Arpad GELLERT

Algoritmi Paraleli şi Distribuiţi

Aplicaţii

Cuprins

1. Operaţii I/O 3
2. Fire de execuţie 7
3. Sincronizarea firelor de execuţie 14
4. Aplicaţii client-server 18
5. Conectarea la o bază de date MySQL 25
6. Aplicaţii web 27
7. Aplicaţii distribuite 32
8. Algoritmi paraleli 40
Bibliografie 45

1. Operaţii I/O

Un flux de intrare/ieşire (I/O) furnizează calea prin care programele pot trimite o secvenţă de octeţi de la o sursă către o destinaţie [7]. Un flux de intrare este o sursă (producător) de octeţi, iar un flux de ieşire este o destinaţie pentru octeţi (consumator). Deşi fluxurile I/O sunt deseori asociate cu accesul la fişierele de pe disc, sursa şi destinaţia unui program pot fi, de asemenea, tastatura, mouse-ul, memoria sau fereastra de afişare [9].

Citirea unui şir de la tastatură

Pentru citirea de la tastatură [7], pot fi folosite clasele DataInputStream sau BufferedReader. Cu DataInputStream citirea se face în felul următor:

```
DataInputStream dis  = new DataInputStream(System.in);
String str = null;
try{
   str = dis.readLine();
}
catch(IOException ioe){
   ioe.printStackTrace();
}
```

În următoarea secvenţă citirea de la tastatură a unui şir de caractere se efectuează folosind clasa BufferedReader:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str = null;
try{
   str = br.readLine();
}
catch(IOException ioe){
   ioe.printStackTrace();
}
```

Conversia unui şir de caractere *str* într-un întreg se poate realiza folosind funcţia statică *parseInt* a clasei *Integer*:

```
int i = Integer.parseInt(str);
```

Pentru conversia la float sau double se pot folosi:

```
float f = Float.parseFloat(str);
double d = Double.parseDouble(str);
```

Pentru evitarea excepţiilor, în cazul conversiilor trebuie tratată excepţia Number*FormatException*.

Citirea unui şir de caractere de la tastatură în C# se face cu funcţia *readLine*:

```
string str = Console.ReadLine();
```

Conversiile pot fi efectuate prin funcţiile clasei *Convert*. Exemple:

```
int i = Convert.ToInt32(str);
double d = Convert.ToDouble(str);
```

Pentru evitarea excepţiilor, în cazul conversiilor trebuie tratată excepţia *FormatException*.

Citirea dintr-un fişier

În exemplul următor, se afişează pe ecran toate liniile citite din fişierul *input.txt*.

```
 FileInputStream fis = null;
try{
  fis = new FileInputStream("input.txt");
}
catch(FileNotFoundException fnfe){
  fnfe.printStackTrace();
}
DataInputStream dis  = new DataInputStream(fis);
String str = null;
try{
  while((str = dis.readLine()) != null)
    System.out.println(str);
  dis.close();
  System.in.read();
}
catch(IOException ioe){
  ioe.printStackTrace();
}
```

```
StreamReader sr = new StreamReader("input.txt");
string line = null;
while ((line = sr.ReadLine()) != null)
   Console.WriteLine(line);
Console.ReadLine();
sr.Close();
```

Procesul de separare în simboluri

Biblioteca Java *util* defineşte clasa *StringTokenizer* care facilitează separarea unui şir de caractere în simboluri. Trebuie creat un obiect *StringTokenizer*, specificând şirul şi delimitatorul. Următoarea secvenţă de program afişează pe ecran simbolurile şirului delimitate prin caracterul spaţiu.

```
 String psd = "Programarea Sistemelor Distribuite";
StringTokenizer st = new StringTokenizer(psd, " ");
while(st.hasMoreTokens())
   System.out.println(st.nextToken());
```

O altă posibilitate de separare a șirurilor de caractere în simboluri în limbajul Java este funcția *split* a clasei *String*. Următorul exemplu extrage dintr-o variabilă de tip *String* cuvintele delimitate prin spații:

```
String str = “Hello world”;
String t[] = str.split(“ ”);                    //Rezultat: {“Hello”, “world”};
```

În C# un şir poate fi separat în simboluri folosind funcţia *Split* a clasei *string*. Următorul exemplu procesează un fişier întreg:

```
StreamReader sr = new StreamReader("input.txt");
string file = sr.ReadToEnd();
char [] separators = {'\n', ' '};
string [] values = file.Split(separators);
foreach (string v in values)
   Console.WriteLine(v);
Console.ReadLine();
sr.Close();
```

Scrierea într-un fişier

Următoarea secvenţă de program scrie în fişierul data.txt întregul 10 şi float-ul 3.14, apoi citeşte aceste valori din fişier şi le afişează pe ecran.

```
try{
  FileOutputStream fos = new FileOutputStream("data.txt");
  DataOutputStream dos = new DataOutputStream(fos);
  dos.writeInt(10);
  dos.writeFloat(3.14f);
  dos.close();
}
catch(IOException ioe){
  ioe.printStackTrace();
}
try{
  FileInputStream fis = new FileInputStream("data.txt");
  DataInputStream dis = new DataInputStream(fis);
  System.out.println(dis.readInt());
  System.out.println(dis.readFloat());
  dis.close();
}
catch(IOException ioe){
  ioe.printStackTrace();
}
```

Următoarea secvenţă de program scrie în fişierul data.txt şirul de caractere “Hello World!”, apoi îl citeşte din fişier şi îl afişează pe ecran.

```
StreamWriter sw = new StreamWriter("data.txt");
sw.WriteLine("Hellow World!");
sw.Close();
StreamReader sr = new StreamReader("data.txt");
Console.WriteLine(sr.ReadLine());
Console.ReadLine();
sr.Close();
```

Aplicaţii

1. a) Citirea unui întreg de la tastatură;
   b) Afişarea pe ecran a întregului citit;
2. a) Citirea unui float/double de la tastatură;
   b) Să se rotunjească în sus (Math.ceil()) valoarea citită şi să se afişeze rezultatul pe ecran;
   c) Să se rotunjească în jos (Math.floor()) valoarea citită şi să se afişeze rezultatul pe ecran;
   d) Să se rotunjească la întregul cel mai apropiat (Math.round()) şi să se afişeze rezultatul;
3. Să se citească dintr-un fişier un şir de întregi şi să se afişeze pe ecran media aritmetică;

2. Fire de execuţie

Un fir de execuţie (thread) este o secvenţă de instrucţiuni ale unui program executabil. Firul de execuţie principal este metoda *main* şi atunci când acesta se termină, se încheie şi programul. Programele pot executa în paralel două sau mai multe fire de execuţie, dar în realitate un singur fir se execută la un moment dat, comutând controlul de la un thread la altul [9].

Pentru implementarea unui fir de execuţie în Java, se poate extinde clasa *Thread*. Deoarece Java nu acceptă moştenirea multiplă, în cazul în care a fost deja extinsă o clasă, pentru crearea unui fir de execuţie trebuie implementată interfaţa *Runnable*. Indiferent de metoda utilizată, se suprascrie metoda *run* care trebuie să conţină instrucţiunile firului.

Aplicaţia următoare porneşte două fire de execuţie: unul pentru afişarea numerelor şi celălalt pentru afişarea literelor. Pentru a observa diferenţele dintre cele două metode de implementare, firul de execuţie *Numbers* extinde clasa *Thread*, în timp ce *Letters* implementează interfaţa *Runnable*.

```
public class Numbers extends Thread {
  public void run(){
    for(int i=0; i<1000; i++){
      System.out.println(i);
    }
  }
}

public class Letters implements Runnable {
  char a = 'a';
  public void run(){
    for(int i=0; i<1000; i++){
      int c = a + i%26;
      System.out.println((char)c);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    Numbers numbers = new Numbers();
    Thread  letters = new Thread(new Letters());
    letters.start();
    numbers.start();
  }
}
```

Pentru setarea priorităţii unui fir se apelează metoda *setPriority* a clasei *Thread*. De asemenea, pentru a afla prioritatea unui fir se foloseşte metoda *getPriority*. Pentru suspendarea unui fir de execuţie pentru o perioada de timp precizată în milisecunde, se utilizează metoda *sleep* a clasei *Thread*.

Pentru implementarea unui fir de execuţie în C# se foloseşte clasa *Thread*. Următoarea aplicaţie de tip *Console Application* porneşte două fire de execuţie: unul pentru afişarea numerelor prin funcţia *numbers* şi celălalt pentru afişarea literelor prin funţia *letters*.

```
public class Fire
{
    public Fire()
    {
        Thread numThread = new Thread(new ThreadStart(numbers));
        Thread letThread = new Thread(new ThreadStart(letters));
        letThread.Start();
        numThread.Start();
    }

    public void numbers(){
        for(int i=0; i<1000; i++)
            Console.WriteLine(i);
    }

    public void letters(){
        char a = 'a';
        for(int i=0; i<1000; i++){
            int c = a + i%26;
            Console.WriteLine((char)c);
        }
    }

    static void Main(string[] args)
    {
        new Fire();
        Console.ReadLine();
    }
}
```

Pentru blocarea unui fir de execuţie pentru o perioada de timp precizată în milisecunde, se utilizează metoda *Sleep* a clasei *Thread*. Firul de execuţie poate fi oprit temporar cu funcţia *Suspend*, repornit cu funcţia *Resume*, respectiv oprit definitiv cu funcţia *Abort* a clasei *Thread*.

Lucrul cu imagini

Pentru a trasa o imagine în cadrul unei componente (ex. fereastră), trebuie obţinut obiectul *Graphics* al componentei respective [4]. În aplicaţia următoare s-a implementat o animaţie simplă: deplasarea unei bile pe orizontală. Deplasarea bilei se face pe fir de execuţie.

Aplicaţia Java este formată din două clase: Ball şi MyFrame. Înainte de a muta bila pe noua poziţie, ea trebuie ştearsă de pe poziţia veche, şi de aceea, pentru un scurt timp bila dispare de pe fereastră. Din această cauză apare o pâlpâire în timpul deplasării. Pentru eliminarea acestei pâlpâiri se poate implementa tehnica *double buffering*: desenarea se face mai întâi într-un buffer,

şi apoi se copiază conţinutul bufferului pe fereastra aplicaţiei [4]. Clasa Ball, instanţiată în clasa MyFrame, are următoarea structură:

```
public class Ball extends Thread {
  int px = 0;
  int py = 0;
  int size = 0;
  Color color = null;
  MyFrame parent = null;
  Image buffer = null;

  public Ball(MyFrame parent, int px, int py, int size, Color color) {
    this.parent = parent;
    this.px = px;
    this.py = py;
    this.size = size;
    this.color = color;
    buffer = parent.createImage(parent.getSize().width, parent.getSize().height);
  }

  void paint(){
    Graphics gbuffer = buffer.getGraphics();
    //se deseneaza mai intai in buffer (tehnica Double Buffering)
    gbuffer.setColor(Color.white);
    gbuffer.fillRect(0, 0, parent.getSize().width, parent.getSize().height);
    gbuffer.setColor(color);
    gbuffer.fillOval(px, py, size, size);
    parent.paint(gbuffer);
    //se copiaza imaginea din buffer pe fereastra (tehnica Double Buffering)
    Graphics g = parent.getGraphics();
    g.drawImage(buffer, 0, 0, parent.getSize().width, parent.getSize().height, 0, 0, parent.getSize().width,
          parent.getSize().height, parent);
  }

  public void run(){
    while(px < parent.getSize().width){
      px++;
      paint();
    }
  }
}
```

Clasa MyFrame are următoarea structură:

```
public class MyFrame extends Frame {
  Ball ball = null;

  public MyFrame() {
    try  {
      jbInit();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    setVisible(true);
    ball = new Ball(this, 20, 50, 20, Color.red);
    ball.start();
  }
```

```
  private void jbInit() throws Exception  {
    this.setSize(new Dimension(400, 300));
    this.setTitle("Balls");
    this.addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        this_windowClosing(e);
      }
    });
  }

  void this_windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

Aplicaţia C# de tip *Windows Application* este formată din două clase: Ball şi MyForm. Clasa Ball, instanţiată în clasa MyForm, are următoarea structură:

```
public class Ball
{
    int px = 0;
    int py = 0;
    int size = 0;
    Color color;
    MyForm parent = null;
    Thread bthread = null;

    public Ball(MyForm parent, int px, int py, int size, Color color)
    {
        this.parent = parent;
        this.px = px;
        this.py = py;
        this.size = size;
        this.color = color;
        bthread = new Thread(new ThreadStart(run));
        bthread.Start();
    }

    public int getPX()
    {
        return px;
    }

    public int getPY()
    {
        return py;
    }

    public int getSize()
    {
        return size;
    }

    public Color getColor()
    {
        return color;
    }
```

```
    public void terminateBallThread()
    {
        bthread.Abort();
    }

    public void run()
    {
        while (px < parent.Size.Width)
        {
            Thread.Sleep(20);
            px++;
            parent.Refresh();
        }
    }
}
```

Clasa MyForm are următoarea structură:

```
public partial class MyForm : Form
{
    Ball ball = null;

    public MyForm()
    {
        InitializeComponent();
        CheckForIllegalCrossThreadCalls = false;
        this.BackColor = Color.White;
        ball = new Ball(this, 0, 50, 20, Color.Red);
    }

    private void MyForm_Load(object sender, EventArgs e)
    {
        this.Size = new Size(400, 300);
        this.Name = "Balls";
    }

    private void MyForm_Paint(object sender, PaintEventArgs e)
    {
        SolidBrush brush = new SolidBrush(Color.White);
        e.Graphics.FillRectangle(brush, 0, 0, Size.Width, Size.Height);
        brush = new SolidBrush(ball.getColor());
        e.Graphics.FillEllipse(brush, ball.getPX(), ball.getPY(), ball.getSize(), ball.getSize());
        brush.Dispose();
    }

    private void MyForm_FormClosing(object sender, FormClosingEventArgs e)
    {
        ball.terminateBallThread();
    }
}
```

Atenţie, fişierul MyForm.cs se va genera automat, urmând să-l completaţi conform clasei MyForm de mai sus. Evenimentelor Load, Paint şi FormClosing trebuie asociate funcţiile corespunzătoare (MyForm_Load, MyForm_Paint şi MyForm_FormClosing) în Design/Properties/Events!

În aplicaţia C# tehnica *double buffering* se activează setând pe *True* proprietatea *DoubleBuffered* în Design/Properties.

Aplicaţii

1. Să se dezvolte clasa *Ball* din aplicaţia prezentată pentru modelarea mişcării unei bile folosind următorul algoritm:

```
gravy = 1;
speed = -30;
speedy = -30;
speedx = 0;
while(true){
        speedy += gravy;
        py += speedy;
        px += speedx;
        paint();
        if(py > frameheight){
                speedy = speed;
                speed += 3;
        }
        if(speed == 0) break;
}
```

2. Să se modifice clasele MyFrame şi Ball în aşa fel încât să poată fi pornite mai multe bile simultan.

3. Să se realizeze o aplicaţie care să afişeze reclamele, câte 5 secunde fiecare, într-o buclă infinită. Fiecare reclamă are o anumită prioritate şi în funcţie de aceasta, ea apare cu o anumită frecvenţă. Căile imaginilor şi priorităţile acestora vor fi preluate dintr-un fişier.

Indicaţii pentru aplicaţia 3

Pentru încărcarea unei imagini în Java se foloseşte metoda *getImage* a clasei *Toolkit*. Una din metodele de a obţine obiectul *Toolkit* este prin folosirea metodei statice *getDefaultToolkit* a clasei *Toolkit*. Pentru preîncărcarea imaginilor în memorie, se foloseşte clasa *MediaTracker*. Metoda *waitForId* a clasei *MediaTracker* permite încărcarea imaginilor care au un anumit ID (mai multe imagini pot avea acelaşi ID). ID-urile se setează la adaugarea imaginilor în obiectul *MediaTracker*. Metoda *waitForAll* încarcă în memorie toate imaginile adăugate în obiectul *MediaTracker*. Dupa ce imaginile au fost încărcate ele pot fi desenate pe orice componentă folosind metoda *drawImage* a clasei Graphics. Următoarea secvenţă de program încarcă două imagini:

```
Image imgA = null;
Image imgB = null;
```

```
MediaTracker mt = new MediaTracker(this);  // this - fereastra
imgA = Toolkit.getDefaultToolkit().getImage(“A.gif”);
mt.addImage(imgA, 0);
imgB = Toolkit.getDefaultToolkit().getImage(“B.gif”);
mt.addImage(imgB, 0);

try{
      mt.waitForAll();
}
      catch(InterruptedException e)
{
      e.printStackTrace();
}
```

Pentru încărcarea unei imagini în C# se poate folosi clasa *Bitmap*, iar desenarea imaginii pe fereastră poate fi efectuată cu funcţia *DrawImage* a clasei *Graphics*. Pentru asta, trebuie obţinut obiectul *Graphics* al ferestrei sau al componentei pe care se doreşte să se afişeze imaginea:

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
      Bitmap bmp = new Bitmap("c:\\t.bmp");
      e.Graphics.DrawImage(bmp, 0, 0, new Rectangle(0, 0, Size.Width, Size.Height),
            GraphicsUnit.Pixel);
}
```

3. Sincronizarea firelor de execuţie

Există mai multe metode de sincronizare a firelor de execuţie [2], cea mai frecvent utilizată fiind sincronizarea prin *semnalizare*. În continuare, va fi prezentată o aplicaţie simplă, rezolvarea ecuaţiei de gradul 2.

Dacă în cadrul unui program Java există un fir de execuţie care creează (produce) date şi un al doilea fir de execuţie care le prelucrează (consumă), de regulă se declară un bloc *synchronized*, ceea ce permite ca un singur fir să aibă acces la resurse (metode, date) la un moment dat. Astfel, sincronizarea se face prin aşteptare. Atunci când un fir de execuţie apelează *wait* în cadrul unui bloc de cod *synchronized*, alt fir poate accesa codul. Iar atunci când un fir de execuţie încheie procesarea codului *synchronized*, el apelează metoda *notify* pentru a anunţa alte fire de execuţie să înceteze aşteptarea. În cazul rezolvării ecuaţiei de gradul 2 – aplicaţie prezentată în continuare – firele *tx1* respectiv *tx2* aşteaptă notificarea firului *tdelta*:

```
public class Ec2{
  private Thread_X1 tx1;
  private Thread_X2 tx2;
  private Thread_Delta tdelta;
  int a = 1;
  int b = -4;
  int c = 1;
  double x1, x2;
  Double d = null;

  public Ec2() {
      tx1 = new Thread_X1(this);
      tx2 = new Thread_X2(this);
      tdelta = new Thread_Delta(this);

      tx1.start();
      tx2.start();
      tdelta.start();
  }

  public synchronized void delta(){
    d = new Double(b*b - 4*a*c);
    System.out.println("delta = " + d);
    System.out.println("tdelta: sleeping");
    try {
      Thread.sleep(5000);
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    this.notifyAll();
    System.out.println("tdelta: notified");
  }
```

```
  public synchronized void x1(){
    if(d == null){
      System.out.println("tx1: waiting...");
      try {
        this.wait();
      }
      catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
    x1 = (-b + Math.sqrt(d.doubleValue())) / (2 * a);
    System.out.println("x1 = " + x1);
  }

  public synchronized void x2(){
    if(d == null){
      System.out.println("tx2: waiting...");
      try {
        this.wait();
      }
      catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
    x2 = (-b - Math.sqrt(d.doubleValue())) / (2 * a);
    System.out.println("x2 = " + x2);
  }

  public static void main(String[] args) {
    new Ec2();
  }
}


public class Thread_Delta extends Thread{
  Ec2 ec2;

  public Thread_Delta(Ec2 ec2) {
    this.ec2 = ec2;
  }

  public void run(){
    ec2.delta();
  }
}


public class Thread_X1 extends Thread{
  Ec2 ec2;

  public Thread_X1(Ec2 ec2) {
    this.ec2 = ec2;
  }

  public void run(){
    ec2.x1();
  }
}
```

```
public class Thread_X2 extends Thread{
  Ec2 ec2;

  public Thread_X2(Ec2 ec2) {
    this.ec2 = ec2;
  }

  public void run(){
   ec2.x2();
  }
}
```

AutoResetEvent şi *ManualResetEvent* permit firelor de execuţie să comunice între ele prin "semnalizare". Un fir aşteaptă un "semnal" apelând *WaitOne* pe *ManualResetEvent* (sau *AutoResetEvent*). Dacă *ManualResetEvent* este în starea *nesetat*, firul se blochează şi aşteaptă până când acesta ajunge în starea *setat*, în urma apelului metodei *Set*. Astfel, dacă în cadrul unui program există un fir de execuţie care creează (produce) date şi un al doilea fir de execuţie care le prelucrează (consumă), firul *consumator* aşteaptă până când este activat de firul *producător*. În cazul rezolvării ecuaţiei de gradul 2 – aplicaţie prezentată în continuare – firele *tx1* respectiv *tx2* aşteaptă notificarea firului *tdelta*:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace ec2
{
   class Program
   {
      static void Main(string[] args)
      {
         new Ec2();
      }
   }

   public class Ec2
   {
      int a = 1;
      int b = -4;
      int c = 1;
      Double x1, x2, d;
      static ManualResetEvent deltaReady = new ManualResetEvent(false);

      public Ec2()
      {
         Thread tx1 = new Thread(new ThreadStart(ComputeX1));
         Thread tx2 = new Thread(new ThreadStart(ComputeX2));
         Thread tdelta = new Thread(new ThreadStart(ComputeDelta));
         tdelta.Start();
         tx1.Start();
         tx2.Start();
         Console.ReadLine();
      }
```

```
        public void ComputeDelta()
        {
            d = b * b - 4 * a * c;
            Console.WriteLine("delta = " + d);
            Console.WriteLine("tdelta: sleeping");
            Thread.Sleep(5000);
            Console.WriteLine("tdelta: ready");
            deltaReady.Set();
        }

        public void ComputeX1()
        {
            Console.WriteLine("tx1: waiting...");
            deltaReady.WaitOne();
            x1 = (-b + Math.Sqrt(d)) / (2 * a);
            Console.WriteLine("x1 = " + x1);
        }

        public void ComputeX2()
        {
            Console.WriteLine("tx2: waiting...");
            deltaReady.WaitOne();
            x2 = (-b - Math.Sqrt(d)) / (2 * a);
            Console.WriteLine("x2 = " + x2);
        }
    }
}
```

Aplicaţii

1. Să se implementeze aplicaţia prezentată în lucrare.
2. Să se implementeze cazul clasic de partajare a datelor: problema *producător/consumator*.
3. Să se implementeze o aplicaţie vizuală: sincronizarea liftului cu locatarii într-un bloc, sincronizarea călătorilor cu o linie de autobuz.

4. Aplicaţii client-server

Reţelele fac posibile multe tipuri noi de aplicaţii, deoarece nu mai este necesar ca un singur calculator să execute totul [4]. În cadrul unei reţele, câteva calculatoare, denumite servere, efectuează activităţi specializate în folosul altor programe. Serverul este un program specific care rulează în mod continuu, cu unicul scop de a furniza un serviciu altor programe. Clientul, pe de altă parte, este un program care primeşte servicii de la un server.

Pentru a scrie aplicaţii client-server [15], trebuie create două programe: clientul şi serverul. În cadrul acestor programe, trebuie definit modul cum comunică cele două programe, modul de interacţiune dintre ele. Regulile de comunicare pe care trebuie să le respecte ambele programe. Se numesc protocoale.

Atunci când două aplicaţii trebuie să comunice, ele trebuie să se găsească reciproc. Dacă două aplicaţii îşi cunosc reciproc soclul, ele pot crea o conexiune de soclu. De obicei este responsabilitatea clientului să caute serverul iniţializând o conexiune de socluri. Serverul creează un soclu pe care îl va utiliza pentru comunicarea cu clientul şi îşi transmite propria adresă de soclu către aplicaţia client în primul său mesaj de răspuns [15].

Pentru implementarea aplicaţiei client în Java, se foloseşte clasa *Socket* precizând IP-ul respectiv portul serverului. Pentru crearea unui server în Java, se foloseşte clasa *ServerSocket*, precizând portul serverului. Apoi serverul apelează metoda *accept* pentru a aştepta conectarea unui client. Atunci când un client lansează o cerere de conectare, metoda *accept* returnează soclul clientului (obiect prin care se va realiza comunicarea dintre cele două aplicaţii). În aplicaţia următoare, clientul trimite o valoare serverului, iar serverul primeşte valoarea şi o afişează. Aplicaţia se va testa pornind prima dată serverul şi apoi clientul.

```
public class MyClient {
        InputStream in = null;
        OutputStream out = null;
        DataInputStream DataIn = null;
        DataOutputStream DataOut = null;
        Socket clientSocket = null;

        public MyClient() {
                try {
                        clientSocket = new Socket("localhost", 8000);
                        sendInt(100);
                }
                catch( UnknownHostException e) {
                        e.printStackTrace();
                }
                catch(IOException e) {
                        e.printStackTrace();
                }
         }

        public void sendInt(int val) throws IOException{
                out = clientSocket.getOutputStream();
                DataOut = new DataOutputStream(out);
                DataOut.writeInt(val);
        }
```

```
	public void receiveInt() throws IOException{
		in = clientSocket.getInputStream();
		DataIn = new DataInputStream(in);
		int val = DataIn.readInt();
		System.out.println(val);
	}

	public static void main(String args[]){
		new MyClient();
	}
}


public class MyServer implements Runnable{
	private Socket socketReturn;
	InputStream in = null;
	OutputStream out = null;
	DataInputStream DataIn = null;
	DataOutputStream DataOut = null;

	public MyServer(Socket s)  throws IOException {
		socketReturn = s;
		receiveInt();  // Clientul incepe, deci serverul asteapta valoarea
	}

	public MyServer() {
		Thread thread = new Thread(this);
		thread.start();
	}

	public void sendInt(int val) throws IOException{
		out = socketReturn.getOutputStream();
		DataOut = new DataOutputStream(out);
		DataOut.writeInt(val);
	}

	public void receiveInt() throws IOException{
		in = socketReturn.getInputStream();
		DataIn = new DataInputStream(in);
		int val = DataIn.readInt();
		System.out.println(val);
	 }

	public void run(){
		ServerSocket s = new ServerSocket(8000);
		try {
			while(true){
				Socket socket = s.accept();
				try {
					new MyServer(socket);
				}
				catch(IOException e) {
					socket.close();
				}
			}
		}
		finally  {s.close();}
	}
```

```
    public static void main(String args[]){
        new MyServer();
    }

}
```

Pentru implementarea aplicaţiei client în C#, se foloseşte clasa *TcpClient* precizând IP-ul respectiv portul serverului. Pentru crearea unui server în C#, se foloseşte clasa *TcpListener*, precizând portul serverului. Apoi serverul apelează metoda *AcceptTcpClient* pentru a aştepta conectarea unui client. Atunci când un client lansează o cerere de conectare, metoda *AcceptTcpClient* returnează soclul clientului (un obiect de tip *TcpClient*), prin care se va realiza comunicarea dintre cele două aplicaţii. În aplicaţia următoare, clientul trimite o valoare serverului, iar serverul primeşte valoarea şi o afişează. Aplicaţia se va testa pornind prima dată serverul şi apoi clientul.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.IO;
using System.Threading;

namespace client
{
   class MyClient
   {
      TcpClient client = null;
      NetworkStream stream = null;
      StreamReader streamReader = null;
      StreamWriter streamWriter = null;

      public MyClient()
      {
         client = new TcpClient("localhost", 8000);
         stream = client.GetStream();
         streamReader = new StreamReader(stream);
         streamWriter = new StreamWriter(stream);
         sendInt(100);
         stream.Close();
         client.Close();
      }

      public void sendInt(int val)
      {
         streamWriter.WriteLine(Convert.ToString(val));
         streamWriter.Flush();
      }

      public void receiveInt()
      {
         int val = Convert.ToInt32(streamReader.ReadLine());
         Console.WriteLine(val);
      }

   }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.IO;
using System.Threading;

namespace server
{
   class MyServer
   {
      TcpListener server = null;
      NetworkStream stream = null;
      StreamReader streamReader = null;
      StreamWriter streamWriter = null;

      public MyServer(TcpClient clientSocket)
      {
         stream = clientSocket.GetStream();
         streamReader = new StreamReader(stream);
         streamWriter = new StreamWriter(stream);
         receiveInt();
         stream.Close();
      }

      public MyServer()
      {
         Thread thread = new Thread(new ThreadStart(run));
         thread.Start();

      }

      public void sendInt(int val)
      {
         streamWriter.WriteLine(Convert.ToString(val));
         streamWriter.Flush();
      }

      public void receiveInt()
      {
         int val = Convert.ToInt32(streamReader.ReadLine());
         Console.WriteLine(val);
      }

      void run()
      {
         server = new TcpListener(8000);
         server.Start();

         while (true)
         {
            TcpClient clientSocket = server.AcceptTcpClient();
            new MyServer(clientSocket);
         }
      }

   }
}
```

Aplicaţii

1. Să se implementeze aplicaţia prezentată în lucrare.

2. Să se modifice aplicaţia prezentată în lucrare astfel încât clientul să trimită serverului mesajul *Hello Server* iar serverul să raspundă *Hello Client*.

3. Realizarea unei aplicaţii client-server pentru listarea fişierelor text la imprimantă. Aplicaţia va avea următorul protocol de comunicare:

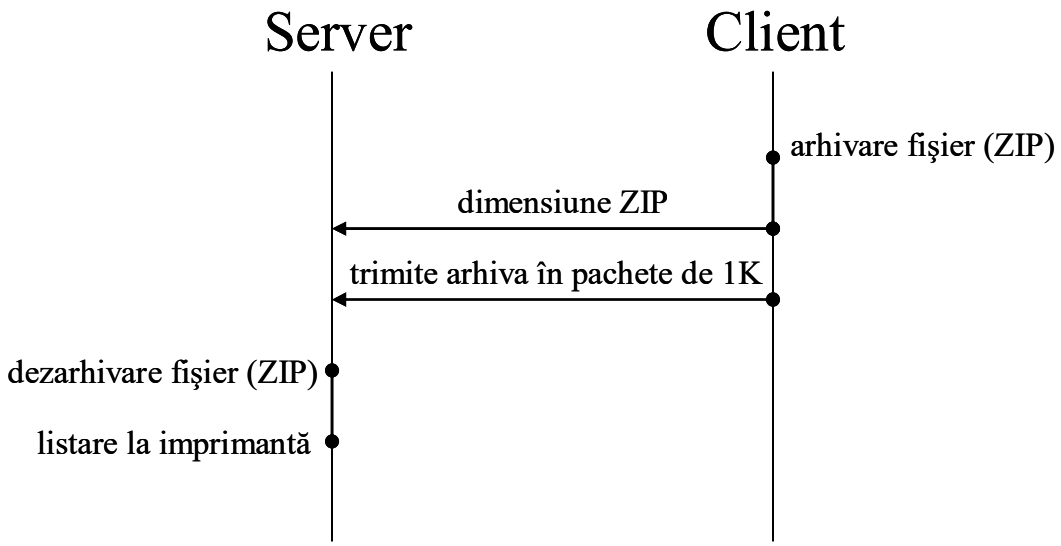

Figura 1. Transfer de fişiere

Indicaţii pentru aplicaţia 3

Aşa cum se poate observa în secvenţele următoare, arhivarea şi dezarhivarea se vor face folosind clasele ZipOutputStream şi respectiv ZipInputStream. Pentru listarea fişierului la imprimantă se poate folosi clasa PrintJob.

SERVER (dezarhivare)

```
try{
        FileInputStream fileIn = new FileInputStream(zip);
        FileOutputStream fileO = new FileOutputStream(numeFisier);
        ZipInputStream zipIn = new ZipInputStream(fileIn);
        zipIn.getNextEntry();
        int m;
        while((m = zipIn.read()) != -1)
                fileO.write(m);
        fileIn.close();
        fileO.close();
}
```

```
catch(ZipException e){
        System.out.println(e.toString());
}
```

CLIENT (arhivare)

```
try{
        FileInputStream fileIn = new FileInputStream(fisier);
        FileOutputStream f = new FileOutputStream(zip);
        ZipOutputStream zipOut = new ZipOutputStream(f);
        zipOut.putNextEntry(new ZipEntry(fisier));
        int m;
        while((m = fileIn.read()) != -1)
                zipOut.write(m);
        zipOut.close();
        fileIn.close();
        f.close();
}
catch(ZipException e){
        System.out.println(e.toString());
}
```

Listarea la imprimantă

```
PrintJob job=Toolkit.getDefaultToolkit().getPrintJob(this, " Print ",(Properties)null );
if (job==null) return;
Graphics pg = job.getGraphics();
pg.setFont(new Font("Times new roman",Font.BOLD,12));
pg.drawString("Programarea Sistemelor Distribuite", 100, 15);
job.end();
```

Pentru listarea la imprimantă în C#, există componente vizuale în *Toolbox*, în categoria *Printing*.

4. Să se modifice aplicaţia 3 în aşa fel încât ea să funcţioneze corect şi în situaţia în care anumite pachete de date se pierd şi nu ajung la server. Modificarea constă în retransmisia pachetelor pierdute.

5. Realizarea unei aplicaţii client-server pentru apelul funcţiilor la distanţă:
   - clientul cere apelul unei funcţii cu anumiţi parametri;
   - serverul apelează funcţia şi trimite rezultatul aplicaţiei client.

6. Să se dezvolte aplicaţia 5 prin introducerea unei strucuri de tip buffer, care să reţină rezultatele celor mai frecvente operaţii efectuate. În cazul in care rezultatul unei operaţii este găsit în acest buffer, operaţia respectivă nu se mai efectuează, rezultatul transmis aplicaţiei client fiind cel din buffer.

7. Să se implementeze o aplicaţie de chat.

Indicaţii pentru aplicaţia 7

Aplicaţia va fi formată din programul server respectiv programul client. Clientul va avea o interfaţă grafică (Figura 2) formată din următoarele componente: lista clienţilor conectaţi, lista de mesaje, căsuţă de editare pentru introducerea mesajelor, buton SEND pentru trimiterea mesajului introdus către clientul selectat. Recepţionarea mesajelor se va face într-o buclă infinită implementată pe un fir de execuţie.

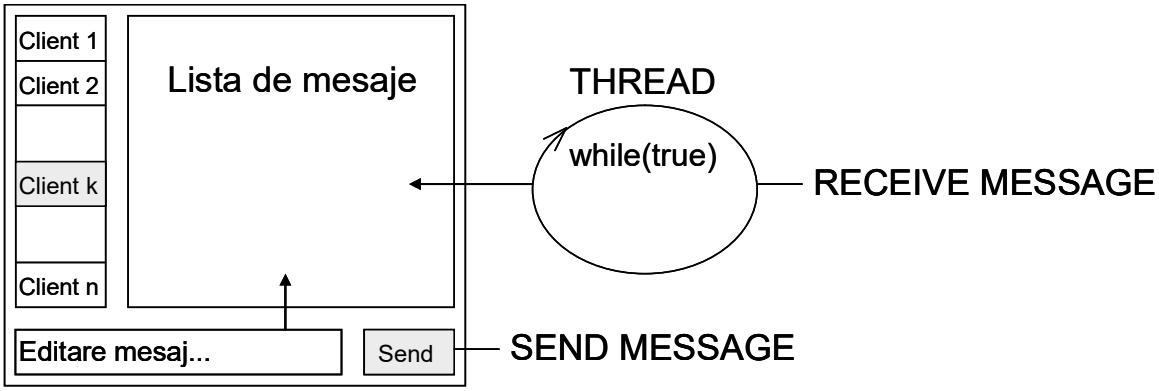

Figura 2. Aplicaţia client

Aplicaţia server, prezentată în Figura 3, păstrează clienţii (soclu, nume, etc.) într-un vector. Pentru fiecare client conectat creează câte un fir de execuţie (thread) care, într-o buclă infinită, recepţionează mesajele de la clientul respectiv şi apoi le trimite mai departe clientului destinaţie. Conectarea clienţilor se face la fel ca în aplicaţia prezentată în lucrare. La conectarea unui nou client serverul trebuie să anunţe toţi clienţii pentru actualizarea listelor de clienţi. De aceea, în afară de mesajele propriu-zise vor exista şi mesaje speciale (de administrare) pentru adăugarea clienţilor conectaţi respectiv ştergerea clienţilor deconectaţi. Diferenţierea dintre cele două tipuri de mesaje se poate efectua prin utilizarea unui antet (header) în cadrul mesajului.

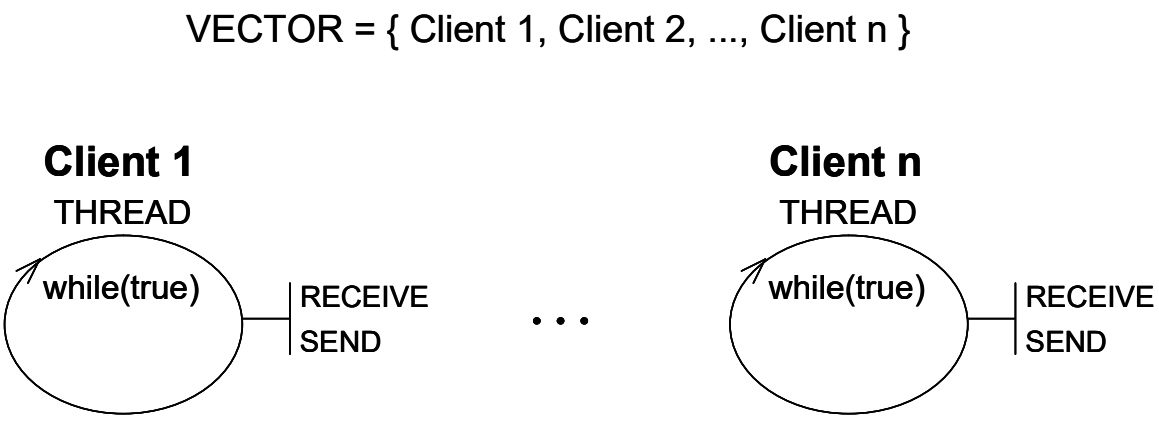

Figura 3. Aplicaţia server

5. Conectarea la o bază de date MySQL

JDBC (Java Database Connectivity) este o interfaţă API (Application Program Interface) realizată de Sun, care permite unui program Java accesul la un SGBD (Sistem de Gestionare a Bazelor de Date). JDBC permite conectarea la baza de date şi execută instrucţiunile SQL.

Proiectul conţine două clase principale: DBConnection şi DBFrame. În constructorul clasei DBConnection se încarcă în memorie driverul necesar pentru realizarea unei conexiuni la o bază de date MySQL:

```
Class.forName(driver);
```

Apoi are loc conectarea la baza de date *database*:

```
con = DriverManager.getConnection("jdbc:mysql://IP/database/", "user", "password");
```

unde IP este adresa IP a calculatorului pe care se află baza de date *database* şi pe care trebuie să fie pornit un server de MySQL. Pentru testare locală, în loc de IP se foloseşte *localhost*. În cazul unei baze de date locale adresa IP nu trebuie precizată:

```
con = DriverManager.getConnection("jdbc:mysql:database", "user", "password");
```

Clasa DBConnection poate fi definită în felul următor:

```
import java.sql.*;
public class DBConnection {
        private String driver = "org.gjt.mm.mysql.Driver";
        public Connection con = null;

        public DBConnection(String database, String user, String password) throws Exception{
                Class.forName(driver);
                con = DriverManager.getConnection("jdbc:mysql://localhost/" + database + "/", user, passw);
        }
}
```

DBConnection va conţine şi metodele prin care se vor insera date în tabele sau metodele prin care se vor interoga aceste tabele. Toate aceste operaţii se vor realiza prin intermediul unui obiect *Statement*:

```
Statement statement = con.createStatement();
```

Inserarea se face cu metoda *executeUpdate* a clasei *Statement*. De exemplu, pentru adăugarea înregistrării *Popescu* în tabela *student*, care are un singur câmp *Nume*, se procedează în felul următor:

```
statement.executeUpdate("insert into student (Nume) values ('Popescu')");
```

Aşa cum se poate observa în următoarea secvenţă, obţinerea rezultatelor unei interogări se realizează prin intermediul unui obiect de tip *ResultSet*:

```
ResultSet rs = statement.executeQuery("select * from student");
while(rs.next()){
      System.out.println(rs.getString("Nume"));
}
```

În constructorul clasei *DBFrame* se creează un obiect de tip *DBConnection*, prin intermediul căruia va putea fi apelată oricare din metodele acestei clase:

```
try{
        connection = new DBConnection(nume_baza_de_date, nume_utilizator, parola);
}
catch(Exception exc){
        exc.printStackTrace();
}
```

Observaţie:

Pentru conectarea aplicaţiei la un server MySQL, în proiect trebuie inclusă librăria care conţine driverul de MySQL: *mm.mysql-2.0.12-bin.jar*. Dacă lucraţi cu Java Builder, librăria se încarcă din "Project\Project Properties\Required Libraries".

Aplicaţia C# prezentată în continuare, se conectează la o bază de date Student creată în Microsoft SQL Server. Următorul exemplu prezintă conectarea la baza de date din SQL Server, unde *connectionString* se poate prelua din proprietățile bazei de date.

```
using System.Data.SqlClient;

string connectionString = @"Data Source=(local);Initial Catalog=Student;Integrated Security=True";
SqlConnection sqlConnection = new SqlConnection(connectionString);
sqlConnection.Open();
```

Interogarea bazei de date se poate realiza după cum urmează:

```
string query = "SELECT * FROM dbo.Student";
SqlCommand sqlCommand = new SqlCommand(query, sqlConnection);
SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
while (sqlDataReader.Read())
        Console.WriteLine(sqlDataReader.GetValue(0) + "; " + sqlDataReader.GetValue(1) + "; " +
                sqlDataReader.GetValue(2));
sqlCommand.Dispose();
sqlDataReader.Dispose();
```

Pentru inserare, modificare și ștergere se vor folosi comenzi de tip *ExecuteNonQuery()*.

Aplicaţii

1. Să se implementeze o aplicaţie pentru conectarea la o baza de date create în SQL Server care păstreză informaţii despre student. Aplicaţia trebuie să aibă o interfaţă care să permită vizualizarea (interogarea) tabelelor unei baze de date şi introducerea unor noi înregistrări.

2. Dezvoltaţi aplicaţia 1, prin introducerea funcţiilor care să permită modificarea şi respectiv ştergerea înregistrărilor existente.

6. Aplicații web

O variantă de aplicație web Java este Servletul. Servleturile sunt componente independente de platformă ale aplicaţiilor server şi au suport Java integrat [12]. Ele sunt independente de protocol, asigurând un cadru general pentru servicii pe baza modelului cerere-răspuns. Acest binecunoscut model este des folosit la programarea sistemelor distribuite, începând cu apeluri de proceduri la distanţă şi terminând cu protocolul HTTP pentru cereri către servere web. Cu ajutorul servlet-urilor extindem deci funcţionalitatea unei aplicaţii de tip server informaţional (nu neapărat server HTTP), un prim domeniu de aplicare fiind, bineînţeles, extinderea serverelor web.

Pentru rularea servlet-urilor se poate utiliza un server *Tomcat*, inclus în *Jbuilder 7*. Servlet-urile trebuie copiate în *Root* sau într-un director nou creat în directorul *webapps* din *Tomcat*, şi în funcţie de asta, trebuie să aibă următoarea structură:

Tip fişier	Root	Nume_Aplicatie
html	Webapps\Root*.html	Webapps\Nume_Aplicatie*.html
class	Webapps\Root\Web-inf\classes*.class	Webapps\Nume_Aplicatie\Web-inf\classes*.class
jar	Webapps\Root\Web-inf\lib*.jar	Webapps\Nume_Aplicatie\Web-inf\lib*.jar

Tabel 1. Structura unui servlet

În funcţie de directorul (Root sau Nume_Aplicatie) şi respectiv staţia pe care se află servlet-ul, cererea HTML va fi următoarea:

IP	Port	Root	Nume_Aplicatie
localhost	8080	http://localhost:8080/*.html	http://localhost:8080/Nume_Aplicatie/*.html
192.168.25.99	8080	http://192.168.25.99:8080/*.html	http://192.168.25.99:8080/Nume_Aplicatie/*.html

Tabel 2. Cereri HTML

Aplicaţia următoare este formată dintr-un HTML care permite introducerea unui *nume* şi apoi la apăsarea unui buton lansează o cerere spre servlet-ul *Hello*, care răspunde cu mesajul *Hello nume*. Clasa *Hello* are următoarea structură:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Hello extends HttpServlet {
  /**Process the HTTP Get request*/
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       String name = request.getParameter("name");
       out.println("<html>");
       out.println("<body>");
       out.println("<head>");
       out.println("<title>Hello, " + name + "</title>");
       out.println("</head>");
       out.println("<body>");
```

```
        out.println("<h1>Hello, " + name + "</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Prin metoda *getParameter* a clasei *HttpServletRequest* sunt preluaţi parametrii cererii (în cazul nostru *name*), iar prin intermediul clasei *HttpServletResponse*, este construit răspunsul sub forma unui HTML.

HTML-ul care lansează cererea spre servlet-ul *Hello*, are următoarea structură:

```
<html>
<head>
<title>Introductions</title>
</head>
<body>
<form method = get action="servlet/Hello">
If you don't mind me asking, what is your name?
<input type=text name="name"><p>
<input type =submit>
</form>
</body>
</html>
```

Observaţii

1. Dacă lucraţi cu Java Builder, pentru cele două aplicaţii se va crea câte un proiect. Clasele create în cadrul proiectelor nu trebuie incluse în pachete. În proiect trebuie încărcată şi librăria "Tomcat\Common\lib\servlet.jar". Se generează fişierele *.class prin comanda *make*.
2. Fişierele *.class şi *.html trebuie copiate în structura de directoare din Tomcat (vezi tabelul 1).
3. Dacă serverul Tomcat nu poate fi pornit, trebuie verificată existenţa variabilelor *Environment* (My Computer\Properties\Advanced): JAVA_HOME (calea spre JDK) şi CATALINA_HOME (calea spre Tomcat). În cazul în care nu există, ele trebuie create.
4. Dacă serverul Tomcat rulează, aplicaţia este pornită printr-un Browser Web (Internet Explorer – vezi tabelul 2).

Pentru realizarea unei aplicaţii web in ASP.NET sunt necesari urmatorii paşi:

- Se porneşte Visual Studio;
- Se creează un proiect nou ASP.NET Empty Project; Proiectul creeat va conţine iniţial Properties, References si web.config. În aplicaţia de faţă nu va fi nevoie editarea niciuneia dintre acestea.
- Se adaugă o nouă pagină în proiect din File->New File, (Ctrl+N) sau selectând proiectul în Solution Explorer şi dând click dreapta pe acesta apoi selectând din meniul care apare Add -> New Item;
- Din fereastra care apare se selectează Web Form şi se dă un nume acesteia, apoi se apasă Add;
- Urmează editarea paginii adaugate.

Se adaugă în pagină următoarea secvenţă de cod:

```
<label>
    If you don't mind me asking, what is your name?
</label>
<br />
<br />
<asp:TextBox runat="server" ID="txtBoxNameResponse"></asp:TextBox>
<br />
<asp:Button runat="server" ID="btnOkResponse" Text="Ok"
OnClick="btnOkResponse_Clicked" />
```

Elementele pe care este posibil să nu le recunoaşteţi până acum sunt:

```
<asp:TextBox runat="server" ID="txtBoxNameResponse"></asp:TextBox>
```

acesta reprezentând un control asemănător unui ‘input’ de tipul ‘text’, cu diferenţa că acesta rulează pe server, având control total asupra lui în momentul unui postback.

```
<asp:Button runat="server" ID="btnOkResponse" Text="Ok"
OnClick="btnOkResponse_Clicked" />
```

asemănător cu un ‘input’ de tipul ‘text’, cu aceleaşi caracteristici ca şi controlul anterior, iar evenimentul ‘OnClick’ se va executa pe server.

Pentru ca pagina să funcţioneze, trebuie adăugat handler-ul evenimentului asociat butonului. Acesta este următorul:

```
protected void btnOkResponse_Clicked(object sender, EventArgs e)
{
        Response.Write("Welcome " + this.txtBoxNameResponse.Text + "!");
}
```

Response reprezintă obiectul care va fi returnat clientului care a făcut cererea. El conţine deja toate elementele iniţiale. Metoda *Write* a acestuia va adăuga la începutul răspunsului ceea ce este trimis ca parametru, în cazul de faţă va fi : ”Welcome x !”, unde ‘x’ reprezintă conţinutul căsuţei text cu id-ul ‘txtBoxNameResponse’. Pentru a testa pagina este suficient să apasaţi F5 sau să daţi click pe Debug -> Start Debugging, eventual Start Without Debuging. Pagina se va deschide în browser, fiind afişate pe pagină textul, o căsuţă de introducere text şi un buton. În momentul apăsării butonului, se va afişa un mesaj la începutul paginii care va conţine şi textul din căsuţă (dacă există).

Cea de-a doua metodă va seta conţinutul unui element din pagină cu textul ce se găseşte în căsuţa text.

```
protected void btnOkPostback_Clicked(object sender, EventArgs e)
{
        var name = this.txtBoxNamePostback.Text;
        this.lblPostBack.InnerText = "Welcome " + name + "!";
}
```

Pentru ca metoda aceasta să funcţioneze, mai trebuie adăugat un element în formă, mai precis un label, care să ruleze pe server (runat=”server”):

```
<label id="lblPostBack" runat="server" style="color: red" />
```

Cea de-a treia metodă foloseşte tehnologia Ajax (Asynchronous JavaScript and XML) pentru a comunica cu serverul. Elementele schimbate sunt:

```
<label id="lblAjaxResponse" style="color: red">
        </label>
<input type="text" id="txtBoxAjax" width="50px"></input>
<input type="button" id="btnOkAjax" value="Ok" name="Ok"
onclick="javascript:btnOkAjax_Clicked(txtBoxAjax.value);" />
```

După cum se observă, aceste elemente sunt standard Html nefiind specifice ASP.NET; se pot utiliza şi cele .NET, însă nu este necesar accesul lor de pe server. Pentru a realiza un apel Ajax pe server trebuie să avem declarat în pagină un element nou:

```
<asp:ScriptManager ID="ScriptManager1" runat="server"
EnablePageMethods="true">
        </asp:ScriptManager>
```

Acesta trebuie să aibă atributul *EnablePageMethods* neaparat pe 'true'. După cum probabil aţi observat, evenimentul 'onclick' al butonului este o metodă javascript. Un exemplu de astfel de metodă este următorul:

```
<script type="text/javascript">
      function btnOkAjax_Clicked(name) {
    try {
       PageMethods.NameEntered(name, OnNameEnteredComplete);
    } catch (e) {
       alert(e.Message);
    }
  }
  function OnNameEnteredComplete(retVal, e) {
    if (retVal != null) {
       var lblAjaxResponse = document.getElementById("lblAjaxResponse");
       if (lblAjaxResponse != null) {
          lblAjaxResponse.innerText = retVal;
       }
    }
  }
</script>
```

Arată ca o funcţie normală, exceptând obiectul 'PageMethods'. Acesta se regăseşte în pagină în urma adăugării 'ScriptManager'-ului şi a atributului precizat. El permite apelul metodelor de pe server. Metoda care este apelată se numeşte 'NameEntered', însă numărul de parametri este diferit de cel de pe server, mai precis, în apel este permis să avem 3 metode ca parametri: *OnComplete*, *OnSuccess* şi *OnError*, însă de obicei este suficient *OnComplete*. Primul parametru este cel care va fi primit în metoda de pe server.

Pe server avem următoarea metodă:

```
[WebMethod]
public static string NameEntered(string name)
{
        return "Welcome " + name + "!";
}
```

Se observă că metoda aceasta prezintă trei diferenţe faţă de cele prezentate anterior:

- Deţine atributul [WebMethod]; va fi necesară folosirea unei noi librării pentru acest atribut (using System.Web.Services;) ce se adaugă deasupra declaraţiei clasei sau a namespace-ului.
- Metoda este publică, altfel nu va fi posibil accesul la ea;
- Returnează un ‘string’.

După apelul metodei, răspunsul va fi primit în funcţia javascript ‘OnNameEnteredComplete’ unde se tratează rezultatul.

Aplicaţii

1. Implementaţi aplicaţia prezentată în lucrare.

2. Realizaţi o aplicaţie servlet care să permită gestionarea unei baze de date MySQL:
   - conectarea la baza de date MySQL;
   - introducerea unei noi inregistrari;
   - afişarea rezultatelor interogărilor într-un tabel.

   Pentru această aplicaţie va fi creat HTML-ul, clasa *DBServlet* care extinde clasa *HttpServlet* şi clasa *DBConnection* preluată din lucrarea anterioară (*Conectarea la o bază de date MySQL*). Driverul de MySQL *mm.mysql-2.0.12-bin.jar* trebuie pus în directorul *lib* din *Web-inf* (vezi tabelul 1).

3. Implementați aplicația 2 în ASP.NET.

7. Aplicaţii distribuite

Java RMI (Remote Method Invocation) permite implementarea aplicaţiilor distribuite Java-to-Java, în care metodele obiectelor pot fi apelate la distanţă [13]. Astfel, o aplicaţie client poate folosi la distanţă obiectele unei aplicaţii server. O clasă poate fi instanţiată la distanţă, dacă ea implementează o interfaţă *Remote*. Figura 4 prezintă structura generală a unei aplicaţii Java RMI.

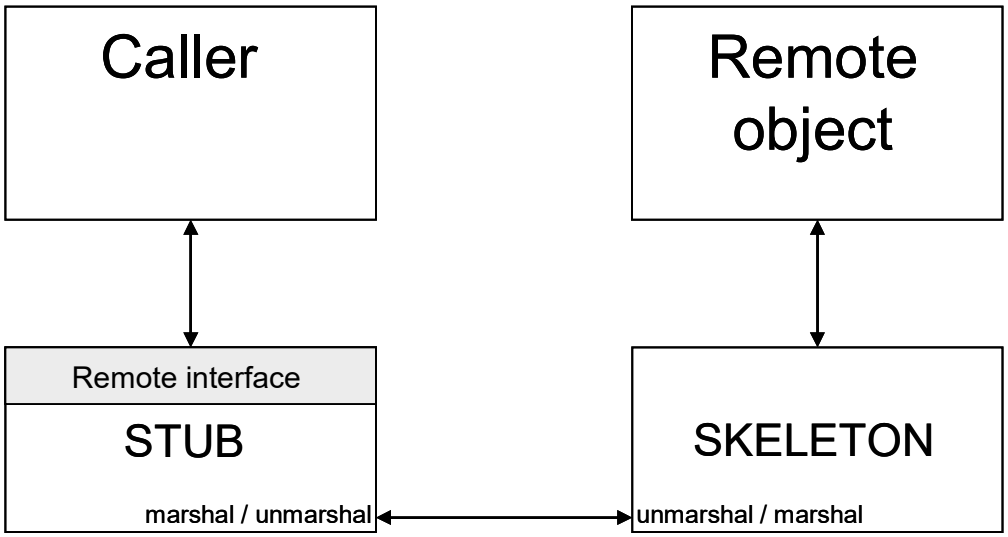

Figura 4. Structura unei aplicaţii Java RMI

La apelul unei metode la distanţă are loc următoarea secvenţă de operaţii:

- Clientul apelează metoda;
- Componenta STUB primeşte apelul, serializează parametrii şi îi trimite apoi spre componenta SKELETON a serverului;
- Componenta SKELETON primeşte apelul, deserializează parametrii şi îi trimite serverului;
- Serverul primeşte apelul şi parametrii, apelează metoda local şi trimite rezultatul componentei SKELETON;
- Componenta SKELETON primeşte rezultatul, îl serializează şi îl trimite mai departe componentei STUB a clientului;
- Componenta STUB primeşte rezultatul, îl deserializează şi îl transmite apelantului;
- Clientul primeşte rezultatul metodei apelate.

În această lucrare este prezentată o aplicaţie Java RMI în care clientul apelează la distanţă metoda *sayHello* a aplicaţiei server. Această metodă returnează mesajul “Hello World!”. Mesajul este preluat şi afişat de client. Interfaţa *Hello* conţine antetul metodei *sayHello* pe care dorim să o apelăm la distanţă.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
	String sayHello() throws RemoteException;
}
```

Clasa unui obiect apelabil la distanţă trebuie să implementeze cel puţin o interfaţă *Remote*. Clasa *HelloImpl* implementează interfaţa *Hello* de tip *Remote*, şi ea reprezintă aplicaţia server. Metoda *main* a serverului creează o instanţă a clasei *HelloImpl*, rezultând astfel obiectul ale cărui metode pot fi apelate la distanţă. Acestei instanţe i se asociază numele “Hello” în registrele RMI.

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.Naming;
import java.io.*;
import java.net.*;
import sun.security.util.*;

public class HelloImpl extends UnicastRemoteObject implements Hello{

	public HelloImpl() throws RemoteException {
		super();
	}

	public String sayHello() {
		return "Hello World!";
	}

	public static void main(String args[]) {
		if (System.getSecurityManager() == null) {
			System.setSecurityManager(new RMISecurityManager());
		}
		try {
			HelloImpl obj = new HelloImpl();
			// se atribuie instantei acestui obiect numele "Hello"
			Naming.rebind("//localhost/Hello", obj);
			System.out.println("Hello bound in registry");
		} catch (Exception e) {
			System.out.println("HelloImpl err: " + e.getMessage());
			e.printStackTrace();
		}
	}
}
```

Compilarea serverului se efectuează prin comenzile JDK:

```
javac server\HelloImpl.java
rmic -d . server.HelloImpl
```

După înregistrarea obiectului, acesta poate fi căutat de aplicaţiile client, folosind metoda *Naming.lookup()*, prin numele care i s-a asociat în registrele RMI. Aplicaţia prezentată mai jos apelează la distanţă metoda *sayHello* a serverului, şi afişează mesajul returnat: “Hello World!”.

```
import java.rmi.Naming;
import java.rmi.RMISecurityException;

public class SayHello {

	public static void main(String args[]) {
		String message = "blank";
		if (System.getSecurityManager() == null) {
			System.setSecurityManager(new RMISecurityManager());
		}
		try {
			Hello hello = (Hello)Naming.lookup("//localhost/Hello");
			message = hello.sayHello();
		} catch (Exception e) {
			System.err.println("Hello-exception: " + e.getMessage());
			e.printStackTrace();
		}
		System.out.println(message);
	}
}
```

Compilarea clientului se efectuează prin comanda JDK:

```
javac client\SayHello.java
```

Aplicaţia poate fi rulată prin următoarele comenzi JDK:

```
rmiregistry
java -Djava.rmi.server.hostname=localhost -Djava.security.policy=java.policy HelloImpl
java -Djava.security.policy=java.policy SayHello localhost 20
```

.NET Remoting

Pentru realizarea aplicațiilor .NET Remoting, cele două programe (server şi client) necesită versiunea 3.5 de .NET Framework pentru a funcţiona. Soluţia este creată în Visual Studio 2010. Pentru a rula programele din Visual Studio se poate da un 'RUN' simplu deoarece atât clientul cât şi serverul sunt setate ca startup projects. Programele pot fi rulate şi prin fişierele 'runClient.bat' si 'runServer.bat'. Atât portul de ascultare al serverului şi URL-ul pentru obiectul care este apelat la distanţă cât şi adresa la care se conectează clientul pot fi modificate din fişierul *app.config* al fiecărui executabil (aflat în bin/Debug/) fără a necesita recompilarea programelor. Urmează în continuare aplicaţia .NET Remoting în care clientul apelează la distanţă metoda *sayHello* a aplicaţiei server. Sunt prezentate: interfaţa (folosită pentru comunicaţie între client şi server), serverul și clientul.

Interfaţa

```
// The interface used to communicate between the client and
// the server using .NET Remoting.
public interface IHello
{
    // Returns a 'hello world' message to the caller.
    // <returns>The message</returns>
    string SayHello();
}
```

Serverul

```
using System;
using Infrastructure;

// A remotable object (MarshalByRefObject makes it remotable) that implements the IHello interface.
public class Hello : MarshalByRefObject, IHello
{
    #region Implementation of IHello

    // Returns a 'hello world' message to the caller.
    // <returns>The message</returns>
    public string SayHello()
    {
        return "Hello from the IHello implementation on the server.";
    }

    #endregion
}
```

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

class Program
{
    static void Main(string[] args)
    {
        AppSettingsReader reader = new AppSettingsReader();
        int port = (int)reader.GetValue("port", typeof(int));
        string objectURL = (string)reader.GetValue("objectURL", typeof(string));

        // Creating a TcpChannel on the 8888 port and binding it for use.
        TcpChannel channel = new TcpChannel(port);
        ChannelServices.RegisterChannel(channel, true);

        // Registering the Hello class on the 'tcp://localhost:8888/hello' address.
        // When the Activator.GetObject method is called this is the URL that will have to be given to
access the IHello interface.
        RemotingConfiguration.RegisterWellKnownServiceType(typeof(Hello), objectURL,
WellKnownObjectMode.SingleCall);

        // Waiting for any clients to use the server.
```

```
            Console.WriteLine("The server is ready, press any key to close the console");
            Console.ReadKey();
        }
    }
```

Fişierul de configurare al serverului

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v2.0.50727"/>
  </startup>
  <appSettings>
    <add key="port" value="8888"/>
    <add key="objectURL" value="hello"/>
  </appSettings>
</configuration>
```

Clientul

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

    class Program
    {
        static void Main(string[] args)
        {
            AppSettingsReader reader = new AppSettingsReader();
            string serverObjectURL = (string)reader.GetValue("serverObjectURL", typeof(string));

            Thread.Sleep(1000); // Wait for the server to start.

            IHello helloImplementation = null;

            // Creating a TcpChannel and binding it in order to use it for Remoting.
            TcpChannel channel = new TcpChannel();
            ChannelServices.RegisterChannel(channel, true);
            try
            {
                // The activator will look for any bound implementation of the IHello interface on the specified
URL.
                helloImplementation = Activator.GetObject(typeof(IHello), serverObjectURL) as IHello;
            }
            catch (RemotingException rex)
            {
                Console.WriteLine(string.Format("Error: RemotingException encoutered - {0}", rex.Message));
            }
            catch (MemberAccessException maex)
            {
                Console.WriteLine(string.Format("Error: MemberAccessException encoutered - {0}",
maex.Message));
            }
```

```
        // Calling the SayHello method through the implementation.
        string message = helloImplementation.SayHello();
        Console.WriteLine(string.Format("The server says: '{0}'", message));

        Console.WriteLine("Press any key to close the console");
        Console.ReadKey();
    }
}
```

Fişierul de configurare al clientului

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v2.0.50727"/>
  </startup>
  <appSettings>
    <add key="serverObjectURL" value="tcp://localhost:8888/hello"/>
  </appSettings>
</configuration>
```

Windows Communication Foundation

Tehnologia .NET Remoting a fost înlocuită cu Windows Communication Foundation (WCF) care permite crearea aplicațiilor orientate pe servicii, inclusiv a unor aplicații client-server cu metode declarate în server și apelate din client. Clientul și serverul sunt independenți din punct de vedere al sistemului de operare sau al limbajului de programare. În continuare, este prezentată aplicația WCF în care clientul apelează la distanţă metoda *sayHello* a aplicaţiei server. Visual Studio trebuie să aibă instalat pachetul WCF (din Tools -> Get Tools and Features -> Individual Components). Interfața trebuie să existe atât în server cât și în client. Serverul trebuie să fie proiect de tip consolă .NET Framework.

Interfaţa

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;

namespace Lab5_Server
{
    [ServiceContract]
    public interface IWCFServer
    {
        [OperationContract]string SayHello();
    }
}
```

Serverul

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Lab5_Server
{
    internal class WCFServer : IWCFServer
    {
        public string SayHello()
        {
            return "Hello from the IHello implementation on the server.";
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;

namespace Lab5_Server
{
    internal class Program
    {
        static void Main(string[] args)
        {
            NetTcpBinding binding = new NetTcpBinding();
            Uri baseAddress = new Uri("net.tcp://localhost:8000/wcfserver");

            using (ServiceHost serviceHost = new ServiceHost(typeof(WCFServer), baseAddress))
            {
                serviceHost.AddServiceEndpoint(typeof(IWCFServer), binding, baseAddress);
                serviceHost.Open();

                Console.WriteLine($"The WCF server is ready at {baseAddress}.");
                Console.WriteLine("Press <ENTER> to terminate service...");
                Console.WriteLine();
                Console.ReadLine();
            }
        }
    }
}
```

Clientul

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Lab5_Client
{
   internal class Program
   {
      static void Main(string[] args)
      {
         NetTcpBinding binding = new NetTcpBinding();
         string url = "net.tcp://localhost:8000/wcfserver";
         EndpointAddress address = new EndpointAddress(url);
         ChannelFactory<IWCFServer> channelFactory = new ChannelFactory<IWCFServer>(binding,
address);
         IWCFServer server = channelFactory.CreateChannel();
         Console.WriteLine(server.SayHello());
      }
   }
}
```

Aplicaţii

1. Să se implementeze aplicaţia prezentată în lucrare.

2. Să se implementeze o aplicaţie Java RMI, .NET Remoting sau WCF, pentru apelul la distanţă al unor metode care să efectueze operaţiile aritmetice elementare.

3. Să se integreze clientul de MySQL implementat anterior într-o aplicaţie Java RMI, .NET Remoting sau WCF.

8. Algoritmi paraleli

Algoritmii paraleli [6] folosesc resursele specifice unui sistem de calcul paralel (un număr de *p* procesoare) pentru rezolvarea unei probleme. Un algoritm secvenţial poate fi paralelizat dacă secvenţa de operaţii elementare generată poate fi descompusă în *p* subsecvenţe, în aşa fel încât operaţiile din fiecare subsecvenţă să poată fi executate concurenţial de procesoarele disponibile. Există situaţii în care această descompunere este naturală, în general însă, algoritmul secvenţial cel mai natural pentru rezolvarea unei probleme nu conduce la o împărţire în astfel de subsecvenţe [6]. Caracteristicile specifice aplicațiilor paralele bazate pe Message Passing Interface (MPI) sunt prezentate în [10].

MPJ express este o implementare Java pentru standardul MPI. Pentru a instala şi utiliza acest pachet sub Windows se recomandă utilizarea mediului de dezvoltare NetBeans. MPJ necesită Java 1.5 SDK sau o versiune superioară. Pachetul de instalare poate fi descărcat de pe www.sun.org. De asemenea, se poate instala doar NetBeans de pe http://netbeans.org/downloads/index.html. Se descarcă MPJExpress de la http://mpj-express.org/download.php. Este necesară autentificarea, dar înregistrarea este gratuită.

În MyComputer->Properties->Advanced Tab-> Environment Variables trebuie adaugată variabila:

MPJ_HOME = [cale către mpj (ex: C:\MPJ)]

şi trebuie adaugată la variabila PATH, calea către directorul bin (ex: C:\MPJ\bin). De aici înainte se presupune ca pachetul MPJ Express este instalat în C:\MPJ. În C:\MPJ\bin se află installmpjd-windows.bat. Acest batch file, trebuie rulat pentru a instala serviciul necesar rulării aplicaţiilor MPI. Dupa ce a fost instalat, poate fi pornit din Start->Run-> service.msc. Din lista de servicii trebuie selectat MPJ Daemon şi pornit.

Se creează un proiect nou, File->New Project->Java->Java Application. Presupunem că acesta se va numi MPJHello. În tab-ul cu ierarhia proiectului, trebuie adăugate librăriile mpi.jar şi mpj.jar prin click dreapta pe MPJ->Libraries, apoi Add JAR/Folder... În tab-ul Files, în directorul proiectului MPJHello trebuie creat fişierul mpj.conf. Acesta va conţine următoarele:

```
# Number of processes
3
# Protocol switch limit
131072
# Entry in the form of machinename@port@rank
localhost@20000@0
localhost@20020@1
localhost@20030@2
```

Pentru fiecare maşină/proces trebuie creată o configuraţie pentru rulare. Pe tab-ul Projects, click dreapta pe MPJHello -> Set Configuration -> Customize şi din panoul din partea dreaptă, se selectează New. Apoi se selectează numele. De exemplu mpj0. În căsuţa de text

Arguments se va specifica: 0 mpj.conf niodev , unde prima cifră este numărul procesului. Pentru fişierul mpj.conf de mai sus, ar trebui create 3 configuraţii:

mpj0: 0 mpj.conf niodev
mpj1: 1 mpj.conf niodev
mpj2: 2 mpj.conf niodev

În cazul în care se doreşte rularea pe mai multe staţii, pentru fiecare staţie se va crea un fişier mpj.conf ce va conţine IP-urile şi rangul tuturor staţiilor.

Funcțiile de transfer sunt:

- Send(Object buffer, int offset, int count, Datatype type, int dest, int tag);
- Recv(Object buffer, int offset, int count, Datatype type, int src, int tag);

Unde Buffer este adresa tabloului trimis / primit, Offset este indexul în cadrul tabloului incepând de la care dorim să se trimită / primească datele, Count este numarul de elemente trimise / primite, *Type* este tipul de date (ex. MPI.CHAR, MPI.INT, MPI.OBJECT, etc.), *Dest* este rangul (identificatorul) procesului destinaţie, *Src* este rangul (identificatorul) procesului sursă, iar *Tag* este un identificator unic al mesajului (cel din Recv trebuie să corespundă cu cel din funcţia Send aferentă).

În continuare se prezintă aplicaţia *PingPong* în care are loc un transfer de mesaje între *p* procese, care pot fi pornite pe calculatoare diferite. Procesul 0 primeşte şi afişează numele staţiilor pe care au fost pornite celelalte procese. După cum se poate observa în codul sursă al aplicaţiei, prezentat mai jos, aceste transferuri de mesaje sunt efectuate prin instrucţiunile *Send* şi *Recv* existente în librăria MPI.

```
import mpi.MPI;

public class MPJHello {
   public static void main(String[] args) {
      MPI.Init(args);
      int me = MPI.COMM_WORLD.Rank();
      int size = MPI.COMM_WORLD.Size();
      if(me==0){
         System.out.println("Node "+me+" is running");
         for(int dest=1; dest<size; ++dest){
            System.out.println("Sending to node " + dest);
            String s = "Hello from node 0";
            char[] message = s.toCharArray();
            char[] str = new char[100];
            MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, dest, 0);
            MPI.COMM_WORLD.Recv(str, 0, 100, MPI.CHAR, dest, 0);
            System.out.println(new String(str).trim());
         }
      }
      else{
         char[] str = new char[100];
         String s = "Hello from node " + me;
         char[] message = s.toCharArray();
         MPI.COMM_WORLD.Recv(str, 0, 100, MPI.CHAR, 0, 0);
         MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 0, 0);
      }
      MPI.Finalize();
   }
```

}

Pentru a rula programul, trebuie setate pe rând configuraţiile de rulare. Mai întâi trebuie să ne asigurăm că proiectul curent este setat ca MainProject:

Run -> Set Main Project -> MPJHello.

Trebuie selectată prima configuraţie:

Run->Set Project Configuration-> mpj0

Apoi se rulează aplicaţia (F6). La fel trebuie procedat pentru restul configuraţiilor (mpj1, mpj2). După ce toate acestea au fost pornite, va începe execuţia programului.

Limbajul C# permite implementarea aplicaţiilor paralele printr-o librărie MPI [14]. Pentru înţelegerea structurii unui program paralel implementat în MPI.NET, această lucrare prezintă aplicaţia *PingPong* în care are loc un transfer de mesaje între *p* procese, care pot fi pornite pe calculatoare diferite. Procesul 0 primeşte şi afişează numele staţiilor pe care au fost pornite celelalte procese. După cum se poate observa în codul sursă al aplicaţiei, prezentat mai jos, aceste transferuri de mesaje sunt efectuate prin instrucţiunile *Send* şi *Receive* existente în librăria MPI.

```
/* Copyright (C) 2007  The Trustees of Indiana University
 *
 * Use, modification and distribution is subject to the Boost Software
 * License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
 * http://www.boost.org/LICENSE_1_0.txt)
 *
 * Authors: Douglas Gregor
 *          Andrew Lumsdaine
 *
 * This test exercises Communicator.ReduceScatter.
 */
using System;
using MPI;
class PingPong
{
   static void Main(string[] args)
   {
      using (new MPI.Environment(ref args))
      {
         Intracommunicator comm = Communicator.world;
         if (comm.Rank == 0)
         {
            Console.WriteLine("Rank 0 is alive and running on " + MPI.Environment.ProcessorName);
            for (int dest = 1; dest < comm.Size; ++dest)
            {
               Console.Write("Pinging process with rank " + dest + "...");
```

```
                comm.Send("Ping!", dest, 0);
                string destHostname = comm.Receive<string>(dest, 1);
                Console.WriteLine(" Pong!");
                Console.WriteLine("  Rank " + dest + " is alive and running on " + destHostname);
            }
        }
        else
        {
            comm.Receive<string>(0, 0);
            comm.Send(MPI.Environment.ProcessorName, 0, 1);
        }
    }
  }
}
```

Pentru transfer de date simple (*int*, *string*, etc.) de la un proces la altul funcţiile blocante *Send* şi *Receive* se apelează după cum urmează:

```
Send<type>(d_src, proc_dest, tag);
d_dest = Receive<type>(proc_src, tag);
```

unde type reprezintă tipul datelor transmise, d_src / d_dest reprezintă variabila care conţine valoarea înainte / după transfer, proc_src / proc_dest reprezintă codurile proceselor sursă / destinaţie, iar tag ne permite să formăm perechi send / receive (datele sunt recepţionate de la procesul sursă doar dacă tag-urile corespund).

Pentru transfer de tablouri se pot folosi variantele:

```
Send(d_src, proc_dest, tag);
Receive(proc_src, tag, ref d_dest);
```

sau, pentru transfer de matrici:

```
Send(d_src, proc_dest, tag);
Receive(proc_src, tag, out d_dest);
```

Variantele neblocante ale funcţiilor de transfer sunt *ImmediateSend* respectiv *ImmediateReceive*. În cazul unui apel al funcţiei *Barrier* sistemul va aştepta ajungerea tuturor proceselor la barieră, permiţând astfel sincronizarea acestora. O altă posibilitate de sincronizare constă în apelul funcţiei *Wait* a obiectului *Request* returnat de o operaţie de transfer neblocantă, determinând astfel aşteptarea finalizării operaţiei respective. Prin apelul metodei *Test* a unui obiect *Request* se poate determina dacă operaţia neblocantă aferentă s-a terminat.

Pentru dezvoltarea aplicaţiilor paralele folosind MPI.NET trebuie instalat MPI.NET SDK, iar pentru rularea acestora se instalează Microsoft Compute Cluster Pack SDK (sdk_x86.msi) [14].

Orice aplicaţie MPI.NET trebuie să aibă introdusă o referinţă la librăria MPI.dll.

Aplicaţiile MPI pot fi pornite pe staţia locală (localhost) prin următoarea comandă:

```
mpiexec –n   nr_procese   NumeAplicatie.exe
```

De exemplu, comanda

```
mpiexec –n 5 PingPong
```

va porni aplicaţia PingPong pe cinci procese, toate pe staţia locală.

Pentru rularea pe mai multe staţii, se copiază executabilul programului pe care vrem sa-l rulăm în folderul Microsoft Compute Cluster Pack\Bin\ al fiecărei staţii folosite şi se porneşte executabilul smpd pe fiecare staţie, prin comanda:

```
smpd –d
```

Apoi se va folosi una din următoarele comenzi:

```
mpiexec –hosts nr_statii nume_statie_1 nume_statie_2 nume_statie_n PingPong
mpiexec –hosts nr_statii nume_st1 nr_procese nume_stn nr_procese PingPong
mpiexec –machinefile machine.file –n nr_procese PingPong
```

unde machine.file conţine fiecare staţie pe câte un rând.

Exemple

Comanda efectuată pe statie10:

```
mpiexec –hosts 4 statie10 statie07 statie09 statie08 PingPong
```

porneşte patru procese pe patru staţii diferite (staie10, statie07, statie09 şi statie08). **Prima staţie trebuie să fie cea locală!**

Comanda efectuată pe statie10:

```
mpiexec –hosts 4 statie10 3 statie07 2 statie09 5 statie08 6 PingPong
```

foloseşte patru staţii, pornind trei procese pe staţia locală (statie10), două procese pe statie07, cinci procese pe statie09, respectiv şase procese pe statie08. **Şi în acest caz prima staţie trebuie să fie cea locală!**

Aplicaţii

1. Să se implementeze şi să se ruleze aplicaţia *PingPong* prezentată în lucrare.
2. Să se implementeze un algoritm paralel care calculează suma elementelor unui tablou.
3. Să se implementeze un algoritm paralel care determină maximul dintr-un tablou.
4. Să se implementeze un algoritm paralel care determină minimul dintr-o matrice.
5. Să se implementeze un algoritm paralel care determină suma elementelor unei matrici.
6. Să se implementeze un algoritm paralel pentru adunarea matricilor.
7. Să se implementeze un algoritm paralel pentru înmulţirea matricilor.
8. Să se implementeze şi să se ruleze algoritmul Quicksort paralel.
9. Să se implementeze şi să se ruleze algoritmul Mergesort paralel.
10. Să se implementeze paralelizat problema celor n regine.

Bibliografie

[1] Ari B., *Principles of Concurrent and Distributed Systems*, Addison Wesley, 1990.

[2] Athanasiu I., *Java ca limbaj pentru programarea distribuită*, Matrix Rom, Bucureşti, 2002.

[3] Attiya H., Welch J., *Distributed Computing*, McGraw Hill, London, 1998.

[4] Chan M. C., Griffith S. W., Iasi A. F., *Java 1001 Secrete pentru Programatori*, Teora, 2000.

[5] Coulouris G., Dollimore J., Kindberg T., *Distributed Systems: Concepts and Design*, 3rd Edition, Addison Wesley, 2001.

[6] Croitoru C., *Introducere în proiectarea algoritmilor paraleli*, Matrix Rom, Bucureşti, 2002.

[7] Eckel B., *Thinking in Java*, 3rd edition (electronic version: www.pythoncriticalmass.com), Prentice-Hall, 2003.

[8] Geary D. M., *Advanced Java Server Pages*, Prentice-Hall, 2001.

[9] Gellert A., *Analiza şi proiectarea algoritmilor: o abordare pragmatică prin aplicaţii Java*, Editura Techno Media, ISBN 978-606-8030-81-4, 2010.

[10] Mihu I.Z., Caprita H.V., Gellert A., *Parallel Programming Using MPI Library on Message-Passing Architectures*, Proceedings of the 6th International Conference on Technical Informatics (CONTI 2004), Transactions on Automatic Control and Computer Science, Vol. 4, pp. 37-42, Timisoara, May 2004.

[11] Gordon R., *Java Native Interface*, Prentice-Hall, 1998.

[12] Hunter J., *Java Servlet Programming*, 2nd Edition, O'Reilly & Associates, 2001.

[13] Mahmoud Q. H., *Distributed Programming with Java*, Manning Press, 2001.

[14] MPI.NET, http://www.osl.iu.edu/research/mpi.net/

[15] Orfali R., Harkey D., *Client/Server Programming with Java and Corba*, 2nd Edition, J. Wiley & Sons, New York, 1998.