



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**  
Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

## 1. Uvod u raspodijeljene sustave

Ak. god. 2020./2021.

# 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."

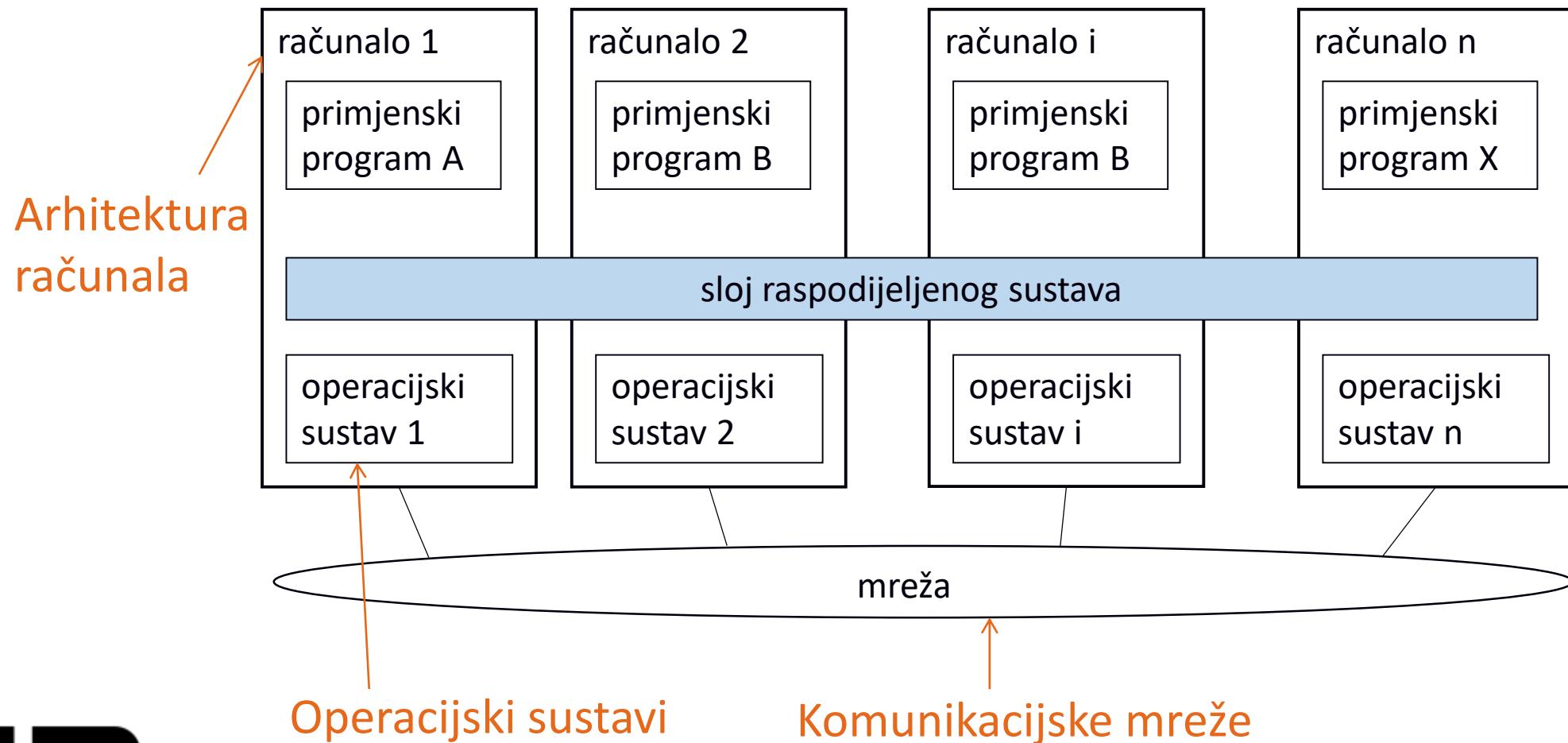
George Coulouris:

- "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)

Raspodijeljeni poslovni i transakcijski sustavi:

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

Sustavi za pružanje informacijskih i komunikacijskih usluga:

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

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

- povezivanje uređaja na Internet – fizičkih i virtualnih objekata
- komunikacija stroja sa strojem (engl. *Machine to Machine*, M2M)

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

*Svi 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 predmetu 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**
  - prikivanje 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 prepostavka 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](http://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 svim procesima (npr. usmjeravanje paketa temeljem poznavanja stanja cijele mreže)

# Skalabilnost (2)

## Tehnike koje omogućuju skalabilnost sustava

- Prikrivanje kašnjenja u komunikaciji
  - “radi nešto korisno dok čekaš odgovor” - asinkrona komunikacija
- Višestrukost
  - više dijelova koji omogućuju funkcionalnost sustava (npr. raspodijeljena baza podataka, sustav imenovanja domena (DNS))
- Replikacija
  - replika = istovjetna kopija dijela sustava ili podatka
  - problem: konzistentnost originala i kopije

# 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

Sustavska arhitektura:

- smještaj programskih komponenata na raspoložive računalne resurse (centralizirana arhitektura ili decentralizirana arhitektura)

# 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

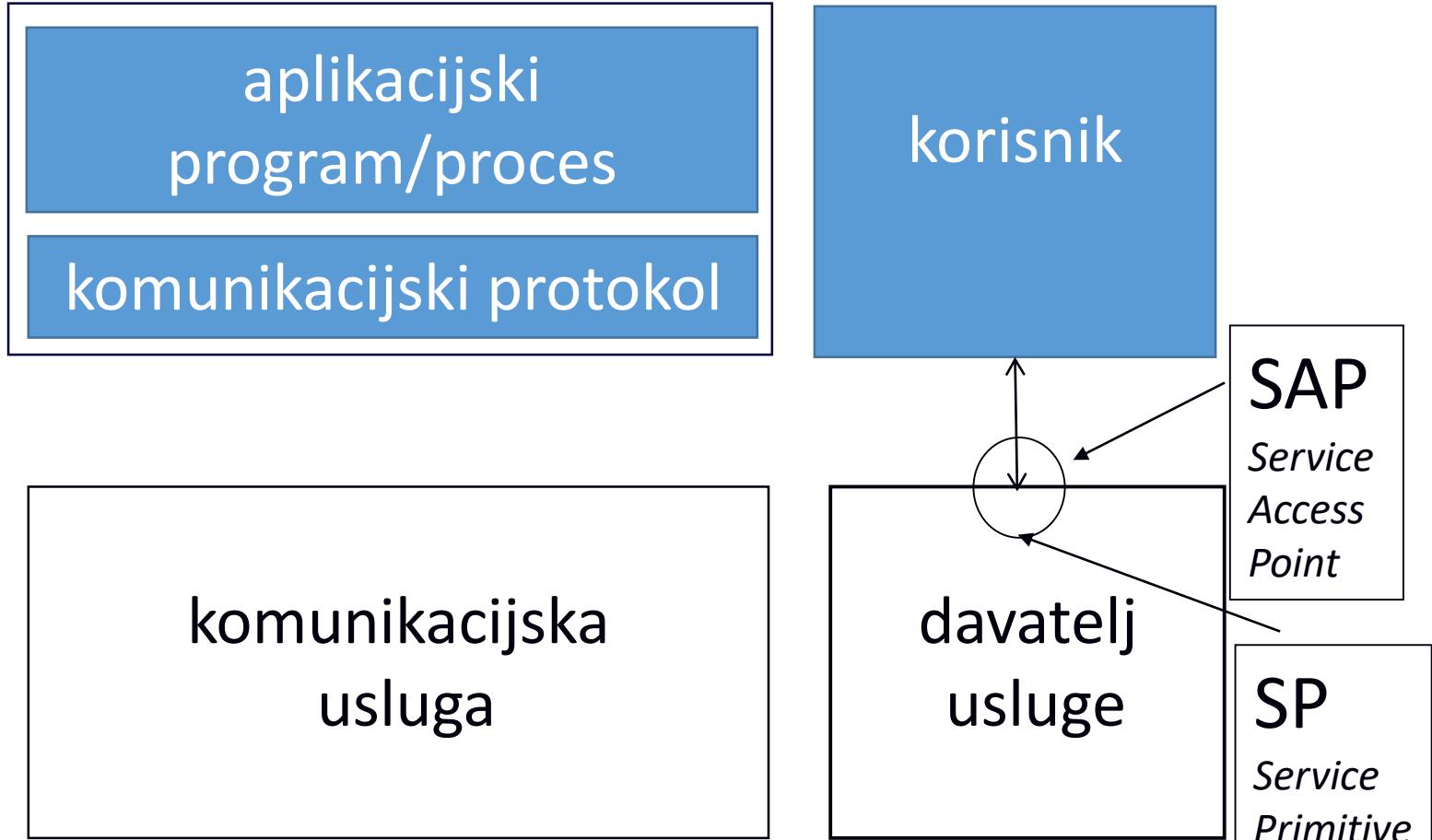
- procesi komuniciraju razmjenom tzv. događaja koji prenose informacije

# Slojevita arhitektura

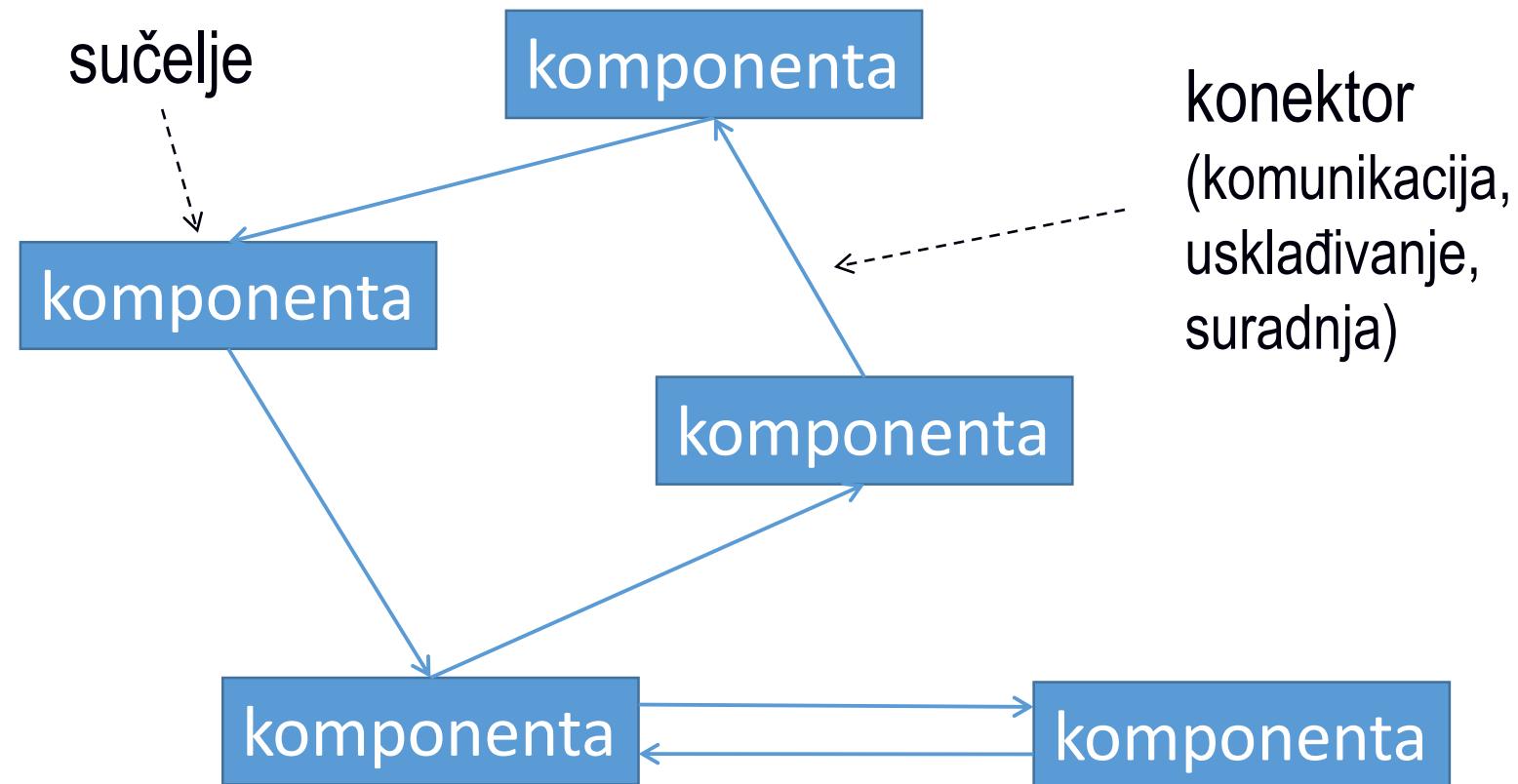
npr.  
www

HTTP

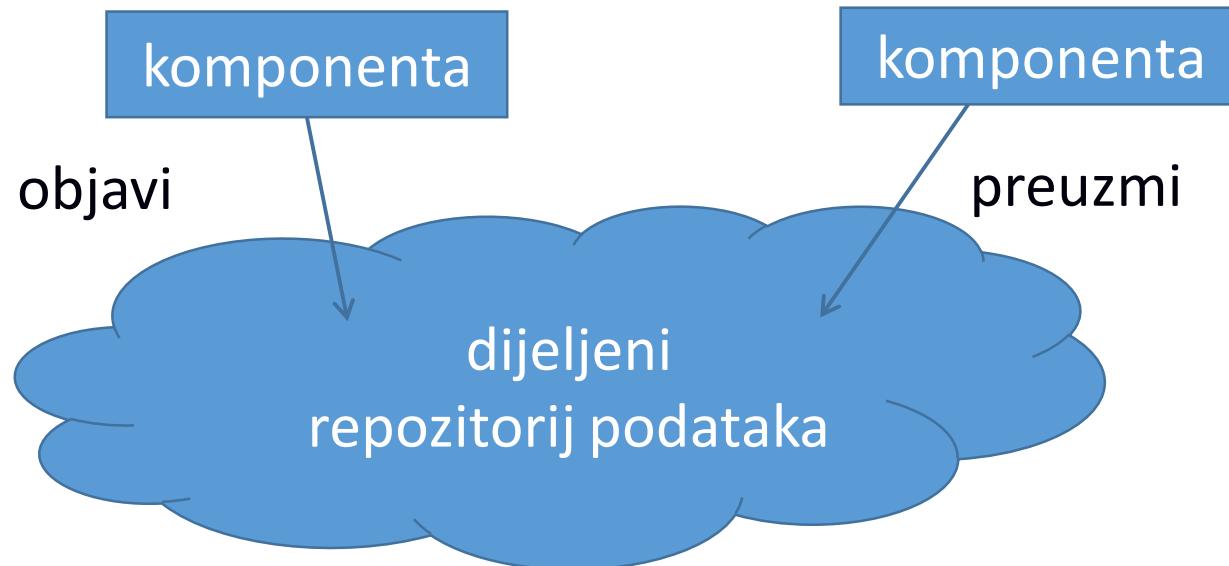
TCP  
IP  
LAN



# Arhitektura temeljena na komponentama

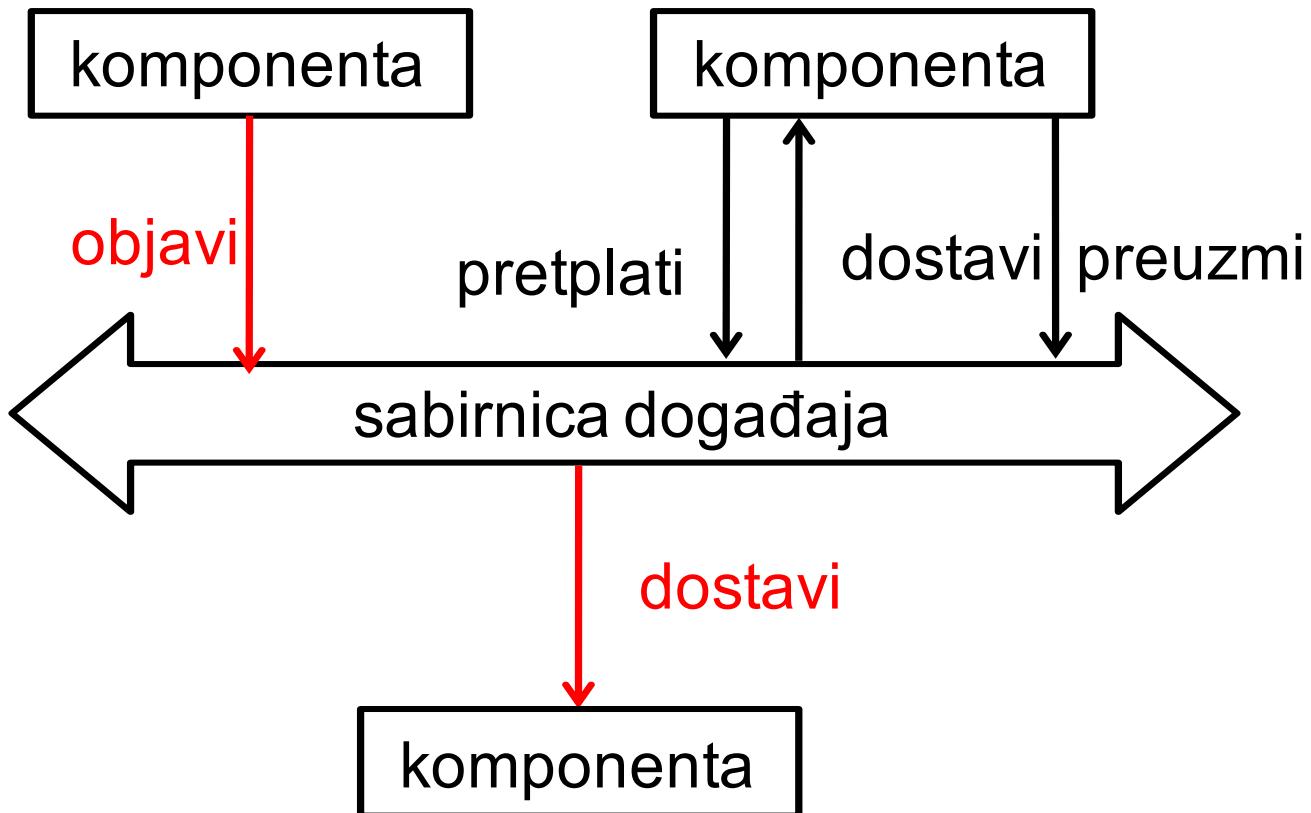


# Arhitektura temeljena na podacima



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

# Arhitektura temeljena na događajima



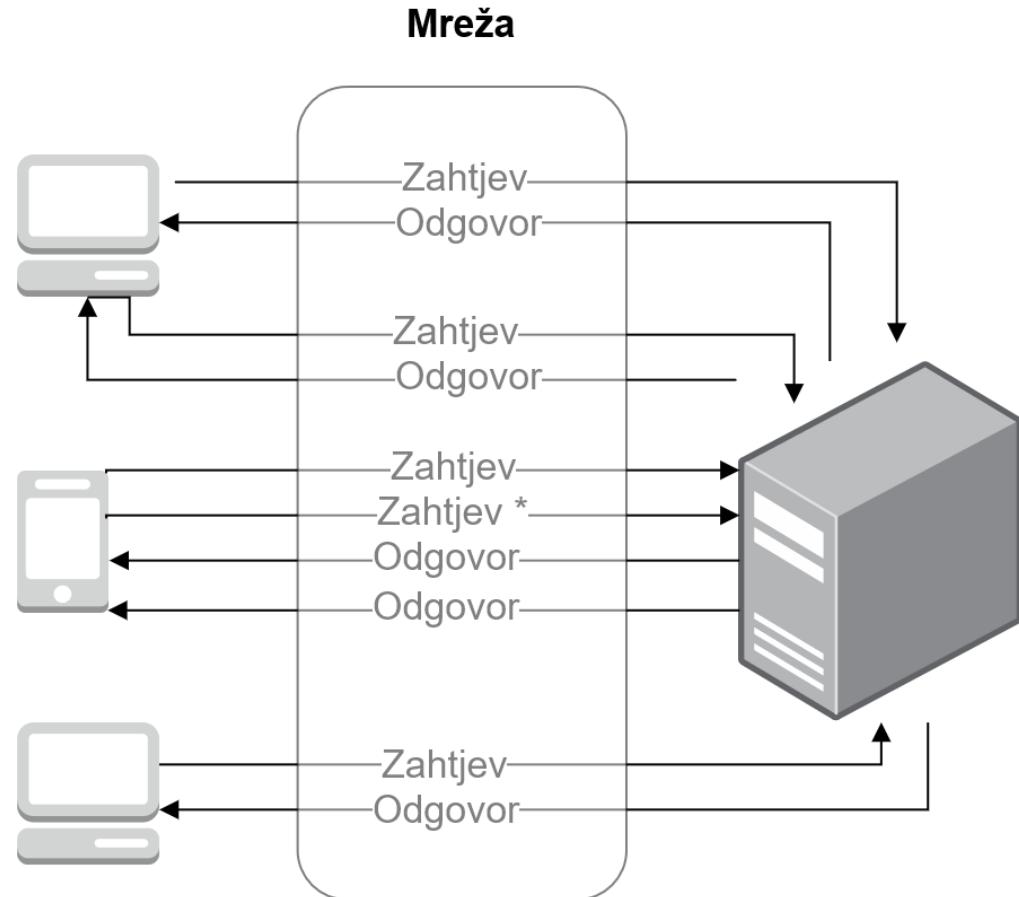
Komponenta se može preplatiti na podatak koji joj je potreban. Kad neka komponenta objavi podatak (događaj!), to će se dostaviti (događaj!) komponenti koja je na njega preplaćena.

# 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

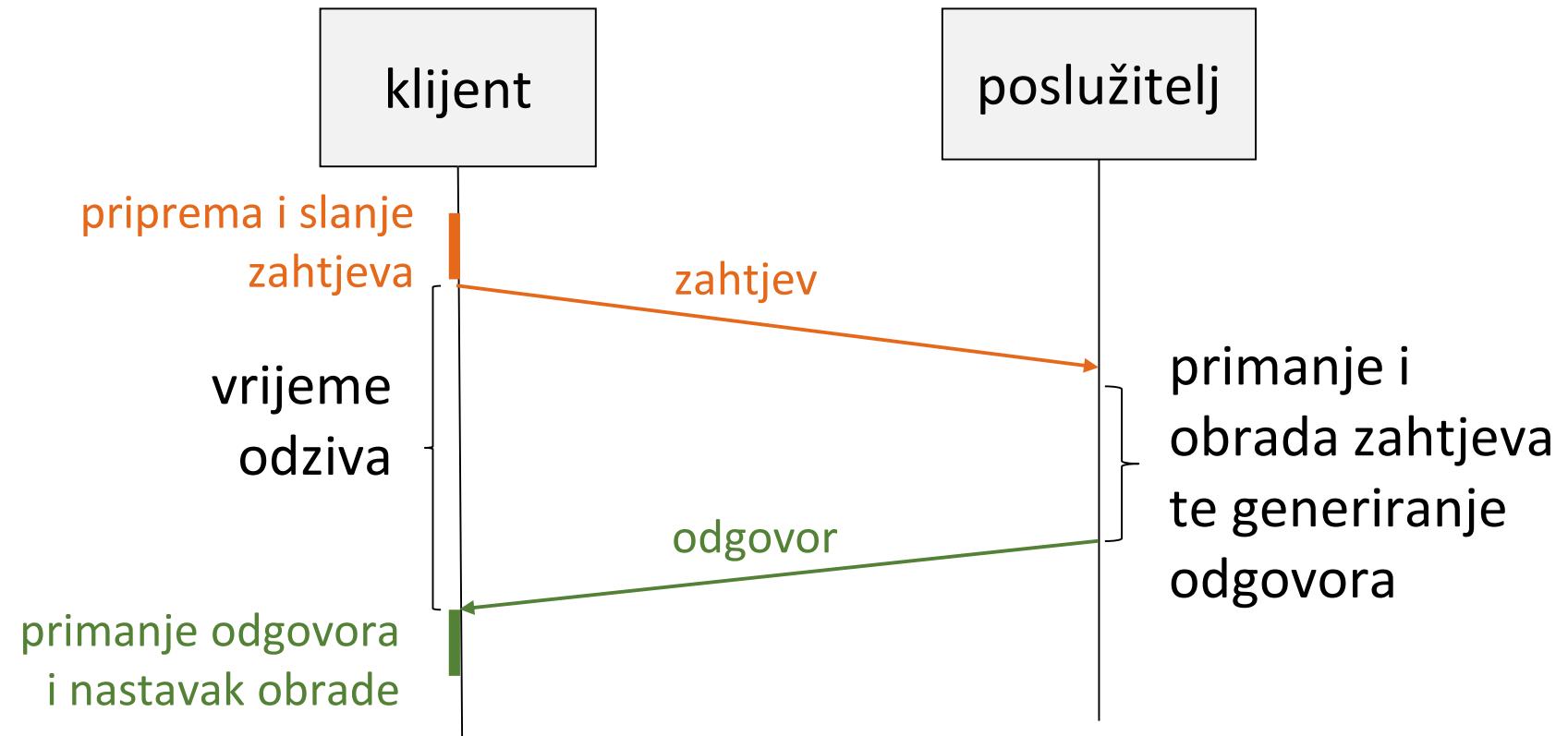
# Model klijent – poslužitelj (1)

- 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

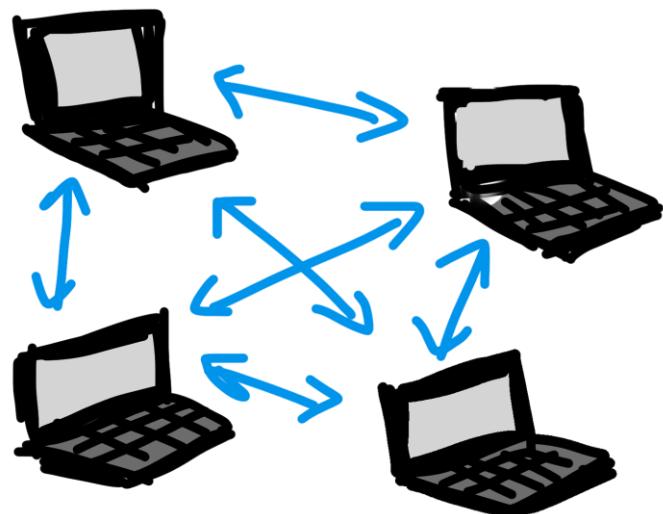


# Model ravnopravnih sudionika (1)

- 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](http://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*)

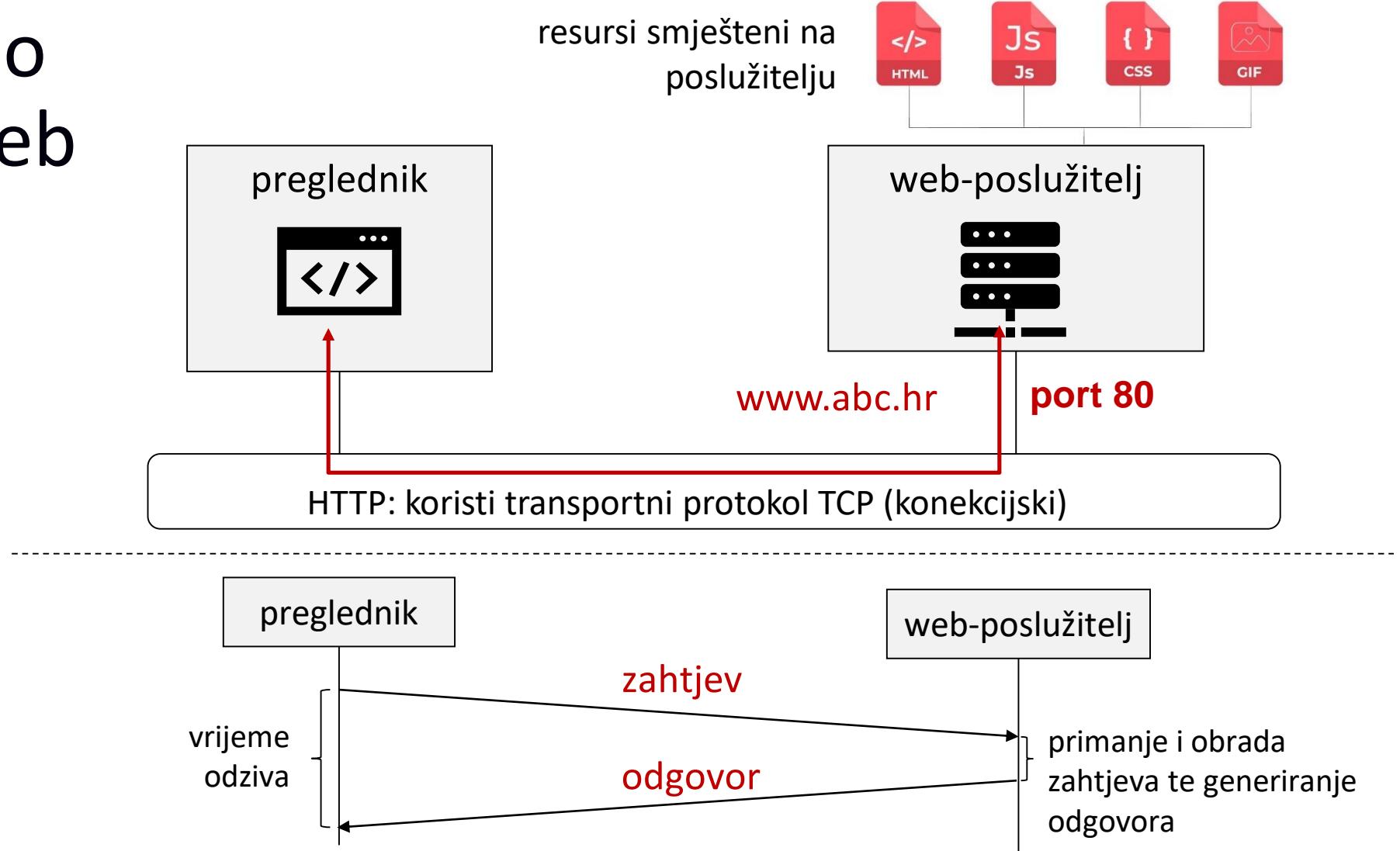


# Hypertext Transfer Protocol (HTTP)

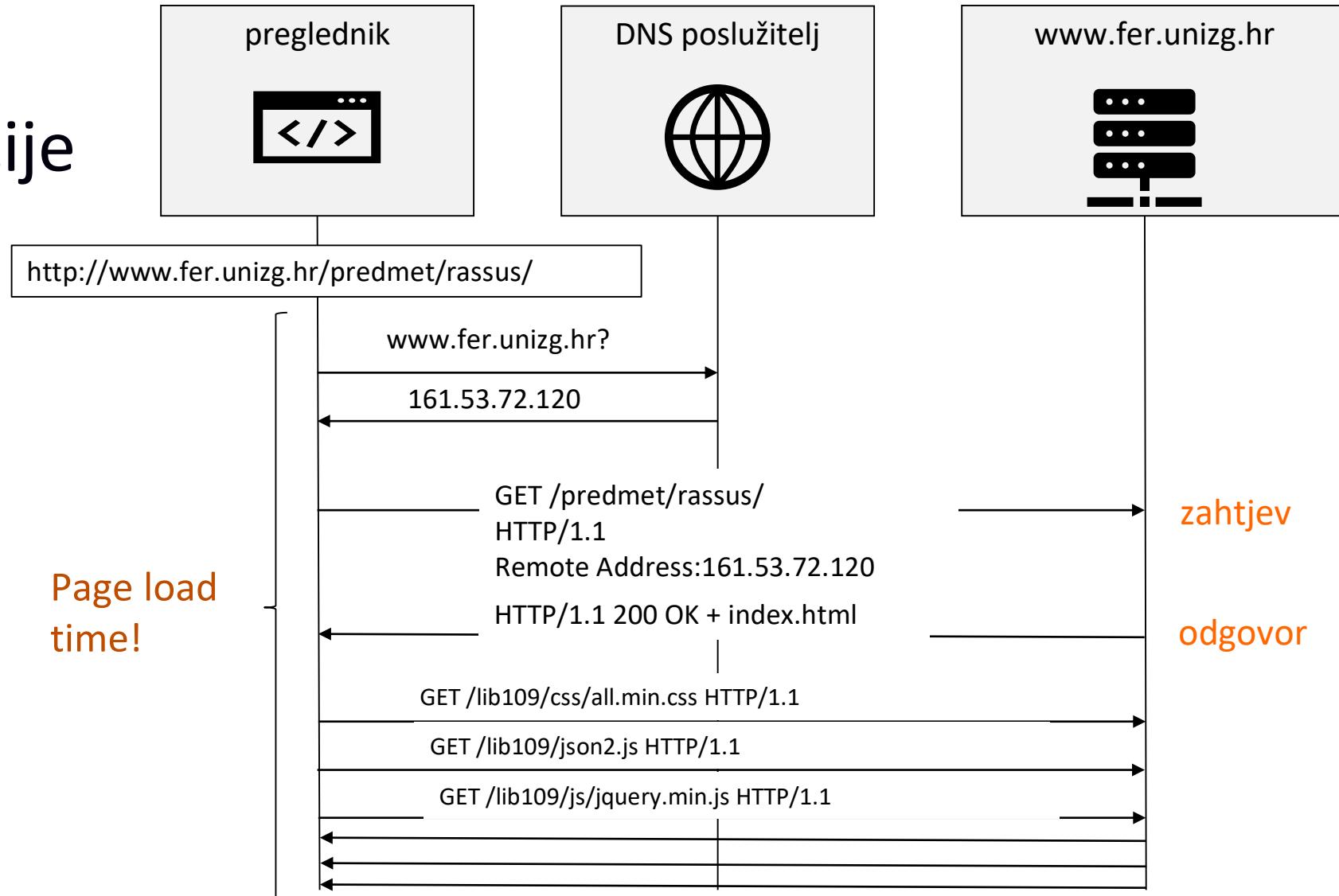
HTTP je **protokol** na aplikacijskom sloju

- 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 ...



# 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

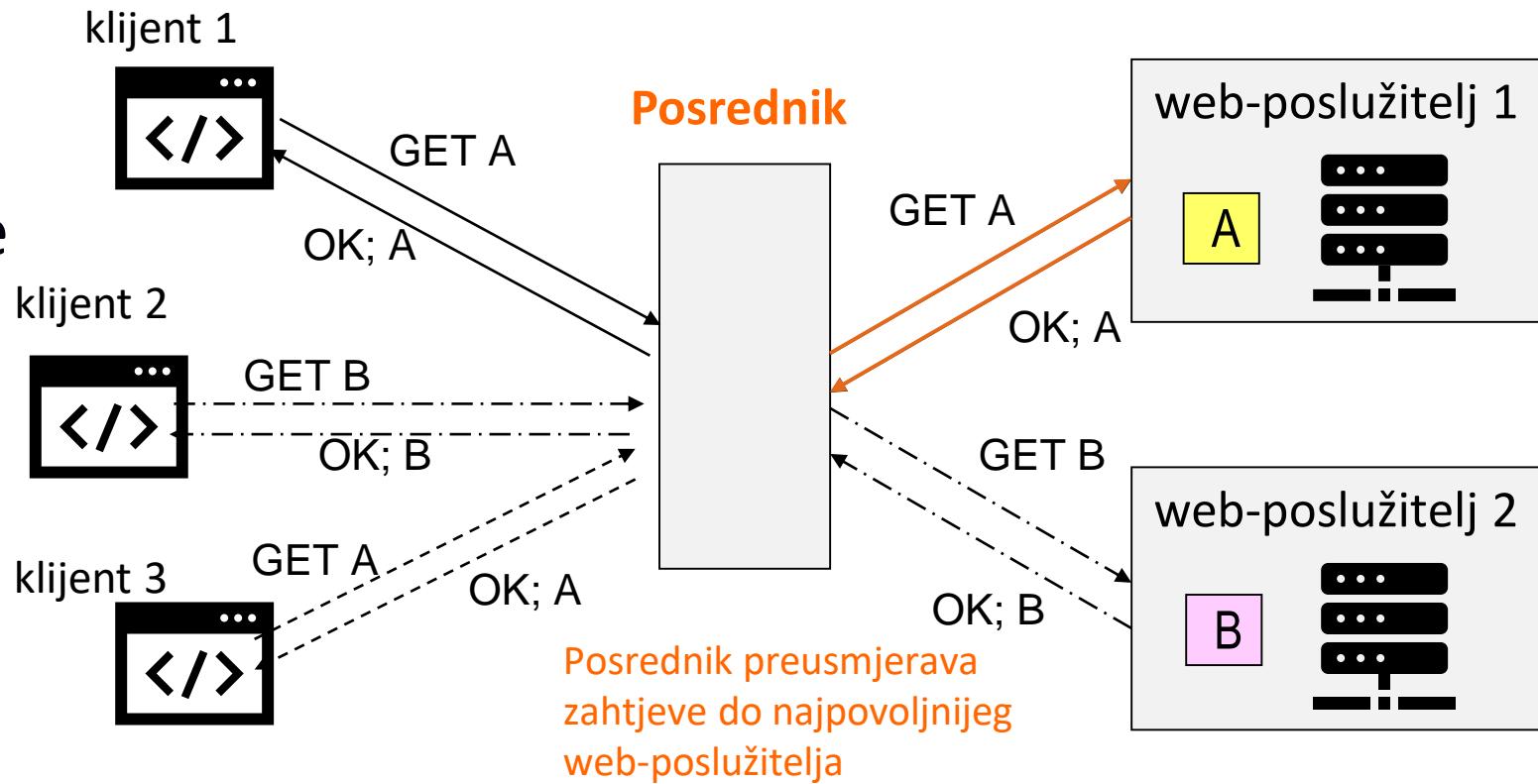
## Relokacijska transparentnost:

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

# Transparentnost web-poslužitelja (2)

Replikacijska transparentnost 1:

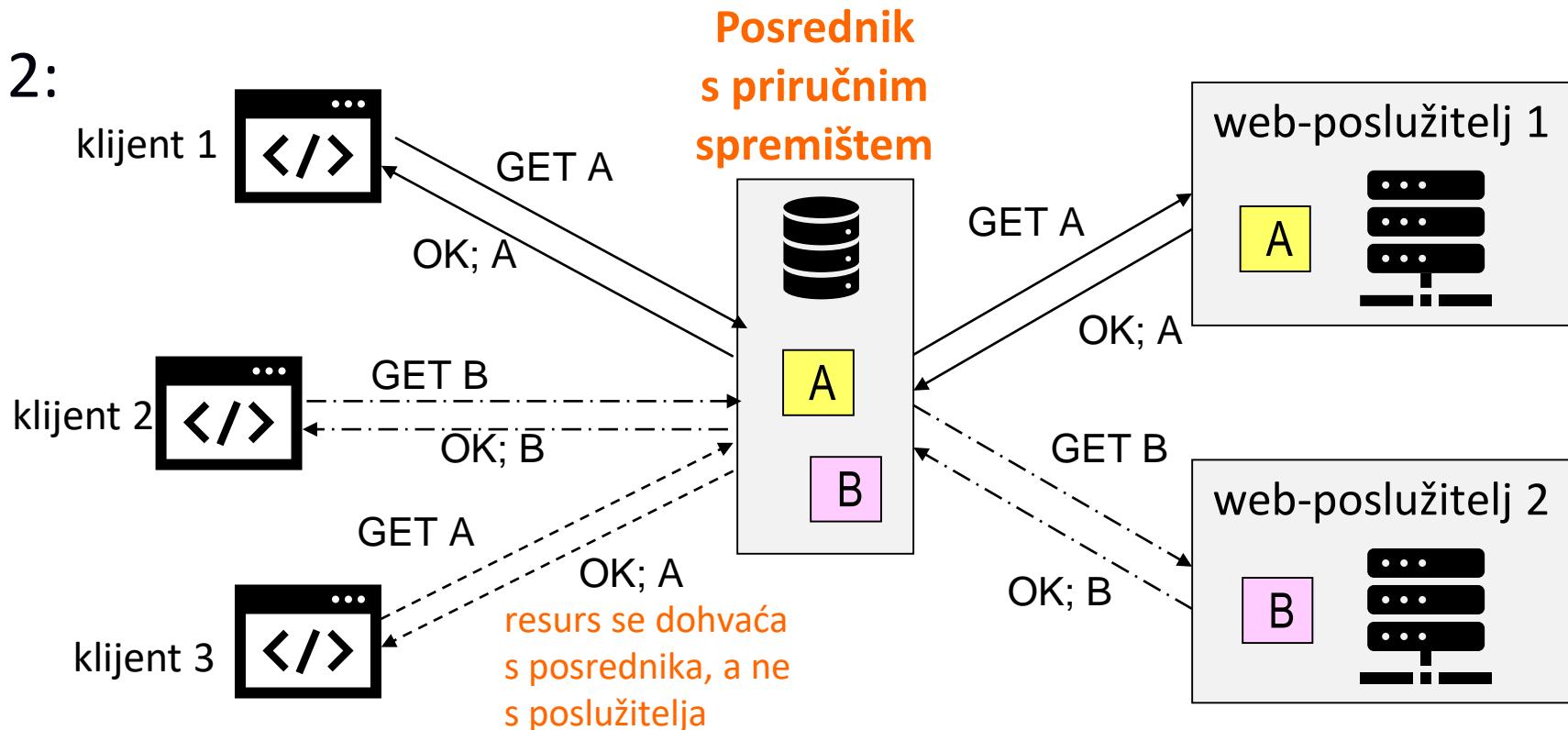
- Poslužiteljima se dostupa posredstvom posrednika (engl. proxy)



# 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**

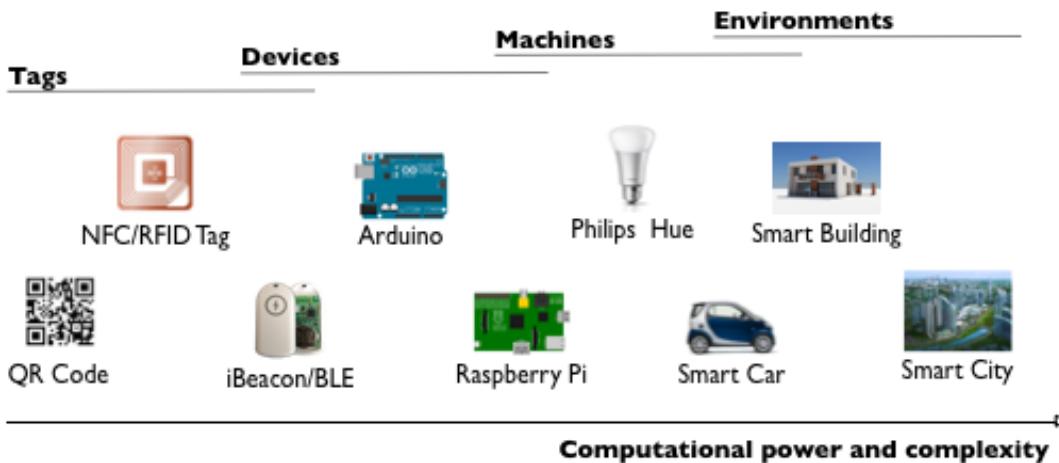
# Uredaji i "stvari" postaju dio Interneta

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



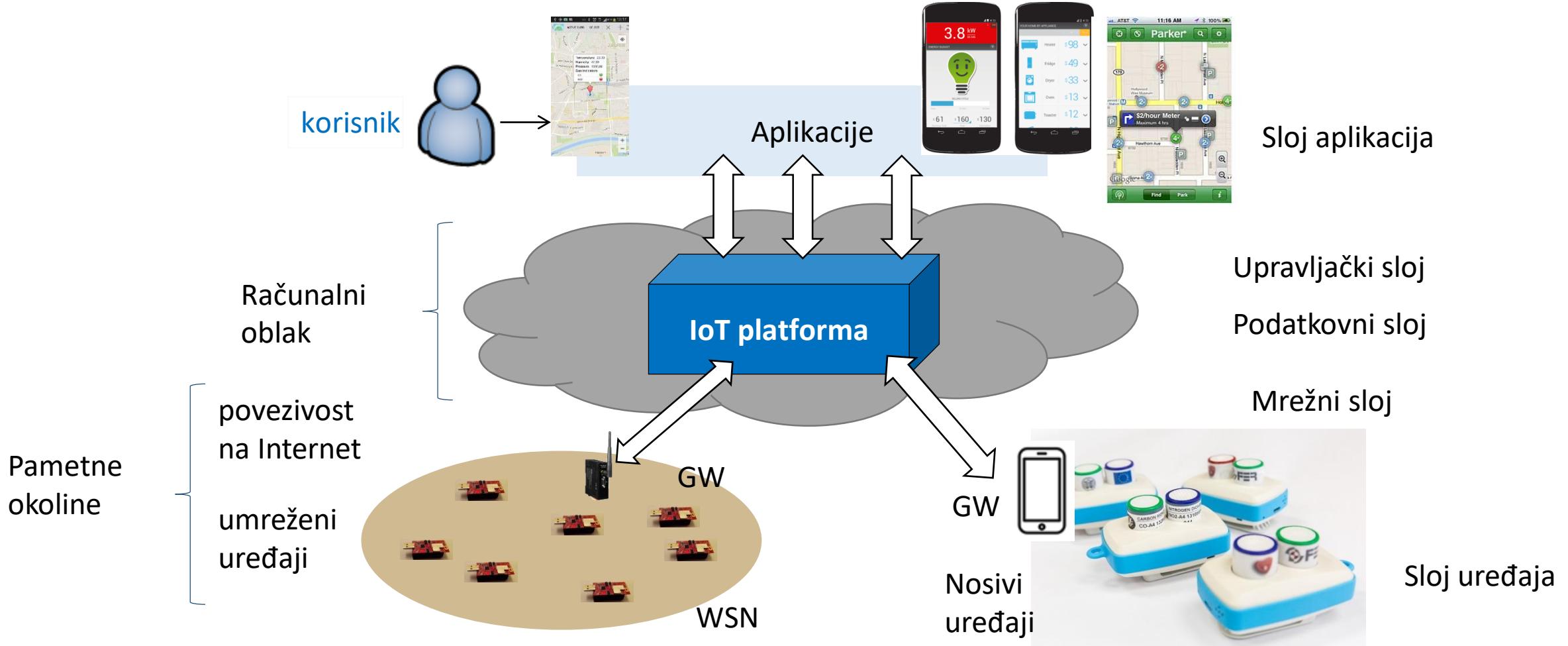
Source: Building the Web of Things: book.webofthings.io  
Creative Commons Attribution 4.0

# 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.*

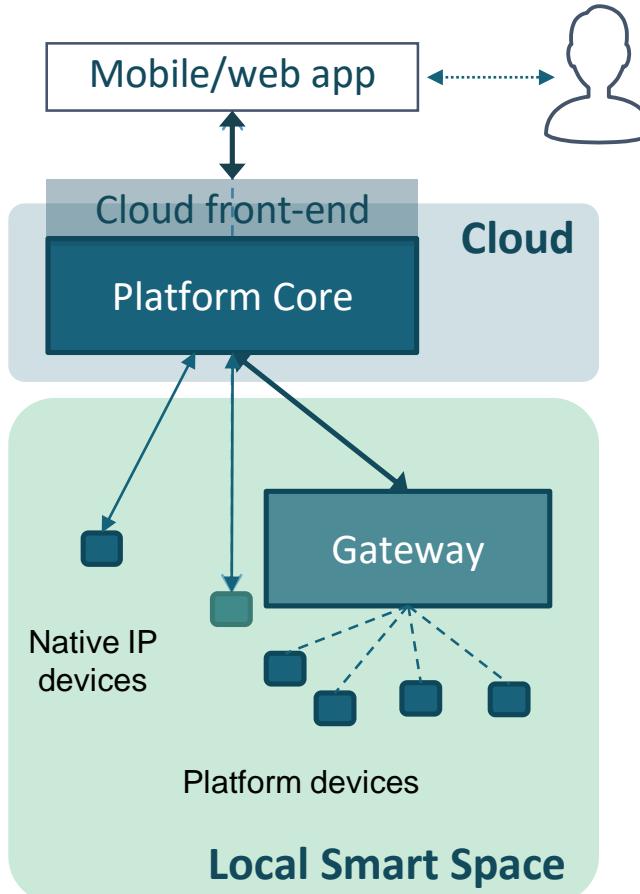
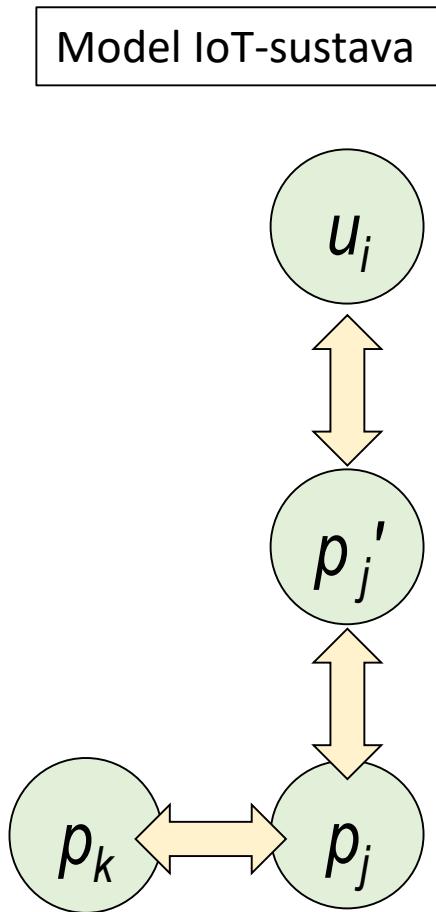
# Pojednostavljena arhitektura Interneta stvari



# 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

# 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. Zašto se koriste perzistentne konekcije u HTTP-u?
7. Kako se definira uvjetni GET i kako funkcionira? Kakve prednosti donosi za performance HTTP-a?
8. Objasnite zašto današnja rješenja za IoT predstavljaju raspodijeljene sustave velikih razmjera?



SVEUČILIŠTE U ZAGREBU



**Diplomski studij**

**Informacijska i  
komunikacijska  
tehnologija:**

Telekomunikacije i  
informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

# Raspodijeljeni sustavi

**2. Procesi i komunikacija: model klijent-poslužitelj**

Ak. god. 2020./2021.

# Creative Commons



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

*U slučaju 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.*

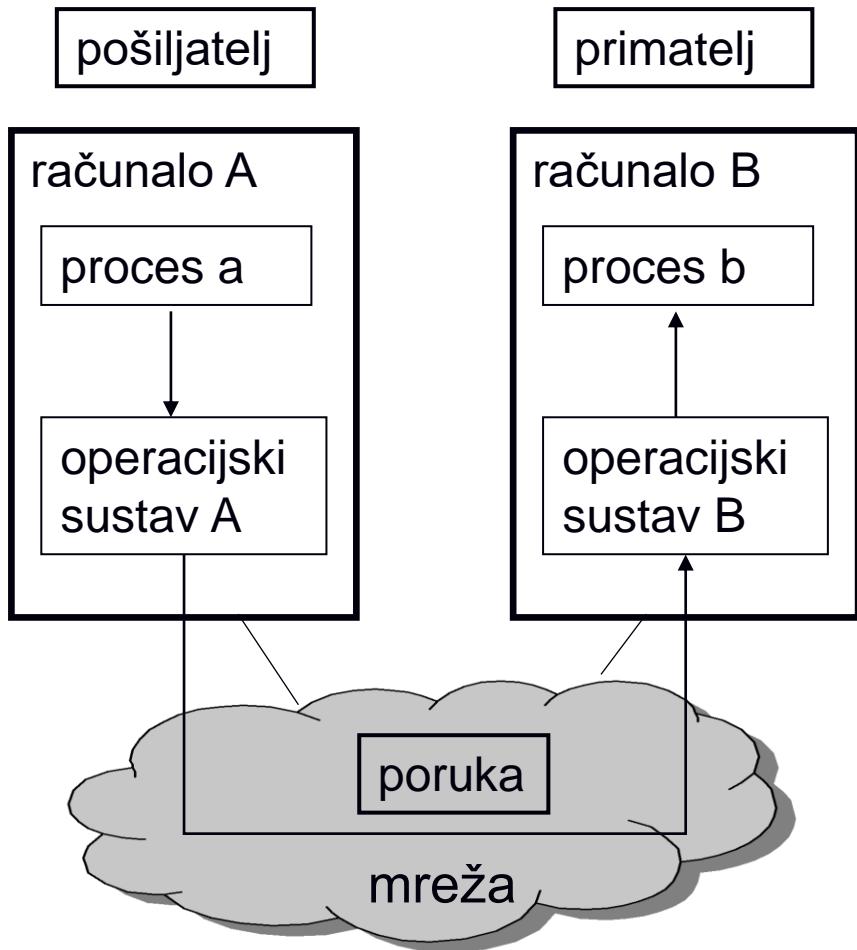
*Tekst licence preuzet je s*

*<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*)

# Osnovni model komunikacije



- **Procesi**

- izvode se na različitim računalima, autonomni su

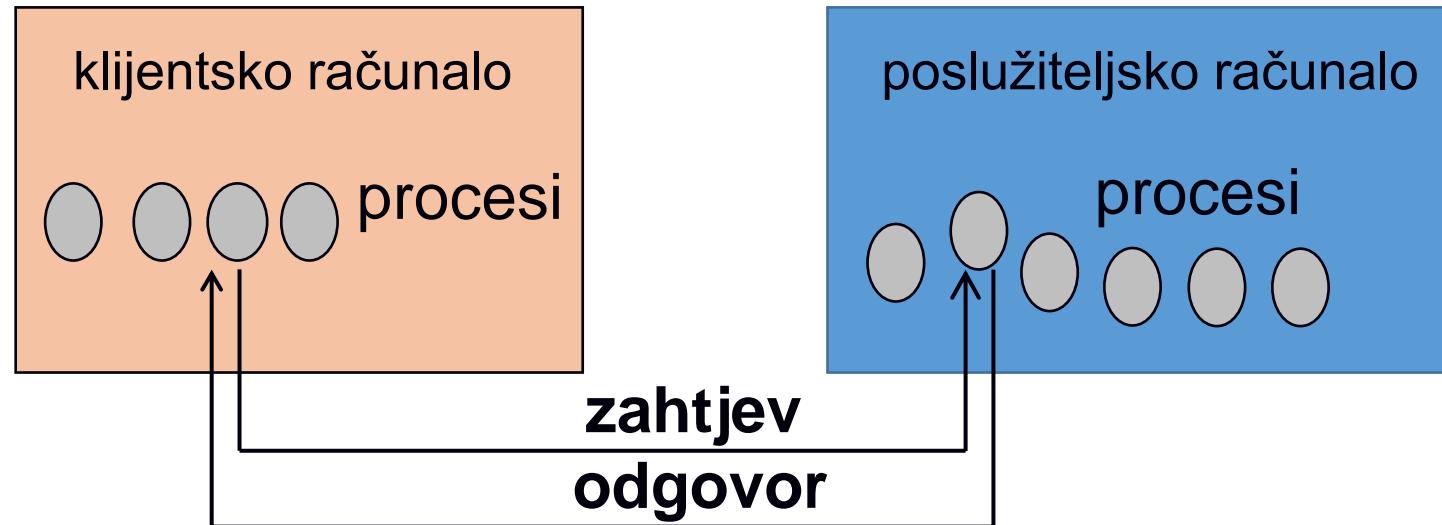
- **Komunikacija**

- 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 eksplicitno 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

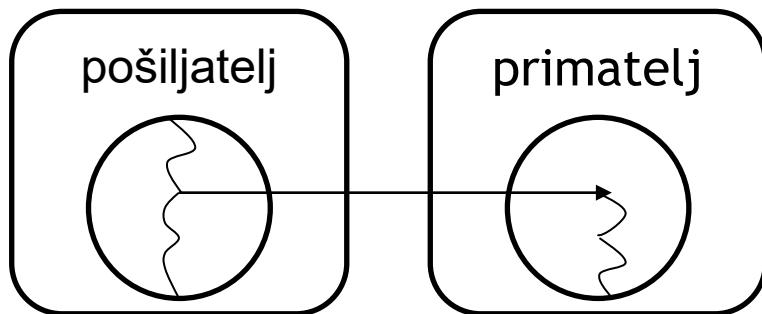
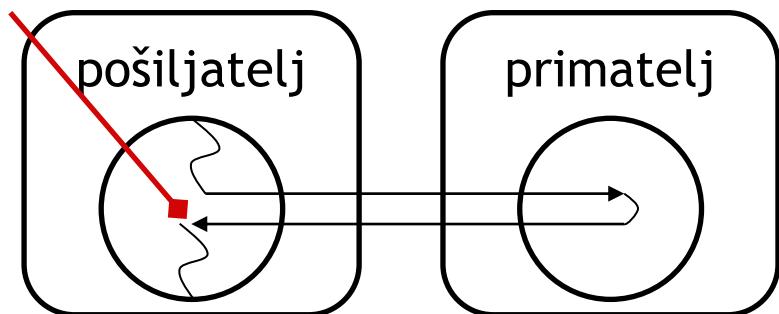
- **sinkrona komunikacija**

- blokira pošiljatelja do primitka potvrde od strane primatelja

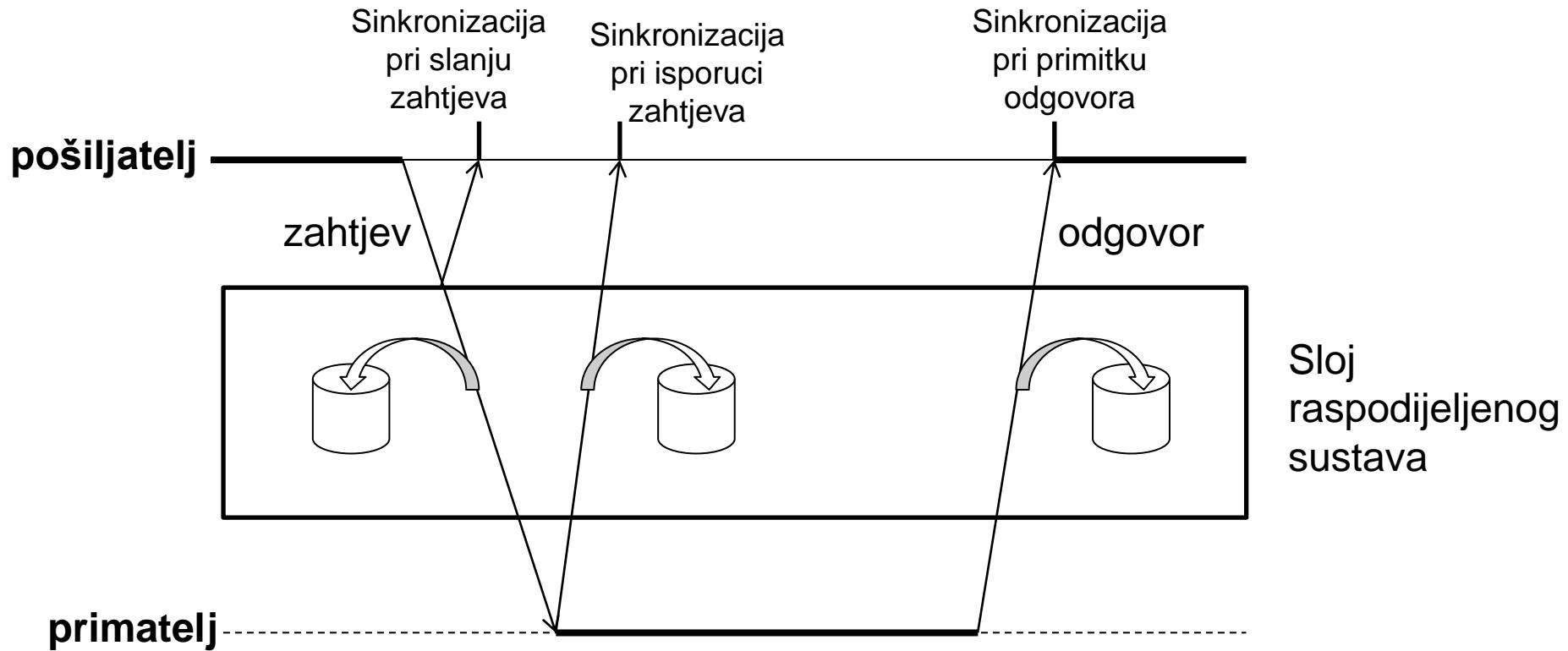
- **asinkrona komunikacija**

- omogućuje pošiljatelju nastavak obrade odmah nakon slanja poruke

blokiranje

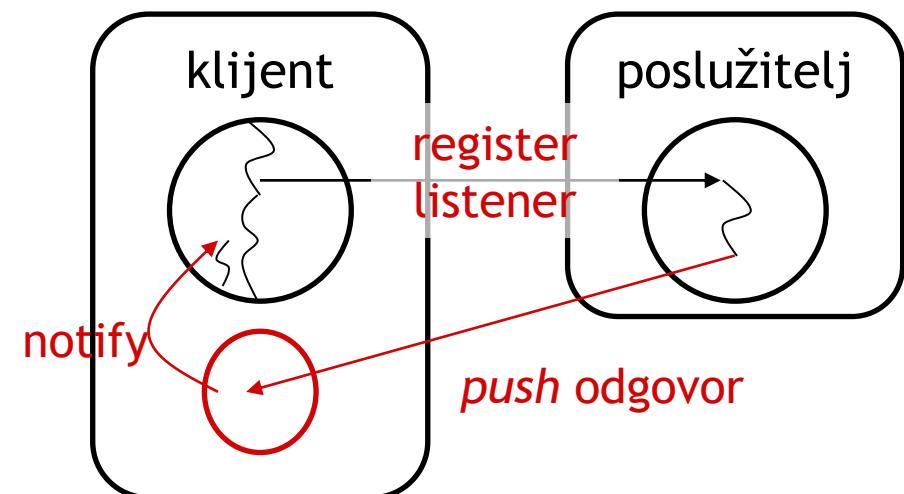
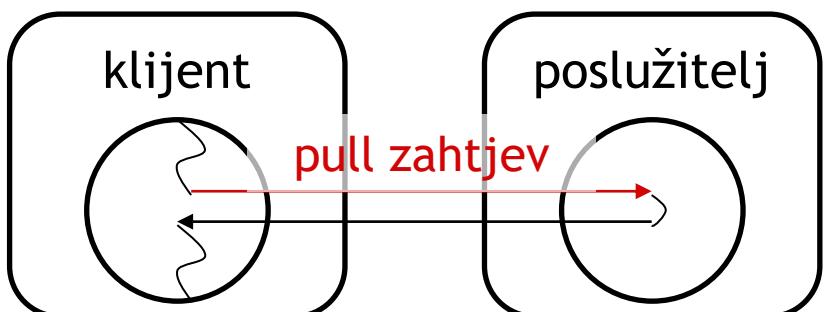


# Točke sinkronizacije



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

# Obilježja procesa

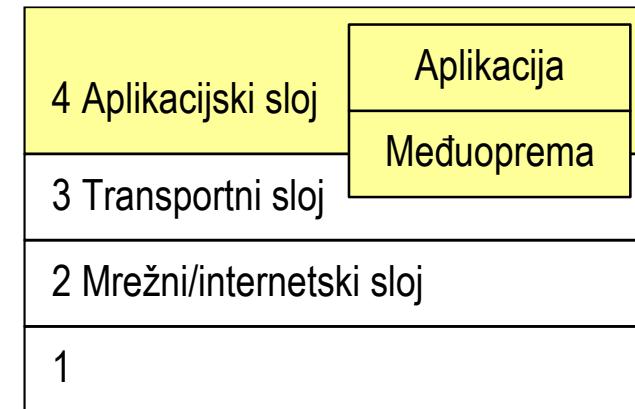
- vremenska (ne)ovisnost
  - vremenski ovisni procesi moraju biti istovremeno aktivni za realizaciju komunikacije
  - vremenski neovisni procesi mogu komunicirati i ako nisu istovremeno aktivni
- ovisnost o referenci “sugovornika”
  - proces je ovisan o referenci “sugovornika” ako mora znati jedinstveni identifikator (adresu) udaljenog procesa s kojim želi komunicirati
  - proces može biti i neovisan o referenci, tj. ne mora znati jedinstveni identifikator udaljenog procesa

# Sadržaj predavanja

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

# Sloj raspodijeljenog sustava za komunikaciju među procesima

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



# Sloj raspodijeljenog sustava za komunikaciju među procesima

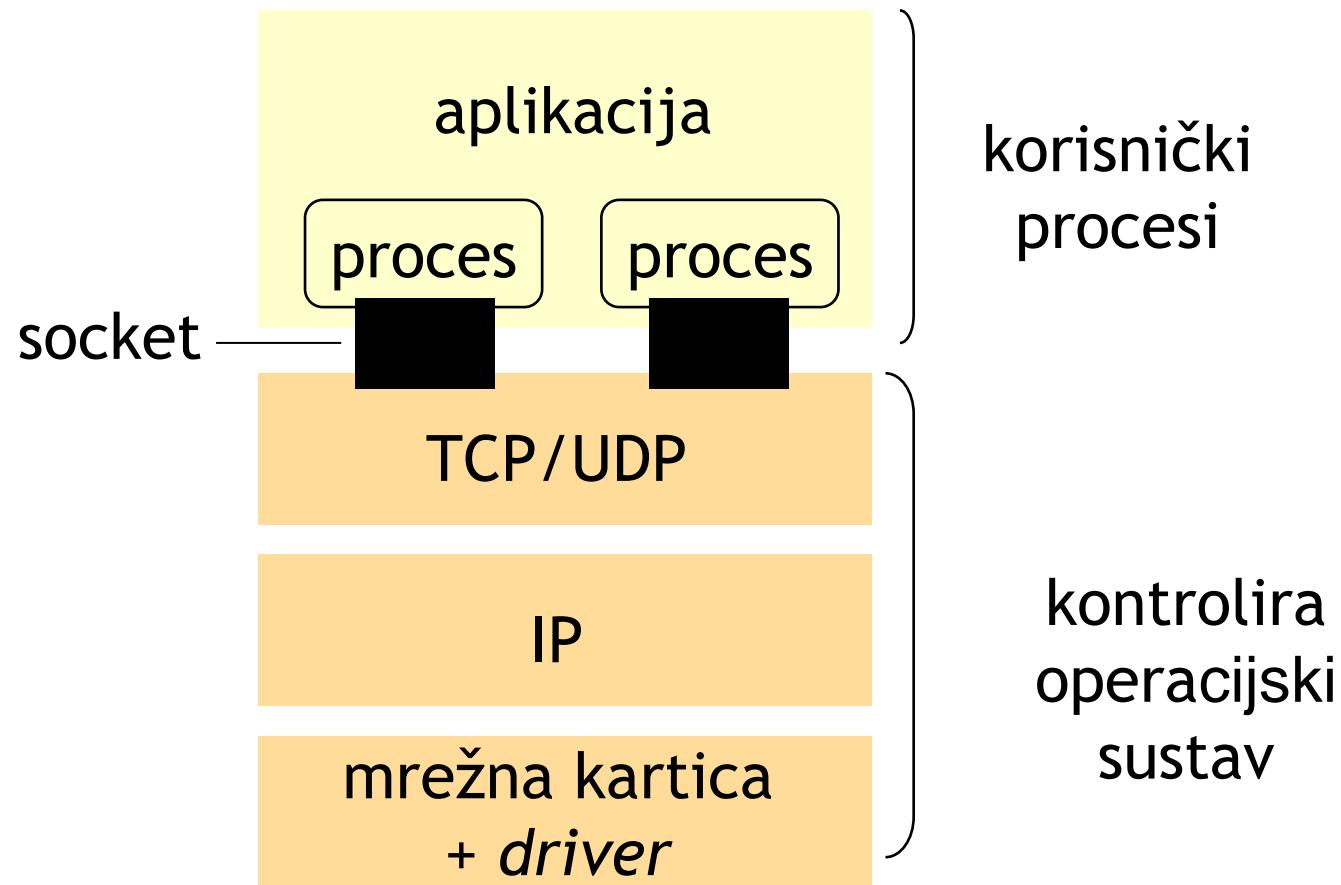
- Postojeća rješenja za komunikaciju raspodijeljenih procesa
  1. komunikacija korištenjem priključnica (*socket API*)
  2. poziv udaljene procedure (*remote procedure call, RPC*)
  3. raspodijeljeni objekti - poziv udaljene metode (*remote method invocation, RMI*)
  4. komunikacija razmjenom poruka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

# Komunikacija korištenjem priključnica

## *Socket API*

- koristi funkcionalnost transportnog sloja
  - TCP – konečkijski 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

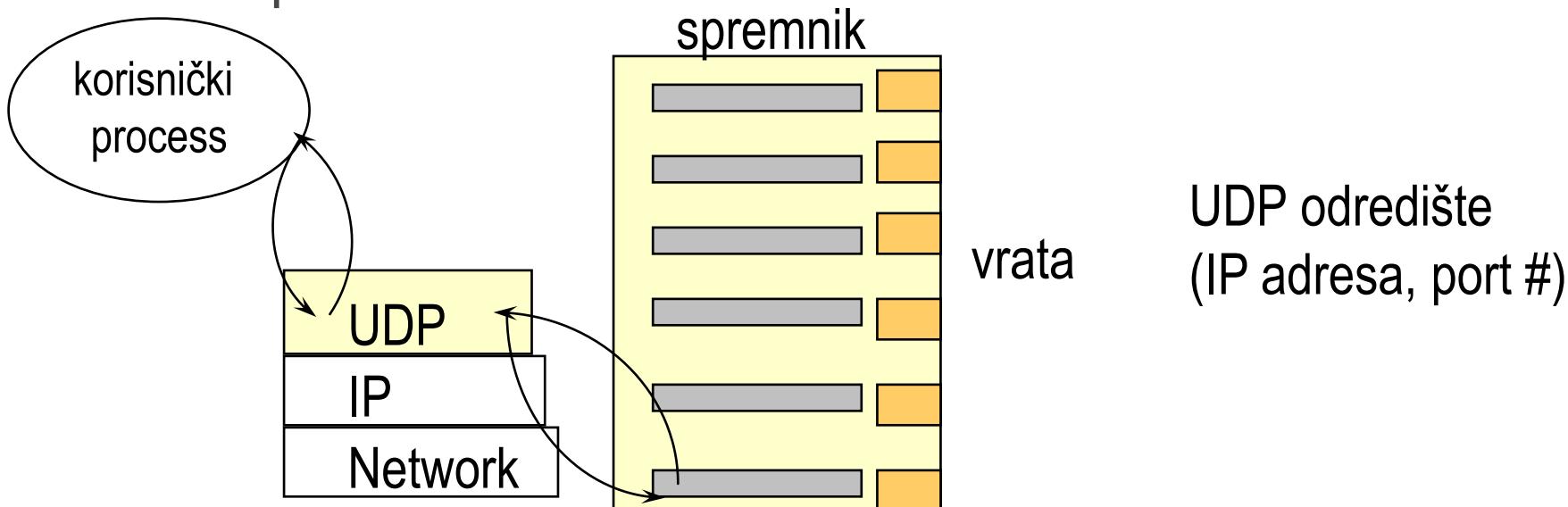
# Komunikacija pomoću socketa



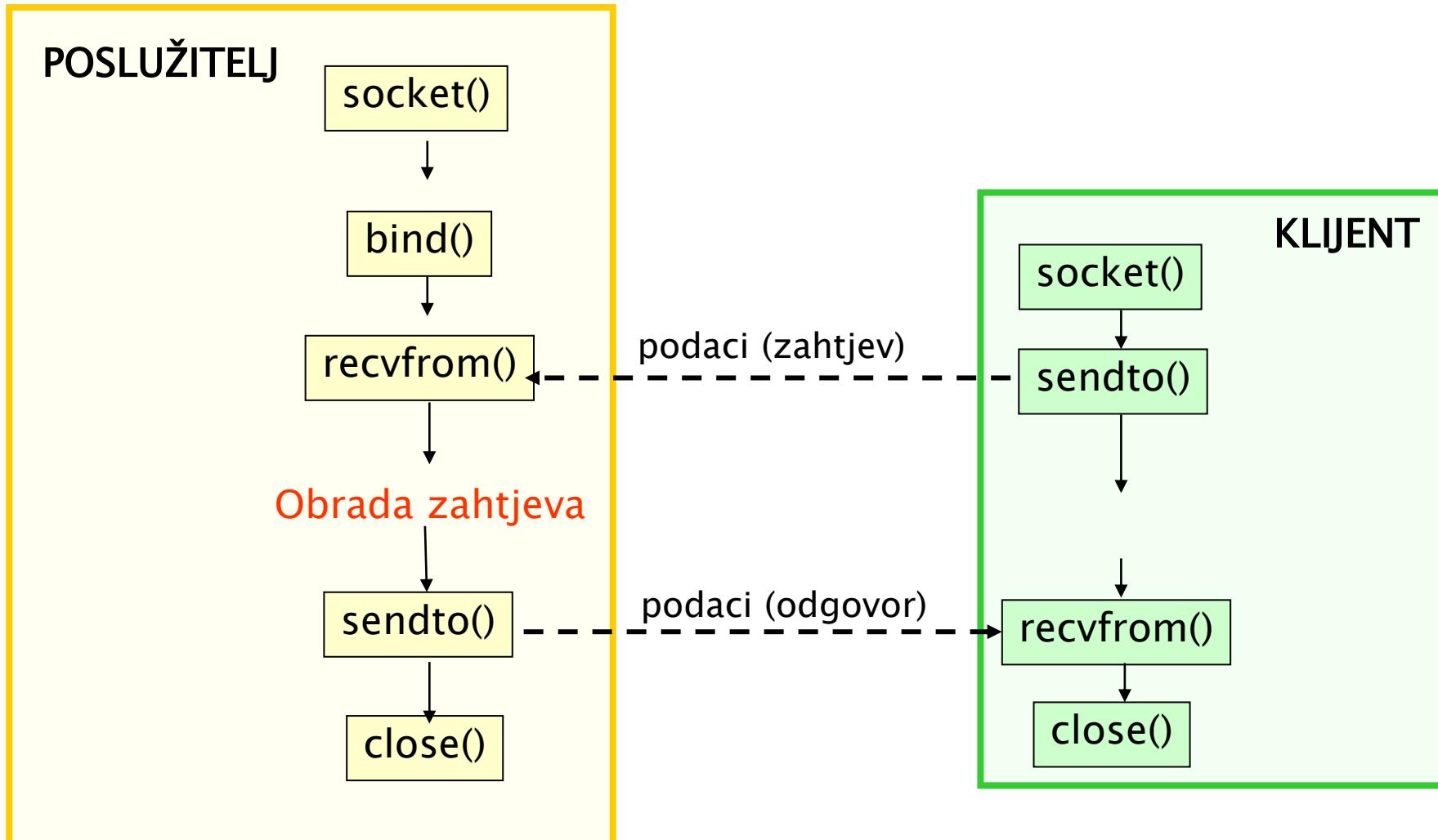
# Transportni protokol UDP

## 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*



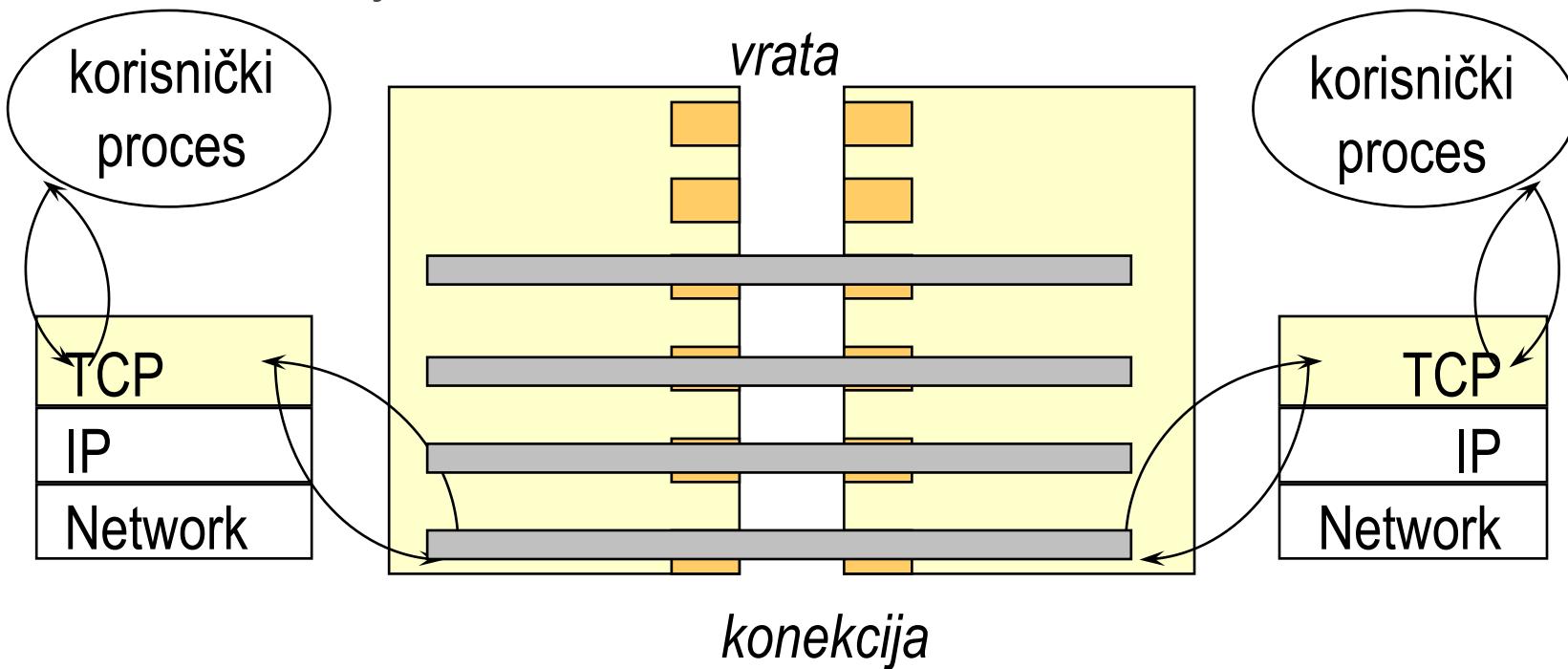
# Obilježja socketa UDP

- 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*

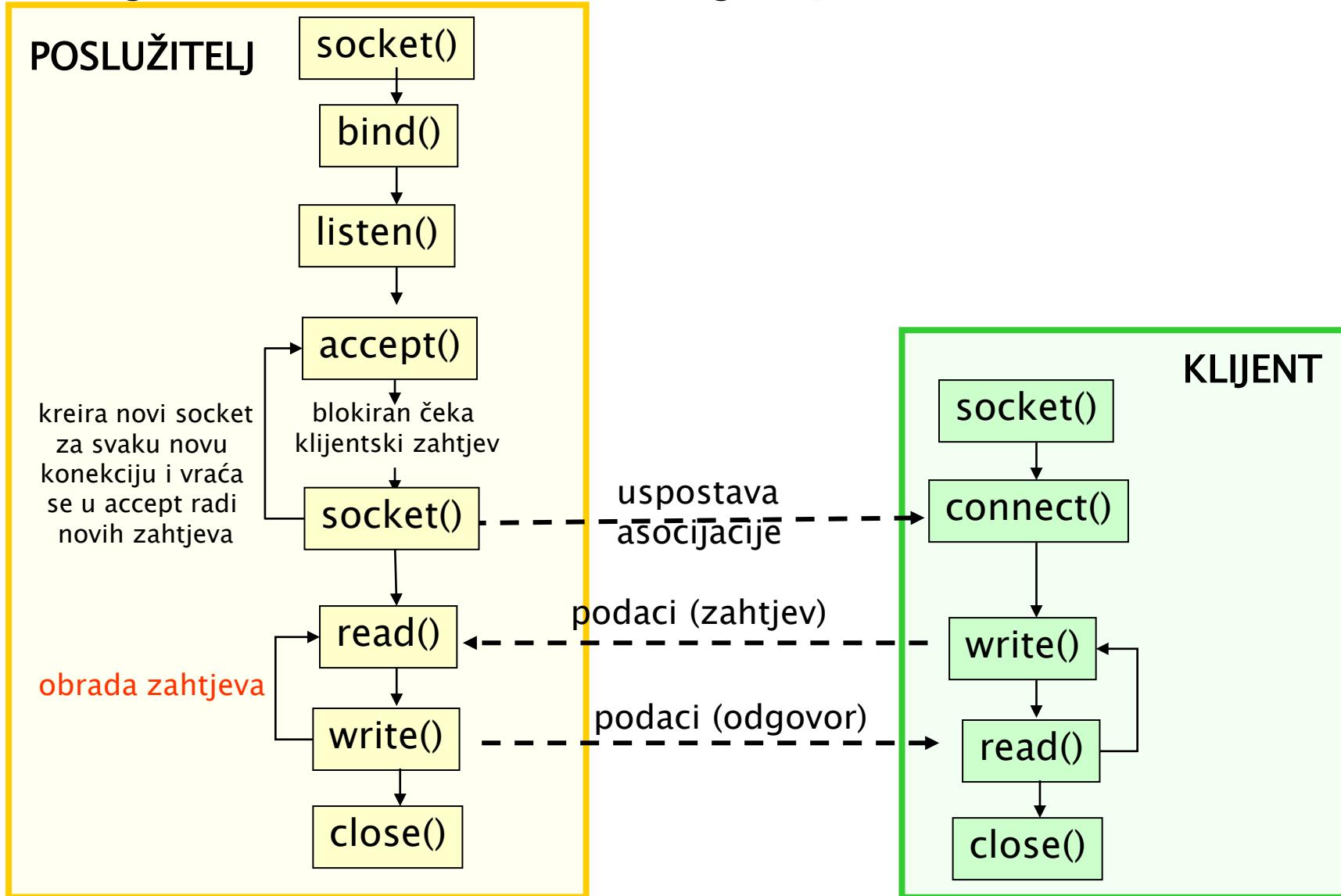
# Transportni protocol TCP

## Transmission Control Protocol (TCP)

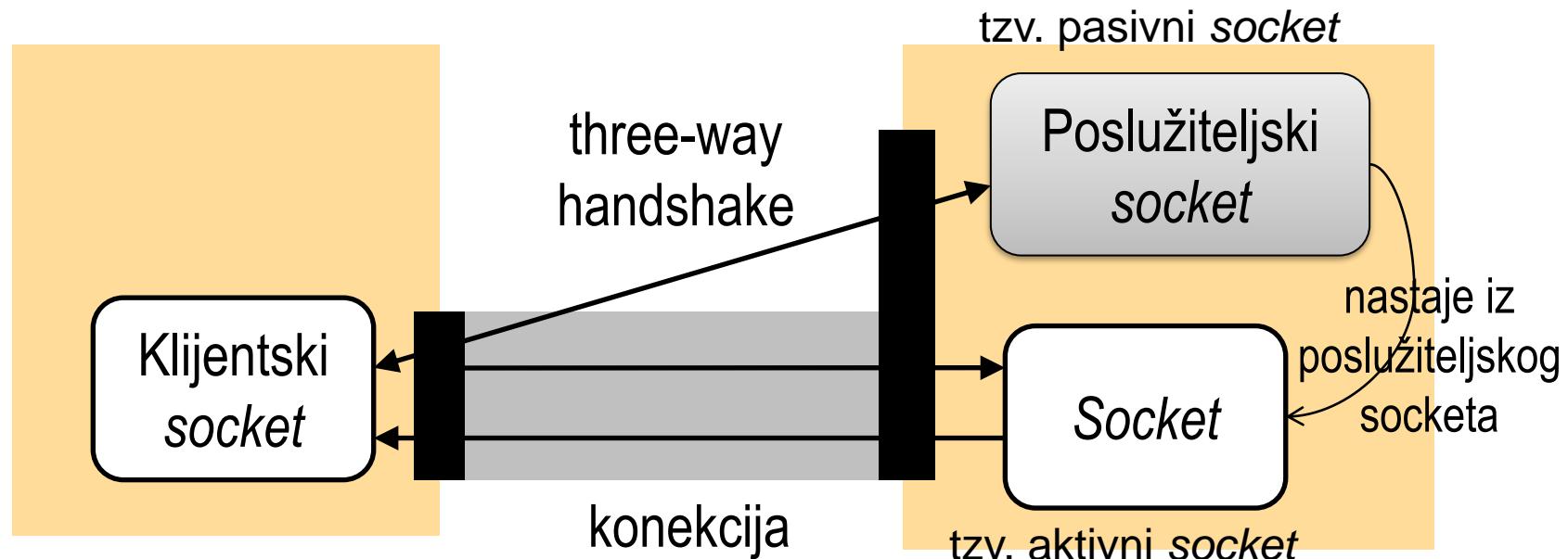
- konekcija između dvije krajnje točke koje se moraju dogovoriti o uspostavi konekcije



# 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*)

# UDP: implementacija poslužitelja

1. Kreirati socket poslužitelja:

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

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

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

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

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

4. Obrada pristiglog paketa i po potrebi odgovor klijentu

5. Zatvoriti socket (gasi poslužitelja):

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

# UDP: implementacija klijenta

1. Kreirati socket:

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

2. Kreirati paket i napuniti ga podacima:

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

3. Slanje paketa:

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

4. Po potrebi obrada i čekanje odgovora

5. Zatvoriti socket:

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

# TCP: implementacija poslužitelja

1. Kreirati socket poslužitelja:

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

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

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

3. Kreirati I/O stream za komunikaciju s klijentom

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

4. Komunikacija s klijentom

5. Zatvoriti kopiju socketa:

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

6. Zatvoriti poslužiteljski socket:

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

# TCP: implementacija klijenta

## 1. Kreirati klijentski socket:

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

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

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

## 3. Komunikacija s poslužiteljem:

- //Receive data from server:  

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

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

## 4. Zatvoriti socket:

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

# Paket `java.net`

- **API specification**

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

- Osnovne klase

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

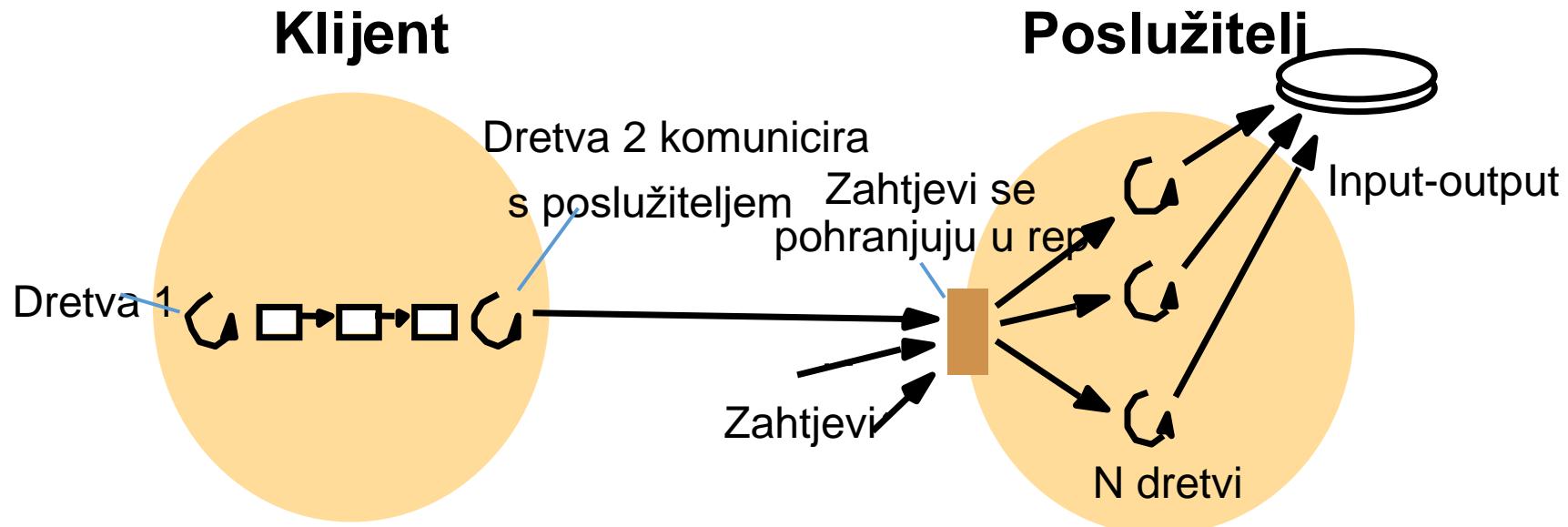
- Java Networking Tutorial

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

# Sadržaj predavanja

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

# Višedretveni poslužitelj i klijenti



Uobičajene zadaće na strani klijenta:

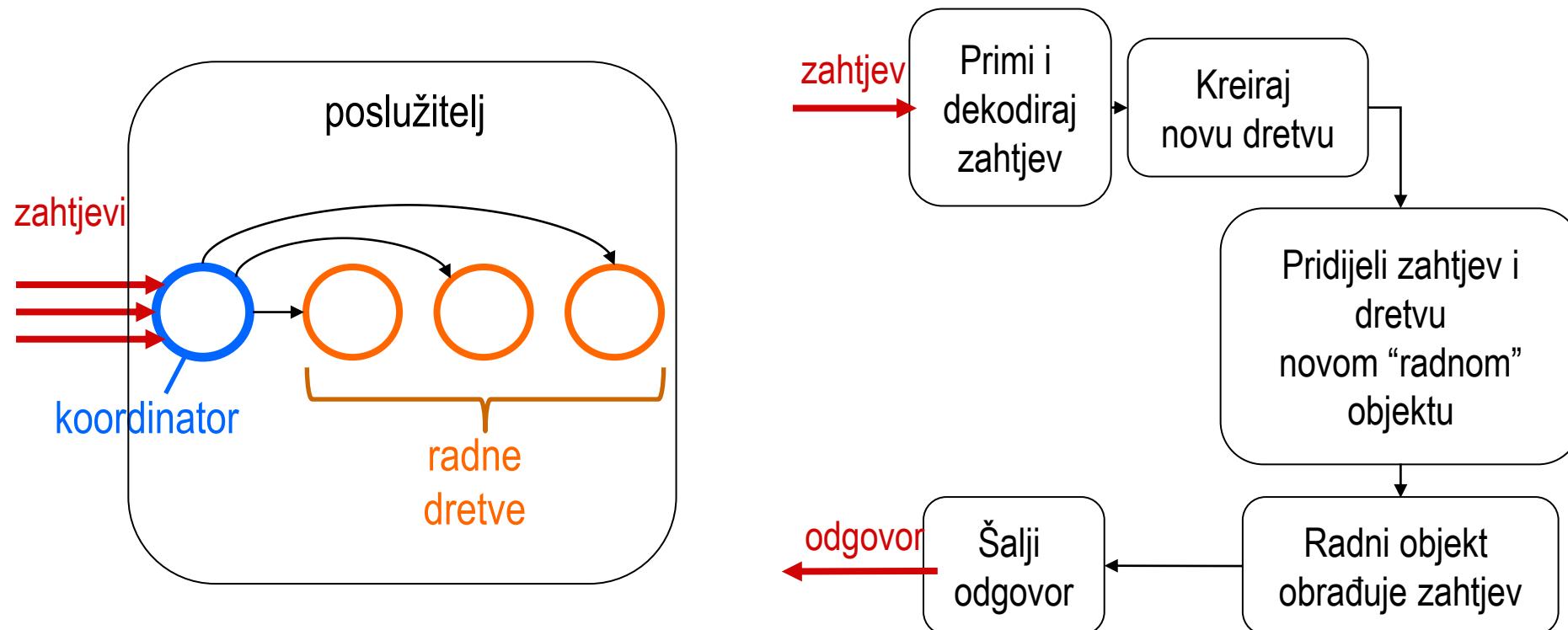
- korisničko sučelje,
- otvaranje mrežne konekcije i primanje podataka

Uobičajene zadaće na strani poslužitelja:

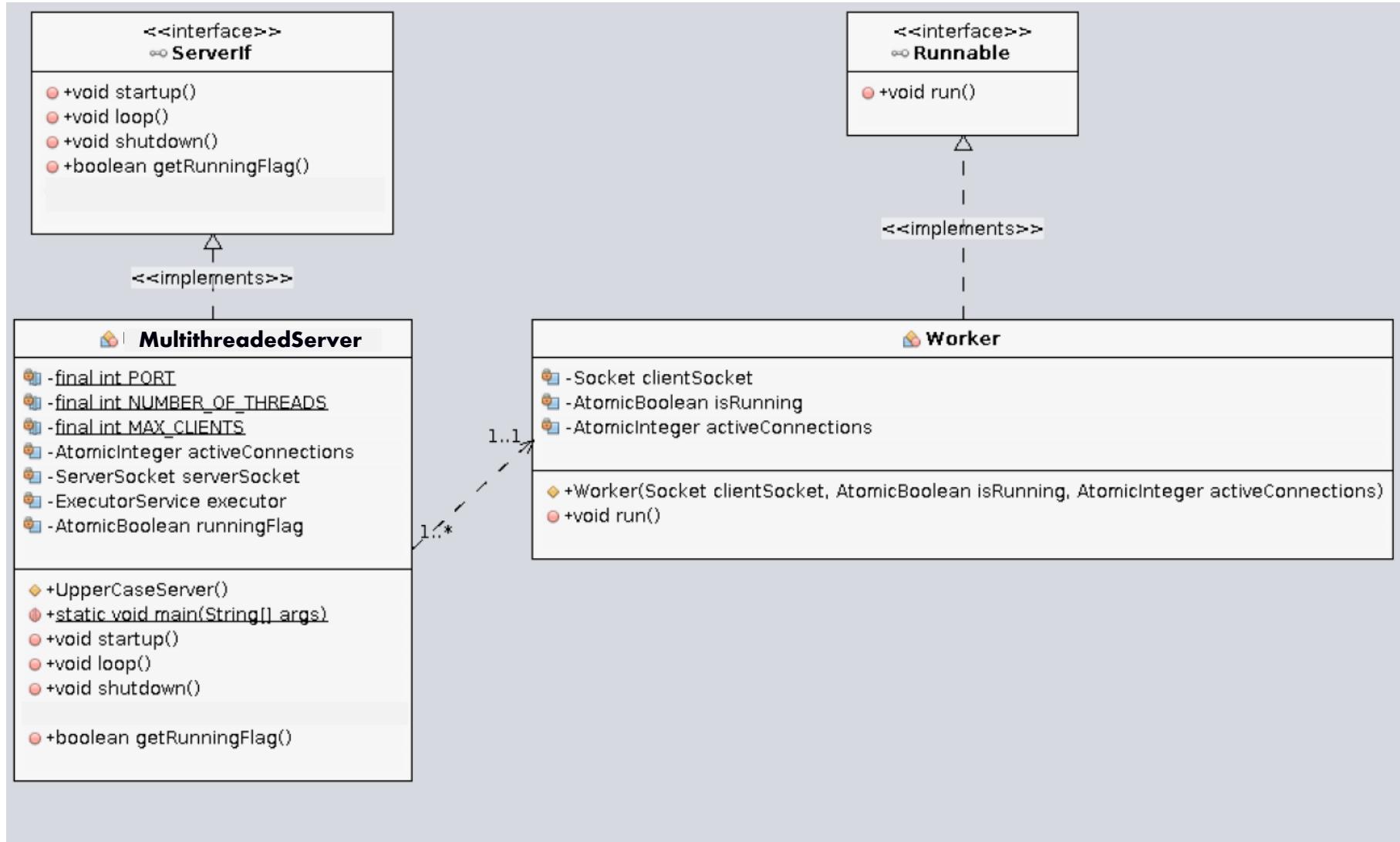
- primanje konkurentnih klijentskih zahtjeva
- složena obrada podataka
- rad s diskom/bazom podataka

# Višedretveni poslužitelj

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



# Primjer višedretvenog poslužitelja



# Sučelje višedretvenog poslužitelja

```
public interface ServerIf {
    // Server startup. Starts all services offered by the server.
    public void startup();

    // Server loops when in running mode. The server must be active
    // to accept client requests.
    public void loop();

    // Server shutdown. Shuts down all services started during
    // startup.
    public void shutdown();

    // Gets the running flag that indicates server running status.
    // @return running flag
    public boolean getRunningFlag();
}
```

# Poslužitelj (1)

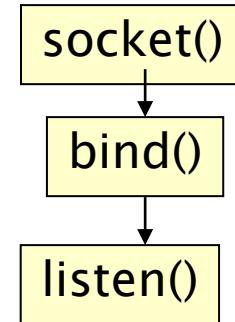
```
public class MultithreadedServer implements ServerIf {  
  
    private static final int PORT = 10002; // server port  
    private static final int NUMBER_OF_THREADS = 4;  
    //Max queue length for incoming connection requests.  
    private static final int BACKLOG = 10;  
  
    private final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

# Poslužitej (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();
}
...
```

# Poslužitej (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);  
    }  
}  
}...
```



# Poslužitej (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);  
        }  
    }  
}
```

# Poslužitej (5)

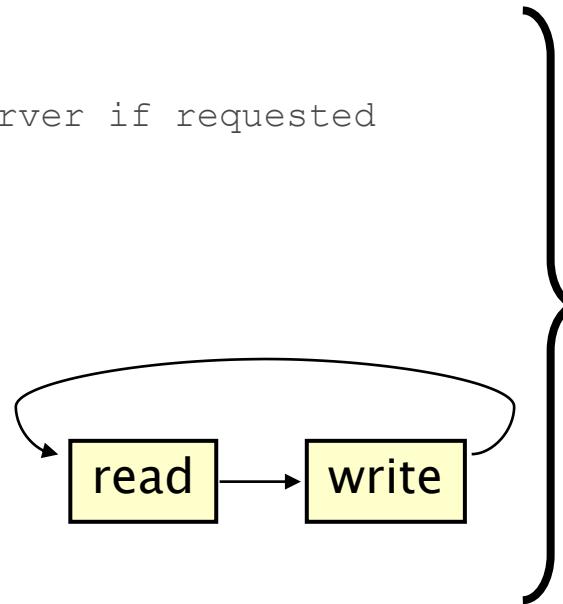
```
@Override  
public void shutdown() {  
    while( activeConnections.get() > 0 ) {  
        System.out.println( "WARNING: There are still active  
                           connections" );  
        try { Thread.sleep( 5000 );  
        } catch( java.lang.InterruptedException e ) {}  
    }  
 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

# Worker (1)

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

# Worker (2)

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

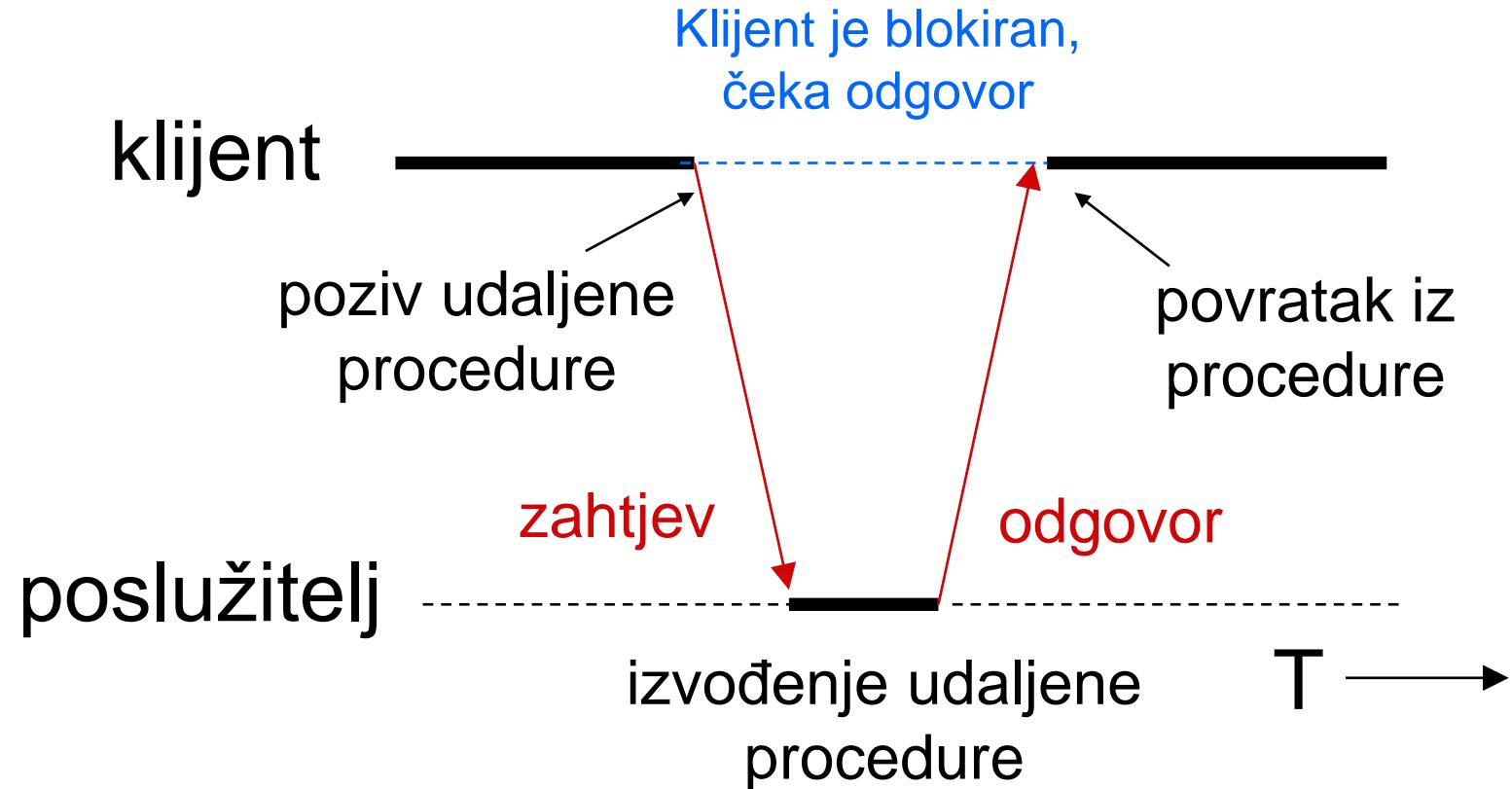


# Sadržaj predavanja

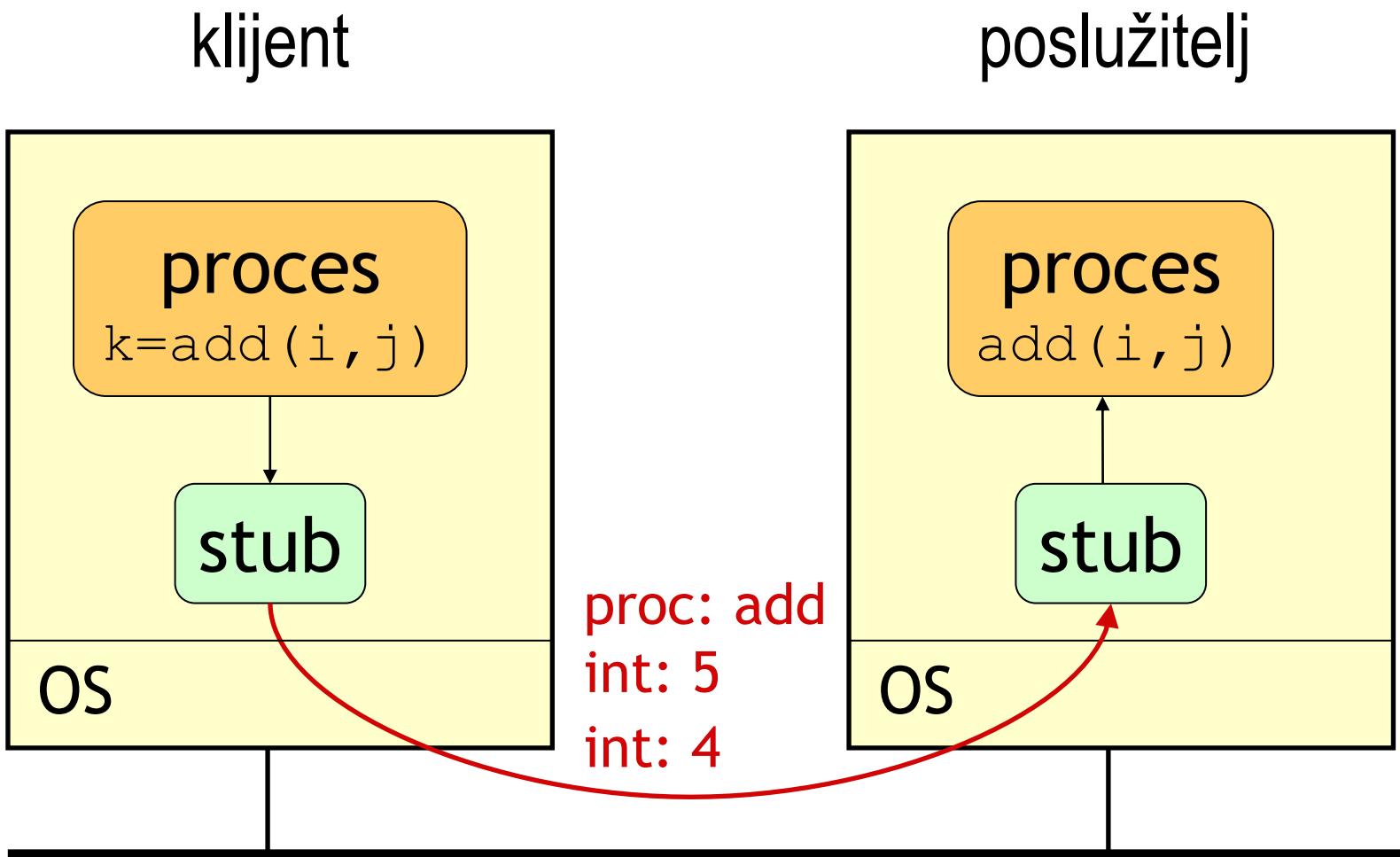
- Osnovni komunikacijski model
  - obilježja komunikacije
  - obilježja procesa
- Sloj raspodijeljenog sustava za komunikaciju među procesima
  - komunikacija korištenjem priključnica (Socket API)
    - primjeri TCP/UDP klijenta i poslužitelja
    - oblikovanje višedretvenog poslužitelja
  - poziv udaljene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

# Poziv udaljene procedure (RPC)

- Omogućuje procesima pozivanje i izvođenje procedura na udaljenom računalu.



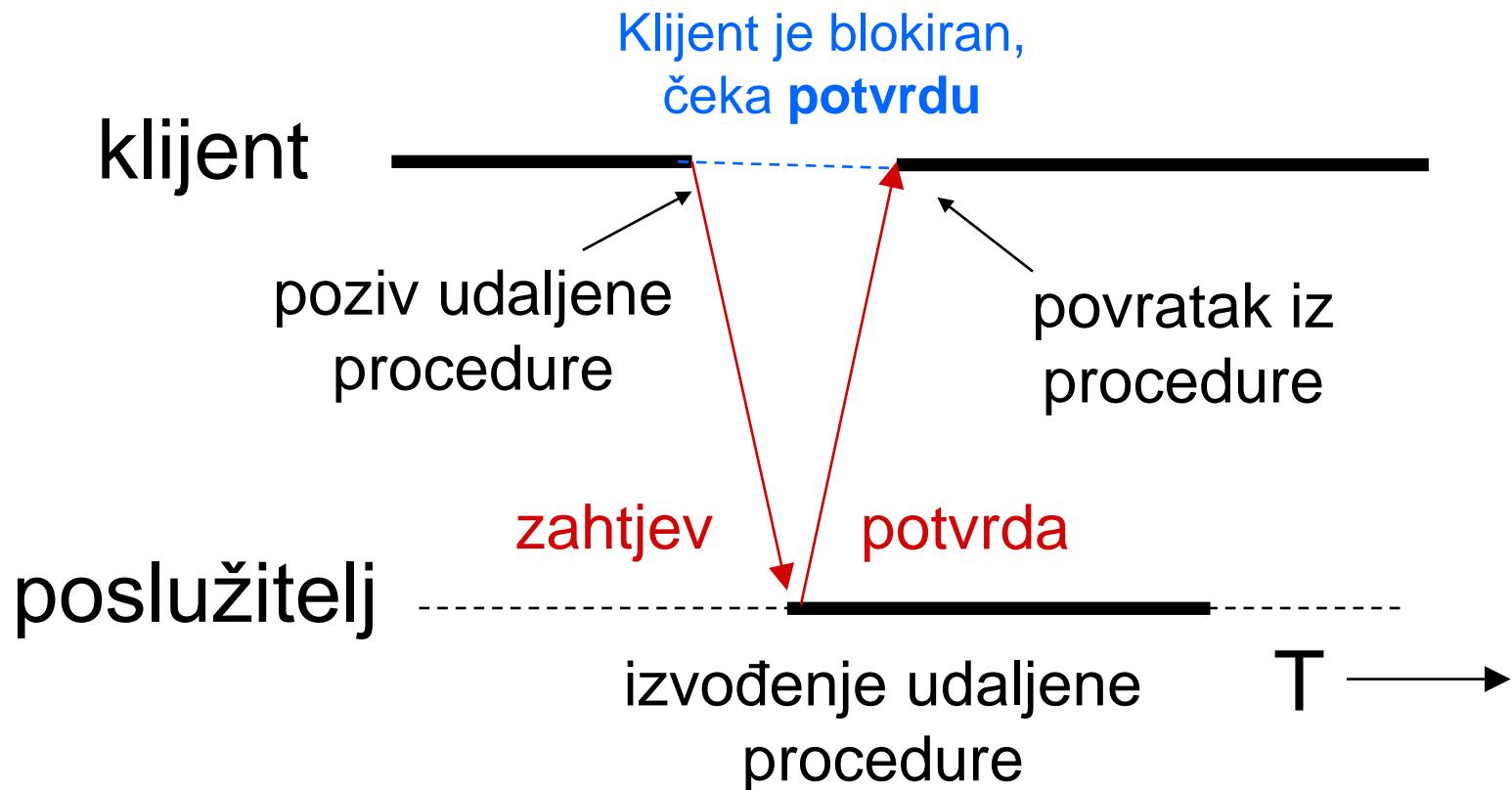
# Izvođenje RPC-a



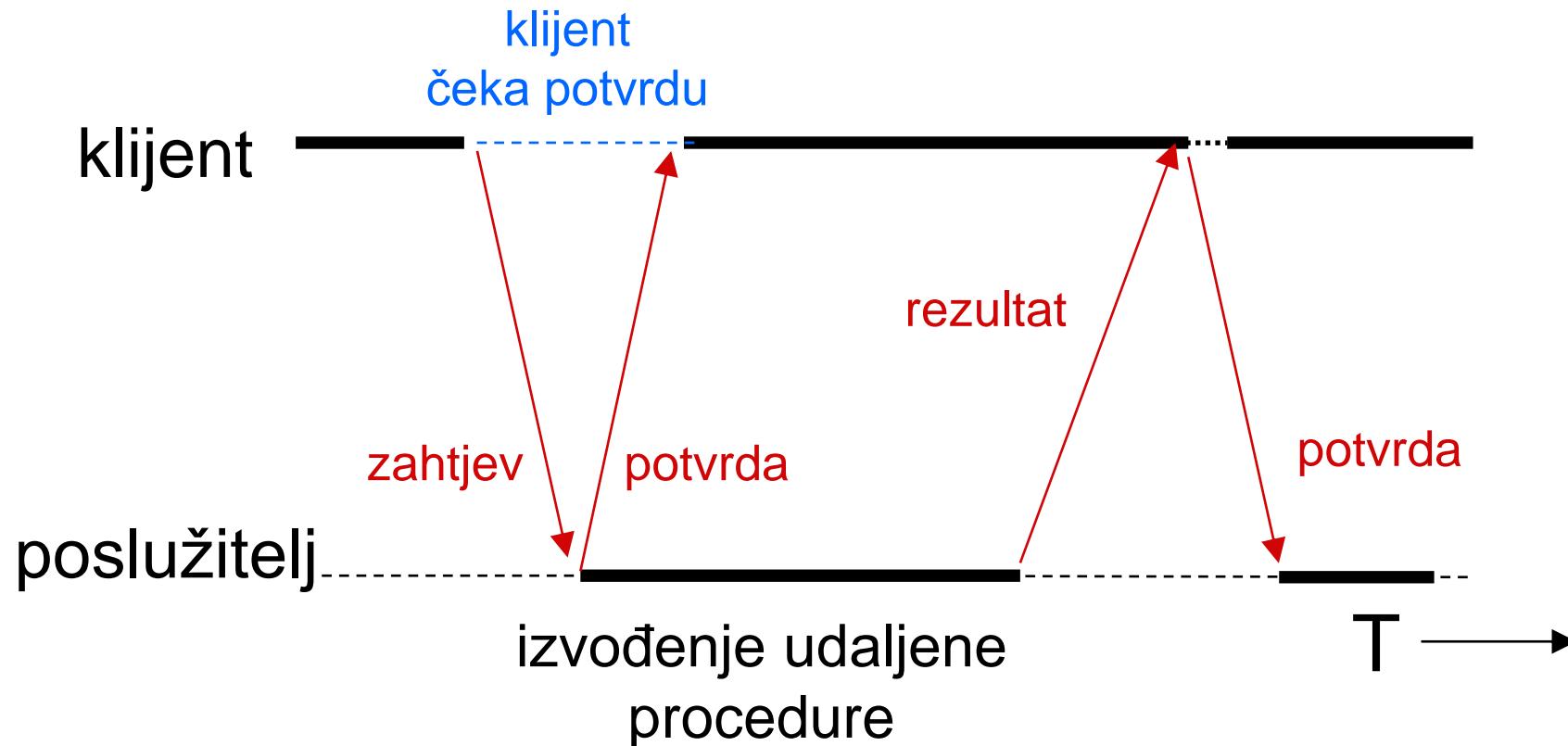
# Prenošenje parametara

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

# Asinkroni RPC



# Odgodjeni sinkroni RPC



# Sadržaj predavanja

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

# Poziv udaljene metode

## Remote Method Invocation (RMI)

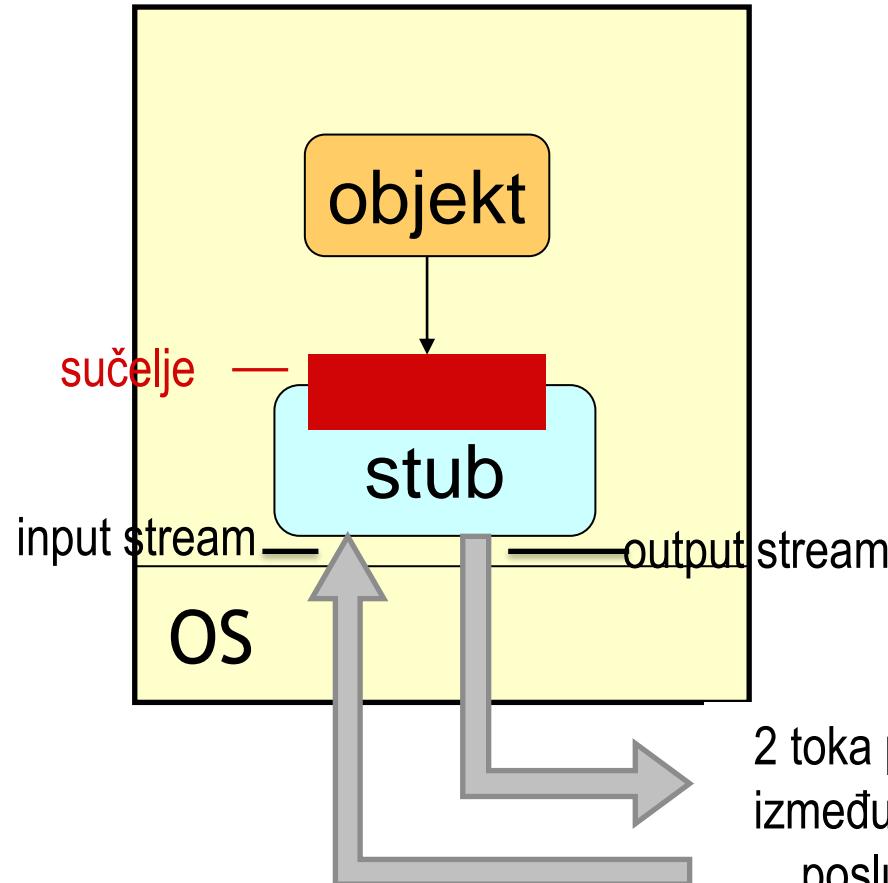
- “nasljednik” poziva udaljene procedure, poziva se metoda udaljenog objekta
- udaljeni objekt
  - proširenje osnovnog objektnog modela za raspodijeljenu okolinu
  - odvajanje sučelja i implementacije objekta
- objekt (klijent) poziva metodu udaljenog objekta (poslužitelja) na transparentan način
  -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

# Udaljeni objekti

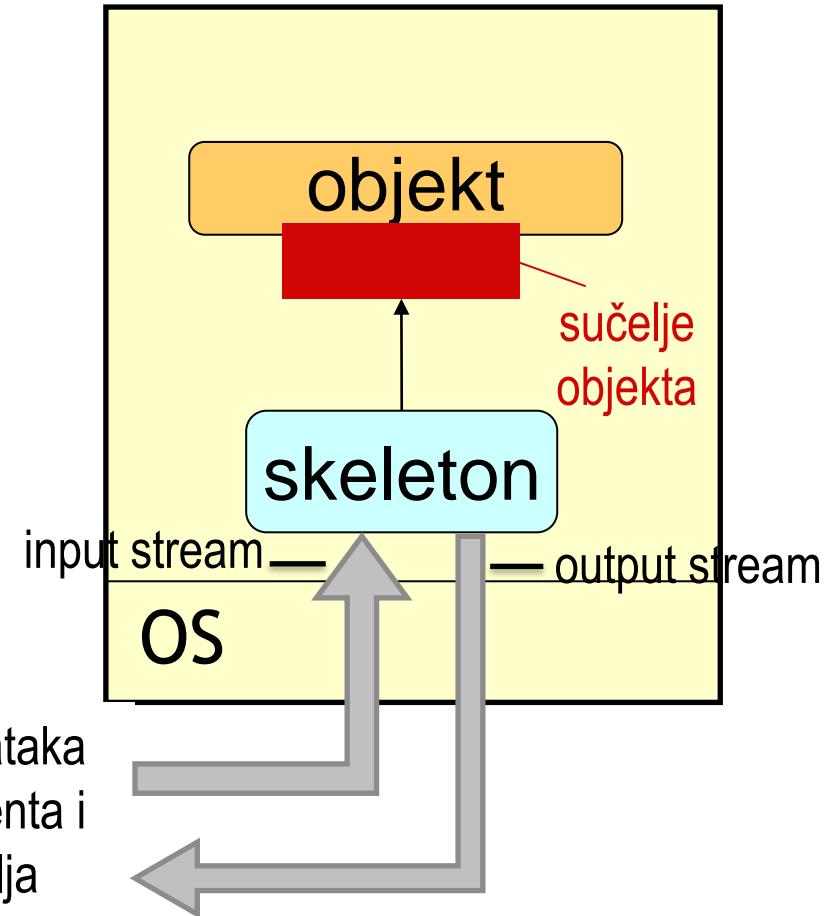
- Postoje reference na lokalne i udaljene objekte
- Svaki udaljeni objekt ima globalno jedinstven identifikator
  - npr. [ref: [endpoint:[161.53.19.24:1251](local),objID:[0]]]]
- Potrebna je usluga za registriranje i pronalaženje udaljenih objekata (*directory service*)

# Izvođenje RMI-a

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*

# 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

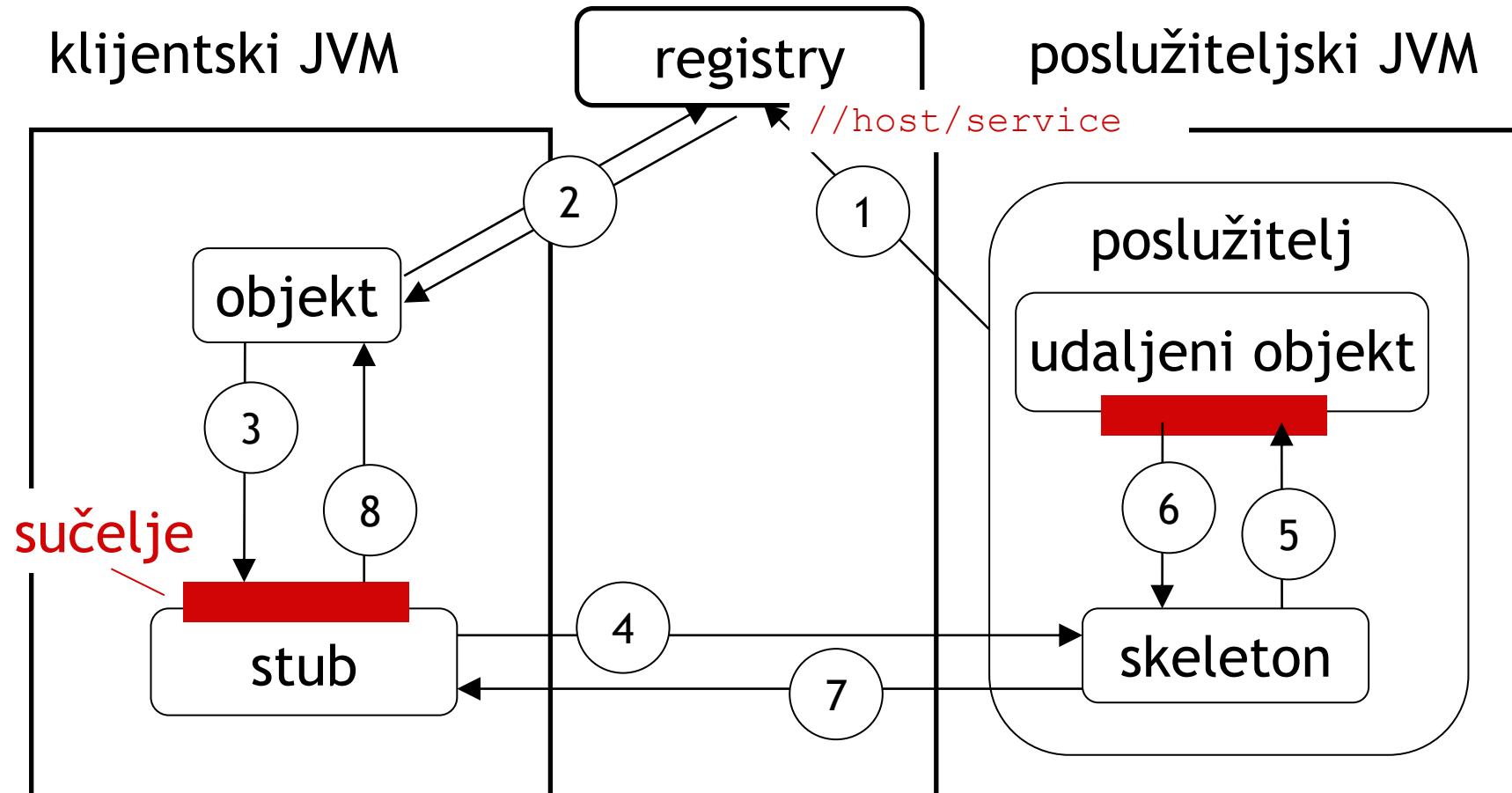
# Javni model objekta

- 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

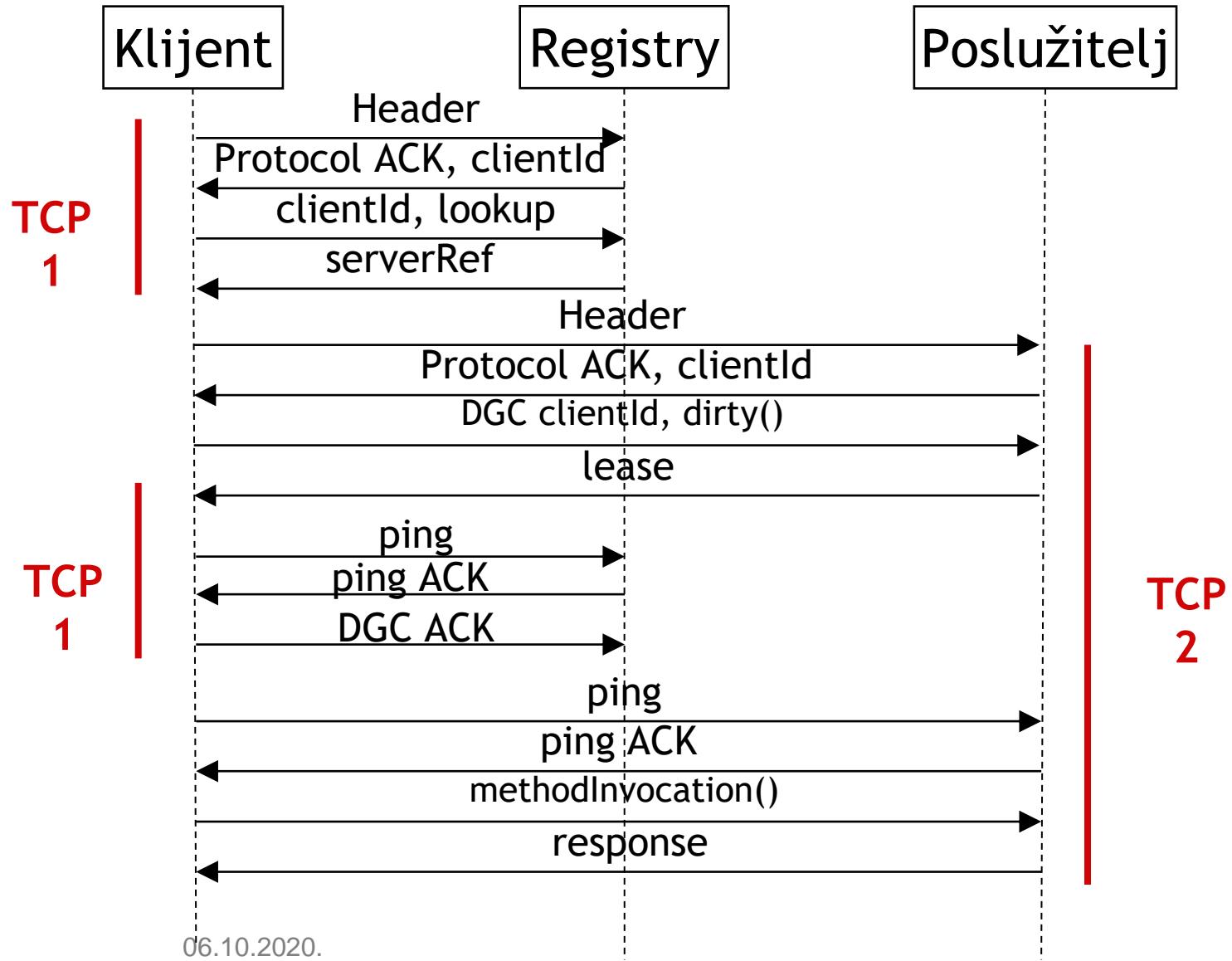
- lokalni objekti moraju se serijalizirati i prenosi se njihova vrijednost (*pass by value*)
  - implementiraju sučelje Serializable
- udaljeni se objekti prenose koristeći referencu (*pass by reference*)
  - implementiraju sučelje java.rmi.Remote i pravilno su eksportirani UnicastRemoteObject.exportObject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

# Protokol Java RMI (1)

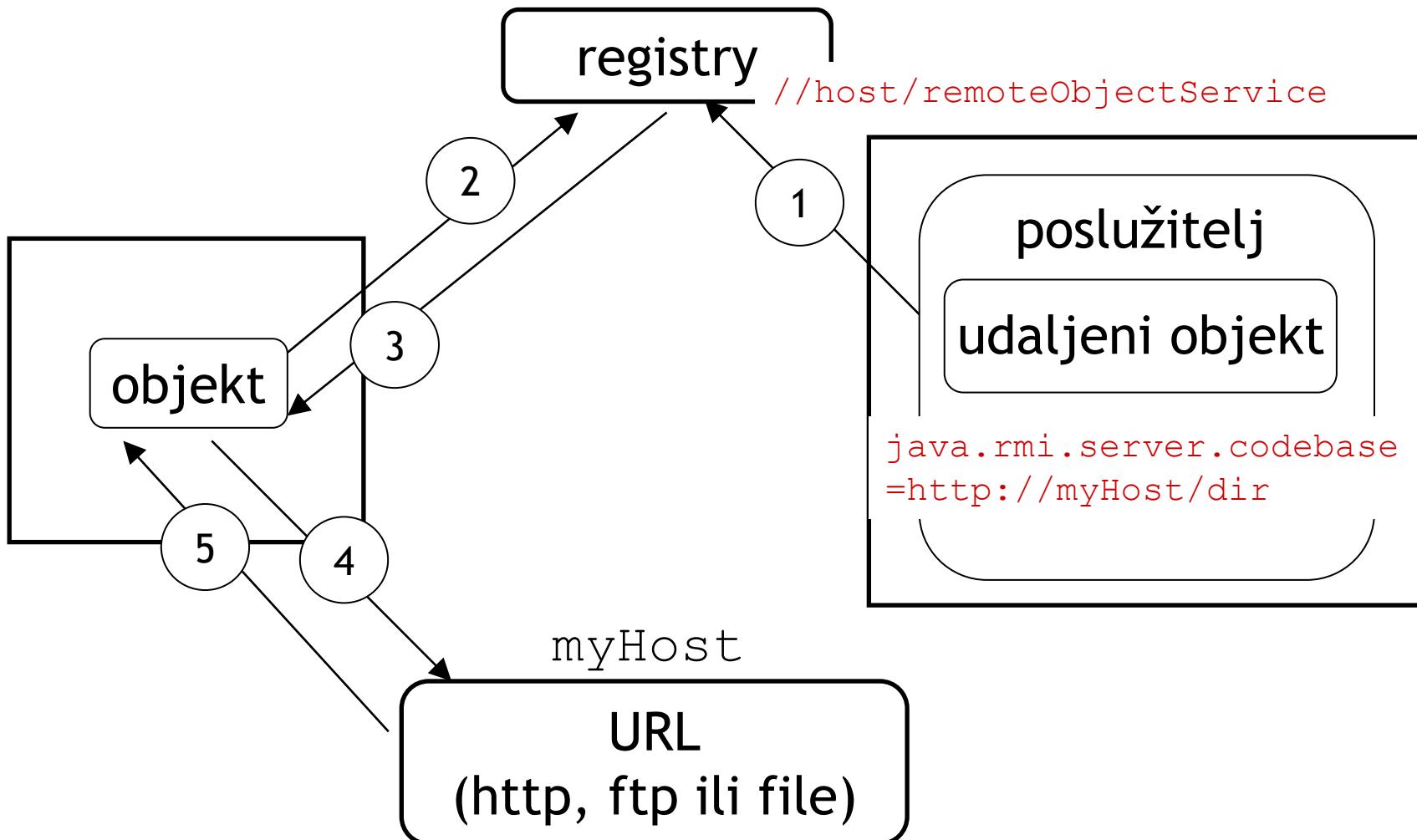


Prepostavka: stub postoji na strani klijenta

# Protokol Java RMI (2)



# Dinamičko učitavanje klase stuba (1)



# Primjer RMI sučelja

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

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

    public String 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/>

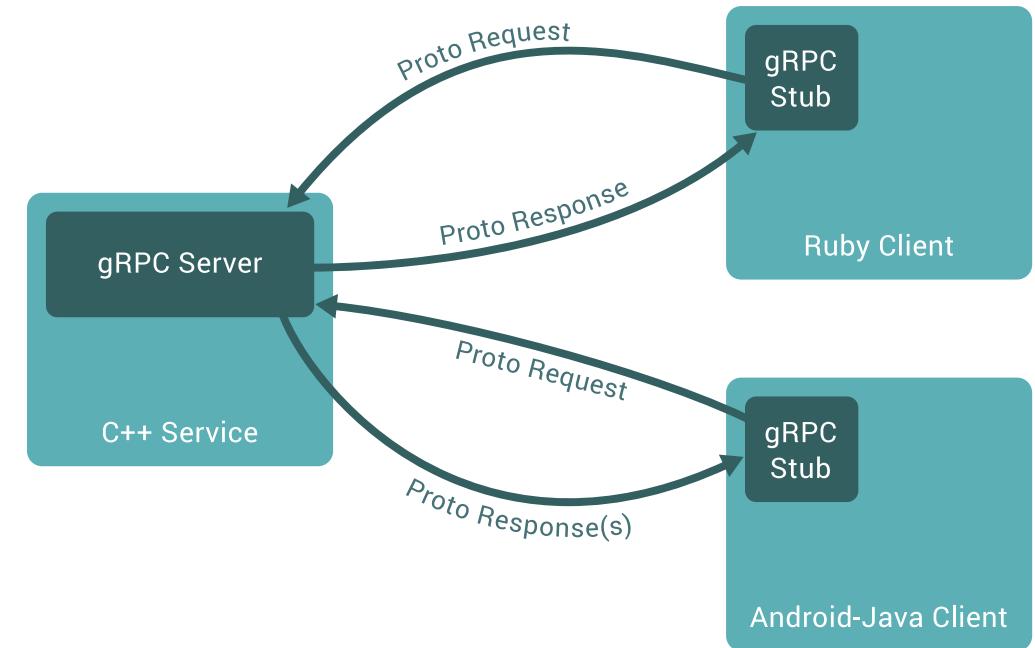
- **Java Remote Method Invocation - Distributed Computing for Java**

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

# gRPC

<https://grpc.io/>

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



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

# Pitanja za učenje i ponavljanje

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

# Literatura

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



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**3. Arhitekture web-aplikacija,  
tehnologije weba**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

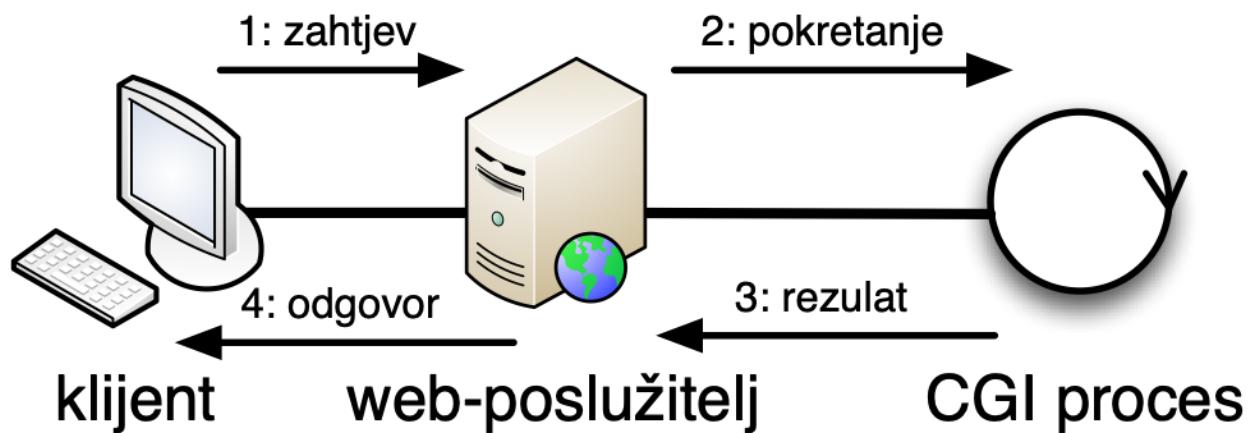
- arhitekture web-aplikacija
- AJAX
- uvod u Web 2.0
- Web-usluge
- jezici i protokoli: SOAP, WSDL, UDDI
- web-usluge temeljene na RPC-u, temeljene na dokumentima
- Web-usluge temeljene prijenosu prikaza stanja resursa (REST)
- Svojstva metoda protokola HTTP
- Model zrelosti web-usluga i relevantni standardi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

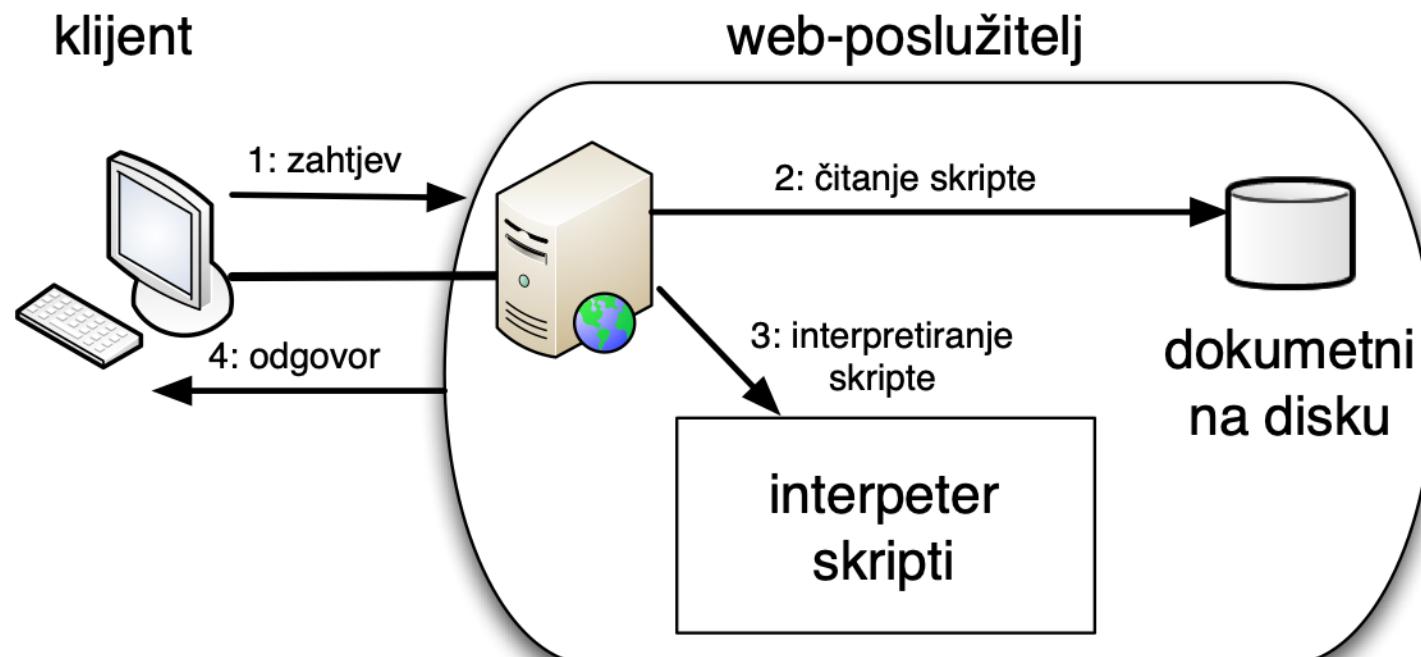
# Web-aplikacije - CGI

- 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



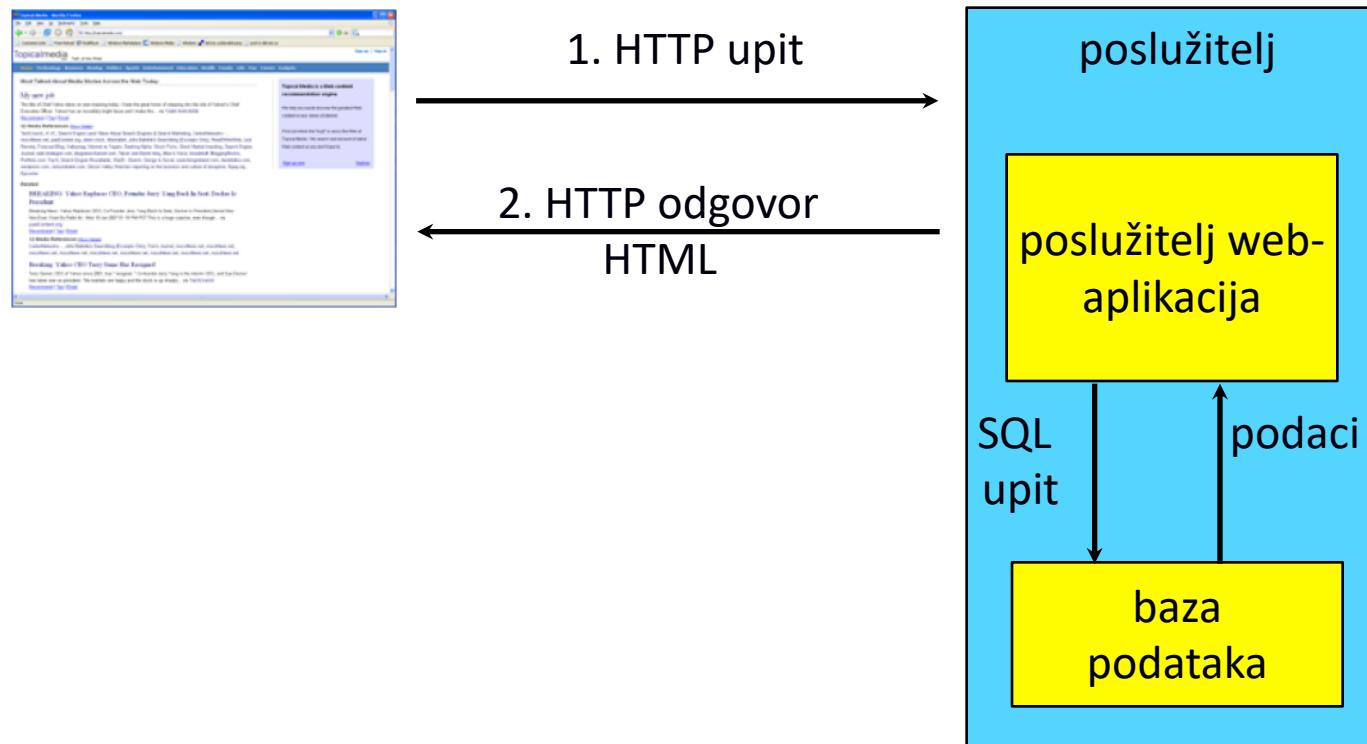
# Web-aplikacije - poslužiteljske skripte

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



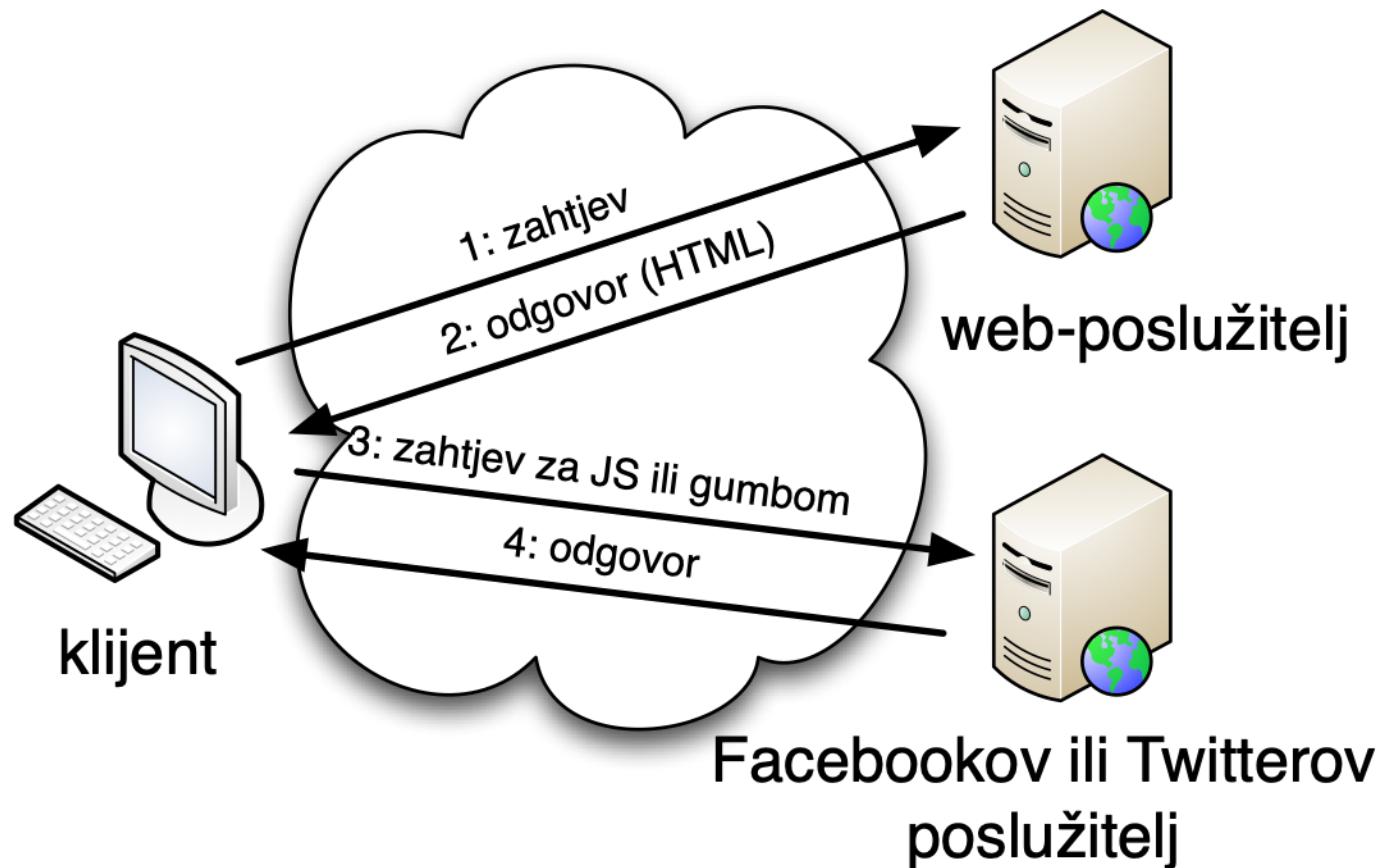
# Tipična web-aplikacija na jednom računalu

- razvijateljska konfiguracija
- dobro za mali broj zahtjeva u produkciji



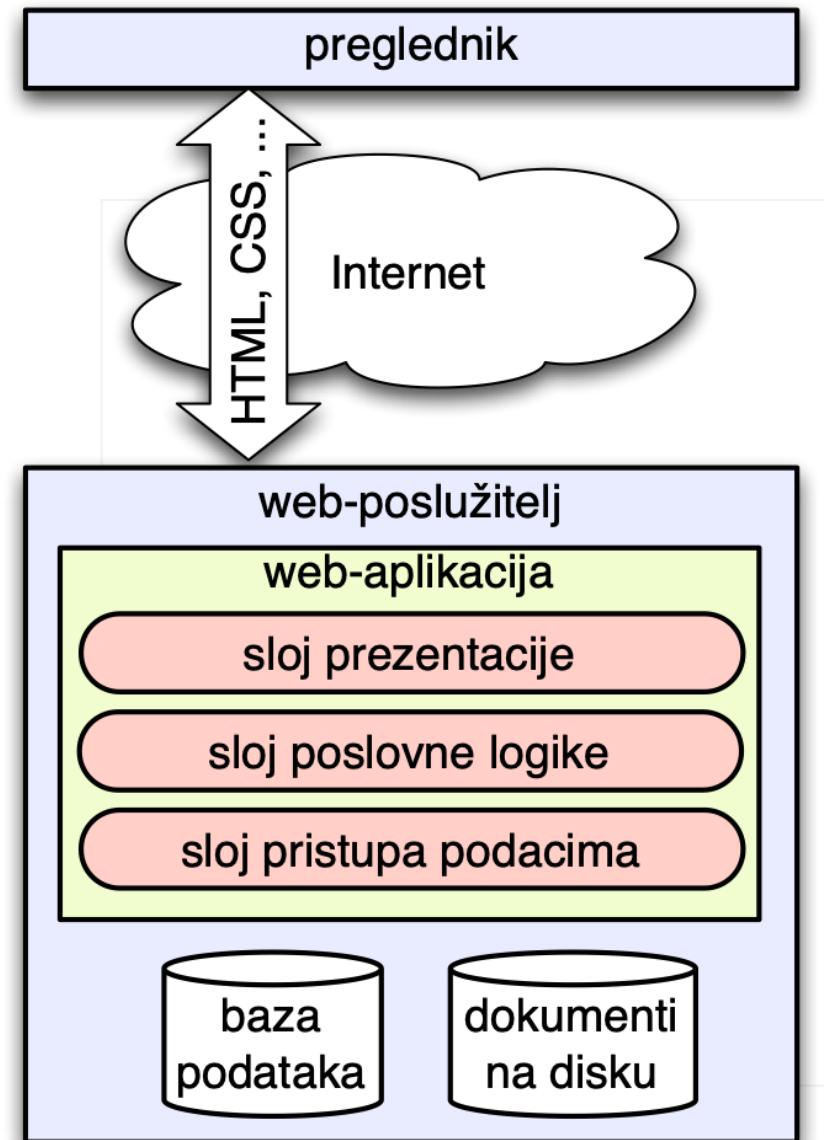
# Web-aplikacija integrirana s drugim sjedištima

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



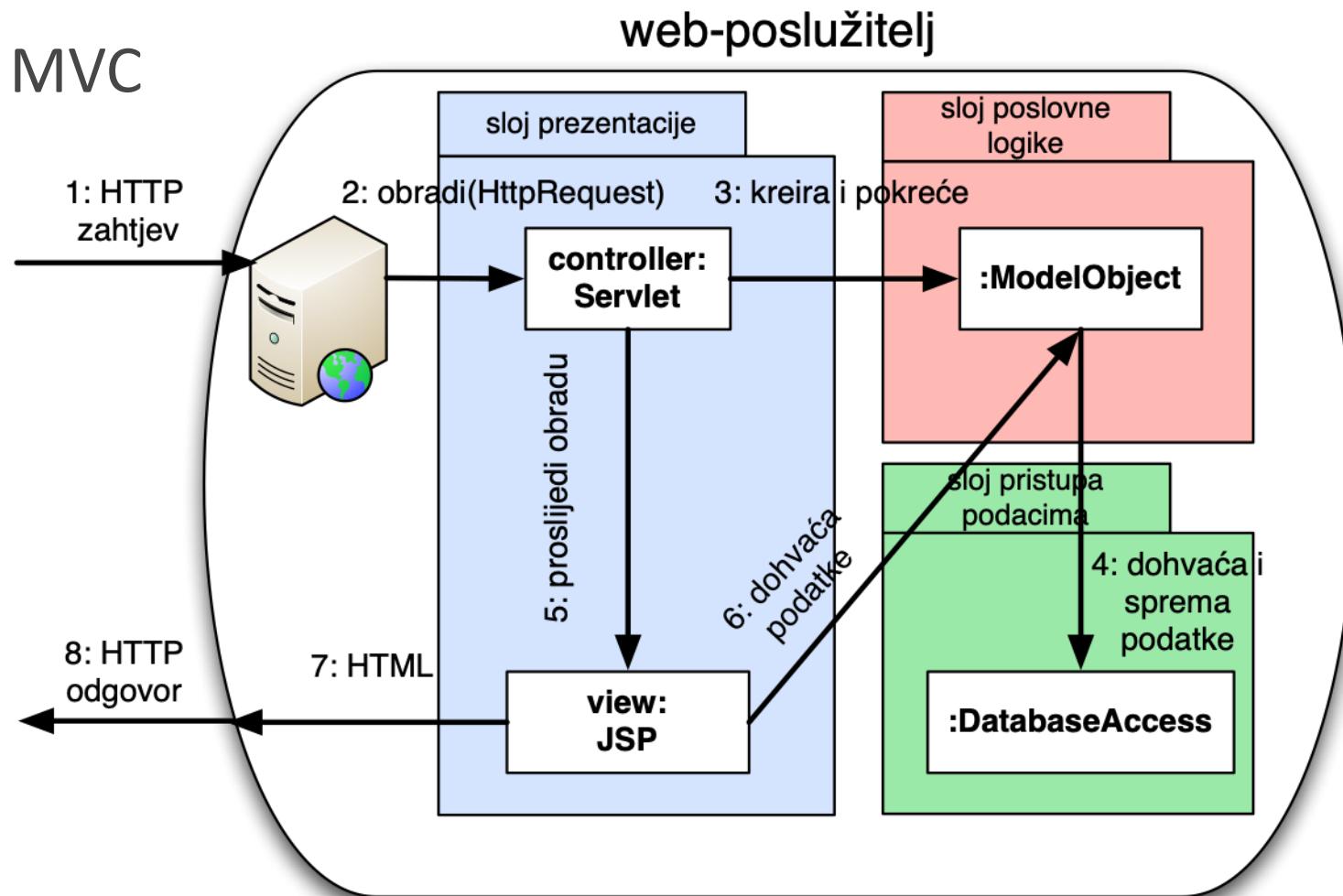
# Arhitektura web-aplikacija

- 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



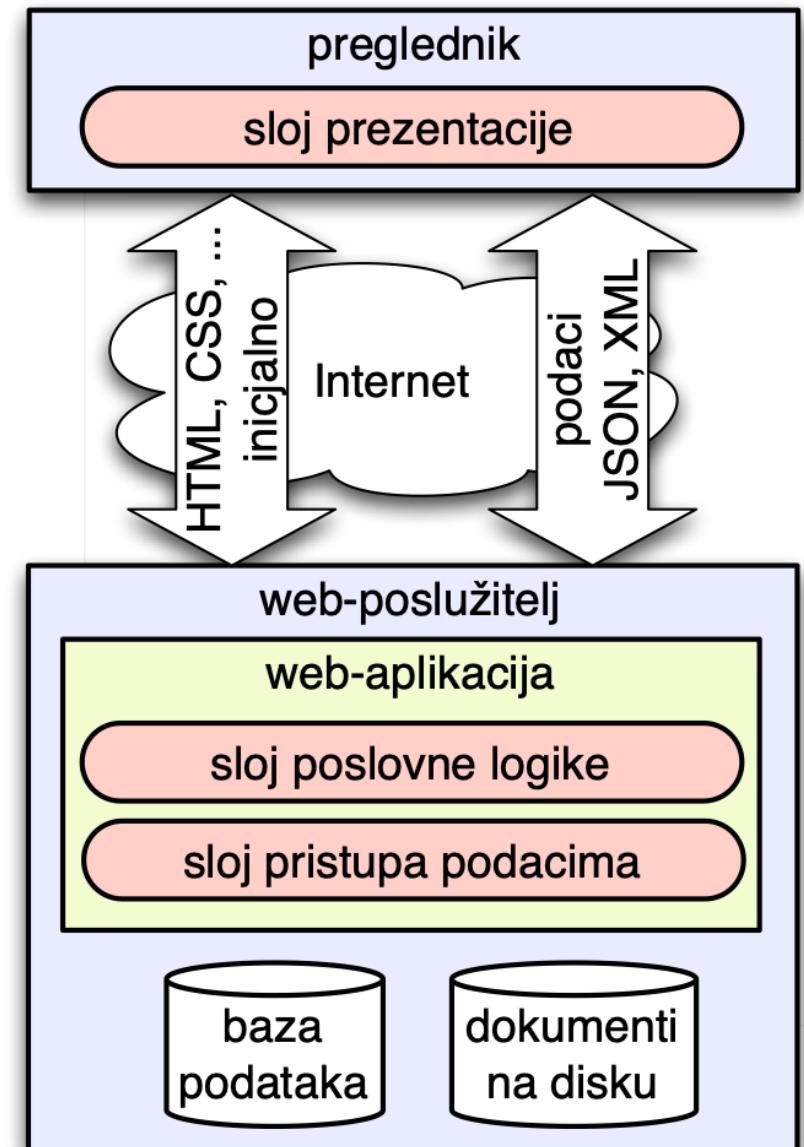
# Arhitektura web-aplikacija - MVC u Javi

- *Model View Controller - MVC*



# Nova arhitektura web-aplikacija

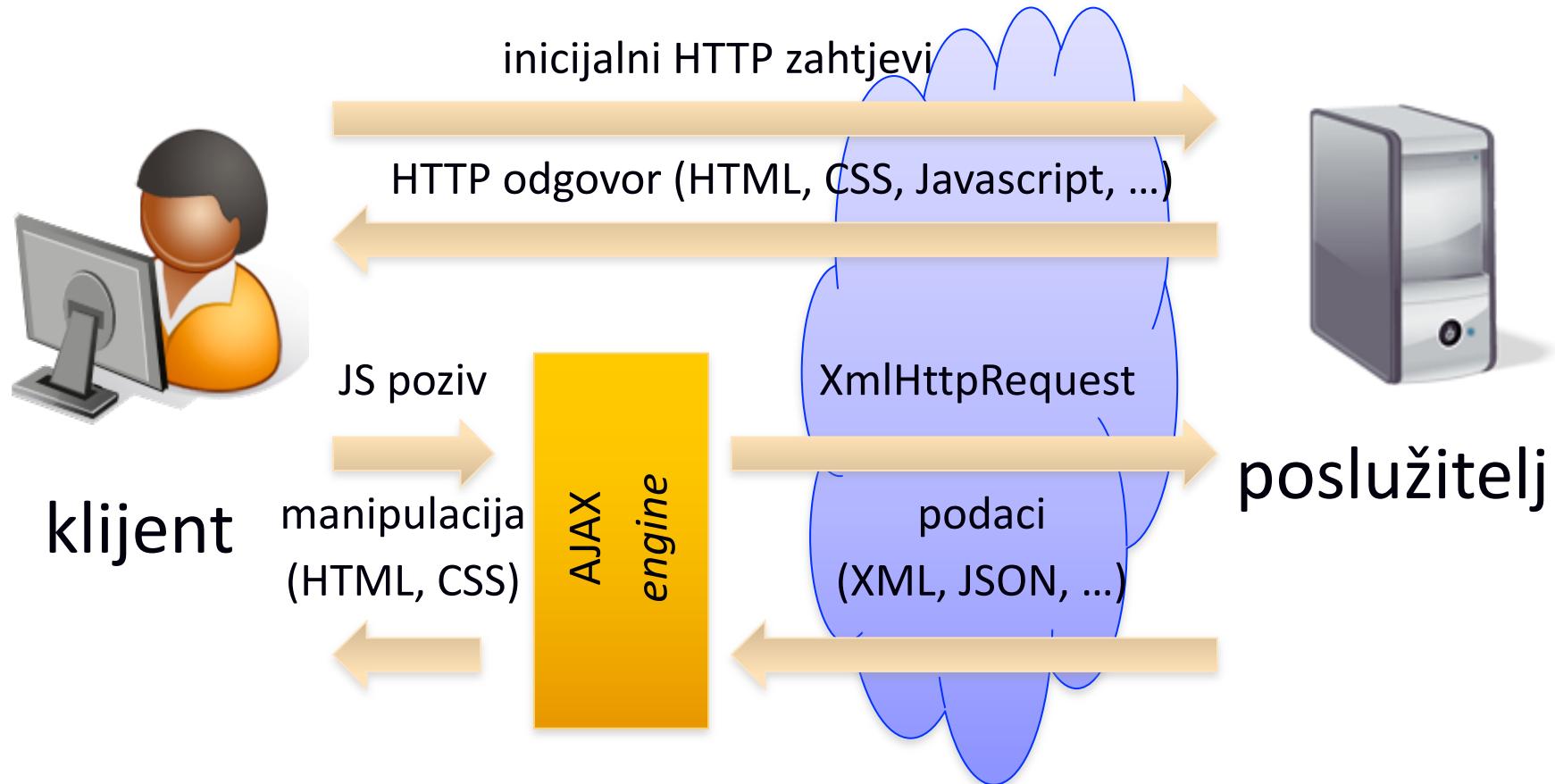
- inicijalno pokretanje
  - HTML, CSS, JavaScript, ...
- za vrijeme korištenja aplikacije
  - podaci putuju u obliku:
    - JSON - JavaScript Object Notation ili
    - XML
  - koristi se AJAX - [Asynchronous JavaScript](#) and XML



# 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/>)
  - jQuery (<http://jquery.com/>)
  - Preact (<https://preactjs.com>)
- Korisničko sučelje (*User Interface*):
  - Material-UI (<https://material-ui.com>)
  - Ant Design (<https://ant.design>)
  - Code Mirror (<http://codemirror.net>)
- Općenito:
  - Angular (<https://angular.io>)
  - React Native (<http://facebook.github.io/react-native/>)
  - Material-UI (<https://material-ui.com>)
  - 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

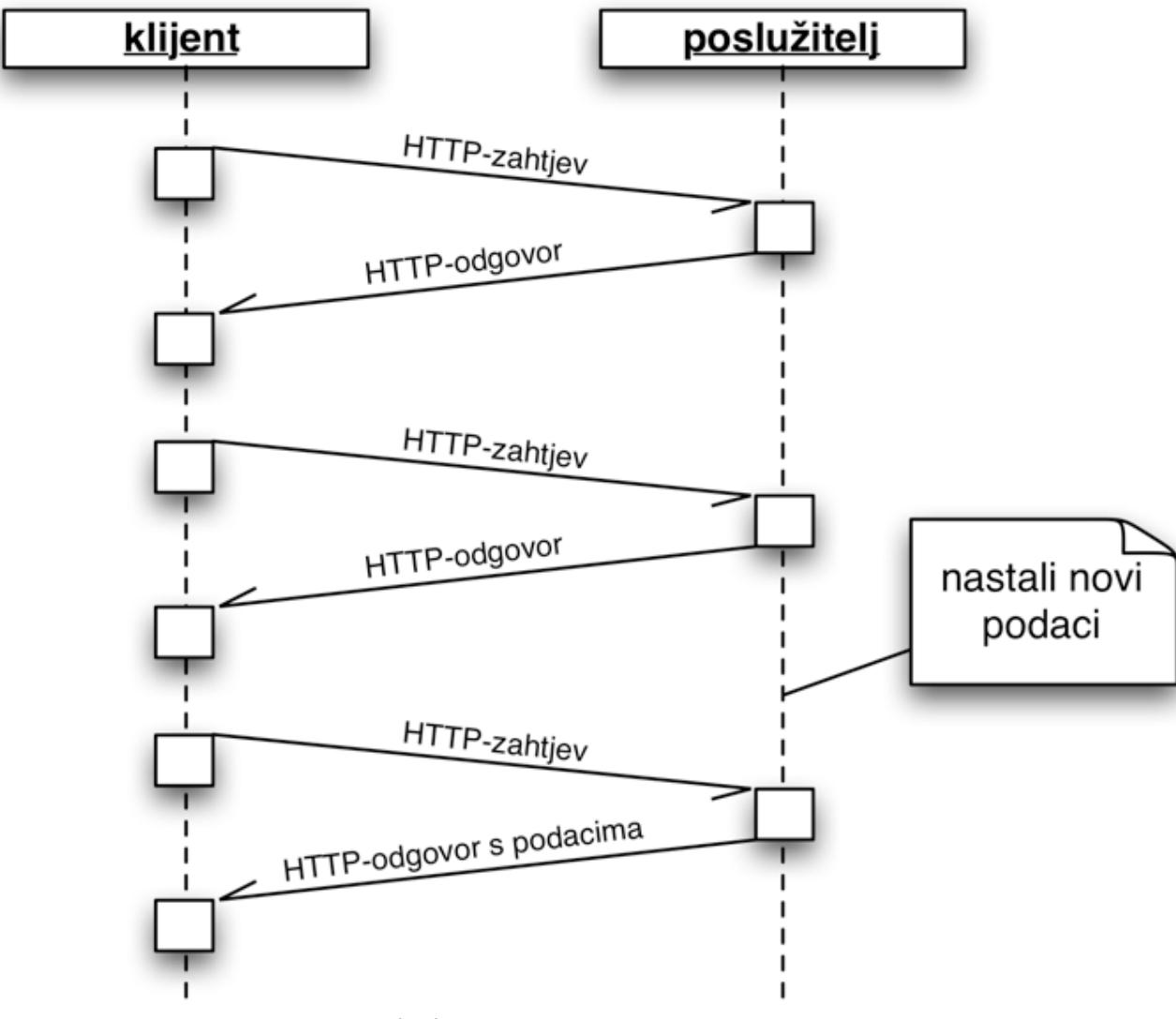
- čitanje informacija
- klijent-poslužitelj
- HTML
- portali
- parsiranje informacija
- posjedovanje
- modemska veza i HW
- direktoriji
- tipična tvrtka: Netscape

## 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, ...

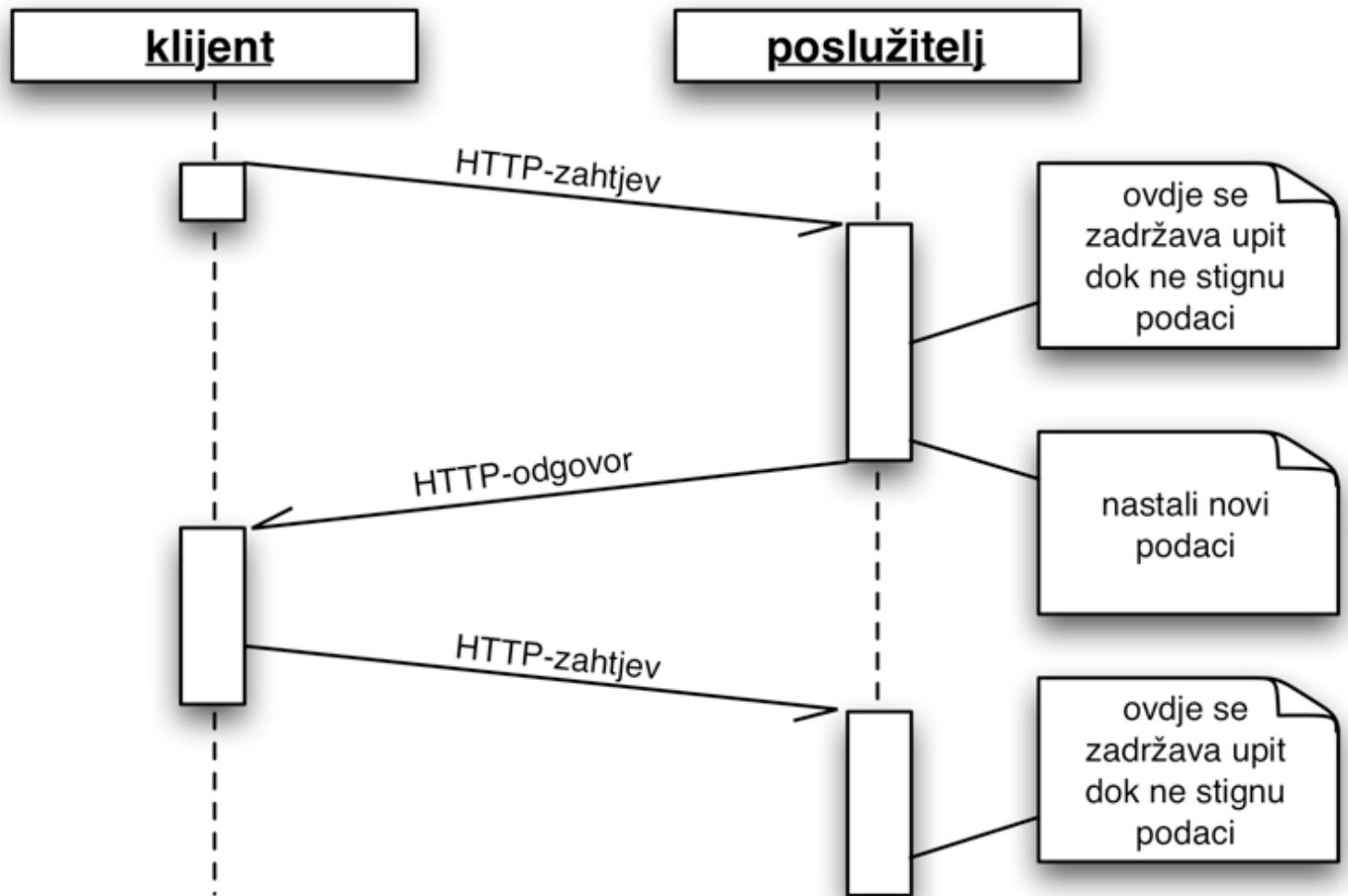
# Slanje događaja klijentu iz preglednika (1) - prozivanje

- 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

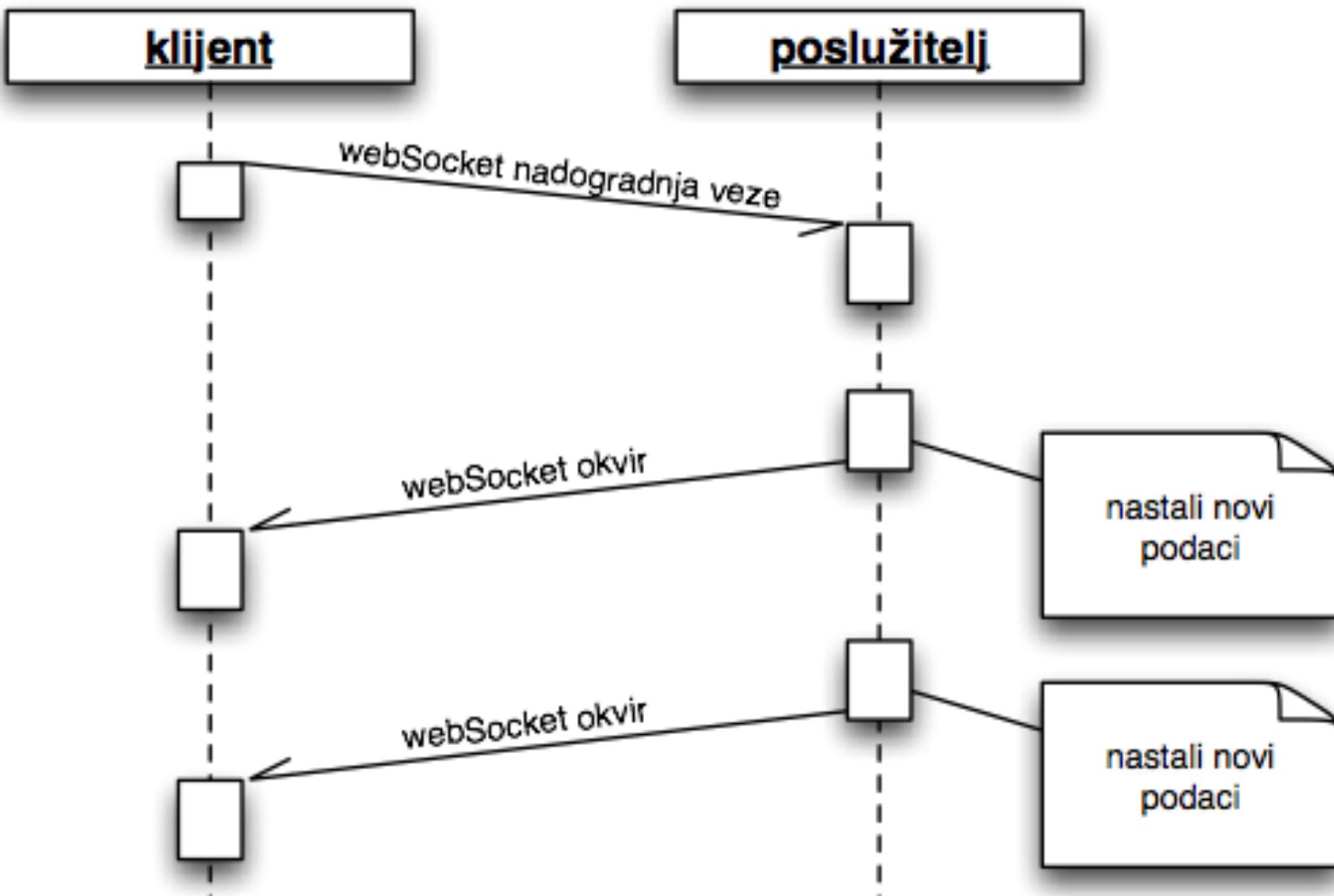
data: Druga poruka  
data: 2. red 2. poruke

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

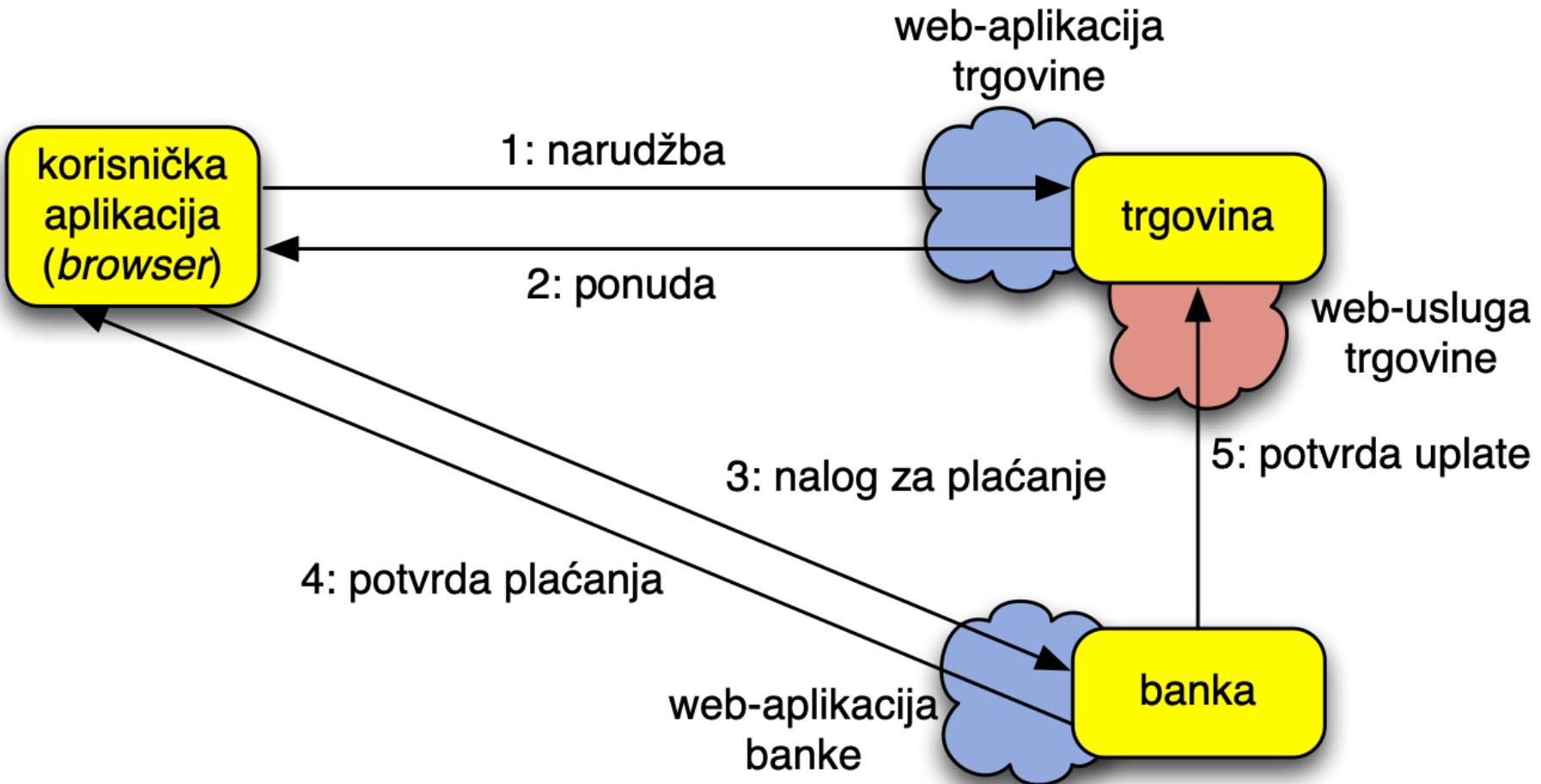


# Web Socket - sadržaj

- 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](http://www.w3.org))

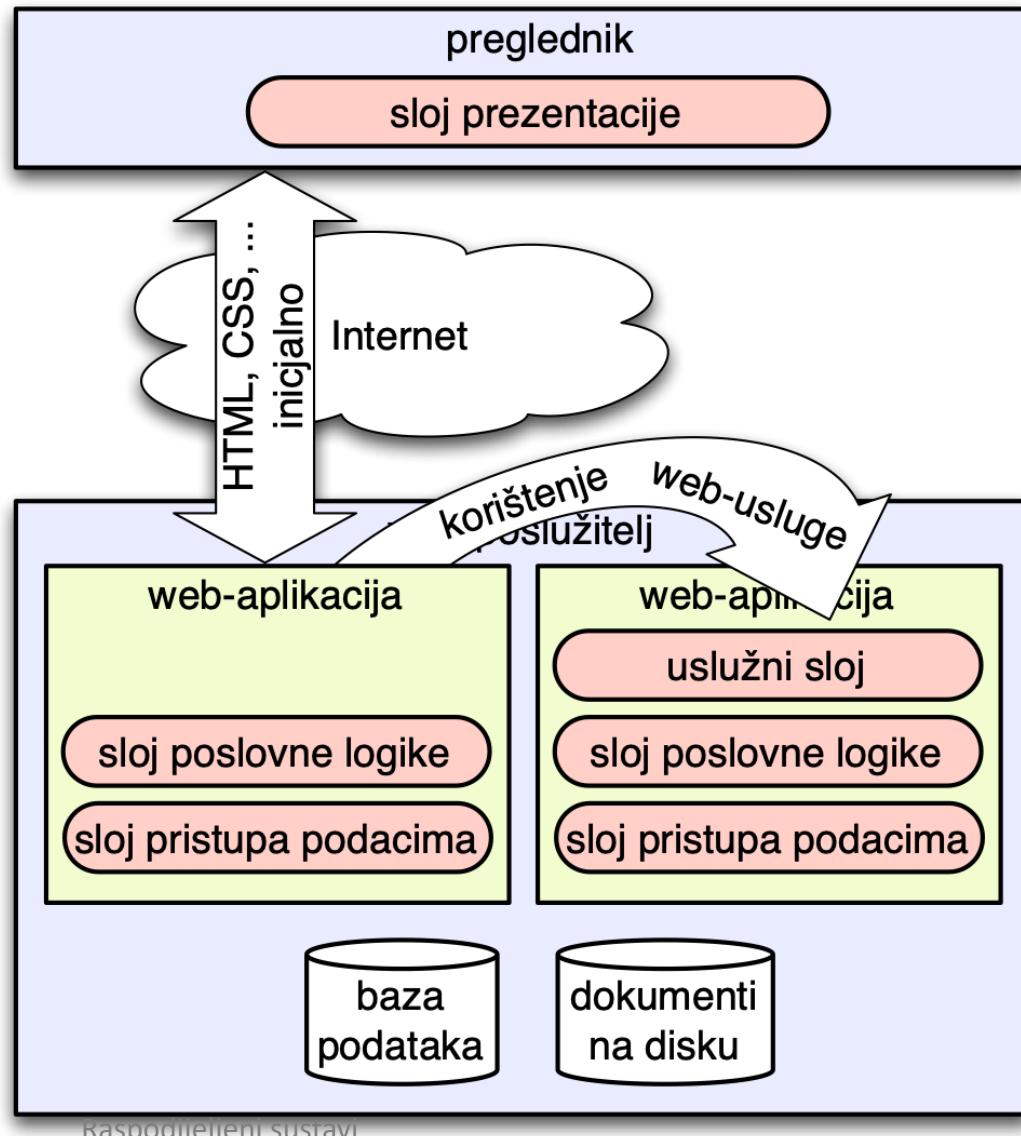
«Tehnologija web-usluga predstavlja novu vrstu web-aplikacija. To su samostalne, samoopisujuće aplikacije građene od modula, a koje se mogu objaviti, otkriti i pobuditi putem Weba. Web-usluge obavljaju funkcije koje mogu biti bilo što, od jednostavnih zahtjeva do komplikiranih poslovnih transakcija...»

(IBM's tutorial, [www.xml.com](http://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

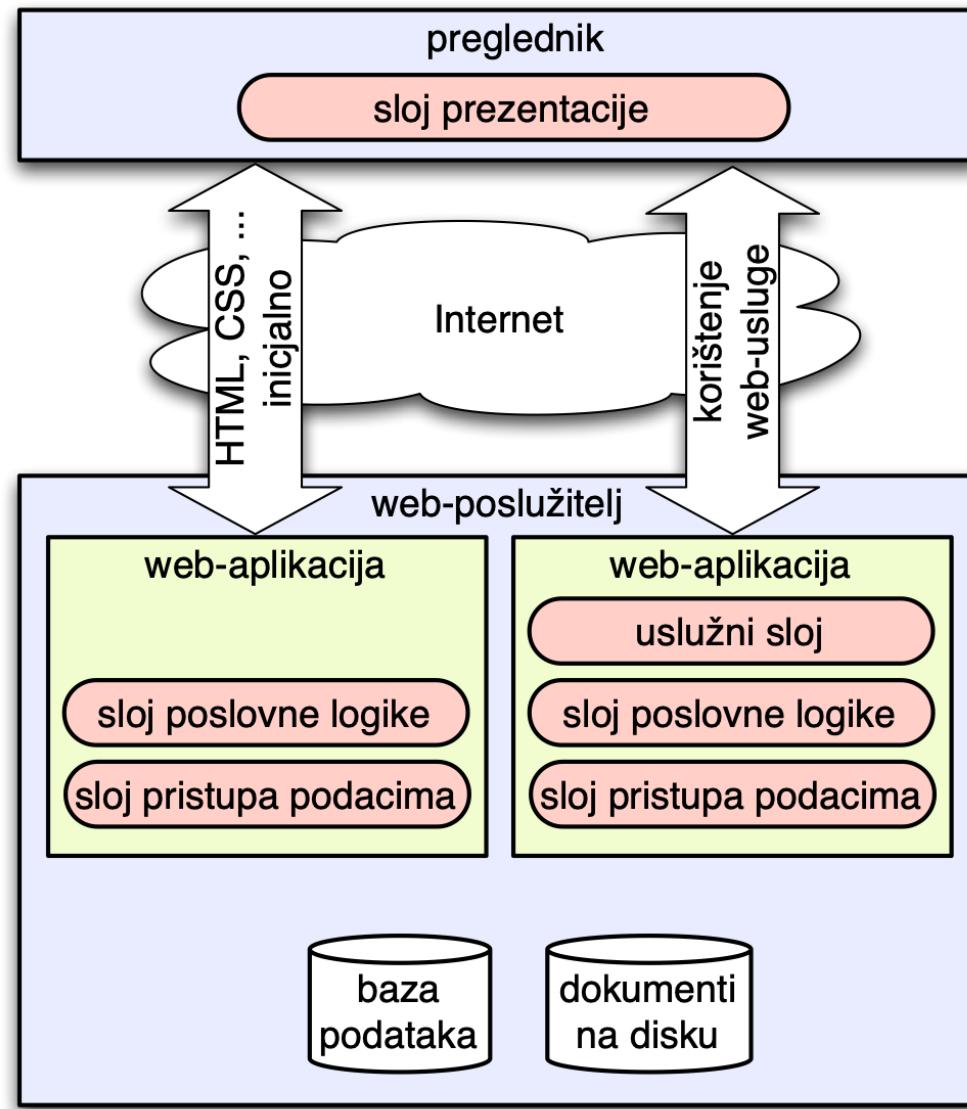
# Web-aplikacija koja koristi web-usluge (1)

- 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

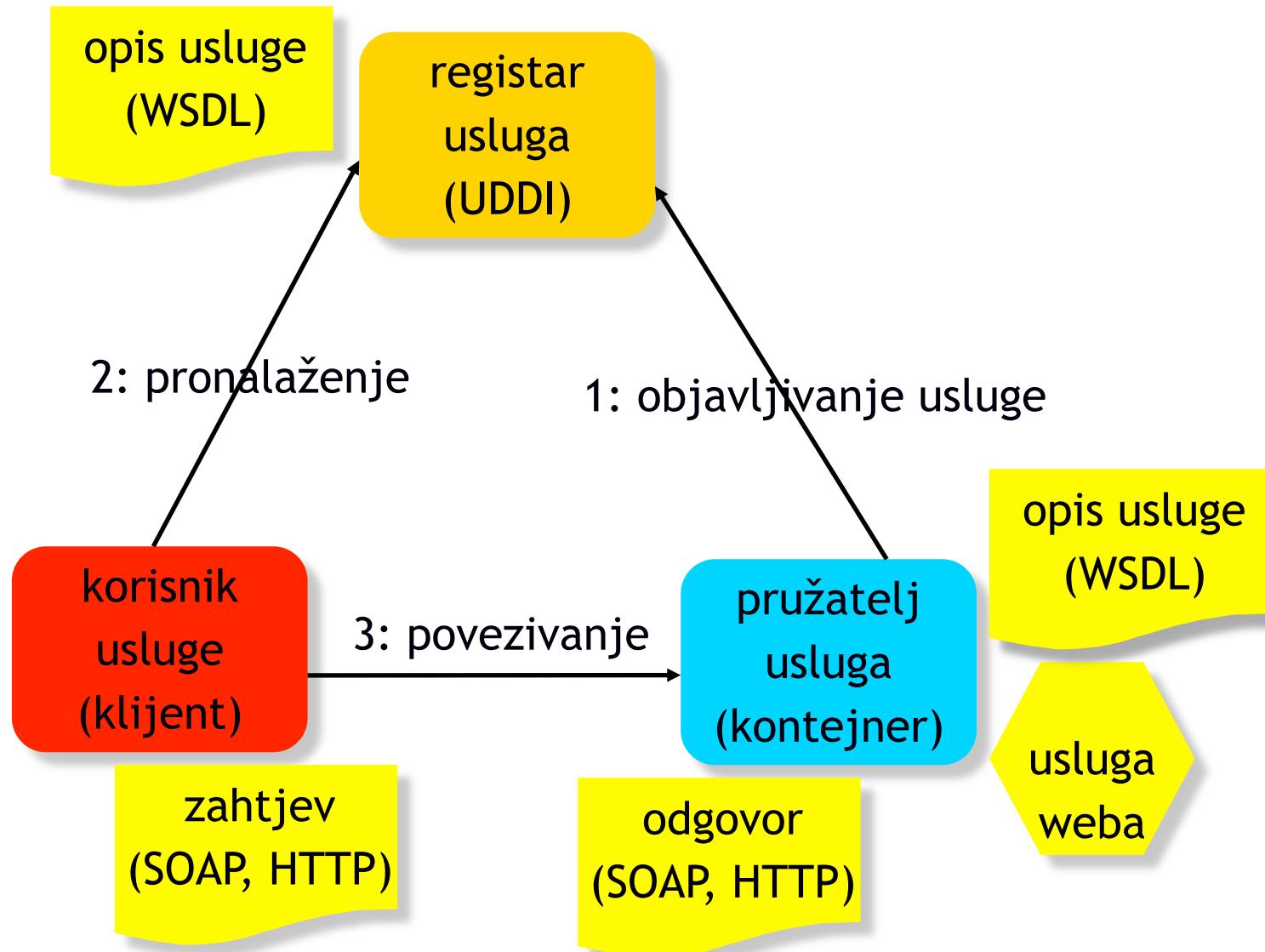
## Tehnologije web-usluga

- 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

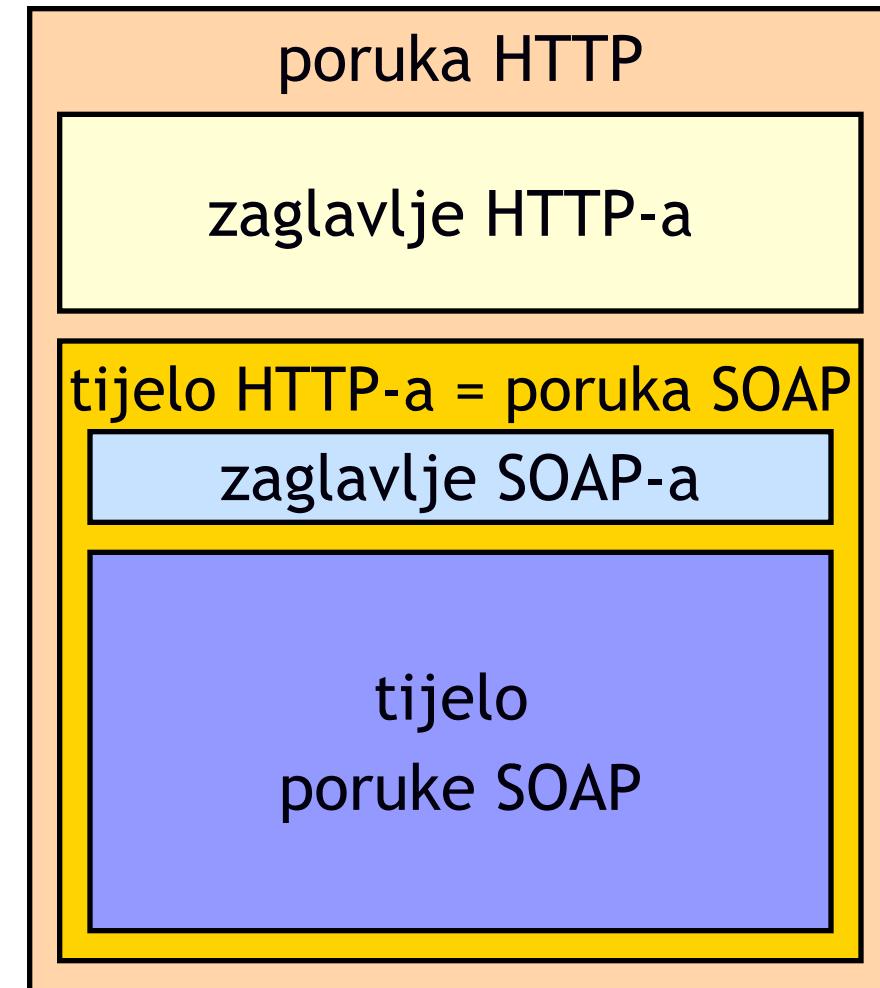
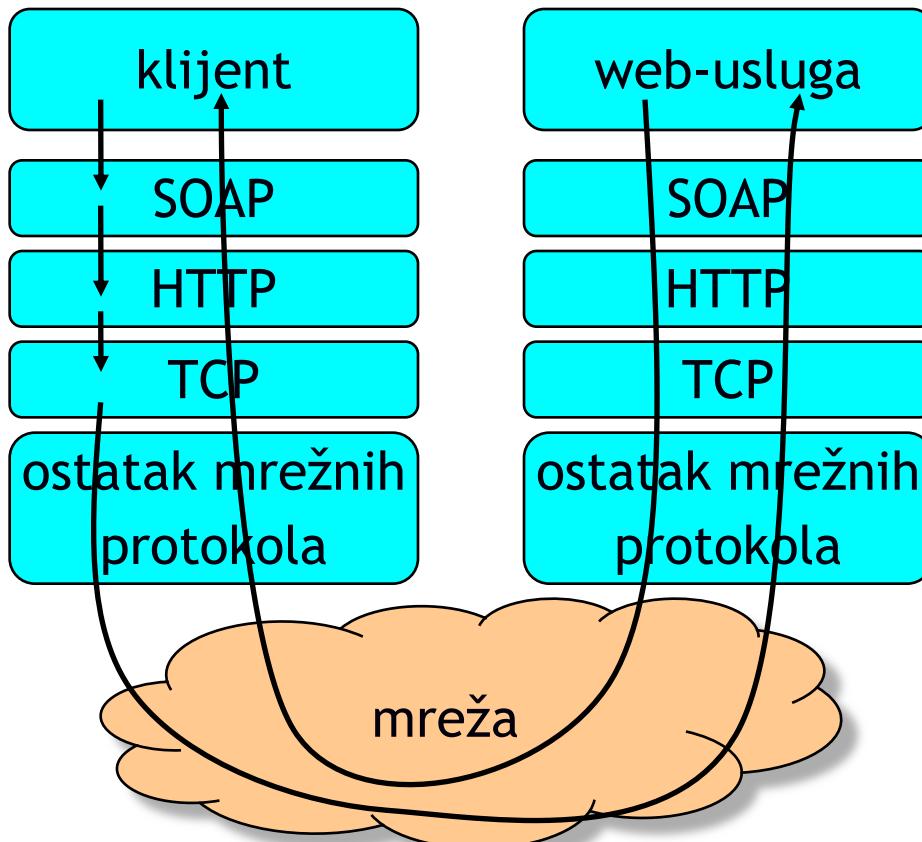
# Arhitektura i korištenje



# 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/>

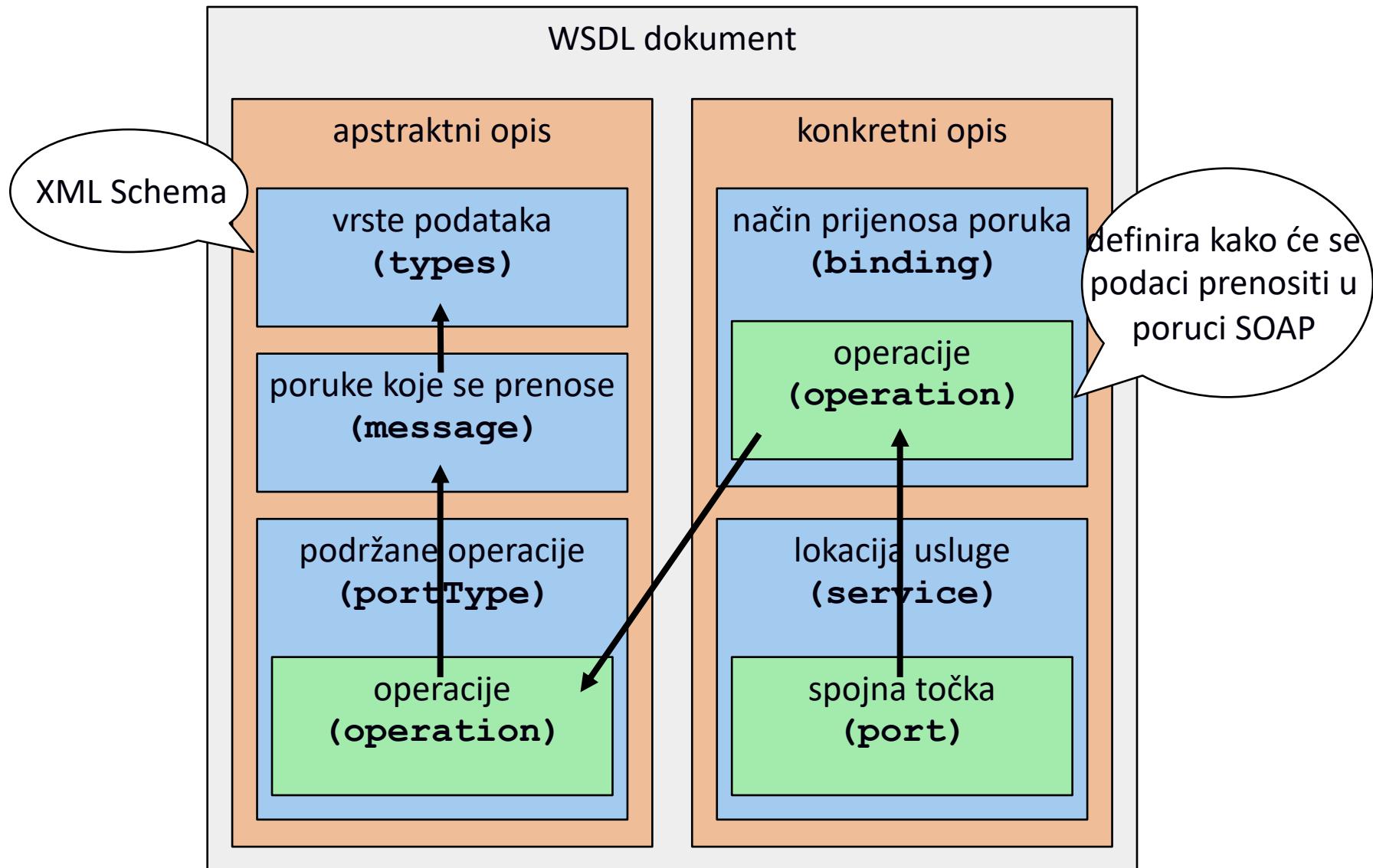
# Prijenos poruke SOAP



# 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, indempotentna, može se privremeno spremiti
  - PUT - idempotentna
  - DELETE - idempotentna
  - HEAD - sigurna, indempotentna
  - POST - ništa
  - PATCH - idempotentna

# 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 /person/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 /person/100
- odgovor:  
404 Not Found

# Primjer upita i odgovora - promjena resursa

- PUT /person/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

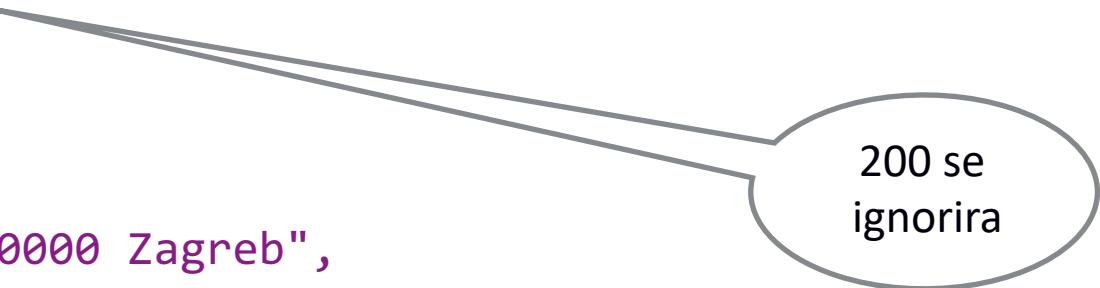
- PUT /person/200

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

- odgovor:

201 Created

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



200 se ignorira

# Primjer upita i odgovora - promjena dijela resursa

- PATCH /person/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"  
}
```

# Primjer upita i odgovora - promjena nepostojećeg resursa

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

- odgovor:

404 Not Found

# Primjer upita i odgovora - osoba koja postoji

- DELETE /person/3
- odgovor:

204 No Content

# Primjer upita i odgovora - osoba koja ne postoji

- DELETE /person/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

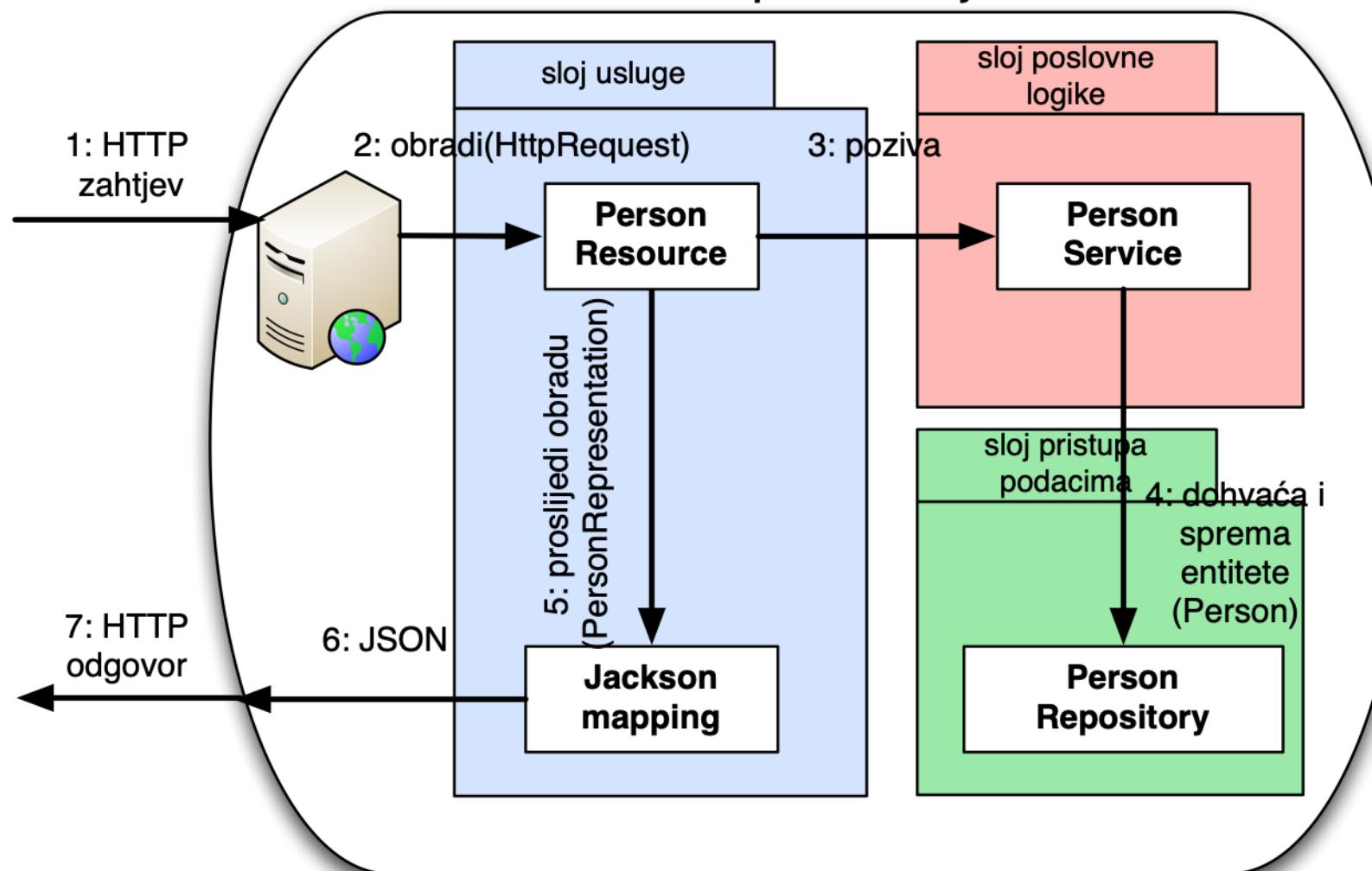
- GET /bills/1
- odgovor:

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 2.3.4)

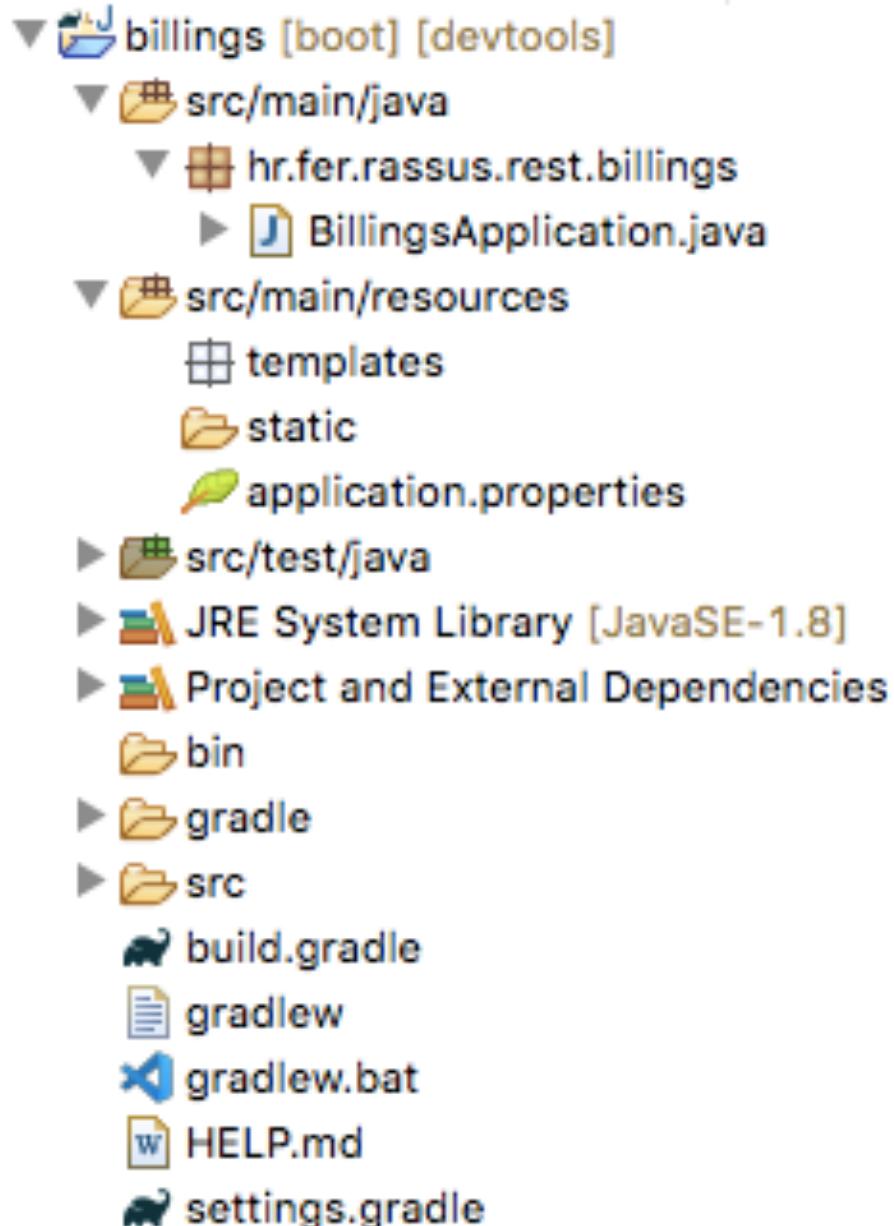
- jedan od podprojekata Springa
  - <http://projects.spring.io/spring-boot/>
- pojednostavljuje korištenje Springa, pogotovo stvaranje novog projekta
- podržava:
  - automatsku konfiguraciju
  - pretraživanja klasa na putu (*path*)
- aplikacija ima manje koda
- kod web-aplikacija - omogućuje izradu samostalnih 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: 11 (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

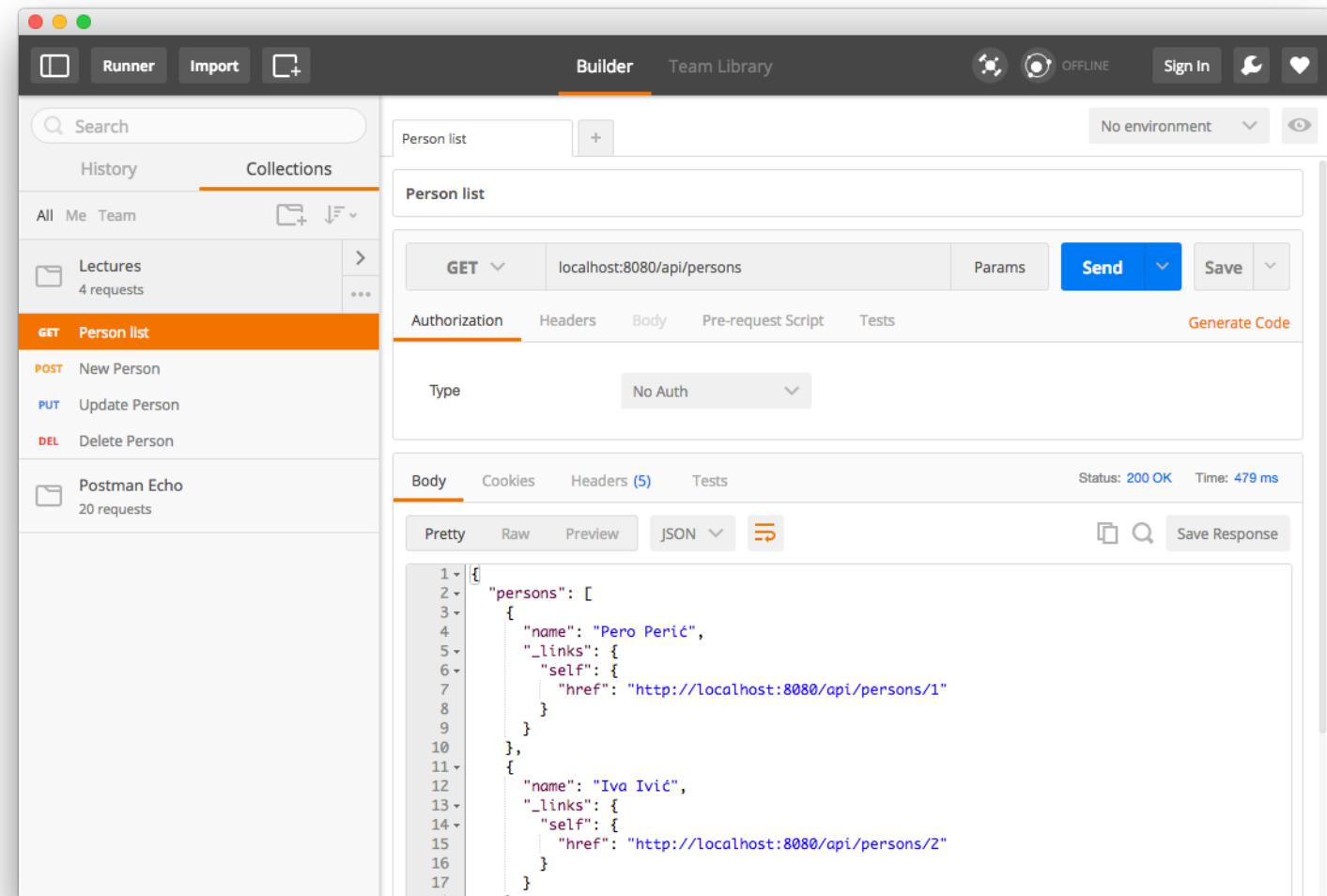
- dva načina:
  - u konfiguracijskoj klasi (npr. SimpleApplication) definirati metodu koja je označena da vraća bean (@Bean)
  - označiti klasu s @Component
    - mehanizam pretraživanja puta može pronaći takvu klasu
    - podvrste:
      - **@Controller** - predstavlja kontroler u Web MVC-u
      - **@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 metoda (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) - kolovoz, 2015., IETF, v02
  - [draft-amundsen-richardson-foster-alps-02](#)
  - format podataka za opis aplikacije i njihove semantike
- [OpenAPI inicijativa](#) (konzorcij)
  - osnovana 2016. u sklopu Linux Fundation
  - OpenAPI specification (bivši Swagger 2.0 spec.) - aktualna verzija [3.0.3](#)



# 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



Fakultet  
elektrotehnike i  
računarstva

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**  
Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

# Raspodijeljeni sustavi

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

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

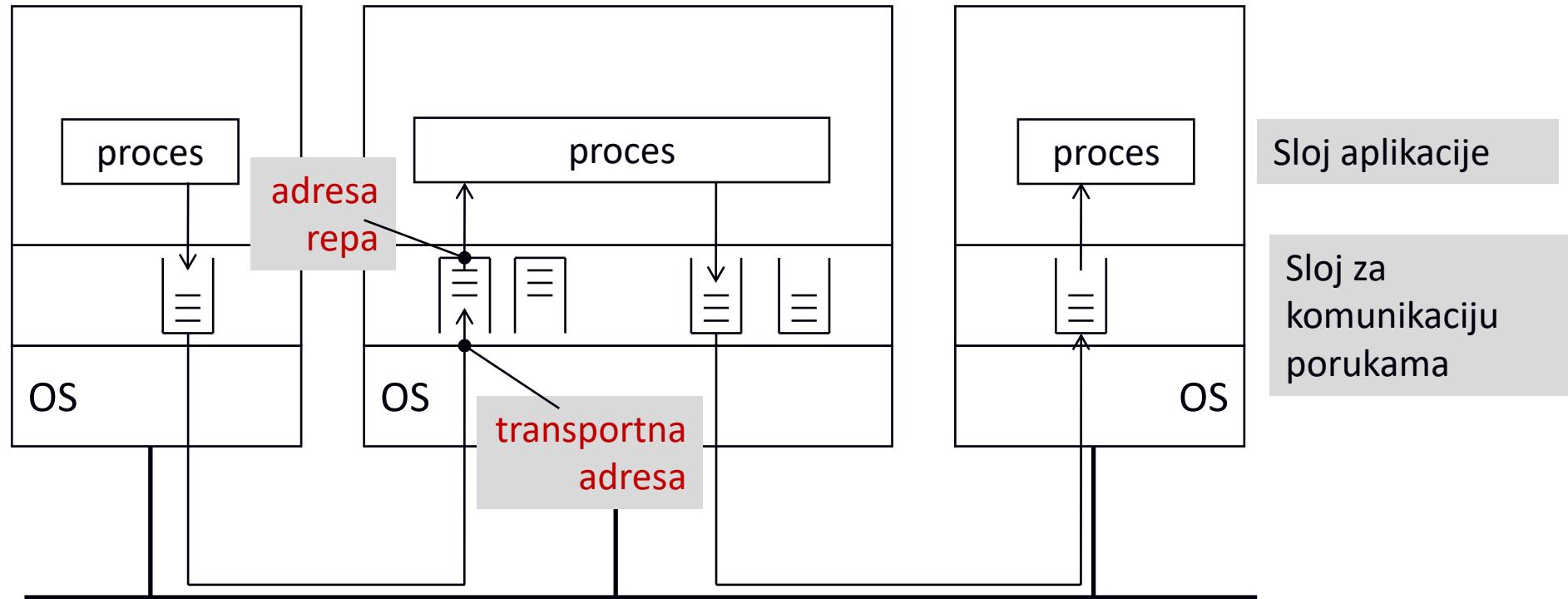
- Procesi/objekti komuniciraju razmjenjujući poruke.
- U komunikaciji sudjeluju izvor (pošiljatelj poruke) i odredište (primatelj poruke).
- Izvor šalje poruku, poruka se pohranjuje u rep koji je pridijeljen odredištu.
- Odredište čita poruku iz repa.
- Poruke sadrže podatke, važna je adresa odredišnog 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 postoje 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 spremi 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 moguća i na načelu *push*

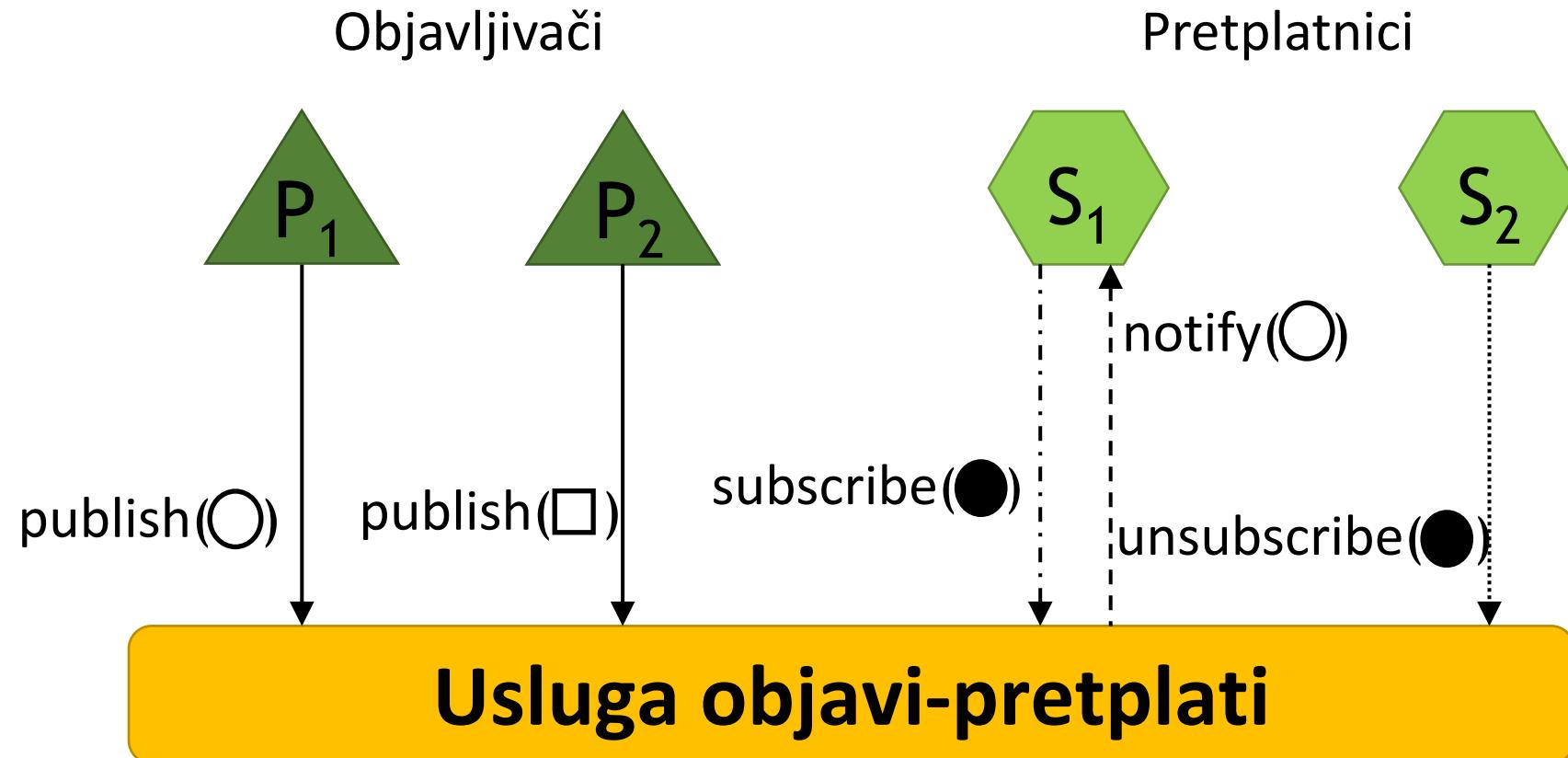


- **notify** – aktivna isporuka poruke iz repa primatelju po primitku poruke (na strani primateljskog procesa nužan je *listener thread*)

# Sadržaj predavanja

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

# Interakcija objavi-preplati



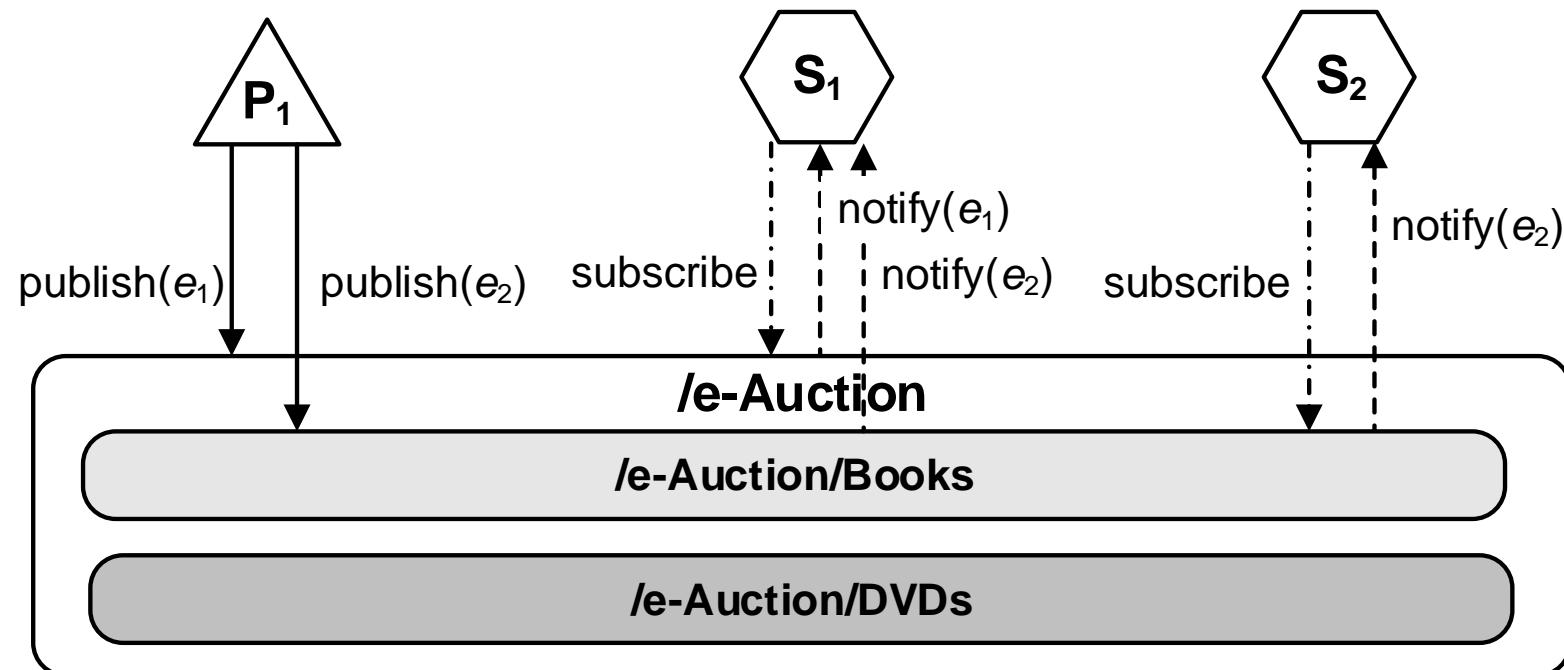
# Osnovni pojmovi

- objavljavači (*publishers*)
  - definiraju obavijesti (*notifications*)
- pretplatnici (*subscribers*)
  - preplatama (*subscriptions*) i odjavama preplata (*unsubscriptions*) izražavaju namjeru primanja određenog skupa obavijesti
- usluga objavi-preplati:
  - sustav za obradu događaja (*event service – ES*)
  - obrađuje i pohranjuje primljene obavijesti/preplate/odjave preplata
  - isporučuje obavijesti pretplatnicima prema njihovim aktivnim preplatama
  - 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



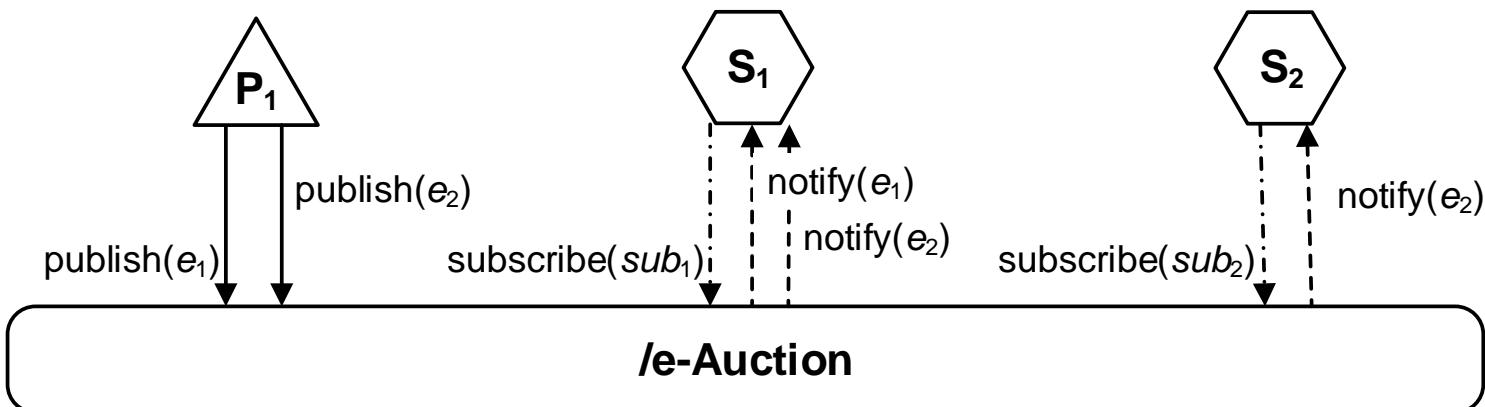
# Pretplata na sadržaj

```
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)
```

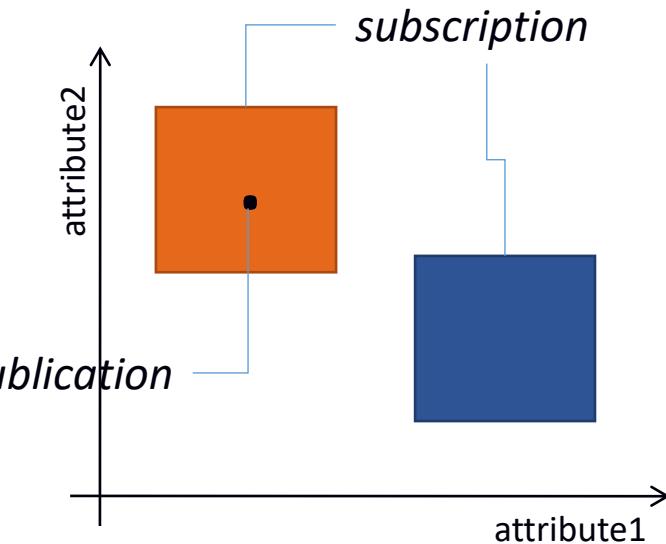
```
sub1 = (category == "books"  
    & price < 20 EUR)
```

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



# Primjer obavijesti/preplate (strukturirani podaci)

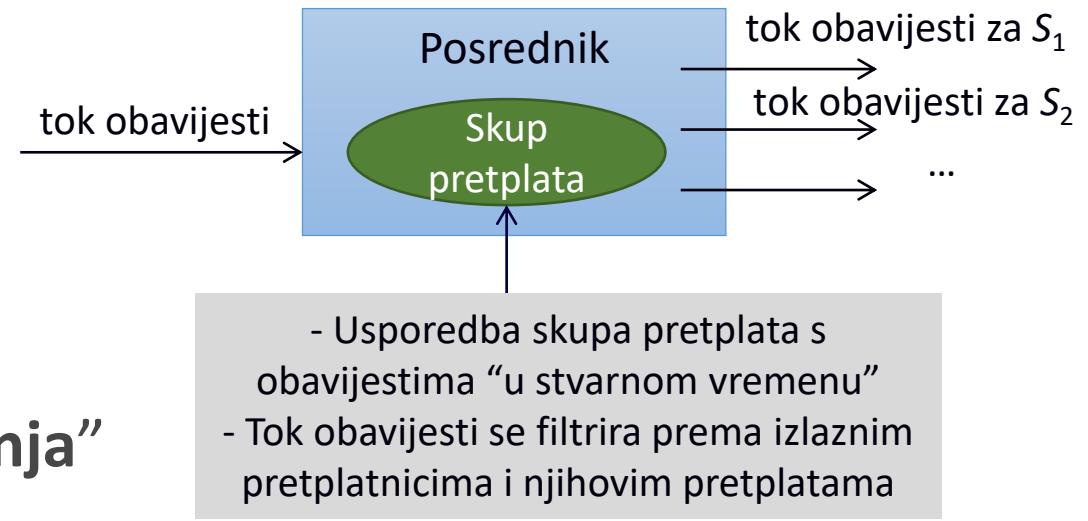
- **obavijest** je najčešće točka u višedimenzionalnom prostoru
  - npr. očitanje senzora, cijena dionice, oglas, vijest
  - objavljivači kontinuirano objavljaju nove obavijesti (često ograničene valjanosti)
- **preplata** je potprostor višedimenzionalnog prostora
  - definira se kao Booleova funkcija 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



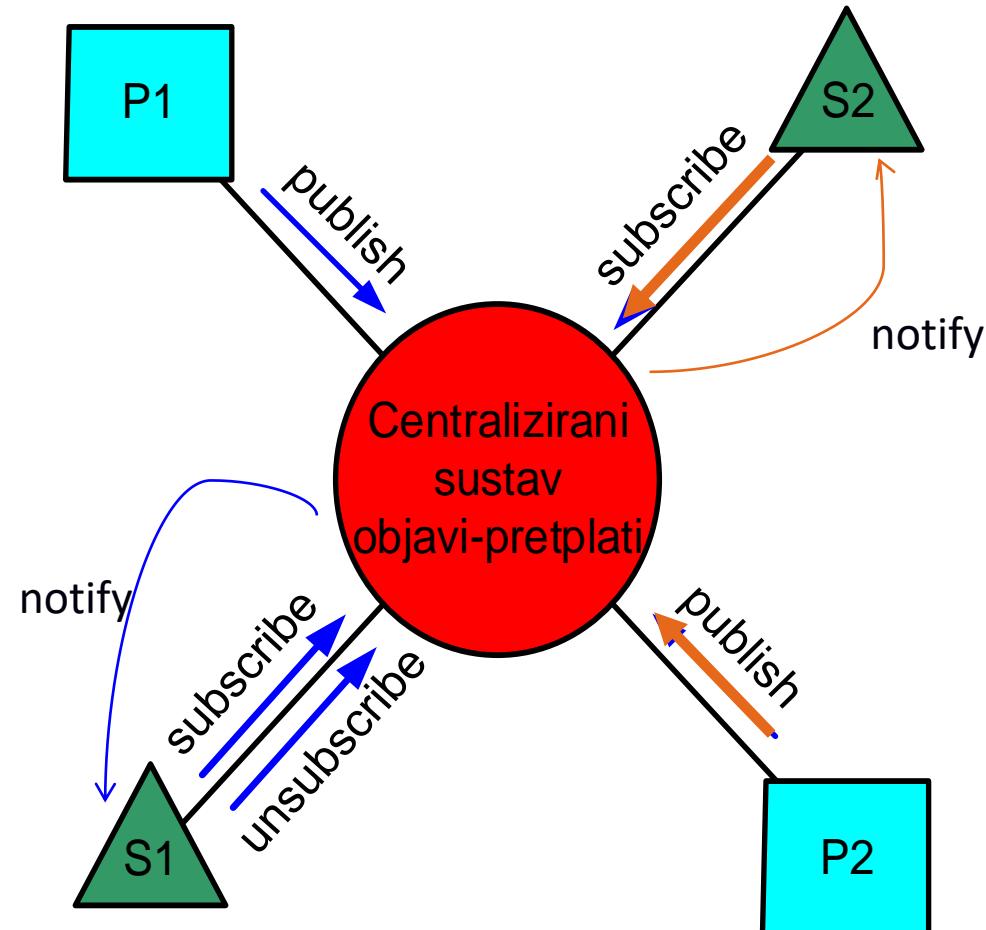
- Usporedba ispituje svojstvo **“prekrivanja”** obavijesti pretplatom

- 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)

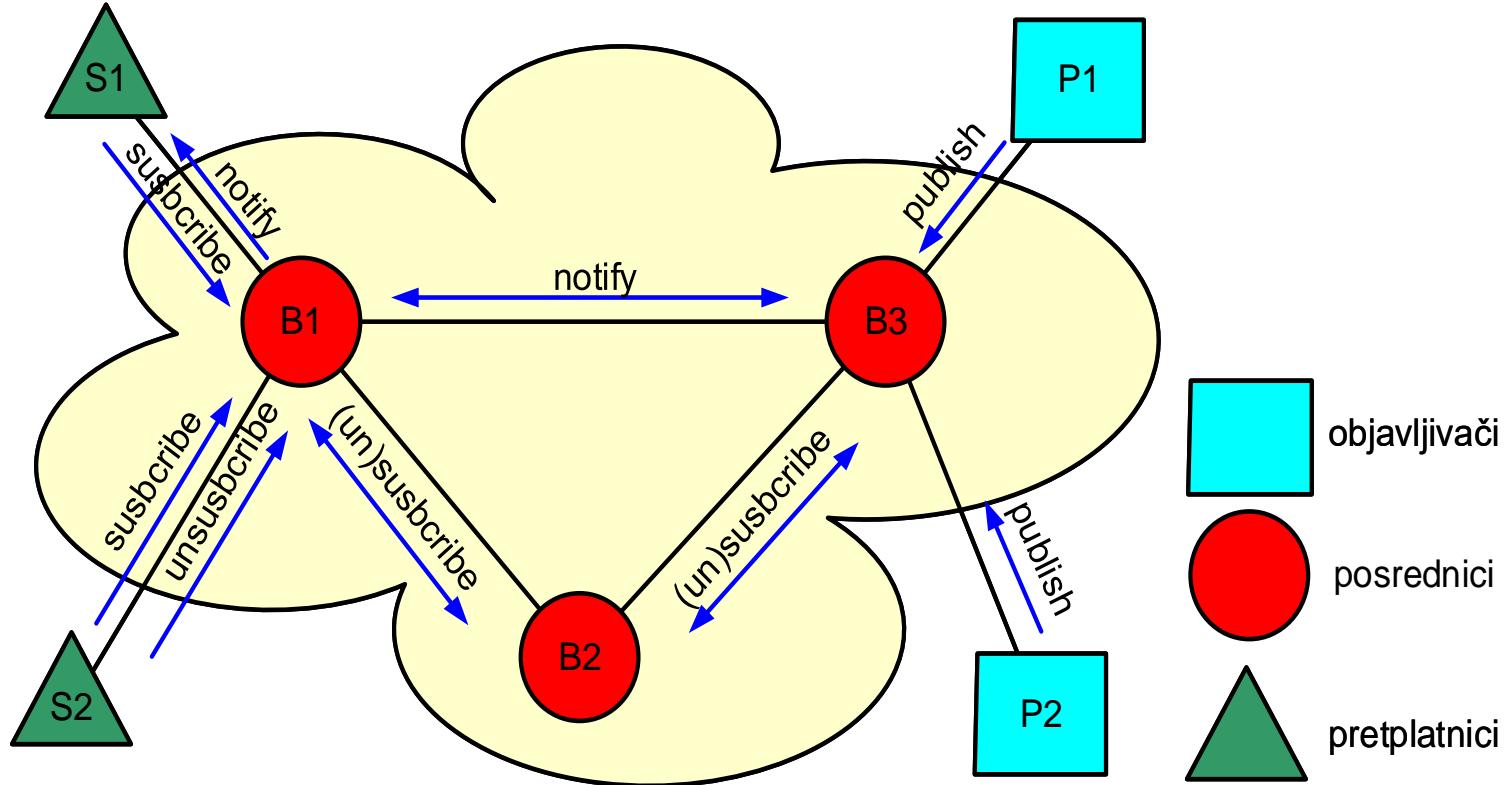
# Centralizirana arhitektura



# 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

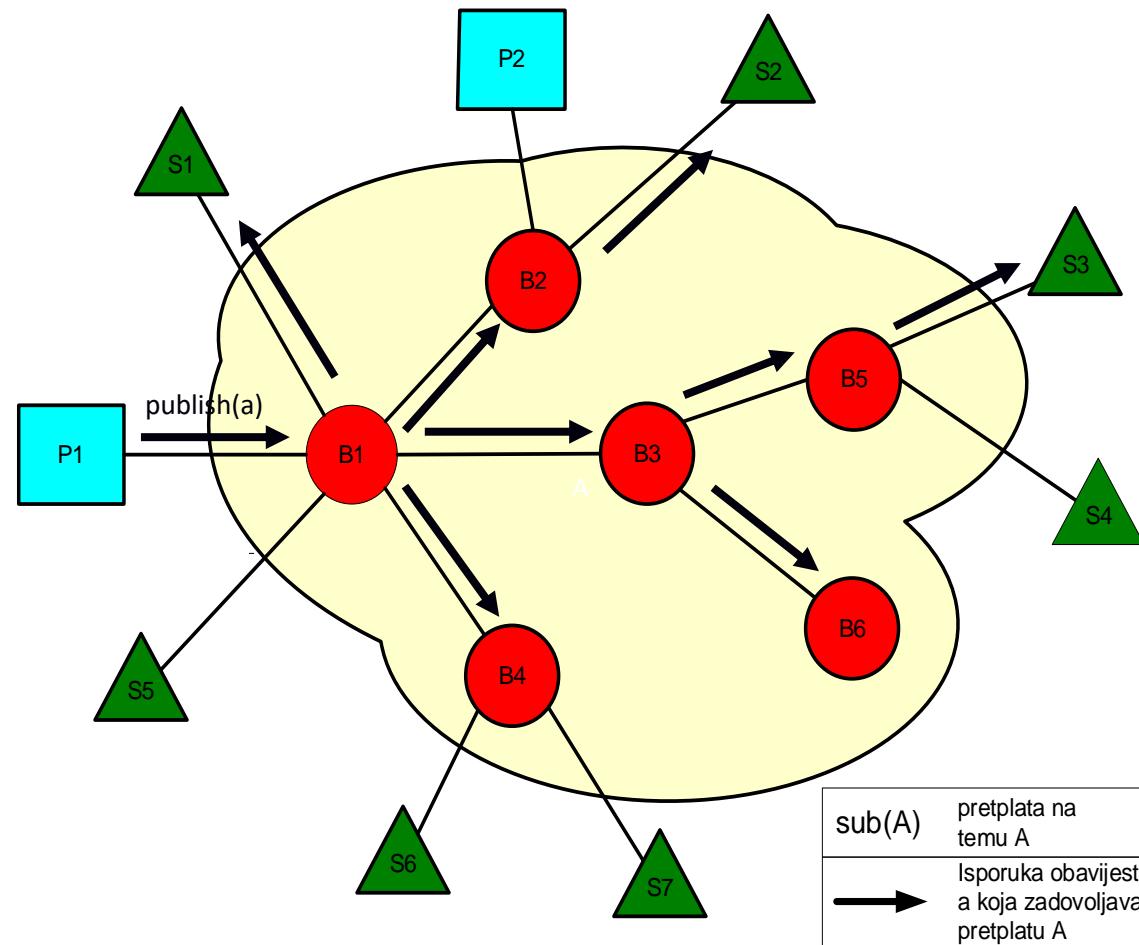
# Raspodijeljena arhitektura



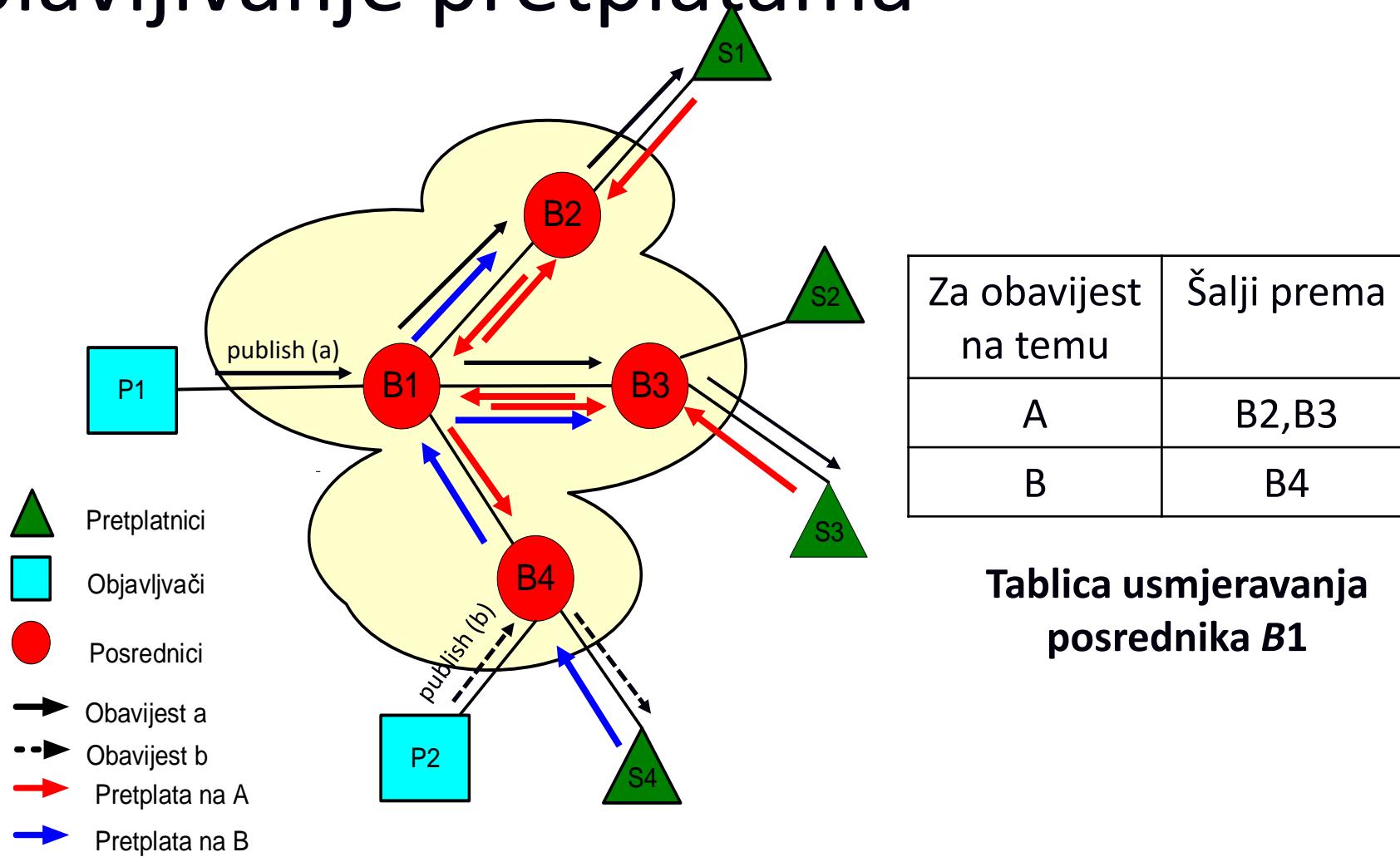
# Osnovna načela usmjeravanja

- 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.

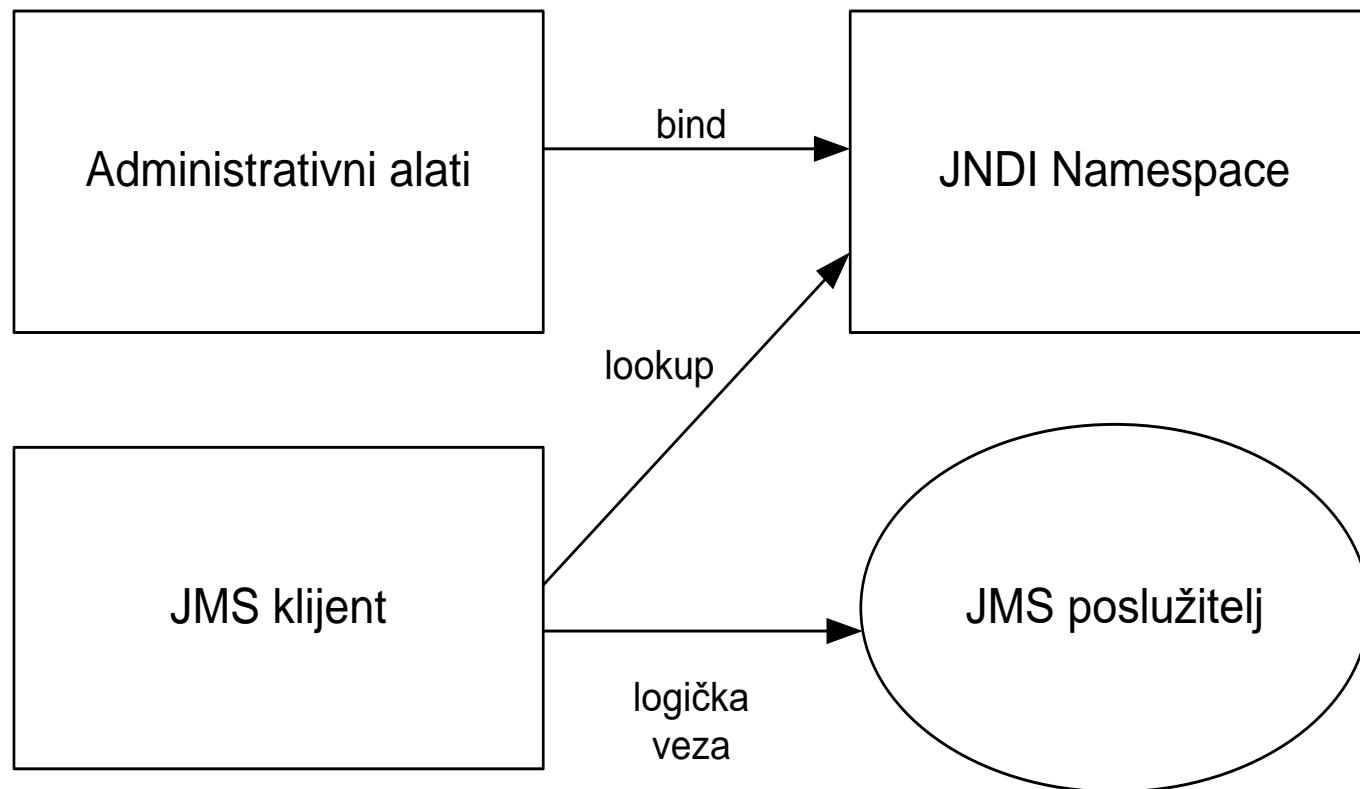
Popularne implementacije: Apache ActiveMQ, IBM WebSphereMQ, HornetQ, OpenJMS

# 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)

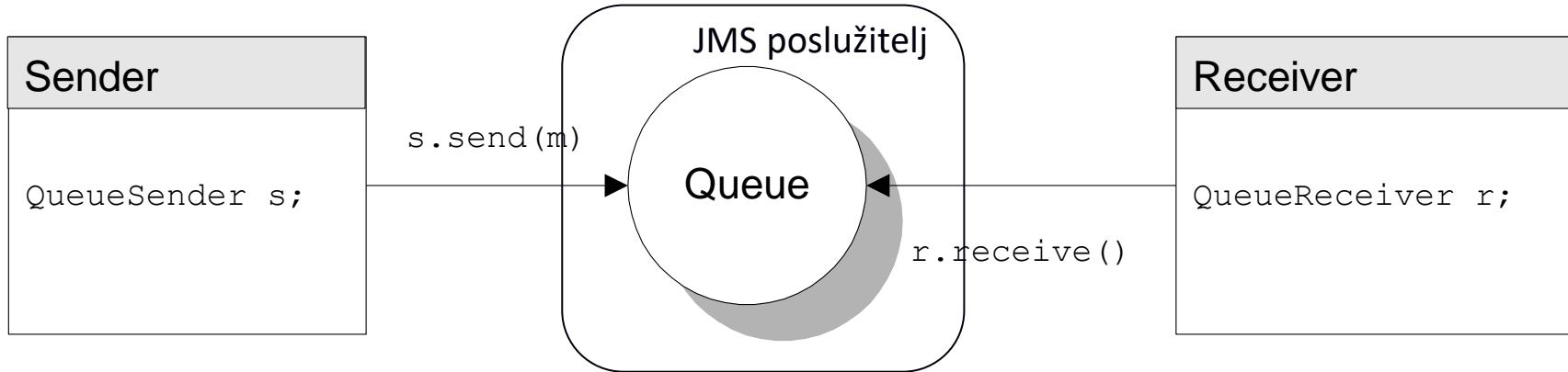


# Modeli JMS-a

JMS implementira sljedeće modele za komunikaciju porukama i obavijestima

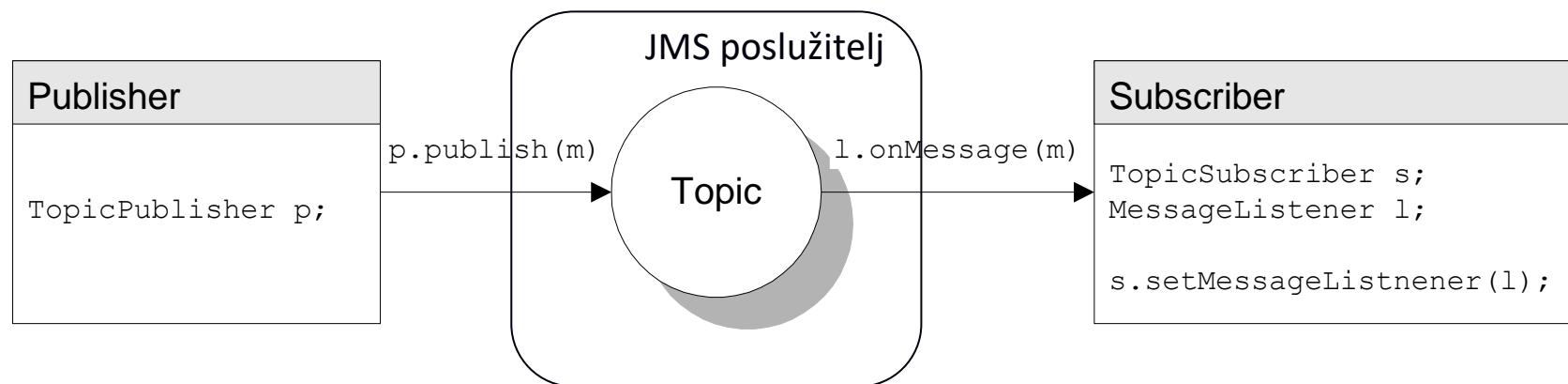
- *Point-to-point*
  - komunikacija porukama, jedna poruka za jedno odredište
- *Publish/subscribe*
  - objavi-preplati, jedna poruka za skup zainteresiranih pretplatnika

# Point-to-point



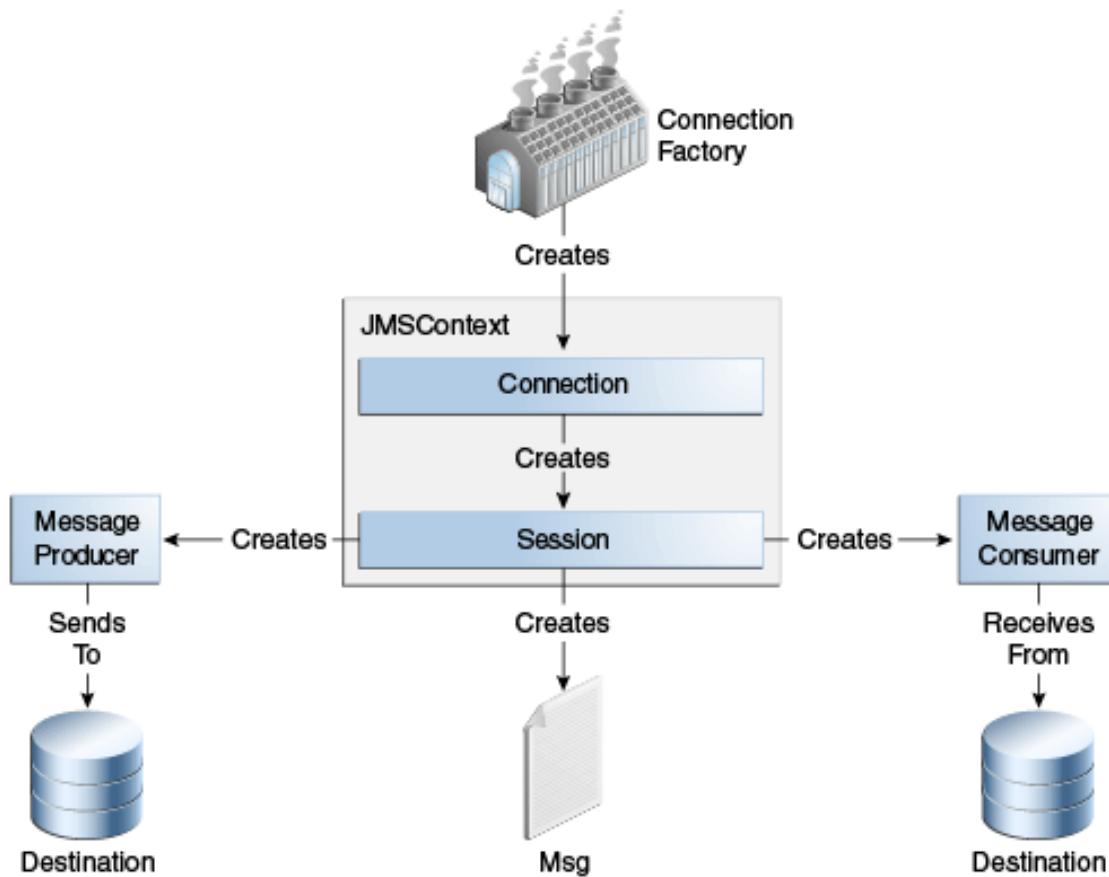
1. Klijent s koji šalje poruku m poziva `s.send(m)`. Poruka se sprema u repu.
2. Klijent koji prihvata poruku mora provjeriti postoji li poruka u repu . Poziva `r.receive()`.
3. Poruka se briše iz repa i šalje klijentu.

# 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 sesije (*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

Version 1.0, OASIS Standard, 29.10.2012.

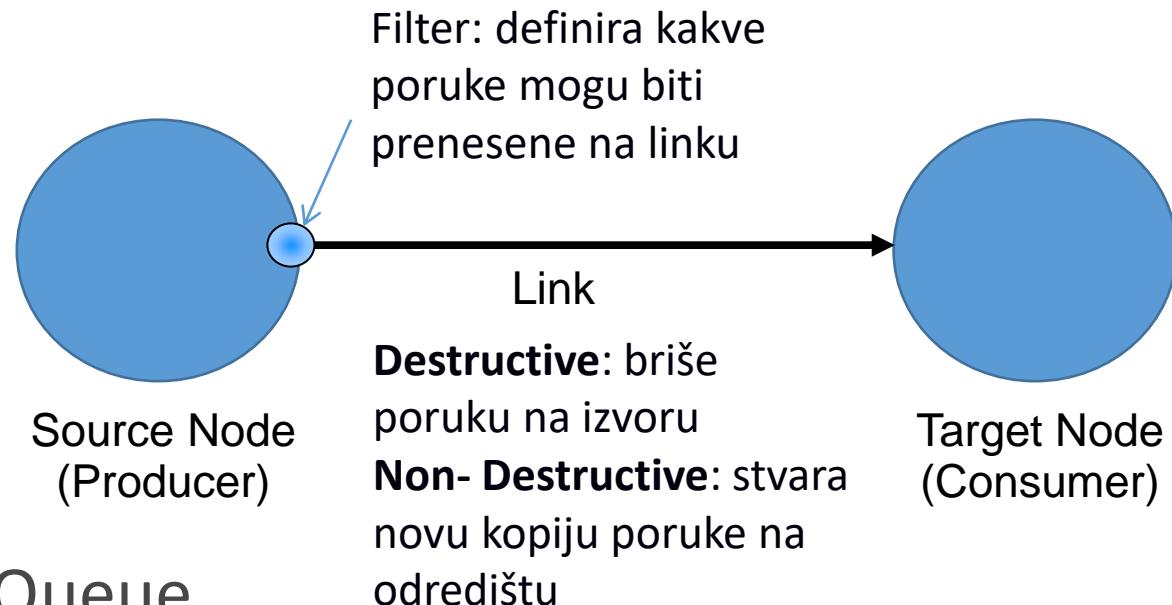
<http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html>

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

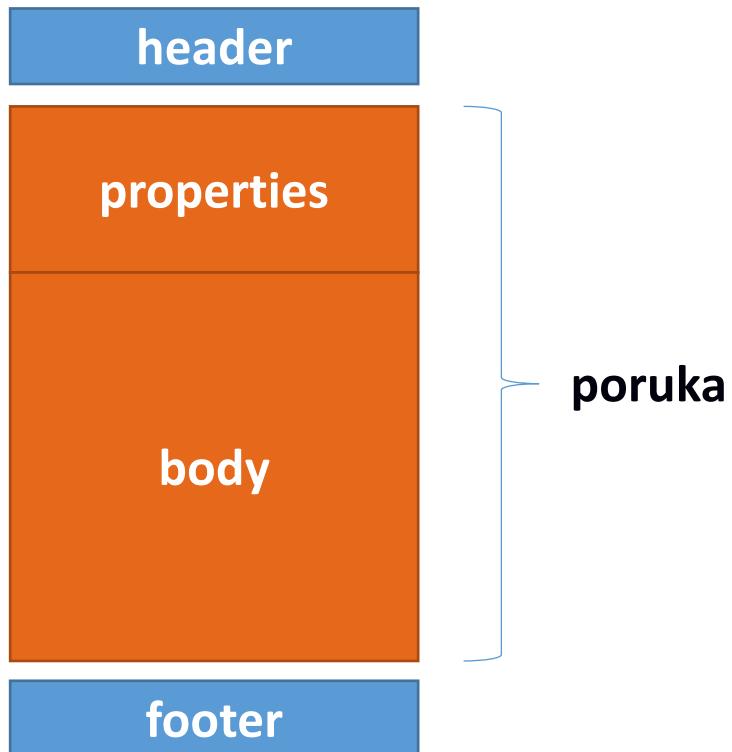
Popularan protokol i raširena primjena zbog implementacije RabbitMQ (**podržava AMQP 0.91**, v1.0 sadrži značajne razlike u odnosu na v0.91, potreban [plugin](#)), OpenAMQ, StortMQ, Apache QPid...

# Mreža AMQP

- *Node*: Producer, Consumer, Queue
- *Container*: Broker ili Client
- *Node* postoji unutar *Containera*,
- 1 *container* može sadržavati više čvorova



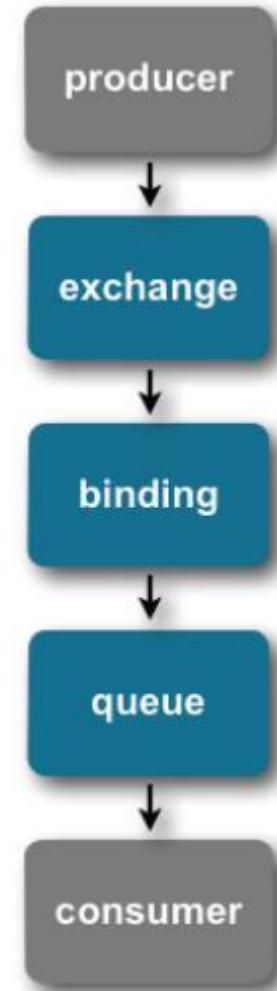
# AMQP poruke



- 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

# RabbitMQ: implementira AMQP v0.9

- *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*

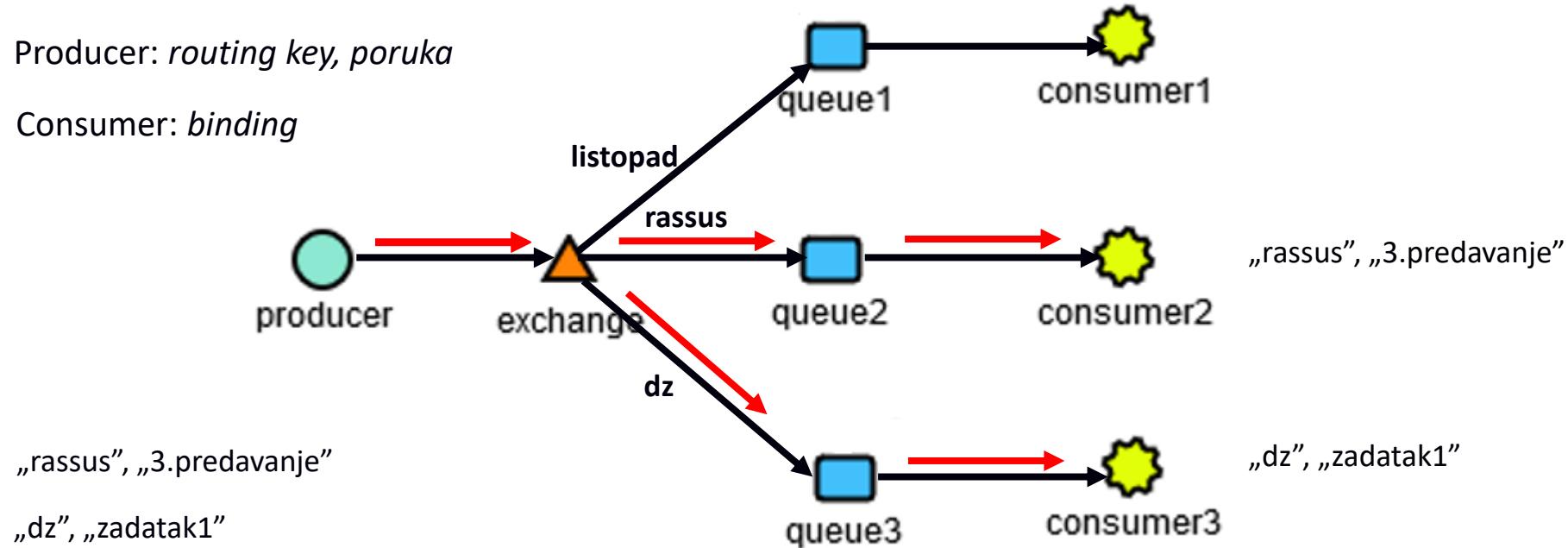


# RabbitMQ: različiti komunikacijski modeli

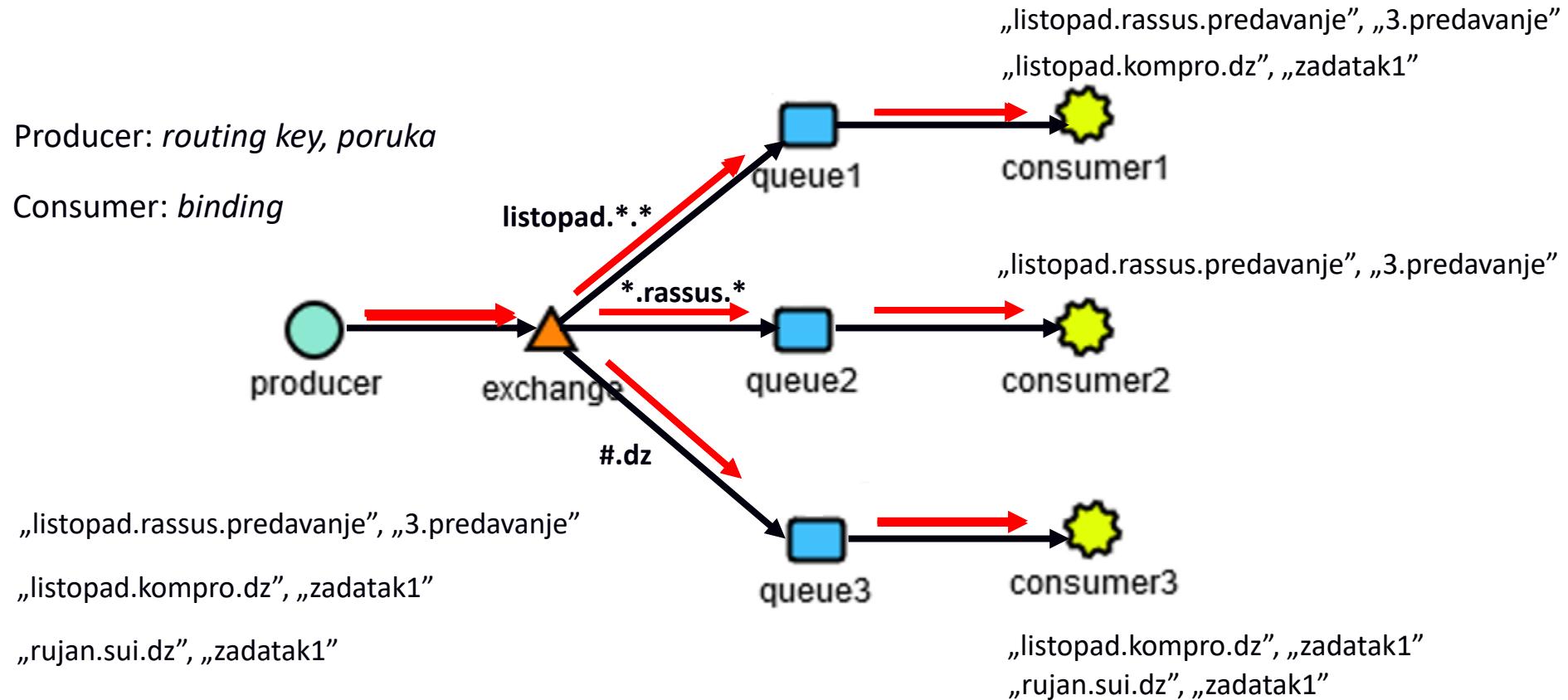
## *Exchange types*

- *Direct Exchange*: odgovara *point-to-point* modelu JMS-a, razlika je u tome što poruka može biti isporučena u više repova ako *binding* odgovara *routing key*-u
- *Fanout Exchange* i *Topic Exchange*: odgovara *publish/subscribe* modelu JMS-a
  - *Fanout* šalje sve poruke na sve repove spojene na exchange
  - *Topic* omogućuje filtriranje poruka i definiranje bindinga uporabom specijalnih znakova # i \*

# RabbitMQ: Direct Exchange



# RabbitMQ: Topic Exchange



# Literatura: AMQP

RabbitMQ Simulator, <http://tryrabbitmq.com/>

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

# 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 = "listopad.rassus.predavanje";
    String message = „3.predavanje”;
    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();

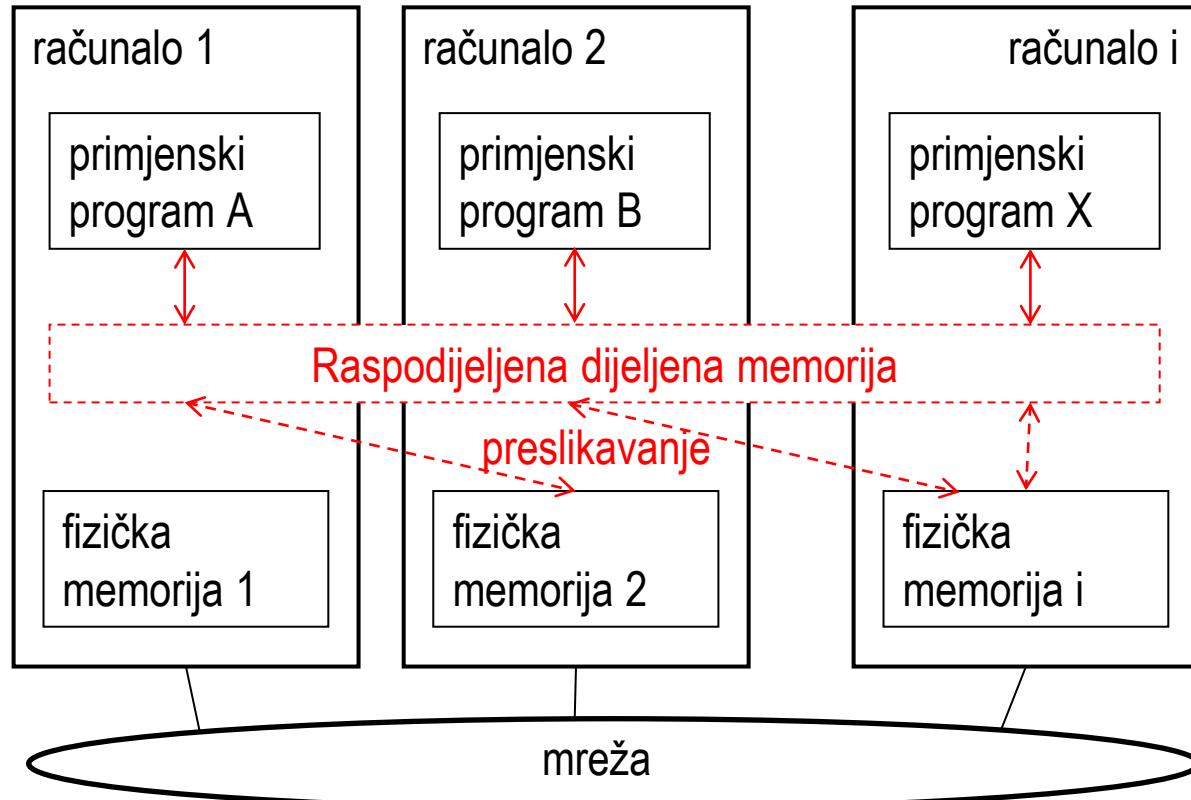
String binding = "*.*.rassus.*";
channel.queueBind(queueName, EXCHANGE_NAME, binding);

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);
```

# 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)*

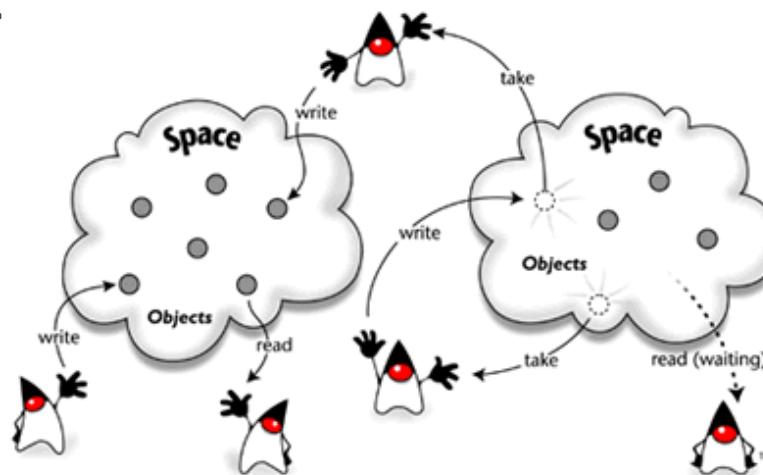
- omogućuje transparentan pristup fizičkoj memoriji na drugim računalima (primjenski program ima dojam da pristupa vlatitoj fizičkoj memoriji)
- upravlja replikama podataka, na računalu i čuvaju se lokalne kopije podataka kojima je nedavno pristupao primjenski program X

# Dijeljeni podatkovni prostor

engl. *shared data/tuple spaces*

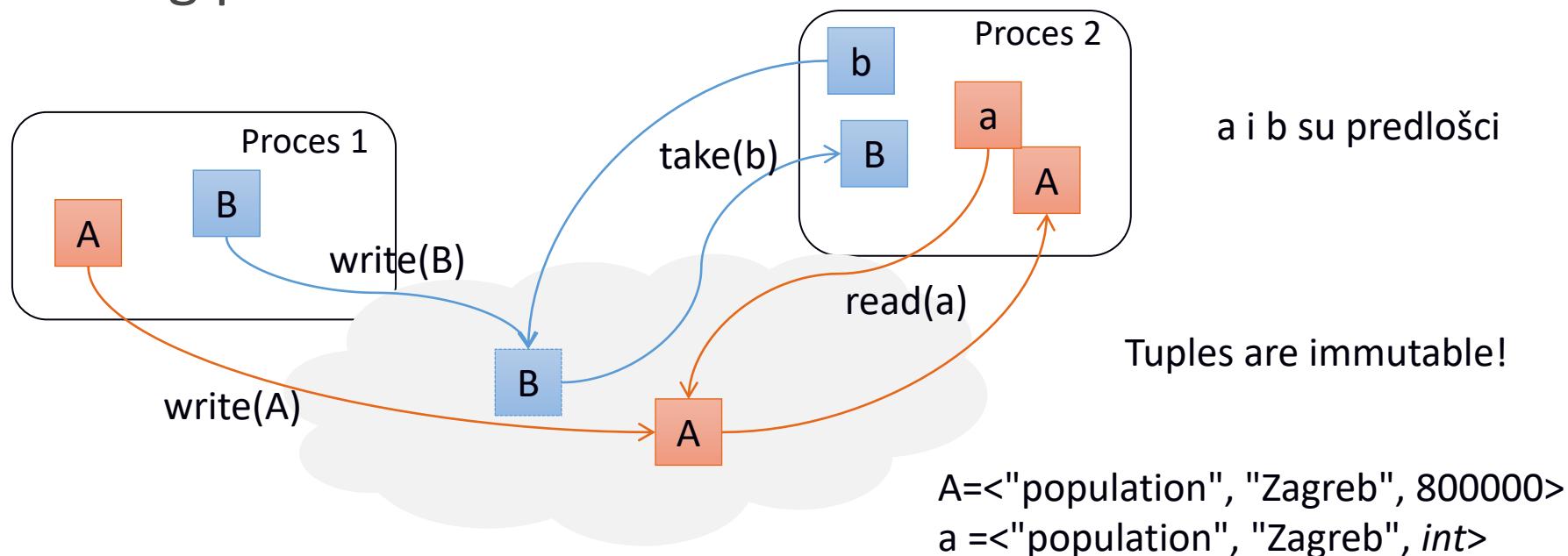
- arhitektura temeljna na podacima (*content-addressable memory*)
- procesi mogu dodati, čitati i “izvaditi” *tuple* iz zajedničkog dijeljenog podatkovnog prostora (*tuple space*)
- *tuple*: slijed podataka, za svaki je definir
- primjeri: Linda, JavaSpaces,  
TSpaces

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 komunikacijski 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?

# Literatura

1. G. Coulouris, J. Dollimore, T. Kindberg: *Distributed Systems: Concepts and Design*, 5th edition, Addison-Wesley, 2012  
poglavlje 6
2. A. S. Tanenbaum, M. Van Steen: *Distributed Systems: Principles and Paradigms*, Second Edition, Prentice Hall, 2007  
poglavlje 4.3



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**5. Formalni model raspodijeljenog sustava  
i primjeri raspodijeljenih algoritama**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

# Raspodijeljeni (distribuirani) algoritam

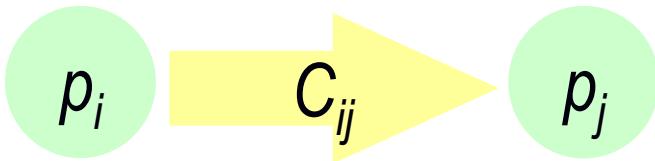
- Algoritam koji se izvodi u raspodijeljenoj okolini, uključuje procese koji komuniciraju razmjenom poruka
- Izvodi se na većem broju računala povezanih mrežom, nije jednostavno odrediti početak i kraj izvođenja algoritma, koliko procesa izvodi raspodijeljeni algoritam, globalno stanje sustava
- Treba biti otporan na ispadne procese jer je to djelomični ispad sustava
- Komunikacija složenost: brojimo poruke koje se generiraju tijekom izvođenja algoritma

# Sadržaj predavanja

- Osnovni model raspodijeljenog sustava
- Model raspodijeljenog izvođenja
- Uzročna ovisnost događaja
- Globalno stanje raspodijeljenog sustava
- Sinkroni model raspodijeljenog sustava
- Asinkroni model raspodijeljenog sustava

# 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  $p_j$

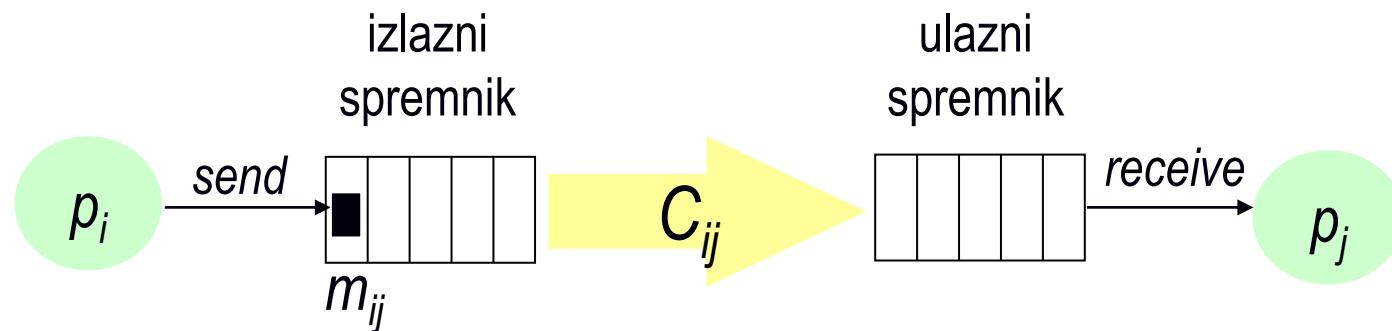


# Svojstva

- 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

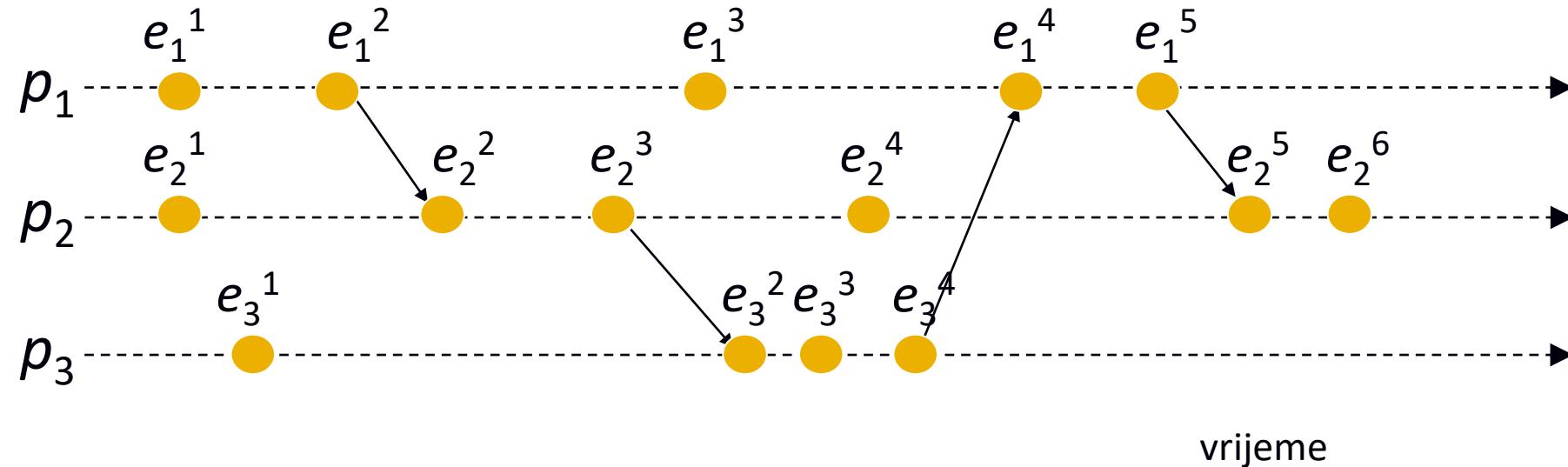
# Model raspodijeljenog izvođenja

- 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  $p_i$ :

$e_i^1, e_i^2, e_i^3, \dots, e_i^x$

( $e_i^2$  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?

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

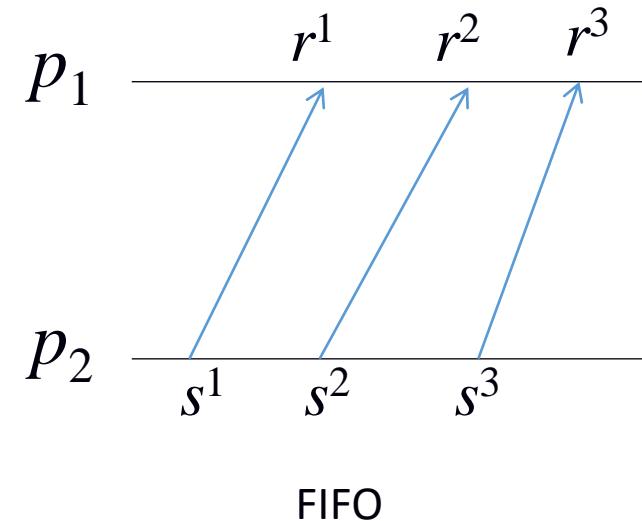
# Uzročna neovisnost događaja

- 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  $e_j$ ,  $e_i \not\rightarrow e_j \Leftrightarrow e_j \not\rightarrow e_i$
  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  $e_j$  konkurentni događaji i to možemo napisati na sljedeći način  $e_i \parallel e_j$

# Model komunikacijskog kanala (1/3)

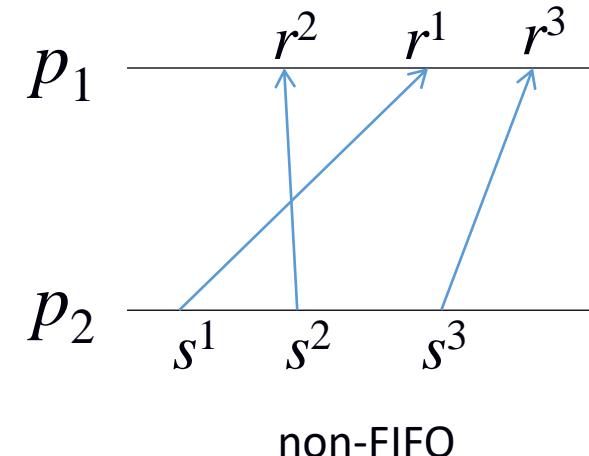
## FIFO (first-in, first-out)

- kanal čuva **slijednost poruka**,  
ponaša se kao rep



## non-FIFO

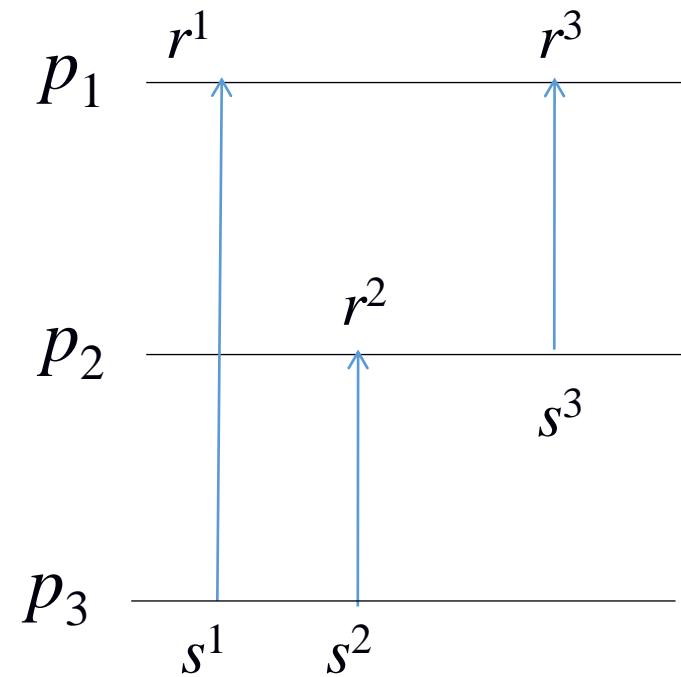
- kanal **ne čuva slijednost** poruka,  
ponaša se kao skup



# Model komunikacijskog kanala (2/3)

## sinkrona slijednost

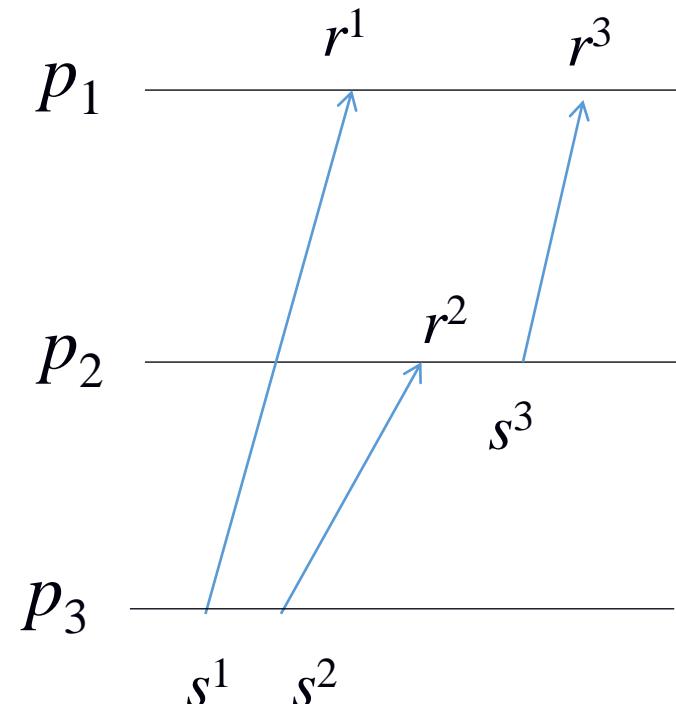
- slanje i primanje poruke događa se istovremeno



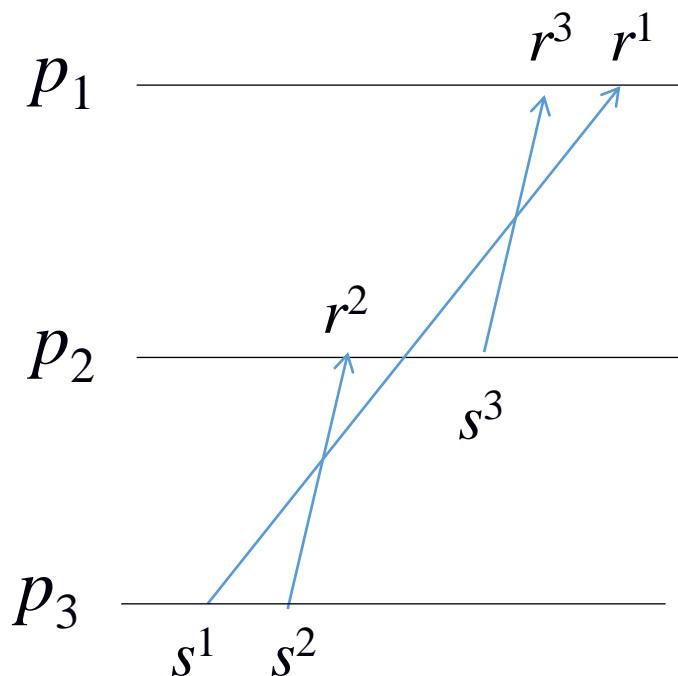
# Model komunikacijskog kanala (3/3)

uzročna slijednost (*causal ordering*, CO)

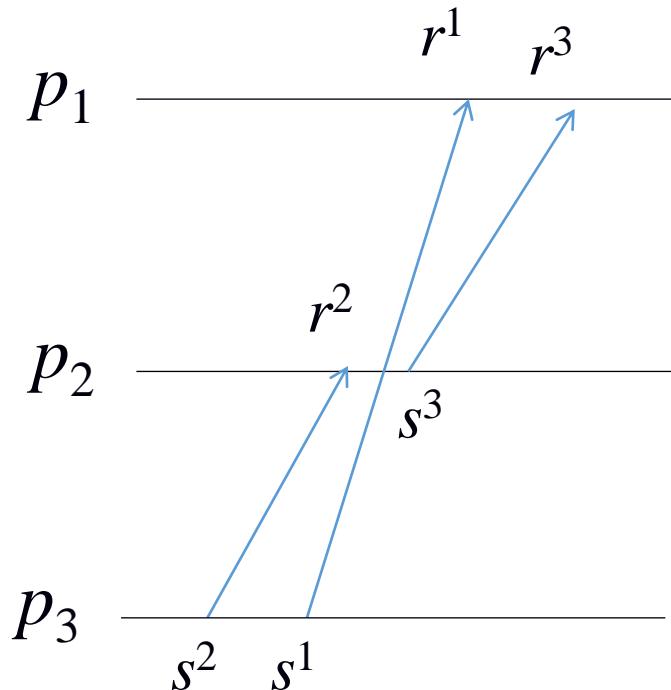
- osigurava da uzročno povezani događaji slanja dviju poruka **istom primatelju rezultiraju** primanjem u slijedu kojim su poslati



# Primjer izvođenja non-CO i CO



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

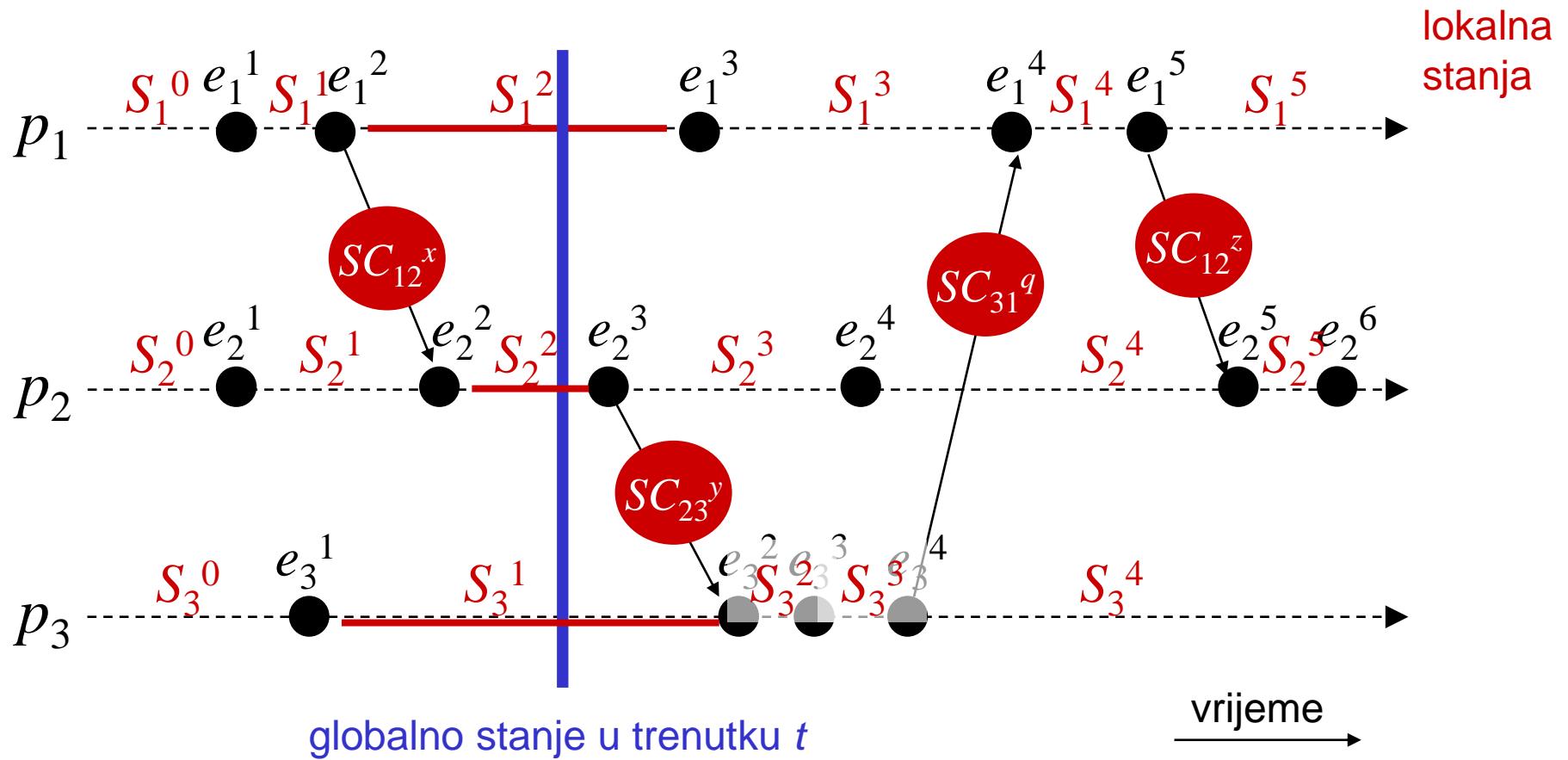


**CO**  
 $s^2 \rightarrow s^1$ ,  $s^2 \rightarrow s^3$  ali odredišta poruka se razlikuju, a kada analiziramo  $r^1$  i  $r^3$ , poruke slanja  $s^1$  i  $s^3$  su neovisne pa je njihov redoslijed irelevantan

# Globalno stanje raspodijeljenog sustava

- Određeno lokanim **stanjima procesa** i kanala
- **Stanje procesa** određeno je **stanjem lokalne memorije** i izvođenjem unutarnjih događaja
- **Stanje kanala** određeno je skupom **primljenih i poslanih** poruka
- Izvođenje događaja mijenja lokano stanje procesa/kanala te istovremeno i globalno stanje raspodijeljenog sustava

# Primjer lokalnog/globalnog stanja



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

# Proširenje osnovnog modela (sinkroni i asinkroni)

## Sinkroni model

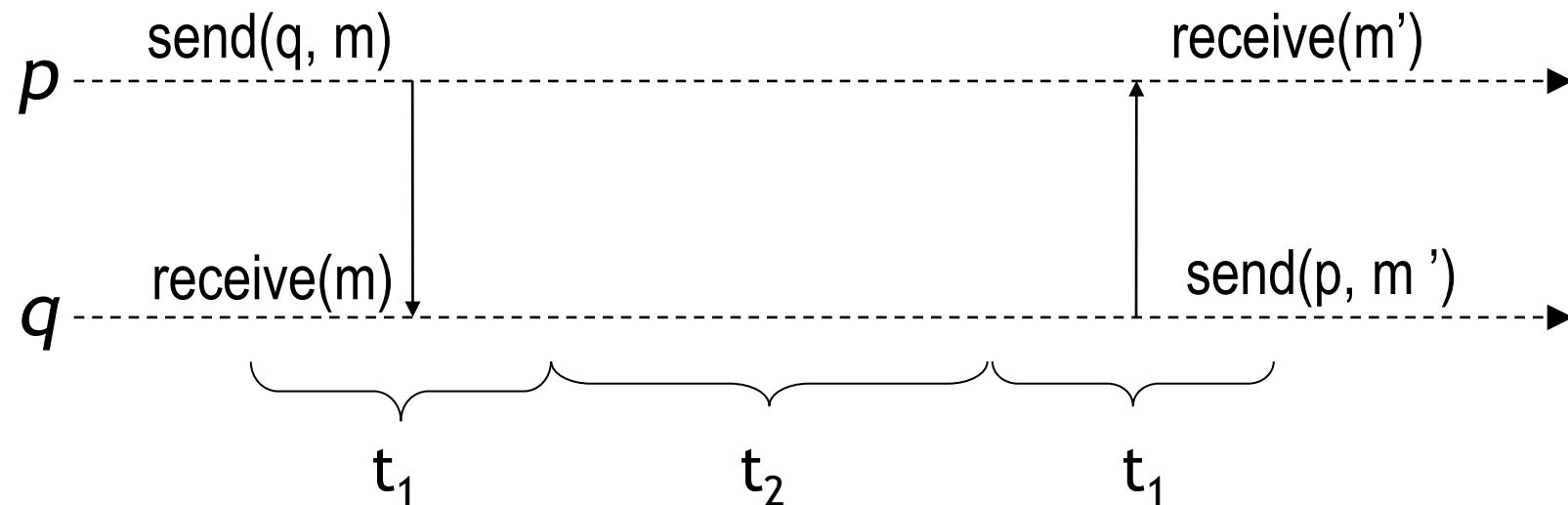
- pretpostavka: svi procesi raspodijeljenog sustava izvode događaje (tj. korake) istovremeno
- pojednostavljenje koje nije realno za raspodijeljene sustave, ali može biti korisno za njihovo razumijevanje i analizu

## Asinkroni model

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

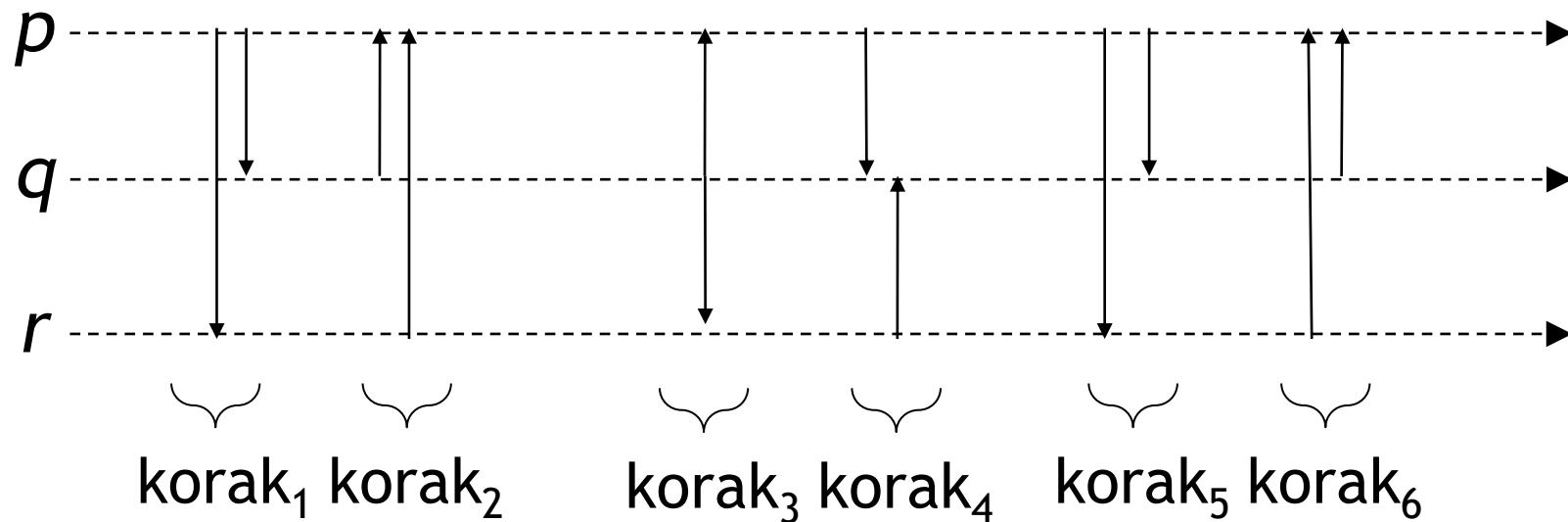
# Sinkroni model

- 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

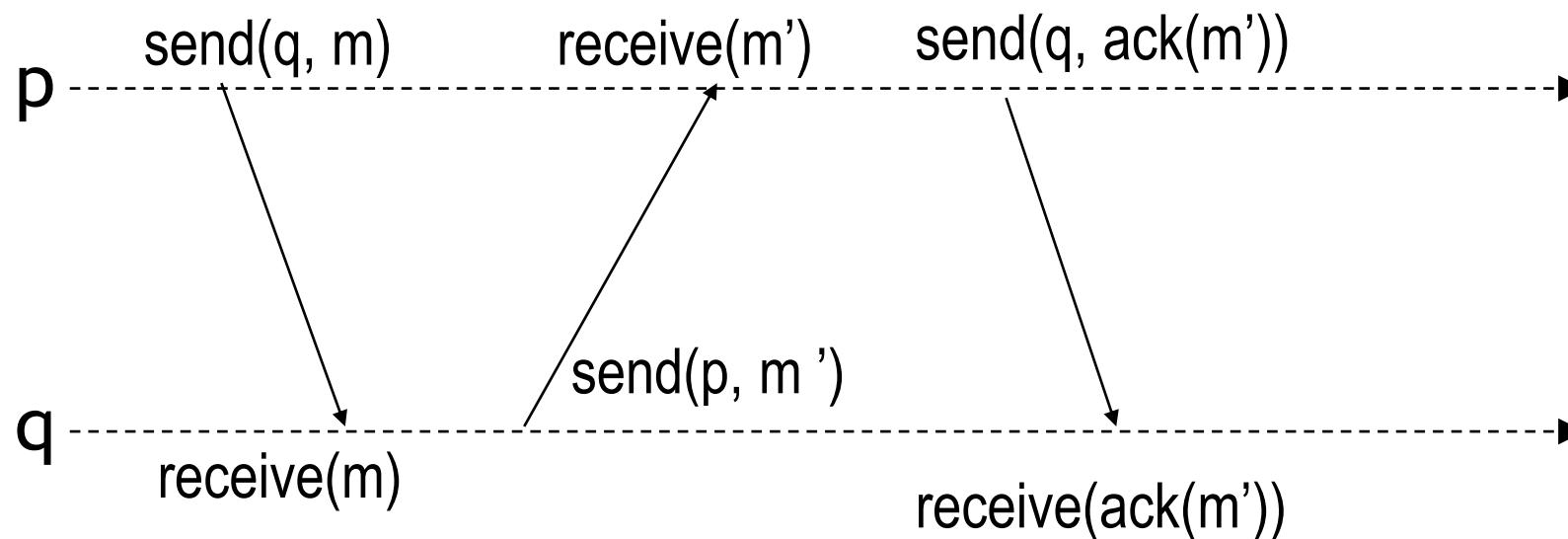
- izvođenje algoritma u sinkronom sustavu organizirano je u koracima
  - pošalji poruke procesima u sustavu
  - primi poruke od drugih procesa u sustavu
  - izvođenje prijelaza: promijeni stanje na temelju primljenih poruka



# 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

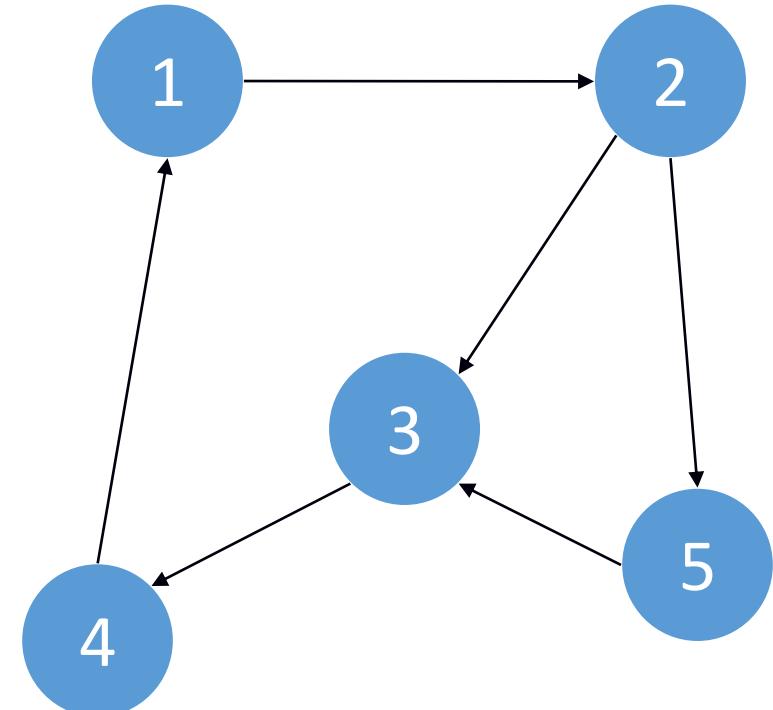


nepouzdani komunikacijski medij, potrebno je modelirati vjerojatnost gubitka poruke na kanalu

# Sinkroni model raspodijeljenog sustava

# Sinkroni model

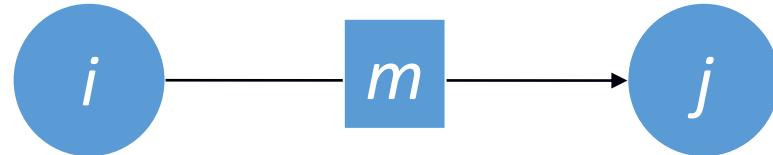
- usmjereni graf  $G = (V, E)$
- $v_i \in V$ , čvor modelira **proces**
- $e_j \in E$ , grana modelira **kanal**
- $M$  je skup poruka, *null* ako na kanalu nema poruka
- $out-nbrs_i$  – izlazni susjedi
- $in-nbrs_i$  – ulazni susjedi
- $distance(i, j)$  – najkraći put između  $i$  i  $j$  u  $G$ , (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

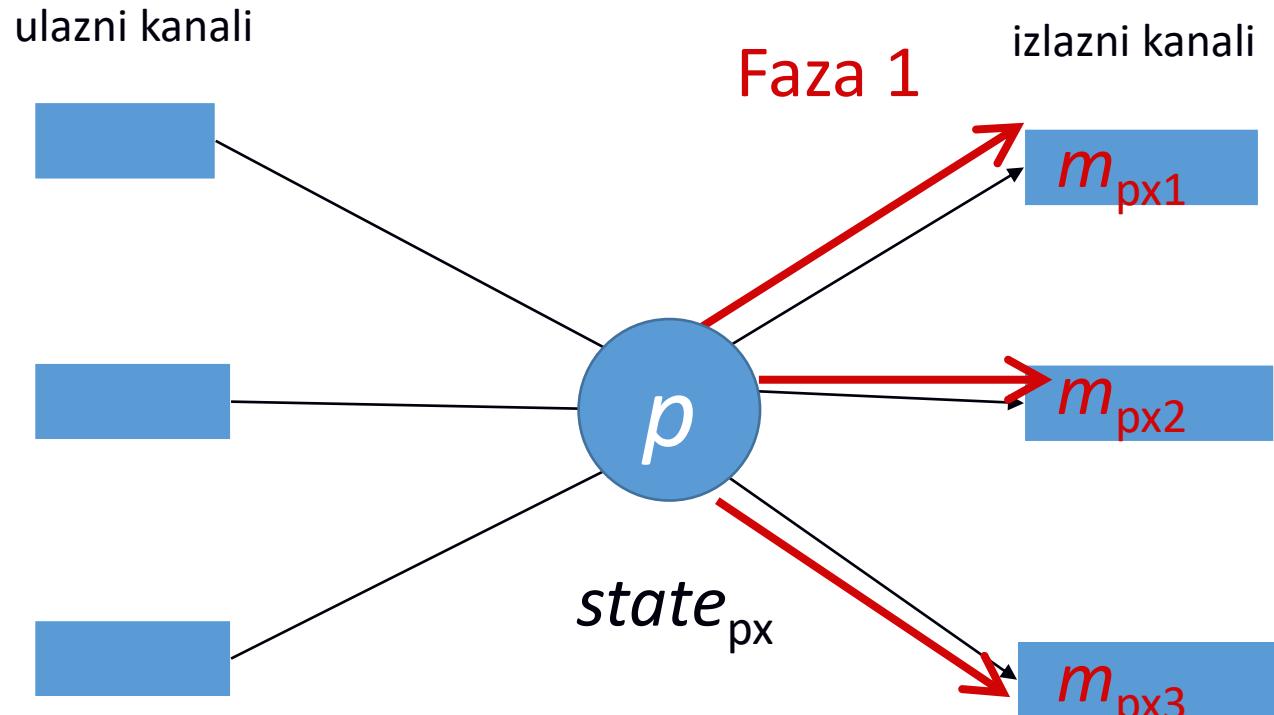


- modelira ga grana između para čvorova  $(i, j)$  iz  $G$
- može primiti poruku  $m$  iz definiranog skupa poruka  $M$  ili *null*
- *null* označava praznu poruku

# Izvođenje sinkronog modela

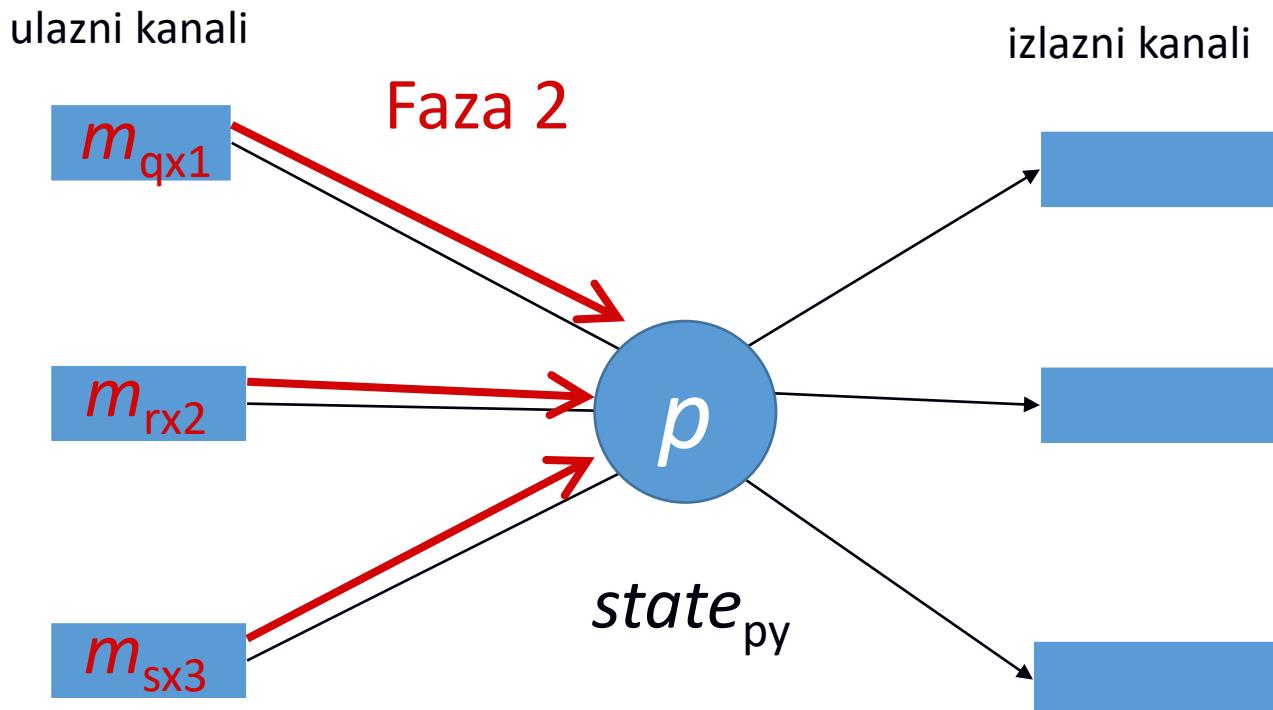
- Algoritmi za sinkrone modele se izvode u **koracima** (*round*). Inicijalno su svi procesi u proizvoljnom početnom stanju i svi su kanali prazni. Nakon toga se izvode koraci.
- Korak se sastoji od 2 faze
  - **Faza 1:** Za svaki proces primjeni funkciju za generiranje poruka (*msg*), a na temelju trenutnog 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!)

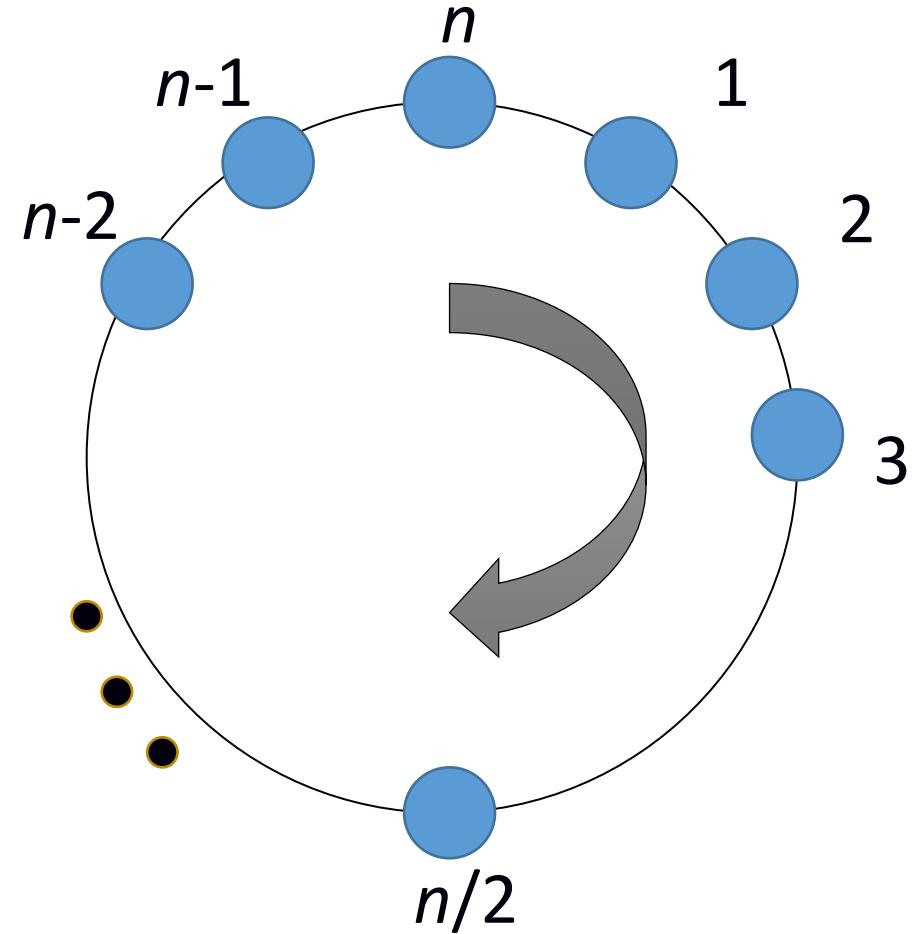
(Usporedite s mjerama složenosti algoritama, vremenska i prostorna)

# 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

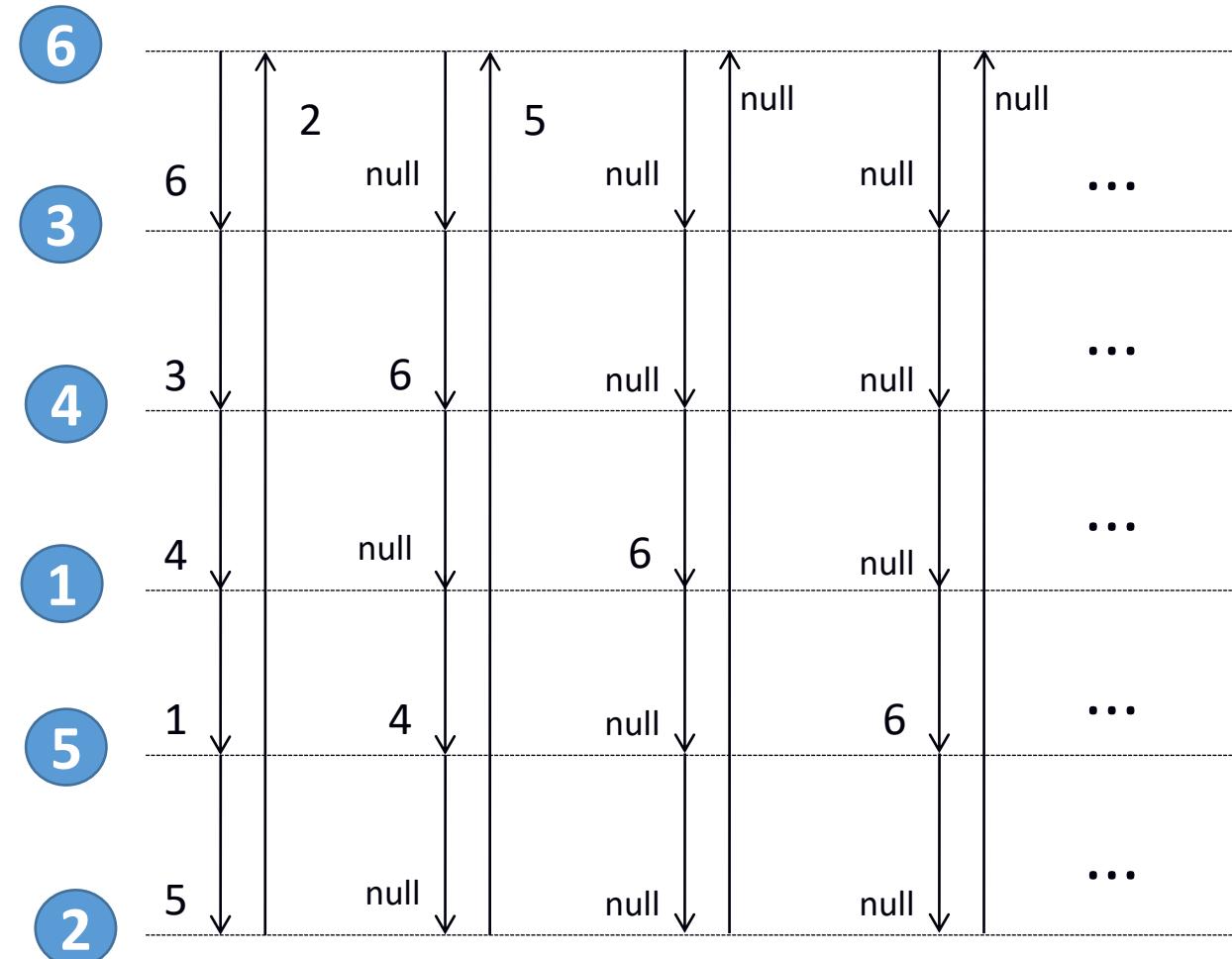
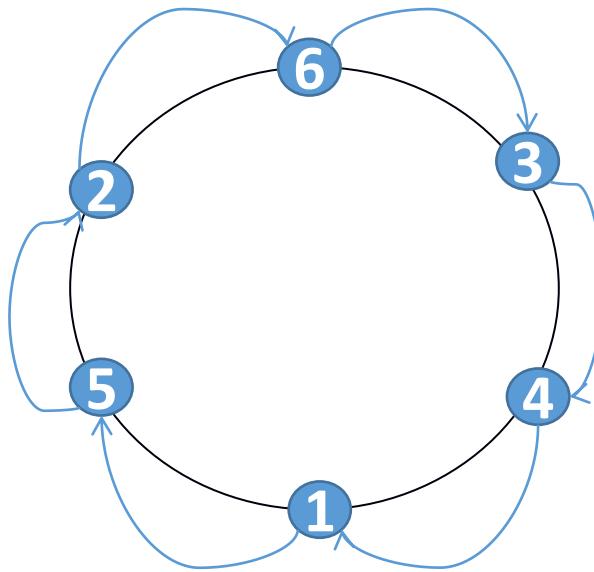
- Svi procesi su identični osim po identifikatoru
  - svaki ima jedinstven identifikator (UID - *Unique ID* )
  - UID nisu sljedbenici u prstenu
  - UID se mogu međusobno uspoređivati
  - (to omogućuje odabir vođe, inače bi svi procesi bili jednaki i vođa se ne bi mogao odabrati)
- Procesi znaju svoje 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 manji 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

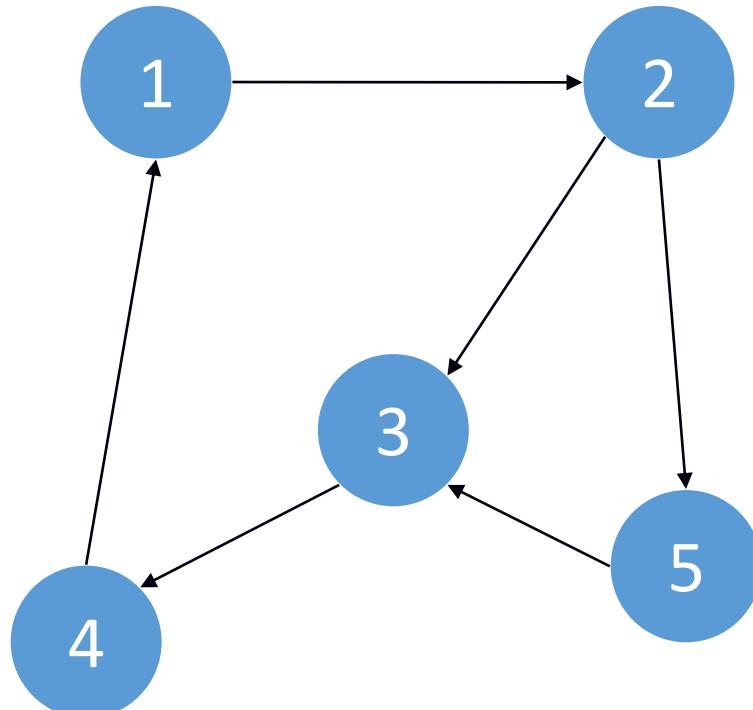
- $M$  – skup poruka čini skup svih UID
- Za svaki proces  $i$ :
  - $states_i = (u, send, status)$ 
    - $u$  – identifikator, inicijalno UID za  $i$
    - $send$  – identifikator ili null, inicijalno UID za  $i$
    - $status \in \{unknown, leader\}$ , inicijalno  $unknown$
  - $start_i = (\text{UID procesa } i, \text{UID procesa } i, unknown)$
  - $msgs_i$  – 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

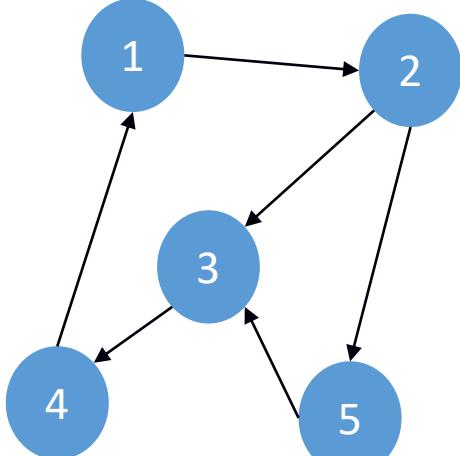
- Povezana usmjerena mreža, za svaki par čvorova postoji konačan  $distance(i, j)$
- Svaki čvor ima jedinstveni identifikator UID
- Izabratи vođу међу procesима у мрежи
- Samo 1 proces mijenja status u *leader*
- $diameter(G) = \max distance(v_i, v_j)$  за sve parove  $(v_i, v_j)$  iz  $G$



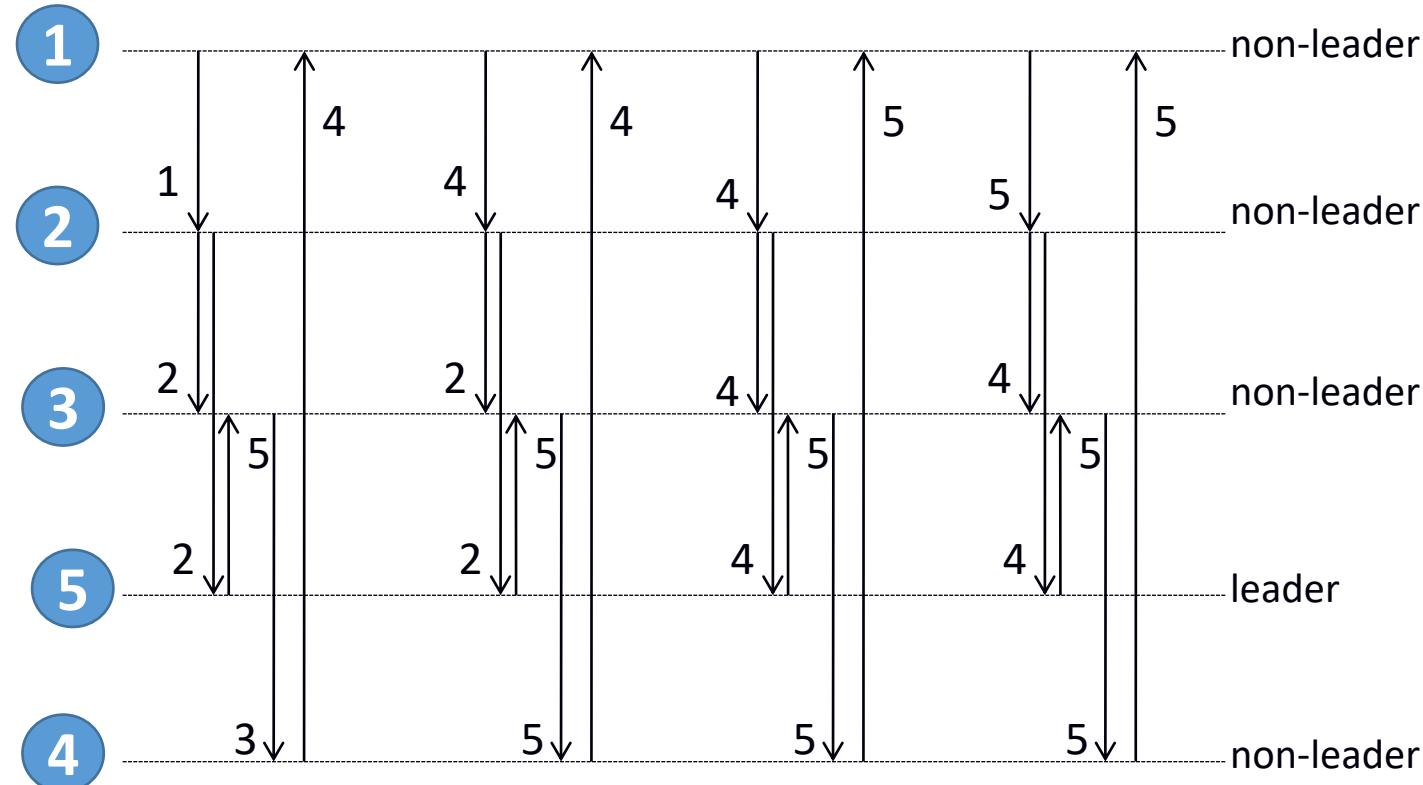
# Rješenje problema 2: Algoritam preplavljanja

- *Simple Flooding Algorithm (SFA)*
- Prepostavke
  - Svaki proces ima UID iz skupa prirodnih brojeva
  - Svaki proces zna  $diameter(G)$
- 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  $max\text{-}uid$  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  $status := non-leader$

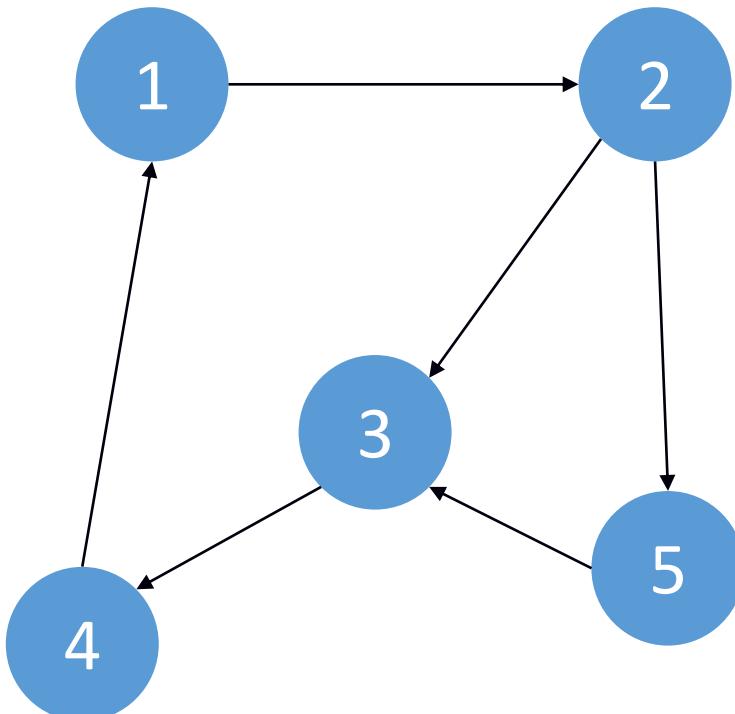
# Složenost

- 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

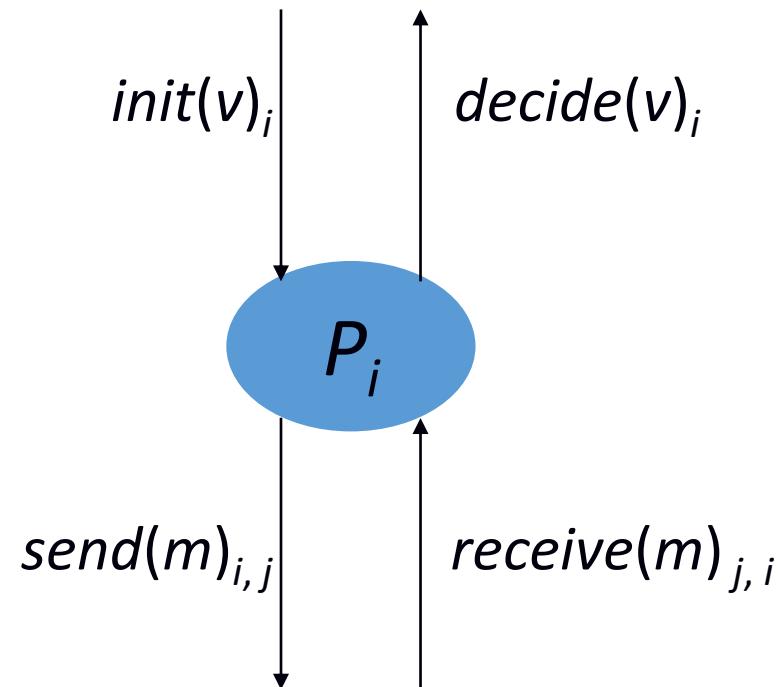
# Asinkroni model

- ♦ 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.

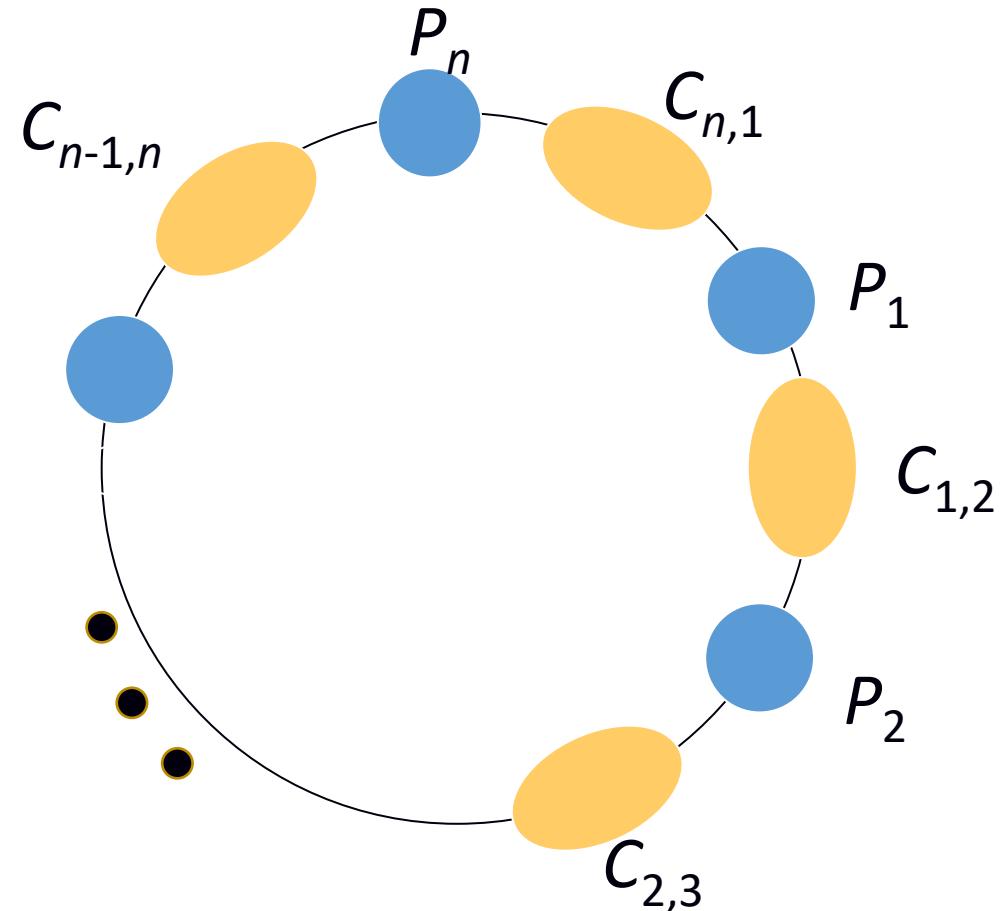
(prema N. Lynch)

# Primjeri asinkronog modela

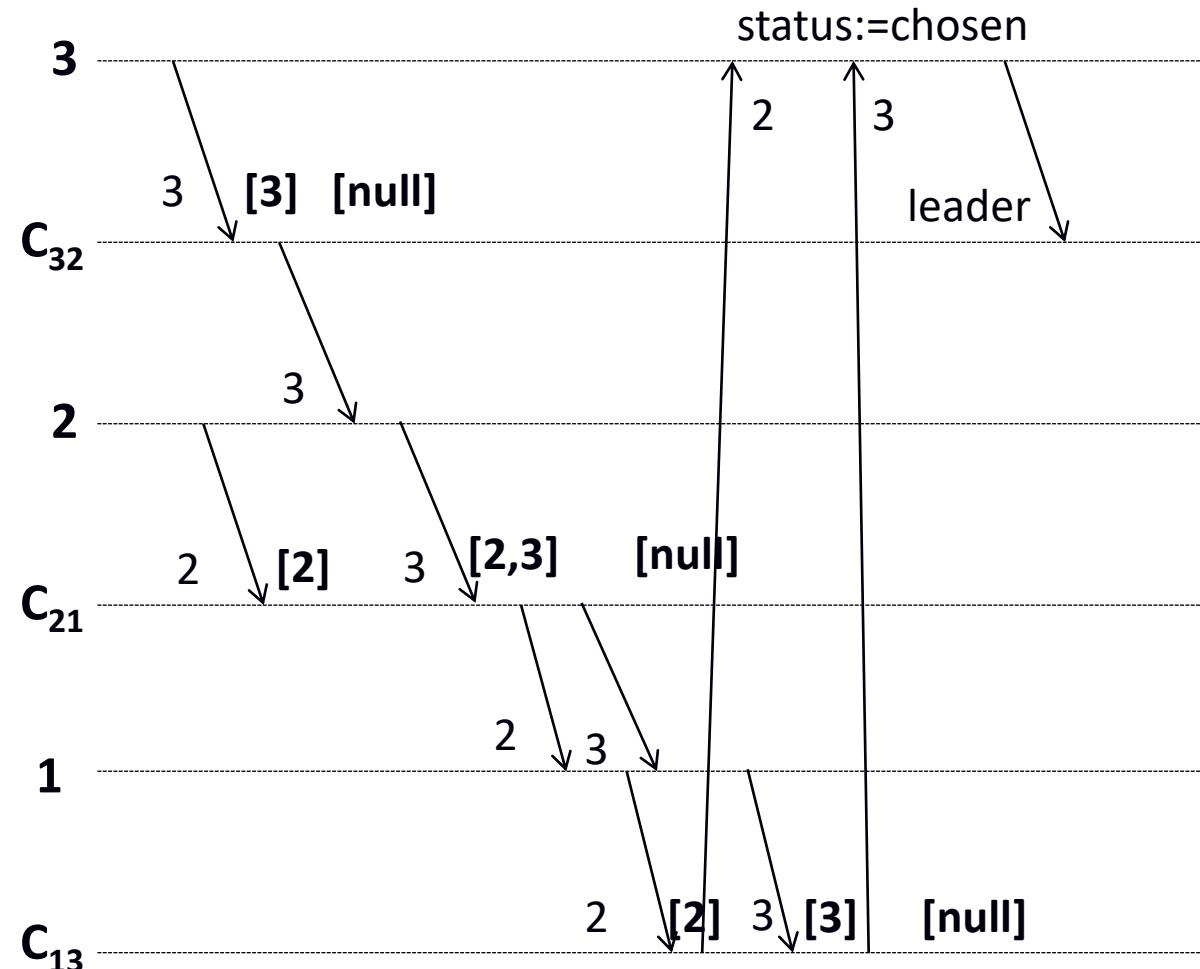
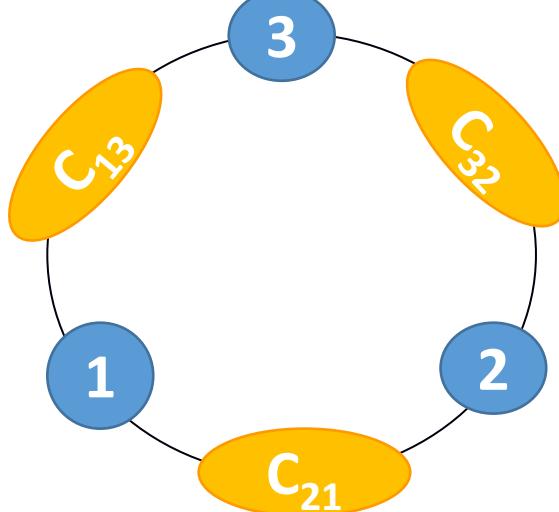
# Odabir vođe u asinkronoj mreži

## 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

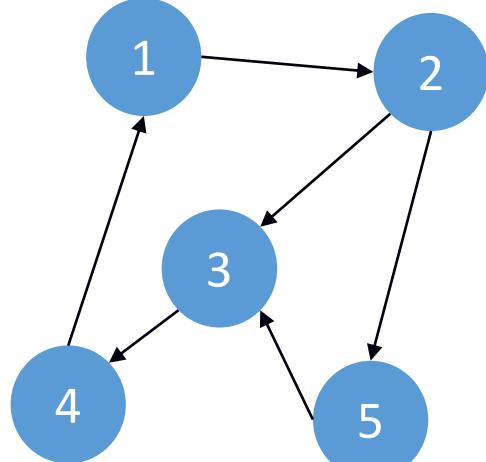
Definicija automata procesa  $P_i$

- 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

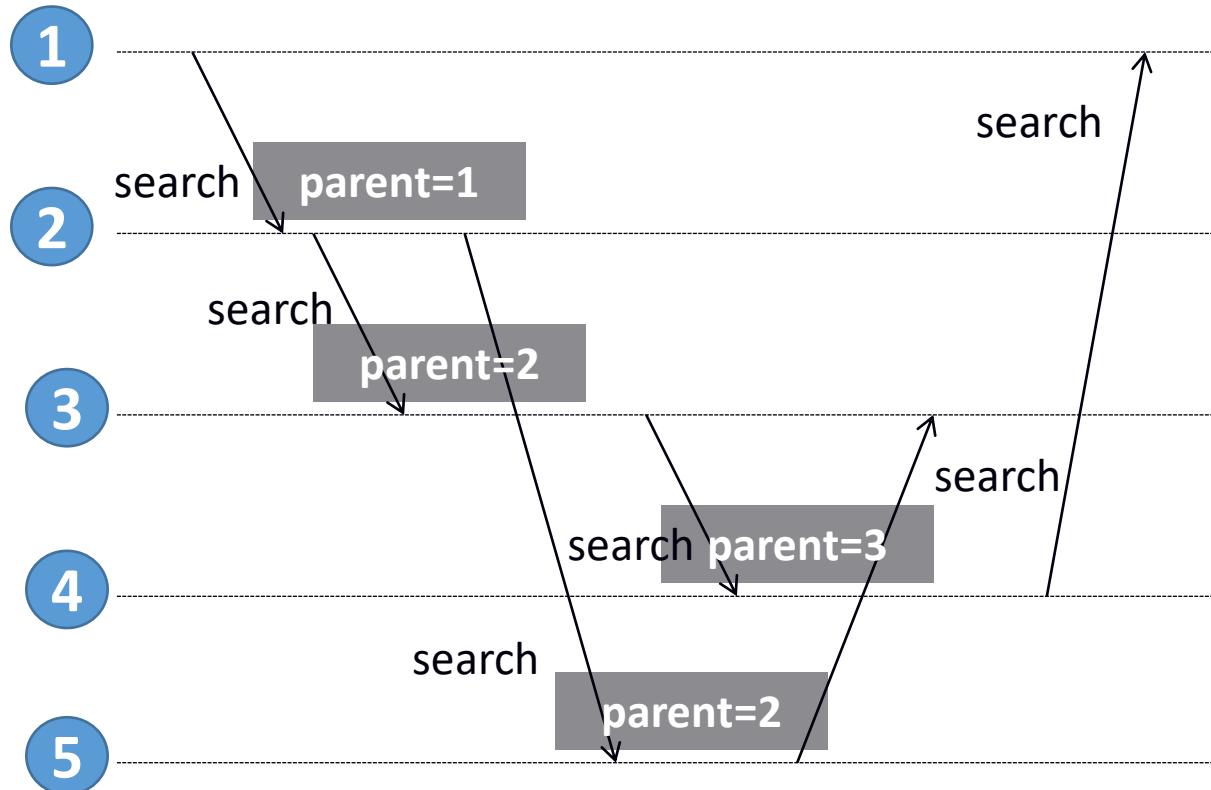
# Kreiranje stabla u asinkronoj mreži

- 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  $P_i$

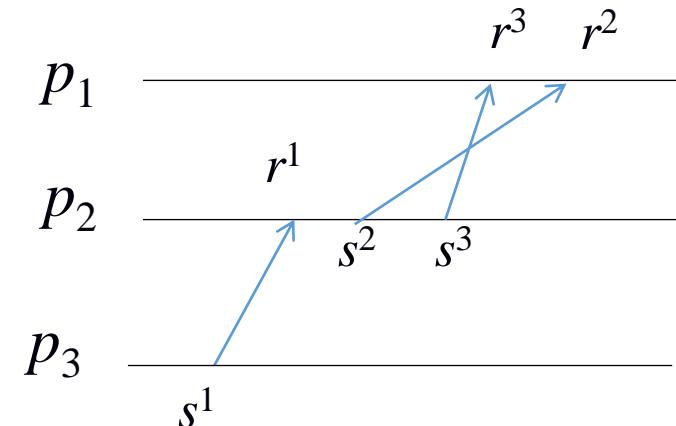
- input:  $receive("search")_{j,i}$ ,  $j \in nbrs$
- output:  $send("search")_{i,j}$ ,  $j \in nbrs$ ;  $parent(j)_i$ ,  $j \in nbrs$
- states:
  - $parent \in nbrs \cup \{null\}$ , inicijalno  $null$
  - $reported$  – boolean, inicijalno  $false$
  - za svaki  $j \in nbrs$  postoji  
 $send(j) \in \{search, null\}$ , 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.
- N. Lynch: *Distributed Algorithms*, Morgan Kaufmann Publishers Inc. 1996

# Pitanja za učenje i ponavljanje

- Za koje je svojstvo raspodijeljenih sustava značajna komunikacijska složenost algoritama? Zašto?
  - a) replikacijska transparentnost
  - b) skalabilnost
  - c) otvorenost
- Objasnite model komunikacijskog kanala koji se temelji na uzročnoj slijednosti. Vrijedi li za slijedeći primjer CO ili non-CO i zašto?





SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**  
Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**6. Sinkronizacija procesa u vremenu**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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



*U slučaju 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.*

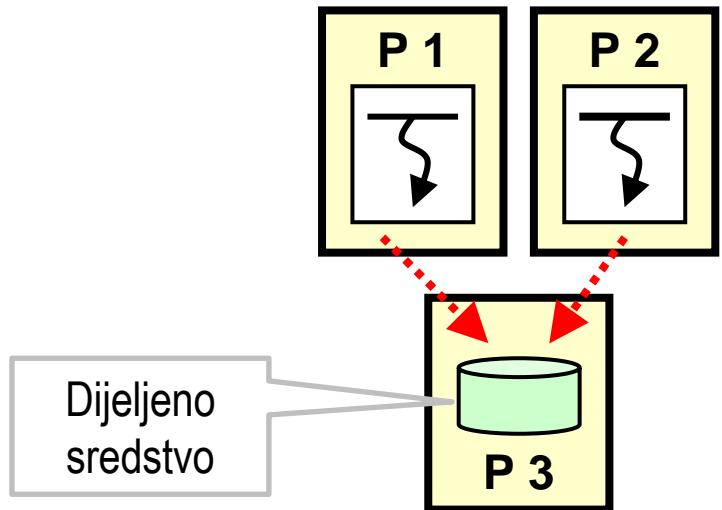
*Tekst licence preuzet je s <http://creativecommons.org/>*

# Sadržaj predavanja

- Motivacija: potreba za sinkronizacijom procesa u raspodijeljenoj okolini
- Primjena sata u jednoprocesorskoj okolini
- Primjena sata u raspodijeljenoj okolini
- Sinkronizacija tijeka izvođenja procesa
- Međusobno isključivanje procesa

# Potreba za sinkronizacijom procesa (1/4)

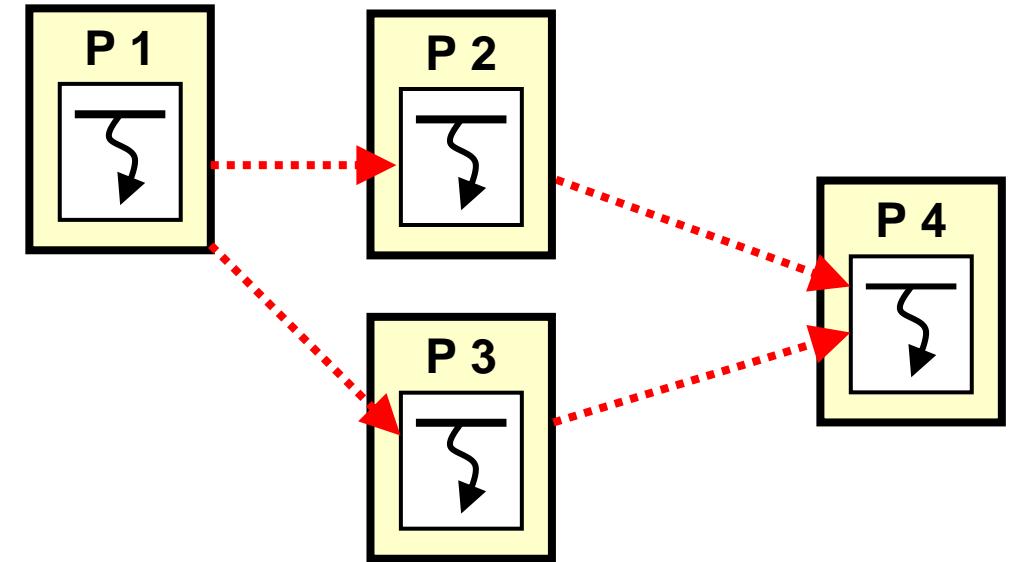
- Uporaba **dijeljenog sredstva** u raspodijeljenoj okolini
  - Procesi istodobno pristupaju dijeljenom sredstvu
  - Potrebno je ostvariti pristup dijeljenom sredstvu na međusobno isključiv način
  - Raspodijeljeni procesi moraju postići dogovor o redoslijedu pristupa sredstvu



# Potreba za sinkronizacijom procesa (2/4)

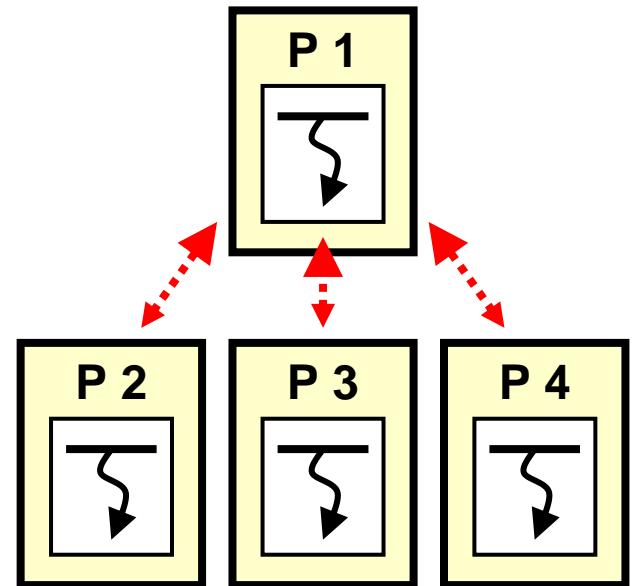
- Usuglašavanje **vremenskog redoslijeda izvođenja akcija**

- Potrebno je omogućiti vremenski tijek izvođenja akcija na procesima u raspodijeljenoj okolini ako postoji međuvisnost među procesima



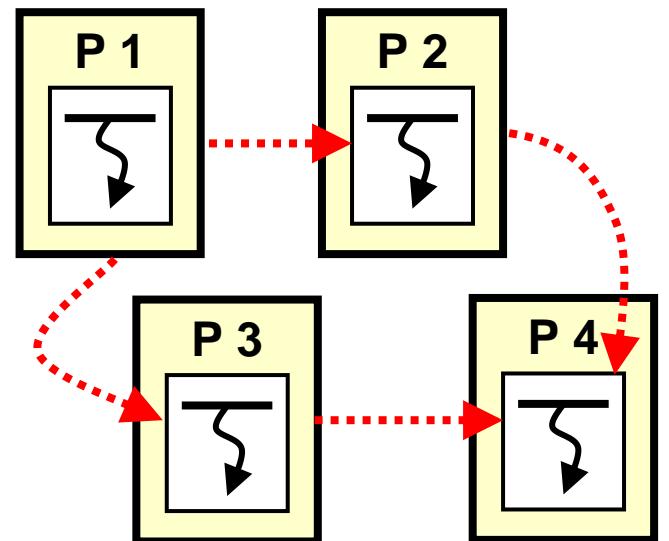
# Potreba za sinkronizacijom procesa (3/4)

- **Nadgledanje i upravljanje** nad izvođenjem poslova u raspodijeljenoj okolini
  - Odabir upravljačkog procesa (sjetite se primjera s prethodnog predavanja)
  - Upravljački proces nadzire i određuje aktivnosti radnih procesa u raspodijeljenoj okolini



# Potreba za sinkronizacijom procesa (4/4)

- Uspostava **suradnje skupa procesa** u raspodijeljenoj okolini
  - Ostvarivanje vremenski i **prostorno** usklađenog raspodijeljenog tijeka izvođenja (proširenje 2. primjera)
  - Primjer: P2P sustav za dijeljenje datoteka

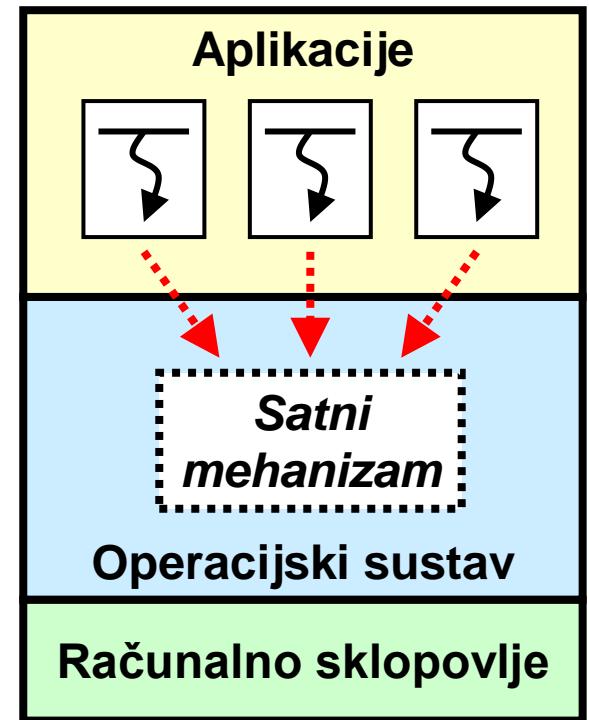


# Sadržaj predavanja

- Motivacija: potreba za sinkronizacijom procesa u raspodijeljenoj okolini
- Primjena sata u jednoprocesorskoj okolini
- Primjena sata u raspodijeljenoj okolini
- Sinkronizacija tijeka izvođenja procesa
- Međusobno isključivanje procesa

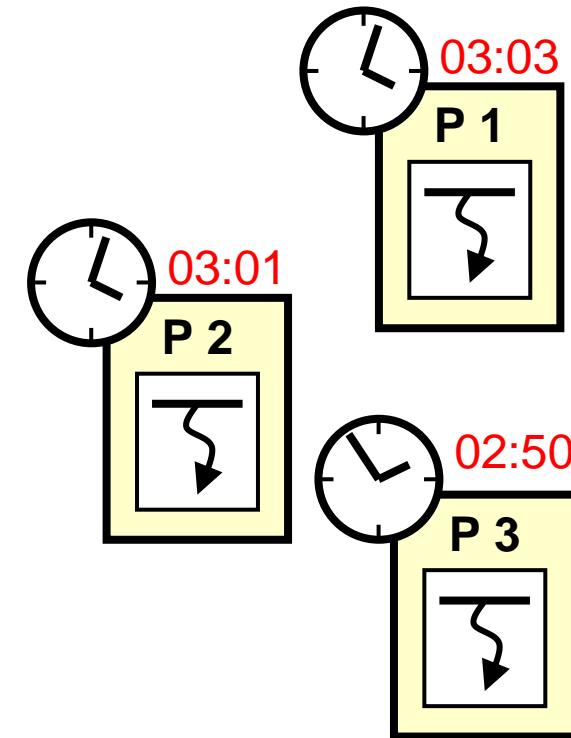
# Primjena sata u jednoprocesorskoj okolini

- Podsjetimo se: satni mehanizam operacijskog sustava
  - Izведен uporabom kristala kvarca, osciliraju pod naponom zbog piezoelektričkog efekta
- Aplikacije
  - Procesi koriste i upravljaju mehanizmom sata
  - Primjena programskih knjižnica za uporabu satnog mehanizma
- Zatvorena okolina
  - Predvidiva vremena izvođenja procesa
  - Jednostavnija sinkronizacija procesa u vremenu



# Fizički i logički sat

- Svako računalo ima vlastiti satni mehanizam
  - Satovi nisu usklađeni
  - Satovi imaju različiti takt
  - Satovi imaju različita odstupanja (pogrešku)
- Usuglašavanje vremena
  - Fizički sat u raspodijeljenoj okolini
  - Logički sat u raspodijeljenoj okolini

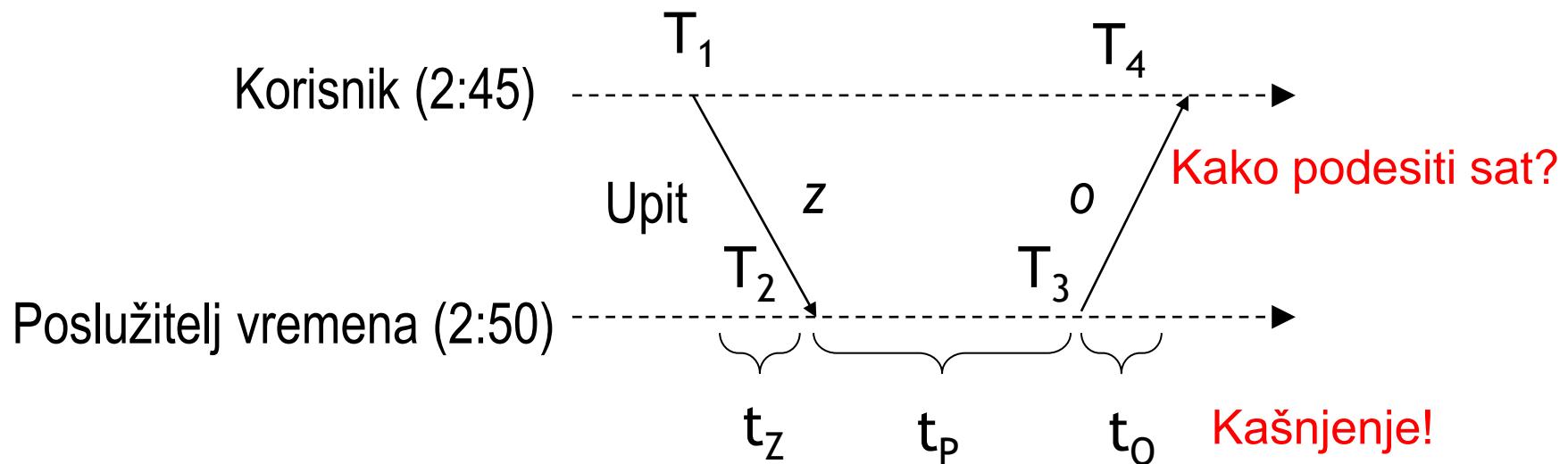


# Fizički sat u raspodijeljenoj okolini

- **Cristianov algoritam**
  - Razvio ga je F. Cristian (1989)
  - Primjena poslužitelja s točnim vremenom, sinkronizacija prema vanjskom izvoru
  - Dohvaćanje informacije o vremenu prema potrebi
- **Algoritam Berkeley**
  - Razvili su ga R. Gusella i S. Zatti na University of California, Berkeley (1989)
  - Primjena upravitelja vremena, sinkronizacija unutar skupine procesa
  - Periodičko odašiljanje informacije o vremenu

# Cristianov algoritam (1/2)

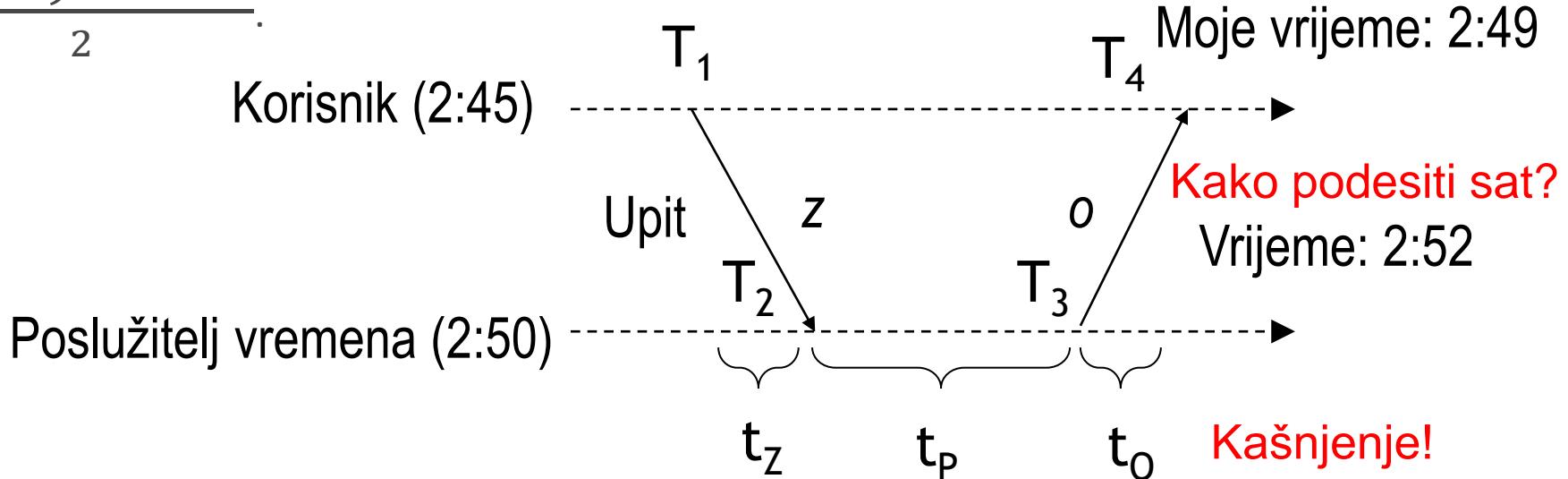
- Primjena poslužitelja vremena
- Koraci algoritma
  - 1) Korisnički proces upućuje zahtjev za dohvata vremena ( $z$ )
  - 2) Poslužitelj vremena prima i obrađuje zahtjev te šalje trenutno vrijeme ( $o$ )



# Cristianov algoritam (1/2)

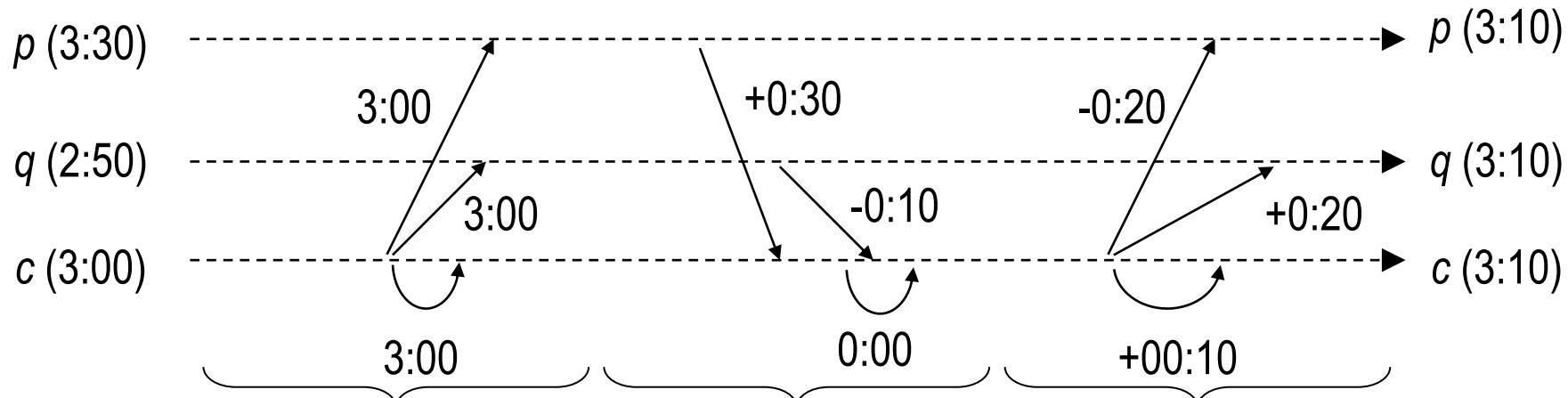
3. Odgovor sadrži  $T_2$  i  $T_3$  na poslužitelju u trenutku slanja odgovora.
4. Klijent na osnovi vremenskih trenutaka koje je primio u poruci i izmjerениh vremenskih trenutaka  $T_1$  i  $T_4$  pomiče svoje lokalno vrijeme za

$$\theta = T_3 + t_O - T_4 \approx T_3 + \frac{t_Z + t_O}{2} - T_4 = T_3 + \frac{(T_4 - T_1) - (T_3 - T_2)}{2} - T_4 = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}.$$



# Algoritam Berkeley

- Primjena upravitelja vremena
- Koraci algoritma
  - 1) Upravljački proces  $c$  šalje vrijeme procesima  $p, q, c$
  - 2) Procesi  $p, q, c$  šalju razliku vremena upravljačkom procesu  $c$
  - 3) Upravljački proces  $c$  šalje pomak procesima  $p, q, c$



# Network Time Protocol (NTP)

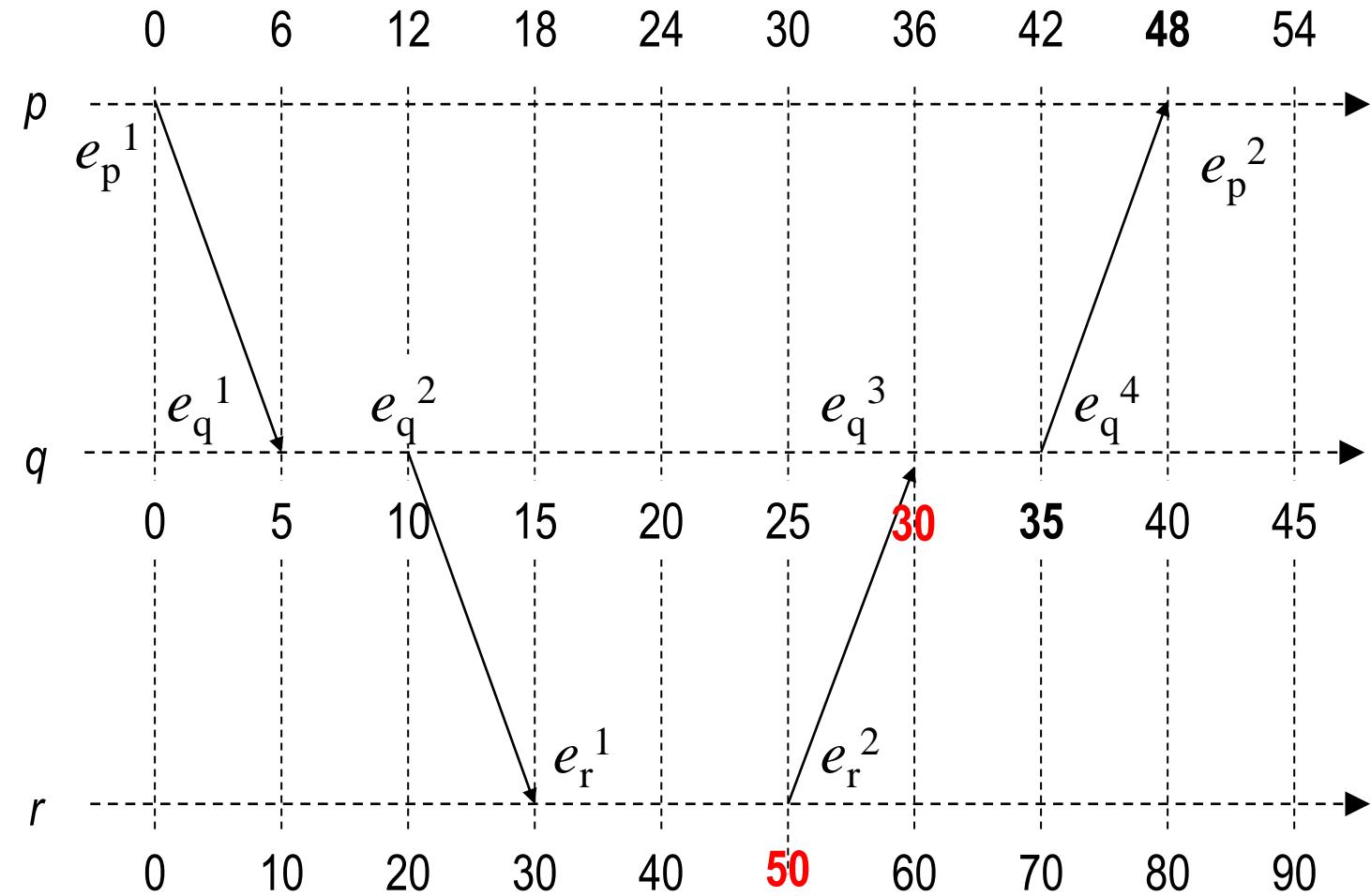
- Definira arhitekturu usluge za sinkronizaciju satnih mehanizama u raspodijeljenoj okolini i protokol za isporuku informacija o vremenu u Internetu
- Hjerarhijska organizacija NTP servisa
  - *primary servers*: povezani direktno na izvor sinkroniziran na UTC (Coordinated Universal Time)
  - *secondary servers*: sinkroniziraju se u odnosu na *primary servers*, itd.
  - preciznost: ~10 ms za računala na javnom Internetu, ~1 ms za računala u LAN-u
- RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification, 2010
- **Nedostatak fizičkog sata:** fizički satni mehanizmi su potpuno neovisni, stoga samo primjenom fizičkih satnih mehanizama nije moguće odrediti odnos događaja u vremenu (npr. redoslijed aktivnosti)

# Primjer nedostatka fizičkog sata

$$e_p^1 \rightarrow e_q^1 \quad e_q^2 \rightarrow e_r^1$$

$$e_r^2 \rightarrow e_q^3 \quad e_q^4 \rightarrow e_p^2$$

$$\mathbf{T(e_r^2) > T(e_q^3)}$$



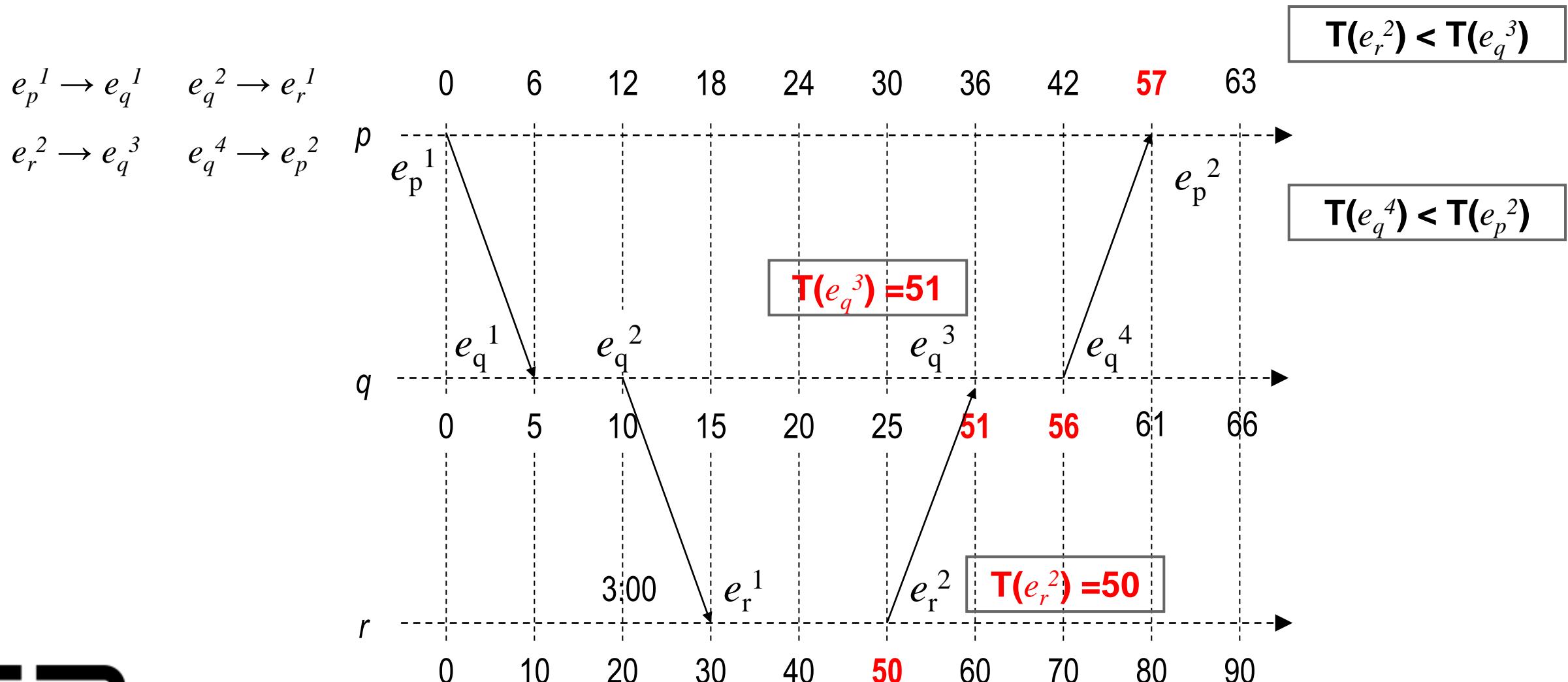
# Logičke oznake vremena

- Usklađivanje globalnog tijeka vremena
  - Primjena logičkih oznaka vremena (*timestamps*)
- Vrste logičkih oznaka
  - Skalarne oznake vremena
  - Vektorske oznake vremena

# Skalarne oznake vremena

- **Globalno logičko vrijeme**
  - Sva računala na jednak način bilježe tijek globalnog logičkog vremena
- **Oznake logičkog vremena**
  - Svakoj akciji  $a$  koju provode procesi u raspodijeljenoj okolini pridružena je jedinstvena oznaka vremena  $T(a)$
  - Ako za događaj  $a$  i  $b$  vrijedi uzročna relacija  $a \rightarrow b$  tada vrijedi da je akcija  $a$  ostvarena u vremenu prije akcije  $b$  [  $T(a) < T(b)$  ]

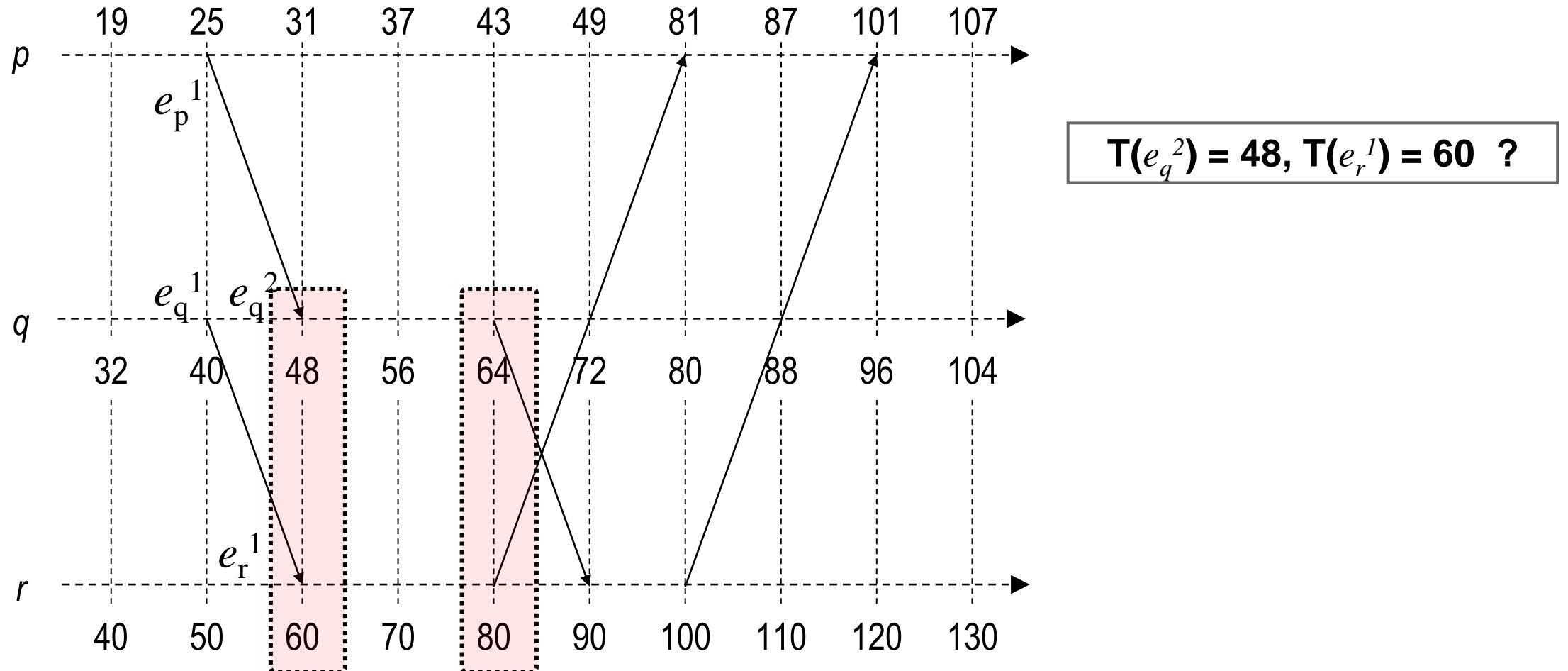
# Primjer uporabe skalarnih oznaka vremena



# Obilježja skalarne oznake vremena

- **Prednosti primjene skalarnih oznaka**
  - Tijek vremena zasnovan je na jednostavnom modelu
  - Svi procesi usklađeni su s globalnim tijekom vremena
  - Usuglašeni su vremenski trenutci nastupanja akcija u raspodijeljenoj okolini
- **Nedostatci primjene skalarnih oznaka**
  - Ako za događaje  $a$  i  $b$  vrijedi da je vremenska oznaka od  $a$  manja od vremenske oznake od  $b$ , to ne povlači nužno da je događaj  $a$  nastupio u vremenu prije događaja  $b$
  - $T(a) < T(b)$  ne povlači  $a \rightarrow b$

# Primjer nedostatka skalarnih oznaka



# Vektorske oznake vremena (1/2)

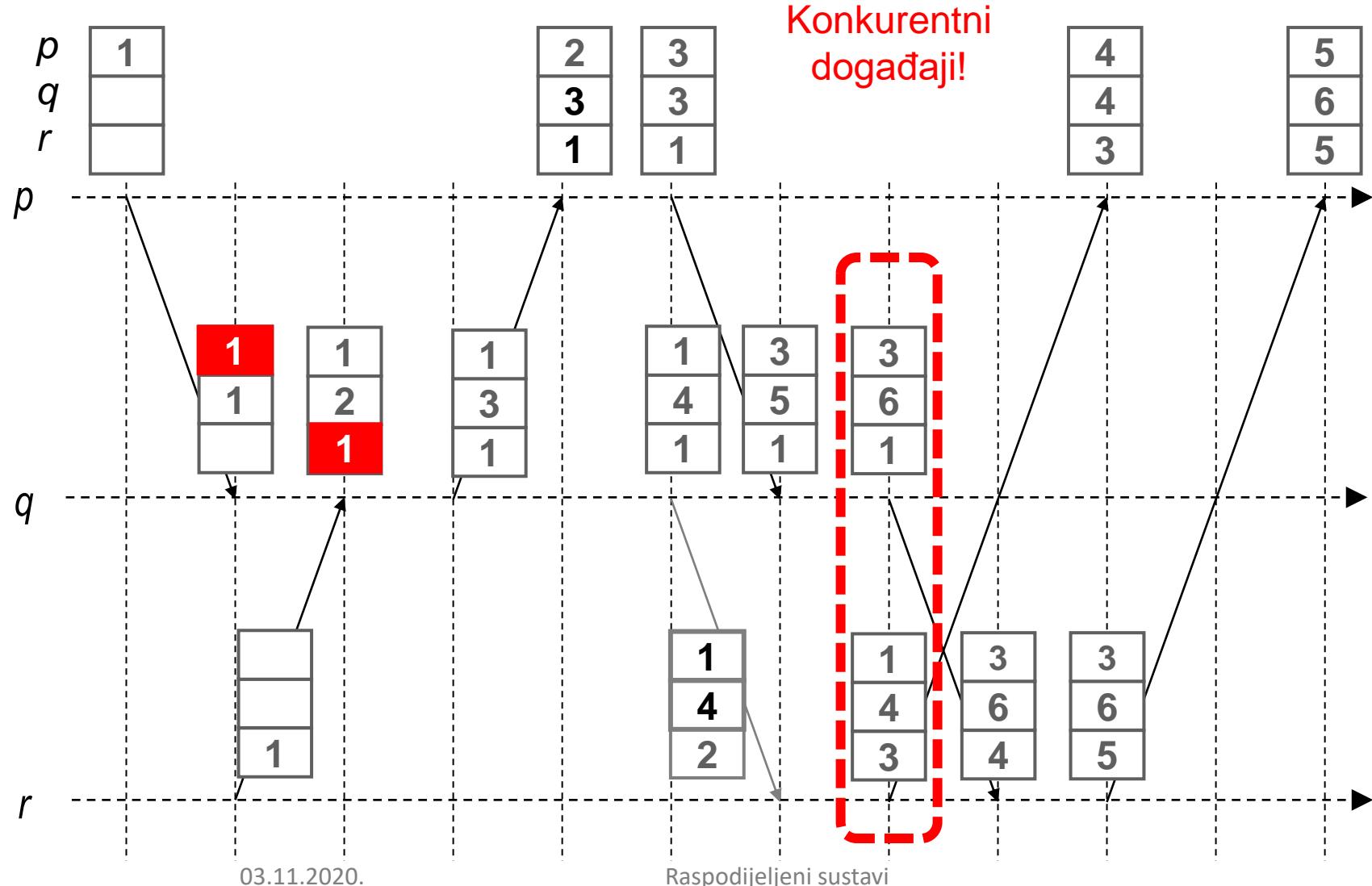
- Vektorska oznaka opisuje uzročno-posljedične veze između događaja u vremenu
  - Polje elemenata  $V[N]$  opisuje broj akcija (unutarnja akcija, slanje poruke, prijam poruke) provedenih od  $N$  procesa u raspodijeljenoj okolini
  - Procesi razmjenjuju vektorske oznake tijekom razmjene poruka
- Vektorska oznaka
  - $V_p[p]$  broj akcija koje je ostvario proces  $P_p$
  - $V_p[m]$  broj akcija za koje proces  $P_p$  zna da su ostvarene od strane procesa  $P_m$



# Vektorske oznake vremena (2/2)

- **Primjena vektorskih oznaka**
  - Ako za događaje  $a$  i  $b$  vrijedi  $V(a) < V(b)$  tada vrijedi da je događaj  $a$  nastupio u vremenu prije događaja  $b$ ,  $a \rightarrow b$
- **Za dvije vektorske oznake  $V_i$  i  $V_j$  vrijedi  $V_i < V_j$  ako:**
  - postoji barem jedan  $k$  za koji vrijedi  $V_i[k] < V_j[k]$ ,
  - za sve ostale  $l \neq k$  vrijedi  $V_i[l] \leq V_j[l]$ ,
  - $i, j, k, l \in [0, N-1]$  i
  - broj procesa u raspodijeljenoj okolini je  $N$

# Primjer uporabe vektorskih oznaka



# Vektorske oznake vremena

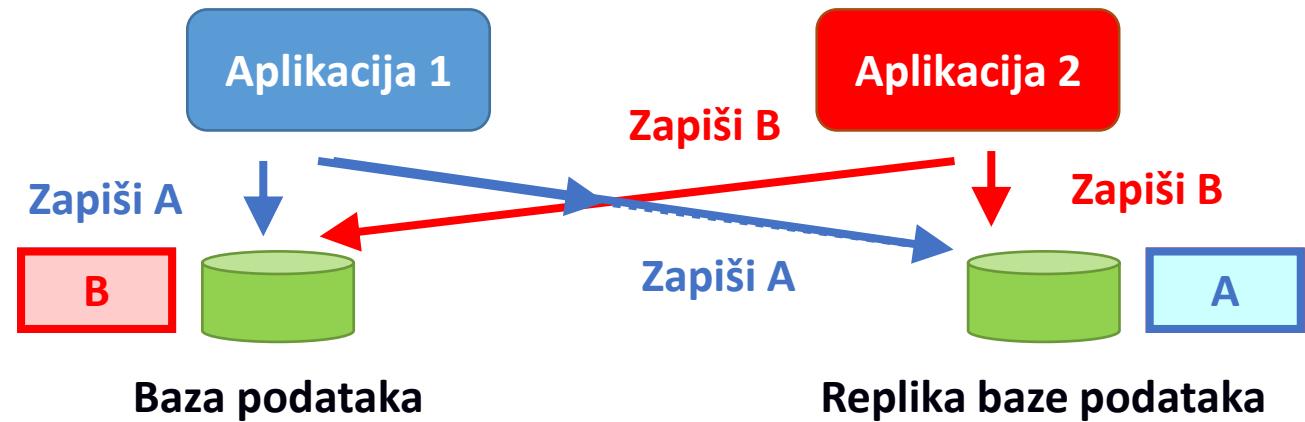
**Koraci algoritma za održavanje vektorskih oznaka:**

- 1) Početne vrijednosti svih vektorskih oznaka su postavljene na 0.
- 2) Za svaku unutarnju akciju procesa  $p$  uvećaj vremensku oznaku na procesu  $p$  pridijeljenu tome procesu za 1, tj.  $V_p[p]+1$ .
- 3) Prije slanja poruke na procesu  $p$  uvećaj oznaku  $V_p[p]$  za 1 i poslanoj poruci pridruži izgrađeni vektor  $V_p$ .
- 4) Nakon primitka poruke od procesa  $p$  na procesu  $k$  uvećaj oznaku  $V_k[k]$  za 1. Za ostale oznake  $i \neq k$  postavi  $V_k[i] = V_p[i]$  ako je  $V_k[i] < V_p[i]$ .



# Primjena sata u raspodijeljenoj okolini

- Uređena razmjena poruka
  - Primjena skalarnih logičkih oznaka vremena
  - Svi procesi na isti način vide redoslijed događaja
- Održavanje konzistentnosti
  - Bez vremenskih oznaka nije moguće odrediti pravilni redoslijed akcija u vremenu



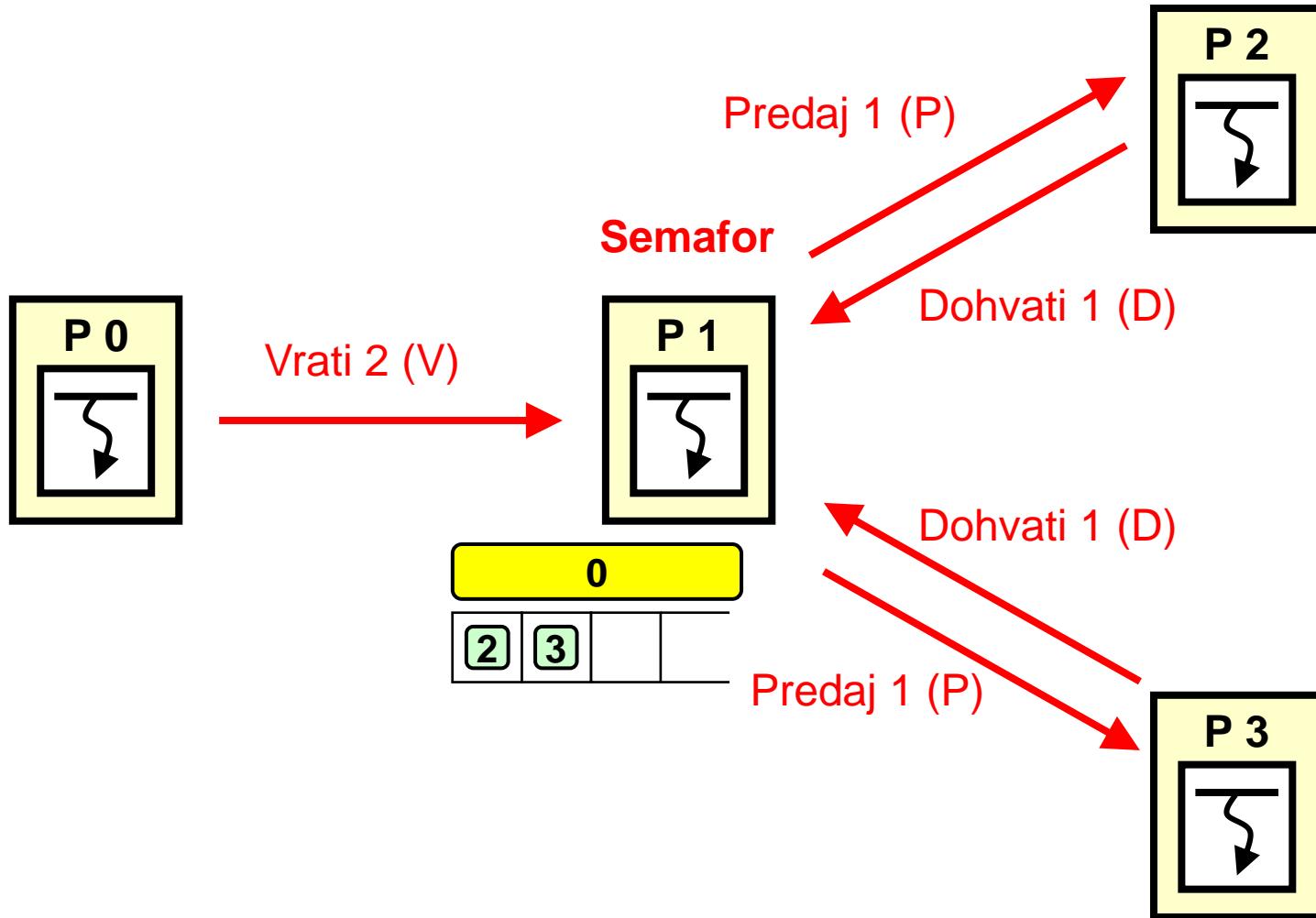
# Sadržaj predavanja

- Potreba za sinkronizacijom procesa
- Primjena sata u jednoprocesorskoj okolini
- Primjena sata raspodijeljenoj okolini
- Sinkronizacija tijeka izvođenja procesa
  - Primjena semafora u raspodijeljenoj okolini
  - Sinkronizacija zasnovana na razmjeni obavijesti
- Međusobno isključivanje procesa

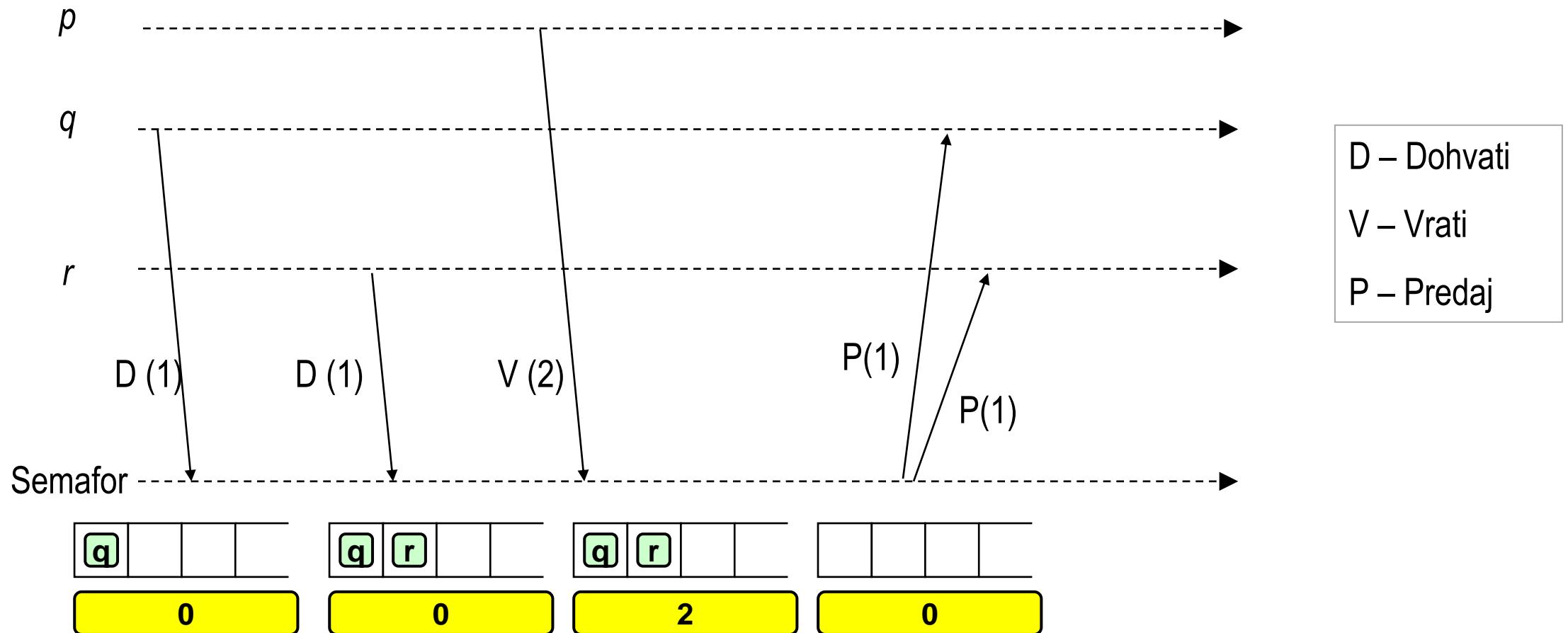
# Primjena semafora u raspodijeljenoj okolini

- **Semafor**
  - Proces koji u spremniku čuva  $N$  znački (*token*)
  - Rep čekanja zasnovan na posluživanju zahtjeva prema redoslijedu prispjeća (*FIFO*)
- **Korisnici**
  - Procesi šalju poruke *zahtjev za dohvat* (*D*)  $n$  znački
  - Ako u spremniku postoji traženi broj znački, proslijeđuje se *potvrda za predaju* (*P*)
  - Ako u spremniku ne postoji traženi broj znački, zahtjev se stavlja u rep čekanja
  - Nakon završetka obrade, procesi vraćaju preuzete značke slanjem poruke *vrati* (*V*)

# Semafor u raspodijeljenoj okolini



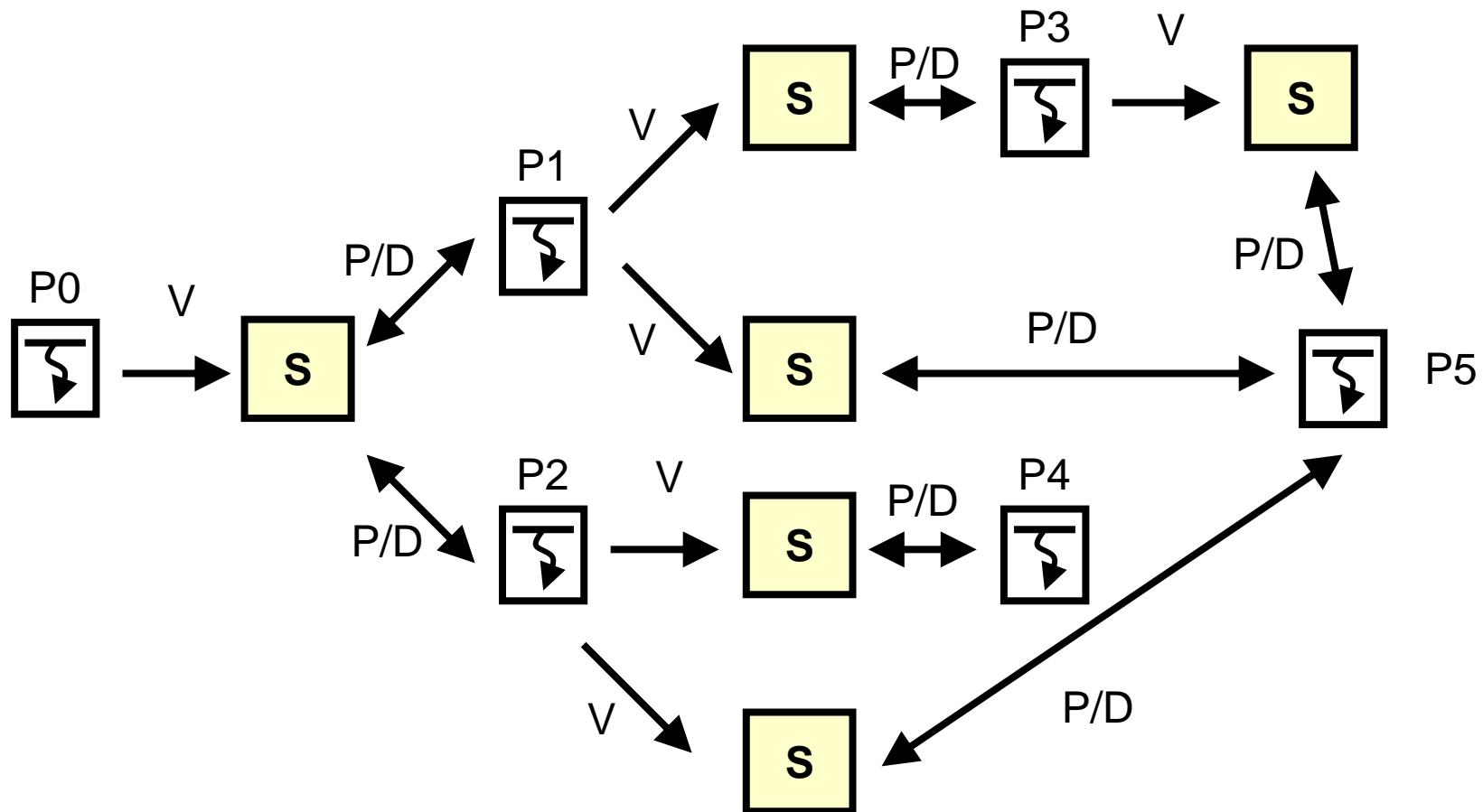
# Primjer sinkronizacije tijeka izvođenja



# Složeni obrasci sinkronizacije

- Semafor je osnovni element za ostvarivanje složenih obrazaca sinkronizacije
- Graf raspodijeljenog tijeka izvođenja procesa
  - Grananje tijeka izvođenja
  - Spajanje tijeka izvođenja
  - Ponavljanje tijeka izvođenja

# Graf raspodijeljenog tijeka izvođenja

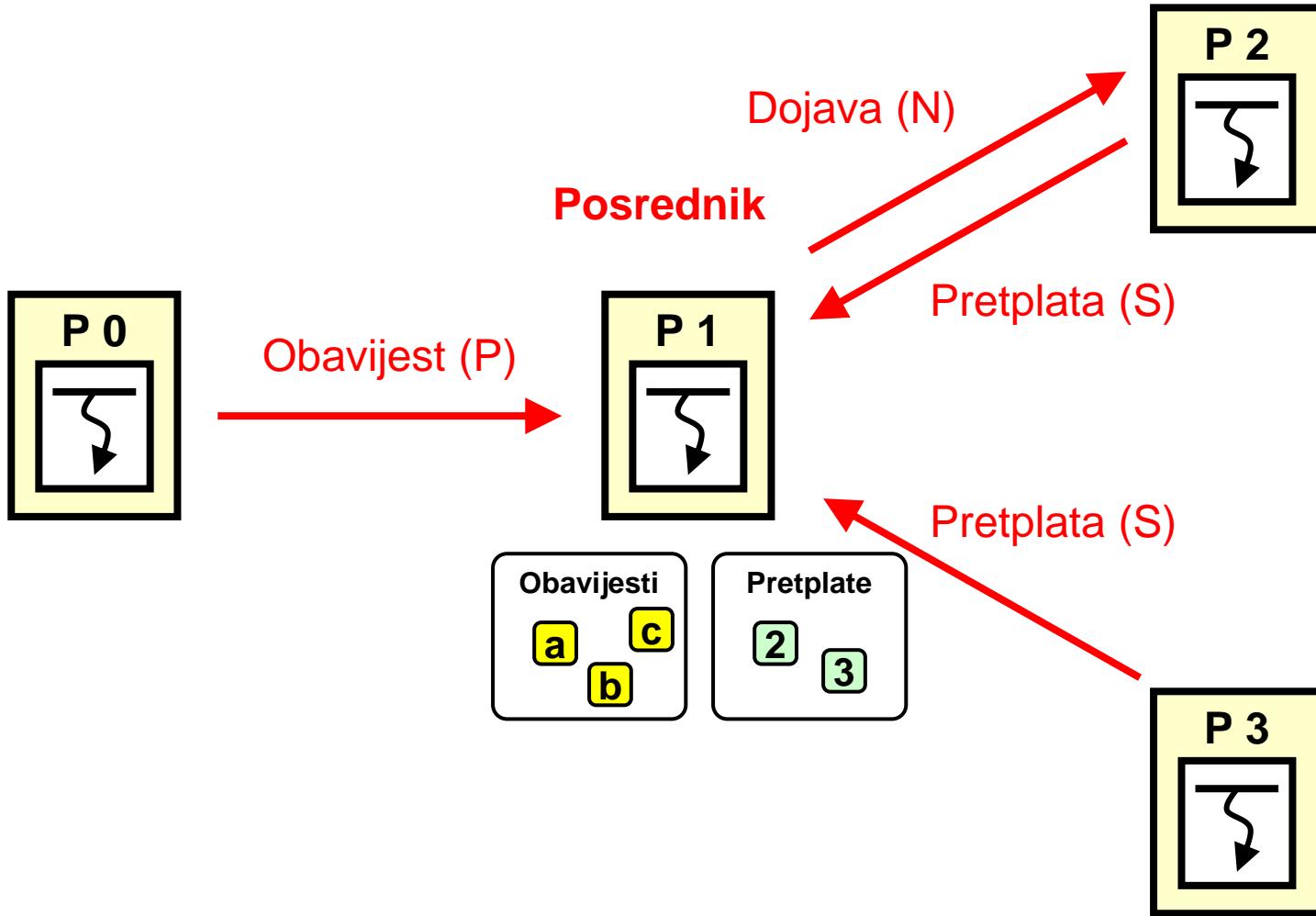


D – Dohvati  
V – Vrati  
P – Predaj

# Sinkronizacija razmjenom obavijesti

- **Posrednik**
  - Sadrži spremnik s obavijestima i spremnik pretplata na obavijesti
  - Ostvaruje postupak usporedbe obavijesti i pretplata prema modelu objavi - pretplati
- **Korisnici**
  - Procesi šalju posredniku pretplate (*S*)
  - Procesi šalju posredniku obavijesti (*P*)
  - Ako posrednik ima aktivnu pretplatu na obavijest, ona se prosljeđuje procesu pretplatniku u poruci dojave (*N*)

# Okolina posrednika obavijesti



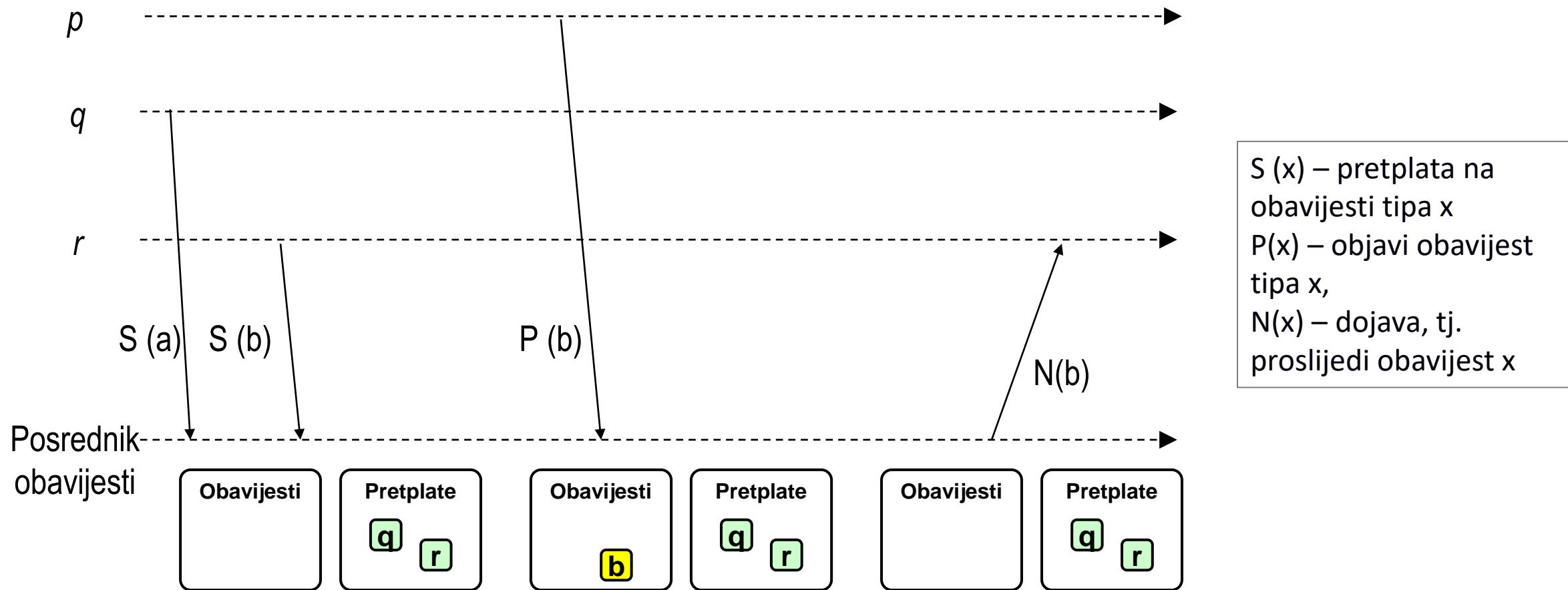
Npr:

Proces P2 se pretplatio (*subscribe*) na obavijesti tipa A

Proces P0 objavljuje (*publish*) obavijest a posredniku

Posrednik proslijeđuje obavijest (*notify*) a procesu P2

# Primjer sinkronizacije razmjenom obavijesti



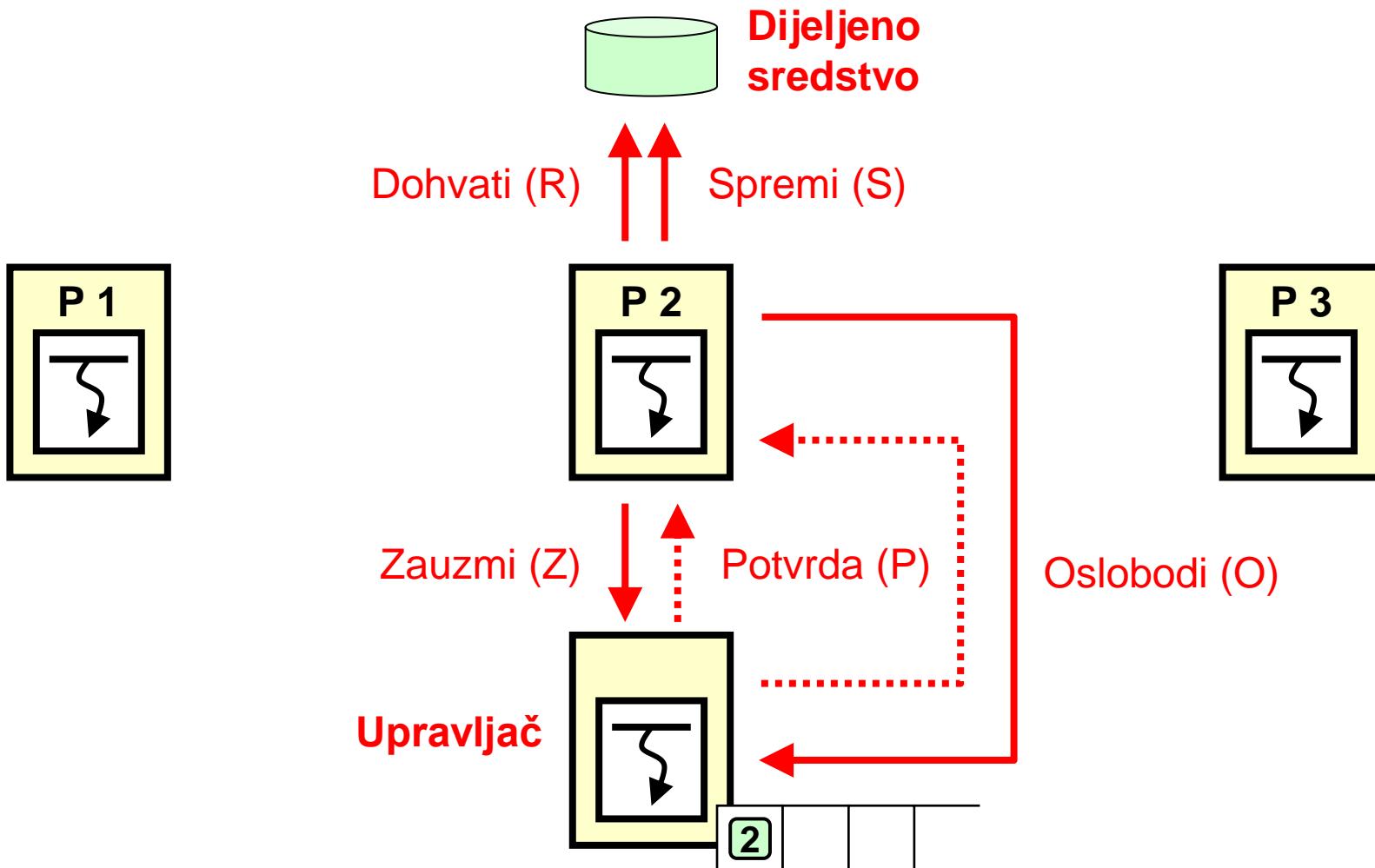
# Sadržaj predavanja

- Potreba za sinkronizacijom procesa
- Primjena sata u jednoprocesorskoj okolini
- Primjena sata raspodijeljenoj okolini
- Sinkronizacija tijeka izvođenja procesa
- Međusobno isključivanje procesa
  - Središnji upravljač s repom čekanja
  - Raspodijeljeno međusobno isključivanje
  - Isključivanje zasnovano na primjeni prstena

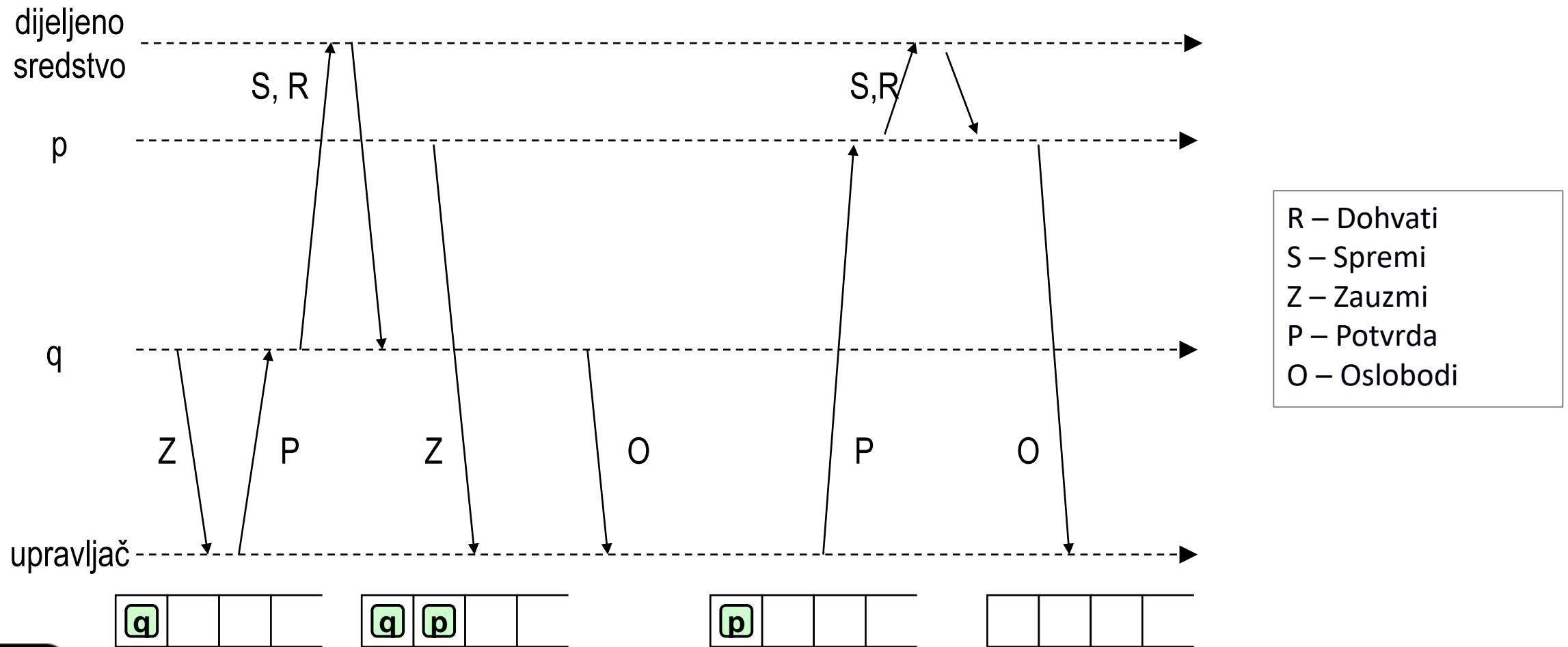
# Međusobno isključivanje procesa

- **Središnji upravljač s repom čekanja**
  - Proces koji čuva stanje repa čekanja
  - Rep čekanja zasnovan na posluživanju zahtjeva prema redoslijedu prispijeća (*FIFO*)
- **Korisnici**
  - Procesi šalju poruke sa *zahtjevom za zauzimanje* (*Z*) tj. pristup sredstvu
  - Procesi ostvaruju pristup sredstvu nakon primitka *poruke potvrde* (*P*), te *dohvaćaju* (*R*) i/ili *spremaju* (*S*) podatke na dijeljeno sredstvo
  - Nakon završetka obrade, procesi otpuštaju zauzeto sredstvo slanjem poruke *oslobodi* (*O*)

# Središnji upravljač s repom čekanja



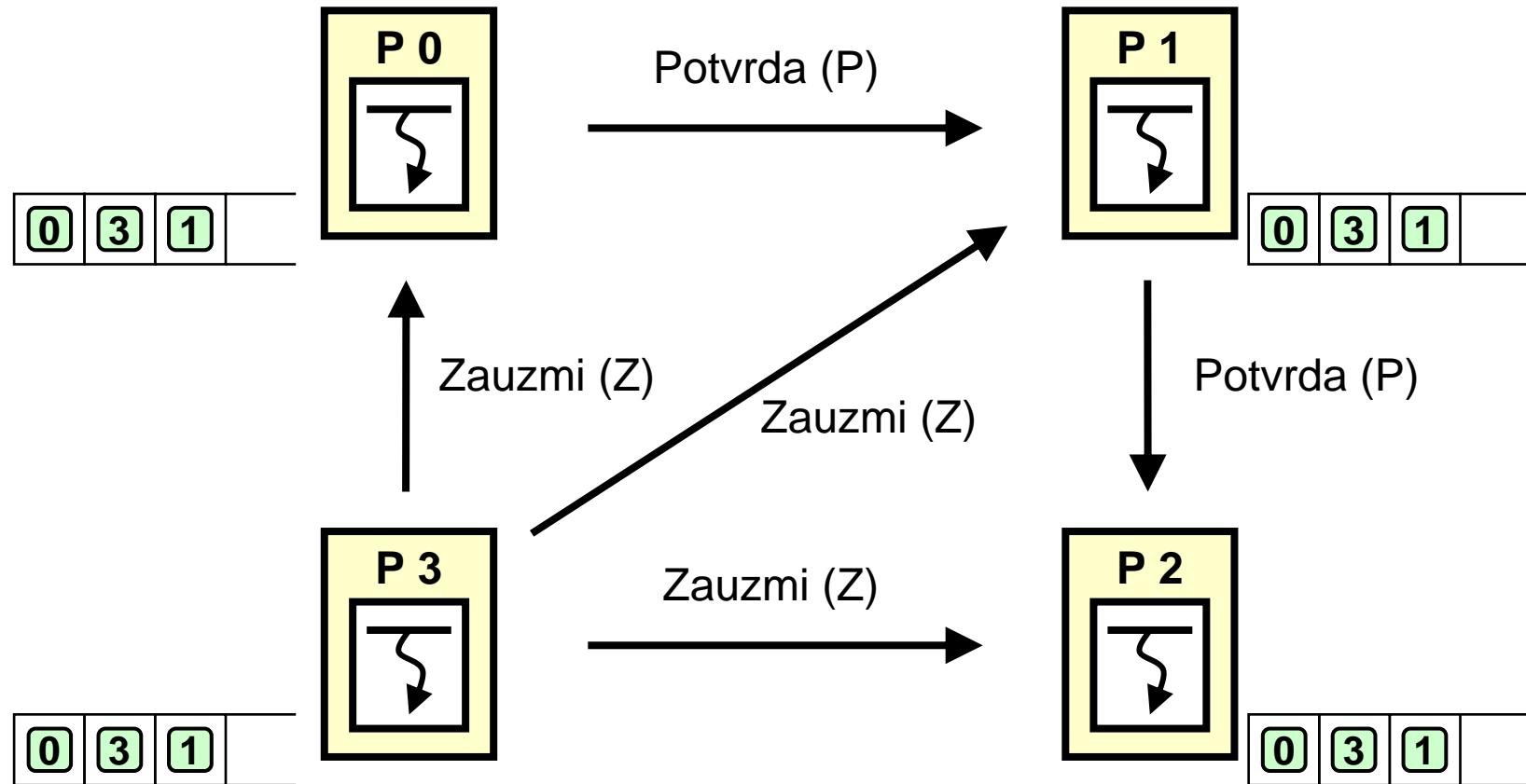
# Isključivanje putem središnjeg upravljača



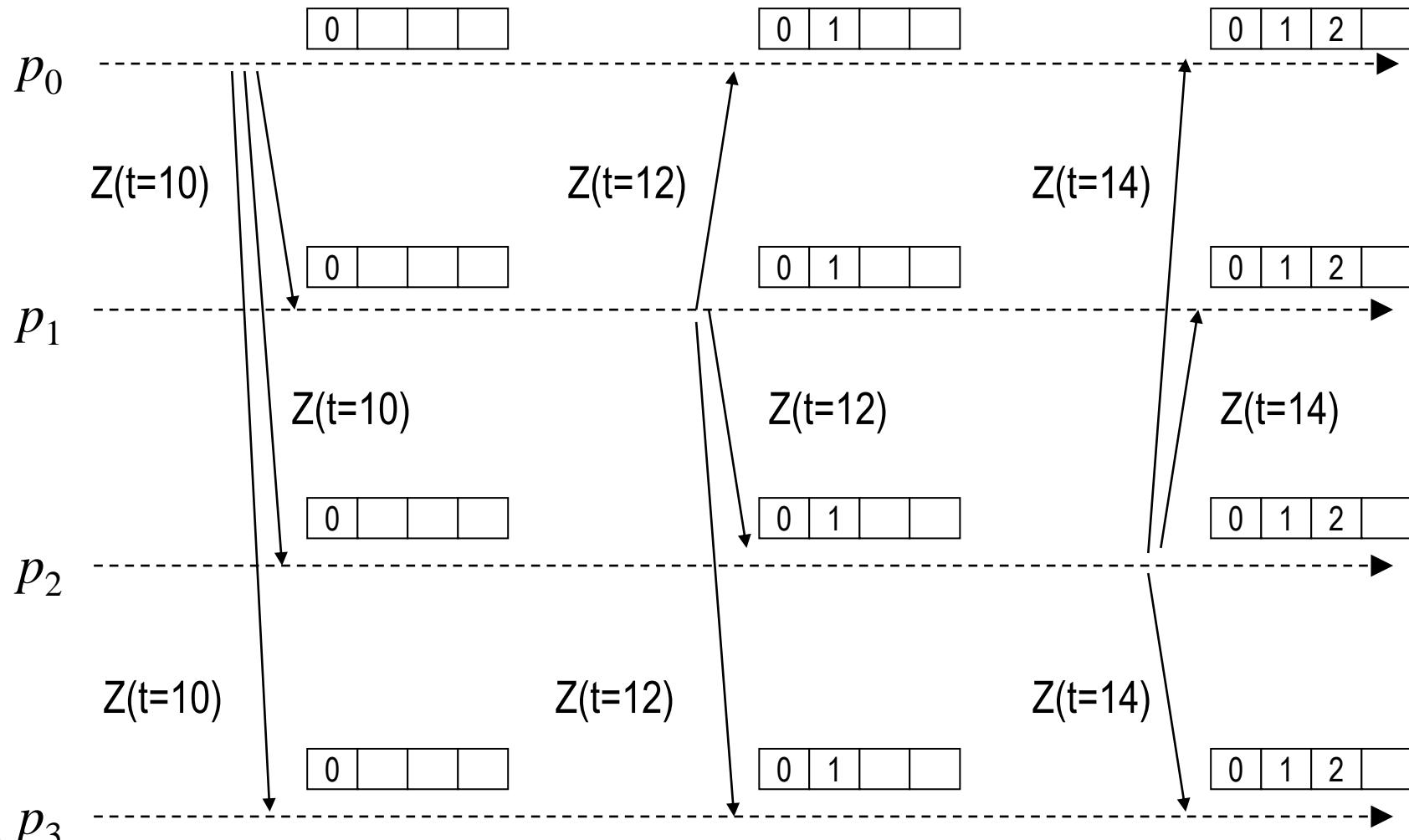
# Raspodijeljeno međusobno isključivanje

- **Raspodijeljeni rep čekanja**
  - Svaki proces ima lokalni rep čekanja
  - Procesi razmjenjuju informacije potrebne za usklađivanje stanja svih repova čekanja u sustavu
- **Prepostavke**
  - Svaki proces ima lokalni satni mehanizam koji je usklađen s ostalim procesima
  - Svaki zahtjev za pristup sredstvu uključuje oznaku trenutka u kojem je proces uputio zahtjev
  - Procesi ostvaruju pristup u skladu s vremenskim oznakama upućivanja zahtjeva

# Elementi raspodijeljenog isključivanja



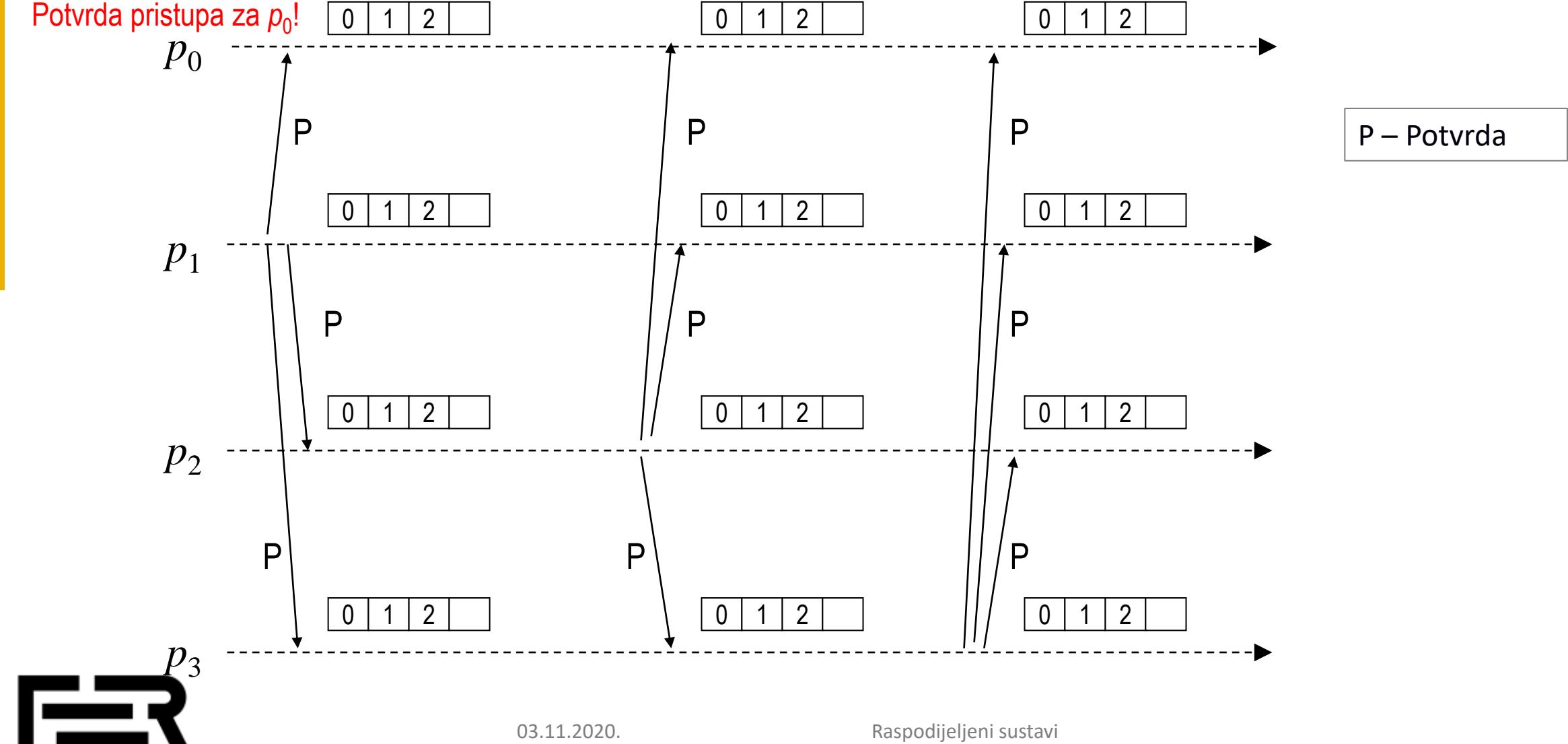
# Raspodijeljeno isključivanje (1/4)



$Z(t=x)$  – Zauzmi,  
vremenska oznaka x

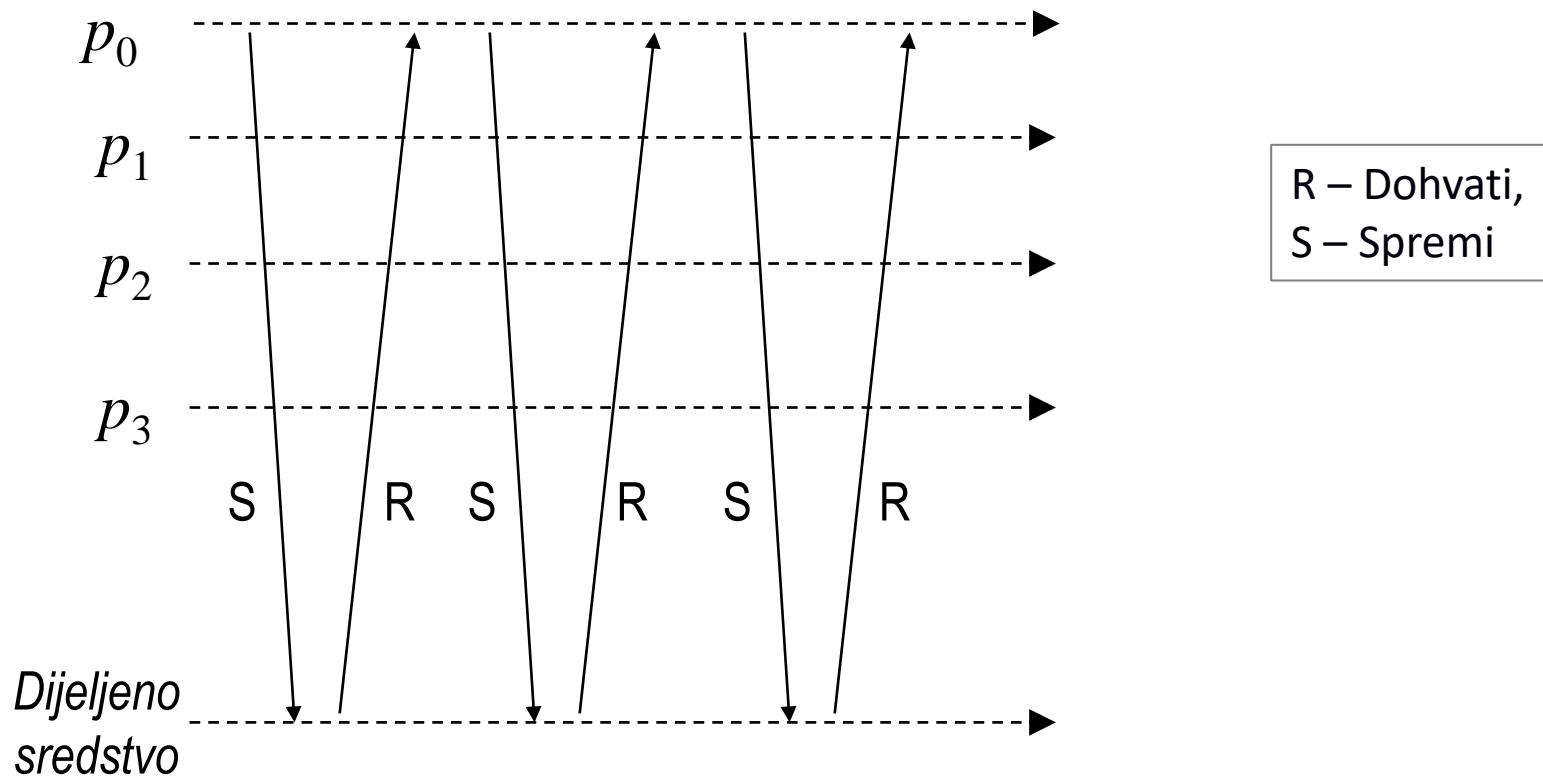
# Raspodijeljeno isključivanje (2/4)

Potvrda pristupa za  $p_0$ !



# Raspodijeljeno isključivanje (3/4)

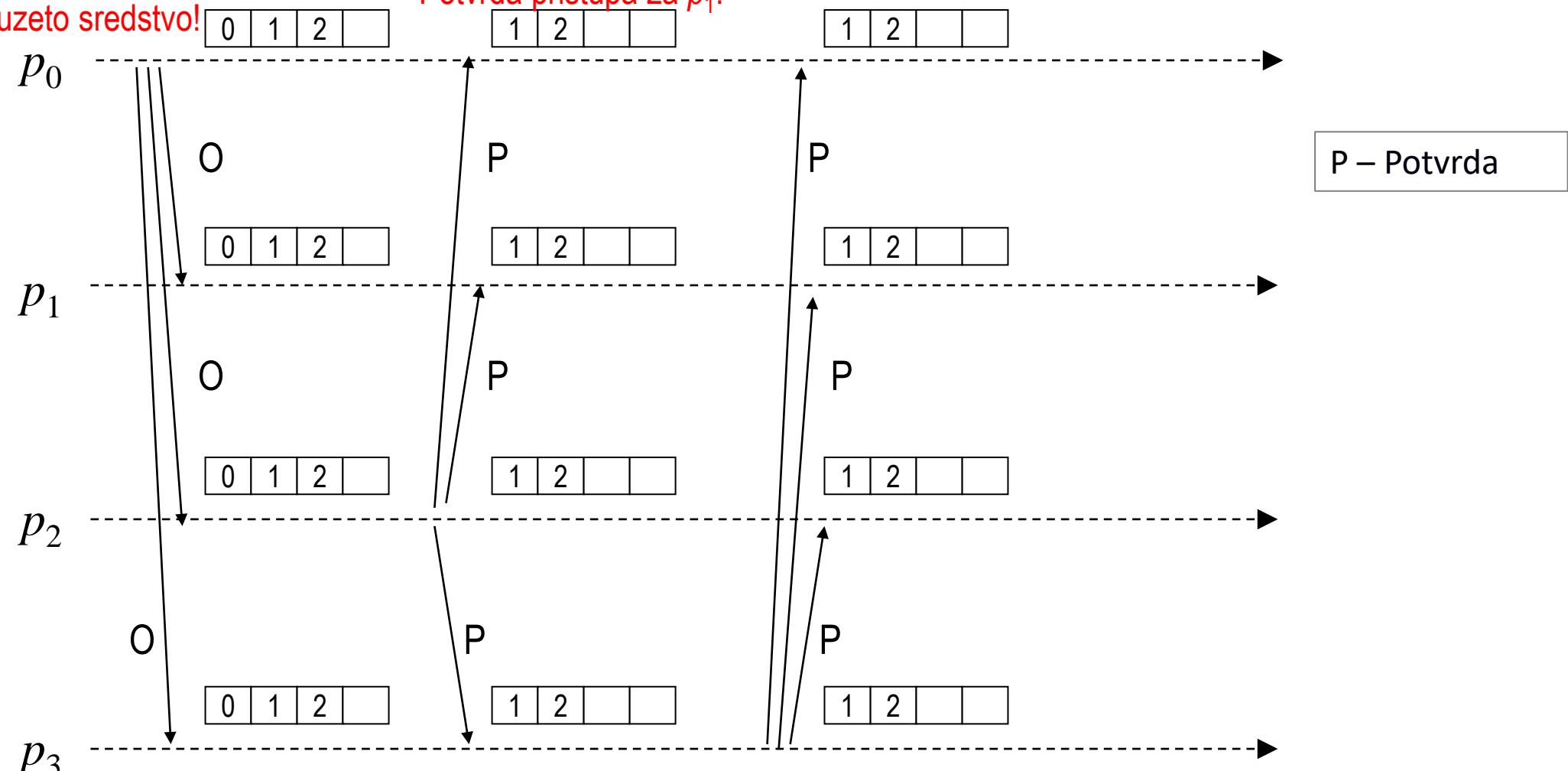
Pristup  
dijeljenom  
sredstvu!



# Raspodijeljeno isključivanje (4/4)

Otpusti zauzeto sredstvo!

Potvrda pristupa za  $p_1$ !



# Međusobno isključivanje primjenom prstena

- **Struktura prstena procesa**
  - Procesi su povezani u logičku mrežu zasnovanu na prstenu
  - Primjenjuju se identifikatori procesa za formiranje prstena
  - Duž prstena ostvaruje se razmjena jedne značke
- **Pristup dijeljenom sredstvu**
  - Pristup ima samo proces koji u određenom trenutku ima značku
  - Nakon završetka pristupa, proces proslijeđuje značku susjednom procesu u prstenu

# Akcije procesa u prstenu (1)

## Proces $n$ prima značku

- 1) Čitanje podataka iz spremnika
- 2) Pisanje podataka u spremnik
- 3) Prosljeđivanje značke procesu (  $n-1$  )

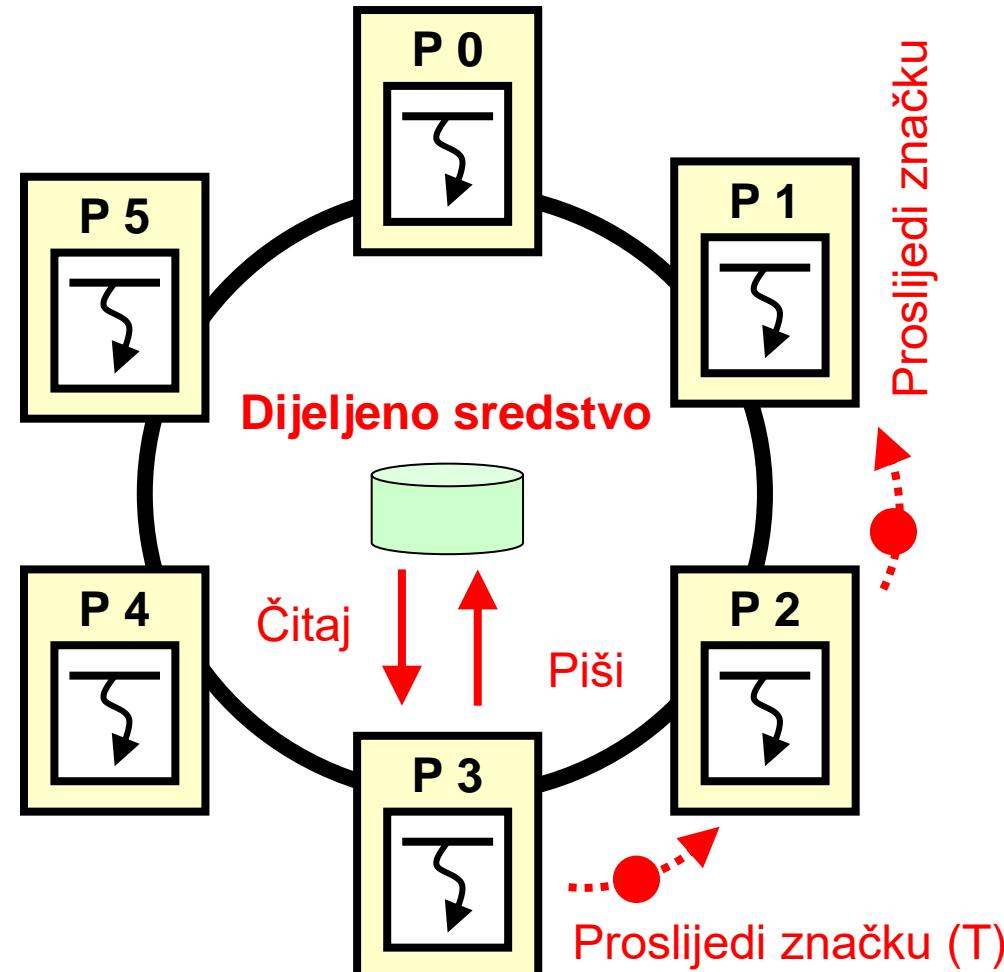
## Proces ( $n-1$ ) prima značku

- 4) Proces ne zahtjeva pristup spremniku
- 5) Prosljeđivanje značke procesu (  $n-2$  )

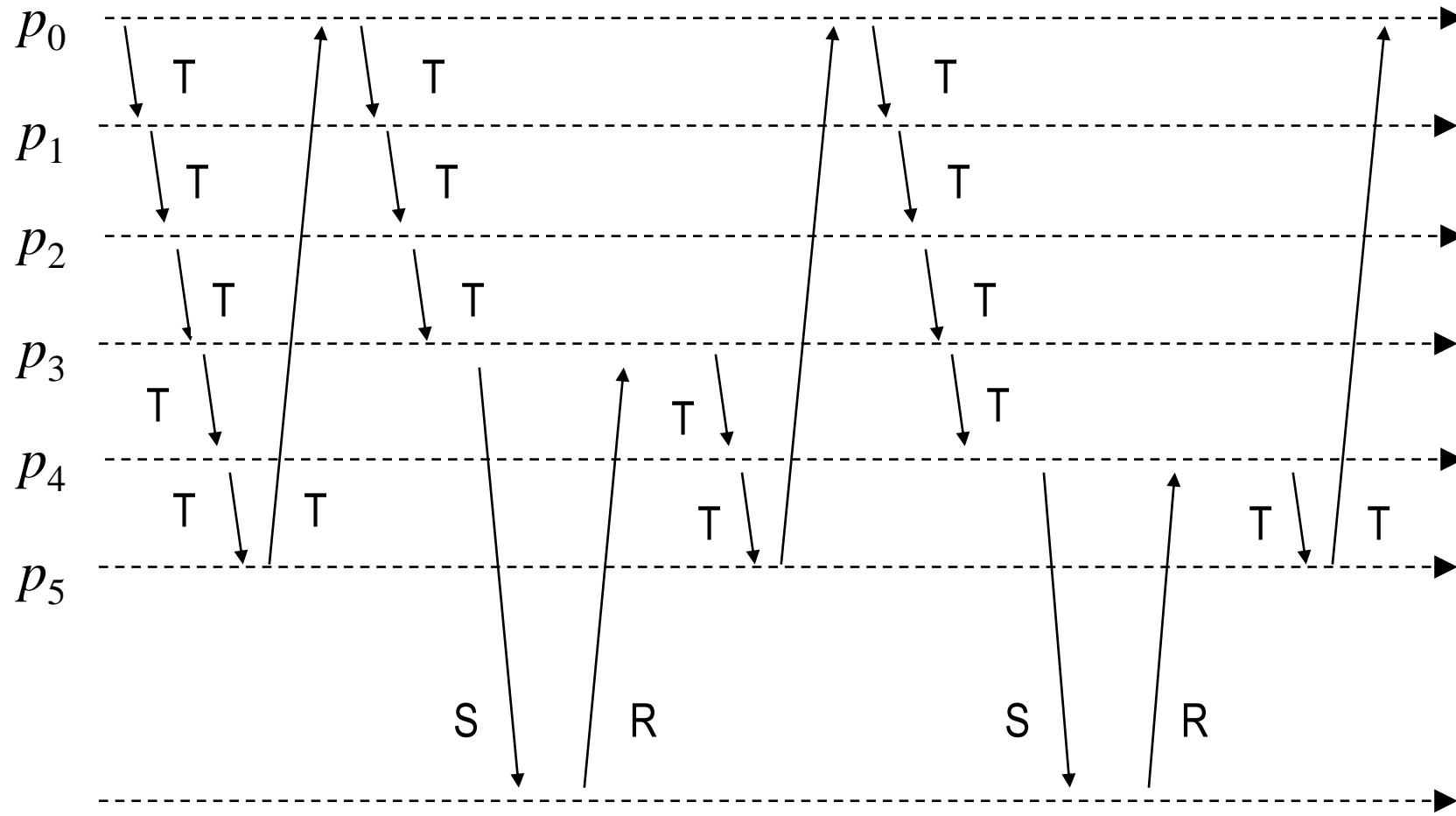
## Proces ( $n-2$ ) prima značku

- 6) ...

# Akcije procesa u prstenu (2)



# Akcije procesa u prstenu (3)



T – Prijenos tokena,  
S – Spremi,  
R – Dohvati

# Primjeri sustava za sinkronizaciju

## Usluga ZooKeeper

- Usluga za pouzdanu koordinaciju tijeka izvođenja skupa procesa u raspodijeljenoj okolini
- Dodatne informacije: <http://zookeeper.apache.org>

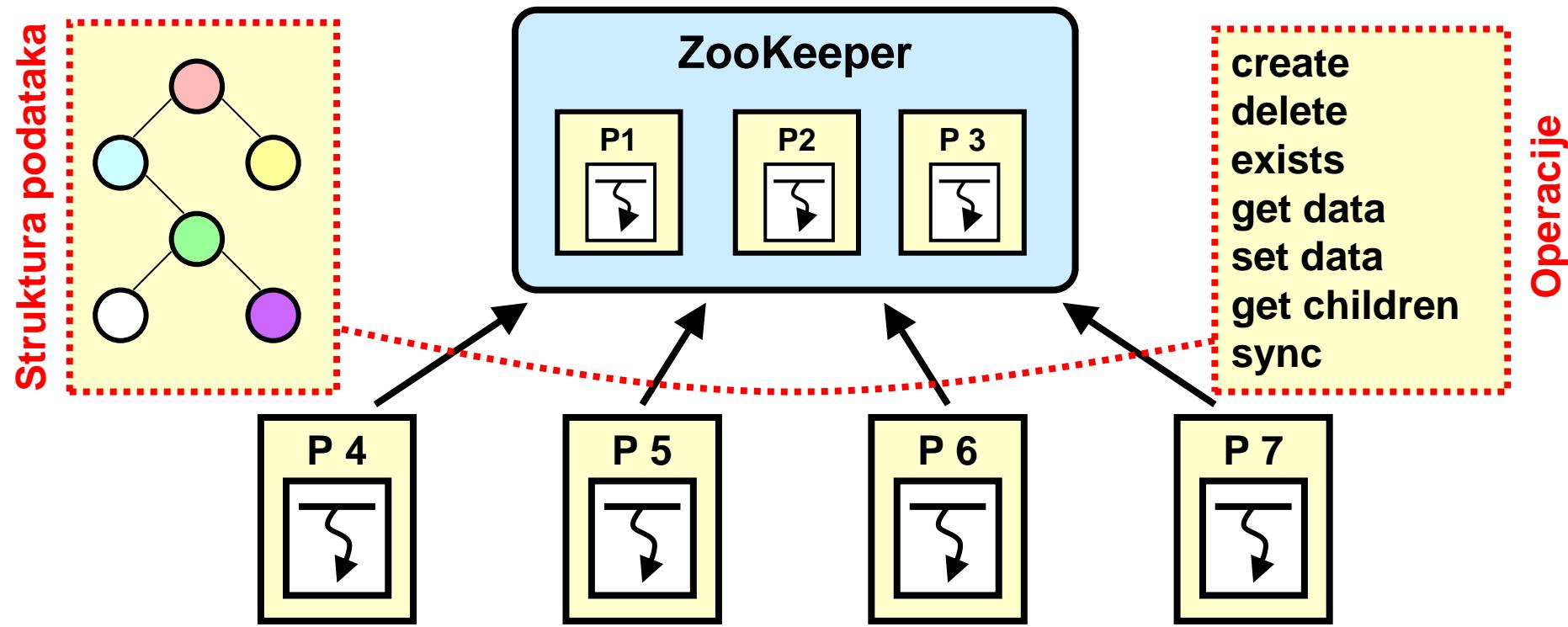
## Okružje Hadoop

- Programsko okružje za provođenje paralelne obrade velike količine podataka (*Big Data*)
- Postoji potreba za sinkronizacijom procesa *map* i *reduce*
- Dodatne informacije: <http://hadoop.apache.org>

# Usluga ZooKeeper (1)

Usluga opće namjene za koordiniranje skupa procesa u raspodijeljenoj okolini

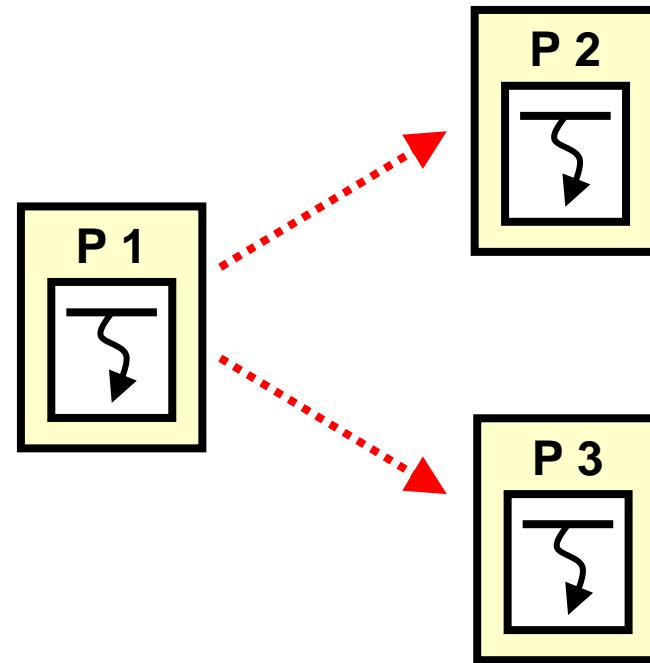
Imenovanje, sinkronizacija, upravljanje grupama, repovi, donošenje odluka, zaključavanje sredstava



# Usluga ZooKeeper (2)

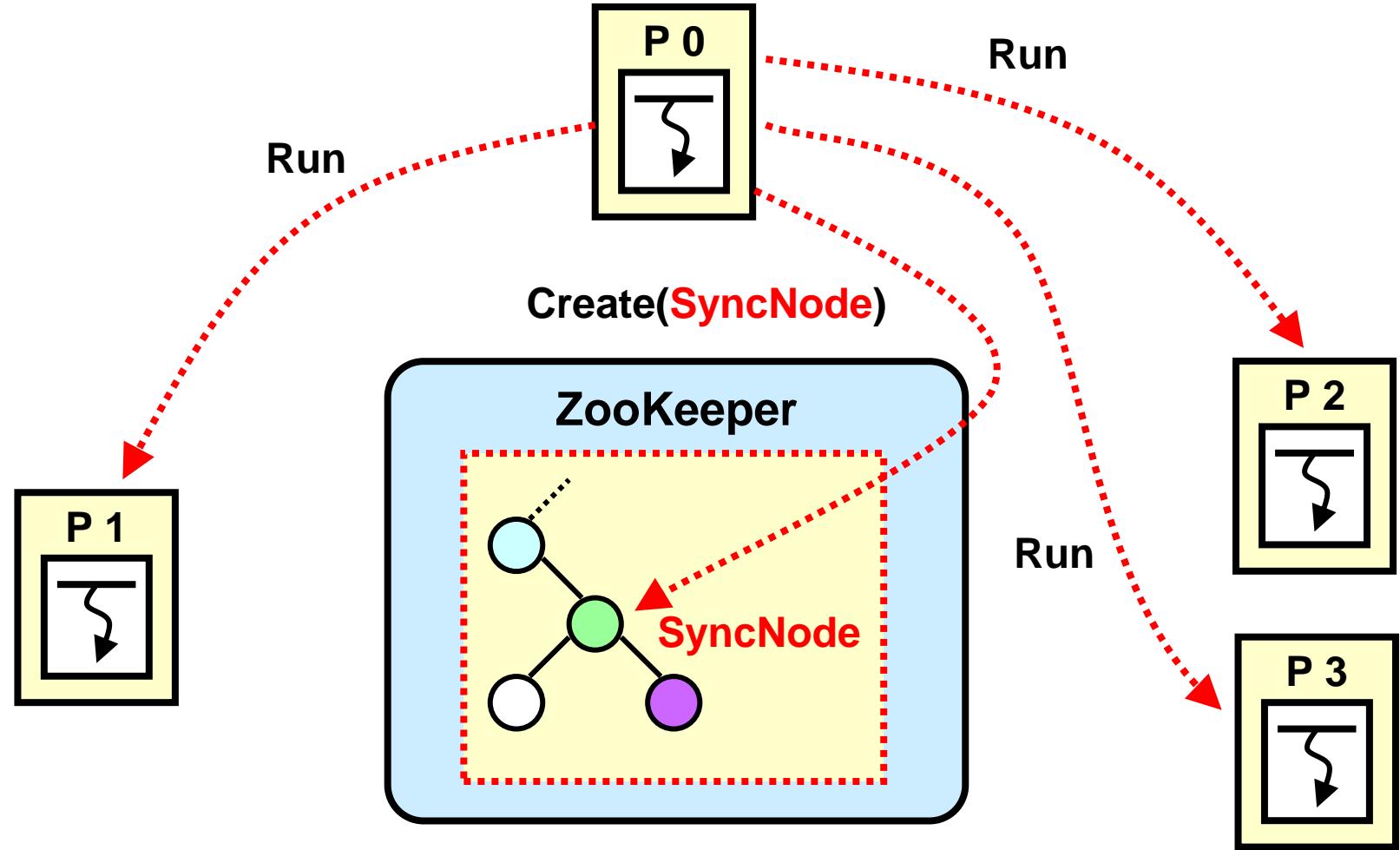
## Primjer: Sinkronizacija procesa

Procesi P2 i P3 započinju s izvođenjem tek nakon što je proces P1 završio s izvođenjem



# Usluga ZooKeeper (3)

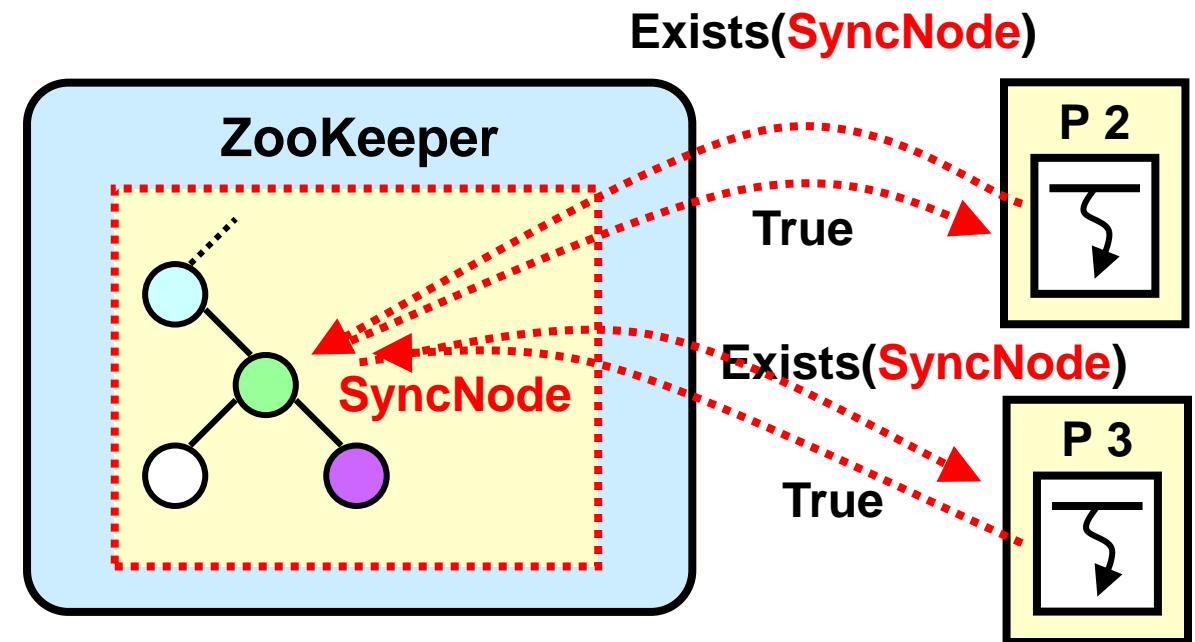
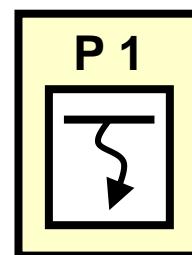
P0 je upravljački proces koji stvara sinkronizacijski čvor **SyncNode**



# Usluga ZooKeeper (4)

Procesi P2 i P3 ispituju postoji li sinkronizacijski čvor **SyncNode**

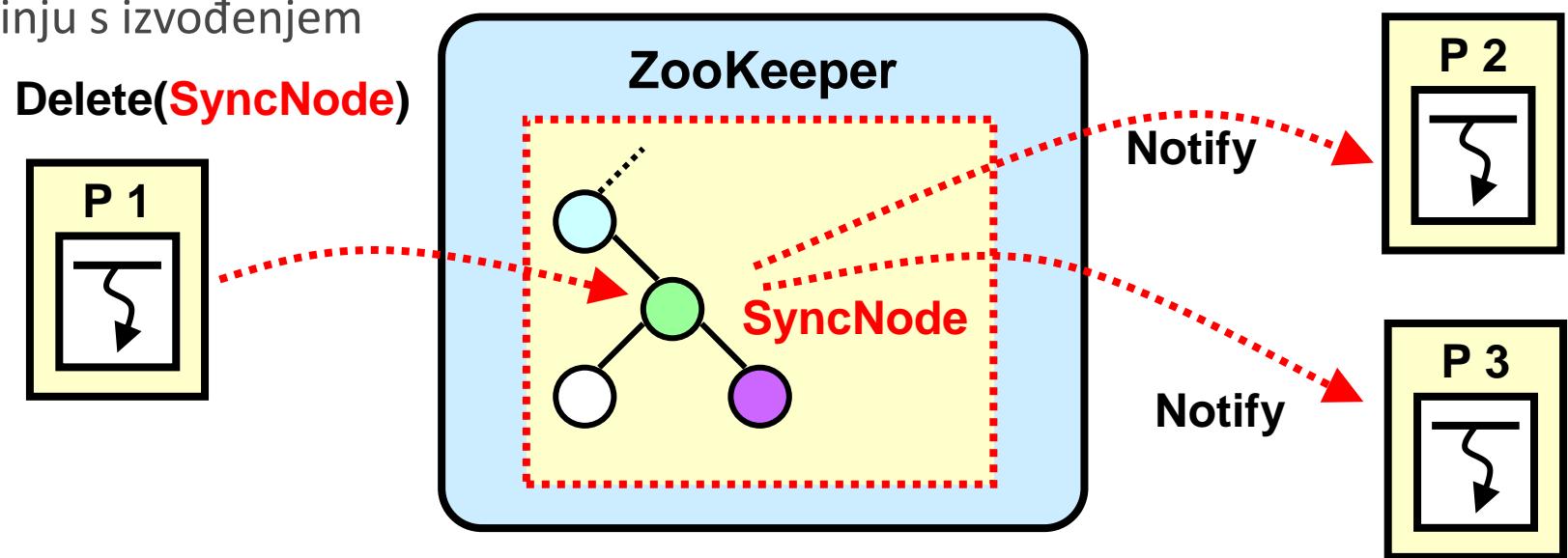
- Ako čvor postoji, čekaju na dojavu o brisanju čvora
- jer kada se čvor izbriše, oni započinju izvođenje



# Usluga ZooKeeper (5)

## Proces P1 završava s izvođenjem

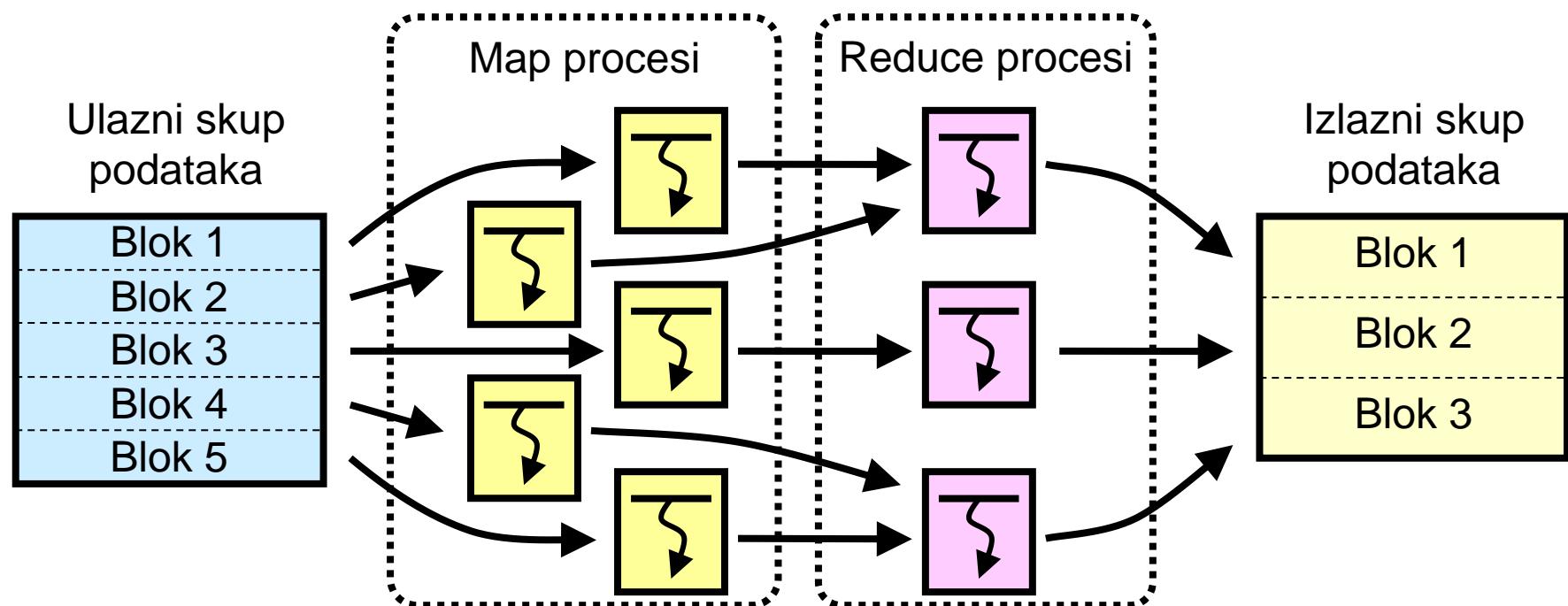
- Briše sinkronizacijski čvor **SyncNode**
- Usluga ZooKeeper dojavljuje brisanje čvora i stoga
- procesi **P2** i **P3** započinju s izvođenjem



# Okružje Hadoop

## MapReduce: analiza i obrada velikih skupova podataka

- Ulazni skup podataka dijeli se na blokove koje paralelno obrađuju nezavisni procesi
- Rezultati obrade grupiraju se u odredišni skup podataka



# Literatura

- S. Tanenbaum, M. van Steen: “**Distributed Systems: Principles and Paradigms**” second edition, Prentice Hall, 2007. (Poglavlje: *Synchronization*)
- H. Attya, J. Welch: “**Distributed Computing: Fundamentals, Simulations, and Advanced Topics**”, Wiley, 2004. (Poglavlje: *Causality and Time*)
- N. A. Lynch: “**Distributed Algorithms**”, Morgan Kaufmann, 1997. (Poglavlje: *Logical Time*)

# Dodatne informacije

- Kolegij na FER-u
  - Raspodijeljena obrada velike količine podataka, 2. semestar
  - <http://www.fer.unizg.hr/predmet/rovkp>
  - Sadržaj kolegija
    - Obrada velike količine podataka: MapReduce
    - Obrada nestrukturiranog teksta
    - Obrada toka podataka



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**7. Mikrousluge (mikroservisi)**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

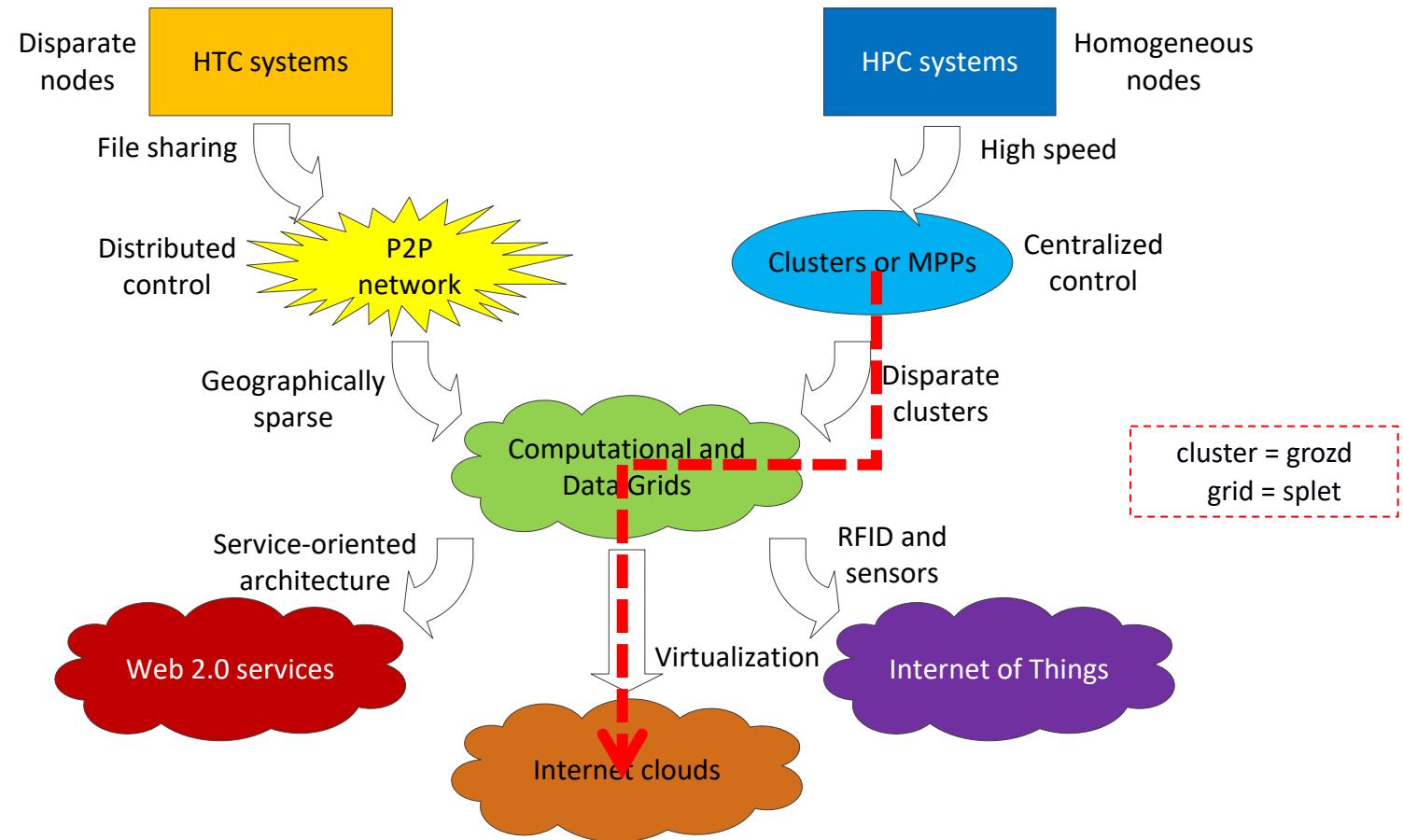
- Uvod
- Grozd računala
- Splet računala
- Računarstvo u oblaku
- Mikrousluge
- Trendovi i smjernice

# Aplikacije koje zahtijevaju obradu velikog broja zadataka

| Domena                  | Primjena                                                                                                                                                                     |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Znanstvene aplikacije   | Znanstvene simulacije, analiza genoma, predviđanje potresa, modeliranje globalnog zagrijavanja, vremenska prognoza, itd.                                                     |
| Industrijske aplikacije | Telekomunikacije, isporuka sadržaja, e-poslovanje, bankarstvo, burze, kontrola zračnog prometa, isporuka električne energije, udaljeno učenje, zdravstvo, telemedicina, itd. |
| Internetske aplikacije  | Tražilice, podatkovni centri, nadgledanje prometa, računalna sigurnost, digitalizacija javne uprave, društvene mreže, itd.                                                   |
| Kritične aplikacije     | Upravljanje kriznim situacijama, kontrola i zapovijedanje u vojnim zadacima                                                                                                  |

# Evolucija raspodijeljenog i paralelnog računarstva

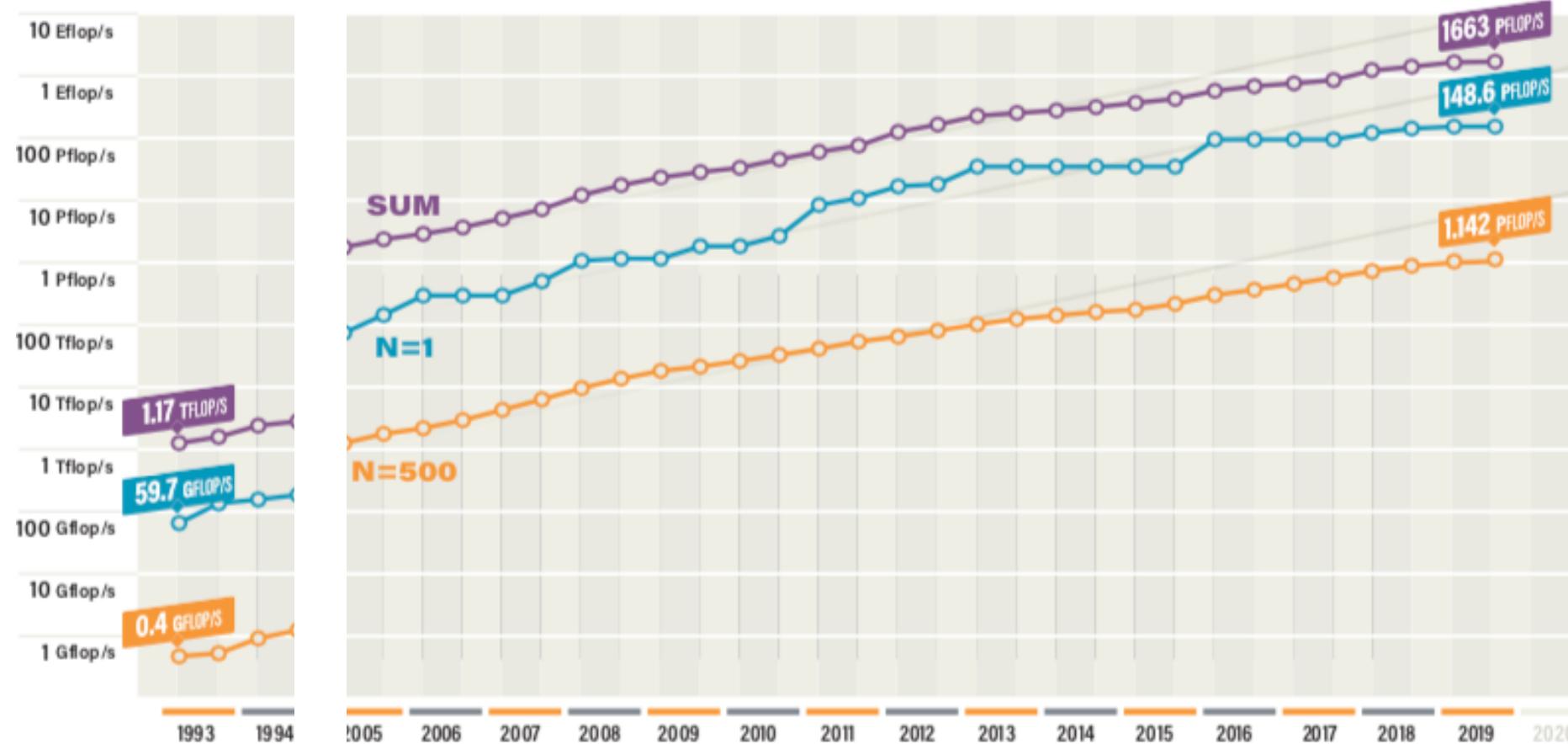
- HTC (High-Throughput Computing)
  - kontinuirana obrada velike količine raznovrsnih i nepovezanih zadataka
- HPC (High-Performance Computing)
  - paralelna obrada velike količine sličnih i povezanih zadataka u kratkom periodu
- MPP (Massively Parallel Processors )
  - centralizirana superračunala



# Usporedba grozdova, spletova i računarstva u oblaku

|             | Grozd ( <i>Cluster</i> )                               | Splet ( <i>Grid</i> )                                            | Oblak ( <i>Cloud</i> )                                                             |
|-------------|--------------------------------------------------------|------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Arhitektura | Skup računala povezanih brzom lokalnom mrežom          | Skup udaljenih računala (ili grozdova) povezanih putem Interneta | Virtualizirani grozd računala koja se nalaze u jednom ili više podatkovnih centara |
| Resursi     | Identična ili vrlo slična računala                     | Raznolika računala                                               | Identična ili vrlo slična računala                                                 |
| Aplikacije  | HPC, tražilice                                         | Raspodijeljeno rješavanje problema                               | Uslužno računarstvo ( <i>utility computing</i> ), pohrana podataka                 |
| Primjeri    | Google search engine,<br>Cray XK7,<br>BlueGene/Q, itd. | Folding@home, BOINC,<br>SETI@home, itd.                          | Microsoft Azure, Amazon EC2,<br>Google App Engine, itd.                            |

# Evolucija performansi superračunala



# Evolucija arhitekture superračunala



SIMD – Single Instruction Multiple Data

MPP – Massively Parallel Processors

SMP – Symmetric multiprocessing

Constellations – više procesora na čvoru nego čvorova

# Sadržaj predavanja

- Uvod
- Grozd računala
  - Evolucija superračunala
  - Definicija i vrste grozda računala
  - Primjeri grozda računala
- Splet računala
- Računarstvo u oblaku
- Mikrousluge

# Definicija grozda računala (*computer cluster*)

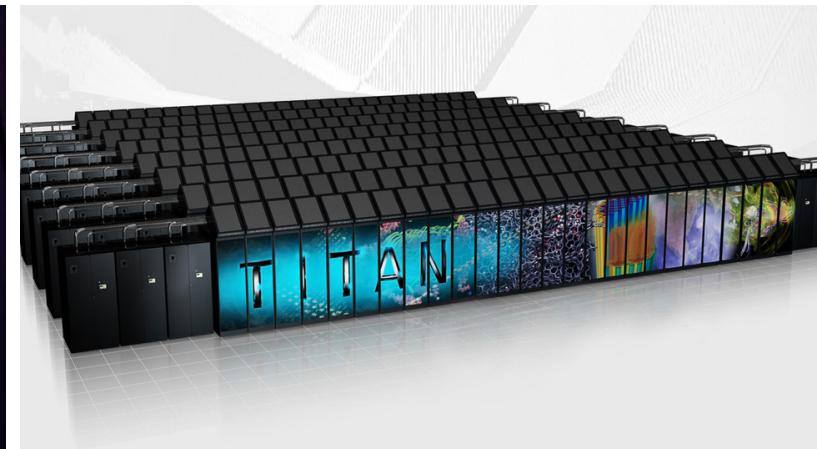
- Skup samostalnih računala
- Međusobno su povezana brzom lokalnom mrežom
- Rade kooperativno
- Čine jedan jedinstveni (integrirani) računalni resurs



JUQUEEN, Njemačka



Tianhe-2 (MilkyWay-2), Kina



Cray

# Vrste i primjene grozda računala

- Računarski grozd (*compute cluster*)
  - Upotrebljava se za kolektivno i kooperativno izvođenje jednog zahtjevnog posla (npr. simulacija vremenskih uvjeta)
  - Vrlo malo korištenje I/O operacija
- Grozd visoke raspoloživosti (*high-availability cluster*)
  - Redundantna računala preuzimaju posao uslijed neispravnosti drugih
  - Ne postoji jedinstvene točke ispada grozda
  - Koristi se za raspodijeljene baze podataka i ostale poslovne aplikacije
- Grozd za raspodjelu opterećenja (*load-balancing cluster*)
  - Raspodjelom opterećenja postiže visoke performanse
  - Međuoprema vrši raspodjelu opterećenja i migraciju procesa između računala

# Najsnažniji grozdovi u svijetu i Europi (kraj 2019.)

| Rank | System                                                                                                                                                                                     | Cores      | Rmax<br>(TFlop/s) | Rpeak<br>(TFlop/s) | Power<br>(kW) |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------|-------------------|--------------------|---------------|
| 1    | <b>Summit</b> - IBM Power System AC922, IBM POWER9<br>22C 3.07GHz, NVIDIA Volta GV100, Dual-rail<br>Mellanox EDR Infiniband , IBM<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 2,414,592  | 148,600.0         | 200,794.9          | 10,096        |
| 2    | <b>Sierra</b> - IBM Power System AC922, IBM POWER9<br>22C 3.1GHz, NVIDIA Volta GV100, Dual-rail<br>Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox<br>DOE/NNSA/LLNL<br>United States     | 1,572,480  | 94,640.0          | 125,712.0          | 7,438         |
| 3    | <b>Sunway TaihuLight</b> - Sunway MPP, Sunway<br>SW26010 260C 1.45GHz, Sunway , NRCPC<br>National Supercomputing Center in Wuxi<br>China                                                   | 10,649,600 | 93,014.6          | 125,435.9          | 15,371        |

# Grozdovi u HR

- Isabella na Srcu
  - <https://www.srce.unizg.hr/isabella/>
- Superačunalo Bura na Sveučilištu u Rijeci
  - HPC
  - pokrenuto 2016.
  - <https://www.hpc.uniri.hr/>
- HR-ZOO – Hrvatski znanstveni i obrazovni oblak
  - dovršetak projekta sredina 2021.
  - <https://www.srce.unizg.hr/hr-zoo>



# Isabella

- 135 računala (270 CPU procesora)
- 3.100 CPU procesorskih jezgri
- 16 TB radne memorije
- 756 TB dijeljenog podatkovnog prostora
- lokalna mreža visokog stupnja propusnosti
  - 2 x HP ProCurve 2848 1 Gb/s Ethernet preklopnik (96 pristupa)
  - 1 x Voltaire Grid Director QDR (40 Gb/s) Infiniband preklopnik (36 pristupa)
  - 6 x Mellanox SX6025 (56 Gb/s) Infiniband preklopnik (36 pristupa)
- temelji se na sustavima:
  - 28 x HP ProLiant SL250s
  - 12 x HP ProLiant SL250s (12)
  - 8 x HP ProLiant SL230s Gen8
  - 8 x HP ProLiant SL230s Gen8
  - 76 x Lenovo NeXtScale nx360 M5
  - 3 x Dell EMC PowerEdge C4140



# Sadržaj predavanja

- Uvod
- Grozd računala
- Splet računala
  - Definicija i vrste spleta računala
  - Primjeri spleta računala
- Računarstvo u oblaku
- Mikrousluge
- Trendovi i smjernice

# Splet računala (grid)

- Ideja: Ukoliko nam je za izvođenje posla potreban slabo povezan grozd (samo neka računala često komuniciraju pri izvođenju posla) posao možemo raspodijeliti među geografski udaljenim grozdovima – ***virtualno superračunalo***
- Definicija spleta računala
  - Skup međusobno udaljenih heterogenih računalnih sredstava
  - Međusobno su (slabo) povezana Internetom
  - Rade na zajedničkom zadatku radi postizanja zajedničkog cilja
- Računalna sredstava su najčešće pod različitim administrativnim domenama (koje pripadaju različitim organizacijama)

# Vrste i primjene spleta računala

| Vrsta                                  | Primjena                                                                                            | Primjeri                                                                                                   |
|----------------------------------------|-----------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|
| Računarski i podatkovni splet računala | Spremanje i obrada velike količine podataka, znanstveni eksperimenti, modeliranje, simulacije, itd. | TeraGrid (SAD), EGEE (EU), DataGrid (EU), ChinaGrid (Kina), NAS (SAD - NASA), LCG (Švicarska - Cern), itd. |
| Informacijski splet računala           | Semantički web i upravljanje znanjem                                                                | Semantic Grid, Ontology Platform, D4Science, Information Power Grid (SAD - NASA), itd.                     |
| Poslovni splet računala                | Poslovne web aplikacije, dubinska analiza velike količine poslovnih podataka                        | BElngGrid (EU), HP eSpeak, Oracle Grid Engine, itd.                                                        |
| Volonterski splet računala             | Znanstvene analize                                                                                  | SETI@Home, Einstein@Home, BOINC, Folding@Home, itd.                                                        |

# Primjeri spletova računala

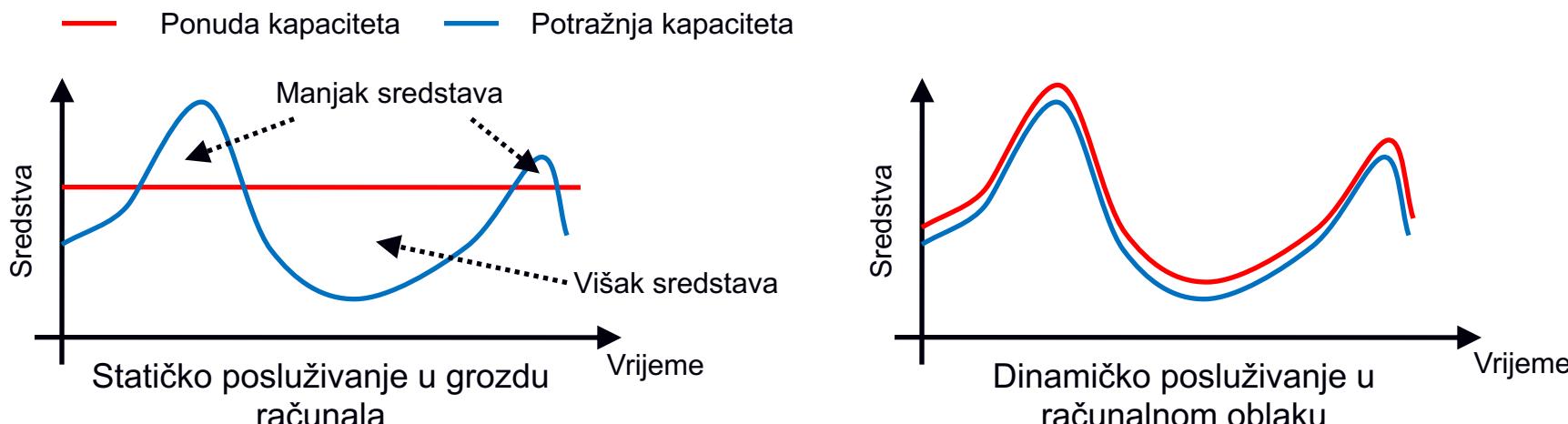
- [Bitcoin Network](#) - 80.000.000 PFLOPS (2019.)
- [BOINC](#) – 22 PFLOPS (2018.)
- [Folding@home](#) – 101 PFLOPS (2016.)
- [Einstein@Home](#) – 3,489 PFLOPS (2018.)
- [SETI@Home](#) – 0,890 PFLOPS (2018.)
- [MilkyWay@Home](#) – 0,941 PFLOPS (2018.)
- [GIMPS](#) – 0,558 PFLOPS (2019.)

# Sadržaj predavanja

- Uvod
- Grozd računala
- Splet računala
- **Računarstvo u oblaku**
  - Definicija i vrste računalnog oblaka
  - Modeli usluga i stranke u računalnom oblaku
  - Prednosti i nedostaci računalnog oblaka
  - Primjeri računalnog oblaka
  - Kontejneri
- Mikrousluge
- Trendovi i smjernice

# Računarstvo u oblaku

- Uslužno računarstvo (*utility computing*)
- Računalni resurs se ne kupuje, već se iznajmljuje po potrebi
- Plaća se onoliko računalnih resursa koliko se doista i koristi po principu „*pay as you go*”
- Privid neograničenosti računalnih resursa u oblaku
- Kapitalni trošak (CapEx) kupovine računalne opreme postaje operativni trošak (OpEx) iznajmljivanja resursa u oblaku
- Učinkovito korištenje računalnih resursa



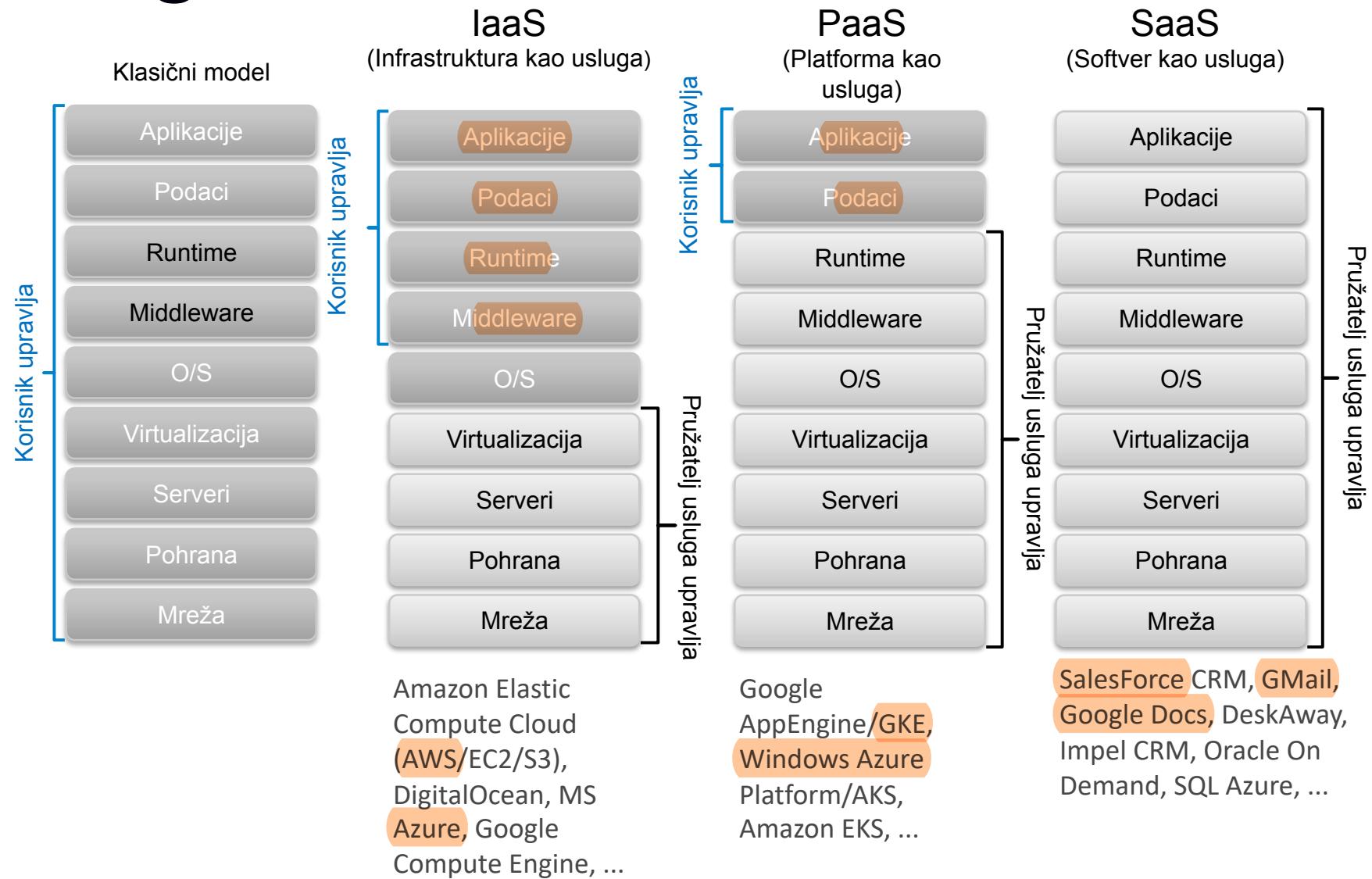
# Vrste računalnog oblaka prema smještaju i svrsi

- Javni oblak (*public*)
  - Iznajmljuje se za **javnu uporabu**
  - U vlasništvu je organizacije koja prodaje usluge u oblaku
- Privatni oblak (*private*)
  - U **privatnom vlasništvu poduzeća i samo** to poduzeće ga koristi
  - Ne smatra pravim „oblakom“
- Zajednički oblak (*community*)
  - Nekoliko **organizacija dijeli jednu infrastrukturu**
- Hibridni oblak (*hybrid*)
  - **Kompozicija** 2 ili više oblaka različitih vrsta

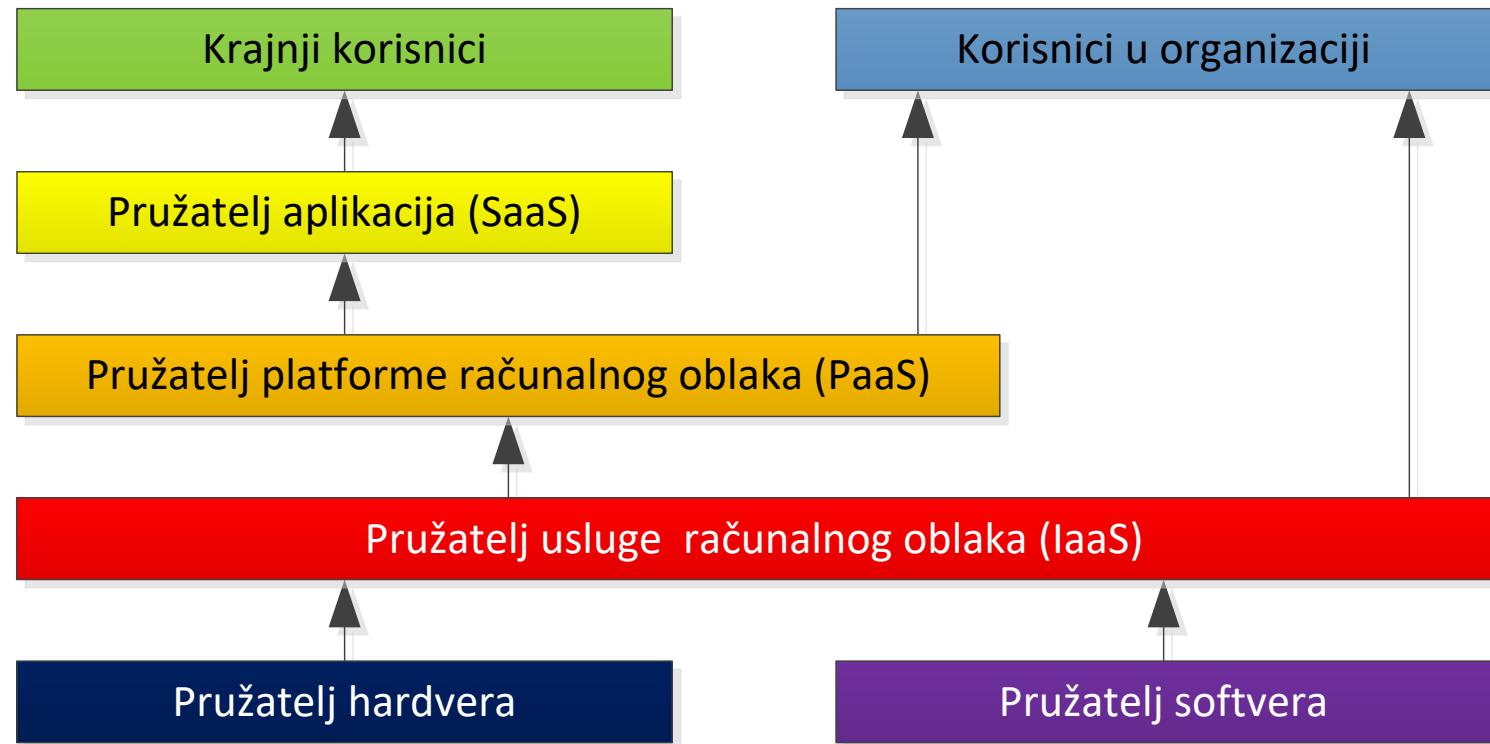
# Najvažniji koncepti

- Apstrakcija implementacije
  - Korisnik i razvijatelj ne znaju specifikaciju sustava na kojem će se aplikacije i usluge izvoditi
  - Podaci spremljeni na lokacije koje nisu poznate
  - Administracija sustava nije pod kontrolom razvijatelja
  - Pristup aplikacijama i uslugama je omogućen putem Interneta
- Virtualizacija
  - Omogućuje izvršavanje više operacijskih sustava na jednom fizičkom ili na više fizičkih računala
  - Isto vrijedi za pohranu podataka
  - Resursi se mogu dijeliti ili udruživati.
  - Računala mogu imati podršku za virtualizaciju (tehnologija hipervizora)
  - Primjeri: Xen, VMware, Wine, ...

# Modeli usluga



# Stranke u računalnom oblaku



Izvor: [1]

# Prednosti računarstva u oblaku u odnosu na vlastitu opremu

- plaćanje prema korištenju
- nadogradnja resursa na zahtjev
- elastičnost resursa
  - „automatsko“ proširivanje mrežnih i računalnih kapaciteta ovisno o opterećenju
- ne zahtijevaju se kapitalni troškovi za izgradnju infrastrukture
- povećana pouzdanost
- jednostavnije upravljanje održavanjem i nadogradnjom sustava
- usluge dostupne preko web-preglednika

# Nedostaci računarstva u oblaku

- nemogućnost prilagodbe klijentu kao u slučaju kada on sam upravlja sam svojim fizičkim računalima
- aplikacije koje su izvan oblaka obično imaju veće mogućnosti, nego aplikacije u oblaku (SaaS)
  - iz godine u godinu se smanjuju razlike, a kod nekih usluga i obrnuto
- ako aplikacija zahtjeva prijenos velikih količina podataka oblak nije najbolje rješenje
- ovisnost o jednom pružatelju usluga u oblaku
  - ovo ovisi o sustavu i tehnologijama koje se koriste
- usvajanje novog načina razvoja aplikacija
- privatnost i sigurnost podataka
  - neznanje o lokaciji spremanja podataka (sada se zna regija - GDPR)
  - lokalnih zakona o sigurnosti podataka (GDPR)

# Primjeri računalnog oblaka



- <https://aws.amazon.com>



Analytics



Application Integration



AR & VR



AWS Cost Management



Blockchain



Business Applications



Compute



Customer Engagement



Database



Developer Tools



End User Computing



Game Tech



Internet of Things



Machine Learning



Management & Governance



Media Services



Migration & Transfer



Mobile



Networking & Content  
Delivery



Quantum Technologies



Robotics



Satellite



Security, Identity &  
Compliance



Storage



See All Products

# Primjeri računalnog oblaka



- <https://cloud.google.com>

GOOGLE CLOUD PLATFORM

## AI and Machine Learning

[Speech-to-Text](#) · [Vision](#) · [Translation](#) · More

## API Management

[Apigee API Platform](#) · [Cloud Endpoints](#) · More

## Compute

[Compute Engine](#) · [Cloud GPUs](#) · More

## Hybrid and Multi-cloud

[Anthos](#) · [Migrate for Anthos](#) · [GKE](#) · More

## Data Analytics

[BigQuery](#) · [Cloud Datalab](#) · More

## Databases

[Cloud SQL](#) · [Cloud Firestore](#) · More

## Developer Tools

[Cloud Build](#) · [Cloud Code](#) · [Cloud SDK](#) · More

## Migration

[Data Transfer](#) · [VM Migration](#) · More

## Networking

[DNS](#) · [CDN](#) · [Virtual Private Cloud](#) · More

## Security and Identity

[Shielded VMs](#) · [Cloud IAM](#) · More

## Serverless Computing

[Cloud Run](#) · [App Engine](#) · [Cloud Functions](#) · More

## Storage

[Cloud Storage](#) · [Persistent Disk](#) · More

# Primjeri računalnog oblaka



- <https://azure.microsoft.com/en-us/>

⚡ Featured      Internet of Things

AI + Machine Learning      Management

Analytics      Media

Blockchain      Migration

Compute      Mixed Reality

Containers      Mobile

Databases      Networking

Developer Tools      Security

DevOps      Storage

Hybrid      Web

Identity      Windows Virtual Desktop

Integration

**Featured**  
Explore some of the most popular Azure products

**Virtual Machines**  
Provision Windows and Linux virtual machines in seconds

**Azure SQL Database**  
Managed, intelligent SQL in the cloud

**Azure Cosmos DB**  
Globally distributed, multi-model database for any scale

**Azure Kubernetes Service (AKS)**  
Simplify the deployment, management, and operations of Kubernetes

**Cognitive Services**  
Add smart API capabilities to enable contextual interactions

**Windows Virtual Desktop**  
The best virtual desktop experience, delivered on Azure

**App Service**  
Quickly create powerful cloud apps for web and mobile

**PlayFab**  
The complete LiveOps back-end platform for building and operating live games

**Azure Functions**  
Process events with serverless code

**Azure Quantum**  
Experience quantum impact today on Azure

# Primjeri računalnog oblaka



- <https://www.digitalocean.com>

## FEATURED PRODUCTS



### Droplets

Scalable compute services



### Kubernetes

Managed Kubernetes clusters



### Databases

Worry-free setup & maintenance



### Spaces

Simple object storage

## COMPUTE

Droplets

Kubernetes

## STORAGE

Spaces Object Storage

Volumes Block Storage

## MANAGED DATABASES

MySQL

PostgreSQL

Redis™

## DEVELOPER TOOLS

API

CLI

Monitoring

Teams

## NETWORKING

Cloud Firewalls

Load Balancers

Floating IPs

DNS

# Računalni oblaci u HR

- <https://ictmarketplace.hr> (t-com)



- <https://sysportal.carnet.hr>



- <http://www.megatrend.com/usluge/cloud-usluge/>

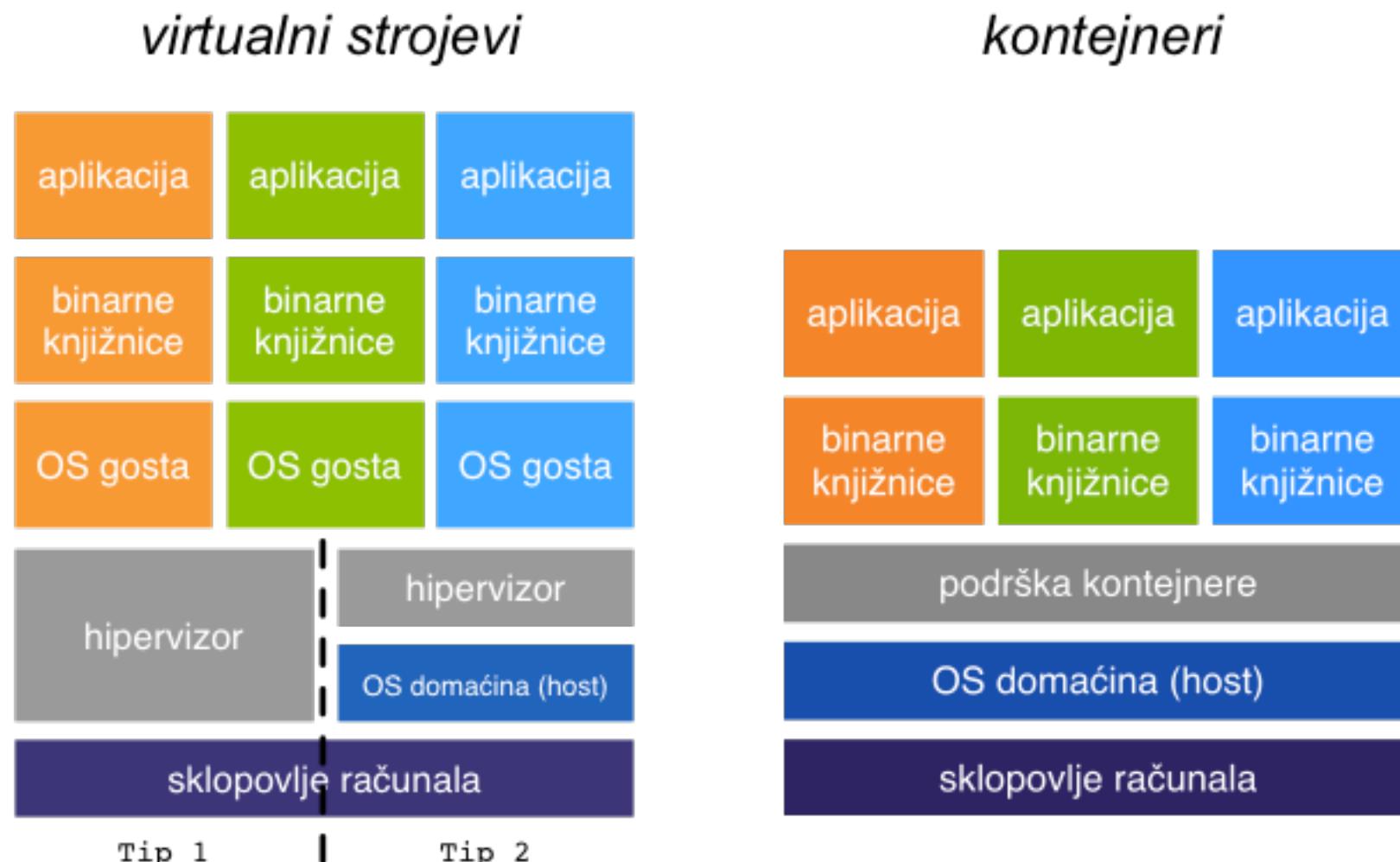


- <https://www.a1.hr/poslovni/ict-rjesenja/aplikacije>



# Kontejneri

# Razlika između kontejnera i virtualnog stroja



# Prednosti i mane kontejnera

- Prednosti:
  - ne pokreće čitav operacijski sustav
  - brže se pokreće
  - koristi manje resursa (procesora, memorije i diska)
- Mane:
  - slabija kontrola korištenja resursa
  - slabija izolacija između kontejnera
    - potencijalni sigurnosni problem
  - svaki kontejner mora koristiti isti kernel domaćinovog OS-a

# Primjeri implementacija

- Imunes – <http://imunes.net> (UniZG – FER – ZZT)
  - prvenstveno za IP mreže
- Docker – <https://www.docker.com>
  - najpoznatiji
- runC – <https://github.com/opencontainers/runc>
- rkt – <https://github.com/coreos/rkt>
  - razvija ga [CoreOS](#)
- containerd, LXC/LXD, OpenVZ, systemd-nspawn, machinectl, qemu-kvm, lkvm, ...
- Više se može naći [ovdje](#).

# Open Container Initiative specification

- <https://www.opencontainers.org>
- Osnovana 2015. na inicijativu Dockera i ostalih
- Definira:
  - izvršnu specifikaciju
  - specifikaciju slika

# Tehnologije potrebne za izvršavanje kontejnera

- Kontejner je zapravo skup izoliranih procesa u korisničkom prostoru operacijskog sustava (Linux/Windows)
- Tehnologije za izvršavanje na Linuxu:
  - **Namespaces** - ograničava što proces vidi od okoline u kojoj se izvršava:
    - Unix Timesharing System (*hostname*), Process IDs, Mounts, Network, User IDs, ...
  - **chroot**
    - promjena korijenskog direktorija nekog procesa
  - **Cgroups** - limitira korištenje resursa
    - memorija, procesor, I/O (količina podataka prenesena), broj procesa, ...

# Docker - [www.docker.com](http://www.docker.com)



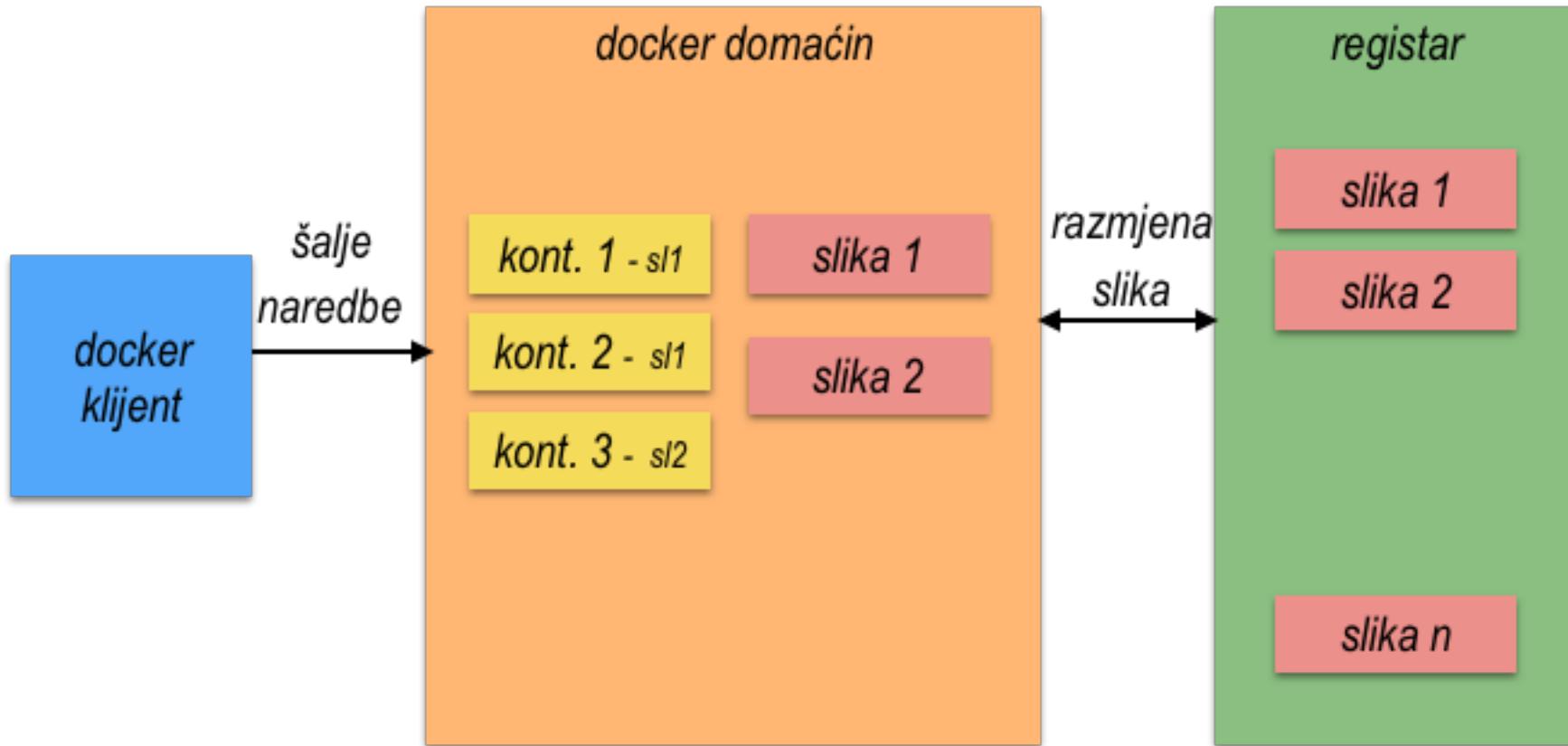
- Pojmovi:
  - *image* - slika OS-a
  - *container* - pokrenuta slika OS-a
  - *docker host* - računalo na kojem je pokrenut sustav docker
  - *docker hub* - portal za razmjenu slika (registar)
    - postoje i drugi: quay.io, gcr.io, registry.redhat.io
  - Dockerfile - tekstualna datoteka s konfiguracijom za izgradnju slike i za pokretanje kontejnera
- Detaljne video materijale možete naći [ovdje](#).

# Aspekti Dockerovog uspjeha



- sve su napravili da se može koristiti jednostavno
- 3 aspekta Dockera:
  - izrada slika
    - svu konfiguraciju stavljamo u datoteku (Dockerfile)
    - izgrađujemo sliku na temelju konfiguracije
    - sliku možemo spremiti u datoteku
    - slike se sastoje od nepromjenjivih slojeva (Union file system)
  - isporuka slika
    - sliku možemo podijeliti na docker hubu
    - lagano možemo pretraživati i skidati slike
    - imenovanje slika:
      - slike pojedinaca <repozitorij>/<proizvod>:<tag>
      - službene slike <proizvod>:<tag>
  - pokretanje slika
    - kontejner je pokrenuta slika

# Arhitektura i izvršavanje



# Izvršavanje Dockera na različitim OS-ovima



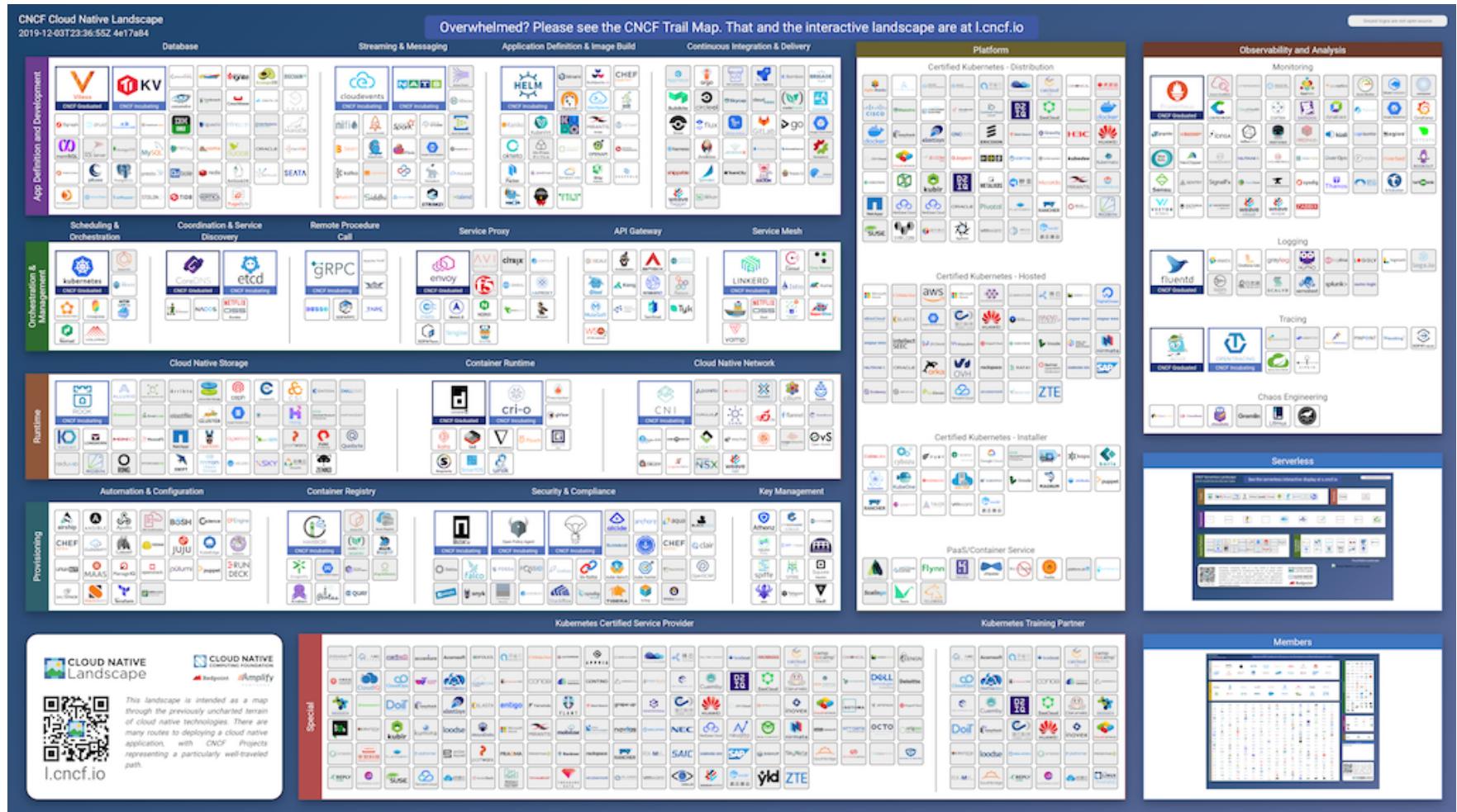
- Docker se izvršava na Linuxu
  - Od 2017. Windows aplikacije se mogu staviti u docker kontejner i pokretati na Windows strojevima
- Za izvršavanje na Windowsima ili Macu je potreban virtualni stroj s linuxom:
  - Docker Machine koji koristi VirtualBox za stvaranje stroja (boot2docker Linux distribution) - danas se rijetko koristi
    - konfigurira varijable okoline da bi se klijent mogao spojiti na taj stroj
  - **Docker Desktop**
    - na Macu koristi *native* Hypervisor.framework
    - na Windowsima koristi Microsoft Hyper-V

# Što kada imamo puno kontejnera?

- Potreba za upravljanjem kontejnerima na puno virtualnih računala:
  - Docker Swarm, Apache Mesos, Kubernetes, ...
- **Kubernetes (K8s):**
  - Automatsko postizanje i održavanje zadanog stanja aplikacije
  - Optimalno korištenje dostupnih resursa računalnog grozda
- *Service mesh*
  - Infrastruktura koja služi za povezivanje usluga u kontejnerima
  - Mogućnosti: otkrivanje usluga, upravljanje opterećenjem, praćenje usluga, autentikacija i autorizacija, šifriranje prometa, upravljanje ispadima
  - Implementacije: **Istio**, Lyft, Linkerd, Consul, ...
- Multicloud
  - Kontejneri se nalaze u više oblaka

# Cloud Native Computing Foundation ([CNCF](#))

- [CNCF Landscape](#)



# Kubernetes (K8s)



- Nastao u Googleu na temelju Borga (2014.)
- Danas ga održava CNCF
- Služi za orkestraciju kontejnera
- Neovisan o oblaku u kojem se izvršava
- Većina pružatelja oblaka pruža K8s kao PaaS ili IaaS
- Mi definiramo krajnje stanje, a K8s se brine da ga održi
- Svojstva:
  - Horizontalno skaliranje
  - Otkrivanje usluga
  - Uravnoteženje opterećenja
  - Samoizlječenje (*self-healing*)
  - Zamjena verzija bez zaustavljanja produkcije
  - Upravljanje konfiguracijama
  - Upravljanje tajnama (zaporke, ...)



# Kubernetes (K8s) – koncepti

- *node* (čvor) – jedno virtualno računalo na kojem se izvršava k8s
- *pod* (kapsula) – najmanji koncept koji se isporučuje, u sebi može imati više kontejnera
- *replica set* – pravila za repliciranje, pazi da određeni broj replika imamo u sustavu
- *deployment* – pravila za čitavu aplikaciju, brine se za verzije slika
- *service* – kada se izvana pristupi usluzi onda se promet preusmjeri na neki od kapsula i na određena vrata (tj. kontejner unutar te kapsule)
- *namespace* – odvaja dijelove k8s grozda
  - može se definirati ograničenje korištenja resursa (*quota*)
  - mogu se definirati prava korisnika
  - obično devops tim definira *namespace* i daje nam prava
- *label* – oznaka nekog koncepta
- *selector* – definira na koje koncepte se odnosi pojedini dio konfiguracije. Odabire se na temelju oznaka (*label*)

# Kubernetes (K8s) – arhitektura



- Dvije vrste čvorova:
  - Glavni (*master*) čvor:
    - **etcd** – raspodijeljena baza podataka sa svim elementima grozda
    - **kube-apiserver** – prima naredbe od kubectl i provodi ih
    - **kube-scheduler** – respordeđuje kapsule na radne čvorove (memorija, procesor, konflikti, ...)
    - **kube-controller-manager** - izvršava ono što je zapisano u etcd
    - Za pouzdani rad trebamo 3-5 instanci
  - Čvor radnik (*worker*):
    - **kapsule (pods)** – u njima se izvršavaju kontejneri
    - **kublet (Node Agent)** – prati što se događa i komunicira s glavnim čvorom (kontroler manager)
    - **kube-proxy (networking component)** – omogućuje uslugama da se neka vrata vide izvana
    - **container runtime (CRI - docker, rkt, ...)** – služi za izvršavanje kontejnera
    - na njima se izvršavaju naši programi, obično ih ima više/puno

# Istio



- Automatsko upravljanje opterećenjem za protokole: HTTP, gRPC, WebSocket i TCP
- Fina kontrola prometa: pravila usmjerenja, ponovno slanje zahtjeva, preusmjerenje u slučaju grešaka, ...
- Definiranje politika: prava pristupa, ograničenja (*limits, quota*)
- Automatska metrika, logovi, *traces*
- Sigurna komunikacija između usluga sa zaštitom identiteta i kriptiranjem komunikacije
- Arhitektura: [ovdje](#)

# Istio – komponente i arhitektura



- Komponente:
  - Envoy proxy – izvršava se u svakoj kapsuli (*side car*)
    - sva komunikacija između usluga ide preko njega
  - Mixer – komunicira s envoyjem
    - šalje pravila po kojima envoy radi
  - Pilot
    - otkrivanje usluga
    - upravljanje prometom za inteligentno usmjeravanje (npr. A/B testovi, canary, ...)
    - otpornost na kvarove (*resiliency*): vrijeme odziva, ponovno slanje zahtjeva, osigurač, ...
  - Citadel
    - sigurnost, certifikati, identiteti
  - Galley
    - upravljanje konfiguracijama

# Mikrousluge

# Motivacija

- Aplikacije se nalaze na svakom pokretnom uređaju (telefoni, tablet), u oblaku na tisućama računala
- Očekivanja:
  - vrijeme odgovora je u milisekundama (za ljude i za strojeve)
    - utjecaj na mrežu i procesiranje
  - usluga radi stalno (100% vremena)
    - bez obzira na to koliko korisnika ju trenutno koristi

# Usporedba arhitekture usluga (2005-2017)

~2005

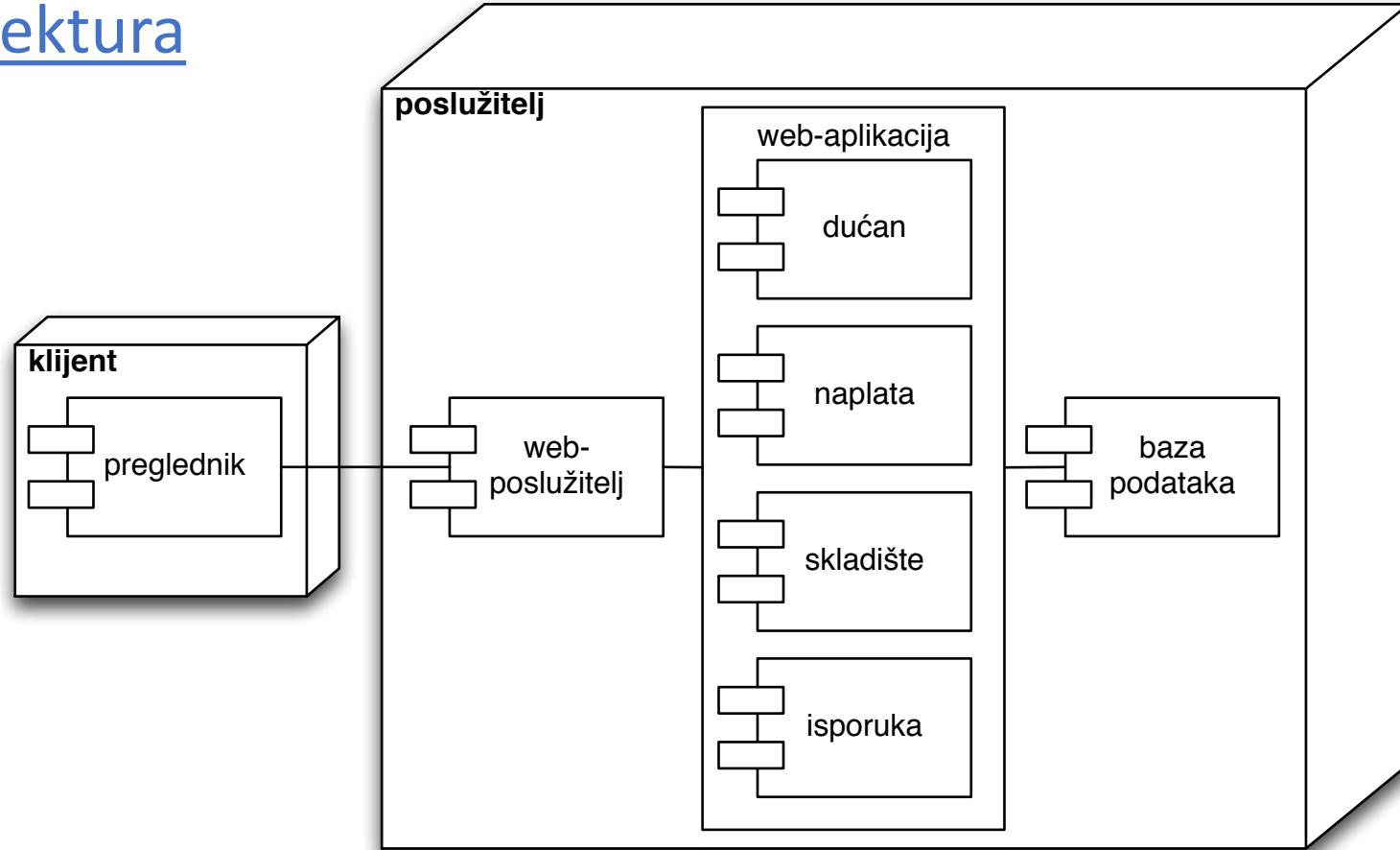
- Jedno računalo
- Jedna jezgra
- Skup RAM
- Skup disk
- Spora mreža
- Nekoliko konkurentnih korisnika
- Mala količina podataka
- Kašnjenje u sekundama
- Monolitne aplikacije
- Isporuka svakih nekoliko mjeseci

~2017

- Grozd računala
- Više jezgri
- Jeftin RAM
- Jeftin disk
- Brza mreža
- Puno konkurentnih korisnika
- Velika količina podataka
- Kašnjenje u milisekundama
- Pristup s mikrouslugama
- Isporuka X puta (10-100x) dnevno (DevOps)

# Tradicionalna monolitna aplikacija

- monolitna arhitektura



# Prednosti monolitne arhitekture

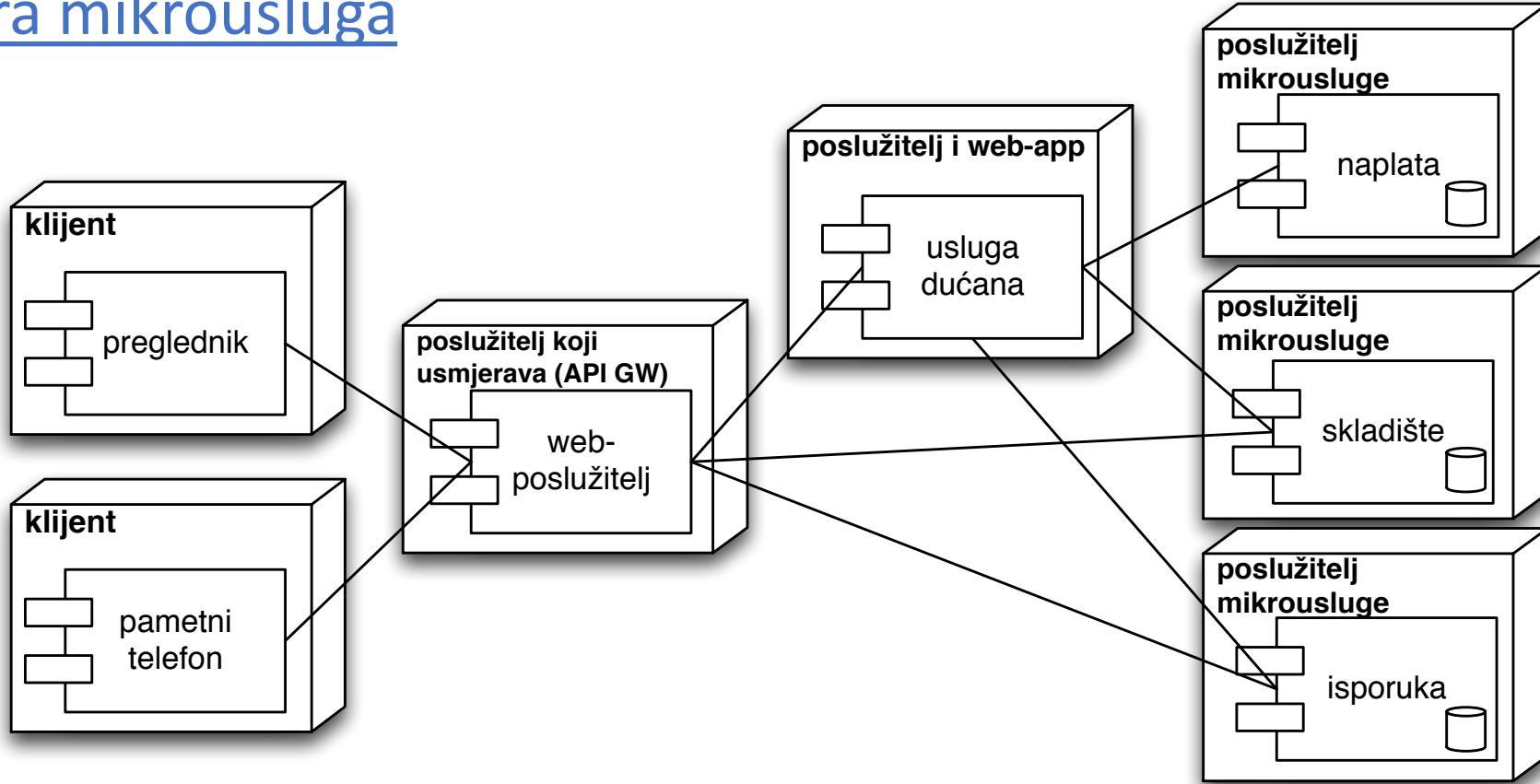
- Jednostavno koristiti druge module
  - Sve u jednom jeziku
  - Sve u jednom procesu pa je korištenje zapravo pozivanje funkcija/metoda što je brzo
- Jednostavno isporučiti – na jedan poslužitelj
- Jednostavno pratiti promjene u kodu (jedan izvorni kod)
- Jednostavno skalirati – jednu aplikaciju pokrenemo više puta
- Jezični konstrukti i radni okviri upravljuju kompleksnošću

# Nedostaci monolitne arhitekture

- Vezanost za programski jezik i programske okvire
  - Sve se radi u jednoj tehnologiji
  - Ne možemo koristiti nove pristupe za pojedine probleme
    - Npr. reaktivno programiranje, noSQL baze, ...
- Shvaćanje cijele aplikacije
  - Jedan razvijatelj ne može znati čitav kod
  - Jedan tim ne može upravljati čitavom aplikacijom
    - Svaki tim je za pojedini modul
  - Veliki timovi (stotinjak članova koji su zapravo puno malih timova)
- Isporuka kao jedna jedinica (npr. war datoteka)
  - Zbog male promjene moramo isporučiti čitavu aplikaciju
  - Jedna promjena može uzrokovati lanac promjena u čitavoj aplikaciji

# Mikrousluge

- arhitektura mikrousluga



# Prednosti mikrousluga

- jednostavno shvatiti jednu uslugu (kod i ulogu), ali ne i čitav sustav
- manji timovi za razvoj jedne usluge
- kontinuirana isporuka, brži razvojni ciklus/česte isporuke
- mogućnost korištenja novih/različitih tehnologija u različitim uslugama
  - baze, programski jezici, poslužitelji, ...
- fleksibilniji raspored usluga u sustavu
- raspodijeljeni podaci (različite baze podataka - SQL, noSQL)
- horizontalna skalabilnost – jednostavno pokrenuti nove instance pojedine usluge
- veća pouzdanost – ako imamo više instanci iste usluge
- moguća brza regeneracija sustava nakon pada performansi
- neovisnost o programskom jeziku, radnom okviru, infrastrukturi oblaka

# Nedostaci mikrousluga

- prednosti ne dolaze same po sebi
- kompleksnost prebačena sa aplikacije na integraciju
  - više održavanja
  - više znanja i vještina trebaju imati razvijatelji i održavatelji (*operations/administratori*)
  - teže testiranje čitavog sustava
- složenost raspodijeljenog sustava
  - usluge mogu biti nedostupne (greška u programu, računalu, mreži)
- asinkronost u komunikaciji je teško kombinirati
- treba više pratiti stanje sustava (automatsko promatranje)
  - potrebna je dodatna infrastruktura
- nema raspodijeljenih transakcija
  - upotrijebiti eventualnu konzistentnost
- razvijatelji moraju biti svjesni raspodijeljenosti sustava

# Mikrousluge – definicije eksperata

“...the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.”

James Lewis, Martin Fowler, ([poveznica](#))



**adrian cockcroft**  
@adrianco

Follow

@kellabyte @mamund I used to call what we did "fine grain SOA". So microservices is SOA with emphasis on small ephemeral components

([poveznica](#))

# Praktična definicija

- Aplikacija sastavljena od puno malih usluga
- Svaka usluga se izvršava u neovisnom procesu
  - Ne moraju biti na istom računalu
- Mikrousluge komuniciraju otvorenim protokolima
  - Npr. HTTP/REST, AMPQ, gRPC, Thrift, ...
- Svaka mikrousluga može biti napisana, isporučena, skalirana i održavana na drugačiji način
  - Npr. koriste se različiti programski jezici i različite baze podataka
- Svaka mikrousluga enkapsulira neki poslovni aspekt
  - Ne programske konstrukte (biblioteka, modul, paket, ...)
- Svaka mikrousluga je neovisna, zamjenjiva i može se samostalno nadograditi

# Što mikrousluge nisu?

- Nije isto što i SOA
  - SOA integrira poslovne (*enterprise*) aplikacije
  - Mikrousluge su namijenjene za dekompoziciju jedne aplikacije
- Nije najbolje rješenje za svaki problem
  - **Korištenje mikrousluga ima svoje nedostatke i rizike**

# Zablude programiranja u raspodijeljenoj okolini (čitaj oblak)

- Mreža je pouzdana
- Nema kašnjenja
- Propusnost je neograničena
- Mreža je sigurna
- Topologija se ne mijenja
- Cijena transporta ne postoji
- Mreža je homogena
- Postoji jedan administrator

# Komponentizacija u mikrousluge

- Melvyn Conway, 1967 (Conway's Law):
  - Organizacija dizajnira sustav koji je kopija komunikacije u toj organizaciji
- Komponente/mikrousluge nisu konstrukti jezika ili radnog okvira
- Izolacija između komponenti je vrlo bitna i utječe na dizajn
  - Pažljivo treba definirati sučelja (npr. REST) koje će druge usluge koristiti
    - Pristup *Consumer Driven Contracts*
  - Ako se sučelja ne mijenjaju
    - Svaka mikrousluga se može mijenjati kada je ona spremna
    - Promjena jedne mikrouslugе **ne smije** zahtijevati promjenu drugih mikrousluga
  - Ako se sučelja mijenjaju treba ih napraviti tako da promjene ne uzrokuju trenutnu promjenu kod drugih usluga
  - Omogućuje lakše: skaliranje, praćenje, *debugging*, neovisno testiranje pojedine usluge
- Kako podijeliti aplikaciju u usluge?
  - Primarni način je prema poslovnoj funkcionalnosti
  - Paziti da se poštuje princip jedne odgovornosti ([Single Responsibility Principle](#)) – slično kao unix alati
  - [Ograničeni kontekst](#)
- Obrasci: [prikupljeni obrasci](#), [najčešće korišteni obrasci](#)

# Komunikacija i upravljanje

- Protokoli: HTTP, TCP, UDP, poručivanje (*messaging*)
  - Sadržaj: JSON, BSON, XML, Protocol Buffers (gRPC), Thrift, ...
- Poručivanje: koristimo događaje, gledamo u povijest, drugačiji pristup
- Tjera nas na definiranje jasnih sučelja
- Komuniciranje ide preko API-a, a ne preko baze podataka
- Treba koristiti najbolji alat za problem koji se rješava (programski jezik, radni okvir, baza podataka)
- Mikrousluge se mijenjaju i isporučuju različitim brzinama
- Mikrousluge mogu mijenjati IP adresu (kod skaliranja ili kod restarta) – potrebna je infrastruktura za otkrivanje usluga

# Problemi

- Što ako je takvih usluga 100?
  - Kako znamo na kojim je vratima (*port*) koja usluga?
  - Kako znamo da je svaka usluga pokrenuta?
  - Što ako trebamo promijeniti konfiguraciju nekih usluga?
  - Što ako neka usluga ne radi (praćenje performansi i grešaka - *monitoring*)?
  - Što ako je neka usluga preopterećena pa ne odgovara na vrijeme, nego sa zakašnjnjem koje nije prihvatljivo?
  - Kako pratiti kompleksnost sustava?
  - Kako znati koji *build*, koja verzija sw-a je na kojem čvoru (*traceability*)?
  - Kako doći do dnevničkih zapisa (raspodijeljeni *logging*)?
- Potrebno je veliko znanje u organizaciji (ulaganje u ljude)
- Svi testovi moraju biti automatizirani (DevOps)

# Aplikacije s 12 faktora

1. Svaka mikrousluga ima jedan repozitorij (npr. git) za kod i iz njega se sve prati i stvaraju izvršne verzije
2. Ovisnosti - eksplisitno definirane ovisnosti o drugima
3. Konfiguracija – u okolini, ne u kod
4. Podržavajuće usluge (*backing service*) – tretirati ih kao resurse koji se mogu konfigurirati
5. Izgradnja, objava, pokretanje – odvojiti faze i definirati pravila za izdavanje
6. Usluge se izvršavaju kao procesi bez stanja
7. Objavljuvanje usluga preko definiranih vrata (*port*)
8. Skaliranje pomoću pokretanja novih procesa
9. Povećanje robusnosti pomoću brzog pokretanja i pristojnog gašenja
10. Paritet između razvojne i produkcijske okoline
11. Tretirati zabilješke (*log*) kao tok događaja
12. Administratorski procesi bi se trebali izvršavati u istoj okolini kao i produkcijski

# Rješenja nekih problema (1)

- konfiguriranje
  - jednostavna primjena
  - jednostavno vraćanje na prethodnu konfiguraciju u slučaju grešaka
- lokacijska transparentnost
  - kada se neka usluga ugasi i druge pokrene na novoj IP adresi i/ili vratima (*port*) usluga koja ju koristi bi ju automatski trebala početi koristiti
- praćenje u radu
  - metrike (opterećenje, memorija, broj zahtjeva, kašnjenje)
    - spremnost za primanje zahtjeva
    - je li usluga pokrenuta ili se srušila
  - *tracing* – praćenje jednog zahtjeva kroz usluge (korelacijski identifikator)
  - dnevnički zapisi (*log*) – skupljanje na jednom mjestu i analiza
  - alarmiranje u slučaju nepredviđenog ponašanja

# Rješenja nekih problema (2)

- otpornost na kvarove
  - kada neka usluga prestane raditi ili ima veliko kašnjenje usluga koja ju koristi to treba detektirati i imati alternativu
    - obrazac osigurača
    - označavanje usluge neispravnom i izbaciti ju iz uravnoteženja opterećenja
    - kada se obnovi potrebno je automatsko uključivanje usluge u sustav
  - verzioniranje
    - potrebno je imati informacije koja verzija usluge se nalazi na kojem resursu kako bi se kod detekcije kvara u usluzi znalo koju verziju treba popraviti
  - skaliranje
    - povećanje/smanjenje broja instanici
    - želimo automatsko
    - skaliranje do 0 instanci
    - problem skaliranja usluga sa stanjem
  - [Chaos engineering \(npr. Netflix Simian Army\)](#)

# Konfiguracije

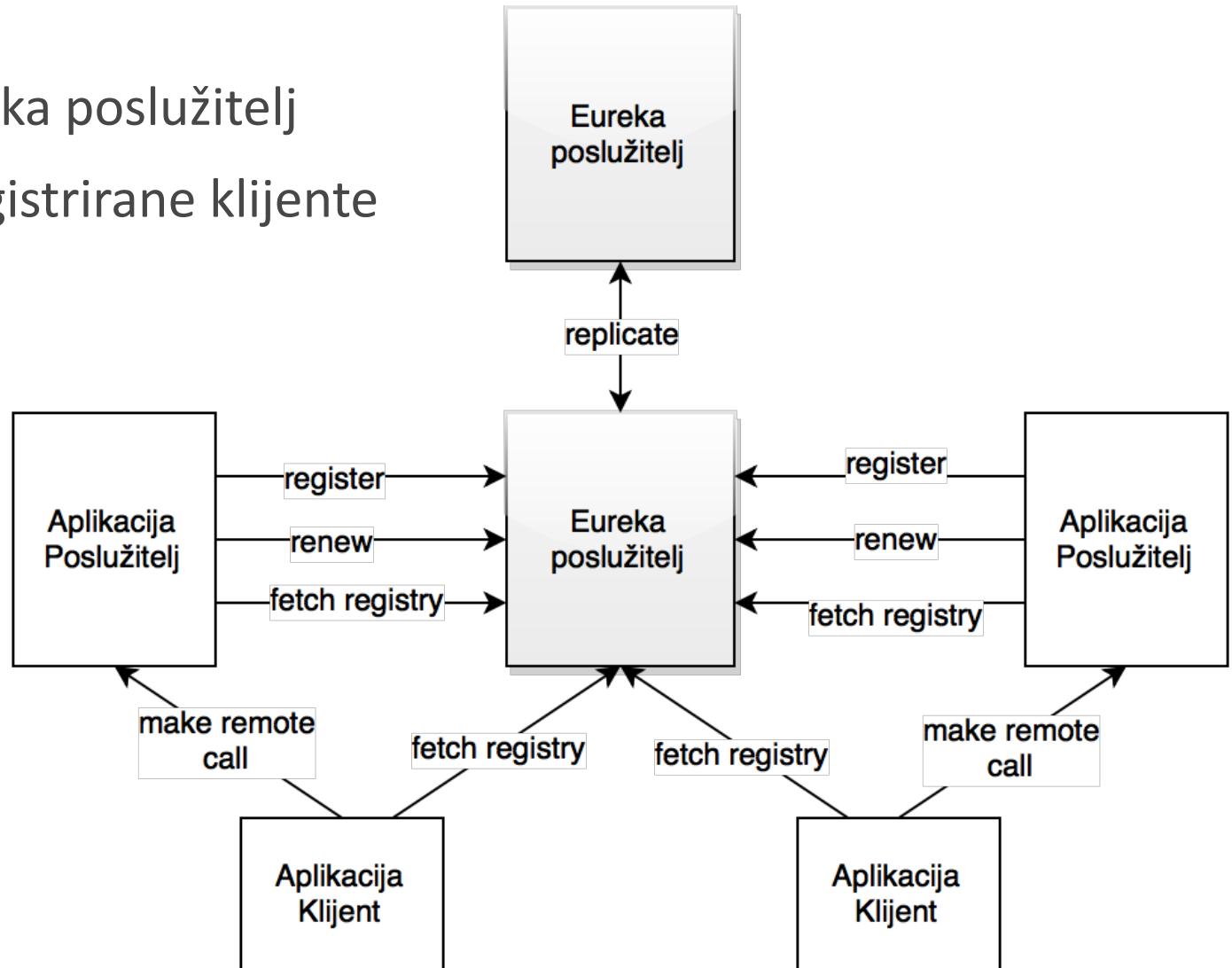
- Svaka aplikacija ima konfiguraciju
  - Npr. na koju bazu podataka se spaja, gdje su resursi, na kojim vratima (*port*) sluša, ...
- Konfiguracija nije u kodu, nego u nekoj datoteci kako bi se lakše moglo instalirati u drugoj okolini
- Gdje se konfiguracija može nalaziti:
  - U arhivi koja se isporučuje – ponovna isporuka u slučaju promjene
  - Konfiguracijske datoteke – na poznatom mjestu na disku (teško je osigurati takvo mjesto u okolini oblaka npr. AWS)
  - Variable okoline – različito na različitim OS-ovima, teško upravljati kada imamo puno podataka
  - Svaki oblak ima svoja rješenja za to – vezani smo za taj oblak
- Kubernetes ima mehanizam dijeljenja varijabli okoline
- Spring: Spring Cloud Config, Spring Cloud Bus (knjižnice)

# Otkrivanje usluga

- Otkrivanje s poslužiteljske strane
  - Komponenta za uravnoteženje opterećenja se brine o otkrivanju gdje se nalazi neka usluga
- Otkrivanje s klijentske strane
  - Klijent se brine o otkrivanju gdje je neka usluga
- Primjeri registara usluga:
  - etcd, consul, Apache Zookeeper, Netflix Eureka
- Spring Cloud Netflix Eureka
  - Aplikacije se same registriraju
  - Aplikacije mogu pretraživati registar
- Kubernetes – koristi DNS i uravnoteženje opterećenja
- Istio – koristi *proxy*

# Eureka

- Mikrousluga se registrira na Eureka poslužitelj
- Svaka usluga može pregledati registrirane klijente



# Uravnoteženje opterećenja

- Spring Cloud Netflix Ribbon (knjižnica)
  - Uravnoteženje opterećenja na klijentu
  - Automatski se integrira s Eurekom (kada je uključena u projekt)
    - Automatski dohvaća konkretnu IP adresu i vrata registrirane usluge
  - Može privremeno spremati rezultate (*caching*)
  - Podržava više protokola (HTTP, TCP, UDP)
  - Postoji više strategija (podrazumijevano *round-robin*)
- Kubernetes:
  - Posebna komponenta u grozdu koja radi uravnoteženje ravnomjerno s obzirom na broj instanci
- Istio:
  - koristi *proxy* kod svake usluge
  - puno finije mogućnosti usmjeravanja prometa

# Metrika mikrousluge

- zahtjev/sec., memorija, procesor, vrijeme odgovora, probe (*liveness, readiness*)
- Spring Boot Actuator (knjižnica)
  - usluga ima URI za različite metrike gdje se to može dohvaćati
- Kubernetes
  - koristi metrike od usluga
- Istio
  - proxy prikuplja podatke
- Tehnologije:
  - ELK stack
    - Elasticsearch - pretraživanje i analitika nad podacima
    - Logstash - prikuplja podatke o metriči sa različitih usluga
    - Kibana - vizualizacija podataka (grafovi, pite, ...)
  - Prometheus - sustav za praćenje i alarmiranje
  - Grafana - vizualizacija podataka

# Zašto je otpornost na kvarove bitna?

- Kada imamo veliki broj mikrousluga neka od njih neće biti dostupna zbog kvarova:
  - Računalo, mreža, preopterećenje, ...
- Zbog toga što jedna usluga poziva drugu velika mogućnost je da postoji kaskada usluga koje ne rade
  - Prepostavimo da sustav ima pouzdanost 99,95% - [Amazon EC2 Service Level Agreement](#)
  - Npr. [Netflix ima 13,000 kontejnera u produkciji \(2016.\)](#)
  - Monolitna aplikacija: ne radi 21,6 minuta mjesечно
  - Sustav mikrosuluga koji su povezane:
    - 10 povezanih mikrousluga: ne radi 3,59 sati mjesечно
    - 50 povezanih mikrousluga: ne radi 17,78 sati mjesечно
    - 100 povezanih usluga: ne radi 35,12 sati mjesечно

# Otpornost na kvarove

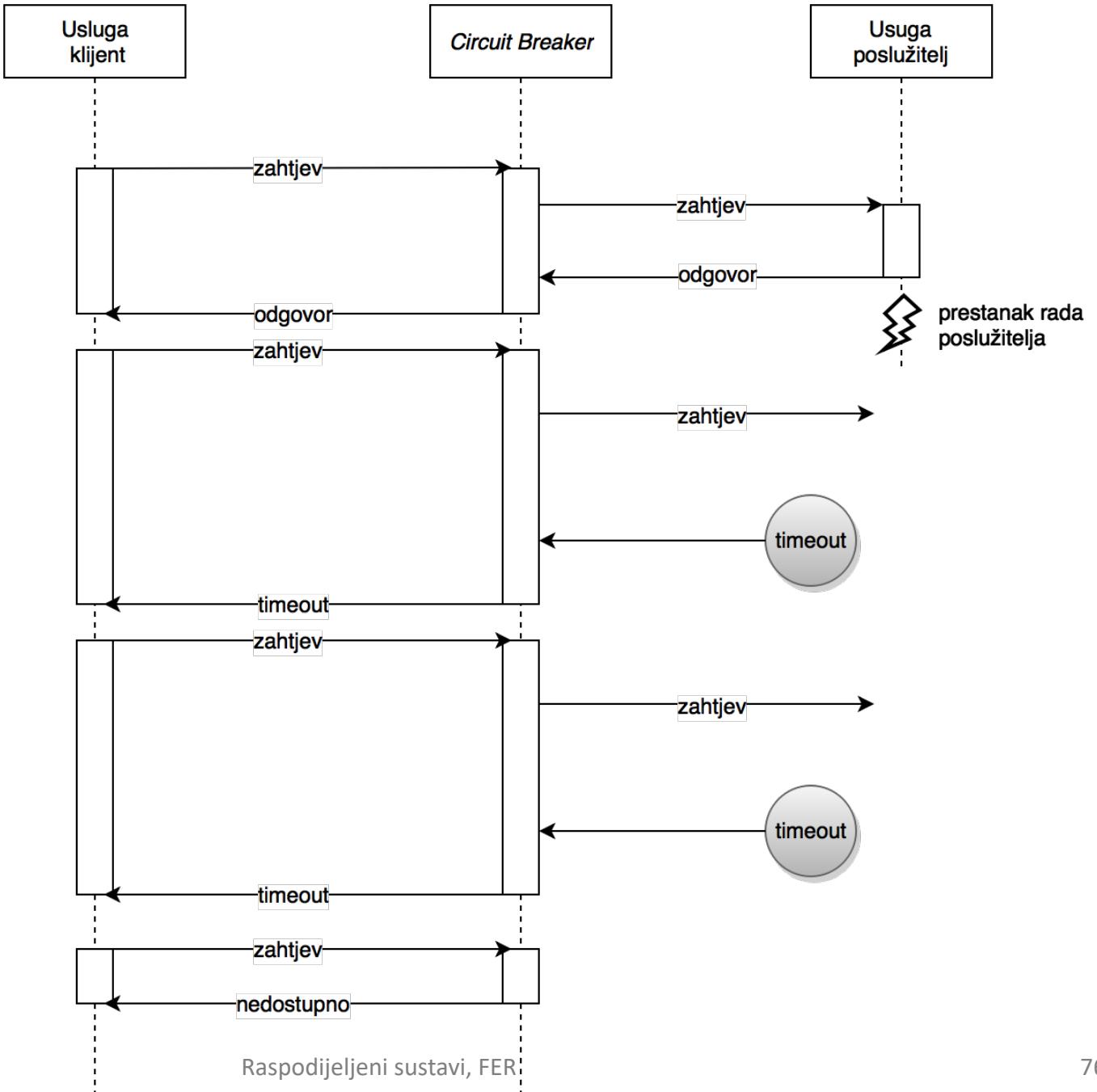
- Redundancija usluga
  - Više usluga može istovremeno raditi isti posao
- Vruća redundancija:
  - Na dva poslužitelja pristižu zahtjevi i oba rade isti posao, ali je jedan glavni. Ako glavni prestane raditi pomoći postaje glavni.
  - Nakon ponovnog uključivanja u rad potrebna je sinkronizacija podataka (ako se podaci spremaju)
- Princip osigurača:
  - Zahtjevi se uravnotežuju i jedna usluga prestane raditi
  - Nakon toga se zahtjevi ne šalju na tu uslugu
  - Npr. Hystrix
- Hijerarhija supervizora
  - Kada neka usluga ne radi supervizor ju gasi i ponovno pokreće
  - Postoji hijerarhija supervizora (npr. Erlang OTP ili Akka Framework)

# Izolacija neispravne usluge

- Kada neki dio ne radi treba ga što prije izolirati i ne slati zahtjeve na taj dio
- Na taj način se sprječavaju kaskadni ispadi
- **Hystrix** (knjižnica) je implementacija mehanizma osigurača između usluga
  - Detektira kvarove (iznimka ili kašnjenje) i izolira nedostupnu uslugu
  - Kvar se detektira statistički – npr. 20 kvarova u 5 sekundi
  - Omogućuje preusmjeravanje na pričuvnu uslugu dok se kvar ne popravi
  - Automatsko preusmjeravanje na popravljenu uslugu kada se vrati u funkciju
    - Nakon definiranog intervala provjere
- **Kubernetes:**
  - točke koje koristi za provjeru ispravnosti usluge povezuje s uravnoteženjem opterećenja
- **Istio**
  - *proxy* detektira ispade i dojavljuje komponenti za konfiguracije koja to proslijedi ostalima

# Hystrix

- implementira obrazac *circuit breaker*
- služi za povećanje otpornosti sustava na greške



# Strategije uvođenja novih verzija usluga

- *Recreate*
  - ugasiti staru verziju i pokrenuti novu
- *Ramped*
  - kada imamo više replika onda dodamo jednu novu verziju, pa ugasimo staru, i tako dok sve ne zamijenimo (Kubernetes)
- *Blue/Green*
  - instaliramo novu verziju (*blue*) i testiramo da radi, nakon toga samo sav promet preusmjerimo na novu verziju (*switch*)
- *Canary*
  - slično Blue/Green samo postepeno postotak sa stare verzije prebacujemo na novu (Istio)
- *A/B Testing*
  - samo neke korisnike prebacimo na novu verziju, primjeri kriterija: beta testeri, geografsko područje, korisnici pojedinog preglednika, jezik, OS, veličina ekrana, ... (Facebook, Istio)
- *Shadow (Dark Launch)*
  - instaliramo obje verzije i zahtjeve šaljemo na obje, ali samo od stare se odgovori vraćaju korisniku (testiranje u produkciji)

# Skaliranje usluge bez podataka

- Koriste se mehanizmi kao kod kvarova:
  - Redundancija usluga, usmjeravanje zahtjeva, dinamičko otkrivanje usluga, supervizor
- Bitna je lokacijska transparentnost
  - Kada se pokrene nova usluga da ju je lako otkriti
- Novi trend – Serverless Architecture - *Functions as a service*
  - Primjeri: [AWS Lambda](#), [Google Cloud Functions](#), [Azure Functions](#), [Project fn](#) (otvoreni kod), [Knative](#) (dodatak za K8S)
  - Ideja:
    - Kada pristigne zahtjev onda se pokreće usluga, proslijeđuje joj se zahtjev i kada ga obradi gasi se
  - Bitno svojstvo: brzo pokretanje
    - u desetinama sekundi, a ne u minutama

# Veličina i brzina pokretanja usluga

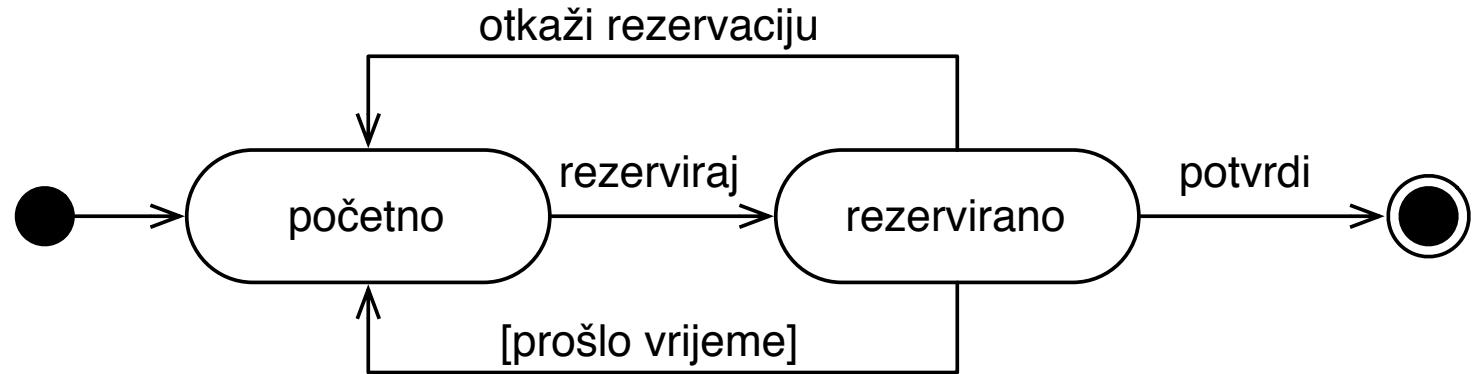
- Potrošnja memorije i brzina pokretanja aplikacija:
  - Node: 92MB, 18s
  - Quarkus: 82MB, 16s
  - Go: memorija slična Quarkusu, 19s
  - SpringBoot: 150-200MB, ~1min
- Quarkus - Supersonic Subatomic Java
- Izvori:
  - [Supersonic, Subatomic Java with Quarkus. Burr Sutter, Red Hat](#)
  - [Kubernetes Native Spring apps on Quarkus by Georgios Andrianakis](#)

# Skaliranje usluge s podacima

- Problem je kako osigurati podatke različitiminstancama usluge.
- Pristupi:
  - Replikacija podataka (*cache*)
  - Raspodijeljene baze podataka:
    - Segmentacija podataka (tablice na različitim računalima, podjela tablice na segmente)
    - Raspodijeljene transakcije (protokol dvofaznog izvršavanja – 2PC, standard Open XA)
  - P2P sustavi – npr. mehanizam Chord
  - Bilježenje događaja (*Event Logging*)
  - Raspodijeljena Saga

# Transakcije

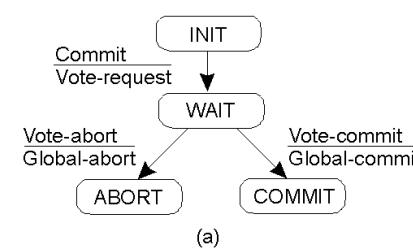
- Model transakcije
- Dva stanja:
  - Početno
  - Rezervirano



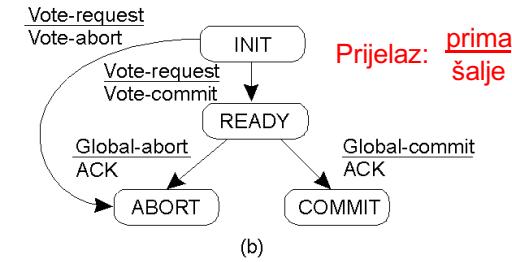
# ACID (izvor: kolegij Baze podataka)

- **Atomicity** – nedjeljivost transakcije (atomarnost) - transakcija se mora obaviti u cijelosti ili se uopće ne smije obaviti
- **Consistency** - konzistentnost- transakcijom baza podataka prelazi iz jednog konzistentnog stanja u drugo konzistentno stanje
- **Isolation** - izolacija - kada se paralelno obavljaju dvije ili više transakcija, njihov učinak mora biti jednak kao da su se obavljale jedna iza druge
- **Durability** - izdržljivost - ako je transakcija obavila svoj posao, njezini efekti ne smiju biti izgubljeni ako se dogodi kvar sustava, čak i u situaciji kada se kvar desi neposredno nakon završetka transakcije

# Raspodijeljene transakcije

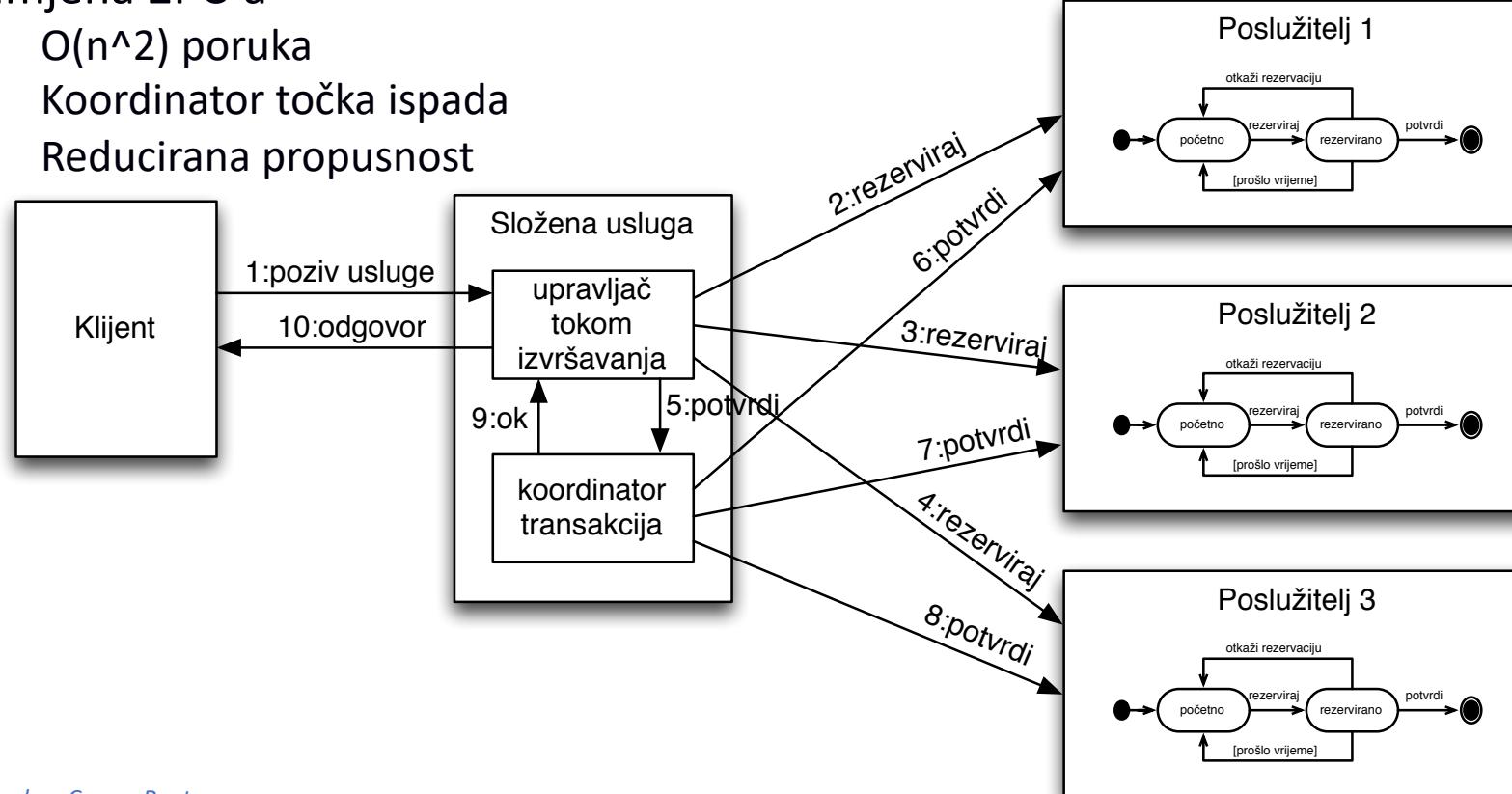


Automat stanja koordinatora



Prijelaz: prima  
šalje

- Primjena 2PC-a
  - $O(n^2)$  poruka
  - Koordinator točka ispada
  - Reducirana propusnost



Guy Pardon Cesare Pautasso  
REST: From Research to Practice

# ACID 2.0

- **Associative** – asocijativnost – grupiranje poruka ne utječe na rezultat (mat.  $a(bc) = (ab)a$ ), a omogućuje obradu naknadnu obradu u pozadini
  - **Commutative** – komutativnost – redoslijed poruka ne utječe na rezultat (mat.  $ab = ba$ )
  - **Idempotent** – idempotentnost – dupliciranje poruka ne utječu na rezultat
  - **Distributed** – raspodijeljenost – poslovi se mogu izvršavati raspodijeljeno
- 
- Primjena kod oblikovnog obrasca Saga
  - Primjer: CRDT (*Conflict-free Replicated Data Types*), ...

Pat Helland, Dave Campbell: “[Building on Quicksand](#)”, CIDR 2009

Joy BanerjeeShail Aditya Gupta: Method and system for event state management in stream processing, [patent US9680893B2](#)

VICTOR GRISHCHENKO: CID 2.0 - Associative, Commutative, Idempotent, Distributed, [poveznica](#)

# Alternativa transakcijama - model BASE (usluge/baze)

- ***Basically Available*** – ne garantira se teorem CAP, svaki zahtjev će dobiti odgovor, ali odgovor može biti “greška”. Slično kao u banci kada se čeka potvrda prijenosa sredstava.
- ***Soft state*** – stanje sustava se može promijeniti kroz vrijeme i u slučaju kada nema zahtjeva. To nazivamo “meko” stanje.
- ***Eventual consistency*** – Sustav će nakon nekog vremena nakon zadnjeg zahtjeva doći u konzistentno stanje (eventualna konzistentnost).
- Primjena kod: *Event Sourcing* i CQRS-a

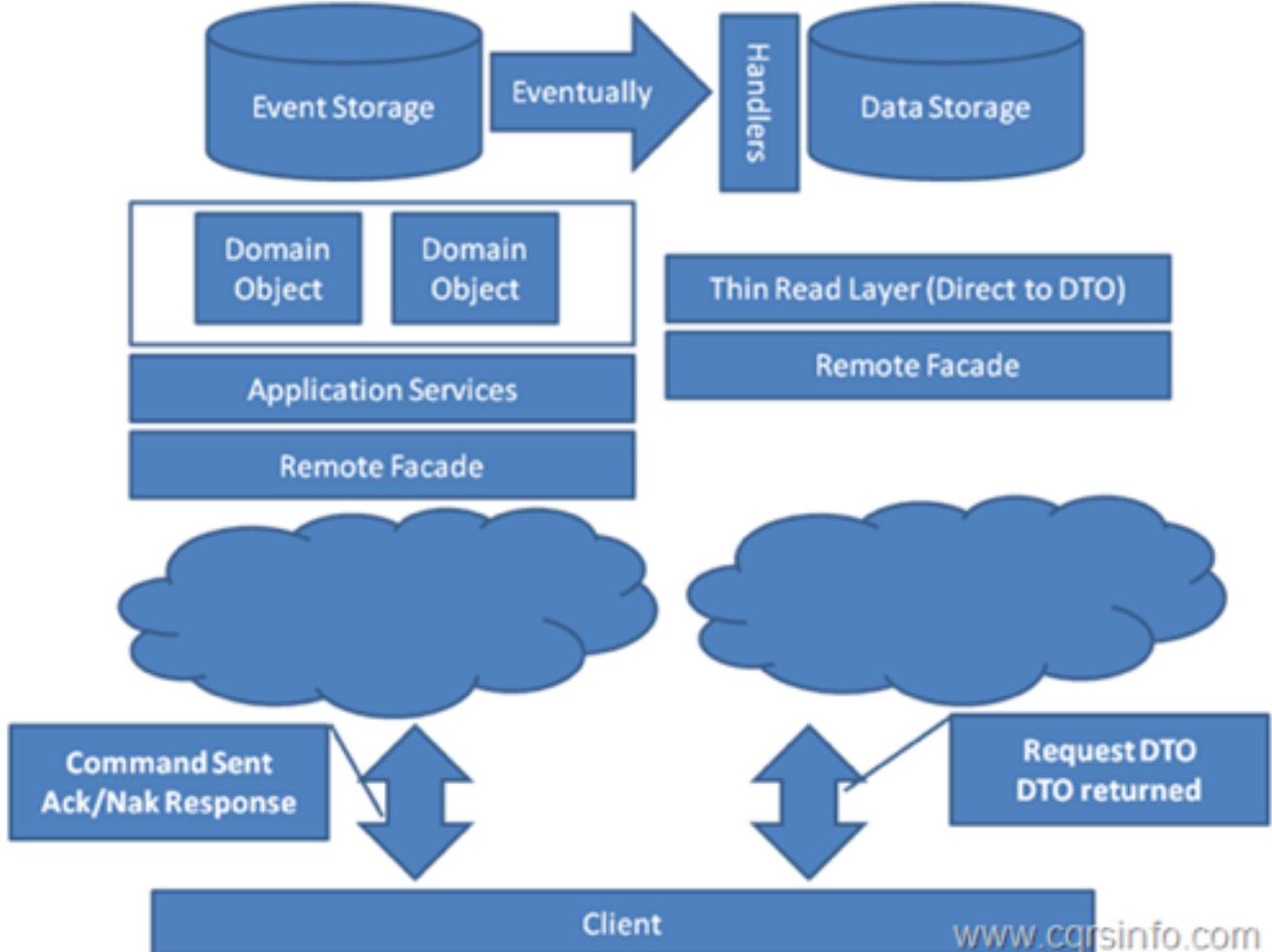
Dan Pritchett: "[Base: An Acid Alternative](#)", Queue, vol. 6, br. 3, str. 48-55

DekaGanesh Chandra: "BASE analysis of NoSQL database", Future Generation Computer Systems, Volume 52, November 2015, Pages 13-21

# Alternativa transakcijama

- *Bilježenje događaja (Event Logging)*
  - spremanje svih događaja u sustavu
  - ne spremi se stanje
  - nema brisanja
- *Event Sourcing*
  - spremjeni događaji se koriste za izračunavanje stanja sustava u nekom vremenu
  - događaji su izvor istine u sustavu
- Arhitekturni obrazac CQRS - *Command Query Responsibility Segregation*
  - autor: Greg Young
  - komande - stvaranje, promjena, brisanje
  - upiti - vraća pripremljeni pogled ili se pretplati na događaje
- Arhitekturni obrazac Saga

# CQRS i Event Sourcing



<https://cqrss.info/documents/cqrs-and-event-sourcing-synergy/>

# Bilježenje događaja (Event Logging)

- U log se bilježe svi događaji
  - Nema brisanja
- Događaji nisu stanja, već zahtjevi ili komande
  - Predstavlja činjenicu koja se već dogodila u prošlosti
- Takav log se koristi kao jedina ispravna istina
  - Primjena obrasca *Event Sourcing*
  - Omogućuje agregiranje događaja u sažetke ili stanje u pojedinom trenutku (obično u memoriji)
- Omogućuje efikasno korištenje obrasca CQRS (*Command Query Responsibility Segregation*) – autor Greg Young
  - Odvajanje model čitanje i pisanja
    - Komande - stvaranje, promjena, brisanje
    - Upiti - vraća pripremljeni pogled ili se pretplati na događaje

# Oblikovni obrazac: Saga

- Primjena u dugim transakcijama u bazama podataka (ne raspodijeljenim)
- Smanjuje vrijeme zadržavanja ključa (kod mehanizma zaključavanja)
- Definicija:
  - Saga je transakcija koja dugo traje, a može se zapisati kao niz transakcija kojima nije bitan redoslijed.
  - Sve transakcije ili:
    - uspješno završe i tada je Saga uspješno završila ili
    - ako jedna ne završi uspješno onda postoje kompenzirajuće transakcije koje se moraju izvršiti za svaku uspješnu transakciju i tako vratiti sustav u prijašnje stanje. Tada je saga neuspješna.
- Ograničenja:
  - Transakcije ne ovise jedna o drugoj tj. ne smije postojati redoslijed.
  - Svaka transakcija u Sagi mora imati kompenzirajuću transakciju koja semantički radi poništavanje transakcije.
- Nema atomarnosti, ali dobijemo uporabljivost sustava (nije zaključan)

Hector Garcia-Molrna, Kenneth Salem: "Sagas", 1987.

# Saga na jednoj bazi

- Postoji komponenta SEC (Saga Execution Coordinator)
  - Nalazi se unutar procesa baze podataka
  - Upravlja Sagom
- Imamo Saga Log sa sljedećim porukama:
  - Akcije: početak (B), kraj (E), prekid (A)
  - Akcije nad: saga (S), transakcija (T), kompenzirajuća transakcija(C)
- Primjeri oznaka:
  - BT2 – početak transakcije 2
  - EC1 – kraj kompenzirajuće transakcije 1
  - AS – prekid sage

# Uspješna saga

- Npr. u sagi imamo 3 transakcije: T1, T2, T3
- Log uspješne sage:
  - BS
  - BT1
  - ET1
  - BT2
  - ET2
  - BT3
  - ET3
  - ES

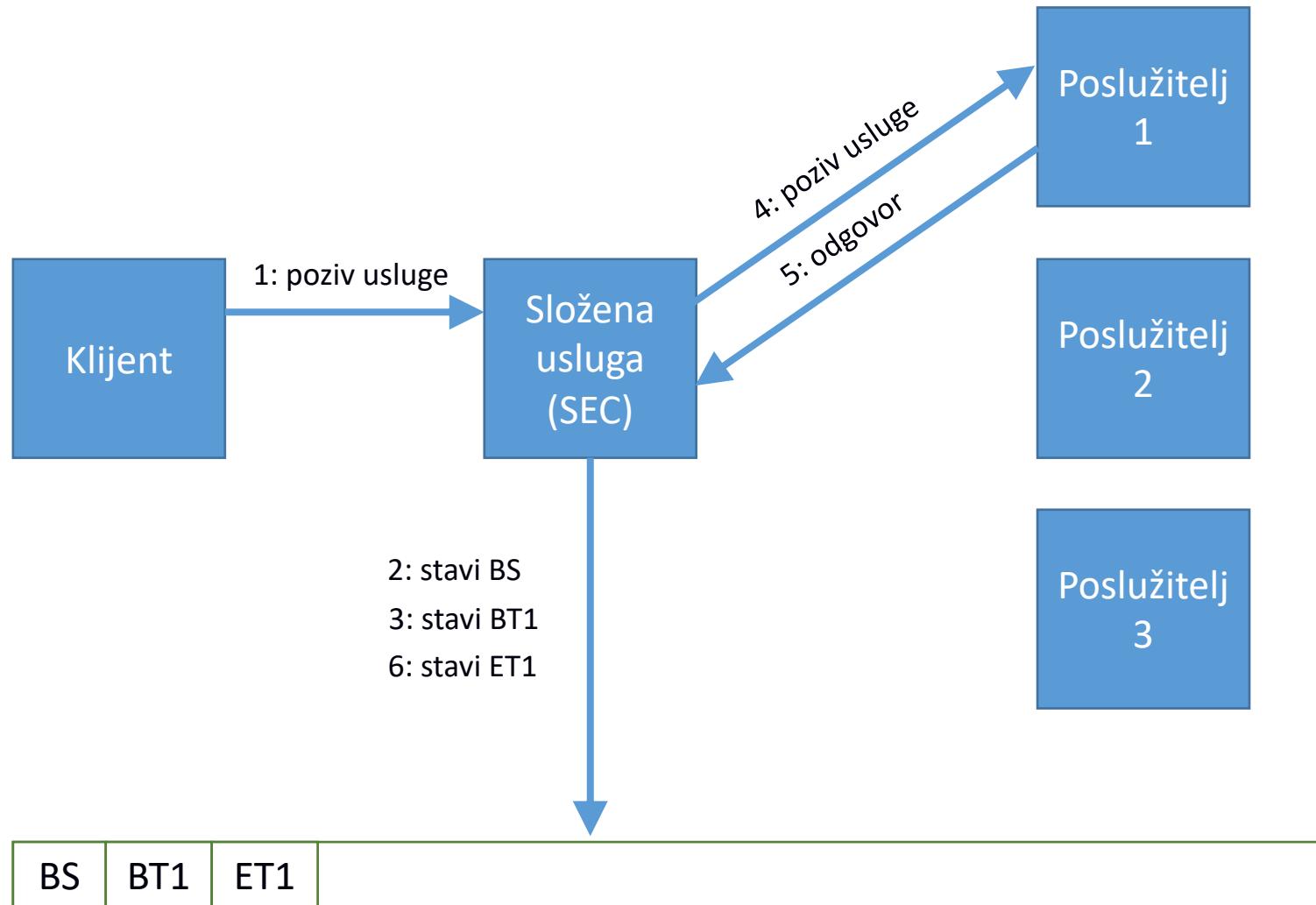
# Neuspješna saga

- Nekoliko načina oporavka:
  - Unatrag, unaprijed, ...
- Ovdje je oporavak unatrag (najčešće se koristi)
- Log neuspješne sage:
  - BS
  - BT1
  - ET1
  - BT2
  - **AS** – transakcija T2 se nije mogla izvršiti
  - **BC1**
  - **EC1**
  - **ES**

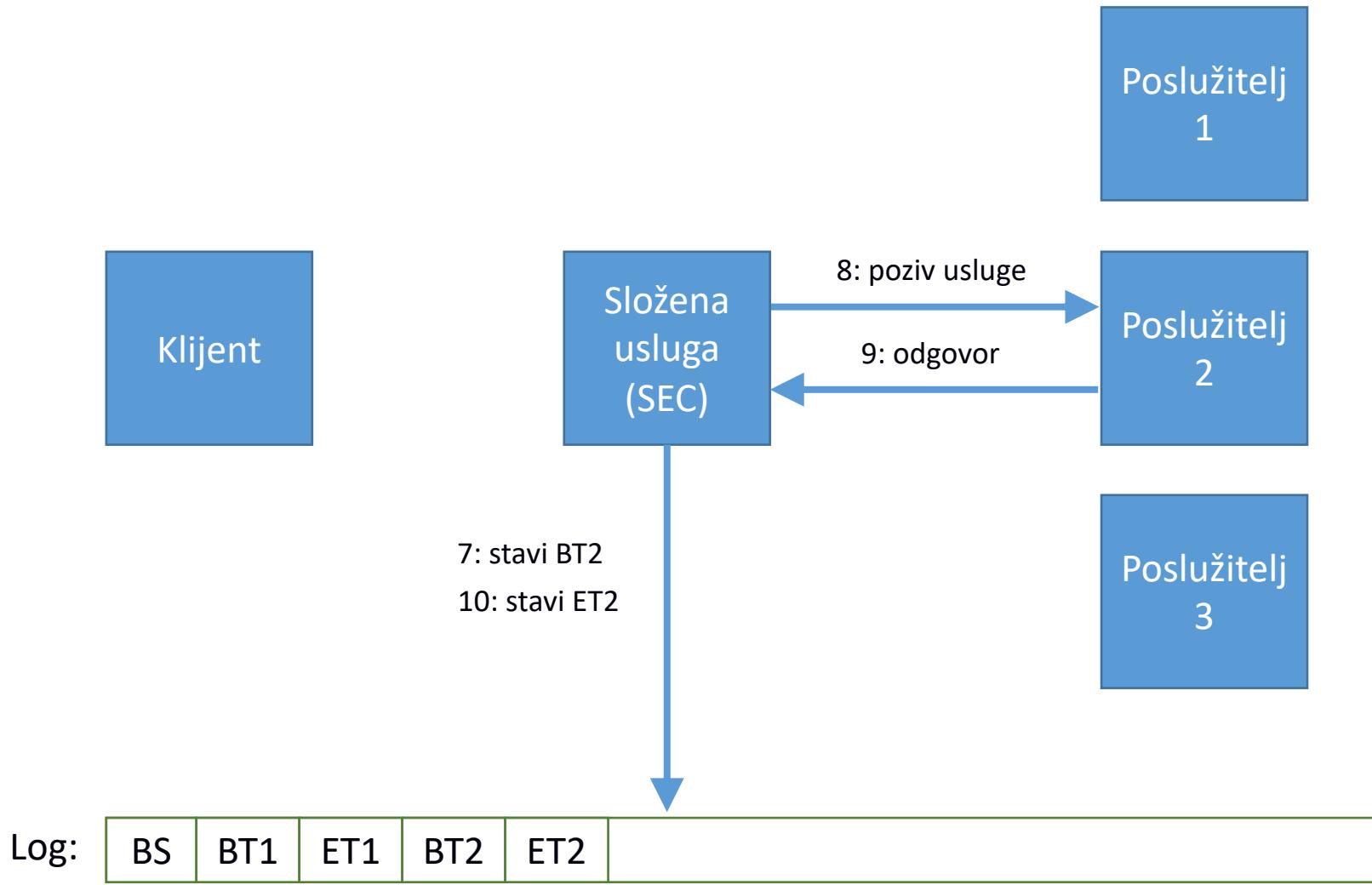
# Raspodijeljena saga

- Semantika je malo drugačija jer nemamo transakcije, nego zahtjeve
  - Obrada zahtjeva može i ne mora podržavati ACID. To ovisi o usluzi koju koristimo.
- Saga Log mora biti raspodijeljen i izdržljiv: npr. RabbitMQ, Kafka, ESB, Blok lanac, ...
  - Zahtjev se treba isporučivati najviše jednom
  - Kompenzirajući zahtjev se mora isporučiti najmanje jednom
- SEC (*Saga Execution Coordinator*) je poseban proces
  - Upravlja sagom
  - Ne pamti stanje, to je u logu
  - Može se pokvariti u bilo kojem trenutku
- Kompenzirajući zahtjevi moraju biti idempotentni i moraju se moći izvršiti.
- Nema atomarnosti niti izolacije

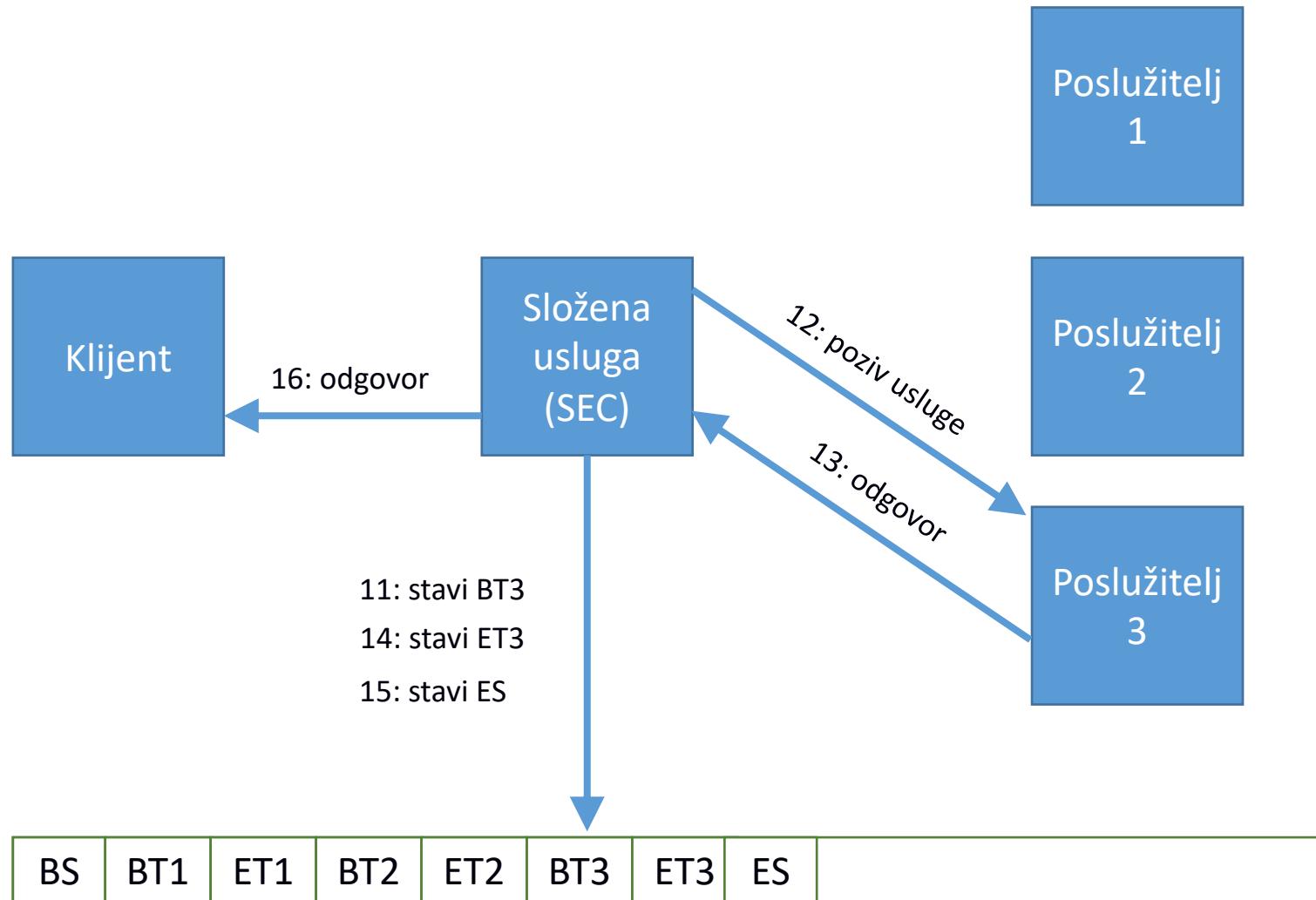
# Uspješna raspodijeljena saga (1)



# Uspješna raspodijeljena saga (2)



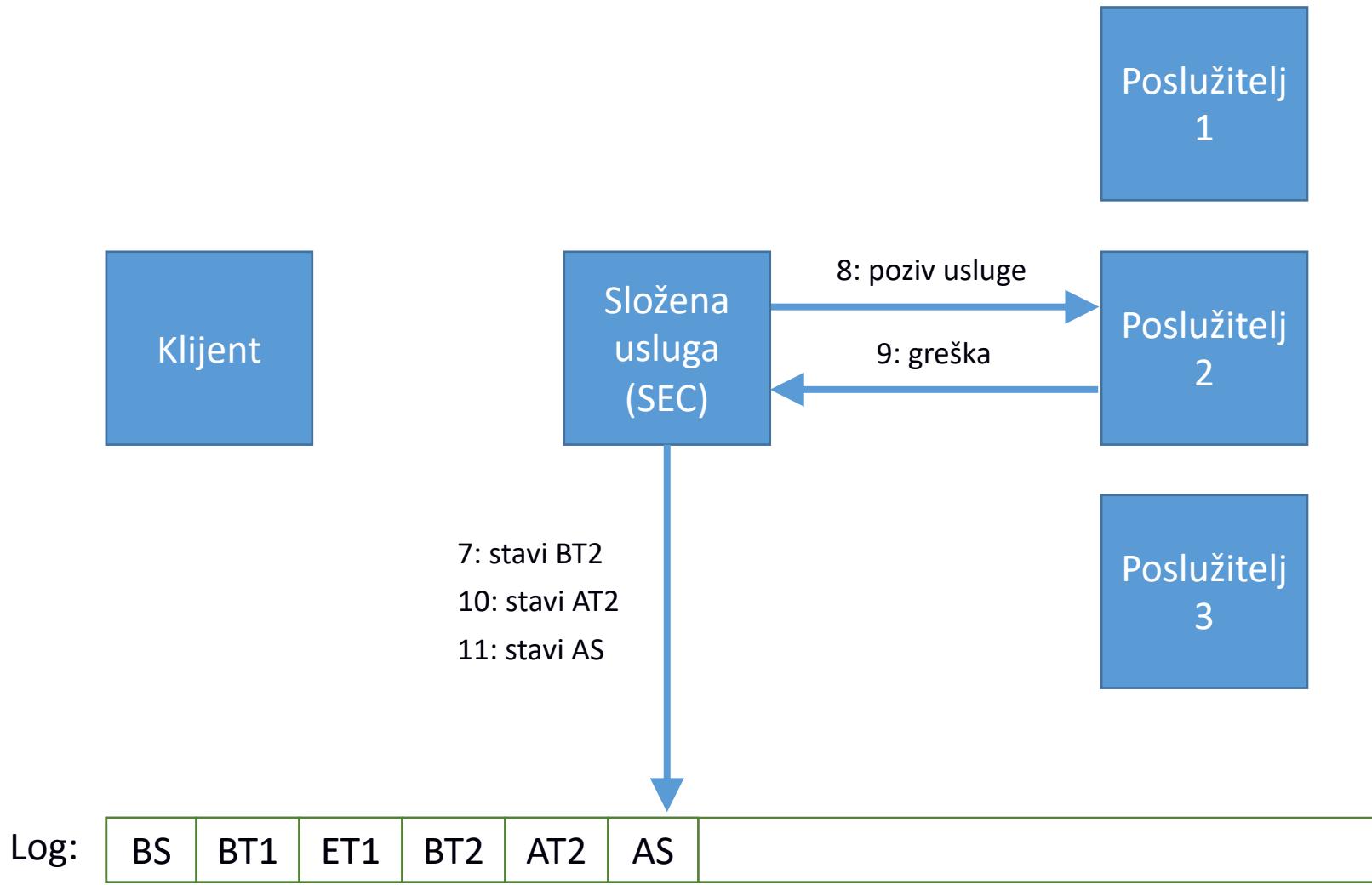
# Uspješna raspodijeljena saga (3)



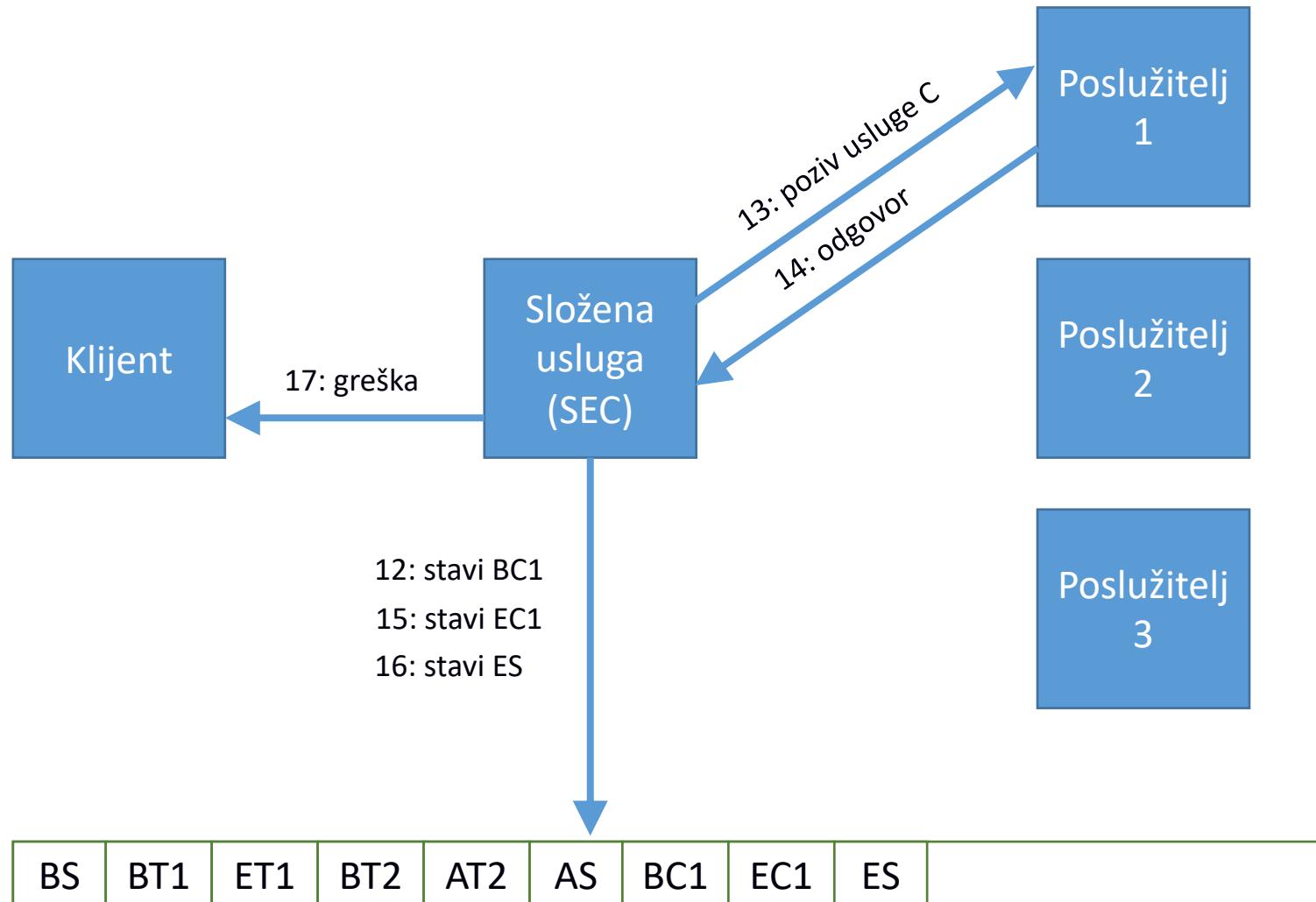
# Prekidi sage

- Korisnik prekida sagu
- Poslužitelj ne može izvršiti upit
- Poslužitelj se sruši
- SEC se sruši (nesigurno stanje)

# Poslužitelj 2 ne može izvršiti upit (1)



# Poslužitelj 2 ne može izvršiti upit (2)



# Što kada kompenzirajući zahtjev ne prođe?

- Zato kompenzirajući zahtjev mora biti idempotentan
- Ponavlja se kompenzirajući zahtjev dok se ne izvrši
- Ako je sistemska greška pa nikad kompenzirajući zahtjev neće proći treba ugraditi mehanizam vremenskog ograničenja.

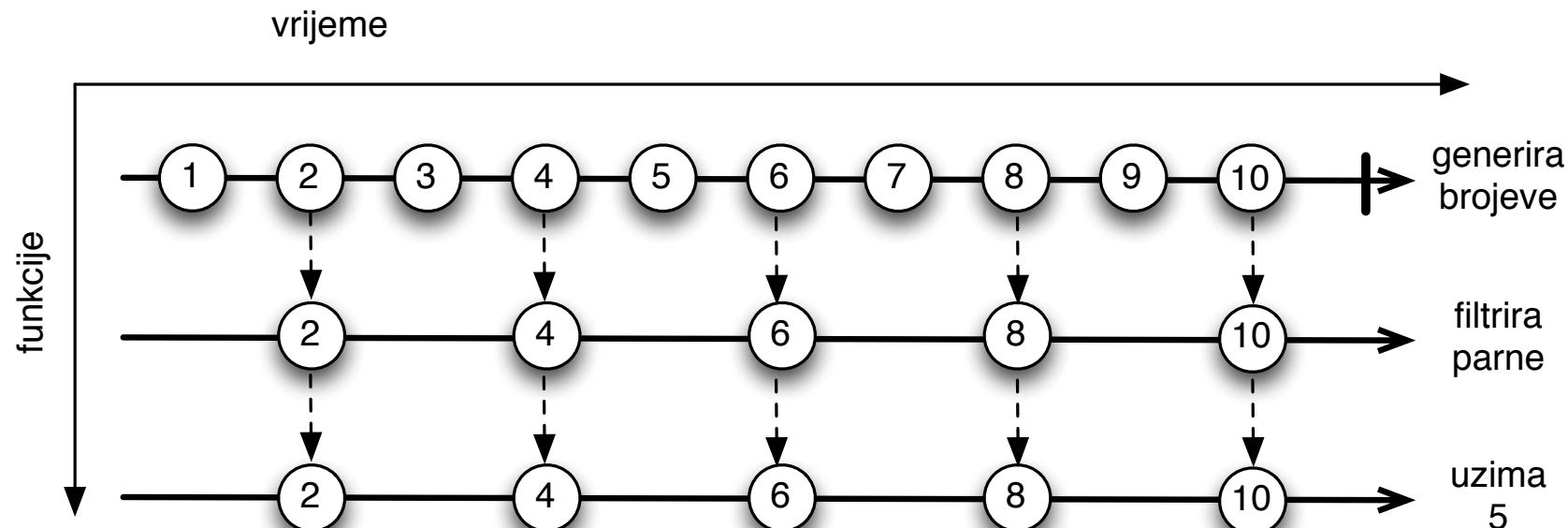
# Što kada se sruši SEC?

- Ponovno se pokreće SEC koji analizira log sage
- Analiza loga:
  - Ako su svi zahtjevi u logu započeli i završili (imamo B i E poruke u logu), SEC se nalazi u sigurnom stanju
    - Staviti AS i započeti s kompenzirajućim zahtjevima za izvršene zahtjeve ili
    - Nastaviti sa izvršavanjem ostalih zahtjeva
  - Inače se nalazi u nesigurnom stanju
    - Staviti AS
    - Započeti s kompenzirajućim zahtjevima za sve zahtjeve u logu uključujući i zahtjeve koji nemaju poruke o kraju

# Trendovi i smjernice

# Reaktivno programiranje

- Ideja: varijable koje se mijenjaju kroz vrijeme promatramo kao tok podataka
- Funkcije definiraju relacije između varijabli i kada se neka promijeni pokreće se izvršavanje funkcija
- Koristi se modificirani oblikovni obrazac *Observer*
  - Ima povratnu vezu
- Modeliranje se vrši pomoću [Marble dijagrama](#)



# Reaktivno programiranje

- Kako je nastalo?
  - Rx.NET - Microsoft Research 2007.-2009. (Jafar Husain)
    - Odgovor na složenost asinkrone obrade
  - Netflix - 2011. Jafar prelazi u Netflix
    - 2013. objavljen RxJava kako otvoreni kod
  - Inicijativa [Reactive Streams](#) (2013.) – Netflix, Pivotal, Typesafe
    - Definirali sučelja (2015.)
    - Implementacije - JSR 166 (Java 9), RxJava 2.0, Project Reactor (Pivotal),...
- Gdje se koristi?
  - Biblioteke: Rx (Reactive Extension) .NET, RxJava (Java), Reactor Project (Spring Java), Typesafe web framework, Bacon.js (javascript), RxJS (javascript)
  - Sustavi: Hystrix, node.js, različite noSQL baze imaju takve *drive*re
- Više o tome na kolegiju: Konkurentno programiranje

# Računarstvo na rubu

- Računarstvo na rubu (*edge computing*)
  - seli obradu podataka bliže izvoru (gdje se podaci generiraju) tj. što bliže rubu sustava
  - rub ima različite „definicije“ ovisno o tome tko govori o tome:
    - za oblak je rub sve što je izvan oblaka
    - za Cisco je rub usmjeritelj na rubu mreže
    - za telekomunikacijske kompanije je rub obrada na baznim stanicama
    - za IoT je rub krajnji uređaj ili mrežni prilaz (gateway) ili računala u industrijskom postrojenju
- svojstva:
  - smanjuje se količina podataka koji se prenose mrežom jer se ne moraju svi podaci slati u oblak
  - povećana sigurnost podataka jer podaci ne izlaze iz lokalne mreže (GDPR)
  - brže vrijeme odziva jer je manje kašnjenje u mreži

Više: [From Cloud Computing to Edge Computing](#), [Eclipse ioFog](#), [Edge computing - the way forward for Eclipse IoT](#)

# Računarstvo u magli

- Računarstvo u magli (*fog computing*)
  - standard koji definira kako bi računarstvo na rubu trebalo raditi
  - OpenFog Consortium započeo standardizaciju 2015.
    - osnivači: Cisco, Intel, Microsoft, Princeton University, Dell i ARM
    - 2016. objavljaju „OpenFog Reference Architecture“ white paper, a 2017. standard
    - 2018. IEEE Standards Association prihvaćaju arhitekturu
    - 2019. se pridružuju Industrial Internet Consortium
      - bave se standardizacijom u području IIoT (Industrial IoT) i Industry 4.0
- primjenjuje koncepte iz oblaka na mrežu uređaja na rubu
- to je horizontalna arhitektura koja raspoređuje i upravlja resursima i uslugama računanja, spremanja podataka, kontrole i umrežavanja
- više: Eclipse fog05,

# Put prema brzim podacima – *fast data*

- Tri faze:
  1. Obrada velike količine podataka “preko noći” – “*data at rest*”
    - Obrada podataka u pozadini
    - Kašnjenja rezultata u satima
    - Npr. Hadoop
  2. Reakcija u realnom vremenu – “*data in motion*”
    - Tok podataka koji se obrađuje u sekundama
    - Rezultat se vraća natrag u sustav i koristi se za druge stvari
    - Potrebne hibridne arhitekture s dva sloja:
      - “*speed layer*” – za odgovor u realnom vremenu
      - “*batch layer*” – za obradu u pozadini
    - Npr. lambda arhitekture kao što su AWS Lambda, ...
  3. Samo “*data in motion*”
    - Kompletno izbacivanje obrade u pozadini
    - “Čista” obrada tokova podataka
    - Rasподijeljeni sustavi kao što su: Flink, Spark streaming, Google Cloud Data flow
- Više o tome na kolegijima:
  - Rasподijeljena obrada velike količine podataka
  - Analiza velikih skupova podataka

# Literatura

- [1] K. Hwang, J. Dongarra, and G. C. Fox, "Distributed and cloud computing: From parallel processing to the internet of things," Morgan Kaufmann, 2011.
- [2] N. R. Adiga, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, et al., "An Overview of the BlueGene/L Supercomputer," SC2002, Baltimore, USA, 2002, p. 60.
- [3] <http://www.top500.org/>
- [4] <http://en.wikipedia.org/wiki/Folding@home>
- [5] S. Venugopal, R. Buyya, K. Ramamohanarao: "A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing", ACM Computing Surveys
- [6] Jonas Bonér: "Reactive Microsystems - The Evolution of Microservices at Scale", O'Reilly, 2017., [technical report](#)



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**8. Otpornost na neispravnosti u  
raspodijeljenom okružju**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

- Uvod, definicije pojmova
- Otpornost procesa na ispade
- Sporazum skupine procesa
- Pouzdana komunikacija skupine procesa
- Raspodijeljeno izvršavanje operacije
- Oporavak nakon ispada

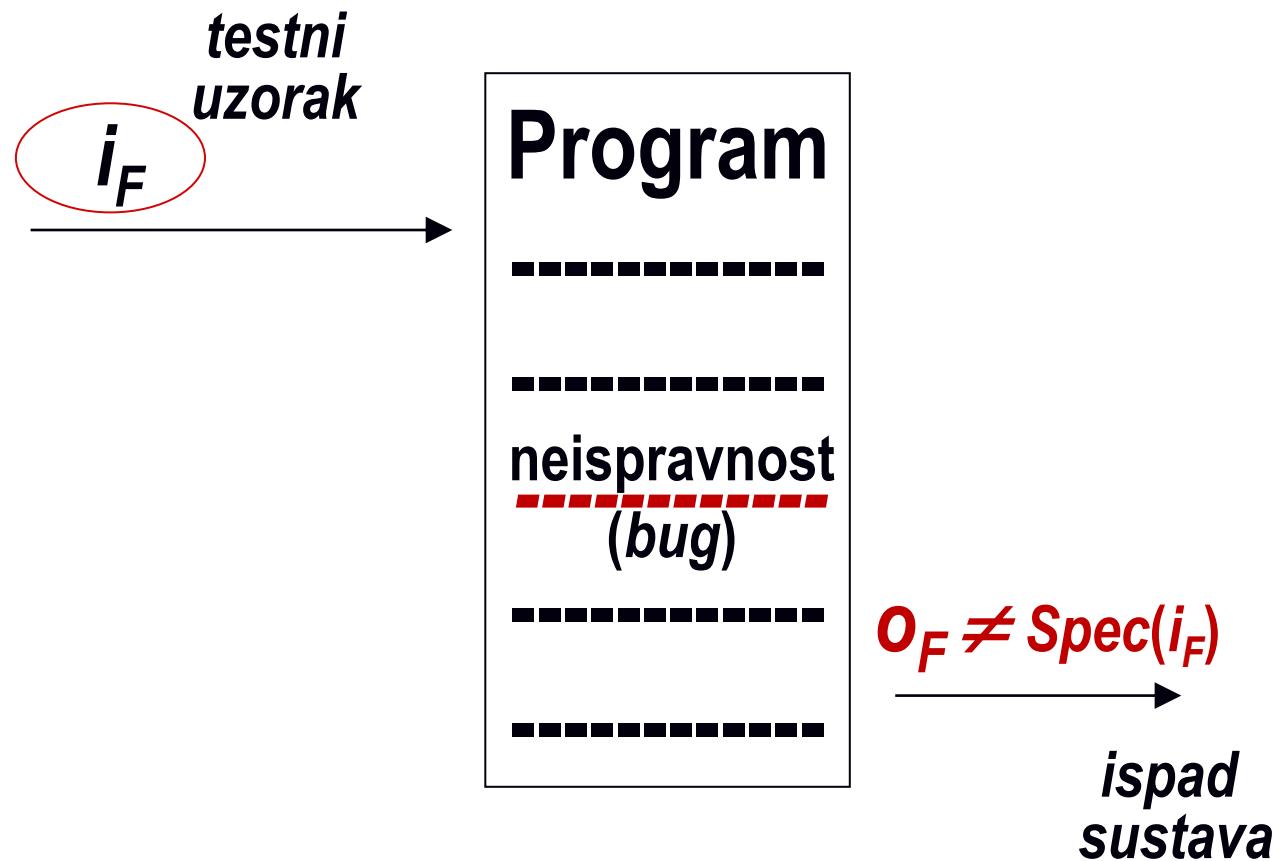
# Otpornost na neispravnosti

*Fault tolerance (provide service even in the presence of faults)*

- sposobnost sustava za obavljanje definirane usluge bez obzira na postojeće **neispravnosti** koje izazivaju **ispad** nekih komponenti sustava
- osobina sustava, može prikriti ispad komponenti raspodijeljenog sustava, definiraju se procedure za oporavak sustava
- funkcionalnost sustava može biti ograničena uz negativan utjecaj na njegove performance

Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”

# Podsjetimo se izvođenja programskog koda...



# Osnovni pojmovi

- Neispravnost (*fault*)
  - npr. dio programskog koda (*bug*), neispravan komunikacijski kanal, pogreške prilikom oblikovanja sustava
  - uzrok ispada sustava, pronalaženje neispravnosti je težak i važan problem
  - prolazne, isprekidane i trajne neispravnosti
- Ispad sustava (*failure*)
  - stanje sustava koje se detektira kroz nemogućnost korištenja jedne ili više njegovih usluga
  - posljedica neispravnosti, signalizira postojanje neispravnosti u raspodijeljenom sustavu

# Vrste ispada u raspodijeljenom okruženju

- Ispad kanala
  - proces  $p$  je poslao poruku procesu  $q$ , ali  $q$  poruku ne prima, jer npr. kanal gubi poruke
- Ispad procesa
  - ispad zaustavljanja (*stoping failure*): proces ne mijenja stanja (ne izvode se prijelazi) premda se ne nalazi u završnom stanju
  - bizantinski ispad (*Byzantine failure*): proces generira proizvoljne izlaze

# Vrste ispada procesa

| Vrsta ispada                                                                       | Opis                                                                                                                                   |
|------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Ispad procesa<br><i>ispad zaustavljanja</i>                                        | Proces neočekivano ulazi u stanje zaustavljanja i ne odgovara na nove zahtjeve.                                                        |
| Pogreška u komunikaciji<br><i>pogreška primanja</i><br><i>pogreška slanja</i>      | Proces ne odgovara na primljeni zahtjev.<br>Proces ne prima zahtjev.<br>Proces ne šalje odgovor.                                       |
| Vremenska pogreška                                                                 | Proces šalje odgovor nakon isteka vremenskog roka.                                                                                     |
| Pogrešan odgovor<br><i>sadržaj</i><br><i>pogrešna promjena stanja poslužitelja</i> | Generirani odgovor je neispravan.<br>Sadržaj odgovora je neispravan.<br>Poslužitelj ulazi u pogrešno stanje nakon primljenog zahtjeva. |
| “Bizantska pogreška”                                                               | Proces proizvodi proizvoljan odgovor u proizvoljnem trenutku.                                                                          |

# Redundancija

Ključna tehnika za prikrivanje neispravnosti raspodijeljenog sustava.

Primjeri redundancije:

- redundancija informacija
  - npr. Hammingov kod
- vremenska redundancija
  - ponavljanje neke operacije u vremenu
- fizička redundancija
  - dodavanje dodatne opreme (npr. vruća rezerva)  
ili procesa u sustav (repliciranje procesa)

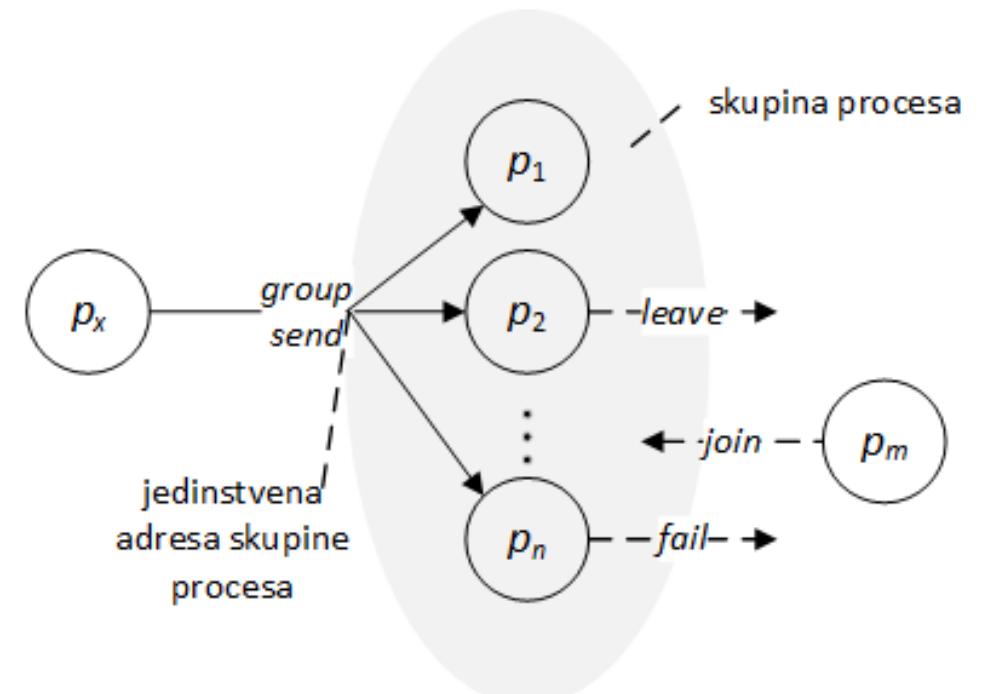


# Sadržaj predavanja

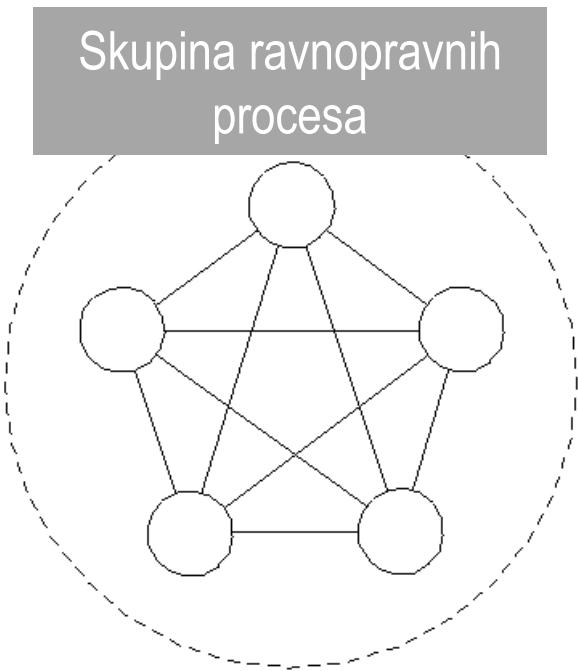
- Uvod, definicije pojmova
- **Otpornost procesa na ispade**
- Sporazum skupine procesa
- Pouzdana komunikacija skupine procesa
- Raspodijeljeno izvršavanje operacije
- Oporavak nakon ispada

# Otpornost procesa na ispade: skupina procesa

- Zamjena jednog procesa skupinom identičnih procesa
  - replikacija procesa radi prikrivanja ispada jednog ili k procesa
  - poruka se šalje skupini procesa, svi procesi uspješno primaju ili ne primaju poruku
  - skupine procesa su dinamične
  - jedan proces može biti član više skupina
  - administracija skupine procesa

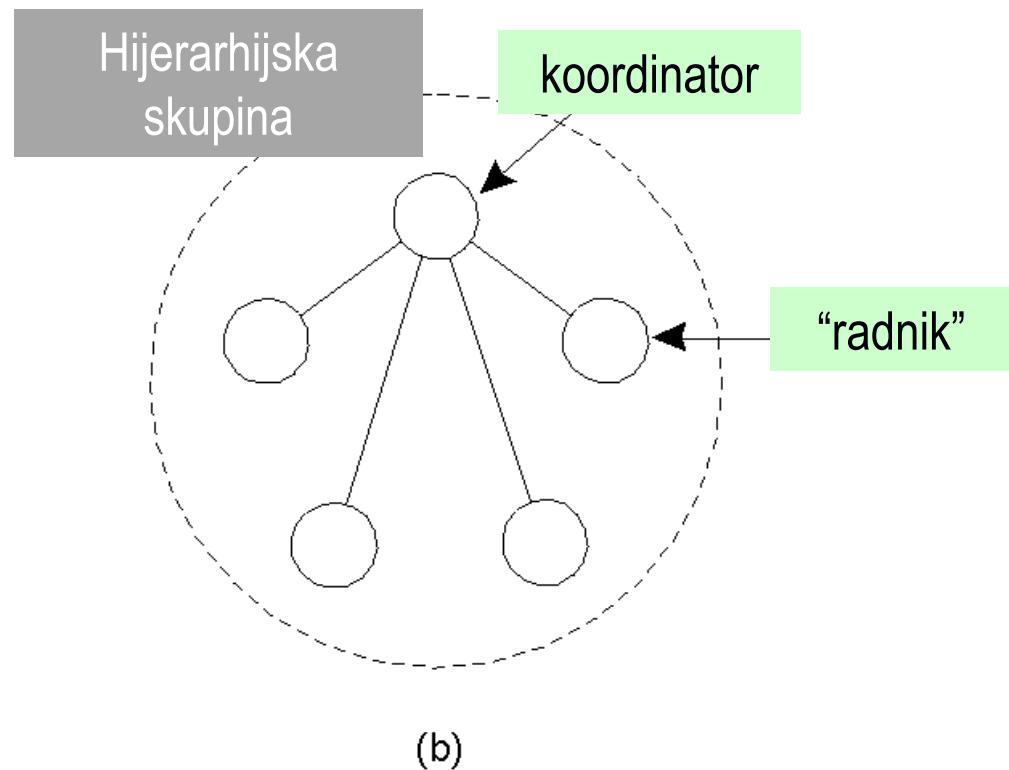


# Organizacija skupine procesa



(a)

Prednost?  
Nedostatak?



(b)

Prednost?  
Nedostatak?

# Otkrivanje ispada procesa iz skupine

U slučaju ravnopravnih procesa u skupini

- svaki proces periodički šalje upit ostalim procesima i provjerava njihovo stanje (“*are you alive?*”)
- proces pasivno čeka i prati poruke koje prima od ostalih članova skupine

U hijerarhijskoj skupini procesa koordinator provjerava stanje ostalih članova skupine.

# Koliko procesa u skupini?

## Tolerancija $k$ ispada

- skupina može “preživjeti” ispad najviše  $k$  procesa
- dovoljan je  $k + 1$  proces da se osigura tolerancija na  $k$  ispada (jedan proces može preuzeti poslove skupine)
- potrebno je  $2k + 1$  procesa u skupini ako se prepostavi  $k$  bizantskih ispada (mehanizam glasovanja:  $k + 1$  ispravan proces će “nadglasati”  $k$  neispravnih)

## Postizanje suglasnosti ili sporazuma

- ako se prepostavi  $k$  bizantskih ispada, koliko je ukupno procesa potrebno da bi se postigla suglasnost (npr. otkrilo koji su ispali)?
- *sporazum skupine procesa*

# Sadržaj predavanja

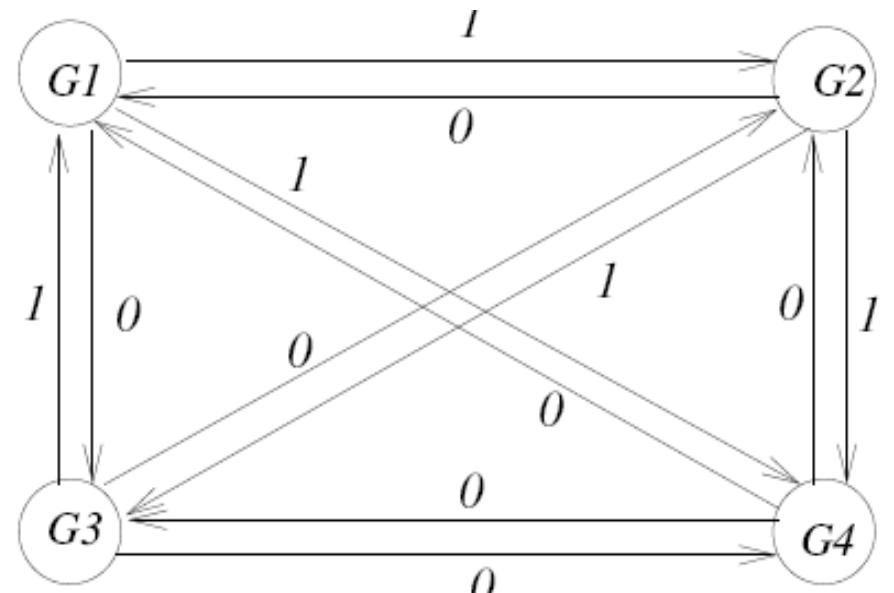
- Uvod, definicija pojmove
- Otpornost procesa na ispade
- **Sporazum skupine procesa**
- Pouzdana komunikacija skupine procesa
- Raspodijeljeno izvršavanje operacije
- Oporavak

# Sporazum skupine procesa

- Procesi trebaju postići sporazum o npr. vrijednosti neke varijable ili o tome hoće li ili neće izvršiti transakciju

Primjer:

- 4 generala trebaju započeti napad koordinirano u isto vrijeme
- za komunikaciju koriste glasnike
- komunikacija je nepouzdana i može trajati prilično dugo
- general izdajnik bi mogao slati pogrešne informacije ostalima



**Mogu li generali postići sporazum o vremenu napada?**

# Problem bizantskog sporazuma (1/2)

*Byzantine agreement problem* [Lamport et al. 1982]

- **Problem:** svaki proces  $i$  definira inicijalnu vrijednost  $v_i$ , a skupina procesa treba postići sporazum o nizu vrijednosti za svaki proces iz skupine
- **Sporazum:** Svi ispravni procesi prihvaćaju isti niz vrijednosti  $[v_1, v_2, \dots, v_n]$ .
- **Ispravnost:** Ako je proces  $i$  ispravan i definira vrijednost  $v_i$ , svi ostali ispravni procesi prihvaćaju vrijednost  $v_i$  kao  $i$ -ti element niza. Ako je proces neispravan, ostali ispravni procesi mogu prihvatiti bilo koju vrijednost za taj proces.
- **Završetak:** Svaki ispravan proces će u konačnici prihvatiti vrijednosti niza za ispravne procese.

# Problem bizantskog sporazuma (2/2)

## Prepostavke:

- $n$  procesa
- $k$  neispravnih procesa (**bizantski ispad**)
- proces  $i$  šalje vrijednost  $v_i$ , ostalim procesima u skupini
- svaki proces treba kreirati niz  $V$  duljine  $n$  takav da vrijedi

$$V[i] = v_i$$

Ponašanje procesa: **sinkrono** ili **asinkrono**

Slanje poruka: **jednoodredišno**, **višeodredišno**

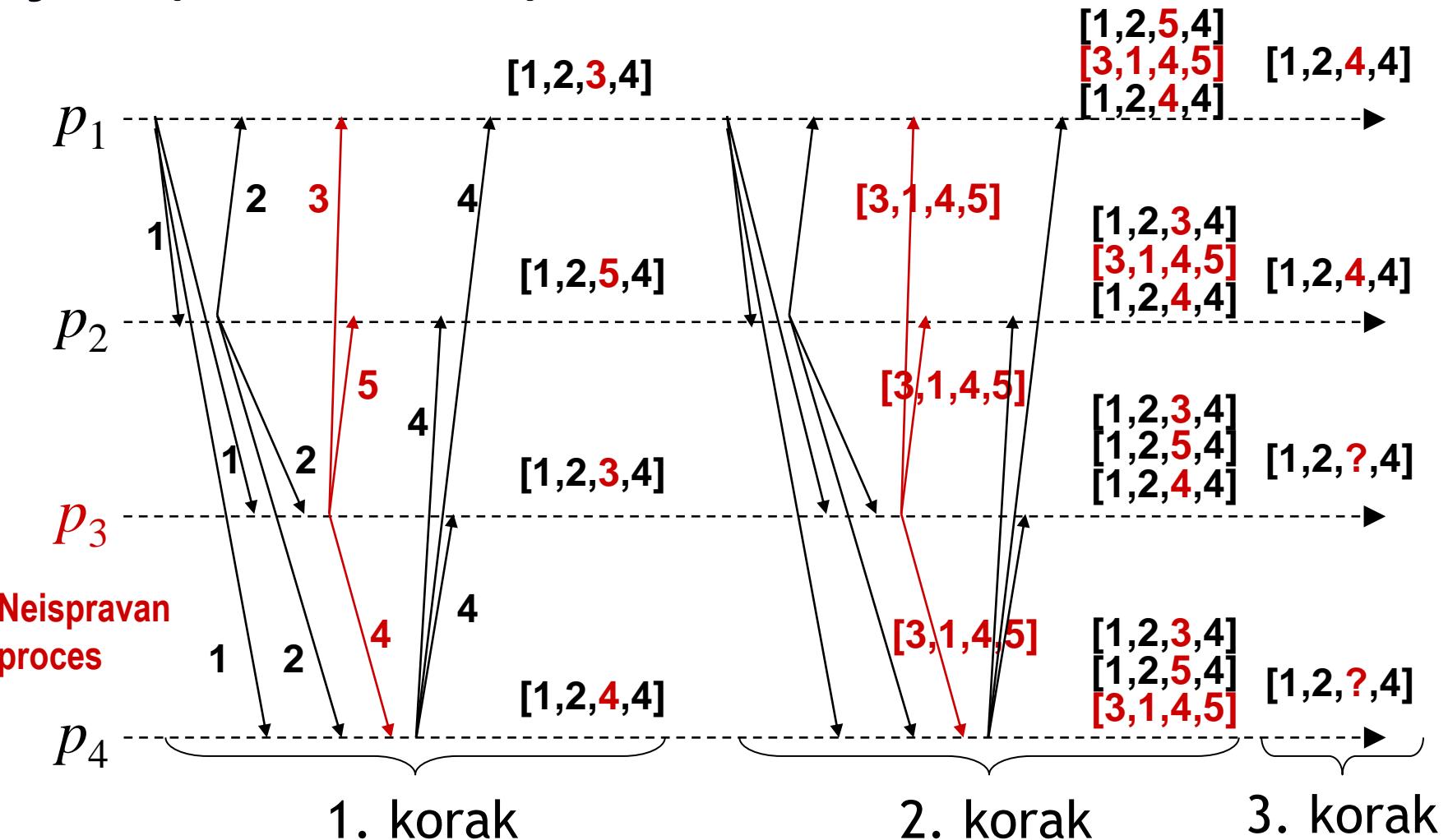
Kašnjenje poruka: **ograničeno**, **neograničeno**

Isporuka poruka: **uređena**, **neuređena**

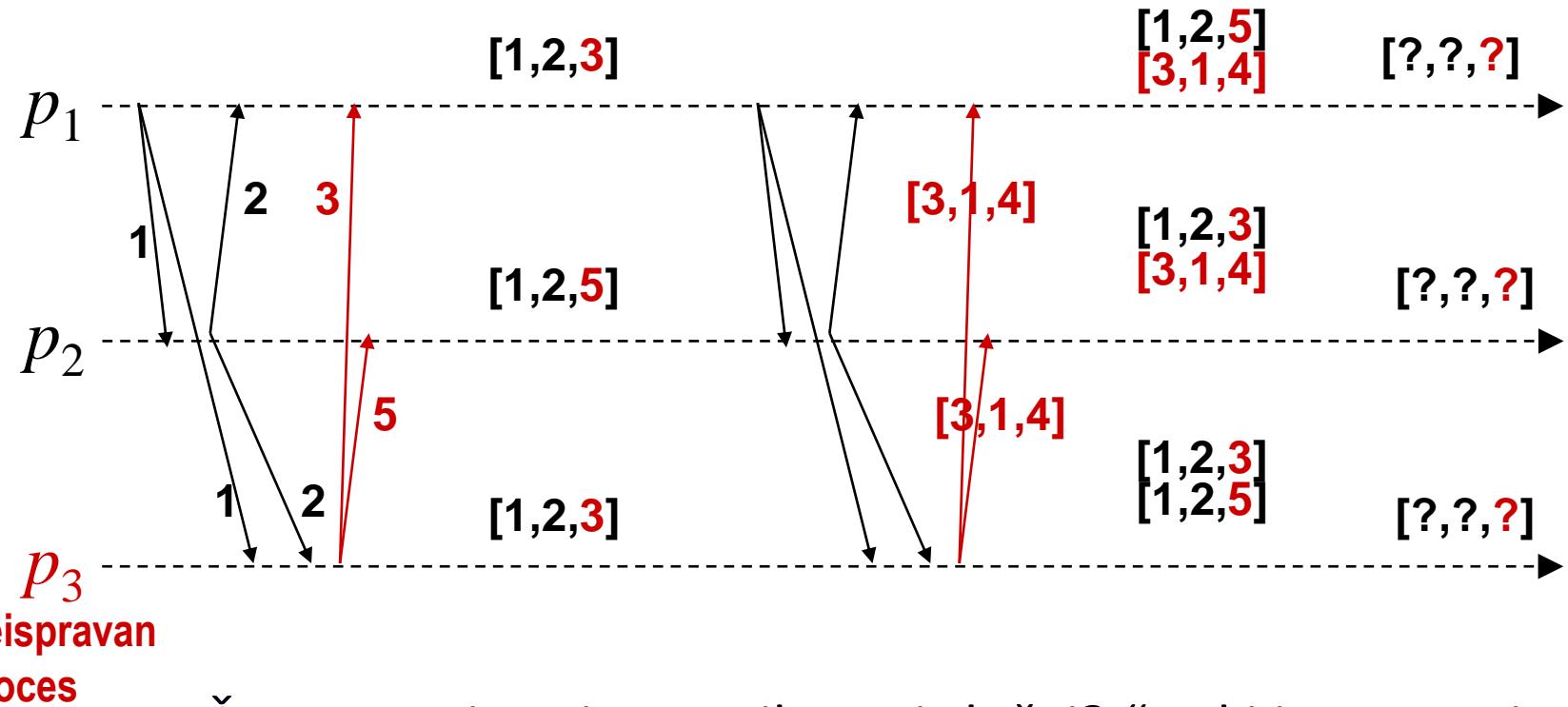
# Skica algoritma za k =1

- **1. korak:**  
Ispravni procesi šalju svoju vrijednost  $v_i$  ostalim procesima, dok neispravni proces šalje proizvoljne vrijednosti.  
Svaki proces prikuplja vrijednosti i sprema ih u  $\mathbf{V}$ .
- **2. korak:**  
Svaki proces šalje ostalima svoj  $\mathbf{V}$ , dok neispravni proces šalje proizvoljni  $\mathbf{V}$ .
- **3. korak:**  
Konačno, svaki proces uspoređuje sve primljene nizove i odlučuje se za većinske vrijednosti.
- Procesi mogu donijeti odluku samo o vrijednostima za ispravne procese!

# Primjer ( $n=4$ , $k=1$ )



# Primjer ( $n=3$ , $k=1$ )



Što procesi znaju za prikazani slučaj? “Neki je proces ispaо!”

Mogu li procesi donijeti odluku za prikazani slučaj?

# Rasprava

- Sporazum u slučaju  $n$  procesa i  $k$  neispravnih procesa u bizantskom ispadu može se postići samo u slučaju **sinkronog sustava** i kada vrijedi

$$k \leq \left\lfloor \frac{n-1}{3} \right\rfloor$$

(za  $k=1$  i  $n=4$  vrijedi, za  $k=1$  i  $n=3$  ne vrijedi)

- Za sporazum u slučaju  $k$  neispravnih procesa, potrebno je  $2k + 1$  ispravnih, tj. ukupno  $n = 3k + 1$  procesa (više od 2/3 procesa treba biti ispravno!)
- Dokazano je da u slučaju **asinkronog sustava** u kojem se ne može jamčiti isporuka poruka procesi ne mogu postići sporazum niti za  $k=1$

# Sadržaj predavanja

- Uvod, definicija pojmove
- Otpornost procesa na ispade
- Sporazum skupine procesa
- **Pouzdana komunikacija skupine procesa**
- Raspodijeljeno izvršavanje operacije
- Oporavak nakon ispada

# Pouzdana komunikacija skupine procesa (1)

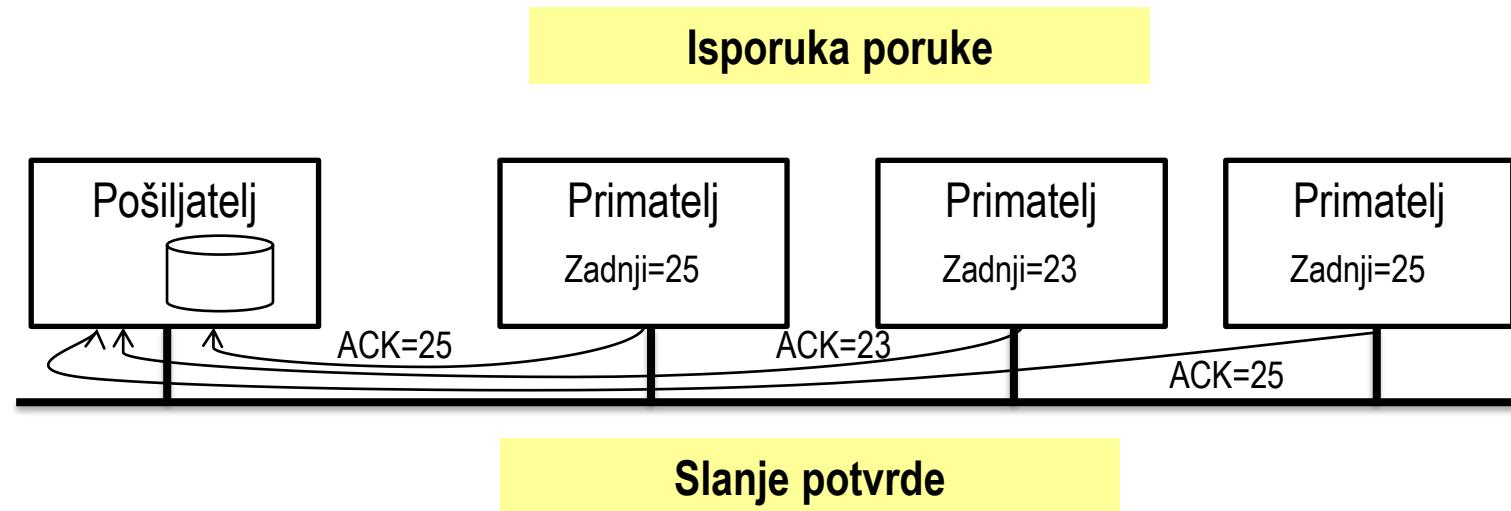
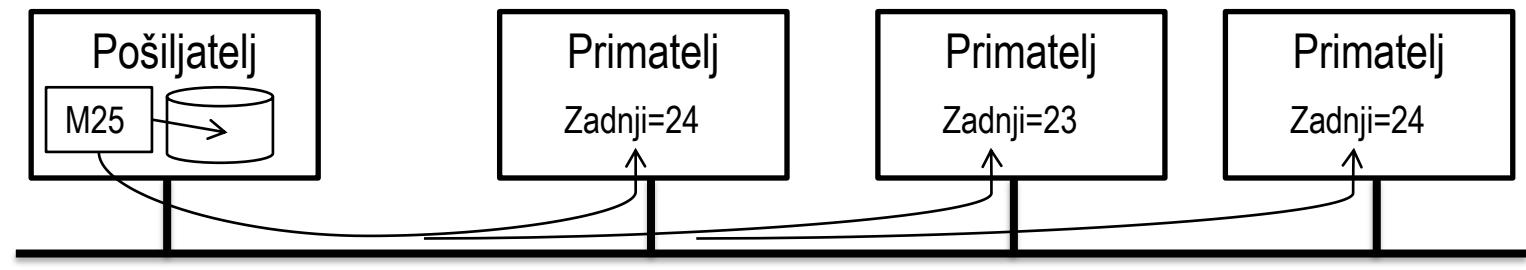
- jamstvo isporuke poruke svim procesima u skupini
- problemi
  - Koji procesi čine skupinu u trenutku slanja poruke?
  - Što se događa ako novi proces ulazi u skupinu procesa dok je isporuka poruke skupini u tijeku?
  - Što se događa ako dođe do ispada pošiljatelja poruke tijekom isporuke poruke ostalim procesima?
  - Što se događa ako jedan od primatelja ispadne tijekom isporuke poruke?

# Pouzdana komunikacija skupine procesa (2)

- najjednostavnija praktična implementacija
  - pouzdana komunikacija između svakog para procesa iz skupine, od točke do točke (npr. TCP)
- učinkovita praktična implementacija
  - pouzdano višeodredišno razašiljanje od jednog prema svim procesima u skupini (*multicast*)
  - skalabilnost

# Pouzdana komunikacija bez mogućih ispada procesa (1/2)

Višeodredišna komunikacija s potvrdom ("nepouzdani kanal")



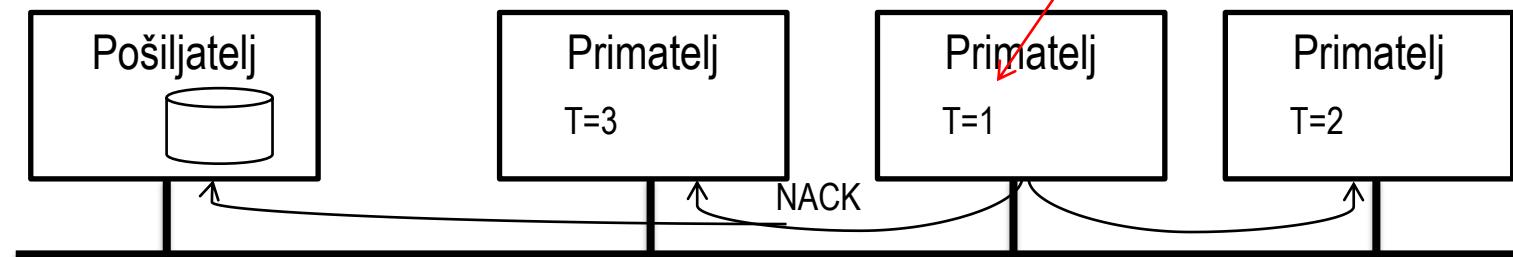
Prema; Tanenbaum, Van Steen

# Pouzdana komunikacija bez mogućih ispada procesa (2/2)

## Potisnuta potvrda:

Višeodredišno razašiljanje jedne negativne potvrde

Slučajno kašnjenje nakon kojeg se šalje negativna potvrda



Višeodredišno slanje negativne potvrde

Prema: Tanenbaum, Van Steen

# Pouzdana komunikacija s ispadima procesa

- jamči isporuku poruke svim ispravnim i dostupnim procesima u skupini ili niti jednom
- potrebno je osigurati i isporuku poruka u određenom redoslijedu

## Notacija

- $p$  - proces
- $G$  – skupina, skup procesa – skupni pogled (*group view*)
- $m$  – generirana poruka
- $vc$  – poruka koja prenosi informaciju o dolasku ili odlasku procesa iz skupine – promjena pogleda (*view change*)

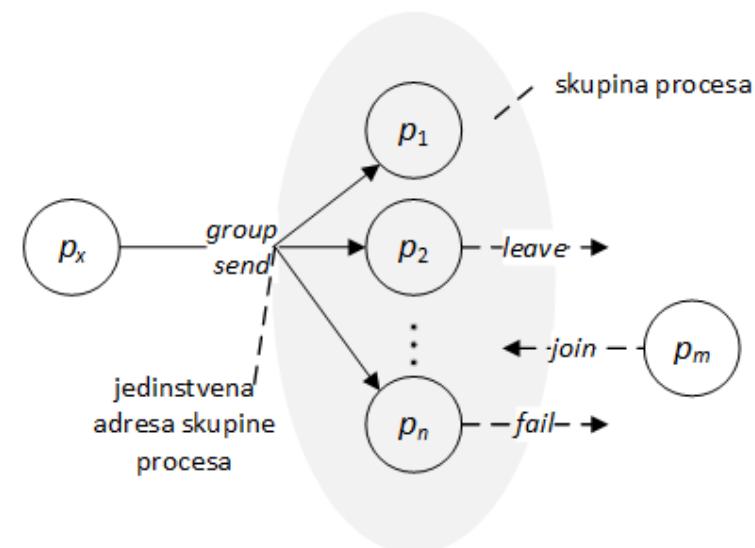
# Virtualna sinkronost

## Scenarij 1

- Proces  $p$  šalje poruku  $m$ , u tome trenutku postoji skupina procesa  $G$
- Tijekom isporuke poruke  $m$  novi proces se uključuje u skupinu i generira se poruka  $vc$  (*view change*) koja se opet šalje svim članovima iz  $G$
- Posljedica: poruke  $m$  i  $vc$  su istovremeno u tranzitu
- 2 moguća rješenja:
  - $m$  isporučen svim članovima  $G$  prije isporuke  $vc$
  - $m$  nije isporučen niti jednom procesu iz  $G$

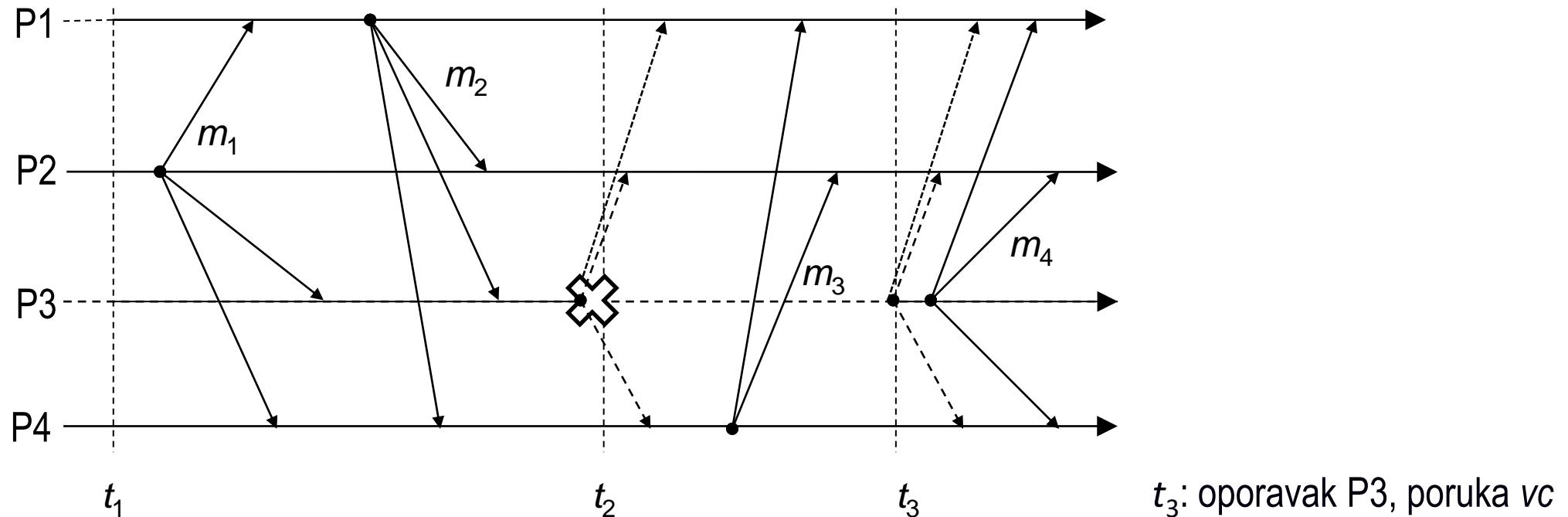
## Scenarij 2

- Ako  $p$  pošalje  $m$  i ispadne prije isporuke  $m$  članovima  $G$ , ostali procesi ignoriraju  $m$  (ne treba osigurati isporuku ostalim ispravnim procesima, kao da je  $p$  ispaо prije slanja  $m$ )



# Primjer virtualne sinkronosti

$t_2$ : ispad P3, generira se poruka vc



Poruka se može isporučiti članovima iz G samo ako ne postoji poruka  $vc$  koja je istovremeno u tranzitu. Implementacija nije trivijalna.

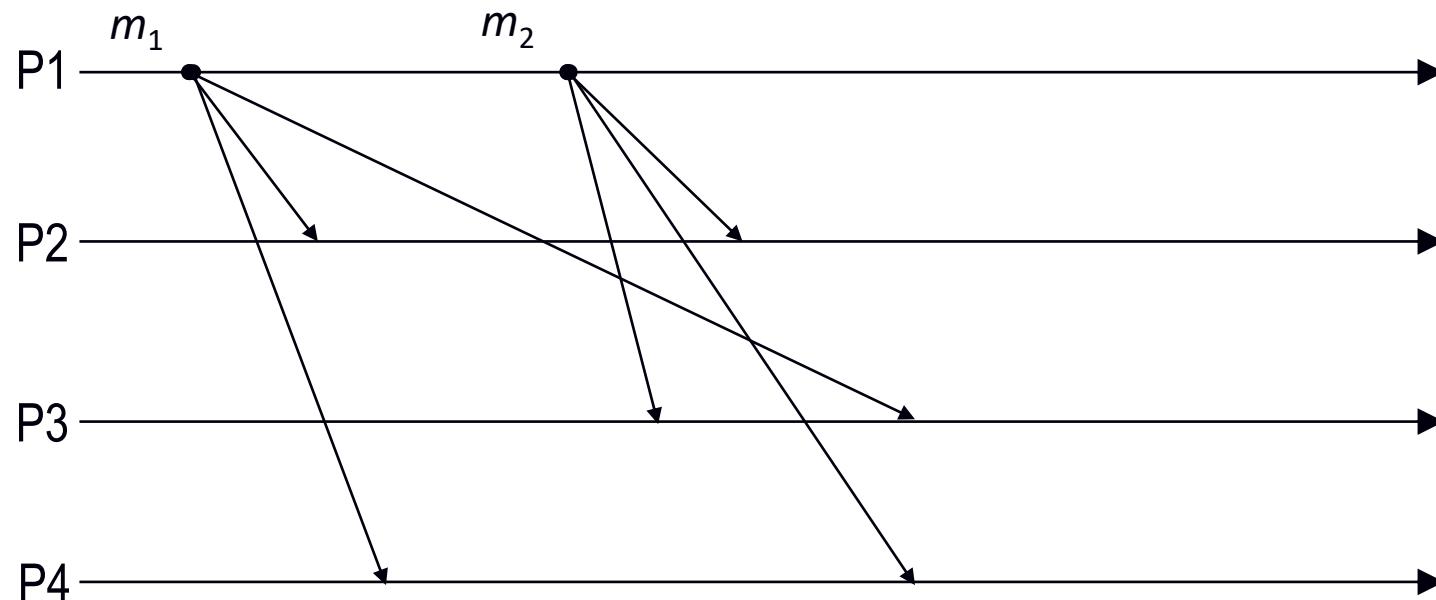
Prema: Tanenbaum, Van Steen

# Pouzdana komunikacija: slijed poruka

- slijed kojim procesi primaju poruke od velike je važnosti, jer utječe na promjene stanja tih procesa
- slijed primljenih poruka može biti:
  - neuređen (*unordered multicast*)
  - FIFO (*FIFO-ordered multicast*)
  - potpuno uređen (*totally-ordered multicast*)

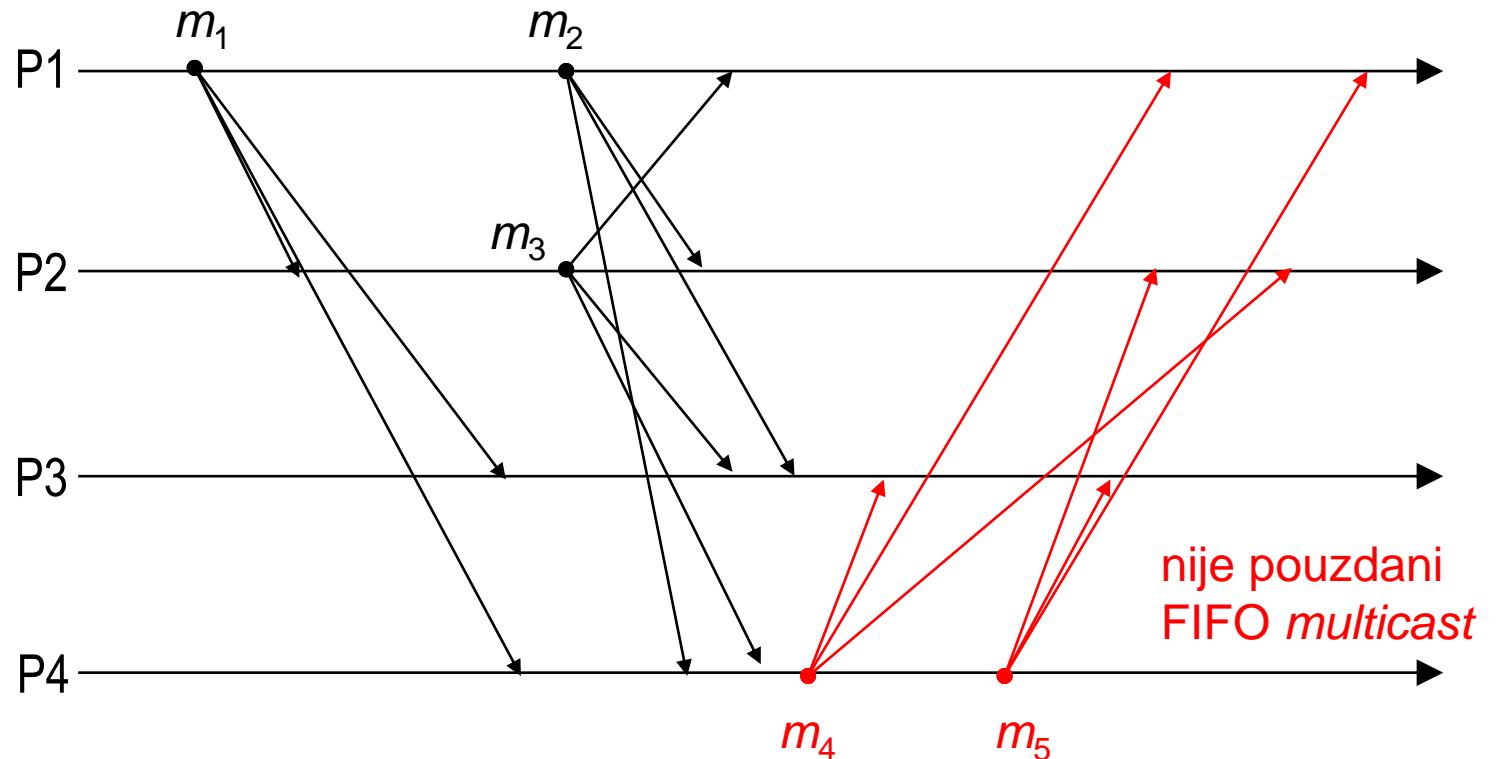
# Neuređeni slijed poruka

- Nije važan redoslijed kojim procesi primaju poruke  $m_1$  i  $m_2$ .
  - Pouzdani neuređeni *multicast* – to je pouzdani *multicast* koji je istovremeno i **virtualno sinkron**.



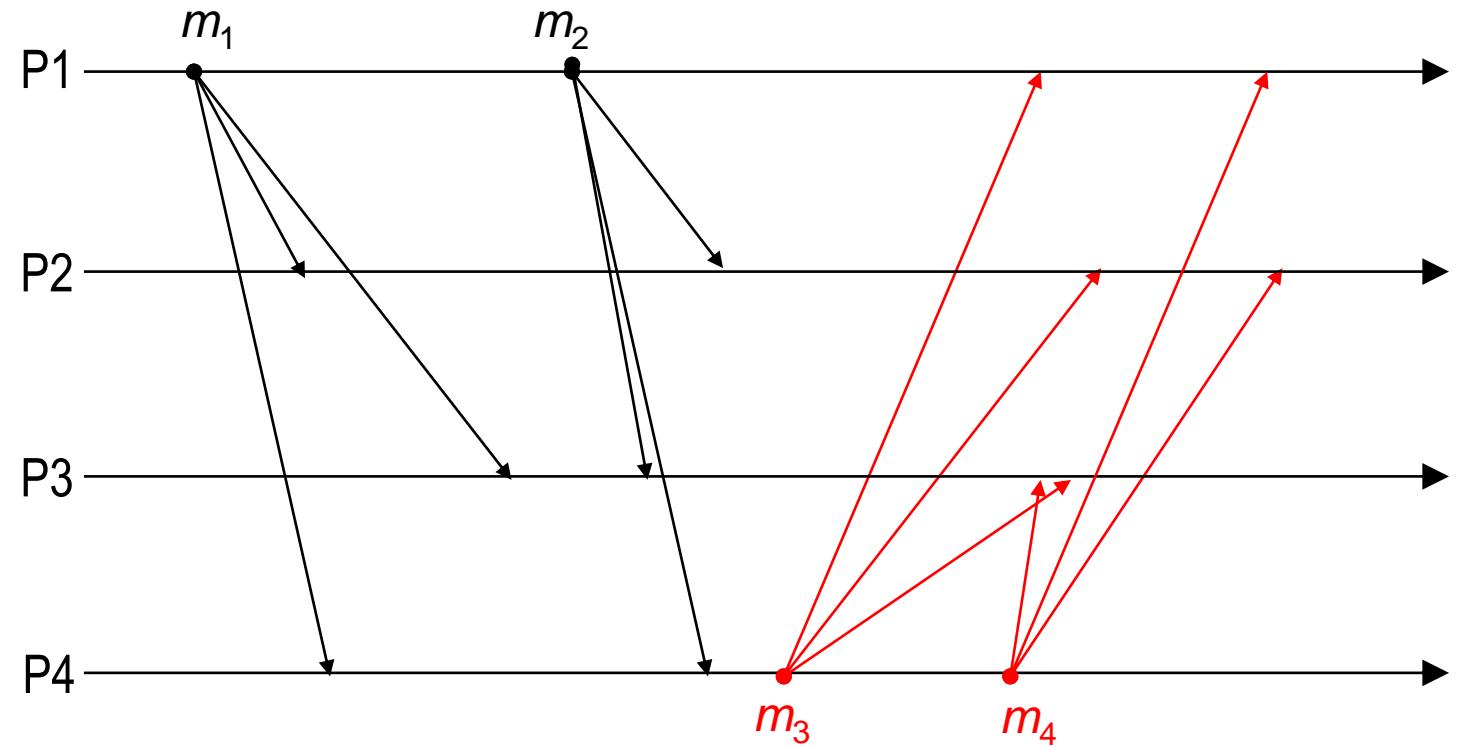
# Slijed poruka FIFO

- Poruke  $m$  koje dolaze od istog procesa moraju se isporučiti u redoslijedu kojim su poslane
  - Pouzdani FIFO *multicast*



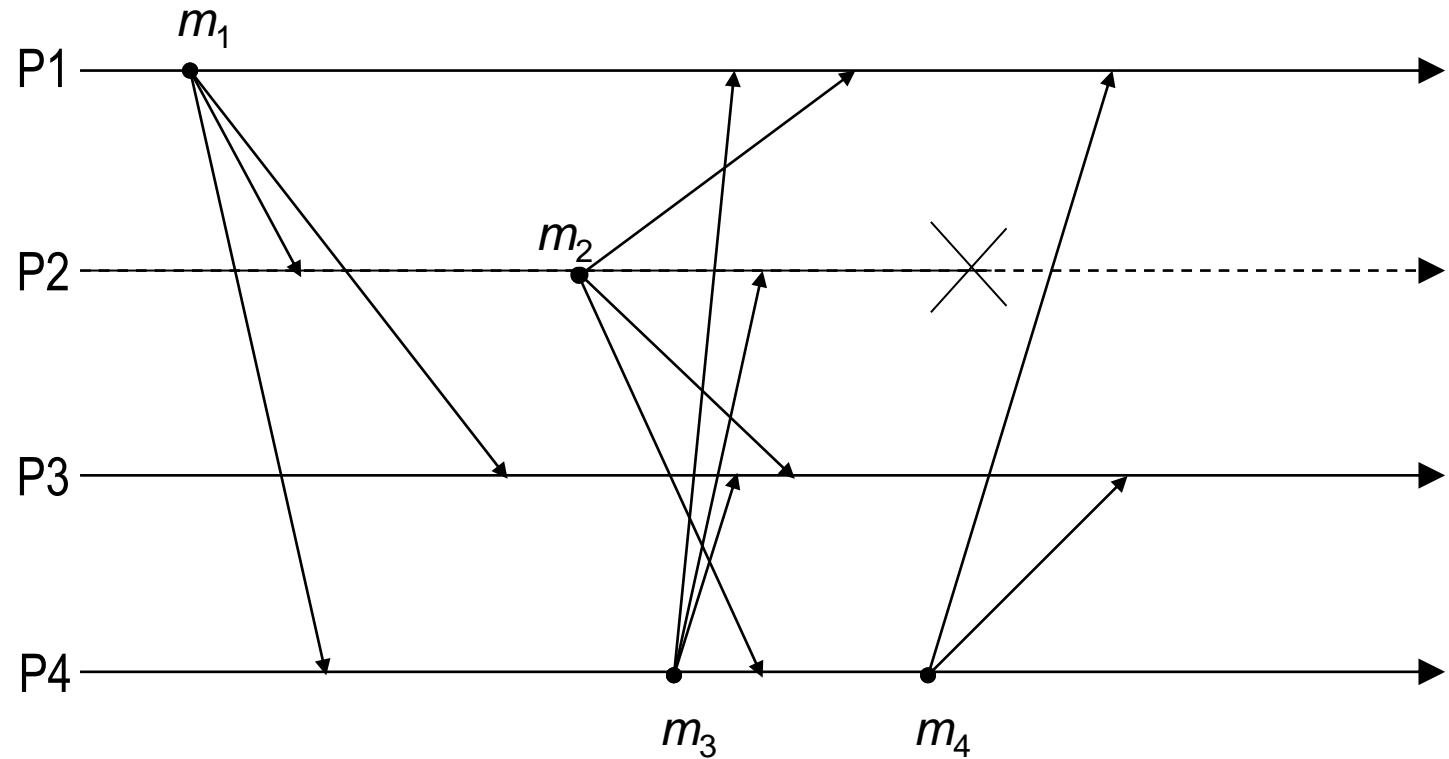
# Potpuno uređen slijed poruka

- Poruke se isporučuju u istom redoslijedu svim procesima u skupini (uz FIFO redoslijed poruka od istog procesa)
  - ***Atomic multicast*** – pouzdani virtualno sinkroni multicast s potpuno uređenim slijedom poruka



# Potpuno uređen slijed poruka

- Možemo li za sljedeći primjer reći da je *atomic multicast*?
- Vide li svi ispravni procesi slijed poruka u istom slijedu?



# Sadržaj predavanja

- Uvod, definicija pojmove
- Otpornost procesa na ispade
- Sporazum skupine procesa
- Pouzdana komunikacija skupine procesa
- **Raspodijeljeno izvršavanje operacije**
- Oporavak nakon ispada

# Transakcije

- Primjeri: operacije nad bazom podataka, slijed klijentskih zahtjeva za izvođenje operacija nad datotekom, bankarske transakcije
- Svojstva ACID
  - *Atomicity*: izvode se sve operacije unutar jedne transakcije ili niti jedna
  - *Consistency*: izvođenje transakcije dovodi sustav u konzistentno stanje
  - *Isolation*: konkurentne transakcije nemaju utjecaja jedna na drugu
  - *Durability*: nakon završetka transakcije sve su promjene trajne

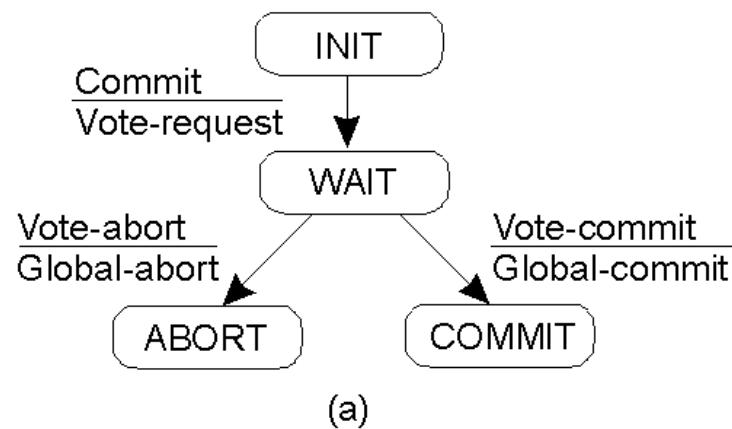
# Raspodijeljeno izvršavanje operacije

## *Distributed commit*

- operaciju izvodi svaki proces u skupini ili niti jedan
- ako je operacija isporuka poruke, riječ je o *atomic multicast*
- česta primjena za izvođenje raspodijeljenih transakcija
- ideja rješenja - jednofazno izvršavanje (*one-phase commit*)
  - postoji koordinator u skupini procesa
  - koordinator šalje zahtjev ostalim procesima za (lokalno) izvršavanje operacije
  - nedostaci: procesi ne mogu obavijestiti koordinatora u slučaju nemogućnosti izvršavanja operacije, ispad koordinatora

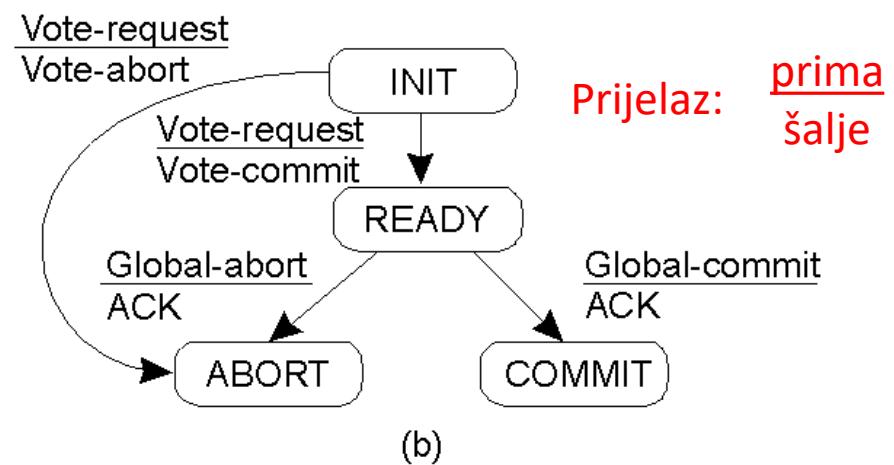
# Protokol dvofaznog izvršavanja (1/3)

Two-phase commit protocol (2PC)



Automat stanja koordinatora

U kojim stanjima može doći do blokiranja procesa i koordinatora?



Automat stanja procesa

Prema: Tanenbaum, Van Steen

# Protokol dvofaznog izvršavanja (2/3)

## Problemi

- Postoje stanja blokiranja na strani koordinatora i procesa
  - Proces je blokiran u stanju INIT kada čeka VOTE\_REQUEST – ako ne primi poruku nakon određenog vremena, proces može lokalno odustati od izvršavanja operacije
  - Koordinator je blokiran u stanju WAIT kada čeka odgovore svih procesa – ako nakon nekog perioda ne primi odgovor od svih procesa, koordinator može zaključiti da treba odustati od izvršavanja operacije i poslati svim procesima GLOBAL\_ABORT
  - Proces je blokiran u stanju READY čekajući konačnu odluku koordinatora – proces treba saznati koju je poruku koordinator posao i pitati druge procese što se događa!

# Protokol dvofaznog izvršavanja (3/3)

Akcije procesa  $p$  kada se nalazi blokiran u stanju READY i kada kontaktira drugi proces  $q$  iz skupine procesa G

| Stanje procesa $q$ | Akcije procesa $p$                                                                  |
|--------------------|-------------------------------------------------------------------------------------|
| COMMIT             | Obavi prijelaz u COMMIT<br>(jer je $q$ primio GLOBAL_COMMIT)                        |
| ABORT              | Obavi prijelaz u ABORT<br>(jer je $q$ primio GLOBAL_ABORT)                          |
| INIT               | Obavi prijelaz u ABORT<br>(jer $q$ nije primio VOTE_REQUEST)                        |
| READY              | Kontaktiraj drugi proces<br>(jer $q$ nije kao ni $p$ dobio odgovor od koordinatora) |

2PC je blokirajući protokol, jer u slučaju ispada koordinatora nakon slanja VOTE\_REQUEST, procesi ne mogu zaključiti o sljedećoj operaciji koju trebaju provesti (svi su u stanju READY)!

# Algoritam za koordinatora

```
while START _2PC to local log;  
multicast VOTE_REQUEST to all participants;  
while not all votes have been collected {  
    wait for any incoming vote;  
    if timeout {  
        while GLOBAL_ABORT to local log;  
        multicast GLOBAL_ABORT to all participants;  
        exit;  
    }  
    record vote;  
}  
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{  
    write GLOBAL_COMMIT to local log;  
    multicast GLOBAL_COMMIT to all participants;  
} else {  
    write GLOBAL_ABORT to local log;  
    multicast GLOBAL_ABORT to all participants;  
}
```



# Algoritam za proces

```
write INIT to local log;  
wait for VOTE_REQUEST from coordinator;  
if timeout {  
    write VOTE_ABORT to local log;  
    exit;  
}  
if participant votes COMMIT {  
    write VOTE_COMMIT to local log;  
    send VOTE_COMMIT to coordinator;  
    wait for DECISION from coordinator;  
    if timeout {  
        multicast DECISION_REQUEST to other participants;  
        wait until DECISION is received; /* remain blocked */  
        write DECISION to local log;  
    }  
    if DECISION == GLOBAL_COMMIT  
        write GLOBAL_COMMIT to local log;  
    else if DECISION == GLOBAL_ABORT  
        write GLOBAL_ABORT to local log;  
} else {  
    write VOTE_ABORT to local log;  
    send VOTE_ABORT to coordinator;  
}
```

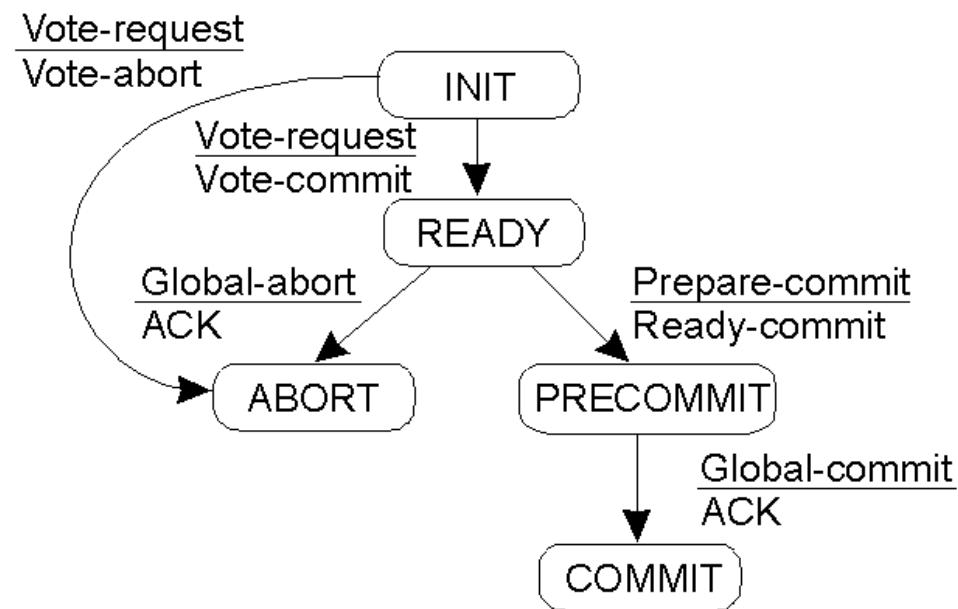
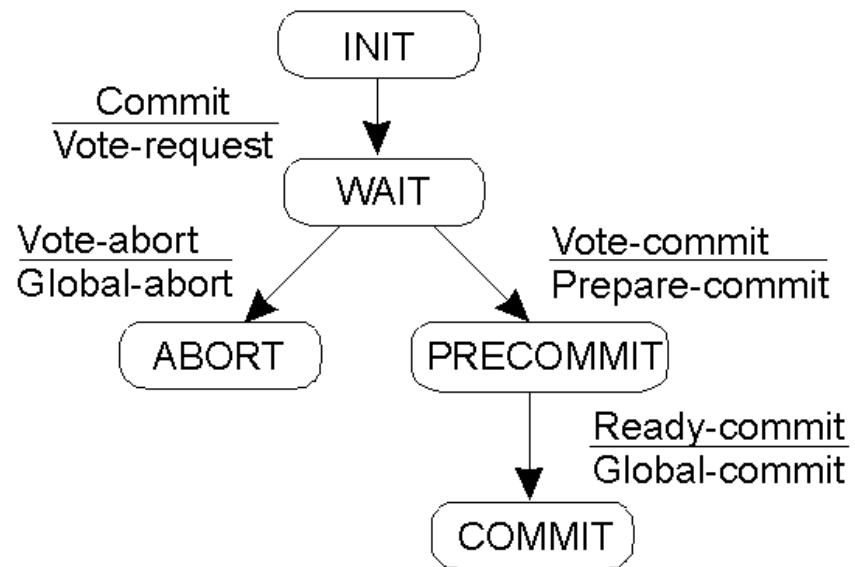


# Protokol trofaznog izvršavanja (1/3)

## *Three-phase commit protocol (3PC)*

- rješava problema blokiranja procesa u slučaju ispada koordinatora
- relativno se slabo koristi u praksi, jer se situacija koja dovodi do stanja blokiranja protokola 2PC događa iznimno rijetko

# Protokol trofaznog izvršavanja (2/3)



Nove poruke:

PREPARE\_COMMIT  
READY\_COMMIT

Novo stanje:

PRECOMMIT

# Protokol trofaznog izvršavanja (3/3)

- Koordinator može biti blokiran u stanju PRECOMMIT zbog ispada jednog procesa, ali može ostalim procesima poslati GLOBAL\_COMMIT
- Proces može biti blokiran u stanjima READY i PRECOMMIT
  - nakon isteka vremenske kontrole zaključuje da je došlo do ispada koordinatora i kontaktira ostale procese
  - ako je  $q$  u stanju COMMIT i  $p$  prelazi u COMMIT
  - ako je  $q$  u stanju ABORT i  $p$  prelazi u ABORT
  - ako su svi procesi (ili većina) u stanju PRECOMMIT, mogu svi preći u COMMIT
  - ako je  $q$  u INIT,  $p$  prelazi u ABORT
  - ako su svi procesi u stanju READY, mogu svi preći u ABORT

# Sadržaj predavanja

- Uvod, definicija pojmove
- Otpornost procesa na ispade
- Sporazum skupine procesa
- Pouzdana komunikacija skupine procesa
- Raspodijeljeno izvršavanje operacije
- **Oporavak nakon ispada**

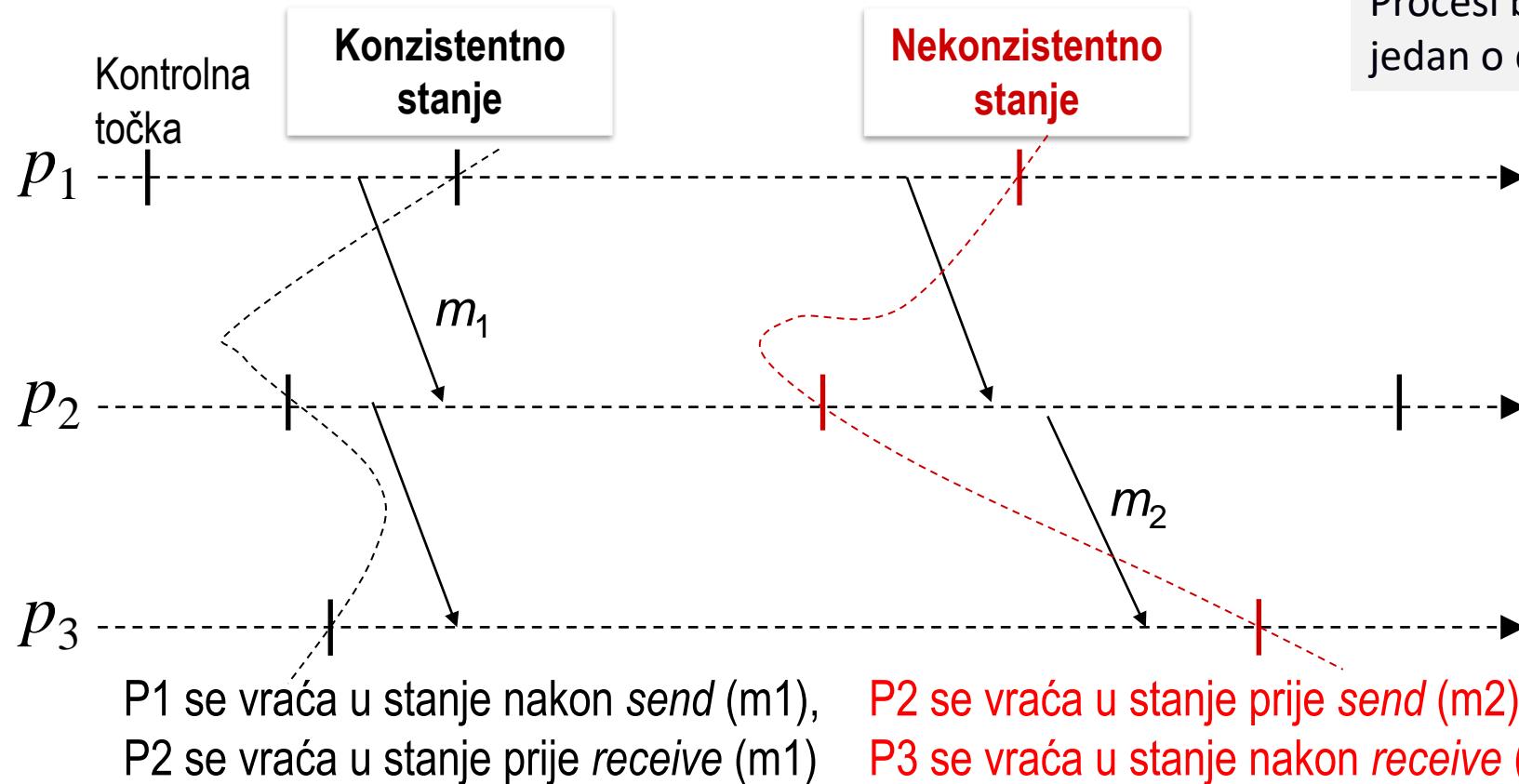
# Oporavak nakon ispada

- Nakon ispada procesa nužan je njegov oporavak i povratak u ispravno stanje
- Oporavak unazad
  - vratiti sustav u ispravno stanje u prošlosti
  - potrebno je s vremena na vrijeme pohraniti stanje sustava (kontrolne točke, *checkpoint*)
  - zapisuju se poslane i primljene poruke u dnevnički zapis (*log*)
- Oporavak korištenjem dnevničkog zapisa
  - proces u ispadu vraća se u prethodno ispravno stanje nakon čega izvodi akcije iz dnevničkog zapisa

# Bilježenje kontrolnih točaka

- Procesi mogu kontrolne točke bilježiti neovisno ili koordinirano
- **neovisno** zahtjeva bilježenje ovisnosti među procesima koji razmjenjuju poruku (šalju i primaju istu poruku) radi povratka u konzistentna stanja – pokazano na sljedećem primjeru
- **koordinirano** koristi koordinatora i jednostavnije je za implementaciju, koordinator šalje naredbu procesima da pohrane stanje sustava gotovo istovremeno (prije toga moraju primiti poruke u tranzitu i privremeno zaustaviti pripremljena slanja poruka)

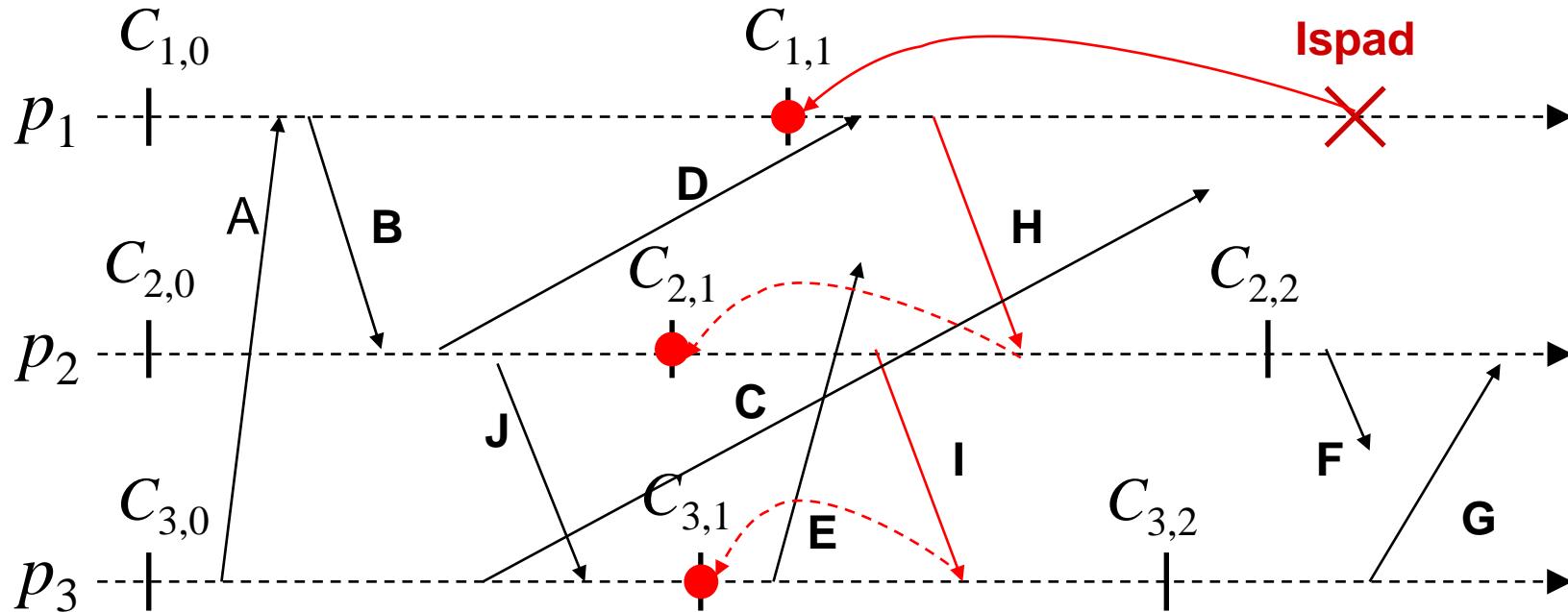
# Konzistentna i nekonzistentna stanja



Procesi bilježe kontrolne točke neovisno jedan o drugom!

Ako se proces P1 vraća u stanje prije send(m), onda se P2 mora vratiti u stanje prije receive(m)!

# Primjer oporavka povratkom unazad



Konzistentno stanje  
sustava:

$C_{1,1}, C_{2,1}, C_{3,1}$

A, B, J: normalne poruke (poslane i primljene)

C: zakašnjela poruka (može stići prije, tijekom ili nakon oporavka)

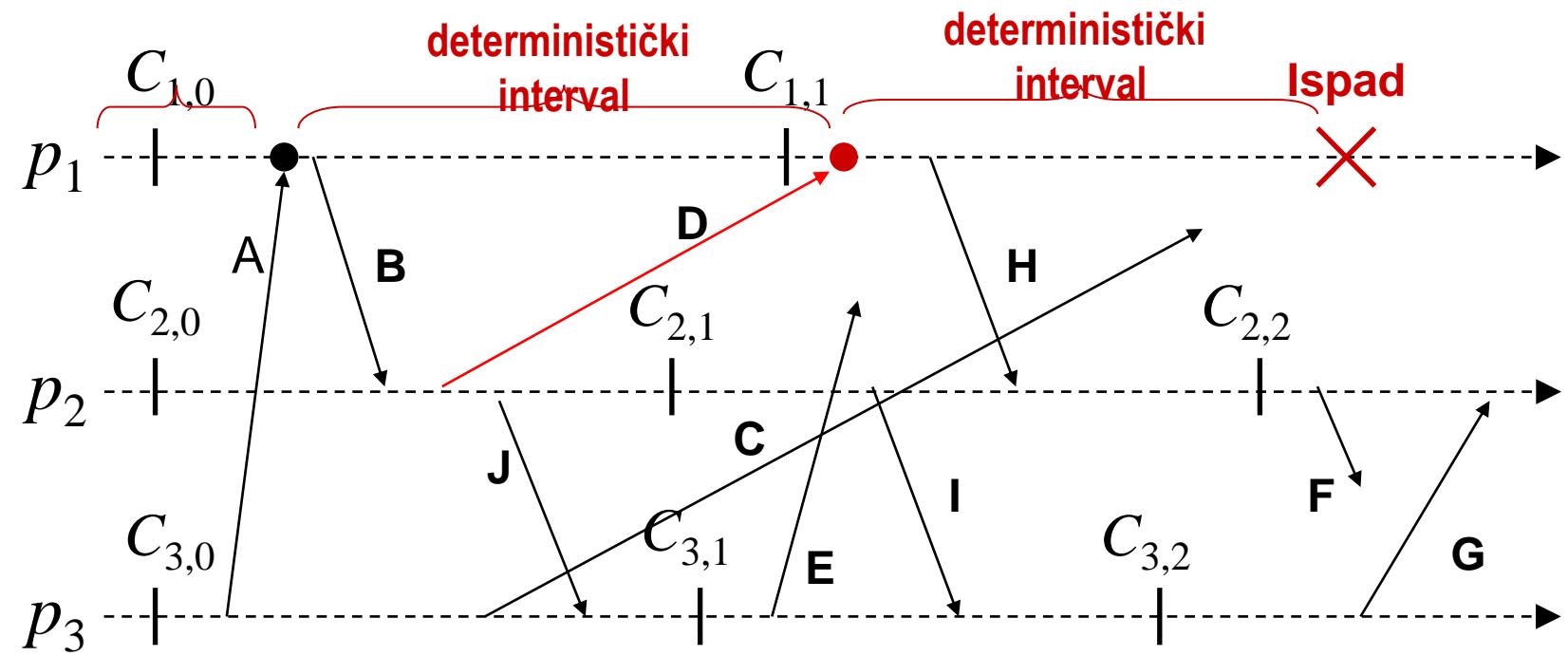
D: izgubljena poruka ( $p_2$  poslao,  $p_1$  nije zabilježio prijam)

G, H, I: poništene poruke, ponovit će se slanje i primanje

E i F: zakašnjele suvišne poruke  
(ako/kad stignu na odredište treba ih  
odbaciti, jer će se ponoviti njihovo  
slanje i primanje)

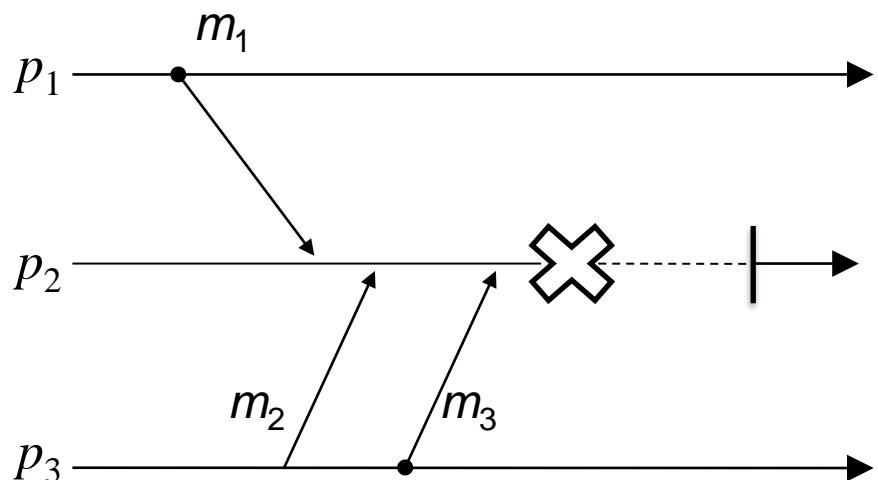
# Oporavak korištenjem dnevničkog zapisa

- Koristi kombinirano kontrolne točke i dnevnički zapis
- Deterministički interval: počinje i završava nedeterminističkim događajem (prijam poruke) koji se zapisuje u dnevnički zapis



# Zapisivanje poruka u dnevnički zapis

- Pravilo: Sve nedeterminističke događaje tj. primitak poruke s pripadnim informacijama (pošiljatelj, primatelj, redni broj poruke) treba bilježiti u dnevnik



Primjer neispravnog zapisa  
u dnevnički zapis

Proces  $p_2$  prima  $m_1$ ,  $m_2$ , i  $m_3$ , ali  $m_2$  ne zapisuje u dnevnički zapis.

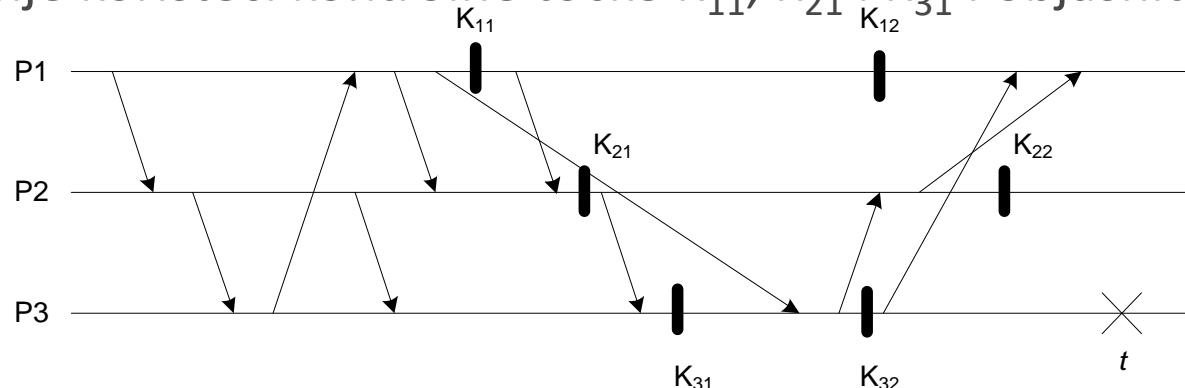
Što se događa nakon ispada i oporavka procesa  $p_2$ ?

$p_2$  čita i rekonstruira  $m_1$  iz zapisa  
 $p_2$  nema  $m_2$  u zapisu i ne može rekonstruirati tu poruku, (može rekonstruirati samo primanje poruke  $m_3$ ) =>

**$p_2$  se ne može vratiti u ispravno stanje  
neposredno prije ispada**

# Pitanja za učenje i ponavljanje

- Prepostavite da postoji skupina od  $m$  jednakih procesa. Navedite toleranciju ove skupine procesa na ispade ako prepostavite „obični“ ispad ili bizantski ispad.
- Identificirajte i objasnite blokirajuća stanja protokola 2PC.
- Slika prikazuje tri procesa i razmjenu poruka među njima. Svaki proces neovisno o drugim procesima bilježi svoja stanja u označenim kontrolnim točkama. U trenutku  $t$  dolazi do ispada procesa P3. Možemo li sustav od tri procesa na slici nakon ispada procesa P3 vratiti u konzistentno stanje koristeći kontrolne točke  $K_{11}$ ,  $K_{21}$ ,  $K_{31}$ ,  $K_{12}$  i  $K_{22}$  i objasnite zašto je to moguće ili nije moguće?



# Literatura

- A. S. Tanenbaum, M. Van Steen: *Distributed Systems: Principles and Paradigms*, Second Edition, Prentice Hall, 2007. (poglavlje 8, *Fault tolerance*)
- Ajay D. Kshemkalyani, Mukesh Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, Cambridge University Press, 2008.
- Rachid Guerraoui, André Schiper, "Software-Based Replication for Fault Tolerance," *Computer*, vol. 30, no. 4, 68-74, Apr., 1997.
- Lamport, L., Shostak, R., and Pease, M. "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, 382-401, Jul. 1982.
- Défago, X., Schiper, A., and Urbán, P. "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.* vol. 36, no. 4, 372-421, Dec. 2004.



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**  
Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**9. Replikacija i konzistentnost podataka**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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



*U slučaju 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.*

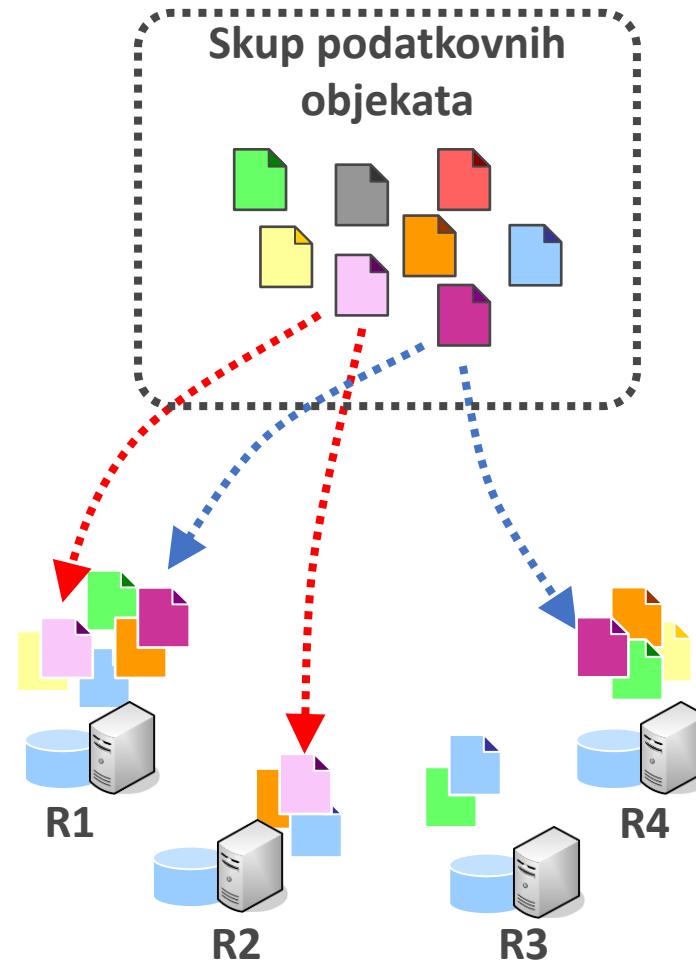
*Tekst licence preuzet je s <http://creativecommons.org/>*

# Sadržaj predavanja

- Replikacija i konzistentnost podataka
  - Svrha replikacije podataka
  - Dijeljeni spremnički prostor
- Modeli održavanja konzistentnosti podataka
- Uspostava replikacije podataka

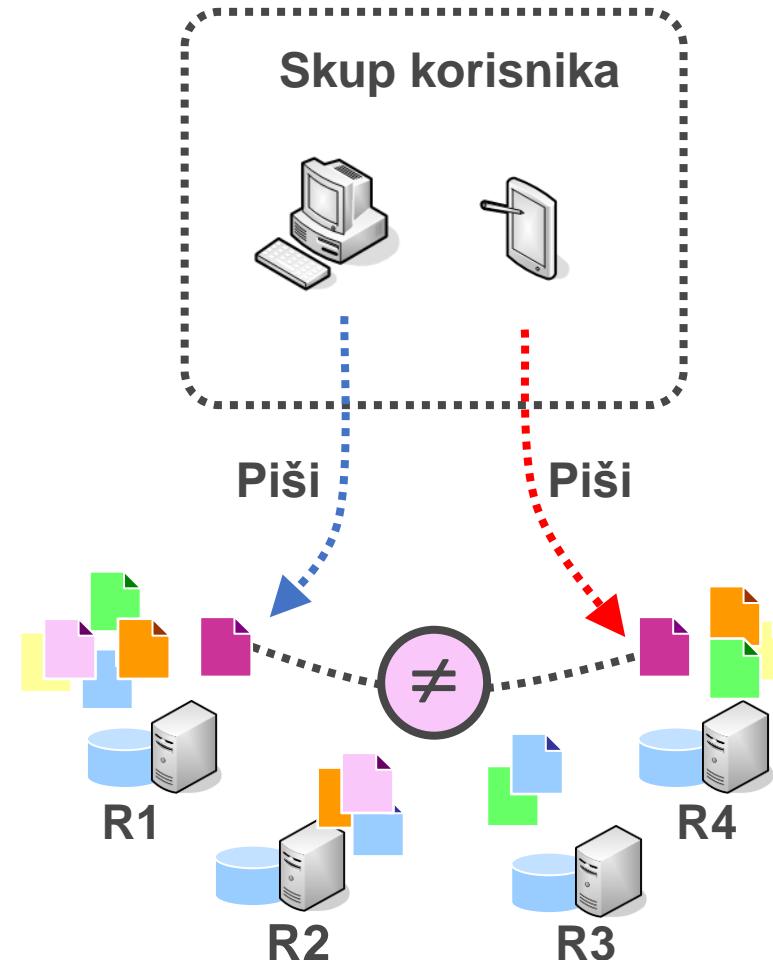
# Replikacija

- Raspodijeljeni sustavi koriste podatkovne objekte postavljene na skupu računala
- Podatkovni objekti su spremljeni u obliku nekoliko kopija (**replika**) na različitim računalima
- Replikacija je postupak stvaranja i upravljanja kopijama podatkovnih objekata



# Konzistentnost podataka

- Replike istog objekta mogu istodobno i nezavisno koristiti različiti korisnici i mijenjati ih na različite načine u vremenu
- Konzistentnost je **narušena** kada postoje replike nekog objekta koje nemaju istovjetno stanje
- Nakon nekog vremena postići će se konzistentno stanje – **eventualna konzistentnost**



# Svrha replikacije

- **Pouzdanost podataka**

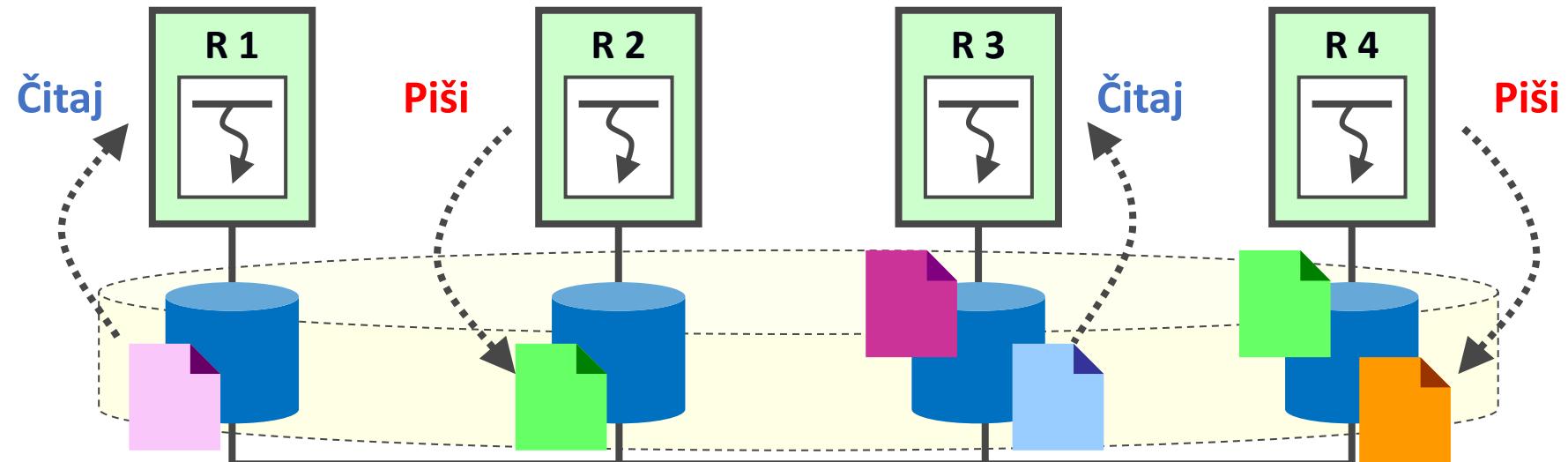
- U slučaju da neka od replika postane nedostupna, sustav proslijeđuje zahtjeve preostalim dostupnim replikama
- U slučaju da neka od replika ima pogrešne zapise, usporedbom zapisa više replika ostvaruje se otpornost na pogreške u zapisima

- **Učinkovito posluživanje podataka**

- U slučaju da neka od replika je preopterećena, pristigli zahtjevi proslijeđuju se ostalim replikama
- Replike je moguće predodrediti za posluživanje različitih razreda zahtjeva

# Model dijeljenog spremničkog prostora

- Skup spremničkih prostora na računalima u raspodijeljenoj okolini
- Osnovne operacije
  - Čitanje podataka (**R**)
  - Pisanje podataka (**W**)



# Hadoop Distributed File System (HDFS)

- Primjer dijeljenog spremničkog prostora
- Čini ga grozd računala (skup računala povezanih brzom lokalnom mrežom) koji se sastoji od jednog glavnog čvora **NameNode** i nekoliko pomoćnih čvorova **DataNode**
- Svaki podatkovni objekt (datoteka) se u HDFS-u
  - Dijeli na jedan ili više blokova podataka (čija je maksimalna veličina unaprijed predefinirana)
  - Pri čemu je svaki blok podataka (uobičajeno) repliciran 3 puta
- Čvorovi **DataNode**
  - Pohranjuju blokove podataka kod sebe
  - Obrađuju klijentske zahtjeve za pisanje i čitanje blokova podataka
- Čvor **NameNode**
  - Upravlja spremničkim prostorom – određuje gdje će biti pohranjene replike pojedinog bloka podataka
  - Regulira pristup klijenata pohranjenim podatkovnim objektima jer jedini zna gdje se nalaze replike određenog bloka podataka

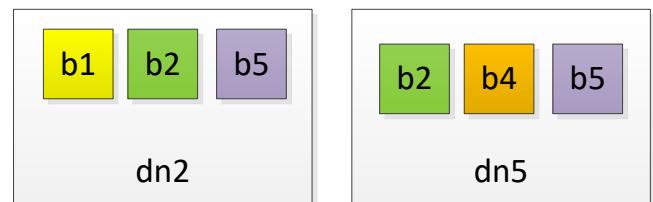
# Replikacija u HDFS-u

- Svaki **DataNode** periodički šalje poruke **NameNode**-u
  - Porukama **HeartBeat** daje do znanja da je i dalje raspoloživ
  - Porukama **BlockReport** dojavljuje koji blokovi podataka su pohranjeni kod njega
- Na osnovu ovih poruka **NameNode**
  - Inicira dodatnu replikaciju ukoliko neki **DataNode**
    - Postane nedostupan (npr. zbog tehničkog kvara) ili
    - Izgubi blokove podataka pohranjene kod njega (npr. u slučaju problema na nekom od njegovih tvrdih diskova)
  - Inicira brisanje viška replika prilikom vraćanja u sustav privremeno nedostupnog **DataNode-a**
  - Prosljeđuje klijentske zahtjeve za pisanje i čitanje blokova podataka samo na ispravne čvorove **DataNode**

# Arhitektura HDFS-a

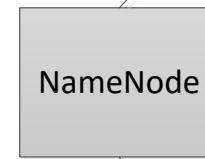
skip

DataNodes:



Rack1

Rack2



NameSpace State (FileName, Block IDs):  
/path1/file1 {b1,b3,b5}  
/path2/file2 {b2,b4}

BlockMap (Block ID, DataNodes):  
b1 {dn1, dn2, dn6}  
b2 {dn2, dn4, dn5}  
b3 {dn1, dn3, dn6}  
b4 {dn3, dn4, dn5}  
b5 {dn2, dn5, dn6}

DataNode State (DataNode, Available):  
dn1 true  
dn2 true  
dn3 true  
dn4 true  
dn5 true

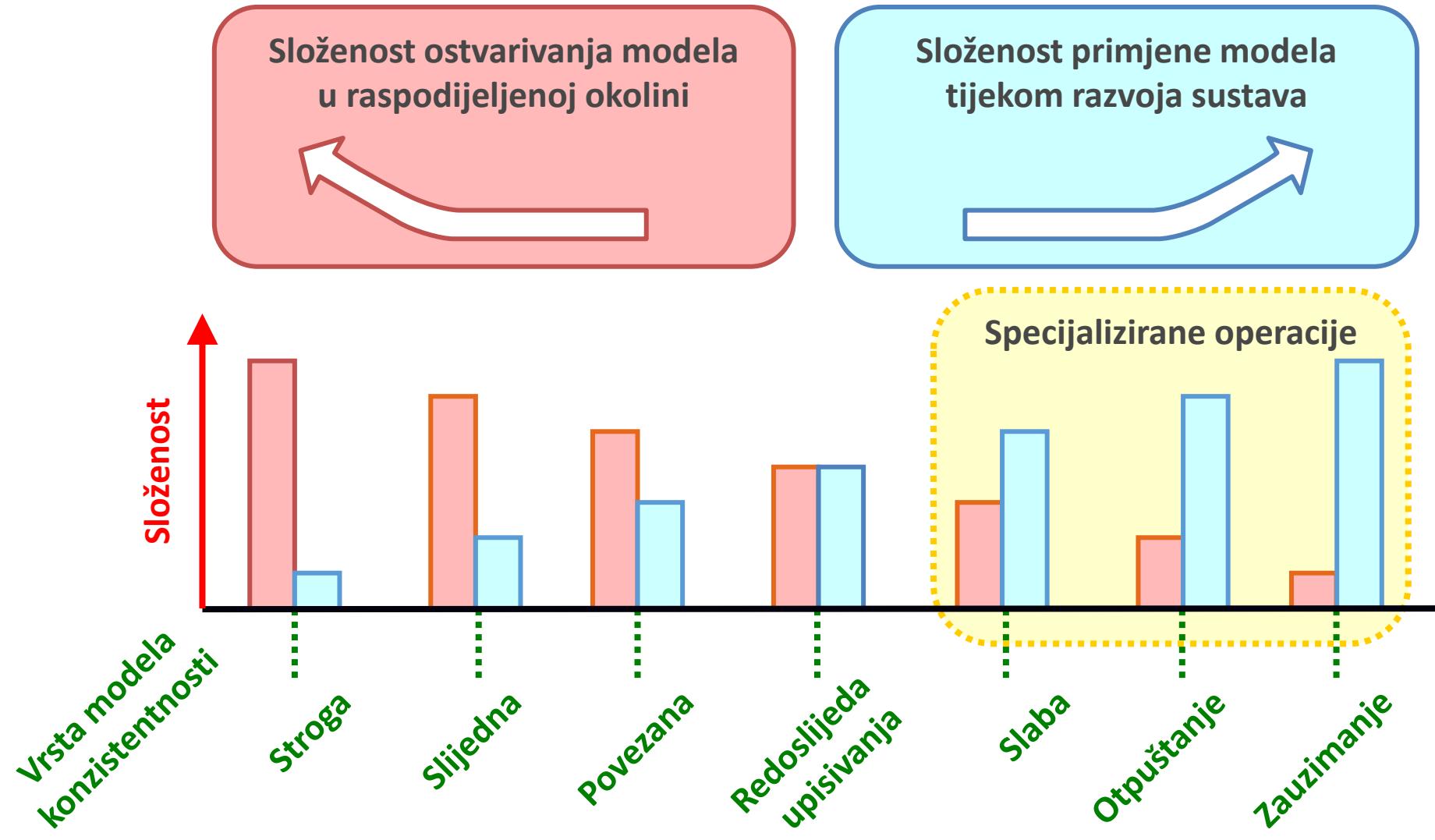
# Sadržaj predavanja

- Replikacija i konzistentnost podataka
  - Svrha replikacije podataka
  - Dijeljeni spremnički prostor
- Modeli održavanja konzistentnosti podataka
- Uspostava replikacije podataka

# Modeli održavanja konzistentnosti podataka

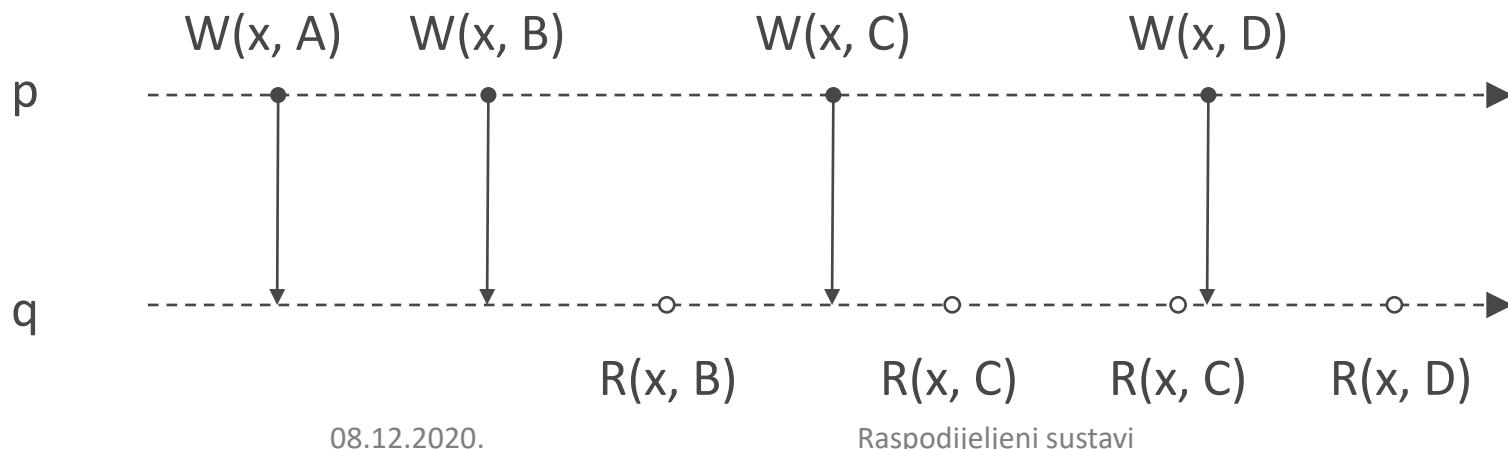
- Raspodijeljeni sustav **podržava** određeni model konzistentnosti kada se:
  - podaci u sustavu **mijenjaju** i te
  - **promjene vide** samo na **načine dozvoljene** tim modelom
- Model konzistentnosti stoga uvodi ograničenja na načine mijenjanja podataka u vremenu te na načine na koje se te promjene vide u sustavu

# Razredba modela konzistentnosti



# Stroga konzistentnost (*Strict consistency*)

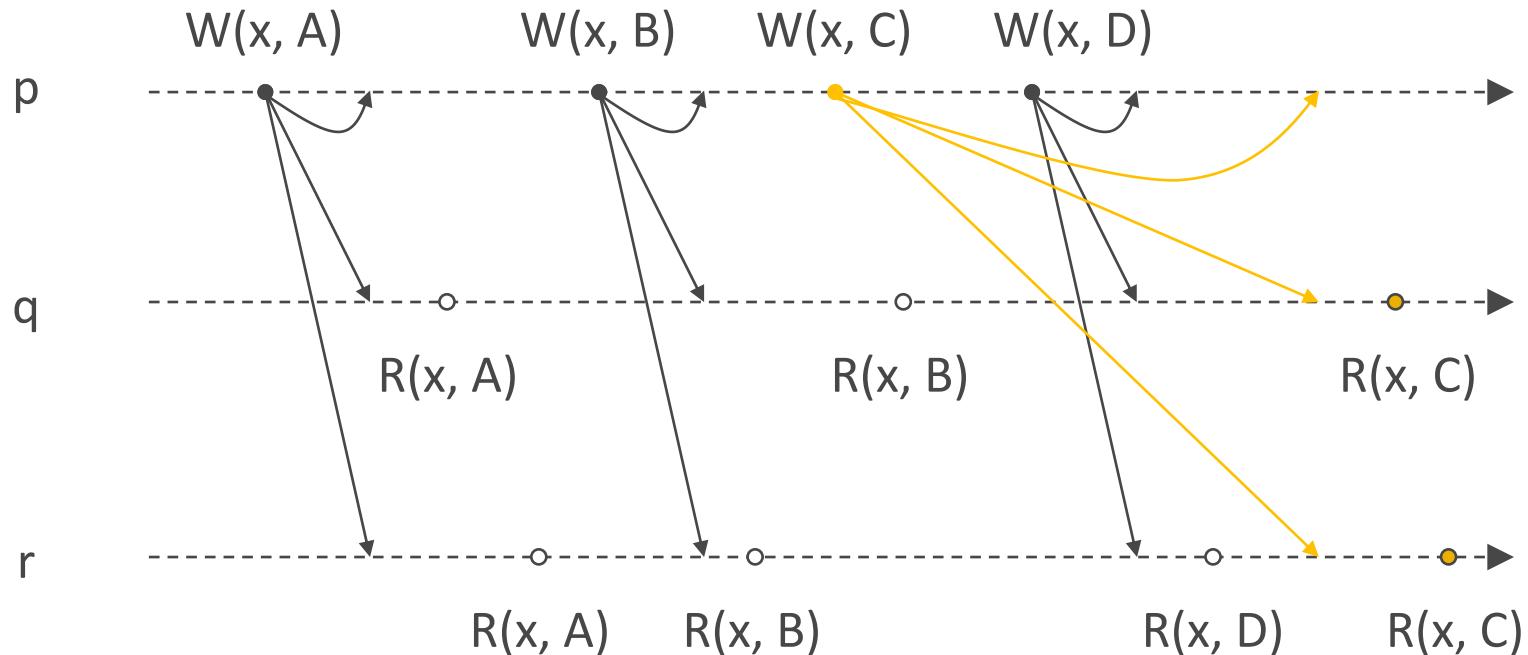
- Čitanje podatka na lokaciji X daje vrijednost koja je posljednja bila zapisana na lokaciju X
  - $W(x, Y)$  – zapis podatka Y na lokaciju x
  - $R(x, Y)$  – čitanje podatka Y s lokacije x
- Značajke modela
  - Primjena globalnog vremena – sinkroni model
  - Ostvaren po uzoru na jednoprocесorske sustave
- Smatran je neizvedivim u praksi (međutim Googleova raspodijeljena baza podataka [Cloud Spanner](#) ga podržava jer koristi raspodijeljeni sat TrueTime)



# Slijedna konzistentnost (*Sequential consistency*)

- Slijed izvođenja operacija može biti **proizvoljan**, ali svi procesi moraju na **jednak način** vidjeti konačni slijed izvođenja akcija u vremenu.
- Značajke modela
  - Nije potrebno održavati globalni tijek vremena
  - Procesi se moraju dogovoriti o globalnom redoslijedu izvođenja akcija u vremenu
- U praksi se najčešće koristi u raspodijeljenim datotečnim sustavima (DFS-ovima) kao što su DEFS, Calypso, Frangipani, xFS, Microsoft Niobe, Ivy DSM, itd.

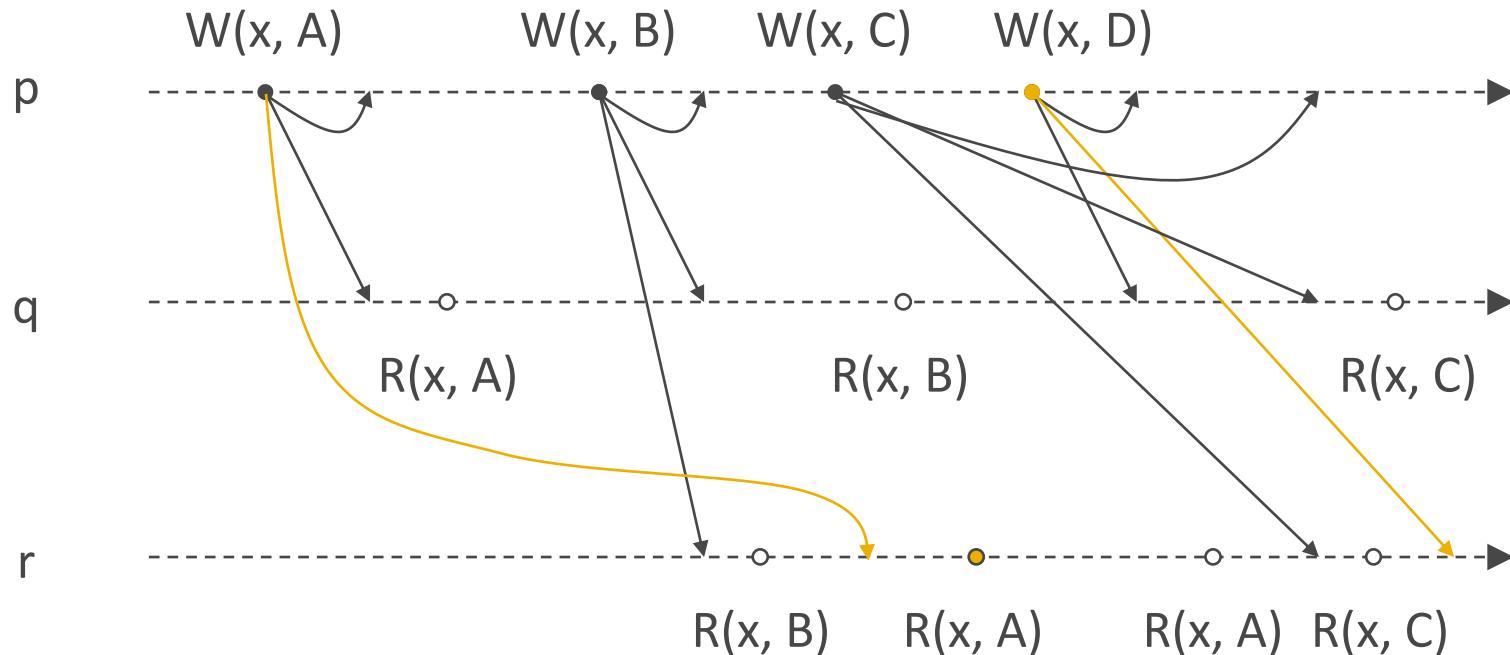
# Primjer slijedne konzistentnosti



Čitanje q:  $A \Rightarrow B \Rightarrow D \Rightarrow C$  ( $D$  nije pročitao)

Čitanje r:  $A \Rightarrow B \Rightarrow D \Rightarrow C$

# Primjer slijedne nekonzistentnosti



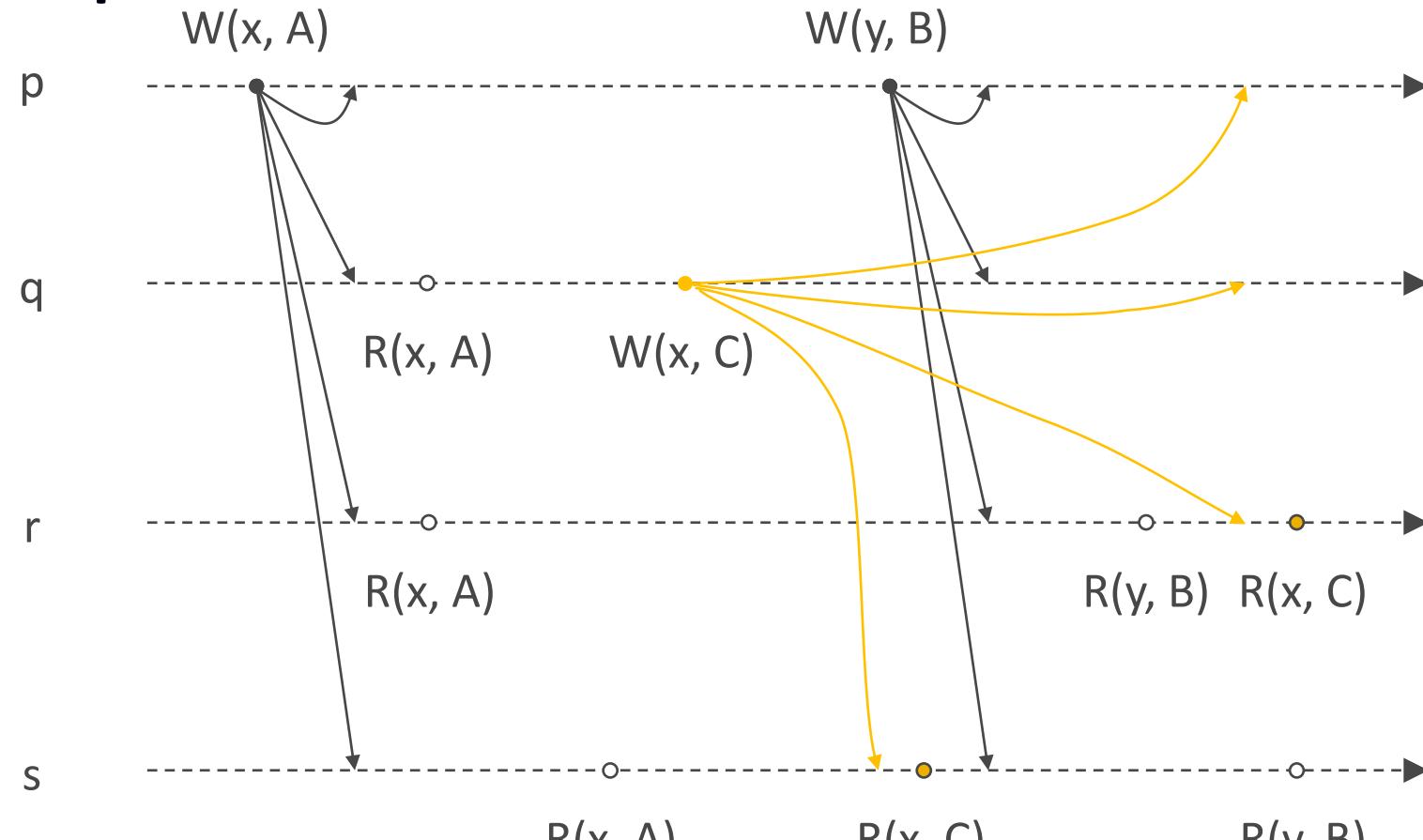
Čitanje q:  
A  $\Rightarrow$  B  $\Rightarrow$  D  $\Rightarrow$  C (D nije pročitao)

Čitanje r:  
B  $\Rightarrow$  A  $\Rightarrow$  A  $\Rightarrow$  C

# Povezana konzistentnost (*Casual consistency*)

- Redoslijed izvođenja **povezanih** operacija pisanja vidljiv je svim procesima na jednak način, dok redoslijed izvođenja operacija pisanja koje **nisu povezane** svakom procesu može biti prikazan na drugačiji način
- Povezanost operacija
  - Operacija pisanja **W** sadržaja u lokaciju **X** prethodi operaciji čitanja sadržaja **R** iz lokacije **X** čime je operacija **R** povezana s operacijom **W**
  - Dvije operacije pisanja nisu povezane ako ostvaruju istodobno zapisivanje sadržaja u različite lokacije dijeljenog spremnika
- U praksi se koristi u nekim NoSQL bazama podataka kao što su MongoDB i AntidoteDB

# Primjer povezane konzistentnosti



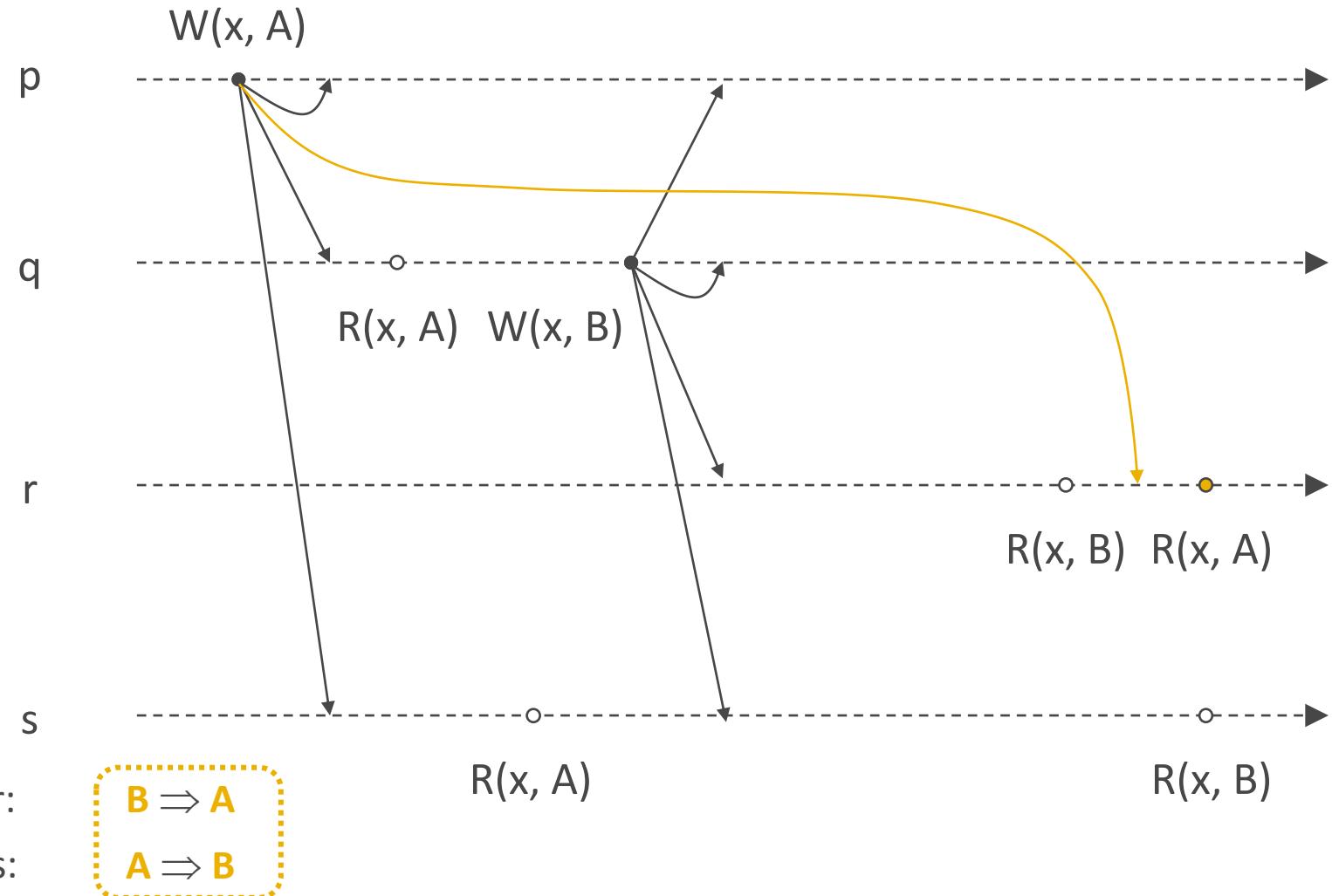
Čitanje r:

$A \Rightarrow B \Rightarrow C$

Čitanje s:

$A \Rightarrow C \Rightarrow B$

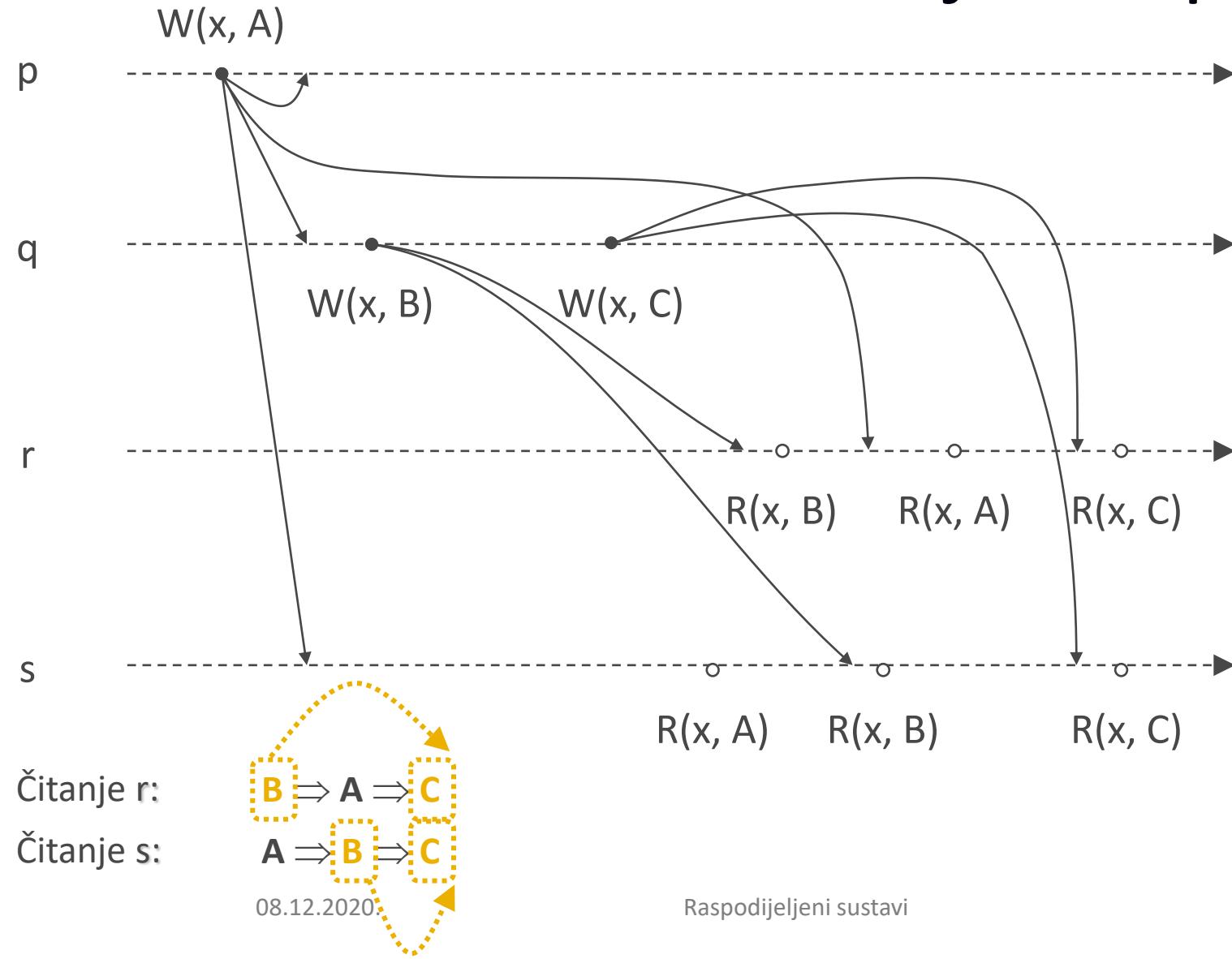
# Primjer (ne)povezane nekonzistentnosti



# Konzistentnost redoslijeda upisivanja (*FIFO con.*)

- Redoslijed izvođenja operacija pisanja provedenih od strane **jednog** procesa vidljiv je na **jednak** način svim ostalim procesima, ali redoslijed izvođenja operacija pisanja **različitih** procesa može biti vidljiv na **proizvoljan** način ostalim procesima.
- Značajke modela
  - Jednostavno ostvarenje zasnovano na pridruživanju jedinstvenih oznaka svakom zahtjevu za pisanje
  - Jedinstvena oznaka uključuje identifikator procesa i redni broj izvođenja operacije
- U praksi se koristi u nekim raspodijeljenim dijeljenim memorijama

# Primjer konzistentnosti redoslijeda upisivanja



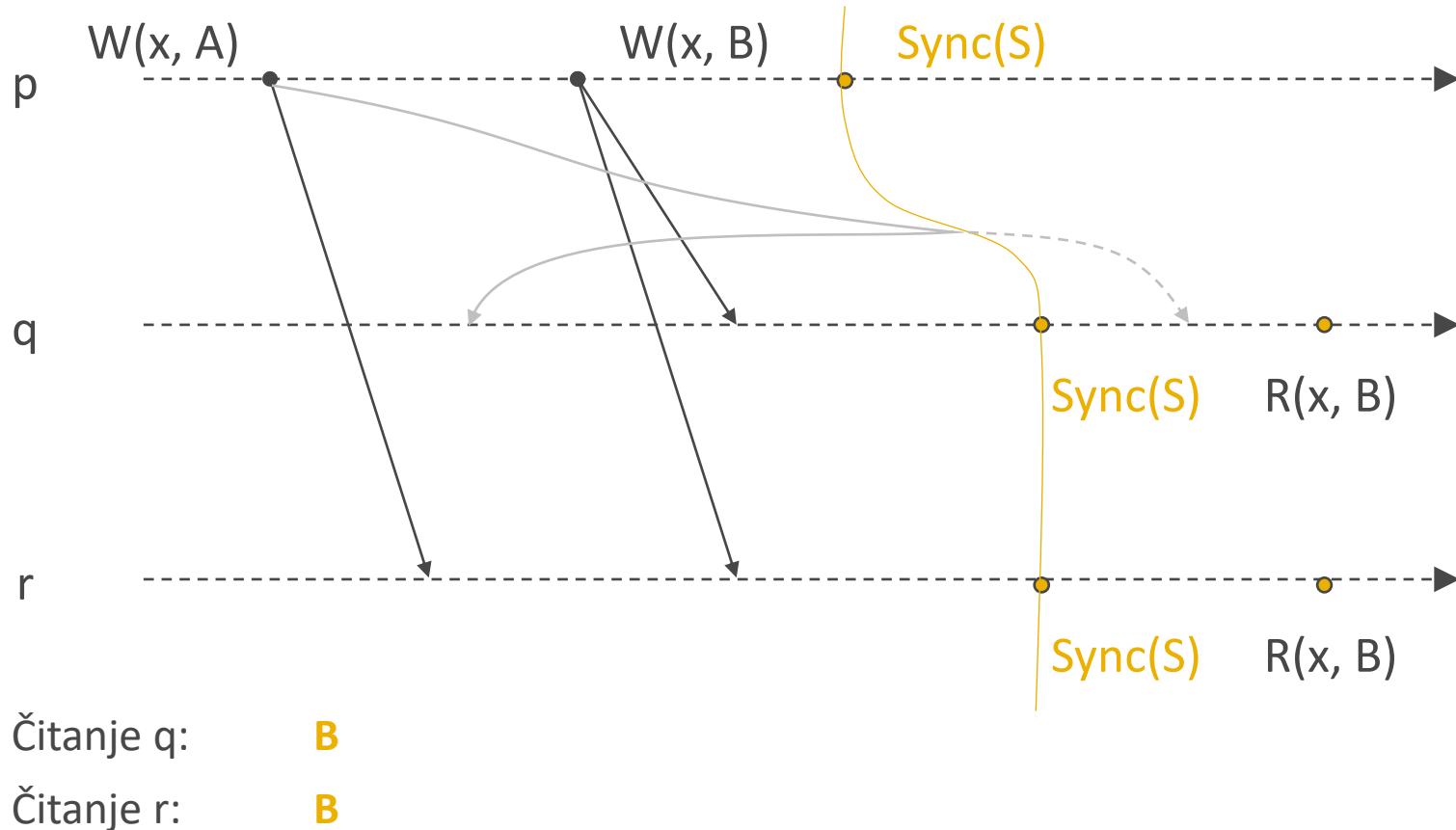
# Slaba konzistentnost (*Weak consistency*)

- Ostvaruje se primjenom sinkronizacijskih varijabli koje ostvaruju upravljanje trenucima sinkronizacije replika u raspodijeljenoj okolini
- Operacija sinkronizacije **Sync (S)**
- Primjena sinkronizacijskih varijabli
  - Usklađivanje svih replika zadano skupa podataka
  - Svi procesi vide istu vrijednost podataka u trenutku nakon što je provedeno usklađivanje podataka – njihova sinkronizacija
  - Slijed akcija nad sinkronizacijskim varijablama vidljiv je na jednak način svim procesima

# Slaba konzistentnost (*Weak consistency*)

- Uvjeti primjene i ostvarenja operacije Sync
  - Operacija nad sinkronizacijskom varijablu dovršava se tek nakon što su završene sve prethodno započete operacije pisanja
  - Nove operacije pisanja i čitanja mogu se izvoditi tek nakon završetka izvođenja operacije nad sinkronizacijskom varijablom
- U praksi se koristi u nekim raspodijeljenim strukturama podataka (npr. raspodijeljeno stablo B-Tree)

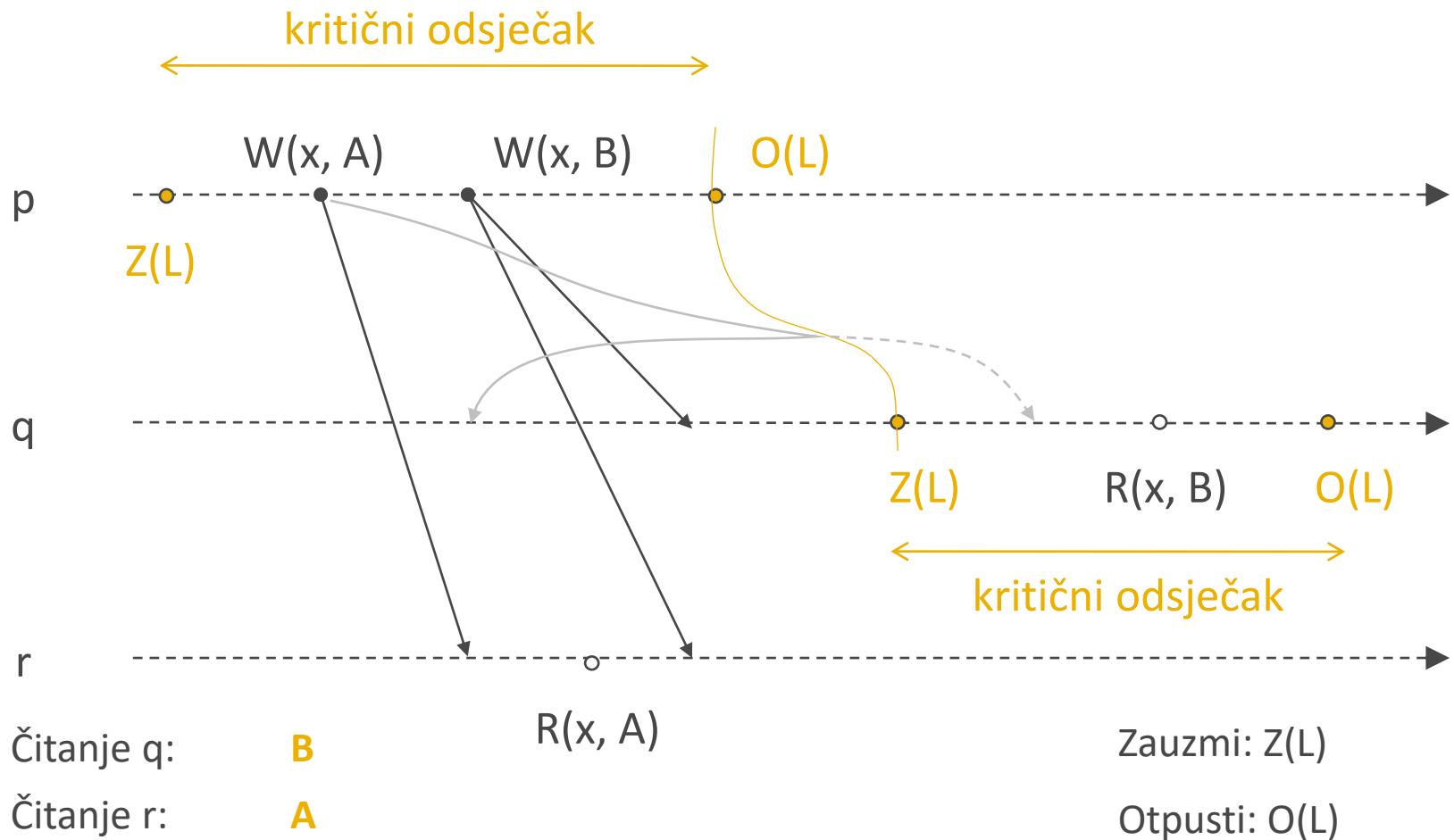
# Primjer slabe konzistentnosti



# Konzistentnost otpuštanja (*Release consistency*)

- Vrsta konzistentnosti zasnovana na primjeni kritičnih odsječaka
  - Konzistentnost se održava nakon izlaska iz kritičnog odsječaka
  - Izgradnja kritičnih odsječaka ostvarena je primjenom operacija **Zauzmi (Z)** i **Otpusti (O)**
- Operacija **Zauzmi**
  - Ulazak u kritični odsječak
  - Isključivi pristup spremniku podataka
- Operacija **Otpusti**
  - Izlazak iz kritičnog odsječka, prije izlaska sve lokalne promjene prosljeđuju se svim replikama podataka
  - Dozvoljen pristup spremniku podataka
- U praksi se koristi u nekim raspodijeljenim dijeljenim memorijama (Munin, Adsmith, itd.), a centralizirani model se koristi u programskim jezicima C++ i Javi

# Primjer konzistentnosti otpuštanja



# Konzistentnost zauzimanja (*Entry consistency*)

- Konzistentnost zasnovana primjeni sinkronizacijskih varijabli
  - Konzistentnost se održava u trenutku ulaska u kritični odsječak
  - Izgradnja kritičnih odsječaka ostvarenih primjenom operacija **Zauzmi (Z)** i **Otpusti (O)**
- Operacija **Zauzmi**
  - Ulazak u kritični odsječak, početak izvođenja odsječka tek nakon što su usklađene sve vrijednosti replika s posljednjom promjenom
- Operacija **Otpusti**
  - Izlazak iz kritičnog odsječaka
- U praksi se koristi u nekim raspodijeljenim dijeljenim memorijama (Midway, DiSOM, itd.)

# Primjer konzistentnosti zauzimanja



Čitanje q:  $A \Rightarrow \emptyset$  (nema jamstva za čitanje s lokacije y)

Čitanje r:  $B$

# Sadržaj predavanja

- Replikacija i konzistentnost podataka
  - Svrha replikacije podataka
  - Dijeljeni spremnički prostor
- Modeli održavanja konzistentnosti podataka
- Uspostava replikacije podataka

# Uspostava replikacije podataka

- Uvod u replikaciju podataka
- Organizacija sustava replika
- Razredba vrsta replika
  - Trajne replike, poslužiteljske replike, korisničke replike
- Održavanje konzistentnosti replika
  - Dohvaćanje promjena stanja replika
  - Prosljeđivanje promjena stanja replika
- Ostvarivanje operacija
  - Operacije pisanja sadržaja
  - Operacije čitanja sadržaja

# Uvod u replikaciju podataka

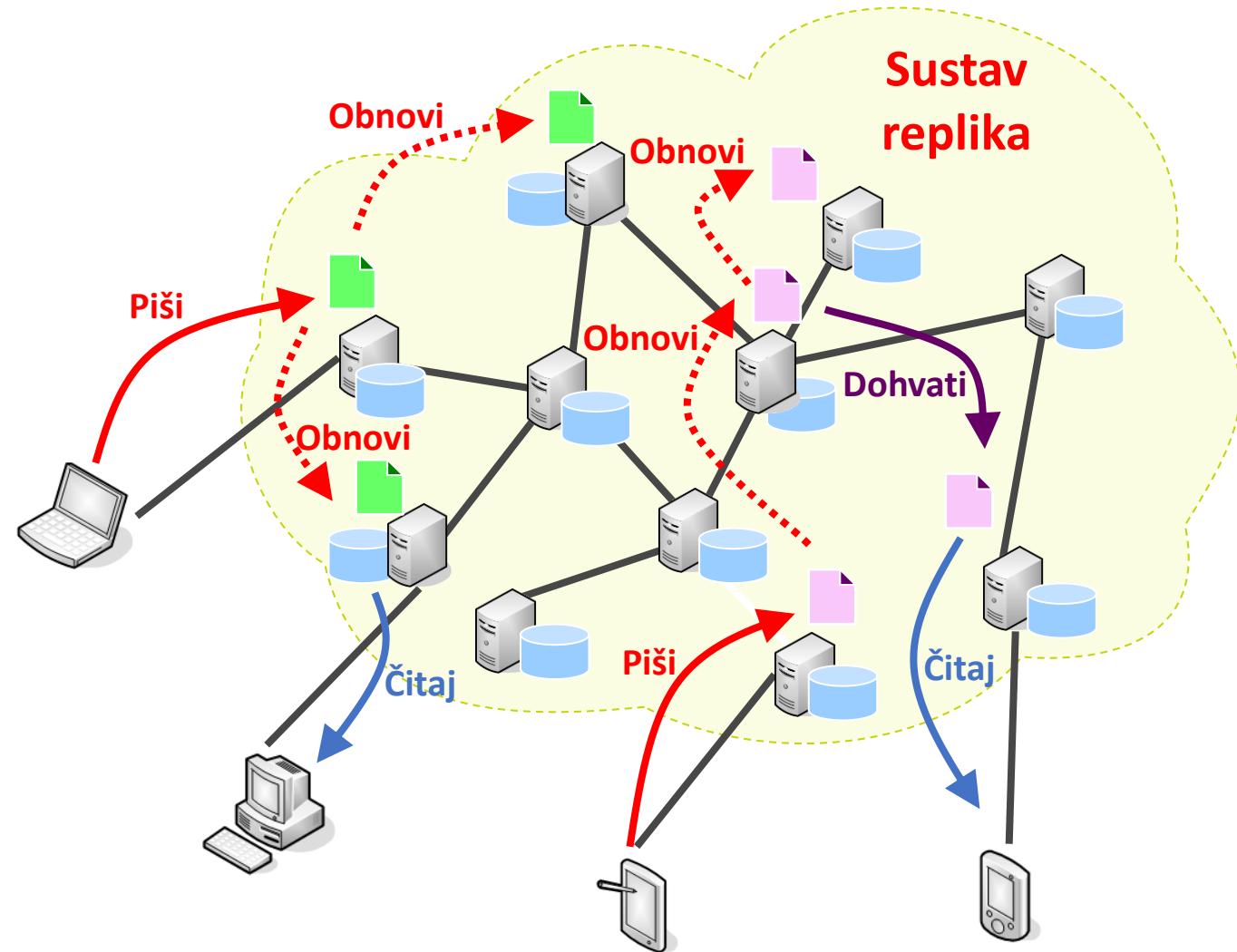
- **Replika** je kopija originalnog podatkovnog objekta (resursa) na nekom drugom računalu u sustavu
- Raspodijeljeni sustav je **konzistentan** u nekom vremenskom trenutku ukoliko su sve replike u njemu nalaze u istom stanju
- **Replikacija** je postupak stvaranja i upravljanja replikama
- Dva glavna problema u sustavu replika:
  - Gdje smjestiti replike?
  - Kako održati konzistentnost replika? (Tko će i kada postavljati i ažurirati replike u sustavu?)

# Prednosti i nedostaci replikacije

- Prednosti replikacije
  - Povećanje pouzdanosti sustava (otpornost na kvarove, otpornost na pogreške)
  - Povećanje performansi sustava (skalabilnost, odziv)
- Nedostaci replikacije
  - Nekonzistentnost replika
  - Održavanje konzistentnosti replika može generirati veliku količinu mrežnog prometa

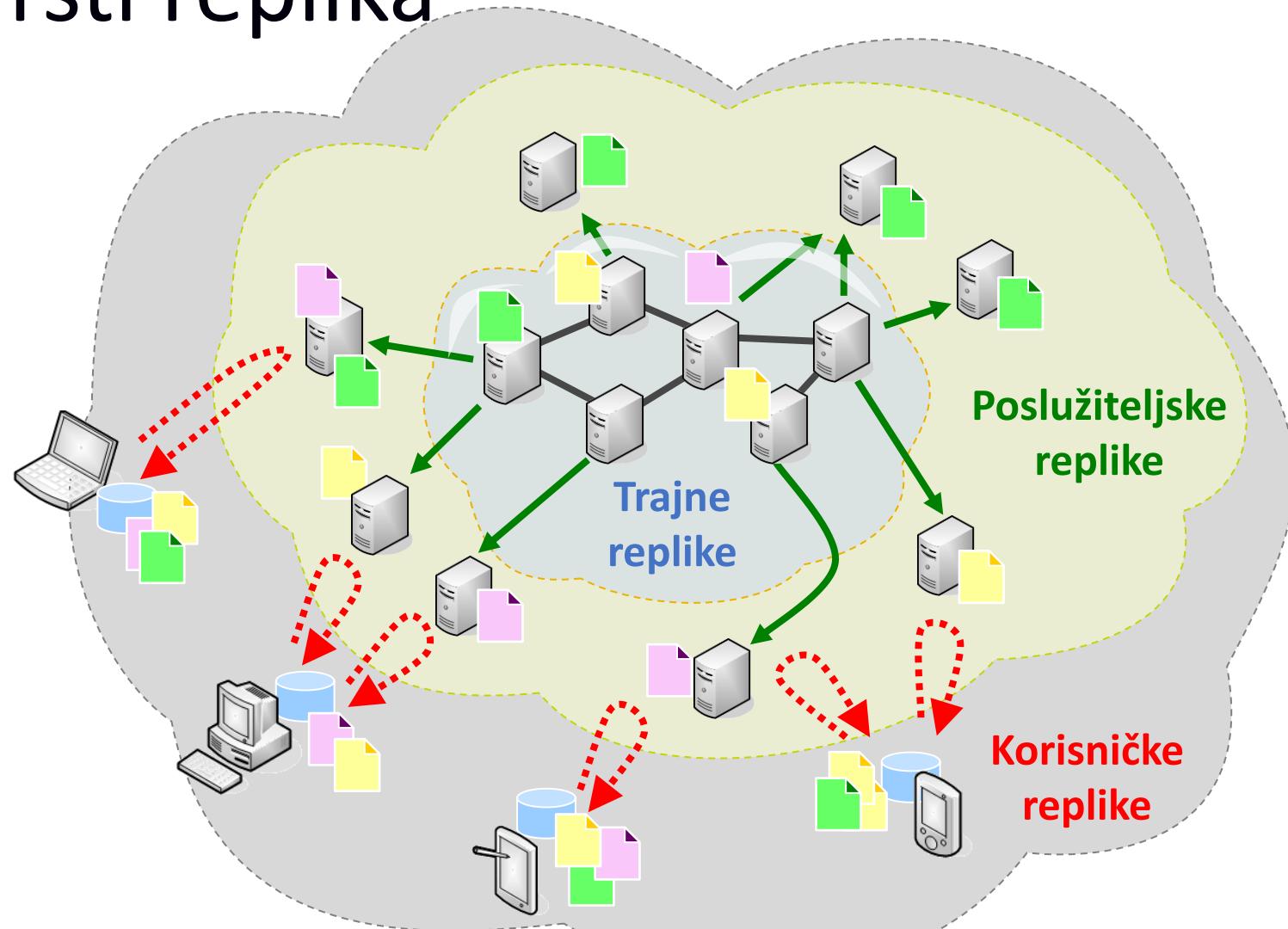
# Organizacija sustava replika

- Elementi sustava
  - Računala
  - Spremniči
  - Replike
- Operacije
  - Čitaj
  - Piši
  - Dohvati
  - Obnovi



# Razredba vrsti replika

- Vrste replika
  - Trajne
  - Korisničke
  - Poslužiteljske



# Trajne replike (1)

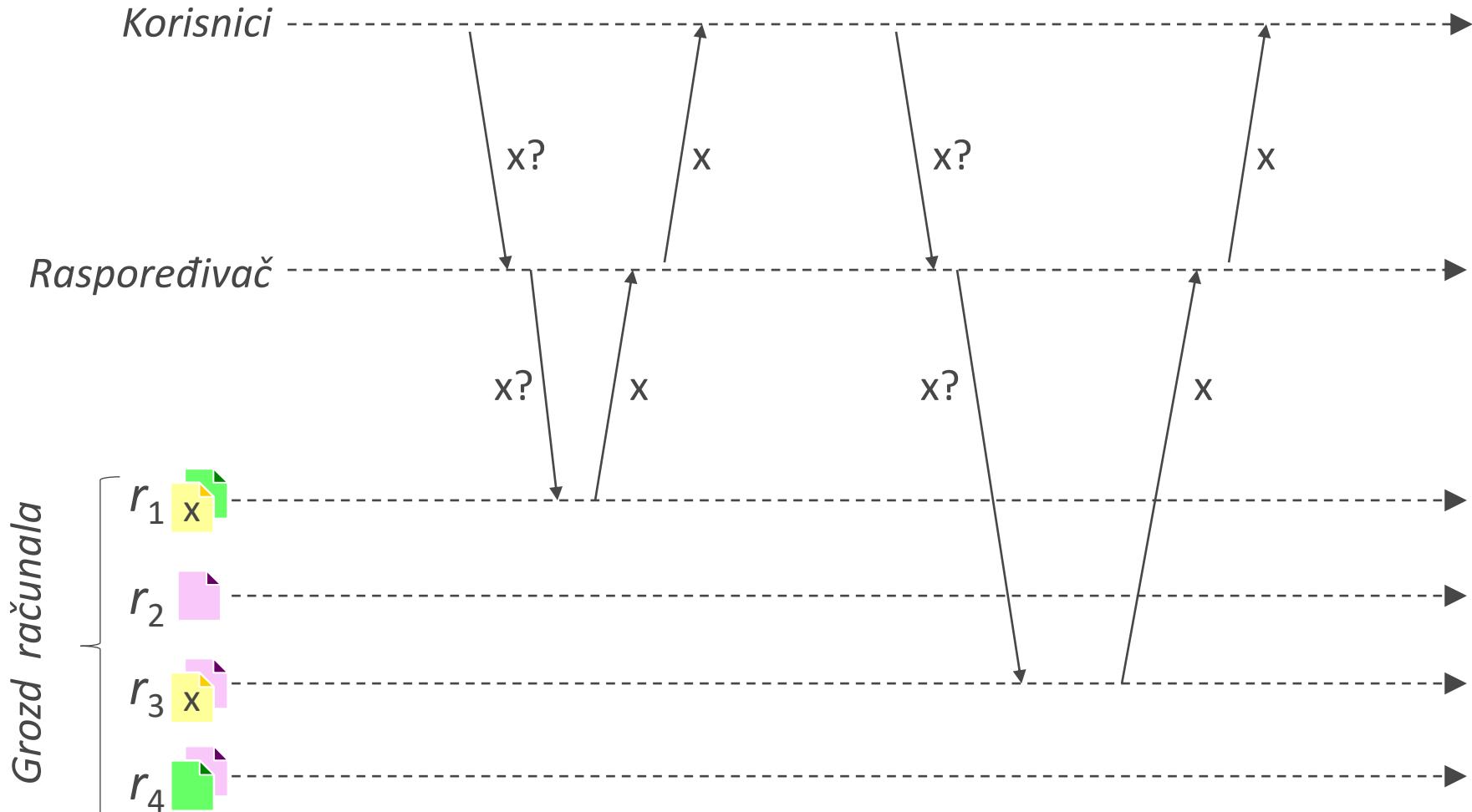
- Početni skup replika postavljen na skupu računala povezanih lokalnom mrežom
  - Grozdovi poslužitelja
  - Replikacijski poslužitelji
- Osnovne značajke
  - Statička organizacija i postavke sustava
  - Većina zahtjeva je čitanje podataka
  - Raspoređivanje zahtjeva na dostupne replike

# Trajne replike (2)

- Organizacija grozda poslužitelja
  - Rasporedjivač je pristupna točka sustava koja sadrži informacije koje opisuju poslužitelje unutar grozda
  - Rasporedjivač prihvata i prosljeđuje zahtjeve poslužiteljima
  - Odabir poslužitelja ostvaruje se tako da se ostvari optimalno raspoređivanje opterećenja po replikama koje se nalaze na poslužiteljima
  - Poslužitelji prihvataju zahtjev te prosljeđuju rezultate obrade
- Primjer: **zastupnik (reverse proxy) poslužitelja weba**

# Primjer uporabe trajnih replika

$x?$  – zahtjev za dohvat replike tipa X, x – sadržaj replike tipa X

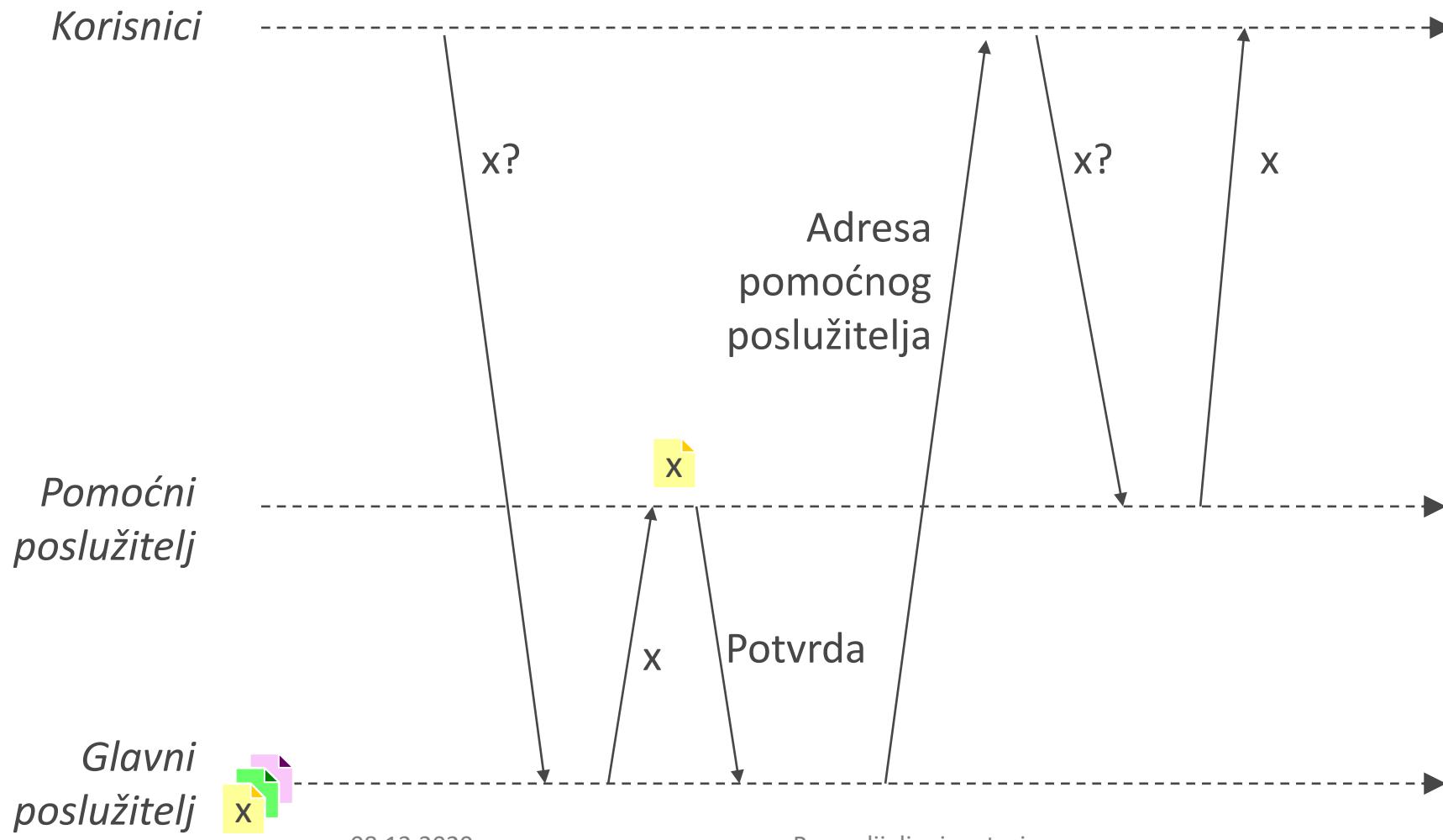


# Poslužiteljske replike

- Poslužitelj sadrži trajne replike koje su dostupne korisnicima
  - U slučaju povećanja potražnje podataka, poslužitelj započinje postupak repliciranja podataka
  - Poslužitelj proslijeđuje replike traženih podataka privremenim poslužiteljima
- Osnovne značajke
  - Poslužitelj prati vlastito opterećenje
  - Odabir i raspoređivanje replika ostvaruje se dinamički tijekom rada sustava
- Primjer: regionalna raspodjela opterećenja poslužitelja weba, poslužitelji za VoD (*video on demand*) kod IPTV-a

# Primjer uporabe poslužiteljskih replika

$x?$  – zahtjev za dohvrat replike tipa X, x – sadržaj replike tipa X



# Korisničke replike (1)

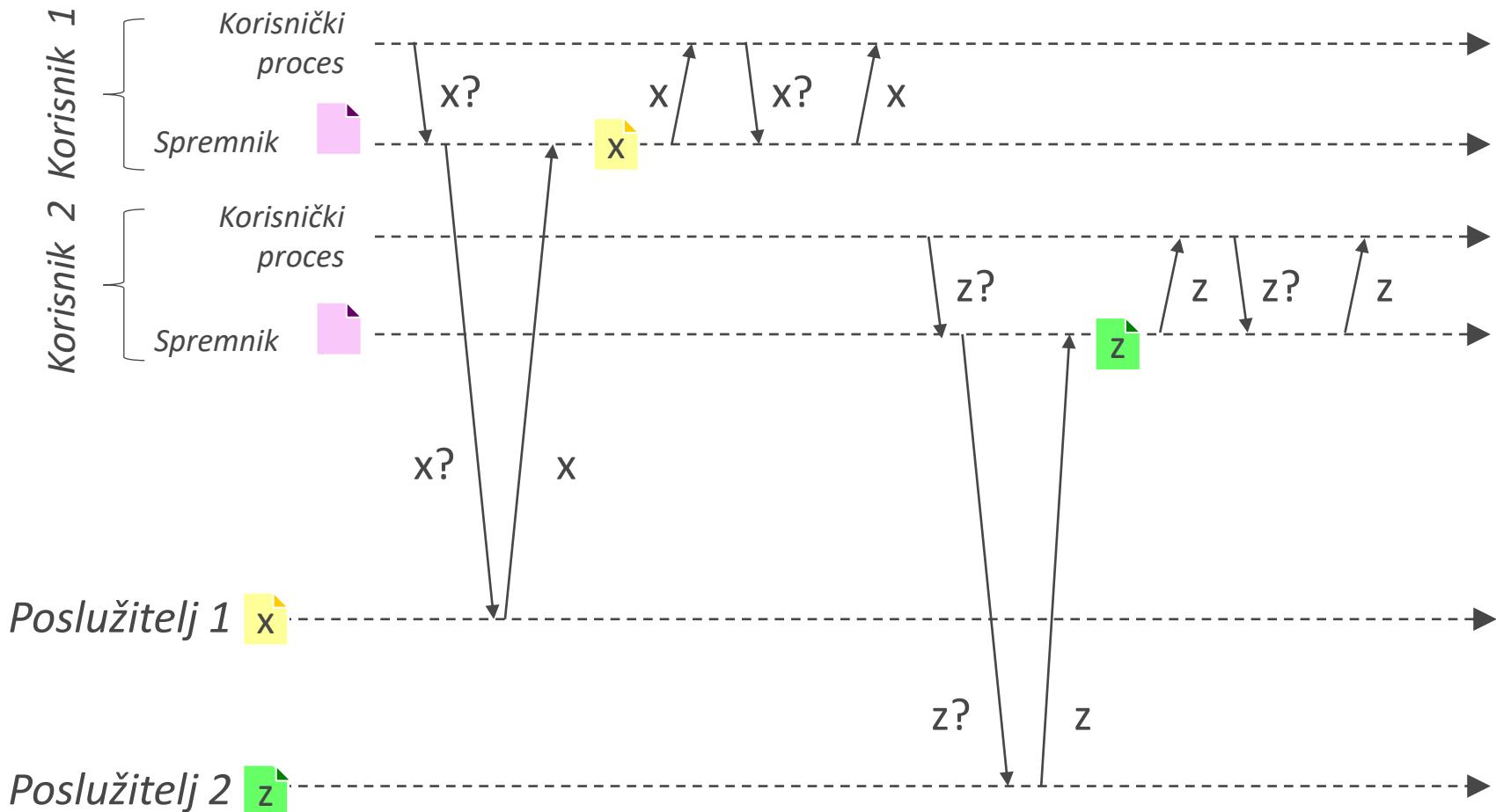
- Korisnički programi koriste lokalni spremnik
  - Dohvaćeni podaci spremaju se u lokalni spremnik
  - U slučaju potrebe za istim podacima, podaci se dohvaćaju iz lokalnog spremnika
  - Potrebno je održavati konzistentnost lokalnog spremnika s poslužiteljem s kojeg su podaci dohvaćeni
  - Lokalni spremnik može biti na istom računalu kao i korisnički programi ili na dijeljenom računalu u lokalnoj mreži
- Osnovne značajke
  - Najpovoljnije je koristiti u slučajevima kada se najčešće provode operacije čitanja
  - Smanjuje se vrijeme dohvata podataka
  - U slučaju kada nekoliko korisnika dijeli lokalni spremnik povećava se učinkovitost primjene korisničkih replika

# Korisničke replike (2)

- Korisnici ostvaruju replikaciju podataka u lokalnom spremniku
  - Svaki podatak koji korisnik želi dohvatiti s udaljenog poslužitelja prvo traži u lokalnom spremniku
  - Ako se traženi podatak ne nalazi u lokalnom spremniku dokument se dohvaća s udaljenog poslužitelja
  - Ako se podatak već nalazi u lokalnom spremniku, dokument se dohvaća iz lokalnog spremnika
- Primjer: **priručno spremište (*cache*) preglednika weba, *forward proxy cache* u lokalnoj mreži klijenata**

# Primjer uporabe korisničkih replika

$x?$  – zahtjev za dohvrat replike tipa X, x – sadržaj replike tipa X



# Uspostava replikacije podataka

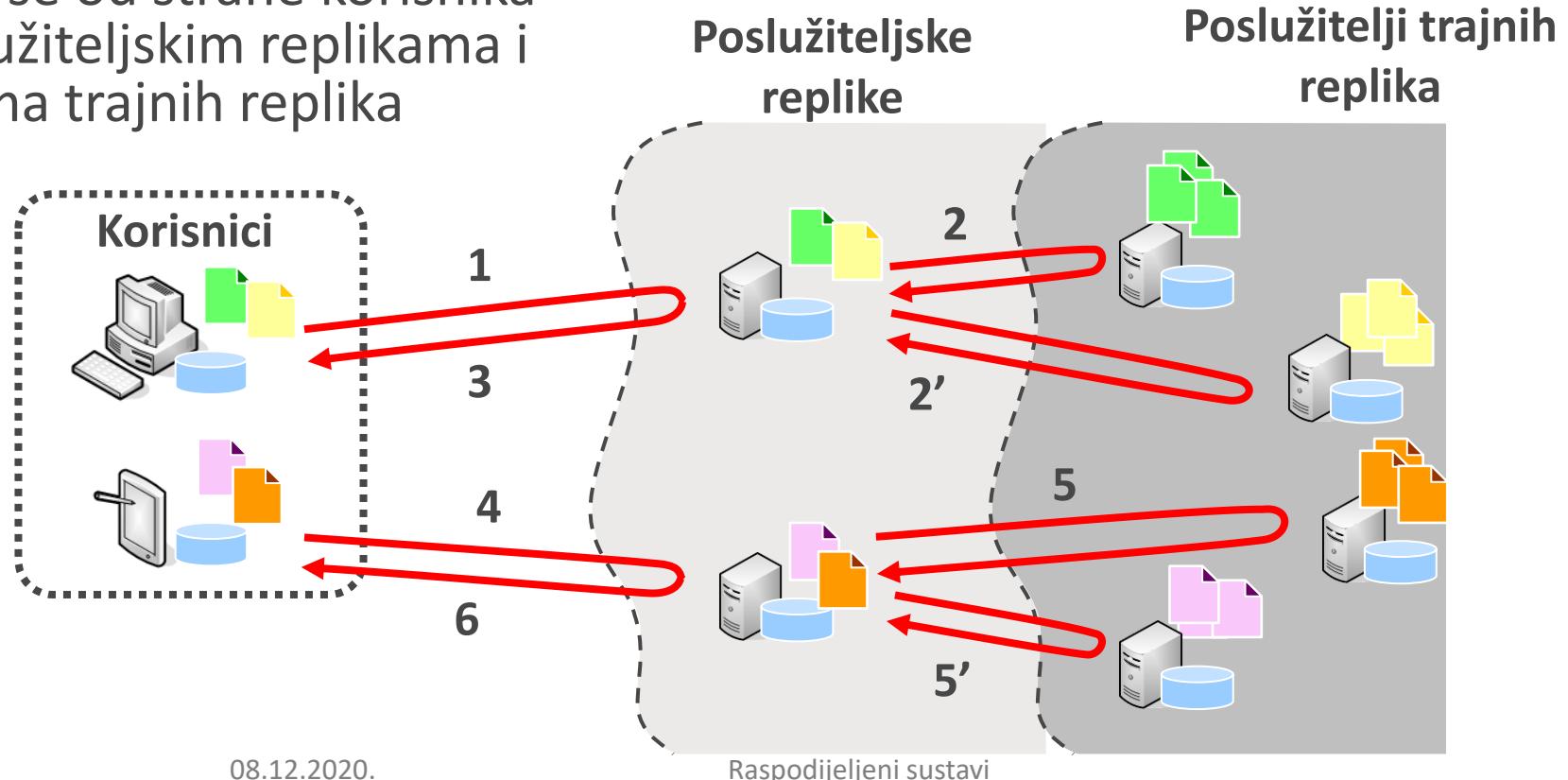
- Uvod u replikaciju podataka
- Organizacija sustava replika
- Razredba vrsta replika
  - Trajne replike, poslužiteljske replike, korisničke replike
- Održavanje konzistentnosti replika
  - Dohvaćanje promjena stanja replika
  - Prosljeđivanje promjena stanja replika
- Ostvarivanje operacija
  - Operacije pisanja sadržaja
  - Operacije čitanja sadržaja

# Održavanje konzistentnosti replika

- Obnavljanje stanja replika
  - Korisničke i poslužiteljske replike je potrebno usklađivati s promjenama stanja trajnih replika
  - Obnavljanje sadržaja replika može biti ostvareno u trenutku promjene sadržaja ili u trenutak prije ostvarivanja pristupa replici
- Osnovne metode održavanja konzistentnosti sadržaja replika
  - Dohvaćanje promjena sadržaja (*pull*)
  - Prosljeđivanje promjena sadržaja (*push*)

# Dohvaćanje promjena sadržaja (1)

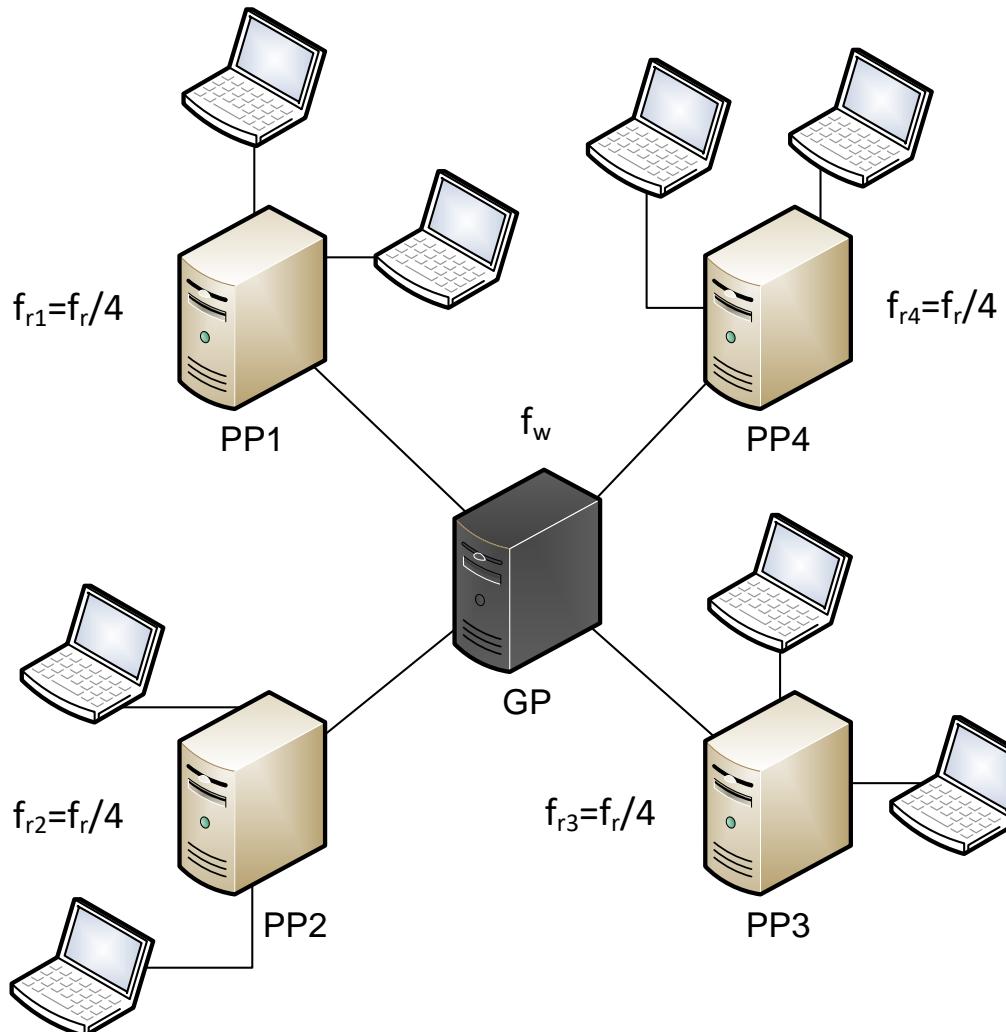
- Korisnici dohvaćaju promjene sadržaja trenutak prije pristupa replikama
- Primjenjuju se od strane korisnika prema poslužiteljskim replikama i poslužiteljima trajnih replika



# Dohvaćanje promjena sadržaja (2)

- Značajke dohvaćanja promjena sadržaja
  - Pogodno za korištenje u slučajevima **čestih izmjena** sadržaja replika tj. kad je frekvencija promjene sadržaja (pisanja) puno veća od frekvencije čitanja
  - Poslužitelji trajnih replika ne moraju znati broj i identitet korisnika
  - Smanjuje se mrežno opterećenje i rasterećuje poslužitelj replika
  - U slučajevima da lokalno stanje replike nije obnovljeno povećava se vrijeme dohvata novog stanja replika

# Primjer [1]



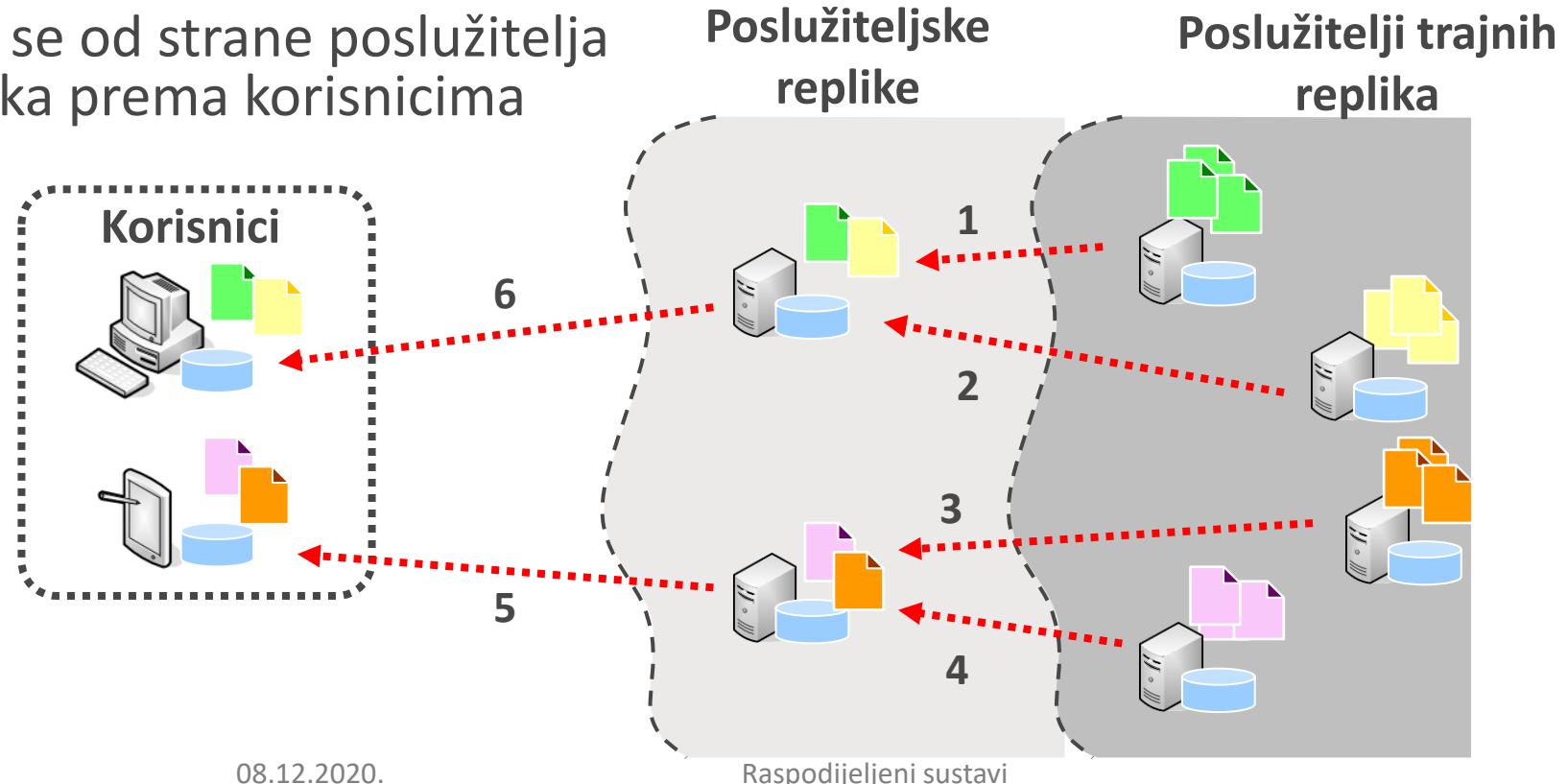
- Klijentski upiti i zahtjevi su ravnomjerno raspoređeni po pomoćnim poslužiteljima
  - $f_r$  - frekvencija čitanja
  - $f_w$  - frekvencija pisanja (tj. promjene sadržaja)
- Mrežno opterećenje GP-a?

# Primjer [2]

- Parametri
  - $f_r$  - frekvencija čitanja
  - $f_w$  - frekvencija pisanja (tj. promjene sadržaja)
  - $I_m$  - veličina poruke (tj. zahtjeva)
  - $I_r$  - veličina replike
  - $n$  – broj (ravnomjerno opterećenih) pomoćnih poslužitelja
- Opterećenje bez replikacije
  - $L1 = f_r \cdot (I_m + I_r)$
- Opterećenje s replikacijom temeljenom na dohvaćanju promjena sadržaja
  - $L2 = f_r \cdot I_m + n \cdot f_w \cdot I_r + (f_r - n \cdot f_w) \cdot I_m$  za  $f_r > n \cdot f_w$
  - $L2 = L1$  za  $f_r \leq n \cdot f_w$

# Prosljeđivanje promjena sadržaja (1)

- Poslužitelji sadržaja prosljeđuju svim replikama promjene stanja sadržaja u trenutku nastanka promjene
- Primjenjuju se od strane poslužitelja trajnih replika prema korisnicima



# Prosljeđivanje promjena sadržaja (2)

- Značajke primjene prosljeđivanja sadržaja
  - Ostvarivanje velikog stupnja konzistentnosti
  - Stvaranje dodatnog mrežnog prometa
  - Poslužitelji trajnih replika moraju imati zabilježene adrese svih replika koje obnavljaju i opis njihova stanja
  - U slučaju da neka od replika ukloni dio svojeg stanja, replika obavještava o promjenama stanja poslužitelja trajnih replika
  - Pogodno za korištenje u slučajevima s **rijetkim izmjenama** sadržaja replika i **velikog broja korisnika**

# Prosljeđivanje promjena sadržaja (3)

- Oblici prosljeđivanja promjena sadržaja
  - Prosljeđivanje novog sadržaja
    - Može se proslijediti samo izmijenjeni dio sadržaja replike ili cjelokupni sadržaj replike
    - Koristi se kad je frekvencija čitanja puno veća od frekvencije pisanja
  - Prosljeđivanje obavijesti o promjenama sadržaja
    - Bolja je od prosljeđivanja novog sadržaja ~~Koristi se~~kad je frekvencija promjene sadržaja (pisanja) puno veća od frekvencije čitanja, ali je lošija od dohvaćanja promjena sadržaja
  - Prosljeđivanje operacija za promjenu sadržaja
    - Postoje slučajevi kada se ne može primijeniti (npr. fotografije)
    - Zahtjeva dodatnu obradu na izvoru i odredištu

# Primjer [3]

- Parametri
  - $I_o$  – veličina poruke s operacijom za promjenu sadržaja
- Replikacija temeljena na prosljeđivanju novog sadržaja
  - $L3 = n \cdot f_w \cdot I_r$
- Replikacija temeljena na prosljeđivanju obavijesti o promjenama sadržaja
  - $L4 = n \cdot f_w \cdot I_m + n \cdot f_w \cdot (I_m + I_r)$  za  $f_r > n \cdot f_w$
  - $L4 = n \cdot f_w \cdot I_m + f_r \cdot (I_m + I_r)$  za  $f_r \leq n \cdot f_w$
- Replikacija temeljena na prosljeđivanju operacija za promjenu sadržaja
  - $L5 = n \cdot f_w \cdot I_o$

# Uspostava replikacije podataka

- Uvod u replikaciju podataka
- Organizacija sustava replika
- Razredba vrsta replika
  - Trajne replike, poslužiteljske replike, korisničke replike
- Održavanje konzistentnosti replika
  - Dohvaćanje promjena stanja replika
  - Prosljeđivanje promjena stanja replika
- Ostvarivanje operacija
  - Operacije pisanja sadržaja
  - Operacije čitanja sadržaja

# Ostvarivanje operacija

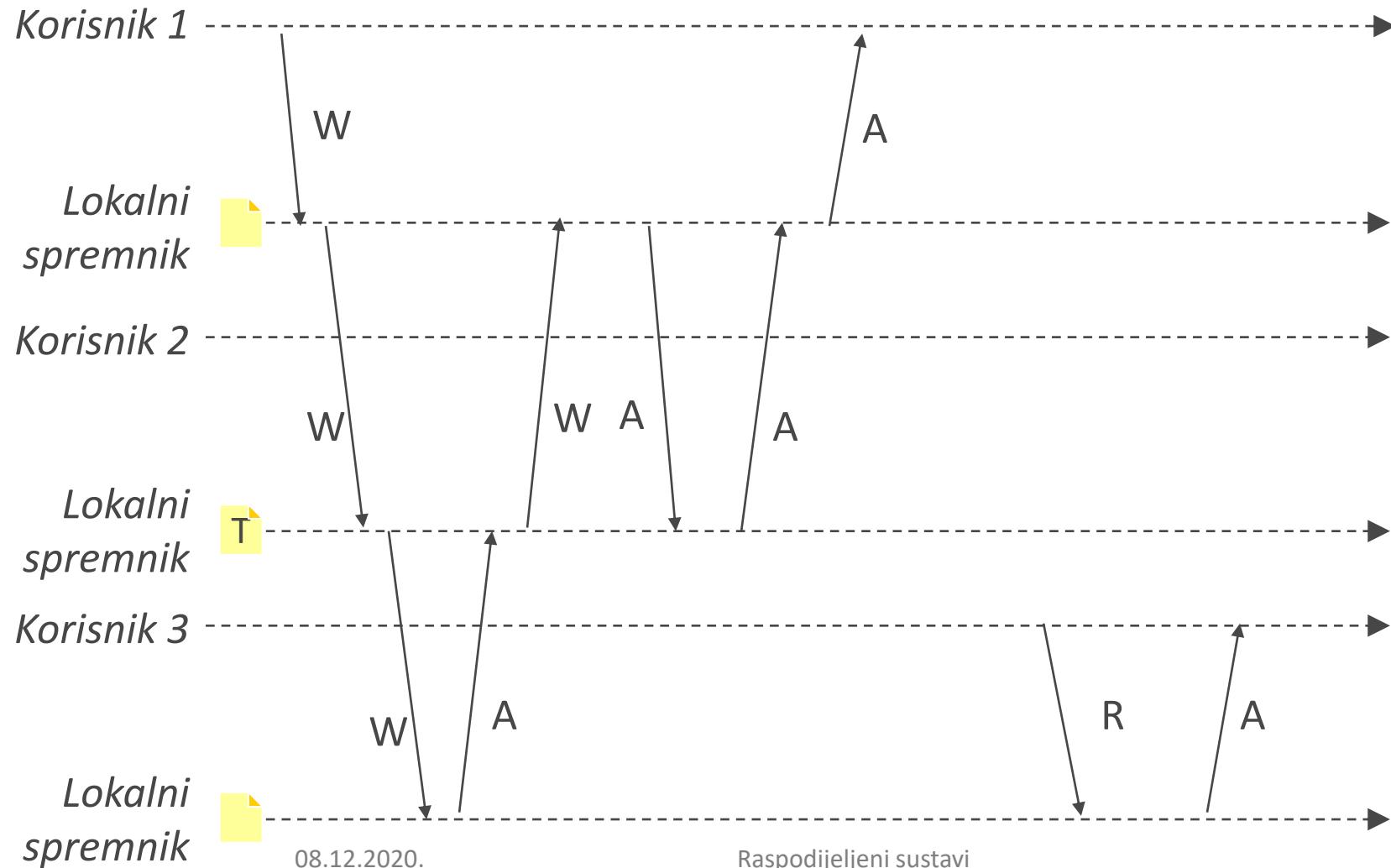
- Održavanje konzistentnosti sadržaja replika tijekom provođenja operacija čitanja i pisanja **od strane korisnika**
- Obnavljanje stanja udaljenih replika
  - Zahtjev za obnavljanje stanja prosljeđuje se udaljenim replikama koje zajednički ostvaruju
- Lokalno obnavljanje stanja replika
  - Replike se dohvaćaju na računalo domaćin te se operacije provode lokalno

# Obnavljanje stanja udaljenih replika (1)

- Model ostvarivanja operacija čitanja i pisanja
  - Zahtjevi za obnavljanje stanja prosljeđuju se udaljenoj trajnoj replici za zadani sadržaj
  - Udaljena trajna replika ostvaruje lokalnu promjenu sadržaja i prosljeđuje zahtjev za pisanje svim ostalim replikama u sustavu
  - Ostale replike nakon promjene sadržaja šalju potvrde te zatim trajna replika prosljeđuje potvrdu korisniku
  - Operacije čitanja se provode na lokalnoj replici ili bilo kojoj drugoj replici u sustavu replika

# Obnavljanje stanja udaljenih replika (2)

W – Pisanje, R – Čitanje sadržaja, A – Rezultat , F – Dohvat replike



# Obnavljanje stanja udaljenih replika (3)

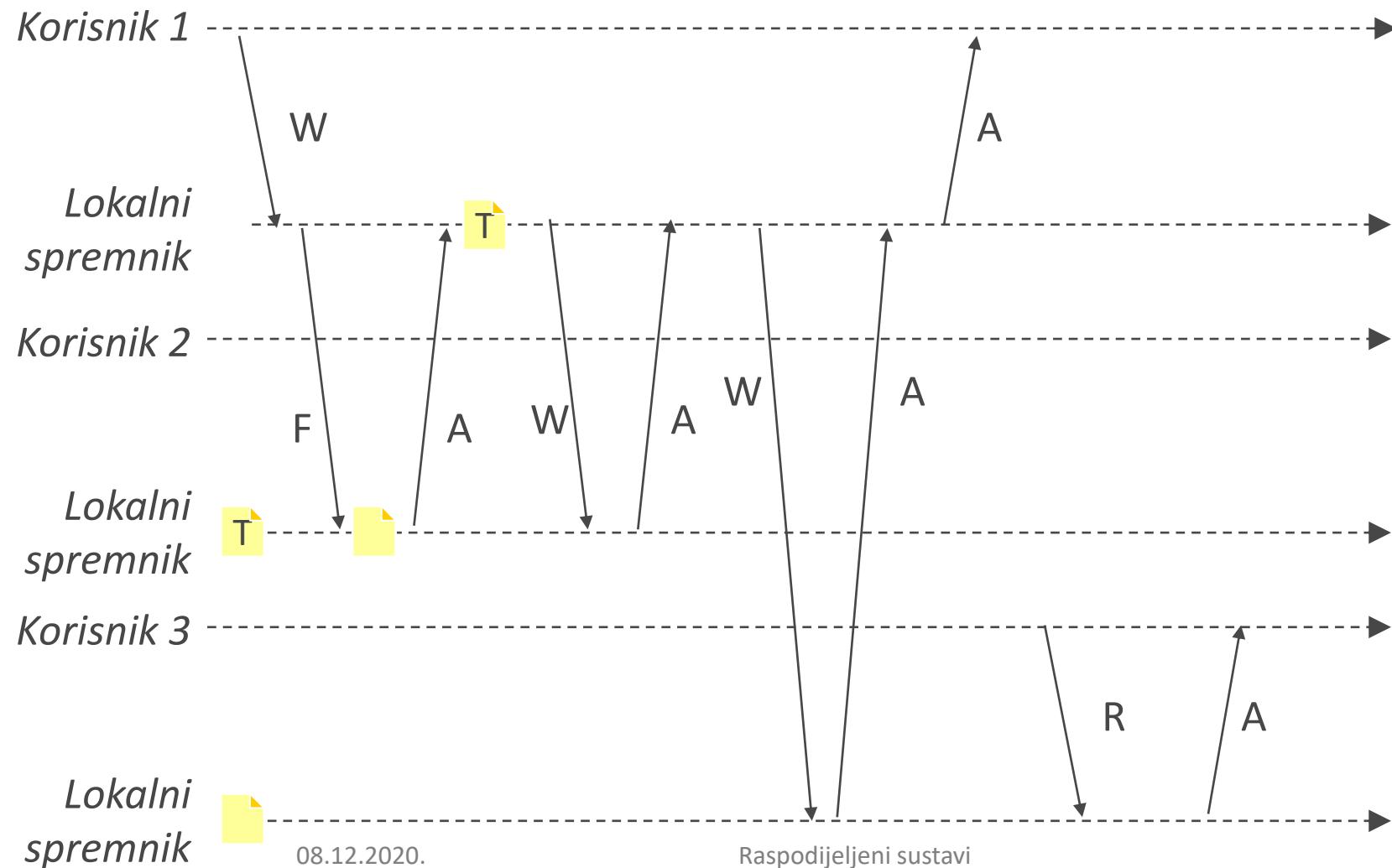
- Značajke modela obnavljanja stanja udaljenih replika
  - Omogućava uspostavu slijedne konzistentnosti obzirom da samo glavna replika provodi operacije pisanja
  - Svi korisnici doživljavaju jednak redoslijed izvođenja operacija pisanja u vremenu bez obzira putem koje replike dohvaćaju sadržaj
  - U slučaju velikog broj pomoćnih replika, izvođenje operacije pisanja može zahtijevati značajnu količinu vremena potrebu za provođenje operacije pisanja na svim pomoćnim replikama

# Lokalno obnavljanje stanja replika (1)

- Model ostvarivanja operacija čitanja i pisanja
  - Trajna replika se dohvaća na računalo domaćin
  - Provodi se operacija pisanja
  - Promjene ostvarene na trajnoj replici u lokalnom spremniku proslijeđuju se svim ostalim replikama u sustavu
  - Operacije čitanja se provode na lokalnoj replici ili bilo kojoj drugoj replici u sustavu replika

# Lokalno obnavljanje stanja replika (2)

W – Pisanje, R – Čitanje sadržaja, A – Rezultat , F – Dohvat replike



# Lokalno obnavljanje stanja replika (3)

- Značajke modela lokalnog obnavljanja stanja replika
  - Uzastopne operacije pisanja mogu biti provedene u kratkom vremenu na računalu domaćinu
  - Rezultati provođenja uzastopnih operacija mogu biti agregirani u jednu operaciju pisanja koja se provodi na pomoćnim replikama u raspodijeljenoj okolini
  - Korisnici koji čitaju sadržaj mogu pristupiti vlastitim lokalnim replikama neovisno o trajnoj replici

# Usporedba modela replikacije u sustavima

|                        | Baza podataka     | P2P            | WWW            |
|------------------------|-------------------|----------------|----------------|
| Upravljanje            | Centralizirano    | Raspodijeljeno | Centralizirano |
| Povezanost             | Čvrsta            | Labava         | Čvrsta         |
| Raspoređivanje         | Centralizirano    | Raspodijeljeno | Centralizirano |
| Vrsta većine operacija | Pisanje i čitanje | Čitanje        | Čitanje        |
| Pouzdanost             | Predvidiva        | Nepredvidiva   | Predvidiva     |
| Raznorodnost sredstava | Ne                | Da             | Ne             |

# Dodatne informacije (1)

- Knjige
  - S. Tanenbaum, M. van Steen: "Distributed Systems: Principles and Paradigms", Prentice Hall, 2002. (Poglavlje: Consistency and Replication)
  - H. Attiya, J. Welch: "Distributed Computing: Fundamentals, Simulations, and Advanced Topics", Wiley, 2004. (Poglavlje: Distributed Shared Memory)

# Dodatne informacije (2)

- Znanstveni radovi
  - S. Goel, S. Buyya: "Data Replication Strategies in Wide-Area Distributed Systems", u: Enterprise Service Computing: From Concept to Deployment (ur. R. Qiu), pp.211-241, IGI Global, 2006.  
<http://www.buyya.com/papers/DataReplicationInDSChapter2006.pdf>
  - P. Padmanabhan, L. Gruenwald, A. Vallur, M. Atiquzzaman: "A Survey of Data Replication Techniques for Mobile and Ad-Hoc Network Databases", Journal of Very Large Data Bases, Vol. 17, pp. 1143-1164, 2008.  
<http://www.cs.ou.edu/~database/documents/VLDB08.pdf>

# Pitanja za učenje i ponavljanje

- Objasnite vezu između replikacije i konzistentnosti.
- Objasnite način replikacije u HDFS-u.
- Objasnite poštuje li se slijedna konzistentnost u slijedu izvođenja operacija prikazanom na slajdu 17.
- Objasnite poštuje li se povezana konzistentnost u slijedu izvođenja operacija prikazanom na slajdu 20. ukoliko pretpostavimo da je operacija pisanja podatka B (uzročno) povezana s operacijom pisanja podatka A.
- Skicirajte slijedni dijagram korištenja poslužiteljskih replika.
- Skicirajte slijedni dijagram lokalnog obnavljanja stanja replike.

# Pitanja za učenje i ponavljanje - nastavak

- U sustavu replika koji se sastoji od glavnog poslužitelja i n=6 podjednako opterećenih pomoćnih poslužitelja, izračunajte prosječno mrežno opterećenje glavnog poslužitelja za sljedeće metode održavanja konzistentnosti: a) pull, b) push s prosljeđivanjem novog sadržaja c) push s prosljeđivanjem operacija za promjenu sadržaja i d) push s prosljeđivanjem obavijesti o promjeni sadržaja. Pri tome prepostavite da korisnike poslužuju samo pomoćni poslužitelji, da je prosječna frekvencija upita  $f_u = 500$  upita/s, prosječna frekvencija promjena  $f_p = 4$  promjene/min te da su prosječne veličine replika, upita/odgovora i operacija za promjenu sadržaja replika  $I_r = 800$  kb,  $I_p = 10$  kb i  $I_o = 600$  kb.



SVEUČILIŠTE U ZAGREBU



**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

# Raspodijeljeni sustavi

**10. Vrednovanje nefunkcijskih obilježja  
raspodijeljenih sustava**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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



*U slučaju 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.*

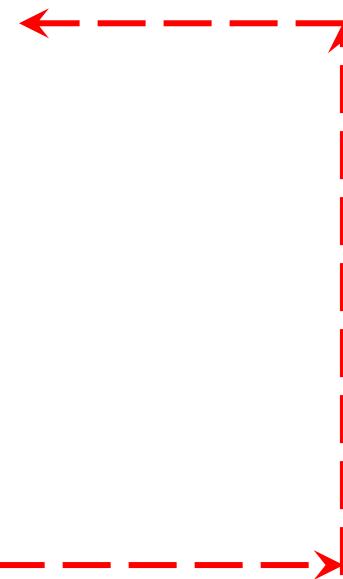
*Tekst licence preuzet je s <http://creativecommons.org/>*

# Sadržaj predavanja

- Životni ciklus sustava, nefunkcijske karakteristike sustava i važnost njihovog vrednovanja u praksi
- Postupci analize nefunkcijskih karakteristika
- Pouzdanost, raspoloživost i ukupna cijena vlasništva raspodijeljenog sustava
- Performance raspodijeljenog sustava
- Prirodne granice rasta ubrzanja i kapaciteta
- Modeliranje raspodijeljenih sustava mrežom repova

# Životni ciklus sustava

- Definicija zahtjeva na sustav
- Analiza funkcijskih obilježja (**ŠTO** sustav radi) i nefunkcijskih obilježja (**KAKO** sustav radi)
- Razvoj odabranog rješenja
- Ispitivanje
- Rad
- Mjerenja
- Modifikacija (ako zahtjevi nisu ispunjeni)



# Najvažniji nefunkcijski zahtjevi

- Performance
  - propusnost (broj zahtjeva/sekunda) i kašnjenje
- Skalabilnost
  - mogućnost povećanja ulaznih zahtjeva uz iste performance ili uz proširenje arhitekture
  - vrste:
    - Vertikalna – povećanje HW-a
    - Horizontalna – dodavanje elemenata
- Raspoloživost (*availability*)
  - postotak vremena kada sustav radi
- Pouzdanost (*reliability*)
  - Vjerovatnost da će sustav ispravno raditi u nekom vremenskom periodu
- Kapacitet
  - Isporuka funkcionalnosti za zadani ulazni teret (broj zahtjeva/sekunda)
- Sigurnost
  - zaštita sustava od malicioznih napada
- Održivost (maintainability)
  - jednostavnost nadogradnje sustava
- Upravljivost (manageability)
  - Jednostavno praćenje sustava dok radi s ciljem traženja pogrešaka i analize rada sustava
- Proširivost (extensibility)
  - Jednostavnost dodavanja novih funkcionalnosti
- Obnovljivost (recovery)
  - Mogućnost obnavljanja funkcionalnosti sustava nakon ispada
- Interoperabilnost
  - Mogućnost razmjene informacija s drugim sustavima
- Uporabljivost (usability)
  - Jednostavnost korištenja

# Nefunkcijske karakteristike sustava

- Nefunkcijske karakteristike sustava skupno se nazivaju **kvaliteta usluge** (QoS – *Quality of Service*)
- Definicija jamčene kvalitete usluge naziva se **ugovor o razini usluge** (SLA – *Service Level Agreement*)
  - SLA je dio ugovora između davatelja i korisnika usluga
  - SLA je sve češće je i dio opisa posla internog ICT-odjela
- Tri važne kategorije kvalitete usluge:
  - Performance (*performance*): vrijeme odziva, propusnost, kapacitet
  - Pouzdanost/raspoloživost (*reliability/availability*)
  - Ukupni trošak vlasništva (TCO – *Total Cost of Ownership*)

# Primjer: web-aplikacija za trgovinu

**Organizacija prodaje putem Interneta**

**Aplikacija za prodaju ima sljedeće značajke:**

- Neuspješni posjeti zbog loše kvalitete usluge
  - 60 % kupaca napušta web-stranicu aplikacije ako je odziv aplikacije **između 4 i 6 sekundi**
  - 95 % kupaca napušta web-stranicu aplikacije ako je odziv aplikacije **veći od 6 sekundi**
- Uspješni posjeti s ostvarenom prodajom
  - 5 % kupaca od svih koji su posjetili web-stranicu aplikacije kupi proizvode za **prosječnu cijenu 1200 kn**

# Primjer: analiza značajki web-aplikacije (1)

- Projektiranje i održavanje web-aplikacije ostvaruje se u skladu s očekivanim brojem i porastom broja korisnika
- Ako se broj posjeta web-aplikaciji – promet poveća za 30%, 60% ili 90%:
  - Hoće li odziv aplikacije biti zadovoljavajući?
  - U kojim će uvjetima odziv aplikacije preći u nezadovoljavajuće područje?
  - Koliki gubitak prihoda uzrokuje gubitak kupaca zbog slabog odziva aplikacije?
  - Koja ulaganja su potrebna da se uz povećanje prometa zadrži sav posao?
  - Kada će se, uz trenutačni trend, potreba za kapacitetom udvostručiti?

## Primjer: analiza značajki web-aplikacije (2)

|                                | Povećanje broja korisnika |           |           |            |
|--------------------------------|---------------------------|-----------|-----------|------------|
|                                | Danas                     | +30 %     | +60 %     | +90 %      |
| Maks. posjeta/sat              | 900.00                    | 1,170.00  | 1,440.00  | 1,710.00   |
| Vrijeme odziva (s)             | 2.96                      | 3.80      | 5.31      | 8.83       |
| Izgubljeni kupci (%)           | 0.00                      | 0.00      | 60.00     | 95.00      |
| Mogući broj prodaja / sat (kn) | 45.00                     | 58.50     | 72.00     | 85.50      |
| Mogući prihod / sat (kn)       | 54,000.00                 | 70,200.00 | 86,400.00 | 102,600.00 |
| Stvarni prihod / sat (kn)      | 54,000.00                 | 70,200.00 | 34,560.00 | 5,130.00   |
| Izgubljeni prihod / sat (kn)   | 0.00                      | 0.00      | 51,840.00 | 97,470.00  |

Poduzeće će izgubiti više od 95% mogućeg prometa na Internetu, ako se broj potencijalnih kupaca udvostruči!

# Primjer: analiza značajki web-aplikacije (3)

- Na temelju prikazanih rezultata može se zaključiti da:
  - Pogrešno projektirana aplikacija može imati katastrofalne posljedice na poslovanje
- Upozorenje:
  - linearna ekstrapolacija najčešće ne daje dovoljno točne rezultate
  - što ako veći broj zahtjeva odstupa od statističke srednje vrijednosti: veći udjel zahtjeva na repu raspodjele vjerojatnosti nego kod normalne raspodjele → dugi rep (*long tail*)
- Na ovom predmetu naučit ćete kako pristupiti vrednovanju performansi raspodijeljenih sustava i planiranju rasta kapaciteta sustava

# Postupci analize nefunkcijskih karakteristika

- **Intuicija i iskustvo**
  - Raspodijeljeni sustavi pokazuju izrazito nelinearno ponašanje pa su procjene vrlo teške
- **Modeliranje**
  - Predočavanje sustava matematičkim modelom
  - Podrazumijeva razvoj modela koji opisuje ovisnost performanci o pojedinim parametrima sustava
- **Simulacija**
  - Postupak kojim se oponaša rad sustava
  - Najtočnija metoda, često preskupa za upotrebu

# Razvoj modela raspodijeljenog sustava

- Razumijevanje funkciranja sustava
- Modeliranje tereta – opterećenja sustava
- **Mjerenja u radu sustava radi utvrđivanja parametara tereta – ovo ćemo vidjeti**
- Razvoj modela performanci
- Verifikacija i validacija modela performanci
  - Verifikacija: provjera ispravnosti (radi li ispravno?)
  - Validacija: provjera valjanosti (radi li ono što se očekuje?)
- Analiza mogućih scenarija promjena u budućnosti
- Procjena promjena tereta u budućnosti
- Predviđanje performanci sustava nakon puštanja u rad te u budućnosti

# Modeli tereta sustava

## Prirodni modeli tereta

- stvarne aplikacije

## Umjetni modeli tereta

- umjetne aplikacije – ogledna vrijednost (*benchmark*)
  - *Standard Performance Evaluation Corporation* (SPEC) [www.spec.org](http://www.spec.org)
  - *Transaction Processing Performance Council* (TPC-C) [www\(tpc.org/tpcc](http://www(tpc.org/tpcc)

## Neizvodivi modeli tereta

- uslužni zahtjevi, intenzitet nailaska/dolaska zahtjeva, razina konkurentnog izvođenja, ...

# Primjer: utvrđivanje značajki realnog tereta (1)

- Srednja vrijednost parametara ne mora biti reprezentativna ako se pojedinačne vrijednosti nalaze u grupama koje se značajno razlikuju po vrijednostima
- U takvom slučaju potrebno je provesti grupiranje i utvrditi značajke za svaku grupu
- Postoje različite metode grupiranja i programi koji obavljaju grupiranje, a najčešće se koristi grupiranje na temelju Euklidske udaljenosti:

$$d = \sqrt{\sum_{n=1}^K (x_{in} - x_{jn})^2}$$

$K$  = broj parametara

# Primjer: utvrđivanje značajki realnog tereta (2)

- Teret web-poslužitelja sastoji se od sljedećih grupa zahtjeva:

| Dokument | Veličina (KB) | Broj pristupa |
|----------|---------------|---------------|
| 1        | 12            | 281           |
| 2        | 150           | 28            |
| 3        | 5             | 293           |
| 4        | 25            | 123           |
| 5        | 7             | 259           |
| 6        | 4             | 241           |
| 7        | 35            | 75            |

- Grupiraj teret u tri odvojene grupe

# Primjer: utvrđivanje značajki realnog tereta (3)

- Budući da su vrijednosti jako različite treba prvo obaviti promjenu mjerila. U ovom slučaju koristimo  $\log_{10}$ .

| Dokument | Veličina (KB) | Broj pristupa |
|----------|---------------|---------------|
| 1        | 1.08          | 2.45          |
| 2        | 2.18          | 1.45          |
| 3        | 0.70          | 2.47          |
| 4        | 1.40          | 2.09          |
| 5        | 0.85          | 2.41          |
| 6        | 0.60          | 2.38          |
| 7        | 1.54          | 1.88          |

# Primjer: utvrđivanje značajki realnog tereta (4)

- Podrazumijevajući da je težište grupe od jedne točke ta točka, izračunavamo Euklidske udaljenosti između težišta:

| Grupa | G1 | G2   | G3   | G4   | G5   | G6   | G7   |
|-------|----|------|------|------|------|------|------|
| G1    | 0  | 1.49 | 0.38 | 0.48 | 0.24 | 0.48 | 0.74 |
| G2    |    | 0    | 1.79 | 1.01 | 1.01 | 1.64 | 1.83 |
| G3    |    |      | 0    | 0.79 | 0.16 | 0.13 | 1.03 |
| G4    |    |      |      | 0    | 0.64 | 0.85 | 0.26 |
| G5    |    |      |      |      | 0    | 0.25 | 0.88 |
| G6    |    |      |      |      |      | 0    | 1.07 |
| G7    |    |      |      |      |      |      | 0    |

- Budući da je udaljenost između G3 i G6 najmanja, svrstavamo ih u novu grupu G36 i računamo joj težište:

$$(0.7+0.6)/2 = 0,65 \quad \text{i} \quad (2.47+2.38)/2 = 2,43$$

# Primjer: utvrđivanje značajki realnog tereta (5)

- Sad računamo Euklidske udaljenosti između novih težišta:

| Grupa | G1 | G2   | G36  | G4   | G5   | G7   |
|-------|----|------|------|------|------|------|
| G1    | 0  | 1.49 | 0.43 | 0.48 | 0.24 | 0.74 |
| G2    |    | 0    | 1.81 | 1.01 | 1.64 | 0.76 |
| G36   |    |      | 0    | 0.82 | 0.19 | 1.05 |
| G4    |    |      |      | 0    | 0.64 | 0.26 |
| G5    |    |      |      |      | 0    | 0.88 |
| G7    |    |      |      |      |      | 0    |

- Budući da je udaljenost između G36 i G5 najmanja izračunamo težište nove grupe G365:

$$(0.65+0.85)/2 = 0.75 \quad \text{i} \quad (2.43+2.41)/2 = 2.42$$

# Primjer: utvrđivanje značajki realnog tereta (6)

- Postupak ponavljamo dok ne dobijemo željene tri grupe:

| Grupa | G1365 | G2   | G47  |
|-------|-------|------|------|
| G1365 | 0     | 1.60 | 0.72 |
| G2    |       | 0    | 0.89 |
| G47   |       |      | 0    |

- Nakon vraćanja parametara u izvorno mjerilo dobivamo novi model tereta:

| Grupa | Tip dokumenta | Velicina (KB) | Broj zahtjeva |
|-------|---------------|---------------|---------------|
| G1356 | Mali          | 8.19          | 271.51        |
| G47   | Srednji       | 20.58         | 96.05         |
| G2    | Veliki        | 150.0         | 28            |

- Ovakvo grupiranje daje puno realističnije postavke za modeliranje sustava

# Najčešće pogreške kod modeliranja

- Presložena analiza problema
- Nije definiran specifični cilj
- Prejudiciranje rezultata
  - model treba dokazati da je “naše” rješenje bolje od “njihovog”!
- Nedovoljno razumijevanje sustava
- Neadekvatne mjere performansi
- Nereprezentativni teret
- Neuključivanje važnih parametara
- Promatranje u krivom intervalu vrijednosti parametara
- Pogrešno rukovanje ekstremnim vrijednostima
- Nedovoljno promatranje evolucije sustava i tereta
- Kriva interpretacija rezultata

# Pouzdanost sustava

## Pouzdanost:

- Vjerojatnost da sustav funkcionira u definiranom vremenskom intervalu  $T$  pod danim uvjetima okružja

U radu sustava događaju se kvarovi koji se moraju otkloniti, što se opisuje parametrima:

- Srednje vrijeme između kvarova (MTBF – *Mean Time Between Failure* )
- Srednje vrijeme popravka (MTTR – *Mean Time To Repair*)

# Raspoloživost sustava

## Raspoloživost:

- Vjerojatnost da će sustav funkcionirati u trenutku  $t$   
(ili: postotak ukupnog vremena koje je sustav u radu)

Računanje raspoloživosti:  $MTBF/(MTBF + MTTR)$

- Raspoloživost od 0,9999 znači da će sustav biti izvan funkcije:  $(1 - 0,9999) \times 30 \times 24 \times 60 = 4,32 \text{ min/mjesec}$
- Raspoloživost serijske kombinacije dva podsustava jednaka je umnošku raspoloživosti pojedinih podsustava:

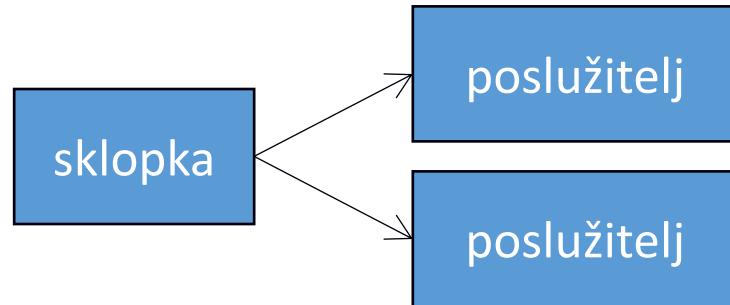
$$R_s = R_1 \times R_2$$

- Raspoloživost paralelne kombinacije dva podsustava jednaka je:

$$R_p = R_1(1 - R_2) + R_2(1 - R_1) + R_1R_2$$

(pretpostavka: jedan raspoloživi podsustav dovoljan za ispravno funkcioniranje sustava)

# Primjer: raspoloživost web-poslužitelja



Izračunajte raspoloživost sustava koji uključuje dva paralelna web-poslužitelja s raspoloživošću od 0,99 te jedne mrežne sklopke za raspodjelu tereta između poslužitelja s MTBF = 1 godina i MTTR = 2 sata.

- Raspoloživost sklopke:  $(365 \times 24)/(365 \times 24 + 2) = 0,9998$
- Raspoloživost 2 paralelna poslužitelja:  
 $0,0099 + 0,0099 + 0,9801 = 0.9999$
- Raspoloživost sustava:  $0,9998 \times 0,9999 = 0,9997$

# Ukupni trošak vlasništva (1)

## Kapitalni trošak (**CAPEX** – *Capital Expenditure*)

- trošak nabavke i izvedbe sustava (oprema)
- trošak amortizacije – zavisi o propisanom vremenu trajanja
  - za računalne sustave 3 godine (33,33% nabavne cijene godišnje)

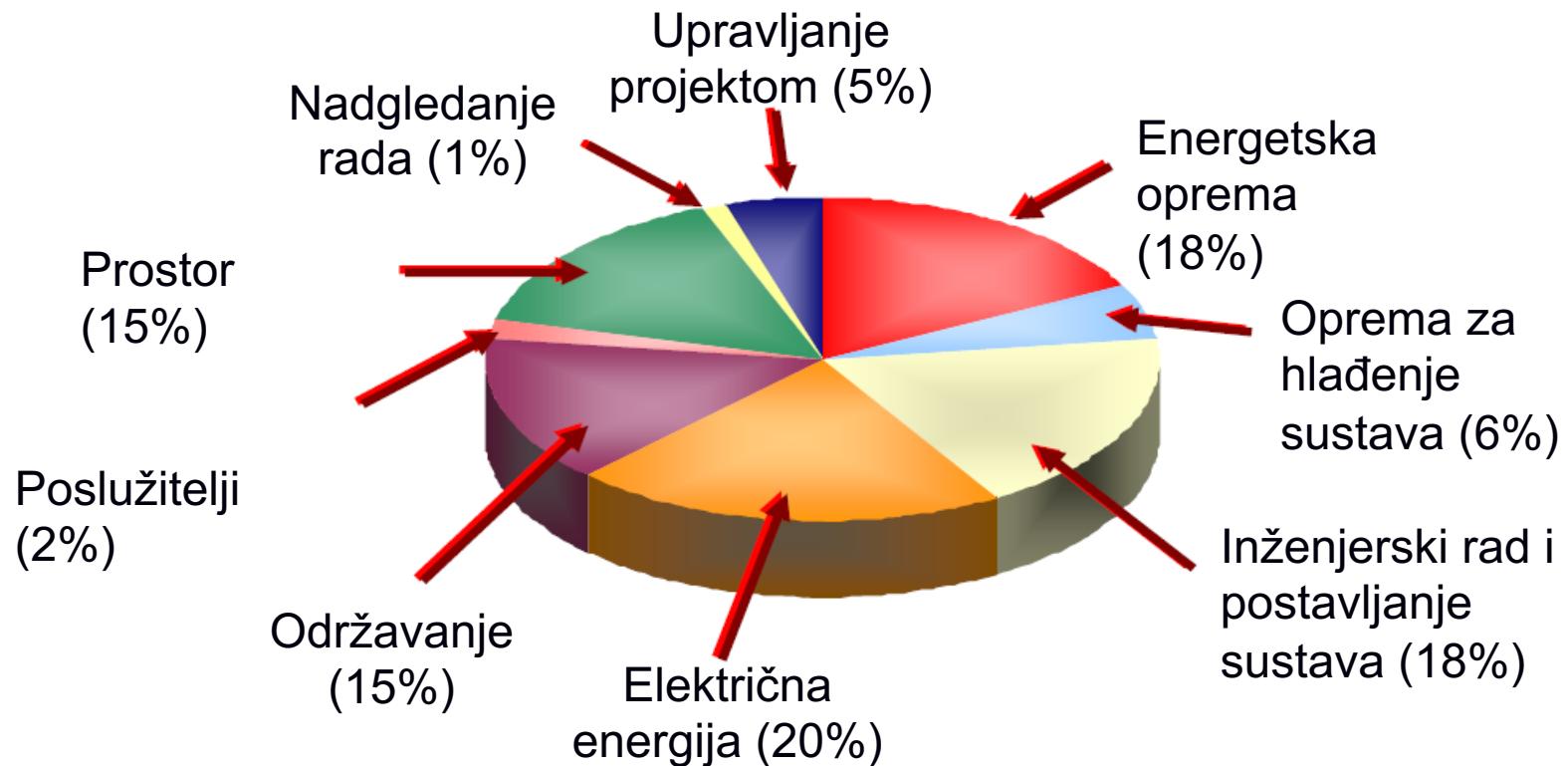
## Operativni trošak (**OPEX** – *Operating Expenditure*)

- trošak rada/pogona sustava:
  - zaposlenici (ICT odjel)
  - prostor i infrastruktura
  - režije: energija, komunikacijske usluge, fizička sigurnost, ...

# Ukupni trošak vlasništva (2)

## Prosječni trošak uporabe poslužiteljskog sustava:

- CAPEX/OPEX = 50/50, kroz tri godine korištenja, ovisno o „geografiji” i godini



Izvor: Intel (internetska anketa 2008.)

# Mogući načini ostvarenja sustava

## Izgradnja vlastitog poslužitelja

## Iznajmljivanje poslužitelja

- Udomljivanje sustava (*hosting*)

- Udomitelj: pruža i upravlja fizičkom infrastrukturom (zgrada, napajanje, pristup Internetu, poslužitelji)
- Zakupnik: postavlja i upravlja sredstvima koja se poslužuju
- Primjer: *web-hosting*

## Računalni oblak (*computing cloud*)

- Infrastruktura kao usluga (*Infrastructure as a Service*, IaaS)
- Platforma kao usluga (*Platform as a Service*, PaaS)
- Softver kao usluga (*Software as a Service*, SaaS)

# Performance sustava

## Vrijeme odziva sustava (*response time*)

- Vrijeme potrebno da sustav odgovori na zahtjev za posluživanjem (npr. vrijeme između pritiska na miša i pojave web-stranice na zaslonu)

## Propusnost sustava (*throughput*)

- Broj posluženih zahtjeva u jedinici vremena (različito od 1/vrijeme odziva, jer se zahtjevi mogu posluživati paralelno)
  - Propusnost je funkcija tereta i kapaciteta sustava

## Kapacitet sustava (*capacity*) = maksimalna moguća propusnost sustava

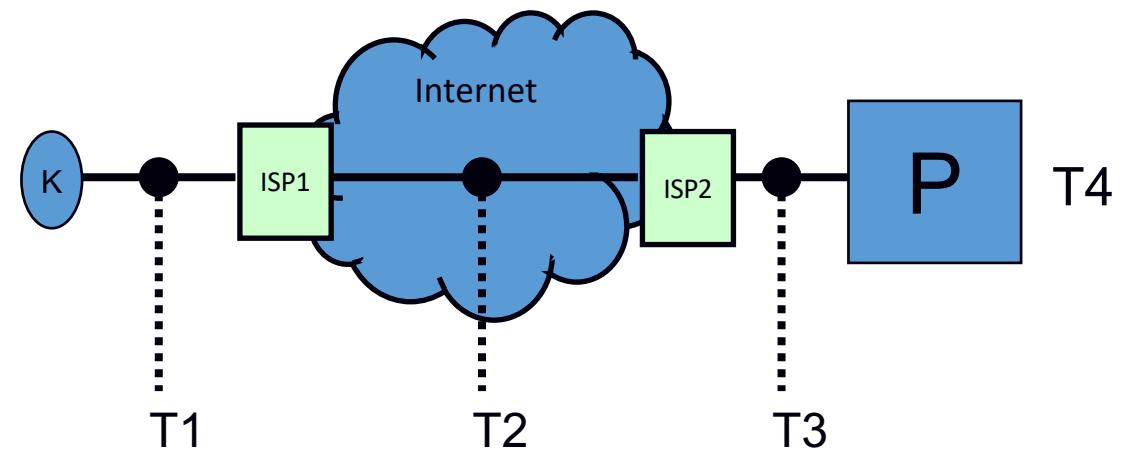
- Sustav je zauzet 100% vremena

# Vrijeme odziva u raspodijeljenom sustavu

## Vrijeme odziva

- Vrijeme u mreži ( $T_1 + T_2 + T_3$ )

- Kašnjenje
  - Vrijeme prijenosa



- Vrijeme na poslužitelju ( $T_4$ )

- Vrijeme posluživanja (CPU, disk, LAN, ...)
  - Vrijeme čekanja na resurse (CPU, disk, LAN, baza podataka, ...)

# Primjer: dobavljanje web-stranice <https://www.websitepulse.com>

24/7 Live chat 24/7 1-888-WSPULSE Login Sign Up

**WebSitePulse™**  
take IT easy

◀ Perform New Test Start a FREE Trial ▶

**www.fer.unizg.hr**

Test performed from Munich, Germany on December 12, 2020 at 11:16

**Test Results** **Recommendations**

|  |                                                                                                                                 |
|--|---------------------------------------------------------------------------------------------------------------------------------|
|  | <b>performance grade</b><br><b>79</b> / 100<br>first visit<br><b>9.39</b> load time <b>7.86</b> DOM ready <b>4 MB</b> page size |
|  | <b>requests</b><br><b>70</b><br>return visit<br><b>3.43</b> load time <b>3.10</b> DOM ready <b>476 KB</b> page size             |

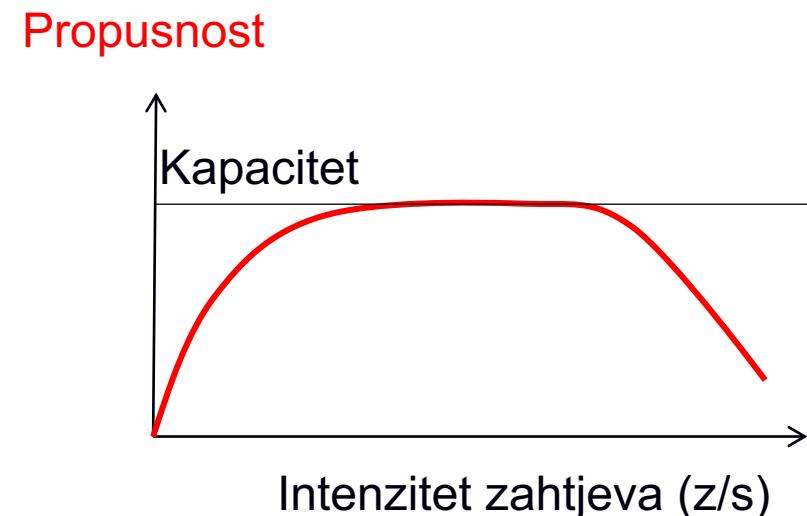
initial load screenshot return load screenshot Share your results [Twitter](#) [Facebook](#) [8+](#) get a permalink download report data

Performance grade 79 out of 100. Your website's performance grade is very good.  
See our recommendations on how to improve performance →

# Propusnost raspodijeljenog sustava

## Broj posluženih zahtjeva u jedinici vremena (Z/S)

- „Zahtjev“ ovisi o razini sustava na kojoj se promatra
- Primjeri tipičnih zahtjeva i jedinica za propusnost:
  - Broj transakcija u sekundi
  - Broj pretinaca u sekundi
  - Broj web-stranica u sekundi
  - Broj poruka u sekundi
  - Broj paketa u sekundi
  - Broj instrukcija u sekundi
  - ...



# Primjer: izračun propusnosti

Kolika je maksimalna propusnost tj. kapacitet diska u sustavu za *on-line* transakcije?

- U/I operacija diska traje prosječno 10 ms
- Iskorištenje diska je 100% (tj. stalno zauzet)
- Maksimalna propusnost tj. kapacitet diska:  
 $1/0.01 = 100$  operacija u sekundi

Napomena: Kapacitet se određuje kod 100% zaposlenosti!

# Ostvarivanje razmjernog rasta (skaliranje)

## Osnovni modeli ostvarivanja razmjernog rasta kapaciteta sustava

- Vertikalno skaliranje podrazumijeva prijelaz na poslužitelja s većim kapacitetom
- Horizontalno skaliranje podrazumijeva dodavanje poslužitelja (obično istog kapaciteta).
- Skaliranje prema gore (većem kapacitetu) je jednako važno kao i skaliranje prema dolje (manjem kapacitetu) zbog potrebe prilagodbe troškova prihodima!

# Poboljšanje performansi

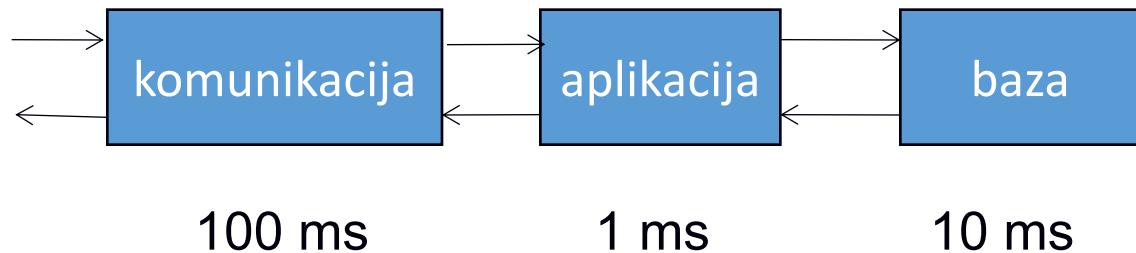
## Arhitekturne promjene za poboljšanje performansi:

- Serijsko preklapanje (*pipeline*)
- Paralelno preklapanje
- Paralelna obrada
- Privremena pohrana (*cache*)

## Primjer: web-poslužitelj s 3 modula

- Mrežni modul s odzivom od 100 ms
- Modul aplikacije s odzivom od 1 ms
- Modul baze podataka s odzivom od 10 ms
- Napomena: deterministički pristup, bez razmatranja raspodjele nailaska zahtjeva, vremena posluživanja i čekanja te interakcije paralelnih aktivnosti

# Serijsko preklapanje



- Svakih 100 ms prihvaca se novi zahtjev, dok prethodni još nije obrađen i završen
- Serijskim preklapanjem ne smanjuje se vrijeme odziva, a povećava se propusnost:

$$T_{odziva} = 100 \text{ ms} + 1 \text{ ms} + 10 \text{ ms} = 111 \text{ ms}$$

$$\text{Propusnost} = 1/0,1\text{s} = 10/\text{s}$$

# Mjerenje odziva web-aplikacije

- alat [Apache JMeter](#)
- za testiranje performansi statičkih i dinamičkih resursa
- podržani protokoli:
  - Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, ...)
  - SOAP / REST Webservices
  - FTP
  - Database via JDBC
  - LDAP
  - Message-oriented middleware (MOM) via JMS
  - Mail - SMTP(S), POP3(S) and IMAP(S)
  - Native commands or shell scripts
  - TCP



# Web-aplikacija u Spring Bootu u Javi

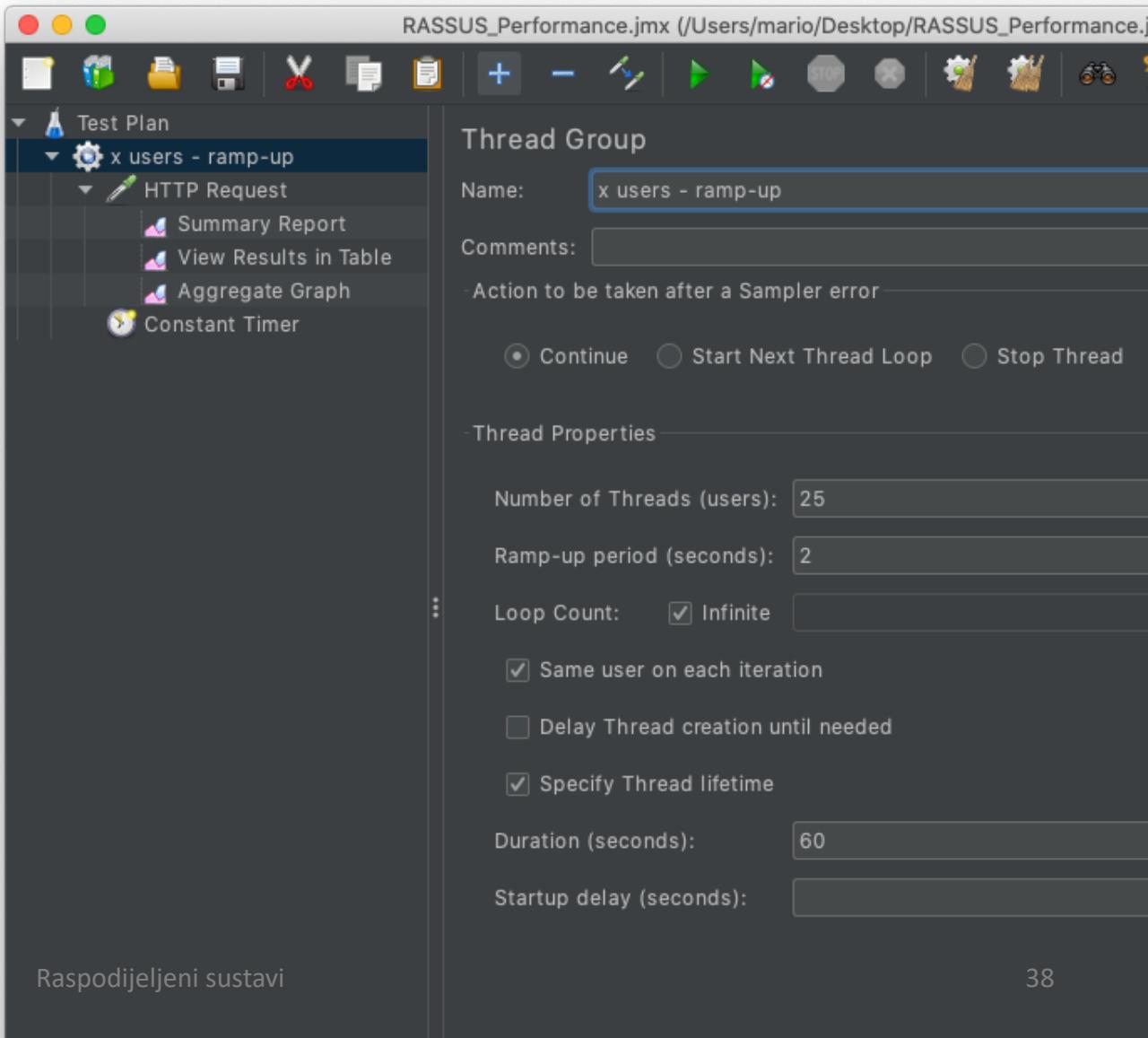
- Problemi kod mjerjenja:
  - Koliko paralelnih niti/dretvi koristi?
  - Koliko konekcija ima prema bazi podataka?
  - Spremanje podataka iz baze u privremenu memoriju?
    - Koliko često se to događa (vjerojatnost)?
  - JVM radi optimizacije za vrijeme rada. Kako anulirati taj dio?
  - Kako mjeriti pojedine komponente (mrežni dio, izvršavanje, dohvaćanje iz baze)?

# Web-aplikacija u Spring Bootu u Javi

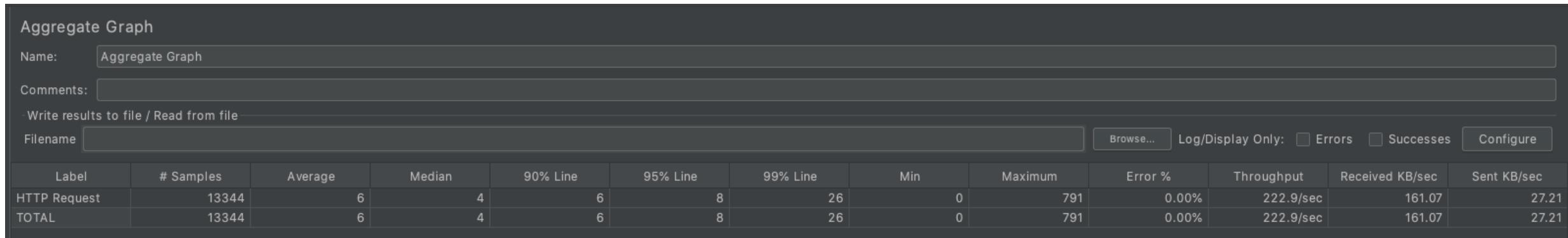
- Koliko paralelnih niti/dretvi koristi?
  - Postaviti konfiguraciju: `server.tomcat.max-threads`
- Koliko konekcija ima prema bazi podataka?
  - Postaviti konfiguraciju (v 2.+): `spring.datasource.hikari.maximum-pool-size`
- Spremanje podataka iz baze u privremenu memoriju?
  - Koliko često se to događa?
    - Unutar jedne sjednice nije moguće isključiti, ali se može napraviti analiza koda i na osnovu toga izračunati vjerojatnost
- JVM radi optimizacije za vrijeme rada. Kako anulirati taj dio?
  - Treba „zagrijati“ JVM da ne radi prevodenje *byte* koda (cca 8-10 tisuća zahtjeva) tj. prije mjerena poslati zahtjeve na poslužitelj, a nakon toga raditi mjerena
- Kako mjeriti pojedine komponente (mrežni dio, izvršavanje, dohvaćanje iz baze)?
  - JRF (Java Flight Control) - <2% utjecaja na izvođenje aplikacije
  - Koristiti Aspektno orijentirano programiranje (AOP) za ubacivanje kontrolnih točki

# JMeter test

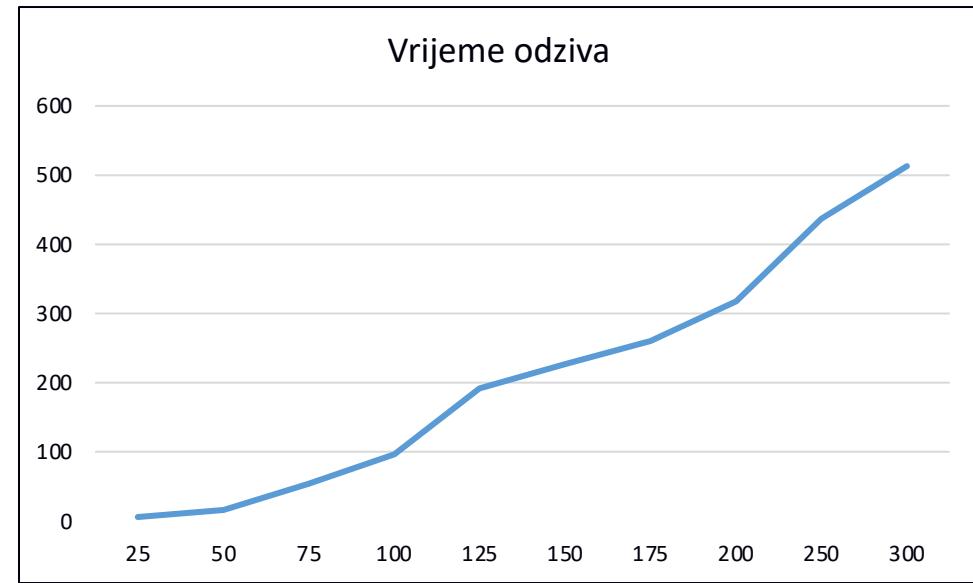
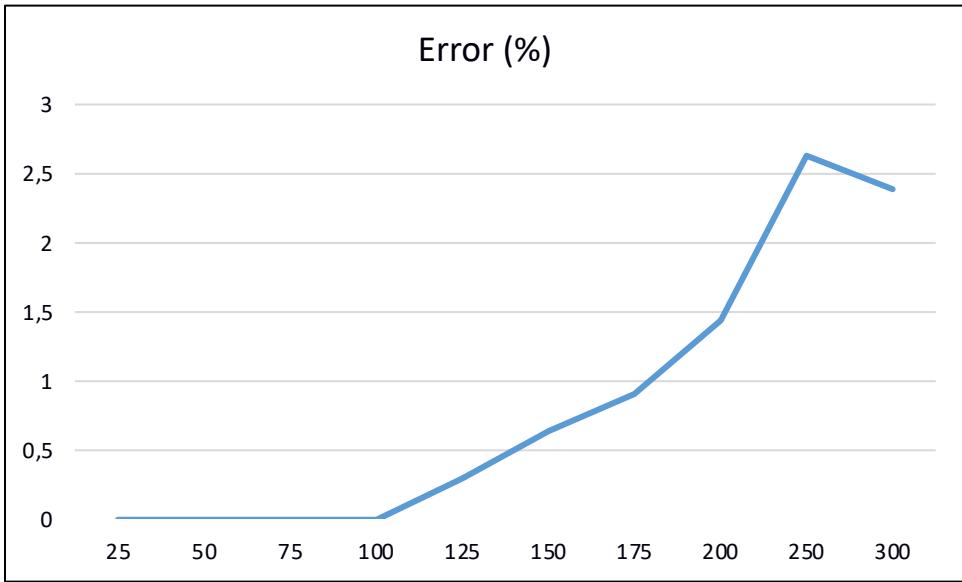
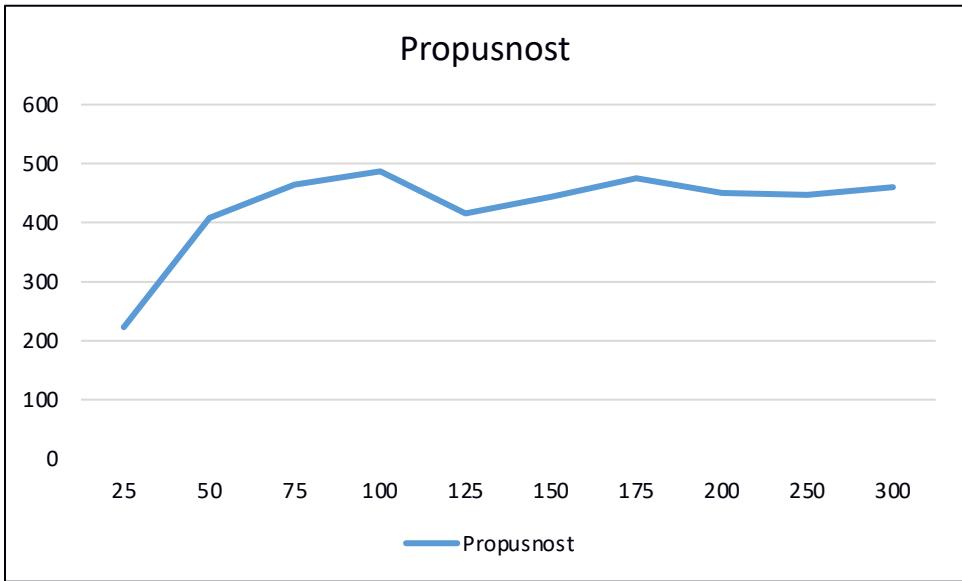
- Korisnik je jedna nit/dretva
- Šalje se jedan GET zahtjev
  - Dohvaća podatke iz baze i računa
- Pauza između zahtjeva 100ms
- Test traje 60s
- Mijenjamo broj korisnika 25-300



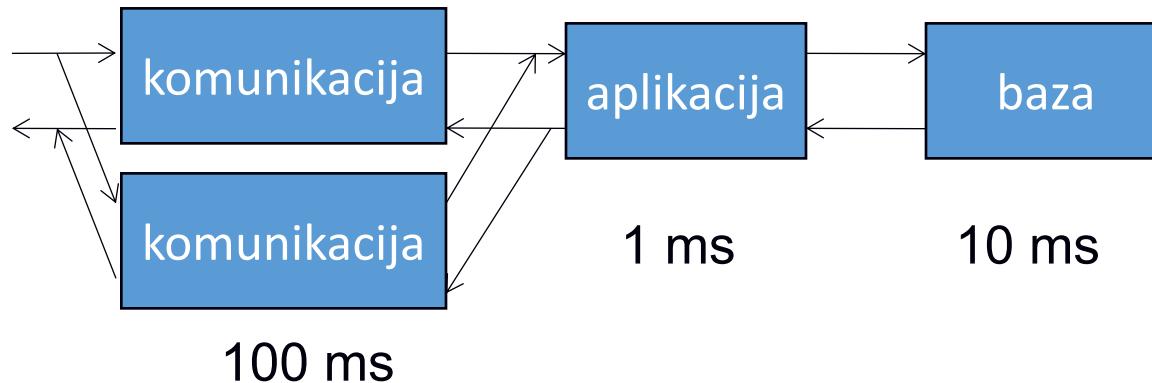
# JMeter – rezultati



|                     |     |      |      |      |      |      |      |      |      |      |
|---------------------|-----|------|------|------|------|------|------|------|------|------|
| Korisnici           | 25  | 50   | 75   | 100  | 125  | 150  | 175  | 200  | 250  | 300  |
| Intenzitet dolazaka | 250 | 500  | 750  | 1000 | 1250 | 1500 | 1750 | 2000 | 2500 | 3000 |
| Propusnost          | 223 | 408  | 464  | 487  | 416  | 444  | 475  | 450  | 447  | 460  |
| Error (%)           | 0   | 0,00 | 0,00 | 0,00 | 0,30 | 0,64 | 0,91 | 1,44 | 2,63 | 2,39 |
| Vrijeme odziva      | 6   | 16   | 54   | 97   | 192  | 227  | 260  | 318  | 437  | 513  |



# Paralelno preklapanje

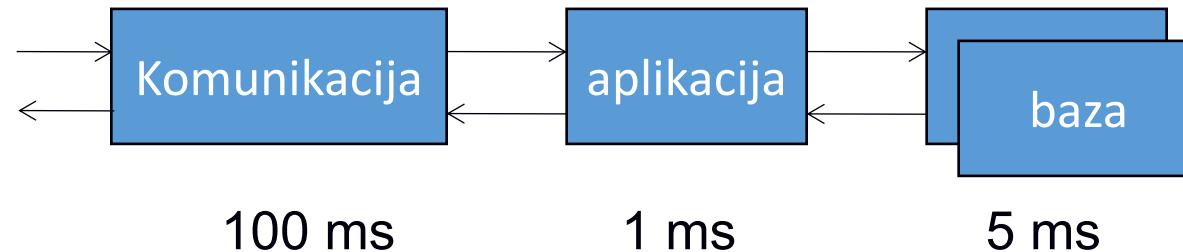


- Istodobno (paralelno) se mogu prihvati 2 zahtjeva
- Paralelnim preklapanjem komunikacije ne smanjuje se vrijeme odziva, a povećava se propusnost:

$$T_{odziva} = 100 \text{ ms} + 1 \text{ ms} + 10 \text{ ms} = 111 \text{ ms}$$

$$\text{Propusnost} = 2/0,1s = 20/s$$

# Paralelna obrada



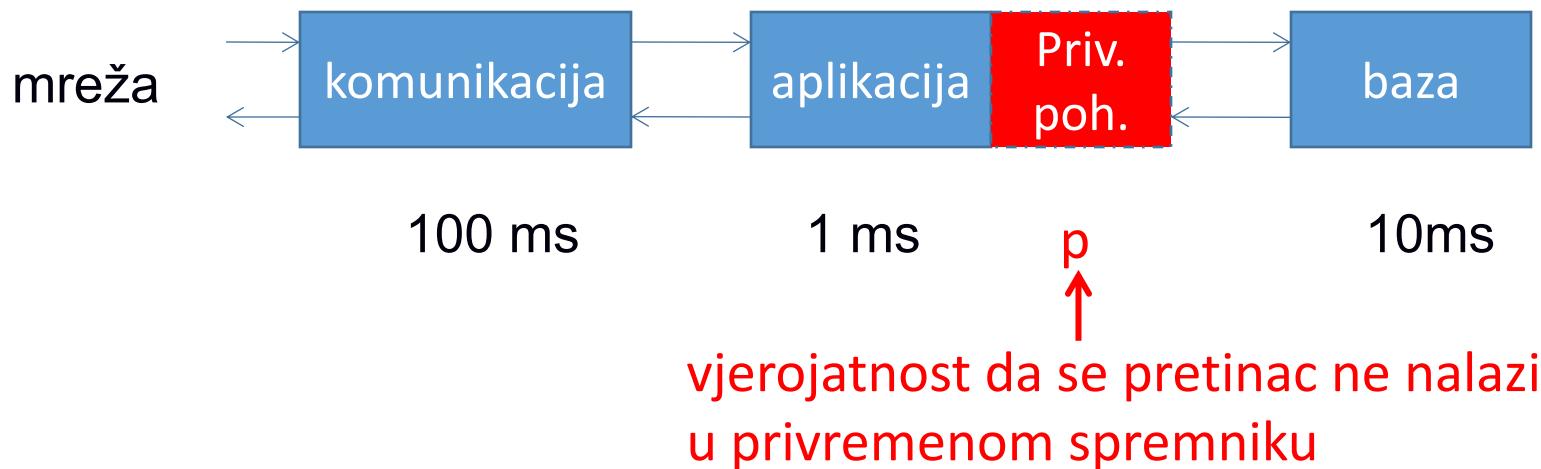
- Zapis i istog pretinca raspodijeljeni su na dva fizička diska pa se dohvaćaju paralelno
- Paralelnom obradom ne utječe se na propusnost, ali smanjuje se vrijeme odziva:

$$\text{Propusnost} = 1/0,1\text{s} = 10/\text{s}$$

$$T_{\text{odziva}} = 106 \text{ ms}$$

Ubrzanje!

# Privremena pohrana



- Korišteni pretinac se pohranjuje u privremeni spremnik
- Privremena pohrana ne utječe na propusnost, ali smanjuje vrijeme odziva:

$$\text{Propusnost} = 1/0,1\text{s} = 10/\text{s}$$

$$T_{\text{odziva}} = 101 + 10(1 - p) \text{ ms}$$

Ubrzanje!

# Ubrzanje obrade zahtjeva

## Ubrzanje (*speedup*):

- Skraćenje vremena odziva, može se postići paralelnom obradom zahtjeva
- Izražava se kao omjer vremena odziva jednog podsustava ( $T_1$ ) i vremena odziva  $p$  paralelnih podsustava ( $T_p$ ) uz jednak teret:

$$S(p) = T_1/T_p$$

- Teorijski model ubrzanja: Amdahlov zakon

# Skaliranje donosi nove izazove...

- Raspoređivanje opterećenja u sustavu
  - Osigurava jednakomjerno opterećenje paralelnih podsustava – uravnoteženje opterećenja
- Raspoređivanje podataka
  - Osiguravanje koherencije (replikacija istovrsnih podataka)
  - Osiguravanje podjele (federacija odvojenih podataka )
- Protokoli za sinkronizaciju rada grupe
  - Osiguravanje vremenskog slijeda
  - Zajamčena dostava

# ...ali i nove mogućnosti

## Visoka raspoloživost sustava (*high availability*)

- Izvedba putem neosjetljivosti na pogreške (*fault tolerance*)
  - Neosjetljivost na pogreške omogućuje ostvarivanje dostupnosti web-aplikacija uz prisutnost pogrešaka u podsustavima
- Budući da povećanje kapaciteta obično zahtjeva ostvarenje replikacije, replikacija se može iskoristiti za ostvarivanje neosjetljivosti na pogreške
- Visoki stupanj eliminacije pogrešaka je veoma skup, često preskup

# Procjena potrebnog kapaciteta sustava

- Ako se kapacitet sustava  $C$  mjeri u pravilnim intervalima, dugoročna potreba za kapacitetom se može procijeniti podrazumijevajući model eksponencijalnog trenda:

$$C_{\text{budući}} = C_{\text{sadašnji}} \times e^{(LT)}$$

L - trend rasta

T - vrijeme kroz koje se trend aproksimira

- Vrijeme do udvostručenja kapaciteta može se izračunati:
  - $T_{\text{dvostruko}} = (\ln 2)/L$

# Literatura, ...

**Sadržaj ovog predavanja nastao je na temelju:**

- N. J. Gunther: "**The practical performance analyst**", *Mcgraw Hill i Authors Choice Press*, 1998 i 2000. (poglavlja 2 i 3)
- D. A. Menasce, V.A.F. Almeida: "**Capacity planning for web services**", *Prentice Hall*, 2002 (poglavlja 1 i 5)
  - D. A. Menasce, V.A.F. Almeida, „Capacity Planning: An Essential Tool for Managing Web Services“, *IT professional*, Vol. 4, No. 4, 2002., pp. 33-38.”
- D. F. Vrsalovic, et. al: "**Performance prediction and calibration for a class of multiprocessors**“, *IEEE Transactions on Computers*, Volume: 37 Issue: 11 , Nov. 1988, pp. 1353 -1365

# ... dodatno za one koji žele dalje istraživati

- A. O. Allen: "**Probability, Statistic, and Queueing Theory with Computer Science Applications**", Academic Press 1978.
- S. Jones, R. Willenborg, K. Hygh: "**Performance analysis for Java Web Sites**", Addison Wesley, 2003
- S. Sounders: "**High Performance Web Sites**", O'Reilly, 2007.
- T. Schlossnagle: "**Scalable Internet Architectures**", Sams Publishing, 2007.
- D. A. Menasce, V.A.F. Almeida, L.W. Dowdy: "**Performance by Design**", Prentice Hall, 2004

# Rekapitulacija (1)

- Koji su elementi životnog ciklusa raspodijeljenog sustava?
  - Definicija zahtjeva, analiza rješenja, sinteza, ispitivanje, rad, mjerenja i modifikacija zahtjeva
- Koji su najvažniji nefunkcijski zahtjevi?
  - Performance, raspoloživost i ukupna cijena vlasništva
- Kako se oni skupno zovu?
  - Kvaliteta usluge (QoS)
- Što je to ugovor o razini usluge (SLA)?
  - Ugovor između korisnika i davatelja usluge koji definira razinu usluge
- Zašto je vrednovanje performansi važno?
  - Zbog planiranja kapaciteta koji je potreban za uspješan rad
- Koje metode analize se koriste u praksi?
  - Iskustvo, modeliranje i simulacija
- Kakve vrste modela tereta postoje u praksi?
  - Prirodne aplikacije, umjetne aplikacije (*benchmarks*) i neizvodivi modeli opisani intenzitetom zahtjeva, prosječnim vremenom obrade i sl.

## Rekapitulacija (2)

- Koji su koraci pri razvoju modela sustava?
  - Razumijevanje funkciranja, modeliranje tereta, mjerjenje sustava u pogonu radi utvrđivanja parametara tereta, razvoj modela, verifikacija i validacija, analiza mogućih scenarija promjena, prognoza promjena tereta u budućnosti, prognoza performansi sustava nakon puštanja u pogon te u budućnosti
- Koje su najčešće greške kod modeliranja?
  - Presložena analiza, nema specifičnog cilja, prejudiciranje, nedovoljno razumijevanje sustava, neadekvatne mjere nereprezentativni teret, neuključivanje važnih parametara, promatranje u krivom intervalu vrijednosti parametara, krivo baratanje ekstremima, nedovoljno promatranje evolucije sustava i tereta, kriva interpretacija rezultata
- Što definiraju pojmovi MTBF i MTTR?
  - Srednje vrijeme između pogrešaka i srednje vrijeme popravka

## Rekapitulacija (3)

- Kako je definirana raspoloživost sustava?

$$D = \text{MTBF}/(\text{MTBF} + \text{MTTR})$$

Postotak vremena u kojemu je sustav na raspolaganju korisnicima

- Kako se računa raspoloživost paralelno ( $D_p$ ) i serijskih ( $D_s$ ) povezanih sustava?

$$D_p = (1-D_1)*D_2 + (1-D_2)*D_1 + D_1*D_2$$

$$D_s = D_1*D_2$$

- Koje su dvije osnovne grupe troškova za gradnju i pogon web-sustava i kako su obično raspodijeljeni?

Kapitalni i operativni troškovi, grubo raspodijeljeni 50/50 kroz tri godine.

- Objasnite pojam vremena odziva i kapaciteta

Vrijeme odziva određuje odziv na jedan zahtjev

Kapacitet odgovara maksimalnom broju zahtjeva koji se mogu obraditi u jedinici vremena

## Rekapitulacija (4)

- Koje se metode za poboljšanje performansi koriste u arhitekturi raspodijeljenih sustava?  
Serijsko preklapanje, paralelno preklapanje, paralelno izvođenje i privremena pohrana
- Kako je definirano ubrzanje?  
Ubrzanje je omjer vremena izvođenja na jednom i više paralelnih podsustava
- Što su tipični uzroci natjecanja za sredstva (*contention*) ili potrebe za usklađivanjem podataka (*coherence*)?
  - Zajedničke funkcije i varijable u operacijskom sustavu
  - Izmjena zajedničkih podataka koji se mijenjaju u privremenim spremnicima (*cache*)
  - Promet podataka u/iz glavne memorije
  - Čekanje na ulaz/izlaz
  - Sinkronizacijski primitivi

## Rekapitulacija (5)

- Koliko razina protokola uključuju web-aplikacije kojima se pristupa putem lokalnih mreža *Ethernet* spojenih na Internet?
  - Četiri: *Ethernet*, IP, TCP i HTTP
- Što uključuje vrijeme odziva web-aplikacije?
  - Vrijeme pristupa klijenta (klijent – ISP), vrijeme prijenosa paketa od klijentovog do poslužiteljevog ISP-a (ISP – ISP), vrijeme dostupa do poslužitelja (ISP – poslužitelj), ISP – davatelj internetske usluge (*Internet Service Provider*)
- Kako se može ostvariti razmjeran rast aplikacije?
  - Vertikalno ili horizontalno
- Koje probleme donosi horizontalan rast?
  - Usklađivanje, raspodjelaa tereta i raspodjelaa podataka



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**11. Modeli za vrednovanje performansi  
raspodijeljenih sustava**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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



*U slučaju 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.*

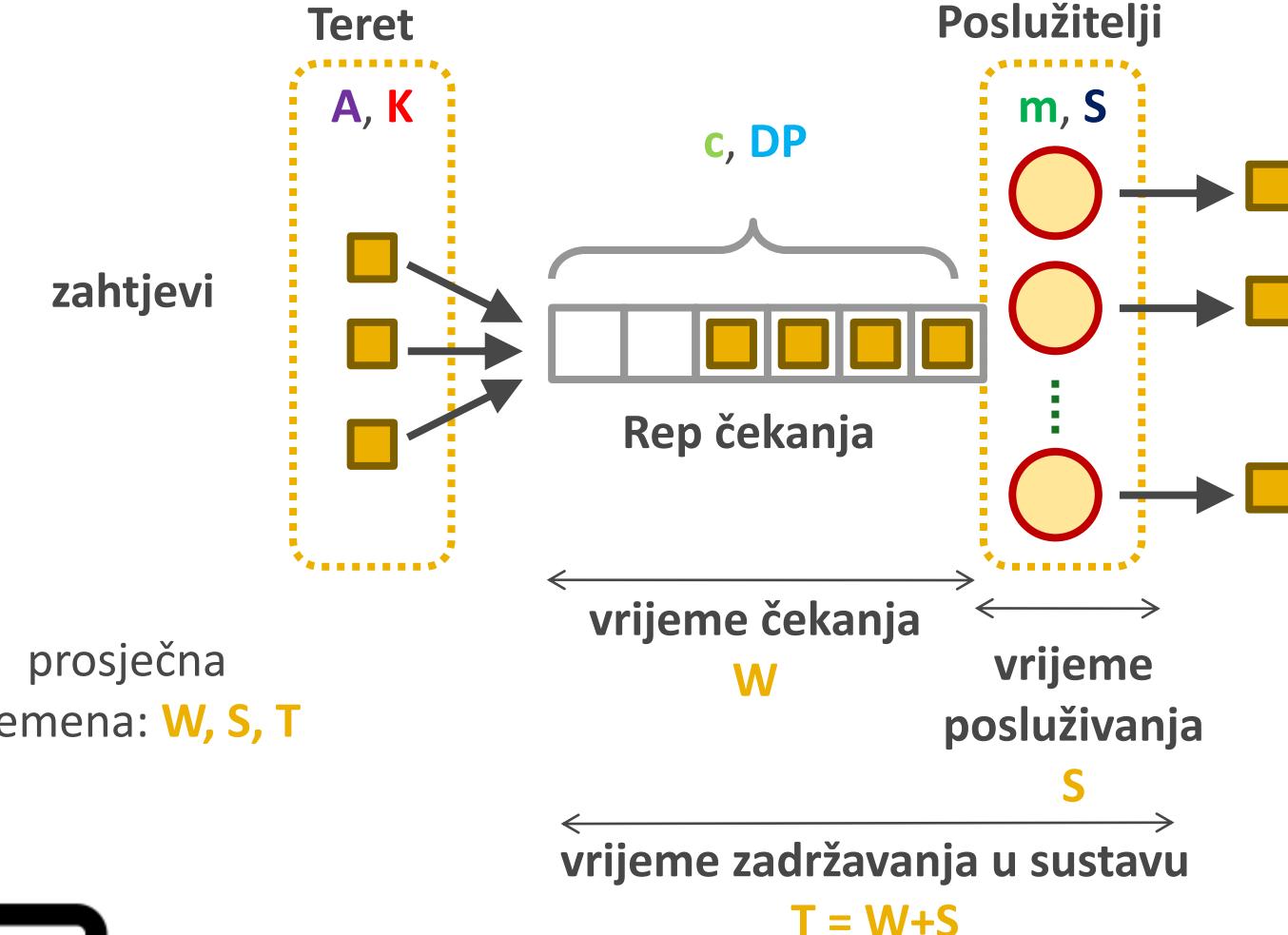
*Tekst licence preuzet je s <http://creativecommons.org/>*

# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - Primjer analize performansi web-aplikacije
- Domaća zadaća

# Ponovimo: osnovni pojmovi teorije repova

Proširena notacija:  $A/S/m/c/K/DP$



- A – razdioba međudolaznih vremena
- S – razdioba vremena posluživanja
- m – broj poslužitelja
- c – kapacitet sustava
- K – veličina ulazne populacije
- DP – disciplina posluživanja

# Utvrđivanje parametara modela

- Model crne kutije s dva ulazna parametra:
  - Teret sustava – karakteriziran prosječnim međudolaznim vremenom ( $1/\lambda$ ) i raspodjelom zahtjeva
  - Posluživanje – karakterizirano prosječnim vremenom posluživanja bez čekanja ( $S$ ) i raspodjelom vremena posluživanja
- Na temelju dva gornja parametra može se izračunati:
  - Prosječni broj zahtjeva u sustavu -  $N$
  - Prosječno vrijeme zadržavanja u sustavu (**odziv**) -  $T$
  - Prosječna iskoristivost sustava (**zaposlenost**) -  $\rho$

# Model jednopoloslužiteljskog sustava (1)

- Proširena notacija: A/S/m/c/K/DP
- Kendallova notacija: A/S/m npr. M/M/1



- M eksponencijalna raspodjela međudolaznih vremena zahtjeva (Poissonov proces)  
M eksponencijalna raspodjela vremena posluživanja  
1 jedan poslužitelj  
(c =  $\infty$  kapacitet sustava, K =  $\infty$  zahtjeva, DP = FIFO)

# Model jednopošlužiteljskog sustava (2)



Osnovne značajke modela:

$t$  – ukupno vrijeme promatranja rada sustava

$\alpha(t)$  – broj dolazaka zahtjeva u vremenu  $t$

$\beta(t)$  – broj odlazaka zahtjeva u vremenu  $t$

$\sigma(t)$  – vrijeme kroz koje je poslužitelj zaposlen u vremenu  $t$

# Model jednopošlužiteljskog sustava (3)



Izvedene veličine:

|                                               |                                                                              |
|-----------------------------------------------|------------------------------------------------------------------------------|
| $\lambda = \alpha(t) / t$                     | intenzitet dolazaka (z/s)                                                    |
| $\delta = \beta(t) / t$                       | intenzitet odlazaka [propusnost sustava] (z/s)                               |
| $S = \sigma(t) / \beta(t)$                    | prosječno vrijeme posluživanja (s/z)                                         |
| $\rho = \sigma(t) / t$<br>$= S \times \delta$ | prosječna iskoristivost [zaposlenost / zauzetost / opterećenje] poslužitelja |

$$\rho = (\sigma / t) \times (\beta / \beta) = (\sigma / \beta) \times (\beta / t) = S \times \delta$$

# Primjer 1: Posluživanje zahtjeva na disku

- Disk za trajno spremanje podataka ispunjava **50 zahtjeva u sekundi**. Prosječno vrijeme obrade zahtjeva operacija pisanja i čitanja je **10 ms**.
- Kolika je prosječna zaposlenost diska?
- Rješenje:
  - Propusnost sustava  **$\delta = 50 \text{ z/s}$**
  - Prosječno vrijeme obrade zahtjeva  **$S = 10 \text{ ms/z}$**
  - Prosječna zaposlenost diska  $\rho$ :  
$$\rho = \delta \times S = 50 \text{ z/s} \times 0,01 \text{ s/z} = 0,5 \text{ ( } 50 \% \text{ )}$$

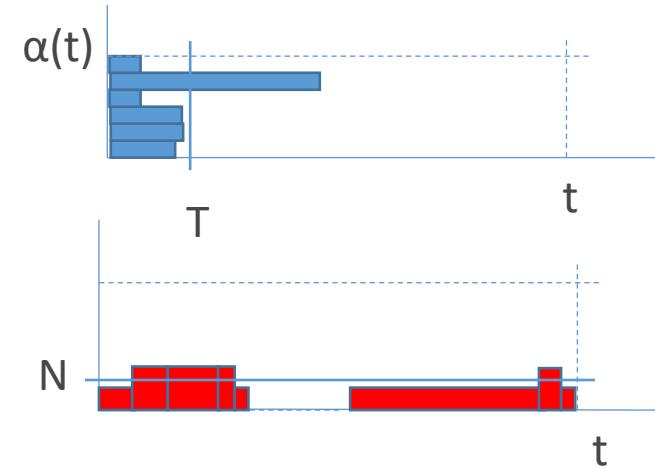
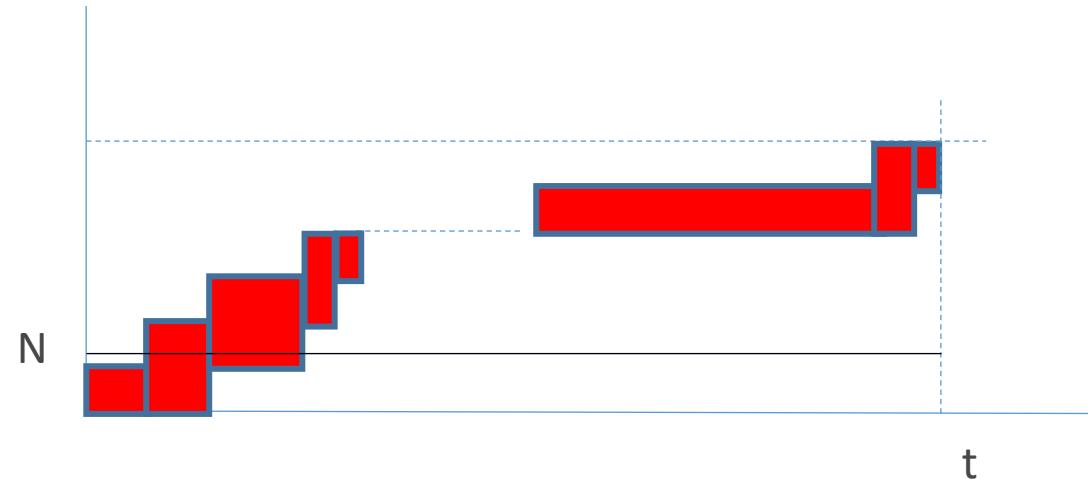
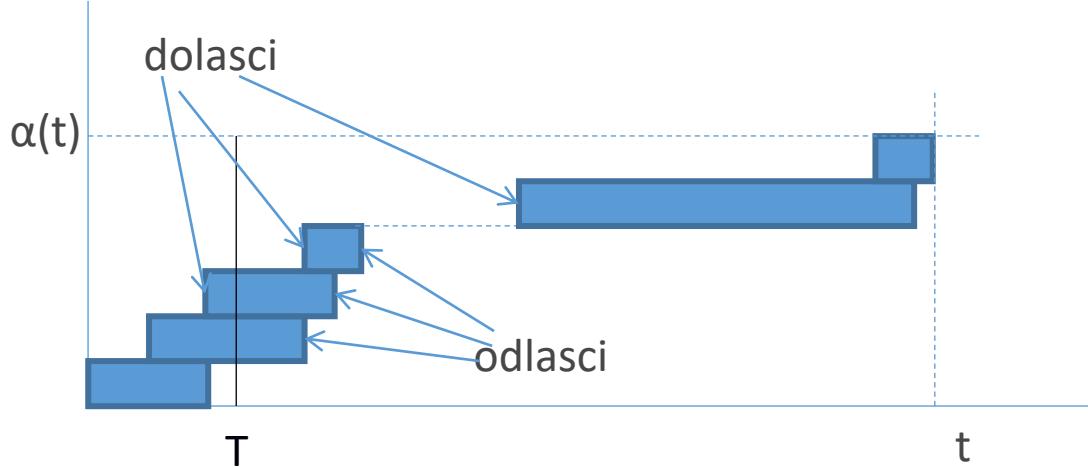
# Littleov zakon (John Little, 1961.)

- Napomena: sve veličine su prosječne vrijednosti!
- Prosječni broj zahtjeva u sustavu jednak je umnošku intenziteta dolazaka zahtjeva i prosječnog vremena zadržavanja zahtjeva u sustavu:
  - $\lambda$  [z/s] – Intenzitet dolazaka zahtjeva
  - $T$  [s] – Prosječno vrijeme zadržavanja zahtjeva u sustavu
  - $N$  [z] – Prosječni broj zahtjeva u sustavu
- Stabilnost sustava: broj prispjelih zahtjeva u vremenu jednak je broju zahtjeva koji napuštaju sustav ( $\lambda = \delta$ )

$$N = \lambda \times T$$

$$N = \delta \times T$$

# Dokaz Littleovog zakona



$$\text{Plava površina} = \alpha(t) \times T$$
$$\text{Crvena površina} = N \times t$$

$$\alpha(t) \times T = N \times t$$
$$\alpha(t)/t \times T = N$$
$$T \times \alpha(t)/t = N$$

$$T \times \lambda = N$$

# Jednopoloslužiteljski sustav

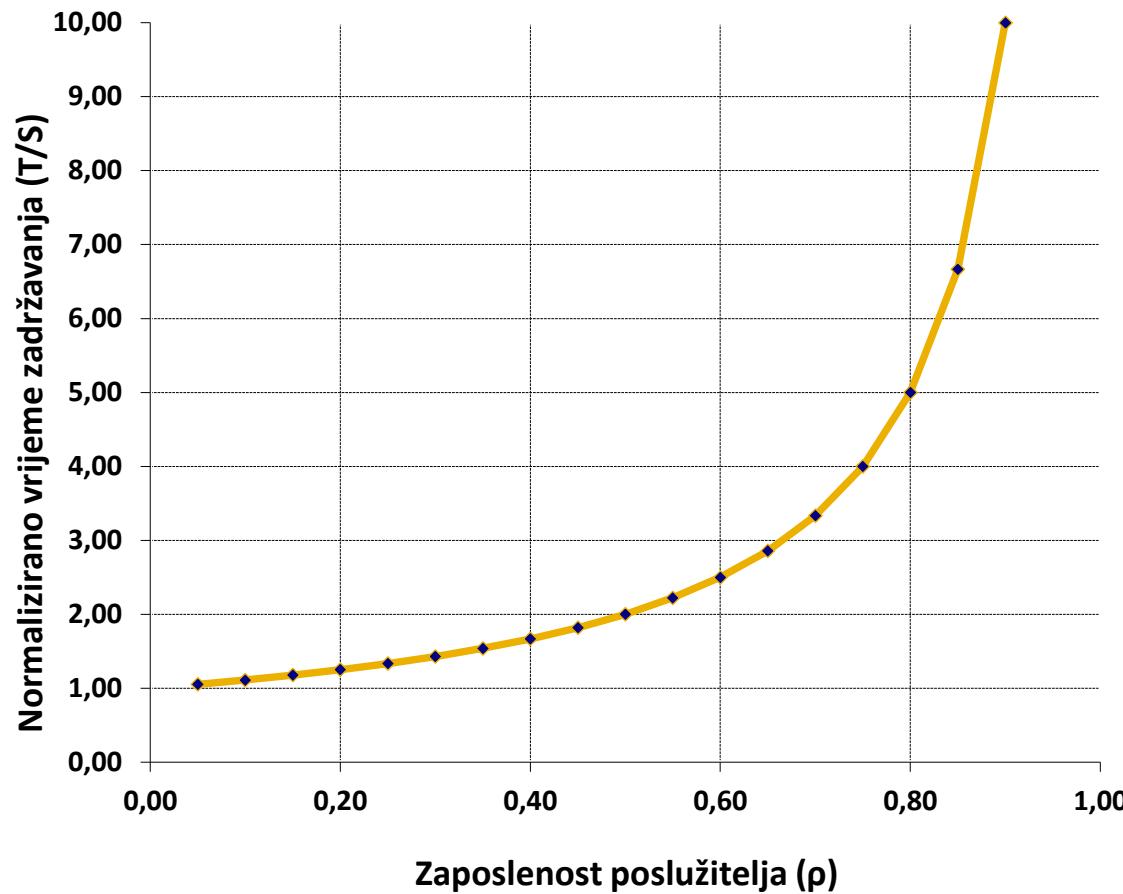
- Prosječno vrijeme zadržavanja zahtjeva u sustavu ( $T$ )
  - Vrijeme obrade svih neobrađenih zahtjeva u sustavu ( $N$ ) uvećano za vrijeme obrade novog zahtjeva  
$$T = S + W = S + S \times N$$
- U stabilnom stanju, primjenom supstitucije  $N = \delta \times T$  (Littleov zakon)  
$$T = S + S \times \delta \times T \rightarrow T = S / (1 - \delta \times S)$$
- Primjenom supstitucije  $\rho = \delta \times S$   
$$T = S / (1 - \rho)$$
- Množenje obje strane sa  $\delta \rightarrow [\delta \times T = (\delta \times S) / (1 - \rho)]$   
$$N = \rho / (1 - \rho)$$
 – prosječni broj zahtjeva u sustavu
- Množenje obje strane sa  $S \rightarrow [S \times N = (S \times \rho) / (1 - \rho)]$   
$$W = S \times N = (S \times \rho) / (1 - \rho)$$
 – prosječno vrijeme čekanja

# Primjer 2: Komunikacijski kanal

- Mjerenjem na pristupnoj točki mreže dobivamo prosječni protok od **125 paketa u sekundi** i prosječno vrijeme posluživanja paketa **0,002 sekunde**.
  - Što je sve moguće zaključiti o promatranom kanalu?
- Rješenje:
$$\delta = 125 \text{ p/s}, S = 0,002 \text{ s/p}$$
  - Prosječna zauzetost komunikacijskog sustava ( $\rho$ )
$$\rho = \delta \times S = 125 \text{ p/s} \times 0,002 \text{ s/p} = 0,25 \text{ (25 \%)} \quad$$
  - Prosječno vrijeme zadržavanja paketa u sustavu ( $T$ )
$$T = S / (1 - \rho) = (0,002 \text{ s/p}) / (1 - 0,25) = 0,0026666 \text{ s}$$
  - Prosječni broj paketa u sustavu ( $N$ )
$$N = \delta \times T = 125 \text{ p/s} \times 0,0026 \text{ s} = 0,333 \text{ p}$$

# Odziv sustava s repovima je izrazito nelinearan

Graf: normalizirano vrijeme zadržavanja (T/S)  
kao funkcija opterećenja (zaposlenosti) poslužitelja  $\rho$



vrijeme zadržavanja /  
vrijeme posluživanja  
 $T/S = 1/(1- \rho)$

# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - Primjer analize performansi web-aplikacije
- Domaća zadaća

# Serijski repovi i poslužitelji

- Littleov zakon

$$N = \lambda \times (T_1 + T_2 + T_3)$$

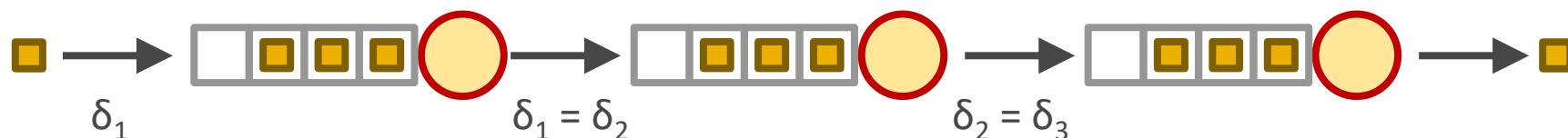
- U stabilnom stanju sustava ( $\lambda = \delta$ ,  $\delta_1 = \delta_2 = \delta_3 = \delta$ )

$$N = \delta \times T = \delta \times (T_1 + T_2 + T_3)$$

- Uz  $T_n = S_n / (1 - \delta \times S_n)$

$$N = \delta((S_1/(1 - \delta \times S_1)) + (S_2/(1 - \delta \times S_2)) + (S_3/(1 - \delta \times S_3)))$$

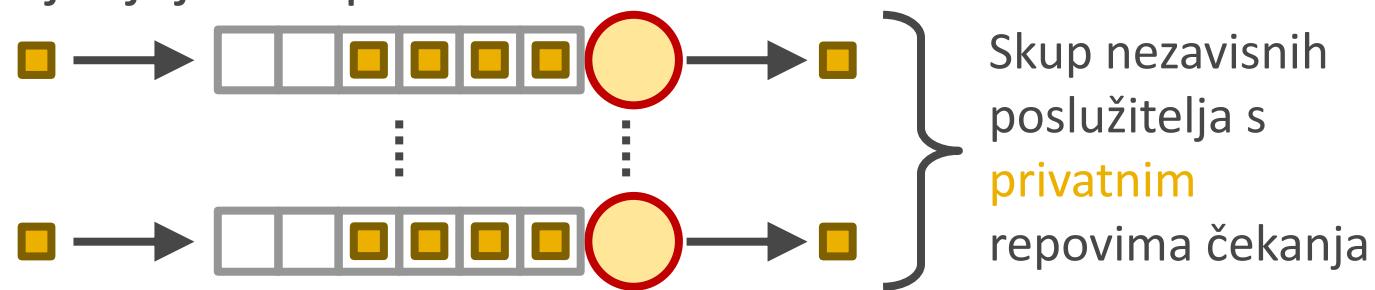
$$T = T_1 + T_2 + T_3$$



# Višestruki paralelni repovi i poslužitelji

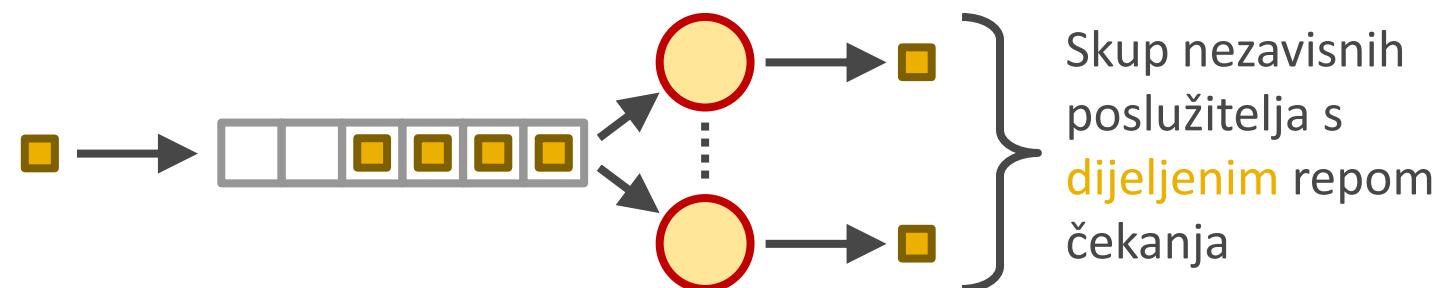
- Multiračunalo

- Model koji se primjenjuje u supermarketima



- Multiprocesor

- Model koji se primjenjuje u bankama



# Sustav s dva paralelna repa i poslužitelja

- Vrijeme zadržavanja u sustavu ( $T$ ):  
$$T = S + (S \times (0,5 \times N))$$
- Primjenom Littleovog zakona  $N = \delta \times T$   
$$T = S/(1 - 0,5 \times \delta \times S)$$
- Primjenom supstitucije  $\delta \times S = \rho$   
$$T = S/(1 - 0,5 \times \rho)$$
- Ukupna zaposlenost  $\rho$  sustava podijeljena s brojem poslužitelja  $m$  određuje faktor iskorištenja  $\rho'$  koji predstavlja vjerojatnost da je poslužitelj zaposlen  
$$\rho' = \rho/m, T = S/(1 - \rho')$$
- **Sustav s beskonačno mnogo repova i poslužitelja**  
$$m \rightarrow \infty ; \rho' \rightarrow 0 ; T \rightarrow S \leftarrow \text{Nema čekanja na posluživanje!}$$

Jedan poslužitelj:  
$$T = S/(1 - \rho)$$

# Sustav s jednim repom i m paralelnih poslužitelja

- Aproksimativno rješenje:

$$T = S + N \times (S/m) \times \rho'^{(m-1)}, \rho' = \rho/m$$

- Primjenom supstitucija  $N = \delta \times T$  i  $S = \rho/\delta$

$$T = S + (\delta \times T) \times (\rho / (\delta \times m)) \times \rho'^{(m-1)}$$

$$T = S + T \times (\rho/m) \times \rho'^{(m-1)}$$

$$T = S + T \times \rho'^m$$

$$T \times (1 - \rho'^m) = S$$

$$T = S / (1 - \rho'^m)$$

množenjem s  $\delta$

$$\delta \times T = \delta \times S / (1 - \rho'^m)$$

uz  $N = \delta \times T$  i  $\rho = \delta \times S$

$$N = \rho / (1 - \rho'^m)$$

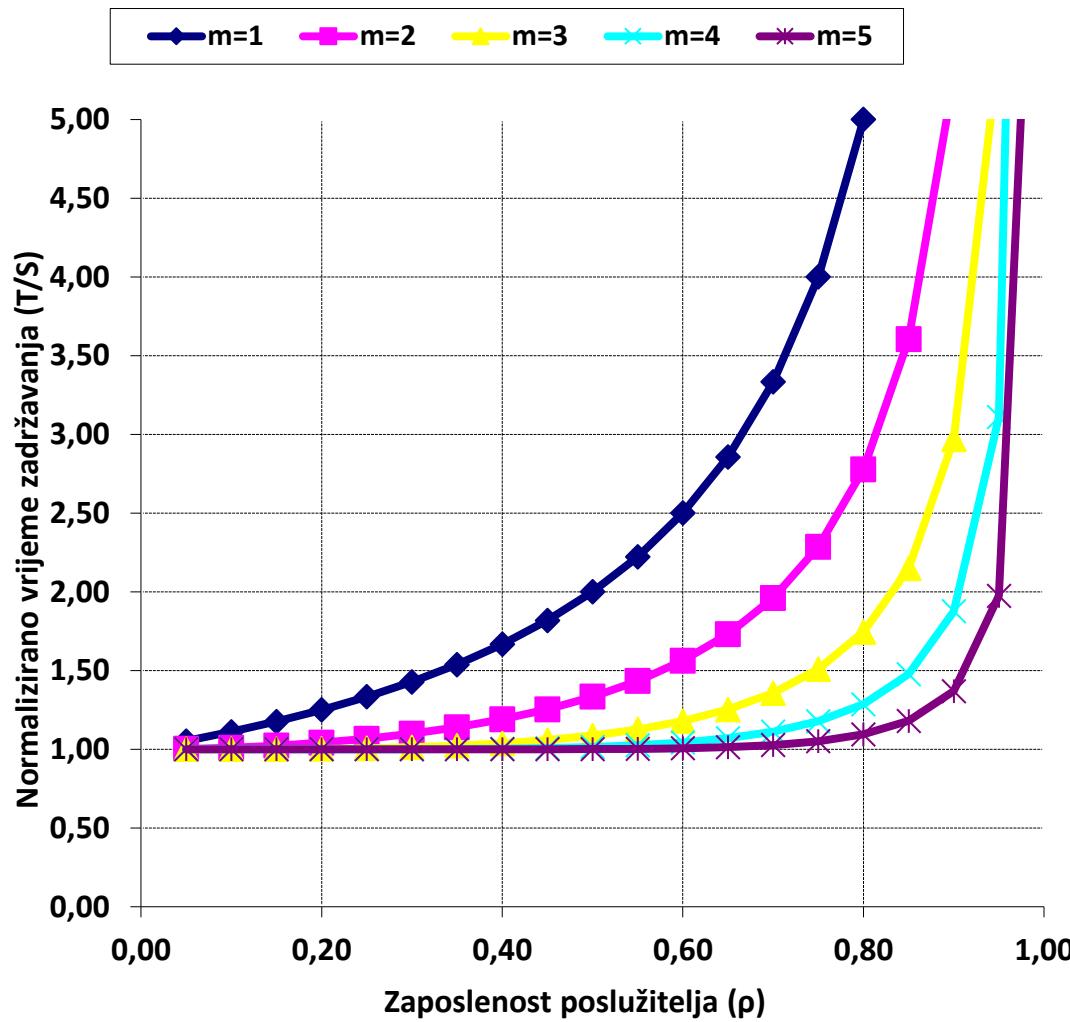
uz  $\rho = m \times \rho'$

$$N = (m \times \rho') / (1 - \rho'^m)$$

# Sustav s jednim repom i dva poslužitelja

- Vrijeme zadržavanja u sustavu ( $T$ ) ovisi o dva čimbenika
  - Broj poslužitelja: dodatni poslužitelj smanjuje vrijeme posluživanja za faktor 0,5
  - Vjerovatnost da je poslužitelj zaposlen:  $\rho' = \rho / 2$   
 $S(\rho') = (0,5 \times S) \times \rho'$   
 $T = S + N \times S(\rho') = S + 0,5 \times S \times \rho' \times N$
- Primjenom supstitucije  $N = \delta \times T$   
 $T = S + (0,5 \times S \times \rho' \times \delta \times T)$
- Primjenom supstitucije  $0,5 \times S \times \delta = 0,5 \times \rho = \rho'$   
 $T = S + T \times \rho'^2$   
 $T = S/(1 - \rho'^2) \rightarrow \text{množenjem sa } \delta \text{ i supstitucijom } S \times \delta = 2 \times \rho'$   
 $N = 2 \times \rho' / (1 - \rho'^2)$

# Usporedba sustava M/M/m - multiprocesor



# Egzaktno rješenje sustava s m paralelnih poslužitelja

- Erlangova formula
  - Analitičko rješenje za vrijeme zadržavanja T u sustavu s m paralelnih poslužitelja

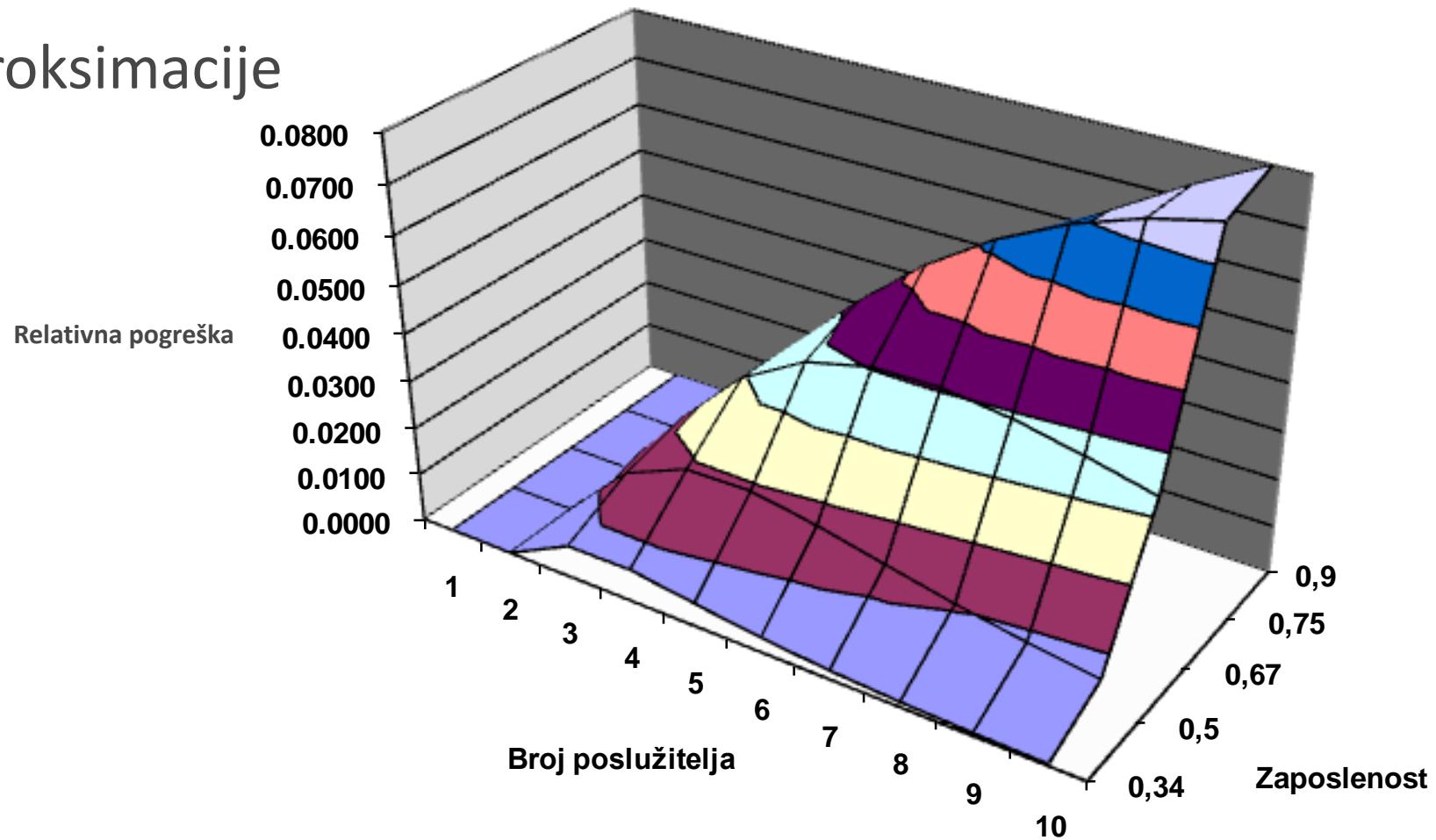
$$T = S * \left[ 1 + \frac{C(m, \rho')}{m * (1 - \rho')} \right]$$

- Koeficijent  $C(m, \rho')$

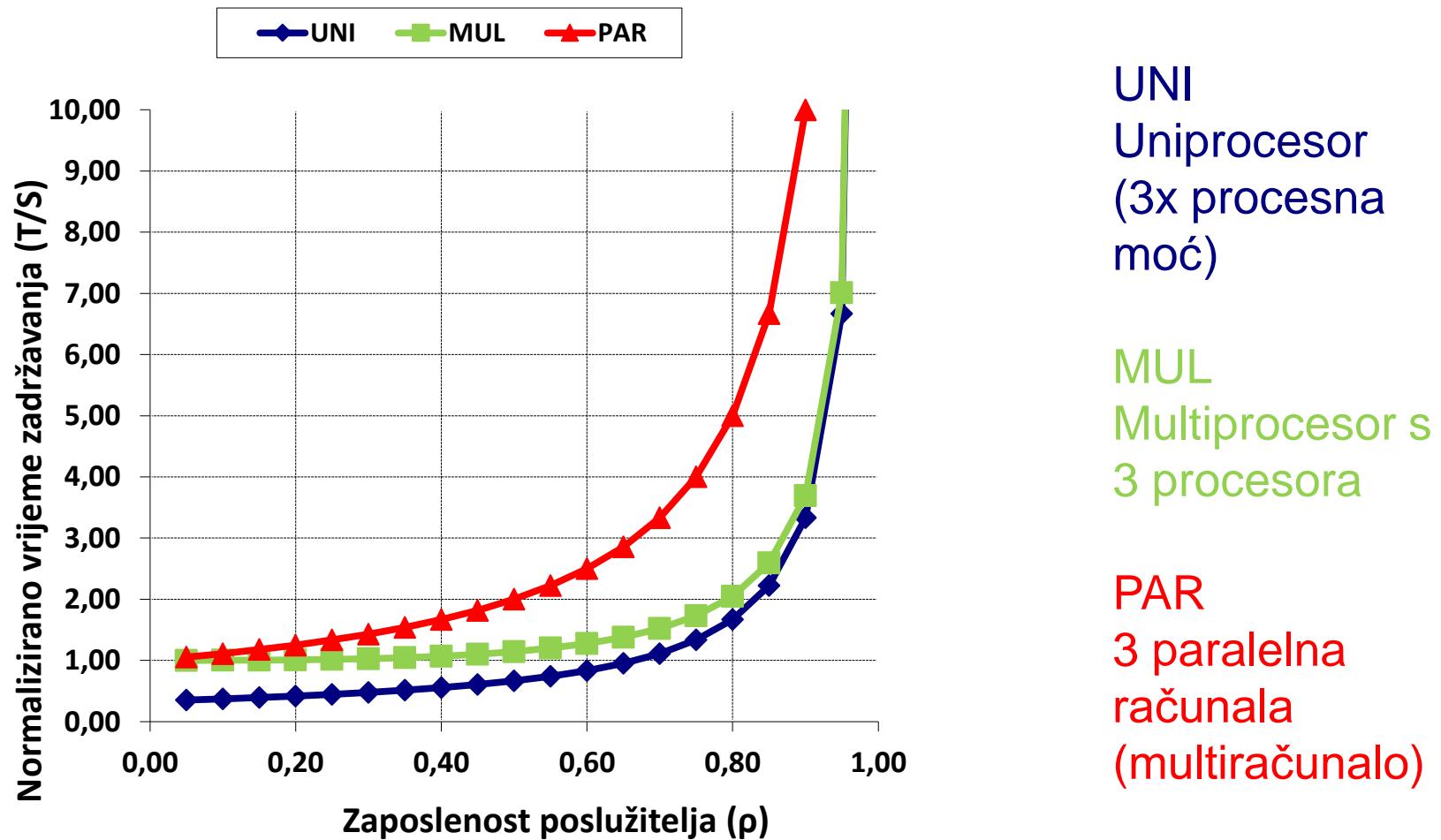
$$C(m, \rho') = \frac{\frac{(N * \rho')^m}{m!}}{(1 - \rho) * \sum_{k=0}^{m-1} \frac{(m * \rho')^k}{k!} + \frac{(m * \rho')^m}{m!}}$$

# Aproksimacija i egzaktno rješenje sustava

- Pogreška aproksimacije

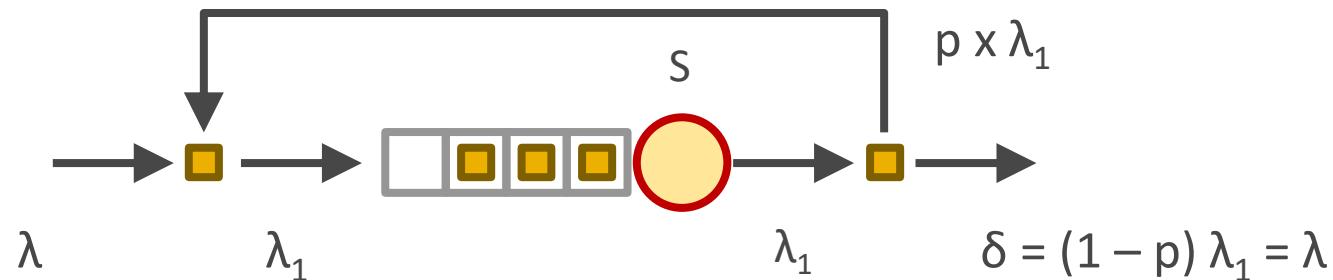


# Koji je model bolji: multiračunalo ili multiprocesor ?



# Poslužitelji s povratnom vezom

- Dio zahtjeva nakon posluživanja ponovno se vraća u rep za čekanje



$$\lambda_1 = \lambda + p \times \lambda_1 = \lambda / (1 - p) = \lambda \times v_1$$

$$\rho = \lambda_1 \times S$$

$$T_1 = S / (1 - \rho)$$

$$v_1 = 1 / (1 - p)$$

$$T = T_1 \times v_1 = T_1 / (1 - p)$$

iskoristivost sustava

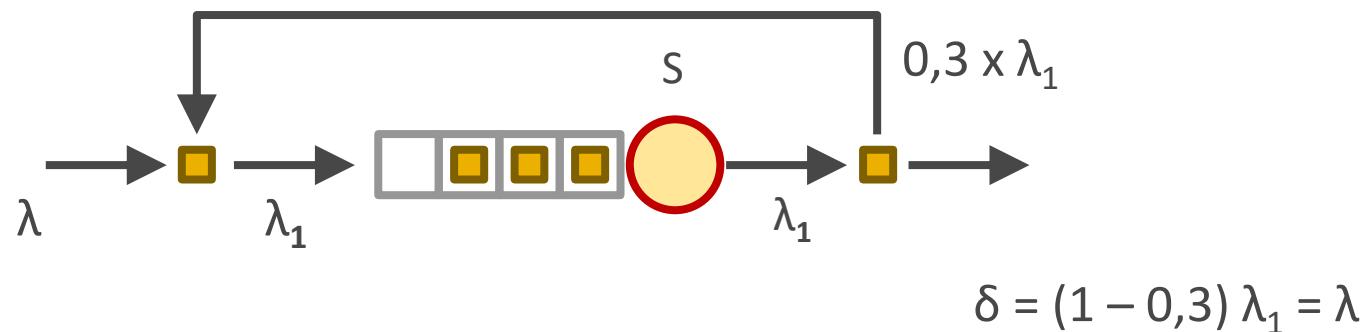
vrijeme zadržavanja za jedan prolaz

broj prolaza

vrijeme zadržavanja u sustavu

# Primjer 3: Komunikacijski kanal s pogreškom (1)

- Paketi dolaze u komunikacijski kanal s intenzitetom 0,5 paketa u sekundi i zahtijevaju 0,75 sekundi za obradu. Za 30 % paketa dogodi se pogreška pri prijenosu i takvi paketi se umeću u rep za ponovno slanje.
- Koliko vremena paket prosječno provede u kanalu?



# Primjer 3: Komunikacijski kanal s pogreškom (2)

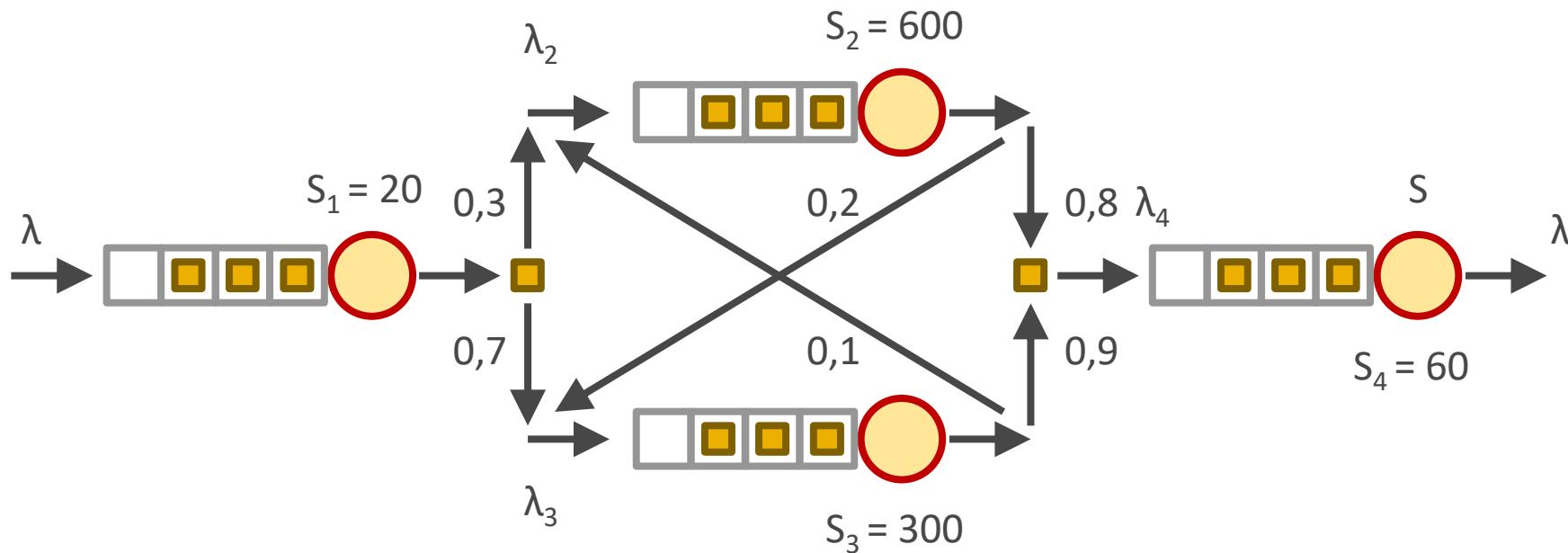
- Rješenje:
    - Broj pristiglih paketa u sekundi  $\lambda = 0,5 \text{ p/s}$
    - Prosječno vrijeme obrade paketa  $S = 0,75 \text{ s/p}$
    - Vjerojatnost pogreške paketa pri prijenosu  $p = 0,3$
- 

$$\lambda_1 = \lambda / (1 - p) = 0,5 / 0,7 = 0,714 \text{ p/s}$$

- Prosječna zauzetost kanala ( $\rho$ )
$$\rho = \lambda_1 \times S = 0,714 \text{ p/s} \times 0,75 \text{ s/p} = 0,536 \text{ (53.6 \%)}$$
- Prosječno vrijeme čekanja u repu ( $W$ )
$$W = S \times \rho / (1 - \rho) = 0,866 \text{ s/p}$$
- Prosječno vrijeme zadržavanja paketa u kanalu ( $T_1$ )
$$T_1 = W + S = 0,866 \text{ s/p} + 0,75 \text{ s/p} = 1,616 \text{ s/p}$$
- Prosječno vrijeme u kanalu:  $T = T_1 / (1-p) = 2.31$

# Višestruke povratne veze (1)

- Mreža repova poruka
  - Mrežna struktura proizvoljne složenosti s povratnim granama



- Izračunajte prosječno vrijeme zadržavanja zahtjeva u sustavu!

# Višestruke povratne veze (2)

- Što znamo na temelju prethodnih razmatranja?
- U stabilnom stanju sustava

$$\rho_n = \delta_n \times S_n \quad (\delta_n = \lambda_n \text{ za stabilni slučaj})$$

$$\rho_1 = \lambda \times S_1 = 20 \times \lambda$$

$$\rho_2 = 600 \times \lambda_2 = 600 (0,3 \times \lambda + 0,1 \times \lambda_3)$$

$$\rho_3 = 300 \times \lambda_3 = 300 (0,7 \times \lambda + 0,2 \times \lambda_2)$$

$$\rho_4 = 60 \times \lambda$$

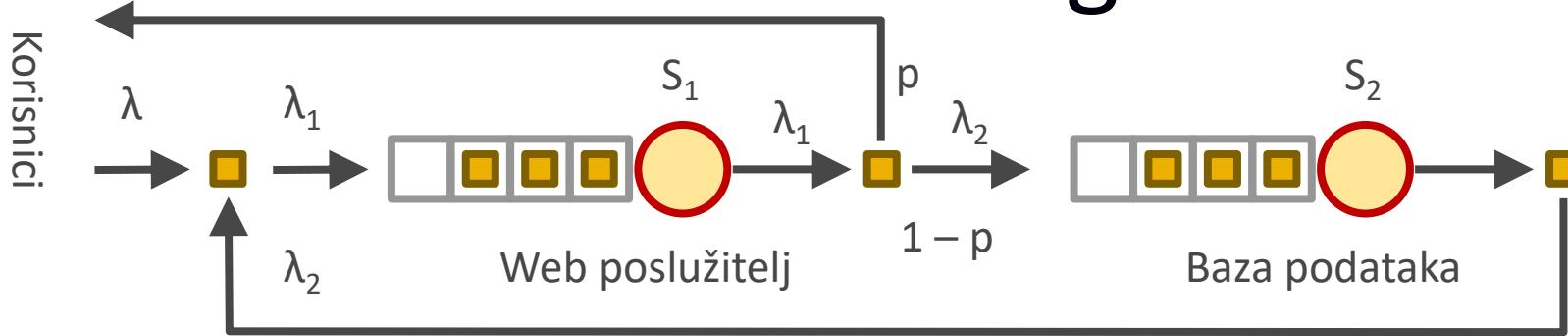
- Nakon rješenja za  $\lambda_2$  i  $\lambda_3$  i izračunavanja  $\rho_1 - \rho_4$ , izračunavamo  $N_1 - N_4$  iz:

$$N_n = \rho_n / (1 - \rho_n)$$

- Vrijeme zadržavanja zahtjeva u sustavu (T):

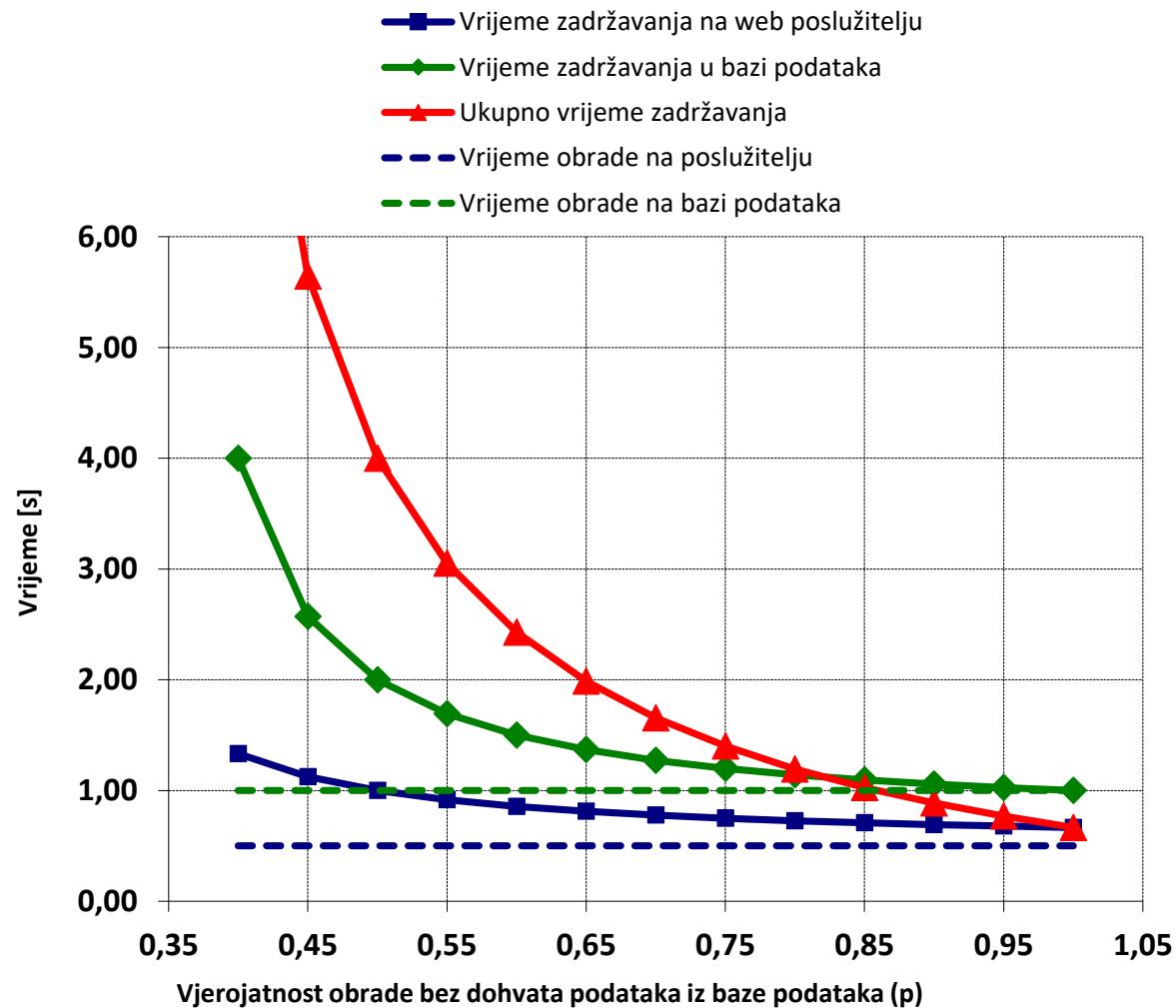
$$T = N / \lambda = (N_1 + N_2 + N_3 + N_4) / \lambda$$

# Jednostavan model web-usluge



- Intenzitet dolazaka zahtjeva ( $\lambda$ ) :
  - $\lambda_1 = \lambda + \lambda_2 = \lambda + (1-p)$   $\lambda_1 = \lambda/p$
  - $\lambda_2 = (1 - p)$   $\lambda_1 = ((1 - p)/p) \times \lambda$
- Vrijeme zadržavanja zahtjeva u sustavu (T):
  - $\delta_1 = \lambda_1$ ;  $\delta_2 = \lambda_2$
  - $\rho_1 = \delta_1 \times S_1 = \lambda \times S_1/p$ ;  $\rho_2 = ((1 - p)/p) \times \lambda \times S_2$
  - $T_1 = S_1/(1 - \rho_1)$ ;  $T_2 = S_2/(1 - \rho_2)$
  - $T = T_1 \times (1 + (1 - p)/p) + T_2 \times (1 - p)/p$

# Utjecaj parametara na ponašanje web-usluge



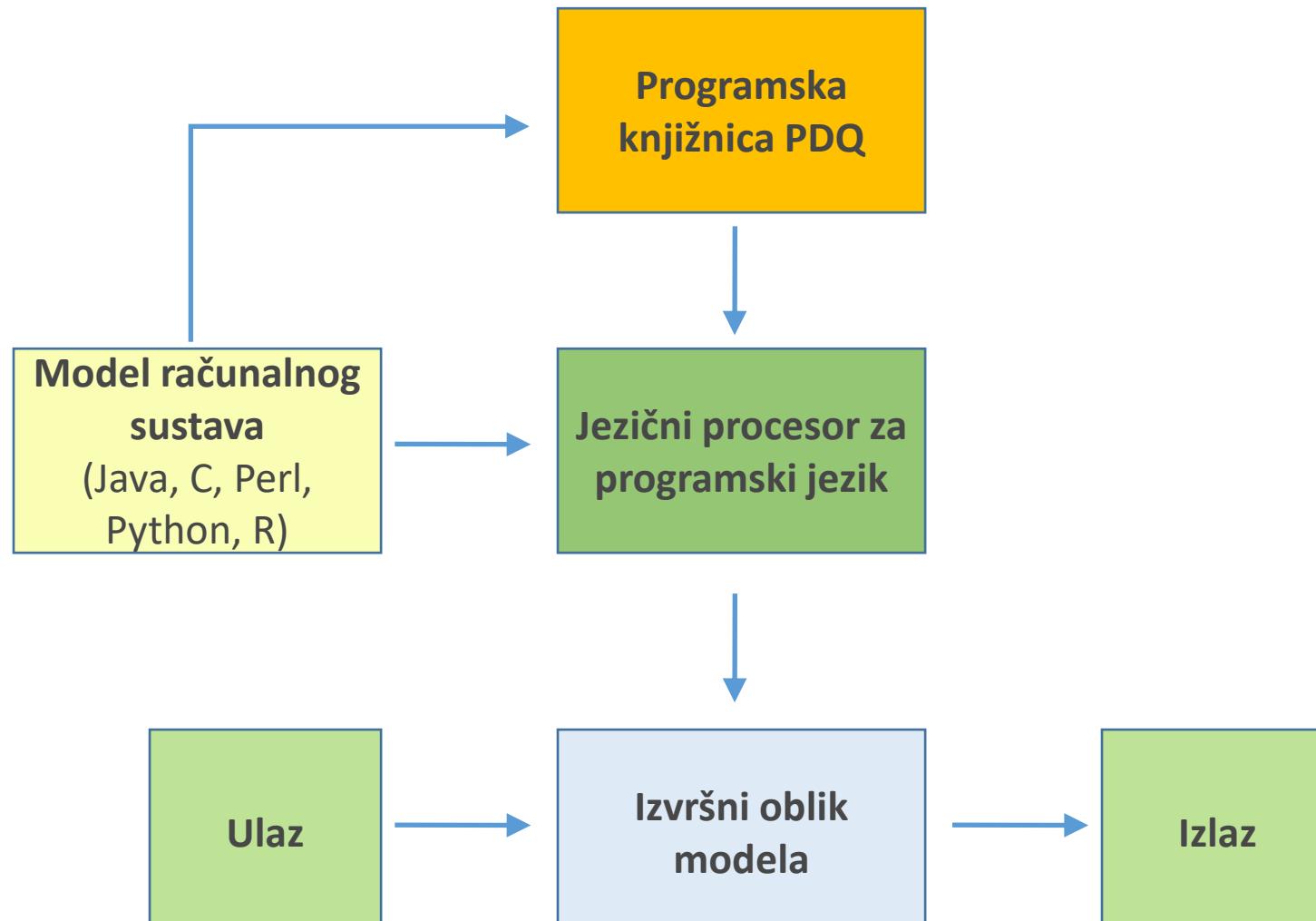
# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - Primjer analize performansi web-aplikacije
- Domaća zadaća

# Alat Pretty Damn Quick (PDQ)

- Omogućuje izgradnju modela za vrednovanje performansi računalnih sustava
- Modeli se grade primjenom načela teorije repova
- Značajke modela izračunavaju se primjenom analitičkih postupka i algoritama
- Dodatne informacije
  - PDQ: Pretty Damn Quick Performance Analyzer  
<http://www.perfdynamics.com/Tools/PDQ.html>
  - PDQ User Manual Online  
<http://www.perfdynamics.com/Tools/PDQman.html>
  - J. Gunther: Analyzing Computer System Performance With Perl, Springer, 2010.

# Alat Pretty Damn Quick (PDQ)

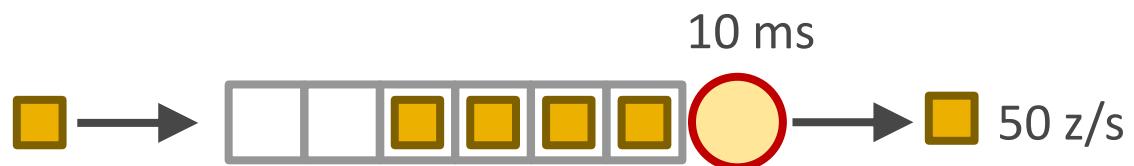


# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - Primjer analize performansi web-aplikacije
- Domaća zadaća

# Primjer 1: Posluživanje zahtjeva na disku (1)

- Disk za trajno spremanje podataka obrađuje 50 zahtjeva u sekundi. Prosječno vrijeme obrade zahtjeva operacija pisanja i čitanja je 10 ms.
- Kolika je prosječna zaposlenost diska?



# Primjer 1: Posluživanje zahtjeva na disku (2)

- Analitičko rješenje
    - Propusnost sustava
$$\delta = 50 \text{ z/s}$$
    - Prosječno vrijeme obrade zahtjeva
$$S = 10 \text{ ms/z}$$
  - Prosječna zaposlenost diska
$$\rho = \delta \times S = 50 \text{ z/s} \times 0,01 \text{ s/z} = 0,5 \text{ (50 \% )}$$
-

# Primjer 1: Posluživanje zahtjeva na disku (3)

```
main() {  
    extern int nodes, streams;  
  
    float L = 50;  
    float S = 0,01;  
  
    PDQ_Init("Diskovni podsustav");  
  
    nodes = PDQ_CreateNode("Posluzitelj", CEN, FCFS);  
    streams = PDQ_CreateOpen("Operacije", L);  
  
    PDQ_SetDemand("Posluzitelj", "Operacije", S);  
    PDQ_Solve(CANON);  
    PDQ_Report();  
}
```

# Primjer 2: Čekanje na posluživanje zahtjeva s diska (1)

- Disk iz prethodnog slučaja ima prosječno 1 zahtjev u repu
  - Koliko je prosječno vrijeme čekanja na obradu zahtjeva?



# Primjer 2: Čekanje na posluživanje zahtjeva s diska (2)

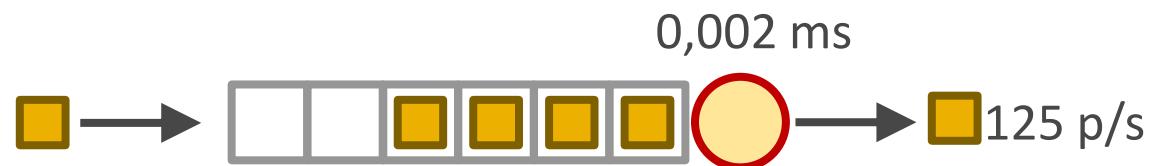
- Analitičko rješenje
  - Intenzitet zahtjeva  $\lambda = 50 \text{ z/s}$
  - Broj zahtjeva u repu  $N = 1 \text{ z}$
  - Vrijeme zadržavanja zahtjeva u sustavu
$$T = N/\lambda = (1 \text{ z}) / (50 \text{ z/s}) = 20 \text{ ms}$$
  - Vrijeme zadržavanja uključuje vrijeme čekanja u repu ( $W$ ) i vrijeme obrade zahtjeva ( $S$ ):
$$T = W + S$$
  - Vrijeme čekanja na obradu
  - $W = T - S = 20 \text{ ms} - 10 \text{ ms} = 10 \text{ ms}$

# Primjer 2: Čekanje na posluživanje zahtjeva s diska (3)

```
main() {  
    extern int nodes, streams;  
  
    float L = 50;  
    float S = 0,01;  
  
    PDQ_Init("Diskovni podsustav");  
  
    nodes = PDQ_CreateNode("Posluzitelj", CEN, FCFS);  
    streams = PDQ_CreateOpen("Operacije", L);  
  
    PDQ_SetDemand("Posluzitelj", "Operacije", S);  
  
    PDQ_Solve(CANON);  
    PDQ_Report();  
}
```

# Primjer 3: Komunikacijski kanal (1)

- Mjerenjem na pristupnoj točki mreže dobivamo prosječni protok od 125 paketa u sekundi i prosječno vrijeme posluživanja 0,002 sekunde.
  - Što je sve moguće zaključiti o promatranom kanalu?



# Primjer 3: Komunikacijski kanal (2)

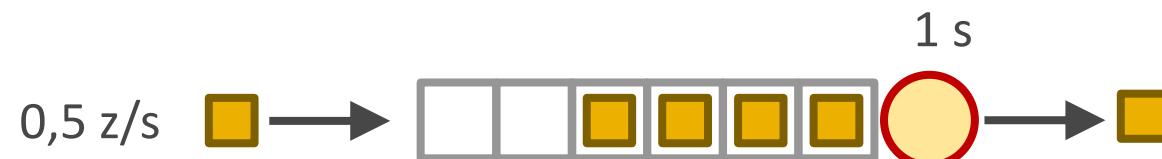
- Analitičko rješenje
  - Prosječni protok paketa  $\delta = 125 \text{ p/s}$
  - Prosječno vrijeme posluživanja paketa  $S = 0,002 \text{ s/p}$
  - Prosječna zaposlenost komunikacijskog sustava
$$\rho = \delta \times S = (125 \text{ p/s}) \times (0,002 \text{ s/p}) = 0,25 (25\%)$$
  - Prosječno vrijeme zadržavanja paketa u sustavu
$$T = S/(1 - \rho) = (0,002 \text{ s/p})/(1 - 0,25) = 0,0026666 \text{ s}$$
  - Prosječni broj paketa u repu
$$N = \delta \times T = 125 \text{ p/s} \times 0,0026 \text{ s} = 0,333 \text{ p}$$

# Primjer 3: Komunikacijski kanal (3)

```
main() {  
    extern int nodes, streams;  
  
    float L = 125;  
    float S = 0,002;  
  
    PDQ_Init("Mrezni podsustav");  
  
    nodes = PDQ_CreateNode("Posluzitelj", CEN, FCFS);  
    streams = PDQ_CreateOpen("Operacije", L);  
  
    PDQ_SetDemand("Posluzitelj", "Operacije", S);  
    PDQ_Solve(CANON);  
  
    PDQ_Report();  
}
```

# Primjer 4: Vrijeme čekanja i broj zahtjeva (1)

- Sustav ima prosječno vrijeme posluživanja 1 sekunda i intenzitet zahtjeva je 0,5 zahtjeva u sekundi.
- Kolika je prosječna vrijednost vremena zadržavanja i prosječna vrijednost broja zahtjeva u sustavu?



# Primjer 4: Vrijeme čekanja i broj zahtjeva (2)

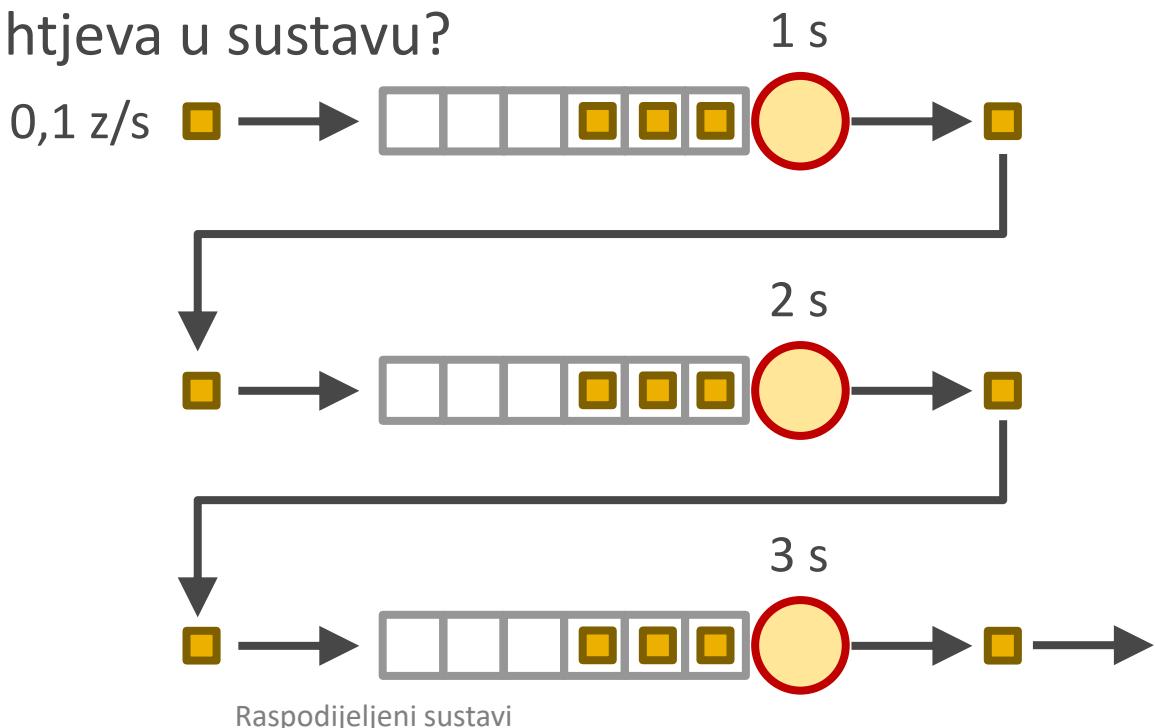
- Analitičko rješenje
  - Prosječno vrijeme posluživanja  $S = 1 \text{ s/z}$
  - Intenzitet zahtjeva  $\lambda = 0,5 \text{ z/s}$
  - Prosječna zaposlenost sustava
$$\rho = S \times \lambda = (1 \text{ s/z}) \times (0,5 \text{ z/s}) = 0,5 \text{ (50 \%)}$$
  - Prosječno vrijeme zadržavanja paketa u sustavu
$$T = S / (1 - \rho) = 1 / (1 - 0,5) = 2 \text{ s}$$
  - Prosječna vrijednost broja zahtjeva u sustavu
$$N = \rho / (1 - \rho) = 0,5 / (1 - 0,5) = 1 \text{ z}$$

# Primjer 4: Vrijeme čekanja i broj zahtjeva (3)

```
main() {  
    extern int nodes, streams;  
  
    float L = 0.5;  
    float S = 1.0;  
  
    PDQ_Init("Posuzitelj s repom");  
  
    nodes = PDQ_CreateNode("Posluzitelj", CEN, FCFS);  
    streams = PDQ_CreateOpen("Zadaci", L);  
  
    PDQ_SetDemand("Posluzitelj", "Zadaci", S);  
    PDQ_Solve(CANON);  
    PDQ_Report();  
}
```

# Primjer 5: Posluživanje u seriji (1)

- Sustav sadrži 3 serijske procesne jedinice s prosječnim vremenima posluživanja 1 s, 2 s i 3 s.
  - Koliko će biti vrijeme zadržavanja u sustavu uz intenzitet zahtjeva od 0,1 z/s?
  - Koliki će biti prosječni broj zahtjeva u sustavu?



# Primjer 5: Posluživanje u seriji (2)

- Analitičko rješenje
  - Prosječna vremena posluživanja  
 $S_1 = 1 \text{ s/z}, S_2 = 2 \text{ s/z}, S_3 = 3 \text{ s/z}$
  - Propusnost sustava  
 $\delta = 0,1 \text{ z/s}$
  - Vremena zadržavanja  $T_N = SN / (1 - \delta \times SN)$   
 $T_1 = 1.11 \text{ s}, T_2 = 2.5 \text{ s}, T_3 = 4.29 \text{ s}$
  - Prosječni broj zahtjeva u sustavu  
 $N = \delta \times (T_1 + T_2 + T_3) =$   
 $0,1 \times (1.11 + 2.5 + 4.29) = 0,79 \text{ z}$

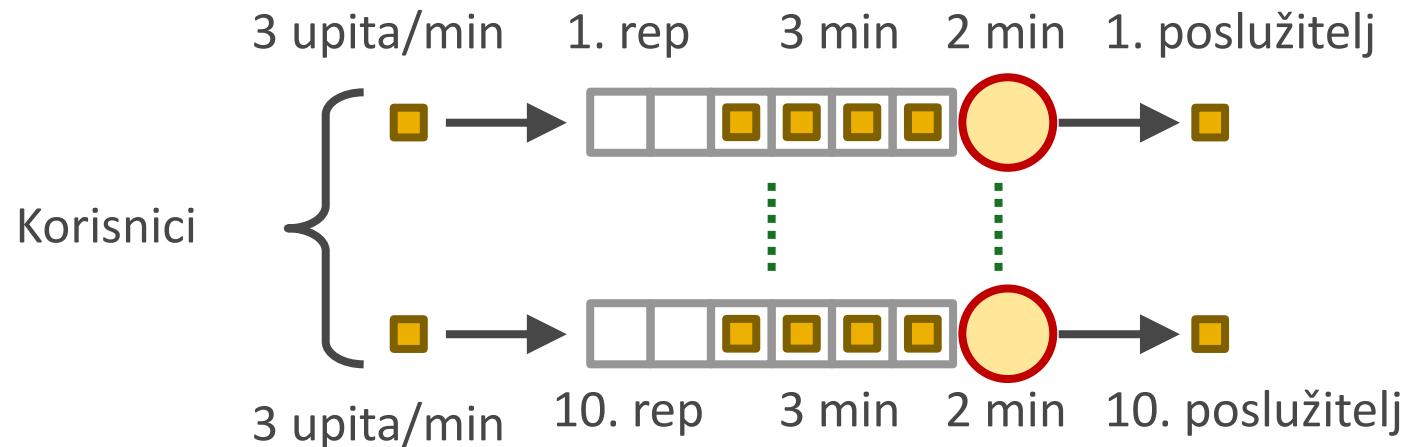
# Primjer 5: Posluživanje u seriji (3)

```
main() {  
    extern int nodes, streams;  
  
    float L = 0,1; float S1 = 1.0; float S2 = 2.0; float S3 = 3.0;  
  
    PDQ_Init("Serija tri posluzitelja");  
  
    streams = PDQ_CreateOpen("Zadaci", L);  
  
    nodes = PDQ_CreateNode("Posluzitelj1", CEN, FCFS);  
    nodes = PDQ_CreateNode("Posluzitelj2", CEN, FCFS);  
    nodes = PDQ_CreateNode("Posluzitelj3", CEN, FCFS);  
  
    PDQ_SetDemand("Posluzitelj1", "Zadaci", S1);  
    PDQ_SetDemand("Posluzitelj2", "Zadaci", S2);  
    PDQ_SetDemand("Posluzitelj3", "Zadaci", S3);  
  
    PDQ_Solve(CANON);  
  
    PDQ_Report();  
}
```

# Primjer 6: Aplikacija korisničke podrške (1)

- Web-aplikacija uključuje podršku korisnicima putem usluge chat. Kupci sami odabiru jedan od 10 repova čekanja. Mjerenja pokazuju da zahtjevi - upiti prosječno dolaze 3 u minuti te da svaki kupac prosječno čeka 3 minute u repu i prosječno provodi 2 minute u konverzaciji.
- Koliko bi dodatnih osoba – poslužitelja trebalo zaposliti da se prosječno vrijeme čekanja svede na 1 minutu?

# Primjer 6: Aplikacija korisničke podrške (2)

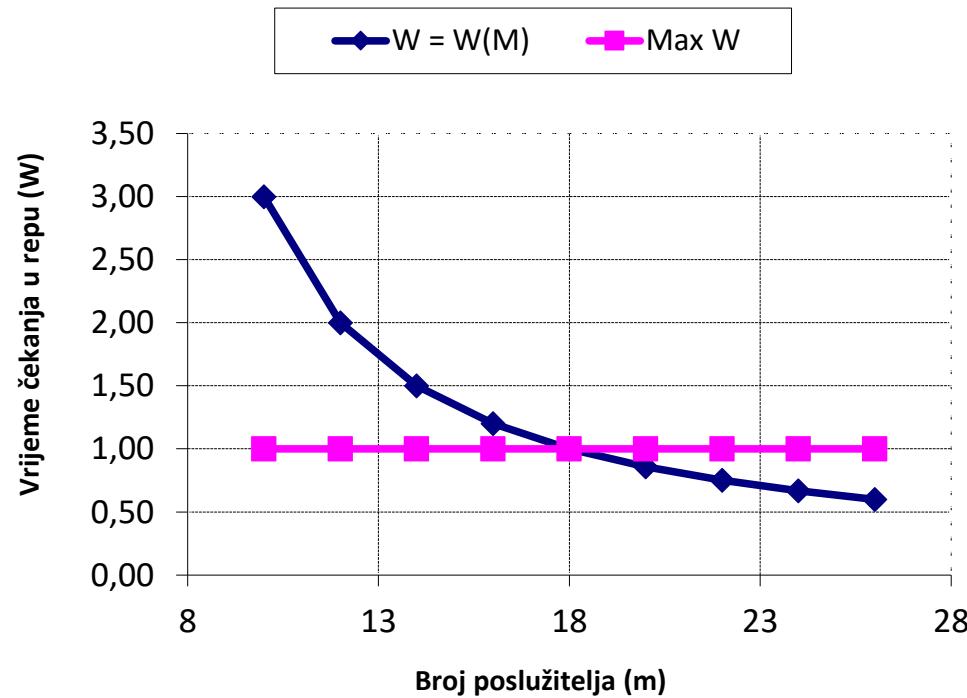


# Primjer 6: Aplikacija korisničke podrške (3)

- Analitičko rješenje
  - Prosječno vrijeme posluživanja  $S = 2 \text{ min/z}$
  - Broj pristiglih zahtjeva u jednom repu  $\lambda = 3 \text{ z/min}$
  - Prosječna zaposlenost sustava
$$\rho = S \lambda = ( 2 \text{ min/z} ) ( 3 \text{ z/min} ) = 6$$
  - Faktor iskorištenja
$$\rho' = \rho/m = 6/10 = 0,6$$
  - Prosječno vrijeme zadržavanja korisnika u sustavu
$$T = S / (1 - \rho') = 2 / (1 - 0,6) = 5 \text{ min}$$
  - Prosječno vrijeme čekanja u repu
$$W = T - S = 5 \text{ min} - 2 \text{ min} = 3 \text{ min}$$

# Primjer 6: Aplikacija korisničke podrške (4)

- Broj poslužitelja
  - Za zadani sustav ne postoji analitičko rješenje. Rješenje se određuje primjenom numeričkih metoda ili primjenom metode pokušaja i promašaja.



- Rješenje: potrebno 18 poslužitelja

# Primjer 6: Aplikacija korisničke podrške (5)

- Odabранo rješenje
  - Broj poslužitelja (tehničara)  $m = 18$
  - Prosječno vrijeme posluživanja  $S = 2 \text{ min/z}$
  - Propusnost sustava  $\delta = 3 \text{ z/min}$
  - Prosječna zaposlenost sustava
  - $\rho = \delta * S = (3 \text{ z/min}) * (2 \text{ min/z}) = 6$
  - Faktor iskorištenja  $\rho' = \rho/m = 6/18 = 1/3$
  - Prosječno vrijeme zadržavanja korisnika u sustavu
  - $T = S / (1 - \rho') = 2 / (1 - 1/3) = 3 \text{ min}$
  - Prosječno vrijeme čekanja u repu ( $W$ )
  - $W = T - S = 3 - 2 = 1 \text{ min}$

# Primjer 6: Aplikacija korisničke podrške (6)

```
main() {  
    extern int    nodes;  
    extern int    streams;  
    double        L = 3;  
    double        S = 2;  
    char nName[30];  
    char cName[30];  
    int   i;  
    int   count = 10;  
  
    PDQ_Init("Aplikacija korisnicke podrske");  
  
    ...
```

# Primjer 6: Aplikacija korisničke podrške (7)

...

```
for( i=0; i<count; i++ ) {
    sprintf(nName, "Serv %2d", i);
    sprintf(cName, "Clnt %2d", i);

    nodes = PDQ_CreateNode(nName, CEN, FCFS);
    streams = PDQ_CreateOpen(cName, L/count);
}

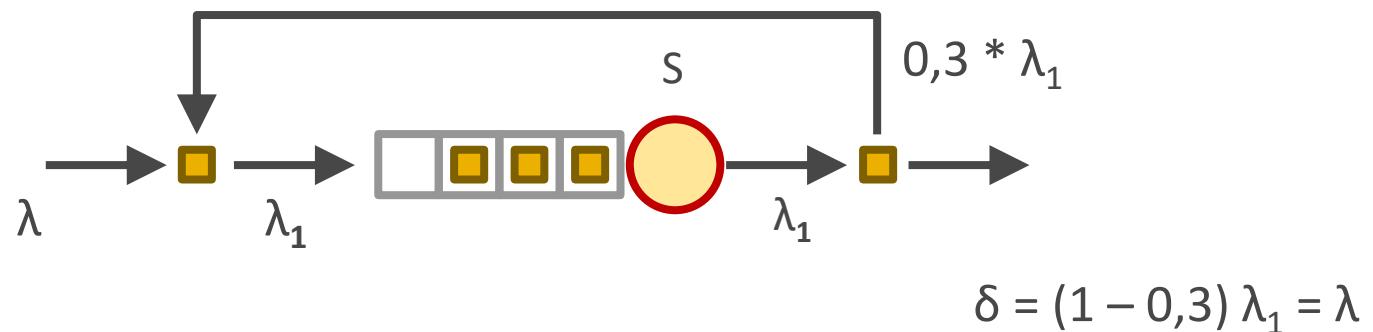
for( i=0; i<count; i++ ) {
    sprintf(nName, "Serv %2d", i);
    sprintf(cName, "Clnt %2d", i);
    PDQ_SetDemand(nName, cName, S);
}

PDQ_Solve(CANON);
PDQ_Report();
}
```

# Primjer 7: Komunikacijski kanal s pogreškom (1)

- Paketi dolaze u komunikacijski kanal s intenzitetom 0,5 paketa u sekundi i zahtijevaju 0,75 sekundi za obradu. Za 30 % paketa dogodi se pogreška pri prijenosu i takvi paketi se umeću u rep za ponovno slanje.
  - Koliko vremena paket prosječno provede u kanalu ?

# Primjer 7: Komunikacijski kanal s pogreškom (2)



# Primjer 7: Komunikacijski kanal s pogreškom (3)

- Analitičko rješenje
  - Intenzitet paketa  $\lambda = 0,5 \text{ p/s}$
  - Prosječno vrijeme obrade paketa  $S = 0,75 \text{ s/p}$
  - Vjerojatnost pogreške paketa pri prijenosu  $p = 0,3$   
$$\lambda_1 = \lambda / (1 - p) = 0,5 / 0,7 = 0,714 \text{ p/s}$$
  - Prosječna zaposlenost kanala  
$$\rho = \lambda_1 \times S = 0,714 \text{ p/s} \times 0,75 \text{ s/p} = 0,536 \text{ (53.6 %)}$$
  - Prosječno vrijeme čekanja paketa u repu  
$$W = S \times \rho / (1 - \rho) = 0,866 \text{ s/p}$$
  - Prosječno vrijeme zadržavanja paketa (1 prolaz)  
$$T_1 = W + S = 0,866 \text{ s/p} + 0,75 \text{ s/p} = 1,616 \text{ s/p}$$
  - Prosječno vrijeme zadržavanja u kanalu:  $T = T_1 / (1-p) = 2,31 \text{ s}$

# Primjer 7: Komunikacijski kanal s pogreškom (4)

```
main() {  
    extern int      nodes, streams;  
    float   p_err = 0,30;  
    float   L     = 0,50;  
    float   S     = 0,75;  
    float   V     = 1.0 / ( 1.0 - p_err );  
  
    PDQ_Init("Posluzitelj s repom i povratnom vezom");  
  
    nodes = PDQ_CreateNode("Kanal", CEN, FCFS);  
    streams = PDQ_CreateOpen("Poruka", L);  
  
    PDQ_SetVisits("Kanal", "Poruka", V, S);  
    PDQ_Solve(CANON);  
    PDQ_Report();  
}
```

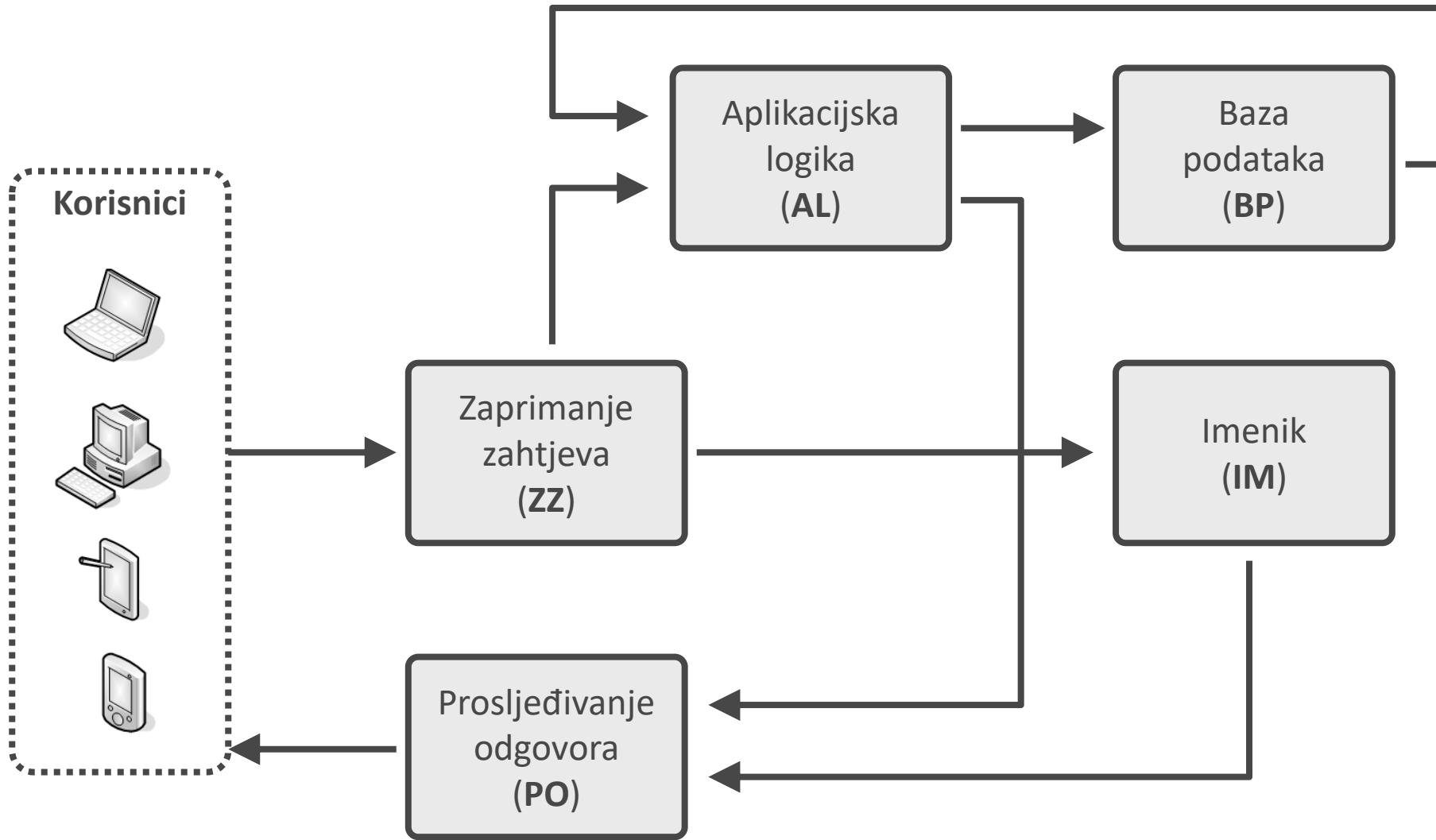
# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - **Primjer analize performansi web-aplikacije**
- Domaća zadaća

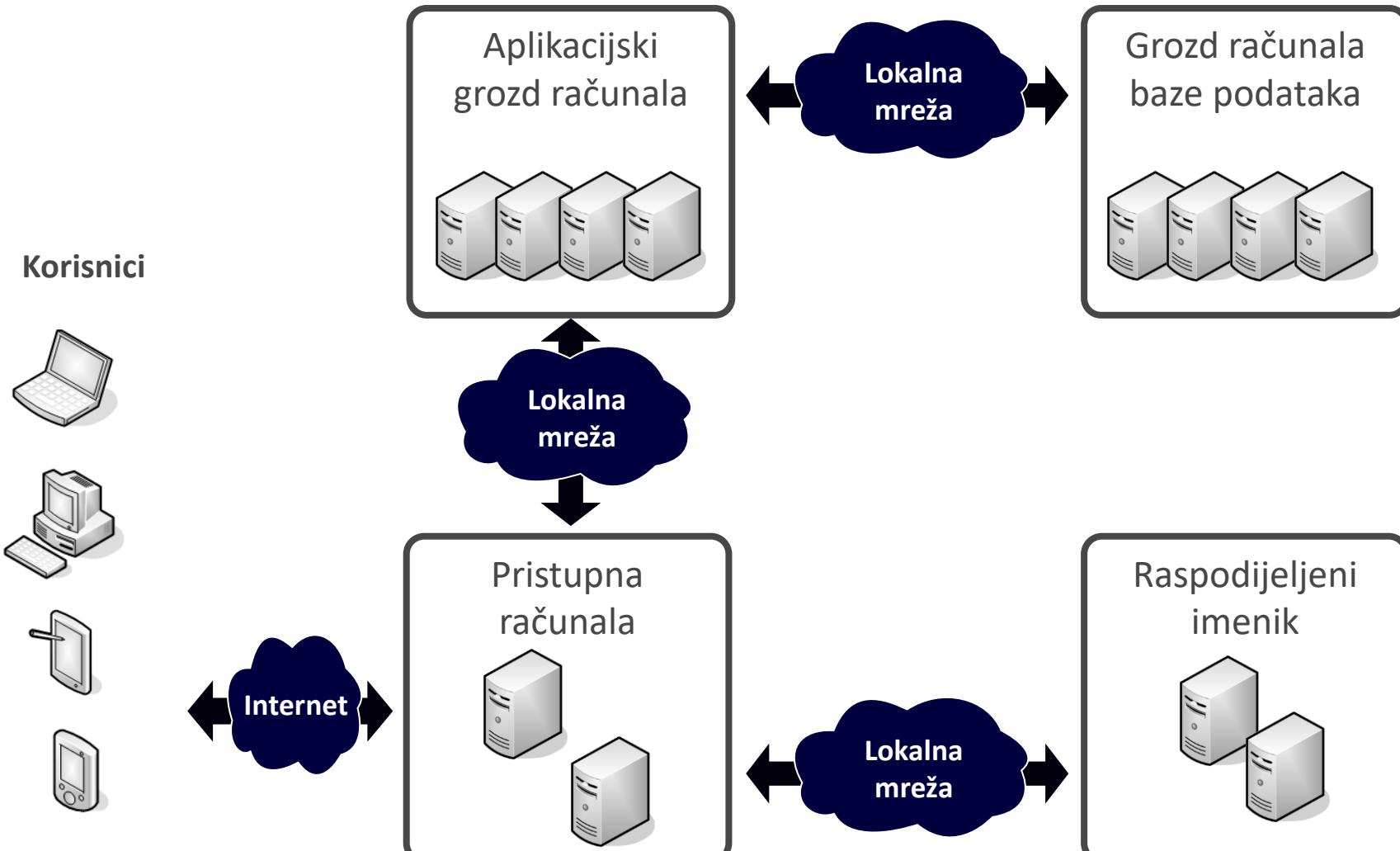
# Primjer analize raspodijeljene aplikacije

- Logička arhitektura raspodijeljene aplikacije
- Fizička arhitektura raspodijeljene aplikacije
- Model raspodijeljene aplikacije
- Vrednovanje značajki performansi aplikacije

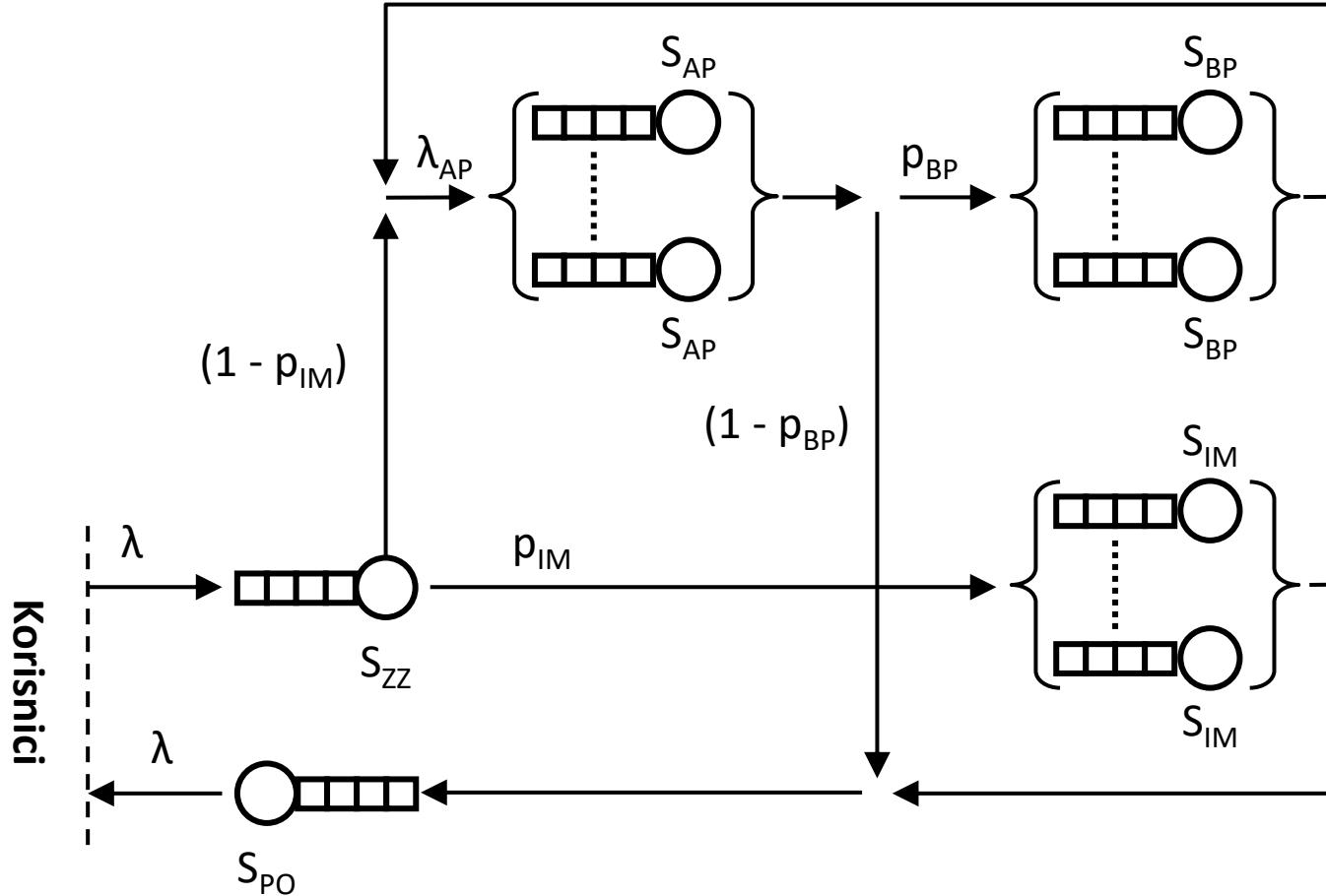
# Logička arhitektura aplikacije



# Fizička arhitektura aplikacije



# Model aplikacije



# Primjer analize raspodijeljene aplikacije (1)

- Intenzitet dolazaka zahtjeva na podsustave IM, AP, BP, ZZ i PO:

$$\lambda_{IM} = p_{IM}\lambda, v_{IM} = p_{IM}$$

$$\lambda_{AP} = p_{BP}\lambda_{AP} + (1 - p_{IM})\lambda \Rightarrow$$

$$\lambda_{AP} = [(1 - p_{IM})/(1 - p_{BP})] \lambda$$

$$v_{AP} = [(1 - p_{IM})/(1 - p_{BP})]$$

$$\lambda_{BP} = p_{BP} \lambda_{AP} \Rightarrow$$

$$\lambda_{BP} = p_{BP} [(1 - p_{IM})/(1 - p_{BP})] \lambda$$

$$v_{BP} = p_{BP} [(1 - p_{IM})/(1 - p_{BP})]$$

$$\lambda_{ZZ} = \lambda, v_{ZZ} = 1$$

$$\lambda_{PO} = \lambda, v_{PO} = 1$$

# Primjer analize raspodijeljene aplikacije (2)

- Skalirano vrijeme posluživanja:

$$D_{IM} = v_{IM} S_{IM} = p_{IM} S_{IM}$$

$$D_{AP} = v_{AP} S_{AP} = [(1 - p_{IM}) / (1 - p_{BP})] S_{AP}$$

$$D_{BP} = v_{BP} S_{BP} = p_{BP} [(1 - p_{IM}) / (1 - p_{BP})] S_{BP}$$

$$D_{ZZ} = v_{ZZ} S_{ZZ} = 1 S_{ZZ}$$

$$D_{PO} = v_{PO} S_{PO} = 1 S_{PO}$$

- Vrijeme zadržavanja zahtjeva:

$$T = v_{IM} T_{IM} + v_{AP} T_{AP} + v_{BP} T_{BP} + v_{ZZ} T_{ZZ} + v_{PO} T_{PO} \Rightarrow$$

$$T = D_{IM} / (1 - \lambda D_{IM}) + D_{AP} / (1 - \lambda D_{AP}) + D_{BP} / (1 - \lambda D_{BP}) + \\ D_{ZZ} / (1 - \lambda D_{ZZ}) + D_{PO} / (1 - \lambda D_{PO})$$

$$\text{gdje je } v_x T_x = v_x S_x / (1 - p_x) = v_x S_x / (1 - v_x \lambda S_x) = D_x / (1 - \lambda D_x)$$

# Vrijeme zadržavanja zahtjeva



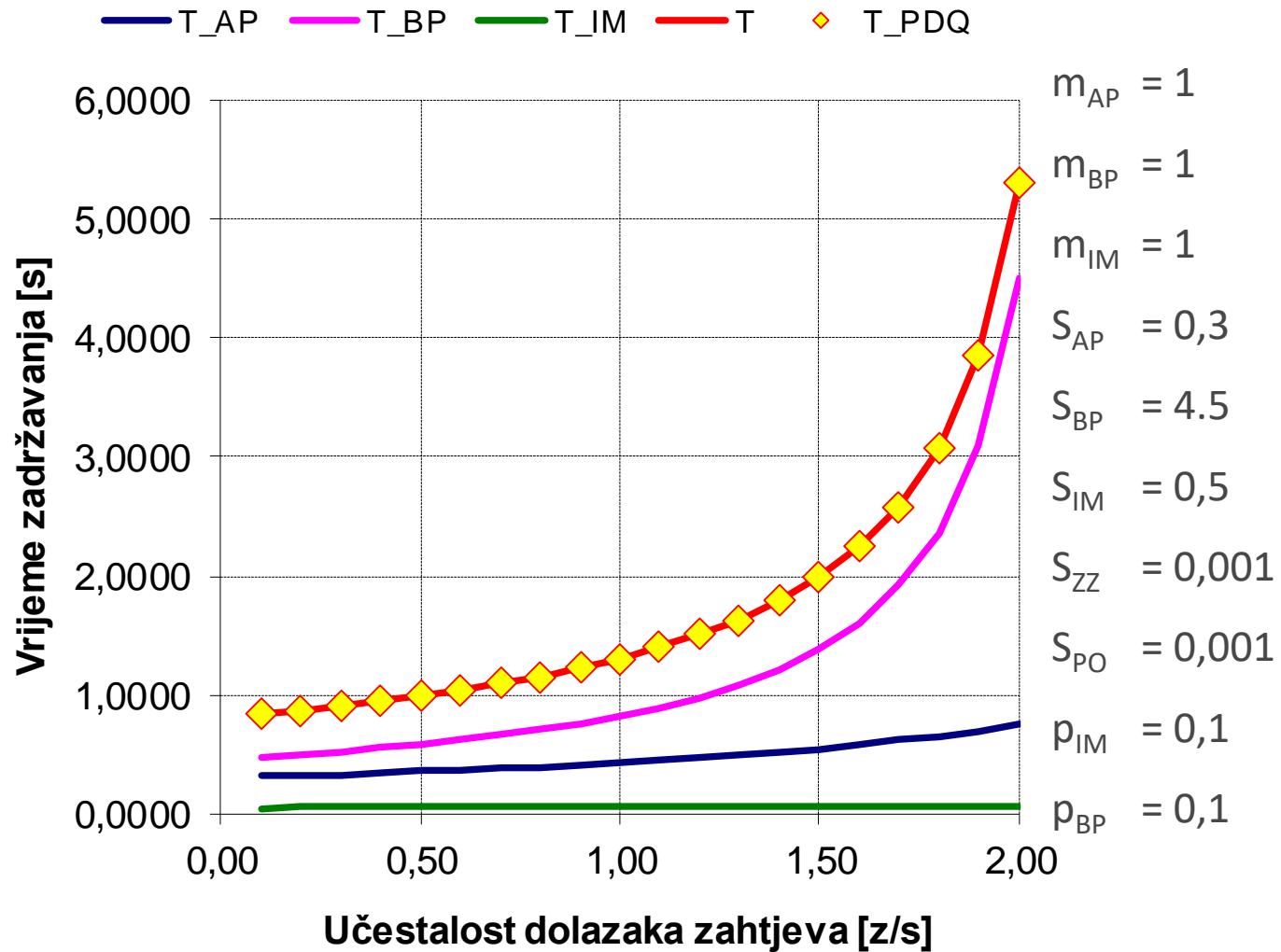
Response.  
java



ap-Zad.c



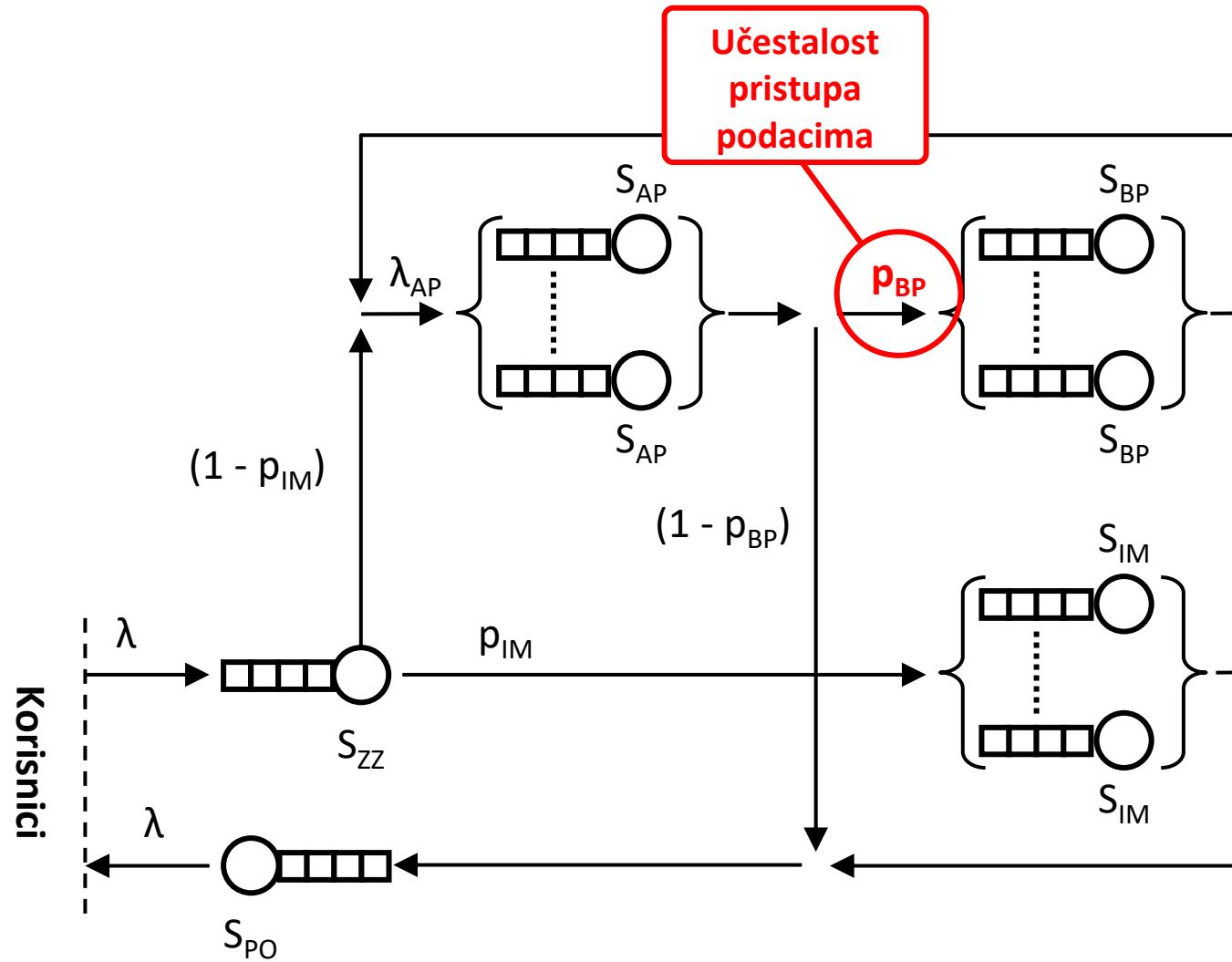
Ap\_Zad.xls



# Vrednovanje značajki performansi

- Učestalosti pristupa podacima
- Veličina grozda baze podataka
- Promjena organizacije podataka
- Promjena stupnja sigurnosti

# Učestalost pristupa podacima



# Vrijeme zadržavanja zahtjeva



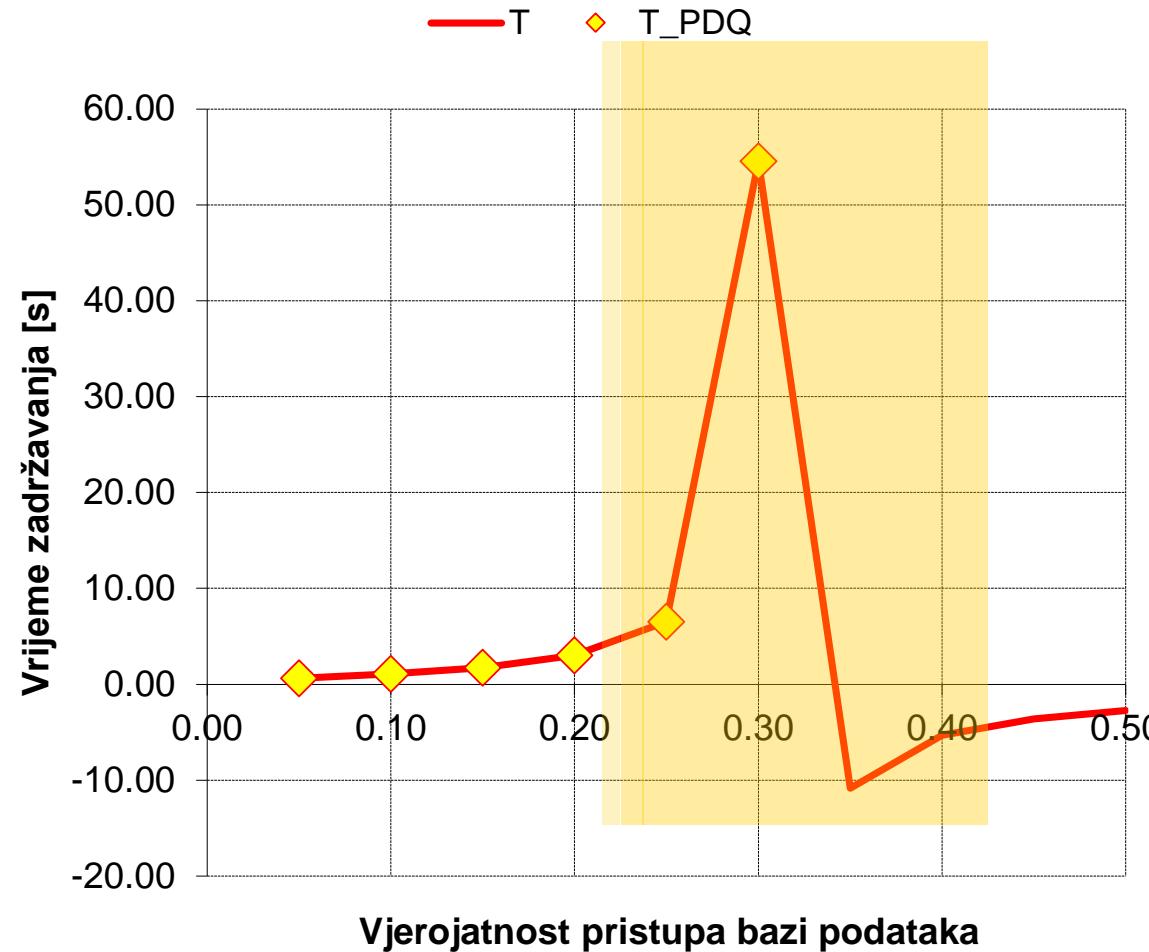
DatabaseAccess.java



ap-Pristup.c

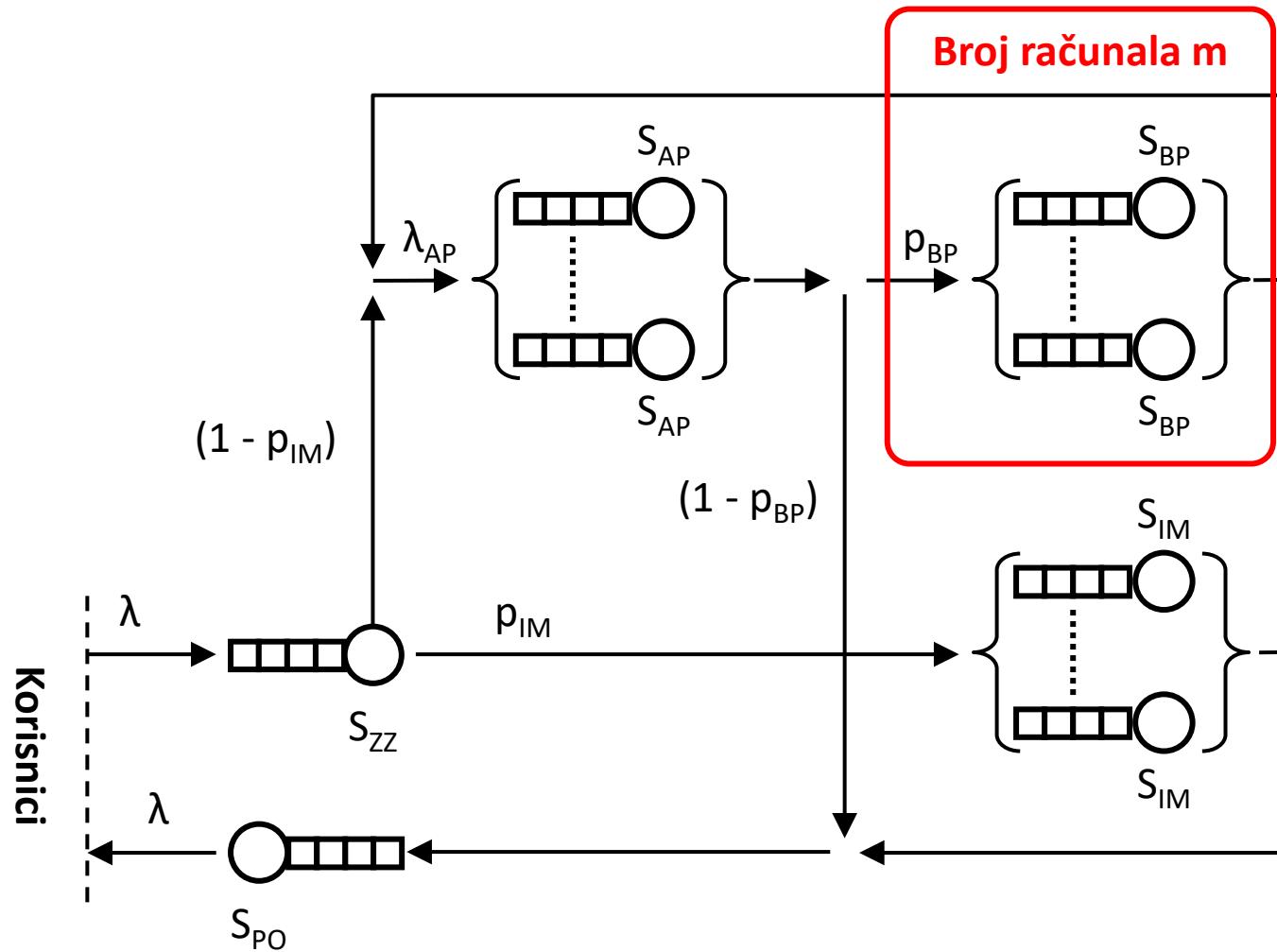


Ap\_Prist.xls



|           |         |
|-----------|---------|
| $m_{AP}$  | = 1     |
| $m_{BP}$  | = 1     |
| $m_{IM}$  | = 1     |
| $S_{AP}$  | = 0,3   |
| $S_{BP}$  | = 4.5   |
| $S_{IM}$  | = 0,5   |
| $S_{ZZ}$  | = 0,001 |
| $S_{PO}$  | = 0,001 |
| $p_{IM}$  | = 0,0   |
| $\lambda$ | = 0,5   |

# Veličina grozda baze podataka



# Vrijeme zadržavanja zahtjeva



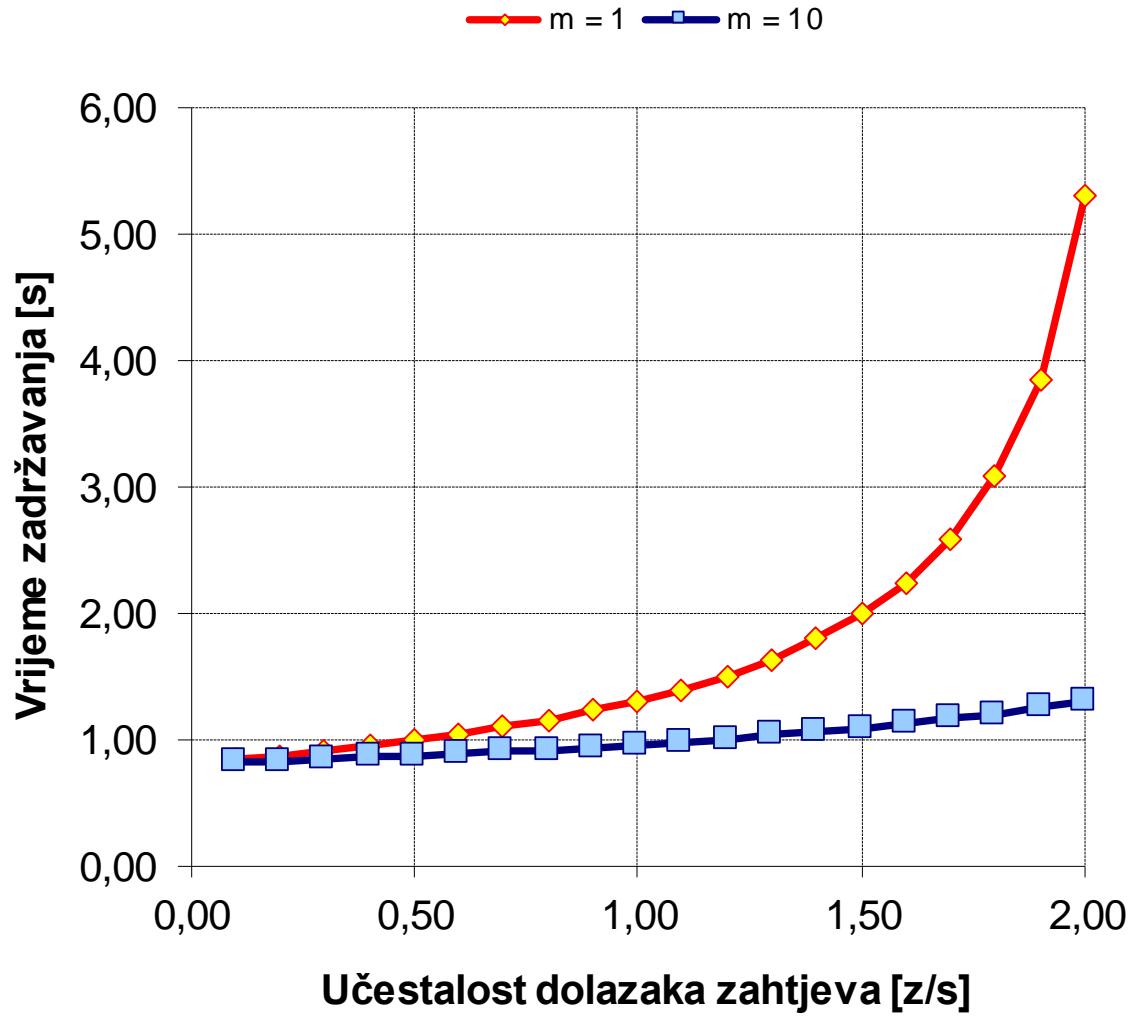
Response.  
java



ap-Zad.c



Ap\_Rac.xls

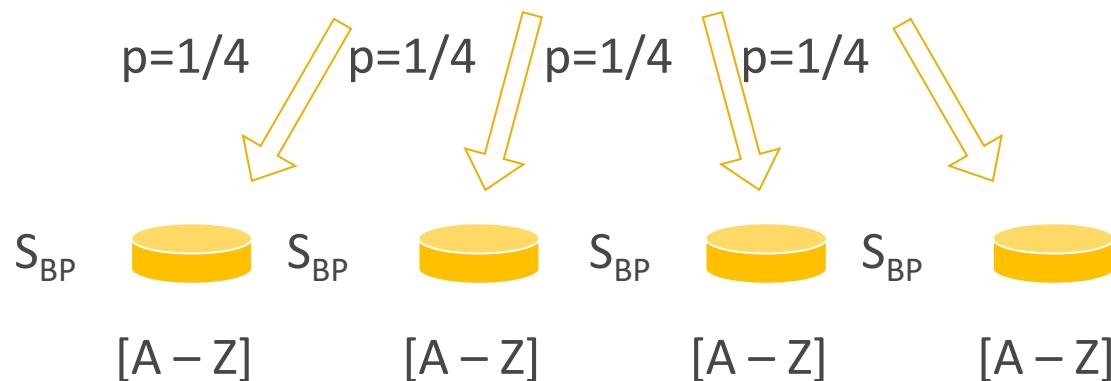


$$\begin{aligned}S_{AP} &= 0,3 \\S_{BP} &= 4,5 \\S_{IM} &= 0,5 \\S_{ZZ} &= 0,001 \\S_{PO} &= 0,001 \\p_{IM} &= 0,1 \\P_{BP} &= 0,1\end{aligned}$$

# Promjena organizacije podataka (1)

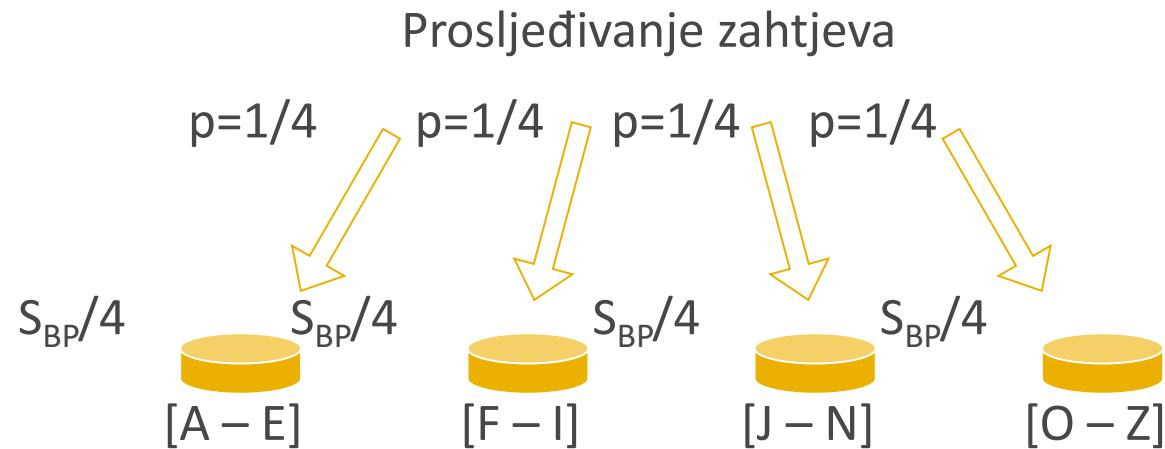
- Replikacija podataka:
  - Skup računala od kojih svako u spremniku sadrži kopiju cijele baze podataka
  - Zahtjevi se raspoređuju na računala s ciljem uravnoteženja opterećenja

Raspoređivanje zahtjeva



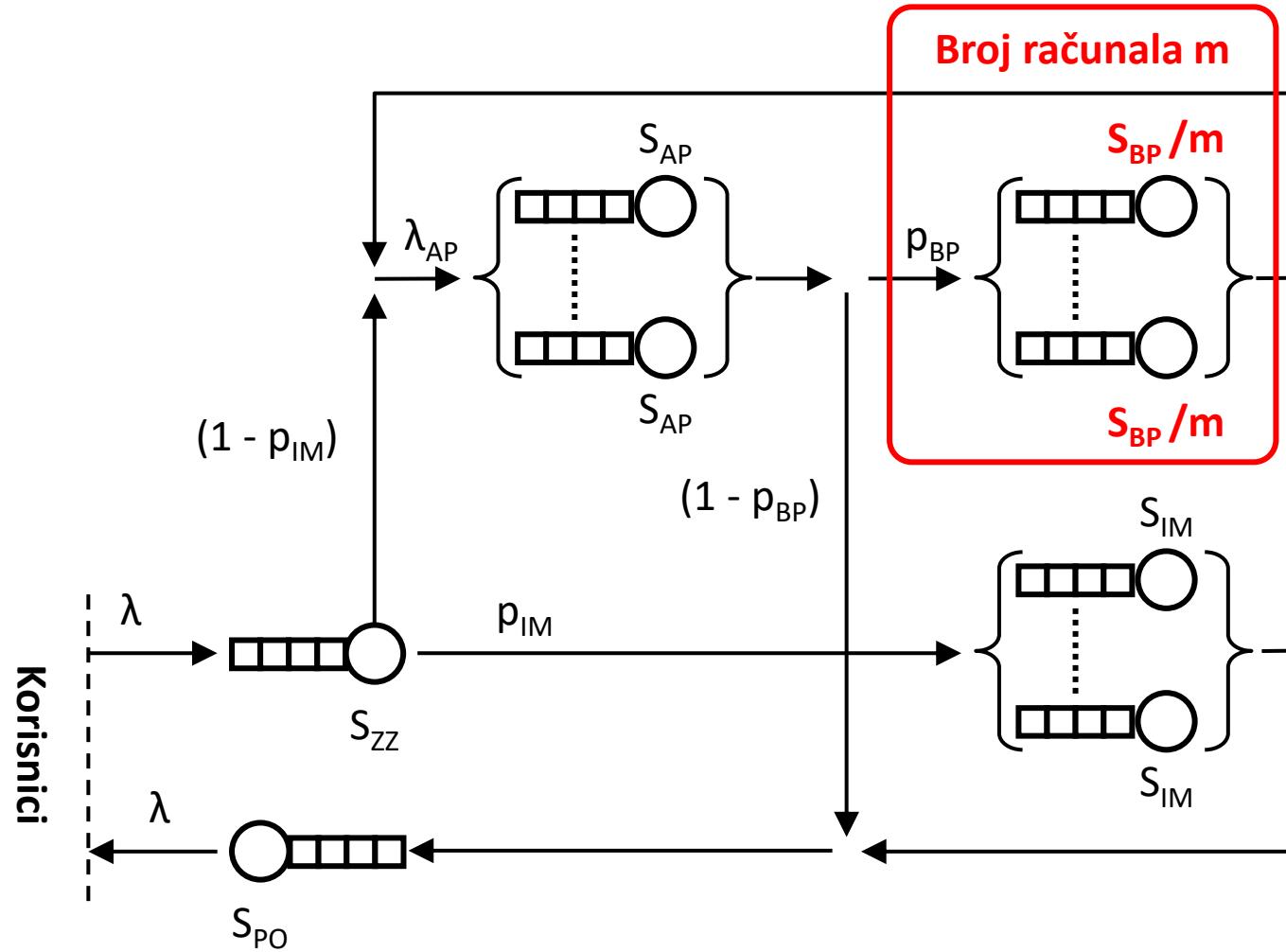
# Promjena organizacije podataka (2)

- Segmentacija podataka:
  - Skup računala od kojih svako u spremniku sadrži dio cijele baze podataka
  - Zahtjevi se proslijeduju prema računalu s traženim zapisima



- Prepostavke
  - Uniformna raspodjela zahtjeva na zapise
  - Linearna složenost obrade zahtjeva o količini zapisa

# Promjena organizacije podataka (3)



# Vrijeme zadržavanja zahtjeva



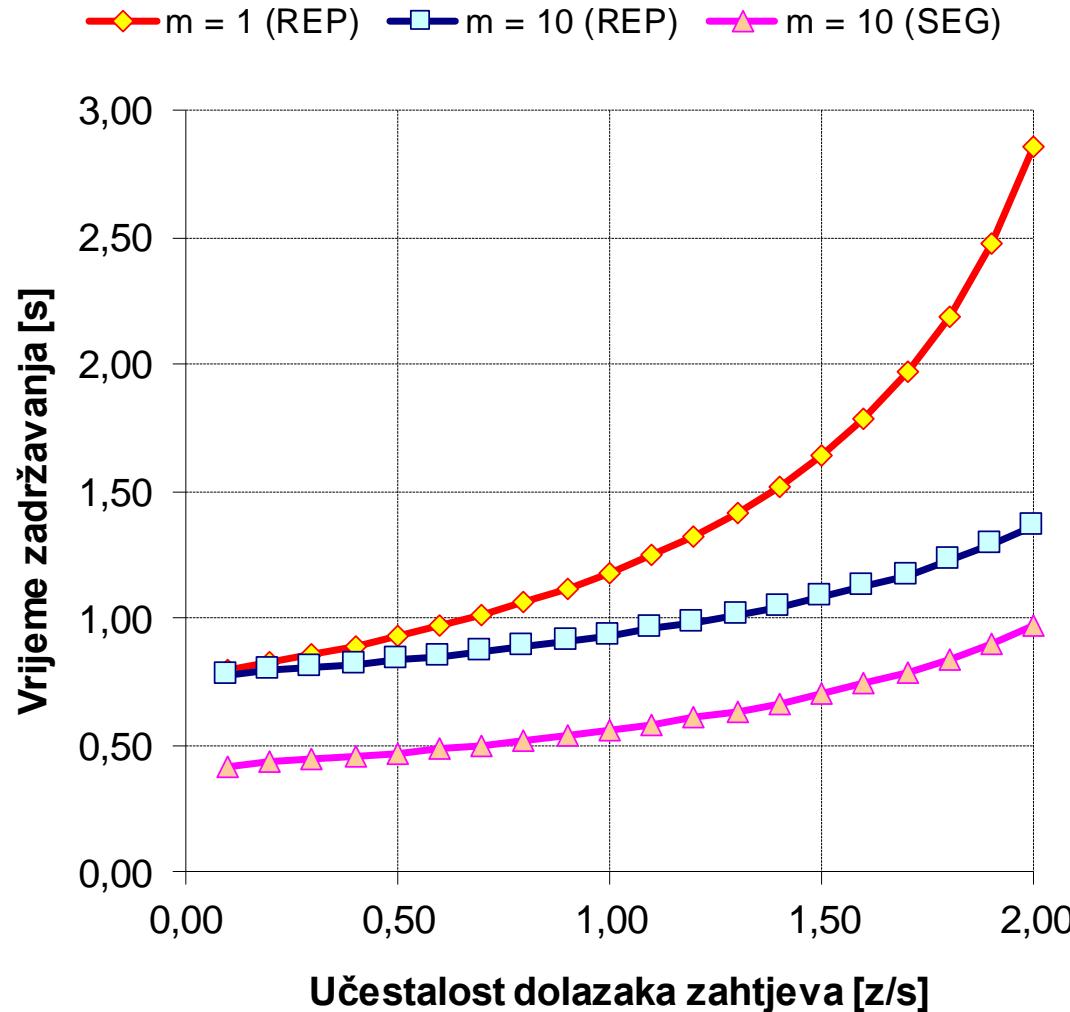
Organizati  
on.java



ap-PodOrg.c



Ap\_Pod.xls

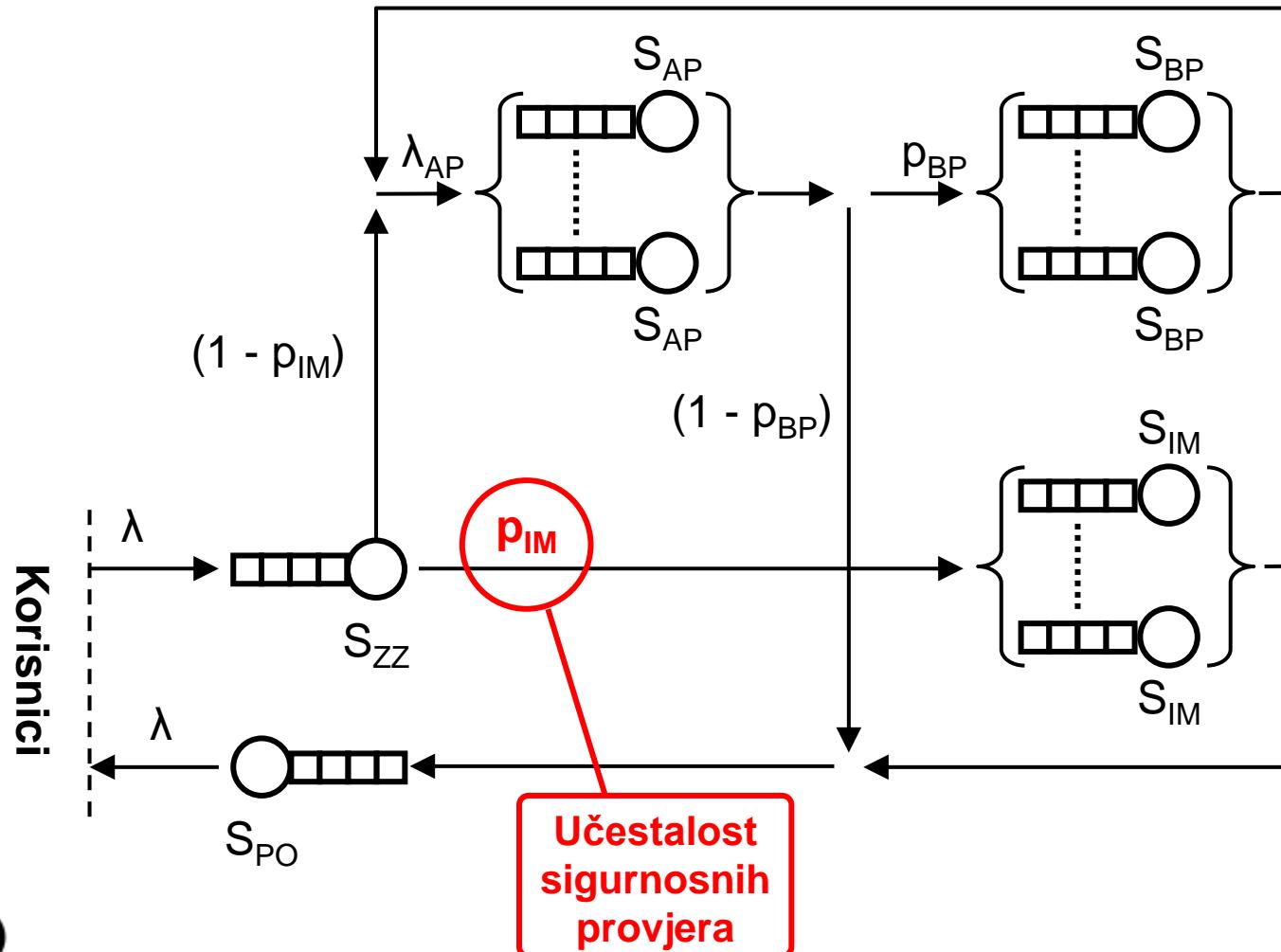


|          |         |
|----------|---------|
| $m_{AP}$ | = 1     |
| $m_{BP}$ | = 1     |
| $m_{IM}$ | = 1     |
| $S_{AP}$ | = 0,3   |
| $S_{BP}$ | = 2,5   |
| $S_{IM}$ | = 0,5   |
| $S_{ZZ}$ | = 0,001 |
| $S_{PO}$ | = 0,001 |
| $p_{IM}$ | = 0,1   |
| $p_{BP}$ | = 0,15  |

# Utjecaj stupnja sigurnosti

- Imenik aplikacije sadrži informacije o korisnicima
  - Korisnički identiteti
  - Korisnička prava pristupa
- Sigurnosna značka
  - Određuje sigurnosne postavke korisnika aplikacije
  - Značka se dohvaca iz imenika
- Životni vijek sigurnosne značke
  - Ograničeni broj pristupa
  - Zadano vrijeme korištenja
  - Ostali sigurnosni modeli

# Promjena stupnja sigurnosti



# Vrijeme zadržavanja zahtjeva



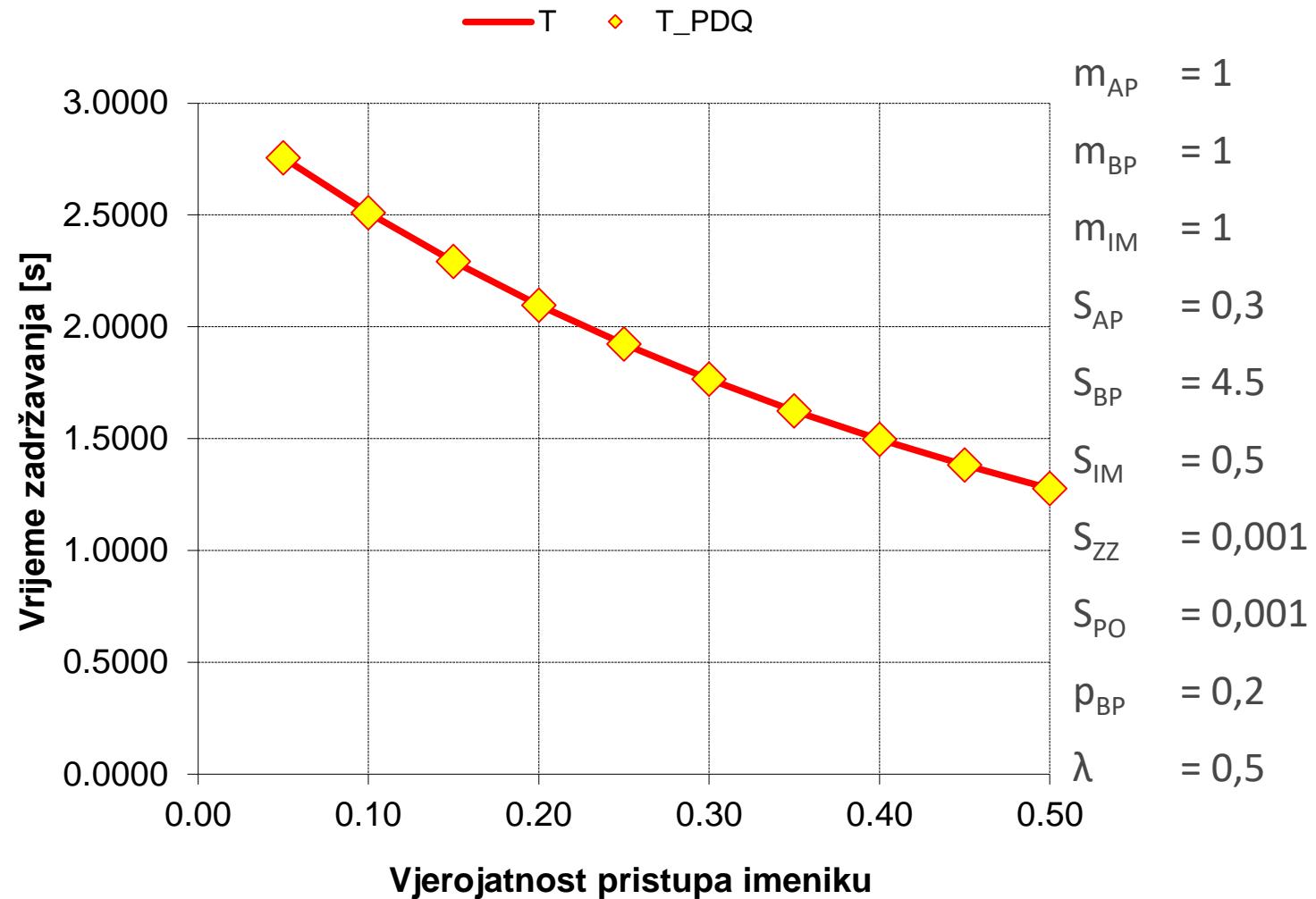
DirectoryAccess.java



ap-Sig.c

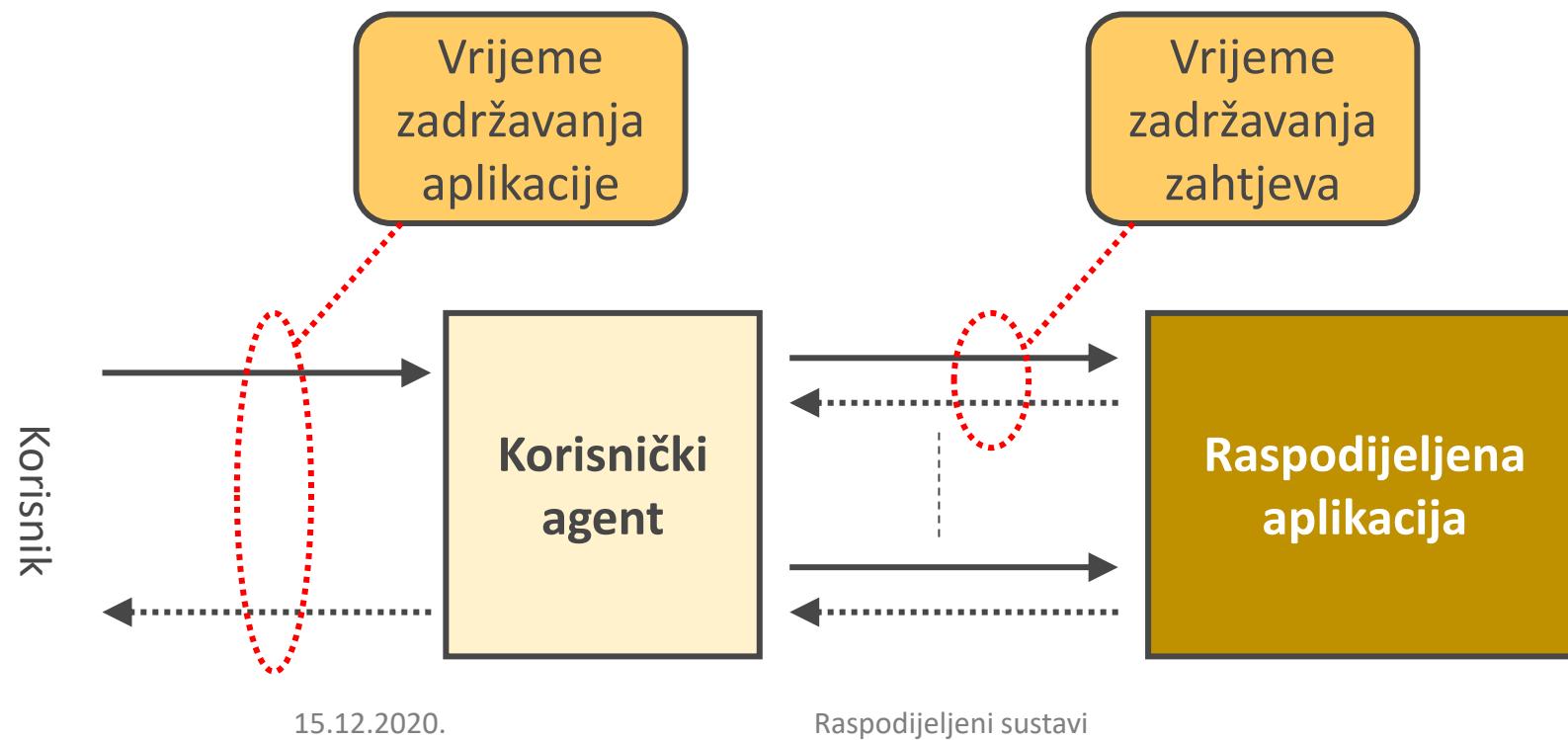


Ap\_Sig.xls



# Utjecaj stupnja sigurnosti

- Zašto vrijeme zadržavanja opada?
  - Modelirano je vrijeme zadržavanja zahtjeva ali ne i ukupno vrijeme zadržavanja aplikacije koje doživljava korisnik



# Sadržaj predavanja

- Modeliranje i analiza raspodijeljenih sustava mrežom repova
  - Jednopoloslužiteljski sustav
  - Littleov zakon
  - Serijski i paralelni poslužitelji
  - Poslužitelj s povratnom vezom
- Alat PDQ (Pretty Damn Quick)
  - Primjeri uporabe alata PDQ
  - Primjer analize performansi web-aplikacije
- Domaća zadaća

# Domaća zadaća

- Zadatak: Oblikovati zadalu raspodijeljenu aplikaciju i provesti analizu performansi ostvarene aplikacije
  - 1) Definirati logičku i fizičku arhitekturu aplikacije
  - 2) Izgraditi model aplikacije primjenom teorije repova
  - Odrediti analitičko rješenje funkcije zadržavanja zahtjeva u aplikaciji  $T = f(\lambda)$
  - 3) Izgraditi model aplikacije za alat PDQ
  - Primjenom izgrađenog modela odrediti vrijednosti funkcije zadržavanja zahtjeva  $T = f(\lambda)$  u nekoliko točaka
  - 4) Usporediti i obrazložiti dobivene rezultate

# Literatura, ...

- Sadržaj ovog predavanja nastao je na temelju:
- N. J. Gunther: "The practical performance analyst", Mcgraw Hill i Authors Choice Press, 1998 i 2000. (poglavlja 2 i 3)
- D. A. Menasce, V.A.F. Almeida: "Capacity planning for web services", Prentice Hall, 2002 (poglavlja 1 i 5)
- D. A. Menasce, V.A.F. Almeida, „Capacity Planning: An Essential Tool for Managing Web Services“, IT professional, Vol. 4, No. 4, 2002., pp. 33-38.”
- D. F. Vrsalovic, et. al: "Performance prediction and calibration for a class of multiprocessors“, IEEE Transactions on Computers, Volume: 37 Issue: 11 , Nov. 1988, pp. 1353 -1365

# Rekapitulacija i pitanja za provjeru znanja (1)

- Koje su osnovne veličine u modelu repa čekanja?
  - Vrijeme promatranja ( $t$ ), broj dolazaka ( $\alpha$ ), broj odlazaka ( $\beta$ ) i vrijeme zaposlenosti poslužitelja ( $\sigma$ ).
- Koje su izvedene veličine ?
  - Ulazni intenzitet ( $\lambda = \alpha/t$ ), izlazni intenzitet ( $\delta = \beta/t$ ), prosječno vrijeme posluživanja ( $S = \sigma/\beta$ ) i zaposlenost poslužitelja ( $\rho = \sigma/t$ )
- Kako se definira stacionarno stanje?
  - $\delta = \lambda$
- Kako glasi Littleov zakon ?
  - Broj zahtjeva u sustavu proporcionalan je intenzitetu dolazaka zahtjeva i vremenu provedenom u sustavu ( $N = \lambda \times T$ )
- Kako je definirano vrijeme čekanja u repu?
  - $W = N \times S$
- Kako je definirano ukupno vrijeme provedeno u sustavu?
  - $T = S + W$

# Rekapitulacija i pitanja za provjeru znanja (2)

- Kako se izračunava vrijeme odziva za serijske repove?
  - $T = (Q_1 + Q_2 + \dots + Q_m)/\lambda$
- Kako se izračunava vrijeme odziva za paralelne repove?
  - $T = S/(1 - \rho')$
- Kako je definirana iskoristivost poslužitelja u sustavu s m poslužitelja?
  - $\rho' = \rho/m$
- Za koliko će se povećati vrijeme zadržavanja u sustavu s povratnom vezom ako se vjerojatnost vraćanja zahtjeva na ponovnu obradu poveća s 10 % na 20 %?
  - $T_a = T_1 / (1 - 0,2)$
  - $T_b = T_1 / (1 - 0,3)$
  - $T_b / T_a = (1 - 0,3) / (1 - 0,2) = 0,7/0,6 = 1,166 (16,6 \%)$



SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

# Raspodijeljeni sustavi

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

**12. Sustavi s ravnopravnim sudionicima**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

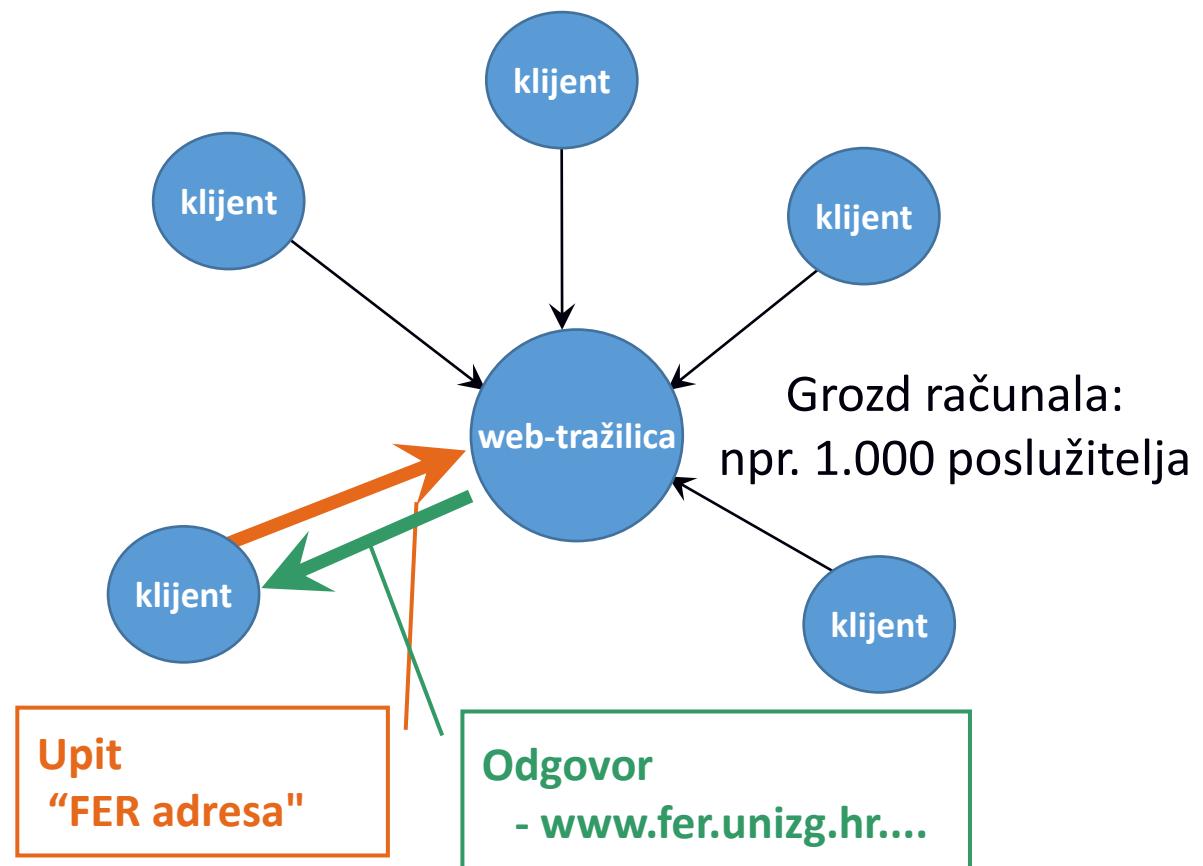
# Sadržaj predavanja

- Centralizirani i decentralizirani raspodijeljeni sustavi
- Definicija sustava P2P
- Nestrukturirani sustavi P2P
- Strukturirani sustavi P2P
- Primjeri sustava P2P

# Centralizirani raspodijeljeni sustavi (1)

Primjer: web-tražilica

- $45 \cdot 10^9$  web stranica  $\approx 225$  TB tekstualnih dokumenata
- za održavanje indeksa veličine 50 TB treba oko 1.000 računala (ovisi o raspoloživoj radnoj memoriji računala)



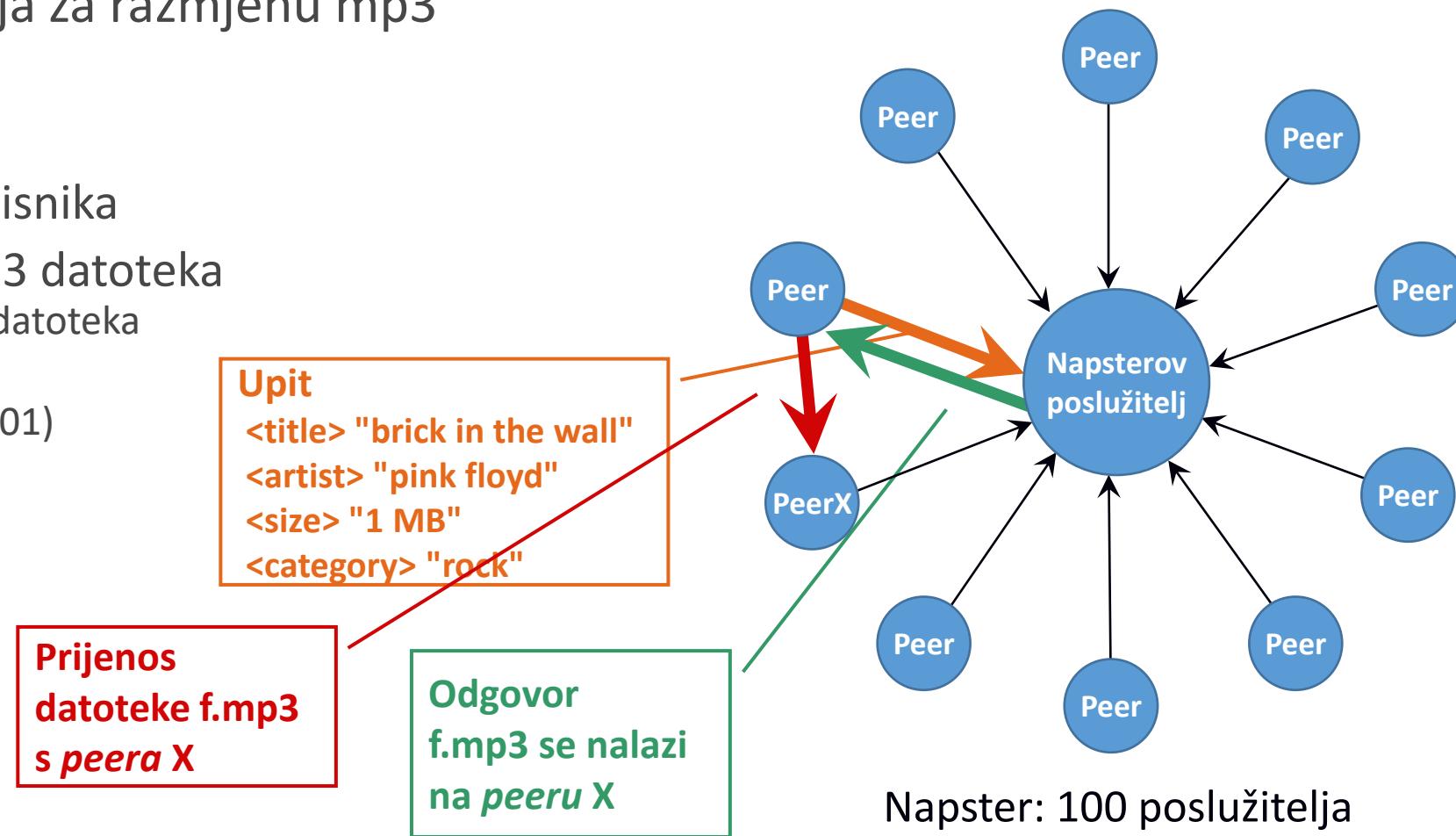
# Centralizirani raspodijeljeni sustavi (2)

## Model klijent-poslužitelj

- centralni koordinator koji prihvaca sve korisnicke upite
- indeks dokumenata je raspodijeljen, pretraživanje je raspodijeljeno u grozdu računala (*cluster*), no organizacija pretraživanja je centralizirana
- prednosti
  - efikasnost, kratko vrijeme odgovora
  - relativno jednostavna organizacija indeksa, globalno rangiranje...
- nedostaci
  - cijena (infrastruktura, administracija...)

# Decentralizirani raspodijeljeni sustavi (1)

- Primjer: aplikacija za razmjenu mp3 datoteka
  - npr. Napster
  - 1,570,000 korisnika
  - 2,000,000 mp3 datoteka (u prosjeku 220 datoteka po korisniku) (podaci za 02/2001)



# Decentralizirani raspodijeljeni sustavi (2)

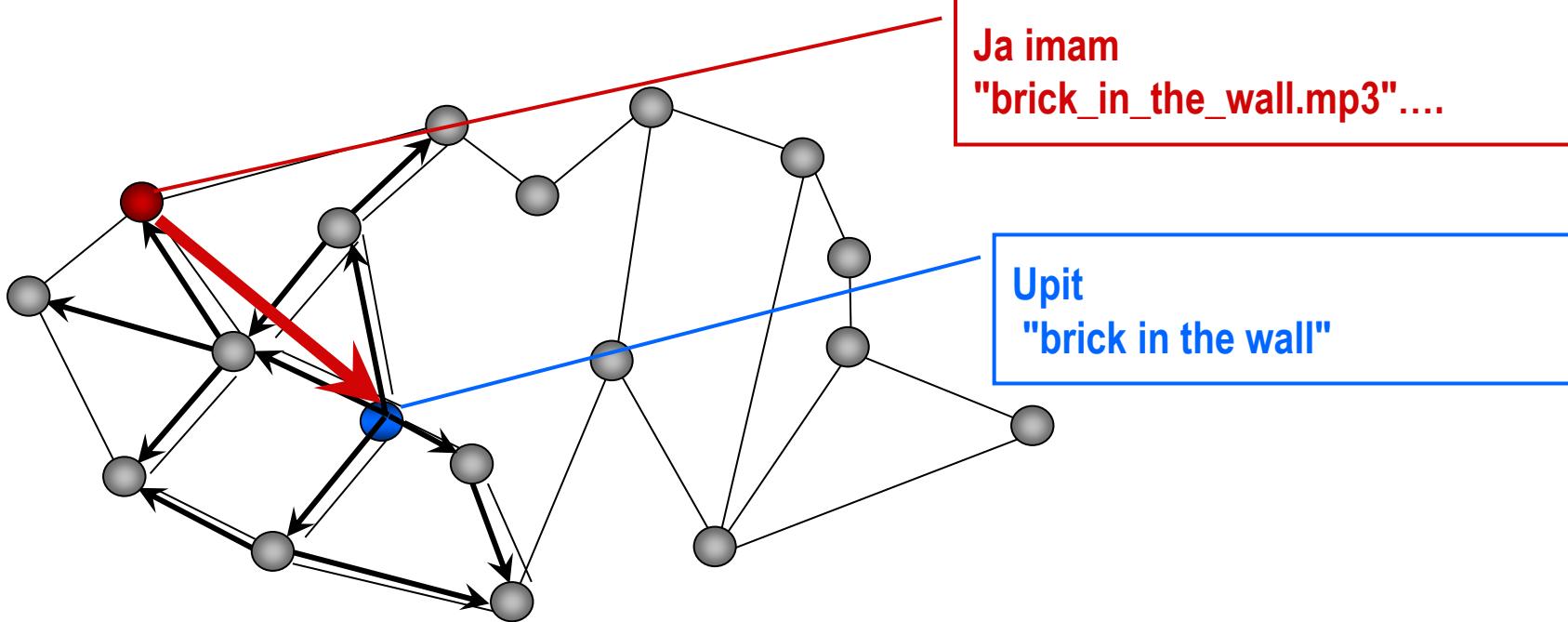
- pretraživanje je i dalje centralizirano
  - postoji centralizirani indeks s podacima o lokaciji datoteka
- pohrana i *download* datoteke je decentraliziran
- broj potrebnih poslužitelja je znatno manji jer se resursno zahtjevne operacije izvode na decentralizirani način
- prednosti
  - dijeljenje resursa, svaki čvor (*peer*) "plaća" sudjelovanje u mreži vlastitim resursima (disk, mreža, datoteke)
  - znatno manja cijena infrastrukture i održavanja
- nedostaci
  - centralizirano pretraživanje i jedinstvena točka ispada

# Decentralizirani raspodijeljeni sustavi (3)

Primjer: aplikacija za razmjenu datoteke

npr. Gnutella

40.000 čvorova,  $3 \cdot 10^6$  datoteka  
(podaci iz 08/2000)



Ja imam  
"brick\_in\_the\_wall.mp3"....

Upit  
"brick in the wall"

Potpuno decentralizirani sustav

# Decentralizirani raspodijeljeni sustavi (4)

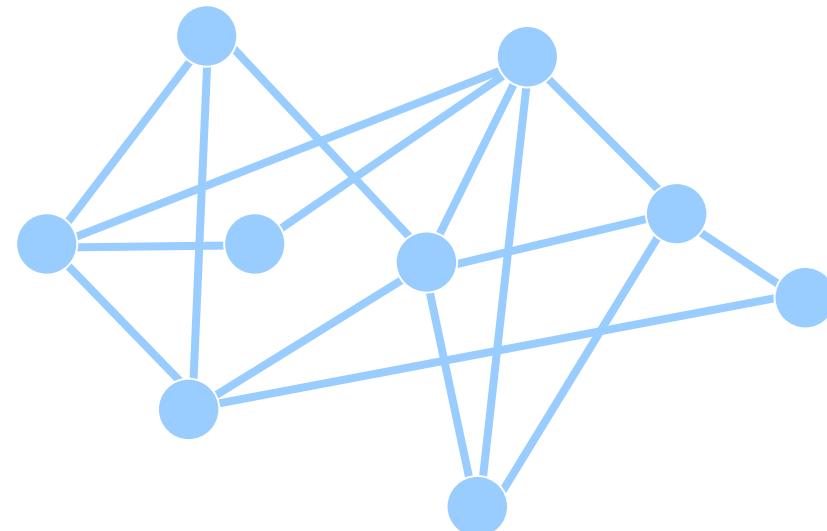
- Gnutella je primjer potpuno decentraliziranog sustava
  - svi čvorovi sudjeluju u procesu pretraživanja (ne postoji centralizirani indeks)
  - brzo pronađi datoteke koje su replicirane na velikom broju čvorova
- prednosti
  - skalabilnost sustava
  - ne postoji posebna infrastruktura niti potreba za održavanjem sustava
  - ne postoji jedinstvena točka ispada
- nedostaci
  - velika količina generiranog mrežnog prometa
  - ne postoji garancija pronađaska tražene datoteke
  - problem: tzv. *free-riding*, postoje peerovi koji ne dijele datoteke

# Sadržaj predavanja

- Centralizirani i decentralizirani raspodijeljeni sustavi
- **Definicija sustava P2P**
- Nestrukturirani sustavi P2P
- Strukturirani sustavi P2P
- Primjeri sustava P2P

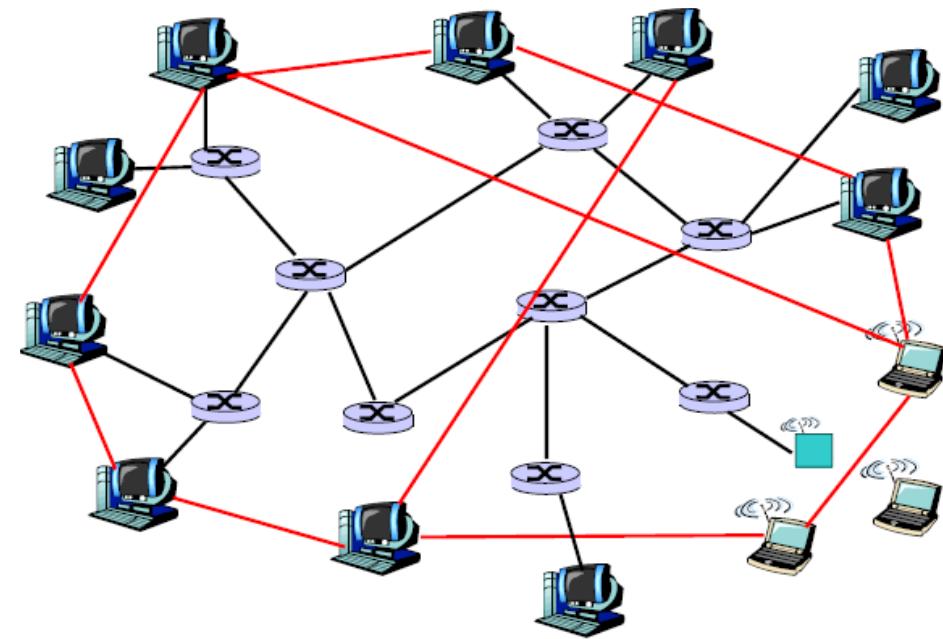
# Definicija sustava peer-to-peer (P2P)

- mreža ravnopravnih sudionika tj. "čvorova" - *peerova*
- svaki *peer* istovremeno obavlja funkciju poslužitelja i klijenta
- svaki čvor "plaća" sudjelovanje u mreži nudeći dio vlastitih resursa (memorija, CPU, mreža) ostalim čvorovima
- *peerovi* ulaze i izlaze iz sustava po volji, dinamična i nestabilna topologija
- potencijalno sustav P2P nudi neograničene resurse (broj *peerova* nije ograničen)



# *Overlay network*

- “prekrivajuća mreža” (*overlay network*) nad stvarnom mrežnom topologijom
- *peerovi* su programi koji se izvode na aplikacijskom sloju
- koristi resurse krajnjih računala koji čine posebnu mrežu na aplikacijskom sloju neovisnu o mrežnoj topologiji
- mreža *peerova* se konstantno mijenja



# Mreža peerova

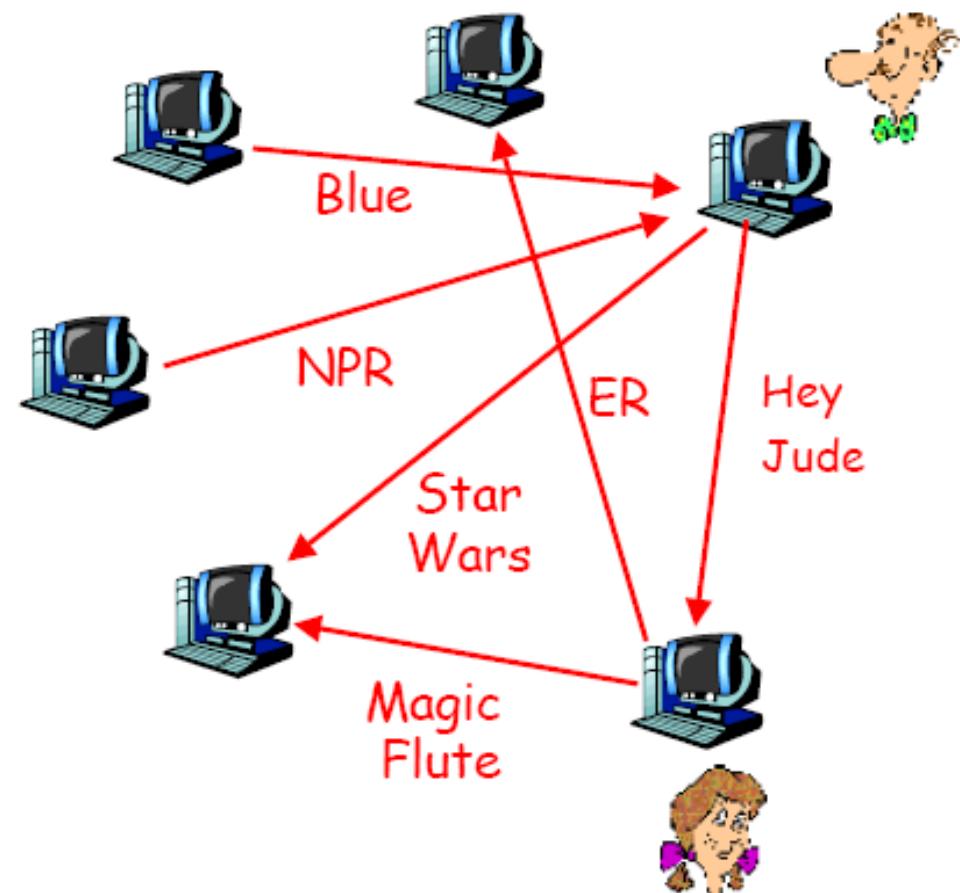
- Kada su 2 *peera* susjedi?
  - otvorena TCP konekcija ili
  - virtualne grane među *peerovima*, *peer* zna IP adresu drugog *peera*
- Kako se održava mreža *peerova*?
  - mreža je izrazito nestabilna
  - npr. *peer* periodički provjerava stanje susjeda (porukama *ping*)
  - ako je susjed nedostupan, briše se iz liste susjeda
  - potreban je poseban algoritam za otkrivanje novih susjeda
  - poseban algoritam za dodavanje novog *peera* u postojeću mrežu (najčešće poznaje listu *peerova* za inicijalni kontakt)

# Obilježja sustava P2P

- decentralizirani raspodijeljeni sustav
  - nema centralizirane koordinacije među *peerovima*
  - ne postoji jedna točka ispada
- samoorganizirajuća mreža čvorova
  - *peerovi* su međusobno neovisni
  - dodavanje novih čvorova, izlazak čvora iz sustava te ispad čvorova je podržano organizacijom mreže P2P i definiranim protokolima
- globalni informacijski sustav bez velikih početnih ulaganja
  - raspodijeljena instalacija peerova i održavanje

# Osnovna zadaća sustava P2P (1/2)

- Pronalaženje podataka, resursa, objekata (npr. datoteka) u sustavima P2P!



# Osnovna zadaća sustava P2P (2/2)

Kako pronaći podatak  $d$  u mreži peerova?

- “naivno rješenje”: poslati upit svim peerovima u mreži
  - problemi: moram znati adrese svih peerova, što je s mrežnim prometom?
- “manje naivno rješenje”: poslati upit odabranim peerovima u mreži
  - problemi: kako odabrati peerove, hoću li sigurno pronaći podatak  $d$ ?
- “pametnije” rješenje
  - pohraniti podatak  $d$  na odabrani peer  $p$  (ili odabране peerove): dovoljno je znati adresu peera  $p$  da mu možemo proslijediti upit
  - postoji algoritam koji povezuje peera  $p$  s podatkom  $d$ , a svi peerovi u mreži znaju taj algoritam
  - isti algoritam se koristi pri pohranjivanju i traženju podatka

# Vrste sustava P2P

- nestrukturirani sustavi
  - mrežna topologija nema definiranu strukturu (“manje naivno rješenje”)
  - mrežu *peerova* čini slučajan graf, npr. peer “poznaje” svoja četiri susjeda i preko njih pretražuje cijelu mrežu
  - primjeri: Freenet, Gnutella, KaZaA, BitTorrent
- strukturirani sustavi
  - mrežna topologija je definirana i ima posebnu strukturu (“pametnije” rješenje)
  - podatku  $d$  možemo pridijeliti ključ  $k$  (svaki peer može odrediti  $k$  za  $d$ )
  - podatak  $d$  je pohranjen na *peeru* koji je “zadužen” za ključ  $k$ , a ne na peeru koji ga kreira
  - primjeri: CAN, Chord, P-Grid, Pastry

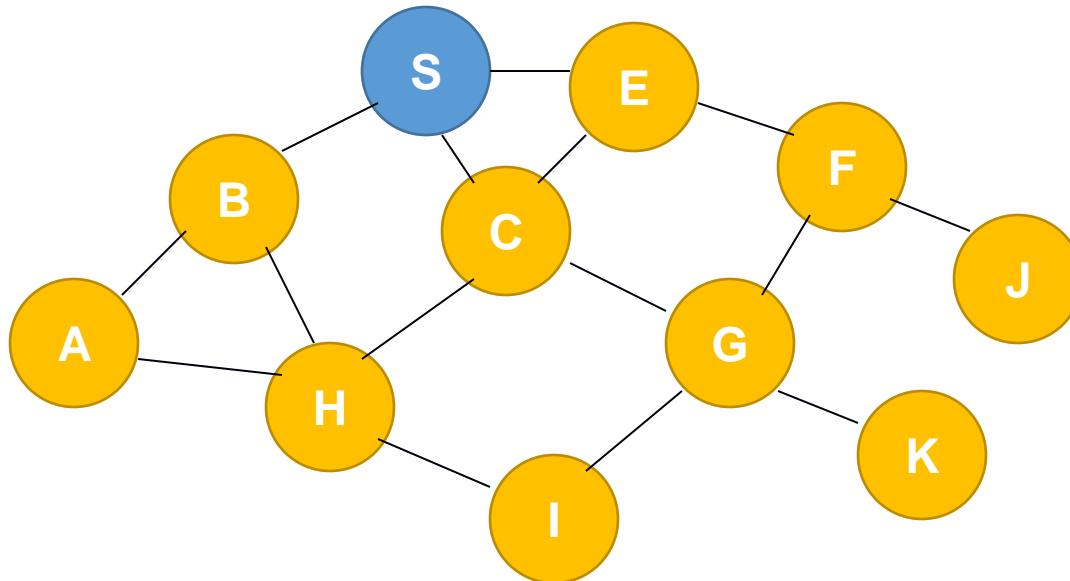
# Sadržaj predavanja

- Centralizirani i decentralizirani raspodijeljeni sustavi
- Definicija sustava P2P
- **Nestrukturirani sustavi P2P**
- Strukturirani sustavi P2P
- Primjeri sustava P2P

# Nestrukturirani sustavi P2P

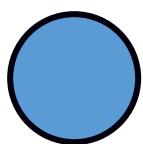
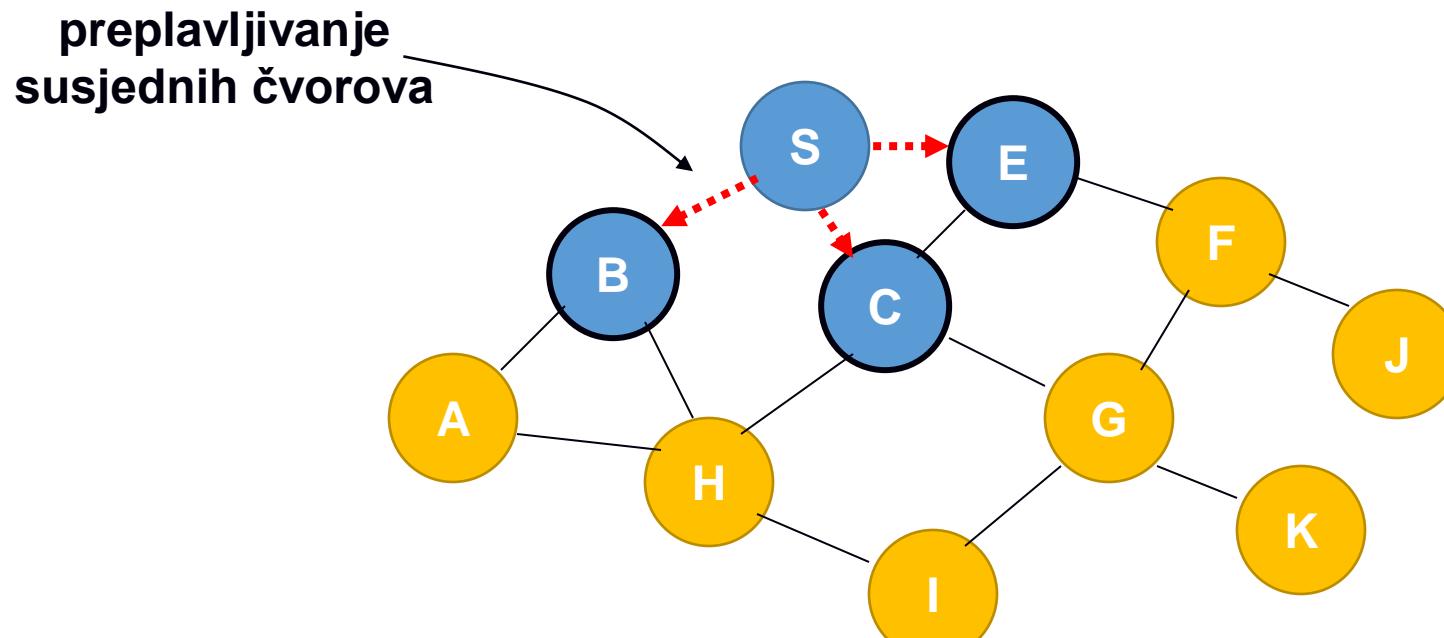
- podatak (npr. datoteka) je pohranjen na *peeru* koji ga kreira, ne postoji veza između podatka *d* i peer-a *p*
- moguće je pohraniti kopiju podatka na peerovima koji ga kopiraju s originalnog peer-a
- pretraživanje se izvodi preplavljivanjem ili slučajnim izborom (*random walk*), itd.

# Usmjeravanje upita: preplavljanje



Oznaka čvora koji je izvor upita "q".

# Usmjeravanje upita: preplavljanje

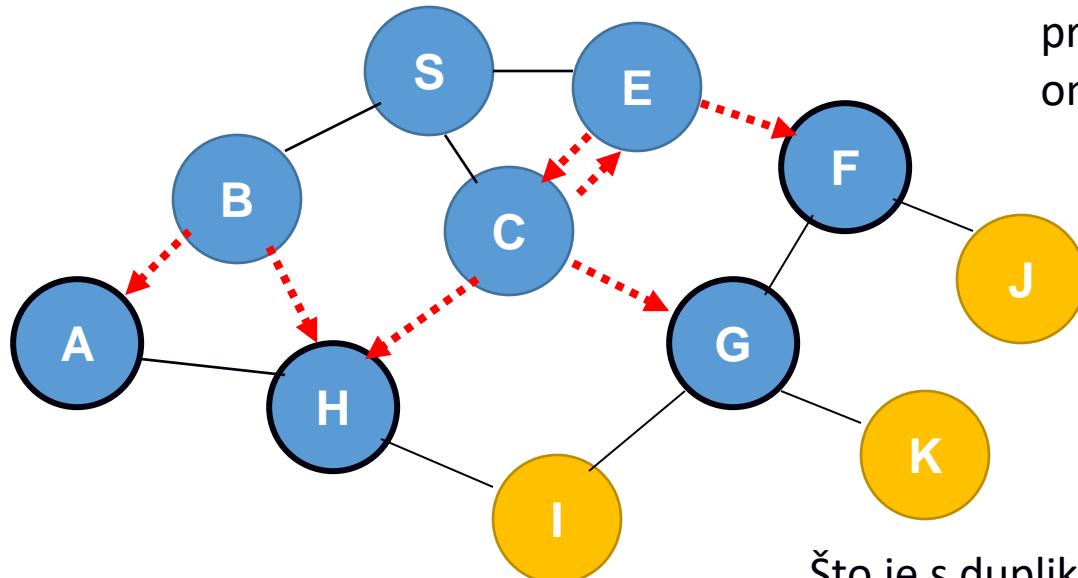


Oznaka čvora koji je primio "q" prvi put.



Prijenos upita "q"

# Usmjeravanje upita: preplavljanje



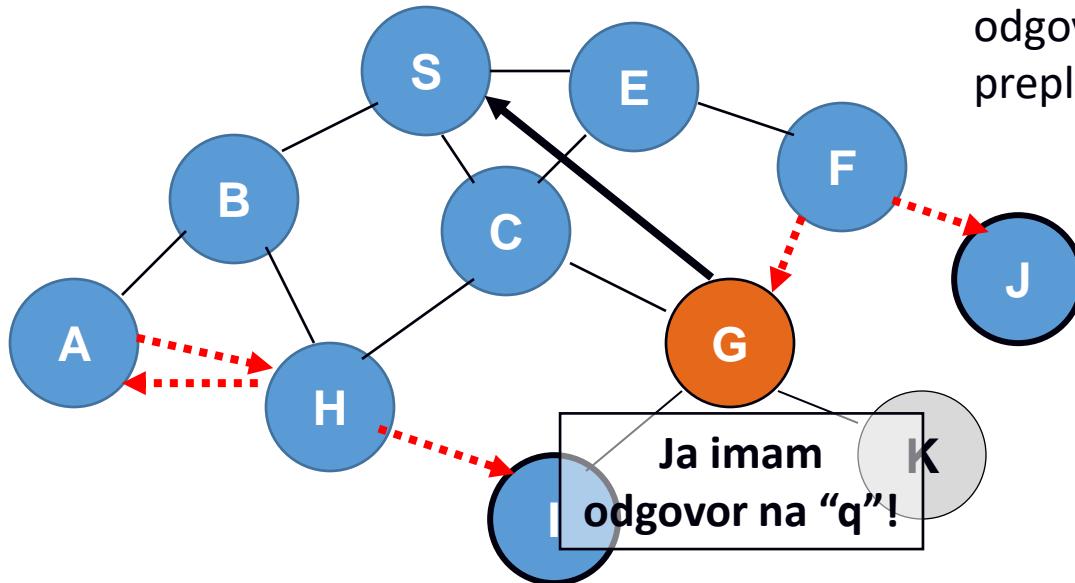
Osnovno načelo:

prosljedi upit svim susjedima osim  
onome od koga si upit primio

Što je s duplikatima?

Npr. H je primio upit od B i C. Ako  
upit ima jedinstveni identifikator, H  
uočava duplikat i ignorira ga.

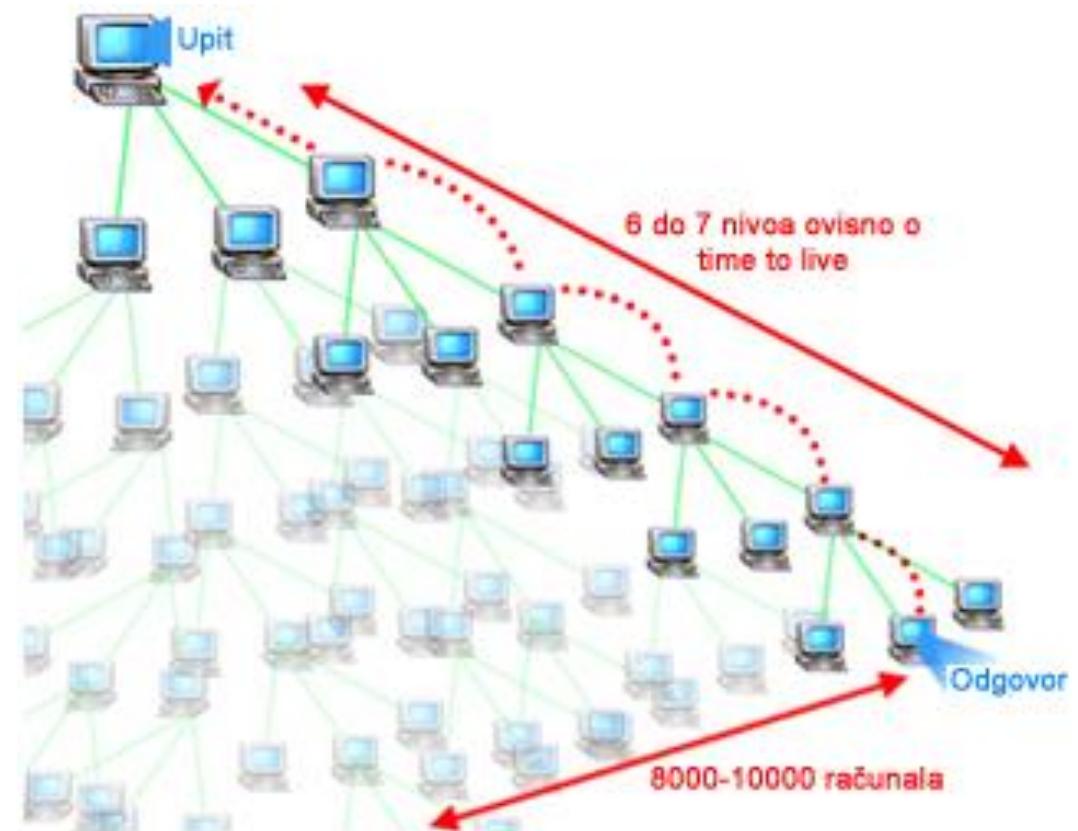
# Usmjeravanje upita: preplavljanje



G šalje odgovor direktno do S, no A, F i H ne znaju da je čvor G poslao odgovor na  $q$  i nastavljaju s preplavljivanjem svojih susjeda!

# Usmjeravanje upita: slučajni izbor

- upit se prosljeđuje odabranom podskupu susjeda
- elementi podskupa odabiru se na slučajan način



# Obilježja nestrukturiranih sustava P2P

- jednostavnost
  - jednostavan protokol za pronalaženje podataka
- robustnost
  - ne postoji jedna točka ispada
- niska cijena objavljivanja novog podatka
  - podatak ostaje pohranjen na peeru koji ga objavljuje
- velika cijena prilikom pretraživanja
  - generira se veliki mrežni promet
  - neskalabilno rješenje, komunikacijska složenost je  $O(n^2)$ , n je broj peerova
- dobro rješenje za pronalaženje podataka koji su replicirani na velikom broju peerova, ali ne za podatke pohranjene na malome broju peerova

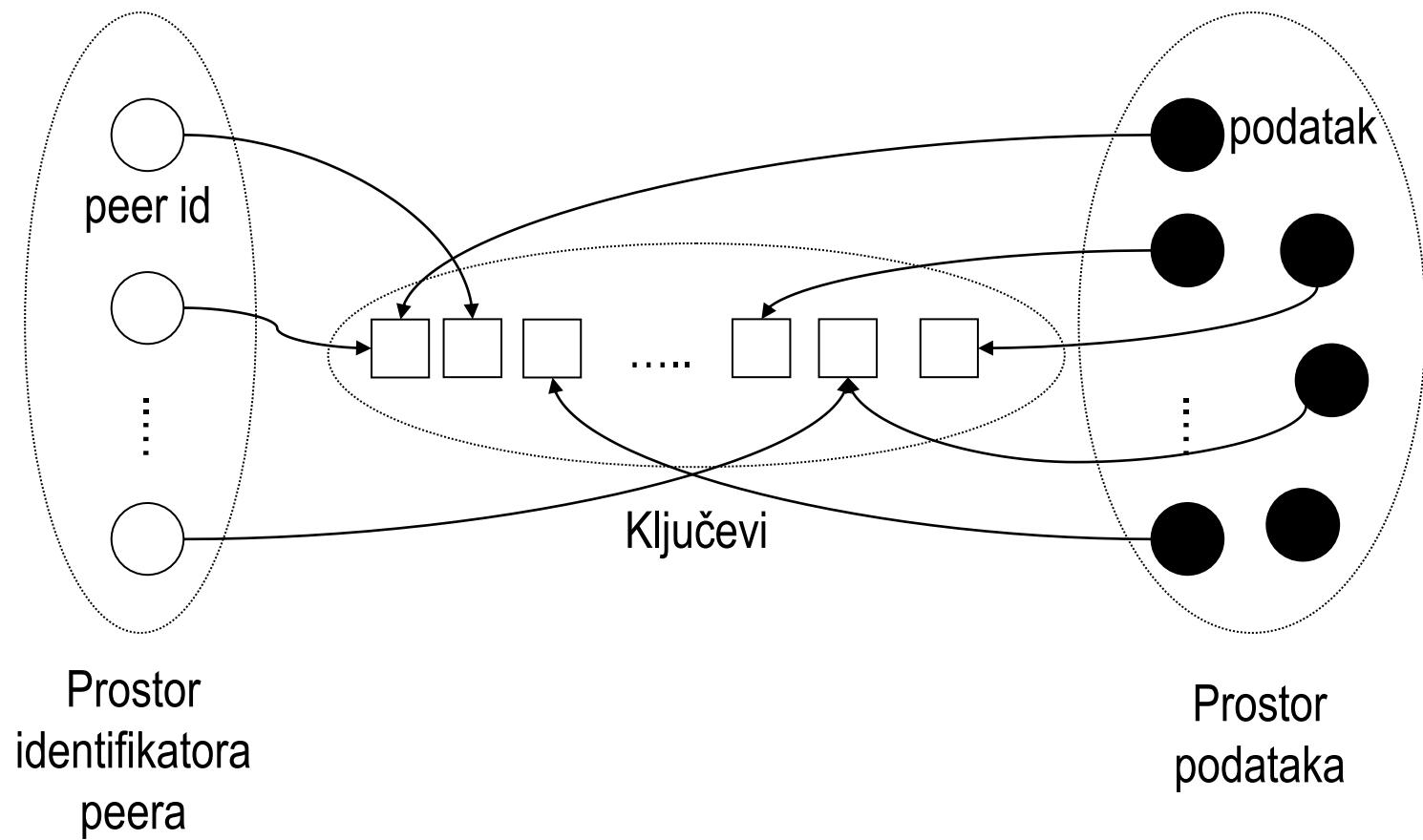
# Sadržaj predavanja

- Centralizirani i decentralizirani raspodijeljeni sustavi
- Definicija sustava P2P
- Nestrukturirani sustavi P2P
- **Strukturirani sustavi P2P**
- Primjeri sustava P2P

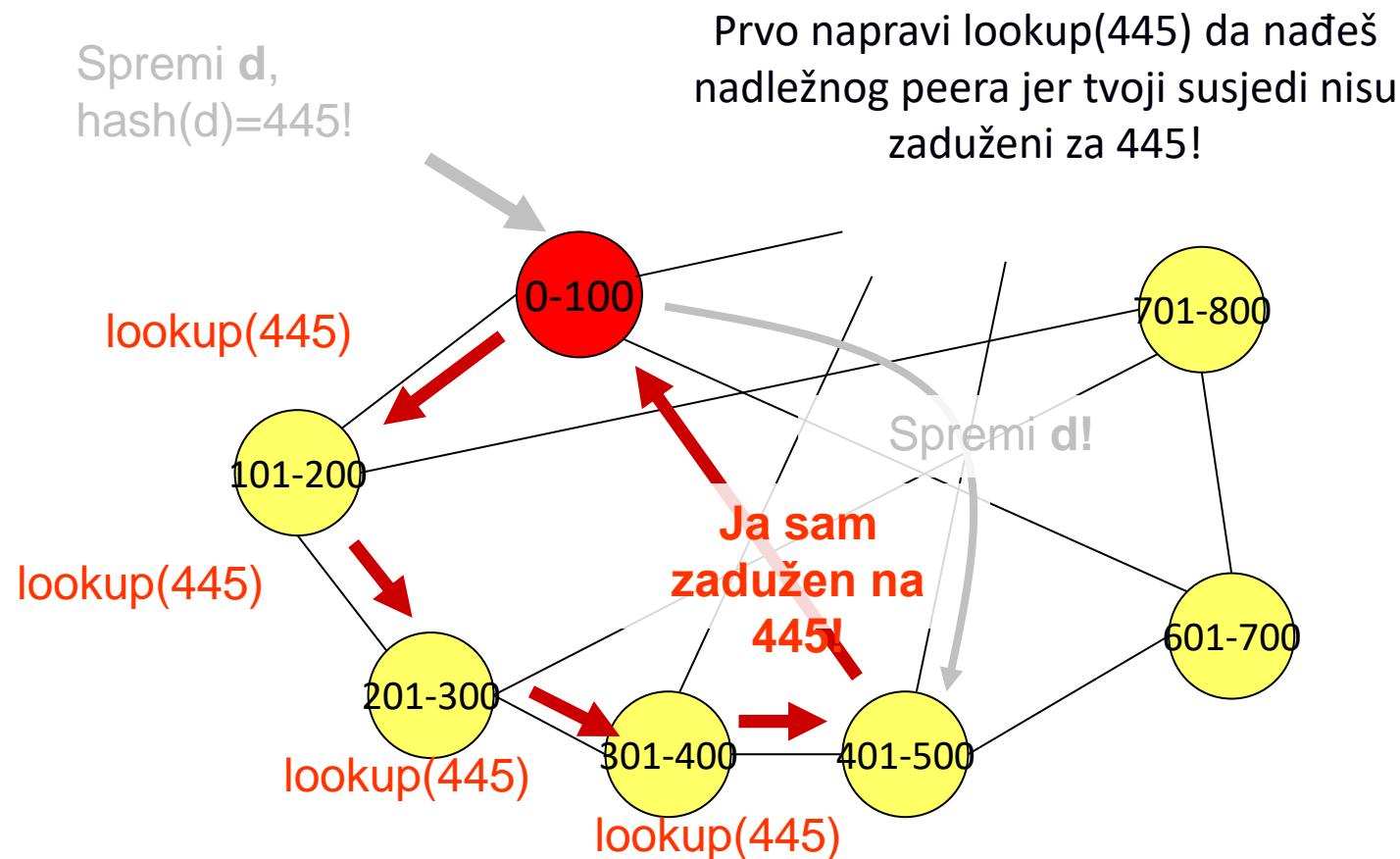
# Strukturirani sustavi P2P

- za podatak  $d$  svaki peer može izračunati ključ  $k$ 
  - npr.  $k = \text{hash}(d)$ , gdje je  $\text{hash}()$  hash funkcija
- za dani ključ  $k$  pronaći *peera p* koji je zadužen za prostor ključeva u koji spada  $k$ 
  - mreža peerova implementira metodu `lookup(k)` koja vraća identifikator *peera* za dani ključ  $k$
  - metoda `lookup(k)` je implementirana distribuirano, ako peer ne zna odgovor na upit, zna ga usmjeriti prema peeru s odgovorom
- za pohranjivanje podatka pronađimo nadležnog peera i proslijedujemo mu podatak
- prilikom pretraživanja pronađimo nadležnog peera i proslijedujemo mu upit koji opet sadrži podatak koji tražimo

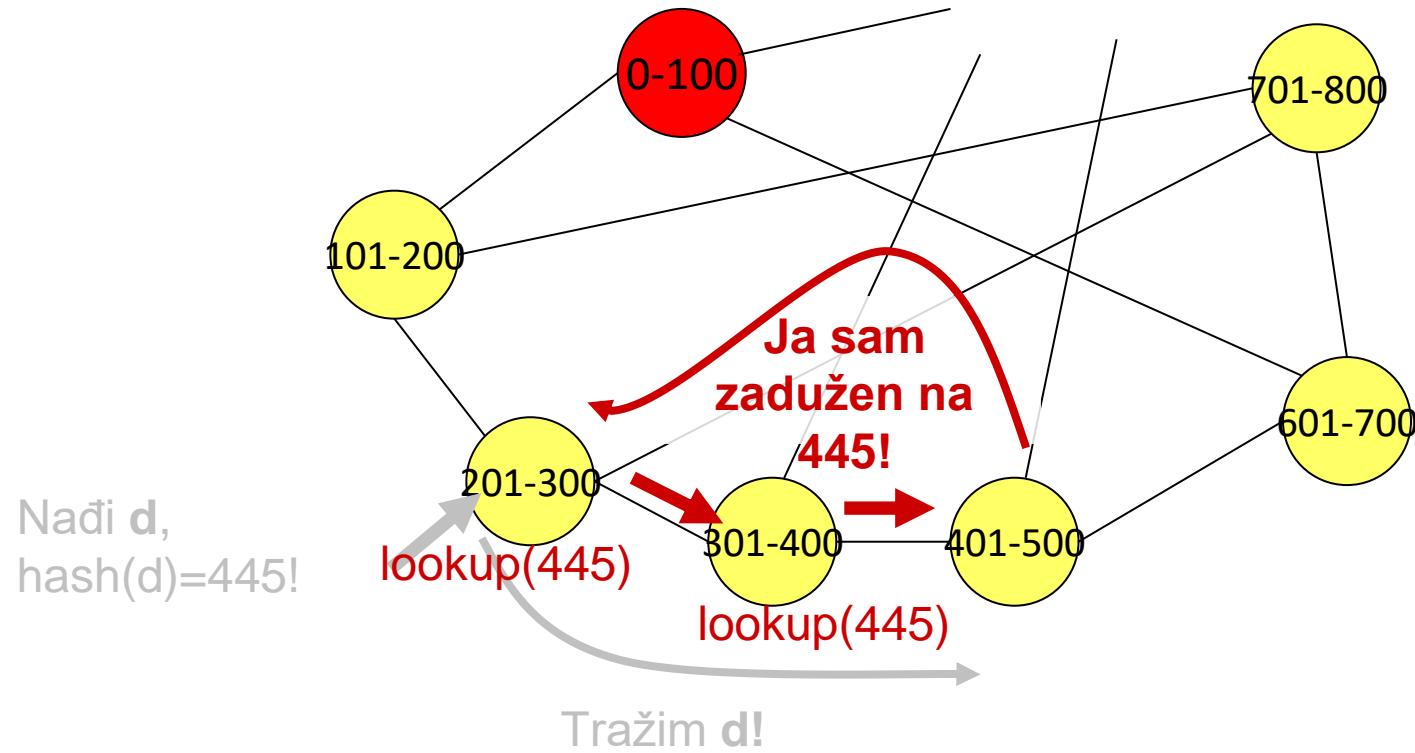
# Odnos između peera, podatka i ključa



# Ideja usmjerenja: pohranjivanje podatka



# Ideja usmjeravanja: upit



# Osobine strukturiranih sustava P2P

- garantira pohranjivanje i pronalaženje podatka u  $O(\log n)$  koraka ( $n$  je broj *peerova* u mreži)
  - skalabilno rješenje u smislu generiranog prometa u odnosu na nestrukturirane sustave
  - komunikacijska složenost  $O(n * \log n)$
- povećana cijena objavljivanja novog podatka u odnosu na nestrukturirane sustave P2P
  - podatak se pohranjuje na *peeru* koji je za njega “zadužen”
  - potrebno je održavati dodatne strukture podataka (tablice usmjerenja) radi umjeravanje upita prema *peerovima* koji pohranjuju tražene podatke

# Sadržaj predavanja

- Centralizirani i decentralizirani raspodijeljeni sustavi
- Definicija sustava P2P
- Nestrukturirani sustavi P2P
- Strukturirani sustavi P2P
- **Primjeri sustava P2P**

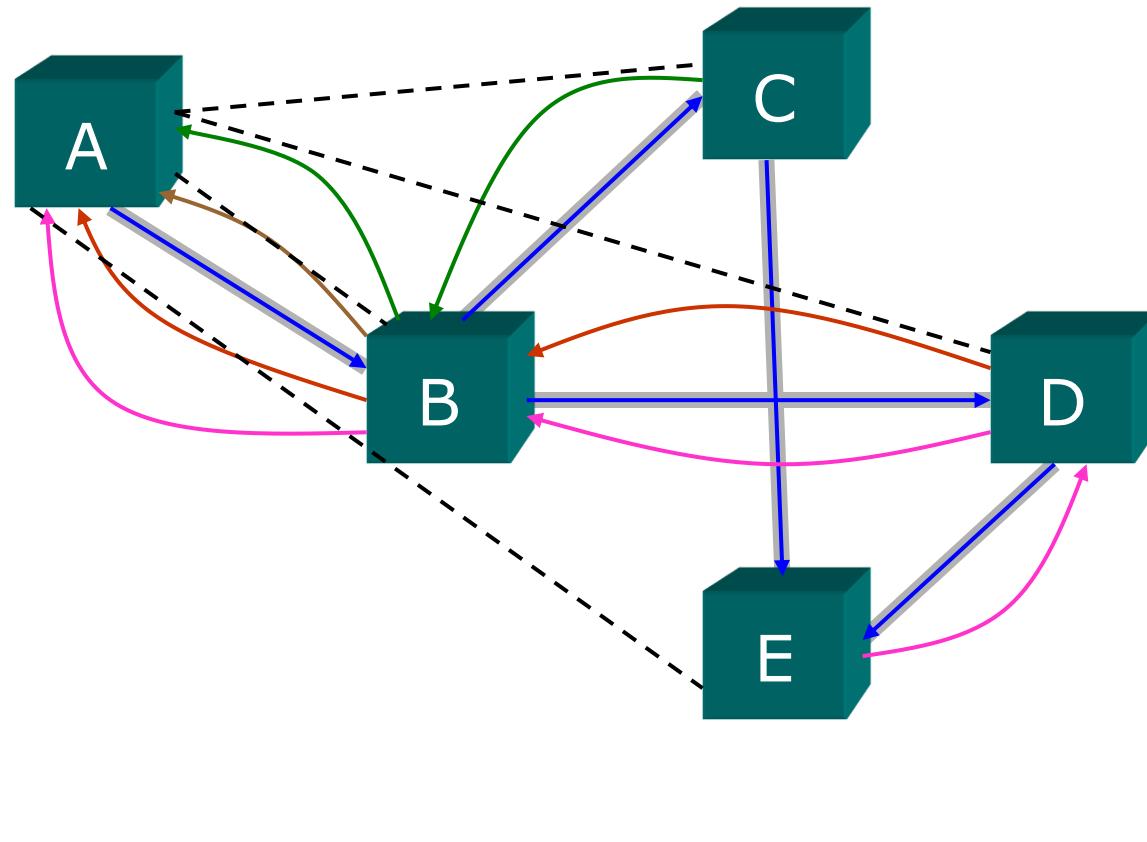
# Gnutella

- primjer nestrukturiranog sustava P2P
- svaki peer u sustavu obavlja sljedeće funkcije:
  - pohranjuje odabrane dokumente
  - generira i usmjerava upite prema svojim susjedima
  - odgovara na upite ako pohranjuje dokument koji odgovara upitu
- koristi ograničeno preplavljivanje prilikom pretraživanja
  - svaki čvor šalje upit svim svojim susjedima
  - širenje upita je ograničeno parametrom *time-to-live* (TTL = 7)
  - svaki upit ima jedinstveni identifikator zbog petlji u mreži
- novi čvor se jednostavno povezuje u sustav tako da se spoji na barem jedan poznati Gnutella čvor

# Gnutella: vrste poruka

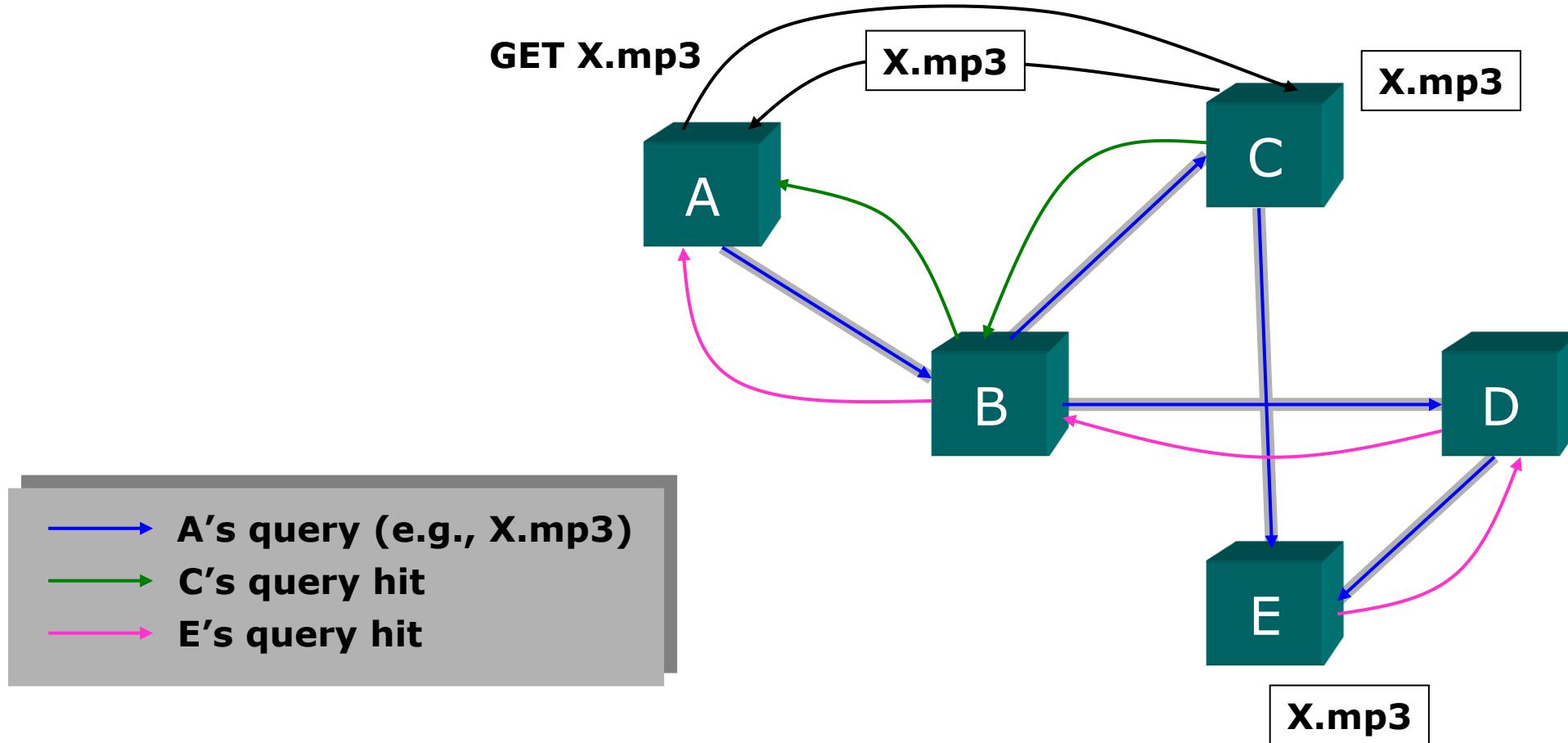
| Type     | Description                                           | Contained Information                                                                           |
|----------|-------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Ping     | Announce availability and probe for other servents    | None                                                                                            |
| Pong     | Response to a ping                                    | IP address and port# of responding servent; number and total kb of files shared                 |
| Query    | Search request                                        | Minimum network bandwidth of responding servent; search criteria                                |
| QueryHit | Returned by servents that have the requested file     | IP address, port# and network bandwidth of responding servent; number of results and result set |
| Push     | File download requests for servents behind a firewall | Servent identifier; index of requested file; IP address and port to send file to                |

# Gnutella: održavanje mrežne topologije (Ping/Pong)

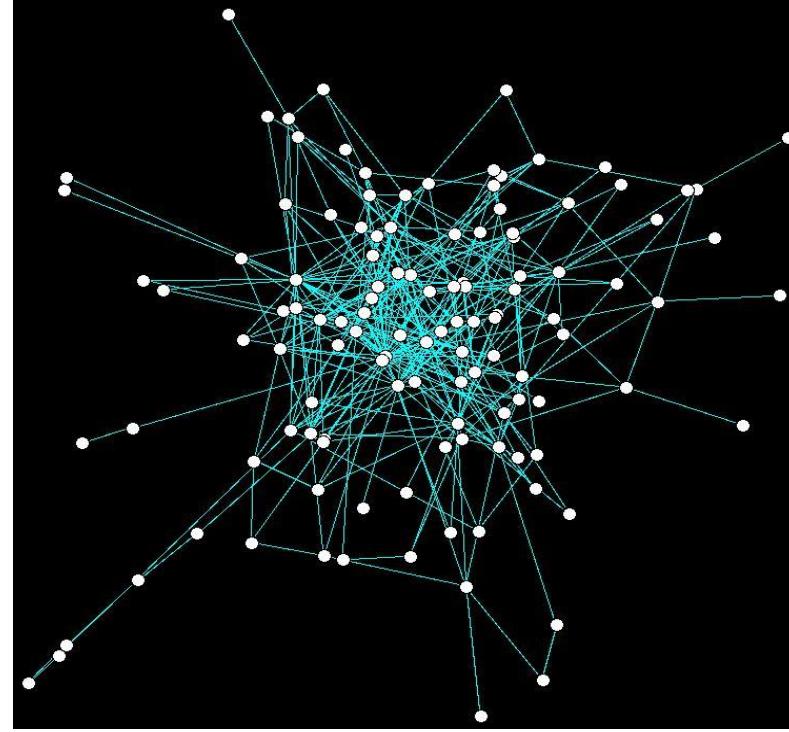
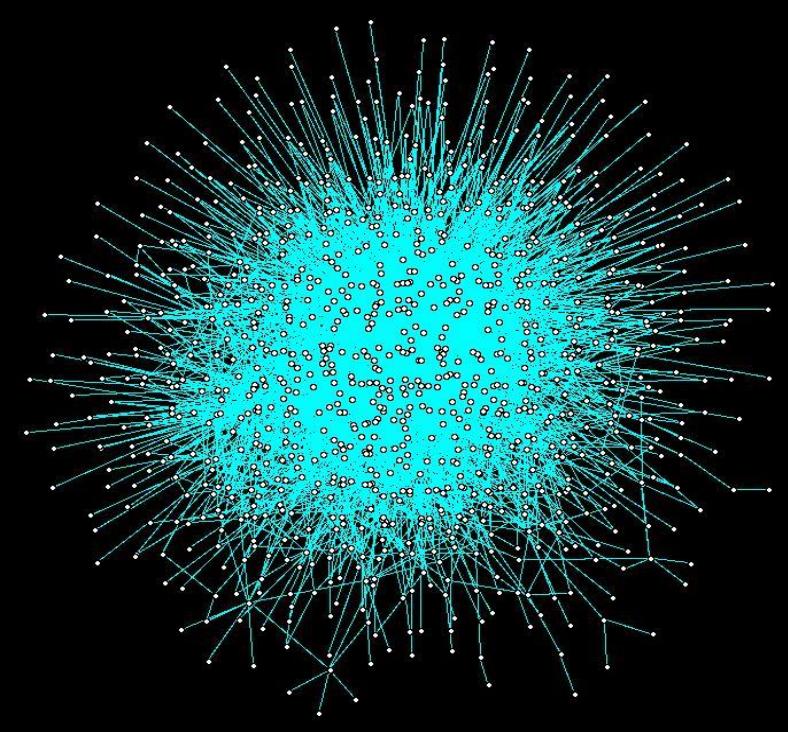


- A's ping
- B's pong
- C's pong
- D's pong
- E's pong

# Gnutella: pretraživanje (Query/QueryHit/GET)



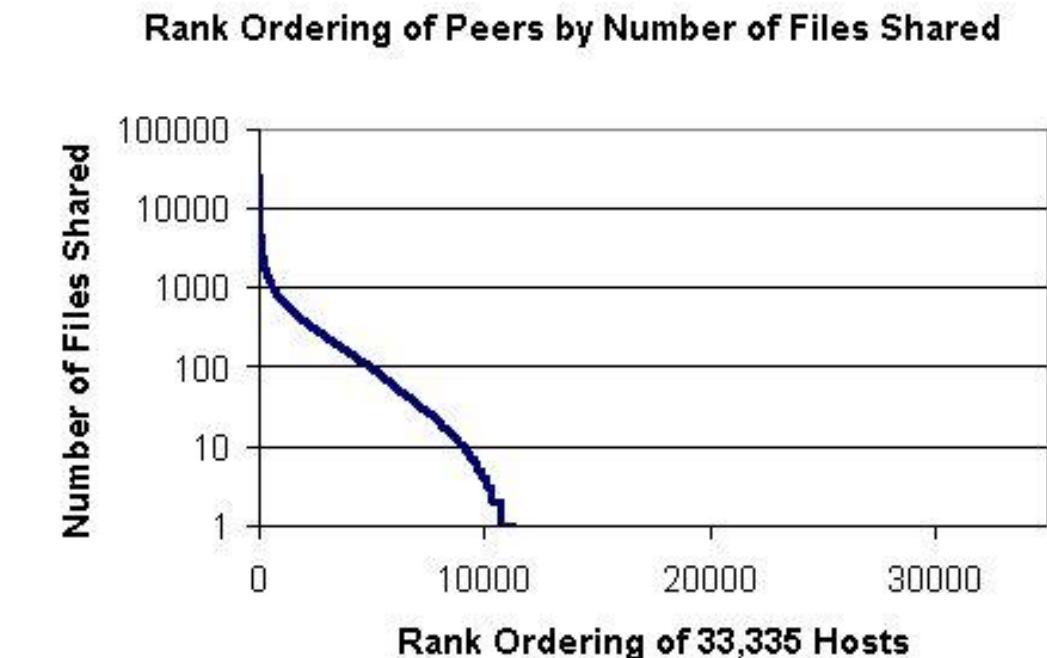
# Topologija mreže Gnutella



Jezgrena mreža

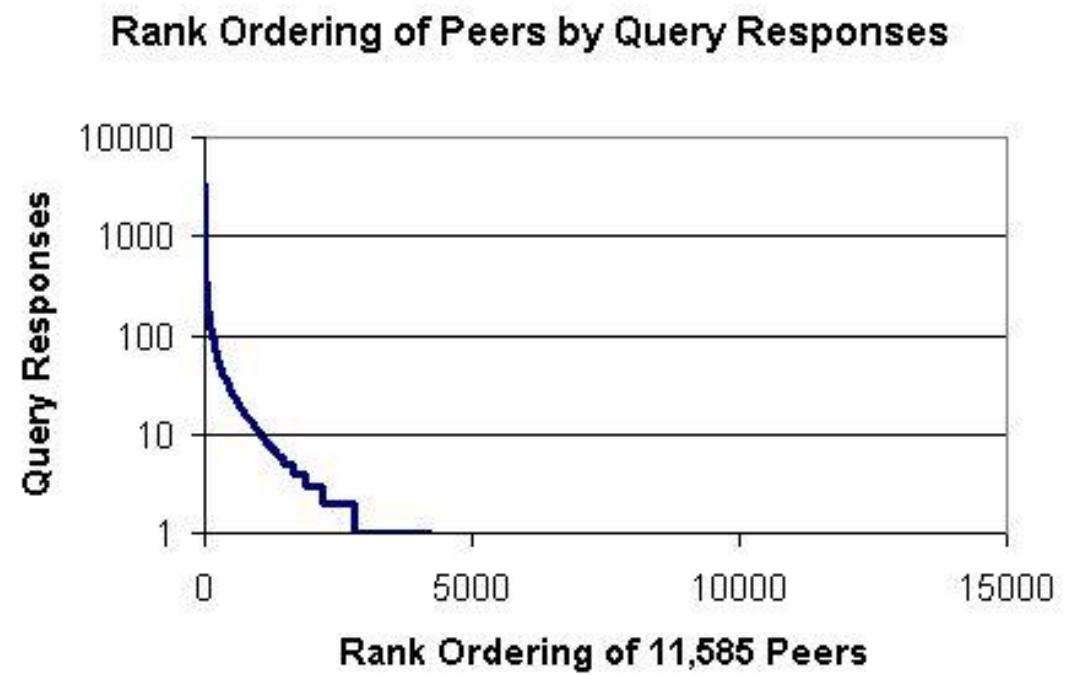
# Gnutella: Free-riding (1/2)

- Veliki postotak korisnika su “*free riders*”
  - 66% peerova ne nudi vlastite datoteke
  - 73% peerova nude 10 ili manje datoteka
  - 1% peerova nude 37% svih datoteka
  - 10% peerova nude 87% svih datoteka



# Gnutella: Free-riding (2/2)

- Veliki broj peerova nudi datoteke koje nikoga ne zanimaju
- od 11,585 peerova koji nude datoteke:
  - 1% peerova odgovara na 47% svih upita
  - 25% peerova odgovara na 98% svih upita
  - 63% peerova nikada ne odgovaraju na upite



# Strategije pretraživanja

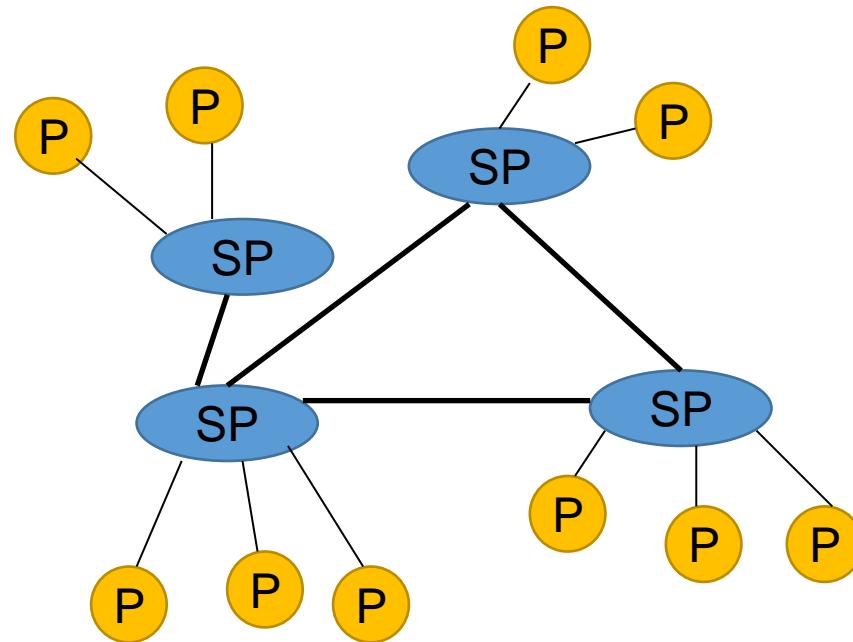
- Preplavljanje
  - Čvor koji ne može odgovoriti na upit prvo provjerava kod izvorišnog čvora treba li proslijediti upit dalje, izvorišni čvor postaje preopterećen
  - Ograničenje udaljenosti od izvorišnog čvora (TTL)
  - Strategija širećeg prstena: TTL je inicijalno mali, a poveća se samo u slučaju da prethodni upit ne vrati rezultat (dobra strategija)
- Slučajna šetnja
  - upit se usmjerava jednom slučajno odabranom čvoru (značajno kašnjenje)
  - pokrene se  $k$  paralelnih šetnji (manje kašnjenje), no svaki "upit-šetač" treba periodički provjeravati je li upit zadovoljen (ako je šetnja se prekida)

# Svojstva Gnutelle

- Robusna arhitektura
  - jednostavno održavanje mreže, jednostavno objavljivanje datoteka
  - otporna na značajne promjene mrežne topologije (*high churn*)
- Pogodno rješenje ako su datoteke često replicirane u mreži
- Neskalabilno rješenje u smislu generiranog prometa prilikom pretraživanja (strategija *k* paralelnih šetnji je bolja u smislu skalabilnosti od preplavljanja)
- Ne garantira pronalaženje datoteke

# Gnutella v6

- Posljednja verzija, uvodi posebne čvorove - *superpeer* (hijeracijska organizacija mreže)
  - *superpeer* je peer s dobrim resursima, na njega se spaja skup *peerova*
  - *peer* na *superpeerovima* sprema kopije vlastitih dokumenata te preko *superpeerova* vrši pretraživanje
  - upit se od *peera* šalje *superpeeru* koji prvo provjerava svoje lokalne podatke pa ako nema odgovor, preplavljuje upitom svoje susjedne *superpeerove*.

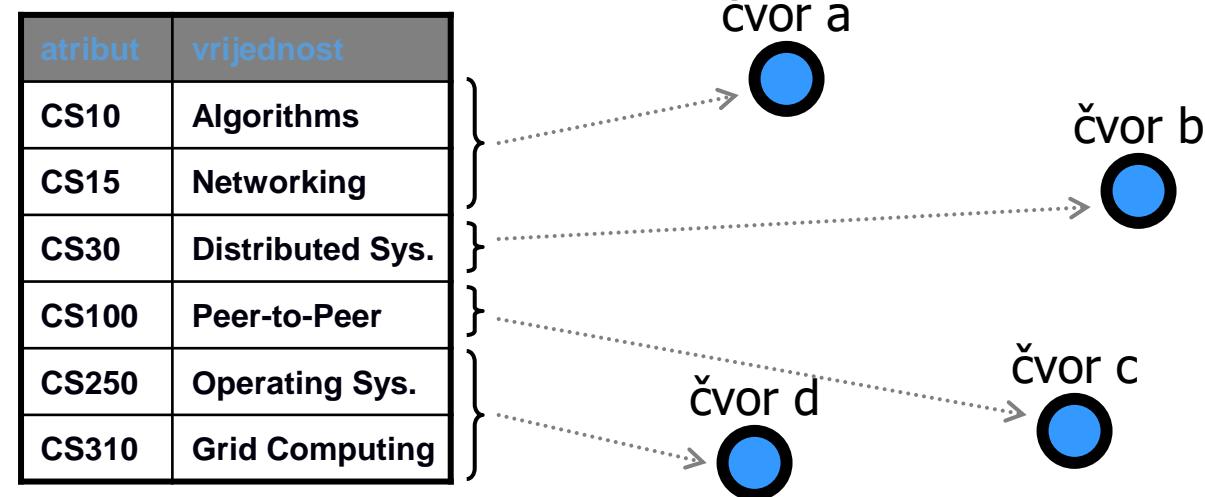


# Chord

Izvor: Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31, 4 (August 2001), 149–160. DOI: <https://doi.org/10.1145/964723.383071>

## Primjer strukturiranog sustava P2P

- koristi ideju raspodijeljene hash tablice - Distributed Hash Table (DHT)
- hash tablica je raspodijeljena na više čvorova.

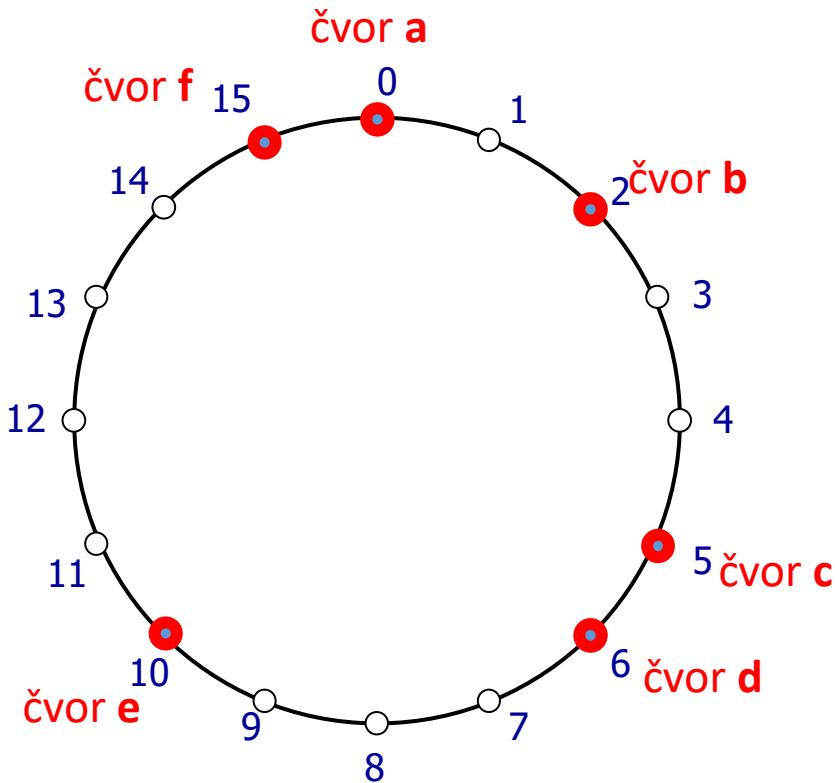


Svaki čvor  
pohranjuje dio  
hash tablice.

# Chord: organizacija prstena

- Koristi jednodimenzionalni prostor ključeva veličine  $2^m$ 
  - povezuje peerove i podatke iz hash tablice (atribut, vrijednost), vrijednost može biti bilo što, npr. podatak, dokument, objekt
  - prikazuje se kao prsten veličine  $N = 2^m$
- Implementira samo jednu operaciju  $\text{lookup}(k)$  koja vraća identifikator peera nadležnog za  $k$
- 2 hash funkcije:  $H_1(\text{peer\_ID})$  i  $H_2(\text{atribut}_x)$ ,  $\text{peer\_ID}$  je IP adresa peera
- Pretpostavka:  $m$  je dovoljno velik da postoji vrlo mala vjerojatnost kolizije za hash funkcije
  - Kolizija *hash* funkcije: proizvodi isti *hash* kod za različite parametre

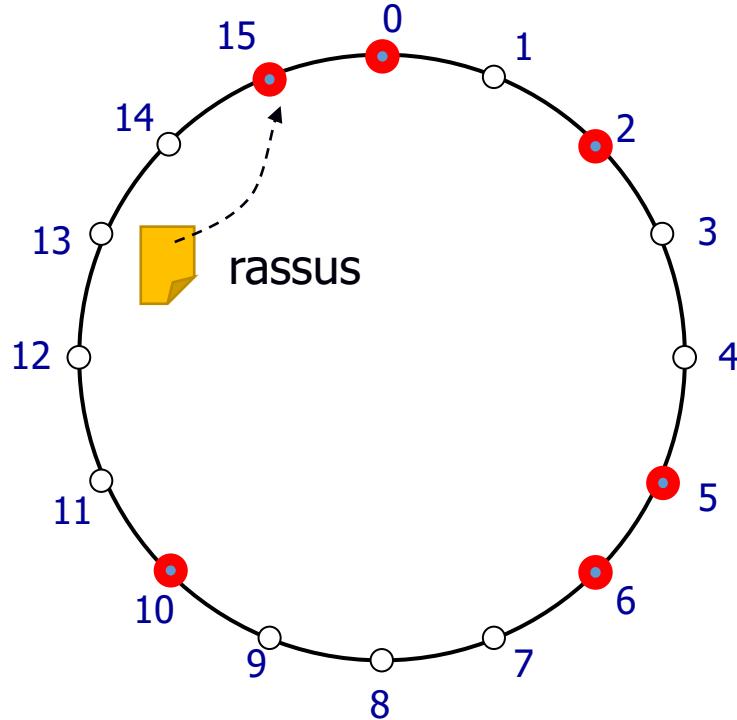
# Chord: Kako složiti čvorove u prsten?



Mogući identifikatori  
čvorova:  
 $\{0, 1, \dots, 15\}$ ,  $N = 16$

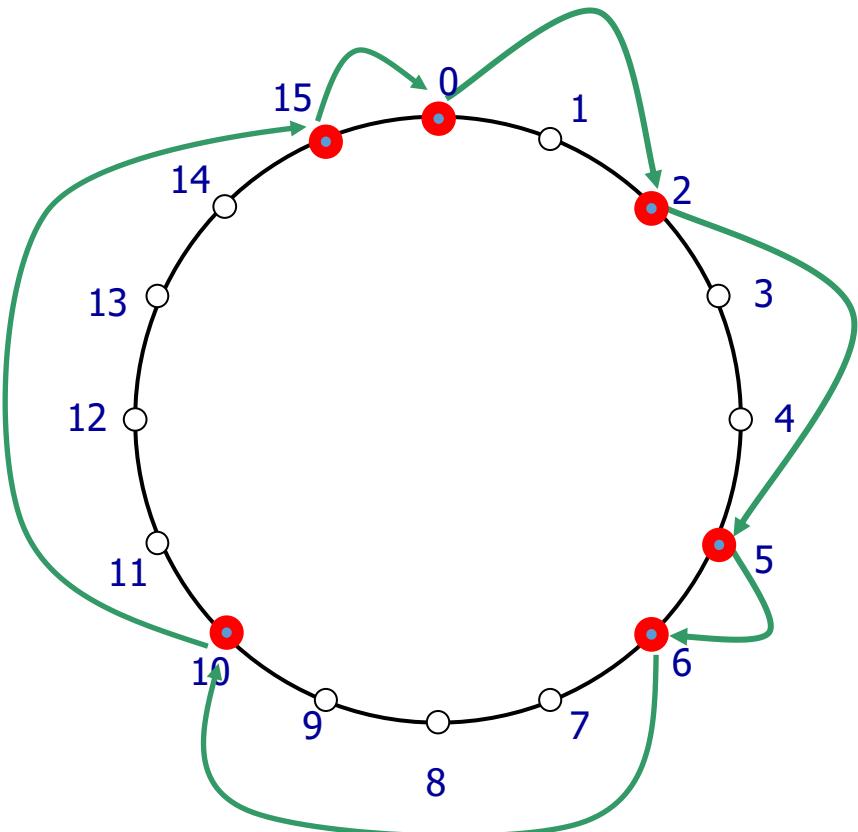
- Primjer mreže sa 6 čvorova {a,b,c,d,e,f}
- Prsten veličine  $N = 16$  ( $m = 4$ ), ovo je broj mogućih ključeva
- Čvorovima se jednoznačno pridjeljuju ključevi uz pomoć posebne funkcije  $H_1$ , npr.  $H_1(a) = 0$ .

# Chord: Kako se podatak dodjeljuje čvoru?



- Podacima se pridjeljuju ključevi iz istog prostora  $\{0, 1, \dots, 15\}$  koristeći funkciju  $H_2$
- (**rassus**, <http://www.fer.hr/predmet/rassus>) dobiva ključ 13 jer vrijedi  $H_2("rassus") = 13$
- Kako u mreži ne postoji čvor s ključem 13, podatak se pohranjuje na prvom sljedećem čvoru (to je u ovom slučaju čvor f kojemu je ključ = 15)

# Chord: Kako ćemo povezati čvorove?



- Svaki čvor održava jedan pokazivač na sljedbenika, tj. na prvi sljedeći čvor na prstenu u smjeru kazaljke na satu

sljedbenik čvora 0 → čvor 2

sljedbenik čvora 2 → čvor 5

sljedbenik čvora 5 → čvor 6

...

Učinkovito pretraživanje?

# Chord: Kako ubrzati pretraživanje?

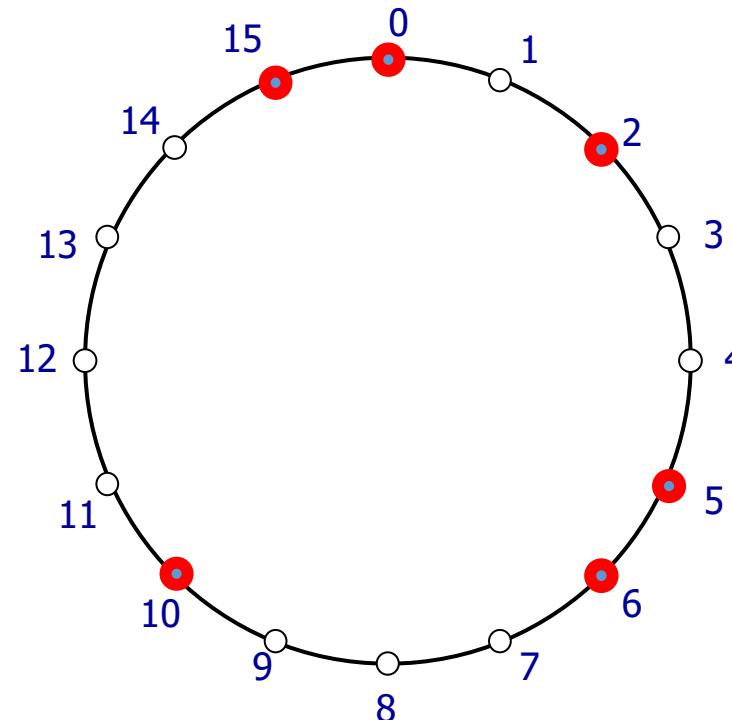
Primjer: tablica usmjeravanja za čvor s ključem 15

ključ +  $2^i$ ,  $i = 0, \dots, m-1$  ( $N = 2^m$ )

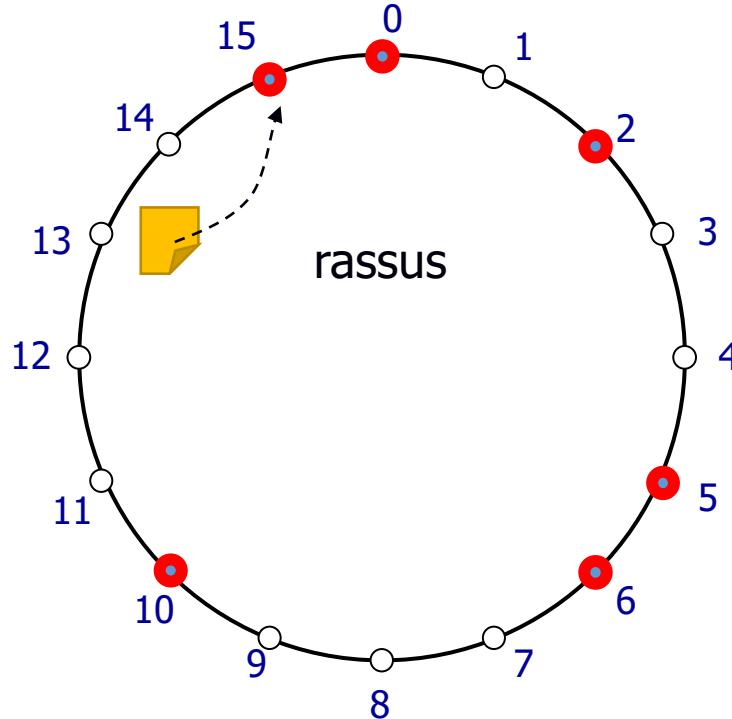
|            |           |
|------------|-----------|
| $15+2^0=0$ | → čvor 0  |
| $15+2^1=1$ | → čvor 2  |
| $15+2^2=3$ | → čvor 5  |
| $15+2^3=7$ | → čvor 10 |

Za čvorove koji ne postoje,  
pokazivač se postavlja na prvog  
sljedbenika!

Koliko je zapisa u tablici  
usmjeravanja svakog čvora?

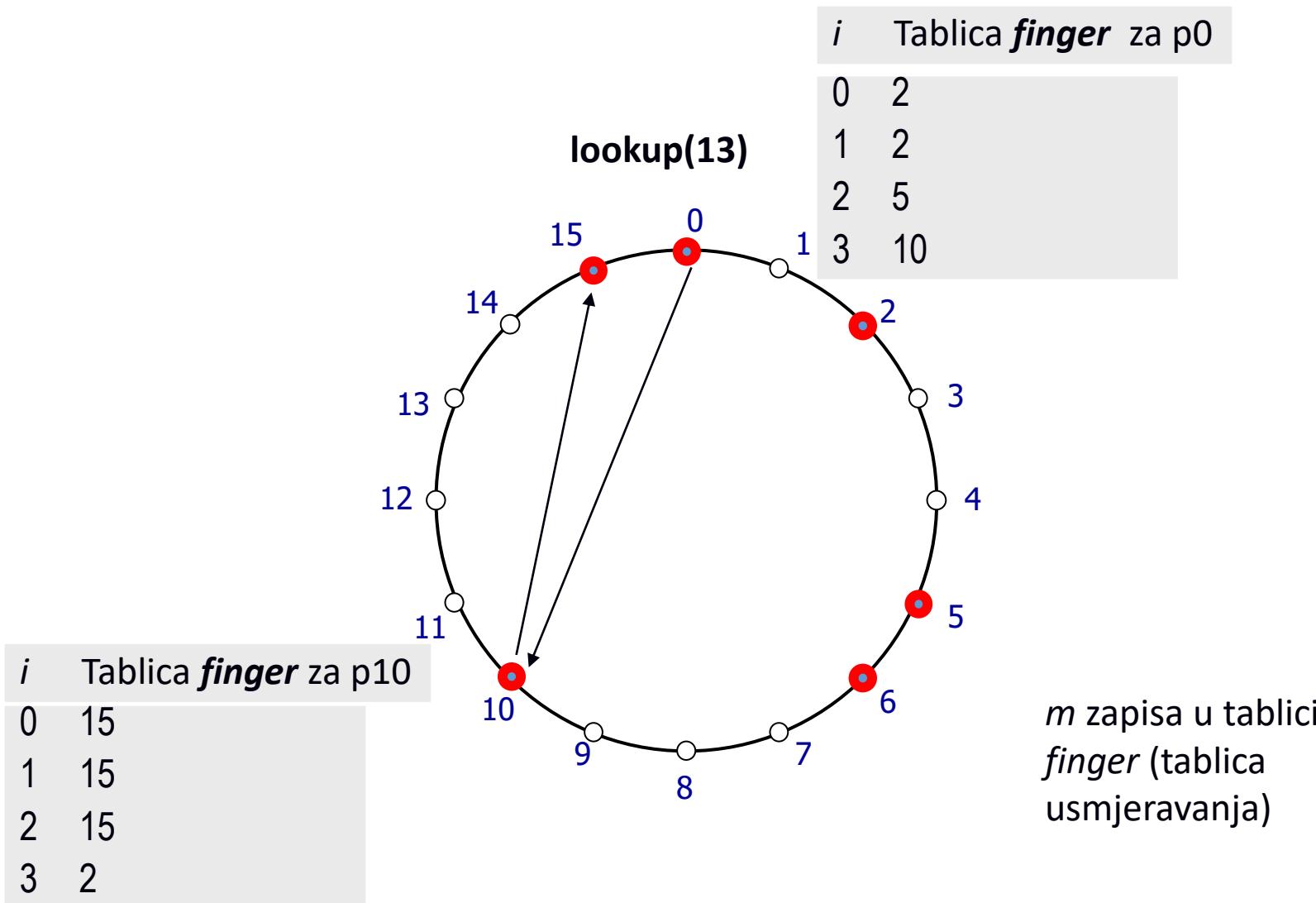


# Chord: Kako pronaći podatak? (1)

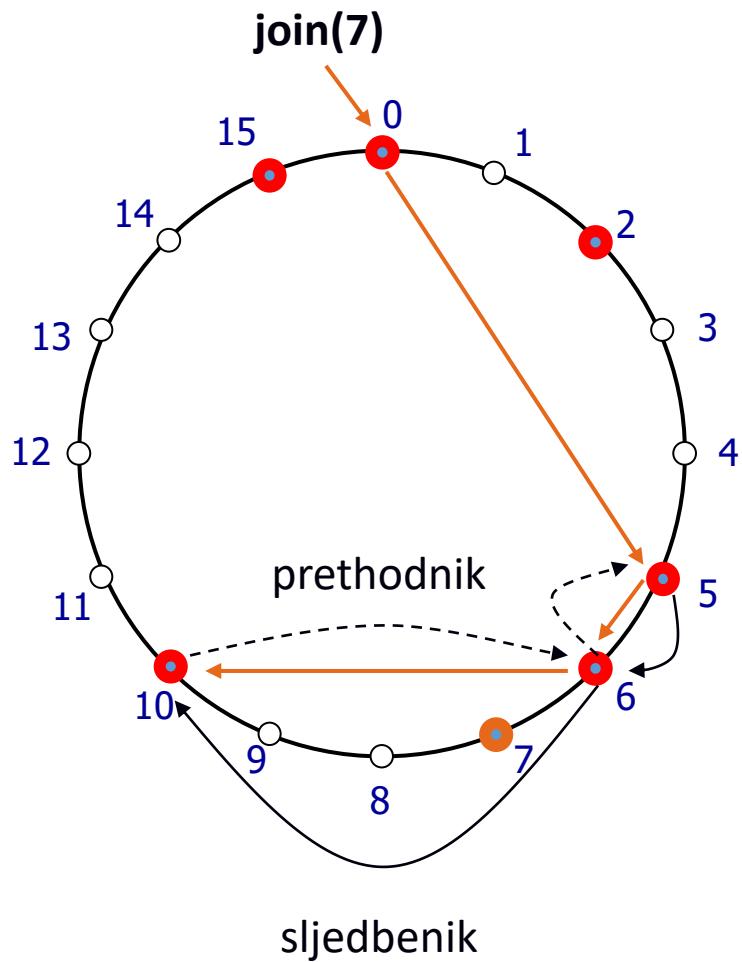


- **lookup(13) ?**
- jer svaki peer može izračunati  $H_2(\text{"rassus"}) = 13$
- vraća IP adresu peera zaduženog za "rassus", kontaktiramo peera direktno da dođemo do traženog URL-a

# Chord: Kako pronaći podatak? (2)

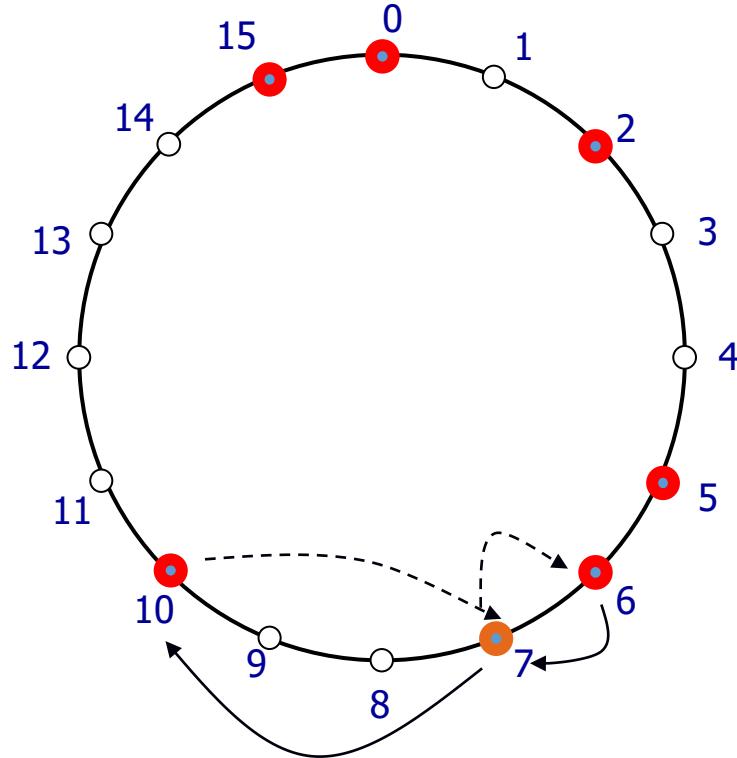


# Chord: dodavanje peera (1)



- Dodatak u tablici usmjerenja, svaki čvor ima pokazivač na prethodnika
1. korak: Čvor p7 pronalazi jednog peera iz mreže i koristeći lookup(7) pronalazi svog sljedbenika

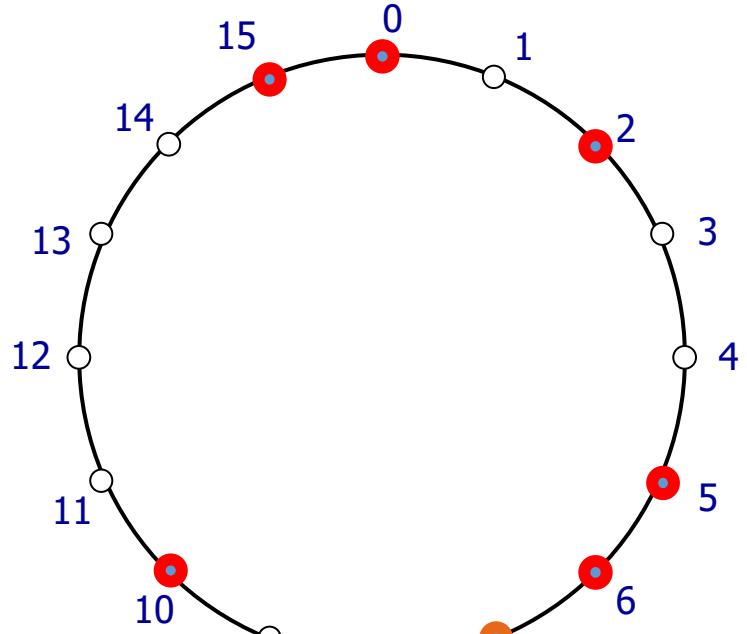
# Chord: dodavanje peera (2)



2. korak: Čvor p7 podešava pokazivač na sljedbenika (p10) te od njega saznaće svog prethodnika (p6)

3. korak: Čvor p10 mijenja prethodnika (p7), a čvor p6 sljedbenika

# Chord: dodavanje peera (3)

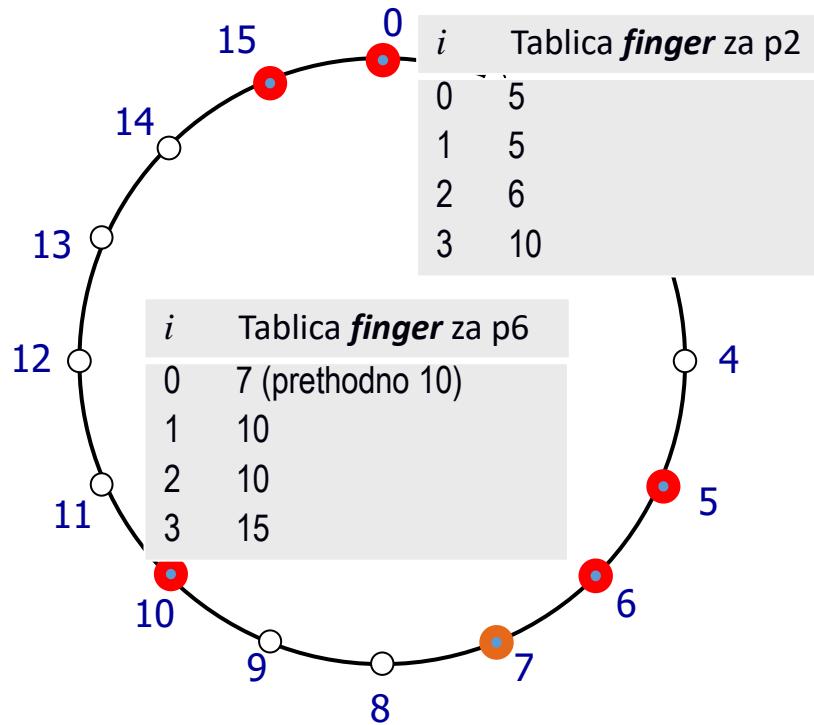


| <i>i</i> | Tablica <i>finger</i> za p7 |
|----------|-----------------------------|
| 0        | 10                          |
| 1        | 10                          |
| 2        | 15                          |
| 3        | 15                          |

4. korak: Podesi tablicu *finger* na p7, za to p7 koristi svog prethodnika (p6) koji izvodi sljedeće operacije:

| <i>i</i> |             |
|----------|-------------|
| 0        | lookup(7+1) |
| 1        | lookup(7+2) |
| 2        | lookup(7+4) |
| 3        | lookup(7+8) |

# Chord: dodavanje peera (4)



- Za  $i=0$ ,  $\text{lookup}(7-1)$  vraća 6, u tablici na  $p_6$  mijenja se prvi redak u  $i=0$ , 7
- Za  $i=2$ ,  $\text{lookup}(7-4)$  vraća 5 što je veće od 3 pa se vraćamo na prethodnika od  $p_5$ , a to je  $p_2$ ; u tablici na  $p_2$  se treći redak ( $i=2$ ) ne mijenja jer je trenutno 6 što je manje od 7

5. korak: Popravi tablice *finger* na sljedećim peerovima u mreži:  $p_7 - 2^i$  (ako postoje) ili na prethodniku tog čvora

Algoritam.

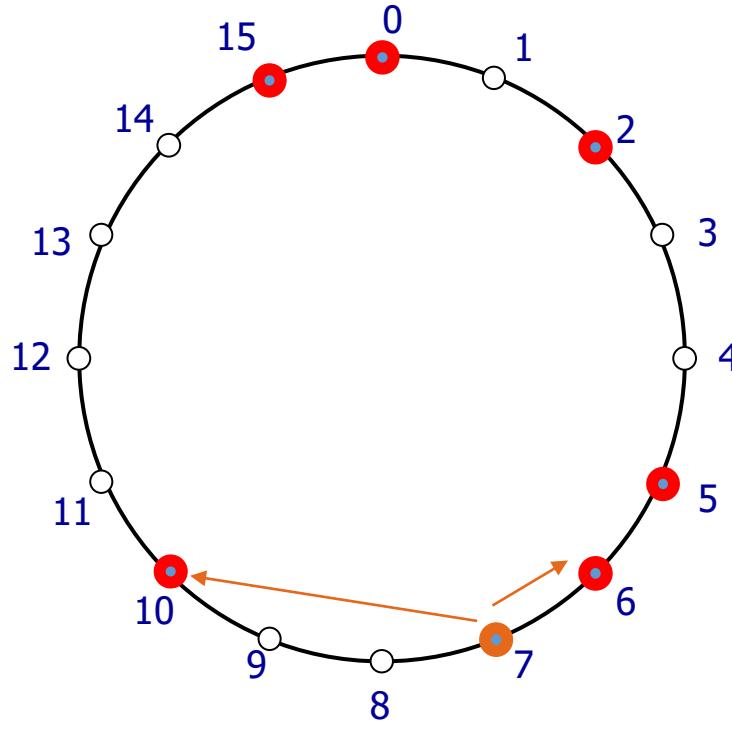
za  $i=0$  do 3

**lookup** ( $p_7 - 2^i$ ) ;

    ako si došao do čvora većeg od  $p_7 - 2^i$   
    vrati se do njegovog prethodnika (peer t)  
    inače peer t =  $p_7 - 2^i$ ;

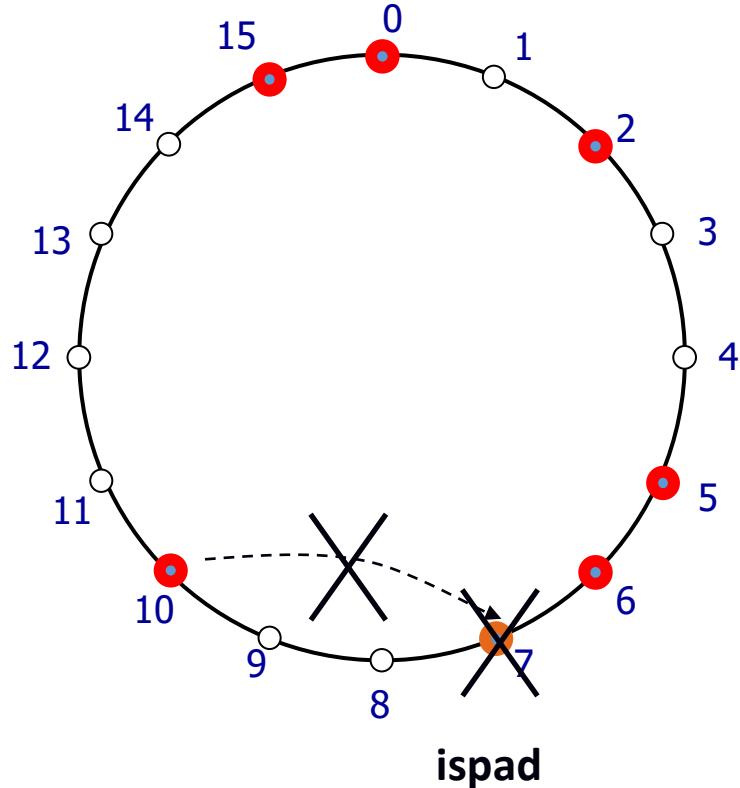
    ako je element iz  $i$ -tog retka peer-a t  
    veći od  $p_7$ , promijeni  $i$ -ti redak u tablici  
    usmjeravanja na 7

# Chord: izlazak peera



1. korak: p7 obavještava svog sljedbenika i prethodnika o napuštanju mreže
2. korak: p7 šalje zahtjev ostalim peerovima u mreži da poprave tablice *finger* (slično postupku dodavanja čvora)

# Chord: ispad peera



- Svaki čvor periodički kontaktira prethodnika i primjećuje njegov ispad
- Taj čvor može popraviti pokazivač na prethodnika samo ako ga drugi čvor dodaje kao svoga sljedbenik-a
- Svaki čvor održava listu sljedbenika, a ne samo jednog, ako prvi sljedbenik ne odgovara, čvor može kontaktirati sljedećeg na listi

# Svojstva Chorda

- Ravnomjerno opterećenje čvorova
  - Za mrežu od  $n$  čvorova, svaki čvor je zadužen za najviše  $(1+\varepsilon)N/n$  ključeva, gdje je  $N$  veličina adresnog prostora ključeva
- Skalabilnost algoritma *lookup*
  - kontaktira  $O(\log_2 n)$  čvorova
- Skalabilna veličina tablice *finger* :  $m$
- Nužno je određeno vrijeme stabilizacije za slučaj velikih promjena u mreži radi održavanja prstenaste strukture

# Usporedba protokola Chord i Gnutella

## Ulagni parametri simulacije

- Chord
  - veličina adresnog prostora =  $2^{160}$
  - broj sljedbenika  $r = 10$
  - za stabilizaciju prstena: 10s za ažuriranje tablice *finger*, 10s za stabilizaciju stanja prethodnika
- Gnutella
  - TTL za PING = 3
  - TTL za QUERY = 5

## Izlazni parametri simulacije

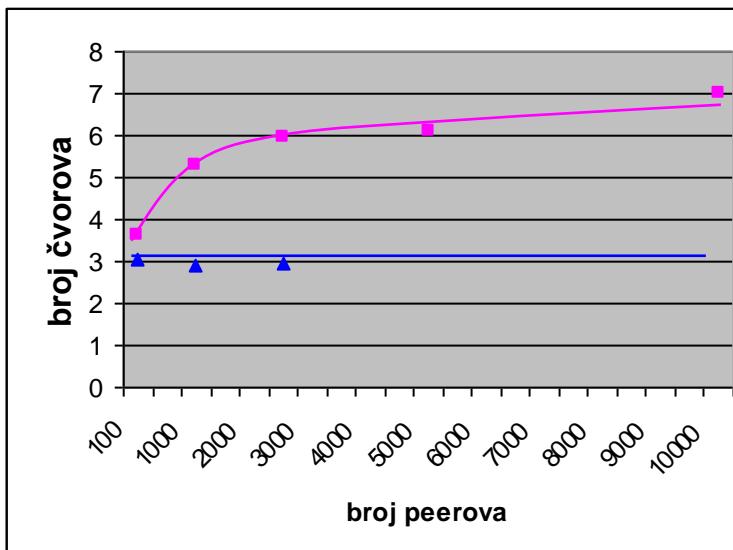
- Prosječan broj posjećenih čvorova po upitu
- Prosječno vrijeme odziva
- Uspješnost (postotak riješenih upita)

Simulator: PeerfactSim.KOM.

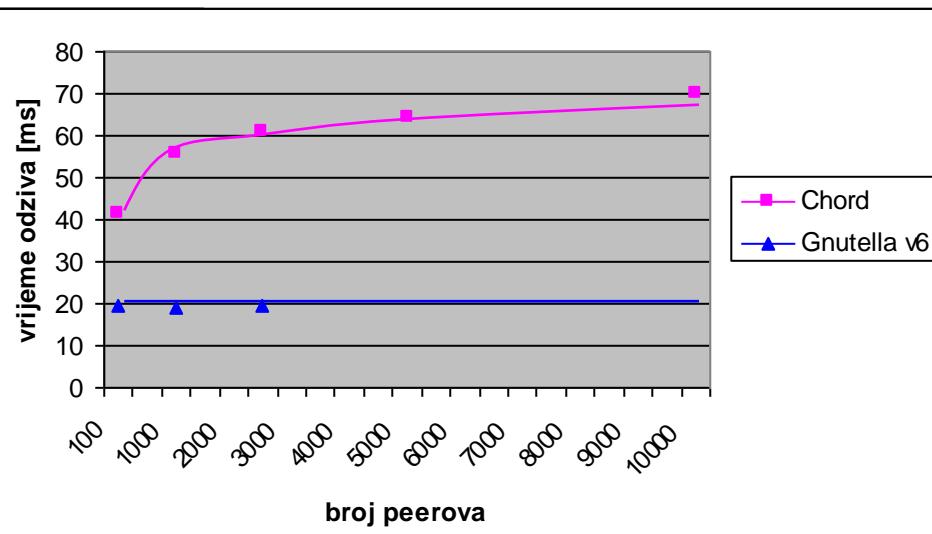
<http://peerfact.kom.e-technik.tu-darmstadt.de/de/downloads/>

# Analiza skalabilnosti (1)

- Statični scenarij
  - inicijalno se formira mreža nakon koje slijedi proces stabilizacije (značajno za Chord) te potom počinje objavljivanje podataka i pretraživanje

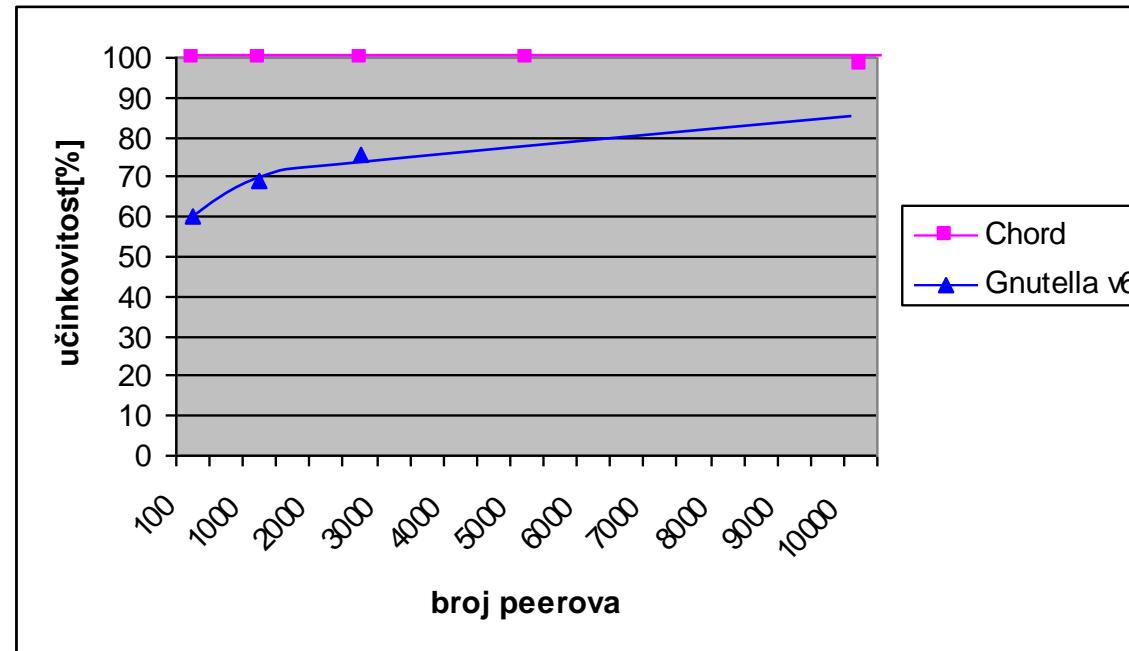


Prosječan broj čvorova po upitu



Prosječno vrijeme odziva

# Analiza skalabilnosti (2)



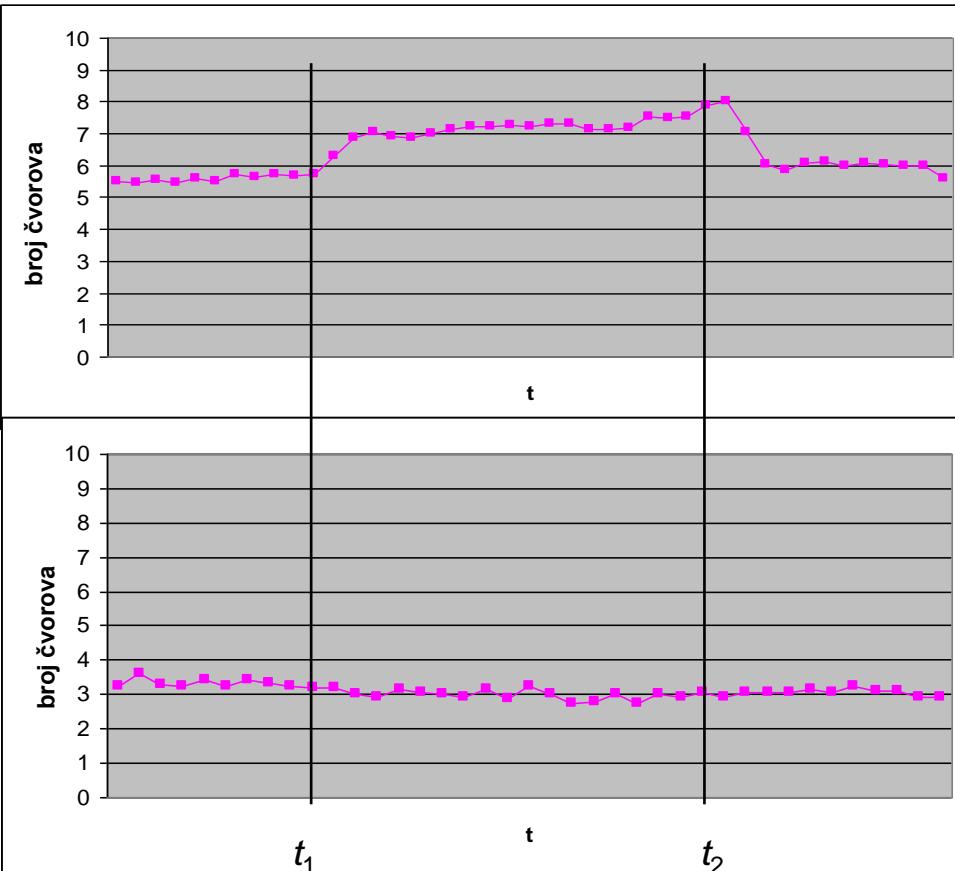
postotak riješenih upita

# Analiza stabilnosti (1)

- Scenarij *churn*

- u mreži je inicijalno 1000 čvorova, u  $t_1$  se dodatno spaja 2000 čvorova, a u  $t_2$  odspaja 2000 čvorova

Chord

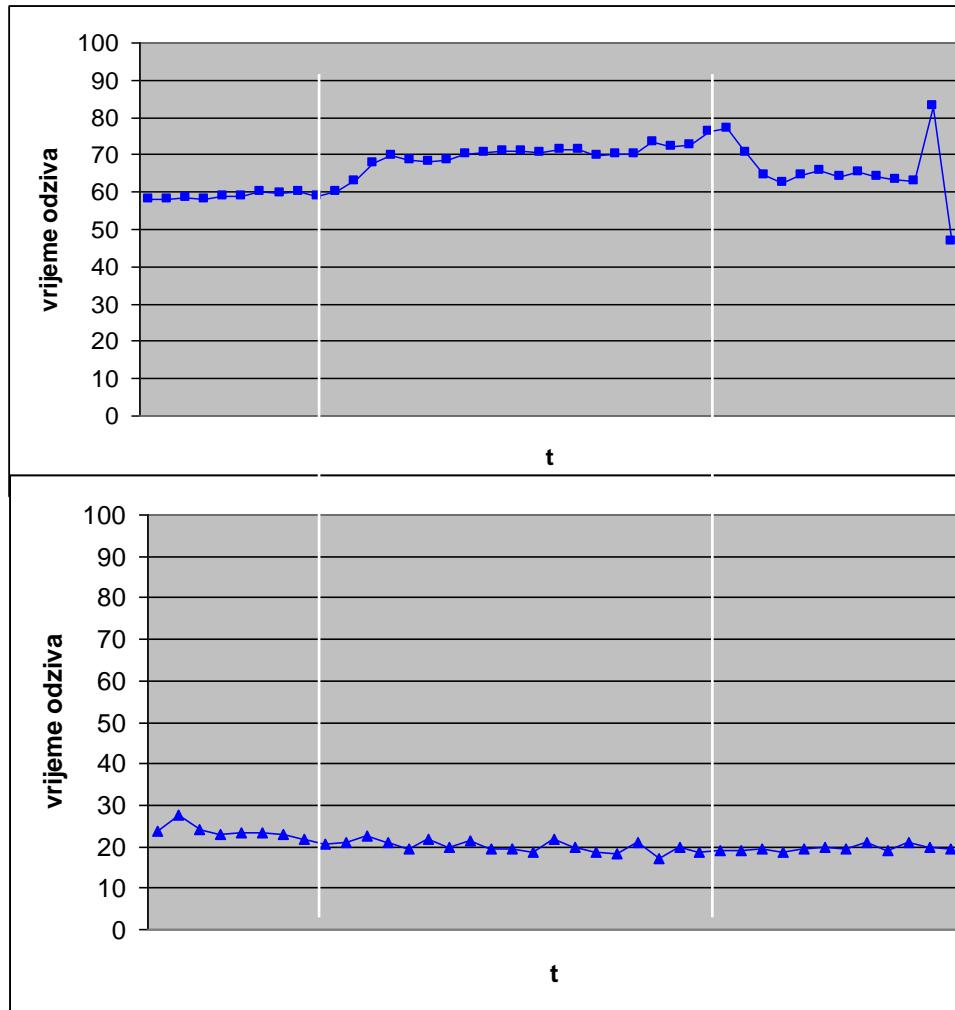


Promjena prosječnog  
broja čvorova po upitu u  
vremenu

Gnutella

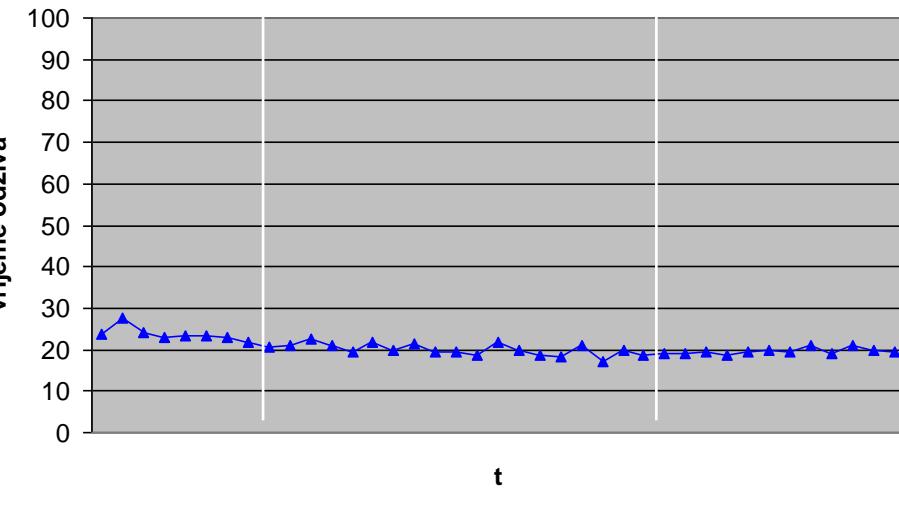
# Analiza stabilnosti (2)

Chord



Gnutella

Promjena  
prosječnog  
vremena odziva  
u vremenu



# Zaključak

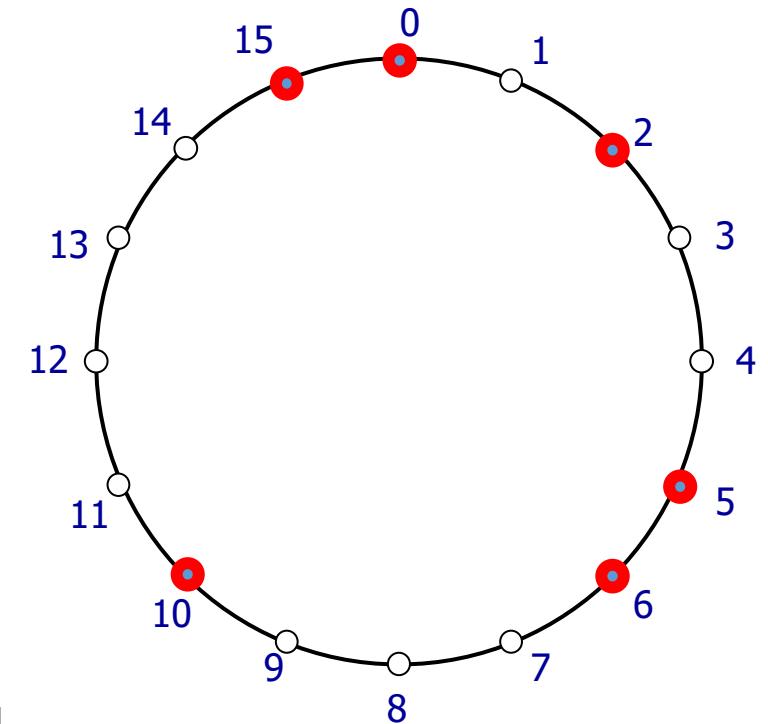
- Chord bilježi gotovo 100% uspješnost pri rješavanju upita
- Gnutella ima nižu učinkovitost koja raste za veće mreže zbog bolje povezanosti čvorova (povećana vjerojatnost pronađenja dokumenta)
- Prosječan broj čvorova po upitu te vrijeme odziva su konstantni za Gnutellu za povećani broj čvorova (TTL je konstantan), dok za Chord bilježimo logaritamski porast
- Chord je izrazito nestabilan prilikom scenarija churn zbog procesa stabilizacije koji nije završen (uspješnost je smanjena na 55%) dok je Gnutella neosjetljiva na churn (osim što je nešto smanjena uspješnost)

# Literatura

- G. Coulouris, J. Dollimore, T. Kindberg, G. Blair: "Distributed Systems: Concepts and Design", poglavlje 10.1-10.4
- Eng Keong Lua Crowcroft, J. Pias, M. Sharma, R. Lim, S., A survey and comparison of peer-to-peer overlay network schemes, IEEE Communications Surveys & Tutorials, 7(2), Second Quarter 2005, pp. 72- 93.  
<http://www.cl.cam.ac.uk/teaching/2005/AdvSysTop/survey.pdf>
- Karl Aberer: Peer-to-Peer Data Management. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2011

# Pitanja za učenje i ponavljanje

- Za Chordov prsten sa slike navedite tablicu usmjerenja za čvor 5.
- Navedite korake koji će pohraniti podatak  $d$  za koji vrijedi  $H(d)=13$  ako podatak želi pohraniti čvor čiji je ključ jednak 2. Na kome čvoru će podatak biti pohranjen?
- Navedite korake za pronalaženje podatka  $d$  iz prethodnog pitanja ako upit za  $d$  dolazi s čvora s ključem 0. Koja je cijena u smislu komunikacije među peerovima?





SVEUČILIŠTE U ZAGREBU



Fakultet  
elektrotehnike i  
računarstva

**Diplomski studij**

**Informacijska i  
komunikacijska tehnologija:**

Telekomunikacije i informatika

**Računarstvo:**

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

# Raspodijeljeni sustavi

**13. Tehnologija raspodijeljene glavne  
knjige  
(engl. *Distributed Ledger Technology*, DLT)**

Ak. god. 2020./2021.

# Creative Commons



- slobodno smijete:

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



- pod sljedećim uvjetima:

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

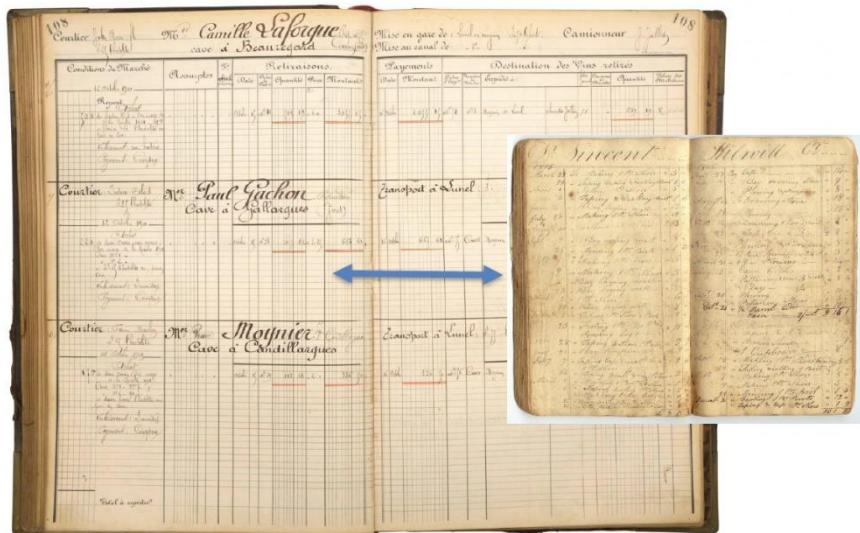
*Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.*

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

# Sadržaj

- Raspodijeljena glavna knjiga: motivacija i pojmovi
- Blockchain: struktura podataka
- Blockchain: centralizirana i decentralizirana izvedba
- Konsenzus
- Bitcoin i druge primjene
- Ethereum i pametni ugovor

# Motivacija za nastanak raspodijeljene glavne (javne) knjige



Ledger: računovodstvena knjiga

Distributed Ledger?



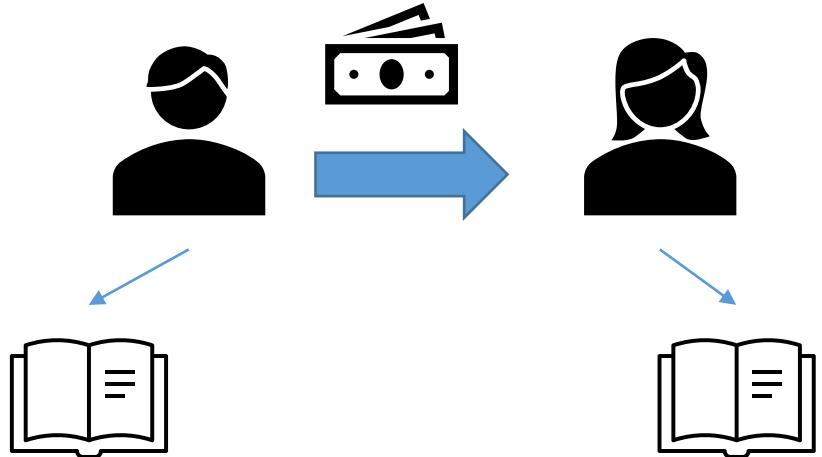
Bitcoin  
Whitepaper  
2008 [4]

Bitcoin, digitalna valuta  
kriptovaluta

2009

# Razmjena digitalnih dobara

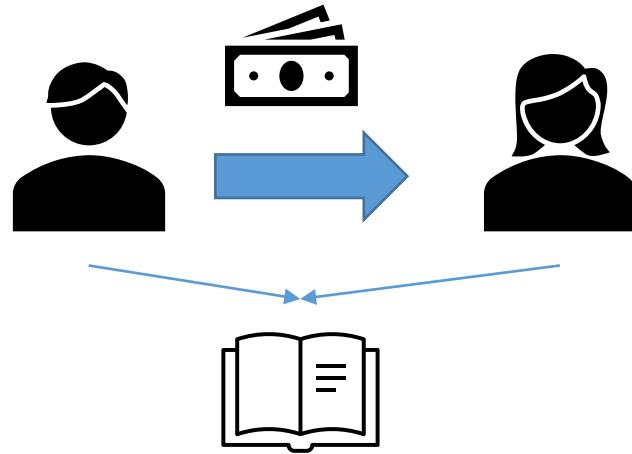
Klasično: svaka strana čuva listu transakcija



Dogovor o provedenoj transakciji  
Moguće pogreške

**Ne postoji povjerenje  
među sudionicima**

DLT



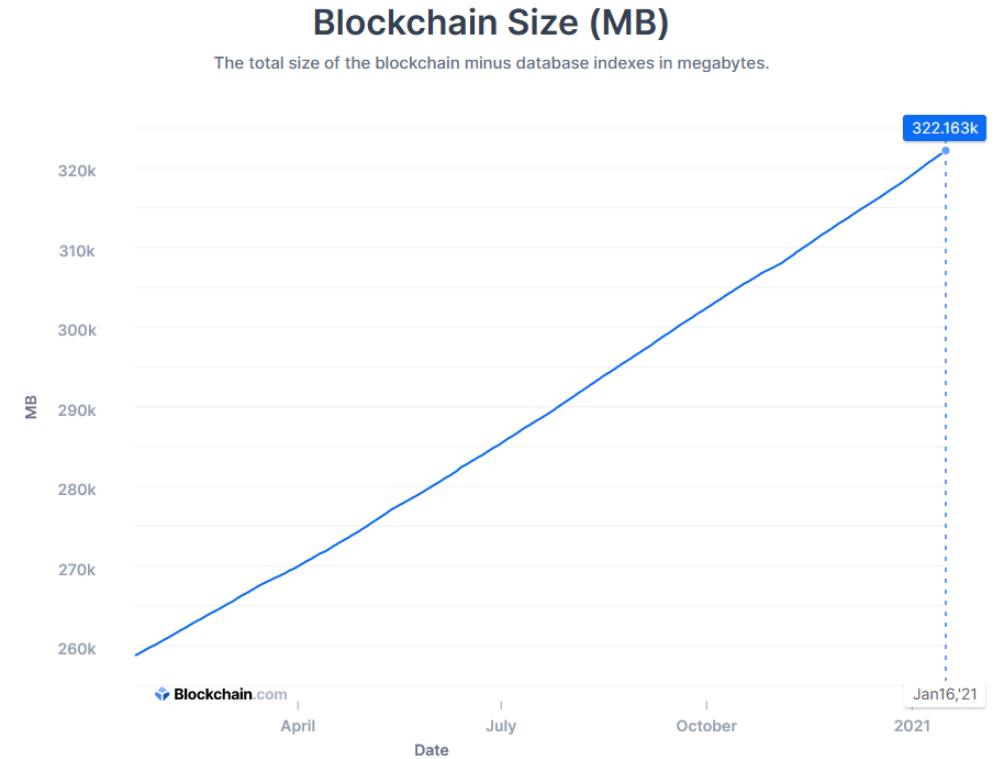
Zajednička dijeljena glavna knjiga  
Zapisi su nepromjenjivi i trajni

# Raspodijeljena glavna knjiga

- Nova vrsta „pouzdanog posrednika“ koja se temelji na raspodijeljenim algoritmima i kriptografiji
- Omogućuje izvođenje transakcija u raspodijeljenom okuženju s velikim brojem dionika koji si međusobno ne vjeruju (*intermediary in a trustless environment*)
- Na primjeru kriptovaluta: zamjena za banku
- Što ova tehnologija može, a ne mogu raspodijeljene baze podataka?

# Tehnologija raspodijeljene glavne knjige

- *Distributed Ledger Technology* (DLT)
- Sustav koji održava popis provedenih transakcija na decentralizirani način u mreži peerova
- Svaki peer održava kopiju svih transakcija
- Omogućuje jedino zapisivanje (konstantno dodavanje) novih transakcija, glavna knjiga kontinuirano raste
- Zapisane transakcije se ne mogu obrisati, nepromjenjive su i svi ih mogu čitati i provjeriti (javni ledger)



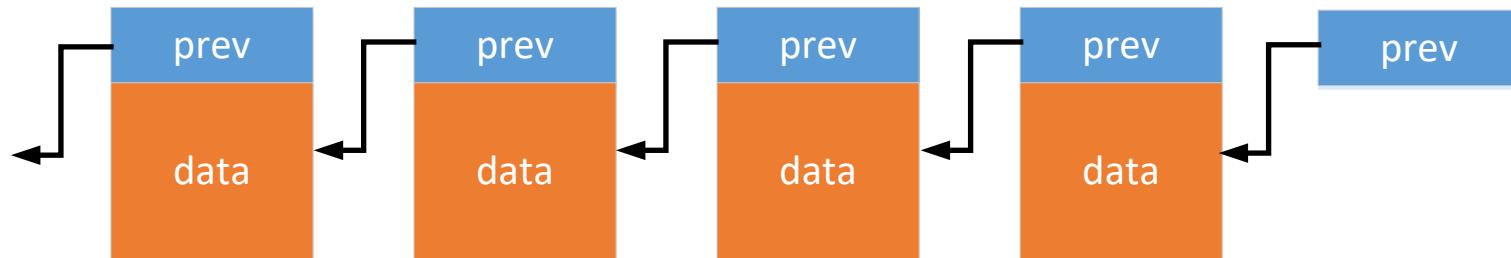
Size of the Bitcoin blockchain  
<https://www.blockchain.com/charts/blocks-size>

# Što je *blockchain*?

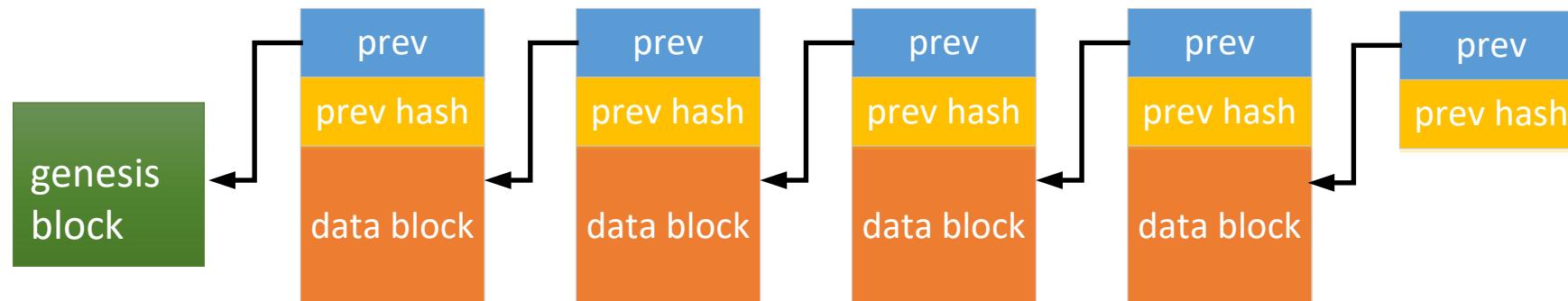
- Struktura podataka koja se sastoji od blokova (blok-lanac)
- Zapisnik (*append-only log*) u koji se **novi blokovi mogu dodavati**, ali se **stari ne mogu brisati ili naknadno mijenjati** (falsificirati)
- Blokovi su uređeni (u vremenu) i pohranjuju transakcije
  - postoji relativna uređenost blokova zapisa u vremenu
  - transakcije unutar bloka su također relativno uređene u vremenu
  - blokovi su otvoreni za čitanje (za public blockchain)
- Zapisuje se i prati **globalno stanje sustava**, no kako se peerovi mogu dogovoriti o uređenosti transakcija i blokova?

# Blok-lanac: struktura podataka

- slična jednostrukoj povezanoj listi

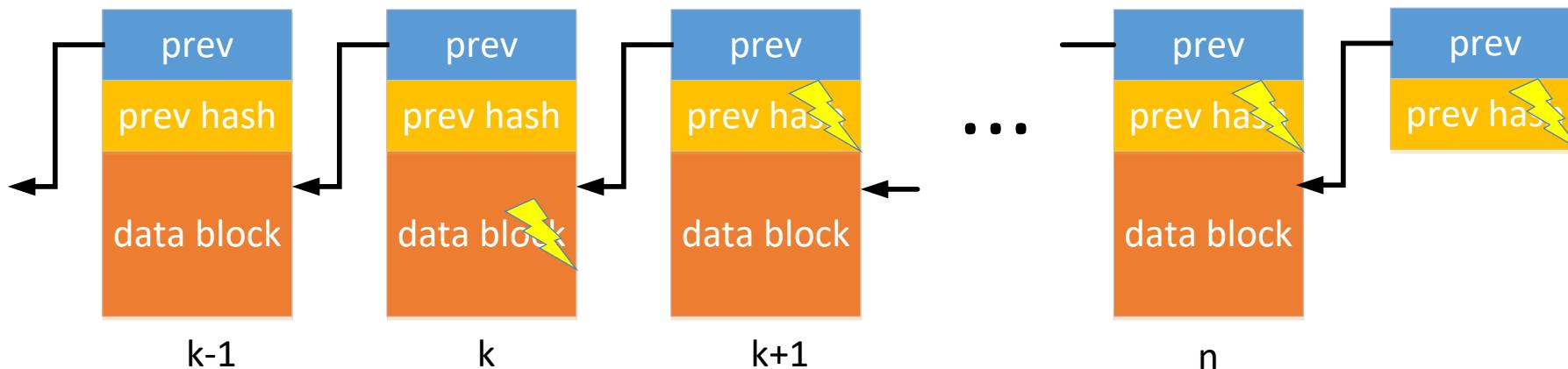


- U *blockchainu* svaki blok ima zaglavlje koje među ostalim metapodacima sadrži pokazivač na prethodni blok i sažetak prethodnog bloka (*hash*) + blok (sadrži transakcije)



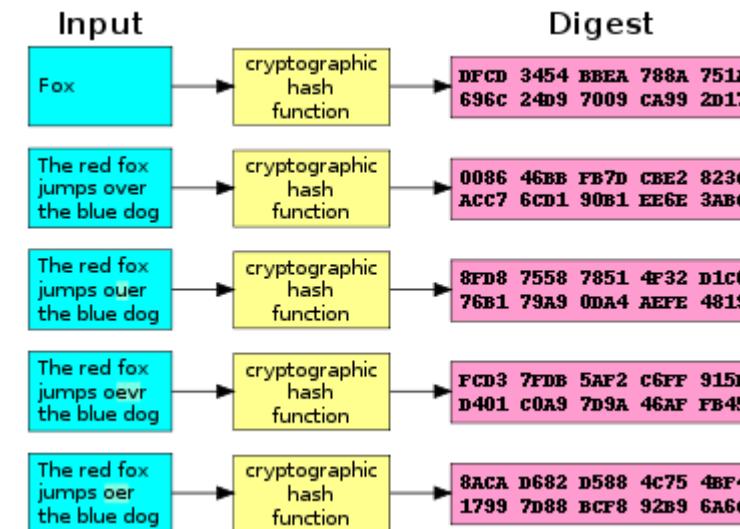
# Zašto se zapisi blok-lanca ne daju falsificirati?

- Tko god ima pohranjen ispravan sažetak (*hash*) prethodnog bloka može utvrditi da je došlo do promjene nekog bloka
  - Nakon izmjena zapisa u bloku  $k$ , njegov novi sažetak neće odgovarati starom sažetku koji je pohranjen u sljedećem bloku  $k+1$
  - Nakon toga niti novi sažetak bloka  $k+1$  neće odgovarati njegovom starom sažetku



# Može li falsificirani blok proći neopaženo?

- Ovo bi se moglo dogoditi kad bi **novi sažetak (*hash*) nekog bloka bio (namješten da bude) identičan njegovom starom sažetku**
- Za izračun sažetka se koriste posebne kriptografske jednosmjerne funkcije (npr. SHA-256) sa sljedećim svojstvima
  - ◆ ulaz: niz znakova proizvoljne duljine
  - ◆ izlaz: niz znakova fiksne duljine
  - ◆ jednostavno je izračunati sažetak za zadani ulaz
  - ◆ nije moguće na osnovu sažetka regenerirati ulaz
  - ◆ **nije moguće odrediti ulaz koji bi imao zadani sažetak**
  - ◆ neizvedivo da se pronađu dva različita ulaza s istim sažetkom
  - ◆ promjena jednog bita na ulazu rezultira potpuno drugačijim izlazom



Izvor:[https://en.wikipedia.org/wiki/Cryptographic\\_hash\\_function#/media/File:Cryptographic\\_Hash\\_Function.svg](https://en.wikipedia.org/wiki/Cryptographic_hash_function#/media/File:Cryptographic_Hash_Function.svg)

# Može li održavanje blok-lanca biti centralizirano?

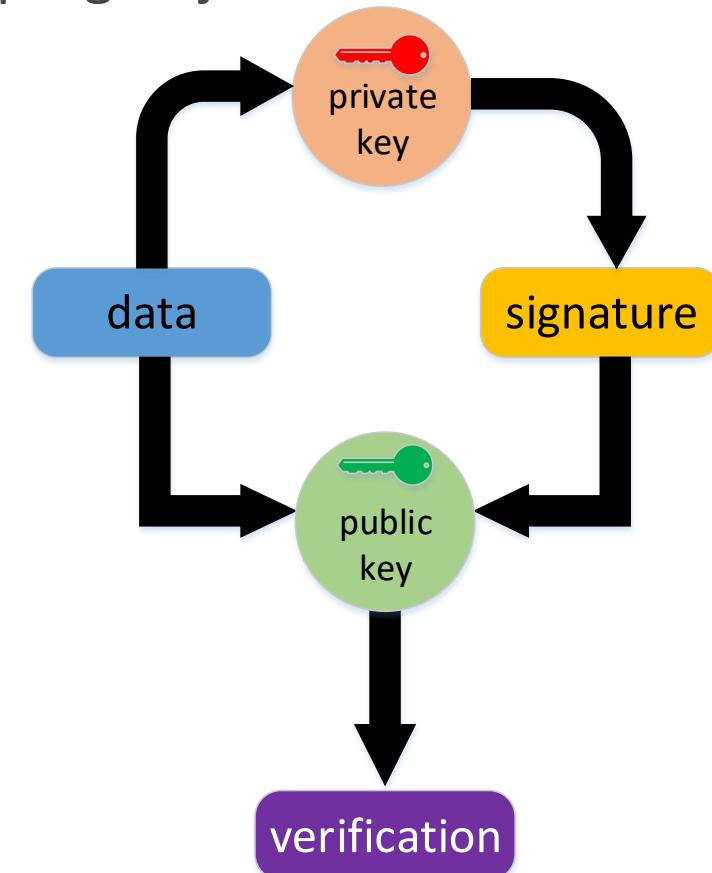
- Procedura centraliziranog pohranjivanja blok-lanca na poslužitelju (tj. centralnom autoritetu) je moguća
  - Transakcije koji se žele dodati u blok-lanca se moraju poslati centralnom autoritetu
  - Centralni autoritet provjerava valjanost primljene transakcije i zapisuje ih u nove blokove koje dodaje u blok-lanac (što je s redoslijedom transakcija?)
  - Centralni autoritet objavljuje nove blokove koje dodaje u blok-lanac
  - Svatko može provjeriti integritet blok-lanca: tko ima *prev\_hash* na neki objavljeni blok može provjeriti da nema naknadnih falsifikata u blokovima koji prethode tom bloku

# Koji su problemi centraliziranog održavanja blok-lanca?

- Centralni autoritet može **narušiti cjelovitost transakcije** (promijeniti primljenu transakciju prije pohrane)
  - U slučaju kriptovaluta centralni autoritet može proizvoljno mijenjati iznose
  - U slučaju papirnog novca to bi bilo identično krađi od strane banke prilikom transakcije
- **Problem cjelovitosti zapisa se rješava digitalnim potpisom!**

# Kako digitalni potpis čuva cjelovitosti zapisa?

- Digitalni potpis se temelji na primjeni asimetrične kriptografije
- Koristi tri algoritma
  - Algoritam za generiranje para ključeva (privatni i javni ključ)
  - Algoritam za digitalno potpisivanje:  
(podatak, privatni ključ) → potpis
  - Algoritam za provjeru potpisa:  
(podatak, potpis, javni ključ) → {T, ⊥}
- Ako se uz zapis u blok-lanac pohrani i njegov potpis
  - algoritmom za provjeru potpisa se lako utvrđuje je li podatak promijenjen ili nije
  - Bitcoin koristi javni ključ za identificiranje korisnika



# Koji su problemi centraliziranog blockchaina?

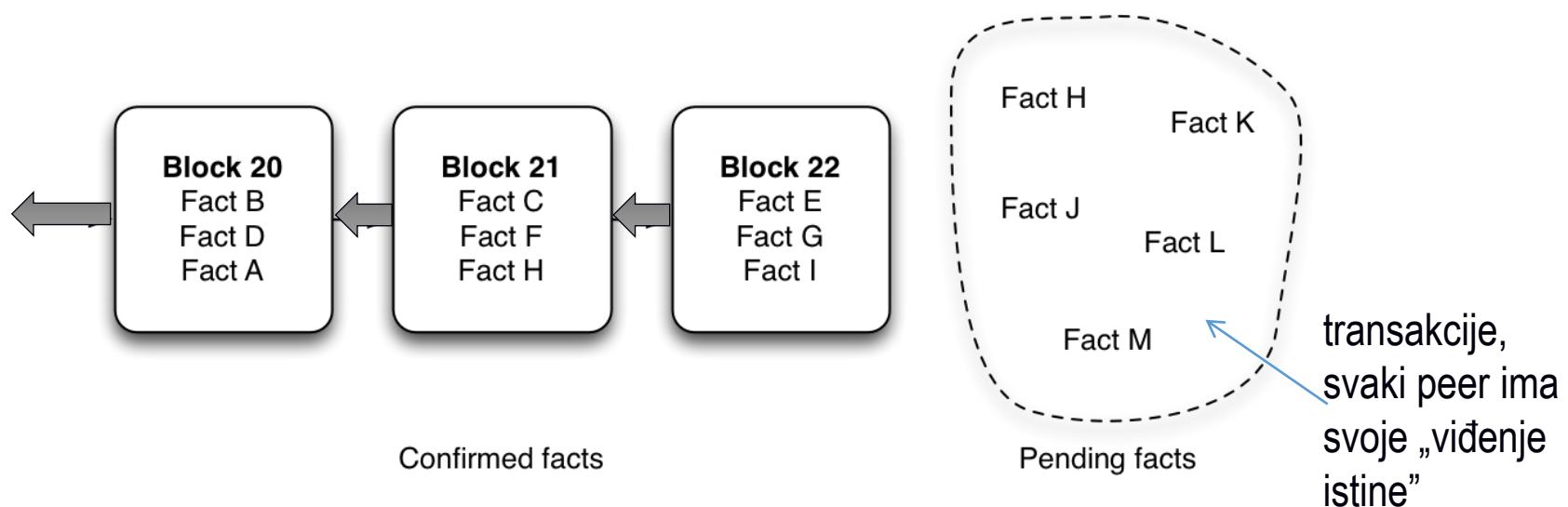
- Centralni autoritet može **uskratiti uslugu** (odbiti transakciju)
  - U slučaju kriptovaluta, centralni autoritet može odbiti provesti transakcije
  - **Problem selekcije zapisa koji se upisuju u blok-lanac se rješava raspodijeljenom implementacijom *blockchain-a***
- **Blockchain održava mreža čvorova (peerova)**
- **Svi čvorovi pohranjuju blok-lanac u potpunosti** (naravno uz sve probleme koje uvodi ovakva replikacija)

# Kako raspodijeljenost sprječava uskraćivanje usluge?

- U raspodijeljenom sustavu za provjeru i pohranu novog bloka može svaki put biti zadužen drugi čvor
- Dok god je većina čvorova dobromanjerna (pa ne vrši selekciju zapisa) postoji garancija da će pohrana transakcije biti provedena u budućnosti
  - Bit će provedena kad na red za provjeru i pohranu novog bloka dođe jedan od većine dobromanjernih čvorova
- Ako postoji puno zlonamjernih čvorova, pohrana zapisa može potrajati, što ruši povjerenje u čitav sustav raspodijeljenog održavanja blok-lanca

# Raspodijeljeno održavanje blok-lanca

- Blockchain održava mreža čvorova (peerova) koji istovremeno i generiraju transakcije (tj. primaju korisničke transakcije)
  - Kako će se peerovi dogovoriti o redoslijedu izvođenja transakcija (koja je transakcija generirana prije, a koja kasnije u P2P mreži)?



<https://marmelab.com/blog/2016/04/28/blockchain-for-web-developers-the-theory.html>

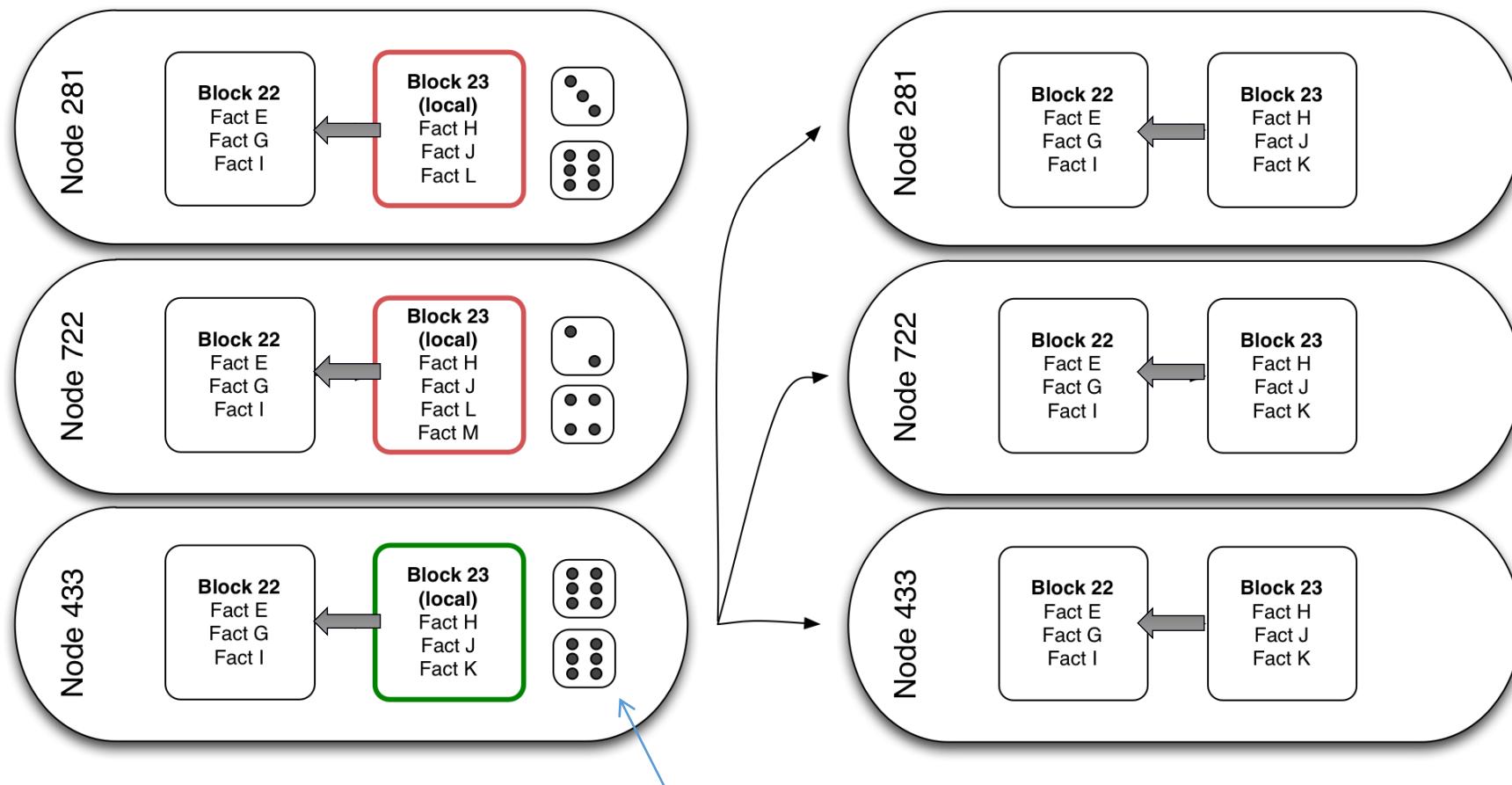
# Raspodijeljeni konsenzus

- Prilikom raspodijeljenog održavanja blok-lanca je potrebno postići dogovor (*konsenzus*) oko sljedećeg:
  1. koje transakcije se trebaju nalaziti u sljedećem bloku u lancu i
  2. koji je njihov ispravan redoslijed u tom bloku
- Postizanje raspodijeljenog *konsenzusa* je općenito složen problem (8. predavanje)
  - raspodijeljeni sustav čini mnogo čvorova od kojih neki mogu biti neispravni, a drugi u Bizantskom ispadu (zlonamjerni)
  - propagacija poruka u asinkronoj mreži: svaki čvor ima drugačiji pogled na slijed primljenih transakcija

# Raspodijeljeni konsenzus

- Pojednostavljeni algoritam koji se koristi za postizanje dogovora oko sljedećeg bloka u lancu
  1. Mreža se preplavljuje transakcijama
  2. Čvorovi prikupljaju ispravne transakcije i slažu ih u blokove
  3. Odabire se jedan čvor za dodavanje novog bloka u blok-lanac (onaj čvor koji riješi kriptografsku zagonetku)
  4. Odabrani čvor preplavljuje mrežu svojim blokom (mala je vjerojatnost da se istovremeno odabere više ovakvih čvorova, ali postoji)
  5. Ostali čvorovi prihvataju blok ako je ispravan i dodaju ga u svoj blok-lanac

# Čvorovi, blockchain i dodavanje novih blokova



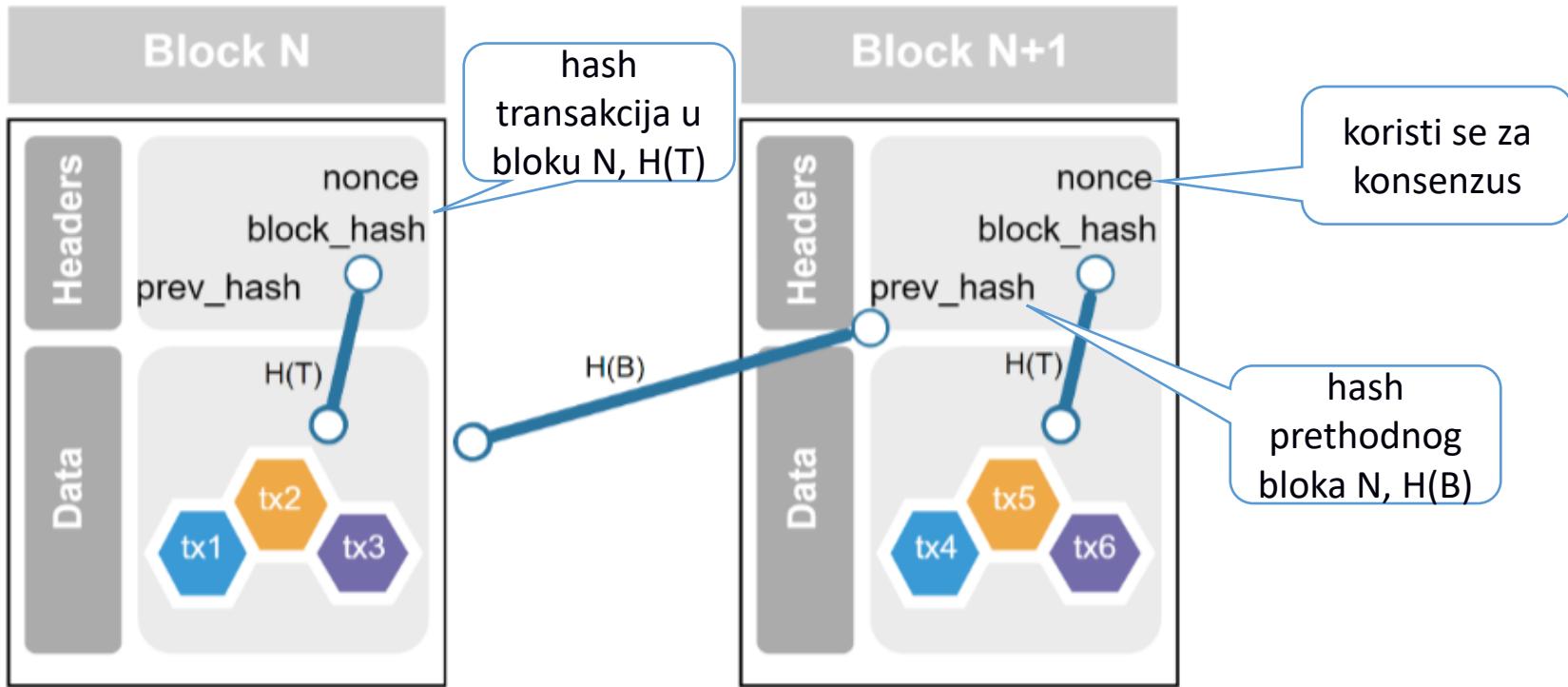
Odabir čvora? U ovom primjeru pobjeđuje onaj čvor koji baci dvije šestice

<https://marmelab.com/blog/2016/04/28/blockchain-for-web-developers-the-theory.html>

# Raspodijeljeni konsenzus: *Proof of Work*

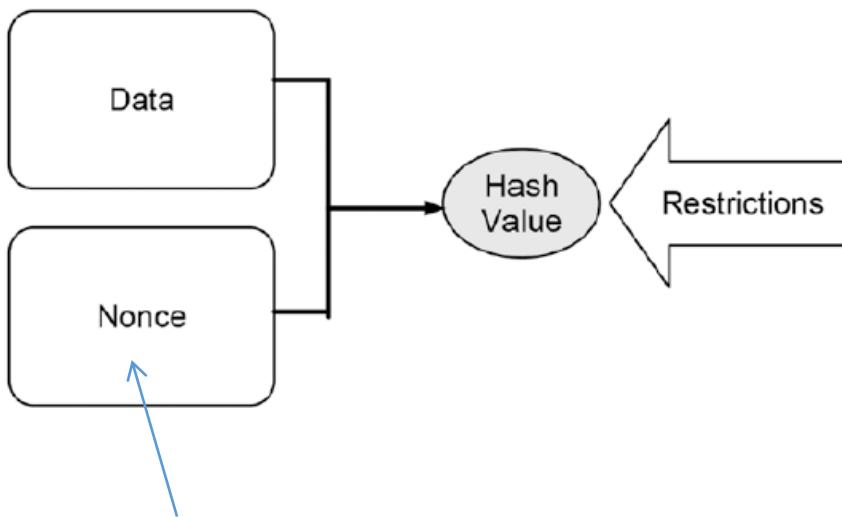
- U praksi pobjeđuje onaj čvor koji riješi kriptografsku zagonetku (tzv. „rudarenje“)
  - Računalno zahtjevan problem
    - kod kriptovalute Bitcoin je potrebno imati znatne procesorske resurse za njeno rješavanje (u prosjeku iz cijele mreže čvorova u 10 min jedan riješi zagonetku, rudarenje troši jako puno el. energije)
  - Težina zagonetke se periodički usklađuje
    - Kod kriptovalute Bitcoin se pokušava prosječno vrijeme do rješenja zagonetke (tj. objave sljedećeg bloka) držati na 10 minuta
    - Trivijalno je provjeriti je li zagonetka riješena
- Zašto bi čvor trošio resurse na rješavanje zagonetke?
  - rudarenje se nagrađuje u kriptovaluti

# Struktura podataka blok-lanca



Nepromjenjiva (engl. *immutable*) struktura podataka zbog korištenja kriptografskih funkcija

# Primjer rješavanja zagonetke (*Proof of Work*)



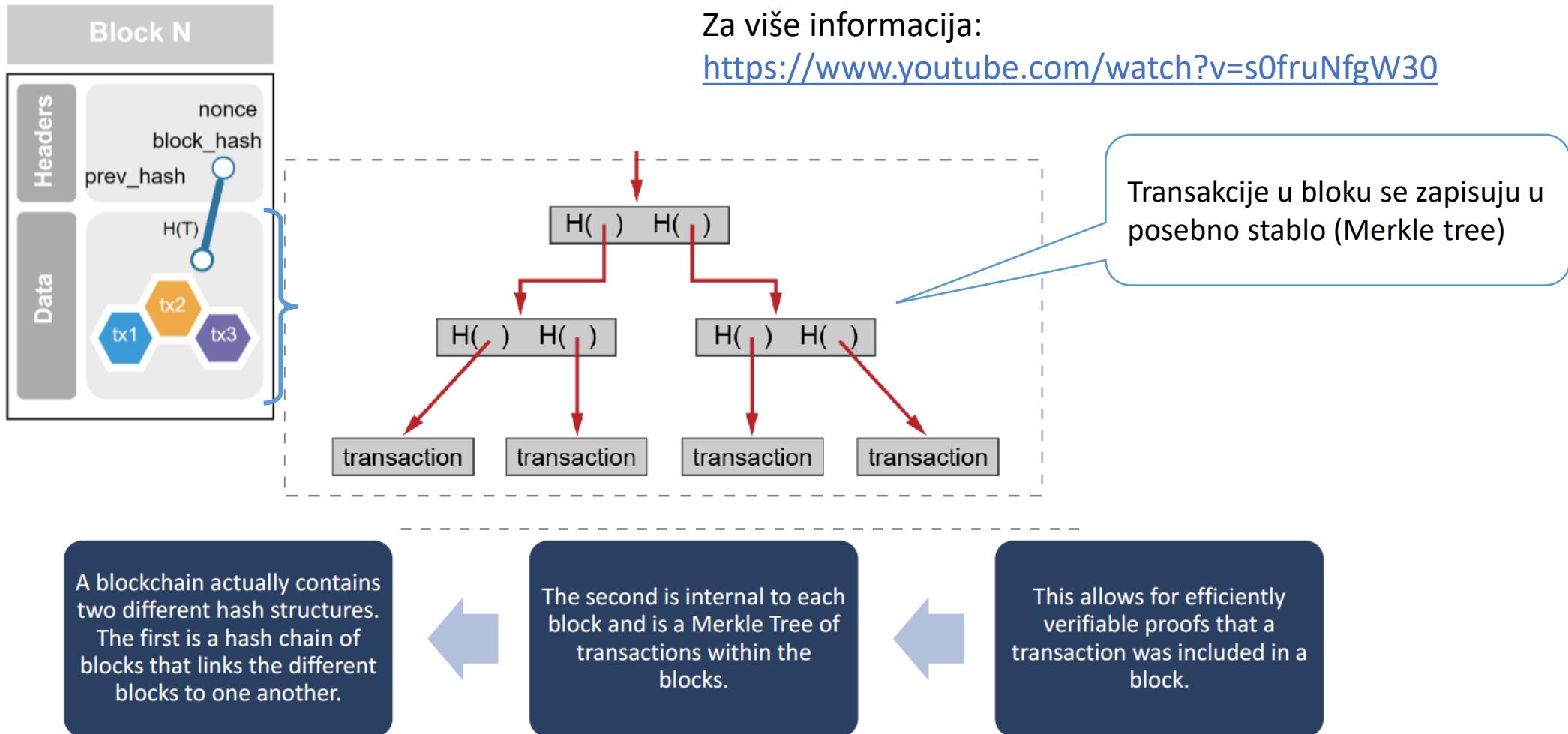
| Nonce | Text to Be Hashed | Output          |
|-------|-------------------|-----------------|
| 0     | Hello World! 0    | 4EE4B774        |
| 1     | Hello World! 1    | 3345B9A3        |
| 2     | Hello World! 2    | 72040842        |
| 3     | Hello World! 3    | 02307D5F        |
| ...   |                   |                 |
| 613   | Hello World! 613  | E861901E        |
| 614   | Hello World! 614  | <b>00068A3C</b> |
| 615   | Hello World! 615  | 5EB7483F        |

Traži se nonce koji uz zadalu poruku generira hash vrijednost koja počinje s npr. 3 nule

<http://www.blockchain-basics.com/HashPuzzle.html>

<https://demoblockchain.org/block>

# Kako se zapisuju transakcije u blok?



# Kako u se u načelu izvode transakcije?

Izvor [3]

- *Users interact with the blockchain via a pair of private/public keys. They use their private key to sign their own transactions, and they are addressable on the network via their public key.*
- *Every signed transaction is broadcasted by a user's node to its one-hop peers. The neighboring peers check this incoming transaction is valid before relaying it any further; invalid transactions are discarded.*
- *The transactions that have been collected and validated by the network using the process above during an agreed-upon time interval, are ordered and packaged into a timestamped candidate block. Nodes perform mining. The winning node broadcasts its block in the network.*
- *The nodes verify that the suggested block (a) contains valid transactions, and (b) references via hash the correct previous block on their chain. If yes, the block is added to the chain. If not, the proposed block is discarded. This marks the end of a round.*

# Primjer drugih algoritama za konsenzus

- *Proof of Stake*
  - Vjerojatnost da čvor bude izabran za rudarenje sljedećeg bloka ovisi o njegovom ulogu koji u vidu kriptovalute ulaže za taj posao
  - Planira se uvesti u Ethereum
- PBFT (*Practical Byzantine Fault Tolerance*)
  - Koristi se u privatnim DL rješenjima (npr. *Hyperledger*)
  - Čvor koji želi dodati blok postaje leader
  - Koristi three-phase commit protokol za glasanje uz prepostavku da je manje od trećine čvorova neispravno (potrebno je  $3f + 1$  ispravnih čvorova da bi se postigao sporazum)
- *Proof of Capacity, Proof of Authority, Proof of Burn, Proof of Elapsed Time*

# Raspodijeljenost?

Što je raspodijeljeno, podaci ili proces izvođenje transakcija?

- Blockchain je primjer ledgera koji se održava na raspodijeljeni način u mreži čvorova, ali **svaki čvor sadrži repliku podatkovne strukture sa svim transakcijama!**
  - Dakle, blockchain kao struktura podataka NIJE RASPODIJELJEN kao Distributed Hastable u P2P mrežama
  - Danas nastaju nove verzije raspodijeljenog ledgera
    - Primjer je IOTA koja se temelji na DAG-u (Directed Acyclic Graph) gdje svaki čvor predstavlja 1 transakciju

# Skalabilnost: značajan problem!

- Na dan 05.01.2018.
  - *186.951 pending transactions in the Bitcoin network and*
  - *around 22.473 pending in the Ethereum network*
- Mjeri se *transaction throughput*
  - ne povećava se s povećanjem broja čvorova
- Bitcoin: novi blok se rudari u prosjeku svakih 10 min uz veličinu bloka od 1 MB
  - 3 do 7 transakcija u sekundi
- Ethereum: novi blok se rudari u prosjeku svakih 15 s uz varijabilnu veličinu bloka
  - 7 do 15 transakcija u sekundi

# Kako kriptovalute koriste raspodijeljeni *ledger*?

<https://coinmarketcap.com/>

- Zapisi predstavljaju transakcije između računa (adresa)
- Svaka transakcija mora definirati ulaze novca i njegove izlaze
- Transakcija je ispravna ako ulazi novca (koji su izlazi neke prethodne transakcije) imaju više novca od onog koji treba završiti na izlazima
- Tri problema
  - Krađa novca od drugog korisnika
    - Nemoguća zbog digitalnog potpisa
  - Uskraćivanje usluge
    - Nemoguće zbog raspodijeljenosti
  - Trošenje novca kojeg nema (double-spending problem)
    - Ovaj problem se rješava odbacivanjem neispravnih transakcija i čekanjem na pojavljivanje transakcije u *blockchainu*



Izvor:<https://s-media-cache-ak0.pinimg.com/originals/25/d2/46/25d246a383ac7fafbe641d6735a51113.png>

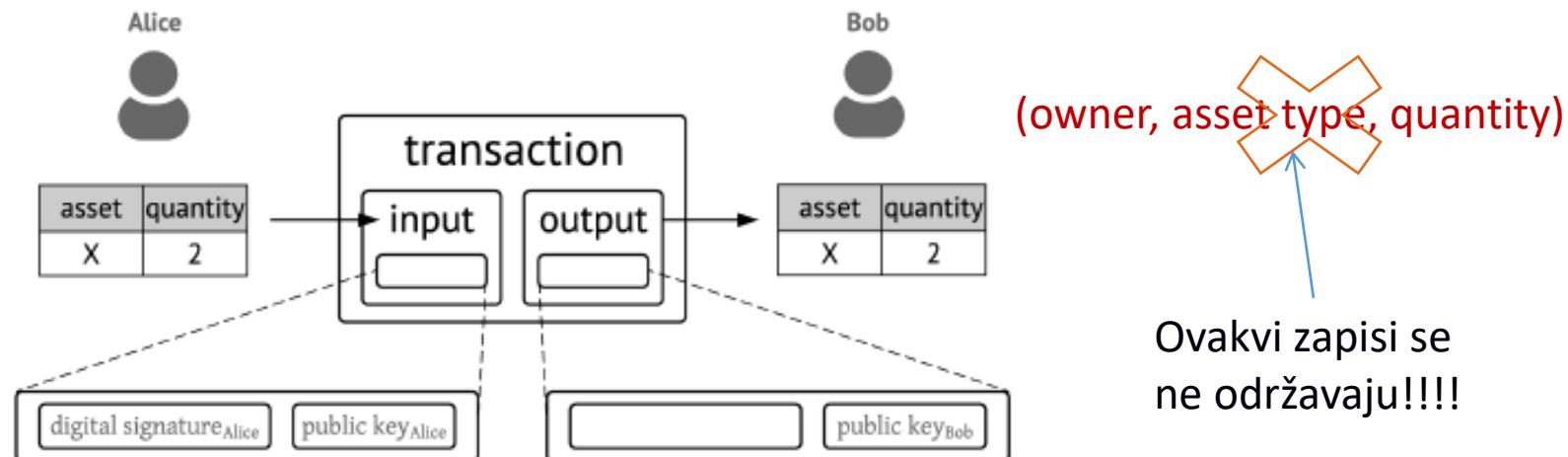
# Bitcoin: pojmovi

- Bitcoin wallet: sadrži privatni i javni ključ vlasnika te ključeve na transakcije koje na izlazu referenciraju vlasnika walleta
- Kada nekome prebacujete bitcoin – referencirate se na njegov javni ključ
- Bitcoin address: hash nad javnim ključem
- Različite verzije čvorova: full node, miner, simple payment verification (SPV) node (wallet + routing)

|                                                               |   |                                                                                                 |                                  |              |
|---------------------------------------------------------------|---|-------------------------------------------------------------------------------------------------|----------------------------------|--------------|
| 1PL6gsm49xCFMvrXqgGcee5cdrG119GoWN (0.00137322 BTC - Output)  | → | 1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3 - (Unspent)<br>1ET3oBGfBJpunyjE7owyVtmBjmvcDycQe - (Unspent) | 0.00033324 BTC<br>0.00093376 BTC | 0.001267 BTC |
| Summary                                                       |   |                                                                                                 |                                  |              |
| Size 226 (bytes)                                              |   |                                                                                                 |                                  |              |
| Weight 904                                                    |   |                                                                                                 |                                  |              |
| Received Time 2017-10-29 16:47:58                             |   |                                                                                                 |                                  |              |
| Included In Blocks 492229 ( 2017-10-29 16:51:42 + 4 minutes ) |   |                                                                                                 |                                  |              |
| Confirmations 731 Confirmations                               |   |                                                                                                 |                                  |              |
| Visualize <a href="#">View Tree Chart</a>                     |   |                                                                                                 |                                  |              |
| Inputs and Outputs                                            |   |                                                                                                 |                                  |              |
| Total Input 0.00137322 BTC                                    |   |                                                                                                 |                                  |              |
| Total Output 0.001267 BTC                                     |   |                                                                                                 |                                  |              |
| Fees 0.00010622 BTC                                           |   |                                                                                                 |                                  |              |
| Fee per byte 47 sat/B                                         |   |                                                                                                 |                                  |              |
| Fee per weight unit 11.75 sat/WU                              |   |                                                                                                 |                                  |              |
| Estimated BTC Transacted 0.00033324 BTC                       |   |                                                                                                 |                                  |              |
| Scripts <a href="#">Hide scripts &amp; coinbase</a>           |   |                                                                                                 |                                  |              |

Primjer transakcije  
Transakcija se smatra  
potvrđenom kada je iz njenog  
dodano još 6 blokova

# Primjer transakcije u Bitcoinu



- Kako bi prebacila 2 bitcoina Bobu, Alice mora dokazati da je u prethodnim transakcijama ona primila 2 ili više bitcoina (Alice se mora referencirati na prethodne transakcije gdje je ona referencirana u izlazu)
- Svaki čvor sadrži kopiju svih transakcija i provjerava sve transakcije od izvorne! Za više informacija: [https://www.youtube.com/watch?time\\_continue=379&v=Lx9zgZCMqXE](https://www.youtube.com/watch?time_continue=379&v=Lx9zgZCMqXE) ili <http://www.imponderablethings.com/2013/07/how-bitcoin-works-under-hood.html>

# Kako izgleda transakcija kod *Bitcoin*a?

- Metapodaci kao što su veličina transakcije, broj ulaza i izlaza novca te sažetak transakcije (predstavlja i njen jedinstveni ID)
- Ulazi
  - Jedinstveni ID prethodne transakcije čiji će se izlaz koristiti kao ulaz ove
  - Redni broj izlaza u prethodnoj transakciji koji će se koristiti kao ulaz ove
  - Skripta `scriptSig` u posebnom jeziku Script (*Bitcoin scripting language*)
- Izlazi
  - Vrijednost
  - Skripta `scriptPubKey` u jeziku Script

Input:

Previous tx:

f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b00  
9ca73dd04470b9a6

Index: 0

scriptSig:

304502206e21798a42fae0e854281abd38bacd1aeed3ee37  
38d9e1446618c4571d10  
90db022100e2ac980643b0b82c0e88ffdfecc6b64e3e6ba35  
e7ba5fdd7d5d6cc8d25c6b241501

Output:

Value: 5000000000

scriptPubKey: OP\_DUP OP\_HASH160  
404371705fa9bd789a2fc52d2c580b65d35549d  
OP\_EQUALVERIFY OP\_CHECKSIG

# Javni i privatni ledgeri

Javni (public and permissionless )



ethereum



IOTA

Privatni (private and permissioned)



**HYPERLEDGER**

svi korisnici su autenticirani, svi čvorovi imaju potrebu rударити, nema opasnosti za Sybil attack (jedan predstavnik uzima više različitih identiteta), manje mreže do npr. 80 čvorova

# Može li DLT podržavati bilo koju aplikaciju?

- *Bitcoinov jezik Script* je vrlo jednostavan
  - Namijenjen je obavljanju jednostavnih poslova (npr. provjera ispravnosti transakcije)
  - Temelji se na stogu te ne koristi dodatnu memoriju niti varijable
  - Može se procijeniti koliko će neko izvođenje trajati i koliko će memorije zahtijevati
  - Nije *Turing-complete* jezik (njime se ne mogu izvesti baš sve proizvoljne funkcije)
- Platforma *Ethereum* podržava nekoliko *Turing-complete* jezika (primarni jezik je *Solidity*)
  - Nad njim se može napraviti bilo kakva (smislena) raspodijeljena aplikacija
  - Ethereum je za *blockchain* ono što je univerzalno računalo za namjenska računala

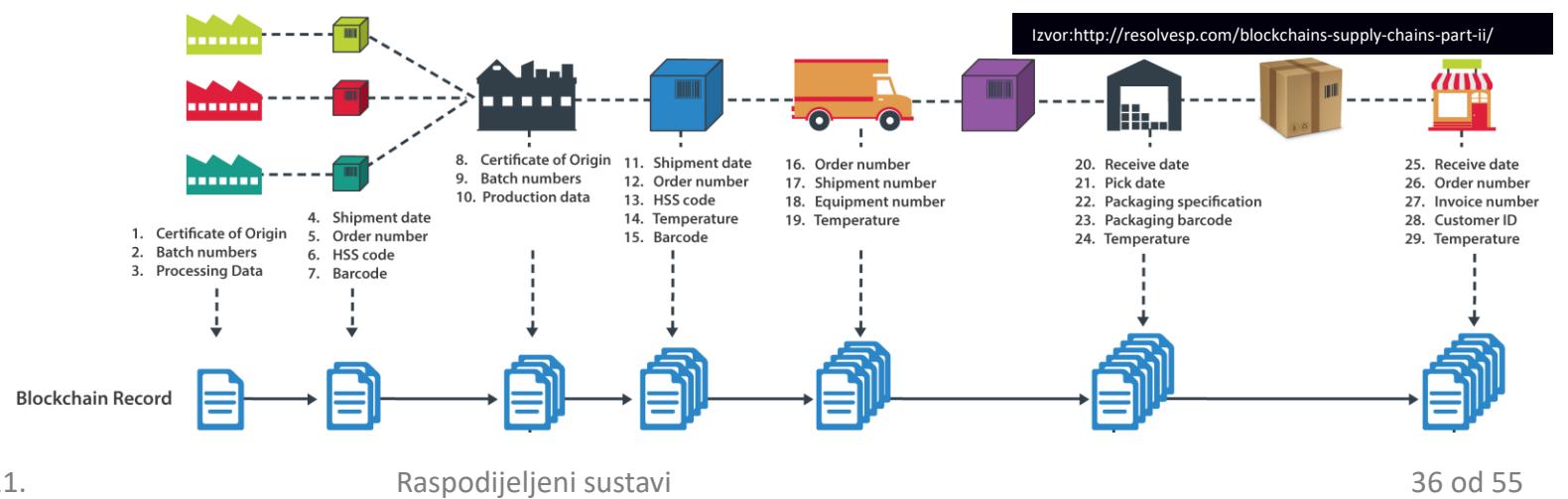
# Koje su primjene tehnologije *blockchain*?

- Bankarstvo
- Plaćanja i prijenosi novca
- Kibernetička sigurnost
- Edukacija
- Glasovanje
- Internet stvari (IoT)
- Trgovanje stvarima i dionicama
- Osiguranje
- Zdravstvo
- Upravljanje opskrbnim lancima (sljedivost)
- Donacije
- Masovno financiranje (*crowdfunding*)

Izvor:<https://www.cbinsights.com/research/industries-disrupted-blockchain/>

# Praćenje sljedivosti

- Uobičajeni problemi u prehrambenim opskrbnim lancima
  - Prijevare vezane uz zamjenu, mijenjanje i krivo označavanje hrane
  - Illegalna proizvodnja hrane
  - Pokvarena hrana
- Korištenjem tehnologije blockchain se čitav lanac opskrbe može digitalno potpisati pa su podaci o lancu opskrbe
  - Djeljivi (*shareable*)
  - Dokazivi (*traceable*)
  - Transparentni (*transparent*)

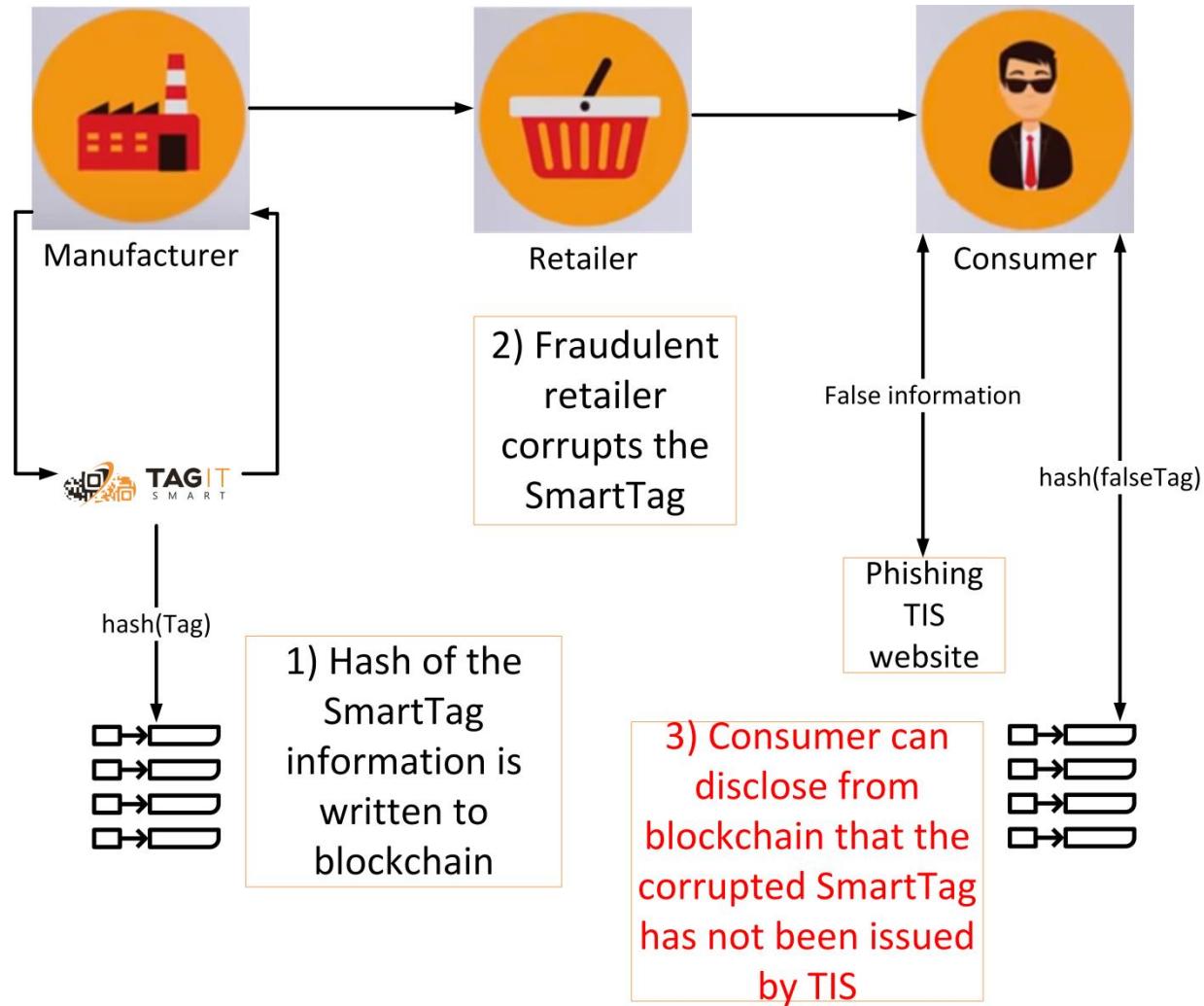


# Rješenje razvijeno na Zavodu za telekomunikacije

DL-Tags: *Decentralized, privacy-preserving and verifiable management of Smart Tags*

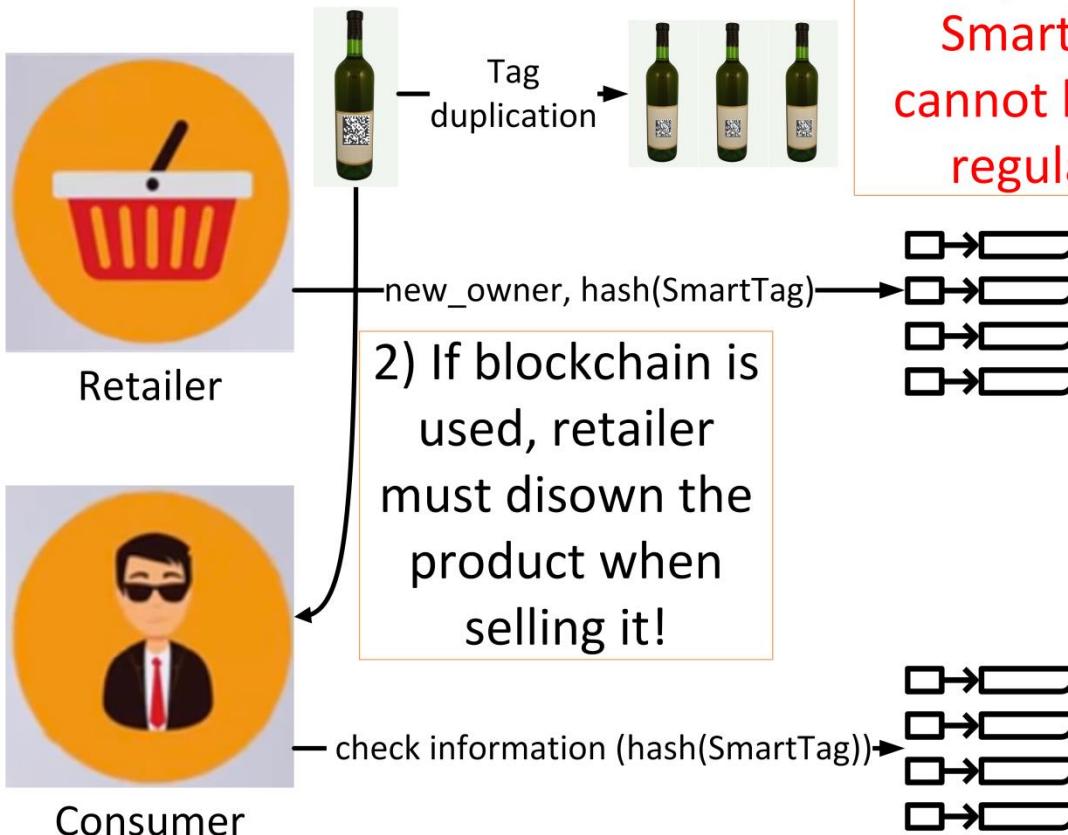
- Distributed Ledger (DL) Technology to track state changes of products labelled by Smart Tags
- DL is a trustless intermediary between the stakeholders
- Involved stakeholders: product owners, TIS platform, eCommerce platform (Magento), customers
- Innovation
  - Decentralized, privacy-preserving and verifiable approach for storing and sharing product-related data, thus ensuring data integrity
  - No need for data storage outside the stakeholders' space

# Prevention of Smart Tag corruption

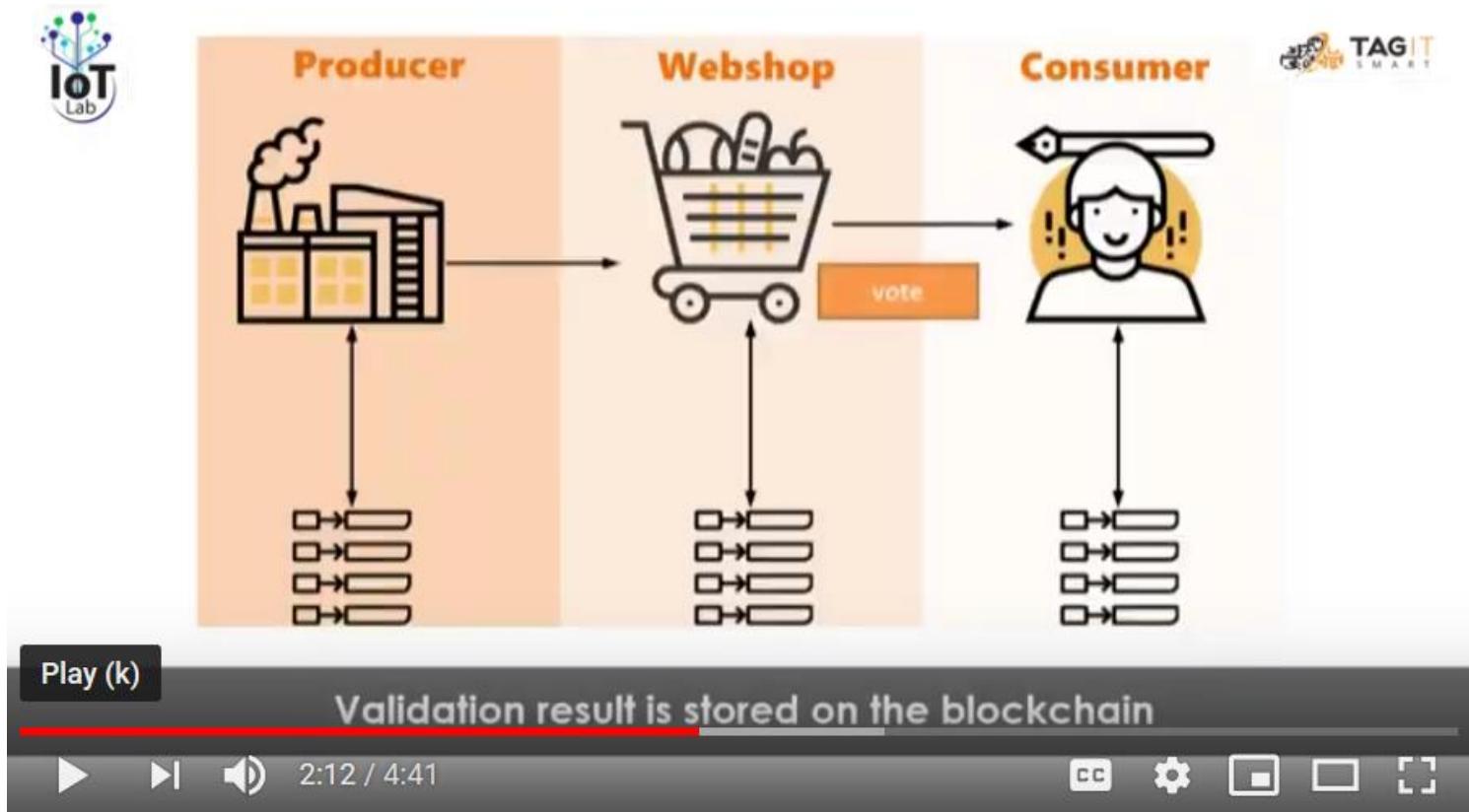


# Prevention of Smart Tag duplication

1) Fraudulent retailer wants to duplicate the SmartTags



# DL-Tags video



<https://www.youtube.com/watch?v=JCC98iMCPOs>

# Ethereum i pametni ugovori

# Sadržaj

- Upoznavanje s terminologijom
- Upoznavanje s razvojnim okruženjem
- Pokretanje lokalnog Ethereum čvora
- Upoznavanje s jezikom Solidity
- Izrada vlastitog tokena po ERC20 standard (demo)

# Što je *Ethereum*?

- Javna platforma otvorenog koda temeljena na tehnologiji blok-lanca, koristi povezanu kriptovalutu *Ether*
- Podržava pametne ugovore (*smart contracts*)
  - Aplikacije sa stanjem pohranjenim u *blockchainu*
  - Mogu biti u interakciji s ostalim pametnim ugovorima
  - Mogu donositi odluke
  - Mogu proslijeđivati *Ethere* drugima
- Problem s resursima pri izvođenju programa
  - Izvođenje svake instrukcije se plaća
  - Svaka pohrana podatka se plaća



Izvor:[https://www.ethereum.org/images/logos/ETHEREUM-LOGO\\_PORTRAIT\\_Black\\_small.png](https://www.ethereum.org/images/logos/ETHEREUM-LOGO_PORTRAIT_Black_small.png)

# Pametni ugovor

*Smart contracts are software programs that live on a blockchain and form the basis of many of the new blockchain applications and schemes. They are essentially automated systems that can provide services in exchange for cryptocurrency. However, because blockchains are not good for storing large amounts of data nor for querying the state of the outside world, they need services that exist off the blockchain to do those things for them.*

Izvor: <https://spectrum.ieee.org/computing/networks/how-smart-contracts-work>

# Upoznavanje s terminologijom

- Ether (ETH) – valuta povrh platforme Ethereum
- Ethereum – platforma, raspodijeljena mreža
  - Najmanja jedinica – 1 Wei

| Jedinica            | Vrijednost u Weima                        |
|---------------------|-------------------------------------------|
| Kwei (babbage)      | 1,000 Wei                                 |
| Mwei (lovelace)     | 1,000,000 Wei                             |
| Gwei (shannon)      | 1,000,000,000 Wei                         |
| microether (szabo)  | 1,000,000,000,000 Wei                     |
| milliether (finney) | 1,000,000,000,000,000 Wei                 |
| ether               | 1,000,000,000,000,000,000 Wei : $10^{18}$ |

# Upoznavanje s terminologijom

- Pametni ugovor
  - aplikacija koja se izvodi u distribuiranom okruženju
- Solidity
  - jezik kojim pišemo Pametne ugovore
  - postoje i Vyper i LLL
  - prevodi se u EVM bytecode
- EVM
  - Ethereum Virtual Machine
  - svaki čvor u mreži ima mogućnost izvršavanja EVM bytecodea
- Ethereum možemo promatrati kao automat stanja
  - Stanje se mijenja provedbom transakcija

# Upoznavanje s terminologijom

- Transakcije se provode onda kada se pronađe novi blok
  - ~ 15 sekundi za blok
- Svaku transakciju koja izaziva operaciju pisanja potrebno je platiti
  - Gas - jedinica kojom plaćamo izvedbu koda na Ethereum platformi
  - Odvojena jedinica od ETH
  - Cijena izvođenja nije volatilna kao cijena ETH
- Gas Limit – maksimalna količina Gasa za transakciju
- Gas Price – količina ETH po jedinici gasa
  - najčešće Gwei

# Upoznavanje s razvojnim okruženjem

- Editor – Visual Studio Code, solidity ekstenzija
  - <https://code.visualstudio.com/>
  - <https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity>
- Lokalni Ethereum čvor
  - <https://www.trufflesuite.com/ganache>
- Razvojni okvir Truffle
  - <https://www.trufflesuite.com/>

# Arhitektura sustava



# Lokalni Ethereum čvor

- Ne moramo preuzimati cijelu raspodijeljenu knjigu
- Ne trošimo Ether (barem ne pravi)
- Ne čekamo novi blok
  - transakcija se izvodi u realnom vremenu
  - PoA
- Lako je obrisati lanac i početi isponova
- Opcije:
  - Ganache
  - ganache-cli

# Ganache

The screenshot shows the Ganache interface with the following details:

| ADDRESS                                    | BALANCE    | TX COUNT | INDEX |
|--------------------------------------------|------------|----------|-------|
| 0x61B73C88c768c423fa7a4EC9D682A8610CFED691 | 99.97 ETH  | 4        | 0     |
| 0xC032e96BaFa2E32a384DfEafA01f10e4a165EA77 | 100.00 ETH | 0        | 1     |
| 0xe6057dA6Fd6dc5BDEE6eEF3E80A3303d80aA43D  | 100.00 ETH | 0        | 2     |
| 0x3d2cC1bfB8cB4eF9Cf1786a442cCe0f6d645a724 | 100.00 ETH | 0        | 3     |
| 0xC7e4C17d307ba5F8513587357fdAD3438EB817a  | 100.00 ETH | 0        | 4     |
| 0x1513Db3d727f1CB3BE7F0D6756b08d3b3B3E855F | 100.00 ETH | 0        | 5     |
| 0x31aEBA8698Dc34056d6AC400dEbe345eB3A40B93 | 100.00 ETH | 0        | 6     |
| 0x71104f8Dd77feA7683bA80EF86DDe896C96eF249 | 100.00 ETH | 0        | 7     |
| 0x20eD2823a77F507b9e9FF63A359b4b9b41De9657 | 100.00 ETH | 0        | 8     |
| 0xf5de8a18197a6d44D0c17E4A9B2300Ae212a38aC | 100.00 ETH | 0        | 9     |

# Komunikacija s čvorom - Json RPC

- Json
  - format za razmjenu podataka, opisuje brojeve, stringove, liste, i mape.
- Json RPC
  - stateless, protokol za poziv udaljene metode (RPC). Neovisan o transportnoj metodi, možemo ga koristiti putem socketa, preko HTTP-a i slično.
  - Primjer poziva:

```
#!/usr/bin/env bash
curl -H "Content-Type: application/json" -X POST --data
'{"id":1337,"jsonrpc":"2.0","method":"evm_mine","params":[1231006505000]}'
http://localhost:7545
```

# Metamask

- Ethereum wallet
  - wallet u sebi sprema privatne ključeve
  - često nudi neke dodatne funkcionalnosti – digitalno potpisivanje
- Metamask
  - Ethereum wallet u pregledniku
  - Interakcija s blockchainom u pregledniku
  - <https://metamask.io/>

# ERC20 token

- Ethereum Improvement Proposals
  - <https://eips.ethereum.org/EIPS/eip-20>
- Valuta povrh Ethereuma (nije ETH)
  - ICO boom 2017
- Neke od funkcija:
  - function name() public view returns (string)
  - function symbol() public view returns (string)
  - function totalSupply() public view returns (uint256)
  - function balanceOf(address \_owner) public view returns (uint256 balance)
  - function transfer(address \_to, uint256 \_value) public returns (bool success)

# Literatura

1. K. Pripužić: Blockchain – sigurnost, sljedivost, predavanje na konferenciji RFID 2017.
2. M. E. Peck: Blockchains: How They Work and Why They'll Change the World, IEEE Spectrum, Sept 2017. <https://spectrum.ieee.org/static/special-report-blockchain-world>
3. K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," in *IEEE Access*, vol. 4, pp. 2292-2303, 2016.
4. S. Nakamoto: "Bitcoin: A peer-to-peer electronic cash system", 2008.
5. V. Buterin: "Ethereum white paper: A Next-Generation Smart Contract and Decentralized Application Platform", 2017.
6. Dinh, Tien Tuan Anh, et al. "Untangling Blockchain: A Data Processing View of Blockchain Systems." arXiv preprint arXiv:1708.05665 (2017).