

Arpad GELLERT

Algoritmi Paraleli și Distribuți

Aplicații

Cuprins

1. Operații I/O	3
2. Fire de execuție	7
3. Sincronizarea firelor de execuție	14
4. Aplicații client-server	18
5. Conectarea la o bază de date MySQL	25
6. Aplicații web	28
7. Aplicații distribuite	33
8. Algoritmi paraleli	39
Bibliografie	44

1. Operații I/O

Un flux de intrare/ieșire (I/O) furnizează calea prin care programele pot trimite o secvență de octeți de la o sursă către o destinație [7]. Un flux de intrare este o sursă (producător) de octeți, iar un flux de ieșire este o destinație pentru octeți (consumator). Deși fluxurile I/O sunt deseori asociate cu accesul la fișierele de pe disc, sursa și destinația unui program pot fi, de asemenea, tastatura, mouse-ul, memoria sau fereastra de afișare [9].

Citirea unui șir 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 secvență citirea de la tastatură a unui șir 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 șir de caractere *str* într-un întreg se poate realiza folosind funcția 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 excepțiilor, în cazul conversiilor trebuie tratată excepția *NumberFormatException*.



Citirea unui șir de caractere de la tastatură în C# se face cu funcția *readLine*:

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

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

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

Pentru evitarea excepțiilor, în cazul conversiilor trebuie tratată excepția *FormatException*.

Citirea dintr-un fișier

În exemplul următor, se afișează pe ecran toate liniile citite din fișierul *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* definește clasa *StringTokenizer* care facilitează separarea unui șir de caractere în simboluri. Trebuie creat un obiect *StringTokenizer*, specificând șirul și delimitatorul. Următoarea secvență de program afișează pe ecran simbolurile șirului delimitate prin caracterul spațiu.

```
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 șir poate fi separat în simboluri folosind funcția *Split* a clasei *string*. Următorul exemplu procesează un fișier 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 fișier



Următoarea secvență de program scrie în fișierul *data.txt* întregul 10 și float-ul 3.14, apoi citește aceste valori din fișier și le afișează 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 secvență de program scrie în fișierul data.txt șirul de caractere “Hello World!”, apoi îl citește din fișier și îl afișează 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();

```

Aplicații

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

2. Fire de execuție

Un fir de execuție (thread) este o secvență de instrucțiuni ale unui program executabil. Firul de execuție principal este metoda *main* și atunci când acesta se termină, se încheie și programul. Programele pot executa în paralel două sau mai multe fire de execuție, 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 execuție în Java, se poate extinde clasa *Thread*. Deoarece Java nu acceptă moștenirea multiplă, în cazul în care a fost deja extinsă o clasă, pentru crearea unui fir de execuție trebuie implementată interfața *Runnable*. Indiferent de metoda utilizată, se suprascrive metoda *run* care trebuie să conțină instrucțiunile firului.

Aplicația următoare pornește două fire de execuție: unul pentru afișarea numerelor și celălalt pentru afișarea literelor. Pentru a observa diferențele dintre cele două metode de implementare, firul de execuție *Numbers* extinde clasa *Thread*, în timp ce *Letters* implementează interfața *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 priorității unui fir se apelează metoda *setPriority* a clasei *Thread*. De asemenea, pentru a afla prioritatea unui fir se folosește metoda *getPriority*. Pentru suspendarea unui fir de execuție pentru o perioadă de timp precizată în milisecunde, se utilizează metoda *sleep* a clasei *Thread*.



Pentru implementarea unui fir de execuție în C# se folosește clasa *Thread*. Următoarea aplicație de tip *Console Application* pornește două fire de execuție: unul pentru afișarea numerelor prin funcția *numbers* și celălalt pentru afișarea literelor prin funcția *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 execuție pentru o perioadă de timp precizată în milisecunde, se utilizează metoda *Sleep* a clasei *Thread*. Firul de execuție poate fi oprit temporar cu funcția *Suspend*, repornit cu funcția *Resume*, respectiv oprit definitiv cu funcția *Abort* a clasei *Thread*.

Lucrul cu imagini

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



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

și apoi se copiază conținutul bufferului pe fereastra aplicației [4]. Clasa Ball, instanțiată 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 {
            jblnit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        setVisible(true);
        ball = new Ball(this, 20, 50, 20, Color.red);
        ball.start();
    }
}
```

```

private void jblnit() 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);
}
}

```



Aplicația C# de tip *Windows Application* este formată din două clase: Ball și MyForm. Clasa Ball, instanțiată 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();
    }
}

```

Atenție, fișierul MyForm.cs se va genera automat, urmând să-l completați conform clasei MyForm de mai sus. Evenimentelor Load, Paint și FormClosing trebuie asociate funcțiile corespunzătoare (MyForm_Load, MyForm_Paint și MyForm_FormClosing) în Design/Properties/Events!

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

Aplicații

1. Să se dezvolte clasa *Ball* din aplicația prezentată pentru modelarea mișcării 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* și *Ball* în așa fel încât să poată fi pornite mai multe bile simultan.
3. Să se realizeze o aplicație care să afișeze reclamele, câte 5 secunde fiecare, într-o buclă infinită. Fiecare reclamă are o anumită prioritate și în funcție de aceasta, ea apare cu o anumită frecvență. Căile imaginilor și prioritățile acestora vor fi preluate dintr-un fișier.

Indicații pentru aplicația 3



Pentru încărcarea unei imagini în Java se folosește metoda *getImage* a clasei *Toolkit*. Una din metodele de a obține obiectul *Toolkit* este prin folosirea metodei statice *getDefaultToolkit* a clasei *Toolkit*. Pentru preîncărcarea imaginilor în memorie, se folosește clasa *MediaTracker*. Metoda *waitForId* a clasei *MediaTracker* permite încărcarea imaginilor care au un anumit ID (mai multe imagini pot avea același ID). ID-urile se setează la adăugarea imaginilor în obiectul *MediaTracker*. Metoda *waitForAll* încarcă în memorie toate imaginile adăugate în obiectul *MediaTracker*. După ce imaginile au fost încărcate ele pot fi desenate pe orice componentă folosind metoda *drawImage* a clasei *Graphics*. Următoarea secvență 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 funcția *DrawImage* a clasei *Graphics*. Pentru asta, trebuie obținut obiectul *Graphics* al ferestrei sau al componentei pe care se dorește să se afișeze 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 execuție

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



Dacă în cadrul unui program Java există un fir de execuție care creează (produce) date și un al doilea fir de execuție 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 așteptare. Atunci când un fir de execuție apelează *wait* în cadrul unui bloc de cod *synchronized*, alt fir poate accesa codul. Iar atunci când un fir de execuție încheie procesarea codului *synchronized*, el apelează metoda *notify* pentru a anunța alte fire de execuție să înceteze așteptarea. În cazul rezolvării ecuației de gradul 2 – aplicație prezentată în continuare – firele *tx1* respectiv *tx2* așteaptă 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 și *ManualResetEvent* permit firelor de execuție să comunice între ele prin “semnalizare”. Un fir așteaptă un “semnal” apelând *WaitOne* pe *ManualResetEvent* (sau *AutoResetEvent*). Dacă *ManualResetEvent* este în starea *nesetat*, firul se blochează și așteaptă până când acesta ajunge în starea *setat*, în urma apelului metodei *Set*. Astfel, dacă în cadrul unui program există un fir de execuție care creează (produce) date și un al doilea fir de execuție care le prelucrează (consumă), firul *consumator* așteaptă până când este activat de firul *producător*. În cazul rezolvării ecuației de gradul 2 – aplicație prezentată în continuare – firele *tx1* respectiv *tx2* așteaptă 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);
}
}
}

```

Aplicații

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

4. Aplicații client-server

Rețelele fac posibile multe tipuri noi de aplicații, deoarece nu mai este necesar ca un singur calculator să execute totul [4]. În cadrul unei rețele, câteva calculatoare, denumite servere, efectuează activități 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 primește servicii de la un server.

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

Atunci când două aplicații trebuie să comunice, ele trebuie să se găsească reciproc. Dacă două aplicații își cunosc reciproc soclul, ele pot crea o conexiune de soclu. De obicei este responsabilitatea clientului să caute serverul inițializând o conexiune de socluri. Serverul creează un soclu pe care îl va utiliza pentru comunicarea cu clientul și își transmite propria adresă de soclu către aplicația client în primul său mesaj de răspuns [16].



Pentru implementarea aplicației client în Java, se folosește clasa *Socket* precizând IP-ul respectiv portul serverului. Pentru crearea unui server în Java, se folosește clasa *ServerSocket*, precizând portul serverului. Apoi serverul apelează metoda *accept* pentru a aștepta 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 aplicații). În aplicația următoare, clientul trimite o valoare serverului, iar serverul primește valoarea și o afișează. Aplicația se va testa pornind prima dată serverul și 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 aplicației client în C#, se folosește clasa *TcpClient* precizând IP-ul respectiv portul serverului. Pentru crearea unui server în C#, se folosește clasa *TcpListener*, precizând portul serverului. Apoi serverul apelează metoda *AcceptTcpClient* pentru a aștepta conectarea unui client. Atunci când un client lansează o cerere de conectare, metoda *AcceptTcpClient* returnează soțul clientului (un obiect de tip *TcpClient*), prin care se va realiza comunicarea dintre cele două aplicații. În aplicația următoare, clientul trimite o valoare serverului, iar serverul primește valoarea și o afișează. Aplicația se va testa pornind prima dată serverul și 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);
            }
        }
    }
}

```

Aplicații

1. Să se implementeze aplicația prezentată în lucrare.
2. Să se modifice aplicația prezentată în lucrare astfel încât clientul să trimită serverului mesajul *Hello Server* iar serverul să raspundă *Hello Client*.
3. Realizarea unei aplicații client-server pentru listarea fișierelor text la imprimantă. Aplicația va avea următorul protocol de comunicare:

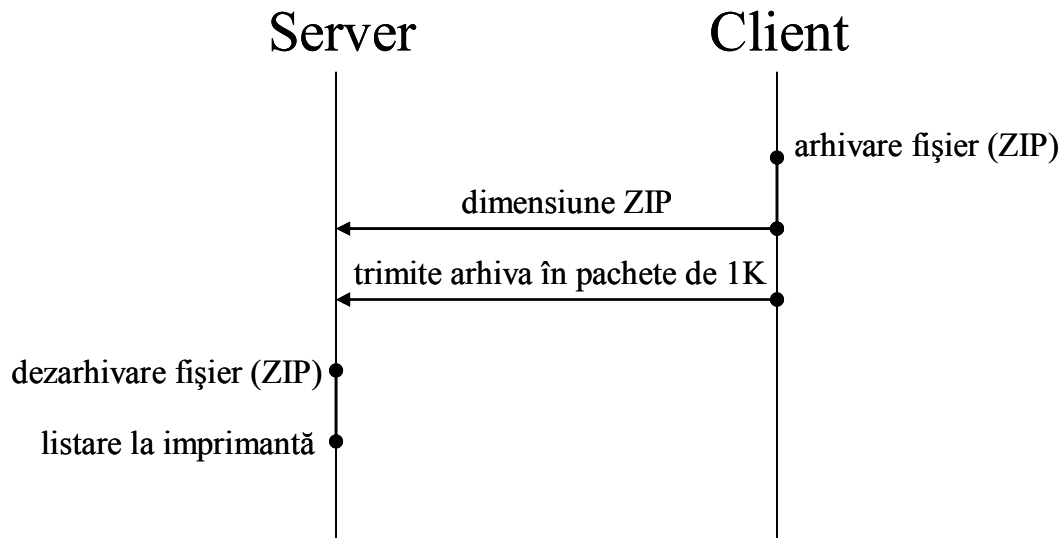


Figura 1. Transfer de fișiere

Indicații pentru aplicația 3

Așa cum se poate observa în secvențele următoare, arhivarea și dezarhivarea se vor face folosind clasele `ZipOutputStream` și respectiv `ZipInputStream`. Pentru listarea fișierului la imprimantă se poate folosi clasa `PrintJob`.



SERVER (dezarhivare)

```
try{
    FileInputStream fileIn = new FileInputStream(zip);
    FileOutputStream fileO = new FileOutputStream(umeFisier);
    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 aplicația 3 în așa fel încât ea să funcționeze corect și în situația în care anumite pachete de date se pierd și nu ajung la server. Modificarea constă în retransmisia pachetelor pierdute.
5. Realizarea unei aplicații client-server pentru apelul funcțiilor la distanță:
 - clientul cere apelul unei funcții cu anumiți parametri;
 - serverul apelează funcția și trimite rezultatul aplicației client.
6. Să se dezvolte aplicația 5 prin introducerea unei structuri de tip buffer, care să rețină rezultatele celor mai frecvente operații efectuate. În cazul în care rezultatul unei operații este găsit în acest buffer, operația respectivă nu se mai efectuează, rezultatul transmis aplicației client fiind cel din buffer.
7. Să se implementeze o aplicație de chat.

Indicații pentru aplicația 7

Aplicația va fi formată din programul server respectiv programul client. Clientul va avea o interfață grafică (Figura 2) formată din următoarele componente: lista clienților conectați, lista de mesaje, căsuță de editare pentru introducerea mesajelor, buton SEND pentru trimiterea mesajului introdus către clientul selectat. Recepționarea mesajelor se va face într-o buclă infinită implementată pe un fir de execuție.

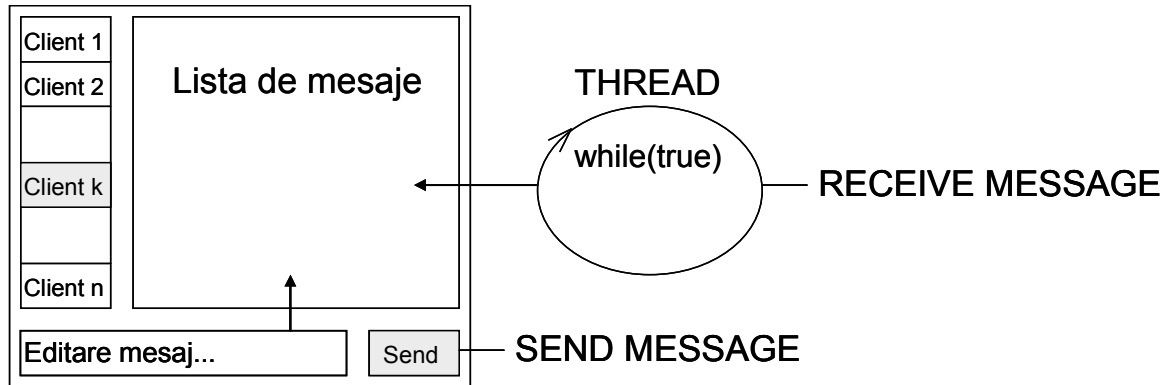


Figura 2. Aplicația client

Aplicația server, prezentată în Figura 3, păstrează clienții (soclu, nume, etc.) într-un vector. Pentru fiecare client conectat creează câte un fir de execuție (thread) care, într-o buclă infinită, recepționează mesajele de la clientul respectiv și apoi le trimite mai departe clientului destinație. Conectarea clienților se face la fel ca în aplicația prezentată în lucrare. La conectarea unui nou client serverul trebuie să anunțe toți clienții pentru actualizarea listelor de clienți. De aceea, în afară de mesajele propriu-zise vor exista și mesaje speciale (de administrare) pentru adăugarea clienților conectați respectiv ștergerea clienților deconectați. Diferențierea dintre cele două tipuri de mesaje se poate efectua prin utilizarea unui antet (header) în cadrul mesajului.

$VECTOR = \{ Client\ 1, Client\ 2, \dots, Client\ n \}$



Figura 3. Aplicația server

5. Conectarea la o bază de date MySQL



JDBC (Java Database Connectivity) este o interfață 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 și execută instrucțiunile SQL.

Proiectul conține două clase principale: `DBConnection` și `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* și pe care trebuie să fie pornit un server de MySQL. Pentru testare locală, în loc de IP se folosește *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 conține și metodele prin care se vor insera date în tabele sau metodele prin care se vor interoga aceste tabele. Toate aceste operații 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')");
```

Așa cum se poate observa în următoarea secvență, obținerea 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(ume_baza_de_date, ume_utilizator, parola);
}
catch(Exception exc){
    exc.printStackTrace();
}
```

Observație:

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



Aplicația C# prezentată în continuare [17], folosește driverul *MySQLDriverCS* [15] pentru conectarea la baza de date. Pentru ca driverul să poată fi folosit, trebuie introdusă în aplicație o referință la *MySQLDriverCS.dll* respectiv trebuie copiată librăria *libMySQL.dll* în directorul */bin*. Următorul exemplu prezintă conectarea la o bază de date MySQL folosind driverul *MySQLDriverCS*.

```
using MySQLDriverCS;
using System.Data;

MySQLConnection myConn;
MySQLDataReader MyReader = null;

try
{
    myConn = new MySQLConnection( new MySQLConnectionString("123.456.789.100",
                                                            "mydatabase",
                                                            "user",
                                                            "pass").AsString );

    myConn.Open();
    string sql = "SELECT * FROM Table";
    MySQLCommand cmd = new MySQLCommand(sql, myConn);

    MyReader = cmd.ExecuteReaderEx();

    while (MyReader.Read())
    {
        Console.WriteLine( MyReader["Product_Name"].ToString() );
    }

    MyReader.Close();
}
catch(Exception ee)
{
    Console.WriteLine( ee.ToString() );
}
```

```
finally
{
    if (MyReader != null && !MyReader.IsClosed)
    {
        MyReader.Close();
    }

    if (myConn != null && myConn.State == ConnectionState.Open)
    {
        myConn.Close();
    }
}
```

După cum se poate observa, obiectele `MySQLConnection` și `MySQLDataReader` sunt declarate înaintea blocului *try/catch*, și ele trebuie închise în cadrul blocului *finally*, asigurând astfel deconectarea de la baza de date, chiar și în situația în care apare o excepție în blocul *try/catch*. Astfel, se pot evita eventualele erori "Too many connections".

Aplicații

1. Să se implementeze o aplicație pentru conectarea la o baza de date MySQL care menține informații despre student. Aplicația trebuie să aibă o interfață care să permită vizualizarea (interogarea) tabelor unei baze de date și introducerea unor noi înregistrări.
2. Dezvoltați aplicația 1, prin introducerea funcțiilor care să permită modificarea și respectiv ștergerea înregistrărilor existente.

6. Aplicații web



O variantă de aplicație web Java este Servletul. Servleturile sunt componente independente de platformă ale aplicațiilor server și 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 distanță și terminând cu protocolul HTTP pentru cereri către servere web. Cu ajutorul servlet-urilor extindem deci funcționalitatea unei aplicații de tip server informațional (nu neapărat server HTTP), un prim domeniu de aplicare fiind, bineînțeles, 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*, și în funcție de asta, trebuie să aibă următoarea structură:

Tip fișier	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 funcție de directorul (Root sau Nume_Aplicatie) și respectiv stația 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

Aplicația următoare este formată dintr-un HTML care permite introducerea unui *nume* și 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 preluați 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>

```

Observații

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



Pentru realizarea unei aplicații web în ASP.NET sunt necesari următorii pași:

- Se pornește Visual Studio;
- Se creează un proiect nou ASP.NET Empty Project; Proiectul creat va conține inițial Properties, References și web.config. În aplicația de față 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 și dând click dreapta pe acesta apoi selectând din meniul care apare Add -> New Item;
- Din fereastra care apare se selectează Web Form și se dă un nume acesteia, apoi se apasă Add;
- Urmează editarea paginii adaugate.

Se adaugă în pagină următoarea secvență 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 recunoașteți până acum sunt:

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

acesta reprezentând un control asemănător unui 'input' de tipul 'text', cu diferența 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 aceleași caracteristici ca și controlul anterior, iar evenimentul 'OnClick' se va executa pe server.

Pentru ca pagina să funcționeze, 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 conține deja toate elementele inițiale. Metoda *Write* a acestuia va adăuga la începutul răspunsului ceea ce este trimis ca parametru, în cazul de față va fi : "Welcome x !", unde 'x' reprezintă conținutul căsuței text cu id-ul 'txtBoxNameResponse'. Pentru a testa pagina este suficient să apăsați F5 sau să dați click pe Debug -> Start Debugging, eventual Start Without Debugging. Pagina se va deschide în browser, fiind afișate pe pagină textul, o căsuță de introducere text și un buton. În momentul apăsării butonului, se va afișa un mesaj la începutul paginii care va conține și textul din căsuță (dacă există).

Cea de-a doua metodă va seta conținutul unui element din pagină cu textul ce se găsește în căsuța text.

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

Pentru ca metoda aceasta să funcționeze, 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 folosește 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 și 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 ați 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.innerHTML = retVal;
      }
    }
  }
}
</script>
```

Arată ca o funcție normală, exceptând obiectul 'PageMethods'. Acesta se regăsește în pagină în urma adăugării 'ScriptManager'-ului și a atributului precizat. El permite apelul metodelor de pe server. Metoda care este apelată se numește '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* și *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 diferențe față de cele prezentate anterior:

- Deține atributul [WebMethod]; va fi necesară folosirea unei noi librării pentru acest atribut (using System.Web.Services;) ce se adaugă deasupra declarației 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 funcția javascript ‘OnNameEnteredComplete’ unde se tratează rezultatul.

Aplicații

1. Implementați aplicația prezentată în lucrare.
2. Realizați o aplicație servlet care să permită gestionarea unei baze de date MySQL:
 - conectarea la baza de date MySQL;
 - introducerea unei noi inregistrari;
 - afișarea rezultatelor interogărilor într-un tabel.

Pentru această aplicație va fi creat HTML-ul, clasa *DBServlet* care extinde clasa *HttpServlet* și 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. Aplicații distribuite



Java RMI (Remote Method Invocation) permite implementarea aplicațiilor distribuite Java-to-Java, în care metodele obiectelor pot fi apelate la distanță [13]. Astfel, o aplicație client poate folosi la distanță obiectele unei aplicații server. O clasă poate fi instanțiată la distanță, dacă ea implementează o interfață *Remote*. Figura 4 prezintă structura generală a unei aplicații Java RMI.

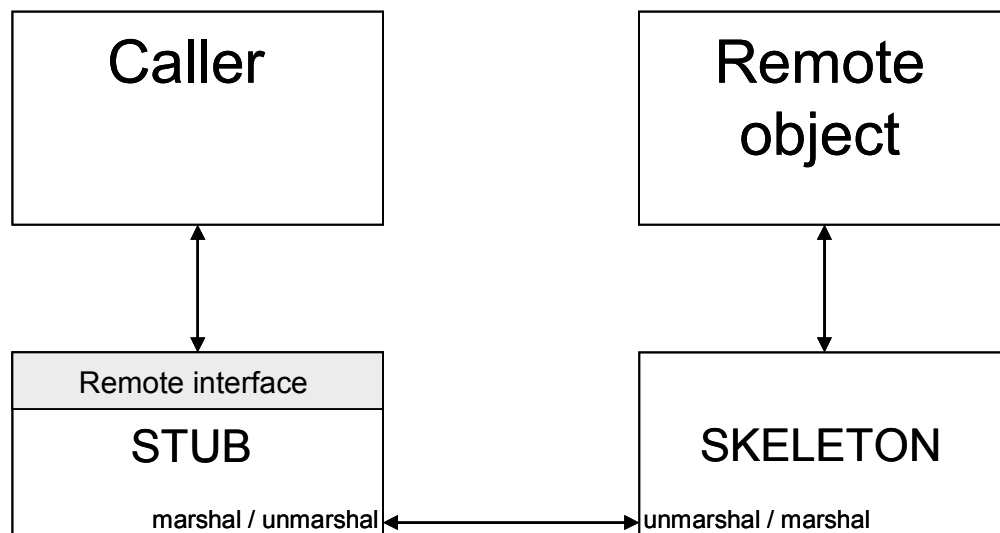


Figura 4. Structura unei aplicații Java RMI

La apelul unei metode la distanță are loc următoarea secvență de operații:

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

În această lucrare este prezentată o aplicație Java RMI în care clientul apelează la distanță metoda *sayHello* a aplicației server. Această metodă returnează mesajul “Hello World!”. Mesajul este preluat și afișat de client. Interfața *Hello* conține antetul metodei *sayHello* pe care dorim să o apelăm la distanță.

```

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

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

```

Clasa unui obiect apelabil la distanță trebuie să implementeze cel puțin o interfață *Remote*. Clasa *HelloImpl* implementează interfața *Hello* de tip *Remote*, și ea reprezintă aplicația server. Metoda *main* a serverului creează o instanță a clasei *HelloImpl*, rezultând astfel obiectul ale cărui metode pot fi apelate la distanță. Acestei instanțe 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 aplicațiile client, folosind metoda *Naming.lookup()*, prin numele care i s-a asociat în registrele RMI. Aplicația prezentată mai jos apelează la distanță metoda *sayHello* a serverului, și afișează mesajul returnat: "Hello World!".

```
import java.rmi.Naming;
import java.rmi.RMIException;

public class SayHello {

    public static void main(String args[]) {
        String message = "blank";
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMIException());
        }
        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
```

Aplicația 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
```



Pentru realizarea aplicațiilor .NET Remoting, cele două programe (server și client) necesită versiunea 3.5 de .NET Framework pentru a funcționa. Soluția este creată în Visual Studio 2010. Pentru a rula programele din Visual Studio se poate da un 'RUN' simplu deoarece atât clientul cât și serverul sunt setate ca startup projects. Programele pot fi rulate și prin fișierele 'runClient.bat' și 'runServer.bat'. Atât portul de ascultare al serverului și URL-ul pentru obiectul care este apelat la distanță cât și adresa la care se conectează clientul pot fi modificate din fișierul *app.config* al fiecărui executabil (aflat în bin/Debug/) fără a necesita recompilarea programelor. În continuare sunt prezentate: interfața (folosită pentru comunicație între client și server), serverul și clientul.

Interfața

```
// 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();
    }
}

```

Fișierul 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 encountered - {0}", rex.Message));
    }
    catch (MemberAccessException maex)
    {
      Console.WriteLine(string.Format("Error: MemberAccessException encountered - {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();
    }
}
```

Fișierul 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>
```

Aplicații

1. Să se implementeze aplicația prezentată în lucrare.
2. Să se implementeze o aplicație Java RMI sau .NET Remoting, pentru apelul la distanță al unor metode care să efectueze operațiile aritmetice elementare.
3. Să se integreze clientul de MySQL implementat anterior într-o aplicație Java RMI sau .NET Remoting.

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 secvențial poate fi paralelizat dacă secvența de operații elementare generată poate fi descompusă în p subsecvențe, în așa fel încât operațiile din fiecare subsecvență să poată fi executate concurențial de procesoarele disponibile. Există situații în care această descompunere este naturală, în general însă, algoritmul secvențial cel mai natural pentru rezolvarea unei probleme nu conduce la o împărțire în astfel de subsecvențe [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 și 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 adăugată variabila:

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

și trebuie adăugată 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 aplicațiilor MPI. După ce a fost instalat, poate fi pornit din Start->Run-> service.msc. Din lista de servicii trebuie selectat MPJ Daemon și 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 și mpj.jar prin click dreapta pe MPJ->Libraries, apoi Add JAR/Folder... În tab-ul Files, în directorul proiectului MPJHello trebuie creat fișierul mpj.conf. Acesta va conține 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 mașină/proces trebuie creată o configurație pentru rulare. Pe tab-ul Projects, click dreapta pe MPJHello -> Set Configuration -> Customize și din panoul din partea dreaptă, se selectează New. Apoi se selectează numele. De exemplu mpj0. În căsuța de text Arguments se va specifica: 0 mpj.conf niodev , unde prima cifră este numărul procesului. Pentru fișierul mpj.conf de mai sus, ar trebui create 3 configurații:

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

În cazul în care se dorește rularea pe mai multe stații, pentru fiecare stație se va crea un fișier mpj.conf ce va conține IP-urile și rangul tuturor stațiilor.

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 începând de la care dorim să se trimită / primească datele, Count este numărul de elemente trimise / primite, Type este tipul de date (ex. MPI.CHAR, MPI.INT, MPI.OBJECT, etc.), Dest este rangul (identificatorul) procesului destinație, Src este rangul (identificatorul) procesului sursă, iar Tag este un identificator unic al mesajului (cel din Recv trebuie să corespundă cu cel din funcția Send aferentă).

În continuare se prezintă aplicația *PingPong* în care are loc un transfer de mesaje între *p* procese, care pot fi pornite pe calculatoare diferite. Procesul 0 primește și afișează numele stațiilor pe care au fost pornite celelalte procese. După cum se poate observa în codul sursă al aplicației, prezentat mai jos, aceste transferuri de mesaje sunt efectuate prin instrucțiunile *Send* și *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 configurațiile de rulare. Mai întâi trebuie să ne asigurăm că proiectul curent este setat ca MainProject:

Run -> Set Main Project -> MPJHello.

Trebuie selectată prima configurație:

Run->Set Project Configuration-> mpj0

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



Limbajul C# permite implementarea aplicațiilor paralele printr-o librărie MPI [14]. Pentru înțelegerea structurii unui program paralel implementat în MPI.NET, această lucrare prezintă aplicația *PingPong* în care are loc un transfer de mesaje între p procese, care pot fi pornite pe calculatoare diferite. Procesul 0 primește și afișează numele stațiilor pe care au fost pornite celelalte procese. După cum se poate observa în codul sursă al aplicației, prezentat mai jos, aceste transferuri de mesaje sunt efectuate prin instrucțiunile *Send* și *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);
                }
            }
        }
    }
}
```


în folderul Microsoft Compute Cluster Pack\Bin\ al fiecărei stații folosite și se pornește executabilul smpd pe fiecare stație, 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 conține fiecare stație pe câte un rând.

Exemple

Comanda efectuată pe stație10:

```
mpiexec -hosts 4 stație10 stație07 stație09 stație08 PingPong
```

pornește patru procese pe patru stații diferite (stație10, stație07, stație09 și stație08). **Prima stație trebuie să fie cea locală!**

Comanda efectuată pe stație10:

```
mpiexec -hosts 4 stație10 3 stație07 2 stație09 5 stație08 6 PingPong
```

folosește patru stații, pornind trei procese pe stația locală (stație10), două procese pe stație07, cinci procese pe stație09, respectiv șase procese pe stație08. **Și în acest caz prima stație trebuie să fie cea locală!**

Aplicații

1. Să se implementeze și să se ruleze aplicația *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 maximum dintr-un tablou.
4. Să se implementeze un algoritm paralel care determină minimum 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 înmulțirea matricilor.
8. Să se implementeze și să se ruleze algoritmul Quicksort paralel.
9. Să se implementeze și 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, București, 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, București, 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 și proiectarea algoritmilor: o abordare pragmatică prin aplicații Java*, Editura Techno Media, ISBN 978-606-8030-81-4, 2010.
- [10] Mihiu 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] MySQLDriverCS, <http://sourceforge.net/projects/mysqldrivercs>.
- [16] Orfali R., Harkey D., *Client/Server Programming with Java and Corba*, 2nd Edition, J. Wiley & Sons, New York, 1998.
- [17] Primary Objects, <http://www.primaryobjects.com/CMS/Article68.aspx>.