



Diplomski studij

Informacijska i  
komunikacijska tehnologija:

Telekomunikacije i  
informatika

Računarstvo:

Programsko inženjerstvo i  
informacijski sustavi

Računarska znanost

Ivana Podnar Žarko  
Krešimir Pripužić  
Ignac Lovrek  
Mario Kušek

## **Raspodijeljeni sustavi**

Radna inačica udžbenika v.1.3

Ak. g. 2016./2017.

## **Predgovor**

Radna inačica udžbenika "Raspodijeljeni sustavi" namijenjena je studentima diplomskog studija Informacijska i komunikacijska tehnologija i Računarstvo na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Ova knjiga se bavi temeljnim konceptima raspodijeljenih sustava, a posebice modelima i tehnoškim rješenjima za komunikaciju udaljenih procesa koji su ilustrirani odabranim praktičkim primjerima.

Autori zahvaljuju dr. sc. Danielu Skrobi na suradnji pri pripremi dijela predavanja, zadataka i laboratorijskih projekata iz Raspodijeljenih sustava, kao i na mnogim lijepim i svršishodnim grafičkim prikazima.

Poglavlje 10. Vrednovanje performansi raspodijeljenih sustava napisano je prema predlošku predavanja kojeg je pripremio prof. dr. sc. Dalibor Vrsalović na temelju knjiga N. J. Gunthera „The practical performance analyst“ te D. A. Menascéa i V. A. F. Almeide „Capacity Planning for Web Services“ i vlastitih istraživanja.

## **Autori**

Copyright© Ivana Podnar Žarko, Krešimir Pripužić, Ignac Lovrek, Mario Kušek, 2013.

Sva prava pridržana. Nijedan dio ove knjige ne smije se preslikati, snimati ili na drugi način umnožavati, spremiti u obliku pogodnom za umnožavanje ili prenositi elektronički ili mehanički u bilo kojem obliku bez prethodne dozvole autora.

## Sadržaj

1	Uvod u raspodijeljene sustave.....	1
1.1	Definicija, obilježja i vrste raspodijeljenih sustava .....	1
1.2	Zahtjevi na raspodijeljene sustave.....	3
1.2.1	Otvorenost.....	3
1.2.2	Transparentnost .....	3
1.2.3	Skalabilnost.....	4
1.2.4	Kvaliteta usluge.....	5
1.3	Arhitektura raspodijeljenih sustava .....	5
1.3.1	Slojeviti arhitekturni model .....	6
1.3.2	Objektni arhitekturni model .....	7
1.3.3	Arhitektura zasnovana na podacima .....	8
1.3.4	Arhitektura zasnovana na događajima.....	8
1.4	Osnovni modeli raspodijeljene obrade.....	9
1.4.1	Model klijent-poslužitelj .....	9
1.4.2	Model ravnopravnih sudionika.....	11
1.4.3	Modeli premještanja programa i programskih agenata.....	12
1.5	Studijski primjer: web .....	14
1.6	Pitanja za učenje i ponavljanje.....	16
2	Procesi i komunikacija .....	17
2.1	Obilježja komunikacije .....	18
2.2	Obilježja procesa.....	20
2.3	Pitanja za učenje i ponavljanje.....	21
3	Sloj raspodijeljenog sustava za komunikaciju među procesima.....	23
3.1	Komunikacija korištenjem priključnica .....	23
3.2	Poziv udaljene procedure/metode .....	31
3.3	Komunikacija porukama .....	36
3.4	Komunikacija na načelu objavi-pretplati .....	38
3.5	Dijeljeni podatkovni prostor .....	43
3.6	Pitanja za učenje i ponavljanje.....	44
4	Arhitekture web-aplikacija .....	46
4.1	Protokol HTTP .....	46
4.2	Jezik HTML.....	48
4.2.1	HTML 5.....	52
4.3	Jezik CSS .....	52
4.4	Priručna spremišta .....	55

4.4.1	Model roka trajanja .....	56
4.4.2	Model validacije.....	56
4.4.3	Posrednički poslužitelj .....	57
4.5	Web-aplikacije.....	58
4.5.1	CGI .....	58
4.5.2	Poslužiteljske skripte .....	59
4.6	Arhitekture web-aplikacija.....	59
4.7	Skripte na klijentu .....	61
4.8	AJAX.....	65
4.9	Integracija s drugim sjedištim .....	67
4.10	Pitanja za učenje i ponavljanje.....	68
5	Web-usluge.....	69
5.1.1	Poziv udaljene procedure .....	71
5.1.2	Usluge temeljene na porukama/dokumentima .....	71
5.1.3	Usluge temeljene na prijenosu prikaza stanja resursa.....	71
5.2	Protokol SOAP .....	71
5.3	Jezik WSDL.....	74
5.3.1	Apstraktni opis.....	74
5.3.2	Konkretni opis.....	75
5.3.3	Elementi WSDL-a .....	75
5.4	UDDI .....	77
5.5	Web-usluge temeljene na pozivu udaljene procedure .....	77
5.6	Web-usluge temeljene na dokumentima .....	83
5.7	Web-usluge temeljene na prijenosu prikaza stanja resursa .....	89
5.7.1	Pokretanje baze podataka Derby i stvaranje tablica .....	92
5.7.2	Stvaranje web-usluge i testiranje .....	93
5.8	Uslužno orijentirana arhitektura.....	94
5.8.1	Jezik BPEL.....	95
5.9	Web-aplikacije koje koriste web-usluge .....	95
5.10	Pitanja za učenje i ponavljanje.....	97
6	Model raspodijeljenog sustava.....	98
6.1	Osnovni model raspodijeljenog sustava .....	98
6.2	Sinkroni model .....	103
6.3	Asinkroni model .....	109
6.4	Pitanja za učenje i ponavljanje.....	114
7	Sinkronizacija procesa u vremenu .....	116

7.1	Primjena sata u raspodijeljenoj okolini.....	117
7.1.1	Usklađivanje fizičkih satova .....	118
7.1.2	Usklađivanje logičkih satova.....	120
7.2	Sinkronizacija tijeka izvođenja procesa.....	123
7.2.1	Mehanizam semafora .....	123
7.2.2	Mehanizam razmjene događaja .....	124
7.3	Međusobno isključivanje procesa.....	125
7.3.1	Isključivanje putem središnjeg upravljača .....	125
7.3.2	Raspodijeljeno isključivanje.....	125
7.3.3	Isključivanje zasnovano na primjeni prstena.....	128
7.4	Pitanja za učenje i ponavljanje.....	128
8	Konzistentnost i replikacija podataka.....	130
8.1	Modeli konzistentnosti podataka .....	130
8.1.1	Stroga konzistentnost.....	131
8.1.2	Slijedna konzistentnost.....	132
8.1.3	Povezana konzistentnost.....	133
8.1.4	Konzistentnost redoslijeda upisivanja .....	134
8.1.5	Slaba konzistentnost.....	135
8.1.6	Konzistentnost otpuštanja i zauzimanja kritičnog odsječka.....	136
8.1.7	Usporedba modela konzistentnosti.....	137
8.2	Organizacija sustava replika i vrste replika .....	137
8.2.1	Trajne replike .....	140
8.2.2	Poslužiteljske replike .....	140
8.2.3	Korisničke replike.....	141
8.3	Metode održavanja konzistentnosti replika .....	142
8.3.1	Dohvaćanje promjena sadržaja replika .....	142
8.3.2	Prosljeđivanje promjena sadržaja replika.....	143
8.4	Protokoli za ostvarivanje operacija čitanja i upisivanja replika .....	145
8.4.1	Pasivna replikacija.....	145
8.4.2	Aktivna replikacija.....	147
8.5	Pitanja za učenje i ponavljanje.....	147
9	Otpornost na neispravnosti .....	149
9.1	Otpornost procesa na ispade.....	150
9.1.1	Organizacija skupine procesa .....	150
9.1.2	Sporazum skupine procesa .....	151
9.2	Pouzdana komunikacija skupine procesa .....	154

9.2.1	Pouzdana komunikacija bez mogućih ispada procesa.....	154
9.2.2	Pouzdana komunikacija uz moguće ispade procesa.....	155
9.2.3	Slijed isporuke poruka .....	156
9.3	Raspodijeljeno izvršavanje operacije .....	158
9.3.1	Protokol dvofaznog izvršavanja .....	158
9.3.2	Protokol trofaznog izvršavanja .....	161
9.4	Oporavak nakon ispada.....	162
9.5	Pitanja za učenje i ponavljanje.....	164
10	Vrednovanje performansi raspodijeljenih sustava.....	165
10.1	Modeli vrednovanja nefunkcijskih obilježja.....	165
10.2	Pouzdanost i raspoloživost .....	170
10.3	Ukupni trošak vlasništva .....	171
10.4	Performance sustava .....	172
10.5	Prirodne granice rasta ubrzanja i kapaciteta .....	173
10.5.1	Arhitekturna unapređenja.....	174
10.5.2	Ubrzanje i smanjenje vremena odziva .....	175
10.5.3	Porast kapaciteta.....	178
10.5.4	Raspodjeljivanje zahtjeva .....	181
10.6	Vrednovanje performansi raspodijeljenih sustava mrežom repova.....	182
10.6.1	Osnovni pojmovi teorije repova i model jednopošlužiteljskog sustava.....	182
10.6.2	Serijski povezani poslužitelji.....	186
10.6.3	Paralelni poslužitelji .....	187
10.6.4	Poslužitelj s povratnom vezom .....	189
10.6.5	Mreže repova s višestrukim povratnim vezama .....	190
10.6.6	Model zatvorenog poslužiteljskog sustava .....	192
11	Sustavi s ravnopravnim sudionicima .....	195
11.1	Centralizirani i decentralizirani raspodijeljeni sustavi .....	195
11.2	Nestrukturirani sustavi P2P.....	198
11.3	Strukturirani sustavi P2P .....	200
11.4	Gnutella.....	202
11.5	Chord.....	204
11.6	Usporedba protokola Gnutella i Chord.....	208
11.7	Pitanja za učenje i ponavljanje.....	210
12	Grozd, splet računala i računalni oblak .....	212
12.1	Grozd računala .....	212
12.1.1	Svojstva .....	213

12.1.2	Primjeri grozdova .....	213
12.2	Splet računala.....	214
12.2.1	Arhitektura .....	214
12.2.2	Primjena .....	216
12.2.3	Primjeri spletova .....	216
12.3	Računalni oblak.....	217
12.3.1	Modeli usluga .....	218
12.3.2	Prednosti .....	219
12.3.3	Nedostaci.....	220
12.4	Mehanizmi .....	220
12.4.1	Prijenos podataka.....	220
12.4.2	Raspoređivanje zahtjeva .....	222
12.5	Pitanja za učenje i ponavljanje.....	227
13	Literatura .....	228

## 1 UVOD U RASPODIJELJENE SUSTAVE

### 1.1 Definicija, obilježja i vrste raspodijeljenih sustava

Raspodijeljeni sustav obilježava pojam raspodijeljenosti ili distribuiranosti. Riječ je o sustavu koji nije cjelovit – monolitan, već je sastavljen od više međusobno povezanih, fizički i logički raspodijeljenih dijelova koji zajedno ostvaruju zadaću kojoj su namijenjeni. Mnogi su autori nastojali definirati raspodijeljene računalne i komunikacijske sustave. Tako Andrew S. Tanenbaum raspodijeljeni sustav definira kao:

“Skup neovisnih računala koji korisniku izgleda kao jedan cjeloviti sustav.” (Tanenbaum & Van Steen, 2007)

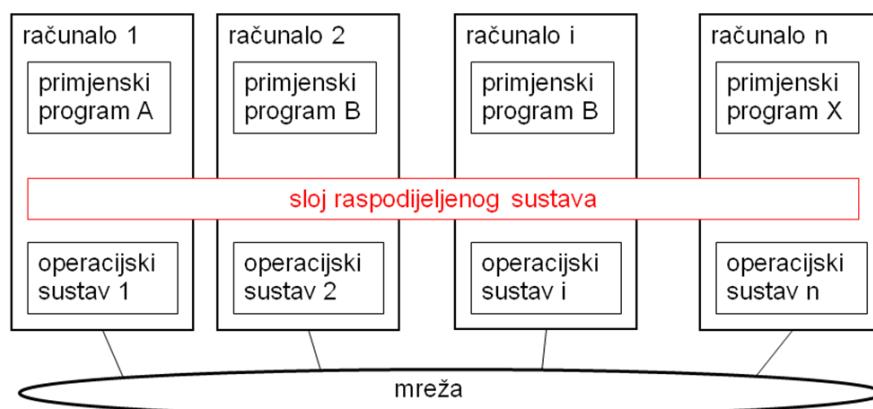
Leslie Lamport se našalio kada je rekao da je to:

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

Za obje definicije zajedničko je da korisnik takvog sustava ne „vidi“ njegovu raspodijeljenost jer je raspodijeljenost dobro izvedenog sustava prikrivena. George Coulouris, nastojeći ukazati na izvedbena obilježja raspodijeljenog sustava, kaže da je to:

“Sustav u kojem programske i sklopovske komponente umreženih računala komuniciraju i usklađuju svoje aktivnosti isključivo razmjenom poruka.” (Coulouris, Dollimore, Kindberg, & Blair, 2012)

Raspodijeljeni sustav predložen slikom (Slika 1.1) slijedi upravo takvu definiciju. Više računala (1, 2, ...i, ..., n), spojeno je na mrežu putem koje razmjenjuju poruke. Svako računalo s vlastitim operacijskim sustavom izvodi primjenske programe, tj. aplikacije (A, B, ..., X), pri čemu se neki od tih programa izvode na samo jednom računalu (A, X), a neki na više njih (B). Ako su za neku zadaću potrebni programi A i B, trebat će se uspostaviti komunikacija između računala\_1 na kojem se izvodi program\_A i jednog od računala s programom\_B, a to je računalo\_2 ili računalo\_i, kako bi se moglo razmijeniti poruke s podacima potrebnima za obavljanje željene zadaće.



Slika 1.1.Prikaz raspodijeljenog sustava

Očevidno je nekoliko problema. Ista se obrada (B) može obaviti na više računala, tako da će trebati odabrati jedno od njih prilikom obavljanja neke zadaće (računalo\_2 ili računalo\_i) prema nekom kriteriju, primjerice, računalo „bliže“ računalu\_1, manje opterećeno računalo i dr. Također je moguće da će prilikom izvođenja različitih zadaća obrada (B) biti izvršena na različitim računalima. Nadalje, po završetku obrade na odabranom računalu, trebat će uskladiti istovrsne podatke na drugom računalu

kako bi se održala njihova konzistentnost. Daljnji problemi nastaju ako sklopovska i programska oprema računala nije jednaka, odnosno ako sustav nije homogen. Svaka raznorodnost, tj. heterogenost, primjerice operacijskih sustava za računalo\_1 i računalo\_2, će tražiti neki oblik međusobne prilagodbe pri komunikaciji programa A i B.

Komunikacija u raspodijeljenom sustavu zasniva se na razmjeni podataka među računalnim procesima koji se u načelu izvode na različitim, i u mnogim situacijama udaljenim računalima s različitom sklopoškom i programskom opremom. Kako bi sustav „izgledao kao jedan cjeloviti sustav“ potreban je posebni sloj raspodijeljenog sustava čija je zadaća prikrivanje raspodijeljenosti procesa i sredstava, tj. resursa, na više računala u mreži, kao i izvedbene, sklopoške i programske, raznorodnosti.

Sloj raspodijeljenog sustava ima ulogu programskog posredničkog sloja, tj. međuopreme (engl. *middleware*) koji omogućuje povezivanje, komunikaciju i suradnju aplikacija, sustava i uređaja te omogućuje interakciju programa na aplikacijskoj razini. Nalazi se između operacijskog sustava i aplikacijskih programa u programskoj arhitekturi sustava. Cilj je olakšati korisnicima i aplikacijama pristup i korištenje udaljenih sredstava na kontroliran i učinkovit način.

Razlozi za raspodijeljene sustave su višestruki. Najprije to je inherentna prostorna raspodijeljenost korisnika, podataka, informacija i sredstava. Dalje, razlozi mogu biti funkcionalni kada je riječ o različitim namjenama i mogućnostima sustava ili različitim ulogama u sustavu (npr. korisnik – davatelj usluge, proizvođač – potrošač, i sl.). Razlogom može biti i veličina sustava, način proširenja sustava, bolja raspodjela opterećenja i mogućnost njegovog uravnoteženja u sustavu. Pouzdanost i raspoloživost ne mogu se postići bez više istovrsnih komponenata sustava smještenih na različitim mjestima i povezanih na više načina. Tu su zatim i gospodarski razlozi, kao što su cijena izgradnje, troškovi održavanja sustava u radu i drugi.

Raspodijeljeni sustavi javljaju se u mnogim područjima. Prvenstveno su to raspodijeljeni računalni sustavi s više ili mnogo umreženih računala, organiziranih na različite načine, u grozd računala (engl. *cluster*), splet računala (engl. *grid*), ili računalni oblak (engl. *cloud*). Raspodijeljeni informacijski sustavi primjenjuju se za obradu transakcija i integraciju poslovnih aplikacija. Raspodijeljeni su i sustavi za pružanje informacijskih i komunikacijskih usluga, u sloju usluga i aplikacija (engl. *service & application layer*) i uslužnom stratumu (engl. *service stratum*) nove generacije mreža. Sveprisutni sustavi (engl. *ubiquitous, pervasive system*) čije su komponente ugrađene u ili povezane s okružjem u kojem djeluju čine tzv. pametne okoline (engl. *smart environments*). Posebno su važni Internet stvari (engl. *Internet of Things, IoT*) i sustavi komunikacije stroja sa strojem (engl. *Machine to Machine, M2M*). „Stvari“ u Internetu su fizički i virtualni objekti, odnosno Internetom povezani objekti (engl. *Internet Connected Object, ICO*) u fizičkom i informacijskom svijetu. Fizički objekt može primjerice djelovati kao senzor koji prikuplja podatke ili aktuator koji provodi neku radnju. Virtualni objekt isto tako može sadržavati podatke i programski provoditi akcije u interakciji s drugim fizičkim ili virtualnim objektima. Pojam „stroja“ u sustavu M2M podrazumijeva pokretni umreženi uređaj – objekt koji prikuplja podatke iz okoliša te ih prenosi i predočuje drugim uređajima, ili poduzima neku radnju.

Obilježja raspodijeljenog sustava mogu se sažeti ovako:

- paralelne aktivnosti: autonomni dijelovi sustava istodobno izvode više aktivnosti,
- komunikacija razmjenom poruka: dijelovi sustava razmjenjuju podatke, a ne dohvataju ih iz zajedničke memorije,
- dijeljenje sredstava: zajedničkim sredstvima može pristupiti više dijelova sustava,
- nema globalnog stanja: jedan proces ne zna stanje svih ostalih procesa u svim dijelovima sustava i
- nema globalnog vremenskog takta: dijelovi sustava nisu pokretani istim vremenskim taktom tako da je ograničena mogućnost vremenskog usklađivanja njihovog rada.

S teorijskog motrišta, svi modeli raspodijeljenih sustava sastoje se od procesa koji međusobno komuniciraju i razmjenjuju podatke, a temeljni formalizmi sadrže tri klase modela:

- interakcijski modeli, koji obuhvaćaju procese, komunikaciju između procesa te vremensku usklađenost odvijanja i komunikacije procesa,
- modeli kvara, koji obuhvaćaju moguće kvarove i njihov utjecaj na odvijanje i komunikaciju procesa i
- modeli sigurnosti, koji obuhvaćaju prijetnje odvijanju i komunikaciji procesa te mjere zaštite.

U ovoj se knjizi na teorijskoj i izvedbenoj razini razrađuju interakcijski modeli i modeli kvarova.

## 1.2 Zahtjevi na raspodijeljene sustave

Osnovni zahtjevi koji se postavljaju pri oblikovanju raspodijeljenih sustava su sljedeći:

- otvorenost,
- transparentnost,
- skalabilnost i
- kvaliteta usluge.

Oni govore o tome na kakvim normama je sustav zasnovan, koje značajke raspodijeljenog sustava treba prikriti i na koji način, kako riješiti prilagodbu sustava broju korisnika i prostoru na kojem se nalaze, te na kraju, o očekivanjima korisnika o načinu rada sustava.

### 1.2.1 Otvorenost

Otvorenim sustavom (engl. *open system*) smatra se onaj koji je izведен i koji pruža usluge sukladno normiranim pravilima te definiranoj sintaksi i semantici. Norma, tj. standard, je specifikacija koja je široko prihvaćena u industriji (*de facto standard*) ili zastupana od priznatog normizacijskog tijela (*de jure standard*), kao što se primjerice IETF (*Internet Engineering Task Force*, [www.ietf.org](http://www.ietf.org)) ili ITU-T (*International Telecommunications Union's Telecommunication Standardization Sector*, [www.itu.int](http://www.itu.int)). Takva specifikacija treba biti dobro definirana, javno dostupna i neutralna, tj. vlasnički neovisna kako bi se mogla primjenjivati bez ograničenja.

Otvorenost je prepostavka za međudjelovanje različitih sustava (engl. *interoperability*), prenosivost rješenja iz jednog sustava u drugi (engl. *portability*), kao i unaprjeđenje i prilagodbu zahtjevima korisnika tijekom njihovog životnog vijeka.

### 1.2.2 Transparentnost

Transparentnost označava sposobnost prikrivanja odabranih značajki raspodijeljenog sustava. Razlikuju se sljedeći aspekti:

1. transparentnost pristupa,
2. lokacijska transparentnost,
3. migracijska transparentnost,
4. relokacijska transparentnost,
5. replikacijska transparentnost,
6. konkurencijska transparentnost i
7. transparentnost na kvar.

**Transparentnost pristupa** (engl. *access transparency*) postiže se prikrivanjem razlika u pristupu sredstvima i predočavanju podataka, primjerice različitim arhitektura računala, različitim operacijskim sustavima, ili različitim baza podataka. Ako je ostvarena transparentnost pristupa, korisnik neće uočiti heterogenost raspodijeljenog sustava. Primjerice, programski jezik Java omogućuje razvoj raspodijeljenih sustava koji su mogu izvoditi u heterogenoj okolini, tj. na računalima s različitim operacijskim sustavim, i time postiže transparentnost pristupa.

**Lokacijska transparentnost** (engl. *location transparency*) odnosi se na prikrivanje lokacije sredstva, tako da položaj sredstva u sustavu ne treba biti i nije poznat korisniku. Lokacijska se transparentnost postiže ako je omogućen dostup sredstvu putem njegovog naziva. Npr. u sustavu weba to se postiže simboličkim označavanjem poslužitelja s informacijskim resursima, kao što je [www.fer.unizg.hr](http://www.fer.unizg.hr), čiju lokaciju – IP-adresu zna sustav imenovanja domena, DNS (*Domain Name System*). Međutim, to ne znači da je poznata i prostorna lokacija resursa, iako se u mnogo slučajeva može prepostaviti (npr. kućna adresa ustanove).

**Migracijska transparentnost** (engl. *migration transparency*) označava prikrivanje promjene lokacije sredstva, tako da promjena lokacije sredstva ne utječe na način dostupa sredstvu. Premještanje sredstva neće utjecati na dostup sredstvu ako se ono ostvaruje putem naziva. Uzmimo još jednom gornji primjer: korisnik neće uočiti registraciju nove IP-adrese poslužitelja u DNS-u za njemu poznati simbolički naziv.

**Relokacijska transparentnost** (engl. *relocation transparency*) zahtjevnija je od migracijske, jer se traži prikrivanje premještanja sredstva tijekom njegove uporabe, tako da je sredstvo dostupno i može se upotrebljavati tijekom njegovog premještanja. Takav oblik transparentnosti zahtjeva posebne funkcije upravljanja pokretljivošću (engl. *mobility management*) koje omogućuju ustanovljavanje trenutne lokacije, npr. u pokretnoj mreži (engl. *mobile network*) ili mrežne adrese u IP-mreži pomoću protokola *Mobile IP*.

**Replikacijska transparentnost** (engl. *replication transparency*) označava prikrivanje više istovrsnih sredstava ili više preslika nekog sredstva, što omogućuje, s motrišta korisnika, primjenu istog naziva za sve replike. Ako je riječ o repliciranim sredstvima koja rukuju s podacima (datoteke, baze podataka) potrebni su posebni mehanizmi za održavanje više replika istih podataka i očuvanje njihove konzistentnosti.

**Konkurencijska transparentnost** (engl. *concurrency transparency*) potrebna je kako bi se prikrila istodobna uporaba istog sredstva od strane više korisnika te kako bi svaki korisnik imao dojam da sustav poslužuje samo njega. Zajednička, odnosno uporaba sredstva dijeljena s drugim korisnicima zahtjeva isto tako očuvanje konzistentnosti te dobre poslužiteljske performance sustava da bi svi korisnici imali očekivanu ili ugovorenu kvalitetu usluge (npr. vrijeme odziva, trajanje transakcije i sl.).

**Transparentnost na kvar** (engl. *failure transparency*) odnosi se na rad sustava uz pojavu kvara. Kvar treba prikriti, tako da korisnicima ne bude uočljiv, kao i postupci otkrivanja kvara i obnavljanja sustava nakon kvara. Problem otkrivanja kvara je težak, jer se i veliko opterećenje ili preopterećenje sustava može očitovati kao kvar (npr. nema odgovora u očekivanom vremenu). Nadalje, kvar se ne mora očitovati samo kao ispad dijela sustava, već i kroz njegov neispravan rad. Transparentnost na kvarove ne može se postići bez više istovrsnih sredstava koja mogu preuzeti isti posao, što se povezuje s replikacijskom transparentnosti.

### 1.2.3 Skalabilnost

Skalabilnost označava sposobnost razmjerne prilagodbe sustava broju korisnika (količina sredstava primjerena broju korisnika) i njihovoj rasprostranjenosti (lokalno, regionalno, globalno, ...), te omogućuje neometano uvođenje novih funkcija u postojeći sustav i nove načine upravljanja u jednoj ili više administrativnih domena.

Raspodijeljeni sustavi bolji su s motrišta skalabilnosti od cjelovitih, jer se mogu proširivati razmjerno porastu broja korisnika i njihovom prostornom rasporedu i tako provoditi usklađivanje kapaciteta sustava s korisničkim zahtjevima. Početnu izvedbu sustava nije potrebno dimenzionirati za pretpostavljeni krajnji broj korisnika, što znatno utječe na smanjenje troškova izgradnje i održavanja sustava. Isto tako, nove funkcije mogu se dodijeliti postojećim ili uvesti nove dijelove sustava, a prema potrebama korisnika. Slično je i s upravljanjem: upravljanje sustavom može se lakše uskladiti s njegovom organizacijom.

Kako bi se uz promjenu broja korisnika održale performance sustava uz prihvatljive troškove potrebno je:

- više (istovrsnih) dijelova,      *Koliko?*
- prostorno raspodijeljenih i      *Gdje?*
- koji komuniciraju.                *Kako?*

Primjeri neskalabilnih rješenja su centralizirana usluga (jedan poslužitelj za sve korisnike), centralizirani podaci (jedan poslužitelj sa svim korisničkim podacima), ili centralizirani algoritam (svi podaci o sustavu poznati, npr. usmjeravanje na temelju poznavanja stanja cijele mreže).

Tehnike koje omogućuju skalabilnost sustava su sljedeće:

- prikrivanje kašnjenja u komunikaciji: npr. asinkrona komunikacija ("radi nešto korisno dok čekaš odgovor"),
- višestrukost: više dijelova koji omogućuju funkcionalnost sustava (npr. raspodijeljena baza podataka, sustavi *peer-to-peer*)
- replikacija: istovjetne kopije podataka, uz očuvanje konzistentnosti izvornika i preslike.

#### 1.2.4 Kvaliteta usluge

Svaki sustav određuju njegova funkcionalna i nefunkcionalna obilježja. Funkcionalna obilježja govore „što“ sustav radi, za što je namijenjen, kakve usluge pruža korisnicima, odnosno o njegovoj funkcionalnosti. Nefunkcionalna obilježja, s druge strane opisuju „kako“ sustav radi, koliko brzo poslužuje korisnički zahtjev, je li raspoloživ, koliko je pouzdan i siguran. Zajednički naziv za opis nefunkcionalnih obilježja je kvaliteta usluge (engl. *Quality of Service*, QoS). Čovjek izražava subjektivni dojam o kvaliteti usluge, primjerice „transakcija je provedena jako brzo“, „dugo se učitava stranica weba“, „dокумент nije potpun“, „treba više puta ponavljati isto“, „ne radi baš kad mi treba“ i slično.

Kvaliteta usluge se definira kao zajednički učinak performansi usluge koji određuje stupanj zadovoljstva korisnika. Subjektivno zadovoljstvo korisnika pruženom uslugom opisuje se pojmom iskustvene kvalitete (engl. *Quality of Experience*, QoE), dok se QoS odnosi na tehničke parametre vezane uz performance sustava (engl. *system performance*) koji utječu na QoE, i razlogom su za „dobru“ ili „lošu“ kvalitetu usluge. Primjerice, sustav loših obradnih performansi neće moći provesti transakciju dovoljno brzo, tj. sa zadovoljavajućim vremenom odziva, a u slučaju velikog opterećenja neće moći poslužiti sve korisnike – imat će se dojam da sustav ne radi.

Oblikovanjem raspodijeljenog sustava treba odgovoriti na sljedeće pitanja:

- Kakve funkcionalne zahtjeve treba ostvariti – ŠTO sustav treba raditi?
- Temelji li se sustav na otvorenim rješenjima i kojima?
- 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?
- Kakve nefunkcionalne zahtjeve treba ostvariti – KAKO sustav treba raditi (kakva se kvaliteta usluge zahtijeva)?

Za raspodijeljene sustave posebno su važna sljedeća nefunkcionalna obilježja: performance iskazane s parametrima vrijeme odziva, propusnost i kapacitet, zatim pouzdanost (engl. *reliability*) i raspoloživost (engl. *availability*) te ukupni trošak vlasništva (engl. *Total Cost of Ownership*, TCO).

### 1.3 Arhitektura raspodijeljenih sustava

Arhitektura raspodijeljenog sustava definira njegove sastavne dijelove i njihove odnose te način obavljanja funkcija kojima je namijenjen.

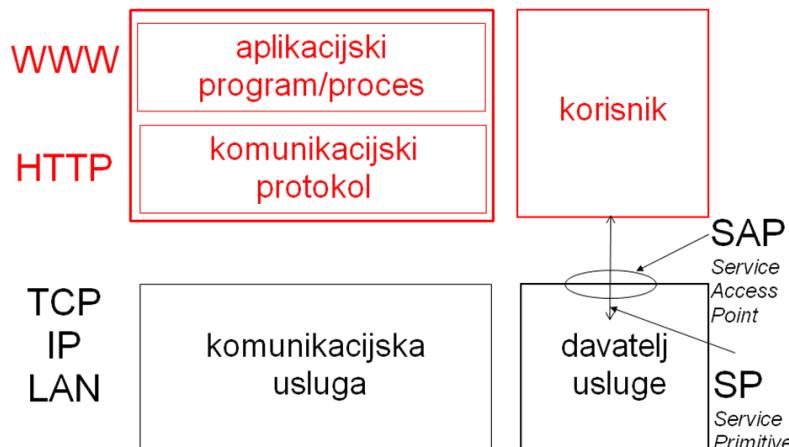
Programska arhitektura određuje logičku strukturu sustava, njegove programske komponente, njihovu organizaciju i međusobnu interakciju. Sustavska arhitektura pokazuje smještaj programskih komponenata na jedno računalo (centralizirana arhitektura) ili više njih (decentralizirana arhitektura), kao u slučaju raspodijeljenog sustava.

Osnovni načini predočavanja arhitekture raspodijeljenog sustava izvedeni su iz slojevitih i objektnih modela te modela zasnovanih na podacima i događajima.

### 1.3.1 Slojeviti arhitekturni model

Slojeviti modeli razvijeni su za rješavanje povezivanja otvorenih sustava i široko se primjenjuju u komunikacijskim sustavima i mrežama. Sustav se vertikalno dekomponira na slojeve, a svakome se sloju dodjeljuju funkcije i specificiraju sučelja sa susjednim slojevima, kako bi viši sloj mogao koristiti usluge nižeg sloja. Osnovna je zamisao da svaki sloj nudi usluge višem sloju, prikrivajući rješenje usluga koje omogućuju niži slojevi. Najviši je uvijek aplikacijski sloj koji predočuje usluge i aplikacije za korisnike, a najniži fizički sloj koji omogućuje stvarni prijenos informacija fizičkim medijem. Osnovni slojeviti modeli su referentni model povezivanja otvorenih sustava (engl. *Open System Interconnection Reference Model*, OSI RM) te referentni model TCP/IP, odnosno internetski model (engl. *TCP/IP Reference Model*).

Funkcionalnost raspodijeljenog sustava određuje najviši, aplikacijski sloj koji sadrži aplikacije i usluge namijenjene korisnicima. Niži slojevi pružaju usluge aplikacijskim procesima kojima se ostvaruje njihovo povezivanje, tj. razmjena podataka i suradnja potrebna kako bi se poslužili korisnici. Korisnik (engl. *user*) je entitet u međudjelovanju s drugim entitetom, davateljem usluge (engl. *service provider*). Točka međudjelovanja između dva susjedna sloja naziva se točkom pristupa usluzi (engl. *Service Access Point*, SAP), a samo međudjelovanje opisuje se uslužnim primitivima (engl. *Service Primitive*, SP) određenih nazivom i skupom parametara. Usluga se definira kao ponašanje davatelja usluge u točki pristupa usluzi opisano skupom uslužnih primitiva i njihovim redoslijedom. Komunikacijski protokol opisuje način ostvarivanja usluge. Pokažimo to na primjeru weba (Slika 1.2).



Slika 1.2. Odnos korisnika i davatelja usluge za aplikaciju weba

Aplikacijski program za web je pretraživač (engl. *browser*) s ulogom klijenta pomoću kojeg se pokreću odgovarajući procesi kojima se pristupa informacijskim sredstvima na odabranim poslužiteljima koji dostavljaju informacijski sadržaj. Aplikacijski protokol weba je HTTP (*Hyper Text Transfer Protocol*). Tako strukturirani aplikacijski sloj weba na strani klijenta i poslužitelja, korisnik je komunikacijske usluge koju mu pružaju niži slojevi koji uključuju transportni protokol TCP (*Transmission Control Protocol*), mrežni protokol IP (*Internet Protocol*) te protokole lokalne mreže (engl. *Local Area Network*, LAN) putem koje se korisničko računalo spaja na Internet. Pojednostavljenno se može reći da

TCP, IP i LAN čine davatelja usluge za aplikaciju weba kao korisnika. Korisnici (klijent i poslužitelj) pristupaju davatelju usluge u točki pristupa usluzi, SAP, primjenom uslužnih primitiva, SP.

Povezanost dva procesa u aplikacijskom sloju naziva se asocijacijom (zdrživanjem), umjesto vezom (konekcijom), kako bi se naglasila slaba povezanost entiteta: pri razmjeni podataka primatelj a-priori ne mora znati tko je pošiljatelj, a raspoloživost i pouzdanost asocijacije mogu varirati.

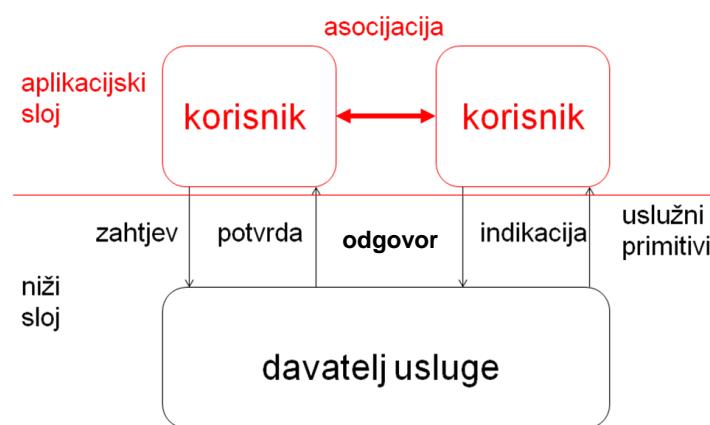
Dva se procesa u aplikacijskom sloju združuju da bi komunicirali i razmijenili podatke, stvarajući asocijaciju posredovanjem nižih slojeva koji imaju ulogu davatelja usluge, putem kojih se ostvaruje komunikacija između računala na kojima su smješteni (Slika 1.3). Procesi su korisnici usluga nižih slojeva, pri čemu se u općem slučaju koriste ovi uslužni primitivi:

- zahtjev (engl. *request*) – upućuje ga korisnik kako bi od davatelja usluge zatražio izvršenje određene funkcije.
- indikacija (engl. *indication*) – upućuje ga davatelj usluge kako bi od korisnika zatražio izvršenje određene funkcije ili ga izvijestio da je od drugog korisnika primio zahtjev.
- odziv (engl. *response*) – upućuje ga korisnik kao odgovor na prethodno primljenu indikaciju.
- potvrda (engl. *confirm*) – upućuje ga davatelj usluge kao odgovor na zahtjev.

Redoslijed primitiva je ovakav:

1. Proses – korisnik koji pokreće asocijaciju (npr. klijent) postavlja nižem sloju (davatelju usluge spajanja) primitiv zahtjev za spajanjem na drugi proces – drugog korisnika (npr. poslužitelj).
2. Niži sloj to prosljeđuje do drugog korisnika primitivom indikacija kojim označava da se traži spajanje.
3. Taj korisnik, ako može provesti spajanje, to označava primitivom odgovor kako bi niži sloj o tome mogao izvijestiti korisnika pokretača spajanja.
4. Niži sloj označava primitivom potvrda da je spajanje uspješno provedeno, čime je asocijacija uspostavljena i razmjena podataka započinje.

U ovim se primjerima pod pojmom „nižeg sloja“ za aplikacijski sloj podrazumijevaju svi niži slojevi do uključujući fizičkog. Treba napomenuti da je jedna od pogodnosti primjene slojevitih modela upravo to: usredotočiti se na odabrane funkcije (aplikacijski sloj), a ostale apstrahirati (tretirati ih kao niži sloj ili slojeve). Primjeri asocijacija su klijent-poslužitelj (npr. web), poziv (npr. govorna komunikacija) i posredovanje (npr. elektronička pošta).

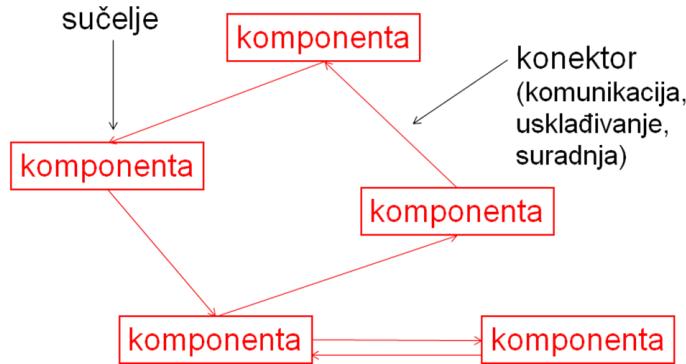


Slika 1.3. Slojevita arhitektura sustava s motrišta asocijacije u aplikacijskom sloju

### 1.3.2 Objektni arhitekturni model

U osnovi slojevitih arhitekturnih modela je funkcionalna podjela, dok su za objektne arhitekture od primarne važnosti gradbene jedinice od kojih se sustav sastoji. Objekti ili komponente sastavljene od

više povezanih objekata s definiranom zadaćom, dijelovi su sustava s dobro definiranim sučeljem (Slika 1.4). Njihovim povezivanjem konektorima postiže se međusobna komunikacija, usklađivanje rada i suradnja.

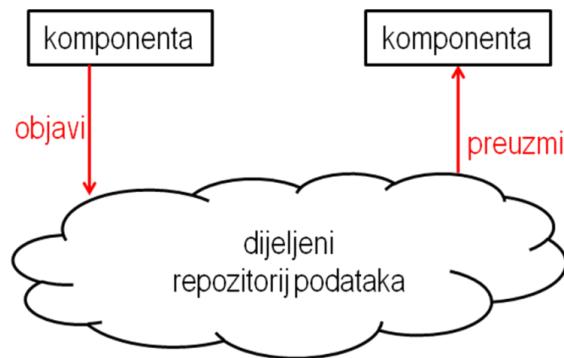


Slika 1.4. Objektna arhitektura sustava

### 1.3.3 Arhitektura zasnovana na podacima

U ovom arhitekturnom modelu procesi komuniciraju putem dijeljenog repozitorija podataka (Slika 1.5). Komponenta sustava koja raspolaže podatkom upisuje ga (objavljuje) u repozitorij podataka koji omogućuje čitanje (preuzimanje) podatka drugim komponentama.

Za raspodijeljene sustave zanimljivi su isto takvi, raspodijeljeni, repozitoriji podataka.

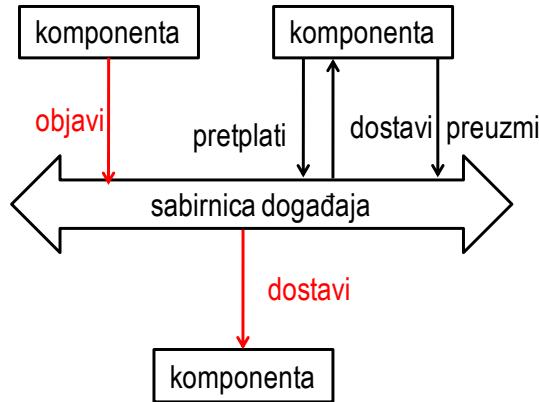


Slika 1.5. Arhitektura zasnovana na podacima

### 1.3.4 Arhitektura zasnovana na događajima

U ovom arhitekturnom modelu procesi komuniciraju razmjenom tzv. događaja (engl. event) kojima se oglašava raspoloživost podataka. Komponenta se može pretplatiti na podatak koji joj je potreban. Kada neka komponenta objavi taj podatak (događaj!), podatak će se dostaviti (događaj!) komponenti koja je na njega pretplaćena, tako da ga može preuzeti (Slika 1.6).

Ovakvo se rješenje naziva sustavom objavi-pretplati (engl. publish-subscribe).



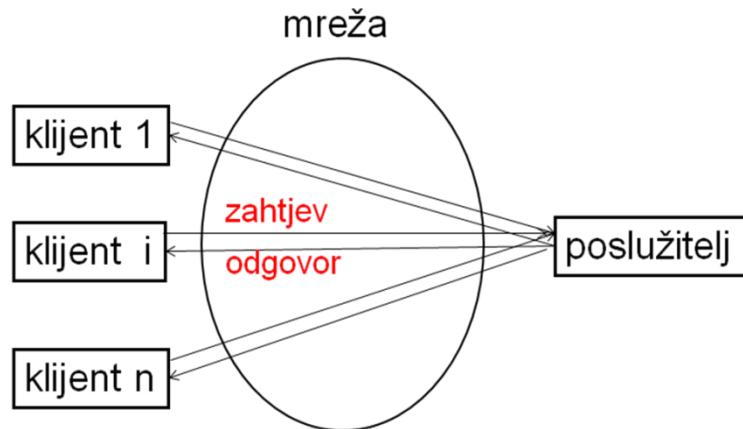
Slika 1.6. Arhitektura zasnovana na događajima

## 1.4 Osnovni modeli raspodijeljene obrade

Uvodno će se razraditi tri modela raspodijeljene obrade i to klijent-poslužitelj, ravnopravni sudionici te pokretni kod i programski agenti.

### 1.4.1 Model klijent-poslužitelj

U modelu klijent-poslužitelj (engl. *client-server*) razlikuju se dvije vrste entiteta, klijenti i poslužitelji. Klijent traži uslugu od poslužitelja, a poslužitelj je pruža za više, u pravilu mnogo klijenata uz svojstvo konkurenčijske transparentnosti (Slika 1.7).

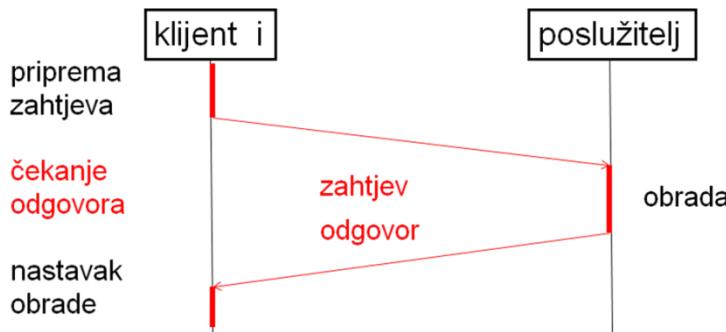


Slika 1.7. Model klijent-poslužitelj

Komunikacija klijenta i poslužitelja odvija se tako da klijent šalje zahtjev za uslugom poslužitelju i čeka odgovor. Poslužitelj prihvata i obrađuje zahtjev te vraća odgovor klijentu što je prikazano slijednim dijagrameom na slici (Slika 1.8). Klijent i poslužitelj sukladno definiciji raspodijeljenog sustava „komuniciraju i usklađuju svoje aktivnosti isključivo razmjenom poruka“ – zahtjeva i odgovora.

Obradni model klijent-poslužitelj, kao što je već rečeno, primijenjen je u webu. Ovaj je model intuitivno jasan, često ga se susreće u stvarnom životu i različitim djelatnostima, ne samo informacijskim, komunikacijskim ili računalnim.

Funkcionalnost modela klijent-poslužitelj za neku primjenu često je odrediti mnogo lakše nego performance sustava potrebne kako bi korisnik dobio očekivanu kvalitetu usluge. Pitanja su mnoga, primjerice, kako broj istodobno aktivnih korisnika (klijenata) i njihova prostorna raspodjela utječe na vrijeme odziva, kako obilježja usluge koju pruža poslužitelj utječu na trajanje obrade, kakva se složenost klijentskog programa preferira. To sve ukazuje na važnost nefunkcijskih zahtjeva.



Slika 1.8. Komunikacija klijenta i poslužitelja

Osnovna interakcija klijenta i poslužitelja uključuje pripremu i odašiljanje zahtjeva te čekanje odgovora od strane klijenta. Poslužitelj primljeni zahtjev obrađuje i vraća odgovor klijentu koji nakon toga nastavlja s obradom. Koje probleme možemo uočiti u ovom osnovnom modelu? Navedimo ih:

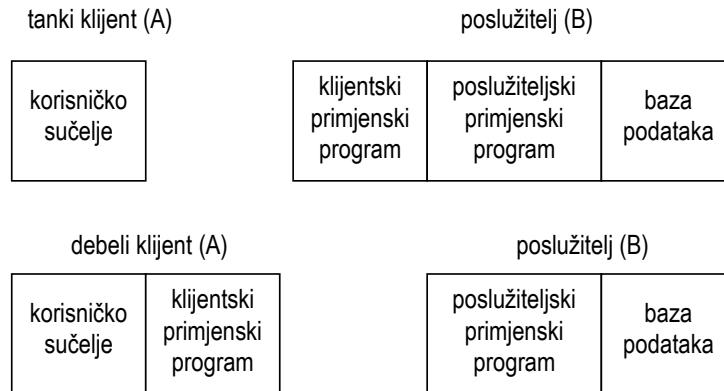
- komunikacija je sinkrona: klijent čeka odgovor i za to vrijeme ne radi ništa;
- poslužitelj može odmah obraditi zahtjev samo ako je slobodan, tj. ako ne poslužuje zahtjev nekog drugog klijenta, a inače će zahtjev čekati na obradu na poslužitelju;
- treba održati mrežnu povezanost klijenta i poslužitelja tijekom interakcije: dulje čekanje zahtjeva veze duljeg trajanja;
- moguća je informacijska asimetrija: količina podataka u odgovoru može biti velika, u odnosu na u pravilu kratki zahtjev, što izaziva veliku uporabu sredstava na strani klijenta za prihvatanje odgovora.

U modelu klijent-poslužitelj svi klijenti koriste isto sučelje na poslužitelju koje definira skup zahtjeva koje mogu postaviti. Funkcionalnost poslužitelja definirana je unaprijed tijekom oblikovanja sustava: poslužitelj mora raspolagati svim sredstvima (programskim, informacijskim) potrebnima za posluživanje zahtjeva. Programska izvedba sustava klijent-poslužitelj obuhvaća sljedeće:

- korisničko sučelje, ako je korisnik čovjek (npr. kod weba, za unos zahtjeva za nekim informacijskim sredstvom, pregled stranice odgovora i drugo),
- klijentski primjenski program (npr. kod weba, pretraživač koji na temelju zahtjeva generira upite poslužitelju, preuzima stranice i predložuje ih korisniku),
- poslužiteljski primjenski program (npr. kod weba, poslužitelj za dohvatanje traženih informacijskih sredstava ili podataka iz baze podataka) i
- bazu podataka s informacijskim sadržajima.

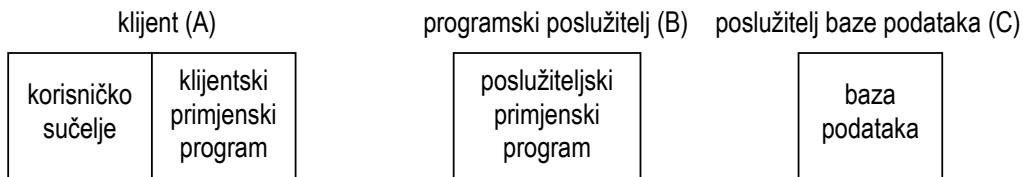
Korisničko sučelje, klijentski i poslužiteljski program te baza podataka mogu biti na različite načine raspodijeljeni između klijenta i poslužitelja.

Osnovni model klijent-poslužitelj predložen je dvorednom fizičkom arhitekturom (engl. *two-tier*), s redom klijenta (računalo s potporom za klijenta) i redom poslužitelja (računalo s potporom za poslužitelja) kojima se korisničko sučelje, primjenski programi i baza podataka mogu dodijeliti na različite načine. Tako klijent može sadržavati samo dio ili cijelo korisničko sučelje. Takav se klijent s minimalnom funkcionalnosti naziva mršavim klijentom (engl. *thin client*). Klijent složenje funkcionalnosti, sa sučeljem i primjenskim programom, ili čak sučeljem s primjenskim programom i dijelom baze podataka namijenjenih korisniku, naziva se debelim klijentom (engl. *fat client*). Rješenje ovisi o namjeni sustava i raspoloživim računalnim sredstvima (npr. za mali pokretni uređaj s ograničenim izvorom napajanja – baterijom, prikladnije je rješenje tankog klijenta). Poslužiteljski red sadrži vlastiti primjenski program i bazu podataka, te primjenski program tankog klijenta ako je riječ o takvoj izvedbi (Slika 1.9).



Slika 1.9. Dvoredna fizička arhitektura klijent-poslužitelj

Ako se poslužiteljski red izvede s tri odvojena računala – poslužitelja riječ je o trorednoj fizičkoj arhitekturi (engl. *three-tier*), kao primjeru višerednih arhitektura (Slika 1.10). U tom slučaju programski poslužitelj (srednji red) djeluje kao poslužitelj prema klijentu, a kao klijent prema poslužitelju baze podataka. Gubi se oštra razlika između klijenta i poslužitelja, jer "A" šalje zahtjev "B", a "B" traži uslugu od "C" kako bi obavio dio zadatka dobivenog od "A". "B" je poslužitelj od "A" i klijent od "C" – uvodi se posredovanje između dva surađujuća entiteta: "B" posreduje između "A" i "C".



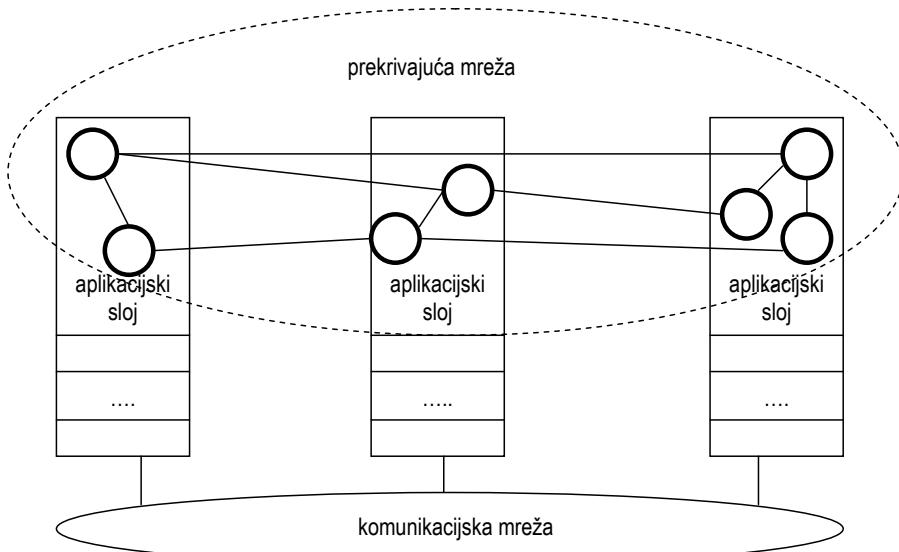
Slika 1.10. Troredna fizička arhitektura klijent-poslužitelj

#### 1.4.2 Model ravnopravnih sudionika

Ravnopravnim sudionikom ili entitetom (engl. *peer*) naziva se onaj primjenski program koji može djelovati i kao klijent i kao poslužitelj. Ravnopravni sudionici tvore sustav u kojem međusobno komuniciraju (engl. *Peer-to-Peer*, P2P) tako da se na aplikacijskom sloju povezuju u "prekrivajuću mrežu" (engl. *overlay network*) čvorova – ravnopravnih sudionika (Slika 1.11).

Prekrivajuća mreža ravnopravnih sudionika izvedena je „iznad“ komunikacijske mreže koja povezuje računala na kojima se izvode *peer* procesi. Svaki *peer* "plaća" sudjelovanje u prekrivajućoj mreži nudeći dio vlastitih sredstava ostalima, tako da model ravnopravnih sudionika potencijalno nudi neograničena sredstva u velikim mrežama.

Sustav P2P decentralizirani je raspodijeljeni sustav u kojemu nema središnje koordinacije ravnopravnih entiteta. Ne postoji središnja točka koja bi upravljala sustavom, pa tako niti točka ispadala koja bi ugrozila rad sustava u slučaju kvara. Prekrivajuća mreža je samoorganizirajuća mreža *peerova*, tj. sudionika koji su međusobno neovisni te ulaze i izlaze iz sustava po volji.



Slika 1.11. Sustav P2P

#### 1.4.3 Modeli premještanja programa i programske agenata

Premještanje programa s jednog na drugo umreženo računalo zahtjeva posebna rješenja za migraciju programskega kôda (engl. *code migration*). Zamislimo nekoliko situacija i moguća rješenja.

Klijent raspolaže programskim kôdom (zna kako riješiti zadaću), međutim nema potrebna sredstva za izvedbu. Umjesto slanja zahtjeva, kôd se šalje poslužitelju od kojeg se zahtjeva izvođenje i vraćanje odgovora s rezultatom klijentu. To je postupak kretanja procesa (engl. *process migration*) koji koristi metodu udaljenog izvršavanja (engl. *remote evaluation*) i temelji se na dostavi procesa koji se trebaju izvesti udaljenom poslužitelju. U ovom slučaju, uz podatke, mrežom se prenosi i proces u određenom stanju, kojeg je potrebno izvršiti, a poslužitelj vraća izvršeni proces s rezultatima.

Klijent ne raspolaže izvršnim kôdom, već kad se javi potreba za obradom (zna koju zadaću treba riješiti), šalje zahtjev poslužitelju koji vraća traženi program iz svog repozitorija. To je postupak dostave kôda na zahtjev (engl. *code on demand*), s poslužitelja koji sadrži repozitorij programa. Nakon što se programski kôd s poslužitelja preseli na klijenta, pokreće se obrada, i nakon izvođenja, programski kôd se briše s klijenta. Tipični primjeri ovog koncepta su Java *applets* koji se preuzimaju s poslužitelja i izvode lokalno na klijentu.

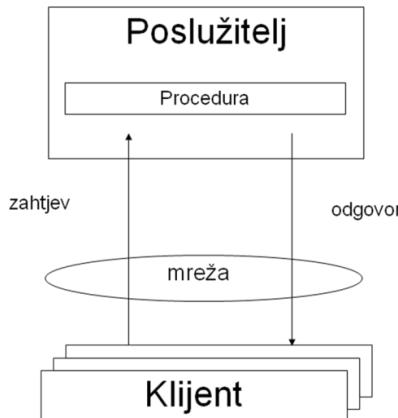
Kod metode aktivne poruke (engl. *active message*) poruka koja sadrži kôd programa kreće se od čvora do čvora, da bi se kôd izveo na zadanom čvoru. U objektnoj paradigmi tome odgovara koncept pokretnog objekta.

Migracija procesa s jednog računala na drugo može se provesti i iz drugih razloga kao što su uravnoteženje opterećenja, uvođenje konkurentnosti i proširenje funkcionalnosti.

Poseban je slučaj pokretnog programskega agenta (engl. *mobile software agent*) ili kraće pokretnog agenta (engl. *mobile agent*). Općenito, programski agent (engl. *software agent*) je program koji obavlja neki posao za svog korisnika ili vlasnika, a raspolaže svojstvima kao što su inteligencija, samostalnost, reaktivnost, proaktivnost, društvenost i druga. Pokretljivost (engl. *mobility*) je dodatno svojstvo koje zahtjeva posebnu programsku izvedbu agenta i okružja u kojem se kreće.

Ako agent svoju zadaću obavlja stalno na istom mjestu u informacijskom prostoru, ili izvedbeno gledajući, na istom računalu ili čvoru mreže, naziva se stacionarnim (engl. *stationary agent*). Agent koji se tijekom svog djelovanja može premjestiti na drugo računalo ili čvor u mreži i tako migrirati s mjesta na mjesto, je pokretni agent. Inteligentni pokretni agent djeluje samostalno, pa tako može i

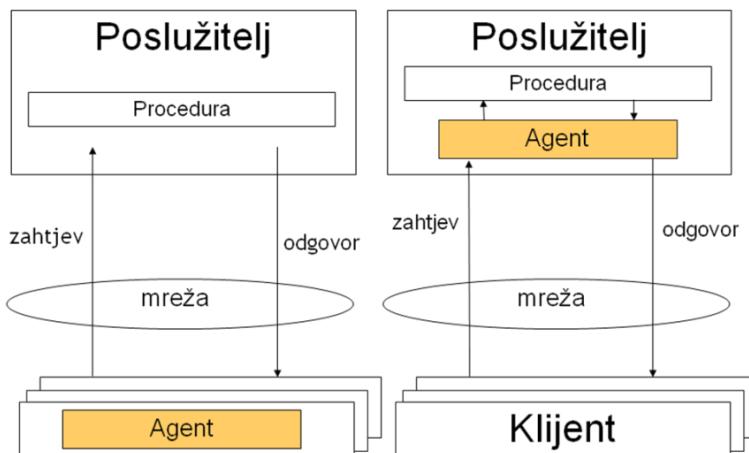
samostalno migrirati ovisno o stanju mreže i umreženih sustava na kojima može obaviti povjereni posao i/ili njegov dio.



Slika 1.12. Pozivanje udaljene procedure

Programsku izvedbu pokretnog agenta i agentske platforme – okružja u kojem agenti djeluju i kreću se objasnit će se polazeći od programskega rješenja modela klijent-poslužitelj s pozivanjem udaljene procedure (engl. *Remote Procedure Call*, RPC), kao što je prikazano na slici (Slika 1.12). Klijent upućuje zahtjev kojim poziva udaljenu proceduru koja se izvršava na poslužitelju. U zahtjevu se nalaze podaci i parametri koji su potrebni za izvođenje udaljene procedure. Poslužitelj vraća odgovor s podacima, rezultatom obrade.

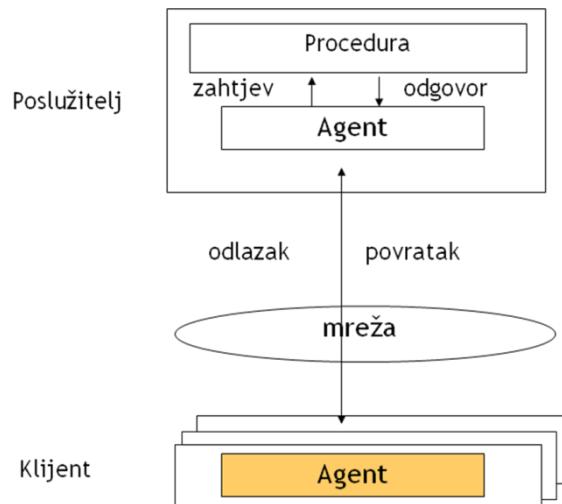
Ovakav se programski model može primjeniti za stacionarne agente, pri čemu agent može zastupati klijenta ili poslužitelja (Slika 1.13), a isto tako oba mogu imati svoje agente.



Slika 1.13. Stacionarni agent na klijentskoj i poslužiteljskoj strani

Pozivanjem udaljene procedure se kroz mrežu razmjenjuju podaci (zahtjev, odgovor) između dva programska entiteta od kojih jedan (klijent ili njegov agent) traži od drugog (poslužitelj ili njegov agent) da za njega obavi posao koji sam ne zna.

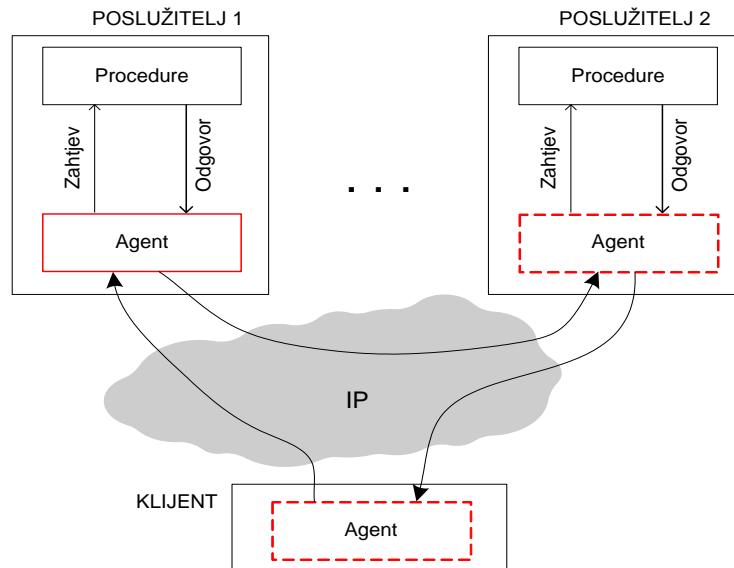
Programski model pokretnog agenta objedinjava prethodno navedene pristupe premještanju programa. Pokretni agent je programski entitet koji se šalje od klijenta prema poslužitelju, a sadrži podatke koje treba obraditi i programski kod koji treba izvesti, sa stanjem. Pokretni agent se ne mora vratiti klijentu nakon obavljenog posla na jednom poslužitelju, već može posjetiti sljedećeg ili više njih. Sva komunikacija između agenta i poslužitelja (zahtjevi i odgovori) odvija se lokalno, tako da nema dodatnog mrežnog opterećenja zbog razmjene podataka između klijenta i poslužitelja. Takav se programski model predložen slikom (Slika 1.14) naziva udaljenim programiranjem (engl. *Remote Programming*, RP).



Slika 1.14. Udaljeno programiranje

Prednost pokretljivosti je u tome što agent migrirajući s čvora na čvor u mreži, na svakom čvoru ili više njih može obaviti dio posla te se na kraju vratiti s rezultatom obrade (Slika 1.15).

Programska infrastruktura potrebna za izvedbu i izvršavanje agenata naziva se agentskom platformom (engl. *agent platform*). Agentska platforma treba biti postavljena na svakom čvoru koji udomljuje agente. Najviše je agentskih platformi razvijeno u jeziku Java čime je omogućena uporaba neovisna o korištenom operacijskom sustavu, a s motrišta prenosivosti rješenja pokrivaju Internet, odnosno IP-mreže.

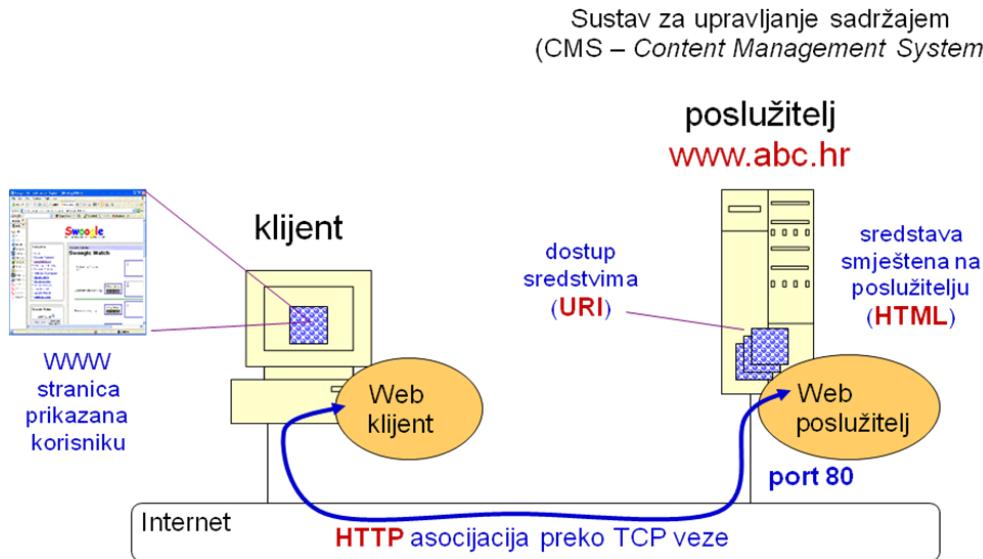


Slika 1.15. Migracija i izvođenje pokretnog agenta

## 1.5 Studijski primjer: web

WWW je stvoren i razvija se kao otvoreni sustav s transparentnim pristupom i konkurenčijskom transparentnosti. Usluge pruža velikom broju korisnika predočenih klijentima sukladno normiranim pravilima te definiranoj sintaksi i semantici, uz očuvanje konzistentnosti sredstva. Temeljne norme za web utvrđuju i razvijaju dva tijela: Internet Engineering Task Force, IETF (<http://www.ietf.org>) i World Wide Web Consortium, W3C (<http://www.w3c.org>).

Podsjetimo se kako web izgleda i kako radi (Slika 1.16).



Slika 1.16. Ilustracija rada weba

Riječ je o modelu klijent-poslužitelj kod kojeg preglednik, tj. web-klijent, sadrži korisničko sučelje i programsku logiku za pristup poslužitelju i prikaz dobavljenog informacijskog sadržaja. Komunikacija klijenta i poslužitelja obavlja se putem Interneta pri čemu se uspostavlja HTTP-asocijacija preko TCP-veze čime se postiže točna i pouzdana razmjena podataka. U aplikacijskom sloju primijenjen je protokol HTTP, a u transportnom TCP. Dostup sredstvima, u ovom slučaju informacijskom sadržaju smještenom na poslužitelju i zapisanom u formatu HTML (*HyperText Markup Language*) obavlja se putem jednoznačnog identifikatora, URI (*Uniform Resource Identifier*). Svako je sredstvo jednoznačno identificirano svojim URI-jem. Stvaranje, uređivanje i održavanje informacijskog sadržaja obavlja se sustavom za upravljanje sadržajem (engl. *Content Management System*, CMS). CMS omogućuje pripremu sadržaja, automatsko generiranje navigacijskih elemenata, indeksiranje i pretraživanje, uspostavljanje korisničkih prava i praćenje uporabe, definiranje sigurnosnih postavki i drugo.

Već smo utvrdili da web ispunjava zahtjeve otvorenog raspodijeljenog sustava. Kakav je web s motrišta transparentnosti? Transparentnost pristupa i konkurencijska transparentnost su ostvarene njegovim osnovnim postavkama. Sama zamisao „hiperteksta“ i normiranog jezika za označavanje (HTML) ključna je za transparentnost pristupa, dok je „konkurenca“ korisnika, tj. klijentskih zahtjeva, inherentna modelu klijent-poslužitelj.

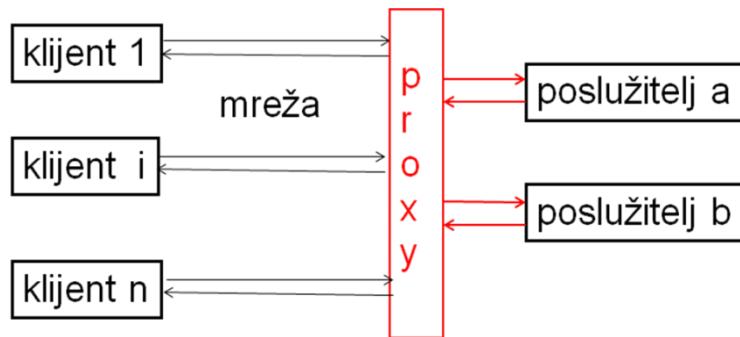
Lokacijska transparentnost web-poslužitelja postiže se simboličkim imenima koja se u sustavu imenovanja domena (DNS) prevode u lokacije poslužitelja (mrežne adrese). Korisnik rabi simboličke nazive, tako da mu položaj poslužitelja (sjedišta weba), kao i bilo kojeg sredstva ne treba biti poznat.

Migracijska transparentnost web-poslužitelja je isto tako omogućena: ako se ne mijenja simbolički naziv, nego samo položaj i/ili mrežno sučelje, mijenja se samo mrežna adresa (lokacija poslužitelja) u DNS-u.

Relokacijska transparentnost se ne zahtijeva, jer je poslužitelj stacionaran i ne kreće se tijekom pružanja usluge, dok model posluživanja ne pamti stanje klijentskog zahtjeva (engl. *stateless*).

**Replikacijska transparentnost** može se postići ako se više web-poslužitelja s istim informacijskim sadržajem „sakrije“ iza jednog simboličkog naziva. U takvoj će situaciji korisnik adresirati „simboličkog“ web-poslužitelja, a poseban uređaj, zastupnik stvarnih poslužitelja u mreži (engl. proxy) će zahtjev usmjeriti na jednog od fizičkih poslužitelja. U primjeru na slici (Slika 1.17), klijenti putem zastupnika pristupaju poslužiteljima *a* ili *b*, a pri tome imaju dojam da odgovori pristižu isključivo sa zastupnika.

Ovakvo rješenje prikladno je i za razmjeri rast poslužiteljskih kapaciteta, jer se mogu dodavati novi, odnosno replicirati, poslužitelji kako raste broj korisnika i frekvencija njihovih zahtjeva.



Slika 1.17. Pristup webu putem zastupnika – proxyja

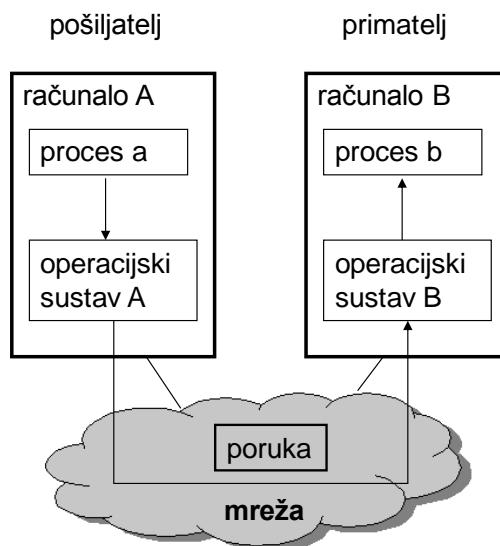
Treba uočiti da web uz širenje primjene doživljava i funkciju transformaciju od "korisnik čita sadržaj" prema "korisnik stvara sadržaj", što se naziva inačicom Web 2.0, te od "korisnik odabire sadržaj" prema "korisnik traži uslugu", čemu odgovara koncept usluga weba (engl. *Web Service*), kojima se detaljno bavi poglavlje 5.

## 1.6 Pitanja za učenje i ponavljanje

- 1.1. Objasnite ulogu programskog posredničkog sloja za raspodijeljene sustave.
- 1.2. Usporedite migracijsku i relokacijsku transparentnost raspodijeljenog sustava. Koje svojstvo (ne)zadovoljava web poslužitelj i zašto?
- 1.3. Objasnite pojam skalabilnosti raspodijeljenog sustava.
- 1.4. Kojim aspektima transparentnosti pridonosi sustav imenovanja domena (DNS)?
- 1.5. Napravite analizu transparentnosti raspodijeljenog sustava elektroničke pošte.
- 1.6. Prikažite i objasnite primjer troredne arhitekture weba.
- 1.7. Kakvi bi se problemi pojavili kad bi se više repliciranih poslužitelja weba priključilo na mrežu izravno, a ne putem zastupnika (proxy)?
- 1.8. Kakvi su tržišni udjeli web-poslužitelja i web-preglednika?
- 1.9. Što je vrijeme odziva za poslužitelja weba?

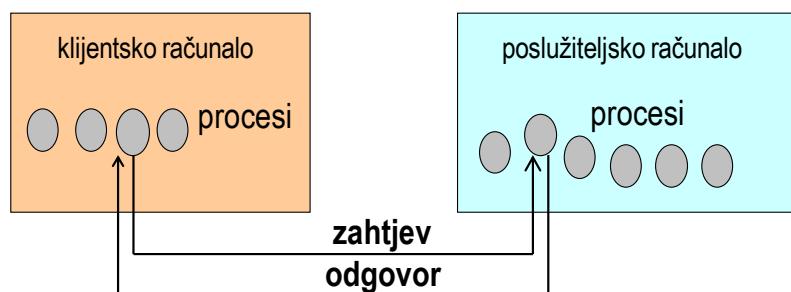
## 2 PROCESI I KOMUNIKACIJA

Svaki raspodijeljeni sustav izgrađen je od procesa koji se izvode na udaljenim računalima, a međusobno komuniciraju putem komunikacijske mreže. Osnovni model komunikacije u raspodijeljenom okružju oslanja se na razmjenu podataka među autonomnim procesima koji se u načelu izvode na različitim računalima. Stoga je komunikacija među udaljenim procesima, tj. razmjena podataka među procesima na različitim računalima osnova svakog raspodijeljenog sustava. Međuprocesna komunikacija (engl. *interprocess communication*, IPC) je realizirana proslijedivanjem poruka (engl. *message passing*) na mrežnom sloju na način da se podaci "pakiraju" u poruku u oblik prikladan za prijenos mrežom. Ova komunikacija osigurava vremensku usklađenost odvijanja procesa jer, prisjetimo se, procesi se izvode na različitim računalima i neovisni su ako međusobno ne komuniciraju.



Slika 2.1. Primjer komunikacije između dva procesa

Slika 2.1 prikazuje primjer komunikacije između dva procesa. Kada proces *a* koji se izvodi na računalu *A* želi komunicirati s procesom *b* koji se izvodi na računalu *B*, tada proces *a* kreira poruku u svome adresnom prostoru i poziva funkciju kojom operacijski sustav računala *A* preko mreže šalje poruku procesu *b* na računalu *B*. Proses *a* je pošiljatelj, a proces *b* primatelj poruke. Kako bi procesi *a* i *b* mogli komunicirati, moraju istovjetno interpretirati niz bitova koji čine prenesenu poruku. Također je na računalu *A* potrebno odražavati spremnik za odlazne poruke procesa *a*, a na računalu *B* spremnik za dolazne poruke procesa (Birrell & Nelson, 1984).



Slika 2.2. Komunikacija zahtjev-odgovor

Model klijent-poslužitelj danas je još uvijek prevladavajući komunikacijski i obradni model na Internetu, a uključuje jedan poslužiteljski proces i najčešće veći broj klijentskih procesa. Poslužiteljski proces nudi određenu uslugu, npr. obrađuje primljene podatke, pohranjuje ih u bazu podataka ili

dohvaća tražene podatke iz baze podataka. Klijentski proces zahtjeva neku uslugu od strane poslužitelja, npr. šalje podatke na obradu ili traži podatke iz baze podataka. Model klijent-poslužitelj koristi model komunikacije poznat pod nazivom "zahtjev-odgovor" (engl. *request-reply*) jer klijentski proces šalje *zahtjev* poslužiteljskom procesu koji po potrebi vraća *odgovor* klijentu (Coulouris, Dollimore, Kindberg, & Blair, 2012, pp. 187-195). Pošiljatelj zahtjeva je klijentski proces, a pošiljatelj odgovora poslužiteljski proces, dok je primatelj zahtjeva poslužiteljski proces, a primatelj odgovora klijentski proces. Slika 2.2 prikazuje procese na klijentskom i poslužiteljskom računalu te komunikaciju među procesima pomoću poruka koje prenose zahtjevi i odgovori. Valja naglasiti da se klijentsko računalo naziva tako samo zbog izvođenja klijentskog procesa te može istovremeno biti i poslužiteljsko računalo ako se na njemu izvode poslužiteljski procesi. Npr. u sustavima s ravnopravnim sudionicima, na istom računalu se izvode i klijentski i poslužiteljski procesi.

Poslužiteljski proces se mora jednoznačno identificirati *adresom* koja je poznata klijentima kako bi klijenti započeli komunikaciju. Adresa je nužna radi definiranja zahtjeva i usmjeravanja poruke do poslužiteljskog procesa. O vrstama adresa će biti više riječi u nastavku.

## 2.1 Obilježja komunikacije

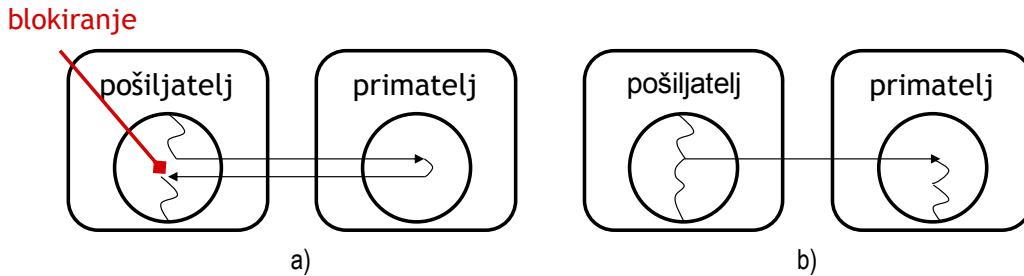
Pravila komunikacije među procesima definiraju se komunikacijskim protokolom. Prvo važno obilježje komunikacije opisuje vrstu komunikacijskog protokola koji može biti **koneksijski ili beskoneksijski** (Tanenbaum & Van Steen, 2007, pp. 116-117). **Komunikacijski protokol je koneksijski** ako procesi prvo razmijene kontrolne poruke, tj. uspostave asocijaciju ili vezu, i pri tome razmijene pravila i parametre komunikacije, a tek potom šalju podatke. **Protokol je beskoneksijski** ako se ne razmjenjuju kontrolne poruke za prethodnu uspostavu asocijacije, već se prenose samo poruke s podacima. Primjer koneksijskog transportnog protokola je *Transmission Control Protocol* (TCP), a beskoneksijskog *User Datagram Protocol* (UDP). U pouzdanoj mreži bez gubitaka paketa komunikacija klijenta i poslužitelja može biti implementirana vrlo jednostavnim beskoneksijskim protokolom, no za stvarne mreže, npr. Internet, potrebno je protokolom osigurati retransmisiju izgubljenih paketa.

Druge važne obilježje komunikacije su vezane uz garanciju isporuke poruke te se opisuju kao **perzistentna ili tranzijentna komunikacija** (Tanenbaum & Van Steen, 2007, pp. 124-125). **Komunikacija je perzistentna** ako je poslana poruka pohranjena u sustavu do trenutka isporuke primatelju. Perzistentna komunikacija garantira isporuku poruke premda primatelj nije aktivan u trenutku nastanka i slanja poruke. Pretpostavlja postojanje posrednika koji pohranjuje poruku do njene isporuke. **Tranzijentna komunikacija** garantira isporuku poruke samo ako su i pošiljatelj i primatelj istovremeno aktivni u trenutku slanja poruke. Npr. transportni sloj (TCP, UDP) nudi tranzijentnu komunikaciju.

Tablica 1. Pregled obilježja komunikacije

- |    |                       |                       |
|----|-----------------------|-----------------------|
| 1. | beskoneksijska        | koneksijska           |
| 2. | tranzijentna          | perzistentna          |
| 3. | asinkrona             | sinkrona              |
| 4. | na načelu <i>pull</i> | na načelu <i>push</i> |

Tablica 1 daje pregled osnovnih obilježja komunikacije udaljenih procesa na način da navodi međusobno suprotna svojstva komunikacije.

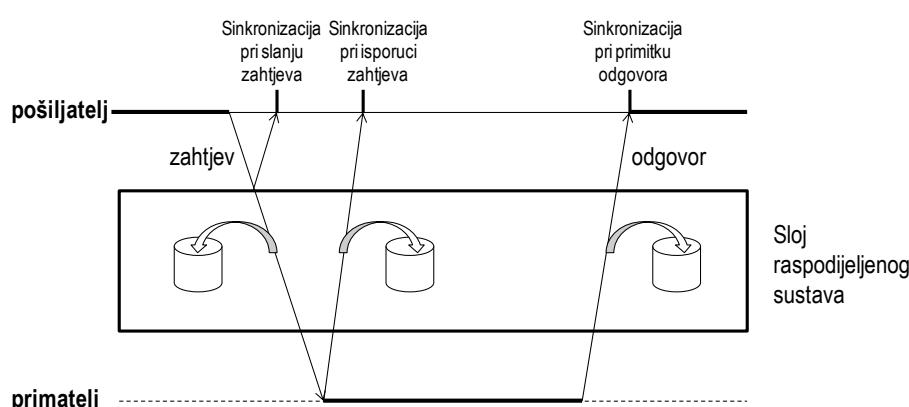


Slika 2.3. a) sinkrona i b) asinkrona komunikacija

Komunikacija među procesima može biti sinkrona ili asinkrona (Tanenbaum & Van Steen, 2007, p. 125), a ilustrirana je slikom (Slika 2.3). Za sinkronu komunikaciju vrijedi sljedeće: pošiljatelj poruke (zahtjeva) je blokiran dok ne primi potvrdu od strane primatelja vezano uz stanje njegovog zahtjeva. Asinkrona komunikacija pretpostavlja da nakon slanja poruke (zahtjeva) pošiljatelj nastavlja obradu bez čekanja na bilo kakvu povratnu informaciju, te se poruka jednostavno pohranjuje u izlazni spremnik pošiljatelja nakon čega operacijski sustav pošiljateljskog računala vodi računa o njenoj isporuci. Stoga operacija slanja zahtjeva nije blokirajuća za asinkronu komunikaciju, za razliku od sinkrone komunikacije.

S obzirom da prilikom sinkrone komunikacije vrijeme odziva može biti razmijerno dugo zbog kašnjenja prijenosa poruke ili duge obrade na strani poslužitelja, sinkronizacijska točka između pošiljatelja i primatelja poruke može se definirati i preciznije ako uzmemo u obzir da se sloj raspodijeljenog sustava brine o isporuci poruke, tj. pohranjuje je do isporuke na odredište. Moguće su tri sinkronizacijske točke kako je prikazano slikom (Slika 2.4). Pošiljatelj može biti blokiran do primitka potvrde o tome da je

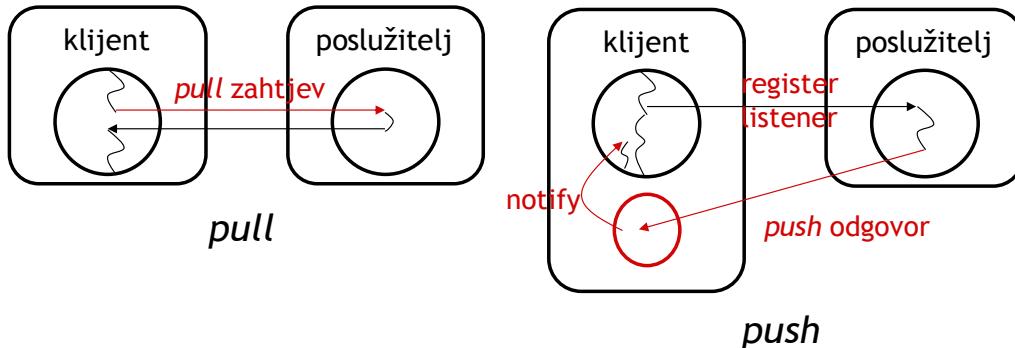
1. sloj raspodijeljenog sustava primio zahtjev te se dalje brine o isporuci poruke,
2. primatelj uspješno primio zahtjev nakon čega će započeti obradu, ili
3. primatelj uspješno primio i obradio poruku (zahtjev) te u odgovoru uz potvrdu šalje i rezultate obrade.



Slika 2.4. Moguće točke sinkronizacije pošiljatelja i primatelja

Postoji još jedno obilježje komunikacije koje se odnosi na isporuku odgovora od strane poslužitelja klijentu. Za sinkronu komunikaciju kažemo da je komunikacija na načelu *pull* jer „dovlači se odgovor s poslužitelja“. S obzirom na potencijalno dugo vrijeme čekanja na odgovor, kao na primjer kada je vrijeme obrade zahtjeva na poslužitelju relativno dugo, definiran je i drugi vid isporuke dogovora pomoću tzv. *listener-a*. Listener se izvodi u posebnoj dretvi i „osluškuje“ odgovor poslužitelja. Ovo je komunikacija na načelu *push* jer klijent šalje zahtjev i nastavlja dalje s obradom, a *listener* je sada u stanju „osluškivanja“ odgovora koji poslužitelj „pogura do klijenta“. Poslužitelj šalje odgovor kada

završi obradu zahtjeva ili kada mu npr. traženi podaci postanu dostupni. *Listener* je zapravo poseban poslužiteljski proces, premda je na strani klijenta, koji mora biti konstantno aktivan kako bi mogao primati poruke iz mreže.



Slika 2.5. Komunikacija na načelu *pull* i *push*

Slika 2.5 ilustrira komunikaciju na načelu *pull* i *push*. Vidljivo je da u slučaju komunikacije na načelu *push* klijent registrira *listener*a koji o primitku odgovora obavještava glavnu klijentsku dretvu.

## 2.2 Obilježja procesa

Proces se definira kao program u izvođenju na jednom od fizičkih ili virtualnih procesora računala. Operacijski sustav za svaki proces kreira nezavisan adresni prostor za pohranu podataka nužnih za izvođenje procesa koji ga štiti od ostalih konkurentnih procesa (Tanenbaum & Van Steen, 2007, pp. 70-75).

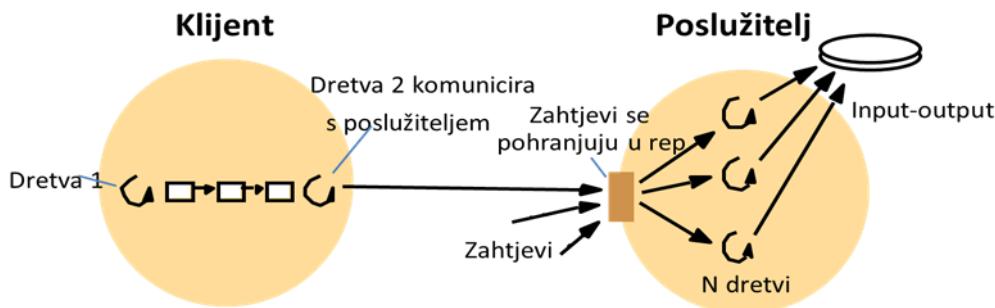
Važno je navesti dva osnovna obilježja procesa koja su značajna za raspodijeljene sustave: vremenska (ne)ovisnost o referenci "sugovornika". Vremenski ovisni procesi moraju biti istovremeno aktivni za realizaciju komunikacije, a vremenski neovisni procesi mogu komunicirati i ako nisu istovremeno aktivni. Primjer je komunikacija porukama u pokretnoj mreži kada se sustav brine o pohranjivanju poruke (SMS, MMS) u međuspremnik tako da je poruku moguće isporučiti i nakon uključivanja pokretnog telefona. Ovisnost o referenci "sugovornika" se odnosi na anonimnost komunikacije. Proces je ovisan o referenci "sugovornika" ako mora znati jedinstveni identifikator, tj. adresu udaljenog procesa s kojim želi komunicirati, što je uobičajena praksa za model klijent-poslužitelj. Primjerice, za protokole TCP i UDP nužno je znati transportnu adresu. Proces može biti i neovisan o referenci, tj. ne mora znati jedinstveni identifikator udaljenog procesa, što je obilježje komunikacije vođene podacima svojstvene sustavima objavi-preplati ili dijeljenom podatkovnom prostoru.

Tablica 2. Obilježja raspodijeljenih procesa

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

Tablica 2 navodi moguća obilježja raspodijeljenih procesa te specifična obilježja poslužiteljskih procesa koji mogu biti iterativni ili konkurentni te ovisni ili neovisni o stanju klijentskog zahtjeva (*stateful* ili *stateless*).

Poslužiteljski procesi se u raspodijeljenim sustavima najčešće izvode višedretveno. Dretve se izvode konkurentno unutar jednog procesa te dijele isti adresni prostor, a koriste se radi poboljšanja performansi i pojednostavljenja strukture procesa. Višedretvenost rješava probleme koje izazivaju metode sa svojstvom blokiranja procesa (npr. metoda `accept` za poslužiteljski *socket* TCP-a), jer blokiranje jedne dretve ne uzrokuje blokiranje procesa. Višedretvenost je posebno važna za učinkovitu implementaciju procesa u raspodijeljenom sustavu jer omogućuje održavanje više logičkih konekcija prema jednom procesu. Npr. višedretveni poslužitelj može paralelno obrađivati više korisničkih zahtjeva, dok višedretveni klijent može nastaviti s obradom dok čeka odgovor poslužitelja (primjer: web preglednik).



Slika 2.6. Višedretveni poslužitelj i klijent

Uobičajene zadaće na strani klijenta su na primjer prikaz korisničkog sučelja, otvaranje mrežne konekcije prema poslužitelju i primanje podataka. Uobičajene zadaće na strani poslužitelja su primanje konkurentnih klijentskih zahtjeva, složena obrada podataka i rad s diskom/bazom podataka.

**Iterativni poslužitelj** istovremeno poslužuje samo jedan klijentski zahtjev i neadekvatan je za primjenu u raspodijeljenim sustavima jer ostali klijenti ne mogu koristiti usluge poslužitelja dok ne završi obrada tekućeg zahtjeva. **Konkurentni poslužitelj** istovremeno može posluživati više klijentskih zahtjeva, a zahtijeva višedretvenu implementaciju koja je znatno složenija od implementacije iterativnog poslužitelja. Konkurentni poslužitelj u načelu dodjeljuje zahtjev za obradom novoj dretvi ili drugom procesu, a osnovna dretva nastavlja čekati nove zahtjeve.

Još je jedno važno svojstvo značajno za oblikovanje poslužitelja, a u vezi je s pamćenjem stanja na poslužitelju. Ako poslužitelj ne pamti stanje klijentskih zahtjeva kažemo da je **stateless**. Tipičan primjer takvog poslužitelja je web-poslužitelj koji svaki novi HTTP zahtjev smatra neovisnim o sljedećem ili prethodnom zahtjevu. Ako poslužitelj održava stanje zahtjeva za svakog klijenta, tada ga nazivamo **stateful** poslužiteljem. Karakterističan primjer je poslužitelj za pohranjivanje datoteka koji pohranjuje kopiju datoteke i održava tablicu koja povezuje klijenta, datoteku i njeno stanje te klijentu omogućuje pristup tim dokumentima. Implementacija poslužitelja koji ne pamti stanje, a želimo da je otporan na ispade, je jednostavnija od implementacije **stateful** poslužitelja koji se mora nakon ispada prilikom pokretanja vratiti u stanje prije ispada.

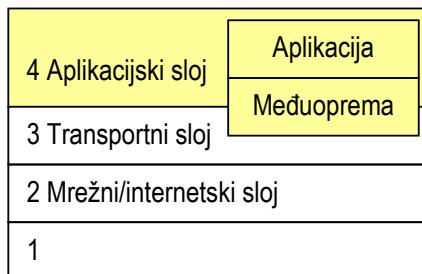
### 2.3 Pitanja za učenje i ponavljanje

- 2.1. Objasnite na primjeru razliku između perzistentne i tranzientne komunikacije.
- 2.2. Objasnite razliku između sinkrone i asinkrone komunikacije.
- 2.3. Objasnite zašto tranzientna sinkrona komunikacija potencijalno pati od problema vezanih uz skalabilnost.
- 2.4. Navedite prednosti koje ima operacija slanja zahtjeva koja je neblokirajuća u odnosu na blokirajuću operaciju.
- 2.5. Ima li smisla ograničiti broj dretvi za obradu korisničkih zahtjeva na višedretvenom poslužitelju? Zašto?

- 2.6. Je li poslužitelj koji održava TCP/IP konekciju prema klijentu *stateful* ili *stateless*? 
- 2.7. Navedite prednosti konkurentnog poslužitelja u odnosu na iterativnog poslužitelja.

### 3 SLOJ RASPODIJELJENOG SUSTAVA ZA KOMUNIKACIJU MEĐU PROCESIMA

Programski posrednički sloj ili međuoprema (engl. *middleware*) je programski sustav na aplikacijskom sloju koji se nalazi između transportnog sloja i aplikacije (Slika 3.1) (Bernstein, 1996). Međuoprema koristi usluge transportnog sloja s ciljem pojednostavljenja oblikovanja i razvoja aplikacija te se izdaje u obliku programskih knjižnica za programere aplikacija. Postoji niz različitih vrsta međuopreme ovisno o funkcionalnosti koju implementira, no za raspodijeljene sustave je temeljna **međuoprema za komunikaciju udaljenih procesa** za koju kažemo da se nalazi na sloju raspodijeljenog sustava. Takva međuoprema implementira komunikacijske protokole među raspodijeljenim procesima, a na višem je nivou apstrakcije od samog transportnog sloja. Stoga omogućuje jednostavniji razvoj raspodijeljenih aplikacija jer skriva kompleksnost i heterogenost transportnog sloja.



Slika 3.1. Sloj međuopreme

U nastavku razmatramo različite izvedbe međuopreme za komunikaciju udaljenih procesa od kojih u prvu skupinu spada međuoprema pogodna za model klijent-poslužitelj (*socket API*, *RPC* i *RMI*), a u drugu međuoprema za komunikaciju porukama i skupnu komunikaciju na temelju podataka (komunikacija na načelu objavi-preplati i dijeljeni podatkovni prostor).

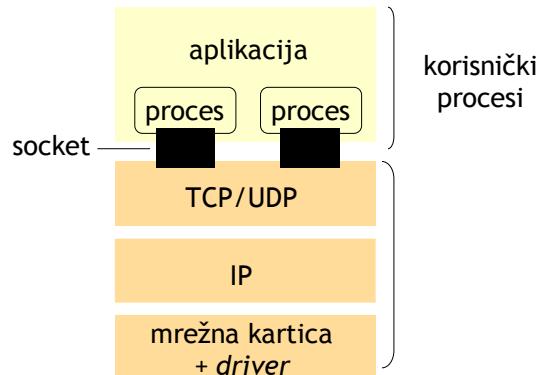
#### 3.1 Komunikacija korištenjem priključnica

Priključnica (engl. *socket*) je pristupna točka preko koje aplikacija šalje podatke u mrežu i iz koje čita primljene podatke, kako je ilustrirano slikom (Slika 3.2). Ova pristupna točka pruža viši nivo apstrakcije nad komunikacijskom točkom koju operacijski sustav koristi za pristup funkcijama transportnog sloja. Priključnica se veže uz vrata (engl. *port*) koja su jednoznačno povezana s procesom kojemu su poruke namijenjene, a vrata se označavaju cijelim brojevima.

*Socket API* je međuoprema koja direktno koristi funkcionalnost transportnog sloja za implementaciju komunikacije među procesima pomoću priključnica, a često se u literaturi naziva *Berkeley Sockets* (procedure definirane za programski jezik C iz 1970, Berkeley UNIX) (Tanenbaum & Van Steen, 2007, pp. 141-142). Podržava *Transmission Control Protocol* (TCP) i *User Datagram Protokol* (UDP) za komunikaciju između poslužitelja i većeg broja klijenata. TCP je konečki protokol koji nudi pouzdan prijenos podataka između udaljenih procesa, a UDP prenosi nezavisne pakete (*datagrami*) i stoga je nepouzdan. Važno je naglasiti da se na transportnom sloju koriste isključivo **transportne adrese** koje čini par (IP adresa, oznaka vrata). Na strani poslužitelja se koriste "dobro poznata vrata" jer klijent mora znati transportnu adresu poslužitelja kako bi započeo komunikaciju, a operacijski sustav prilikom primjeka podataka iz mreže mora jednoznačno identificirati poslužiteljski proces kome su podaci namijenjeni. Na strani klijenta operacijski sustav dinamički dodjeljuje slobodna vrata priključnici.

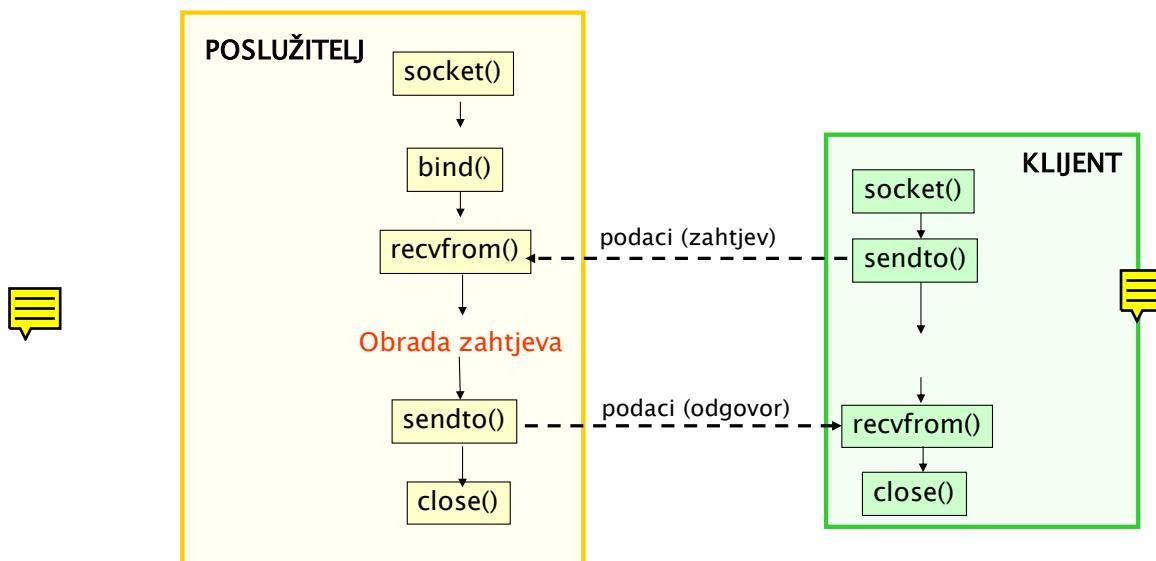
*User Datagram Protokol* (UDP) je beskonečki transportni protokol za prijenos nezavisnih paketa (*datagrami*). Omogućuje nepouzdanu komunikaciju uz jednostavnu nadogradnju protokola IP jer nema ugrađene mehanizme za detekciju i retransmisiju datagrama. UDP datagrami sadrže izvoršnu i odredišnu transportnu adresu. Datagrami poslati od strane klijentskog procesa se privremeno spremaju u izlazni spremnik povezan s vratima iz izvoršne transportne adrese na klijentskom računalu. Stoga je operacija slanja poruke neblokirajuća jer nastavlja s obradom nakon što preda

datagram operacijskom sustavu, tj. slojevima UDP i IP (Coulouris, Dollimore, Kindberg, & Blair, 2012, p. 150). Kada stignu na poslužiteljsko računalo, datagrami se spremaju u spremnik povezan s vratima iz odredišne transportne adrese dok ih poslužiteljski proces ne pročita. Moguće su dvije izvedbe primanja poruke: blokirajuća i neblokirajuća. Kada proces čita poruku iz spremnika može ostati blokiran ako je spremnik prazan dok ne stigne sljedeća poruka i ovakva se izvedba češće koristi od neblokirajuće.



Slika 3.2. Priključnica (socket)

Potrebno je naglasiti da ne postoji trajna veza između klijenta i poslužitelja te poslužitelj koristi izorišnu adresu iz datograma ako je potrebno poslati odgovor klijentu.



Slika 3.3. Dijagram tijeka komunikacije pomoću priključnice UDP

Slika 3.3 prikazuje dijagram tijeka komunikacije pomoću priključnice UDP. Koriste se sljedeće metode:

- `socket`: kreira novu komunikacijsku točku tj. priključnicu,
- `bind`: povezuje transportnu adresu (IP adresa, port) i `socket`,
- `recvfrom`: metoda za primanje datagrama (zahtjeva ili odgovora,)
- `sendto`: metoda za slanje datagrama (zahtjev ili odgovor) i
- `close`: zatvara `socket`.

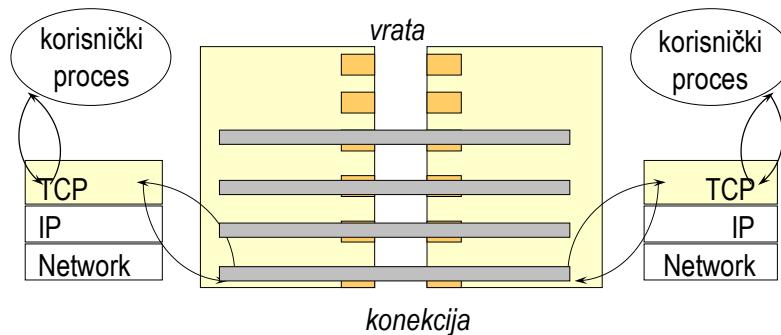
Na strani poslužitelja se prvo kreira `socket` te se pomoću metode `bind` povezuje s oznakom vrata. Potom poslužiteljski proces ulazi u blokirajuće stanje u metodi `recvfrom` jer čeka podatke iz mreže (blokirajuća izvedba je češća od neblokirajuće). Po primitu datagrama, obrađuju se primljeni podaci i poslužitelj po potrebi šalje odgovor klijentu koristeći metodu `sendto`. Na strani klijenta dijagram

tijeka je sličan kao i na poslužitelju, osim što se ne koristi metoda `bind` jer operacijski sustav samostalno povezuje socket sa slobodnim vratima.

Obilježja komunikacije pomoću UDP-a su:

- model klijent-poslužitelj
- vremenska ovisnost procesa: poslužitelj mora biti aktivan za primanje datagrama
- klijent mora znati identifikator poslužitelja
- tranzientna 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*

**Transmission Control Protocol (TCP)** je konečni transportni protokol koji osigurava pouzdan prijenos paketa između klijenta i poslužitelja tako što izgrađuje konekciju među njima te se brine o retransmisiji neisporučenih paketa. Konekcija se ostvaruje preko para transportnih adresa koristeći dvije priključnice, jednu na strani klijenta, a drugu na strani poslužitelja (Slika 3.4). Klijent zatraži uspostavu konekcije i ostaje blokiran čekajući da poslužitelj prihvati konekciju, a potom je moguće slati i primati podatke u oba smjera. Svaka priključnica za komunikaciju koristi jedan ulazni i jedan izlazni tok podataka (engl. *input and output stream*). Kada je komunikacija završena, svaka strana mora zatvoriti konekciju.

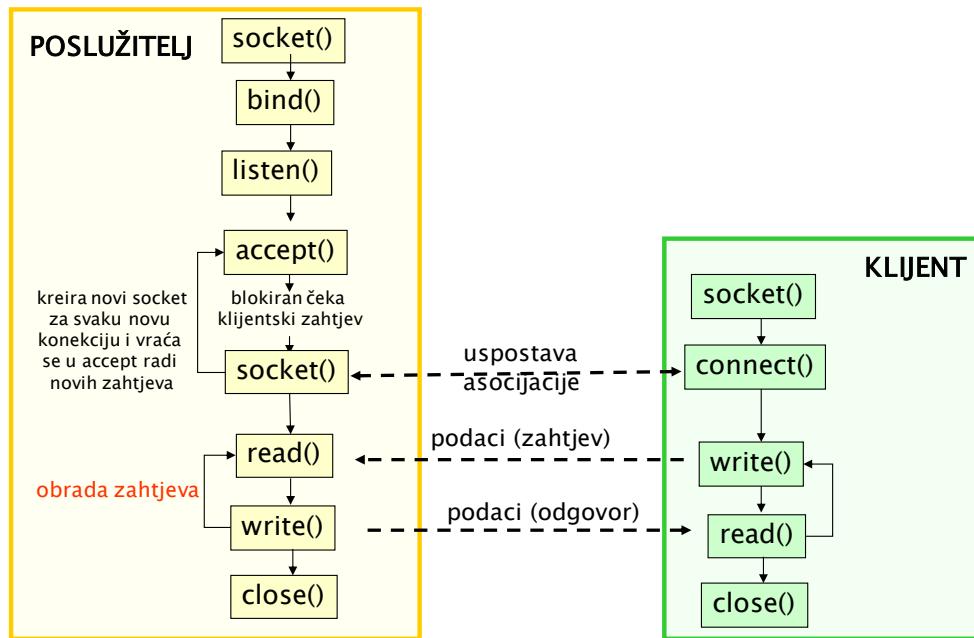


Slika 3.4. Komunikacija pomoću TCP-a

Slika 3.5 prikazuje dijagram tijeka komunikacije pomoću priključnice TCP. Koriste se sljedeće metode na strani poslužitelja:

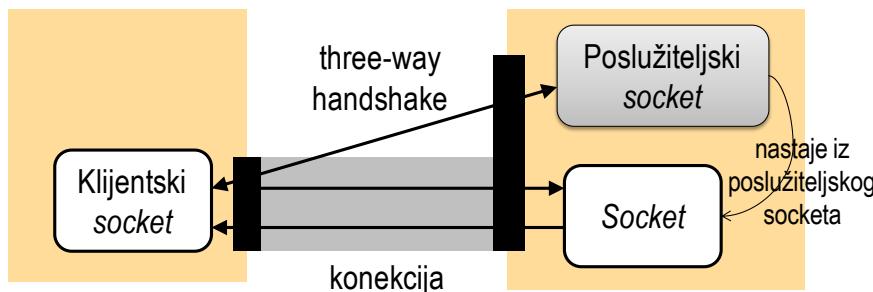
- `socket`: kreira komunikacijsku točku, operacijski sustav rezervira sredstva koja će omogućiti slanje i primanje podataka pomoću odabranog transportnog protokola
- `bind`: povezuje transportnu adresu sa socketom.
- `listen`: omogućuje operacijskom sustavu rezerviranje sredstva (spremnika) za specificirani maksimalni broj konekcija.
- `accept`: poslužitelj prima zahtjev za otvaranjem konekcije od strane klijenta (`connect`). Poslužitelj stvara novi `socket` koji se koristi za komunikaciju s klijentom. Originalni `socket` se koristi za primanje novih zahtjeva.
- `read i write`: metode za slanje i primanje podataka.
- `close`: zatvaranje konekcije i `socketa`.

Na strani klijenta nije potrebna metoda `bind` jer operacijski sustav dinamički pridjeljuje slobodna vrata `socketu` pri kreiranju konekcije. Metoda `connect` šalje zahtjev za kreiranje konekcije poslužitelju. Klijent mora definirati transportnu adresu na koju se šalje zahtjev za kreiranje konekcije te je blokiran do njene uspostave.



Slika 3.5. Dijagram tijeka komunikacije pomoću priključnice TCP

Važno je naglasiti da je metoda `accept` blokirajuća jer osluškuje zahtjeve iz mreže, a za svaki primljeni korisnički zahtjev kreira se novi `socket` koji je kopija poslužiteljskog `socketa` kako je ilustrirano slikom (Slika 3.6). Taj novi `socket` zapravo komunicira s klijentskim `socketom`, a vezan je uz ista vrata kao i originalni poslužiteljski `socket`. Operacijski sustav zna koji `socket` komunicira s pojedinim klijentskim `socketom` s obzirom da je konekcija identificirana pomoću para transportnih adresa između klijenta i poslužitelja.



Slika 3.6. Kreiranje nove priključnice za komunikaciju s klijentom

Obilježja komunikacije pomoću TCP-a su:

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

U nastavku su dane upute za implementaciju klijenata i poslužitelja pomoću priključnica u programskom jeziku Java. Koristi se paket `java.net`<sup>1</sup> i klase `DatagramSocket` i `DatagramPacket` za komunikaciju UDP-om, a klase `ServerSocket` i `Socket` za komunikaciju TCP-om.

Kako implementirati poslužitelja koji koristi UDP socket?

**1. Kreirati socket poslužitelja**

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

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

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

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

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

**4. Obraditi pristigli paketa i po potrebi poslati odgovor klijentu**

**5. Zatvoriti poslužiteljski socket**

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

Kako implementirati klijenta koji koristi UDP socket?

**1. Kreirati socket**

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

**2. Kreirati paket (prazan, priprema za primanje paketa iz mreže)**

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

**3. Slanje paketa**

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

**4. Po potrebi obrada i čekanje odgovora**

**5. Zatvoriti socket**

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

U slučaju TCP-a, potrebno je prvo instancirati klasu `ServerSocket` iz kojega metodom `accept` nastaje nova instanca objekta `Socket` za komunikaciju s klijentskim `socketom`.

Kako implementirati poslužitelja koji koristi TCP socket?

**1. Kreirati socket poslužitelja**

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

**2. Čekati korisnički zahtjev (proces je blokiran do klijentskog zahtjeva) i kreirati kopiju originalnog `socketa`**

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

**3. Kreirati I/O stream za komunikaciju s klijentom**

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

**4. Komunikacija s klijentom**

**5. Zatvoriti kopiju `socketa`**

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

**6. Zatvoriti poslužiteljski socket**

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

<sup>1</sup><http://download.oracle.com/javase/7/docs/api/java/net/package-summary.html>

Kako implementirati klijenta koji koristi TCP *socket*?

1. Kreirati klijentski *socket*

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

2. Kreirati I/O stream za komunikaciju s poslužiteljem

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

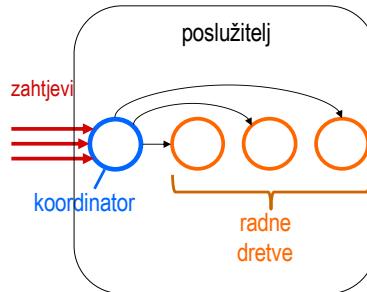
3. Komunikacija s poslužiteljem

```
//Receive data from server:
String line = is.readLine();
//Send data to server:
os.writeBytes("Hello\n");
```

4. Zatvoriti *socket*

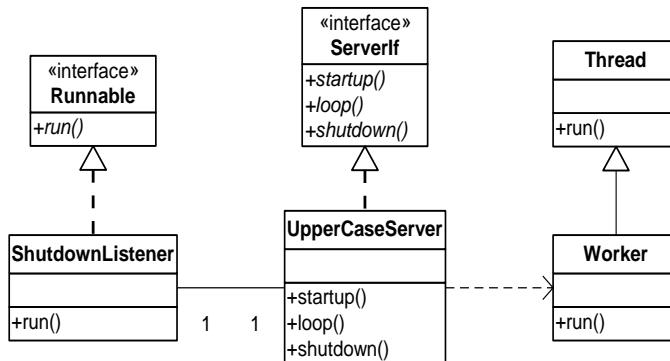
```
clientSocket.close();
```

**Višedretveni poslužitelj.** Za implementaciju konkurentnog poslužitelja, potrebno je kreirati novu dretvu za svaki klijentski zahtjev tako da postoji jedna po konekciji. U ovu svrhu se često koristi model koordinator/radna dretva (engl. *dispatcher/worker model*) ilustriran slikom (Slika 3.7) (Tanenbaum & Van Steen, 2007, pp. 77-79). Koordinator prima zahtjeve, kreira novu dretvu koju pridjeli zahtjevu i novom radnom objektu, a on obrađuje zahtjev i komunicira s klijentom. No kako je kreiranje i izvođenje dretvi procesorski zahtjevno, preporučuje se ograničiti broj aktivnih dretvi na strani poslužitelja (engl. *thread pool*) tako da koordinator može istovremeno obrađivati samo ograničen broj zahtjeva. Stoga neki zahtjevi mogu biti odbačeni ili se pohranjuju u spremnik i čekaju "oslobađanje" dretve.



Slika 3.7. Model koordinator/radna dretva

U nastavku je dan primjer višedretvenog poslužitelja u programskom jeziku Java. Poslužitelj obavlja jednostavnu funkciju: prima korisničke zahtjeve koristeći TCP i pretvara primljene *stringove* u velika slova koja vraća klijentu. Slika 3.8 prikazuje dijagram klasa višedretvenog poslužitelja. Osnovna klasa je `UpperCaseServer`, a implementira sučelje `ServerIf` koje definira metode koje svaki poslužiteljski objekt mora sadržavati. To su metoda `startup` koja instancira sve potrebne objekte i dovodi poslužiteljski objekt u radno stanje, metoda `loop` u kojoj se obrađuju zahtjevi, te metoda `shutdown` koja je inverzija metode `startup`.



Slika 3.8. Primjer višedretvenog poslužitelja: dijagram klasa

Višedretvenog poslužitelja implementira klasa `UpperCaseServer` čiji je kôd dan u nastavku (Programski kôd1). Ova klasa kreira poslužiteljski *socket* (`serverSocket`) koji prima klijentske zahtjeve na definiranom broju vrata (`port`) s ograničenjem veličine repa za dolazne zahtjeve na `serverSocket`-u (tzv. *backlog*) na 10. Koristi se zastavica `runningFlag` koja označava kada je poslužitelj spreman primati zahtjeve. U posebnoj dretvi se izvodi `shutdownListener` koji prima iz mreže zahtjev za gašenjem poslužitelja te zastavicu `runningFlag` postavlja u *false*. Reci kôda (9-21) definiraju konstruktor klase `UpperCaseServer` u kome se određuje naziv računala na kome se izvodi poslužiteljski proces i provjerava oznaka vrata tako da je veća ili jednaka 1024. Metoda `startup` (22-32) instancira poslužiteljski *socket* s ograničenjem broja konkurentnih konekcija i u novoj dretvi započinje izvođenje `shutdownListener` nakon čega je poslužitelj u radnom stanju. Metoda `loop` (33-51) je osnovna radna metoda poslužiteljskog procesa u kojoj se nad poslužiteljskim *socketom* postavlja vremensko ograničenje (`timeout`) od 500 ms na trajanje blokiranja metode `accept` prilikom čekanja klijentskih zahtjeva tako da se nakon isteka 500 ms podiže `SocketTimeoutException`. Dok je zastavica `runningFlag` u stanju *true*, objekt `serverSocket` prima klijentske zahtjeve pomoću metode `accept` i predaje objekt klase `Socket` koji nastaje kao posljedica klijentskog zahtjeva i izvođenja metode `accept` kao parametar radnom objektu `Worker`. Radni objekt `Worker` se izvodi u posebnoj dretvi, a broj aktivnih konekcija se povećava za 1. U slučaju da nema novih zahtjeva, nakon isteka 500 ms podiže se `SocketTimeoutException` i provjerava stanje zastavice `runningFlag`. Dok god je ona u *true*, poslužiteljski *socket* je u stanju osluškivanja. U nastavku (Programski kôd2) se nalaze metode `shutdown` i `main`. Metoda `shutdown` (52-70) provjerava postoje li aktivne konekcije prema poslužitelju, te ako postoje čeka se da komunikacija s klijentom završi. Kada nema konekcija, zatvara se poslužiteljski *socket* (64). U metodi `main` se instancira novi objekt klase `UpperCaseServer` te se redom pozivaju metode `startup`, `loop` i `shutdown`.

```

1 public class UpperCaseServer implements ServerIf {
2     private String hostName = null;
3     private int port;
4     private ServerSocket serverSocket;
5     private boolean runningFlag = false;
6     //reference to a shutdown listener, it accepts shutdownrequests
7     private ShutdownListener shutdownListener = null;
8     //number of active TCP connections
9     private int activeConnections = 0;
10    private final static int MAX_CLIENTS = 10;
11
12    public UpperCaseServer( int portNo ) {
13        try { //determine the local hostname
14            hostName = InetAddress.getLocalHost().getHostName();
15        } catch (UnknownHostException e) {
16            e.printStackTrace();
17            System.exit(-1);
18        }
19        port = portNo;
20        if( port < 1024 ) {
21            System.err.println( "UpperCaseServer: illegal port number ["+
22                port + "]");
23            System.exit(-1);
24        }
25    }
26
27    public void shutdown() {
28        if( activeConnections > 0 ) {
29            try {
30                shutdownListener.shutdown();
31            } catch (IOException e) {
32                e.printStackTrace();
33            }
34        }
35        try {
36            serverSocket.close();
37        } catch (IOException e) {
38            e.printStackTrace();
39        }
40    }
41
42    public void startup() {
43        Thread shutdownThread = new Thread(shutdownListener);
44        shutdownThread.start();
45
46        try {
47            serverSocket = new ServerSocket(port, MAX_CLIENTS);
48        } catch (IOException e) {
49            e.printStackTrace();
50        }
51
52        while( !runningFlag ) {
53            try {
54                Socket clientSocket = serverSocket.accept();
55                Worker worker = new Worker(clientSocket);
56                worker.start();
57            } catch (SocketTimeoutException e) {
58                continue;
59            }
60        }
61    }
62
63    public void loop() {
64        if( activeConnections > 0 ) {
65            try {
66                Thread.sleep(timeout);
67            } catch (InterruptedException e) {
68                e.printStackTrace();
69            }
70        }
71        if( !runningFlag ) {
72            shutdown();
73        }
74    }
75
76    public void main() {
77        startup();
78        loop();
79        shutdown();
80    }
81
82    public void printInfo() {
83        System.out.println("hostName: " + hostName);
84        System.out.println("port: " + port);
85        System.out.println("activeConnections: " + activeConnections);
86    }
87
88    public void shutdown() {
89        shutdown();
90    }
91
92    public void startup() {
93        startup();
94    }
95
96    public void loop() {
97        loop();
98    }
99
100   public void main() {
101        main();
102    }
103 }
```

Programski kôd1. Višedretveni poslužitelj `UpperCaseServer` (1)

```

//starts all required server services
22 public void startup() {
23     try {
24         //create server socket, bind it to the specified port
25         //and set the max queue length for client requests
26         serverSocket = new ServerSocket(port, MAX_CLIENTS);
27         //create shutdown listener in a new thread
28         shutdownListener = new ShutdownListener(this, 4456);
29         new Thread( shutdownListener ).start();
30         runningFlag = true;
31     } catch( IOException e) {
32         e.printStackTrace();
33         System.exit(-1);
34     }
35     //main loop for accepting client requests
36     public void loop() {
37         try { //set timeout to avoid blocking
38             serverSocket.setSoTimeout( 500 );
39         } catch( SocketException e ){
40             e.printStackTrace();
41             System.exit(-1);
42         }
43         while( runningFlag ) {
44             try {
45                 //accept requests and create a new Worker object
46                 //in a new thread, or throw a SocketTimeoutException
47                 Thread w = (Thread)new Worker(serverSocket.accept());
48                 w.start();
49                 activeConnections++;
50             } catch( SocketTimeoutException ste ) {
51                 //do nothing, check the runningFlag
52             } catch( IOException e ) {
53                 e.printStackTrace();
54                 System.exit(-1);
55             }
56         }
57     }
58     public void shutdown() {
59         while( activeConnections > 0 ) {
60             System.out.println( "WARNING: There are still active
61             connections" );
62             try {
63                 Thread.sleep( 5000 );
64             } catch( java.lang.InterruptedException e ){}
65         }
66         if( activeConnections == 0 ) {
67             System.out.println( "Server shutdown." );
68             if( serverSocket != null )
69             try {
70                 serverSocket.close();
71             } catch( IOException e ) {
72                 e.printStackTrace();
73                 System.exit(-1);
74             }
75         }
76     }
77     //Creates a new server object, initiates serverstartup, calls the loop
78     //method, and when the loop terminates invokes the shutdown method.
79     public static void main(String argv[]) {
80        UpperCaseServer server = new UpperCaseServer(10002);
81         server.startup();
82         server.loop();
83         server.shutdown();
84     }

```

Ono što se može zamijetiti na temelju prethodno analiziranog programskog kôda je da klasa `UpperCaseServer` samo omogućuje primanje klijentskih zahtjeva, ali ih ne obrađuje. Stoga ova klasa implementira koordinatora, dok je implementacija radnih objekata dana u nastavku (Programski kôd3). Klasa `Worker` je u primjeru unutarnja klasa klase `UpperCaseServer` koja u konstruktoru prima referencu na instancu klase `Socket`. Taj objekt se koristi za komunikaciju s klijentom, a nastao je na temelju poziva metode `accept` nad poslužiteljskim *socketom*. Važno je uočiti da je *socket* povezan s ulaznim i izlaznim tokom podataka (`InputStream` i `OutputStream`) koji služe za dvosmjernu komunikaciju s klijentom. Ovi se tokovi podataka koriste unutar metode `run` (prisjetimo se, radni objekt se izvodi u vlastitoj dretvi) za čitanje pojedinog retka teksta posланог od klijenta poslužitelju koji poslužitelj onda pretvara u velika slova i šalje kao odgovor klijentu. Na ovaj je način ostvaren dijalog između klijenta i radnog objekta dok klijent ne pošalje prazan redak (21) nakon čega se zatvaraju tokovi podataka (27-28), *socket* (30)i smanjuje broj aktivnih konekcija za 1 (31).

```

1 //inner class within UpperCaseServer
2 private class Worker extends Thread {
3     Socket socket = null;
4     Worker( Socket sckt ){
5         super( "UpperCaseServer.Worker" );
6         socket = sckt;
7     }
8     //Creates reader and writer for the socket, reads a line from
9     //the reader, transforms it to upper case, and sends the
10    //transformed string to client.
11    public void run() {
12        String clientInput = null;
13        String capitalizedInput = null;
14        try {
15            PrintWriter outToClient =
16                new PrintWriter(socket.getOutputStream(), true);
17            BufferedReader inFromClient =
18                new BufferedReader(new InputStreamReader(
19                    socket.getInputStream()));
20            while( (clientInput = inFromClient.readLine()) != null ){
21                System.out.println( "Server received from " +
22                    socket.getRemoteSocketAddress().toString() + ":" + clientInput );
23                if( clientInput.equals( "\n" ) )
24                    break;
25                capitalizedInput = clientInput.toUpperCase();
26                outToClient.println(capitalizedInput);
27                System.out.println( "Server sends:" + capitalizedInput );
28            }
29            outToClient.close();
30            inFromClient.close();
31            if( socket != null )
32                socket.close();
33            activeConnections--;
34        } catch( IOException e ) {
35            e.printStackTrace();
36        }
37    }
38 }
```

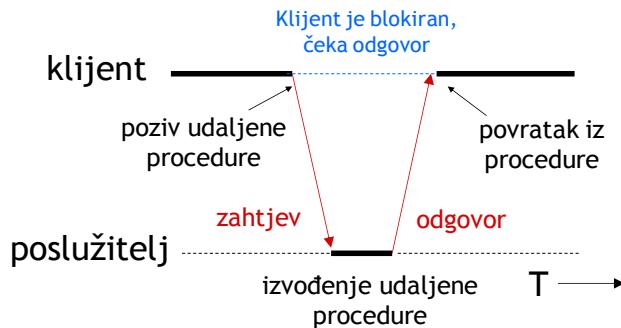
Programski kôd3. Klasa `Worker` za kreiranje radnih objekata

### 3.2 Poziv udaljene procedure/metode

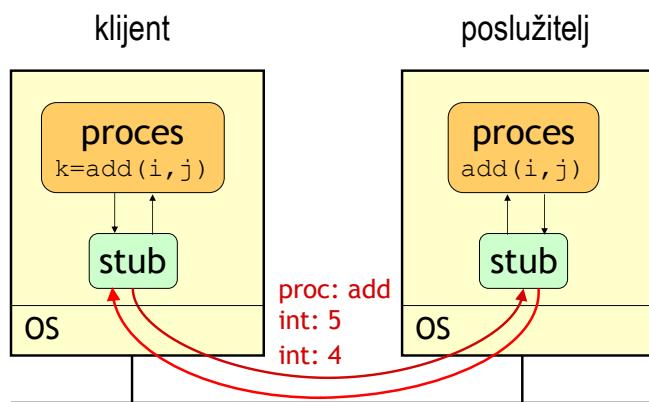
Poziv udaljene procedure (engl. *Remote Procedure Call*, RPC) omogućuje procesima pozivanje i izvođenje procedura na udaljenom računalu. Kada proces koji se izvodi na računalu A poziva proceduru na računalu B, pozivajući proces na računalu A šalje parametre za izvođenje procedure na računalo B i blokiran je čekajući rezultate izvođenja procedure. Računalo B izvodi proceduru koristeći primljene parametre i šalje odgovor računalu A. Riječ je o relativno jednostavnoj ideji koja poziv

procedure na udaljenom računalu želi prikazati kao poziv lokalne procedure što izvedbeno i nije tako jednostavno (Birrell & Nelson, 1984). Time je RPC **transparentan** jer pozivajuća procedura nije svjesna da se proces izvodi na udaljenom računalu.

Slika 3.9 prikazuje interakciju između klijenta i poslužitelja prilikom RPC-a. Klijent poziva udaljenu proceduru putem posebne komponente nazvane *stub* koja šalje zahtjev za izvođenje procedure na poslužitelju. Na poslužitelju će zahtjev primiti odgovarajući *stub* te pozvati proceduru i nakon njenog izvođenja vratiti odgovor klijentu. RPC se dakle također temelji na modelu klijent-poslužitelj gdje je klijent blokirani dok ne primi odgovor poslužitelja.



Slika 3.9. Interakcija između klijenta i poslužitelja prilikom poziva udaljene metode

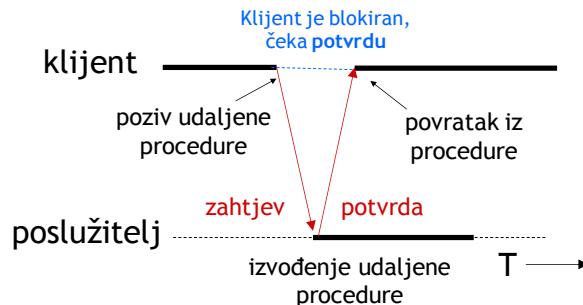


Slika 3.10. Koraci poziva udaljene procedure

Koraci poziva udaljene metode ilustrirani su slikom (Slika 3.10) na kojoj se vidi uloga *stuba* na strani klijenta i poslužitelja. *Stub* je posrednik koji preusmjerava poziv procedure s klijentskog procesa s potrebnim parametrima do poslužiteljskog stuba koji potom poziva proceduru na poslužitelju. U ovom primjeru poziva se procedura zbrajanja dva cijela broja, a koraci izvođenja su sljedeći:

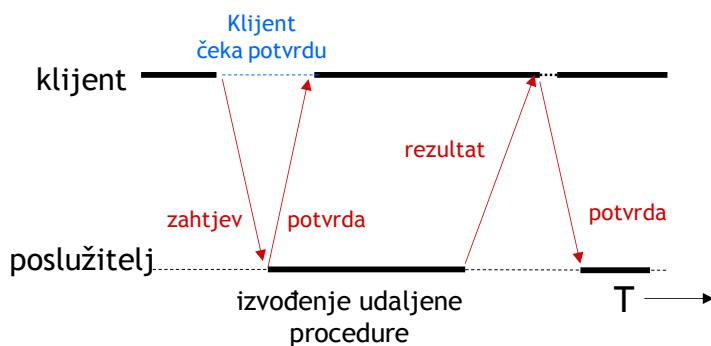
1. Klijent poziva udaljenu proceduru `add(i, j)` koristeći lokalni *stub*.
2. *Stub* "pakira" parametre i identifikator procedure i poziva operacijski sustav (OS) na klijentskom računalu.
3. OS klijentskog računala šalje poruku na udaljeno računalo.
4. OS udaljenog računala predaje poruku *stubu* poslužitelja.
5. *Stub* poslužitelja "raspakira" parametre i poziva proceduru `add(i, j)` koristeći primljene parametre.
6. Procedura vraća rezultat izvođenja poslužiteljskom *stub-u*.
7. *Stub* poslužitelja "pakira" rezultat u poruku i poziva OS.
8. OS poslužitelja šalje poruku OS-u klijenta.
9. OS klijenta predaje poruku *stubu*.
10. *Stub* "raspakira" rezultat i predaje ga klijentskom procesu.

Iz ovoga je vidljivo da *stub* osim uloge posrednika ima i ulogu pripreme poruke za prijenos mrežom. Postupak "pakiranja" parametara ili rezultata izvođenja procedure u poruku naziva se *marshaling*, dok je čitanje parametara ili rezultata iz poruke *unmarshaling*. Na primjer, prilikom poziva metode `add(i, j)`, u poruku se upisuje ime ili identifikator procedure, `int 5` i `int 4`. U slučaju da se prilikom poziva procedure za parametre koriste reference, situacija je složenija jer referenca ima smisla samo u adresnom prostoru procesa koji je koristi. Npr. ako je potrebno prenijeti *string*, onda se cijeli *string* zapisuje u poruku i prenosi do udaljene procedure (engl. *call by value*).



Slika 3.11. Asinkroni RPC

S obzirom da je prilikom RPC-a klijent blokiran do primitka odgovora, definirana je i posebna verzija RPC-a, tzv. asinkroni RPC ilustriran slikom (Slika 3.11). Klijent šalje zahtjev poslužitelju, ali ne očekuje odgovor, tj. rezultat izvođenja procedure, već samo potvrdu o primitku zahtjeva nakon čega nastavlja s vlastitom obradom. Koristan je i u slučaju kada klijent ne želi čekati odgovor poslužitelja te se tada koriste dva asinkrona RPC-a. U prvome klijent šalje zahtjev i nastavlja s obradom nakon što primi potvrdu, a rezultat izvođenja procedure poslužitelj šalje koristeći drugi asinkroni RPC (Slika 3.12). Ovo je tzv. odgođeni sinkroni RPC (engl. *deferred synchronous RPC*) (Tanenbaum & Van Steen, 2007, p. 134).

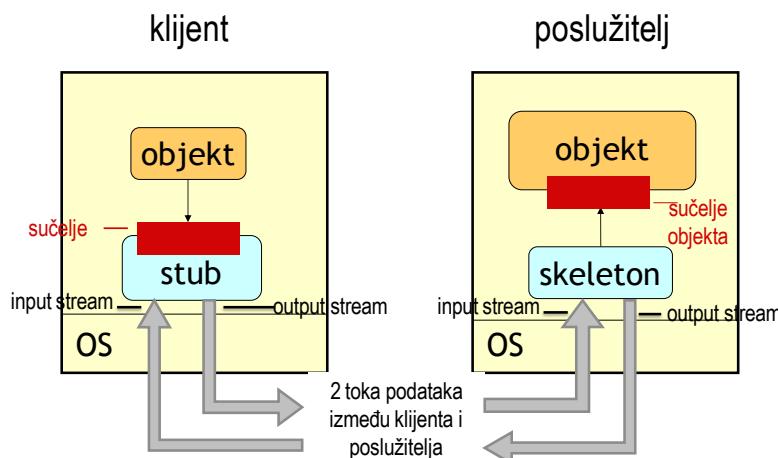


Slika 3.12. Odgođeni sinkroni RPC

**Poziv udaljene metode** (engl. *Remote Method Invocation*, RMI) je nasljednik poziva udaljene procedure razvijen za objektne jezike jer se poziva metoda udaljenog objekta. **Udaljeni objekt** je proširenje osnovnog objektnog modela koje je prilagođeno potrebama raspodijeljene okoline, a temelji se na osnovnoj značajki objekta koja odvaja sučelje od implementacije objekta (Ince, 2004, pp. 258-259). Pri tome svaki udaljeni objekt ima globalno jedinstven identifikator te sustav mora nuditi uslugu imenika za registriranje i pronalaženje udaljenih objekata (engl. *directory service*).

Ideja RMI-a je identična ideji RPC-a: klijentski objekt poziva metodu udaljenog poslužiteljskog objekta na transparentan način tj. identično pozivu metode lokalnog objekta. Pri tome se komponenta na strani poslužitelja naziva *skeletonom* umjesto *stubom* (Slika 3.13). *Stub* implementira sučelje udaljenog objekta, ali samo kao posrednik koji poziv metode zajedno s parametrima pakira u oblik

prikladan za prijenos mrežom i šalje do *skeletona* koji ga pak preusmjerava do udaljenog objekta. *Stub* i *skeleton* koriste 2 toka za prijenos podataka u oba smjera (Bacon & Harris, 2003, p. 475).



Slika 3.13. Izvođenje RMI-ja

Važno je zapaziti kako s obzirom da postoje reference na udaljene objekte, takve reference mogu biti prenesene kao parametar poziva udaljene metode (engl. *call by reference*).

Obilježja komunikacije poziva udaljene procedure/metode su:

- model klijent-poslužitelj
- vremenska ovisnost klijenta i poslužitelja
- klijent mora znati identifikator poslužitelja
- tranzijentna komunikacija
- sinkrona komunikacija (osim u slučaju asinkronog RPC-a)
- pokretanje komunikacije na načelu *pull*

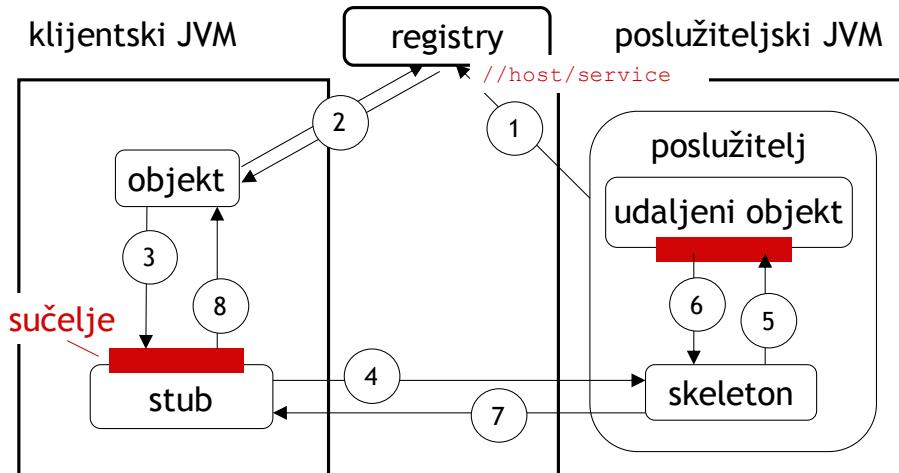
**Java RMI.** Kao primjer sustava za komunikaciju udaljenih objekata u nastavku se razmatra *Java Remote Method Invocation*, rješenje oblikovano isključivo za programski jezik Java (Bacon & Harris, 2003, pp. 469-475). Programski jezik Java definira udaljene objekte imajući na umu transparentnost kao najznačajnije svojstvo RMI-a. Pri tome stanje objekta može biti sačuvano samo na jednom računalu, a njegovo sučelje (tj. metode sučelja) je na raspolaganju procesima koji se izvode na drugim računalima (izvode se u drugom Javinom virtualnom stroju (engl. *Java Virtual Machine*, JVM)). Za implementaciju se koriste klase iz paketa `java.rmi`<sup>2</sup>.

Javin model objekta osigurava transparentnost pristupa udaljenim objektima na način da je referenca na udaljeni objekt istovjetna referenci na lokalni objekt, s tom razlikom da svi udaljeni objekti moraju implementirati sučelje `java.rmi.Remote`. Sučelje udaljenog objekta implementira *stub* (*proxy*) u adresnom prostoru klijentskog računala, a klase *stub* i *skeleton* se generiraju iz implementacije, a ne iz sučelja udaljenog objekta. Ovo svojstvo može predstavljati problem u slučaju česte promjene implementacije objekta jer se *stub* mora sukladno tome mijenjati na strani klijenta. Na transportnom sloju se koristi TCP.

Pri pozivu udaljene metode svi se lokalni parametri (primitivi i objekti) moraju serijalizirati jer se prenosi njihova vrijednost te moraju implementirati sučelje `Serializable`. U slučaju udaljenog objekta prenosi se njegova referenca koja je jedinstvena u raspodijeljenom sustavu, a sadrži adresu računala, broj vrata i identifikator udaljenog objekta. Udaljeni objekt mora implementirati sučelje

<sup>2</sup><http://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html>

`java.rmi.Remote` i biti pravilno eksportiran pomoću metode `UnicastRemoteObject.exportObject()` kako bi se mogao koristiti kao udaljeni objekt.



Slika 3.14. Koraci izvođenja Javinog RMI-ja

Komunikacija između klijentskog Javinog objekta i odaljenog objekta ilustrirana je slikom (Slika 3.14). Za registraciju udaljenih objekata se koristi posebna usluga poznata pod nazivom *rmiregistry*. Uz pretpostavku da u adresnom prostoru klijentskog računala postoji *stub* udaljenog objekta, koraci izvođenja Javinog RMI-ja su:

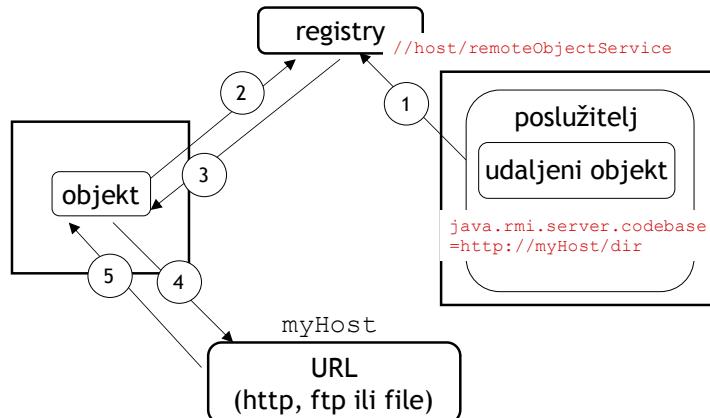
1. Poslužiteljski objekt se registrira pod odabranim imenom (npr. `//host/service`).
2. Klijent od *registrya* traži referencu na udaljeni objekt pomoću registriranog imena.
3. Klijent poziva metodu *stuba* koji je dostupan na klijentskom računalu.
4. *Stub* serijalizira parametre i šalje ih *skeletonu*.
5. *Skeleton* deserijalizira parametre i poziva metodu udaljenog objekta.
6. Udaljeni objekt vraća rezultat izvođenja metode *skeletonu*.
7. *Skeleton* serijalizira rezultat i šalje ga *stubu*.
8. *Stub* deserijalizira rezultat i dostavlja ga klijentu.

S obzirom da Java podržava dinamičko učitavanje klasa s FTP ili HTTP poslužitelja, ono se može koristiti za dohvaćanje stuba na strani klijenta. Naime, s obzirom da se *stub* može promijeniti zbog promjene implementacije udaljenog objekta, klijentski objekt mora pribaviti aktualni *stub* u svoj adresni prostor. Slika 3.15 prikazuje slijed akcija za dinamičko učitavanje klase *stuba* na stranu klijenta:

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

Može se zaključiti da Javina implementacija RMI-a osigurava visok nivo transparentnosti jer poziv udaljene metode ima jednaku sintaksu pozivu lokalne metode. Time je osigurana jednostavna i brza implementacija raspodijeljenog sustava, a programski kôd je značajno jednostavniji i čitljiviji u odnosu na rješenje koje koristi *Socket API*. Još jedno pozitivno svojstvo je dinamičko učitavanje klase *stuba* na strani klijenta. No negativno svojstvo je prije svega vezano uz performance: poziv udaljene metode je puno sporiji od poziva metode lokalnog objekta, čak i ako su udaljeni objekt i klijent na

istom računalu jer se koriste dvije TCP konekcije, jedna između klijenta i poslužitelja, a druga između klijenta i *registryja*, a protokol je oblikovan tako da generira značajan kontrolni promet.



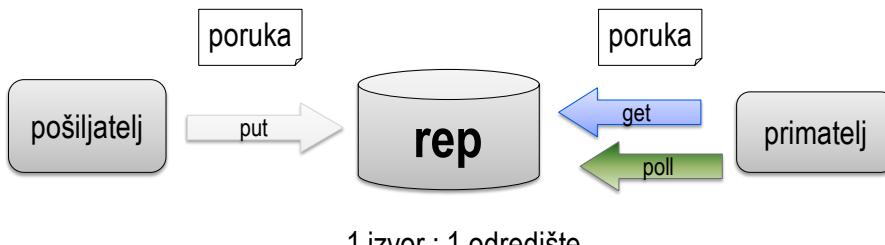
Slika 3.15. Dinamičko učitavanje klase stuba

### 3.3 Komunikacija porukama

Komunikacija porukama spada u važnu skupinu programske infrastrukture za komunikaciju raspodijeljenih procesa poznatu pod imenom *message-queuing systems* ili *Message-Oriented Middleware* (MOM) (Tanenbaum & Van Steen, 2007, pp. 145-152). Omogućuje perzistentnu i asinkronu komunikaciju preko posrednika koji pohranjuje poruke, a da pri tom izvor i odredište ne moraju biti istovremeno aktivni.

Osnovna ideja komunikacije porukama je razmjena poruka među procesima preko posrednika, tj. postoji poseban rep pridijeljen odredišnom procesu (primatelj poruke) koji se održava na poslužitelju-posredniku. U komunikaciji sudjeluje proces koji je izvor informacije i stoga pošiljatelj poruke te odredišni proces koji je primatelj poruke. Poruka se pohranjuje u poseban rep poruka pridijeljen odredištu, pa se ovaj način komunikacije prema modelu interakcije često uspoređuje s e-mailom ili SMS-om (koji su naravno aplikacije, a ne međuoprema).

Pošiljatelju se u načelu garantira isporuka poruke u primateljev rep, ali ne i isporuka poruke primatelju. Primatelj može pročitati poruku iz repa u bilo kojem trenutku nakon njene pohrane u rep. Stoga su pošiljatelj i primatelj poruke vremenski neovisni, a komunikacija je perzistentna.



1 izvor : 1 odredište

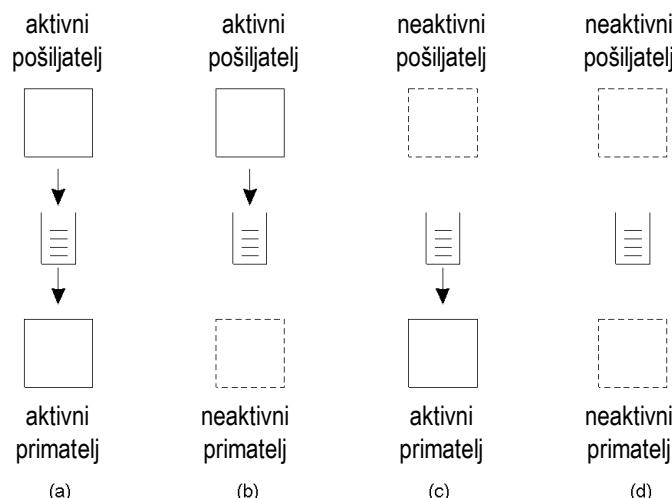
Slika 3.16. Izvođenje komunikacije porukama

Slika 3.16 shematski prikazuje komunikaciju porukama između para procesa. Pošiljatelj dodaje poruku u rep primatelja pomoću metode `put`. Primatelj čita poruku iz repa pomoću dvije metode: `get` i `poll`. Metoda `get` dohvata poruku iz repa (pri tome se kopija poruke briše iz repa), ali je primateljski proces blokiran dok se poruka ne isporuči, što znači da će u slučaju kada je rep prazan primateljski proces biti blokiran dok neki pošiljatelj ne isporuči poruku u rep. Metoda `poll` rješava problem blokiranja primatelja tako da provjerava stanje repa te dohvata jednu poruku iz njega ako

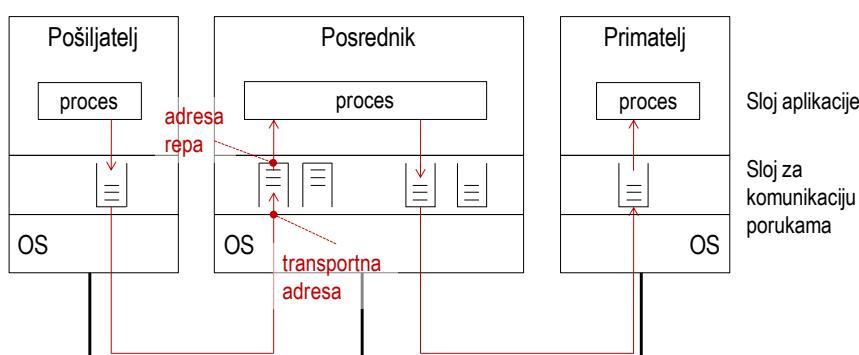
takva postoji, a u suprotnom nastavlja obradu. Važno je napomenuti da je ovo komunikacija jednog izvora i jednog odredišta te da pri tom primatelj šalje zahtjev i provjerava stanje repa te šalje zahtjev za čitanje poruke. Postoje i takve izvedbe međuopreme za komunikaciju porukama koje implementiraju mehanizam obavještavanja primatelja o postojanju poruke u njegovom repu.

Pošiljatelj i primatelj poruke su vremenski neovisni jer primatelj ne mora biti aktivan kada se poruka pohranjuje u rep. Poruka se čuva u repu do sljedećeg čitanja ili brisanja bez obzira na stanje pošiljatelja i primatelja. Postoje 4 moguće kombinacije stanja pošiljatelja i primatelja tijekom komunikacije, kako je prikazano slikom (Slika 3.17). Na slici a) su i pošiljatelj i primatelj aktivni tijekom prijenosa poruke prvo do repa, a zatim do primatelja. U drugoj je situaciji pošiljatelj aktivan tijekom isporuke i pohrane poruke u rep, dok je primatelj neaktivan. Kombinacija pasivnog pošiljatelja i aktivnog primatelja je prikazana na c) kada primatelj čita prethodno pohranjenu poruku iz repa. U posljednjoj situaciji su pošiljatelj i primatelj neaktivni, no poruka je pohranjena u repu (moguć je i prijenos te poruke kroz sustav za komunikaciju porukama ako je implementiran raspodijeljeno skupom poslužitelja).

Sustav za komunikaciju porukama sastoji se od posrednika na kome se održavaju repovi primatelja s neisporučenim porukama. Svaki rep ima jedinstveno ime (tzv. adresa repa) neovisno o samoj transportnoj adresi te je potrebna usluga koja povezuje ime repa s transportnom adresom (analogno DNS-u). Adresiranje se izvodi najčešće na nivou sustava, a svaki rep ima jedinstveni identifikator i stoga komunikacija porukama nije anonimna. Pošiljatelj i primatelj ne moraju znati ništa jedan o drugome, no moraju znati adresu repa i format poruke kako bi mogli komunicirati.



Slika 3.17. Neovisnost pošiljatelja i primatelja poruke



Slika 3.18. Arhitektura sustava za komunikaciju porukama

Slika 3.18 prikazuje primjer arhitekture sustava za komunikaciju porukama (Tanenbaum & Van Steen, 2007, p. 150). Na posredniku mora postojati poseban proces koji upravlja skupom repova (engl. *queue manager*). Ovaj proces može biti implementiran i kao posrednik koji prevodi jedan format poruke u drugi, tj. pretvara pošiljatelja u format razumljiv primatelju. Posrednik može biti implementiran i raspodijeljeno na više računala. Na sloju za komunikaciju porukama na strani pošiljatelja možemo imati rep s izlaznim porukama, a na strani primatelja rep s dolaznim porukama.

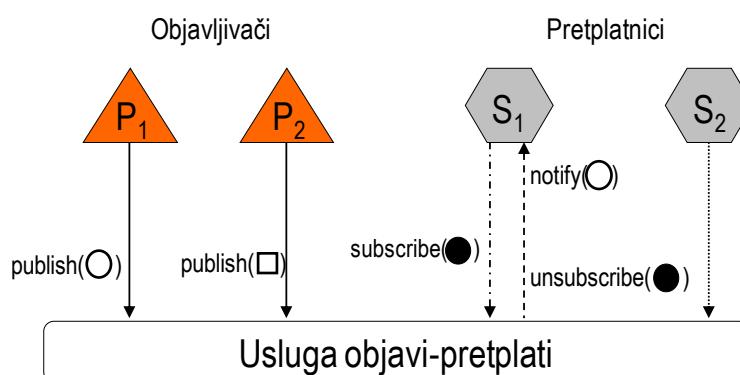
Obilježja komunikacije porukama su:

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

### 3.4 Komunikacija na načelu objavi-preplati

Model komunikacije objavi-preplati (engl. *publish/subscribe*) omogućuje asinkronu komunikaciju između objavljivača informacija/podataka/sadržaja (engl. *publishers*) i skupine preplatnika (engl. *subscribers*) (Podnar, 2004). Objavljivači definiraju strukturu i sadržaj poruka tj. obavijesti (engl. *notification*), a preplatnici definiraju svojstva poruka koje žele primati pomoću preplata (engl. *subscription*). Preplata se može zamisliti kao predložak (engl. *template*) obavijesti. Objavljivači ne šalju poruke direktno na odredišnu adresu zainteresiranih preplatnika, već se razmjena obavlja preko posrednika (engl. *broker*) na temelju usporedbe preplate i obavijesti. U slučaju kada obavijest odgovara sadržaju preplate, ona se proslijeđuje do preplatnika. Pri tome niti objavljivači niti preplatnici ne znaju jedni za druge. Ovo razdvajanje objavljivača i preplatnika povoljno utječe na skalabilnost sustava i dinamičniju mrežnu topologiju.

Slika 3.19 prikazuje komunikaciju među objavljivačima (pošiljatelj poruke) i preplatnicima (primatelj pouke) na načelu objavi-preplati. Usluga objavi-preplati je posrednik između strana u komunikaciji i zadužena je za isporuku obavijesti zainteresiranim preplatnicima te se često naziva informacijskom sabirnicom koja povezuje grupu preplatnika s grupom objavljivača. Na primjeru vidimo da preplatnik  $S_1$  definira novu preplatu jer ga zanimaju informacije o kružićima, dok je preplatnik  $S_2$  upravo prekinuo postojeću preplatu na kružice. Nakon toga objavljivač  $P_1$  objavljuje obavijest o bijelim kružićima koja je stoga isporučena do  $S_1$  (metoda *notify*), dok obavijest objavljivača  $P_2$  nije isporučena jer trenutno usluga objavi-preplati nema registrirane preplate kojima odgovara obavijest o bijelim kvadratićima.



Slika 3.19. Komunikacija na načelu objavi-preplati

Objavljavači i preplatnici komuniciraju tako da definiraju sljedeće entitete:

- obavijesti (engl. *notifications*),
- preplate (engl. *subscriptions*) i
- odjave preplata (engl. *unsubscriptions*).

Objavljavači definiraju obavijesti, dok preplatnici preplatama i odjavama preplata izražavaju namjeru primanja određenog skupa obavijesti. Usluga objavi-preplati obrađuje i pohranjuje primljene obavijest/preplate/odjave preplata, te isporučuje obavijesti preplatnicima prema njihovim aktivnim preplatama i to u trenutku neposredno nakon njihove objave. Stoga su sustavi objavi-preplati prikladni za širenje informacija čije se vrijeme nastanka i objavljivanja ne može unaprijed predvidjeti, npr. promjene vrijednosti dionica ili agencijske vijesti. Često se stoga u literaturi nazivaju sustavima vođenim događajima (engl. *event-based systems*).

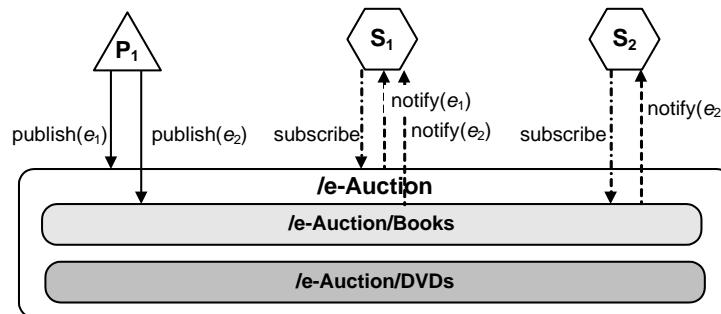
Model objavi-preplati izvrće klasični način rješavanja upita kakav nalazimo kod npr. baza podataka ili web tražilica kada se upit izvršava nad velikim skupom podataka. U ovom sustavu imamo preplate koje možemo zvati i trajnim upitim, a njih je potrebno usporediti s nadolazećim podacima koji se objavljuju u vremenu. Stoga sustav objavi-preplati mora održavati skup preplata (tj. upita, a ne podataka!) te svaku preplatu usporediti s obavijesti kako bi se ispitalo svojstvo "prekrivanja" (engl. *matching property*): Preplata "prekriva" obavijest kada obavijest zadovoljava sve uvjete definirane preplatom. Npr. preplata  $[a < 10, b \leq 20]$  prekriva obavijest  $[a=5, b=20]$ , ali ne prekriva obavijest  $[a=5, b=25]$

Ako preplata prekriva obavijest, tada se obavijest mora isporučiti preplatniku (tj. njegovom *listeneru*) koji je prethodno definirao preplatu. Učinkovita implementacija algoritama za usporedbu obavijesti sa skupom preplata je zahtjevna.

Modeli preplata koji se pojavljuju u literaturi razlikuju se po načinu definiranja uvjeta i preciznosti pomoću koje je dolazni skup obavijesti moguće filtrirati za pojedinu preplatu. Preplate zapravo omogućuju filtriranje obavijesti svih objavljavača tako da svaki preplatnik primi samo osobni skup obavijesti koje odgovaraju njegovim trenutnim interesima, tj. aktivnim preplatama. Postoji više vrsta preplata u sustavima objavi-preplati. Spomenut ćemo dvije najraširenije: preplata na kanal i na sadržaj.

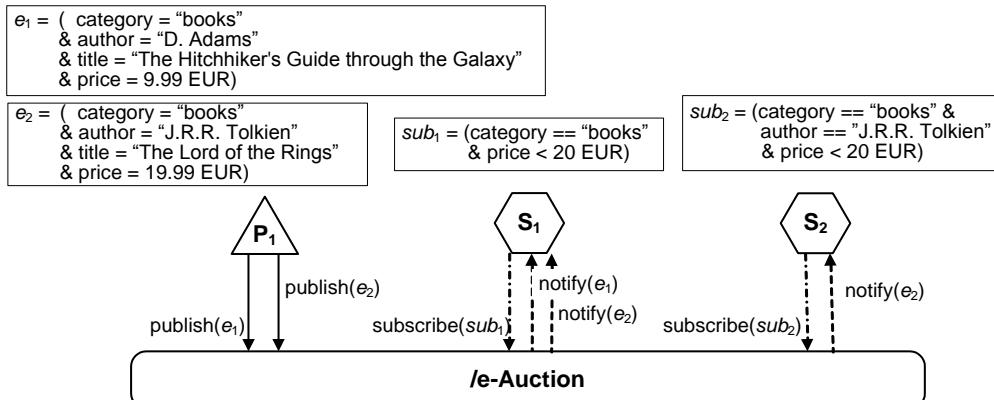
Preplata na kanal omogućuje tematsko grupiranje obavijesti i klasificira svaku obavijest u određenu kategoriju, tj. kanal, koji karakterizira sadržaj obavijesti. Kanal se može smatrati logičkom poveznicom među objavljavačima i preplatnicima. Kanali se mogu organizirati u hijerarhiju (slično nasljeđivanju kod objektnih jezika).

Preplata na sadržaj podržava znatno složenije i fleksibilnije preplate jer je moguće definirati uvjete nad atributima koje pojedina obavijest mora zadovoljiti, a da pri tome zadovoljava informacijske potrebe preplatnika. Uvjeti se definiraju atributom, vrijednošću i operatorom među njima (npr. =, <, >). Stoga je preplata podskup višeatributnog prostora.



Slika 3.20. Primjer preplate na kanal

Primjer na slici (Slika 3.20) prikazuje pretplatu na kanal koji modelira aukciju s 2 podteme: knjige i DVD-i. Pretplatnici  $S_2$  zanimaju samo knjige i stoga prima samo obavijest  $e_2$ , dok je  $S_1$  pretplaćen na cijeli kanal /e-Auction pa stoga prima  $e_1$  i  $e_2$ . Iz primjera se vidi mogućnost filtriranja obavijesti za  $S_2$  s obzirom da prima samo obavijesti tematski vezane uz /e-Auction/Books, a ne sve obavijesti.



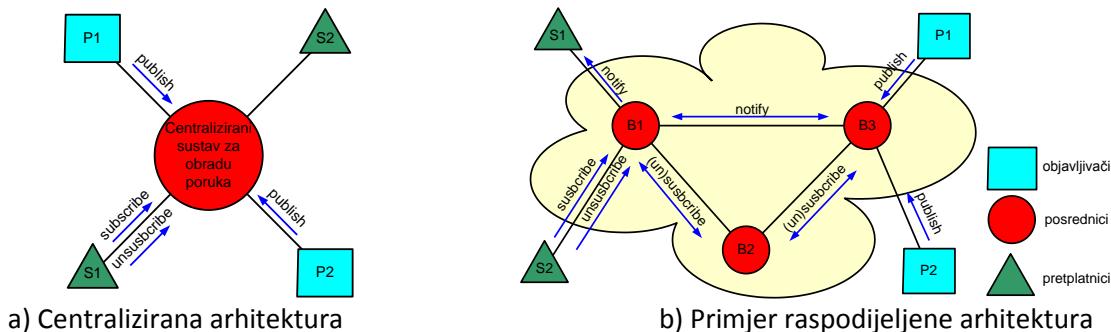
Slika 3.21. Primjer preplate na sadržaj

Slika 3.21 daje primjer preplate na sadržaj za strukturirane podatke u 4-dimenzionalnom atributnom prostoru gdje pretplatnici definiraju uvjete nad atributima koji moraju biti zadovoljeni kako bi im obavijest bila dostavljena.

**Arhitektura usluge objavi-preplati** može biti izvedena centralizirano ili raspodijeljeno (

Slika 3.22). Centralizirani sustav objavi-preplati sastoji se samo od jednog posrednika u interakciji sa cjelokupnim skupom pretplatnika i objavljavača, dok je raspodijeljeni sustav izgrađen od mreže posrednika od kojih svaki obrađuje dio zahtjeva klijenata (objavljavača i pretplatnika) te je zadužen za podskup pretplatnika i objavljavača koji su na njega direktno spojeni.

U centraliziranoj izvedbi posrednik posjeduje potpuno znanje o svim pretplatama i obavijestima u sustavu, te o svim pretplatnicima i objavljavačima. Osnovni je problem izvesti učinkoviti algoritam za usporedbu obavijesti s definiranim pretplatama radi brzog identificiranja skupa pretplatnika čije preplate "prekrivaju" definiranu obavijest. Glavni nedostatak centralizirane izvedbe je neskalabilnost i nepouzdanost rješenja jer je centralni poslužitelj opterećen svim korisničkim zahtjevima i predstavlja jedinstvenu točku ispada. Raspodijeljena arhitektura rješava problem skalabilnosti time što obradu obavijesti raspodjeljuje na više poslužitelja, a time je otpornija na ispadu. Ona se sastoji od skupa posrednika koji tvore mrežu poslužitelja (engl. *overlay network*).



Slika 3.22. Moguće arhitekture usluge objavi-preplati

U raspodijeljenoj izvedbi posrednici poslužuju lokalne objavljavače i pretplatnike te usmjeravaju obavijesti prema drugim posrednicima koji imaju zainteresirane pretplatnike. Isto tako usmjeravaju preplate susjednim posrednicima kako bi smanjili broj razmijenjenih obavijesti i time promet u

mreži. Promet koji generira razmjena pretplata među posrednicima je signalizacijski promet, a nastaje s ciljem smanjenja ukupnog prometa u mreži kako bi se sprječilo širenje obavijesti prema posrednicima koji nemaju pretplatnike zainteresirane za takve obavijesti.

Posrednici međusobno razmjenjuju poruke i održavaju tablicu usmjeravanja. Svaka poruka sadrži zaglavlje s podacima o izvoru i odredištu poruke te prenosi obavijest, pretplatu ili odjavu pretplate koja predstavlja korisnu informaciju za susjednog posrednika. Kontrolne poruke (pretplate i odjave pretplate) uzrokuju promjene u tablici usmjeravanja posrednika. Posrednik prosljeđuje obavijesti lokalnim pretplatnicima i susjednim posrednicima koji su za njih zainteresirani.

Osnovna načela usmjeravanja koja se koriste su:

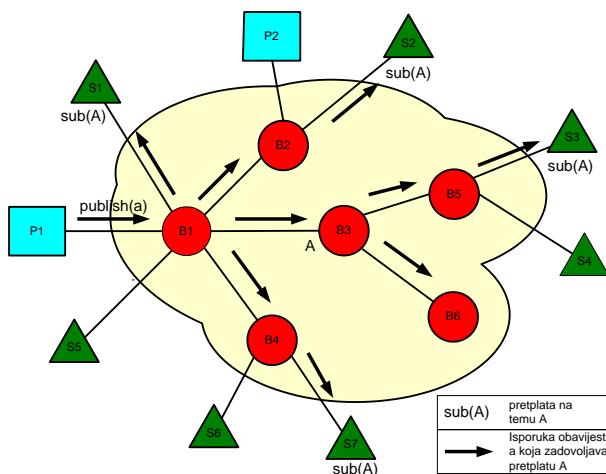
**Preplavljivanje:** svaka primljena poruka (obavijest ili pretplata) prosljeđuje se svim susjedima osim onome od koga je poruka primljena jer mrežu posrednika možemo preplavljivati obavijestima ali i pretplatama.

**Filtriranje poruka:** filtriranje poruka se izvodi usporedbom obavijesti sa skupom aktivnih pretplata koje definiraju svojstva obavijesti za koje je pretplatnik zainteresiran. Osnovni cilj je isporuka poruka samo do zainteresiranih pretplatnika ili posrednika koji imaju pretplatnike zainteresirane za poruku čime se smanjenje promet u mreži posrednika zbog sprječavanja širenja obavijesti "nezainteresiranim" posrednicima.

U nastavku su na primjeru objašnjena dva algoritma za usmjeravanje u sustavima objavi-pretplati: preplavljivanje obavijestima i preplavljivanje pretplatama.

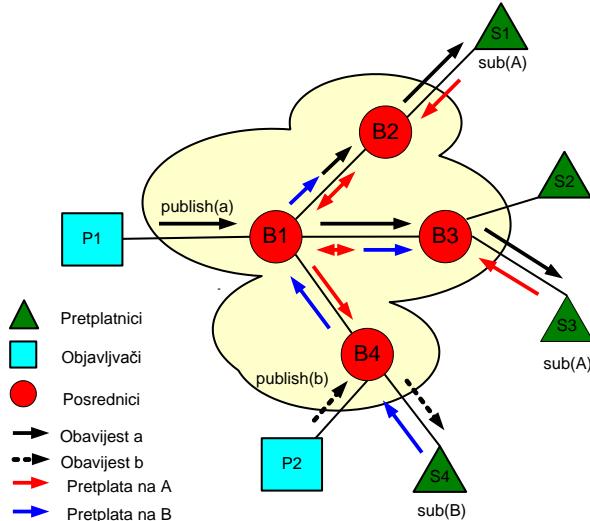
Preplavljivanje sustava obavijestima je najjednostavniji algoritam usmjeravanja u raspodijeljenim sustavima objavi-pretplati. Svaka obavijest se isporučuje svim posrednicima u sustavu, a filtriranje obavijesti vrši krajnji posrednik prije isporuke obavijesti lokalnim pretplatnicima te se samo obavijesti koje zadovoljavaju definirane pretplate isporučuju krajnjim lokalnim pretplatnicima. Svaki posrednik posjeduje statičku tablicu usmjeravanja koja sadrži informacije o svim njemu susjednim posrednicima te dinamičku tablicu svojih lokalnih pretplatnika i njihovih aktivnih pretplata. Usmjeravanje se izvodi na sljedeći način (primjer sa slike Slika 3.23):

- Kada posrednik primi obavijest od objavljavača, obavještava sve svoje lokalne zainteresirane pretplatnike te je prosljeđuje svim susjednim posrednicima. Npr.  $B_1$  prima obavijest "a" i prosljeđuje je do  $S_1$  jer  $S_1$  ima definiranu pretplatu koja "prekriva" obavijest "a", te do svojih susjeda  $B_2$ ,  $B_3$  i  $B_4$ .
- Kada posrednik primi obavijest od susjednog posrednika, on obavještava sve svoje zainteresirane pretplatnike, te prosljeđuje obavijest svim susjednim posrednicima osim onome od kojega je primio obavijest.



Slika 3.23. Preplavljivanje obavijestima

Problem ovog algoritma je veliki promet koji generira preplavljanje obavijesti u slučaju kada posrednici nemaju pretplatnike zainteresirane za većinu objavljenih obavijesti (bespotrebno se troše mrežna sredstva). Preplavljanje mreže posrednika pretplatama se koristi kako bi se smanjio broj obavijesti koje su nepotrebno proslijeđene posrednicima. Stoga se mreža preplavljuje kontrolnim informacijama, tj. pretplatama koje određuju smjer isporuke obavijesti.



Slika 3.24. Preplavljanje pretplatama

Na primjeru na slici (Slika 3.24), pretplata na kanal A koju definira  $S_1$  se proslijeđuje posredniku  $B_2$ ,  $B_2$  proslijeđuje pretplatu do  $B_1$ , a  $B_1$  do  $B_3$  i  $B_4$ . Isto se događa s pretplatom na kanal A koju definira  $S_3$ . Ta je pretplata proslijeđena s  $B_3$  na  $B_1$ , a  $B_1$  je šalje do  $B_2$  ali ne do  $B_4$  jer je tamo prethodno proslijeđena radi identične pretplate koju je definirao  $S_1$ . Slijed isporučenih pretplata definira putove za isporuku obavijesti. Stoga će bilo koja obavijest objavljena na  $B_1$ ,  $B_2$ ,  $B_3$  ili  $B_4$  biti proslijeđena do  $S_1$  i  $S_3$ , no samo ako zadovoljava pretplatu A (npr. obavijest "a" koju objavljuje  $P_1$ ). Neke obavijesti uopće neće biti usmjeravane kroz mrežu posrednika (npr. obavijest "b" se isporučuje samo pretplatniku  $S_4$  preko posrednika  $B_4$ ).

Kako bi odredio minimalan skup susjednih posrednika prema kojima treba proslijediti obavijesti, posrednik mora održavati tablicu usmjeravanja. Tablica usmjeravanja gradi se i osvježava na temelju pretplata koje je posrednik primio, bilo od pretplatnika, bilo od susjednih posrednika. Dakle, proslijeđivanjem pretplata drugim posrednicima u sustavu, posrednici najavljuju svoju zainteresiranost za određeni skup obavijesti, a time je omogućeno filtriranje objavljenih poruka "na izvoru", tj. na posredniku objavlivača. Na primjeru sa slike (Slika 3.24), tablica usmjeravanja na posredniku  $B_1$  definirana je u tablici (Tablica 1).

Tablica 3. Tablica usmjeravanja posrednika  $B_1$

Za obavijest na kanal	Šalji prema
A	B2,B3
B	B4

Obilježja modela objavi-preplati su sljedeća:

1. vremenska neovisnost: objavljuvачi i pretplatnici ne moraju istovremeno biti aktivni, posrednik pohranjuje poruku
2. objavljuvач ne mora znati identifikator pretplatnika (anonimnost), o tome se brine posrednik
3. komunikacija je perzistentna

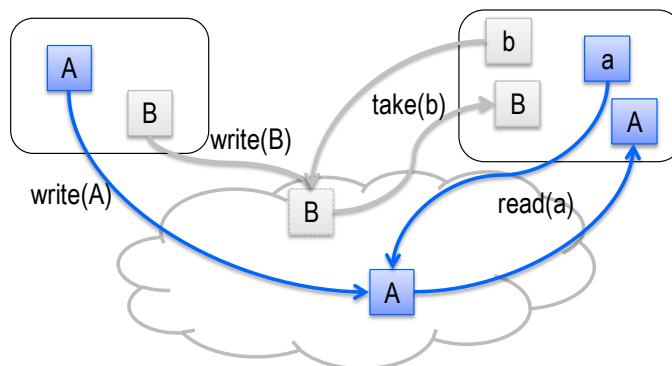
4. asinkrona komunikacija: objavljavač šalje poruku i nastavlja obradu neovisno o odgovoru od strane odredišta
5. pokretanje komunikacije na načelu *push*: objavljavač šalje poruku posredniku koji je proslijeđuje preplatnicima bez prethodnog eksplicitnog zahtjeva
6. personalizacija primljenog sadržaja: filtriranje objavljenih poruka prema preplatama
7. proširivost sustava: dodavanje novog objavljavača ili preplatnika ne utječe na ostale strane u komunikaciji
8. skalabilnost: raspodijeljena arhitektura

### 3.5 Dijeljeni podatkovni prostor

Dijeljeni podatkovni prostor se temelji na dijeljenoj perzistentnoj memoriji u koju se pohranjuju podaci (tzv. *tuple*), a može biti implementirana i raspodijeljeno (Ince, 2004, pp. 94-99). Pojavljuje se u ranim 1980-tim kada je nastao programski jezik Linda (Bacon & Harris, 2003, pp. 424-426). Procesi ne koriste poruke za komunikaciju, već podatke koje čine objekti programskog jezika: npr. JavaSpaces<sup>3</sup> pohranjuje svaki tuple u serijaliziranom obliku u dijeljeni podatkovni prostor. Proces podatke iz prostora čita tako da definira predložak i iz prostora mu se dostavlja podatak koji odgovara danom predlošku i na taj se način realizira komunikacija između izvořišta i skupine odredišta. Ideja je slična modelu objavi-preplati s tom razlikom što odredišta eksplicitno čitaju podatke iz dijeljenog prostora za razliku od aktivne isporuke podataka.

Operacije podržane ovim modelom su:

- **write (t)**: dodaj tuple *t* u raspodijeljeni podatkovni prostor
- **read (s) → t**: vraća tuple *t* koji odgovara predlošku *s*
- **take (s) → t**: vraća tuple *t* koji odgovara predlošku *s* i briše ga iz podatkovnog prostora

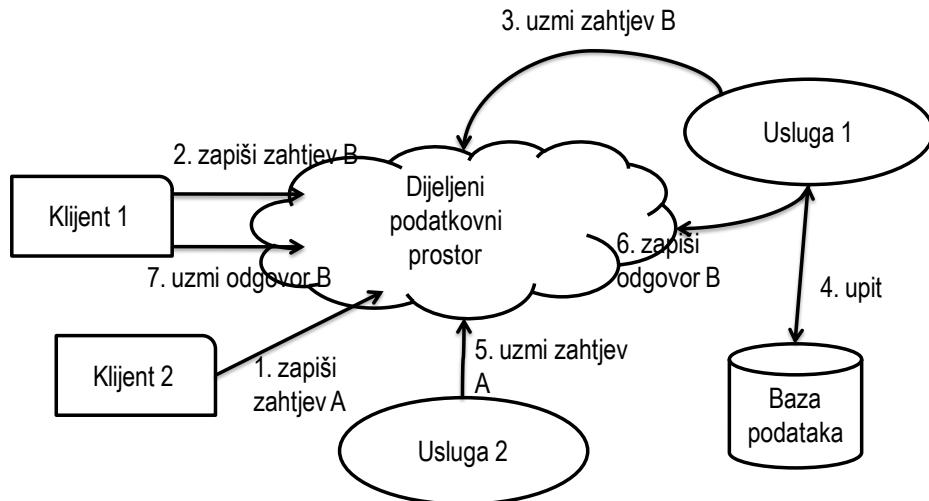


Slika 3.25. Primjer operacija u dijeljenom podatkovnom prostoru

Na primjeru sa slike (Slika 3.25) lijevi proces zapisuje podatke A i B u dijeljeni prostor. Desni proces definira 2 predloška *a* i *b*. Pozivom metode *read(a)*, iz prostora se čita A, a kopija A ostaje i dalje u prostoru. Pozivom metode *take(b)* iz prostora se čita B i briše iz prostora.

Slika 3.26 prikazuje primjer arhitekture usluge koja koristi dijeljeni podatkovni prostor radi uravnoteženja opterećenja i dijeljenja zahtjeva između dviju usluga (Usluga 1 i usluga 2). Pokazuje se asinkronost interakcije između usluga koje iz prostora čitaju zahtjeve koji odgovaraju njihovom predlošku, potom ih obrađuju, a odgovore također pohranjuju u dijeljeni prostor. Klijent čita odgovor također iz dijeljenog prostora i pri tome mora znati odgovarajući predložak odgovora.

<sup>3</sup><http://www.oracle.com/technetwork/articles/javase/javaspaces-140665.html>



Slika 3.26. Primjer arhitekture usluge koja koristi dijeljeni podatkovni prostor

Obilježja komunikacije pomoću dijeljenog podatkovnog prostora su:

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

### 3.6 Pitanja za učenje i ponavljanje

- 3.1. Objasnite na primjeru razliku između perzistentne i tranzientne komunikacije.
- 3.2. Objasnite zašto tranzientna sinkrona komunikacija potencijalno pati od problema vezanih uz skalabilnost.
- 3.3. Poslužitelj je implementiran pomoću *socketa TCP* s ograničenjem veličine repa za dolazne zahtjeve na instanci klase *ServerSocket* (tzv. *backlog*) na 10. Što se događa s novim zahtjevom ako je prilikom dolaska novog zahtjeva u tom repu već na čekanju 10 zahtjeva?
- 3.4. Na koji je način moguće ograničiti broj paralelnih konekcija prema višedretvenom poslužitelju kao što je npr. *UpperCaseServer*?
- 3.5. U tablicama su prikazane metode na klijentskoj i poslužiteljskoj strani *socketa TCP*. Upišite ispravan redoslijed izvođenja metoda u tablice.

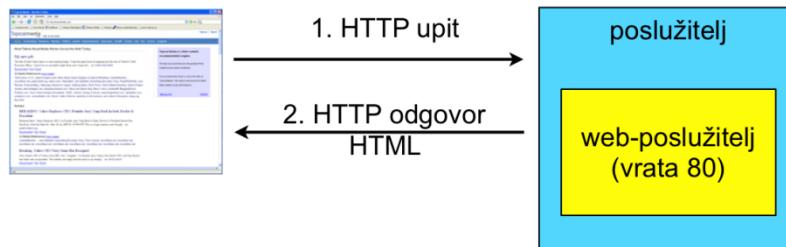
Klijent	Poslužitelj
socket()	listen()
write()	socket()
read()	accept()
close()	write()
connect()	read()
	close()
	bind()

- 3.6. Može li se pomoću UDP-a implementirati protokol za pouzdanu komunikaciju između klijenta i poslužitelja? Ako može, na koji način?

- 3.7. Skicirajte tijek komunikacije između klijenta i poslužitelja te objasnite odgođeni sinkroni poziv udaljene procedure RPC (*Remote Procedure Call*).
- 3.8. Skicirajte model pozivanja udaljene metode Java RMI (*Remote Method Invocation*). Navedite korake u komunikaciji potrebne da bi klijent pozvao metodu dostupnu na poslužitelju, uz pretpostavku da klasa *stub* već postoji i dostupna je na klijentskoj strani.
- 3.9. Imali li smisla implementirati perzistentnu asinkronu komunikaciju pomoću RMI-ja? Zašto?
- 3.10. Objasnite razliku između transportne adrese i adrese repa u sustavima za komunikaciju porukama?
- 3.11. Skicirajte i objasnite primjer komunikacije porukama između dva procesa/objekta (primatelja i pošiljatelja). Kakva je komunikacija porukama s obzirom na vremensku ovisnost primatelja i pošiljatelja?
- 3.12. Navedite i objasnite operacije koje implementira programska infrastruktura dijeljenog podatkovnog prostora.
- 3.13. Navedite sličnosti i razlike komunikacije na načelu objavi-preplati i dijeljenog podatkovnog prostora.
- 3.14. Usporedite preplatu u sustavima objavi-preplati i predložak u sustavima s dijeljenim podatkovnim prostorom. Može li se koristeći navedenu međuopremu realizirati tzv. anonimnu komunikaciju (pošiljatelj ne zna adresu primatelja) i zašto?
- 3.15. Gdje se filtriraju obavijesti u raspodijeljenom sustavu objavi-preplati koji koristi preplavljivanje obavijestima?

## 4 ARHITEKTURE WEB-APLIKACIJA

Web odnosno WWW (*World Wide Web*) se temelji na modelu klijent-poslužitelj. Klijentska strana je realizirana programom kojeg nazivamo web-preglednikom (engl. *browser*), dok je poslužiteljska strana realizirana programom koji poslužuje web-stranice. Komunikacija između klijenta i poslužitelja odvija se protokolom HTTP (*Hypertext Transfer Protocol*). Klijent šalje zahtjev poslužitelju na dogovorenih vrata 80, poslužitelj obrađuje zahtjev i vraća odgovor (Slika 4.1). Odgovor je sadržaj web-stranice. Jedna web-stranica uobičajeno referencira nekoliko različitih datoteka u različitim formatima npr. HTML, CSS, slike itd. U tom slučaju preglednik nakon što dohvati inicijalni dokument u HTML-u počne ga interpretirati kako bi ga prikazao i ako mu je potrebna dodatna datoteka tj. resurs ponovno šalje upit poslužitelju. Za dohvaćanje svakog resursa se šalje po jedan upit.



Slika 4.1. Dohvaćanje web-stranice

### 4.1 Protokol HTTP

Protokol HTTP (*Hypertext Transfer Protocol*) je protokol aplikacijskog sloja. Protokol HTTP definira format i način razmjene poruka između web-klijenta i web-poslužitelja. Za razmjenu poruka koristi se tekstualan zapis koji je sličan formatu e-mail poruke i standarda MIME. Klijent uvijek šalje zahtjev, a poslužitelj na zahtjev šalje odgovor. Svaki zahtjev sastoji se od „metode“, resursa i inačice protokola koji se koristi. Naziv metoda potječe iz objektno-orientiranih jezika i predstavlja operaciju koju poslužitelj treba izvršiti. Resurs je zapravo URI web-stranice koju se želi dohvatiti, a inačica protokola je najčešće 1.1. Poslužiteljev odgovor je rezultat obrade zahtjeva pa u svakom odgovoru mora biti statusni kôd o uspješnosti obrade zahtjeva. Neki odgovori imaju i sadržaj zatraženog resursa.

Prva inačica protokola HTTP koja se koristila u praksi je 0.9. Imala je ograničene mogućnosti jer podržava prijenos samo hipertekstualnih dokumenata, a vrlo brzo se pokazala potreba za prijenosom slika. U svibnju 1996. definiran je HTTP 1.0 koji je objavljen u RFC-1945<sup>4</sup>. HTTP 1.0 podržava prijenos različitih vrsta podataka tako da koristi koncepte iz standarda MIME. Važno je da zadržava kompatibilnost unazad. 90-tih godina promet pomoću protokola HTTP na Internetu postaje dominantan tj. više podataka se prenese protokolom HTTP, nego FTP-om, Telnetom, POP-om ili SMTP-om. U to doba se pokazalo da protokol HTTP 1.0 ima ograničenja koja rezultiraju neučinkovitim prijenosom podataka. Neka ograničenja su:

- svako sjedište mora biti na drugom poslužitelju,
- preko jedne TCP-konekcije ostvaruje se jedan HTTP-zahtjev,
- nema podrške za upravljanje performancama – priručno spremište (*cache, proxy*), djelomično dohvaćanje resursa, ...

Zbog tih ograničenja i sporog načina rada WWW je prozvan „World Wide Wait“. U lipnju 1999. definirana je nova inačica protokola HTTP 1.1 koji je objavljen u RFC-2616<sup>5</sup>. I ta inačica protokola

<sup>4</sup> RFC-1945 Hypertext Transfer Protocol -- HTTP/1.0, <http://www.ietf.org/rfc/rfc1945.txt>

<sup>5</sup> RFC-2616 Hypertext Transfer Protocol -- HTTP/1.1, <http://www.ietf.org/rfc/rfc2616.txt>

zadržava kompatibilnost prema unazad. Dodatna poboljšanja vezana uz sigurnost i autentifikaciju korisnika objavljeni su u dokumentu RFC-2617<sup>6</sup>. Najvažnija poboljšanja su sljedeća:

- na jednom fizičkom računalu moguće je ostvariti više web-poslužitelja ("virtualni host"),
- trajne konekcije koje omogućuju prijenos više zahtjeva i odgovora preko jedne TCP konekcije,
- podrška za djelomično dohvaćanje sadržaja,
- bolja podrška za priručna spremišta (engl. *cache*) i posredničke poslužitelje (engl. *proxy*),
- pregovaranje o sadržaju datoteke tj. klijent definira jezik ili *encoding* podataka i
- bolji sigurnosni mehanizmi.

Primjer jednog HTTP-zahtjeva izgleda ovako:

1. GET /predmet/rassus HTTP/1.1
2. Host: www.fer.unizg.hr
3. User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
4. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8
5. Accept-Language: hr,en;q=0.7,en-us;q=0.3
6. Accept-Encoding: gzip, deflate
7. Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.7
8. Connection: keep-alive
9. Cache-Control: max-age=0
- 10.

Napomena: Zahtjev ne sadrži redne brojeve, već su oni u ovom primjeru oni dodani radi preglednosti.  
Zahtjev se može sastojati od:

- početnog retka (redak 1) koji je obavezan,
- zaglavla koja se sastoje od polja (retci 2-9), neka polja su obavezna,
- prazan redak (redak 10) koji je obavezan i
- tijelo poruke (u ovom slučaju je nema, ali dolazi iza pravnog retka).

Obavezni dijelovi su:

- početni redak (redak 1),
- dio zaglavla s poljem `Host` (redak 2) i
- prazan redak (redak 10).

Početni redak se sastoji od metode koja se koristi, u ovom slučaju je to metoda `GET`, zatim od putanje do resursa (`/predmet/rassus`) i inačice protokola (`HTTP/1.1`). Polje zaglavla `Host` iza dvotočke i razmaka sadrži ime poslužitelja sa kojeg se traži resurs. Iz ta dva obavezna dijela može se konstruirati URI koji kada upišemo u web-preglednik generira zahtjev sličan ovome. URI je:

`http://www.fer.unizg.hr/predmet/rassus`

Početni dio označava protokol koji koristimo što je ovdje HTTP, iza toga dolazi dvotočka pa `//` iza kojeg se navodi simboličko ime računala (engl. *host*) te putanja (engl. *path*) do resursa.

Metoda zahtjeva određuje što se traži od resursa tj. što će poslužitelj napraviti s tim zahtjevom. U HTTP-u 1.1 definirano je 8 metoda i omogućeno je dodavanje novih metoda pomoću ekstenzija (engl. *extensions*). Definirane metode su sljedeće:

- `OPTIONS` – služi za informiranje i dohvaćanja mogućnosti resursa i poslužitelja,
- `GET` – koristi se za dohvaćanje resursa (najčešće u uporabi),
- `HEAD` – koristi se za dohvaćanje podataka o resursu (npr. veličina, provjera postojanja),
- `POST` – služi za aktiviranje resursa (npr. slanje podataka obrazaca),
- `PUT` – postavljanje resursa (npr. promjena podataka u web-uslugama),
- `DELETE` – brisanje resursa (npr. kod web-usluga),

---

<sup>6</sup> RFC-2617 HTTP Authentication: Basic and Digest Access Authentication, <http://www.ietf.org/rfc/rfc2617.txt>

- TRACE – koristi se za dijagnostiku i
- CONNECT – predviđeno za buduću uporabu.

Slijedi primjer jednog HTTP-odgovora:

```

1. HTTP/1.1 200 OK
2. Date: Wed, 12 Oct 2011 08:19:32 GMT
3. Server: Apache/2.2.20 (FreeBSD) mod_ssl/2.2.20 OpenSSL/0.9.8q mod_fcgid/2.3.6
4. Expires: Thu, 19 Nov 1981 08:52:00 GMT
5. Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
6. Pragma: no-cache
7. P3P: CP="NOI CURa ADMa DEVa TAIa PSAa PSDa IVAa IVDa HISa OTPa OUR BUS IND
UNI COM NAV INT"
8. Set-Cookie: CMS=2p72ge55hqm92itadsfu83pc25; expires=Wed, 19-Oct-2011 20:19:32
GMT; path=/; domain=www.fer.unizg.hr; HttpOnly
9. Vary: Accept-Encoding
10. Transfer-Encoding: chunked
11. Content-Type: text/html; charset=utf-8
12. Content-Length: 5045
13.
14. d9e7
15. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
16. <html xmlns="http://www.w3.org/1999/xhtml">
17. <head>
18. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
...

```

Odgovor se sastoji od istih osnovnih dijelova kao i zahtjev: početnog retka, polja zaglavlja, praznog retka i tijela poruke. Početni redak započinje inačicom protokola nakon koje dolazi statusni kod iza kojeg je opis odgovora. U ovom primjeru se koristi protokol HTTP 1.1, statusni kod je 200, a opis odgovora je OK. Statusni kod je podijeljen u kategorije:

- 1xx – *Informative* – označava odgovor koji je informativne prirode, tj. niti uspjeh, niti neuspjeh,
- 2xx – Uspjeh - poslužitelj je primio, razumio i ispunio zahtjev,
- 3xx – Preusmjeravanje - potrebno poduzeti dodatne akcije,
- 4xx – Greška na klijentu – klijent je posao neispravan zahtjev i
- 5xx – Greška na poslužitelju – klijent je posao ispravan zahtjev, ali ga poslužitelj ne može ispuniti zbog greške na poslužitelju.

Ako je poslužitelj uspješno ispunio zahtjev za dohvaćanjem nekog resursa onda se u tijelu odgovora prenosi sadržaj tog resursa. U tom slučaju su važna polja zaglavlja Content-Length i Content-Type koji klijentu prenose duljinu tijela i vrstu sadržaja.

## 4.2 Jezik HTML

HTML (*Hypertext Markup Language*) je jezik za označavanje (engl. *markup*) teksta. U običan tekst se umeću oznake koje utječu na predočavanje teksta i služe za uvođenje hiperveza (poveznica). HTML standardizira tijelo W3C (*World Wide Web Consortium*)<sup>7</sup>. Prva inačica jezika je izašla 1992. godine i imala je mogućnost označavanja dijelova teksta tako da drugačije prikazuju (npr. naslov, podebljano, koso, novi red, ...) u web-pregledniku i imala je mogućnost stavljanja poveznica na druge dokumente. Trenutno je posljednja standardizirana inačica 4.01 iz 1999. godine<sup>8</sup>. U međuvremenu se pojavio jezik XML koji je također jezik za označavanje, a koristi proizvoljno definirane oznake uz strožu sintaksu za razliku od HTML-a. Stoga je zbog strože sintakse lakše je implementirati program koji čita i analizira XML dokumente. Zbog toga se javlja jezik XHTML (*Extensible Hypertext Markup Language*) koji je

<sup>7</sup>W3C - <http://www.w3.org>

<sup>8</sup>HTML 4.01 Specification, W3C Recommendation 24 December 1999 <http://www.w3.org/TR/html401/>

zapravo HTML sa strožom sintaksom pa je ujedno i jezik po standardima XML-a. Najnovija verzija HTML inačice 5 je u izradi koja je započeta 2004. godine, prva javna verzija prijedloga standarda je objavljena 2008., a trenutno je objavljen *draft 25* iz listopada 2012<sup>9</sup>. Mnogi danas popularni web-preglednici podržavaju većinu objavljenih funkcija jezika HTML5<sup>10</sup>.

U nastavku su navedene osnove HTML-a. Primjer jedne oznake je:

```
<body>Ovo je neki tekst</body>
```

U ovom primjeru vidimo oznaku pod imenom `body` koja označava tekst: Ovo je neki tekst. Svaka oznaka se sastoji od dvije oznake:

- otvarajuće - `<imeOznake>` - riječ unutar znakova `< i >`
- zatvarajuće - `</imeOznake>` - riječ unutar znakova `</ i >`.

Ako imamo otvarajuću i zatvarajuću oznaku bez sadržaja kao u sljedećem primjeru

```
<br></br>
```

onda ju možemo pisati u skraćenom obliku `<br/>`.

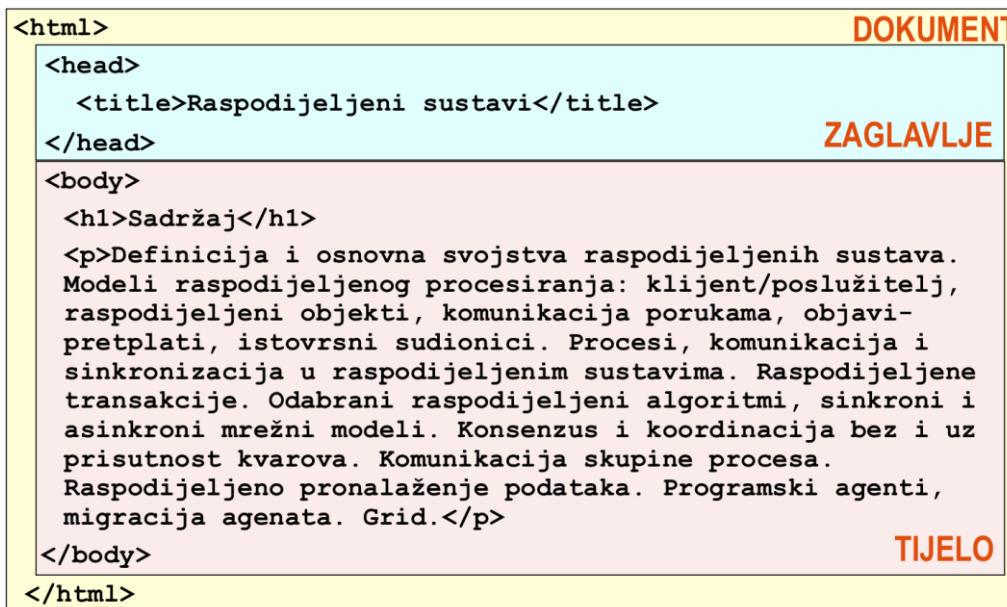
Svaka oznaka može imati i atribut. Atribut se postavlja u otvarajući dio oznake ili se može postaviti i u skraćeni oblik ako nema sadržaja oznake. Sljedeći primjer pokazuje kako se postavlja atribut u otvarajući dio oznake:

```
<body class="tijelo">
```

U ovom primjeru imamo atribut koji ima naziv `class`, a vrijednost atributa je `tijelo`. Ako imamo više atributa onda ih jednostavno navodimo jedan iza drugoga kao u sljedećem primjeru:

```
<body id="naziv" class="tijelo">
```

U prethodnom primjeru imamo dva atributa pod nazivima `id` i `class`. Oznaka ne smije imati više atributa s istim imenom.



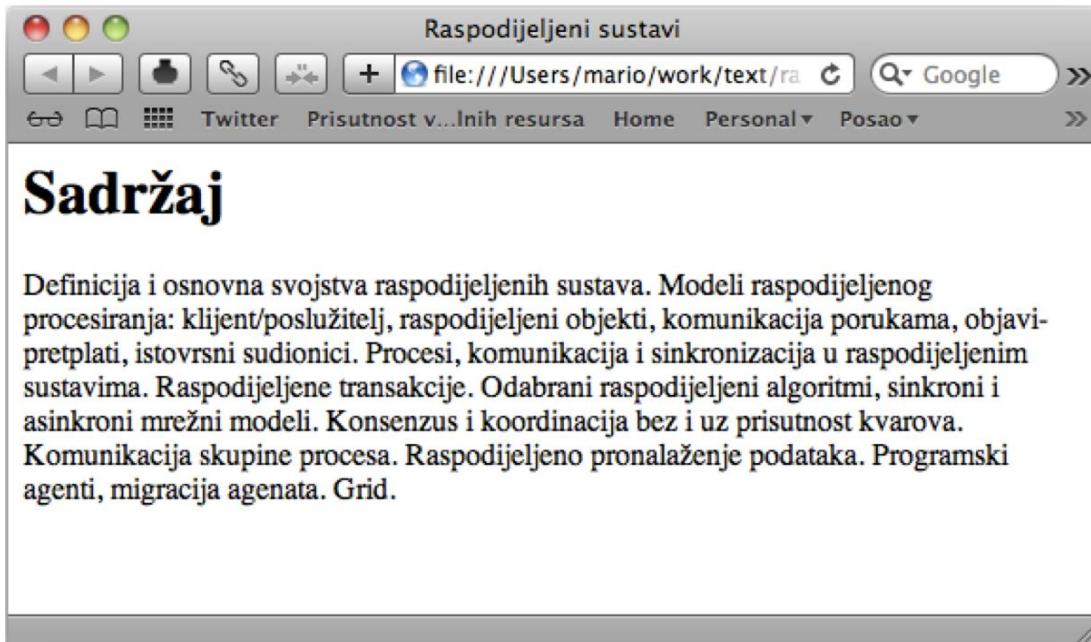
Slika 4.2. Struktura HTML-ovog dokumenta

<sup>9</sup>HTML5, A vocabulary and associated APIs for HTML and XHTML, Working Draft 25 October 2012, <http://dev.w3.org/html5/spec/Overview.html>

<sup>10</sup>HTML5 & CSS3 READINESS, <http://html5readiness.com/>

Struktura jednog HTML-ovog dokumenta prikazana je na slici (Slika 4.2), a sastoji se od zaglavlja i tijela. Dokument započinje oznakom `html`. Unutar nje nalazi se oznaka `head` u koju se postavljaju elementi zaglavlja dokumenta kao primjerice `title` unutar kojega se navodi naslov dokumenta. Druga oznaka unutar oznake `html` je `body`, a prenosi tijelo dokumenta. U oznaci `body` se nalazi sadržaj dokumenta kao što je oznaka `h1` koja predstavlja naslov 1. razine ili oznaka `p` koja označava jedan odlomak odnosno paragraf. Ovakav dokument se snimi kao datoteka s nastavkom `html` ili `htm`.

Na slici (Slika 4.3) je prikazan izgled ovog dokumenta u pregledniku Safari.



Slika 4.3. Izgled HTML-ovog dokumenta u pregledniku

Osnovni elementi HTML-a su sljedeći:

- naslovi: `h1, h2, ... h6` – `<h1> naslov ... </h1>`
- tekstualni elementi:
  - `<p>` odlomak ili paragraf `</p>`
  - `<br/>` - prelazak u novi red
  - `<hr/>` - horizontalna crta
  - logički stilovi:
    - `<em>istaknuti tekst</em>`
    - `<strong>pojačano označen tekst</strong>`
    - `<code>izvorni kod programa</code>`
  - fizički stilovi:
    - `<b>podebljani tekst</b>`
    - `<i>ukošen tekst</i>`
- poveznice:
  - obična:
    - `<a href="http://www.fer.unizg.hr">Tekst koji klikom odlazi na FER-ov web.</a>`
  - prikazuje sliku:
    - ``
  - slanje emaila:
    - `<a href="mailto:webmaster@fer.hr">Šalji e-mail</a>`
  - imenovane pozicije u dokumentu:
    - `<a name="imePozicije">neki tekst</a>`

- <a href="#imePozicije">poveznica na poziciju</a>
- liste:
  - neporedane (točke)
    - <ul>
 <li>stavka</li>
 <li>stavka</li>
 </ul>
  - poredane (brojevi)
    - <ol>
 <li>stavka</li>
 ...
 </ol>
  - definicijske (objašnjenja)
    - <dl>
 <dt>pojam</dt>
 <dd>definicija</dd>
 ...
 </dl>
- tablica – primjer slijedi
  - <table border="1">
 <tr>
 <th>zaglavlje tablice - stupac 1</th>
 <th>zaglavlje tablice - stupac 2</th>
 </tr>
 <tr>
 <td>tekst 1. retka 1. stupca</td>
 <td>tekst 1. retka 2. stupca</td>
 </tr>
 </table>
- obrasci
  - <form action="http://..." method="post ili get">
 <input type="text" name="email" size="40" maxlength="50"/>
 <input type="password"/>
 <input type="checkbox" checked="checked"/>
 <input type="radio" checked="checked"/>
 <input type="submit" value="Šalji"/>
 <input type="reset"/>
 <input type="hidden"/>
 <select>
 <option>predavanja</option>
 <option selected="selected">laboratorij</option>
 <option>zadaća</option>
 </select>
 <textarea name="komentar" rows="60" cols="20"></textarea>
 </form>

Više informacija o HTML-u može se pronaći u standardu<sup>11</sup> ili u vodiču<sup>12</sup>.

Najveći problemi dizajnera i programera koji koriste HTML je u tome što preglednici različito vizualno prikazuju pojedine oznake tj. dijelove dokumenta. Najviše problema se veže uz starije verzije preglednika, npr. Internet Explorer 6, koji se u velikoj mjeri ne pridržavaju standarda. Drugi značajan problem je drugačije programsko sučelje za pristupanje elementima HTML-a iz JavaScripta, pa je prije pristupanju elementima potrebno prvo provjeriti koji se preglednik koristi i na temelju toga onda izvršiti dio kôda koji je prilagođen tom pregledniku.

<sup>11</sup>HTML 4.01 Specification, <http://www.w3.org/TR/html401/>

<sup>12</sup>HTML Tutorial, <http://www.w3schools.com/html/>

#### 4.2.1 HTML 5

HTML-ova inačica 5 još uvijek nije standard i prema procjenama još neko duže vrijeme neće postati standard. No, kako se današnja tehnologija brzo mijenja, razvijatelji preglednika počeli su ugrađivati podršku za HTML5 tako da većina najpopularnijih preglednika podržava veliku većinu funkcija iz prijedloga standarda. Najvažnija svojstva HTML-a 5 su:

- manje kôda u zaglavlju HTML-a,
- više semantike u oznakama (npr. header, nav, article, section, aside, footer, figure, figurecaption,...),
- oznake za višemedijski sadržaj (video, audio),
- geolokacija u JavaScriptu,
- platno za crtanje iz JavaScripta (canvas),
- nove vrste podataka u obrascima,
- spremnik podataka za rad bez mreže,
- podrška za povlačenje elemenata (drag),
- komunikacija između domena (različiti izvori koda),
- sadržaj koji se može uređivati (editable),
- mrežne utičnice (WebSocket), ...

Ako želite ispitati koja svojstva iz standarda podržava vaš preglednik otvorite stranicu<sup>13</sup>.

### 4.3 Jezik CSS

Jezik CSS (*Cascading Style Sheets*) je jezik za definiranje pravila prikaza sadržaja HTML-ovog dokumenta, a nastao je iz potrebe odvajanja sadržaja i prikaza dokumenta. Inicijalno je HTML miješao sadržaj s pravilima za njegovo formatiranje i pozicioniranje, primjerice koristile su se oznake font i align, što je prisiljavalo dizajnere stranica da upute o prezentaciji teksta uključuju u sam HTML-ov dokument. Pogledajmo sljedeći primjer.

```
...
<p align="center">
<font face="Arial,Helvetica,Sans-serif" size="15px" color="#ff0000">Raspodijeljeni
sustavi</font>
</p>
...
```

Ako u jednom dokumentu postoji više dijelova koji trebaju biti prikazani na isti način, onda je potrebno kopirati oznake za formatiranje na svako od tih mesta. Ako nakon toga želimo promijeniti izgled tih dijelova, potrebno je također na svakom mjestu promijeniti i označke. Ovaj se problem dalje multiplicira na razinu web-sjedišta (engl. site) koje sadrži desetke ili stotine dokumenata. To je bio jedan od razloga uvođenja jezika CSS. Drugi razlog je taj što se na ovaj način odvaja opis prikaza od samog sadržaja kojeg je onda lakše prepoznati i promijeniti. Dodatna pogodnost je ta da se mogu definirati različiti stilovi za isti dokument koji mogu biti prilagođeni za različite veličine ekrana i ispisa (monitor, printer, PDA, pametni telefon).

Jezik CSS je standardiziralo tijelo W3C<sup>14</sup> i njegova primjena nije isključivo za HTML nego se može koristiti i za prezentaciju različitih vrsta dokumenata kao što su: XML, XHTML, SVG i XUL. Postoje 3 razine CSS-a:

- CSS level 1 – objavljen 1996., a cilj je bilo zamijeniti postojeće oznake u HTML-u;

---

<sup>13</sup>THE HTML5 TEST – HOW WELL DOES YOUR BROWSER SUPPORT HTML5?, <http://html5test.com/>

<sup>14</sup>Cascading Style Sheets home page, <http://www.w3.org/Style/CSS/>

- CSS level 2 – objavljen 1998., dodane su nove mogućnosti kao što su absolutno i relativno pozicioniranje, `z-index` koji definira što je iznad, a što ispod elemenata koji se prekrivaju, itd.;
- CSS level 3 – uvodi module koji definiraju pojedini dio CSS-a jer su prethodne specifikacije vrlo opsežne tako da danas postoji više od 40 modula.

CSS-ovi stilovi mogu biti spremljeni u HTML-ov dokument unutar oznake `<style type="text/css"></style>` koja se obično nalazi unutar oznake `head`. Na taj način možemo definirati stilove za jedan dokument na jednom mjestu. Pogledajmo prethodni primjer koji je napravljen pomoći CSS-a u istom dokumentu:

```
<html>
<head>
<style type="text/css">
p {
  color: #ff0000;
  font: 15px Arial,Helvetica,Sans-serif;
  text-align:center;
}
</style>
</head>
<body>
<p>Raspodijeljeni sustavi</p>
...
</body>
</html>
```

U ovom primjeru se u jednom dokumentu definira njegov stil i sadržaj, no vidljivo je da je stil odvojen od samog sadržaja dokumenta te je dokument pregledniji. Nedostatak ovog pristupa je taj što se isti stil ne može koristiti u različitim dokumentima tj. moramo kopirati stil u svaki dokument na web-sjedištu. Drugi način korištenja stilova odvaja stil u poseban dokument s nastavkom `css` koji se potom uključi u zaglavljje HTML-ovog dokumenta. Takav način korištenja je pokazan na sljedećem primjeru:

```
<html>
<head>
<title>Raspodijeljeni sustavi</title>
<link rel="stylesheet" type="text/css" href="primjer.css"/>
</head>
<body>
<h1>Sadržaj</h1>
<p>Definicija i osnovna svojstva raspodijeljenih sustava. Modeli raspodijeljenog procesiranja: klijent/poslužitelj, raspodijeljeni objekti, komunikacija porukama, objavi-preplati, istovrsni sudionici. Procesi, komunikacija i sinkronizacija u raspodijeljenim sustavima. Raspodijeljene transakcije. Odabrani raspodijeljeni algoritmi, sinkroni i asinkroni mrežni modeli. Konsenzus i koordinacija bez i uz prisutnost kvarova. Komunikacija skupine procesa. Raspodijeljeno pronalaženje podataka. Programski agenti, migracija agenata. Grid.</p>
</body>
</html>
```

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

- `rel` – označava relaciju između HTML-ovog dokumenta i povezanog dokumenta, za stilove se koristi vrijednost `stylesheet`;
- `type` – specificira vrstu dokumenta prema standardu MIME, za stilove se koristi `text/css`;
- `href` – specificira lokaciju dokumenta pomoći URL-a.

U ovom primjeru je stil definiran u datoteci `primjer.css` koja slijedi:

```
/* ovo je komentar */
body {
  background-color:#d0e4fe;
}
```

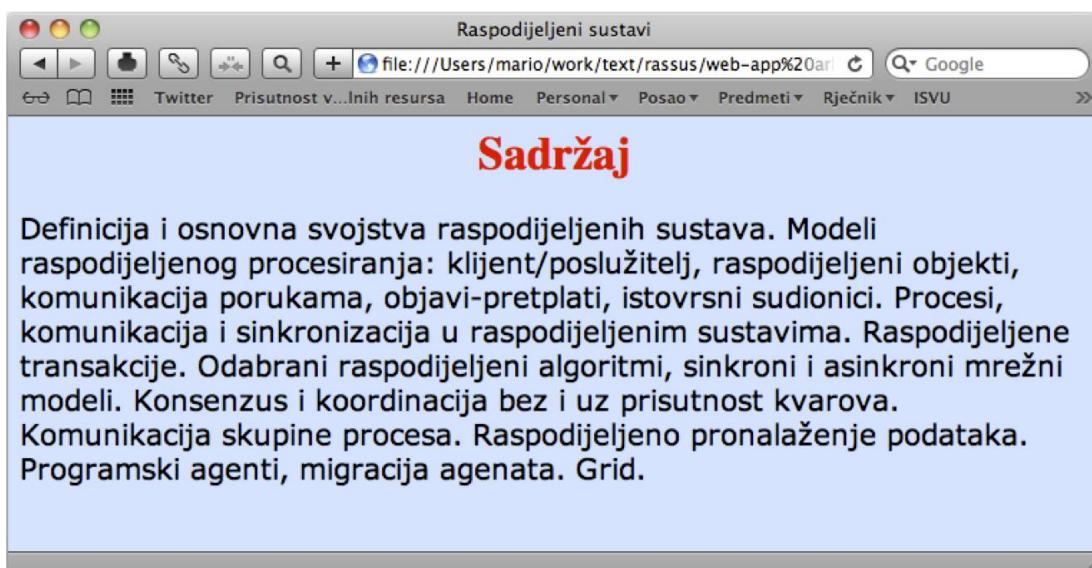
```

h1 {
    color:red;
    text-align:center;
}

p {
    font-family:verdana, helvetica, sans-serif;
    font-size:20px;
}

```

Kada web-preglednik pristupa resursu, tj. HTML-ovom dokumentu koji referencira stil u posebnoj datoteci, prvo se učita HTML-ov dokument koji se kreće interpretirati kako bi se prikazao. Preglednik nailazi na oznaku link koja označava pregledniku da je za prikaz potreban i dokument primjer.css te stoga preglednik šalje novi GET zahtjev za njegovim dohvaćanjem. Kada se i taj dokument dohvati onda preglednik može prikazati dokument u prozoru. Prethodni primjer je prikazan na slici (Slika 4.4).



Slika 4.4. Izgled web-stranice sa stilovima

Pogledajmo sada kako se definiraju pravila stilova. Ako želimo postaviti komentare u dokument sa stilovima onda komentar započinje znakovima /\*, a završava \*/. Komentar može biti i preko više redaka. Sve što se nalazi unutar tih znakova je komentirano.

Dokument sa stilovima se sastoji od niza pravila. Svako pravilo započinje selektorom koji definira na koju oznaku u HTML-u se odnosi taj stil. Ako stil nije definiran za neku oznaku, onda se koristi inicijalni stil preglednika, a inače se koristi specifični stil, tj. mijenjaju se svojstva redefinirana u novom stilu. Pogledajmo primjer jednog pravila koji se odnosi na oznaku h1:

```

h1 {
    color:red;
    text-align:center;
}

```

Ovaj stil vrijedi za sve oznake h1 u HTML-ovom dokumentu. Nakon selektora (h1) se unutar vitičastih zagrada navodi lista deklaracija. U primjeru imamo dvije deklaracije. Svaka deklaracija se sastoji od imena svojstva koje se postavlja iza koje slijedi dvotočka i nakon dvotočke se navodi vrijednost svojstva. Na kraju svaka deklaracija završava točka-zarezom. Prvo svojstvo koje se postavlja je svojstvo color, a vrijednost je red. Svojstvo color označava boju kojom će se tekst prikazati. Vrijednost može biti ime boje koje su definirane ili komponentni opis boje pomoću RGB-a. Drugo svojstvo koje se postavlja je text-align, a vrijednost je center. Ono označava da će tekst biti

centralno pozicioniran u retku. Sva svojstva, njihove moguće vrijednosti i što znače pregledniku tj. kako će preglednik prikazati to pravilo definirano je standardima.

Selektori mogu odabirati različite oznake. U primjeru smo vidjeli da je selektor odabrao sve oznake h1. Selektori mogu odabratи:

- oznake,
  - h1 { ... } - pravilo vrijedi za sve oznake h1
- atribut id (identifikator),
  - #prozor { ... } - pravilo vrijedi za oznaku koja ima atribut id s vrijednošću prozor
  - u jednom HTML-ovom dokumentu vrijednosti id-a su jedinstvene
- atribut class (klasa/vrsta),
  - .center { ... } - pravilo vrijedi za sve oznake koje imaju atribut class s vrijednošću center
- složeni odabiri:
  - p.center { ... } - oznake p s atributom class koji ima vrijednost center
  - #okvir p { ... } - oznake p koje se nalaze unutar oznake čiji je id okvir
  - ...

Više o svojstvima i njihovim vrijednostima pogledajte u vodiču za CSS<sup>15</sup>.

#### 4.4 Priručna spremišta

Priručno spremište (engl. *cache*) se u raspodijeljenom sustavu koristi za pohranjivanje kopije resursa na mjestu povoljnijem za njegovo korištenje npr. na mjestu koje smanjuje mrežnu udaljenost klijenta i poslužitelja. Osnovna ideja priručnih spremišta je sljedeća: dohvati resurs jednom i pohrani ga u memoriju priručnog spremišta, a vrati kao rezultat više puta. Priručno spremištem prilikom prvog zahtjeva dohvaća resurs s izvornog poslužitelja i spremi ga u svoju memoriju. Kada klijent zatraži isti resurs, taj se više ne mora dohvaćati s udaljenog poslužitelja već se samo prosljeđuje kopija iz memorije priručnog spremišta. Uporaba priručnog spremišta nosi dobrobiti za 1) korisnika, 2) izvorni poslužitelj i 3) mrežu na način da skraćuje vrijeme odziva sustava opaženo od strane korisnika, smanjuje opterećenje na izvornom poslužitelju s obzirom da odgovara na manji broj zahtjeva te smanjuje generirani mrežni promet ako se priručno spremište nalazi u blizini klijenata (mjereno mrežnom odaljenošću).

Smještaj priručnog spremišta može biti:

- na klijentu – npr. web-preglednik spremište sadržaj na disk klijentskog računala te buduće korisničke zahtjeve poslužuje pomoću lokalne kopije resursa;
- na strani izvornog poslužitelja – kopija izvornog resursa (npr. HTML-ov dokument) ili izračunati rezultat (npr. odgovor web-tražilice na neki popularni upit) se spremi za kasniju dostavu klijentu. Ako neki klijent zatraži isti resurs onda mu pružatelj usluge može odmah proslijediti traženi sadržaj i ne mora ponovno izvršavati kôd koji generira rezultat. Na taj se način smanjuje opterećenje izvornog poslužitelja pružatelja usluge;
- negdje u mreži – poslužitelj posrednik (engl. *caching proxy*) koji je obično u mreži blizu korisnika (npr. posrednički poslužitelj u nekoj organizaciji). Na taj način se smanjuje mrežni promet između organizacije i mreže pružatelja usluge.

Kako bi privremeno spremište ispravno funkcionalo, potrebno je definirati pravila o tome što se može pohraniti u privremeno spremište te koliko dugo. Postoje dva modela za upravljanje sadržajem koji se može pohraniti u privremeno spremište, a ugrađena su u protokol HTTP:

---

<sup>15</sup>CSS Tutorial, <http://www.w3schools.com/css/>

- model roka trajanja i
- model validacije tj. provjere valjanosti dokumenta.

#### 4.4.1 Model roka trajanja

Pomoću modela roka trajanja poslužitelj može definirati koliko dugo će neki resurs biti valjan tj. „svjež“. Klijenti i priručna spremišta takav resurs mogu pohraniti lokalno te ga prilikom sljedećeg zahtjeva dohvaćaju iz spremišta. Tek kada rok trajanja resursa istekne za budući zahtjev je potrebno kontaktirati poslužitelja kako bi priručno spremište provjerilo valjanost resursa sa svojom lokalnom kopijom.

Model roka trajanja je dobar mehanizam kada se resurs mijenja u poznato vrijeme ili se mijenja vrlo rijetko. Primjer takvih resursa su slike za koje se uobičajeno postavlja valjanost od 24 sata. Preglednik će takve resurse učitati jednom i u sljedeća 24 sata će ih koristiti iz priručnog spremišta bez obzira koliko puta korisnik pristupio nekoj web-stranici sa sjedišta.

Ako su resursi dinamički tj. generiraju se korištenjem poslužiteljskih skripti kao što je npr. vremenska prognoza, onda ne možemo koristiti ovaj model.

Ovaj model u protokolu HTTP koristi zaglavljje `Expires`. Najjednostavniji slučaj je kada to zaglavljje sadrži datum i vrijeme do kada je rok trajanja ovog resursa:

```
Expires: Sat, 15 Oct 2011 11:49:28 GMT
```

Polje zaglavja `Expires` radi ispravno kada su satovi na klijentu i poslužitelju sinkronizirani. Zbog toga je u protokolu HTTP 1.1 dodano zaglavljje `Cache-Control` koje nudi fleksibilniju kontrolu. Pogledajmo primjer:

```
Cache-Control: max-age=86400, must-revalidate
```

Ovo zaglavljje može imati različite vrijednosti koje kontroliraju upravljanje priručnim spremištem. U prikazanom primjeru polje ima vrijednost `max-age=86400` što znači da je rok trajanja ovog resursa sljedećih 86400 sekundi, dakle imamo relativno vrijeme u odnosu na sadašnji trenutak. Druga vrijednost je `must-revalidate` koja znači da priručno spremište nakon isteka roka valjanosti mora kontaktirati izvorni poslužitelj i provjeriti valjanost resursa.

#### 4.4.2 Model validacije

Model validacije omogućuje klijentu da provjeri je li pohranjena inačica resursa i dalje svježa (koristi se uvjetni GET zahtjev). Ako klijent nema svježu inačicu, poslužitelj će u odgovoru poslati svježu inačicu, a u suprotnom poslužitelj šalje odgovor 304 *Not Modified* bez resursa.

Ovaj model je koristan jer smanjuje mrežni promet između poslužitelja i klijenta. Ako se resurs generira dinamički te se u međuvremenu nije promijenio, onda poslužitelj na ovaj način štedi na vremenu obrade zahtjeva. Ovaj model je dobar u situacijama kada se ne zna vrijeme kada se resurs može promjeniti.

HTTP nudi dvije vrste zaglavja za implementaciju modela validacije: `Last-Modified` i `Etag`. Zaglavje `Last-Modified` je definirano u HTTP 1.0 i šalje klijentu datum i vrijeme kada je resurs zadnji put bio promijenjen. Primjer:

```
Last-Modified: Thu, 12 Feb 2009 15:00:22 GMT
```

Kada klijent pošalje uvjetni GET sa zaglavljem `If-Modified-Since` koje definira datum i vrijeme kada je resurs zadnji put promijenjen, poslužitelj mu u slučaju da je to svježa inačica odgovara s:

```
304 Not Modified
```

Ovo zaglavljje ima probleme kao i zaglavje `Expires` pa se umjesto njega u HTTP 1.1 koristi zaglavje `Etag`. Vrijednost zaglavja `Etag` je niz znakova koji jedinstveno identificiraju inačicu resursa. Svaki put kada se resurs promijeni mora se promijeniti i `Etag`. Uobičajena implementacija vrijednosti

zaglavljia `ETag` je *hash* vrijednost resursa ili URL-a, datuma i vremena promjene. Najčešća se koristi algoritam MD5.

Klijent u zahtjevu šalje zaglavljje `If-None-Match`, a vrijednost je *hash* inačice iz priručne memorije. Poslužitelj odgovara na isti način kao i kod korištenja zaglavljia `Last-Modified`.

#### 4.4.3 Posrednički poslužitelj

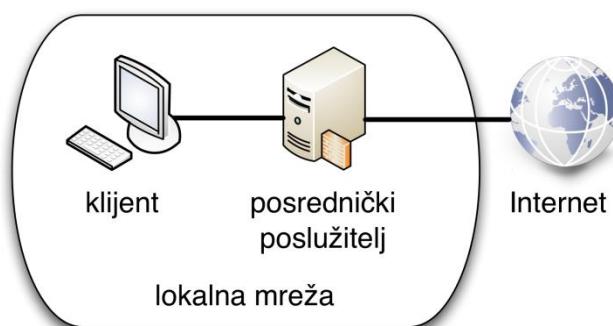
Posrednički poslužitelj posreduje između klijenta i poslužitelja usluge. Posrednički poslužitelji se mogu koristiti u različite svrhe:

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

Prema smještaju posrednički poslužitelji se mogu nalaziti u:

- lokalnoj mreži klijenta,
- javnoj mreži ili
- lokalnoj mreži na poslužiteljskoj strani.

Kada se posrednički poslužitelj nalazi u lokalnoj mreži klijenta (engl. *forward proxy*) onda se smanjuje korištenje veze između lokalne mreže i ostatka mreže što radi uštedu organizaciji (Slika 4.5). Organizacija može filtrirati i kontrolirati sadržaj pomoću ovakvog posredničkog poslužitelja.



Slika 4.5. Posrednički poslužitelj u lokalnoj mreži klijenta

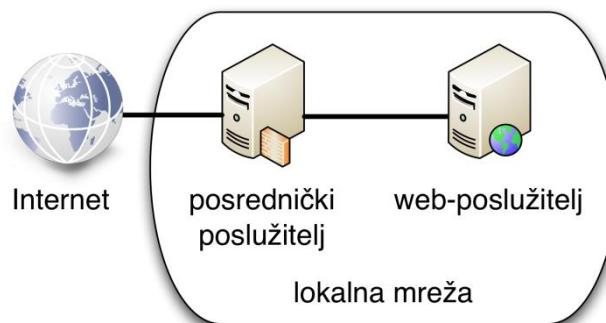
Posrednički poslužitelj u javnoj mreži (engl. *open proxy*) služi isključivo za anonimiziranje klijenata (Slika 4.6) tako da poslužitelj usluga nema pristup IP-adresi klijenta jer zahtjev dolazi sa posredničkog poslužitelja.



Slika 4.6. Posrednički poslužitelj u javnoj mreži

Posrednički poslužitelj u lokalnoj mreži poslužitelja (engl. *reverse proxy*) može pružatelju usluga poslužiti za sljedeće svrhe (Slika 4.7)

- enkripcija ili komprimiranje podataka – rasterće se poslužitelj usluge (npr. web-poslužitelj);
- uravnoteženje opterećenja – posrednički poslužitelj bira na koji će poslužitelj usluga usmjeriti zahtjev klijenta i na taj način više poslužitelja usluga mogu paralelno obrađivati zahtjeve klijenata;
- privremeno spremanje izračunatih podataka – posrednički poslužitelj može svježe resurse spremiti u svoje priručno spremište i njih slati kao odgovor te na taj način rasteretiti poslužitelja usluge;
- sigurnosne postavke – može dozvoliti pristup samo nekim uslugama i na taj način poslužitelj usluge ne mora biti toliko siguran jer ga štiti posrednički poslužitelj koji može osiguravati SSL komunikaciju između klijenta i posredničkog poslužitelja, a komunikacija između posredničkog poslužitelja i poslužitelja usluga ne treba biti zaštićena.



Slika 4.7. Posrednički poslužitelj u lokalnoj mreži na poslužiteljskoj strani

## 4.5 Web-aplikacije

Postoji puno definicija web-aplikacija. Najprikladnija se pokazala sljedeća<sup>16</sup>: „Web-aplikacije su ostvarene na web-poslužiteljima i koriste alate kao što su baze podataka, JavaScript (Ajax ili Silverlight), PHP (ili ASP.Net) kako bi se omogućile funkcionalnosti koje nisu standardne web-stranice ili web-obrasci.“

Web-aplikacije možemo podijeliti u dvije vrste:

- koje izgledaju kao normalne web-stranice (npr. portali) ili
- koje izgledaju kako „normalne“ aplikacije – imaju bogato korisničko sučelje (npr. Google Mail).

Tehnologije i programski jezici koji se koriste u web-aplikacijama su razni. Najpoznatije su: HTML, CSS, JavaScript, PHP, ASP, JSP, Flash, Ruby on Rails, Java Servlets, Cold Fusion, baze podataka (MySQL, PostgreSQL, SQL Server, Oracle DB, ...).

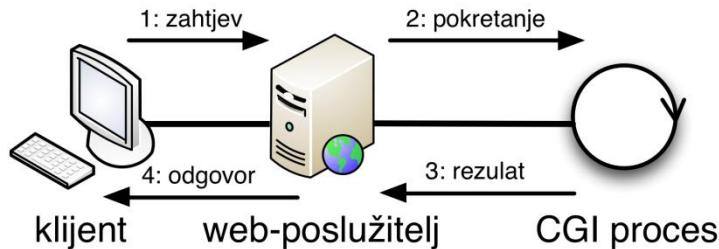
### 4.5.1 CGI

CGI (*Common Gateway Interface*)<sup>17</sup> je jednostavno sučelje za pokretanje eksternih programa iz web-poslužitelja na platformski i programski neovisan način (Slika 4.8). Ovo sučelje se koristi od 1993. godine, a definirano je u RFC-3875. Kod svakog zahtjeva se pokreće novi proces, a podaci između

<sup>16</sup>Web application definition, <http://webdesign.about.com/od/web20/g/web-application-definition.htm>

<sup>17</sup>RFC-3875 - The Common Gateway Interface (CGI) Version 1.1, <http://www.ietf.org/rfc/rfc3875>

poslužitelja i procesa šalju se preko varijabli okoline i tokova podataka. Nakon svake obrade proces se gasi. CGI se uobičajeno koristi u skriptnim jezicima kao što su Bash, Perl i ostali.

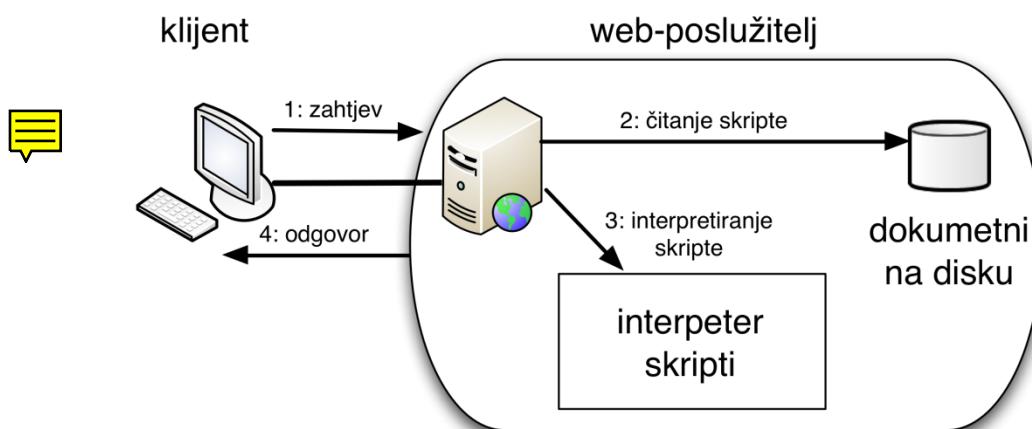


Slika 4.8. CGI

Nedostatak CGI-a je što se kod svakog zahtjeva pokreće novi proces i nakon obrade gasi što je zahtjevno za resurse (procesorsko vrijeme i memorija) pa kod velikog broja zahtjeva na poslužitelju to znatno utječe na performance.

#### 4.5.2 Poslužiteljske skripte

Drugi mehanizam koji se koristi su poslužiteljske skripte (engl. *server side scripts*). One isto kao i CGI dinamički generiraju HTML-dokumente, ali je razlika u tome što se za svaki zahtjev ne pokreće novi proces i na taj način se štede resursi. Primjeri jezika i tehnologija koje koriste ovaj mehanizam su: PHP, ASP, JSP, Django, Ruby on Rails, ...

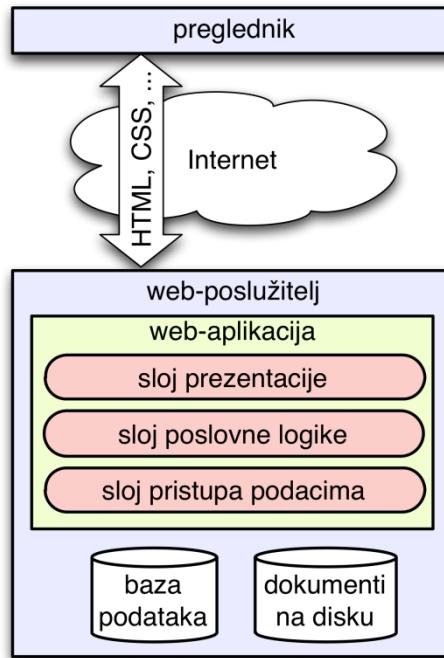


Slika 4.9. Poslužiteljske skripte

### 4.6 Arhitekture web-aplikacija

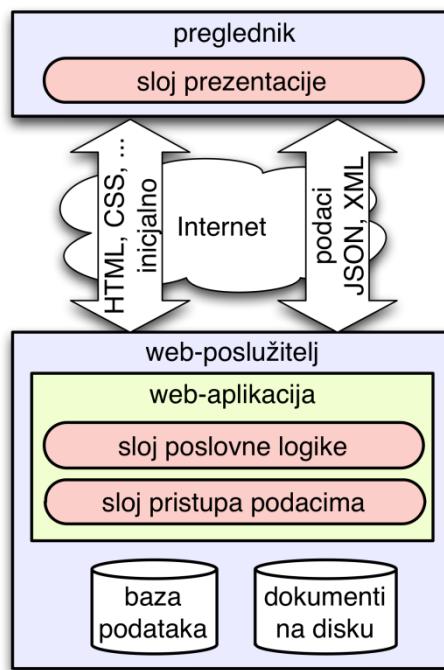
Web-aplikacije imaju višeslojnju arhitekturu (Slika 4.10) gdje svaki sloj implementira definirane funkcionalnosti te je povezan sa slojem iznad ili ispod, slično slojevima u mrežnom složaju. Web-aplikacije su obično organizirane u 3 sloja:

- sloj prezentacije – služi za prikaz informacija (GUI, HTML, klikovi mišem, ...) i za obradu HTTP-zahtjeva;
- sloj poslovne (domenske) logike – obrađuje podatke koje je dobio od sloja prezentacije. To je glavni dio sustava koji radi ono za što je sustav namijenjen;
- sloj pristupa podacima – komunicira s bazom podataka i drugim komunikacijskim sustavima te se brine o transakcijama i o pohrani podataka.



Slika 4.10. Arhitektura web-aplikacije

Zadnjih nekoliko godina pojavljuje se nova arhitektura koja se temelji na bogatom korisničkom sučelju (Slika 4.11). U ovoj arhitekturi se sloj prezentacije nalazi na klijentu. Pokretanje aplikacije se vrši inicijalnim dohvaćanjem web-stranice sa svim popratnim sadržajima (HTML, CSS, JavaScript, ...). Nakon toga korisnik interakcijom sa slojem prezentacije zapravo pokreće dijelove programa u JavaScriptu koji mijenjaju korisničko sučelje i komuniciraju s poslužiteljem pomoću protokola HTTP. Sadržaj odgovora je obično u formatu JSON (*JavaScript Object Notation*) ili XML. Za komunikaciju se koristi AJAX (*Asynchronous JavaScript and XML*).



Slika 4.11. Nova arhitektura web-aplikacija

## 4.7 Skripte na klijentu

Skripte na klijentu su uključene u HTML-ov dokument ili se definiraju u posebnoj datoteci koja je povezana s HTML-ovim dokumentom. Programski jezik za skripte na klijentu je JavaScript. Naziv JavaScript je smislila tvrtka Netscape. Kasnije je Microsoft napravio jezik koji ima vrlo slična svojstva i nazvao ga JScript. Pravo ime tih jezika je ECMAScript i pod tim nazivom je jezik standardiziran kod standardizacijskih tijela ECMA i ISO. Specifikacije jezika su u standardima ECMA-262 i ISO/IEC 16262. Prva inačica standarda je prihvaćena 1998. godine, ali se jezik i dalje razvija i standardizira.

Cilj jezika JavaScript je bio povećati interaktivnost HTML-ovih stranica. Današnje inačice se koriste za: dinamičko stvaranje elemenata u HTML-u, interakciju s klijentima (reagiranje na događaje), provjeru obrazaca, komunikaciju s poslužiteljem pomoću AJAX-a. JavaScript je skriptni jezik koji se interpretira. Osnovna svojstva su: dinamičnost, slaba povezanost, objektna orijentiranost, ima i svojstva funkcionalnog jezika. Bilo tko ga može koristiti bez kupovanja licenci. Bitno je naglasiti da JavaScript nema nikakve veze s programskim jezikom Java. Jedina sličnost je u imenu.

Pogledajmo kako se JavaScript može postaviti u HTML:

```
<html>
<body>

<script type="text/javascript">
document.write("Ovo je napisano iz prvog JavaScripta!");
</script>

</body>
</html>
```

U primjeru se vidi da se JavaScript kôd dodaje u oznaku `script` koja ima atribut `type` i vrijednost `text/javascript`. Ova skripta se izvršava kada se HTML priprema za prikaz u pregledniku. U primjeru se koristi objekt `document` i poziva metoda `write` koja na mjestu skripte dodaje tekst koji je parametar metode. Na taj je način dinamički postavljen tekst u HTML-ov dokument. Izgled učitane stranice je na slici (Slika 4.12).



Slika 4.12. Prvi primjer JavaScripta

U sljedećem primjeru stvaramo novu oznaku `h1` u HTML-u:

```
<html>
<body>

<script type="text/javascript">
document.write("<h1>Ovo je naslov!</h1>");
</script>
```

```
</body>
</html>
```

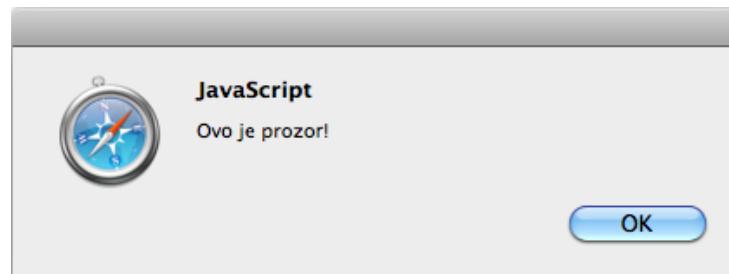
U sljedećem primjeru je napravljena funkcija i poziv funkcije na klik:

```
<html>
<head>
<script type="text/javascript">
function disp_alert()
{
    alert("Ovo je prozor!");
}
</script>
</head>
<body>

<input type="button" onclick="disp_alert()" value="Prikaži prozor" />

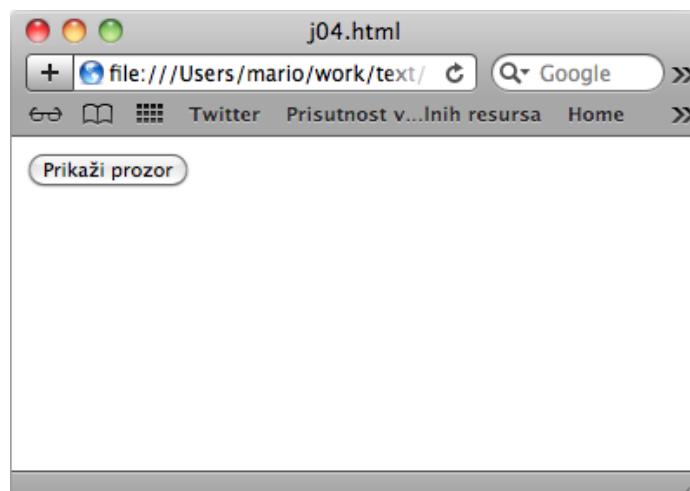
</body>
</html>
```

Funkcija se definira pomoću ključne riječi `function` nakon čega dolazi ime funkcije i parametri. Iza toga se u vitičastim zagradama piše kôd funkcije. U primjeru je definirana funkcija pod imenom `disp_alert` koja poziva funkciju `alert`. Funkcija `alert` je ugrađena funkcija koja prima jedan parametar, a to je tekst koji će se prikazati u novom dijalogu (Slika 4.13).



Slika 4.13. Dijalog u JavaScriptu

U HTML-u imamo oznaku `input` koja ima atribut `onclick`. Vrijednost atributa `onclick` je zapravo kôd koji se poziva prilikom klika mišem na oznaku `input`. U ovom primjeru je to poziv funkcije `disp_alert()`. Izgled stranice se nalazi na slici (Slika 4.14).



Slika 4.14. Obrada događaja

Ako želimo definirati funkciju koja prima parametre onda to izgleda ovako:

```
function myfunction(txt)
{
    alert(txt);
}
```

Ovdje je parametar spremljen u varijablu `txt`. Kao što se vidi u kôdu prije korištenja varijable nije ju potrebno deklarirati, već ju jednostavno koristimo.

Pogledajmo primjer obrade greške:

```
<html>
<head>
<script type="text/javascript">
txt="";
function message() {
    try {
        adddlert("Pozdrav!");
    } catch(err) {
        txt="Dogodila se greška.\n\n";
        txt+="Opis: " + err.description + "\n\n";
        txt+="Kliknite OK za nastavak.\n\n";
        alert(txt);
    }
}
</script>
</head>

<body>
<input type="button" value="Poruka" onclick="message()" />
</body>

</html>
```

Greška se obrađuje tako da se definira blok `try` u kojem se može dogoditi greška, a ako se dogodi onda se u bloku `catch` ta greška hvata i obrađuje. U navedenom primjeru se poziva funkcija `adddlert` koja ne postoji pa se zato tu događa greška koja se hvata i ispisuje se dijalog s parametrima greške.

U JavaScriptu se mogu provjeriti i podaci o pregledniku u kojem se izvršava taj kôd (Slika 4.15):

```
<html>
<body>
<script type="text/javascript">
document.write("<p>Preglednik: ");
document.write(navigator.appName + "</p>");

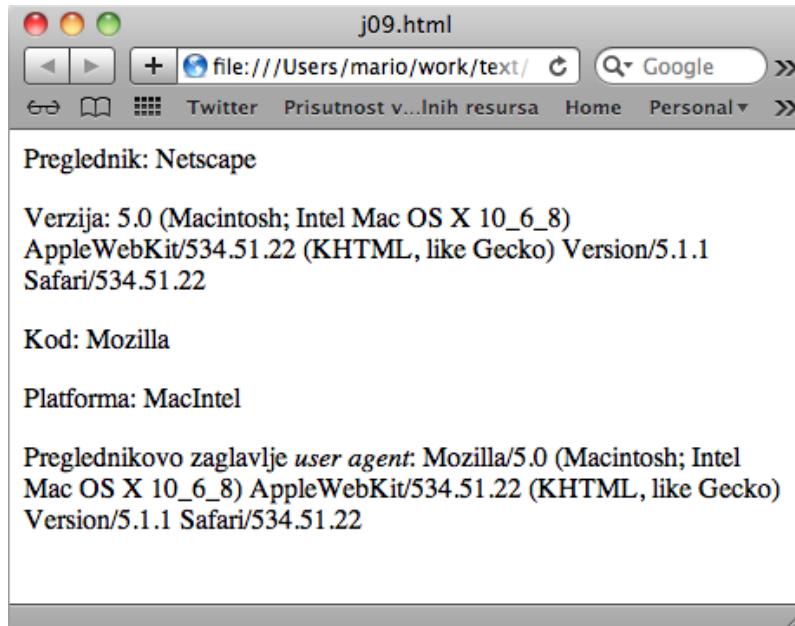
document.write("<p>Verzija: ");
document.write(navigator.appVersion + "</p>");

document.write("<p>Kod: ");
document.write(navigator.appCodeName + "</p>");

document.write("<p>Platforma: ");
document.write(navigator.platform + "</p>");

document.write("<p>Preglednikovo zaglavlje <i>user agent</i>: ");
document.write(navigator.userAgent + "</p>");
</script>
</body>
</html>
```

Ovdje vidimo kako se može provjeriti naziv preglednika (`navigator.appName`), inačica (`navigator.appVersion`), kodno ime preglednika (`navigator.appCodeName`), platforma na kojoj se izvršava (`navigator.platform`) i zaglavlje o pregledniku koje se šalje protokolom HTTP (`navigator.userAgent`).



Slika 4.15. Ispis podataka o pregledniku

Sljedeći primjer pokazuje kako dohvatiti HTML-ove oznake i u njih dodati tekst ili druge oznake:

```

<html>
<head>
<title>Neki naslov</title>
</head>

<body>
<script type="text/javascript">
document.write("Naslov:" + document.title + "<br>");

function addDiv() {
    x = document.getElementById("mojId");
    x.innerHTML = x.innerHTML + "*";
}

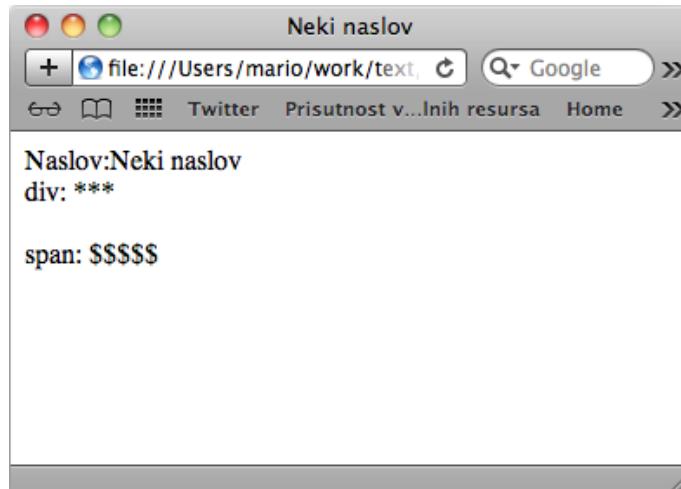
function addSpan() {
    x = document.getElementsByName("imeOznake");
    x[0].innerHTML = x[0].innerHTML + "$";
}
</script>

<div id="mojId" onclick="addDiv()">div: </div><br>
<span name="imeOznake" onclick="addSpan()">span: </span>
</body>

</html>
  
```

Ovdje imamo definiranu jednu oznaku `div` koja ima atribut `id`. Kada se klikne na tu oznaku poziva se funkcija `addDiv` koja pomoću `document.getElementById` dohvati referencu na oznaku i pomoću svojstva `innerHTML` promijeni sadržaj oznake. U ovom slučaju dodaje se znak „\*“.

U istom primjeru imamo oznaku `span` koja ima atribut `name`. Kada se klikne na tu oznaku onda se pozove funkcija `addSpan`. Ta funkcija dohvaća oznake prema imenu koristeći funkciju `document.getElementsByName`. Pošto više oznaka može imati isto ime, ova funkcija vraća polje u kojemu su sve oznake s tim imenom. U ovom primjeru je to samo jedna oznaka. Na isti način kao i kod oznake `div` dodajemo znak „\$“ sadržaju oznake. Izgled primjera u pregledniku je na slici (Slika 4.16).

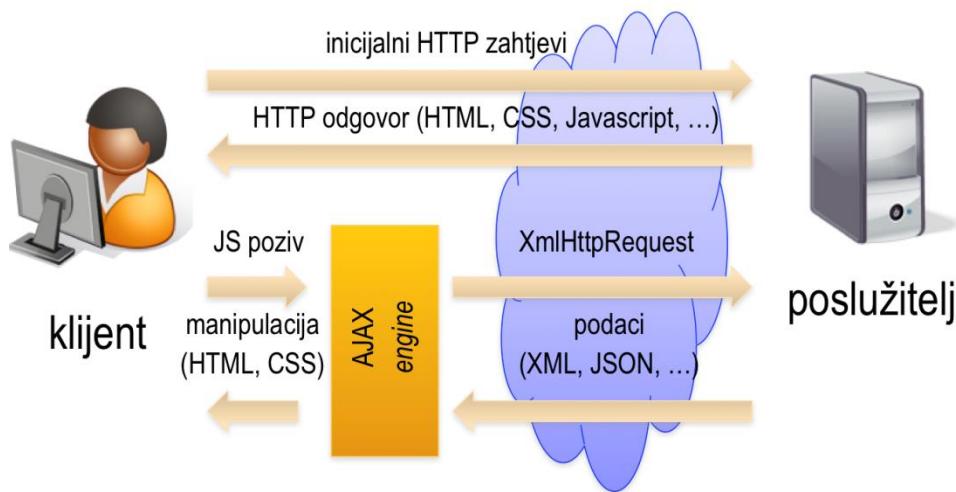


Slika 4.16. Primjer dohvaćanja oznaka

U praksi se događa da se jedna skripta ispravno izvodi u jednom pregledniku, a u drugom ne. U tom slučaju je potrebno prije izvršavanje skripte detektirati u kojem se pregledniku skripta izvršava i na osnovu toga pripremiti različiti kôda za različite preglednike. Za pojednostavljenje ove zadaće se koriste knjižnice skript koje programeri mogu koristiti na jednak način u bilo kojem pregledniku. Najpoznatije knjižnice su sljedeće: jQuery, The Yahoo! User Interface Library (YUI), script.aculo.us, Prototype, Moo Tools, Dojo Toolkit, Ext JS, MochiKit i Spry Framework.

#### 4.8 AJAX

AJAX (*Asynchronous JavaScript and XML*) je tehnika koja se koristi za dinamičko dohvaćanje podataka sa poslužitelja. Kod klasičnih web-stranica prilikom interakcije s korisnikom (npr. klik) se na poslužitelj šalje HTTP-ov zahtjev za čitavom stranicom. AJAX omogućuje da se web-stranice mogu ažurirati asinkrono razmjenjujući podatke s poslužiteljem u pozadini tj. da se u pozadini pošalje zahtjev na poslužitelj i u odgovoru dobiju samo podaci koji se pomoću JavaScripta ugrađuju u trenutnu stranicu. Primjer web-aplikacija koje koriste AJAX su: Google Maps, Gmail, Youtube, Facebook itd.



Slika 4.17. Komunikacija pomoću AJAX-a

Na slici (Slika 4.17) se nalazi primjer koji ilustrira komunikaciju pomoću AJAX-a. Kada klijent upiše u preglednik URL, preglednik pomoću HTTP-a šalje inicijalni zahtjev za stranicom te prima HTML-ov dokument koji se počne interpretirati i prikazivati u prozoru preglednika. Kako pregledniku trebaju za prikaz te stranice još i slike, CSS i JavaScript onda se šalju dodatni zahtjevi i primaju odgovori na te

zahtjeve. Nakon što su svi elementi dohvaćeni i prikazani korisnik može interakcijom (npr. klik mišem na neki element) pokrenuti pozivanje funkcije u JavaScriptu. Ako ta funkcija treba podatke s poslužitelja onda ona koristi objekte za AJAX (na slici je to *AJAX engine*) koji šalju HTTP-ov zahtjev na poslužitelj, a nakon slanja zahtjeva nastavlja se izvršavanje skripte. Kada poslužitelj obradi zahtjev onda šalje odgovor kojega u pregledniku prima *AJAX engine*. U *AJAX engineu* je za svaki zahtjev registrirana funkcija u JavaScriptu koju treba obavijestiti kada je pristigao dio odgovora. Ta funkcija u svakom pozivu provjerava podatke koji su stigli i po potrebi podatke ugrađuje u web-stranicu manipulacijom oznaka u HTML-u.

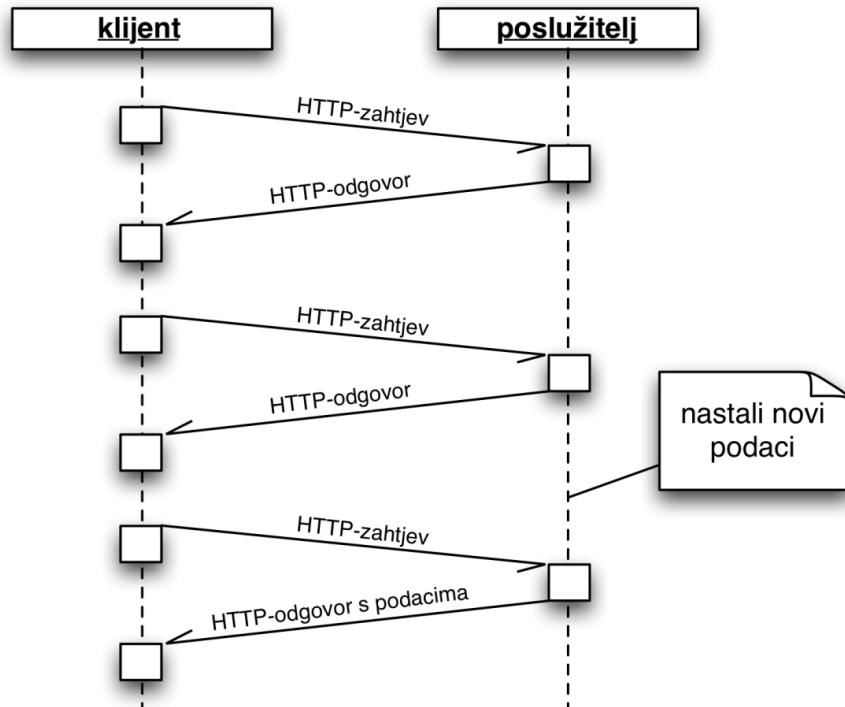
Kako su pravila korištenja AJAX-a u preglednicima različita, potrebna je provjera vrste preglednika i izvršavanje kôda prilagođenog tom pregledniku. Druga mogućnost je korištenje gotovih knjižnica iz prethodnog poglavlja, a postoje i specijalizirane knjižnice baš za AJAX kao što je DWR (*Direct Web Remoting*).

Google je izgradio još jedan koncept izrade web-aplikacija s bogatim korisničkim sučeljem, a to je Google Web Toolkit u kojem se aplikacija implementira u programskom jeziku Java, a prevoditelj prevodi taj kôd iz Java u JavaScript, CSS i HTML. U ovom slučaju se programer ne mora brinuti o vrstama preglednika, a aplikacija izgleda kao i „obična“ aplikacija na računalu.

Slanje događaja klijentu objasnit ćemo u nastavku na primjeru web-aplikaciju za čavrjanje (engl. *chat*). Klijent u pregledniku šalje poruku drugom klijentu koji koristi drugi preglednik. Jedini način za realizaciju takve aplikacije je korištenjem AJAX-a koji šalje i prima poruke preko web-poslužitelja. Problem koji se ovdje javlja je taj što poslužitelj ne može poslati zahtjev klijentu, već klijent mora slati zahtjev poslužitelju, a on mu samo šalje odgovor.

Taj problem se rješava na dva načina:

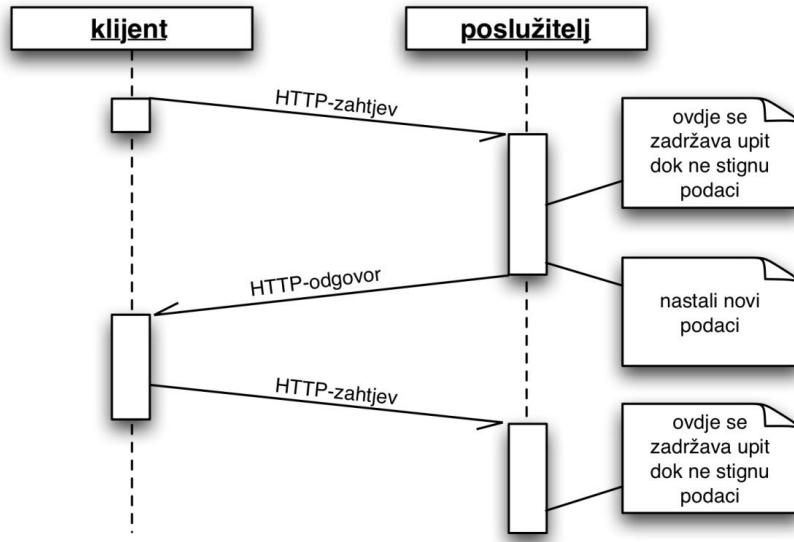
- prozivanje poslužitelja (engl. *poll*) ili
- dugo prozivanje poslužitelja (engl. *long poll*).



Slika 4.18. Prozivanje poslužitelja

Kod prozivanja poslužitelja (Slika 4.18) klijent periodički šalje zahtjeve na poslužitelj i prima odgovore. U slučaju da nema nikakvog događaja tj. poruka za njega, prima prazan odgovor. Kada na poslužitelj

stigne poruka od drugog klijenta onda prilikom sljedećeg upita prvi klijent u odgovoru prima i poruku. Nedostatak ovog pristupa je što u slučajevima kada nema puno poruka klijent opterećeće mrežu i poslužitelj nepotrebnim zahtjevima. Ako se vrijeme periodičkog slanja upita poveća onda se gubi na interaktivnosti. Zbog toga je bolje koristiti dugo prozivanje poslužitelja ilustrirano slikom (Slika 4.19).



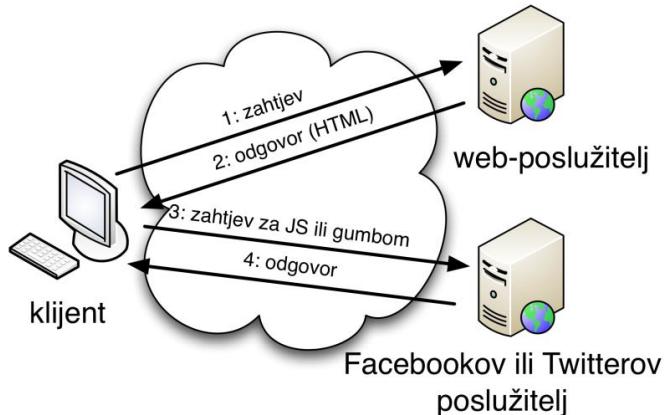
Slika 4.19. Dugo prozivanje poslužitelja

U ovom slučaju klijent šalje upit na poslužitelj, ali mu poslužitelj ne odgovara nego čeka, događaj, tj. poruku drugog korisnika, kada poslužitelj šalje odgovor. Kada klijent primi odgovor odmah šalje novi upit koji poslužitelj zadržava dok ne se ne dogodi novi događaj.

Ovaj pristup je bolji od običnog prozivanja poslužitelja, ali i on ima nedostatke. Klijent stalno mora imati otvorenu konekciju prema poslužitelju što u starijim preglednicima stvara probleme ako korisnik ima otvoreno više kartica sa stranicama jer preglednici imaju ograničenje na broj dretvi. Drugi problem se javlja na poslužitelju jer poslužitelj mora imati otvoren velik broj konekcija prema klijentima što zauzima resurse.

#### 4.9 Integracija s drugim sjedištem

Kada na neku web-stranicu želimo postaviti gumb Facebooka ili Twittera ili neke druge društvene mreže onda trebamo u tu web-stranicu staviti poveznicu na JavaScript kod koji se učitava s web-sjedišta te društvene mreže (Slika 4.20).



Slika 4.20. Povezivanje s drugim web-sjedištem

To znači da će se u pregledniku, koji ima učitanu takvu web-stranicu, izvršavati kod JavaScripta koji kontroliraju programeri drugog web-sjedišta. To može predstavljati potencijalni sigurnosni problem. Na taj način Facebook može saznati na kojim stranicama je neki korisnik bio (isto tako i Google ako koristimo Google Analytics).

#### 4.10 Pitanja za učenje i ponavljanje

- 4.1. Navedite i objasnite najčešće korištene metode protokola HTTP/1.1.
- 4.2. Objasnite namjenu jezika CSS. Zašto se pojavila potreba za definiranjem takvog jezika?
- 4.3. Korisnik nakon ispunjavanja obrasca na Web-u odabire opciju *Submit*, čime pošalje podatke Web-poslužitelju na adresu [www.tel.fer.hr/obrazac/accept](http://www.tel.fer.hr/obrazac/accept) korištenjem protokola HTTP verzije 1.1. Kojim se HTTP zahtjevom šalju podaci poslužitelju i kako je definiran prvi redak zahtjeva?
- 4.4. Objasnite opći format poruka protokola HTTP. Navedite kako glasi potpun i apsolutan URI koji identificira resurs zatražen u zahtjevu, ako prva 2 retka HTTP zahtjeva sadrže sljedeće podatke:

```
GET /predmet/rassus HTTP/1.1  
Host: www.fer.unizg.hr
```

- 4.5. Objasnite ulogu posredničkog poslužitelja s priručnim spremištem koji se nalazi u mreži između web-preglednika i web-poslužitelja. Gdje se sve može nalaziti u mreži?
- 4.6. Navedite i objasnite prednosti korištenja posredničkog poslužitelja s priručnim spremištem za korisnika, izvorni poslužitelj i komunikacijsku mrežu.
- 4.7. Pretpostavite da se sjedište weba sastoji od 2 poslužitelja priključena na Internet preko posrednika (*proxy*). Navedite i objasnite svojstva ovog raspodijeljenog sustava.
- 4.8. Objasnite razliku između web-aplikacija temeljenih na CGI (Common Gateway Interface) i poslužiteljskim skriptama.
- 4.9. Skicirajte i objasnite slojevitu arhitekturu web-aplikacija.
- 4.10. Koja je prednost korištenja tehnike dugog prozivanja poslužitelja za web-aplikacije?

## 5 WEB-USLUGE

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

„Obični“ Web tipično koriste ljudi za pribavljanje informacija dok su web-usluge namijenjene za programe. Tehnologija *Web Services* (WS) omogućuje programima lakše korištenje Weba umjesto RPC-a/Java RMI-a, CORBA-e, DCOM-a, itd. Osnovna svojstva web-usluga su:

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

Postoji puno definicija web-usluga. Sljedeće dvije najbolje opisuju što su to web-usluge.

Prema konzorciju W3C: „Web-usluga je aplikacija identificirana URI-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”.

Prema IBM-ovom vodiču<sup>18</sup>: „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. ...“

Dakle možemo reći da je web-usluga program koji:

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

Web-usluge su definirane i standardima. U osnovnu skupinu standarda spadaju:

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

Osnovni standardi koji se koriste za web-uslugu su SOAP i WSDL. UDDI nije zaživio toliku popularnost u praksi kao prva dva standarda, ali se UDDI često koristi unutar jedne tvrtke.

Druga generacija standarda web-usluga nadovezuje se na osnovne standarde. Druga generacija standarda omogućuje korištenje web-usluga u poslovnom okruženju koje mora biti sigurno, zaštićeno i fleksibilno. U drugoj generaciji standarda web-usluga (WS-\*) popravljaju se nedostaci koji su uočeni u osnovnim standardima:

- WS-Coordination – protokol za koordinaciju raspodijeljenih aplikacija,
- WS-Transaction – podrška za transakcije,

---

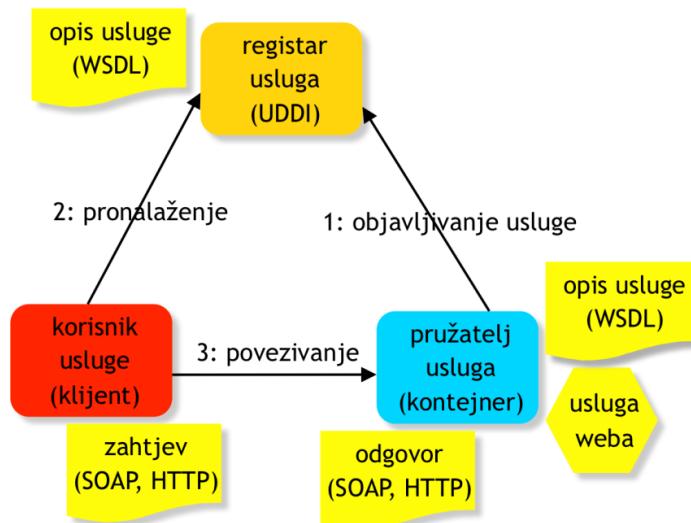
<sup>18</sup>Definicija iz [www.xml.com](http://www.xml.com)

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

Arhitektura web-usluge se sastoji od 3 entiteta:

- pružatelja usluge,
- registra usluga i
- korisnika usluge.

Slika 5.1 prikazuje interakciju između navedena tri entiteta web-usluge. Kada se na pružatelja usluge instalira web-usluga, potrebno ju je prvo registrirati u registru usluga. Pomoću protokola UDDI pružatelj usluge objavi uslugu i registru šalje opis usluge u WSDL-u. Kada klijent želi pronaći uslugu prvo registru pošalje upit za pronađenje, a registar mu odgovara opisom usluge koja odgovara zadanim kriterijima. Nakon toga klijent se može povezati s pružateljem usluge jer ima opis usluge. Klijent šalje pomoću protokola HTTP i SOAP zahtjev za pozivom usluge, pružatelj usluge pokreće izvođenje usluge i šalje natrag rezultat u odgovoru isto tako pomoću HTTP-a u SOAP-a.



Slika 5.1. Arhitektura web-usluge

Tablica 4. Usporedba modela klijent-poslužitelj i web-usluge

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

Web-usluge u načelu slijede klasični model komunikacije klijent-poslužitelj i RPC te ih stoga tablica (Tablica 4) uspoređuje.

Isto tako možemo usporediti i web-aplikacije s web-uslugama (Tablica 5).

Tablica 5. Usporedba web-aplikacija i web-usluga

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

Postoje tri vrste web-usluga:

- usluge temeljene na pozivu udaljene procedure (RPC),
- usluge temeljene na dokumentima/porukama i
- usluge temeljene na prijenosu prikaza stanja resursa.

### 5.1.1 Poziv udaljene procedure

Web-usluge temeljene na pozivu udaljene procedure koriste poznato načelo poziva udaljene procedure (RPC) koju čini operacija deklarirana i opisana WSDL-om.

Prve web-usluge su bile temeljene na mehanizmu RPC tako da je danas taj mehanizam jako dobro podržan. Taj mehanizam je kritiziran jer usko povezuje klijenta i uslugu koju se poziva. Usluga je u tom slučaju usko povezana za implementacijski jezik što predstavlja problem kod implementacije klijenta u jeziku različitom od jezika u kojem je implementirana usluga. Mnogi proizvođači su zaključili da je to loše i da se tu ništa ne može popraviti pa je mehanizam RPC za web-usluge dodan u WS-I Basic Profile i ne razvija se više.

### 5.1.2 Usluge temeljene na porukama/dokumentima

Druga vrsta su web-usluge gdje je osnovna jedinica komunikacije poruka, a ne operacija kao kod RPC-a. Taj mehanizam je temelj uslužno orijentirane arhitekture (SOA - *Service Oriented Architecture*). Ovu vrstu usluga podržava većina proizvođača programske potpore i industrijskih analitičara. Za razliku od RPC-a, usluge temeljene na porukama ne povezuju samo klijent i poslužitelj i njihove implementacijske jezike jer je fokus na „ugovorima“ koji su propisani WSDL-om (što se prenosi i na koji način).

### 5.1.3 Usluge temeljene na prijenosu prikaza stanja resursa

Ova vrsta usluga koristi HTTP i slične protokole tako da je sučelje poznato i definirano (npr. metode GET, PUT, POST i DELETE protokola HTTP). Ovdje je fokus na interakciji s resursima koji imaju stanje, a ne na porukama ili procedurama. Ova vrsta usluga može koristiti WSDL i SOAP poručivanje preko protokola HTTP (npr. WS-Transfer). WSDL inačice 2.0 omogućuje povezivanje zahtjeva HTTP-a i tako podržava bolju implementaciju ovih vrsta usluga. Problem je što je podrška za WSDL 2.0 trenutno slaba u razvojnim alatima. Zbog toga danas ova vrsta usluga najčešće koristi samo protokol HTTP bez WSDL-a i SOAP-a.

## 5.2 Protokol SOAP

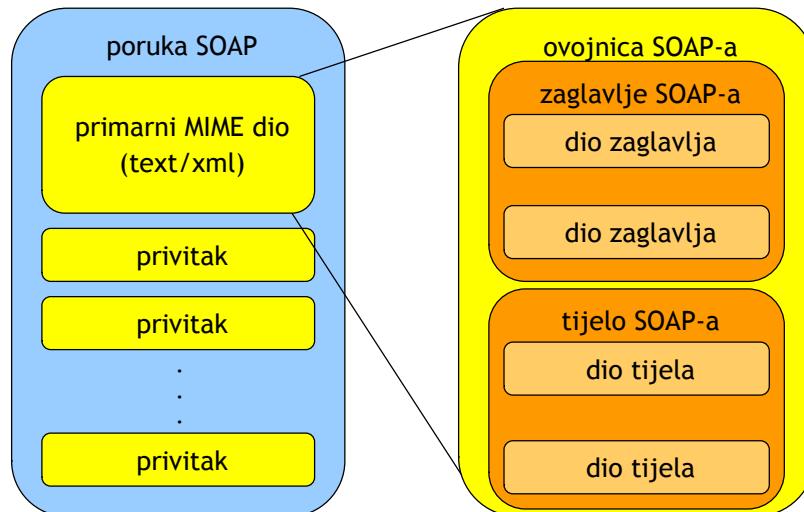
Protokol SOAP<sup>19</sup> (*Simple Object Access Protocol*) omogućuje komunikaciju s web-uslugom. Postoje dva osnovna načina rada:

---

<sup>19</sup>SOAP Version 1.2, <http://www.w3.org/TR/soap/>

- poziv udaljenih procedura (RPC) – slično kao CORBA, DCOM, Java RMI. Služi za prijenos serijaliziranih parametara i rezultata. Posljedice su:
  - dobro definirana sučelja i tipovi podataka i
  - prilagodni kôd može biti generiran automatski;
- razmjena dokumenata/poruka – koriste se XML-ovi dokumenti u razmjeni poruka. Ovaj način je fleksibilniji u odnosu na RPC. XSLT i XQuery se koriste za prilagodbu dokumenata. Lakše se koriste uzorci razmjene poruka.

Struktura formata poruke SOAP je prikazana na slici (Slika 5.2).



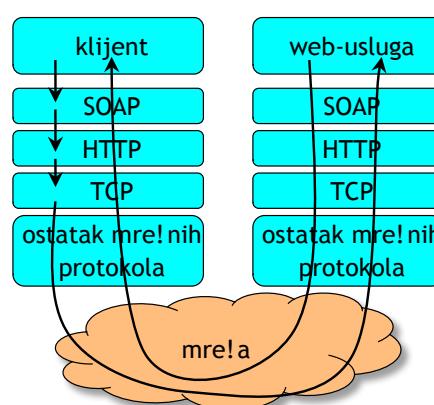
Slika 5.2. Struktura formata poruke u SOAP-u

Poruka se sastoji od primarnog dijela (MIME – text/xml) koji je osnovni dio svake poruke. Nakon toga slijede različiti privici. Taj primarni dio se sastoji od ovojnica SOAP-a unutar koje su dva dijela: zaglavlj SOAP-a i tijela SOAP-a. Primjer jedne takve poruke je u nastavku.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope">
<Header>
<!-- sadržaj zaglavlj poruke SOAP --&gt;
&lt;/Header&gt;
&lt;Body&gt;
<!-- sadržaj poruke SOAP --&gt;
&lt;/Body&gt;
&lt;/Envelope&gt;</pre>

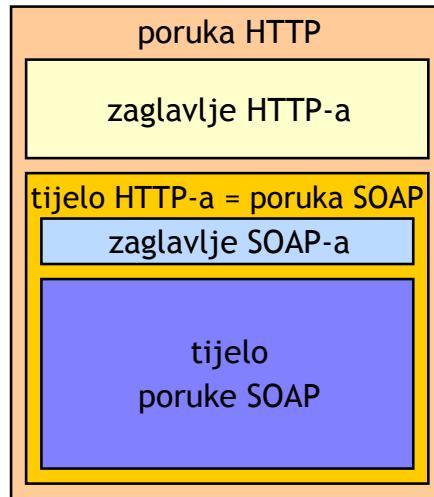
```

Poruka se prilikom komunikacije klijenta i poslužitelja web-usluge prenosi preko mrežnih slojeva (Slika 5.3).



Slika 5.3. Prijenos poruke SOAP kroz slojeve

Svaka poruka se šalje sloju ispod i postavlja se kao sadržaj protokola nižeg sloja što je prikazano slikom (Slika 5.4).



Slika 5.4. Učahurivanje poruke SOAP u druge protokole

U nastavku slijedi primjer stvarne poruke zahtjeva.

```

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:add xmlns:ns2="http://fromjava.ws.tel.fer.hr/">
<a>5</a>
<b>4</b>
</ns2:add>
</S:Body>
</S:Envelope>
  
```

Ovdje vidimo da je čitava poruka XML-ov dokument koji ima osnovnu oznaku `Envelope`. Unutar nje se nalazi oznaka `Body`, ali nema oznake `Header` jer u ovom slučaju nije potrebna. Unutar oznake `Body` imamo oznaku `add` koja se nalazi u drugom prostoru imena (`ns2`). Oznaka `add` označava operaciju web-usluge koja će se izvršiti, a unutar te oznake se nalaze parametri. Ovdje imamo dva parametra označena oznakama `a` i `b` čiji je sadržaj tekstualni element koji se može pretvoriti u broj. Kada poslužitelj usluge primi ovakvu poruku prvo ju treba parsirati i pozvati pravi dio usluge s parametrima. Kada se usluga izvrši generira se rezultat koji je isto poruka SOAP:

```

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:addResponse xmlns:ns2="http://fromjava.ws.tel.fer.hr/">
<return>9</return>
</ns2:addResponse>
</S:Body>
</S:Envelope>
  
```

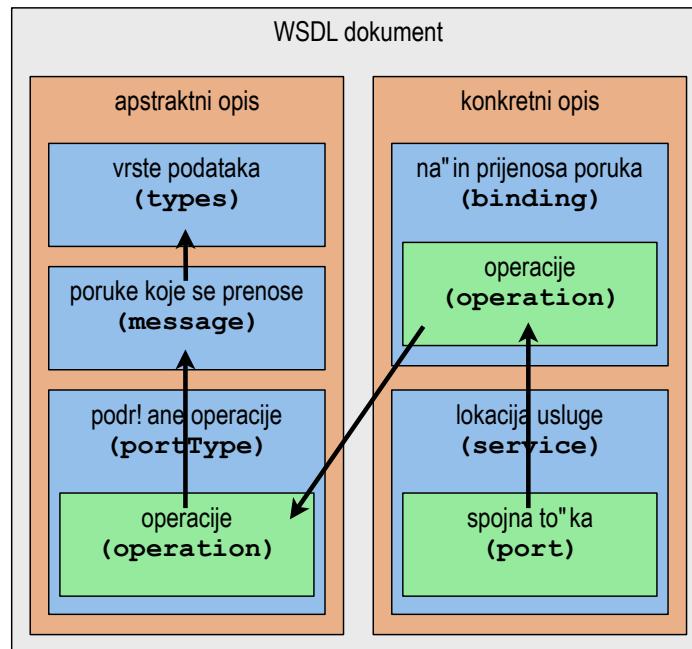
Iz primjera se vidi da je odgovor vrlo sličan zahtjevu. Razlikuju se u sadržaju oznake `Body`. U odgovoru imamo oznaku `addResponse` unutar koje je oznaka `return` koja označava rezultat izvođenja operacije web-usluge. Sadržaj je tekstualni element koji se može pretvoriti u broj.

### 5.3 Jezik WSDL

WSDL (*Web Services Description Language*) je jezik za opis web-usluga koji koristi XML. W3C konzorcij je zadužen za standardizaciju WSDL-a te danas postoje dvije inačice 1.1<sup>20</sup> i 2.0<sup>21</sup>. WSDL se osim za opis web-usluge koristi i za lociranje web-usluga u kombinaciji s UDDI-om.

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

- apstraktog opisa i
- konkretnog opisa.



Slika 5.5. Struktura WSDL-a

#### 5.3.1 Apstraktni opis

Interakcija između klijenta i poslužitelja se sastoji od razmjene niza poruka. Poslužitelj prihvata dolaznu poruku i može vratiti odgovor odlaznom porukom ili može baciti iznimku. Svaka konkretna poruka pripada nekoj vrsti poruke, a svaka vrsta poruke se sastoji od niza podataka gdje svaki podatak mora imati definiranu vrstu. Kada imamo više procedura/operacija koje se mogu pozvati onda je moguće da takve operacije koriste istu vrstu poruka. Za opis svih tih informacija trebamo:

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

Navedeni opisi trebaju biti platformski i jezično neovisni tj. ne smiju ovisiti o implementaciji. Tako se omogućuje korištenje usluga različitim klijentima. Apstraktan opis u WSDL-ovom dokumentu se sastoji od 4 važna elementa u XML-u:

1. **types**: definira vrste podataka neovisne o platformi i jeziku (koristi se XML Schema),
2. **message**: definiraju ulazne i izlazne poruke koje se mogu koristiti kao parametri usluge,

<sup>20</sup>Web Services Description Language (WSDL) 1.1, 2001., <http://www.w3.org/TR/wsdl>

<sup>21</sup>Web Services Description Language (WSDL) Version 2.0, 2007., <http://www.w3.org/TR/wsdl20/>

3. **operation:** predstavlja jednu operaciju/metodu/proceduru koja je definirana u usluzi, a sastoji se od definicija ulaznih, izlaznih i iznimnih poruka koje se mogu razmjenjivati korištenjem ove operacije,
4. **portType:** koristi poruke (pod 2) kako bi opisao sve operacije koje pruža usluga.

### 5.3.2 Konkretni opis

Svaka web-usluga mora imati definiran URI (jedinstveni identifikator) pomoću koje se pristupa definiranoj usluzi. Web-usluga ima definiran protokol i format tako da usluga može prihvati i obraditi ulazne poruke i na odgovarajući način odgovoriti. Te definicije su specifične za implementaciju usluge i definirane su u konkretnom opisu u WSDL-ovom dokumentu. Konkretan opis se sastoji od 2 dijela:

1. **binding:** definira kako je konkretna implementacija povezana s operacijama u apstraktnom opisu i definira format u kojem će se poruke prenositi (protokol i elemente)
2. **service:** definira URI na kojem je usluga isporučena tj. na kojoj adresi se može pozvati usluga (taj URI je definiran u spojnoj točki – oznaci **port**).

### 5.3.3 Elementi WSDL-a

Slijedi jedan primjer opisa usluge:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="myWSDL"
  targetNamespace="http://j2ee.netbeans.org/wsdl/myWSDL"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://j2ee.netbeans.org/wsdl/myWSDL"
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<types/>
<message name="myOperationRequest">
<part name="part1" type="xsd:string"/>
</message>
<message name="myOperationResponse">
<part name="result" type="xsd:int"/>
</message>
<portType name="myPortType">
<operation name="myOperation">
<input name="input1" message="tns:myOperationRequest"/>
<output name="output1" message="tns:myOperationResponse"/>
</operation>
</portType>
<binding name="myBinding" type="tns:myPortType">
<soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="myOperation">
<soap:operation/>
<input name="input1">
<soap:body use="literal"
  namespace="http://j2ee.netbeans.org/wsdl/myWSDL"/>
</input>
<output name="output1">
<soap:body use="literal"
  namespace="http://j2ee.netbeans.org/wsdl/myWSDL"/>
</output>
</operation>
</binding>
<service name="myService">
<port name="myPort" binding="tns:myBinding">
<soap:address
  location="http://ws.tel.fer.hr:8080/myService/myPort"/>
</port>
```

```
</service>
<plnk:partnerLinkType name="myWSDL">
<plnk:role name="myPortTypeRole" portType="tns:myPortType"/>
</plnk:partnerLinkType>
</definitions>
```

Osnovna oznaka (korijenski čvor) WSDL-ovog dokumenta je `definitions`. Iza toga slijedi oznaka `types` koja je u ovom slučaju prazna. Oznaka `types` se koristi za opis podataka pomoću XML Scheme. Ako se koriste osnovni podaci iz Scheme onda se stavlja prazna oznaka kao što je ovdje slučaj. Sljedeći dio su oznake `message`.

```
<message name="myOperationRequest">
<part name="part1" type="xsd:string"/>
</message>
```

One opisuju jednosmjerne poruke tako da definiraju ime poruke atributom `name`. Poruka se sastoji od dijelova, oznaka `part` koja također ima svoje ime, ali i vrstu podatka (atribut `type`). U ovom slučaju je to `xsd:string` što označava vrstu `string` iz XML Scheme. Svaki dio se referencira na vrste podataka iz dijela `types` ako smo ih tamo definirali.

Sljedeći dio je `portType` koji definira operacije, primjerice:

```
<portType name="myPortType">
<operation name="myOperation">
<input name="input1" message="tns:myOperationRequest"/>
<output name="output1" message="tns:myOperationResponse"/>
</operation>
</portType>
```

Svaka operacija sastoji se od poruka (reference na poruke iz dijela `message`). Poruke mogu biti:

- parametri (ulazne poruke) ili
- vrijednosti koje se vraćaju (rezultati).

U ovom primjeru imamo definiranu operaciju `myOperation` u oznaci `operation`. Ona se sastoji od dvije poruke. Jedna je ulazna (za uslugu) i označena je oznakom `input`, a atributom `message` se povezuje s porukom koja se zove `tns:myOperationRequest`. Druga poruka je rezultat izvođenja usluge pa se označava oznakom `output`. Ona također ima atribut `message` koji ju povezuje s definicijom poruke.

Nakon toga dolazi konkretni dio opisa web-usluge koji započinje oznakom `binding`:

```
<binding name="myBinding" type="tns:myPortType">
<soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="myOperation">
<soap:operation/>
<input name="input1">
<soap:body use="literal"
    namespace="http://j2ee.netbeans.org/wsdl/myWSDL"/>
</input>
<output name="output1">
<soap:body use="literal"
    namespace="http://j2ee.netbeans.org/wsdl/myWSDL"/>
</output>
</operation>
</binding>
```

On definira kako će se poruke iz operacije prenosi, tj. definira transportne protokole (HTTP, SOAP, SMTP) i stil. Stil definira vrstu čitave poruke koja može biti:

- `rpc` – zahtjev će imati omotač u kojem će pisati naziv funkcije koja se poziva ili
- `document` – zahtjev i odgovori će imati „obične“ XML dokumente.

Poruke mogu koristiti dvije vrste pakiranja poruka:

- **literal** – onako kako definira Schema ili
- **encoded** – kodirano pravilima u SOAP-u.

U primjeru imamo oznaku **binding** koja se atributom **type** povezuje s dijelom u **portType**-u. Unutar nje imamo oznaku **soap:binding** koja ima dva atributa. Atribut **style** definira stil poruke koji je u ovom slučaju **rpc**, a atribut **transport** definira kako će se poruka prenosi mrežom. U ovom slučaju je to protokol HTTP unutar kojeg će biti SOAP.

Unutar oznake **binding** nalazi se i oznaka **operation** koja mora biti ekvivalentna oznaci **operation** iz apstraktnog dijela. Unutar nje imamo oznaku **soap:operation** koja označava da je to operacija iz SOAP-a. Nakon toga dolaze oznake **input** i **output** koje odgovaraju takvim oznakama iz apstraktnog dijela. U njima imamo oznake **soap:body** koje označavaju kako će se te poruke pakirati. Pakiranje je definirano atributom **use**.

Nakon toga dolazi oznaka **services** koja definira lokaciju usluge tj. URL:

```
<service name="myService">
<port name="myPort" binding="tns:myBinding">
<soap:address
    location="http://ws.tel.fer.hr:8080/myService/myPort"/>
</port>
</service>
```

Oznaka **service** ima u sebi oznaku **port** koja je atributom **binding** povezana s dijelom definiranim u oznaci **binding**. Unutar oznake **port** se nalazi oznaka **soap:address** čiji atribut **location** definira URL na kojem se usluga nalazi. U ovom slučaju je to <http://ws.tel.fer.hr:8080/myService/myPort>.

Osim ovih oznaka možemo imati i oznaku **import** koja se koristi za uključivanje drugih dokumenata WSDL ili XML Schema. Naime, ako imamo iste vrste podataka u različitim web-uslugama onda dio koji bi dolazio u oznaci **types** možemo staviti u posebnu datoteku i uključiti ih pomoću **import**. Isto tako možemo i dijelove WSDL-a koji se koriste u različitim uslugama (npr. opisi poruka) uključiti pomoću oznake **import**.

## 5.4 UDDI

UDDI (Universal Description, Discovery, and Integration) je protokol koji definira na koji način se mogu objaviti i otkriti web-usluge. Specificira ga OASIS<sup>22</sup>. Registar je poslužitelj koji zna komunicirati protokolom UDDI i čija je svrha čuvanje podataka i metapodataka o web-uslugama. Obično se UDDI register koristi unutar neke organizacije. Register se sastoji od 3 dijela:

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

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

## 5.5 Web-usluge temeljene na pozivu udaljene procedure

Web-usluge koje su temeljene na pozivu udaljene procedure (RPC) su ovisne o alatima i programskom jeziku koji se koristi pri implementaciji web-usluge jer se za svaki programski jezik koristi alat koji automatski iz programskog jezika na svoj način definira elemente WSDL-a. U nastavku

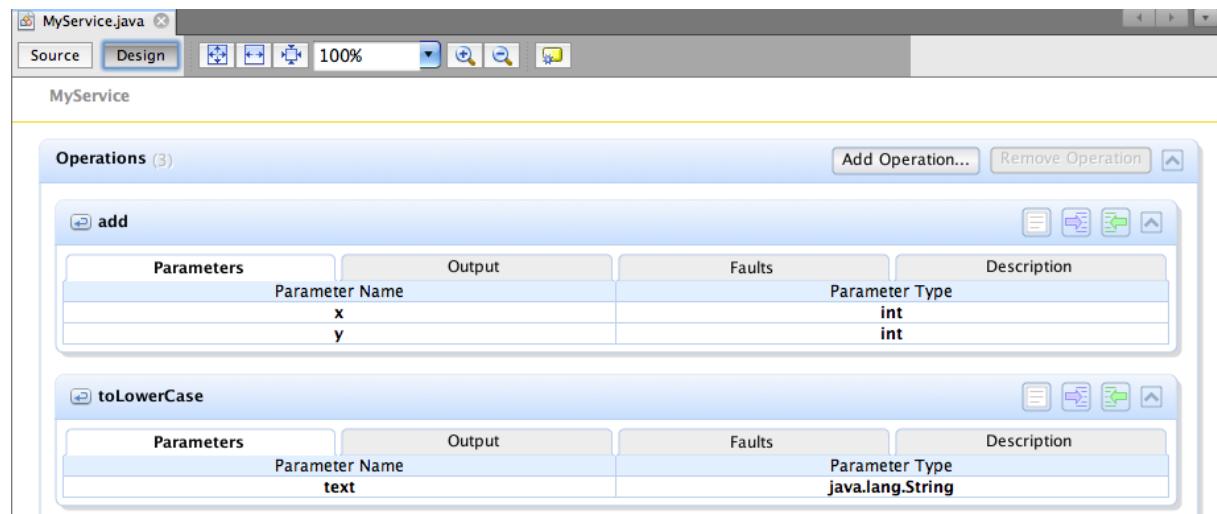
---

<sup>22</sup>UDDI Spec TC, Version 3.0.2, 2004., <http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm>

je pokazano kako se takve web-usluge mogu napraviti u programskom jeziku Java pomoću alata NetBeans<sup>23</sup> inačice 6 ili više i poslužitelja GlassFish<sup>24</sup> koji dolazi s tim alatom.

Detaljne upute možete naći u vodiču<sup>25</sup>. Usluga ima dvije jednostavne operacije. Prva operacija je zbrajanje dva broja, a druga usluga prima tekst u kojem se velika slova pretvaraju u mala.

Prvo je potrebno napraviti projekt web-aplikacije i definirati poslužitelj na kojem će ta web-aplikacija biti postavljena. Nakon toga se u projektu kreira web-usluga te se u pogledu Design dodaju operacije i parametri (Slika 5.6).



Slika 5.6. Pregleda operacija u pogledu dizajna

Nakon toga alat je generirao sljedeći kôd:

```
@WebService()
public class MyService {
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "a") int a,
                   @WebParam(name = "b") int b) {
        // TODO ovdje dodati implementaciju
    }

    @WebMethod(operationName = "toLowerCase")
    public String toLowerCase(
        @WebParam(name = "text") String text) {
        // TODO ovdje dodati implementaciju
        return null;
    }
}
```

Generirani kôd je klasa koja je pomoću bilješki (engl. *annotations*) označena kao web-usluga (@WebService()). Za svaku operaciju je napravljena jedna metoda. Za operaciju add je definirana metoda add i isto tako označena bilješkom @WebMethod(operationName = "add"). Parametri metode su dva broja a i b koji su također označeni bilješkama @WebParam(name = "a") i @WebParam(name = "b"). Na mjesto komentara označenog s ključnom riječi TODO treba postaviti implementaciju usluge.

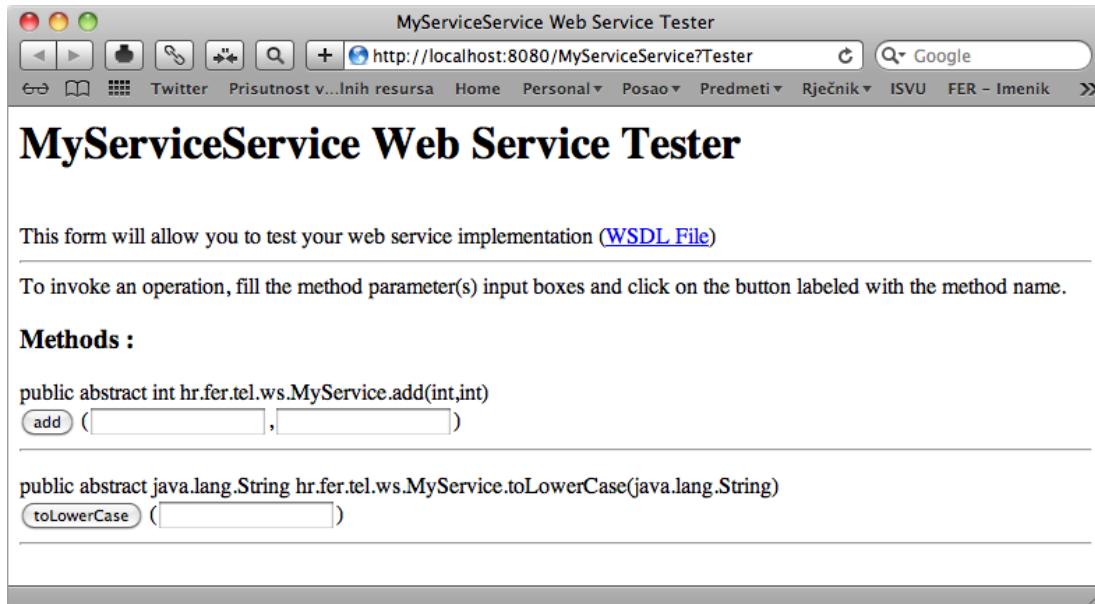
Nakon toga može se isporučiti web-uslugu na poslužitelj i možemo ju koristiti. U alatu NetBeans možemo i testirati uslugu ako ima jednostavne parametre što je u ovom slučaju točno. To radimo

<sup>23</sup> NetBeans, <http://www.netbeans.org>

<sup>24</sup> GlassFish, <https://glassfish.dev.java.net>

<sup>25</sup> Getting Started with JAX-WS Web Services, <http://netbeans.org/kb/docs/websvc/jax-ws.html>

tako da unutar projekta otvaramo: Web Services → MyService, desni klik na MyService, kliknuti na Test Web Service čime se pokreće web-preglednik (Slika 5.7) u kojem možemo upisati parametre usluge i pozvati ju.



Slika 5.7. Testiranje web-usluge iz preglednika

Nakon poziva npr. operacije add s parametrima 5 i 6 i klikom na gumb add dobijemo izvješće o pozvanoj web-usluzi (Slika 5.8). Ovdje vidimo kako točno izgleda SOAP poziva usluge:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Header/>
<S:Body>
<ns2:add xmlns:ns2="http://ws.tel.fer.hr/">
<a>5</a>
<b>6</b>
</ns2:add>
</S:Body>
</S:Envelope>
```

Isto tako vidimo i odgovor usluge s rezultatom:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:addResponse xmlns:ns2="http://ws.tel.fer.hr/">
<return>11</return>
</ns2:addResponse>
</S:Body>
</S:Envelope>
```

**Method parameter(s)**

Type	Value
int	5
int	6

**Method returned**

int : "11"

**SOAP Request**

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:add xmlns:ns2="http://ws.tel.fer.hr/">
      <a>5</a>
      <b>6</b>
    </ns2:add>
  </S:Body>
</S:Envelope>
```

**SOAP Response**

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:addResponse xmlns:ns2="http://ws.tel.fer.hr/">
      <return>11</return>
    </ns2:addResponse>
  </S:Body>
</S:Envelope>
```

Slika 5.8. Rezultat poziva web-usluge iz preglednika

Možemo pogledati i kako izgleda generirani WSDL:

```
<?xml version='1.0' encoding='UTF-8'?><!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.1.1-b09 (branches/2.1-6834; 2011-07-16T17:14:48+0000) JAXWS-RI/2.2.5-promoted-b04 JAXWS/2.2. --><!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.1.1-b09 (branches/2.1-6834; 2011-07-16T17:14:48+0000) JAXWS-RI/2.2.5-promoted-b04 JAXWS/2.2. -->
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
  xmlns:wspl_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://ws.tel.fer.hr/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://ws.tel.fer.hr/"
  name="MyServiceService">
<types>
<xsd:schema>
```

```

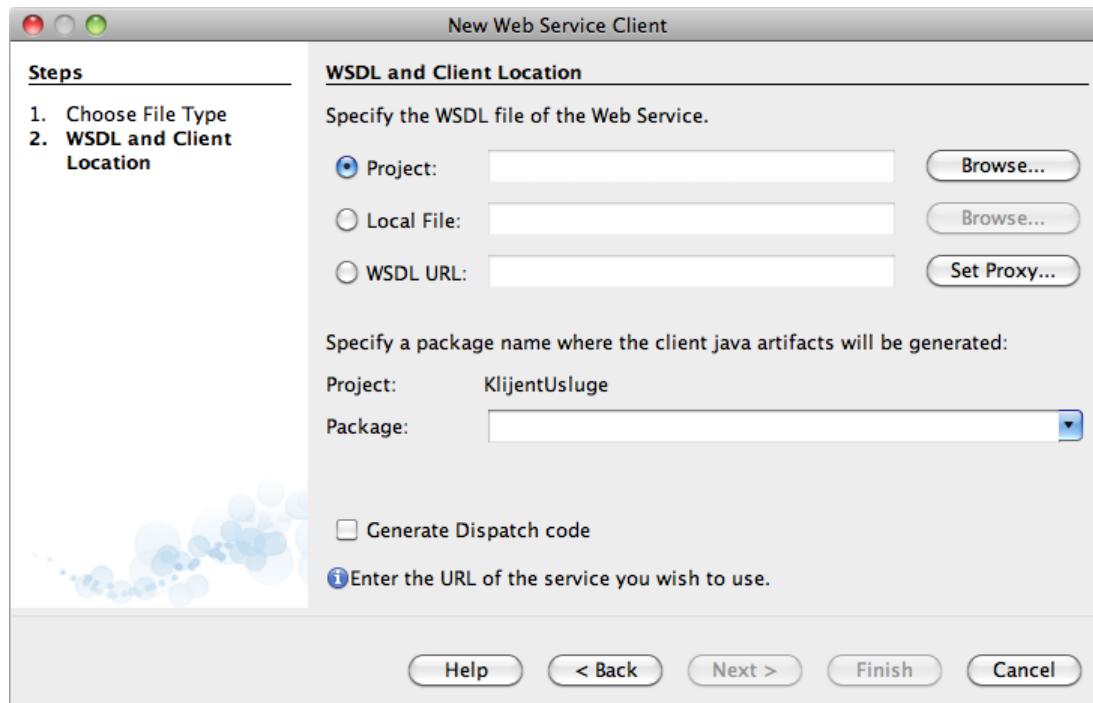
<xsd:import namespace="http://ws.tel.fer.hr/"
schemaLocation="http://localhost:8080/MyServiceService?xsd=1"/>
</xsd:schema>
</types>
<message name="add">
<part name="parameters" element="tns:add"/>
</message>
<message name="addResponse">
<part name="parameters" element="tns:addResponse"/>
</message>
<message name="toLowerCase">
<part name="parameters" element="tns:toLowerCase"/>
</message>
<message name="toLowerCaseResponse">
<part name="parameters" element="tns:toLowerCaseResponse"/>
</message>
<portType name="MyService">
<operation name="add">
<input wsam:Action="http://ws.tel.fer.hr/MyService/addRequest" message="tns:add"/>
<output wsam:Action="http://ws.tel.fer.hr/MyService/addResponse"
message="tns:addResponse"/>
</operation>
<operation name="toLowerCase">
<input wsam:Action="http://ws.tel.fer.hr/MyService/toLowerCaseRequest"
message="tns:toLowerCase"/>
<output wsam:Action="http://ws.tel.fer.hr/MyService/toLowerCaseResponse"
message="tns:toLowerCaseResponse"/>
</operation>
</portType>
<binding name="MyServicePortBinding" type="tns:MyService">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="add">
<soap:operation soapAction="" />
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
<operation name="toLowerCase">
<soap:operation soapAction="" />
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="MyServiceService">
<port name="MyServicePort" binding="tns:MyServicePortBinding">
<soap:address location="http://localhost:8080/MyServiceService"/>
</port>
</service>
</definitions>

```

Kako bismo iskoristili web-uslugu koju smo stvorili potrebno je pripremiti klijenta. Pošto klijent može biti bilo koji program onda treba napraviti novi projekt za Javine aplikacije i u njemu stvoriti klijenta web-usluge: File -> New File..., odabratи Web Services, Web Service Client. Nakon toga se otvara čarobnjak (Slika 5.9). Čarobnjak zahtijeva da ga se usmjeri na WSDL-ov dokument iz kojega se onda generira kôd koji će pozvati tu web-uslugu. Postoje 3 načina kako možemo doći do WSDL-ovog dokumenta:

- dohvaćanjem iz drugog projekta – što je moguće ako u istom alatu imamo i drugi projekt,
- lokalna datoteka – ako smo skinuli WSDL-ov dokument od nekud ili

- preko URL-a – možemo dohvatiti WSDL-ov dokument postojeće usluge na Internetu.



Slika 5.9. Čarobnjak za stvaranje klijenta web-usluge

Kada kliknemo na Finish onda se u pozadini kreira kôd koji ćemo koristiti za klijenta. Ako želimo u nekoj klasi pozvati uslugu onda trebamo otići na projekt i otvorimo: Web Service References -> MyServiceService -> MyServiceService -> MyServicePort. Nakon toga prevučemo add iz MyServicePort-a u klasu pa se u toj klasi generira sljedeći kôd:

```
private static int add(int a, int b) {
    hr.fer.tel.ws.MyServiceService service =
        new hr.fer.tel.ws.MyServiceService();
    hr.fer.tel.ws.MyService port = service.getMyServicePort();
    return port.add(a, b);
}
```

Jedino što trebamo da bismo pozvali ovu web-uslugu je pozvati metodu add s parametrima.

Primjer usluge koju možemo iskoristiti na ovaj način je CARNetov sustav za registraciju i upravljanje .hr domenama za registrare<sup>26</sup>.

U prošlom primjeru smo napravili uslugu koja koristi samo primitivne vrste podataka. U realnom slučaju usluge uglavnom koriste objekte, a rjeđe primitivne vrste. Ako želimo napraviti uslugu koja koristi objekte onda je prvo potrebno napraviti objekt koji će se koristiti. Kao primjer ćemo napraviti uslugu kojoj se šalje objekt vrste Person koji ima atribute ime, prezime i starost osobe. Slijedi kôd klase Person:

```
package hr.fer.tel.ws.fromjava;

public class Person {
    private String firstName, lastName;
    private int age;
```

---

<sup>26</sup> Uputa za web-usluge CARNetovog sustava za registraciju i upravljanje .hr domenama za registrare, <https://registrar.carnet.hr/wiki/index.php/Uvod>

```

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
}

```

Nakon što smo napravili klasu potrebno je u dizajnu usluge dodati novu uslugu pod nazivom complexOperation, a njoj dodamo parametar pod imenom person, a vrsta podatka je klasa Person (Slika 5.10).

complexOperation		Output	Faults	Description
Parameters	Parameter Name			Parameter Type
	person			hr.fer.tel.ws.fromjava.Person

Slika 5.10. Operacija s objektom kao parametrom

Alat generira implementaciju koja izgleda ovako:

```

@WebMethod(operationName = "complexOperation")
public String complexOperation(@WebParam(name = "person") Person person) {
}

```

Sada je potrebno napraviti implementaciju usluge koja vraća odgovor koji odgovara na pitanje je li osoba punoljetna ili nije. Implementacija je ovakva:

```

@WebMethod(operationName = "complexOperation")
public String complexOperation(@WebParam(name = "person") Person person) {
    if(person.getAge() >= 18)
        return "punoljetan/na";
    else
        return "maloljetan/na";
}

```

Na klijentskoj strani će alat stvoriti klasu Person koja ima samo metode set i get za svaki atribut i na taj način klijent može pozivati tu uslugu.

## 5.6 Web-usluge temeljene na dokumentima

Web-usluge temeljene na dokumentima razmjenjuju dokumente koji nisu preslikavanje objekata u XML, već se prvo definira struktura dokumenta koji se prenosi. Struktura dokumenta se definira pomoću XML Schemae. Nakon što se definira struktura dokumenta treba se napraviti WSDL-ov dokument koji uključuje strukturu (XML Schema). Iz WSDL-a se pomoću alata može napraviti kostur usluge koji se onda dopuni implementacijom. Isto tako se iz WSDL-a može napraviti i kostur klijenta

kao i kod Web-usluga temeljenih na pozivu udaljenih procedura. Dakle centralni dokument iz kojeg se radi klijent i poslužitelj je dokument WSDL koji uključuje strukturu dokumenta koji se prenosi, a opisan je XML Schemaom.

U ovom pristupu izradi web-usluga je naglasak na razmjeni dokumenata, a ne na slanju akcija koje usluga treba napraviti. Obično su ti dokumenti samostojeći tj. kontekst informacije je sadržan u dokumentu.

Na primjeru usluge bankovne transakcije ćemo pokazati kako se implementira usluga koja je temeljena na dokumentima. Usluga predstavlja bankovni sustav koji prima zahtjeve za transakcijama i transakcije može izvršiti ili odbiti. Transakcija se odbija ako je iznos koji se prenosi veći od 1000 i u tom slučaju se vraća dokument u kojem je opisan razlog odbijanja transakcije.

Prvo je potrebno napraviti strukturu dokumenata koji se prenose. U upitu se prenosi dokument koji je definiran imenom `transaction`, a njegova vrsta je složeni tip podataka `transactionType`. On se sastoji od broja računa (`accountNumber`) s kojeg se skida novac (`from`), broj računa na koji se stavlja novac (`to`) i iznos novca koji se prebacuje (`amount`). Broj računa je definiran kao primitivna vrsta podataka `int` isto kao i iznos. Rezultat provođenja transakcije (`transactionResultType`) je vrsta podatka koja može vratiti ili status `OK` u obliku primitivne vrste `boolean` ili će vratiti grešku koja je vrste `string`, a opisuje grešku.

Kako bismo napravili novu XML Schema potrebno je instalirati plugin XML Tools<sup>27</sup>. U alatu NetBeans prvo je potrebno stvoriti novu XML Schema (`New → Other ..., XML → XML Schema`) `bank.xsd`. Potrebno je napraviti ovakvu schemu (upute za rad se nalaze u arhivi koju treba skinuti s Interneta<sup>28</sup>):

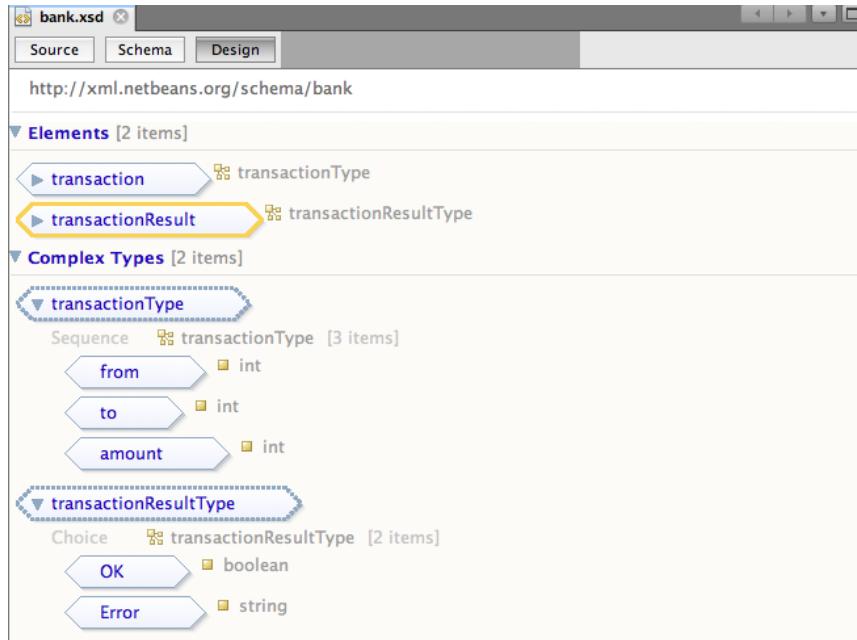
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://xml.netbeans.org/schema/bank"
    xmlns:tns="http://xml.netbeans.org/schema/bank"
    elementFormDefault="qualified">
    <xsd:complexType name="transactionType">
        <xsd:sequence>
            <xsd:element name="from" type="xsd:int"/>
            <xsd:element name="to" type="xsd:int"/>
            <xsd:element name="amount" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="transactionResultType">
        <xsd:choice>
            <xsd:element name="OK" type="xsd:boolean"/>
            <xsd:element name="Error" type="xsd:string"/>
        </xsd:choice>
    </xsd:complexType>
    <xsd:element name="transaction" type="tns:transactionType"/>
    <xsd:element name="transactionResult" type="tns:transactionResultType"/>
</xsd:schema>
```

Unutar korijenske oznake `xsd:schema` se nalaze oznake `xsd:complexType` koje definiraju vrste podataka i oznake `xsd:element` koji definiraju elemente. Prva oznaka `xsd:complexType` definira slijed (`xsd:sequence`) od 3 elementa (`xsd:element`). Svaki taj element ima atribut `name` i `type`. Atribut `type` definira vrstu podataka koji mogu biti u tom elementu. U ovom slučaju je to `xsd:int` što znači da u njemu mogu biti cijeli brojevi. Atribut `name` definira ime tog elementa tj. u SOAP-u će to biti naziv oznake. Druga oznaka `xsd:complexType` ima u sebi oznaku `xsd:choice` koja označava da samo jedan element može biti u tom kompleksnom dijelu. U ovom slučaju to je izvor

<sup>27</sup> Geertjan Wielenga, XML Schema Editor in NetBeans IDE 7.0.1,  
[http://blogs.oracle.com/geertjan/entry/xml\\_schema\\_editor\\_in\\_netbeans](http://blogs.oracle.com/geertjan/entry/xml_schema_editor_in_netbeans), (20.11.2011.)

<sup>28</sup> Getting Started With XML Schema Tools,  
<http://netbeans.org/projects/usersguide/downloads/download/NB61-SOAdocs.zip>/kb/61/soa/xmltools-start.html (20.11.2011.)

između elemenata s imenom `OK` ili `Error`. Ova XML Schema u pogledu dizajna je prikazana na slici (Slika 5.11).



Slika 5.11. XML Schema u pogledu dizajna

Nakon što je napravljena XML Schema treba napraviti WSDL (New → Other..., XML → WSDL Document). Nakon što je napravljen osnovni WSDL-ov dokument potrebno ga je promijeniti tako da izgleda ovako (upute za rad s WSDL Editorom<sup>29</sup>):

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="newWSDL" targetNamespace="http://j2ee.netbeans.org/wsdl/Bank"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns="http://xml.netbeans.org/schema/bank"
    xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:tns="http://j2ee.netbeans.org/wsdl/Bank"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <types>
        <xsd:schema targetNamespace="http://j2ee.netbeans.org/wsdl/Bank"
            xmlns:tns="http://j2ee.netbeans.org/wsdl/Bank">
            <xsd:import namespace="http://xml.netbeans.org/schema/bank"
                schemaLocation="bank.xsd"/>
        </xsd:schema>
    </types>
    <message name="TransferRequest">
        <part name="TransferRequest" element="ns:transaction"/>
    </message>
    <message name="TransferResponse">
        <part name="TransferResponse" element="ns:transactionResult"/>
    </message>
    <portType name="BankPortType">
        <operation name="Transfer">
            <input name="input1" message="tns:TransferRequest"/>
            <output name="output1" message="tns:TransferResponse"/>
        </operation>
    </portType>
    <binding name="BankBinding" type="tns:BankPortType">
        <operation name="Transfer">
            <input name="input1" message="tns:TransferRequest"/>
            <output name="output1" message="tns:TransferResponse"/>
        </operation>
    </binding>
</definitions>

```

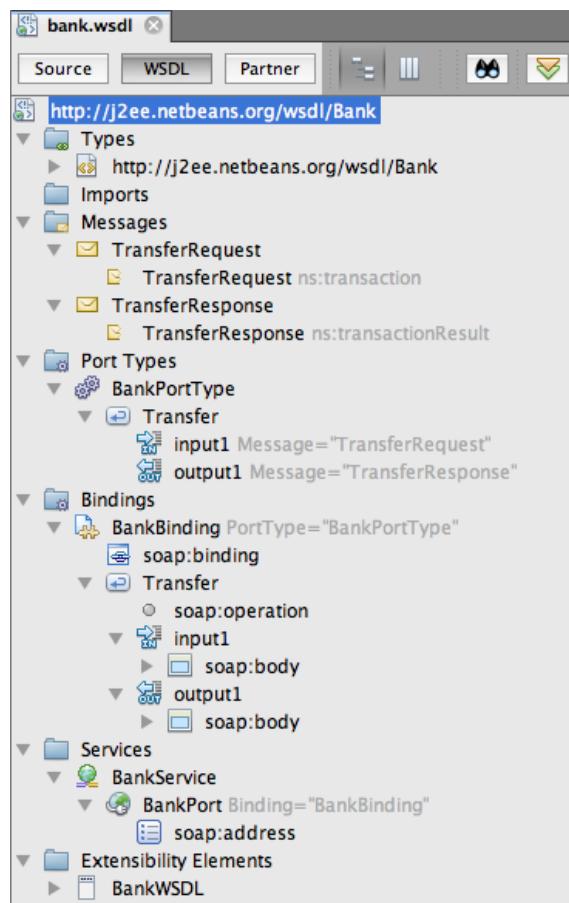
<sup>29</sup>Using the WSDL Editor, <http://docs.oracle.com/cd/E19182-01/820-6712/6ni1703g2/index.html>, (20.11.2011.)

```

<soap:binding style="document"
  transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="Transfer">
<soap:operation/>
<input name="input1">
<soap:body use="literal"/>
</input>
<output name="output1">
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="BankService">
<port name="BankPort" binding="tns:BankBinding">
<soap:address
location="http://localhost:8080/UWPosluzitelj/BankService"/>
</port>
</service>
<plnk:partnerLinkType name="BankWSDL">
<plnk:role name="BankPortTypeRole" portType="tns:BankPortType"/>
</plnk:partnerLinkType>
</definitions>

```

U ovom WSDL-ovom dokumentu možemo uočiti da je uključena shema bank.xsd u oznaci xsd:import. Atribut schemaLocation određuje gdje možemo naći XML Schemae. U oznaci soap:binding postavljen je atribut style koji ima vrijednost document što označava da razmjenjujemo dokumente, a ne preslikane objekte. U oznaci soap:body imamo atribut use koji ima vrijednost literal što označava da se podaci ne prenese kodirani, već da se koriste elementi koji su definirani u XML Schema i tako se prenose. Ovaj dokument u pogledu dizajna je prikazan na slici (Slika 5.12). Ostali elementi dokumenta WSDL nisu objašnjavani jer su objašnjeni u poglavljju 5.3.

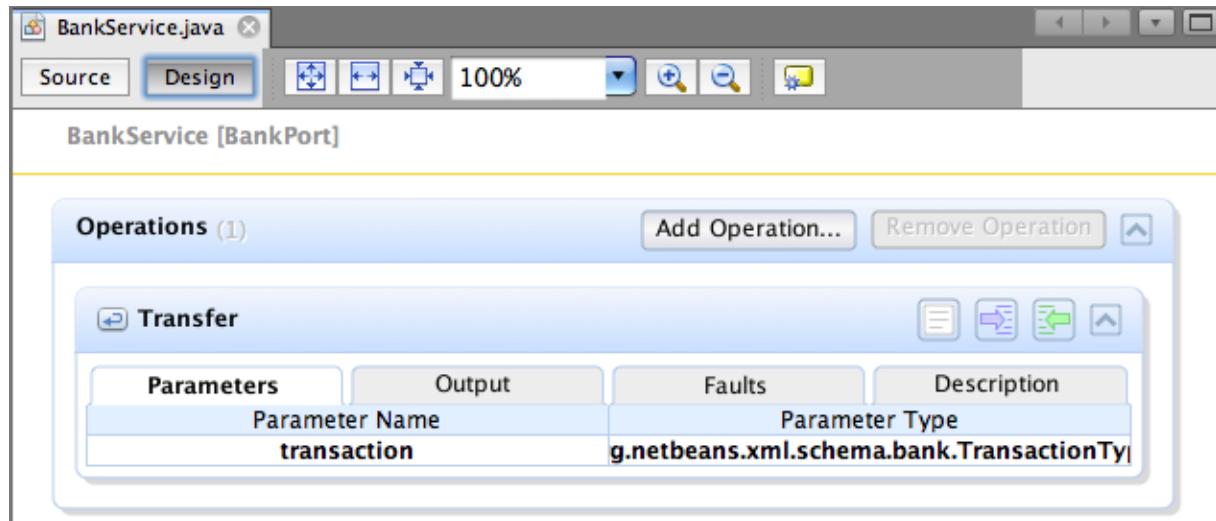


Slika 5.12. Dokument WSDL u pogledu dizajna

Nakon što je napravljen WSDL potrebno je napraviti i samu uslugu iz WSDL-a (New → Other..., Web Service → Web Service from WSDL). U čarobnjaku je potrebno napisati naziv usluge (BankService), projekt u kojem se stvara usluga, paket u kojem će biti generirani kostur usluge i treba odabrati napravljeni WSDL. Kostur usluge izgleda ovako:

```
@WebService(serviceName = "BankService", portName = "BankPort", endpointInterface =
"org.netbeans.j2ee.wsdl.bank.BankPortType",
targetNamespace = "http://j2ee.netbeans.org/wsdl/Bank",
wsdlLocation = "WEB-INF/wsdl/BankService/bank.wsdl")
public class BankService {
    public org.netbeans.xml.schema.bank.TransactionResultType
        transfer(org.netbeans.xml.schema.bank.TransactionType transferRequest)
    {
        //TODO implement this method
        throw new UnsupportedOperationException("Not implemented yet.");
    }
}
```

Pogled usluge u dizajnu je na slici (Slika 5.13).



Slika 5.13. Usluge temeljene na dokumentima iz pogleda dizajna

Sada je potrebno isprogramirati implementaciju. Ona izgleda ovako:

```
@WebService(serviceName = "BankService", portName = "BankPort", endpointInterface =
"org.netbeans.j2ee.wsdl.bank.BankPortType",
targetNamespace = "http://j2ee.netbeans.org/wsdl/Bank",
wsdlLocation = "WEB-INF/wsdl/BankService/bank.wsdl")
public class BankService {

    public TransactionResultType transfer(TransactionType transferRequest) {
        TransactionResultType result = new TransactionResultType();
        if(transferRequest.getAmount() > 1000) {
            result.setOK(false);
            result.setError("Iznos prijenosa je premašio Vaš limit.");
        } else {
            result.setOK(true);
        }

        return result;
    }
}
```

Klijent web-usluge se stvara na isti način kao i kod web-usluge temeljene na pozivu udaljenih procedura. Generirani kôd je sljedeći:

```
private static TransactionResultType transfer(TransactionType transferRequest) {
    BankService service = new BankService();
```

```

        BankPortType port = service.getBankPort();
        return port.transfer(transferRequest);
    }
}

```

U klijentu je potrebno još pozvati ovu metodu:

```

public static void main(String[] args) {
    TransactionType transaction = new TransactionType();
    transaction.setFrom(123);
    transaction.setTo(234);
    transaction.setAmount(1500);

    TransactionResultType result = transfer(transaction);

    if(result.isOK()) {
        System.out.println("Transakcija je uspjela!");
    } else {
        System.out.println("Transakcija nije uspjela! Razlog:" +
            result.getError());
    }
}
}

```

Prije poziva metode `transfer` potrebno je napraviti objekt iz kojeg će se generirati dokument koji se šalje web-usluzi. To radimo tako da stvorimo objekt iz klase `TransactionType` i pomoću metoda `set` postavljamo vrijednosti dva računa (`from` i `to`) i iznos koji se prenosi.

Nakon toga se poziva metoda `transfer` koja poziva uslugu, a njen rezultat je objekt `TransactionResultType` u koji su dodani podaci iz dokumenta koji nam je vratila usluga. Pomoću metoda `getError` ili `isOK` možemo dohvatiti podatke iz tog dokumenta.

Pogledajmo kako su izgledali upit i odgovor u HTTP-u za uspješnu transakciju:

- **upit:**

```

POST /UWPosluzitelj/BankService HTTP/1.1
Content-type: text/xml; charset="utf-8"
Soapaction: ""
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2,
*/*; q=.2
User-Agent: JAX-WS RI 2.1.6 in JDK 6
Host: localhost:8080
Connection: keep-alive
Content-Length: 235

```

```

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<transaction xmlns="http://xml.netbeans.org/schema/bank">
<from>123</from>
<to>234</to>
<amount>500</amount>
</transaction>
</S:Body>
</S:Envelope>

```

- **odgovor:**

```

HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.1
Java/Apple Inc./1.6)
Server: GlassFish Server Open Source Edition 3.1.1
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
Date: Wed, 21 Dec 2011 09:36:30 GMT

```

```

<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<transactionResult xmlns="http://xml.netbeans.org/schema/bank">
<OK>true</OK>

```

```
</transactionResult>
</S:Body>
</S:Envelope>
```

Odgovor u slučaju neuspjeha transakcije izgleda ovako:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/3.0 JSP/2.2 (GlassFish Server Open Source Edition 3.1.1
Java/Apple Inc./1.6)
Server: GlassFish Server Open Source Edition 3.1.1
Content-Type: text/xml; charset=utf-8
Transfer-Encoding: chunked
Date: Wed, 21 Dec 2011 09:42:11 GMT

<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<transactionResult xmlns="http://xml.netbeans.org/schema/bank">
<OK>false</OK>
<Error>Iznos prijenosa je premašio Vaš limit.</Error>
</transactionResult>
</S:Body>
</S:Envelope>
```

## 5.7 Web-usluge temeljene na prijenosu prikaza stanja resursa

Koncept je osmislio Roy Thomas Fielding u svojoj doktorskoj disertaciji (Fielding, 2000). U 5. poglavljju pod naslovom «Representational State Transfer (REST)» je definirao usluge koje se temelje na stanju resursa. REST je arhitekturni stil, a ne API ili standard. Postojeći standardi koji specificiraju URL, HTTP, XML mogu se koristiti za implementaciju sustava temeljenog na REST-u<sup>30</sup>. REST se koristi za izradu raspodijeljenih aplikacija kao što su web-aplikacije i web-usluge. Poslužitelj REST-a je svaki poslužitelj koji poslužuje zahtjeve temeljene na REST-u. Najrašireniji primjer arhitekture REST je Web.

Pojam REST se još pojavljuje u obliku «REST Way» i RESTful». Tri su stvari bitne za shvaćanje rada arhitekture REST<sup>31</sup>:

- klijent šalje zahtjev za resursom usluzi. U zahtjevu se šalje identifikator resursa te je poželjno da se pošalje i vrsta podataka koja se očekuje u odgovoru;
- odgovor sadrži prikaz stanja resursa u obliku niza okteta zajedno s metapodacima koji opisuju taj resurs (npr. parovi ime-vrijednost u zaglavlju protokola HTTP);
- dohvaćanje prikaza može uzrokovati promjenu stanja resursa.

Najčešće izvedbe usluga temeljenih na stanju resursa koriste protokol HTTP za prijenos podataka, URL-ove za identificiranje resursa te XML ili JSON kao format podataka koji se prenose. Pogledajmo na primjeru dva resursa. Prvi resurs može biti identificiran sljedećim URL-om: <http://localhost:8080/REST2011/rest/persons>. Kada se pošalje upit protokolom HTTP pomoću metode GET dobiti ćemo kao odgovor prikaz popisa korisnika koji će biti formatiran XML-om:

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
    <person uri="http://localhost:8080/REST2011/rest/person/1">
        <name>Ignac Lovrek</name>
    </person>
    <person uri="http://localhost:8080/REST2011/rest/person/2">
        <name>Ivana Podnar Žarko</name>
    </person>
    <person uri="http://localhost:8080/REST2011/rest/person/3">
        <name>Mario Kušek</name>
```

<sup>30</sup> Joe Gregorio, An Introduction to REST, <http://bitworking.org/news/373/An-Introduction-to-REST> (20.12.2011.)

<sup>31</sup> Mark Volkmann, REST Architectural Style, <http://jnb.ociweb.com/jnb/jnbNov2004.html> (20.12.2011.)

```
</person>
</persons>
```

Ovdje vidimo da je u XML-ovom dokumentu oznakama `person` označen svaki korisnik. Unutar te oznake imamo oznaku `name` koja u kojoj je ime i prezime korisnika, a u oznaci `person` imamo i atribut `uri` koji predstavlja identifikator pomoću kojeg možemo dohvatiti detalje korisnika. Kada pomoći protokola HTTP i njegove metode GET zatražimo resurs s identifikatorom <http://localhost:8080/REST2011/rest/person/2> dobiti ćemo prikaz korisnika:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
<id>2</id>
<firstname>Ivana</firstname>
<lastname>Podnar Žarko</lastname>
<phone>287</phone>
<room>C7-15</room>
</person>
```

REST pristup izradi web-usluga omogućuje efikasno korištenje metoda protokola HTTP koje se najčešće koriste za:

- GET – za dohvaćanje prikaza stanja resursa,
- POST – za slanje bloka podataka na obradu (npr. stvaranje novog resursa s novim identifikatorom), obično se koristi kada se druge metode ne mogu koristiti,
- PUT – za stvaranje ili promjenu postojećeg resursa,
- DELETE – za brisanje resursa.

Ako se metode koriste u prije navedene svrhe onda za svaku metodu možemo promatrati sljedeća tri svojstva:

- sigurnost (*safe*) – metoda je sigurna ako nema posljedica za podatke tj. podaci se ne mijenjaju nakon njene obrade;
- idempotentnost – metoda se može pozvati više puta i uvijek će rezultat obrade biti isti;
- *cachable* – rezultat se može privremeno spremiti u priručnu memoriju za kasniju upotrebu.

Razmotrimo sada koja metoda ima koja svojstva:

- GET
  - sigurnost – ne mijenja stanje resursa,
  - idempotentnost – nakon uzastopnog višekratnog izvršavanja je rezultat isti,
  - *cachable* – rezultat se može spremiti u priručnu memoriju,
- PUT
  - idempotentnost – nakon uzastopnog izvršavanja svaki puta će se stanje resursa postaviti u istu vrijednost i nje bitno koliko puta će se pozvati ova metoda,
- DELETE
  - idempotentnost – nakon prvog puta će resurs biti izbrisani i nakon svakog sljedećeg izvršavanja rezultat će biti izbrisani resurs,
- POST
  - nema niti jedno od navedenih svojstava.

Usluge temeljene na resursima su bez stanja što znači da usluga ne povezuje upite, već ih smatra neovisnima. Na taj način je moguće koristiti standardne mehanizme za privremeno spremanje podataka (*cache*) koji postoje na Webu.

Postupak izrade usluge temeljena na resursima se obično sastoji od sljedećih koraka<sup>32</sup>:

1. Potrebno je identificirati entitete kojima se želi omogućiti pristupanje (npr. lista korisnika, podaci o korisnicima);
2. Za svaki resurs treba definirati shemu identifikatora pomoću URL-a. Shema treba završavati imenicama, a ne glagolima jer glagoli označavaju radnju. Radnja nije resurs. Npr. <http://localhost:8080/REST2011/rest/persons> za popis korisnika i <http://localhost:8080/REST2011/rest/person/id> za pojedinog korisnika s identifikatorom id;
3. Za svaki resurs treba odrediti da li je moguće samo dohvaćati podatke o resursu (korištenje metode GET) ili je potrebno i manipulirati s njima (korištenje metoda POST, PUT i DELETE);
4. Svi resursi koji se dohvaćaju pomoću metode GET za tu metodu moraju imati svojstvo sigurnosti tj. obrada metode GET na poslužitelju ne smije mijenjati stanje resursa;
5. Svaki resurs treba vratiti što manje detalja tj. treba stavljati poveznice na druge resurse u slučaju da klijent želi detalje resursa ili saznati povezanost s drugim resursima;
6. Usluge treba dizajnirati tako da se postepeno (kroz više upita koji se otkrivaju preko poveznica tj. URL-ova) otkrivaju podaci o resursima. Nije dobro sve staviti u jedan veliki dokument;
7. Za svaku metodu svakog resursa potrebno je definirati formate upita i odgovora. Ako se koristi XML onda je dobro za te opise koristiti XML Schema;
8. Opisati kako se web-usluge mogu koristiti npr. WSDL-om ili HTML-om.

Za implementaciju usluga temeljenih na resursima za svaki programski jezik postoje različite programske knjižnice. U Javi je definiran standard JSR-311<sup>33</sup>. Aktualna inačica je 1.0 koja je standardizirana 2008. Osim standarda postoje i knjižnice za izradu web-usluga temeljenih na resursima: Restlet, AXIS 2.0, REST EASY, Apache CXF, Jearsy, Serfj, ... U alatu Netbeans je ugrađena podrška za korištenje knjižnice Jearsy koja podržava standard i koja će biti pokazana na primjeru. Upute za izradu web-usluga možete naći na različitim stranicama<sup>34</sup>.

Web-usluga omogućuje pristupanje/manipulaciju s podacima koji su negdje spremљeni. Uobičajeno mjesto je da su ti podaci spremljenu u relacijsku bazu podataka. Za pristup tim podacima možemo koristiti SQL upite koristeći implementacije JDBC-a (Java Database Connection) za bazu koju koristimo. Pošto Java koristi objekte iz odgovora SQL-a je potrebno podatke prebaciti u objekte koji odgovaraju podacima iz baze podataka. Klase koje opisuju takve objekte se zovu *Entity Classes*. U Javi postoje knjižnice koje znaju stvoriti *Entity Classes* i automatski pretvarati objekte u SQL upite prilikom stvaranja ili mijenjanja podataka i obrnuto kada samo dohvaćamo podatke iz baze. Za to se koristi JPA (Java Persistence API). S alatom Netbeans dolazi i aplikacijski poslužitelj u kojem se nalazi web-poslužitelj i baza podataka Derby koji ćemo iskoristiti za izradu primjera web-usluge.

---

<sup>32</sup>Roger L. Costello, Building Web Services the REST Way, <http://www.xfront.com/REST-Web-Services.html> (20.12.2011.)

<sup>33</sup>JAX-RS: Java™ API for RESTful Web Services, <http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html> (20.12.2011.)

<sup>34</sup>JAX-RS Tutorial, <http://www.mkyong.com/tutorials/jax-rs-tutorials/> (22.12.2011.)

Getting Started with RESTful Web Services, <http://netbeans.org/kb/docs/websvc/rest.html> (22.12.2011.)

Sangeetha S.: JAX-RS: Developing RESTful Web Services in Java, <http://www.devx.com/Java/Article/42873/1954> (22.12.2011.)

Jeff Rubinoff: NetBeans to Generate Simpler RESTful Web Services, <http://netbeans.dzone.com/nb-generate-simpler-rest> (22.12.2011.)

Lars Vogel: REST with Java (JAX-RS) using Jersey – Tutorial, <http://www.vogella.de/articles/REST/article.html> (22.12.2011.)

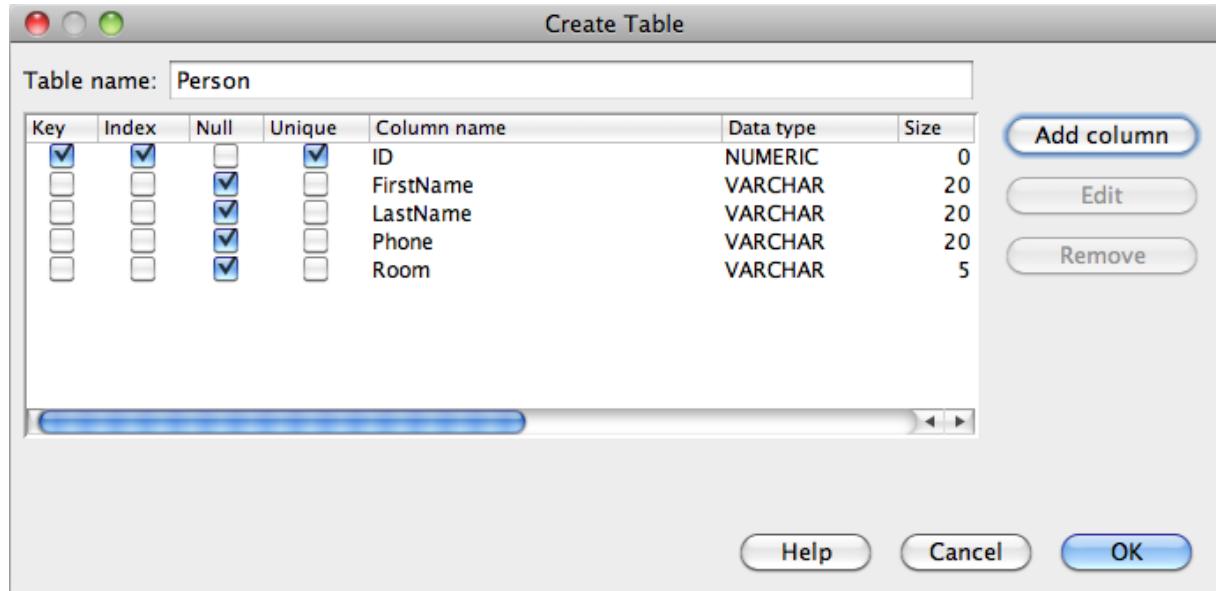
### 5.7.1 Pokretanje baze podataka Derby i stvaranje tablica

U kratici services odaberemo → Databases → Java DB → Start Server. Kada se poslužitelj pokrene možemo stvoriti bazu podataka: services → Databases → Java DB → Create Database. Upišemo sljedeće parametre: Database Name: rassus, User Name: app (mora biti app), passwd: app.

Nakon toga je potrebno postaviti vezu prema bazi. Prvo se pokreće aplikacijski poslužitelj (services → Servers → GlassFish Server 3.1 → Start), a nakon toga se treba preko administratorske konzole podesiti JDBC (services → Servers → GlassFish Server 3.1 → View Admin Console). Inicijalni administratorski parametri su: Login: admin i Password: adminadmin. Nakon toga odaberemo Resources → JDBC → Connection Pools i tamo kliknemo na new. U čarobnjaku upišemo sljedeće parametre: Name: rassus, Resource Type: javax.sql.DataSource, Database Vendor: JavaDB. Nakon toga kliknemo na Next i podesimo Additional Parameter: DatabaseName: rassus, Password: app, PortNumber: 1527, ServerName: localhost, User: app, ConncetionAttributes: ;create=true, a ostale parametre pobrisati. U kartici General kliknuti na ping za provjeru konekcije.

Sljedeći korak je stvaranje JDBC resursa. To radimo isto tako preko Admin Console (Resources → JDBC → JDBC Resources). Napravimo novi resurs sa sljedećim parametrima: JNDI Name: jdbc/rassus, Pool: odabratи onaj gore napravljen.

Sada je još potrebno stvoriti tablice i upisati neke podatke. Tablicu stvaramo tako da u Netbeansu odaberemo services → Databases → odabratи jdbc:derby://localhost:1527/rassus. Na to napravimo desni klik → Connect. Potrebno je otvoriti APP, pa desni klik na Tables → Create Table... i upišemo Table Name: Person. Onda dodamo kolone sa parametrima na slici (Slika 5.14).



Slika 5.14. Parametri tablice Person u bazi podataka

Kada smo napravili tablicu treba dodati i neke podatke. To se radi tako da se otvori Tables, odabere tablica Person, desni klik → View Data... U tom prozoru je potrebno dodati novu kolonu i upisati podatke (Slika 5.15).

#	ID	FIRSTNAME	LASTNAME	PHONE	ROOM
1	1	Ignac	Lovrek	302	<NULL>
2	2	Ivana	Podnar Žarko	287	C7-15
3	3	Mario	Kušek	301	C8-05

Slika 5.15. Podaci u tablici Person

### 5.7.2 Stvaranje web-usluge i testiranje

Svaka web-usluga se mora izvršavati unutar web-aplikacije pa je prvo potrebno napraviti web-aplikaciju pod nazivom RassusREST(pogledajte u poglavlju 5.5). U web-aplikaciji treba stvoriti *Entity Classes* (New File ... -> Persistence -> Entity Classes from Database, odabratи Data Source: jdbc/rassus). U čarobnjaku treba prebaciti u *selected tables* one tablice koje želimo koristiti. Upisati paket u kojem želimo da se stvore te klase (Package: hr.fer.tel.rassus.entities).

Sljedeći korak je stvaranje web-usluga tako da desnim klikom na paket u kojem se nalaze *Entity Classes* u padajućem izborniku odaberemo New → RESTful Web Services from Entity Classes... U čarobnjaku odaberemo klasu Person. Zatim klinemo na Next, u Resource Package stavimo hr.fer.tel.rassus.entities.service. Kada kliknemo Finish otvara se novi čarobnjak u kojem treba odabrati Create default Jearsy REST servlet adaptor in web.xml, a u REST Resources Path: stavimo /rest.

Stvorena je klasa PersonFacadeREST u kojoj je pomoću bilješki (annotations). Pogledajmo jedan isječak koda:

```
@Stateless
@Path("hr.fer.tel.rassus.entities.person")
public class PersonFacadeREST extends AbstractFacade<Person> {
    ...
    @GET
    @Path("{id}")
    @Produces({"application/xml", "application/json"})
    public Person find(@PathParam("id") Integer id) {
        return super.find(id);
    }

    @GET
    @Override
    @Produces({"application/xml", "application/json"})
    public List<Person> findAll() {
        return super.findAll();
    }
    ...
}
```

Bilješka @Path ima za parametar put do usluge. Vrijednost hr.fer.tel.rassus.entities.person treba promijeniti u person. Nakon što snimimo klasu i isporučimo projekt na aplikacijski poslužitelj, uslugu možemo testirati desnim klikom na projekt → Test RESTful Web Services ili otvaranjem preglednika na URL-u <http://localhost:8080/RassusREST/rest/person>. Rezultat je:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<people>
<person>
<firstname>Ignac</firstname>
<id>1</id>
<lastname>Lovrek</lastname>
<phone>302</phone>
</person>
<person>
<firstname>Ivana</firstname>
```

```

<id>2</id>
<lastname>Podnar Žarko</lastname>
<phone>287</phone>
<room>C7-15</room>
</person>
<person>
<firstname>Mario</firstname>
<id>3</id>
<lastname>Kušek</lastname>
<phone>301</phone>
<room>C8-05</room>
</person>
</people>

```

Na URL-u <http://localhost:8080/RassusREST/rest/person/2> se dobije:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
<firstname>Ivana</firstname>
<id>2</id>
<lastname>Podnar Žarko</lastname>
<phone>287</phone>
<room>C7-15</room>
</person>

```

Metoda `findAll` označena je sa sljedećim bilješkama veznima za REST:

- `@GET` – ova metoda se poziva kada se šalje zahtjev HTTP GET i
- `@Produces {"application/xml", "application/json"}` – metoda generira sadržaj u formatima XML ili JSON ovisno o tome što je klijent u zahtjevu poslao da može primiti.

Najvažnije bilješke su:

- `@GET, @POST, @PUT, @DELETE i @HEAD` – označava koje HTTP-ove metode će pozvati ovu metodu u Javi,
- `@Path` – definira dio URL-a unutar web-aplikacije,
- `@Consumes` – definira koji format podataka može prihvatiti,
- `@Produces` – definira koji format podataka može vratiti kao rezultat,
- `@PathParam` – u bilješki `@Path` u putu možemo u vitičastim zagradama označiti dio puta, a pomoću `@PathParam` možemo dohvatiti te parametre,
- ...

## 5.8 Uslužno orijentirana arhitektura

Uslužno orijentirana arhitektura (SOA – Service Oriented Architecture) je pristup/način izrade usluga. Prilikom dizajniranja usluga u uslužnoj orijentiranoj arhitekturi bitno svojstvo je razdvajanje interesa (engl. *separation of concerns*). Usluge su raspodijeljene fizički i na različite pružatelje usluga. Npr. usluga odobravanja kredita koju pruža banka može pozivati uslugu provjere hipoteke nad nekretninom koja se stavlja kao osiguranje za kredit. Tu uslugu provjere hipoteke može pružati neka državna institucija. Isto tako banka može provjeriti podatke klijenta pomoći usluge u policijskoj upravi i sl.

U ovakvoj arhitekturi svaka se usluga može nadograđivati bez obzira na druge usluge i na taj način tvoriti složene usluge. Kako bi jedna usluga koristila drugu mora znati kako ju koristiti (lokacija, opis, poruke). Najčešća implementacija uslužno orijentirane arhitekture je pomoći web-usluga zato što ih je tako najlakše napraviti i to je najraširenija tehnologija, ali SOA nije tehnički vezana za neku tehnologiju.

Svojstva uslužno orijentirane arhitekture su:

- slaba povezanost usluga – jedna usluga ne ovisi o tehnologiji implementacije druge usluge,

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

### 5.8.1 Jezik BPEL

Jezik BPEL (*Business Process Execution Language*) je jezik za opis poslovnog procesa pomoću XML-a<sup>35</sup>. Pomoću njega se mogu opisati interakcije između web-usluga. BPEL nije metodologija ili proces. On definira ponašanje jednog entiteta, omogućuje jednostavnu manipulaciju podacima. BPEL ima elemente programskog jezika kao što su: petlje, uvjete, variable, pozivanje usluga, čekanje poziva usluge, ..., ali nije namijenjen za složene proračune.

Za izvršavanje BPEL-a postoje «procesori» koji mogu izvršavati BPEL procese. Jedna implementacija u Javi je Java Business Integration (JBI) runtime environment koji se može koristiti u alatu Netbeans.

## 5.9 Web-aplikacije koje koriste web-usluge

Kako svaki program može koristiti web-usluge tako i web-aplikacija može koristiti web-usluge. Postoje dva modela korištenja:

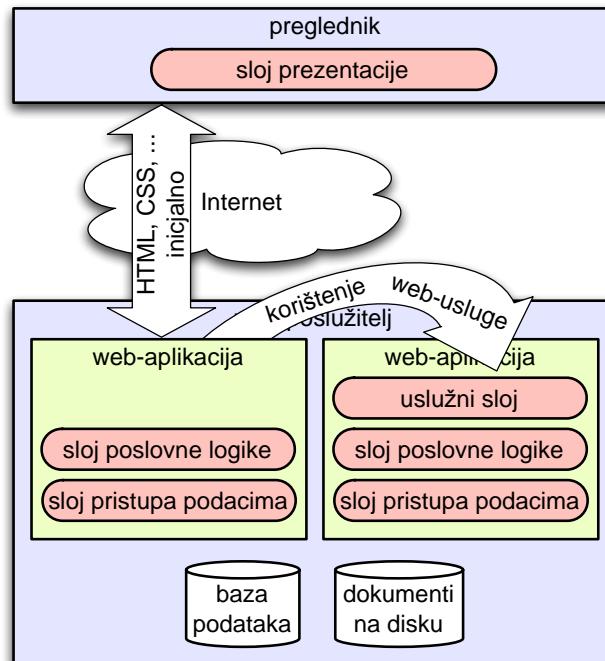
- dio web-aplikacije na poslužitelju koristi web-usluge i
- dio web-aplikacije na klijentu (u pregledniku) koristi web-usluge.

Pogledajmo slučaj kada dio na web-poslužitelju koristi web-uslugu (Slika 5.16).

Kada klijent pošalje zahtjev na web-poslužitelj onda web-poslužitelj u obradi tog zahtjeva može pozvati drugu web-uslugu koja može biti na istom ili nekom drugom web-poslužitelju. Osim korištenja samo jedne web-usluge moguće je korištenje više njih kako bi se integrirali rezultati u jednom odgovoru. Ovakav pristup je uobičajen u poslovnom okruženju gdje tvrtka ne želi omogućiti da se usluge koriste izvan poslovne mreže.

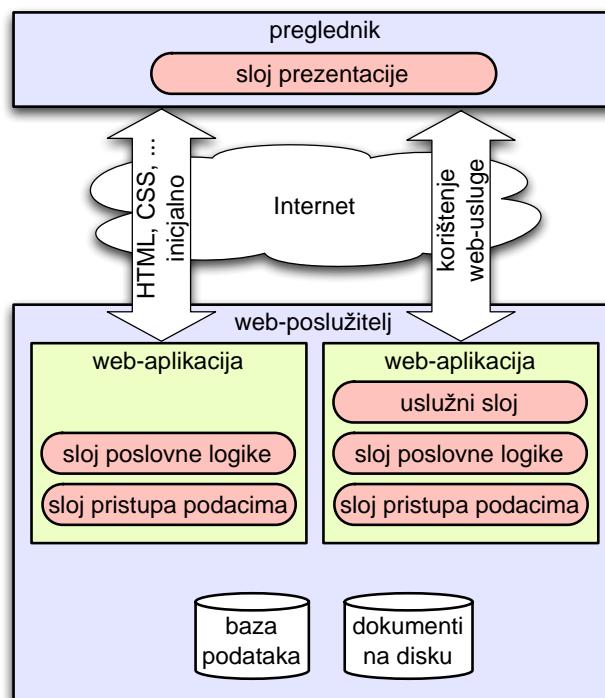
---

<sup>35</sup>Web Services Business Process Execution Language Version 2.0, <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html> (22.12.2011.)



Slika 5.16. Dio web-aplikacije na poslužitelju koristi web-usluge

Drugi pristup možemo vidjeti na slici (Slika 5.17).



Slika 5.17. Dio web-aplikacije na klijentu koristi web-usluge

U ovom slučaju preglednik kroz pozive iz JavaScripta koristi web-uslugu na poslužitelju. Ovakve usluge su danas uobičajene na Internetu. Na taj način se omogućuje trećim entitetima da koriste usluge i integriraju ih u svoje web-aplikacije. Ovakve web-usluge su vrlo često usluge temeljene na resursima.

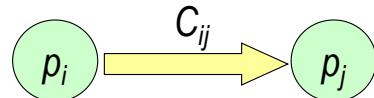
**5.10 Pitanja za učenje i ponavljanje**

- 5.1. Skicirajte i objasnite arhitekturu web-usluge te interakcije između komponenti web-usluge.
- 5.2. Navedite i ukratko objasnite tri vrste web-usluga.
- 5.3. Prikažite arhitekturu i objasnite korištenje usluge Weba.
- 5.4. Navedite dva osnovna načina rada protokola SOAP i objasnite kako se poruka SOAP šalje pomoću protokola HTTP.
- 5.5. Navedite i objasnite primjenu dokumenta u WSDL-u. Od kojih se dijelova sastoji takav dokument?
- 5.6. Objasnite svojstvo idempotentnosti za web-usluge temeljene na REST-u.

## 6 MODEL RASPODIJELJENOG SUSTAVA

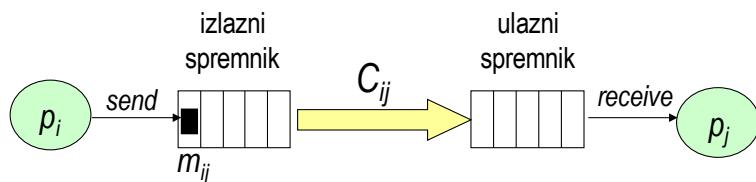
### 6.1 Osnovni model raspodijeljenog sustava

Osnovni model raspodijeljenog sustava se sastoji od skupa autonomnih i asinkronih procesa  $p_1, p_2, \dots, p_n$  koji komuniciraju razmjenom poruka putem komunikacijske mreže. Komunikacijsku poveznicu između 2 procesa  $p_i$  i  $p_j$  možemo modelirati kanalom  $C_{ij}$  koji opisuje virtualnu poveznicu među procesima, a ne fizički link, kao što je prikazano slikom (Slika 6.1). Kanal  $C_{ij}$  je usmjereni kanal tako da proces  $p_i$  može slati poruke samo do procesa  $p_j$  (npr. poruku  $m_{ij}$ ).



Slika 6.1. Primjer dva procesa i komunikacijskog kanala

Važno je navesti svojstva ovog modela raspodijeljenog sustava. 1) Ako prepostavimo da se procesi izvode na različitim procesorima, oni su međusobno autonomni i asinkroni, a prijenos poruka je također asinkron. 2) Procesi ne dijele zajednički memoriji prostor. 3) Zbog kašnjenja paketa pri prijenosu na fizičkom liku, neminovno se javlja kašnjenje poruka pri komunikaciji procesa. 4) Procesi ne koriste jedinstveni zajednički sat te se javlja problem sinkronizacije vremena u raspodijeljenim sustavima.

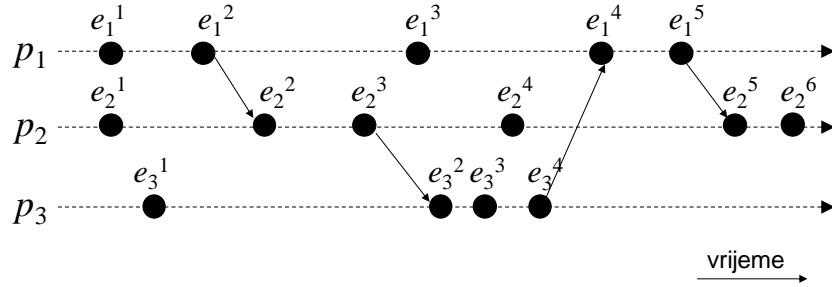


Slika 6.2. Primjer komunikacije dva procesa

Prilikom komunikacije procesi međusobno razmjenjuju poruke putem komunikacijskog medija tako da koriste operatore *send* i *receive*. Slika 6.2 prikazuje primjer takve komunikacije. Izvođeni proces koristi operator *send* kojim pohranjuje poruku u izlazni spremnik i priprema je za prijenos kanalom, dok operator *receive* čita poruku iz dolaznog spremnika i prosljeđuje je odredišnom procesu.

Sljedeće je pitanje kako opisati raspodijeljeno izvođenje algoritama na procesima ovim modelom. Raspodijeljeno izvođenje nužno je određeno izvođenjem akcija na pojedinim procesima te njihovom komunikacijom. Izvođenje procesa sastoji se od izvođenja akcija tijekom vremena. Akcije se modeliraju događajima koji mogu biti 1) unutarnji događaji (npr. obrada podataka na procesu koja proces dovodi u novo stanje), 2) slanje poruke i 3) primanje poruke. Izvođenje svakog od navedenih događaja mijenja stanje procesa, a događaji slanja i primanja poruke uz promjenu stanja procesa mijenjaju i stanje komunikacijskog kanala (slanje i primanje poruke utječe na komunikacijski kanal koji prenosi poruku). Slijed događaja na procesu  $p_i$ :  $e_i^1, e_i^2, e_i^3, \dots, e_i^x$ , (gdje se događaj  $e_i^2$  dogodio prije  $e_i^3$ ).

Primjer izvođenja 3 raspodijeljena procesa prikazan je vremenskim dijagramom na slici (Slika 6.3). Točke označavaju događaje, a strelice prijenos poruke na kanalu između 2 procesa. Kanali na ovoj slici nisu eksplicitno označeni. Događaji  $e_1^1, e_2^1, e_3^1, e_1^3, e_3^3, e_2^4$  i  $e_2^6$  predstavljaju unutarnje događaje. Događaji  $e_1^2, e_2^3, e_3^4$  i  $e_1^5$  predstavljaju slanje poruke, a  $e_2^2, e_3^2, e_1^4$  i  $e_2^5$  primanje poruke.



Slika 6.3. Primjer izvođenja 3 raspodijeljena procesa

*Uzročna ovisnost događaja.* Uzročna relacija označava uzročnu ovisnost dva događaja tijekom raspodijeljenog izvođenja. Lamport je naziva relacijom “*happens before*” jer je u raspodijeljenom sustavu za određene primjene važno donijeti odluku o tome koji se događaj sa sigurnošću zbio prije nekog drugog događaja (Kshemkalyani & Singhal, 2008). Stoga je definirana *uzročna relacija ovisnosti događaja* (označava se znakom  $\rightarrow$ ): Za dva događaja  $e_i^x$  i  $e_j^y$  kažemo da je događaj  $e_j^y$  uzročno povezan s događajem  $e_i^x$  ako se može pokazati da se događaj  $e_i^x$  nužno dogodio prije događaja  $e_j^y$  i pišemo  $e_i^x \rightarrow e_j^y$ .

Nedvojbeno se može tvrditi da je događaj slanja poruke na nekom komunikacijskom kanalu nužno prethodio događaju primanja te iste poruke. Stoga nužno postoji uzročna ovisnost između slanja i primanja iste poruke preko jednog kanala. Na primjeru sa slike (Slika 6.3) vrijedi  $e_1^2 \rightarrow e_2^2$  i  $e_3^2 \rightarrow e_1^4$ . Osim uzročne ovisnosti slanja i primanja iste poruke moguće je isto tako na jednome procesu odrediti slijednost događaja, te je stoga moguće odlučiti o njihovoj uzročnoj ovisnosti, npr.  $e_1^1$  se nužno dogodio prije  $e_1^2$ . Posljednje pravilo uzročnosti je tranzitivna uzročnost događaja za koju vrijedi sljedeće: za tri procesa  $e_i^x$ ,  $e_j^y$  i  $e_k^z$  vrijedi sljedeće:  $e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \Rightarrow e_i^x \rightarrow e_j^y$ . Navedena tri pravila uzročne ovisnosti događaja definirana su sljedećom formulom

$$e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow e_j^y, (i = j) \wedge (x < y) \\ e_i^x \rightarrow_{msg} e_j^y \\ e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$$

u kojoj prvi redak označava slijedne događaje na istome procesu, drugi redak se odnosi na slanje i primanje poruke *msg*, a treći redak definira tranzitivnu uzročnost. Primjerice, na temelju prethodne formule možemo pisati  $e_1^1 \rightarrow e_3^3$  i  $e_3^3 \rightarrow e_2^6 \Rightarrow e_1^1 \rightarrow e_2^6$ .

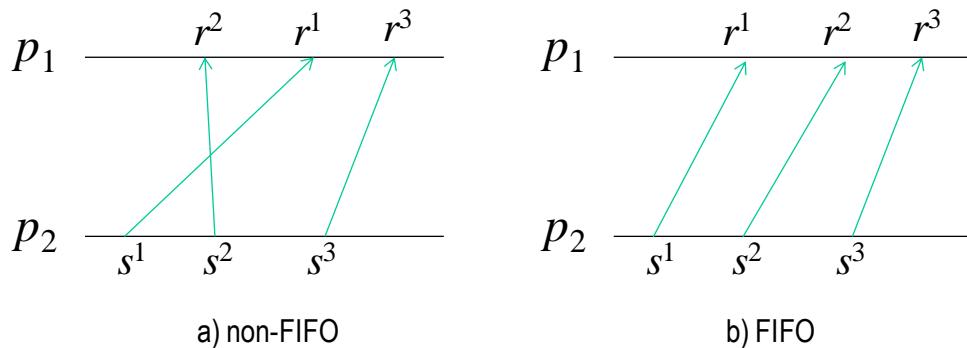
Uzročna relacija neovisnosti dvaju događaja (oznaka:  $\not\rightarrow$ ) označava neovisnost dvaju događaja tijekom raspodijeljenog izvođenja te stoga ne možemo prepostaviti koji se događaj zbio prije nekog drugog događaja. Na primjeru sa slike (Slika 6.3) vrijedi sljedeće  $e_1^3 \not\rightarrow e_3^3$  i  $e_2^4 \not\rightarrow e_3^1$ . Za dva događaja  $e_i$  i  $e_j$  vrijede sljedeća pravila:

- $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$ , tj. ako je  $e_j$  uzročno neovisan o  $e_i$ , ne možemo ništa zaključiti o neovisnosti  $e_i$  i  $e_j$  te oni mogu biti uzročno ovisni ili uzročno neovisni,
- $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$ , tj. ako je  $e_j$  uzročno ovisan o  $e_i$ , to implicira uzročnu neovisnost  $e_i$  o  $e_j$  i
- ako 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  $e_i \parallel e_j$ . Valja primjetiti da relacija  $\parallel$  nije tranzitivna, tj.  $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow (e_i \parallel e_k)$ .

Prilikom modeliranja komunikacijskog kanala potrebno je definirati slijednost isporuke poruka od izvorišnog do odredišnog procesa. Slijed isporuke poruka je važan aspekt raspodijeljenog sustava jer

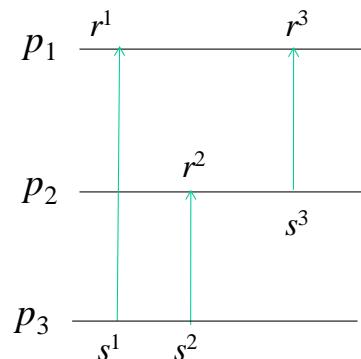
direktno utječe na ponašanje i implementaciju raspodijeljenih procesa. Koriste se sljedeći modeli kanala:

- FIFO (first-in, first-out) kanal čuva slijednost poruka i ponaša se kao rep,
- non-FIFO kanal ne čuva slijednost poruka i ponaša se kao skup,
- kanal koji osigurava sinkronu slijednost: slanje i primanje poruke događa se istovremeno i
- kanal koji osigurava uzročnu slijednost (*causal ordering*, CO): osigurava da uzročno povezani događaji slanja dviju poruka istom primatelju rezultiraju primanjem u slijedu kojim su poslati.



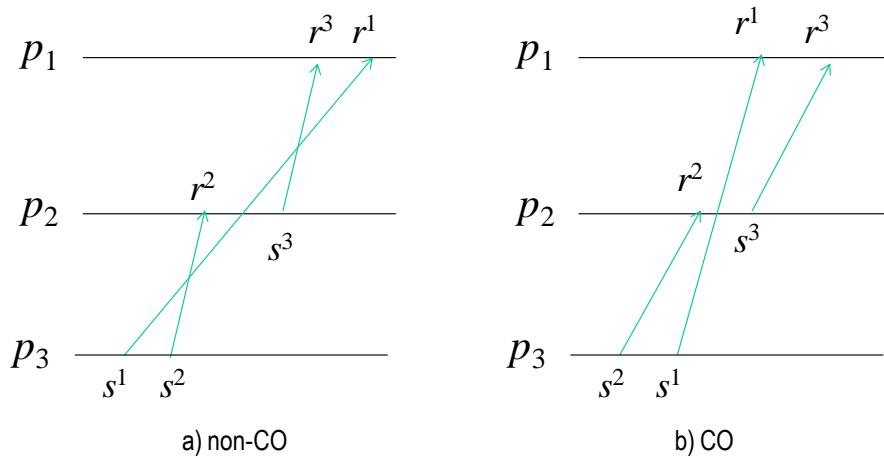
Slika 6.4. Primjer izvođenja a) non-FIFO i b) FIFO

Primjeri kanala FIFO i non-FIFO su prikazani slikom (Slika 6.4). Na slici a) se vidi da poruke nisu primljene u redoslijedu kojim su poslane na kanalu  $C_{21}$ , dok su na slici b) sve poruke primljene redoslijedom kojim su poslane na kanalu  $C_{21}$ . Slika 6.5 prikazuje kanal koji osigurava sinkronu slijednost i kod kojega se prepostavlja da se slanje i primanje poruka na kanalu izvodi istovremeno.



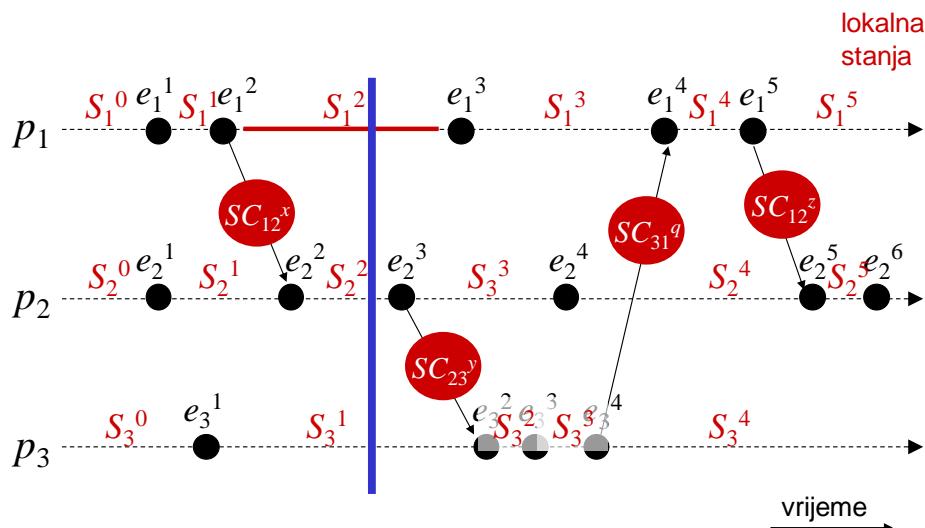
Slika 6.5. Primjer sinkronog izvođenja

Slika 6.6.a) prikazuje primjer kanala koji ne osiguravaju uzročnu slijednost jer su poruke primljene na procesu  $p_1$  (događaji  $r_3$  i  $r_1$ ) u obrnutom redoslijedu od slanja tih poruka s obzirom da su događaji slanja poruka  $s_1$  i  $s_3$  uzročno povezani. Slika 6.6.b) prikazuje primjer kanala koji osiguravaju uzročnu slijednost jer se za uzročno ovisne procese ( $s^2 \rightarrow s^1$ ,  $s^2 \rightarrow s^3$ ) razlikuju odredišta poruka.



Slika 6.6. Primjer izvođenja a) non-CO i b) CO

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



Slika 6.7. Primjer lokalnog/globalnog stanja

Na slici (Slika 6.7) su crveno naznačena lokalna stanja procesa i kanala. Na promjenu svakog stanja utječu događaji (unutarnji, primanje ili slanje poruke). Svaki proces možemo također opisati automatom stanja gdje prijelaze između stanja čine navedeni događaji. Globalno stanje je skup lokalnih stanja procesa i kanala te se kontinuirano mijenja u vremenu. Na slici (Slika 6.7) je plavom crtom naznačen trenutak  $t$  kada je globalno stanje sustava  $GS(t) = \{S_1^2, S_2^2, S_3^1, SC_{12}^x, SC_{23}^y, SC_{31}^q, SC_{12}^z\}$ , tj. skup stanja svih procesa i kanala. Na primjeru prepostavljamo da se koriste samo 3 kanala na čija stanja utječu poruke prenesene tim kalanima.

Osnovni model raspodijeljenog sustava može se proširiti tako da se raspodijeljeni sustav modelira kao sinkroni ili asinkroni sustav.

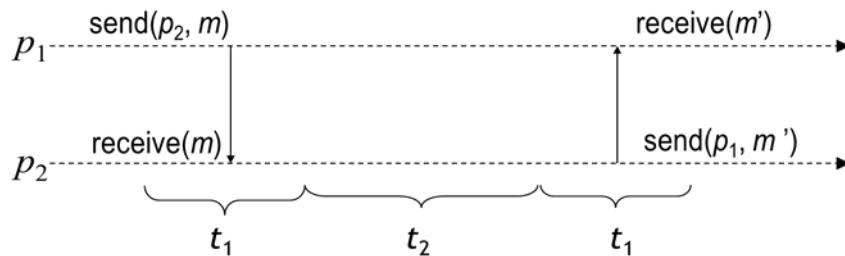
Sinkroni model se koristi isključivo radi pojednostavljenja osnovnog modela uz pretpostavku da svi procesi izvode korake istovremeno, dok se za asinkroni model pretpostavlja da procesi izvode akcije u proizvolnjom slijedu te da postoji neodređenost vezana uz izvođenje akcija u vremenu. Asinkroni model predstavlja realnu situaciju za raspodijeljene sustave, no prilično ga je teško analizirati te se

stoga često koristi sinkroni model kao pojednostavljenje koje može biti korisno za razumijevanje i analizu raspodijeljenog sustava.

Prepostavke sinkronog modela su sljedeće:

- svi procesi u sustavu imaju potpuno sinkronizirana lokalna vremena i
- poznata je gornja vremenska granica za a) izvođenje prijelaza nekog procesa i b) trajanje prijenosa poruke kanalom.

Zbog ovih je prepostavki moguće sinkronizirati procese tako da se sve promjene stanja izvode u koracima. Slika 6.8 prikazuje primjer sinkrone komunikacije između dva procesa. Može se uočiti da se komunikacija označava okomitim strelicama kao da se događa trenutno, dok je  $t_1$  vremenski period koji označava maksimalno trajanje prijenosa poruke u sustavu. Vremenski period  $t_2$  označava maksimalno trajanje prijelaza za procese u sustavu.

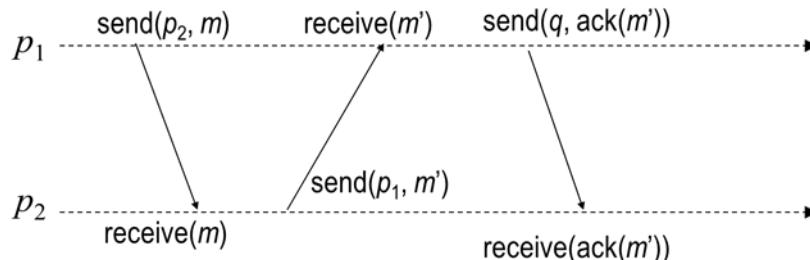


Slika 6.8. Primjer sinkrone komunikacije

Izvođenje algoritma u sinkronom sustavu organizirano je koracima. Inicijalno su svi procesi u početnom stanju i svi su kanali prazni. Nakon toga se izvode koraci. Korak se sastoji od dvije faze i izvodi se na svim procesima istovremeno. U prvoj fazi svaki proces generira poruke koje će biti poslane izlaznim susjedima i postavlja te poruke na izlazne kanale. U drugoj fazi svaki proces na temelju trenutnog stanja i primljenih poruka izvodi prijelaz (obrađuje primljene poruke i izvodi unutarnje događaje) te određuje sljedeće stanje procesa.

Prepostavke asinkronog modela su sljedeće:

- procesi **nemaju** sinkronizirana lokalna vremena,
- ne postoji gornja vremenska granica za izvođenje prijelaza nekog procesa, no trajanje prijelaza je uvijek konačno i
- ne postoji gornja vremenska granica za trajanje prijenosa poruke kanalom.



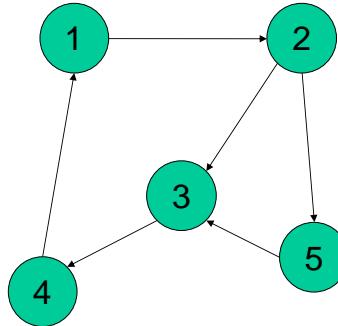
Slika 6.9. Primjer asinkrone komunikacije

Prepostavke asinkronog modela su bliske obilježjima stvarnih raspodijeljenih sustava i stoga se ovaj model češće razmatra, što znatno komplificira modeliranje i analizu raspodijeljenog sustava. Slika 6.9 prikazuje primjer asinkrone komunikacije između dva procesa  $p_1$  i  $p_2$ . Može se zamjetiti da trajanje komunikacije nije ograničeno i različito je za prijenos pojedinih poruka, a isto tako je i trajanje prijelaza vremenski neograničeno. Stoga je teško unaprijed predvidjeti odvijanje algoritma u ovom modelu jer je ono slučajno, tj. izvođenje događaja je slučajni proces. Valja naglasiti da se kanal može

modelirati kao pouzdani komunikacijski medij bez gubitaka ili kao nepouzdani komunikacijski medij kada je potrebno modelirati vjerojatnost gubitka poruke na kanalu

## 6.2 Sinkroni model

Sinkroni model raspodijeljenog sustava može se prikazati usmjerenim grafom  $G = (V, E)$  u kojem čvorovi iz skupa  $V$  ( $v_i \in V$ ) modeliraju procese, a usmjerene grane iz skupa  $E$  ( $e_j \in E$ ) modeliraju komunikacijske kanale među procesima. Procesi razmjenjuju poruke iz skupa poruka  $M$ , npr. poruka  $m_{ik}$  je poruka poslana od čvora  $v_i$  do čvora  $v_k$ . Ako na kanalu u nekom trenutku nema poruka to označavamo s *null*.



Slika 6.10. Primjer sinkronog modela sustava od pet procesa i šest kanala

Slika 6.10 prikazuje model sinkronog raspodijeljenog sustava koji se sastoji od pet procesa  $v_1, v_2, \dots, v_5$  i šest usmjerenih kanala, npr.  $e_{12}$  je usmjerena grana između procesa  $v_1$  i  $v_2$ . Stoga svaki proces ima skup svojih izlaznih susjeda  $out-nbrs_i$  i skup ulaznih susjeda  $in-nbrs_i$ . Npr. proces  $v_2$  ima 2 izlazna susjeda i  $out-nbrs_2 = \{v_3, v_5\}$ , a proces  $v_3$  ima 2 ulazna susjeda i stoga je  $in-nbrs_3 = \{v_2, v_5\}$ . Iz ovoga je vidljivo jedno važno svojstvo raspodijeljenih sustava, a to je *lokalnost*: svaki proces poznaje samo svoje neposredne susjede, ulazne i izlazne, ali niti jedan proces nema potpuno znanje o topologiji cjelokupnog sustava.

Postoje još dvije mjerne važne za analizu sinkronih modela:

- $distance(v_i, v_j)$  označava najkraći put između para čvorova  $v_i$  i  $v_j$ :  $v_i, v_j \in V$ , a
- $diameter(G)$  je najdulji od najkraćih putova grafa  $G$  i definira se kao  $\max distance(v_i, v_j)$  za sve parove  $(v_i, v_j)$  iz  $G$ .

**Model procesa.** Svaki se proces  $v_i \in V$  modelira kao uređena četvorka:  $(states_i, start_i, msgs_i, trans_i)$ . Skup  $states_i$  označava skup svih mogućih stanja procesa  $v_i$ , a  $start_i$  definira početno stanje procesa  $v_i$  i vrijedi  $start_i \subset states_i$ ,  $start_i \neq \emptyset$ . Drugim riječima svakom je procesu pridružen skup stanja u koja može doći tijekom izvođenja algoritama nad raspodijeljenim sustavom, a u skupu stanja značajno mjesto ima početno stanje procesa jer skup početnih stanja svih procesa čini početno stanje sustava iz kojega kreće izvođenje algoritma. Skup stanja jednoga procesa ne mora nužno biti konačan što omogućuje npr. modeliranje brojača.

Osim skupova  $states_i$  i  $start_i$  za proces  $v_i$  je značajna funkcija za generiranje poruka  $msgs_i$  koja određuje izlaznu poruku za svakog izlaznog susjeda procesa  $v_i$ , a na temelju njegovog trenutnog stanja. Izlazne poruke su iz skupa mogućih poruka  $M$  ili *null*. Stoga se funkciju za generiranje poruka definira nad skupom svih mogućih stanja procesa  $states_i$  i izlaznih susjeda na sljedeći način:  $msgs_i : states_i \times out-nbrs_i \rightarrow M \subset M \cup \{\text{null}\}$ . Posljednji element uređene četvorke za modeliranje procesa  $v_i$  je funkcija prijelaza  $trans_i$  koja za svako moguće stanje procesa i skup primljenih poruka od ulaznih susjeda definira sljedeće stanje procesa. Time se definiraju pravila za izvođenje procesa.

Slijedi pregledna definicija modela procesa  $v_i \in V$ :

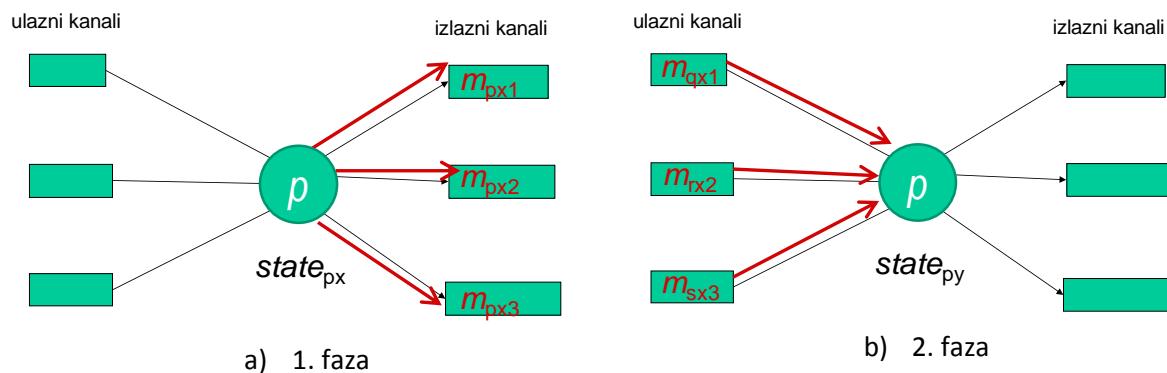
$v_i \in V$  se modelira kao uređena četvorka:  $(states_i, start_i, msgs_i, trans_i)$

- $states_i$  – skup mogućih stanja procesa
- $start_i$  – skup početnih stanja,  $start_i \subset states_i$ ,  $start_i \neq \emptyset$
- $msgs_i$  – funkcija za generiranje poruka koja određuje izlaznu poruku za svakog susjeda na temelju trenutnog stanja procesa, tj.  $msgs_i : states_i \times out-nbrs_i \rightarrow M_i \subset M \cup \{\text{null}\}$
- $trans_i$  – funkcija prijelaza, određuje sljedeće stanje na temelju trenutnog stanja i primljenih poruka od ulaznih susjeda

**Model kanala.** Model kanala raspodijeljenog sustava je značajno jednostavniji od modela procesa. Kanal modelira grana  $e_{ij}$  između para čvorova  $v_i$  i  $v_j$ :  $v_i, v_j \in V$ . Kanal može primiti poruku  $m$  iz definiranog skupa poruka  $M$  ili  $null$ , gdje  $null$  označava praznu poruku.

Algoritmi se izvode nad sinkronim modelom raspodijeljenog sustava u koracima. Inicijalno su svi procesi u početnom stanju definiranom skupom  $start_i$  za sve procese, a svi su kanali prazni. Nakon toga se izvode koraci. Korak se sastoji od 2 faza:

1. faza: Primijeni funkciju za generiranje poruka  $msgs_i$  za svaki proces  $v_i$ , a na temelju trenutnog stanja procesa. Funkcija generira poruke koje se postavljaju na izlazne kanale te će biti poslane izlaznim susjedima.
- 2 faza: Primijeni funkciju prijelaza  $trans_i$  za svaki proces  $v_i$  koja će na temelju trenutnog stanja i primljenih poruka na ulaznim kanalima odrediti sljedeće stanje procesa  $v_i$ . Briši sve poruke na kanalima.



Slika 6.11. Izvođenje algoritama za sinkroni model u koracima

Slika 6.11 prikazuje primjer izvođenja navedena dva koraka za proces  $p$ . Slika a) prikazuje 1. fazu izvođenja koraka kada proces  $p$  koji je u stanju  $state_{px}$  na temelju svoje funkcije za generiranje poruka  $msgs_i$  postavlja poruke na 3 izlazna kanala svojih susjeda. Istovremeno će i svi ulazni susjedi procesa  $p$  postaviti poruke na ulazne kanale procesa  $p$ . U 2. fazi prikazanoj slikom b) proces  $p$  primjenjuje funkciju prijelaza  $trans_i$ , koja na temelju trenutnog stanja  $state_{px}$  i 3 primljene poruke na ulaznim kanalima određuje njegovo sljedeće stanje. To je stanje  $state_{py}$ .

Kako bi se moglo analizirati ponašanje sinkronog modela raspodijeljenog sustava, potreban je formalni model njegova izvođenja. Formalni model izvođenja sinkronog modela može se opisati beskonačnim slijedom  $C_0, M_1, N_1, \dots, C_k, M_k, N_k, C_{k+1}, \dots$  gdje je  $C_k$  vektor svih stanja procesa nakon  $k$  koraka,  $M_k$  vektor poslanih poruka na svim kanalima tijekom koraka  $k$  i  $N_k$  vektor primljenih poruka na svim kanalima tijekom koraka  $k$ . U sustavima s pouzdanim kanalima  $M_k = N_k$  za svaki  $k$ , dok je  $M_k \neq N_k$  ako dođe do gubitka poruke na nekom kanalu.

Mjere složenosti algoritama za sinkrone sustave su sljedeće:

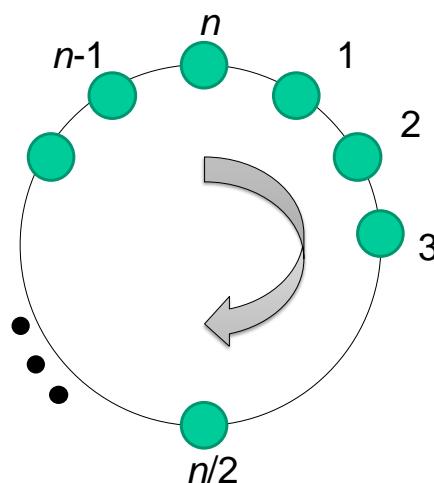
1. **vremenska složenost:** mjeri se brojem izvedenih koraka (engl. *rounds*,  $r$ ) koji dovodi do završnog stanja algoritma tj. do stanja u kome su svi procesi zaustavljeni ili kada se više ne proizvode novi izlazi.
2. **komunikacijska složenost:** mjeri se broj pripremljenih i poslanih poruka na kanalima.

Pri određivanju mjere složenosti uvijek se analizira najgori mogući scenarij izvođenja algoritma.

U nastavku se analiziraju dva primjera algoritama za sinkroni model prema (Lynch, 1996): odabir vođe u sinkronom prstenu i odabir vođe u usmjerenoj mreži.

### 1. primjer algoritma: Odabir vođe u sinkronom prstenu

Odabir vođe u sinkronom prstenu jedan je od osnovnih sinkronih algoritama koji omogućuje odabir jednog procesa za vođu prstena, dok svi ostali nisu vođe. Pretpostavka je da se raspodijeljeni sustav sastoji od  $n$  procesa koji su povezani u prstenastu mrežu kako je prikazano slikom (Slika 6.12), tj. svaki ima jednog prethodnika i jednog sljedbenika, dok procesi mogu razlučiti prethodnika od sljedbenika. Zahtjev algoritma je da na kraju njegovog izvođenja samo jedan proces promijeni svoj status u *leader*, dok svi ostali nemaju status *leader*. Dakle, u bilo kojem koraku samo jedan proces može postati vođa i promijeniti status u *leader*.



Slika 6.12. Sinkroni prsten od  $n$  procesa

Pretpostavka modela sinkronog prstena od  $n$  procesa je da su svi procesi jednaki osim što svaki ima jedinstveni identifikator (*unique ID*, UID). Uz to je identifikatore procesa moguće uspoređivati i odlučiti koji je od njih veći od drugoga. U suprotnom kada bi svi procesi bili potpuno jednaki bilo bi nemoguće odabrati vođu. Dodatna je pretpostavka da procesi mogu na slučajan način odabrati UID, te da svaki proces zna svog ulaznog i izlaznog susjeda. Broj procesa u prstenu ( $n$ ) može biti poznat ili nepoznat svim procesima.

Algoritam za odabir vođe u sinkronom prstenu temelji se na sljedećim pretpostavkama:

- procesi u prstenu koriste jednosmjernu komunikaciju (model sustava je usmjereni graf s komunikacijom 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 i
- algoritam za vođu izabire proces s najvećim UID.

Ideja izvođenja algoritma je sljedeća: Svaki proces initialno 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.

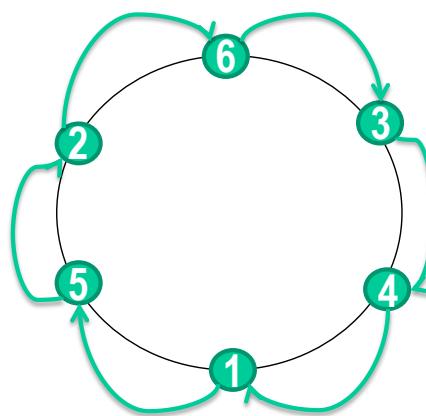
Formalni model algoritma za odabir vođe u sinkronom prstenu dan je u nastavku:

$M$  – skup poruka čini skup svih UID  
 Za svaki proces  $p_i$  definira se uređena četvorka ( $states_i$ ,  $start_i$ ,  $msgs_i$ ,  $trans_i$ )

1.  $states_i \in \{u, send, status\}$ 
  - $u$  – identifikator, initialno UID za  $p_i$
  - $send$  – identifikator ili null, initialno UID za  $p_i$
  - $status \in \{unknown, leader\}$ , initialno *unknown*
2.  $start_i = \{\text{UID procesa } p_i, \text{UID procesa } p_i, \text{unknown}\}$
3.  $msgs_i$  – poslati vrijednost varijable  $send$  sljedećem procesu
4.  $trans_i$  – prijelaz se izvodi na temelju sljedećeg pseudokoda
 

```
send := null (pobriši poruke na kanalima)
receive v
if v > u then send := v
if v = u then status := leader
if v < u then do nothing
```

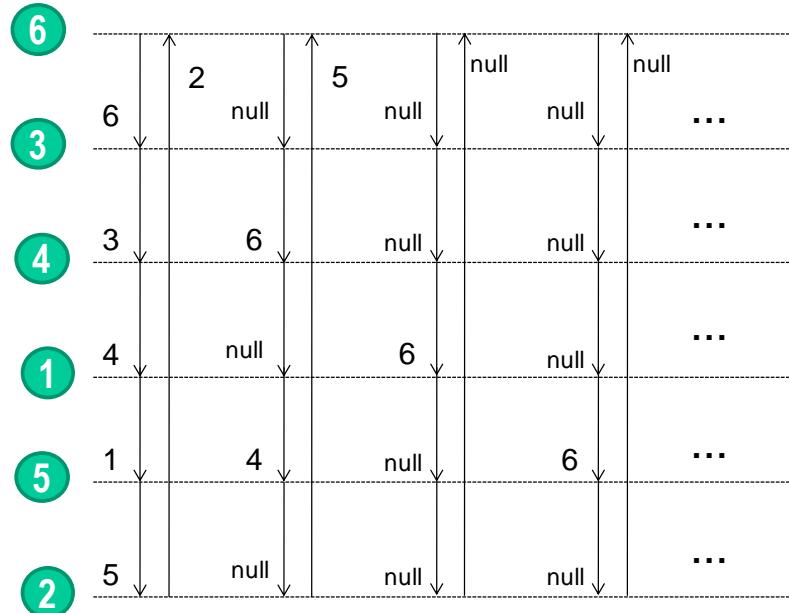
Na temelju formalnog modela može se uočiti da skup poruka u sustavu  $M$  čine jedinstveni identifikatori svih procesa UID. Stanje svakog procesa  $p_i$  modelira se pomoću 3 varijable  $\{u, send, status\}$ , gdje  $u$  pohranjuje identifikator procesa,  $send$  je varijabla koja se šalje sljedećem procesu, a  $status$  označava stanje procesa (*unknown* ili *leader*). Početno je stanje procesa  $p_i$   $\{\text{UID procesa } p_i, \text{UID procesa } p_i, \text{unknown}\}$ . Proces  $p_i$  u svakom koraku šalje svome sljedbeniku vrijednost iz svoje lokalne varijable  $send$ . Kada proces  $p_i$  primi poruku na ulaznom kanalu, ako je primljena vrijednost  $v$  veća od njegovog identifikatora spremlijenog u varijabli  $u$ , onda se primljeni UID, tj. vrijednost  $v$  upisuje u varijablu  $send$  i u sljedećem koraku šalje susjedu. Kada je primljena vrijednost  $v$  jednaka identifikatoru procesa, proces  $p_i$  sebe proglašava vođom jer je upravo njegov identifikator "preživio" sve usporedbe. U slučaju kada je primljena vrijednost  $v$  manja od lokalne varijable  $u$ , proces ne mijenja vrijednost varijable i ne šalje ništa svojim susjedima (na kanalu je *null*).



Slika 6.13. Primjer prstena od 6 procesa

Slika 6.13 prikazuje primjer prstena od 6 procesa radi ilustracije osnovnog algoritma za odabir vođe u prstenu. Broj unutar kružnice označava UID pojedinog procesa, a strelice smjer komunikacije među procesima. Slika 6.14 prikazuje primjer izvođenja algoritma za prsten od 6 procesa. U prvom koraku svaki proces šalje svoj UID susjednom procesu. U drugom koraku procesi susjedima šalju samo primljeni UID koji je veći od njihovog, npr. proces  $p_3$  svome susjedu  $p_4$  šalje poruku 6 jer je u

prethodnom koraku primio 6, a  $6 > 3$ . Na taj način poruka 6 nastavlja put po prstenu dok ne dođe na ulazni kanal procesa  $p_6$  koji kada je primi mijenja stanje u *leader*.



Slika 6.14. Primjer izvođenja koraka algoritma za odabir vođe u prstenu sa slike (Slika 6.13)

Kako bi se odredila vremenska složenost algoritma potrebno je odgovoriti na sljedeće pitanje: Koliko je koraka potrebno izvesti za odabir vođe u ovom prstenu? To je svakako 6 koraka koliko je potrebno da najveći UID bude prenesen svim kanalima te stigne do procesa s najvećim UID. Stoga je vremenska složenost algoritma  $O(n)$  jer nakon  $n$  koraka proces s najvećim UID mijenja svoj status u *leader*.

Komunikacijska složenost algoritma je  $O(n^2)$  jer tijekom svakog koraka u najgorem slučaju postoji  $n$  poruka na svakom kanalu.

Može se dokazati da će proces s najvećim UID promijeniti status u *leader* nakon  $n$  koraka s obzirom da algoritam završava u slučaju kada čvor s najvećim UID ponovo primi vlastitu poruku, a potrebno je  $n$  koraka da ta poruka stigne do vođe u prstenu s  $n$  čvorova. No samo će proces koji je vođa znati da je algoritam završen jer je primio poruku s vlastitim UID, ali ostali procesi nisu svjesni da je algoritam završen. Stoga vođa može poslati posebnu poruku (*halt*) po prstenu s obavijesti da je vođa izabran: U tom je slučaju za izvođenje algoritma potrebno  $2n$  koraka jer je  $n$  koraka potrebno za pronašak vođe i potom  $n$  koraka za slanje poruke zaustavljanja ostalim procesima koji nisu izabrani za vođu.

## 2. primjer algoritma: Odabir vođe u usmjerenoj mreži

Odabir vođe u usmjerenoj mreži razlikuje se od prethodnog algoritma po topologiji mreže  $G(V, E)$  koja modelira procese i komunikacijske kanale. Riječ je o povezanoj usmjerenoj mreži procesa, a svi parovi procesa su na konačnoj udaljenosti tj. postoji konačna vrijednost  $distance(v_i, v_j)$  za svaki par procesa. Svaki proces u sustavu ima jedinstveni identifikator UID, a u takvoj je mreži potrebno također izabrati vođu tako da samo jedan proces promijeni svoj status u *leader*, dok svi ostali nemaju status *leader*. Dakle, u bilo kojem koraku samo jedan proces može postati vođa i promijeniti status u *leader*.

Algoritam za odabir vođe u usmjerenom mreži temelji se na preplavljuvanju. Prepostavke algoritma su sljedeće:

- svaki proces ima jedinstveni identifikator UID iz skupa prirodnih brojeva, UID se procesu dodjeljuje na slučajan način i
- svaki proces zna  $diameter(G)$  mreže  $G$ .

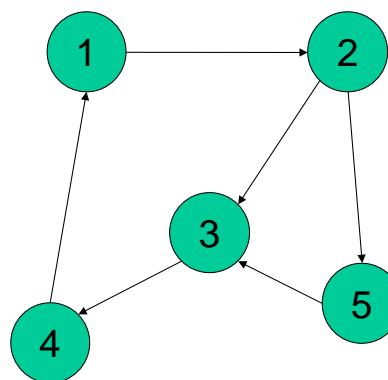
Ideja algoritma za odabir vođe u usmjerenoj mreži je sljedeća: Svaki proces bilježi primljeni UID s najvećom vrijednosti (inicijalno je to vlastiti UID). U svakom koraku proces šalje tu maksimalnu vrijednost na izlaznim kanalima svim susjedima. Nakon  $diameter(G)$  koraka ako je maksimalna vrijednost jednaka vlastitom UID proces se proglašava vođom, u suprotnom nije vođa.

Formalni model algoritma preplavljuvanja za odabir vođe u usmjerenoj mreži dan je u nastavku:

$M$  – skup poruka čini skup svih UID  
 Za svaki proces  $p_i$  definira se uređena četvorka ( $states_i$ ,  $start_i$ ,  $msgs_i$ ,  $trans_i$ )

5.  $states_i \in \{u, max-uid, status, rounds\}$ 
  - $u$  – identifikator, inicijalno UID za  $p_i$
  - $max-uid$  – najveći UID, inicijalno UID za  $p_i$
  - $status \in \{unknown, leader, non-leader\}$ , inicijalno *unknown*
  - $rounds$  – cijeli broj, inicijalno 0
6.  $start_i = \{\text{UID procesa } p_i, \text{UID procesa } p_i, \text{unknown}, 0\}$
7.  $msgs_i$  – poslati vrijednost varijable  $max-id$  svim susjedima ako je  $rounds < diameter$   
 if  $rounds < diameter$  then  
 send  $max-id$  to all  $j \in out-nbrs$
8.  $trans_i$  – prijelaz se izvodi na temelju sljedećeg pseudokoda  
 $rounds := rounds + 1$   
 receive set of UIDs  $U$  from neighbors  
 if  $rounds = diameter$  then  
 if  $max-uid = u$  then  $status := leader$   
 else  $status := non-leader$

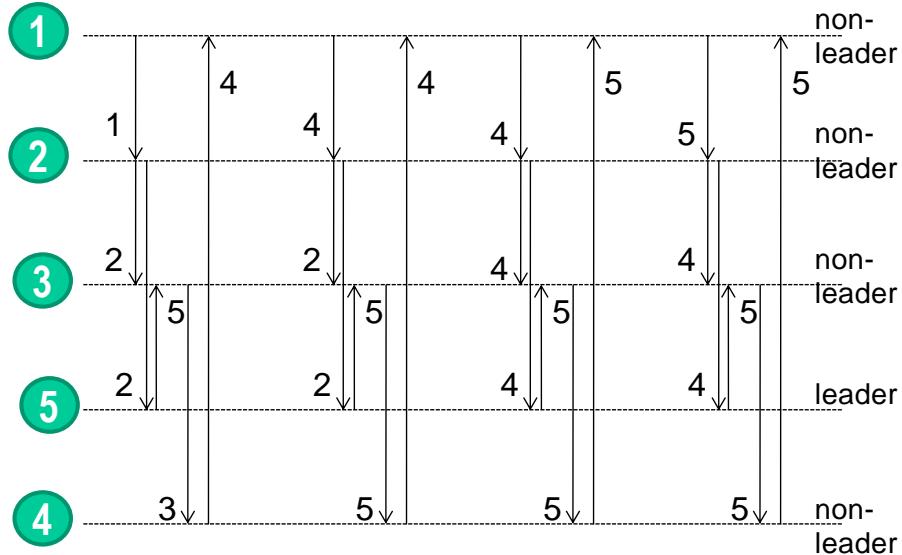
Na temelju formalnog modela može se uočiti da skup poruka u sustavu  $M$  čine jedinstveni identifikatori svih procesa UID. Stanje svakog procesa  $p_i$  modelira se pomoću 4 varijable  $\{u, max-uid, status, rounds\}$ , gdje varijabla  $u$  pohranjuje UID procesa, varijabla  $max-uid$  pohranjuje najveći viđeni UID, varijabla  $status$  pohranjuje izlaz algoritma, a varijabla  $rounds$  broj izvedenih koraka. Proces  $p_i$  u svakom koraku šalje svim svojim izlaznim susjedima vrijednost iz varijable  $max-id$  ako je broj koraka zapisan u varijabli  $rounds$  manji od  $diameter(G)$ . Nakon toga povećava se  $rounds$  za 1, čitaju se poruke s ulaznih kanala i za  $max-uid$  postavlja maksimalni primljeni UID ako je veći o trenutnog  $max-uid$ . Kada je zadovoljen uvjet da je broj koraka jednak dijametru grafa, ako je  $max-uid = u$ , tada se proces proglašava vođom, a u suprotnom nije vođa.



Slika 6.15. Primjer usmjerene mreže za odabir vođe

Slika 6.15 prikazuje primjer usmjerene mreže koja modelira sustav od 5 procesa i 6 usmjerenih komunikacijskih kanala među njima, a  $diameter(G)$  je 4 jer je  $distance(5, 2) = 4$ . Na slici (Slika 6.16) je prikazano izvođenje algoritma za odabir vođe u toj mreži. Vidljivo je da se algoritam izvodi u 4 koraka te da u prvom koraku svi procesi šalju vlastiti UID svim svojim susjedima, dok u sljedećim koracima

šalju najveći viđeni UID. Algoritam završava tako da se proces kojemu je najveći viđeni UID jednak njegovom proglašava vođom, dok ostali zaključuju da nisu vođa.



Slika 6.16. Primjer izvođenja algoritma preplavljanja radi odabira vođe u usmjerenoj mreži

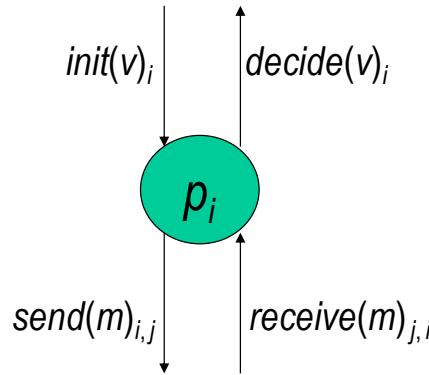
Vremenska složenost algoritma je određena vrijednošću  $diameter(G)$  jer algoritam završava nakon toliko koraka i svaki proces po završetku izvođenja zna svoj status (*leader* ili *non-leader*), ali ne i tko je izabran za vođu. Komunikacijska složenost određena je brojem poslanih poruka koja je umnožak dijametra mreže i broja usmjerenih grana grafa, tj.  $diameter(G) \times |E|$  jer se jedna poruka šalje na svaku granu u svakom koraku algoritma. Jednostavnom optimizacijom moguće je smanjiti broj poslanih poruka tako da proces šalje vrijednost varijable *max-uid* susjedima samo ako se vrijednost varijable *max-uid* promijeni.

### 6.3 Asinkroni model

Asinkroni model raspodijeljenog sustava može se kao i sinkroni model prikazati usmjerenim grafom  $G = (V, E)$  u kojemu čvorovi iz skupa  $V$  ( $v_i \in V$ ) modeliraju procese, a usmjerene grane iz skupa  $E$  ( $e_j \in E$ ) modeliraju komunikacijske kanale među procesima. Svaki proces  $p_i$  održava skup izlaznih susjeda  $out-nbrs_i$  i skup ulaznih susjeda  $in-nbrs_i$ . Procesi razmjenjuju poruke iz skupa poruka  $M$ , npr. poruka  $m_{ik}$  je poruka posljana od čvora  $v_i$  do čvora  $v_k$ . Ako na kanalu u nekom trenutku nema poruka to označavamo s *null*.

Razlike u odnosu na sinkroni model se očituju kroz asinkronost izvođenja akcija na procesima i komunikacije pomoću događaja čiji se vremenski trenutak ne može unaprijed odrediti, te se svaki proces i svaki kanal modeliraju ulazno-izlaznim automatom. Ulazno-izlazni (I/O) automat je automat stanja koji ima ulazne i izlazne događaje (modeliraju interakciju s okolinom) te unutarnje događaje. Prisjetimo se: za raspodijeljene sustave značajni su *ulazni*, *izlazni* ili *unutarnji* događaji. I/O automat modelira svaku komponentu raspodijeljenog sustava (proces ili kanal) i njenu interakciju s ostalim komponentama.

Slika 6.17 prikazuje model procesa kao primjer I/O automata. Automat je prikazan kao krug označen s  $p_i$ , ulazne strelice modeliraju ulazne događaje, a izlazne strelice izlazne događaje. Primitak ulazne vrijednosti za varijablu  $v$  označen je porukom  $init(v)_i$ , a automat proizvodi izlaz u obliku  $decide(v)_i$  koji modelira određenu odluku vezanu uz varijablu  $v$ . Poruke  $init(v)_i$  i  $decide(v)_i$  predstavljaju interakciju procesa  $p_i$  s okolinom. Proces  $p_i$  komunicira s ostalim procesima raspodijeljenog sustava pomoću 2 događaja:  $send(m)_{i,j}$  ( $p_i$  šalje poruku  $m$  procesu  $p_j$ ) i  $receive(m)_{j,i}$  ( $p_i$  prima poruku  $m$  od procesa  $p_j$ ).



Slika 6.17. Primjer modela procesa u asinkronom raspodijeljenom sustavu

Formalna definicija I/O automata dana je u nastavku:

I/O automat  $A$  se definira kao uređena četvorka  $(\text{sig}(A), \text{states}(A), \text{start}(A), \text{trans}(A))$  i sastoji se od sljedećih komponenti:

- $\text{sig}(A)$  je signatura automata  $A$ ,  $\text{sig}(A)=\{ \text{in}(A), \text{out}(A), \text{int}(A) \}$ , je skup koji definira ulazne, izlazne i unutarnje događaje automata  $A$
- $\text{states}(A)$  definira skup mogućih stanja automata
- $\text{start}(A)$  je skup početnih stanja automata,  $\text{start}(A) \neq \emptyset$  i podskup je skupa  $\text{states}(A)$
- $\text{trans}(A)$  je funkcija prijelaza koja za svako stanje  $s$  iz skupa  $\text{states}(A)$  i svaki ulazni događaj  $\pi$  iz skupa  $\text{in}(A)$  definira prijelaz  $(s, \pi, s') \in \text{trans}(A)$  gdje je  $s' \in \text{states}(A)$  novo stanje automata

Prilikom izvođenja automata izmjenjuju se stanja i događaji jer se prijelazi modeliraju parom stanja i događajem. Stoga se automat  $A$  izvodi kao konačan ili beskonačan slijed stanja i događaja, npr.  $s_0, \pi_0, s_1, \pi_1, s_2, \pi_2, s_3, \dots, \pi_k, s_k, \dots$  je primjer izvođenja nekog automata za koji je definiran prijelaz  $(s_k, \pi_k, s_{k+1}) \in \text{trans}(A)$  za svaki korak algoritma  $k \geq 0$ .

Slika 6.18 prikazuje model kanala FIFO kao I/O automata označenog s  $C_{i,j}$ . Signatura automata definirana je događajem primanja i slanja poruke iz skupa poruka  $M$ , dok kanal nema unutarnjih događaja, tj.  $\text{sig}(C_{i,j}) = (\text{send}(m)_{i,j}, \text{receive}(m)_{i,j}, 0)$ ,  $m \in M$ . Stanja kanala modeliraju se varijablom  $queue$ , koja je FIFO rep. Prijelazi se izvode na način da primitak poruke  $\text{send}(m)_{i,j}$  dodaje poruku  $m$  u varijablu  $queue$ , a slanje poruke događajem  $\text{receive}(m)_{i,j}$  briše poruku  $m$  iz  $queue$ . Preuvjet za izvođenje događaja  $\text{receive}(m)_{i,j}$  je postojanje poruke  $m$  na 1. mjestu u  $queue$ .



Slika 6.18. Primjer I/O automata kanala FIFO

Primjeri izvođenja I/O automata  $C_{i,j}$  su sljedeći (u uglatim zagradama su navedena stanja varijable  $queue$ ):

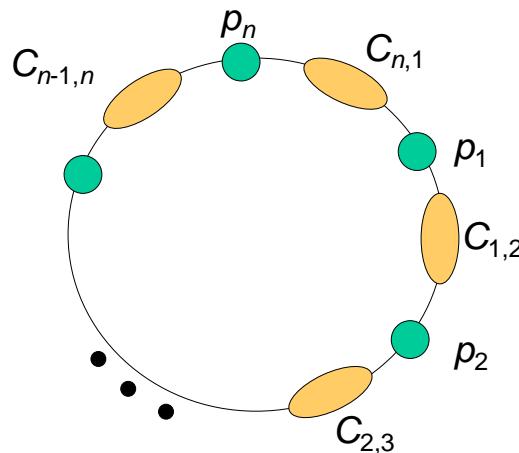
- [null],  $\text{send}(1)_{i,j}$ , [1],  $\text{receive}(1)_{i,j}$ , [null],  $\text{send}(2)_{i,j}$ , [2],  $\text{receive}(2)_{i,j}$ , [null]
- [null],  $\text{send}(1)_{i,j}$ , [1],  $\text{send}(1)_{i,j}$ , [11],  $\text{send}(1)_{i,j}$ , [111]...

Iz navedenih primjera se može zaključiti da ne postoji determinizam u izvođenju asinkronog modela, te je moguće definirati beskonačno mnogo primjera izvođenja za navedeni model FIFO kanala. Iz drugog primjera je vidljivo gomilanje poruka na kanalu jer I/O automat ne definira pravila za

izvođenje događaja slanja kao posljedice primanja poruke, te se i ulazni i izlazni događaji mogu zbiti u proizvoljnim vremenskim trenucima. U nastavku su dana dva primjera algoritama za asinkrone modele.

### 1. primjer algoritma: Odabir vođe u asinkronom prstenu

Analogno algoritmu za odabir vođe u sinkronom prstenu, i u ovom je algoritmu potrebno izabrati jedan i samo jedan proces za vođu prstena koji se sastoji od  $n$  procesa i  $n$  kanala, dok svi ostali procesi nisu vođe. Razlika u odnosu na algoritam u sinkronom prstenu je asinkroni model prstena koji se modelira pomoću dvije vrste I/O automata: jedan automat modelira proces, a drugi kanal kako je prikazano slikom (Slika 6.19). Pretpostavke algoritma su da je kanal modeliran kao pouzdani FIFO kanal i da svaki proces ima ulazni spremnik koji može primiti maksimalno  $n$  poruka (poruke se mogu gomilati zbog asinkronosti komunikacije).



Slika 6.19. Asinkroni prsten od  $n$  procesa i  $n$  kanala

Ideja algoritma za odabir vođe u sinkronom sustavu može se adaptirati za asinkroni sustav. I u ovom slučaju svaki proces odabire na slučajan način vlastiti jedinstveni identifikator UID te se proces s najvećim UID izabire za vođu. Osnovna je razlika u tome što svaki proces mora imati ulazni spremnik veličine  $n$  (označen sa *send*) zbog mogućnosti nagomilavanja poruka na kanalu.

Formalna definicija I/O automata procesa  $P_i$  dana je u nastavku:

Signatura je definirana sljedećim ulaznim i izlaznim događajima:

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

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

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

Funkcija prijelaza  $\text{trans}_i$ :

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

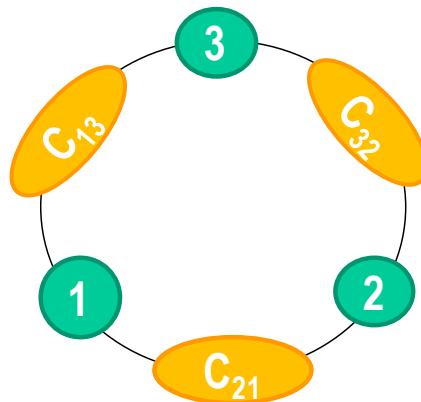
Signatura procesa  $p_i$  definirana je ulaznim događajem  $\text{receive}(v)_{i-1,i}$ , kada proces  $p_i$  prima  $v$  s ulaznog kanala, gdje je  $v$  UID, te izlaznim događajima  $\text{send}(v)_{i,i+1}$  i  $\text{leader}_i$  jer je proces  $p_i$  zadužen za dvije

operacije: za slanje UID procesu  $p_{i+1}$  putem kanala  $C_{i, i+1}$  i za objavu da postaje vođa. Proces  $p_i$  nema unutarnjih događaja. Skup mogućih stanja procesa opisan je pomoću tri varijable: varijabla  $u$  je zapravo konstanta koja sadrži UID za proces  $p_i$ , varijabla  $send$  je FIFO rep koji može primiti  $n$  UID-ova, a inicijalno sadrži samo UID za proces  $p_i$  i varijabla  $status$  može biti *unknown*, *chosen*, ili *reported*, a inicijalno se postavlja u *unknown*. Funkcija prijelaza može izvesti jedan od 3 sljedeća događaja:

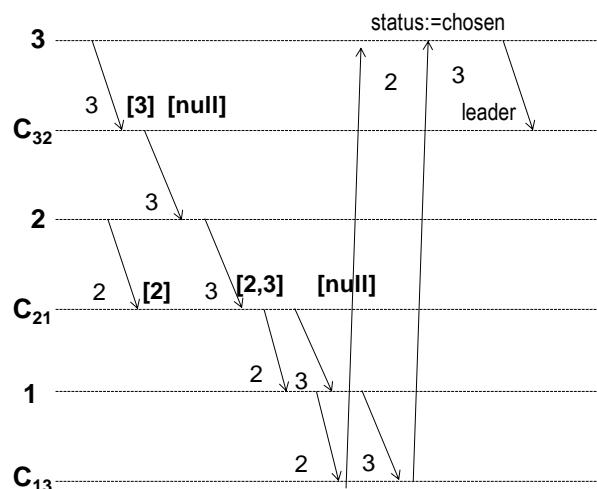
- izlazni događaj  $send(v)_{i,i+1}$  šalje  $v$  na izlazni kanal procesa  $p_i$ , a preduvjet izvođenja je da je  $v$  1. element iz FIFO repa  $send$  dok mu je posljedica brisanje  $v$  iz repa  $send$ ,
- izlazni događaj *leader*, postavlja poruku *leader*, na izlazni kanal procesa  $p_i$  uz preduvjet da je varijabla  $status$  na procesu  $p_i$  *chosen*, nakon čega se  $status$  mijenja u *reported* i
- ulazni događaj  $receive(v)_{i-1,i}$  prima  $v$  s ulaznog kanala procesa  $p_i$  te ako je vrijednost  $v$  veća od  $u$ , dodaje  $v$  u FIFO rep  $send$ , ako je  $v = u$  tada postavlja  $status$  u *chosen*, a inače ne radi ništa.

Formalna definicija I/O automata pouzdanog FIFO kanala  $C_{ij}$  je prethodno ilustrirana i objašnjena na slici (Slika 6.18) te je stoga nije potrebno ponavljati.

Slika 6.20 prikazuje primjer asinkronog prstena od 3 procesa i 3 kanala radi ilustracije algoritma za odabir vođe u asinkronom prstenu. Broj unutar zelenih kružnica označava UID pojedinog procesa, a kanali su usmjereni tako da npr. kanal  $C_{32}$  omogućuje prijenos poruka od procesa 3 do procesa 2.



Slika 6.20. Primjer asinkronog prstena od 3 procesa i 3 kanala



Slika 6.21. Primjer izvođenja algoritma za odabir vođe u asinkronom prstenu

Slika 6.21 prikazuje primjer izvođenja algoritma za prethodno navedeni prsten. Prvo se može uočiti da procesi proizvoljno odlučuju kada će poslati vlastiti UID na susjedni kanal (procesi 3 i 2). Nakon toga se poruka s kanala  $C_{32}$  proslijeđuje do procesa 2 koji je dalje proslijeđuje na kanal  $C_{21}$ . Sada su na

kanalu  $C_{21}$  dvije poruke [2, 3] koje se u tom slijedu isporučuju procesu 1 jer je riječ o FIFO kanalu. Proces 1 isporučuje prvo poruku 2, a potom 3 na kanal  $C_{12}$  koji u istom redoslijedu isporučuje poruke do procesa 3. U trenutku kada proces 3 primi poruku 3, on mijenja svoj status u *chosen* i na izlazni kanal postavlja poruku *leader*. Iz ovog je primjera vidljivo da postoji nebrojeno mnogo mogućih scenarija izvođenja navedenog algoritma te je nemoguće odrediti vremensku složenost algoritma. Komunikacijska službenost jednaka je složenosti sinkronog algoritma.

## 2. primjer algoritma: Kreiranje stabla u asinkronoj mreži

Jedan od vrlo važnih algoritama u asinkronom raspodijeljenom sustavu je algoritam za izgradnju stabla s definiranim korijenskim čvorom koji prekriva sve procese. Ovakvo se stablo može koristiti kao osnova za preplavljivanje svih procesa porukama iz korijenskog čvora.

Sljedeći algoritam (*AsynchSpanningTree*) kreira stablo pod pretpostavkom da svaki proces odabere prethodnika (*parent*), no to stablo neće nužno biti minimalno. Ostale pretpostavke algoritma su:

- mreža je modelirana grafom  $G(V, E)$  koji je usmjeren i povezan,
- kreira se stablo s definiranim korijenskim procesom  $i_0$  i
- procesi ne znaju dijametar mreže.

Ideja algoritma *AsynchSpanningTree* je sljedeća: Inicialno je proces  $i_0$  označen te šalje poruku *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 poruku *search* svim svojim susjedima.

Formalna definicija I/O automata procesa  $p_i$  dana je u nastavku:

Signatura je definirana sljedećim ulaznim i izlaznim događajima:

- ◆ input:  $receive("search")_{j,i}, j \in nbrs$
- ◆ output:  $send("search")_{i,j}, j \in nbrs; parent(j)_i, j \in nbrs$

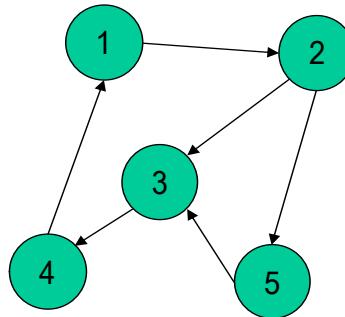
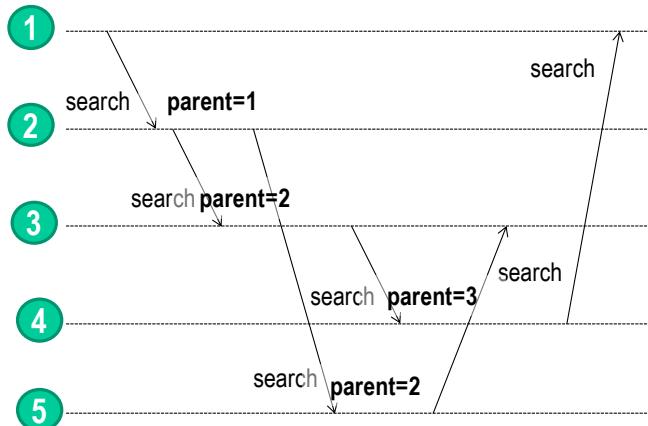
Skup mogućih stanja  $states_i = \{parent, reported, send\}$

- ◆  $parent \in nbrs \cup \{null\}$ , 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

Funkcija prijelaza *trans*:

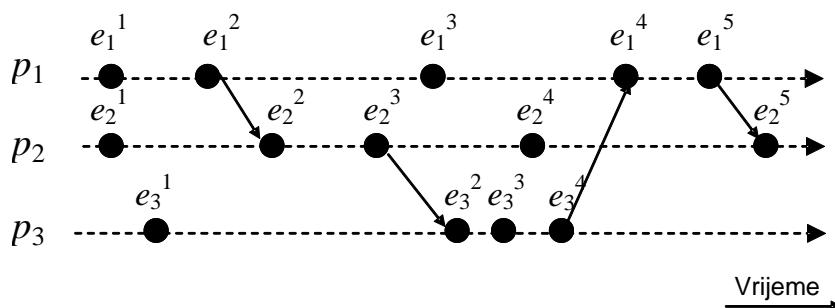
- ◆  $send("search")_{i,j}$  – preduvjet:  $send(j) = search$ , posljedica:  $send(j) := null$
- ◆  $parent(j)_i$  – preduvjet:  $parent = j$ ,  $chosen = false$ , posljedica:  $reported := true$
- ◆  $receive("search")_{j,i}$ 
  - if  $i \neq i_0$  and  $parent = null$   
 $parent := j$
  - for all  $k \in nbrs \setminus \{j\}$   
 $send(k) := search$

Slika 6.22 prikazuje primjer mreže procesa s korijenskim čvorom  $i_0 = 1$  za koji je potrebno izgraditi stablo uz pretpostavku FIFO kanala koje radi pojednostavljenja ovdje posebno ne razmatramo. Slika 6.23 daje primjer jednog izvođenja algoritma za navedenu mrežu. U prvom koraku korijenski proces šalje poruku *search* svome susjedu koji taj proces definira za svoj roditeljski proces. Nakon toga čvor 2 šalje poruku *search* do svojih susjeda (procesi 3 i 5) koji njega odabiru za roditeljski proces te nastavljaju slati poruke *search* svojim susjedima. Proces 3 će ponovno primiti poruku *search* od procesa 5, ali će je ignorirati jer je već odabrao roditeljski proces. Na ovaj način nastaje stablo koje pokriva sve čvorove grafa, a počinje u korijenskom čvoru 1.

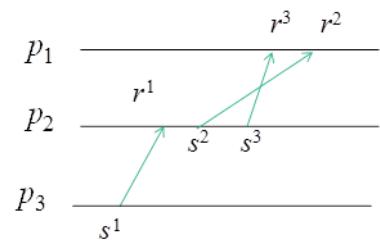
Slika 6.22. Primjer mreže procesa s korijenskim čvorom  $i_0 = 1$ Slika 6.23. Primjer izvođenja algoritma *AsynchSpanningTree*

#### 6.4 Pitanja za učenje i ponavljanje

- 6.1. Za koje je svojstvo raspodijeljenih sustava značajna komunikacijska složenost algoritama? Zašto?
- 6.2. Na temelju primjera procesa sa slike objasnite jesu li sljedeći parovi događaja uzročno povezani ili nisu? a) e13 i e22 i b) e22 i e15.



- 6.3. Objasnite model komunikacijskog kanala koji se temelji na uzročnoj slijednosti.
- 6.4. Objasnite model komunikacijskog kanala koji se temelji na uzročnoj slijednosti. Vrijedi li za sljedeći primjer CO ili non-CO i zašto?



## 7 SINKRONIZACIJA PROCESA U VREMENU

Sinkronizacija procesa u vremenu označava koordinaciju njihova izvođenja i međudjelovanja u svrhu skladnog rada cjelokupnog sustava tijekom vremena. Usklađivanje izvođenja procesa u vremenu je iznimno važno za raspodijeljene sustave jer njihova neusklađenost može uzrokovati djelomični ili čak potpuni ispad sustava. Na primjer, kod lokalnih računalnih mreža vrlo je čest slučaj da nekoliko klijenata rabi isti fizički (žični ili bežični) medij za međusobnu komunikaciju. Ukoliko dva ili više klijenata istovremeno pristupi takvom dijeljenom mediju dogodit će se pogreška u komunikacijskoj kolizija. Za uspješno odvijanje komunikacije dijeljenim medijem potrebno je sinkronizirati klijente čime se omogućuje uspješno razrješenje kolizija te sprječava njihovo opetovano pojavljivanje<sup>36</sup>.

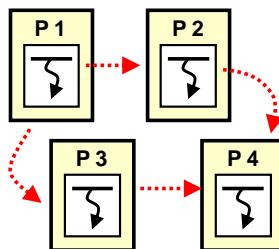
Za ispravni rad nekog raspodijeljenog sustava neophodno je osigurati usklađeno djelovanje njegovih raspodijeljenih procesa. Četiri osnovna razloga za primjenu sinkronizacije procesa u raspodijeljenoj okolini su:

1. uporaba dijeljenih sredstava,
2. usuglašavanje vremenskog redoslijeda akcija,
3. nadgledanje i upravljanje zadaćama skupa procesa te
4. uspostava suradnje skupa procesa.

Slično kao što je u lokalnim mrežama potrebno izbjegići istovremeni pristup dijeljenom mediju, u raspodijeljenoj okolini je potrebno sprječiti istovremeni pristup dijeljenom sredstvu, na primjer uredskom pisaču ili datoteci na mrežnom disku. Bez koordinacije slanja dokumenata koji se ispisuju, na pisaču bi moglo doći do odbacivanja nekog dokumenta zbog nemogućnosti pohrane u lokalnu memoriju pisača. U slučaju dijeljene datoteke, ukoliko bi dva ili više korisnika istovremeno upisivala sadržaj u datoteku moglo bi doći do narušavanja njene konzistentnosti.

U raspodijeljenoj okolini se često javlja potreba za usuglašavanjem vremenskog redoslijeda izvođenja akcija jer u suprotnom može doći do javljanja raznovrsnih pogrešaka. Primjer je raspodijeljeni razvoj programskog proizvoda pri kojem je potrebno usuglasiti točan vremenski redoslijed izmjena koda jer u suprotnom može doći do prepisivanja novije inačice koda starijom.

U raspodijeljenim sustavima procesi najčešće nisu međusobno ravnopravni, već se često zahtijeva da jedan proces bude zadužen za upravljanje i nadgledanje izvođenja zadaća preostalih procesa, tj. da im bude nadređen. U prethodnom poglavlju je kao primjer raspodijeljenog algoritma obrađen algoritam odabira vođe u sinkronom prstenu. Kako se ovaj algoritam izvodi u koracima, za uspješan odabir vođe je neophodno da procesi budu međusobno sinkronizirani.



Slika 7.1. Primjer uspostave suradnje skupa procesa

Mnoge raspodijeljene aplikacije zahtijevaju vremenski i prostorno usklađenu suradnju skupa procesa s ciljem ostvarivanja raspodijeljenog tijeka njihova izvođenja. Slika 7.1 prikazuje jedan jednostavni primjer uspostave suradnje četiri procesa. Primjerice, prvo početni proces P1 dohvata ulazne podatke i ostvara njihovu pripremu za obradu. Nakon pripreme, procesi P2 i P3 nezavisno i

<sup>36</sup> Za to je zadužen sloj podatkovne poveznice (DLL - Data Link Layer), preciznije njegov dio za kontrolu pristupa mediju (MAC - Media Access Control).

istovremeno obrađuju odvojene skupove podataka. Konačno, proces P4 prikuplja rezultate obrade i prosljeđuje ih u izlazni spremnik. Drugim riječima, u ovom slučaju se zahtijeva da procesi budu međusobni sinkronizirani na način da se procesi P2 i P3 počnu izvoditi nakon završetka izvođenja procesa P1, a da se proces P4 počne izvoditi tek nakon završetka izvođenja procesa P2 i P3.

## 7.1 Primjena sata u raspodijeljenoj okolini

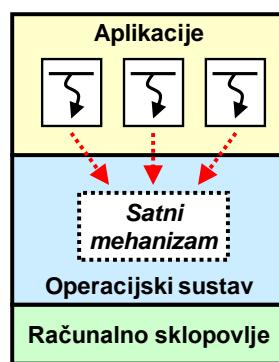
Sva današnja računala imaju vlastite satne mehanizme. Kao i kod većine ručnih satova, oni su u praksi najčešće izvedeni uporabom kristala kvarca koji pod naponom osciliraju zbog piezoelektričnog efekta. Međutim, frekvencija osciliranja kvarcnog oscilatora ovisi o njegovoj izvedbi i temperaturi njegove okoline te zbog toga u praksi nije idealno stabilna, već uvijek malo varira oko svoje srednje vrijednosti. S vremenom se ova mala varijacija frekvencije akumulira u obliku pogreške odstupanja satnog mehanizma (od nekoliko desetina sekundi na godinu dana) te se zbog toga npr. ručni kvarcni mehanizmi moraju s vremena na vrijeme ponovo podešavati da bi pokazivali točno vrijeme.

Zbog navedenih nedostataka praktične implementacije satnih mehanizama, u raspodijeljenim sustavima se javljaju sljedeći problemi:

1. satovi imaju različita odstupanja,
2. vrijednosti satova nisu usklađene i
3. satovi imaju različiti takt.

Zbog akumulirane pogreške odstupanja tijekom vremena, vrijednosti satova u raspodijeljenoj okolini međusobno nisu usklađene. Primjerice, jedan satni mehanizam može kasniti 1 milisekundu na dan, dok drugi može kasniti 1 mikrosekundu u 2 dana. Zbog toga je satne mehanizme u računalima, kao što je slučaj i s ručnim kvarnim satovima, potrebno s vremena na vrijeme međusobno usklađivati. Pri tome dodatni problem predstavlja i to što satni mehanizmi različitih proizvođača imaju različite taktove. Primjerice, jedan satni mehanizam može koristiti mikrosekundu kao osnovu vremensku jedinicu dok kod drugog to može biti nanosekunda.

U centraliziranim jednoprocesorskim sustavima svi procesi vide isto vrijeme jer im je omogućen pristup središnjem satnom mehanizmu računala domaćina putem usluga operacijskog sustava, kao što je prikazano na slici (Slika 7.2). Osim toga svi procesi su pod utjecajem središnjeg satnog mehanizma što ovu okolinu čini čvrsto povezanom i zatvorenom te omogućava puno jednostavnije usklađivanje procesa nego u raspodijeljenoj okolini.



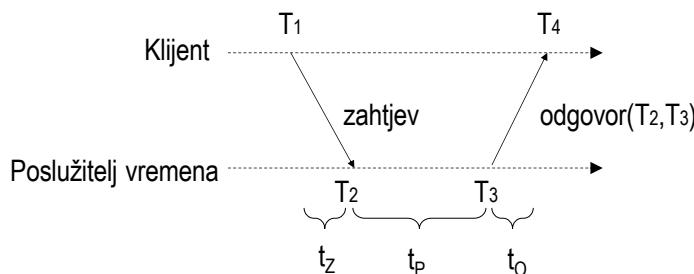
Slika 7.2. Određivanje točnog vremena u centraliziranom sustavu

U raspodijeljenoj okolini se koriste dva pristupa usklađivanju satnih mehanizama, a to su fizički i logički satovi. Kod usklađivanja fizičkih satova, satni mehanizmi različitih računala se pokušavaju što točnije uskladiti. Pri tome dogovoren vrijeme može, ali ne mora nužno, odgovarati stvarnom vremenu. Međutim, kako su satni mehanizmi različitih procesa potpuno nezavisni, primjenom isključivo fizičkih satnih mehanizama nije moguće odrediti točan redoslijed događaja u vremenu. Iz prethodnog razmatranja je poznato da je u raspodijeljenim sustavima većinom potrebno poznavati točan redoslijed izvođenja događaja, a ne točne vremenske trenutke njihova javljanja. Ova činjenica

se koristi pri usklađivanju logičkih satova kod kojih se namjerno zanemaruje usklađenost satnih mehanizama računala sa stvarnim vremenom nauštrb postizanja usuglašenosti oko točnog vremenskog redoslijeda događaja. Dodatno, bitno je uočiti da ukoliko između dva procesa u raspodijeljenoj okolini ne postoji nikakva interakcija, tada nije niti potrebno uskladiti njihove satove.

### 7.1.1 Usklađivanje fizičkih satova

Dva najčešće korištena algoritma za usklađivanje fizičkih satova su Cristianov algoritam koji je 1989. godine razvio F. Cristian i algoritam Berkeley kojeg su iste godine razvili R. Gusella i S. Zatti sa Sveučilišta Berkeley u Kaliforniji. Prvim algoritmom se satni mehanizmi usklađuju sa stvarnim vremenom dok se u drugom slučaju oni samo međusobno usklađuju, ali dogovoren vrijeme ne mora odgovarati stvarnom vremenu. Pri tome se kod algoritma Berkley koristi činjenica da ako neki skup računala ne komunicira sa vanjskim svijetom, već samo međusobno, sasvim je dovoljno da su njihovi satovi međusobno usklađeni pa makar to vrijeme odstupalo od točnog stvarnog vremena.



Slika 7.3. Izvođenje Cristianovog algoritma

**Cristianov algoritam** se temelji na primjeni poslužitelja s točnim vremenom od kojega klijenti po potrebi dohvaćaju informacije o stvarnom vremenu. Pri tome se prepostavlja da poslužitelj zna točno stvarno vrijeme jer ima pristup vrlo točnom satnom mehanizmu koji usklađuje svoje vrijeme putem signala sa zemaljskih stanica UTC (*Universal Time, Coordinated*) ili sa satelita sustava GPS (*Global Positioning System*). Kao što je prikazano na slici (Slika 7.3), koraci Cristinovog algoritma su sljedeći:

1. Korisnik šalje zahtjev poslužitelju. Zahtjev putuje  $t_z$  vremenskih jedinica do poslužitelja.
2. Poslužitelj prima zahtjev te unutar sljedećih  $t_p$  vremenskih jedinica obrađuje primljeni zahtjev i šalje odgovor klijentu.
3. Odgovor putuje  $t_o$  vremenskih jedinica do klijenta, a sadrži točno vrijeme  $T_2$  na poslužitelju u trenutku primanja zahtjeva i točno vrijeme  $T_3$  na poslužitelju u trenutku slanja odgovora.
4. U zadnjem koraku klijent prima odgovor te na osnovi vremenskih trenutaka koje je primio u poruci i izmjerениh vremenskih trenutaka  $T_1$  i  $T_4$  pomiče svoje lokalno vrijeme za  $\theta = T_3 + t_o - T_4 \approx T_3 + \frac{t_z + t_o}{2} - T_4 = T_3 + \frac{(T_4 - T_1) - (T_3 - T_2)}{2} - T_4 = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}$ .

Bitno je naglasiti da klijent ne može imati sasvim usklađeno vrijeme s poslužiteljem vremena iz dva razloga. Prvi razlog je to što putovanje poruke kroz mrežu zahtjeva  $t_z$  vremenskih jedinica u jednom smjeru i  $t_o$  u drugom, a  $t_o$  je u gornjoj formuli aproksimiran sa  $(t_z + t_o)/2$ , a drugi razlog je razlika u odstupanjima klijentskog i poslužiteljskog sata koja uzrokuje odstupanje izmjerene duljine intervala  $T_4 - T_1$  od njegove stvarne vrijednosti. Cristianovim algoritmom se može postići sinkronizacija satnih mehanizama jedino u slučaju kada je vrijeme povratka (*Round Trip Time*)  $RTT = T_4 - T_1$  puno manje od preciznosti koja se želi postići<sup>37</sup>.

Glavni nedostatak Cristianovog algoritma je postojanje jedinstvene točke ispada– poslužitelja vremena. Neispravan rad poslužitelja vremena ili njegova satnog mehanizma može u ovom slučaju

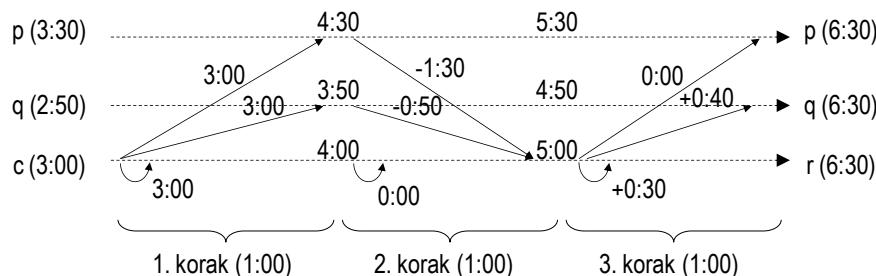
<sup>37</sup> Npr., RTT u LAN-u je od 1 do 10 milisekundi.

uzrokovati fatalne pogreške u radu raspodijeljenog sustava. Algoritam Berkeley, koji je predstavljen u nastavku, dijelom rješava probleme s neispravnim satnim mehanizmima.

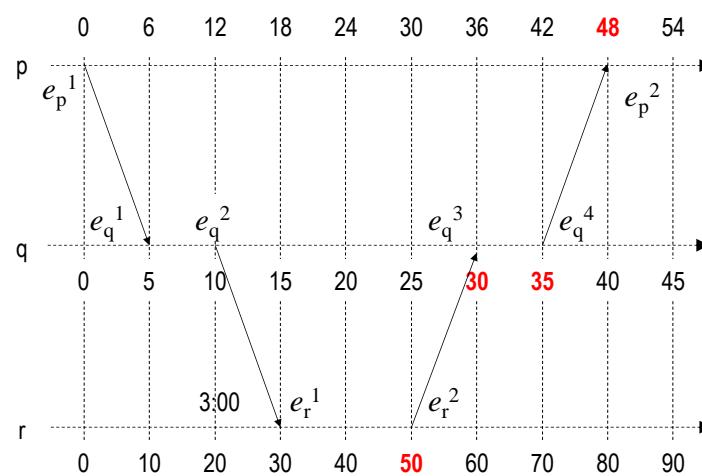
**Algoritam Berkeley** je zasnovan na primjeni upravitelja vremena koji periodički odašilje točno vrijeme skupu računala u raspodijeljenoj okolini. Pri tome je točno vrijeme ne mora odgovarati stvarnom vremenu nego je samo važno da je skupina procesa međusobno sinkronizirana. Na slici (Slika 7.4) su prikazani koraci algoritma Berkeley:

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

Pomak može biti izračunat na proizvoljan način jer je satne mehanizme moguće pomaknuti i unaprijed i unazad. Zbog izbjegavanja komplikacija koje se događaju prilikom pomicanja satnih mehanizama unazad čime budući događaji imaju manju vremensku oznaku od prethodnih, pomak se najčešće računa tako da se svi satni mehanizmi pomaknu na vrijednost koju ima sat s najvećom vrijednosti. Pri tome je bitno naglasiti da upravitelj, slično poslužitelju kod Cristianovog algoritma, aproksimira vrijeme putovanja zahtjeva  $t_z$  sa  $(t_z + t_o)/2$  te zbog toga klijenti ne mogu imati sasvim usklajeno vrijeme. Kako bi se riješio problem s neispravnim satnim mehanizmima, upravitelj može ignorirati vremena satova s najvećim odstupanjima prilikom određivanja točnog vremena i odgovarajućih pomaka satova.



Slika 7.4. Primjer uporabe algoritma Berkeley



Slika 7.5. Primjer nedostatka fizičkog sata

Glavni nedostatak uporabe fizičkih satova u raspodijeljenoj okolini je nemogućnost određivanja točnih vremenskih odnosa. Primjer na slici (Slika 7.5) pokazuje ovaj nedostatak u praksi. Tri procesa ( $p, q, r$ ) koja imaju svoje vlastite fizičke satove međusobno razmjenjuju poruke. Za događaje slanja i primanja prve dvije poruke ( $p \rightarrow q$  i  $q \rightarrow r$ ) vrijedi da je događaj slanja nastupio prije događaja primanja poruke. Međutim, trenutak slanja treće poruke ( $e_r^2$ ) s procesa  $r$  veći je od trenutka primanja iste poruke na procesu  $q$  ( $e_q^3$ ). Slično vrijedi i za događaje slanja i primanja četvrte poruke ( $e_q^4, e_p^2$ ). Zbog navedenih okolnosti, narušena je konzistentnost globalnog tijeka vremena i redoslijeda izvođenja akcija u vremenu kako ih doživljavaju procesi u raspodijeljenoj okolini.

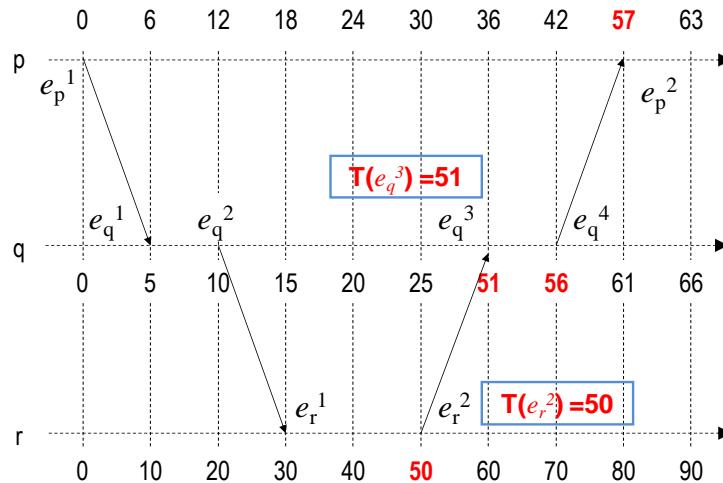
### 7.1.2 Usklađivanje logičkih satova

Kako bi se riješio navedeni nedostatak fizičkih satova, u raspodijeljene sustave se uvode logičke oznake vremena. Logički sat je funkcija  $T(e) = t_e$  koja vrši mapiranje sa skupa događaja na skup logičkih oznaka vremena pri čemu je zadovoljeno *svojstvo konzistentnosti sata* tako da za neka dva događaja  $e_i$  i  $e_j$  vrijedi:  $e_i \rightarrow e_j \Rightarrow T(e_i) < T(e_j)$ . Drugim riječima, ako događaj  $e_i$  prethodi događaju  $e_j$  tada vrijednost logičke oznake vremena  $T(e_i)$  događaja  $e_i$  mora biti manja od vrijednosti logičke oznake vremena  $T(e_j)$  događaja  $e_j$ . Ukoliko vrijedi i obratno, tj.  $e_i \rightarrow e_j \Leftrightarrow T(e_i) < T(e_j)$ , tada je zadovoljeno *svojstvo stroge konzistentnosti sata*. U praksi se najčešće koriste dvije vrste logičkih oznaka vremena, a to su skalarne i vektorske oznake vremena. Skalarne oznake vremena posjeduju svojstvo konzistentnosti sata, a vektorske svojstvo stroge konzistentnosti sata.

Uporabu skalarnih oznaka vremena je 1978. godine predložio L. Lamport. Skalarne oznake vremena su prirodni brojevi pri čijem se određivanju moraju poštivati tri pravila:

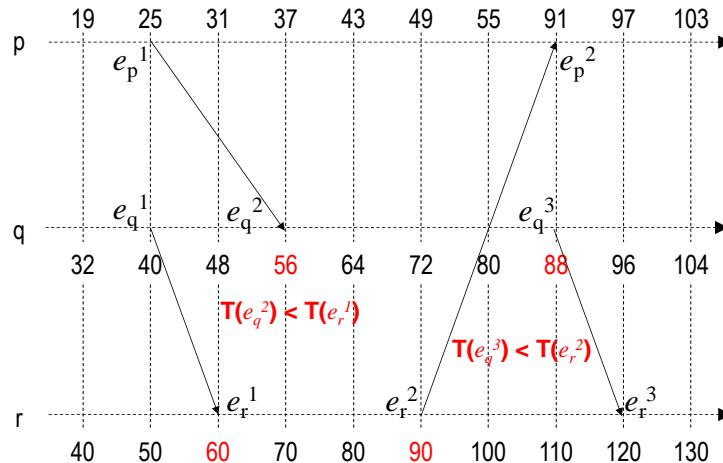
1. Prije izvođenja nekog događaja  $e_p^i$ , proces  $p$  treba odrediti vrijednost njegove skalarne oznake vremena  $T(e_p^i)$ . Ova oznaka se dobiva na način da se prethodnoj skalarnej oznaci vremena pribroji cijelobrojna konstanta  $d$ .
2. Prilikom slanja poruke u raspodijeljenom sustavu (događaj slanja  $e_r^j$ ), proces  $r$  koji ju šalje pridjeljuje joj skalarnu oznaku vremena  $T(e_r^j)$  koju je prethodno odredio.
3. Prilikom primanja poruke u raspodijeljenom sustavu (događaj primanja  $e_q^k$ ), proces  $q$  koji ju prima određuje njenu skalaru oznaku vremena na način  $T'(e_q^k) = \max[T(e_r^j) + 1, T(e_q^k)]$ , gdje je  $T(e_q^k)$  skalarna oznaka vremena događaja  $e_q^k$  koju je unaprijed odredio proces  $q$ , a  $T'(e_q^k)$  je ažurirana skalarna oznaka vremena događaja  $e_q^k$ .

Pomicanjem skalarnih oznaka vremena na veće vrijednosti postiže se poštivanje svojstva konzistentnosti globalnog sata. Na slici (Slika 7.6) to je prikazano u praksi pomoću slijeda događaja koji je identičan onome na slici (Slika 7.5). Procesi  $p, q$  i  $r$  imaju različite vrijednosti konstante  $d$ , a redom su to vrijednosti 6, 5 i 10. Prilikom primanja prve dvije poruke ne pomiče se vrijednost skalarnih oznaka vremena događaja primanja. Nasuprot tome, prilikom primanja druge dvije poruke vrijednost skalarnih oznaka vremena se mora pomaknuti na veću vrijednost radi očuvanja globalnog tijeka vremena.



Slika 7.6. Primjer uporabe skalarnih oznaka vremena

Glavni nedostatak skalarnih oznaka vremena je to što one ne zadovoljavaju svojstvo stroge konzistentnosti sata. To je u praksi prikazano na jednostavnom primjeru na slici (Slika 7.7). Usporedbom vrijednosti skalarnih oznaka vremena nekih događaja moguće je donijeti pogrešne zaključke o njihovom redoslijedu u vremenu. Primjerice, ako se usporedi vremenska oznaka  $T(e_q^2) = 56$  događaja primitka poruke na procesu q s vremenskom oznakom  $T(e_r^1) = 60$  događaja primitka poruke na procesu r, može se pogrešno zaključiti da je prvi događaj nastupio prije drugog događaja ( $56 < 60$ ). Slično bi se moglo zaključiti za događaje slanja poruke na procesima q i r jer je  $T(e_r^2) = 90 > T(e_q^3) = 88$ .



Slika 7.7. Primjer nedostatka skalarnih oznaka vremena

**Vektorske oznake vremena** su uvedene da bi se uklonio nedostatak skalarnih oznaka. Neovisno su ih krajem 80-ih godina prošlog stoljeća razvili i počeli primjenjivati C. Fidge, F. Mattern i F. Schmuck. Vektorska oznaka je vektor (polje) cijelih brojeva u kojem i-ta komponenta čuva redni broj zadnjeg poznatog događaja na i-tom procesu. Skup procesa tijekom rada međusobno razmjenjuje vektorske oznake kako bi razmijenili informacije o broju događaja koje procesi izvode tijekom rada. Pri određivanju vektorskih oznaka vremena moraju se poštivati sljedeća tri pravila:

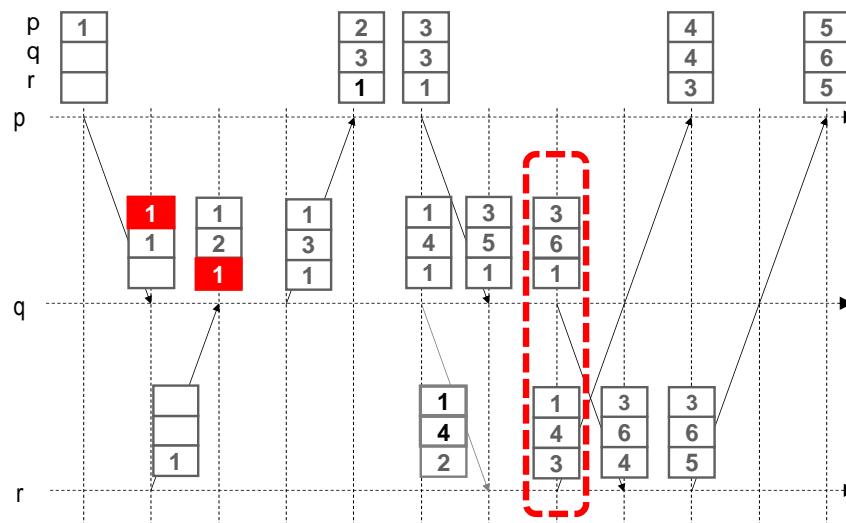
- Prije izvođenja nekog događaja  $e_{p_x}^i$ , proces  $p_x$  treba odrediti njegov redni broj  $i$  te ga zapisati kao x-tu komponentu vektorske oznake vremena  $\vec{T}(e_{p_x}^i)$ :  $\vec{T}(e_{p_x}^i)[x] = i$ .

2. Prilikom slanja poruke u raspodijeljenom sustavu (događaj slanja  $e_{p_y}^j$ ), proces  $p_y$  koji ju šalje pridjeljuje joj prethodno određenu vektorsku oznaku vremena  $\vec{T}(e_{p_y}^j)$ .
3. Prilikom primanja poruke u raspodijeljenom sustavu (događaj primanja  $e_{p_z}^k$ ), proces  $p_z$  koji ju prima određuje z-tu komponentu njene vektorske oznake vremena kao  $\vec{T}'(e_{p_z}^k)[z] = \max\{\vec{T}(e_{p_y}^j)[z] + 1, k\}$ , a preostale komponente  $\forall i \neq z$  ove vektorske oznake vremena određuje na sljedeći način:  $\vec{T}'(e_{p_z}^k)[i] = \max\{\vec{T}(e_{p_y}^j)[i], \vec{T}(e_{p_z}^k)[i]\}$ . Pri tome je  $\vec{T}(e_{p_y}^j)$  vektorska oznaka vremena događaja  $e_{p_y}^j$  koju je unaprijed odredio proces  $p_y$  i pridjelio je poruci prilikom slanja,  $\vec{T}(e_{p_z}^k)$  je vektorska oznaka vremena koju je unaprijed odredio proces  $p_z$ , a  $\vec{T}'(e_{p_z}^k)$  je ažurirana vektorska oznaka vremena koje se na kraju stvarno pridjeljuje događaju  $e_{p_z}^k$ .

Bitno je zamijetiti da se vektorske oznake vremena pridjeljuju isključivo novim događajima u sustavu dok se skalarne oznake vremena mijenjaju na procesima i u slučajevima kada nema novih događaja.

Za razliku od skalarnih oznaka, primjena vektorskih oznaka omogućava donošenje ispravnih zaključaka o vremenskom slijedu događaja u raspodijeljenoj okolini izravnom usporedbom vrijednosti vektorskih oznaka. Primjerice, ako za događaj  $a$  s vektorskog oznakom  $\vec{T}(a)$  te za događaj  $b$  s vektorskog oznakom  $\vec{T}(b)$  vrijedi  $\vec{T}(a) < \vec{T}(b)$  tada vrijedi da je događaj  $a$  izведен prije događaja  $b$  odnosno vrijedi relacija  $a \rightarrow b$ . Pri tome  $\vec{T}(a) < \vec{T}(b)$  vrijedi ukoliko  $\exists i: \{\vec{T}(a)[i] < \vec{T}(b)[i] \wedge \forall j \neq i: \vec{T}(a)[j] \leq \vec{T}(b)[j]\}$ , tj. ukoliko postoji neka  $i$ -ta komponenta vektora  $\vec{T}(a)$  koja je manja od iste komponente vektora  $\vec{T}(b)$ , dok su vrijednosti svih ostalih komponenti vektora  $\vec{T}(a)$  manje ili jednake vrijednostima istih komponenti vektora  $\vec{T}(b)$ .

Na slici (Slika 7.8) je u praksi prikazana uporaba vektorskih oznaka vremena. U ovom primjeru vremenska oznaka [1, 2, 1] pridružena je prijemu prve poruke koju je proces r poslao procesu q, a oznaka [3, 3, 1] pridružena je slanju druge poruke od procesa p procesu q. Usporedbom komponenata tih vektorskih oznaka može se utvrditi da je događaj "prijem prve poruke koju je proces r poslao procesu q" prethodio događaju "slanje druge poruke od procesa p procesu q". Za vektorsku oznaku [3, 6, 1] događaja „slanje zadnje poruke od procesa q prema procesu r“ ne vrijedi da je manja (dogodio se ranije) ili veća (dogodio se kasnije) od vrijednosti vektorske oznake [1, 4, 3] događaja „slanja predzadnje poruke procesa r prema procesu p“. Drugim riječima, ne može se utvrditi da se jedan događaj prethodi drugome. Zbog navedenih okolnosti, moguće je zaključiti da su ta dva događaja nezavisna u vremenu.



Slika 7.8. Primjer uporabe vektorskih oznaka vremena

## 7.2 Sinkronizacija tijeka izvođenja procesa

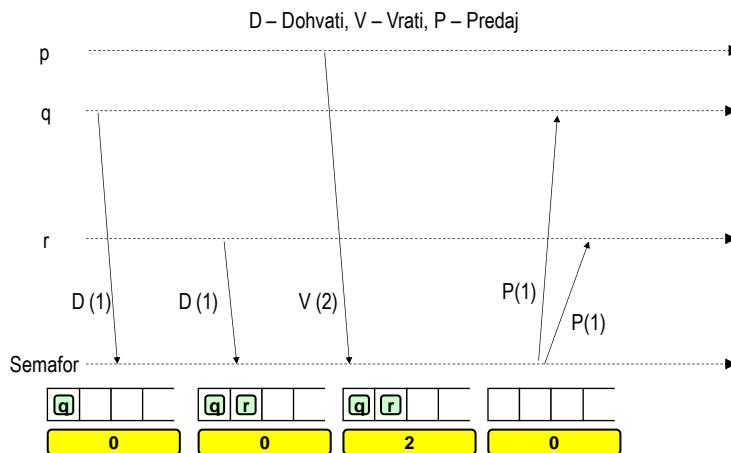
U operacijskim sustavima se koriste raznovrsni mehanizmi za isključivanje i međusobnu sinkronizaciju procesa koji se izvode na jednom računalu. Neke od tih mehanizama je također moguće primjeniti i pri usklađivanju procesa u raspodijeljenim sustavima. U nastavku su opisana dva najčešće korištena mehanizma sinkronizacije tijeka izvođena procesa u raspodijeljenoj okolini: **mehanizam semafora** i **mehanizam razmjene događaja**.

### 7.2.1 Mehanizam semafora

Semafor je proces koji sadrži spremnik s konačnim brojem znački i repom. Značke se dodjeljuju klijentima prema redoslijedu prispjeća (*First In First Out–FIFO*). Kada semafor ostane bez slobodnih znački, jer je sve raspoložive značke dodijelio klijentima, naknadno pristigle zahtjeve će spremiti u rep u kojem će oni čekati na oslobođanje znački. Primjenom odgovarajućih akcija klijenti mogu zatražiti dohvati ili semaforu vratiti jednu ili više znački. Tijek komunikacije se u slučaju primjene mehanizma semafora sastoji od sljedeća četiri koraka:

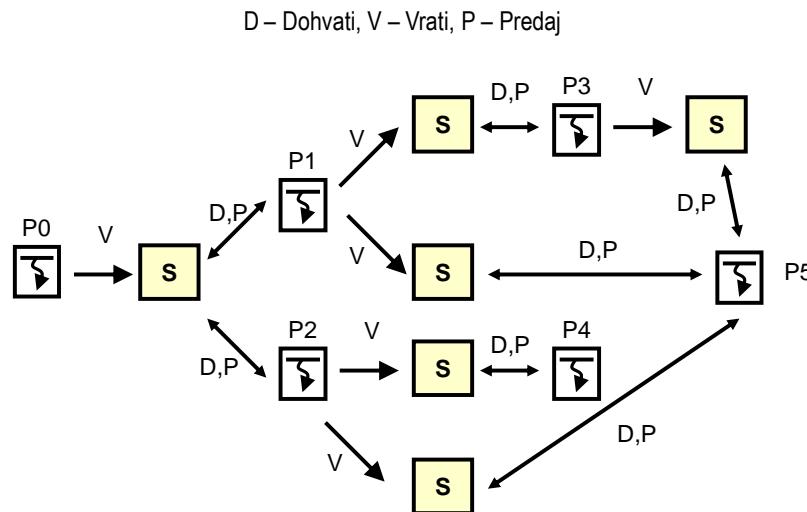
1. Proces šalje zahtjev semaforu za dohvati (D) jedne ili više znački,
2. Ako u spremniku semafora postoji traženi broj znački, semafor će procesu predati značke (P),
3. Ukoliko spremnik ne posjeduje dovoljan broj znački, zahtjev će se staviti u rep čekanja i
4. Nakon završetka obrade, proces će vratiti semaforu preuzete značke slanjem poruke vrat (V).

Slijedni dijagram na slici (Slika 7.9) prikazuje primjer sinkronizacije izvođenja tri procesa (p, q, r) putem semafora. Na početku izvođenja procesi q i r pokušavaju dohvatiti po jednu značku iz spremnika semafora. Međutim, s obzirom da spremnik ne sadrži značke, njihovi zahtjevi se spremaju u rep čekanja. Procesi q i r nakon toga čekaju na dodjelu znački od strane semafora. U određenom trenutku proces p vraća dvije značke u spremnik, čime je omogućeno proslijđivanje po jedne značke procesima q i r. Po primitku značke, ovi procesi nastavljaju sa svojim izvođenjem. Na opisani način je ostvarena sinkronizacija tijeka izvođenja u kojoj procesi q i r čekaju na signal od strane procesa p da bi započeli svoje izvođenje. Drugim riječima, ostvareno je grananje tijeka izvođenja s jednog procesa na dva dodatna nezavisna tijeka izvođenja.



Slika 7.9. Primjer sinkronizacije tijeka izvođenja procesa putem mehanizma semafora

Višestrukim grananjem tijeka izvođenja procesa se mogu ostvariti i puno složeniji obrasci sinkronizacije raspodijeljenih procesa. Osim grananja tijeka izvođenja, moguće je postići i spajanje te ponavljanje tijeka izvođenja. Većinom se ovi složeni obrasci sinkronizacije procesa prikazuju u obliku grafa tijeka izvođenja raspodijeljenih procesa. Na slici (Slika 7.10) je prikazan jedan takav graf.



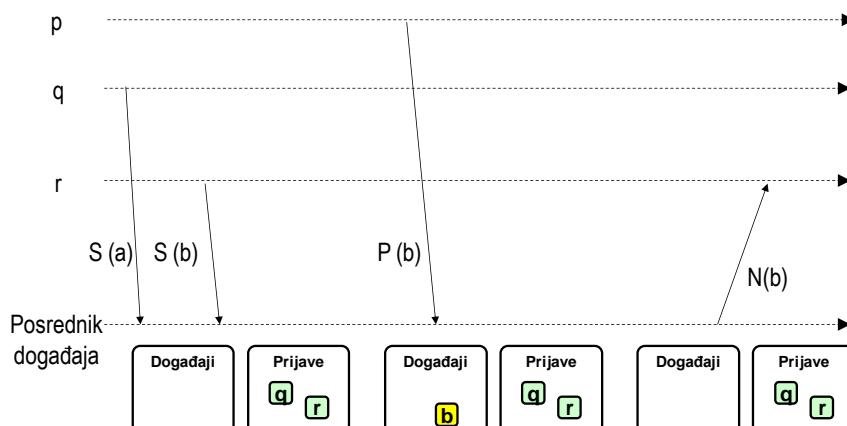
Slika 7.10. Primjer grafa tijeka izvođenja raspodijeljenih procesa

### 7.2.2 Mehanizam razmjene događaja

Sinkronizaciju tijeka izvođenja procesa je također moguće postići i mehanizmom razmjene događaja. Posrednik događaja je proces koji sadrži dva spremnika: spremnik s objavljenim događajima i spremnik pretplata (tj. prijava na događaje). U ovom slučaju se usporedba između pretplata i događaja temelji na modelu objavi-preplatni. Tijek komunikacije se u slučaju primjene mehanizma razmjene događaja sastoji od sljedeća tri koraka:

1. Proces posredniku šalje svoju pretplatu (S) koju on pohranjuje u svoj spremnik pretplata,
2. Neki drugi proces objavljuje događaj (P) posredniku koji ga sprema u svoj spremnik događaja,
3. Ako posrednik ima spremljenu pretplatu za objavljeni događaj, proslijedit će ga procesu pretplatniku u poruci dojave (N).

$S(x)$  – Prijava za događaje tipa  $x$ ,  $P(x)$  – Objava događaja tipa  $x$ ,  
 $N(x)$  – Dojavi događaj  $x$



Slika 7.11. Primjer sinkronizacije tijeka izvođenja procesa putem mehanizma razmjene događaja

Na Sliku 7.11 je prikazan slijedni dijagram jednog jednostavnog primjera sinkronizacije procesa putem mehanizma razmjene događaja. U prikazanom primjeru ostvarena je sinkronizacija tijeka izvođenja u kojoj proces p objavom odgovarajućeg događaja okida izvođenje procesa r. Na početku se procesi q i r pretplaćuju kod posrednika na različite vrste događaja. Nakon toga proces p objavljuje događaj vrste b. Posrednik uspoređuje događaj b s postojećim preplatama procesa q i r te ga proslijeđuje procesu r čiju pretplatu ovaj događaj zadovoljava.

### 7.3 Međusobno isključivanje procesa

Druga skupina postupaka za koje je potrebno ostvariti usklađivanje procesa u vremenu su postupci međusobnog isključivanja tijeka izvođenja procesa u raspodijeljenoj okolini. U nastavku su opisana tri najčešće korištena mehanizma međusobnog isključivanja procesa u raspodijeljenoj okolini: isključivanje putem središnjeg upravljača, raspodijeljeno isključivanje te isključivanje zasnovano na primjeni prstena.

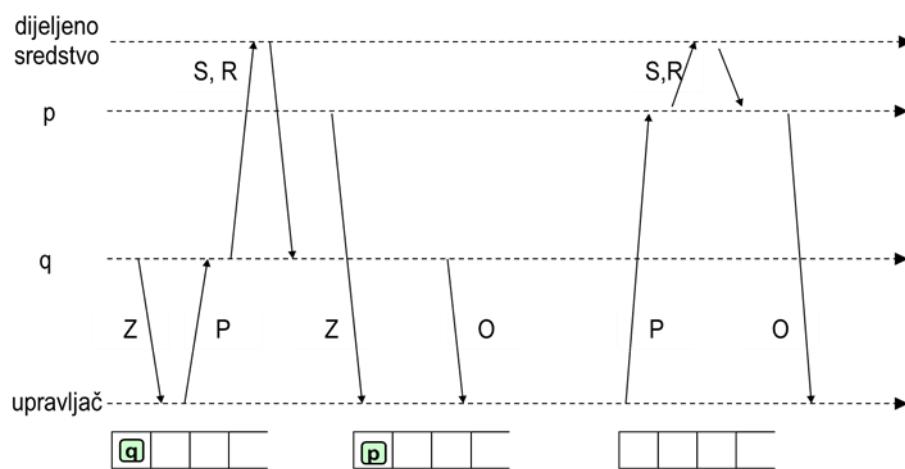
#### 7.3.1 Isključivanje putem središnjeg upravljača

Središnji upravljač s repom čekanja je proces koji upravlja pristupom dijeljenom sredstvu. On pristigle klijentske zahtjeve obrađuje prema redoslijedu prispijeća (*FIFO*). U slučaju zauzeća dijeljenog sredstva, pristigle klijentske zahtjeve će do oslobađanja sredstva čuvati u svom repu čekanja. Tijek komunikacije se u ovom slučaju primjene sastoji od sljedeća tri koraka:

1. Proces šalje zahtjev za zauzimanjem dijeljenog sredstva (Z) središnjem upravljaču,
2. Proces ostvarjuje pristup dijeljenom sredstvu nakon primitka potvrde (P) od strane središnjeg upravljača. Proces obavlja akciju dohvaćanja (R) i/ili spremanja (S) dijeljenog sredstva i
3. Nakon završetka obrade, proces otpušta zauzeto sredstvo slanjem poruke oslobodi (O) središnjem upravljaču.

Na slici (Slika 7.12) je prikazan slijedni dijagram primjera isključivanja 2 procesa (q i r) putem središnjeg upravljača koji održava rep čekanja za procese koji žele pristupiti dijeljenom sredstvu. Proces q prvo zauzima sredstvo slanjem poruke zahtjeva (Z) središnjem upravljaču i primitkom potvrde (P). Nakon toga proces q pristupa dijeljenom sredstvom. U međuvremenu proces p također traži pristup dijeljenom sredstvu. Međutim, s obzirom da je to sredstvo trenutno nedostupno, središnji upravljač pohranjuje ovaj zahtjev u svoj rep čekanja. Nakon završetka korištenja sredstva, proces p oslobađa pristup sredstvu slanjem poruke oslobađanja (O) središnjem upravljaču. Procesu p se nakon toga šalje potvrde odobravanja pristupa dijeljenom sredstvu nakon koje on ima mogućnost spremanja i/ili dohvaćanja dijeljenog sredstva.

R – Dohvati, S – Spremi, Z – Zauzmi, P – Potvrda, O – Oslobodi



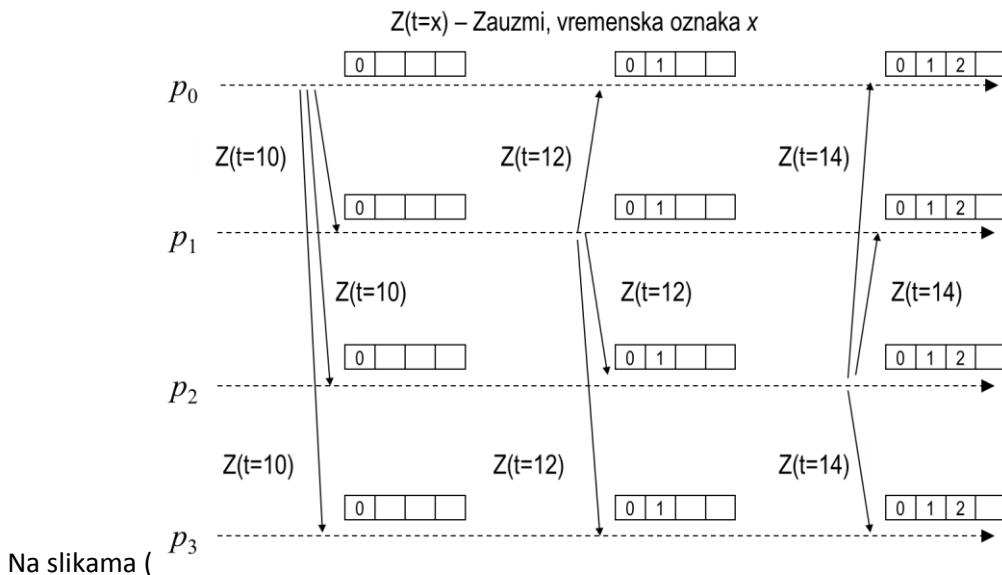
Slika 7.12. Primjer isključivanja procesa putem središnjeg upravljača

#### 7.3.2 Raspodijeljeno isključivanje

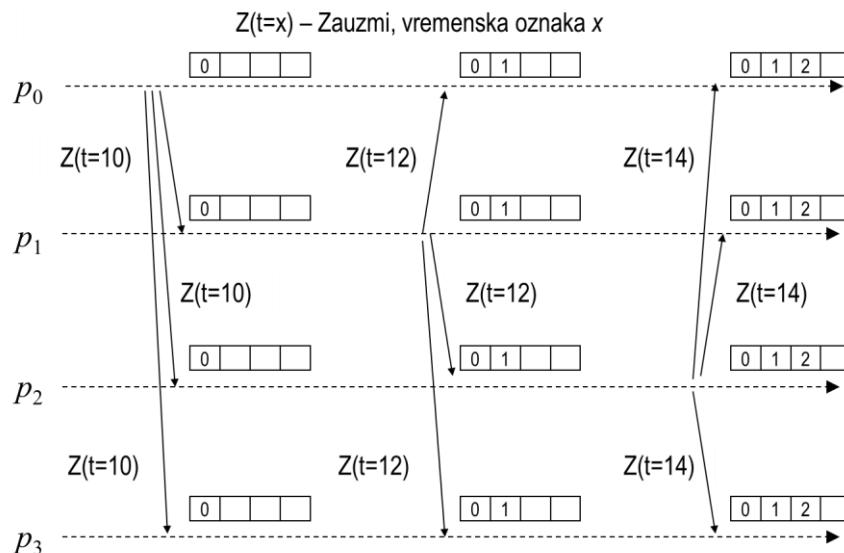
Kod raspodijeljenog međusobnog isključivanja procesa ne postoji centralizirani središnji upravljač s repom čekanja, već svaki proces ima svoj vlastiti rep čekanja, a procesi međusobno razmjenjuju

informacije potrebne za usklađivanje stanja svih repova čekanja u sustavu. Drugim riječima, rep čekanja je u ovom slučaju raspodijeljen i repliciran između skupa računala u raspodijeljenoj okolini.

Procesi međusobno razmjenjuju dvije vrste poruka: zahtjeve za pristup sredstvu i potvrde prihvaćanja zahtjeva za pristup sredstvu. Svaki zahtjev za pristup sredstvu uključuje logičku oznaku trenutka u kojem je proces uputio zahtjev i prosljeđuje se svim drugim procesima u sustavu. Zahtjevi se spremaju u lokalne repove čekanja i obrađuju prema redoslijedu prispjeća (*FIFO*). Nakon završetka pristupa dijeljenom sredstvu, proces koji je pristupao šalje svim drugim procesima potvrdu završetka pristupa sredstvu. Nakon toga svi procesi uzimaju prvi proces iz svoj repa čekanja te preostalim procesima šalju potvrde prihvaćanja tog zahtjeva za pristup dijeljenom sredstvu. Proces čiji je zahtjev odabran za pristup dijeljenom sredstvu započinje pristup nakon primitka potvrda prihvaćanja od preostalih procesa.

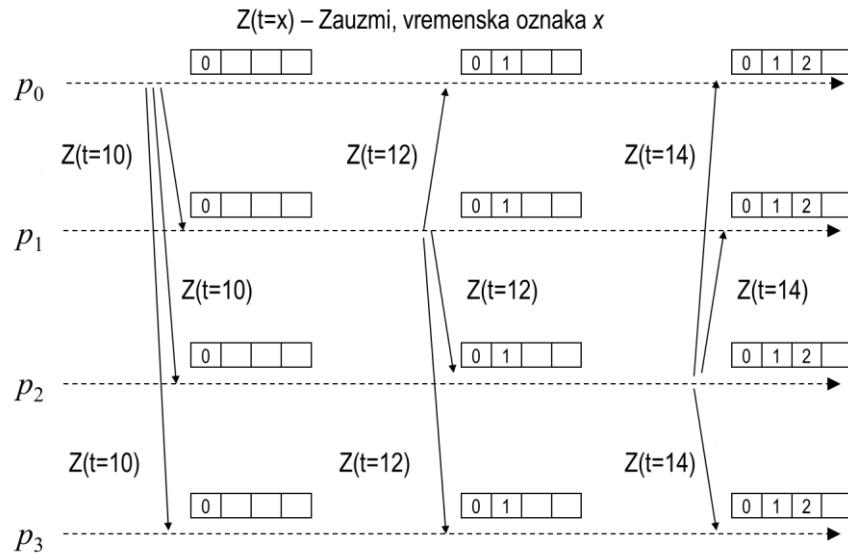


Slika 7.13 – 7.16) je prikazan primjer raspodijeljenog isključivanja procesa. U prvom koraku (



Slika 7.13) procesi  $p_0$ ,  $p_1$  i  $p_2$  šalju redom poruku za pristup sredstvu ("zauzmi") svim ostalim procesima. Zahtjevi se navedenim redoslijedom spremaju u lokalne repove čekanja. Proses  $p_3$  ne šalje poruku obzirom da u tom trenutku nije zainteresiran za pristup dijeljenom sredstvu. Nakon

završetka faze slanja poruka zauzeća, pravo pristupa ima proces  $p_0$  jer njegov zahtjev ima najmanju vremensku oznaku i prvi je smešten u rep.

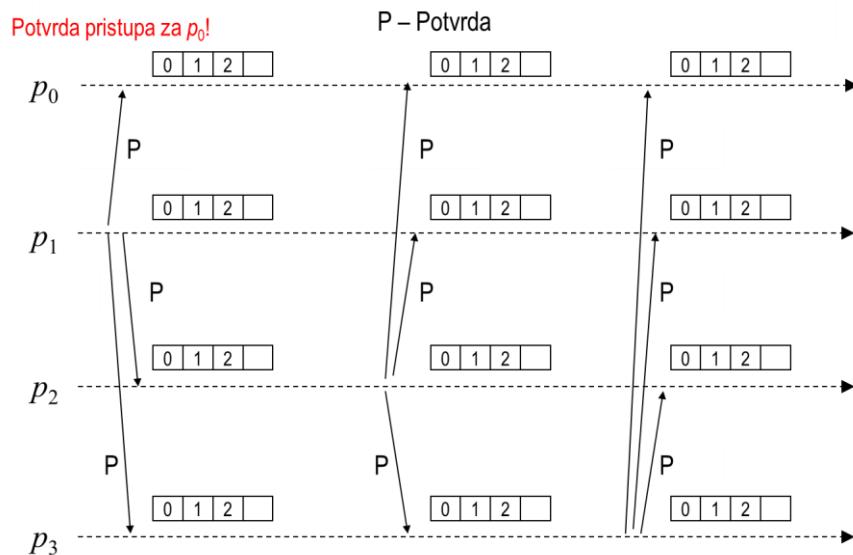


Slika 7.13. Primjer raspodijeljenog isključivanja procesa – prvi korak

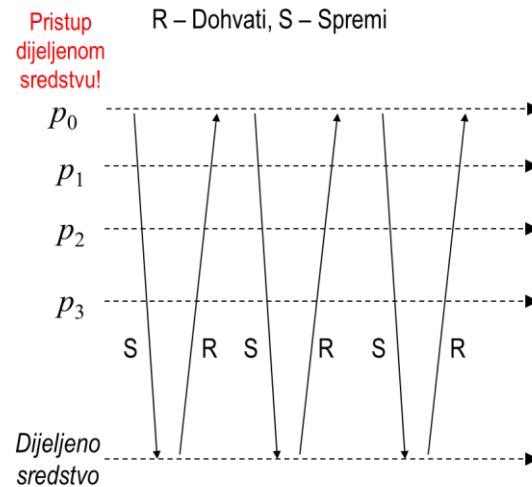
U drugom koraku svaki preostali proces koji nije ostvario pravo pristupa šalje potvrde svim ostalim procesima. U prikazanom primjeru, potvrde za pravo pristupa procesa  $p_0$  šalju procesi  $p_1$ ,  $p_2$  i  $p_3$  (Slika 7.14).

Nakon što je u drugom koraku primio potvrde od svih procesa, u trećem koraku proces  $p_0$  pristupa dijeljenom sredstvu te provodi operacije pisanja ("spremi") i čitanja ("dohvati") sadržaja (Slika 7.15).

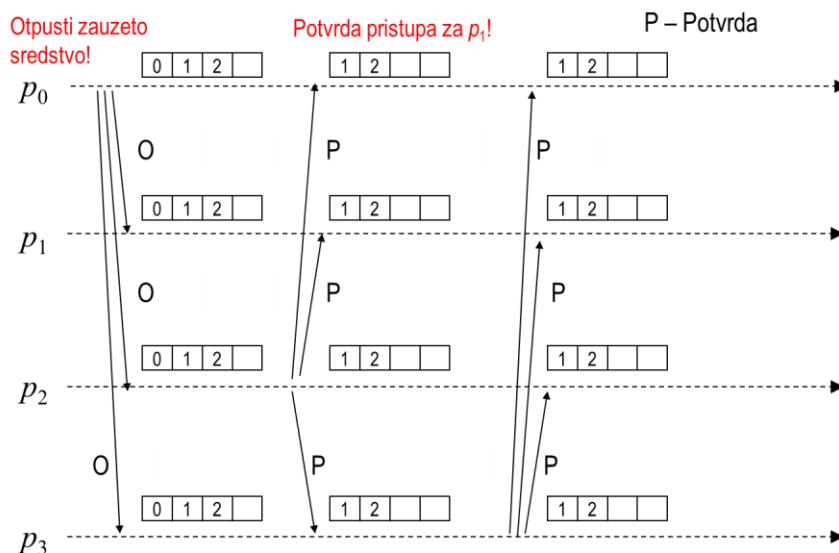
Završetkom provođenja operacija, u četvrtom koraku (Slika 7.16), proces  $p_0$  potvrđuje svim ostalim procesima da otpušta zauzeto sredstvo i procesu  $p_1$  omogućuje pristup dijeljenom sredstvu. Svi procesi uklanjaju oznaku procesa  $p_0$  iz svog lokalnog repa čekanja. Time zahtjev procesa  $p_1$  postaje prvi u svim lokalnim repovima čekanja, pa i ostali procesi osim procesa  $p_1$  ( $p_2$  i  $p_3$ ) šalju potvrdu svim procesima, čime započinje dodjela dijeljenog sredstva procesu  $p_1$ .



Slika 7.14. Primjer raspodijeljenog isključivanja procesa – drugi korak



Slika 7.15. Primjer raspodijeljenog isključivanja procesa – treći korak



Slika 7.16. Primjer raspodijeljenog isključivanja procesa

### 7.3.3 Isključivanje zasnovano na primjeni prstena

Primjenom prstena se omogućava vrlo jednostavno isključivanje procesa u vremenu. Pri tome su procesi povezani u logičku mrežu zasnovanu na prstenu. U sustavu postoji samo jedna značka kojom se ostvaruje pravo pristupa dijeljenom sredstvu te se ona razmjenjuje između procesa u prstenu. Prijestup dijeljenom sredstvu ima samo proces koji u određenom trenutku ima značku, a nakon završetka ili u slučaju nezainteresiranosti za pristup dijeljenom sredstvu proces prosljeđuje značku susjednom procesu u prstenu. Algoritam koji primjenjuje svaki proces se sastoji od 3 koraka:

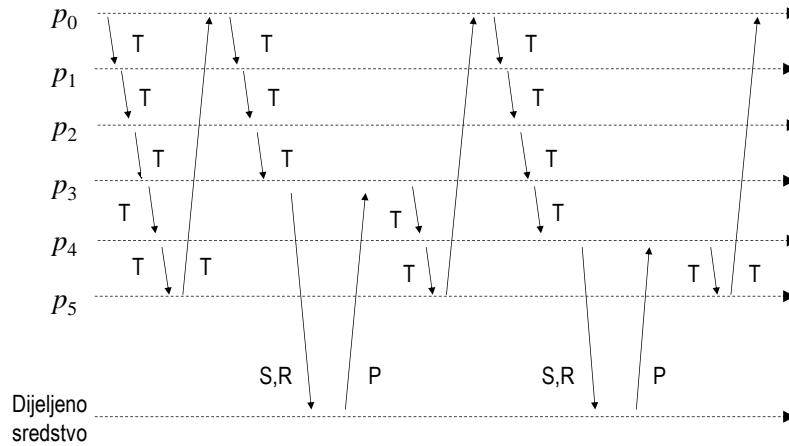
1. Čekaj poruku sa značkom od svog lijevog susjeda, a kada ju primiš ostvario si pravo na pristup dijeljenom sredstvu,
2. Ukoliko si zainteresiran za pristup dijeljenom sredstvu provedi operacije pisanja ("spremi") i/ili čitanja ("dohvati") sadržaja i
3. Prosljedi poruku sa značkom desnom susjedu u prstenu.

U slučaju kada svaki proces ostvaruje po jedan pristup dijeljenom sredstvu, vrijeme prijenosa značke za cijeli krug oko prstena koji uključuje N procesa je  $T = N \cdot (T_z + T_p + T_o) + N \cdot T_T$ . Ukupno vrijeme jednako je zbroju vremena slanja poruke zahtjeva ( $T_z$ ), obrade ( $T_o$ ) i primitka poruke odgovora ( $T_o$ ) za

$N$  procesa koji ostvaruju pristup dijeljenom sredstvu te vremena prijenosa poruke sa značkom ( $T_T$ ) duž  $N$  segmenata između procesa u prstenu.

Slijedni dijagram na slici (Slika 7.17) prikazuje primjer isključivanja raspodijeljenih procesa primjenom prstena. Značka kruži prstenom od jednog procesa do drugog sve dok ne dođe do procesa koji je zainteresiran za pristup dijeljenom sredstvu. U prikazanom primjeru su zainteresirani procesi bili proces  $p_3$  i  $p_4$ . Ovi procesi obavljaju operacije pisanja ("spremi") i/ili čitanja ("dohvati") sadržaja te isto kao i nezainteresirani procesi nakon toga prosljeđuju značku dalje svom drugom susjedu.

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

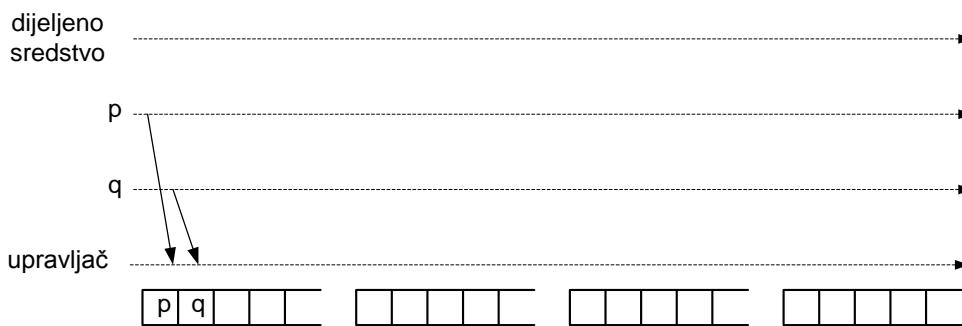


Slika 7.17. Primjer isključivanja procesa primjenom prstena

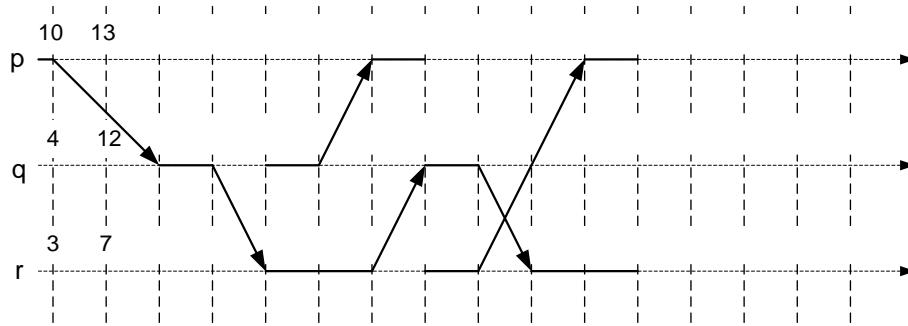
#### 7.4 Pitanja za učenje i ponavljanje

- 7.1 Prikažite i objasnite korake algoritma Berkeley za usklađivanje satnih mehanizama tri računala u raspodijeljenoj okolini. Računala imaju sljedeće vrijednosti satova  $T(p)=03:02:00$ ,  $T(q)=03:08:00$  i  $T(c)=03:12:00$ . Upravitelj je treće računalo. Prepostavite da prijenos poruke između 2 računala traje 1 minutu i da upravitelj koristi svoje lokalno vrijeme kao zajedničko pri usklađivanju satnih mehanizama.
- 7.2 Opišite postupak međusobnog isključivanja dvaju procesa ( $p$  i  $q$ ) primjenom središnjeg upravljača s repom čekanja tako da nacrtate redoslijed operacija i objasnite ih. Nakon zauzimanja dijeljenog spremnika, proces provodi jednu operaciju čitanja ili pisanja nad dijeljenim spremnikom.

R –Dohvati, S –Spremi, Z –Zauzmi, P –Potvrda, O –Oslobodi



- 7.3 Za slijed razmjene poruka između tri računala prikazan na slici uspostavite globalni tijek vremena primjenom skalarnih oznaka logičkog vremena. Navedite i opišite trenutke u kojima se ostvaruje korekcija lokalnih satnih mehanizama.



- 7.4 Pet procesa postavljenih na različita računala u raspodijeljenoj okolini ostvaruje međusobno isključivanje primjenom prstena. Vrijeme prijenosa poruke zahtjeva i odgovora pri pristupu dijeljenom sredstvu jednako je 3 ms, vrijeme obrade poruke zahtjeva na sredstvu je 5 ms, vrijeme prijenosa *tokena* između dva susjedna procesa u prstenu je 2 ms. Kada primi *token*, proces može maksimalno jednom ostvariti pristup dijeljenom sredstvu prije nego što prosljedi *token* idućem susjedu. Naznačite navedena vremena na dijagramu. Koje je minimalno, a koje maksimalno vrijeme čekanja bilo kojeg procesa u prstenu za pristup dijeljenom sredstvu.



## 8 KONZISTENTNOST I REPLIKACIJA PODATAKA

Replikacija podataka označava postupak stvaranja i održavanja više istovrsnih preslika – kopija izvornih podataka u raspodijeljenom sustavu. Razlozi za replikaciju su višestruki. Sustav s više istovrsnih preslika pouzdaniji je, jer u slučaju otkaza ili nemogućnosti pristupa jednoj replici sustav može nastaviti s obavljanjem svojih zadaća koristeći druge raspoložive replike. U slučaju otkrivenih pogrešaka, usporedbom više replika može se ustanoviti točno stanje podataka. Nadalje, može se povećati obradni kapacitet sustava, jer se više replika može koristiti za istodobno posluživanje više korisnika. Isto tako, može se skratiti vrijeme dostupa podacima, ako se replike postave na računalne sustave u područja s velikom koncentracijom korisnika i tako se „približe“ korisnički i poslužiteljski sustavi.

Postupkom stvaranja replika se izvorni podatkovni objekt postavljen na jednom računalnom sustavu preslikava u replicirani podatkovni objekt – repliku na drugom računalnom sustavu ili više njih. U trenutku stvaranja replike su jednake izvornom podatkovnom objektu – imaju isto stanje i podaci su konzistentni: pristupanjem bilo kojoj replici očitat će se ista vrijednost nekog podataka. Problemi nastaju nakon promjene podatka u jednoj od replika. Replike istog podatkovnog objekta mogu istodobno i nezavisno koristiti različiti korisnici i stoga mijenjati njihovo stanje u različitim vremenskim trenucima, tako da je održavanje konzistentnosti prijeko potrebno za ispravan rad raspodijeljenog sustava.

Postupci stvaranja i održavanja konzistentnosti replika su složeni, jer je s korisničkog motrišta potrebno postići transparentno rukovanje podacima. Korisnika se „ne tiče“ niti raspodijeljenost niti replikacija. Uz replikacijsku transparentnost treba postići transparentnost pristupa podacima (neovisnost o načinu pristupa i formatu podataka), lokacijsku i migracijsku transparentnost (neovisnost o lokaciji i promjeni lokacije podatkovnog objekta), konkurenčijsku transparentnost (prikrivanje višestruke istodobne uporabe podatkovnog objekta od strane više korisnika) i transparentnost na kvar (prikrivanje kvara). Kako se sve više aktivnosti odvija pristupom raspodijeljenim sustavima putem pokretne mreže, važni postaju i zahtjevi na relokacijsku transparentnost, odnosno mogućnost premještanja podatkovnog objekta, odnosno sredstva na kojem se nalazi, tijekom uporabe. U transparentno izvedenom sustavu korisniku „se čini“ da su svi podaci pohranjeni u jednom spremničkom prostoru, iako je riječ o raspodijeljenom spremničkom prostoru s podatkovnim objektima repliciranim na više umreženih računalnih sustava.

Idealno bi bilo svaku promjenu podatka u jednoj replici trenutno provesti u svima ostalima, što je naravno nemoguće postići u stvarnim sustavima u kojim su sve obradne i komunikacijske aktivnosti realnog vremenskog trajanja i svaki se događaj u jednom sustavu propagira do ostalih s nekim kašnjenjem. Stoga je za svaku primjenu potrebno odrediti prikladan model kojim će se utvrditi kakva se konzistentnost zahtijeva i u kojim se okolnostima postiže.

Modeli konzistentnosti podataka određuju što se postiže pri konkurentnim operacijama s više podatkovnih objekata u raspodijeljenim sustavima.

### 8.1 Modeli konzistentnosti podataka

Razmotrimo skup procesa:

$$\{p, q, r, s, \dots\}$$

skup podataka:

$$\{A, B, C, D, \dots\}$$

i skup lokacija u raspodijeljenom spremničkom prostoru:

$$\{x, y, z, w, \dots\}$$

Svaki proces može neki podatak upisati na neku lokaciju, primjerice proces p upisuje podatak A na lokaciju x operacijom pisanja W:

$W(x, A)$ .

Isto tako svaki proces može pročitati neki podatak upisan na neku lokaciju, primjerice proces q čita podatak B s lokacije y operacijom čitanja R:

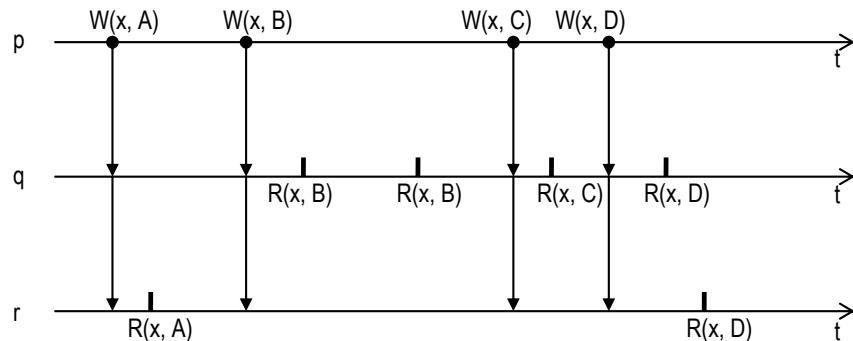
$R(y, B)$

Modeli konzistentnosti podataka govore o tome kako slijed upisa podataka na istu ili različite lokacije od strane jednog ili više procesa, „vide“ drugi procesi koji očitavaju te lokacije. Primjerice čitaju li samo zadnju upisanu vrijednost, mogu li pratiti slijed sukcesivnih upisa (promjena stanja), mogu li uočiti vremensku uređenost operacija (što je bilo prije ili poslije) i drugo.

### 8.1.1 Stroga konzistentnost

Stroga konzistentnost odgovara idealnoj situaciji kod koje je svaka promjena vrijednosti podatka koju provodi jedan proces upisom podatka na bilo koju lokaciju, trenutno vidljiva svim ostalim procesima, tj. pretpostavlja se sinkroni model raspodijeljenog sustava. Globalnim fizičkim taktom određen je trenutak promjene, tako da se postiže potpuna vremenska uređenost promjena stanja podataka , kako na jednoj lokaciji tako i na svim lokacijama.

U primjeru (Slika 8.1) u kojem vodoravne linije određuju vremensku os, proces p redom upisuje podatke A, B, C i D na lokaciju x. Upis vrijednosti podatka provodi se trenutno u spremniku podataka i kao takve vrijednosti su dostupne drugim procesima koji ih mogu pročitati, što pokazuju okomite strelice. Kad proces q čita podatak s lokacije x, uvijek će pročitati zadnju upisanu vrijednost, redom B, B, C pa D. Isto tako će proces r pročitati najprije A, pa zatim D.

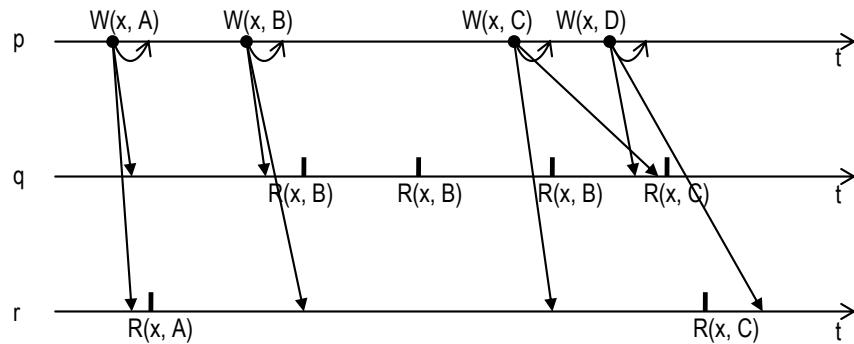


Slika 8.1. Primjer stroge konzistentnosti

U modelu stroge konzistentnosti rezultati svih operacija upisivanja trenutno su vidljivi svim ostalim procesima u raspodijeljenom sustavu, tako da operacija čitanja procesa na bilo kojem računalu daje upravo sadržaj zapisan u prethodnoj operaciji pisanja.

Zašto ovaj model nije realan i provediv u praktičkoj izvedbi raspodijeljenog sustava? Lokalna promjena stanja podatka koju je proveo proces p će biti vidljiva procesima q i r nakon komunikacijskog kašnjenja potrebnog da im se proslijedi promjena, odnosno nakon provedene promjene stanja replike s kojom rade. Primjer na slici (Slika 8.2) pokazuje utjecaj kašnjenja na narušavanje konzistentnosti podataka. Operacije upisivanja koje provodi proces p vidljive su procesima q i r s kašnjenjima, i to različitima, ovisnim o informacijskom prometu u mreži. Tako će proces q očitati na lokaciji x vrijednosti podatka B, B, B pa C, a proces r A pa C. U ovom primjeru q nije u trećem, već u četvrtom čitanju pročitao C, a podatak D uopće nije pročitao. Proses r još nije stigao

pročitati D. Treba uočiti da i lokalna promjena stanja podataka (u „svojoj“ replici) nije zanemarivog trajanja!



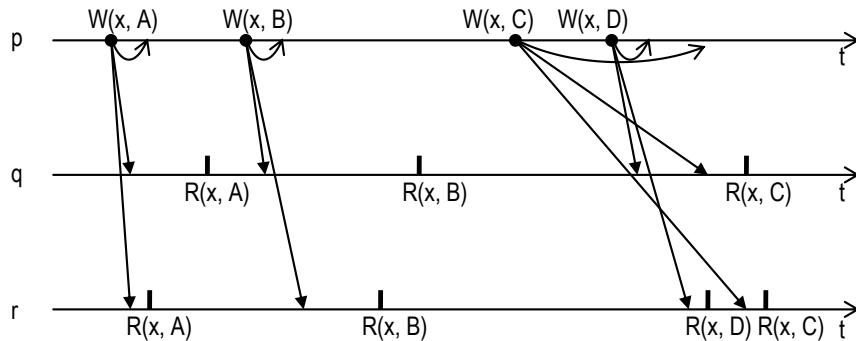
Slika 8.2. Primjer narušavanja konzistentnosti zbog kašnjenja

Očvidno je da za praktičku izvedbu raspodijeljenih sustava treba uvesti modele konzistentnosti koji uvažavaju realnost kašnjenja i postavljaju blaže uvjete na konzistentnost. To se odnosi i na same vrijednosti podataka i na konzistentnost uređenosti operacija nad podacima. Drugim riječima, u takvim su modelima određene operacije vidljive svim procesima u istom redoslijedu. Takvi modeli ne zahtijevaju održavanje globalnog vremenskog tijeka, što je primjereno raspodijeljenim sustavima.

### 8.1.2 Slijedna konzistentnost

Model slijedne konzistentnosti (engl. *sequential consistency*) zahtijeva da svi procesi moraju slijed izvođenja operacija čitanja i pisanja u vremenu vidjeti na jednak način. Za svaki proces to se odnosi na operacije upisivanja koje izvode drugi procesi i vlastite operacije čitanja.

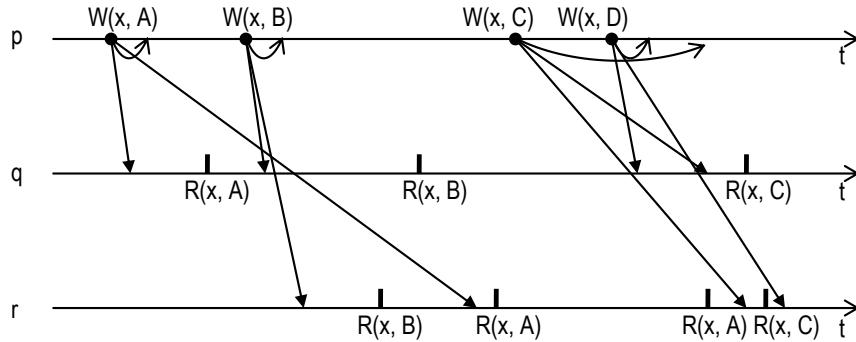
Kod slijedne konzistentnosti redoslijed operacija upisivanja i čitanja može biti proizvoljan, ali ga svi procesi moraju vidjeti jednako. Pokažimo to primjerom tri procesa na slici (Slika 8.3).



Slika 8.3. Primjer slijedne konzistentnosti

Proces p upisuje redom u lokaciju x podatak A, B, C i D, pri čemu je zbog kašnjenja za sva tri procesa operacija upisivanja C završena nakon operacije upisivanja D. Proses q čita redom A, B i C, a proces r A, B, D i C. Slijedna konzistentnost je postignuta, jer procesi q i r očitavaju vrijednosti podatka na lokaciji x u istom redoslijedu tj., A, B, (D), C. Oba procesa su vrijednosti koje su očitali (A, B, C) očitali u istom redoslijedu. To što proces q nije pročitao D ne utječe na slijednu konzistentnost ovih operacija upisivanja i čitanja.

Kad bi se operacija upisivanja A provela s većim kašnjenjem prema procesu r, a proces r čita kao na slici (Slika 8.4), narušila bi se slijedna konzistentnost. Proces q čita A, B pa C, a proces r čita B, A, C: A i B nisu očitani u istom redoslijedu zahtijevanom za slijednu konzistentnost. Isto tako, proces r može pročitati D nakon C, što proces q više ne može pa bi se i time narušila slijedna konzistentnost.



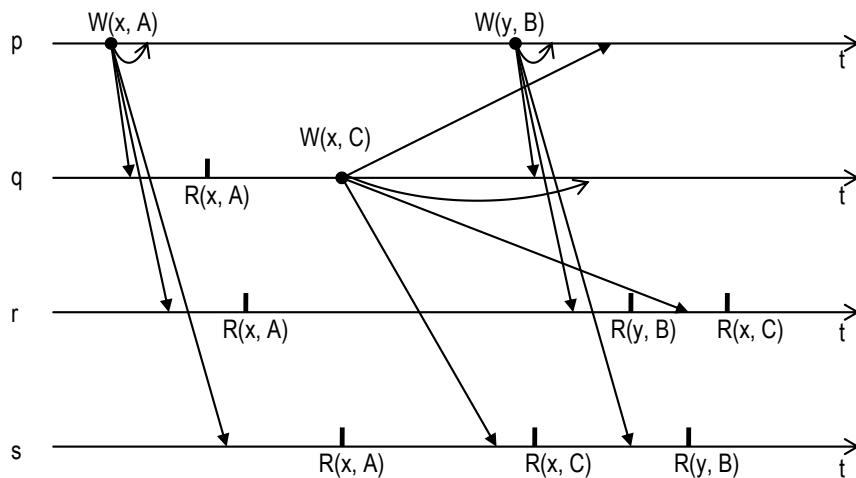
Slika 8.4. Primjer narušavanje slijedne konzistentnosti

### 8.1.3 Povezana konzistentnost

Povezana ili kauzalna konzistentnost (engl. *causal consistency*) oblik je slabije slijedne povezanosti kod koje se pozornost obraća samo potencijalno povezanim operacijama, odnosno operacijama koje mogu biti u uzročno-posljedičnoj vezi. Tako povezane operacije uvek se moraju vidjeti jedna (uzrok) za drugom (posljedica).

Model povezane konzistentnosti zahtijeva da povezane operacije svi procesi moraju vidjeti u istom redoslijedu. Redoslijed izvođenja povezanih operacija upisivanja vidljiv je svim procesima na jednak način, dok redoslijed izvođenja operacija upisivanja koje nisu povezane, svakom procesu može biti prikazan na drukčiji način.

Pokažimo to na primjeru četiri procesa na slici (Slika 8.5).



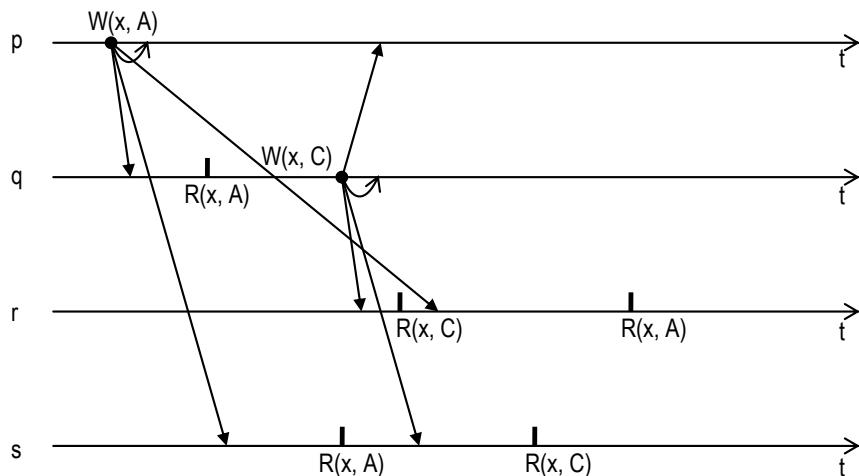
Slika 8.5. Primjer povezane konzistentnosti

Proces p upisuje u lokaciju x vrijednost A, a nakon toga u lokaciju y vrijednost B. Proses q čita A iz lokacije x i upisuje C na lokaciju x. Operacije R(x, A) i W(x, C) su potencijalno povezane, jer očitana

vrijednost A može utjecati na izračun vrijednosti C. Operacije  $W(y, B)$  i  $W(x, C)$  nisu povezane, jer upisuju dva neovisna podatka na dvije različite lokacije. To je tipičan primjer konkurentnih operacija.

Procesi r i s trebaju povezane operacije vidjeti na isti način da bi vrijedila povezana konzistentnost. U ovom primjeru proces r čita redom A, B i C, a proces s čita A, C i B. Rezultate povezanih operacija  $R(x, A)$  i  $R(x, C)$  oba vide u istom redoslijedu, A prije C, čime je zadovoljen uvjet povezane konzistentnosti. B i C nisu povezani, tako da njihov redoslijed nije važan.

Zamislimo situaciju u kojoj je upis A na lokaciju x proslijeden do procesa r uz veliko kašnjenje (Slika 8.6). Kao što je prije ustanovljeno  $R(x, A)$  i  $W(x, C)$  su potencijalno povezane operacije i takvima bi ih trebali vidjeti procesi r i s.



Slika 8.6. Primjer narušavanja povezane konzistentnosti

U ovom primjeru povezana konzistentnost je narušena, jer proces r očitava C prije A, a proces s čita A prije C, s iste lokacije.

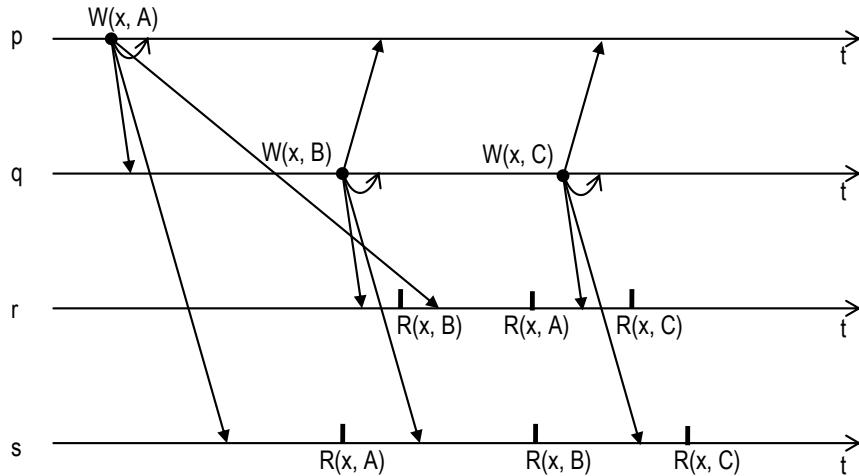
#### 8.1.4 Konzistentnost redoslijeda upisivanja

Ovakva je konzistentnost ostvarena ako je redoslijed izvođenja operacija upisivanja provedeni od strane jednog procesa vidljiv na jednak način svim ostalim procesima. Pritom redoslijed izvođenja operacija upisivanja različitih procesa može biti vidljiv na proizvoljan način ostalim procesima.

U primjeru na slici (Slika 8.7) proces p upisuje A na lokaciju x, a zatim proces q upisuje na istu lokaciju B pa C. Proces r čita redom B, A, C, a proces s čita A, B, C. Konzistentnost redoslijeda upisivanja je zadovoljena, jer procesi r i s očitavaju B i C u redoslijedu u kojem ih je proces q upisao. Redoslijed operacija upisivanja različitih procesa, u ovom primjeru procesa p i q, ostali procesi mogu vidjeti različito, a to je i slučaj: B, A, C, odnosno A, B, C. Učinak operacije upisivanja A na lokaciju x koju provodi proces p može biti vidljiv na proizvoljan način procesima r i s, jer je to jedina operacija upisivanja procesa p.

Model konzistentnosti redoslijeda naziva se i FIFO-konzistentnost (engl. *FIFO consistency, First In First Out consistency*), jer se ranije upisani (*First In*) čita prije (*First Out*) kasnije upisanoga podatka.

Ostvarenje konzistentnosti redoslijeda upisivanja jednostavno je postići pridruživanjem jedinstvenih oznaka svakom zahtjevu za upisivanje. Takva oznaka uključuje identifikator procesa i redni broj izvođenja operacije. U ovom bi se primjeru operacije upisivanja mogle označiti s  $W_{p1}(x, A)$ ,  $W_{q1}(x, B)$  i  $W_{q2}(x, C)$ .



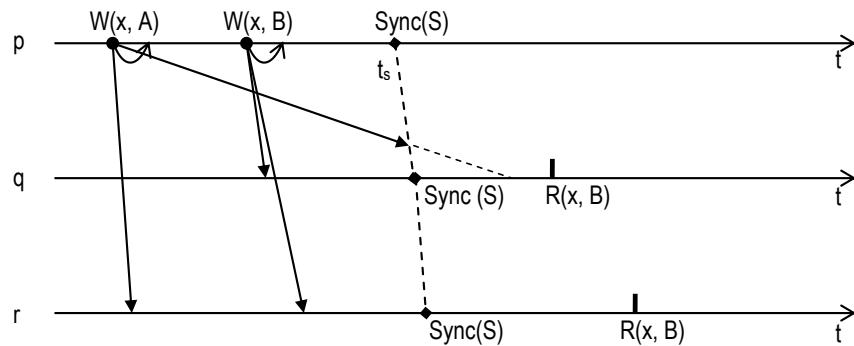
Slika 8.7. Primjer konzistentnosti redoslijeda upisivanja

Kad bi kašnjenje upisa  $W_{q_1}(x, B)$  bilo takvo da proces r očita s lokacije x redom A, C, B konzistentnost redoslijeda bi bila narušena.

### 8.1.5 Slaba konzistentnost

Slaba konzistentnost ostvaruje se primjenom sinkronizacijskih varijabli kojima se upravlja usklađivanjem replika podatkovnog objekta. Proces koji provodi upisivanje podataka operaciju sinkronizacije pokreće nakon što je završio sve prethodno započete operacije upisivanja. Svi ostali procesi mogu izvoditi operacije pisanja i čitanja po završetku izvođenja operacije sinkronizacije.

Za razliku od stroge konzistentnosti (Slika 8.1) kod koje je svaka promjena podatka bila vidljiva trenutno svim procesima, model slabe konzistentnosti omogućuje da svi procesi vide istu vrijednost podataka u trenutku nakon što je provedeno usklađivanje podataka – njihova sinkronizacija. Na slici (Slika 8.8) prikazano je kako se može ostvariti slaba konzistentnost primjenom sinkronizacijske varijable Sync(S).



Slika 8.8. Primjer slabe konzistentnosti

Po završetku upisa podataka, proces p u trenutku  $t_s$  pokreće usklađivanje podataka prema procesima q i r. Zadaća operacije sinkronizacije je da uskladi vrijednosti replika sa zadnjom vrijednosti koju je upisao proces koji pokreće sinkronizaciju, tj. (x, B) i da po njenom završetku svi procesi vide (x, B).

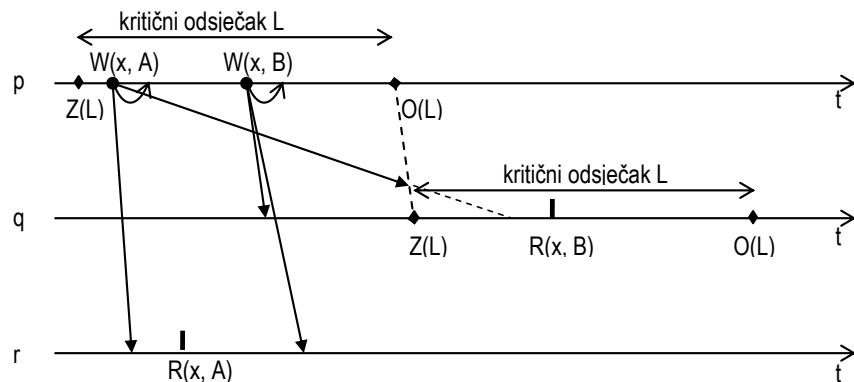
Što bi se dogodilo da nije bilo sinkronizacije? Kad bi procesi q i r očitali lokaciju x u trenutku  $t_s$  rezultat bi bio R(x, B). Međutim, prosljeđivanje upisa W(x, A) koji kasni, „prebrisalo“ bi vrijednost B i proces q bi nakon toga čitao prethodnu vrijednost, R(x, A), tj. ne bi video stvarno stanje, odnosno zadnju upisanu vrijednost B. Provođenjem operacije sinkronizacije onemogućit će se naknadna promjena vrijednosti u (x, A) koja bi se u ovom primjeru dogodila nakon zakašnjelog prosljeđivanja prve operacije upisivanja W(x,A) koju je proveo proces p.

### 8.1.6 Konzistentnost otpuštanja i zauzimanja kritičnog odsječka

Konzistentnost otpuštanja i konzistentnost zauzimanja odnose se na primjenu kritičnih odsječaka od strane procesa u raspodijeljenom sustavu. Ulazak procesa u kritični odsječak određen je operacijom zauzimanja, a izlazak operacijom otpuštanja koje se koriste za isključivi pristup spremniku podataka, tijekom kojeg može privremeno biti narušena konzistentnost.

Konzistentnost otpuštanja odgovara modelu kod kojeg se prije izlaska iz kritičnog odsječka sve lokalne promjene podataka prosljeđuju svim replikama i uspostavlja konzistentnost podataka. Konzistentnost zauzimanja odgovara modelu kod kojeg se nakon ulaska u kritičnog odsječka preuzimaju sve promjene podataka koje su proveli drugi procesi i uspostavlja konzistentnost podataka.

Pogledajmo najprije konzistentnost otpuštanja na primjeru kritičnih odsječaka i operacija zauzimanja (Z) i otpuštanja (O) na slici (Slika 8.9). S motrišta upisivanja podataka situacija je jednaka onoj kojom je ilustrirana slaba konzistentnost: proces p zauzima kritični odsječak L i upisuje redom W(x, A) i W(x, B).



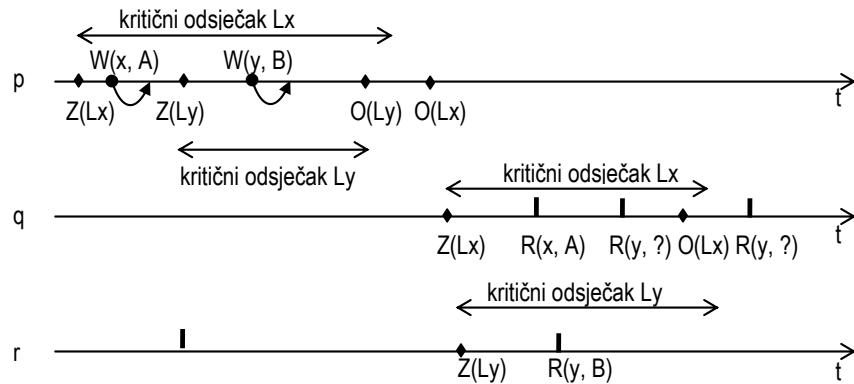
Slika 8.9. Primjer konzistentnosti otpuštanja

Operacijom otpuštanja kritičnog odsječka O(L) zadnja upisana vrijednost (x, B) prosljeđuje se procesu q koji po ulasku u kritični odsječak čita (x, B). Ovakvim je postupkom sprječena naknadna promjena koju bi uzrokovalo zakašnjelo preuzimanje (x, A). Drugim riječima, upis A nije prosljeđen do procesa q nakon B i nije narušena konzistentnost. Proces r provodi operaciju čitanja izvan kritičnog odsječka zbog čega vrijednost koja dohvata može biti bilo koja od vrijednosti zapisane od p, ovisno o trenutku čitanja, u ovom primjeru to je R(x, A).

Model konzistentnosti zauzimanja oslanja se na ulazak u kritični odsječak: u trenutku ulaska u kritični odsječak uspostavlja se konzistentnost i operacije u odsječku započinju tek nakon što su usklađene vrijednosti podataka. U prikazanom primjeru (Slika 8.10), proces p u kritičnom odsječku L<sub>x</sub> zapisuje vrijednost A na lokaciju x. U istom odsječku proces p ulazi u ugniježđeni kritični odsječak L<sub>y</sub> u kojem zapisuje vrijednost B na lokaciju y.

Po izlasku procesa p iz kritičnih odsječaka, nema nikakvog jamstva za čitanje vrijednosti s lokacija x i y. Kad nakon toga proces q zauzima kritični odsječak L<sub>x</sub>, ostvaruje se konzistentnost podataka, tj. (x,

A) i proces q čita  $R(x, A)$ . Kako se proces q ne nalazi u kritičnom odsječku  $Ly$ , ne može se reći koju će vrijednost podatka na lokaciji y pročitati – ne može se jamčiti konzistentnost tog podatka.



Slika 8.10. Primjer konzistentnosti zauzimanja

Proces r ulazi u kritični odsječak  $Ly$ , nakon čega se ostvaruje konzistentnost, te proces čita odgovarajuću vrijednost B s lokacije y.

### 8.1.7 Usporedba modela konzistentnosti

Modeli konzistentnosti mogu sa usporediti s dva motrišta: složenosti ostvarivanja i složenosti primjene. Složenost ostvarivanja opisuje koliko je zahtjevno ostvariti programsku potporu za uspostavu modela konzistentnosti u raspodijeljenoj okolini, dok složenost primjene opisuje koliko je složeno koristiti ostvareni model (Tablica 8.1).

S obzirom na složenost ostvarivanja, najsjloženiji je model stroge konzistentnosti. Međutim, ova vrsta modela je najjednostavnija za korištenje s obzirom da je slična modelu izgradnje programa u jednoprocесorskim sustavima.

S druge strane, najjednostavniji za ostvarenje je model konzistentnosti zasnovan na zauzimanju sredstava, ali je složenost uporabe ove vrste modela najveća. Naime, ovaj model omogućava izgradnju vrlo složenih međuzavisnosti toka podataka između procesa u raspodijeljenoj okolini.

Tablica 8.1 Usporedba modela konzistentnosti

Složenost ostvarivanja	Model konzistentnosti	Složenost primjene
najveća ↑ najmanja	stroga konzistentnost slijedna konzistentnost povezana konzistentnost slaba konzistentnost konzistentnost otpuštanja konzistentnost zauzimanja	najmanja ↓ najveća

Općenito vrijedi pravilo, da je složenije modele teže programski ostvariti, lakše moguće koristiti tijekom razvoja raspodijeljenog sustava. Modeli koji nisu složeni za ostvarenje oslanjaju se na arhitekturna obilježja raspodijeljenog sustava. To usložnjava primjenu, jer razvijatelj mora voditi računa o svim značajkama raspodijeljenih sustava tijekom izrade raspodijeljene aplikacije.

## 8.2 Organizacija sustava replika i vrste replika

Replikacija podataka se vrlo često koristi u raspodijeljenim sustavima te stoga predstavlja i jedno od najvažnijih pitanja pri njihovom razvoju. U tipičnom raspodijeljenom sustavu, podatkovni objekti

(resursi) se nalaze na različitim računalima pri čemu se na jednom računalu mogu nalaziti različiti resursi, a isti resursi se mogu nalaziti na različitim računalima u sustavu. Replika predstavlja kopiju izvornog resursa na nekom drugom računalu u raspodijeljenom sustavu. Replikacija podataka je postupak upravljanja replikama u sustavu, a sastoji se od stvaranja novih replika i održavanja konzistentnosti već postojećih replika.

Već je napomenuto da se od raspodijeljenih sustava očekuje replikacijska transparentnost – korisnici ne bi smjeli biti svjesni postojanja fizičkih kopija resursa u sustavu. Oni vide resurse kao jedinstvene logičke podatkovne objekte neovisne o broju i lokaciji njihovih fizičkih kopija u sustavu. Osim toga, rezultate operacija nad njima također vide jednoznačno te očekuju točne i nedvosmislene odgovore.

Tri glavna problema kojima se bavi ovo poglavlje su odabir lokacija na koje će se smjestiti replike, postavljanje replika na odabrane lokacije i način(i) održavanja njihove konzistentnosti. Raspodijeljeni sustav je konzistentan u nekom vremenskom trenutku ukoliko se sve replike u njemu nalaze u istom stanju kao i njihovi izvornici. U praksi je nemoguće postići konzistentnost raspodijeljenog sustava u svim vremenskim trenutcima, već se konzistentnost u pravilu postiže nakon nekog (kratkog) vremenskog perioda – kad se sve promjene izmijenjenog podatkovnog objekta prenesu do svih računala u sustavu koja održavaju njegove replike.

Prilikom odabira lokacija na koje ćemo smjestiti replike moramo se zapitati što uopće želimo postići replikacijom podataka u raspodijeljenom sustavu. Dvije su glavne prednosti korištenja replikacije: 1) povećanje pouzdanosti i raspoloživosti raspodijeljenih sustava te 2) poboljšanje njihovih performansi.

Korisnici očekuju visoku raspoloživost informacijskih sustava – dostupnost sustava u svakom trenutku. Centralizirani sustavi to ne mogu osigurati, jer imaju jedinstvenu točku ispada – središnjeg poslužitelja. Primjer takvog sustava je poslužitelj weba čijim ispadom trenutno prestaje mogućnost posluživanja klijentata. Replikacijom podataka svi se potrebni resursi web-poslužitelja mogu prenijeti na dodatni (rezervni) poslužitelj koji će korisnike opsluživati nakon ispada glavnog poslužitelja. Ovime se postiže veća razina raspoloživosti, jer je vjerojatnost ispada oba poslužitelja u istom trenutku puno manja od vjerojatnosti ispada samo jednog od njih.

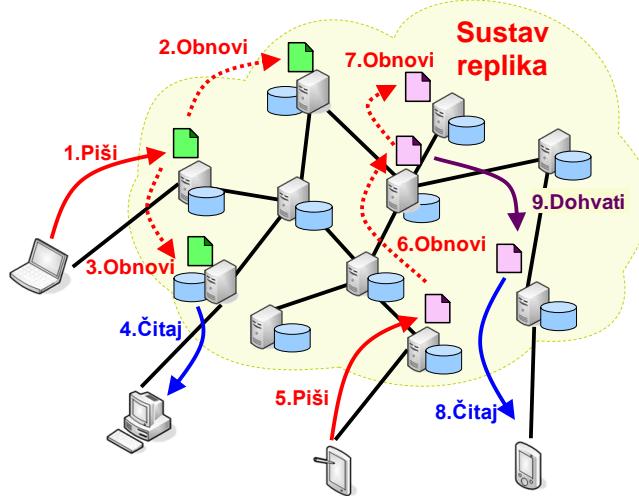
Također je moguća situacija da je neki poslužitelj dostupan, ali nije ispravan. Replikacijom resursa na više poslužitelja omogućuje se tehnikom glasovanja odlučiti o (ne)ispravnom poslužitelju, isključiti informacijski promet prema njemu i omogućiti ispravan rad sustava. Ovime se povećava se pouzdanost sustava, jer se smanjuje vjerojatnost njegova neispravnog rada tijekom vremena.

Nadalje, centralizirani sustavi nisu skalabilni zato što se povećanjem njihova opterećenja povećava i vrijeme odziva. Performance se mogu poboljšati izgradnjom raspodijeljenog sustava, repliciranjem podatkovnih na nekoliko poslužitelja i raspodjeljom opterećenja među njima. Na ovaj način se postiže dobra skalabilnost i brže vrijeme odziva zato što sada svaki od poslužitelja poslužuje samo dio korisnika. U raspodijeljenim sustavima, npr. webu, se često koristi tehniku priručne pohrane podataka na strani korisnika. Njome se postiže povećanje brzine odziva i smanjuje opterećenje poslužitelja jer se izbjegava kontaktiranje poslužitelja u slučaju raspoloživosti podataka na klijentskoj strani. Replikacija resursa na lokacije bliže korisnicima isto tako se često koristi za smanjenje vremena odziva.

Glavni nedostaci replikacije su dodatna obrada i mrežni promet koji se javljaju prilikom postavljanja i održavanja konzistentnosti replika u raspodijeljenom sustavu, o čemu je potrebno voditi računa pri organizaciji sustava replika.

Sustav replika je složeni raspodijeljeni sustav u kojem se replike spremaju i razmjenjuju između skupa klijentskih računala i lokalnih spremnika poslužiteljskih računala. Slika 8.11 prikazuje interakciju u sustavu replika. Ovaj sustav se sastoji od nekoliko poslužitelja od kojih svaki sadrži određene replike podatkovnih objekata. Klijente uvjek poslužuju poslužitelji s kojima su oni izravno povezani, tako da klijenti mogu vršiti operacije pisanja i čitanja replika na poslužiteljima. U slučaju da klijent od

poslužitelja zatraži repliku koju on ne sadrži, tražena replika će biti dohvaćena od najbližeg poslužitelja koji njome raspolaže i zatim proslijeđena klijentu.



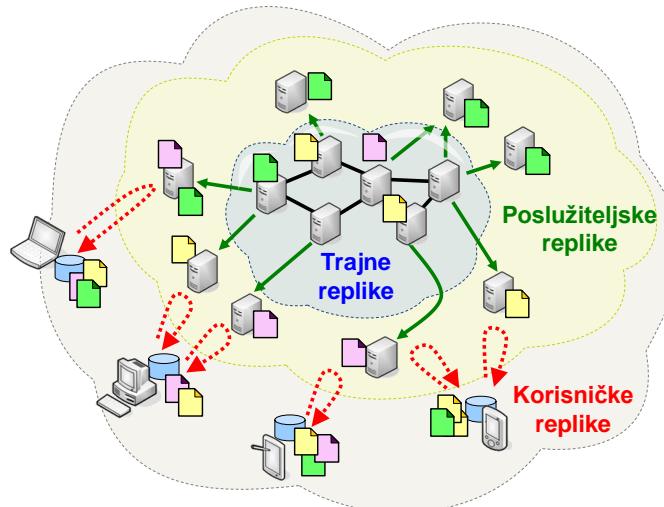
Slika 8.11. Sustav replika

Poslužiteljska računala provode, uz operacije dohvaćanja replika s drugih poslužitelja, i obnavljanje replika. Prilikom upisivanja, novo stanje replike se operacijama obnavljanja dostavlja svim poslužiteljima koji je sadrže. Na ovaj način se postiže konzistentnost replika u sustavu.

Ovisno o lokaciji na koju su postavljane replike u raspodijeljenim sustavima, dijelimo ih na sljedeće tri vrste:

- trajne replike,
- poslužiteljske replike i
- korisničke replike.

Kao što je prikazano na slici (Slika 8.12), trajne replike su postavljene na replikacijskim poslužiteljima koji se nalaze dalje od korisnika, poslužiteljske replike su postavljene na poslužiteljima bliže korisnicima i na kraju, korisničke replike su postavljene na korisničkim računalima tj. najbliže korisnicima.



Slika 8.12. Vrste replika ovisno o njihovoj lokaciji

Pritom je bitno razlikovati lokaciju samih replika u raspodijeljenom sustavu od lokacije replikacijskih poslužitelja koja najčešće nije optimizacijski problem kojeg je potrebno riješiti, već je vezan uz

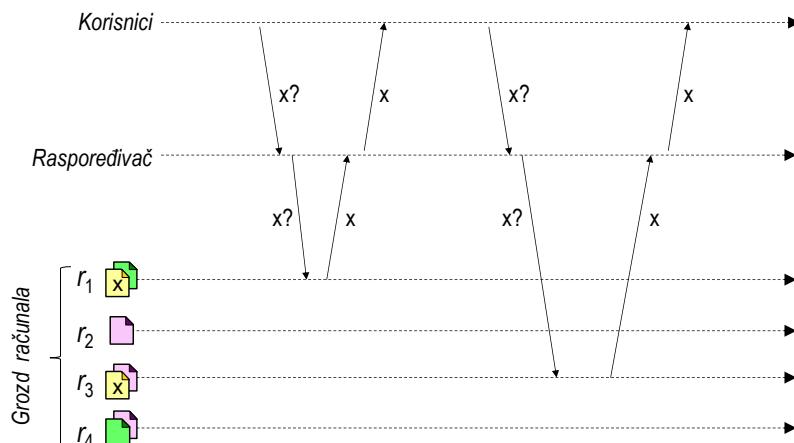
postavljanje, cijenu i isplativost raspodijeljenog sustava. Postavljanje velikog broja replikacijskih poslužitelja blizu korisnika osigurava najveću moguću pouzdanost i odlične performance, ali zasigurno nije najisplativije rješenje. Stoga je pri oblikovanju arhitekture raspodijeljenog sustava i određivanja lokacije replikacijskih poslužitelja potrebno pažljivo uravnotežiti ekonomsku isplativost sustava i zadovoljstvo korisnika. U nastavku se isključivo orientiramo na razmještaj replika u raspodijeljenim sustavima replika za koje ćemo pretpostaviti da imaju unaprijed određenu statičnu arhitekturu.

### 8.2.1 Trajne replike

Trajnim replikama naziva se početni skup stalno postavljenih replika na skupu replikacijskih poslužitelja povezanih lokalnom mrežom (grodz računala). Organizacija replikacijskih poslužitelja je statična, jer ih postavlja administrator sustava. Postavljanje trajnih replika najčešće radi vlasnik podataka, a skupovi trajnih replika su najčešće malobrojni. Osnovna operacija koja se vrši nad trajnim replikama je njihovo čitanje, tj. dohvatanje. Primjer korištenja ove vrste replika kod weba je prilikom raspodjele opterećenja između poslužitelja na istoj geografskoj lokaciji i raspodjele opterećenja zrcaljenjem (engl. *mirroring*) replika na različitim geografskim lokacijama.

Pristup replikacijskim poslužiteljima ostvaren je putem raspoređivača zahtjeva kojem je poznata raspodjela trajnih replika po replikacijskim poslužiteljima. Raspoređivač prima korisničke zahtjeve za resursima te ih dalje proslijeđuje odgovarajućim poslužiteljima u grozdu. Osim toga, raspoređivač prima i odgovore od poslužitelja te ih proslijeđuje odgovarajućim korisnicima. Raspodjelom opterećenja po poslužiteljima postiže se njihovo ravnomjerno i učinkovito opterećenje. U slučaju ispada nekog od poslužitelja, raspoređivač može odabrati zamjenskog poslužitelja čime se ostvaruje visoka razina raspoloživosti i pouzdanosti sustava. Bitno je naglasiti da raspoređivač vrši odabir poslužitelja dinamički, ovisno o odabranoj strategiji raspodjele opterećenja, iako je sam sustav statičan i „ručno“ postavljen.

Slika 8.13 prikazuje jednostavni primjer uporabe trajnih replika u sustavu s četiri replikacijska poslužitelja ( $r_1, r_2, r_3, r_4$ ) u grozdu poslužiteljskih računala.



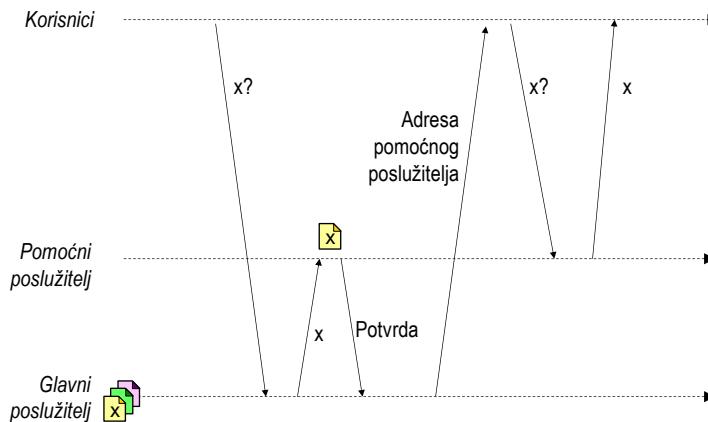
Slika 8.13. Primjer uporabe trajnih replika

Raspoređivač raspoređuje korisničke zahtjeve za replikom  $x$  na poslužitelje  $r_1$  i  $r_3$  koji je sadrže. Raspoređivač predstavlja posrednika u komunikaciji između korisnika i replikacijskih poslužitelja te vrši raspodjelu opterećenja između poslužitelja u grozdu, u ovom primjeru upućujući ih naizmjenično replikacijskim poslužiteljima  $r_1$  i  $r_3$ .

### 8.2.2 Poslužiteljske replike

Replike potaknute od strane poslužitelja, skraćeno poslužiteljske replike, su dinamički, u stvarnom vremenu stvorene replike prilikom povećanja potražnje za nekim podatkovnim objektom. U ovom

slučaju, izvorni, tj. glavni replikacijski poslužitelj prati svoje vlastito opterećenje i dinamički započinje postupak stvaranja poslužiteljskih replika i njihovo proslijedivanje privremenim, tj. pomoćnim poslužiteljima koji su bliži korisnicima. Ovako se postiže dinamička i potpuno automatizirana replikacija. Nedostatak je povećano vrijeme odziva tijekom stvaranja novih replika. Međutim, ovom vrstom replikacije se smanjuje opterećenje izvornog poslužitelja i buduće vrijeme odziva cjelokupnog sustava. Na slici (Slika 8.14) prikazan je primjer uporabe poslužiteljskih replika.



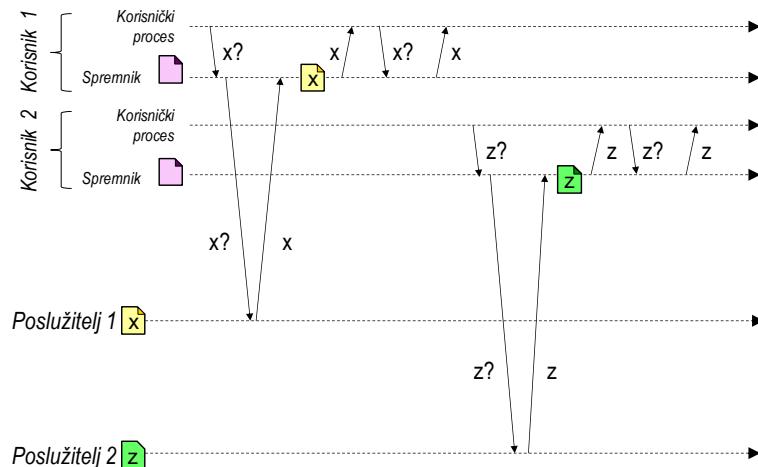
Slika 8.14. Primjer uporabe poslužiteljskih replika

Prilikom korisničkog zahtjeva za resursom  $x$  glavni poslužitelj ocjenjuje da postoji potreba za repliciranjem jako korištenog resursa  $x$  na pomoćnom poslužitelju koji je bliži korisnicima. Stoga glavni poslužitelj pokreće proces replikacije i dostavlja resurs  $x$  pomoćnom poslužitelju koji mu odgovara potvrdom uspješne replikacije. Nakon toga glavni poslužitelj obaveštava klijenta da se resurs  $x$  nalazi kod pomoćnog poslužitelja i šalje mu njegovu adresu. Klijent zatim šalje zahtjev za resursom  $x$  pomoćnom poslužitelju koji mu odgovara proslijednjem replike traženog resursa.

Poslužiteljske replike se često koriste prilikom geografske raspodjele opterećenja na webu.

### 8.2.3 Korisničke replike

Replike koje su pohranjene u klijentskim priručnim spremišta nazivaju se replikama potaknutim od strane korisnika ili skraćeno korisničkim replikama. Prilikom dohvaćanja resursa s originalnih poslužitelja, oni se spremaju u lokalna priručna spremišta iz kojih se dohvaćaju u slučaju ponovne potrebe za njima. Lokalni spremnik pritom može biti na istom računalu kao i korisnički programi ili na nekom drugom (dijeljenom) računalu u mreži. Dijeljenjem spremnika između većeg broja korisnika povećava se vjerojatnost pronalaska traženog resursa u spremniku. Na slici (Slika 8.15) korisnički procesi 1 i 2 dohvaćaju resurs  $x$ , odnosno resurs  $z$ .



Slika 8.15. Primjer uporabe korisničkih replika

U oba slučaja se replika pohranjuje u lokalnom spremniku te se iz njega dohvaća prilikom uzastopnog ponovnog zahtjeva za istim resursom.

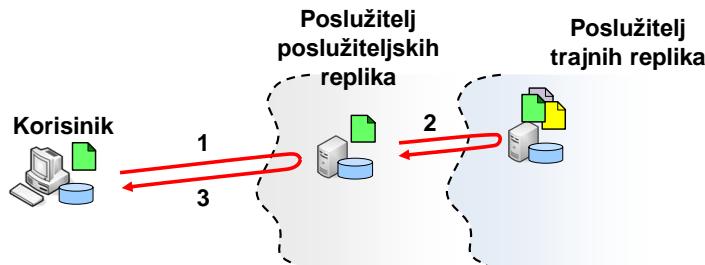
Ovu vrstu replikacije često koriste davatelji internetske usluge radi smanjenja mrežnog prometa prema vanjskim poslužiteljima weba. Smanjenje odziva za korisnike je pri tome izravna posljedica ove vrste replikacije. Osim toga, svi preglednici weba posjeduju lokalne spremnike u koje pohranjuju korisničke replike, prvenstveno radi podrške opcije povratka unatrag, na prethodno dohvaćeni sadržaj.

### 8.3 Metode održavanja konzistentnosti replika

Korisničke i poslužiteljske replike je potrebno usklađivati s promjenama stanja njihovih izvornih trajnih replika. Usklađivanje sadržaja replika može biti ostvareno u trenutku prije ostvarivanja pristupa replici ili u trenutku nakon promjene njihova sadržaja. Ovisno o tome razlikujemo metode dohvaćanja (*pull*) promjena sadržaja replika i prosljeđivanja (*push*) promjena sadržaja replika.

#### 8.3.1 Dohvaćanje promjena sadržaja replika

Na slici (Slika 8.16) prikazan je primjer dohvaćanja replike metodom *pull*. U trenutku kad klijent zahtijeva pristup resursu, on se prvo pokušava dohvatiti iz njegova lokalnog spremnika. Ako se resurs nalazi u lokalnom spremniku, prije dostavljanja resursa korisniku mora se utvrditi je li u međuvremenu došlo do promjene njegova sadržaja. S obzirom da poslužiteljska replika također ne mora biti ažurirana, zahtjev za provjeru se nadalje prosljeđuje poslužitelju trajnih replika. Ako je resurs u međuvremenu promijenjen, poslužitelj trajnih replika prosljeđuju novu verziju resursa poslužitelju poslužiteljskih replika koji obnavlja stanje svojih replika te prosljeđuju novu inačicu klijentu. U lokalnom spremniku se zatim obnavlja stanje korisničke replike te se sadržaj prikazuje korisniku. U slučaju da nije došlo do promjene sadržaja replike, poslužitelj trajnih replika odgovara negativnom potvrdom poslužitelju poslužiteljskih replika, koji zatim prosljeđuje negativnu potvrdu klijentu te se na kraju korisniku prikazuje resurs iz njegova lokalnog spremnika.

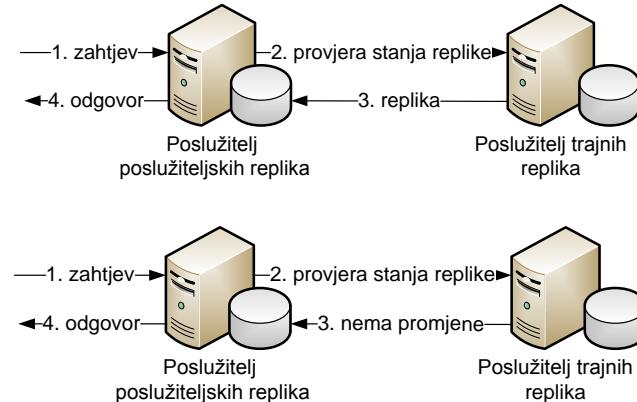
Slika 8.16. Primjer dohvaćanja replike metodom *pull*

Na slici (Slika 8.17) detaljnije je prikazana interakcija između poslužitelja trajnih i poslužitelja poslužiteljskih replika pri korištenju metode *pull*. Prilikom provjere sadržaja replike moguća su dva slučaja: replika je u međuvremenu promijenjena (gornji slučaj) ili replika nije promijenjena (donji slučaj).

Učinkovitost ove metode ocijenit će se na temelju analize odnosa učestalosti čitanja i učestalosti upisivanja (promjene) replike. Ako je frekvencija čitanja  $f_r$  veća od frekvencije upisivanja  $f_w$  ( $f_r > f_w$ ) gornji slučaj se javlja s frekvencijom  $f_w$ , a donji s frekvencijom  $f_r - f_w$ .

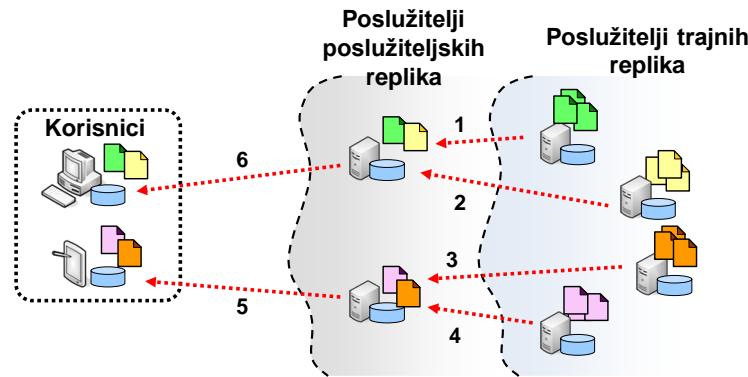
Međutim, ako je  $f_r \leq f_w$ , vjerojatnost donjeg slučaja teži nuli te time frekvencija gornjeg postaje  $f_r$ . Ova situacija podsjeća na situaciju bez replikacije zato što se gotovo za svaki korisnički zahtjev za replikom dohvata nova inačica.

Stoga je prilikom replikacije podataka, metodu *pull* bolje koristiti kad je frekvencija promjene replika (pisanja) veća od frekvencija čitanja replika, jer je u tom slučaju neefikasno prosljeđivati promjene sadržaja za koje je malo vjerojatno da će biti pročitane. Nedostatak metode *pull* je povećanje vremena odziva u slučajevima kada lokalno stanje replike nije obnovljeno pa se ona mora dohvatiti prije isporuke korisniku.

Slika 8.17. Interakcija između poslužitelja kod metode *pull*

### 8.3.2 Prosljeđivanje promjena sadržaja replika

Slika 8.18 prikazuje primjer dohvaćanja replike metodom *push*. U trenutku promjene sadržaja trajnih replika, poslužitelji trajnih replika prosljeđuju promjene sadržaja poslužiteljima poslužiteljskih replika. Oni obnavljaju stanje poslužiteljskih replika te zatim prosljeđuju promjene klijentima koji također obnavljaju stanje svojih korisničkih replika. Tijekom rada sustava, korisnici uvijek dohvaćaju samo sadržaj spremlijen u lokalnom spremniku i nikada ne šalju upite poslužiteljima te se na ovaj način postiže minimalno vrijeme odziva raspodijeljenog sustava replika.

Slika 8.18. Primjer dohvaćanja replika metodom *push*

U praksi postoje tri postupka prosljeđivanja promjena sadržaja:

- prosljeđivanje novog sadržaja,
- prosljeđivanje operacija za promjenu sadržaja te
- prosljeđivanje obavijesti o promjenama sadržaja.

Kod prvog postupka može se proslijediti samo izmijenjeni dio sadržaja replike ili njezin cijelokupni sadržaj.

Za razliku od prvog postupka, drugi se temelji na slanju operacija koje je potrebno obaviti nad replikom da bi se ona dovela u konzistentno stanje. Zbog toga se to ne može koristiti u slučajevima kada se izmijenjeno stanje replike ne može opisati slijedom operacija kao što je primjerice čest slučaj kod sinkronizacije digitalnih fotografija. Dodatni nedostatak ovog postupka je povećano procesiranje koje se javlja na strani replikacijskih poslužitelja prilikom provođenja operacija usklađivanja replika.

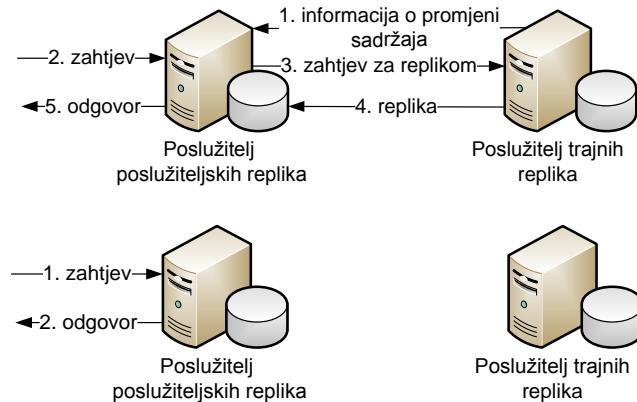
Treći postupak podsjeća na osnovnu metodu *pull*, jer se usklađivanje sadržaja replika vrši u trenutku prije ostvarivanja pristupa replici. Međutim, u tom trenutku je već poznato treba li ažurirati repliku ili ne te zato nije potrebno vršiti dodatnu provjeru kao kod metode *pull*.

Na slici (Slika 8.19) prikazana je interakcija između poslužitelja trajnih i poslužiteljskih replika korištenjem metode prosljeđivanja novog sadržaja ili prosljeđivanja operacija za promjenu sadržaja. U oba slučaja korisnički zahtjevi dolaze frekvencijom čitanja  $f_r$ , a poslužitelj trajnih replika frekvencijom pisanja  $f_w$  prosljeđuje novi sadržaj replike ili operacije za promjenu sadržaja.



Slika 8.19. Interakcija između poslužitelja kod metoda prosljeđivanja novog sadržaja i operacija za promjenom sadržaja replika

Za razliku od prethodne dvije metode kod kojih poslužitelj poslužiteljskih replika uvijek ima ažuriran sadržaj replike, kod metode prosljeđivanja obavijesti o promjenama sadržaja potrebno je ažurirati repliku u slučaju da je u međuvremenu došlo do promjene njenog sadržaja. Drugim riječima, moguća su dva slučaja: replika je u međuvremenu promijenjena ili replika nije promijenjena. Slika 8.20 prikazuje ova dva slučaja prilikom interakcije poslužitelja trajnih i poslužiteljskih replika. Informacije o promjeni sadržaja se prosljeđuju frekvencijom pisanja  $f_w$  te je to ujedno i frekvencija prvog (gornjeg) slučaja. Iz tога nadalje proizlazi da se drugi (donji) slučaj se događa frekvencijom  $f_r - f_w$ .



Slika 8.20. Interakcija između poslužitelja kod metode proslijedivanja informacija o promjeni sadržaja replika

Metodu *push* je bolje koristiti u slučajevima kada je frekvencija čitanja veća od frekvencije upisivanja jer bi se korištenjem metode *pull* prvi svakom čitanju bespotrebno provjeravalo je li došlo do promjene stanja replike. Iznimka je metoda proslijedivanja informacija o promjeni sadržaja replika koja se može koristiti neovisno o odnosu frekvencije čitanja i upisivanja replika. Ipak, zbog dodatnih poruka ova metoda generira malo više mrežnog prometa nego metoda proslijedivanja novog sadržaja replika u slučaju  $f_r > f_w$  i također malo više mrežnog prometa nego metoda *pull* u slučaju  $f_r \leq f_w$ . Međutim, zbog svoje univerzalne mogućnosti primjene često se primjenjuje praksi. Nedostatak metode *push* u odnosu na metodu *pull* je u tome što poslužitelji trajnih replika moraju imati zabilježene adrese svih poslužitelja poslužiteljskih replika te opise stanja njihovih replika. Analogno, poslužitelji poslužiteljskih replika moraju imati zabilježene adrese svih klijenata koji repliciraju njihove replike i opise stanja njihovih korisničkih replika.

## 8.4 Protokoli za ostvarivanje operacija čitanja i upisivanja replika

U prethodnom razmatranju metoda održavanja konzistentnosti replika prepostavili smo da se sve promjene replika događaju na strani poslužitelja trajnih replika i da su inicirane od strane vlasnika podataka. Međutim, u praksi se vrlo često događa da su promjene replika inicirane od strane samih korisnika raspodijeljenog sustava. U ovom poglavlju se bavimo protokolima koji omogućuju ostvarivanje operacija čitanja i pisanja replika od strane korisnika. Ovi protokoli se dijele na protokole s aktivnom i protokole s pasivnom replikacijom.

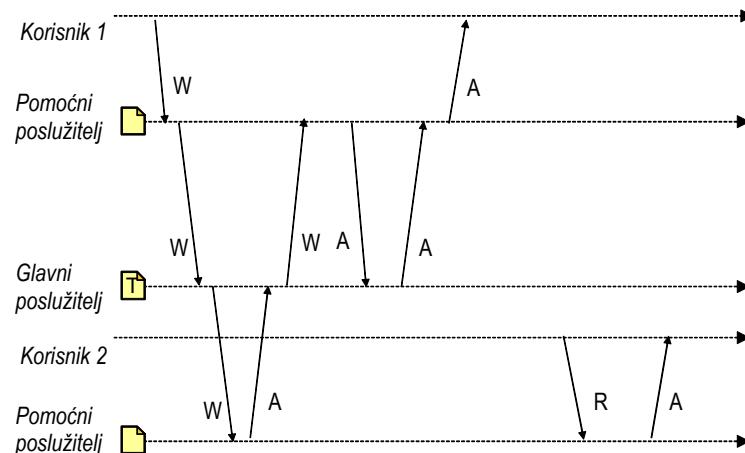
### 8.4.1 Pasivna replikacija

Protokoli s pasivnom replikacijom razlikuju glavnu (trajnu) repliku od njenih pomoćnih replika. Postoji samo jedna glavna replika resursa u sustavu i ona je smještena na glavnom replikacijskom poslužitelju te replike. Ostali replikacijski poslužitelji koji održavaju pomoćne replike ovog resursa nazivaju se pomoćnim poslužiteljima. Sve operacije pisanja se provode nad glavnom replikom te se zatim proslijeduju pomoćnim replikacijskim poslužiteljima radi obnove stanja njezinih pomoćnih replika. U slučaju ispada glavnog poslužitelja, bilo koji od pomoćnih poslužitelja može preuzeti ulogu glavnog poslužitelja. Operacije čitanja se međutim mogu provoditi nad bilo kojom replikom u sustavu, a najčešće se provode nad replikom koja je najbliža korisniku.

Postoje dva osnovna protokola pasivne replikacije u raspodijeljenim sustavima. Prvi protokol prepostavlja statičnost glavne replike i temelji se na proslijedivanju zahtjeva za pisanje prema njenom glavnom poslužitelju. Nakon uspješnog obavljanja operacije pisanja, glavni poslužitelj proslijede promjene svim pomoćnim poslužiteljima ove replike. Kako se obnavljanje stanja replike provodi kod njenog glavnog poslužitelja ovaj protokol se naziva udaljeno obnavljanje stanja replike. Kod drugog protokola glavna replika je dinamična te se prilikom operacije pisanja najprije dohvata na

najbliži replikacijski poslužitelj. Promjene se zatim prosljeđuju pomoćnim poslužiteljima ove replike. Ovaj protokol se naziva lokalno obnavljanje stanja replike jer se operacije pisanja provode nad lokalnim replikacijskim poslužiteljem korisnika.

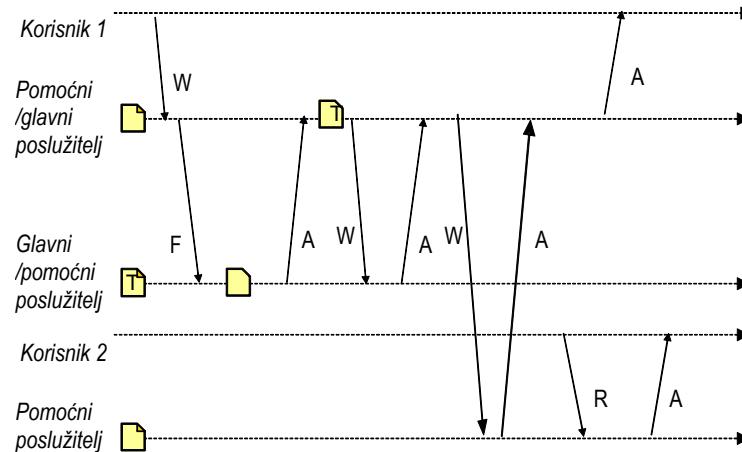
Na slici (Slika 8.21) prikazan je primjer udaljenog obnavljanja stanja replike. Korisnik 1 započinje operaciju upisivanja (W) prosljeđujući zahtjev lokalnom pomoćnom poslužitelju. On prima zahtjev i prosljeđuje ga replikacijskom poslužitelju koji u svom lokalnom spremniku sadrži glavnu repliku (T). Nakon primitka zahtjeva, glavni poslužitelj replike obnavlja stanje glavne replike i prosljeđuje zahtjev za pisanje svim replikacijskim poslužiteljima koji sadrže pomoćne replike. Oni primaju i provode operacije upisivanja nad replikama u svojim lokanim spremnicima. Nakon obnavljanja stanja, pomoćni poslužitelji prosljeđuju potvrde (A) glavnom poslužitelju koji zatim korisniku koji je započeo operaciju upisivanja prosljeđuje potvrdu o uspješno obavljenoj operaciji. Korisnik 2 koji samo čita repliku kojom raspolaže njegov pomoćni poslužitelj isto tako dobiva potvrdu obnavljanje replike.



Slika 8.21. Primjer udaljenog obnavljanja stanja replike

Glavna prednost udaljenog obnavljana stanja replike je uspostava slijedne konzistentnosti obzirom da se operacije pisanja provode samo nad glavnom replikom. Na ovaj način svi korisnici doživljavaju jednak redoslijed izvođenja operacija pisanja u vremenu neovisno o replici koju čitaju. Osnovni nedostatak ovog protokola je produljeno trajanje izvođenja operacije upisivanja nad replikom zbog potrebe za prosljeđivanjem i provođenjem operacija upisivanja nad svim pomoćnim replikama u sustavu.

Primjer lokalnog obnavljanja stanja replika prikazan je na slici (Slika 8.22). Korisnik 1 upućuje zahtjev za provođenje operacije upisivanja (W) nad replikom svom lokalnom pomoćnom poslužitelju. Kako on ne sadrži glavnu repliku (T), prosljeđuje zahtjev (F) i dohvata ju s glavnog replikacijskog poslužitelja. Pri tome se glavna replika pretvara u običnu i poslužitelj mijenja ulogu u „pomoćni poslužitelj“, a dohvaćena replika u glavnu, na pomoćnom poslužitelju koji preuzima ulogu „glavnog poslužitelja“. Nakon obnavljanja stanja glavne replike, lokalni poslužitelj korisnika 1 prosljeđuje operacije pisanja replikacijskim poslužiteljima koji sadrže pomoćne replike. Nakon što svi pomoćni poslužitelji pošalju potvrde korisnikovom spremniku, on korisniku prosljeđuje potvrdu o uspješno obavljenoj operaciji pisanja.



Slika 8.22. Lokalno obnavljanje stanja replika

Glavna prednost lokalnog obnavljanja stanja replike je to što se uzastopne operacije pisanja mogu provesti vrlo brzo nad lokalnim replikacijskim poslužiteljem. Osim toga, rezultati provođenja uzastopnih operacija pisanja mogu biti agregirani u jednu operaciju koja se zatim može provesti nad svim pomoćnim replikama u sustavu.

#### 8.4.2 Aktivna replikacija

Kod aktivne replikacije operacije upisivanja se provode odjednom i jednakim redoslijedom nad svim replikama u raspodijeljenom sustavu. Da bi se podržala takva vrsta provođenja operacija upisivanja, raspodijeljeni sustav treba podržavati potpuno uređeno višeodredišno razašiljanje (engl. *totally ordered multicast*). Višeodredišno razašiljanje omogućuje proslijedjivanje operacija pisanja samo odgovarajućim replikacijskim poslužiteljima, a potpuna uređenost omogućava da svi replikacijski poslužitelji vide jednak redoslijed operacija pisanja (detaljno je objašnjeno u poglavlju 9.2.3). Na ovaj način se postiže slijedna konzistentnost operacija. Dvije su moguće implementacije ove vrste višeodredišnog razašiljanja. Prva se temelji na primjeni logičkih oznaka vremena, a druga na uvođenju središnjeg koordinatora koji vodi brigu o redoslijedu izvođenja operacija pisanja te time uvelike podsjeća na pasivnu replikaciju.

Aktivna replikacija se izvodi na sljedeći način. Prvo se svaki korisnički zahtjev za upisivanjem upućuje u raspodijeljeni sustav. Svi zahtjevi se putem potpuno uređenog višeodredišnog razašiljanja proslijeduju jednakim redoslijedom svim poslužiteljima te replike. Nakon toga svi poslužitelji provode operaciju upisivanja te potvrđuju pokretaču operacije.

### 8.5 Pitanja za učenje i ponavljanje

- 8.1. Objasnite što je replika podatka, a što je nekonzistentnost replike podatka.
- 8.2. Objasnite što je povezana konzistentnost operacija u raspodijeljenim sustavima? Na primjeru procesa p, q i r prikažite slijed operacija čitanja i pisanja koji je a) u skladu i b) nije u skladu s načelima povezane konzistentnosti.
- 8.3. Raspodijeljeni sustav uključuje tri računala ( $R_0, R_1, R_2$ ) s lokalnim spremnicima. U lokalnom spremniku računala  $R_1$  nalazi se trajna replika dokumenta, dok se u lokalnom spremniku računala  $R_2$  nalazi obična replika dokumenta. Korisnik putem računala  $R_3$  provodi operaciju pisanja nad dokumentom primjenom postupka *lokalnog obnavljanja stanja replike*. Skicirajte

i objasnite korake postupka.

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



- 8.4. U sustavu replika koji se sastoji od glavnog poslužitelja i n=4 podjednako opterećena pomoćna poslužitelja, odredite metodu održavanja konzistentnosti replika za koju će prosječno mrežno (prometno) opterećenje poslužitelja L biti najmanje. Pri tome pretpostavite da korisnike isključivo poslužuju pomoćni poslužitelji, da je prosječna frekvencija upita fu=5 upita/s, prosječna frekvencija promjena fp=1 promjena/min te da su prosječne veličine upita/odgovora, operacija za promjenu sadržaja i replika lp=1kB, lo=50 kB i lr=100 kB. Usporedite dobivena opterećenja s centraliziranim slučajem kada korisnike poslužuje glavni poslužitelj.

## 9 OTPORNOST NA NEISPRAVNOSTI

Otpornost na neispravnosti (engl. *fault tolerance*) je iznimno značajno svojstvo za raspodijeljene sustave jer sustav ili poslužitelj otporan na neispravnosti treba i dalje treba biti dostupan i nuditi svoje usluge bez obzira na ispade procesa ili izgubljene poruke. Pri tome funkcionalnost sustava može biti ograničena uz negativan utjecaj na njegove performance, no važno je da je usluga i dalje dostupna i funkcionalna. Otpornost na neispravnosti se definira kao sposobnost sustava za obavljanje definirane usluge bez obzira na postojeće *neispravnosti* koje izazivaju *ispad* nekih komponenti sustava.

Raspodijeljeni sustav, kao svaki programski sustav, u sebi krije neispravnosti koje ljudsko djelovanje unosi u programski kôd ili koje je posljedica pogrešaka u komunikacijskoj mreži. Neispravnosti (engl. *fault*) se manifestiraju kroz ispade sustava (engl. *failure*). Neispravnost je primjerice dio programskog kôda sa skrivenom pogreškom (engl. *bug*) koja može npr. prouzročiti nedostupnost poslužitelja koji se nalazi u neregularnom blokirajućem stanju. Neispravnost sustava može biti posljedica ljudske pogreške prilikom oblikovanja sustava kada je npr. komunikacijski protokol pogrešno specificiran. Osim toga neispravnost može biti vezana uz komunikacijsku mrežu kada zbog npr. prekinutog komunikacijskog kanala poslužitelj postaje nedostupan ili je poruka na kanalu izgubljena ili pogrešno prenesena. Pronalaženje neispravnosti u programskom dijelu raspodijeljenog sustava je težak i važan zadatak prilikom razvoja sustava koji se u najvećoj mjeri provodi tijekom njegovog testiranja.

Ispad sustava je stanje sustava koje se detektira kroz nemogućnost korištenja jedne ili više njegovih usluga. Ono je uvijek posljedica neispravnosti, bilo programskog kôda ili komunikacijske mreže. Stoga ispade možemo podijeliti na ispade procesa i ispade kanala:

- Kod ispada procesa, proces ne mijenja stanja premda se ne nalazi u završnom stanju i riječ je tzv. ispadu zaustavljanja ili generira proizvoljne izlaze pa je riječ o tzv. bizantinskom ispadu. Pregled različitih vrsta ispada dan je u tablici (Tablica 9.1).
- Kod ispada kanala događa se sljedeće: proces *p* je poslao poruku procesu *q*, ali *q* poruku ne prima, jer npr. kanal gubi poruke.

Tablica 9.1. Pregled različitih vrsta ispada procesa

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

S obzirom da je ispade sustava nemoguće u potpunosti izbjegći (npr. iscrpnim testiranjem je moguće smanjiti vjerojatnost ispada programskog kôda), razvijatelji raspodijeljenog sustava koriste posebne tehnike kojima će smanjiti utjecaj ispada jedne od njegovih komponenti na cjelokupni sustav i također definiraju procedure za oporavak sustava nakon ispada te njegov povratak u regularno radno

stanje. U nastavku se razmatraju tehnike kojima je moguće prikriti neispravnosti procesa, omogućiti pouzdanu komunikaciju, osigurati izvođenje transakcija te oporavak sustava nakon ispada.

## 9.1 Otpornost procesa na ispade

Ključna tehnika za prikrivanje neispravnosti raspodijeljenog sustava je **redundancija**. Redundancija ili zalihost uvodi dodatne elemente i opremu u sustav te je riječ o fizičkoj redundanciji, ili ponavlja operacije u vremenu, pa govorimo o vremenskoj redundanciji.

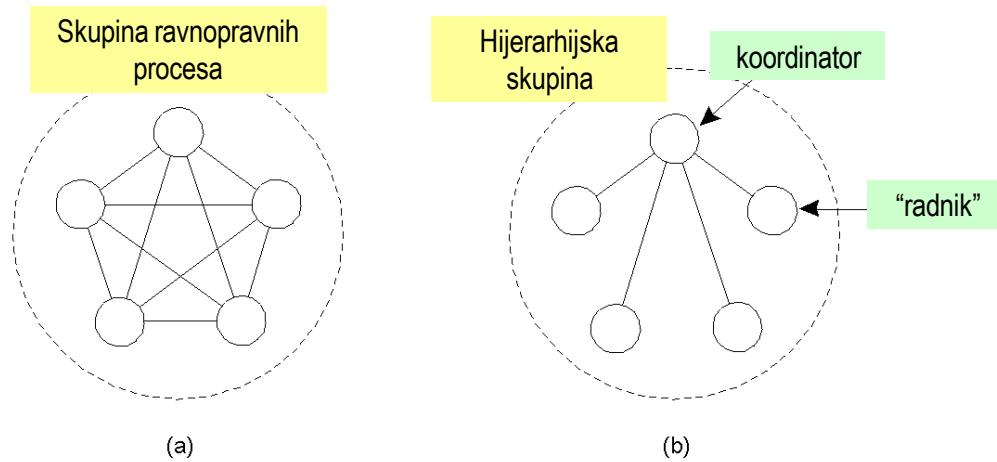
U raspodijeljenim sustavima fizička redundancija se uvodi na nivou procesa te se jedan proces zamjenjuje skupinom identičnih procesa (replika) stoga što bilo koji proces iz skupine, pa čak i jedan jedini, može preuzeti sve korisničke zahteve i osigurati kontinuiranost usluge. Svi procesi iz skupine virtualno čine jedinstveni proces te su na taj način percipirani od strane okoline. Stoga bi svi procesi iz skupine procesa morali sve poruke iz okoline primiti istovremeno (kao da je riječ o sinkronom sustavu) kako bi u svakom trenutku bili u istom stanju. Ovo je naravno teško realizirati u realnosti i stoga se koristi tehnika tzv. virtualne sinkronosti koja će biti objašnjena u nastavku.

Prije svega je potrebno analizirati koliko je procesa potrebno u skupini replika radi prikrivanja ispada  $k$  procesa iz skupine. Otpornost na  $k$  ispada je svojstvo koje osigurava da skupina procesa može "preživjeti" ispad najviše  $k$  procesa tako da prema okolini ostane potpuno funkcionalna uz potencijalnu degradaciju performansi. Analizirajmo koliko je procesa nužno da se osigura otpornost na  $k$  ispada:

- dovoljan je samo  $k + 1$  proces da se osigura tolerancija na  $k$  ispada jer jedan proces može preuzeti poslove cijelokupne skupine,
- potrebno je  $2k + 1$  procesa u skupini ako se prepostavi  $k$  bizantskih ispada. Ovdje se koristi mehanizam glasovanja tako da će  $k + 1$  ispravan proces "nadglasati"  $k$  neispravnih.

### 9.1.1 Organizacija skupine procesa

Skupine procesa mogu biti organizirane na dva načina kako je prikazano slikom (Slika 9.1): 1) kao skupina ravnopravnih procesa u kojoj su svi procesi jednaki ili 2) kao hijerarhijska skupina u kojoj postoji jedan poseban proces „koordinator“ skupine, dok su ostali tzv. radni procesi.



Slika 9.1. Organizacija skupine procesa a) ravnopravni procesi i b) hijerarhijska skupina

Skupina s ravnopravnim procesima uključuje procese koji kolektivno donose odluke. Prednost ove organizacije skupine procesa je u tome što ne postoji jedinstvena točka ispada jer su svi procesi jednakopravni, a očiti nedostatak su kompleksni protokoli za komunikaciju s procesima u skupini i složene procedure za donošenje zajedničkih odluka. Prilikom prijenosa poruke do ovakve skupine procesa potrebno je koristiti npr. pouzdanu višeodredišnu komunikaciju koja je objašnjena u

poglavlju 9.2. Hijerarhijske skupine procesa su centralizirane jer je jedan proces koordinator, a ostali su radni procesi. Kada se generira poruka za skupinu, bilo od vanjskog procesa ili jednog od radnih procesa, ona se uvijek šalje do koordinatora koji poruku prosljeđuje svim radnim procesima. Koordinator se također može koristiti za raspoređivanje dolaznih zahtjeva u skupini procesa. Nedostatak ovog rješenja je postojanje jedinstvene točke ispada jer je koordinatora koji je u ispadu potrebno zamijeniti drugim procesom (najprije treba otkriti da je koordinator u ispadu, a potom pokrenuti protokol za izbor novog koordinatora skupine). Osnovna prednost hijerarhijske skupine procesa je svakako jednostavnost protokola za komunikaciju sa skupinom te jednostavni postupci za donošenje odluka.

Sljedeće važno pitanje je kako pratiti stanja procesa u skupini te bilježiti ispade procesa iz skupine. Postoje dvije osnovne metode za otkrivanje ispada procesa iz skupine. Prva je proaktivna i temelji se na propitivanju stanja procesa. U ravnopravnoj skupini procesa svaki proces periodički šalje upit ostalim procesima i provjerava njihovo stanje ("are you alive?") te bilježi odgovore na temelju kojih zaključuje koji su procesi u ispadu. U hijerarhijskoj skupini ovu proceduru izvodi samo koordinator skupine. Druga metoda za otkrivanje ispada procesa iz skupine je reaktivna jer se temelji na pasivnom praćenju primljenih poruka od ostalih članova skupine. Na taj način proces ili koordinator zaključuje koji su procesi aktivni s obzirom da šalju poruke, a koji su potencijalno u ispadu. Moguće je koristiti i kombinaciju navedenih metoda tako da se nakon reaktivnog praćenja stanja procesa aktivno šalju upiti do procesa koji su potencijalno u ispadu.

Skupine procesa su dinamične. Proces se može priključiti skupini ili je napustiti tijekom rada skupine, a isti proces može biti član više skupina. Isto tako je moguće da tijekom rada dođe do ispada nekog procesa. Stoga je za funkcioniranje skupine procesa kao virtualno jedinstvenog procesa nužna usluga administriranja skupine procesa (engl. *group membership*) koja omogućuje 1) kreiranje skupine, 2) dodavanje procesa u skupinu, 3) izlazak procesa iz skupine i 4) komunikaciju okoline sa skupinom procesa. Za realizaciju komunikacije okoline sa skupinom procesa potrebno je osigurati isporuku poruke svim članovima skupine. S obzirom da vanjski proces poznaje samo jedinstvenu adresu skupine te ne zna identifikatore pojedinih procesa u skupini, usluga za administriranje članova skupine se brine o isporuci poruka te o redoslijedu isporuke poruka ispravnim procesima skupine.

Najjednostavnije rješenje za realizaciju usluge administriranja je poseban poslužitelj (engl. *group server*) koji prima sve zahtjeve iz okoline te prati stanje procesa u skupini, ali je nedostatak ovog rješenja taj da poslužitelj predstavlja jedinstvenu točku ispada. Bolje je rješenje upravljati grupom procesa na raspodijeljeni način primjenom npr. pouzdane višeodredišne komunikacije.

### 9.1.2 Sporazum skupine procesa

Sporazum skupine procesa (engl. *group agreement*) je postupak kojim skupina postiže suglasnost o nekom za skupinu važnom pitanju, npr. o vrijednosti neke varijable ili o tome hoće li ili neće izvršiti transakciju. Sporazum je potreban za cijeli niz praktičnih aplikacija, npr. kada treba odabrati koordinatora skupine, raspodijeliti zahtjeve među skupinom procesa ili odlučiti o stanju transakcije. Sporazum procesa uključuje komunikaciju i razmjenu informacija između procesa u skupini radi njihove koordinacije, a prije izvođenja neke buduće akcije. Tipičan primjer je iz područja baza podataka kada procesi zajednički odlučuju hoće li izvršiti ili prekinuti transakciju.

U literaturi se posebno razmatra problem sporazuma skupine procesa ako se prepostavi da *k* procesa može biti u stanju bizantskog ispada, tzv. *problem bizantskog sporazuma* (engl. *Byzantine agreement problem*) (Lamport, Shostak, & Pease, 1982). Proces je u stanju bizantskog ispada kada na primljeni upit daje proizvoljan odgovor koji nije u skladu s ispravnim funkcioniranjem procesa, ili kolokvijalno rečeno „proces ne govori istinu“. Postavlja se pitanje koliko je ukupno procesa potrebno kako bi se postigla suglasnost o vrijednosti neke varijable.

Problem bizantskog sporazuma definiran je uz prepostavku *n* procesa u skupini od kojih je *k* neispravnih u stanju bizantskog ispada. Svaki proces *i* u skupini definira proizvoljno vlastitu vrijednost

$v_i$ , čime skupina procesa definira vektor vrijednost  $\mathbf{V} = [v_1, v_2, \dots, v_n]$ . Inicijalno svaki proces zna samo vrijednost za  $\mathbf{V}[i] = v_i$ , a skupina procesa treba postići sporazum o vektoru vrijednosti za sve procese iz skupine, tj. svi ispravni procesi trebaju na kraju izvođenja algoritma prihvati ispravne vrijednosti  $v_i$  za  $i$ -ti element vektora  $\mathbf{V}$  ako je proces  $i$  ispravan, a proizvoljne vrijednosti ako je proces neispravan u bizantskom ispadu. Algoritam završava ako svaki ispravan proces iz skupine u konačnici prihvati ispravne vrijednosti vektora za ispravne procese. Sažeti pregled problema bizantskog sporazuma dan je u nastavku:

**Problem:** svaki proces  $i$  definira inicijalnu vrijednost  $v_i$ , a skupina procesa treba postići sporazum o vektoru vrijednosti za svaki proces iz skupine

**Sporazum:** Svi ispravni procesi prihvaćaju isti vektor  $\mathbf{V} = [v_1, v_2, \dots, v_n]$ .

**Ispravnost:** Ako je proces  $i$  ispravan i definira vrijednost  $v_i$ , svi ostali ispravni procesi prihvaćaju vrijednost  $v_i$  kao  $i$ -ti element vektora. Ako je proces neispravan, ostali ispravni procesi mogu prihvati bilo koju vrijednost za taj proces.

**Završetak:** Svaki ispravan proces će u konačnici prihvati vrijednosti vektora.

Ideja algoritma za postizanje bizantskog sporazuma temelji se na sljedećem: svaki proces  $i$  šalje vrijednost  $v_i$  ostalim procesima u grupi, potom svaki popunjava vektor  $\mathbf{V} = [v_1, v_2, \dots, v_n]$  primljenim vrijednostima te u sljedećem koraku cjelokupni vektor šalje svim procesima u grupi. Na kraju svaki proces na temelju primljenih vektora zaključuje o ispravnim vrijednostima  $v_i$  na temelju većine vrijednosti na mjestu  $i$  svih primljenih vektora.

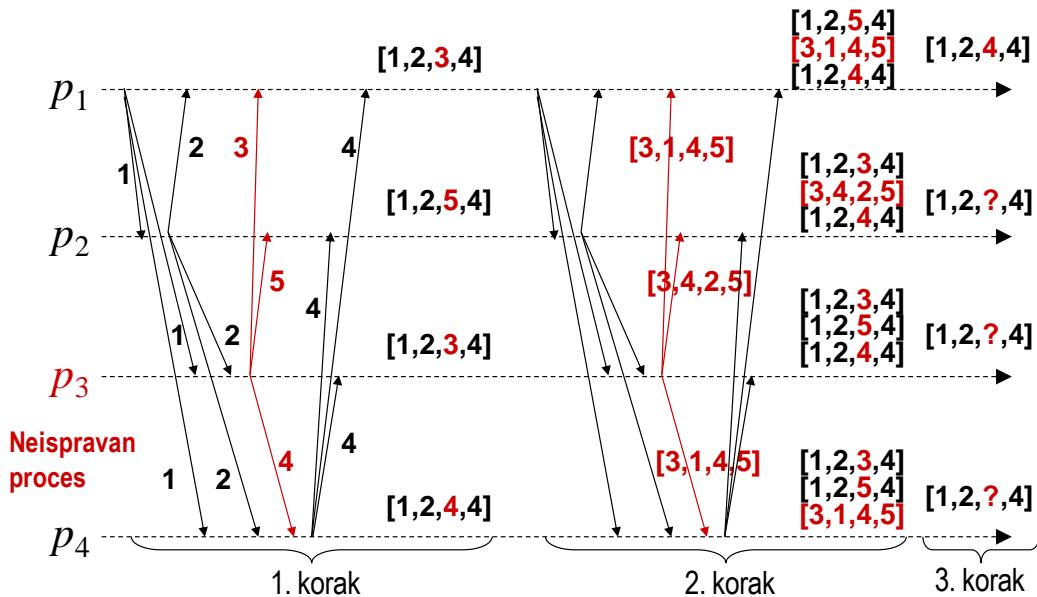
Važno je naglasiti da se ispravan završetak navedenog algoritma može garantirati samo uz prepostavku **sinkronog** modela skupine procesa, tj. komunikacija procesa se odvija u koracima s ograničenim trajanjem i urednim slijedom isporuke poruka te uz jednoodredišnu pouzdanu komunikaciju u skupini procesa.

Koraci algoritma za postizanje bizantskog sporazuma  $n$  procesa uz  $k=1$  su:

- **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 ostalim procesima svoj  $\mathbf{V}$ , dok neispravni proces šalje proizvoljni  $\mathbf{V}$ .
- **3. korak:** Konačno, svaki proces uspoređuje sve primljene vektore i odlučuje se za većinske vrijednosti. Pri tome procesi mogu donijeti ispravnu odluku samo o vrijednostima za ispravne procese, dok za neispravne procese ne mogu donijeti odluku ili donose proizvoljnu odluku.

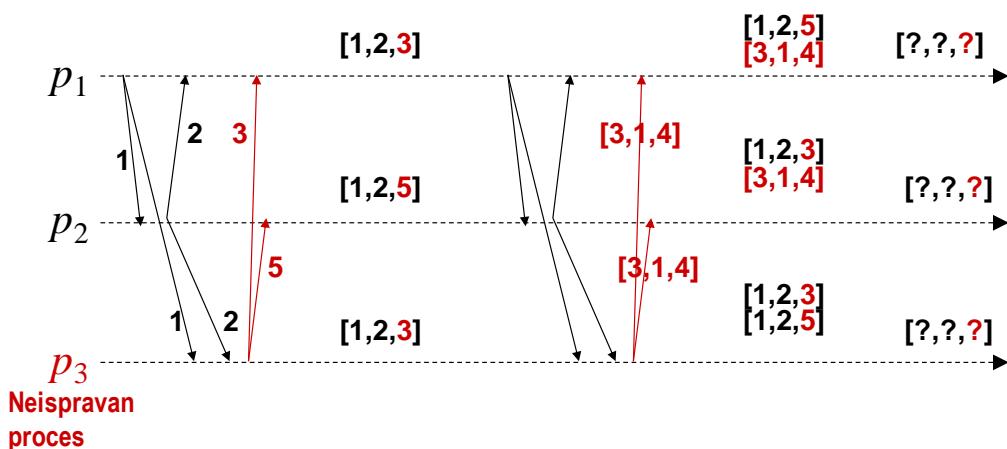
Slika 9.2 prikazuje primjer izvođenja algoritma za četiri procesa  $p_i$ ,  $1 \leq i \leq 4$ , od kojih je proces  $p_3$  neispravan i u stanju bizantskog ispada. U prvom koraku svi procesi šalju ostalim procesima vlastitu vrijednost  $i$ , dok proces  $p_3$  šalje proizvoljne vrijednosti (3, 5 i 4) procesima  $p_1$ ,  $p_2$ , i  $p_4$ . Na temelju primljenih vrijednosti svaki proces izgrađuje vlastiti vektor vrijednosti. U drugom koraku svaki proces šalje vektor vrijednosti ostalim procesima (npr. proces  $p_1$  šalje vektor [1, 2, 3, 4]), dok neispravni proces  $p_3$  šalje ostalim procesima proizvoljne vrijednosti vektora. U trećem koraku svaki proces na temelju primljenih vektora i većine vrijednosti na  $i$ -tom mjestu svih vektora zaključuje o vrijednostima vektora. Tako primjerice proces  $p_2$  donosi odluku da vektor izgleda ovako [1, 2, ?, 4] na temelju čega ima ispravne vrijednosti za procese  $p_1$ ,  $p_2$ , i  $p_4$ . Jednaku odluku o ispravnim procesima donosi i proces  $p_4$ , dok proces  $p_1$  zaključuje da su vrijednosti [1, 2, 4, 4] što je također u skladu sa zahtjevima algoritma jer su vrijednosti za ispravne procese ispravne (sporazum ispravnih procesa je postignut), a za neispravni proces je moguće prihvati bilo koju vrijednost, što je i u ovom slučaju

zadovoljeno. Dakle, procesi  $p_1$ ,  $p_2$  i  $p_4$  znaju ispravne vrijednosti za ispravne procese ( $p_1$ ,  $p_2$  i  $p_4$ ), a za neispravan proces  $p_3$  ispravni procesi donose proizvoljnu odluku ili je ne mogu donijeti.



Slika 9.2. Primjer bizantskog sporazuma za  $n = 4$  i  $k = 1$

Slika 9.3 prikazuje primjer izvođenja algoritma za tri procesa  $p_i$ ,  $1 \leq i \leq 3$ , od kojih je proces  $p_3$  neispravan i u stanju bizantskog ispadu. Koraci algoritma jednaki su kao u prethodnom slučaju. Ovdje vidimo da procesi ne mogu donijeti odluku na temelju većine vrijednosti primljenih vektora jer su potrebna dva vektora kako bi „nadglasali“ vektor neispravnog procesa. Stoga sporazum nije moguće postići i procesi samo mogu zaključiti da je neki od procesa u (bizantskom) ispadu.



Slika 9.3. Primjer nemogućnosti bizantskog sporazuma za  $n = 3$  i  $k = 1$

Na temelju prethodna dva primjera može se zaključiti da se sporazum u slučaju  $n$  procesa i  $k$  neispravnih procesa u bizantskom ispadu može postići samo kada vrijedi sljedeći uvjet:  $k \leq \left\lfloor \frac{n-1}{3} \right\rfloor$ .

Dakle, potrebno je  **$3k + 1$**  procesa u skupini za postizanje sporazuma ako se pretpostavi  $k$  bizantskih ispadova. (Lamport, Shostak, & Pease, 1982) su dokazali prethodnu tvrdnju te pokazali da se sporazum može postići samo u slučaju sinkrone i pouzdane komunikacije skupine procesa, dok za asinkroni sustav sporazum nije moguće postići niti za  $k=1$ .

## 9.2 Pouzdana komunikacija skupine procesa

Pouzdana komunikacija skupine procesa jamči isporuku poruka svim procesima u skupini. Ako pretpostavimo mogućnost ispada procesa koji čine skupinu procesa, pouzdana komunikacija garantira isporuku poruke svim ispravnim procesima u trenutku slanja poruke. Iz poglavlja 9.1 je vidljivo da je pouzdana komunikacija skupine procesa iznimno važna za ispravno funkcioniranje skupine te je također nužno osigurati i ispravnu slijednost isporučenih poruka svim procesima.

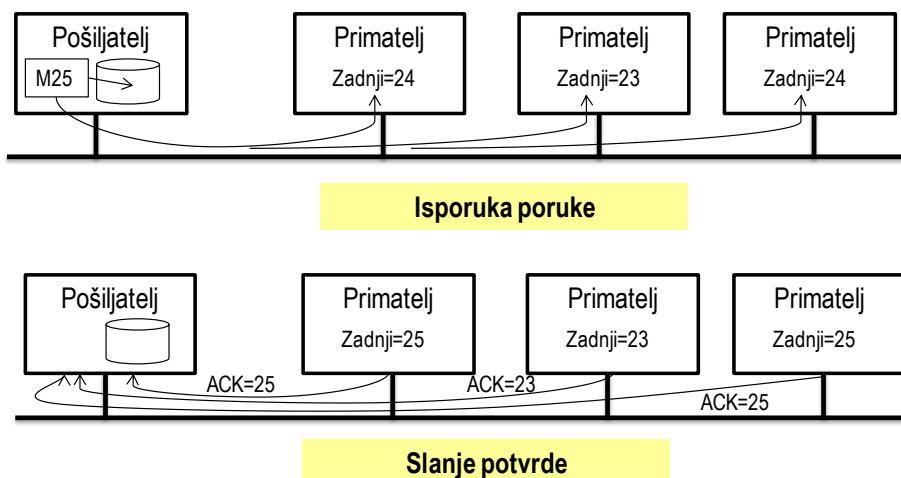
Najjednostavnija praktična implementacija pouzdane komunikacije u skupini procesa temelji se na pouzданoj jednoodredišnoj komunikaciji između svakog para procesa iz skupine koju nudi TCP. Učinkovita praktična implementacija koristi višeodredišno razašiljanje na mrežnom sloju (IP *multicast*) od jednog prema svim procesima u skupini. No oba rješenja, kako ono na temelju TCP-a pa tako i ono na temelju IP *multicast*-a, je potrebno nadograditi jer se uz pouzdanu komunikaciju vežu sljedeći problemi i pitanja na koja je potrebno dati odgovore:

- Koji procesi čine grupu u trenutku slanja poruke?
- Što se događa ako novi proces ulazi u grupu procesa dok je isporuka poruke grupi 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?

U nastavku se prvo razmatraju postojeća rješenja za pouzdanu višeodredišnu komunikaciju uz pretpostavku da se ispadi procesa ne mogu dogoditi, ali je moguć gubitak poruke tijekom komunikacije. Nakon toga se razmatraju rješenja koja prepostavljaju mogućnost ispada procesa. I posljednje se razmatraju različiti zahtjevi i ograničenja za slijednost isporučenih poruka članovima skupine.

### 9.2.1 Pouzdana komunikacija bez mogućih ispada procesa

Sljedeći algoritam omogućuje pouzdanu komunikaciju skupine od  $n$  procesa čiji su identifikatori poznati pošiljatelju poruka uz pretpostavku da procesi ne mogu biti u ispadu, dok je komunikacijski medij nepouzdan te su mogući gubici poruka i potvrda na kanalu.

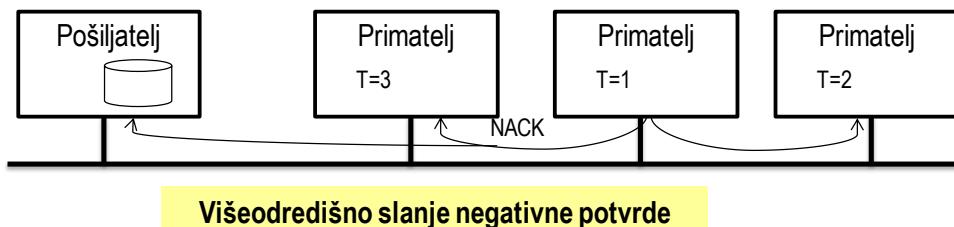


Slika 9.4. Višeodredišna pouzdana komunikacija s potvrdom

Jednostavno rješenje koje koristi nepouzdanu višeodredišnu komunikaciju (npr. IP *multicast*) je prikazano slikom (Slika 9.4). Pošiljatelj šalje poruku koristeći višeodredišno razašiljanje do skupine od 3 procesa, dodjeljuje oznaku svakoj poslanoj poruci i čuva poslane poruke u posebnom spremniku dok ne primi potvrde primitka od svih primatelja. Potvrde se šalju jednoodredišno od svakog primatelja do pošiljatelja. Prvi i treći primatelj šalju potvrde za uspješno primljenu poruku označenu s

25. Drugi primatelj opaža gubitak poruke jer se poruke šalju u slijedu i šalje zahtjev za ponavljanjem poruke 24 (ponavlja potvrdu za poruku 23). Nakon toga pošiljatelj može poruku 24 poslati direktno do drugog primatelja.

Ovo rješenje generira veliki broj potvrda i nije pogodno za velike skupine procesa jer generira  $n-1$  potvrda za svaku poslanu poruku te je neskalabilno.



Slika 9.5. Višeodredišna pouzdana komunikacija s potisnutom potvrdom

Slika 9.5 prikazuje alternativno rješenje za slanje potvrda, tzv. višeodredišnu pouzdanu komunikaciju s potisnutom potvrdom. Ono se temelji na višeodredišnom razašiljanju jedne negativne potvrde u slučaju kada primatelj nije primio poruku. U primjeru sa slike drugi pošiljatelj višeodredišno šalje poruku do ostalih primatelja i pošiljatelja. Pošiljatelj u ovom slučaju ne zna koliko dugo treba čuvati poslane poruke u spremniku jer ne prima potvrde i ne zna hoće li i kada primiti negativnu potvrdu. Stoga u praksi briše poruke iz spremnika nakon isteka nekog "razumnog" vremenskog perioda kada može zaključiti da je završena isporuka te je protekao vremenski period u kome bi se mogla poslati negativna potvrda. Negativne potvrde se šalju višeodredišno kako bi se osim pošiljatelja i ostali procesi koji potencijalno nisu primili navedenu poruku obavijestili da je slanje negativne potvrde u tijeku kako ne bi slali duplu negativnu potvrdu (potisnuta potvrda). Pošiljatelju je dovoljna jedna negativna potvrda nakon koje će ponovo poslati poruku svim članovima skupine koristeći višeodredišno razašiljanje.

Za opisano rješenje najveći problem je mogućnost paralelnog slanja negativne potvrde različitim primateljima što se rješava na način da prije razašiljanja negativne potvrde proces čeka neki vremenski period (trajanje čekanja se određuje slučajno, u primjeru su navedena vremena  $T=3$ ,  $T=1$  i  $T=2$ ).

### 9.2.2 Pouzdana komunikacija uz moguće israde procesa

Pouzdana komunikacija skupine procesa uz moguće israde procesa jamči isporuku poruke svim ispravnim i dostupnim procesima u skupini ili niti jednom. Pritom je potrebno osigurati i isporuku poruka u određenom redoslijedu.

Razmatraju se algoritmi za procese  $p_i$ ,  $1 \leq i \leq n$ , koji čine skupinu procesa  $G$ . U sustavu se prenose dvije vrste poruka:  $m$  je oznaka generirane poruke s korisnim sadržajem, a  $vc$  (engl. *view change*) oznaka poruke koja prenosi informaciju o dolasku ili odlasku procesa iz skupine.

U ovakovom sustavu su moguća dva različita scenarija s obzirom na mogućnost paralelnog prijenosa dvije vrste poruka:

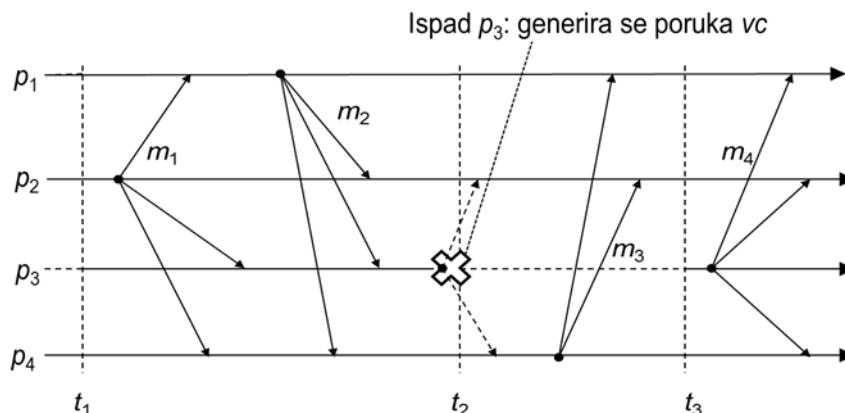
1. scenarij: Jedan od procesa iz skupine šalje poruku  $m$ , a tijekom njene isporuke novi se proces uključuje u skupinu.

- Proces  $p$  šalje poruku  $m$ , u tome trenutku postoji skupina procesa  $G$ .
- Tijekom isporuke poruke  $m$  novi proces se uključuje u  $G$  i generira se poruka  $vc$  (*view change*) koja se šalje svim procesima iz  $G$ . Posljedica:  $p$  i  $vc$  su istovremeno u tranzitu.
- 2 su moguća rješenja: 1) poruka  $m$  se isporučuje svim članovima iz  $G$  prije isporuke  $vc$  ili poruka  $m$  nije isporučena niti jednom procesu iz  $G$ .

2. scenarij: Jedan od procesa iz skupine šalje poruku  $m$ , a tijekom njene isporuke se dogodi ispad procesa pošiljatelja poruke.

- Ako  $p$  pošalje  $m$  i ispadne prije isporuke  $m$  svim procesima iz  $G$ , procesi ignoriraju  $m$  (ne treba osigurati isporuku ostalim ispravnim procesima, kao da je  $p$  ispašao prije slanja  $m$ ).

Navedena dva scenarija i ponuđena rješenja omogućuju tzv. virtualnu sinkronost skupine procesa jer su procesi „sinkronizirani“ u trenucima kada se mijenja skupina procesa  $G$ , tj. poslane poruke se isporučuju samo unutar intervala u kojima se ne mijenja članstvo u  $G$ . Stoga se poruka može isporučiti članovima iz  $G$  samo ako ne postoji poruka  $vc$  koja je istovremeno u tranzitu. Implementacija virtualne sinkronosti nije trivijalna, a jedan od prvih sustava koji implementira virtualnu sinkronost poznat je pod nazivom Isis<sup>38</sup>.



Slika 9.6. Primjer virtualne sinkronosti

Slika 9.6 prikazuje primjer skupine  $G$  od 4 procesa koji razmjenjuju poruke. U trenucima  $t_1$ ,  $t_2$  i  $t_3$  mijenja se  $G$  jer se u trenutku  $t_1$  proces  $p_3$  uključuje u  $G$ , u trenutku  $t_2$  dolazi do ispada procesa  $p_3$ , a u trenutku  $t_3$  proces  $p_3$  se ponovno uključuje u  $G$ . Na slici nije prikazan prijenos poruka  $vc$  jer je prepostavka da se ove poruke prenose trenutno te je sustav sinkroniziran u navedenim trenucima. Može se uočiti da se poruke prenose među svim procesima iz  $G$  samo unutar sinkronizirajućih intervala te da u slučaju kada dođe do ispada procesa  $p_3$  koji je generirao poruku koja je u tranzitu, procesi koji su primili poruku od procesa  $p_3$  navedenu poruku ignoriraju.

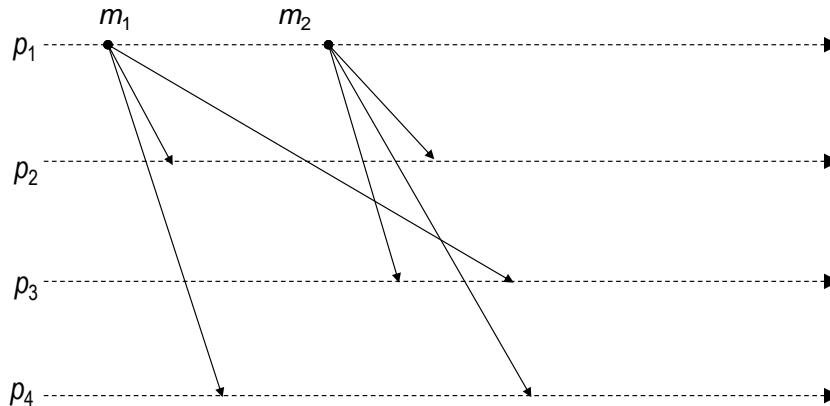
### 9.2.3 Slijed isporuke poruka

Za pouzdanu komunikaciju skupine procesa od velike je važnosti slijed kojim procesi primaju poruke i slijed primanja poruka u odnosu na njihovo slanje, jer utječe na promjene stanja tih procesa. Slijed primljenih poruka može biti:

1. neuređen (engl. *unordered multicast*),
2. FIFO (engl. *FIFO-ordered multicast*) i
3. potpuno uređen (engl. *totally-ordered multicast*).

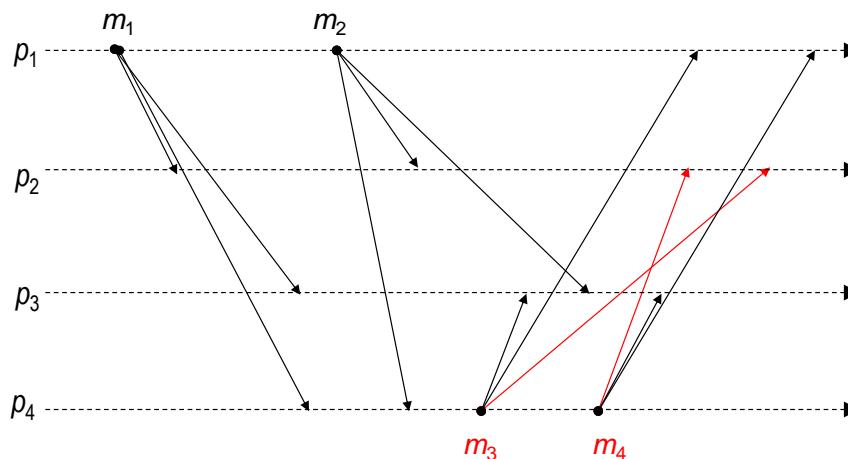
Neuređeni slijed poruka prilikom višeodredišne komunikacije označava isporuku poruka procesima u skupini  $G$  u proizvoljnem redoslijedu. Pouzdana višeodredišna neuređena komunikacija je zapravo pouzdana višeodredišna komunikacija koja je istovremeno i **virtualno sinkrona**. Slika 9.7 prikazuje slijed isporuke dviju poruka prilikom pouzdane višeodredišne komunikacije kada je slijednost isporuke poruka neuređena. Vidljivo je da procesi  $p_2$  i  $p_4$  primaju poruke na isti način i u redoslijedu kojim su poruke poslane, ali proces  $p_3$  poruke prima u suprotnom redoslijedu.

<sup>38</sup> <http://www.cs.cornell.edu/ken/history.pdf>



Slika 9.7. Primjer neuređenog slijeda poruka prilikom pouzdane višeodredišne komunikacije

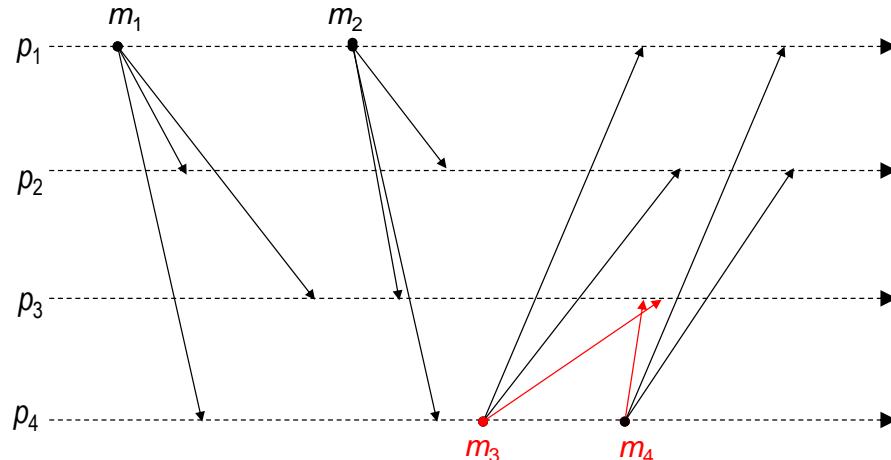
Slijed poruka FIFO prilikom višeodredišne komunikacije garantira isporuku poruka poslanih od istog procesa u redoslijedu kojim su poslane. Slika 9.8 ilustrira ovakav način komunikacije. Kako proces  $p_1$  šalje prvo poruku  $m_1$  a zatim  $m_2$ , navedene poruke moraju ostalim procesima biti isporučene u tom redoslijedu. Navedeno vrijedi za sve procese ( $p_2$ ,  $p_3$  i  $p_4$ ), ali ne vrijedi za poruke  $m_3$  i  $m_4$  jer proces  $p_2$  prima navedene poruke u različitom slijedu. Vidljivo je da proces  $p_3$  prima poruke u sljedećem slijedu:  $m_1, m_3, m_2, m_4$  što je ispravno za slijed FIFO jer su poruke od jednog izvora ispravno poredane.



Slika 9.8. Primjer slijeda poruka FIFO prilikom pouzdane višeodredišne komunikacije

Potpuno uređen slijed poruka prilikom višeodredišne komunikacije garantira isporuku poruka u istom redoslijedu svim procesima u grupi uz redoslijed isporuke poruka FIFO od istog procesa. Na ovaj način svi procesi primaju poruke u jednakom redoslijedu te imaju isti pogled na sustav uz isti način mijenjanja lokalnih stanja. Ovaj se način komunikacije naziva u literaturi *atomic multicast* i riječ je o pouzdanom virtualno sinkronom multicast-u s potpuno uređenom slijednošću isporuke poruka.

Ako se dogodi ispad procesa u trenutku isporuke poruke, poruka se isporučuje ili svim ispravnim procesima ili niti jednome od njih jer se poštuje pravilo virtualne sinkronosti. Ispravni procesi se moraju prethodno dogovoriti o članovima grupe (isporka vc u G čime je neispravan proces "izbačen" iz grupe). Kada se proces oporavi treba ga vratiti u stanje prije ispada, a zatim sinkronizirati s drugim procesima i dovesti u radno stanje u kome su ostali procesi. Samo nakon toga je moguće priključiti taj proces ponovo u grupu procesa (treba ga dovesti u isto stanje kao ostale procese).

Slika 9.9. Primjer komunikacije *atomic multicast*

Slika 9.9 ilustrira komunikaciju koja slijedi načelo *atomic multicast*-a. Poruke se isporučuju u istom redoslijedu svim procesima u grupi (treba uzeti u obzir i FIFO redoslijed poruka koje dolaze od istog procesa, što znači da npr. poruka  $m_2$  ne može biti isporučena prije  $m_1$ ). Valja uočiti da je načelo narušeno prilikom isporuke poruka  $m_3$  i  $m_4$  procesu  $p_3$ .

### 9.3 Raspodijeljeno izvršavanje operacija

Raspodijeljeno izvršavanje operacija (engl. *distributed commit*) veže se uz pojam transakcije koja se prvi puta pojavljuje u području bankarskih operacija nad bazama podataka. Cilj je izvršiti operaciju nad svim procesima iz skupine i to na način da se prvo svi procesi slože i obvezu izvesti operaciju (u suprotnom se operacija ne izvodi), a potom svi procesi operaciju i izvode ili operaciju ne izvodi niti jedan od njih. Ako je riječ o operaciji isporuke poruke, raspodijeljeno izvršavanje operacija svodi se na *atomic multicast*. Primjeri transakcija su: operacije nad bazom podataka i slijed klijentskih zahtjeva za izvođenje operacija nad datotekama u raspodijeljenom datotečnom sustavu. Za transakcije su značajna tzv. 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 i
- *Durability*: nakon završetka transakcije sve su promjene trajne.

Kada bi se operacije izvršavale u jednoj fazi (engl. *one-phase commit*), morao bi postojati koordinator u skupini procesa koji šalje zahtjev ostalim procesima za (lokalno) izvršavanje operacije. No ovo rješenje ima očiti nedostatak, a taj je da procesi ne mogu obavijestiti koordinatora u slučaju nemogućnosti izvršavanja operacije i odustati od transakcije. Stoga je nužno izvršavanje operacija kao transakcija u skupini procesa u dvije faze: u prvoj se fazi procesi obvezuju na izvođenje operacije, a u drugoj se operacija i izvršava.

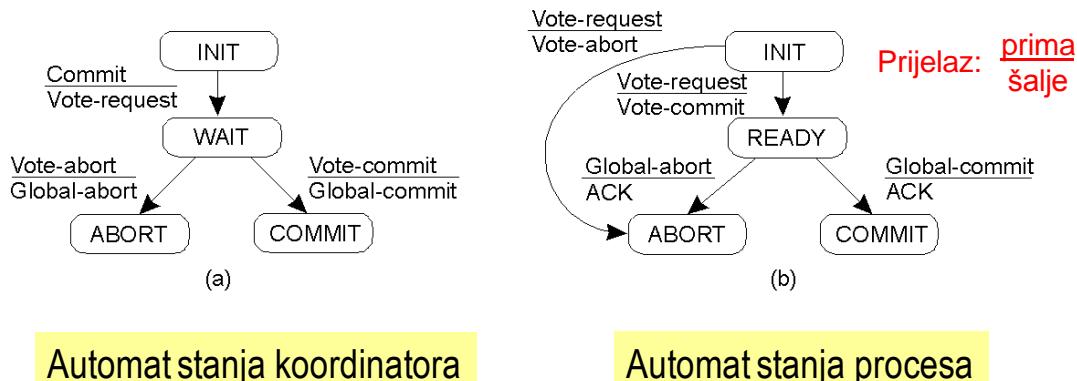
#### 9.3.1 Protokol dvofaznog izvršavanja

Protokol dvofaznog izvršavanja operacije u skupini procesa (engl. *two-phase commit protocol*, 2PC) opisan je automatsima stanja za koordinatoru i procesu na slici (Slika 9.10). Koordinator može biti u četiri stanja: u stanju INIT kada čeka signal iz okoline kako bi započeo transakciju, u stanju WAIT kada čeka odgovore od procesa jesu li spremni izvršiti operaciju, u stanju ABORT kada donosi odluku o neizvršavanju operacije ili u stanju COMMIT kada donosi odluku o izvršavanju transakcije. Svaki proces također može biti u četiri moguća stanja: u stanju INIT kada od koordinatora očekuje poruku

VOTE\_REQUEST, u stanju READY kada je spremjan za izvršavanje operacije, u stanju ABORT kada odustaje od izvršavanja operacije i u stanju COMMIT kada izvršava operaciju.

Protokol 2PC se izvodi na sljedeći način:

1. Koordinator šalje poruku VOTE\_REQUEST svim procesima iz skupine.
2. Kada proces primi VOTE\_REQUEST, odgovara porukom VOTE\_COMMIT ako je spremjan izvršiti traženu operaciju ili u suprotnom šalje VOTE\_ABORT.
3. Koordinator skuplja sve odgovore. Ako su svi procesi vratili VOTE\_COMMIT, koordinator šalje svima GLOBAL\_COMMIT. Ako je barem jedan proces odgovorio s VOTE\_ABORT, koordinator šalje poruku GLOBAL\_ABORT.
4. Svaki proces koji je odgovorio s VOTE\_COMMIT čeka konačnu odluku koordinatora. Ako primi GLOBAL\_COMMIT, obavlja operaciju lokalno. Ako primi GLOBAL\_ABORT, odustaje od izvršavanja operacije.



Slika 9.10. Automati stanja za protokol 2PC

Postavlja se pitanje u kojim stanjima može doći do blokiranja procesa i koordinatora jer su to rizična stanja za izvođenje transakcije.

- Proces je blokiran u stanju INIT kada čeka VOTE\_REQUEST: Ako ne primi poruku nakon određenog vremenskog intervala, proces može lokalno odustati od izvršavanja operacije i prijeći u stanje ABORT.
- Koordinator je blokiran u stanju WAIT kada čeka odgovore svih procesa: Ako nakon nekog perioda ne primi odgovore od svih procesa, koordinator može zaključiti da treba odustati od izvršavanja operacije jer je neki proces potencijalno u ispadu i poslati svim procesima GLOBAL\_ABORT nakon čega i sam ulazi u stanje ABORT.
- Proces je blokiran u stanju READY čekajući konačnu odluku koordinatora: U ovom slučaju se dogodio ispad koordinatora i proces treba saznati koju je poruku koordinator prethodno posao ostalim procesima jer je to njegova konačna odluka i pitati drugi proces što se događa.

Prva dva blokirajuća stanja nisu problematična jer se rješavaju jednostavnim uvođenjem brojača vremena, no 2PC je blokirajući protokol zbog trećeg stanja READY. U slučaju ispada koordinatora nakon slanja VOTE\_REQUEST, procesi ne mogu zaključiti koja je sljedeća operacija koju trebaju provesti jer koordinator ne donosi odluku o tome koje je sljedeće ispravno stanje. Procesi mogu propitivati ostale procese jesu li primili odluku koordinatora i na temelju toga odlučiti koje je ispravno stanje. Ovo se događa kada koordinator ispadne tijekom slanja poruke GLOBAL\_COMMIT ili GLOBAL\_ABORT. U tablici (Tablica 9.2) su navedene akcije procesa  $p_i$  kada se nalazi blokiran u stanju READY i kada kontaktira drugi proces  $p_j$  iz skupine procesa kako bi donio odluku o svom sljedećem ispravnom stanju.

Tablica 9.2. Utjecaj stanja procesa  $p_j$  na odluku o sljedećem stanju procesa  $p_i$ 

Stanje procesa $p_j$	Akcije procesa $p_i$
COMMIT	Obavi prijelaz u COMMIT (jer je proces $p_j$ primio GLOBAL_COMMIT)
ABORT	Obavi prijelaz u ABORT (jer je proces $p_j$ primio GLOBAL_ABORT)
INIT	Obavi prijelaz u ABORT (jer proces $p_j$ nije primio VOTE_REQUEST)
READY	Kontaktiraj drugi proces (jer proces $p_j$ nije dobio odgovor od koordinatora)

U nastavku je dan algoritam za koordinatora protokola 2PC (Tanenbaum & Van Steen, 2007):

```

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;
}

```

U nastavku je dan algoritam za proces protokola 2PC (Tanenbaum & Van Steen, 2007):

```

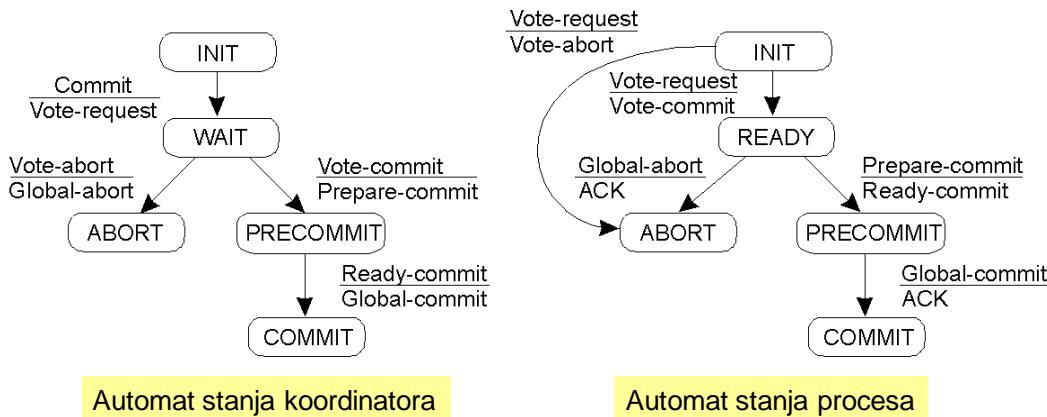
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast 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;
}

```

Moguće je da proces bude blokiran u slučaju ispada koordinatora do njegovog oporavka. To se može dogoditi u situaciji kada svi procesi primaju od koordinatora poruku VOTE\_REQUEST, a koordinator u međuvremenu ispadne. U tom slučaju procesi ne mogu bez pomoći koordinatora zaključiti koja je sljedeća ispravna operacija. Stoga je protokol 2PC blokirajući protokol te je definiran protokol novi trofazni protokol.

### 9.3.2 Protokol trofaznog izvršavanja

Protokol trofaznog izvršavanja operacije u skupini procesa (*three-phase commit protocol*, 3PC) rješava problema blokiranja procesa za protokol 2PC u slučaju ispada koordinatora tako da dodaje još jednu fazu i dodatnu provjeru izvršavanja operacije. Treća faza komplikira protokol i produžava trajanje procesa transakcije te se stoga relativno slabo koristi u praksi, tim više jer se situacija koja dovodi do stanja blokiranja protokola 2PC događa iznimno rijetko.



Slika 9.11. Automati stanja za protokol 3PC

Slika 9.11 definira automate stanja za koordinatora i procese za protokol 3PC. Vidljivo je novo stanje koordinatora i procesa PRECOMMIT te dvije nove poruke: PREPARE\_COMMIT i READY\_COMMIT.

Protokol je sličan protokolu 2PC s tom razlikom što koordinator nakon odluke za izvođenje operacije šalje poruku PREPARE\_COMMIT i ulazi u stanje PRECOMMIT na koju procesi odgovaraju s READY\_COMMIT. Nakon što primi poruku READY\_COMMIT od svih procesa, koordinator šalje GLOBAL\_COMMIT. Proses također nakon primitka poruke PREPARE\_COMMIT ulazi u stanje PRECOMMIT u kome očekuje poruku GLOBAL\_COMMIT od koordinatora.

Postavlja se ponovno pitanje koja su stanja blokirajuća.

1. Koordinator može biti blokiran u stanju PRECOMMIT zbog ispada jednog procesa, ali može ostalim procesima poslati GOBAL\_COMMIT.
2. Proces može biti blokiran u stanjima READY i PRECOMMIT. Nakon isteka vremenske kontrole zaključuje da je došlo do ispada koordinatora i kontaktira ostale procese, no ovaj puta može na temelju stanja ostalih procesa donijeti konačnu odluku o ispravnom sljedećem stanju na temelju ovih pravila ( $p_i$  i  $p_j$  su procesi iz skupine procesa  $G$ ):
  - ako je  $p_j$  u stanju COMMIT i  $p_i$  prelazi u COMMIT,
  - ako je  $p_j$  u stanju ABORT i  $p_i$  prelazi u ABORT,
  - ako su svi procesi iz  $G$  (ili većina) u stanju PRECOMMIT, mogu svi prijeći u COMMIT,
  - ako je  $p_j$  u INIT,  $p_i$  prelazi u ABORT i
  - ako su svi procesi u stanju READY, mogu svi prijeći u ABORT.

## 9.4 Oporavak nakon ispada

Nakon ispada procesa nužan je njegov oporavak i povratak u ispravno stanje. Koriste se dvije osnovne tehnike za oporavak raspodijeljenog sustava nakon ispada:

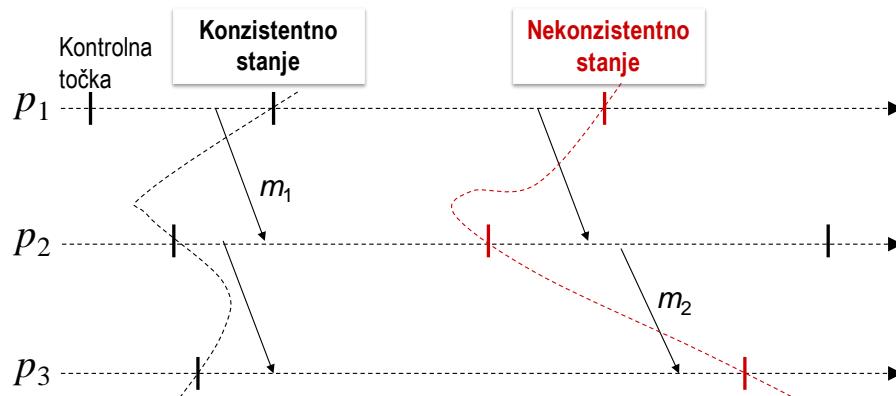
- 1) oporavak unazad: Tehnika oporavaka unazad ima za cilj vratiti sustav u ispravno stanje u prošlosti i temelji se na bilježenju stanja sustava u proizvoljnim kontrolnim točkama (engl. *checkpoint*). Dakle, potrebno je s vremena na vrijeme pohraniti stanje sustava u dnevnički zapis (engl. *log file*).
- 2) oporavak korištenjem dnevničkog zapisa. Ova tehnika osim stanja sustava u kontrolnim točkama bilježi i primljene poruke. Oporavak se temelji na sljedećoj proceduri: proces u ispadu vraća se u prethodno ispravno stanje nakon čega izvodi akcije iz dnevničkog zapisa.

Za oporavak unazad značajno je bilježenje kontrolnih točaka. Procesi mogu kontrolne točke bilježiti koordinirano ili neovisno.

Koordinirano bilježenje kontrolnih točaka koristi koordinatora i jednostavnije je za implementaciju, jer koordinator šalje naredbu procesima da pohrane stanje sustava gotovo istovremeno (prije toga moraju primiti poruke u tranzitu i privremeno zaustaviti pripremljena slanja poruka).

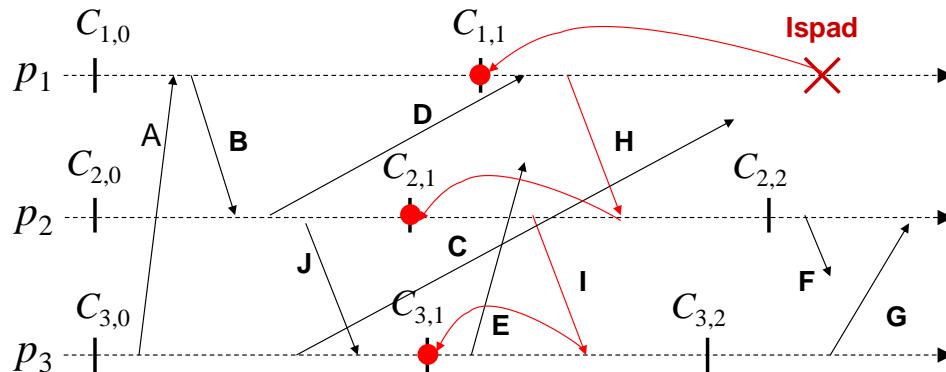
Neovisno bilježenje kontrolnih točaka koristi se kada procese nije moguće sinkronizirati. Ono zahtjeva bilježenje ovisnosti među procesima koji razmjenjuju poruku, tj. šalju i primaju istu poruku, te su stoga povezani uzročnom relacijom (definiranom u poglavlju 6.1), a radi povratka u konzistentno stanje sustava. Ako su dva procesa međusobno ovisni onda vrijedi sljedeće: Ako se proces  $p_1$  vraća u stanje prije događaja  $\text{send}(m)$ , onda se proces  $p_2$  koji je primatelj poruke  $m$  mora nužno vratiti u stanje prije događaja  $\text{receive}(m)$ . U protivnom je sustav u nekonzistentnom stanju.

Slika 9.12 prikazuje primjere konzistentnog i nekonzistentnog stanja. Procesi na slici bilježe kontrolne točke neovisno. Kontrolne točke označene crno čine konzistentno stanje sustava jer je poruka  $m_1$  poslana, no još uvijek nije primljena (što je ispravno). Kontrolne točke označene crveno daju nekonzistentno stanje sustava jer vraćaju proces  $p_2$  u stanje prije slanja poruke  $m_2$ , a proces  $p_3$  u stanje nakon primitka poruke  $m_2$ , a prema uzročnoj relaciji takvo stanje je neispravno.



Slika 9.12. Primjeri konzistentnog i nekonzistentnog stanja

Slika 9.13 prikazuje primjer oporavka povratkom unazad za tri procesa (Kshemkalyani & Singhal, 2008). Ispad se dogodio na procesu  $p_1$ , a prvo ispravno konzistentno stanje u koje se sustav može vratiti je ono u koje vraća procese u kontrolne točke  $C_{1,1}$ ,  $C_{2,1}$ ,  $C_{3,1}$ . Proses  $p_2$  nije moguće vratiti u kontrolnu točku  $C_{2,2}$  zbog poruke **H** koju bi taj proces primio, a proces  $p_1$  je do kontrolne točke  $C_{1,1}$  još nije poslao. Isto vrijedi i za kontrolnu točku procesa  $p_3$   $C_{3,2}$  zbog poruke **I**.

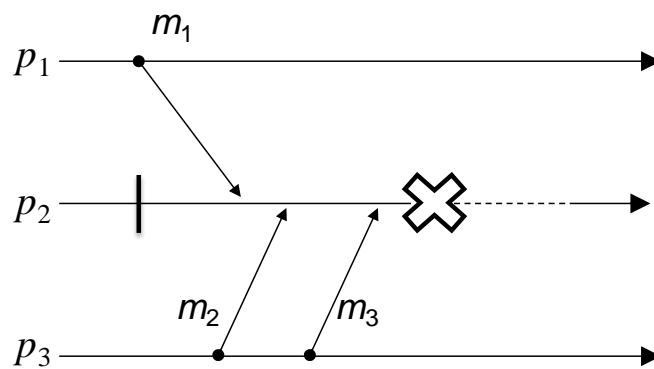


Slika 9.13. Primjer oporavka povratkom unazad

Za poruke iz ovog primjera možemo zaključiti sljedeće:

- **A, B, J** su normalne poruke koje su poslane i primljene prije kontrolnih točki  $C_{1,1}$ ,  $C_{2,1}$ ,  $C_{3,1}$ .
- **C** je zakašnjela poruka koja može stići do procesa  $p_1$  prije njegovog oporavka, tijekom oporavka ili nakon oporavka. Moguće je da se poruka **C** izgubi ako stigne prije ili tijekom oporavka, no gubitak poruke ne predstavlja problem za sustav jer na nepouzdanom kanalu i tako može doći do gubitka poruke.
- **D** je izgubljena poruka jer ju je proces  $p_2$  poslao, no  $p_1$  je vraćen u stanje kada poruka **D** još uvijek nije primljena. Proces  $p_2$  neće ponovno slati poruku **D**, no gubitak poruke ne predstavlja problem za sustav jer na nepouzdanom kanalu i tako može doći do gubitka poruke.
- **G, H, I** su poništene poruke, ponovit će se odašiljanje nakon kontrolnih točki  $C_{1,1}$ ,  $C_{2,1}$ ,  $C_{3,1}$ .
- **E** i **F** su zakašnjele suvišne poruke koje su u tranzitu u trenutku ispada. One predstavljaju najveći problem za sustav jer ako/kad stignu na odredište treba ih odbaciti zbog toga što su svi procesi vraćeni u stanja prije njihovog slanja te će se ponoviti njihovo odašiljanje.

Oporavak korištenjem dnevničkog zapisa koristi kombinirano kontrolne točke i dnevnički zapis u koji se unose primljene poruke na procesu. Primitak poruke smatra se nedeterminističkim događajem te se primitak poruke s pripadnim informacijama (pošiljatelj, primatelj, redni broj poruke, sadržaj) treba bilježiti u dnevnički zapis. U nastavku je na primjeru objašnjen oporavak korištenjem dnevničkog zapisa.



Slika 9.14. Primjer bilježenja nedeterminističkih događaja u dnevnički zapis

U primjeru na slici (Slika 9.14) su tri procesa od kojih proces  $p_2$  bilježi primitak poruka  $m_1$ ,  $m_2$  i  $m_3$  u dnevnički zapis. Nakon ispada procesa  $p_2$ , taj se proces vraća u posljednju zabilježenu kontrolnu točku nakon čega proces na temelju dnevničkog zapisa simulira primitak poruka u vremenu čime se ponovno vraća u ispravno radno stanje.

### 9.5 Pitanja za učenje i ponavljanje

- 9.1. Objasnite razliku između ispada sustava i neispravnosti u sustavu.
- 9.2. Prepostavite da skupina procesa treba postići sporazum. U slučaju da su dva procesa skupine u stanju bizantskog ispada, koji je minimalni ukupni broj procesa u skupini za postizanje sporazuma?
- 9.3. Objasnite razliku protokola three-phase commit u odnosu na two-phase commit.
- 9.4. U skupini od 4 procesa ( $p_1, p_2, p_3$  i  $p_4$ ) proces  $p_1$  je neispravan (prepostavite bizantski ispad). Skupina procesa želi postići sporazum o identifikatorima ostalih procesa grupe. U koracima 1 i 3 procesi međusobno razmjenjuju podatke, a u koracima 2 i 4 prikupljaju i analiziraju primljene podatke. Nacrtajte na slici podatke koje procesi razmjenjuju u koracima 1 i 3, a za korake 2 i 4 navedite podatke koje pojedini proces ima na raspolaganju radi donošenja odluke o sporazumu.

## 10 VREDNOVANJE PERFORMANCI RASPODIJELJENIH SUSTAVA

Svaki se sustav, pa tako i raspodijeljeni može promatrati s dva motrišta, funkciskog i nefunkcijskog i pridijeliti mu isto takva, funkcija i nefunkcija obilježja. Dok funkcija obilježja govore o tome ŠTO sustav radi, kakva mu je namjena i koje funkcije obavlja, nefunkcija obilježja opisuju KAKO sustav, primjerice koliko korisnika može poslužiti uz zadovoljavajuće vrijeme odziva, kakva mu je pouzdanost ili koliki su troškovi njegovog rada. To kako sustav radi, obuhvaćeno je pojmom kvalitete usluge (engl. *Quality of Service*, QoS), skupnim nazivom za nefunkcija obilježja sustava. Dogovoren ili jamčena kvaliteta usluge definira se ugovorom o kvaliteti usluge (engl. *Service Level Agreement*, SLA) kojim korisnik i davatelj usluge određuju kakvu uslugu očekuju, odnosno obvezuju se pružiti. Stoga je vrednovanje nefunkcija obilježja važan i neizostavan dio postupaka oblikovanja raspodijeljenih sustava.

Tri kategorije nefunkcija obilježja određuju kvalitetu usluge i to:

- performance sustava,
- raspoloživost i pouzdanost sustava te
- ukupni trošak vlasništva sustava.

Performance se definiraju na različite načine i uključuju različite izračunljive i mjerljive parametre koji opisuju sposobnost sustava da ostvari funkcije kojima je namijenjen. Primjerice, mrežne performance definirane su kao sposobnost mreže ili dijela mreže da ostvari funkcije potrebne za komunikaciju između korisnika ili korisnika i poslužiteljskih sustava. Performance računalnih sustava mogu se definirati kao sposobnost obrade i pohrane podataka te posluživanja korisničkih zahtjeva. Općenito, performance govore koliko posla sustav može obaviti (kakav je njegov kapacitet i kakva je propusnost) i kako brzo (kakvo je vrijeme odziva).

Pouzdanost sustava (engl. *system reliability*) je definirana kao vjerojatnost da sustav radi ispravno u vremenskom periodu  $t$  pod definiranim uvjetima okružja. Raspoloživost sustava (engl. *system availability*) definirana kao vjerojatnost da sustav radi ispravno u trenutku  $t$ , odnosno u trenutku kad ga korisnik treba. U praksi se češće rabi neraspoloživost sustava (engl. *system unavailability*), komplementarna od raspoloživosti koju opisuje vjerojatnost da sustav NE radi ispravno u trenutku  $t$ . U nastavku će se pouzdanost i raspoloživost obraditi kao posebna nefunkcija obilježja, iako se mogu tretirati i kao parametri performanci sustava.

Ukupni trošak vlasništva sustava (engl. *Total Cost of Ownership*, TCO) obuhvaća troškove ulaganja u nabavu i uspostavljanje sustava te troškove njegovog rada, pri čemu vlasnik sustava može na različite načine izgraditi sustav, s većim ili manjim udjelom vlastite mrežne i računalne opreme, dijeleći je s drugima i zakupljujući je od drugih.

### 10.1 Modeli vrednovanja nefunkcija obilježja

Životni ciklus svakog sustava (engl. *system lifecycle*) obuhvaća faze od početne razrade i definicije zahtjeva kojima treba udovoljiti, pa sve do povlačenja sustava iz uporabe i prestanka njegovog rada. Nakon što se definiraju zahtjevi, slijedi analiza funkcija i nefunkcija obilježja na temelju koje se odabire i razvija rješenje sustava te ispituje prije puštanja u rad. Tijekom rada sustava provode se ispitivanja i mjerjenja kako bi se ustanovilo ispunjavanje zahtijevanih funkcija i nefunkcija obilježja. Iskustva u radu sustava potvrđena mjerjenjima njegovih performanci ukazat će na potrebne promjene i unaprjeđenja. Ako zahtjevi i početne pretpostavke nisu bili korektni ili cjeloviti, trebat će ih modificirati i provesti određene zahvate u sustavu. Kontinuirano praćenje sustava u radu i iterativni postupak prilagodbe sustava stvarnoj situaciji prijeko su potrebni da bi se održala očekivana i ugovorenata kvaliteta usluge.

Pokažimo to na primjeru web-dućana, odnosno web-aplikacije za električku trgovinu: poduzeće želi organizirati prodaju putem Interneta i to predstavlja osnovni funkcionalni zahtjev na sustav koji treba izgraditi.

Kako se mogu postaviti nefunkcionalni zahtjevi? Iskustva o internetskom trgovovanju ima mnogo, istražuje se ponašanje kupaca i njihove sklonosti, razlozi za uspješnu i neuspješnu prodaju. Pretpostavimo sljedeće:

- 60 % kupaca napušta web-stranicu ako je odziv aplikacije 4-6 s;
- 90 % kupaca napušta web-stranicu aplikacije ako je odziv aplikacije veći od 6 s;
- 5 % od svih kupaca koji su posjetili web-stranicu aplikacije kupuje proizvode za prosječnu cijenu 1200 kn.

Očvidno je da će uz nezadovoljavajuću kvalitetu usluge izraženu preugim vremenom odziva – odziva web-aplikacije, većina korisnika odustati od pokušaja kupnje i uopće neće dočekati „otvaranje“ web-stranice za električku trgovinu<sup>39</sup>. O tome se mora voditi računa pri oblikovanju i projektiranju sustava, a analizom nefunkcionalnih zahtjeva, u ovom primjeru vremena odziva, obuhvatiti ovakva pitanja:

- Kakav se odziv sustava može postići za početno opterećenje izraženo očekivanim brojem posjeta web-aplikaciji na sat (početni promet)?
- Kako će se sustav ponašati ako se broj posjeta web-aplikaciji poveća za 30%, 60% ili 90% (povećanje prometa)?
- U kojim će uvjetima odziv aplikacije preći u nezadovoljavajuće područje?
- Koliki gubitak prihoda uzrokuje gubitak kupaca zbog slabog odziva aplikacije?
- Kolika su ulaganja potrebna da se uz povećanje prometa zadrži sav posao?
- Kada će se, uz trenutačni trend, potreba za kapacitetom sustava udvostručiti?

Neka je za primjer, za početnih očekivanih 900 posjeta/sat, ustanovljeno vrijeme odziva 2,96 s (Tablica 10.1). U takvim uvjetima, kvaliteta usluge je prihvatljiva, tj. vrijeme odziva manje od 4 s, pa neće biti niti izgubljenih kupaca, ni izgubljenog prihoda.

Tablica 10.1 Primjer analize web-aplikacije

Povećanje broja korisnika	Početno	+30 %	+60 %	+90 %
Posjet/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

<sup>39</sup> Istraživanja koja je proveo Google pokazuju da svako čekanje za 0,5 s veće od uobičajenog smanjuje informacijski promet za 20 %, a istraživanja Amazona da svakih 0,1 s dodatnog kašnjenja smanjuje vjerojatnost kupnje za 1 %.

Problemi s kvalitetom usluge nastaju uz povećanje broja korisnika za 60%, kad zbog porasta vremena odziva i s time povezanih izgubljenih kupaca dolazi do većeg gubitka prihoda. Još je gora situacija uz porast broja kupaca za 90 %, kad gubitak prihoda postaje drastičan. Na temelju prikazanih rezultata može se zaključiti da pogrešno projektirana aplikacija može imati katastrofalne posljedice na poslovanje. Treba napomenuti da je u ovom primjeru zbog jednostavnosti primijenjena linearna ekstrapolacija koja najčešće ne daje točne rezultate – stvarna situacija bi vjerojatno bila još gora!

Već ovakvo početno vrednovanje performansi pokazuje da je planiranje rasta kapaciteta prijeko potrebno. Ako je web-poslužitelj planiran kao jedan, monolitni, sustav (centralizirano rješenje) trebat će ga zamijeniti s poslužiteljem većeg kapaciteta kad dođe do većeg porasta broja korisnika. Međutim ako je riječ o raspodijeljenom sustavu, porast kapaciteta može se postići uvođenjem dodatnih poslužitelja koji će preuzeti dio opterećenja i tako održati vrijeme odziva prihvatljivim.

Postupci vrednovanja nefunkcijskih obilježja oslanjaju se na tri pristupa:

- Procjene zasnovane na intuiciji i iskustvu, koje su veoma teške, jer raspodijeljeni sustavi pokazuju izrazito nelinearno ponašanje;
- Modeliranje, koje podrazumijeva razvoj matematičkog modela koji opisuje ovisnost performansi o parametrima sustava;
- Simulacija, kao najtočnija metoda, ali često preskupa za primjenu.

U nastavku je težište na modelskom pristupu.

Razumijevanje funkcioniranja sustava je polazna je pretpostavka pri razvoju modela raspodijeljenog sustava prikladnog za vrednovanje performansi. Kad je to zadovoljeno, sustavni postupak nastavlja se sljedećim koracima:

- modeliranje tereta – korisničkih zahtjeva koje sustav treba poslužiti,
- mjerenje radi utvrđivanja parametara tereta,
- razvoj modela performansi,
- verifikacija i validacija modela performansi,
- analiza mogućih scenarija promjena u budućnosti,
- predviđanje promjena tereta u budućnosti,
- predviđanje performansi sustava nakon puštanja u rad te u budućnosti.

U prethodnom primjeru riječ je bila o webu čije funkcioniranje dobro razumijemo, pa predimo na modeliranje tereta. Korisnički upiti su kratki i odnose se na pribavljanje informacija proizvodima koji se nude za odabir i kupnju. Informacija o proizvodu koju treba dostaviti korisniku tijekom kupovine može biti posebni kratki dokument ili sadržana u katalogu – većem dokumentu. Jedinica tereta za web-aplikaciju i poslužitelja na kojem se izvodi je informacijski sadržaj – dokument, opisan s dva parametra: veličina i učestalost zahtjeva, odnosno broj pristupa u određenom vremenu.

Mjerenje omogućuje utvrđivanje parametara tereta, kao što je to prikazano primjerom (Tablica 10.2)<sup>40</sup>. Mjerenjem je ustanovljeno da se s web-poslužitelja dohvata sedam vrsta dokumenata različite veličine i s različitim brojem pristupa. To je polazište za utvrđivanje značajki realnog tereta razmatranog sustava.

Srednja vrijednost parametara može biti reprezentativna ako se pojedinačne vrijednosti značajno ne razlikuju. Međutim u ovom primjeru razlike su velike: veličina dokumenta 7-150 kB, a broj pristupa

<sup>40</sup> Primjer preuzet iz: D.A. Menascé, V.A.F. Almeida, „Capacity Planning for Web Services“, 6. Understanding and Characterizing the Workload, Prentice Hall, 2002.

75-293 (srednja veličina 34 kB, sa srednjim brojem pristupa 185,7), tako da je prikladnije ustanoviti klase tereta, odnosno grupe sličnih dokumenta.

Postoje različite metode grupiranja, a najčešće se koristi grupiranje na temelju Euklidske udaljenosti,  $d$ , između elemenata  $x_i$  i  $x_j$  opisanih s  $K$  parametara:

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

U ovom primjeru računa se Euklidska udaljenost dokumenata opisanih s  $K = 2$  parametra ( $k = 1$  – veličina,  $k = 2$  – broj pristupa) i provodi grupiranje u tri grupe. Kako su razlike veličine parametara zнатне, provodi se promjena mjerila – u ovom primjeru koristi se  $\log_{10}$  (Tablica 10.3).

Tablica 10.2. Rezultati mjerjenja tereta

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

Tablica 10.3. Rezultati mjerjenja tereta (mjerilo  $\log_{10}$ )

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

Početne grupe G1 – G7 odgovaraju pojedinačnim dokumentima. Podrazumijevajući da je težiste grupe od jedne točke upravo ta točka, izračunavamo Euklidske udaljenosti između težišta (Tablica 10.4). Primjerice, Euklidska udaljenost između G3 i G6 iznosi:

$$d_{36} = \sqrt{(x_{31} - x_{61})^2 + (x_{32} - x_{62})^2} = \sqrt{(0,7 - 0,6)^2 + (2,47 - 2,38)^2} = \sqrt{0,0181} = 0,13$$

Tablica 10.4. Grupiranje – 1. korak

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	<b>0,13</b>	1,03
G4				0	0,64	0,85	0,26
G5					0	0,25	0,88
G6						0	1,07
G7							0

Kako je udaljenost između G3 i G6 najmanja, svrstavamo ih u novu grupu G36 i računamo joj težište kao srednju veličinu dokumenta i srednji broj pristupa, tj. 0,65 i 2,43. U drugom se koraku računaju Euklidske udaljenosti između težišta novih grupa (Tablica 10.5).

Tablica 10.5. Grupiranje – 2. korak

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	<b>0,19</b>	1,05
G4				0	0,64	0,26
G5					0	0,88
G7						0

Sad je najmanja je udaljenost između G36 i G5, pa računamo težište nove grupe G365 (veličina dokumenta 0,75 i broj pristupa 2,42). Postupak se ponavlja dok se ne dobiju željene tri grupe (Tablica 10.6), a završava vraćanjem u izvorno mjerilo (Tablica 10.7). Ovakvo grupiranje na tri tipa dokumenta (mali, srednji, veliki) daje realističnije postavke za modeliranje sustava od „prosječnog“ tereta predočenog dokumentom srednje veličine 34 K okteta, sa srednjim brojem pristupa 185,7.

Tablica 10.6. Grupiranje – završni korak

Grupa	G1365	G2	G47
G1365	0	1,60	0,72
G2		0	0,89
G47			0

Tablica 10.7. Povrat u izvorno mjerilo

Grupa	Tip dokumenta	Veličina (kB)	Broj pristupa
G1356	mali	8,19	271,51
G47	srednji	20,58	96,05
G2	veliki	150,0	28,0

U ovom se primjeru model tereta izveo promatranjem i mjeranjem web-aplikacije u radu, što se naziva prirodnim modelom tereta. Prirodni model tereta moguće je izvesti tijekom rada sustava, pri čemu pozornost treba obratiti uvjetima u kojima se mjerjenje provodi, tj. je li riječ o tipičnom (normalnom) opterećenju, ili povećanom, zbog posebnih okolnosti i sl.

Opterećenje sustava se može predočiti i modelom umjetnog tereta koji je rezultat istraživanja ponašanja sustava, odnosno specifičnih umjetnih (zamišljenih) aplikacija koje predočuju stvarne aplikacije.

Primjerice, manjim web-dućanima kojima pripada rješavani primjer, mogao bi odgovarati ovakav umjetni model tereta:

- 70 % pristupa malim dokumentima veličine 10 kB,
- 25 % pristupa srednjim dokumentima veličine 30 kB i
- 5 % pristupa velikim dokumentima veličine 100 kB.

Za konkretnu web-aplikaciju ovakav umjetni teret se razlikuje od prirodnog, međutim dobro definirani umjetni teret može poslužiti kao ogledna vrijednost (engl. *benchmark*) za evaluaciju pojedinog rješenja i njegovu usporedbu s drugim rješenjima.

Razvijeni su i dostupni mnogi ogledni umjetni modeli tereta za različite primjene. Nепrofitna organizacija *Standard Performance Evaluation Corporation (SPEC)*<sup>41</sup> razvija i održava skup oglednih modela tereta, među kojima i one za web-poslužitelje za različite domene primjene pa tako i elektroničko trgovanje. Nепrofitna organizacija *Transaction Processing Performance Council (TPC-C)*<sup>42</sup> usmjerena je na razvoj i održavanje oglednih modela za transakcijsku obradu podataka. Uz njih, na području vrednovanja performanci djeluju i druge organizacije, usmjerene na opća ili specifična područja primjene.

Teret koji nedovoljno dobro predočuje opterećenje stvarnog sustava, samo je jedan od mogućih izvora pogrešaka pri razvoju modela raspodijeljenog sustava prikladnog za vrednovanje performanci. Raj Jain sustavno je pobrojio najčešće pogreške kod modeliranja koje se javljaju zbog nedovoljnog razumijevanja sustava i/ili samih postupaka vrednovanja, kao i previda ili nastojanja za sveobuhvatnim rješenjem koje postaje presloženo za analizu<sup>43</sup>. Temeljna je pogreška nejasan ili unaprijed postavljeni cilj. Cilj treba biti dobro definiran da bi se odabralo, razvio i primijenio odgovarajući postupak vrednovanja: Međutim ako je cilj, bolje reći rezultat, unaprijed postavljen, čak i nesvesno ili prikriveno („rješenje je dobro“, „rješenje je bolje od prethodnoga“, „naše rješenje bolje je od ostalih“, ...), teško da će vrednovanje biti objektivno.

Daljnje pogreške koje će ugroziti vrednovanje odnose se na neadekvatnu metriku i odabir parametara, promatranje parametra u krivom intervalu vrijednosti, ili krivo rukovanje ekstremnim vrijednostima. Primjerice, različiti sustavi poručivanja ne mogu se vrednovati prema vremenu odziva, jer neki rade u stvarnom vremenu, a neki uz privremenu pohranu i naknadno proslijđivanje poruka („neadekvatna metrika“). Ako se zanemari ili podcijeni ukupan broj korisnika („neadekvatna odabir parametara“ i „promatranje parametra u krivom intervalu vrijednosti“) ili intenzitet njihovih aktivnosti („krivo rukovanje ekstremnim vrijednostima“) dobit će se kriva slika o performancama sustava.

Čak ako je sve provedeno dobro, može se pogriješiti pri interpretaciji ili prezentaciji rezultata ili odabiru načina prezentacije rezultata. Pokažimo samo jednostavan primjer: web-aplikacija  $W_A$  daje vrijeme odziva 2 s, a web-aplikacija  $W_B$  4 s. Što ćemo uzeti kao referentnu vrijednost za njihovu usporedbu pri prezentaciji rezultata nekome tko treba donijeti odluku o ulaganju u sustav? Uzmemo li 2s, tad je  $W_A$  100% brža od  $W_B$ . Ako je referentna vrijednost 4, tad je vrijeme odziva web-aplikacije  $W_A$  50% onoga od  $W_B$ . 100 % može djelovati uvjerljivije od 50 %! A tek kad se (ne) vodi računa o statistici ...<sup>44</sup>.

Na važnost promatranja evolucije sustava i tereta i pogreške koje iz toga mogu proizaći već je ranije upozorenio.

Modeliranje raspodijeljenog sustava podrazumijeva razvoj matematičkih modela koji opisuju ovisnost performanci, pouzdanosti i raspoloživosti te ukupnog troška vlasništva o pojedinim parametrima sustava. Obraditi će se samo osnovne postavke pouzdanosti i raspoloživosti te ukupnog troška vlasništva, a veću pozornost obratiti performancama raspodijeljenih sustava.

## 10.2 Pouzdanost i raspoloživost

Pouzdanost sustava je definirana kao vjerojatnost da sustav radi ispravno u vremenskom periodu  $t$  pod definiranim uvjetima okružja. Raspoloživost sustava,  $A$ , definira se kao vjerojatnost da sustav radi ispravno u trenutku  $t$ .

<sup>41</sup> [www.spec.org](http://www.spec.org)

<sup>42</sup> [www\(tpc.org](http://www(tpc.org)

<sup>43</sup> R. Jain, „The Art of Computer Systems Performance Analysis“, 2 Common mistakes and how to avoid them, Wiley, 1991.

<sup>44</sup> D. Huff, „How to Lie with Statistics“, W. W. Norton & Company, 1954 (paperback reissue 1993).

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

- srednje vrijeme između kvarova (engl. *Mean Time Between Failure*, MTBF)
- srednje vrijeme popravka (engl. *Mean Time To Repair*, MTTR)

Raspoloživost se računa ovako:

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

U praksi se češće rabi neraspoloživost sustava,  $U$ , koja opisuje vjerojatnost da sustav NE radi ispravno u trenutku  $t$ :

$$U = 1 - R$$

Pogledajmo malo brojke. Raspoloživost od 0,9999 znači da će sustav biti izvan funkcije, neraspoloživ,  $(1 - 0,9999) \times 30 \times 24 \times 60 = 4,32$  min/mjesec. Raspoloživost od 0,99 znači da će sustav biti neraspoloživ  $(1 - 0,99) \times 30 \times 24 \times 60 = 4320$  min/mjesec ili 72 sata/mjesec!

Raspodijeljeni sustavi sastavljeni su od više dijelova – podsustava povezanih na različite načine, o čemu treba voditi računa pri izračunu raspoloživosti. Raspoloživost serijske kombinacije,  $A_s$ , dva podsustava raspoloživosti  $A_1$  i  $A_2$  jednaka je umnošku raspoloživosti pojedinih podsustava:

$$A_s = A_1 \times A_2,$$

Raspoloživost paralelne kombinacije,  $A_p$ , dva podsustava raspoloživosti  $A_1$  i  $A_2$  jednaka je:

$$A_p = A_1(1 - A_2) + A_2(1 - A_1) + A_1 \times A_2,$$

uz pretpostavku se da je jedan raspoloživ podsustav dovoljan za ispravno funkcioniranje sustava.

Primjer računanja je sljedeći: odrediti raspoloživost sustava s dva paralelna web-poslužitelja s raspoloživošću od  $A_W = 0,99$  te mrežne sklopke za raspodjelu opterećenja na web-poslužitelje s MTBF = 1 godine i MTTR od 2 sata.

Raspoloživost mrežne sklopke iznosi:

$$A_{MS} = (365 \times 24) / (365 \times 24 + 2) = 0,9998,$$

a raspoloživost dva paralelna web-poslužitelja:

$$A_{PW} = A_W(1 - A_W) + A_W(1 - A_W) + A_W \times A_W = 0,0099 + 0,0099 + 0,9801 = 0,9999.$$

Riječ je o serijskoj kombinaciji mrežne sklopke i dva paralelna web-poslužitelja, tako da je raspoloživost sustava definirana s:

$$A_s = A_{MS} \times A_{PW} = 0,9998 \times 0,9999 = 0,9997.$$

### 10.3 Ukupni trošak vlasništva

Ukupni trošak vlasništva uključuje dvije vrste troškova: trošak izvedbe sustava, odnosno kapitalni trošak (engl. *Capital Expenditure*, CAPEX) i trošak rada sustava, odnosno operativni trošak (engl. *Operating Expenditure*, OPEX).

Trošak izvedbe sustava obuhvaća računalnu i mrežnu opremu s pripadnom fizičkom infrastrukturom te programsku opremu potrebnu za ostvarivanje funkcija kojima je sustav namijenjen, kao i ljudski rad potreban za razvoj i postavljanje sustava. Trošak rada sustava uključuje troškove zaposlenika, prostora, energije, komunikacijskih usluga i druge troškove.

Osim izgradnjom vlastitog sustava, rješenje se može ostvariti najmom poslužiteljske infrastrukture od poslovnog entiteta koji se bavi udjmljivanjem (engl. *hosting*) računalnih i komunikacijskih sredstava. On raspolaže fizičkom infrastrukturom (zgrada, napajanje, klimatizacija), pristupom Internetu i drugim mrežama, računalnim i mrežnim sustavima, sustavskom programskom opremom (operacijski

sustavi, sustavi upravljanja bazama podataka, ...), a može nuditi i aplikacijska rješenja (udomljivanje weba, računovodstvo, knjigovodstvo, ...). Grubom podjelom, „predmet“ najma bio bi: fizička infrastruktura (u koju se smješta vlastita oprema), fizička infrastruktura i računalna oprema (na kojoj se izvode vlastite aplikacije), fizička infrastruktura, računalna oprema i aplikacijska programska oprema (kojom se poslužuju vlastiti korisnici). U prvom poslovnom modelu zakupac rješava samo smještaj opreme, u drugom i samu opremu, a u trećem unajmljuje cjelokupnu potporu sustava, a međusobni odnosi reguliraju se ugovorom o razini usluge.

Takve usluge nudi sve više entiteta koji se bave informacijskim i komunikacijskim poslovanjem, kao što su davatelji informacijskih usluga (npr. Google), davatelji usluga električkog trgovanja (npr. Amazon), davatelji internetskih usluga, mrežni operatori i drugi davatelji.

Pritom su moguća rješenja namijenjena samo jednom zakupcu – on je isključivi korisnik iznajmljenog sustava, ili za više korisnika unutar dijeljenog, zajedničkog sustava. U prvom slučaju korisnik može utjecati na performance i to plaća većom cijenom, jer su troškovi takvog rješenja veći, a u drugom ne, ali je cijena najma niža. Većinom se koriste zajednički sustavi, a dobar je primjer za to udomljivanje weba (engl. *web hosting*) koje pružaju davatelji internetske usluge svojim korisnicima, na svojim poslužiteljskim sustavima.

Računalni oblak (engl. *computing cloud*) sadrži zajednička sredstva (računala, programska oprema) koja se nude kao usluga putem Interneta – „negdje“ na mreži izvedeni su poslužiteljski sustavi koji pružaju računalne usluge. Korisnici se ne trebaju brinuti za fizičku infrastrukturu, računalnu i komunikacijsku opremu, sklopovsku i programsku.

Modeli pružanja usluga su različiti, mreža kao usluga (engl. *Network as a Service*, NaaS), informacija kao usluga (engl. *Information as a Service*, IaaS), platforma kao usluga (engl. *Platform as a Service*, PaaS), pri čemu se pod platformom podrazumijeva programska oprema potrebna za potporu aplikacija i usluga, ili program kao usluga (engl. *Software as a Service*, SaaS).

Izgraditi u potpunosti vlastiti sustav ili ga ostvariti s nekim oblikom najma, odluka je koja se temelji na trošku (engl. *cost-based decision making*) i drugim parametrima, kao što je primjerice sigurnost.

#### 10.4 Performance sustava

Za svaki sustav važno je kako brzo odgovara na postavljene zahtjeve, koliko zahtjeva može obraditi i koliko je iskorišten. Iz toga proizlaze osnovne mjere performansi: vrijeme odziva i propusnost, te s njima povezan kapacitet i iskoristivost sustava.

Vrijeme odziva odgovara vremenu potrebnom da sustav odgovori na zahtjev za posluživanjem (npr. vrijeme između pritiska na miša i pojave web-stranice na zaslonu, vrijeme između postavljanja upita i dobivanja odgovora, ...).

Propusnost sustava izražava se brojem zahtjeva posluženih u jedinici vremena. Najveći broj zahtjeva koji sustav može poslužiti u jedinici vremena jednak je njegovom kapacitetu – kapacitet odgovara maksimalnoj propusnosti sustava.

Iskoristivost sustava izražava se kao dio ili postotak kapaciteta iskorišten za posluživanje zahtjeva.

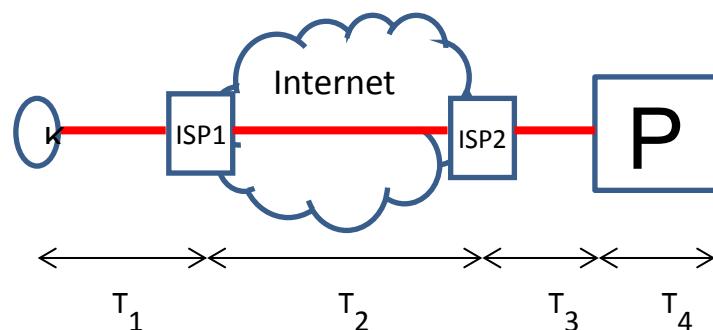
Primjer na sl. 10.1 poslužit će za ilustraciju. Korisnik se putem svojeg davatelja internetske usluge (ISP1) spaja na Internet putem kojega se povezuje na mrežu davatelja internetske usluge (ISP2) na koju je spojen web-poslužitelj P. Na vrijeme odziva korisniku K utječe:

- $T_1$  – vrijeme komunikacije u mreži ISP1,
- $T_2$  – vrijeme komunikacije Internetom,
- $T_3$  – vrijeme komunikacije u mreži ISP2,
- $T_4$  – vrijeme zadržavanja na poslužitelju P.

Komunikacijsko vrijeme zbroj je prijenosnog kašnjenja (vrijeme potrebno za prijenos podataka određenom brzinom) i propagacijskog kašnjenja (vrijeme potrebno za rasprostiranje signala – nositelja podataka na određenu udaljenost) u mreži. Vrijeme zadržavanja na poslužitelju zbroj je vremena čekanja na obradu (zbog zauzetih sredstava: procesna jedinica, memorija, komunikacijski kanali, ...) i vremena posluživanja (obrada podataka pri dohvaćanju web-stranice).

Vrijeme odziva jednako je zbroju vremena komunikacije i zadržavanja u poslužitelju:

$$T = T_1 + T_2 + T_3 + T_4.$$



Slika 10.1. Vrijeme odziva sustava s web-poslužiteljem

Propusnost sustava s motrišta korisnika odgovara broju posluženih zahtjeva u jedinici vremena, u ovom primjer web-stranica u sekundi. Ovisno o dijelu ili razini sustava jedinice propusnost može biti iskazana različitim parametrima, npr. broj IP-datagrama ili broj poruka u sekundi za komunikacijske mreže i sustave, broj instrukcija u sekundi za procesnu jedinicu, broj pretinaca za bazu podataka, a broj transakcija za poslužitelja.

Primjer izračuna propusnosti koji slijedi odnosi se na jednu funkciju poslužitelja – operacije s diskom. Pretpostavimo da ulazno/izlazne operacije diska u poslužitelju za traju prosječno 10 ms. Maksimalna propusnost tj. kapacitet diska iznosi  $1/0,01 = 100$  operacija/s. Ako posluživanje svakog zahtjeva iziskuje operaciju s diskom, time je ograničen i kapacitet poslužitelja: ne može obraditi više od 100 zahtjev/s. Kad obrađuje 100 zahtjev/s, poslužitelj je 100 % zauzet, odnosno potpuno iskorišten.

Pretpostavimo nadalje da ukupno komunikacijsko kašnjenje ( $T_1 + T_2 + T_3$ ) iznosi 100 ms. Maksimalna propusnost cijelog sustava u tom slučaju iznosi  $1/0,1 = 10$  zahtjeva/s, a iskoristivost poslužitelja samo  $10/100 = 0,1$ , odnosno 10 %.

Očevidna je potreba da se poprave performance sustava i postigne mogućnost njegovog razmjernog rasta, sukladno porastu broja korisnika i broja zahtjeva. Pritom ovakvo, determinističko, razmatranje neće biti dovoljno, već će trebati razmotriti stohastičku prirodu procesa posluživanja te moguće paralelne aktivnosti, kao i ograničenja koja proizlaze iz njihove interakcije.

## 10.5 Prirodne granice rasta ubrzanja i kapaciteta

Oblikovanjem raspodijeljenog sustava treba postići prihvatljivo vrijeme odziva i kapacitet dovoljan za posluživanje korisničkih zahtjeva. Vrijeme odziva ovisi o arhitekturi sustava i trajanju operacija pri posluživanju zahtjeva. Ako se kao osnovno rješenje promatra sustav s jednim poslužiteljem i posluživanjem zahtjeva u slijedu – serijski, naprednjim rješenjima kojima se pojedine operacije izvode istodobno – paralelno nastoji se ubrzati rad sustava i skratiti vrijeme odziva. Ubrzanje (engl. *speed-up*) ovisi o arhitekturi sustava i obilježjima tereta i kao takvo ima prirodne granice rasta nakon kojih uvođenje dodatnih paralelnih jedinica ili dijelova sustava više ne pridonosi performancama.

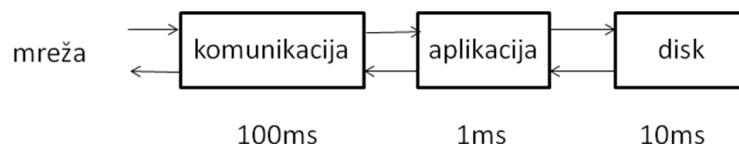
Isto je s kapacitetom raspodijeljenog sustava: svaka izvedba i svaki način proširenja ima svoje prirodne granice. Skalabilnost označava razmjerni rast kapaciteta sukladan porastu opterećenja sustava, a može se postići na dva načina:

- vertikalnim skaliranjem (engl. *scale vertically, scale-up*) pod čime se podrazumijeva zamjena postojećeg poslužitelja poslužiteljem većeg kapaciteta, ili
- horizontalnim skaliranjem (engl. *scale horizontally, scale-out*) koje podrazumijeva dodavanje novog poslužitelja, obično istog kapaciteta.

S istom pažnjom kao pri povećanju kapaciteta (skaliranje na gore) treba pristupiti i smanjenju kapaciteta (skaliranje na dolje) ako se smanjuje opterećenje sustava, kako bi se troškovi rada sustava uskladili s prometom, odnosno prihodima.

#### 10.5.1 Arhitekturama unapređenja

Web-poslužitelj koji će poslužiti kao primjer za arhitekturama unaprjeđenja predočit će se s tri modula „komunikacija“, „aplikacija“ i disk“ (sl. 10.2). Komunikacijski modul apstrahira sve mrežne operacije s vremenom odziva od 100 ms, aplikacijski modul operacije obrade podatka s vremenom odziva 1 ms, a diskovni modul operacije pohrane i dohvata podataka s vremenom odziva 10 ms.

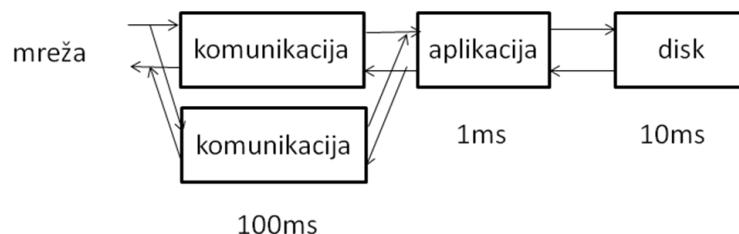


Slika 10.2. Osnovni model web-poslužitelja

Operacije se odvijaju u slijedu (serijski) tako da je ukupno vrijeme odziva 111 ms. Zahtjevi se poslužuju jedan po jedan, tako da maksimalna propusnost – kapacitet iznosi  $1/0,111 \text{ s} = 9 \text{ zahtjeva/s}$ . Promjenama u arhitekturi sustava može se postići skraćenje vremena odziva i povećanje kapaciteta.

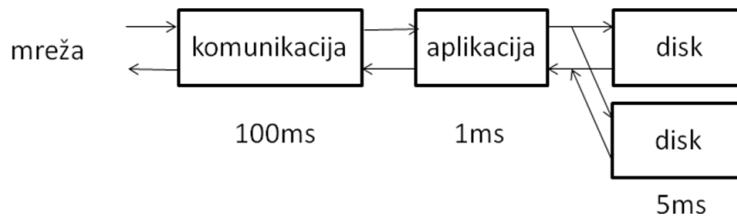
Prvo je unaprjeđenje serijsko preklapanje (engl. *pipelining*) komunikacijskih operacija s obradnjima: tijekom komunikacije jednog zahtjeva, prethodni se obrađuje. Vrijeme odziva ostaje nepromijenjeno (111 ms), ali se povećava kapacitet na  $1/0,1 = 10 \text{ zahtjev/s}$ , jer se svakih 100 ms može obraditi jedan zahtjev.

Veći porast kapaciteta postiže se ako se uvede dodatni, paralelni, pristup mreži (sl. 10.3). Vrijeme odziva opet ostaje nepromijenjeno (111 ms), ali se kapacitet povećava na  $2/0,1 = 20 \text{ zahtjev/s}$ , jer se zbog paralelne komunikacije može priхватiti dvostruko više zahtjeva.



Slika 10.3. Model web-poslužitelja s paralelnim pristupom mreži

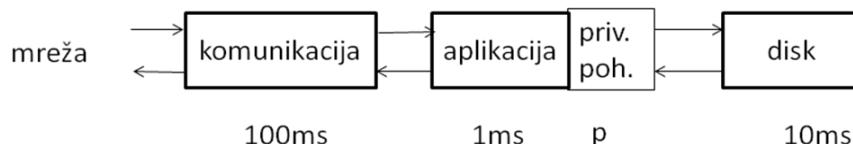
Ako se zapisi istog pretinca raspodijele na dva fizička diska i mogu se dohvaćati istodobno, uvodi se paralelizam u obradu (sl. 10.4).



Slika 10.4. Model web-poslužitelja s paralelnom obradom

Vrijeme odziva diska smanjuje se na 5 ms, što se odražava na ukupno vrijeme odziva (106 ms). Međutim, paralelna obrada ne utječe na kapacitet koji ostaje kao i kod osnovnog modela sa serijskim preklapanjem  $1/0,1 = 10$  zahtjeva/s.

Privremenom pohranom (engl. *cache*) često dohvaćanih podataka isto tako se može utjecati na skraćenje ukupnog vremena odziva, ali bez utjecaja na kapacitet (sl. 10.5).



Slika 10.5. Model web-poslužitelja s privremenom pohranom podataka

Vrijeme odziva smanjuje se ovisno o vjerojatnosti,  $p$ , da se zahtijevani podatak nalazi u privremenom spremniku:

$$T = 100 + 1 + 10(1 - p),$$

tako da primjerice, uz  $p = 0,2$  vrijeme odziva iznosi 109 ms.

### 10.5.2 Ubrzanje i smanjenje vremena odziva

Arhitekturna unaprjeđenja pokazuju da se preklapanjem operacija (npr. komunikacija – obrada) i uvođenjem više istovrsnih jedinica (npr. diskova) koje ih provode paralelno može postići kraće vrijeme odziva i veći kapacitet. To ukazuje na potrebu sustavne analize tereta kako bi se ustanovila mogućnost i način paralelne izvedbe.

Paralelna izvedba zahtijeva više dodatnih podsustava koji istodobno obrađuju teret. Za zadani teret ubrzanje,  $S(p)$ , se definira kao omjer vremena odziva jednog podsustava ( $T_1$ ) i vremena odziva  $p$  paralelnih podsustava ( $T_p$ ):

$$S(p) = \frac{T_1}{T_p}.$$

Analizom tereta treba ustanoviti konkurentne dijelove koji se mogu paralelizirati ( $T_{par}$ ) i dio koji se izvodi serijski ( $T_s$ ) zbog prirode obrade – slijeda operacija, pristupa dijeljenim sredstvima i usklađivanja zajedničkih podataka.

Idealizirani model u kojem se cijelokupni teret može paralelizirati predočen je na sl. 10.6. Teret se raspoređuje na  $p$  paralelnih poslužitelja od kojih svaki preuzima jednaki dio operacija u trajanju  $T_{par}/p$ . Izvedba na jednopošlužiteljskom sustavu ima vrijeme odziva:

$$T_1 = T_{par},$$

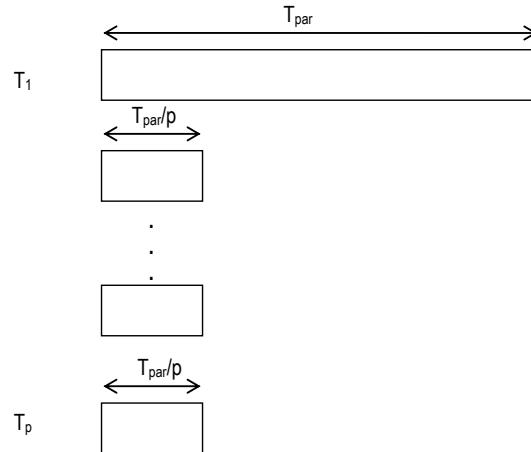
a na sustavu s  $p$  poslužitelja:

$$T_p = \frac{T_{par}}{p},$$

tako da ubrzanje iznosi:

$$S(p) = \frac{T_1}{T_p} = \frac{\frac{T_{par}}{p}}{\frac{T_{par}}{p}} = p.$$

Postignuto je najveće moguće ubrzanje, vrijeme odziva je najkraće, ali je plaćeno visokim ukupnim troškovima vlasništva: izvodi se i u radu održava raspodijeljeni sustav s  $p$  paralelnih poslužitelja.



Slika 10.6. Idealna paralelizacija tereta

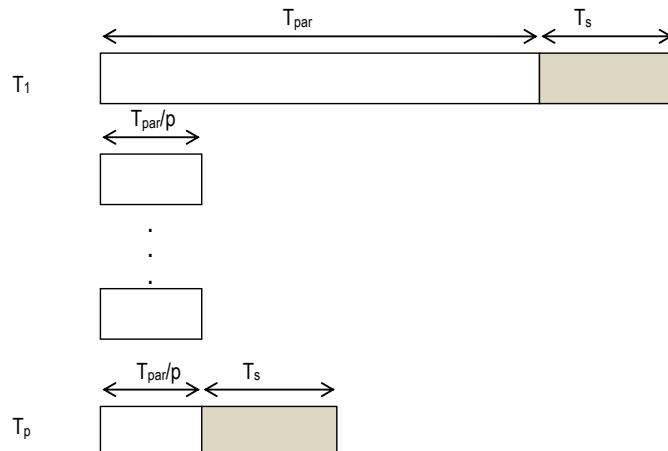
Serijski dio tereta je realnost, jer čak i ako se obrada može u potpunosti paralelizirati, serijski dio ostaje zbog natjecanja za dijeljena sredstva, primjerice komunikacijske jedinice i usklađivanje zajedničkih podataka.

U primjeru na sl. 10.7 teret ima paralelni i serijski dio. Paralelni dio raspoređuje se na  $p$  paralelnih poslužitelja od kojih svaki preuzima jednaki dio operacija u trajanju  $T_{par}/p$ . Izvedba na jednopošlužiteljskom sustavu ima vrijeme odziva:

$$T_1 = T_{par} + T_s,$$

a na sustavu s  $p$  poslužitelja:

$$T_p = \frac{T_{par}}{p} + T_s.$$



Slika 10.7. Paralelni i serijski dio tereta

Udjel serijskog tereta označiti će se sa:

$$s = \frac{T_s}{T_{par} + T_s},$$

a vrijeme odziva opisati izrazom:

$$T_p = sT_1 + (1-s)\frac{T_1}{p},$$

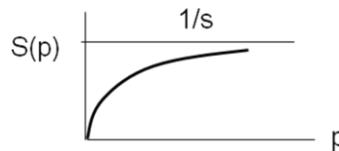
tako da ubrzanje iznosi:

$$S(p) = \frac{T_1}{T_p} = \frac{T_1}{sT_1 + (1-s)\frac{T_1}{p}} = \frac{1}{s + \frac{1-s}{p}}$$

Amdahlovim zakonom<sup>45</sup> naziva se ubrzanje zapisano na drugi način:

$$S(p) = \frac{p}{1+s(p-1)}.$$

Amdahlov zakon govori o prirodnoj granici ubrzanja koja se može postići paralelizacijom: kako  $p \rightarrow \infty$ , tako  $S(p) \rightarrow 1/s$ , što se naziva Amdahlovom asymptotom (sl. 10.8). Za praktičnu primjenu važno je uočiti da se učinkovita paralelizacija može postići samo uz nizak udjel serijskog dijela tereta, odnosno  $s \ll 1$ .



Slika 10.8. Grafička interpretacija Amdahlovog zakona

Djelotvornost (engl. *efficiency*) sustava s paralelnim poslužiteljima se izražava mjerom,  $E(p)$ , kojom svaki paralelni podsustav pridonosi poboljšanju vremena odziva:

$$E(p) = \frac{S(p)}{p} = \frac{1}{1+s(p-1)},$$

odnosno koliko je svaki korisno upotrijebljen.

Kako bismo mogli eksperimentalno odrediti „sastav“ tereta, njegov paralelni i serijski dio? Podsetimo se kako je definirano ubrzanje:

$$S(p) = \frac{T_1}{T_p}$$

Dakle, trebalo bi izmjeriti odziv jednopošlužiteljskog sustava ( $T_1$ ) i sustava s  $p$  poslužitelja ( $T_p$ ) i odrediti  $S(p)$ . Jedina nepoznanica koja preostaje je udjel serijskog tereta, što nakon nekoliko transformacija izraza za ubrzanje daje rješenje:

$$S(p) = \frac{p}{1+s(p-1)},$$

$$1 + s(p - 1) = \frac{p}{S(p)},$$

$$s(p - 1) = \frac{p}{S(p)} - 1,$$

$$s = \frac{\frac{p}{S(p)} - 1}{p - 1}$$

<sup>45</sup> Gene M. Amdahl, američki znanstvenik i poduzetnik, istraživač višeprocesorskih i paralelnih sustava

Uz prethodne modele ubrzanja, primjenjuju se i drugi, od kojih se navode sljedeći:

- geometrijski model ubrzanja,
- kvadratni model ubrzanja i
- eksponencijalni model ubrzanja.

Geometrijski model ubrzanja definiran je funkcijom:

$$S(p) = \frac{1-\Phi^p}{1-\Phi}.$$

Parametar  $\Phi$  ( $0 \leq \Phi \leq 1$ ) označava dio kapaciteta sustava namijenjen samoj aplikaciji – posluživanju tereta, nakon što je izuzet dio koji se koristi za sustavske poslove. Model je prikladan za opis sustava s manjim brojem poslužitelja.

Kvadratni model ubrzanja opisuje relacija:

$$S(p) = p - \gamma p(p-1)$$

u kojoj parametar  $\gamma$  ( $0 \leq \gamma \leq 1$ ) određuje dodatne sustavske operacije potrebne za izvedbu aplikacije – posluživanje tereta. Ubrzanje je maksimalno za:

$$p^* = \frac{1+\gamma}{2\gamma}.$$

Eksponencijalni model ubrzanja definiran je s:

$$\begin{aligned} S(p) &= p(1-\alpha)^{(p-1)} \quad \text{za } 0 \leq \alpha < 1 \\ S(p) &\rightarrow p e^{-\alpha(p-1)} \quad \text{za } p \rightarrow \infty, \end{aligned}$$

a upotrebljava se za analizu performansi komunikacijskih protokola namijenjenih za zajednički medij. Dobre rezultate pokazuje za mali broj čvorova, dok za veći daje preveliki pad ubrzanja.

### 10.5.3 Porast kapaciteta

Horizontalnim skaliranjem postiže se rast kapaciteta s rastom broja paralelnih podsustava. Pritom se mjeri količina tereta koju sustav može poslužiti, odnosno njegova propusnost.

Jednopoloslužiteljski sustav koji u vremenu  $T_1$  može obraditi  $C_1$  zahtjeva, odnosno poslužiti  $C_1$  tereta ima propusnost:

$$X_1 = \frac{C_1}{T_1}.$$

Ako sustav s  $p$  poslužitelja ima propusnost  $X_p$ , njegov relativni kapacitet u odnosu na jednopoloslužiteljski sustav opisan kao:

$$C(p) = \frac{X_p}{X_1},$$

gdje  $C(p)$  govori kako dodavanje paralelnih poslužitelja utječe na kapacitet sustava.

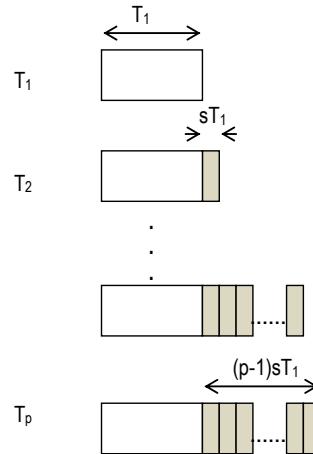
Idealizirani model u kojem poslužitelji ne „troše“ nikakvo vrijeme za međusobnu komunikaciju i usklađivanje dao bi jednostavno rješenje:  $C(p) = p$ . Prvi korak prema stvarnim uvjetima jest uvođenje kašnjenja zbog zajedničkog rada više paralelnih podsustava (natjecanje za dijeljena sredstva, raspoređivanje zahtjeva, ...) koje će se odraziti kao dodatni serijski teret koji produžuje vrijeme odziva jednopoloslužiteljskog sustava (sl. 10.9).

Vrijeme odziva sustava s dva paralelna podsustava koji obrađuje dvostruko više zahtjeva ( $C_2 = 2C_1$ ) iznosi:

$$T_2 = T_1 + sT_1,$$

a propusnost:

$$X_2 = \frac{2C_1}{T_1 + sT_1} = \frac{2\frac{C_1}{T_1}}{1+s} = \frac{2X_1}{1+s}.$$



Slika 10.9. Vrijeme odziva pri horizontalnom skaliranju zbog natjecanja za dijeljena sredstava

U općem slučaju za  $p$  paralelnih poslužitelja:

$$T_p = T_1 + s(p-1)T_1,$$

$$X_p = \frac{pC_1}{T_1 + s(p-1)T_1} = \frac{p \frac{C_1}{T_1}}{1 + s(p-1)} = \frac{pX_1}{1 + s(p-1)},$$

iz čega proizlazi relativni kapacitet:

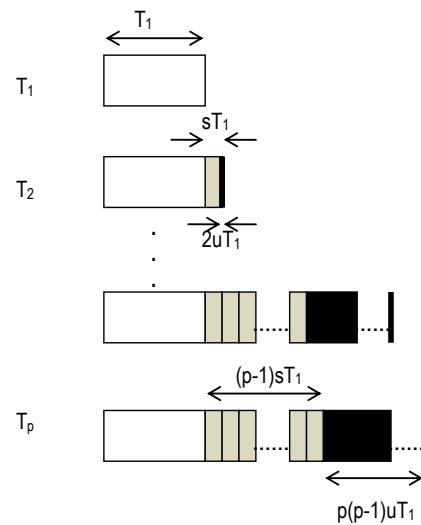
$$C(p) = \frac{X_p}{X_1} = \frac{p}{1+s(p-1)}.$$

Treba uočiti da ova funkcija odgovara Amdahovom zakonu!

Potpuniji i realniji opis dobiva se ako se uključi još i utjecaj usklađivanje podataka. Serijski dio tereta ovisi ne samo o natjecanju za dijeljene resurse, nego i o potrebi usklađivanja podataka  $p$  paralelnih podsustava (izmjena promijenjenih zajedničkih vrijednosti). Pritom se svaki od  $p$  podsustava mora uskladiti s ostalih  $p - 1$  podsustava. Time se prelazi na univerzalni model skaliranja kapaciteta koji je razvio Neil J. Gunther<sup>46</sup>, u kojem serijski teret obuhvaća i dodatno vrijeme potrebno za postizanje konzistentnosti podataka, pri čemu svako pojedinačno usklađivanje traje  $uT_1$ .(sl. 10.10).

---

<sup>46</sup> Neil J. Gunther, australski znanstvenik, istražuje performance računalnih sustava. Razvio programski alat otvorenog koda za modeliranje i analizu performansi PDQ (*Pretty Damn Quick*).



Slika 10.10. Vrijeme odziva pri horizontalnom skaliranju zbog natjecanja za zajednička sredstava i usklađivanje podataka (univerzalni model skaliranja)

Vrijeme odziva za ovaj model skaliranja:

$$T_p = T_1 + s(p-1)T_1 + up(p-1)T_1,$$

ovisi o broju paralelnih podsustava ( $p$ ), koeficijentu natjecanja ( $s$ ) i koeficijentu usklađivanja ( $u$ ).

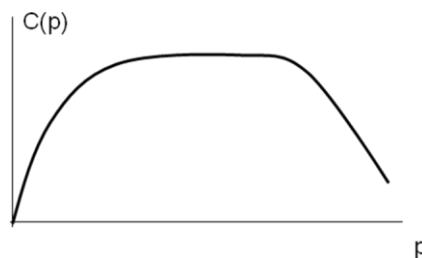
Relativni kapacitet iznosi:

$$C(p) = \frac{x_p}{x_1} = \frac{p}{1+s(p-1)+up(p-1)}.$$

Maksimalni relativni kapacitet postiže  $p^*$  paralelnih podsustava, pri čemu  $p^*$  ovisi o koeficijentima  $s$  i  $u$ . Funkcija  $C(p)$  ima maksimum za sljedeći broj poslužitelja:

$$p^* = \sqrt{\frac{1-s}{u}}.$$

Drugim riječima, povećanje broja paralelnih poslužitelja koje prelazi prirodne granice rasta sustava dovodi do smanjenja kapaciteta, jer podsustavi sve više vremena troše na međusobno natjecanje za sredstva i usklađivanje podataka (sl. 10.11).



Slika 10.11. Grafička interpretacija univerzalnog modela skaliranja kapaciteta

Složeniji je univerzalni model skaliranja grozda računala (engl. *cluster*) koji polazi od sljedećeg:

- $n$  – broj čvorova u grozdu, od kojih svaki sadrži više računala
- $p$  – ukupni broj računala u grozdu
- $p_n$  – broj procesora po čvoru, za homogeni sustav  $p_n = p/n$

- $s_g$  – koeficijent natjecanja između čvorova
- $u_g$  – koeficijent usklađivanja između čvorova
- $C(p_n)$  – kapacitet svakog čvora s  $p_n = p/n$  računala.

Relativni kapacitet grozda određen je izrazom:

$$C(p, n) = \frac{nC(p_n)}{1 + s_g(n - 1)C(p_n) + u_g(n - 1)C(p_n)^2}.$$

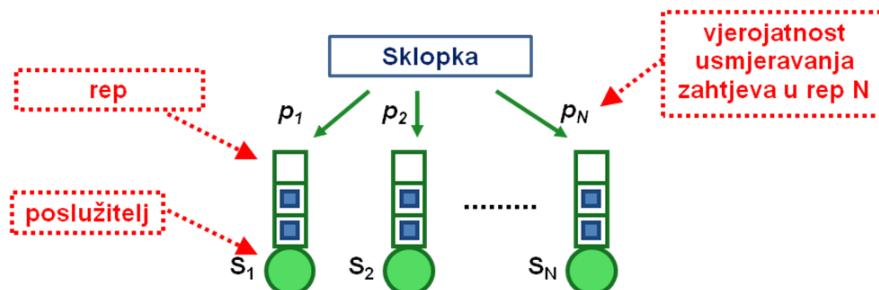
Kapacitet je jednak kapacitetu čvora pomnoženim sa brojem čvorova umanjenom prema univerzalnom modelu skaliranja zbog natjecanja i usklađivanja podataka između čvorova.

#### 10.5.4 Raspodjeljivanje zahtjeva

Horizontalno skaliranje i uvođenje paralelnih podsustava traži rješenja za raspoređivanje opterećenja u raspodijeljenom sustavu kako bi se ostvarilo jednakomjerno opterećenje paralelnih podsustava, raspodjeljivanje podataka uz osiguranje konzistentnosti repliciranih podataka (obrađeno u poglavlju 8. Konzistentnost i replikacija podataka) te usklađivanje rada skupine paralelnih podsustava (obrađeno u poglavlju 7. Sinkronizacija procesa u vremenu).

S motrišta transparentnosti, horizontalnom skaliranju svojstvena je replikacijska transparentnost i s njom povezana konkurenčijska transparentnost i transparentnost na kvar. Činjenica da sustav raspolaze s više funkcionalnih istovrsnih dijelova omoguće postizanje visoke raspoloživosti (engl. *high availability*), odnosno izvedbu sustava neosjetljivu na pogreške (obrađeno u poglavlju 9. Otpornost na neispravnosti). Praktički se koriste dva modela organizacije. U prvom je jedan od podsustava aktivan i poslužuje korisnike, a drugi pripravan (engl. *stand-by*) da preuzme posluživanje u slučaju otkaza aktivnoga. Ovakvo je rješenje prikladno za sustave koji ne pamte stanje (engl. *stateless*) kao što su sustavi weba. Drugi je model s dva aktivna sustava koji međusobno održavaju konzistentnost podataka i usklađuju zajedničke podatke, tako da jedan može preuzeti posluživanje u slučaju ispada drugog. Takvo rješenje za koje je važno stanje sustava, primjereno je bazama podataka. Podsjetimo samo na osnovne postavke raspoređivanja opterećenja, raspodjeljivanja podataka i zajedničkog rada grupe poslužitelja.

Model raspoređivanja opterećenja sklopkom sadrži samu sklopku koja prosljeđuje zahtjeve na skup računala – poslužitelja te skup poslužitelja koji obrađuju zahtjeve. Dolazni zahtjevi čekaju u repu ako i dok je poslužitelj zauzet (sl. 10.12).



Slika 10.12. Model sustava sa sklopkom za raspoređivanje opterećenja

Sklopka može raspodjeljivati zahtjeve na različite načine i prema različitim kriterijima. Disciplina raspoređivanja određuje vjerovatnost raspoređivanja zahtjeva na poslužitelje ( $p_1, p_2, \dots, p_N$ ). Najčešće discipline su kružno raspoređivanje (engl. *Round Robin*) i raspoređivanje po najmanjem opterećenju (engl. *Least Loaded First*). Raspoređivati ima smisla samo na ispravne poslužitelje, tako da sklopka provjerava stanje poslužitelja slanjem ispitnih poruka (engl. *heart beat*) te preskače poslužitelje koji

ne odgovaraju. Raspoređivanje se u Internetu može riješiti na više mjesta, DNS-om, transportnim (TCP) ili aplikacijskim slojem (URL).

Raspoređivanje podataka uključuje replikaciju podataka čime se istovrsni podaci pridjeljuju različitim poslužiteljima te federaciju podataka, odnosno njihovu raspodjelu podataka na grupe koje se pridjeljuju različitim poslužiteljima tako da su dostupni svakom poslužitelju.

Usklađivanjem rada grupe poslužitelja i procesa koji se u njima odvijaju treba osigurati vremenski slijed operacija i dostavu podataka komunicirajućim poslužiteljima, odnosno procesima.

## 10.6 Vrednovanje performansi raspodijeljenih sustava mrežom repova

Prethodno uvedeni kvantitativni modeli vrednovanja performansi raspodijeljenih sustava temelje se na relativno jednostavnoj karakterizaciji tereta i vremena potrebnog za njegovo posluživanje u raspodijeljenom sustavu s više podsustava – poslužitelja. Pritom se promatrao sastav tereta, njegov serijski i paralelni dio, način i posljedice paralelizacije, pri čemu je težište bilo na pitanju KAKO obraditi teret, odnosno poslužiti zahtjev.

Korisnički zahtjevi javljaju se slučajno, a sustav u trenutku nailaska zahtjeva može biti slobodan pa će ga poslužiti odmah, ili zauzet tako da će zahtjev trebati čekati na oslobođanje zauzetih sredstava. Kad količina zahtjeva – tereta nadilazi kapacitet sustava, zahtjev treba čekati na posluživanje u repu. Stoga vrednovanje performansi treba proširiti pitanjima KAD se zahtjev poslužuje i KOLIKO je zahtjeva istodobno u sustavu.

Riječ je o stohastičkim procesima u umreženom sustavu raspodijeljene strukture s međusobno povezanim dijelovima – podsustavima. Takvi se procesi i sustavi sa čekanjem (engl. *queueing system*) mogu modelirati mrežom repova.

### 10.6.1 Osnovni pojmovi teorije repova i model jednopošlužiteljskog sustava

Temeljni model obuhvaća korisnike – izvore (generatore) zahtjeva, rep za čekanje u koji se smještaju zahtjevi i poslužitelje koji preuzimaju zahtjeve iz repa i obrađuju ih (sl. 10.13). Osnovni pojmovi teorije repova pojasnit će se primjenom Kendallove notacije<sup>47</sup>.

Ulazak u poslužiteljski sustav predstavljaju zahtjevi koje treba poslužiti. Korisnici generiraju zahtjeve neovisno jedan o drugima, a za prepostaviti je da je dolazak zahtjeva potpuno slučajan. Zahtjeve opisuju parametri:

- $D$  – raspodjela dolazaka/nailazaka zahtjeva (npr.  $D = M$  – eksponencijalna raspodjela) i
- $Y$  – maksimalni broj zahtjeva ( $Y = m$  zahtjeva ili  $\infty$  zahtjeva).

Rep opisuje:

- $E$  – maksimalni kapacitet repa ( $E = m$  ćelija ili  $\infty$  ćelija) i
- $F$  – disciplina posluživanja (prvi unutra-prvi van (engl. *First In First Out*, FIFO), zadnji unutra-prvi van (engl. *Last In First Out*, LIFO), ...),

a poslužitelje:

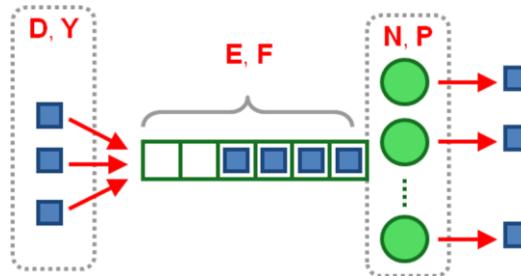
- $P$  – raspodjela posluživanja zahtjeva (npr.  $P = M$  – eksponencijalna raspodjela) i
- $N$  – broj poslužitelja.

Ti su parametri obuhvaćeni Kendallovom notacijom  $D/P/N/Y/E/F$ .

---

<sup>47</sup> David G. Kendall, engleski znanstvenik, definirao notaciju za teoriju repova

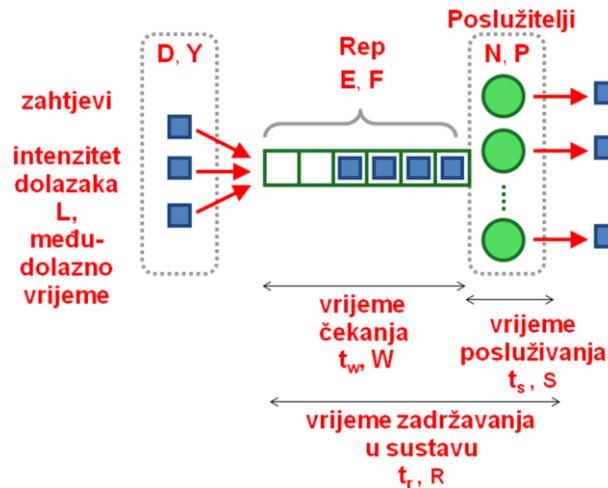
Sustav u kojem broj zahtjeva nije ograničen ( $Y = \infty$  zahtjeva) naziva se otvorenim poslužiteljskim sustavom. Ako se parametri  $Y$ ,  $E$  i  $F$  ne specificiraju, pretpostavlja se da je riječ o otvorenom sustavu s disciplinom posluživanja FIFO. Zatvoreni poslužiteljski sustav je onaj kod kojeg su ograničeni broj izvora zahtjeva i broj zahtjeva ( $Y = m$  zahtjeva), pri čemu izvor može generirati novi zahtjev tek nakon isteka nekog vremena postavljanja zahtjeva.



Slika 10.13. Temeljni model repa opisan Kendallovom notacijom

U nastavku će se veća pozornost usmjeriti otvorenim poslužiteljskim sustavima, jer većina raspodijeljenih sustava djeluje kao otvoreni sustavi kojima se ne može ograničiti broj korisnika, pa tako niti broj zahtjeva. Posebno će se pokazati samo neke specifičnosti zatvorenih poslužiteljskih sustava.

Zahtjevi koji su opisani srednjim intenzitetom dolazaka  $L$  (engl. *arrival intensity*) i međudolaznim vremenom (engl. *interarrival time*), čekaju u repu vrijeme  $t_w$  sa srednjom vrijednosti  $W$  i poslužuju se u vremenu  $t_s$  sa srednjom vrijednosti  $S$ , tako da je ukupno vrijeme zadržavanja u sustavu  $t_r = t_w + t_s$  sa srednjom vrijednosti  $R$  (sl. 10.14).



Slika 10.14. Posluživanje zahtjeva

Povezujući to s prethodnim postupkom analize performansi, teret sustava predočen je raspodjeljom dolazaka zahtjeva i raspodjelom posluživanja zahtjeva, što je dovoljno za izračun srednjeg broja zahtjeva u sustavu i srednjeg vremena zadržavanja u sustavu. Kako će korisnik dobiti odgovor po izlasku zahtjeva iz sustava, srednje vrijeme odziva jednak je srednjem vremenu zadržavanja u sustavu.

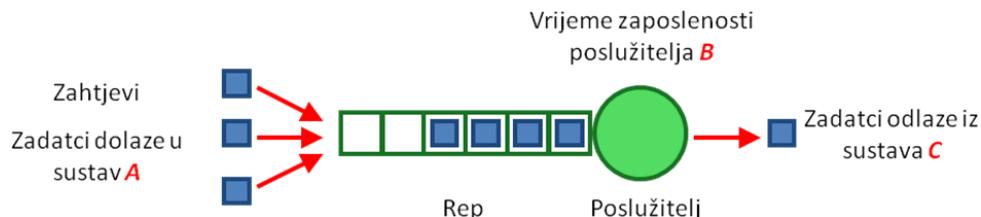
Kako bi se teorijom repova predočio model web-poslužitelja sa sl.10.1? Najprije, riječ je o jednom poslužitelju, tako da treba napraviti model jednoposlužiteljskog sustava. Zatim treba odrediti parametre modela prema Kendallovoj notaciji. Prepostaviti će se eksponencijalna raspodjela

međudolaznih vremena zahtjeva i eksponencijalna raspodjela vremena posluživanja, pri čemu broj zahtjeva i kapacitet repa nisu ograničeni, a zahtjevi se poslužuju disciplinom FIFO. Kako je  $D = M$ ,  $P = M$ ,  $N = 1$ ,  $Y = \infty$ ,  $E = \infty$  i  $F = \text{FIFO}$ , riječ je o modelu M/M/1 prikazanom na sl. 10.15.

U ovom je primjeru ulazni tok zahtjeva definiran kao Poissonov<sup>48</sup> proces s funkcijom raspodjele vjerojatnosti zahtjeva u vremenu:

$$F(t) = 1 - e^{-Lt},$$

uz srednji intenzitet dolazaka,  $L$ . Poissonov proces obilježava neovisnost zahtjeva, tako da su zahtjevi u nekom intervalu promatranja potpuno neovisni o zahtjevima koji su pristigli ranije ili će doći kasnije. Primjenjuju se i druge raspodjele međudolaznih vremena i vremena posluživanja (npr. Erlangova<sup>49</sup>, hipereksponecijalna, ...)<sup>50</sup>.



Slika 10.15. Model jednopoloslužiteljskog sustava M/M/1

Model jednopoloslužiteljskog sustava obilježavaju sljedeći parametri:

- $T$  – vrijeme promatranja rada sustava
- $A$  – broj dolazaka zahtjeva/zadataka (z) u vremenu  $T$
- $B$  – vrijeme kroz koje je poslužitelj zaposlen u vremenu  $T$
- $C$  – broj odlazaka iz sustava u vremenu  $T$

Intenzitet dolazaka zadataka iznosi:

$$L = \frac{A}{T} [\text{z/s}],$$

propusnost sustava:

$$X = \frac{C}{T} [\text{z/s}],$$

srednje vrijeme posluživanja:

$$S = \frac{B}{C} [\text{s/z}],$$

a srednja zaposlenost (zauzetost, opterećenje) poslužitelja:

$$U = \frac{B}{T} = \frac{B}{T} \times \frac{C}{C} = \frac{B}{C} \times \frac{C}{T} = S \times X.$$

Pokažimo to na primjeru diska sa srednjim vremenom obrade zahtjeva pisanja i čitanja od 10 ms koji ispunjava 50 zahtjeva u sekundi, tj.

$$L = 50 \text{ z/s}, X = 50 \text{ z/s}, S = 0,01 \text{ s/z},$$

<sup>48</sup> Siméon Denis Poisson (1781. – 1840.), francuski matematičar i fizičar

<sup>49</sup> Agner Krarup Erlang (1878. – 1929.), danski znanstvenik, začetnik teorije prometa i teorije repova

<sup>50</sup> V. Sinković, „Informacijske mreže“, 5. Osnove analize sustava posluživanja, Školska knjiga, Zagreb, 1994.

a srednja zaposlenost diska je  $U = S \times X = 0,01 \times 50 = 0,5$  (50 %).

Ponašanje ovakvog modela opisano je Littleovim zakonom<sup>51</sup>.

Littleov zakon kaže da je srednji broj zahtjeva u sustavu,  $Q$ , jednak je umnošku intenziteta dolazaka zahtjeva,  $L$ , i prosječnog vremena zadržavanja zahtjeva u sustavu,  $R$ , odnosno:

$$Q = L \times R.$$

Ako je sustav stabilan, broj prispjelih zahtjeva jednak je broju zahtjeva koji napuštaju sustav u vremenu promatranja,  $L = X$ , tako da se Littleov zakon može zapisati i u obliku:

$$Q = X \times R.$$

Ukupno vrijeme zadržavanja zahtjeva u sustavu,  $R$ , jednako je vremenu čekanja,  $W = S \times Q$ , potrebnom za obradu svih  $Q$  neobrađenih zahtjeva u sustavu, uvećanom za vrijeme obrade novog zahtjeva,  $S$ , odnosno uz primjenu Littleovog zakona:

$$R = W + S = S \times Q + S = S \times X \times R + S,$$

$$R = \frac{S}{1-X \times S} = \frac{S}{1-U}.$$

Množenjem obje strane s  $X$  dobije se:

$$X \times R = \frac{X \times S}{1-U},$$

iz čega proizlazi srednji broj zahtjeva u sustavu:

$$Q = \frac{U}{1-U}.$$

Množenjem obje strane sa  $S$  dobije se:

$$S \times Q = \frac{S \times U}{1-U},$$

iz čega proizlazi srednje vrijeme čekanja:

$$W = \frac{S \times U}{1-U}.$$

Primjer koji slijedi se odnosi na mjerjenje na pristupnoj točki mreže kojim se ustanavljava protok paketa od  $X = 125$  paket/s i srednje vrijeme posluživanja paketa  $S = 0,002$  s. Što se može zaključiti o promatranom komunikacijskom sustavu?

Srednja zaposlenost komunikacijskog sustava iznosi:

$$U = X \times S = 125 \times 0,002 = 0,25 \text{ (25 \%)}.$$

Srednje vrijeme zadržavanja paketa u sustavu iznosi:

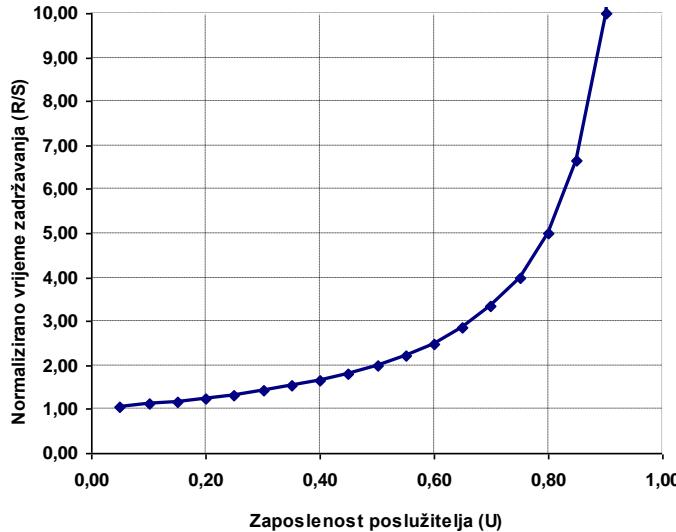
$$R = \frac{S}{1-U} = \frac{0,002}{1-0,25} = 0,002666.$$

Srednji broj paketa u sustavu iznosi:

$$Q = X \times R = 125 \times 0,0026 = 0,333.$$

Vrijeme odziva sustava s čekanjem izrazito je nelinearno, što je predočeno grafom vremena zadržavanja u odnosu na zaposlenost sustava (sl. 10.16).

<sup>51</sup> John D.C. Little, američki znanstvenik, istraživač u području operacijskih istraživanja i marketinga, 1961. g.: "The average number of customers in a system (over some interval) is equal to their average arrival rate, multiplied by their average time in the system." A corollary: "The average time in the system is equal to the average time in queue plus the average time it takes to receive service."



Slika 10.16. Ovisnost vremena zadržavanja o opterećenju sustava M/M/1

Normalizirano vrijeme zadržavanja izraženo odnosom vremena zadržavanja i vremena posluživanja:

$$\frac{R}{S} = \frac{1}{1-U}.$$

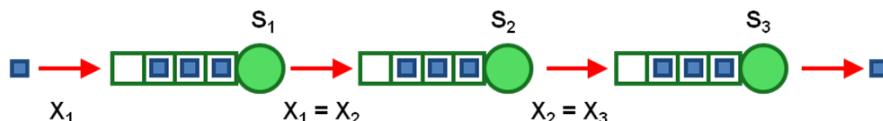
raste nelinearno s porastom zaposlenosti, zbog sve većeg čekanja na posluživanje, što ukazuje na potrebu održavanja radnih uvjeta sustava u području ispod 60-80 % opterećenja.

Jednopolužiteljski modeli nisu dovoljni za vrednovanje performansi raspodijeljenih sustava, pa će se u nastavku razmotriti međusobno povezivanje modela jednopolužiteljskih sustava, sustava s jednim repom, u mrežu repova (engl. *queueing network*). Osnovne konfiguracije su serijske i paralelne mreže repova, te rep s povratnom vezom.

### 10.6.2 Serijski povezani poslužitelji

Serijski povezani poslužitelji i serijski repovi (sl. 10.17) nastaju povezivanjem izlaza jednog poslužitelja (posluženi zahtjevi) s ulazom sljedećeg repa (zahtjevi za poslužiti). U ovom primjeru serijski su povezana tri poslužitelja, a kako je za svakoga broj prispjelih zahtjeva jednak broju zahtjeva koji napuštaju sustav u vremenu promatranja, riječ je o stabilnom sustavu:

$$L = X = X_1 = X_2 = X_3.$$



Slika 10.17. Serijsko posluživanje

Opis serijskog repa Littleovim zakonom slijedi:

$$Q = L \times R = L \times (R_1 + R_2 + R_3),$$

$$Q = X \times R = X \times (R_1 + R_2 + R_3),$$

ili drugčije zapisano:

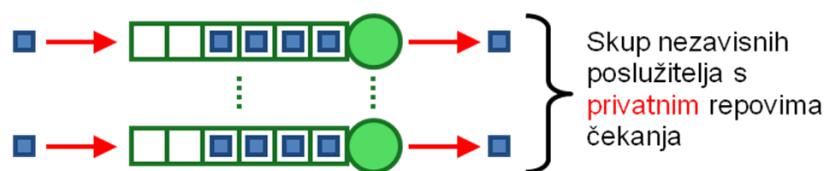
$$Q = X \left( \frac{S_1}{1-XS_1} + \frac{S_2}{1-XS_2} + \frac{S_3}{1-XS_3} \right).$$

Model serijskog repa poznat je iz svakodnevnog života i tipičan je za birokratiziranu administraciju: da bi se obavio neki posao obilazi se urede, tako da ispred svakog treba stati u rep, i neki dokument prenositi redom iz ureda u ured.

### 10.6.3 Paralelni poslužitelji

Paralelno posluživanje zahtjeva može se postaviti na dva načina: više paralelnih repova, svaki za svojeg poslužitelja ili jedan rep zajednički za više paralelnih poslužitelja.

Prvi model, u kojem svaki poslužitelj ima vlastiti rep, predočuje sustav s više računala (sl.10.18). U svakodnevnom životu susreće se u trgovinama – rep ispred svakog naplatnog mjesta.



Slika 10.18. Paralelno posluživanje s privatnim repovima

Vrijeme zadržavanja u ovakovom sustavu iznosi:

$$R = S + (S \times 0,5Q),$$

odnosno primjenom Littleovog zakona:

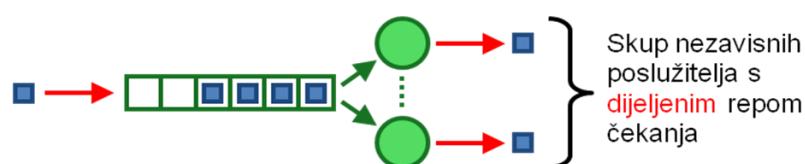
$$R = \frac{S}{1 - 0,5X \times S} = \frac{S}{1 - 0,5U}.$$

Ukupna zaposlenost sustava,  $U$ , podijeljena s brojem poslužitelja,  $N$ , je koeficijent iskorištenja,  $r_o = U/N$ , koji predstavlja vjerojatnost da je poslužitelj zaposlen. Vrijeme zadržavanja u sustavu može izraziti kao:

$$R = \frac{S}{1 - r_o}.$$

Za model paralelnog posluživanja s privatnim repovima vrijeme zadržavanja u sustavu ovisi o broju poslužitelja. Samo u sustavu s beskonačno mnogo poslužitelja ( $N \rightarrow \infty, r_o \rightarrow 0$ ), nema čekanja na posluživanje ( $R \rightarrow S$ )!

Drugi model, u kojem poslužitelji preuzimaju zahtjeve iz dijeljenog repa, predočuje višeprocesorski sustav (sl.10.19). U svakodnevnom životu susreće se u bankama – zajednički rep za sva naplatna mjesta.



Slika 10.19. Paralelno posluživanje s dijeljenim repom

U odnosu na jednopošlužiteljski sustav  $M/M/1$  (sl. 10.14), Kendallovom notacijom se ovakav višepošlužiteljski sustav zapisuje s  $M/M/m$ , gdje  $m$  označava broj poslužitelja.

Vrijeme zadržavanja u ovakovom sustavu,  $R$ , ovisi o dva čimbenika, o broju poslužitelja,  $N$ , i vjerojatnosti da je poslužitelj zaposlen,  $r_o$ . U primjeru s dva poslužitelja, dodatni poslužitelj smanjuje

vrijeme posluživanja za 0,5, a vjerojatnost da je poslužitelj zaposlen iznosi  $r_0 = U/2$ , tako da je vrijeme posluživanja definirano s:

$$S(r_0) = 0,5S \times r_0,$$

a vrijeme zadržavanja u sustavu s:

$$R = S + Q \times S(r_0) = S + 0,5S \times r_0 \times Q,$$

što uz supsticiju prema Littleovom zakonu,  $Q = X \times R$ , daje:

$$R = S + 0,5S \times r_0 \times XR,$$

a kako je  $0,5S \times r_0 \times X = 0,5U = r_0$ , proizlazi sljedeće vrijeme zadržavanja u sustavu:

$$R = S + R \times r_0^2,$$

$$R = \frac{S}{1-r_0^2}.$$

Množenjem s  $X$  i supsticijom  $S \times X = 2r_0$  dobiva se srednji broj zahtjeva u sustavu:

$$Q = \frac{2r_0}{1-r_0^2}.$$

Poopćenje na sustav s  $N$  paralelnih poslužitelja opisano je aproksimativnim rješenjem vremena zadržavanja:

$$R = S + Q \frac{S}{N} r_0^{N-1}.$$

Supsticijama  $Q = X \times R$  i  $S = U/X$  dobiva se:

$$R = S + X \times R \times \frac{U}{X \times N} \times r_0^{N-1},$$

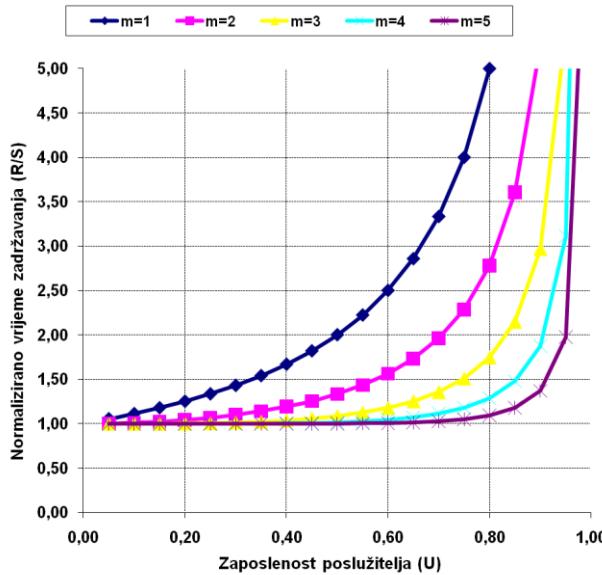
$$R = S + R \times \frac{U}{N} \times r_0^{N-1},$$

$$R = S + R \times r_0^N.$$

Srednji broj zahtjeva u sustavu dobiva se množenjem prethodnog izraza sa  $X$  i supsticijama, redom,  $Q = X \times R$ ,  $U = X \times S$ ,  $U = N \times r_0$ :

$$Q = \frac{N \times r_0}{1-r_0^N}.$$

Kao i za jednopoloslužiteljski sustav (sl. 10.15), pokazat će se ovisnost vremena zadržavanja o zaposlenosti sustava (sl. 10.20). Utjecaj broja poslužitelja na vrijeme zadržavanja u sustavu je očevidan.



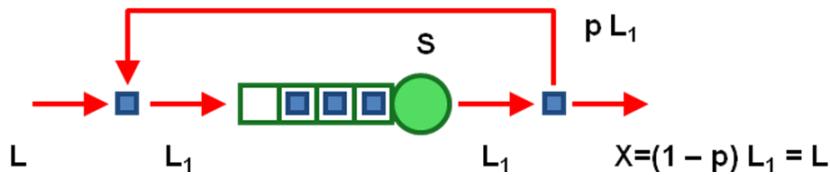
Slika 10.20. Ovisnost vremena zadržavanja o opterećenosti sustava M/M/m

Egzaktno rješenje sustava s paralelnim poslužiteljima dobiva se primjenom Erlangove formule<sup>52</sup>.

#### 10.6.4 Poslužitelj s povratnom vezom

Ako se dio zahtjeva nakon posluživanja ponovno vraća u rep, riječ je o poslužitelju s povratnom vezom (sl. 10.21). Na ulazu u rep superponiraju se dva slučajna procesa: proces nailazaka novih zahtjeva sa srednjim intenzitetom  $L$  i proces zahtjeva vraćenih u rep na ponovnu obradu sa srednjim intenzitetom  $pL_1$ , pri čemu  $p$  označava dio ukupnih zahtjeva vraćen na ponovnu obradu:

$$L_1 = L + pL_1 = \frac{L}{1-p}.$$



Slika 10.21. Poslužitelj s povratnom vezom

Zaposlenost sustava je:

$$U = L_1 \times S = \frac{L \times S}{1-p},$$

a vrijeme zadržavanja zahtjeva za jedan prolaz:

$$R_1 = \frac{S}{1-U}.$$

Ukupno vrijeme zadržavanja u sustavu s povratnom vezom iznosi:

$$R = R_1 \times (1 + \frac{p}{1-p}) = \frac{R_1}{1-p}.$$

<sup>52</sup> V. Sinković, „Informacijske mreže“, 5. Osnove analize sustava posluživanja, Školska knjiga, Zagreb, 1994.

Zamislimo za primjer komunikacijski kanal s teškim smetnjama u kojem se za 30 % prenesenih paketa dogodi pogreška, tako da se mora ponoviti njihovo odašiljanje. Izračunati treba koliko vremena paket provede prosječno u kanalu, ako paketi dolaze s intenzitetom  $L = 0,5 \text{ p/s}$ , a obrada paketa traje  $S = 750 \text{ ms}$ .

Komunikacijski sustav, primjerice usmjeritelja, promatramo kao poslužitelja s povratnom vezom definiranom s  $p = 0,3$ , što odgovara vjerojatnosti pogreške u prijenosu. Odredimo najprije ulazni tok u rep:

$$L_1 = \frac{L}{1-p} = \frac{0,5}{1-0,3} = 0,714 \text{ p/s.}$$

a zatim srednju zaposlenost kanala:

$$U = L_1 \times S = 0,714 \times 0,75 = 0,536 (53,6\%).$$

Srednje vrijeme čekanja paketa u repu iznosi:

$$W = \frac{S \times U}{1-U} = \frac{0,5}{1-0,3} = 0,866 \text{ s},$$

srednje vrijeme zadržavanja paketa u jednom prolazu kanalom:

$$R_1 = W + S = 0,866 + 0,75 = 1,616 \text{ s},$$

ili isti rezultat dobiven na drugi način:

$$R_1 = \frac{S}{1-U} = \frac{0,75}{1-0,536} = 1,616 \text{ s.}$$

Ukupno prosječno vrijeme zadržavanja u kanalu je:

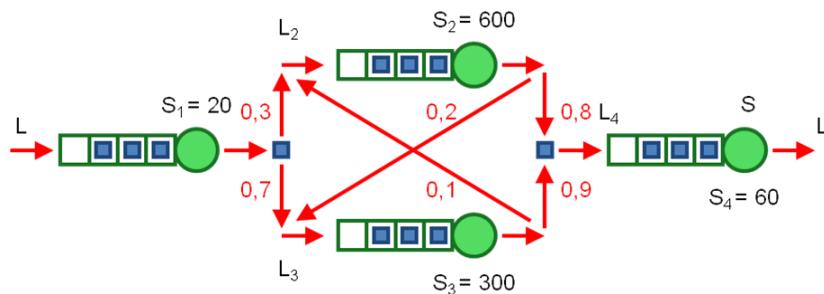
$$R = \frac{R_1}{1-p} = \frac{1,616}{1-0,3} = 2,31 \text{ s.}$$

Posluživanje s povratnom vezom javlja se i u svakodnevnom životu: koliko puta ste se morali vratiti u isti rep, jer nešto niste „znali“?

### 10.6.5 Mreže repova s višestrukim povratnim vezama

Raspodijeljeni sustavi u pravilu su složenije strukture, kako po broju raznovrsnih i istovrsnih dijelova, tako i njihovoj povezanosti i zavisnosti. S motrišta modeliranja, takvi se sustavi mogu predložiti mrežama repova u kojima se kombiniraju osnovne konfiguracije serijskih i paralelnih poslužitelja s višestrukim povratnim vezama.

Započnimo, prije razrade modela web-usluge, s jednim apstraktnim primjerom serijskog posluživanja s tri stupnja (sl. 10.22).



Slika 10.22. Primjer mreže repova s višestrukim povratnim vezama

Prvi poslužitelj u seriji prihvata zahtjeve intenziteta  $L$  koje nakon obrade raspoređuje na dva paralelna poslužitelja od kojih svaki dio zahtjeva vraća na dodatnu obradu drugom poslužitelju. Po završenom posluživanju u srednjem serijskom stupnju, zahtjevi se prosljeđuju zadnjem serijskom

poslužitelju koji zaključuje posluživanje. Za svakog poslužitelja definirano je vrijeme posluživanja,  $S_i$ , te vjerojatnosti raspoređivanja i povrata zahtjeva u rep.

Odredimo prvo intenzitete zahtjeva u stabilnom stanju:

$$L_2 = 0,3L + 0,1L_3$$

$$L_3 = 0,7L + 0,2L_2$$

$$L_4 = 0,8L_2 + 0,9L_3 = L$$

iz čega proizlazi  $L_2 = 0,378L$  i  $L_3 = 0,776L$ .

Srednja zaposlenost svakog od poslužitelja iznosi:

$$U_1 = S_1 \times L = 20L$$

$$U_2 = S_2 \times L_2 = 600 \times (0,3L + 0,1L_3) = 225,6L$$

$$U_3 = S_3 \times L_3 = 300 \times (0,7L + 0,2L_2) = 225,6L$$

$$U_4 = S_4 \times L_4 = 60L$$

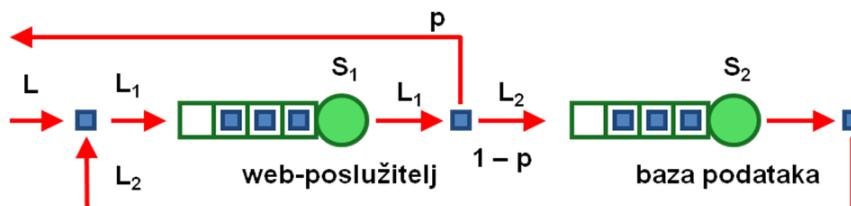
Srednji broj zahtjeva u sustavu po poslužitelju,  $P_i$ , za zadani ulazni intenzitet zahtjeva,  $L$ , može se izračunati prema:

$$Q_i = \frac{U_i}{1-U_i}.$$

a vrijeme zadržavanja zahtjeva u sustavu prema:

$$R = \frac{Q}{L} = \frac{Q_1+Q_2+Q_3+Q_4}{L}.$$

Prelazimo na model web-usluge. Korisnički zahtjevi za uslugom dolaze na web-poslužitelj koji dio zahtjeva može poslužiti izravno, a za drugi dio mora najprije dohvatiti podatke iz baze podataka te nakon toga nastaviti s obradom zahtjeva, što se može ponoviti više puta, ovisno o zahtijevanim informacijama. Riječ je o sustavu s dva serijski povezana poslužitelja,  $P_1$  (web-poslužitelj) i  $P_2$  (poslužitelj baze podataka), svakog sa svojim repom (sl. 10.23). Korisnici generiraju zahtjeve intenziteta  $L$ . Web-poslužitelj izravno poslužuje  $p$  zahtjeva, a  $1 - p$  zahtjeva upućuje poslužitelju baze podataka koji zahtjev vraća na dodatnu obradu u rep web-poslužitelja.



Slika 10.23. Model web-usluge

Intenzitet dolazaka zahtjeva na poslužitelje je sljedeći:

$$L_1 = L + L_2 = L + (1 - p)L_1 = \frac{L}{p},$$

$$L_2 = (1 - p)L_1 = \frac{1-p}{p}L,$$

zauzetost poslužitelja:

$$U_1 = S_1 \frac{L}{p},$$

$$U_2 = S_2 \frac{1-p}{p}L,$$

vrijeme zadržavanja na poslužitelju:

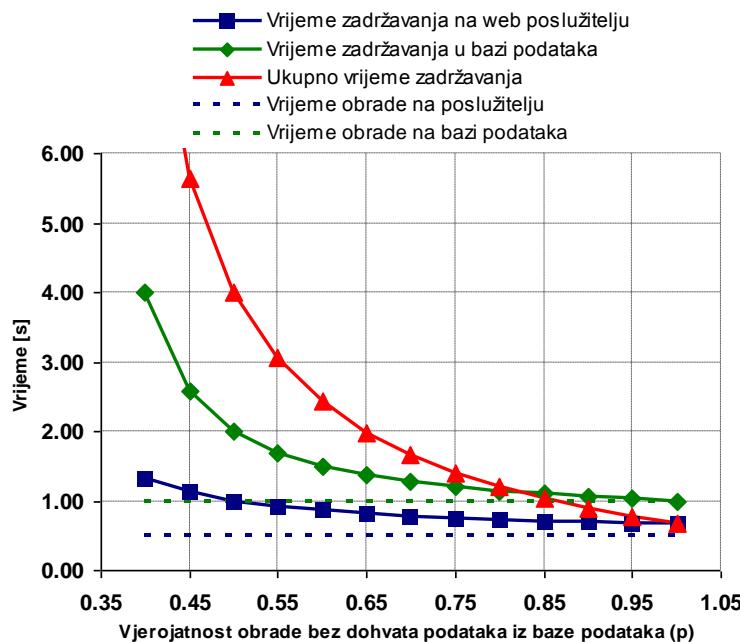
$$R_1 = \frac{S_1}{1 - U_1},$$

$$R_2 = \frac{S_2}{1 - U_2},$$

a ukupno vrijeme zadržavanja u sustavu:

$$R = R_1 \left( 1 + \frac{1-p}{p} \right) + R_2 \frac{1-p}{p}.$$

Utjecaj parametara na performance ovakvog poslužitelja prikazane su grafom na sl. 10.24.



Slika 10.24. Utjecaj parametra na ponašanje web-poslužitelja s bazom podataka

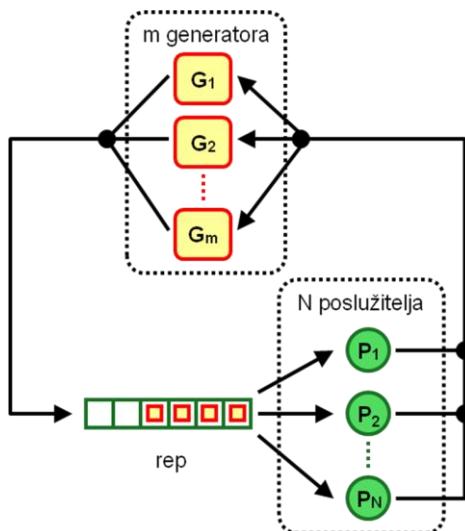
Kada bi se svi korisnički zahtjevi mogli poslužiti bez dohvata podataka iz baze ( $p = 1$ ), ukupno vrijeme zadržavanja bilo bi  $R = R_1$ . Ako za svaki deseti zahtjev web-poslužitelju treba podatke iz baze ( $p = 0,9$ ),  $R = 1,11R_1 + 0,11R_2$ . Pritom treba uočiti da se ne razlikuje prvi dohvata podataka (korisnički zahtjev iz  $L_1$ ) od dodatnog dohvata (povratni zahtjev iz  $L_2$ ) – za srednje vrijeme zadržavanja u sustavu to je svejedno. Kako ovaj model ne ograničava broj ponavljanja dohvata podataka po jednom korisničkom zahtjevu, teorijski bi  $R \rightarrow \infty$  u slučaju kad se nijedan zahtjev ne bi mogao poslužiti bez dohvatanja podataka iz baze ( $p \rightarrow 0$ ).

### 10.6.6 Model zatvorenog poslužiteljskog sustava

Zatvoreni poslužiteljski sustav, kao što je uvodno rečeno, je onaj kod kojeg su ograničeni broj izvora zahtjeva i broj zahtjeva te kapacitet repa, pri čemu izvor može generirati novi zahtjev tek nakon što je prethodni poslužen.

Sukladno Kendallovoj notaciji,  $D/P/N/Y/E/F$ , zatvoren sustav s eksponencijalnom raspodjeljom međudolaznih vremena zahtjeva i vremena posluživanja i  $N$  poslužitelja te brojem zahtjeva i kapacitetom repa ograničenim na  $m$ , opisuje se s M/M/N/m/m (sl. 10.25). Izvori zahtjeva ( $G_1, G_2, \dots, G_m$ ) generiraju zahtjeve koji se smještaju u rep gdje čekaju na obradu na jednom od poslužitelja ( $P_1, P_2, \dots, P_m$ ).

Proces generiranja zahtjeva opisuje dodatni vremenski parametar, vrijeme postavljanja zahtjeva, tijekom kojega izvor može postaviti zahtjev. Izvor nakon postavljanja zahtjeva ostaje zauzet tako dugo dok se ne završi posluživanje tog zahtjeva. Tad izvor postaje slobodan i može postaviti novi zahtjev. Generatori zahtjeva mogu biti u dva stanja, slobodan kad „razmišlja“ o postavljanju zahtjeva ili zauzet nakon što je postavio zahtjev i čeka odgovor. Maksimalni broj zahtjeva u sustavu je  $m$  i odgovara situaciji kad su svi generatori postavili zahtjeve koji čekaju na posluživanje u repu ili se već poslužuju.



Slika 10.25. Model zatvorenog poslužiteljskog sustava

Analitički izrazi za propusnost i srednje vrijeme zadržavanja izvest će za jednopoloslužiteljski zatvoreni sustav  $M/M/1/m/m$ . U takvom sustavu u repu može biti najviše  $m - 1$  zahtjeva, uz jedan na poslužitelju.

Srednja propusnost sustava određena je brojem zahtjeva koji slobodni generatori mogu uputiti tijekom vremena postavljanja zahtjeva,  $Z$ :

$$X(m) = \frac{m - Q}{Z},$$

pri čemu  $Q$  označava srednji broj zahtjeva u sustavu. Množenjem sa  $Z$  te supsticijom  $Q = X \times R$ , dobiva se:

$$X(m) = \frac{m}{R + Z}.$$

Srednja propusnost proporcionalna je broju izvora zahtjeva, a obrnuto proporcionalna zbroju vremena postavljanja zahtjeva i zadržavanja u sustavu.

Da bi moglo usporediti otvorene i zatvorene sustave istih poslužiteljskih mogućnosti, tj. s jednakim brojem poslužitelja,  $N$ , i jednakim vremenom posluživanja,  $S$ , provest će se normalizacija vremena zadržavanja u sustavu kako slijedi:

$$R = \frac{m}{X(m)} - Z,$$

$$\frac{R}{S} = \frac{m}{X(m)S} - \frac{Z}{S'}$$

$$\frac{R}{S} = \frac{m}{U} - \frac{Z}{S'}$$

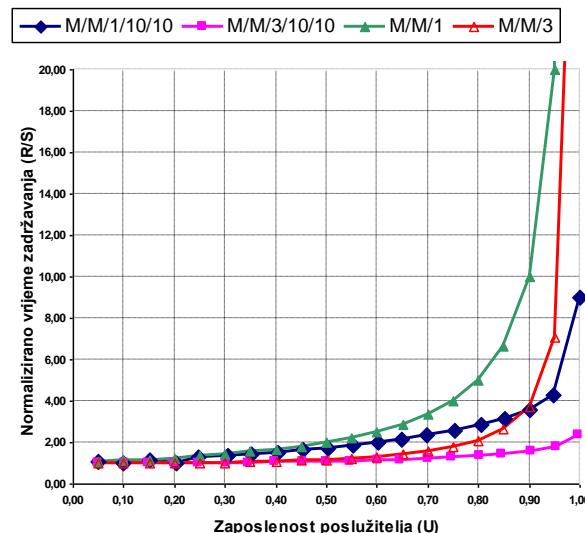
a kako je  $U = r_0 \times N$ :

$$\frac{R}{S} = \frac{m}{r_0 N} - \frac{Z}{S'}$$

U slučaju kad koeficijent iskoristivost  $r_0 \rightarrow 1$ , a vrijeme postavljanja zahtjeva  $Z \rightarrow 0$ , normalizirano vrijeme zadržavanja iznosi:

$$\frac{R}{S} = \frac{m}{N}.$$

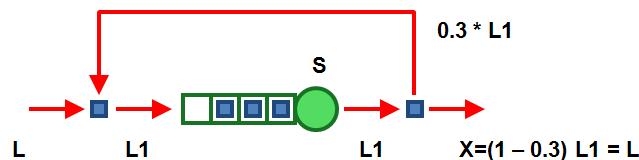
Ono je ograničeno, jer je broj zahtjeva u repu u zatvorenom sustavu ograničen na  $m - 1$ . Grafički prikaz usporedbe otvorenih i zatvorenih poslužiteljskih sustava je na sl. 10.26.



Slika 10.26. Usporedba otvorenih i zatvorenih poslužiteljskih sustava

### Pitanja za učenje i ponavljanje

- 10.1 Disk za trajno spremanje podataka ispunjava 50 zahtjeva u sekundi. Srednje vrijeme obrade zahtjeva operacija pisanja i čitanja je 10 ms. Disk ima prosječno 1 zahtjev u repu. Koliko je prosječno vrijeme čekanja na obradu zahtjeva?
- 10.2 Web aplikacija uključuje podršku korisnicima putem chat usluge. Kupci sami odabiru jedan od 10 repova čekanja. Mjerenja pokazuju da zahtjevi prosječno dolaze 3 upita u minuti te da svaki kupac prosječno čeka 3 minute u repu i prosječno provodi 2 minute u konverzaciji. Koliko je srednje vrijeme zadržavanja kupaca za zadani sustav?
- 10.3 Prikazite elemente osnovnog modela repa čekanja. Koje su osnovne veličine, a koje izvedene u modelu repa čekanja? Kako je definirano stacionarno stanje sustava?
- 10.4 Upiti dolaze na poslužitelj s učestalošću od 12 upita u sekundi te zahtijevaju 0,75 sekundi za obradu. Za 30 % paketa dogodi se pogreška pri obradi te se oni moraju ponovno obraditi. Izračunajte koliko vremena paket prosječno provede u sustavu?

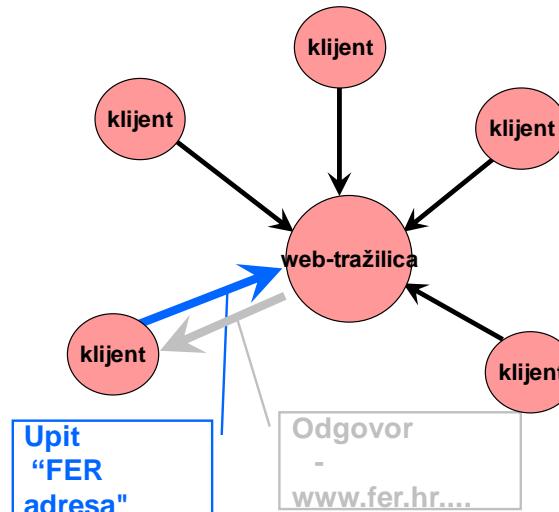


## 11 SUSTAVI S RAVNOPRAVNIM SUDIONICIMA

Sustavi s ravnopravnim sudionicima (engl. *peer-to-peer*, P2P) su raspodijeljeni decentralizirani sustavi međusobno povezanih sudionika (čvorova, peerova) bez hijerarhijske organizacije ili centralnog koordinatora. Koriste se za razvoj decentraliziranih, raspodijeljenih aplikacija i osnova su mnogim popularnim i vrlo raširenim aplikacijama na Internetu kao što su komunikacija porukama, VoIP, dijeljenje podataka i resursa. Aplikacije P2P su zaslužne za veliki rast internetskog prometa te je njihov razvoj značajan za daljnji razvoj Interneta.

### 11.1 Centralizirani i decentralizirani raspodijeljeni sustavi

Centralizirani raspodijeljeni sustavi temelje se na modelu klijent-poslužitelj. U sustavu postoji koordinator koji prihvata zahtjeve i raspodjeljuje ih među ostalim procesima u sustavu, kako je prikazano slikom (Slika 11.1). Tipičan primjer ovakvog sustava je web-tražilica koja podržava veliki broj korisnika. Klijent šalje upit posebnom poslužitelju, koordinatoru, koji prosljeđuje upit u **grozd računala**. Svako od tih računala generira odgovor na korisnički upit, odgovori se vraćaju koordinatoru koji ih pak integrira i kreira rangiranu listu dokumenata te šalje odgovor na upit. Osnovni zahtjev u smislu performansi ovog sustava je vrlo kratko vrijeme odziva te indeks kolekcije dokumenta mora biti pohranjen u radnoj memoriji računala. S obzirom da su kolekcije dokumenata prilično velike, potreban je i veliki broj računala za održavanje njihovog indeksa. Npr. za 100 TB tekstualnih dokumenata generira se indeks veličine 25 TB za čije održavanje treba oko 3.000 računala. Stoga je potrebna infrastruktura izrazito složena i skupa, a generira i izrazito visoke troškove održavanja.

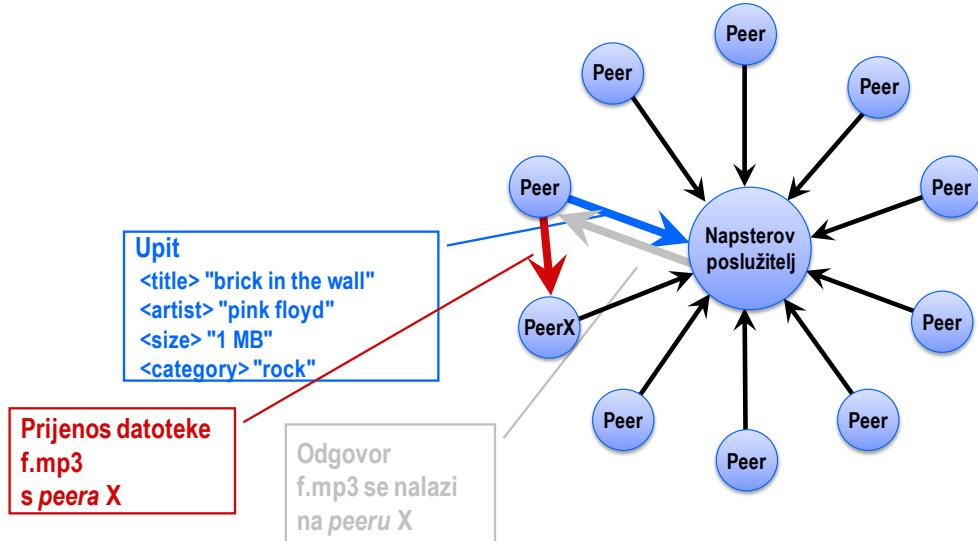


Slika 11.1. Primjer centraliziranog raspodijeljenog sustava (web-tražilica)

Aplikacija za razmjenu datoteka Napster koristi drugačije rješenje kako bi poslužila zahtjeve velikog broja korisnika (Slika 11.2). U ovom je sustavu pretraživanje i dalje centralizirano jer postoji centralizirani indeks s podacima o lokaciji datoteka, ali su resursno zahtjevne funkcije kao pohrana i razmjena datoteka decentralizirani. Tako se u prvom koraku zahtjev prosljeđuje do Napsterovog indeksa, a potom se razmjena datoteka vrši između Napsterovih klijenata. Treba uočiti da je broj potrebnih poslužitelja za održavanje indeksa znatno manji u usporedbi s web-tražilicom.

Napster je primjer centraliziranog sustava P2P koji koristi centralni poslužitelj za pohranjivanje i održavanje indeksa o dijeljenim datotekama. Pri svakom pokretanju aplikacije, *peerovi* javljaju poslužitelju svoju trenutnu IP adresu i podatke o datotekama koje žele dijeliti. Od informacija koje prikupi od *peerova*, poslužitelj dinamički održava centraliziranu bazu podataka koja povezuje imena podataka sa skupom IP adresama. Svi upiti se šalju poslužitelju koji lokalno pretražuje svoju bazu podataka. Ako se pronađe odgovor, uspostavlja se direktna poveznica od peera koji traži do peera koji dijeli podatak te se obavlja direktni prijenos podatka. Nikada podatak koji se pohranjuje neće biti

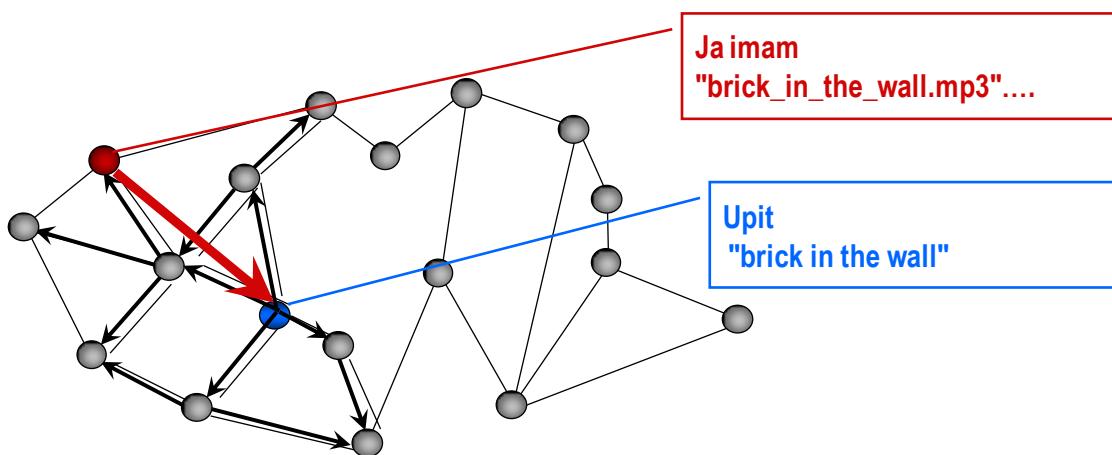
smješten na centralnom poslužitelju, već on ima samo ukupnu listu svih podataka koji se dijele u mreži. Pretraživanje je stoga centralizirano te postoji jedinstvena točka ispada cijelog sustava. Osim Napstera primjeri centraliziranih sustava P2P su seti@home i folding@home.



Slika 11.2. Načelo rada centraliziranog sustava P2P (Napster)

Prednosti ovog rješenja su dijeljenje resursa za zahtjevne računalne operacije čime je početna cijena ulaganja u infrastrukturu znatno niža, a time i cijena održavanja. No, bez obzira na uvriježeno mišljenje korisnika da su aplikacije P2P besplatne, svaki *peer* "plaća" sudjelovanje u mreži vlastitim resursima, npr. disk, mreža, datoteke.

Gnutella je primjer potpuno decentraliziranog sustava koji pretražuje susjedne čvorove s ciljem pronađaska željene datoteke. Za ovakve sustave kažemo da se sastoje od *peerova* koji su međusobno ravnopravni, a globalno ponašanje sustava proizlazi iz lokalnih interakcija između *peerova*. Svi *peerovi* sudjeluju u procesu pretraživanja (ne postoji centralizirani indeks), a posebno je pogodno rješenje za pronađenu datoteke koje su replicirane na velikom broju *peerova*. Ne postoji posebna infrastruktura niti potreba za održavanjem sustava, a time niti jedinstvena točka ispada. Glavni nedostatak ovog rješenja je velika količina generiranog mrežnog prometa pri pretraživanju, a sustav ne može garantirati pronađazak tražene datoteke.



Slika 11.3. Primjer potpuno decentraliziranog sustava

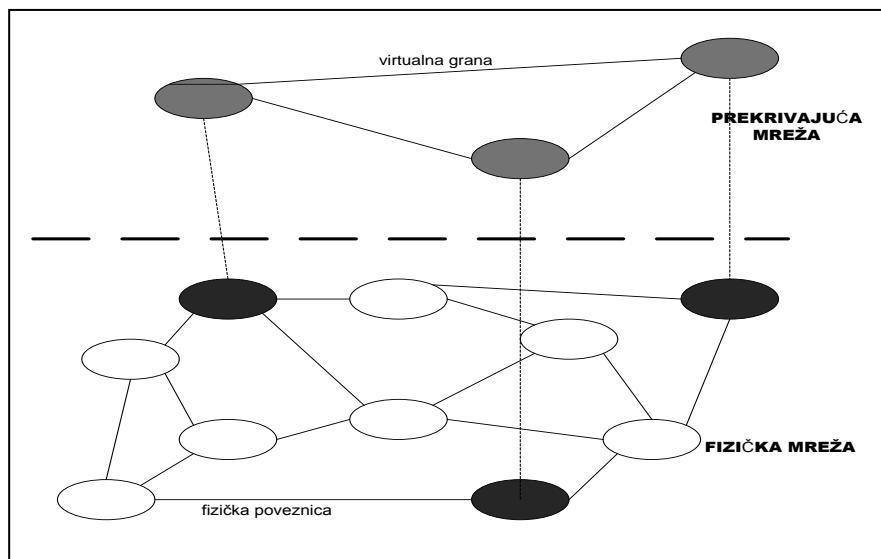
Decentralizirani sustavi se dijele na strukturirane i nestrukturirane. Strukturirani sustavi imaju uređenu mrežnu topologiju koju čini strukturirani graf za razliku od nestrukturiranih mreža koje čine slučajan graf. U strukturiranim sustavima podaci se pohranjuju na točno određenom *peeru* što

zahtjeva dodatne operacije za razliku od **nestrukturiranih** gdje se podatak pohranjuje na *peeru* koji ga je objavio te nikakve dodatne operacije nisu potrebne. Strukturirani sustavi garantiraju pronađak podatka ako postoji u sustavu, međutim mnogo su složeniji i zahtijevaju dodatne tablice usmjeravanja. Nestrukturirani sustavi ne garantiraju pronađak podatka. Pri pretraživanju generiraju veliki mrežni promet, ali daju dobre rezultate pri pretraživanju podataka koji su replicirani na velikom broju *peerova*.

**Definicija sustava P2P.** Sustavi s ravnopravnim sudionicima su decentralizirani raspodijeljeni sustavi koje čini mreža računala u kojoj su svi sudionici međusobno jednaki, istovremeno vrše funkciju i poslužitelja i klijenta. Sudionici, tj. *peerovi* imaju sposobnost samoorganizacije u mrežne topologije sa svrhom dijeljenja resursa kao što su digitalni sadržaj, CPU, podaci ili širina pojasa. Svaki *peer* "plaća" sudjelovanje u mreži nudeći dio vlastitih resursa (memorija, CPU, mreža) ostalim *peerovima* i u načelu ovakav sustav potencijalno nudi neograničene resurse jer broj *peerova* nije ograničen.

Također imaju sposobnost prilagodbe ispadu jednog ili grupe *peerova* te prilagodbe na tranzientne promjene populacije *peerova* uslijed njihovog spajanja i odspajanja iz mreže *peerova*. Stoga je topologija mreža P2P *izrazito dinamična i nestabilna*, a često se stoga u literaturi naziva "samoorganizirajućom" (engl. *self-organizing*) mrežom.

Nastaje tzv. "prekrivajuća mreža" (engl *overlay network*) nad stvarnom mrežnom topologijom jer su *peerovi* programi koji se izvode na aplikacijskom sloju, te koriste resurse krajnjih računala koji čine posebnu mrežu na aplikacijskom sloju neovisnu o mrežnoj topologiji kako je prikazano slikom (Slika 11.4). U fizičkoj mreži dva susjedna računala su povezana fizičkom poveznicom. U prekrivajućoj mreži dva *peera* su susjedi ako imaju otvorenu **TCP konekciju ili ih povezuju virtualne grane** tj. *peer* zna IP adresu drugog *peera*. Ta dva *peera* ne moraju biti i najčešće nisu pravi susjedi u stvarnoj fizičkoj mreži.



Slika 11.4. Prekrivajuća mreža koja povezuje računala na rubu mreže

Postavlja se pitanje kako se održava prekrivajuća mreža *peerova* te kada su 2 *peera* susjedi. Dva *peera* su susjedi kada mogu komunicirati, npr. imaju otvorenu TCP konekciju za komunikaciju ili znaju adresu drugog *peera*. U načelu je dovoljno da *peer* poznaje jednog susjeda te protokol za komunikaciju s tim susjedom i time postaje dio mreže P2P. No kako su ove mreže izrazito dinamične, susjedi često postaju nedostupni. Stoga je potrebno povećavati i održavati listu poznatih susjeda, a za to svaki sustav P2P definira poseban algoritam.

**Osnovna zadaća sustava P2P.** Problem pronađaka resursa u mreži *peerova* je osnovni problem u sustavima P2P. Resurs može biti bilo što (npr. datoteka, podatak) što se može jednoznačno identificirati jedinstvenim ključem. Kako bi pronašli resurs u mreži čvorova, treba znati vezu između

ključa resursa i adrese *peera* koji je zadužen za resurs. Ovo je težak problem u decentraliziranoj mreži.

Kada želimo pronaći podatak u mreži P2P možemo koristiti različite strategije:

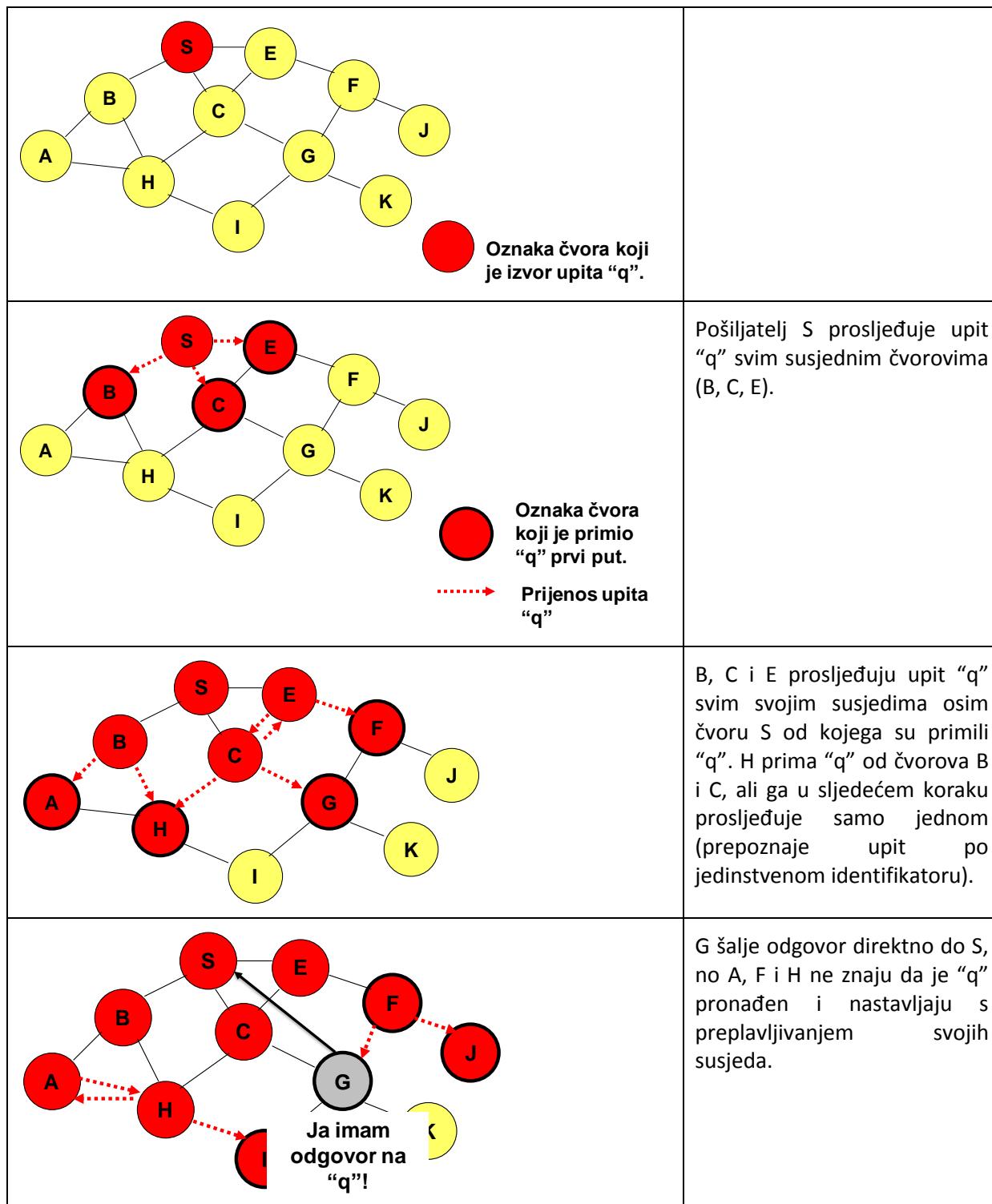
1. Najjednostavnije rješenje je poslati upit svim *peerovima* u mreži, no to nije isplativo rješenje jer prije svega svaki peer mora imati potpuno znanje o mreži, a generira se veliki mrežni promet.
2. Manje naivno rješenje bi koristilo strategiju za slanje upita samo odabranim čvorovima u mreži (npr. samo svojim susjedima u prekrivajućoj mreži), a ti čvorovi ako nemaju traženi podatak mogu proslijediti upit dalje svojim susjedima. Ovo rješenje je praktičnije od prethodnog, no postoje problemi s odabirom "najisplativijih" *peerova* koji bi mogli pohranjivati traženi podatak. Isto tako je nemoguće garantirati da će se traženi podatak svakako pronaći, osim ako opet u najgorem slučaju ne ispitamo sve *peerove* u mreži. Ova se strategija koristi kod nestrukturiranih sustava P2P.
3. Treća strategija je nešto "pametnija" no traži posebne algoritme za održavanje informacija o lokaciji podatka, tj. podatak se pridjeljuje samo određenom *peeru*, a ostali *peerovi* znaju algoritam za pridjeljivanja podatka *peeru* te ga stoga i mogu naći u mreži P2P. Navedena strategija čini osnovu strukturiranih sustava P2P.

## 11.2 Nestrukturirani sustavi P2P

Nestrukturirani sustavi P2P nemaju definiranu strukturu mrežne topologije. Mreža *peerova* čini slučajan graf. Podaci su pohranjeni na *peerovima* koji ih kreiraju, a osim toga ne postoji posebna povezanost između podatka i peera koji ga pohranjuje u smislu dodjele odgovornosti za određene podatke predefiniranim *peerovima*. Moguće je pohraniti kopiju podatka na *peerovima* koji ga kopiraju s originalnog peera. U nestrukturiranom sustavu *peer* zna samo svoje susjede i preko njih pretražuje cijelu mrežu. Pretraživanje se najčešće izvodi preplavljanjem, slučajnim izborom ili prstenastim pretraživanjem s ograničenim TTL-om.

Nestrukturirani sustavi P2P daju dobre rezultate pretraživanja kada su podaci replicirani na velikom broju *peerova* („popularni”), a loše za podatke koji nisu popularni i imaju mali broj replika.

Prilikom preplavljanja, čvor koji je izvor upita "q" šalje navedeni upit svim svojim susjedima, a svaki upit ima jedinstveni identifikator radi sprječavanja ponovljenog preplavljanja istim upitom zbog petlji u mreži. Sljedeći slijed slike objašnjava načelo preplavljanja.



Svaki čvor koji primi upit proslijeđuje ga svim svojim susjedima osim onome od koga je upit primio. Upit "q" ima jedinstven identifikacijski broj koji služi za sprječavanje ponovljenog preplavljuvanja susjeda (svaki čvor vodi evidenciju o upitimima koje je prethodno proslijedio susjedima i zanemaruje onaj koji je prethodno primio i proslijedio). Kada upit dođe do čvora koji na njega može odgovoriti, odgovor se šalje direktno nazad izvoršnom čvoru. Problem je što ostali čvorovi ne znaju da je pronađen odgovor te i dalje preplavljaju svoje susjede. Ovaj se problem rješava na način da se definira parametar TTL (engl. *time-to-live*) koji određuje maksimalni broj koraka od izvoršnog čvora, tj. radius širenja upita (svaki čvor smanjuje TTL za 1 kada primi upit).

S obzirom da se preplavljanjem generira velika količina poruka u mreži peerova, rješenje je neskalabilno u smislu generiranog prometa. Slučajnim izborom susjeda kojima se prosljeđuje upit (npr. svaki peer odabire slučajno 3 susjeda kojima prosljeđuje upit) se smanjuje generirani promet, no ne postoji garancija da će podatak biti pronađen. Također je potrebno definirati TTL radi "zaustavljanja" upita u mreži P2P.

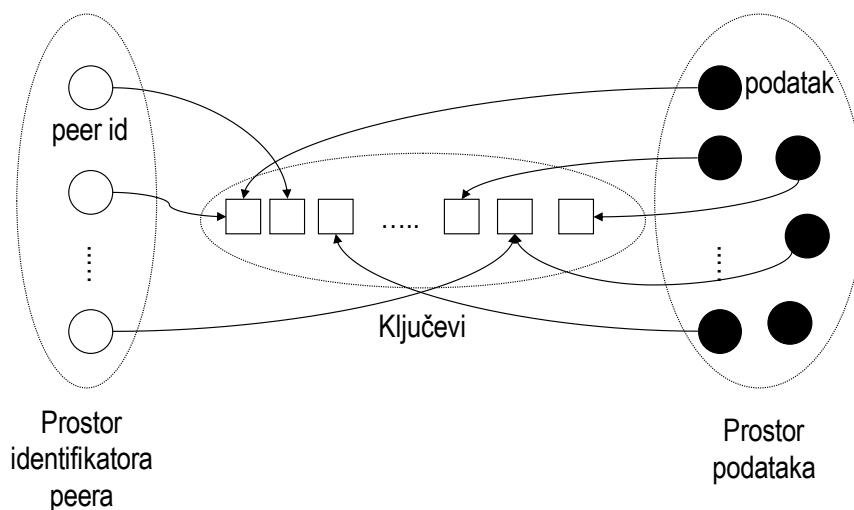
Obilježja nestrukturiranih sustava su sljedeća:

- jednostavnost,
- robusnost budući da ne postoji jedinstvena točka ispada,
- niska cijena objavljivanja novog podatka jer se podatak pohranjuje na čvoru koji ga je kreirao,
- velika cijena pretraživanja podataka jer se generira veliki mrežni promet (tj. mrežni promet raste s  $O(n^2)$ , gdje je  $n$  broj peerova),
- ne postoji garancija pronalaska podatka,
- ne postoji veza između podatka i peer-a na kojemu je podatak pohranjen.

### 11.3 Strukturirani sustavi P2P

Strukturirani sustavi P2P imaju definiranu mrežnu topologiju tj. peerovi međusobno tvore strukturirani graf. Sadržaj nije pohranjen na slučajno odabranom peeru, već na točno određenom peeru što čini upite mnogo učinkovitijima jer je podatkovni objekt smješten na peeru koji je zadužen za ključ koji odgovara jedinstvenom ključu objekta.

Slika 11.5 prikazuje na koji se način povezuju podaci i peerovi. Hash funkcijom podatku pridjelujemo ključ. Na isti način i identifikatoru peer-a pridjelujemo ključ tj. skup ključeva budući da je najčešće broj podataka veći od broja peerova pa jedan peer može biti zadužen za više podataka. Svaki peer zna izračunati ključ  $k$  koji se dodjeljuje podatku  $d$  jer poznaje hash funkciju, tj.  $k=hash(d)$ . Podatak  $d$  se pohranjuje na peeru koji je zadužen za ključ  $k$ , a ne na peeru koji ga je kreirao.



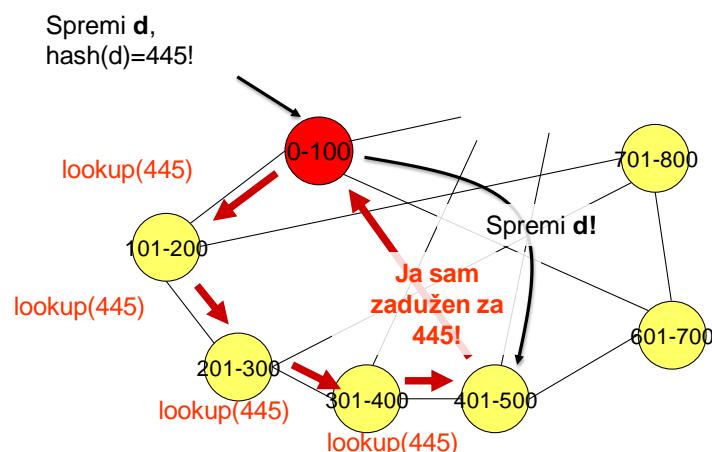
Slika 11.5 Odnos između peer-a, podatka i ključa

Operacija za pohranu podatka na peer je **put(ključ, podatak)**, a operacija pronađaska je **podatak = get(ključ)**. Svaki peer implementira algoritam koji ukoliko peer nije zadužen za podatak koji se pohranjuje ili traži, može prosljediti zahtjev dalje u mrežu peerova, ali do peer-a koji je „bliži“ ključu podatka u prostoru ključeva. Svaki peer u tu svrhu održava tablicu usmjeravanja u kojoj su upisani

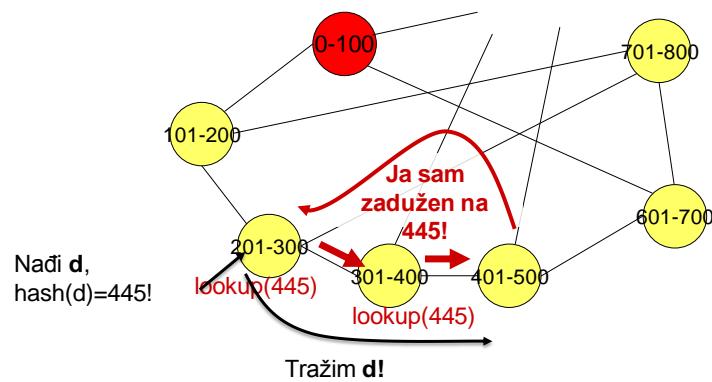
odabrani identifikatori *peerova* povezani s informacijom o prostoru ključeva za koje su zaduženi. Svaki strukturirani sustav P2P definira vlastita pravila za dijeljenje prostora ključeva među *peerovima* te raspodijeljeni protokol za usmjerenje zahtjeva i održavanje tablica usmjerenja na *peerovima*.

Strukturirani sustavi P2P garantiraju pronalazak svakog podatkovnog objekta, ako taj postoji u sustavu, i pohranu podataka u  $O(\log n)$  koraka, gdje je  $n$  broj *peerova* u sustavu.

Slika 11.6 prikazuje ideju usmjerenja u strukturiranim sustavima P2P prilikom pohranjivanja podatka uz napomenu da je ovo samo prikaz ideje radi jednostavnijeg razumijevanja, a ne stvarni algoritam. Oznaka u pojedinom čvoru prikazuje podskup ključeva za koje je čvor zadužen (ovaj prostor je pridijeljen *peeru* na temelju njegovog identifikatora i *hash* funkcije). Svaki čvor ima granu prema svom neposrednom susjedu u adresnom prostoru ključeva, te jednu (ili više) granu prema "udaljenim" čvorovima. Kada primi zahtjev za pohranjivanjem podatka, *peer* prvo izračuna hash vrijednost za podatak te potom koristi funkciju *lookup(k)* radi pronalaženja nadležnog *peera*, gdje je argument funkcije hash vrijednost podatka tj.  $k$ . Ako je ključ manji od područja odgovornosti "udaljenog" čvora, tak čvor opet šalje podatak svome neposrednom susjedu koji će primijeniti isto pravilo. Na kraju će zahtjev *lookup* biti isporučen *peeru* koji je za njega zadužen te će se ovaj čvor javiti izvornom *peeru* koji je poslao zahtjev. Nakon toga se podatak može poslati nadležnom *peeru* koji ga pohranjuje.



Slika 11.6. Ideja usmjerenja prilikom pohranjivanja podatka



Slika 11.7. Ideja usmjerenja prilikom pretraživanja

Usmjerenje upita prilikom pretraživanja temelji se također na raspodijeljenoj izvedbi metode *lookup(k)* kako je ilustrirano slikom (Slika 11.7). Pomoću navedene funkcije se prvo pronađe *peer* koji je nadležan za ključ podatka, a nakon toga se od njega traži podatak.

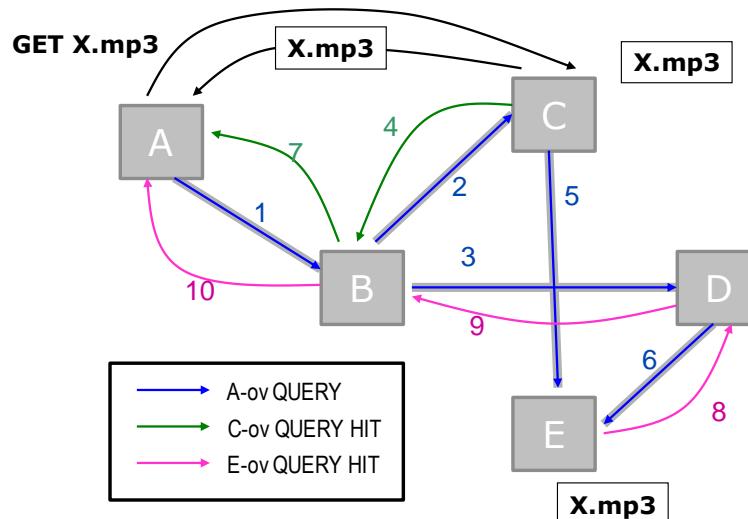
Obilježja strukturiranih sustava P2P su sljedeća:

- garantira se pohranjivanje i pronalaženje podatka u  $O(\log n)$  koraka, gdje je  $n$  broj peerova u mreži što je skalabilno u smislu generiranog prometa u odnosu na nestrukturirane sustave,
- povećana je cijena objavljivanja novog podatka u odnosu na nestrukturirane sustave P2P (jednaka je cijeni pretraživanja),
- podatak se pohranjuje na *peeru* koji je za njega "zadužen" i
- potrebno je održavati dodatne strukture podataka (tablice usmjeravanja) radi umjeravanja upita prema peerovima koji pohranjuju tražene podatke.

#### 11.4 Gnutella

Gnutella je primjer nestruktuiranog sustava P2P koji je razvijen početkom 2000 godine. Aplikacije je povučena s web-sjedišta Nullsofta nedugo nakon javne objave te je aktualni protokol definiran reverznim inženjerstvom. Danas postoji brojne aplikacije (Gnutella klijenti) koje se temelje na Gnutellinom protokolu.

Gnutellini peerovi su međusobno povezani u ravnu ad hoc topologiju te je svaki *peer* istovremeno klijent i poslužitelj, tzv. *Gnutella servent* (SERVer i cliENT). Glavna karakteristika Gnutelle je sposobnost rada u dinamičnom okružju što je temeljna odlika sustava P2P. Zbog raspodijeljene prirode, mreža serventa koji implementiraju protokol Gnutella je otporna na ispadne serventa te mrežne funkcije neće biti onemogućene ako skup serventa postane nedostupan. Ne postoji nikakva uređena topologija mreže pa se novi peerovi spajaju u sustav samo poznavajući neki *peer* koji je već prisutan u mreži, bez ikakvih dodatnih operacija kao što je slučaj kod strukturiranih sustava. Također i pohrana podataka je jednostavna budući da se podatak pohranjuje na *peeru* koji ga je objavio bez ikakvih dodatnih operacija.



Slika 11.8. Pronalaženje i dohvaćanje podatka u Gnutelli

Za pronalaženje podataka *servent* ispituje svoje susjede metodom preplavljanja. Upitom *lookup* se preplavljaju svi susjedi unutar određenog radijusa. Takav dizajn je izrazito otporan na dolazak/odlazak peerova iz sustava. Međutim ovakav mehanizam pretraživanja nije potpuno skalabilan budući da generira veliki promet u mreži. Slika 11.8 prikazuje primjer u kojem se preplavljanjem svih susjeda upitom (QUERY) pronalazi traženi podatak. Odgovor (QUERY HIT) vraća se suprotnim smjerom, a sam prijenos podatka (GET) izvršava se direktno između dva serventa. Pri tome primjerice čvor A s obzirom da je primio dvije poruke QUERY HIT odabire čvor C za dijeljenje datoteke.

Kako bi se novi *servent* priključio postojećoj mreži mora prvo pronaći adresu *serventa* koji je već dio mreže. Novi *servent* šalje poruku GNUTELLA CONNECT/<verzija protokola> jednom ili više *serventa* u mreži koji mogu prihvati konekciju s GNUTELLA OK ili odbiti slanjem bilo kojeg drugog odgovora. Kada se spoje u mrežu, *serventi* odašilju poruke kako bi ostvarili međusobnu interakciju. Odgovori se propagiraju nazad suprotnim putem od inicijalno poslane poruke. Svaka poruka ima jedinstveni identifikator generiran slučajnim izborom. Također svaki *peer* pamti nedavno usmjeravane poruke da bi spriječio ponovno razašiljanje istih poruka. Svaka poruke ima ograničenje propagiranja definirano s vrijednošću TTL.

Poruke koje se koriste u mreži su:

- poruke za održavanje mreže (PING i PONG)

*Peer* pri spajanju na mrežu razašilje poruku PING kako bi objavio svoju prisutnost. Poruka se prosljeđuje svim njegovim susjedima i inicira suprotno propagiranu poruku PONG koja ima informacije o *peeru* koji je primio poruku PONG, kao što su IP adresa, broj i veličina podatka za dijeljenje. Širenje poruka PING i PONG jednako je kao i za poruke QUERY i QUERY HIT.

- poruke za pretraživanje (QUERY i QUERY HIT)

QUERY je upit koji sadrži korisnikov specificirani string za pretraživanje te svaki *peer* koji ga primi uspoređuje string sa svojim lokalno pohranjenim podacima. QUERY HIT je odgovor koji se propagira suprotnim putem od upita, odgovara na upit i ima informacije neophodne za prijenos podataka.

- poruke za prijenos podataka (GET i PUSH)

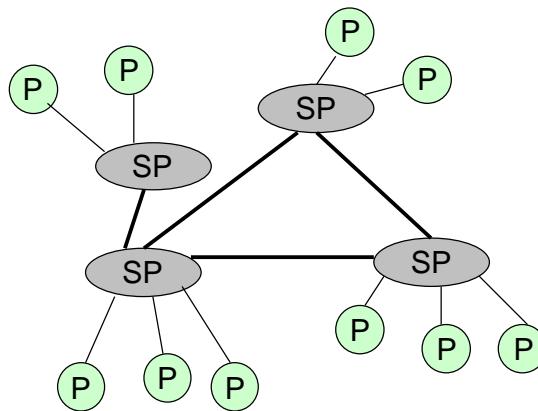
Prijenos podataka se odvija direktno između dva *peera* pomoću ovih poruka. PUSH se koristi u slučaju da je *servent* iza vatrozida.

U Gnutelli se fleksibilnost mreže postiže periodičkim slanjem kontrolnih poruka drugim, susjednim *serventima* kako bi se spriječilo odcjepljenje nekog dijela mreže ako dio *serventa* napusti mrežu te da bi se održala komunikacija između *serventa* koji su u mreži. No Gnutella ima i mnoge probleme koji utječu na njene performance. Problem su poruke PING i PONG koje se konstantno šalju u mreži te čine više od 50% prometa što sprječava brzo širenje poruke kao što su QUERY, QUERY HIT i GET.

Gnutella također ne prisiljava korisnike da dijele podatke što potiče korisnike na tzv. *free riding* tj. većina se pridruži mreži i ne dijeli nikakve podatke. Tako se smanjuje vjerojatnost pronalaska traženih podataka.

Gnutellin protokol nije skalabilan zbog periodičkog širenja kontrolnih poruka preplavljinjem koje zahtijevaju veliki kapacitet. Broj korisnika je ograničen kapacitetom koji leži ispod prekrivajuće mreže, a to je u kontradikciji s osnovnom idejom Gnutelle da podržava neograničeni broj korisnika. TTL polje u zaglavlju poruka i priručna memorija s prethodno viđenim porukama služe za djelomično rješavanje problema skalabilnosti. Poboljšanje performansi je moguće i upotrebom *superpeerova* u novoj arhitekturi sustava P2P i pohranjivanjem rezultata u priručnu memoriju.

Posljednja verzija Gnutelle v6 upotrebljava *superpeerove* kako bi se poboljšale performance usmjeravanja kroz mrežu te se uvodi hijerarhijska organizacija mreže kako je prikazano slikom (Slika 11.9). Kada se *peer* pridružuje mreži prvo kontaktira tzv. *bootstrap peer*. Ukoliko ima odlike postati *superpeer*, *bootstrap peer* mu prosljeđuje listu *peerova* u sustavu za koje će biti zadužen. Ukoliko nema odlike postati *superpeer* spaja se kao krajnji *peer*, odabire neke *superpeerove* i njima objavljuje svoju listu podataka koje dijeli. Upit od krajnjeg *peera* se šalje *superpeeru* koji prvo provjerava svoje lokalne podatke. Ako ima informacije o traženom podatku, odgovara *peeru* na kojem drugom *peeru* se nalazi traženi podatak. Izmjena podataka se odvija direktno između ta dva *peera*. Ako nema odgovor, preplavljuje upitom svoje susjedne *superpeerove* dok ne pronađe odgovor ili do isteka ograničenja broja skokova tj. dok TTL ne postane 0.



Slika 11.9. Primjer hijerarhijske organizacije mreže Gnutella

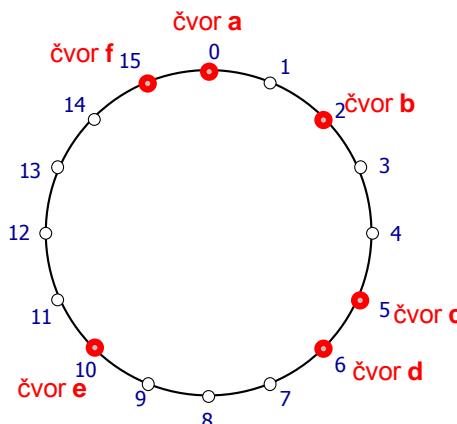
Arhitektura mreže je hijerarhijska budući da objedinjuje mrežu *superpeerova* koji su međusobno povezani na decentralizirani nestrukturirani način i centralizirani način spajanja običnih *peerova* na *superpeerove*. Iako se ovim poboljšanjem smanjuje mrežni promet koji se generira, još uvijek je prisutan mehanizam preplavljanja koji se upotrebljava za komunikaciju među *superpeerovima*. Zbog karakteristika centralnih sustava postoji mala vjerojatnost ispadom *superpeerova*.

## 11.5 Chord

Chord je primjer strukturiranog raspodijeljenog sustava P2P koji se temelji na protokolu čija je zadaća izvršiti samo jednu jednostavnu operaciju – pridijeliti ključ *peeru* u mreži na skalabilan način, pod pretpostavkom da niti jedan *peer* nema globalnu sliku mreže. Sustav je oblikovan tako da uravnoteži opterećenje budući da se svakom *peeru* pridjeljuje gotovo jednak broj ključeva. Kada se *peerovi* pridružuju ili izlaze iz sustava samo mali broj ključeva se premješta među *peerovima*. U stabilnom stanju, za  $n$  *peerova* u sustavu, svaki *peer* održava tablicu usmjeravanja za oko  $O(\log n)$  drugih *peerova*, a upiti generiraju  $O(\log n)$  poruka. Međutim ako se dogodi da informacije nisu ažurne, performance sustava opadaju. Pretraživanje se provodi u ograničenom broju koraka uz garanciju da će podatak biti pronađen. Kada *peer* primi upit, ako ga ne može riješiti jer nije zadužen za traženi ključ, proslijeđuje ga drugome *peeru* koji će s većom vjerojatnošću moći odgovoriti na upit.

Chordov protokol je razvijen na ideji organizacije *peerova* u jednodimenzionalnom prstenu. Prsten se koristi za raspodijeljeno pohranjivanje *hash* tablice tako da je svaki *peer* zadužen za podskup ključeva i njima pridruženih podataka. *Hash* funkcije dodjeljuju *peerovima* i ključevima podataka  $m$ -bitne identifikatore koristeći SHA-1 (Secure Hash Algoritam) tj. skup *hash* funkcija za kriptiranje. *Peerov* identifikator nastaje hashirajući *peerovu* IP adresu, a ključa hashirajući atribut podatka koji se zapisuje u DHT. Ključeve je moguće prezentirati na kružnici koju nazivamo Chordov prsten s modulo aritmetikom gdje je  $m$  broj bitova koji se koristi za kodiranje svakog ključa tj. ključevi su prikazani kao prsten s mjestima označenim brojevima od 0 do  $2^m - 1$  (ovo je ujedno i adresni prostor ključeva u Chordovom prstenu).

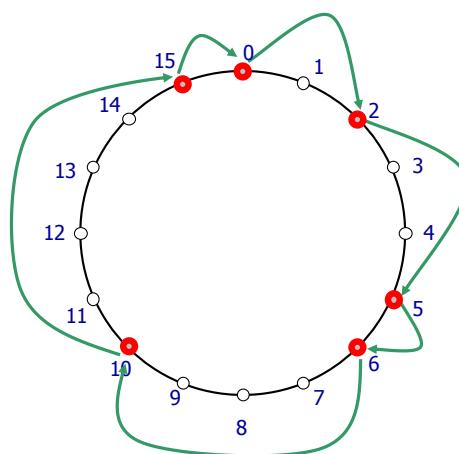
Primjer prstena te smještaja 6 *peerova* **a**, **b**, **c**, **d**, **e**, **f** na prstenu je prikazano slikom (Slika 11.10). Prsten je veličine  $N = 16$  jer je  $m = 4$ , te je  $N$  i broj mogućih ključeva tj. veličina adresnog prostora. *Peerovima* se jednoznačno pridjeljuju ključevi uz pomoć posebne *hash* funkcije  $H_1$ , npr.  $H_1(a) = 0$ . U navedenom primjeru se koristi vrlo mali prostor ključeva veličine  $N=16$  koji je naravno puno veći za realne sustave. U danom primjeru postoji samo 6 *peerova* u mreži (označeni crveno), a ostala mjesta na prstenu se ne koriste jer je broj čvorova u mreži manji od veličine adresnog prostora ključeva. Valja naglasiti da u stvarnom sustavu  $m$  treba biti dovoljno velik da postoji vrlo mala vjerojatnost kolizije za *hash* funkcije, tj. da *hash* funkcija proizvodi isti *hash* kod za različite parametre.



Slika 11.10. Primjer Chordovog prstena

Chord koristi 2 hash funkcije: prva se koristi za pridjeljivanje ključa peerovima, npr.  $H_1(\text{peer\_ID})$ , gdje je peer\_ID IP adresa peera, a druga za pridjeljivanje ključa jednom zapisu u hash tablici, npr.  $H_2(\text{atribut}_x)$  za zapis u tablici ( $\text{atribut}_x$ , vrijednost\_x). Podacima se pridjeljuju ključevi iz istog prostora  $\{0, 1, \dots, 15\}$  koristeći funkciju  $H_2$ . Npr. za zapis u hash tablici (rassus, <http://www.fer.unizg.hr/predmet/rassus>), dobiva se ključ 13 jer vrijedi  $H_2(\text{"rassus"}) = 13$ . Podaci se pohranjuju na čvorovima s istim ključem ako takvi čvorovi postoje u mreži. U suprotnom se pohranjuju na prvom sljedećem čvoru, tj. na prvom čvoru s većim ključem koji se nalazi u smjeru kazaljke na prstenu. Kako u primjeru ne postoji čvor s ključem 13, podatak se pohranjuje na prvom sljedećem čvoru. To je u ovom slučaju čvor f kojemu je ključ = 15.

Zapisivanje podatka u mrežu i pretraživanje mreže s ciljem pronalaska određenog podatka se svodi na pronalaženje *peera* zaduženog za ključ koji je dodijeljen tom podatku. Stoga osnovu svih algoritama u sustavu Chord čini raspodijeljena izvedba upita *lookup* pomoću koje se pronalazi peer zadužen za traženi ključ. Kako bi se mogao realizirati upit *lookup* potrebno je na neki način povezati peerove iz prstena u mrežu tako da svaki peer u svojoj tablici usmjeravanja ima pokazivač samo na svog sljedbenika u prstenu kako je prikazano slikom (Slika 11.11).



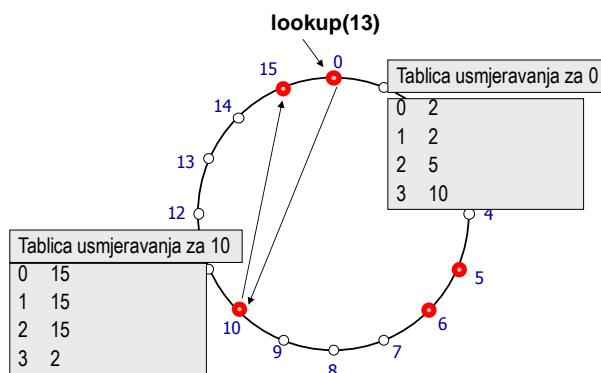
Slika 11.11. Jednostavno povezivanje čvorova na Chordovom prstenu

Ovo je prvo moguće rješenje koje zahtijeva da svaki čvor održava jednostavnu tablicu usmjeravanja, a ta se sastoji od samo jednog čvora (čvora sljedbenika u prstenu). Ovakvu je tablicu usmjeravanja jednostavno održavati premda je mreža dinamična, te se čvorovi mogu po volji spajati i odspajati iz sustava. Ali ovakvo rješenje nije učinkovito u smislu generiranog prometa te vremena odziva sustava jer će u najgorem slučaju za pronalaženje peera zaduženog za neki ključ trebati kontaktirati sve peerove na prstenu (složenost pretraživanja je  $O(n)$ ).

Chord predlaže drugačiju organizaciju tablice usmjeravanja koja ima  $m$  zapisa. Svaki čvor s ključem  $k$  osim što pokazuje na svog sljedbenika, održava pokazivače i na čvorove sa sljedećim ključem:  $k+2^0, k+2^1, k+2^2, \dots, k+2^{i-1}, \dots, k+2^{m-1}$ . U slučaju kada na prstenu ne postoji čvor s odgovarajućim ključem, u tablicu usmjeravanja se zapisuje prvi sljedeći čvor na prstenu. Primjerice za čvor s ključem 15, tablica usmjeravanja bi izgledala ovako:

$i=0$	$15+2^0=0$	$\rightarrow \text{čvor } 0$
$i=1$	$15+2^1=1$	$\rightarrow \text{čvor } 2$
$i=2$	$15+2^2=3$	$\rightarrow \text{čvor } 5$
$i=3$	$15+2^3=7$	$\rightarrow \text{čvor } 10$

Općenito, svaki *peer* čuva informaciju samo o malenom broju drugih *peerova* ( $m$ ) i zna više samo o *peerovima* koji ga blisko slijede na Chordovom prstenu. Tablica usmjeravanja koju posjeduje svaki *peer* ne sadrži uvijek dovoljno informacija za direktno pronalaženje nekog zadanog ključa  $k$ . *Peer* mora proslijediti upit drugim *peerove* kako bi pronašao traženi ljuč, no pri tome uvijek usmjerava upit do *peera* koji se u adresnom prostoru nalazi bliže traženom ključu od njega samoga.

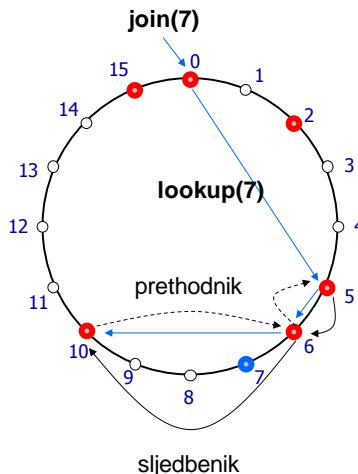


Slika 11.12. Primjer pretraživanja u Chordovom prstenu

Na slici (Slika 11.12) je prikazan primjer skalabilnog pretraživanja ključa 13 uz pomoć tablica usmjeravanja koje posjeduje svaki čvor. Budući da čvor s ključem 0 ima u tablici podatke samo do čvora s ključem 10, proslijedi se upit tom čvoru. Čvor 10 nije zadužen za ključ 13 te s obzirom da ne može direktno pronaći traženi ključ u svojoj tablici usmjeravanja, proslijedi upit svom sljedbeniku jer je taj čvor zadužen za ključ 13.

Općenito, kada *peer* dobije upit za određenim ključem za koji nije zadužen i o kojem nema informacije u svojoj tablici usmjeravanja, proslijedi upit najvećem mogućem prethodniku tog ključa iz svoje tablice usmjeravanja. Ta procedura se odvija rekursivno sve dok se ne odredi *peer* koji je odgovoran za taj ključ. Svaki *peer* ima tablicu usmjeravanja konstruiranu tako da u svakom koraku može proslijediti upit za minimalno pola preostale udaljenosti po prstenu pri pronalasku *peera* koji je odgovoran za traženi ključ.

**Dodavanje *peera* u prsten.** Radi dodavanja novog *peera* u prsten, Chord na svakom peeru održava informaciju o peerovom sljedbeniku i prethodniku. Novi čvor  $p_n$  prvo mora pronaći svoje mjesto na prstenu i stoga šalje zahtjev  $lookup(k)$ , gdje je  $k=H_1(p_n)$  kako bi pronašao svog sljedbenika. Na primjeru sa slike (Slika 11.13) novi čvor čiji je ključ  $k=7$  šalje zahtjev za spajanje u mrežu preko čvora 0. Čvor 0 stoga koristi metodu  $lookup(7)$  koja završava na čvoru 10 jer je taj zadužen za ključ 7. S obzirom da čvor 10 ima pokazivač na svog prethodnika, a to je čvor 6, novi čvor oznake 7 pronalazi svoje mjesto na prstenu, tj. dobiva informaciju od čvora 10 o svome prethodniku kojemu se javlja i predstavlja kao novi sljedbenik. Na taj način novi čvor oznake 7 podešava svoga prethodnika i sljedbenika, a isto čine i čvorovi s ključem 6 i 10.



Slika 11.13. Slanje zahtjeva za spajanjem u mrežu

U sljedećem koraku novi čvor  $p_n$  treba definirati vlastitu tablicu usmjeravanja. Za to koristi svog prethodnika koji izvršava operaciju  $lookup\ m$  puta, tj.  $lookup(k+2^i)$ , te rezultate šalje čvoru  $p_n$  koji popunjava tablicu usmjeravanja. Na primjeru sa slike (Slika 11.13) čvor označen brojem 7 koristi svog prethodnika, a to je čvor označen brojem 6, kako bi popunio svoju tablicu usmjeravanja. Stoga čvor označen brojem 6 izvršava sljedeće operacije:  $lookup(7+1)$ ,  $lookup(7+2)$ ,  $lookup(7+4)$  i  $lookup(7+8)$  na temelju kojig tablica usmjeravanja čvora s označom 7 izgleda ovako:

Tablica usmjeravanja za čvor označen brojem 7	
0	10
1	10
2	15
3	15

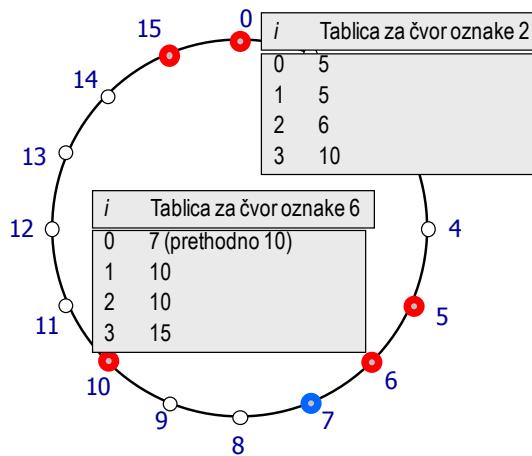
Posljednji korak procedure za dodavanje novog čvora je popravljanje tablica usmjeravanja na Chordovom prstenu tako da ostali čvorovi postanu svjesni novog čvora. Za to se koristi sljedeći algoritam:

```

za i=0 do m-1
    lookup (k-2i);
    ako si došao do čvora većeg od k-2i vrati se do njegovog prethodnika
        koji postaje peer t
    inače peer t = k-2i;
    ako je element iz i-tog retka tablice usmjeravanja na peeru t veći od k,
        promijeni i-ti redak u tablici usmjeravanja na k
  
```

Primjerice za prsten na slici (Slika 11.14) vrijedi sljedeće:

- Za  $i=0$ ,  $lookup(7-1)$  vraća adresu čvora označenog brojem 6, a u tablici na čvoru 6 se mijenja prvi redak u  $i=0$ , 7.
- Za  $i=1$ ,  $lookup(7-2)$  vraća adresu čvora označenog brojem 5, u tablici na čvoru 5 mijenja se drugi redak u  $i=1$ , 7.
- Za  $i=2$ ,  $lookup(7-4)$  vraća adresu čvora označenog brojem 5 što je veće od 3 pa se vraćamo na njegovog prethodnika, a to je čvor označen brojem 2. U tablici na čvoru 2 se treći redak ( $i=2$ ) ne mijenja jer je trenutna vrijednost 6 što je manje od 7.
- Za  $i=3$ ,  $lookup(7-8)=lookup(15)$  vraća adresu čvora s označom 15, a u njegovoj tablici usmjeravanja mijenja se četvrti redak iz 10 u 7 (jer je  $10 > 7$ ).



Slika 11.14. Tablice usmjeravanja nakon dolaska novog čvora oznake 7 u mrežu

Prilikom izlaska peer-a iz *Chordovog* prstena čvor prvo o tome obavještava svog prethodnika i sljedbenika koji podešavaju pokazivače, a potom se ponavlja procedura popravljanja tablica na ostalim čvorovima, slično kao prilikom dodavanja novog čvora u mrežu. Kako bi Chordov protokol bio otporan na ispade, koristi se aktivno propitivanje stanja prethodnika kako bi se zabilježio njegov ispad. Neki čvor može popraviti pokazivač na svog prethodnika samo ako ga drugi čvorovi dodaju kao svoga sljedbenika. Stoga uvodi redundanciju tako da svaki čvor održava cijelu listu sljedbenika, a ne samo jednog (duljina liste je poseban parametar, označava se s  $r$ ). Ako prvi sljedbenik u listi ne odgovara na upite, čvor može kontaktirati sljedećeg na listi.

Obilježja protokola Chord su sljedeća:

- ravnomjerno opterećenje čvorova jer je za mrežu od  $n$  čvorova, svaki čvor zadužen za najviše  $(1+\epsilon)N/n$  ključeva, gdje je  $N$  veličina adresnog prostora ključeva
- skalabilnost algoritma lookup s obzirom da je potrebno kontaktirati  $O(\log_2 n)$  čvorova
- skalabilna veličina tablice finger jer ovisi o  $m$

No negativna svojstva su vezana u složenost samog algoritma za održavanje prstena. Nužno je određeno vrijeme stabilizacije za slučaj velikih promjena u mreži radi održavanja prstenaste strukture.

## 11.6 Usporedba protokola Gnutella i Chord

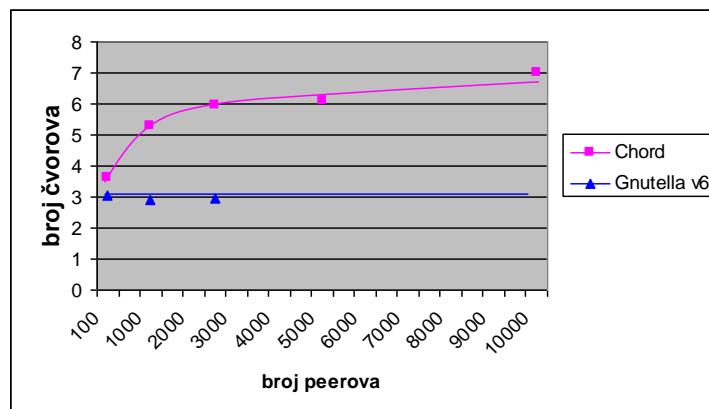
U ovom su poglavljju pokazani rezultati usporedbe algoritama Chord i Gnutella pomoću simulatora PeerfactSim.KOM<sup>53</sup>. U prvom eksperimentu se ispituje skalabilnost algoritama u statičnom scenariju kada tijekom ispitivanja nema značajnih promjena mreže. Koristi se scenarij promjenjive veličine mreže tj. spajanja grupe čvorova koji potom objavljaju svoje podatke, te započinju pretraživanje.

Slike (Slika 11.15, Slika 11.16 i Slika 11.17) prikazuju rezultate prvog eksperimenta.

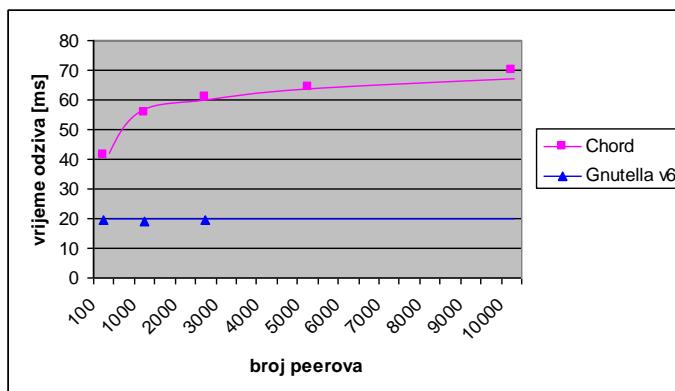
Vidljivo je da s povećanjem broja peerova u mreži Chord raste broj skokova i vrijeme odziva. Taj porast nije linearan, već se uočava logaritamski porast, što dokazuje da je složenost pretraživanja kod Chorda  $O(\log n)$  pri čemu je  $n$  broj peerova u mreži. Učinkovitost pretraživanja, tj. postotak uspješno odgovorenih upita je velika što pokazuje slika 11.17.

<sup>53</sup> <http://peerfact.kom.e-technik.tu-darmstadt.de/de/downloads/>

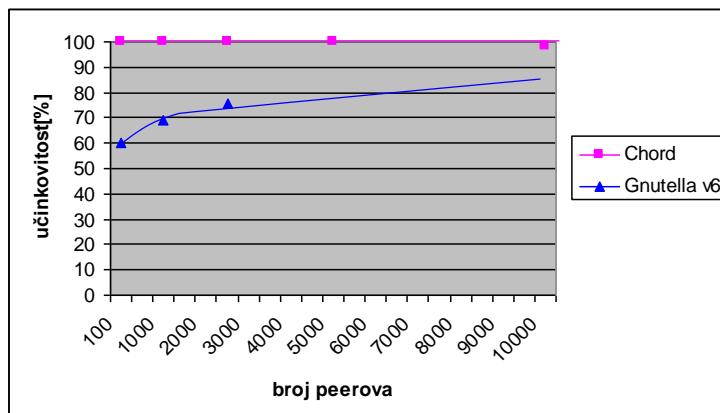
Iz slike 11.15. i 11.16. vidljivo je da je kod Gnutelle v6 gotovo jednako vrijeme odziva i broj skokova bez obzira na broj peerova u sustavu, međutim učinkovitost pronaleta podataka raste s povećanjem broja peerova zbog veće povezanosti peerova u mreži što je uočljivo iz slike 11.17.



Slika 11.15. Prosječan broj kontaktiranih čvorova po upitu



Slika 11.16. Prosječno vrijeme odziva



Slika 11.17. Postotak uspješno odgovorenih upita

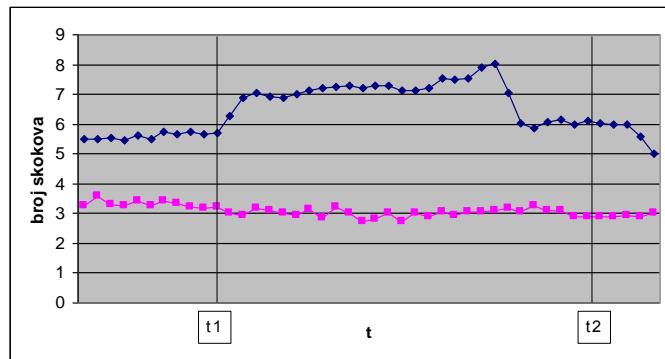
Sljedeći eksperiment testira svojstvo stabilnosti sustava prilikom značajnih promjena u mreži. Stabilan sustav je onaj koji može održati sve funkcionalnosti pri neočekivanim i iznenadnim

promjenama, a to je sustave P2P istovremeni dolazak ili odlazak velikog broja peerova u mrežu (engl. *churn*).

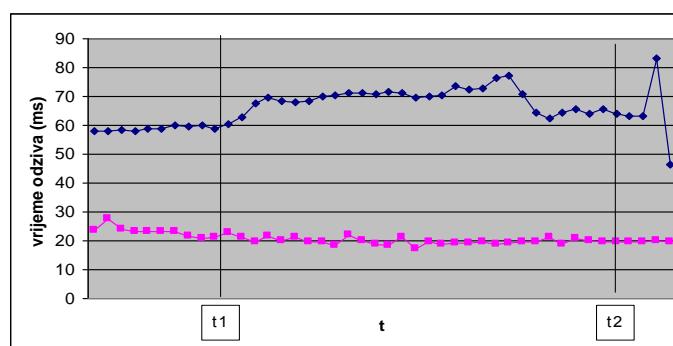
Eksperiment za scenarij *churn* je izведен na sljedeći način:

- u mreži se nalazi 1000 peerova koji obavljaju operacije pretraživanja podataka,
- u trenutku  $t_1$  spaja se 2000 novih čvorova u mrežu, a
- u trenutku  $t_2$  iznenadno se odspaja 2000 čvorova.

Na slikama 11.18. i 11.19. dani su rezultati eksperimenta za scenarij *churn*. Vidljivo je da su vrijednosti za Chord veće od očekivanih prosječnih vrijednosti, a u intervalu od  $t_1$  do  $t_2$  maksimalne pojedinačne vršne vrijednosti su bile i do 12 skokova i 120ms vremena odziva što je dvostruko veće od prosječnih vrijednosti u normalnim uvjetima. Razlog tome je što se u prstenu nije uspio odviti proces stabilizacije pa tablice usmjeravanja nisu ažурне te usmjeravanje nije skalabilno. Rezultati eksperimenta s očekivani za Gnutella v6. Očito je da Gnutella v6 ima stabilne performance koje se ne pogoršavaju ni dolaskom/odlaskom velikog broja peerova iz mreže jer je protokol jednostavan čime se osigurava njegova robustnost. Chord je izrazito nestabilan prilikom scenarija *churn* zbog procesa stabilizacije koji nije završen, a uspješnost mu je smanjena na 55%, dok je Gnutella neosjetljiva na *churn* osim što se bilježi nešto smanjena uspješnost.



Slika 11.18. Usporedba prosječnog broja kontaktiranih čvorova po upitu za Chord i Gnutellu pri testiranju stabilnosti



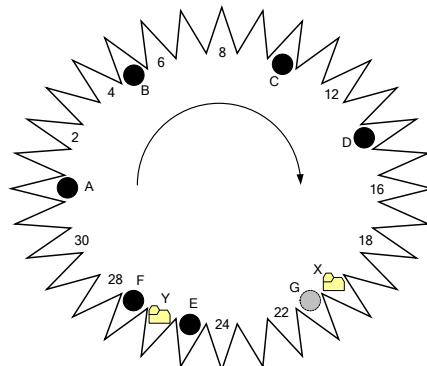
Slika 11.19. Usporedba vremena odziva za Chord i Gnutellu v6 pri testiranju stabilnosti

## 11.7 Pitanja za učenje i ponavljanje

- 11.1. Uspoređite svojstva centraliziranih i decentraliziranih raspodijeljenih sustava na primjeru.
- 11.2. Kako se izvodi pretraživanje kod strukturiranih, a kako kod nestrukturiranih sustava sustava P2P (*peer-to-peer*)? Koji od ovih sustava su skalabilni i zašto?

11.3. Na slici je prikazana mreža Chord koja se sastoji od 6 čvorova (A, B, C, D, E i F) i koristi prostor identifikatora duljine N=32 (dovoljno je m=5 bita za kodiranje). Ukoliko je  $H_1(A)=0$ ,  $H_1(B)=5$ ,  $H_1(C)=10$ ,  $H_1(D)=14$ ,  $H_1(E)=25$  i  $H_1(F)=27$ , odgovorite na sljedeća pitanja:

- Definirajte tablice usmjeravanja na čvorovima A i F.
- Na kojem će se čvoru pohraniti podatak X s ključem  $H_2(X)=20$ ?
- Odredite slijed čvorova preko kojih se usmjerava upit od čvora A s ciljem pronađaska podatka Y s ključem  $H_2(Y)=26$ .
- Dodan je novi čvor G ( $H_1(G)=21$ ) u mrežu. Što će se promjeniti u tablici usmjeravanja čvora A?

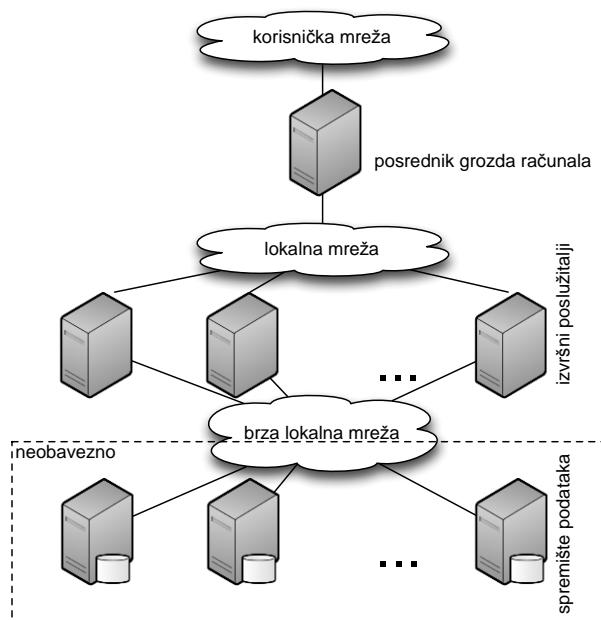


11.4. Komentirajte rezultate eksperimenta kojim se ispituje svojstvo skalabilnosti protokola Gnutella i Chord u statičnom scenariju. Kako objašnjavate krivulje kojima se prikazuje prosječan broj čvorova po upitu i prosječno vrijeme odziva?

## 12 GROZD, SPLET RAČUNALA I RAČUNALNI OBLAK

### 12.1 Grozd računala

Grozd (engl. *cluster*) računala je sustav slabo povezanih računala koji rade zajedno tako da ga se može gledati kao jedno veliko računalo. Računala u grozdu su povezana brzom lokalnom mrežom, obično optičkom, kako bi brzo mogli razmijeniti podatke. Svako računalo ima pokrenutu svoju instancu operacijskog sustava. Primjer arhitekture jednog grozda prikazano je na slici (Slika 12.1). Grozdovi računala nastali su razvojem tehnologija jeftinih mikroprocesora, brzih mreža i softvera za visoko dostupne raspodijeljene sustave.



Slika 12.1. Arhitektura grozda računala

Za takav sustav potrebno je programirati aplikacije koje koriste paralelnost i raspodijeljenost kako bi se što bolje iskoristili resursi tog sustava.

Programska potpora grozda računala se sastoji od:

- operacijskih sustava – najčešće se koriste raznovrsni komercijalni ili javno dostupni operacijski sustavi. Ponekad se funkcionalnosti postojećih operacijskih sustava prilagođavaju specifičnim potrebama grozdova računala;
- posredničkog sustava – ostvaruje skup funkcionalnosti na temelju kojih se objedinjuje raspodijeljeno sklopolje sustava u usklađenu cjelinu. Posrednički sustav je obično centraliziran. Osnovu sustava čine podsustavi:
  - za upravljanje poslovima,
  - za nadgledanje rada,
  - razvojno okružje i
  - za dijeljenje podataka.

**Podsustav za upravljanje poslovima** omogućuje korisnicima postavljanje i izvođenje poslova koje ostvaruju aplikaciju. Omogućuje postavljanje parametara izvršavanja aplikacija kao što su prioritet izvođenja i količina sredstava pridruženih aplikaciji, upravljanje procesima i aplikacijama, korisničke

postavke za upravljanje sredstvima itd. Primjeri podsustava su: Sun Grid Engine (SGE), Portable Batch System (PBS) i Condor.

**Podsustav za nadgledanje rada** omogućuje korisnicima uvid u stanje sredstava. Postoje raznovrsni podsustavi za ostvarivanje upravljanja poslovima i nadgledanje, od kojih su dva najpoznatija Ganglia i Supermon.

**Razvojno okružje** omogućuje izgradnju logike procesa koji ostvaruju funkcionalnosti aplikacija i logike suradnje procesa koji čine paralelnu i raspodijeljenu aplikaciju. Procesi najčešće ostvaruju komunikaciju razmjenom poruka. Logika procesa i logika suradnje procesa ostvaruje se primjenom različitih programskih knjižnica i programskih jezika, od kojih su najpoznatiji: Message Passing Interface (MPI), Linda, High Performace Fortran (HPF) i Z-Level Programming Language (ZPL).

**Podsustav za dijeljenje podataka** omogućuje procesima spremanje i dohvaćanje velike količine podataka koje procesi koriste tijekom izvođenja aplikacije. Za to je potrebna brza lokalna mreža i programska podrška kao što su: Network File System (NFS), AFS i Lustre.

### 12.1.1 Svojstva

Grozd može biti napravljen za različite svrhe, od sustava opće namjene npr. za podršku web-uslugama, do sustava za znanstveno računanje. Za različite primjene bitna su različita svojstva u realizaciji grozda. Tri osnovna svojstva koja se promatraju su:

- uravnoteženje opterećenja,
- računsko intenzivna obrada i
- visoka dostupnost.

Uravnoteženje opterećenja u grozdu se postavlja tako da čvorovi u grozdu dijele opterećenje računanja s ciljem što boljih performansi sustava. Na primjer, upite koji dolaze na posrednika, on raspoređuje tako da je ukupno vrijeme odgovora na upit što kraće. Pristup uravnoteženju opterećenja može se značajno razlikovati od aplikacije do aplikacije.

Računsko intenzivna obrada je u potpunosti drugačija od obrade koja je orijentirana dohvaćanju podataka iz spremišta podataka (datoteke ili baza podataka) i isporuci klijentu. Primjer takvog korištenja je simulacija vremena (misli se na prognozu) ili analiza podataka koji su prikupljeni iz nekog stroja (npr. analiza podataka iz automobila u Formuli 1).

Visoka dostupnost (engl. *high availability*) u grozdovima omogućuje da grozd i dalje radi iako neki dijelovi grozda nisu u funkciji zbog kvara. To se rješava redundantnim čvorovima koji preuzimaju obradu kada pojedini čvor nije u funkciji. U takvim grozdovima ne postoji jedna točka ispada tj. ne postoji niti jedno mjesto u grozdu koje u slučaju kvara zaustavlja rad grozda.

### 12.1.2 Primjeri grozdova

Jedan od najpoznatijih primjera grozda je Beowulf<sup>54</sup>. Ovaj sustav omogućava povezivanje do 1024 radnih računala primjenom standardnih mrežnih tehnologija te uporabom operacijskog sustava Linux. Najjednostavniji sustav omogućuje najmanje 2 računala i brzu mrežu.

U Hrvatskoj također postoji nekoliko grozdova računala. Jedan od najpoznatijih je grozd Isabella<sup>55</sup> koji poslužuje Sveučilišni računski centar u Zagrebu (SRCE)

<sup>54</sup> [www.beowulf.org](http://www.beowulf.org) (29.12.2011.)

<sup>55</sup> [www.srce.unizg.hr/isabella](http://www.srce.unizg.hr/isabella)

## 12.2 Splet računala

Splet računala (engl. *grid*) je raspodijeljeno računalno okružje koje uključuje federaciju računalnih sredstava koja su pod različitim administrativnim domenama (Foster & Kesselman, 2003). Za razliku od grozda, splet je slabije povezan, heterogen i geografski udaljen. Obično se splet koristi za određene vrste aplikacija, ali bez obzira na primjenu splet koristi međusloj koji je općenit i nije specijaliziran za neku vrstu aplikacija.

Pojam „Grid“ je nastao sredinom 1990-ih godina koji je tada predstavljao raspodijeljenu računalnu infrastrukturu za naprednu znanost i inženjerstvo. Osnovna ideja je bila dijeljenje računalnih sredstava pomoću Gigabitne mreže. Pod pojmom računalna sredstva se misli na podatke, programsku podršku, resurse računanja pa i specijalizirane resurse kao što su teleskopi ili mikroskopi. Posljedica je razvoj različitih vrsta aplikacija koja uključuju raspodijeljeno računanje za analizu podataka, federacija različitih raspodijeljenih podataka, suradna vizualizacija velikih količina znanstvenih podataka itd.

Splet se koristi na različite načine, ali svi ti načini imaju skup zajedničkih zahtjeva za koordinacijom dijeljenja sredstava i rješavanja problema u dinamičkoj i višeinsticionalnoj suradnji. Ovdje je uveden pojam virtualne organizacije te se primjenjuje na ovaku suradnju koja je raspodijeljena i privremena. Isti ovakvi zahtjevi se javljaju i u gospodarskoj suradnji npr. u integraciji poslovnih aplikacija, pružanju usluga na zahtjev, federacija podatkovnih centara i suradnja B2B (engl. *business-to-business*) preko Interneta.

Uspjeh spletova je u tome što se relativno brzo došlo do arhitekture sustava i zapravo programske podrške koja je danas *de facto* standardna programska podrška (Globus Toolkit<sup>56</sup>). Ona je rezultat agresivnih prvih korisnika koji su imali izazovne probleme koje je trebalo riješiti i žive međunarodne zajednice razvijatelja i korisnika spletova. Ta kombinacija čimbenika dovila je do iskustva koje je rezultiralo definiranja arhitekture OGSA<sup>57</sup> (engl. *Open Grid Services Architecture*) na kojoj su temeljeni današnji komercijalni spletovi i spletovi temeljeni na otvorenom kodu (Nabrzyski, Schopf, & Weglarz, 2004).

### 12.2.1 Arhitektura

Arhitektura spleta računala koja prikazuje osnovne komponente sustava, specificira funkcije komponenti i interakciju između njih, a prikazana je u slojevitom prikazu slikom (Slika 12.2). Za svaki sloj definirane su funkcije koje se realiziraju korištenjem nižih slojeva.



Slika 12.2. Slojevita arhitektura spleta računala

<sup>56</sup> Globus Toolkit Homepage, <http://www.globus.org/toolkit/> (29.12.2011.)

<sup>57</sup> The Open Grid Services Architecture, Version 1.5, [http://www.ogf.org/documents/GFD\\_80.pdf](http://www.ogf.org/documents/GFD_80.pdf) (29.12.2011.)

**Sloj osnovnih sredstava** upravlja osnovnim funkcionalnostima sleteta računala. On implementira lokalne operacije koje su specifične svakom sredstvu po vrsti i implementaciji te omogućuje korištenje tih operacija na višim slojevima. Vrste sredstava kojima posreduje su:

- računalna sredstva (engl. *computational resources*) – mehanizmi za pokretanje programa, nadgledanje i upravljanje procesima,
- spremnički prostor (engl. *storage systems*) – mehanizmi za stavljanje i dohvaćanje datoteka ili dijelova datoteka, mehanizmi upravljanja sredstvima za prijenos podataka (prostor na tvrdom disku, mrežna propusnost, procesorsko vrijeme),
- katalozi sredstava (engl. *resource catalogues*) – pronalaženje i registriranje sredstava,
- mrežna sredstva (engl. *network resources*) – upravljački mehanizmi kontrole dodijeljenih mrežnih sredstava (prioriteti i rezervacije) te mehanizmi analize potrošnje mrežnog kapaciteta,
- osjetila i aktuatori (engl. *sensors and actuators*) – osjetila su izvori podataka koje treba spremiti u neki spremnički prostor kada se generiraju, a mogu generirati velike količine podataka u vrlo kratkom vremenu dok se pomoću aktuatora može upravljati eksperimentima npr. parametri rada motora
- itd.

Sloj osnovnih sredstava obavezno posreduje računalnim sredstvima, spremničkim prostorom i mrežnim sredstvima, a ostale vrste sredstava su optionalne tj. ovise o konkretnom spletu.

**Sloj komunikacijskih protokola** definira protokole komunikacije i autentikacije koji su potrebni za transakcije u spletu.

Komunikacijski protokoli omogućuju razmjenu podataka između resursa na sloju osnovnih sredstava. Komunikacijski dio zadužen je za transport, usmjeravanje i imenovanje temeljeno na protokolima složaja TCP/IP.

Autentikacijski protokoli nadovezuju se na komunikacijske tako da pružaju sigurnosne mehanizme identifikacije i autorizacije korisnika i resursa. Autentikacijski protokoli temeljeni su na kriptografskim algoritmima. Bitne funkcije kojima se mogu realizirati virtualne organizacije su:

- prijava na jednom mjestu (engl. *single sign-on*),
- delegacija poslova – korisnik može pokrenuti neki program sa svojim pravima,
- integracija s lokalnim sigurnosnim mehanizmima u heterogenoj okolini i
- mehanizmi povjerenja temeljeni na korisnicima.

**Sloj protokola za sredstva** omogućuje korisniku interakciju s udaljenim resursima i uslugama. On definira protokole za sigurno pregovaranje, pokretanje, praćenje, kontrolu, obračun (engl. *accounting*) i naplatu dijeljenih operacija i individualnih resursa. Postoje dvije osnovne vrste protokola u ovom sloju:

- informacijski protokoli – dohvaćanje, pregledavanje i analiza stanja (konfiguracija, zauzeće sredstava, broj korisnika itd.) i
- upravljački protokoli – pregovaranje o pristupu i upravljanje postavkama za dijeljenje i korištenje (prava pristupa, dostupnost, udruživanje, usmjeravanje itd.).

**Sloj zajedničkih usluga** definira protokole i usluge koji su zaduženi za upravljanje grupom sredstava, a ne za pojedino sredstvo. U ovom sloju se mogu ugraditi mehanizmi za vrlo različite načine dijeljenja sredstava:

- imeničke usluge – za pronalaženje sredstava prema svojstvima (npr. vrsta, dostupnost, opterećenje, ...),
- usluge za raspoređivanje zahtjeva za pristup sredstvima – za rezervaciju, vremensko planiranje (engl. *scheduling*) i posredovanje (engl. *brokering*) sredstvima,
- usluge za nadgledanje rada skupine sredstava – za otkrivanje kvarova, napada, preopterećenja, ...,
- okružja za razvoj logike suradnje sredstava – programski modeli, upravljanje tokom, otkrivanje usluga, suradne usluge i
- usluge za naplatu korištenja sredstava – autorizacija poslužitelja, obračunavanje, naplata, ograničavanje sredstava.

### 12.2.2 Primjena

Spletovi računala mogu biti različitih vrsta ovisno o tome za što su primarno namijenjeni. Osnovne vrste spletova su:

- spletovi za složene proračune (engl. *computational grids*) – za modeliranje i simuliranje složenih znanstvenih eksperimenata,
- podatkovni spletovi računala (engl. *data grids*) – spremanje i obrada velike količine podataka,
- poslovni spletovi računala (engl. *business grids*) – dubinska analiza velike količine poslovnih podataka i
- bežični spletovi računala (engl. *wireless grids*) – potpora bežičnim korisničkim uređajima.

### 12.2.3 Primjeri spletova

TeraGrid<sup>58</sup> je bio jedan od najvećih spletova računala u SAD-u. Projekt je započeo 2001., a od 2004. do 2011. se koristio za istraživanja. Znanstvenici (više od njih 10.000) s više od 200 sveučilišta su ga koristili za izvođenje složenih znanstvenih aplikacija iz područja: molekularne biologije, fizike, astronomije, kemije, istraživanja materijala, ... Nakon završetka projekta 2011. napravljena je tranzicija na projekt XSEDE (engl. *Extreme Science and Engineering Discovery Environment*)<sup>59</sup>. U projektu XSEDE se koriste resursi koji imaju više od 100.000 procesora i u 24 sata generira više od 100 TB podataka. Pogledajmo jedan veći resurs pod nazivom Kraken-XT5 koji ima sljedeća svojstva: temelji se na super računalu Cray XT5, ima 9408 čvorova od kojih svaki ima 12 procesora što ukupno daje 112.896 procesora, 1,33 GB memorije po procesoru, veličina diska 2,4 PB, najveća brzina 1,174 petaflopsa (engl. *Floating-point OPeration*). Svaki čvor na sebi ima operacijski sustav Compute Node Linux (CNL) 2.2.

Drugi sustav je EGEE (engl. *Enabling Grids for E-sciencE*)<sup>60</sup> koji je splet računala u EU. Projekt je trajao od 2002. do 2010. Ovaj splet računala gradio se s ciljem obrade podataka dobivenih radom sustava LHC (*Large Hadron Colider*), obrade biomedicinskih podataka te modeliranje i simuliranje različitih prirodnih pojava kao što su potresi. Njegova sredstva u zadnjoj fazi (EGEE-III) bila su: 72.000 procesnih jedinica, 20 PB diskovnog prostora, 10.000 registriranih korisnika. Nastavak je projekt EGI (engl. *European Grid Infrastructure*) kojem je cilj povezati nacionalne europske spletove u jedan veliki splet. U njemu sudjeluje i hrvatski splet CGI NGI<sup>61</sup>.

<sup>58</sup> TeraGrid, <https://www.xsede.org/tg-archives> (29.12.2011.)

<sup>59</sup> XSEDE, <https://www.xsede.org/> (29.12.2011.)

<sup>60</sup> Enabling Grids for E-sciencE, <http://www.eu-egee.org> (29.12.2011.)

<sup>61</sup> Hrvatska nacionalna grid infrastruktura (CRO NGI), [www.srce.unizg.hr/cro-ngi](http://www.srce.unizg.hr/cro-ngi)

U Hrvatskoj je 20014. pokrenuta inicijativa Cro-GRID koja je ostvarena u suradnji nekoliko institucija i organizacija (Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, Ericsson Nikola Tesla, SRCE, Sveučilište J. J. Strossmayera Elektrotehnički fakultet, Sveučilište u Splitu Fakultet elektrotehnike, strojarstva i brodogradnje, Sveučilište u Rijeci Građevinski fakultet i Tehnički fakultet). Projekt CRO-GRID koji je trajao do 2006. uključivao je razvoj aplikacija, infrastrukture i posredničkog sustava spleta računala. Ovaj splet je bio prvenstveno namijenjen za dubinsku analizu podataka, analizu proteina i pametni transport.

### 12.3 Računalni oblak

Postoje različite definicije računarstva u oblaku (engl. *Cloud Computing*) i svaka od njih naglašava jedan aspekt. Osnovna ideja računarstva u oblaku je da se računalni resurs može iznajmiti po potrebi. Pod računalnim resursima misli se na različite stvari kao što su: čitava računala, usluge, aplikacije itd. Korisnik plaća samo ono što je koristio. U trenutku kada trebamo više računalnih resursa zbog pojave vršnog opterećenja, unajmimo ih bez potrebe za interakcijom s osobljem i bez dodatnih ugovora, a otpustimo ih kada potrebe za njima nestanu te nas ne brine hoće li skupo plaćena oprema biti iskorištena kao kada sami izgrađujemo sustav. Na taj način se ulaganje u kapitalnu infrastrukturu opremu tj. poslužitelje, mrežne i struine kapacitete, posebne poslužiteljske prostorije, pretvara u operativne troškove jer se plaća prema korištenju (Sosinsky, 2011) (Reese, 2009).

Ako se neka organizacija i tvrtka odluči napraviti svoj podatkovni centar (engl. *Data Centre*) potrebno je imati stručne inženjere koji znaju posložiti (instaliranje i konfiguriranje) i kasnije održavati takav centar. Osim toga je potrebno osigurati: mrežni pristup dovoljnog kapaciteta, opskrbu električne energije (npr. 20.000 računala troši 500KW električne energije), hlađenje prostora u kojem su računala, fizičku sigurnost, barem dvije lokacije zbog pouzdanosti, ... Cijena jednog velikog podatkovnog centra košta od \$300.000.000 do \$2.000.000.000<sup>62</sup>, a prostor na kojem su napravljeni se kreće od 30.000 m<sup>2</sup> do 300.000 m<sup>2</sup>. Velike tvrtke kao što su Google, Microsoft, Amazon imaju od 5 do 30 takvih centara diljem svijeta. Poslužitelji u tim centrima se koriste od 3 do 5 godina, a njihova isplativost se postiže nakon 18 mjeseci prvenstveno zbog potrošnje energije. Za takav centar se biraju lokacije koje imaju pristup brzom Internetu (blizu magistralnih veza), jeftinu električnu energiju i mogućnost korištenja ekoloških izvora energije (vjetroelektrane, solarni paneli, ...).

U računalnom oblaku postoje tri strane:

- pružatelji infrastrukture – grade i održavaju računalnu infrastrukturu za učinkovito izvođenje aplikacija dostupnih udaljenim korisnicima. Infrastruktura se naziva računalni oblak (*Cloud Computing System*);
- pružatelji aplikacija/usluga – grade/postavljaju/održavaju aplikacije/usluge koje je moguće izvoditi u računalnom oblaku. Oni su ujedno i korisnici infrastrukture;
- krajnji korisnici – pristupaju oblaku s bilo koje lokacije na Zemlji s koje imaju pristup Internetu i koriste aplikacije dostupne u oblaku.

NIST (*U.S. National Institute of Standards and Technology*) je u svojim dokumentima napravio podjelu oblaka prema tome gdje je smješten i prema svrsi na:

- javni oblak (*public*) – iznajmljuje se za javnu uporabu, a u vlasništvu je organizacije koja prodaje usluge u oblaku;
- privatni oblak (*private*) – u privatnom vlasništvu poduzeća i samo to poduzeće ga koristi. Često se i ne smatra pravim „oblakom“;

---

<sup>62</sup> Podaci preuzeti iz: The Economics of the Cloud, Microsoft, November 2010, <http://www.microsoft.com/presspass/presskits/cloud/docs/The-Economics-of-the-Cloud.pdf>

- zajednički oblak (*community*) – nekoliko organizacija dijeli jednu infrastrukturu;
- hibridni oblak (*hybrid*) - kompozicija 2 ili više oblaka različitih vrsta.

Dva najvažnija koncepta koja su potaknula razvoj oblaka su:

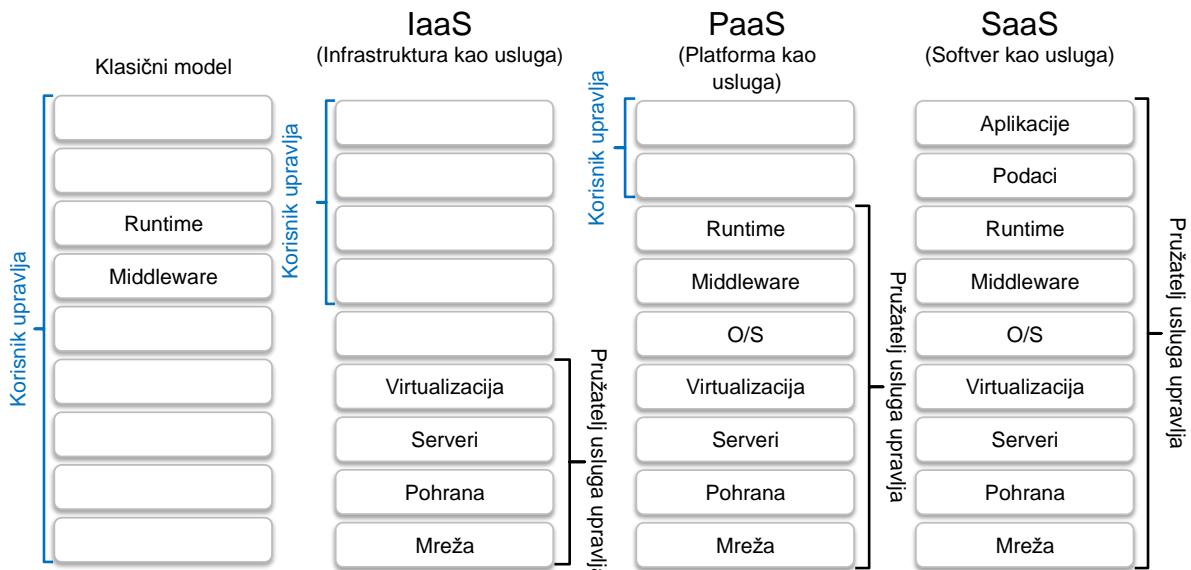
- apstrakcija implementacije – u oblaku korisnik i razvijatelj ne znaju specifikaciju sustava na kojem će se aplikacije i usluge izvoditi. Podaci će biti spremjeni na lokacije koje nisu poznate, a administracija sustava nije pod kontrolom razvijatelja. Pristup aplikacijama i uslugama je omogućen s bilo kojeg mesta koje ima pristup Internetu;
- virtualizacija – omogućuje izvršavanje više operacijskih sustava na jednom fizičkom ili na više fizičkih računala. Isto to vrijedi i za pohranu podataka. Po potrebi se resursi mogu dijeliti ili udruživati. Fizička računala mogu imati podršku za virtualizaciju (npr. IBM S/370, Intel VT ili AMD-V) i na taj način smanjiti degradaciju performansi zbog dijeljenja fizičkih resursa. Primjeri programske podrške koje to omogućuju: Xen, VMware, KVM.

### 12.3.1 Modeli usluga

U oblaku postoji više modela pružanja usluga koje se mogu isporučiti klijentima. Postoji kratica XaaS (engl. *X as a Service*) tj. X kao usluga. Tri osnovne usluge koje se mogu isporučiti klijentima su (Slika 12.3):

- IaaS – infrastruktura kao usluga (engl. *Infrastructure as a Service*) – klijentu infrastrukture isporučuje virtualno računalo, spremište podataka ili neku drugu infrastrukturu. Isporučitelj upravlja infrastrukturom, a klijent infrastrukture je odgovoran za druge aspekte isporuke (npr. operacijski sustav i aplikacije) krajnjem klijentu. Primjeri: Amazon Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3), Eucalyptus, GoGrid, ...;
- PaaS – platforma kao usluga (engl. *Platform as a Service*) – klijentu isporučuje virtualno računalo s operacijskim sustavom na kojem su različita programska sučelja koja se mogu koristiti za razvoj aplikacija. Klijent platforme razvija aplikaciju u programskoj jeziku koji je podržan i isporuči je u oblak te ju onda krajnji korisnici mogu koristiti. Isporučitelj je zadužen za upravljanje i održavanje platforme (infrastruktura, operacijski sustav, programska sučelja), a klijent platforme je zadužen za instaliranje, upravljanje i isporuku aplikacije. Primjeri: Force.com, GoGrid Cloud Center, Google AppEngine, Windows Azure Platform, ...;
- SaaS – softver kako usluga (engl. *Software as a Service*) – klijentu isporučuje kompletну okolinu s operacijskim sustavom i aplikacijama te korisničkim sučeljem. Isporuka aplikacije klijentu je obično preko tankog klijenta, najčešće u pregledniku (engl. *browser*). Klijent je zadužen za svoje podatke i korisničku interakciju. Primjeri: SalesForce CRM, GMail, Google Docs, DeskAway, Impel CRM, Oracle On Demand, SQL Azure, ...

Ova tri modela se još nazivaju i modeli SPI. Postoje još neki modeli usluga kao što su: StaaS – spremište ako usluga (engl. *Storage as a Service*), IdaaS – identitet kao usluga (engl. *Identity as a Service*) itd. Ti drugi modeli nisu toliko rašireni i rjeđe se koriste.



Slika 12.3. Modeli usluga u oblaku (izvor: Microsoft)

### 12.3.2 Prednosti

Osnovne prednosti računarstva u oblaku su:

- samoposluživanje na zahtjev – klijent oblaka može bez interakcije s osobljem isporučitelja sam iznajmiti infrastrukturu, pokrenuti korištenje usluge ili aplikacije i sl.;
- usluge dostupne preko web-preglednika – većina usluga ima sučelja na webu koja su dostupna s bilo kojeg mesta s pristupom Internetu;
- plaćanje prema korištenju – klijentu se naplaćuje samo korištenje usluga. Ako netko nije koristio napredne mogućnosti neke aplikacije jer mu nisu potrebne onda se one neće naplatiti;
- nadogradnja resursa na zahtjev – ako klijent infrastrukture ili platforme treba više poslužitelja na neko vrijeme to je moguće napraviti na zahtjev preko weba. Nakon korištenja se resursi mogu oslobođiti;
- elastičnost resursa – automatsko proširivanje mrežnih i računalnih kapaciteta ovisno o opterećenju. Npr. ako web-aplikacija koju smo postavili u oblak zahtjeva više računalnog kapaciteta u nekom periodu (npr. mrežni kapacitet ili više procesora prije Božića i Nove godine) sustav će to detektirati i povećati kapacitete u tom periodu ili smanjiti kapacitete u nekom drugom periodu;
- ne zahtijevaju se kapitalni troškovi za izgradnju infrastrukture – nije potrebno kupovati, instalirati i održavati poslužitelje. Ne zahtjeva se niti vrijeme niti inženjersko znanje za postavljanje sustava;
- povećana pouzdanost – u oblaku su ugrađeni sustavi koji omogućuju uravnoteženje opterećenja, visoku dostupnost, a svaki oblak ima više fizički raspodijeljenih podatkovnih centara koji su povezani brzom mrežom. Sustavi u oblaku obično imaju najkvalitetnije sustave koji omogućuju bolju pouzdanost sustava;
- jednostavnije upravljanje održavanjem i nadogradnjom sustava.

### 12.3.3 Nedostaci

Uz prednosti koje nudi računarstvo u oblaku postoje i nedostaci koje također treba uzeti u obzir prilikom odluke o prelasku na oblak. Nedostaci su:

- ne postoji mogućnost prilagodbe klijentu kao kada klijent upravlja sam sa svojim fizičkim računalima;
- aplikacije koje su izvan oblaka obično imaju veće mogućnosti nego aplikacije u oblaku;
- ako aplikacija zahtjeva prijenos velikih količina podataka oblak nije najbolje rješenje;
- oblak je u principu sustav bez stanja kao i kod protokola HTTP što ponekad može biti nedostatak;
- ovisnost o jednom pružatelju usluga u oblaku jer ne postoje standardi;
- usvajanje novog načina razvoja aplikacija;
- privatnost i sigurnost podataka je najveći nedostatak jer klijent ne zna gdje su podaci i na koji način su pohranjeni. Postoji i problem lokacije podatkovnih centara jer se lokalni zakoni o sigurnosti podataka razlikuju. Npr. utjecaj politike na filtriranje podataka događao se kod Wikileaks u SAD-u u jednom slučaju, a u drugom slučaju je Google imao centar podataka u Kini pa je pristup njima bio filtriran.

## 12.4 Mehanizmi

Za izradu grozdova, spletova i računalnog oblaka potrebni su različiti mehanizmi. U nastavku slijede neki od bitnih mehanizama koji se koriste u takvim sustavima.

### 12.4.1 Prijenos podataka

**Transportni sloj** je osnovni sloj koji omogućava uspostavu komunikacije između elemenata spleta računala ili računalnog oblaka. On omogućuje zapisivanje i prijenos podataka. Najčešći protokoli koji se koriste su FTP (File Transfer Protocol) i GridFTP (GRID File Transfer Protocol). Za razliku od FTP-a koji je standardni protokol u Internetu, GridFTP je njegovo proširenje koji ima ugrađenu: sigurnost, upravljanje prijenosom od treće strane, istodobni prijenos, djelomični prijenos, otpornost na pogreške i optimizacija postavki prijenosa.

**Sloj prividne mreže** omogućuje upravljanje komunikacijskim sredstvima u spletu računala. On ostvaruje programsku apstrakciju fizičke komunikacijske infrastrukture. Prividna mreža uspostavlja logičku organizaciju komunikacijske mreže te ostvaruje preslikavanje čvorova prividne mreže u fizičke čvorove u raspodijeljenoj okolini. Upravljački sloj prividne mreže omogućuje prilagodbu komunikacijske infrastrukture potrebama primjenskog sustava tako da osigurava:

- imenovanje sredstava i razlučivanje adresa sredstava,
- upravljanje prijavama i odjavama korisnika,
- pouzdanost i dostupnost sadržaja i
- usmjeravanje sadržaja.

Prividne mreže se upotrebljavaju u:

- sustavima spletala računala,
- sustavima ravnopravnih sudionika (P2P) i
- sustavima mreža osjetila i aktuatora.

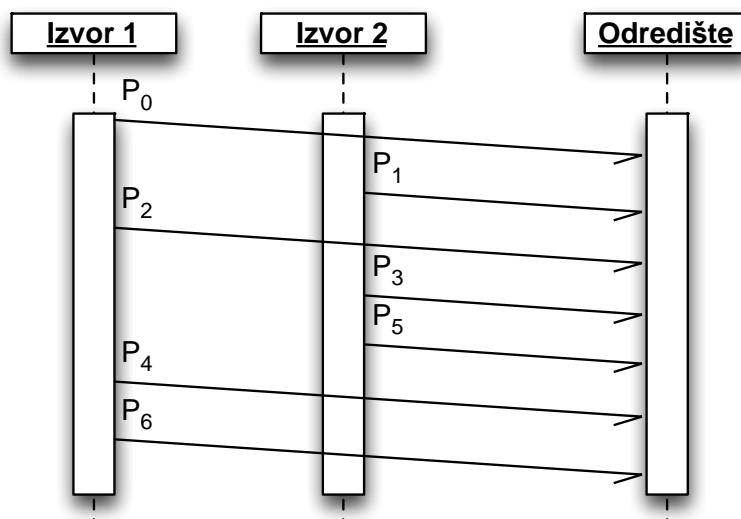
**Otpornost na pogreške** važan je čimbenik u spletu računala i u računalnom oblaku. U spletu računala se razmjenjuje velika količina podataka pa je potrebno ostvariti učinkovite postupke oporavka od pogreške koji ne uvode dodatno opterećenje na sustav, ali pravovremeno otkrivaju i uklanjaju nastale pogreške. Osnovne metode uspostave otpornosti na pogreške su:

- potpuna retransmisija cijelog sadržaja,
- nastavljanje transmisije od mesta gdje se detektira pogreška i
- uporaba međuspremnika pa se u slučaju pogreške radi retransmisija samo onih podataka kod kojih je došlo do pogreške.

**Model prijenosa podataka** kada je potrebno osigurati visoki stupanj propusnosti uz mala vremena odziva moraju koristiti specijalizirane modele prijenosa kao što su:

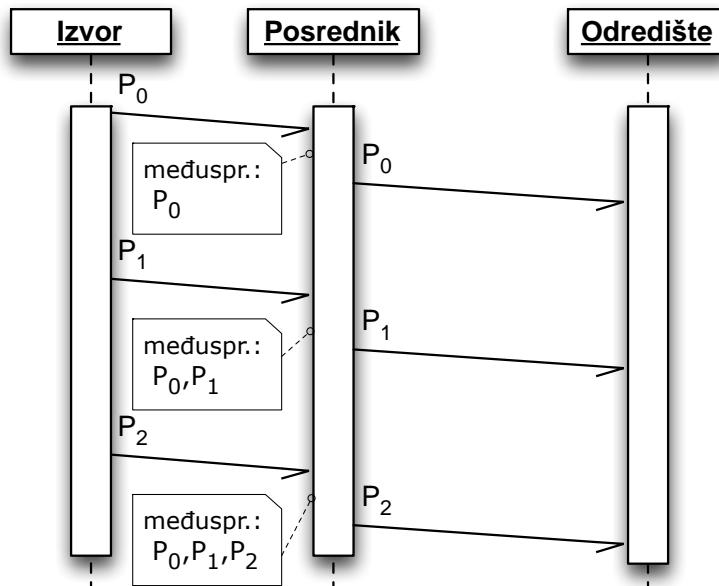
- istodobni prijenos različitih blokova podataka,
- automatska prilagodba veličine međuspremnika i
- pakiranje podataka.

**Prijenos različitih blokova istog skupa podataka** prikazan je slikom (Slika 12.4). Ovdje imamo dva izvora podataka od kojih svaki izvor isporučuje samo neke podatke koji se na odredištu slažu u jednu cjelinu. Blokovi podataka 1, 3 i 5 šalje Izvor 1, dok blokove 0, 2, 4 i 6 šalje Izvor 2. Ovakav model omogućuje ubrzanje ukupnog prijenosa skupa podataka. Različiti blokovi skupa podatka mogu se istodobno prenositi s različitih mesta u spletu računala.



Slika 12.4. Prijenos različitih blokova iz različitih izvora

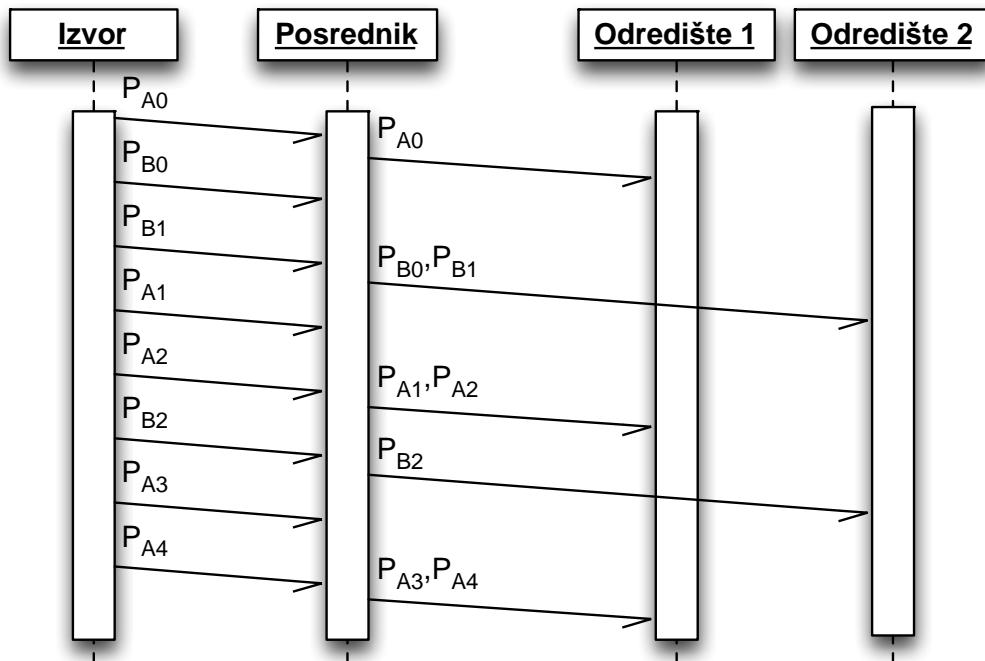
**Automatska prilagodba veličine međuspremnika** prikazana je slikom (Slika 12.5). Izvor podataka šalje podatke posredniku na kojem se nalazi međuspremnik. Kako izvor šalje podatke tako međuspremnik spremi podatke, a u slučaju da još netko treba te podatke izvor ih ne treba ponovno slati, već ih međuspremnik prosljedi odredištu. Primjena ove metode omogućuje učinkovito upravljanje međuspremničkim prostorom i rasterećenje izvora. Zapisi koji su stavljeni u međuspremnik u prošlosti, a rijetko se koriste mogu biti izbrisani iz međuspremnika kako bi se oslobođio prostor za nove zapise. Mogu se koristiti različite discipline posluživanja kao što su FIFO (First-In-First-Out), LIFO (Last-In-First-Out) i LRU (Least-Recently-Used).



Slika 12.5. Automatska prilagodba veličine međuspremnika

**Pakiranje podataka** je prikazano slikom (Slika 12.6). U ovom primjeru imamo dva odredišta. Jednom odredištu se šalju podaci iz skupine A, a drugom iz skupine B. Izvor šalje podatke posredniku koji ne šalje odmah podatke odredištu, nego ih sprema tako da podatke iz iste skupine može odredištu poslati u jednoj poruci tj. pakira ih i takve pakirane podatke šalje odredištu.

Primjena postupaka pakiranja podataka omogućava sažimanje i pakiranje više blokova podatka u jednu poruku. Na taj način moguće je ostvariti učinkovitiji prijenos podataka između izvořišta i odredišta.



Slika 12.6. Pakiranje podataka

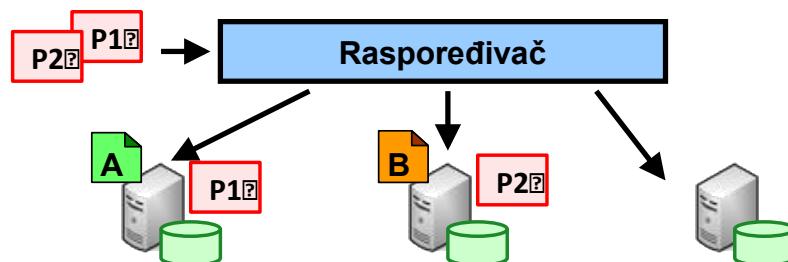
#### 12.4.2 Raspoređivanje zahtjeva

Model sustava za raspoređivanje zahtjeva uključuje korisnika, raspoređivača i skup poslužitelja. Korisnik upućuje zahtjeve raspoređivaču koji primjenom neke discipline raspoređivanja upućuje

zahtjeve na odgovarajuće poslužitelje. Raspoređivanje u složenim sustavima se može realizirati tako da se uzmu u obzir različita svojstva:

- lokalnost podataka (vrijeme i prostor),
- doseg raspoređivača,
- funkcija cilja,
- razine raspoređivanja i
- smještaj raspoređivača.

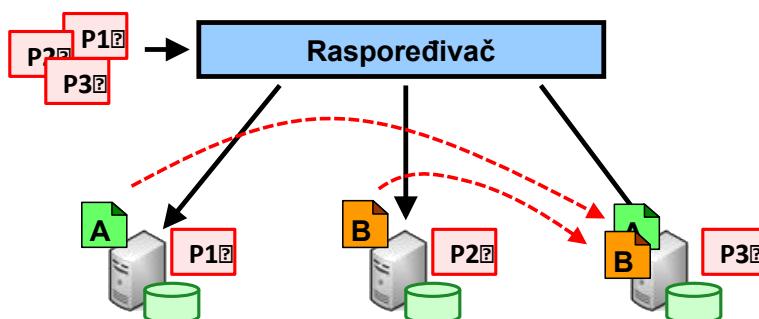
**Lokalnost** je svojstvo koje uzima lokalne parametre ovisno o vremenskoj i prostornoj dostupnosti podataka. Kod prostorne lokalnosti se zahtjevi raspoređuju na čvorove koji sadrže podatke potrebne za obradu zahtjeva. U tom slučaju se poslovi obrade zahtjeva približavaju podacima. Vremenska lokalnost je slučaj kada se poslovi postavljaju na čvorove gdje se izvode poslovi koji koriste dobivene podatke tj. podaci se približavaju poslovima.



Slika 12.7. Primjer prostorne lokalnosti

U primjeru prostorne lokalnosti (Slika 12.7) posao P1, koji koristi podatak A, raspoređuje se na čvor na kojem se nalazi podatak A, dok se posao P2, koji koristi podatak B, raspoređuje se na računalo na kojem se nalazi podataka B.

U primjeru vremenske lokalnosti (Slika 12.8) se za prvi dio raspoređivanja koristi raspoređivanje na temelju prostorne lokalnosti. Zatim se rezultati obrade podatka s oba računala sele se na računalo na koje je raspoređen posao P3 koji koristi rezultate korištenja prethodna dva posla. Na opisani način ostvaruje se vremenska lokalnost.



Slika 12.8. Primjer vremenske lokalnosti

**Doseg raspoređivača** definira količinu informacija koju raspoređivač koristi za donošenje odluke o načinu raspoređivanja. Tako imamo doseg usmjeren jednom korisniku ili na skupine korisnika. Kada je doseg usmjeren jednom korisniku onda svaki korisnik ima svoj raspoređivač koji raspoređuje poslove na najmanje opterećene čvorove. Kada je doseg usmjeren na skupinu korisnika onda jedna skupina ima jedan raspoređivač.

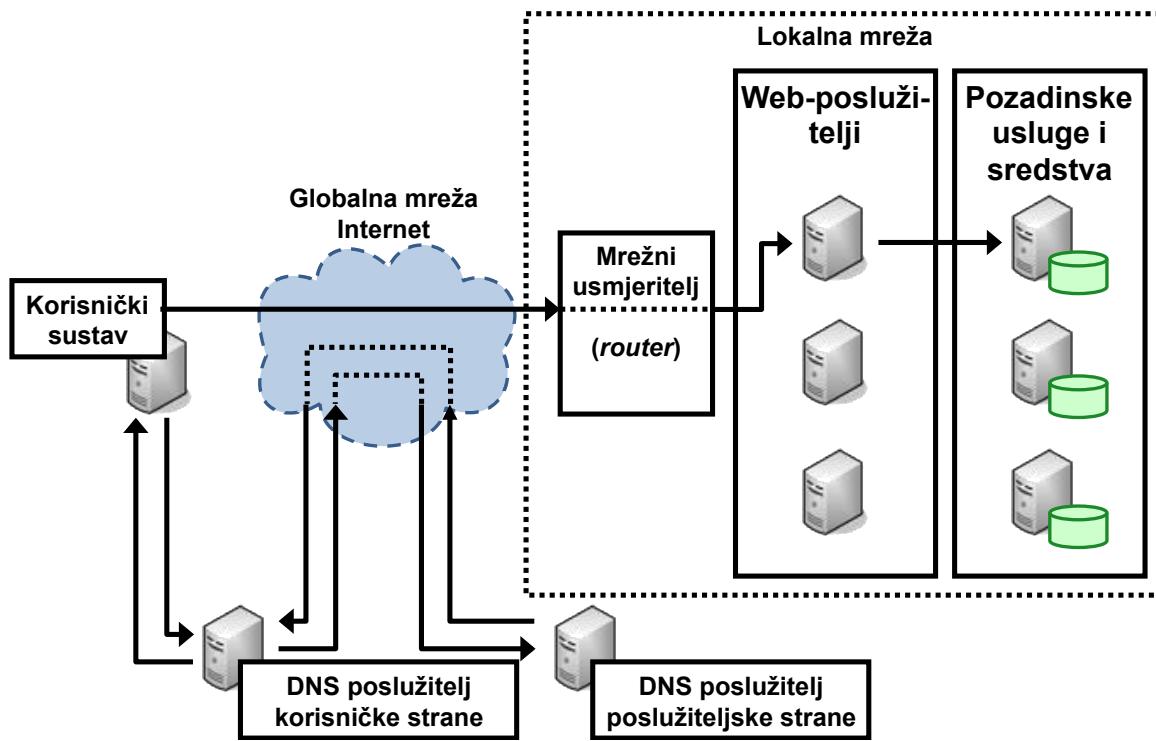
Funkcija cilja određuje kriterij mjerjenja učinkovitosti postupka raspoređivanja. Cilj može biti temeljen na ravnomjernom opterećenju sredstava tako da se ostvari maksimalna iskoristivost sredstava. Druga

vrsta cilja može biti zasnovana na ekonomskim strategijama koje raspoređivanje usklađuju s korisničkim mogućnostima, cijeni izvođenja poslova i korištenja sredstava sustava. Korisnici u tom slučaju definiraju željeni stupanj kvalitete usluge: ukupna cijena izvođenja, maksimalni vremenski rok izvođenja, razina sigurnosnih postavki, ...

**Razine raspoređivanja poslova** definiraju koliko detaljno obradu nekog zahtjeva raspoređivač može podijeliti i rasporediti. Postoje tri razine raspoređivanja:

- raspoređivanje procesa – u tom slučaju su procesi kompaktne cjeline koje se ne mogu dijeliti na manje dijelove;
- raspoređivanje zadaća – to je skup nezavisnih procesa koji ostvaruju istu logiku obrade podataka. Raspoređivač određuje broj zadaća koja je potrebna da bi se ostvarila obrada ulaznih podataka i
- raspoređivanje ovisnih poslova – poslovi imaju složene međuovisnosti (moraju se izvoditi slijedno, mogu se izvoditi istodobno tj. konkurentno, moguća su grananja tijeka izvođenja, potrebno je spojiti izvođenje poslova). Raspoređivanje se ostvaruje na temelju dostupnosti sredstava i međuovisnosti poslova.

Da bi se objasnio **smještaj raspoređivača** prvo pogledajmo model grozda računala (Slika 12.9) na kojem ćemo objasniti moguće smještaje raspoređivača.



Slika 12.9. Model grozda računala

Elementi prikazanog modela su:

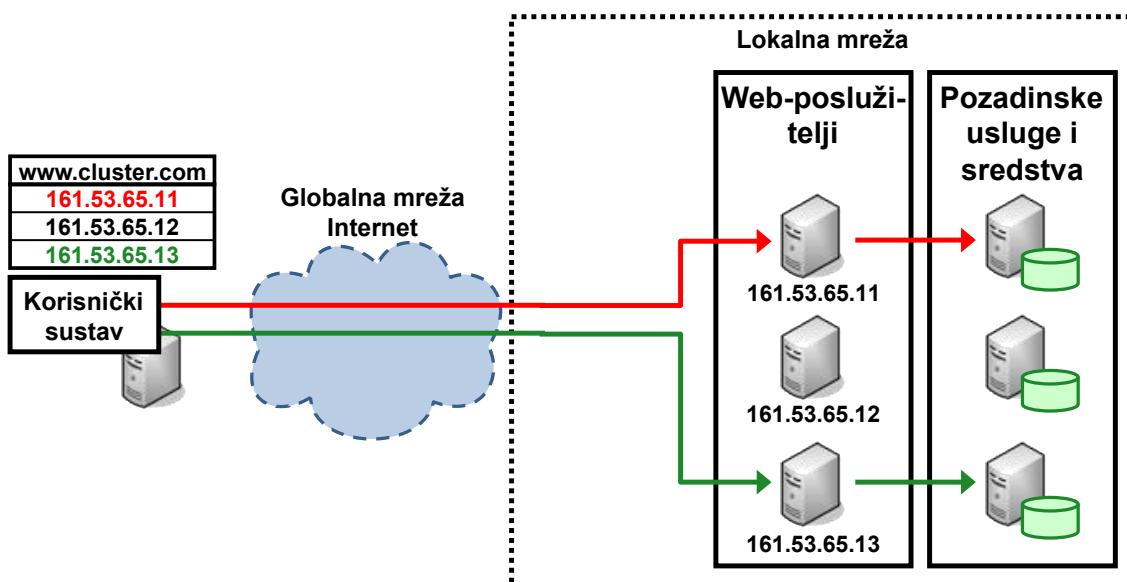
- korisnički sustav – klijentska aplikacija kojom korisnik ostvaruje pristup i koristi sredstva i usluge na grozdu računala,
- DNS poslužitelj korisničke strane – pomoću njega korisnički sustav razlučuje adrese udaljenih računala u Internetu,
- DNS poslužitelj poslužiteljske strane – razlučuje adrese poslužitelja u lokalnoj mreži,
- mrežni usmjeritelj – prihvata, analizira i usmjerava pristigne zahtjeve,

- web-poslužitelji i pozadinska sredstva i usluge.

Raspoređivanje se može vršiti na:

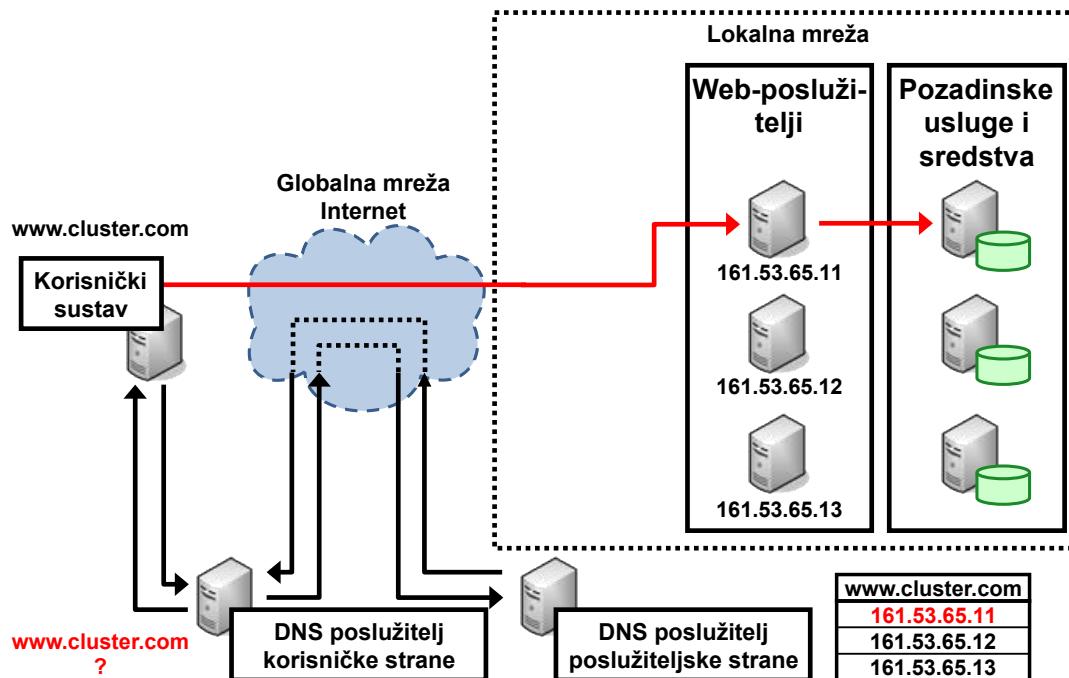
- strani klijenta,
- na DNS poslužitelju ili
- na mrežnom usmjeritelju.

Raspoređivanje na klijentu je prikazano slikom (Slika 12.10). Korisnički sustav sadrži popis svih pristupnih adresa grozda računala. On odabire jedno od pristupnih računala i proslijeđuje zahtjev odabranom računalu (označeno crveno). Sljedeći put odabire iduće pristupno računalo i proslijeđuje zahtjev odabranom računalu (označeno zeleno). Postupak odabira može biti zasnovan na različitim disciplinama posluživanja. Prednost ovog pristupa je primjena jednostavne arhitekture koja ne uključuje dodatne sklopojske i programske elemente za ostvarivanje proslijeđivanja. Nedostatak je da ovaj pristup zahtjeva izmjene u logici korisničkog sustava s obzirom da korisnički sustav mora biti svjesan raspodijeljenosti sustava poslužitelja.



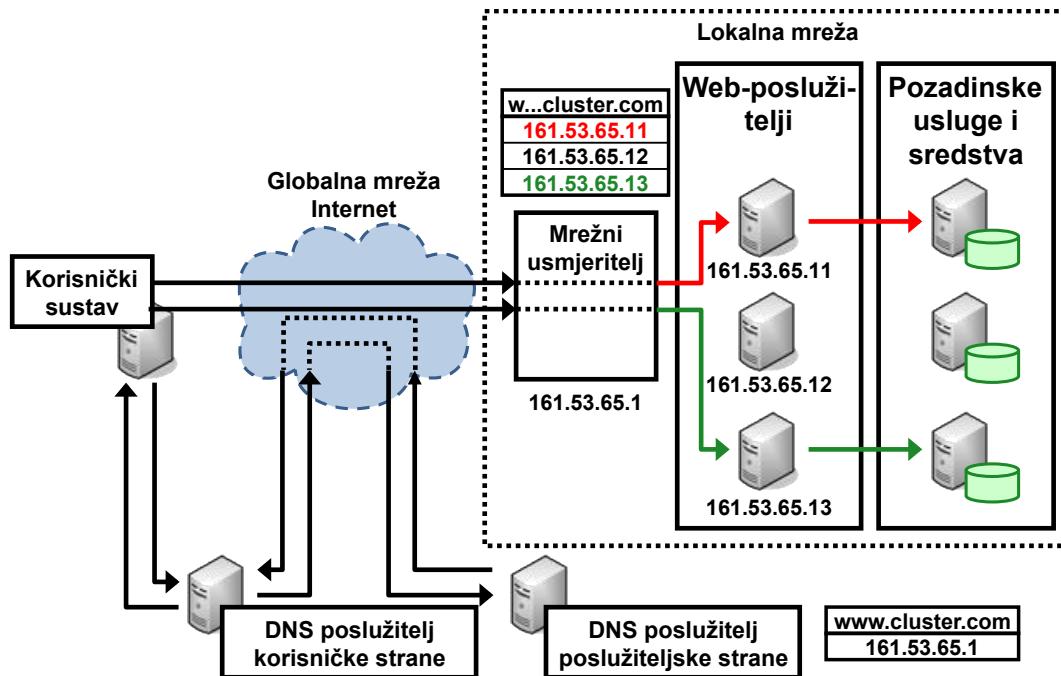
Slika 12.10. Raspoređivanje na strani klijenta

Raspoređivanje na DNS poslužitelju je prikazano slikom (Slika 12.11). Grozd računala ima jedno ime (www.cluster.com). Na temelju zadano imena, primjenom DNS-a ostvaruje se razlučivanje imena na strani poslužiteljskog DNS poslužitelja. Poslužiteljski DNS poslužitelj sadrži više adresa grozda računala i odabire odredišno računalo te IP adresu odabranog računala vraća kao rezultat. Nakon razlučivanja imena, korisnik proslijeđuje zahtjev na odabrano pristupno računalo (označeno crveno). Prednost je da korisnička strana ne mora znati za raspodijeljenost grozda, nego ga koristi kao da je to jedno računalo. Nedostatak je održavanje zapisa u DNS poslužitelju koje uzrokuje da jedan klijent neko vrijeme (dok ne istekne valjanost razlučivanja) koristi jedno poslužiteljsko računalo.



Slika 12.11. Raspoređivanje na DNS poslužitelju

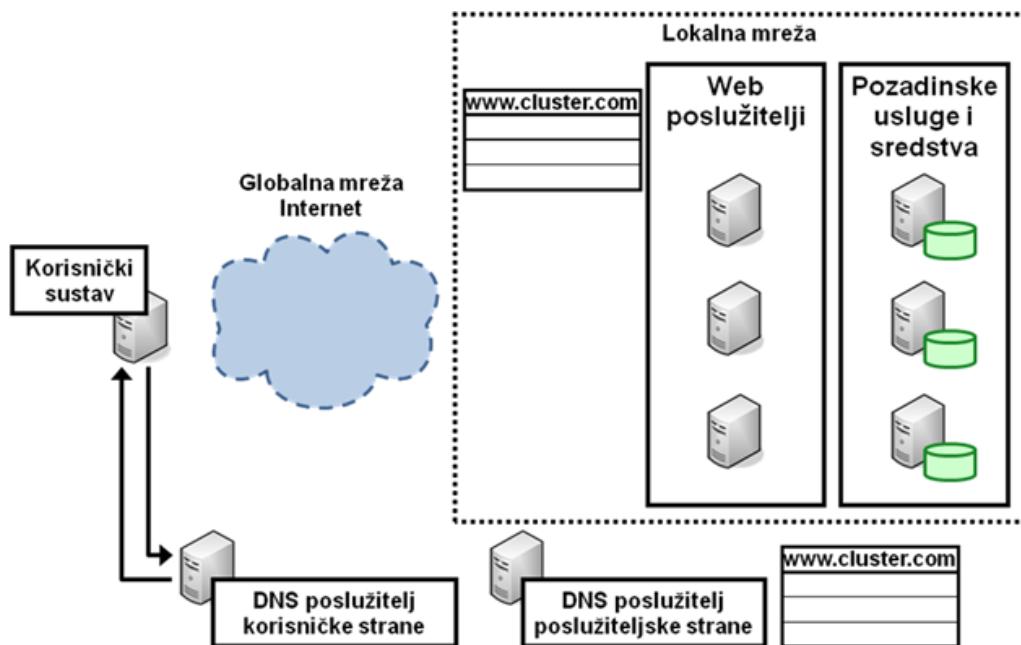
Raspoređivanje na mrežnom usmjeritelju je prikazano slikom (Slika 12.12). Grozd računala i ovdje ima jedno ime ([www.cluster.com](http://www.cluster.com)). U DNS sustavu za ime [www.cluster.com](http://www.cluster.com) prijavljena je samo jedna adresa, adresa mrežnog usmjeritelja koji je pristupna točka grozdu računala. Nakon razlučivanja imena, korisnik prosjeđuje zahtjev mrežnom usmjeritelju (označeno crveno). Mrežni usmjeritelj sadrži popis svih adresa web-poslužitelja grozda. On odabire jedan web-poslužitelj i prosjeđuje mu zahtjev (označeno crveno). Za sljedeći zahtjev mrežni usmjeritelj odabire neki drugi web-poslužitelj (označeno zeleno). Postupak odabira može biti zasnovan na različitim disciplinama posluživanja. Prednost je da korisnički sustav ne razlikuje korištenje grozda od korištenja jednog poslužitelja. Nedostatak je uporaba i održavanje zapisa u mrežnom usmjeritelju kao i sav promet koji prolazi kroz njega prema svim klijentima.



Slika 12.12. Raspoređivanje na mrežnom usmjeritelju

## 12.5 Pitanja za učenje i ponavljanje

- 12.1. Skicirajte i ukratko objasnite slojevitu arhitekturu spleta računala.
- 12.2. Na primjeru opišite značajke raspoređivanja zasnovanog na korištenju prostorne lokalnosti.
- 12.3. Prikažite i opišite elemente modela grozda računala.
- 12.4. Nadopunite skicu i objasnite primjer ostvarivanja razmjernog rasta sustava primjenom metode DNS poslužitelja.



## 13 LITERATURA

- Bacon, J., & Harris, T. (2003). *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley.
- Bernstein, P. A. (1996). Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39 (2), 86-98.
- Birrell, A. D., & Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2 (1), 39-59.
- Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design (5th edition)*. Addison-Wesley.
- Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doktorska disertacija, University of California.
- Gunther, N.J. (2010). *Analyzing Computer System Performance with Perl*, Springer.
- Gunther, N.J. (2007). *Guerilla Capacity Planning*, Springer, 2007.
- Gunther, N.J. (1998 i 2000). *The practical performance analyst*, Mcgraw Hill i Authors Choice Press
- Ince, D. (2004). *Developing Distributed and E-commerce Applications 2nd edition*. Pearson Education Limited.
- Lua, Keong; Pias, Marcelo; Sharma, Ravi; Lim, Steven : "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes", IEEE communication survey and tutorial, 2005, pp. 72- 93. <http://www.cl.cam.ac.uk/teaching/2005/AdvSysTop/survey.pdf>
- Kshemkalyani, A., & Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.
- Lamport, L., Shostak, R., & Pease, M. (1982). The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4 (3), 382-401.
- Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc.
- Menascé, D.A. & Almeida, V.A.F. (2002). *Capacity Planning for Web Services*, Prentice Hall.
- Nemeth, G., & Lovrek, I. & Sinkovic, V. (1997). *Scheduling Problems in Parallel Systems for Telecommunications*, Computing, Vol. 58, No. 3, pp. 199-223.
- Foster, I., & Kesselman, C. (Ur.). (2003). The Grid 2: Blueprint for a New Computing Infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Nabrzyski, J., Schopf, J. M., & Weglarz, J. (Ur.). (2004). Grid resource management: state of the art and future trends. Norwell, MA, USA: Kluwer Academic Publishers.
- Perić, Katarina. Usporedba mreža s ravnopravnim sudionicima / diplomski rad. Zagreb : Fakultet elektrotehnike i računarstva, 17.11. 2009, 62 str. Mentor: Podnar Žarko, Ivana.
- Podnar, I. (2004). *Service Architecture for Content Dissemination to Mobile Users*. Zagreb: Doktorska disertacija, Sveučilište u Zagrebu (FER).
- Sinković, V. (1994.) *Informacijske mreže*, Školska knjiga, Zagreb.
- Sinkovic, V. & Lovrek, I. & Nemeth, G. (1999). *Load Balancing in Distributed Parallel Systems for Telecommunications*, Computing, Vol. 63, No. 3, pp. 201-218.

Sinkovic, V. & Lovrek, I. (1997). *A Model of Massively Parallel Call and Service Processing in Telecommunications*, Journal of Systems Architecture, Vol. 43, No. 6-7, pp. 479-490.

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall.

Vrsalovic, D.F. & al. (1998). *Performance prediction and calibration for a class of multiprocessors*, IEEE Transactions on Computers, Vol. 37, No. 11, pp. 1353 -1365.