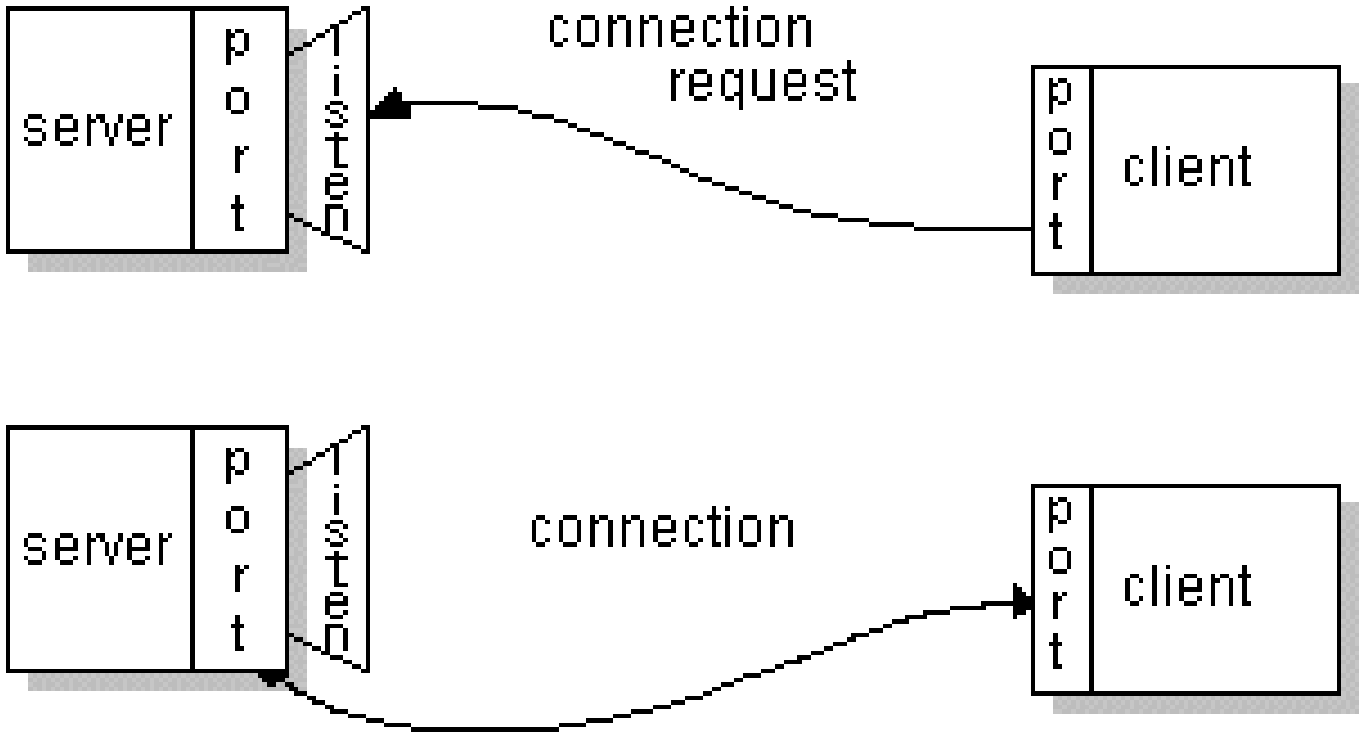
Java - NET

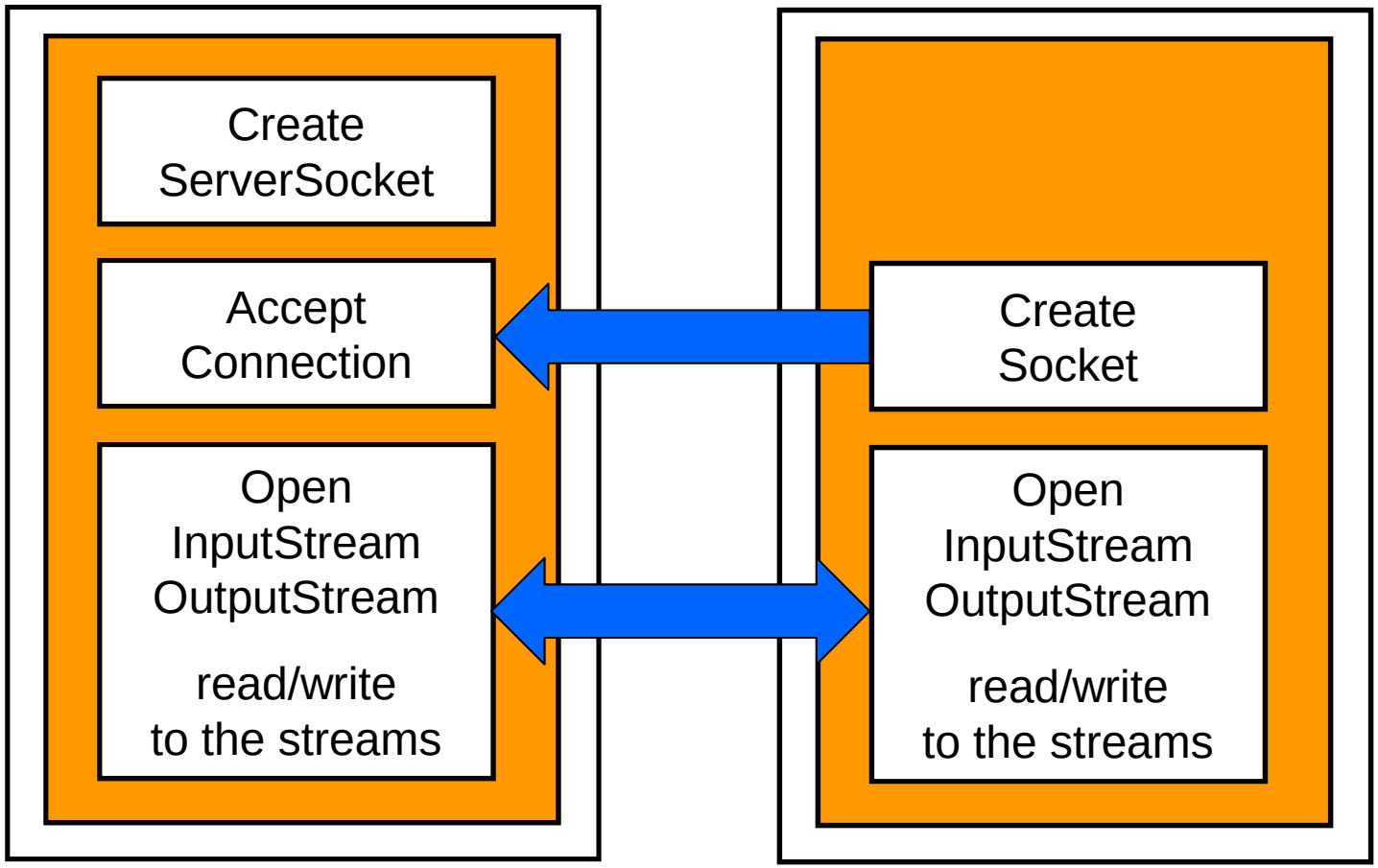
java.net

- **URL** (*Uniform Resource Locator*) представља референцу на неки ресурс на Интернету.
- **InetAddress** – Представља Интернет адресу. Пружа методе за добијање имена рачунара ...
- **Socket** (прикључница) – Крајња тачка комуникације
- **ServerSocket** – Серверска прикључница која слуша клијентске захтеве
- **DatagramSocket, DatagramPacket, MulticastSocket, DatagramPacket, ...**

Клијент-Сервер архитектура



Клијент-Сервер архитектура



Серверска страна

Подизање серверског програма на неком порту:

```
try {
    serverSocket = new ServerSocket(port);
} catch (IOException e) {
    System.exit(-1);
}
```

Серверска страна

Успостављање везе сервера са клијентом:

```
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
} catch (IOException e) {
    System.out.println("Accept failed");
    System.exit(-1);
}
```

Добијање улазног тока

Добијање улазног тока података из прикључнице:

```
BufferedReader in = new BufferedReader(
        new InputStreamReader(
                clientSocket.getInputStream()));
```

Добијање излазног тока

Добијање излазног тока података из прикључнице:

```
OutputStream out = server.getOutputStream();
PrintWriter outp = new PrintWriter(
                        new OutputStreamWriter(out), true);
```

Серверска страна

```
import java.net.*;
public class Server {
    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[0]);
            ServerSocket server = new ServerSocket(port);
            Socket client = server.accept();
            getStreams(client);
            …
             server.close();
        } catch (Exception ex) { ex.printStackTrace();}
    }
}
```

Сервер са више нити

```
import java.net.*;
public class Server {
    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[0]);
            ServerSocket server= new ServerSocket(port);
            while (true) {
                try {
                    Socket client = server.accept();
                    new WorkingThread(client).start();
                }catch (Exception ex1 ) { }
            }
             server.close();
        } catch (Exception ex) { ex.printStackTrace();}
    }
}
```

Клијентска страна

Повезивање клијента са серверским програмом на неком порту:

```
Socket server = null;
try {
    Socket server = new Socket(host, port);
} catch (IOException e) {
    System.exit(-1);
}
```

Клијентска страна

```
import java.net.*;
public class Client {
    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[1]);
            String host = args[0];
            Socket server = new Socket(host, port);
            getStreams(client);
            …
            server.close();
        } catch (Exception ex) {ex.printStackTrace();}
    }
}
```

Коришћење класе URL

```
import java.net.*;
import java.io.*;

public class URLReader {
    public static void main(String[] args) throws Exception {
    URL site = new URL("http://www.yahoo.com/");

    URLConnection uc = site.openConnection();

    BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                 uc.getInputStream()));
    String inputLine;
    while ((inputLine = in.readLine()) != null){
            System.out.println(inputLine);
    }
    in.close();
    }
}
```

Састав класе Socket

Modifier and Type	Method and Description
`InputStream`	`getInputStream()` Returns an input stream for this socket.
`OutputStream`	`getOutputStream()` Returns an output stream for this socket.
`SocketChannel`	`getChannel()` Returns the unique `SocketChannel` object associated with this socket, if any.
`InetAddress`	`getInetAddress()` Returns the address to which the socket is connected.
`int`	`getPort()` Returns the remote port number to which this socket is connected.
`InetAddress`	`getLocalAddress()` Gets the local address to which the socket is bound.
`int`	`getLocalPort()` Returns the local port number to which this socket is bound.
`int`	`getSoTimeout()` Returns setting for `SO_TIMEOUT`.

Састав класе Socket

Modifier and Type	Method and Description
void	**bind(SocketAddress** bindpoint) Binds the socket to a local address.
void	**close()** Closes this socket.
void	**connect(SocketAddress** endpoint) Connects this socket to the server.
void	**connect(SocketAddress** endpoint, int timeout) Connects this socket to the server with a specified timeout value.
boolean	**isBound()** Returns the binding state of the socket.
boolean	**isClosed()** Returns the closed state of the socket.
boolean	**isConnected()** Returns the connection state of the socket.
boolean	**isInputShutdown()** Returns whether the read-half of the socket connection is closed.
boolean	**isOutputShutdown()** Returns whether the write-half of the socket connection is closed.

...

Задаци

Chat

У програмском језику Јава је потребно написати скуп класа за интерактивну комуникацију између корисника (CHAT). Потребно је реализовати решење код кога корисник може да пошаље већи број порука без чекања одговора на претходне.

Chat

```
import java.net.*;

public class Server {

    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[0]);
            ServerSocket listener = new ServerSocket(port);
            Socket client = listener.accept();
            Chat c = new Chat(client);
            c.communicate();
            listener.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

}
```

Chat

```
import java.net.*;

public class Client {

    public static void main(String[] args) {
        try {
            int port = Integer.parseInt(args[1]);
            String host = args[0];
            Socket server = new Socket(host, port);
            Chat s = new Chat(server);
            s.communicate();
            server.close();
        } catch (Exception ex) {
            System.out.println(ex);
            ex.printStackTrace();
        }
    }
}
```

Chat

```
import java.net.*;
public class Chat {
    Socket client;
    public Chat(Socket client) {
        this.client = client;
    }
    public void communicate() {
        try {
            ReadThread read = new ReadThread(client);
            WriteThread write = new WriteThread(client);
            read.start();
            write.start();
            read.join();
//          write.interrupt();
            write.join();
        } catch (InterruptedException ex) { ex.printStackTrace();}
    }
}
```

Chat

```
import java.io.*;
import java.net.*;

public class ReadThread extends Thread {

    Socket client;
    InputStream in;
    BufferedReader pin;

    public ReadThread(Socket client) {
        try {
            this.client = client;
            this.in = client.getInputStream();
            pin = new BufferedReader(new InputStreamReader(in));
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

Chat

```
    public void run() {
        try {
            String s;
            while ((s = pin.readLine()) != null) {
                System.out.println("> " + s);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            try {
                pin.close();
                in.close();
                client.close();
            } catch (IOException ex1) {
                ex1.printStackTrace();
            }
        }
    }
}
```

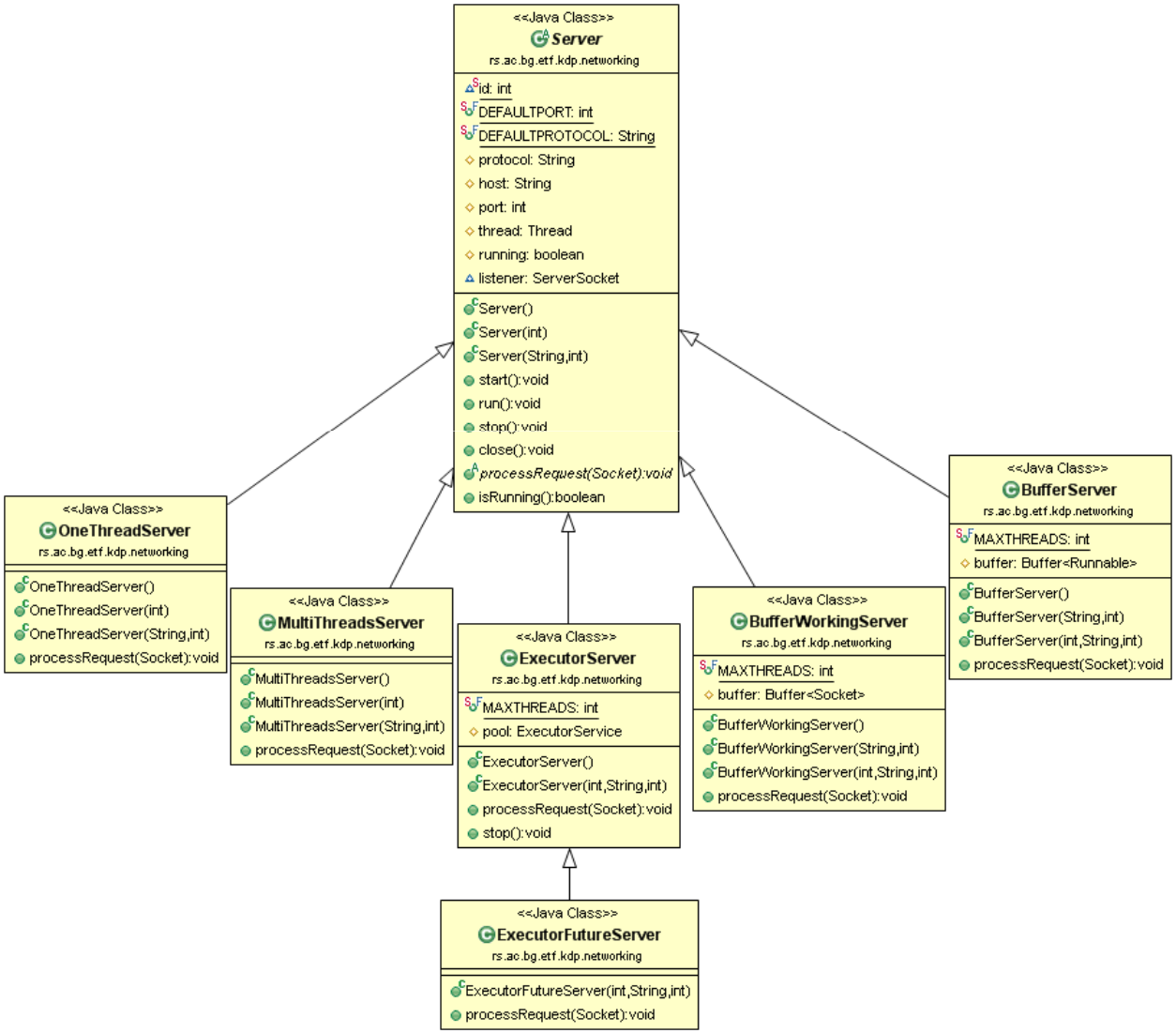
Chat

```
import java.io.*;
import java.net.*;

public class WriteThread extends Thread {
    Socket client;

    OutputStream out;

    PrintWriter pout;

    public WriteThread(Socket client) {
        try {
            this.client = client;
            this.out = client.getOutputStream();
            pout = new PrintWriter(out, true);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

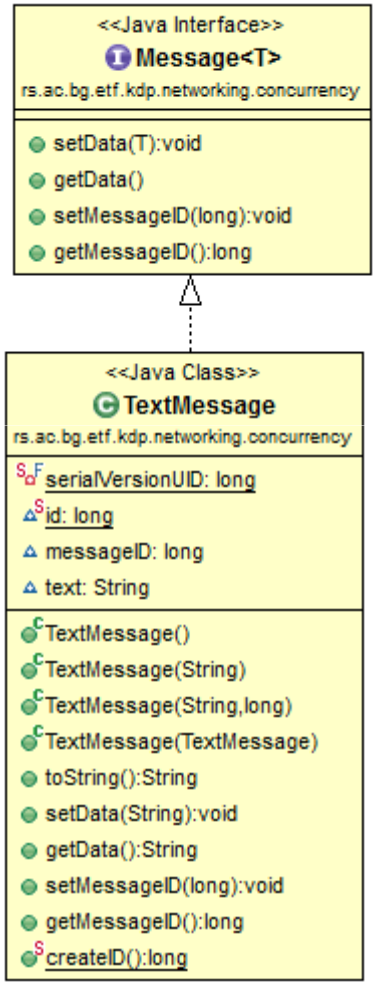
Chat

```
public void run() {
        try {
            InputStream is = System.in;
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String s;
            while (!pout.checkError() && ((s = br.readLine()) != null)) {
                    pout.println(s);
            }
        } catch (Exception ex) {
        } finally {
            try {
                    pout.close();
                    out.close();
                    client.close();
            } catch (IOException ex1) {
            }
        }
    }
}
```

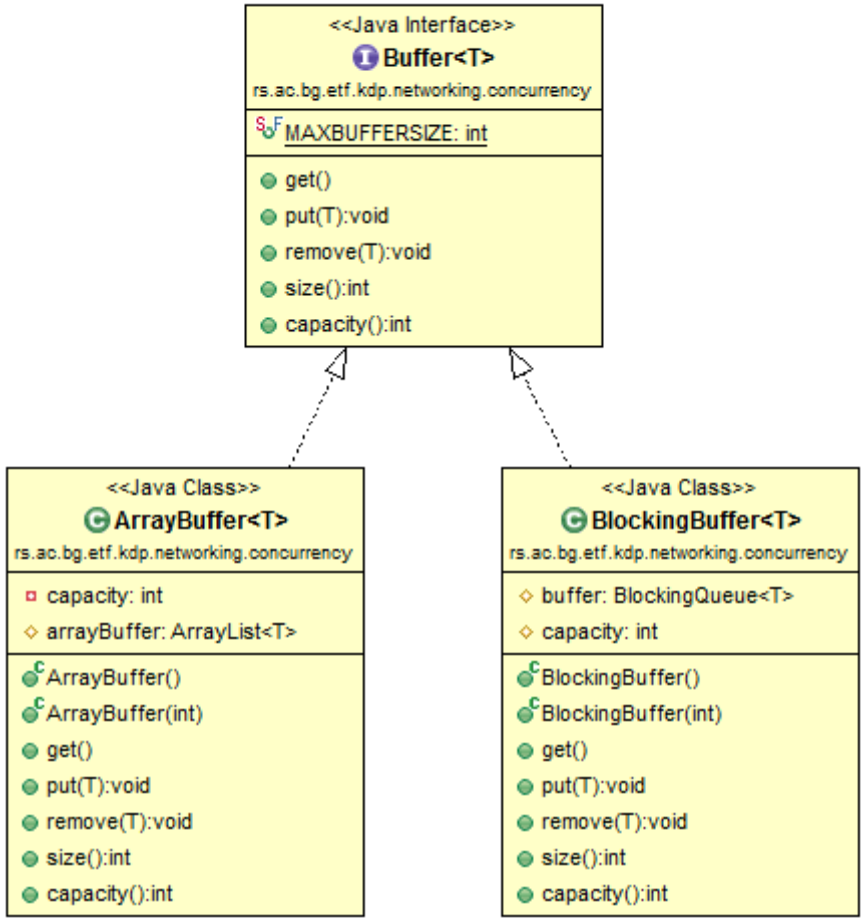
Клијент-сервер



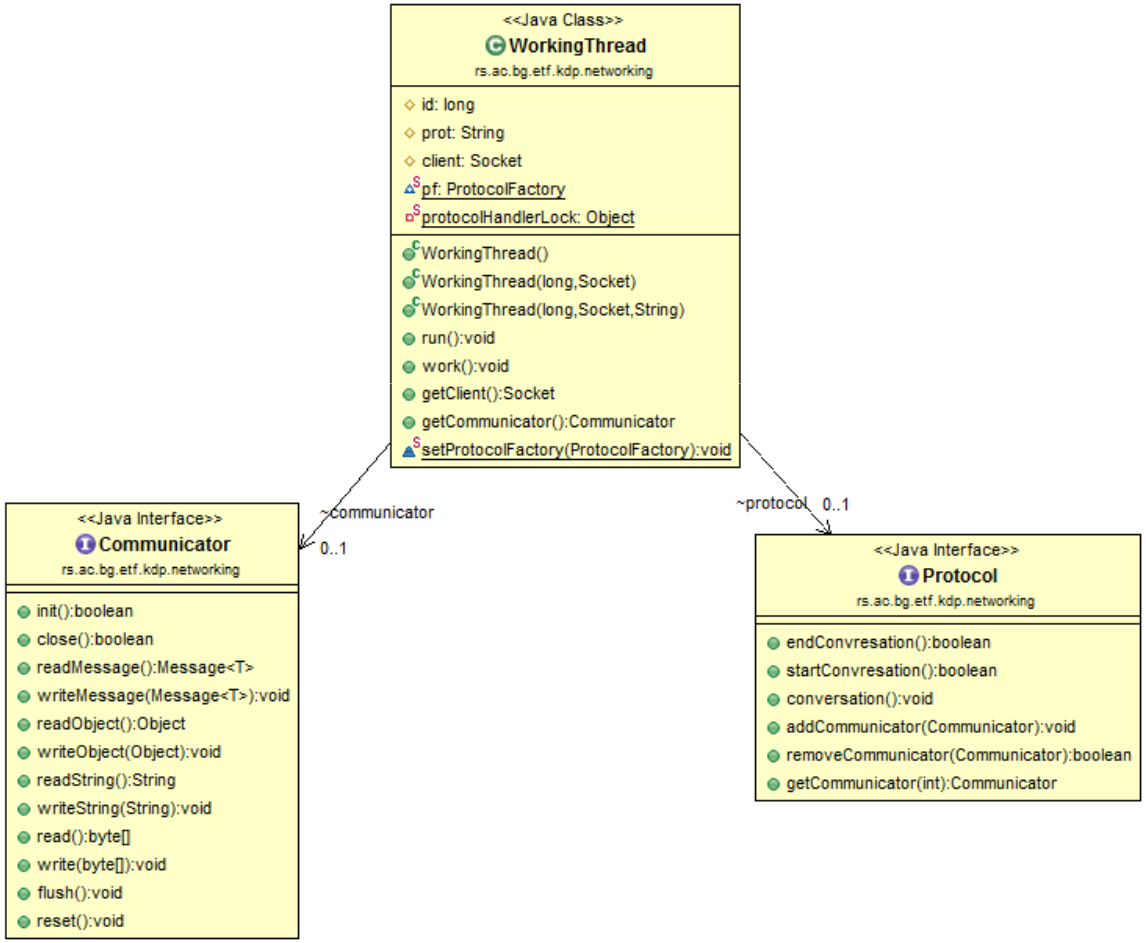
Клијент-сервер



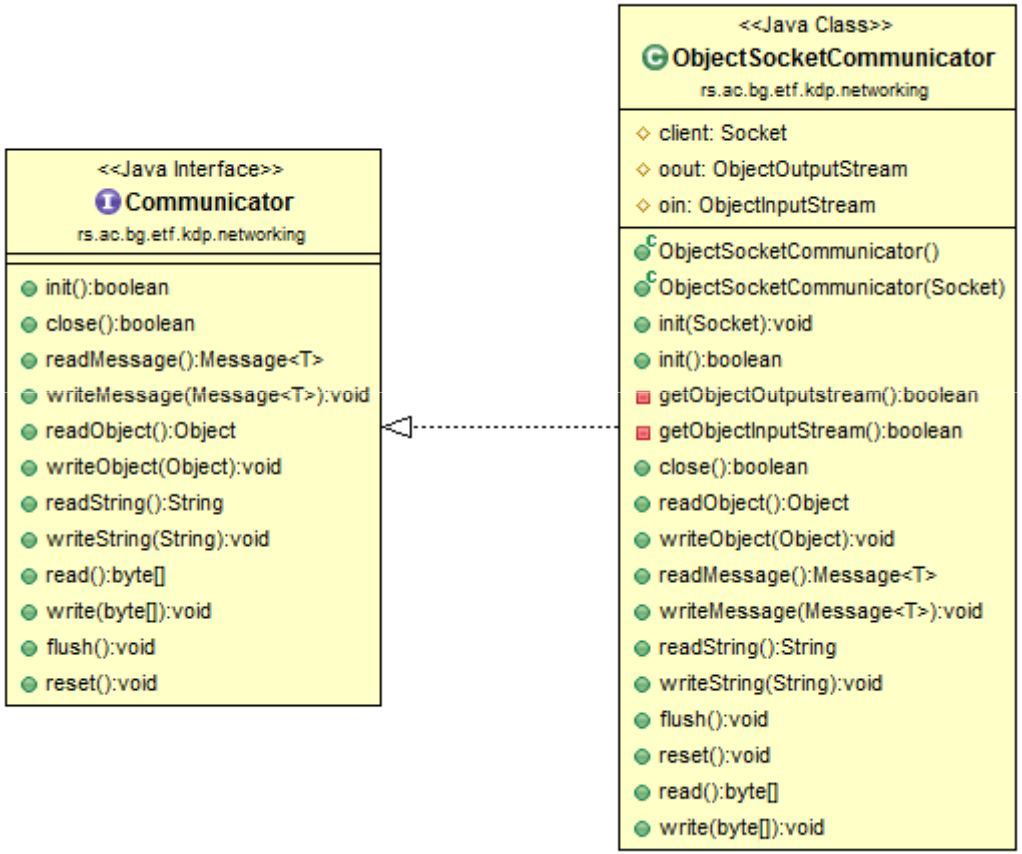
Клијент-сервер



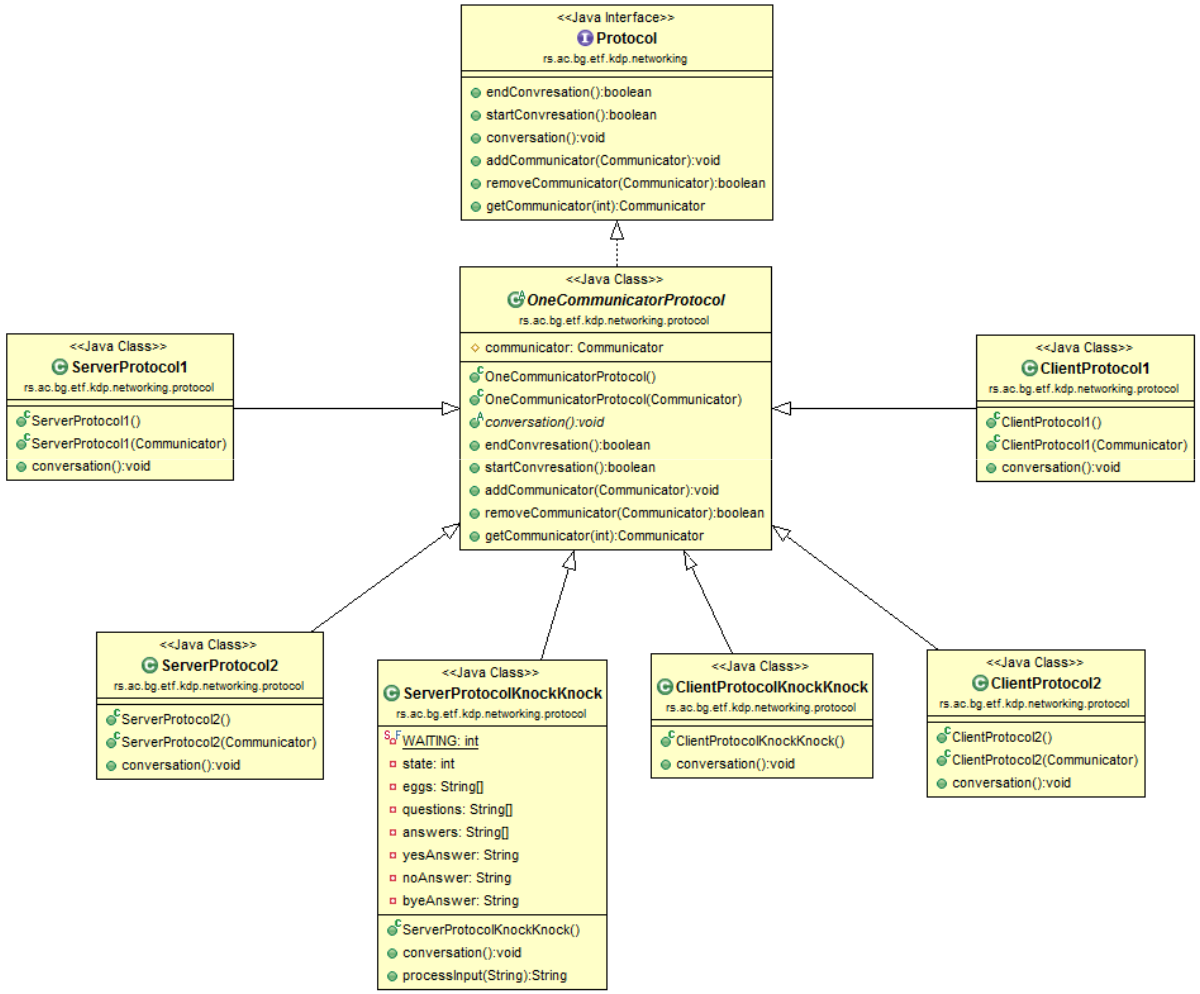
Клијент-сервер



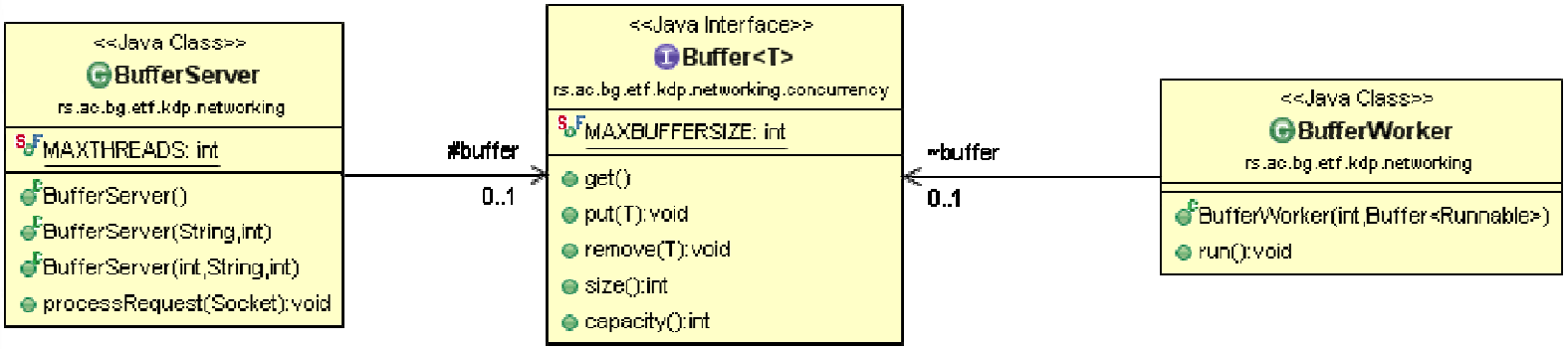
Клијент-сервер

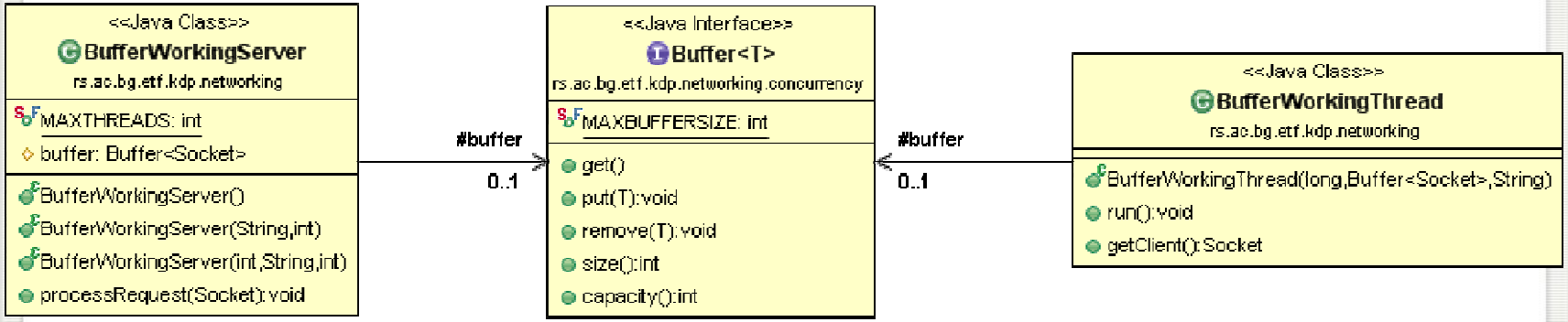


Клијент-сервер



Клијент-сервер





Клијент-сервер

```
import java.net.*;
public abstract class Server implements Runnable {
    static int id = 0;
    public static final int DEFAULTPORT = -1;
    public static final String DEFAULTPROTOCOL = "ServerProtocol1";
    protected String protocol;
    protected String host;
    protected int port;

    public Server(String protocol, int port) {
        this.port = port;
        this.protocol = protocol;
        thread = null;
        running = false;
    }
```

Клијент-сервер

```
protected ServerSocket listener  = null;
public void run() {
        try {
                listener = new ServerSocket(port);
                while (running) {
                        try {
                                Socket client = listener.accept();
                                processRequest(client);
                        } catch (Exception e) {
                        }
                }
        } catch (Exception ex) {
                ex.printStackTrace();
        } finally{
                close();
        }
}
```

Клијент-сервер

```
    public abstract void processRequest(Socket client);

    public void start() {
            if (thread == null) {
                    thread = new Thread(this, "Server");
                    running = true;
                    thread.start();
            }
    }
    public void stop() {
            running = false;
//          thread.interrupt();
            close();
    }
    public void close() {
    try {
            listener.close();
    } catch (IOException e) {}
}
```

Клијент-сервер

```
import java.net.*;
public class OneThreadServer extends Server{

    public OneThreadServer(String protocol, int port) {
        super(protocol, port);
    }
    public void processRequest (Socket client){
        try{
            WorkingThread tserver =
                new WorkingThread(id++, client, protocol);
//          tserver.start();
//          tserver.join();
            tserver.run();
        }catch (Exception ex ) {}
    }
}
```

Клијент-сервер

```
public class MultiThreadsServer extends Server {

    ...

    public void processRequest(Socket client) {
            new WorkingThread(id++, client, protocol).start();
    }

}
```

ExecutorService

boolean	`awaitTermination(long timeout, TimeUnit unit)` Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
<T> List<Future<T>>	`invokeAll(Collection<? extends Callable<T>> tasks)` Executes the given tasks, returning a list of Futures holding their status and results when all complete.
<T> List<Future<T>>	`invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first.
<T> T	`invokeAny(Collection<? extends Callable<T>> tasks)` Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.
<T> T	`invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses.
boolean	`isShutdown()` Returns `true` if this executor has been shut down.
boolean	`isTerminated()` Returns `true` if all tasks have completed following shut down.
void	`shutdown()` Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.
List<Runnable>	`shutdownNow()` Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.
<T> Future<T>	`submit(Callable<T> task)` Submits a value-returning task for execution and returns a Future representing the pending results of the task.
Future<?>	`submit(Runnable task)` Submits a Runnable task for execution and returns a Future representing that task.
<T> Future<T>	`submit(Runnable task, T result)` Submits a Runnable task for execution and returns a Future representing that task.
void	`execute(Runnable command)` Executes the given command at some time in the future.

Клијент-сервер

```
import java.net.*;
import java.util.concurrent.*;

public class ExecutorServer extends Server {
    public static final int MAXTHREADS = 10;
    protected ExecutorService pool;

    public ExecutorServer(int numOfThreads, String protocol, int port) {
        super(protocol, port);
        pool = Executors.newFixedThreadPool(numOfThreads);
    }

    public void processRequest(Socket client) {
        pool.execute(new WorkingThread(0, client, protocol));
    }
}
```

Клијент-сервер

```
public void stop() {
    super.stop();

    pool.shutdown();
    try {
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
            pool.shutdownNow();
            if (!pool.awaitTermination(60, TimeUnit.SECONDS))
                System.err.println("Pool did not terminate");
    }
    } catch (InterruptedException ie) {
        pool.shutdownNow();
        Thread.currentThread().interrupt();
    }
}
```

Callable, Future

Modifier and Type	Method and Description
V	call() Computes a result, or throws an exception if unable to do so.

Modifier and Type	Method and Description
boolean	cancel(boolean mayInterruptIfRunning) Attempts to cancel execution of this task.
V	get() Waits if necessary for the computation to complete, and then retrieves its result.
V	get(long timeout, TimeUnit unit) Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
boolean	isCancelled() Returns true if this task was cancelled before it completed normally.
boolean	isDone() Returns true if this task completed.

Клијент-сервер

```
import java.net.*;
import java.util.concurrent.*;
public class ExecutorFutureServer extends ExecutorServer {
    public void processRequest(final Socket client) {
        Future<Void> future = pool.submit(new Callable<Void>() {
            public Void call() {
                WorkingThread wt = new WorkingThread(0, client, protocol);
                wt.work();
                return null;
            }
        });
/*
        try {
            future.get();
        } catch (Exception ex) {
        }
*/
    }
}
```

Клијент-сервер

```
public class BufferServer extends Server {
    public static final int MAXTHREADS = 10;
    protected Buffer<Runnable> buffer;
    public BufferServer(int num, String protocol, int port) {
        super(port);
        buffer = new ArrayBuffer<Runnable>();
        for (int i = 0; i < num; i++) {
            Thread t = new BufferWorker(i, buffer);
            t.setDaemon(true);
            t.start();
        }
    }
    public void processRequest(Socket client) {
        buffer.put(new WorkingThread(id++, client, protocol));
    }
}
```

Клијент-сервер

```
public class BufferWorker extends Thread {
    Buffer<Runnable> buffer;

    public BufferWorker(int id, Buffer<Runnable> buffer) {
        super("BufferWorker" + id);
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            Runnable r = buffer.get();
            r.run();
        }
    }
}
```

Клијент-сервер

```
public interface Buffer<T> {
    public static final int MAXBUFFERSIZE = 150;
    public T get();
    public void put(T data);
    public void remove(T data);
    public int size();
    public int capacity();
}
```

Клијент-сервер

```
import java.util.*;
public class ArrayBuffer<T> implements Buffer<T> {
    private int capacity;
    protected List<T> arrayBuffer;
    public ArrayBuffer() {
        this(MAXBUFFERSIZE);
    }
    public ArrayBuffer(int newCapacity) {
        if ((newCapacity > 0) && (newCapacity <= MAXBUFFERSIZE))
            capacity = newCapacity;
        else
            capacity = MAXBUFFERSIZE;
        arrayBuffer = new ArrayList<T>();
    }
```

Клијент-сервер

```
public synchronized T get() {
        while (arrayBuffer.size() == 0) {
                try {
                        wait();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
        T data = arrayBuffer.remove(0);
        notifyAll();
        return data;
    }
```

Клијент-сервер

```
public synchronized void put(T value) {
        while (arrayBuffer.size() == capacity) {
                try {
                        wait();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
        arrayBuffer.add(value);
        notifyAll();
    }
```

Клијент-сервер

```
public synchronized void remove(T data) {
        try {
                int index = arrayBuffer.indexOf(data);
                if (index < 0)
                        return;
                arrayBuffer.remove(index);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
    }
    public synchronized int size() {
        return arrayBuffer.size();
    }
    public synchronized int capacity() {
        return capacity;
    }
```

Клијент-сервер

```
public interface Message<T> extends Serializable {
    public void setData(T data);
    public T getData();
    public void setMessageID(long messageID);
    public long getMessageID();
}
```

Клијент-сервер

```
public class TextMessage implements Message<String>{
    static long id = 0;

    long messageID;
    String text;

    public TextMessage(String s){
        messageID = createID() ;
        text = s;
    }

    public TextMessage(TextMessage m){
        this(m.getText(), m.getMessageID());
    }

    public static synchronized long createID() {
        return id++;
    }
    ...
```

Клијент-сервер

```
    public void setData(String data) {
            text = (String)data;
    }

    public String getData() {
            return text;
    }
    public void setMessageID(long messageID) {
            this.messageID = messageID;
    }

    public long getMessageID() {
            return messageID;
    }
}
```

Клијент-сервер

```
import java.net.*;
public class WorkingThread extends Thread {
    protected long id;
    protected String protocol;
    protected Socket client;

    public WorkingThread(long id, Socket client, String protocol) {
        super("Working Thread " + id);
        this.id = id;
        this.client = client;
        this.protocol = protocol;
    }
```

Клијент-сервер

```
public void run() {
        work();
}
public void work() {
    Communicator communicator = getCommunicator();
    try {
        Protocol prot = pf.createProtocol(protocol);
        if (protocol == null) return;
        communicator.init();
        prot.addCommunicator(communicator);
        prot.conversation();
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally{
        communicator.close();
    }
}
```

Клијент-сервер

```
public Socket getClient() {
        System.out.println("getClient " + this.getClass());
        return client;
}
public Communicator getCommunicator() {
        return new ObjectSocketCommunicator(getClient());
}
```

Клијент-сервер

```
public interface Communicator {
    public boolean init();
    public boolean close();
    public <T> Message<T> readMessage() throws
    CommunicationException;
    public <T> void writeMessage(Message<T> data)
                    throws CommunicationException;
    public Object readObject() throws CommunicationException;
    public void writeObject(Object data) throws
    CommunicationException;
    public String readString() throws CommunicationException;
    public void writeString(String data) throws CommunicationException;
    public byte[] read() throws CommunicationException;
    public void write(byte[] data) throws CommunicationException;
    public void flush() throws CommunicationException;
    public void reset() throws CommunicationException;
}
```

Клијент-сервер

```
import java.net.*;
import java.io.*;

public class ObjectSocketCommunicator implements Communicator {
    protected Socket client;
    protected ObjectOutputStream oout;
    protected ObjectInputStream oin;

    public ObjectSocketCommunicator() {
    }

    public ObjectSocketCommunicator(Socket client) {
        this.client = client;
    }
```

Клијент-сервер

```
public void init(Socket client) {
        this.client = client;
        init();
}

public boolean init() {
        boolean ok = true;
        ok = getObjectOutputstream();
        ok |= getObjectInputStream();
        return ok;
}
```

Клијент-сервер

```
public boolean close() {
        boolean ok = true;
        try {
                oin.close();
        } catch (IOException ex) {
                ok = false;
        }
        try {
                oout.close();
        } catch (IOException ex) {
                ok = false;
        }
        try {
                client.close();
        } catch (IOException ex) {
                ok = false;
        }
        return ok;
    }
```

Клијент-сервер

```
public Object readObject() throws CommunicationException {
        try {
                Object m = oin.readObject();
                return m;
        } catch (Exception ex) {
                ex.printStackTrace();
                throw new CommunicationException();
        }
}
public void writeObject(Object m) throws CommunicationException {
        try {
                oout.writeObject(m);
        } catch (Exception ex) {
                ex.printStackTrace();
                throw new CommunicationException();
        }
}
```

Клијент-сервер

```
@SuppressWarnings("unchecked")
public <T> Message<T> readMessage() throws
CommunicationException {
        return (Message<T>) this.readObject();
}
public <T> void writeMessage(Message<T> m) throws
CommunicationException {
        this.writeObject(m);
}
public String readString() throws CommunicationException {
        return (String) this.readObject();
}
public void writeString(String s) throws CommunicationException {
        this.writeObject(s);
}
```

Клијент-сервер

```
public void flush() throws CommunicationException {
        try {
                oout.flush();
        } catch (Exception e) {
                e.printStackTrace();
                throw new CommunicationException();
        }
}
public void reset() throws CommunicationException {
        try {
                oout.reset();
                oin.reset();
        } catch (Exception e) {
                e.printStackTrace();
                throw new CommunicationException();
        }
}
```

Клијент-сервер

```
    public byte[] read() throws CommunicationException {
        byte[] b = new byte[1024];
        try {
            int num = oin.read(b);
            b = Arrays.copyOf(b, num);
        } catch (Exception e) {
            e.printStackTrace();
            throw new CommunicationException();
        }
        return b;
    }
    public void write(byte[] data) throws CommunicationException {
        try {
            oout.write(data);
        } catch (Exception e) {
            e.printStackTrace();
            throw new CommunicationException();
        }
    }
}
```

Клијент-сервер

```
public interface Protocol {
    public boolean endConvresation();
    public boolean startConvresation();
    public void conversation();
    public void addCommunicator(Communicator communicator);
    public boolean removeCommunicator(Communicator communicator);
    public Communicator getCommunicator(int id);
}
```

Клијент-сервер

```
public class ProtocolFactory {
    public Protocol createProtocol(String protocol) {
      Protocol prot = null;
      String protocolPackageRoot =
            Protocol.class.getPackage().getName();
      try {
            String clsName = protocolPackageRoot + ".protocol." + protocol;
            Class<?> cls = null;
            try {
                        cls = Class.forName(clsName);
            } catch (ClassNotFoundException e) { }
            if (cls != null) {
                        prot = (Protocol) cls.newInstance();
            }
       } catch (Exception e) { }
       return prot;
    }
}
```

Клијент-сервер

```
static ProtocolFactory pf;
private static Object protocolHandlerLock = new Object();
static void setProtocolFactory(ProtocolFactory protocol) {
        synchronized (protocolHandlerLock) {
            if (pf != null) {
                throw new Error("factory already defined");
            }
            SecurityManager security = System.getSecurityManager();
            if (security != null) {
                security.checkSetFactory();
            }
            pf = protocol;
        }
}
static {
        setProtocolFactory(new ProtocolFactory());
}
```

Клијент-сервер

```
public abstract class OneCommunicatorProtocol implements Protocol{
    Communicator c;

    public OneCommunicatorProtocol(Communicator c){
        this.c = c;
    }

    ...

    public abstract void conversation();
```

Клијент-сервер

```
    public void addCommunicator(Communicator c){
        this.c = c;
    }

    public boolean removeCommunicator(Communicator c) {
        if(this.c.equals(c)){
            c = null;
            return true;
        }
        return false;
    }
    public Communicator getCommunicator(int id){
        return c;
    }
}
```

Клијент-сервер

```
public class ServerProtocolKnockKnock extends
    OneCommunicatorProtocol {
    private static final int WAITING = 0;
    private int state = WAITING;
    private String[] eggs = { "Belo", "Plavo", "Crveno", "Zeleno",
    "Zuto","Crno" };
    private String[] questions = { "Kuc Kuc", "Djavo 's neba", "Jedno jaje" };
    private String[] answers = { "Koje?", "Sta Vam treba?", "Koje boje?" };
    private String yesAnswer = "Ima";
    private String noAnswer = "Nema ";
    private String byeAnswer = "Zdravo";.

    public ServerProtocolKnockKnock() {
            super();
    }
```

Клијент-сервер

```
public void conversation() {
        try {
                String outputLine = null;
                String inputLine = null;
                while ((inputLine = communicator.readString()) != null) {
                        System.out.println("Klient: " + inputLine);
                        outputLine = processInput(inputLine);
                        communicator.writeString(outputLine);
                        System.out.println("Server: " + outputLine);
                        if (outputLine.equals("Zdravo"))
                                break;
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                communicator.close();
        }
    }
```

Клијент-сервер

```
public String processInput(String theInput) {
        String theOutput = null;
        if (theInput == null)
                theInput = "";
        switch (state) {
        case 0:
        case 1:
        case 2:
                if (theInput.equalsIgnoreCase(questions[state])) {
                        theOutput = answers[state];
                        state++;
                } else {
                        theOutput = byeAnswer;
                        state = 0;
                }
                break;
        case 3:
                theOutput = noAnswer + theInput + " " + answers[2];
                for (int i = 0; i < eggs.length; i++) {
                        if (theInput.equalsIgnoreCase(eggs[i])) {
                                theOutput = yesAnswer;
                                state++;
                                break;
                        }
                }
                break;
        default:
                theOutput = byeAnswer;
                break;
        }
        return theOutput;
    }
}
```

Клијент-сервер

```
import java.io.*;
public class ClientProtocolKnockKnock extends
    OneCommunicatorProtocol {
    public ClientProtocolKnockKnock() {
        super();
    }
```

Клијент-сервер

```
public void conversation() {
        String fromServer = ""; String fromUser = "";
        try {
        BufferedReader stdIn = new BufferedReader(new
    InputStreamReader(System.in));
                System.out.print("Klient: ");
                fromUser = stdIn.readLine();
                communicator.writeString(fromUser);
                while ((fromServer = communicator.readString()) != null) {
                        System.out.println("Server: " + fromServer);
                        if (fromServer.equals("Zdravo"))        break;
                        System.out.print("Klient: ");
                        fromUser = stdIn.readLine();
                        if (fromUser != null) {
                                communicator.writeString(fromUser);
                        }
                }
        } catch (Exception ex) {
        } finally {
                communicator.close();
        }
    }
```

Клијент-сервер

```
import java.net.*;
public class ClientTest1{
    static final int port = 88;
    static final String host = "127.0.0.1";
    public static void main(String [] args) {
        for(int i = 0; i < 10; i ++){
            try {
                Socket server = new Socket(host, port);
                WorkingThread tc = new WorkingThread(
                        server, i, “ClientProtocol”);
                tc.start();
                tc.join();
                server.close();
                Thread.sleep(5000+(int)(Math.random()*1000));
            } catch(InterruptedException e) {e.printStackTrace();}
        }
    }
}
```

Клијент-сервер

```
public class ServerTest1{

    public static void main(String [] args) {
        Server s = new BufferServer(new ServerProtocol());
        s.start();
    }
}
```

Клијент-сервер

```
public class BufferWorkingServer extends Server {
    public static final int MAXTHREADS = 10;
    protected Buffer<Socket> buffer;
    public BufferWorkingServer(int num, String protocol, int port) {
        super(port);
        buffer = new ArrayBuffer<Socket>();
        for (int i = 0; i < num; i++) {
            WorkingThread t = new BufferWorkingThread(i,buffer,protocol);
            t.setDaemon(true);
            t.start();
        }
    }
    public void processRequest(Socket client) {
        buffer.put(client);
    }
}
```

Клијент-сервер

```
public class BufferWorkingThread extends WorkingThread {
    protected Buffer<Socket> buffer;
 public BufferWorkingThread(Buffer<Socket> buffer, String protocol) {
        this.buffer = buffer;
        this.protocol = protocol;
    }
    public void run() {
        while (true) {
            work();
        }
    }
    public Socket getClient() {
        return buffer.get();
    }
}
```

Питања?

Захарије Радивојевић
Електротехнички Факултет
Универзитет у Београду
zaki@etf.rs