stanje. To je pogodno bilo za spremanje objekta na neki medij, bilo za kreiranje identičnih objekata na udaljenom računalu. C# ima ugrađenu podršku za serijalizaciju. Svi ugrađeni tipovi se mogu serializirati, a kod vlastitih tipova je dovoljno klasi navesti atribut [Serializable] (i sva polja klase moraju imati taj atribut). Ukoliko neko polje klase nije potrebno serializirati, kod njega navedemo atribut [NonSerialized].
Sljedeći primjer varijablu (a može i više njih) sprema u datoteku auto.txt u .xhtml obliku (za pokretanje je u reference potrebno dodati System.Runtime.Serialization.Formatters.Soap).

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
```

```
class Program
{
    [Serializable]
    class automobil
    {
        int presaoKilometara;
        string registracija;

        [NonSerialized]
        int trenutnaBrzina;

        public automobil(string reg, int km, int brz)
        {
            presaoKilometara = km;
            registracija = reg;
            trenutnaBrzina = brz;
        }
        public void ispis() {
            Console.WriteLine("{0}, {1}, {2}", registracija,
presaoKilometara, trenutnaBrzina);
        }
    }
    static void Main(string[] args)
    {
        automobil a = new automobil("ZG-000-ZG", 123456, 50);
        FileStream fs = File.Create("auto.txt");

        new SoapFormatter().Serialize(fs, a);

        fs.Close();

        Citaj();
    }
    static void Citaj()
    {
        FileStream fs = File.Open("auto.txt", FileMode.Open);

        automobil a = (automobil)new SoapFormatter().Deserialize(fs);

        fs.Close();
        a.ispis();
    }

}
```

Ukoliko bismo željeli podatke spremiti binarno, a ne u xml obliku, možemo koristiti `BinaryFormatter` klasu (koja se nalazi u `...Formatters.Binary` i njega ne moramo dodati u reference).
Ponekad ipak defaultno ponašanje Serializacije nije dovoljno. Ako želimo bolju kontrolu, trebamo dodati dvije metode i ISerizable interface:

```
using System.Runtime.Serialization;
    [Serializable]
    class automobil : ISerializable
    {
...
        public void GetObjectData(SerializationInfo i, StreamingContext
c)
        {
            i.AddValue("presaoKilometara", presaoKilometara);
            i.AddValue("registracija", registracija);
        }
```

```
        automobil(SerializationInfo i, StreamingContext c)
        {
            presaoKilometara = i.GetInt32("presaoKilometara");
            if (c.State != StreamingContextStates.CrossProcess)
                registracija = i.GetString("registracija");
            //  x = (Tip)i.GetValue("x", typeof(Tip));
        }
    }
```

Na mjestu točkica je isti dio klase kao i u prethodnom primjeru, dok bi zakomentirani oblik trebali koristiti ako neki vlastiti tip imamo u klasi.
Na ovaj način možemo promijeniti što se serializira. Ako bismo željeli mijenjati kako se stvari serializiraju, trebali bismo napraviti klasu iz Formatter klase.

3. Poglavlje: **Threadovi**

C# pruža brojne načine za kreiranje i rad sa dretvama (engl. thread). Svaki thread ima svoj stog (lokalne varijable, returne na koje se treba vratiti, svoje handlere za Exceptione).

Kreiranje dretve

Ukoliko želimo kontrolu nad threadom, koristit ćemo objekt tipa System.Threading.Thread:

```
using System;
using System.Threading;
class Program {
  int threadNo = 0;
  Program(int no) { threadNo = no; }
  void drugiThread() {
    for (int i = 0; i < 10; ++i) {
      Thread.Sleep(100);
      Console.WriteLine("Thread: {0:D}, i={1:D}", threadNo, i);
    }
  }
  static void Main() {
    Program p1 = new Program(1);
    Thread t = new Thread(new ThreadStart(p1.drugiThread)); t.Start();
    Program p = new Program(0); p.drugiThread();
  }
}
```

Ukoliko threadu želimo poslati nekakve parametre, možemo koristiti oblik:

```
using System;
using System.Threading;
class Program {
  static void f(object o) {
    for (int i = 0; i < 10; ++i) {
      Thread.Sleep(100);
      Console.WriteLine("Thread: {0:D}, i={1:D}", o, i);
    }
  }
  static void Main() {
    Thread t1 = new Thread(new ParameterizedThreadStart(f));
    t1.Start(1);
    Thread t2 = new Thread(new ParameterizedThreadStart(f));
    t2.Start(2);
    f(0);
  }
}
```

Ako su threadovi ovako kreirani, oni će se nastaviti izvršavati do kraja i nakon što glavni thread završi izvršavanje.

`System.Threading.Thread` klasa ima mnogo funkcija. Mi ćemo spomenuti samo neke. Spomenimo prvo statičke (odnose se na trenutni thread):

- `Sleep(int miliSec)` – čeka određeni broj milisekundi, procesor nije opterećen
- `SpinWait(int vrijeme)` – vrti petlju određeno vrijeme. Ovisi o procesoru koliko, procesor je opterećen
- `VolatileRead(obj)` – čita podatak iz memorije (a ne iz cachea ili drugdje)
- `VolatileWrite(obj)` – piše podatak u memoriju

Te nestatičke (iz jednog threada kontroliramo drugi):

- `Start()` – pokreće thread
- `Abort()` – prekida izvršavanje threada
- `Suspend()` – pauzira thread
- `Resume()` – pokreće pauzirani thread
- `Join(int miliSec)` – čeka završetak threada ili istek vremena
- `Interrupt()` – prekida čekanje threada.

Sinhronizacija

Globalne i statičke varijable (a i drugi resursi, pa i lokalni podaci) mogu biti zajednički threadovima, pa je ponekad potrebno osigurati da samo jedan thread pristupa varijabli. Jedan od najjednostavnijih načina je koristeći `lock`:

```
lock(obj) {
  ...
}
```

Na ovaj način samo jedan thread može ući u kritični dio koda zaključan varijablom `obj`. `obj` je bilo koji tip izveden iz `Object` (dakle svaka klasa). Više različitih dijelova koda možemo zaključati istim `obj`-em, ali moramo paziti da je i `obj` zajednička varijabla threadovima (da se ne bi dogodilo da svaki thread ima svoj ključ).

```
using System;
using System.Threading;
namespace Lock {
  class Account {
    private Object thisLock = new Object();
    int balance;
    Random r = new Random();
    public Account(int initial) { balance = initial; }
    int Withdraw(int amount)
    { // Ovo se nece dogoditi ako je lock aktivan
      if (balance < 0) { throw new Exception("Negativno stanje"); }
      // Komentirajte sljedecu liniju
      lock (thisLock)
      {
        if (balance >= amount) {
          Console.WriteLine("Stanje prije : " + balance);
          Console.WriteLine("Iznos koji se podize : -" + amount);
          balance = balance - amount;
          Console.WriteLine("Stanje nakon : " + balance);
          return amount;
        }
        else { return 0; // transakcija odbijena }
      }
    }
  public void DoTransactions() {
    for (int i = 0; i < 100; i++)
      Withdraw(r.Next(1, 100));
    }
  }
  class Program {
    static void Main() {
      Thread[] threads = new Thread[10];
      Account acc = new Account(1000); // zajednicki objekt za threadove
      for (int i = 0; i < 10; i++)
        threads[i] = new Thread(new ThreadStart(acc.DoTransactions));
      for (int i = 0; i < 10; i++) threads[i].Start();
      for (int i = 0; i < 10; ++i) threads[i].Join();
```

```
      Console.ReadKey();
    }
  }
}
```

AutoResetEvent

Služi nam da signaliziramo drugom threadu da se desio neki događaj (tako da može nastaviti sa nekim poslom)

- `Wait(int miliSec)` – čeka sve dok ne istekne vrijeme ili objekt ne bude signaliziran naredbom `Set`. Samo jedan thread može nastaviti s radom, te objekt odmah prelazi u nesignalizirano stanje
- `Set()` – signalizira da jedan thread može nastaviti sa radom (ako čeka)
- `Reset()` – ako je objekt u signaliziranom stanju, prelazi u nesignalizirano

`ManualResetEvent` – nakon `Set`, svi čekajući threadovi mogu nastaviti s radom, sve dok se ne pozove `Reset`.

Na sljedećem primjeru izmijenite Manual/AutoReset evente, te izmijenite vremena čekanja i komentirajte promjene:

```
using System;
using System.Threading;
namespace AutoReset {
  class Account {
  //    public AutoResetEvent a = new AutoResetEvent(false);
  public ManualResetEvent a = new ManualResetEvent(false);
  public void Radi() {
    Console.WriteLine("{0} poceo cekanje", Thread.CurrentThread.Name);
    a.WaitOne(3000);
    Thread.Sleep(200);
    Console.WriteLine("{0} zavrsio cekanje", Thread.CurrentThread.Name);
    a.WaitOne(3000);
    Console.WriteLine("{0} zavrsio 2 cekanje",
Thread.CurrentThread.Name);
    }
  }
  class Program {
    static void Main() {
      Thread[] threads = new Thread[10];
      Account acc = new Account(); // acc će biti zajednički objekt
      for (int i = 0; i < 2; i++) {
        Thread t = new Thread(new ThreadStart(acc.Radi));
        t.Name = i.ToString();
        threads[i] = t;
      }
      for (int i = 0; i < 2; i++) {
        threads[i].Start();
      }
      Console.WriteLine("threadovi zapoceli");
      acc.a.Set();
      Thread.Sleep(100);
      acc.a.Set();
      acc.a.Reset();
    }
  }
}
```

Postoje i brojni drugi načini za kreiranje threada, te za asinhrono pozivanje funkcija (neki su navedeni kasnije u primjerima)

```
using System;
using System.Threading;
```

```
public class App {
  static WaitHandle[] waitHandles = new WaitHandle[] { new
AutoResetEvent(false), new AutoResetEvent(false) };
  static Random r = new Random();
  static void DoTask(Object state) {
    AutoResetEvent are = (AutoResetEvent)state;
    int time = 1000 * r.Next(2, 10);
    Console.WriteLine("Zapoceo posao od {0} milisekundi.", time);
    Thread.Sleep(time); are.Set();
  }
  static void Main() {
    DateTime dt = DateTime.Now;
    Console.WriteLine("Glavni thread ceka da OBA threada zavrse");
    ThreadPool.QueueUserWorkItem(DoTask, waitHandles[0]);
    ThreadPool.QueueUserWorkItem(DoTask, waitHandles[1]);
    WaitHandle.WaitAll(waitHandles); // Vrijeme najduzeg posla.
    Console.WriteLine("Oba threada su zavrsila (proteklo vrijeme={0})",
(DateTime.Now - dt).TotalMilliseconds);

    dt = DateTime.Now;
    Console.WriteLine();
    Console.WriteLine("Glavni thread ceka da bilo koji thread zavrsi");
    ThreadPool.QueueUserWorkItem(DoTask, waitHandles[0]);
    ThreadPool.QueueUserWorkItem(DoTask, waitHandles[1]);
    int index = WaitHandle.WaitAny(waitHandles); // Vrijeme najkraceg
posla, rezultat je indeks handlea
    Console.WriteLine("Thread {0} je zavrsio (proteklo vrijeme={1}).",
index + 1, (DateTime.Now - dt).TotalMilliseconds);
  }
}
```

Ukoliko glavni thread završi izvršavanje, poslovi koji se u pozadini izvršavaju, kreirani na `ThreadPool`-u neće biti završeni (nego prekinuti završetkom glavnog threada)!

`System.Threading` sadrži još mnoštvo sinhronizacijskih mehanizama, poput `Interlocked`, `Mutex`, `Monitor`, `Semaphore`, `ReaderWriterLock`, te neke objekte za callbackove kao npr. `Timer` koji poziva callback nakon isteka određenog vremena. Za detalje pogledati help od `System.Threading`.

4. Poglavlje: **Delegati**

Delegati (engl. *delegates*) u programskom jeziku C# su referentni tipovi koji se koriste za učahurivanje metoda. U delegata se može učahuriti bilo koja odgovarajuća metoda (s određenim potpisom i povratnim tipom) – slično kao pointeri na funkcije u C++-u, s tim da delegati imaju brojne prednosti.

Deklaracija delegata

Delegat se stvara pomoću ključne riječi **delegate** iza koje se navodi povratni tip i potpis metoda koje mu se mogu delegirati:

```
[modifikator pristupa] delegate NazivDelegata(potpisMetode);
```

Na primjer, sljedećom deklaracijom:

```
public delegate int KojiJePrvi(object o1, object o2);
```

definira se delegat `KojiJePrvi` koji će učahuriti bilo koju metodu koja uzima dva parametra tipa `object` i vraća tip `int`.
Nakon definiranja delegata u njega možemo učahuriti metodu članicu tako što ćemo ga instancirati prosljeđivanjem metode koja odgovara navedenom povratnom tipu i potpisu. Također možemo koristiti i *anonimne* metode, što je opisano kasnije. U oba slučaja delegat se može koristiti za pozivanje učahurene metode.

Primjer korištenja delegata

```
public class Osoba {
  public string ime, prezime;
}
public delegate int usporedi(Osoba o1, Osoba o2);
class Program {
  public static int usporediPoImenu(Osoba o1, Osoba o2) {
    return o1.ime.CompareTo(o2.ime);
  }
  public static int usporediPoPrezimenu(Osoba o1, Osoba o2) {
    return o1.prezime.CompareTo(o2.prezime);
  }
  static void Main(string[] args) {
    usporedi u = new usporedi(usporediPoImenu);
    // ili u = usporediPoImenu;
    Osoba o1, o2;
     ...
    if (u(o1, o2) > 0) { ... }
  }
}
```

Delegate možemo i poslati metodi:

```
... void Poredaj (usporedi delegiranaMetoda) {
  if (delegiranaMetoda(x, y)) ...
}
```

Višeodredišni delegati

Ponekad je korisno kroz jednog delegata pozvati dvije (ili više) implementirane metode. To možemo postići kombiniranjem delegata pomoću **operatora zbrajanja** (`+`). Rezultat je novi

višeodredišni delegat (engl. *multicast delegate*) koji poziva obje izvorne implementirane metode.
Na primjer, imamo li definirane delegate `Zapisivac` i `Biljeznik`, oni se mogu kombinirati u novi višeodredišni delegat `ViseodredisniDelegat` na sljedeći način:

```
ViseodredisniDelegat = Zapisivac + Biljeznik;
```

Delegate možemo dodavati višeodredišnom delegatu i pomoću operatora **plus-jednako** (`+=`), koji će delegata zdesna dodati višeodredišnom delegatu slijeva.
Na primjer, želimo li našem višeodredišnom delegatu `ViseodredisniDelegat` dodati delegat `Odasiljac`, možemo to učiniti koristeći operator `+=` ovako:

```
ViseodredisniDelegat += Odasiljac;
```

Za uklanjanje pojedinog delegata iz višeodredišnog delegata na isti se način koristi operator **minus-jednako** (`-=`).
Na primjer, želimo li iz odredišnog delegata `ViseodredisniDelegat` ukloniti delegat `Biljeznik`, možemo to učiniti ovako:

```
ViseodredisniDelegat -= Biljeznik;
```

Anonimne metode

Da bi skratili pisanje kôda, delegate možemo koristiti za kreiranje anonimnih metoda

```
  delegate int binOp(int x, int y);
  static void Main(string[] args) {
    binOp usp = delegate(int x, int y) {
      return x-y;
    };
    ...
    int x = usp(10, 20)
  }
```

Anonimne metode možemo koristiti i kao parametre za funkcije koje startaju novi thread:

```
using System;
using System.Threading;
class Program {
    static void Main(string[] args) {
        int x = 10;
        ThreadPool.QueueUserWorkItem(delegate {
            while (true) {
                Console.WriteLine(x);
                Thread.Sleep(100);
            }
        });
        for (x = 100; x < 1000; ++x)
            Thread.Sleep(10);
    }
}
```

Ili

```
    static void Main(string[] args) {
        int x = 10;
        Thread t = new Thread(delegate() {
            while (true) {
                Console.WriteLine(x);
                Thread.Sleep(100);
            }
        });
        t.Start();
```

```
        for (x = 100; x < 200; ++x)
            Thread.Sleep(10);
        t.Abort();
    }
```

Kao varijable koje koristimo u tijelu anonimne metode, ne smijemo koristiti ref i izlazne parametre, ali ulazne i lokalne varijable smijemo (kao i sve globalno/statičko). One će živjeti i nakon što se završi blok kod-a u kojem je kreirana anonimna metoda.
U vs2008 imamo lambda operator koji još pojednostavljuje kreiranje anonimnih metoda.

Asinhrono pozivanje delegata

Osim sinhronog pozivanja funkcija, delegati nam omogućuju i asinhrono pozivanje (`BeginInvoke`) – automatski se kreira novi thread koji izvršava metodu, a naš thread nastavlja sa radom. Za ovakav način, delegat ne smije biti višeodredišni.

```
using System;
using System.Threading;

delegate int binOp(int x, int y);
class Program {
  static int zbroj(int x, int y) {
    Console.WriteLine("Zbroj {0}\n",
      Thread.CurrentThread.GetHashCode());
    return x + y;
  }
  static void radi() {
    Console.WriteLine("radi {0}\n",
      Thread.CurrentThread.GetHashCode());
    }
  static void Main(string[] args) {
    zbroj(10, 20);
    binOp o = new binOp(zbroj);
    o(10, 20);
    o.BeginInvoke(10, 20, null, null);
    ThreadPool.QueueUserWorkItem(delegate { radi(); });
    Thread.Sleep(1000);
  }
}
```

U prethodnom primjeru nam je bila potrebna naredba `Sleep`, jer se u protivnom dijelovi kôda koje izvršavaju radne dretve vjerojatno ne bi izvršili, jer bi glavna dretva završila s radom. Ukoliko želimo dobiti povratne rezultate izvršavanja `BeginInvoke` naredbe, možemo koristiti sa `EndInvoke`.

```
IAsyncResult i = o.BeginInvoke(10, 20, null, null);
...
int rez = o.EndInvoke(i);
```

Potpis naredbe `EndInvoke` sadrži sve ref/izlazne parametre početne funkcije, te rezultat `BeginInvoke` naredbe (jer istim delegatom možemo više puta pozvati više različitih `BeginInvoke` naredbi). Npr. za delegat:

```
delegate string del(int x, ref int y, out double z);
```

potpis bi bio:

```
del d = ...
int y = 0, y1 = 0; double z, z1;
IAsyncResult i = d.BeginInvoke(10, ref y, out z),
  i1 = d.BeginInvoke(20, ref y1, out z1);
...
string s = d.EndInvoke(ref y, out z, i),
```

```
  s1 = d.EndInvoke(ref y1, out z1, i1);
```

`EndInvoke` pauzira trenutnu dretvu dok pozvana funkcija ne završi, te nakon toga lijepi na parametre povratne vrijednosti. Bez nje bi izlazne/povratne vrijednosti bile izgubljene, iako se završilo izvršavanje!
Osim `EndInvoke`, stanje izvršavanja možemo provjeriti i nekim drugim dijelovima `IAsyncResult` interfacea.

```
using System;
using System.Threading;
delegate void del();
class Program {
  static void radi() {
    Console.WriteLine("radi poceo {0}\n",
Thread.CurrentThread.GetHashCode());
    Thread.Sleep(1000);
    Console.WriteLine("radi zavrsio {0}\n",
Thread.CurrentThread.GetHashCode());
  }
  static void Main(string[] args) {
    del d = radi;
    IAsyncResult ar = d.BeginInvoke(null, null);
    while (!ar.IsCompleted) {
      Console.WriteLine("Glavni thread nešto radi {0}\n",
Thread.CurrentThread.GetHashCode());
      Thread.Sleep(100);
    }
  }
}
```

Interface ima i event koji će biti signaliziran kada završi izvršavanje asinhronog poziva (ali tada još uvijek povratni parametri neće biti preneseni).

```
    del d = radi;
    IAsyncResult ar = d.BeginInvoke(null, null);
    while (!ar.AsyncWaitHandle.WaitOne(100)) {
      Console.WriteLine("Glavni thread malo čeka, pa nešto radi {0}\n",
Thread.CurrentThread.GetHashCode());
      Thread.Sleep(100);
    }
```

Ponekad je korisno dodati callback metodu koja će se izvršiti nakon što završi izvršavanje `BeginInvoke` naredbe (koja će eventualno nekome signalizirati nešto, ili izvršiti `EndInvoke`, da bismo pokupili izlazne parametre). Takav callback je procedura koja ima jedan parametar tipa `IAsyncResult`:

```
  static void Main(string[] args) {
    del d = radi;
    IAsyncResult ar = d.BeginInvoke(AddCallback, null);
    Console.WriteLine("Glavni thread radi {0}\n",
Thread.CurrentThread.GetHashCode());
    d.EndInvoke(ar);
    Console.WriteLine("Glavni thread ide dalje\n");
  }
  static void AddCallback(IAsyncResult ar) {
    Console.WriteLine("AddCallback se izvršava {0}",
    Thread.CurrentThread.GetHashCode());
  }
```

Često je korisno izvršiti i `EndInvoke` u callbacku. `IAsyncCallback` nema podatke o delegatu koji ga je pozvao, pa ga trebamo skastati u `AsyncCallback` (bez I), i uz još mao posla dobijemo:

```
delegate int binOp(int x, int y);
class Program {
  static int radi(int x, int y) {
    Console.WriteLine("radi poceo {0}\n",
Thread.CurrentThread.GetHashCode());
    Thread.Sleep(1000);
    return x+y;
  }
  static void Main(string[] args) {
    binOp d = radi;
    IAsyncResult ar = d.BeginInvoke(10, 20, AddCallback, null);
    Console.WriteLine("Glavni thread radi {0}\n",
Thread.CurrentThread.GetHashCode());
    Thread.Sleep(2000);
    Console.WriteLine("Glavni thread ide dalje\n");
  }
  static void AddCallback(IAsyncResult ar) {
    Console.WriteLine("AddCallback se izvršava {0}",
Thread.CurrentThread.GetHashCode());
    int value = Convert.ToInt32(ar.AsyncState);
    // poslijednji parametar BeginInvoke naredbe - kod nas null
    Console.WriteLine(value);

    System.Runtime.Remoting.Messaging.AsyncResult aResult =
(System.Runtime.Remoting.Messaging.AsyncResult)ar;
    binOp b = (binOp)aResult.AsyncDelegate;

    int rez = b.EndInvoke(ar);
    Console.WriteLine("Rezultat je {0}\n", rez);
  }
}
```

Kroz posljednji parametar možemo poslati (budući da je tipa `object`) u stvari što god želimo callback metodi.

5. Poglavlje: .NET remoting – kreiranje objekata

.net framework ima više mogućnosti međuprocesne komunikacije. Za komunikaciju na istom računalu, najbrži način je korištenjem anonymous/named pipes. Nešto sporiji način, ali koji pruža mogućnost komunikacije i procesa na udaljenim računalima je korištenjem Socketa, koji radi tako da nizove podataka šaljemo/primamo, pa je zbog toga u imalo složenijim problemima izuzetno komplicirano. .net remoting je vrlo jednostavan način koji nam omogućuje kreiranje distribuiranih programa, radeći s objektima kao da su lokalni, a pruža i veliku fleksibilnost u odabiru protokola komunikacije (zavisno o security i drugim ograničenjima), pa ćemo u ovom kolegiju raditi isključivo .net remoting.
Osim kreiranja referentnih objekata koristeći ključnu riječ `new`, objekte možemo kreirati i na druge načine. `Activator.GetObject` je način vrlo pogodan u .NET remotingu. Da bi se kreirao objekt, mora biti poznat njegov tip, te gdje se objekt nalazi. Funkcija vraća objekt, pa ga je nakon toga još potrebno skastati u potreban tip. Ukoliko se objekt stvarno nalazi u procesu, dobit ćemo stvarni objekt, a ako se nalazi u drugom procesu/računalu, dobit ćemo proxy, koji će glumiti objekt. Mi ćemo mu pristupati potpuno jednako kao da je pravi objekt, a .NET framework će se brinuti da sve pozive funkcijama, parametre, povrtatne vrijednosti… mrežom proslijedi do stvarnog objekta, tamo izvrši željenu metodu, te sve povratne parametre vrati nazad pozivatelju. Za vrijeme toga, klijentov thred čeka, a na serveru se kreira novi thread koji izvrši zahtjev. Taj novi thread slijedi logički thread klijenta (i sve dodatne parametre logičkog threada – o čemu ćemo govoriti kasnije).

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace ClientServerTalk {
  class Server : MarshalByRefObject {
    public void Talk(string s) {
      Console.WriteLine("Klijent pozvao Talk {0}\n", s);
    }
  }
  class Program {
    static void ServerMain() {
      RemotingConfiguration.RegisterWellKnownServiceType(typeof(Server),
"ClientServerTalk.Server", WellKnownObjectMode.Singleton);
      Console.WriteLine("Server started (Singleton)");
      Console.ReadLine();
    }
    static void ClientMain() {
      Server server = (Server)Activator.GetObject(typeof(Server),
@"tcp://localhost:12345/ClientServerTalk.Server");
      if (server == null) {
        System.Console.WriteLine("Nema servera");
        return;
      }
      string s;
      do {
        s = Console.ReadLine();
        server.Talk(s);
      } while (s != "");
    }
    static void Main(string[] args) {
```

```
      bool server = true;
      TcpChannel chan;
      try {
        chan = new TcpChannel(12345);
      }
      catch {
        server = false;
        chan = new TcpChannel();
      }
      ChannelServices.RegisterChannel(chan);
      if (server) ServerMain(); else ClientMain();
    }
  }
}
```

Da bi program bio uspješno kompajliran, u reference je potrebno dodati .net referencu `System.Runtime.Remoting`. Da ne bismo komplicirali sa više projekata (inače su programi obično grupirani u više projekata. Jedan projekt je server, jedan klijent, a ukoliko kao parametre/povratne vrijednosti koristimo neke vlastite tipove, običaj je i te tipove grupirati u zasebni library), u programu je i serverski i klijentski program. Ukoliko je port 12345 slobodan, na njemu će se startati server, a ako nije, startati će se klijentski main.
Server jedino dopusti objektu tipa `Server` (koji mora biti izveden iz `MarshalByRefObject`, i imati defaultni konstruktor) da bude kreiran koristeći remoting. Moramo mu navesti tip, naziv i da li je `Singleton` (statički objekt na serveru – svim klijentima se daje isti objekt) ili `SingleCall` (prilikom svakog poziva – i od istog klijenta – se kreira novi objekt, izvrši funkcija, te se on uništi, pa nam na ovom kolegiju takvi objekti neće biti interesantni, jer uglavnom nema dijeljenja resursa).
Na početku kao kanal možemo koristiti `HttpChannel` (po defaultu koristi tekstualni formatter) ili `TcpChannel` (po defaultu ima binarni formatter).
Prilikom kreiranja objekta na klijentu, naziv igleda:

```
tcp://ime_hosta:port/naziv
```

gdje je prva riječ `http:` ili `tcp:`, `ime_hosta` je ip adresa ili naziv računala na mreži, `port` je port koji smo odabrali prilikom registracije kanala servera, a `naziv` je naziv koji smo na početku rada servera odabrali, prilikom poziva `RegisterWellKnownType` (jedan server je mogao registrirati i više objekata). Kada aktivator kreira objekt, on u stvari samo kreira proxy, i u tom trenutku uopće još nema mrežne komunikacije. Prva komunikacija (i kreiranje objekta) se dogodi prilikom poziva prve metode, ili pristupa prvom parametru.
Svi parametri koji se šalju ili vraćaju mrežom, se moraju moći serializirati (pretvoriti u niz byteova) ili moraju biti izvedeni iz `MarshalByRefObject` (u tom slučaju se na serveru kreira proxy na taj objekt).
U sljedećem primjeru (potrebno je dodati na početku using System.Threading), a glavni main ostaviti istim:

```
[Serializable]
class Podaci {
  public int x;
  public void f() { Console.WriteLine("Završna vrijednost: {0}", x); }
}
class Server : MarshalByRefObject {
  public void Talk(ref Podaci p) {
    for(p.x=0; p.x<100; ++p.x)
      Thread.Sleep(10);
    p.f();
  }
}
class Program {
  static void ServerMain() {
```

```
      RemotingConfiguration.RegisterWellKnownServiceType(typeof(Server),
"ClientServerTalk.Server", WellKnownObjectMode.Singleton);
      Console.WriteLine("Server started (Singleton)");
      Console.ReadLine();
    }
    static void ClientMain() {
      Server server = (Server)Activator.GetObject(typeof(Server),
@"tcp://localhost:12345/ClientServerTalk.Server");
      if (server == null) {
        System.Console.WriteLine("Nema servera");
        return;
      }
      Podaci p = new Podaci();
      p.x=0;
      ThreadPool.QueueUserWorkItem(delegate { while (true) {
Console.WriteLine(p.x); Thread.Sleep(100); } });
      server.Talk(ref p);
      Console.WriteLine(p.x);
    }
```

Vidimo da će se `p.f()` izvršiti na serveru, a klijentski main, iako se varijabla mijenja neće to primjetiti sve dok server ne završi izvršavanje funkcije (tek onda se povratna vrijednost šalje nazad klijentu).

Ukoliko klasu `Podaci` ne navedemo kao serializabilnu, nego izvedemo iz `MarshalByRefObject`, na serveru će se umjesto njene kopije kreirati proxy, koji će glumiti stvarnu klasu, a varijablu `x` uvećavati na klijentu, te funkciju `f` izvršiti na klijentu. Da bi to moglo raditi, klijentski se mora ponašati slično kao server, tj. kanal je potrebno registrirati na nekom portu, ili portu 0 (.net framework će sam slobodno odabrati port – kao i prije), pa će i klijent kreirati thread koji će osluškivati nadolazeće zahtjeve. Još je potrebno formatteru servera omogućiti da dopusti marshaliranje svih tipova podataka (a ne samo ugrađenih i potpisanih – to je bilo uvedeno zbog sigurnosti). Sljedeći program uistinu uvećava varijablu `x` na klijentu i radi kako je opisano:

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Threading;

namespace ClientServerTalk {
  class Podaci : MarshalByRefObject {
    public int x;
    public void f() { Console.WriteLine("Završna vrijednost: {0}", x); }
  }
  class Server : MarshalByRefObject {
    public void Talk(Podaci p) {
      for(p.x=0; p.x<100; ++p.x)
         Thread.Sleep(10);
      p.f();
    }
  }

  class Program {
    static void ServerMain() {
      RemotingConfiguration.RegisterWellKnownServiceType(typeof(Server),
"ClientServerTalk.Server", WellKnownObjectMode.Singleton);
      Console.WriteLine("Server started (Singleton)");
      Console.ReadLine();
    }
    static void ClientMain() {
```

```
      Server server = (Server)Activator.GetObject(typeof(Server),
@"tcp://localhost:12345/ClientServerTalk.Server");
      if (server == null) {
        System.Console.WriteLine("Nema servera");
        return;
      }
      Podaci p = new Podaci();
      p.x=0;
      ThreadPool.QueueUserWorkItem(delegate { while (true) {
Console.WriteLine(p.x); Thread.Sleep(100); } });
      server.Talk(p);
      Console.WriteLine(p.x);
    }
    static void Main(string[] args) {
      bool server = true;
      TcpChannel chan;
      try {
        BinaryServerFormatterSinkProvider provider = new
BinaryServerFormatterSinkProvider();
        provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
        IDictionary props = new Hashtable();
        props["port"] = 12345;
        chan = new TcpChannel(props, null, provider);
      }
      catch {
        chan = new TcpChannel(0);
        server = false;
      }
      ChannelServices.RegisterChannel(chan);
      if (server) ServerMain(); else ClientMain();
    }
  }
}
```

Aplikacijske domene i konteksti

Aplikacijska domena je .net zamjena za proces operacijskog sustava. Jedan stvarni proces ima bar jednu, a može imati više aplikacijski domena. Aplikacijske domene nemaju zajedničke podatke, tako da rušenjem jedne, ostale domene nastavljaju normalno sa radom. A jedna od prednosti (u odnosu na procese) je da se smanjuje vrijeme potrebno za međuprocesnu komunikaciju. Mi u ovome kolegiju nećemo imati potrebe za rad s njima, pa o tome nećemo više govoriti.

Unutar svake aplikacijske domene postoji barem jedan kontekst (a može ih biti i više). Oni služe da se objekti sa sličnim specijalnim zahtjevima kreiraju tamo, pa da se onda pozivi upućeni njima izvršavaju kreiranjem međuproxy-a (to .net automatski radi).

Kontekstno slobodni objekti su svi oni koji nisu izvedeni iz ContextBoundObject, i oni se mogu izvršavati u svakom kontekstu. Ali napravimo li recimo sljedeću klasu

```
    [Synchronization]
    class sink : ContextBoundObject
    {
        ...
    }
```

Ona će se izvršavati u svom kontekstu, što će u ovom specijalnom slučaju značiti da će se svako njeno korištenje rezultirati proxy-em koji će paziti da samo jedan thread istovremeno može pristupati elementima (poljima/funkcijama). Pokušajte sljedeći program izvršiti sa i bez toga atributa (ili nasljeđivanja):

```
using System.Runtime.Remoting.Contexts;
```

```
class Program
{
    [Synchronization]
    class sink : ContextBoundObject
    {
        public int x=0;
        public void f()
        {
            System.Threading.Thread.Sleep(5000);
        }

    }

    static void Main(string[] args)
    {
        sink s = new sink();
        System.Threading.ThreadPool.QueueUserWorkItem(delegate
        {
            s.f();
        });
        System.Threading.Thread.Sleep(1000);
        Console.WriteLine(s.x);
        Console.WriteLine("Program gotov");
    }
}
```

Statičke metode i varijable

Statičke metode se izvršavaju uvijek izvršavaju u pozivateljevoj app. domeni. Statičkim varijablama ne možemo pristupiti remotely.

Eventi i remoting

Eventi i većina drugih sinhronizacijskih objekata iz `System.Threading` su izvedeni iz `MarshalByRefObject`, te se kao takvi mogu koristiti i iz drugih procesa:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Threading;

namespace ClientServerEvent {
  class ServerskaKlasa : MarshalByRefObject {
    public EventWaitHandle ev = new AutoResetEvent(false);

    public ServerskaKlasa() {
      ThreadPool.QueueUserWorkItem(delegate {
        while(true) {
          ev.WaitOne();
          Console.WriteLine("Netko je signalizirao event");
        }
      } );
    }
  }
  class Program {
    static void ServerMain() {
```

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerskaKlasa
), "KlasaSer", WellKnownObjectMode.Singleton);
      Console.WriteLine("Server started (Singleton)");
      Console.ReadLine();
    }
    static void ClientMain() {
      ServerskaKlasa server =
(ServerskaKlasa)Activator.GetObject(typeof(ServerskaKlasa),
@"tcp://localhost:12345/KlasaSer");
      if (server == null) {
        System.Console.WriteLine("Could not locate server");
        return;
      }
      while(true) {
        Console.ReadKey();
        server.ev.Set();
      }
    }
    static void Main(string[] args) {
      bool server = true;
      TcpChannel chan;
      try {
        chan = new TcpChannel(12345);
      }
      catch {
        chan = new TcpChannel();
        server = false;
      }
      ChannelServices.RegisterChannel(chan);
      if (server) ServerMain(); else ClientMain();
    }
  }
}
```

Ukoliko sa serverske strane želimo signalizirati event na klijentu, na klijentu moramo kreirati kanal sa čekajućim threadom, a formatter servera mora omogućavati sve tipove

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Threading;

namespace ClientServerEvent {
  class ServerskaKlasa : MarshalByRefObject {
    public void Signal(EventWaitHandle e) { e.Set(); }
  }
  class Program {
    static void ServerMain() {

RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerskaKlasa
), "KlasaSer", WellKnownObjectMode.Singleton);
      Console.WriteLine("Server started (Singleton)");
      Console.ReadLine();
    }
    static void ClientMain() {
      ServerskaKlasa server =
(ServerskaKlasa)Activator.GetObject(typeof(ServerskaKlasa),
@"tcp://localhost:12345/KlasaSer");
      if (server == null) {
```

```
        System.Console.WriteLine("Could not locate server");
        return;
      }
      AutoResetEvent ev = new AutoResetEvent(false);
      ThreadPool.QueueUserWorkItem(delegate {
        while (true) {
          ev.WaitOne();
          Console.WriteLine("Netko je signalizirao event");
        }
      });
      while (true) {
        Console.ReadKey();
        server.Signal(ev);
      }
    }
    static void Main(string[] args) {
      bool server = true;
      TcpChannel chan;
      try {
        BinaryServerFormatterSinkProvider provider = new
BinaryServerFormatterSinkProvider();
        provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
        IDictionary props = new Hashtable();
        props["port"] = 12345;
        chan = new TcpChannel(props, null, provider);
      }
      catch {
        chan = new TcpChannel(0);
        server = false;
      }
      ChannelServices.RegisterChannel(chan);
      if (server) ServerMain(); else ClientMain();
    }
  }
}
```

Delegati i remoting, OneWay

Cijela priča sa delegatima potpuno dobro radi i u remoting situaciji (ako je metoda koju delegat učahuruje u drugom procesu/računalu). Jedino ako koristimo callback moramo biti oprezni, jer će se callback izvršavati na klijentskom računalu (pa klijent mora glumiti i servera – imati kanal sa kreiranim threadom koji će osluškivati nadolazeće remoting zahtjeve).
Ukoliko nas ne zanima rezultat funkcije sa servera (što bi smanjilo mrežnu komunikaciju), kao ni da li se desila greška (ni da li server uopće postoji) bolje rješenje je prilikom pisanja serverske klase, određenoj metodi dodati `OneWay` atribut (nebitno je što je ovdje http kanal):

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Threading;

class SerKl : MarshalByRefObject {
  [System.Runtime.Remoting.Messaging.OneWay]
  public void Radi() {
    Console.Write("Poceo... {0}", Thread.CurrentThread.GetHashCode());
    Thread.Sleep(5000);
    Console.WriteLine("Gotov {0}", Thread.CurrentThread.GetHashCode());
  }
}
```

```
class Program {
  static void ServerMain() {
    RemotingConfiguration.RegisterWellKnownServiceType(typeof(SerKl),
"KlasaSer", WellKnownObjectMode.Singleton);
    Console.WriteLine("Server started (Singleton)");
    Console.ReadLine();
  }
  static void ClientMain() {
    SerKl server = (SerKl)Activator.GetObject(typeof(SerKl),
@"http://localhost:12345/KlasaSer");
    if (server == null) return;
    while (true) {
      Console.ReadKey();
      Console.WriteLine("Poceo... ");
      server.Radi();
      Console.WriteLine("Gotov");
    }
  }
  static void Main(string[] args) {
    bool server = true;
    HttpChannel chan;
    try { chan = new HttpChannel(12345); }
    catch {
      chan = new HttpChannel();
      server = false;
    }
    ChannelServices.RegisterChannel(chan);
    if (server) ServerMain(); else ClientMain();
  }
}
```

Exceptioni

Ukoliko se za vrijeme klijentskog poziva neke funkcije dogodi Exception na serveru, srušit će se klijentski program, a server će normalno nastaviti izvršavanje. Sa klijentske možemo hvatati iznimku tipa Exception, ali ako želimo dodatne informacije koje je server eventualno poslao, nećemo moći razlikovati tip iznimke. Da bismo to mogli, klasa preko koje šaljemo podatke o iznimki ne smije biti izvedena iz System.Exception, nego iz RemotingException, te mora biti Seriazibilna i mora imati implementacije funkcija, te ručno moramo napraviti Serializaciju:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Serialization;

namespace ClientServerTalk
{
  [Serializable]
  public class Ex : RemotingException {
    public string _poruka;
    public Ex(string message) { _poruka = message; }
    public Ex(SerializationInfo info, StreamingContext context) {
      _poruka = (string)info.GetValue("_tekstPoruke", typeof(string));
    }
    public override void GetObjectData(SerializationInfo info,
StreamingContext context) {
      info.AddValue("_tekstPoruke", _poruka);
    }
    public override string Message {
      get { return "Ovo je iznimka: \"" + _poruka + "\""; }
```

```
    }
  }
  class Server : MarshalByRefObject {
    public void Radi() {
      throw new Ex("Neka greska");
    }
  }
  class Program {
    static void ServerMain(){

RemotingConfiguration.RegisterWellKnownServiceType(typeof(Server),
"ClientServerTalk.Server", WellKnownObjectMode.Singleton);
            Console.WriteLine("Server started (Singleton)");
            Console.ReadLine();
        }
    static void ClientMain() {
      Server server = (Server)Activator.GetObject(typeof(Server),
@"tcp://localhost:12345/ClientServerTalk.Server");
      if (server == null) {
        System.Console.WriteLine("Nema servera");
        return;
      }
      try {
        server.Radi();
      }
      catch (Ex e) {
        Console.WriteLine("Greska: {0}", e._poruka);
      }
    }
    static void Main(string[] args) {
      bool server = true;
      TcpChannel chan;
      try {
        chan = new TcpChannel(12345);
      }
      catch {
        server = false;
        chan = new TcpChannel();
      }
      ChannelServices.RegisterChannel(chan);
      if (server) ServerMain(); else ClientMain();
    }
  }
}
```

6. Poglavlje: Životni vijek objekata

Lokalni objekati žive dok su nekom potrebni, a kad više nikom nisu potrebni (broj referenci padne na nulu), prepušteni su Garbrage Collectoru (GC-u) da ih uništi kada mu to bude odgovaralo. Sa remote objektima je stvar donekle drugačija. Da se ne bi nepotrebno opterećivala mreža (a možda je neki klijent nestao s mreže), remote objekti žive određeno vrijeme (po defaultu je to 5 minuta mirovanja), i nakon toga budu prepušteni GC-u na milost i nemilost. Dakle iako držimo referencu na neki objekt, ako ne pozovemo ni jednu njegovu metodu 5 minuta, on će biti uništen. Srećom te stvari možemo mijenjati i kontrolirati.

ILease

MarshalByRefObject klasa ima metodu GetLifeTimeService, pomoći koje možemo dobiti defaultno vrijeme. Dobiveni objekt je tipa ILease, a on ima sljedeće elemente:

- CurrentLeaseTime – Trenutno preostalo vrijeme života objekta
- InitialLeaseTime – Vrijeme koje se dobije na početku
- RenewOnCallTime – Vrijeme koje se dobije prilikom svakog poziva funkcije

Ako želimo sami promijeniti ove stvari, na serverskoj klasi je potrebno izmijeniti funkciju:

```
using System.Runtime.Remoting.Lifetime;

public class Klasa : MarshalByRefObject {
  public override object InitializeLifetimeService() {
    // return null; // ako želimo beskonačni vijek
    ILease i = (ILease)base.InitializeLifetimeService();
    i.InitialLeaseTime = TimeSpan.FromSeconds(10);
    i.RenewOnCallTime = TimeSpan.FromSeconds(5);
    return i;
  }
}
```

ISponsor

Svaki objekt može imati i sponzora, koji može nakon isteka vremena produljiti životni vijek nekom objektu. Sponsor može biti bilo kakav objekt koji zadovoljava ISponsor interface. Kada životni vijek nekom objektu istekne, .net framework prvo provjeri ima li objekt sponzora (može ih imati i više), te ih redom ispituje da li žele produljiti vijek. Ako ne žele, uklanjaju se s liste sponzora. A ako žele, vijek biva obnovljen. Ako nitko ne želi, objekt će biti poslan GC-u.
ISponsor ima samo jednu metodu, kako se vidi da sljedećem primjeru (klasa će tri puta produljiti vijek za 5 sekundi)

```
public class Sponzor : ISponsor {
  private int pomoziDruguUNevolji = 0;
  public TimeSpan Renewal(ILease l) {
    if (++pomoziDruguUNevolji < 3) return TimeSpan.FromSeconds(5);
    return TimeSpan.FromSeconds(0);
  }
}
```

Još je potrebno i dodati sponzora. Logično mijesto bi bilo prilikom inizijalizacije Lifetime-a:

```
public class Klasa : MarshalByRefObject {
  public override object InitializeLifetimeService() {
    ILease i = (ILease)base.InitializeLifetimeService();
    i.Register(new Sponzor());
    return i;
```

```
  }
}
```

Sponzore možemo dodavati i na drugim mjestima, pa i klijentske klase mogu biti sponzori serverskima, ali onda sponzor mora biti izveden iz MarshalByRefObject, zadovoljavati ISponsor interface i klijent mora biti startan s kanalom koji ima thread koji osluškuje zahtjeve. Međutim, što ako je klijentski sponzor nedostupan (bilo da je mreža isključena/jako spora, ili je bio remotly kreiran, pa je njegov život istekao)? ILeaseInfo ima i element SponsorshipTimeout – vrijeme koje će se čekati da sponzor odgovori na zahtjev.
`LifetimeServices.LeaseManagerPollTime` – možemo podesiti koliko često će .net framework provjeravati ima li objekata kojima je isteklo vrijeme.
Ukoliko želimo da se objekt uništi odmah nakon što mu istenke period (a ne kada to bude odgovaralo GC-u), možemo iskorisiti sponzore na sljedeći način:

```
public class Klasa : MarshalByRefObject, IDisposable {
  public override object InitializeLifetimeService() {
    ILease i = (ILease)base.InitializeLifetimeService();
    i.Register(new DispSponzor(this));
    return i;
  }
  public void Dispose() {
    // oslobađanje unmanaged resursa

    GC.SuppressFinalize();
  }
  ~Klasa() { }
}
public class DispSponzor : ISponsor {
  private IDisposable obj;
  public DispSponzor(IDisposable o) { obj = o; }
  public TimeSpan Renewal(ILease l) {
    obj.Dispose();
    return TimeSpan.FromSeconds(0);
  }
}
```

Ako kasnije i dodamo još sponzora, ovaj će biti zadnji pozvan, kada svi otkažu sponzorstvo. (Napomenimo samo još to da konstruktor i Dispose neće neće biti pozvani iz istih threadova).

7. Poglavlje: **Remoting Configuration**

Do sada smo naučili nešto o kreiranju objekata i pravljenju mrežnih aplikacija. Međutim, pristup koji smo do sada koristili ima i neke nedostatke:

- Ukoliko se bilo što promijeni (kanal, port, sastav mreže), sve moramo prekompajlirati ponovo
- Serverski objekti moraju biti kreirani koristeći defaultni konstruktor
- Svi klijenti moraju imati referencu na serverski objekt

Prvi problem možemo riješiti koristeći konfiguracijske datoteke. Umjesto da direktno u kôdu pišemo gdje i kako se server/klijent startaju, .net framework pruža mogućnost da to napravimo koristeći konfiguracijske datoteke. One su u .xml obliku, te je ukoliko se nešto u infrastrukturi promijeni, dovoljno je promijeniti te datoteke (i ponovo startati program), pa nema potrebe za rekompajliranjem svega.

Remoting Configuration

Svi remoting elementi se trebaju staviti unutar <system.runtime.remoting> taga u <configuration> tagu .config datoteke (ukoliko takva ne postoji, treba ju dodati u Solution Exproreru->Add->New Item…->Application Configuration File.
Početni primjer (servera) s poglavlja o remotingu bi u tom slučaju izgledao ovako (izmjena je jedino što koristimo HTTP kanal):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton" type="ClientServerTalk.Server,
ClientServerTalk" objectUri="mojObjekt" />
      </service>
      <channels>
        <channel port="12345"
type="System.Runtime.Remoting.Channels.Http.HttpChannel,
System.Runtime.Remoting, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Obratite pažnju da prekinute linije trebaju biti jedna linija. Umjesto ove duge definicije kanala, možemo pisati samo:

```
        <channel port="12345" ref="http" />
```

A ref može imati vrijednosti „http“, „http client“, „http server“, „tcp“, „tcp client“ i „tcp server“.
A sam kôd servera bi bio (ne trebaju nam dosta namespaceova):

```
using System;
using System.Runtime.Remoting;

namespace ClientServerTalk
{
    public class Server : MarshalByRefObject
    {
        public void Talk(string s)
```

```
        {
            Console.WriteLine("Klijent pozvao Talk {0}\n", s);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure("Server.exe.config");
            Console.WriteLine("Server started (Singleton)");
            Console.ReadLine();
        }
    }
}
```

A kod klijenta bi bio:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName="biloSto">
        <wellknown type="ClientServerTalk.Server, ClientServerTalk"
url="http://localhost/mojObjekt" />
      </client>
      <channels>
        <channel ref="http" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

A u samom kôdu objekte kreiramo koristeći najobičniji new (i trebamo dodati referencu na servera):

```
using System;
using System.Runtime.Remoting;

using ClientServerTalk;

namespace ClientServerTalk
{
    class Program {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure("Client.exe.config");
            Server s = new Server();
            s.Talk("nesto");

        }
    }
}
```

Klijent-aktivirajući objekti

Osim korištenjem *well-known* objekata (negdje se kaže i *server-activated)*, MBR objekte ponuditi prema van i korištenjem *client*-activated objekata. Osnovna razlika u njihovom korištenju je kako se kreiraju i koliki im je životni vijek. Kod *wll-known* objekata smo ih kreirali tako da smo naveli ime servera i njegovo jedinstveno ime i on je bio inicijaliziran defaultnim konstruktorom. Takav objekt su ili dijelili svi klijenti, ili je živio samo za vrijeme poziva jedne funkcije.

Client-activated objekte možemo kreirati korištenjem bilo kojeg konstruktora. Klijentski aktivirani objekti nisu ni singlecall ni singleton, nego svako kreiranje rezultira novim kreiranjem na serveru, ali objekt živi dok je potreban klijentu (slično kao normalno kreirani objekti). Ukoliko mu istekne vrijeme (po defaultu 5 min) da nitko nije pozivao njegove funkcije, objekt će biti uništen (osim ako to nismo promijenili ili ako ima sponzora…).
Budući da nam za poterbe ovog kolegija takvi objekti nisu pretjerano interesantni (dijeljeni resursi nisu odmah vidljivi, moramo ih ručno učiniti takvima – problemi dijeljenja su kao u poglavlju o threadovima) samo ćemo napraviti jedan primjer kako napraviti takav klijent/server. Prvo navodimo primjer bez konfiguracijskih datoteka (static polje je dijeljeno, da se vidi da se mogu postići dijeljeni resursi) u kojem su klijent i server spojeni u jedan program:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace ClientServerTalk
{
    class Server : MarshalByRefObject
    {
        string ime;
        static int brojac = 0;
        public Server(string s) { ime = s; }
        public void Talk(string s)
        {
            Console.WriteLine("{2}. Klijent {0} pozvao Talk {1}\n", ime,
s, ++brojac);
        }
    }
    class Program
    {
        static void ServerMain()
        {

RemotingConfiguration.RegisterActivatedServiceType(typeof(Server));
            Console.WriteLine("Server started (Singleton)");
            Console.ReadLine();
        }
        static void ClientMain()
        {
            string ime = Console.ReadLine();

RemotingConfiguration.RegisterActivatedClientType(typeof(Server),
"tcp://localhost:12345");
            Server server = new Server(ime);
            if (server == null)
            {
                System.Console.WriteLine("Nema servera");
                return;
            }
            string s;
            do
            {
                s = Console.ReadLine();
                server.Talk(s);
            } while (s != "");
        }
        static void Main(string[] args)
        {
```

```
            bool server = true;
            TcpChannel chan;
            try
            {
                chan = new TcpChannel(12345);
            }
            catch
            {
                server = false;
                chan = new TcpChannel();
            }
            ChannelServices.RegisterChannel(chan);
            if (server) ServerMain(); else ClientMain();
        }
    }
}
```

Ukoliko želimo koristiti konfiguracijske datoteke, klijentsko i serversko konfiguriranje je gotovo jednako kao u slučaju wellknown objekata, jedino tag u .xml-u nije <welknown…>, nego

```
        <service type="ClientServerTalk.Server, ClientServerTalk" />
```

Sa serverske strane i client element s klijentske bi trebao biti:

```
       <client url="http://localhost:12345">
        <activated type="ClientServerTalk.Server, ClientServerTalk" />
       </client>
```

Inače, tih activated i wellknown tipova u jednom projektu možemo imati više i jedne i druge vrste. Što se tiče klijentski aktiviranih objekata, svaki objekt određenog tipa mora biti istanciran na istom serveru. Pa ipak možemo imati više različitih <client> tagova, pa pomoću njih možemo na jednom klijentu podizati objekte (različitih tipova) sa različitih servera.
U .xml datoteci možemo podešavati i druge stvari, pa npr. u <application> tag možemo ubaciti element (ili samo dio njegovih svojstava):

```
<lifetime leaseTime="10s" renewOnCallTime="5s" sponsorshipTimeout="5s"
leaseManagerPollTime="5s"/>
```

8. Poglavlje: CallContextData & ThreadStorage

Svaki thread ima svoj stog i svoje lokalne varijable. Ukoliko želimo da neki objekt bude zajednički za više threadova, označiti ćemo ga sa static, te će onda njegovo mjesto biti u kôdu, a ne na stogu. Takvi objekti se samo jednom inicijaliziraju, čak i kada su lokalne varijable neke funkcije. Ukoliko želimo da se neki objekt ponaša kao statički, ali da ne bude dijeljeni objekt threadovima, to moramo navesti kompajleru atributom [ThreadStatic]. To nema smisla za lokalne varijable, nego samo za elemente klase. Takav objekt će biti statički za svaki thread posebno. Dakle, svim klasama koje kreiramo unutar istog threada će biti zajednički, ali klasama koje se izvršavaju u različitim threadovima neće.

```
using System;
using System.Threading;

class Program
{
    [ThreadStatic] static int x = 0;
    static void Main(string[] args)
    {
        Console.WriteLine(++x);
        ThreadPool.QueueUserWorkItem(delegate {
            Console.WriteLine(++x);
        });
        Console.ReadLine();
    }
}
```

Prilikom poziva funkcija remote objekata .net framework thread zamijenjuje logičkim threadom. To znači da se thread koji poziva funkciju pauzira, parametri se pošalju serveru, na serveru se kreira novi thread, te se izvrši funkcija, uništava se novokreirani thread, povratne vrijednosti se šalju klijentu, te klijentski thread nastavlja sa radom. Međutim, statičke varijable i statičke varijable za pojedini thread se ne dijele između udaljenih računala. Statičkim varijablama udaljenog objekta ne možemo ni pristupiti (za nestatičke se ustvari kreiraju getteri i setteri).
Zbog toga je uveden objekt CallContext. On ima metode GetData i SetData, pomoću kojeg možemo napraviti analognu stvar. Sljedeći program radi slično kao i prethodni:

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

class Program
{
    static void Main(string[] args)
    {
        CallContext.SetData("x", 0);
        Console.WriteLine(CallContext.GetData("x"));
        CallContext.SetData("x", (int)CallContext.GetData("x") + 1);
        ThreadPool.QueueUserWorkItem(delegate {
            Object o = CallContext.GetData("x");
            if (o == null)
                CallContext.SetData("x", 0);
            Console.WriteLine(CallContext.GetData("x"));
            CallContext.SetData("x", (int)CallContext.GetData("x") + 1);
        });
```

```
            Console.ReadLine();
        }
    }
```

Razlika između statičkih varijabli i CallContext objekata je da CallContext objekti mogu slijediti logički thread, čak i ako logički thread nije isti fizičkom threadu. Elementi CallContexta se satoje od parova string/object i predstavljaju ime=vrijednost. Neće svi objekti putovati do udaljenog objekta, nego samo oni parovi čija vrijednost zadovoljava interface `ILogicalThreadAffinative` (i taj tip bi trebao imati atribut Seriazible, ili izveden iz `MarshalByRefObject`).

Primjer

Osim prilikom poziva udaljenih objekata, CallContext podaci koji su izvedeni iz `ILogicalThreadAffinative` mogu biti korisni prilikom asinhronih poziva funkcija (pokrenite sljedeći primjer kao što je navedeno i tako da klasa cl ne zadovoljava interface, pa komentirajte razlike):

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

class cl : ILogicalThreadAffinative
{
    public int x;
}
class Program
{
    delegate void del();
    static void Main(string[] args)
    {
        cl c = new cl();
        c.x=0;
        CallContext.SetData("x", c);
        Console.WriteLine(((cl)CallContext.GetData("x")).x);
        c.x++;
        //CallContext.SetData("x", (int)CallContext.GetData("x").x + 1);
        ThreadPool.QueueUserWorkItem(delegate {
            Console.WriteLine(((cl)CallContext.GetData("x")).x);
        });
/*        del d = delegate
        {
            Object o = CallContext.GetData("x");
            if (o == null)
                CallContext.SetData("x", 0);
            Console.WriteLine(((cl)CallContext.GetData("x")).x);
        };
        d.BeginInvoke(null, null);*/
        Console.ReadLine();
    }
}
```

Ukoliko nam neki dodatni podaci trebaju samo za jedan funkcijski poziv, možemo koristiti CallContext Headers element. Takvi elementi ne moraju biti izvedeni iz `ILogicalThreadAffinative`.

```
        CallContext.SetHeaders(new Header[] { new Header("ime",
"vrijednost"), new Header("ime1", "vrijednost1") });

...
```

```
        Header[] h = CallContext.GetHeaders();
        string ime = h[0].Value;
```

9. Poglavlje: **Windows Service**

Za serverske programe nije uvijek potrebno da imaju konzolu/prozor. Moguće je (a često i bolje) imati u pozadini program (pa nema opasnosti da ćemo ga zabunom isključiti). Windows Servisi su napravljeni upravo u tu svrhu. Mogu se automatski paliti kada se windowsi startaju (prije logiranja korisnika) i mogu biti aktivni sve do gašenja Windowsa (i ako se svi programi pogase, i korisnik odlogira…).
VS ima vrlo jednostavnu mogućnost kreiranja Windows Servisa. Kreiramo novi projekt, odaberemo Windows Service, te preradimo metodu OnStart (u koju dodamo kôd za otvaranje kanala/registraciju objekata…). I još sve što treba napraviti je dodati installer, što VS opet ima vrlo jednostavan način kako se to radi.
Klijente pravimo potpuno jednako kao i prije. Kada dodamo referencu na klijenta, automatski se napravi kod za kreiranje potrebnih proxy-a i kreiranje objekta.
Opišimo postupak kreiranje servisa od početka. Recimo da želimo napraviti Chat server. Kreiramo novi C# Projekt->Windows service. U property prozoru (Design viewa datoteke Service1.cs) promijenimo ime servisa (ServiceName i (Name)) iz Service1 u npr. ChatService. Otvorimo kôd samog servisa (Service1.cs), te u OnStart obacimo kôd iz servera (zbog jednostavnosti su tu ubačene i klase koje će se koristiti).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.ServiceProcess;
using System.Text;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace Chat
{
    public partial class ChatService : ServiceBase
    {
        public ChatService()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            BinaryServerFormatterSinkProvider provider = new
BinaryServerFormatterSinkProvider();
            provider.TypeFilterLevel =
System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
            IDictionary props = new Hashtable();
            props["port"] = 12345;
            TcpChannel chan = new TcpChannel(props, null, provider);
            ChannelServices.RegisterChannel(chan);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(ChatServer),
"Chat.ChatServer", WellKnownObjectMode.Singleton);
```

```
        }

        protected override void OnStop()
        {
            // TODO: Add code here to perform any tear-down necessary to
stop your service.
        }
    }
    public class ChatClient : MarshalByRefObject
    {
        public void Reci(string s)
        {
            Console.WriteLine(s);
        }
    }
    public class ChatServer : MarshalByRefObject
    {
        List<ChatClient> lc = new List<ChatClient>();
        public void DodajKlijenta(ChatClient c)
        {
            lc.Add(c);
        }
        public void ReciSvima(ChatClient c, string s)
        {
            for (int i = 0; i < lc.Count; ++i)
            {
                try
                {
                    if (lc[i] != c) lc[i].Reci(s);
                }
                catch
                {
                    lc.RemoveAt(i); --i;
                }
            }
        }
    }
}
```

Budući da WS nema ni prozora ni konzole, ukoliko želimo neke informacije poslati korisniku, možemo to pisati u Windows event log. System.Diagnostics.EventLog klasa ima podršku za to (npr. funkciju WriteEntry(ime programa, poruka)).
Još je potrebno dodati instalaciju. Vratimo se na Design view datoteke Service1.cs, te u Propertyima kliknemo Add Installer.
U propertyima serviceInstaller1 objekta možemo podesiti dodatne informacije, kao npr. opis, da li želimo da se automatski starta (pod defaultu je manual, ali nejčešće se koristi automatic).
Kliknemo na serviceProcessInstaller1 i u propertyima podesimo account pod kojim želimo da se izvršava (najčešće je LocalSystem).
Sada možemo kompajlirati program. Nakon što je program kompajliran, sa installutil chat.exe (iz komandne linije, sa administratorskim ovlastima) instaliramo servis.
Ako je Servis bio podešen na Automatic startup, možemo ga koristiti, a ako ne, trebamo otići u
Control Pannel->Administrative Tools->Services, tamo ga pronaći, kliknit desnim gumbom i Start
Klijentski kôd pišemo potpuno jednako kao i ranije (u reference trebamo dodati i System.Runtime.Remoting i .exe servisa):

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using Chat;

namespace ChatCli
{
    class Program
    {
        static void Main(string[] args)
        {
            ChatClient ja = new ChatClient();

            TcpChannel chan = new TcpChannel(0);
            ChannelServices.RegisterChannel(chan);

            ChatServer server =
(ChatServer)Activator.GetObject(typeof(ChatServer),
@"tcp://localhost:12345/Chat.ChatServer");
            if (server == null)
            {
                System.Console.WriteLine("Nema servera");
                return;
            }
            server.DodajKlijenta(ja);
            while (true)
            {
                string s = Console.ReadLine();
                if (s == "") break;
                server.ReciSvima(ja, s);
            }

        }
    }
}
```

Ukoliko želimo ispraviti neku grešku na serveru, server prvo treba deinstalirati (isto kao i instalirati samo koristimo /u komandnu opciju), pa izmijeniti pogreške, kompajlirati ga i ponovo instalirati.

Debuggiranje Windows Servisa

Ukoliko želimo debugirati windows servis, potrebno je ići na Debug->Attach to process… pa označiti i sistemske procese (i/ili procese od svih korisnika i svih sessiona), tamo pronaći Chat.exe, te kliknuti na njega. Za naše potrebe je dovoljno debuggirati CLR kod, te ako imamo administratorske ovlasti (ili ovlasti pod kojima se on izvršava), moći ćemo ući u njegovo normalno debuggiranje

IIS servisi

Osim Windows serisa, remote objekte može posluživati i IIS, korištenjem Internet Services Managera i ispravno podešenih konfiguracijskih datoteka, ali o tome nećemo, jer i za njih klijent mora biti .net framework. Zbog toga na web xml servisi puno više mogućnosti daju, pa ćemo njih malo detaljnije proučiti u sljedećem poglavlju.

10. Poglavlje: **XML Web Services**

Do sada smo se susretali gotovo isključivo s distribuiranim aplikacijama gdje su i klijent i server morali biti pisani u .net okruženju. Komunikaciji između COM objekata i .net-a je moguća (što smo vidjeli na primjeru podizanja Worda, a moguće je napraviti i obratnu stvar – podići .net klasu u npr. Wordu). Ali sve te tehnologije su pod Microsoftom, i budući da je (D)COM bio preteća .net remotinga, to i nije čudno.
Korak dalje u povezivanju aplikacija pružaju Web Service. To su serveri koji se ponašaju slično kao i serveri koje smo do sada spominjali, ali se izvršavaju pod IIS-om (Microsoftova verzija servera za hosting WEBa), te je svaki poziv funkcije i njen odgovor ustvari XML datoteka, pa klijenti mogu raditi pod svakakvim sustavima koji znaju (a danas to znaju skoro svi, od mobitela, do računala pod svakakvim operativnim sustavima). Visual Studio će nam automatski generirati datoteku koju dodamo u reference, ako klijenta želimo pisati u .net remotingu (pa ćemo s njim raditi potpuno jednako kao i sa dosadašnjim klijentima, .net framework će se pobrinuti da kreira ispravne proxy objekte), ali i sve ostale mogućnosti su otvorene.
Iako web servise možemo napisati u najobičnijem tekst-editoru, pa ručno podesiti Virtualne direktorije IIS-a, (IIS će ih automatski kompajlirati prilikom prve uporabe), puno ih je jednostavnije kreirati koristeći VS.

Kreiranje web servisa u VS-u

Potrebno je kreirati novi projekt->Visual C#->ASP.NET Web Service ili New Web site->ASP.NET Web Service. VS će nam tada automatski kreirati Virtualni folder i podesiti sve datoteke da projekt uredno radi.
Po defaultu će nam kreirati npr. datoteku Service1.asmx (i Service1.asmx.cs) što će biti naziv servisa. Ime možemo promijeniti jednostavnim renameom u Solution Exploreru, npr. možemo promijeniti u MojServis (automatski su obje datoteke promijenjene, ali sam naziv klase nije, pa njega ručno promijenimo u kôdu i to i prvoj liniji u .asmx datoteke, i u .asmx.cs datoteci).
Dijelove kôda servisa možemo slobodno izbaciti ako želimo, a slobodno možemo i dodavati funkcije. Međutim, samo one funkcije koje su označene atributom [WebMethod] će biti vidljive klijentima. I to je sve što je potrebno za sasvim funkcionalan Web servis. Pokretanjem programa, VS automatski starta Internet explorer koji pokreće zadani servis.
Ukoliko servis pozovemo bez parametara, IIS će automatski odgovoriti HTML datotekom u kojoj možemo klikanjem pozivati funkcije servisa (i između ostalog promatranjem sourcea HTML datoteke vidjeti kako pozivati metode s drugih platformi, iako ima više načina kako to dobiti direktno, npr. ?wsdl direktivom u IE, ili dodavanjem web reference u VS-u).

Stanje servisa

Web servisi sami po sebi ne mogu pamtiti svoje stanje odnosno varijable (prilikom svakog poziva se kreira novi objekt). Ali stanje možemo pamtiti na više načina.
Sesstion objekti žive dok im ne istekne Session Time (kojeg možemo najjednostavnije podesiti u Web.config datoteci u <system.web> tagu sa `<sessionState timeout ="1"/>` gdje broj označava broj minuta).
Statički objekti su zajednički za sve instance objekta (i življet će dokle god je bar jedan Session aktivan, a možda i malo duže, ovisno o zauzetosti servera), a Session objekti (ukoliko je to uključeno), samo za objekte u jednom sessionu (što bi značilo u jednom Exploreru).
koristeći Session objekta (slično kao i kod običnih .aspx web stranica).

Ukoliko želimo koristiti Session objekte, metoda mora imati atribut [WebMethod(EnableSession = true)], te nakon toga u takvim metodama možemo koristiti objekt Session[ime_objekta] i postavljati mu ili očitavati vrijednost. On dakle ima uređene parove string/object, gdje prvi element predstavlja ime objekta, a drugi njegovu vrijednost. Budući da je vrijednost tipa object, u nju možemo učahuriti bilo koji tip.
Po defaultu ASP.NET koristi cookie da pamti session kojem pripada, pa se podrazumijeva da klijent znam prihvatiti cookie i poslati ih nazat u sljedećem requestu. To će se po defaultu automstski dogoditi u većini web browsera i ako koristimo VS. Ako to nije slučaj, ručno moramo dodati podršku za cookie (što opet u većini programskih jezika možemo u nekoliko linija kôda, jer najčešće postoje gotove klase za to).
Ukoliko želimo imati neki objekt zajednički svim Sessionima, možemo koristiti Application objekt (on je isto mapa string=boject), i njegov životni vijek je kao i kod statičkih objekata.
Ukoliko iz nekog razloga našu klasu ne možemo izvesti iz WebService klase, tada se Session objekt zove HttpContext.Current.Session
Ukoliko želimo da naša aplikacija bude postavljena negdje na Internet (ili na localhost), potrebno je ići na Build->Publish project (za localhost VS treba biti startan sa Administratorskim accountom, a za publishiranje negdje na Internet, najčešće se koriste FrontPage ekstenzije).

Kreiranje klijenta

Klijentska aplikacija može biti bilo kojeg tipa, radi jednostavnosti, odaberimo Console Application. Potrebno je dodati Web referencu (pod pretpostavkom da smo Servis publicirali na localhost, Prilikom publiciranja namespace od Servisa je zamijenjeno nazivom servera). Namespace pod kojim je on na klijentu je namespaceKlijenta.nazivServera, te ga je potrebno uključiti (npr. ovjde bio bio ConsoleApplication1.localhost).
Nakon toga možemo bez problema kreirati objekt tipa MojServis, te pozivati njegove metode. Osim normalnih metoda, automatski su dodane metode za asinhrono pozivanje BeginHelloWorld i EndHelloWorld. Međutim callback od Begin… metode nemožemo koristiti da automatski pozovemo End…, jer parametar callback metode nije izveden iz AsyncResult, nego iz WebClientAsyncResult.
Ukoliko želimo pamtiti stanje objekta, moramo aktivirati cookije, a to možemo sa npr.

```
MojServis m = new MojServis();
m.CookieContainer = new System.Net.CookieContainer();
```

Slanje/vraćanje parametara

U Web servisima, podaci se uvijek šalju/vraćaju po vrijednost, čak i kad su izvedeni iz MBR objekta. Pa ipak i kompliciranije objekte (koje smo sami napravili) možemo slati kao parametar ili primiti kao povratnu vrijednost. Kod web servisa, objekti ne moraju biti označeni kao Seriazable, i web servisi ne koriste ni Soap ni binarni formatter, nego koriste XmlSerializer iz System.Xml.Serialization. On može serializirati samo javne dijelove klase, ne serializira tipske informacije i automatski radi. Ako ga ručno želimo koristiti, možemo koristeći atribute kao [XmlRoot(ime)], [XmlElement(ime)], [XmlAttribute()]. Prvi stavljamo ispred klase, a druga dva za proizvoljni element. Posljednji će značiti da se vrijednost prenosi kao dodatni atribut root-a, a za srednji će biti generiran posebni tag unutar root-a s tim elementom.