



SVEUČILIŠTE U ZAGREBU



Fakultet
elektrotehnike i
računarstva

Diplomski studij

Informacijska i komunikacijska tehnologija:

Telekomunikacije i
informatika

Računarstvo:

Programsko inženjerstvo i
informacijski sustavi

Računarska znanost

Raspodijeljeni sustavi

2. Procesi i komunikacija: model klijent-poslužitelj

Ak. god. 2020./2021.

Creative Commons



- slobodno smijete:

- **dijeliti** — umnožavati, distribuirati i javnosti priopćavati djelo
- **prerađivati** djelo



- pod sljedećim uvjetima:

- **imenovanje:** morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
- **nekomercijalno:** ovo djelo ne smijete koristiti u komercijalne svrhe.
- **dijeli pod istim uvjetima:** ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.



U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.

Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.

Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.

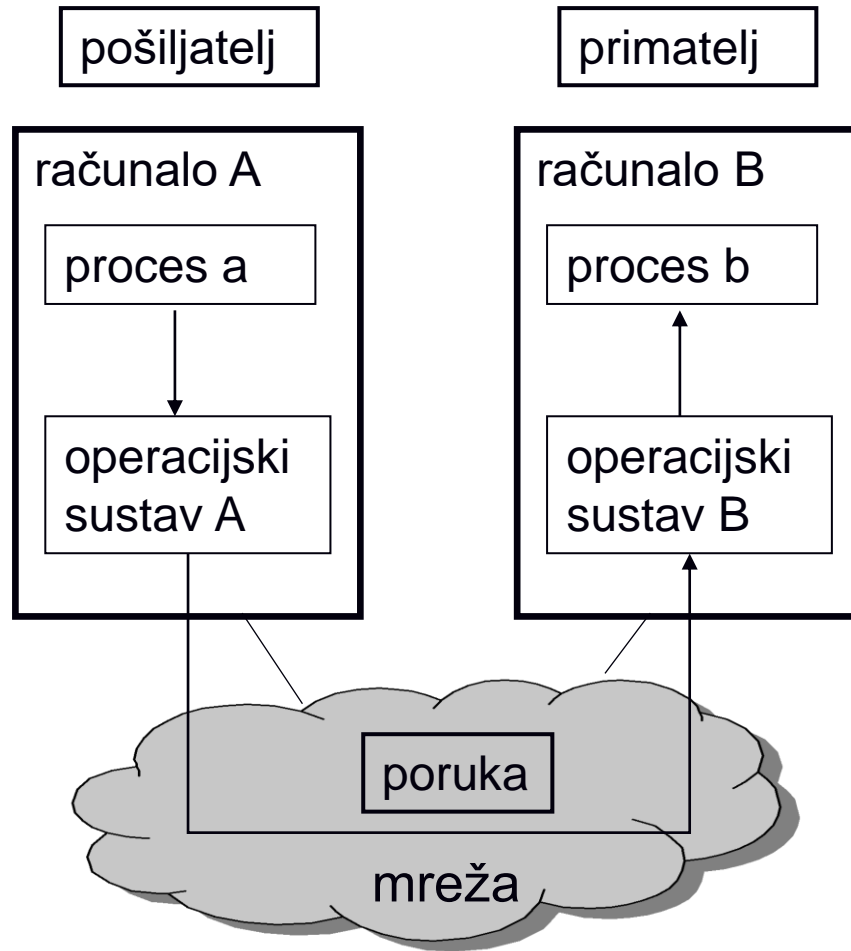
Tekst licence preuzet je s

<http://creativecommons.org/>

Sadržaj predavanja

- Osnovni model komunikacije u raspodijeljenom okruženju
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

Osnovni model komunikacije



- **Procesi**

- izvode se na različitim računalima, autonomni su

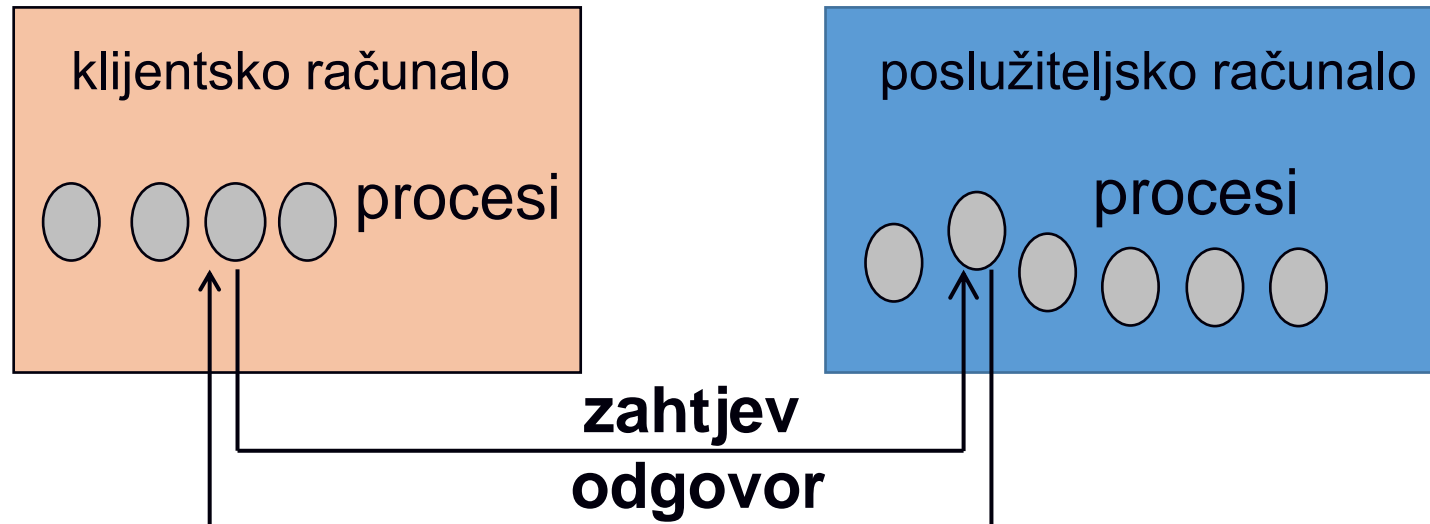
- **Komunikacija**

- prosljeđivanje poruka (engl. *message passing*), tj. razmjena poruka na mrežnom sloju

- **Međuprocesna komunikacija**

- engl. *interprocess communication* (IPC)
- potrebno je osigurati vremensku usklađenost procesa

Prisjetimo se modela klijent-poslužitelj



- KLIJENT

- zahtjeva uslugu
- šalje zahtjev poslužitelju i čeka odgovor

- POSLUŽITELJ

- nudi usluge
- prima i obrađuje dolazne zahtjeve te šalje odgovor klijentima

Obilježja komunikacije

- **konekcijska**

- procesi eksplicitno kreiraju konekciju prije razmjene podataka, postoje kontrolne poruke za uspostavu konekcije

- **bezkonekcijska**

- sve poruke prenose podatke, nema kontrolnih poruka za uspostavu konekcije među procesima

- **perzistentna komunikacija**

- garantira isporuku poruke, poruka se pohranjuje u sustavu i isporučuje primatelju kada je to moguće

- **tranzijentna komunikacija**

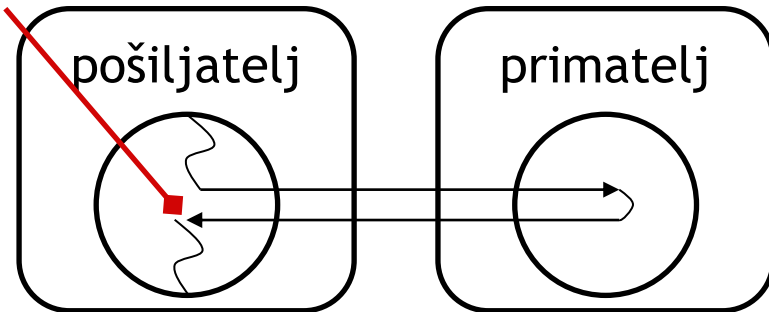
- nepouzdana, garantira isporuku poruke samo ako su pošiljalac i primatelj poruke istovremeno dostupni

Obilježja komunikacije

- **sinkrona komunikacija**

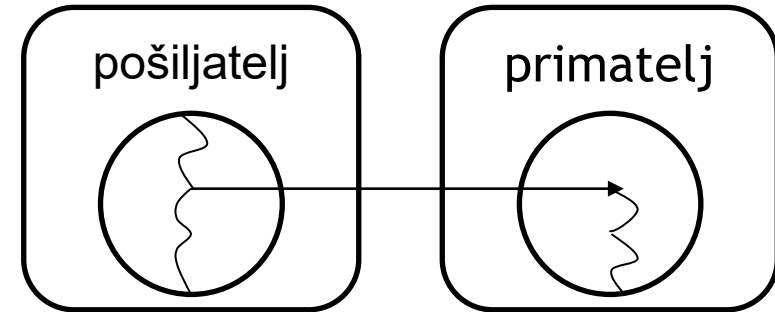
- blokira pošiljatelja do primitka potvrde od strane primatelja

blokiranje

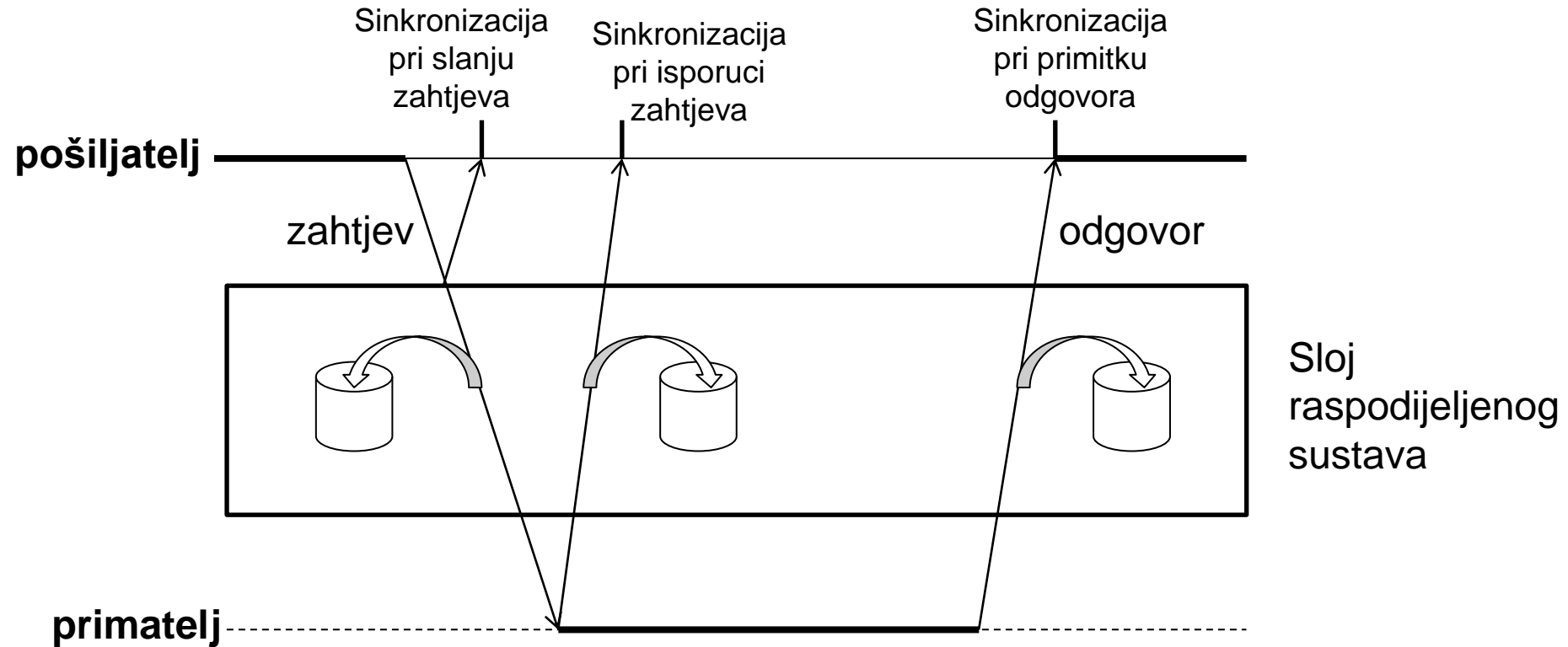


- **asinkrona komunikacija**

- omogućuje pošiljatelju nastavak obrade odmah nakon slanja poruke

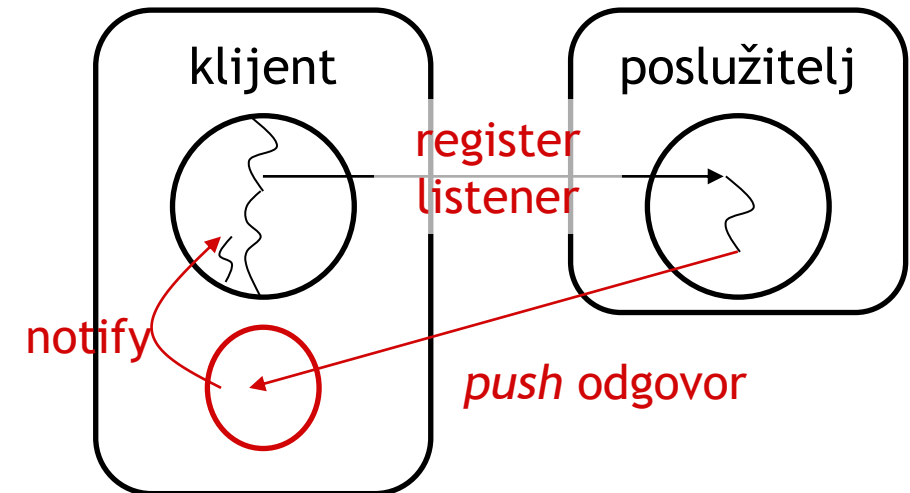
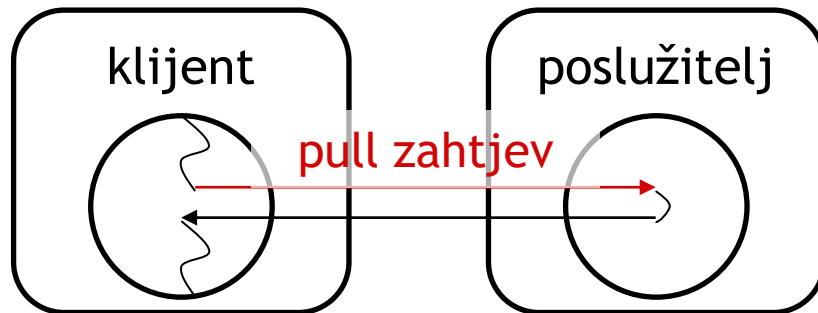


Točke sinkronizacije



Obilježja komunikacije

- **komunikacija na načelu *pull* ili *push***
 - *pull* – “klasični” model zahtjev-odgovor
 - *push* – klijent registrira zahtjev i “sluša” odgovor, poslužitelj šalje odgovor nakon što završi obradu zahtjeva



Procesi

- Definira se kao program u izvođenju (prisjetimo se operacijskih sustava)
- Višedretvenost je važna za učinkovitu implementaciju raspodijelih procesa
 - omogućuje održavanje više logičkih konekcija s jednim procesom
 - višedretveni poslužitelj može paralelno obrađivati korisničke zahtjeve
 - višedretveni klijent može nastaviti s obradom dok čeka odgovor poslužitelja (primjer: Web preglednik)

Obilježja procesa

- **vremenska (ne)ovisnost**

- vremenski ovisni procesi moraju biti istovremeno aktivni za realizaciju komunikacije
- vremenski neovisni procesi mogu komunicirati i ako nisu istovremeno aktivni

- **ovisnost o referenci “sugovornika”**

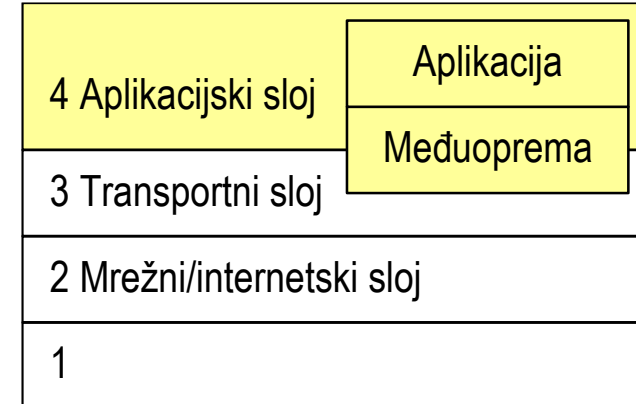
- proces je ovisan o referenci “sugovornika” ako mora znati jedinstveni identifikator (adresu) udaljenog procesa s kojim želi komunicirati
- proces može biti i neovisan o referenci, tj. ne mora znati jedinstveni identifikator udaljenog procesa

Sadržaj predavanja

- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

Sloj raspodijeljenog sustava za komunikaciju među procesima

- vrsta programskog posredničkog sloja (međuopreme)
- implementira komunikacijske protokole za raspodijeljene procese na višem nivou apstrakcije od transportnog sloja
- omogućuje jednostavniji razvoj raspodijeljenih aplikacija, sakriva kompleksnost i heterogenost nižih slojeva



Sloj raspodijeljenog sustava za komunikaciju među procesima

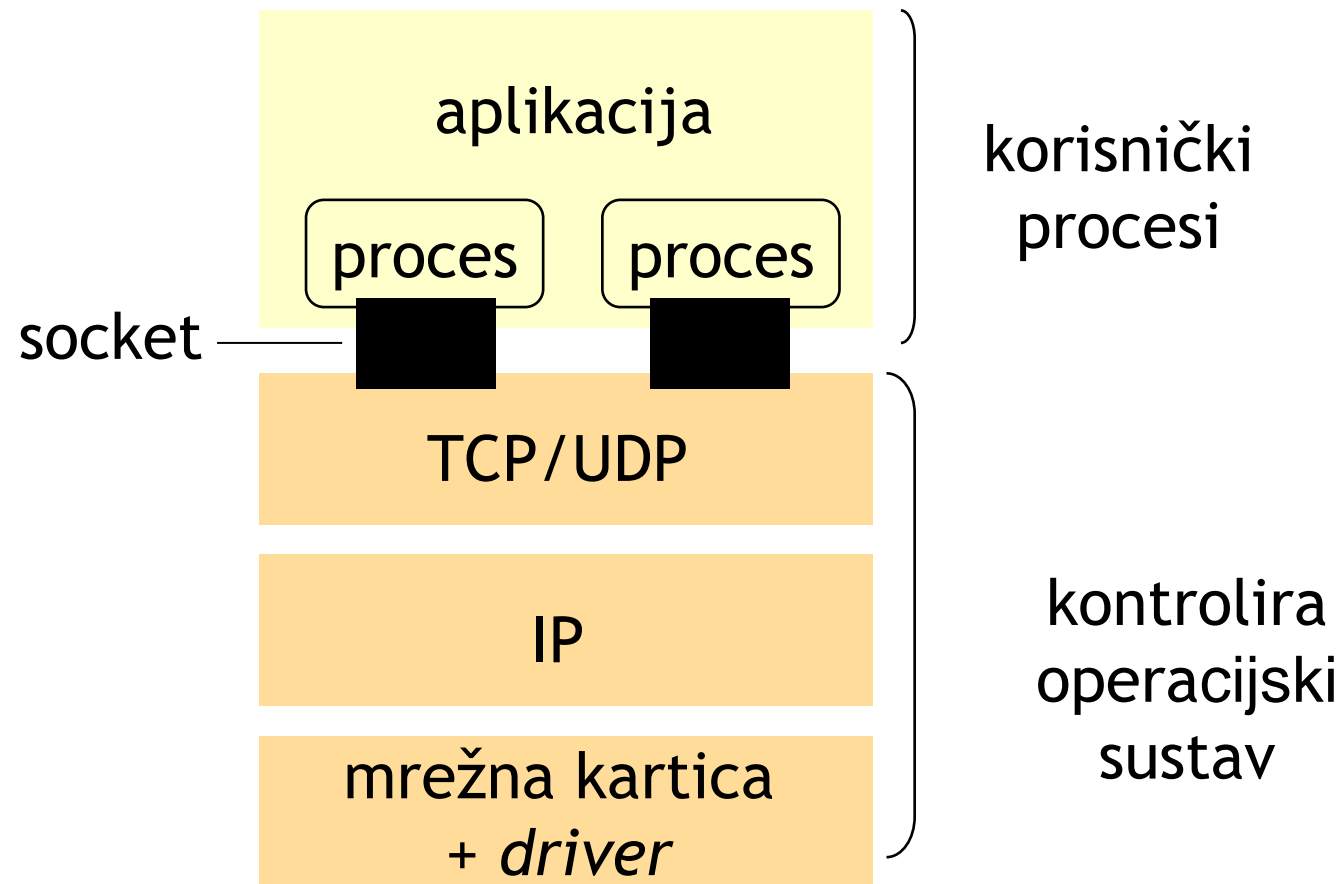
- Postojeća rješenja za komunikaciju raspodijeljenih procesa
 1. komunikacija korištenjem priključnica (*socket API*)
 2. poziv udaljene procedure (*remote procedure call, RPC*)
 3. raspodijeljeni objekti - poziv udaljene metode (*remote method invocation, RMI*)
 4. komunikacija razmjenom poruka (*message-oriented interaction*)
 5. model objavi-pretplati (*publish/subscribe*)
- U nastavku analiziramo prva 3 rješenja koja se temelje na modelu klijent-poslužitelj

Komunikacija korištenjem priključnica

Socket API

- koristi funkcionalnost transportnog sloja
 - TCP – konekcijski protokol, pouzdan prijenos podataka
 - UDP – prijenos nezavisnih paketa (*datagrami*), nepouzdan prijenos
- priključnica (engl. *socket*)
 - pristupna točka preko koje aplikacija šalje podatke u mrežu i iz koje čita primljene podatke
 - viši nivo apstrakcije nad komunikacijskom točkom koju operacijski sustav koristi za pristup transportnom sloju
 - veže se uz vrata (engl. *port*) koja jednoznačno određuju aplikaciju kojoj su poruke namijenjene

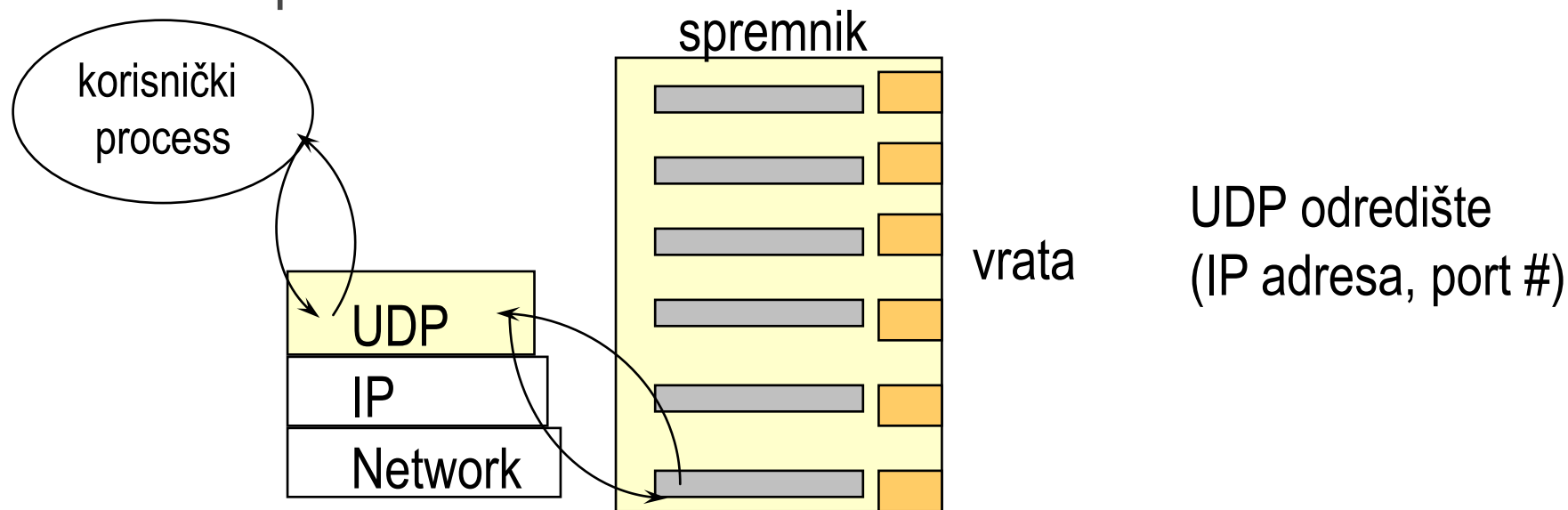
Komunikacija pomoću *socket*a



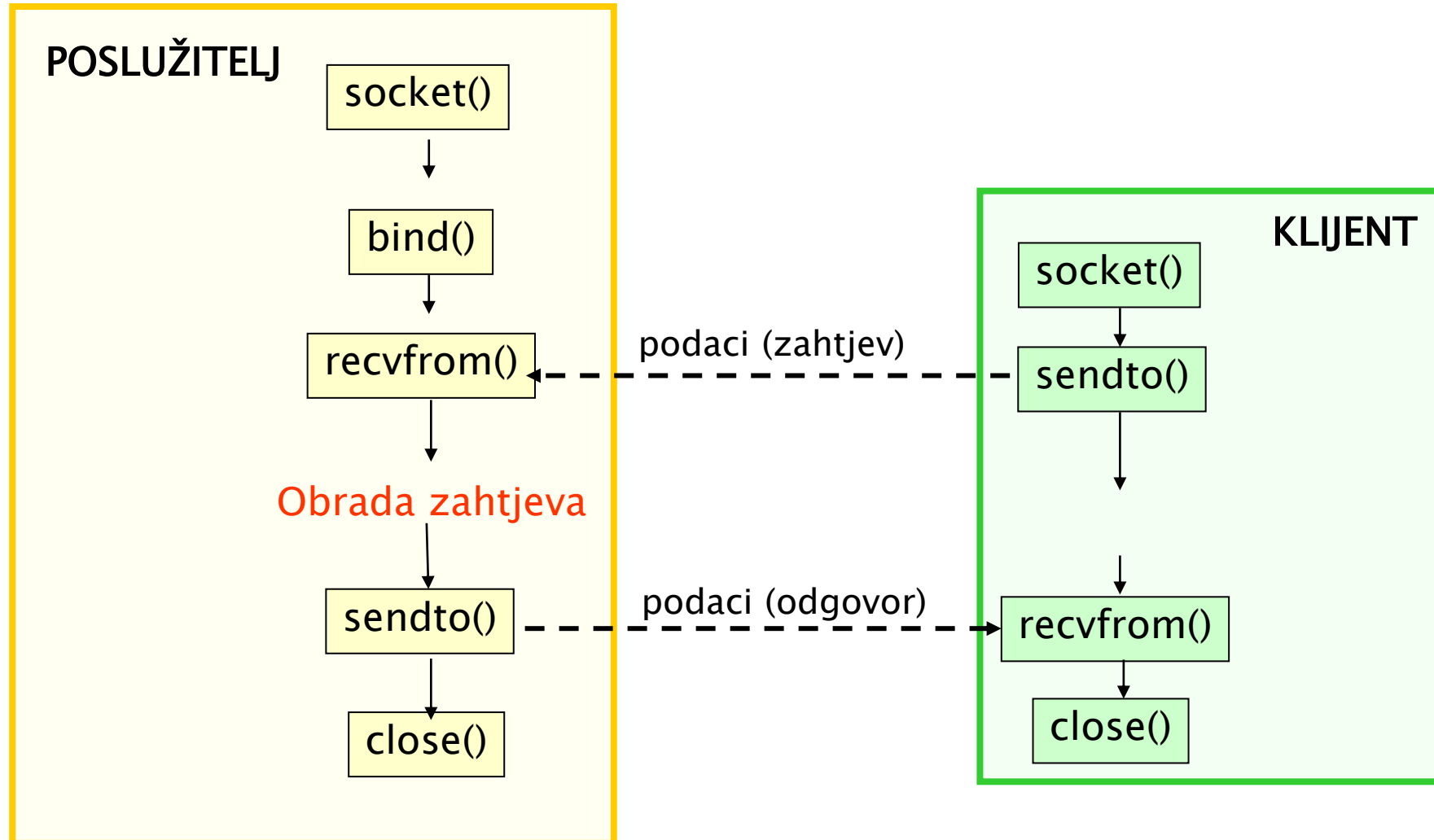
Transportni protokol UDP

User Datagram Protocol (UDP)

- komunikacija se odvija preko vrata (engl. *portova*) koje dodjeljuje operacijski sustav na strani klijenta, na strani poslužitelja se koriste “dobro poznata vrata”



Komunikacija pomoću *socketa UDP*



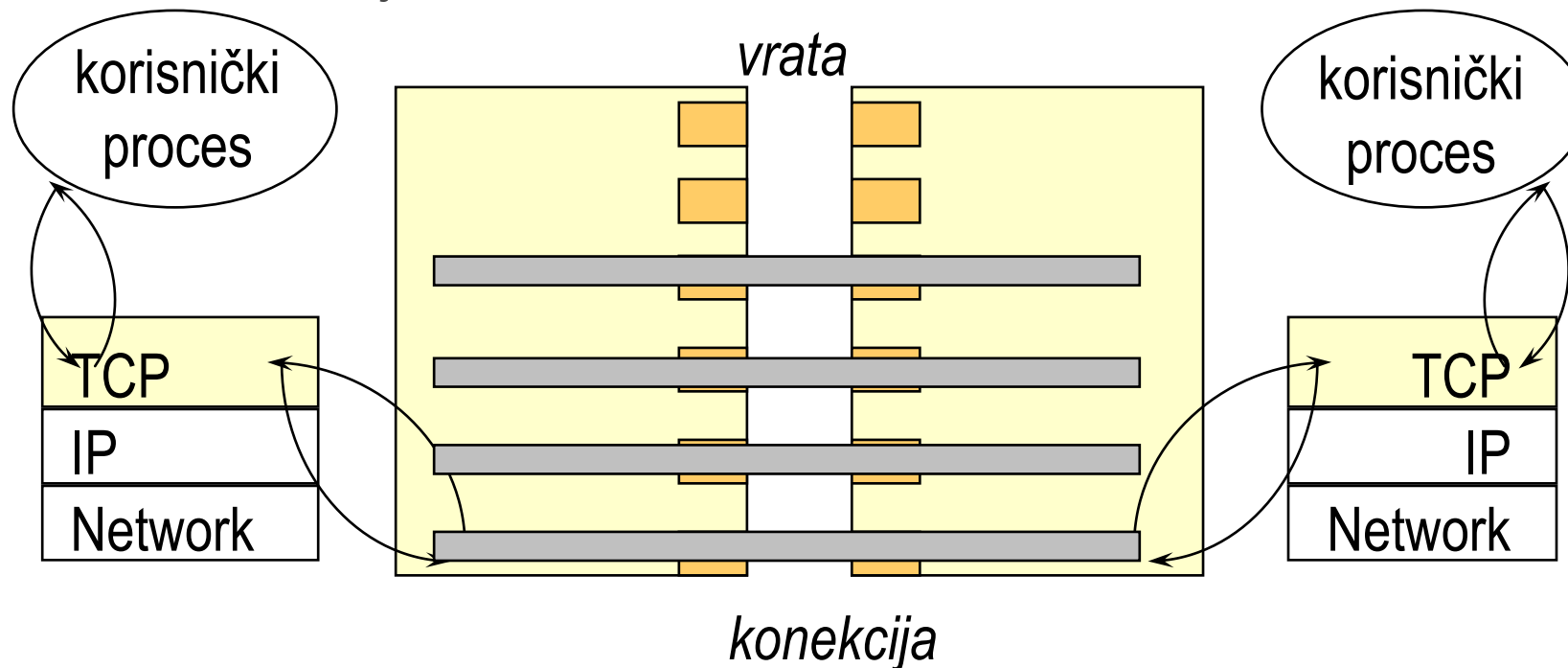
Obilježja *socket* UDP

- model klijent-poslužitelj
- vremenska ovisnost procesa
 - poslužitelj mora biti aktivan za primanje datagrama
- klijent mora znati identifikator poslužitelja
- tranzijentna komunikacija
- asinkrona komunikacija
 - klijent šalje datagram i nastavlja obradu, nema blokiranja pošiljatelja
- nepouzdana komunikacija
- može se koristiti za implementaciju komunikacije na načelu *pull* i *push*

Transportni protocol TCP

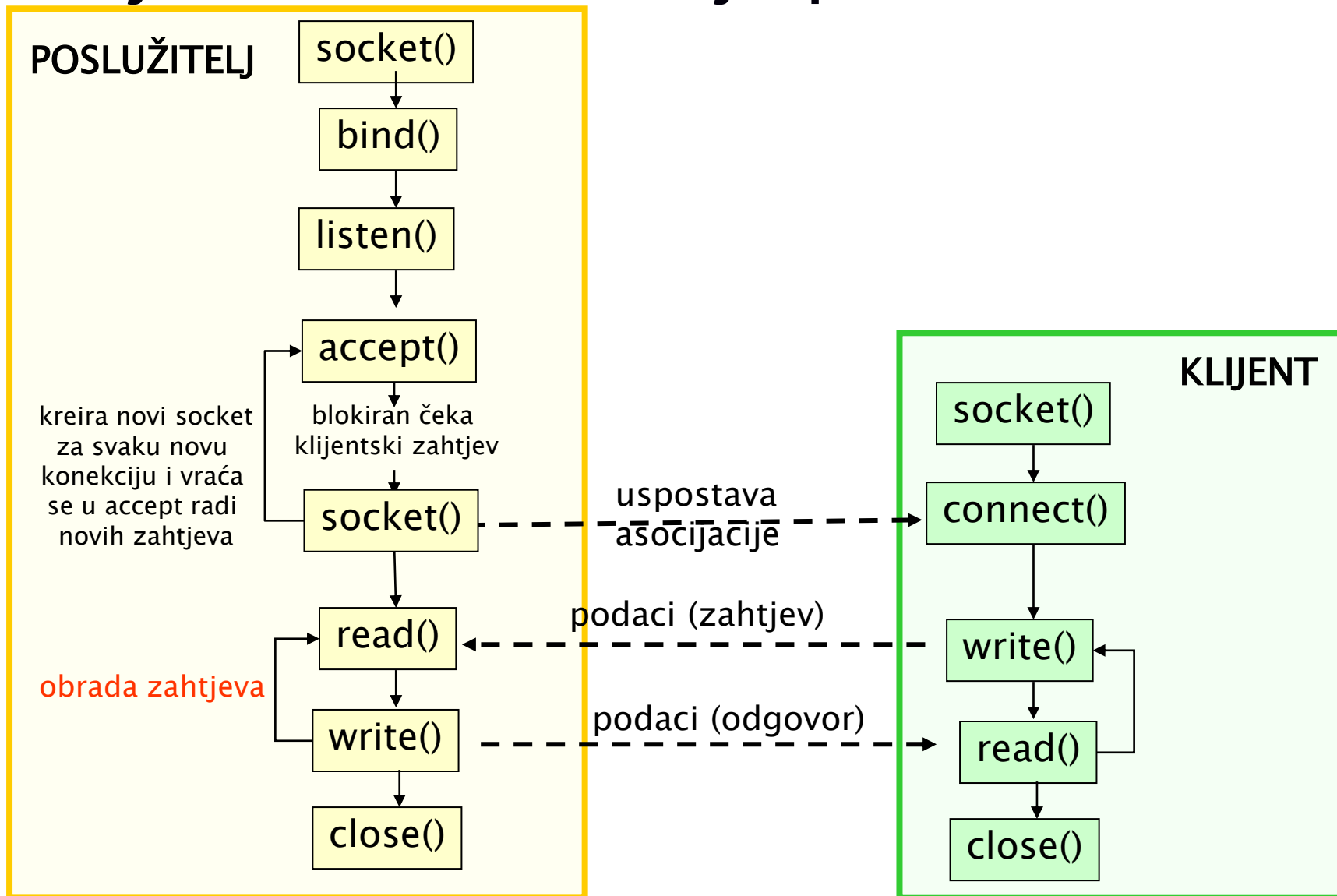
Transmission Control Protocol (TCP)

- konekcija između dvije krajnje točke koje se moraju dogovoriti o uspostavi konekcije

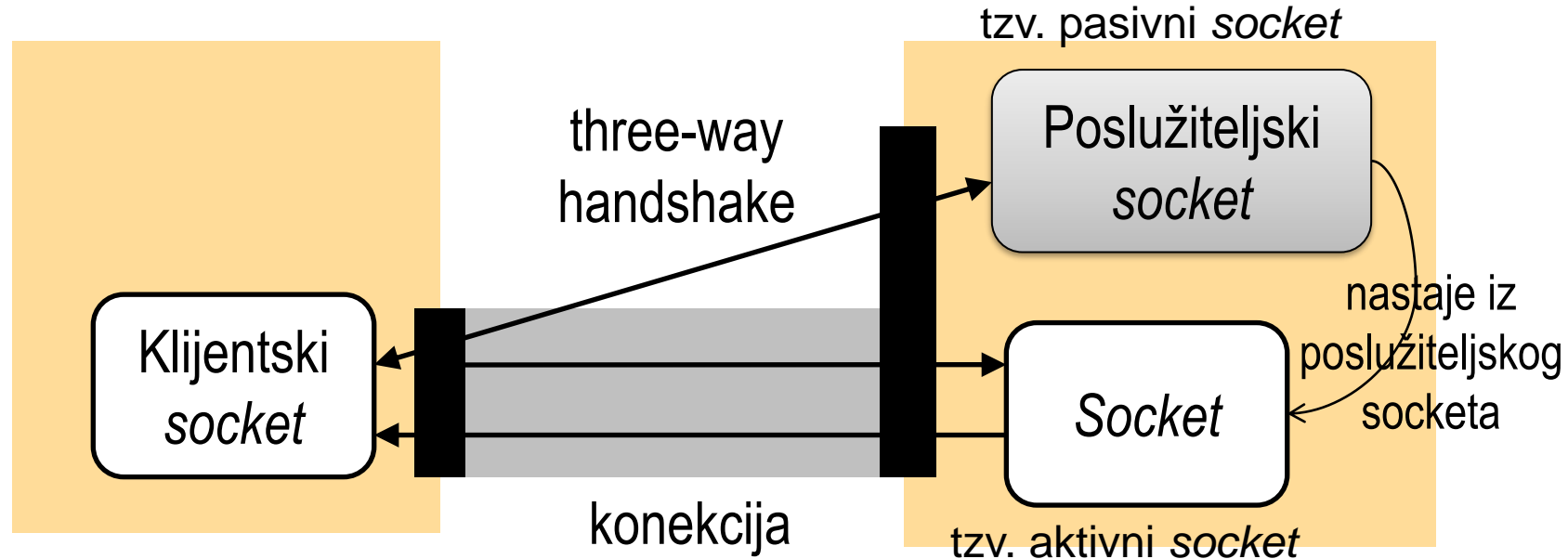


Konekcijska komunikacija pomoću *socket*a

TCP



Konkurentni korisnički zahtjevi



- ♦ za svaki novi korisnički zahtjev kreira se **novi socket** (s dva *buffer*-a, *in* i *out*) **koji se veže uz konekciju** <poslužiteljska IP adresa i broj vrata (broj vrata ostaje isti kao za poslužiteljski socket), klijentska IP adresa i broj vrata>
- ♦ originalni poslužiteljski socket mora konstantno biti u stanju “osluškivanja”

Obilježja *socket* TCP

- model klijent-poslužitelj
- vremenska ovisnost
 - klijent i poslužitelj moraju biti istovremeno dostupni
- klijent mora znati identifikator poslužitelja
- tranzijentna komunikacija
- sinkrona komunikacija
 - klijent šalje zahtjev za kreiranje konekcije i proces je blokiran do uspostave konekcije
- pokretanje komunikacije na načelu *pull*

Sadržaj predavanja

- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

UDP: implementacija poslužitelja

1. Kreirati socket poslužitelja:

```
DatagramSocket serverSocket;  
serverSocket = new DatagramSocket( PORT );
```

2. Kreirati paket (prazan, priprema za primanje):

```
byte[] rcvBuf = new byte[256];  
DatagramPacket packet =  
    new DatagramPacket(rcvBuf, rcvBuf.length);
```

3. Čekati korisnički paket (blokira proces do klijentskog zahtjeva!):

```
serverSocket.receive( packet );
```

4. Obrada pristiglog paketa i po potrebi odgovor klijentu

5. Zatvoriti socket (gasi poslužitelja!):

```
serverSocket.close();
```

UDP: implementacija klijenta

1. Kreirati socket:

```
DatagramSocket clientSocket;  
clientSocket = new DatagramSocket();
```

2. Kreirati paket i napuniti ga podacima:

```
byte[] sendBuf = new byte[256];  
DatagramPacket packet =  
    new DatagramPacket(sendBuf, sendBuf.length, destAddress,  
destPort);
```

3. Slanje paketa:

```
clientSocket.send( packet );
```

4. Po potrebi obrada i čekanje odgovora

5. Zatvoriti socket:

```
clientSocket.close();
```

TCP: implementacija poslužitelja

1. Kreirati socket poslužitelja:

```
ServerSocket serverSocket;  
serverSocket = new ServerSocket( PORT );
```

2. Čekati korisnički zahtjev (blokira proces do klijentskog zahtjeva!!!) i kreirati kopiju originalnog socketa:

```
Socket copySocket = serverSocket.accept();
```

3. Kreirati I/O stream za komunikaciju s klijentom

```
DataInputStream is = new DataInputStream( copySocket.getInputStream() );  
DataOutputStream os = new DataOutputStream( copySocket.getOutputStream() );
```

4. Komunikacija s klijentom

5. Zatvoriti kopiju socketa:

```
copySocket.close();
```

6. Zatvoriti poslužiteljski socket:

```
serverSocket.close();
```

TCP: implementacija klijenta

1. Kreirati klijentski socket:

```
clientSocket = new Socket( address, port );
```

2. Kreirati I/O stream za komunikaciju s poslužiteljem:

```
is = new DataInputStream( clientSocket.getInputStream() );  
os = new DataOutputStream( clientSocket.getOutputStream() );
```

3. Komunikacija s poslužiteljem:

- //Receive data from server:

```
String line = is.readLine();
```
- //Send data to server:

```
os.writeBytes("Hello\n");
```

4. Zatvoriti socket:

```
clientSocket.close();
```

Paket `java.net`

- **API specification**

<https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html>

- Osnovne klase

- Socket, ServerSocket, URL, URLConnection, (koriste TCP)
- DatagramPacket, DatagramSocket, MulticastSocket (koriste UDP)

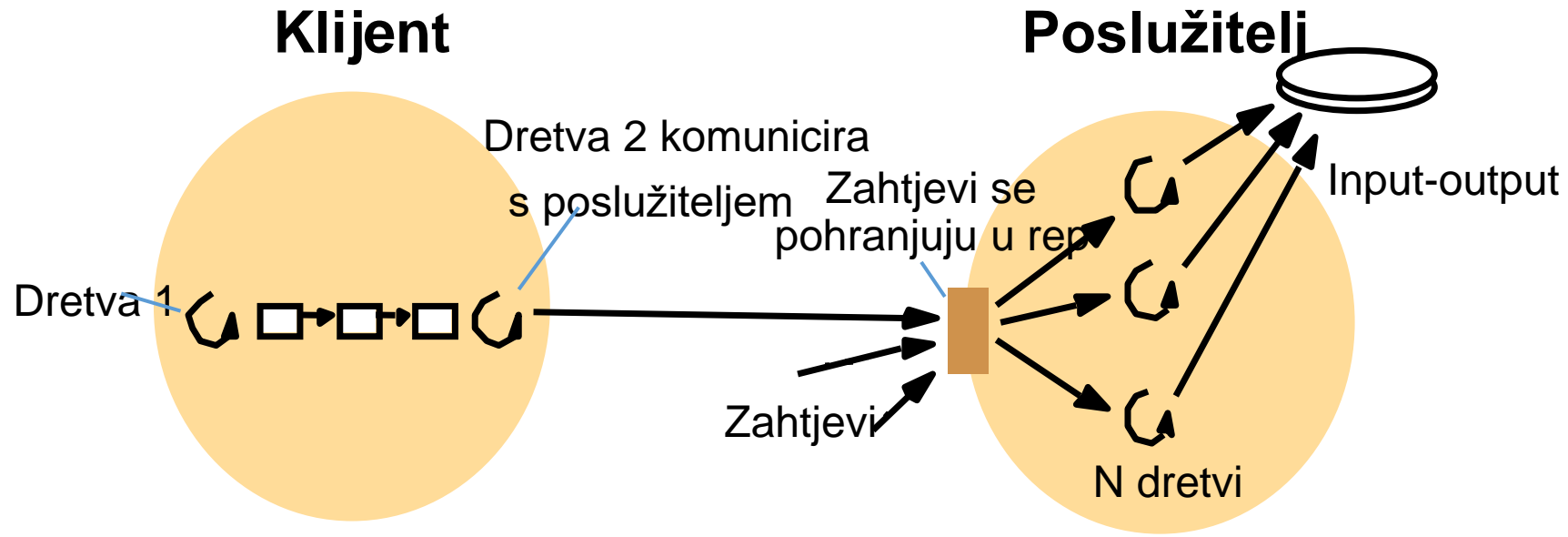
- Java Networking Tutorial

<http://docs.oracle.com/javase/tutorial/networking/>

Sadržaj predavanja

- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

Višedretveni poslužitelj i klijenti



Uobičajene zadaće na strani klijenta:

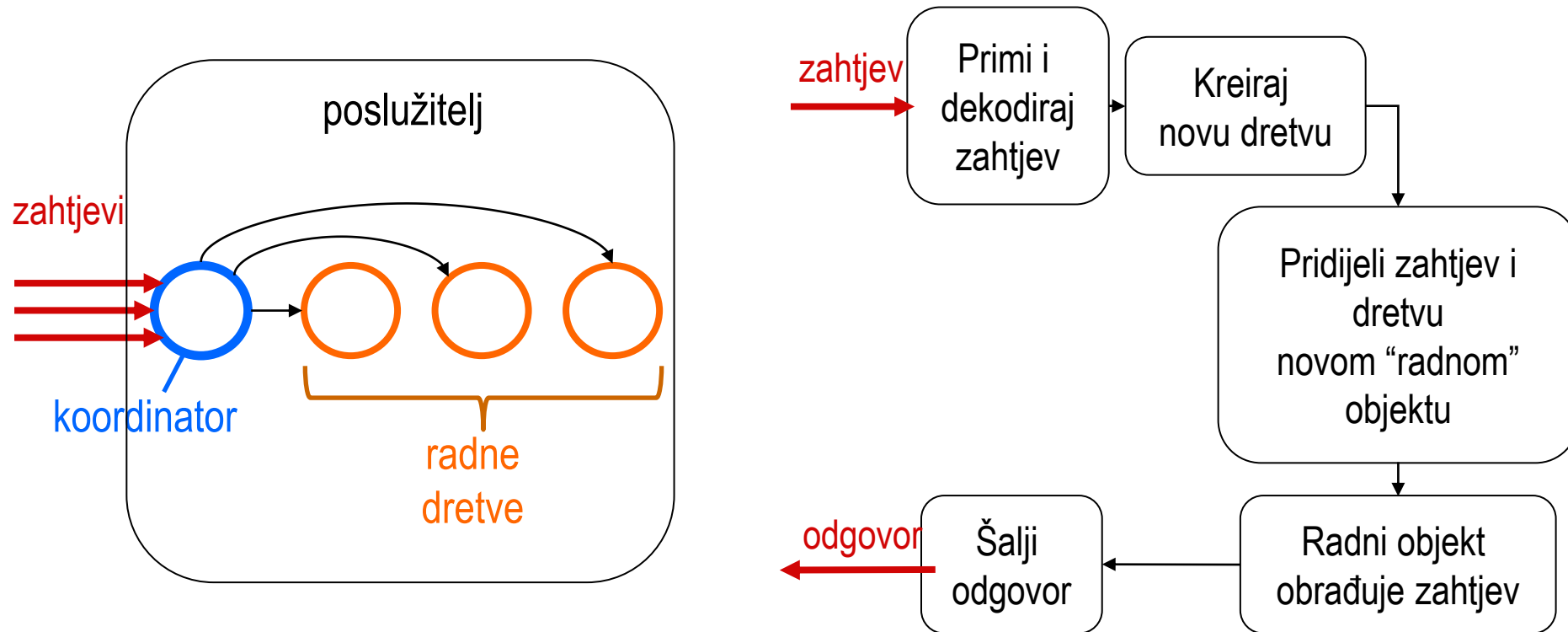
- korisničko sučelje,
- otvaranje mrežne konekcije i primanje podataka

Uobičajene zadaće na strani poslužitelja:

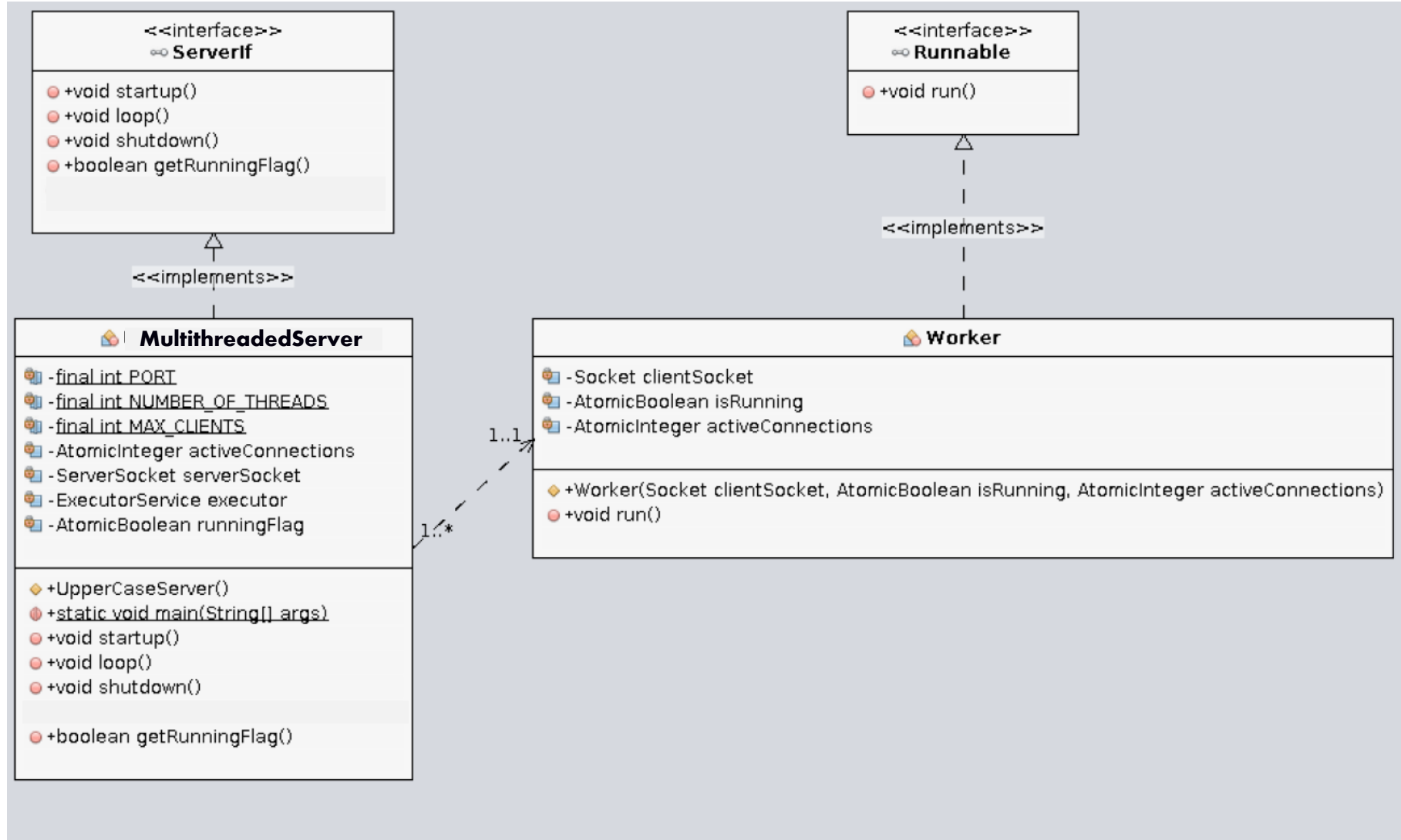
- primanje konkurentnih klijentskih zahtjeva
- složena obrada podataka
- rad s diskom/bazom podataka

Višedretveni poslužitelj

Model koordinator/radna dretva (*dispatcher/worker model*)



Primjer višedretvenog poslužitelja



Sučelje višedretvenog poslužitelja

```
public interface ServerIf {  
    // Server startup. Starts all services offered by the server.  
    public void startup();  
  
    // Server loops when in running mode. The server must be active  
    // to accept client requests.  
    public void loop();  
  
    // Server shutdown. Shuts down all services started during  
    //startup.  
    public void shutdown();  
  
    // Gets the running flag that indicates server running status.  
    // @return running flag  
    public boolean getRunningFlag();  
}
```

Poslužitelj (1)

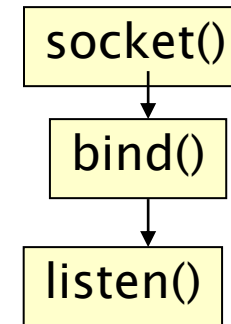
```
public class MultithreadedServer implements ServerIf {  
  
    private static final int PORT = 10002; // server port  
    private static final int NUMBER_OF_THREADS = 4;  
    //Max queue length for incoming connection requests.  
    private static final int BACKLOG = 10;  
  
    private final AtomicInteger activeConnections;  
    private ServerSocket serverSocket;  
    private final ExecutorService executor;  
    private final AtomicBoolean runningFlag;  
  
    ...  
}
```

Poslužitelj (2)

```
...  
    public MultithreadedServer () {  
        activeConnections = new AtomicInteger(0);  
        executor =  
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);  
        runningFlag = new AtomicBoolean(false);  
    }  
    public static void main(String[] args) {  
        ServerIF server = new MultithreadedServer ();  
        //start all required services  
        server.startup();  
        // run the main loop to accept client requests  
        server.loop()  
        //initiate shutdown when such request is received  
        server.shutdown();  
    }  
    ...
```

Poslužitelj (3)

```
//Starts all required server services.  
@Override  
public void startup() {  
    // create a server socket, bind it to the specified port  
    // on the local host and set the backlog for  
    // client requests  
    try {  
        this.serverSocket = new ServerSocket(PORT, BACKLOG);  
        // set timeout to avoid blocking  
        serverSocket.setSoTimeout(500);  
        runningFlag.set(true);  
        System.out.println("Server is ready!");  
    } catch (SocketException e1) {  
        System.err.println("Exception caught when setting server socket timeout: " +  
e1);  
    } catch (IOException ex) {  
        System.err.println("Exception caught when opening or setting the server  
socket: " + ex);  
    }  
} ...
```



Poslužitelj (4)

```
// The main loop for accepting client requests.
@Override
public void loop() {
    while(runningFlag.get()) {
        try{// create a new socket, accept and listen for a connection made to this socket
            Socket clientSocket = serverSocket.accept(); accept()
            // execute a tcp request handler in a new thread
            Runnable worker= new Worker(clientSocket, runningFlag,  activeConnections);
            executor.execute(worker);
            activeConnections.getAndIncrement();
        } catch(SocketTimeoutException ste) {
            // do nothing, check runningFlag
        } catch(IOException ex) {
            System.err.println("Exception caught when waiting for a connection: " + ex);
        }
    }
}
```

Poslužitelj (5)

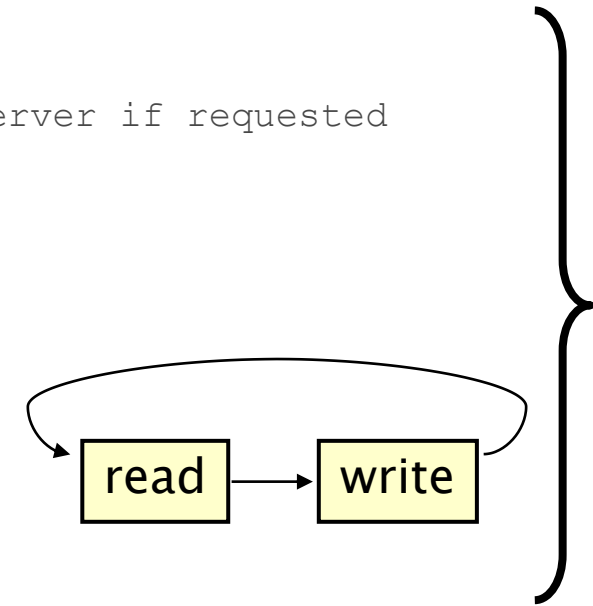
```
@Override
public void shutdown() {
    while( activeConnections.get() > 0 ) {
        System.out.println( "WARNING: There are still active
            connections" );
        try { Thread.sleep( 5000 );
        } catch( java.lang.InterruptedException e ){}
    }
    if( activeConnections.get() == 0 ) {
        System.out.println( "Starting server shutdown." );
        try { serverSocket.close(); close()
        } catch (IOException e) {
            System.err.println("Exception caught when closing the server socket: " + e);
        } finally { executor.shutdown();
        }
        System.out.println("Server has been shutdown.");
    }
}
```

Worker (1)

```
public class Worker implements Runnable {
    private final Socket clientSocket;
    private final AtomicBoolean isRunning;
    private final AtomicInteger activeConnections;
    public Worker(Socket clientSocket, AtomicBoolean isRunning, AtomicInteger activeConnections)
    {
        this.clientSocket = clientSocket;
        this.isRunning = isRunning;
        this.activeConnections = activeConnections;
    }
    @Override
    public void run() {
        try (//create a new BufferedReader from an existing InputStream
            BufferedReader inFromClient = new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));
            //create a PrintWriter from an existing OutputStream
            PrintWriter outToClient = new PrintWriter(new
                OutputStreamWriter(clientSocket.getOutputStream()), true);)
        {
        }
    }
}
```


Worker (2)

```
String receivedString;  
// read a few lines of text  
while ((receivedString=inFromClient.readLine()) != null {  
    System.out.println("Server received:"+receivedString);  
    if (receivedString.contains("shutdown")) {//shutdown the server if requested  
        outToClient.println("Initiating server shutdown!");  
        isRunning.set(false);  
        activeConnections.getAndDecrement();  
        return;  
    }  
    String stringToSend = receivedString.toUpperCase();  
    // send a String then terminate the line and flush  
    outToClient.println(stringToSend);  
    System.out.println("Server sent: " + stringToSend);  
}  
activeConnections.getAndDecrement();  
} catch (IOException ex){  
    System.err.println("Exception caught when trying to read or send data: " + ex);  
}
```

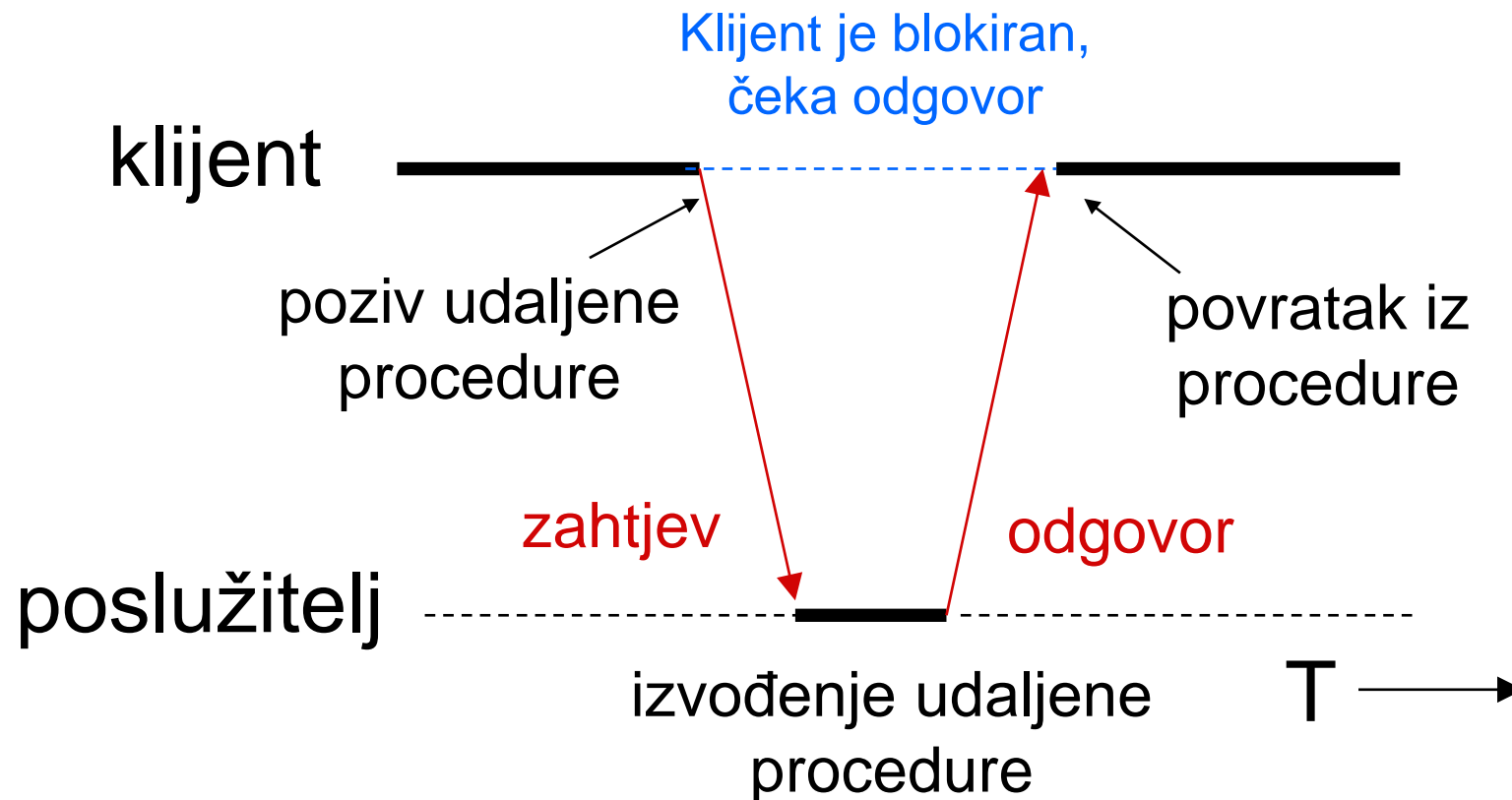


Sadržaj predavanja

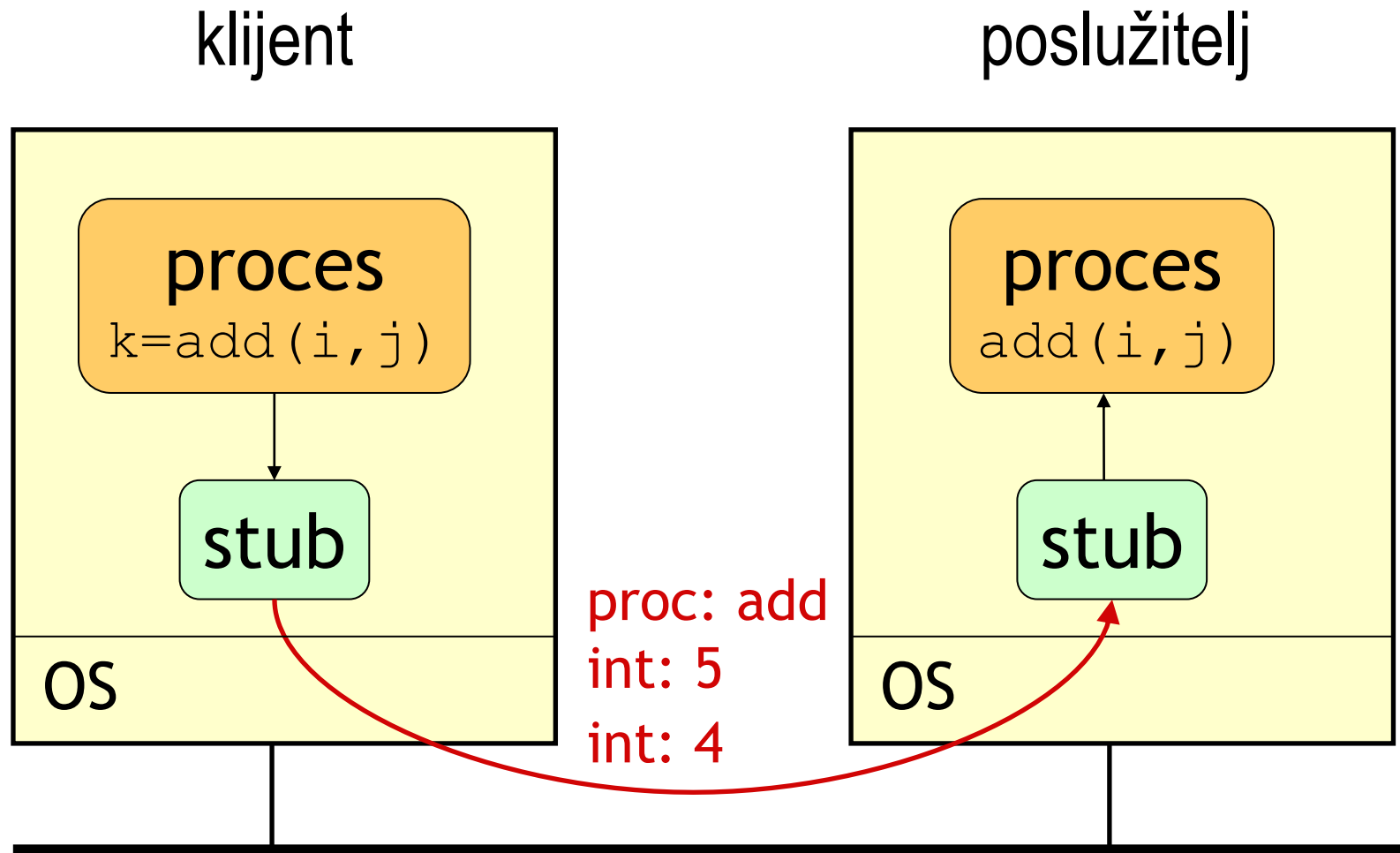
- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

Poziv udaljene procedure (RPC)

- Omogućuje procesima pozivanje i izvođenje procedura na udaljenom računalu.



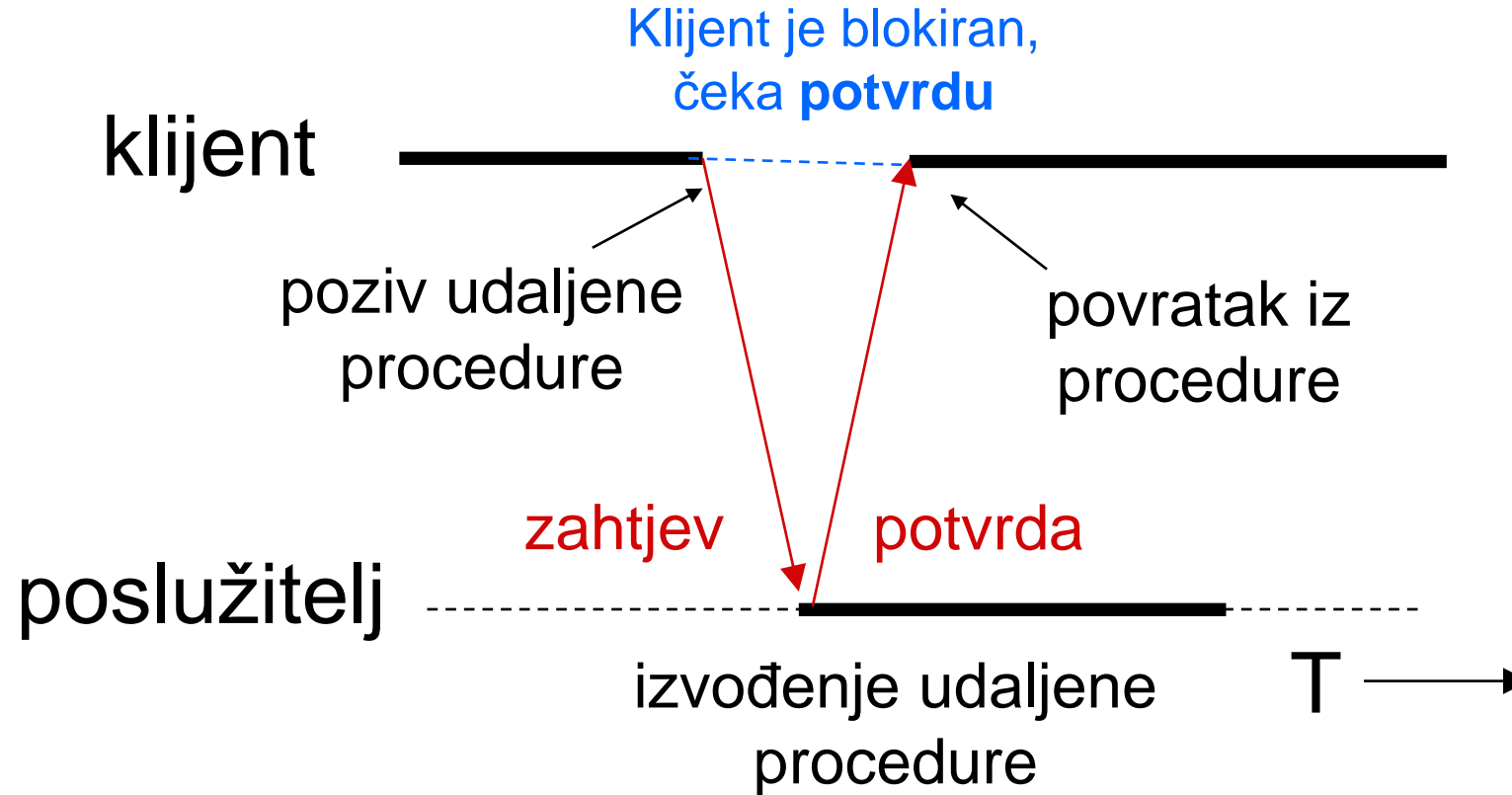
Izvođenje RPC-a



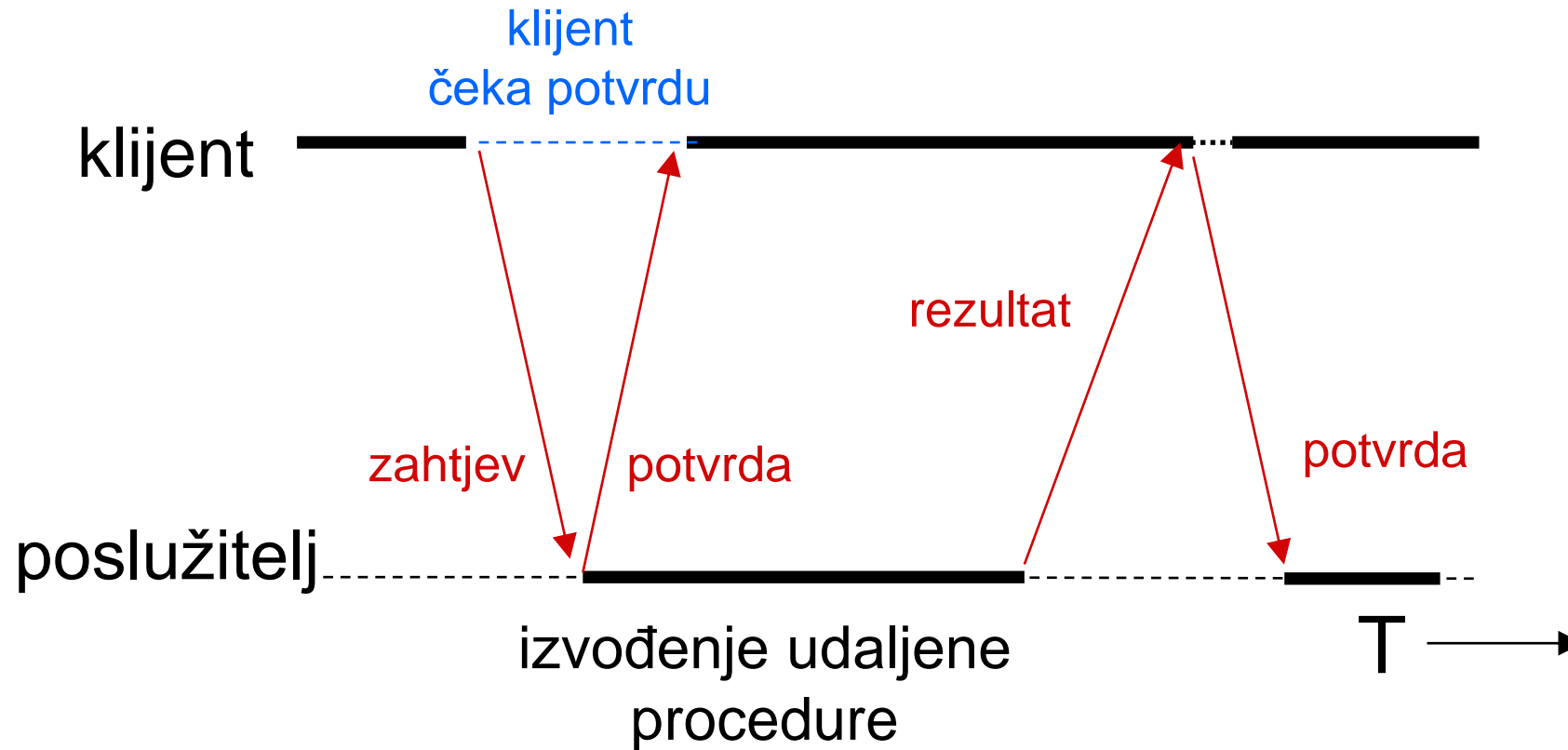
Prenošenje parametara

- *Marshaling* – “pakiranje” parametara ili rezultata u poruku
- *Unmarshaling* – čitanje parametara ili rezultata iz poruke
- Prenošenje vrijednosti parametra
 - Navodi se tip (npr. int, char, long) i vrijednost
 - Različiti OS-ovi često koriste različite prikaze znakova
- Prenošenje parametara koristeći reference
 - Referenca ima smisla samo u adresnom prostoru procesa koji je koristi!
 - Kako prenijeti string na udaljeno računalo?
 - nije moguće koristiti referencu na string!
 - kopiranje cijelog stringa i “pakiranje” u poruku

Asinkroni RPC



Odgođeni sinkroni RPC



Sadržaj predavanja

- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure (*Remote Procedure Call* - RPC)
 - poziv udaljene metode (*Remote Method Invocation* - RMI)

Poziv udaljene metode

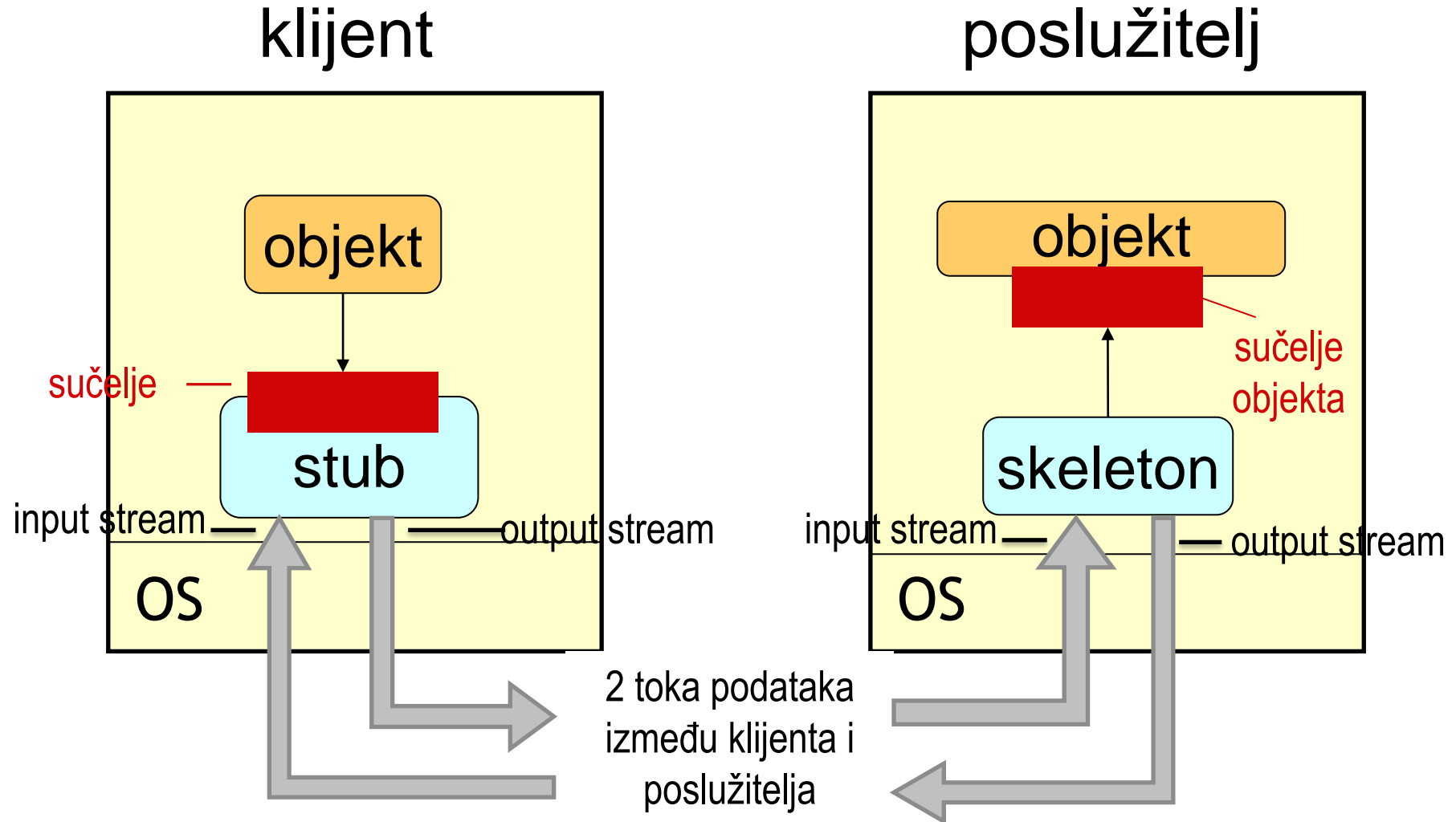
Remote Method Invocation (RMI)

- “nasljednik” poziva udaljene procedure, poziva se metoda udaljenog objekta
- udaljeni objekt
 - proširenje osnovnog objektnog modela za raspodijeljenu okolinu
 - odvajanje sučelja i implementacije objekta
- objekt (klijent) poziva metodu udaljenog objekta (poslužitelja) na transparentan način
 - identično pozivu metode lokalnog objekta

Udaljeni objekti

- Postoje reference na lokalne i udaljene objekte
- Svaki udaljeni objekt ima globalno jedinstven identifikator
 - npr. [ref: [endpoint:[161.53.19.24:1251](local),objID:[0]]]
- Potrebna je usluga za registriranje i pronalaženje udaljenih objekata (*directory service*)

Izvođenje RMI-a



Obilježja RPC/RMI

- model klijent-poslužitelj
- vremenska ovisnost klijenta i poslužitelja
- klijent mora znati identifikator poslužitelja
- tranzijentna komunikacija
- sinkrona komunikacija
 - klijent je blokiran dok ne primi odgovor od strane poslužitelja
- pokretanje komunikacije na načelu *pull*

Java RMI

Java Remote Method Invocation

Sunovo rješenje za komunikaciju udaljenih objekata na načelu poziva udaljene procedure/metode

Oblikovan isključivo za programski jezik Java: omogućuje jednostavniju komunikaciju objekata koji se izvode u različitim JVM (*Java Virtual Machine*)

Implementacija koristi TCP kao transportni protokol

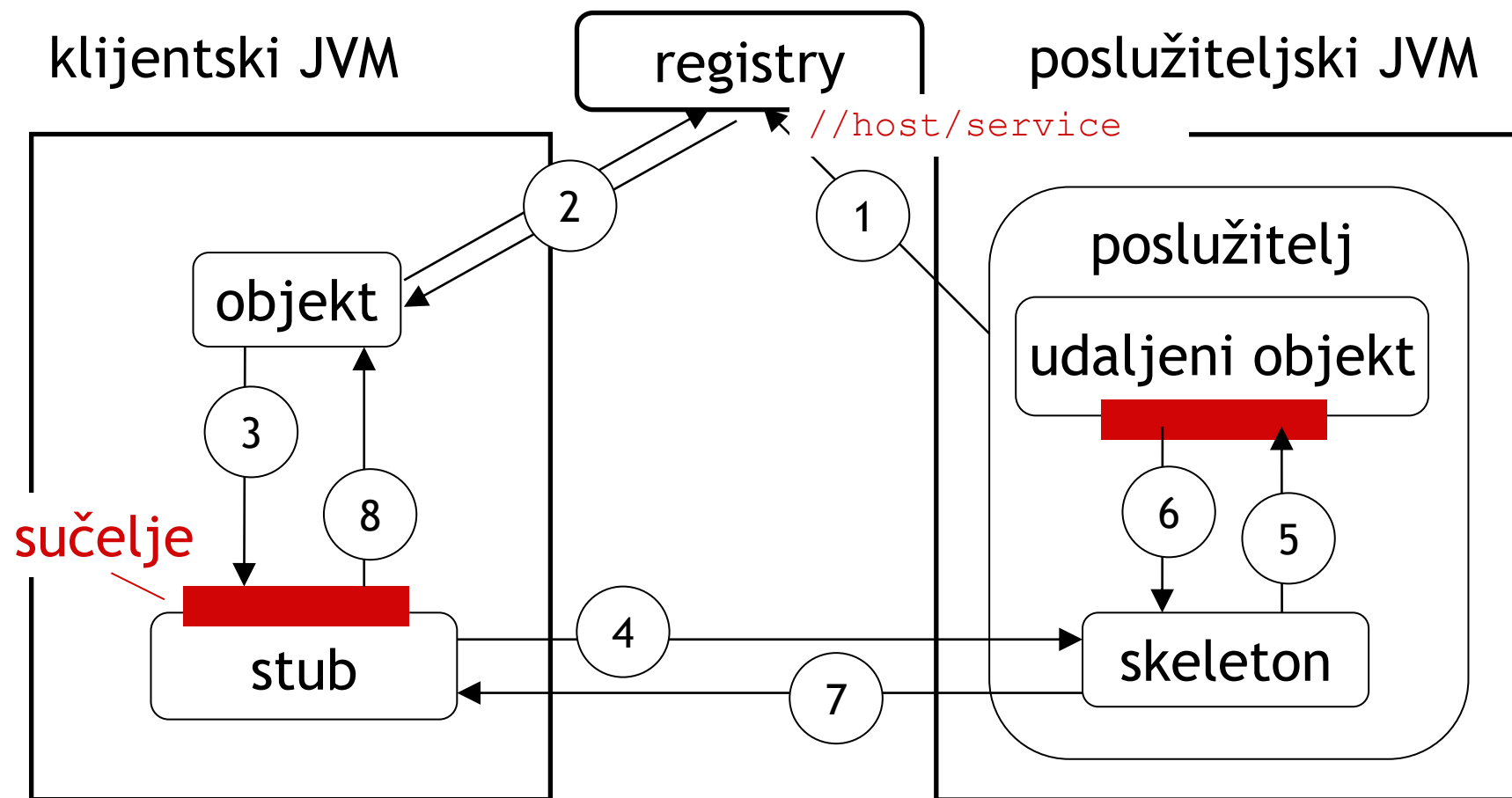
Javin model objekta

- transparentnost pristupa udaljenim objektima
 - referenca na udaljeni objekt istovjetna je referenci na lokalni objekt, no moraju implementirati sučelje `java.rmi.Remote`
- sučelja udaljenog objekta omogućuju komunikaciju s udaljenim objektom
- sučelje udaljenog objekta implementira *stub* (*proxy*) u adresnom prostoru klijentskog računala
- klase *stub* i *skeleton* generiraju se iz implementacije, a ne iz sučelja udaljenog objekta

Prenošenje parametara udaljenoj metodi

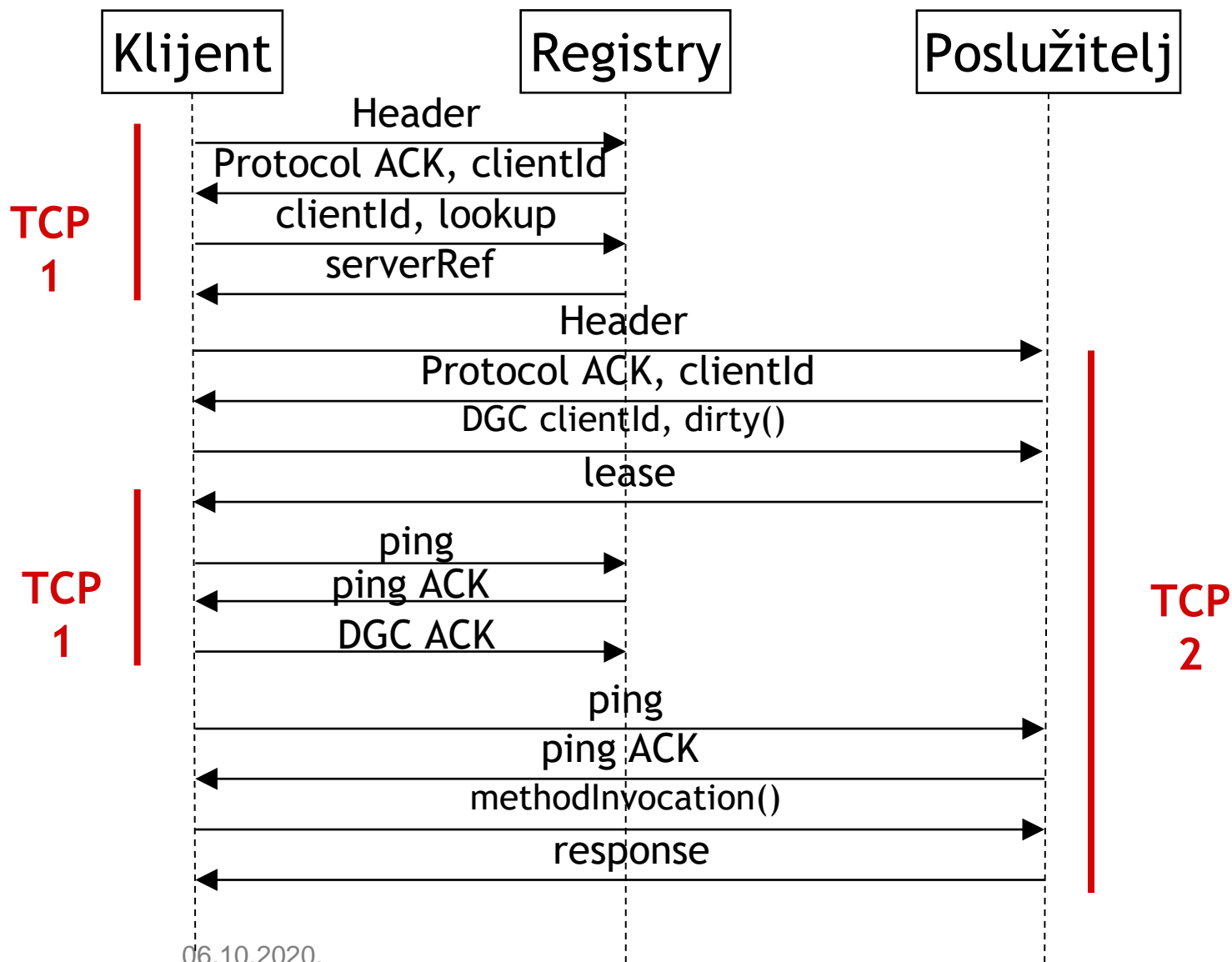
- lokalni objekti moraju se serijalizirati i prenosi se njihova vrijednost (*pass by value*)
 - implementiraju sučelje `Serializable`
- udaljeni se objekti prenose koristeći referencu (*pass by reference*)
 - implementiraju sučelje `java.rmi.Remote` i pravilno su eksportirani `UnicastRemoteObject.exportObject()`
 - referenca = adresa računala + port + identifikator udaljenog objekta
 - referenca udaljenog objekta je jedinstvena u raspodijeljenom sustavu

Protokol Java RMI (1)

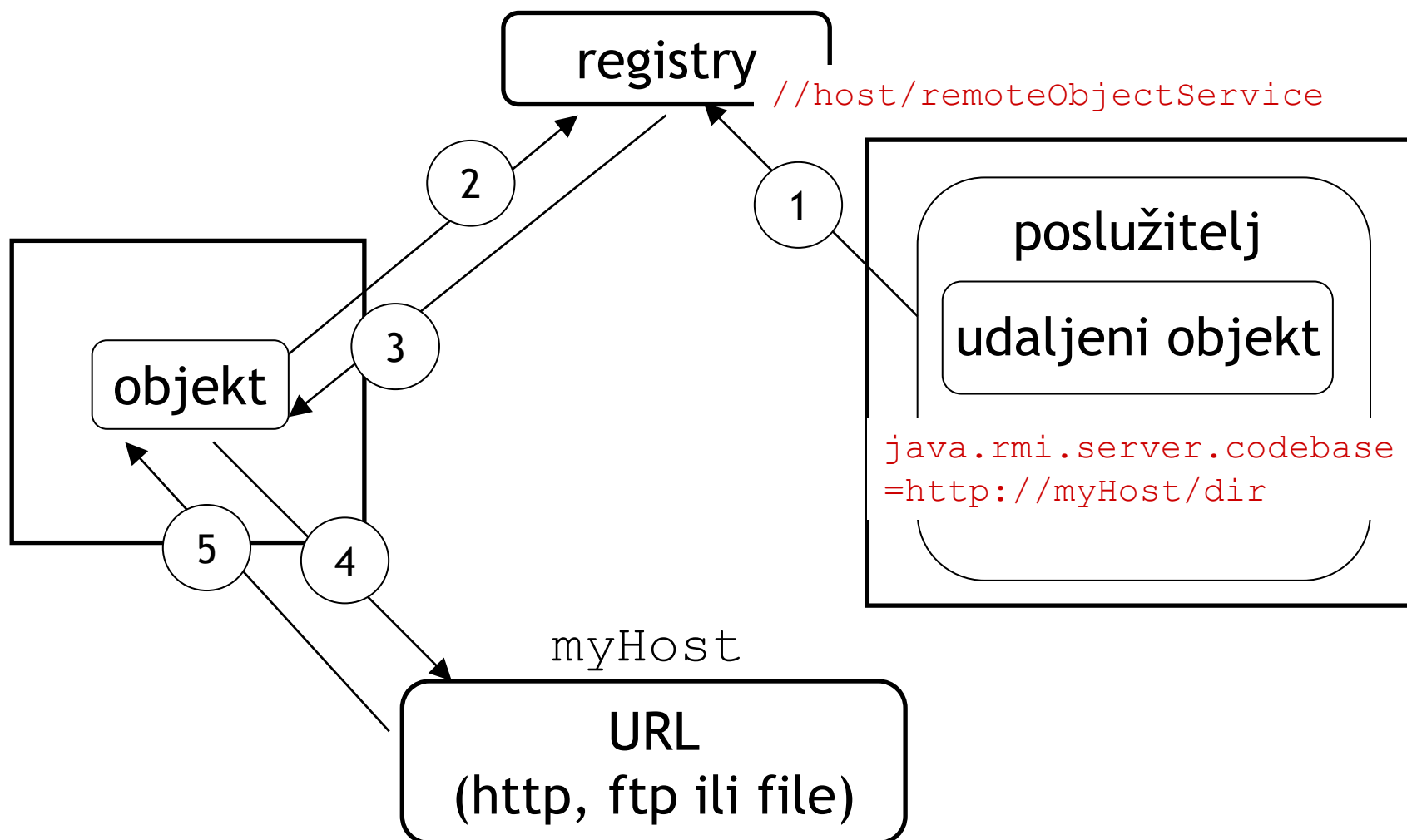


Pretpostavka: stub postoji na strani klijenta

Protokol Java RMI (2)



Dinamično učitavanje klase stuba (1)



Primjer RMI sučelja

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

/**
 * Remote object offers the service of converting a string
 * to upper case.
 */
public interface UpperCase extends Remote {

    public String toUpperCase
        (String originalString) throws RemoteException;

}
```

Primjer RMI poslužitelja (1)

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class UpperCaseImpl extends UnicastRemoteObject
    implements UpperCase {
    private static final String rmiUrl = "rmi://localhost:1099/UpperCase4U";
    public UpperCaseImpl() throws RemoteException {
        super();
    }
    public String toUpperCase( String originalString )
        throws RemoteException {
        return( originalString.toUpperCase() );
    }...
}
```

Primjer RMI poslužitelja (2)

```
...
public static void main(String[] args) {
    try {
        if (System.getSecurityManager() == null)
            System.setSecurityManager(
                new RMISecurityManager());

        UpperCaseImpl serverObject = new UpperCaseImpl();
        Naming.rebind(rmiUrl, serverObject);
        System.out.println("UpperCase object bound to " + rmiUrl);

    } catch (Exception e) {
        e.printStackTrace();
    }
}}
```

Primjer RMI klijenta (1)

```
import java.rmi.RemoteException;
import java.rmi.NotBoundException;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class UpperCaseClient {
    private static final String rmiUrl = "rmi://localhost:1099/UpperCase4U";
    private UpperCase uc = null;
    public UpperCaseClient() {
        try { uc = (UpperCase) Naming.lookup( rmiUrl );
            System.out.println( "Found remote object " + uc.toString() );
        } catch( Exception e ) {
            e.printStackTrace();
        }
    }
    ...
}
```

Primjer RMI klijenta (2)

```
...
public static void main(String[] args) {
    if (System.getSecurityManager() == null)
        System.setSecurityManager(new RMISecurityManager());
    UpperCaseClient client = new UpperCaseClient();
    try {
        String any = new String( "Any string...");
        System.out.println( "Sending\t" + any );
        System.out.println("Received\t"
            + client.uc.toUpperCase(any));
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    System.exit(0);
}}
```

Obilježja Java RMI-a

- pozitivna svojstva
 - visok nivo transparentnosti
 - poziv udaljene metode ima jednaku sintaksu pozivu lokalne metode
 - podržava dinamičko učitavanje klasa
 - jednostavna i brza implementacija raspodijeljenog sustava
 - jednostavniji i čitljiviji kod programa
- negativna svojstva
 - performanse: poziv udaljene metode je puno sporiji od poziva metode lokalnog objekta, čak i ako su udaljeni objekt i klijent na istom računalu (TCP + dizajn protokola s velikim brojem ping paketa)

Paket `java.rmi`

- **API specification**

<http://docs.oracle.com/javase/8/docs/api/java/rmi/package-summary.html>

- **The Java Tutorials, Trail: RMI**

<http://docs.oracle.com/javase/tutorial/rmi/>

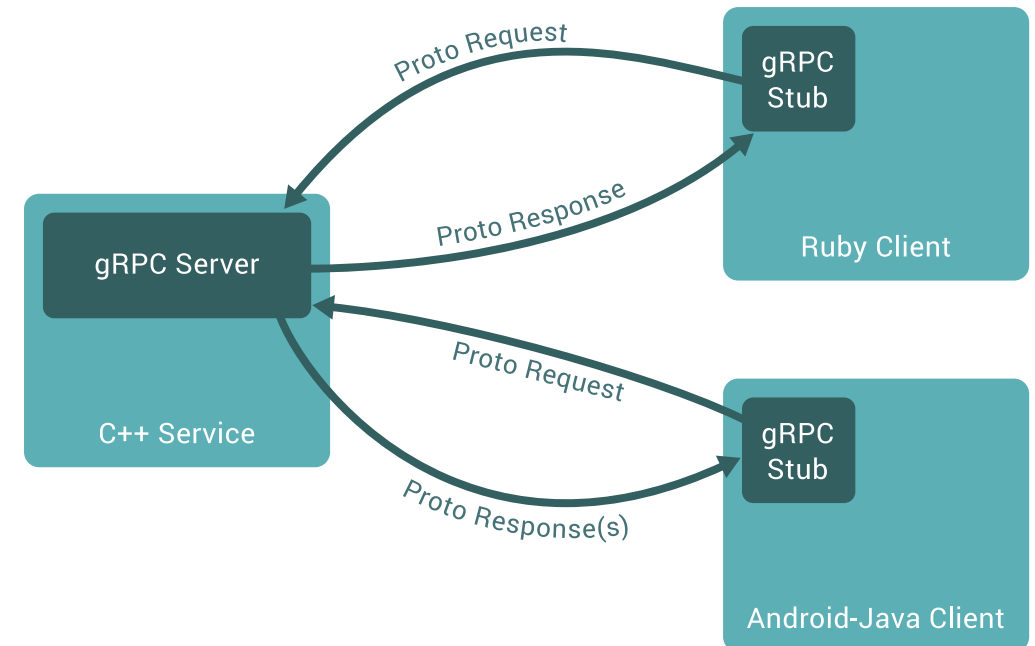
- **Java Remote Method Invocation - Distributed Computing for Java**

<http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html>

gRPC

<https://grpc.io/>

- implementacija otvorenog koda nastala na temelju Googleovog projekta **Stubby**
- podržava niz jezika: Java, Go, C++, Python...
- koristi posebnu implementaciju za serijalizaciju podataka [Protocol Buffers](#)



Izvor: <https://grpc.io/docs/what-is-grpc/introduction/>

Pitanja za učenje i ponavljanje

- Objasnite zašto tranzijentna sinkrona komunikacija potencijalno pati od problema vezanih uz skalabilnost.
- Može li se pomoću UDP-a implementirati protokol za pouzdanu komunikaciju između klijenta i poslužitelja? Ako može, na koji način?
- Poslužitelj je implementiran pomoću socketa TCP na portu 10000 s ograničenjem `NUMBER_OF_THREADS=2`. Objasnite detaljno operacije prilikom dolaska prvog klijentskog zahtjeva na poslužitelj. Što se događa kada stigne drugi, pa treći klijentski zahtjev, a prve dvije konekcije su još uvijek aktivne? Koliko socketa je vezano uz port 10000?
- Koliko byte-a se može maksimalno zapisati u UDP datagram?

Literatura

- A. S. Tanenbaum, M. Van Steen: Distributed Systems: Principles and Paradigms, Second Edition, Prentice Hall, 2007
poglavlja 3 i 4
- G. Coulouris, J. Dollimore, T. Kindberg: Distributed Systems: Concepts and Design, 5th edition, Addison-Wesley, 2012
poglavlja 4 i 5