

Uvod u raspodijeljene sustave

1. Definicija, obilježja i vrste raspodijeljenih sustava

Definicija, obilježja i vrste raspodijeljenih sustava:

- "Skup neovisnih računala koji korisniku izgleda kao jedan cjeloviti sustav."
- "Sustav u kojem programske i sklopovske komponente umreženih računala komuniciraju uskladjuju svoje aktivnosti isključivo razmjenom poruka."

Sloj raspodijeljenog sustava:

- Prikriva činjenicu da su procesi i sredstva raspodijeljen na više računala
- Povezivanje i suradnju aplikacija, sustava i uređaja
- Između OS i aplikacijskih programa
- Iznad mrežnog ispod aplikacijskog

Razlozi za raspodijeljene sustave:

- Inherentna raspodijeljenost
 - Korisnika, informacija, sredstava
- Funkcionalno odvajanje:
 - Različite namjene, mogućnosti (korisnik-davatelj, proizvođač- potrošač)
- Opterećenje
- Pouzdanost i raspoloživost
- Cijena, troškovi, održavanje..

Vrste rasus:

1. Raspod računalni sustavi:
 - Grozd, nakupina (cluster)
 - Splet, cloud
2. Sveprisutni sustavi (pervasive system):
 - Nternet of things
 - Machine to machine (M2M)
3. Raspod informacijski sustavi:
 - Poslovni sustavi

- Transakcijski sustavi
4. Sustavi za pružanje informacijskih i komunikacijskih usluga:
- Usluge, aplikacije...

Obilježja rasus:

- Paralelne aktivnosti:
 - autonomne komponente sustava istodobno izvode više aktivnost
- komunikacija razmjenom poruka:
 - komponente razmjenjuju podatke, ne vade ih iz memorije
- Dijeljenje sredstava:
 - Zajedničkim sredstvima pristupa više komponenata
- Nema globalnog stanja:
 - Niti jedan proces ne zna stanje svih ostalih
- Nema globalnog vremenskog takta:
 - Ograničenost vremenskog usklađivanja

2. Zahtjevi na rasus (otvorenost, transparentnost, skalabilnost i kvaliteta)

1. Otvorenost:

- pruža usluge sukladno normiranim pravilima te definiranoj sintaksi i semantici
- Norma:
 - Široko prihvaćena u industriji (**de facto standard**) ili zastupana od tijela (**de jure standard**)
 - Dobro definirana
 - Javno dostupna
- Pretpostavka za :
 - Međudjelovanje
 - Prenosivost
 - proširljivost

2. Transparentnost:

- prikrivanje odabranih značajki raspodijeljenog sustava
- Transparentnost pristupa:
 - Prikrivanje razlika u pristupu resursima (razliciti OS, DB...)

- Lokacijska transparentnost:
 - i. Prikrivanje lokacije sredstva
- Migracijska transparentnost:
 - i. Prikrivanje promjene lokacije: promjena lokacije sredstva ne utječe na način dostupa sredstvu
- Relokacijska transparentnost:
 - i. Prikrivanje premještanja sredstva tijekom njegove uporabe

- Replikacijska transparentnost:
 - i. Prikrivanje više istovrsnih sredstava ili više preslika nekog sredstva, što zahtijeva primjenu istog naziva za sve replike
- Konkurencijska transparentnost:
 - i. Prikrivanje istodobne uporabe istog resursa od strane više korisnika: zajednička/dijeljena uporaba sredstva uz očuvanje konzistentnosti
- Transparentnost na kvar:
 - i. Prikrivanje kvara: otkrivanje kvara i obnavljanje sustava nakon kvara nije uočljivo korisnicima
 - ii. Problem otkrivanja kvara: veliko opterećenje može se očitovati kao kvar (npr. nema odgovora u očekivanom vremenu)

3. Skalabilnost:

- sposobnost razmjerne prilagodbe veličini (broj korisnika – količina sredstva), rasprostranjenosti (lokalno, regionalno, globalno, ...) i načinu upravljanja (jedna ili više administrativnih domena)
- kako bi se za više korisnika održale performanse treba:
 - i. više dijelova (koliko?)
 - ii. prostorno raspod (gdje?)
 - iii. koji komuniciraju asinkron (kako?)
- Tehnike koje omogućuju skalabilnost sustava:
 - i. Prikrivanje kašnjenja u komunikaciji (radi nešto korisno dok čekaš odg)
 - ii. Višestrukost raspodijeljena baza podataka
 - iii. Replikacija (problem konzistentnost originala i kopije)

4. Kvaliteta usluge:

- Quality of Service (skupni naziv za)

i. Performanse

response time - period od slanja zahtjeva do primitika

ii. Pouzdanost/raspoloživost

throughput - mjeri promet na serveru (bit/s)

iii. Ukupni trošak vlasništva

availability (usluga dostupna u trenutku t)

Quality of Experience - zadovoljstvo korisnika

Oblikovanje raus

- Kakve funkcijeske zahtjeve treba ostvariti – ŠTO sustav treba raditi?
- Kakve nefunkcijeske zahtjeve treba ostvariti – KAKO sustav treba raditi (kakva se kvaliteta usluge zahtijeva)?
- Temelji li se sustav na otvorenim rješenjima?
- Kakav je stupanj transparentnosti potreban i kako utječe na složenost, performanse i troškove sustava?
- Kakva je skalabilnost sustava potrebna s motrišta veličine, rasprostranjenosti i upravljanja?

3. Arhitektura rasus-a

Programska arhitektura:

- Logička organizacija sustava, programske komponente, njihova org i interakcija

Sustavska arhitektura:

- Centralizirana ili decentralizirana

Kako predočiti rasus?

1. Slojevita arhitektura

- U središtu aplikacijski sloj
- Aplikacijski programi te usluge koje im pružaju niži slojevi
- Funkcionalnost određuje aplikacijski sloj (najviši) a niži slojevi pružaju uslugu
- Točka međudjelovanja dva susjedna sloja je Service Access Point (SAP) a samo međudjelovanje opisuje se Service Primitives (SP)
- Service Primitives :
 - i. request - korisnik upućuje davatelju
 - ii. indication – izdaje ga davatelj usluge kako bi od korisnika zatražio izvršenje određene funkcije ili ga izvijestio da je primio zahtjev od drugog korisnika

- iii. *response*- korisnik kao odgovor na indikaciju
- iv. *confirm* – davatelj usluge kao odgovor na request

- Redoslijed SP-a je prilikom uspostave asocijacije:
 1. Korisnik (klijent) nižem sloju upućuje *request* za spajanjem na drugi proces
 2. Niži sloj prosljeđuje drugom korisniku *indication* kojom traži to spajanje
 3. Taj korisnik vraća *response* ako može provesti zatraženo
 4. Niži sloj vraća *confirm* do prvog korisnika ako je uspješno provedeno spajanje

2. Objektna arhitektura: (komponentama)

- Objekti su dijelovi sustava s dobro definiranim sučeljem mikrousluge --> dobro definirana sucelja - komunikacija i suradnje mikroservisa
- Mehanizam komunikacije, usklađivanja i suradnje objekata

3. Arhitektura temeljena na podacima:

- Procesi komuniciraju putem repozitorija
- Jedna komponenta objavljuje podatke u repo a druga čita

4. Arhitektura temeljena na događajima:

- Događaji prenose podatke
- Jedna komponenta se preplati na neki podatak a kad druga komponenta objavi podatak (događaj) onda se taj podatak dostavlja pretplaćenoj komponenti (događaj)

4. Osnovni modeli raspodijeljene obrade

1. Klijent poslužitelj

- Problemi sinkrone komunikacije
- Ako je poslužitelj „zauzet“ onda svih ostali zahtjevi čekaju
- Asimetrija dolaznih i odlaznih podataka moguća

2. Ravnopravni sudionici (peer):

- Može obaviti funkciju poslužitelja i klijenta
- P2P komunikacija tako da na app sloju povezuju u prekrivajuću mrežu nad stvarnom mrežnom topologijom
- Svaki čvor plaća sudjelovanje tako da nudi dio sredstava ostalim čvorovima
- Decentralizirani rasus
- Samoorganizirajuća mreža čvorova

3. Pokretni kod i programski agenti:

- **Process migration:**
 - Klijent raspolaže kodom međutim nedostaju mu sredstva za izvedbu, on šalje kod na server te se tamo izvrši kod (remote evaluation) i vraća mu se rezultat
- **Code on demand:**
 - Klijent nema kod, već po potrebi šalje serveru zahtjev, server iz repozitorija dohvaca kod, šalje ga klijentu gdje se izvršava lokalno i potom se briše
 - Primjer java appletA
- **Active message:**
 - Poruka sa kodom ide od čvora do čvora di bi se kod izveo na određenom
 - Analogija pokretnog objekta u objektnoj arhitekturi
- **Programski agent:**
 - program koji obavlja neki posao za vlasnika a raspolaže svojstvima kao što su inteligencija, samostalnost, proaktivnost itd.
- **Pokretni agent:**
 - Programski agent koji predstavlja korisnika i samostalno se kreće između čvorova u mreži
 - Smanjen mrežni promet jer se sve izvodi lokalno na poslužitelju/ klijentu
 - Moguće je na različitim čvorevima obavljati dio posla i na kraju dobiti rezultat obrade
 - Programska infrastruktura potrebna za izvedbu i izvršavanje agenata naziva se agentskom platformom (engl. agent platform) i mora se nalaziti u svakom čvoru koji udomljuje agente

5. Web kao case-study

WWW je stvoren i razvija se kao otvoreni sustav s transparentnim pristupom i konkurencijskom transparentnosti. Usluge pruža velikom broju korisnika predočenih klijentima sukladno normiranim pravilima te definiranoj sintaksi i semantici, uz očuvanje konzistentnosti sredstva.

- Transparentnost pristupa i konkurencijska transparentnost su ostvarene njegovim osnovnim postavkama. Sama zamisao „hiperteksta“ i normiranog jezika za označavanje (HTML) ključna je za transparentnost pristupa, dok je „konkurenca“ korisnika, tj. klijentskih zahtjeva, inherentna modelu klijent-poslužitelj

- Lokacijska transparentnost postiže se simboličkim sustavom imena gdje DNS vodi brigu o pretvorbi u IP adrese
- Migracijska transparentnost korisnik nema pojma je li promjenjena ip adresa dok nije promijenjen simbolički naziv
- Replikacijska transparentnost npr. Više posluzitelja iza simboličkog naziva gdje proxy usmjerava na pripadajući fizički poslužitelj

Procesi i komunikacija

Međuprocesna komunikacija (interprocess communication IPC) realizirana je proslijđivanjem poruka na mrežnom sloju tako što se podaci pakiraju u oblik prikladan za prijenos mrežom.

2.1 Obilježja komunikacije

Pravila komunikacije među procesima definiraju se komunikacijskim protokolom. Prvo važno obilježje komunikacije opisuje vrstu komunikacijskog protokola koji može biti *konekcijski* ili *beskonekcijski*.

Konekcijski protokol kad procesi prvo izmjene kontrolne poruke (uspostave konekciju logično) i pritom razmjene pravila, parametre komunikacije i tek onda kreće prijenos podataka.(TCP)

Beskonekcijski protokol ako se ne izmjenjuju kontrolne poruke prije slanja podataka. (UDP)

Perzistentna komunikacija:

- garantira isporuku poruke premda primatelj nije aktivan u trenutku nastanka i slanja poruke.
- Prepostavlja postojanje posrednika koji pohranjuje poruku do njene isporuke.

Tranzijentna komunikacija:

- Garantira isporuku poruke samo ako su pošiljatelj i primatelj istovremeno aktivni

Sinkrona komunikacija:

- Pošiljatelj je blokiran dok ne primi potvrdu od primatelja
- Može biti blokiran sve do primitka potvrde o tome da je:
 1. sloj raspodijeljenog sustava primio zahtjev te se dalje brine o isporuci poruke
 2. primatelj uspješno primio zahtjev nakon čega će započeti obradu
 3. primatelj uspješno primio i obradio poruku te uz potvrdu šalje i rezultate
- *pull* načelo (dovlači odgovor s servera)
- zbog potencijalno dugog čekanja na odgovor definiramo *listener* i tada imamo *push* načelo

Asinkrona komunikacija:

- prepostavlja da nakon slanja poruke pošiljatelj nastavlja obradu bez čekanja na povratnu informaciju, te se poruka jednostavno pohranjuje u izlazni spremnik pošiljatelja nakon čega operacijski sustav pošiljateljskog računala vodi računa o njenoj isporuci.

2.2 Obilježja procesa

Def. Proces je program u izvođenju na jednom od fizičkih ili virtualnih procesora računala.

Dva osnovna obilježja procesa:

1. vremenska (ne)ovisnost
 - vremenski ovisni procesi moraju istovremeno biti aktivni za realizaciju komunikacije
 - primjer SMS

2. ovisnost o referenci sugovornika

- anonimnost komunikacije
- Proces je ovisan ako mora znati jedinstveni identifikator, tj. adresu udaljenog procesa s kojim želi komunicirati (client-server)
- Primjer neovisnog procesa sustavi objava-preplata li dijeljenom podatkovnom prostoru

Tablica 2. Obilježja raspodijeljenih procesa

1.	vremenska ovisnost	vremenska neovisnost
2.	ovisnost o referenci "sugovornika"	neovisnost o referenci "sugovornika"
3.	iterativni poslužitelj	konkurentni poslužitelj
4.	<i>stateless</i> poslužitelj	<i>stateful</i> poslužitelj

Poslužiteljski procesi se u raspodijeljenim sustavima najčešće izvode višedretveno. Dretve se izvode konkurentno unutar jednog procesa te dijele isti adresni prostor. Višedretvenost rješava probleme koje izazivaju metode sa svojstvom blokiranja procesa (npr. Accept za poslužiteljski socket TCP-a).

Uobičajene zadaće klijenta:

- GUI
- Otvaranje mrežne konekcije prema serveru
- Primanje podatak

Uobičajene zadaće servera:

- Primanje zahtjeva
- Obrada podataka i rad sa bazama podataka

Poslužitelj:

- Iterativni:
 - Istovremeno samo jedan zahtjev
 - Neadekvatan za rasus
- Konkurentni:
 - Istodobno više zahtjeva
 - Složeniji, višedretvenost
- Stateless:
 - Ne pamti stanje klijentskih zahtjeva
 - Npr. Web poslužitelj (svaki HTTP je zasebna jedinica i neovisna o drugima)
- Stateful:
 - Obrnuto od stateless
 - Npr. Poslužitelj za pohranjivanje datoteka koji pohranjuje kopiju datoteke i radi tablicu kojom povezuje klijenta, kopiju i original datoteke

3. Sloj raspodijeljenog sustava za komunikaciju među procesima

Programski posrednički sloj ili međuoprema (engl. *middleware*) je programski sustav na aplikacijskom sloju koji se nalazi između transportnog sloja i aplikacije. Pojednostavljenje oblikovanja razvoja aplikacija te se izdaje u obliku library-a.

Međuoprema za komunikaciju udaljenih procesa

- Nalazi se na sloju rasus
- Implementira protokole među raspod procesima

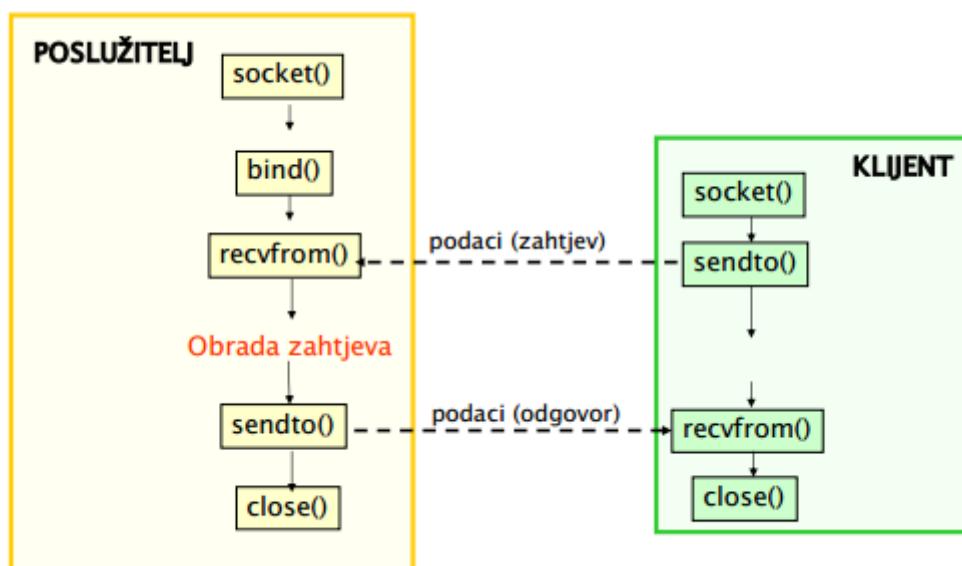
3.1 Komunikacija korištenjem priključnica

Priključnica (socket):

- Pristupna točka koja šalje podatke u mrežu i iz koje čita primljene podatke
- Viši nivo apstrakcije od komunikacijske točke koju OS koristi za pristup funkcijama transportnog sloja
- Veže se uz adresu i port

User Datagram Protocol (UDP):

- Nespojna usluga (bezkoneksijski protokol)
- Prenosi datagrame
- Poslani datagrami spremaju se privremeno u izlazni spremnik povezan s vratima iz izvorišne adrese na klijentu (ne blokirajuća operacija slanja jer se dalje odvija bez čekanja na potvrdu)
- Kod primanja moguće blokirajuće i neblokirajuće (blokirajuće kad je spremnik iz kojeg se čita prazan)
- Vremenska ovisnost procesa (server mora bit aktivran)
- Tranzijentna komunikacija
- Asinkrona komunikacija



Kako implementirati poslužitelja koji koristi UDP socket?

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. Obraditi pristigli paketa i po potrebi poslati odgovor klijentu
5. Zatvoriti poslužiteljski socket

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

Kako implementirati klijenta koji koristi UDP socket?

1. Kreirati socket

```
DatagramSocket clientSocket;
clientSocket = new DatagramSocket();
```
2. Kreirati paket (prazan, priprema za primanje paketa iz mreže)

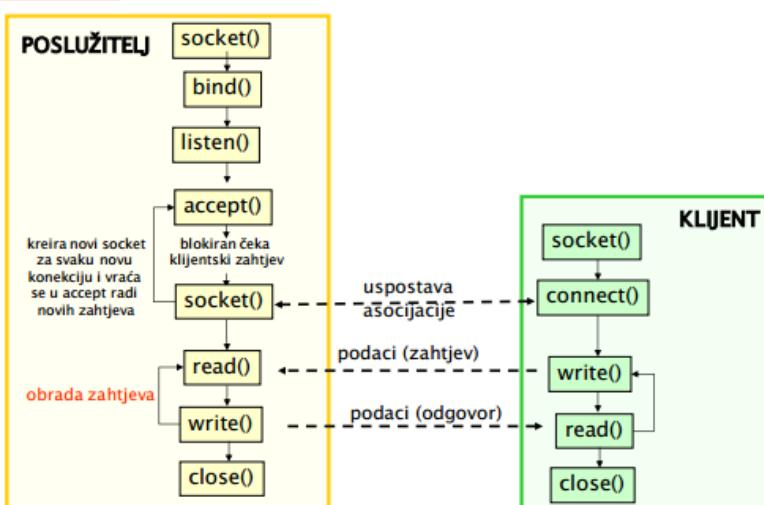
```
byte[] sendBuf = new byte[256];
DatagramPacket packet =
new DatagramPacket(sendBuf, sendBuf.length, destAddress, destPort);
```
3. Slanje paketa

```
clientSocket.send( packet );
```
4. Po potrebi obrada i čekanje odgovora
5. Zatvoriti socket

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

Transmission Control Protocol (TCP):

- Konekcijski protokol
- Pouzdan prijenos paketa uz kontrolu retransmisije
- Dva socketa (klijent + server)
- Client-server
- Vremenska ovisnost
- Klijent mora znati ID servera
- Tranzijentna komunikacija
- Sinkrona komunikacija
- Pull načelo



- *Socket*: kreira komunikacijsku točku, operacijski sustav rezervira sredstva koja će omogućiti slanje i primanje podataka pomoću odabranog transportnog protokola
- *Bind*: povezuje adresu sa socketom
- *Listen*: omogućuje os rezerviranje sredstava za max broj konekcija
- *Accept*: poslužitelj prima zahtjev za otvaranjem konekcije od strane klijenta (connect). Poslužitelj stvara novi socket koji se koristi za komunikaciju s klijentom. Originalni socket se koristi za primanje novih zahtjeva. Blokirajuća metoda !!

Kako implementirati poslužitelja koji koristi TCP socket?

1. Kreirati *socket* poslužitelja

```
ServerSocket serverSocket;
serverSocket = new ServerSocket( PORT );
```
2. Čekati korisnički zahtjev (proces je blokiran 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();
```

Kako implementirati klijenta koji koristi TCP socket?

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();
```

Višedretveni poslužitelj:

- Za implementaciju konkurentnog poslužitelja, potrebno je kreirati novu dretvu za svaki klijentski zahtjev tako da postoji jedna po konekciji
- Model koordinator/radna dretva (engl. dispatcher/worker model)
 - Koordinator prima zahtjeve i komunicira s klijentom
 - Ograničiti broj dretvi na strani poslužitelja *thread pool* (višak se ili odbauje ili spremi u red čekanja *backlog*)
 - Glavna ddretva samo prima konekcije i svaku (do MAX_KON) pokreće u svojoj dretvi

3.2 Poziv udaljene procedure/metode

Remote Procedure Call (RPC)

- omogućuje pozivanje i izvođenje procedura na udaljenom računalu.
- RPC je transparentan jer pozivajuća procedura nije svjesna da se proces izvodi na udaljenom računalu
- Koristi se posebna komponenta **stub** (klijent je blokiran za vrijeme izvođenja)
- Koraci poziva:
 1. Klijent poziva udaljenu proceduru
 2. Stub pakira parametre i ID procedure i poziva OS na klijentu
 3. OS klijenta šalje poruku na server
 4. OS servera predaje poruku stubu servera
 5. Stub servera raspakira pakete, izvadi parametre itd. i poziva proceduru
 6. Procedura vraća rezultat serverskom stubu
 7. Stub servera pakira rez u poruku i poziva OS
 8. Os servera šalje poruku OS-u klijenta
 9. OS klijenta predaje poruku stubu
 10. Stub raspakira rez i predaje klijentskom procesu
- Postupak „pakiranja“ parametara ili rezultata u poruku je *marshalling*
- Postupak „raspaketiranja“ odnosno čitanja podataka iz poruke je *unmarshalling*
- Ukoliko se prenose objekti tada se cijeli objekt zapiše u poruku (call by value) a ne preko referenca jer reference imaju smisla samo u adresnom prostoru procesa

Asinkroni RPC:

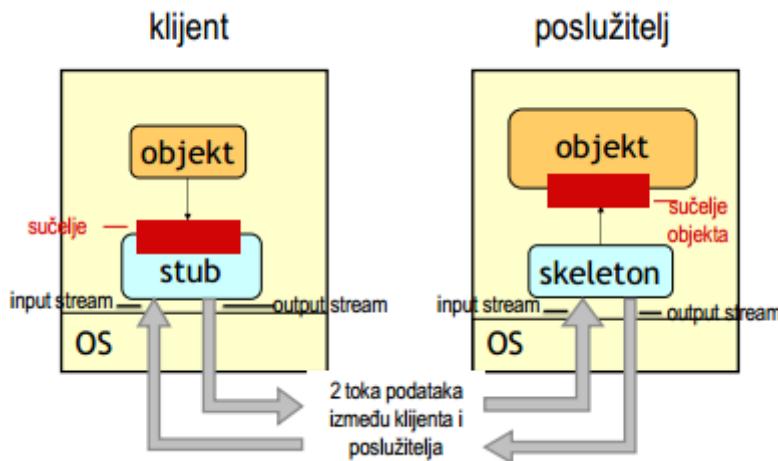
- Kako bi se smanjilo beskorisno čekanje



Slika 3.12. Odgođeni sinkroni RPC

Poziv udaljene metode (engl. Remote Method Invocation, RMI)

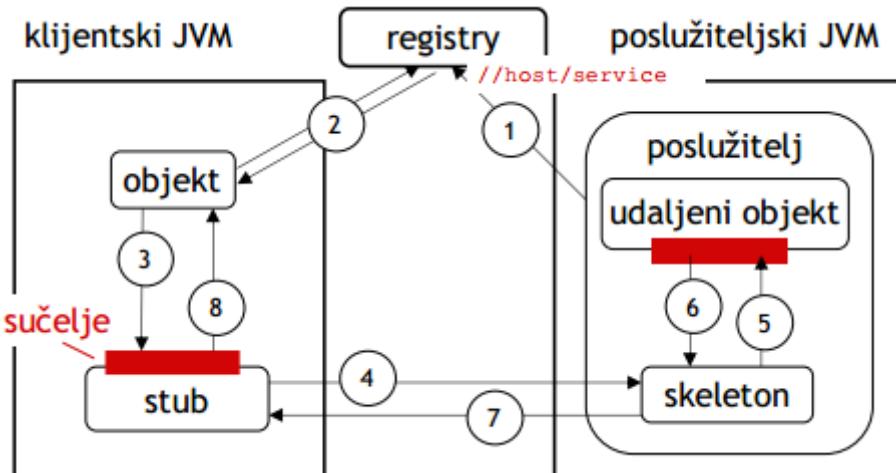
- RPC za objektne jezike jer se poziva metoda udaljenog objekta
- Sustav mora nuditi uslugu za registriranje i pronašetak udaljenih objekata (directory service)
- Ideja == RPC (samo s objektima)
- Skeleton == stub na serveru
- Stub u ovom slučaju implementira sučelje udaljnog objekta ali samo kao posrednik koji poziv metode i parametre pakira u oblik prikladan za slanje mrežom i šalje do skeletona koji dalje preusmjerava do objekta



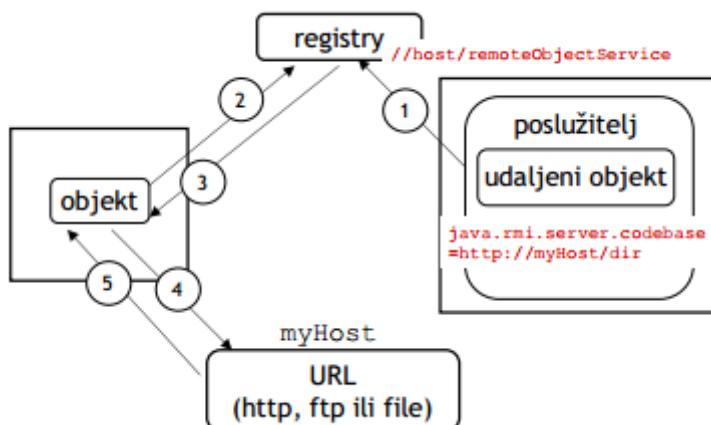
- Za razliku od RPC ovdje mogu postojati reference na objekte i prenose se kao parametar udaljene metode (call by reference)
- Obježja komunikacije poziva udaljene procedure/metode:
 - Klijent server
 - Vremenska ovisnost
 - Klijent mora znati id servera
 - Tranzientna komunikacija
 - Sinkrona komunikacija (osim kod asinkronog RPC)
 - Pull načelo pokretanja komunikacije

Java RMI (Remote Method Invocation)

- Transparentnost kao najvažnije svojstvo
- Referenca na udaljeni objekt istovjetna referenci na lokalni objekt s tom razlikom da svi udaljeni objekti moraju implementirati sučelje `java.rmi.Remote`
- Sučelje udaljenog objekta implementira stub (proxy) u adresnom prostoru klijentskog računala, a klase stub i skeleton se generiraju iz implementacije, a ne iz sučelja udaljenog objekta.
- Pri pozivu metode svi se parametri (primitivi i objekti) moraju serijalizirati te moraju implementirati sučelje `Serializable`
- U slučaju udaljenog objekta prenosi se njegova referenca koja je jedinstvena u raspodijeljenom sustavu, a sadrži adresu računala, broj vrata i identifikator udaljenog objekta



- Koraci komunikacije:
 1. Poslužiteljski objekt se **registrira** pod odabranim imenom (npr. //host/service).
 2. Klijent od **registrya traži referencu** na udaljeni objekt pomoću registriranog imena.
 3. Klijent poziva **metodu stuba** koji je dostupan na klijentskom računalu.
 4. Stub **serijalizira** parametre i šalje ih **skeletonu**.
 5. **Skeleton deserijalizira** parametre i poziva metodu udaljenog objekta.
 6. **Udaljeni objekt vraća rezultat** izvođenja metode skeletonu.
 7. **Skeleton serijalizira** rezultat i šalje ga stubu.
 8. **Stub deserijalizira** rezultat i dostavlja ga klijentu.
- S obzirom da se stub može mijenjati bog promjene implementacije udaljenog objekta, klijent mora pribavi najaktualniji stub u svoj adresni prostor



- Redoslijed akcija za dinamičko učitavanje klase stuba na stranu klijenta:
 1. Poslužitelj definira codebase **udaljenog objekta** i registrira taj udaljeni objekt pod odabranim imenom. (Codebase definira URL s kojega se Javine klase mogu učitavati u JVM.)
 2. Klijent od **registrya traži referencu** na udaljeni objekt koristeći registrirano ime.
 3. Registry vraća **podatke o klasi** stuba. Ako se klasa stuba može pronaći na klijentskoj strani učitava se lokalna verzija klase. U suprotnom će klijent učitati klasu koristeći definirani codebase.
 4. Klijent traži **klasu stuba** koristeći codebase.
 5. Klasa stuba se dostavlja klijentu. Klijent može pozivati metode udaljenog objekta koristeći primljeni stub

3.3 Komunikacija porukama

Komunikacija porukama spada u važnu skupinu programske infrastrukture za komunikaciju

raspodijeljenih procesa poznatu pod imenom message-queuing systems ili Message-Oriented Middleware (MOM). Omogućuje **perzistentnu i asinkronu** komunikaciju preko posrednika koji pohranjuje poruke, a da pri tom **izvor i odredište** ne moraju biti **istovremeno aktivni**.

Osnovna ideja:

- Preko posrednika tj. postoji rep na primatelju
- Pošiljatelju se garantira isporuka na rep ali ne i do primatelja
- Na rep se dodaje metodom *put* a čita sa *get i poll*
- *Get* briše kopiju iz repa ali blokira primatelja ako je rep prazan
- *Poll* provjerava stanje repa te ovisno o tome isporuči ili ne poruku
- Pošiljatelj i primatelj su vremenski neovisni
- Svaki rep ima jedinstveno ime (tzv. Adresa repa) neovisno o transportnoj adresi te je potrebna usluga koja povezuje ime s adresom (DNS analogija)
- Na posredniku postoi queue manager proces koji upravlja skupom repova

Obilježja komunikacije:

- Vremenska neovisnost
- Sender mora znati id odredište tj. repa
- Perzistentna komunikacija
- Asinkrona komunikacija
- Pull načelo (primatelj provjerava postoji li poruka za njega)
- Moguće i push načelo preko notify()

3.4 Komunikacija na načelu objavi-preplati

- Omogućuje asinkronu komunikaciju između publishera i subscribersa
- Preplata se može zamisliti kao predložak (engl. template) obavijesti
- Razmjena se odvija preko posrednika, *brokera*
- Povoljno utječe na skalabilnost sustava
- Obavijesti (notifications), preplate (subscriptions) i odjave preplata(unsupscription)
- Prikladno za sustave di se promjena vrijednosti odnosno događaj ne može predvidjeti
- Matching property:
 - Preplata prekriva obavijest ako zadovoljava sve uvjete definirane preplatom
 - Ako preplata prekriva obavijest onda se mora isporučiti
- Najraširenije vrste preplata:
 - Preplata na kanal
 - omogućuje tematsko grupiranje obavijesti i klasificira svaku obavijest
 - Kanal se može smatrati logičkom poveznicom među objavljuvачima i preplatnicima
 - Kanali se mogu organizirati hijerarhijski (vrijeme u svijetu, europi, hrvatskoj..)
 - Preplata na sadržaj
 - Složenije i fleksibilnije preplate
 - Definiraju se uvjeti nad atributima
 - Obavijest je najčešće točka u višedimenzionalnom prostoru(senzor...)
 - Preplata je potprostor višedimenzionalnog prostora

Arhitektura usluge objavi-pretplati:

- Centralizirano :
 - 1 posrednik sa cjelokupnim skupom preplatnika i publishera
 - Posrednik ima sve informacije o svemu u sustavu
 - **Najteže smisliti** algoritam
 - Glavni nedostatak **je skalabilnost i nepouzdanost** rješenja jer ako ispadne taj jedan čvor jede sve u k
- Raspodijeljeno
 - Mreža **posrednika** od kojih **svaki obrađuje dio** klijenata te je **zadužen za podskup** preplatnika koji su direktno na nj spojeni
 - **Skalabilnost na više poslužitelja**
 - Direktno se poslužuju lokalni objavljavači i preplatnici te se usmjeravaju obavijesti drugim posrednicima
 - Održava se tablica usmjeravanja:
 - Svaka poruka sadrži zaglavje s podacima o izvoru i odredištu poruke te prenosi obavijest, preplatu ili odjavu preplate
 - Kontrolne poruke (preplate i odjave) mijenjaju tablicu
 - Osnovne metode usmjeravanja:
 - **Preplavljanje:**
 - **Svaka primljena poruka se prosljeđuje** svima
 - **Filtriranje poruka:**
 - **Samo zainteresiranim**

Obilježja modela objavi-pretplati su sljedeća:

1. **vremenska neovisnost**: objavljavač i preplatnici **ne moraju istovremeno biti aktivni**, posrednik pohranjuje poruku
2. **objavljavač ne mora znati identifikator** preplatnika (**anonimnost**), o tome se brine posrednik
3. komunikacija je **perzistentna**
4. **asinkrona** komunikacija: objavljavač šalje poruku i nastavlja obradu neovisno o odgovoru od strane odredišta
5. pokretanje komunikacije na načelu **push**: objavljavač **šalje poruku** posredniku koji je prosljeđuje preplatnicima bez prethodnog eksplicitnog zahtjeva
6. **personalizacija** primljenog sadržaja: **filtriranje** objavljenih poruka prema preplatama
7. **proširivost** sustava: **dodavanje novog** objavljavača ili preplatnika ne utječe na ostale strane u komunikaciji
8. **skalabilnost**: **raspodijeljena arhitektura**

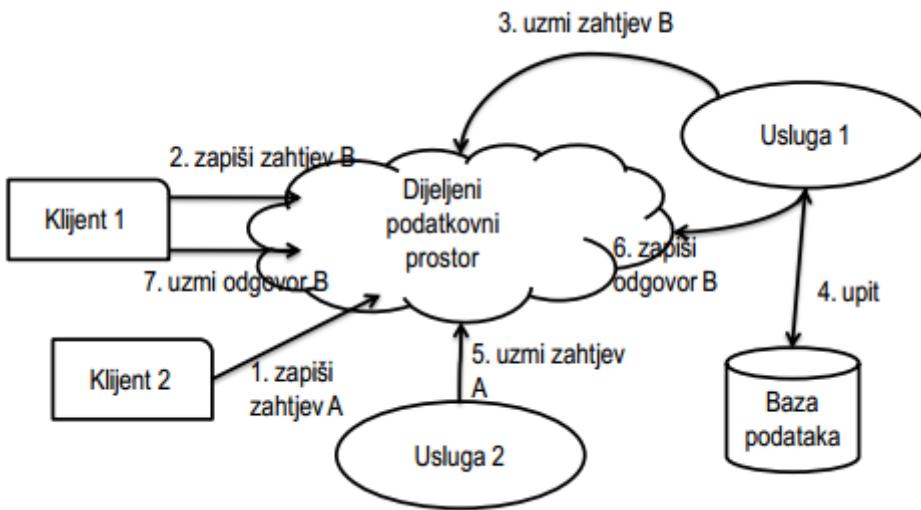
3.5 Dijeljeni podatkovni prostor

Temelji se na dijeljenoj perzistentnoj memoriji u koju se pohranjuju podaci (tzv. Tuple).

Slično kao i objavi-preplati iz memorije se čitaju podaci koji odgovaraju predlošku s razlikom što odredišta eksplisitno čitaju podatke a ne aktivna isporuka.

Operacije podržane:

- **Write(t)** dodaj tuple t
- **Read(s)** vraća tuple t koji odgovara template s
- **Take(s)** vraća tuple t koji odgovara template s i briše ga iz prostora



Obilježja komunikacije pomoći dijeljenog podatkovnog prostora su:

1. **vremenska neovisnost**: procesi ne moraju istovremeno biti aktivni radi komunikacije, dijeljeni prostor pohranjuje poruku
2. **anonimna komunikacija** (temelji se na sadržaju podataka)
3. komunikacija je **perzistentna**
4. **asinkrona komunikacija**: proces dodaje podatak u podatkovni prostor i nastavlja obradu
5. pokretanje komunikacije na **načelu pull**: proces eksplisitno šalje zahtjev za čitanje podatka iz dijeljenog podatkovnog prostora

4. Arhitekture web- aplikacija

4.1 HTTP protokol

- Zahtjev i odgovor
- Port 80
- GET, PUT, DELETE, POST, HEAD i OPTION - znaš
- TRACE- dijagnostika
- CONNECT- za buduću uporabu?
- Zaglavje prazan red tijelo
- TCP konekcija
- 1xx- Informative , 2xx- Success, 3xx- Redirecting, 4xx Client error, 5xx – Server error

4.2.1 HTML 5

- Najvažnija svojstva HTML-a 5 su:
 - Manje koda u zaglavlju
 - Više semantike u oznakama (header, nav, section...)
 - Oznake za multimediju (video, audio)
 - Geolokacija u JavaScriptu
 - Nove vrste podataka u formama
 - Spremnik podataka za offline rad
 - Drag
 - ...

4.3 CSS

Za uključivanje stila koristi se oznaka *link* unutar *head*. U oznaci imamo 3 atributa:

1. *Rel*- označava relaciju između HTML-ovog dokumenta i povezanog dokumenta, za stilove se koristi vrijednost *stylesheet*
2. *Type*- specificira vrstu dokumenta prema standardu MIME, za stilove se koristi *text/css*
3. *Href* – specificira lokaciju pomoću URL-a

Prilikom dohvata resursa, najprije se učita HTML i zatim prkeo link-a novim GET zahtjevom i CSS. Tek nakon što se i on dohvati , korisniku se prikazuje stranica u pregledniku.

- /* komentar ide ovdje*/
- Selektor mogu odabrati razne oznake:
 - Oznake:
 - h1 {...} pravilo vrijedi za sve h1
 - atribut id:
 - #prozor {...} svi atributi sa id vrijednošću prozor
 - Atribut class:
 - .center {...}

4.4 Priručna spremišta (cache)

U rasus se koristi za spremanje kopije resursa na mjestu povoljnijem za njegovo korištenje (bliže..).

Ideja:

- Kod prvog dohvaćanja resursa, spremi ga u cache
- Kod svakog slijedećeg zahtjeva za resursom vraćaj iz memorije ne preko mreže
- Dobrobit za klijenta, server i mrežu

Smještaj spremišta može biti:

- Na klijentu:
 - web-preglednik sprema sadržaj na disk klijentskog računala
- na strani servera:
 - kopija izvornog resusa (npr. HTML-ov dokument) ili izračunati rezultat (npr. odgovor web-tražilice na neki popularni upit) se spremi za kasniju dostavu klijentu. Ako neki klijent zatraži isti resurs onda mu pružatelj usluge može odmah proslijediti traženi sadržaj i ne mora ponovno izvršavati kôd koji generira rezultat. Na taj se način smanjuje opterećenje izvornog poslužitelja pružatelja usluge
- u mreži:
 - poslužitelj posrednik (engl. caching proxy)

Postoje 2 modela za upravljanje spremištem:

1. **Model roka trajanja**
2. **Model validacije (valjanosti dokumenta)**

4.4.1 Model roka trajanja

- Server definira koliko je resurs valjan/ svjež
- Dobar kada znamo kad će se i koliko često resurs mijenjati ili kada se rijetko mijenja
- Zaglavje *Expires* u HTTP (*Expires: Sat, 15 Oct 2011 11:49:28 GMT*)
 - problem ako satovi nisu sinkronizirani na serveru i klijentu
 - *Cache-Control* zaglavje (*max age=6497*)

4.4.2 Model validacije

- Jeli spremljena inačica i dalje svježa
- *Last-Modified* i *Etag*
 - Etag koristi hash (MD5) (klinet šalje zahtjev sa zaglavljem *If-None-Match: hash*)

4.4.3 Posrednički poslužitelj

- filtriranje informacija - npr. blokiranje nepoželjnog sadržaja za djecu ili članove organizacije;
- privremeno spremište - ubrzavanje posluživanja tj. manje korištenje mreže između
- posredničkog poslužitelja i poslužitelja usluga
- anonimiziranje klijenta – poslužitelj usluga vidi klijenta kao posrednički poslužitelj, a ne krajnjeg klijenta;
- bilježenje korištenja mreže - npr. u organizaciji se može bilježiti koji korisnik/skupina korisnika je koliko koristila mrežu;
- provjera sadržaja - npr. provjera na viruse, analiza informacija koje se šalju iz organizacije itd

Prema smještaju:

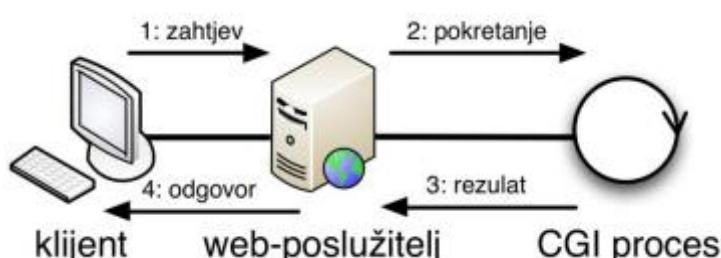
- lokalnoj mreži klijenta (forward proxy)
 - smanjuje promet između lokalne i vanjske mreže
- javnoj mreži (open proxy)
 - isključivo za anonimiziranje klijenata
- lokalnoj mreži na poslužiteljskoj strani (reverse proxy):
 - enkripcija ili komprimiranje podataka – rastereće se poslužitelj usluge (npr. webposlužitelj);
 - uravnoteženje opterećenja – posrednički poslužitelj bira na koji će poslužitelj usluga usmjeriti zahtjev klijenta i na taj način više poslužitelja usluga mogu paralelno obrađivati zahtjeve klijenata
 - privremeno spremanje izračunatih podataka – posrednički poslužitelj može svježe resurse spremiti u svoje priručno spremište i njih slati kao odgovor te na taj način rasteretiti poslužitelja usluge;
 - sigurnosne postavke – može dozvoliti pristup samo nekim uslugama i na taj način poslužitelj usluge ne mora biti toliko siguran jer ga štiti posrednički poslužitelj koji može osiguravati SSL komunikaciju između klijenta i posredničkog poslužitelja, a komunikacija između posredničkog poslužitelja i poslužitelja usluga ne treba biti zaštićena.

4.5 Web-aplikacije

Def. „Web-aplikacije su ostvarene na web-poslužiteljima i koriste alate kao što su baze podataka, JavaScript (Ajax ili Silverlight), PHP (ili ASP.Net) kako bi se omogućile funkcionalnosti koje nisu standardne web-stranice ili web-obrasci.“

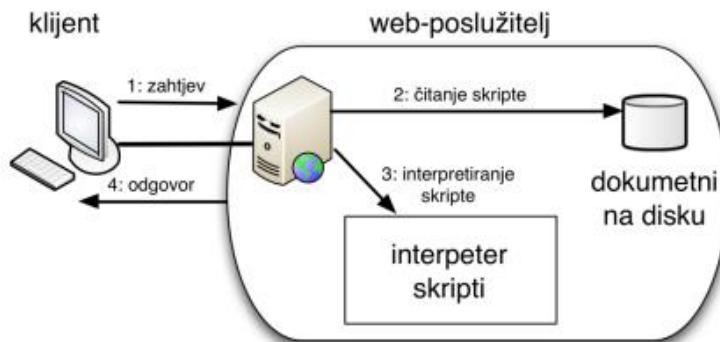
4.5.1 CGI (Common gateway Interface)

Jednostavno sučelje za pokretanje eksternih programa iz web-poslužitelja na platformski i programski neovisan način. Kod svakog zahtjeva se pokreće novi proces, a podaci između poslužitelja i procesa šalju se preko varijabli okoline i tokova podataka. Nakon svake obrade proces se gasi.



Nedostatak je e što se kod svakog zahtjeva pokreće novi proces i nakon obrade gasi što je zahtjevno za resurse (procesorsko vrijeme i memorija).

4.5.2 Poslužiteljske skripte (server-side scripts)



- za svaki zahtjev se NE pokreće novi proces
- PHP, ASP, JSO, Ruby on Rails...

4.6 Arhitekture web-aplikacija

Imaju višeslojnu arhitekturu:

- **Presentation layer (PL)**
- **Bussiness logic layer (BLL)**
- **Data access layer (DAL)**
- Zadnjih godina PL se preselio na klijenta pa se AJAX-om mijenja GUI a podaci se prenose npr. JSON-om ili XML-om

4.8 AJAX

- Tehnika koja služi za dinamičko dohvaćanje podataka sa servera
- AJAX omogućuje da se dijelovi stranice mijenjaju (ažuriraju asinkrono)
- Kako su pravila korištenja AJAX-a u preglednicima različita, potrebna je provjera vrste preglednika i izvršavanje kôda prilagođenog tom pregledniku

Problem chat-a:

- Jedan korisnik u browseru šalje poruku drugom korisniku koji koristi drugi browser
- Jedini način je AJAX preko servera ALI
- Poslužitelj može samo slat odgovor na zahtjev, ne može inicirati komunikaciju
- **Poll (prozivanje):**
 - Klijent periodički šalje zahtjeve i prima prazne odgovore ako nema ništa pametnog
 - Slaba iskoristivost mreže (puno praznih zahtjeva)
- **Long poll (dugo prozivanje):**
 - Klijent pošalje zahtjev ali server čeka sa odgovorom dok ne dobije podatke
 - Problem uvijek otvorene konekcije prema serveru (dok se čeka odg)

5. Web usluge

Pod pojmom web-usluge mogu se podrazumijevati dvije stvari: web-usluga koja se temelji na tehnologijama SOAP, WSDL, UDDI ili web-usluga koja je zapravo usluga/aplikacija na nekom web sjedištu npr. Facebook.

Obični web koriste ljudi dok su web usluge namijenjene za programe. Web services (WS) omogućuje lakše korištenje weba umjesto RPC/Java RMI...

Osnovna ideja:

- komunikacija se temelji na XML-u
- koristi već postojeću internetsku infrastrukturu i protokole
- usluge su dostupne putem Interneta
- omogućuje integraciju između različitih aplikacija
- ne ovisi o programskom jeziku ili zatvorenoj tehnologiji jedne tvrtke
- omogućuje otkrivanje usluga koje nude te aplikacije
- usluge su slabo povezane
- temelji se na industrijskim standardima

Web usluga je program koji :

- možemo identificirati URL-jem,
- komunicira s klijentskim programima putem Weba,
- ima sučelje (API) opisano standardima web-usluga i
- omogućuje korištenje neovisno o platformama i programskim jezicima

Osnovna skupina standarda:

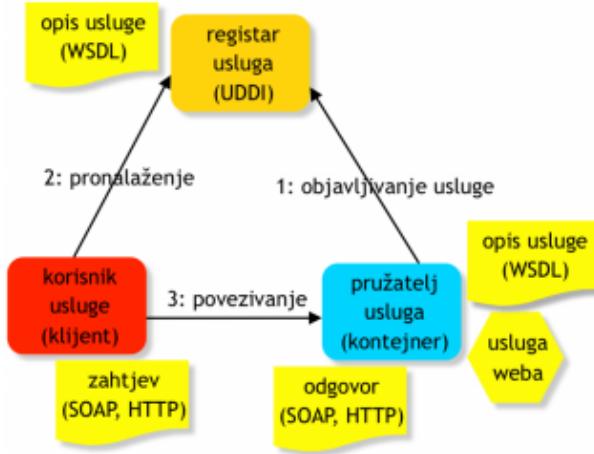
- WSDL (Web Services Definition Language) – opisuje uslugu,
- SOAP (Simple Object Access Protocol) – format poruke
- UDDI (Universal Description, Discovery and Integration) – za otkrivanje usluga

Druga generacija standarda omogućuje korištenje u poslovnom okruženju koje mora biti sigurno i fleksibilno:

- WS-Coordination – protokol za koordinaciju raspodijeljenih aplikacija,
- WS-Transaction – podrška za transakcije,
- BPEL4WS (Business Process Execution Language) – jezik za formalnu specifikaciju poslovnih procesa i interakcijskih protokola,
- WS-Security – sigurnosni protokol (TLS, integritet, privatnost, ...),
- WS-ReliableMessaging – za pouzdano slanje i primanje podataka,
- WS-Policy – za uređivanje pravila i prava,
- WS-Attachments – za slanje dodataka u zahtjevima i odgovorima,
- WS-Addressing – adresiranje usluga i poruka

Arhitektura web-usluge se sastoji od 3 entiteta:

- pružatelja usluge,
- registra usluga i
- korisnika usluge



- najprije davatelj registrira uslugu u registru preko UDDI i šalje opis u WSDL
- korisnik dohvata opis iz registra i dalje preko SOAP i HTTP komunicira sa davateljem

klijent-poslužitelj	web-usluge
unutar tvrtke	između tvrtki
ograničeno na skup programskih jezika	neovisne o programskom jeziku
proceduralno	temelji se na porukama
obično ograničeno na određeni transportni sloj	jednostavno se koristi različitim transportnim mehanizmima
jako povezani dijelovi	slabo povezane usluge
učinkovita obrada (prostor/vrijeme)	relativno neefikasna obrada

web-aplikacije	web-usluge
interakcija korisnik-program	interakcija program-program
statička integracija komponenti	moguća dinamička integracija komponenti
monolitna usluga	moguće je sastavljanje usluga od jednostavnijih

5.1.1 RPC Poziv udaljene procedure

- kritiziran jer usko povezuje klijenta i uslugu
- usluga usko vezana uz implementacijski jezik

5.1.2 Usluge temeljene na porukama/dokumentima

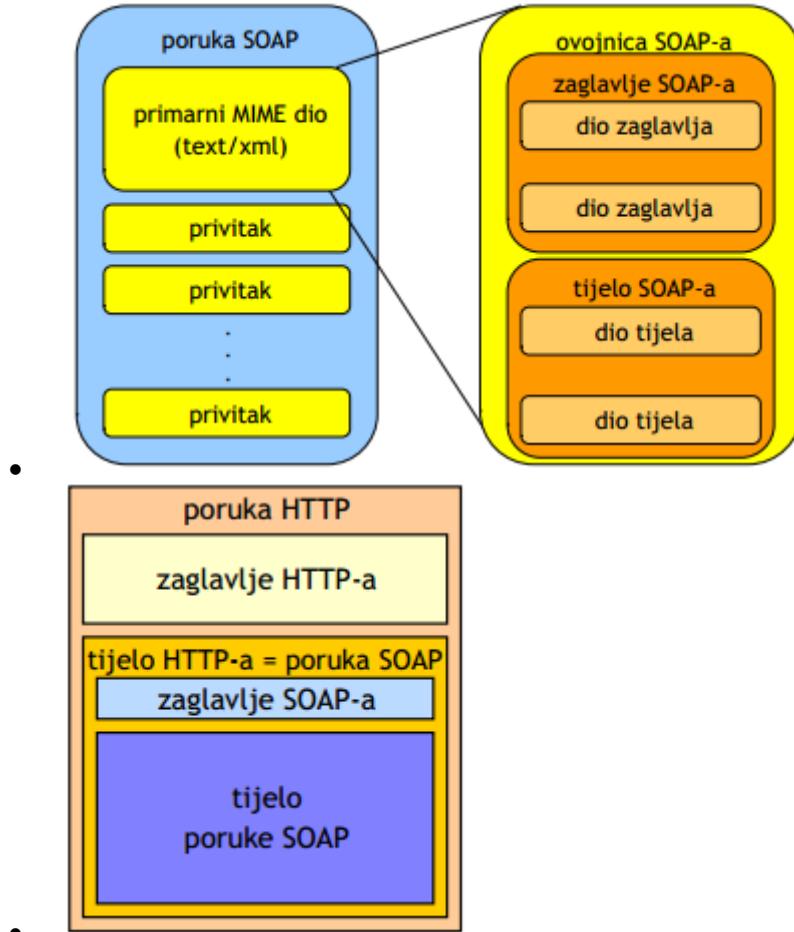
- SOA (Service Oriented Architecture)
- Implementacija temeljena na ugovorima propisanim WSDL-om (što se prenosi i kako?)

5.1.3 Usluge temeljene na prijenosu prikaza stanja resursa

- HTTP se koristi → jasno sučelje (GET, POST...)
- Može koristiti WSDL i SOAP
- RESTful

5.2 Protokol SOAP

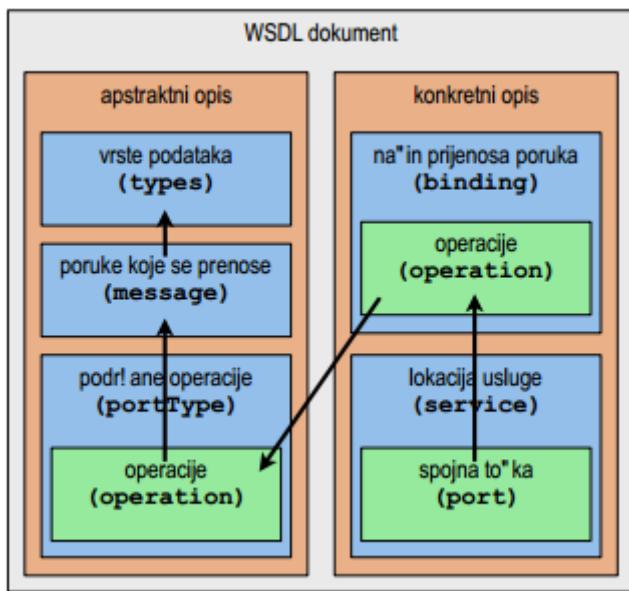
- Simbol Object Access Protocol omogućuje komunikaciju sa web uslugom
- **Poziv udaljenih procedura (RPC):**
 - Prijenos **serijaliziranih parametara** i rezultata
 - Dobro definirana sučelja i tipovi podataka
 - Prilagodni kod može bit generiran automatski
- **Razmjena dokumenata/poruka:**
 - Koriste se **XML dokumentu**



5.3 Jezik WSDL

Opis web-usluge se sastoji od dva dijela koji se međusobno referenciraju i stoga su povezani:

- apstraktnog opisa
- konkretnog opisa.



5.3.1 Apstrakti opis

Interakcija klijent-server sastoji se od razmjene poruka. Server prihvata poruku i može vratiti poruku ili baciti iznimku. Svaka poruka pripada nekoj vrsti, a svaka vrsta se sastoji od niza podataka.

Trebamo:

1. opisati sve vrste podataka u svim porukama
2. navesti sve poruke koje se mogu koristiti a svaka je poruka predstavljena nizom podataka
3. opisati svaku metodu/operaciju/proceduru kao kolekciju ulaznih, izlaznih i iznimnih poruka

Navedeni opisi moraju biti platformski i jezično neovisni.

Apstraktan opis se sastoji od 4 važna elementa:

1. **types**: definira vrste podataka neovisne o platformi i jeziku (koristi se XML Schema)
2. **message**: definiraju ulazne i izlazne poruke koje se mogu koristiti kao parametri usluge
3. **operation**: predstavlja jednu operaciju/metodu/proceduru koja je definirana u usluzi, a sastoji se od definicija ulaznih, izlaznih i iznimnih poruka koje se mogu razmjenjivati korištenjem ove operacije
4. **portType**: koristi poruke (pod 2) kako bi opisao sve operacije

5.3.2 Konkretni opis

Svaka usluga mora imati jedinstven URI preko kojeg joj se pristupa. Ima definiran protokol i format i te su definicije sadržane u konkretnom opisu koji se sadrži od 2 dijela:

1. *binding* :
 - definira kako je konkretna implementacija povezana s operacijama u apstraktnom opisu i definira format poruka. Definira transportne protokole i stil
2. *service* :
 - definira URI na kojem je usluga (definiran u spojnoj točki –port)

5.3.3 Element WSDL-a

- **definitions**
 - **osnovni element**
- **types**
 - **opis podatka pomoću XML-a**
 - samo kad se koriste podaci koji nisu osnovni
- **message**
 - **opis jednosmjerne poruke**
 - definira ime poruke
 - sastoji se od dijelova **part**
 - svaki dio se referencira na vrste podataka iz types
- **portType**
 - **definira operacije**
 - reference na poruke koje mogu biti:
 - parametri
 - rezultati
- **binding**
 - kako će se poruke iz operacije prenositi
 - **(HTTP GET, POST, SOAP, SMTP)**
 - Stil definira vrstu poruke:
 - **Rpc**
 - zahtjev će imati **omotač** u kojem će pisati naziv funkcije koja se poziva
 - **document**
 - zahtjev i odgovori će imati **“obične” XML dokumente**
- **services**
 - definira lokaciju URL
- **import**

5.4 UDDI

UDDI (Universal Description, Discovery, and Integration) je protokol koji definira na koji način se mogu objaviti i otkriti web-usluge. Obično se UDDI registar koristi unutar neke organizacije. Registrar se sastoji od 3 dijela:

- **imenika** (engl. white pages) – koji sprema **adrese, kontakte i identifikatore**,
- **poslovnog imenika** (engl. yellow pages) – koji ima **kategorizaciju područja, usluga i proizvoda te lokacije i**
- **tehničke informacije** (green pages) – koji sadrži **tehničke informacije** o uslugama (dokumente u WSDL-u).

Ne postoji centralni registar na Internetu, već svaka koristi organizacija vlastiti.

5.7 Web usluge temeljene na prikazu stanja resursa (REST)

Representational State Transfer je arhitekturni stil a ne API ili standard.

- Klijent šalje zahtjev za resursom (id resursa i vrsta podataka koji se očekuje)
- Odgovor sadrži prikaz resursa i niz parametara koji ga opisuju
- dohvati može uzrokovati promjenu stanja
- upiti su neovisni jedan o drugom

Metode koje se koriste:

- **GET:**
 - Dohvat resursa
 - **Sigurna**
 - **Idempotentnost**
 - **cachale**
- **POST:**
 - Slanje na obradu ili dodavanje u kolekciju
- **PUT:**
 - Srvaranje ili **obrada postojećeg resursa**
 - Idempotentnost
- **DELETE:**
 - Idempotentnost

Postupak izrade **ovakve usluge**:

1. Potrebno je identificirati entitete kojima se želi omogućiti pristupanje (npr. lista korisnika, podaci o korisnicima);
2. Za svaki resurs treba definirati shemu identifikatora pomoću URL-a. Shema treba završavati imenicama, a ne glagolima jer glagoli označavaju radnju. Radnja nije resurs. Npr.
<http://localhost:8080/REST2011/rest/persons> za popis korisnika i
<http://localhost:8080/REST2011/rest/person/id> za pojedinog korisnika s identifikatorom id;
3. Za svaki resurs treba odrediti da li je moguće samo dohvaćati podatke o resursu (korištenje metode GET) ili je potrebno i manipulirati s njima (korištenje metoda POST, PUT i DELETE);

4. Svi resursi koji se dohvaćaju pomoću metode GET za tu metodu moraju imati svojstvo sigurnosti tj. obrada metode GET na poslužitelju ne smije mijenjati stanje resursa;
5. Svaki resurs treba vratiti što manje detalja tj. treba stavljati poveznice na druge resurse u slučaju da klijent želi detalje resursa ili saznati povezanost s drugim resursima;
6. Usluge treba dizajnirati tako da se postepeno (kroz više upita koji se otkrivaju preko poveznica tj. URL-ova) otkrivaju podaci o resursima. Nije dobro sve staviti u jedan veliki dokument;
7. Za svaku metodu svakog resursa potrebno je definirati formate upita i odgovora. Ako se koristi XML onda je dobro za te opise koristiti XML Schema;
8. Opisati kako se web-usluge mogu koristiti npr. WSDL-om ili HTML-om.

5.8 Service Oriented Architecture (SOA)

- Bitno svojstvo je razdvajanje interesa (separation of concerns)
- Svaka se usluga može nadograđivati na druge usluge i na taj način stvarati kompleksne usluge
- SOA nije vezana za konkretnu tehnologiju iako je najčešće kao web-usluga

Svojstva uslužno orijentirane arhitekture su:

- slaba povezanost usluga – jedna usluga ne ovisi o tehnologiji implementacije druge usluge,
- korištenje uslužnih ugovora – vlasnici usluga i klijenata trebaju koristiti i ugovore da bi se riješio pravni segment,
- dogovor o komunikaciji (opis usluge – WSDL, XML Schema, policy, pravni dokumenti),
- autonomnost – klijent može biti autonoman,
- usluga ima kontrolu nad logikom koju enkapsulira,
- apstrakcija – dogovara se sučelje, a ne implementacija,
- izvana se vidi samo ono što je u ugovoru,
- ponovna iskoristivost – usluga se može iskoristiti kao dio druge usluge,
- uslugu mogu koristiti druge usluge,
- mogućnost slaganja u složene usluge,
- usluge bez stanja su skalabilne
- mogućnost otkrivanja usluga

6 Model Raspodijeljenog Sustava

6.1 Osnovni model

Osnovni model raspodijeljenog sustava se sastoji od skupa autonomnih i asinkronih procesa p_1, p_2, \dots, p_n koji komuniciraju razmjenom poruka putem komunikacijske mreže. Cij je kanal d pi do p_i samo to.

1. Ako se procesi izvode na različitim procesorima onda su međusobno autonomni i asinkroni
2. Procesi ne dijele memorijski prostor
3. Zbog kašnjenja paketa pri prijenosu na fizičkom liku, neminovno se javlja kašnjenje poruka pri komunikaciji procesa
4. Procesi ne koriste jedinstveni zajednički sat te je problem sinkronizacije vremena

Izvođenje procesa = akcije tijekom vremena. Akcije su:

- Unutarnji događaji (npr. obrada podataka na procesu koja proces dovodi u novo stanje)
- Slanje poruke
- Primanje poruke

Uzročna relacija označava uzročnu ovisnost 2 događaja tijekom raspodijeljenog izvođenja (\rightarrow). Vrijedi i tranzitivna uzročnost $a \rightarrow b$ i $b \rightarrow c \rightarrow a \rightarrow c$.

Uzročna neovisnost ako ne možemo sa sigurnošću reći koji je bio prije.

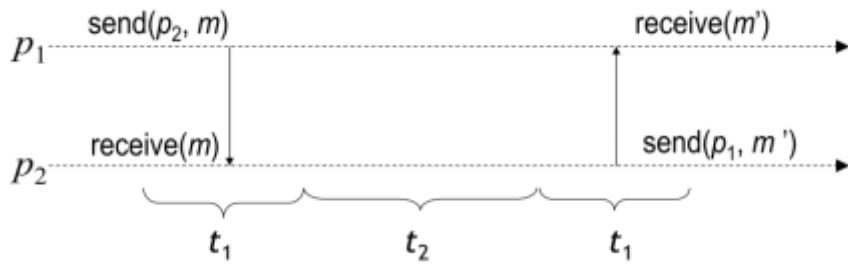
Slike isporuka poruka :

- FIFO
- Non-FIFO
- Kanal koji osigurava sinkronu slijednost (slanje i primanje istovremeno)
- -|- uzročnu slijednost

Globalno stanje raspodijeljenog sustava određeno je stanjem pojedinih procesa i komunikacijskih kanala. Stanje procesa određeno je stanjem lokalne memorije i izvođenjem unutarnjih događaja. Stanje kanala određeno je skupom primljenih i poslanih poruka. Izvođenje bilo kojeg događaja mijenja lokano stanje procesa/kanala, ali istovremeno i globalno stanje raspodijeljenog sustava.

Osnovni model rassu proširuje se kao sinkroni ili asinkroni model sustava:

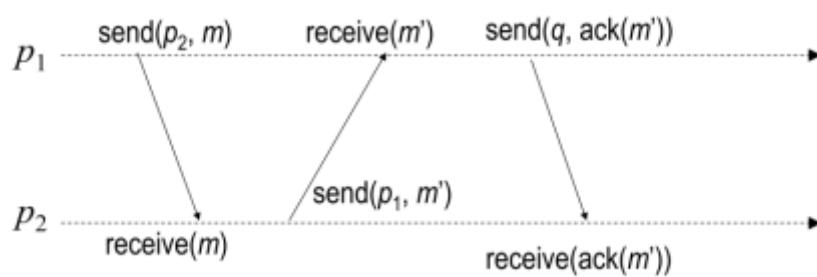
- Sinkroni:
 - Pojednostavljenje (isključivo) pretpostavka da svi procesi izvode korakre istovremeno
 - Svi procesi imaju sinkronizirana vremena lokalna
 - Poznata gornja vremenska granica za
 - Izvođenje prijelaza nekog procesa
 - Trajanje prijenosa poruke kanalom



-
- Inicialno svi procesi u početnom stanju i svi kanali prazni
- Prva faza-proces generira poruke i postavlja ih na izlazne kanale
- Druga faza- tren stanje+ poruka == prijelaz

- Asinkroni:

- Akcije u proizvolnjom slijedu
- Kompliciran te se najčešće koristi sinkroni za objašnjavanje
- Prepostavke:
 - Procesi NEMAJU sinkronizirana lokalna vremena
 - Ne postoji gornja granica vremenska za izvodnje prijelaza nekog procesa no uvijek je konačno
 - Ne postoji vremenska gornja granica za prijenos poruke kanalom



-

6.2 Sinkroni model

Lokalnost- svaki proces poznae samo svoje neposredne susjede, ulazne i izlazni ali **NEMA centralnog procesa** koji poznae sve o svemu u topologiji.

Model procesa:

$v_i \in V$ se modelira kao uređena četvorka: $(states_i, start_i, msgs_i, trans_i)$

$states_i$ – skup mogućih stanja procesa

$start_i$ – skup početnih stanja, $start_i \subset states_i$, $start_i \neq \emptyset$

$msgs_i$ – funkcija za generiranje poruka koja određuje izlaznu poruku za svakog susjeda na temelju trenutnog stanja procesa, tj. $msgs_i : states_i \times out-nbrs_i \rightarrow M_i \subset M \cup \{\text{null}\}$

$trans_i$ – funkcija prijelaza, određuje sljedeće stanje na temelju trenutnog stanja i primljenih poruka od ulaznih susjeda

Algoritmi se izvode nad sinkronim modelom u koracima. Inicialno svi procesi u poč stanju a svi kanali prazni. Korak ima 2 faze:

1. **Faza:** primjeni $msgs$ za svaki proces v na temelju tren. Stanja. Poruke se postavljaju na izlazne kanale i proslijeđuju se susjedima

2. Faza: primjeni **trans** za svaki proces v , koja na temelju tren. Stanja i primljenih poruka određuje sljed. Stanje . briše sve poruke na kanalima.

Mjera složenosti algoritama:

1. **Vremenska složenost**
 - Mjeri se brojem izvedenih koraka
2. **Komunikacijska složenost**
 - Mjeri se broj pripremljenih i poslanih poruka na kanalima

Primjer 1. Odabir vođe u sinkronom prstenu

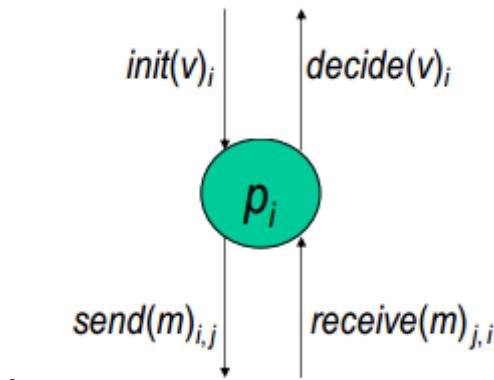
- Svi procesi su jednaki osim UID-a koji su usporedivi ($< >$)
- Proces random može odabrauti UID i zna za svoje susjede
- Jednosmjerna komunikacija (kazaljka na satu)
- Procesi ne znaju veličinu n
- **Vođa je najveći UID**
- Susjedu se šalje UID, ako je veći od vlastitog prosljijedi, ako je == proglaši se vođom, inače 0
- Vremenska složenost je $O(n)$
- **Komunikacijska složenost** je $O(n^2)$
- Leader šalje posebnu poruku *halt* kako bi ostalima javio da je algoritam gotov onda je **složenost $O(2n)$**

Primjer 2. Odabir vođe u usmjerenoj mreži

- Svi parovi procesa su na konačnoj udaljenosti
- Svaki proces ima UID
- Temelji se na **preplavljanju**
 - Svaki proces ima **UID koji se random odabire**
 - Svaki proces zna diameter(G) mreže G
- U svakom koraku šalje **svim susjedima max vrijednost** koju ima zapisanu (inicijalno svoju)
- Vremenska složenost je **$\text{diametter}(G)$** a komunikacijska je $diameter(G) \times |E|$

6.3 Asinkroni model

- Svaki se proces i svaki kanal definira **ulazno-izlaznim automatom** koji definira svaku komponentu rasus njenu interakciju sa ostalim komponentama
- Primitak ulazne vrijesnosti za varijablu v je **$init(v)$** , odluka **$decide(v)$** je izlaz
- Proces pi komunicira sa ostalim procesima pomoću **$send(m)ij$ i $receive(m)ji$**



- I/O automat A se definira kao uređena četvorka $(\text{sig}(A), \text{states}(A), \text{start}(A), \text{trans}(A))$ i sastoji se od sljedećih komponenti:
 - $\text{sig}(A)$ je **signatura** automata A , $\text{sig}(A)=\{ \text{in}(A), \text{out}(A), \text{int}(A) \}$, je skup koji definira ulazne, izlazne i unutarnje događaje automata A
 - $\text{states}(A)$ definira skup mogućih **stanja automata**
 - $\text{start}(A)$ je skup početnih stanja automata, $\text{start}(A) \neq \emptyset$ i podskup je skupa $\text{states}(A)$
 - $\text{trans}(A)$ je **funkcija prijelaza** koja za svako stanje s iz skupa $\text{states}(A)$ i svaki ulazni događaj π iz skupa $\text{in}(A)$ definira prijelaz $(s, \pi, s') \in \text{trans}(A)$ gdje je $s' \in \text{states}(A)$ novo stanje automata

Primjer 1: odabir vođe u asinkronom prstenu

- Prsten je modeliran pomoću 2 vrste I/O automata
 - Jedan za proces
 - Drugi za kanal
- FIFO kanal
- Svaki proces ima ulazni spremnik od max n poruka

Signatura je definirana sljedećim ulaznim i izlaznim događajima:

- in(A): $\text{receive}(v)_{i,i}$, gdje je v UID
- out(A): $\text{send}(v)_{i,i+1}; \text{leader}_i$

Skup mogućih stanja $\text{states}_i = \{ u, \text{send}, \text{status} \}$

- u je UID, inicijalno UID za proces p_i
- send je FIFO queue UID-ova veličine n , inicijalno sadrži UID za proces p_i
- $\text{status} \in \{\text{unknown}, \text{chosen}, \text{reported}\}$, inicijalno unknown

Funkcija prijelaza trans_i :

- $\text{send}(v)_{i,i+1}$ – preduvjet: v je 1. element iz send , posljedica: briši v iz send
- leader_i – preduvjet: $\text{status} = \text{chosen}$, posljedica: $\text{status} := \text{reported}$
- $\text{receive}(v)_{i-1,i}$
 - if $v > u$: add v to send
 - if $v = u$: then $\text{status} := \text{chosen}$
 - if $v < u$: do nothing

7 SINKRONIZACIJA PROCESA U VREMENU

Def. Sinkronizacija procesa u vremenu označava koordinaciju njihova izvođenja i međudjelovanja u svrhu skladnog rada cjelokupnog sustava tijekom vremena.

Za ispravni rad nekog raspodijeljenog sustava neophodno je osigurati usklađeno djelovanje njegovih raspodijeljenih procesa. Četiri osnovna razloga za primjenu sinkronizacije procesa u raspodijeljenoj okolini su:

1. uporaba dijeljenih sredstava, - ako 2 procesa pokusaju pristupiti
2. usuglašavanje vremenskog redoslijeda akcija, - ako ovise procesi medusobno
3. nadgledanje i upravljanje zadaćama skupa procesa te
4. uspostava suradnje skupa procesa - p2p za dijeljenje datoteka

7.1 Primjena sata u raspodijeljenoj okolini

- Javljuju se slijedeći problemi:
 - Satovi imaju različita odstupanja
 - Vrijednosti satova nisu usklađene
 - Satovi imaju različiti takt
- Koriste se dva pristupa:
 - Fizički sat
 - Logički sat- bitan redoslijed a ne točni vremenski trenutci

7.1.1 Usklađivanje fizičkih satova

- Dva najčešće korištena algoritma:
 - Criatianov- satovi se usklađuju stvarnim vremenom
 1. Korisnik šalje zahtjev poslužitelju. Zahtjev putuje t_z vremenskih jedinica do poslužitelja.
 2. Poslužitelj prima zahtjev te unutar sljedećih t_p vremenskih jedinica obrađuje primljeni zahtjev i šalje odgovor klijentu.
 3. Odgovor putuje t_o vremenskih jedinica do klijenta, a sadrži točno vrijeme T_3 na poslužitelju u trenutku primanja zahtjeva i točno vrijeme T_4 na poslužitelju u trenutku slanja odgovora.
 4. U zadnjem koraku klijent prima odgovor te na osnovi vremenskih trenutaka koje je primio poruci i izmjereni vremenskih trenutaka T_1 i T_4 pomiče svoje lokalno vrijeme za $T_3 + t_o - T_4 \approx T_3 + \frac{t_z+t_o}{2} - T_4 = T_3 + \frac{(T_4-T_1)-(T_3-T_2)}{2} - T_4 = \frac{(T_2-T_1)-(T_3-T_4)}{2}$.
 - Problem je neusklađenosti satova na poslužitelju i na računalu
 - Nedovoljno precizan jer vrijeme dolaska signala do servera i odlazak sa servera je aproksimiran
 - Sinkronizacija je dobra ako je RTT = T_4-T_1 puno manje od preciznosti koja se želi postići

- Berkley – međusobno se usklađuju
 1. Upravitelj šalje svoje trenutno vrijeme preostalim procesima u raspodijeljenoj okolini te od njih traži da mu pošalju svoja vremena.
 2. Procesi primaju zahtjev, određuju razliku svog trenutnog lokalnog vremena u odnosu na vrijeme primljeno u poruci te zatim upravitelju šalju odgovor u kojem je zapisana razlika izračunatih vremena.
 3. Upravitelj prima odgovore te računa pomak za satni mehanizam svakog pojedinog procesa u raspodijeljenoj okolini pa i sebe samog te zatim izračunate pomake proslijedi odgovarajućim procesima.

7.1.2 Usklađivanje logičkih satova

- Logički sat je funkcija koja vrši mapiranje sa skupa događaja na oznake vremena uz očuvanje konzistentnosti
- Odnosno ako $a \rightarrow b$ tada je $T(a) < T(b)$, ukoliko vrijedi i obrat onda pričamo o strogoj konzistentnosti

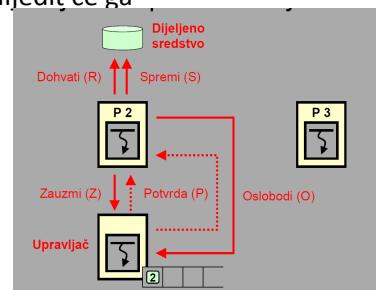
7.2 Sinkronizacija tijeka izvođenja procesa

7.2.1 Mehanizam semafora

- Semafor je proces koji sadrži spremnik s konačnim brojem znački i repom.
- Značke FIFO
- Kada ostane bez znački (sve je podijelio) sve zahtjeve spremi u rep
- Tijek komunikacije (D P V) :
 1. Proces šalje zahtjev semaforu za dohvata (D) jedne ili više znački,
 2. Ako u spremniku semafora postoji traženi broj znački, semafor će procesu predati značke (P),
 3. Ukoliko spremnik ne posjeduje dovoljan broj znački, zahtjev će se staviti u rep čekanja i
 4. Nakon završetka obrade, proces će vratiti semaforu preuzete značke slanjem poruke vrat (V)

7.2.2 Mehanizam razmjene događaja

- Posrednik događaja je proces sa 2 spremnika:
 - Spremnik s objavljenim događajima
 - Spremnik preplata
- Tijek komunikacije sastoji se od :
 - Proces posredniku šalje svoju preplatu (S) koju on sprema u spremnik
 - Neki drugi proces objavljuje događaj (P) posredniku koji ga sprema
 - Ako posrednik ima spremljenu preplatu za objavljeni događaj, proslijedit će ga procesu preplatniku dojave(N)



7.3 Međusobno isključivanje procesa

7.3.1 Isključivanje putem središnjeg upravljača

- FIFO
- Tijek komunikacije:
 1. Proces šalje zahtjev za zauzimanjem dijeljenog sredstva (Z)
 2. Proces ostvaruje pristup dijeljenom sredstvu nakon primitka potvrde (P) od strane središnjeg upravljača. Proces obavlja akciju dohvaćanja (R) i/ili spremanja (S) dijeljenog sredstva

3. Nakon završetka obrade, proces otpušta zauzeto sredstvo slanjem poruke **oslobodi** (O) središnjem upravljaču.

7.3.2 Raspodijeljeno isključivanje

- Svaki proces **ima vlastiti rep** čekanja a svi međusobno razmjenjuju informacije potrebne za usklađivanje **stanja svih repova**
- Razmjenjuju se 2 vrste poruka:
 - Zahtjev za pristup:
 - Uključuj logičku oznaku trenutka kojeg prosljeđuje svim drugim procesima
 - Potvrda prihvaćanja zahtjeva
- Svi šalju svima, **bira se** onaj sa **najmanjim vremenom** (najprije je došao) tj. ostali ga **potvrđuju**, kada on odradi svoje šalje svima potvrdu o kraju i tako daje...

7.3.3 Isključivanje zasnovano na primjeni prstena

- „**značka**“ za **dodijelu** pristupa
- Algoritam:
 1. Čekaj **poruku sa značkom od lijevog susjeda** (tad imaš pristup)
 2. Ako si zainteresiran **onda čitaj/spremaj**
 3. Prosljedi **poruku sa značkom desno** od sebe
- Ako svaki proces 1 pristupi za cijeli krug potrebno je $T = N * (T_z + T_p + T_o) + N * T_t$ gdje je:
 - $T_z \rightarrow$ vrijeme slanja zahtjeva
 - $T_p \rightarrow$ vrijeme primitka poruke odgovora
 - $T_o \rightarrow$ vrijeme obrade
 - $T_t \rightarrow$ vrijeme prijenosa poruke sa značkom

8 KONZISTENTNOST I REPLIKACIJA PODATAKA

- 8.prezu si preskocio
- 7.preza je na pola
- 9. preza je ovdje

Replikacija označava stvaranje i **održavanje istovrsnih kopija**. Razlozi mogu biti razni. Više kopija == veća pouzdanost. Problem nastaje kada se jedna od replika (ili original) promijeni.

Postupci očuvanja konzistentnosti su složeni jer treba postići:

- Transparentnost pristupa podacima
- Transparentnost lokacijsku i migracijsku
- Konkurencijsku Transparentnost
- Transparentnost na kvar

8.1 Modeli konzistentnosti podataka

Skup procesa: { p,q,r,s,...}, skup podataka {A,B,C,D,...}, skup lokacija {x,y,z,w,...}.

Svaki proces može neki podatak zapisati negdje npr. proces p upisuje podatak A na lokaciju x operacijom pisanja $W \rightarrow W(x, A)$.

Modeli konzistentnosti podataka govore o tome kako slijed upisa podataka na istu ili različite lokacije od strane jednog ili više procesa, „vide“ drugi procesi koji očitavaju te lokacije.

8.1.1 Stroga konzistentnost

- Odgovara idealnoj situaciji kada je promjena na jednoj lokaciji momentalno vidljiva na svim
- U **stvarnosti**, promjene će na drugim lokacijama **biti vidljive tek nakon komunikacijskog kašnjenja**

8.1.2 Slijedna konzistentnost

- Zahtijeva da **svi procesi** moraju slijed izvođenja **operacija čitanja i pisanja** u vremenu vidjeti na **jednak** način
- Bitno je da su **zajednički** učitani elementi u **istom redoslijedu**, npr. A, B , C i A,B,D,C su **A=>B** **B=>A** slijedno konzistentni

8.1.3 Povezana konzistentnost

- Oblik slabije slijedne konzistentnosti kod koje se pozornost obraća samo potencijalno povezanim operacijama koje mogu biti u **uzročno posljedičnoj vezi** i takve operacije se **moraju vidjeti jedna za drugom**

$$\begin{array}{c} A \Rightarrow B \Rightarrow C \\ A \Rightarrow C \Rightarrow B \end{array}$$

8.1.4 Konzistentnost redoslijeda upisivanja

- Ukoliko je redoslijed upisivanja podataka od jednog procesa vidljiv jednakost ostalim procesima
- Npr. proces **x** upisuje redom **A, B**. U ostalim procesima mora biti **očuvan taj redoslijed** (može biti i npr. A, C, B)
- FIFO
- Postiže se dodavanjem jedinstvenih oznaka zahtjevu za upisivanjem npr. $Wp1(x, C..)$

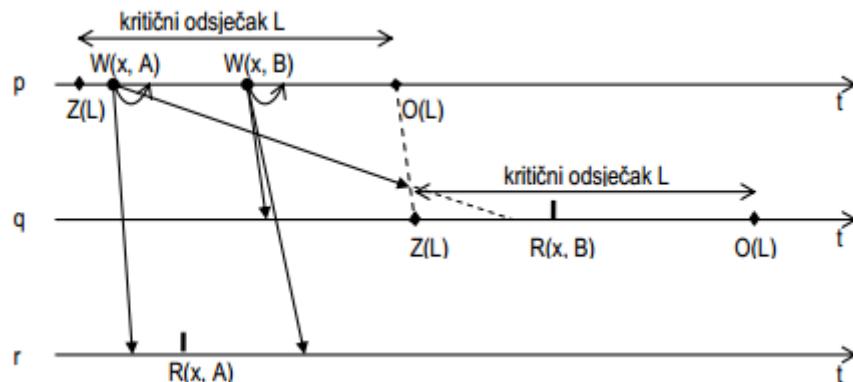
8.1.5 Slaba konzistentnost

- ostvaruje se primjenom **sinkronizacijskih varijabli** kojima se upravlja **usklađivanjem replika podatkovnog objekta**

- pri završetku upisivanja podataka proces pokreće sinkronizaciju prema ostalim procesima preko varijable Sync(S) kako bi svi vidjeli zadnje upisanu vrijednost (zbog mogućeg kašnjenja bi neki procesi mogli vidjeti prije upisanu vrijednost)

8.1.6 Konzistentnost otpuštanja i zauzimanja kritičnog odsječka

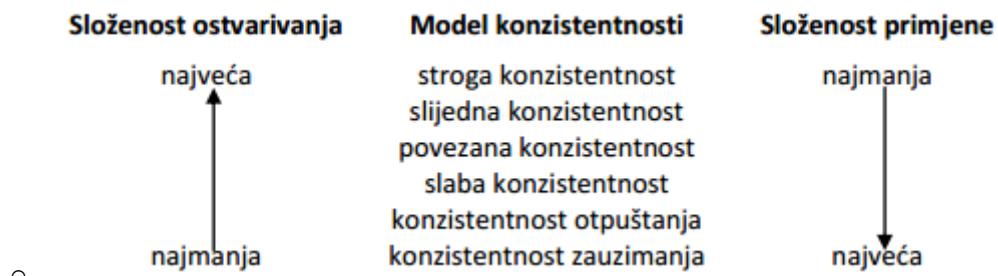
- ulazak u KO određen je operacijom zauzimanja dok je izlazak određen operacijom otpuštanja
- konz otpuštanja odgovara modelu kod kojeg se prije izlaska iz KO sve lokalne promjene prosleđuju svim replikama



- konz zauzimanja odgovara modelu kod kojeg se nakon ulaska u KO preuzimaju sve promjene podataka i uspostavlja konz.

8.1.7 Usporedba modela konzistentnosti

- mogu se usporediti sa 2 motrišta:
 - složenosti ostvarivanja : Opisuje koliko je zahtjevno ostvariti programsku potporu za uspostavu modela konzistentnosti
 - složenosti primjene: Opisuje koliko je složeno koristiti takav model



8.2 Organizacija sustava replika i vrste replika

Glavni problemi replikacije:

1. Odabir lokacije gdje će se smjestiti replike
 - a. Želimo li povećanje pouzdanosti i raspoloživosti ili
 - b. Poboljšanje performansi
2. Postavljanje replika na te lokacije
3. Održavanje konzistentnosti

Replike, ovisno o lokaciji dijelimo na:

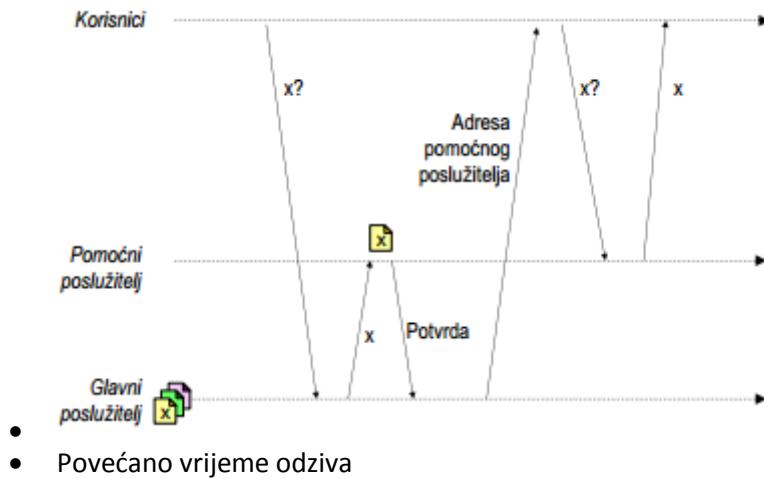
- Trajne
- Poslužiteljske
- Korisničke

8.2.1 Trajne replike

- Početni skup postavljenih replika na skupu replikacijskih servera povezani lokalnom mrežom (grozd računala)
- Statična organizacija (obavlja admin)
- Primjer korištenja ove vrste replika kod weba je prilikom raspodjele opterećenja između poslužitelja na istoj geografskoj lokaciji i raspodjele opterećenja zrcaljenjem (engl. mirroring) replika na različitim geografskim lokacijama
- Pristup takvim serverima preko raspoređivača zahtjeva

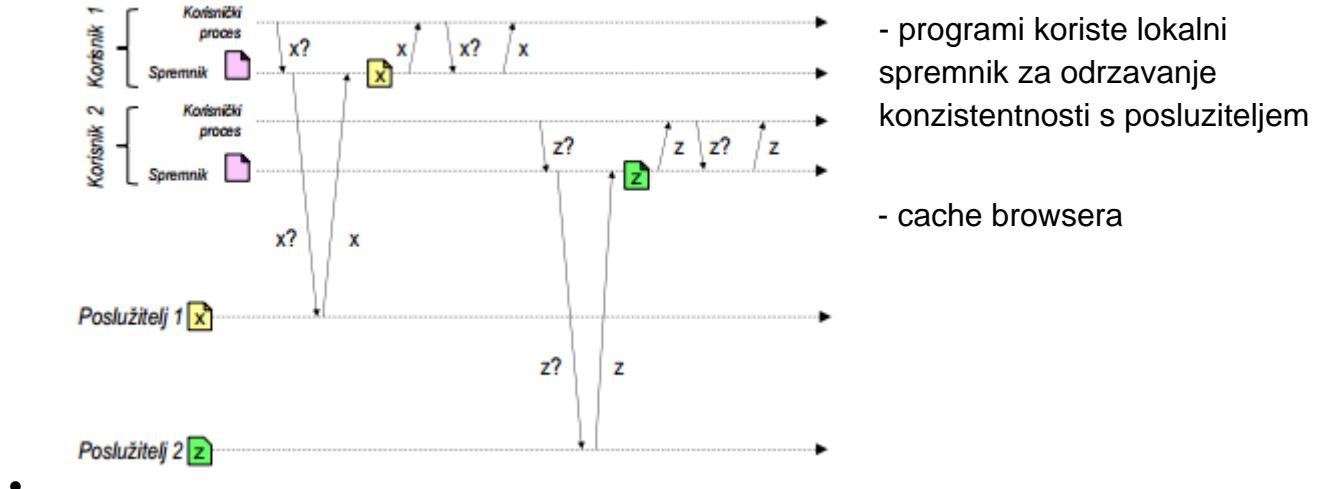
8.2.2 Serverske replike

- Dinamički, u real-time stvorene replike prilikom povećanja potražnje za nekim resursom



- Povećano vrijeme odziva

8.2.3 Korisničke replike

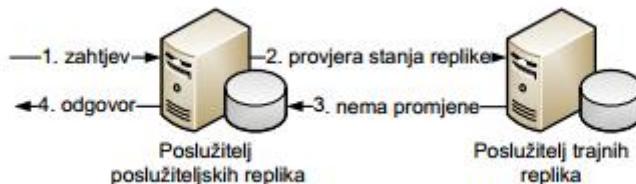
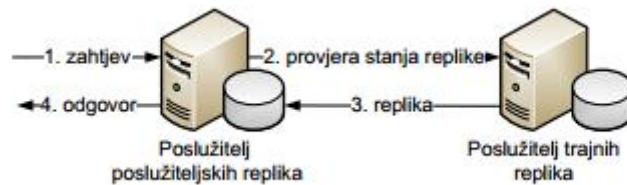


8.3 Metode održavanja konzistentnosti replika

Razlikujemo **pull** i **push** metode ovisno dali se **traži** ili **prosljeđuje**.

8.3.1 Dohvaćanje promjena sadržaja replika

- Klijent dohvaca resurs, najprije se provjerava je li podatak iz cache-a validan sve do vrha
- **Učinkovitost** se računa:
 - Ako je $fr(\text{frekv. Čitanja}) > fw(\text{frekv.pisanja})$ gornji slučaj fw a donji fr-fw
 - Ako je $fr \leq fw$, donji slučaj $\rightarrow 0$ a gornji fr



8.3.2 Prosljeđivanje promjena sadržaja replika

- Uvijek se čita samo iz lokalnog spremnika
- Kad nastupi promjena, trani server šalje niz hijerarhiju :
 - Novi sadržaj (cijela ili samo dio)
 - Operaciju za promjenu sadržaja
 - Obavijest o promjeni.
 - Zatim se koristi pull
- Metoda push je učinkovitija kada je $fr > fw$ jer bi sa pull pri svakom čitanju bespotrebno provjeravali je li došlo do promene
- Nedostatak je što server trajnih replika mora imati adrese svih poslužitelja replika te opise stanja njihovih replika

8.4 Protokoli za ostvarivanja operacija čitanja i upisivanja replika

U praksi je čest slučaj gdje korisnik/klijent inducira promjenu na replikama. Postoje protokoli kojima omogućuju ostvarivanje čitanja i pisanja replika od strane korisnika. Dijele se na protokole sa aktivnom i pasivnom replikacijom.

8.4.1 Pasivna replikacija

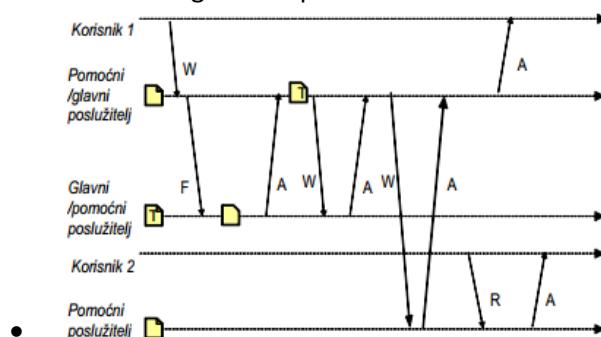
- Razlikuju glavnu (trajnu) repliku od ostalih te se operacije pisanja provode nad tom replikom dok se operacije čitanja provode nad bilo kojom replikom (najčešće najbližom)

1. **Udaljeno obnavljanje stanja replike:**

- Prepostavlja statičnost glavne replike
- Korisnik piše na glavnu repliku i onda se prosljeđuju promjene dalje
- Uspostava slijedne konzistentnosti
- Povećano vrijeme izvođenja operacije

2. **Lokalno obnavljanje stanja replike:**

- Dinamična glavna replika



- uzastopno vrijeme pisanja u kratkom vremenu na domaćinu
- korisnik može pristupiti lokalnoj replici neovisno o trajnoj



SVEUČILIŠTE U ZAGREBU



Diplomski studij

Računarstvo

Znanost o mrežama

Programsko inženjerstvo i
informacijski sustavi

Računalno inženjerstvo

**Ostali (slobodni izborni
predmet)**

Raspodijeljeni sustavi

1. Raspodijeljene arhitekture programskih
sustava. Centralizirana i decentralizirana
rješenja.

Ak. god. 2022./2023.

Sadržaj predavanja

- **Definicija, obilježja i vrste raspodijeljenih sustava**
- Zahtjevi na raspodijeljene sustave: otvorenost, transparentnost, skalabilnost i kvaliteta usluge
- Arhitektura raspodijeljenih sustava
- Primjeri modela raspodijeljene obrade
- Studijski primjeri:
 - Raspodijeljeni sustav weba
 - Internet stvari

Definicija raspodijeljenog sustava (1)

Andrew S. Tanenbaum:

- “Skup neovisnih računala koji korisniku izgleda kao jedan cjeloviti sustav.”

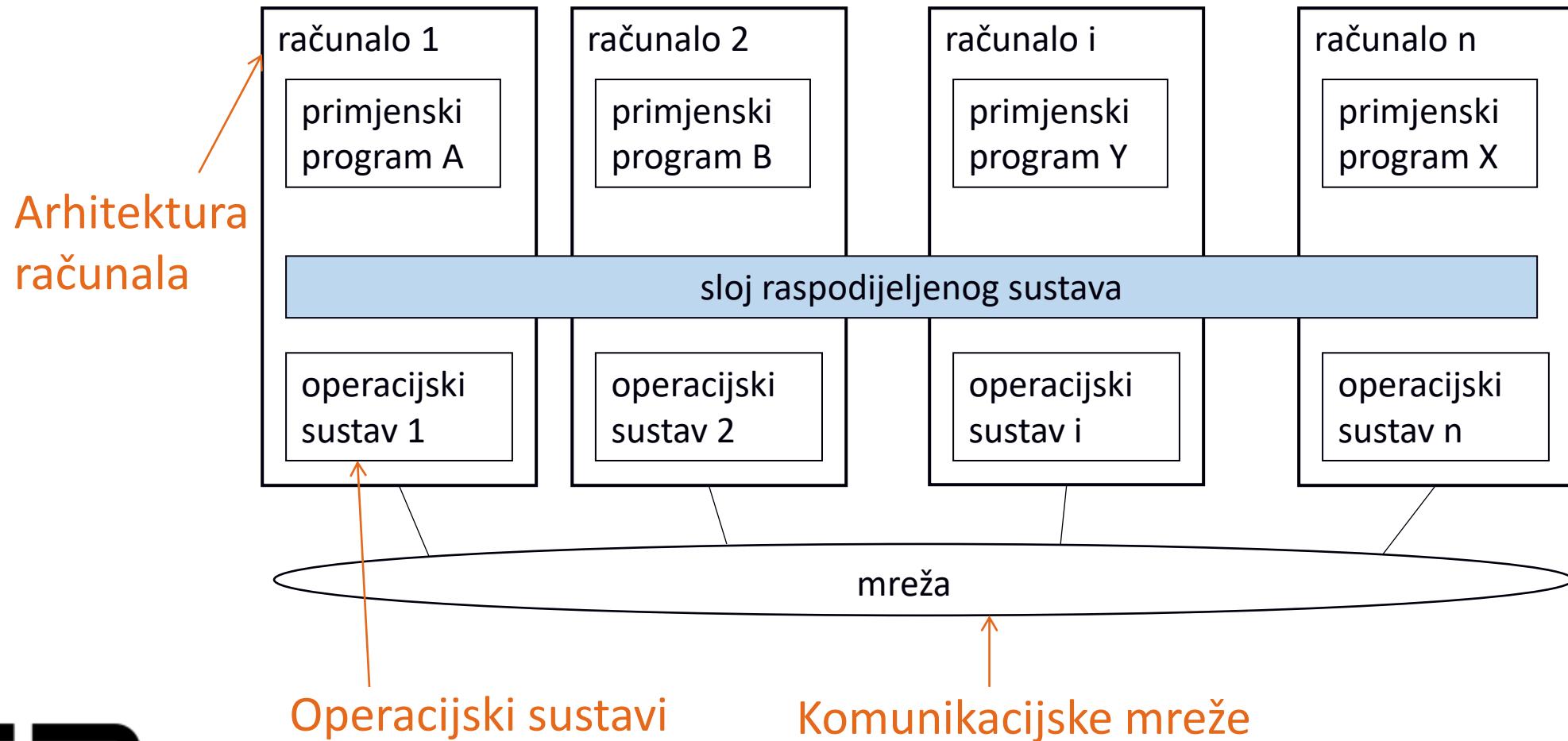
George Coulouris:

- “Sustav u kojem programske i sklopovske komponente umreženih računala komuniciraju i usklađuju svoje aktivnosti isključivo razmjenom poruka.”

Leslie Lamport:

- “Sustav u kojem kvar računala za koje uopće ne znate da postoji može učiniti vaše računalo neupotrebljivim.”

Definicija raspodijeljenog sustava (2)



Sloj raspodijeljenog sustava

Programski posrednički sloj raspodijeljenog sustava, međuoprema
(engl. *middleware*)

- prikriva činjenicu da su procesi i sredstva (resursi) raspodijeljeni na više umreženih računala
- omogućuje povezivanje i suradnju aplikacija, sustava i uređaja
- omogućuje interakciju programa na aplikacijskoj razini

Razlozi za raspodijeljene sustave

Inherentna raspodijeljenost:

- korisnika, uređaja, stvari, informacija, sredstava, ...

Funkcijsko odvajanje:

- različite namjene, različite mogućnosti, različite uloge (korisnik – davatelj usluge, proizvođač – potrošač,)

Opterećenje:

- mogućnost raspodjele i uravnoteženja

Pouzdanost i raspoloživost:

- ostvarene s više komponenata na različitim mjestima

Cijena, troškovi, održavanje...

Vrste raspodijeljenih sustava (1)

Raspodijeljeni računalni sustavi:

- grozd računala (engl. *cluster*)



Računalni klaster Isabella

<http://www.srce.unizg.hr/isabella>

135 računalnih čvorova s 3100 procesorskih jezgri i 12 grafičkih procesora

- splet računala (engl. *grid*)



CRO NGI, Hrvatska nacionalna grid infrastruktura

<https://www.srce.unizg.hr/cro-ngi>

1.868 procesorskih jezgri, 36 grafičkih procesora i 205 TB podatkovnog prostora

- računalni oblak (engl. *cloud*)



<https://www.srce.unizg.hr/hr-zoo>

Vrste raspodijeljenih sustava (2)

Sustavi za pružanje informacijskih i komunikacijskih usluga

- usluge (Skype, Facebook, MS Teams, Gmail, Twitter...)

Raspodijeljeni poslovni i transakcijski sustavi:

- poslovni i transakcijski sustavi (Amazon online shop, Bitcoin, ...)

Internet stvari (engl. *Internet of Things*, IoT)

- povezivanje stvari/uređaja na Internet – fizičkih i virtualnih objekata

Internet svega (engl. *Internet of Everything*, IoE)

- inteligentno povezivanje ljudi, procesa, podataka i stvari

Obilježja raspodijeljenog sustava

Paralelne i konkurentne aktivnosti:

- autonomne komponente sustava istodobno izvode više aktivnosti

Komunikacija razmjenom poruka:

- komponente sustava razmjenjuju podatke porukama, ne dohvaćaju ih iz zajedničke memorije

Dijeljenje sredstava:

- zajedničkim sredstvima pristupa više komponenata sustava

Nema globalnog stanja:

- niti jedan proces ne zna stanje svih ostalih procesa u svim komponentama sustava

Nema globalnog vremenskog takta:

- ograničena mogućnost vremenskog usklađivanja

Temeljni teorijski modeli

Modeli raspodijeljenih sustava sastoje se od procesa koji međusobno komuniciraju i razmjenjuju poruke putem komunikacijskog kanala

Temeljni formalizmi:

- **Komunikacijski i interakcijski model:** procesi, komunikacija, vremenska usklađenost odvijanja i komunikacije procesa
- **Model kvara:** kvarovi i njihov utjecaj na odvijanje i komunikaciju procesa
- Model sigurnosti: prijetnje odvijanju i komunikaciji procesa te mjere zaštite (ne obrađuje se u okviru predmeta Raspodijeljeni sustavi)

Sadržaj predavanja

- Definicija, obilježja i vrste raspodijeljenih sustava
- **Zahtjevi na raspodijeljene sustave: otvorenost, transparentnost, skalabilnost i kvaliteta usluge**
- Arhitektura raspodijeljenih sustava
- Primjeri modela raspodijeljene obrade
- Studijski primjeri: raspodijeljeni sustav weba, Internet stvari

Zahtjevi na raspodijeljene sustave

- **Otvorenost**
 - otvoreni sustav (engl. *open system*): pruža usluge sukladno normiranim pravilima te definiranoj sintaksi i semantici
- **Transparentnost**
 - prikrivanje odabranih značajki raspodijeljenog sustava
- **Skalabilnost**
 - sposobnost razmjerne prilagodbe veličini (broj korisnika – količina sredstva), rasprostranjenosti (lokalno, regionalno, globalno, ...) i načinu upravljanja (jedna ili više administrativnih domena)
- **Kvaliteta usluge**
 - performance (npr. vrijeme odziva), raspoloživost/pouzdanost, trošak

Otvorenost

Norma ili standard je specifikacija koja je:

- široko prihvaćena u industriji (*de facto standard*) ili zastupana od normizacijskog tijela (*de jure standard*),
- dobro definirana,
- neutralna, tj. vlasnički neovisna i
- javno dostupna.

Otvorenost je preduvjet za:

- međudjelovanje (engl. *interoperability*)
- prenosivost (engl. *portability*)
- proširljivost (engl. *extensibility*)

Transparentnost (1)

Transparentnost pristupa (engl. *access transparency*)

- prikrivanje razlika u pristupu sredstvima i predočavanju podataka (različite arhitekture računala, različiti operacijski sustavi, različite baze podataka, ...)

Lokacijska transparentnost (engl. *location transparency*)

- prikrivanje lokacije sredstva: položaj sredstva u sustavu ne treba biti i nije poznat korisniku
- primjer: poslužitelj www.fer.unizg.hr čiju lokaciju (IP-adresu) zna DNS-poslužitelj

Transparentnost (2)

Migracijska transparentnost (engl. *migration transparency*)

- prikrivanje promjene lokacije: promjena lokacije sredstva ne utječe na način pristupa sredstvu

Relokacijska transparentnost (engl. *relocation transparency*)

- prikrivanje premještanja sredstva tijekom njegove uporabe: sredstvu se može pristupiti i može se upotrebljavati tijekom njegove relokacije, tj. premještanja

Replikacijska transparentnost (engl. *replication transparency*)

- prikrivanje više istovrsnih sredstava ili više preslika nekog sredstva (sve replike nude istu funkcionalnost)

Transparentnost (3)

Konkurencijska transparentnost (engl. *concurrency transparency*)

- prikrivanje istodobne uporabe istog resursa od strane više korisnika: zajednička/dijeljena uporaba sredstva uz očuvanje konzistentnosti

Transparentnost na kvar (engl. *failure transparency*)

- prikrivanje kvara: otkrivanje kvara i obnavljanje sustava nakon kvara nije uočljivo korisnicima
- problem otkrivanja kvara: veliko opterećenje može se očitovati kao kvar (npr. nema odgovora u očekivanom vremenu)

Skalabilnost (1)

Kako bi se uz promjenu broja korisnika održale performance sustava uz prihvatljive troškove treba osigurati:

- više (istovrsnih) dijelova koliko?
- prostorno raspodijeljenih gdje?
- koji komuniciraju kako?

Primjeri neskalabilnih rješenja

- centralizirana usluga: jedan poslužitelj za sve korisnike
- centralizirani podaci: jedan poslužitelj sa svim korisničkim podacima
- centralizirani algoritam: svi podaci o sustavu poznati na glavnom procesu

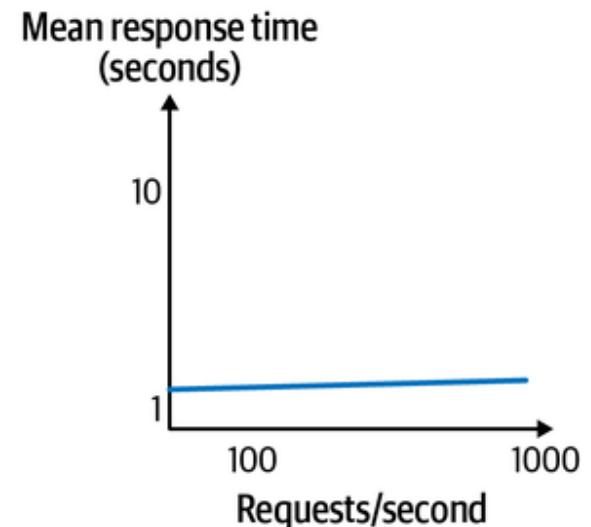
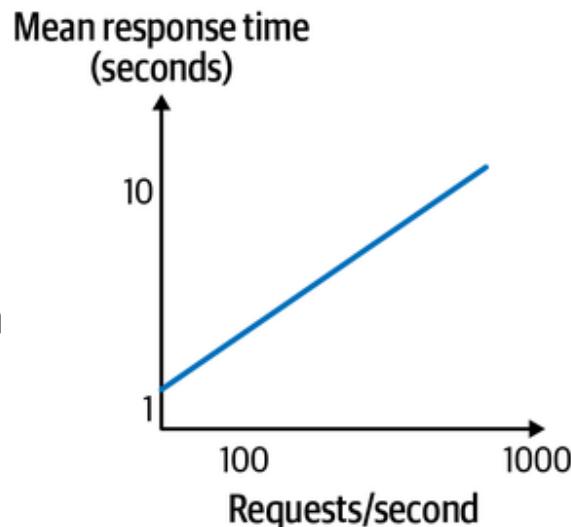
Mogu li se centralizirana rješenja nositi s povećanim brojem korisnika?

- Za motivaciju pogledati <https://www.internetlivestats.com/>

Skalabilnost (2)

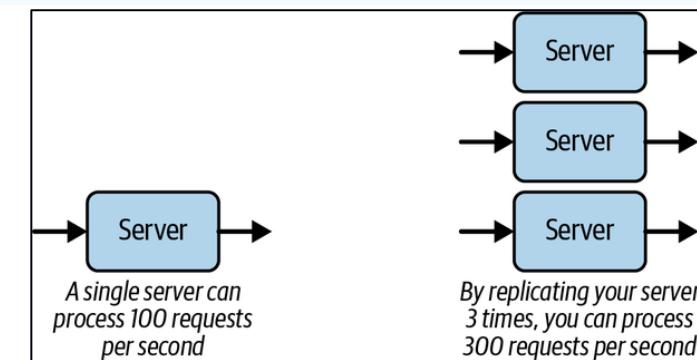
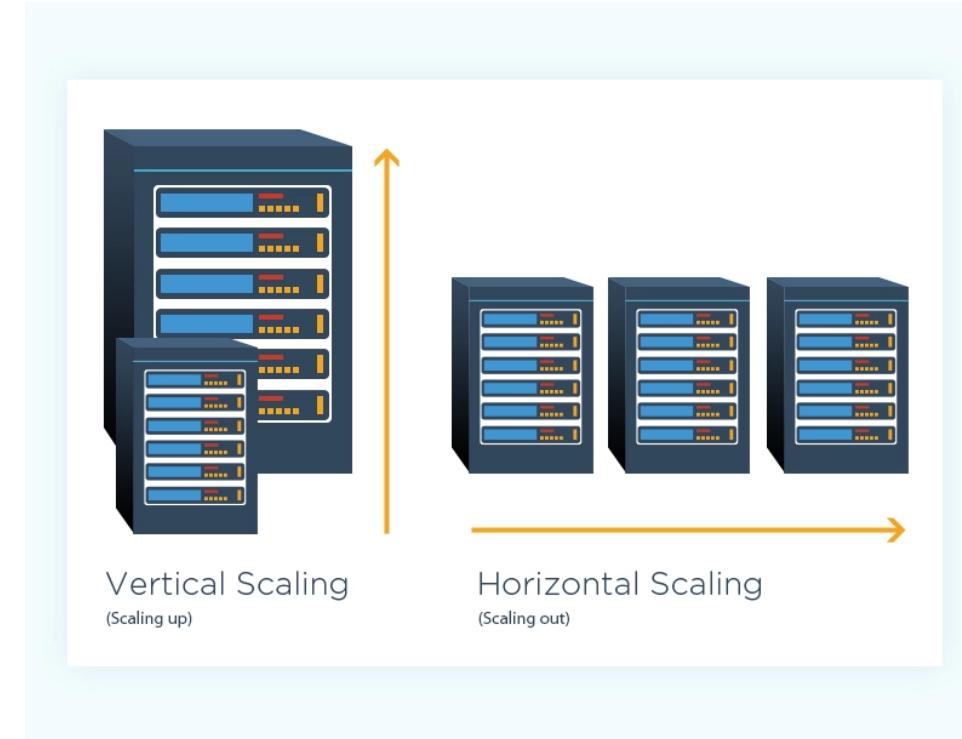
Tehnike koje omogućuju skalabilnost sustava

- Prikrivanje kašnjenja u komunikaciji
 - “radi nešto korisno dok čekaš odgovor” - asinkrona komunikacija
- Komponentno oblikovanje
 - više komponenti koji omogućuju funkcionalnost sustava (npr. raspodijeljena baza podataka, sustav imenovanja domena (DNS))
 - optimizacija implementacije pojedine komponente i cijelog sustava



Skalabilnost (3)

- Replikacija
 - replika = istovjetna kopija dijela sustava (funkcionalnosti) ili podatka
 - **horizontalno skaliranje, tzv. scale out** (radi povećanja propusnosti sustava)
 - problem: konzistentnost originala i kopije, upravljanje skupinom replika



povećanje propusnosti (broj obrađenih zahtjeva u sekundi)

Kvaliteta usluge

Kvaliteta usluge (engl. *Quality of Service, QoS*)

- skupni naziv za nefunkcijska obilježja sustava, od kojih su posebno važna sljedeća:
 - **vrijeme odziva** (engl. *response time*): vremenski period od slanja zahtjeva do primitka odgovora
 - **propusnost** (engl. *throughput*): mjeri promet na poslužitelju ili usluzi, a izražava se brojem zahtjeva u sekundi ili bit/s.
 - **raspoloživost** (engl. *availability*): vjerojatnost da je usluga dostupna u trenutku t i da generira odgovor na korisnički zahtjev.

Iskustvena kvaliteta (engl. *Quality of Experience, QoE*)

- mjeri korisnikovog zadovoljstva uslugom sustava

Oblikovanje raspodijeljenih sustava

Definiranje zahtjeva, potrebno odgovoriti na sljedeća pitanja

- Koje funkcijalne zahtjeve treba ostvariti – **ŠTO** sustav treba raditi?
- Kakve nefunkcijalne zahtjeve treba ostvariti – **KAKO** sustav treba raditi (kakva se kvaliteta usluge zahtijeva)?
- Temelji li se sustav na otvorenim rješenjima?
- Kakav je stupanj transparentnosti potreban i kako utječe na složenost, performance i troškove sustava?
- Kakva je skalabilnost sustava potrebna s motrišta veličine, rasprostranjenosti i upravljanja?

Sadržaj predavanja

- Definicija, obilježja i vrste raspodijeljenih sustava
- Zahtjevi na raspodijeljene sisteme: otvorenost, transparentnost, skalabilnost i kvaliteta usluge
- **Arhitektura raspodijeljenih sustava**
- Primjeri modela raspodijeljene obrade
- Studijski primjeri: raspodijeljeni sustav weba, Internet stvari

Arhitektura raspodijeljenih sustava

Programska arhitektura:

- logička organizacija sustava: programske komponente sustava, njihova organizacija i interakcija

(centralizirana arhitektura ili decentralizirana arhitektura)

Sustavska arhitektura (engl. *deployment*):

- smještaj programskih komponenti na raspoložive računalne resurse

Kako predočiti raspodijeljeni sustav?

Slojevita arhitektura

- u središtu pozornosti aplikacijski sloj (sloj primjene)
- aplikacijski programi i procesi te usluge koje im pružaju niži slojevi

Arhitektura temeljena na komponentama

- npr. mikrousluga: komponenta sustava s dobro definiranim sučeljem
- mehanizam komunikacije, usklađivanja i suradnje mikroservisa

Arhitektura temeljena na podacima

- procesi komuniciraju putem zajedničkog, dijeljenog (raspodijeljenog) repozitorija

Arhitektura temeljena na događajima

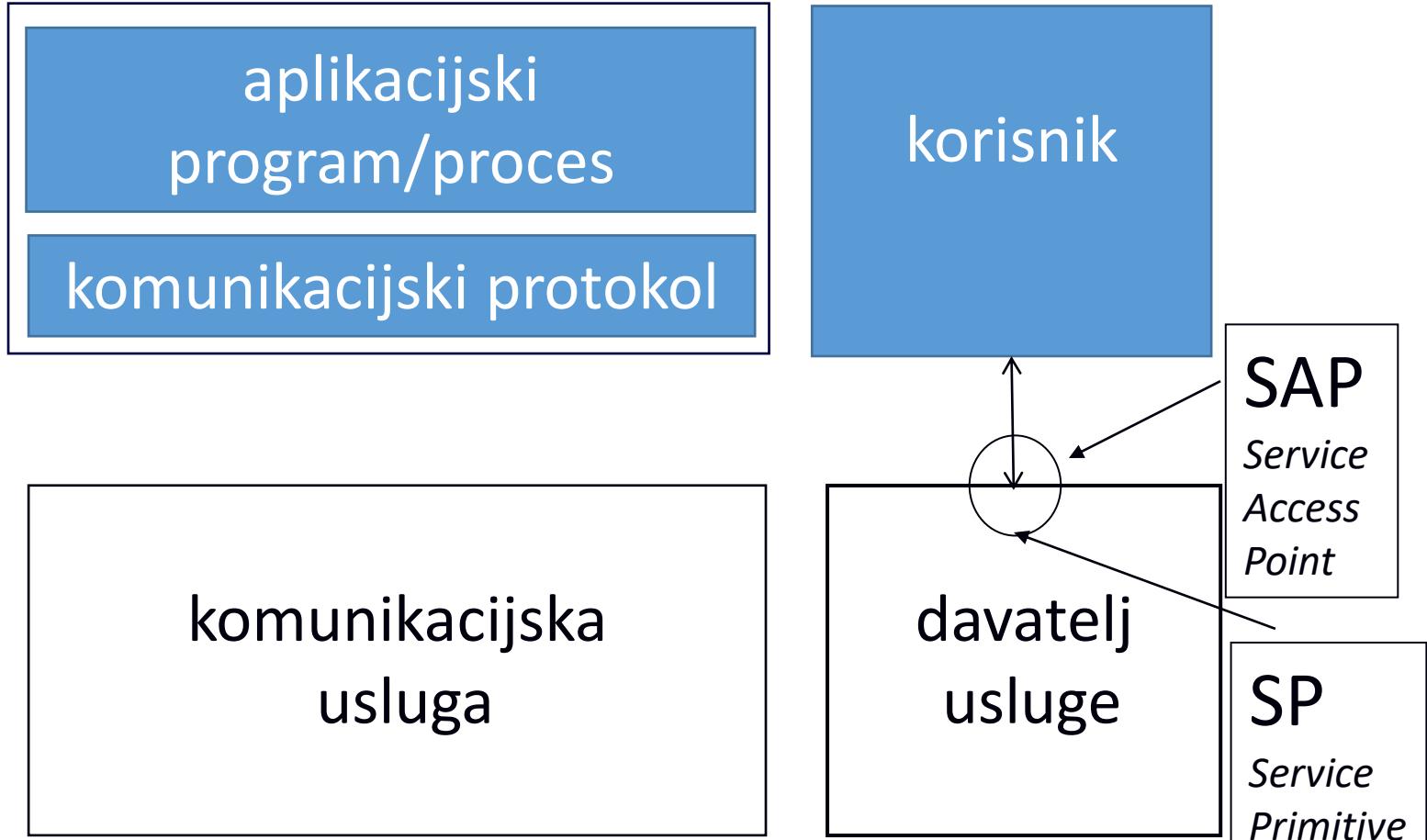
- procesi komuniciraju razmjenom tzv. događaja koji prenose informacije

Slojevita arhitektura

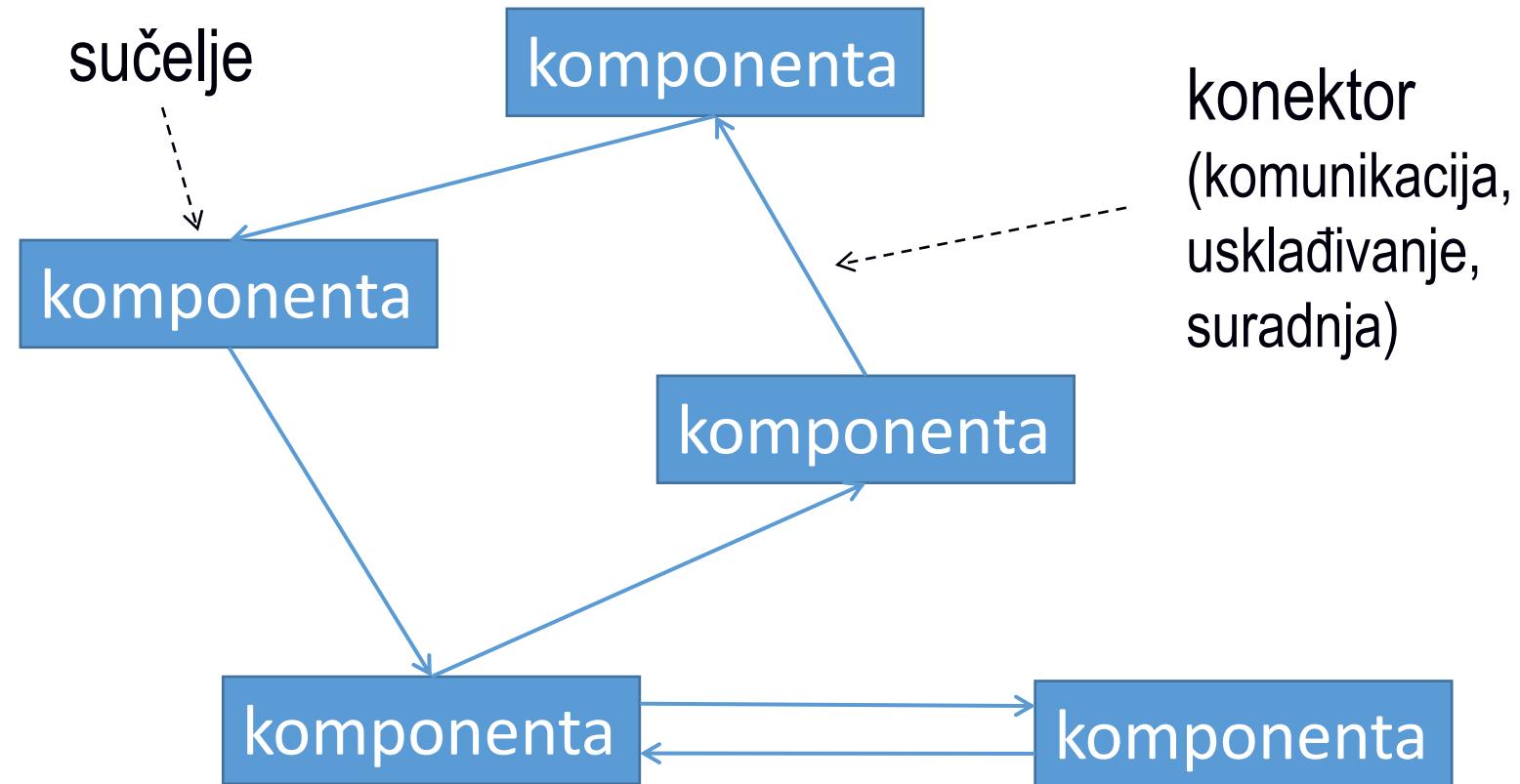
npr.
www

HTTP

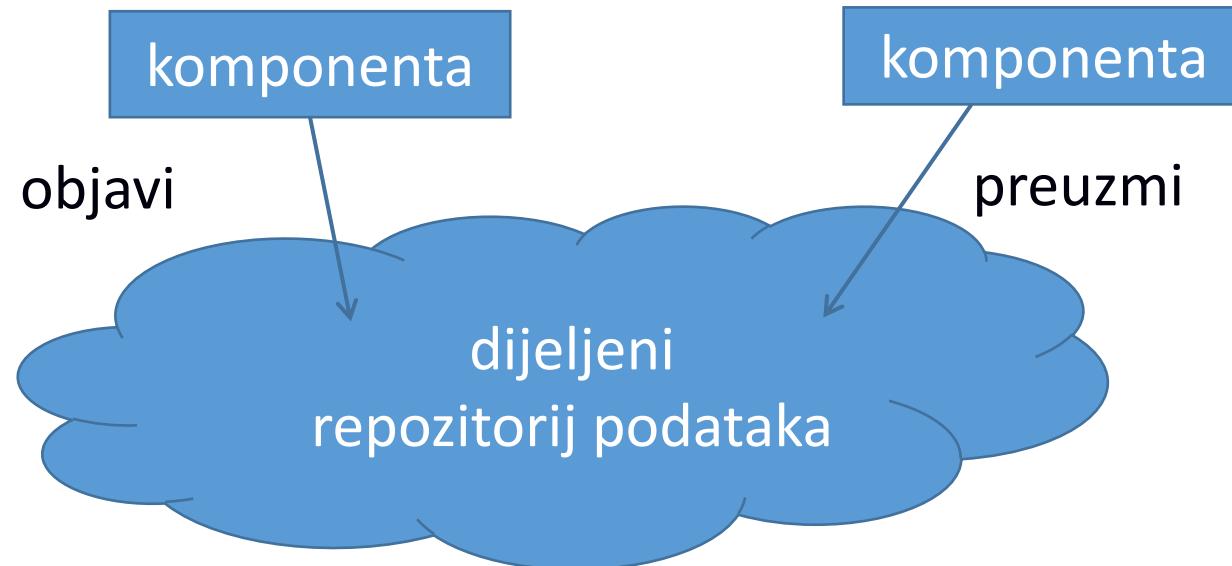
TCP
IP
LAN



Arhitektura temeljena na komponentama

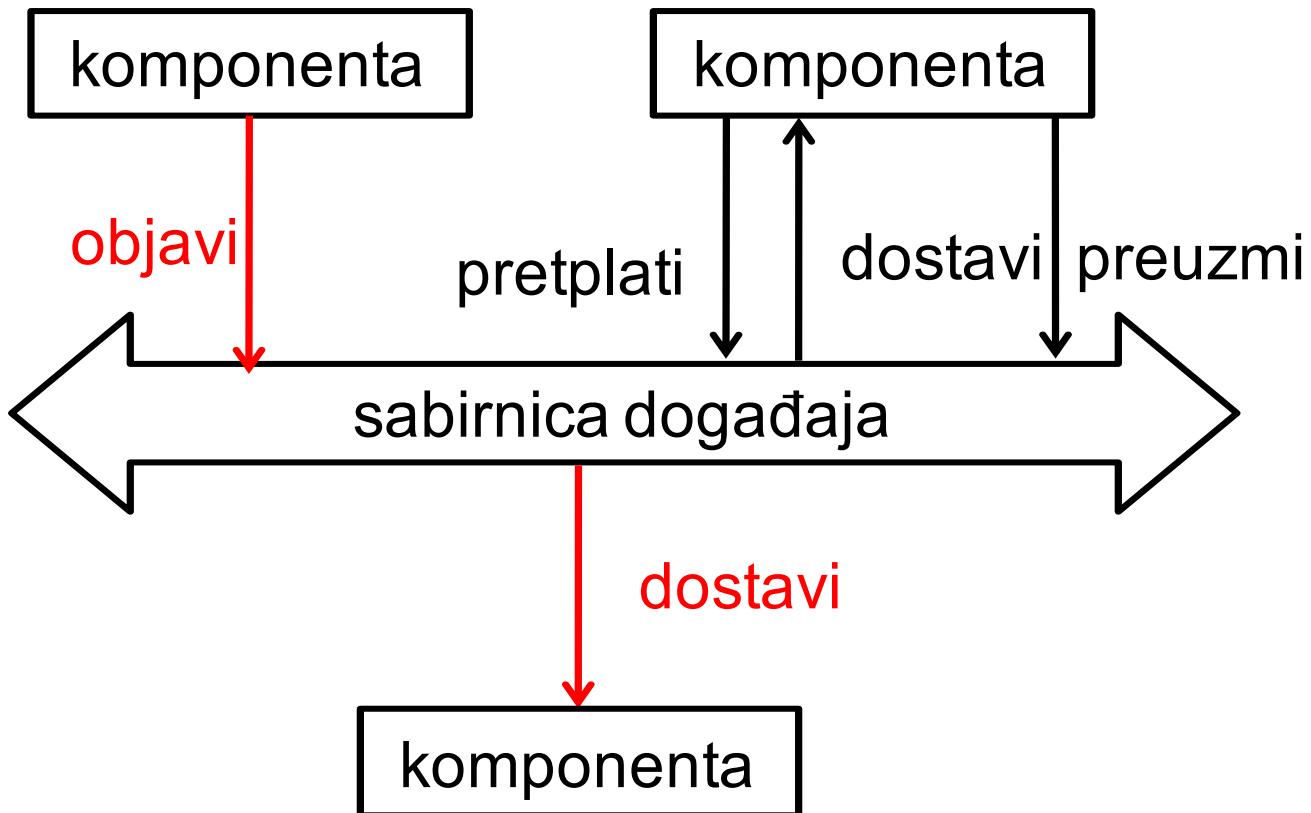


Arhitektura temeljena na podacima



Komponenta sustava upisuje (objavljuje) podatak u dijeljeni repozitorij koji omogućuje čitanje (preuzimanje) podatka drugim komponentama.

Arhitektura temeljena na događajima



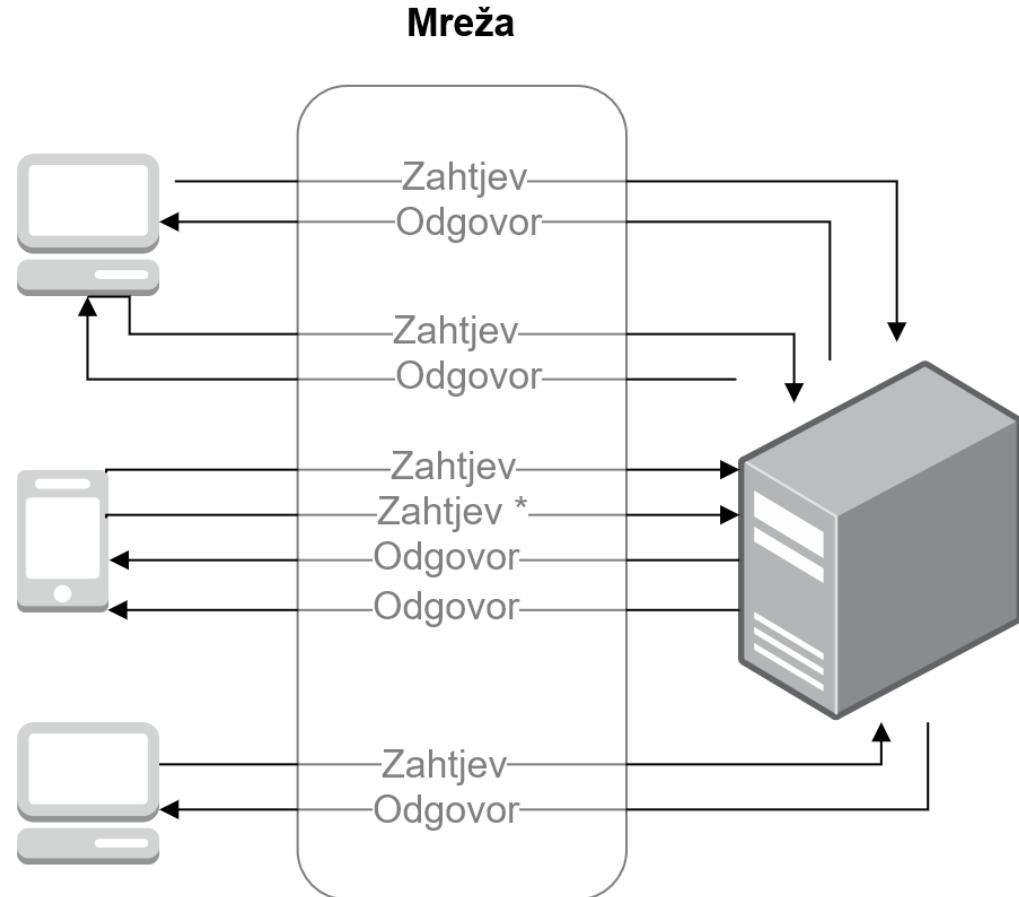
Komponenta se može preplatiti na podatak koji joj je potreban. Kad neka komponenta objavi podatak (događaj!), to će se dostaviti (događaj!) komponenti koja je na njega preplaćena.

Sadržaj predavanja

- Definicija, obilježja i vrste raspodijeljenih sustava
- Zahtjevi na raspodijeljene sisteme: otvorenost, transparentnost, skalabilnost i kvaliteta usluge
- Arhitektura raspodijeljenih sustava
- **Primjeri modela raspodijeljene obrade**
- Studijski primjeri: raspodijeljeni sustav weba, Internet stvari

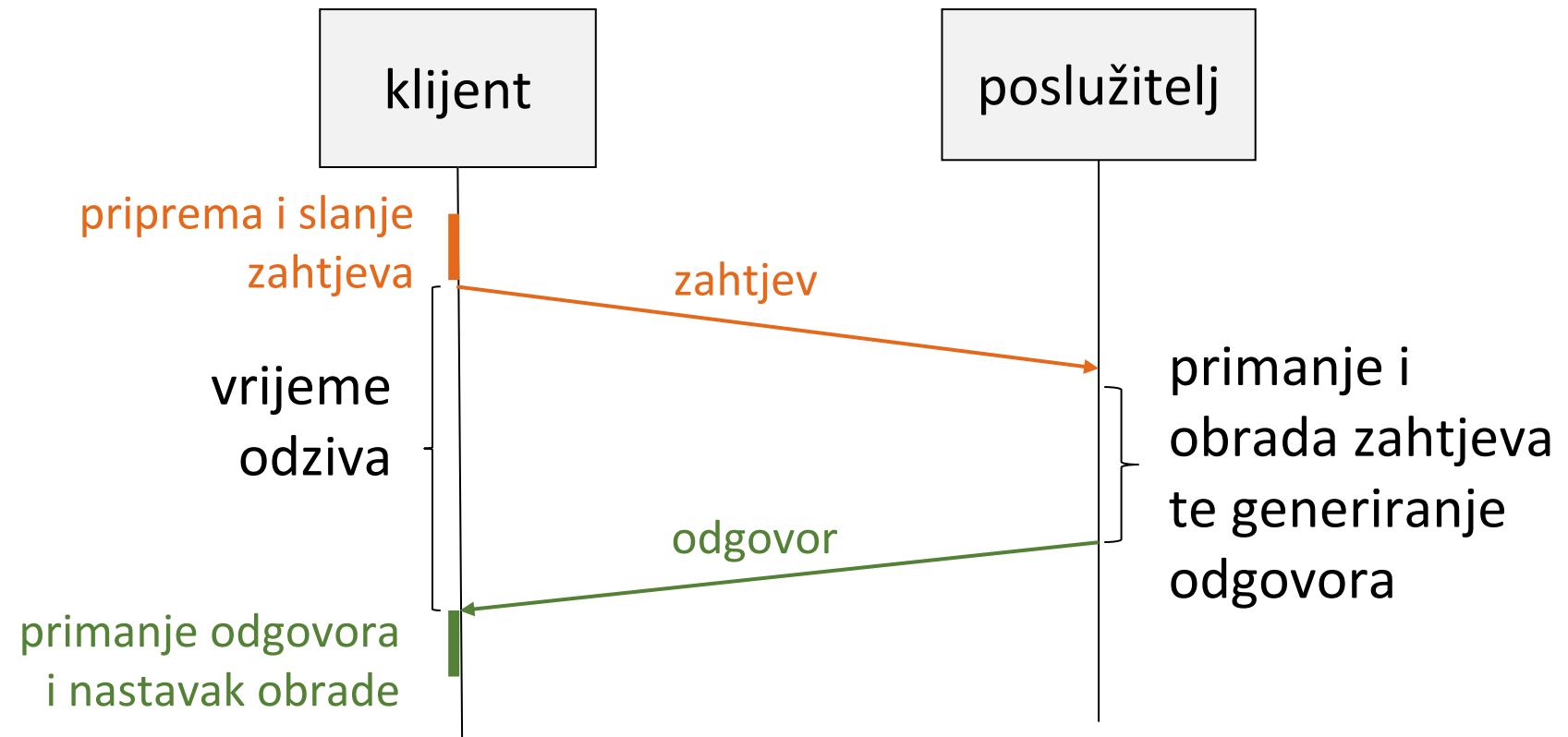
Model klijent – poslužitelj (1)

- Klijent: traži uslugu (zahtjev)
- Poslužitelj: pruža uslugu (odgovor) za više/mnogo klijenta



Model klijent – poslužitelj (2)

- Klijent šalje zahtjev i čeka odgovor
- Poslužitelj: prihvata i obrađuje zahtjev te vraća odgovor

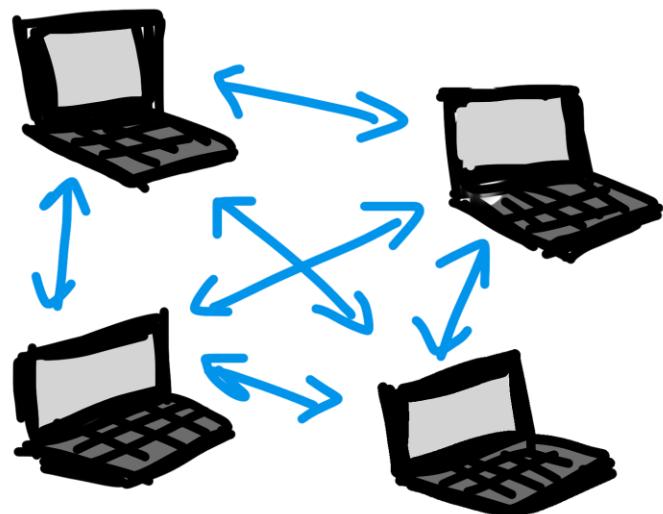


Model ravnopravnih sudionika (1)

- ravnopravni sudionik (engl. *peer*) je onaj koji može obaviti i funkciju poslužitelja i funkciju klijenta
- ravnopravni sudionici međusobno komuniciraju (engl. *Peer-to-Peer, P2P*) tako da se na aplikacijskom sloju povezuju u “prekrivajuću mrežu” (engl. *overlay network*) nad stvarnom mrežnom topologijom
- svaki čvor “plaća” sudjelovanje u mreži nudeći dio vlastitih sredstava ostalim čvorovima
- model ravnopravnih sudionika potencijalno nudi neograničena sredstva u velikim mrežama s puno čvorova

Model ravnopravnih sudionika (2)

Peer-to-peer, P2P



Decentralizirani raspodijeljeni sustav

- nema centralizirane koordinacije među *peerovima*
- ne postoji jedna točka ispada

Samoorganizirajuća mreža čvorova

- *peerovi* su međusobno neovisni, ulaze i izlaze iz sustava po volji

Programski agenti i premještanje programa

- **Programski agent** (engl. *software agent*): program koji obavlja neki posao za svog korisnika ili vlasnika, a raspolaže svojstvima kao što su **inteligencija, samostalnost, reaktivnost, proaktivnost** itd.
- **Migracija programa** (engl. *code migration*): razmjena programa između umreženih čvorova:
 - migracija procesa s jednog računala na drugo zbog (proširenja) funkcionalnosti, (uravnoteženja) opterećenja, (uvođenja) konkurentnosti,
- **Pokretni agent** (engl. *mobile agent*): programski agent koji predočuje korisnika u mreži i za njega obavlja neki posao krećući se samostalno između čvorova u mreži;

Sadržaj predavanja

- Definicija, obilježja i vrste raspodijeljenih sustava
- Zahtjevi na raspodijeljene sustave: otvorenost, transparentnost, skalabilnost i kvaliteta usluge
- Arhitektura raspodijeljenih sustava
- Primjeri modela raspodijeljene obrade
- **Studijski primjer: raspodijeljeni sustav weba**

Osnovne postavke weba

- Internetska aplikacija:
 - komunikacijski protokol aplikacijskog sloja - HTTP
 - jezik za označavanje HTML
 - standardi: *World Wide Web Consortium* (www.w3.org)
- Model klijent – poslužitelj
- Otvoreni sustav s transparentnim pristupom i konkurencijskom transparentnosti
- Transformacija weba:
 - “korisnik čita sadržaj” → “korisnik stvara sadržaj” (Web 2.0)
 - “korisnik odabire sadržaj” → “korisnik traži uslugu”: web-usluga (engl. *Web Service*)



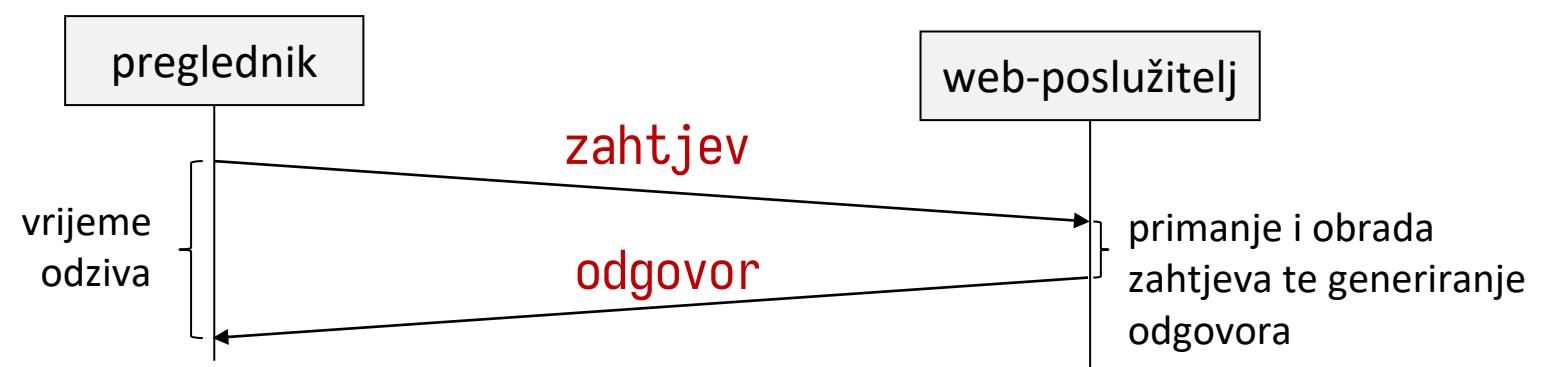
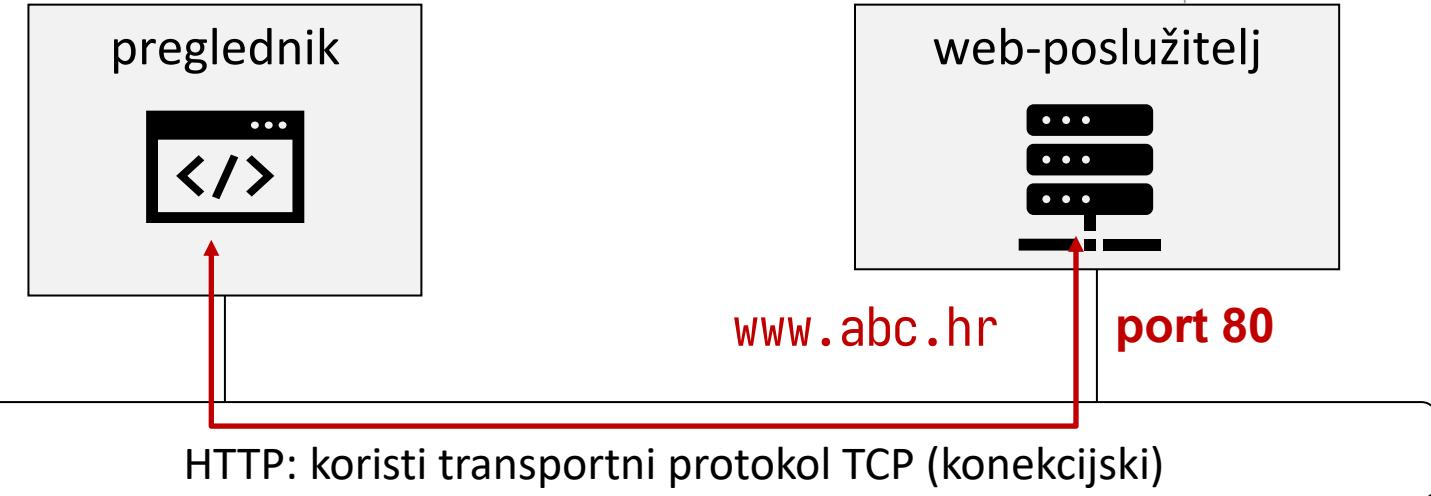
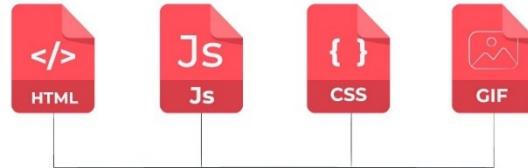
Hypertext Transfer Protocol (HTTP)

HTTP je **protokol** na aplikacijskom sloju

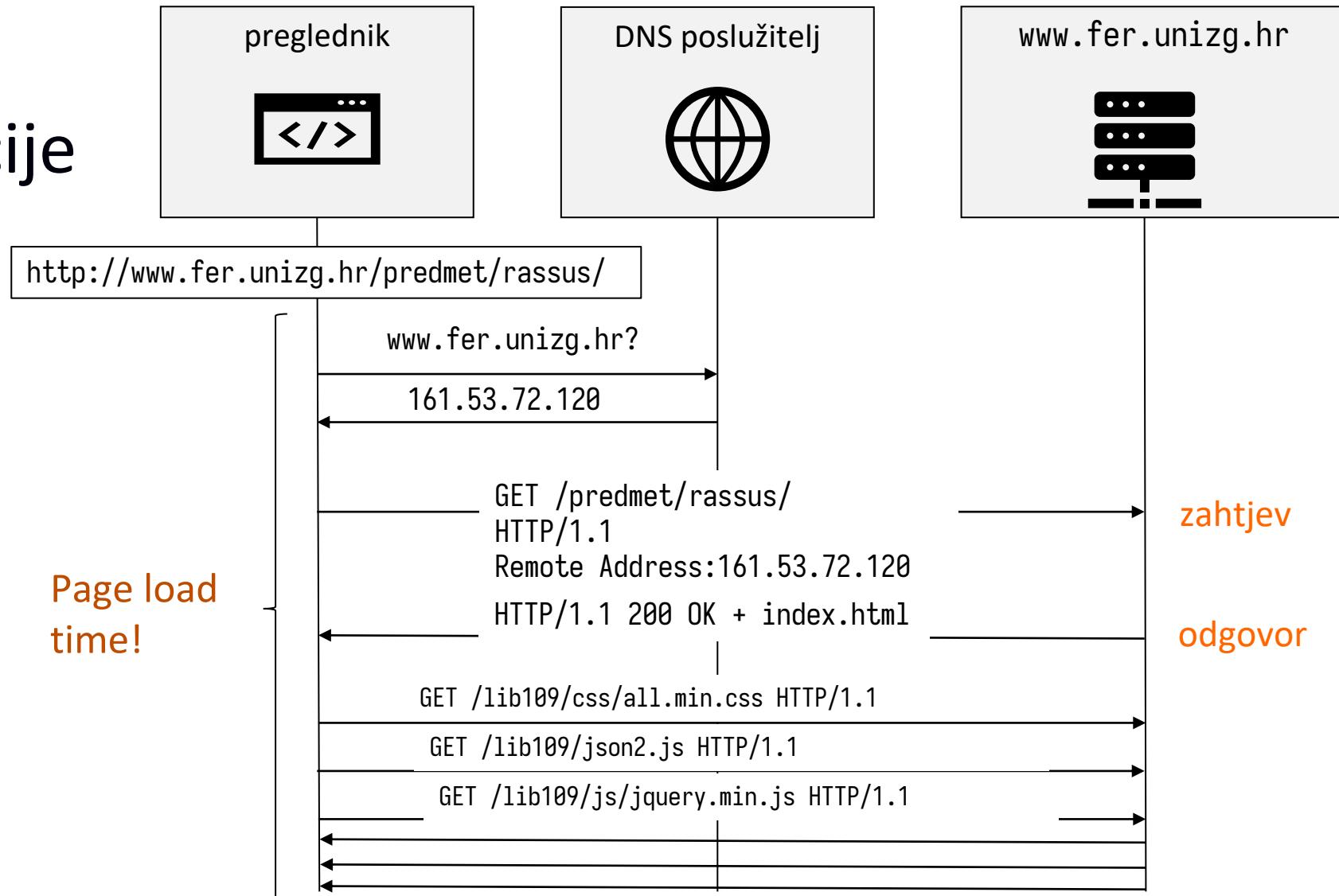
- protokol definira format i sadržaj poruka (zahtjeva i odgovora) te očekivano „ponašanje” poslužitelja, tj. pravila za generiranje odgovora na primljeni zahtjev
- **Zahtjev**: definira operaciju (tzv. metodu), oznaku resursa, verziju protokola, itd.
- **Odgovor**: rezultat (uspjeh, neuspjeh, pogreška,..) opisan statusnim kôdom i sadržaj resursa (npr. datoteka HTML, CSS, JPEG...)
- Poslužitelj ne čuva stanje između dviju konverzacija s istim klijentom jer je HTTP **protokol bez očuvanja stanja (engl. *stateless protocol*)**

Podsjetimo se kako web izgleda ...

resursi smješteni na poslužitelju



Primjer komunikacije



Transparentnost web-poslužitelja (1)

Lokacijska transparentnost:

- postiže se simboličkim imenima koja se u sustavu imenovanja domena (DNS) prevode u lokacije poslužitelja (mrežne adrese):
 - korisnik rabi simbolička imena, položaj poslužitelja (sjedišta weba) kao i bilo kojeg resursa ne treba biti i nije poznat korisniku

Migracijska transparentnost:

- ne mijenja se simboličko ime, već se samo mijenja lokacija poslužitelja (mrežna adresa) u DNS-u

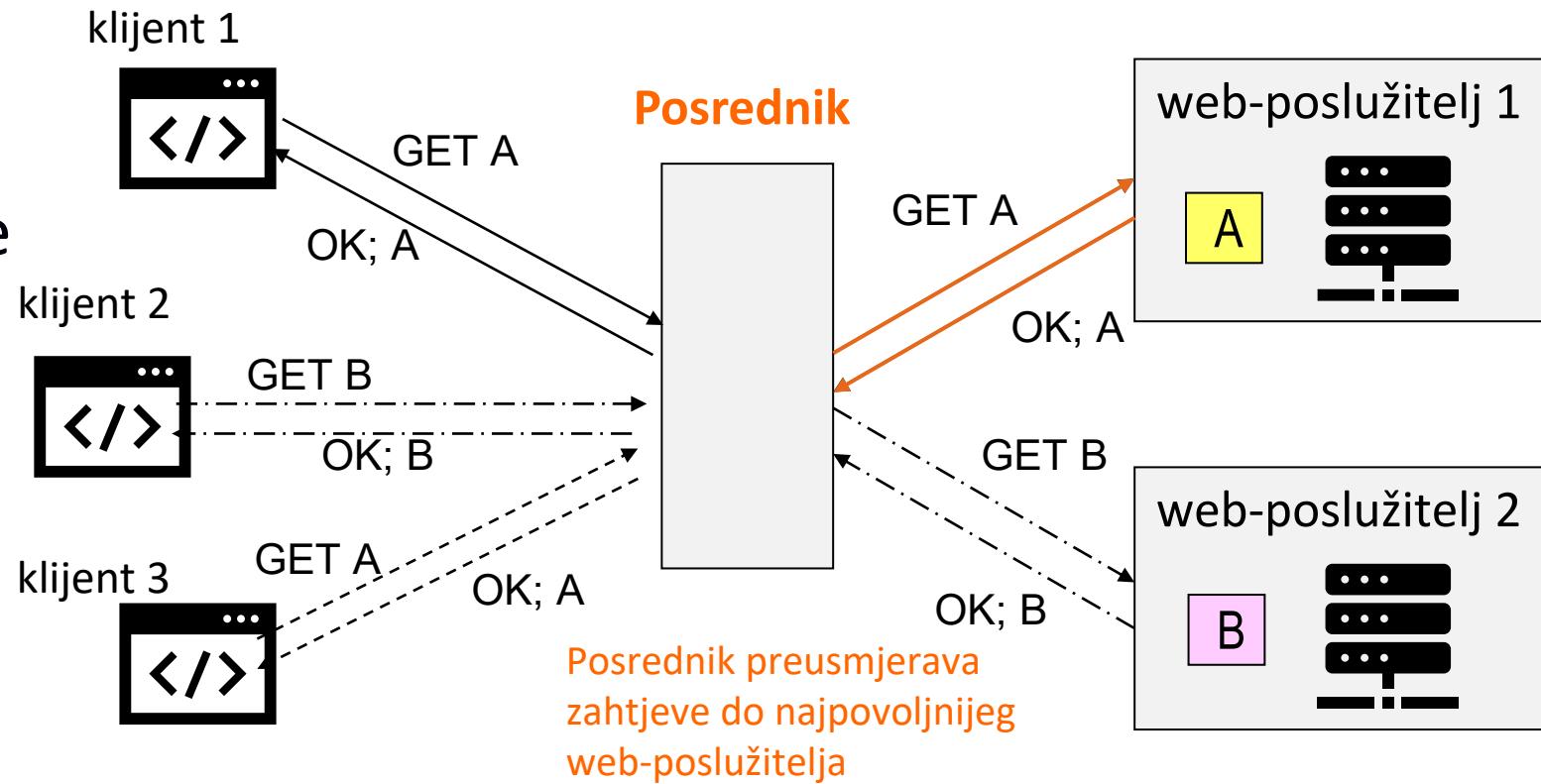
Relokacijska transparentnost:

- ne zahtijeva se, poslužitelj je stacionaran i ne kreće se tijekom pružanja usluge

Transparentnost web-poslužitelja (2)

Replikacijska transparentnost 1:

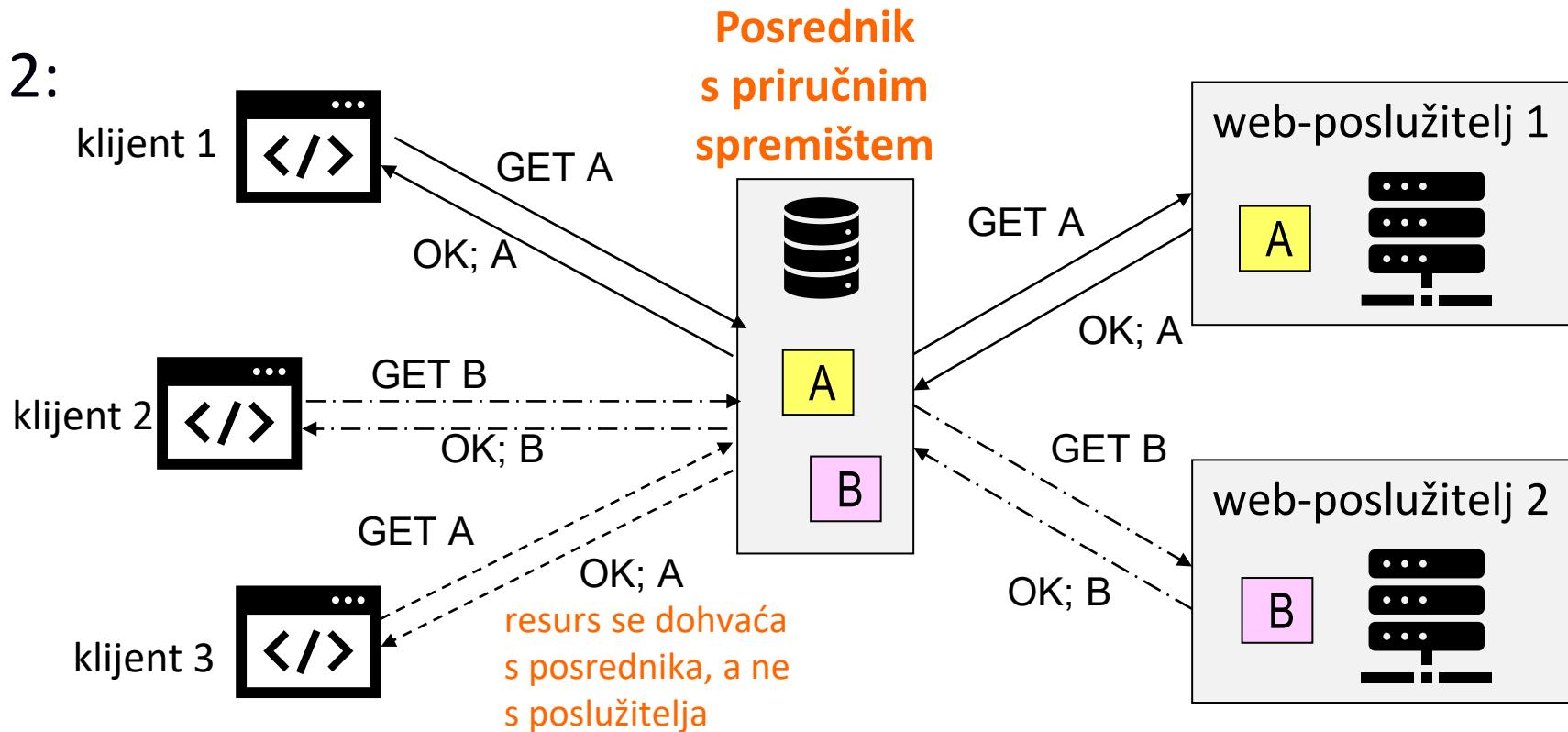
- Poslužiteljima se dostupa posredstvom posrednika (engl. *proxy*)



Transparentnost web-poslužitelja (3)

Replikacijska
transparentnost 2:

- Uvođenje
priručnog
spremišta



Sadržaj predavanja

- Definicija, obilježja i vrste raspodijeljenih sustava
- Zahtjevi na raspodijeljene sisteme: otvorenost, transparentnost, skalabilnost i kvaliteta usluge
- Arhitektura raspodijeljenih sustava
- Primjeri modela raspodijeljene obrade
- **Studijski primjer: Internet stvari**

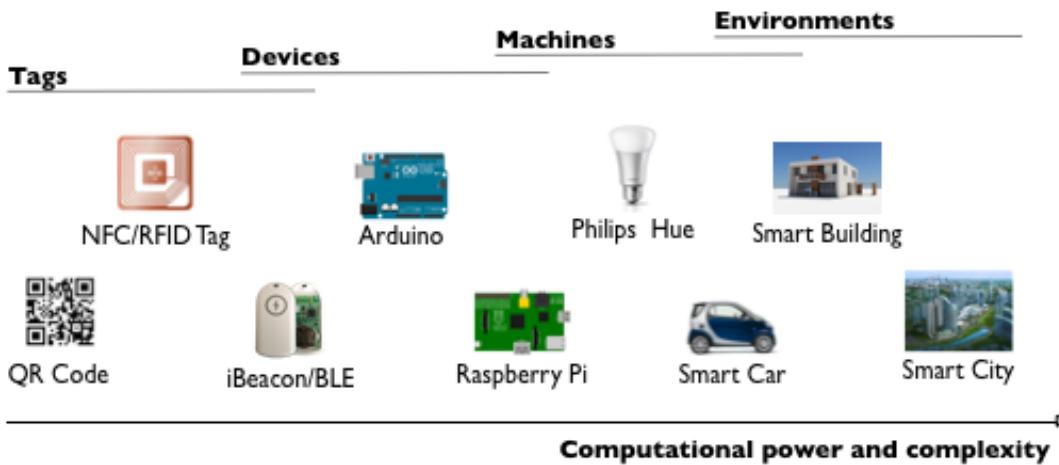
Uredaji i "stvari" postaju dio Interneta

Internet Connected
Object (ICO)



Što je "stvar"?

- Objekt iz fizičkog svijeta ili virtualnog digitalnog svijeta (virtualni objekt)
 - ima jedinstveni identifikator i povezan je na Internet (direktno ili putem posrednika), *Internet Connected Object* (ICO)
 - senzor: opažanje okoline, potencijalno kontinuirano generira podatke
 - aktuator: može izvršiti određene funkcije



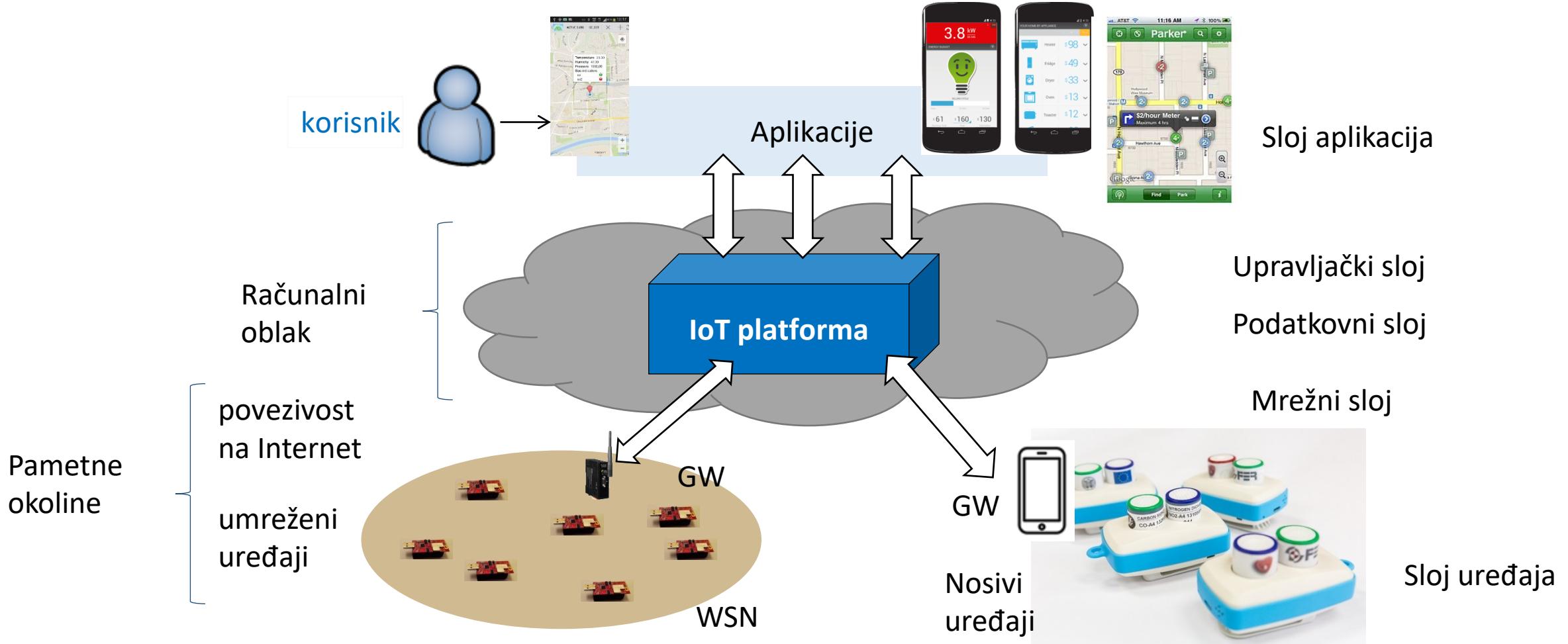
Source: Building the Web of Things: book.webofthings.io
Creative Commons Attribution 4.0

Internet of Things (IoT): definicija

ITU-T Recommendation Y.2060, 06/2012:

- *A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) Things based on, existing and evolving, interoperable information and communication technologies.*
 - *Through the exploitation of identification, data capture, processing and communication capabilities, the IoT makes full use of things to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled.*
 - *In a broad perspective, the IoT can be perceived as a vision with technological and societal implications.*

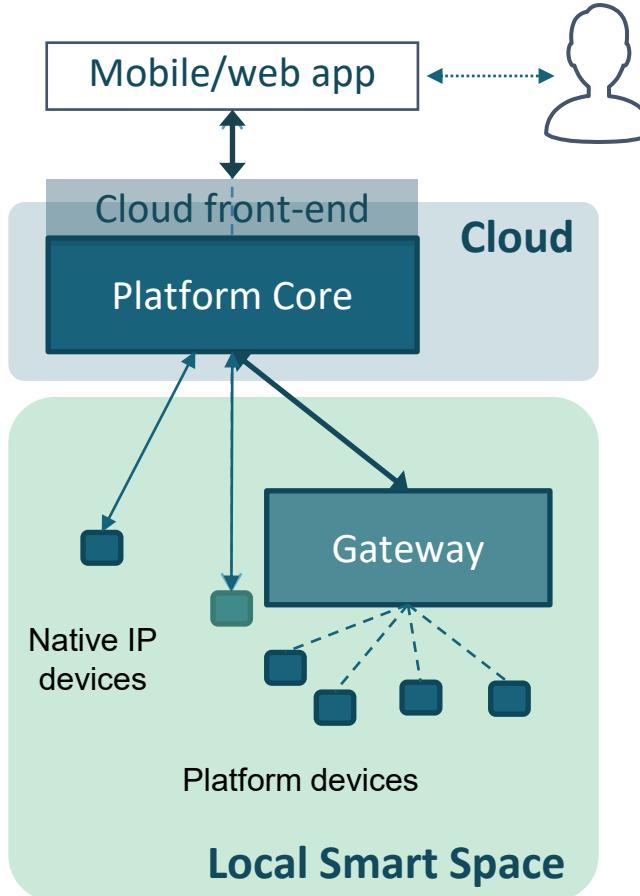
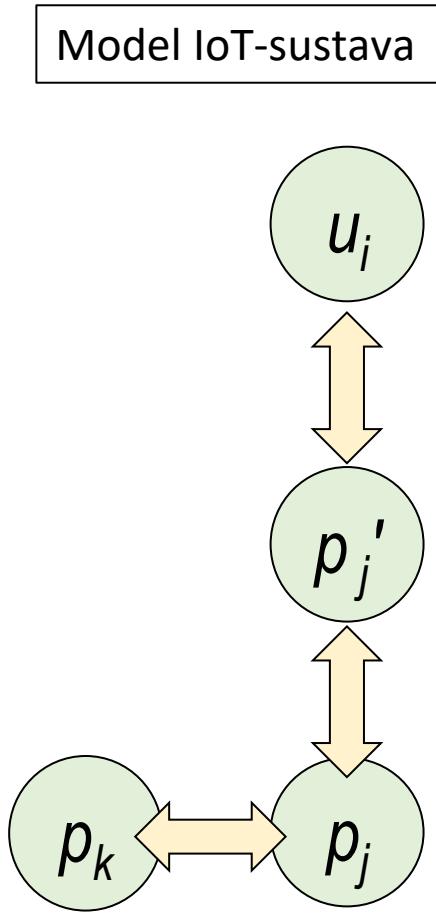
Pojednostavljena arhitektura Interneta stvari



Kako integrirati "stvari" i ponuditi aplikacije korisnicima?

- Pomoću programskih platformi (**IoT-platforma**) koje integriraju i upravljaju uređajima
 - raspodijeljeni sustav velikih razmjera
 - uređaji: često imaju vrlo ograničene resurse te su povezani na Internet putem prilaznog uređaja (engl. *gateway*)
 - potrebno je objediniti i na jedinstveni način zapisati podatke primljene iz različitih izvora
 - potreba za obradom velike količine podataka (često u stvarnom vremenu)
 - *Web of Things*: koncept koji povezuje uređaje direktno na WWW (tehnologije vezane uz protokol HTTP)

IoT-rješenje je složeni raspodijeljeni sustav



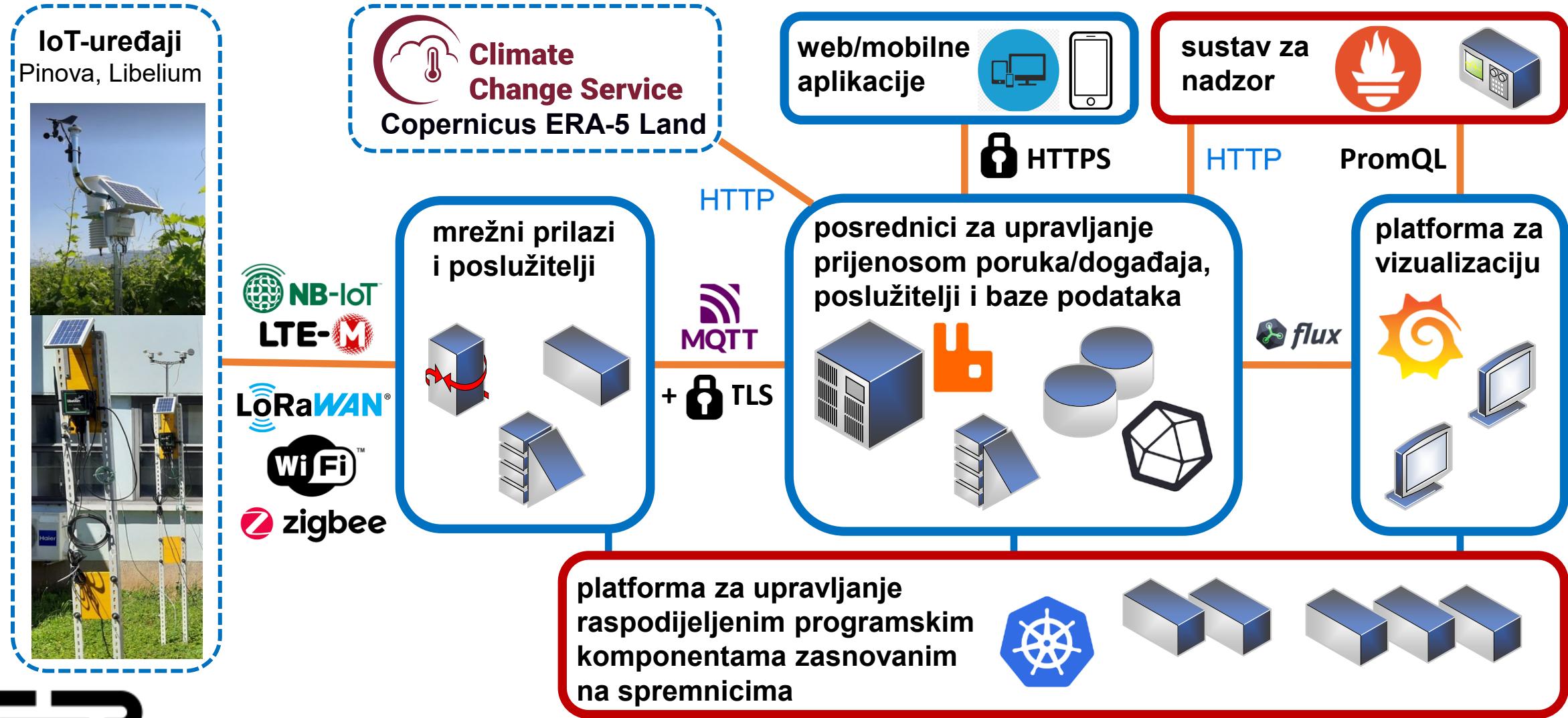
Virtualni entitet predstavlja stvarni uređaj

- programska platforma održava metapodatke o uređajima
- pohranjuje senzorska očitanja, stanja aktuatora, obrađuje podatke

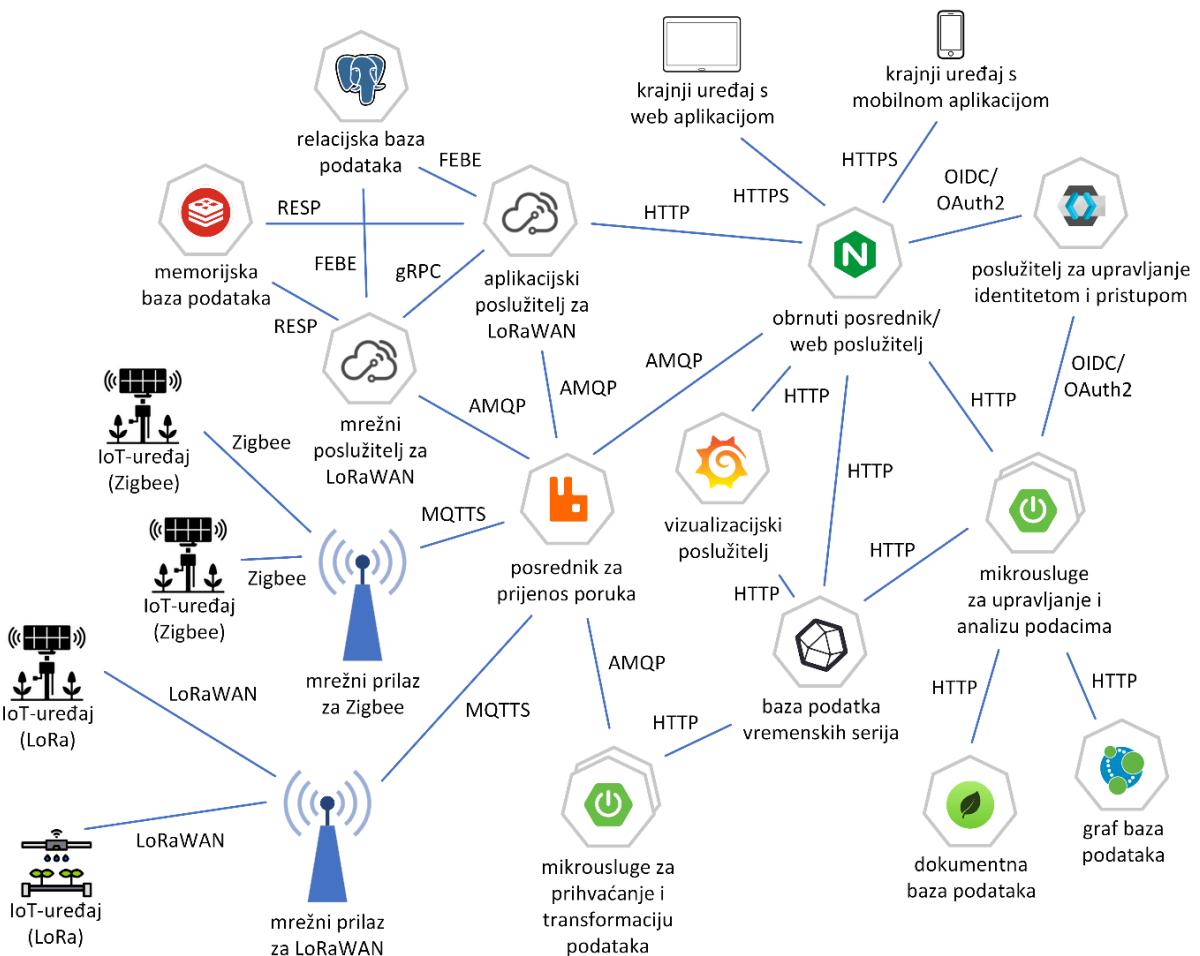
Interoperabilna IoT-platforma



<http://www.iot.fer.hr/>



Arhitektura IoT-platforme



Zadaci (1)

1. Čime je definirana otvorenost weba?
2. Kojim aspektima transparentnosti pridonosi sustav imenovanja domena (DNS)?
3. Kako replikacija pridonosi otpornosti na kvarove i skalabilnosti?
4. Kakvi bi se problemi pojavili kada bi se više repliciranih poslužitelja weba priključilo na mrežu izravno, a ne posredstvom zastupnika (*proxy*)?
5. Što sve utječe na vrijeme odziva poslužitelja weba?

Zadaci (2)

6. Na primjeru weba objasnite razliku između vertikalnog i horizontalnog skaliranja sustava.
7. Zašto se koriste perzistentne konekcije u HTTP-u?
8. Kako se definira uvjetni GET i kako funkcionira? Kakve prednosti donosi za performance HTTP-a?
9. Objasnite zašto današnja rješenja za IoT predstavljaju raspodijeljene sustave velikih razmjera?



SVEUČILIŠTE U ZAGREBU



**Diplomski studij
Računarstvo**

Znanost o mrežama
Programsko inženjerstvo i
informacijski sustavi
Računalno inženjerstvo
**Ostali (slobodni izborni
predmet)**

Raspodijeljeni sustavi

2. Procesi i komunikacija: model klijent-poslužitelj

Ak. god. 2022./2023.

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 daljnog 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.

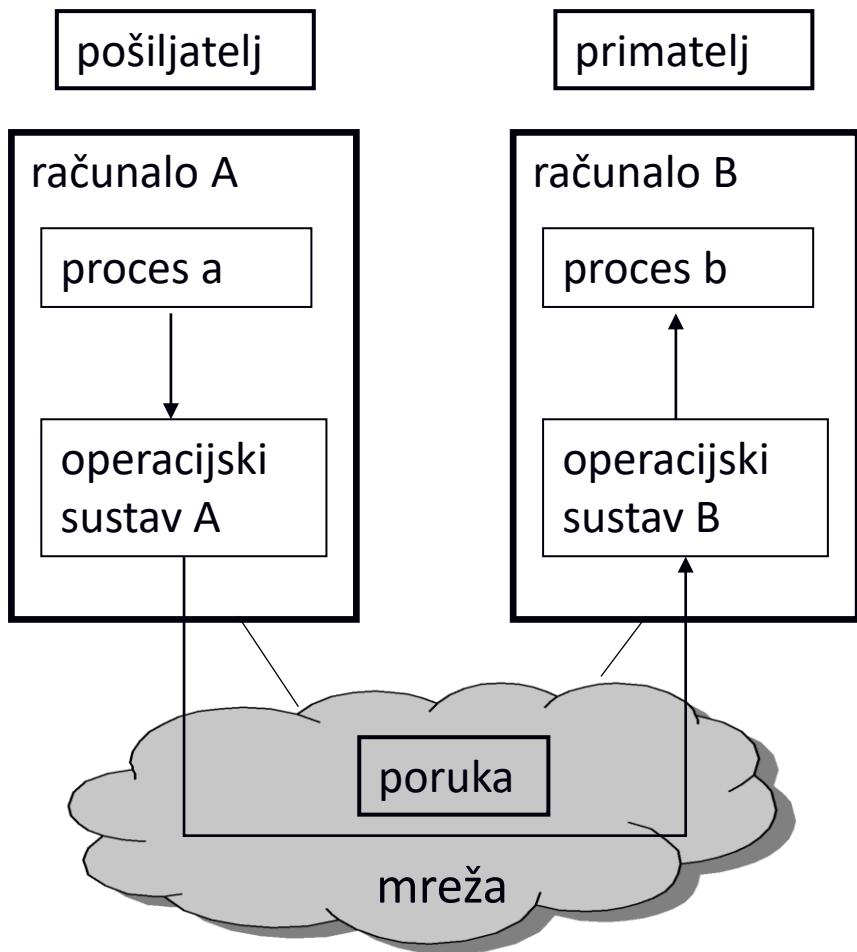
Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.

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*)
 - Java RMI
 - gRPC

Osnovni model komunikacije



- **Procesi**

- izvode se na različitim računalima, autonomni su

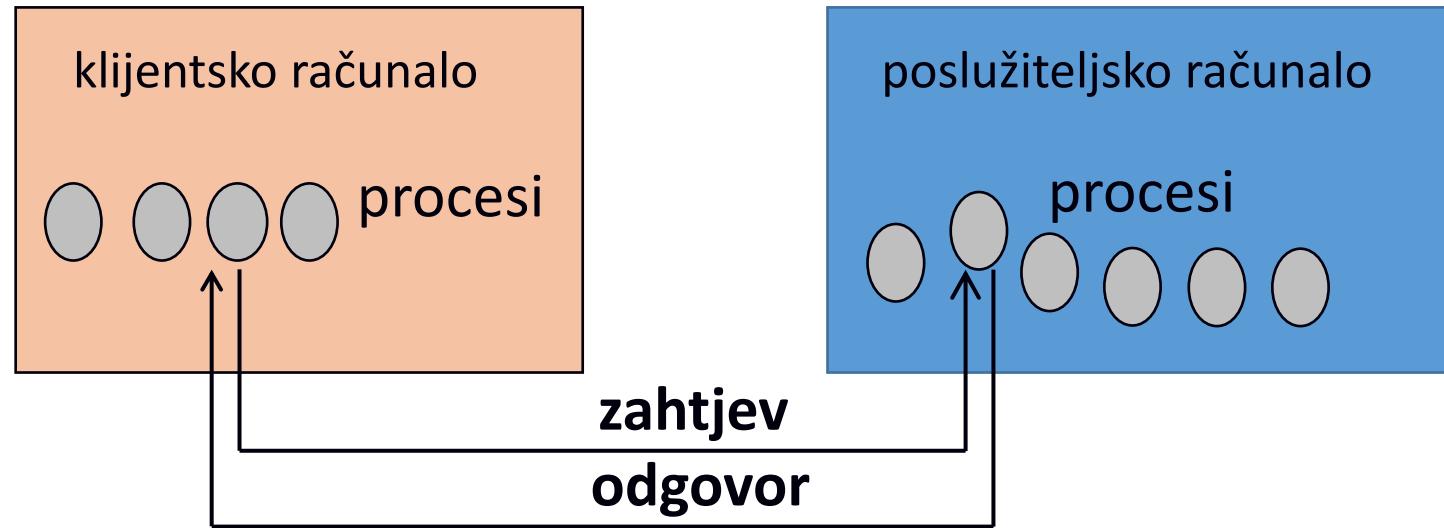
- **Komunikacija**

- proslijedivanje 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

- **koneksijska**
 - procesi eksplisitno kreiraju konekciju prije razmjene podataka, postoje kontrolne poruke za uspostavu konekcije
- **bezkoneksijska**
 - 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 pošiljatelj i primatelj poruke istovremeno dostupni

Obilježja komunikacije

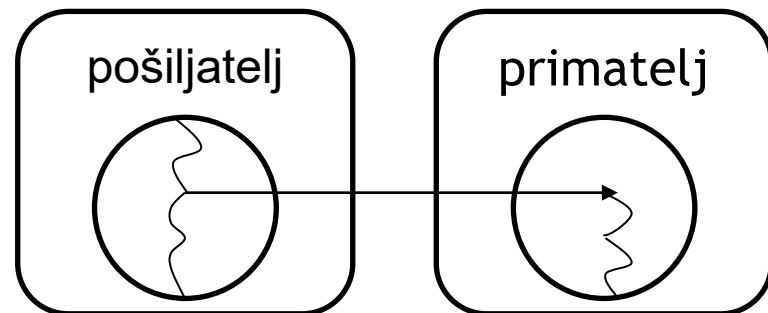
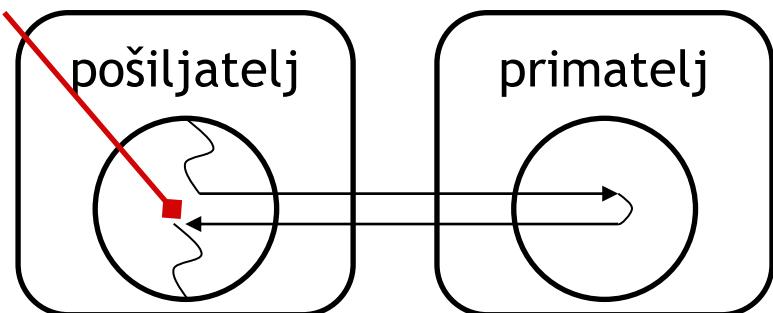
- **sinkrona komunikacija**

- blokira pošiljatelja do primitka potvrde od strane primatelja

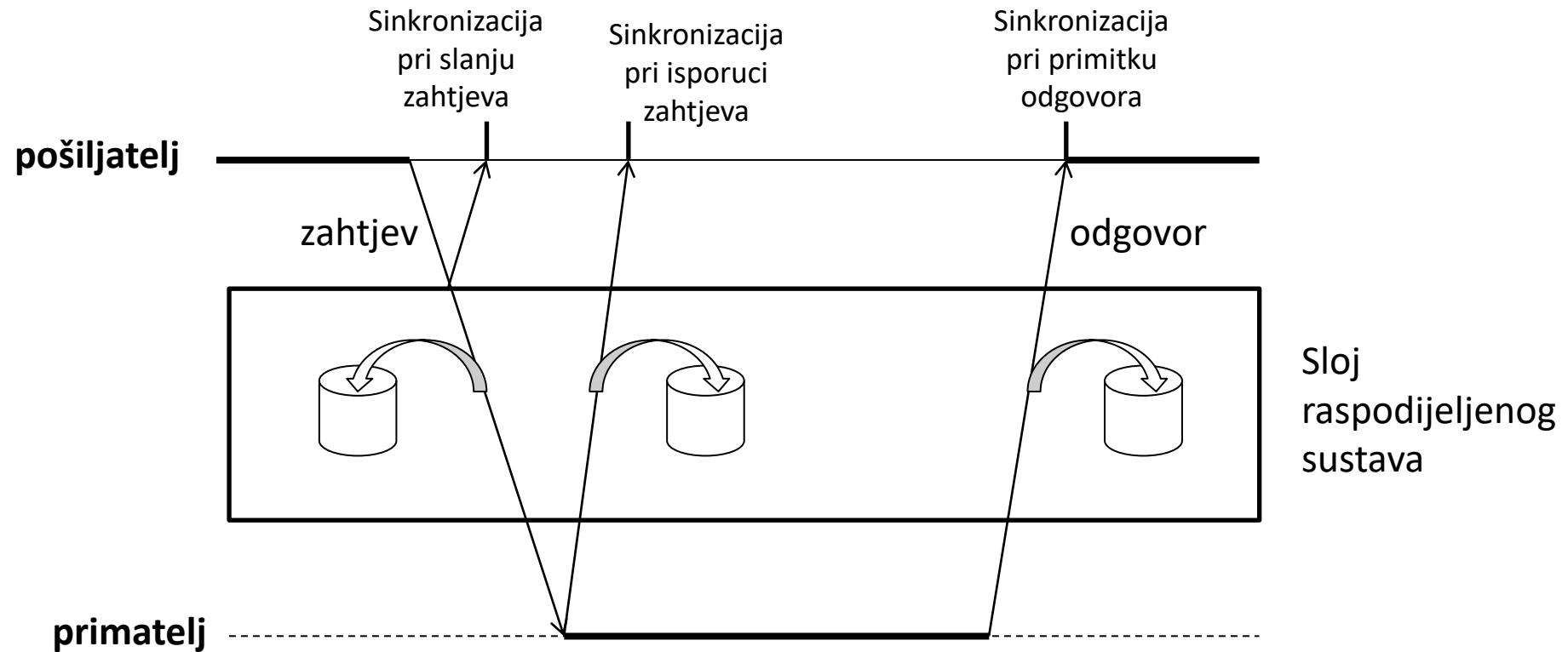
- **asinkrona komunikacija**

- omogućuje pošiljatelju nastavak obrade odmah nakon slanja poruke

blokiranje

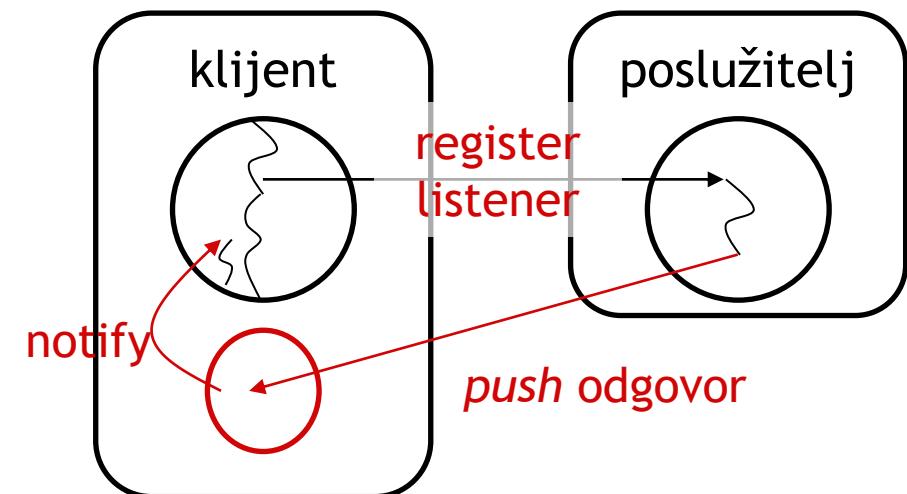
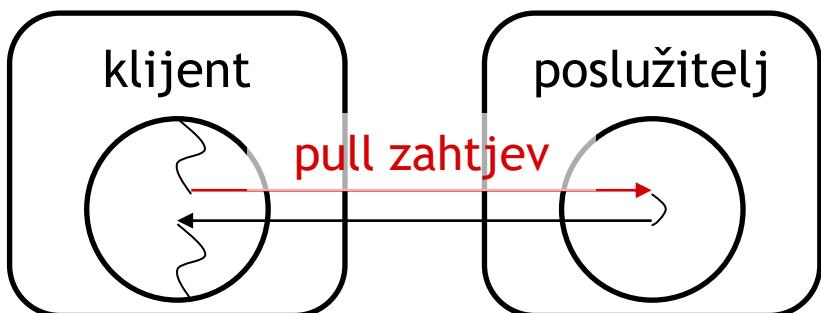


Točke sinkronizacije



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 obradivati korisničke zahtjeve
 - višedretveni klijent može nastaviti s obradom dok čeka odgovor poslužitelja
(primjer: Web preglednik)

Obilježja procesa

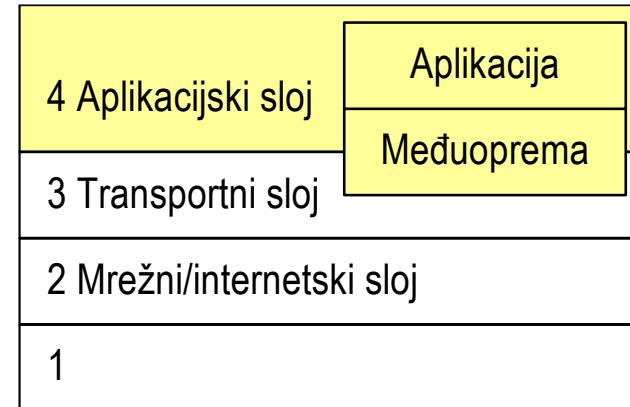
- **vremenska (ne)ovisnost**
 - vremenski ovisni procesi moraju biti istovremeno aktivni za realizaciju komunikacije
 - vremenski neovisni procesi mogu komunicirati i ako nisu istovremeno aktivni
- **ovisnost o referenci “sugovornika”**
 - proces je ovisan o referenci “sugovornika” ako mora znati jedinstveni 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*)
 - Java RMI
 - gRPC

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

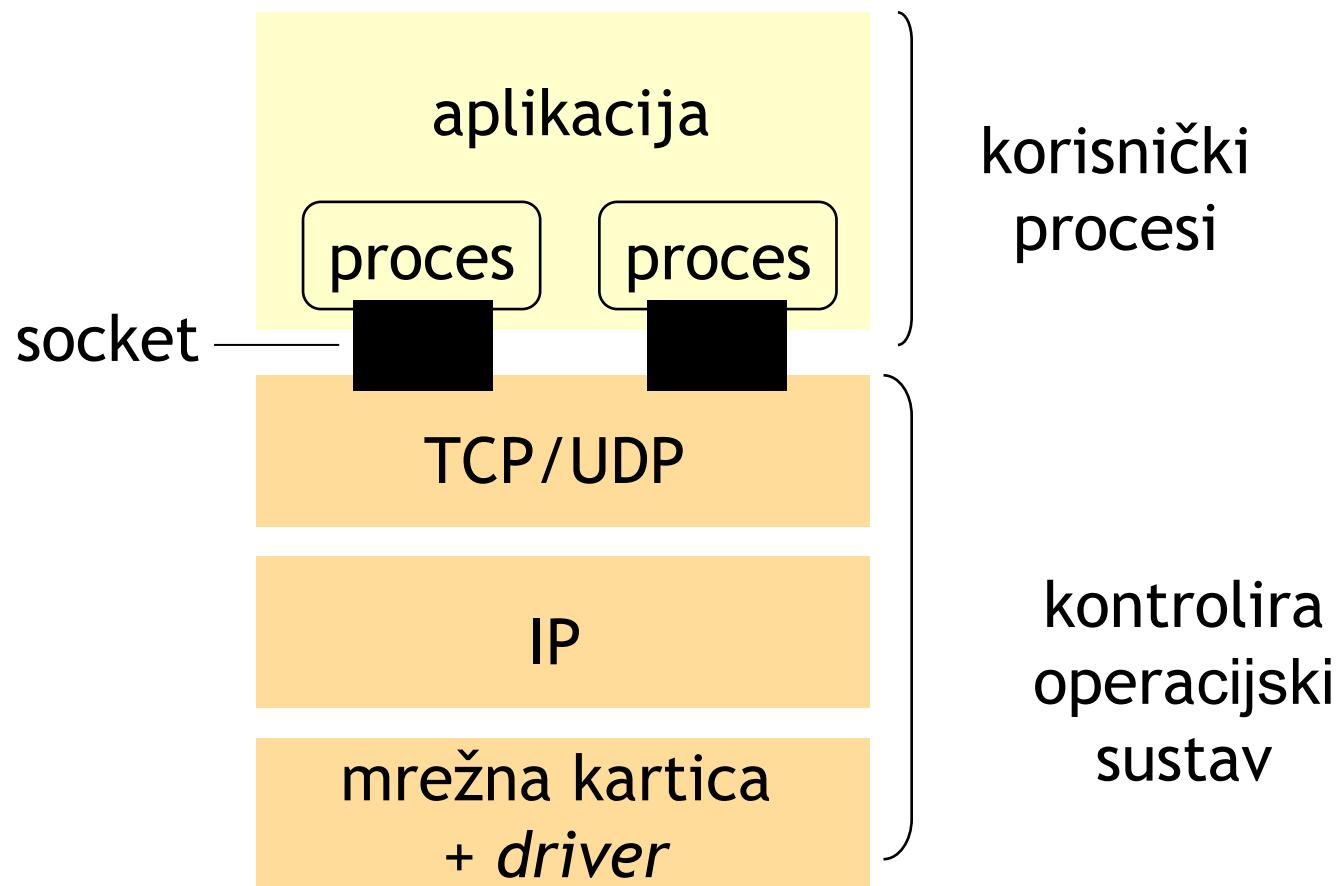
- Postojeća rješenja za komunikaciju raspodijeljenih procesa
 1. komunikacija korištenjem priključnica (*socket API*)
 2. poziv udaljene procedure (*remote procedure call, RPC*)
 3. raspodijeljeni objekti - poziv udaljene metode (*remote method invocation, RMI*)
 4. komunikacija razmjenom poruka (*message-oriented interaction*)
 5. model objavi-preplati (*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

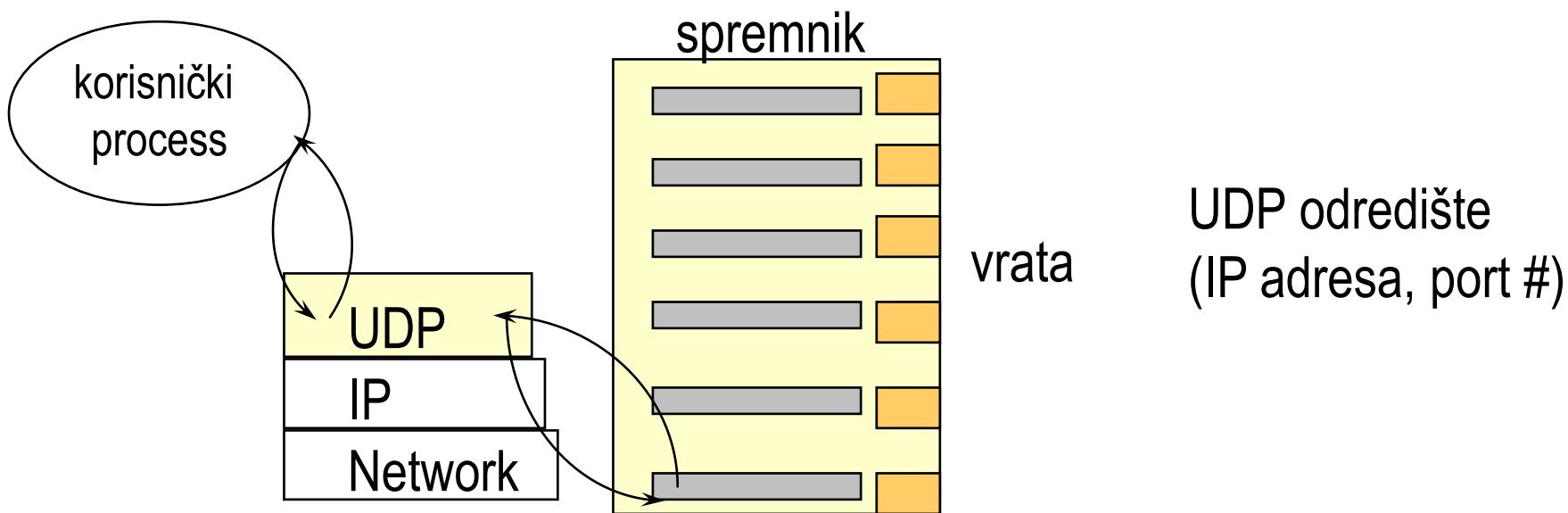
Komunikacija pomoću *socketa*



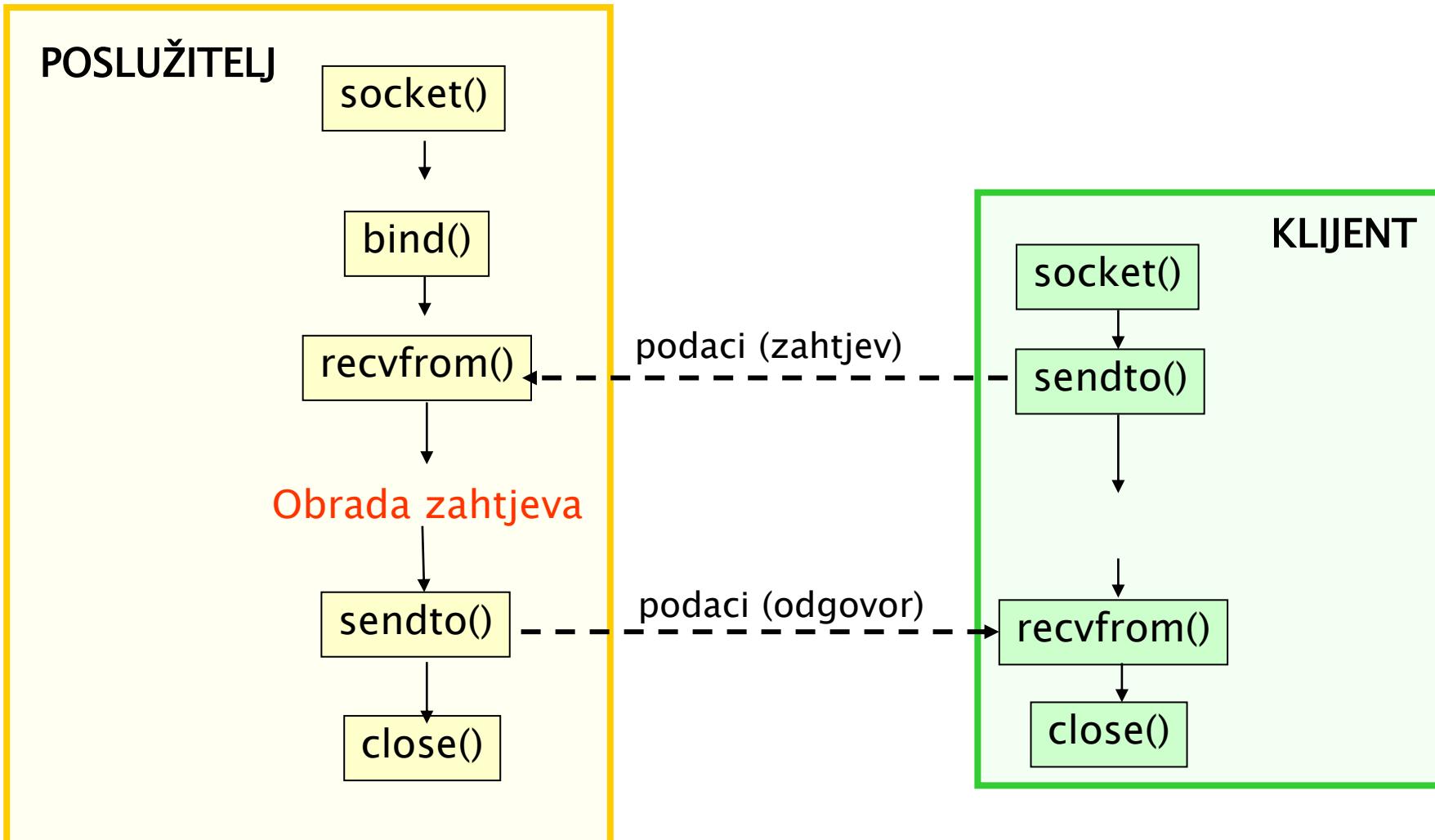
Transportni protokol UDP

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*



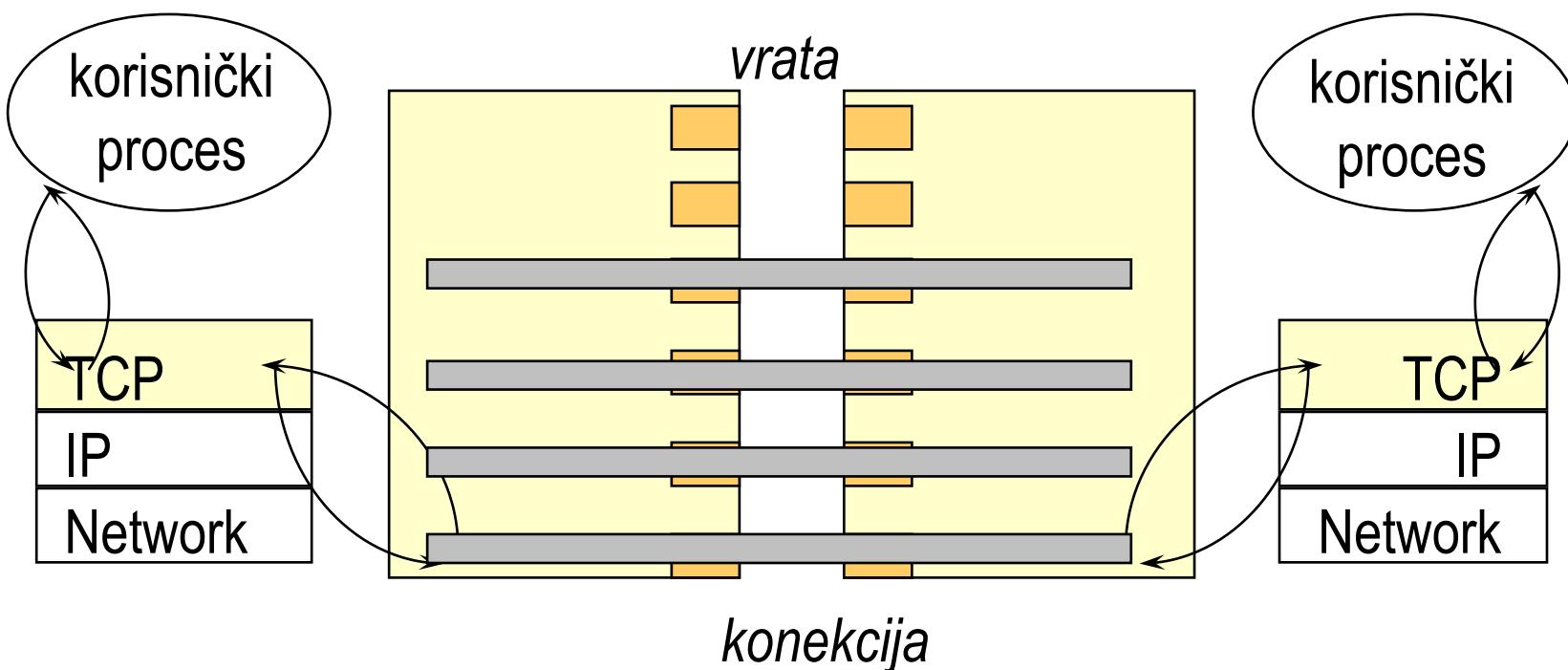
Obilježja socketa UDP

- 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*

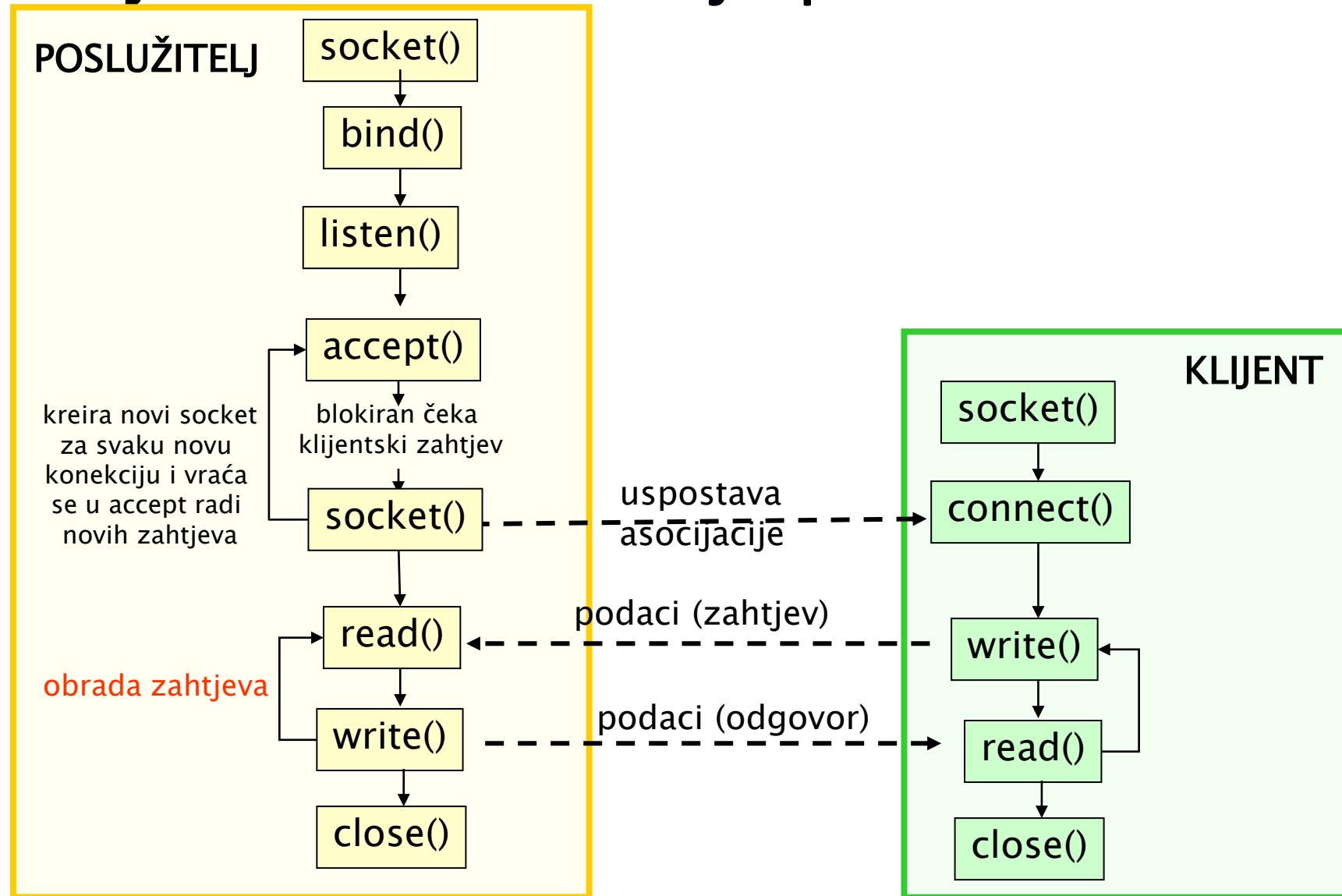
Transportni protocol TCP

Transmission Control Protocol (TCP)

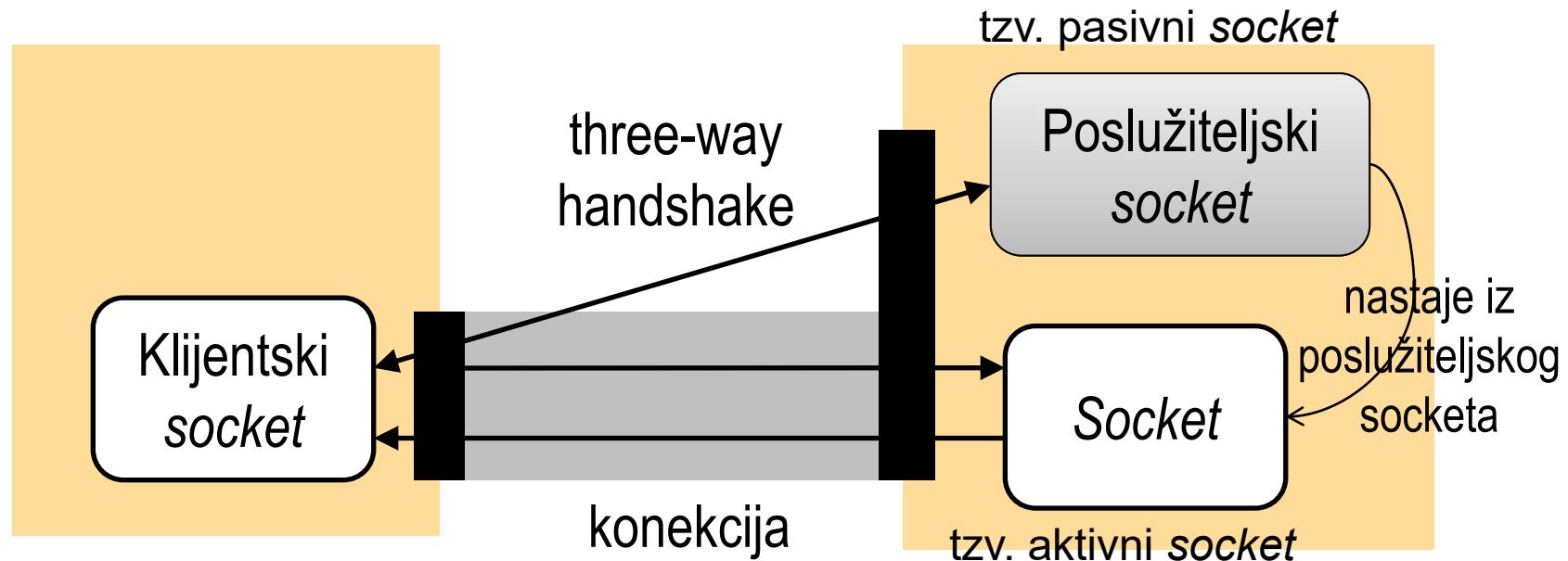
- konekcija između dvije krajnje točke koje se moraju dogovoriti o uspostavi konekcije



Konekcijska komunikacija pomoću socketa 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 socketa 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*)
 - Java RMI
 - gRPC

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)

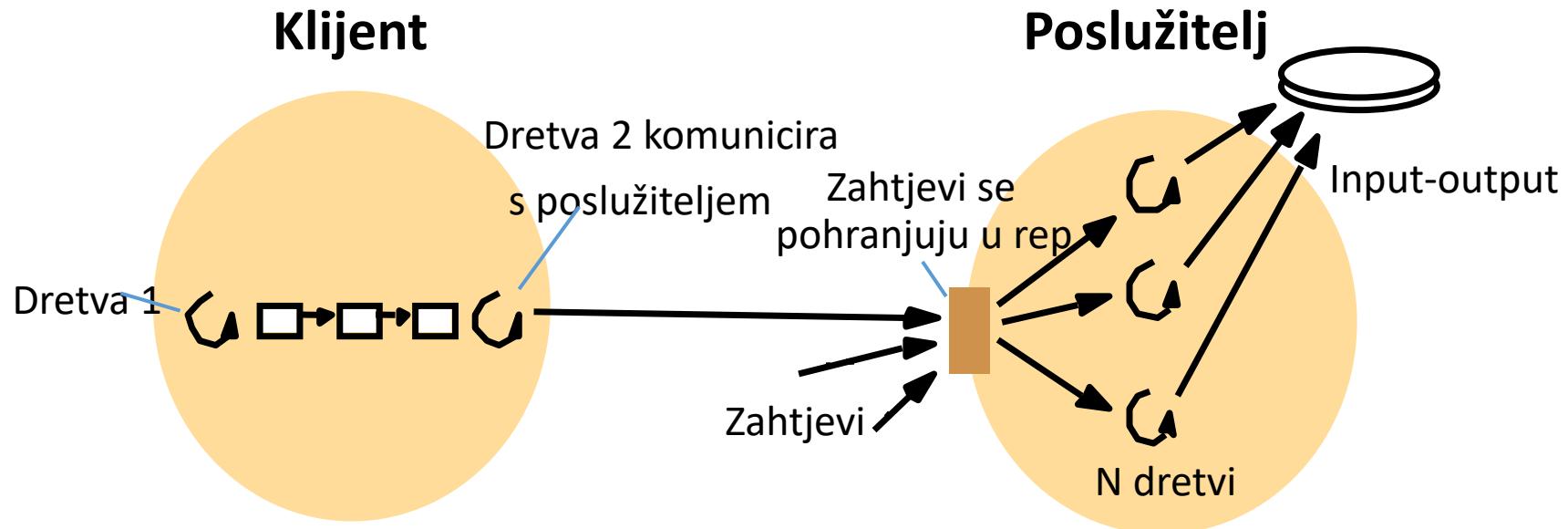
- Java Networking Tutorial

<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*)
 - Java RMI
 - gRPC

Višedretveni poslužitelj i klijenti



Uobičajene zadaće na strani klijenta:

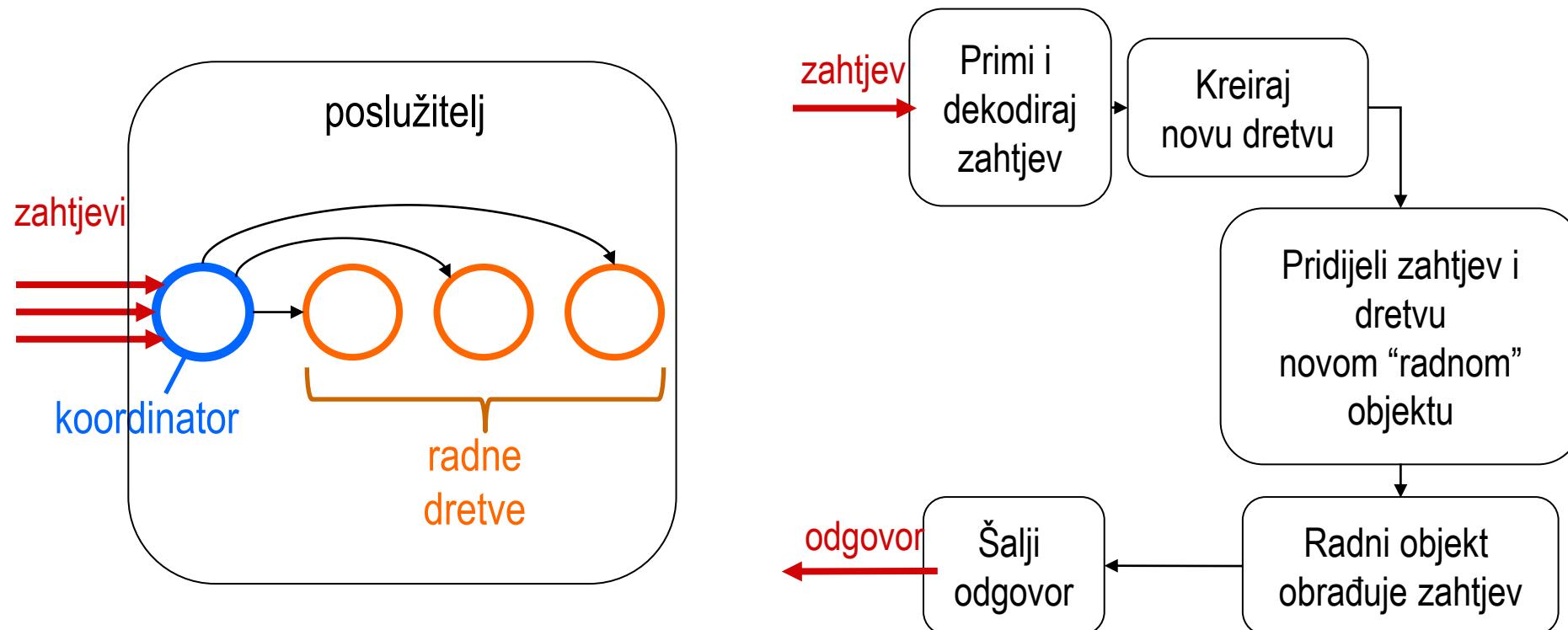
- korisničko sučelje,
- otvaranje mrežne konekcije i primanje podataka

Uobičajene zadaće na strani poslužitelja:

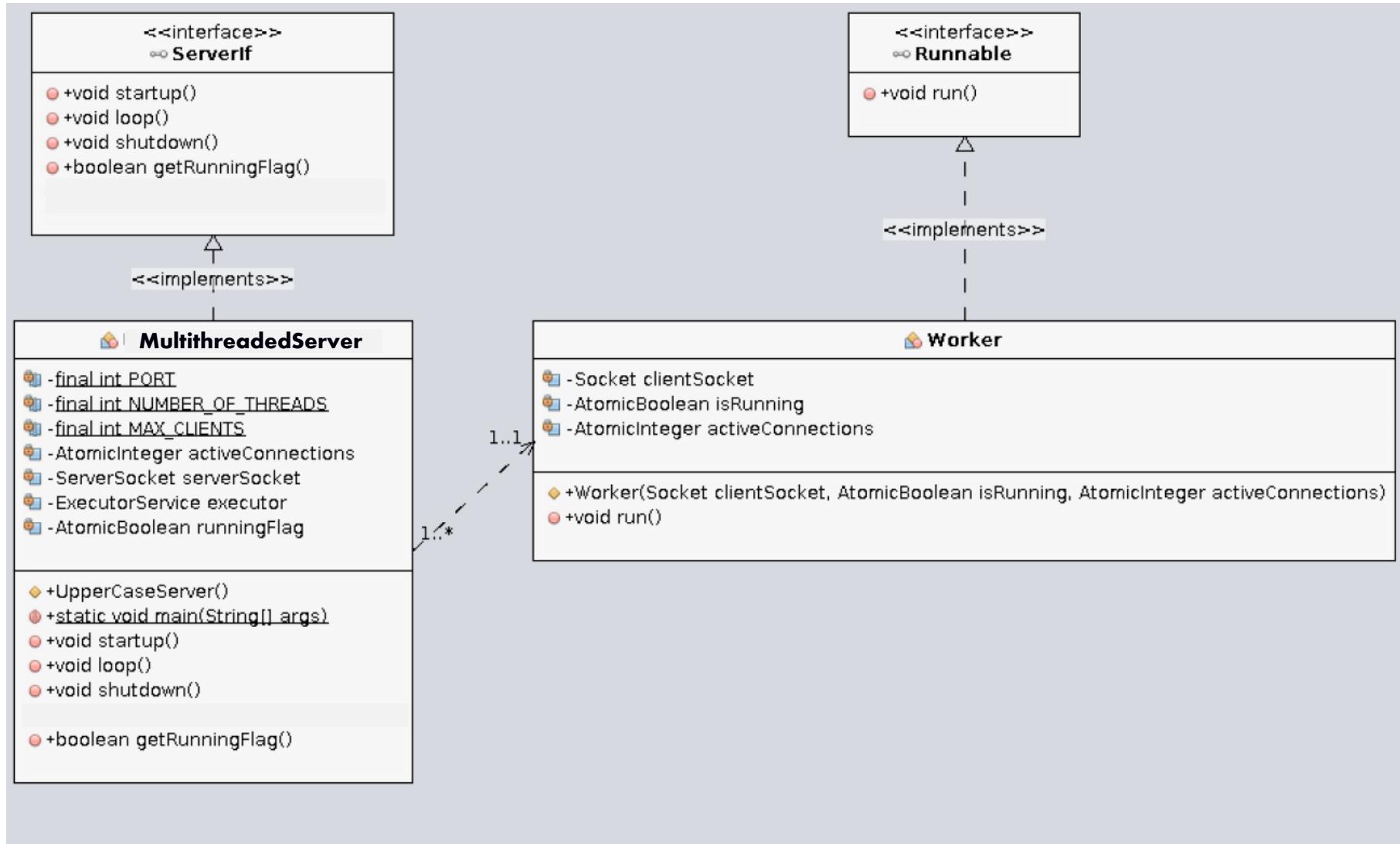
- primanje konkurentnih klijentskih zahtjeva
- složena obrada podataka
- rad s diskom/bazom podataka

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)

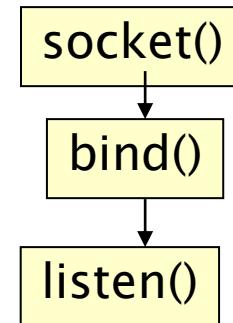
```
public class MultithreadedServer implements ServerIf {  
  
    private static final int PORT = 10002; // server port  
    private static final int NUMBER_OF_THREADS = 4;  
    //Max queue length for incoming connection requests.  
    private static final int BACKLOG = 10;  
  
    private final 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(IOExceptionex) {  
            System.err.println("Exception caught when waiting for a connection: " + ex);  
        }  
    }  
}
```

Poslužitelj (5)

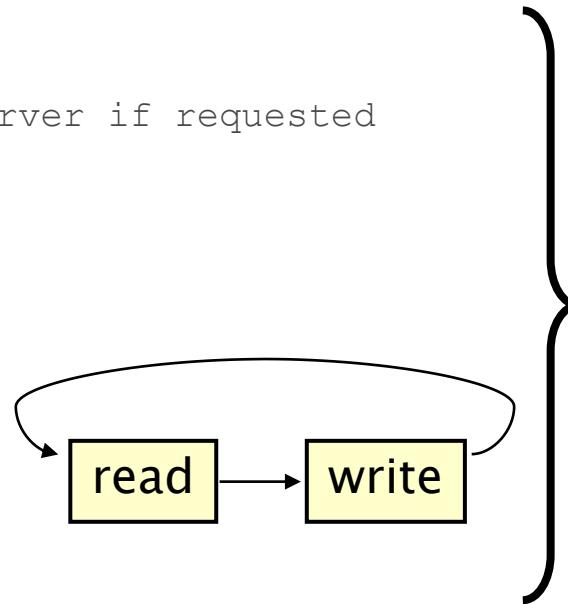
```
@Override  
public void shutdown() {  
    while( activeConnections.get() > 0 ) {  
        System.out.println( "WARNING: There are still active  
                           connections" );  
        try { Thread.sleep( 5000 );  
        } catch( java.lang.InterruptedException e ) {}  
    }  
    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);
```

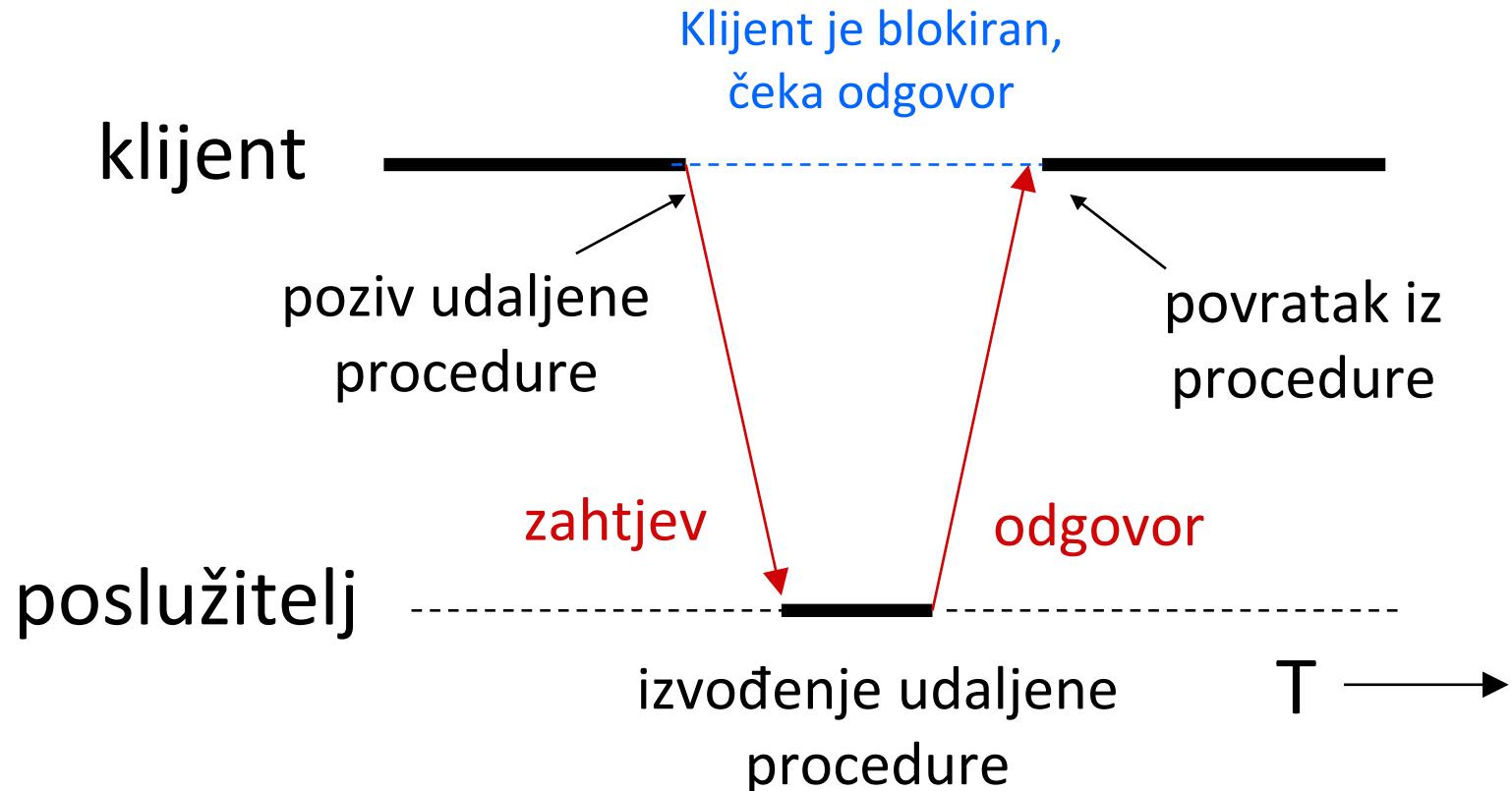


Sadržaj predavanja

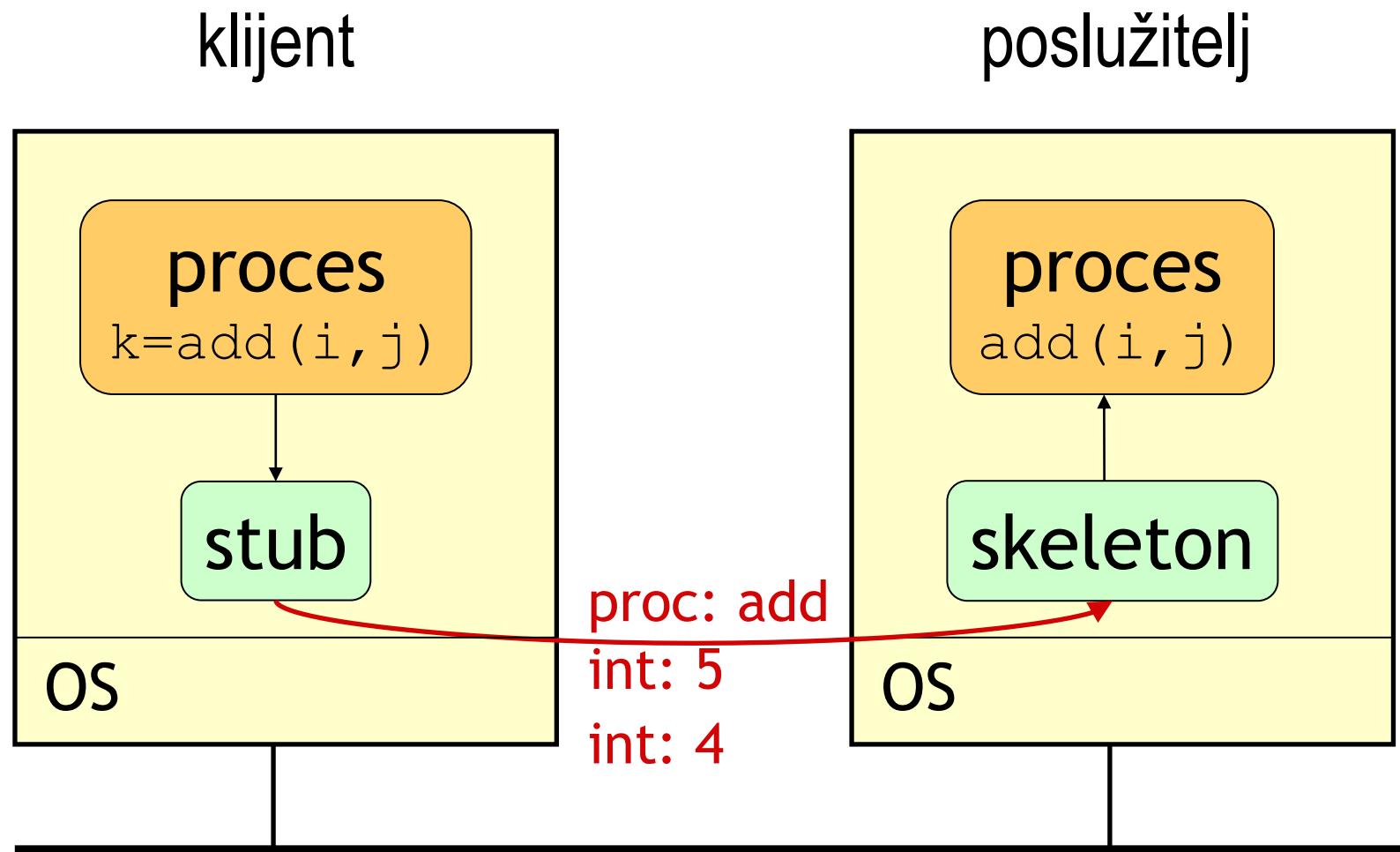
- Osnovni komunikacijski model
 - obilježja komunikacije
 - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
 - komunikacija korištenjem priključnica (Socket API)
 - primjeri TCP/UDP klijenta i poslužitelja
 - oblikovanje višedretvenog poslužitelja
 - poziv udaljene procedure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

Poziv udaljene procedure (RPC)

- Omogućuje procesima pozivanje i izvođenje procedura na udaljenom računalu.



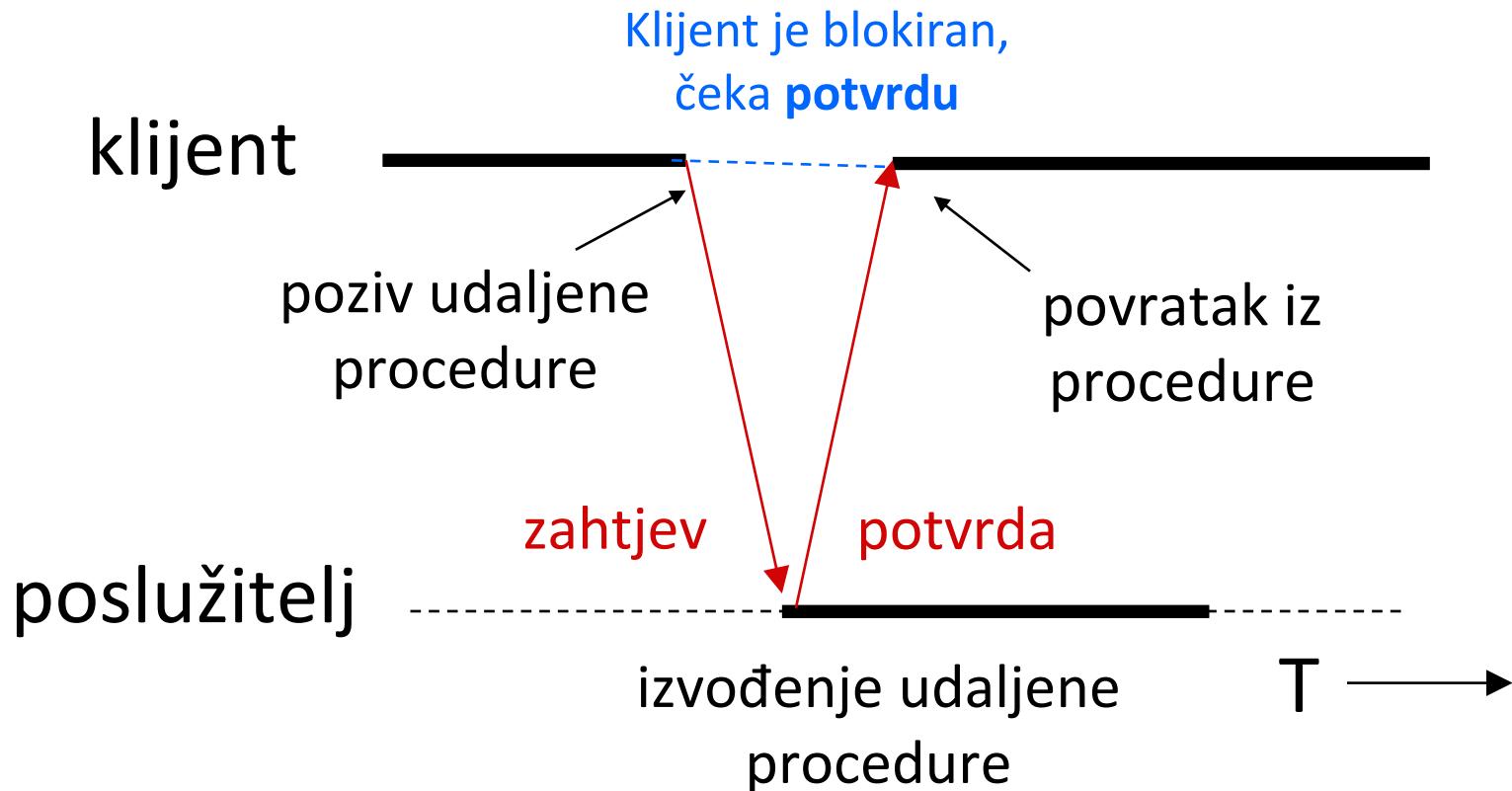
Izvođenje RPC-a



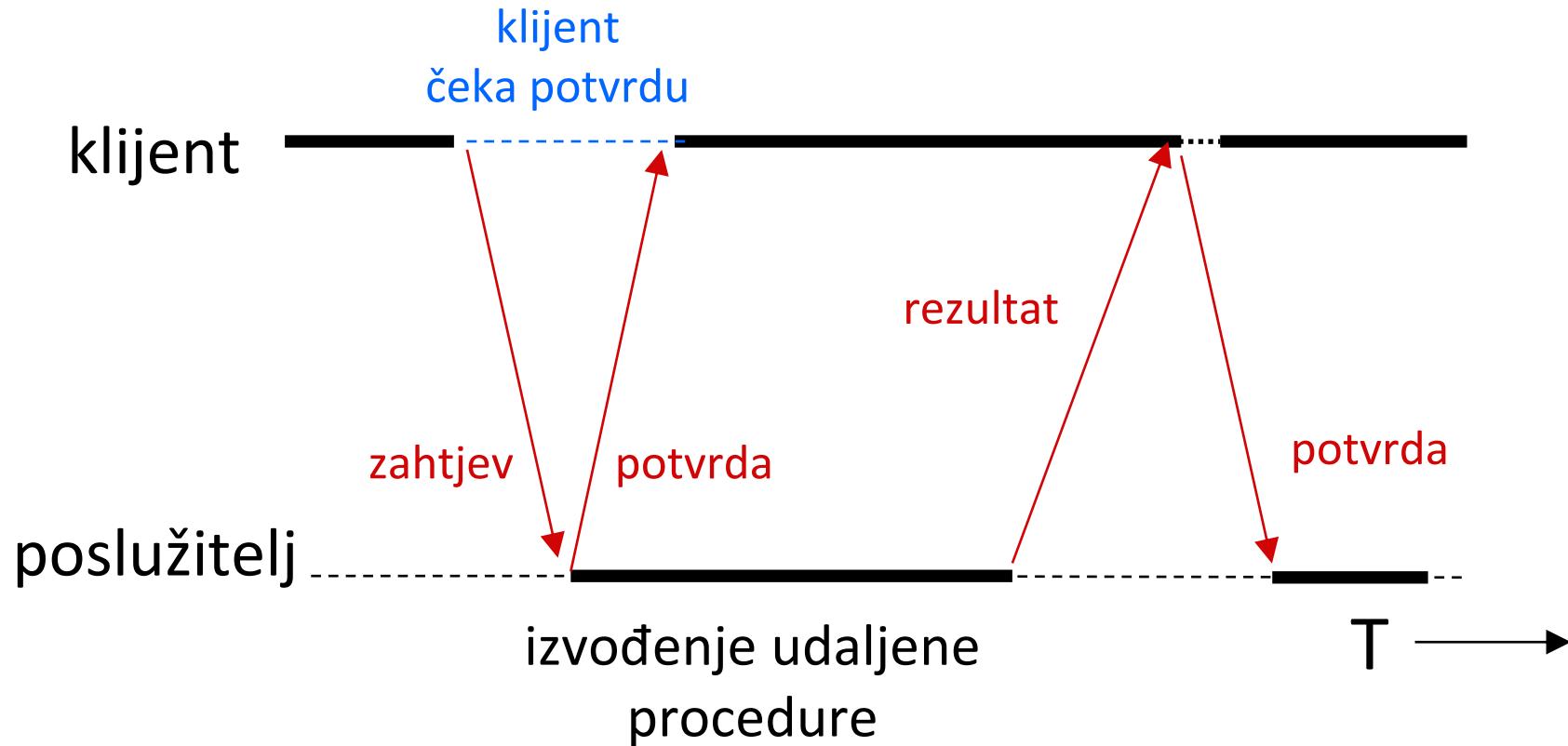
Prenošenje parametara

- *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



Odgodjeni sinkroni RPC



Poziv udaljene metode

Remote Method Invocation (RMI)

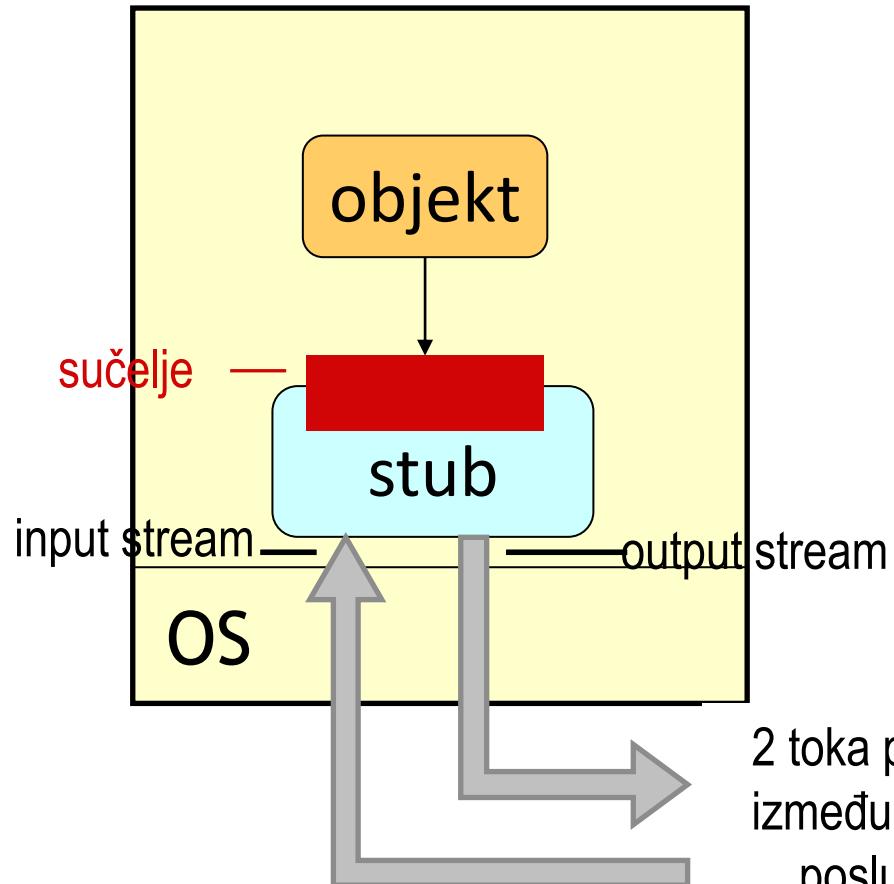
- “nasljednik” poziva udaljene procedure, poziva se metoda udaljenog objekta
- udaljeni objekt
 - proširenje osnovnog objektnog modela za raspodijeljenu okolinu
 - odvajanje sučelja i implementacije objekta
- objekt (klijent) poziva metodu udaljenog objekta (poslužitelja) na transparentan način
 - identično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

Udaljeni objekti

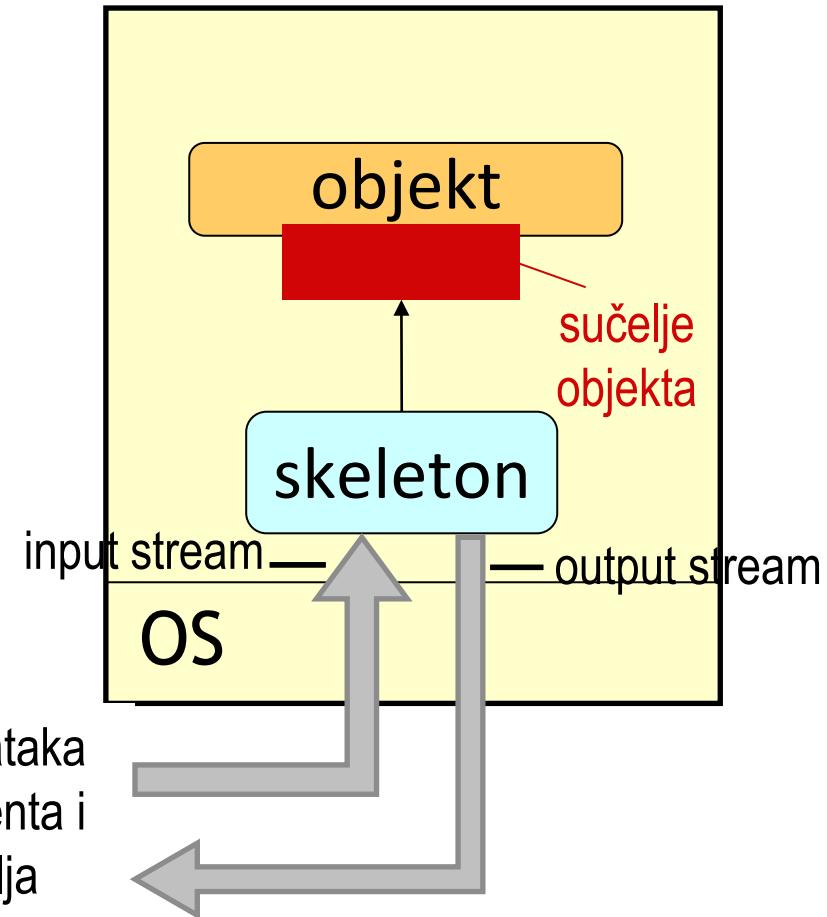
- Postoje reference na lokalne i udaljene objekte
- Svaki udaljeni objekt ima globalno jedinstven identifikator
 - npr. [ref: [endpoint:[161.53.19.24:1251](local),objID:[0]]]]
- Potrebna je usluga za registriranje i pronalaženje udaljenih objekata (*directory service*)

Izvođenje RMI-a

klijent



poslužitelj



2 toka podataka
između klijenta i
poslužitelja

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*

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*)
 - Java RMI
 - gRPC

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

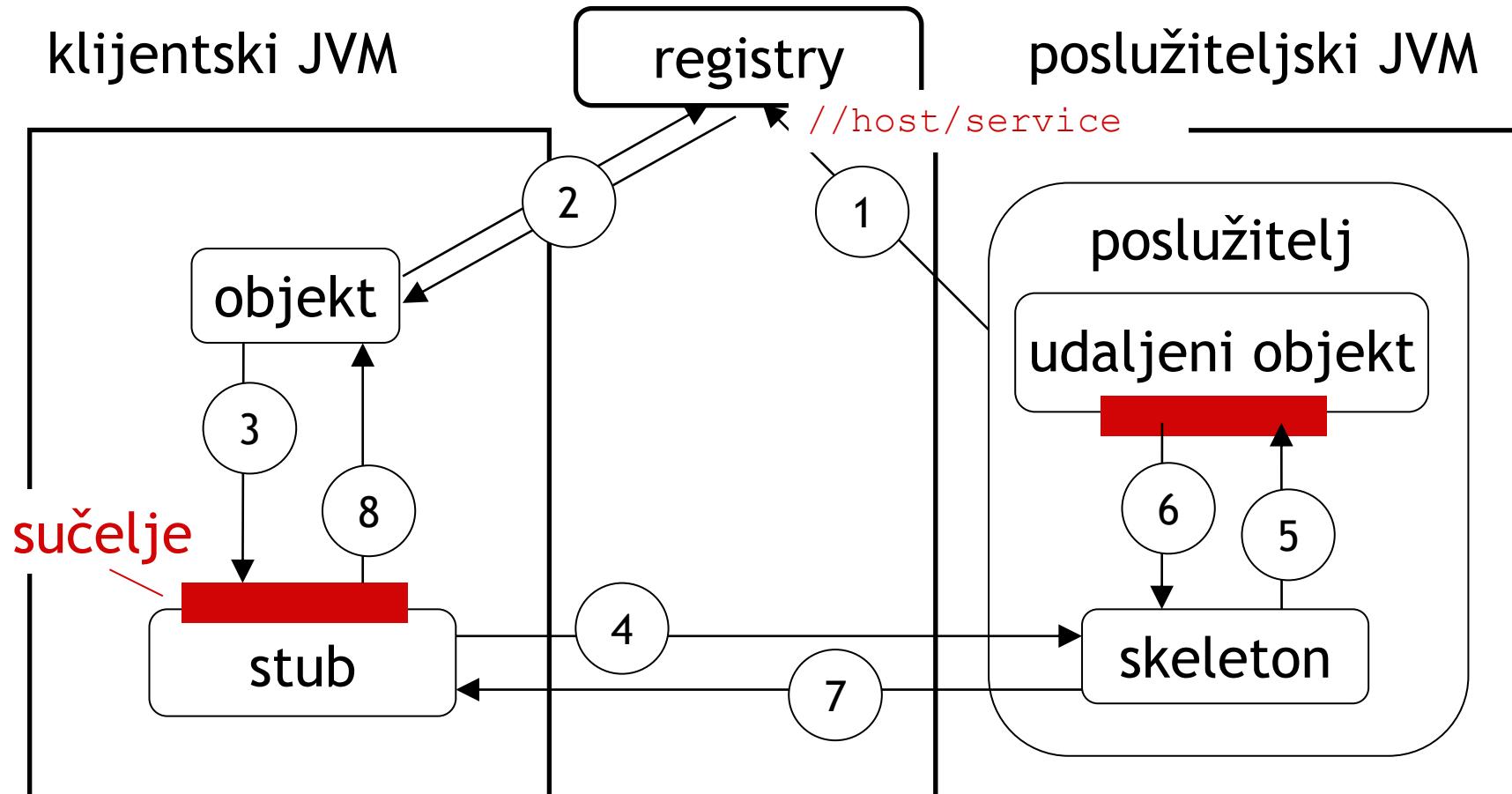
Javni model objekta

- 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 u daljenoj metodi

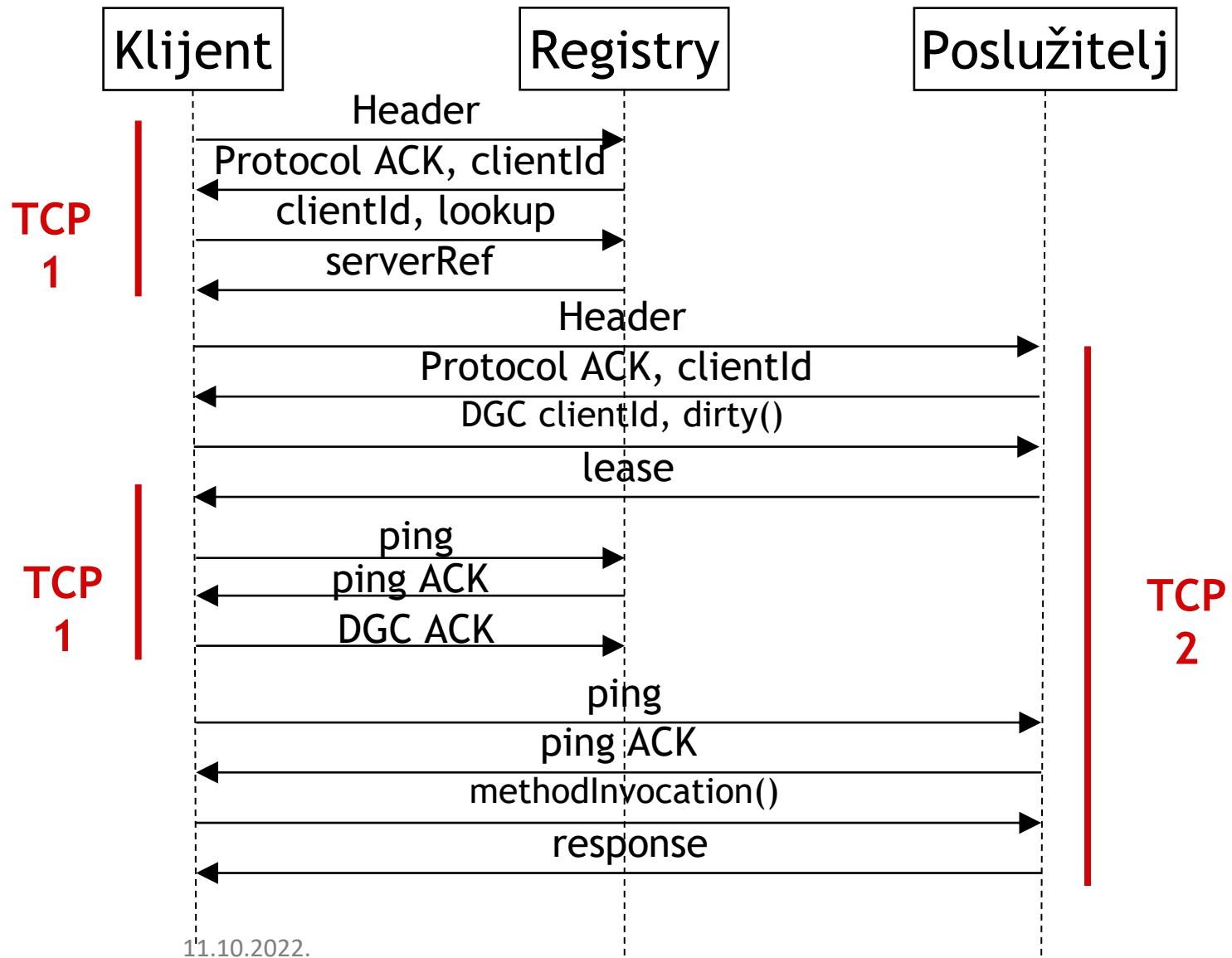
- lokalni objekti moraju se serijalizirati i prenosi se njihova vrijednost (*pass by value*)
 - implementiraju sučelje Serializable
- udaljeni se objekti prenose koristeći referencu (*pass by reference*)
 - implementiraju sučelje java.rmi.Remote i pravilno su eksportirani UnicastRemoteObject.exportObject()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

Protokol Java RMI (1)

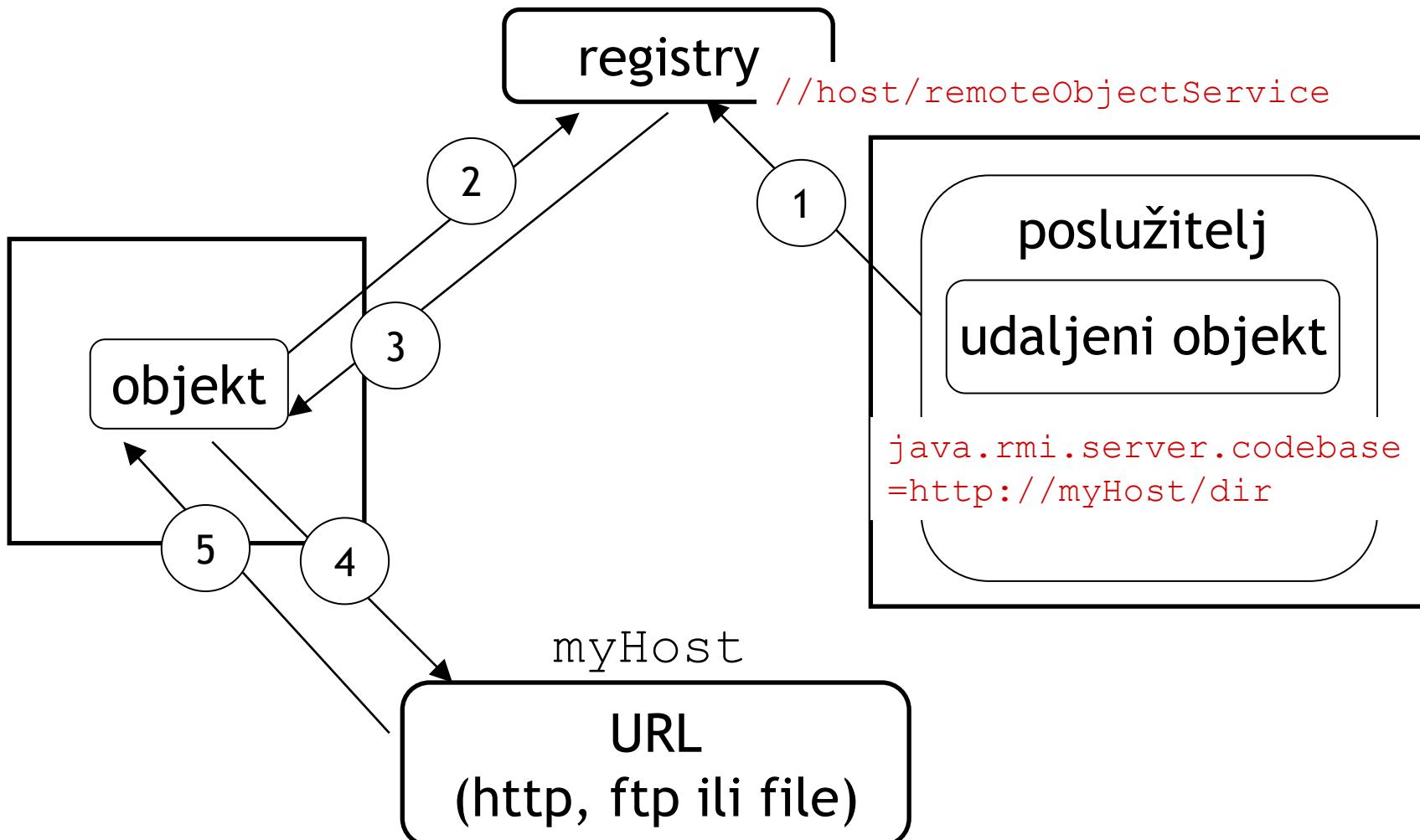


Prepostavka: stub postoji na strani klijenta

Protokol Java RMI (2)



Dinamičko učitavanje klase stuba



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 uppercase
        (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 uppercase( 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>

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*)
 - Java RMI
 - gRPC

gRPC

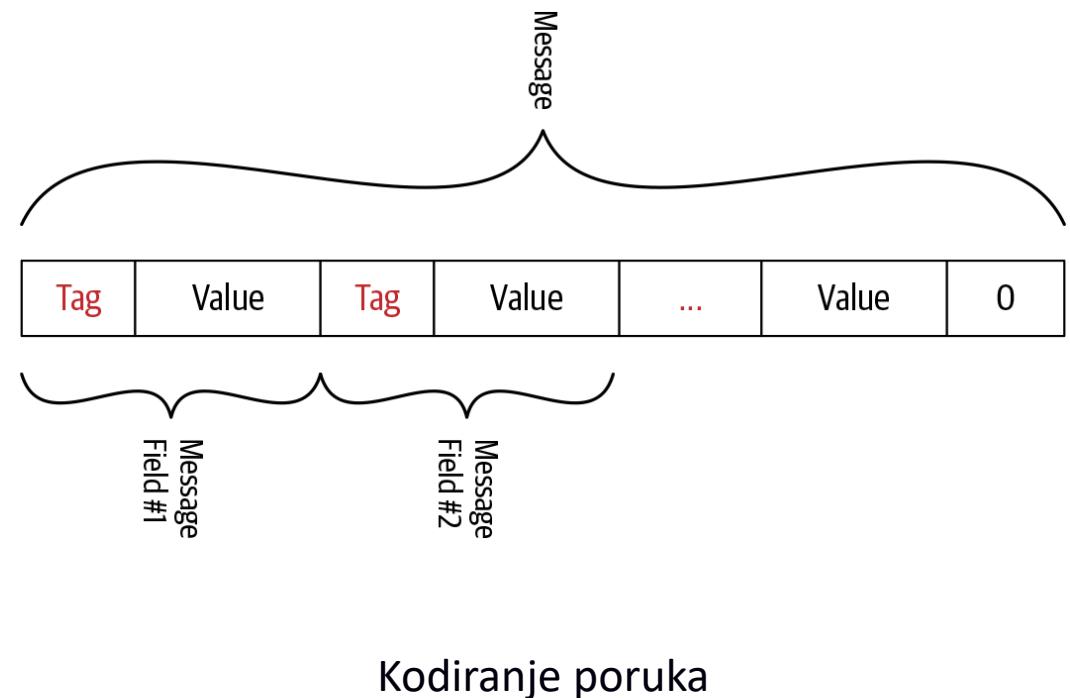
<https://grpc.io/>

- implementacija RPC-a **otvorenog koda** nastala na temelju Googleovog projekta Stubby
- podržava niz jezika: Java, Go, C++, Python...
- koristi posebnu implementaciju za prijenos i serijalizaciju podataka Protocol Buffers (HTTP/2)

- Od 2015.: *open source RPC framework*
- “*... a modern, bandwidth and CPU efficient, low latency way to create massively distributed systems that span data centers*”
- Danas ima raširenu uporabu: Netflix, Square, Lyft, Docker, Cisco, CoreOS
- Cloud Native Computing Foundation (CNCF), [https://www.cncf.io/projects/\(incubating\)](https://www.cncf.io/projects/(incubating))

Protocol buffers

- IDL za definiranje sučelja: piše se u tekstualnu datoteku .proto (koristeći jednostavni format za definiranje RPC metoda i njihovih parametara)
- Protokol je neovisan o programskom jeziku
- Vrlo učinkovit mehanizam serijalizacije podataka, binarno kodiranje (poruke su značajno manje i jednostavnije za računalnu obradu od JSON-a)



Primjer ProductInfo.proto

```
syntax = "proto3";

package ecommerce;

service ProductInfo {

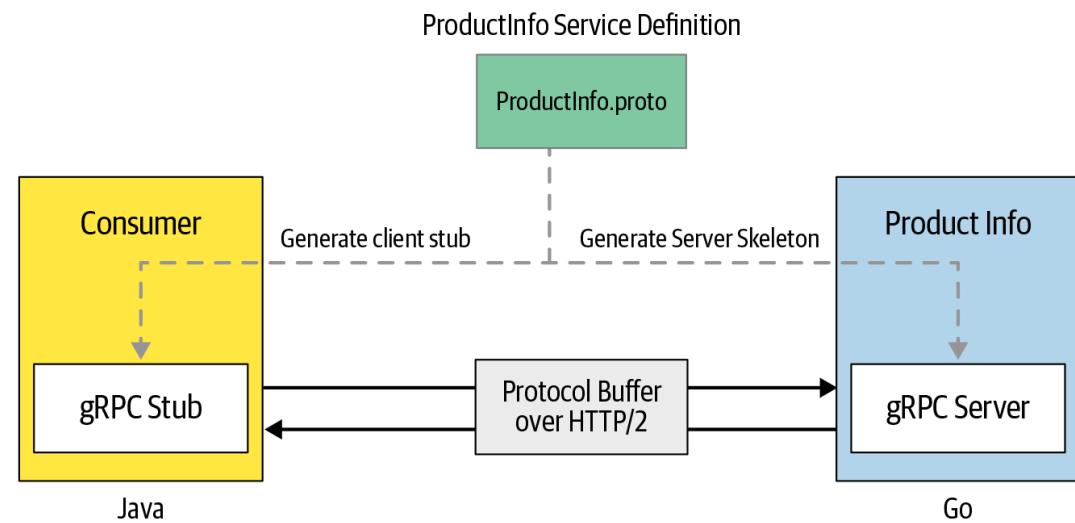
    rpc addProduct(Product) returns (ProductID);

    rpc getProduct(ProductID) returns (Product);
}

message Product {
    string id = 1;
    string name = 2;
    string description = 3;
}

message ProductID {
    string value = 1;
}
```

Koristi se kompjajler *protoc* za generiranje poslužiteljskog i klijentskog koda



Primjer gRPC for UpperCase

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "hr.fer.tel.rassus.examples";
package hr.fer.tel.rassus;

// The uppercase service definition.
service Uppercase {
    // Sends back a message converted to upper case
    rpc RequestUppercase (Message) returns (Message) {}
}

// Definition of request and response message
message Message {
    string payload = 1;
}
```

Primjer definicije
sučelja servisa
Uppercase koje definira
samo jednu metodu
RPC RequestUppercase

Koraci prilikom implementacije i izvođenja

Poslužitelj

1. generiraj skeleton servisa na temelju .proto opisa
2. dodaj logiku generiranim metodama (*override*)
3. pokreni poslužitelja koji će kontinuirano primati korisničke zahtjeve

Klijent

1. kreiraj konekciju (kanal) prema udaljenom poslužitelju
2. poveži klijentski stub uz tu konekciju
3. pozovi udaljenu metodu na poslužitelju, a implementacija Protocol Buffers će se pobrinuti da prenese podatke u jednom i drugom smjeru

Uppercase service

```
public class UppercaseService extends UppercaseGrpc.UppercaseImplBase {  
    private static final Logger logger =  
        Logger.getLogger(UppercaseService.class.getName()); // Generira plug-in  
    @Override  
    public void requestUppercase( // Prima obavijesti iz stremna poruka (observable pattern), koristi se za slanje i primanje poruka  
        Message request, StreamObserver<Message> responseObserver ) {  
        logger.info("Got a new message: " + request.getPayload());  
        // Create response  
        Message response =  
            Message.newBuilder().setPayload(request.getPayload().toUpperCase()).build();  
        // Send response  
        responseObserver.onNext( response ); // Šalje odgovor klijentu  
        logger.info("Responding with: " + response.getPayload());  
        // Send a notification of successful stream completion.  
        responseObserver.onCompleted(); // Zatvori stream  
    } }
```

gRPC poslužitelj (1/2)

```
public class SimpleUnaryRPCServer {  
    private static final Logger logger = Logger.getLogger(SimpleUnaryRPCServer.class.getName());  
    private Server server;  
    private final UppercaseService service;  
    private final int port;  
  
    public SimpleUnaryRPCServer(UppercaseService service, int port) {  
        this.service = service;  
        this.port = port;  
    }  
    public void start() throws IOException {  
        // Register the service  
        server = ServerBuilder.forPort(port).addService(service).build().start();  
        logger.info("Server started on " + port);  
        // Clean shutdown of server in case of JVM shutdown  
        Runtime.getRuntime().addShutdownHook(new Thread(() -> { System.err.println("Shutting down gRPC server since JVM is  
shutting down");  
            try {  
                SimpleUnaryRPCServer.this.stop();  
            } catch (InterruptedException e) {  
                e.printStackTrace(System.err);  
            }  
            System.err.println("Server shut down");  
        }));  
    }...  
}
```

Kreira instancu poslužitelja koji osluškuje na definiranom portu

gRPC poslužitelj (2/2)

```
public void stop() throws InterruptedException {
    if (server != null) {
        server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
    }
}

public void blockUntilShutdown() throws InterruptedException {
    if (server != null) {
        server.awaitTermination();
    }
}

public static void main(String[] args) throws IOException, InterruptedException {
    final SimpleUnaryRPCServer server = new SimpleUnaryRPCServer(new UppercaseService(),
3000);
    server.start();
    server.blockUntilShutdown();
}
```

Radna dretva poslužitelja se zadržava sve dok ne stigne zahtjev za gašenjem poslužitelja.

gRPC klijent (1/2)

```
public class SimpleUnaryRPCCClient {  
    private static final Logger logger =  
        Logger.getLogger(SimpleUnaryRPCCClient.class.getName());  
    private final ManagedChannel channel;  
    private final UppercaseGrpc.UppercaseBlockingStub uppercaseBlockingStub;  
  
    public SimpleUnaryRPCCClient(String host, int port) {  
        this.channel = ManagedChannelBuilder.forAddress(host, port).usePlaintext().build();  
        uppercaseBlockingStub = UppercaseGrpc.newBlockingStub(channel);  
    }  
    public void stop() throws InterruptedException {  
        // Initiates an orderly shutdown in which preexisting calls continue but new calls are  
        // immediately cancelled. Waits for the channel to become terminated, giving up if the  
        // timeout is reached.  
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);  
    } ...  
}
```

Kreira gRPC kanal prema transportnoj adresi poslužitelja.

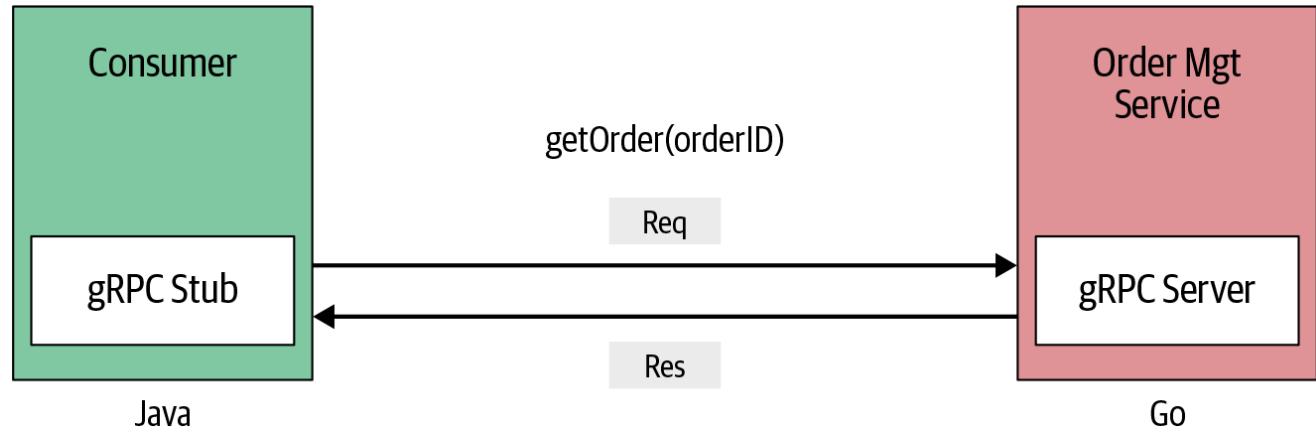
Kreira klijentski stub (blokirajući) pomoću kanala.

gRPC klijent (2/2)

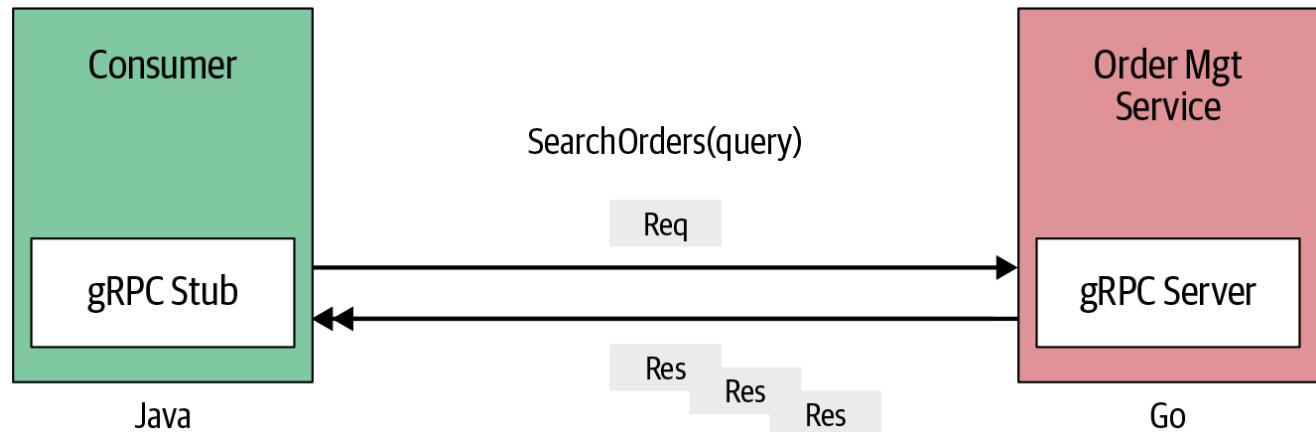
```
public void requestUppercase() {  
    final String payload = "message";  
    Message request = Message.newBuilder().setPayload(payload).build();  
    logger.info("Sending: " + request.getPayload());  
    try {  
        Message response = uppercaseBlockingStub.requestUppercase(request);  
        logger.info("Received: " + response.getPayload());  
    } catch (StatusRuntimeException e) {  
        logger.info("RPC failed: " + e.getMessage());  
    }  
}  
  
public static void main(String[] args) throws InterruptedException {  
    SimpleUnaryRPCClient client = new SimpleUnaryRPCClient("127.0.0.1", 3000);  
    client.requestUppercase();  
    client.stop();  
}
```

Mogući oblici komunikacije (1/2)

- Simple RPC (Unary RPC)

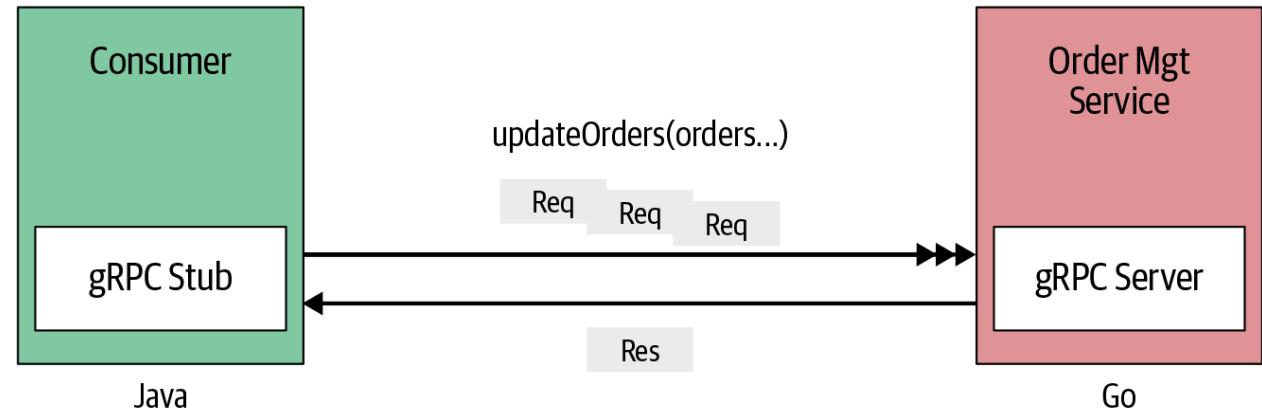


- Server-Streaming RPC

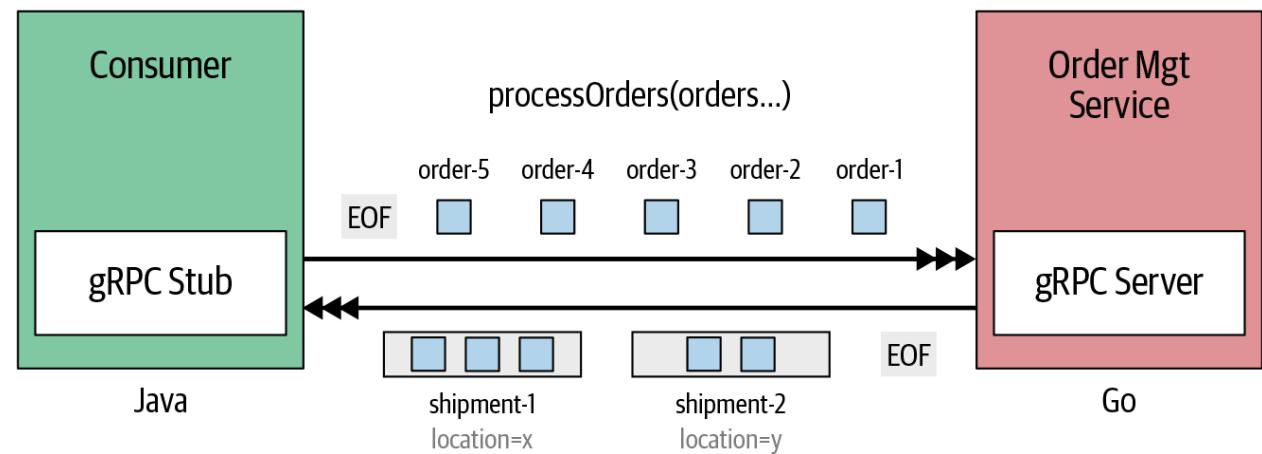


Mogući oblici komunikacije (2/2)

- **Client-Streaming RPC**



- **Bidirectional-Streaming RPC**



Pitanja za učenje i ponavljanje

- Objasnite zašto tranzientna 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?
- Usporedite gRPC i RESTful servise u pogledu performansi.

Literatura

- Maarten van Steen, Andrew S. Tanenbaum (2017.), *Distributed Systems 3rd edition*, Createspace Independent Publishing Platform poglavlja 4.2 (RPC) i 4.3 (samo dio o Socketima)
- G. Coulouris, J. Dollimore, T. Kindberg: *Distributed Systems: Concepts and Design*, 5th edition, Addison-Wesley, 2012 poglavlja 4.1, 4.2 i 4.3. (bez 4.3.1 CORBA i 4.3.3 XML) i poglavlje 5



SVEUČILIŠTE U ZAGREBU



Diplomski studij

Računarstvo

Raspodijeljeni sustavi

3. Arhitekture web-aplikacija,
tehnologije weba

Ak. god. 2022./2023.

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 daljnog 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

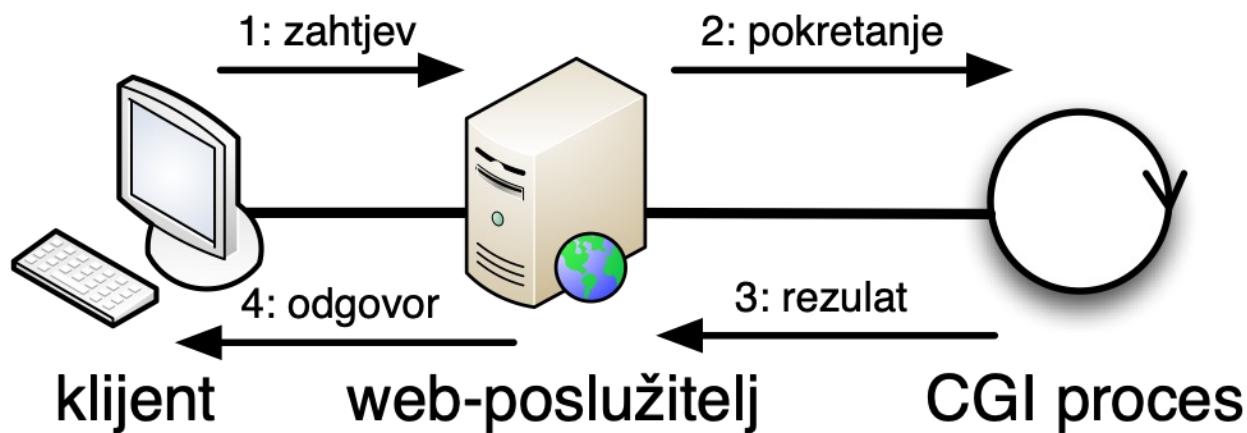
- arhitekture web-aplikacija
- AJAX
- uvod u Web 2.0
- Web-usluge
- jezici i protokoli: SOAP, WSDL, UDDI
- web-usluge temeljene na RPC-u, temeljene na dokumentima
- Web-usluge temeljene prijenosu prikaza stanja resursa (REST)
- Svojstva metoda protokola HTTP
- Model zrelosti web-usluga i relevantni standardi
- Implementacija REST-a u Springu

Web-aplikacije

- Definicija:
 - "*Web applications are stored on web servers, and use tools like databases, JavaScript (or Ajax or Silverlight), and PHP (or ASP.Net) to deliver experiences beyond the standard web page or web form.*"
- dvije vrste:
 - koje izgledaju kao normalne web-stranice (npr. portali)
 - koje izgledaju kako normalne aplikacije - bogato korisničko sučelje (npr. Google Mail)
- koriste tehnologije weba:
 - HTML, CSS, JavaScript, PHP, ASP, JSP, Ruby on Rails, Java Servlets, Cold Fusion, ...

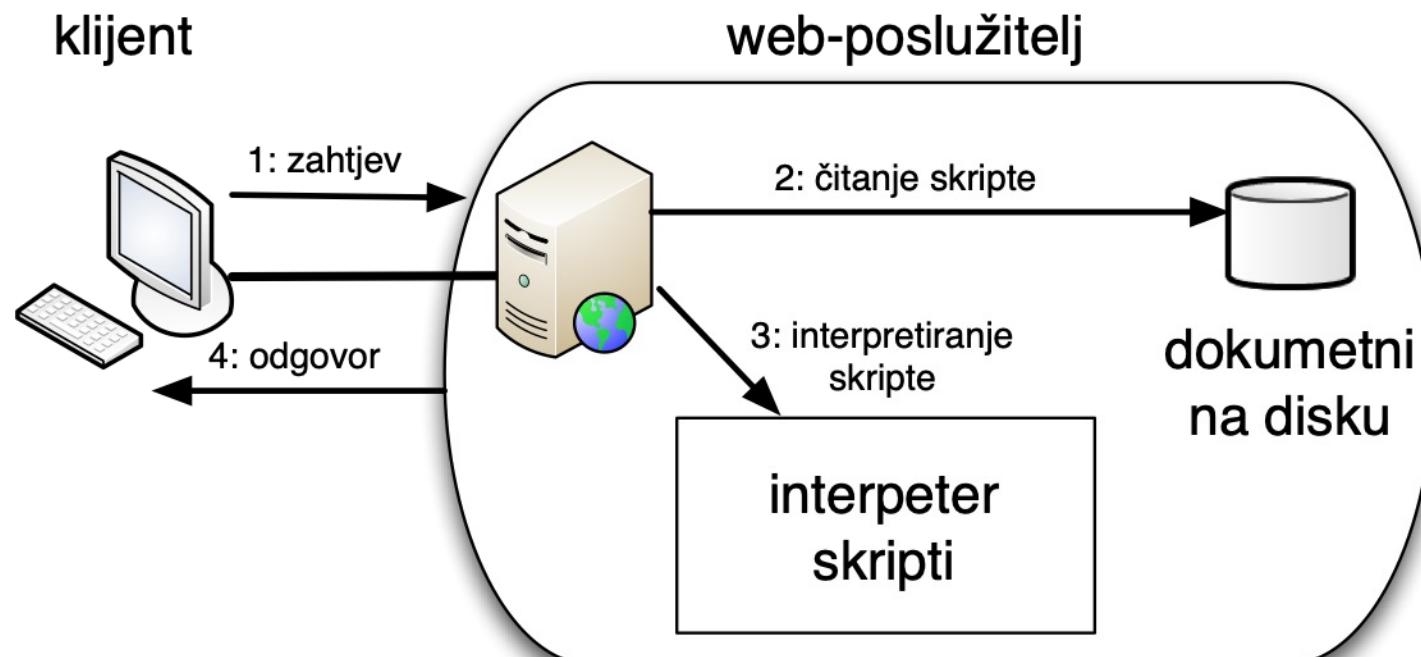
Web-aplikacije - CGI

- CGI - *Common Gateway Interface* - RFC 3875
- kod svakog zahtjeva se pokreće proces
- podaci između poslužitelja i procesa se šalju preko varijabli okoline i tokova podataka
- nakon svake obrade proces se gasi
- Bash, Perl i ostali skriptni jezici



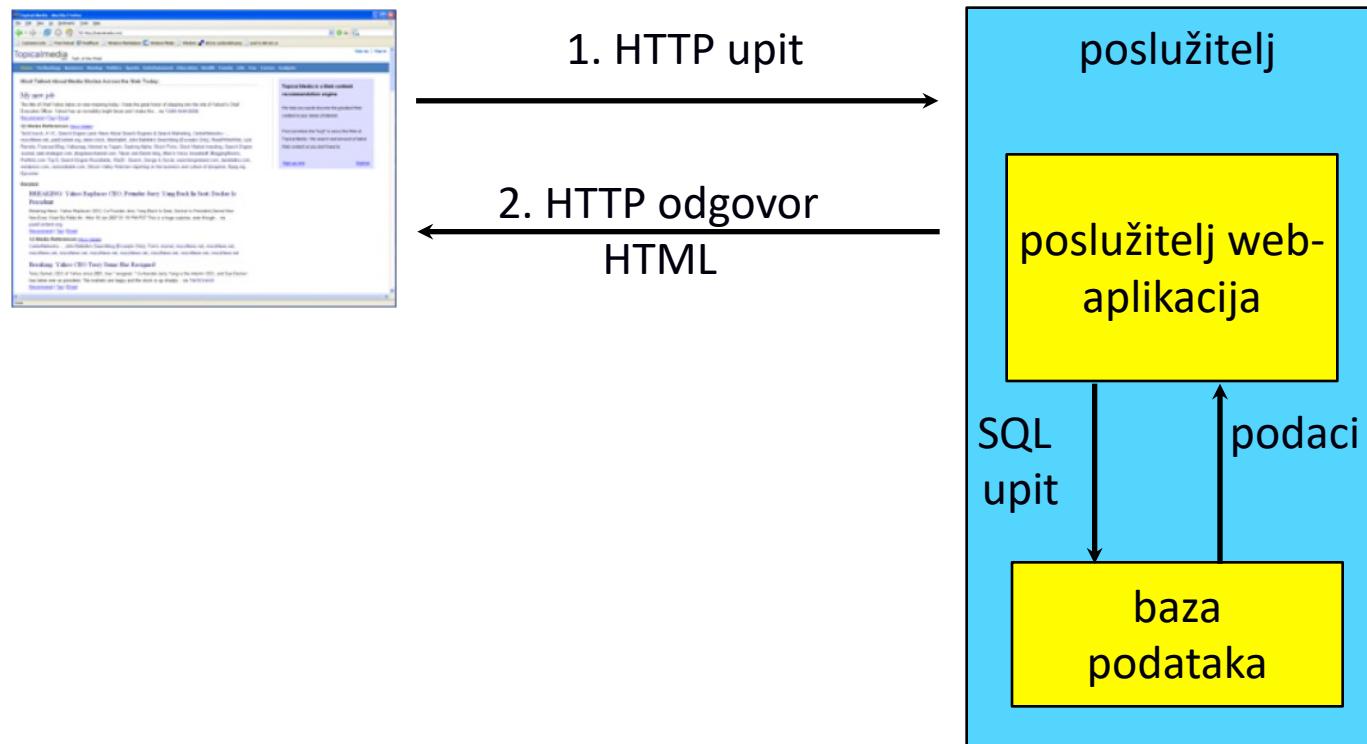
Web-aplikacije - poslužiteljske skripte

- poslužiteljske skripte - *server side scripts*
- dinamički se generira HTML-dokument (iz skripte)
- primjeri: PHP, ASP, JSP, Django, Ruby on Rails, ...



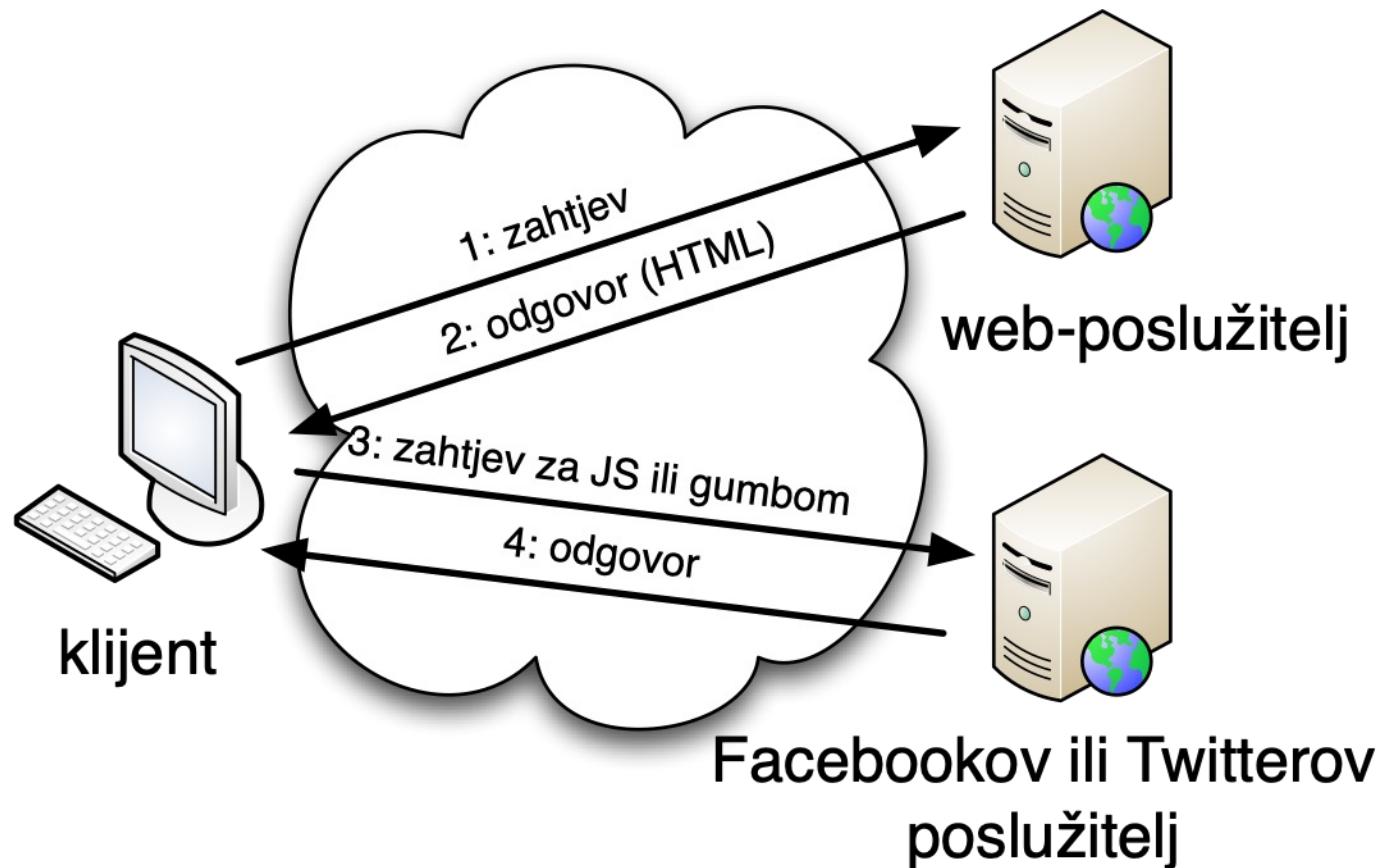
Tipična web-aplikacija na jednom računalu

- razvijateljska konfiguracija
- dobro za mali broj zahtjeva u produkciji



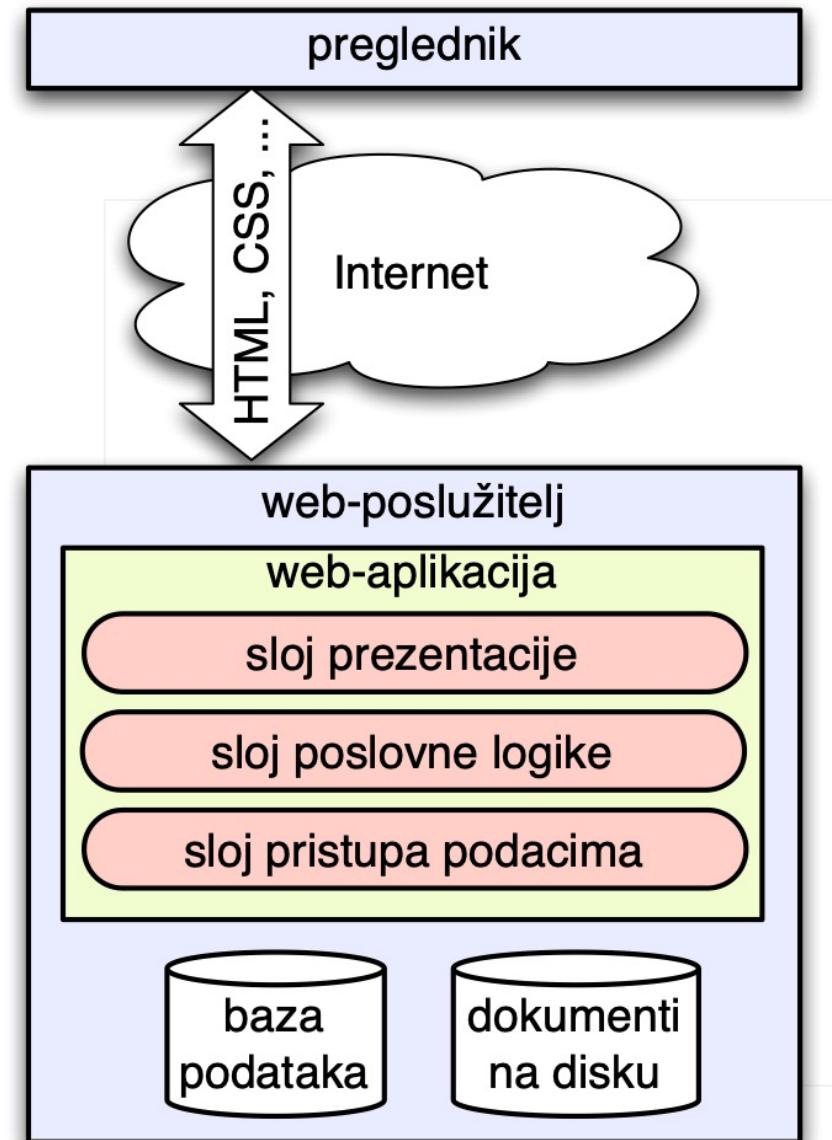
Web-aplikacija integrirana s drugim sjedištima

- gumbi za objavljivanje na drugim stranicama (npr. Facebook ili Twitter)



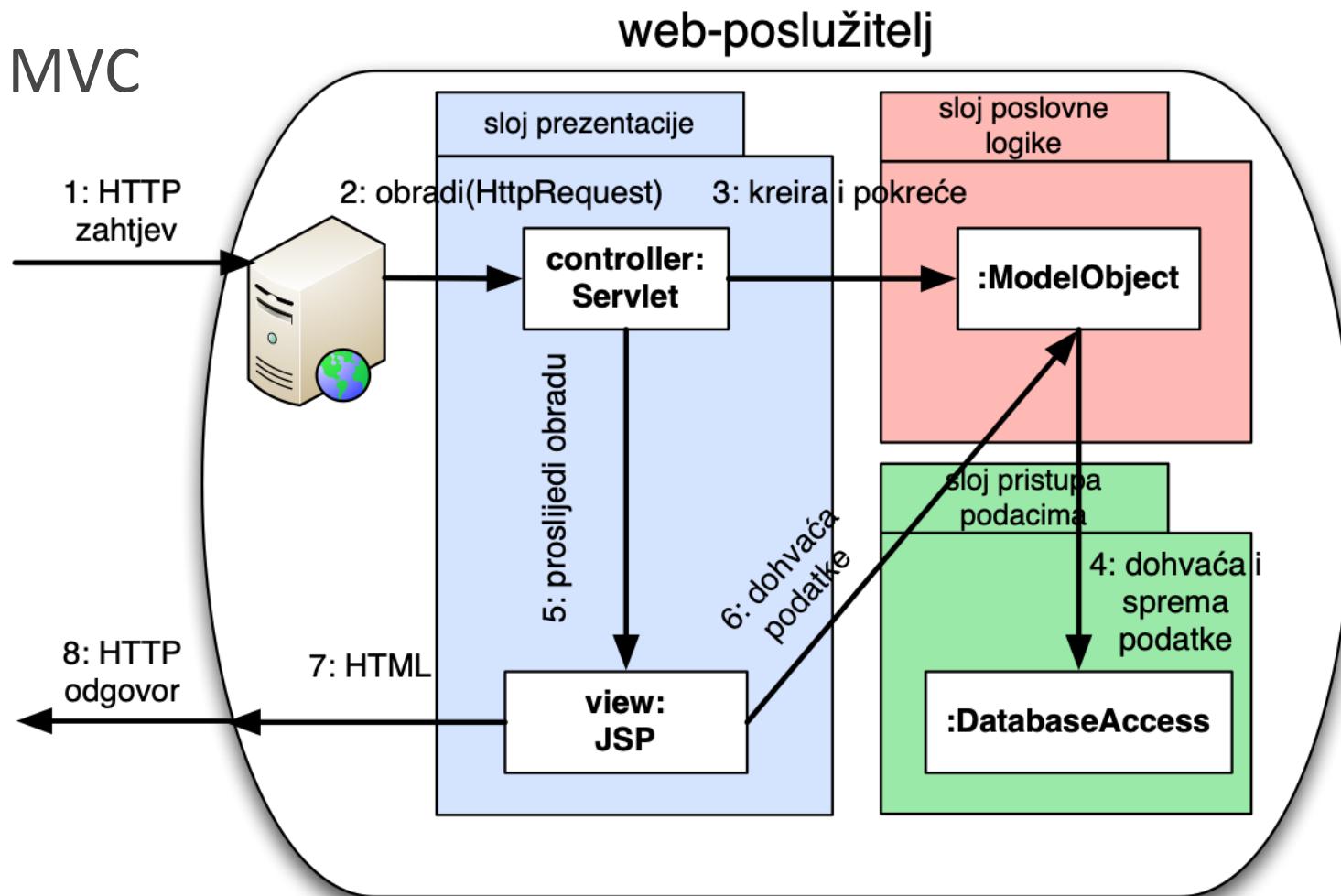
Arhitektura web-aplikacija

- višeslojna arhitektura
- sloj prezentacije
 - prikaz informacija: GUI, HTML, klikovi mišem, ...
 - obrada HTTP zahtjeva
- sloj poslovne (domenske) logike
 - obrada podataka
 - glavni dio sustava koji radi ono za što je sustav namijenjen
- sloj pristupa podacima
 - komunicira s bazom podataka i drugim komunikacijskim sustavima
 - brine se o transakcijama
 - brine se za pohranu podataka



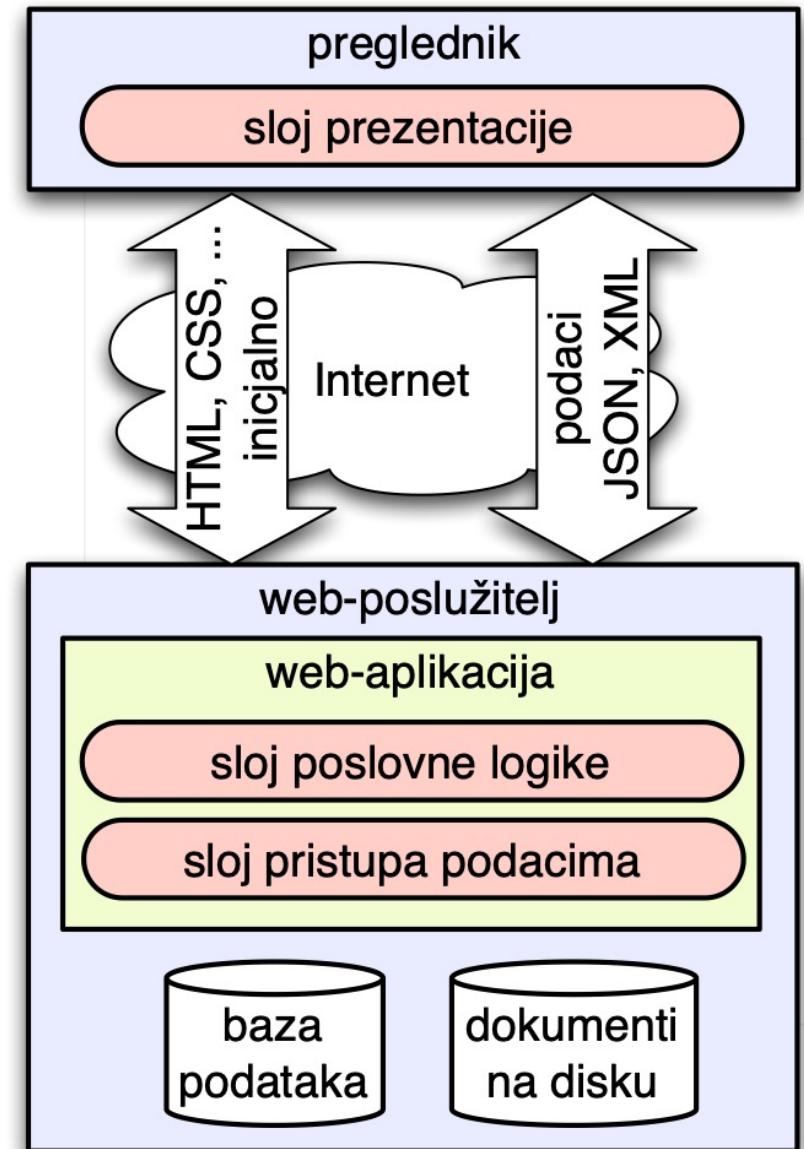
Arhitektura web-aplikacija - MVC u Javi

- *Model View Controller - MVC*



Nova arhitektura web-aplikacija

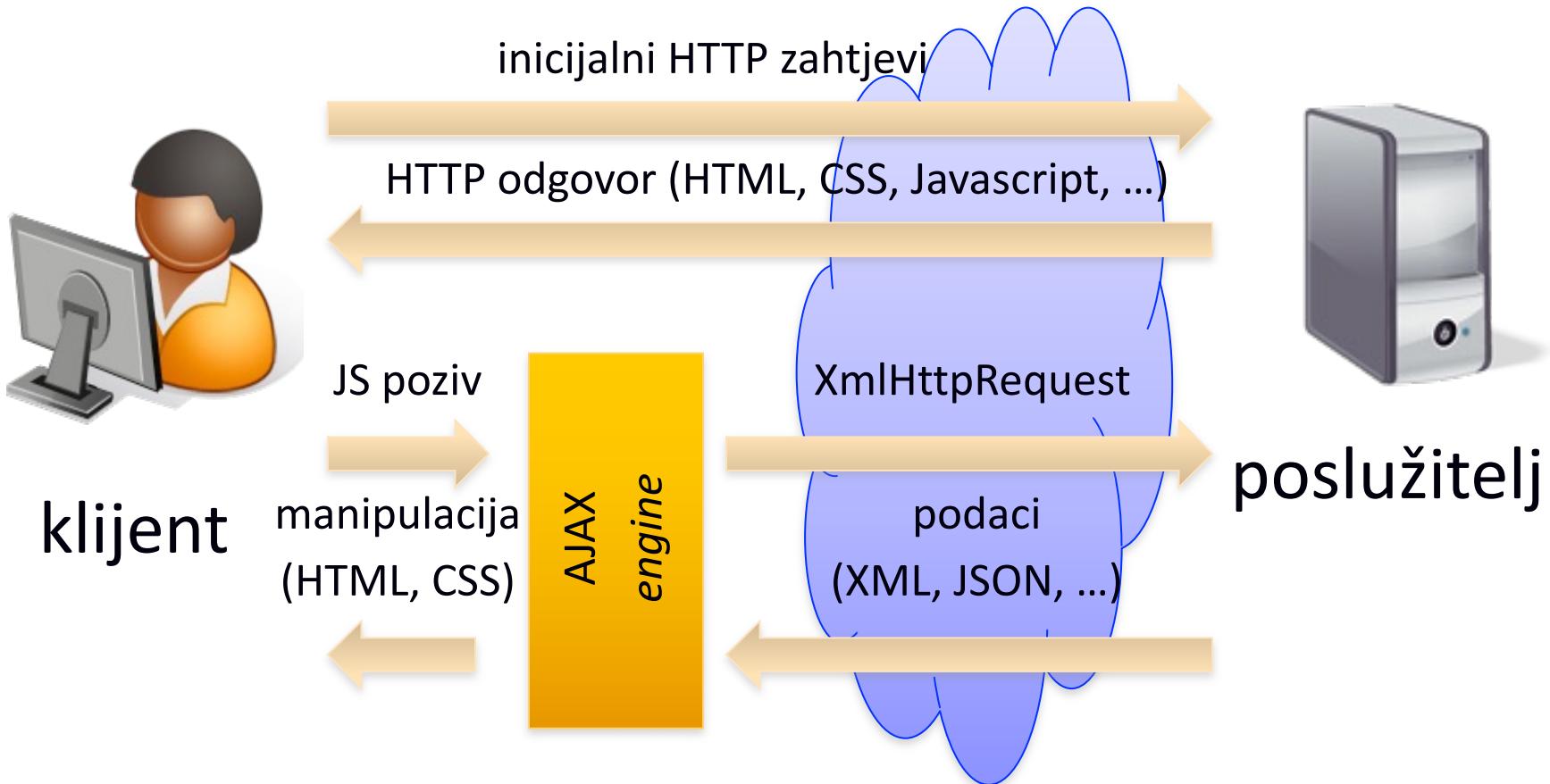
- inicijalno pokretanje
 - HTML, CSS, JavaScript, ...
- za vrijeme korištenja aplikacije
 - podaci putuju u obliku:
 - JSON - JavaScript Object Notation ili
 - XML
 - koristi se AJAX - [Asynchronous JavaScript](#) and XML



Skripte na klijentu - dio sloja prezentacije

- uključene u HTML ili u posebnoj datoteci koja je povezana
- obično se koristi:
 - **JavaScript** (Netscape), JScript (Microsoft), ECMAScript
- ECMAScript – standardiziran
 - specifikacije ECMA-262 i ISO/IEC 16262
 - svojstva: dinamički, slabo povezan, objektni, funkcijski
 - nema veze s jezikom Java osim imena JavaScript
- svrha:
 - dinamički elementi
 - interakcija s korisnikom
 - provjera obrazaca
 - komunikacija s poslužiteljem (AJAX)
 - ...

AJAX (Asynchronous JavaScript and XML)



- iz JavaScripta poslati upit na poslužitelj
- odgovor nije nova stranica već podatak (u formatu **JSON**, **XML** ili ...)
- dobije se na dinamičnosti web-stranice

Poznate knjižnice u JavaScriptu

- popis se stalno mijenja i stalno raste
 - <https://www.javascripting.com>
- DOM manipulation:
 - React (<https://facebook.github.io/react/>)
 - Gatsby (<https://www.gatsbyjs.com>)
 - Chakra Ui (<https://chakra-ui.com>)
- Korisničko sučelje (*User Interface*):
 - Ant Designl (<http://ant.design/>)
 - Material Ui (<https://material-ui.com/>)
 - Github Readme Stats (<http://github.com/anuraghazra/github-readme-stats>)
- Općenito:
 - React Native (<http://facebook.github.io/react-native/>)
 - Next.js (<https://zeit.co/blog/next>)
 - Electron (<https://electron.atom.io/>)
 - Ant Design (<https://ant.design>)

Web 2.0

- pojam nastao u O'Reilly Media (2003.)
- ideja: Internet kao platforma poput operacijskog sustava
- potiče inovacije i sastavljanje funkcionalnih dijelova iz neovisnih izvora
- korisnici postaju središte
 - interaktivnost, sudjelovanje, objavljivanje
- višemedijski sadržaji (glazba, video, lokacije, slike, karte, ...)

Usporedba weba 1.0 i 2.0

Web 1.0

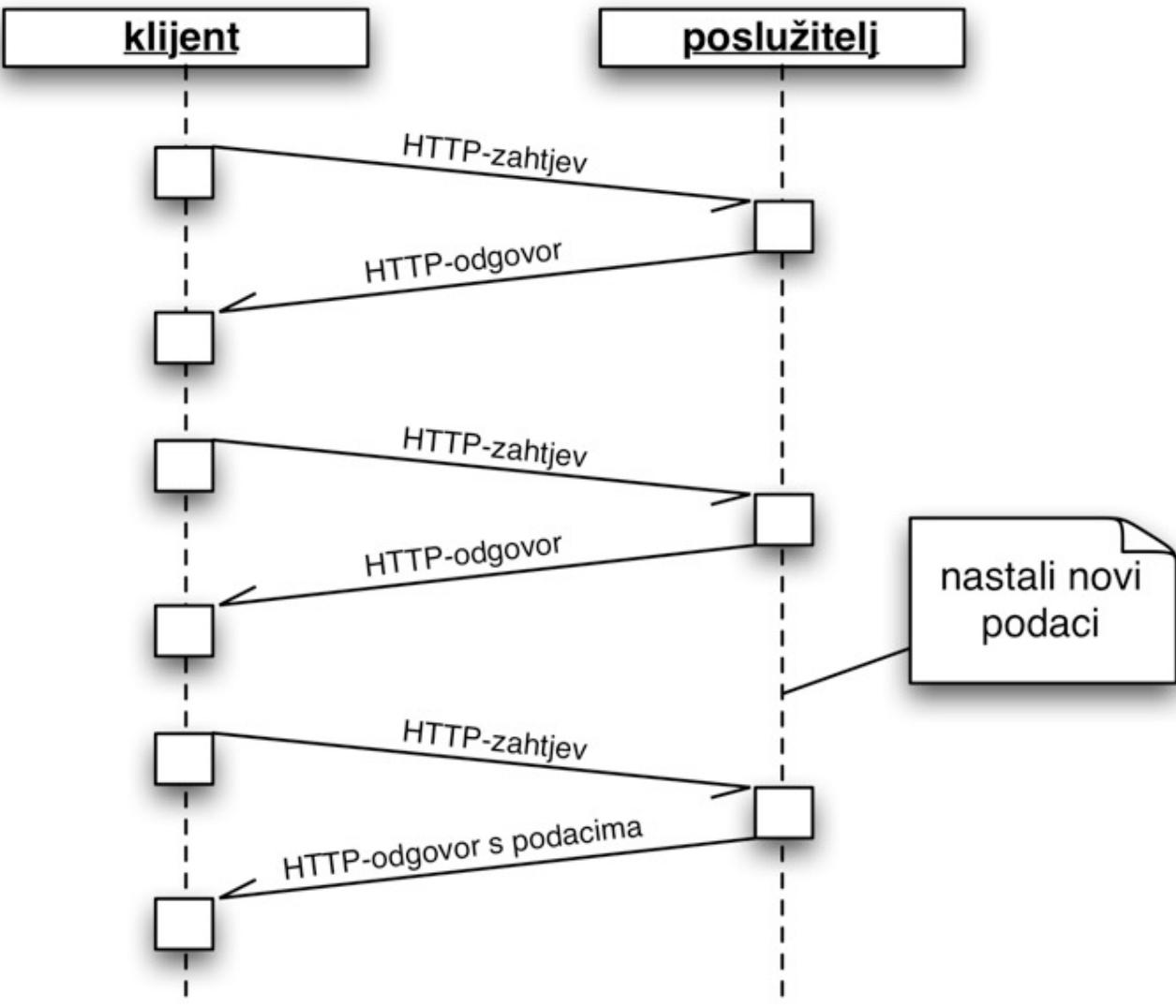
- čitanje informacija
- klijent-poslužitelj
- HTML
- portali
- parsiranje informacija
- posjedovanje
- modemska veza i HW
- direktoriji
- tipična tvrtka: Netscape

Web 2.0

- pisanje informacija
- P2P (osobe i aplikacije)
- XML, JSON, HTML5
- usluge (*services*)
- REST API sučelja, RSS
- dijeljenje
- širokopojasna veza i SW
- oznake (*tag*)
- tipične tvrtke: Google, Facebook, Twitter, ...

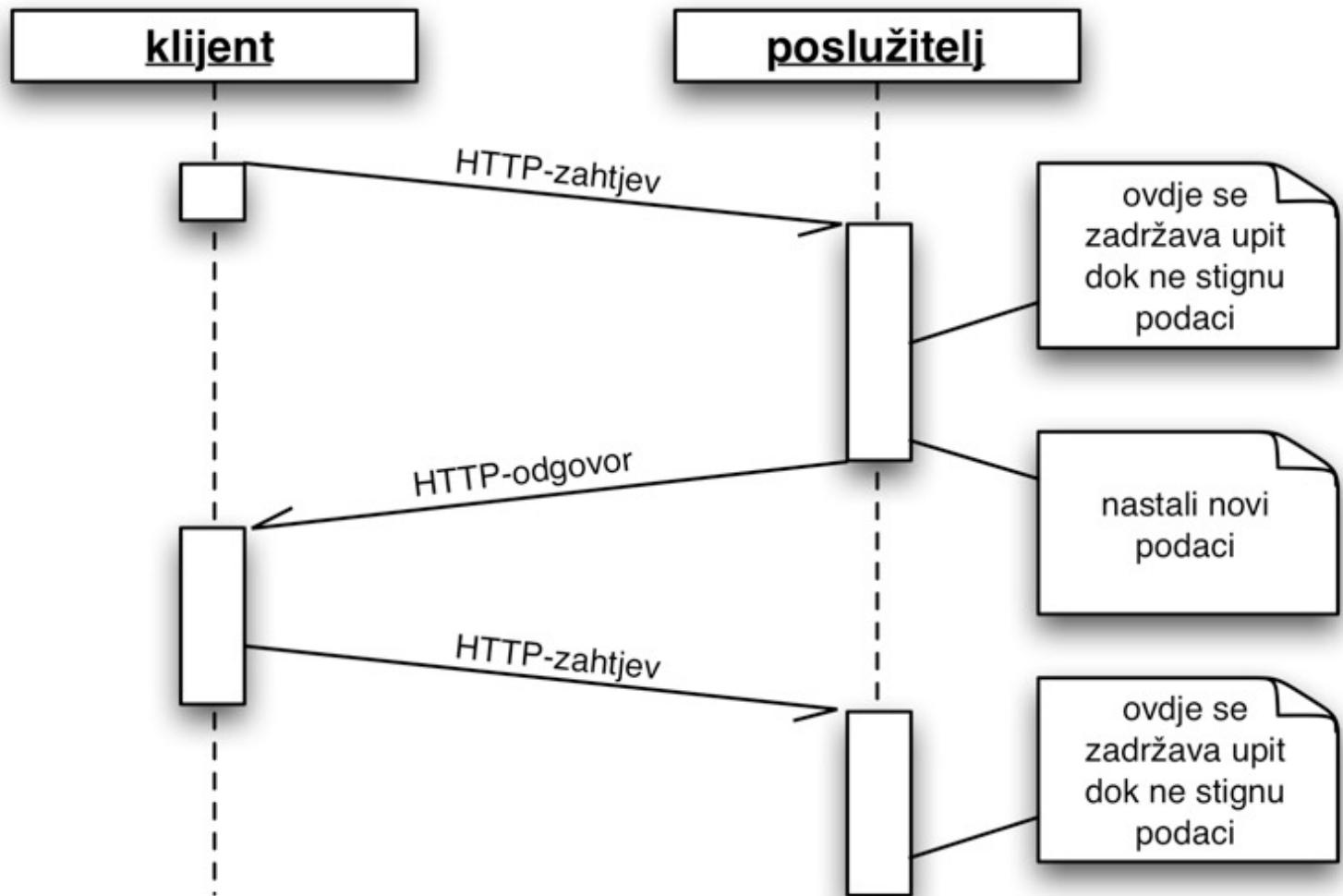
Slanje događaja klijentu iz preglednika (1) - prozivanje

- prozivanje poslužitelja (*poll*)



Slanje događaja klijentu iz preglednika (2) – dugo prozivanje

- dugo prozivanje poslužitelja
(long poll)



Slanje događaja klijentu iz preglednika (3) - SSE

- **Server-Sent Events** - SSE
- definirano standardom [Server-Sent Events](#) iz 2015.
- omogućuje slanje događaja klijentu kroz otvorenu konekciju
- klijent šalje zaglavljje Accept: text/event-stream
- poslužitelj kroz istu konekciju šalje tekst
 - svaka poruka je odijeljena praznim retkom
 - poruka ima polja koja su slična zaglavljima HTTP-a:
 - *event* - vrsta događaja (opcionalno)
 - *data* - podaci (obavezno)
 - *id* - identifikator paketa (opcionalno)
 - *retry* - vrijeme ponovnog spajanja u milisekundama (opcionalno)
- [članak - SSE, PHP primjer](#)

Primjer poslanih podataka:
data: Prva poruka

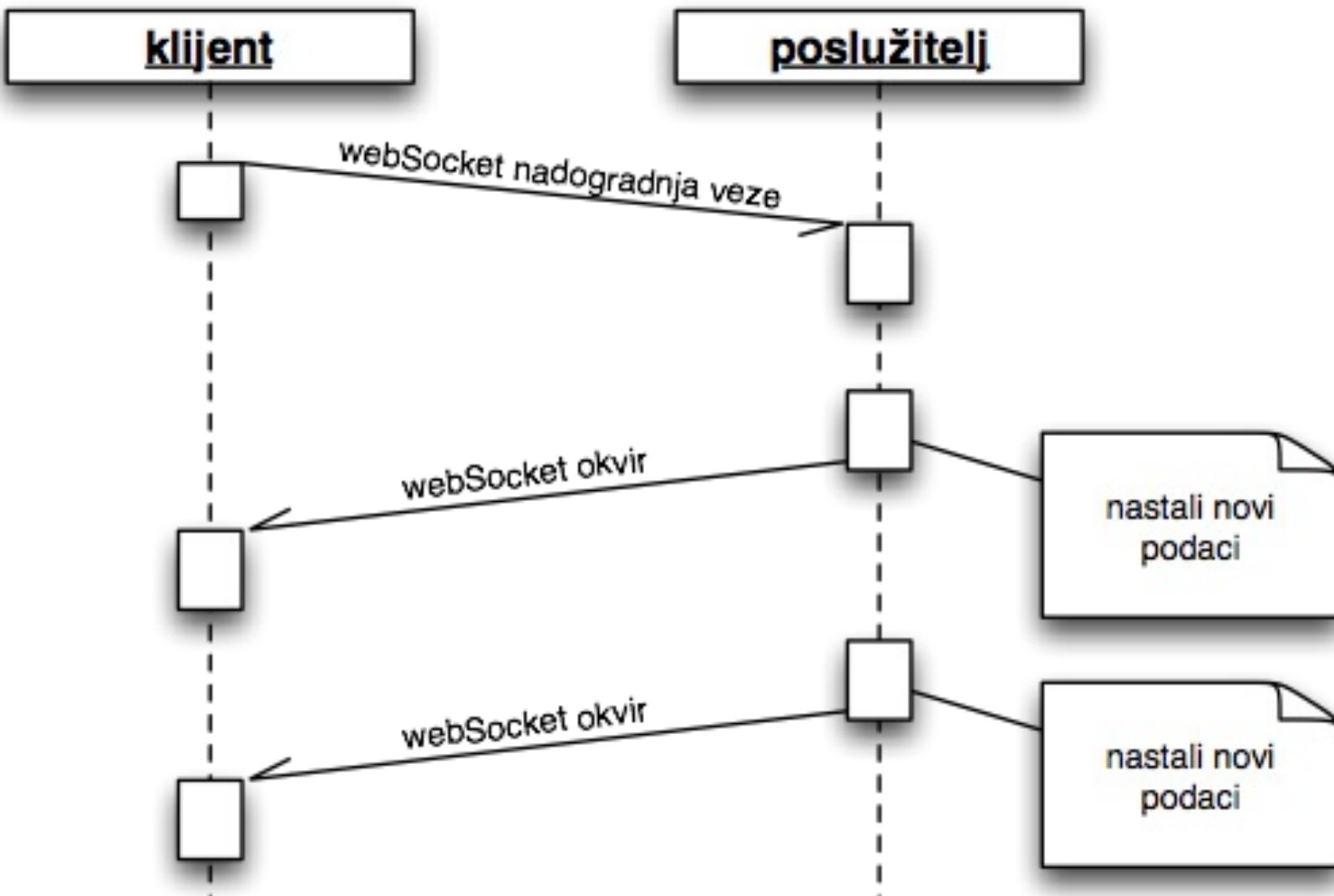
data: Druga poruka
data: 2. red 2. poruke

data: Treća poruka

Slanje događaja klijentu iz preglednika (4) - WS

- The **WebSocket (WS)** Protocol - RFC 6455 - prosinac 2011.
- omogućuje komunikaciju nalik TCP-ovskoj komunikaciji, ali iz internetskih preglednika
- podržavaju full-duplex, istovremeno je moguće primati i slati podatke
- inicijalna uspostava veze je kompatibilna s HTTP-om da bi isti poslužitelji mogli na istim vratima primati i HTTP-ovske zahtjeve i Web Socket zahtjeve
- nakon toga slijedi razmjena podataka u okrvirima
- URL shema: ws: (80) ili wss: (443) - ista vrata kao i HTTP
- koristi se: igre, kolaborativne aplikacije, financijske aplikacije, feed na društvenim mrežama, strujanje video sadržaja, ...

Web Socket - komunikacija

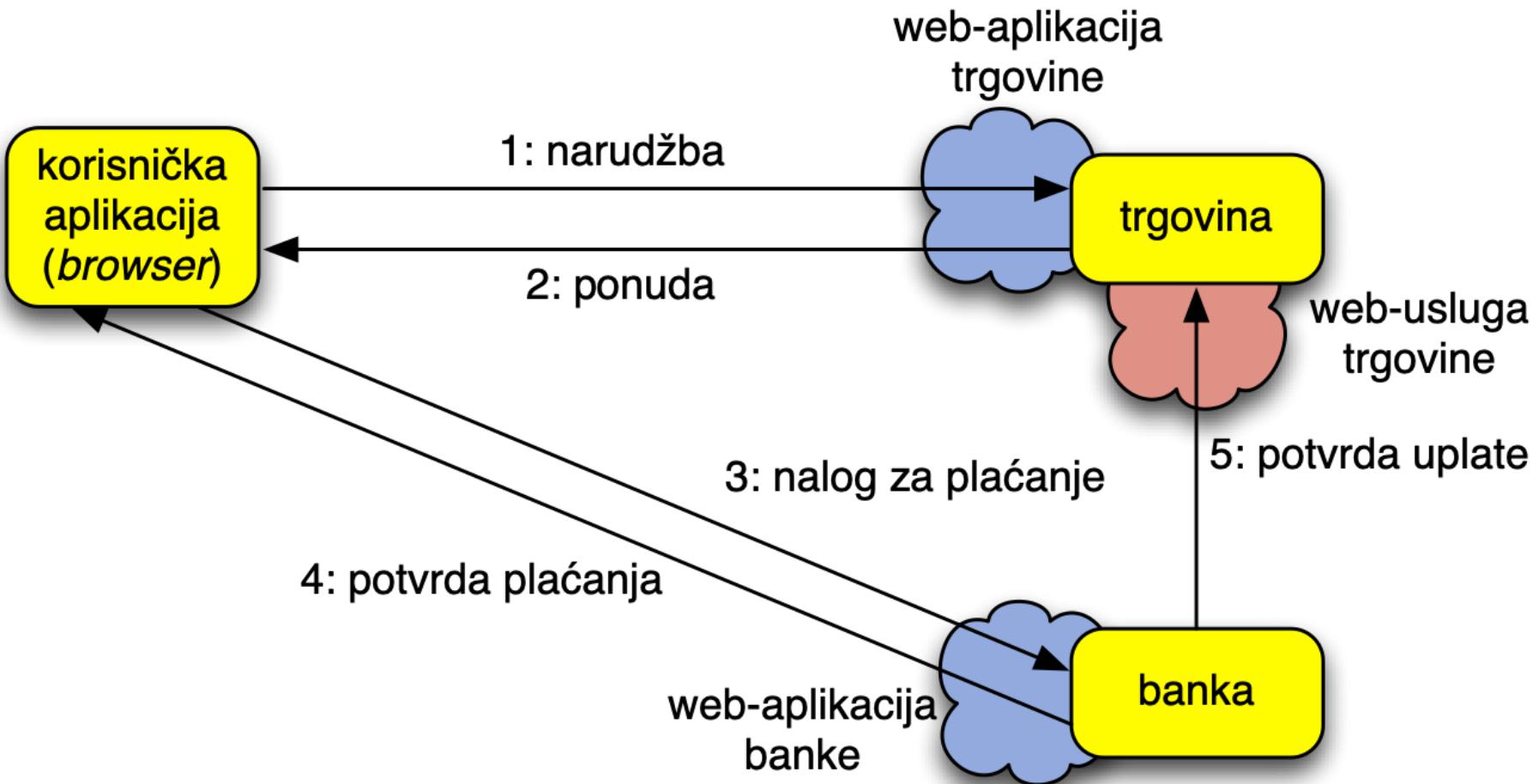


Web Socket - sadržaj

- 3 vrste sadržaja poruke:
 - tekstualni
 - binarni
 - ping-pong
 - ping-pong služi za provjeru dostupnosti druge strane
 - na poruku ping druga strana mora čim prije odgovoriti porukom pong jednakog sadržaja (maksimalna dužina sadržaja ping i pong poruka je 125 okteta)
 - na više primljenih ping poruka odgovara se samo jednom pong porukom, a samoinicijativnu pong poruku se ignorira
- podprotokoli
 - WebSocket ne definira aplikacijski protokol
 - Previše "posla" za programera kod parsiranja poruka
 - Možemo koristiti aplikacijske protokole preko WebSocketsa
 - dogovor prilikom pregovaranja
 - npr. STOMP, WAMP, MQTT, XMPP, ...

Web-usluge

Motivacija: scenarij kupovine



Uvod u web-usluge

- “obični” Web tipično koriste ljudi za pribavljanje informacija
- tehnologija Web Services (WS) omogućuje programima lakše korištenje Weba
 - umjesto RPC-a/Java RMI-a, CORBA-e, DCOM-a, itd.
- svojstva:
 - komunikacija se temelji na XML-u
 - koristi već postojeću internetsku infrastrukturu i protokole
 - usluge dostupne putem Interneta
 - omogućuje integraciju između različitih aplikacija
 - ne ovisi o programskom jeziku ili zatvorenoj tehnologiji jedne tvrtke
 - omogućuje otkrivanje usluga koje nude te aplikacije
 - usluge su slabo povezane
 - temelji se na industrijskim standardima

Što je web-usluga?

«Web-usluga je aplikacija identificirana URL-jem, čija se sučelja i veze mogu definirati, opisati i otkriti XML-om i koja podržava direktnu interakciju s drugim aplikacijama putem internetskih protokola koristeći poruke temeljene na XML-u»

(www.w3.org)

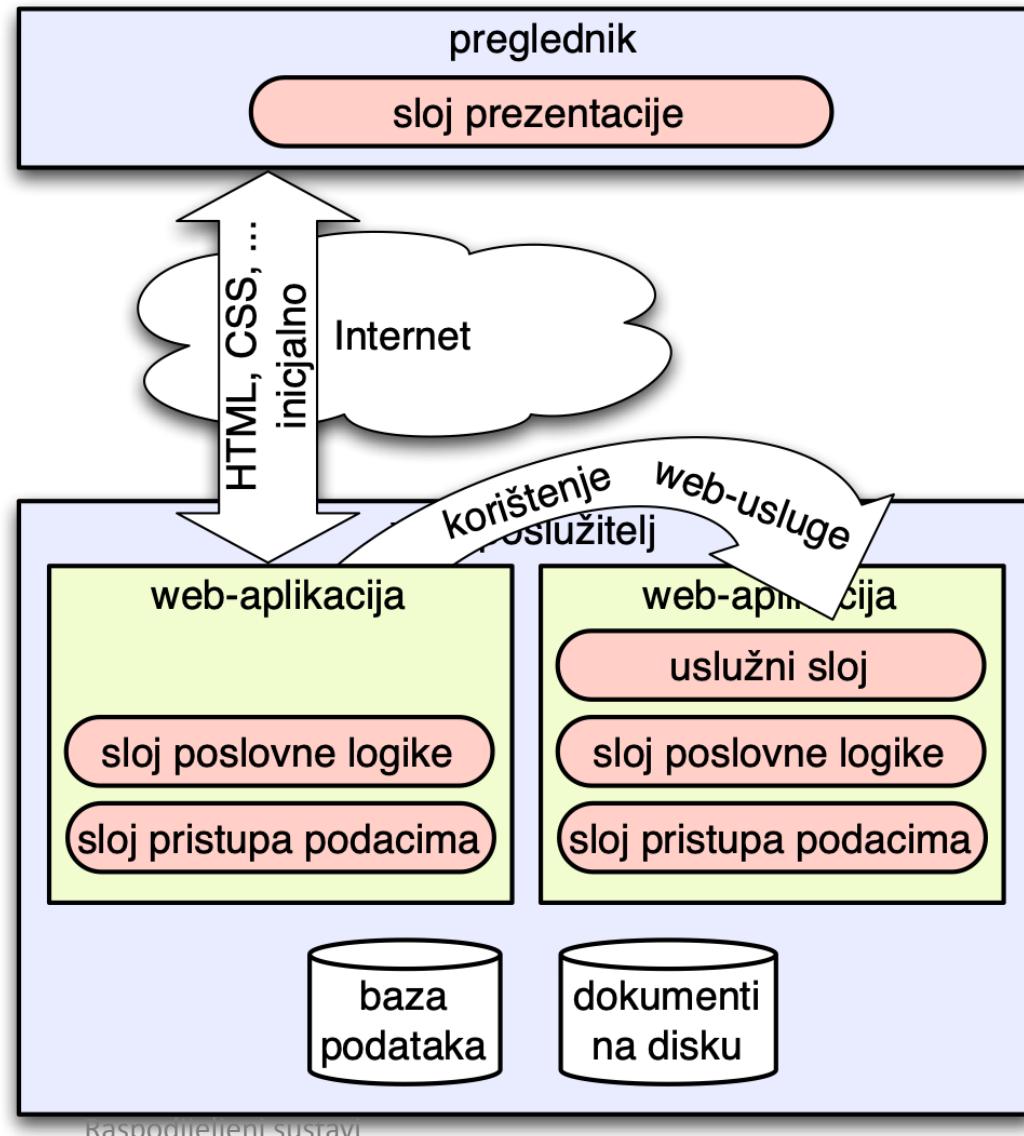
«Tehnologija web-usluga predstavlja novu vrstu web-aplikacija. To su samostalne, samoopisujuće aplikacije građene od modula, a koje se mogu objaviti, otkriti i pobuditi putem Weba. Web-usluge obavljaju funkcije koje mogu biti bilo što, od jednostavnih zahtjeva do komplikiranih poslovnih transakcija...»

(IBM's tutorial, www.xml.com)

- web-usluga je program koji:
 - je identificiran URI-jem
 - komunicira s klijentskim programima putem Weba
 - ima sučelje (API) opisano standardima web-usluga
 - omogućuje korištenje neovisno o platformama i programskim jezicima

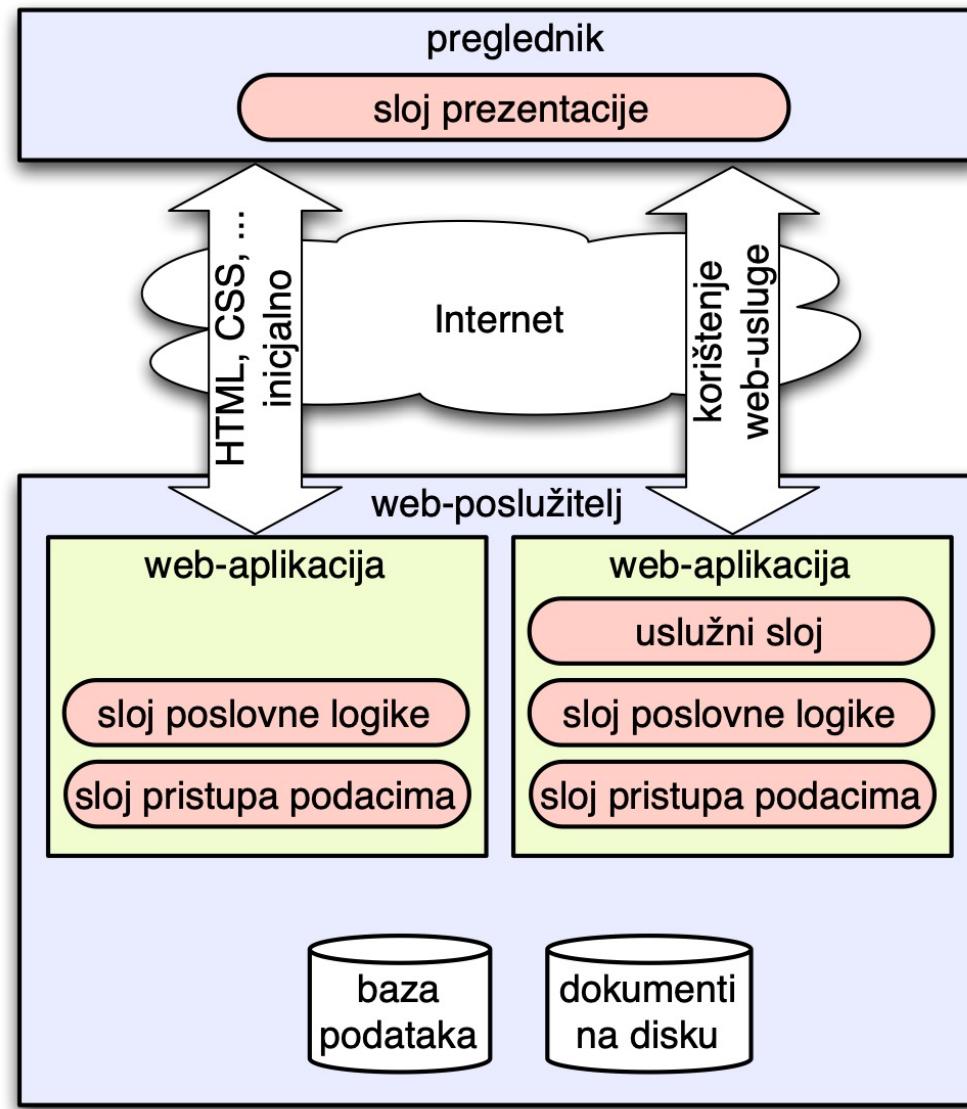
Web-aplikacija koja koristi web-usluge (1)

- web-aplikacija koristi web-uslugu
- web-usluga ne mora biti na istom računalu (u principu nije)



Web-aplikacija koja koristi web-usluge (2)

- klijent koristi web-uslugu (npr. klijent je aplikacija na pametnom telefonu ili preglednik koji koristi JavaScript za korištenje usluge)
- ovdje se obično radi o web-usluzi koja se temelji na REST-u



Vrste web-usluga

- **Usluge temeljene na udaljenim procedurama (RPC)**
 - rade kao poziv udaljenih procedura
 - koriste protokol SOAP i specifikaciju WSDL
 - na poslužitelju se poziva metoda u objektu
 - podaci su povezani s metodama koje se pozivaju
- **Usluge temeljene na dokumentima/porukama**
 - definiraju se poruke koje se razmjenjuju (XML Schema, WSDL)
 - koriste protokol SOAP i specifikaciju WSDL
 - nisu jako povezane
 - nije bitna implementacija, već samo podaci
- **Usluge temeljene na prijenosu prikaza stanja resursa (REST)**
 - RESTful (*Representational state transfer*) ili REST-usluge
 - temelje se na protokolu HTTP
 - koriste metode protokola: GET, PUT, DELETE, POST, PATCH

Tehnologije

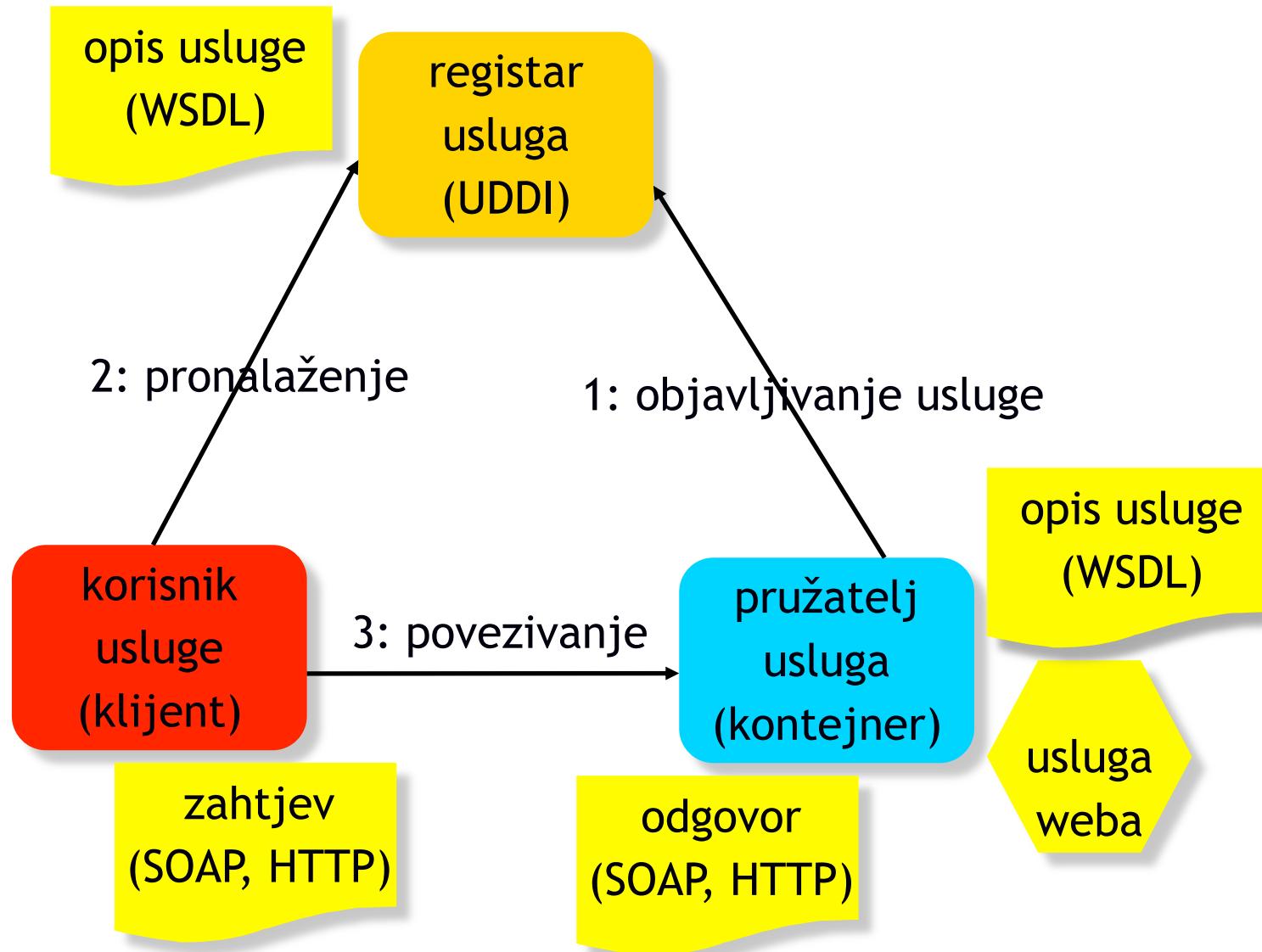
Tehnologije web-usluga

- WSDL (*Web Services Definition Language*)
 - opisuje uslugu
- SOAP (*Simple Object Access Protocol*)
 - format poruke
- UDDI (*Universal Description, Discovery and Integration*)
 - za otkrivanje usluga

Druga generacija web-usluga (WS-*)

- WS-Coordination
 - protokol za koordinaciju distribuiranih aplikacija
- WS-Transaction
- BPEL4WS (Business Process Execution Language)
 - jezik za formalnu specifikaciju poslovnih procesa i interakcijskih protokola
- WS-Security
 - sigurnosni protokol (TLS, integritet, privatnost, ...)
- WS-ReliableMessaging
- WS-Policy
- WS-Attachments
- WS-Addressing
 - adresiranje usluga i poruka

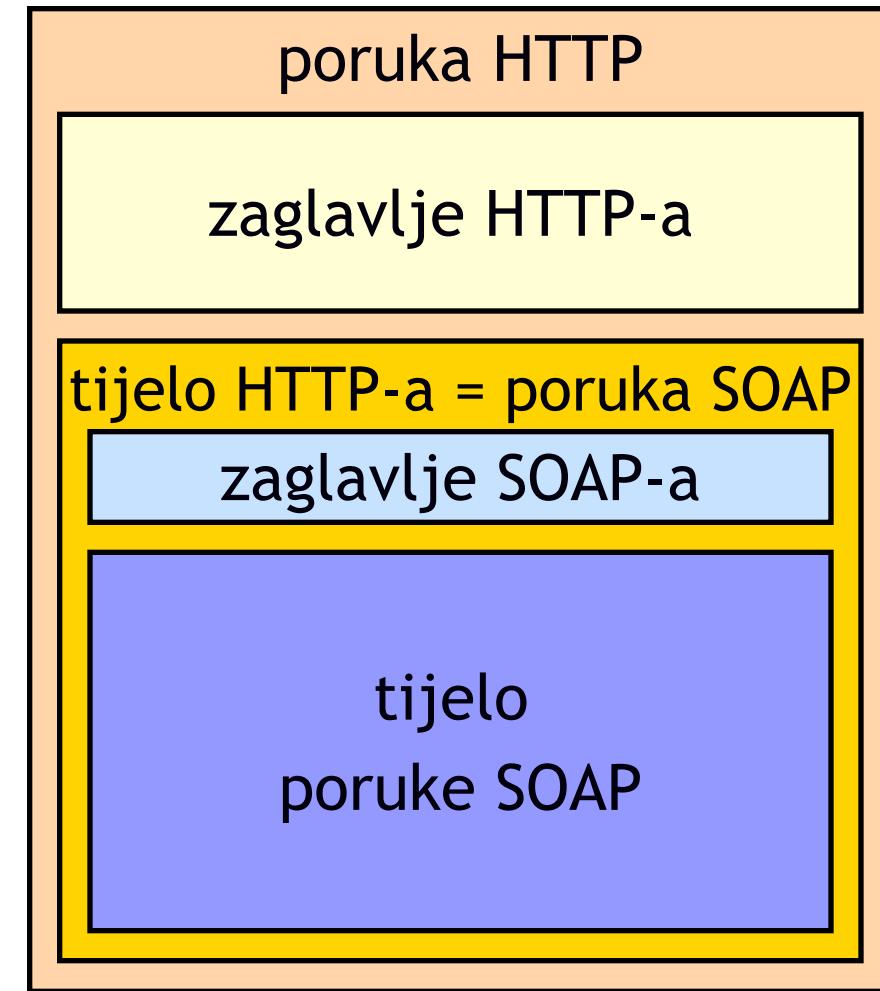
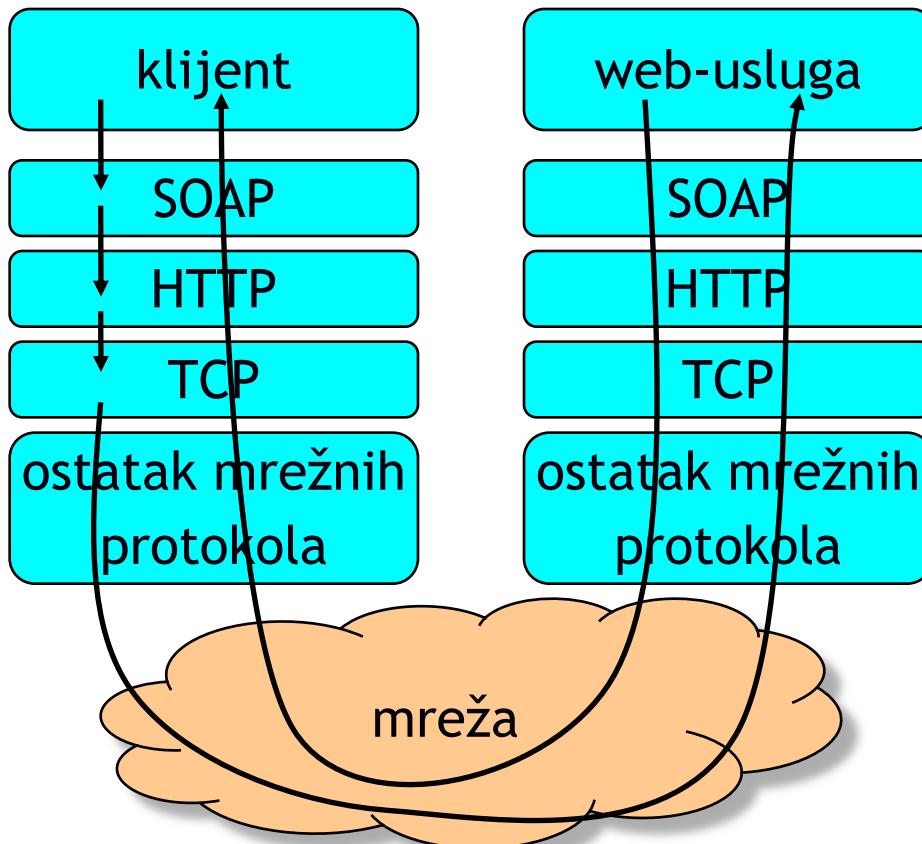
Arhitektura i korištenje



SOAP (*Simple Object Access Protocol*)

- omogućuje komunikaciju s web-uslugom
- dva osnovna načina rada:
 - poziv udaljenih procedura (RPC)
 - slično kao Corba, DCOM, Java RMI
 - služi za prijenos serijaliziranih parametara i rezultata
 - posljedica:
 - dobro definirana sučelja i tipovi podataka
 - prilagodni kod može biti generiran automatski
 - razmjena dokumenata/poruka
 - sadrži XML dokument
 - fleksibilnije u odnosu na RPC
 - XSLT i XQuery se koristi za prilagodbu dokumenata
 - lakše se koriste uzorci razmjene poruka
 - polako se uključuje semantički web (ontologije, procesiranje i sl.)
- specifikacija: <http://www.w3.org/TR/soap/>

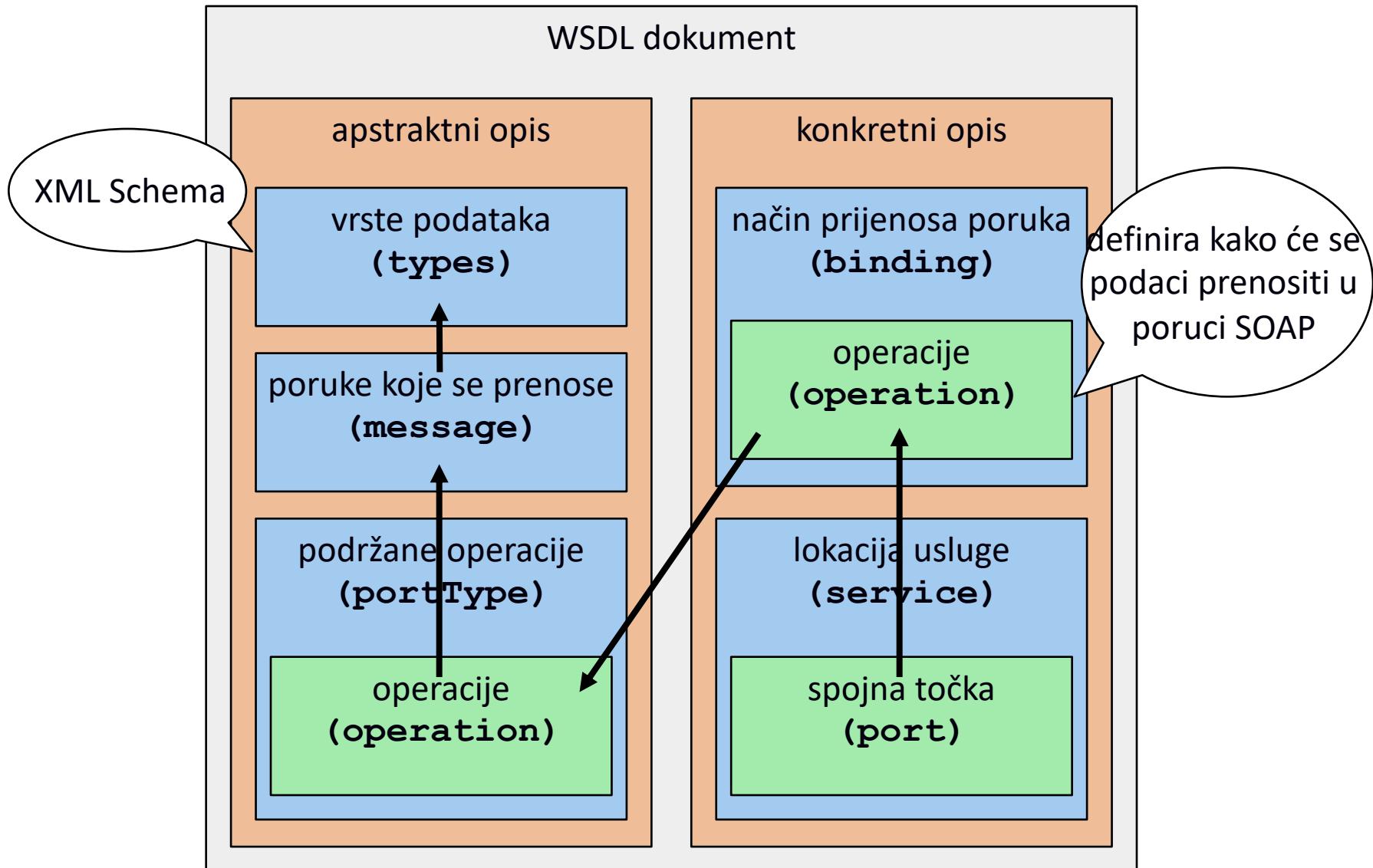
Prijenos poruke SOAP



WSDL (Web Services Description Language)

- jezik za opis web-usluga
- web-usluga je opisana skupom komunikacijskih krajnjih točaka (*ports*)
- krajnja točka se sastoji od dva dijela:
 - apstraktne definicije operacija i poruka
 - specifikacije mrežnog protokola i pojedine krajnje točke te formata poruke
- opisuje komunikacijske detalje između klijenta i usluge
 - strojevi (računala) mogu pročitati WSDL
 - mogu pozvati uslugu definiranu WSDL-om
- jedan je od mehanizama koji omogućuje da se usluga može otkriti pomoću registra
- specifikacija: <https://www.w3.org/TR/wsdl>

Struktura WSDL-a



Elementi WSDL-a (1)

- **definitions**
 - osnovni element WSDL dokumenta
- **types**
 - opis podataka pomoću XML Scheme
 - nepotreban ako se koriste osnovni podaci iz Scheme
- **message**
 - opis jednosmjerne poruke
 - definira ime poruke
 - poruka se sastoji od dijelova (**part**)
 - svaki dio se referencira na vrste podataka iz dijela **types**
- **portType**
 - definira operacije
 - svaka operacija se sastoji od poruka (reference na poruke)
 - poruke koji mogu biti:
 - parametri (ulazne poruke)
 - vrijednosti koje se vraćaju (rezultati)

Elementi WSDL-a (2)

- **binding**
 - definira kako će se poruke iz operacije prenositi
 - definira koje transportne protokole će koristiti (HTTP GET, HTTP POST, SOAP, SMTP)
 - stil definira vrstu čitave poruke:
 - **rpc** - zahtjev će imati omotač u kojem će pisati naziv funkcije koja se poziva
 - **document** - zahtjev i odgovori će imati “obične” XML dokumente
 - poruke mogu koristiti dvije vrste pakiranja poruka:
 - **literal** - onako kako definira Schema
 - **encoded** - kodirano pravilima u SOAP-u
- **services**
 - definira lokaciju usluge (URL)
- **import**
 - koristi se za uključivanje drugih WSDL ili XML Schema dokumenata

Pronalaženje usluga

- UDDI (*Universal Description, Discovery and Integration*)
- osigurava platformu za otkrivanje usluga na Internetu
- sastoji se od 3 dijela:
 - imenik (*White pages*)
 - adrese, kontakti i identifikatori
 - poslovni imenik (*Yellow pages*)
 - kategorizacija područja, usluga i proizvoda te lokacije
 - tehničke informacije (*Green pages*)
 - sadrži tehničke informacije o uslugama
- nema centralnog registra
- više informacija: <http://uddi.xml.org/uddi-org>
- standard: <https://www.oasis-open.org/standards#uddiv3.0.2>

Web-usluge temeljene na RPC-u

- napravimo novi projekt (npr. NetBeans)
- stvorimo novu web-uslugu i stvorimo operacije/metode kao u primjeru dolje
- alat sam generira WSDL i iz WDSL-a klijenta

```
@WebService()  
public class MyService {  
    @WebMethod(operationName = "add")  
    public int add(@WebParam(name = "x") int x,  
                  @WebParam(name = "y") int y)  
    {  
        return x+y;  
    }  
  
    @WebMethod(operationName = "toLowerCase")  
    public String toLowerCase(  
        @WebParam(name = "text")  
        String text)  
    {  
        return text.toLowerCase();  
    }  
}
```

Usluge temeljene na dokumentima

- Usluge temeljene na dokumentima
 - razmjenjuju se dokumenti
 - dogovor o dokumentima (XML Schema)
 - obično su asinkrone
 - parametri kod povezivanja u WSDL-u su: Document-literal
- Postupak izrade:
 1. definiranje XML Scheme dokumenata
 - obično u posebnoj datoteci
 2. definiranje WSDL-a
 - uključuje XML Scheme
 3. iz WSDL-a se mogu generirati:
 - kostur usluge
 - primer klijenta

Web-usluge temeljene prijenosu prikaza stanja resursa (REST)

- REST (*Representational State Transfer*)
- pojmovi: *The REST Way* ili *RESTful services*
- pojam je skovao Roy Fielding u svojoj doktorskoj disertaciji
- nije standard već arhitekturni stil
- sve se temelji na resursima koji su predstavljeni URL-ovima:
 - `http://localhost:8080/RassusRest/rest/persons` - popis svih korisnika
 - `http://localhost:8080/RassusRest/rest/persons/1` - korisnik s identifikatorom 1
- koristi protokol: HTTP (GET, POST, PUT, DELETE, PATCH)
- koristi podatke: JSON, XML, sirovi (npr. za slike, video)
- bez stanja (*stateless*), priručni spremnik (*cache*)

Format JSON (Javascript Object Notation)

- podatak - par: **ime/vrijednost**

"firstName": "Ivana"

- vrijednosti mogu biti:

- broj, *string*, *boolean* (`true`, `false`), polje, objekt, null

- objekt su parovi u vitičastim zagradama, npr.:

{ "firstName" : "Ivana", "lastName" : "Podnar Žarko" }

- ime u objektu mora biti jedinstveno

- polje su vrijednosti u uglatim zagradama, npr.:

```
[ { "name": "Ignac Lovrek", ... },  
  { "name": "Ivana Podnar Žarko", ... },  
  ... ]
```

Primjer podataka u JSON-u i XML-u

- JSON:

```
{  
  "firstName" : "Ivana",  
  "lastName" : "Podnar Žarko",  
  "room" : "C7-12",  
  "phone" : "261",  
  "_links" : {  
    "self" : {  
      "href" : "http://localhost:8080/persons/2  
    }  
  }  
}
```

- XML:

```
<person>  
  <firstName>Ivana</firstName>  
  <lastName>Podnar Žarko</lastName>  
  <room>C7-12</room>  
  <phone>261</phone>  
  <links>  
    <link>  
      <rel>self</rel>  
      <href>http://localhost:8080/persons/2</href>  
    </link>  
  </links>  
</person>
```

Usporedba JSON-a i XML-a

- JSON
 - za
 - format
 - ugrađen u preglednike
 - protiv
 - nema ugrađene poveznice (koristi se kao tekst)
 - nema mogućnosti definiranja sheme
 - ograničen skup vrsta podataka
 - problem binarnih podataka
- XML
 - za
 - mogu se zapisati složene strukture podataka
 - ima standardne poveznice
 - dobar skup alata za transformaciju i obradu
 - protiv
 - problem binarnih podataka
 - loš za strukture bez redoslijeda
 - ograničenja DTD-a
 - postoji puno standarda koji ga komplikiraju
 - brzina procesiranja

Primjer zahtjeva i odgovora (1)

HTTP GET <http://localhost:8080/persons>

```
[  
  {  
    "id" : 1,  
    "name" : "Ignac Lovrek",  
  },  
  {  
    "id" : 2,  
    "name" : "Ivana Podnar Žarko",  
  },  
  {  
    "id" : 3,  
    "name" : "Mario Kušek",  
  },  
  {  
    "id" : 4,  
    "name" : "Krešimir Pripužić",  
  },  
  ...  
]
```

Primjer zahtjeva i odgovora (2)

HTTP GET <http://localhost:8080/persons/2>

```
{  
    "firstName" : "Ivana",  
    "lastName" : "Podnar Žarko",  
    "phone" : "261",  
    "room" : "C7-12"  
}
```

Svojstva metoda protokola HTTP

- Svojstva:
 - Sigurna (*safe*) - bez posljedica za podatke
 - *Idempotentna* - može se izvršavati više puta
 - Može se privremeno spremiti odgovor (*cachable*)
- Metode:
 - GET - sigurna, idempotentna, može se privremeno spremiti
 - PUT - idempotentna
 - DELETE - idempotentna
 - HEAD - sigurna, idempotentna
 - POST - ništa
 - PATCH - ništa, ali se može napraviti da je idempotentna (za više vidi [ovdje](#)) što je česti slučaj u praksi

Tipično korištenje usluga REST

- za *Create, Read, Update and Delete* (CRUD)

HTTP	CRUD
POST	Create, (Overwrite/Replace)
GET	Read
PUT	Update, (Create, Delete)
DELETE	Delete
PATCH	Partial update

- primjer gotove usluge: *Twitter*

Dizajniranje usluge

- paziti na mrežu:
 - što i koliko se podataka prenosi
 - koliko zahtjeva i odgovora imamo da bismo nešto prikazali na klijentu
- napraviti URL za svaki resurs
- definirati metode za svaki resurs
- stavljati poveznice u resursima
 - ne vraćati čitavu strukturu
- specificirati format za svaki resurs
- povezati usluge tako da sve kreće preko jednog URL-a
 - HATEOAS - *Hypermedia as the Engine of Application State* (3. razina zrelosti web-usluga)
- preko vrsta medija dogovorati verzije usluga (API-ja)

Zadatak

- Napraviti uslugu koja služi za evidenciju plaćenih računa
- U sustavu imamo osobe
- Svaka osoba ima osobne podatke
- Osobe se mogu stvoriti, obrisati i promijeniti im podatke
- Svaka osoba ima račune koje je platila za pojedini mjesec
- Kada je neki račun unesen više ga nije moguće mijenjati niti obrisati
- Rješenje projekta i klijenata koji ga koriste se nalazi na:
<https://gitlab.tel.fer.hr/spring>

Tablica dizajna usluge

resurs	metode	šalje	svrha
/persons	POST	osoba	stvara novu osobu
	GET		vraća popis osoba
/persons/{id}	GET		vraća osobu s ID-om
	DELETE		briše osobu (brišu se i svi računi te osobe)
	PUT	osoba	mijenja podatke o osobi
	PATCH	dio osobe	mijenja samo podatka koji su poslati
/persons/{id}/bills	POST	račun	stvara novi račun za osobu s ID-om
	GET		vraća popis računa te osobe
/bills/{bid}	GET		vraća račun s ID-om bid

Primjer upita i odgovora - stvaranje resursa

- POST /persons

- sadržaj zahtjeva:

```
{  
    "firstName": "Jura",  
    "lastName": "Jurić",  
    "address": "Unska 3, 1000 Zagreb",  
    "phone": "+385 1 6129 999",  
    "email": "jura.juric@fer.hr"  
}
```

- odgovor:

201 Created

Location: <http://localhost:8080/persons/4>

Primjer upita i odgovora - lista osoba

- GET /persons
- odgovor:

200 OK

```
[  
  {  
    "id" : 1,  
    "name" : "Ignac Lovrek",  
  },  
  {  
    "id" : 2,  
    "name" : "Ivana Podnar Žarko",  
  },  
  {  
    "id" : 3,  
    "name" : "Mario Kušek",  
  },  
  {  
    "id" : 4,  
    "name" : "Krešimir Pripužić",  
  },  
  ...  
]
```

Primjer upita i odgovora - dohvaćanje resursa

- GET /persons/2

- odgovor:

```
{  
    "firstName" : "Ivana",  
    "lastName" : "Podnar Žarko",  
    "address": "Unska 3, 10000 Zagreb",  
    "phone": "+385 1 6129 999",  
    "email: "ivana.podnar@fer.hr"  
}
```

Primjer upita i odgovora - dohvaćanje nepostojećeg

- GET /persons/100
- odgovor:

404 Not Found

Primjer upita i odgovora - promjena resursa

- PUT /persons/2
 - {
 - "firstName" : "Ivana",
 - "lastName" : "Podnar Žarko",
 - "address": "Unska 3, 10000 Zagreb",
 - "phone" : "+385 1 6129 761",
 - "email": "ivana.podnar@fer.hr"
 - }

- odgovor:

204 No Content

Primjer upita i odgovora - stvaranje resursa

- PUT /persons/200

```
{  
    "firstName" : "Ana",  
    "lastName" : "Anić",  
    "address": "Unska 3, 10000 Zagreb",  
    "phone": "+385 1 6129 999",  
    "email: "ana.anic@fer.hr"  
}
```

200 se ignorira

- odgovor:

201 Created

Location: <http://localhost:8080/persons/5>

Primjer upita i odgovora - promjena dijela resursa

- PATCH /persons/2

```
{  
    "phone" : "+385 1 6129 761"  
}
```

- odgovor:

200 OK

```
{  
    "firstName" : "Ivana",  
    "lastName" : "Podnar Žarko",  
    "address": "Unska 3, 10000 Zagreb",  
    "phone" : "+385 1 6129 761",  
    "email: "ivana.podnar@fer.hr"  
}
```

- ovo je idempotentno

Primjer upita i odgovora - promjena nepostojećeg resursa

- PATCH /persons/200
- ```
{
 "phone" : "+385 1 6129 761"
}
```

- odgovor:

404 Not Found

# Primjer upita i odgovora - osoba koja postoji

- DELETE /persons/3
- odgovor:

204 No Content

# Primjer upita i odgovora - osoba koja ne postoji

- DELETE /persons/300
- odgovor:

404 Not Found

# Primjer upita i odgovora - stvaranje resursa

- POST /persons/2/bills

- sadržaj zahtjeva:

```
{
 "month": 3,
 "year": 2019,
 "amount": 250
}
```

- odgovor:

201 Created

Location: <http://localhost:8080/bills/1>

# Primjer upita i odgovora - lista računa

- GET /persons/2/bills
- odgovor:

200 OK

```
[
 {
 "id": 1,
 "peroid": "2019-3"
 },
 {
 "id": 3,
 "peroid": "2019-4"
 },
 ...
]
```

# Primjer upita i odgovora - jedan račun

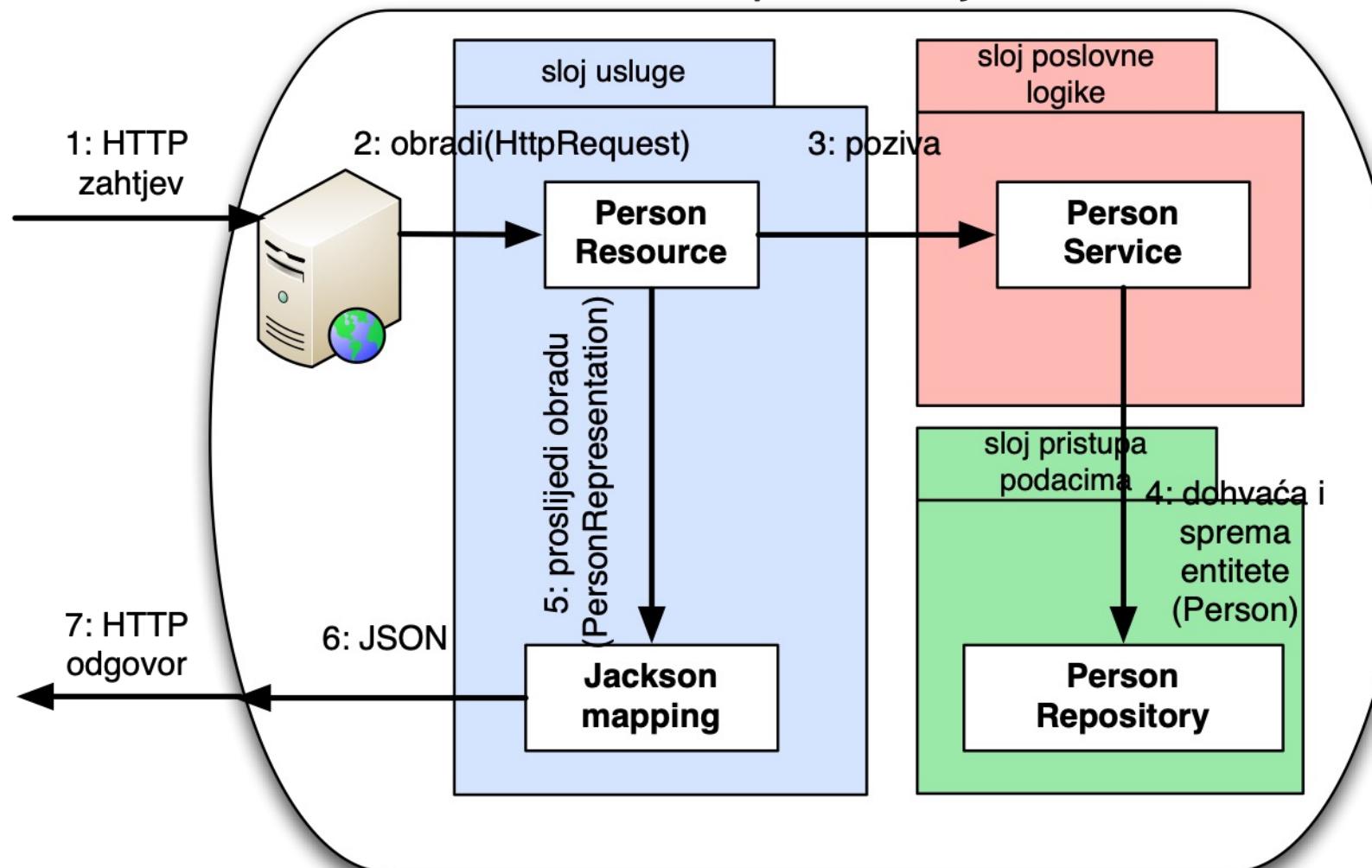
- GET /bills/1
- odgovor:

200 OK

```
{
 "month": 3,
 "year": 2019,
 "amount": 250
}
```

# Arhitektura web-aplikacije (REST)

web-poslužitelj



# Spring Framework - <http://spring.io>



- prva verzija 2003. koju je napravio Rod Johnson
- radni okvir za lakši razvoj aplikacija
- bavi se konfiguracijom objekata u sustavu (*IoC – inversion of control*)
  - upravlja poslovnim objektima kao običnim objektima (*POJO – plain old java objects*)
  - brine se za kreiranje objekata
  - povezuje kreirane objekte (*IoC, wiring up, DI - dependency injection*)
  - upravlja njihovim životnim ciklusom
- složene veze između objekata se definiraju u XML-u ili pomoću bilješki (*annotation*)
- odvaja poslovnu logiku od mehanizama za ispravan rad sustava (transakcije, logiranje, ...)
- vrlo je složen za početnika jer ima puno stvari ugrađeno

# Spring Boot (trenutna verzija 2.7.4)

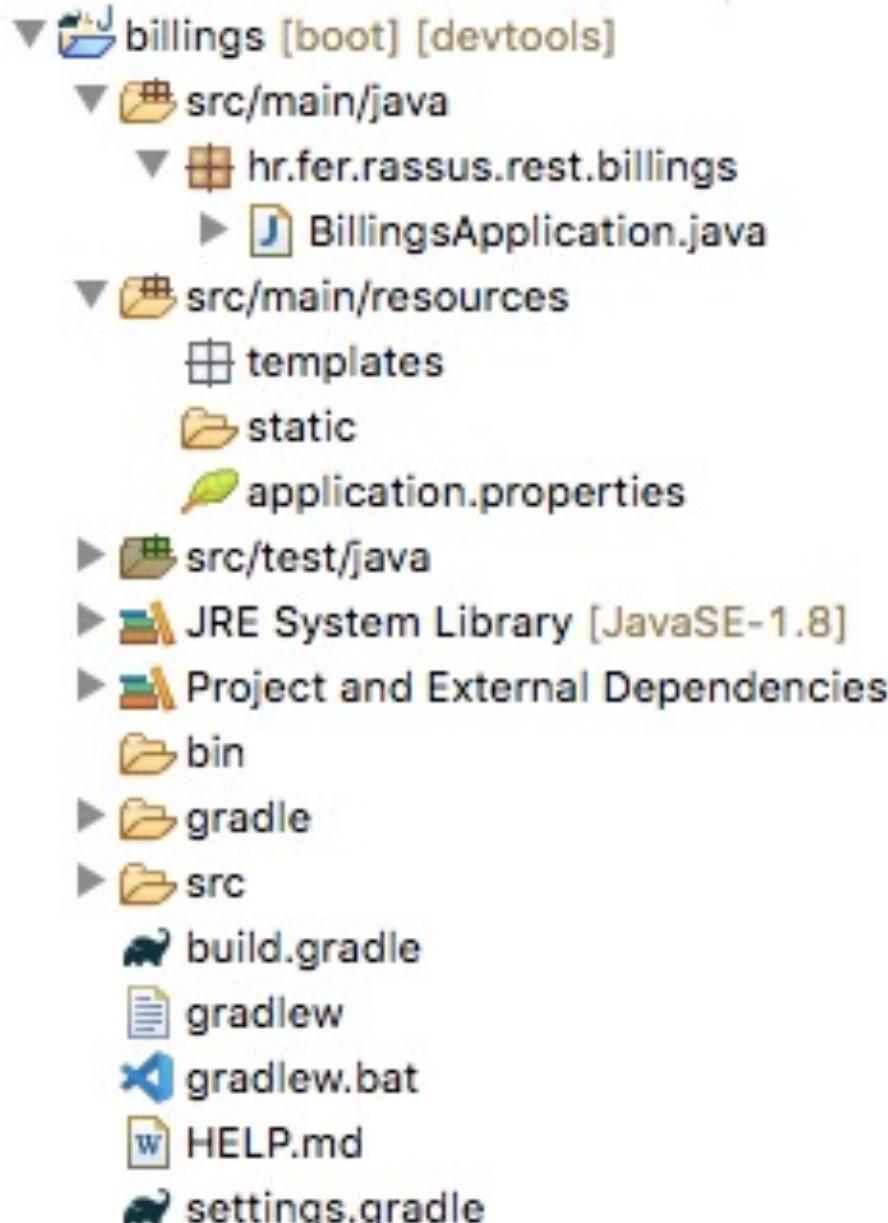
- jedan od podprojekata Springa
  - <http://projects.spring.io/spring-boot/>
- pojednostavljuje korištenje Springa, pogotovo stvaranje novog projekta
- podržava:
  - automatsku konfiguraciju
  - pretraživanja klasa na putu (*path*)
- aplikacija ima manje koda
- kod web-aplikacija - omogućuje izradu samostalnih aplikacija
  - web-poslužitelj zapakiran u jar
  - jednostavnije instaliranje
  - aplikacija spremna za produkcijsku okolinu
- [primjeri projekata](#), [Spring framework guru tutorials](#)
- **Video materijali:**
  - Spring predavanja za prediplomski projekt - [youtube](#),
  - Vještina RUAZOSA - meduza ([1. dio](#), [2. dio](#))

# Stvaranje Spring projekta

- otvoriti stranicu <https://start.spring.io>
- klik na "Switch to the full version"
- popuniti:
  - Group: hr.fer.rassus.rest
  - Artifact: billings
  - Name: billings
  - Packaging: Jar
  - Java Version: 17 (može i novija verzija)
  - Language: Java
- odabratи: Web, HATEOAS, DevTools, (Lombok?)
- klik na Generate Project - napravi zip koji preglednik skine
- trebamo u IDE-u otvoriti projekt u koji smo raspakirali

# Struktura projekta

- pogledati:
  - BillingsApplication
    - klasa koja se pokreće
  - build.gradle
    - skripta za "građenje"
  - application.properties
    - vanjska konfiguracija



# klasa LecturesApplication

```
@SpringBootApplication
public class BillingsApplication {

 // metoda koja sve pokreće
 public static void main(String[] args) {
 SpringApplication.run(BillingsApplication.class, args);
 }
}
```

# Beanovi i DI (*Dependency Injection*)

- Objekte koje stvara i s kojima upravlja Springov kontejner zovu se *Springovi beanovi* (skraćeno samo *bean*)
- DI - *dependency injection*
  - objekti definiraju ovisnosti o drugim objektima
  - mora imati ili: atribute, setere ili konstruktor kroz koji se ovisnosti postavljaju
  - kontejner onda ubacuje ovisnosti kada stvara baenove
  - nepotrebne posebne metode za instanciranje i postavljanje ovisnih objekata da bi objekti normalno funkcionali
- doseg beanova - `@Scope("tip")`
  - **singleton (podrazumijevano)** - jedna instanca u JVM-u
  - prototip - novi objekt svaki put kada se zahtjeva bean
  - zahtjev (*request*) - kod svakog zahtjeva se stvara novi bean
  - sjednica (*session*) - za svakog korisnika jedan bean
  - globalna sjednica - koristi se kod portleta

# Ubrizgavanje beanova

1. preko atributa (*field*) - ne preporuča se

```
@Autowired
private NekiBean b;
```

2. preko konstruktora

```
private NekiBean b;

public XApplication(NekiBean b) {
 this.b = b;
}
```

3. preko setera

```
private NekiBean b;

@Autowired
public void setB(NekiBean b) {
 this.b = b;
}
```

# Definiranje beanova

- dva načina:
  - u konfiguracijskoj klasi (npr. SimpleApplication) definirati metodu koja je označena da vraća bean (@Bean)
  - označiti klasu s @Component
    - mehanizam pretraživanja puta može pronaći takvu klasu
    - podvrste:
      - @Controller - predstavlja kontroler u Web MVC-u
      - @Service - predstavlja namjeru da je to usluga
      - @Repository - predstavlja komponentu koja pristupa podacima
      - @RestController - predstavlja kontroler u Web MVC-u koja vraća podatke koji se serializiraju u JSON ili XML

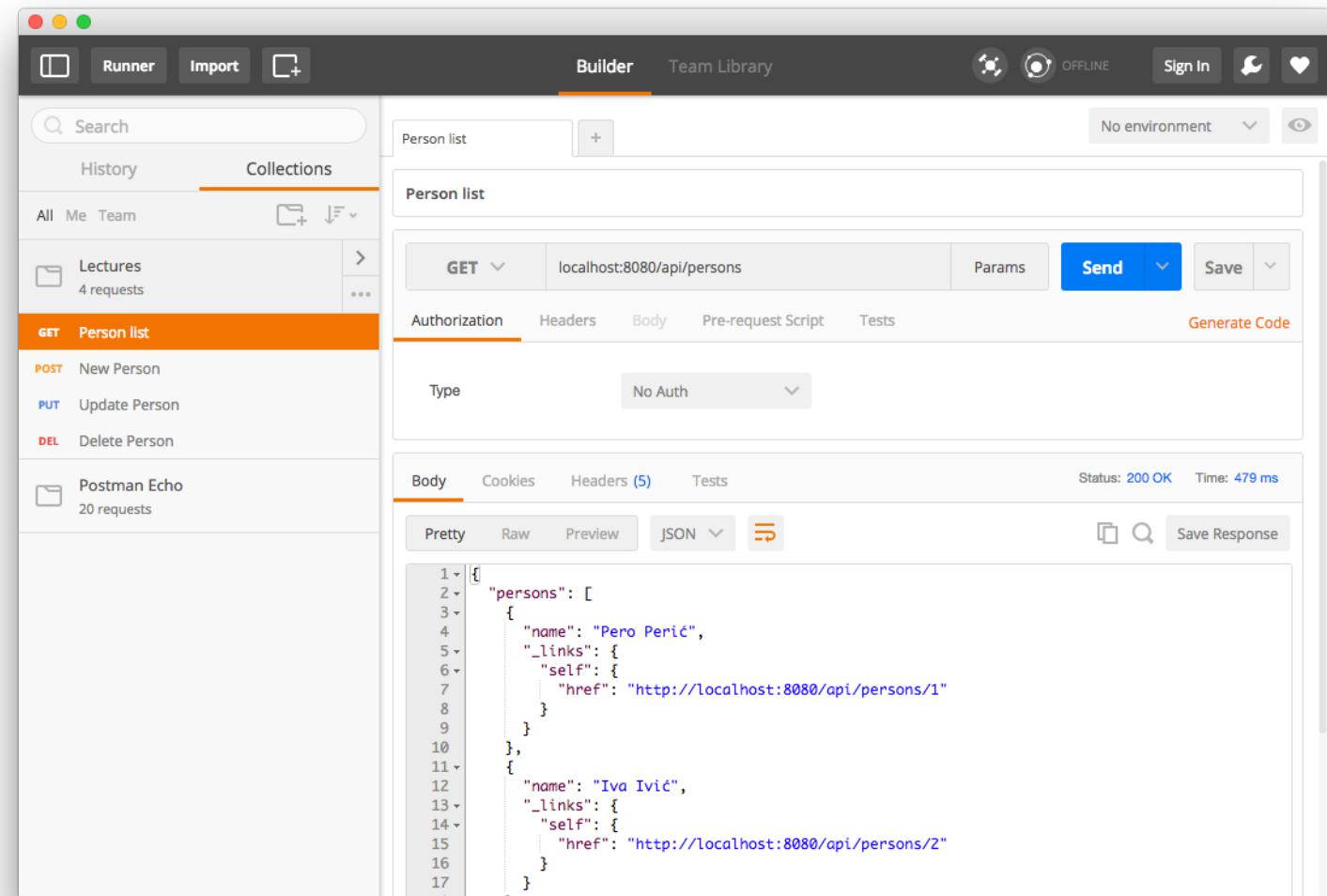
# REST kontroler - *bean*

```
@RestController
public class PersonResourceController {

 @GetMapping("/persons")
 public String getPersonsList() {
 return "Pero i Ana";
 }
}
```

# Primjer dohvaćanja

- Chrome extensions  
(<https://chrome.google.com/webstore/search/rest>):
  - Advanced Rest Client ili
  - Postman
- u terminalu:
  - curl
  - HTTPie



# REST kontroler - *bean* (2)

```
@RestController
public class PersonResourceController {

 @GetMapping("/persons")
 public String getPersonsList() {
 return "Pero i Ana";
 }

 @GetMapping("/persons/{id}")
 public String getPerson(@PathParam("id") String id) {
 return "Ana";
 }
}
```

# REST kontroler - *bean* (3)

```
@RestController
public class PersonResourceController {

 @GetMapping("/persons")
 public String getPersonsList() {
 return "Pero i Ana";
 }

 @GetMapping("/persons/{id}")
 public PersonRepresentation getPerson(@PathParam("id") String id) {
 return new PersonRepresentation("Ana", "Anić",
 "Unska 3, 10000 Zagreb", "+385 1 6129 999", "ana.anic@fer.hr");
 }
}

public class PersonRepresentation {
 private String firstName, lastName, address, phone, email;

 // getters, setters, konstruktori (prazan, atributi), toString
}
```

# PersonService

```
@Service
public class PersonService {

 private int pidCounter = 0;
 private Map<Integer, Person> persons = new HashMap<>();

 public Collection<Person> getPersons() {
 return persons.values();
 }
}
```

# PersonResourceController

```
@RestController
public class PersonResourceController {

 private PersonService personService;

 public PersonResourceController(PersonService personService) {
 this.personService = personService;
 }

 @GetMapping("/persons")
 public Collection<ShortPersonRepresentation> getPersonsList() {
 return personService.getPersons().stream()
 .map(p -> PersonAssembler.toShortPersonRepresentation(p))
 .collect(Collectors.toList());
 }

 ...
}
```

# ShortPersonRepresentation

```
public class ShortPersonRepresentation {
 private int id;
 private String name;

 // setters/getters/konstruktori
}
```

# PersonAssembler

```
public class PersonAssembler {

 public static ShortPersonRepresentation toShortPersonRepresentation(
 Person person) {
 return new ShortPersonRepresentation(
 person.getId(),
 person.getFirstName() + " " + person.getLastName());
 }

 public static PersonRepresentation toPersonRepresentation(Person person)
 { ... }

 public static Person toPerson(PersonRepresentation personRepresentation)
 { ... }

 public static Person toPerson(int id,
 PersonRepresentation personRepresentation) { ... }

 public static void updatePersonForNotNullValues(Person person,
 PersonRepresentation personRepresentation) { ... }
}
```

# Dohvaćanje jedne osobe

```
@RestController
public class PersonResourceController {
 ...
 @GetMapping("/persons/{id}")
 public ResponseEntity<PersonRepresentation> getPerson(
 @PathVariable("id") Integer id)
 {
 Person person = personService.getPerson(id);
 if(person != null) {
 return ResponseEntity.ok(
 PersonAssembler.toPersonRepresentation(person));
 }
 return ResponseEntity.notFound().build();
 }
}
```

# Kreiranje nove osobe

```
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;
...
@RestController
@RequestMapping("/persons")
public class PersonResourceController {
...
 @PostMapping()
 public ResponseEntity<?> newPerson(
 @RequestBody PersonRepresentation personRepresentation) {
 Person person = PersonAssembler.toPerson(personRepresentation);
 personService.newPerson(person);

 return ResponseEntity
 .noContent()
 .location(linkTo(
 methodOn(this.getClass()).getPerson(person.getId())).toUri())
 .build();
 }
...
}
```

# Model zrelosti web-usluga

- model je napravio Leonard Richardson
  - <http://www.crummy.com/writing/speaking/2008-QCon/act3.html>
- Razine:
  - 0
    - pozivanje usluga je većinom pozivanje udaljenih procedura
    - jedan URI jedna HTTP metoda (većinom XML-RPC i SOAP)
  - 1
    - koriste više resursa, ali su nazivi metoda i parametara enkodirani u URL
    - više URI-ja, jedna HTTP meotda (GET)
  - 2
    - koriste više resursa i HTTP metoda te statusne kodove
    - ne koriste svoju shemu (vrste podataka koji se prenose)
    - primjer Amazon S3
  - 3
    - isto kao razina 2, ali koristi hipermedijske vrste
    - resurs opisuje svoje mogućnosti i veze

# Standardi, prijedlozi ...

- URI Template - ožujak 2012., IETF, [RFC 6570](#)
  - standard za ekspanziju varijabli iz URI-ja
- JSON Hypertext Application Language (HAL) - svibanj 2016., IETF, v08
  - [draft-kelly-json-hal-08](#)
  - prijedlog višemedijskih tipova za reprezentaciju resursa i njihovih relacija
- Application-Level Profile Semantics (ALPS) - svibanj, 2021., IETF, v07
  - [draft-amundsen-richardson-foster-alps-07](#)
  - format podataka za opis aplikacije i njihove semantike
- [OpenAPI inicijativa](#) (konzorcij)
  - osnovana 2016. u sklopu Linux Foundationa
  - OpenAPI specification (bivši Swagger 2.0 spec. 3.0.3)
  - aktualna verzija [v3.1.0](#)



# Pitanja za ponavljanje

- Koje su dvije vrste web-aplikacija i koja je razlika između njih?
- Kakva je arhitektura web-aplikacije i svrha svakog sloja?
- Koja je razlika između nove i stare arhitekture web-aplikacija?
- Čemu služe skripte na klijentu?
- Što je i kako radi AJAX?
- Usporedite dva načina slanja događaja s poslužitelja (web-aplikacije) klijentu (preglednik).
- Što su i čemu služe web-usluge?
- Objasnite dva načina kako web-aplikacija koristi web-usluge.
- Koja je razlika između Weba 1.0 i Weba 2.0?
- Koje su 3 vrste web-usluga i razlike između njih?

# Pitanja za ponavljanje

- Što je SOAP i čemu služi?
- Što je WSDL i čemu služi?
- Kakva je struktura WSDL-a u čemu služe pojedini elementi?
- Što je UDDI i čemu služi?
- Koja je razlika između web-usluga RPC i temeljenih na dokumentima?
- Kakve su web-usluge temeljene prijenosu prikaza stanja resursa i što ih karakterizira?
- Koje su razlike između JSON-a i XML-a?
- Koja su svojstva metoda protokola HTTP te ih objasnite?
- Objasnite model zrelosti web-usluga.
- Na što treba paziti kod dizajniranja REST usluge?
- Objasniti postupak dizajniranja REST usluge?



SVEUČILIŠTE U ZAGREBU



**Diplomski studij  
Računarstvo**

Znanost o mrežama  
Programsko inženjerstvo i  
informacijski sustavi  
Računalno inženjerstvo  
**Ostali (slobodni izborni  
predmet)**

# Raspodijeljeni sustavi

## 4. Formalni model raspodijeljenog sustava i primjeri raspodijeljenih algoritama

Ak. god. 2022./2023.

# 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 daljnog 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.*

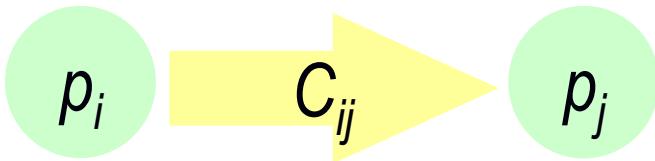
*Tekst licence preuzet je s <http://creativecommons.org/>*

# Sadržaj predavanja

- Model raspodijeljenog sustava
  - model raspodijeljenog izvođenja
  - uzročna ovisnost događaja
  - globalno stanje raspodijeljenog sustava
  - raspodijeljeni algoritam
- Proširenje osnovnog modela raspodijeljenog sustava
  - sinkroni model
  - asinkroni model

# Osnovni model raspodijeljenog sustava

- skup autonomnih procesa  $p_1, p_2, \dots, p_n$
- $C_{ij}$  – kanal koji povezuje procese  $p_i$  i  $p_j$
- $m_{ij}$  – poruka od  $p_i$  za  $p_j$

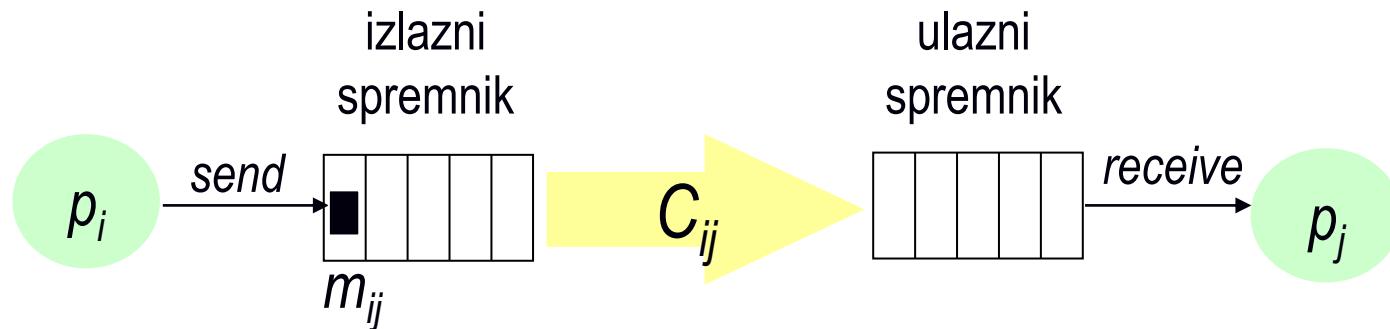


# Svojstva

- Izvođenje procesa i prijenos poruka su asinkroni
- Procesi ne dijele zajednički memorijski prostor
- Pri komunikaciji procesa neminovno se javlja kašnjenje
- Procesi ne koriste jedinstveni globalni sat

# Komunikacija procesa

- procesi međusobno komuniciraju razmjenom poruka (*message passing*) preko komunikacijskog medija (komunikacijske mreže)

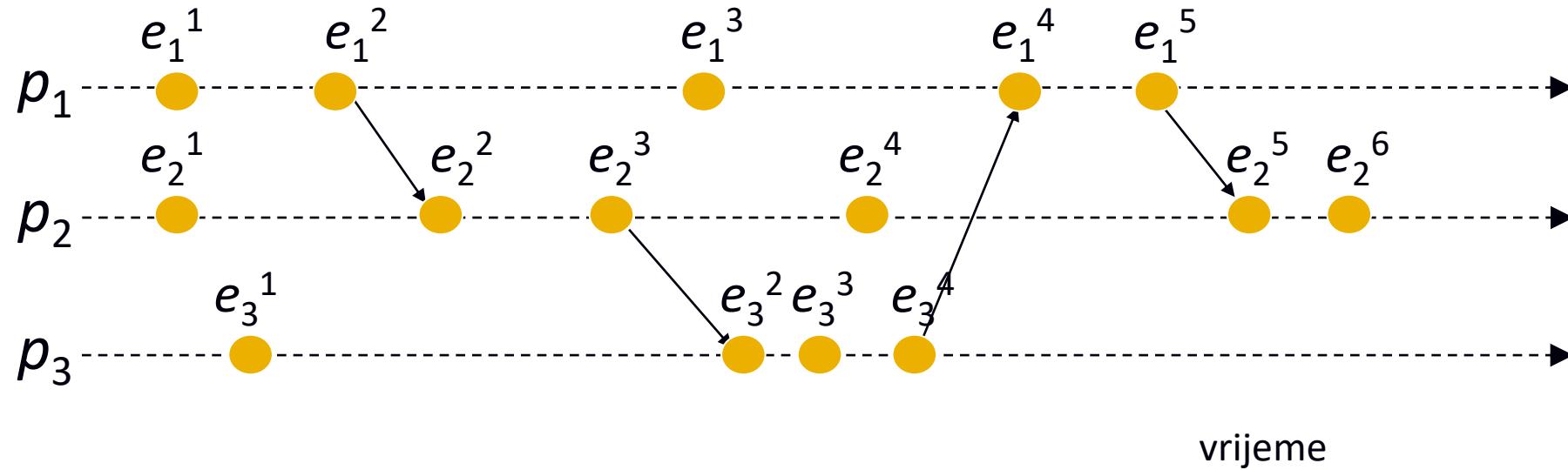


- procesi koriste operatore *send* i *receive*
- send*: pohranjuje poruku u izlazni spremnik i priprema za prijenos preko kanala
- receive*: čita poruku iz dolaznog spremnika i proslijeđuje procesu

# Model raspodijeljenog izvođenja

- Izvođenje procesa: slijedno izvođenje akcija procesa
- **Akcije** se modeliraju sljedećim događajima:
  - unutarnji događaj
  - slanje poruke
  - primanje poruke
- Događaj mijenja stanje procesa i komunikacijskog kanala
- Slijed događaja na procesu  $p_i$ :
$$e_i^1, e_i^2, e_i^3, \dots, e_i^x$$
$$(e_i^2 \text{ se dogodio prije } e_i^3)$$

# Primjer raspodijeljenog izvođenja



# Uzročna ovisnost događaja (1)

- Uzročna relacija ovisnosti događaja (oznaka:  $\rightarrow$ )
  - izražava uzročnu ovisnost između dva događaja tijekom raspodijeljenog izvođenja, uzročnost može biti direktna ili tranzitivna
- $e_i^x \rightarrow e_i^y$ 
  - događaj  $e_i^x$  je izvršen na procesu  $p_i$  prije događaja  $e_i^y$  te su oni uzročno povezani ( $e_i^x$  se nužno dogodio prije  $e_i^y$ )
- $send(m) \rightarrow_{msg} receive(m)$ 
  - uzročna ovisnost vezana uz slanje i primanje poruke, da bi poruka bila primljena, mora prethodno nužno biti poslana na kanal
- $e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \Rightarrow e_i^x \rightarrow e_j^y$ 
  - primjer tranzitivne uzročnosti događaja izvršenih na 3 različita procesa

# Uzročna ovisnost događaja (2)

- Kada su 2 događaja uzročno ovisna?

$$e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow e_j^y, (i = j) \wedge (x < y) & \text{slijedni događaji na istom procesu} \\ e_i^x \rightarrow_{msg} e_j^y & \text{slanje i primanje poruke } msg \\ e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y & \text{tranzitivna uzročnost} \end{cases}$$

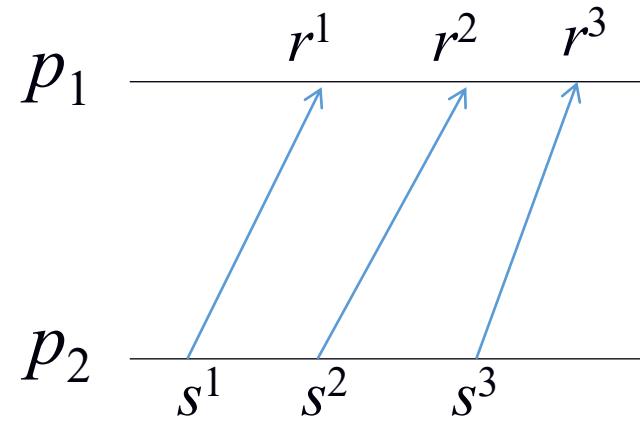
# Uzročna neovisnost događaja

- Uzročna relacija neovisnosti dvaju događaja (oznaka:  $\not\rightarrow$ )
  - označava neovisnost dvaju događaja tijekom raspodijeljenog izvođenja
- $e_i \not\rightarrow e_j$ 
  - događaj  $e_j$  nije ovisan o događaju  $e_i$
- Vrijede sljedeća pravila
  1. za 2 događaja  $e_i$  i  $e_j$ ,  $e_i \not\rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$
  2. za 2 događaja  $e_i$  i  $e_j$ ,  $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$
  3. ako za 2 događaja  $e_i$  i  $e_j$ , vrijedi  $e_i \not\rightarrow e_j$  i  $e_j \not\rightarrow e_i$ , onda su  $e_i$  i  $e_j$  konkurenti događaji i to možemo napisati na sljedeći način  $e_i \parallel e_j$

# Model komunikacijskog kanala (1/3)

## FIFO (first-in, first-out)

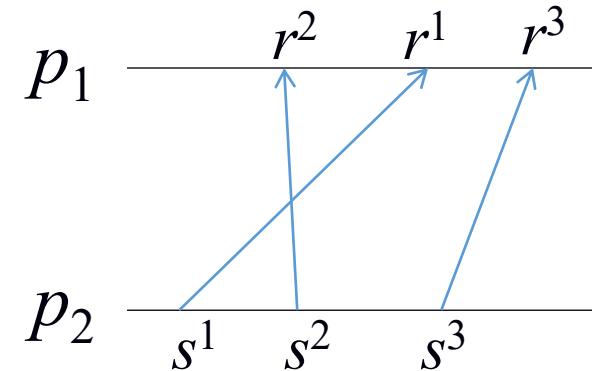
- kanal čuva slijednost poruka,  
ponaša se kao rep



FIFO

## non-FIFO

- kanal ne čuva slijednost poruka,  
ponaša se kao skup

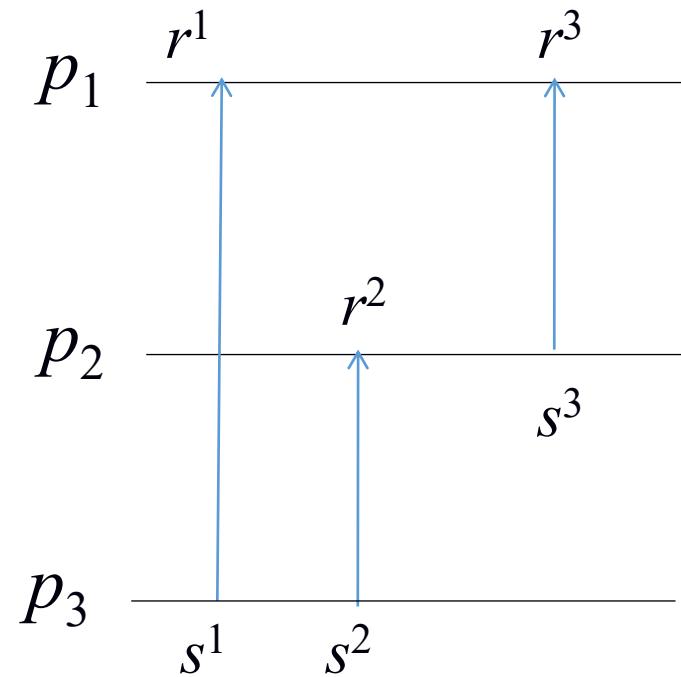


non-FIFO

# Model komunikacijskog kanala (2/3)

## sinkrona slijednost

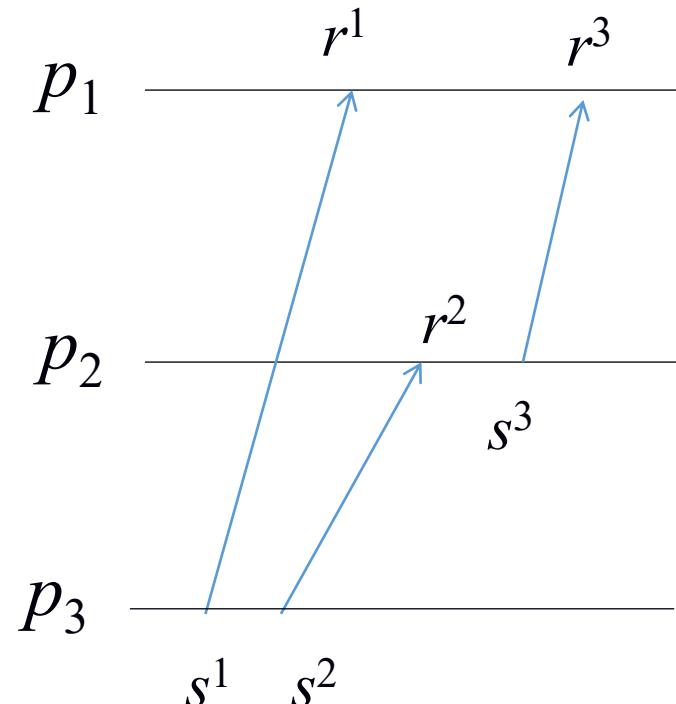
- slanje i primanje poruke događa se istovremeno



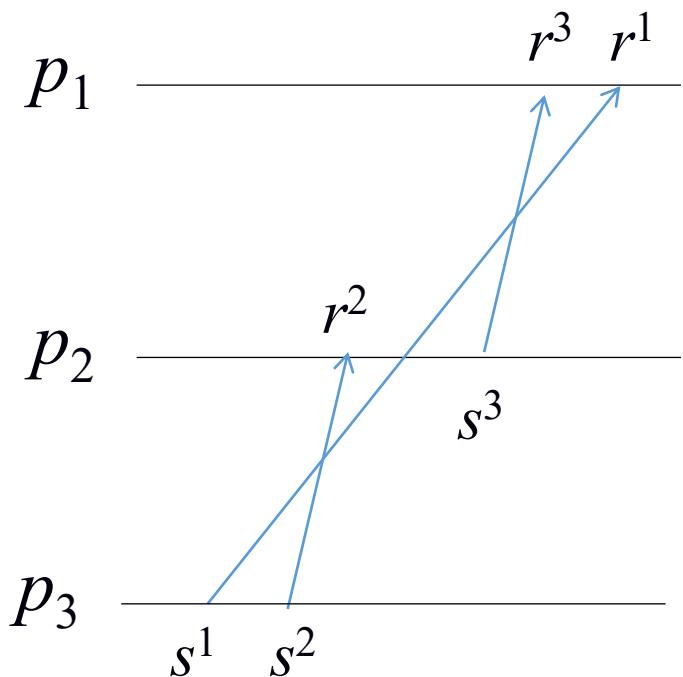
# Model komunikacijskog kanala (3/3)

uzročna slijednost (*causal ordering*, CO)

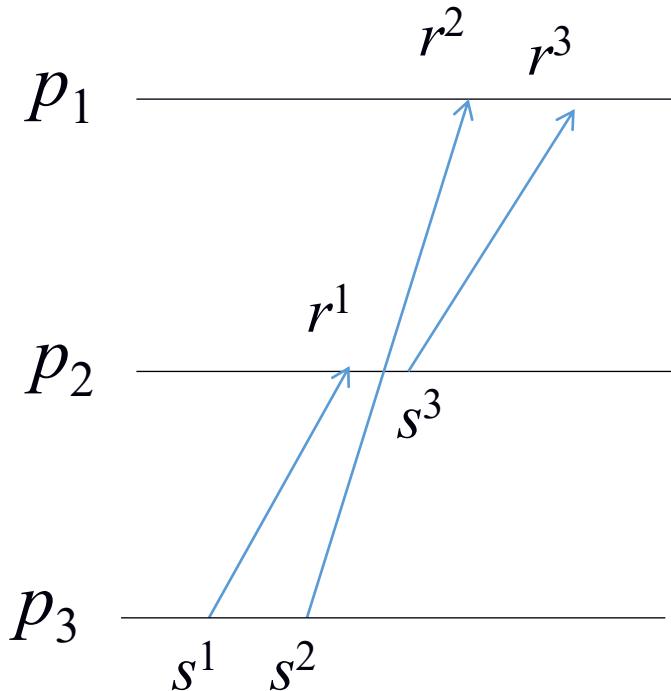
- osigurava da uzročno povezani događaji slanja dviju poruka istom primatelju rezultiraju primanjem u slijedu kojim su poslati
  - ako su dva događaja slanja poruke istom primatelju uzročno povezana (kao  $s^1$  i  $s^3$ ) onda primanje poruka mora slijediti redoslijed slanja ( $r^1$  se mora dogoditi prije  $r^3$  kako bi bilo zadovoljeno svojstvo CO)
- koristan za razvoj distribuiranih algoritama, pojednostavljuje razvoj jer ima ugrađeni mehanizam „sinkronizacije”
  - npr. replicirana baza podataka, svaki proces koji osvježava repliku mora primiti zahtjeve za update u istom slijedu (važno zbog konzistentnosti baze)



# Primjer izvođenja non-CO i CO



**non-CO**  
 $s^1 \rightarrow s^3$ , jer na  $p_1$  se dogodilo  $r^3 \rightarrow r^1$



**CO**  
 $s^1 \rightarrow s^2$ ,  $s^1 \rightarrow s^3$  ali odredišta poruka se razlikuju, a kada analiziramo  $r^2$  i  $r^3$ , poruke slanja  $s^2$  i  $s^3$  su neovisne pa je njihov redoslijed irelevantan

# Globalno stanje raspodijeljenog sustava

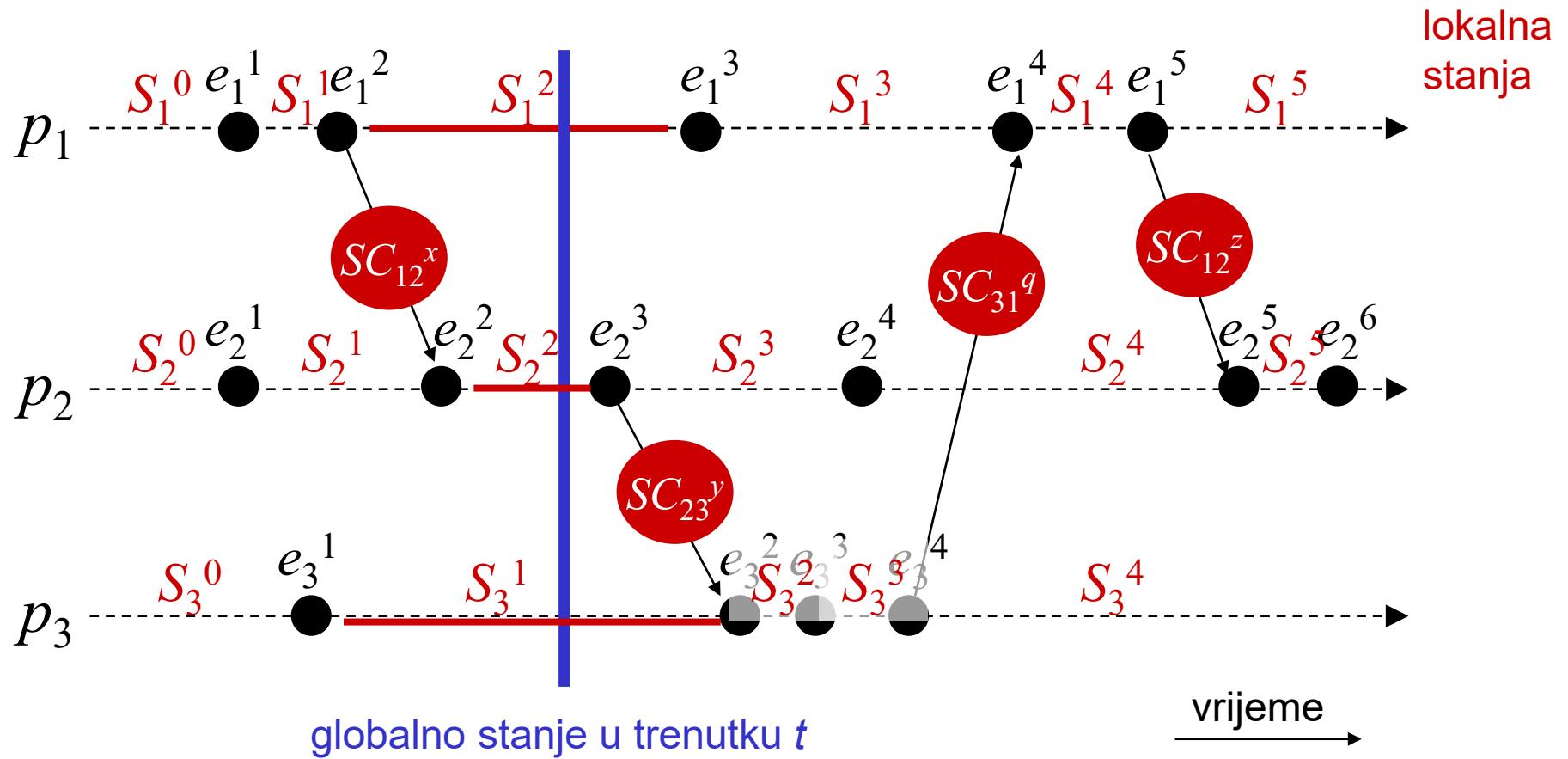
## Lokalno stanje procesa ili kanala

- Stanje procesa određeno je stanjem lokalne memorije i izvođenjem unutarnjih događaja, lokalno stanje procesa je potpuno privatno (ne može ga mijenjati drugi proces)
- Stanje kanala određeno je skupom primljenih i poslanih poruka

## Globalno stanje

- Određeno (trenutnim) lokanim stanjima svih procesa i kanala, kontinuirano se mijenja uslijed akcija tj. događaja na procesima i kanalima
- Izvođenje događaja mijenja lokano stanje procesa/kanala te istovremeno mijenja i globalno stanje raspodijeljenog sustava

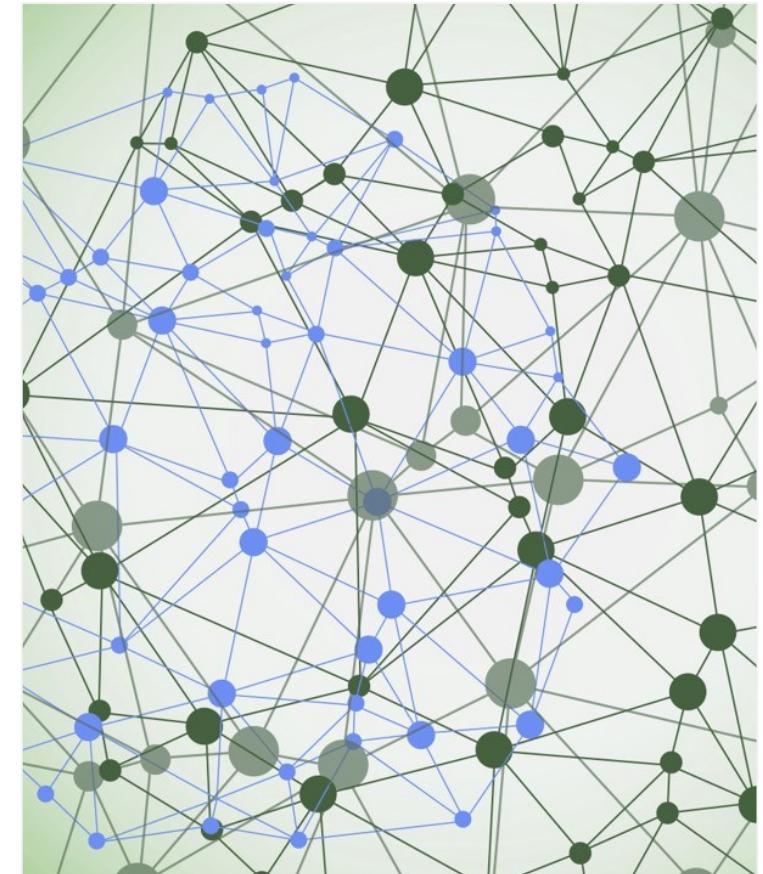
# Primjer lokalnog/globalnog stanja



$$GS(t) = \{S_1^2, S_2^2, S_3^1, SC_{12}^x, SC_{23}^y, SC_{31}^q\}$$

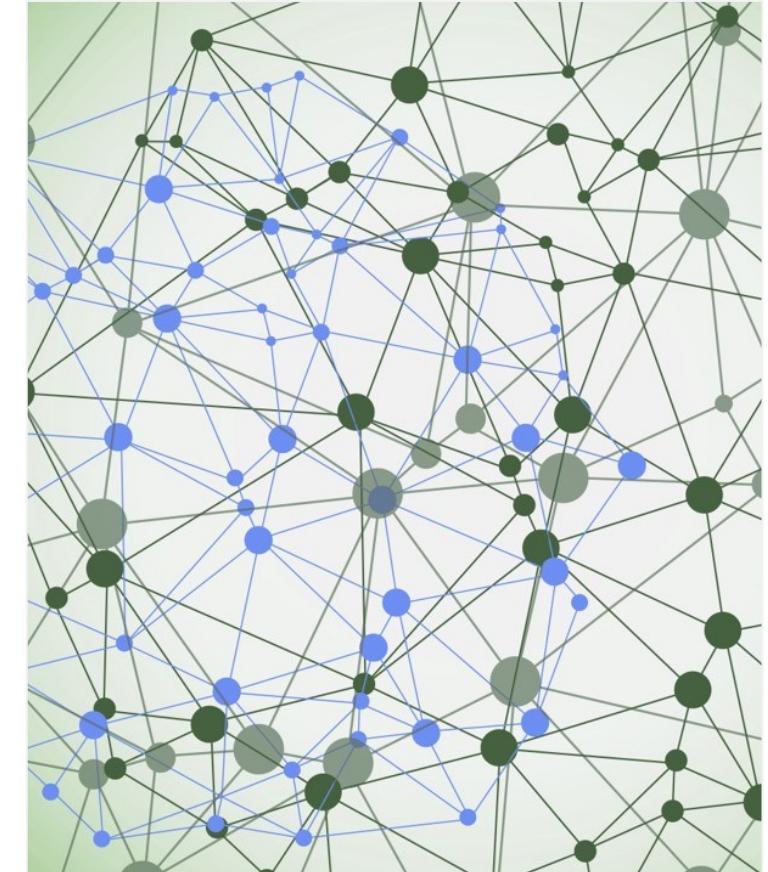
# Raspodijeljeni (distribuirani) algoritam

- Algoritam koji se izvodi u raspodijeljenoj okolini na 2 ili više procesa koji komuniciraju razmjenom poruka
- Definira akcije koje izvodi svaki proces sustava, uključujući prijenos poruka između procesa. Poruke se prenose radi prijenosa informacija i omogućuju koordinaciju aktivnosti procesa koja vodi zajedničkom cilju – implementacija funkcionalnosti koju nudi raspodijeljeni algoritam.



# Raspodijeljeni (distribuirani) algoritam

- izvodi se na većem broju računala povezanih mrežom
- nije jednostavno odrediti početak i kraj izvođenja algoritma, koliko procesa izvodi raspodijeljeni algoritam u svakom trenutku, globalno stanje sustava
- treba biti otporan na ispade procesa jer je to djelomični ispad sustava
- **komunikacija složenost:** brojimo poruke koje se generiraju tijekom izvođenja algoritma



# Sadržaj predavanja

- Model raspodijeljenog sustava
  - model raspodijeljenog izvođenja
  - uzročna ovisnost događaja
  - globalno stanje raspodijeljenog sustava
  - raspodijeljeni algoritam
- Proširenje osnovnog modela raspodijeljenog sustava
  - sinkroni model
  - asinkroni model

# Proširenje osnovnog modela (sinkroni i asinkroni)

## Sinkroni model

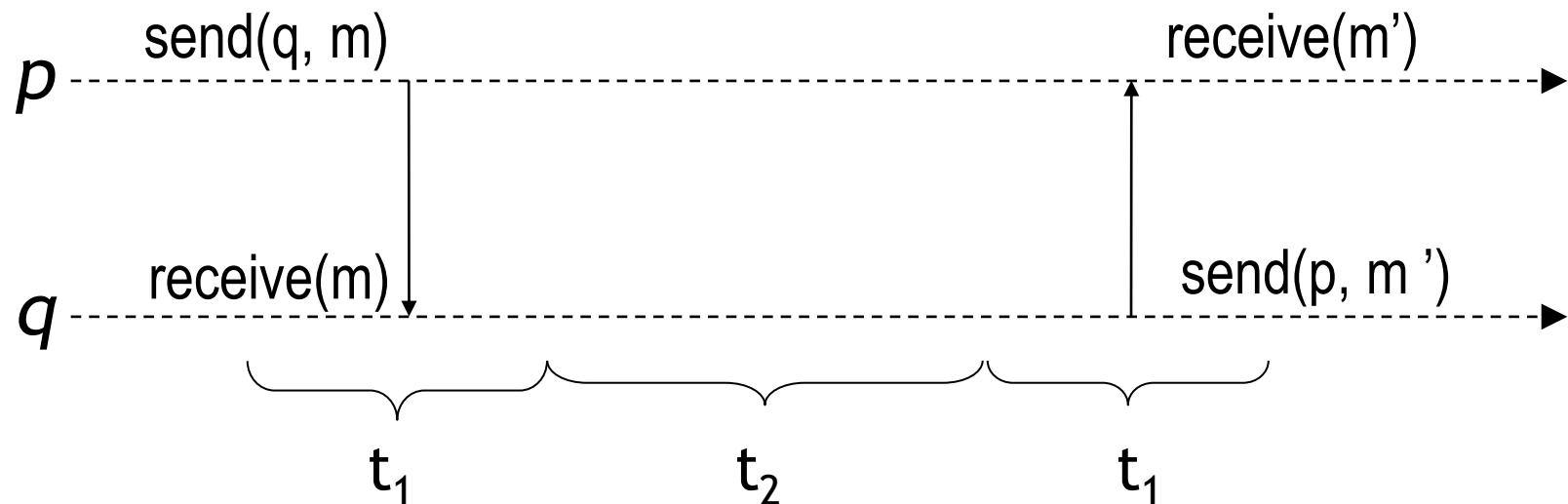
- prepostavka: svi procesi raspodijeljenog sustava izvode događaje (tj. korake) istovremeno
- pojednostavljenje koje nije realno za raspodijeljene sustave, ali može biti korisno za njihovo razumijevanje i analizu

## Asinkroni model

- prepostavka: procesi izvode događaje u proizvolnjom slijedu
- postoji neodređenost vezana uz slijed događaja
- realna situacija

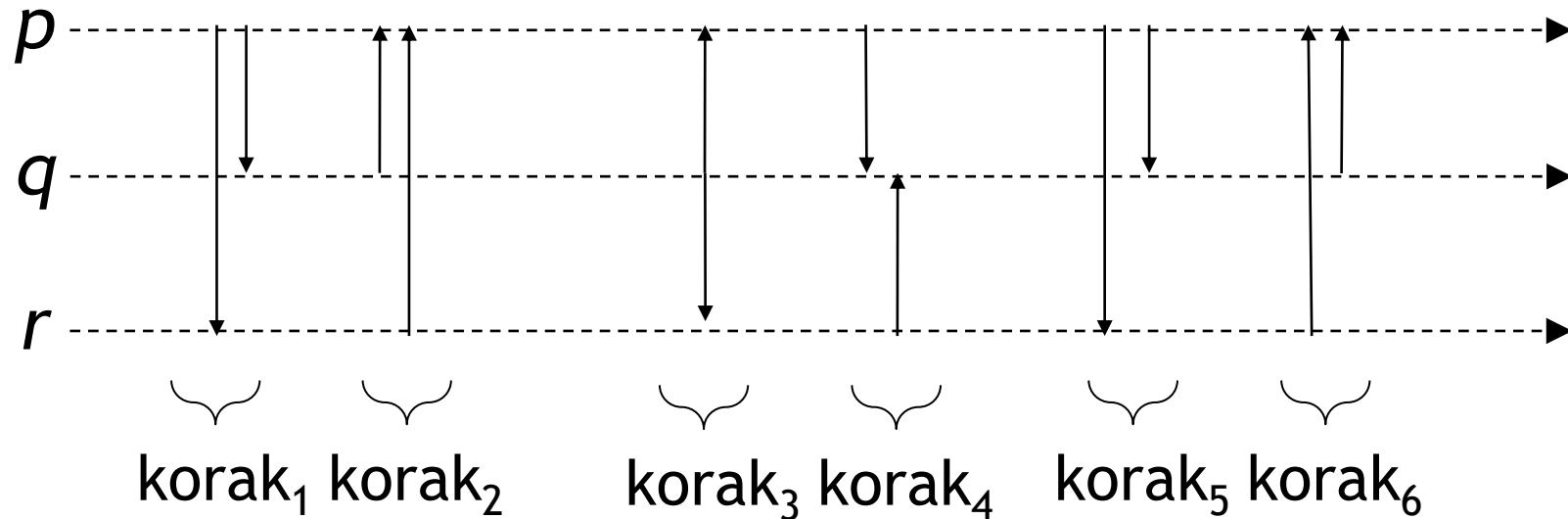
# Sinkroni model

- Poznata je gornja vremenska granica za
  - trajanje prijenosa poruke kanalom ( $t_1$ ) i izvođenje prijelaza nekog procesa ( $t_2$ )
- Pretpostavka
  - procesi imaju potpuno sinkronizirana lokalna vremena



# Primjer sinkrone komunikacije

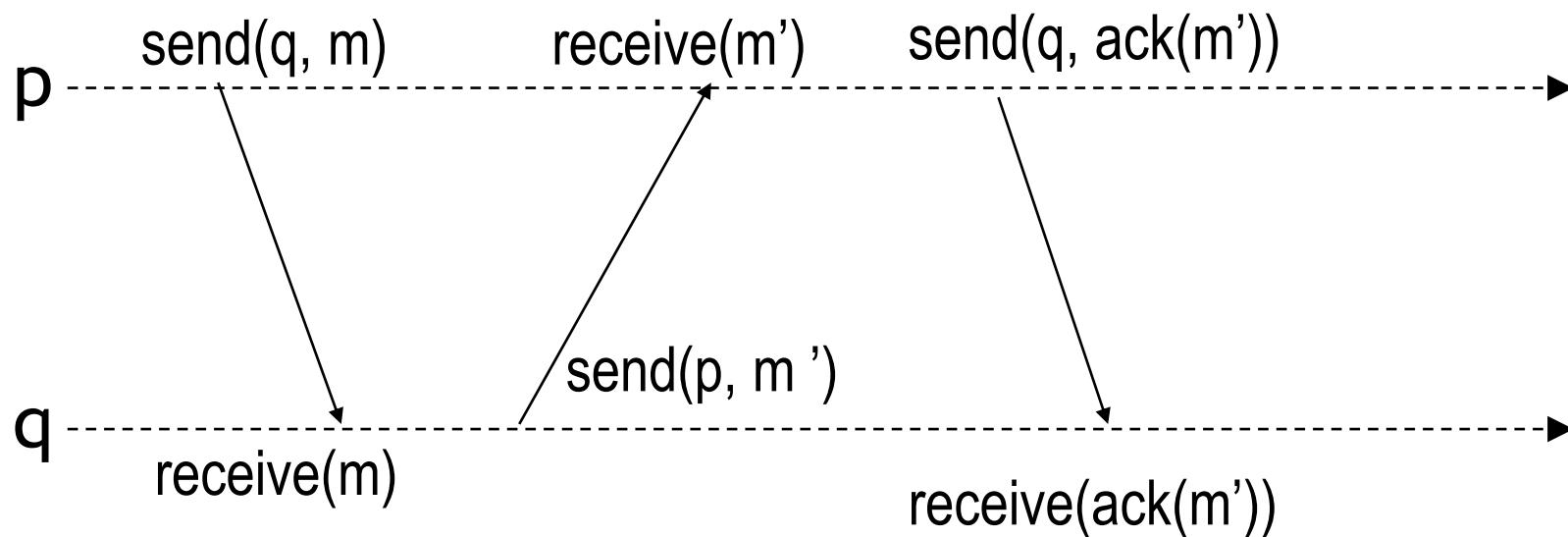
- izvođenje algoritma u sinkronom sustavu organizirano je u koracima
  - pošalji poruke procesima u sustavu
  - primi poruke od drugih procesa u sustavu
  - izvođenje prijelaza: promijeni stanje na temelju primljenih poruka



# Asinkroni model

- Ne postoji gornja vremenska granica za
  - izvođenje prijelaza nekog procesa (no trajanje prijelaza je uvijek konačno)
  - trajanje prijenosa poruke kanalom
- Pretpostavka
  - procesi **nemaju** sinkronizirana lokalna vremena
- Realni slučaj koji ćemo najčešće razmatrati, znatno komplificira model i analizu distribuiranog algoritma raspodijeljenog sustava

# Primjer asinkrone komunikacije

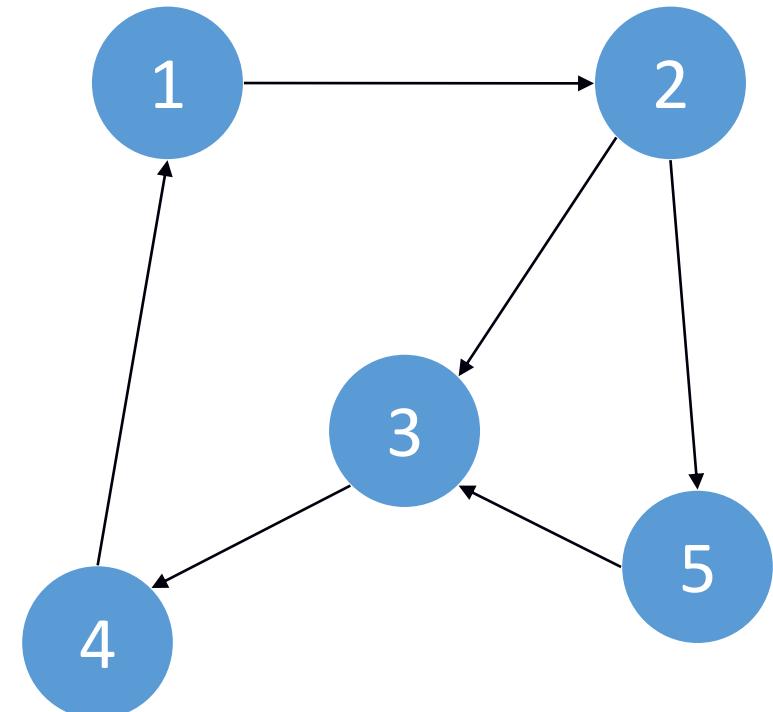


nepouzdani komunikacijski medij, potrebno je modelirati vjerojatnost gubitka poruke na kanalu

# Sinkroni model raspodijeljenog sustava

# Sinkroni model

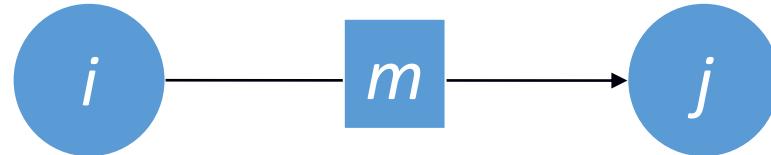
- usmjereni graf  $G = (V, E)$
- $v_i \in V$ , čvor modelira **proces**
- $e_j \in E$ , grana modelira **kanal**
- $M$  je skup poruka, *null* ako na kanalu nema poruka
- $out-nbrs_i$  – izlazni susjedi
- $in-nbrs_i$  – ulazni susjedi
- $distance(i, j)$  – najkraći put između  $i$  i  $j$  u  $G$ , ( $i, j \in V$ )
- $diameter(G)$  – max  $distance(i, j)$  za sve parove  $(i, j)$



# Model procesa

- svaki se proces vezan uz čvor  $v_i \in V$  modelira kao uređena četvorka:  $(states_i, start_i, msgs_i, trans_i)$
- $\underline{states_i}$  – skup mogućih stanja procesa
- $start_i$  – skup početnih stanja
  - $start_i \subset states_i$ ,  $start_i \neq \emptyset$
- $msgs_i$  – funkcija za generiranje poruka
  - određuje izlaznu poruku za svakog susjeda na temelju trenutnog stanja procesa
  - $states_i \times out-nbrs_i \rightarrow M_i \subset M \cup \{\text{null}\}$
- $trans_i$  – funkcija prijelaza, određuje sljedeće stanje na temelju trenutnog stanja i primljenih poruka od ulaznih susjeda

# Model kanala

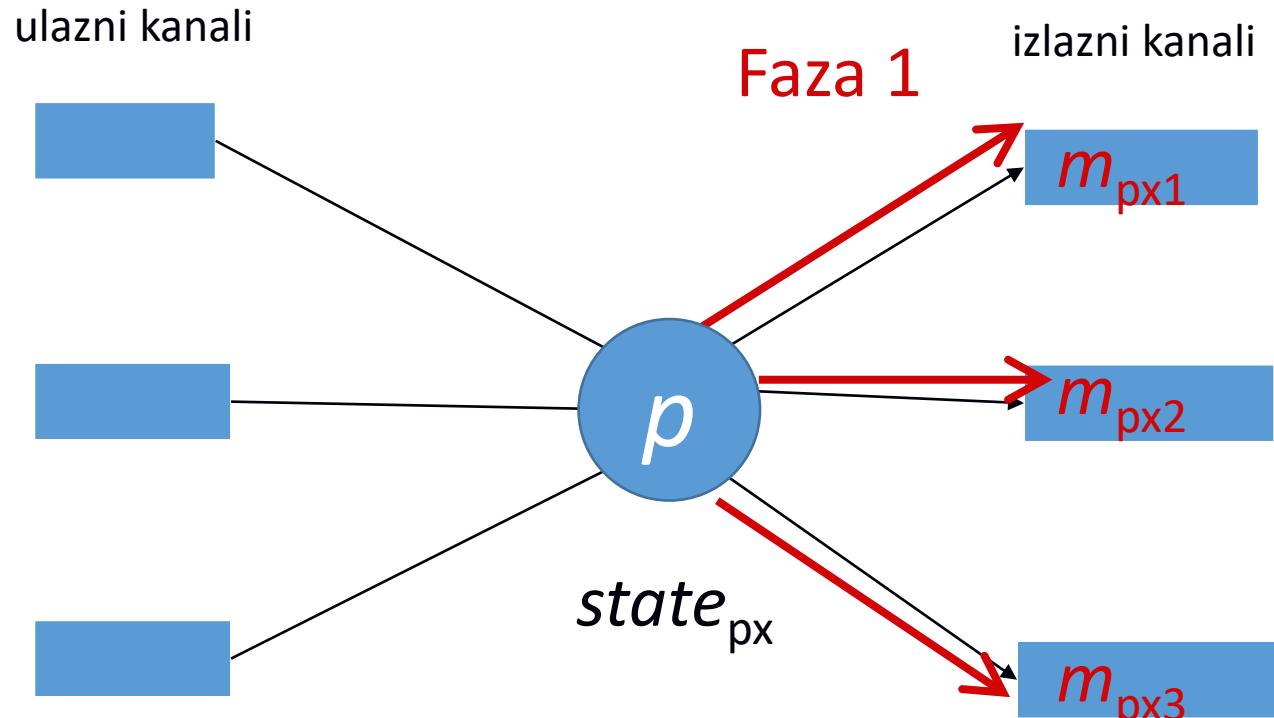


- modelira ga grana između para čvorova  $(i, j)$  iz  $G$
- može primiti poruku  $m$  iz definiranog skupa poruka  $M$  ili  $null$
- $null$  označava praznu poruku

# Izvođenje sinkronog modela

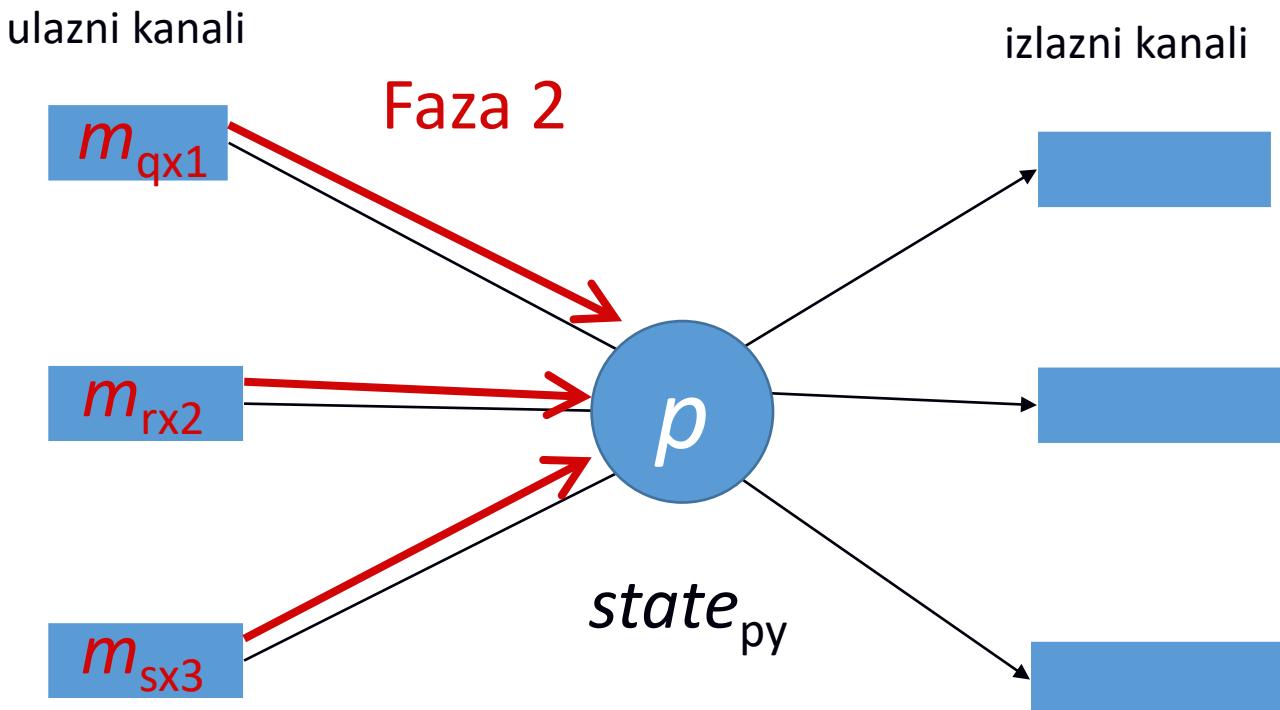
- Algoritmi za sinkrone modele se izvode u **koracima** (*round*). Inicijalno su svi procesi u proizvoljnom početnom stanju i svi su kanali prazni. Nakon toga se izvode koraci.
- Korak se sastoji od 2 faze
  - **Faza 1:** Za svaki proces primjeni funkciju za generiranje poruka (*msg*), a na temelju trenutnog stanja. Generiraj poruke koje će biti poslane izlaznim susjedima i postavi te poruke na izlazne kanale.
  - **Faza 2:** Primjeni funkciju prijelaza (*trans*) koja će na temelju trenutnog stanja i primljenih poruka odrediti sljedeće stanje procesa. Briši sve poruke na kanalima.

# Izvođenje (faza 1)



na temelju trenutnog stanja generiraj  
poruke i postavi ih na izlazne kanale

# Izvođenje (faza 2)



primijeni funkciju prijelaza koja na temelju  
primljenih poruka određuje sljedeće stanje  
procesa

# Formalni model izvođenja

- Beskonačni slijed:  
 $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \dots$
- $C_k$  - stanje svih procesa nakon  $k$  koraka
- $M_k$  – poslane poruke na svim kanalima nakon  $k$  koraka
- $N_k$  – primljene poruke na svim kanalima nakon  $k$  koraka
- $M_k \neq N_k$  – ako dođe do ispada na nekom kanalu, inače je u sustavima bez gubitaka poruka  $M_k = N_k$  za svaki  $k$

# Složenost sinkronog algoritma

- vremenska složenost
  - mjeri se brojem izvedenih koraka (*rounds, r*) koji dovodi do završnog stanja algoritma tj. do stanja u kome su svi procesi zaustavljeni ili kada se više ne proizvode novi izlazi
- komunikacijska složenost
  - mjeri se broj kreiranih i poslanih poruka na kanalima

(Pri određivanju mjere složenosti uvijek se analizira najgori mogući scenarij izvođenja algoritma!)

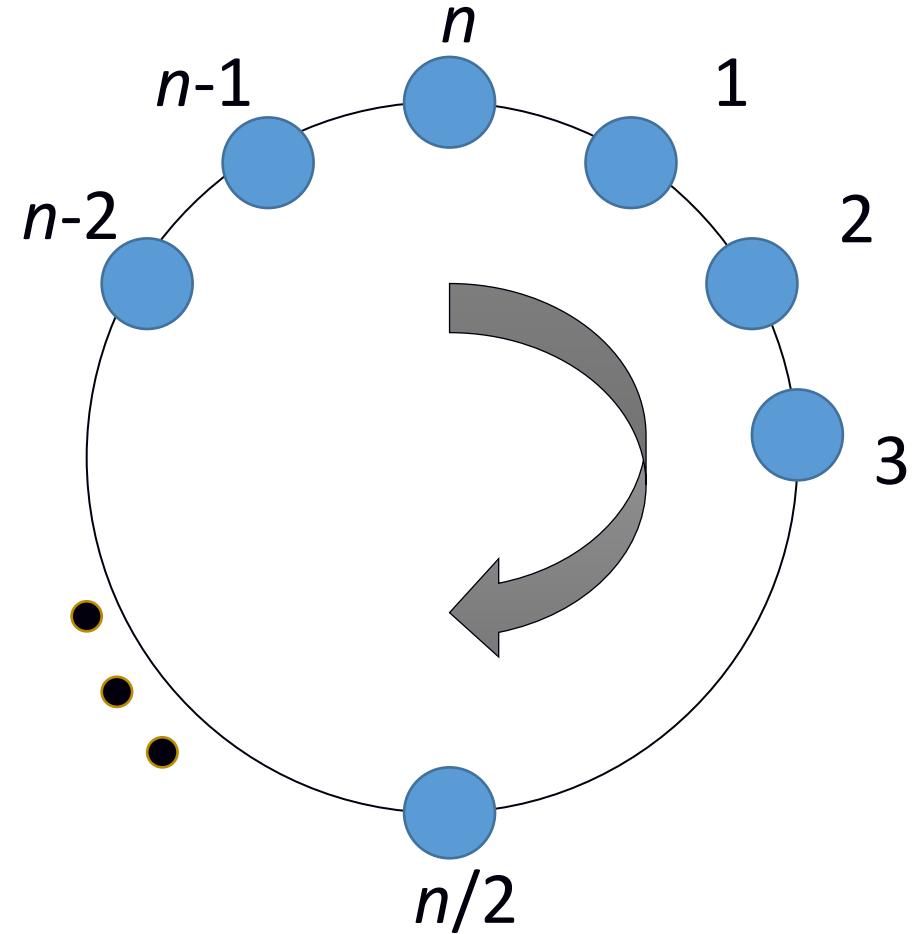
(Usporedite s mjerama složenosti algoritama, vremenska i prostorna)

# Primjeri sinkronog modela

# Problem 1: Odabir vođe u sinkronom prstenu

## Definicija problema

- Izabrati jedinstvenog “vođu” među procesima u mreži
- U bilo kojem koraku samo 1 proces može postati vođa i promijeniti status u *leader*
- Pretpostavka jednostavne mreže od  $n$  čvorova
  - *token ring*
  - svaki čvor je označen brojem od 1 do  $n$



# Dodatni zahtjevi

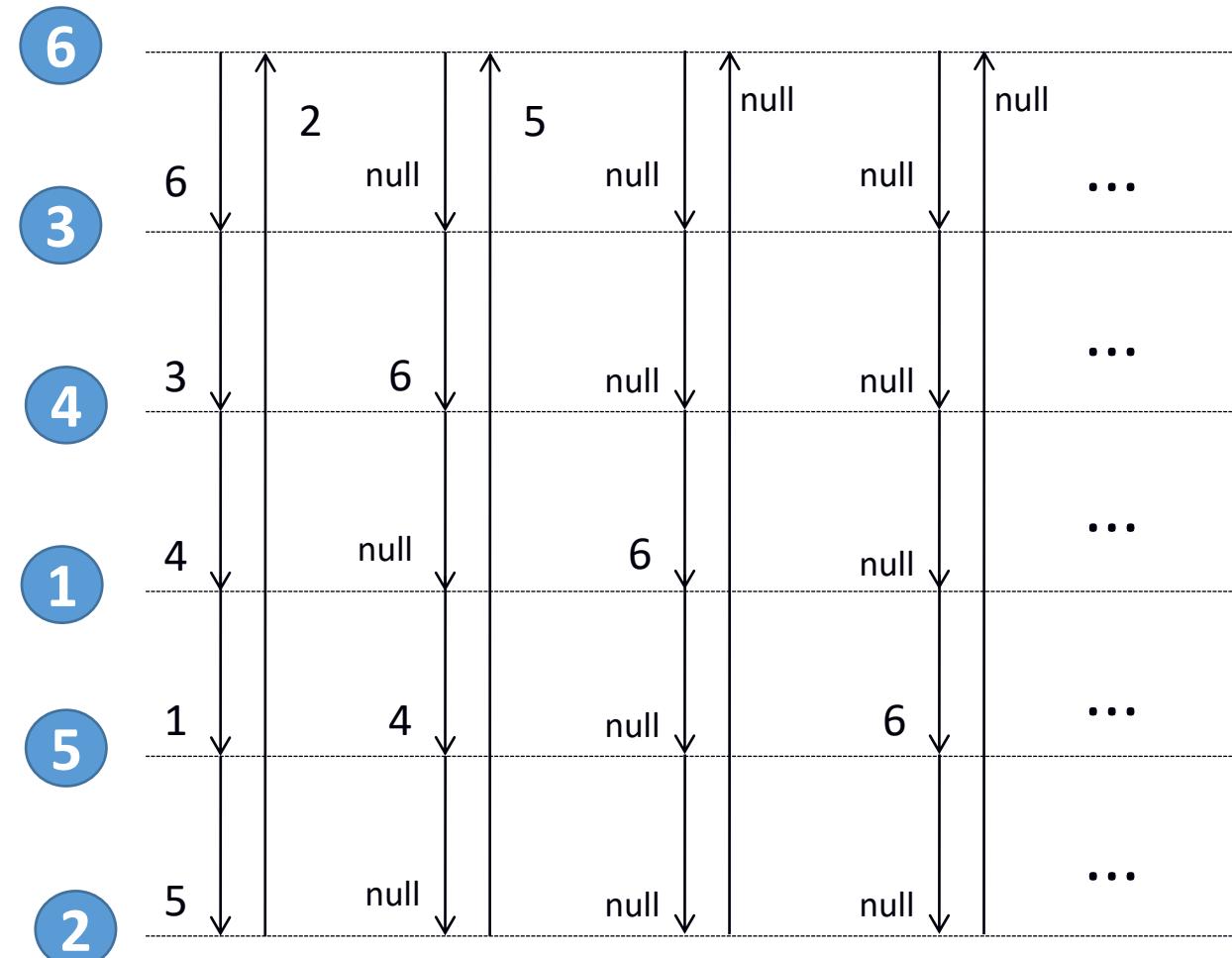
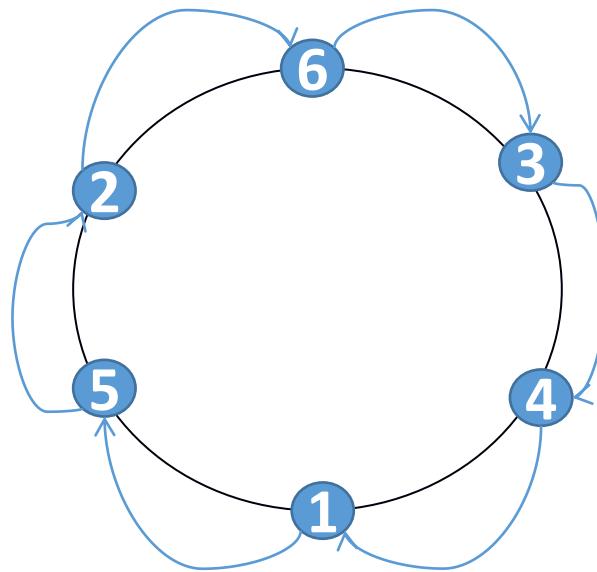
- Svi procesi su identični osim po identifikatoru
  - svaki ima jedinstven identifikator (UID - *Unique ID* )
  - UID nisu sljedbenici u prstenu
  - UID se mogu međusobno uspoređivati
  - (to omogućuje odabir vođe, inače bi svi procesi bili jednaki i vođa se ne bi mogao odabrati)
- Procesi znaju svoje susjede (ulazni ili izlazni)
- Broj procesa u prstenu ( $n$ ) može biti poznat ili nepoznat svim procesima

# Osnovni algoritam za odabir vođe

- Pretpostavke
  - Jednosmjerna komunikacija među procesima u prstenu (usmjereni graf u smjeru kazaljke na satu)
  - Procesi ne znaju veličinu prstena  $n$
  - Svaki proces ima jedinstveni identifikator UID iz skupa prirodnih brojeva, UID se procesu dodjeljuje na slučajan način
- Vođa je proces s najvećim UID
- Skica algoritma:

Svaki proces inicijalno šalje svoj UID susjedu. Kada proces primi UID, ako je taj veći od njegovog UID-a prosljeđuje ga dalje, ako je primljeni UID malji od njegovog UID-a primljeni UID se odbacuje, a ako je primljeni UID jednak njegovom UID-u proces objavljuje sebe kao vođu

# Primjer prstena i algoritma za odabir vode



# Formalni model osnovnog algoritma

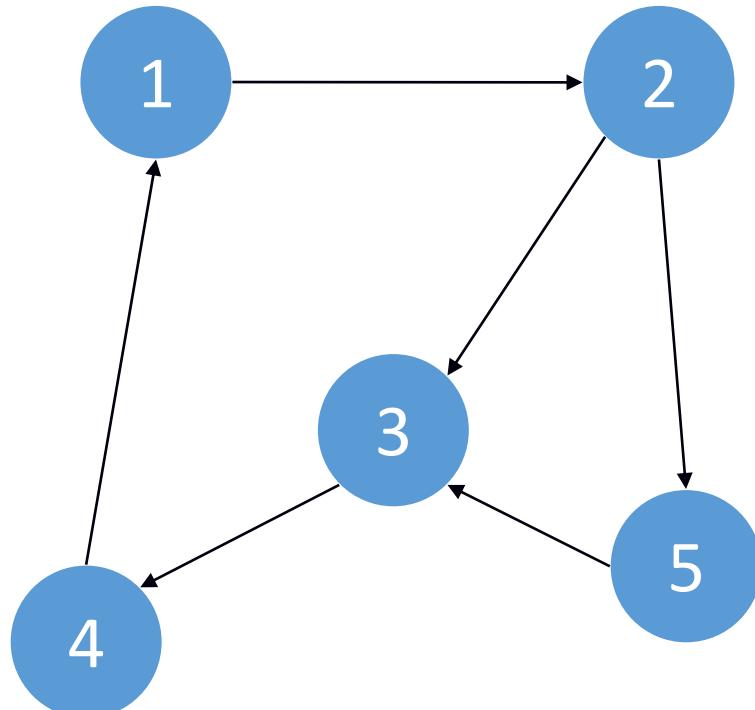
- $M$  – skup poruka čini skup svih UID
- Za svaki proces  $i$ :
  - $states_i = (u, send, status)$ 
    - $u$  – identifikator, inicijalno UID za  $i$
    - $send$  – identifikator ili null, inicijalno UID za  $i$
    - $status \in \{unknown, leader\}$ , inicijalno  $unknown$
  - $start_i = (\text{UID procesa } i, \text{UID procesa } i, unknown)$
  - $msgs_i$  – poslati vrijednost varijable  $send$  sljedećem procesu
  - $trans_i$  –
    - receive  $v$
    - $send := \text{null}$  (pobriši poruke na kanalima)
    - if  $v > u$  then  $send := v$
    - if  $v = u$  then  $status := leader$
    - if  $v < u$  then *do nothing*

# Složenost algoritma

- Vremenska
  - s obzirom da algoritam završava u slučaju kada čvor s najvećim UID ponovo primi vlastitu poruku, potrebno je  $n$  koraka da ta poruka stigne do vođe u prstenu s  $n$  čvorova
  - samo će proces koji je vođa znati da je algoritam završen (primio je poruku identičnu vlastitom UID), stoga može poslati posebnu poruku (*halt*) s obavijesti da je vođa izabran – potrebno je  $2n$  koraka za pronađak vođe i slanje poruka zaustavljanja
- Komunikacijska
  - u mreži se generira  $O(n^2)$  poruka - pri svakom koraku svaki čvor potencijalno generira novu poruku ( $n^2 = n$  poruka po koraku  $\times n$  koraka do završetka algoritma)
  - Ako analiziramo max broj generiranih poruka uzimajući u obzir *null* poruke, onda je to za mrežu s padajućim UID u smjeru kazaljke na satu kada se za svaki korak generira  $n+(n-1)+(n-2)+\dots+1$ , što je ukupno  $n*(n+1)/2$  poruka, što je  $O(n^2)$

## Problem 2: Odabir vođe u usmjerenoj mreži

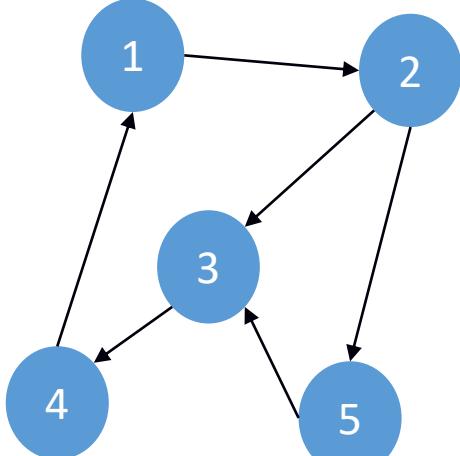
- Povezana usmjerena mreža, za svaki par čvorova postoji konačan  $distance(i, j)$
- Svaki čvor ima jedinstveni identifikator UID
- Izabrati vođu među procesima u mreži
- Samo 1 proces mijenja status u *leader*
- $diameter(G) = \max distance(v_i, v_j)$  za sve parove  $(v_i, v_j)$  iz  $G$



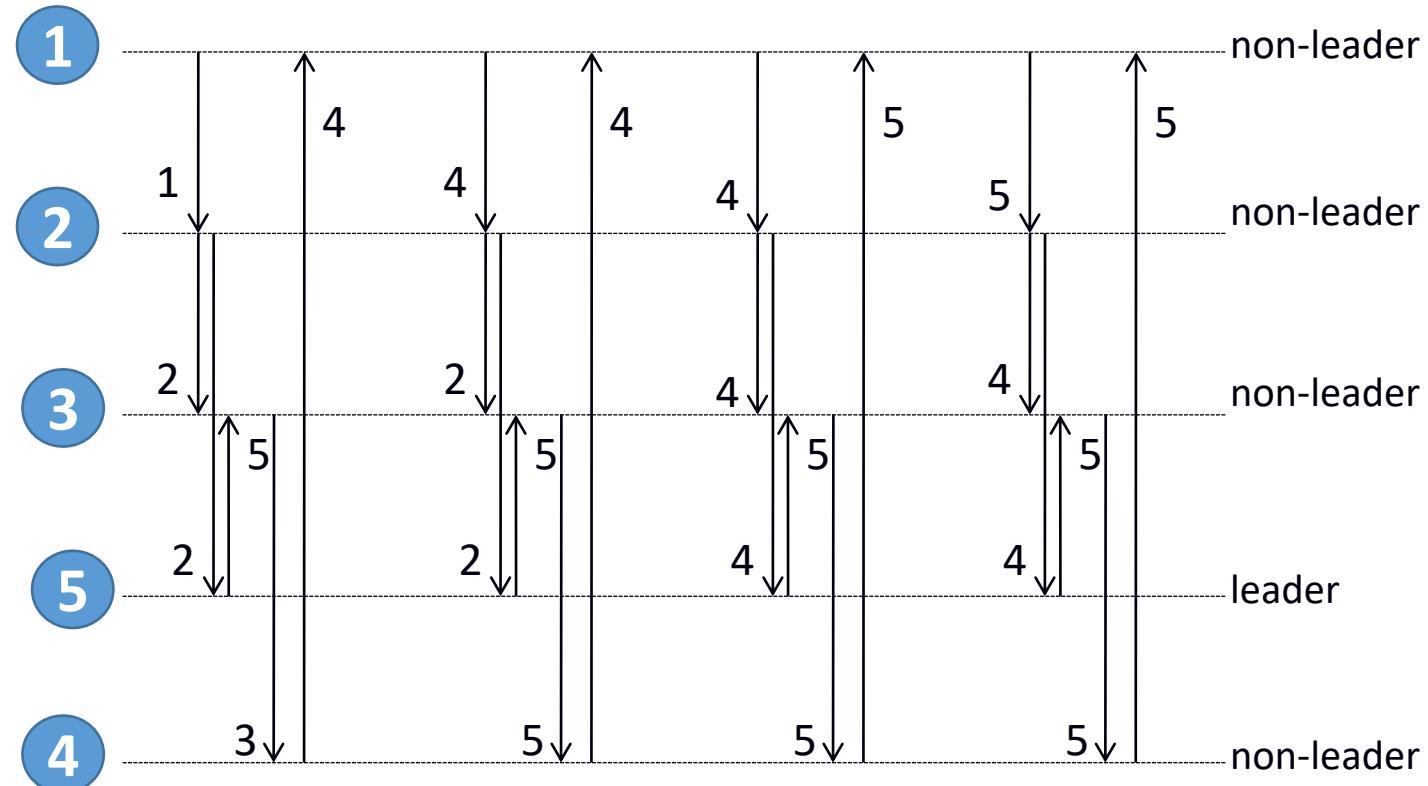
# Rješenje problema 2: Algoritam preplavljanja

- *Simple Flooding Algorithm (SFA)*
- Prepostavke
  - Svaki proces ima UID iz skupa prirodnih brojeva
  - Svaki proces zna  $diameter(G)$  – (**pažnja: ovo je globalno znanje!**)
- Skica algoritma
  - Svaki proces bilježi max primljeni UID (inicijalno je to vlastiti UID). U svakom koraku proces šalje tu maksimalnu vrijednost na izlaznim kanalima svim susjedima. Nakon  $diameter(G)$  koraka ako je maksimalna vrijednost jednaka vlastitom UID, proces se proglašava vođom, a u suprotnom nije vođa.

# Primjer algoritma za odabir vođe u usmjerenoj mreži



diameter = 4 jer je  
distance (5,2) = 4



# Formalni model algoritma preplavljanja

- $states_i = (u, max\text{-}uid, status, rounds)$   
 $u$  – UID, inicialno UID za  $i$   
 $max\text{-}uid$  – UID, inicialno UID za  $i$   
 $status \in \{unknown, leader, non-leader\}$ , inicialno *unknown*  
 $rounds$  – cijeli broj, inicialno 0
- $msgs_i$  –  
if  $rounds < diameter$  then  
    send *max-uid* to all  $j \in out\text{-}nbrs$
- $trans_i$  –  
 $rounds := rounds + 1$   
receive set of UIDs  $U$  from neighbors  
 $max\text{-}uid := \max(\{max\text{-}uid\} \cup u)$   
if  $rounds = diameter$  then  
    if  $max\text{-}uid = u$  then  $status := leader$   
    else  $status := non-leader$

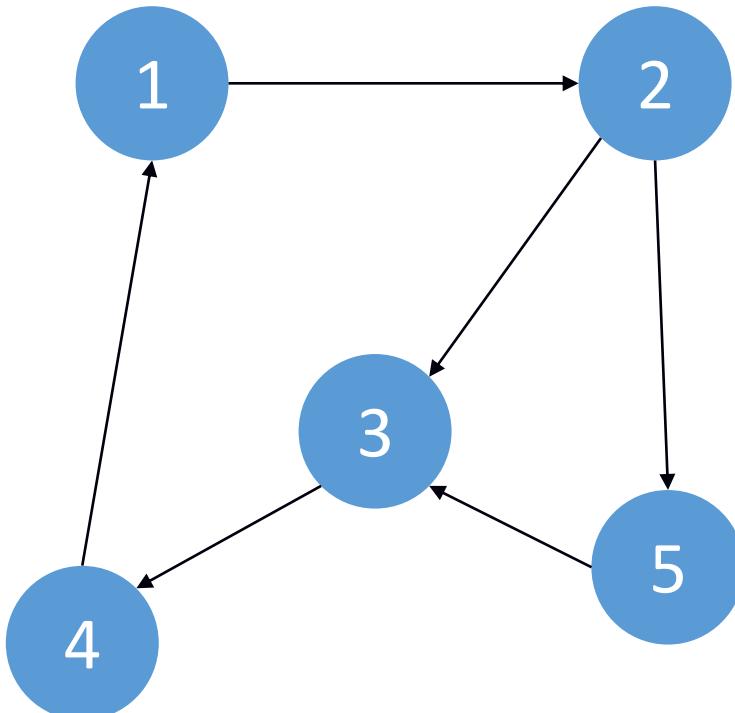
# Složenost

- Vremenska
  - Određena vrijednošću  $diameter(G)$
- Komunikacijska
  - broj poruka = diameter  $\cdot |E|$ , gdje je  $|E|$  broj usmjerenih grana grafa
  - poruka se šalje na svaku granu za svaki korak algoritma
  - jednostavna optimizacija koja smanjuje broj poruka – proces šalje *max-uid* susjedima samo ako se vrijednost *max-uid* promijeni

# Asinkroni model raspodijeljenog sustava

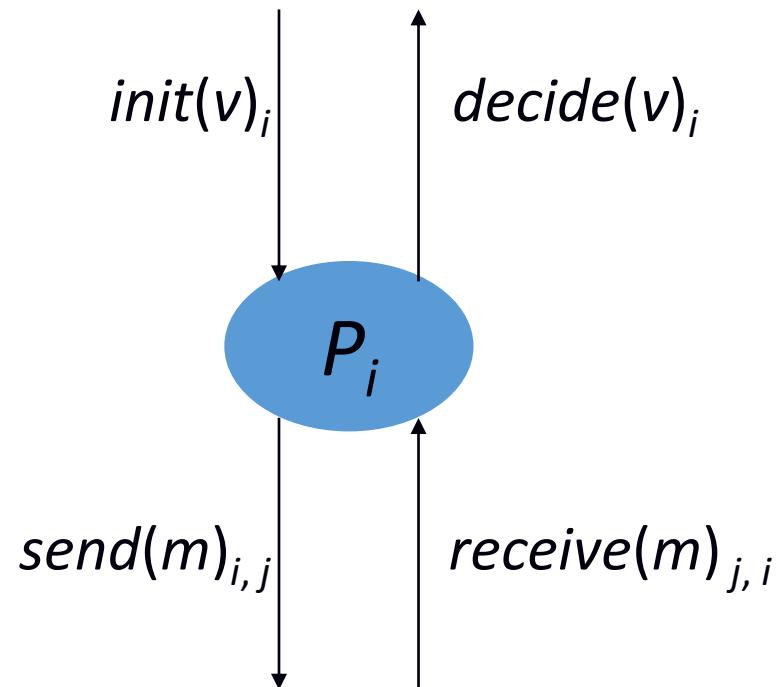
# Asinkroni model

- ♦ usmjereni graf  $G = (V, E)$
- ♦  $v_i \in V$ , čvor modelira proces
- ♦  $e_j \in E$ , grana modelira kanal
- ♦  $out-nbrs_i$  – izlazni susjedi
- ♦  $in-nbrs_i$  – ulazni susjedi
- ♦ asinkronost izvođenja procesa i komunikacije (razlika u odnosu na sinkroni model)
- ♦ svaki proces i svaki kanal se modeliraju I/O automatom



# Ulazno/izlazni (I/O) automat

- formalni model za asinkrone sustave
- I/O automat modelira komponentu raspodijeljenog sustava koja je u interakciji s ostalim komponentama
- prijelazi su vezani uz **događaje**
- događaji mogu biti *ulazni*, *izlazni* ili *unutarnji*



primjer procesa u asinkronom raspodijeljenom sustavu

# Formalna definicija I/O automata

I/O automat  $A$  se sastoji od sljedećih komponenti:

- $\text{sig}(A)$  – signatura  
 $\text{sig}(A) = \{ \text{in}(A), \text{out}(A), \text{int}(A) \}$  – opis ulaznih, izlaznih i unutarnjih događaja)
- $\text{states}(A)$  – skup stanja automata
- $\text{start}(A)$  – skup početnih stanja,  $\text{start}(A) \neq \emptyset$
- $\text{trans}(A)$  – funkcija prijelaza,  
npr.  $(s, \pi, s')$  –  $s$  i  $s'$  su stanja, a  $\pi$  je događaj
  - za svako stanje  $s$  i svaki ulazni događaj  $\pi$  postoji prijelaz  $(s, \pi, s') \in \text{trans}(A)$

# Izvođenje automata

- automat  $A$  se izvodi kao konačan ili beskonačan slijed stanja i događaja, npr.

$s_0, \pi_0, s_1, \pi_1, s_2, \pi_2, s_2, \dots \pi_k, s_k, \dots$

- $(s_k, \pi_k, s_{k+1}) \in \text{trans}(A)$ , za svaki  $k \geq 0$

# Primjer: automat kanala FIFO (1)



- $sig(C_{i,j}) = (send(m)_{i,j}, receive(m)_{i,j}, 0), m \in M$
- states:
  - *queue*, a FIFO queue
- trans:
  - $send(m)_{i,j}$  – dodaj  $m$  u *queue*
  - $receive(m)_{i,j}$  – preduvjet:  $m$  je 1. element iz *queue*, posljedica: briši  $m$  iz *queue*

# Primjer: automat kanala FIFO (2)

primjeri izvođenja – tj. slijed događaja (engl. *trace*)

- [null],  $send(1)_{i,j}$ , [1],  $receive(1)_{i,j}$ , [null],  $send(2)_{i,j}$ , [2],  $receive(2)_{i,j}$ , [null]
- [null],  $send(1)_{i,j}$ , [1],  $send(1)_{i,j}$ , [11],  $send(1)_{i,j}$ , [111]...

Za “*trace*” su važna sljedeća 2 svojstva:

- A **safety property** is often interpreted as saying that some particular “bad” thing never happens.
- A **liveness property** is often informally understood as saying that some particular “good” thing eventually happens.

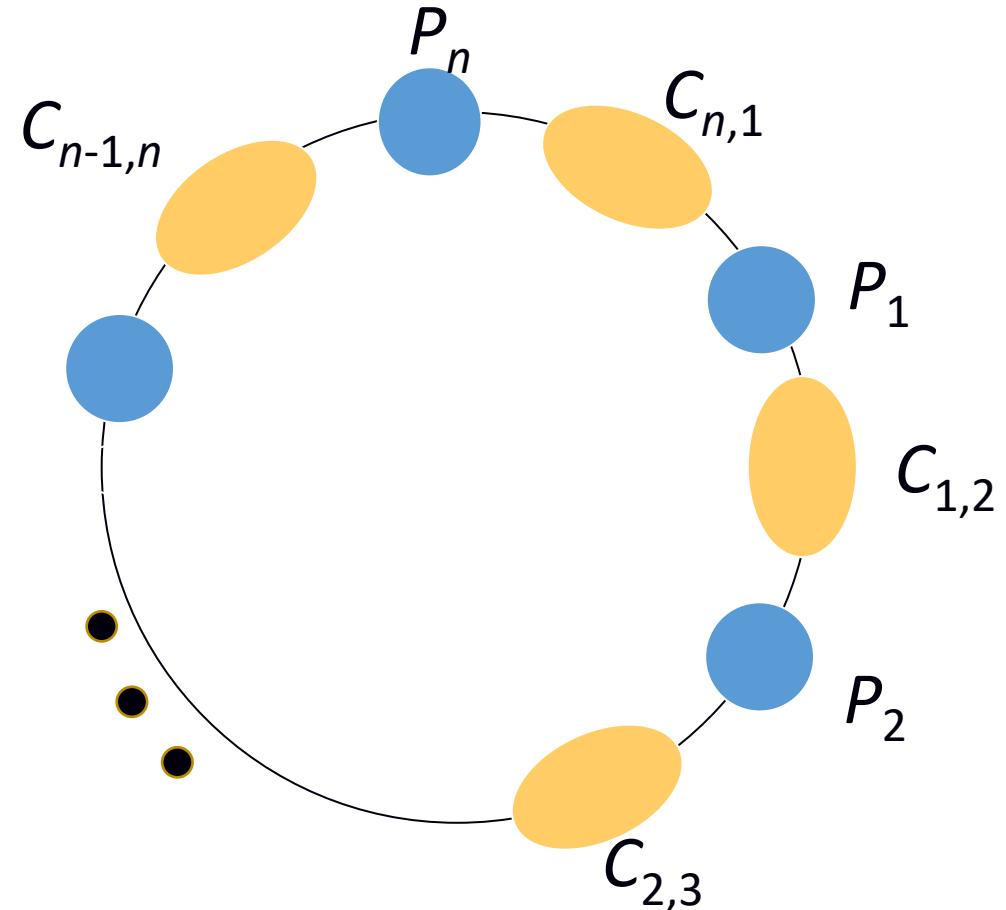
(prema N. Lynch)

# Primjeri asinkronog modela

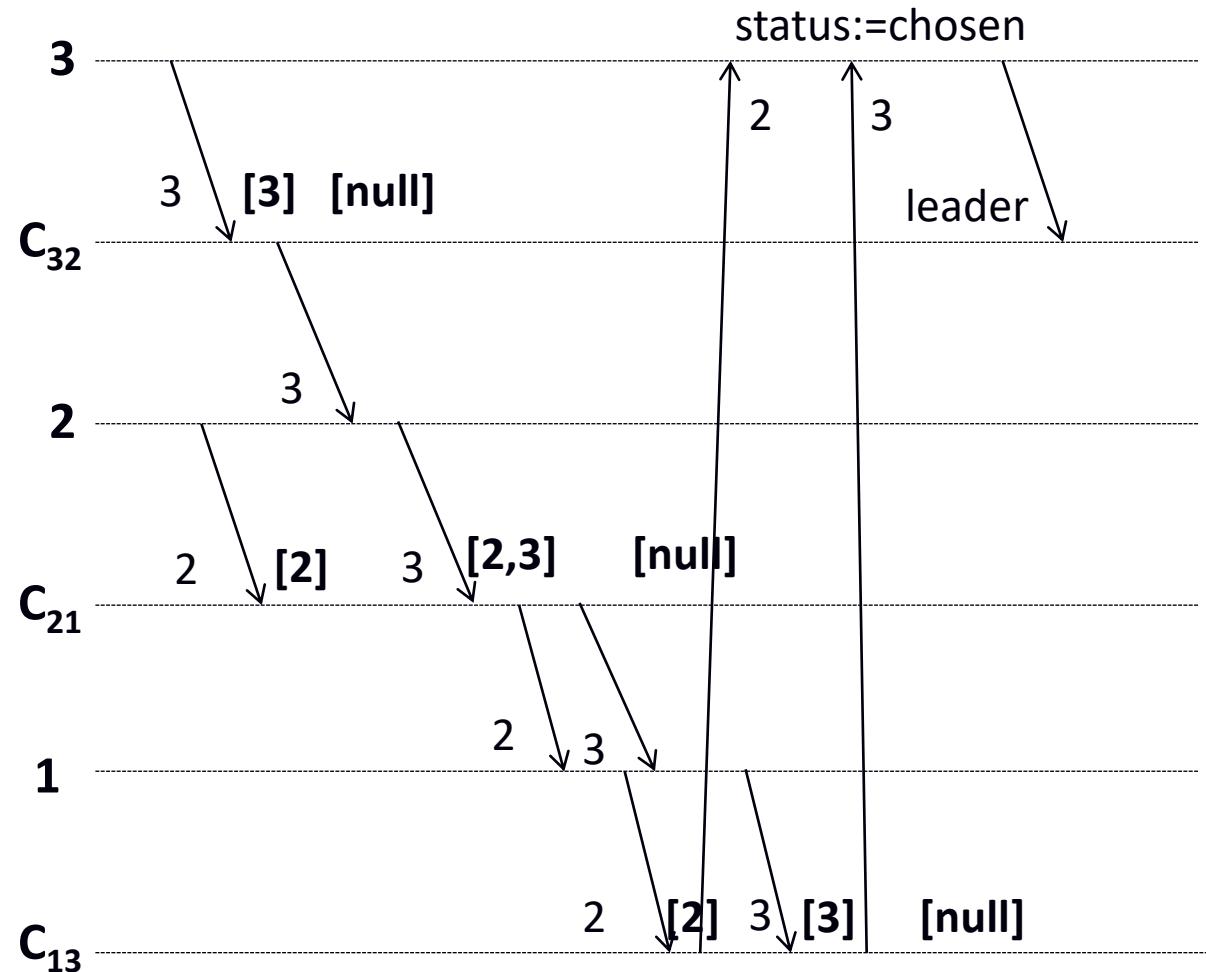
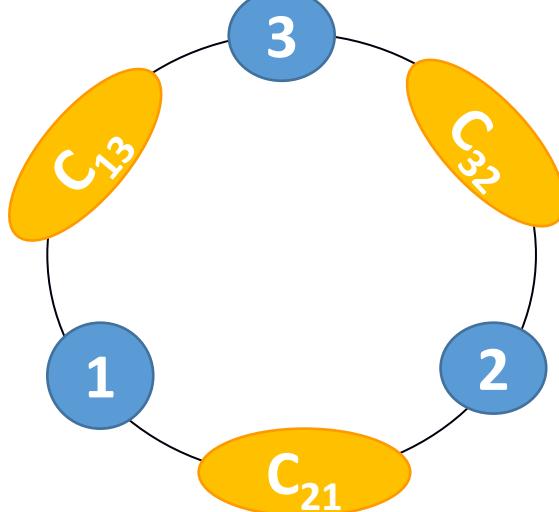
# Odabir vođe u asinkronoj mreži

## Definicija problema

- izabrati “vođu” među procesima u mreži
- samo 1 proces mijenja status u *leader*
- adaptacija sinkronog algoritma – svaki proces ima ulazni spremnik koji može primiti maksimalno  $n$  poruka (poruke se mogu gomilati zbog asinkronosti komunikacije)
- procesi: modelirani I/O automatom
- kanali: prepostavka je pouzdani FIFO



# Primjer asinkronog prstena i algoritma za odabir vođe



# Osnovni algoritam za asinkroni model

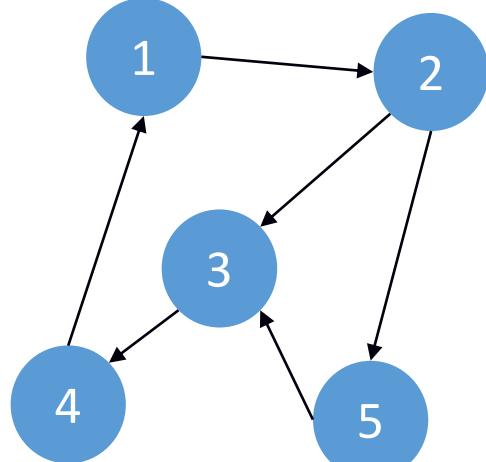
Definicija automata procesa  $P_i$

- input:  $receive(v)_{i-1,i}$ ,  $v$  je UID
- output:  $send(v)_{i,i+1}$ ;  $leader_i$
- $states_i$ :
  - $u$  – UID, inicijalno UID za  $i$
  - $send$  – FIFO queue UID-ova veličine  $n$ , inicijalno sadrži UID za  $i$
  - $status \in \{unknown, chosen, reported\}$ , inicijalno  $unknown$
- trans:
  - $send(v)_{i,i+1}$  – preduvjet:  $v$  je 1. element iz  $send$ , posljedica: briši  $v$  iz  $send$
  - $leader_i$  – preduvjet:  $status = chosen$ , posljedica:  $status := reported$
  - $receive(v)_{i-1,i}$ 
    - if  $v > u$ : add  $v$  to  $send$
    - if  $v = u$ : then  $status := chosen$
    - if  $v < u$ : do nothing

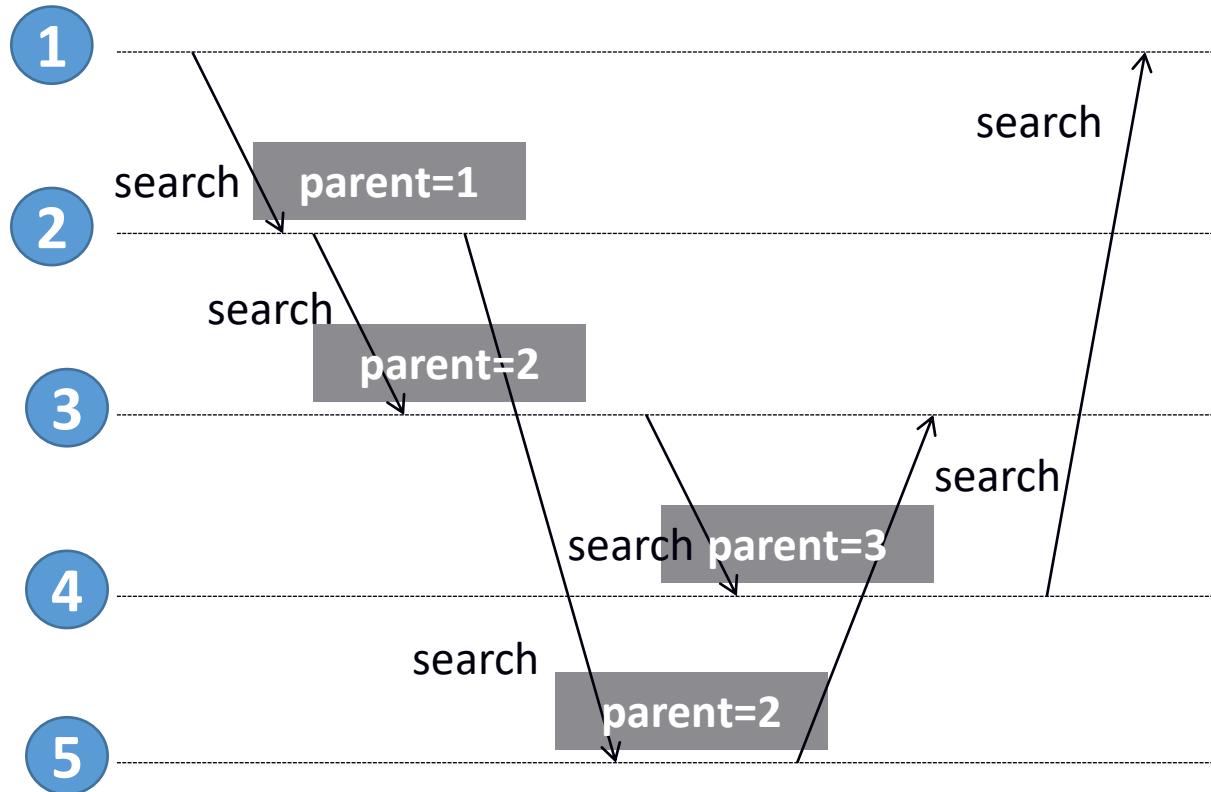
# Kreiranje stabla u asinkronoj mreži

- kreiranje stabla s definiranim korijenskim čvorom  $i_0$
- mreža je modelirana grafom  $G(V, E)$  koji je usmjeren i povezan
- procesi ne znaju dijametar mreže
- cilj: svaki proces treba odrediti prethodnika (*parent*)
- Skica algoritma *AsynchSpanningTree*
  - Inicijalno je  $i_0$  označen.  $i_0$  šalje *search* svim izlaznim susjedima. Kada proces primi *search* taj proces postaje označen, odabire jedan od susjeda od kojih je primio poruku za *parent* i šalje *search* svim svojim susjedima.

# Primjer algoritma *AsynchSpanningTree*



$$i_0 = 1$$



# AsynchSpanningTree

Definicija automata procesa  $P_i$

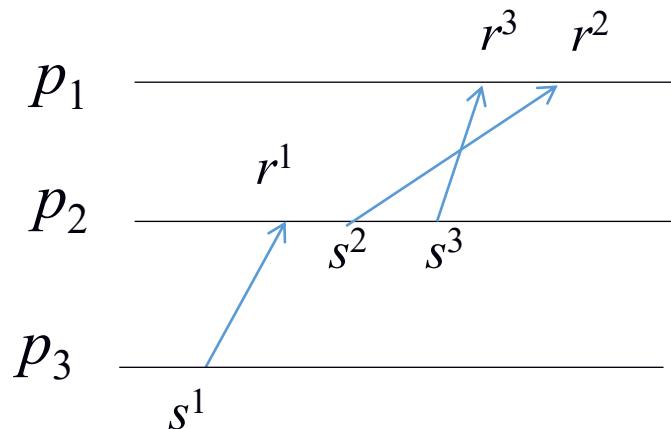
- input:  $receive("search")_{j,i}$ ,  $j \in nbrs$
- output:  $send("search")_{i,j}$ ,  $j \in nbrs$ ;  $parent(j)_i$ ,  $j \in nbrs$
- states:
  - $parent \in nbrs \cup \{null\}$ , inicijalno  $null$
  - $reported$  – boolean, inicijalno  $false$
  - za svaki  $j \in nbrs$  postoji  
 $send(j) \in \{search, null\}$ , inicijalno  $search$  ako je  $i = i_0$  inače  $null$
- trans:
  - $send("search")_{i,j}$  – preduvjet:  $send(j) = search$ , posljedica:  $send(j) := null$
  - $parent(j)_i$  – preduvjet:  $parent = j$ ,  $chosen = false$ , posljedica:  $reported := true$
  - $receive("search")_{j,i}$ 
    - if  $i \neq i_0$  and  $parent = null$   
 $parent := j$
    - for all  $k \in nbrs \setminus \{j\}$   
 $send(k) := search$

# Literatura

- A. D. Kshemkalyani, M. Singhal: *Distributed Computing: Principles, Algorithms, and Systems*, *Cambridge University Press*, 2008.  
poglavlja 2.1-2.4
- N. Lynch: *Distributed Algorithms*, *Morgan Kaufmann Publishers Inc.* 1996.
  - poglavlje 2: Modelling I: Synchronous Network Model
  - poglavlje 3: Leader Election in a Synchronous Ring (osnovni algoritam do 3.4)
  - poglavlje 8: Modelling II: Asynchronous System Model
  - poglavlje 15.1: Leader Election in a Ring

# Pitanja za učenje i ponavljanje

1. Za koje je svojstvo raspodijeljenih sustava značajna komunikacijska složenost algoritama? Zašto?
  - a) replikacijska transparentnost
  - b) skalabilnost
  - c) otvorenost
2. Objasnite model komunikacijskog kanala koji se temelji na uzročnoj slijednosti. Vrijedi li za primjer na slici CO ili non-CO i zašto?
3. Proučite formalnu definiciju algoritma *AsynchSpanningTree* na slajdu 60.





SVEUČILIŠTE U ZAGREBU



**Diplomski studij**  
**Računarstvo**  
Znanost o mrežama  
Programsko inženjerstvo i  
informacijski sustavi  
Računalno inženjerstvo  
**Ostali (slobodni izborni  
predmet)**

# Raspodijeljeni sustavi

**5. Procesi i komunikacija:**  
komunikacija porukama, model objavi-  
preplati, dijeljeni podatkovni prostor

Ak. god. 2022./2023.

# 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 daljnog 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/>*

# „Neizravna“ komunikacija

- engl. *indirect communication*
- komunikacija među procesima raspodijeljenog sustava **putem posrednika** bez direktne interakcije pošiljatelja i primatelja
- područja primjene
  - pokretne mreže i okoline
  - tokovi podataka (npr. financijski sustavi)
  - aplikacije u području Interneta stvari (senzori kontinuirano generiraju podatke)
- osigurava **prostornu i vremensku neovisnost procesa** (engl. *space uncoupling and time uncoupling*)

# Sadržaj predavanja

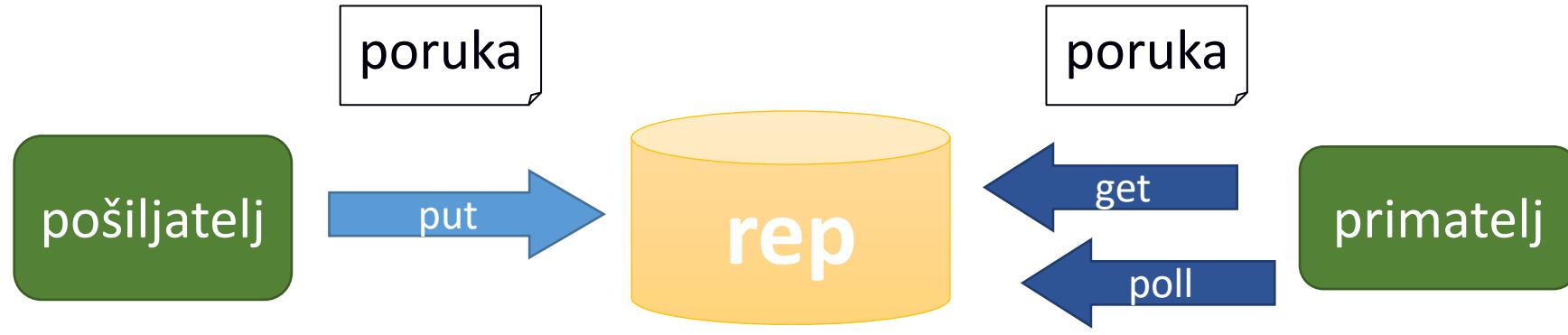
- Komunikacija porukama
- Model objavi-preplati
- Primjeri protokola za komunikaciju porukama: JMS i AMQP
- Dijeljeni podatkovni prostor

# Komunikacija porukama

engl. *message-queuing systems, Message-Oriented Middleware (MOM)*

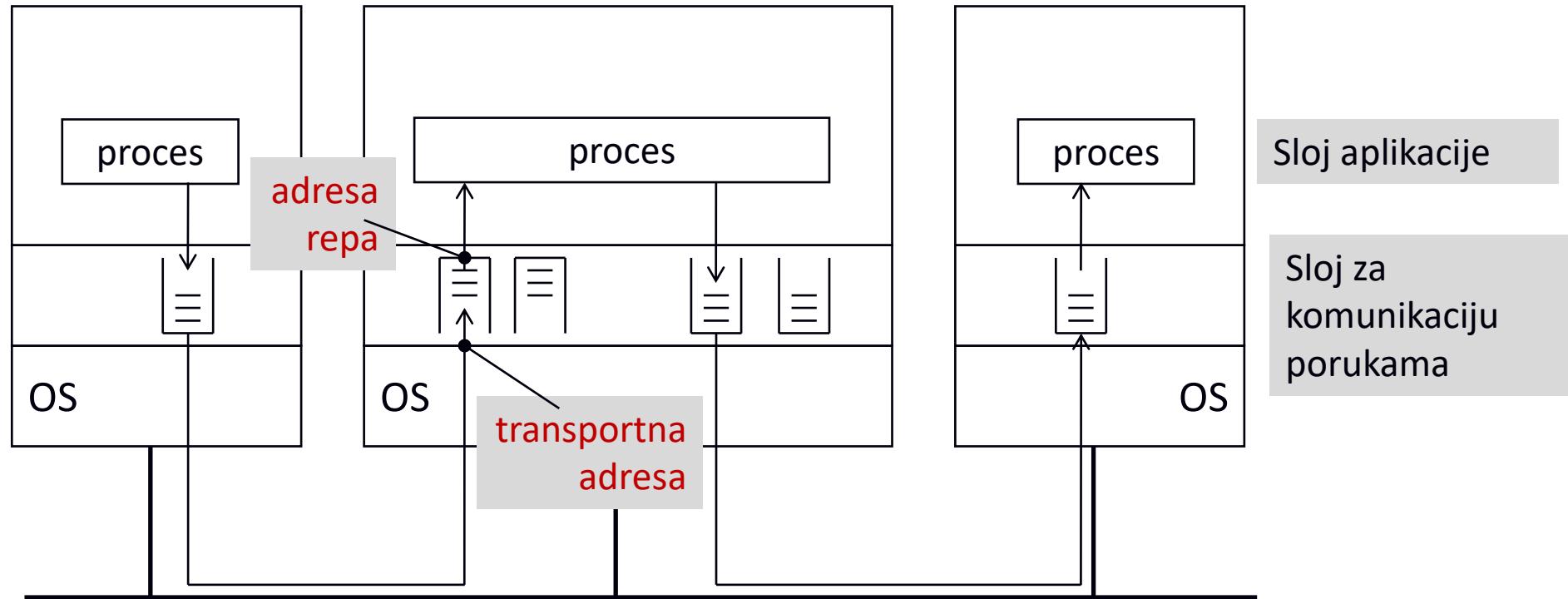
- Procesi/objekti komuniciraju razmjenjujući poruke.
- U komunikaciji sudjeluju izvor (pošiljatelj poruke) i odredište (primatelj poruke).
- Izvor šalje poruku, poruka se pohranjuje u rep koji je pridijeljen odredištu.
- Odredište čita poruku iz repa.
- Poruke sadrže podatke, važna je adresa odredišnog repa.
- Adresiranje se izvodi najčešće na nivou sustava, svaki rep ima jedinstven identifikator u sustavu.

# Izvođenje komunikacije porukama



- **put** – dodaj poruku u rep
- **get** – pročitaj poruku iz repa, primatelj je blokiran ako je rep prazan
- **poll** – provjeri postoji li poruke u repu i pročitaj prvu poruku ako takva postoji, primatelj nije blokiran

# Arhitektura sustava za komunikaciju porukama



# Obilježja komunikacije porukama

- vremenska neovisnost
  - primatelji i pošiljatelji ne moraju istovremeno biti aktivni, poruka se sprema u rep
- pošiljatelj mora znati identifikator odredišta, tj. njegovog repa
- komunikacija je **perzistentna**
- asinkrona komunikacija
  - pošiljatelj šalje poruku i nastavlja obradu neovisno o odgovoru od strane primatelja
- pokretanje komunikacije na načelu *pull*
  - primatelj provjerava postoji li poruka u repu

# Komunikacija je moguća i na načelu *push*

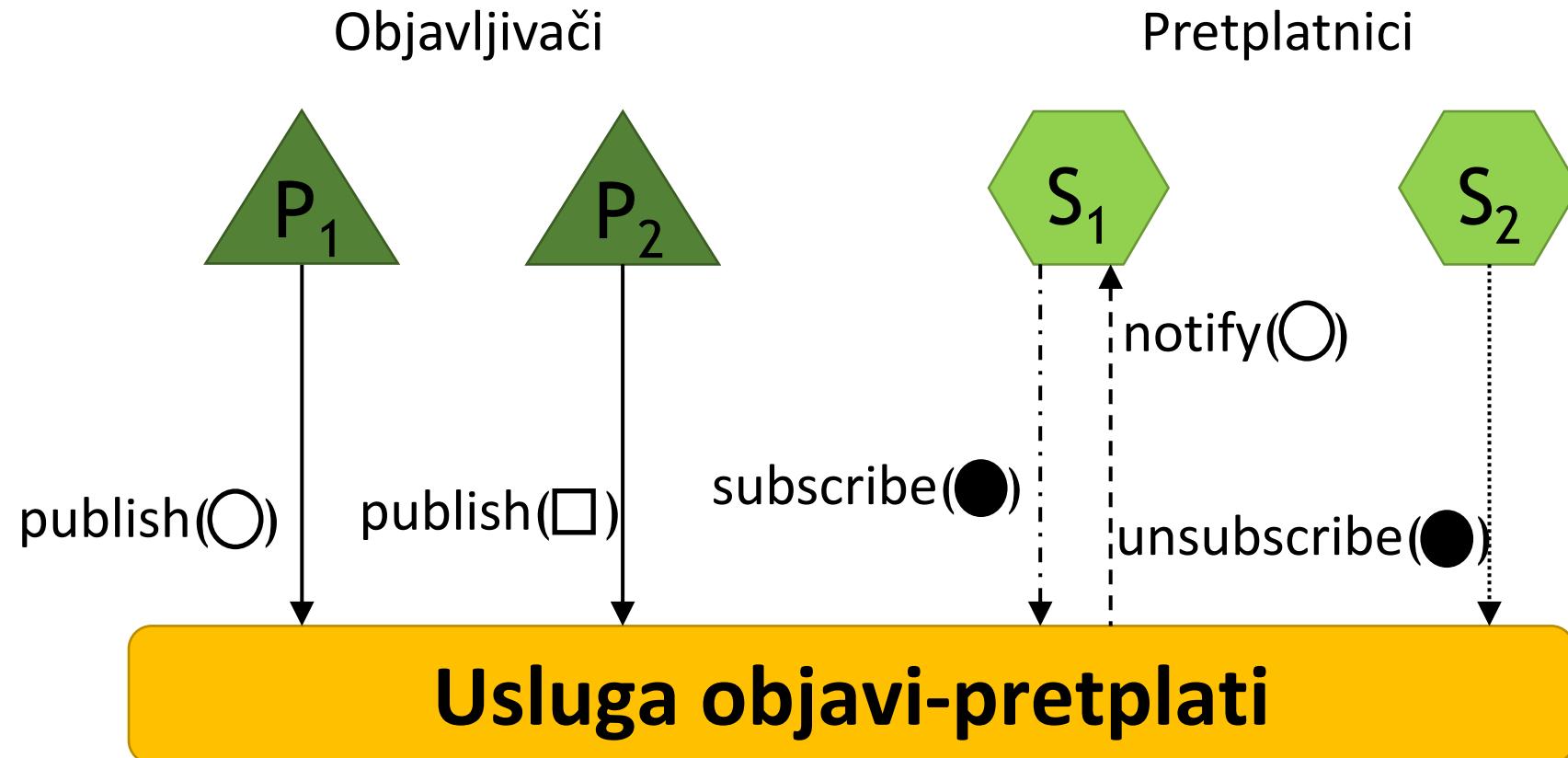


- **notify** – aktivna isporuka poruke iz repa primatelju po primitku poruke (na strani primateljskog procesa nužan je *listener thread*)

# Sadržaj predavanja

- Komunikacija porukama
- **Model objavi-preplati**
- Primjeri protokola za komunikaciju porukama: JMS i AMQP
- Dijeljeni podatkovni prostor

# Interakcija objavi-preplati



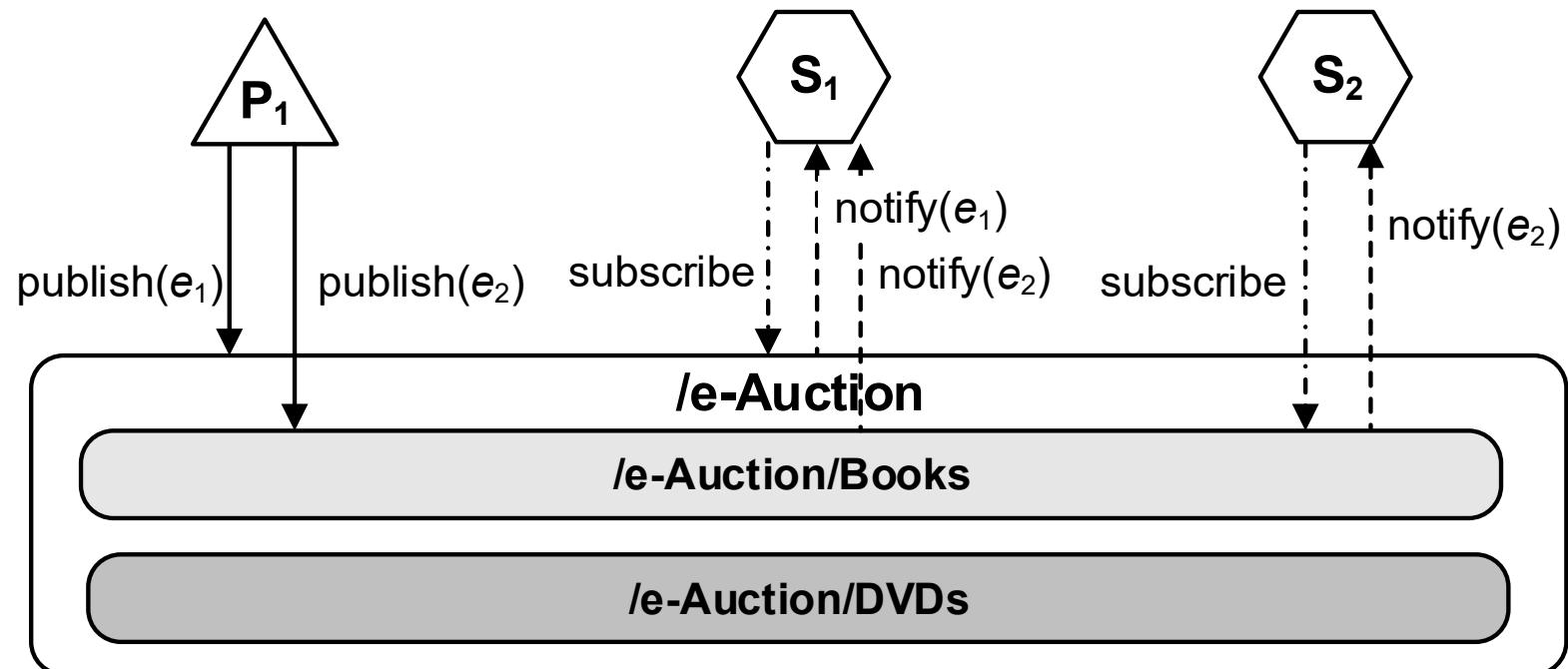
# Osnovni pojmovi

- objavljavači (*publishers*)
  - definiraju obavijesti (*notifications*)
- pretplatnici (*subscribers*)
  - pretplatama (*subscriptions*) i odjavama pretplata (*unsubscriptions*) izražavaju namjeru primanja određenog skupa obavijesti
- usluga objavi-pretplati:
  - sustav za obradu događaja (*event service – ES*)
  - obrađuje i pohranjuje primljene obavijesti/pretpiske/odjave pretplata
  - isporučuje obavijesti pretplatnicima prema njihovim aktivnim pretplatama
  - omogućuje persistenciju komunikaciju između objavljavača i pretplatnika

# Pretplate

- „kontinuirani upiti”
- pretplata na kanal/temu (engl. *topic-based subscription*)
  - kanal – logička veza između izvora i odredišta koja služi za tematsko grupiranje obavijesti (npr. vrijeme, sport, itd.)
  - hijerarhijski odnos kanala (npr. vrijeme u Europi, Hrvatskoj, Zagrebu)
- pretplata na sadržaj (engl. *content-based subscription*)
  - pretplata se definira ovisno o svojstvima i sadržaju obavijesti (skup atributa i vrijednosti)

# Pretplata na kanal



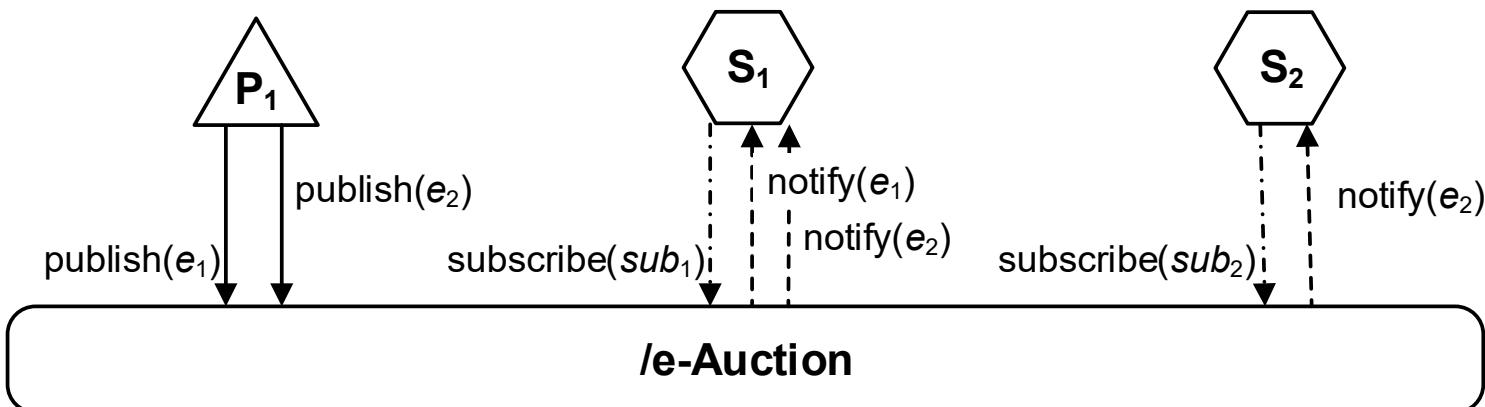
# Pretplata na sadržaj

```
e1 = (category = "books"
 & author = "D. Adams"
 & title = "The Hitchhiker's Guide through the Galaxy"
 & price = 9.99 EUR)
```

```
e2 = (category = "books"
 & author = "J.R.R. Tolkien"
 & title = "The Lord of the Rings"
 & price = 19.99 EUR)
```

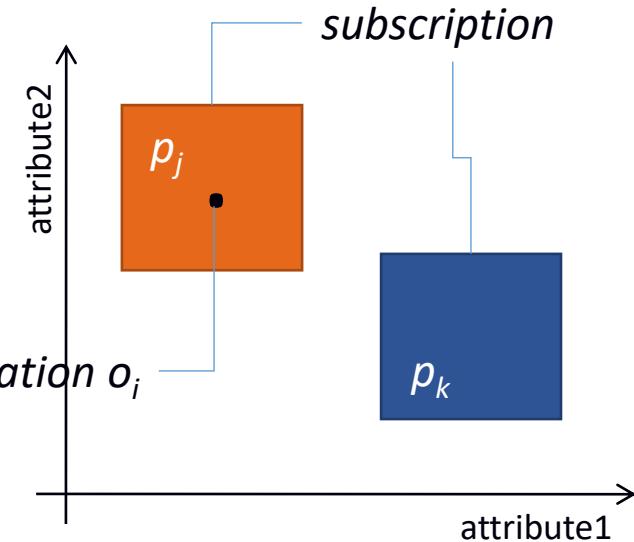
```
sub1 = (category == "books"
 & price < 20 EUR)
```

```
sub2 = (category == "books" &
 author == "J.R.R. Tolkien"
 & price < 20 EUR)
```



# Primjer obavijesti/preplate (strukturirani podaci)

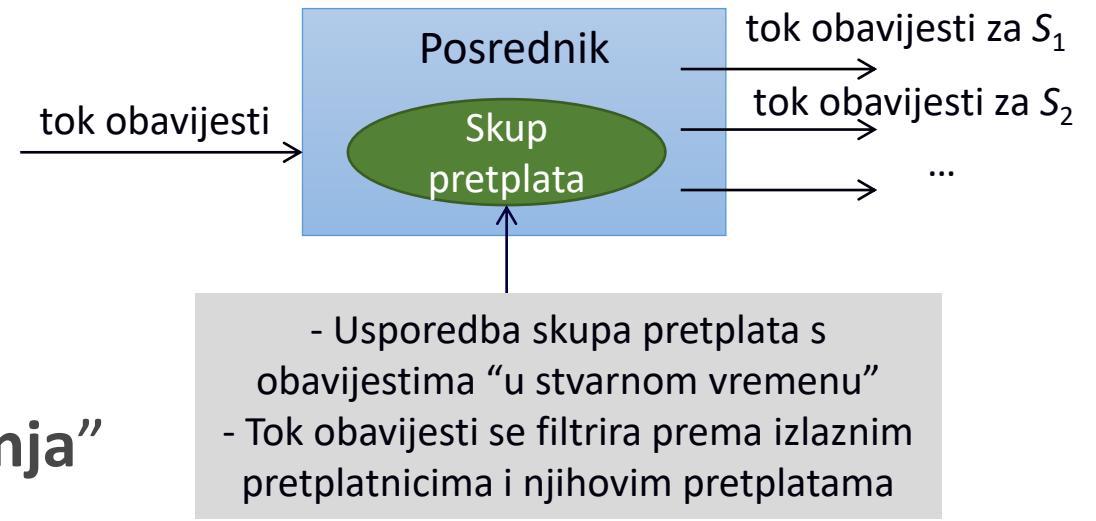
- **obavijest** je najčešće točka u višedimenzionalnom prostoru
  - npr. očitanje senzora, cijena dionice, oglas, vijest
  - objavljivači kontinuirano objavljaju nove obavijesti (često ograničene valjanosti)
- **preplata** je potprostor višedimenzionalnog prostora
  - definira se kao Booleova funkcija nad parom (obavijest, preplata)
  - za preplatu kažemo da **prekriva** obavijest kada obavijest zadovoljava uvjete preplate, tj.  $f(o_i, p_j) = T$



Poseban implementacijski izazov:  
učinkovita usporedba objave sa  
skupom preplate jer je u stvarnom  
vremenu potrebno odrediti  
podskup preplate koje prekrivaju  
obavijest kako bi se isporučila svim  
zainteresiranim preplatnicima

# Usporedba obavijesti sa skupom pretplatama

- Sustav objavi-preplati održava skup pretplata koje se uspoređuju s novoobjavljenom obavijesti



- Usporedba ispituje svojstvo "**prekrivanja**" obavijesti pretplatom

- Pretplata "prekriva" obavijest kada obavijest zadovoljava sve uvjete definirane pretplatom
- Pretplata  $[a < 10, b \leq 20]$  prekriva obavijest  $[a=5, b=20]$ , ali ne prekriva obavijest  $[a=5, b=25]$
- Pretplata  $[\text{sveučilište==Zagreb}, \text{fakultet==FER}]$  prekriva obavijest  $[\text{sveučilište=Zagreb}, \text{fakultet=FER}, \text{vijest=Proslavljen dan FER-a!}]$

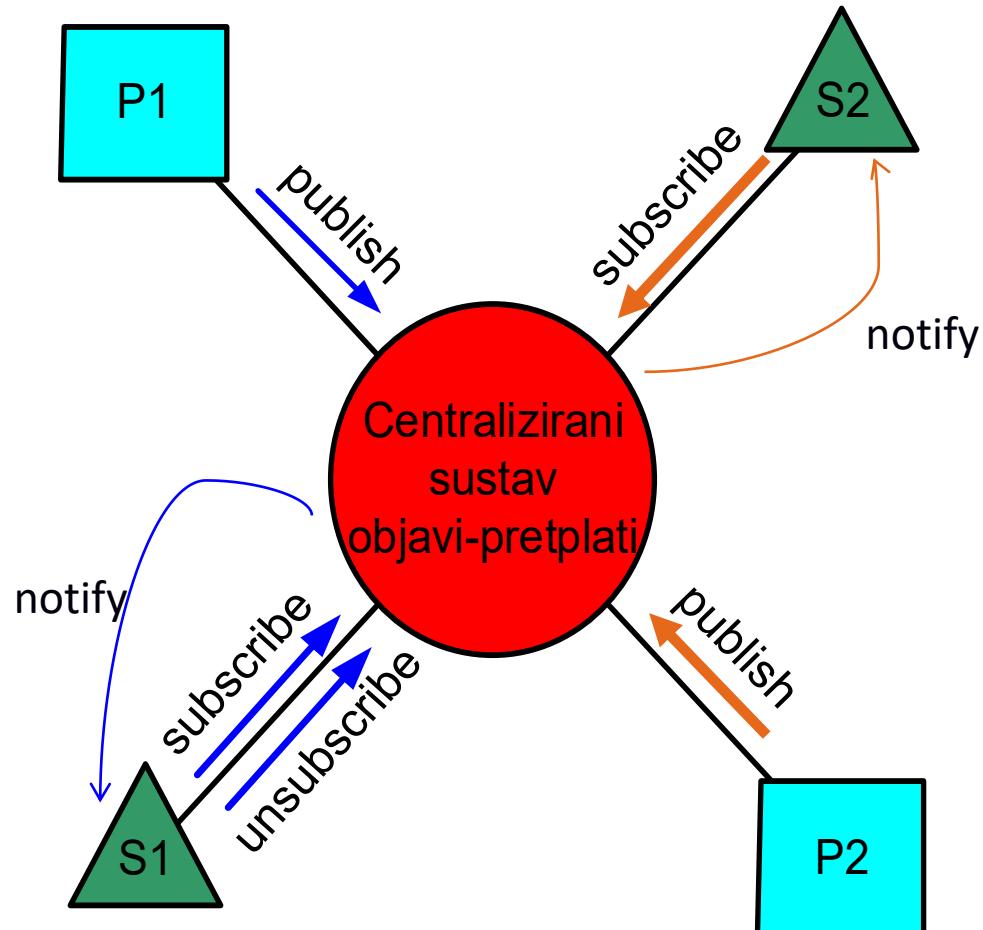
# Kvaliteta usluge za komunikaciju porukama

- Vezana uz garanciju isporuke poruke
  - najviše jednom (*at-most-once*) – ne postoje mehanizmi koji osiguravaju isporuku poruke u slučaju ispada
  - barem jednom (*at-least-once*) – postoje mehanizmi koji će u slučaju ispada ponoviti operaciju, moguće je da će primatelj primiti poruku više puta
  - sigurno jednom (*exactly once*) – primatelj će primiti poruku samo jednom
- Poruke mogu biti perzistentne (imaju vremenski definiran period valjanosti) i neperzistentne poruke („vrijede” u trenutku u kome su definirane)

# Arhitektura usluge objavi-pretplati

- Centralizirana
  - svi objavljavači i pretplatnici razmjenjuju obavijesti i definiraju preplate preko jednog poslužitelja posrednika
  - poslužitelj pohranjuje sve preplate i prosljeđuje obavijesti
- Raspodijeljena
  - skup poslužitelja, svaki je poslužitelj zadužen za objavljavače i pretplatnike u svojoj domeni
  - algoritmi za usmjerenje informacija o preplatama i usmjerenje obavijesti

# Centralizirana arhitektura

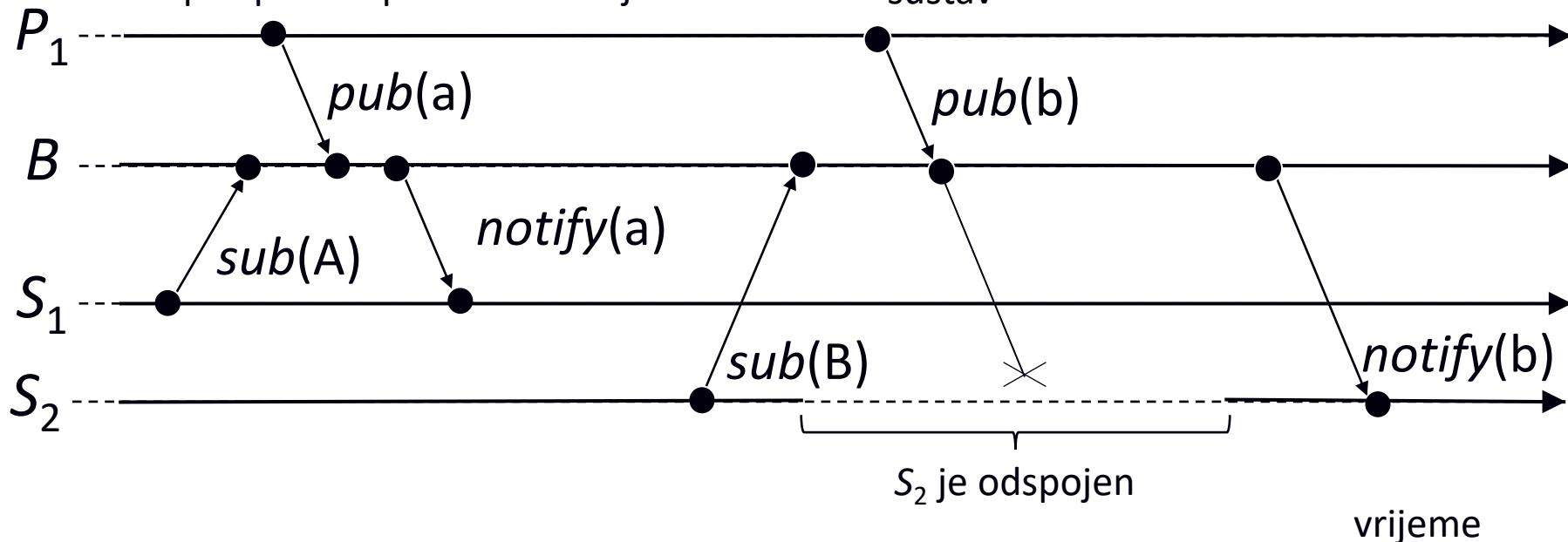


# Primjer raspodijeljenog izvođenja

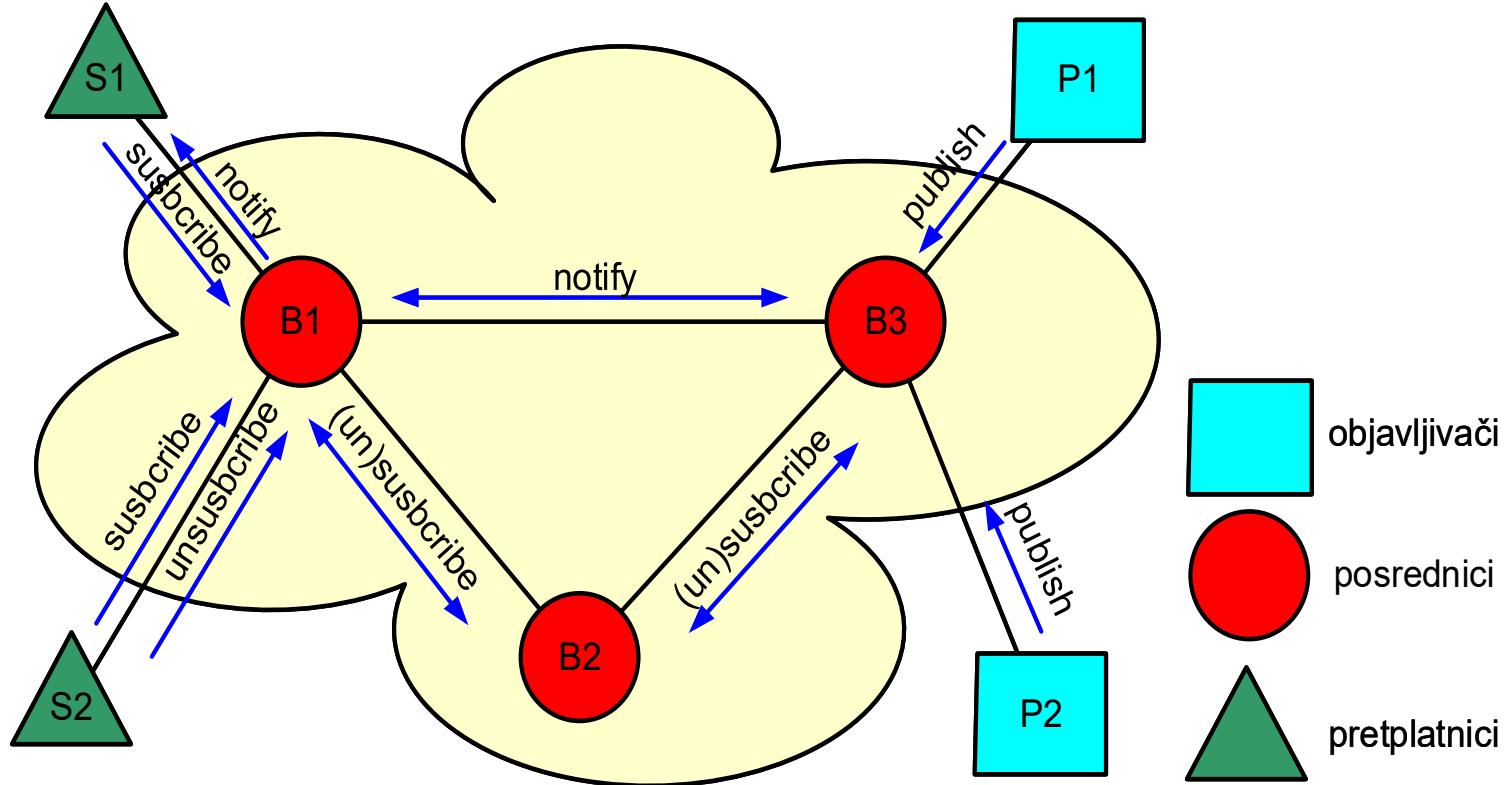
(asinkroni model, centralizirano)

primjer tipičnog slijeda događaja,  
 $sub \rightarrow pub \rightarrow notify$   
kada pretplata A prekriva obavijest a

primjer isporuke **perzistentne obavijesti**,  
B je prezistentna pretplata (*durable subscription*),  
pa posrednik čuva *matching* obavijesti koje ne može  
isporučiti i isporučuje ih kada se  $S_2$  ponovno spoji u  
sustav



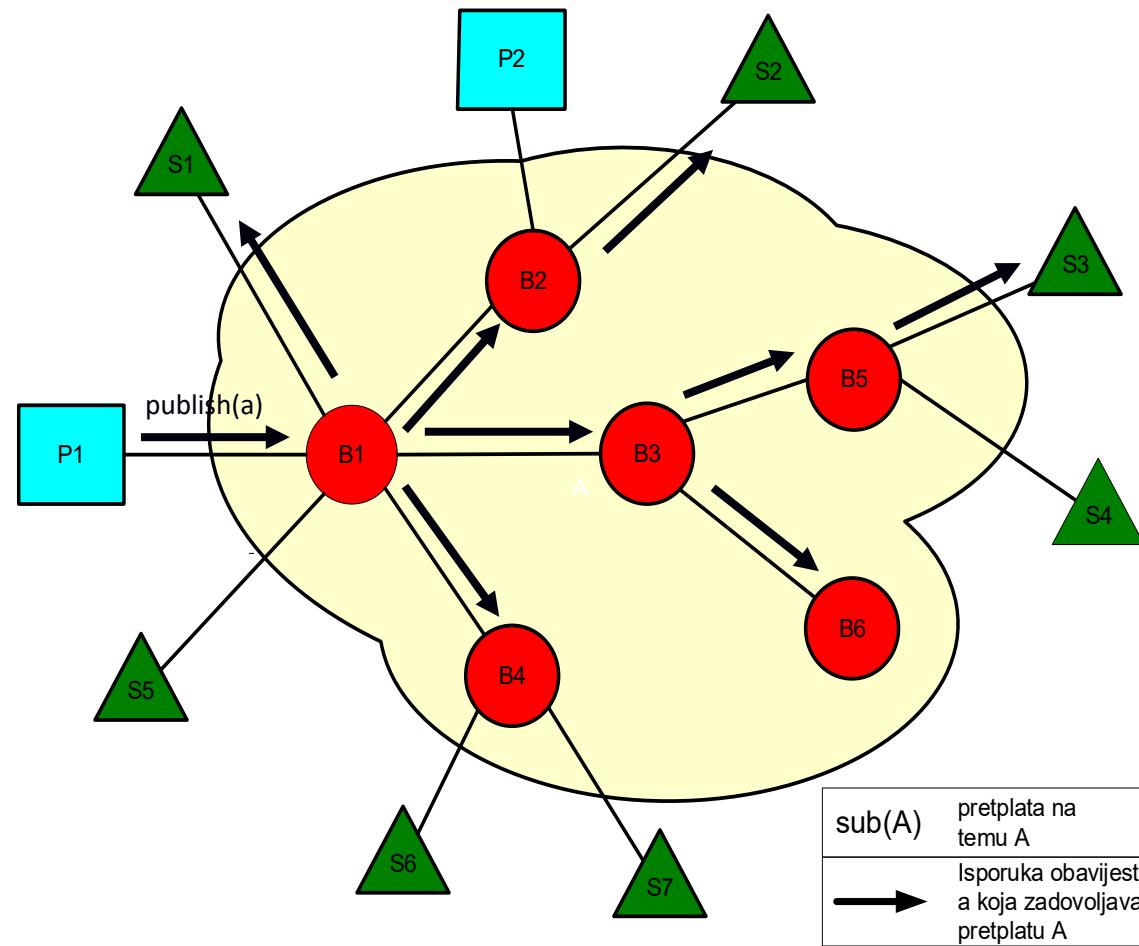
# Raspodijeljena arhitektura



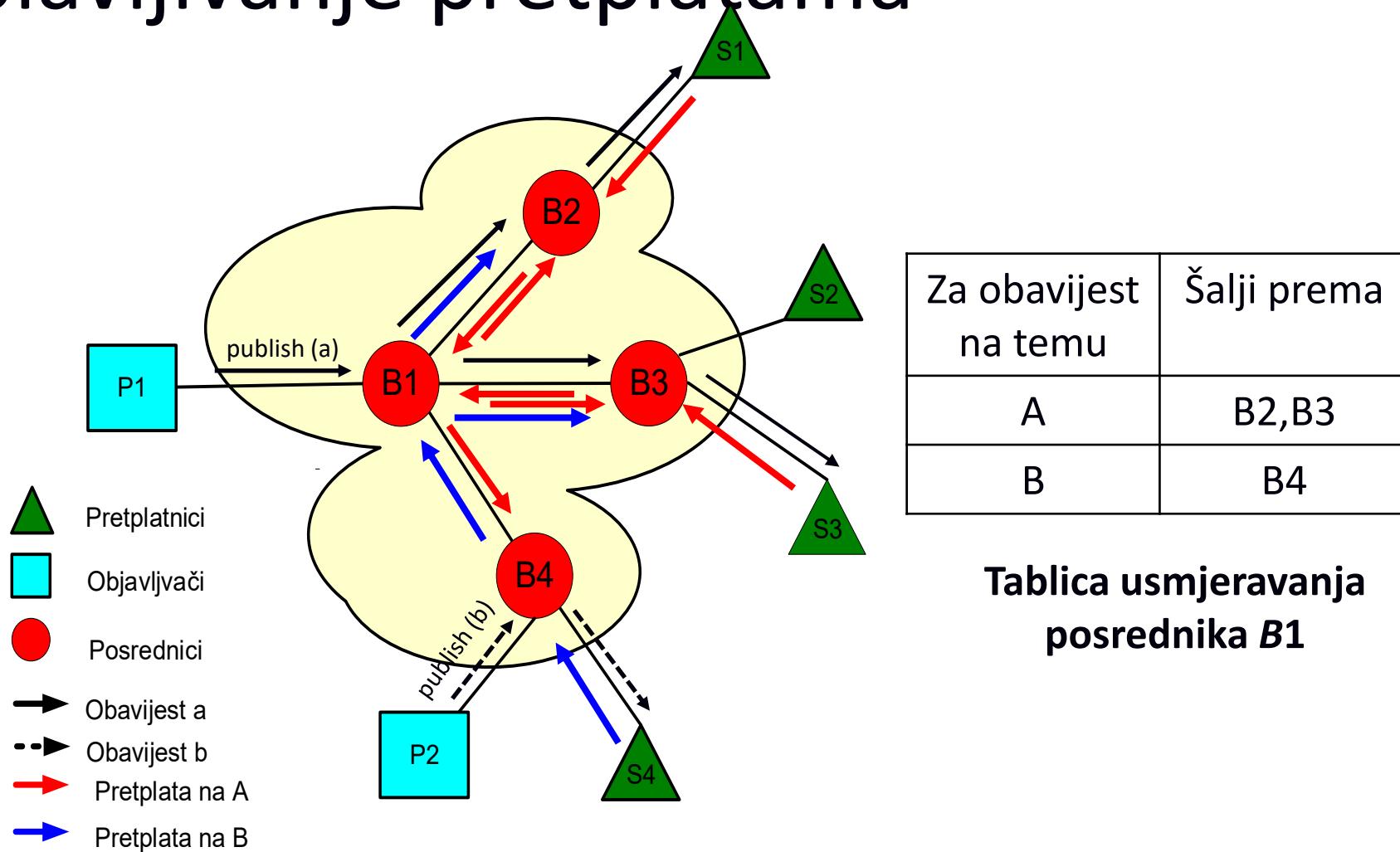
# Osnovna načela usmjeravanja

- preplavljanje
  - svaka primljena poruka (obavijest, pretplata ili odjava pretplate) prosljeđuje se svim susjedima osim onome od koga je poruka primljena
  - posrednik posjeduje tablicu usmjeravanja koja sadrži informacije o svim susjednim posrednicima i lokalnim pretplatnicima
- filtriranje poruka
  - filtriranje poruka se izvodi usporedbom obavijesti s aktivnim pretplatama koje definiraju svojstva obavijesti za koje je pretplatnik zainteresiran
  - osnovni cilj je isporuka samo onih obavijesti koje pretplatnika zanimaju
  - omogućuje i smanjenje prometa u mreži posrednika zbog sprječavanja širenja obavijesti "nezainteresiranim" posrednicima

# Preplavljanje obavijestima



# Preplavljanje pretplatama



# Obilježja modela objavi-preplati

- **vremenska neovisnost**
  - objavljavač i pretplatnici ne moraju istovremeno biti aktivni, posrednik pohranjuje poruku
- objavljavač ne mora znati identifikator pretplatnika (**anonimnost**), o tome se brine posrednik – **prostorna neovisnost**
- komunikacija je **perzistentna**
- **asinkrona komunikacija**
  - objavljavač šalje poruku i nastavlja obradu neovisno o odgovoru od strane odredišta – **vremenska neovisnost**
- pokretanje komunikacije na načelu **push**
  - objavljavač šalje poruku posredniku koji je prosljeđuje pretplatnicima bez prethodnog eksplicitnog zahtjeva

# Obilježja modela objavi-preplati (2)

- personalizacija primljenog sadržaja
  - filtriranje objavljenih poruka prema pretplatama
- proširivost sustava
  - dodavanje novog objavljivača ili pretplatnika ne utječe na ostale strane u komunikaciji
- skalabilnost
  - raspodijeljena arhitektura

# Sadržaj predavanja

- Komunikacija porukama
- Model objavi-preplati
- **Primjeri rješenja za komunikaciju porukama: JMS i AMQP**
- Dijeljeni podatkovni prostor

# JMS

## Java Message Service

JMS 2.0, Java Community Process, 21.05.2013.

<https://java.net/projects/jms-spec/pages/JMS20FinalRelease>

Specifikacija otvorenog protokola za komunikaciju porukama i komunikaciju na načelu objavi-preplati.

JMS API definira skup sučelja i pripadajuću semantiku koja omogućuje programima pisanim u Javi komunikaciju razmjenom poruka i na načelu objavi-preplati.

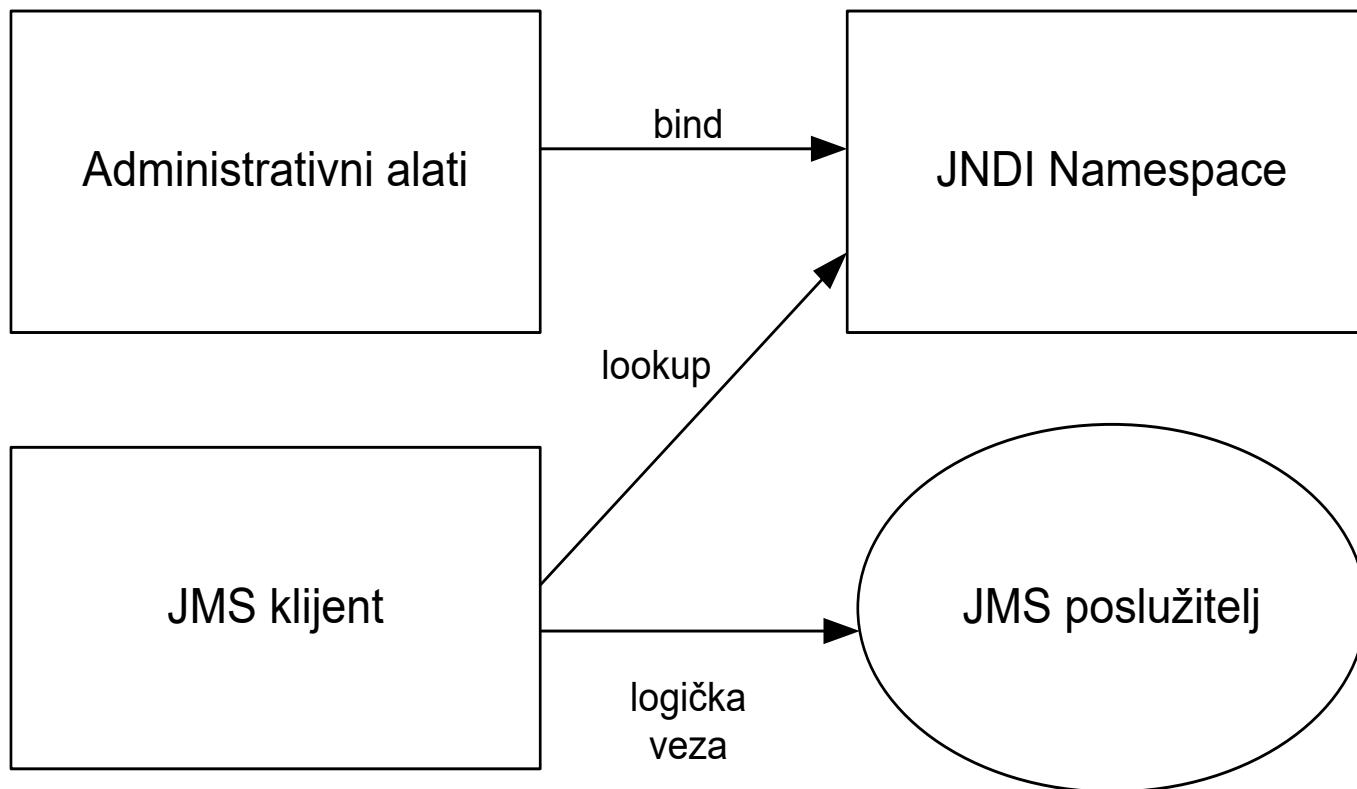
Popularne implementacije: Apache ActiveMQ, IBM WebSphereMQ, HornetQ, OpenJMS

# Arhitektura JMS-a (1)

- JMS poslužitelj
  - sustav za razmjenu poruka koji implementira JMS sučelja i nudi administrativne i kontrolne usluge
- Klijent
  - bilo koji objekt, proces ili aplikacija koja stvara ili konzumira poruke
- Poruka (*message*)
  - objekt koji se sastoji od zaglavljiva koje prenosi identifikacijske i adresne informacije i tijela koje prenosi podatke
- Odredište (*destination*)
  - objekt koji sadrži informacije o odredištu poruke

# Arhitektura JMS-a (2)

JNDI  
(Java Naming and Directory Interface)

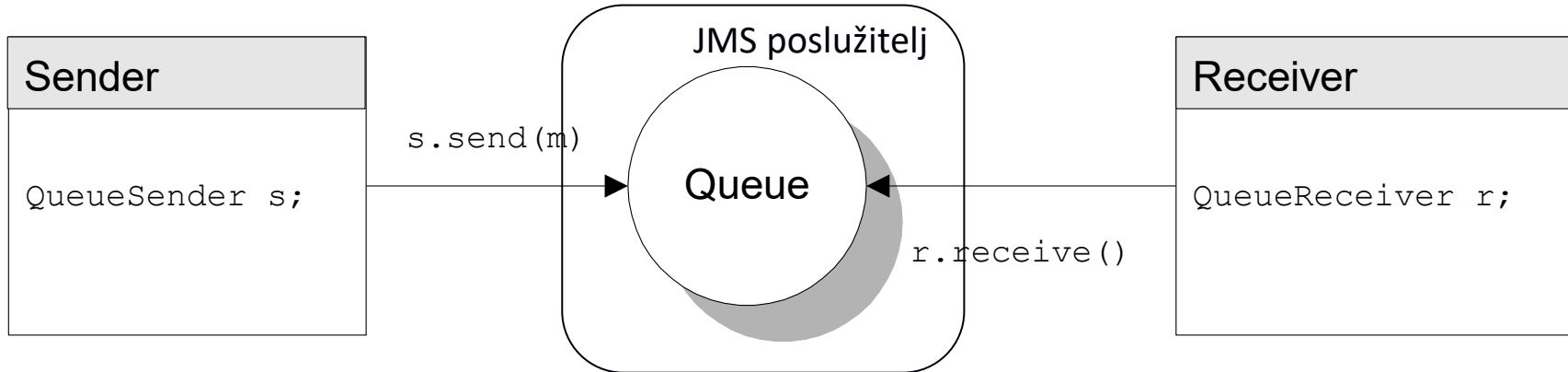


# Modeli JMS-a

JMS implementira sljedeće modele za komunikaciju porukama i obavijestima

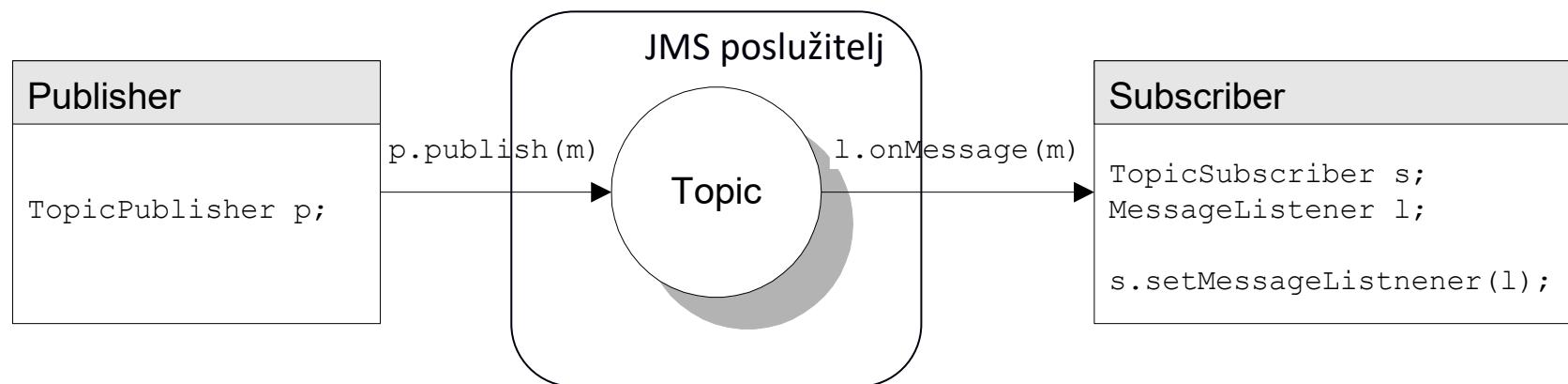
- *Point-to-point*
  - komunikacija porukama, jedna poruka za jedno odredište
- *Publish/subscribe*
  - objavi-preplati, jedna poruka za skup zainteresiranih pretplatnika

# Point-to-point



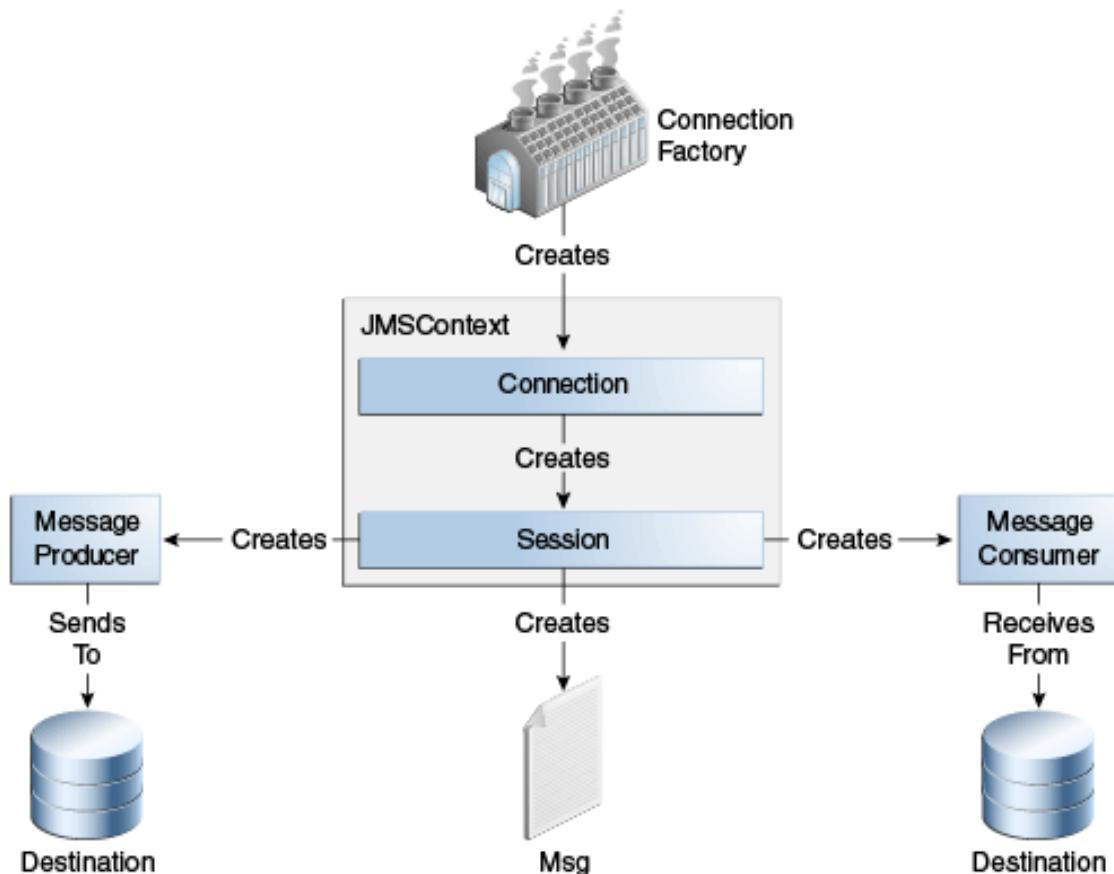
1. Klijent s koji šalje poruku m poziva s.send(m). Poruka se sprema u repu.
2. Klijent koji prihvata poruku mora provjeriti postoji li poruka u repu. Poziva r.receive().
3. Poruka se briše iz repa i šalje klijentu.

# Publish/subscribe



1. Tijekom inicijalizacije pretplatnik registrira instancu klase koja implementira sučelje `MessageListener` pozivajući `s.setMessageListener(l)`. Topic pamti sve preplate.
2. Izvor objavljuje poruku `m` sa `p.publish(m)`.
3. Topic isporučuje poruku pretplatniku pozivajući `l.onMessage(m)`.

# Programski model JMS API-ja



Izvor: **The Java EE 7 Tutorial**  
Poglavlje 45: Java Message Service Concepts

# Sučelja JMS-a (1)

|                   |                        |                        |
|-------------------|------------------------|------------------------|
| Nad-sučelje       | Point-to-point         | Publish/subscribe      |
| Destination       | Queue                  | Topic                  |
| ConnectionFactory | QueueConnectionFactory | TopicConnectionFactory |
| Connection        | QueueConnection        | TopicConnection        |

- ◆ **Destination**
  - administrirani objekt
  - predstavlja odredište - identitet ili adresu repa/teme.
- ◆ **ConnectionFactory**
  - administrirani objekt koji sadrži konfiguracijske parametre
  - klijenti ga koriste za stvaranje objekta *Connection*.
- ◆ **Connection**
  - predstavlja aktivnu konekciju prema JMS poslužitelju
  - klijenti ga koriste za stvaranje sjednice (*Session*).

# Sučelja JMS-a (2)

|                 |                |                   |
|-----------------|----------------|-------------------|
| Nad-sučelje     | Point-to-point | Publish/subscribe |
| Session         | QueueSession   | TopicSession      |
| MessageProducer | QueueSender    | TopicPublisher    |
| MessageConsumer | QueueReceiver  | TopicSubscriber   |

- ◆ **Session**
  - dretva u kojoj se primaju odnosno šalju poruke
  - klijenti koriste sesiju da stvore jedan ili više *MessageProducer* ili *MessageConsumer* objekata
- ◆ **MessageProducer**
  - objekt za slanje poruka odredištu
- ◆ **MessageConsumer**
  - objekt za primanje poruka koje su poslane odredištu

# Poruke JMS-a

- zaglavje
  - skup definiranih polja koja sadrže vrijednosti koje identificiraju i usmjeravaju poruku
- svojstva poruke
  - opcionalni parovi ime-vrijednost, a vrijednost može biti *boolean*, *byte*, *short*, *int*, *long*, *float*, *double* ili *String*
- tijelo poruke
  - *TextMessage* sadrži *java.lang.String*. (npr. za slanje XML dokumenata)
  - *StreamMessage* za niz Javinih primitiva.
  - *MapMessage* kada tijelo sadrži skup parova ime-vrijednost.
  - *ObjectMessage* sadrži Java objekt.
  - *ByteMessage* za tijelo koje sadrži niz neinterpretiranih *byte*-ova.

# Literatura: JMS

## The Java EE 7 Tutorial

- Chapter 45: Java Message Service Concepts

<https://docs.oracle.com/javaee/7/tutorial/jms-concepts.htm>

- What's New in JMS 2.0, Part One: Ease of Use

<http://www.oracle.com/technetwork/articles/java/jms20-1947669.html>

- What's New in JMS 2.0, Part Two—New Messaging Features

<http://www.oracle.com/technetwork/articles/java/jms2messaging-1954190.html>

- Enterprise Integration Patterns

<http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.htm>

!

# Primjer 1: JMS

## 1. Perform a JNDI lookup of the ConnectionFactory and Queue:

```
/* Create a JNDI API InitialContext object if none exists yet. */
Context jndiContext = null;
try {
 jndiContext = new InitialContext();
} catch (NamingException e) {
 System.out.println("Could not create JNDI API " + "context: " + e.toString());
 System.exit(1);
}

/* Look up connection factory and destination. If either does not exist, exit. */
QueueConnectionFactory connectionFactory = null;
Queue queue = null;
try {
 connectionFactory = (QueueConnectionFactory)
 jndiContext.lookup("jms/QueueConnectionFactory");
 queue = (Queue) jndiContext.lookup("queue");
}
catch (Exception e) {
 System.out.println("JNDI API lookup failed: " + e.toString());
 e.printStackTrace();
 System.exit(1);
}
```

# Queue Sender (2)

## 2. Create a Connection and a Session:

```
QueueConnection connection =
 connectionFactory.createQueueConnection();
QueueSession session = connection.createQueueSession(false,
 Session.AUTO_ACKNOWLEDGE);
```

## 3. Create a QueueSender and a TextMessage:

```
QueueSender sender = session.createSender(queue);
TextMessage message = session.createTextMessage();
```

# Queue Sender (3)

4. Send one or more messages to the queue:

```
for (int i = 0; i < NUM_MSGS; i++) {
 message.setText("This is message " + (i + 1));
 System.out.println("Sending message: " +
 message.getText());
 sender.send(message);
}
```

5. Send an empty control message to indicate the end of the message stream. Sending an empty message of no specified type is a convenient way to indicate to the consumer that the final message has arrived.

```
sender.send(session.createMessage());
```

# Queue Sender (4)

6. Close the connection in a finally block, automatically closing the session and QueueSender:

```
} finally {
 if (connection != null) {
 try {
 connection.close();
 } catch (JMSEException e) {}
 }
}
```

# Queue Receiver (1)

1. Performs a JNDI lookup of the ConnectionFactory and Queue.
2. Creates a Connection and a Session.
3. Creates a QueueReceiver:

```
QueueReceiver receiver = session.createReceiver(queue);
```

4. Starts the connection, causing message delivery to begin:

```
connection.start();
```

# Queue Receiver (2)

5. Receives the messages sent to the destination until the end-of-message-stream control message is received:

```
while (true) {
 Message m = receiver.receive();
 if (m != null) {
 if (m instanceof TextMessage) {
 message = (TextMessage) m;
 System.out.println("Reading message: " +
 message.getText());
 } else {
 break;
 }
 }
}
```

- Since the control message is not a TextMessage, the receiving program terminates the while loop and stops receiving messages after the control message arrives.
6. Closes the connection in a finally block, automatically closing the session and QueueReceiver.

# TopicPublisher

1. Perform a JNDI lookup of the TopicConnectionFactory and Topic.
2. Create a TopicConnection and a TopicSession.
3. Create a TopicPublisher and a TextMessage.
4. Send one or more messages to the topic.
5. Send an empty control message to indicate the end of the message stream.
6. Close the connection in a finally block, automatically closing the session and TopicPublisher.

# TopicSubscriber (1)

1. Perform a JNDI lookup of the TopicConnectionFactory and Topic.
2. Create a TopicConnection and a TopicSession.
3. Create a TopicSubscriber.
4. Create an instance of the TextListener class and registers it as the message listener for the TopicSubscriber:

```
listener = new TextListener();
subscriber.setMessageListener(listener);
```

5. Start the connection, causing message delivery to begin.

# TopicSubscriber (2)

6. Listen for the messages published to the topic, stopping when the user types the character q or Q:

```
System.out.println("To end program, type Q or q, " +
 "then <return>");

InputStreamReader = new InputStreamReader(System.in);
while (!((answer == 'q') || (answer == 'Q'))) {
 try {
 answer = (char) inputStreamReader.read();
 } catch (IOException e) {
 System.out.println("I/O exception: "
 + e.toString());
 }
}
```

7. Close the connection, which automatically closes the session and TopicSubscriber.

# Message Listener

1. When a message arrives, the `onMessage` method is called automatically.
2. The `onMessage` method converts the incoming message to a `TextMessage` and displays its content. If the message is not a text message, it reports this fact:

```
public void onMessage(Message message) {
 TextMessage msg = null;

 try {
 if (message instanceof TextMessage) {
 msg = (TextMessage) message;
 System.out.println("Reading message: " +
 msg.getText());
 } else {
 System.out.println("Message is not a " +
 "TextMessage");
 }
 } catch (JMSEException e) {
 System.out.println("JMSEException in onMessage(): " +
 e.toString());
 } catch (Throwable t) {
 System.out.println("Exception in onMessage(): " +
 t.getMessage());
 }
}
```

# AMQP

## Advanced Message Queuing Protocol

- Specifikacija otvorenog protokola za komunikaciju porukama i komunikaciju na načelu objavi-preplati.
- Popularan protokol i raširena primjena zbog niza implementacija: RabbitMQ, OpenAMQ, StortMQ, Apache QPid...

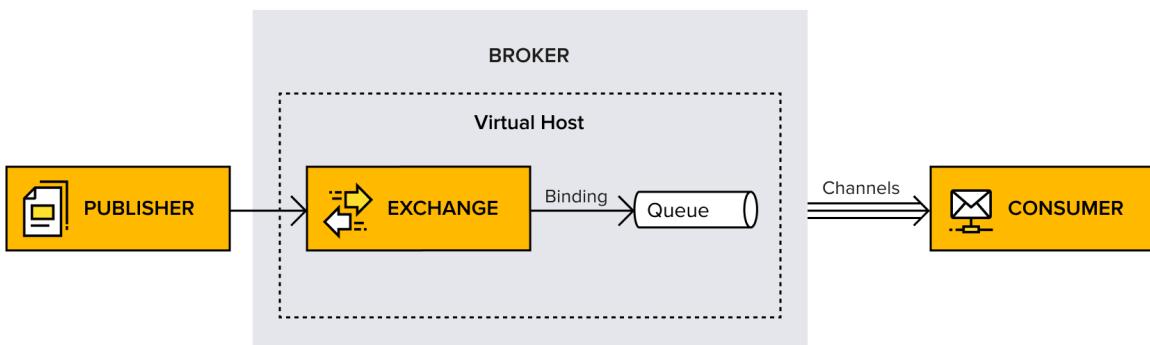
Version 1.0, OASIS Standard, 29.10.2012.

<http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html>

Version 0.91, specifikacija [AMQP WG](#) iz 2008.

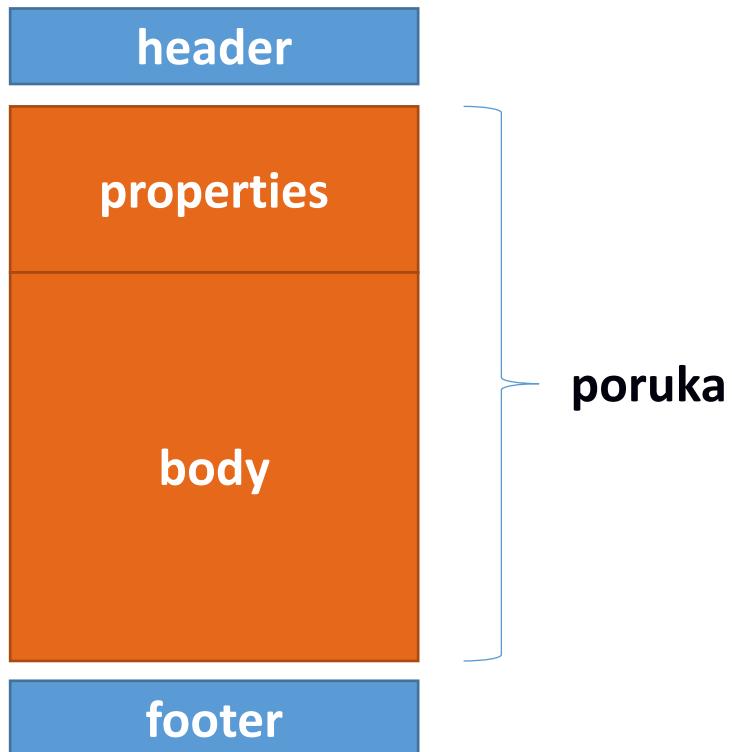
<http://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf>.

# Osnovni koncepti AMQP-a (v0.91)



- *Exchange*: entitet unutar posrednika koji usmjerava poruke u repove
- *Virtual host*: logički kontejner posrednika, odvaja različite aplikacije koje koriste istu instancu RabbitMQ posrednika
- *Channel*: virtualna veza unutar TCP konekcije koja povezuje posrednika s klijentom
- *Binding*: virtualna poveznica između *exchange-a* i repa
- *Publisher* (ili *producer*): objavljuje poruke na *exchange*
- *Consumer*: vezan je uz rep (*queue*) i definira *binding*

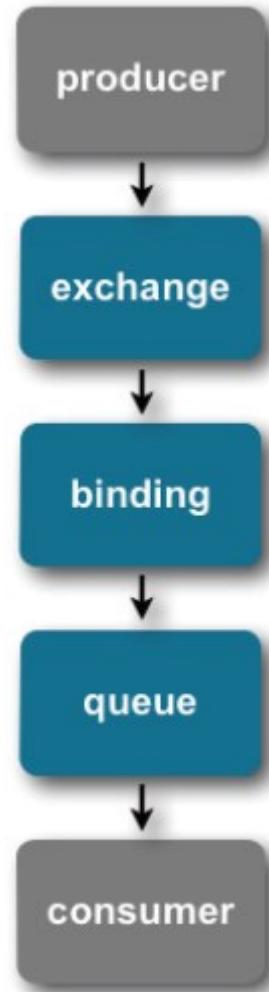
# AMQP poruke



- Mreža ne smije mijenjati poruku
- Header i footer se mogu mijenjati u mreži
- Svaka poruka dobiva jedinstveni ID
- Na čvoru smije postojati samo jedna kopija poruke
- Body type: byte message

# Kako se dostavljaju poruke?

- *Producer*: šalje poruke u *exchange* i dodaje *routing key* uz poruku
- Exchange je povezan s repom putem poveznice (*binding*)
- Binding definira *consumer* (*consumer-driven messaging*), a specificira kakve poruke trebaju biti usmjerene iz *exchangea* do repa
- Consumer je vezan uz rep i prima poruke iz repa
- Uspoređuje se *routing key* i *binding*, ako je uvjet zadovoljen, poruka se isporučuje repu s definiranim *bindingom*



# RabbitMQ

- *open source message queuing software*
- Pisan u programskom jeziku Erlang
- implementira AMQP v0.91, postoji [plugin](#) za AMQP v1.0

STANDALONE

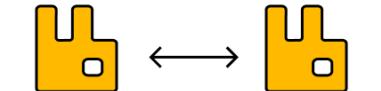


RabbitMQ  
Broker

CLUSTER



RabbitMQ  
Broker



FEDERATION



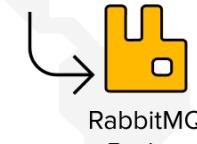
RabbitMQ  
Broker



RabbitMQ  
Broker



RabbitMQ  
Broker

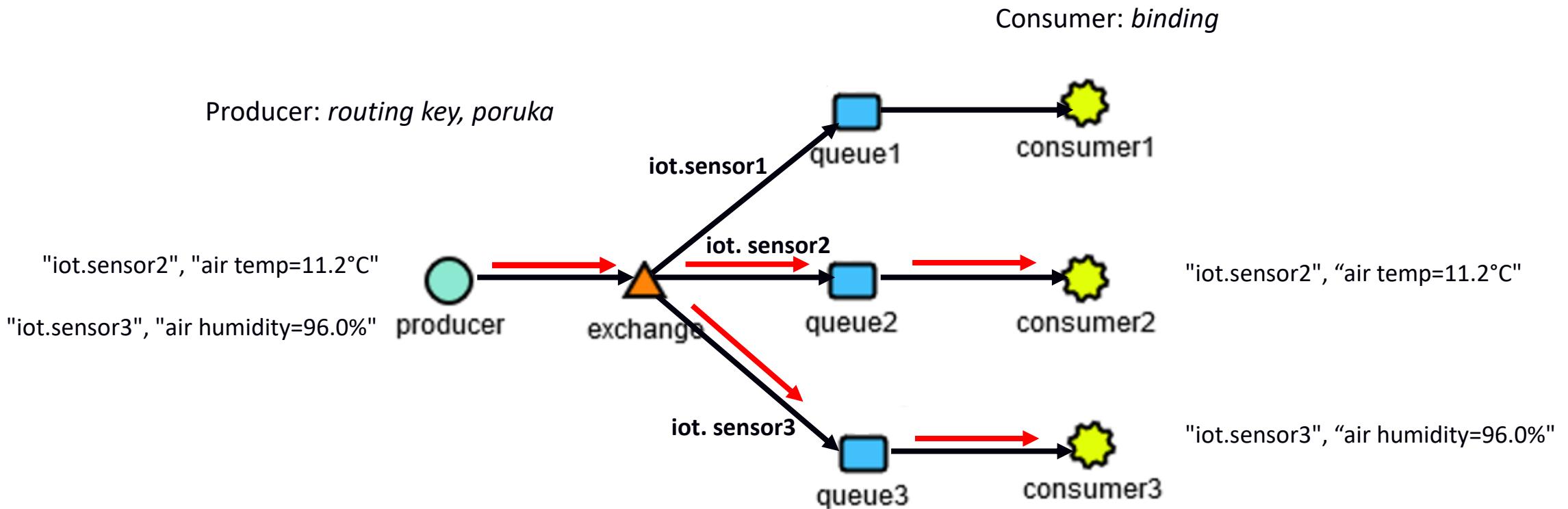


Različite topologije i organizacije posrednika

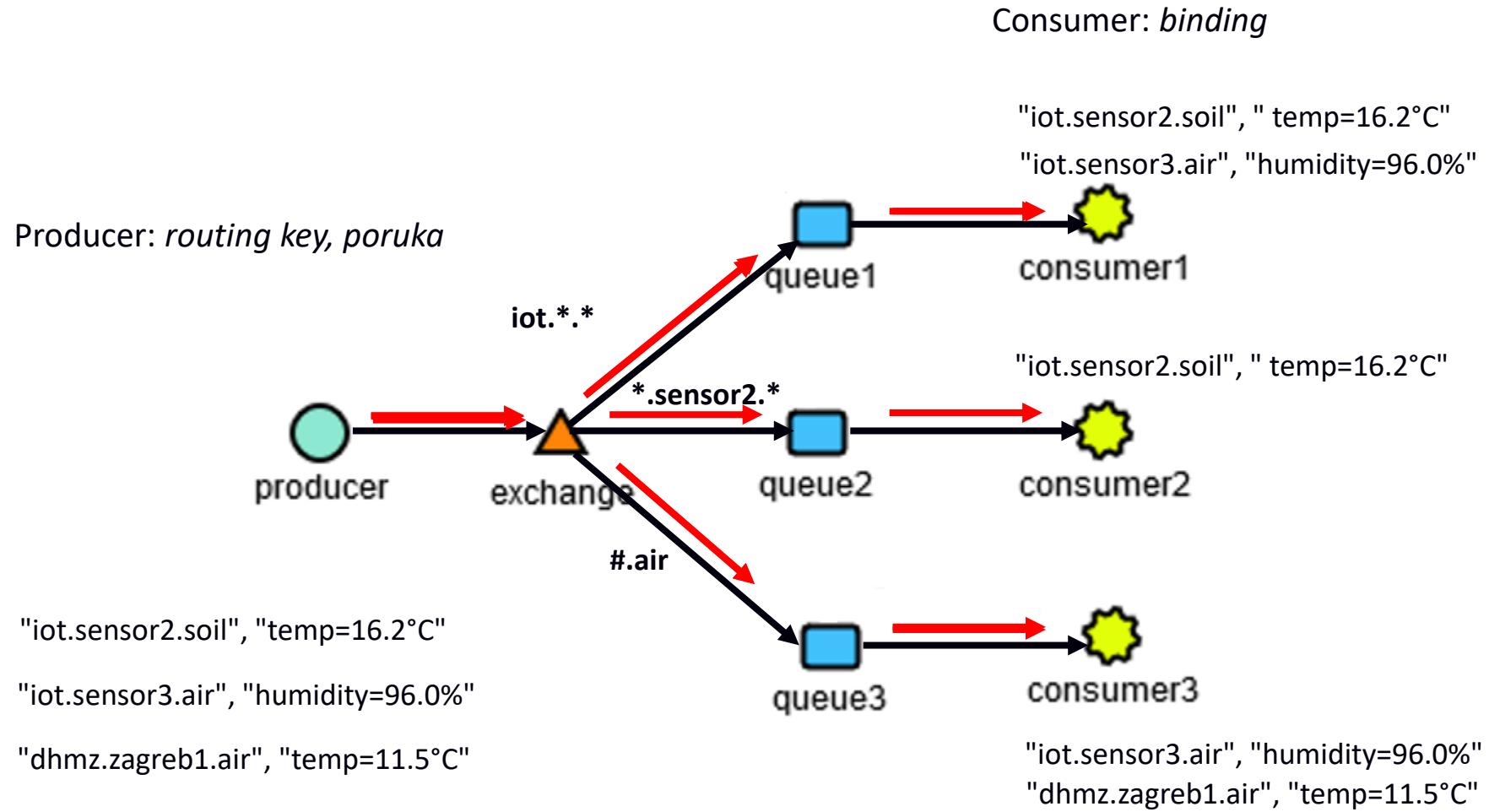
# RabbitMQ: različiti komunikacijski modeli

- *Direct Exchange*: odgovara *point-to-point* modelu JMS-a
  - isporučuje poruku u rep samo ako *routing key*-a poruke zadovoljava uvjet *binding*-a vezanog uz rep
  - poruka može biti isporučena u više repova ako više *binding*-a odgovara *routing key*-u
- *Fanout i Topic Exchange*: odgovara *publish/subscribe* modelu JMS-a
  - *Fanout Exchange*: šalje sve poruke na sve repove spojene na *exchange*
  - *Topic Exchange*: omogućuje filtriranje poruka i definiranje *binding*-a uporabom specijalnih znakova # i \*

# RabbitMQ: Direct Exchange



# RabbitMQ: Topic Exchange



# Primjer 2: AMQP - Producer

```
private final static String EXCHANGE_NAME = "MyExchange";

public static void main(String[] args) throws Exception {
 ConnectionFactory factory = new ConnectionFactory();
 factory.setHost("localhost");
 Connection connection = factory.newConnection();
 Channel channel = connection.createChannel();

 channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
 String routingKey = "iot.sensor2.soil";
 String message = "temp 16.2 C";
 channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());

 channel.close();
 connection.close();
}
```

RabbitMQ: producer šalje poruku na exchange pod nazivom "MyExchange", veže *routing key* uz poruku

# AMQP - Consumer

Consumer definira rep i povezuje ga s *exchangeom*, kada binding odgovara *routing key*-u, poruka se isporučuje

```
...
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);
String queueName = channel.queueDeclare().getQueue();

String binding = "*.sensor2.*";
channel.queueBind(queueName, EXCHANGE_NAME, binding);

Consumer consumer = new DefaultConsumer(channel) {
 @Override
 public void handleDelivery(String consumerTag, Envelope envelope,
 AMQP.BasicProperties properties, byte[] body) throws IOException{
 String message = new String(body, "UTF-8");
 System.out.println("Received: " + message);
 }
};
channel.basicConsume(queueName, true, consumer);
```

# Literatura: AMQP

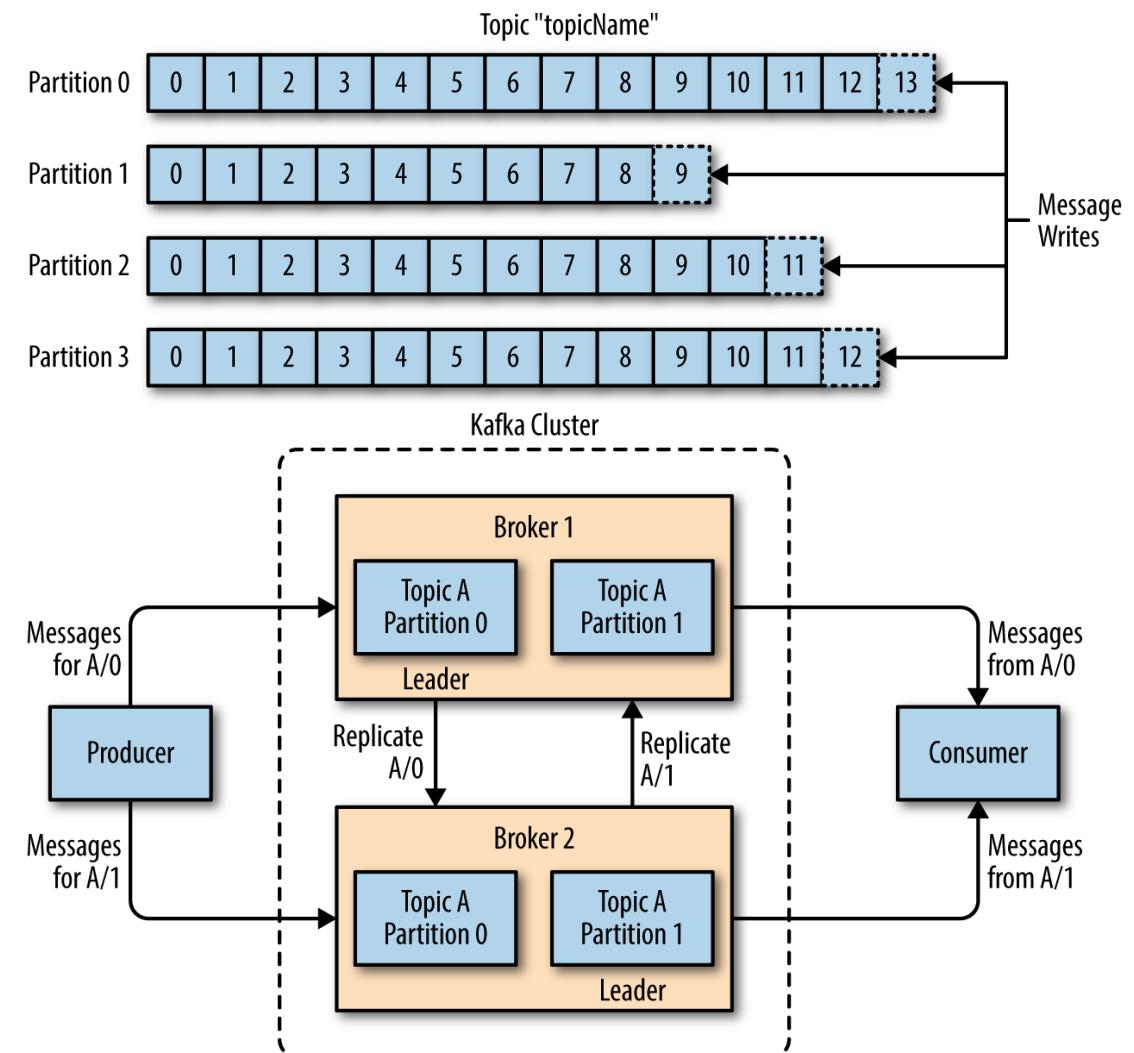
- RabbitMQ Tutorials, <https://www.rabbitmq.com/getstarted.html>
- RabbitMQ Simulator, <http://tryrabbitmq.com/>
- Mark Richards: Understanding the Differences between AMQP & JMS, 2011  
<http://www.wmrichards.com/amqp.pdf>
- Spring messaging with RabbitMQ  
<https://spring.io/guides/gs/messaging-rabbitmq/>

# Apache Kafka

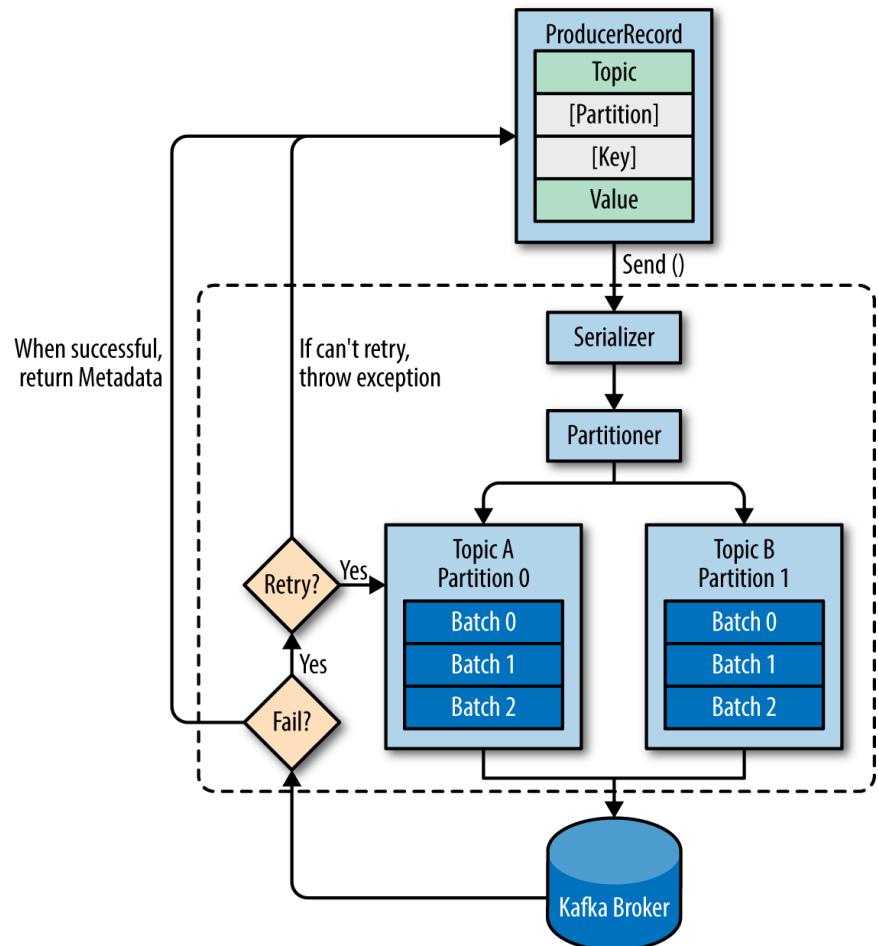
- *streaming platform*: sustav koji omogućuje objavljivanje i pretplatu na tokove podataka, njihovo pohranjivanje i obradu
- Platforma inicijalno razvijena u LinkedIn-u za “*user activity tracking*”
- Obrada „velikih“ tokova podataka, veliki broj objavljivača i pretplatnika, skalabilnost je na prvom mjestu dizajna sustava
- *Data retention*: omogućuje pohranu i čuvanje podataka iz toka, podaci se pohranjuju na disk
- Može se konfigurirati Kafka klaster koji koristi više brokera, a i više klastera koji pokrivaju veći broj podatkovnih centara

# Terminologija

- *Message*: (key, value), key pridjeljuje poruku particiji, a ne mora biti definiran
- *Batch*: niz poruka koje se objavljuju na isti *topic* i particiju, može se koristiti kompresija
- *Topic*: sadrži više particija
- *Partition*: uređeni niz poruka
- *Producer*: objavljuje poruke na particiju
- *Consumer*: čita poruke iz particije
- Broker
- Cluster



# Kafka Producer



```
Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers",
"broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
producer = new KafkaProducer<String,
String>(kafkaProps);
```

Novi producer s nužnim parametrima

# Slanje poruke

1. *Fire-and-forget*: poruka se šalje bez potvrde primitka
2. *Synchronous send*: producer je uвijek asinkron (шалје се порука, а метода `send()` враћа Future object). У овом случају се користи метода `get()` која чека информацију да је `send()` био успјешан
3. *Asynchronous send*: pozива методу `send()` с посебном callback функцијом која prima одговор од брокера у будућности о томе је ли метода `send()` успјешно извршена или не

```
ProducerRecord<String, String>
record =
 new ProducerRecord<>(Topic,
key, value);
try {
 producer.send(record);
} catch (Exception e) {
 e.printStackTrace();
}
```

# Consumer

```
Properties props = new Properties();

props.put("bootstrap.servers",
"broker1:9092,broker2:9092");

props.put("group.id", "CountryCounter");

props.put("key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer =
new KafkaConsumer<String, String>(props);
```

- Jednostavna pretplata na topic

```
consumer.subscribe(Collections.singleton (Topic))
```

- Asynchronous Commit

```
Duration timeout =
Duration.ofMillis(100);

while (true) {

 ConsumerRecords<String, String>
records = consumer.poll(timeout);

 for (ConsumerRecord<String, String>
record : records) {

 System.out.printf("topic = %s,
partition = %s, offset = %d, customer =
%s, country = %s\n", record.topic(),
record.partition(), record.offset(),
record.key(), record.value()));

 }

 consumer.commitAsync(); 1
}
```

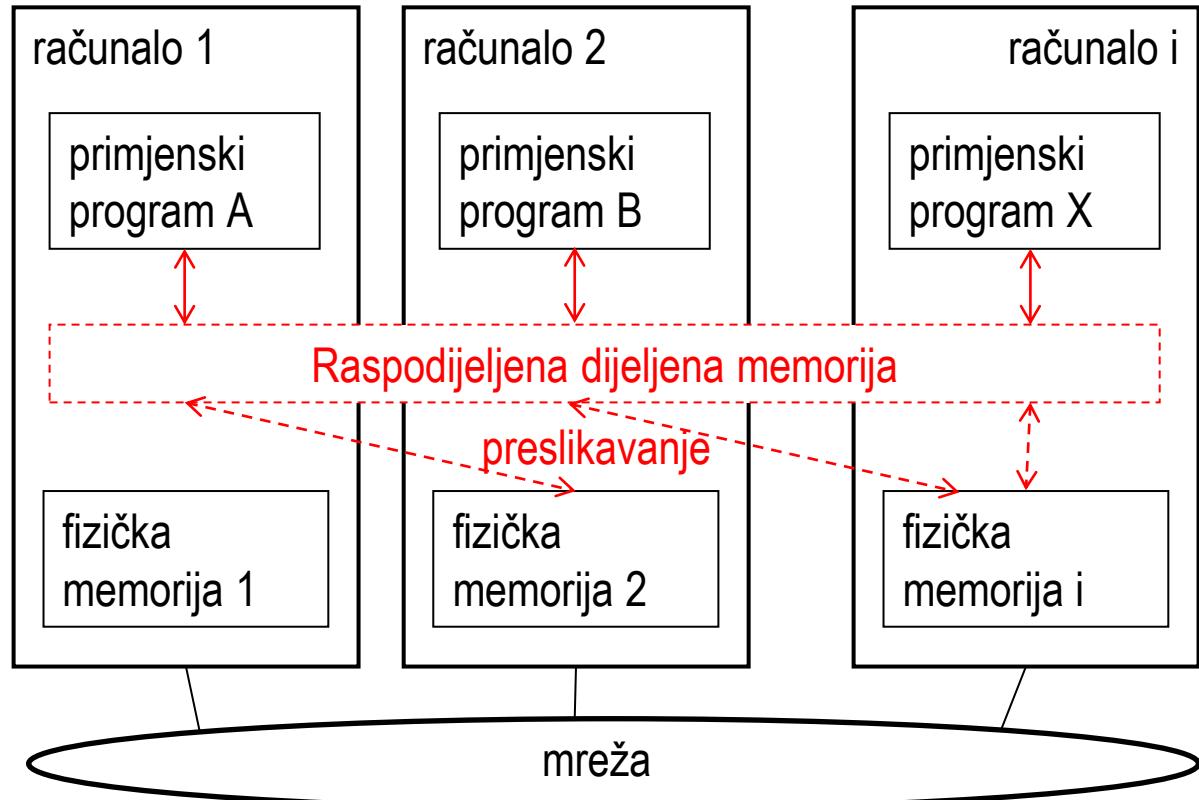
# Literatura

- **Kafka: The Definitive Guide, 2nd Edition, By Gwen Shapira, Todd Palino, Rajini Sivaram, Krit Petty, PUBLISHED BY:O'Reilly Media, Inc. PUBLICATION DATE:November 2021, PRINT LENGTH:455 pages**
- <http://kafka.apache.org/downloads> (koristimo verziju 2.8.1)

# Sadržaj predavanja

- Komunikacija porukama
- Model objavi-preplati
- Primjeri protokola za komunikaciju porukama: JMS i AMQP
- **Dijeljeni podatkovni prostor**

# Raspodijeljena dijeljena memorija



- Posrednički sloj koji nudi transparentan pristup dijeljenoj memoriji računala bez zajedničke fizičke memorije

*Distributed Shared Memory (DSM)*

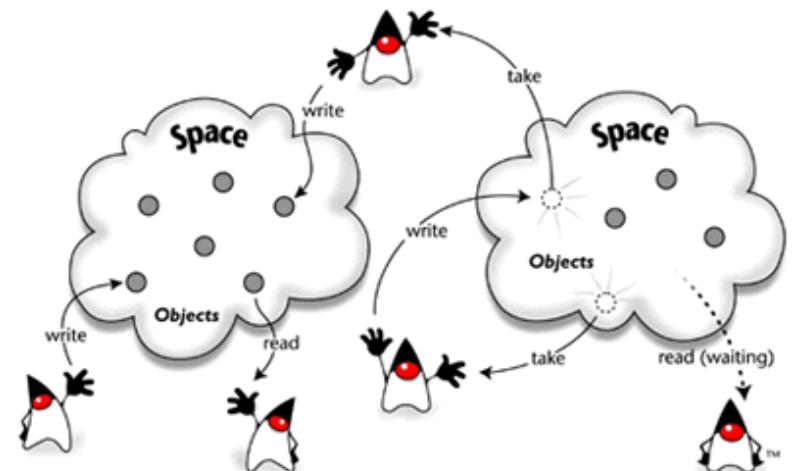
- omogućuje transparentan pristup fizičkoj memoriji na drugim računalima (primjenski program ima dojam da pristupa vlastitoj fizičkoj memoriji)
- upravlja replikama podataka, na računalu se čuvaju lokalne kopije podataka kojima je nedavno pristupao primjenski program X

# Dijeljeni podatkovni prostor

engl. *shared data/tuple spaces*

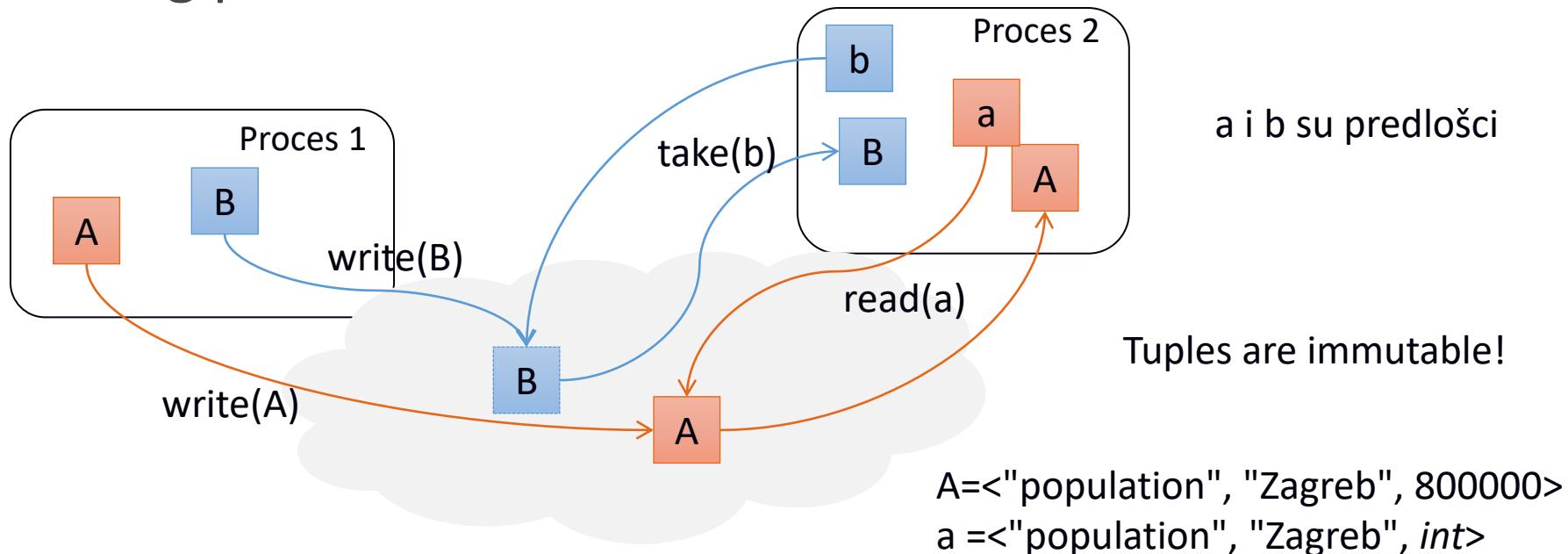
- arhitektura temeljna na podacima (*content-addressable memory*)
- procesi mogu dodati, čitati i “izvaditi” *tuple* iz zajedničkog dijeljenog podatkovnog prostora (*tuple space*)
- *tuple*: slijed podataka, za svaki je definiran tip
- primjeri: Linda, JavaSpaces,  
TSpaces

Izvor: "JavaSpaces Principles, Patterns, and Practice"  
<http://java.sun.com/developer/Books/JavaSpaces/introduction.html>



# Operacije

- **write (A)** – dodaj tuple A u raspodijeljeni podatkovni prostor
- **read (a) → A** – vraća tuple A koji odgovara predlošku a
- **take (b) → B** – vraća tuple B koji odgovara predlošku b i biće ga iz podatkovnog prostora



# Primjer aplikacije



# Obilježja dijeljenog podatkovnog prostora

- **vremenska neovisnost**
  - procesi ne moraju istovremeno biti aktivni radi komunikacije, dijeljeni podatkovni prostor pohranjuje poruku
- **anonimna komunikacija** (temelji se na sadržaju podataka)
- komunikacija je **perzistentna**
- **asinkrona komunikacija**
  - proces dodaje podatak u podatkovni prostor i nastavlja obradu
- pokretanje komunikacije na načelu ***pull***
  - proces eksplicitno šalje zahtjev za čitanje podatka iz dijeljenog podatkovnog prostora

# Pitanja za učenje i ponavljanje

- Objasnite značenje vremenske i prostorne neovisnosti za komunikaciju procesa. Navedite jesu li komunikacija porukama i komunikacija na načelu objavi-preplati vremenski i prostorno ovisne ili neovisne.
- Navedite sličnosti i razlike komunikacije na načelu objavi-preplati i dijeljenog podatkovnog prostora.
- Usporedite preplatu u sustavima objavi-preplati i predložak u sustavima s dijeljenim podatkovnim prostorom. Zašto je moguće realizirati tzv. vremenski i prostorno neovisnu komunikaciju?
- Gdje se filtriraju obavijesti u raspodijeljenom sustavu objavi-preplati koji koristi preplavljivanje obavijestima?
- Zašto za raspodijeljeni sustav objavi-preplati koji koristi preplavljivanje preplatama kažemo da filtrira obavijesti na samom ulazu u mrežu posrednika?
- Skicirajte primjer raspodijeljenog izvođenja sustava objavi-preplati s 3 posrednika gdje je  $P_1$  spojen na  $B_1$ ,  $S_1$  na  $B_2$ , a  $S_3$  na  $B_3$  za slijed događaja sa slajda 22.

# Literatura

1. G. Coulouris, J. Dollimore, T. Kindberg: *Distributed Systems: Concepts and Design*, 5th edition, Addison-Wesley, 2012  
poglavlje 6
2. Maarten van Steen, Andrew S. Tanenbaum (2017.), *Distributed Systems 3<sup>rd</sup> edition*, Createspace Independent Publishing Platform  
poglavlje 4.3 (bez dijela o Socketima)