

# Distribuirani sistemi

## (VIII semestar 2+2+1)



Predavanja: Emina Milovanović, Natalija Stojanović  
Rač. vežbe: Natalija Stojanović, Aleksandar Veljanovski, Igor Antolović  
Lab. vežbe: Aleksandar Veljanovski, Aleksandra Stojnev  
Web stranica predmeta: [cs.elfak.ni.ac.rs/nastava/](http://cs.elfak.ni.ac.rs/nastava/)

# Način polaganja i literatura

## Način polaganja:

- |   |               |
|---|---------------|
| 1. Lab. Vežbe   | : 0 – 20      |
| (nakon svake oblasti, 4 lab. vežbe, ocena<br>preko kviza) |               |
| 2. I kolokvijum   | : 0 – 20(>10) |
| 3. II kolokvijum  | : 0 – 20(>10) |
| 4. Pismeni  | : 0 – 40(>20) |
| 5. Usmeni   | : 0 – 40(>20) |

1. Konačna ocena =  $1+2+3+5$  ( $1+2+3 > 30$ )  
2. Konačna ocena =  $1+4+5$  ( $1+4 > 30$ )  
Ako svaki zadatak na pisanom delu ispita  
nosi 25 poena potrebno je imati minimum 10  
poena na svakom zadatku

## Literatura:

1. Andrew S. Tanenbaum, Maarten Van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, NJ, 2007.
2. George Coulouris, Jean Dollimore, Tim Kondberg, *Distributed Systems: Concepts and design*, Fifth Edition, Addison Wesley, 2012.

# Uvod

- \* Od 1945. do sredine 1980-ih računari su bili veoma skupi i glomazni.
- \* Sredinom 1980-ih napredak u tehnologiji je doprineo promeni situacije.
- \* Dve stvari su odigrale ključnu ulogu:
  - Pojava mikroprocesora
    - Pojava mikroprocesora omogućila je gradnju RS po niskoj ceni uz istovremeno poboljšanje performansi
    - računari su postali manji, jeftiniji, brži, efikasniji sa stanovišta potrošnje energije
  - Mogućnost projektovanja veoma brzih računarskih mreža
    - Veoma brze mreže omogućile su razmenu poruka između dva računara brzinom od 10Mbit/s do 1Gbit/s.
  - Kao rezultat ovih tehnologija postalo je ne samo moguće, već i isplativo, povezati veliki broj računara korišćenjem veoma brzih mreža.

memorije  
16,000x jeftije  
4,000x veći kapacitet

godina	Cena \$/MB	Tipična veličina
1977	\$32,000	16K
1987	\$250	640K-2MB
1997	\$2	64MB-256MB
2007	\$0.06	512MB-2GB+

35,000x  
brzina prenosa

## \* brzine prenosa u LAN

- 1980: Original Ethernet: 2.94 Mbps
- 1985: thick Ethernet: 10 Mbps
- 1991: 10BaseT - twisted pair: 10 Mbps
- 1995: 100 Mbps Ethernet
- 1998: 1 Gbps (Gigabit) Ethernet
- 1999: 802.11b (wireless Ethernet) standardized
- 2001: 10 Gbps
- 2005: 100 Gbps (preko optičkog linka)

# Rast interneta

Date	Hosts
Dec 1979	188
July 1989	130,000
July 1999	56,218,000
July 2001	125,888,197
July 2003	171,638,297
July 2005	353,284,187
July 2006	439,286,364
July 2007	489,774,269
July 2008	570,937,778
July 2009	681,064,561
July 2010	768,913,036
July 2011	849,869,781
July 2012	908,585,739
July 2013	996,230,757
July 2014	1,028,544,414
July 2015	1,033,836,245

Date	Hosts	Web servers
July 1993	1,776,000	130
July 1995	6,642,000	23,500
July 1997	19,540,000	1,203,096
July 1999	56,218,000	6,598,697
July 2001	125,888,197	31,299,592
July 2003	212,570,000	42,298,371
July 2005	353,284,187	67,571,581
July 2006	439,286,364	88,166,395
July 2007	489,774,269	125,626,329
July 2008	570,937,778	175,480,931
July 2009	681,064,561	239,611,111
July 2010	768,913,036	205,714,253
July 2011	849,869,781	357,292,065
July 2012	908,585,739	665,916,461
July 2013	996,230,757	698,823,509
July 2014	1,028,544,414	996,106,380
July 2015	1,033,836,245	849,602,745

Source: Internet Systems Consortium

# Šta je Distribuirani sistem?

*“A collection of independent computers that appears to its users as a single coherent system”*

-- A.S. Tanenbaum, M. van Steen

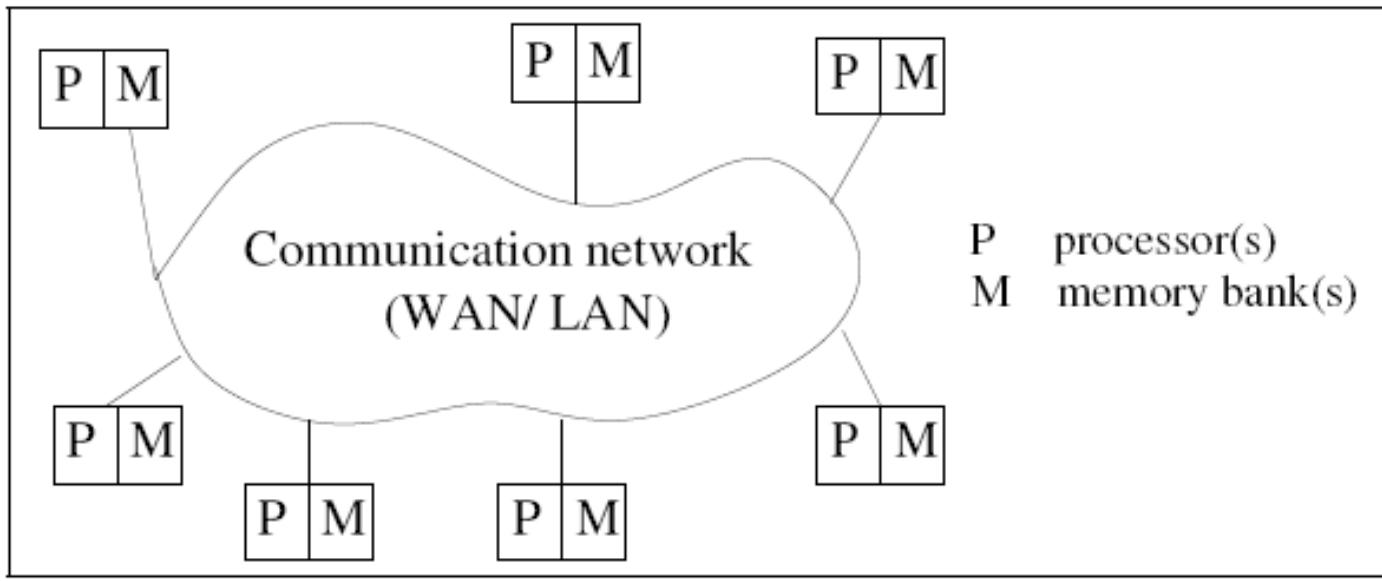
*“One in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages”*

-- G. Coulouris, J. Dollimore, T. Kindberg

*“One on which I cannot get any work done because some machine I have never heard of has crashed”*

-- L. Lamport

# Model distribuiranog sistema



DS povezuje procesore pomoću komunikacione podmreže

# Prednosti DS nad centralizovanim

## \* Ekonomski

- DS imaju bolji odnos cena/performanse od velikih (mainframe) računara.
  - Grace Hopper (poznati naučnik u oblasti CS): "In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers".
  - Umesto jednog mainframe računara koji košta desetine miliona dolara, ista moć obrade se može dobiti gradnjom klastera za detekciju hiljada dolara

## \* Brzina:

- DS može imati veću ukupnu moć obrade nego mainframe.
  - Npr. Jedna velika baza podataka može biti razbijena na manje baze. Na taj način se vreme odziva može poboljšati
  - Distribuirane baze mogu biti hijerarhijski organizovane (primer DNS).
  - Google

## \* Pouzdanost:

- Ako imate centralizovani sistem i računar otkaže, aplikacija ne može da se izvršava
- Kod DS (ako je dobro projektovan) ako 15% mašina otkaže, aplikacija i dalje može da se izvršava, tj. sistem i dalje može ostati u funkciji sa samo 15% degradacije u performansama.
  - Npr. Google (hiljade čvorova može biti van funkcije u svakom trenutku, ali je usluga pretraživanja i dalje dostupna).
    - To je zato što je sistem dobro projektovan!

## \* Inkrementalni rast:

- sistem se može postupno povećavati dodavanjem novih računara.

## \* Deljenje resursa:

- Omogućeno je da više korisnika pristupa zajedničkim bazama podataka i periferijama.
  - Tipičan primer distribuirani fajl sistem
  - web

## \* Komunikacija:

- olakšana komunikacija između ljudi.

## \* Efikasno korišćenje resursa:

- omogućava da se opterećenje rasporedi na raspoložive računare na najefikasniji način

# Mane

## \* Mnogo veća kompleksnost DS od centralizovanog

- Softver:

- teže je razviti softver za DS nego za centralizovane sisteme
  - Ima više aktivnih procesa, koji međusobno komuniciraju
    - » Debugiranje ( u slučaju da postoji greška) može biti noćna mora
- Neke od kompleksnosti mogu biti skrivene ili pomoću programskog jezika (biblioteke, framework-a) koji se koristi za pisanje aplikacije, ili OS ali to samo pomera problem sa aplikacije na PJ i/ili OS – sistem ostaje kompleksan
  - Distribuiranu aplikaciju ne možete napisati ako PJ koji koristite ne podržava interprocesnu komunikaciju.
  - Isto važi i za OS

## \* Umrežavanje:

- Povezanost je ključna za funkcionisanje DS

- mreža može otkazati ili postati zagušena

## \* Bezbednosni problemi

- Laka dostupnost resursa, dovodi do problema bezbednosti

- Dosadašnje metode zaštite pružaju relativno nizak nivo bezbednosti
  - lozinke i druge osteljive informacije se često šalju kroz mrežu kao običan tekst (bez šifriranja)

# Osnovne osobine DS

- \* Heterogenost
- \* Transparentnost
- \* Otvorenost
- \* Skalabilnost

# Heterogenost

\* DS je sastavljen od heterogenog skupa računara.

- Heterogenost se ogleda u sledećem
  - Hardver računara – različit skup instrukcija, različita interna prezentacija podataka.
  - Operativni sistemi – interfejs za razmenu poruka se razlikuje od OS do OS (npr. Berkely i Winsoc).
  - Programski jezici – karakteri i strukture podataka se različito predstavljaju u različitim programskim jezicima, što može biti problem ako aplikacije treba međusobno da komuniciraju.
    - Implementacije od strane različitih projektanata – sve dok se zajednički standardi ne usvoje, različite implementacije ne mogu međusobno da komuniciraju. (OSI model je jedan način za rešavanje problema heterogenosti).

# Transparentnost

## \* Transparentnost znači skrivanje nečega

- Važan cilj kod projektovanja DS je skrivanje činjenice da su njegovi procesi i resursi fizički distribuirani.
  - DS koji je u stanju da sakrije ove činjenice naziva se transparentnim
- Transparentnost je važan zahtev pri realizaciji jedinstvenog pogleda na sistem (single system image) koje čine da se DS može koristiti sa istom lakoćom kao jednoprocesorski sistem.
- Npr. Web – dozvoljava pristup informacijama, a da pri tome ne moramo znati na kom računaru se one nalaze; dovoljno je samo kliknuti na link!

# Tipovi transparentnosti

## \* Pristupna transparentnost –

- podacima i resursima se pristupa na jedinstven način, bez obzira da li se oni nalaze na udaljenom ili lokalnom računaru
  - Npr. Različiti OS mogu koristiti različite načine imenovanja fajlova.
    - Razlike u imenovanju fajlova i način manipulacije fajlovima mora biti skriven za korisnika i aplikaciju

## \* Lokacijska transparentnost –

- korisnik ne zna (i ne mora da zna) gde je resurs fizički lociran u sistemu.
  - Imenovanje i ovde igra važnu ulogu.
  - Lokaciona transparentnost se može postići dodeljivanjem logičkih imena resursima
    - Primer URL imena:[http:// www.prenhall.com/index.htm](http://www.prenhall.com/index.htm) ne nagoveštava gde se nalazi glavni web server izdavačke kuće Prentice Hall
    - Korisnik sa bilo koje lokacije na isti način pristupa resursu

## \* Migraciona transparentnost-

- Resurs može promeniti lokaciju a da klijent to ne primeti.
  - resurs se može kretati a da pri tome ne dolazi do promene imena resursa.

# Tipovi transparentnosti (nast.)

## \* Transparentnost konkurencije

- Omogućava da više procesa jednovremeno koristi isti resurs
  - Npr. Istovremeni pristup istom fajl severu ili istoj tabeli u deljivoj bazi podataka.
- Važno je postići da korisnici ne primete da se isti resurs koristi jednovremeno
- Jednovremeni pristup deljivom resursu mora ostaviti resurs u konzistentnom stanju
- Konzistentnost se postiže mehanizmima zaključavanja i sinhronizacije tako da se pristup deljivom resursu serijalizuje (npr. korišćenjem semafora).

# Tipovi transparentnosti (nast.)

## \* Transparentnost replikacije (umnožavanja)

- Omogućava postojanje više kopija istog resursa u DS u cilju povećanja performansi i pouzdanosti
  - Kopija se postavlja bliže mestu odakle se obavlja pristup
  - Sve kopoje imaju isto ime
    - To znači da ako DS podržava replikaciju da automatski podržava i lokacionu transparentnost!
- Korisnici ili aplikacija nisu svesni postojanja više kopija resursa.
- Primer: DFS

## \* Transparentnost za otkaze-

- korisnik ne zna da je resurs u kvaru ( ovo je jedan od najtežih problema kod DS. Sistem koji je otporan na otkaze mora biti u stanju da izvrši realokaciju resursa na delove sistema koji korektno funkcionišu)
  - Ponekad je veoma teško razlučiti da li je neki server (usluga) nedostupan ili je mreža zagušena.

# Tipovi transparentnosti (nast.)

## \* Transparentnost paralelizacije

- Paralelizacija procesa se izvršava transparentno za aplikativnog programera i korisnika aplikacije
- Korisnik nije svestan da i kako je aplikacija paralelizovana i gde se izvršavaju procesi
- Primer: Hadoop

# Otvorenost

\* **Otvoreni DS dozvoljavaju dodavanje novih servisa i omogućavaju dostupnost servisima od strane različitih klijenata**

- Npr web servisu se može pristupiti prko mnogo različitih klijenta (chrome, opera, explorer,...)

● **Otvoreni DS je sistem koji nudi usluge (servise) shodno standardnim pravilima koja definišu sintaksu i semantiku usluga.**

- Npr. U RM standardi definišu format, sadržaj i značenje poruka koje se razmenjuju između istih nivoa u protokolu steku. (pravila su opisana protokolima i javno su dostupna u obliku RFC, na primer)

● **Kod DS usluge su definisane interfejsima.**

● **Interfejs definiše kako se pristupa usluzi**

● **Interfejsi se opisuju pomoću posebnog jezika (IDL – Interface definition Language)**

- Opis interfeisa sadrži definicije funkcija koje su dostupne zajedno sa tipom parametara, tipom rezultata i mogući izuzecima (exceptions)
- Ako je interfejs dobro definisan, on omogućava proizvoljnom procesu kome je usluga potrebana da pristupi usluzi

# Skalabilnost

- \* Skalabilnost (proširljivost) DS je jedan od najvažnijih projektantskih zadataka.
- \* Skalabilnost se može posmatrati kroz tri dimenzijs:
  - Skalabilnost u odnosu na broj korisnika i resursa
  - Skalabilnost u odnosu na geografsku udaljenost resursa i korisnika
  - Administrativna skalabilnost – sistemom se može lako upravljati čak i ako se prostire kroz više administrativnih domena.
- \* Skalabilnost se ipak plaća performansama sistema!
  - Ako je potrebno podržati više korisnika ili resursa često dolazi do problema zbog centralizovanih usluga, podataka ili algoritama.
    - Mnoge usluge su centralizovane u smislu da su implementirane samo na jednom serveru u DS.
      - Server može postati usko grlo u sistemu sa porastom broja korisnika.
      - Replikacija je moguća, ali nekad je neophodno korišćenje samo jednog servera (npr. Server sa bankovnim računima građana - kopiranje servera na nekoliko lokacija bi stvorilo dodatne bezbednosne probleme)
    - Centralizovani podaci, kao i centralizovane usluge, mogu stvoriti probleme
      - Npr. Da DNS nije implementiran kao distribuirana baza podataka, Internet bi imao malo smisla, jer bi zahtevi za preslikavanje imena host u IP adresu bili upućivani istom serveru koji bi postao usko grlo
    - Korišćenje centralizovanih algoritama nije poželjno u DS
      - Npr centralizovani algoritam za rutiranje
    - Treba koristiti samo decentralizovane algoritme

# Skalabilnost (nast.)

- Decentralizovani algoritmi imaju sledeće osobine
  - Ni jedna mašina nema kompletну informaciju o stanju sistema
  - Mašine donose odluku samo na osnovu lokalnih podataka
  - Otkaz jedne mašine ne narušava sistem
  - Nema prepostavke o postojanju globalnog časovnika!
- Poslednja osobina je veoma važna (mada nije očigledna)
  - Npr. Bilo koji algoritam koji bi zahtevao sledeće: "Tačno u 12:00:00 mašne treba da provere veličinu svog reda čekanja" ne bi korektno radio jer je nemoguće postići tačnu sinhronizaciju časovnika u DS.
  - Distribuirani algoritmi moraju da uzmu u obzir da ne postoji tačna (egzaktna) sinhronizacija časovnika.
  - Što je veći sistem, to je neslaganje veće.

# Skalabilnost (nast.)

## \* Geografska skalabilnost ima svoje probleme

- Jedan od problema je što je teško proširiti DS koji je bio projektovan u okviru LAN.
  - U LAN je komunikacija između procesa sinhrona
    - Kod sinhrone komunikacije, strana koja zahteva uslugu (klijent) se blokira dok čeka na odgovor od severa.
    - Ovakva komunikacija dobro funkcioniše u LAN jer je vreme čekanja kratko (nekoliko  $\mu$ s), dok je u WAN duže i za tri reda veličine (što je za interaktivne aplikacije dugo)

# Tehnike skaliranja

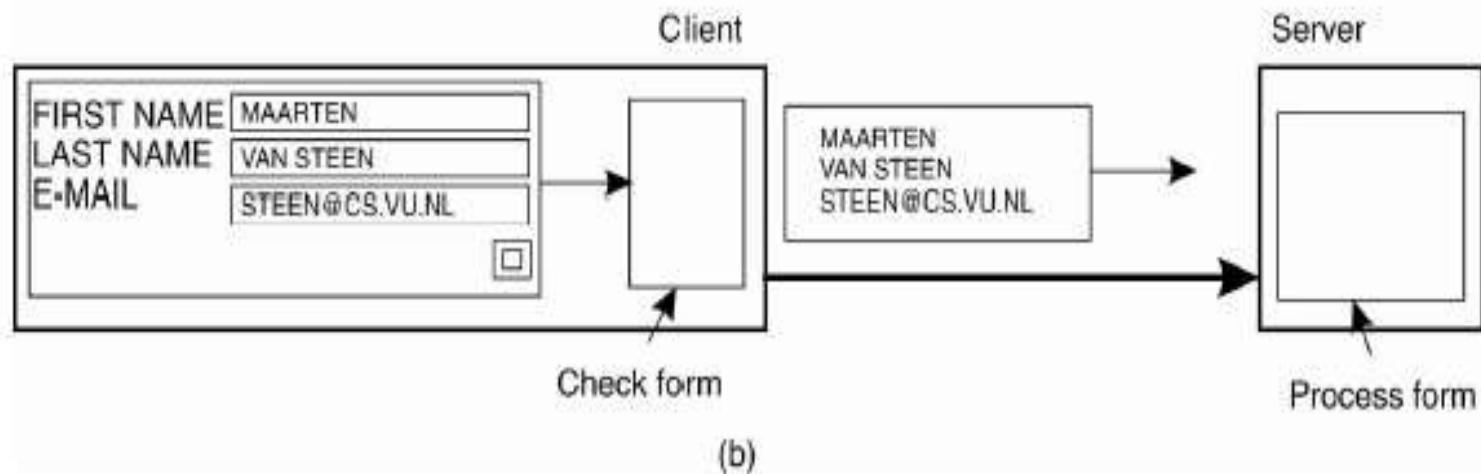
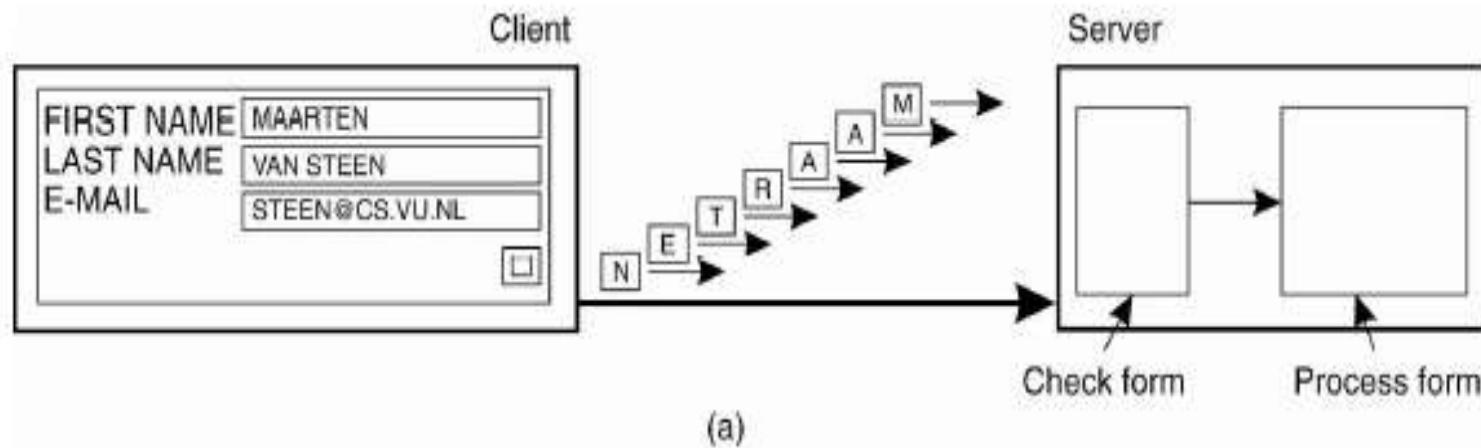
## \* Tri tehnike

- Skrivanje komunikacionog kašnjenja
- Distribucija
- Replikacija

## \* Skrivanje komunikacionog kašnjenja

- Koriste se asinhrone komunikacije umesto sinhronih
  - Kada se uputi zahtev udaljenom servisu, klijent se ne blokira dok čeka da stigne odgovor, već radi neki drugi posao
  - Kada stigne odgovor od servera, generiše se prekid koji obaveštava da je poruka stigla
- Download-ovati deo koda na klijent stranu da bi se ubrzala obrada
  - Asinhrone komunikacije nisu od koristi kod interaktivnih aplikacija
    - U takvim slučajevima je mnogo bolje rešenje smanjiti potrebe za komunikacijom, tako što se recimo, deo obrade koji se normalno izvršava na serverskoj strani, izvršava na strani klijenta (ovo se koristi kod weba u formi Java Script koda).
    - Tipičan primer gde ovaj prilaz ima smisla je pristup bazama podataka koje koriste forme.
      - » Uobičajeno je da se popunjavanje forme obavlja slanjem odvojene poruke za svako polje u formi, a zatim čeka potvrda od servera. (server recimo može obaviti sintaksnu proveru pre nego što prihvati upis)
      - » Mnogo bolje rešenje je da se klijentu uz formu isporuči i deo koda za popunjavanje forme ( i eventualno proveru grešaka), a da klijent serveru vrati kompletno popunjenu formu

# Tehnike skaliranja (nast.)



- a) Tradicionalni način procesiranja formi
- b) Procesiranje formi na strani klijenta

# Tehnike skaliranja (nast.)

## \* Distribucija

- Podrazumeva da se komponente dele na manje delove, a zatim se ti delovi distribuiraju na više mašina u sistemu
  - Tipičan primer je organizacija DNS
  - Web je takođe primer korišćenja distribucije.
    - Korisniku web može izgledati kao jedinstveni sistem u kome se nalaze dokumenti koji su identifikovani svojim jedinstvenim imenima (URL)
    - Međutim, web je fizički distribuiran kroz veliki broj web servera
      - » Jedino zbog ovakve distribucije dokumenta web je u stanju da opsluži veliki broj korisnika.

## \* Replikacija

- Pomaže da se poveća dostupnost i balansira opterećenje u sistemu da bi se postigle bolje performanse
  - U sistemima koji su geografski distribuirani poželjno je da postoji kopija resursa blizu mesta korišćenja kako bi se smanjilo komunikaciono kašnjenje.
    - Postojanje replika može da dovede do problema konzistentnosti (npr. modifikovanje jedne kopije)

## \* tehnike skaliranja mogu prouzrokovati probleme

- postojanje više kopija (u kešu ili repliciranih) dovodi do problema konzistencije – modifikacija jedne kopije dovodi do neslaganja sa ostalim kopijama.
- da bi se kopije stalno održavale konzistentnim potrebna je globalna sinhronizacija
- globalnu sinhronizaciju je gotovo nemoguće postići u distribuiranom sistemu (ili je neisplativo sa stanovišta performansi)

## \* DS može tolerisati izvestan nivo nekonzistentnosti!

# Šta je middleware

\* Komunikacija između procesa je u srcu svih distribuiranih sistema.

- Komunikacija u DS je uvek bazirana na slanju poruka, tj. ne postoji zajednički adresni prostor.
- Da bi se pojednostavilo pisanje i razvoj distribuiranih aplikacija potrebno je obezbediti softversku podršku koja će aplikativnog programera osloboditi detalja vezanih za interprocesnu komunikaciju, sinhronizaciju, bezbednost itd.

\* U OSI referentnom modelu, poslednji nivo je aplikativni.

- Ovaj nivo je originalno trebalo podrška za standarne mrežne aplikacije kao što su e-mail, prenos fajlova, emulaciju terminala i sl.

\* Ono što OSI referentnom modelu nedostaje je jasna razlika (granica) između aplikacije, aplikativno-specifičnih protokola i protokola opšte namene (general purpose protocols).

- Npr. Internetov FTP protokol je protokol za razmenu fajlova između klijenta i servera.
  - Ovaj protokol ne treba poistovećivati sa *ftp* programom koji predstavlja korisničku aplikaciju za prenos fajlova.
- Drugi primer aplikativno specifičnog protokola je HTTP koji omogućava prenos web stranica.
  - Ovaj protokol je implementiran u aplikacijama kao što su web browser-i i web serveri.

\* Postoji i mnogo protokola opšte namene koji su od koristi za mnoge aplikacije, a ne mogu se kvalifikovati kao transportni protokoli.

- Ti protokoli spadaju u kategoriju *middleware* protokola.
  - Ti protokoli pružaju različite usluge aplikaciji, tj. aplikativnom nivou.
- Npr. jedna od usluga koju ovi protokoli pružaju je usluga autentifikacije (provjere identiteta) i autorizacije (prava pristupa) koje se zahtevaju u mnogim aplikacijama.
  - Zbog toga ima smisla ovu uslugu integrisati u middleware sistem kao opštu slugu.
- Drugi primer usluge koja se često zahteva je sinhronizacija.
  - Pri tome pod sinhronizacijom podrazumevamo neku vrstu vremenske sinhronizacije (da se utvrdi redosled događaja u DS) ili sinhronizaciju kod pristupa zajedničkom deljivom resursu (uzajamno isključivanje).
    - Ta usluga je nezavisna od aplikacije i ima je smisla implementirati kao uslugu koju mogu koristiti različite aplikacije

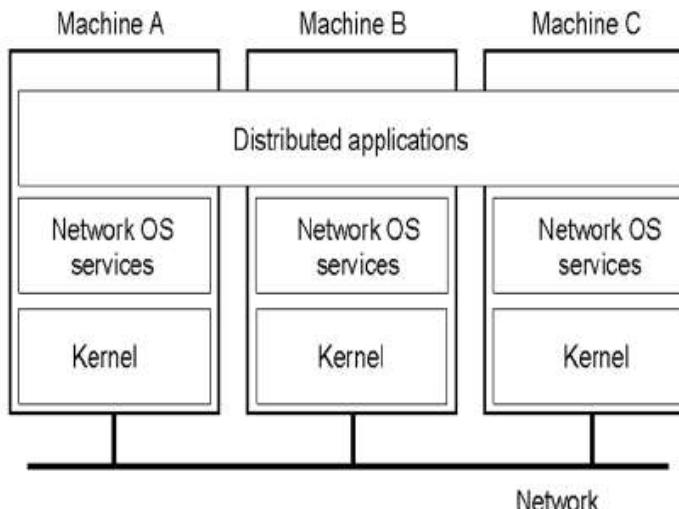
- Kada je reč o komunikaciji, videli smo da transportni protokoli TCP i UDP omogućavaju razmenu poruka između distribuiranih procesa.

- Međutim, ta usluga je implementirana na niskom nivou i zahteva dosta napora pri programiranju distribuiranih aplikacija.
  - Npr. da bi aplikacija pristupila transportnim uslugama programer koristi skup API-ja koje mu nudi lokalni OS: kreiranje i otvaranje socket-a, kreiranje bafera, upis u bafer, čitanje iz bafera itd.
- Middleware komunikacioni protokoli pružaju viši nivo komunikacionih usluga i oslobođaju aplikativnog programera detalja vezanih za komunikaciju između procesa.
  - Komunikacija je skrivena iza poziva procedure (remote procedure call - RPC) ili metoda (remote method invocation - RMI) ili je na raspolaganju skup funkcija za razmenu poruka (message oriented ).

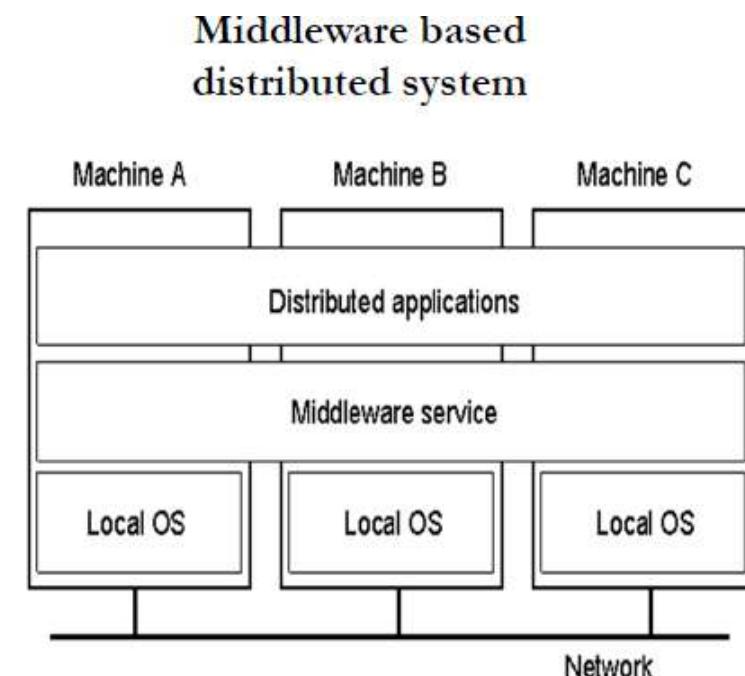
# Middleware

## \* Platforma na kojoj su izgrađeni DS su računarske mreže

- Mrežni OS pruža mogućnost korisnicima da pristupe uslugama koje su locirane na nekoj drugoj mašini (implementira npr. TCP/IP protokol stek)
- Da bi se pojednostavilo pisanje distribuiranih aplikacija i njihova integracija u DS na Mrežni OS je moguće je dodati još jedan softverski sloj (MIDDLEWARE) sa ciljem da sakrije heterogenost platforme na kojoj je sistem izgrađen od aplikacije, i da se sakrije komunikacija, tj. distribuciju.



Network/OS based  
distributed system



Middleware based  
distributed system

# Middleware (nast.)

\* Svaki lokalni OS obavlja upravljanje lokalnim resursima, pored toga što obezbeđuje komunikaciju sa drugim računarima

- Middleware ne uparavlja radom pojedinačnog čvora u mreži, to radi lokalni OS
- Osnovni cilj middleware je da se sakrije heterogenost platforme na kojoj je sistem izgrađen od aplikacije.
- Middleware sistemi nude kompletan skup usluga aplikaciji i ne dozvoljavaju korišćenje ničeg drugog do njihovih interfejsa prema uslugama
  - Nije moguće preskočiti Middleware nivo i direktno pozvati uslugu OS (npr. Koristiti sockete za komunikaciju sa drugim procesom)
- Na žalost ne postoji jedan standard koji definiše kako izgledaju protokoli u middleware nivou.
  - Kada je reč o middleware *komunikacionim* protokolima postoji nekoliko modela middleware: RPC, RMI, message oriented (MPI).

# RPC komunikacioni middleware model

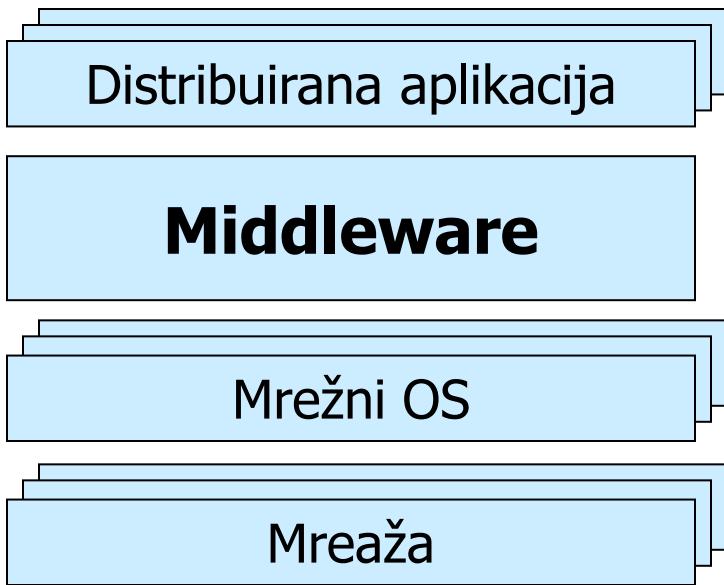
- \* Middleware model zasniva se na pozivu udaljenih procedura – RPC model (Remote Procedure Call). –
- \* resursi se modeliraju kao funkcije (procedure)
  - U ovom modelu naglasak je stavljen na skrivanje mrežnih komunikacija tako što je omogućeno procesima da pozivaju procedure čija je implementacija locirana na drugoj mašini.
  - Kada se poziva takva procedura, parametri se transparentno prenose do udaljene mašine na kojoj se zatim procedura izvršava, a rezultat izvršenja šalje pozivaocu.
  - Zbog transparentnosti komunikacije, pozivajući proces ima utisak da se procedura izvršila lokalno, tj. nije "svestan" da je došlo do komunikacije kroz mrežu (osim možda što je osetio usporenje u radu)
- \* Na ovaj način se postiže transparentnost u odnosu na lokaciju, implementaciju i jezik
- \* Problemi
  - Kako preneti parametre
  - Povezivanje (pronalaženje udaljene procedure)
  - Semantika u slučaju grešaka

# Modeli middleware (nast.)

- \* Objektno orijentisani pristup
- \* resursi se modeluju kao objekti koji sadrže skup podataka i funkcija nad podacima. Udaljenom resursu se pristupa kao objektu

- Logična posledica RPC modela: ako se pozivi procedura mogu realizovati van granica jedne mašine, onda je moguće pozvati i objekte koji su smešteni na udaljeniim mašinama.
- Ovaj prilaz doveo je do pojave različitih middleware sistema koji podržavaju koncept distribuiranih objekata.
  - Suština distribuiranih objekata je da svaki objekt implementira interfeis koji skriva sve unutrašnje detalje objekta od korisnika.
  - Interfeis sadrži deklaraciju metoda koje objekat implementira
  - Jedino što proces vidi od objekta je njegov interfeis.
  - Distribuirani objekti su često implementirani tako da je svaki objekt lociran na jednoj mašini, a interfeis je raspoloživ na mnogim drugim mašinama.
  - Kada proces pozove metod, interfeis lociran na lokalnoj mašini jednostavno transformiše poziv metoda u poruku koja se prosleđuje objektu
  - Objekt izvršava traženi metod i vraća rezultat
  - Interfeis transformiše odgovor u vrednost koja se vraća koju zatim koristi pozivajući proces.
  - Kao i u slučaju RPC proces ne mora biti svestan mrežne komunikacije.

# Middleware modeli



(RPC, RMI, MPI, CORBA - remote calls, object invocation, messages, ...)

(sockets, IP, TCP, UDP, ...)

# Tipovi DS

## \* Na arhitekturnom nivou

- klijent-server
- peer-to-peer

## \* U odnosu na oblasti primene

- Distribuirani računarski sistemi
  - Klasteri
  - Grid
- Distribuirani informacioni sistemi
  - Orjentisani ka poslovnim aplikacijama
    - Sistemi za obradu transakcija
    - Sistemi za integraciju poslovnih aplikacija
- Ugrađeni (sveprisutni) DS
  - Moblini računari, ugrađeni (embeded) računari, komunikacioni sistemi

# Distribuirani računarski sistemi

\* Koriste se za izvršenje visokoperformansnih zadataka (izračunavanja)

- Klasteri

- Računari se sastoje od grupe sličnih radnih stanica ili PC, koji se nalaze na malom rastojanju, povezani preko veoma brze lokalne mreže.
- Svaki čvor izvršava isti OS

- Grid

- Sastoji se od grupe (federacije) računarskih sistema, koji mogu biti u različitim administrativnim domenima, razlikovati se u pogledu hw, sw i mrežne tehnologije koju koriste

# Klasteri

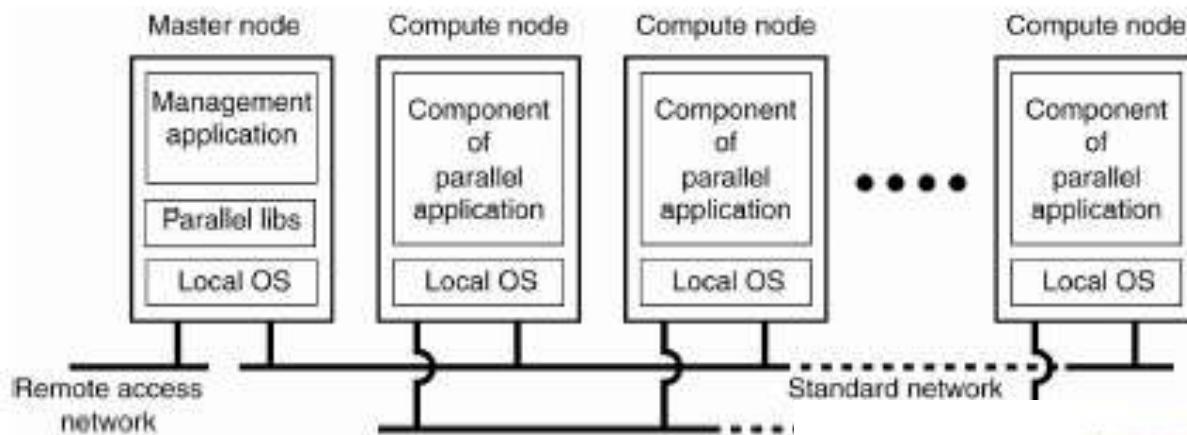
## \* Postali su popularni kada je odnos cena/performanse PC i radnih stanica postao povoljan.

- Postalo je isplativo i tehnički atraktivno napraviti superračunar koristeći raspoloživu tehnologiju povezivanjem relativno jednostavnih računara brzom komunikacionom mrežom.
- Ključna osobina homogenost
  - Svi čvorovi imaju isti OS, svi su povezani istom mrežom
  - Klasteri se koriste za paralelno izračunavanje

## \* Beowulf

- Klaster baziran na Linux OS.
- Sastoji se od grupe računarskih čvorova kojima upravlja jedan master čvor
  - Master obavlja alokaciju zadataka, upravlja redom čekanja procesa, obezbeđuje interfejs sa korisnikom.
  - Master u suštini izvršava middleware potreban za izvršenje programa i upravljanje klasterom
  - Ostali čvorovi najčešće imaju samo standardni OS

# Klasteri (nast.)



- \* Middleware sadrži biblioteku funkcija za izvršenje paralelnih programa
  - Funkcije obezbeđuju efikasne mehanizme za razmenu poruka
- \* MPI – Message Passing Interface
  - Najčešći tip middleware kod klastera



# Grid

\* Grid je distribuirani računarski sistem čije komponente mogu biti locirane na velikom geografskoom području

- Grid sistemi su veoma heterogeni a integracija je postignuta preko zajedničke servisno-orjentisane softverske arhitekture (middleware) .
- Resursi iz različitih organizacija (i administrativnih domena)
  - Resursi su različite usluge, računari, klasteri, diskovi, baze podataka, senzori,...
- Korisnici grida su organizovani u tzv. "viruelne organizacije"
  - Korisnici koji pripadaju istoj viruelnoj organizaciji imaju pravo pristupa resursima koje poseduje viruelna organizacija
- Middleware grid sistema treba da obezbedi pristup resursima iz različitih administrativnih domena i to samo onim osobama i aplikacijama koji pripadaju virtualnoj organizaciji.
  - Neophodno je obezrediti mehanizme za autentifikaciju (proveru identiteta) korisnika ili aplikacije.

# Distribuirani informacioni sistemi

## Sistemi za obradu transakcija

\* **Transakcija predstavlja skup operacija koje se obavljaju kao jedna nedeljiva (atomična) operacija.**

- Sistem za obradu transakcija obezbeđuje da sve ili ni jedna operacija u transakciji budu izvršene bez greške.
- nakon obavljanja transakcije sistem (obično baze podataka ili fajl sistem) mora da ostane u poznatom, konzistentnom stanju tako što obezbeđuje da se operacija koje su medjusobno zavisne ili sve izvrše ili se ni jedna ne izvrši.
- Primeri sistema
  - plećanje putem platnih kartica (bankomat vam isporučuje novac i umanjuje vrednost na vašem računu – ili će se obe operacije obaviti ili ni jedna)
  - sistem za rezervaciju avio karata (avio kompanija potvrdjuje rezervaciju i smanjuje broj slobodnih mesta, skida novac sa vaše platne kartice, (ponekad) povećava broj obroka za dati let,...)
- **transakcije se moraju procesirati uvek na isti način.** Da bi se to postiglo sistemi za obradu transakcija se projektuju tako da pružaju isti interfejs za svaku transakciju bez obzira na korisnika.

# tipične komande za rad sa transakcijama

## \* *Begin\_transaction*

- početak transakcije

## \* *End\_transaction*

- okončanje transakcije
  - izmedju *Begin\_transaction* i *End\_transaction* se nalazi telo transakcije

## \* *Abort\_transaction*

- okončanje transakcije i vraćanje na prethodne vrednosti

## \* *Read*

- čitanje podataka iz fajla, tabele,... za transakciju

## \* *Write*

- upis podataka u fajl, tabelu,.. za transakciju

# Osobine transakcija - ACID

- \* Svaka transakcija mora da zadovolji ACID test pre nego što se dozvoli modifikacija sadržaja
- \* A – Atomicity –
  - podrazumeva da se transakcija obavi kompletno ili se uopšte ne obavi
- \* C - Consistency –
  - transakcija ne ugrožava skup invarijanti (ograničenja, konstanti) sistema.
    - Npr. ako je uslov da sve transakcije nad bazom podataka moraju imati pozitivnu vrednost, bilo koja transakcija sa negativnom vrednošću će biti odbijena. (npr. ako pokušate da podignite više novca nego što imate na računu)
- \* I – Isolation –
  - transakcije su medjusobno izolovane, tj. dve konkurentne transakcije moraju biti serijalizovane (izvršavaju se u nekom nedeterminisanom redosledu, ali taj redosled mora biti vidljiv na isti način za sve transakcije)
- \* D - Durability –
  - kada se transakcija obavi ne može se poništiti!

# Sveprisutni Distribuirani sistemi

- \* Prva dva tipa DS se karakterišuu relativnom visokom stabilnošću:
  - čvorovi su fiksni i imaju uglavnom stalnu i kvalitetnu mrežnu konekciju
- \* Sa pojavom mobilnih i ugrađenih (embeded) računarskih sistema, stvari se menjaju: nestabilnost je uobičajeno ponašanje.
  - Uređaji u ovim DS su uglavnom mali, napajaju se pomoću baterija, mobilni su, imaju samo bežične veze.
  - Važna osobina ovakvih DS je odsustvo administrativnog upravljanja
    - Uređaji sami moraju da otkriju okruženje i ugnjezde se u njega
    - Uređaji moraju biti svesni da se okruženje stalno može menjati
- \* Primeri
  - Kućni sistemi
    - Organizovani oko jednog PC (automatsko paljenje svetla, sistemi za navodnjavanje, alarmni sistemi, različiti kućni aparati...)
  - Elektronski sistemi za monitoring pacijenata
  - Senzorske mreže – sistemi za akviziciju podataka i nadzor
    - Merenje temperature, vlažnosti,... i slanje informacija do bazne stanice gde se obavlja procesiranje

# Distribuirani sistemi



Komunikacija

# Poziv udaljene procedure (Remote Procedure Call – RPC)

\* DS se baziraju na eksplisitnom slanju poruka između procesa.

- Send/receive primitive ne skrivaju komunikaciju što je bitno da bi se postigla transparentnost pristupa
- Problem je bio dugo poznat, ali se nije nalazilo rešenje
- 1984. god Birell i Nelson su predložili potpuno novi način komunikacije:

➤ Dozvoliti programima da pozivaju procedure koje su locirane na drugoj mašini:

- Kada proces na mašini A pozove proceduru na mašini B, pozivni proces na mašini A se suspenduje, a izvršenje pozvane procedure se odvija na mašini B.
- parametri procedure se prenose kao mrežne poruke između pozivajućeg i pozvanog programa
  - » za programera nikakvo slanje poruka nije vidljivo
- Metod je nazvan Remote Procedure Call – RPC.
- Klijent-server model: klijent pristupa udaljenom servisu pozivanjem odgovarajuće procedure koja implementira taj servis

# RPC (nast.)

\* Ideja je jednostavna i elegantna, ali ima problema

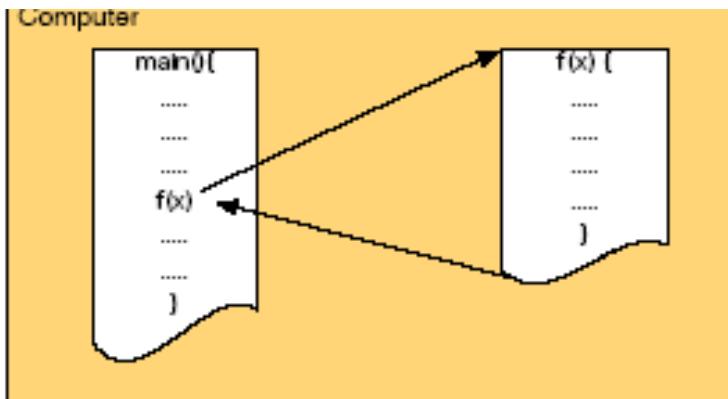
- Kako preneti parametre
  - Po vrednosti ili po referenci?
- Klijent i server mašine mogu biti različite
  - Koriste različite formate za predstavljanje podataka;
- Kako pronaći mašinu koja implementira udaljenu proceduru
- Koja semantika važi u slučaju da nastupi greška:
  - obe mašine mogu otkazati, što može uzrokovati različite probleme;

\* Većinu problema RPC može da reši i danas je to široko korišćena tehnika na kojoj su bazirani mnogi DS.

- SUN RPC, DCE RPC, MS DCOM, CORBA, Java RMI,... su RPC sistemi

# Kako RPC funkcioniše?

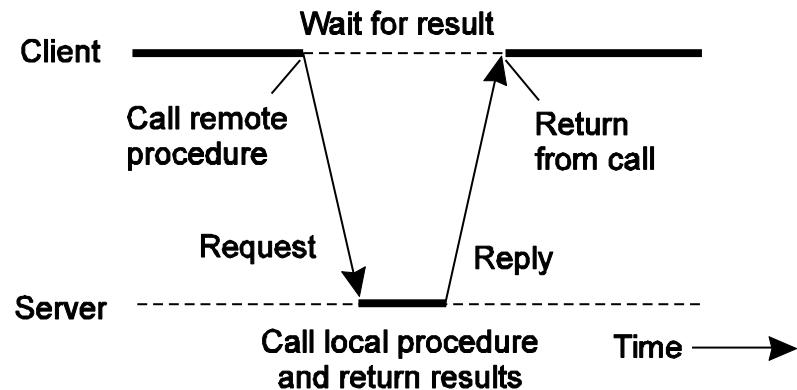
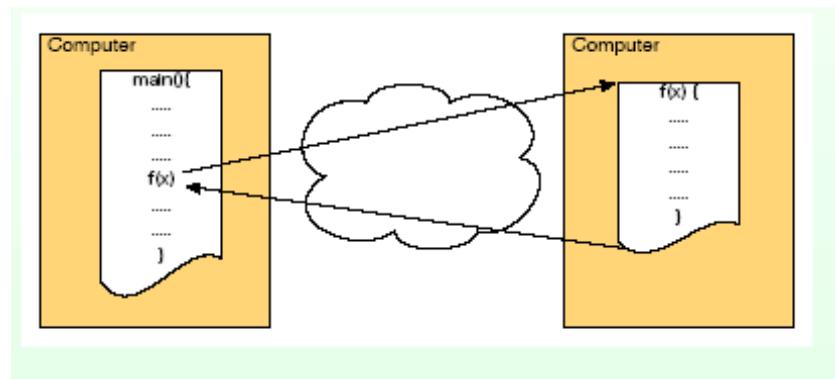
- Razmotrićemo prvo poziv konvencionalnih procedura, a zatim kako se poziv može razdeliti na klijentski i serverski deo koji se izvršavaju na različitim mašinama.



Poziv konvencionalne procedure:

- Nakon poziva procedure upravljanje se prenosi na pozvanu proceduru;
- na kraju izvršenja pozvana procedura vraća upravljanje glavnom programu.

I glavni program i pozvana procedura se izvršavaju na istoj mašini



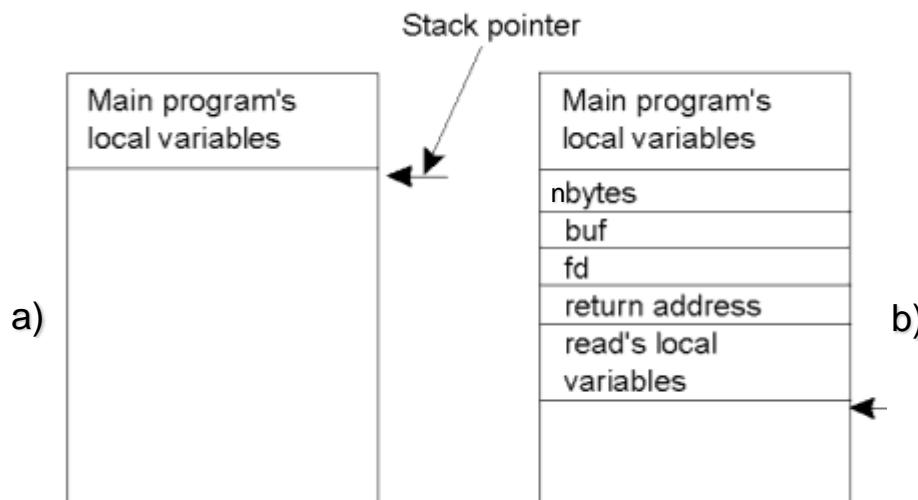
Poziv udaljene procedure

# Poziv konvencionalnih procedura – prenos parametara

## Primer

- count=read(fd, buf, nbytes);
- fd, je integer koji identificuje fajl
- buf, polje u koje se učitavaju karakteri
- nbytes, integer koji kaže koliko bajtova treba pročitati

- Ako je poziv upućen iz glavnog programa, onda gl. program upisuje parametre procedure u stek (u redosledu prvo poslednji kod C-a)



- a) Izgled steka pre poziva `read`;
- b) izgled steka dok je pozvana procedura aktivna

Kada se `read` okonča, rezultat se upisuje u registar i upravljanje vraća glavnom programu. Prenos parametara se može obaviti

- po vrednosti (call-by-value)
- po referenci (call-by-reference)
- call-by-copy/restore

# Prenos parametara

## \* Prenos po vrednosti

- Vrednosti parametara se kopiraju u stek (npr. fd i nbytes)
  - Za pozvanu proceduru ovi parametri predstavljaju inicijalne vrednosti lokalnih promenljivih
  - Pozvana procedura može modifikovati ove vrednosti, ali to ne utiče na originalne vrednosti na strani pozivaoca.

## \* Prenos po referenci

- U stek se upisuje adresa parametra, a ne vrednost
  - U pozivu *read* drugi parametar se prenosi po referenci, jer se polja uvek prenose po referenci u C-u
- Ako pozvana procedura modifikuje preneti parametar, promeniće se i originalna vrednost na strani pozivaoca

## \* Call by-copy/restore

- Kod poziva procedure parametri procedure se kopiraju u stek kao kod poziva po vrednosti
- Nakon okončanja poziva, vrednosti se upisuju preko originalnih vrednosti parametara, kao kod poziva po referenci.
  - Ada kompjajleri koriste copy/restore za in out parametre, a za ostale pozive po referenci.

## \* Koji način prenos parametara se koristi zavisi od programske jezike

- Nekada to zavisi i od tipa podataka koji se prenose
  - U C-u se skalarni tipovi prenose po vrednosti, a polja po referenci

# Copy/restore

```
* Int y;  
* Call_procedure( )  
* {  
*     y = 10;  
*     Copy-restore(y); // L-value of Y is passed.  
*     Printf y;      // prints 99  
* }  
* Copy-restore(x)  
* {  
*     x = 99;    // y still has the value of 10 (unaffected)  
*     y = 0;    // y is now 0  
* }
```

Pre poziva f-je:	Poziv f-je:	Izvršenje f-je:	Posle poziva f-je:
x	x=10	x=99	x=99
y=10	y=10	y=0	y=99

# RPC (nast.)

\* Osnovna ideja RPC je da poziv udaljene procedure izgleda što sličniji pozivu lokalne procedure.

- RPC obezbeđuje infrastrukturu neophodnu za transformisanje poziva procedure u poziv udaljene procedure na uniforman i transparentan način.
  - proces koji poziva udaljenu proceduru ne sme biti svestan da se pozvana procedura izvršava na drugoj mašini.

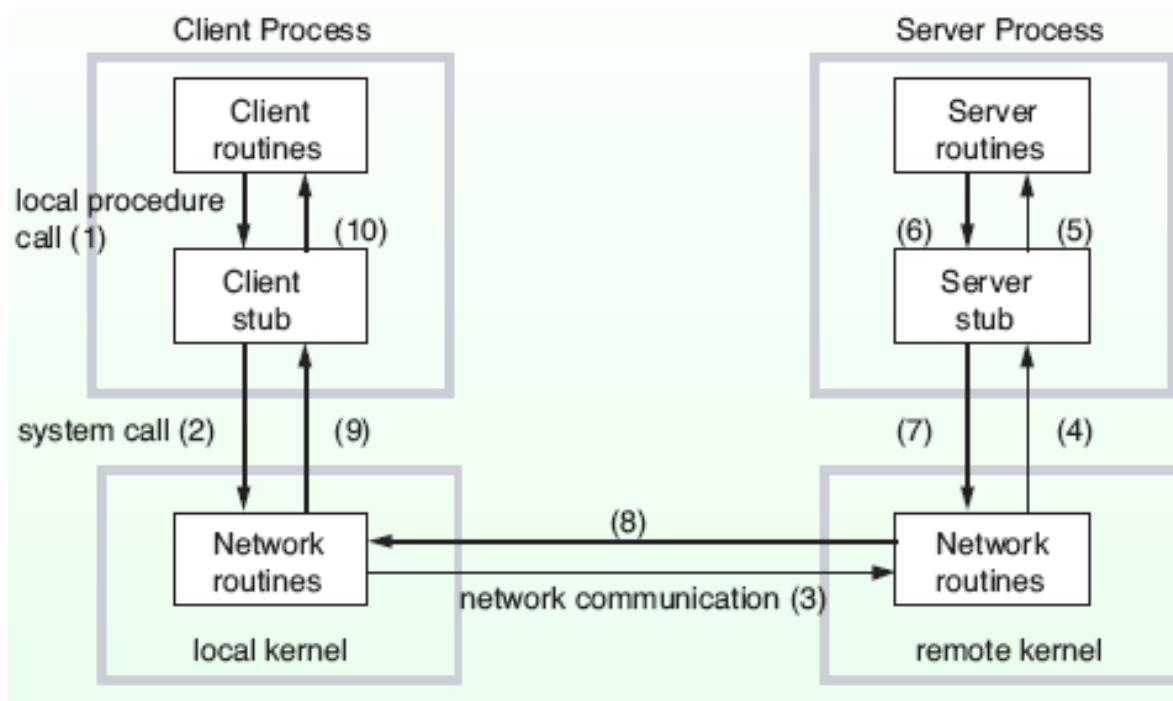
\* Primer

- Prepostavimo da program treba da pročita neke podatke iz fajla
  - Programer stavlja poziv *read* procedure u programske kod da bi pročitao podatke.
  - U jednoprocесорском систему *read* је библиотечка процедура и њен код се помоћу линкера убације у објекти код програма
    - То је kratка процедура која је имплементирана тако што се poziva еквивалентни системски poziv *READ*.
    - *read* процедура је једна врста интерфејса између корисниčког програма и локалног OS.
    - И ако *read* обавља системски poziv, она се pozива на уобичајени начин, смеštanjem параметара у стек (programer nije svestan da se u suštini obavlja sistemski poziv)

# Klijent i server "stub" (nast.)

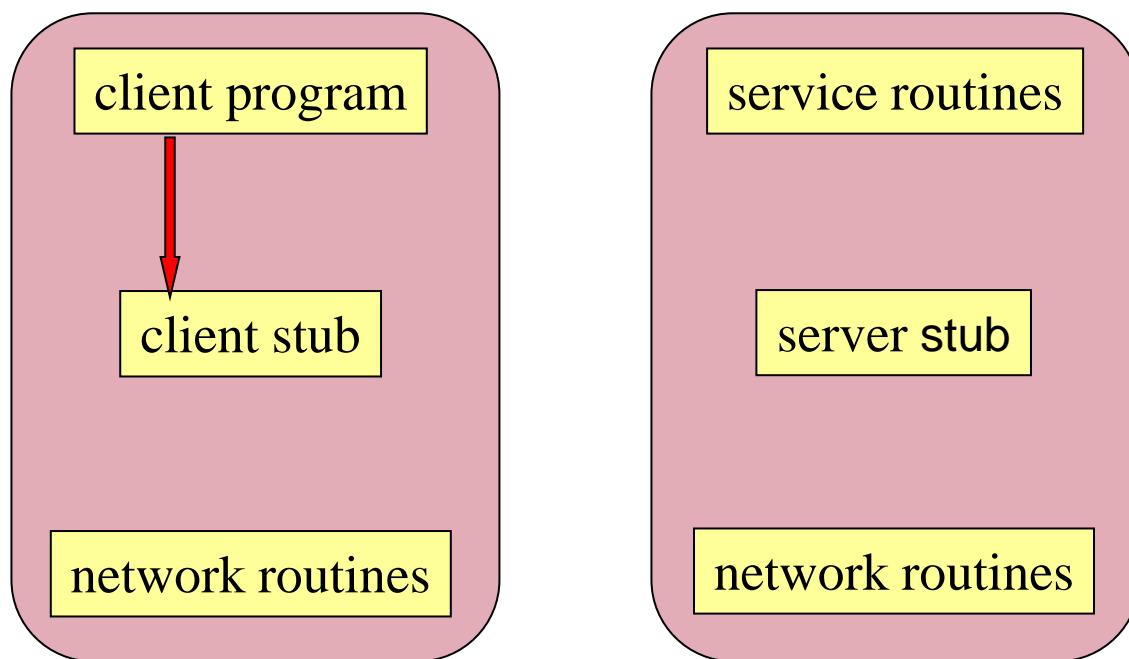
## \* RPC postiže transparentnost na sličan način

- Ako je *read* udaljena procedura (koja će se izvršiti na serveru) drugačija verzija *read*, koja se zove klijent stub se poziva koja na osnovu osnovu primljenih parametara gradi poruku i poziva lokalni OS da prenese poruku do udaljenog servera.
  - Stub funkcija izgleda kao funkcija koju korisnik želi da pozove, ali ona stvarno sadrži kod za slanje i prijem poruka kroz mrežu.



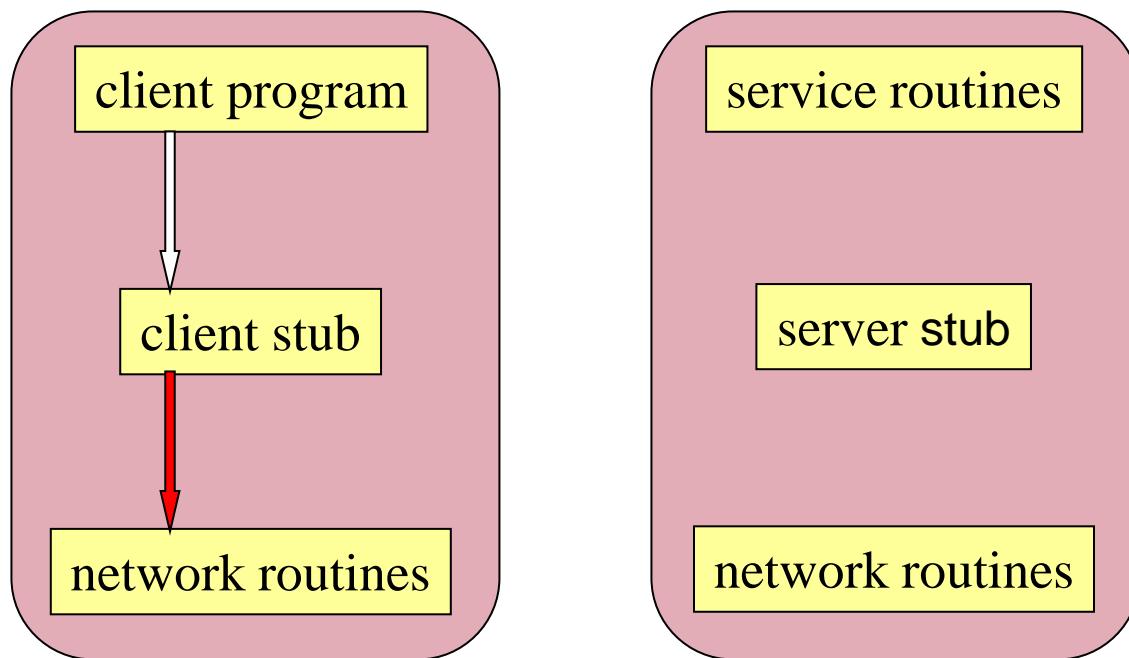
# Koraci kod poziva RPC

1. Klijent poziva lokalnu proceduru (klijent stub) (parametri se smeštaju u stek)



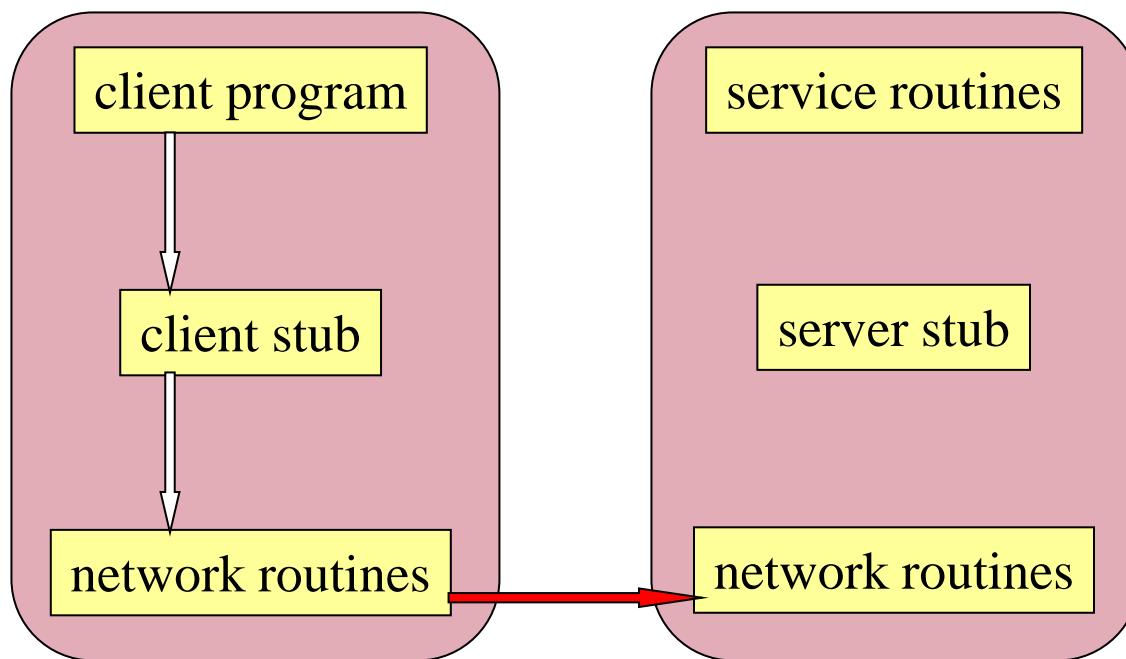
# Koraci kod poziva RPC

2. Klijent stub pakuje argumente (marshaling) za udaljenu proceduru (ova aktivnost može uključiti i konverziju u standardni format) i gradi jednu ili više mrežnih poruka (messages) i poziva lokalni OS (poziva *send* primitivu, a nakon toga i *receive* primitivu i čeka se dok ne stigne odgovor).



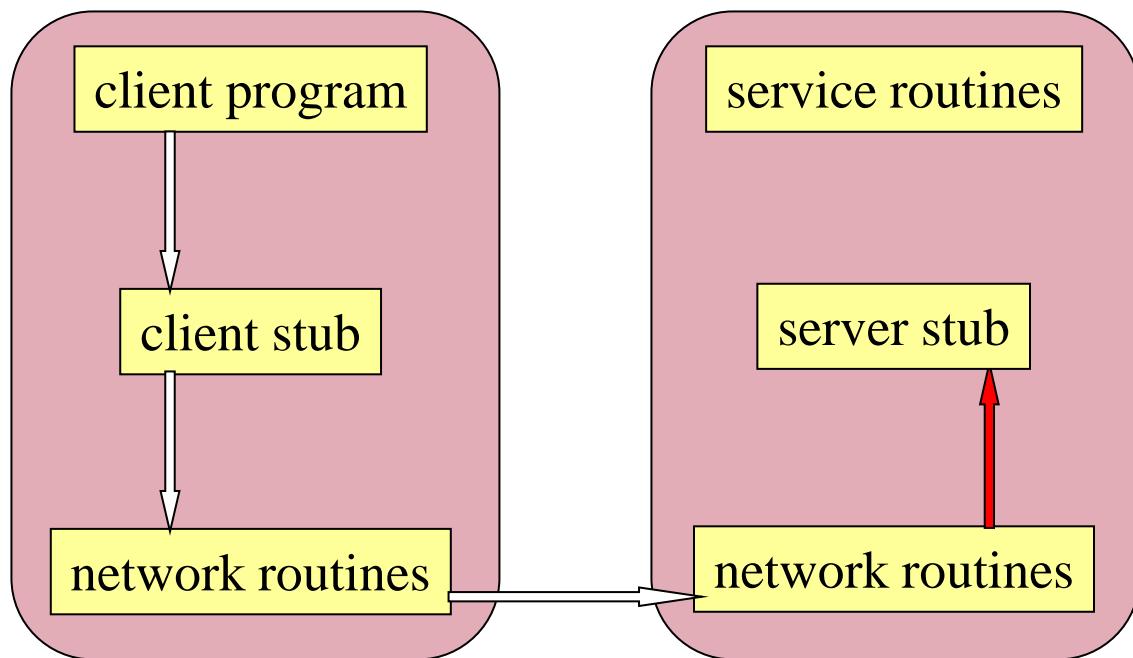
# Koraci kod poziva RPC

3. Lokalni OS šalje poruku udaljenom OS (korišćenjem transportnog protokola -TCP ili UDP )



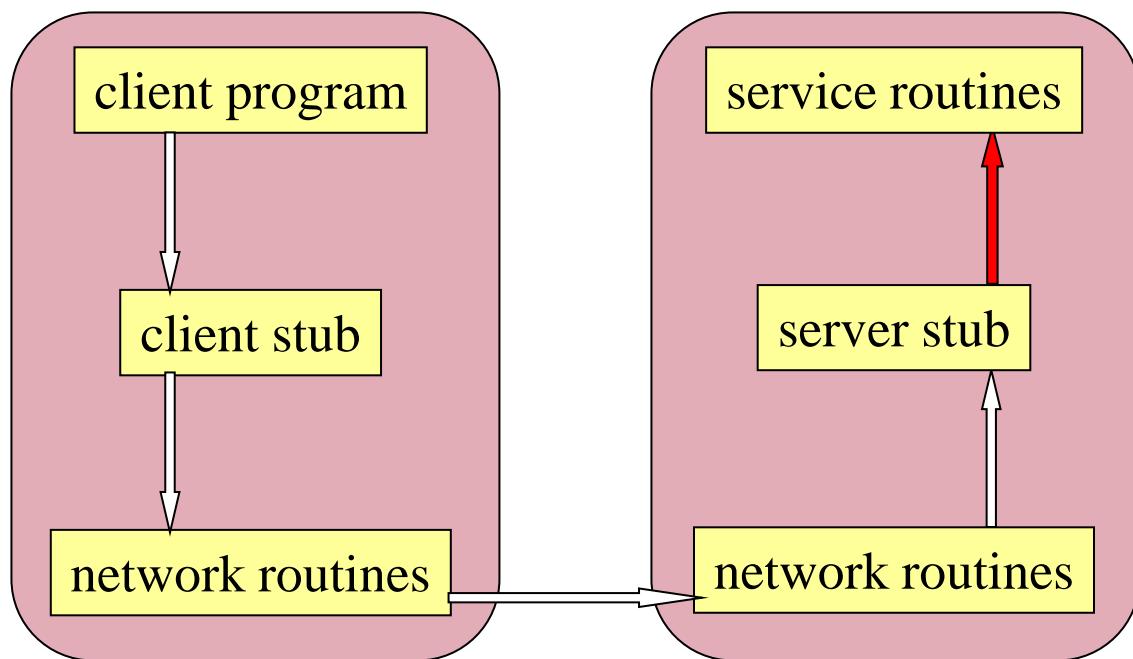
# Koraci kod poziva RPC

4. Prijem poruke: mrežna poruka stiže do odredišta , OS prosleđuje poruku serverskom stub-u (na osnovu broja odredišnog porta). Serverska stub funkcija prima poruku od lokalnog OS (izvršava receive primitivu), raspakuje je (unmarshaling) i izvlači argumente za poziv lokalne procedure



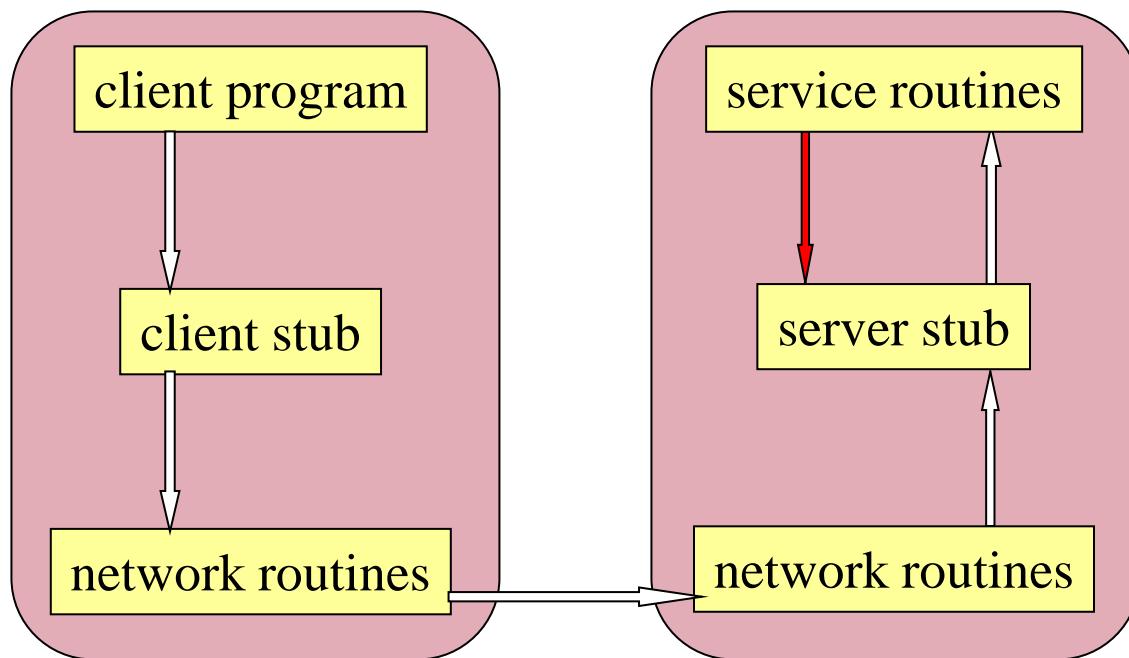
# Koraci kod poziva RPC

5. Serverski stub poziva željenu serversku proceduru i predaje joj parametre koje je primio od klijenta (stavljući ih u stek – kao kod poziva lokalne procedure).



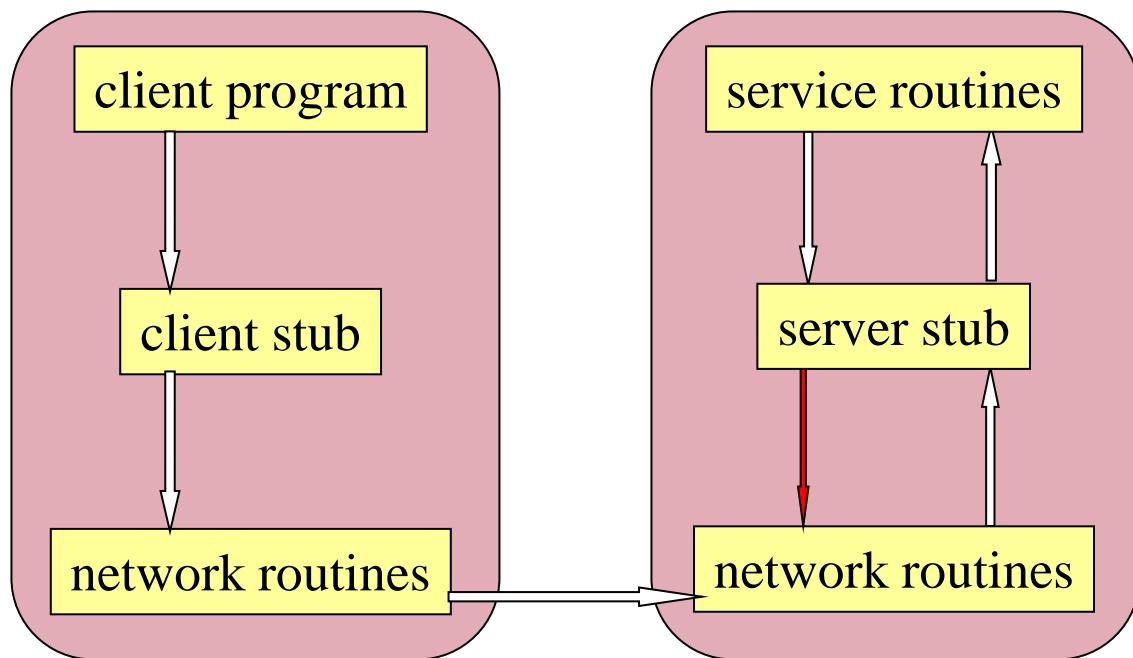
# Koraci kod poziva RPC

6. Server izvršava pozvanu proceduru i vraća rezultat stub-u.



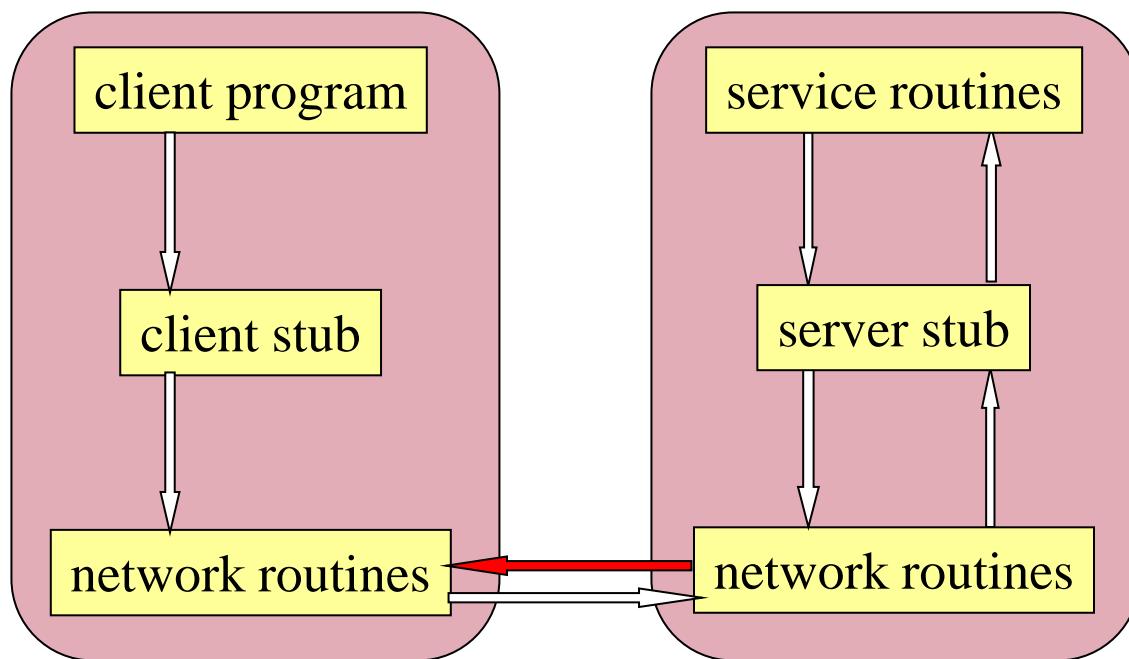
# Koraci kod poziva RPC

7. Serverski stub pakuje rezultat u poruku i poziva svoj lokalni OS (poziva *send* primitivu, a nakon toga ponovo *receive* primitivu čekajući novi zahtev).



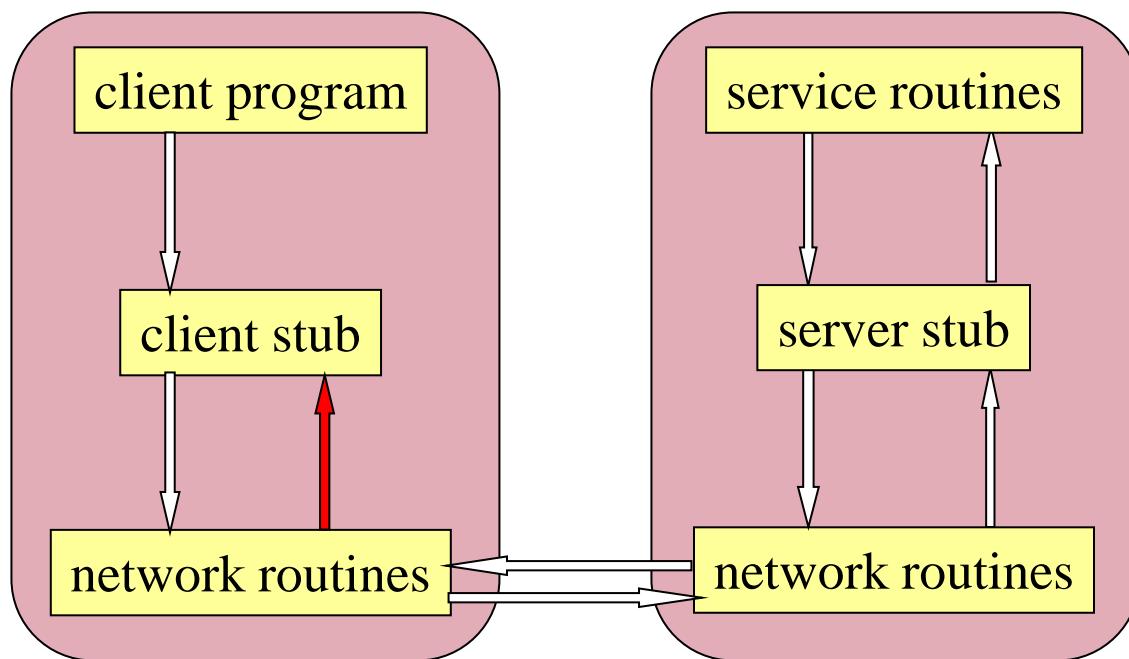
# Koraci kod poziva RPC

8. Slanje poruke kroz mrežu: Serverski OS šalje poruku klijentskom OS



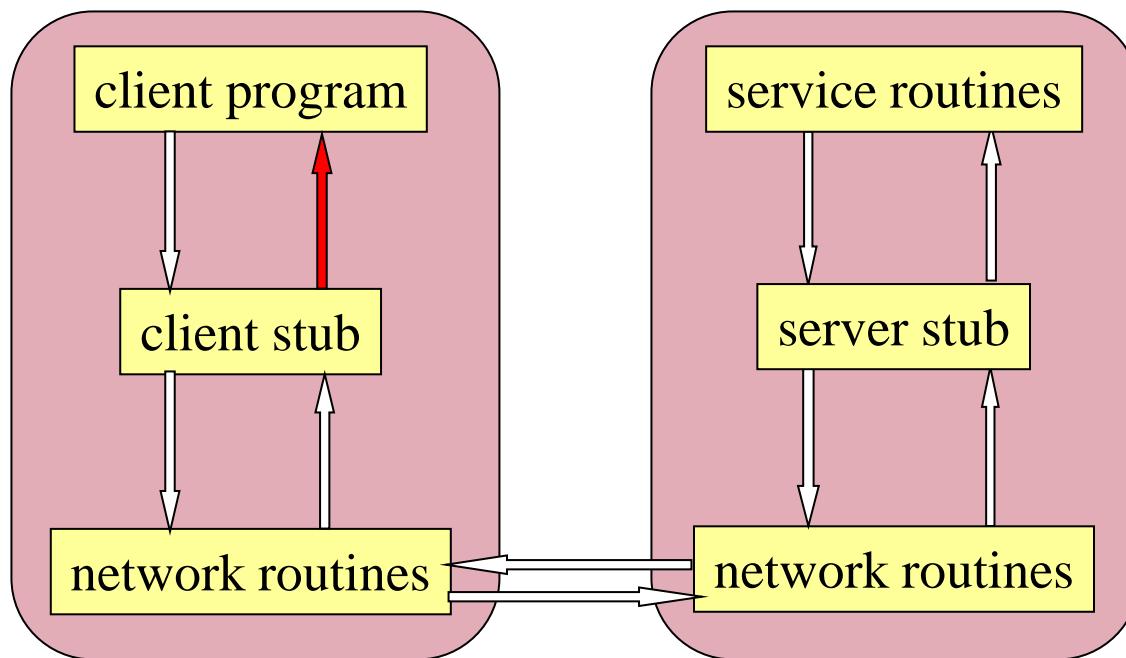
# Koraci kod poziva RPC

9. Lokalni OS prosleđuje poruku klijnt stub-u



# Koraci kod poziva RPC

10. Klijent stub izvlači rezultat iz poruke i vraća klijentskom procesu



# 10 koraka kod poziva RPC

1. Klijent poziva lokalnu proceduru (klijent stub)
  - Za klijentski proces ona izgleda kao aktuelna procedura koju treba pozvati.
2. Klijent stub pakuje argumente za udaljenu proceduru (ova aktivnost može uključiti i konverziju u standardni format) i gradi jednu ili više mrežnih poruka (messages) i poziva lokalni OS (poziva *send* primitivu, a nakon toga i *receive* primitivu i čeka se dok ne stigne odgovor).
  - Pakovanje argumenata u mrežnu poruku se zove "parameter marshaling" (uređivanje parametara)
3. Lokalni OS šalje poruku udaljenom OS (korišćenjem transportnog protokola)
4. Udaljeni OS prosleđuje poruku serverskom stub-u. Serverska stub funkcija prima poruku od lokalnog OS (izvršava *receive* primitivu), raspakuje je (unmarshaling) i izvlači argumente za poziv procedure.
5. Serverski stub poziva željenu serversku proceduru i predaje joj parametre koje je primio od klijenta (stavljaći ih u stek – kao kod poziva lokalne procedure).
6. Server izvršava pozvanu proceduru i vraća rezultat stub-u.
7. Serverski stub pakuje rezultat u poruku i poziva svoj lokalni OS (poziva *send* primitivu, a nakon toga ponovo *receive* primitivu čekajući novi zahtev).
8. Serverski OS šalje poruku klijentskom OS
9. Lokalni OS prosleđuje poruku klijent stub-u
10. Klijent stub izvlači rezultat iz poruke i vraća klijentskom procesu
  - Klijent nastavlja dalje sa izvršenjem

- \* Udaljenim servisima se pristupa uobičajenim pozivima procedura, a ne pozivanjem *send* i *receive*.
- \* Svi detalji prosleđivanja poruka su skriveni u klijentskim i serververskim stub funkcijama, kao što su detalji stvarnih sistemskih poziva skriveni u bibliotečkim funkcijama.
  - Stub funkcije se kreiraju za svaku udaljenu proceduru

# Prenos parametara kod RPC

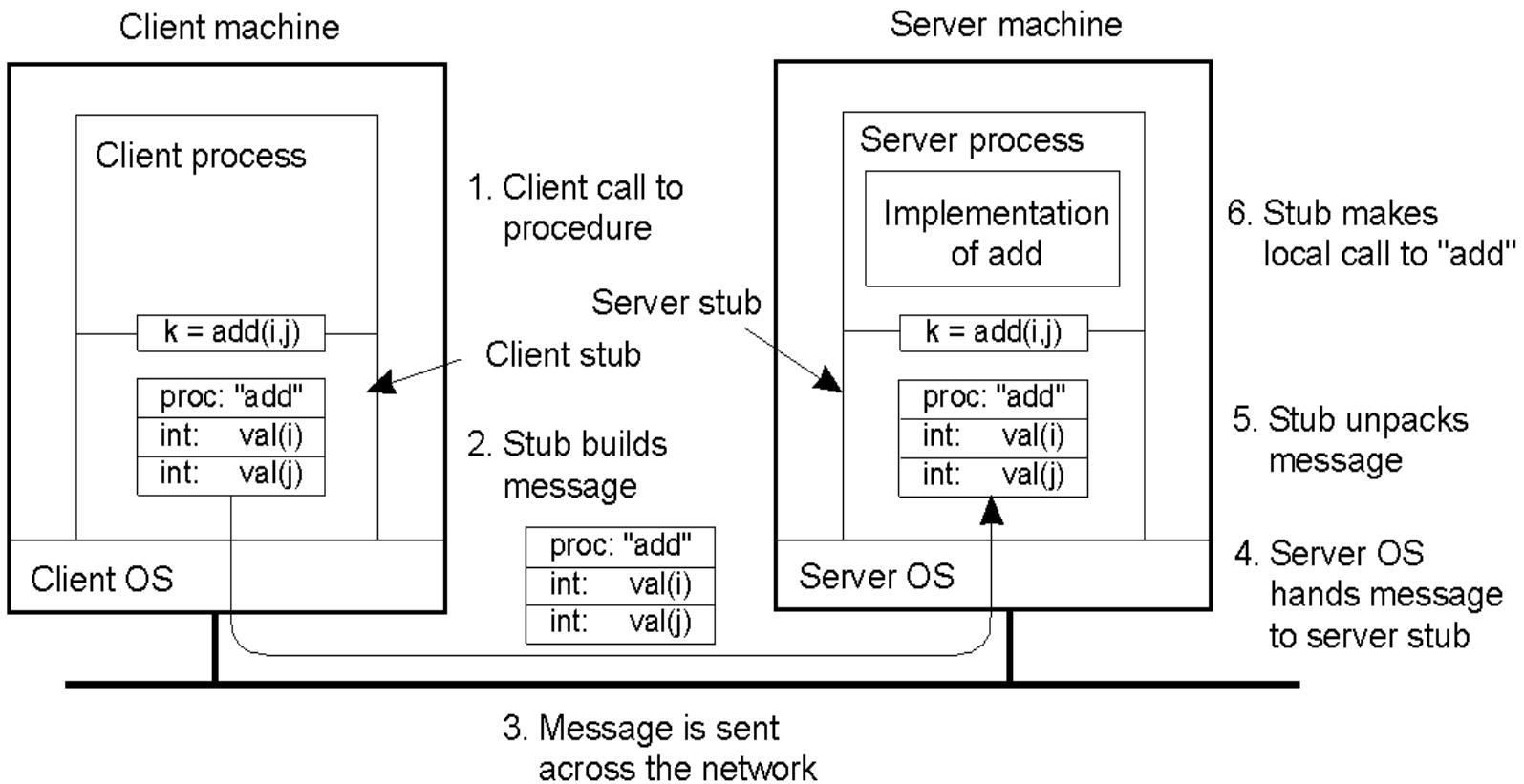
- Funkcija klijentskog stub-a je da preuzme parametre procedure od klijenta, spakuje ih u poruku (parameter marshaling) i pošalje serverskom stub-u.

## \* Kako se prenose parametri?

- Prenos po vrednosti je jednostavan
  - vrednosti se jednostavno kopiraju u mrežne poruke

## \* Primer

- Neka je add(i,j) udaljena procedura koja ima dva parametra tipa integer i kao rezultat vraća zbir dva integera.
- Poziv udaljene add će prihvati klijent stub
- Klijentski stub uzima parametre i stavlja ih u poruku
- U poruku stavlja i ime ili broj procedure koja se poziva (add) jer server može podržati više poziva udaljenih procedura
- Kada poruka stigne do servera , serverski stub ispituje poruku da bi ustanovio koja procedura se poziva, i zatim upućuje odgovarajući poziv
  - Poziv koji stub upućuje serveru je sličan originalnom klijentovom pozivu, osim što su parametri promenljive koje se inicijalizuju iz dolazne poruke.
- **Kada se procedura izvrši, serverski stub ponovo preuzima kontrolu:**
  - Prihvata rezultat od izvršenja procedure, pakuje ga u poruku, šalje poruku klijentu
  - Klijentski stub raspakuje poruku i šalje rezultat klijentskoj proceduri.



ã Sve dok su klijent i server mašina identične i dok su parametri skalarnog tipa ovo dobro funkcioniše

ã Međutim, u DS je uobičajeno da postoji više tipova mašina:

I svaka mašina može imati različit način predstavljanja brojeva (big-endian, little-endian, dvoični ili jedinični komplemet,...), karaktera (EBCDIC, ASCII)

I Nije moguće preneti parametre procedure kao što je opisano

I Problem se najčešće rešava tako koristi "standardno" kodiranje svih tipova podataka koji se mogu prenositi kao parametri.

IKlijentski i serverski stub moraju obavljati konverzije iz lokalnog u standardni (i obrnuto) način predstavljanja podataka.

# Prenos parametara

- Prenos po referenci je teško implementirati

- Nema nikakvog smisla proslediti adresu udaljenoj mašini.
- Ako se želi postići efekat prenosa po referenci, neophodno je iskopirati parametar (npr. polje) u poruku i poslati je serveru.
- Serverski stub može pozvati serversku proceduru sa pointerom na podatak (npr. polje) koji je primljen i nalazi se u baferu ali će se vrednost pointera razlikovati od vrednosti na klijent strani.
- Modifikacije podataka koje serverska procedura uradi će direktno preko pointera uticati na sadržaj bafera u server stub-u.
- Kada server okonča izvršenje procedure, poruka će biti vraćena klijent stub-u, koji je zatim kopira na klijenta
- U suštini je poziv po referenci zamenjen pozivom copy/restore.

# Kako locirati udaljeni server i proceduru?

- \* Da bi se realizovao poziv udaljene procedure neophodno je locirati udaljeni host i odgovarajući proces na hostu
  - Povezivanje (binding).
- \* Dva rešenja su moguća
  - Statičko povezivanje: Klijent zna koji host treba da kontaktira (adresa hosta se nalazi u klijent stub-u), a kada klijent pozove odgovarajuću proceduru, klijent stub jednostavno prosledi poziv serveru.
    - Poseban program (portmapper) na udaljenom hostu pamti preslikavanja imena programa i broja verzije u broj porta
  - Prednosti ovakvog rešenja:
    - Jednostavnost i efikasnost, nema potrebe za dodatnom infrastrukturom osim klijent i server stub-a
  - Nedostaci
    - Klijent i server postaju čvrsto povezani (ako server otkaže, klijent neće moći da radi)
    - Ako server promeni lokaciju (IP adresu) klijent mora da se rekompilira sa novim stub-om koji ukazuje na pravu lokaciju

# Kako locirati udaljeni server i proceduru? (nast.)

## \* Dinamičko povezivanje

- Drugo rešenje je da postoji centralizovana baza podataka (smeštena u name i directory serverima) koja može locirati host koji obezbeđuje željeni servis (rešenje koje su predložili Birell i Nelson)
- Ovi serveri imena i direktorijuma vraćaju adresu servera na osnovu "potpisa" procedure (imena i parametara) koja se poziva.
  - Kada klijent pozove udaljenu proceduru, klijentski stub kontaktira server imena da bi dobio adresu servera koji tu proceduru izvršava.
  - Server imena šalje adresu klijent stub-u koji zatim uspostavlja vezu sa željenim serverom.
- Ako server koji implementira željenu proceduru promeni adresu, dovoljno je promeniti ulaz u serveru imena
- Serverski stub registruje serversku proceduru u serveru imena i direktorijuma prilikom stratovanja servera

# Problemi

\* kod poziva konvencionalne (lokalne) procedure ona će se izvršiti tačno jednom

- ako nastupi greška ceo proces će biti uništen

\* kod poziva udaljene procedure postoji više mogućnosti za nastupanje grešaka

- server može generisati grešku
- mogu nastupiti problemi u mreži
- server može otkazati
- klijent može otkazati dok server izvršava pozvanu proceduru

\* Transparentnost se ovde završava!

- aplikacija mora biti pripremljena za ove probleme (testirati greške)

# semantika poziva udaljenih procedura

## \* kod poziva lokalnih procedura semantika je jednostavna

- procedura će se izvršiti tačno jednom kada je pozvana

## \* udaljena procedura može se izvršiti

- 0 puta, ako server otkaže pre izvršenja serverskog koda
- jednom, ako je sve ok
- jednom ili više puta, ako je komunikaciono kašnjenje veliko ili se izgubi odgovor od strane servera pa klijent izvrši retransmisiju

## \* Većina RPC sistema obezbedjuje

### • “bar jednom” semantiku

- ako su u pitanju idenpotentne funkcije (mogu se izvršavati bez posledica više puta, npr. datum, vreme, čitanje statičkih podataka)

### • “samo jednom” semantiku

- funkcija nije idenpotentna (poziv funkcije generiše efekte, npr. modifikovanje fajla)

# Programiranje sa RPC

- Većina programskih jezika (C, C++, Java,...) ne podržavaju koncept udaljenih procedura i nemaju mogućnost da generišu klijent i server stub funkcije.
- Da bi se omogućilo korišćenje poziva udaljenih procedura iz ovih jezika, koristi se poseban kompjuler koji generiše klijent i server stub funkcije na osnovu interfejsa procedura.
  - U klijent-server sistemu baziranom na RPC, lepak koji omogućava da se sve poveže u jedinstvenu celinu je definicija interfeisa napisana na IDL (Interface Definition Language).
- Definicija interfeisa je slična deklaraciji prototipa funkcije :
  - Ona sadrži set funkcija zajedno sa ulaznim parametrima i vrednostima koje se vraćaju.
- Kompajliranjem interfeisa pomoću RPC kompjulera generišu se klijentski serverski stub-ovi
- Nakon što se obavi kompajliranje uz pomoć IDL kompjulera, mogu se kompajlirati klijentski i serverski programi i povezati (linkovati) sa odgovarajućim stub funkcijama.

# Sun RPC

\* Inicijalno projektovan za Sun OS, ali je danas raspoloživ za veliki broj platformi

- Sun RPC sistem obezbeđuje jezik za definisanje interfeisa (XDR – eXternal Data Representation) i interfeis kompjajler (rpcgen) koji se koristi za generisanje klijent i server stub f-ja.
- rpcgen kompjajler generiše tri fajla (npr rpcgen primer.x)
  - header fajl (npr. primer.h)
    - Header fajl sadrži jedinstveni identifikator interfeisa, definiciju tipova, konstanti i prototipova funkcija.
    - On treba da bude uključen (korišćenjem #include) i u klijent i u server kod
  - klijent stub (**primer\_clnt.c**)
    - Klijent stub sadrži procedure koje će klijent program pozivati
    - Ove procedure su zadužene za pakovanje parametara u poruke, slanje poruka, prijem poruka, izvalačenje rezultata poziva procedure i prosleđivanje klijentu
  - server stub (**primer\_svc.c**) –
    - Sadrži procedure koje se pozivaju kada stigne poruka do servera i koje zatim pozivaju odgovarajuću serversku proceduru

# Koraci kompilacije za RPC

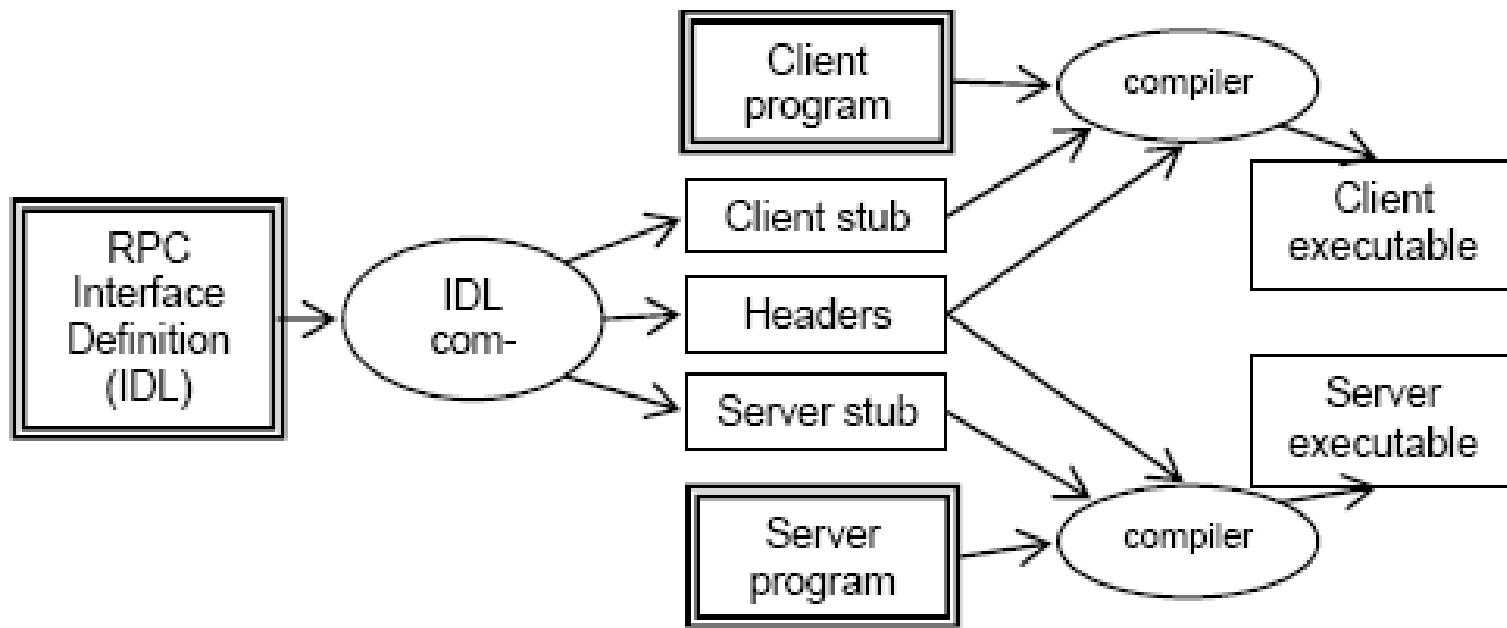


Figure 2. Compilation steps for Remote Procedure Calls

# definicija interfeisa

- \* svaka RPC procedura je definisana u okviru nekog programa i identifikovana je na jedinstveni način
  - brojem programa – ASCII string predstavljen u Hex notaciji
    - identificuje grupu udaljenih procedura, od kojih svaka ima svoj broj
  - brojem verzije (programa)
    - svaki program ima i broj verzije, tako da ako se napravi izmena u udaljenom servisu (programu), npr. dodavanje nove procedure, programu se daje novi broj verzije, a ne novi broj programa
  - brojem procedure
    - numeracija najčešće kreće od 1

## \* PRIMER

```
PROGRAM ime_programa {  
    def_verzije  
    def_verzije  
}= 0xYYYYYYYY  
32-bitni identifikator
```

0x00000000-0x1fffffff: defined by sun  
0x20000000-0x3fffffff defined by the user  
0x40000000-0x5fffffff transient processes  
0x60000000-0x7fffffff reserved

```
VERSION ime_verzije {  
    def_procedure1;  
    def_procedure2;  
    ...  
}=konstanta
```

### definicija procedure

```
tip ime_procedure (tip_argumenta)  
=konstanta;
```

# Primer: poziv lokalne procedure

```
#include <stdio.h>
int saberi();
int oduzmi();
main(int argc, char *argv)
{
    int a,b, rez1, rez2;
    if (argc!=3){
        poruka o gresci; exit(1);
    }
    a=argv[1]; b=argv[2];
    rez1=saberi(a,b);
    rez2=oduzmi(a,b);
    printf("zbir je=%d, razlika je = %d\n", rez1, rez2);
    exit(0);
}
int saberi( x, y){
    int x, y;
    return (x+y);
}
int oduzmi(x,y){
    int x, y;
    return (x-y);
}
```

# Primer: Definicija interfeisa za pristup udaljenom servisu SABOD ( Sun RPC)

\* Sve udaljene procedure se deklarišu kao deo udaljenog programa

- PRIMER: srevis KALK za sabiranje i oduzimanje dva cela broja, ima dve procedure:
  - SABERI (x,y)
  - ODUZMI (x,y)
- Kod Sun RPC poziv udaljene procedure može imati samo jedan argument

```
/* definicije tipova koji se prosledjuju udaljenim procedurama SABER i ODUZMI */  
struct operandi {  
    int x;  
    int y;  
};  
/*definicija procedura i identifikatora program KALK {  
    version KALK_VERSION {  
        int SABERI(operandi) = 1;  
        int ODUZMi(operandi) = 2;  
    }=1  
}=0x29999999  
*/
```

ime verzije

broj verzije

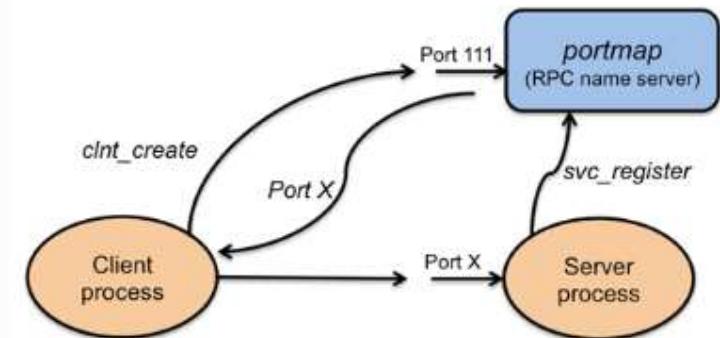
brojevi procedura

broj programa

- ime programa, ime verzije, imena procedura u IDL fajlu se po pravilu pišu velikim slovima
- udaljene procedure koje su definisane pomoću IDL se u klijent programu pozivaju malim slovima navodnjem imena procedure iza kojeg sledi donja crtica (\_) i broj verzije
  - (imeprocedure\_brojverzije)
- ove procedure u server programu imaju ime
  - imeprocedure\_brojverzije\_svc (malim slovima)
- Nakon komajliranja IDL fajla komandom
  - rpcgen –C primer.x
- dobijaju se sldeći fajlovi
  - primer.h (header fajl)
  - primer\_clnt.c (klijent stub)
  - primer\_svc.c (server stub)

# klijent strana

```
#include<stdio.h>
#include<rpc/rpc.h> /*standardna biblioteka za RPC*/
#include "primer.h" /* fajl generisan pomocu rpcgen*/
int main(int argc, char *argv[])
{
CLIENT *cln; /* RPC handle */
char *server;
int *zbir, *razlika; /* povratne vrednosti iz udaljenih procedura - operandi op; /*struktura definisana u IDL fajlu*/
if (argc<2) {poruka o gresci; exit(1)}
server=argv[1];
cln =clnt_create(server, KALK, KALK_VERSION, "udp") /*kreira se socket i klient handle i povezuje sa serverom*/
if (cln==NULL) {poruka o gresci exit(2)} /*konekcija sa serverom nije uspostavljena */
op.x=atoi(argv[2]); op.y=atoi(argv[3]);
zbir=saberi_1(&op, cln); /*poziv udaljene procedure (u suštini poziva se stub funkcija)*/
if (zbir==NULL){poruka o gresci; exit(3)}
printf("suma je %d/n", *zbir)
razlika=oduzmi_1(&op, cln) /*poziv udaljene procedure; argument procedure je pointer na argument čiji je tip definisan u IDL fajlu; procedura mora vratiti pointer na rezultat */
if (razlika==NULL){poruka o gresci; exit(4)}
printf("razlika je %d/n", *razlika)
clnt_destroy(cln); /* zatvara se konekcija sa serverom */
exit(0);
}
```



# server strana

\* pomoću rpcgen može se generisati template za serverski kod korišćenjem prethodno definisanog interfeisa (u našem primeru primer.x)

- rpcgen -C -Ss primer.x >server.c

➤ server.c je ime fajla u kome je zapamćen serverski kod

- rpcgen će generisati sledeći kod

```
#include primer.h
int *saberi_1_svc(operandi *argp, struct svc_req *rqstp)
{
    static int result;
    /*ubaciti serverski kod ovde*/
    return &result
}
int *oduzmi_1_svc(operandi *argp, struct svc_req *rqstp)
{
    static int result;
    /*ubaciti serverski kod ovde*/
    return &result
}
```

# server strana

```
#include<stdio.h>
#include<rpc/rpc.h> /*standardna biblioteka za rpc */
#include "primer.h" /* fajl generisan pomocu rpcgen*/
int *saberi_1_svc(operandi *a, struct svc_req *rqstp)
{
static int zbir;
zbir=a->x+a->y;
return &zbir;
}
int *oduzmi_1_svc(operandi *a, struct svc_req *rqstp)
{
static int razlika;
razlika=a->x-a->y;
return &razlika;
}
```

# povezivanje klijenta i servera kod Sun RPC

- \* klijent mora znati ime udaljenog servera
- \* server registruje svoje usluge preko portmapper deamon procesa na serverskoj mašini (uvek na portu 111).
- \* kompajliranje – linkovanje- izvršenje
  - generisanje stub funkcija
    - rpcgen -C primer.x
  - kompajliranje i linkovanje klijenta i klijent stub
    - cc -o klijent klijent.c primer\_clnt.c
  - kompajliranje i linkovanje servera i server stub
    - cc -o server server.c primer\_svc.c
  - Izvršenje
  - pozvati serverski program na izvršenje (recimo na hostu remus)
    - \$ ./server
  - pozvati klijent program
    - \$ ./klijent -h remus 12 15

# SUN RPC prednosti

- \* Aplikacija ne mora da vodi računa o dobijanju jedinstvene transportne adrese (broja porta)
  - Kod SUN RPC potreban je jedinstveni broj programa po serveru
  - Veća portabilnost
- \* Aplikacija ne mora da vodi računa o veličini poruke, fragmentaciji, reasembliranju
- \* Aplikacija treba da zna samo jednu transportnu adresu – adresu port mappera

# Primer2: Distribuirano računarsko okruženje (DCE)

- \* Distributed Computing Environment (DCE) – distribuirano računarsko okruženje, je middleware baziran na RPC.
  - Open Software Foundation (OSF), danas Open Group (OG)
  - to je skup servisa i alata koji se može instalirati na vrhu postojećeg OS, koji služi kao platforma za gradnju distribuiranih aplikacija
- \* DCE je prvi middleware sistem koji je projektovan kao nivo apstrakcije između mrežnog OS i distribuirane aplikacije.
  - Inicijalno je projektovan za UNIX, ali sada postoji za sve veće OS (VMS, desktop OS)
  - Ideja je bila da korisnik može na postojeći skup računara dodati DCE sotver i nakon toga biti u stanju da izvršava distr. appl. ne narušavajući postojeće (nedistribuirane apl.)
    - Većina DCE sw se izvršavaju korisničkom prostoru
      - Distribuirani fajl sistem mora biti dodat u kernel

# DCE (nast.)

## \* Programski model na kome se baziran DCE je klijent-server.

- Korisnički proces se ponaša kao klijent da bi pristupio udaljenom servisu koji pruža serverski proces.
- Sva komunikacija između klijenta i servera se odvija pomoću RPC

## \* DCE servisi

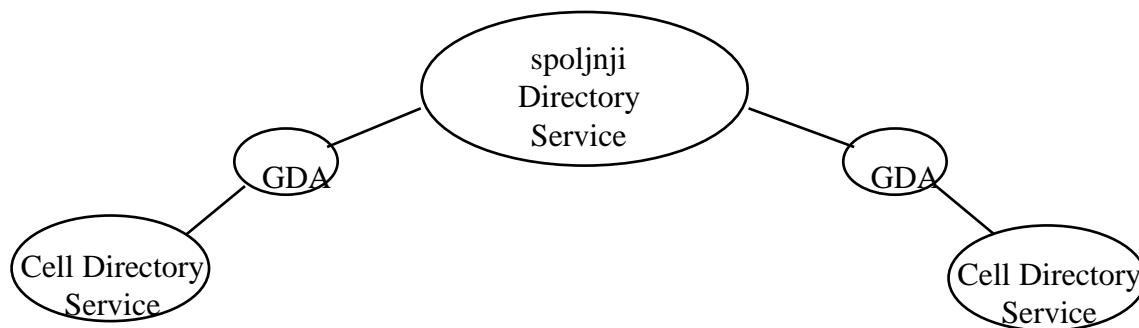
- RPC
- Direktorijumski servis
- Distribuirani fajl servis
- Bezbednosni servis
- Servis distribuiranog vremena

# DCE RPC

- \* fundamentalni komunikacioni mehanizam
  - svi ostali servisi koriste RPC za implementaciju svojih usluga
- \* omogućava direktno pozivanje procedura na udaljenim sistemima, kao da su u pitanju lokalne procedure
- \* pojednostavljuje pisanje distribuiranih aplikacija, elimišući potrebu za eksplicitnim programiranjem mrežne komunikacije izmedju klijenta i servera.
- \* skriva razlike u predstavljanju podataka na različitim hardverskim platformama i omogućva distribuiranim programima da se izvršavaju na heterogenim sistemima

# Direktorijumski servis

- \* Servis koji na osnovu imena objekta vraća informaciju o imenovanom.
- \* Npr:
  - ime osobe => broj tel. osobe
  - ime hosta => informacija o host (npr. IP adresa)
  - ime RPC servera => binding informacija za dati server
- \* Efikasan direktorijumski servis je ključan u distribuiranom okruženju.
- \* DCE Cell Directory Servis (CDS) je mehanizam koji omogućava korišćenje logičkih imena unutar DCE celije (grupa klijent i server mašina najčešće u okviru LAN)
- \* aplikacije identifikuju resurse po imenu, bez potrebe da znaju gde je resurs lociran (za razliku od Sun RPC)
  - ovaj servis se koristi za lociranje servera na kome je implementirana udaljena procedura
- \* DCE celije mogu medjusobno komunicirati da bi locirali resurs koji je van lokalne DCE celije (slično kao što funkcioniše DNS)
- \* integrisan u druge DCE servise i drugi servisi ga koriste
- \* Zove se još i servis imena (*Name Service*)



# Bezbednosni servis

\* Omogućava da resursi svih vrsta budu zaštićeni tako da pristup bude dozvoljen samo autorizovanim korisnicima

- omogućava procesima na različitim mašinama da utvrde identitet onog drugog procesa (autentifikacija)
- omogućava serveru da utvrdi da li je korisniku dozvoljeno da pristpi odredjenom resursu (autorizacija)

# distribuiran fajl servis

- \* DCE distribuirani fajl servis (DFS) pruža bezbedan metod za deljenje udaljenih fajlova
- \* DFS korisniku izgleda kao lokalni fajl sistem, obezbeđuje pristup fajlovima sa bilo koje lokacije u mreži korišćenjem istog imena (uniformni pristup fajlovima)
- \* DFS pruža mnoge usluge koje tradicionalni distribuirani fajl servisi ne pružaju, kao npr. keširanje, bezbednost, skalabilnost.

## servis distribuiranog vremena

obezbeđuje mehanizme za sinhronizaciju časovnika na različitim mašinama u distribuiranom sistemu

➤ olakšva postizanje konzistentnosti u DS

# DCE (nast.)

U klijent-server sistemu baziranom na RPC lepak koji sve drži na okupu je definicija interfejsa specificirana u IDLu.

- IDL omogućava deklaraciju procedura slično deklaraciji prototipa funkcija u C-u
- IDL fajl može sadržati definicije tipova, deklaracije konstanti, deklaracije procedura i druge informacije potrebne za korektno pakovanje i raspakovanje parametara.
- Ključni element u svakom IDL fajlu je globalno jedinstveni identifikator interfeisa.
  - Klijent šalje ovaj identifikator u prvoj RPC poruci i kontaktirani server proverava da li takav identifikator (interfeis) postoji

# DCE (nast.)

\* Prvi koraku pisanju klijent/server aplikacije je obično poziv **uuidgen** (universally unique identifier ) programa od koga se traži da generiše prototip IDL fajla koji sadrži identifikator interfejsa i garantuje da se isti identifikator neće nigde pojaviti u DS generisanim sa **uuidgen**.

- Jedinstvenost se postiže kodiranjem i lokacije i trenutka kreiranja.
  - Identifikator je 128-bitni broj predstavljen u IDL fajlu kao ASCII string u heksadecimalnoj notaciji
- NPR.
  - `uuidgen imefajla.idl` (`uuidgen primer.idl`)
  - kreira se fajl `primer.idl` koji sadrži sledeće
    - [  
    `uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd),`  
    `version(1.0)`  
]
    - `interface INTERFACENAME`
    - {
    - }

# DCE (nast.)

- \* Sledeći korak je editovanje IDL fajla, tj. popunjavanje imena udaljenih procedura i njihovih parametara
- \* Npr.

- sada treba zameniti INTERFACENAME imenom interfeisa
  - Npr.
  - [  
    uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd),  
    version(1.0)  
]  
interface math  
{  
  
}

- \* sada treba deklarisati operacije (procedure) u okviru interfeisa

- \* Npr

- [uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd), version(1.0) ]

```
interface math{
```

```
    long get_sum([in] long first, [in] long second);
```

```
}
```

# semantika poziva procedura

\* DCE podržava dve semantčke opcije kod poziva udaljene procedure

- “bar jednom” semantiku

- ako su u pitanju idenpotentne funkcije (mogu se izvršavati bez posledica više puta)
- takve procedure se moraju označiti kao idempotent u IDL –u

- “samo jednom” semantiku

- funkcija nije idenpotentna (poziv funkcije generiše efekte, npr. modifikovanje podataka)

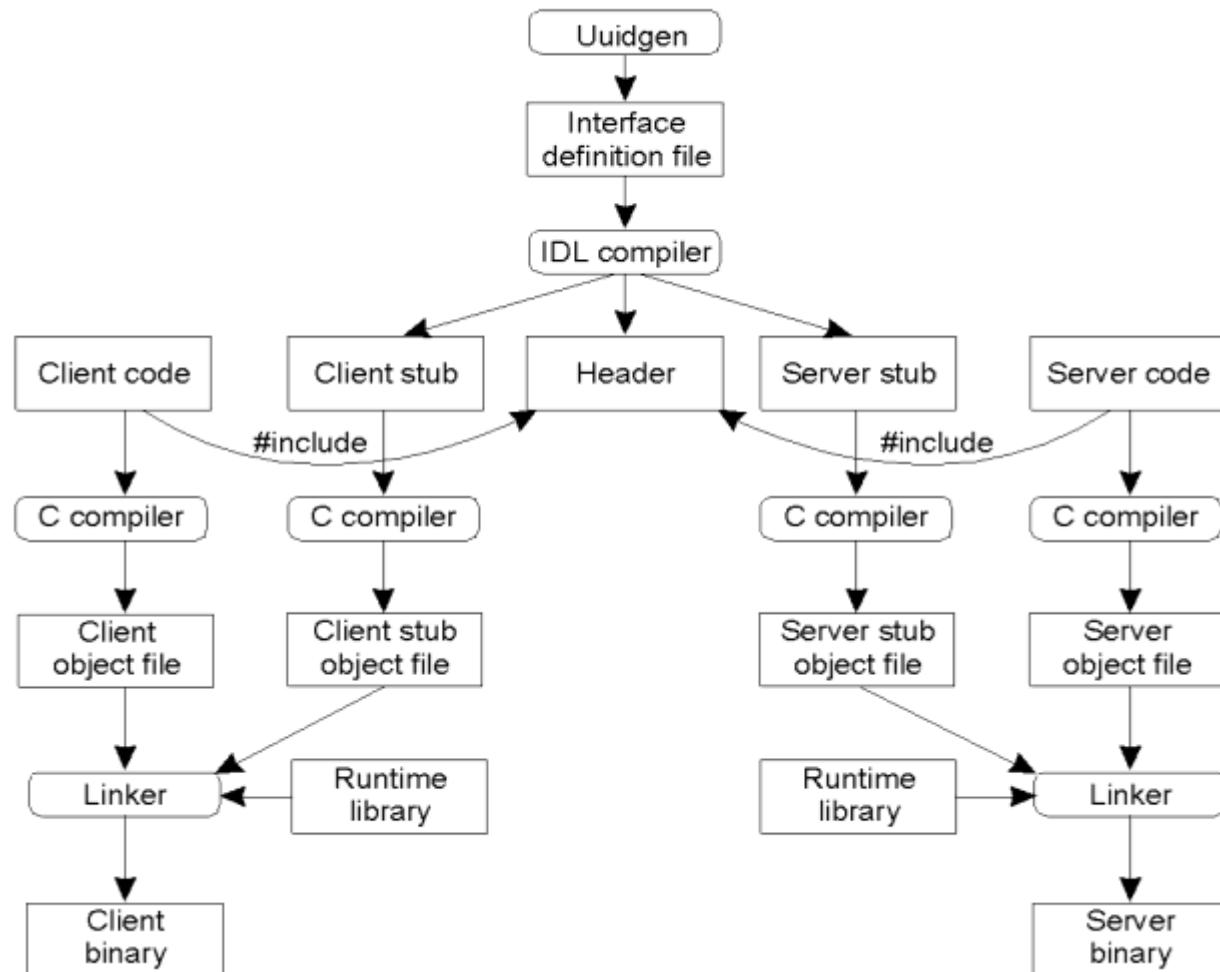
# \* Kada je IDL fajl potpun, poziva se IDL kompjajler.

- **idl primer.idl**
- Izlaz iz IDL kompjajlera sastoji se od 3 fajla
  - header fajla (npr. primer.h)
    - Header fajl sadrži jedinstveni identifikator, definiciju tipova, konstanti i prototipova funkcija.
    - On treba da bude uključen (korišćenjem #include) i u klijent i u server kod
  - klijent stub (**primer\_cstub.c**)
    - Klijent stub sadrži procedure koje će klijent program pozivati
    - Ove procedure su zadužene za pakovanje parametara u poruke, slanje poruka, prijem poruka, izvalačenje rezultata poziva procedure i prosleđivanje klijentu
  - server stub (**primer\_sstub.c**) –
    - Sadrži procedure koje se pozivaju kada stigne poruka do servera i koje zatim pozivaju odgovarajuću serversku proceduru

- \* Sledeći korak je pisanje klijent i server programa, a zatim kompiliranje ovih programa i klijent i server stub-a i povezivanje sa odgovarajućim stub-om da bi se dobio izvršni kod

- u opštem slučaju, DCE RPC aplikativni programer piše tri programa:

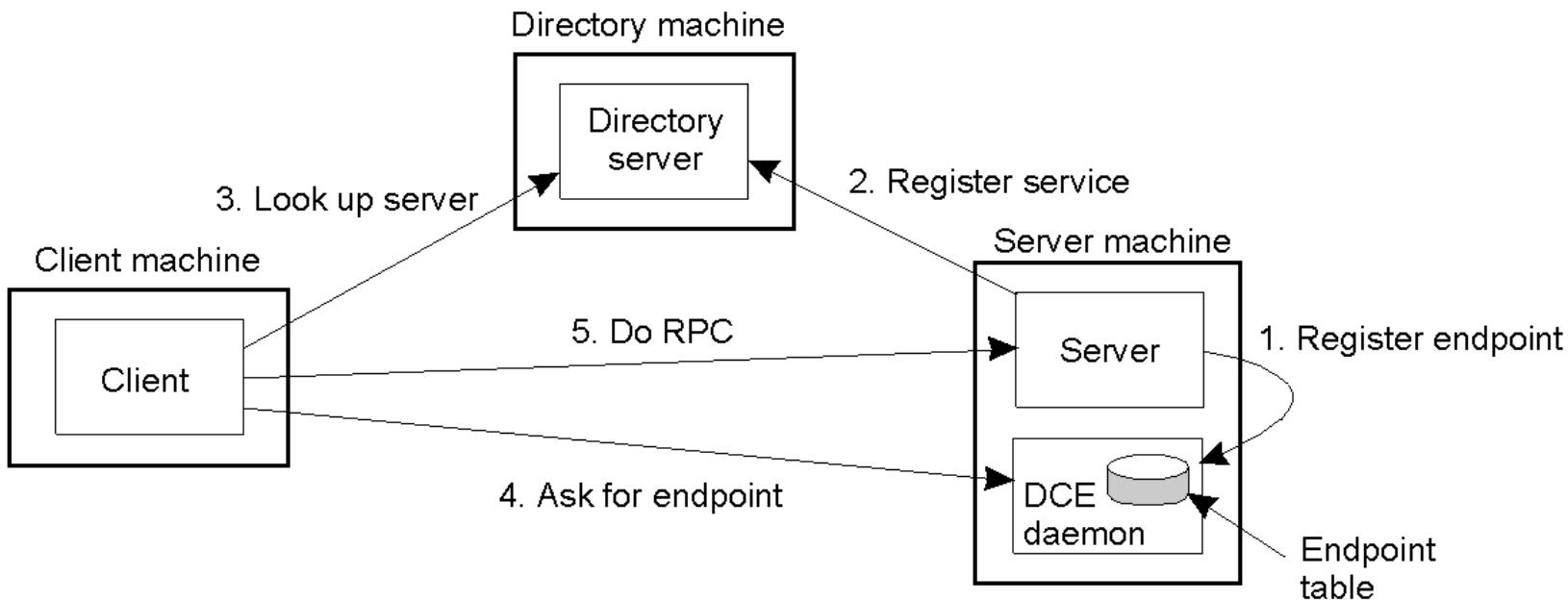
- klijent kod
- server inicijacijski kod kojim se server registruje u direktorijumskom serveru
- server operativni kod



# Povezivanje klijenta i servera

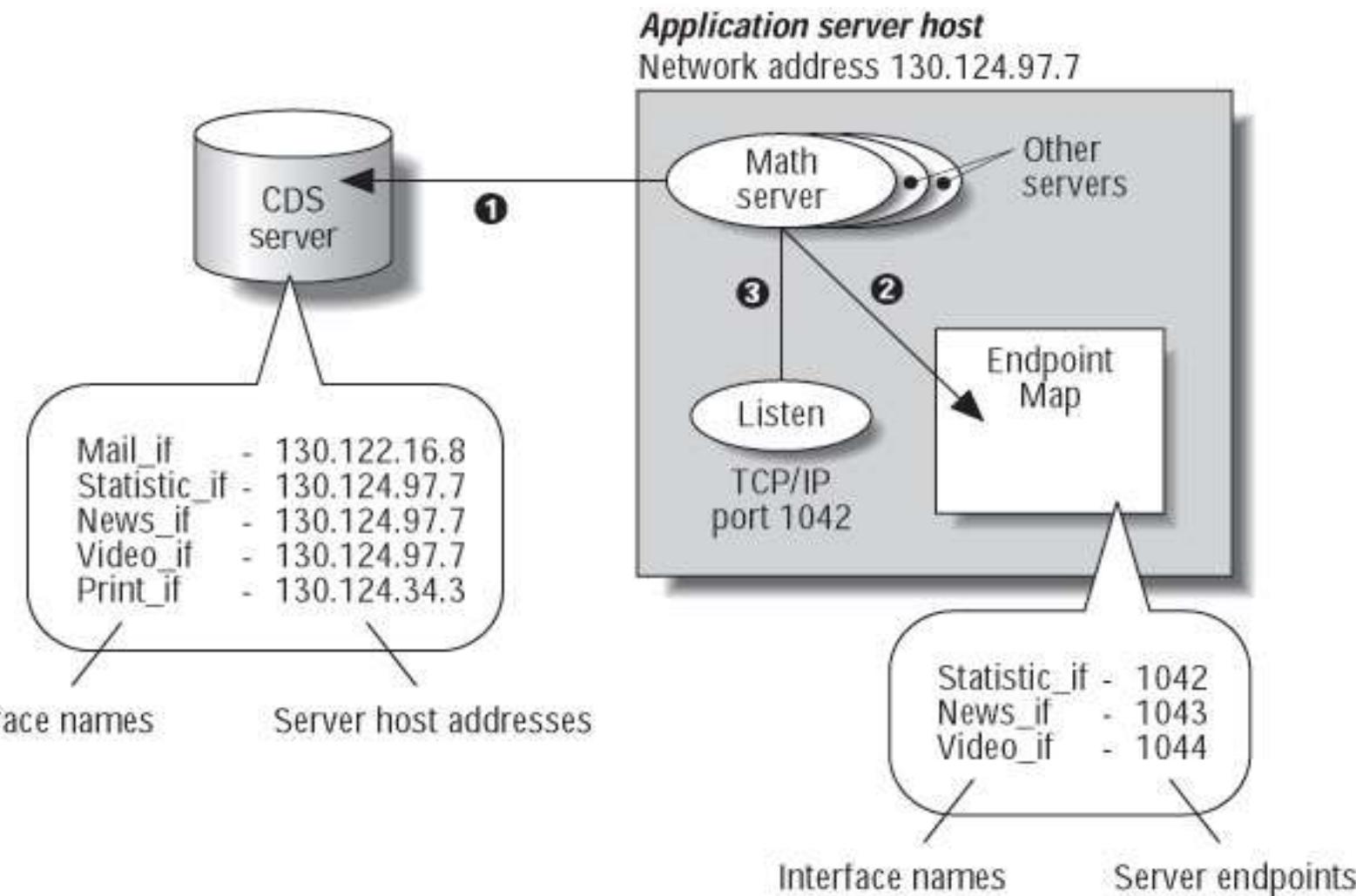
- \* Da bi klijent mogao da poziva server, neophodno je da server bude registrovan i spremjan da primi poziv.
  - Registracija servera omogućva klijentu da locira server i da se poveže sa njim
  - DCE klijent pronalazi server u dva koraka
    - Lociranje serverske mašine
    - Lociranje servera (tj. odgovarajućeg procesa) na serverskoj mašini.
  - Da bi klijent mogao da komunicira sa serverom (tj. procesom) on mora da zna broj porta (end point) na serverskoj mašini na koji može poslati poruke
    - Broj porta koristi serverski OS da bi prosledio poruku korektnom procesu (serveru)
    - U DCE-u na svakoj serverskoj mašini postoji tabela sa parovima (server, broj\_porta) koju održava poseban proces, DCE deamon (*rpcd*) (koji ima poznati broj porta)
    - Server mora od OS da dobije broj porta koji se zatim registruje u DCE deamonu
    - Server se registruje i u **direktorijumskom serveru** tako što dostavlja ime serverske mašine i svoje interfeise

# Povezivanje klijenta i servera



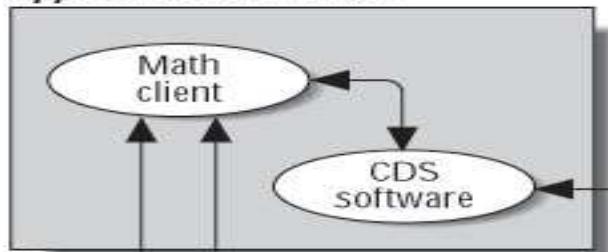
1. Registrovanje broja porta servera (procesa) u DCE deamonu
2. Registrovanje servera u direktorijumskom servisu (adresa serverske mašine, ime servera i interfeisi koje implementira)
3. Klijent kontaktira direktorijumski server da dobije adresu server mašine (IP adresu) na kojoj se izvršava server
4. Klijent kontaktira DCE deamon da bi dobio broj porta servera kome želi da pristupi
5. Poziv udaljene procedure

## registrovanje servera i njegovih interfeis u direktorijumskom serveru (CDS)

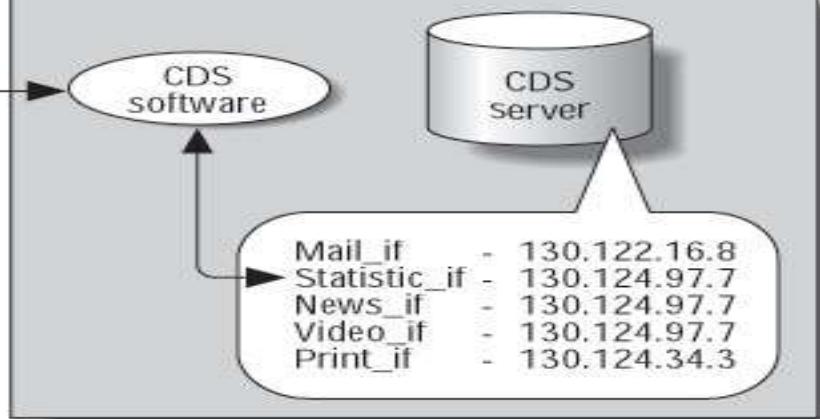


# \* pronalaženje Servera

*Application client host*



*DCE Directory Service host*

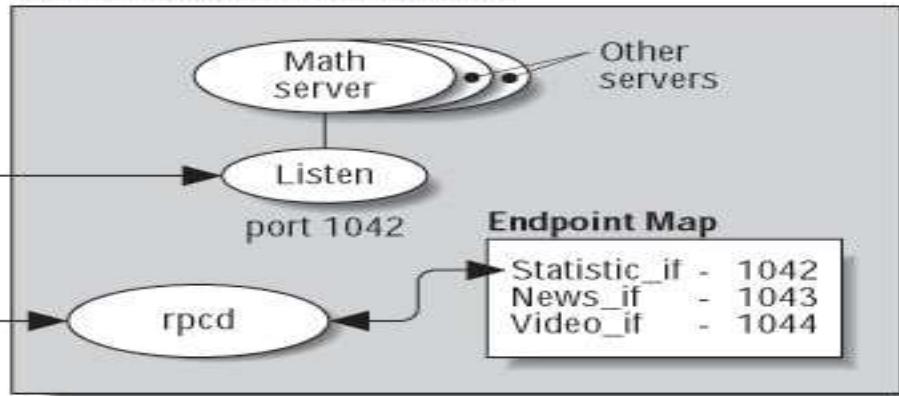


②

③

*Application server host*

Network address 130.124.97.7

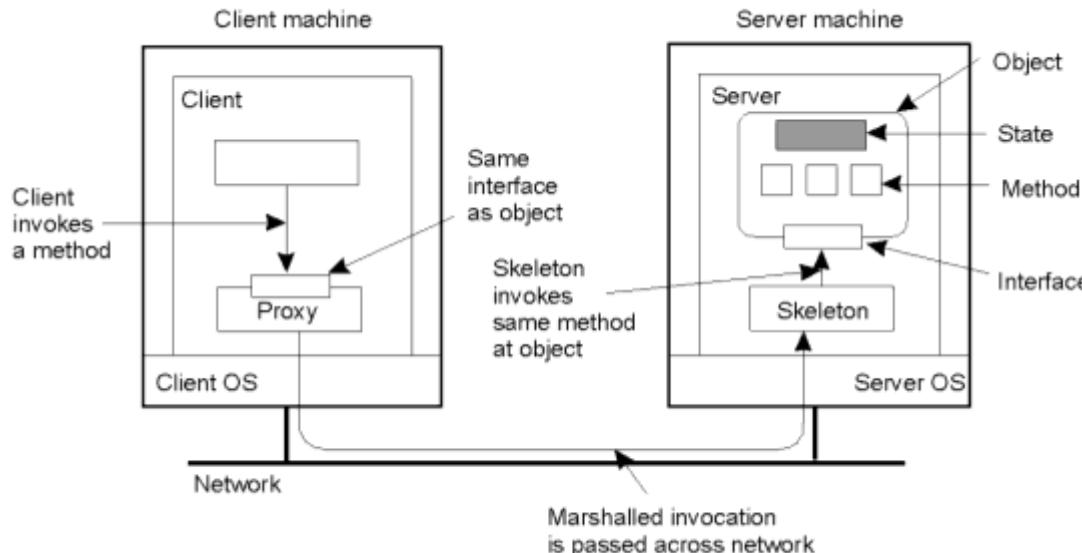


# OO pristup -Poziv udaljenih metoda (RMI)

- \* RPC na OO način – omogućava pozivanje metoda objekata koji se nalaze na drugom hostu
- \* Jedana od najvažnijih karakteristika objekta je da on skriva svoju unutrašnjost (podatke- stanja i operacije nad podacima –metode) od spoljnog okruženja uz pomoć dobro definisanog interfeisa.
  - ova osobina je ključna za skrivanje heterogenosti u DS
- \* objekat može imati skup metoda koje se mogu pozivati sa udaljenih računara.
  - Ovi metodi su definisani u remote interface fajlu. (objekat može imati i druge metode koje se mogu pozivati samo lokalno)
  - Ne postoji ni jedan drugi način da se pristupi podacima unutar objekta osim preko metoda koji su dostupni preko interfeisa
    - Interfeis predstavlja grupu metoda sa praznim telom (interfeis sadrži samo definicije metoda)
    - samo server ima implementacioni kod za objekat (i za lokalne i za udaljene metode)
    - Pristup objektima preko interfeisa je ključan faktor za DS:
      - Interfeisi mogu biti locirani na jednoj mašini a objekat na drugoj: ovakva organizacija poznata je pod nazivom distribuirani objekti ili udaljeni objekti.
        - » Objekat nije distribuiran (on se nalazi na jednoj mašini), a distribuiran je interfeis.

# Distribuirani objekti (nast.)

- \* Šta se dešava kada proces na jednoj mašini pozove metod (RMI – Remote Method Invocation) objekta koji se nalazi na drugoj mašini?



• Poziv se prosleđuje proxy-ju. Proxy je predstavnik server objekta u adresnom prostoru klijenta

IProxy implementira isti interfeis kao i server (ali na drugi način)

IProxy je analogan klijent stub-u: poziv metoda pakuje u poruku i prosleđuje serveru (poruka sadrži referencu udaljenog objekta, identifikator metoda i parametre poziva)

Objekat se nalazi na serveru na kome se nalazi isti interfeis kao na klijent strani

Dolazni poziv se prvo prosleđuje serverskom stub-u (skeleton), koji raspakuje poruku u poziv odgovarajućeg metoda na server strani

\* Kada klijent pozove metod udaljenog objekta, poziv se prosledjuje serverskom procesu u kome se nalazi udaljeni objekat.

- ova poruka mora da specificira objekat čiji se metod poziva

- koristi se identifikator udaljenog objekta koji je jedinstvem u čitavom distribuiranom sistemu – referenca udaljenog objekta (remote object reference)
- referenca udaljenog objekta se prenosi u poruci da bi se odredio objekat čiji se metod poziva
- referenca udaljenog objekta mora biti takva da se garantuje njena jedinstvenost u prostoru i vremenu
- jedan od načina da se to postigne je da referenca sadrži
  - IP adresu računara i broj porta procesa koji je kreirao objekat, vreme kreiranja, lokalni broj objekta (koji se inkrementira svaki put kada se objekat kreira u procesu)
  - referenca udaljenog objekta može sadržati i informacije o interfeisu udaljenog objekta (npr. ime interfeisa)

# povezivanje klijenta i servera

\* Kako klijent program doznaje referencu udaljenog objekta?

- koristi se poseban distribuirani servis - *binder*
- u binder-u postoji tabela sa preslikavanjem tekstualnog imena u referencu udaljenog objekta
- server registruje svoje udaljene objekte u binder-u
- klijent koristi binder da pronadje referencu udaljenog objekta

## \* Primeri middleware baziranih na pozivu udaljenih metoda

- CORBA (Common Object Request Broker Architecture )
- Java RMI
- DCOM (Distributed Component Object Model)

# Tipovi komunikacija

## \* Perzistentne (istrajne)

- poruka se pamti u komunikacionom serveru koliko je potrebno da bi se isporučila odredištu

## \* Tranzijentne

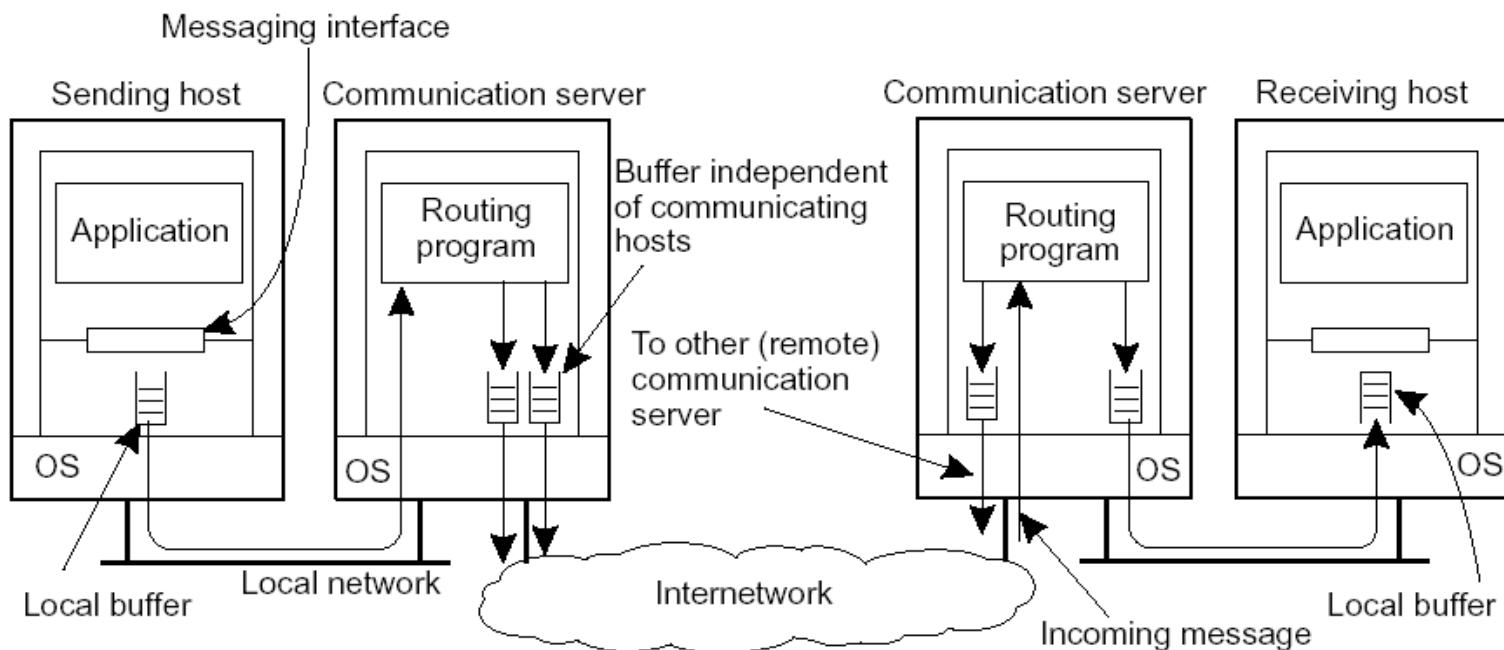
- poruka se odbacuje ako komunikacioni server nije u stanju da je isporuči sledećem serveru ili odredištu

## \* Sinhrone

## \* Asinhrone

# Perzistentne komunikacije

- \* poruke koje se prenose se pamte u komunikacionim serverima dok se ne proslede sledećem komunikacionom serveru.
- \* izvor ne mora biti aktivan nakon što dostavi poruku komunikacionom serveru
- \* prijemnik se ne mora izvršavati u trenutku kada izvor pošalje poruku.
  - Kod perzistentne komunikacije poruka koja treba da se pošalje je zapamćena u komunikacionom serveru onoliko dugo koliko je potrebno da bi bila preneta do prijemnika. (drugog komunikacionog servera)
    - Zbog toga nije neophodno da aplikacija koja je poslala poruku nastavi da se izvršava nakon predaje poruke kom. serveru.
    - Slično, korisnički agent na prijemnoj strani ne mora da se izvršava kada poruka stigne



# Primer: E-MAIL

\* E-mail sistem je tipičan primer perzistentne (istrajne) komunikacije

- Na hostu se izvršava korisnički agent (aplikacija pomoću koje korisnik može da kreira, šalje i prima poruke)
- Svaki host je povezan na tačno jedan mail server (odgovara komunikacionom serveru)
- Kada korisnički agent prosledi poruku za slanje svom hostu, host je dalje prosleđuje ka svom lokalnom mail serveru u kome se poruka privremeno skladišti
- Mail server obrađuje poruku, pronalazi adresu odredišnog mail servera i uspostavlja vezu sa odredišnim serverom.
- Odredišni server pamti primljenu poruku u ulaznom baferu (mail boxu primaoca)
- Ako je odredišni mail server privremeno nedostupan, lokalni mail server će u svom baferu nastaviti da čuva poruku.
- Korisnički agent pristupa lokalnom mail serveru i kopira poruke iz mail boxa u lokalni bafer.

# Tranzientne komunikacije

\* Kod tranzientnih komunikacija poruke se skladište samo dok se izvorna i odredišna aplikacija izvršavaju.

- Ako komunikacioni server ne može da isporuči poruku sledećem serveru ili prijemniku, poruka će biti odbačena.

\* Ako se agenti ne izvršavaju, poruke se odbacuju.

- Komunikacioni server u ovom slučaju je a store-and-forward ruter.
  - ako ruter ne može da prosledi poruku sledećem ruteru ili odredišnom hostu, poruka se odbacuje (npr. istekao time out)

# sinhrone i asinhrone komunikacije

- \* K-ka asinhronih komunikacija da pošiljalac nastavlja sa radom odmah nakon što prosledi poruku za slanje
    - Poruka se pamti ili u lokalnom baferu ili u komunikacionom serveru
  - \* Kod sinhronih komunikacija pošiljalac se blokira dok se poruka ne zapamti u lokalnom baferu odredišnog hosta (tj. dok ne stigne do odredišta)
- 
- \* Postoji nekoliko mogućih kombinacija ovih tipova komunikacija
    - Perzistentne asinhrone (e-mail sistem)
    - Perzistentne sinhrone
    - Tranzientne asinhrone
    - Tranzientne sinhrone

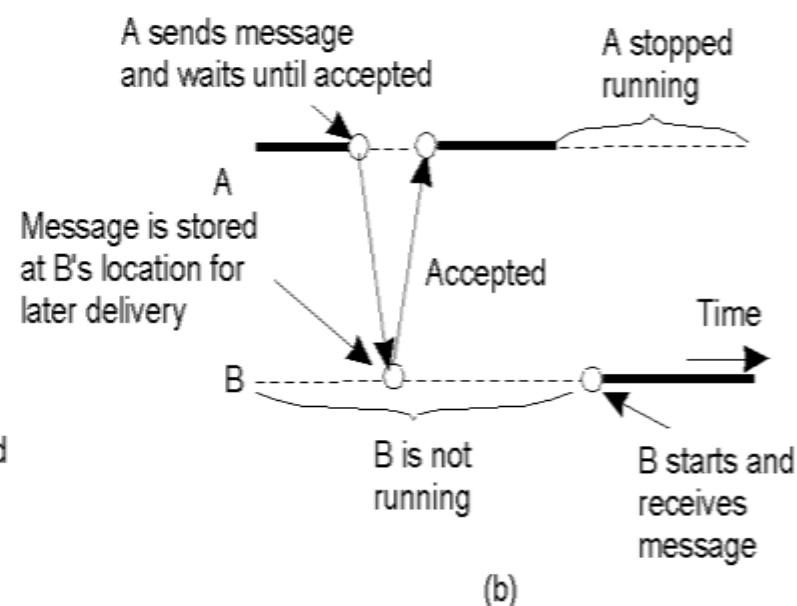
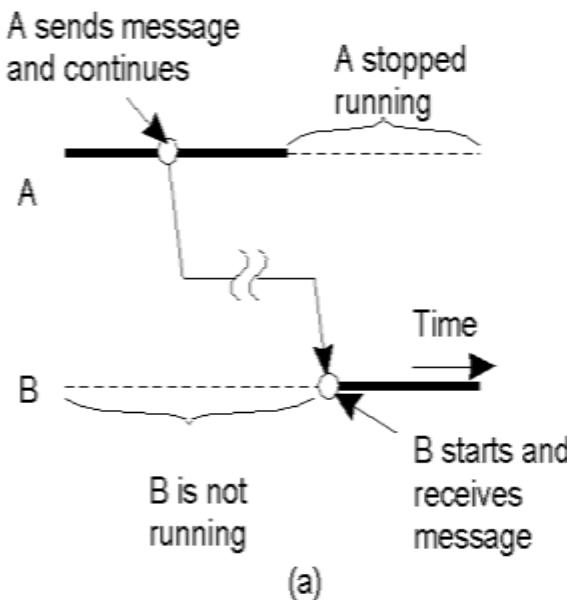
# Tipovi komunikacija (nast.)

## \* a) Perzistentne asinhrone

- Poruka je zapamćena ili u baferu lokalnog hosta, ili u prvom komunikacionom serveru sve dok se ne isporuči.
  - Pošiljalac nije blokiran dok čeka na isporuku svoje poruke

## \* b) Perzistentne sinhrone

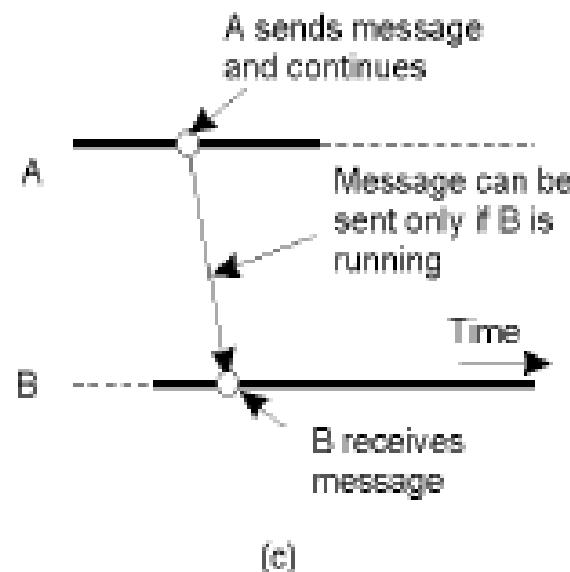
- Poruka mora biti zapamćena u odredišnom hostu da bi se pošiljalac deblokirao.
  - Nije neophodno da se aplikacija na odredišnoj strani izvršava da bi se poruka zapamtila u lokalnom hostu
    - Slabiji zahtev je blokiranje izvora dok se poruka ne zapamti u komunikacionom serveru odredišnog hosta.



# Tipovi komunikacija (nast.)

## \* c) Tranzijentne asinhronne komunikacije

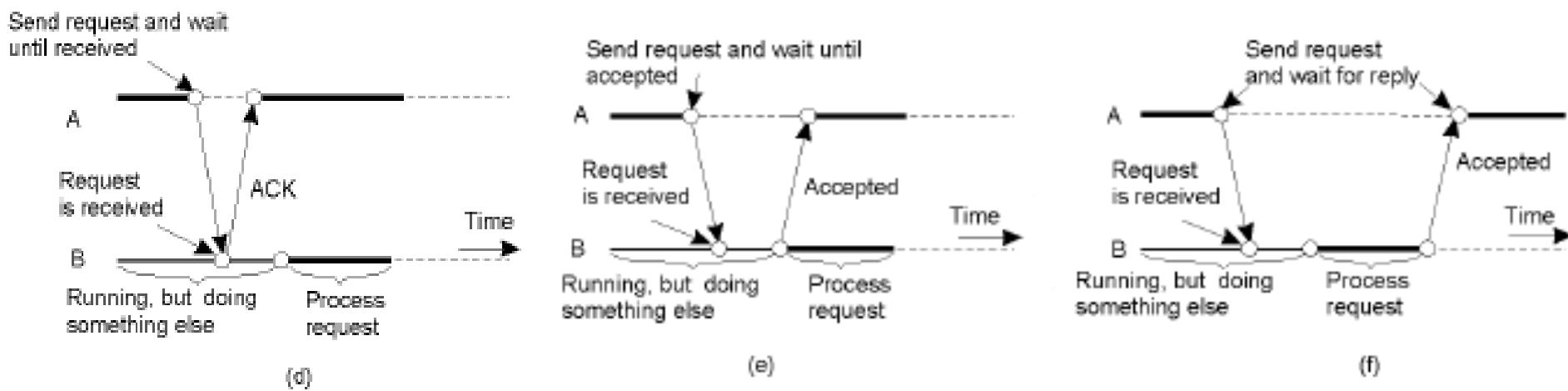
- Poruka se privremeno pamti u lokalnom baferu izvornog hosta, nakon čega aplikacija nastavlja sa izvršenjem
  - Paralelno, komunikacioni sistem prosleđuje poruku do odredišta gde se poruka skladišti u lokalnom baferu
  - Ako prijemnik nije aktivan u trenutku kada poruka stiže do njega, poruka je izgubljena



# Tipovi komunikacija (nast.)

\* Tranzientne sinhrone komunikacije se javljaju u nekoliko oblika

- d) Najslabija forma zahteva blokiranje izvora dok se poruka ne zapamti u baferu odredišnog hosta
  - Kada primi potvrdu izvor se deblokira
- e) izvor je blokiran sve dok se poruka ne prosledi odredišnom procesu za dalju obradu
- f) izvor je blokiran sve dok od odredišnog procesa ne primi odgovor (RPC i RMI funkcionišu po ovoj šemi)



# Middleware bazirani na razmeni poruka (message-oriented communication)

\* I RPC i RMI su po prirodi sinhroni – klijent se blokira dok njegov zahtev ne bude obrađen.

- Nekada takav vid komunikacije nije podoban

\* Rešenje nude DS orjentisani na razmenu poruka

- Prvi sistemi su se bazirali na tranzijentnim komunikacijama
  - Pružali su podršku za komunikaciju između procesa koji se jednovremeno izvršavaju.
    - Ovakve komunikacije često nisu adekvatne ako se ima u vidu potencijalno velika geografska distribucija procesa.
- Potreba za perzistentnim komunikacionim uslugama se pojavila kada je bilo potrebno integrisati aplikacije koje su se izvršavale na geografski jako udaljenim računarima (WAN).

# Tranzientne komunikacije: MPI

\* MPI (Message Passing Interface) middleware sistem koji se bazira na prenosu poruka.

- Namenjen je za podršku komunikacija u paralelnim sistemima (klasteri, mreža radnih stanica, multiračunarima, multiprocesorima)
  - Socket-i podržavaju komunikaciju između distribuiranih procesa razmenom poruka, ali su projektovani za TCP/IP protokol stek
    - Nisu pogodni za intrrprocesnu komunikaciju kada se koriste drugi protokoli
- Pre pojave ovog standarda, svaki proizvođač paralelnih sistema je nudio svoju varijantu sistema za interprocesorsku komunikaciju razmenom poruka.
  - Javio se problem prenosivosti programa

\* Pojavila se potreba za definisanjem standarda nezavisnog od hardverske platforme.

- MPI podržava tranzientne komunikacije između procesa.

# MPI (nast.)

- \* MPI podržava komunikaciju izmedju konkurentnih procesa
  - point-to-point
  - grupna komunikacija
- \* Implementiran je kao biblioteka funkcija koje se mogu pozivati iz konvencionalnih programskih jezika, kao što su Fortran, C, C++, MPI 3 standard preko 440 funkcija
- \* MPI podrazumeva da se komunikacija obavlja u okviru poznate grupe procesa.
  - Svakoj grupi se dodeljuje identifikator
  - Svakom procesu unutar grupe se takođe dodeljuje (lokalni) identifikator.
    - Par (identifikator\_grupe, identifikator\_procesa) na jedinstveni način identificuje izvor ili odredište poruke i koristi se umesto transprtih adresa.
  - U sistemu može postojati više grupa procesa (koje se mogu i preklapati) koje obavljaju izračunavanje i koje se izvršavaju u isto vreme.
- \* Srž MPI čine funkcije za razmenu poruka između procesa.
  - MPI podržava skoro sve oblike tranzientnih komunikacija.

# MPI (nast.)

\* mpiexec -n 10 hello.exe

## Hello World

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello world from %d of %d\n", rank, size)

    MPI_Finalize();
    return 0;
}
```

```
Hello from 1 of 10
Hello from 2 of 10
Hello from 6 of 10
Hello from 8 of 10
Hello from 9 of 10
Hello from 0 of 10
Hello from 5 of 10
Hello from 3 of 10
Hello from 7 of 10
Hello from 4 of 10
```

# MPI komunikacione primitive

Primitive	Značenje
<b><code>MPI_bsend</code></b>	Kopiraj izlaznu poruku u lokalni bafer MPI sistema i nastavi sa radom(neblokirajuća primitiva- tr. asinhr. kom.) (varijanta c) u prethodnom primeru)
<b><code>MPI_send</code></b>	Pošalji poruku i čekaj dok se ne iskopira u bafer odredišnog hosta (izvor se blokira dok poruka ne bude smeštena u MPI bafer odredišnog hosta – varijanta d).
<b><code>MPI_ssенд</code></b>	Pošali poruku i čekaj dok ne počne prijem (izvor se blokira dok odredište ne inicira prijem (receive) – varijanta e))
<b><code>MPI_sendrecv</code></b>	Pošali poruku i čekaj odgovor (izvor je blokiran dok ne stigne odgovor iz odredišta –varijanta f)) – ponaša se isto kao RPC
<b><code>MPI_isend</code></b>	Prosledi referencu na izlaznu poruku i nastavi sa izvršenjem (nema potrebe za kopiranjem poruke iz korisničkog bafera u bafer lokalnog MPI sistema) Da bi se izbeglo prepisivanje poruke postoji mogućnost provere da li je komunikacija završena ili se blokiranja dok se ne završi
<b><code>MPI_issend</code></b>	Prosledi referencu na izlaznu poruku i čekaj dok ne počne prijem
<b><code>MPI_recv</code></b>	Prihvati poruku; prijemnik se blokira dok ne stigne poruka
<b><code>MPI irecv</code></b>	Prijemnik je spremjan da prihvati poruku – prijemnik može da proveri da li je poruka stigla ili da čeka dok poruka ne stigne.

Zašto ovoliko različitih primitiva?

-Pružaju mogućnost projektantu programa da optimizira performanse

# Perzistentne komunikacije: sistemi sa redovima poruka (Message queuing systems)

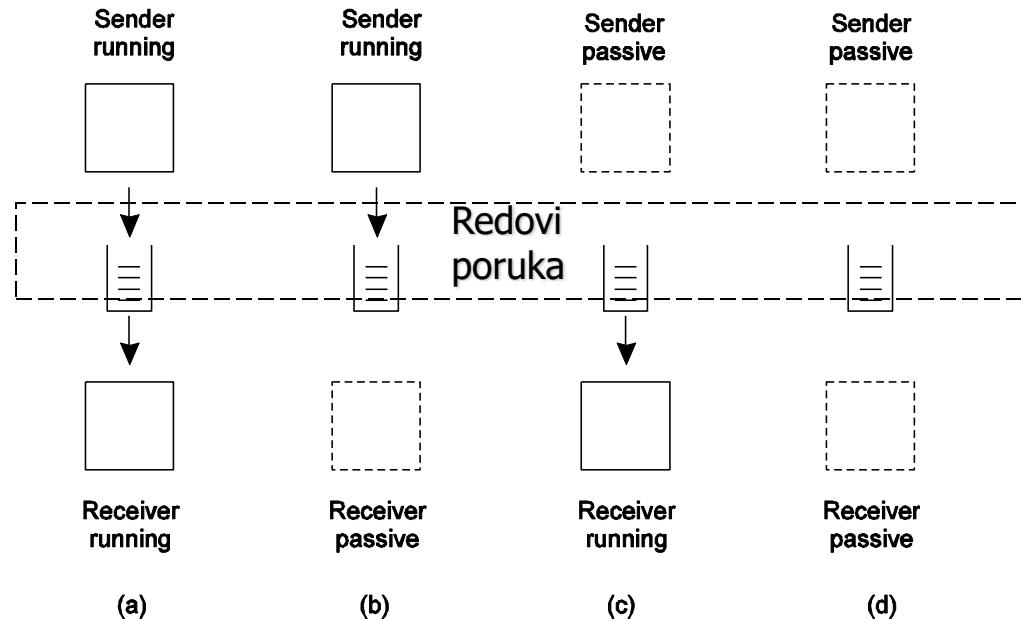
## \* važna klasa middleware sistema zasnovana na slanju poruka

- pružaju podršku perzistentnim asinhronim komunikacijama
- namenjeni su aplikacijama kod kojih je dozvoljeno da prenos poruka traje i nekoliko minuta (umesto nekoliko sec ili msec)
- Ovi sistemi poseduju smeštajne kapacitete za usputno pamćenje poruka i ne zahtevaju ni od izvora ni od odredista da budu aktivni za vreme prenosa poruka.

## \* Aplikacije komuniciraju smeštanjem poruka u posebne redove čekanja (queues).

- poruke se prenose preko niza komunikacionih servera
- Pošiljaocu se samo garantuje da će njegova poruka biti upisana u red čekanja primaoca.
- ne pružaju se garancije da li će (i kada će) poruka biti pročitana od strane primaoca.
- Ne postoji potreba da se pošiljalac izvršava u trenutku kad se poruka preuzima od strane primaoca
- Pošiljalac i primalac se izvršavaju potpuno nezavisno jedan od drugog
- Jednom kad se poruka isporuči u red čekanja, ostaje тамо, dok se ne ukloni, nezavisno od toga da li se pošiljalac ili primalac izvršavaju
- Ovo nam omogućava četiri kombinacije u odnosu na način izvršenja pošiljaoca i primaoca koji su prikazani na slici

# Sistemi sa redovima poruka (Message queuing systems)



- a) Pošiljalac i primalac se izvršavaju za celokupno vreme prenosa poruke
- b) Samo se pošiljalac izvršava, dok je primalac pasivan, tj. u stanju kad isporučivanje poruke nije moguće. I pored toga, pošiljalac i dalje može da šalje poruke
- c) Ovde je prikazan pasivni pošiljalac i aktivni primalac (primalac koji se izvršava) U ovom slučaju, primalac može čitati poruke koje su mu poslate, ali nije neophodno da pošiljalac poruke bude aktivan
- d) Opisuje situaciju gde sistem skladišti (i moguće prenosi) poruke dok su i pošiljalac i primalac pasivni. Neki smatraju da kad je ova konfiguracija podržana, sistem zaista podržava perizistentnu komunikaciju.

# Sistemi sa redovima poruka (Message queuing systems)

Veoma bitan aspekt iz perspektive middleware sistema je da su poruke u sistemu pravilno adresirane.

Adresiranje je omogućeno obezbeđivanjem jedinstvenog imena odredišnog reda u sistemu

Osnovni interfejs koji je na raspolaganju aplikacijama za pristup redu poruka je veoma jednostavan i sadrži 4 operacije: put, get, poll i notify.

Operation	Description
put	Append a message to a specified queue
get	Block until the specified queue is nonempty, and remove the first message
poll	Check a specified queue for messages, and remove the first. Never block
notify	Install a handler to be called when a message is put into the specified queue

# Sistemi sa redovima poruka (Message queuing systems)

Put operacija se poziva od strane pošiljaoca koji prosleđuje poruku sistemu koja treba biti dodata u specificirani red. Ovo je neblokirajući poziv.

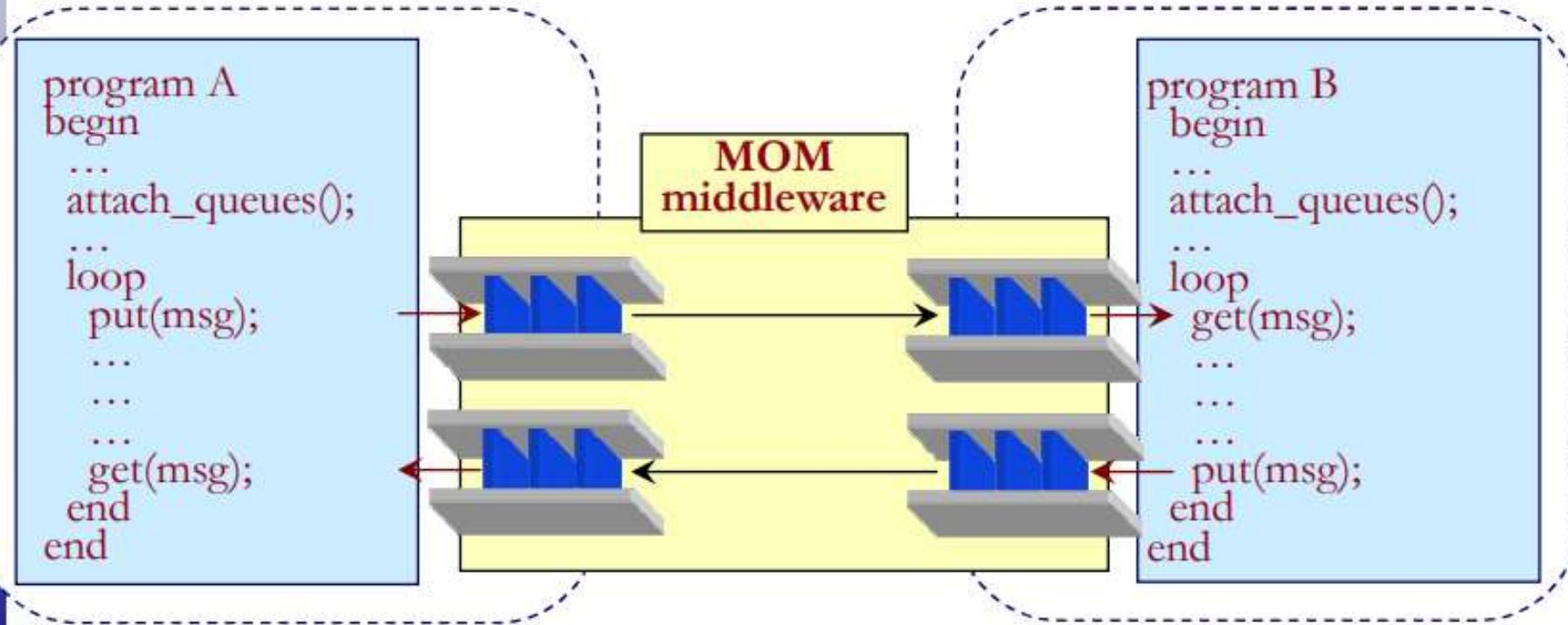
Get operacija predstavlja blokirajući poziv koja omogućava autorizovanom procesu da pribavi iz specificiranog reda proruku koja je najduže u njemu. Proces se blokira ako je samo ako je red prazan. Varijacije ovog poziva omogućavaju traženje specifične poruke u redu (pretraživanje poruke po sadržaju)

Poll operacija je neblokirajuća varijanta get operacije. Ako je red prazan, ili specifična poruka ne postoji pozivajući proces nastavlja sa izvršenjem

Notify operacija. Omogućeno je procesu da instalira handler koji se automatski poziva kad god je poruka dodata u red. Može se iskoristiti i da automatski startuje proces koji će pribavljati poruke iz reda, ako se nijedan proces u tom trenutku ne izvršava

Ovaj pristup je često implementiran kroz demon na strani prmaoca koji neprekidno nadgleda (osluškuje) dolazeće poruke u red i upravlja njima

# Sistemi sa redovima poruka (Message queuing systems)



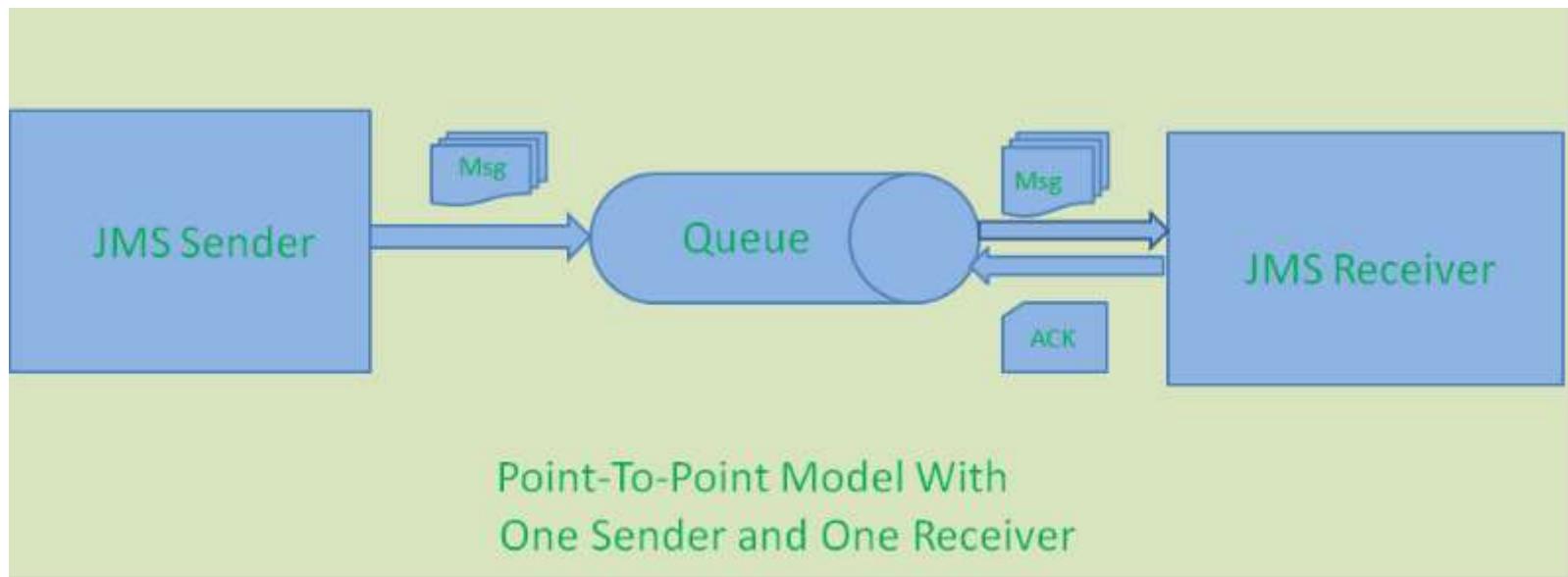
# Sistemi sa redovima poruka

Sistemi sa redovima poruka podržavaju Point-to-Point model razmene poruka koji može biti implementiran na sledeće načine:

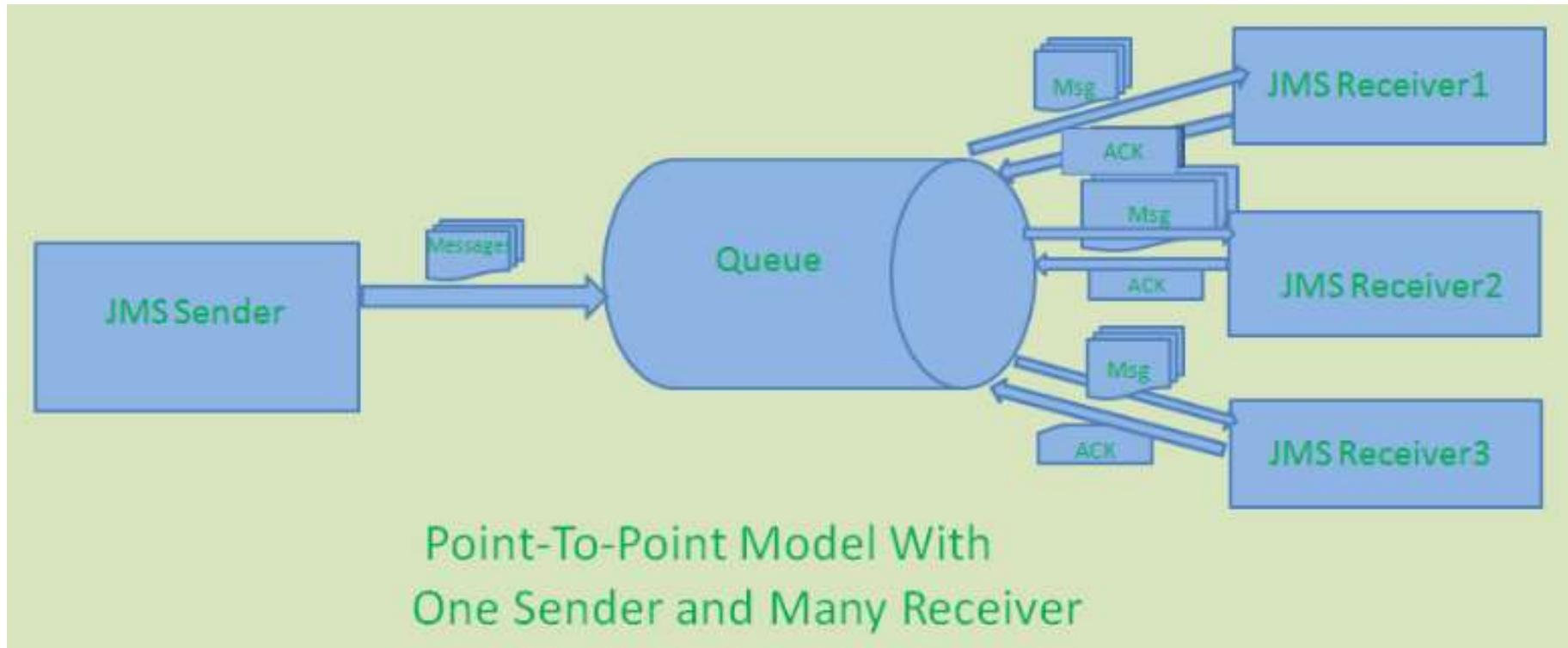
- \* Jedan pošiljalac i jedan primalac povezani preko reda
- \* Jedan pošiljalac i više primalaca povezani preko reda
- \* Više pošiljalaca i jedan primalac povezani preko reda
- \* Više pošiljalaca i više primalaca povezani preko reda

Nevezano za to koliki je broj pošiljalaca i primalaca poruka poslata od strane jednog pošiljaoca može biti primljena od strane samo jednog primaoca.  
U P-to-P modelu poruka od jednog pošiljaoca ne može biti preneta do više primalaca.

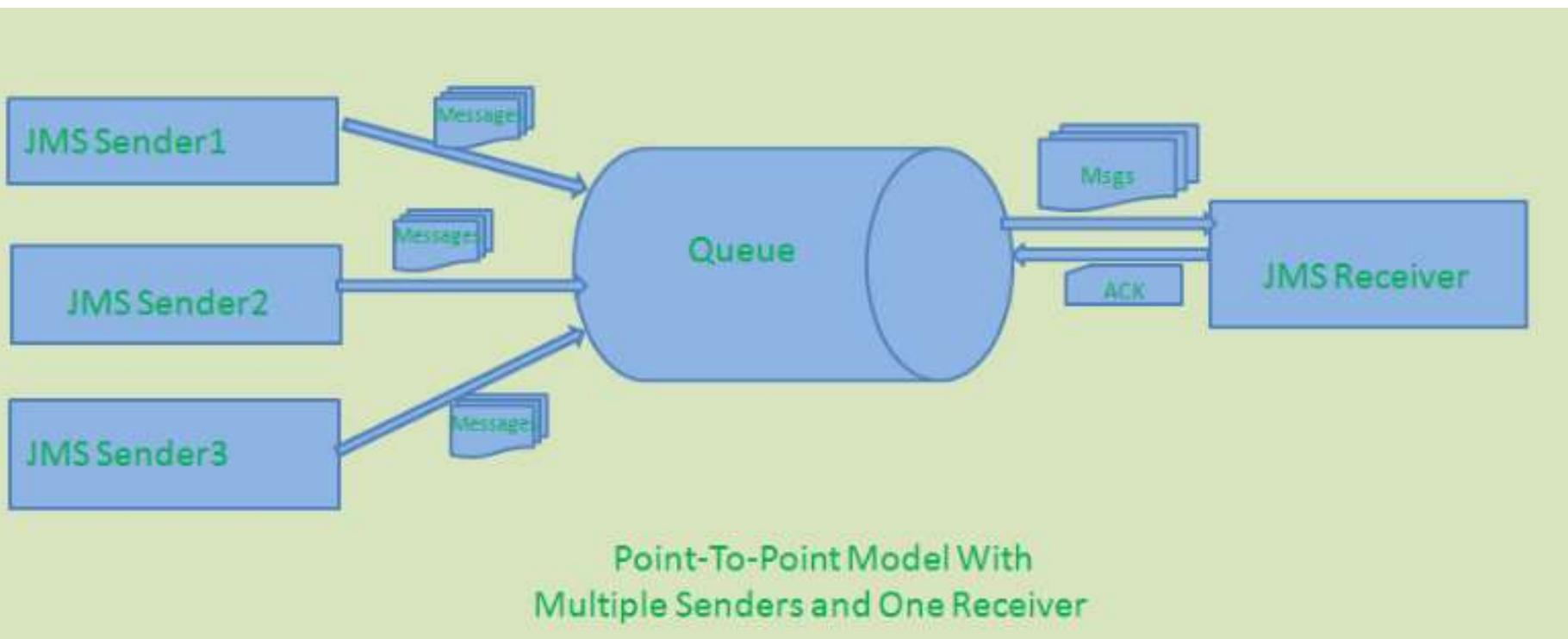
# Sistemi sa redovima poruka



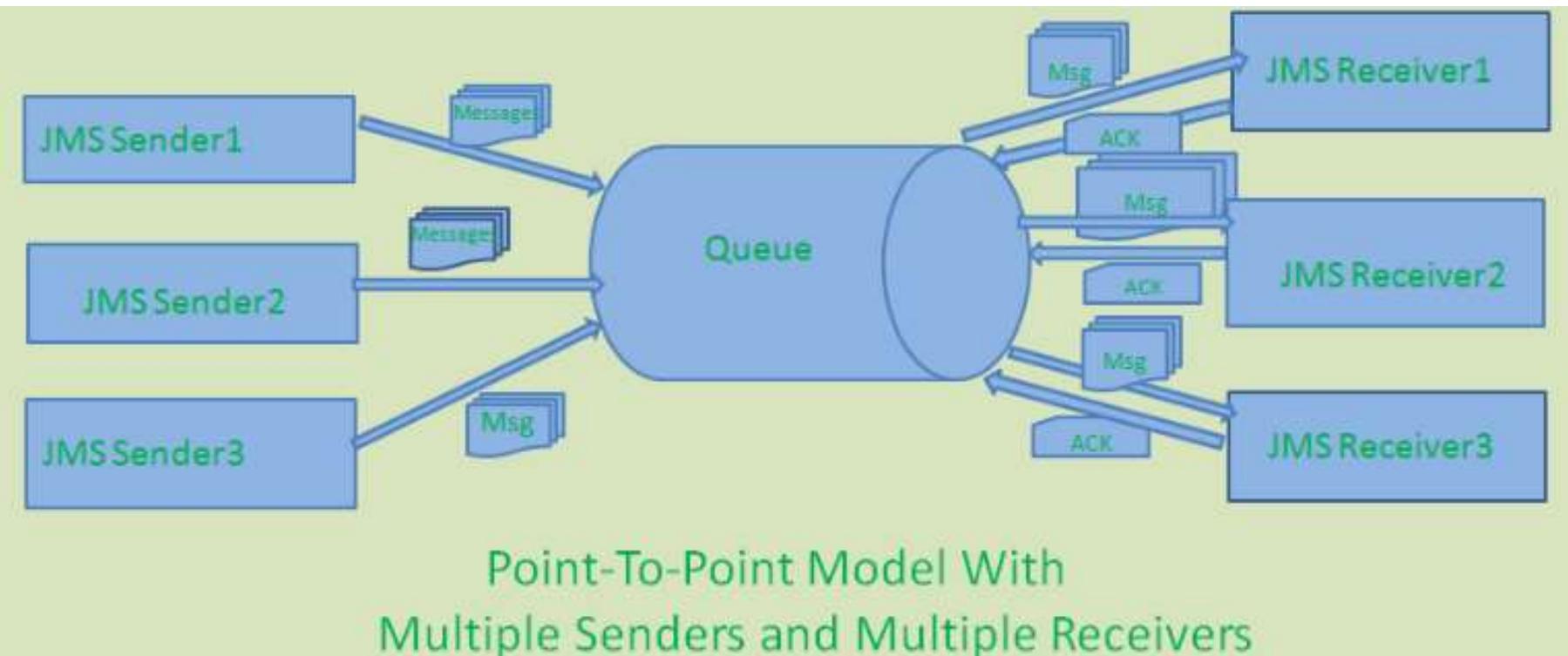
# Sistemi sa redovima poruka



# Sistemi sa redovima poruka

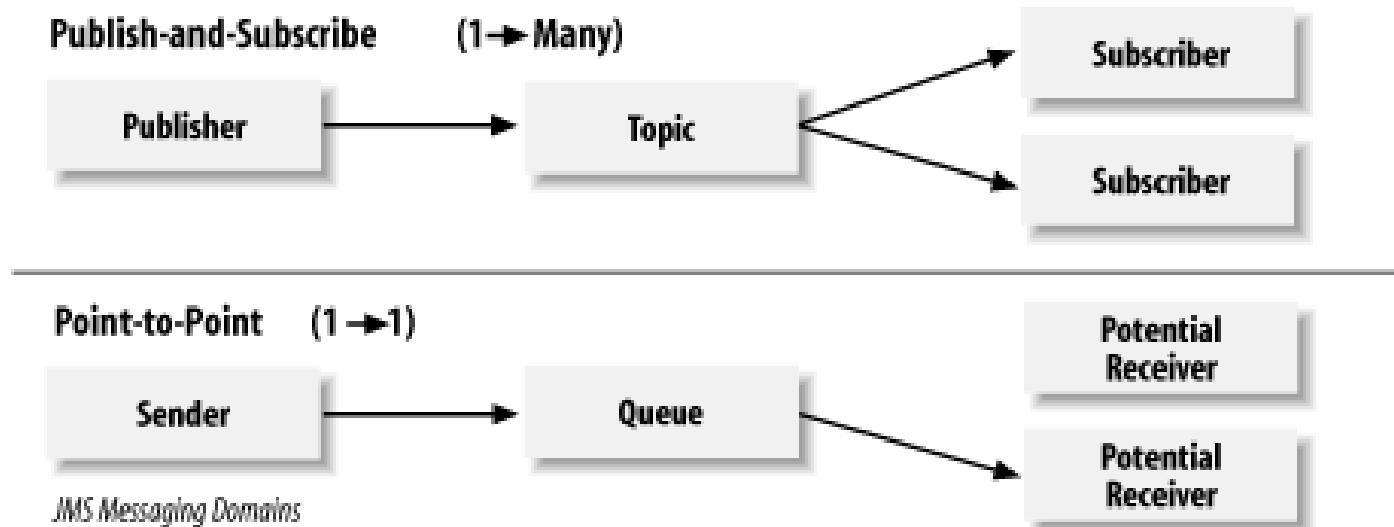


# Sistemi sa redovima poruka



# Publish-subscribe sistemi za razmenu poruka

- \* Umesto pošiljaoca i primaoca imamo Publisher i Subscriber.
- \* Za razliku od message-queing sistema gde se poruka prima samo od strane jednog primaoca kod Publish-Subscribe sistema poruka može biti primljena od više Subscriber-a
- \* Publisher kreira poruku i publikuje (objavljuje) poruku u Topic-u (temi) (ovde je destinacija Topic umesto reda (queue))
- Više različitih Subscriber-a se može prijaviti (pretplatiti) za Topic i koristiti poruke koje su objavljene u tom Topic-u.
- \* Subscriber može primiti poruku objavljenu u Topic-u, samo ako se Subscriber već prijavio za taj Topic. Svaka poruka poslata pre njegove prijave neće biti primljena od strane Subscriber-a
- \* Implementacija ovog modela može naložiti i da poruka ne bude primljena ako Subscriber nije aktivan, mada većina implementacija ne zahteva ovo ograničenje



# Sistemi za razmenu poruka

- Sistemi sa radovima poruka (i/ili Publish Subscribe sistemi) su implementirani u
- JMS, MSMQ, IBMMQ...

# Distribuirani sistemi - Sinhronizacija



- Fizički časovnici
- Logički časovnici
- Uzajamno isključivanje
- Algoritmi izbora koordinatora

# Sinhronizacija u DS

\* Usko povezana sa komunikacijom između procesa je i pitanje kako procesi međusobno sinhronizuju svoje aktivnosti u DS

- Važno je da procesi ne pristupaju jednovremeno deljivom resursu (npr. štampaču)
- Nekada je potrebno da se više procesa usaglase oko redosleda događaja (npr. da li je poruka m1 iz procesa P poslata pre ili posle poruke m2 iz procesa Q)
- Nekada je potrebno da se svi procesi usaglase oko izbora koordinatora
  - Taj problem se rešave pomoću algoritama izbora (election algorithms)

\* Problem kod DS, i generalno kod RM, je što ne postoji pojam globalnog vremena (svaka mašina ima svoj časovnik).

- Procesi koji se izvršavaju na različitim mašinama imaju različitu predstavu o vremenu.
- Problemi:
  - Kod sistema za rad u realnom vremenu neke aktivnosti treba obaviti u tačno određenom trenutku.
    - u tom slučaju DS treba da obezbedi striktnu sinhronizaciju časovnika
  - Održavanje konzistentnosti distribuiranih podataka se često bazira na vremenu kada je određena modifikacija obavljena

# Sinhronizacija u DS (nast.)

\* Postoji više načina za sinhronizaciju fizičkih časovnika u DS.

- Sve metode se u suštini svode na razmenu vrednosti časovnika uzimajući u obzir i vreme potrebno za prenos poruka.

➤ Varijacije u komunikacionom kašnjenju utiču na pouzdanost algoritama za sinhronizaciju časovnika.

\* U mnogim slučajevima poznavanje absolutnog vremena nije neophodno.

- Bitno je postići da se uzajamno zavisni događaji odvijaju u korektnom redosledu.
- Lamport je uveo pojam logičkog časovnika koji omogućava da se postigne neophodna sinhronizacija međusobno zavisnih događaja bez potrebe za sinhronizacijom fizičkih časovnika.

# Sinhronizacija časovnika

## \* U centralizovanim sistemima vreme je nedvosmisлено

- Kada proces želi da sazna vreme, on poziva sistemsku proceduru i dobija odgovor.
  - Ako proces A upita koliko je sati, a nešto kasnije proces B, vrednost koju proces B dobije biće veća (ili jednaka, ali nikako manja) od vrednosti koju je dobio A.

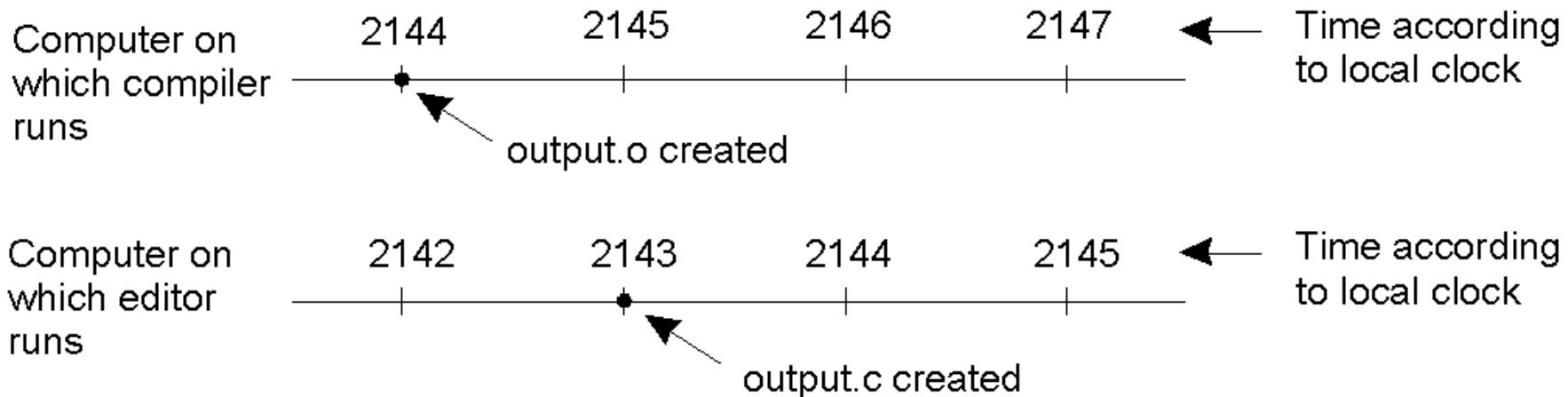
## \* Primer

- Kod UNIX-a je uobičajeno da se veliki programi podele na veći broj izvornih (source) fajlova tako da promene u jednom izvornom fajlu zahtevaju da se samo taj fajl rekompilira.
  - Ako se program sastojao od 100 fajlova a samo jedan je bio modifikovan, samo taj fajl će biti ponovo kompiliran
- Kompajliranje i likovanje programa UNIX-u se obavlja pozivom *make* programa.
  - Make radi tako što proverava vremena kada su fajlovi sa sufiksom .c i .o generisani.
    - Ako je vreme generisanja izvornog fajla (.c) manje od vremena generisanja objektnog fajla (.o), nema potrebe da se dati fajl kompilira.
    - Ako važi obrnuto, izvorni fajl je modifikovan i potrebno ga je rekompilirati.

# Sinhronizacija časovnika (nast.)

## \* Primer 2

- Prepostavimo da imamo DS pri čemu se kreiranje i editovanje programa obavlja na jednoj mašini a komajliranje na drugoj.
- Neka je fajl **output.o** generisan na jednoj mašini po lokalnom vremenu u trenutku 2144
- Neka je nakon toga fajl **output.c** modifikovan na drugoj mašini i neka nosi oznaku trenutka kreiranja 2143 (jer časovnik na toj mašini malo kasni u odnosu na časovnik na drugoj mašini).
- Kada se pozove make program, on će pogrešno doći do zaključka da je fajl **output.c** ne treba rekompilirati.



# Fizički časovnik

## \* Svi računari imaju kolo koje beleži vreme

- To kolo naziva se tajmer (timer)
- Tajmer je u suštini kvarcni kristal koji kada je pod naponom osciluje na poznatoj frekvenciji.
  - Brzina oscilacije zavisi od vrste kristala, načina sečenje i veličine napona.
- Svakom kristalu pridružena su dva registra
  - Brojač (counter) i holding registar
    - » Svaka oscilacija kristala dekrementira sadržaj brojača.
    - » Kada vrednost brojača dostigne 0, generiše se prekid a brojač se ponovo puni na početnu vrednost iz holding registra
    - » Na taj način moguće je programirati tajmer da generiše prekide u željenim vremenskim intervalima (npr. svake ms)
    - » Svaki prekid koji se generiše zove se otkucaj časovnika
  - Kada se sistem po prvi put podiže, korisnik unosi datum i vreme.
    - » nakon unošenja startnog datuma vreme se konvertuje u odgovarajući broj otkucaja i pamti u memoriji
    - » Vreme se pamti u CMOS RAM koji se napaja baterijom, tako da nije potrebno svaki put kod uključenja sistema unositi vreme i datum
    - » Nakon svakog otkucaja časovnika inkrementira se vrednost zapamćena u memoriji (softverski časovnik).

# Fizički časovnik (nast.)

\* Kada postoji samo jedan procesor i jedan časovnik, nije mnogo bitno da li je časovnik malo pomeren ili ne u odnosu na realno vreme.

- Pošto svi procesi na mašini koriste isti časovnik, oni će biti interno konzistentni.
- Ako ima više procesora, svaki sa svojim časovnikom, situacija se komplikuje.
  - Mada je frekvencija na kojoj osciluje kristalni oscilator veoma stabilna, nemoguće je garantovati da će kristali u različitim računarima raditi na istoj frekvenciji
  - Ako sistem ima n računara, svih n kristala će oscilovati neznatno različitim brzinama.
    - To će uzrokovati da softverski časovnici postepeno ispadnu iz sinhronizma i daju različite vrednosti pri očitavanju vremena i mogu dovesti do grešaka u radu sistema (kao u primeru *make* programa)

# Merenje vremena

- \* Od pojave mehaničkih časovnika u 17. veku, vreme se merilo astronomski.
  - Solarni dan označava vremenski period između dva uzastopna prolaska sunca kroz zenit.
    - Solarni dan traje 24h i ima 86400sec ( $1\text{sec}=1/86400\text{solarnog dana}$ )
    - 1940. god ustanovljeno je da period zemljine rotacije nije konstantan (pre 300 miliona godina, zemaljska godina je imala 400 dana)
    - Zbog oscilacija u brzini zemljine rotacije, astronomi računaju srednju dužinu solarnog dana na osnovu merenja velikog broja dana i pronalaženjem srednje vrednosti).
- \* Od pronaleta atomskog časovnika, 1948. god. postalo je moguće meriti vreme mnogo pouzdanije, (i nezavisno od brzine rotacije zemlje) brojanjem tranzicija u atomu cezijuma 133.
  - Fizičari su od astronoma preuzeli na sebe obavezu merenja vremena.
  - Fizičari definišu sec kao vreme koje je potrebno cezijumu 133 da načini **9,192,631,770 tranzicija**.
    - Ova brojka je uzeta da bi se atomska sekunda izjednačila sa solarnom sekundom.
  - Danas u svetu postoji oko 50 laboratorijskih časovnika.
    - Periodično, svaka laboratorija obaveštava Internacionalni biro za vreme u Parizu (Bureau International de l'Heure – BIH) koliko puta je njen časovnik otkucao od ponoći 1. januara 1958. godine. BIH pronađe srednju vrednost i generiše Internacionalno atomsko vreme (International Atomic Time – TAI)
    - Zbog oscilacija u brzini rotacije zemlje dolazi do neslaganja između TAI vremena i solarnog vremena. Solarni dan je duži za 0.001sec od TAI dana.

# Merenje vremena (nast.)

\* BIH rešava problem usaglašavanja solarnog i atomskog vremena tako što kada razlika postane 800ms, ubacuje se prestupna sekunda (leap second) u TAI vreme i tada TAI dan traje 864001 sec.

- Na ovaj način se vreme mereno atomskim časovnikom usaglašava sa solarnim vremenom
  - Takvo vreme zove se Univerzalno koordinisano vreme (Universal Coordinated Time – UTC).
  - UTC je zamenio raniji standard Srednje Griničko vreme.
  - od kada je ovaj sistem korekcije uveden 1972 god. ubačeno je 27 prestupnih sec, obično 31. decembra ili 30. juna.
    - tako npr ako je vreme bilo 23:59:59 sledeće će biti 23:59:60 (umesto 00:00:00), a zatim 00:00:00
    - Poslednja sec preskoka je ubačena 31. dec. 2016.
    - 30.juna 2020 će biti ubačena prestupna sec.

Za potrebe preciznog merenja vremena na raspolaganju je nekoliko UTC servisa

- Radio stanice koje emituju kratke impulse na početku svake UTC sekunde (WWV u USA, MSF u Engleskoj)
- Nekoliko geostacionarnih satelita
- Neophodno je posedovati odgovarajući prijemnik koji je podešen da radi na frekvenciji odgovarajućeg predajnika (prijemnici su komercijalno raspoloživi)

Izgled časovnika na [www.time.gov](http://www.time.gov) u toku  
prestupne sekunde 31. decembra 2016.



# GPS –global positioning system

- \* Da bi specijalne radio stanice ili geostacionarni sateliti mogli da se iskoriste za precizno merenje vremena, neophodno je poznavanje položaja izvora tačnog vremena i odredišta, da bi se kompenzovalo komunikaciono kašnjenje.
- \* problem određivanja geografske pozicije objekta se rešava uz pomoć specijalnog distribuiranog sistema: GPS.
  - GPS se sastoji od 29 satelita koji kruže u zemljinoj orbiti na visini od 20.000km.
  - svaki satelit ima 4 atomska časovnika koji se regularno kalibrišu sa specijalne stanice na zemlji.
  - satelit stalno emituje svoju poziciju i vreme  $[x_i, y_i, z_i, s_i]$
  - Da bi se odredio položaj objekta na zemlji neophodna su najmanje 4 satelita.

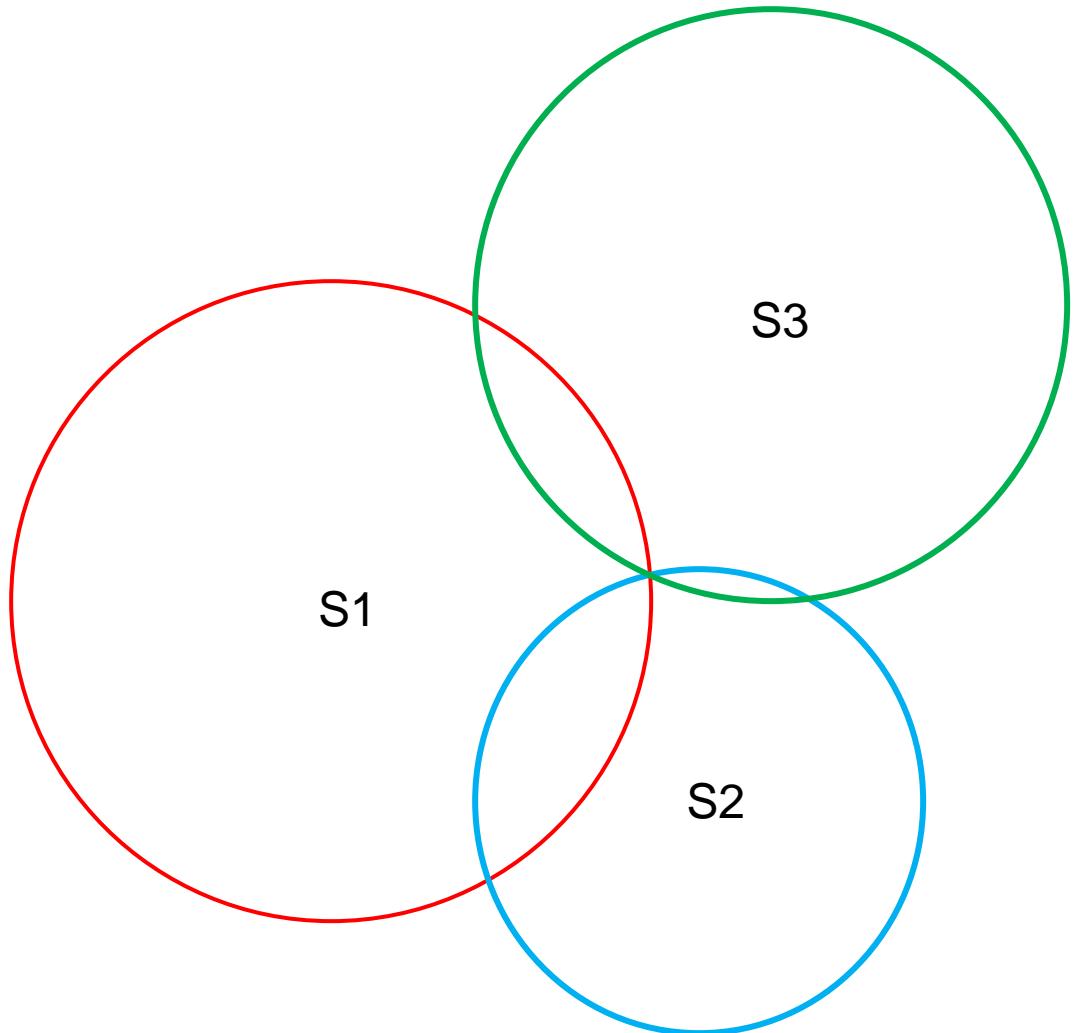
# GPS



# GPS: pozicioniranje i vreme

- Označimo koordinate i vreme satelita  $i$  sa  $[x_i, y_i, z_i, s_i]$
- neka su  $[x, y, z]$  nepoznate koordinate prijemnika na zemlji
  - neka poruka sa satelita  $i$ ,  $[x_i, y_i, z_i, s_i]$ , bude primljena po lokalno izmerenom vremenu u trenutku  $t'_i = t_i + \Delta$ , gde je  $\Delta$  pomeraj lokalnog časovnika u odnosu na tačno vreme ( $t'_i$  je vreme objekata na zemlji,  $t_i$  tačno vreme,  $\Delta$  odstupanje od tačnog vremena)
  - vreme prenosa poruke je  $(t'_i - \Delta - s_i)$ , pa je rastojanje od satelita do tačke na zemlji  $d_i = c(t'_i - \Delta - s_i)$ , ( $c$  brzina svetlosti)
  - očigledno je da postoji mnogo tačaka koje se nalaze na rastojanju  $d_i$  od satelita. To su sve tačke na sferi čiji je poluprečnik  $d_i$ .
    - jednačina te sfere je :

$$d_i^2 = (c(t_i' - \Delta - S_i))^2 = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2, \quad i = 1, 2, 3, \dots$$



# GPS: pozicioniranje i vreme

\* Stvari nisu tako jednostavne:

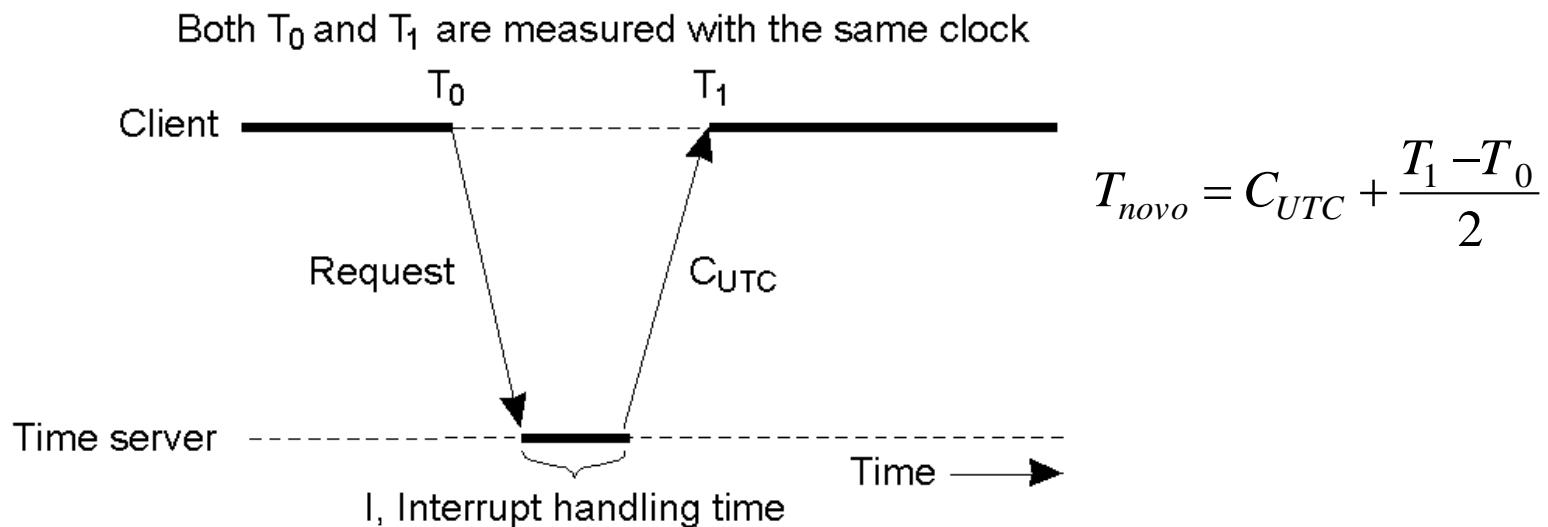
- zemlja nije savršena sfera
- atomski časovnici u satelitima nisu u svršenom sinhronizmu
- pozicija satelita nije precizno poznta
- brzina prostiranja signala nije konstantna (signal usporava kad uđe u jonosferu)
- ...

\* Ipak, i sa vrlo jeftinim GPS prijemnicima moguće je postići preciznost lokacije u granicama 1-5 metara i vremena od nekoliko destetina ns.

# Algoritmi za sinhronizaciju časovnika u DS

## \* Kristijanov algoritam – eksterna sinhronizacija

- U sistemu postoji jedna mašina sa WWV prijemnikom (server vremena – time server), a cilj je da sve ostale mašine budu sinhronizovane sa tom mašinom.
  - Periodično svaka mašina šalje poruku serveru vremena tražeći informaciju o trenutnom vremenu.
  - Server odgovara porukom u kojoj se nalazi informacija o vremenu,  $C_{UTC}$ .



# Kristijanov algoritam (nast.)

- Kada primi informaciju o vremenu klijent jednostavno može postaviti svoj časovnik na vrednost  $C_{\text{UTC}}$ .

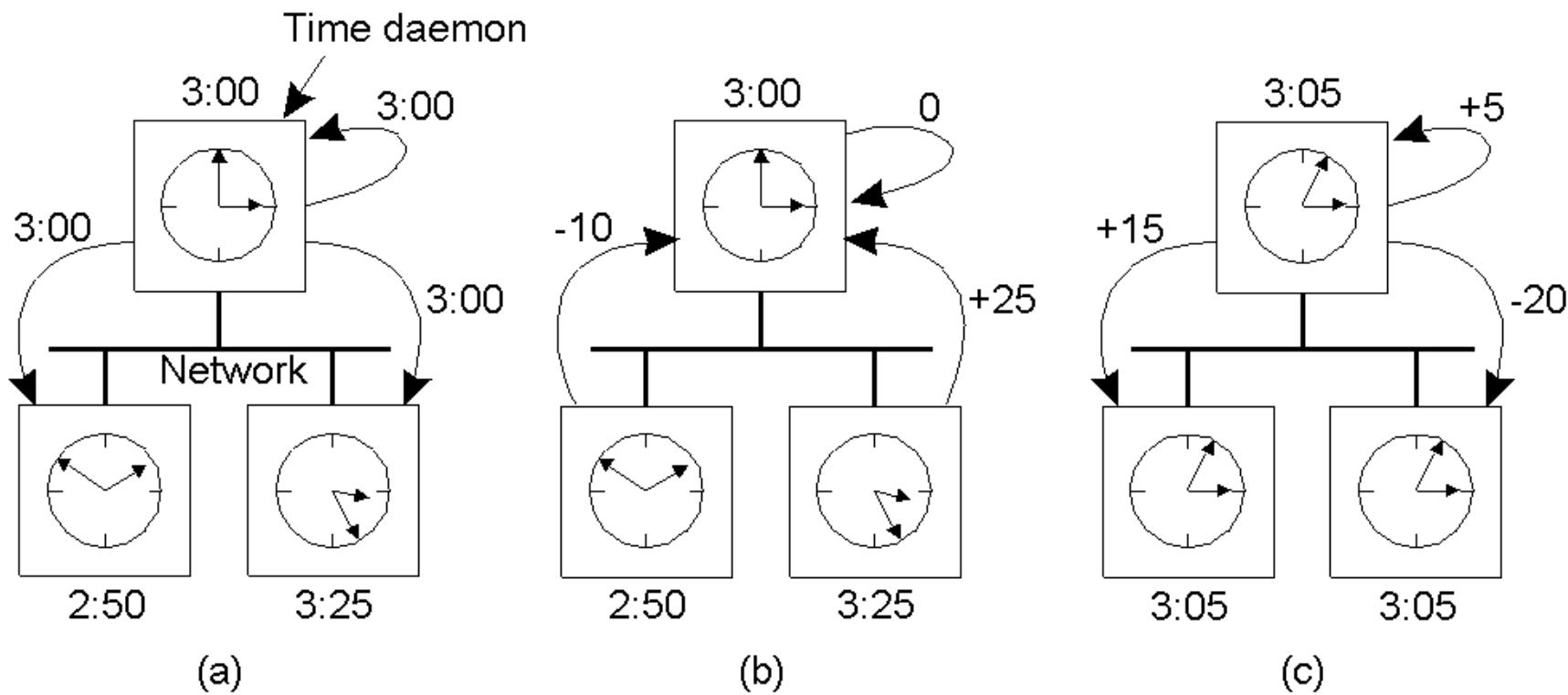
## \* Problemi

- Vreme nikad ne sme da ide u nazad.
- Ako je klijentov sat brži, onda će  $C_{\text{UTC}}$  biti manje od trenutne vrednosti klijentovog sata.
  - Jednostavno preuzimanje vrednosti časovnika dobijenog od servera bi moglo da stvori mnogo problema (npr. kod *make* programa)
  - Promena časovnika mora se obaviti postepeno (usporavanjem lokalnog)
    - Npr. ako je tajmer bio postavljen da generiše 100 prekida u sekundi, svaki prekid će dodati vremenu 10ms
    - Ako je potrebno usporiti vreme, svaki prekid će dodati, recimo, 9ms vremenu dok se ne postigne sinhronizacija.
- Propagaciono kašnjenje
  - Potrebno je da klijent izmeri vreme od trenutka kada je uputio zahtev serveru vremena ( $T_0$ ) do trenutka kada je stigao odgovor ( $T_1$ ).
  - Zatim se dobijena vrednost vremena od servera koriguje za  $(T_0-T_1)/2$

# Berkeley algoritam – interna sinhronizacija

- \* Kod Kristijanovog algoritma server vremena je pasivan (samo periodično odgovara na zahteve koje dobija od klijenata).
- \* Berekely UNIX koristi suprotan prilaz.
  - Server vremena (u suštini time deamon) je aktivan.
    - Obavlja prozivku svake mašine periodično da dozna koliko je vreme na dатој mašini.
    - Na osnovu dobijenih odgovora, server izračunava srednje vreme i saopštava ostalim mašinama kako da podese svoje časovnike.
    - Ovaj metod je pogodan za sisteme u kojima ne postoji WWV prijemnik

# Berkeley algoritam



- a) U 3:00 time deamon saopštava svoje vreme drugim mašinama i proziva ih da odgovore svojim vremenom
- b) Prozvane mašine odgovaraju tako što daju vrednost razlike
- c) Demon izračunava srednje vreme i saopštava ostalim mašinama kako da podeše svoje časovnike.

# Sinhronizacija fizičkih časovnika

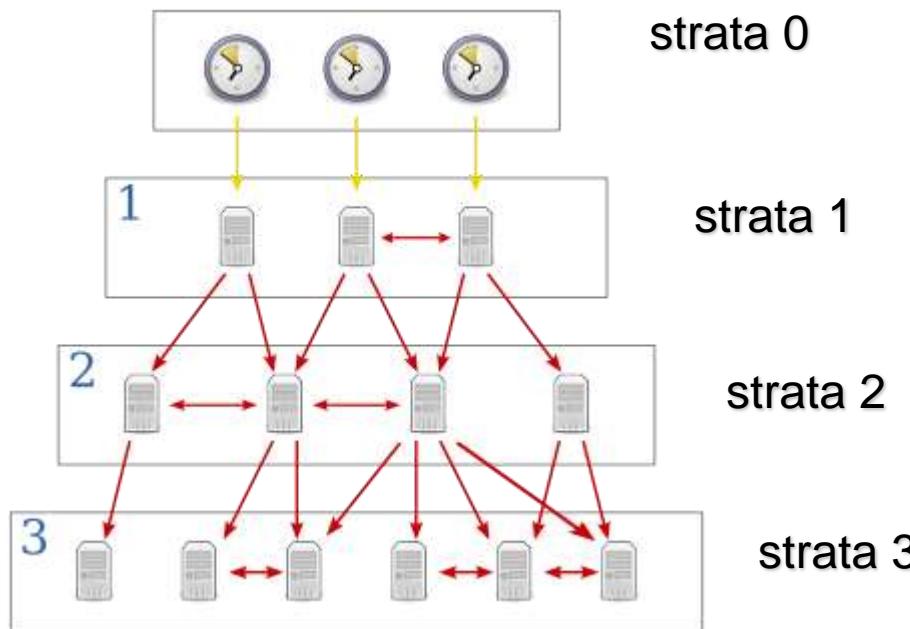
- \* Kristijanov algoritam i Berkley algoritam su prvenstveno projektovani da omoguće sinhronizaciju u intranet-u (privatnim mrežama).
  - Da bi se obavila sinhronizacija koristi se jedan server vremena
- \* Network Time Protocol (NTP) je projektovan za sinhronizaciju u Internetu (ili drugim nepouzdanim mrežama)
  - NTP koristi mnogo servera za sinhronizaciju
  - NTP je protokol aplikativnog nivoa

# NTP (network time protocol) protokol

\* NTP je internetov klijent-server protokol za sinhronizaciju časovnika u klijentima sa UTC vremenom sa visokim stepenom tačnosti.

- Serveri su izvori informacija o tačnom vremenu.
- Na transportnom nivou se koristi UDP protokol.
  - Server osluškuje klijente na portu 123.
- Serveri vremena su hijerarhijski organizovani u slojeve, tvz. stratum-e. Svakom sloju je dodeljen broj.
  - Numeracija kreće od 0.
  - Na nivou 0 se nalazi izvor tačnog vremena, neki atomski časovnik.
  - Na nivou 1 se nalaze serveri koji su direktno povezani sa atomskim časovnikom.
  - Na nivou 2 se nalaze serveri koji su direktno povezani sa serverima sa nivoa 1, itd.
  - Serveri nivoa i mogu biti međusobno povezani i mogu biti povezani sa više servera nivoa i-1.
  - Svi serveri zajedno čine sinhronizacionu podmrežu

# NTP sinhronizaciona podmreža



žute linije označavaju direktnu vezu  
crvene linije označavaju mrežnu vezu

# NTP protokol

- \* RFC 5905
- \* Postoje tri režima za sinhronizaciju časovnika u hostovima:
  - Simetrični režim – ovaj režim obezbeđuje najprecizniju sinhronizaciju i koristi se za sinhronizaciju master servera.
    - Par servera razmenjuje poruke koje sadrže informacije o vremenu.
  - klijent-server (RPC) režim – radi slično kao Kristijanov algoritam.
  - multicast režim – koristi se u brzim LAN.
    - NTP server periodično emituje (broadcast) vreme koje ostali hostovi koriste da se sinhronizuju.

# NTP –format poruka

0	1	4	7	15	23	31
LI	VN	Mode		Stratum	Poll	Precision
Root Delay						
Root Dispersion						
Reference Identifier						
Reference Timestamp (64)						
Origin Timestamp (64)						
Receive Timestamp (64)						
Transmit Timestamp (64)						
Optional Extension Field 1 (variable)						
Optional Extension Field 2 (variable)						
Optional Key/Algorithm Identifier (32)						
Optional Message Digest (128)						

Li – LI Leap Indicator (2 bits) – ukazje da li će zadnji sat toga dana imati prestupnu sekundu VN-verzija protokola

mode – režim sinhronizacije (simetrični, RPC, broadcast)

Stratum – nivo NTP servera koji šalje poruku

Poll- ceo broj koji predstavlja maksimalni interval izražen u log2 sec između dve uzastopne poruke (tipično 6 do 10)

precision – preciznost časovnika izražena u log2 sec. Npr. -18 ( $2^{-18}$  sec) odgovara preciznosti od 1 micro sec.

Rootdelay – kružno vreme propagacije do referentnog servera

Refid- 32-bitni identifikator servera koji se koristi kao referentni časovnik

Reftime – kada je sistemski časovnik poslednji put postavljen ili korigovan

# RPC režim

- Klijent kontaktira NTP server da bi sinhronizovao svoj časovnik
  - u poruci navodi svoje lokalno vreme ( $t_0$ )
  - server odgovara porukom u kojoj se nalazi  $t_0$ , vreme pristizanja poruke  $t_1$  (vreme na serveru), vreme slanja poruke  $t_2$  (serversko vreme)
  - Klijent beleži vreme pristizanja odgovora od servera,  $t_3$ , i određuje korekciju, tj. tačno vreme

$$\delta = (t_1 - t_0) + (t_3 - t_2)$$

kruzno vreme propagacije

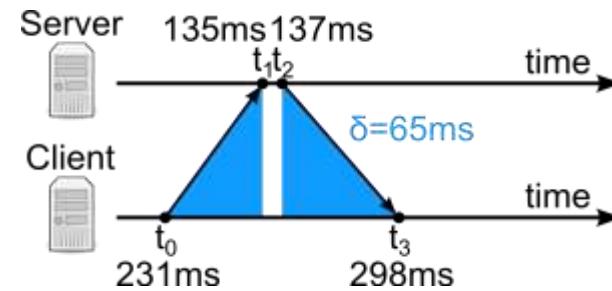
Tačno vreme

$$t_3 = t_2 + \frac{\delta}{2} + \theta \quad (\text{klijentsko vreme u trenutku prijema poruke od servera})$$

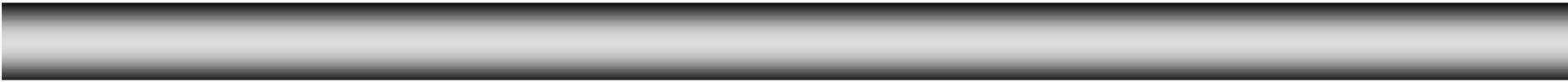
$$\theta = t_2 - t_3 + \frac{\delta}{2} \quad \text{offset klijentovog časovnika}$$

$$\theta = \frac{(t_1 - t_0) - (t_3 - t_2)}{2} = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

$$\text{Tacno vreme: } t_3' = t_3 + \theta$$



# Distribuirani sistemi - Sinhronizacija



- Logički časovnici
- Uzajamno isključivanje
- Algoritmi izbora koordinatora

# Logički časovnici

\* U mnogim aplikacijama nije važno fizičko vreme kada je neki događaj nastupio, već relativni redosled događaja( npr. *make* program)

- dovoljno je da se mašine koje ostavruju međusobnu interakciju usaglase oko relativnog vremena i to vreme ne mora da se slaže sa fizičkim vremenom.
  - Bitan je samo relativni odnos događaja
- Realtivno uređenje događaja bazira se na tzv. logičkim časovnicima.

# Lamportov algoritam sinhronizacije

\* Određuje redosled dogaćaja, ali ne vrši sinhronizaciju časovnika.

- Koristi se za sinhronizaciju logičkih časovnika

\* Lamport je definisao relaciju “desilo se pre” (happened before)

- A → B čita se “A se desilo pre B”.

- To znači da se svi procesi slažu da je prvo nastupio događaj A a zatim događaj B.

- Ova realcija može biti zadovoljena u dva slučaja

- Ako su  $a$  i  $b$  događaji u istom procesu, i ako  $a$  nastupa pre  $b$ , tada je relacija  $a \rightarrow b$  tačna.

- Ako je  $a$  događaj koji predstavlja slanje poruke iz jednog procesa, a  $b$  događaj prijema poruke u drugom procesu, tada je  $a \rightarrow b$  takođe tačna.

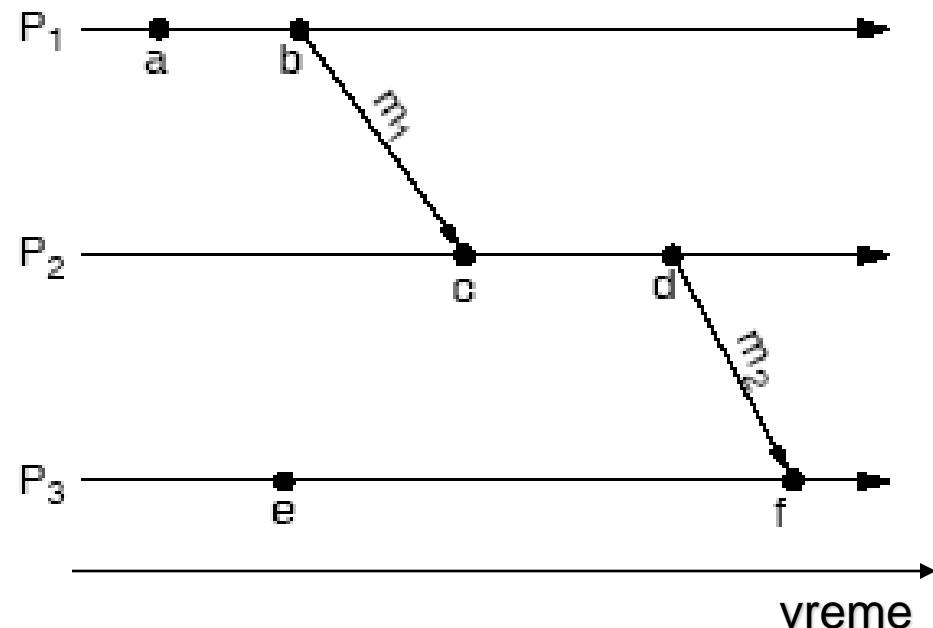
- Poruka ne može biti primljena pre nego što je poslata!

# Lamportov algoritam (nast.)

Ova operacija je tranzitivna:

$a \rightarrow b, b \rightarrow c$ , tada vazi  $a \rightarrow c$

Ako su se događaji  $x$  i  $y$  desili u dva procesa koji ne razmenjuju poruke (čak ni posredno, preko treće strane) tada relacija  $x \rightarrow y$  ili  $y \rightarrow x$  ne važi, a za događaje  $x$  i  $y$  se kaže da su konkurentni



$P_1, P_2, P_3$ : processes;

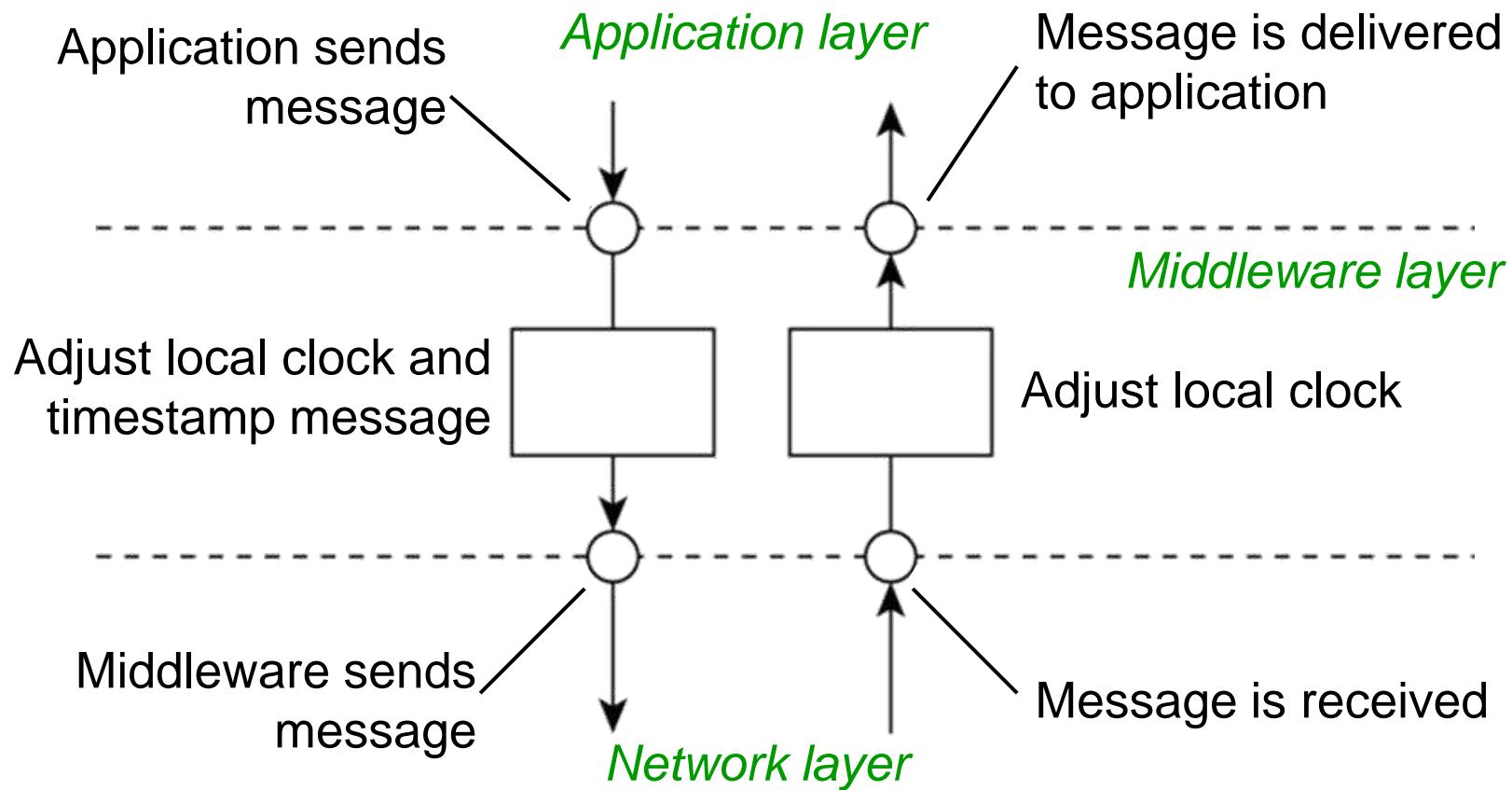
$a, b, c, d, e, f$ : events;

$a \rightarrow b, c \rightarrow d, e \rightarrow f, b \rightarrow c, d \rightarrow f$

$a \rightarrow c, a \rightarrow d, a \rightarrow f, b \rightarrow d, b \rightarrow f, \dots$

$a \parallel e, c \parallel e, \dots$

# Pozicioniranje Lamportovih logičkih časovnika u DS

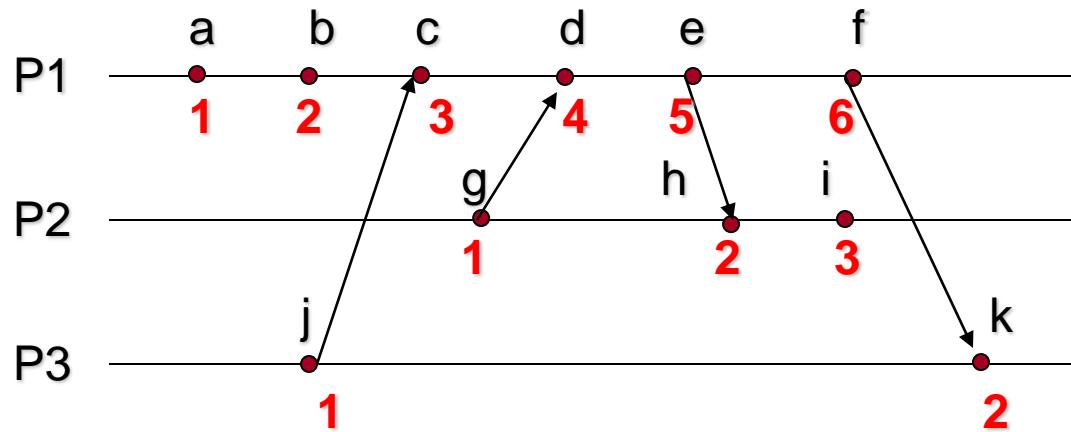


# Lamportov algoritam (nast.)

- \* Ako je realcija  $a \rightarrow b$  tačna, tada je potrebno ostvariti sinhronizaciju logičkih časovnika tako da važi da je  $T(a) < T(b)$ .
  - Svakom događaju se dodeljuje vremenska markica (timestamp)
    - Između svaka dva događaja sat mora da otkuca bar jednom
  - Sinhronizacija se ostvaruje korekcijom logičkih časovnika tako što se časovniku dodaje neka vrednost (vreme mora uvek da raste, ne može da se vraća u nazad)
- \* Pravila
  1. Logički časovnik  $L_i$  se inkrementira pre izvršenja bilo koje aktivnosti (slanje poruke kroz mrežu, isporuka poruke aplikaciji, ili neki drugi interni dogadjaj) u procesu  $P_i$ ,  $L_i = L_i + 1$ ;
  2. Kada proces  $P_i$  šalje poruku  $m$ , on u nju ubacuje i vrednost  $t = L_i$ ;
  3. Kada proces  $P_j$  primi poruku  $(m, t)$ , on podešava svoj lokalni logički časovnik tako što određuje  $L_j = \max(L_j, t)$  i primenjuje pravilo 1, a zatim prosledjuje poruku aplikaciji

# Primer

- \* Tri procesa P1, P2 i P3 na tri mašine
- \* Događaji a, b, c, ...
- \* Lokalni brojač događaja na svakoj mašini
- \* Procesi povremeno razmenjuju poruke.



Loša uređenost:

$$e \rightarrow h, \text{ ali je } T(e) > T(h)$$

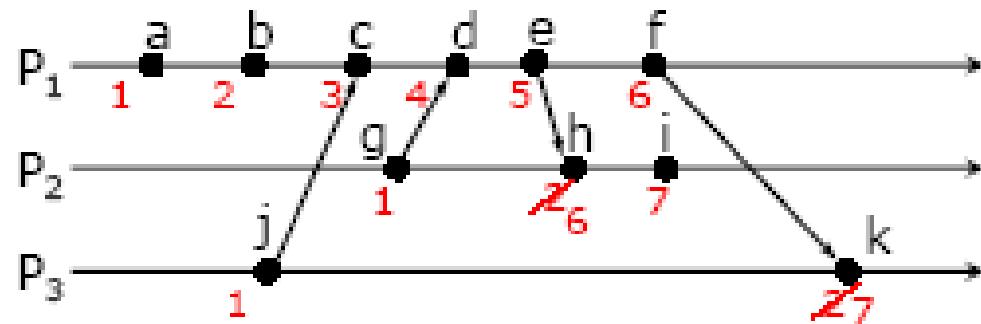
$$f \rightarrow k, \text{ ali je } T(f) > T(k)$$

# Lamportov algoritam

- \* Svaka poruka nosi vremensku markicu izvornog časovnika
- \* Kada poruka stigne proverava se

```
if lokalni_sat < vremenska_markica_poruke then  
    postavi lokalni sat na (vremenska markica + 1)  
else ništa nije potrebno korigovati
```

- Između bilo koja dva događaja sat mora otkucati bar jednom.



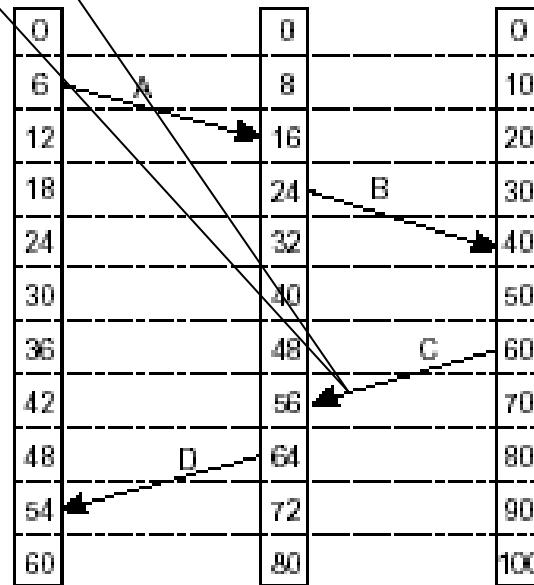
$$e \rightarrow h, \quad T(e) < T(h)$$

$$f \rightarrow k, \quad T(f) < T(k)$$

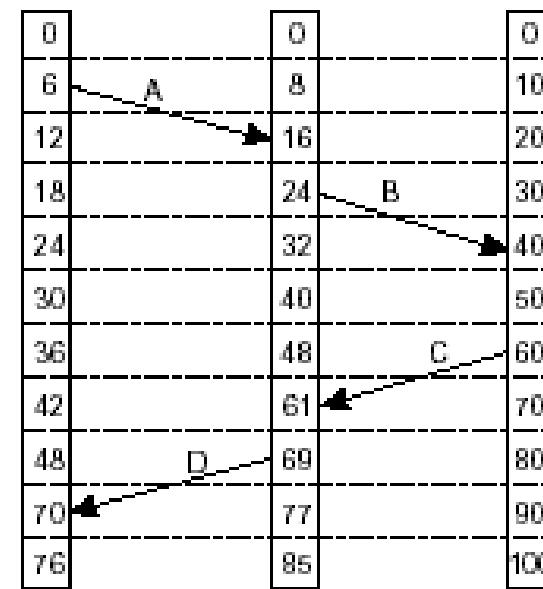
Algoritam omogućava da se održi vremensko uređenje međusobno uslovljenih događaja. (parcijalna uređenost unutar jednog procesa)

# Lamportove vemenske markice

poruka primljena pre nego što je poslata



(a)

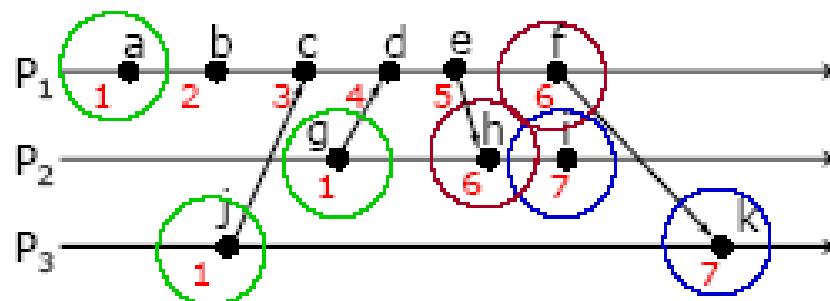


(b)

- a) Tri procesa sa svojim lokalnim satovima. Satovi rade različitim brzinama
- b) Lamportov algoritam vrši korekciju satova

# Problemi sa Lamportovim algoritmom\_1

\* Moguće je da više događaja koji nisu međusobno uslovljeni (tj. konkurentni su) imaju identične vremenske markice



- Ovo može dovesti do konfuzije ako više procesa treba da doneše odluku na osnovu vremenskih markica dva događaja.
  - Ako su događaji konkurentni, redosled nije bitan, ali potrebno je da svi procesi donešu istu odluku
    - To je teško postići ako su vremenske markice identične.

# Problemi sa Lamportovim algoritmom\_1 (nast.)

## \* Rešenje

- Prisiliti da svaka markica bude jedinstvena.
  - Definisati globalnu vremensku markicu ( $T_i, i$ )
    - $T_i$  predstavlja lokalnu Lamportovu markicu
    - $i$  predstavlja broj procesa (globalno jedinstven)
      - » Npr. adresa hosta, proces ID

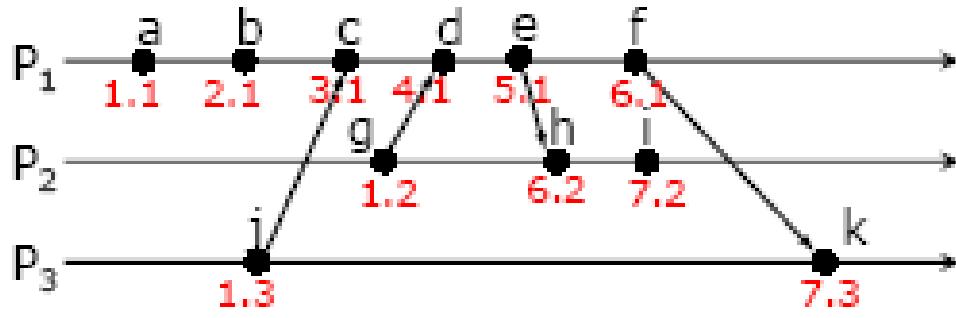
## ● Poređenje markica

$$(T_i, i) < (T_j, j)$$

ako i samo ako

$$T_i < T_j, \text{ ili}$$

$$T_i = T_j, \text{ i } i < j$$



# Primer: potpuno uredjena grupna komunikacija

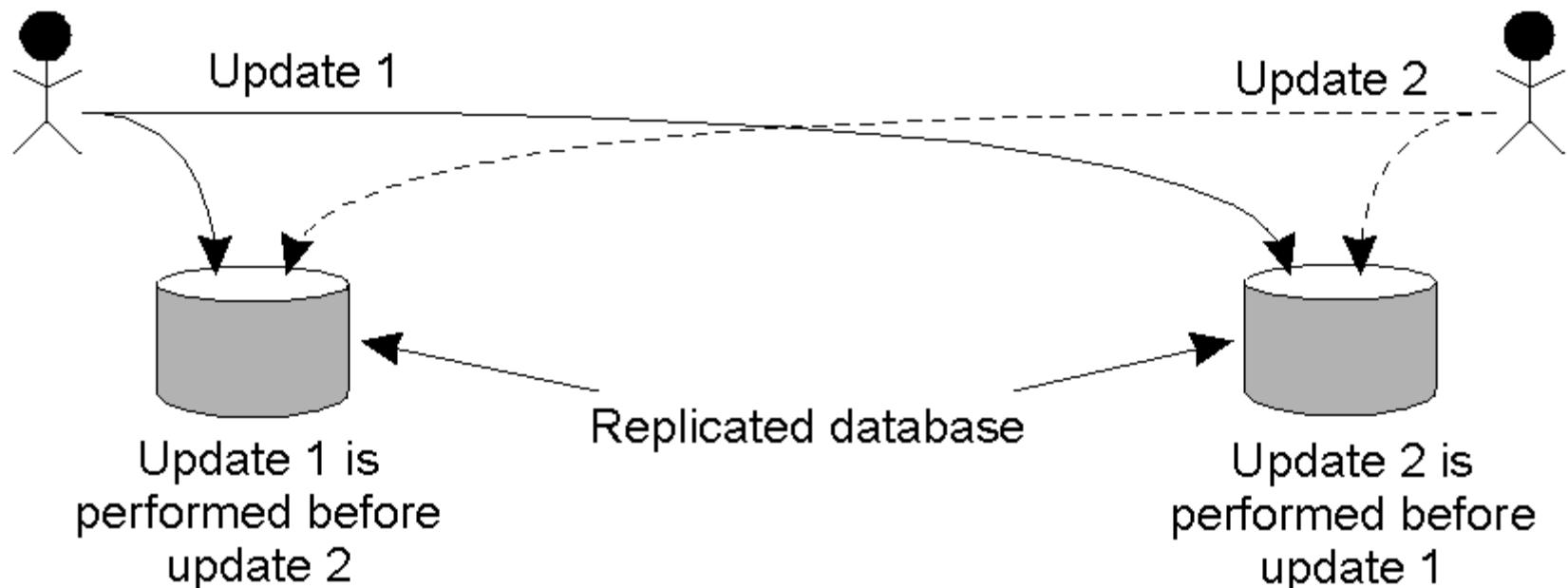
- \* Primena Lamportovih markica

- \* Scenario

- Replicirani bankovni računi u New Yorku (NY) i San Francisku (SF)
- Dve transakcije desavaju se jednovremeno i obavlja se multicast (zbog ažuriranja replika)
  - tekuće stanje: \$1,000
  - klijent dodaje \$100 na svoj račun u SF
  - službenik dodaje kamatu na račun od 1% u NY
  - ako ažuriranja računa nisu obavljena u istom redosledu u obe kopije (NY i SF) nakon ažuriranja jedna kopija će sadržati stanje računa \$1,111 a druga \$1,110.

# Primer: potpuno uredjena grupna komunikacija

\* Ažuriranje replicirane baze podataka koje ostavlja kopije u nekonzistentnom stanju.



# Primer: potpuno uredjena grupna komunikacija

- \* Potrebno je obezbediti da se operacije ažuriranja obave na obe strane u istom redosledu.
  - Mada postoji razlika da li će se prvo obaviti zaračunavanje kamate pa dodavanje depozita ili obrnuto, sa stanovišta konzistentnosti kopija (replika) nije važno koji se redosled poštuje.
- \* Neophodno je obezbediti potpuno uredjenu grupnu komunikaciju (multicast), tj. operaciju kojom se sve poruke isporučuju u istom redosledu svim prijemnicima.
  - NAPOMENA: Multicast se odnosi na slanje poruke iz jednog izvora u više odredišta (prijemnika).

# Primer: potpuno uredjena grupna komunikacija

## \* Algoritam

- Poruka ažuriranja se obeležava logičkim vremenom izvora
- Poruka ažuriranja se prosledjuje svima u grupi (multicast) uključujući i samog sebe
- Kada poruka stigne
  - smešta se u lokalni red čekanja (queue)
  - uredjuju se po markicama,
  - Poruka potvrde (ack) se prosledjuje svima (multicast) uključujući i samog sebe
- poruke iz istog izvora stižu u redosledu u kome su poslate (FIFO)

# Primer: potpuno uredjena grupna komunikacija

- \* Poruka se prosledjuje aplikaciji samo ako
  - se nalazi na vrhu reda
  - potvrđena je od strane svih procesa
- \*  $P_i$  šalje potvrdu (ack) za poruku koju je primio od  $P_j$  ako
  - $P_i$  nije poslao poruku ažuriranja
  - $P_i$ -ov identifikator je veći od identifikatora  $P_j$ -a
  - $P_i$ -ov zahtev za ažuriranjem je već obradjen;
- \* Lamportov algoritam garantuje totalno uredjenje dogadjaja

# Primer: potpuno uredjena grupna komunikacija

- \* Poruka m odgovara poruci "dodaj \$100" a n odgovara poruci "Dodaj kamatu od 1%".
- \* Kada se šalju poruke ažuriranja (npr., m, n) poruka će sadržati markicu koja je generisana kada je ažuriranje izdato.
  - Slanje poruke m sastoji se od slanja poruke ažuriranja i vremena kada je zahtev za ažuriranjem izdat (u ovom slučaju to je trenutak 1.1)
  - Slanje poruke n sastoji se od slanja poruke ažuriranja i vremena kada je zahtev za ažuriranjem izdat (u ovom slučaju to je trenutak 1.2)

# Primer: potpuno uredjena grupna komunikacija

- \* Poruke se prosledjuju svim procesima u grupi uključujući i samog sebe.
  - usvajamo da poruka koju proces šalje samom sebi stiže trenutno (tj. nema kašnjenja)
- \* Nakon slanja poruka redovi čekanja u procesima sadrže sledeće poruke:
  - P1: (m,1.1), (n,1.2)
  - P2: (m,1.1), (n,1.2)
- \* P1 će proslediti svima u grupi (multicast) potvrdu prijema za poruku (m,1.1) ali ne i za (n,1.2).
  - Zašto? Identifikator procesa P1 je niži od identifikatora procesa P2, a P1 je izdao poruku ažuriranja
  - $1.1 < 1.2$
  - P1 neće poslati potvrdu za (n,1.2) dok se operacija  $m$  ne izvrši.
- \* P2 će proslediti svima u grupi potvrdu za (m,1.1) i (n,1.2)
  - Zašto? Identifikator procesa P2 je viši od identifikatora procesa P1
  - $1.1 < 1.2$

# Primer: potpuno uredjena grupna komunikacija

- \* Ako P2 primi  $(n, 1.2)$  pre  $(m, 1.1)$  da li on svima u grupi prosledjuje potvrdu za  $(n, 1.2)$ ?
  - Da!
- \* Postavlja se pitanje kako P2 zna da postoje druga ažuriranja koja treba da budu obavljena pre njegovog?
  - Ne zna;
  - Proces P2 ne može da obavi ažuriranje definisano porukom  $(n, 1.2)$  dok ne dobije potvrde od svih ostalih procesa, što u ovom primeru znači od procesa P1.
    - poruke iz jednog izvora uvek stižu po redosledu, što znači da ne može da se desi da ack iz P1 stigne pre njegove poruke  $(m, 1.1)$  koju je ranije poslao
- \* Da li proces P2 šalje svima u grupi potvrdu za  $(m, 1.1)$  kada primi ovu?
  - Da , zato što je  $1.1 < 1.2$

# Primer: potpuno uredjena grupna komunikacija

\* Do sada su poslate sledeće poruke :

- P1 i P2 su izdali zahteve za ažuriranjem, (m,1.1) i (n,1.2).
- P1 je prosledio svima u grupi potvrdu za poruku (m,1.1).
- P2 je prosledio svima u grupi potvrdu za poruku (m,1.1), (n,1.2).

\* P1 i P2 su primili potvrde od svih procesa iz grupe za poruku (m,1.1).

\* Ažuriranje koje se zahteva porukom  $m$  može biti obavljeno i u P1 i u P2.

## Primer: potpuno uredjena grupna komunikacija

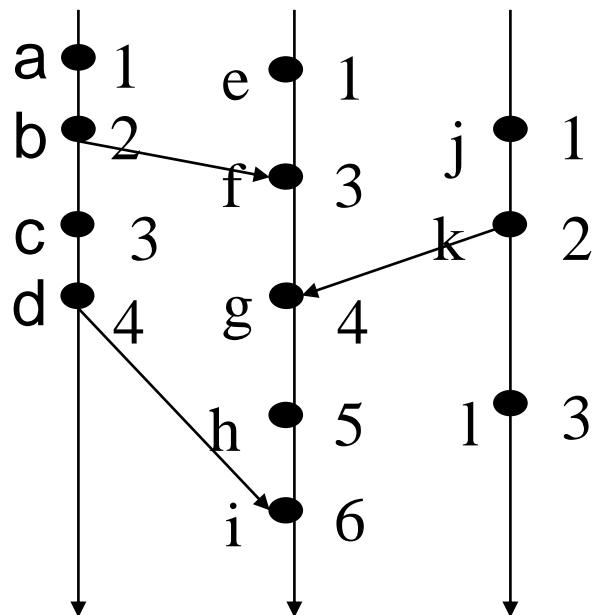
- \* Kada P1 završi sa ažuriranjem  $m$ , on šalje svim procesima u grupi potvrdu za  $(n,1.2)$ .
- \* Kada potvrdu prime P1 i P2, znači da je potvrda za  $(n,1.2)$  stigla od svih procesa iz grupe
- \* Sada ažuriranje koje se zahteva porukom  $n$  može da se obavi u oba procesa, P1 i P2.

# Primer: potpuno uredjena grupna komunikacija

- \* Šta da je postojao treći proces, npr. P3, koji je izdao zahtev za ažuriranjem ( $O, 1.3$ ) skoro u isto vreme kad i procesi P1 i P2?
- \* Algoritam funkcioniše na isti način:
  - P1 neće proslediti ostalim procesima u grupi potvrdu za  $O$ , dok se  $m$  ne obavi.
  - P2 neće poslati potvrdu ostalim procesima u grupi dok se  $n$  ne obavi.
- \* Pošto operacija ne može da se obavi dok iz svih procesa ne stignu potvrde, operacija ažuriranja  $O$  se neće obaviti dok se operacije  $m$  i  $n$  ne obave..

# Vektorski časovnici

- \* Lamportovim markicama se obezbeđuje da ako za dogadjaje  $a$  i  $b$  važi da je  $a \rightarrow b$ , tada važi  $T(a) < T(b)$
- \* Problem je što se na osnovu njih ne može utvrditi koji su dogadjaji medjusobno uslovljeni, a koji su konkurentni.
  - Ako važi da je  $T(a) < T(b)$  ne možemo zaključiti da se  $a$  desilo pre  $b$  (tj.  $a \rightarrow b$ ).



- Primer:  $T(e) = 1$  i  $T(b) = 2$ ; tj.  $T(e) < T(b)$  ali ne možemo reći da je  $e \rightarrow b$
- potreban nam je mehanizam za obeležavanje dogđaja takav da ako važi da je npr.  $T(a) < T(b)$ , tada možemo da zaklučimo da  $a \rightarrow b$

# Vektorski časovnici

- \* **Predloženo rešenje zasniva se na vektorskim časovnicima.**
  - Vektorski časovnik u sistemu sa N procesa je vektor od N celobrojnih elemenata.
  - Svaki proces ima sopstveni vektorski časovnik kojim beleži lokalne događaje.
  - Kao i Lamportove vremenske markice, i vektorski časovnik se šalje sa svakom porukom.
- **Pravila:**
  1. Vektor se inicijalizuje na 0 u svim procesima:  $V_i[j]=0$ , za  $i,j=1,\dots,N$
  2. Proces  $P_i$  inkrementira  $i$ -ti element vektora u lokalnom vektoru pre nego što obeleži događaj:  $V_i[i]=V_i[i]+1$ 
    - Poruka se šalje iz procesa  $P_i$  zajedno sa  $V_i$ .
  3. Kada  $P_j$  primi poruku poredi lokalni vektor sa primljenim, element po element, i postavlja element u lokalnom vektoru na veću od vrednosti

$$V_j(k) = \max\{V_j(k), V_i(k)\}, \quad k = 1, 2, \dots, N$$

# Vektorski časovnici (nast.)

## \* Poređenje vektora

$V = V'$  iff  $V[i] = V'[i]$ , za  $i = 1, \dots, N$

$V \leq V'$  iff  $V[i] \leq V'[i]$ , za  $i = 1, \dots, N$

## \* Za bilo koja dva događaja $e$ i $e'$ važi sledeće

ako je  $e \rightarrow e'$  tada  $V(e) < V(e')$  (kao kod Lamportovog algoritma)

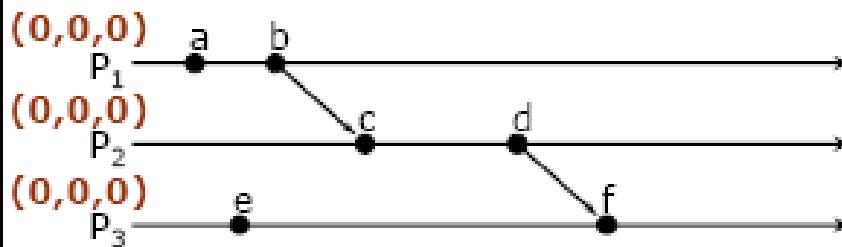
ako je  $V(e) < V(e')$  tada je  $e \rightarrow e'$

## \* Ako se ne može uspostaviti relacija $V(e) \leq V(e')$ ili $V(e) \geq V(e')$ , tada su događaji konkurentni (tj. nisu međusobno uslovljeni)

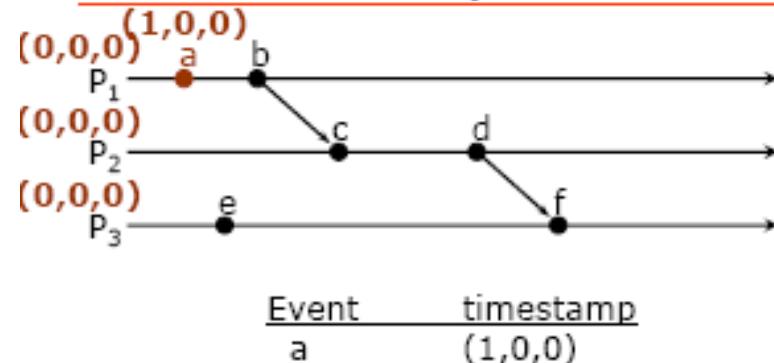
- Za proces  $P_i$ ,  $V_i[i]$  predstavlja broj dogadjaja koji su se desili u procesu  $P_i$ ,
- Ako je  $V_i[j] = k$ , tada  $P_i$  zna da se u  $P_j$  desilo  $k$  dogadjaja

# Primer

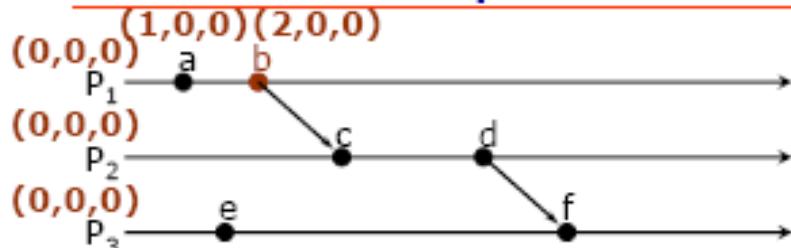
## Vector timestamps



## Vector timestamps

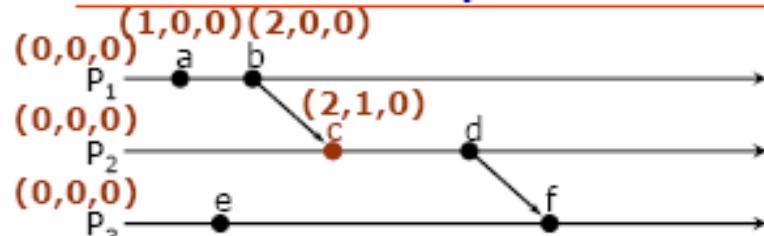


## Vector timestamps



Event	timestamp
a	$(1, 0, 0)$
b	$(2, 0, 0)$

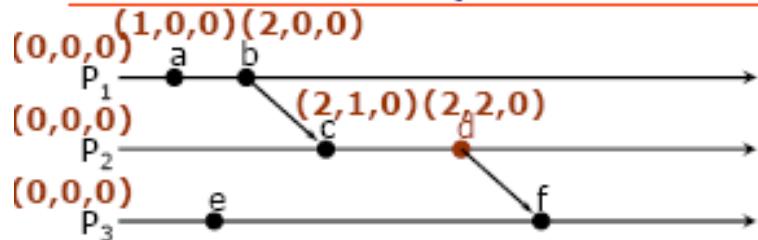
## Vector timestamps



Event	timestamp
a	$(1, 0, 0)$
b	$(2, 0, 0)$
c	$(2, 1, 0)$

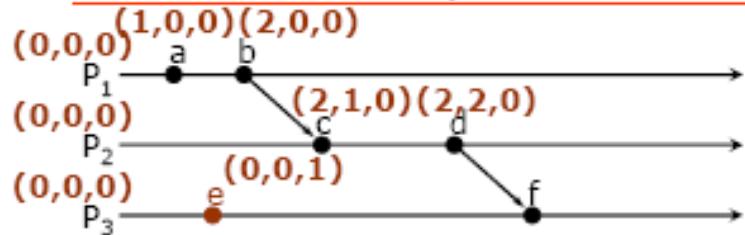
# Primer (nast.)

## Vector timestamps



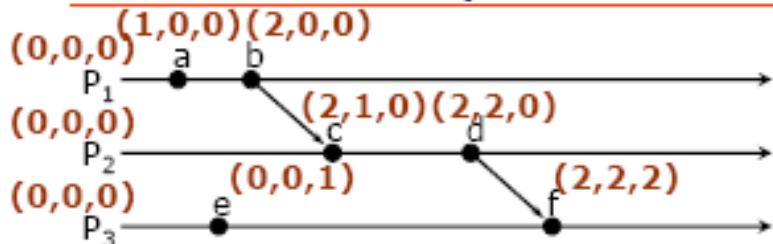
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)

## Vector timestamps



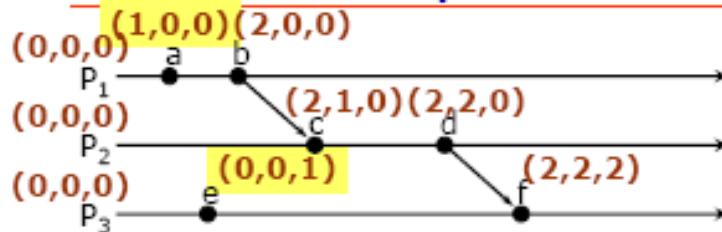
Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)

## Vector timestamps



Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

## Vector timestamps

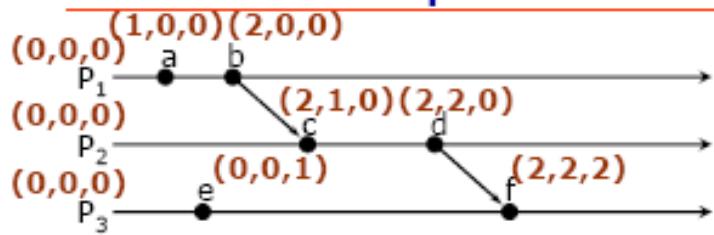


Event	timestamp
a	(1,0,0)
b	(2,0,0)
c	(2,1,0)
d	(2,2,0)
e	(0,0,1)
f	(2,2,2)

concurrent  
events

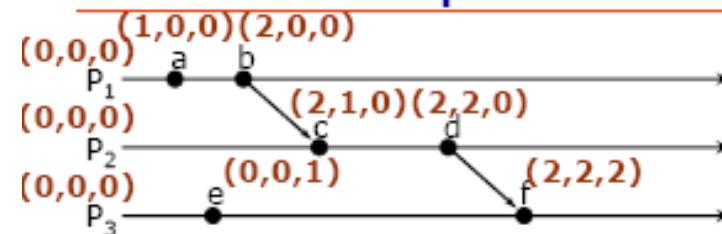
# Primer (nast.)

## Vector timestamps



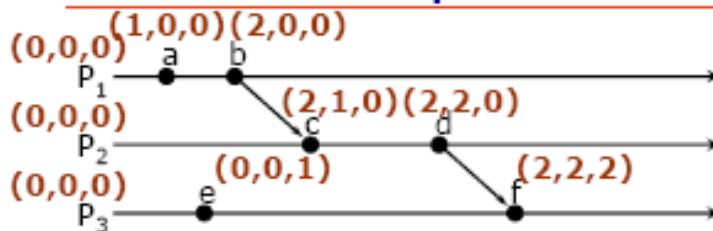
Event	timestamp
a	$(1, 0, 0)$
b	$(2, 0, 0)$
c	$(2, 1, 0)$
d	$(2, 2, 0)$
e	$(0, 0, 1)$
f	$(2, 2, 2)$

## Vector timestamps



Event	timestamp
a	$(1, 0, 0)$
b	$(2, 0, 0)$
c	$(2, 1, 0)$
d	$(2, 2, 0)$
e	$(0, 0, 1)$
f	$(2, 2, 2)$

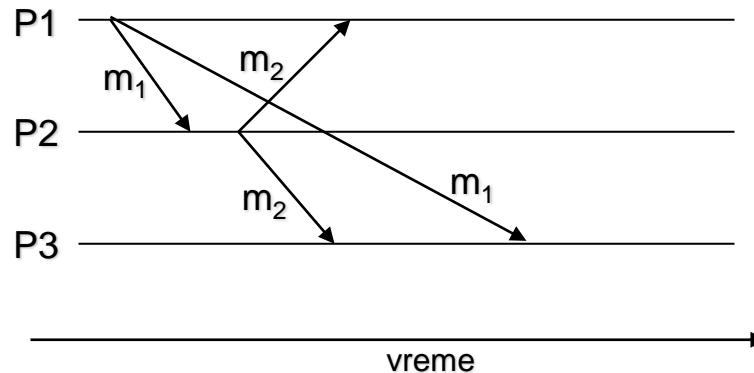
## Vector timestamps



Event	timestamp
a	$(1, 0, 0)$
b	$(2, 0, 0)$
c	$(2, 1, 0)$
d	$(2, 2, 0)$
e	$(0, 0, 1)$
f	$(2, 2, 2)$

# Primer primene: Uređenje međusobno zavisnih događaja

Vektorski časovnici se mogu iskoristiti za korektno uređenje međusobno zavisnih dogadjaja



## \* Poruka m1 se šalje svim procesima u grupi (multicast)

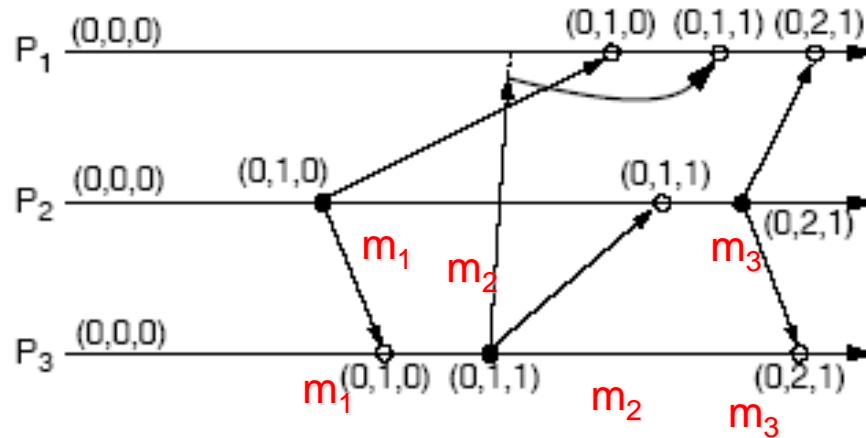
- Poruka m2 je odgovor na poruku m1 i takođe se šalje svim procesima.
- Potrebno je osigurati da svi procesi prvo procesiraju poruku m1 a zatim m2.
  - Ako m2 stigne pre mora se baferovati dok ne stigne poruka m1
  - Uređenje zavisnih događaja
  - Vektorski časovnici se mogu iskoristiti za postizanje željenog uređenja
    - Časovnik se inkrementira samo kod slanja poruke

# Pravila

1. Pre emisije poruke  $m$ , proces  $P_i$  inkrementira svoj vektorski časovnik
  - $V_i[i] = V_i[i] + 1$
  - Markica  $tm = V_i$  se šalje sa porukom  $m$
2. Na prijemnoj strani poruka  $m$  se ne prosleđuje procesu  $P_j$  dok se ne zadovolje sledeća dva uslova
  - A)  $tm[i] = V_j[i] + 1$ 
    - Ovaj uslov obezbeđuje da proces  $P_j$  primi sve poruke koje je poslao proces  $P_i$  pre slanja poruke  $m$
    - Neka je  $V_j[i] = 5$ . Ovo znači da  $P_j$  zna da je  $P_i$  poslao 5 poruka i da od njega očekuje 6. poruku.
    - Ako je  $tm[i] = 8$  tada poruke 6,7 još nisu stigle
  - B)  $tm[k] \leq V_j[k]$ , za svako  $k \neq i$ 
    - Obezbeđuje da proces  $P_j$  primi sve poruke koje je primio  $P_i$  pre slanja poruke  $m$
3. Kada se poruka prosledi procesu, njegov lokalni časovnik se ažurira

$$V_j[i] = \max\{tm[i], V_j[i]\}, \text{ za } i, j = 1, \dots, N$$

# Primer (nast)



Poruka  $m_2$  će biti primljena u  $P_1$  sa  $tm_2=(0,1,1)$ ,

- Lokalni vektor u  $P_1$  je  $V1=(0,0,0)$ ,
- Proveravaju se uslovi A) i B)
  - A)  $tm_2[3]=1=V1[3]+1=0+1=1$ ,
  - B) ali je  $tm_2[2]=1>V1[2]=0$  ( a mora biti  $<$  ili  $=$ ); → poruka mora biti baferovana (proces  $P_1$  nije video poruku iz  $P_2$  koju je video  $P_3$  pre slanja poruke  $m_2$ , pa mora sačekati da primi poruku iz proces  $P_1$ )

Kada stigne  $m_1$  sa  $tm1=(0,1,0)$ , proveravaju se uslovi A) i B)

- A)  $tm1[2]=1=V1[2]+1=0+1=1$
- B)  $tm1[1]=0=V1[1]=0$ ,  $tm1[3]=0=V1[3]=0$

- Uslovi A) i B) su zadovoljeni; poruka se prosleđuje procesu, a lokalni časovnik se ažurira shodno pravilu 3, i postaje  $V1=[0,1,0]$
- Poruka  $m_2$  sada može biti isporučena!

# Sinhronizacija procesa: Uzajamno isključivanje

\* Pristup zajedničkim resursima (uređaji, mem. lokacije) zahteva međusobnu koordinaciju procesa da bi se ostvarilo uzajamno isključivo pravo pristupa zajedničkom resursu.

- U sistemima sa zajedničkom memorijom uzajamno isključivanje se može postići hardverskim mehanizmima (`test_and_set`) ili softverski pomoću semafora ili monitora.
- U DS uzajamno isključivanje se ne može ostvariti semaforima ili monitorima.
  - Jedini način je razmenom poruka.

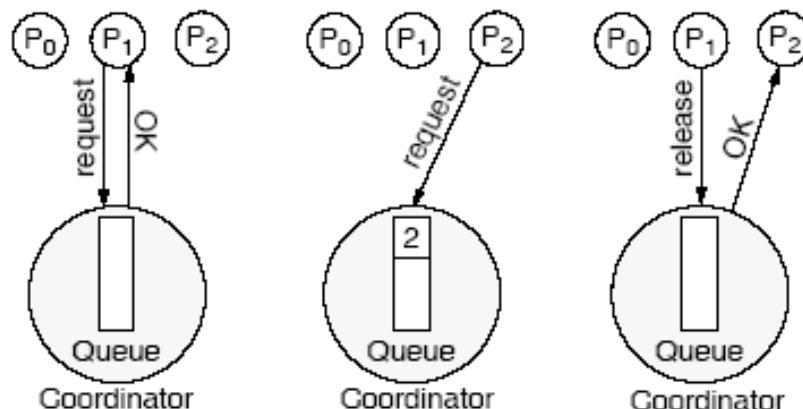
\* Algoritmi uzajamnog isključivanja u DS mogu biti

- Centralizovani
- Distribuirani
- Bazirani na žetonima (token)

# Centralizovani algoritam

- \* Oponaša jednoprocесорски систем
- \* Jedan процес у DS је одабран за координатора

- Kada процес жели да приступи делјивом ресурсу (критичној секцији) шалje захтев координатору идентификујући критичну секцију којој жели да приступи (по имениу или броју).
- Ако ниједан други процес није у датој критичној секцији, координатор шалje дозволу и означава да процес користи дату КС.
- Ако је неки други процес у КС, координатор не шалje одговор, а процес који је упутio захтев se блокира.
- Када процес окончја приступ КС шалje поруку oslobođanja координатору
  - Координатор затим може послati дозволу приступа КС процесу који је bio блокиран чекајући на улазак у КС.



# Centralizovani algoritam (nast.)

## \* Dobre strane

- Jednostavan i lak za implementaciju
- Zahteva razmenu samo tri poruke (request, OK, release)
- Fer: svi zahtevi se opslužuju po redosledu prijema

## \* Loše strane

- Koordinator može postati usko grlo sa stanovišta performansi
- Otkaz koordinatora
  - U slučaju otkaza potrebno je izabrati novog koordinatora
- Proces ne može razlikovati otkaz koordinatora od čekanja na oslobođanje KS
  - da bi se ovaj problem prevazišo, koordinator može poslati poruku odbijanja "permission denied"

# Distribuirani algoritam: Ricart & Agrawala

## \* Koristi multicast (grupnu komunikaciju) i logičke časovnike.

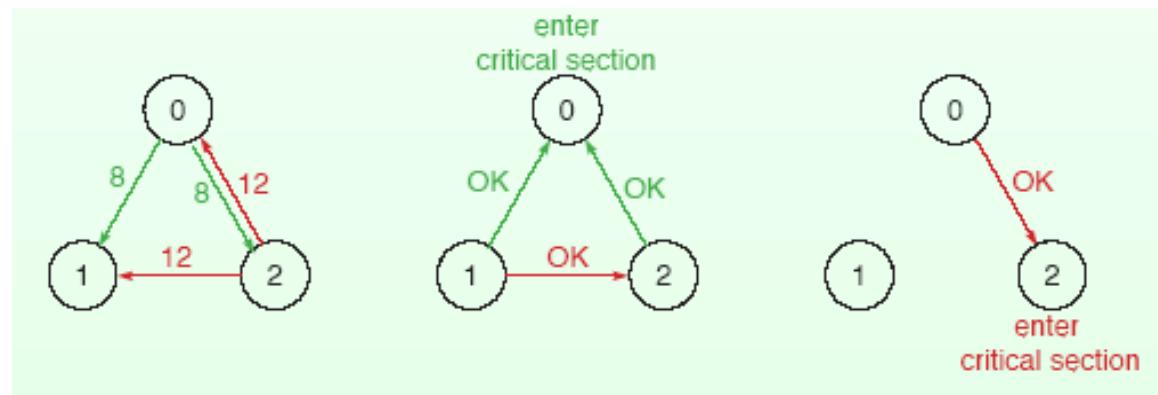
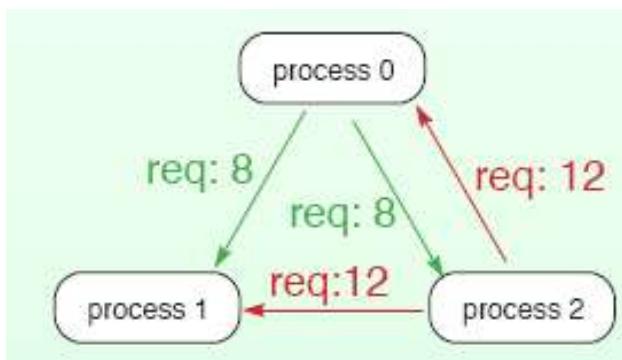
- Proces koji želi da uđe u KS

- Kreira poruku koja sadrži:
  - Identifikator (host\_ID, proces\_ID)
  - Ime resursa
  - Logički časovnik
- Šalje zahtev svim procesima u grupi
- Čeka na dozvolu od svih procesa
- Ulazi u KS (koristi resurs)

- Kada proces primi zahtev

- Ako nije zainteresovan za korišćenje resursa šalje potvrdu (OK) pošiljaocu
- Ako je prijemnik u KS, ne odgovara i smešta zahtev u svoj red čekanja
- Ako je prijemnik upravo poslao zahtev za pristup KS
  - Poredi vremenske markice poslatog i primljenog zahteva
    - » Proces sa manjom markicom pobeđuje
    - » Prijemnik šalje potvrdu (OK)
    - » Ako je prijemnik pobednik, ne šalje potvrdu već smešta zahtev u svoj red čekanja
- Kada okonča pristup KS, šalje potvrdu (OK) svim zahtevima koji su bili u redu čekanja.

# Distribuirani algoritam: Ricart & Agrawala (nast.)

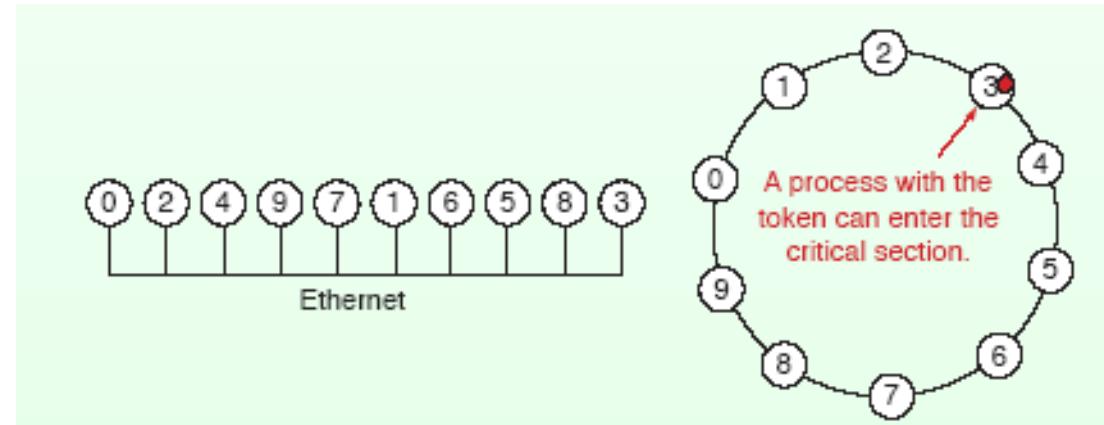


- \* Algoritam garantuje uzajamno isključivo pravo pristupa bez samrtnog zagrljaja (deadlock) i "izgladnjivanja"
- \* Algoritam zahteva razmenu  $2(n-1)$  poruka za pristup KS ( $n-1$  zahtev i  $n-1$  potvrda)
- \* Da bi se dobilo pravo pristupa resursu, potrebno je da se primi potvrda od svih procesa ( $n-1$ )
- \* Greška u bilo kom procesu blokira ceo sistem
- \* Pokazuje da je potpuno distribuirani algoritam uzajamnog isključivanja moguć.
- \* Algoritam se može popraviti tako što će proces uvek slati odgovor na zahtev (dozvolu ili odbijanje)
  - Proces koji je izdao zahtev može da koristi timeout mehanizam i da ponovi zahtev dok ne dobije odgovor ili zaključi da je proces mrtav
- \* Može se ublažiti zahtev i da se dobije potvrda od  $m > n/2$  procesa
  - U ovom slučaju proces koji je dao dozvolu nekom drugom procesu da uđe u KS ne sme dati dozvolu sledećem dok se resurs ne osloboodi.

# Algoritam sa prstenom i žetonima (token ring algoritam)

\* Konstruiše se logički prsten dodelom rednih brojeva procesima

- Logički prsten je softverski generisan i nema nikakve veze sa fizičkim vezama između računara.
- Proces komunicira sa svojim logičkim susedom
  - Svaki proces ima adresu (identifikator) svog desnog suseda u pravcu okretanja kazaljke na satu.



# Token ring algoritam (nast.)

- \* Inicijalno proces 0 dobija žeton za resurs R.
- \* Žeton se prenosi u pravcu kazaljke na satu od procesa do procesa dok se ne dođe do procesa koji zahteva resurs R.
  - Proces koji zahteva resurs R čeka dok ne dobije žeton od svog levog suseda.
    - Kada proces dobije žeton, zadržava ga i pristupa resursu R (ulazi u KS).
    - Kada izadje iz KS prosleđuje žeton sledećem procesu u prstenu.

# Token ring algoritam (nast.)

## \* Osobine

- Samo jedan proces može posedovati žeton u jednom trenutku
  - Uzajamno isključivanje je zagarantovano
- Redosled pristupa resursu je dobro definisan
  - Svakom procesu se garantuje pristup resursu R
- Ne garantuje FIFO redosled pristupa resursu.



- Ako se žeton izgubi (npr. greška u procesu) neophodno je izvršiti rekonfiguraciju prstena da bi se isključio proces koji je otkazao.
  - Potrebno je regenerisati žeton i ponovo ga pustiti u ring (neophodno je startovati algoritam izbora – election algorithm)

# Algoritmi izbora (election algorithms)

\* Distribuirani algoritmi često zahtevaju da jedan proces bude odabran kao koordinator ili server.

- Algoritmi za selekciju koordinatora ili servera se zovu *algoritmi izbora*.
- Polazne prepostavke
  - svaki proces ima jedinstveni identifikator (može biti kombinacija adrese računara i proces ID).
  - Svaki proces zna jedinstvene identifikatore ostalih procesa, ali ne zna da li je odgovarajući proces živ.
- Cilj algoritama izbora je da pronađu aktivni proces sa najvećim identifikatorom i proglase ga za koordinatora.
  - Svi procesi se moraju složiti oko izbora koordinatora.
- Proces izbora se obično odvija u dve faze
  - Selekcija lidera sa najvećim brojem
  - Obaveštavanje svih procesa o pobjedniku

# Bully algoritam

\* Proces,  $P_i$ , koji detektuje da koordinator ne odgovara startuje izbor.

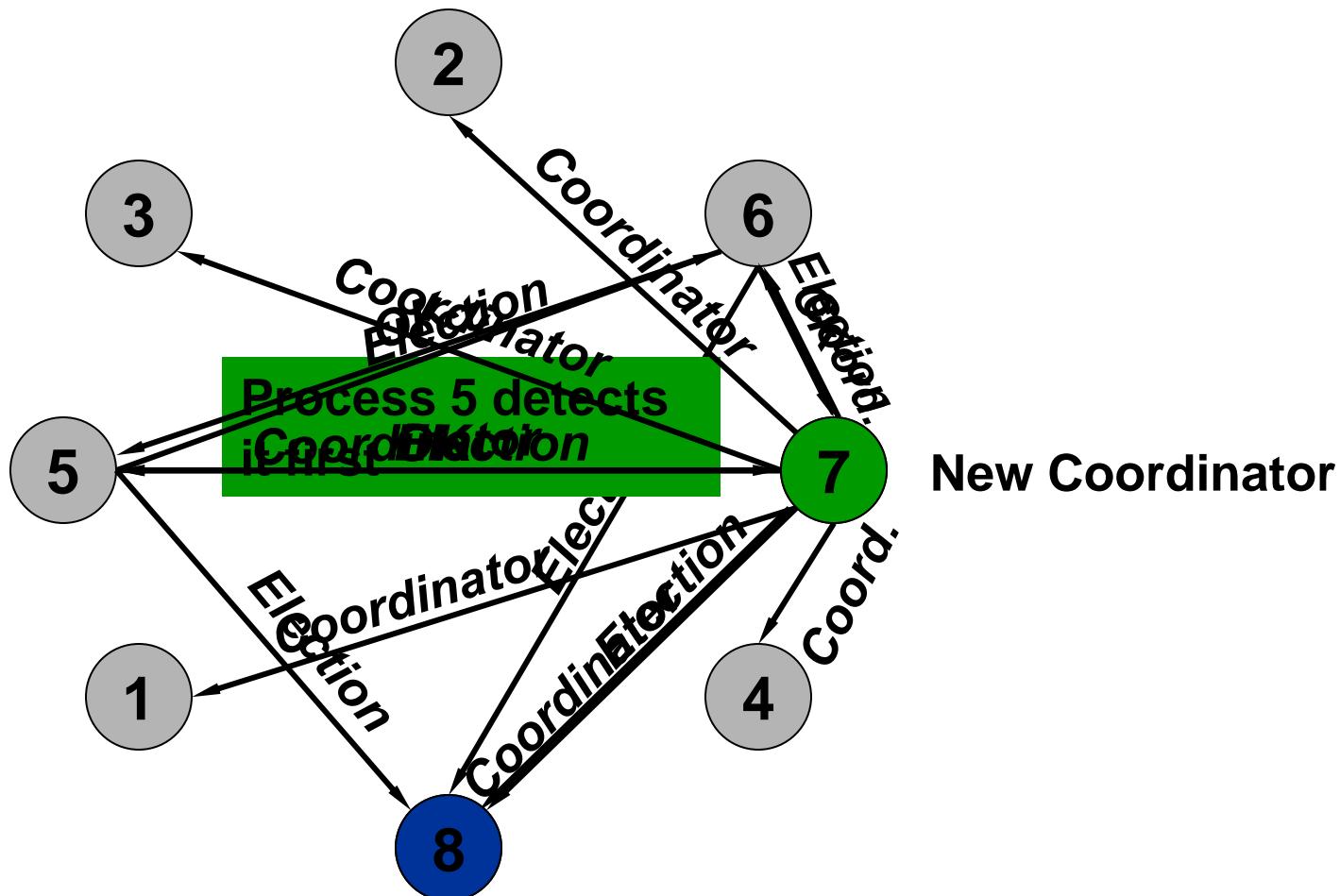
- Izborna poruka se šalje svim procesima sa većim identifikatorom i čeka se odgovor.

➤ Ako u okviru oderđenog vremenskog intervala ne stigne odgovor ni od jednog procesa, to znači da ni jedan proces sa većim identifikatorom nije aktivan i pobednik je proces koji je inicirao ozbole (tj.  $P_i$ ).

–  $P_i$  obaveštava sve procese da je on koordinator

➤ Ako pristigne odgovor na poruku izbora, proces  $P_i$  se povlači i čeka da dobije poruku od novog koordinatora.

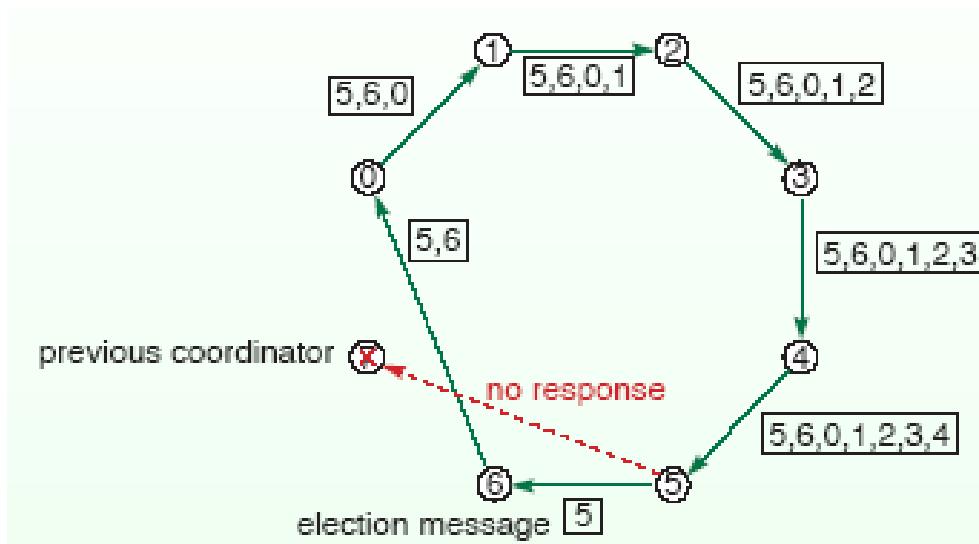
# Bully Algoritam - primer



# Ring algoritam izbora

## \* Procesi formiraju logički prsten.

- Ako neki proces detektuje da koordinator ne funkcioniše, startuje algoritam izbora novog koordinatora.
  - Šalje poruku izbora (election message) sa svojim ID svom susedu u prstenu.
  - Ako sused nije aktivan poruka se prosleđuje sledećem procesu u prstenu.
  - Po prijemu poruke izbora, svaki proces joj pridodaje svoj ID i prosleđuje sledećem procesu.
  - Kada poruka stigne do procesa koji je inicirao izbore, on detektuje u poruci svoj ID.
    - Na osnovu primljene poruke proces zaključuje ko su članovi prstena
    - Šalje novu poruku "KOORDINATOR" sa brojem procesa sa najvećim identifikatortom da obavesti ostalne procese o novom koordinatoru.



# Distribuirani sistemi



Konzistentnost i replikacija

# Razlozi replikacije

## \* Povećanje pouzdanosti

- Npr. ako je fajl sistem repliciran, moguće je nastaviti sa radom ako neka kopija bude narušena jednostavnim komutiranjem na drugu kopiju.

## \* Poboljšanje performansi

- U slučaju velike geografske razuđenosti DS replikacija je poželjna jer smanjuje vreme pristupa podacima (npr. replicirani Web serveri)

## \* Replikacija i keširanje su široko prihvaćene tehnike skaliranja (proširljivosti sistema) uz održanje performansi.

- Skalabilnost se obično odražava na performanse sistema.
  - Repliciranje i smeštanje podataka bliže procesima koji ih koriste može poboljšati performanse redukovanjem vremena pristupa, rešavajući problem skalabilnosti.

# Problemi uzrokovani replikacijom

## \* Replikacija dovodi do problema konzistentnosti kopija.

- Da bi kopije bile usaglašene potrebno je izvršiti ažuriranje tako da se privremena nekonzistentnost ne primeti;
- Ažuriranje kopija dovodi do degradacije performansi, naročito u velikim DS jer povećava saobraćaj kroz mrežu.
- Održavanje kopija konzistentnim može ugroziti skalabilnost.

## \* Kopije su konzistentne ako read operacija nad bilo kojom kopijom uvek vraća isti rezultat.

- Kada se obavlja ažuriranje na jednoj kopiji i sve ostale kopije se moraju ažurirati pre nego što se obavi neka druga operacija.
- Sve replike na globalnom nivou treba da se usaglase kada tačno ažuriranje treba da bude urađeno.
  - Npr. Korišćenjem Lamportovih vremenskih markica ili koordinatora.

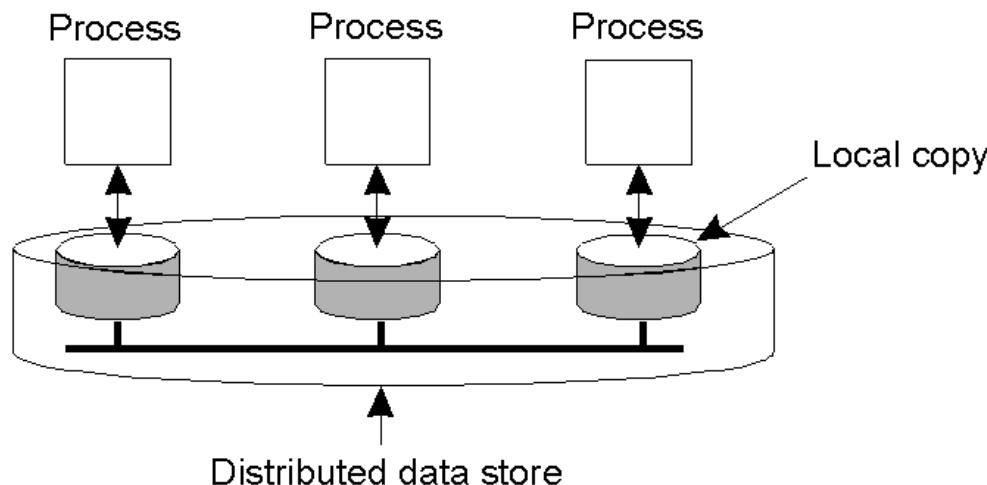
## \* Dilema:

- Problem skalabilnosti može se rešiti korišćenjem replika i keširanja
- Da bi se kopije održale konzistentnim zahteva se globalna sinhronizacija, koja je veoma skupa u pogledu performansi
  - Ilek može biti opasniji od bolesti!
  - Kada i kako će se izvršiti ažuriranje kopija, određuje cenu replikacije.

# Modeli konzistencije

## \* Konzistencija se uvek razmatra u kontekstu read i write operacija

- Operacije se mogu obavljati nad deljivim podacima koji mogu biti zapamćeni u deljivoj memoriji, deljivoj (distribuiranoj) bazi podataka, ili (distribuiranom) fajl sistemu.
  - Termin skladište podataka (data store) se koristi za ovakve podatke.
  - Skladište podataka može fizički biti distribuirano na više mašina.
  - Usvaja se da svaki proces koji može da pristupi skladištu podataka ima lokalnu (ili u blizini) kopiju celog skladišta.
  - Operacija se označava kao write ako menja podatak, u suprotnom se označava kao read.
  - Write operacije se prenose (propagiraju) drugim kopijama



# Modeli konzistencije (nast.)

- \* Problem održavanja konzistentnosti distribuiranih skladišta podataka se javlja zbog konačnog vremena potrebnog za distribuciju ažuriranja (komunikaciono kašnjenje) i nepostojanja globalnog časovnika.
- \* Konzistencija je definisana u kontekstu read i write operacija nad deljivim podacima
  - Proces koji obavlja read operaciju očekuje da će dobiti vrednost koja predstavlja rezultat poslednje write operacije
    - U odsustvu globalnog časovnika teško je odrediti koja write operacija je bila poslednja.
    - Zbog toga je potrebno dati druge definicije, koje nisu bazirane na poznavanju globalnog vremena, što dovodi do niza modela konzistentnosti.
    - Svaki model definiše vrednosti koje read operacija nad podatkovim može vratiti.
    - Koji će se model konzistencije koristiti zavisi od aplikacije

# Modeli konzistencije

- \* Striktna
- \* Sekvencijalna
- \* Kauzalna
- \* FIFO

# Striktna konzistencija

## \* Najjači model konzistencije:

- *Bilo koja operacija nad podatkom X vraća rezultat poslednje write operacije nad X.*

## \* Definicija je prirodna i očigledna, ali implicitno usvaja postojanje globalnog vremena, tako da određivanje poslednje write operacije bude nedvosmisлено.

- Jednoprocesorski sistemi tradicionalno podržavaju striktnu konzistenciju.

➤ Npr. Ako u programu stoje naredbe `a:=1; a:=2; print(a)`, rezultat `print(a)` uvek mora biti 2!.

- U sistemu u kome se podaci nalaze na više mašina, kojima može pristupati više procesa, situacija je mnogo složenija:

➤ Pretpostavimo da proces  $p_i$  ažurira vrednost promenljive  $x$  sa 4 na 5 u trenutku  $t_1$  i šalje novu vrednost svim replikama u grupi (multicast)

➤ Proses  $p_j$  čita (read) vrednost promenljive  $x$  u trenutku  $t_2$  ( $t_2 > t_1$ ).

➤ Proses  $p_j$  treba da pročita vrednost 5, bez obzira kolika je razlika ( $t_2 - t_1$ ).

- \* U sistemu u kome se podaci nalaze na više mašina, kojima može pristupati više procesa, situacija je mnogo složenija:
  - Prepostavimo da proces Pi ažurira vrednost promenljive x sa 4 na 5 u trenutku t1 i šalje novu vrednost svim replikama u grupi (multicast)
  - Proces Pj čita (read) vrednost promenljive x u trenutku t2 ( $t_2 > t_1$ ).
  - Proces Pj treba da pročita vrednost 5, bez obzira kolika je razlika ( $t_2 - t_1$ ).
  - Šta ako je  $t_2 - t_1 = 1$  nsec i koriste se optička vlakna izmedju dve host mašine (na kojima se izvršavaju ova dva procesa) koje se nalaze na rastojanju od 3 metra?.
  - Poruka kojom se zahteva ažuriranje morala bi da putuje 10 puta brže od brzine svetlosti
  - Po Ajnštajnovoj specijalnoj teoriji relativiteta to nije moguće!
- \* Zaključak: striktnu konzistenciju u distribuiranom sistemu je nemoguće postići!
- \* Problem sa striktnom konzistencijom je što se model zasniva na apsolutnom globalnom vremenu.

# Striktna konzistencija (nast.)

## \* Oznake:

- $Wi(x)a$  - označava da proces i obavlja write nad podatkom  $x$ , nakon čega je vrednost  $x=a$
- $Ri(x)b$  – označava da proces i obavlja read nad  $X$  i dobija vrednost  $b$ .
  - Svaki podatak je inicijalno postavljen na NILL

P1:	$W(x)a$
P2:	$R(x)a$

(a)

korektno

P1:	$W(x)a$	
P2:	$R(x)NIL$	$R(x)a$

(b)

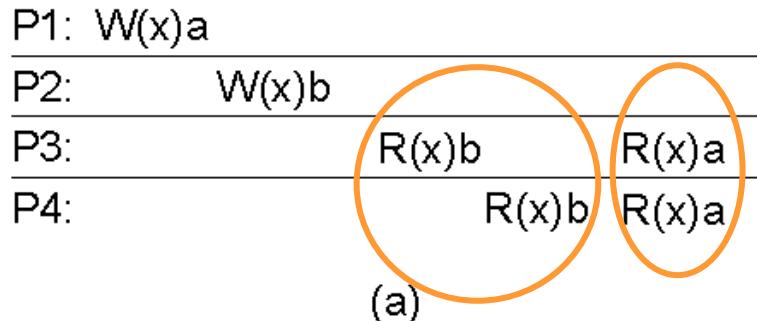
Nije korektno sa stanovišta striktne konzistencije.

# Striktna konzistencija (nast.)

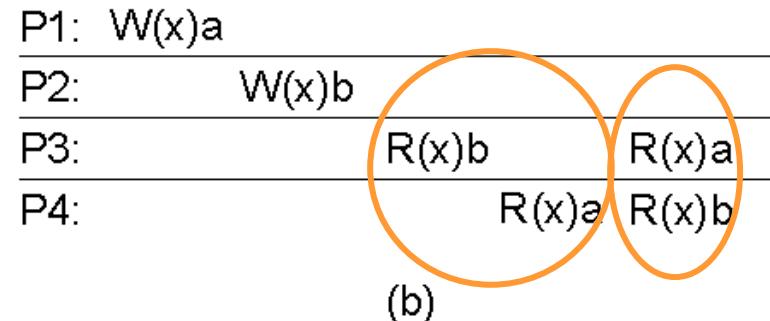
- \* Ako je skladište podataka striktno konzistentno, svi upisi (write) su trenutno vidljivi svim procesima, bez obzira koliko se brzo dešavaju događaji.
- \* Ovakav vid konzistentnosti je praktično nemoguće postići u DS.
- \* Uvode se slabiji modeli, koji nisu bazirani na absolutnom vremenu.
  - Iskustva poakzuju da se programeri mogu izboriti sa slabijim modelima konzistencije.
    - U OS i paralelnim sistemima koriste se kritične sekcije i uzajamno isključivanje.
    - Nikakve pretpostavke o brzini izvršenja procea se ne mogu uvoditi
    - Ako je potrebno procese sinhronizovati, to mora biti ugrađeno u sam program
    - Kritične sekcije omogućavaju da se pristup deljivom resursu linearizuje, tako što će procesi pristupati resursu uzajamno isključivo.
    - Ako je potrebno ostvariti određeni redosled pristupa deljivom resursu to je moguće ostvariti korišćenjem semafora.

# Sekvencijalna konzistencija

- \* Sekvencijalna konzistencija je slabiji model od striktne:
  - Rezultat bilo kog izvršenja je isti kao da su (read i write) operacije svih procesa na skladištu podataka izvršene u nekom sekvencijalnom redosledu i operacije svakog pojedinačnog procesa pojavljuju se u ovoj skvenci u redosledu koji je određen njihovim programom.
- \* Def. kaže da kada se procesi izvršavaju konkurentno na (moguće) različitim mašinama, bilo koje validno preplitanje read i write operacija je prihvatljivo, ali svi procesi vide isto preplitanje operacija.
  - Ovo smo videli na primeru bankovnog računa
    - jedna moguća implementacija je pomoću Lamportovih vremenskih markica.
  - U definiciji se ne spominje vreme, tj. Nema termina "poslednja write operacija"



Sekvencijalno konzistentno skladište



Narušena sekvencijalna konzistentnost  
jer svi procesi ne vide na isti način  
operacije.

# Sekvencijalna konzistencija: Primer

Process P1	Process P2	Process P3	
$x = 1;$ print ( y, z);	$y = 1;$ print (x, z);	$z = 1;$ print (x, y);	write read

## \* Posmatramo tri procesa koji se konkurentno izvršavaju

- $x, y$  i  $z$  su celobrojne promenljive inicijalizovane na 0 i zapamćene u deljivom sekvencijalno konzistentnom skladištu podataka.
- Naredba dodeljivanja odgovara write naredbi, print odgovara read operaciji dva argumenta.
- Moguća su različita prelitanja nizova izvršenja naredbi.
  - Sa 6 naredbi moguće je dobiti  $6! = 720$  različitih prelitanja, mada neka od njih narušavaju programski redosled.
  - Razmotrimo  $5!$  (120) sekvenci koje počinju sa  $x=1$ .
    - Pola od njih ima  $\text{print}(x,z)$  pre  $y=1$  i tako narušava sekvencijalno uređenje programa.
    - Takođe, pola od preostalih permutacija će imati  $\text{print}(x,y)$  pre  $z=1$  i takođe narušava sekvencijalni redosled.
    - Samo  $\frac{1}{4}$  od 120 mogućih sekvenci (tj. 30) je validno.
    - Drugih 30 validnih sekvenci moguće je za  $y=1$  na početku, i još 30 za  $z=1$  na početku, tj. postoji 90 validnih sekvenci izvršenja!

# Primer (nast.)

```
x = 1;  
print (y, z);  
y = 1;  
print (x, z);  
z = 1;  
print (x, y);
```

```
x = 1;  
y = 1;  
print (x,z);  
print(y, z);  
z = 1;  
print (x, y);
```

```
y = 1;  
z = 1;  
print (x, y);  
print (x, z);  
x = 1;  
print (y, z);
```

```
y = 1;  
x = 1;  
z = 1;  
print (x, z);  
print (y, z);  
print (x, y);
```

Prints: 001011

Prints: 101011

Prints: 010111

Prints: 111111

(a)

(b)

(c)

(d)

vreme

Četiri ispravna niza izvršenja naredbi za procese. Vertikalna osa je vreme.

a) Procesi se izvršavaju po redosledu P1, P2, P3.

Sledeća tri primera demonstriraju različita, ali validna preplitanje naredbi u vremenu. Svaki od procesa štampa dve promenljive. Jedine vrednosti koje promenljive mogu da uzmu su početna vrednost (0) i dodeljena vrednost 1. svaki proces proizvodi 2-bitni niz. Vrednosti iza prints su izlazi koji se pojavljuju na izlaznom uređaju.

90 različitih validnih sekvenci naredbi generiše različite rezultate (<64) koji su dozvoljeni pod prepostavkom sekvenčialne konzistencije.

# Primer (nast.)

```
x = 1;  
print (y, z);  
y = 1;  
print (x, z);  
z = 1;  
print (x, y);
```

```
x = 1;  
y = 1;  
print (x,z);  
print(y, z);  
z = 1;  
print (x, y);
```

```
y = 1;  
z = 1;  
print (x, y);  
print (x, z);  
x = 1;  
print (y, z);
```

```
y = 1;  
x = 1;  
z = 1;  
print (x, z);  
print (y, z);  
print (x, y);
```

Prints: 001011

Potpis: 001011

(a)

Prints: 101011

101011

(b)

Prints: 010111

110101

(c)

Prints: 111111

111111

(d)

- \* ako izlaze iz P1, P2 i P3 navedemo u tom redosledu, dobija se 6-to bitni niz koji karakteriše pojedina preplitanja naredbi (potpis)
- \* pošto su moguće vrednosti promenljivih 0 i 1, to znači da ima  $2^6=64$  različitih potpisa, ali svi potpisi (tj. preplitanja naredbi nisu dozvoljeni)
- \* Npr 000000 nije validan potpis jer bi to značilo da se print naredba izvršila pre naredbe dodeljivanja, narušavajući sekvensijalnu konzistentnost (da se naredbe izvršavaju u redosledu definisanom u svakom od procesa)
- \* Šta je sa potpisom 001001?

# Primer2

\* Procesi P1, P2 i P3 se konkurentno izvršavaju.  
Inicijalno su vrednosti svih promenljivih postavljene na nulu. Nakon okončanja procesa, koje vrednosti promenljivih u, v i w nisu moguće pod uslovima sekvensijalne konzistencije?

P1	P2	P3
A=1	u=A	v=B
B=1	w=A	

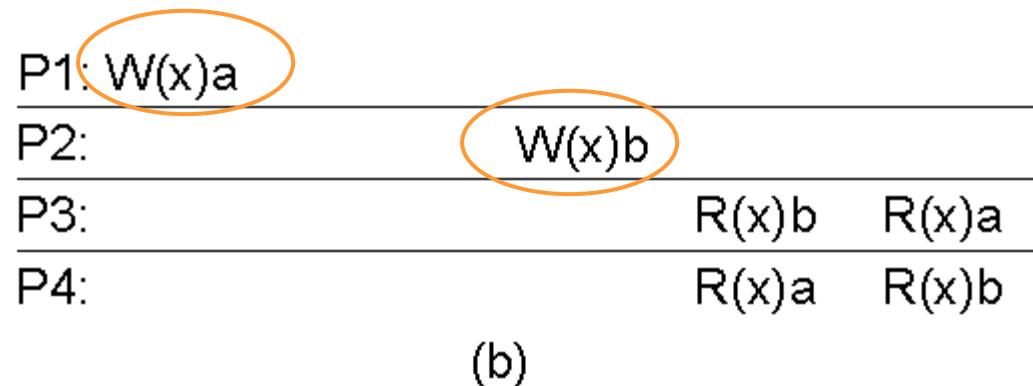
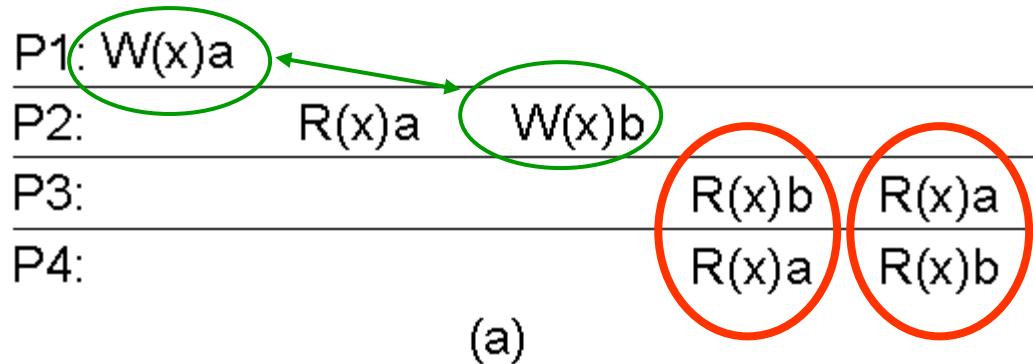
# uslovna (kauzalna) konzistencija

- \* Upisi (write) koji su potencijalno uslovljeni moraju se videti u svim procesima u istom redosledu
- \* Konkurentni upisi se mogu videti u različitom redosledu u različitim procesima

P1:	W(x)a		W(x)c
P2:	R(x)a	W(x)b	
P3:	R(x)a		R(x)c R(x)b
P4:	R(x)a		R(x)b R(x)c

- $W(x)a \rightarrow R(x)a$ ,  $R(x)a \rightarrow W(x)b$ ,  $\Rightarrow W(x)a \rightarrow W(x)b$  (tj. ova dva upisa su potencijalno uslovljena)
- $W(x)c$  i  $W(x)b$  nisu potencijalno uslovljeni, pa rezultate ovih upisa različiti procesi mogu videti u različitom redosledu!
- ovakvo skladište podataka je uslovno konzistentno, ali ne i sekvenčijalno!

# uslovna konzistencija (nast.)



- a) nije ispoštovana uslovna konzistencija
- b) ispoštovana je uslovna konzistencija jer  $W(x)a$  i  $W(x)b$  nisu medjusobno uslovljeni
- da bi se implementirala uslovna konzistencija mogu se koristiti vektorski časovnici za uređenje medjusobno zavisnih dogadjaja

# FIFO konzistencija

- \* Upisi koje obavi jedan proces se vide od strane drugih procesa po redosledu u kome su izdati.
- \* ... ali upisi različitih procesa mogu se videti u različitom redosledu u različitim procesima.
- \* drugim rečima, ne postoji nikakve garancije o tome kako različiti procesi vide upise drugih procesa, osim što dva ili više upisa iz istog izvora moraju stići po redosledu.
- \* Jednostavna implementacija:
  - Svaki proces dodaje poruci ažuriranja : (proces id, redni broj)
  - Svi ostali procesi primenjuju ažuriranja po redosledu u kome su ona obavljena u izvornom procesu.

# FIFO konzistencija

P1:  $W(x)a$

---

P2:  $R(x)a$   $W(x)b$   $W(x)c$

---

P3:  $R(x)b$   $R(x)a$   $R(x)c$

---

P4:  $R(x)a$   $R(x)b$   $R(x)c$

Ni jedan od prethodnih modela konzistencije ne dozvoljava ovakav redosled dogadjaja

# FIFO konzistencija

Process P1	Process P2	Process P3
x = 1; print ( y, z);	y = 1; print (x, z);	z = 1; print (x, y);

```
x = 1;  
print (y, z);  
y = 1;  
print(x, z);  
z = 1;  
print (x, y);
```

Prints: 00

(a)

```
x = 1;  
y = 1;  
print(x, z);  
print ( y, z);  
z = 1;  
print (x, y);
```

Prints: 10

(b)

```
y = 1;  
print (x, z);  
z = 1;  
print (x, y);  
x = 1;  
print (y, z);
```

Prints: 01

(c)

\* naredbe označene masnim slovima generišu prikazane izlaze.

- Konkatenacijom izlaza dobija se vrednost 001001, koja nije dozvoljena u slučaju striktne konzistencije.

# Sekvencijalna naspram FIFO konzistencije

- \* U oba slučaja redosled izvršenja nije determinisan
- \* Sekvencijalna: procesi na isti način vide dogadjaje
- \* FIFO: procesi mogu videti dogadjaje u različitom redosledu (sem onih koji potiču iz istog procesa)

Process P1	Process P2
$x = 1;$ <code>if (<math>y == 0</math>) kill (P2);</code>	$y = 1;$ <code>if (<math>x == 0</math>) kill (P1);</code>

Prepostavimo da je inicijalno  $x = y = 0$

Sekvencijalna: mogući ishodi: P1 ili P2 ili ni jedna proces nije ubijen

FIFO: moguće je da oba procesa budu ubijena jer različiti procesi mogu videti operacije u različitom redosledu!

# Slaba konzistencija

- \* Sekvencialna konzistencija je dosta restriktivna za mnoge aplikacije jer zahteva da sve write operacije jednog procesa budu vidljive svuda u sistemu.
  - Npr. dok se jedan proces nalazi u kritičnoj sekciji i modifikuje podatke u repliciranoj bazi podataka, sve write operacije tog procesa moraju da budu prosleđene svim kopijama, mada dok je proces u kritičnoj sekciji drugi proces ne može pristupiti deljivim podacima.
  - Bolje rešenje je dozvoliti procesu da okonča izvršenje kritične sekcije, a onda obezbediti da konačni rezultat bude prosleđen svima u sistemu.
    - Ovo se može postići korišćenjem eksplicitne sinhronizacije pomoću sinhronizacionih promenljivih.
    - Na sinhronizacionoj promenljivoj,  $S$ , se može obavljati samo jedna operacija synchronize ( $S$ ) , koja sinhronizuje sve lokalne kopije skladišta podataka.
    - Proces P obavlja operacije samo nad svojom lokalnom kopijom skladišta podataka i nema garancija kada će modifikacije biti vidljive drugim procesima.
    - Tek kada se skladište podataka sinhronizuje, svi lokalni upisi procesa P se prosleđuju ostalim kopijama skladišta podataka, a upisi drugih procesa donose u P-ovu kopiju.
  - Model slabe konzistencije ostvaruje konzistenciju nad grupom operacija, a ne nad individualnim read i write operacijama.

# Slaba konzistencija

```
int a, b, c, d, e, x, y;          /* variables */
int *p, *q;                      /* pointers */
int f( int *p, int *q);          /* function
prototype */

a = x * x;
register */                      /* a stored in
b = y * y;                      /* b as well */
c = a*a*a + b*b + a * b;
d = a * a * c;
p = &a;
q = &b;
e = f(p, q)                      /* used later */
                                    /* used later */
                                    /* p gets address of a */
                                    /* q gets address of b */
                                    /* function call */
```

I kod centralizovanih (jednoprocesorskih) sistema postoji privremena nekonzistentnost memorije.

# Distribuirani sistemi



Konzistentnost sa stanovišta klijenta (client-centric)

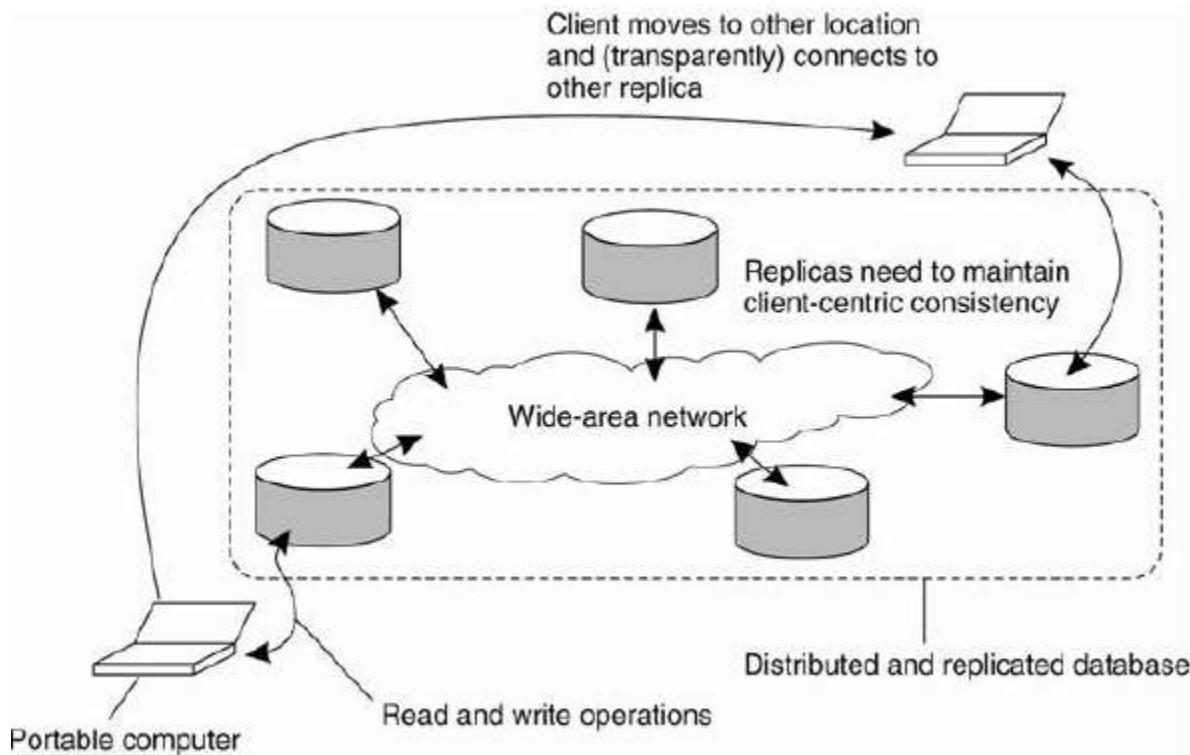
Pozicioniranje replika

Protokoli konzistencije

# konzistentnost sa stanovišta klijenta

## \* Client-centric consistency

- ovde je naglasak na održavanju konzistentnog pogleda na skladište podataka sa stanovište pojedinačnog klijentskog procesa
  - klijent je mobilan i može sa različitim lokacijama čitati/modifikovati kopije skladišta podataka.
- postavlja se pitanje koje promene klijent može videti kada promeni lokaciju



# Client-centric modeli konzistencije

\* postoje četiri različita modela konzistencije posmatrano u odnosu na klijenta:

- monotona čitanja (monotonic reads)
- monotoni upisi (monotonic writes)
- čitaj svoje upise (read your writes)
- upisi nakon čitanja (writes follow reads)

\* Označavanje

- $x_i[t]$  označava verziju podatka  $x$  u lokalnoj kopiji  $L_i$  u trenutku  $t$
- $WS\ x_i[t]$  označava skup write operacija na lokalnoj kopiji  $L_i$  koje su dovele do verzije  $x_i$  promenljive  $x$  (u trenutku  $t$ )
- ako se set operacija  $WS\ x_i[t_1]$  obavio i na lokalnoj kopiji  $L_j$  a zatim i novo ažuriranje na lokaciji  $L_j$  to se označava sa  $WS(x_i[t_1], x_j[t_2])$ 
  - ako je vremenski redosled očigledan, oznaka vremena se može izostaviti

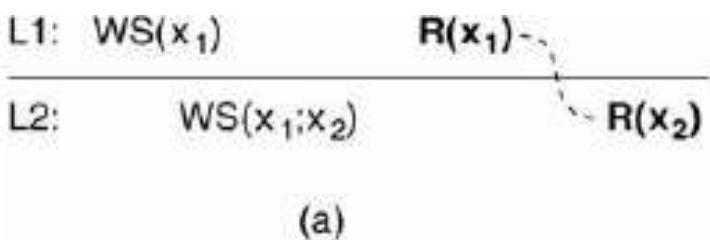
# monotona čitanja (monotonic reads)

\* Ako proces pročita vrednost podatka  $x$  na jednoj lokaciji, svako novo čitanje tog podatka će vratiti istu ili noviju vrednost bez obzira na kojoj lokaciji se obavlja čitanje.

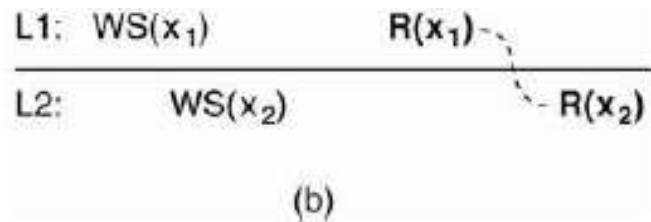
- Primer: čitanje email-a dok je klijent u pokretu

➤ svaki put kada se klijent poveže na drugi e-mail server klijent vidi sve mailove koje je video prethodnog puta, a može i novije

- a) Proces P prvo čita  $x$  na lokaciji L1 i vidi vrednost  $x_1$ 
  - ova vrednost je rezultat operacije  $WS(x_1)$  koja je obavljena na lokaciji L1
  - Kasnije P čita  $x$  sa lokacije L2, što je označeno sa  $R(x_2)$
  - da bi se garantovala konzistentnost, skup operacija  $WS(x_1)$  mora da budu propagiran i na lokaciju L2 pre drugog čitanja:
    - na lokaciji L2 je nakon ažuriranja  $WS(x_1)$  obavljeno ažuriranje  $WS(x_2)$  što je označeno sa  $WS(x_1,x_2)$



# monotona čitanja (monotonic reads)



\* situacija u kojoj konzistencija tipa monotonic-read nije garantovana

- nakon što je proces P pročitao  $x=x_1$  na lokaciji L1, kasnije čita  $x=x_2$  sa lokacije L2
  - na lokaciji L2 su obavljene samo operacije  $WS(x_2)$
  - nema garancija da ovaj skup  $WS(x_2)$  sadrži sve operacije koje su bile obavljene u skupu  $WS(x_1)$

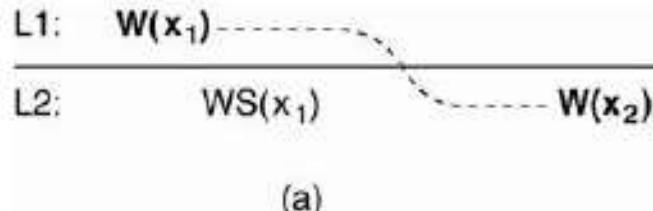
# Monotoni upisi (Monotonic Writes)

\* kod ovog modela konzistencije važe sledeći uslovi:

- Ako proces P obavlja modifikaciju (write) podatka x na lokaciji L1, proces P može obaviti modifikaciju podatka x na lokaciji L2 tek nakon što se na toj lokaciji obavi ažuriranje sa lokacije L1.

\* Primer

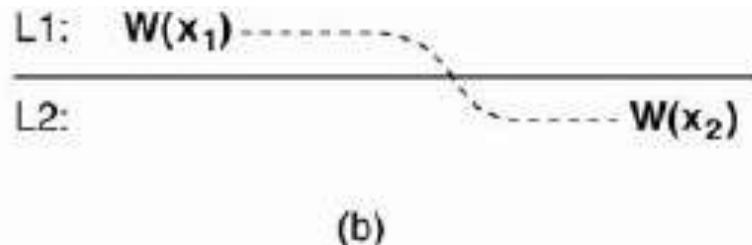
- proces P obavlja write operaciju na lokaciji L1,  $X(x_1)$
- kasnije proces P obavlja drugu write operaciju nad x na lokaciji L2,  $W(x_2)$
- da bi se ispoštovala konzistencija "monotoni upisi" prethodna write operacija sa L1 mora biti prosleđena na lokaciju L2
  - to je označeno sa  $WS(x_1)$  na lokaciji L2 pre obavljanja  $W(x_2)$



(a)

Primer: ažuriranje softvera

# Monotoni upisi (Monotonic Writes)



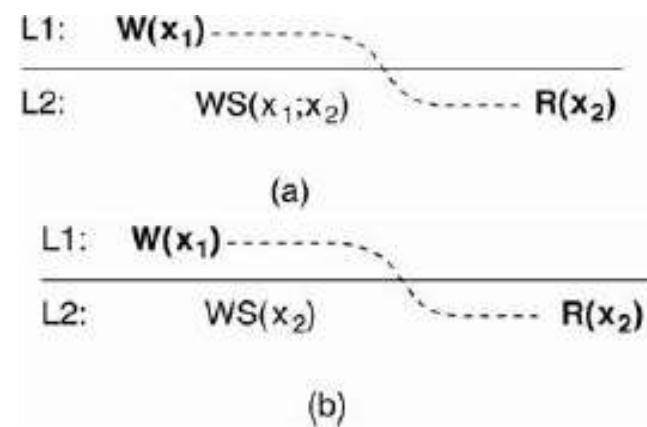
(b)

\* situacija u kojoj konzistencija monotonic-write nije zagarantovana

- nedostaje propagiranje ažuriranja  $W(x_1)$  sa lokacije L1 na lokaciju L2
- nema garancija da kopija podatka  $x$  na kojoj je obavljena druga write operacija ima istu ili noviju vrednost od  $W(x_1)$  sa lokacije L1

# Čitaj svoje upise (Read-your-writes)

- \* model konzistencije koji je veoma blizak modelu monotonog čitanja
- \* skladište podataka obezbeđuje read-your-writes konzistenciju ako je zadovoljeno sledeće:
  - efekat write operacije koju obavlja proces P nad podatkom x će uvek biti viđen kasnijim čitanjem podataka x od strane istog procesa



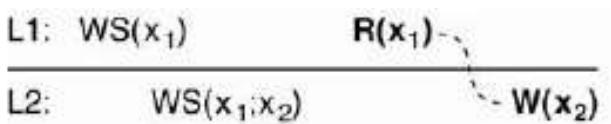
- a) Process P obavlja write operaciju  $W(x_1)$  na lokaciji L1, a kasnije obavlja čitanje sa lokacije L2
  - Read-your-writes consistencija garantuje da će efekat write operacije  $W(x_1)$  biti vidljiv od strane kasnijih čitanja
    - to je označeno sa  $WS(x_1; x_2)$ , koje ukazuje da je  $W(x_1)$  obuhvaćeno sa  $WS(x_2)$
- b)  $W(x_1)$  nije obuhvaćeno sa  $WS(x_2)$ , što znači da efekat prethodne write operacije procesa P nije propagiran na lokaciju L2

# upis sledi čitanje (Writes Follow Reads)

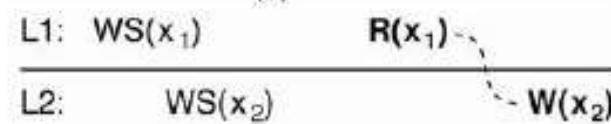
\* skladište podataka obezbeđuje writes-follow-reads konzistenciju ako važi sledeće:

- write operacija koju je obavio proces P nad podatkom x nakon prethodnog čitanja tog podatka se obavlja nad istom ili novijom vrednošću podatka x

- a) proces čita x na lokalnoj kopiji L1
  - write operacija WS(x1) koja je dala pročitanu vrednost R(x1) se pojavljuje i u skupu WS(x1;x2) na lokaciji L2 gde isti proces kasnije obavlja upis
  - garantuje da je W(x2) urađeno na kopiji koja je konzistentna sa kopijom na lokaciji L1
- b) nema garancija da je W(x2) urađeno na kopiji koja je konzistentna sa kopijom na lokaciji L1



(a)



(b)

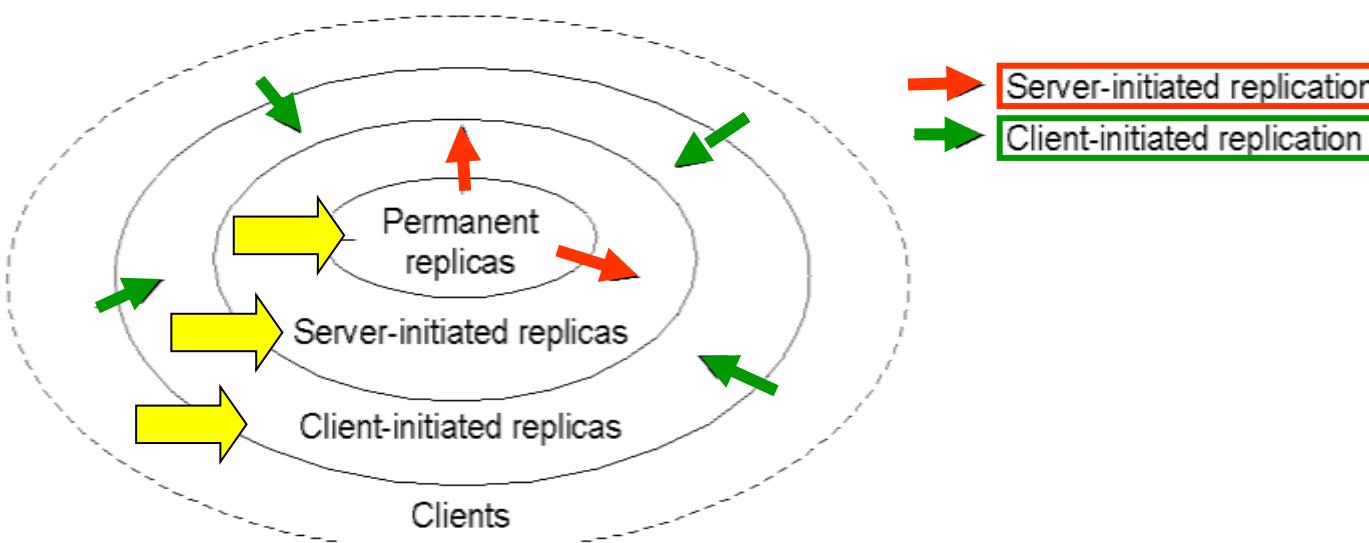
# Pozicioniranje replika i protokoli konzistencije



vrste replika  
protokoli za ažuriranje replika

# Replikiranje sadržaja i pozicioniranje replika

- \* Ključno pitanje u DS koji podržava replikaciju je gde, kada i ko može obavljati replikiranje i koji se mehanizmi koriste za održavanje konzistencije.
- \* Tri tipa replika:
  - Permanentne replike
  - Replike inicirane od strane servera
  - Replike inicirane od strane klijenta



Logička organizacija tri vrste replika

# Permanentne replike

\* Mogu se posmatrati kao početni skup replika koje obrazuju distribuirano skladište podataka.

- Broj permanentnih replika je mali.

➤ Primer: Distribucija Web sajta se javlja u dva oblika

- Repliciranje fajlova koji čine Web sajt na ograničenom broju servera koji se nalaze na jednoj lokaciji (klaster servera). Kada stigne neki zahtev on se prosleđuje jednom od servera, najčešće po kružnom redosledu (round robin)
- Mirroring – Web sajt se kopira na ograničeni broj servera (mirror serveri) koji su geografski dislocirani u Internetu. Klijent odabira jedan od ponuđenih mirror servera.

➤ Baze podataka takođe mogu biti replicirane na više servera koji zajedno formiraju klaster radnih stanica, ili na veći broj geografski distribuiranih sajtova

# Replike inicirane od strane servera

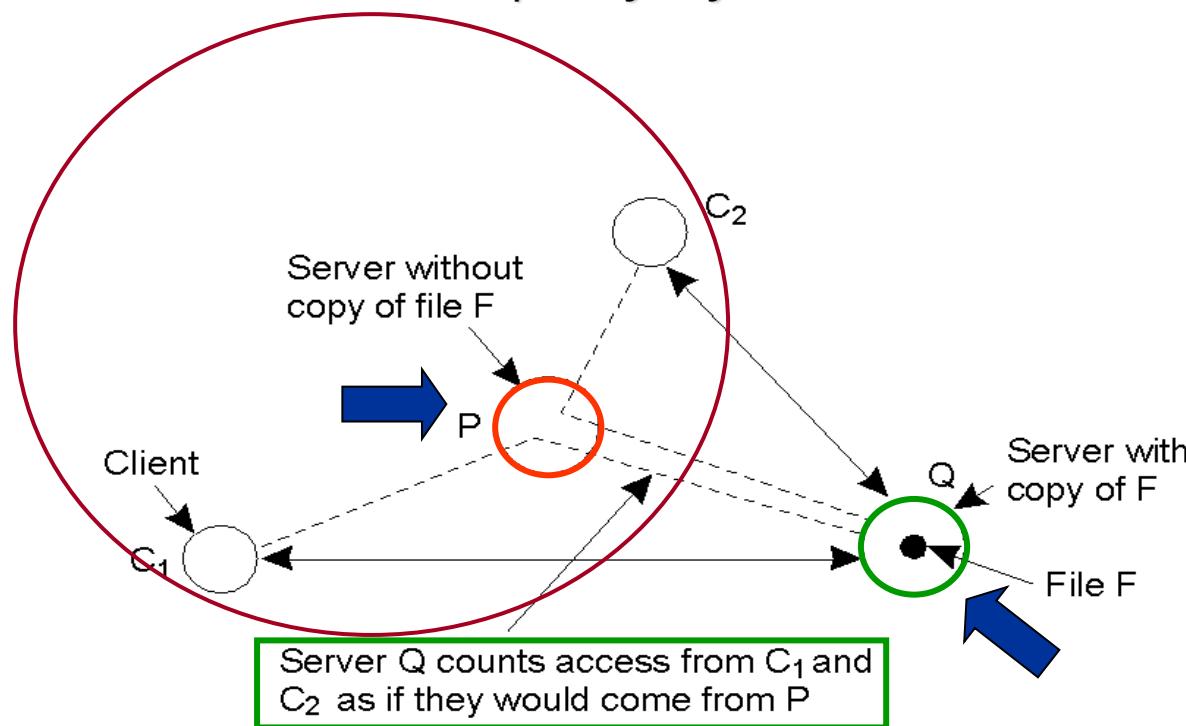
\* Formiraju se na inicijativu vlasnika skladišta podataka da bi se poboljšale performanse (vreme odziva) ili rasteretio server ako u nekom trenutku dolazi veliki broj zahteva sa neke udaljene lokacije.

- to su privremene replike postavljene u regione iz kojih dolazi veliki broj zahteva
- Svaki server vodi računa o broju pristupa određenom fajlu i odakle dolaze.
  - Ako je u određenom trenutku broj zahteva sa neke lokacije veliki, svrsishodno je instalirati privremene replike u regionima odakle dolaze zahtevi. (Takve replike zovu se i push replike).
  - Web hosting servis – nudi relativno statican skup servera u Internetu koji mogu sadržati fajlove i podržati pristup fajlovima koji pripadaju drugom serveru.
    - Da bi se obezbedile optimalne k-ke Web hosting usluge mogu dinamički replicirati fajlove na servere gde su ti fajlovi potrebni da bi se poboljšale performanse, tj. bliže grupi klijenata.

# Replike inicirane od strane servera (nast.)

- Gde i kada replike treba kreirati ili obrisati?

- Algoritam dinamičke replikacije uzima dve stvari u obzir: broj obraćanja fajlu i odakle dolaze zahtevi.
- Za svakog klijenta, C, server može odrediti koji je najbliži server u Web hosting servisu.
  - Ako dva klijenta, C<sub>1</sub> i C<sub>2</sub>, imaju isti najbliži server P, svi zahtevi za pristup fajlu F u serveru Q, od strane C<sub>1</sub> i C<sub>2</sub> se jedinstveno registruju u serveru Q:  $\text{cnt}_Q(P,F)$ , tj. kao da dolaze od servera P.
  - Ako je broj obraćanja fajlu F na serveru Q viši od praga replikacije  $\text{rep}(Q,F)$  donosi se odluka da se izvrši replikacija fajla F na server P.



# Replike inicirane od strane servera (nast.)

- Kada broj obraćanja fajlu F na serveru P padne ispod praga  $\text{del}(P,F)$ , fajl može biti uklonjen sa servera P, pod uslovom da to nije posledja (jedina) kopija tog fajla.
    - Na taj način se broj replika redukuje.
    - Vodi se računa da bar jedna replika svakog fajla postoji.
  - Ako je broj zahteva za pristup nekom fajlu između praga brisanja i praga replikacije, fajl samo može migrirati sa jednog servera na drugi
- \* Serverski inicirane replike se najčešće koriste kao read-only kopije za klijente.
- \* Permanentne replike su jedine koje se mogu menjati da bi se sistem održao konzistentnim.

# Replike inicirane od strane klijenta

\* Važnu klasu replika čine replike inicirane od strane klijenta.

- Poznate su pod nazivom **klijent keš**.

- Keš je lokalna memorija koja se koristi od strane klijenta da privremeno zapamti kopiju podataka koje je klijent upravo zahtevao (i dobio).
- Keš može biti lociran na klijent mašini ili na posebnoj mašini u okviru klijentovog LANa.
- Upravljanje kešom je u potpunosti prepušteno klijentu. Skladište podataka iz koga su podaci preuzeti nema nikakvu obavezu u održavanju konzistencije keša.
  - Klijent može uz učešće skladišta utvrditi da li je kopija ustajala ili važeća.
- Klijentski keš se koristi da bi se smanjilo vreme pristupa.
  - Kada klijent želi da pristupi nekim podacima, on se povezuje na najbližu kopiju skladišta podataka, odakle uzima podatke koje želi da pročita ili gde smešta podatke koje je modifikovao.
    - » Ako je u pitanju samo čitanje podataka klijentu se može omogućiti da zapamti podatke u keš.
    - » Sledeći put kada klijent zatraži čitanje istih podataka, može ih preuzeti iz svog lokalnog keša.
    - » Ovo funkcioniše dobro sve dok se replicirani podaci ne promene

# Distribucija ažuriranja

## \* Šta se propagira kod ažuriranja?

- Samo obaveštenje o obavljenom ažuriranu

- Svodi se na invalidaciju ostalih kopija

- Prednost: mali saobraćaj kroz mrežu

- Protokoli sa invalidacijom dobro funkcionišu kada je broj ažuriranja mnogo veći od broja čitanja (mnoga ažuriranja mogu biti obavljena lokalno, bez ponovne invalidacije) .

- Podaci koji su ažurirani

- Ažurirani podaci se prenose replikama.

- Dobro funkcioniše kada je odnos čitanje-upis relativno visok (tj. broj čitanja je mnogo veći od broja upisa).

- Operacija koje su izazvale ažuriranje.

- Ne prenose se podaci, već operacije koje svaka replika treba da obavi (ovakav prilaz poznat je pod nazivom **aktivno repliciranje**)

- Ažuriranja često mogu biti obavljena uz minimalni saobraćaj

- Zahteva više procesorskog vremena, naročito ako su operacije kompleksne.

# Ko inicira ažuriranje?

## \* Ažuriranje inicirano od strane servera (push prilaz)

- Ažuriranje se prenosi replikama na inicijativu servera, mada replike to nisu zahtevale.
- Ovaj prilaz se koristi između permanentnih i serverski iniciranih replika, tj. kada je potrebno ostvariti visok nivo konzistencije.

## \* Ažuriranje na zahtev klijenta (pull prilaz)

- Server ili klijent zahtevaju od drugog servera da mu prosledi ažuriranje ako je postojalo.
  - Npr. uobičajena strategija koja se primenjuje kod Web keševa, je prvo provera da li su keširani podaci još uvek validni.
    - Kada keš primi zahtev za čitanjem podataka koji se još uvek nalaze u kešu, on prvo kontaktira originalni Web server da proveri da li su podaci u međuvremenu bili modifikovani (klijent proziva server da bi ustanovio da li su podaci bili modifikovani).
      - » **client to server: {data X, timestamp ti , OK?}**
      - Ako su podaci bili modifikovani, prenose se sa servera u keš, a zatim prosleđuju klijentu
        - » **server to client: OK or {data X, timestamp ti+k}**

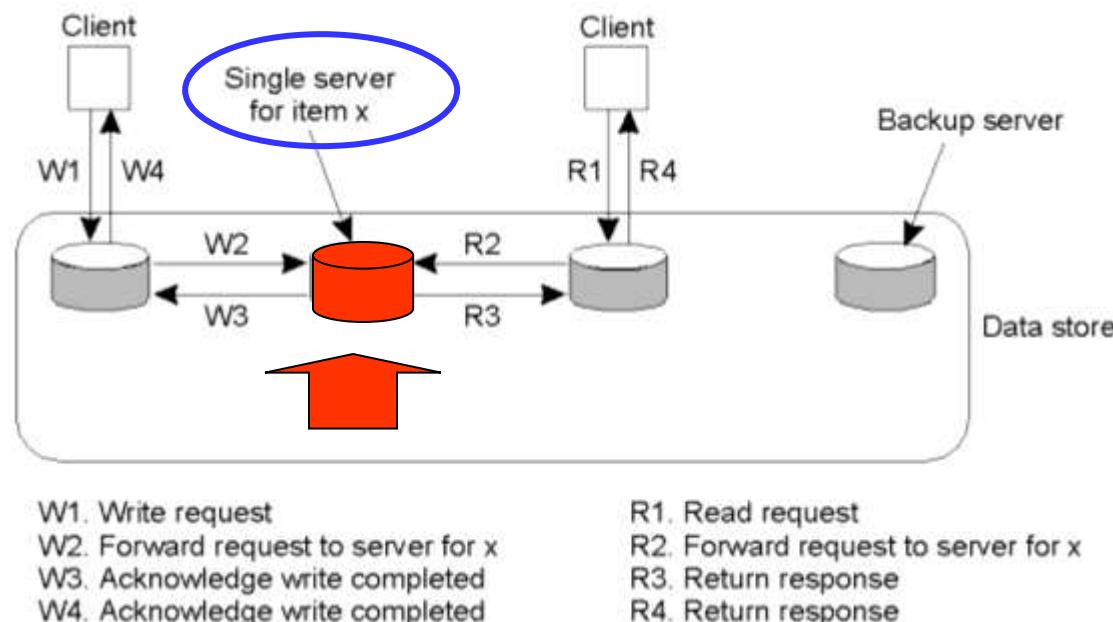
# Protokoli konzistencije

- \* Protokoli konzistencije opisuju implementaciju određenog modela konzistencije.
- \* Tri vrste:
  - Protokoli zasnovani na postojanju primarne kopije (i backup kopija)
    - Write operacija se može izvršiti samo na primarnoj kopiji
  - Protokoli sa više ravnopravnih replika.
    - write operacija može biti inicirana u bilo kojoj replici.
  - Keš koherentni protokoli
    - inicirani od strane klijenta, a ne servera
- \* Ako se zahteva postizanje sekvenčialne konzistencije najčešće se koriste protokoli zasnovani na primarnoj kopiji.
  - Kod ovih protokola svaki podatak X u skladištu podataka ima pridruženi primar koji je zadužen za koordinaciju upisa u X.
    - Može se napraviti razlika u odnosu na to da li je primar fiksiran na udaljenom serveru - **Remote-write protokoli** (protokoli sa udaljenim upisom)
    - ili se write operacija može obaviti lokalno nakon što se primar pomeri na proces koji je inicirao write - **Local-write protokoli** (protokoli sa lokalnim upisom)

# Remote-write protokoli

\* Najjednostavniji protokol baziran na postojanju primarne kopije je protokol kod koga se sve read i write operacije obavljaju na udaljenom serveru.

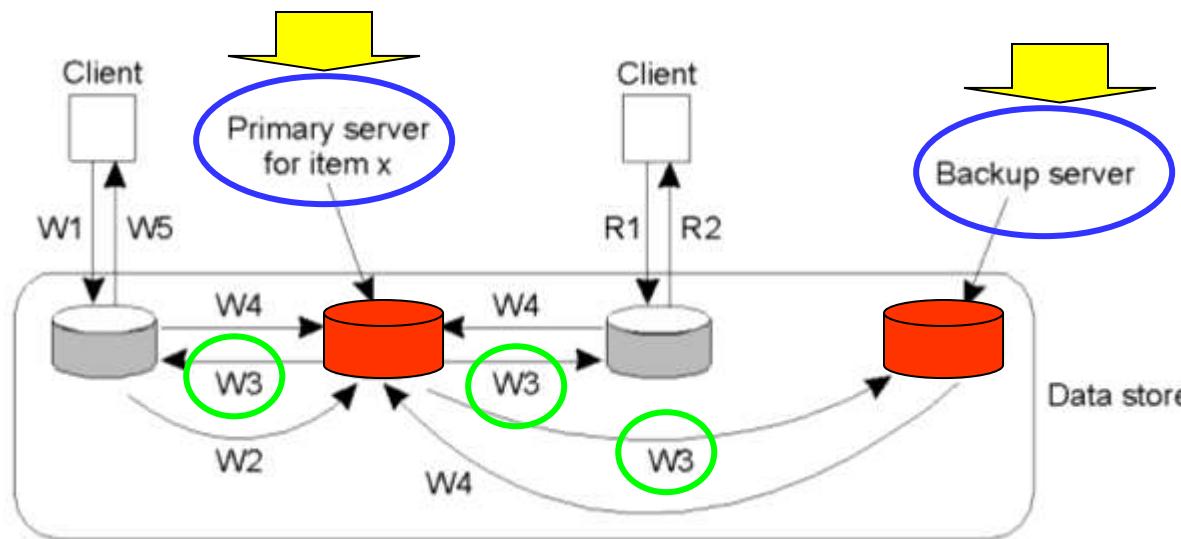
- U suštini podaci nisu uopšte replicirani, već se nalaze na jednom serveru odakle se ne mogu pomerati.
- Svi zahtevi se upućuju primarnoj kopiji, a ostale kopije služe kao backup



Protokol baziran na postojanju primernog skladišta podataka kod koga se sva čitanja (read) i upisi (write) prosleđuju primaru.

# Remote-write protokoli - modifikacija

- \* Sa stanovišta konzistencije su mnogo interesantniji protokoli koji dozvoljavaju procesima da obave read na lokalno raspoloživoj kopiji, a operacije write na fiksnoj, primarnoj, kopiji.
  - Poznati su pod nazivom primary - backup protokoli



W1. Write request

W2. Forward request to primary

W3. Tell backups to update

W4. Acknowledge update

W5. Acknowledge write completed

R1. Read request

R2. Response to read

Proces koji čeka da obavi write operaciju nad X, prosleđuje operaciju odgovarajućem serveru (primaru). Primar obavlja ažuriranje na svojoj lokalnoj kopiji, a zatim prosleđuje ažuriranje backup serverima. Svaki backup obavlja ažuriranje i šalje potvrdu primaru. Kada od svih backup servera primi potvrde, primar šalje potvrdu procesu koji je inicirao write operaciju. Može se desiti da proces koji je inicirao write operaciju dugo čeka na dozvolu da nastavi sa radom (ažuriranje je blokirajuća operacija). Druga mogućnost je da se proces blokira samo dok primar ne obavi ažuriranje. Backup serveri se nakon toga ažuriraju.

# Remote-write protokoli (nast.)

\* Protokoli bazirani na postojanju primarne kopije obezbeđuju laku implementaciju sekvencijalne konzistencije.

- Primar može urediti sve dolazne write zahteve na globalno jedinstveni način – svi procesi vide sve write operacije u istom redosledu, bez obzira koji backup server koriste da pročitaju podatke (tj. implementira sekvencijalnu konzistenciju)
- tradicionalno se primenjuje kod distribuiranih baza podataka i fajl sistema koji zahtevaju visoki nivo otpornosti na greške. Replike su najčešće locirane u istom LANu

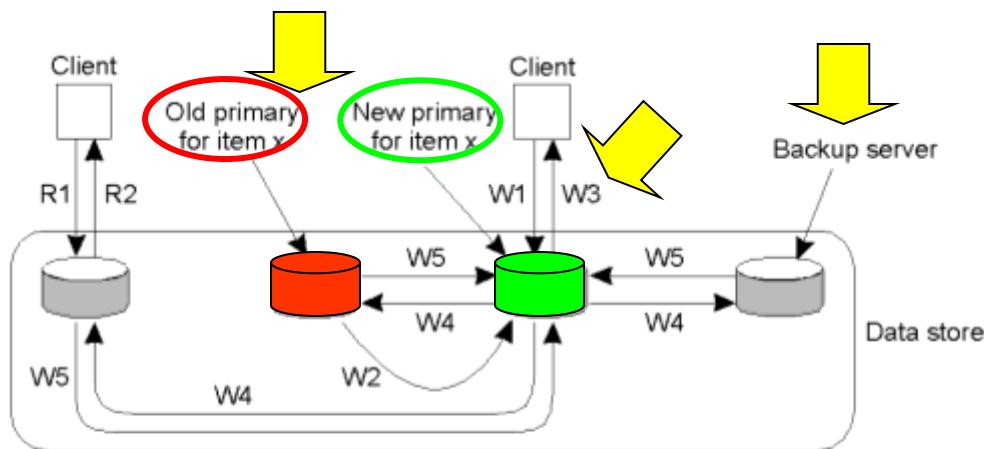
# Local-write protokoli

\* Primarna kopija migrira između procesa koji žele da obave write operaciju.

- Kada proces želi da ažurira podatak X, on prvo locira primarnu kopiju X, a zatim je privlači u sopstvenu lokaciju.

➤ Osnovna prednost ovakvog prilaza je što više uzastopnih write operacija može biti obavljeno lokalno, dok procesi koji obavljaju samo čitanje mogu pristupati svojim lokalnim kopijama.

- Ovo se može postići samo ako se koristi neblokirajući protokol koji dozvoljava da se ažuriraranja proslede replikama tek kada je primar obavio ažuriranje.



W1. Write request  
W2. Move item x to new primary  
W3. Acknowledge write completed  
W4. Tell backups to update  
W5. Acknowledge update

R1. Read request  
R2. Response to read

Primary-backup protokol u kome se primarna kopija seli izmedju procesa koji žele da izvrše operaciju upisa

# Local-write protokoli (nast.)

\* Ovaj prilaz se koristi u distribuiranim fajl sistemima.

- U ovom slučaju može postojati centralni server preko koga se sve write operacije obavljaju (kao kod remote-write primarnog backup protokola).
- Server privremeno dozvoljava jednoj od replika da obavi seriju lokalnih upisa.
- Kada replika server obavi posao, ažuriranje se prosleđuje centralnom serveru odakle se prosleđuje svim replikama.

# Replicirani write protokoli

- \* Kod ovih protokola write operacija se može obaviti na više replika, umesto na jednoj kao kod protokola zasnovanim na primarnim kopijama.
  - Aktivna replikacija – operacija se prosleđuje svim replikama
  - Protokoli zasnovani na kvorumu i većinskom glasanju
- \* Aktivna replikacija
  - Sve replike su ravnopravne
  - Svaka replika ima proces koji obavlja ažuriranja.
  - Ažuriranja se prosleđuju uz pomoć write operacije (operacija, a ne podaci, se prosleđuje replikama)
  - Problem je što operacije svuda moraju biti obavljene u istom redosledu.
    - Neophodno je implementirati mehanizam potpuno uređene grupne komunikacije.
      - Mogu se iskoristiti Lamportove vremenske markice
      - Alternativno, totalno uređenje se može postići korišćenjem centralnog koordinatora (sekvencer).
        - » Svaka operacija se prvo prosleđuje sekvenceru, koji joj dodeljuje jedinstveni redni broj, a zatim se prosleđuje svim replikama.
        - » Operacije se zatim izvršavaju po redosledu definisanom rednim brojevima (ova implementacija podseća na protokole bazirane na primarnim kopijama)
    - Javlja se problem skaliranja.

# Replicirani write protokoli (nast.)

## \* Protokoli bazirani na kvorumu

- Osnovna ideja je da klijenti zahtevaju i dobijaju dozvolu od više servera pre nego što obave čitanje/upis repliciranih podataka.
- Razmotrimo fajl sistem u kome je fajl repliciran na N servera.
  - Da bi se fajl ažurirao klijent mora kontaktirati najmanje  $N/2+1$  (većinu) servera i dobiti dozvolu od njih.
  - Kada se postigne konsenzus, fajl se menja i dodeljuje mu se novi broj verzije koji je isti za sve novonastale fajlove.
  - Da bi pročitao replicirani fajl klijent takođe mora kontaktirati najmanje  $N/2+1$  servera i tražiti da mu pošalju broj verzije zajedno sa fajлом.
    - Bar jedan server će sadržati najnoviju verziju fajla.
      - » Npr. ako ima 5 servera od kojih 3 imaju verziju 8, nemoguće je da preostali serveri imaju verziju 9.

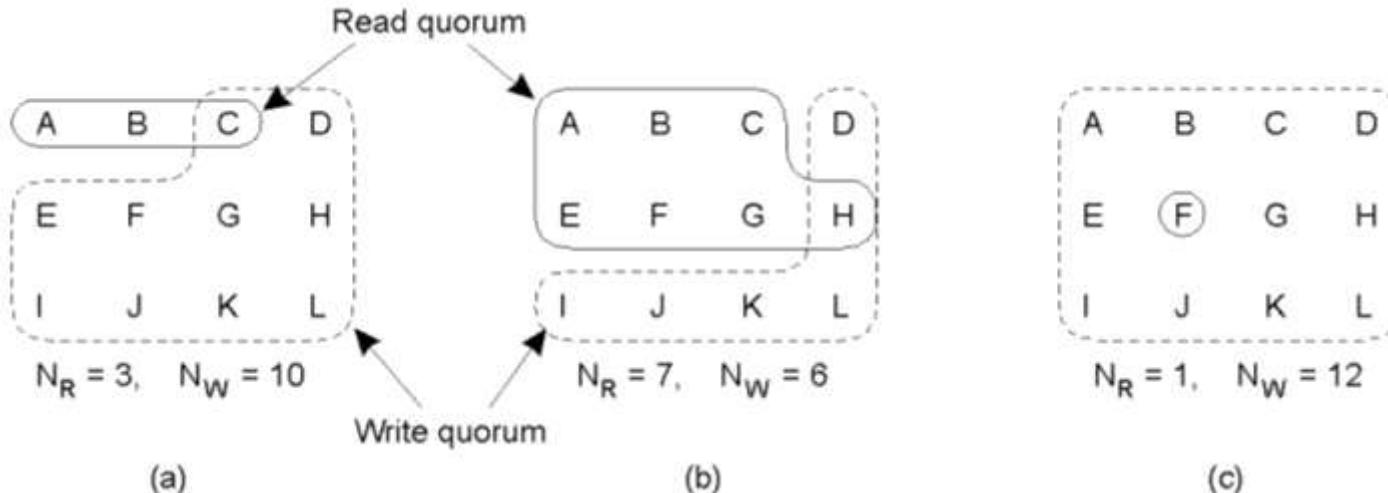
# Protokoli bazirani na kvorumu (nast.)

## \* Opštija šema (Giffordova) kaže

- ako ima  $N$  replika, klijent mora da postigne read kvorum na skupu od najmanje  $N_R$  ili više servera.
- Da bi modifikovao fajl, write kvorum od najmanje  $N_W$  servera se zahteva.
- Moraju da važe sledeći uslovi za  $N_R$  i  $N_W$ :
  1.  $N_R + N_w > N$
  2.  $N_w > N / 2$
- Prvi uslov sprečava read-write konflikte, a drugi write-write konflikte (tj. da postoji preklapanje između dva write skupa). Samo ako se odgovarajući broj servera složi, može se obaviti read ili write.

# Protokoli bazirani na kvorumu (nast.)

## PRIMER



Tri primera algoritma glasanja (fajl je repliciran na N servera): a) korektan izbor read i write skupa, b) izbor koji može dovesti do write-write konflikata i c) korektan izbor, poznat kao ROWA (read one, write all)

Posmatramo sliku a) gde je  $N_R=3$  i  $N_W=10$ . Zamislimo da je najnoviji (poslednji) write kvorum sastavljen od 10 servera od C do L. Svaki od njih uzima novu verziju i broj nove verzije. Svaki sledeći read kvorum od tri servera će morati da sadrži najmanje jedan član iz ovog skupa. Kada klijent vidi brojeve verzije, znaće koji je najskoriji i uzeće taj.

Na slici b) može doći do konflikta zato što je  $N_W \leq N/2$ . Osobito, ako jedan klijent izabere {A, B, C, E, F, G} kao svoj write skup a drugi klijent izabere {D, H, I, J, K, L} kao svoj write skup, onda dolazi do problema zato što će oba ažuriranja biti prihvaćena bez detekcije da su stvarno u konfliktu.

Situacija na slici c) je interesantna zato što postavlja  $N_R$  na 1, čineći mogućim čitanje- read repliciranog fajla nalazeći bilo koju kopiju. Međutim, u ovoj situaciji neophodno je izvršiti ažuriranje svih kopija.

# Distribuirani sistemi



Otpornost na greške (fault tolerance)

# Otpornost na greške

- \* Tolerantnost sistema na defekte (fault tolerance) predstavlja osobinu sistema da korektno funkcioniše uprkos pojavi grešaka.
- \* Hardwer, softwer i mreže ne mogu biti 100% pouzdani
- \* Računarski sistemi imaju široku primenu u svakodnevnom životu.
  - Veliki broj takvih sistema koristi se u hazardnim uslovima i za daljinsko upravljanje ( u nuklearnim reaktorima, avionima i svemirskim brodovima)
  - U takvom okruženju računarski sistemi su, pre svega zbog uticaja zračenja, u velikoj meri podložni greškama.
  - što je sistem složeniji, to su šanse za nastupanje defekata veće
    - Zbog toga, pored performansi, otpornost na defekte postaje vrlo važno pitanje.
- \* distribuirani sistemi mogu biti otporniji na defekte od centralizovanih sistema (kod kojih je otkaz najčešće fatalan), ali sa više hostova, verovatnoća individualnih defekata se povećava

# Defekti (Faults), greške (Errors) i otkazi (Failures)



- \* Defekt je trajni ili privremeni nedostatak koji se javlja u nekoj hardverskoj ili softverskoj komponenti
- \* Greške su manifestacija defekta i predstavljaju odstupanje vrednosti podatka od očekivane (defekt je uzrok greške)
- \* Otkaz nastupa kada sistem više ne može da obavlja funkciju za koju je projektovan
  
- \* Za sistem se kaže da je otporan (tolerantan) na greške (ili defekte) ako je u stanju da korektno funkcioniše čak i u prisustvu grešaka

# kategorizacija grešaka

## \* U odnosu na trajanje

- Prolazne (transient faults)

- Pojave se jednom i nestanu

- Npr. Istekao time out za prenos poruke, ali nakon ponovnog slanja poruka je uspešno otplaćena

- Peridične (intermittent)

- Greška se pojavi, zatim nestane, pa ponovo pojavi. Najnezgodniji tip grešaka

- Npr greška u mrežnom kablu kod koga postoji neki prekid

- Stalne (permanent)

- Greške koje postoje sve dok se komponenta ne zameni ispravnom

- Greška u hard disku, pregorela komponenta, bagoviti sw.

## \* Druga klasifikacija grešaka

- Tvr. "mirna" (fail-silent, ili fail-stop) greška – komponenta prestaje sa radom i ne generiše nikakv izlaz ili generiše poruku o grešci
- Vizantijske greške – komponenta nasatavlja sa radom i generiše pogrešan rezultat.

# Kako sistem učiniti otpornim na greške?

\* Opšti prilaz u projektovanju sistema otpornih na greške je redundansa.

- Može se primeniti na nekoliko nivoa

- Informaciona

- Otpornost na greške se postiže repliciranjem ili kodiranjem podataka (npr. Hammingovi kodovi, ABFT)

- Vremenska

- Tolerantnost na greške se postiže izvršenjem operacija više puta u vremenu. Npr. timeouti i retransmisija poruka.
    - Ova vrsta redundantnosti je korisna u slučaju prolaznih ili periodičnih grešaka.

- Fizička redundansa

- hardverska

- » Bavi se uređajima a ne podacima. Hardver se umnožava da bi sistem mogao tolerisati otkaze. Npr. RAID diskovi (redundant array of independent disks) i backup servri.

- softverska

- » replicirani procesi

# Koliko otpornosti na greške?

- \* 100% tolerantnost na greške se nikada ne može postići.
  - Što smo bliži cifri od 100%, to je sistem skuplji i složeniji.
- \* Za sistem se kaže da je k-tolerantan na greške ako može podneti k grešaka a da pri tome nastavi da korektno funkcioniše.
  - Ako je reč o "mirnim" greškama potrebno je  $k+1$  komponenta da bi sistem i dalje funkcionsao
    - Ako k otkaže, postoji još uvek jedna koja korektno radi
  - Ako je reč o Vizantijskim greškama potrebno je  $2k+1$  komponenti
    - k može generisati pogrešne rezultate, ali će  $k+1$  korektno funkcionsati i većinskim glasanjem izglasati korektan rezultat.

# Primer1: fizička hardverska redundansa

\* Tehnika za postizanje visoke pouzdanosti kroz fizičku redundantnost.

- Najjednostavnija varijanta- TMR (Tripple Modular Redundancy)
- Koriste se tri primerka komponente da bi se detektovala i korigovala jednostruka greška.
- Primer sistem bez redundantne



Otkaz bilo koje komponente, dovodi do otkaza celog sistema

Ako komponenta **i** ima pouzdanost **R<sub>i</sub>**, pouzdanost celog sistema je

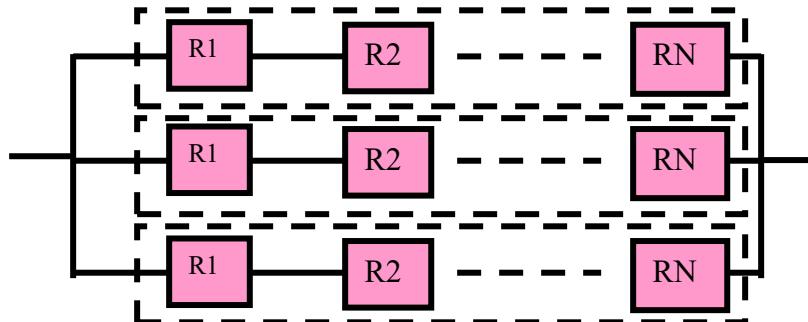
$$R = R_1 R_2 R_3 \dots R_N = \prod_{i=1}^N R_i$$

Npr. ako sistem ima 100 komponenti, otkaz bilo koje komponente će izazvati otkaz celog sistema. Ako svak komponenta ima pouzdanost 0.999, pouzdanost celog sistema je

$$R = R_1 R_2 R_3 \dots R_{100} = (0.999)^{100} = 0.905$$

# fizička hardverska redundansa (nast.)

- \* ako hardver repliciramo 3 puta, pouzdanost sistema će se povećati



- \* sada postoji tri paralelna puta.
- \* ako je pouzdanost na jednom putu  $R_i$ , a nepouzdanost  $Q_i = 1 - R_i$ , tada je nepouzdanost celog sistema

$$Q = Q_1 Q_2 Q_3 \dots Q_N$$

tj.

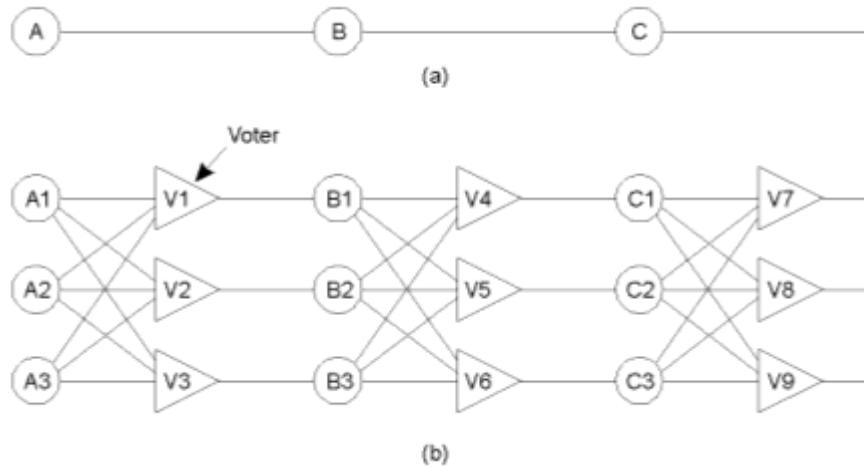
$$1 - R = [1 - R_1][1 - R_2][1 - R_3] \dots [1 - R_N]$$

Za prethodni primer gde je pouzdanost sistema iznosila 0.905, nakon tripliciranja, pouzdanost iznosi

$$R = 1 - (1 - 0.905)^3 = 0.9991$$

# fizička hardverska redundansa (nast.)

\* TMR tehnika se primenjuje u sprezi sa glasačima (voterima)



- a) Sistem bez redundantnosti: signal prolazi kroz uređaje A, B i C sekvenčijalno. Ako je jedan od uređaja neispravan, konačni rezultat će verovatno biti nekorektan.
- b) TMR: svaka komponenta je replicirana 3 puta. Izlaz iz svake komponente se vodi na glasač (voter). I voteri su replicirani. Ako su tri ili dva ulaza ista, izlaz je jednak ulazu. Ako su tri ulaza različita, izlaz je nedefinisan.

# Primer2: informaciona redundansa – ABFT tehnika (Algorithm base fault tolerance)

\* tehnika koja se može primeniti kod odedjene klase algoritama

- tipično za matrična izračunavanja (množenje matrica, LU dekompozicija, inverzija..)

\* Primer: množenje matrica  $C = A * B$ , A, B, C su dimenzija  $n \times n$

- Matrica A se proširuje vrstom čiji elementi predstavljaju sumu elemenata odgovarajuće kolone (matrica  $A_C$  dimenzija  $(n+1) \times n$ )
- Matrica B se proširuje kolonom čiji elementi predstavljaju sumu elemenata odgovarajuće vrste (matrica  $B_R$  dimenzija  $n \times (n+1)$ ).
- Rezultujuća matrica  $C_f$  ima je dimenzija  $(n+1) \times (n+1)$

$$C = A * B, \Rightarrow C_f = A_c * B_r$$

$$A_c = \begin{bmatrix} A \\ \vec{e}^T A \end{bmatrix}, \quad B_r = \begin{bmatrix} B & B\vec{e} \end{bmatrix}, \quad C_f = \begin{bmatrix} AB & AB\vec{e} \\ \vec{e}^T AB & \vec{e}^T AB\vec{e} \end{bmatrix}$$

$$\vec{e}^T = [1 \ 1 \ 1 \dots 1]$$

# Primer2 (nast.)

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad A_c = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ \hline 2 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad B_r = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix}$$

$$C_f = A_c * B_r = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 4 \\ 5 & 2 & 4 \\ 4 & 4 & 8 \end{bmatrix}$$

Npr. Element  $c_{21}$  je pogrešan i ima vrednost 5 umesto 2. izračunavanjem sume elementa prve kolone (druge vrste) i poređenjem sa kontrolnom sumom odgovarajuće vrste i kolone detektovaće se i locirati greška.  
Korekcija greške:  
 $C_{12} = 5 + (4 - 7) = 5 - 3 = 2$

**Detekcija greške:** naći sume elemenata svake vrste / svake kolone (S1) i uprediti ih sa dobijenim kontrolnim sumama (S2); ako postojje razlike detektovana je greška

**Lokacija greške:** u preseku vrste i kolone gde je otkriveno neslaganje kontrolnih suma i suma elemenata

**Korekcija greške:**  $c := c' + (S2 - S1)$

# Erasure coding

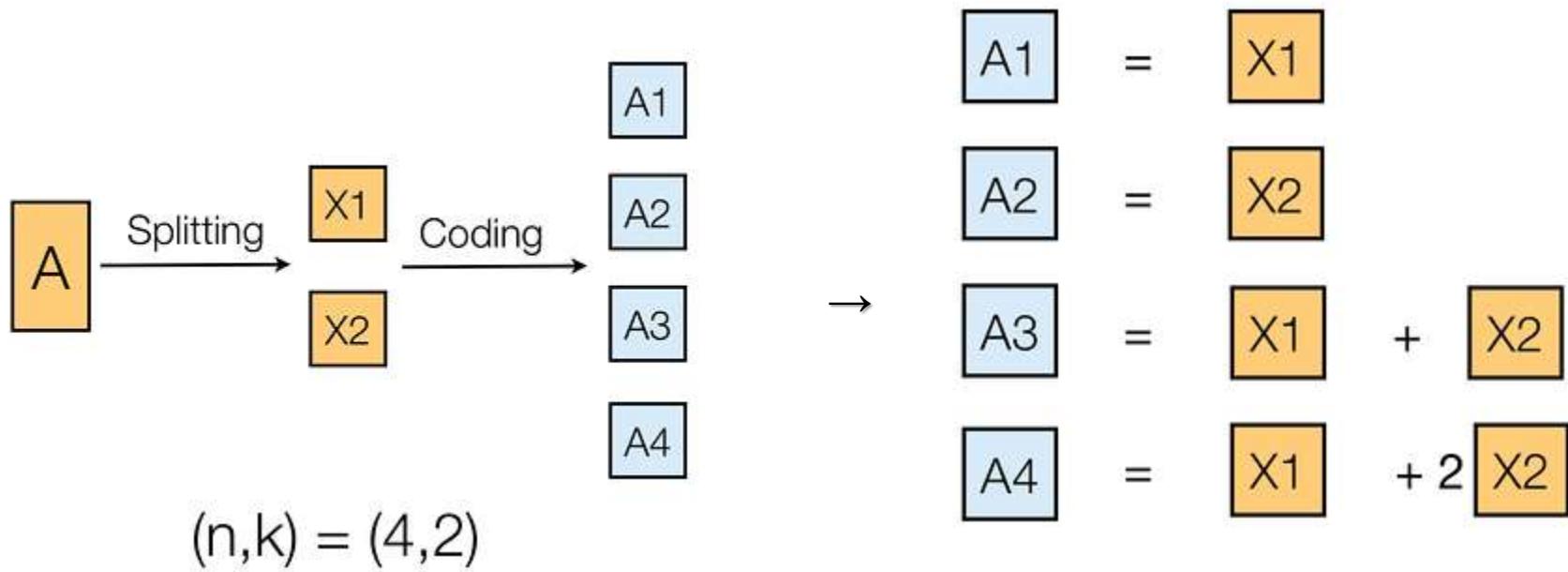
- \* Tehnika za postizanje otpornosti na greške (fault tolerance)
- \* Tehnika kodiranja kod koje se skup od k podataka kodira skupom od n,  $n > k$  podataka tako da se na osnovu bilo kog skupa od k podataka mogu regenerisati originalni podaci.
  - (n,k) erasure code omogućava tolerisanje  $n-k$  grešaka.
  - Tehnika koja je široko prihvaćena kod HDF (Hadoop file system)
    - Hadoop je sistem za skladištenje, pretraživanje i procesiranje velike količine podataka (big data)

# Erasure coding - primer upotrebe u DFS

- \* Posmatrajmo fajl veličine  $M$
- \* Podeliti fajl na  $k$  blokova veličine  $M/k$
- \* Primeniti  $(n, k)$  code na ovih  $k$  blokova da bi se dobilo  $n$  blokova, svaki veličine  $M/k$ .
  - Na taj način fajl je proširen  $n/k$  puta
  - Potrebno je da  $n$  bude veće ili jednako  $k$ .
    - Ako je  $n$  jednako  $k$ , fajl je samo podeljen na blokove, ali nema nikakvog kodiranja
- \* Bilo koji podskup  $k$  od  $n$  blokova se može uzeti da bi se rekonstruisao fajl
  - to znači da kod može tolerisati  $(n-k)$  grešaka

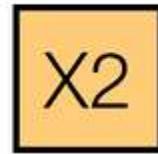
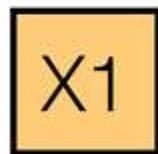
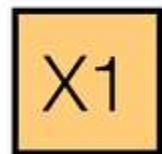
# Primer(4,2) koda

- \* Podelimo fajl na 2 bloka i formrajmo 4 nova bloka
- \* Neka se 4 bloka pamte u 4 različita čvora



Zašto je ovo bolje nego da smo jednostavno svaki blok zapamtili u 2 različita čvora?

# Primer(4,2) koda



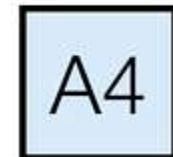
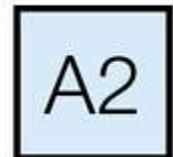
Node 1

Node 2

Node 3

Node 4

- Šta ako čvorovi 1 i 3 otkažu?
  - Fajl se ne može regenerisati!



Node 1

Node 2

Node 3

Node 4

Moguće je  
regenerisati fajl

$$A1=X1$$

$$A2=X2$$

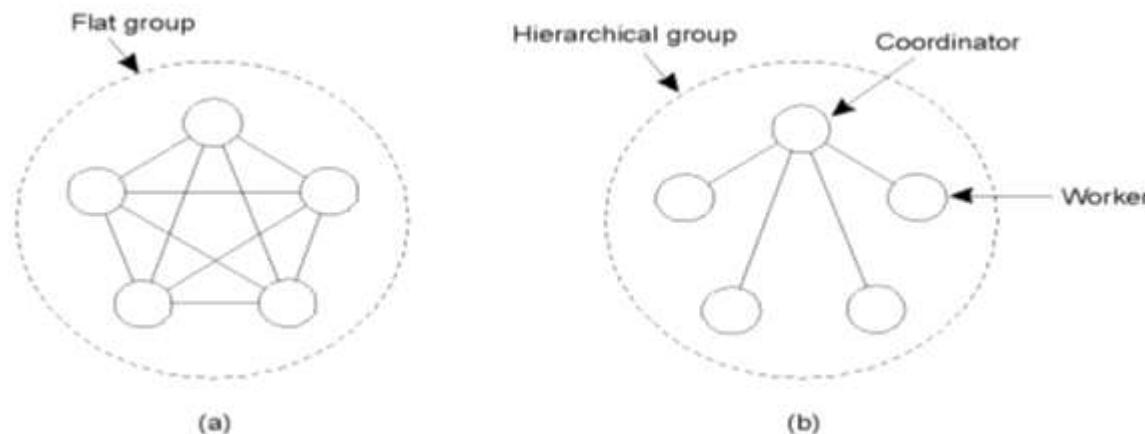
$$A3=X1+X2$$

$$A4=X1+2*X2$$

# Kako se tolerantnost na greške postiže u DS?

## \* Kroz replikaciju

- nekoliko identičnih kopija procesa se organizuje u grupu.
- Kada se neka poruka pošalje grupi, svi članovi grupe je primaju
  - ako jedan proces u grupi otkaže (greši) postoji drugi koji obavlja isti posao
- replicirani procesi mogu biti organizovani u grupe ravноправnih procesa (flat) ili hijerarhijske grupe
  - ravne grupe – svi procesi su jednaki
  - hijerarhijske – postoji jedan koordinator



# Kako se tolerantnost na greške postiže u DS? (nast.)

## \* Replikacija može biti bazirana na postojanju primarne i backup kopija (pasivna replikacija)

- koristi se kod hijerarhijski organizovanih grupa

- Primarni server obavlja ceo posao
- Ako primar otkaže, jedan od backup servera preuzima posao.
  - Otkaz primara treba da je nevidljiv za aplikativne programe

- Kako backup zna da je primar otkazao

- Periodično mu šalje poruku "da li si živ?"
- Ako istekne timeout i na pitanje se ne dobije odgovor, backup server preuzima posao.

## \* aktivna replikacija

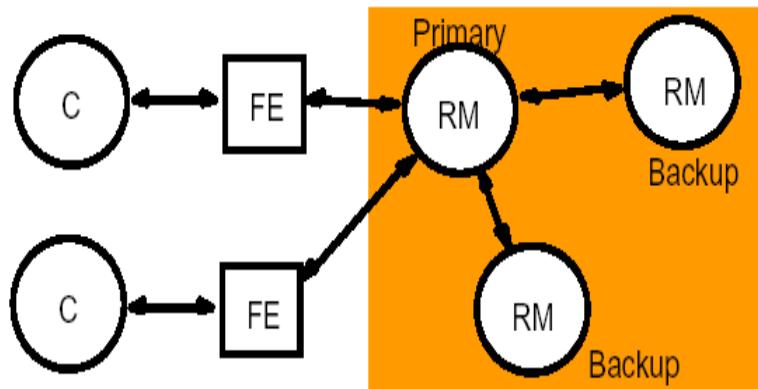
- koristi se kod grupa ravnopravnih procesa

- klijentski zahtev se prosledjuje svim procesima u grupi
- Zahtevi moraju da stignu u sve servere u istom redosledu, tj. neophodna je sinhronizacija
- Ako se zahtevi procesiraju po redosledu, svi serveri koji korektno funkcionišu će dati isti odgovor.

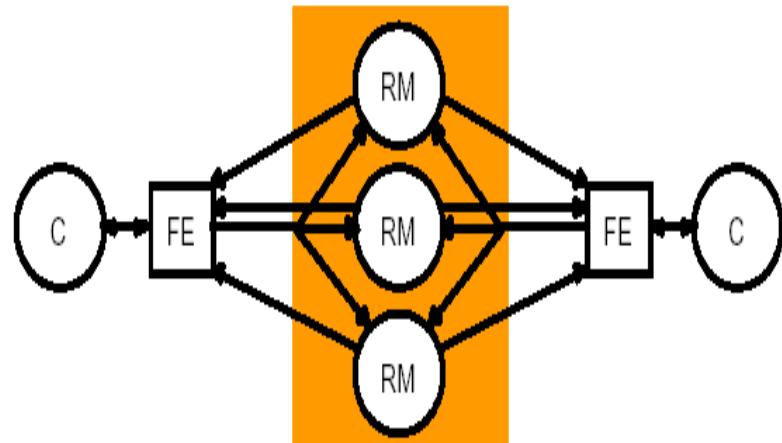
# Pasivna i aktivna replikacija

## \* Pasivna replikacija

- Klijent (C), Front End (FE) = klijent interface, Replica Manager (RM) = davalac usluge (server)



## \* Aktivna replikacija



## \* Prednosti replikacije bazirane na postojanju primarne kopije u odnosu na aktivnu replikaciju

- Jednostavniji dizajn:

- dovoljan je samo jedan backup server
- Nema potrebe za grupnom komunikacijom (multicast) i nema problema sa sinhronizacijom (poruka se šalje samo jednom serveru)

## \* Nedostaci

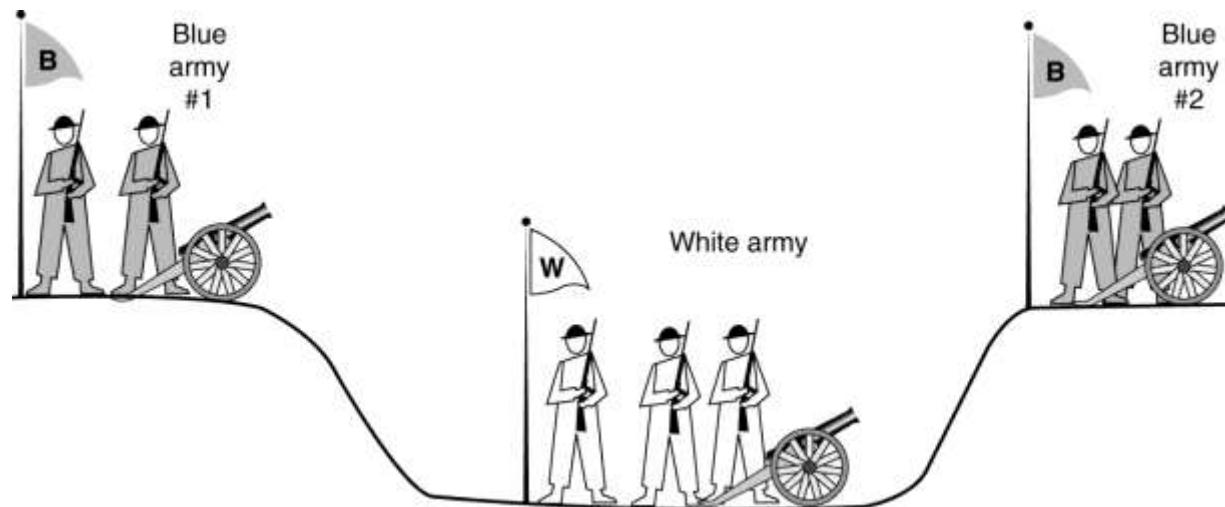
- Loše funkcioniše u prisustvu Vizantijskih grešaka – primar može pogrešno da radi a da greška ne bude otkrivena
  - jedini način za postizanje tolerantnosti na greške vizantijskog tipa je kroz aktivnu replikaciju

# Usaglašavanje u prisustvu grešaka

- \* Distribuirani procesi često moraju da se usaglase oko nečega.
  - Npr. Izbor koordinatora, okončanje transakcije, pristup kritičnoj sekциji, ili šta je korektan rezultat, itd.
- \* Kako postići konsenzus ako neki procesi u grupi nekorektno funkcionišu ili je komunikacioni kanal nepouzdan?
- \* Usaglašavanje je potrebno postići u konačnom broju koraka.
- \* Posmatraju se dva odvojena scenarija:
  - Ispravni procesori, a nesavršeni komunikacioni kanali ("Problem dve armije")
  - Nesavršeni procesori, a pouzdani komunikacioni kanal ("Problem vizantijskih generala")

# Problem dve armije (two army problem)

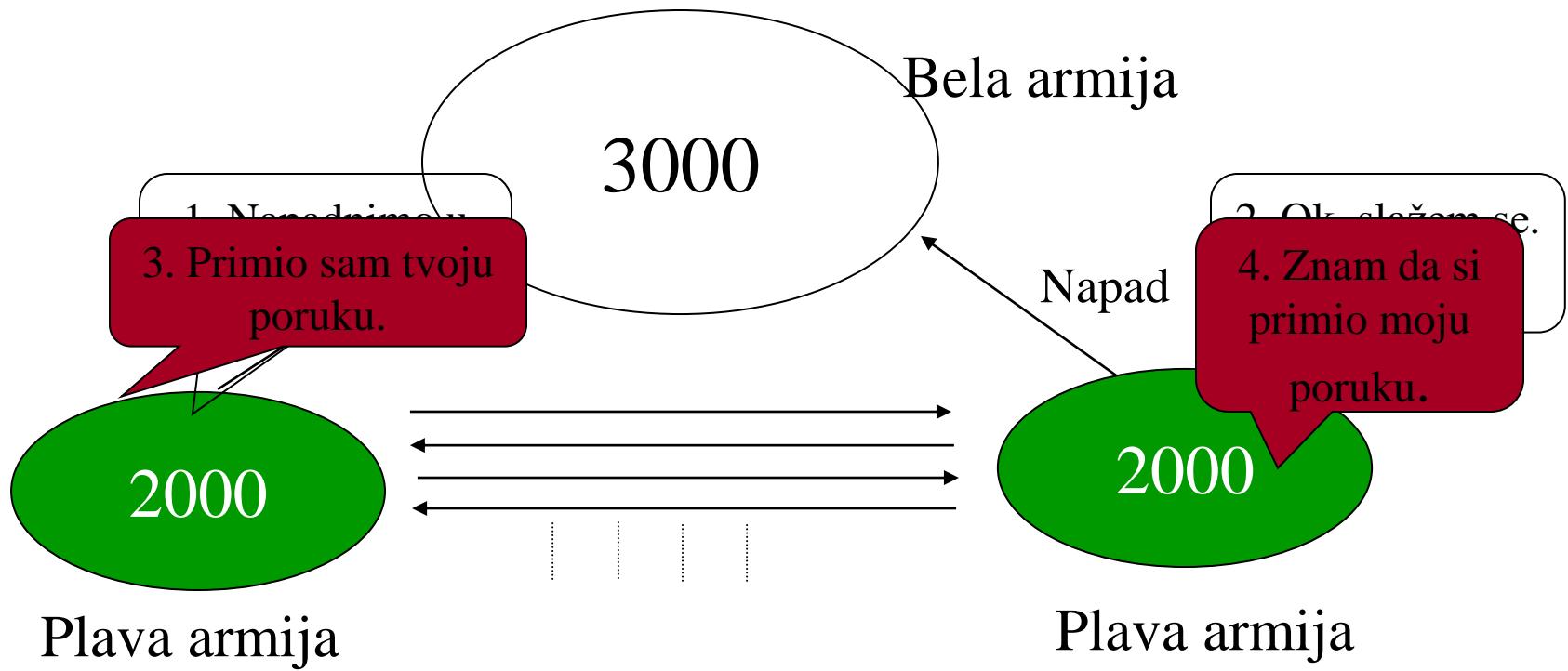
- \* Dve divizje, A i B, jedne armije, treba da koordinišu napad na drugu armiju, C.
- \* A i B imaju po 2000 vojnika, a armija C 3000.
  - Ako se A i B dogovore da zajedno napadnu, mogu da pobede C



# Problem dve armije (nast.)

- \* A i B su fizički odvojene i koriste kurira za razmenu poruka.
  - A šalje poruku B u kojoj piše "izvršimo napad u zoru".
  - B prima poruku i slaže se sa predlogom (šalje ok – potvrdu)
  - Kurir stiže u A sa povratnom porukom (potvrdom), ali tada A zaključuje da B ne zna da li je A primio potvrdu i neće biti siguran da li da obave napad.
  - A može da odluči da pošalje poruku B-u "primio sam potvrdu", ali tada A neće biti siguran da li je ta poruka stigla u B, itd.
- \* Problem je poznat pod nazivom problem višestrukih potvrda (*multiple acknowledgment* problem).
- \* On pokazuje da čak i u prisustvu ispravnih procesora, usaglašavanje između dva procesa nije moguće ako su komunikacije nepouzdane.
  - Jedino se možemo nadati da najčešće jesu!
- \* Ne postoji protokol koji može da reši problem nepouzanih kanala!

# Problem dve armije



→ Može se pokazati da generali plave armije nikada ne mogu postići konsenzus (zbog nepouzdane komunikacije)

# Problem vizantijskih generala

- \* Pouzdane komunikacije, nepouzdani procesori.
- \* Postoji  $n$  generala svaki sa svojom divizijom i treba da donesu odluku da li da napadnu neprijatelja ili ne.
  - Komunikacije su pouzdane (radio ili telefon), ali  $m$  generala su izdajnici i pokušavaju ostale generale da spreče da postignu konsenzus šaljući im netačne i kontradiktorne informacije.
- \* Pitanje je da li lojalni generali ipak mogu postići konsenzus (doneti istu odluku) ?
  - Svaki general daje svoj predlog za akciju: napad (N) ili povlačenje (P) šaljući poruku svim ostalim generalima, i prima poruke od ostalih generala sa njihovim predlozima
  - generali izdajnici mogu slati različite poruke ostalim generalima
  - Svaki general donosi odluku o tome koju će akciju preuzeti većinskim izglasavanjem primljenih predloga uzimajući u obzir i svoj predlog
    - Kako da svi lojalni generali izglasaju isti predlog (napad ili povlačenje)?

# Lamportov algoritam za problem vizantijskih generala

- \* Lamport je predložio rešenje koje radi pod određenim uslovima.
- \* Zaključak je da bilo koje rešenje za prevazilaženje problema  $m$  generala izdajnika zahteva najmanje  $3m+1$  učesnika ( $2m+1$  lojalnih generala), tj. Više od  $2/3$  generala mora biti lojalno.
- \* Pokazano je da je potrebno obaviti  $m+1$  rundu razmene poruka i razmeniti  $O(mn^2)$  poruka
- \* Ovo rešenje je veoma skupo.
- \* Ima ga smisla primeniti u nekom hardveru specijalne namene, ali je neefikasno ga koristiti u distribiranom računarskom okruženju.

# BGP - problem

- \* Problem n vizantijskih generala se može posmatrati kao n identičnih problema
  - postoji jedan general i n-1 poručnik
  - m poručnika (ili m-1 poručnik i general) mogu biti izdajnici
  - lojalni general šalje svim poručnicima istu poruku (plan akcije: napad ili povlačenje)
  - svaki poručnik primljenu poruku prosleđuje ostalim poručnicima (tj. ponaša se kao general)
  - kada se sve poruke prikupe primenjuje se većinsko glasanje
    - ako je moguće izglasati vrednost, ona se izglosa, inače se koristi podrazumevana akcija (npr. povlačenje)
- \* Želimo da postignemo sledeće:
  1. svi lojalni poručnici moraju da donešu istu odluku
  2. ako je general lojalan, tada svi lojalni poručnici poštuju njegovo naređenje.

# Byzantine Generals Problem (BGP)

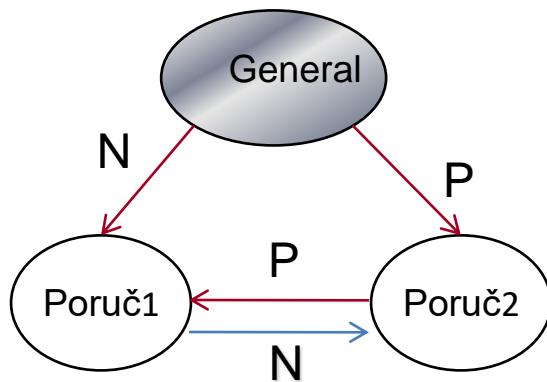
## Mora se postići da

1. Svi lojalni poručnici donesu istu odluku
2. ako je general lojalan, tada svaki lojalni poručnik poštuje generalovo naređenje

Prepostavke:

1. Svaka poruka se prima korektno (komunikacioni kanal ne unosi greške)
2. Prijemnik zna ko je posalo poruku
3. Odsustvo poruke se može detektovati

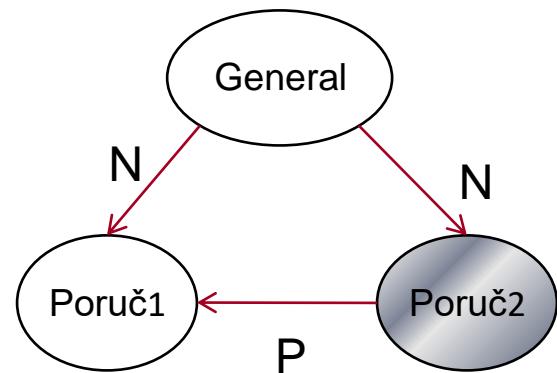
general je izdajnik



lojalni poručnici će doneti istu odluku

N – napad  
P - povlačenje

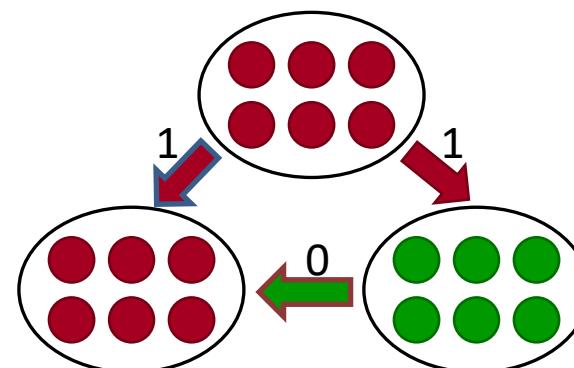
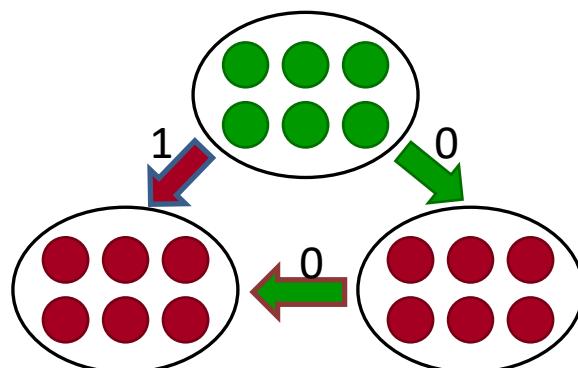
•poručnik 2 je izdajnik



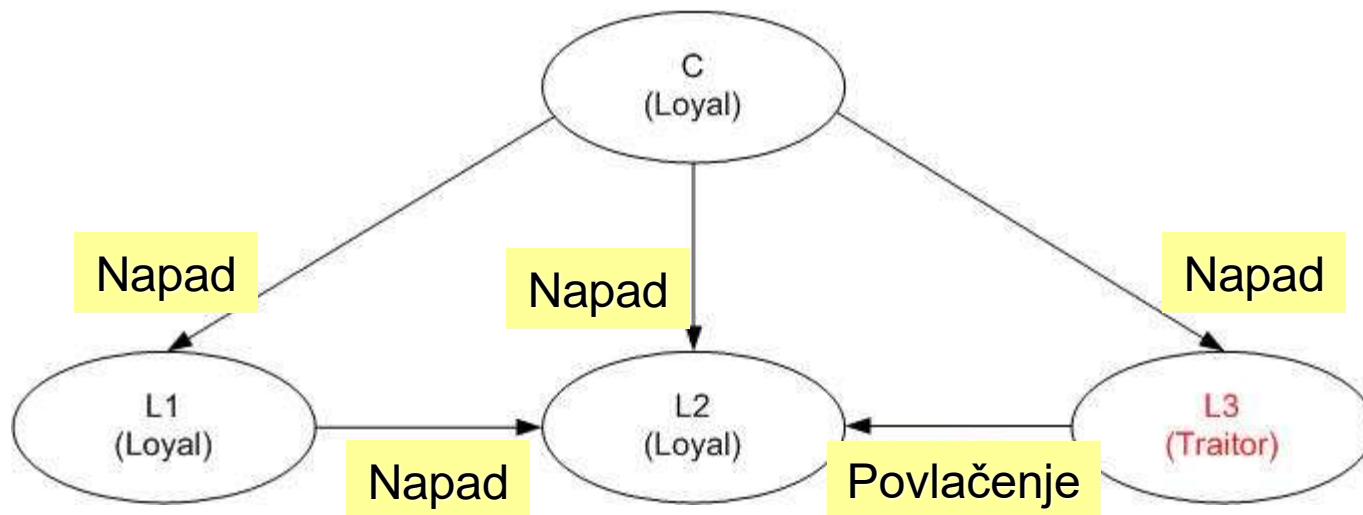
• lojalni poručnik 1 neće poslušati  
naređenje lojalnog generala!

# BGP

- \* ako postoji  $m$  izdajnika i  $n \leq 3m$  generala nemoguće je postići konsenzus
- \* da bi moglo da se toleriše  $m$  izdajnika mora postojati najmanje  $2m+1$  lojalnih generala, tj minimalni broj generala mora biti  $n=3m+1$



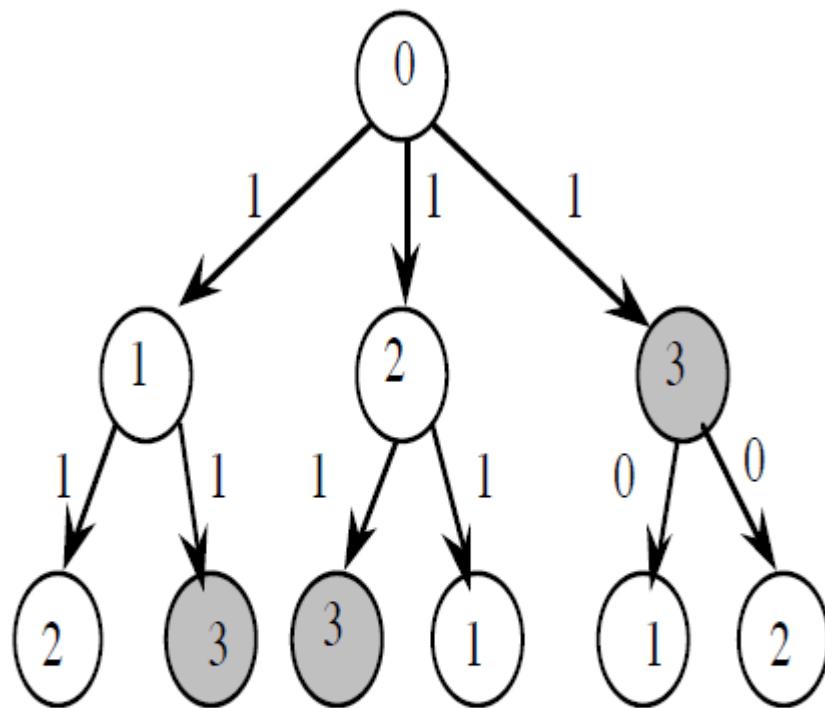
# Primer n=4, m=1



- \*  $n=4$  generala;  $m=1$  izdajnik
- \* L2 računa  $\text{majority}(\text{Napad}, \text{Napad}, \text{Povlačenje}) = \text{Napad}$

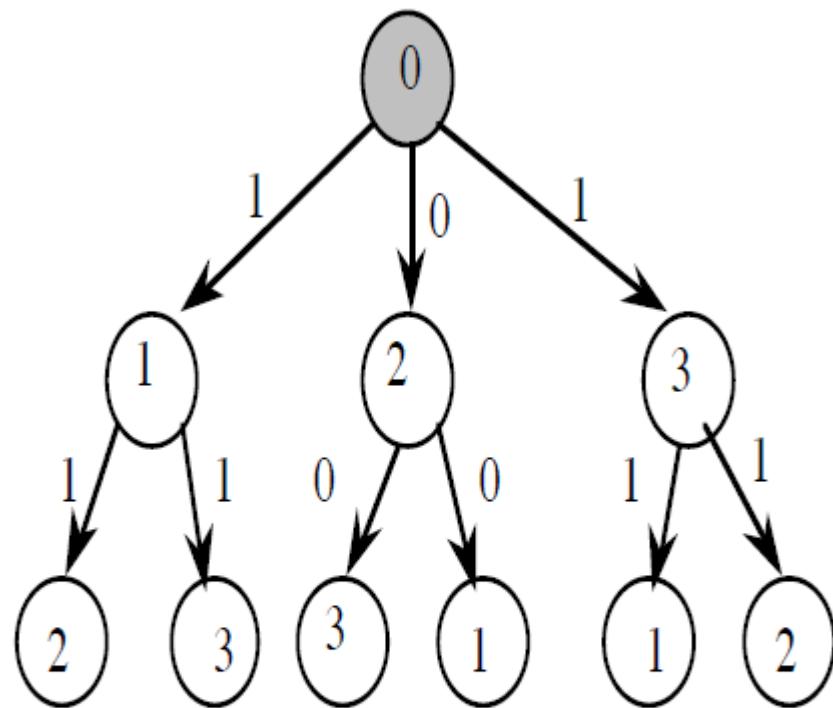
# BGP- ekivalentni prikaz rešenja

commander



(a)

commander



(b)

Example with  $m=1$  and  $n=4$

# Lamportov rekuzivni algoritam OM(m)

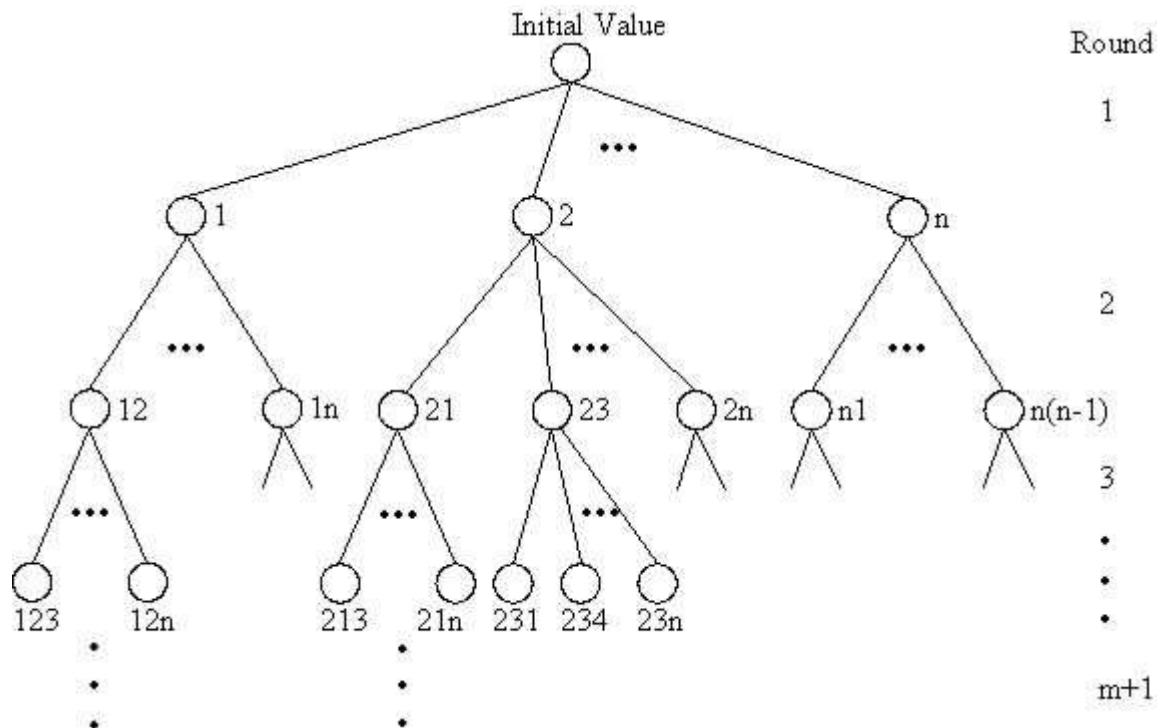
Rekuzivna definicija: slučaj  $m=0$  okončava rekurziju

**Algoritam OM(0)** – (pošto je  $m=0$  nema izdajnika)

1. General šalje svoje naređenje svim poručnicima
2. Svaki poručnik poštuje naređenje koje je dobio od generala

**Algoritam OM( $m$ ),  $m > 0$**

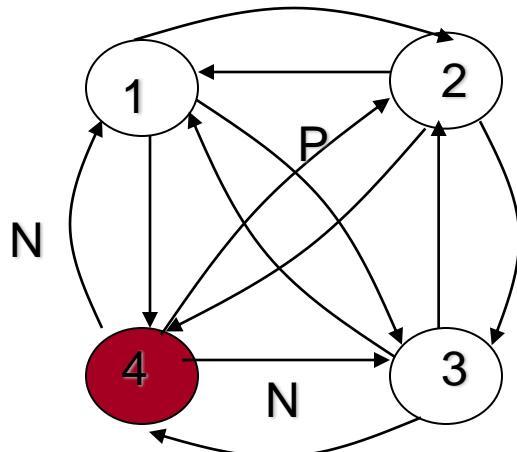
1. General šalje svoje naređenje svakom poručniku
2. Za svako  $i$ , neka  $v_i$  označava vrednost koju je poručnik  $i$  primio od generala. Poručnik  $i$  deluje kao general u Algoritmu OM( $m-1$ ) da bi poslao vrednost  $v_i$  ostalim  $n-2$  poručnicima.
3. Za svako  $i$ , i svako  $j \neq i$ , neka  $v_j$  predstavlja vrednost koju je poručnik  $i$  primio od poručnika  $j$  u koraku 2 (korišćenjem OM( $m-1$ )). Poručnik  $i$  izračunava  $\text{majority}(v_1, \dots, v_{n-1})$ .



# primer1: 4 generala, 1 izdajnik

- \* 3 lojalna generala i m=1 izdajnik
- \* potrbno je obaviti  $m+1=1+1=2$  runde razmene poruka da bi se postigao konsenzus
- \* u prvoj rundi svaki general šalje svim ostalim generalima svoj predlog
- \* u drugoj rundi svaki general šalje ono što je primio od drugih generala
  - lojalni generali prosledjuju tačno poruke koje su primili od drugih generala
  - izdajnici mogu promeniti poruke
  - kada se prikupe sve poruke primenjuje se većinsko glasanje da bi se odredila veličina trupe svakog generala i donela odluka

# primer1: 4 generala, 1 izdajnik (nast.)



I runda

1. predlaže N
  2. predlaže N
  3. predlaže P
  4. šalje predlog N 1. i 3.  
generalu, a 2. predlog P
- \*\*\*\*\*

1. general ima predloge (N,N,P,N)
2. general ima predloge (N,N,P,P)
3. general ima predloge (N,N,P,N)
4. general ima predloge (N,N,P,\*)

**LOJALNI GENERALI NE MOGU POSTIĆI KONSENZUS  
POSLE PRVE RUNDE!!**

II runda:

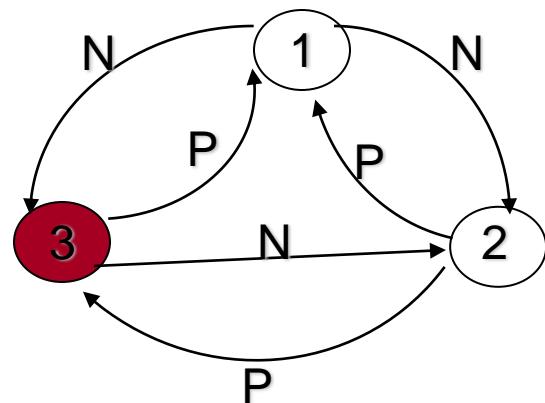
1. prima  
(N,N,P,P)
2. prima  
(N,N,P,N)
3. prima  
(\* , \* , \* , \*)
4. prima  
(N,N,P,\*)

1. prima  
(N,N,P,N)
2. prima  
(N,N,P,N)
3. prima  
(\* , \* , \* , \*)
4. prima  
(N,N,P,\*)

1. prima  
(N,N,P,N)
2. prima  
(N,N,P,P)
3. prima  
(\* , \* , \* , \*)
4. prima  
(N,N,P,N)

**(N,N,P,\*) – svi lojalni generali imaju isti vektor i mogu većinski izglasati istu odluku**

# primer2: 3 generala, 1 izdajnik



## I runda

- general ima predloge (N,P,P)
  - general ima predloge (N,P,N)
  - general ima predloge (N,P,\*)

## II runda

1. prima vektore  $(N, P, N)$   
 $(*, *, *)$

(?, ?, ?)

(?, ?, ?) – lojalni generali ne mogu postići konsenzus!

Broj lojalnih generala (korektnih procesa) mora biti veći od  $\frac{2}{3}$  ukupnog broja da bi se većinskim glasanjem postigao konsenzus.

# Strategije za oporavak od greške

- \* Kada nastupi greška, potrebno je izvršiti oporavak od greške i vratiti sistem u korektno stanje
- \* To se može postići na dva načina:
- \* Povratak u stanje pre nastupanja greške (Backward Recovery):
  - vratiti sistem u neko prethodno korektno stanje korišćenjem tačaka provere (checkpoints), a zatim nastaviti sa radom
    - npr retransmisija u slučaju gubitka paketa ili greške u primljenom paketu
- \* Prelazak u novo stanje (Forward Recovery):
  - prevesti sistem u korektno novo stanje nakon čega se može nastaviti sa radom
    - Erasure coding,  $(n,k)$ -tehnika kodiranja podataka koja omogućava korekciju u slučaju da je nastupila greška (skup od  $k$  poruka se kodira u  $n$  poruka tako da bilo kojih  $k$  poruka na odredištu mogu da se iskoriste da se rekonstruiše  $k$  originalnih poruka).

# Forward i Backward Recovery

## \* Nedostatak oporavka sa povratkom u nazad

- beleženje stanja sistema ubacivanjem tačaka provere (Checkpointing) može biti skupo (pogotovo ako su greške retke)

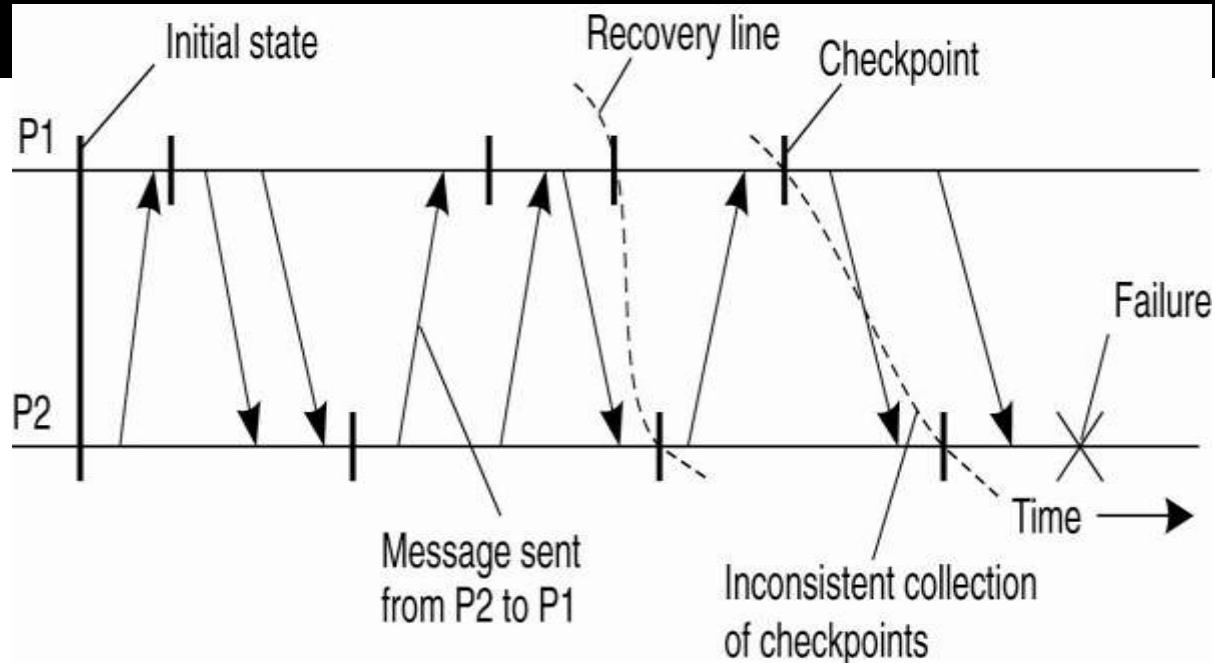
➤ uprkos ceni, ova tehnika se češće koristi od forward.

– “logging” se može smatrati jednom vrstom checkpoint-a

## \* Forward Recovery:

- da bi ova tehnika funkcionisala neophodno je predvideti sve potencijalne greške
- kada nastupi greška, mehanizam za oporavak zna šta treba da uradi da bi doveo sistem u naredno korektno stanje.

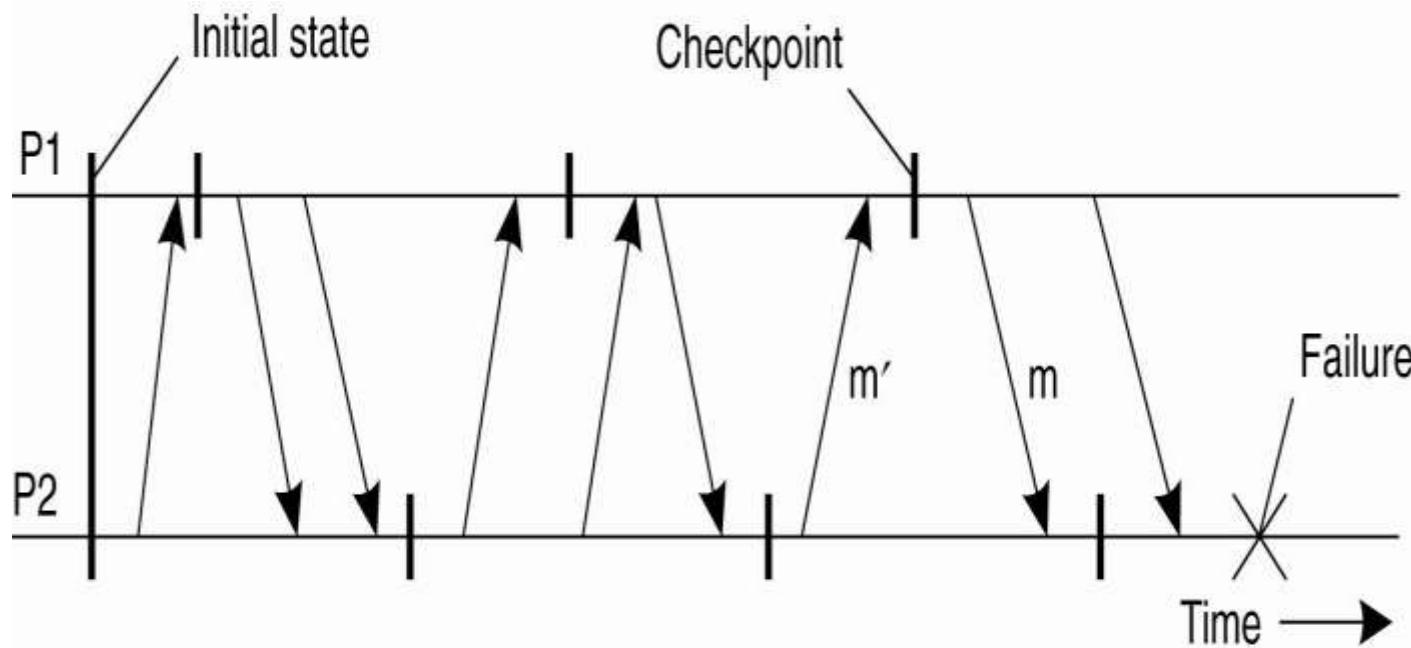
# Checkpointing



- Svaki proces povremeno pamti svoje stanje na disku (npr. vrednosti promenljivih, primljene poruke, poslate poruke) u tačkama provere.
- Ako nastupi greška, proces se vraća u stanje pre nastupanja greške na osnovu podataka zapamćenih u checkpoint.
- U DS potrebno je da se svi procesi vrate u stanje odakle je moguće izvršiti oporavak, tj. potrebno je pronaći liniju oporavka (konzistentni presek). Skup tačaka provere  $C$  je konzistentan ako za sve događaje  $e$  i  $e'$  važi
$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$
- to znči da ako je npr. proces P zabeležio prijem poruke iz procesa Q, tada u skupu mora postojati i tačka u kojoj je proces Q zabeležio slanje poruke procesu P (poruka je morala da stigne od nekog).

# domino efekat kod nezavisnog beleženja stanja (checkpinting)

Međutim, ako svaki proces za sebe donosi odluku kada da kreira checkpoint, problem pronalaženja linije oporavka postaje veoma složen, čak može dovesti do tzv. domino efekta, tj. do vraćanja u polazno stanje.



**Domino efekat**- greška u P2 zahteva povratak u prethodno stanje i procesa P2 i procesa P1, ali checkpoint-i nisu konzistentni: P2 je zabeležio prijem poruke m, ali P1 nije zabeležio slanje poruke m. Zato se P2 mora vratiti još u nazad, ali sada imamo da je P1 zabeležio prijem poruke m' a P2 nije zabeležio slanje m' itd.

# koordinisano beleženje stanja

- \* svaki proces kreira checkpoint nakon globalno koordinisane akcije.
- \* 2-phase blockirajući protokol
  - Koordinator šalje svim procesima u grupi poruku *checkpoint\_REQUEST*
  - Svi procesi primaju poruku, kreiraju checkpoint, prestaju sa slanjem aplikativnih poruka, i šalju koordinatoru poruku *checkpoint\_ACK*
  - Kada prikupi potvrde od svih procesa, koordinator šalje poruku *checkpoint\_DONE* svim procesima, nakon čega se procesi deblokiraju i nastavljaju sa radom.
  - Zapamćeno stanje u svim procesima je konzistentno
    - izbegnut je domino efekat

# Klijent-server komunikacija sa RPC

\* Cilj RPC je da sakrije komunikaciju tako što pokušava da poziv udaljene procedure izgleda kao poziv lokalne

- dok nema grešaka u sistemu ovo funkcioniše dobro
- ako nastupe greške nije uvek moguće sakriti razlike izmedju poziva udaljene i lokalne procedure.

\* Greške koje mogu nastupiti u RPC sistemu

- klijent nije u stanju da locira server
- zahtev upućen od klijenta ka serveru je izgubljen
- otkaz servera nakon prijema zahteva od klijenta
- odgovor servera klijentu izgubljen

\* svaka od ovih kategorija grešaka izaziva različite probleme i zahteva različita rešenja

# Klijent nije u stanju da locira server

\* Razlog može biti otkaz servera ili neodgovarajuća verzija klijent stub-a

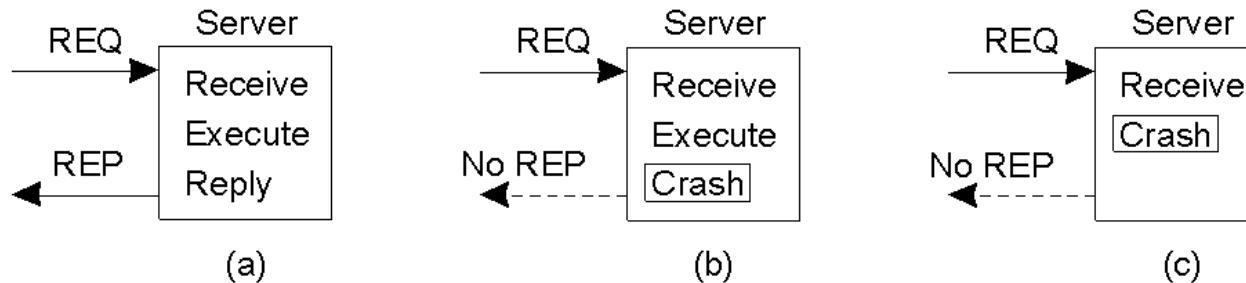
- novija verzija interfeisa i stub funkcija je instalirana na serveru
  - kada klijent pozove RPC binder neće biti u stanju da locira server
- neophodno je da kada se desi takva greška da se emituje exception
  - programer mora da napiše specijalnu proceduru koja se poziva kada se detektuje odredjena greška – transparentnost distribucije više ne postoji

## \* zahtev upućen od klijenta ka serveru je izgubljen

- najlakše se rešava

- klijent treba da startuje časovnik kada uputi zahtev
- ako istekne time out pre nego što stigne odgovor ili potvrda, vrši se retransmisija
- ako zahtev nije bio izgubljen, onda je ili server otkazao ili je izgubljen odgovor od servera ⇒ server mora biti u stanju da prepozna retransmisiju

# Otkaz servera nakon prijema zahteva od klijenta



a) normalan rad sistema

Server može da otkaže u bilo kom trenutku

b) otkaz servera nakon izvršenja zahteva, a pre slanja odgovora

c) otkaz servera pre izvršenja zahteva

Problem je što različite akcije treba da se preduzmu u slučaju b) i c) a klijent ne može razlikovati slučaj b) i c). U oba slučaja na strani klijenta će isteći time-out! U slučaju b) klijent treba da emituje grešku (exception), u slučaju c) samo da obavi retransmisiju

Dva rešenja su moguća:

1. klijent šalje zahteve sve dok ne dobije odgovor (at least one semantika – garantuje da će se RPC izvršiti bar jenom, a možda i više puta - primenljivo kod idempotentnih funkcija)
2. Odmah signalizirati grešku nakon isteka time-outa (at most one semantika – RPC će se izvršiti najviše jednom, a možda i ni jednom)

# Primer: otkaz servera

- \* najboje bi bilo obezbediti tačno jedno izvršenje (exactly one semanika), ali nema načina da se u opštem slučaju to garantuje.
- \* PRIMER:
  - prepostavimo da udaljena operacija zahteva štampanje teksta na udaljenom printeru preko servera za štampanje
  - kada klijent pošalje zahtev, on prima potvrdu da je zahtev prosledjen serveru
    - server može poslati klijentu potvrdu pre nego što prosledi nalog štampaču ili nakon što je tekst odštampan.
  - Prepostavimo da je server otkazao i zatim se oporavio od otkaza i obaveštava klijent da ponovo radi
    - problem je što klijent ne zna da li je njegov zahtev za štampanjem obavljen ili ne.
    - postoje 4 strategije koje klijent može da primeni:
      1. klijent može da odluči da nikada ne ponavlja zahtev, po cenu da tekst ne bude odštampan
      2. klijent uvek ponavlja zahtev, što može dovesti da tekst bude odštampan dva puta
      3. klijent će ponoviti zahtev ako nije primio potvrdu od servera
      4. klijent ponavlja zahtev samo ako je primio potvrdu za prethodno izdati zahtev

# Primer: otkaz servera (nast.)

- \* Na strani servera tri dogadjaja mogu da nastupe:
  - slanje poruke o okončaju zadatka (M)
  - štampanje teksta (P)
  - otkaz servera (C)
- \* Ovi dogadjaji mogu da nastupe u 6 različitih redosleda:
  1. M → P → C : otkaz servera se desio nakon slanja potvrde i štampanja teksta
  2. M → C (→ P): otkaz servera se desio nakon slanja potvrde, a pre štampanja teksta
  3. P → M → C: otkaz servera nastupio nakon štampanja i slanja potvrde
  4. P → C(→ M): tekst odštampan, server otkazao pre slanja potvrde
  5. C( → P → M): server otkazao pre nego što je bilo šta uradio
  6. C(→ M → P): server otkazao pre nego što je bilo šta uradio

\* zagrade () ukazuju na dogadjaje koji ne mogu da nastupe jer je server već otkazio

# Primer: otkaz servera (nast.)

Različite kombinacije klijent i server strategija pri otkazu servera i rezultati štampanja: OK – štampano tačno jednom, DUP – duplirano, ZERO – ni jednom

Klijent	Strategija M → P			Strategija P → M		
Ponavljanje zahteva	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
uvek	DUP	OK	OK	DUP	DUP	OK
nikad	OK	ZERO	ZERO	OK	OK	ZERO
Samo ako je primljen ACK	DUP	OK	ZERO	DUP	OK	ZERO
Samo ako nije primljen ACK	OK	ZERO	OK	OK	DUP	OK

Ne postoji ni jedna kombinacija klijent i server strategije koja će garantovati da će se štampanje obaviti samo jednom za svaku kombinaciju dogadjaja na server strani: klijent nikada ne može znati da li je tekst otštampan pre nego što je server otkazao ili ne

# odgovor servera klijentu izgubljen

## \* Klijent može ponovo poslati zahtev serveru

- da li je bezbedno ponovo poslati zahtev?
  - kod idempotentnih zahteva nema problema
    - Npr: čitanje fajla
  - *kontraprimer:* transfer novca sa bankovnog računa

## \* Rešenje: dodeliti redne brojeve zahtevima

- Server vodi računa o rednim brojevima klijentskih zahteva i može otkriti duplikat i odbiti da izvrši zahtev ponovo
- ili, postaviti u zagлавju poruke RETRANSMISSION bit kako bi se razlikovao originalni zahtev od retransmisije
  - ovo rešenje ne zahteva da server vodi računa o rednim brojevima zahteva klijenata
- u obe varijante server mora poslati ponovo poruku klijentu

# Distribuirani sistemi

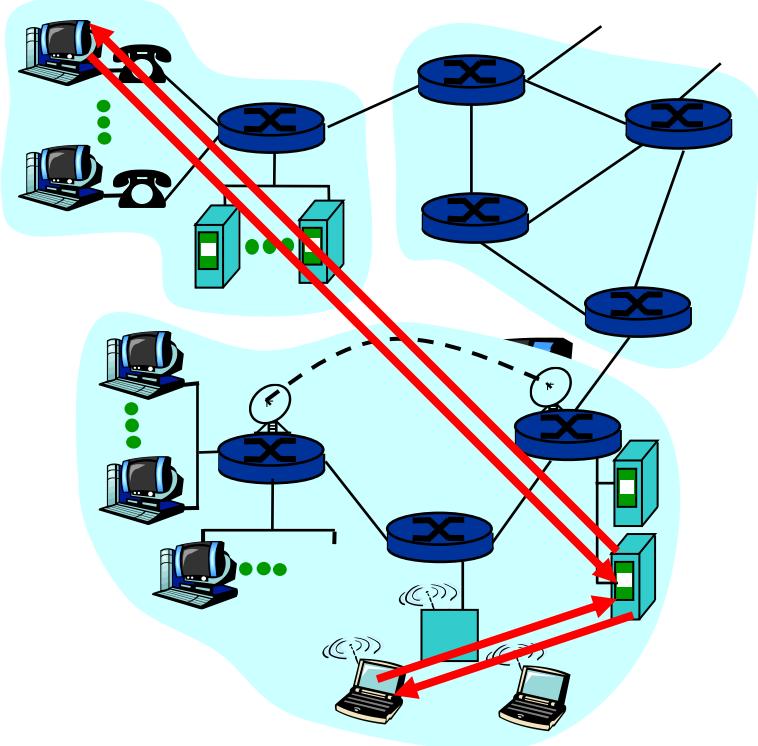


Peer-to-peer sistemi

# Arhitekture DS

- \* Klijent-server
- \* Peer-to-peer (P2P)
- \* Hibridne (klijent-server i P2P)

# Kljent-server



## server:

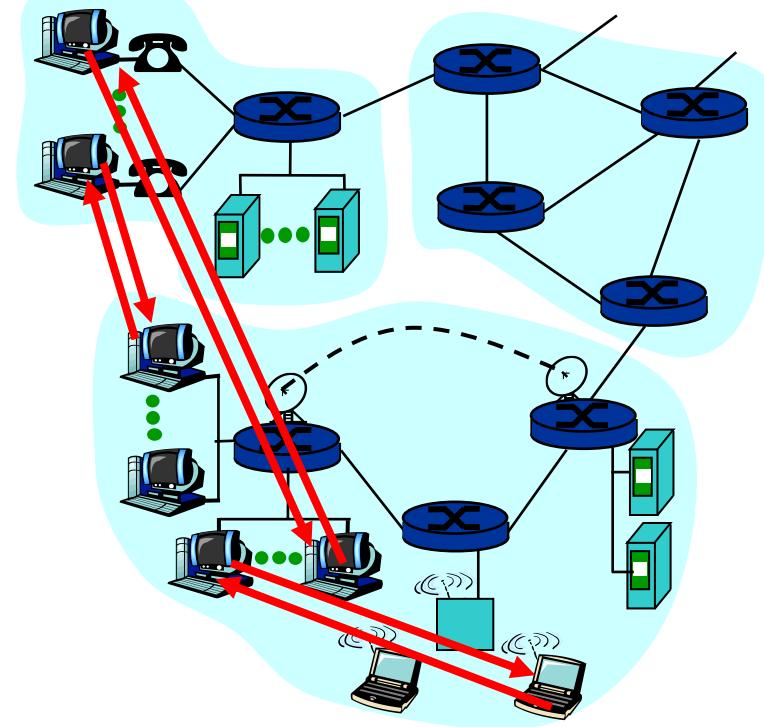
- Host je stalno prisutan
- Ima stalnu IP adresu
- Mogu se koristiti serverske farme zbog skaliranja

## klijenti:

- komuniciraju sa serverom
- Povremeno se povezuju na mrežu.
- Imaju dinamičke IP adrese.
- Ne komuniciraju direktno između sebe

# Prave peer-to-peer (P2P) arhitekture

- \* Ne postoji uvek prisutni server u centru aplikacije.
- \* Krajnji hostovi direktno komuniciraju
- \* Peer hostovi se povremeno konektuju u mrežu i imaju promenljive IP adrese.



Visoko skalabilne arhitekture.

Upravljanje je komplikovano

# Dve vrste p2p

## \* Aktivni p2p čvorovi formiraju apstraktnu logičku mrežu (overlay mreža)

- Ako peer X ima TCP konekciju sa peer Y, tada postoji poteg između X i Y.
- Svi aktivni peer čvorovi i potezi čine overlay mrežu
- Poteg ne pretstavlja fizički link.

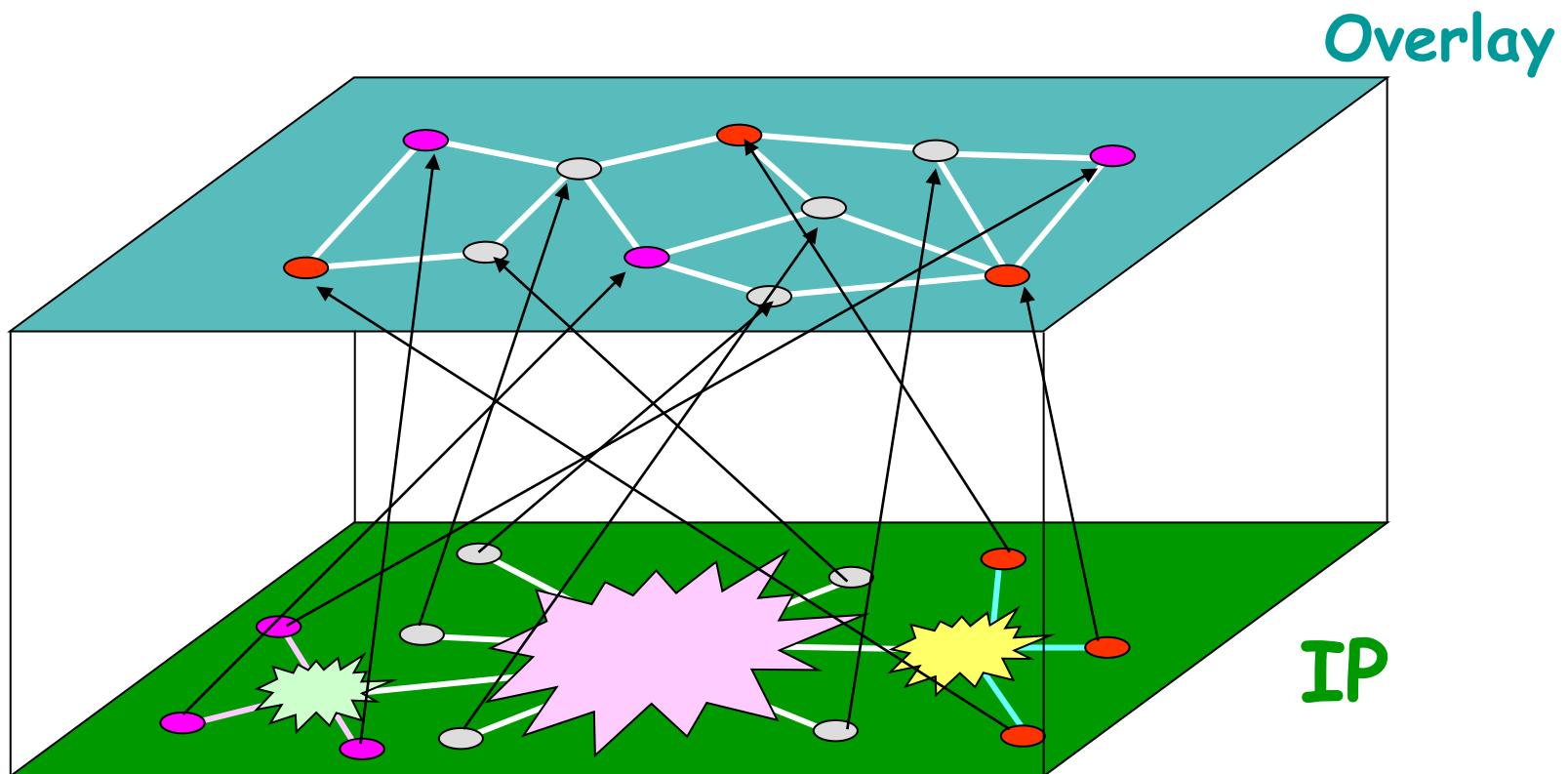
## \* Struktuirani

- Čvorovi formiraju overlay mrežu koja ima pravilnu strukturu
  - Prsten, hiperkup, stablo

## \* Nestruktuirani

- Overlay mreža nema pravilnu strukturu

# Overlay mreža



# Karakteristike p2p

## \* Eksplatišu resurse na granici globalne mreže

- Smeštajne kapacitete, sadržaje, CPU.
- svaki čvor donira svoje resurse sistemu (deli fajlove, donira procesorsko vreme,...)

## \* Poseduju značajnu autonomiju u poređenju sa centralizovanim sistemima

- Svaki čvor deluje i kao klijent i kao server
- Visok stepen skalabilnosti
- korektno funkcionisanje ne zavisi od nekog centralno administriranog sistema

## \* Čvorovi na ivici se periodično povezuju, sistem se stalno menja pridruživanjem ili odlaženjem čvorova

- Infrastruktura na kojoj su izgrađeni nije od poverenja i komponente nisu pouzdane.

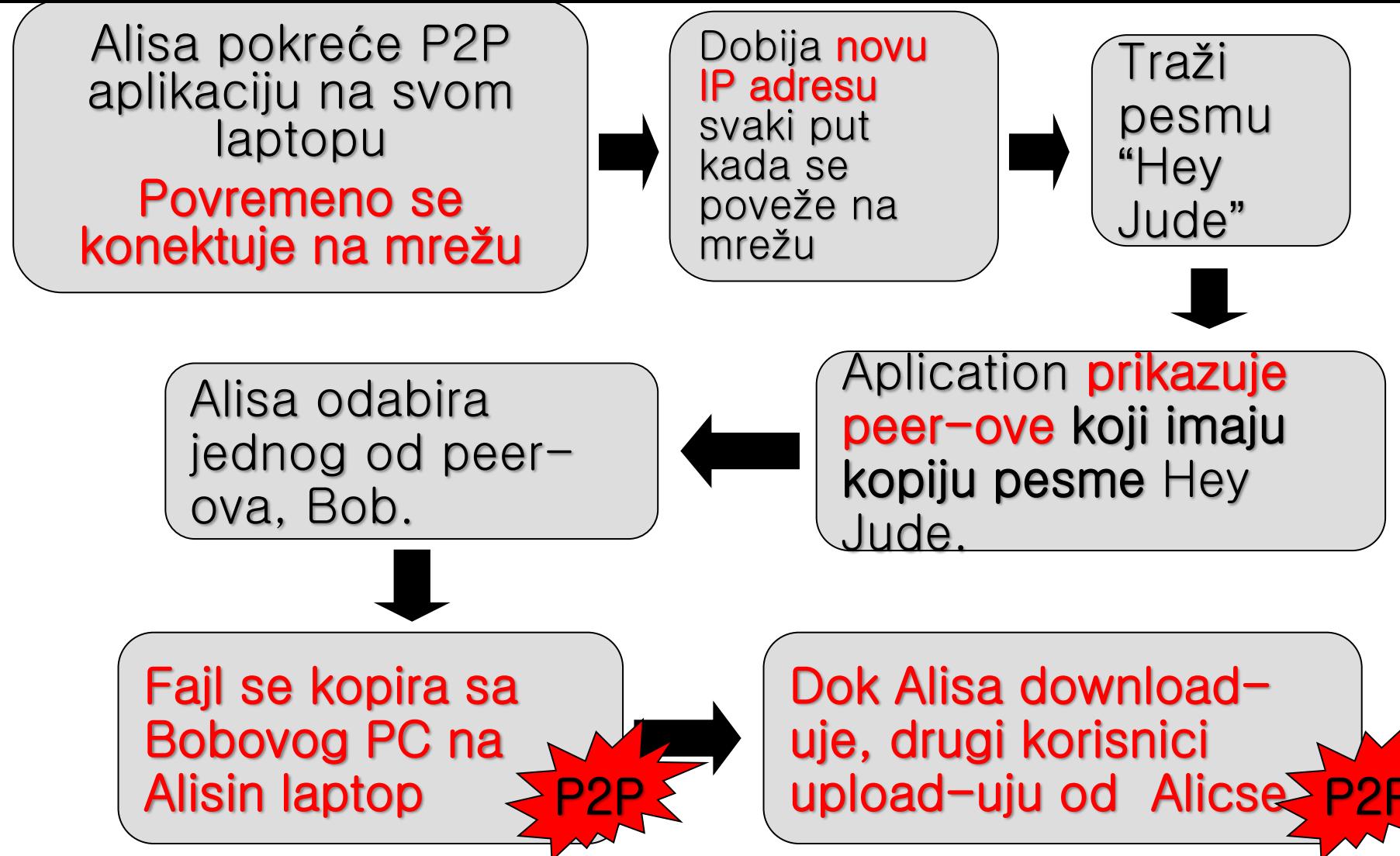
## \* Razlozi projektovanja P2P

- Iskoristiti računarsku moć i smeštajne kapacitete velikog broja računara koji se nalaze u mreži.

## \* Primene

- File-sharing aplikacije
- Distribuirane baze podataka
- Distribuirana izračunavanja (grid)
- Distribuirane igre
- Instant messaging

# P2P aplikacije: File Sharing

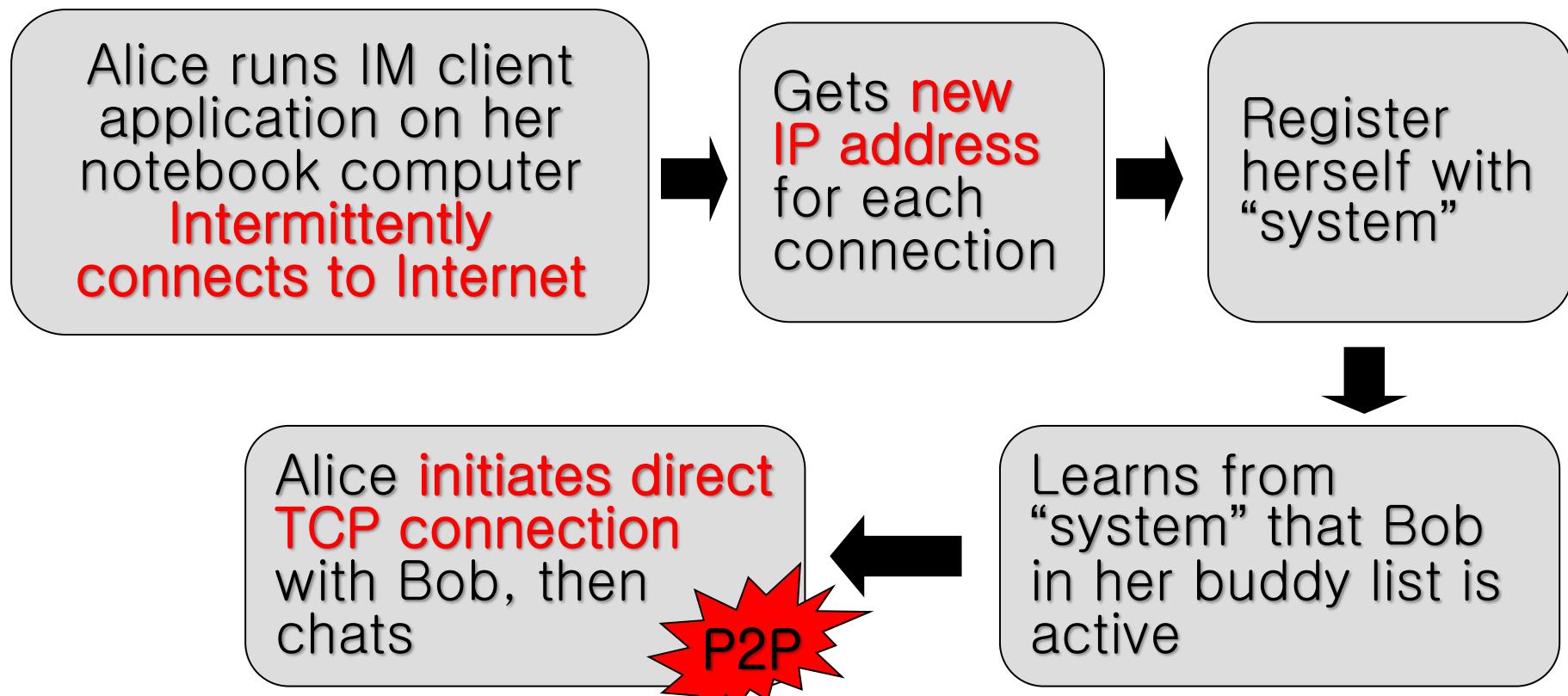


Napster, Gnutella, Kazaa, BitTorrent  
Chord, CAN, Pastry/Tapestry, ...

# P2P aplikacije: P2P komunikacija

\* Instant Messaging

\* Skype



# P2P aplikacije: distribuirano izračunavanje

## \* Distribuirano izračunavanje

- Seti@home – projekat pokrenut 1999

- Traženje vanzemaljske inteligencije
- Centralni sajt prikuplja podatke sa radio teleskopa
- Deli podatke na blokove od po of 300 Kbytes
- Peer koji želi da donira svoj CPU resurs, instalira klijent program koji se izvršava u pozadini (background)
- Peer uspostavlja TCP konekciju sa centralnim računarom i download-uje blok podataka
- Peer obavlja FFT, upload-uje rezultat, i uzima novi blok podataka

# P2P

## \* Osnovni problem kod P2P aplikacija je efikasno lociranje čvorova koji sadrže tražene podatke.

- kako čvor u P2P sistemu zna koji čvor sadrži podatke koji su mu potrebni?
- Mehanizmi za lociranje podataka mogu biti
  - centralizovani (prva generacija)
    - koristi se jedan ili više servera koji sadrže centralne direktorijume gde svi peer čvorovi registruju svoje sadržaje koje su spremni da dele sa drugim peer čvorovima
    - nije pravi p2p već hibridni (npr. Napster, ICQ, Skype)
  - decentralizovani nestruktuirani (druga generacija)
    - svaki peer čvor sadrži tabelu sa sadržajem koji je spreman da deli sa drugima (npr. Gnutela).
    - Overlay mreža nema pravilnu strukturu.
    - Pronalaženje željenog sadržaja se ostvaruje korišćenjem ograničene bujice.
  - decentralizovani struktuirani (treća generacija)
    - Overlay mreža ima pravilnu strukturu (npr. prsten, stablo hiperkub,...)
    - informacije o podacima koji se pretražuju su rasejane po čvorovima u P2P mreži.
    - Za pronalaženje informacija koriste se tabele slične ruting tabelama, ali se pretraživanje tabela ne obavlja na osnovu IP adresa već na osnovu ključa (tvz. DHT-distributed hash table - sistemi, npr. Chord)

# Primeri hibridnih p2p : klijent-server i P2P

## \* Napster

- Prenos fajlova P2P
- Pronalaženje fajlova je centralizovano:
  - Peer registruju svoje sadržaje na centralnom serveru
  - Peer kontaktira server da bi locirao fajl

## \* Instant messaging

- Čatovanje između dva korisnika je P2P
- Detekcija prisustva korisnika je centralizovana:
  - Korisnik registruje svoju IP adresu na centralnom serveru kada pristupi mreži.
  - Korisnici kontaktiraju centralni server da bi pronašli osobu sa kojom žele komunikaciju.

## \* Skype

- Centralizovani server: pronalaženje adrese udaljene strane
- Klijent-klijent konekcija je direktna (ne ide preko servera)

# P2P: centralizovani direktorijum

Originalni "Napster"  
projekat namenjen za  
razmenu mp3 fajlova

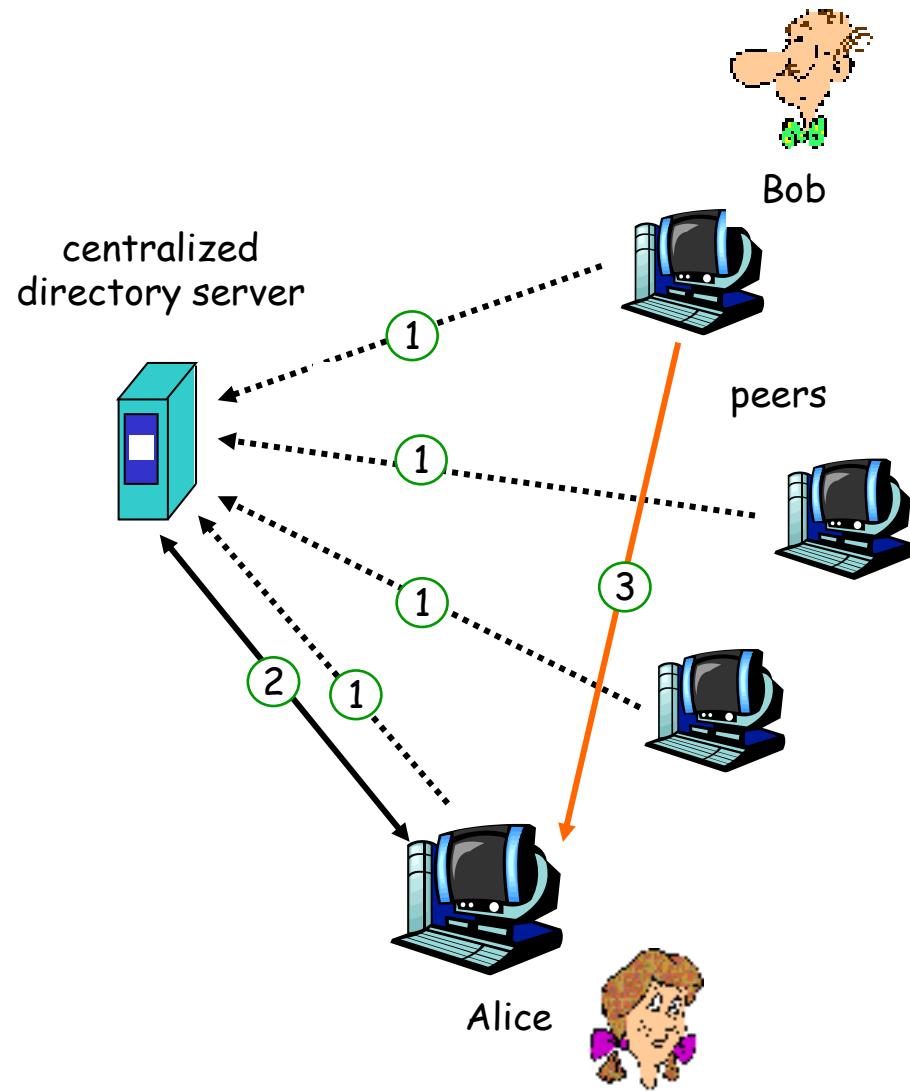
Bio aktivan od 1999-2001

1) Kada se peer konektuje,  
on informiše centralni  
server:

- IP adresa
- sadržaj

2) Alisa šalje upit za "Hey  
Jude"

3) Alisa zahteva fajl sa  
Bobovog računara.



# Centralizovani model (nast.)

- \* Kada aktivni peer dobije novi objekat, ili ukloni objekat, on informiše centrali server koji zatim ažurira svoju bazu podataka
- \* Da bi održao bazu podataka validnom, server mora biti u stanju da odredi kada se neki peer čvor diskonektuje.
  - Peer čvor se može diskonektovati tako što će zatvoriti P2P aplikaciju ili se diskonektovati sa Interneta.
  - Jedan način da server ima validne informacije je da se periodično vrši prozivka peer čvorova.
  - Ako server ustanovi da peer ne odgovara na prozivku, uklanja njegovu IP adresu iz baze podataka

# P2P: problemi sa centralizovanim direktorijumom

- \* Slaba otpornost na greške (otkaz centralnog direktorijuma je fatalan za ceo sistem)
- \* Centralni direktorijum je usko grlo –
  - u P2P sistemu može biti stotine hiljada korisnika, pa server mora održavati veliku bazu podataka i odgovarati na hiljade zahteva u sec.
- \* Prava izdavača-
  - produkcijske kuće su prigovarale da pomoći P2P file sharing aplikacije korisnici lako mogu doći do sadržaja koji su bili zaštićeni.
  - Kada postoji centralni direktorijumski server, zakonskom procedurom se kompanija može natrerati da ugasi server.
    - U potpuno decentralizovanoj arhitekturi to je mnogo teže učiniti.
- \* prenos fajlova je decentralizovan, ali je lociranje sadržaja visoko centralizovano

# Potpuno distribuirani model: Gnutella

## \* Potpuno distribuiran sistem

- Nema centralnog servera.

## \* besplatno dostupna aplikacija za deljenje fajlova

## \* Protokol se nalazi u javnom domenu

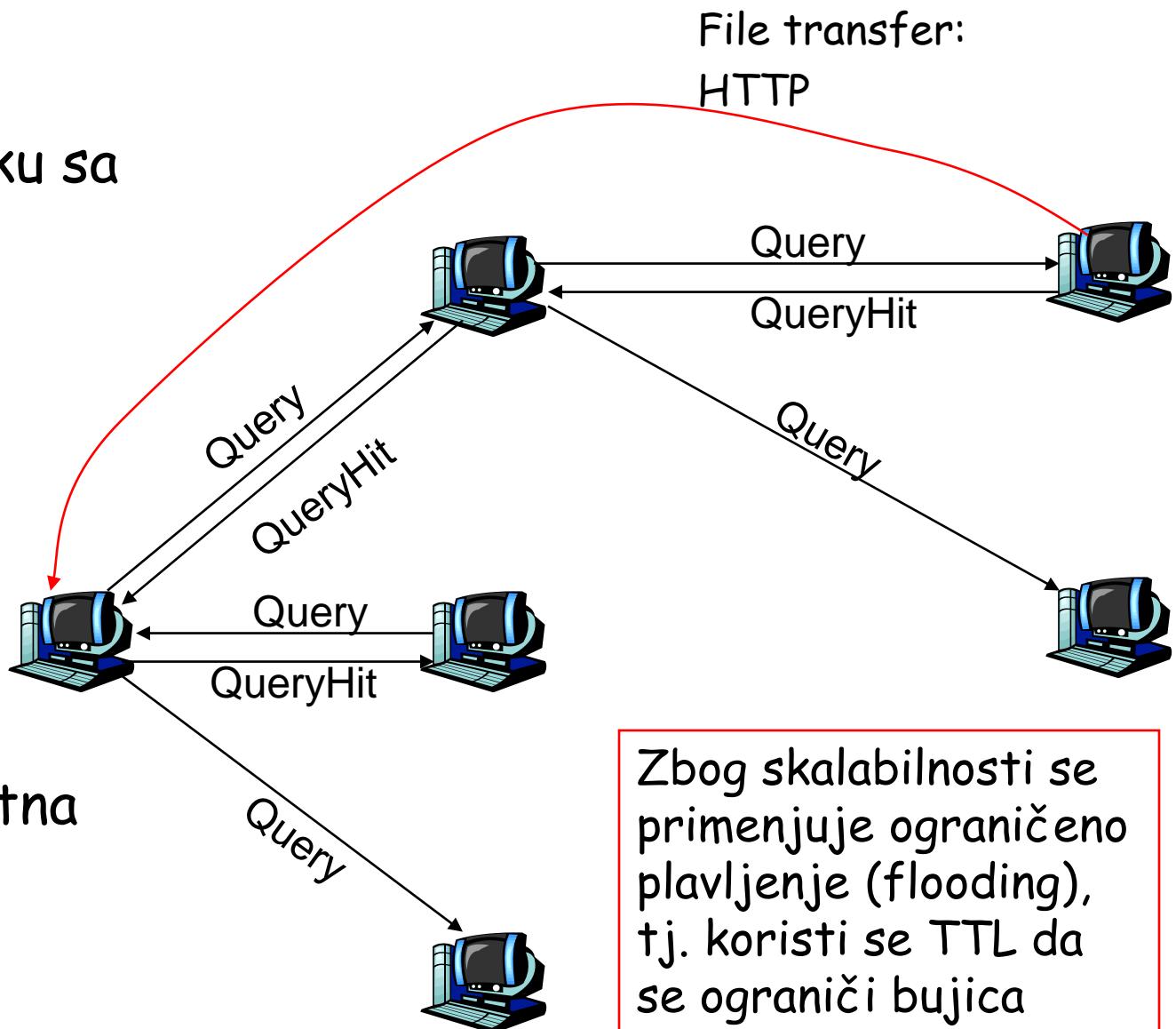
## \* Mnogo Gnutella klijenata implementira protokol.

- BearShare, Gnuclues, Morpheus, LimeWire (za windows)
- Mutella, Qtella, LimeWire (za unix/Linux)

## \* U Gnutelli klijenti formiraju nestruktuiranu abstraktnu logičku mrežu – overlay mreža (prekrivajuća mreža)

# Gnutella: lociranje i pretraživanje sadržaja

- ❑ peer šalje poruku sa upitom preko postojećih TCP konekcija
- ❑ peer čvorovi prosleđuju poruku svojim susedima u overlay mreži
- ❑ kada se locira traženi fajl povratna poruka se vraća obrnutim potom.



# Gnutella (nast.)

- \* Peer čvor može dobiti kao odgovor na svoj upit poruke od više drugih peer čvorova koji poseduju traženi fajl.
  - Odabira se jedan od peer čvorova i sa njim se uspostavlja direktna TCP konekcija
    - Šalje se HTTP GET poruka sa imenom željenog fajla
    - U HTTP odgovoru je sadržan željeni fajl

# BearShare klijent prozor

Search Show Artists ▾ Advanced Search

Music & Video All SEARCH BearShare Premium only

Top Albums:

Speakerboxxx... OutKast 2003 <a href="#">Download</a>	Speakerbox... OutKast 2003 <a href="#">Download</a>	Stankonia (P... OutKast 2000 <a href="#">Download</a>	1. OutKast 2. Chrome Dreams... 3. Goodie Mob 4. Mystikal
--	--	--	---

Track Title	Artist	Album	Length	Availability	Size	Get
The Whole World	OutKast	Big Boi & Dre Present...	4:55	★★★★★	4.5 MB	<a href="#">Ringtone</a>
Ain't No Thang	OutKast	Big Boi & Dre Present...	5:39	★★★★★	5.2 MB	<a href="#">Download</a>
Intro	OutKast	Big Boi & Dre Present...	1:07	★★★★★	1.0 MB	<a href="#">Download</a>
Aquemini	OutKast	Big Boi & Dre Present...	4:42	★★★★★	4.3 MB	<a href="#">Download</a>
Elevators (Me & You)	OutKast	Big Boi & Dre Present...	4:18	★★★★★	3.9 MB	<a href="#">Download</a>
Crumblin' Erb	OutKast	Big Boi & Dre Present...	5:17	★★★★★	4.8 MB	<a href="#">Download</a>
So Fresh, So Clean	OutKast	Big Boi & Dre Present...	4:02	★★★★★	3.7 MB	<a href="#">Download</a>
B.O.B.	OutKast	Big Boi & Dre Present...	4:38	★★★★★	4.3 MB	<a href="#">Download</a>
Spottieottiedopaliscious	OutKast	Big Boi & Dre Present...	5:58	★★★★★	5.5 MB	<a href="#">Download</a>
Movin' Cool (The After Pa...	OutKast	Big Boi & Dre Present...	3:59	★★★★★	3.7 MB	<a href="#">Download</a>
Southernplayalisticadillac...	OutKast	Big Boi & Dre Present...	4:11	★★★★★	3.8 MB	<a href="#">Download</a>
Player's Ball (Original Ver...	OutKast	Big Boi & Dre Present...	4:23	★★★★★	4.0 MB	<a href="#">Download</a>
Ms. Jackson	OutKast	Big Boi & Dre Present...	3:59	★★★★★	3.7 MB	<a href="#">Download</a>
Git Up, Git Out	OutKast	Big Boi & Dre Present...	7:26	★★★★★	6.8 MB	<a href="#">Download</a>
Funkin' Around	OutKast	Big Boi & Dre Present...	4:34	★★★★★	4.2 MB	<a href="#">Download</a>
Player's Ball (Original Version)	OutKast	Big Boi & Dre Present...	4:13	★★★★★	3.9 MB	<a href="#">Download</a>
Southernplayalisticadillacmuzik	OutKast	Big Boi & Dre Present...	4:12	★★★★★	3.8 MB	<a href="#">Download</a>
Funkin' Around	OutKast	Big Boi & Dre Present...	4:33	★★★★★	4.2 MB	<a href="#">Download</a>
Crumblin' Erb	OutKast	Big Boi & Dre Present...	5:15	★★★★★	4.8 MB	<a href="#">Download</a>

# Gnutella: pridruživanje peer čvora

1. Peer čvor X mora prvo pronaći neki drugi peer čvor koji se već nalazi u Gnutella mreži:
  - koristi listu potencijalnih kandidata za koje se zna da su često prisutni ili kontaktira Gnutella sajt koji sadrži takvu listu.
2. X sekvensijalno pokušava da uspostavi TCP konekciju sa peer čvorovima iz liste, dok sa nekim ne uspostavi konekciju, npr. Y
3. Nakon uspostavljanja TCP konekcije X šalje Ping poruku Y;
  - ping poruka sadrži IP adresu i broj porta izvora
  - Y prosleđuje Ping poruku drugim peer čvorovima u overlay mreži
  - Poruka sadrži i brojač koji se dekrementira pri prolasku kroz svaki peer čvor da bi se bujica držala pod kontrolom.
4. Svi peer čvorovi koji prime Ping poruku odgovaraju sa Pong porukom.
  - Pong poruka sadrži IP adresu peer čvora, broj fajlova koje čvor ima na raspolaganju i ukupnu veličinu fajlova u Kbyte
5. X može primiti mnogo Pong poruka.
  - X može nakon toga uspostaviti nove TCP konekcije i dodati nove potege u overlay mrežu.

# Gnutela - osobine

- \* Koristi ograničenu bujicu za pronalaženje željenog sadržaja
- \* Generiše se relativno veliki saobraćaj
- \* Zbog ograničene bujice može se desiti da se željeni sadržaj ne pronađe mada postoji u sistemu.

# KaZaA – hijerarhijska organizacija peer čvorova

\* Napster i Gnutella koriste dijametralno suprotne prilaze za lociranje sadržaja.

- Napster koristi centralizovani direktorijumski server i uvek locira sadržaj, ako postoji
- Gnutella koristi potpuno distribuiranu arhitekturu, ali locira samo sadržaje koji se nalaze u obližnjim peer čvorovima u overlay mreži.

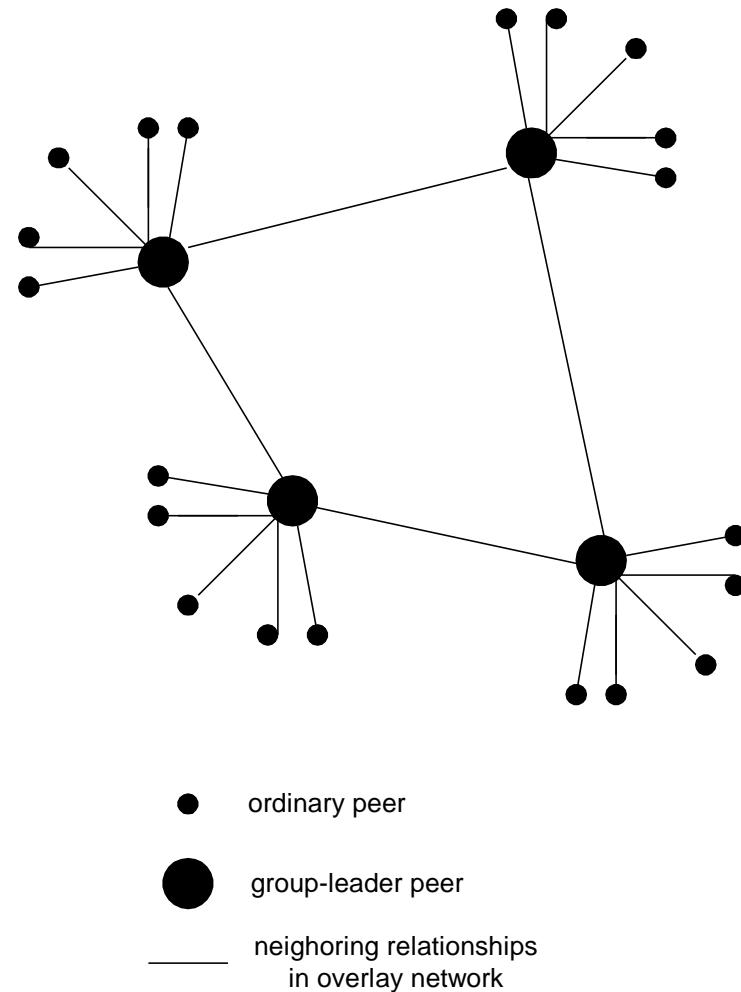
\* Kazaa koristi ideju i od Napstera i od Gnutelle

- Podseća na Gnutellu jer ne koristi server za pronalaženje sadržaja
- U KaZaA nisu sve peer čvorovi jednaki – peer čvorovi sa većom propusnom moći i većom povezanošću su lideri grupe.

\* svaki peer je ili lider grupe ili član grupe (pridružen lideru).

- Uspostavlja se TCP konekcija između peer čvora i njegovog lidera grupe.
- Postoji TCP konekcija između nekih lidera grupa..

\* Lider grupe vodi evidenciju o sadržajima svih članova grupe.



# Kazaa

\* Kada peer pokrene Kazaa aplikaciju, uspostavlja se TCP konekcija sa liderom grupe

- Peer informiše lidera o svim fajlovima koje poseduje i koji se mogu deliti.
- Na ovaj način lider grupe održava bazu podataka sa identifikatorima svih fajlova koje članovi grupe imaju i IP adresama članova grupe
- Lideri grupa su međusobno povezani TCP konekcijama kreirajući mrežu koja podseća na Gnutella mrežu.
- Kazaa deli informacije o sadržaju između različitih grupa
  - Lider grupe može preuzeti bazu podataka sa drugog lidera i pridodati je svoj bazi
  - Alternativno, lider može proslediti upit koji dobije drugim liderima sa kojima je povezan

# KaZaA: pretraživanje sadržaja (Querying)

- \* U Kazaa svaki fajl je identifikovan parom (ključ, ime\_fajla)
  - ključ se dobija primenom hash funkcije nad imenom fajla
  - Pored imena fajla može stajati i tekstualni opis
- \* Klijent šalje upit postavljanjem ključnih reči, svom lideru grupe.
- \* Ako lider grupe pronađe uparivanje ključnih reči sa nekim ključem, šalje odgovor za svako poklapanje sa:
  - ključ (dobijen hash funkcijom), IP addresa
- \* Ako lider ne pronađe poklapanje, prosleđuje upit drugim liderima grupa, koji eventualno pronalaze poklapanje i šalju odgovor.
- \* Klijent zatim selektuje fajl koji želi da privuče (download)
  - Prosleđuje se HTTP zahtev korišćenjem ključa kao identifikatora fajla peer čvoru koji sadrži željeni fajl.

# KaZaA – tehnike za poboljšanje performansi

- \* Svaki Kazaa peer može postaviti ograničenje za broj jednovremenih uploadovanja sa svog računara (3-7)
- \* Ako ima više zahteva, smeštaju se u red čekanja.
- \* Podsticajni prioriteti
  - Ako se u redu čekanja nađe više zahteva, pripritet se može dati peer čvoru koji je u prošlosti više puta uploadovao svoje fajlove nego što je downloadovao druge fajlove
- \* Paraleno privlačenje fajla.
  - Peer može iskoristiti polje opseg bajtova (byte range) u zaglavlju HTTP da zahteva različite delove fajla od više peer čvorova paralelno.

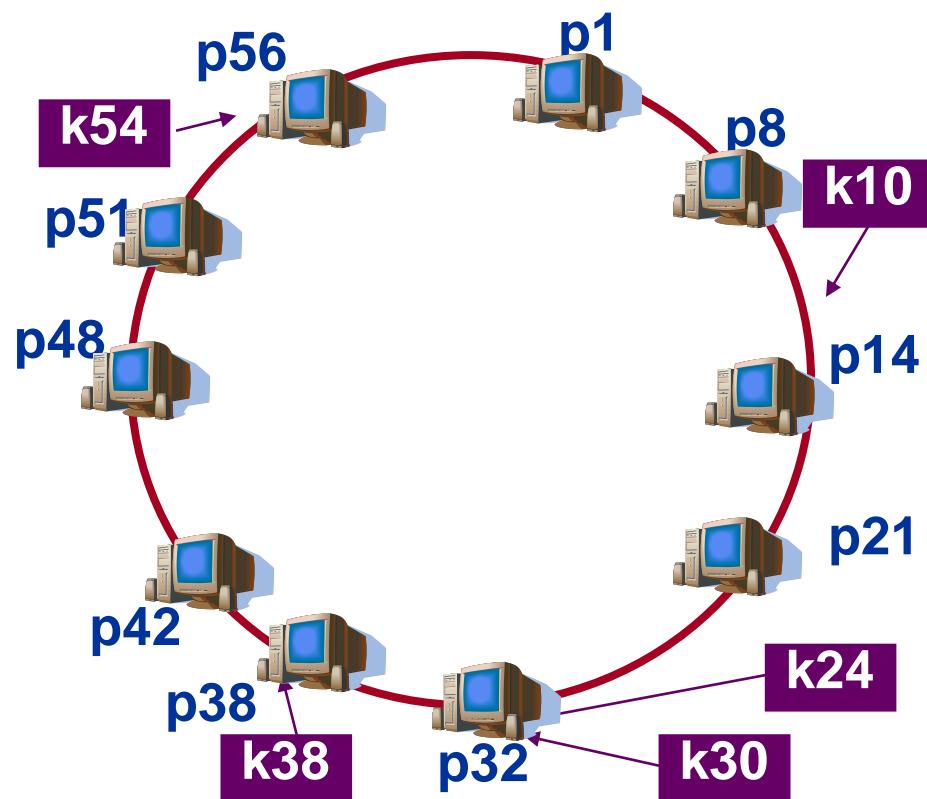
# Chord protokol – struktuirani p2p

- \* Ne postoji centralni server, kao kod Napstera, ili neki super peer (lider), kao kod Kazaa.
  - svi čvorovi su podjednako važni.
  - Problem sa Gnutelom je što se generiše veliki broj poruka u procesu lociranja željenog sadržaja (koristi se bujica)
  - Kod Chord protokola broj koraka potrebnih da se locira željeni sadržaj raste logaritamski sa brojem čvorova u mreži
    - ako ima  $N$  čvorova, sadržaj se pronalazi za najviše  $\log N$  koraka.
    - zbog ove osobine Chord je jako skalabilan i može se koristiti za velike sisteme.
  - Koristi DHT (distribuirane hash tabele) za lociranje željenog objekta
- \* Chord protokol definiše
  - kako pronaći sadržaj na osnovu ključa
  - kako se čvor pridružuje sistemu
  - kako se obavlja oporavak od otkaza nekog čvora.

# Chord protokol (nast.)

- \* Koristi m-bitne identifikatore raspoređene na prstenu veličine  $2^m$  (chord ring)
- \* jedinstveni prostor se koristi za identifikovanje peer čvorova i sadržaja koji se nalaze u čvorovima u mreži
  - vrši preslikavanje peer čvorova i objekata u identifikatore na prstenu korišćenjem hash funkcije SHA-1.
    - identifikator peer čvora se dobija primenom SHA-1 funkcije nad IP adresom čvora i brojem porta
      - ID=hash(IP, broj porta)
      - svaki peer čvor zna identifikator svog neposrednog prethodnika (predecessor) i sledbenika (successor)
    - ključ za lociranje željenog objekta se dobija primenom SHA-1 funkcije nad imenom fajla
      - $k=\text{hash}(\text{file name})$
    - fajl kome odgovara ključ  $k$  se dodeljuje peer čvoru čiji je ID  $p$ , tako da važi da je  $k \leq p$ , i ne postoji peer  $q$  za koji važi da je  $k \leq q$  i  $q < p$ 
      - $p$  je prvi čvor na ringu posmatrano u smeru okretanja kazaljke na satu koji ima ID veći od ključa  $k$
    - za čvor  $p$  se kaže da je sledbenik (successor) od  $k$ ,  $p=\text{succ}(k)$

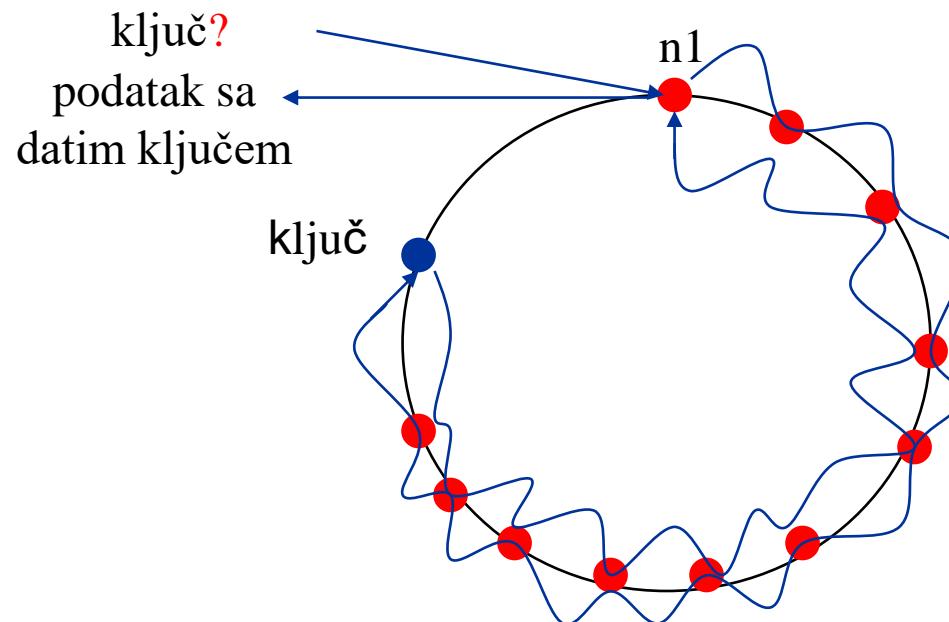
$m=6$  (6-to bitni identifikatori)



# pronalaženje sadržaja

\* svaki čvor u ringu zna svog naslednika (successor) i prethodnika (predecessor).

- sukcesivnim kontaktiranjem naslednika se uvek može pronaći željeni sadržaj (ako postoji)



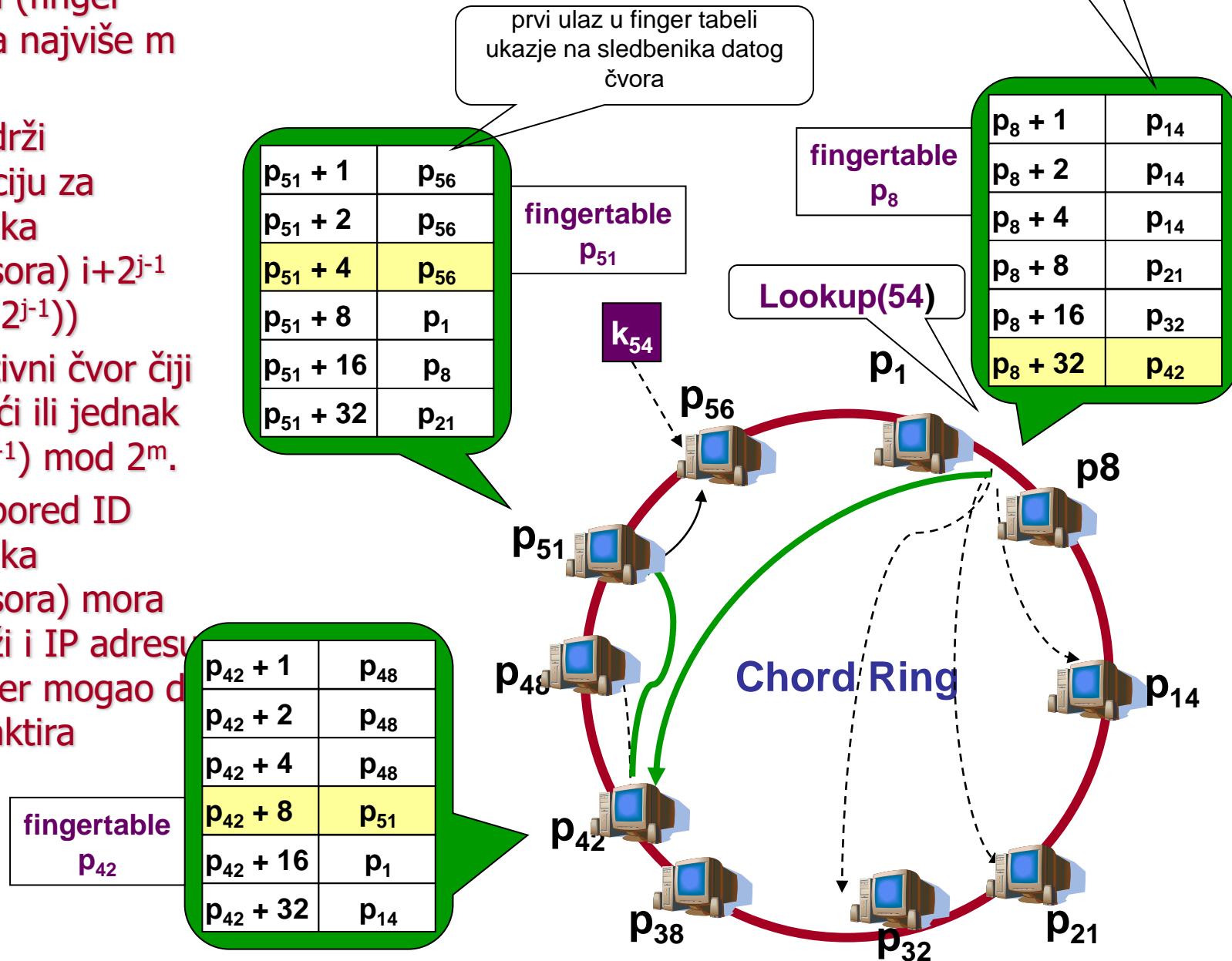
peer / sadrži tabelu rutiranja (finger table) sa najviše m vrsta.

j-ta vrsta sadrži informaciju za naslednika (successora)  $i+2^{j-1}$   
 $(\text{succ}(i+2^{j-1}))$

to je prvi aktivni čvor čiji  
je ID veći ili jednak  
od  $(i+2^{j-1}) \bmod 2^m$ .

svaka vrsta pored ID  
naslednika  
(successora) mora  
da sadrži i IP adresu  
da bi peer mogao da  
se kontaktira

## Brži način za pronalaženje sadržaja



# karakteristike finger tabela

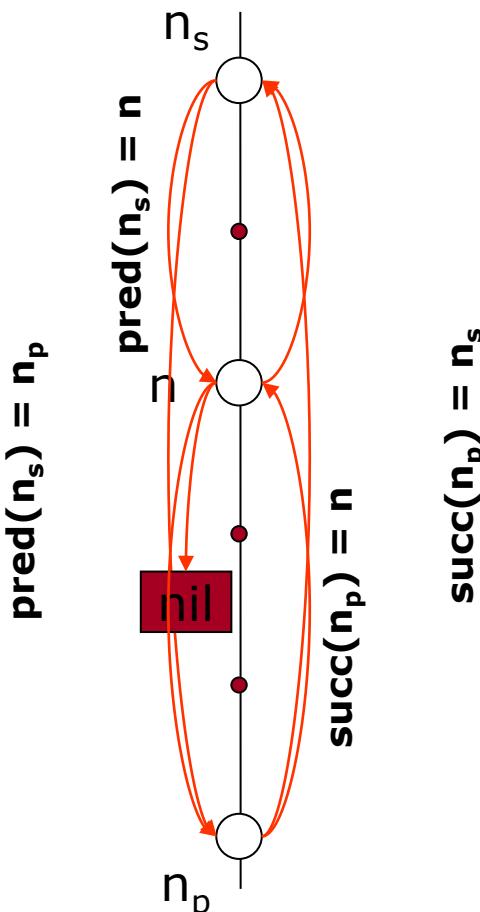
- \* svaki čvor sadrži informacije o malom broju drugih čvorova, i zna više o čvorovima koj su mu bliži nego očvorovima koji su dalji od njega (posmatrano na prstenu).
- \* finger tabela čvora u opštem slučaju ne sadrži dovoljno informacija da se odredi naslednik (successor) za proizvoljni ključ  $k$
- \* sukcesivnim pretraživanjem finger tabela će se pronaći čvor koji ukazuje na naslednika (successora) datog ključa.
- \* Kada se od čvora traži da pronadje lokaciju za zadati ključ –  $\text{lookup}(\text{ključ})$ , on će pretražiti svoju finger tabelu da bi pronašao prethodnika sa najvećim identifikatorom u odnosu na dati ključ
  - kada se pronadje takav čvor, njemu se prosledjuje  $\text{lookup}(\text{ključ})$
  - Ako nema informacija u svojoj finger tabeli prosleđuje zahtev svom sledbeniku
  - ponavljanjem ovih koraka rekurzivno se određuje čvor koji je odgovoran za podatke sa zadatim ključem.

# pridruživanje čvora

- \* da bi se obezbedilo pronalaženje sadržaja neophodno je obezbediti da pointer na sledeći čvor (successor) uvek bude validan bez obzira da li se novi čvorovi pridružuju ili odlaze iz mreže.
  - ako ostali ulazi u finger tabeli nisu validni to će samo usporiti pronalaženje sadržaja
  - svaki čvor u chord prstenu periodično izvršava protokol za stabilizaciju da bi otkrio novopridružne čvorove
- \* Kada se novi čvor,  $n$ , pridružuje mreži on poziva funkciju  $join(n')$ , gde je  $n'$  bilo koji poznati čvor na chord prstenu
  - join funkcija traži od čvora  $n'$  da pronadje neposrednog sledbenika za čvor  $n$
  - ova funkcija ne obaveštava ostale čvorove u mreži o prisustvu čvora  $n$ .
    - ona samo omogućva čvoru  $n$  da nadje svog sledbenika (successora)

# Stabilizacija nakon pridruživanja

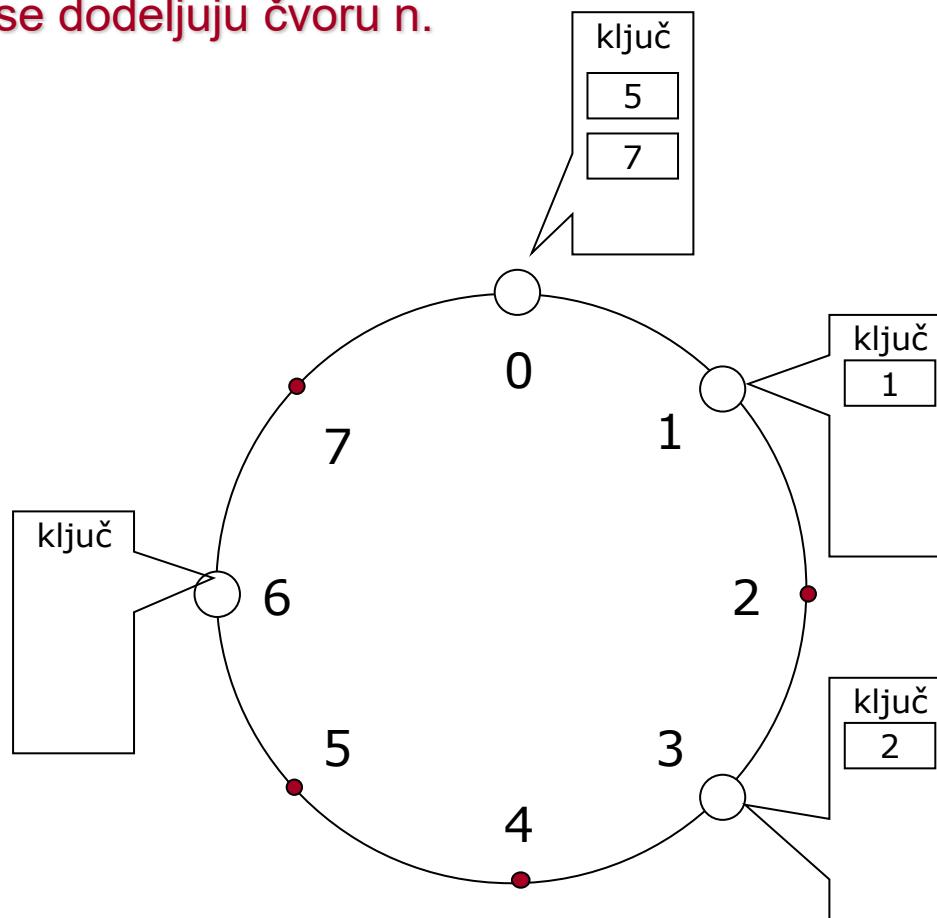
$\check{c}vor n_s$  je sledbenik  
 $\check{c}vora n_p$



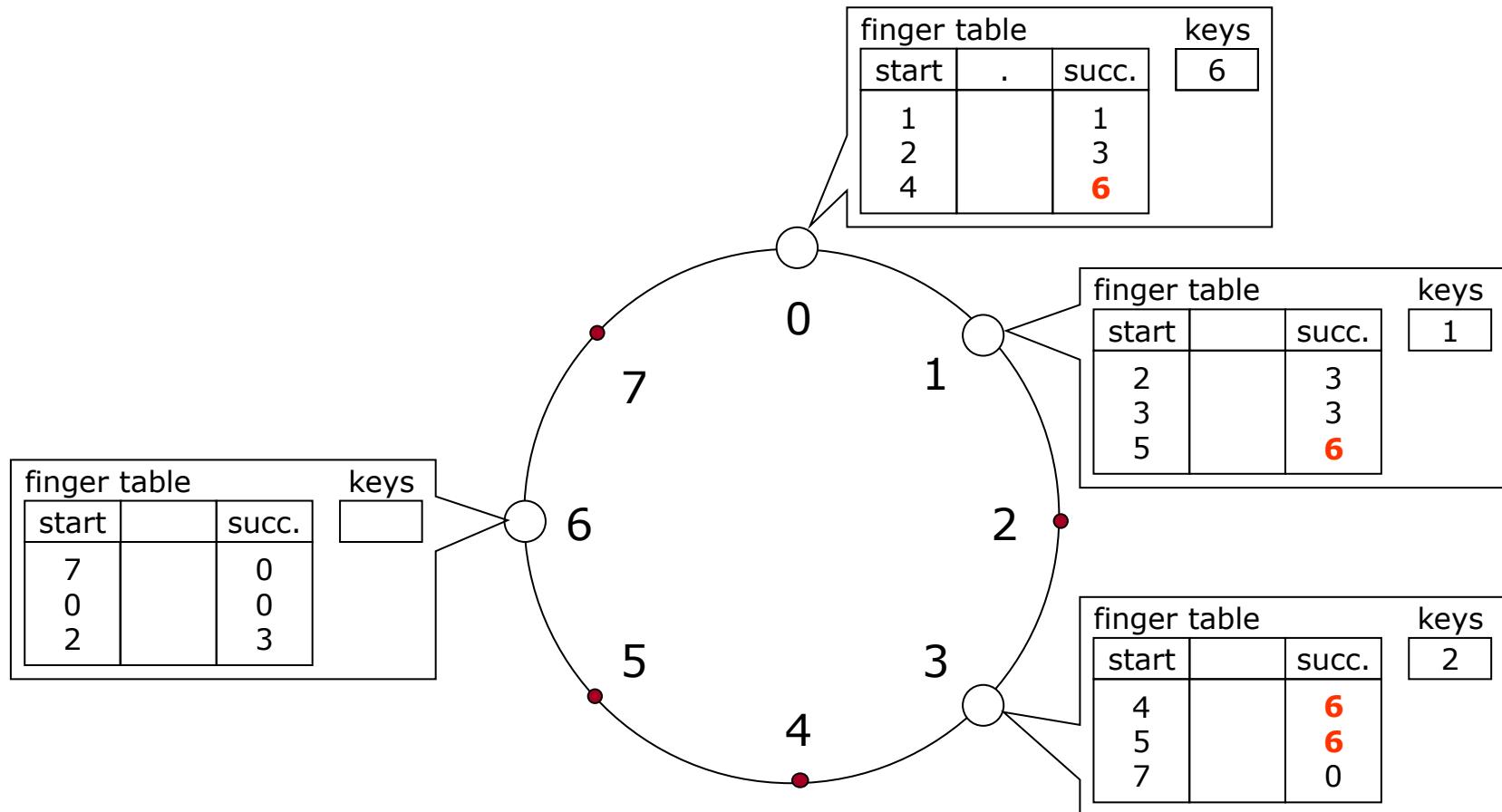
- \* **n se pridružuje**
    - predecessor = nil (ne zna prethodnika)
    - n sazna da je njegov sledbenik čvor  $n_s$  (preko nekog čvora n')
    - n obaveštava  $n_s$  da je on novi prethodnik za njega
    - $n_s$  označava da je n njegov prethodnik
  - \*  **$n_p$  pokreće protokol za stabilizaciju**
    - $n_p$  pita čvor  $n_s$  koji je njegov prethodnik (to je sada čvor n)
    - $n_p$  sazna da je n njegov sledbenik (jer mu je čvor  $n_s$  odgovorio da je njegov prethodnik čvor n)
    - $n_p$  obaveštava čvor n
    - n označava  $n_p$  kao svog prethodnika
  - \* **svi pointeri na prethodnike (predecessor) i sledbenike (successor) su sada korektni**
  - \* **ostali ulazi u finger tabele treba da se koriguju, ali protokol može da pronadje sadžaje.**

# pridruživanje čvora

kada se čvor  $n$  pridruži mreži, neki ključevi koji su bili dodeljeni njegovom sledbeniku sada se dodeljuju čvoru  $n$ .

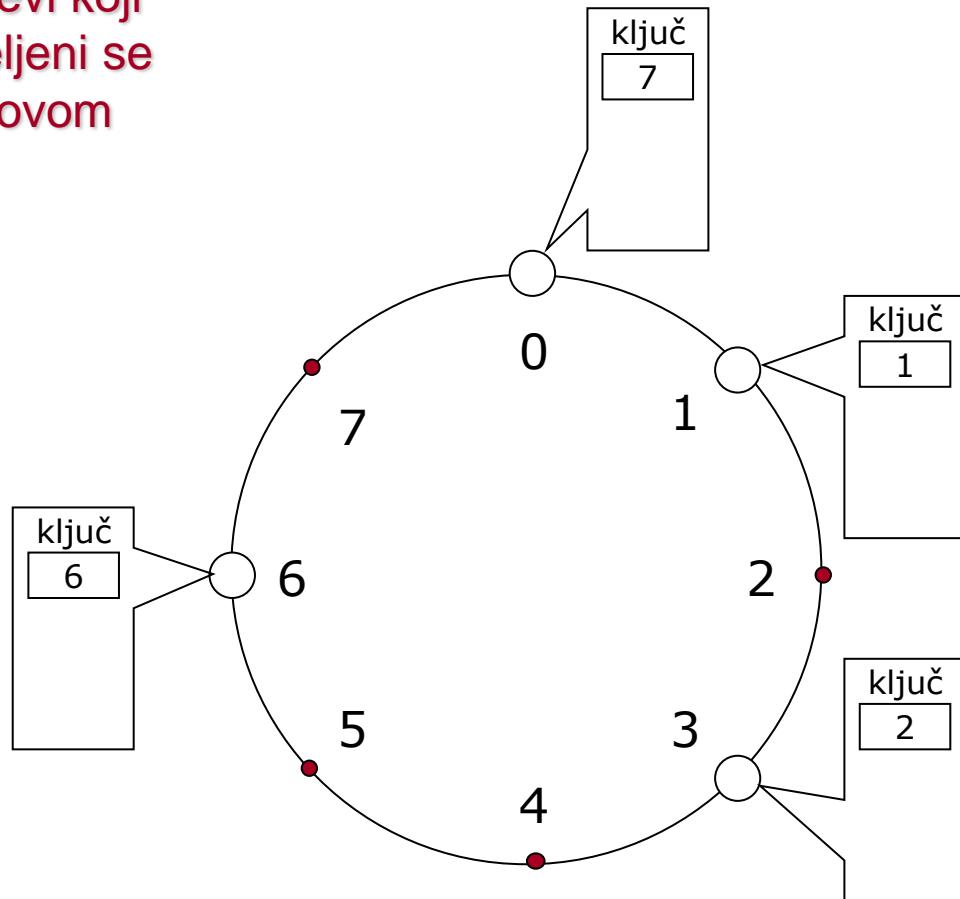


# pridruživanje čvora i promene u finger tabelama

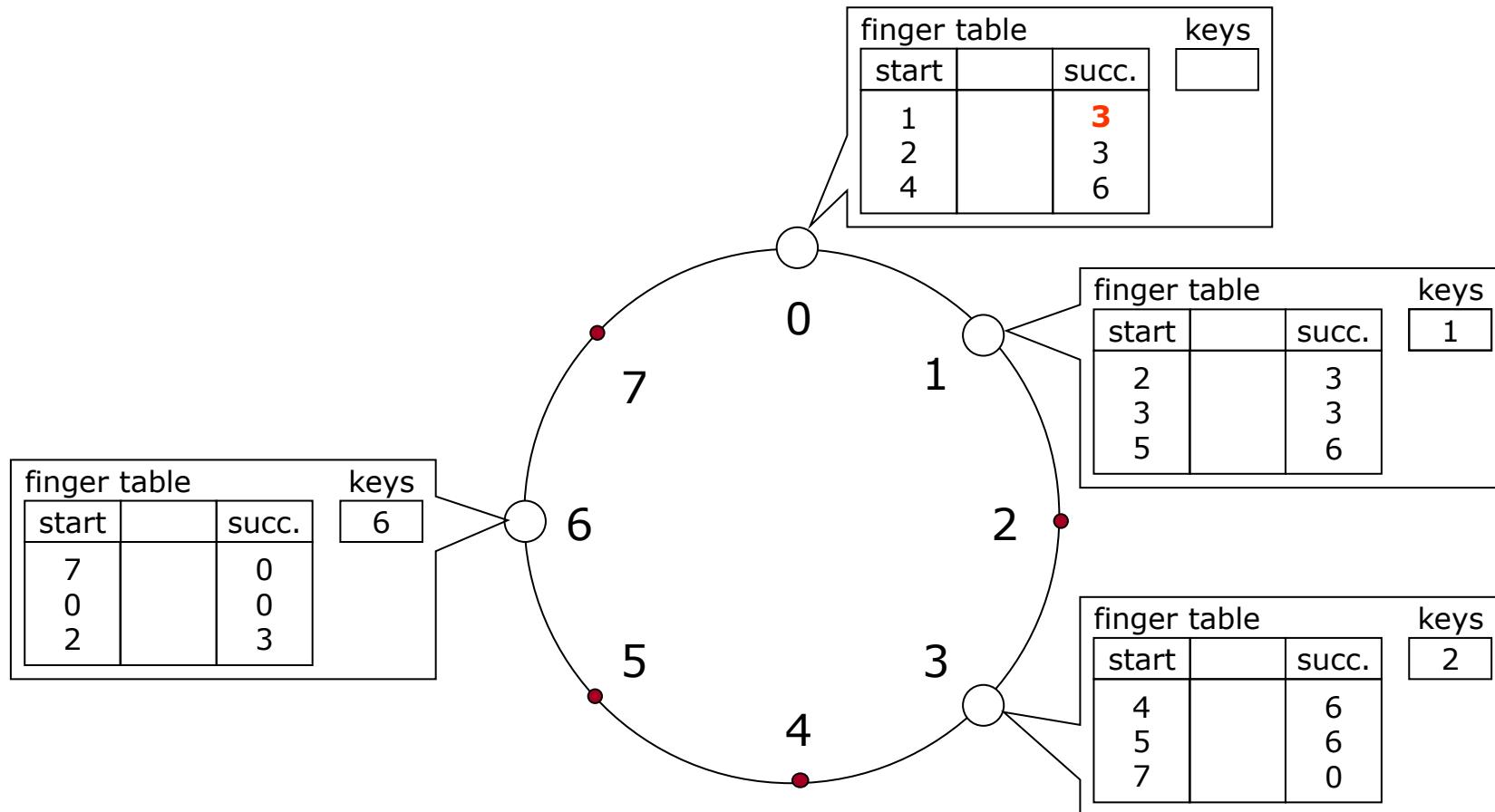


# napuštanje mreže

\*kada čvor napušta  
mrežu svi ključevi koji  
su mu bili dodeljeni se  
prebacuju njegovom  
sledbeniku.



# odlazak čvora iz mreže i promene u finger tabelama



# otkaz čvora

- \* ključni korak u oporavku od greške je održavanje korektne liste pointera na sledbenike.
- \* svaki čvor održava listu r najbližih sledbenika na ringu.
- \* ako čvor n primeti da je njegov neposredni sledbenik otakazio, on zamenjuje pointer pointerom prvog živog čvora u svojoj listi
  - sve dok čvor ima korektan pointer na svog sledbenika moguće je pronaći željeni sadržaj (ako postoji)

# Chord protokol - zaključak

- \* potpuno decentralizovan sistem – ni jedan čvor nije važniji od nekog drugog.
- \* omogućva dobro balansiranje opterećenja u P2P mreži ravnomernom distribucijom ključeva po čvorovima
- \* skalabilan – cena pronalaženja sadržaja raste logaritamski sa porastom broja čvorova
  - zahvaljujući tome moguće je formirati P2P sisteme sa velikim brojem čvorova

# Zaključak

## \* P2P sistemi su atraktivni iz nekoliko razloga :

- Barijere za gradnju i proširenje takvih sistema su male, jer ovi sistemi ne zahtevaju posebno administrativne i finansijske sporazume, za razliku od centralizovanih sistema..
- P2P sistemi pružaju mogućnost okupljanja i iskorišćavanja ogromnih kapaciteta za pamćenje podataka na računarima koji se nalaze na Intrernetu.
- Decentralizovana i distribuirana priroda P2P sistema ih čini otpornim na greške ili zlonamerne napade, čineći ih idealnim za pamćenje podataka kao i za duga izračunavanja.

# Distribuirani sistemi



## Distribuirani fajl sistemi

# Fajl sistem

## \* fajl

- imenovani skup podataka trajno zapamćen na disku (ili nekom drugom medijumu, CD, DVD,...)

## \* Direktorijumi

- Specijalna vrsta fajlova koja služi za organizaciju drugih fajlova
- Sadržaj direktorijuma može ukazivati na druge direktorijume ili fajlove

## \* Fajl sistem je komponenta operativnog sistema odgovorana za organizaciju, skladištenje, imenovanje, pretraživanje, deljenje i zaštitu fajlova.

- Obezbeđuje korisnički interfejs za rad sa fajlovima.
  - operacije nad fajlovima
    - Open, Close, Read, Write, Seek,
  - Operacije nad direktorijumima
    - Lookup, List, Add, Remove, Rename, ...

# smeštanje fajlova na disku

## \* kontinualno

- blokovi fajla su zapamćeni u sukcesivnim sektorima na disku
  - Direktorijum ukazuje na lokaciju prvog bloka

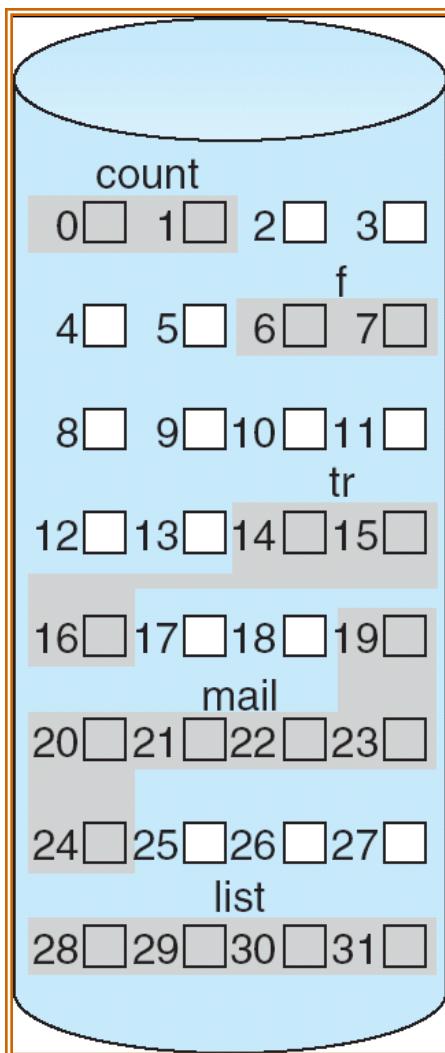
## \* ulančano

- blokovi jednog fajla mogu biti rasejani na disku, kao lančane liste
  - Direktorijum ukazuje na lokaciju prvog bloka

## \* Indeksirano

- poseban blok (indeks blok) sadrži pointere na blokove fajla
  - direktorijum ukazuje na lokaciju indeks bloka

# kontinualno smeštanje



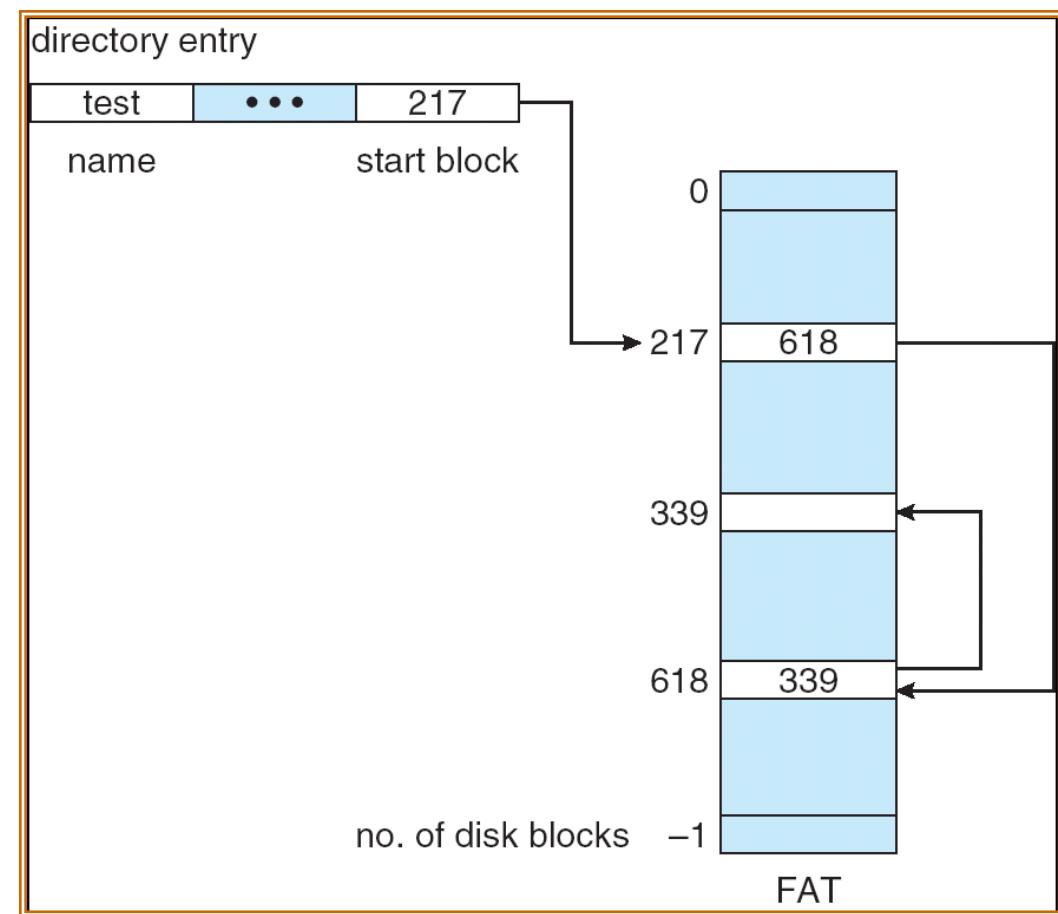
directory		
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

idealno za velike, statične fajlove (baze podataka, OS kod, Multi-media video i audio)

- CD-ROM, DVD

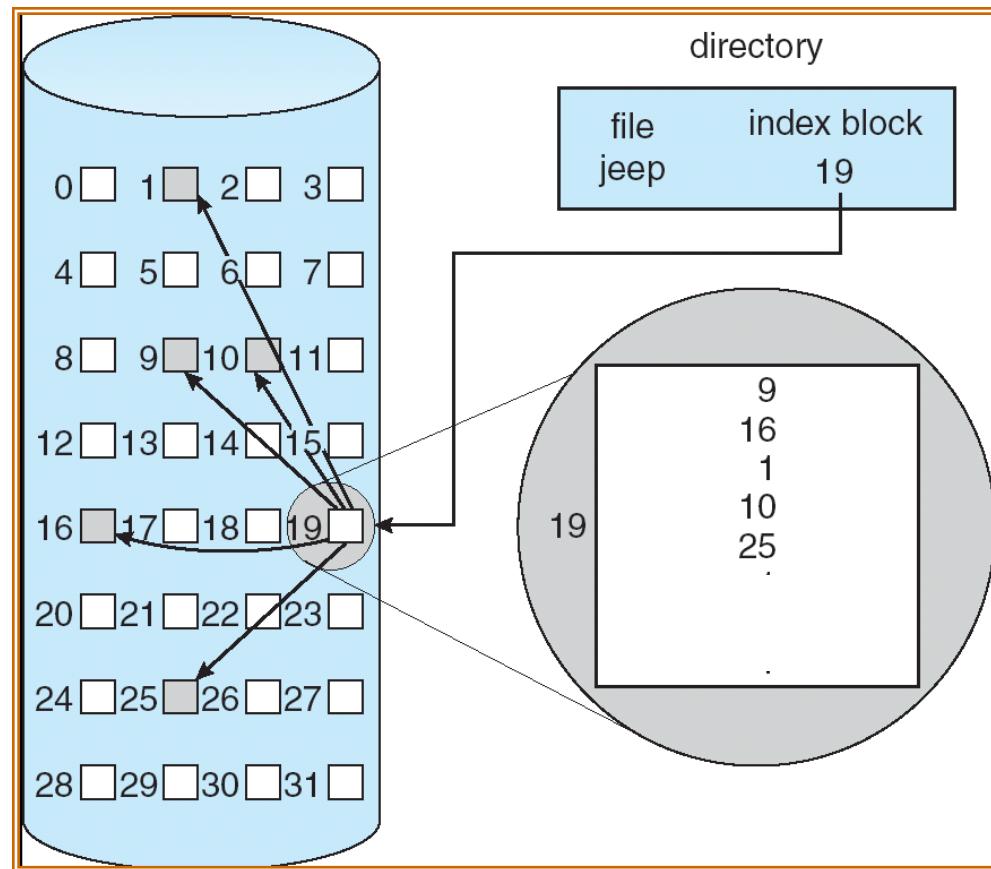
# ulančano smeštanje

- Direktorijum ukazuje na prvi blok fajla
- svaki blok ukazuje na sledeći blok (ili *EOF*)



# indeksirana alokacija

- direktorijum ukazuje na indeks blok *i-node*:
  - i-node sadrži adresu svakog bloka fajla
  - svakom bloku se pristupa preko i-node
  - za svaki pristup fajlu zahteva se pristup i-node

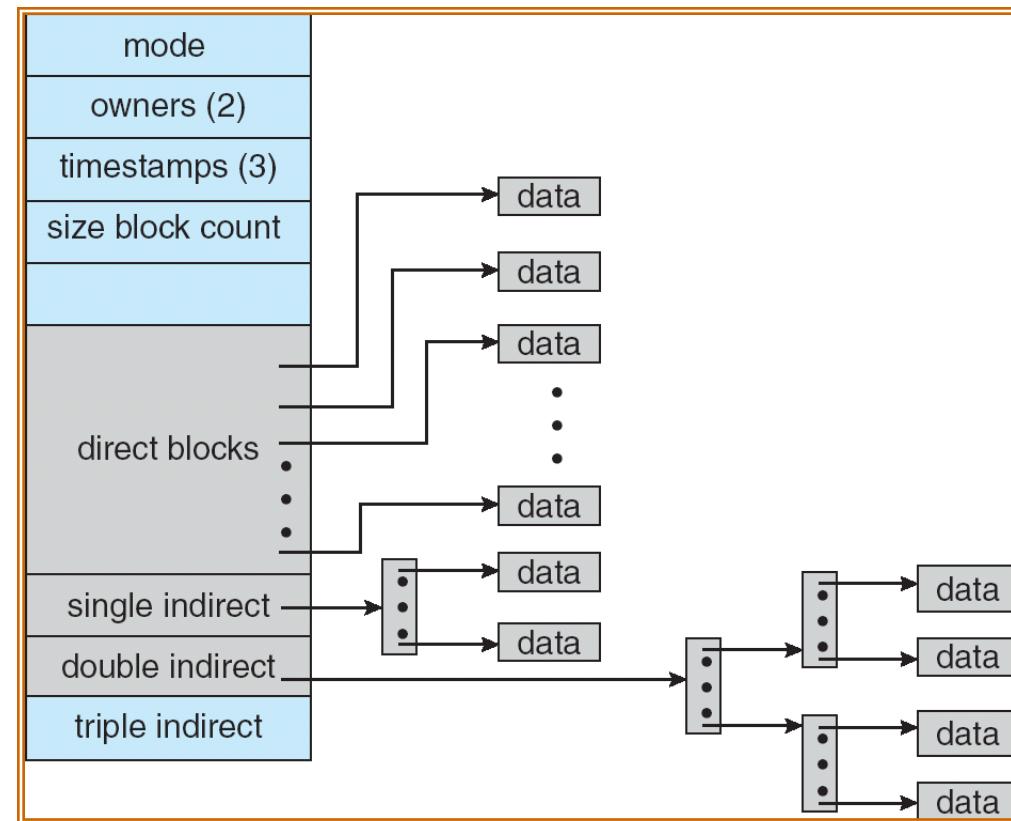


# izgled i-node u UNIX/LINUX

\* Referenciranje fajlu se ne vrši na osnovu imena već na osnovu i-node

- Svaki direktorijum sadrži tabelu sa imenom fajla i brojem inode. (može se izlistati sa ls -i)

- Svi inode su zapamćeni u spoljajalnoj lokaciji na disku – super blok
- Inode pamti atribute fajla i pointere na blokove u kojima su zapamćeni podaci datog fajla (12 direktnih pointera, 1 indirektni, 1 dvostruki indirektni i 1 trostruki indirektni pointer).



# Distribuirani fajl sistem(DFS)

- \* Implementacija DFS zahteva sve komponente tradicionalnog fajl sistema sa dodatnim komponentama za klijent-server komunikaciju, imenovanje i lociranje fajlova u distribuiranom sistemu.
- \* DFS -Faj listem se nalazi na više autonomnih mašina.
  - omogućava da se pristup udaljenim fajlovima obavlja isto kao lokalnim fajlovima.
- \* DFS su bazirani na klijent-server modelu.
  - DFS pruža API za pristup fajl sistemu
    - API se razlikuju za različite DFS
- \* DFS se sastoji od
  - direktorijumskog servisa
    - bavi se imenovanjem i lociranjem fajlova
    - vrši preslikavanje tekstualnog imena fajla u jedinstveni fajl identifikator na osnovu kojeg se može locirati fajl u DS
    - imena moraju da budu jedinstvena u celom DS, a ne unutar jednog fajl sistema
    - direktorijumski servis može biti na odvojenoj mašini (sadrži i-node)
      - u tradicionalnim fajl sistemima to nije slučaj (direktorijumski servis i fajl servis su na istoj mašini)
  - fajl servisa
    - bavi se održavanjem sadržaja fajla
    - može da se uradi lokalno u svakom serveru u kome se fajl nalazi

## \* DFS treba da obezbedi

- transparentnost pristupa.

- Udaljenim i lokalnim fajlovima se pristupa na isti način

- Lokaciona transparentnost

- Isti način imenovanja se koristi i za lokalne i za udaljene fajlove; ime fajla ne otkriva njegovu fizičku lokaciju

- Transparentnost konkurencije

- Svi klijenti imaju isti pogled na stanje sistema; ako neki proces modifikuje fajl, procesi na istom ili udaljenom sistemu će videti modifikaciju na koherentan način.

- Transparentnost replikacije

- Fajlovi mogu biti replicirani na više servera; klijent nije svestan replikacije

- Migraciona transparentnost

- Fajl može da migrira sa jednog servera na drugi a da to ne utiče na pristup fajlu od strane klijenta

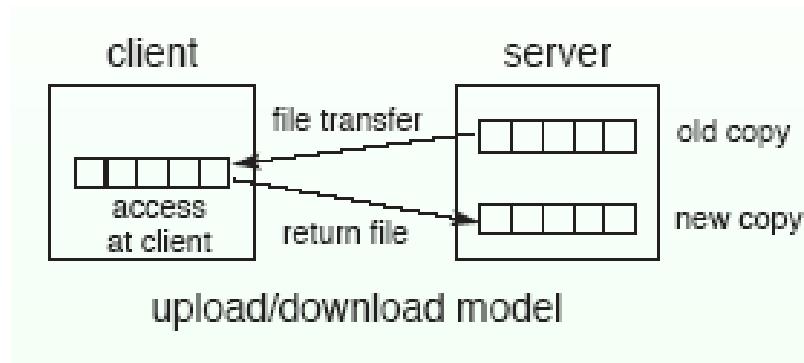
- Heterogenost

- Pristup udaljenim fajlovima je moguć bez obzira na razlike u hardveru i OS.

# Modeli pristupa udaljenom fajlu

## \* Upload/download model

- Jedini servisi koji su obezbeđeni su read i write.
  - read preuzima (download) fjal sa servera i pamti ga na klijent mašini; obavlja operacije nad fajlom lokalno; kada završi vraća (upload) fajl na server.
  - Primer Internetov FTP servis
- Jednostavan model
- performanse su bolje jer se sve obavlja lokalno, nema mrežnog saobraćaja
- Problemi:
  - Šta ako klijent nema dovoljno prostora da zapamti ceo fajl?
  - Šta ako je klijentu potreban samo mali deo fajla?
  - Šta ako i drugi klijenti žele da modifikuju fajl?



# modeli pristupa udaljenom fajlu (nast.)

## \* Model udaljenog pristupa

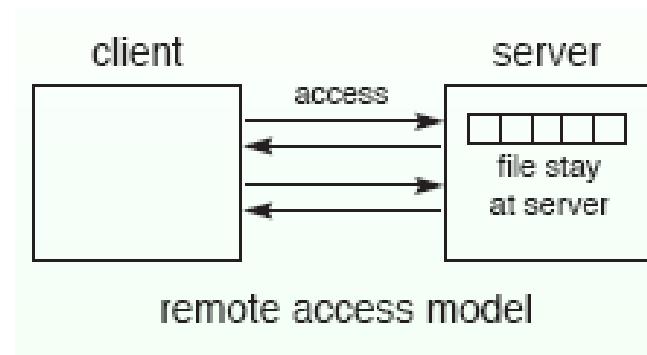
- Fajl servis obezbeđuje izvršenje operacija na udaljenoj mašini (open, close, read, write, read byte, write byte,...); operacije se izvršavaju na serveru (fajl se ne pomera sa servera)
  - npr. klijent može zahtevati da se otvori određeni fajl, a zatim da se pročita određeni blok fajla i samo taj blok će biti prenet klijentu
  - sve se realizuje preko RPC mehanizma

## \* Prednost

- lakše je implementirati deljenje fajlova
  - ako neki klijent modifikuje fajl ta modifikacija je odmah vidljiva svim klijentima

## \* Nedostaci

- Moguće zagušenje mreže i servera
  - Pristup serveru traje za vreme celog pristupa fajlu umesto samo za vreme upload/download.
- Lošije performanse
  - čitanje sa udaljenog diska je sporije nego čitanje sa lokalnog



# Statefull i stateless server

## \* Stateless server

- server ne pamti koji klijenti pristupaju nekom fajlu, koji fajlovi su otvoreni
- nikakva informacija o klijentskim zahtevima se ne pamti na serveru
- sve informacije potrebne da bi se obslužio zahtev mora da pamti i obezbedi klijent pri svakom zahtevu
- otporniji je na otkaz servera jer se nikakve informacije ne pamte na serveru
  - ako server otkaže i ponovo se podigne nikakve informacije neće biti izgubljene
- ako neki klijent modifikuje fajl, server nema načina da obavesti ostale klijente da je fajl modifikovan
  - klijent mora da vodi računa o konzistenciji, jer server nema nikakve informacije

## \* Stateful server

- server pamti informacije o tome koji klijent je otvorio koji fajl
- kraće su poruke u kojima klijent šalje zahteve serveru
- Bolje performanse
- lakše je ostvariti konzistenciju
  - ako jedan klijent modifikuje fajl, server zna koji klijenti pristupaju tom fajlu pa može preduzeti akcije da obavesti druge klijente da je fajl promenjen
  - moguće primeniti fajl locking (zaključavanje)
- teže postići tolerantnost na otkaze
  - ako server otkaže, pa se ponovo podigne informacije o tome koji klijenti pristupaju kom fajlu će biti izgubljene
  - Mora se periodično pamtitи stanje servera da bi se izvršio oporavak od greške
    - Beleženje stanja može biti dosta zahtevno sa stanovišta performansi

# Projektantska pitanje

- \* Imenovanje: lociranje fajla/direktorijuma u DFS na osnovu imena.
- \* Semantika deljenja fajlova
- \* Lokacija keša: disk, memorija, oba.
  - Politika upisa: ažuriranje izvora kada se keš modifikuje.

# Imenovanje fajlova

## \* Pitanja

- – kako su fajlovi imenovani?
- – Da li ime fajla govori o njegovoj lokaciji u DS?
- – Da li se ime fajla menja ako fajl promeni lokaciju?
- – Da li se ime fajla menja ako korisnik (klijent) promeni lokaciju?

## \* Lokaciona transparentnost:

- ime fajla ne otkriva fizičku lokaciju fajla.

## \* transparentnost migracije (lokaciona nezavisnost ):

- Ime fajla ne mora da se promeni ako se promeni njegova lokacija.

## \* Većina šema za imenovanje fajlova nema migracionu transparentnost, ali može imati lokacionu transparentnost.

# Tri načina imenovanja

\* Imenoavlje fajlova kombinacijom imena hosta i lokalnog imena fajla. (ime\_hosta:lokalno\_ime)

- Garantuje jedinstvenost imena.

- Npr. Windows Network Places
- URL kod http
  - [http://match.pmf.kg.ac.rs/electronic\\_versions/Match81/n1/match81n1\\_163-174.pdf](http://match.pmf.kg.ac.rs/electronic_versions/Match81/n1/match81n1_163-174.pdf)

- Prednosti:

- Pronalaženje fajla je jednostavno

- Nedostaci:

- Korisnik mora znati kompletno ime i svestan je koji su fajlovi lokalni a koji udaljeni.
- Ime fajla je lokaciono zavisno i fajl se ne može pomerati.
- Nema transparentnosti. Pomeranje fajla na drugi host, zahteva promenu imena.

\* Mounting: montiranje udaljenog direktorijuma na lokalni direktorijum. Lokaciona transparentnost se postiže nakon montiranja. (koristi se u Sun NFS).

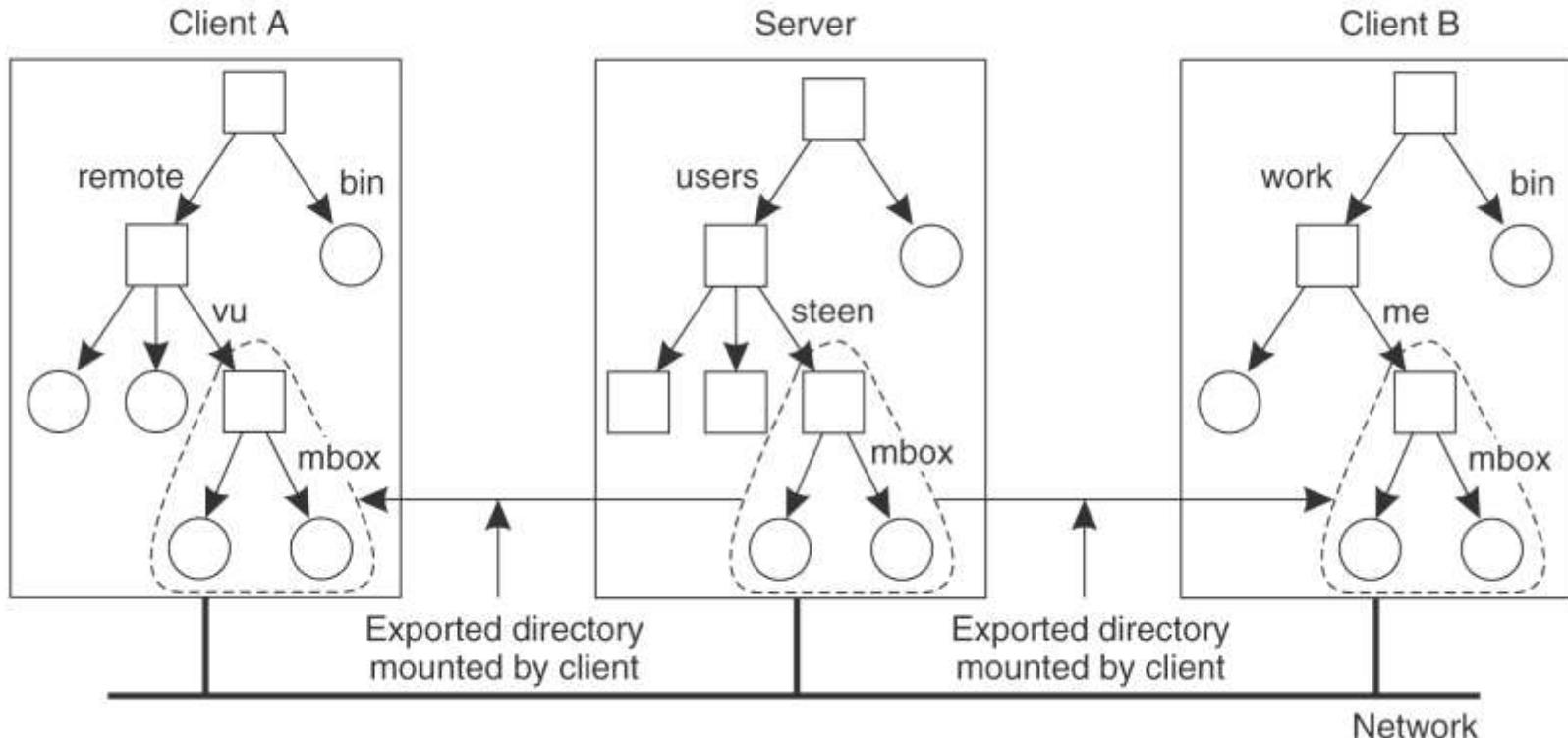
- Montiranim direktorijumima se pristupa kao lokalnim.

- npr. mapirani drajv u Windowsu

# Mounting

- \* Ubacivanje (mounting) udaljenog direktorijuma na lokalni direktorijum. (NFS – Sun-ov Network File System)
- \* Svaki host ima skup lokalnih imena za obraćanje udaljenim lokacijama
  - Ova imena se zovu tačke montiranja jer su udaljeni fajlovi ili direktorijumi povezani i prikačeni na lokalni fajl sistem.
- \* Svaki host ima tabelu (/etc/fstab) koja definiše preslikavanje
  - <remote path name@ machine name> u <local path name>.
- \* Kada se sistem uključi, ostvaruje se povezivanje sa udaljenim serverom.
- \* Nakon toga korisnik pristupa udaljenom fajlu korišćenjem lokalnog imena, a DFS vodi računa o preslikavanju
- \* • Prednost: lokaciona transparentnost – kada se ostvari povezivanje sa udaljenim serverom klijent ne mora da vodi računa da li je fajl kome pristupa lokalni ili udaljeni
- \* Udaljeni server može da se promeni a da to ne utiče na imenovanje fajla
- \* • Nedostatak: isti fajl može imati više imena u zavisnosti od klijenta

# Mountovanje udaljenih direktorijuma na lokalni fajl sistem



\* Imena nisu globalno jedinstvena

- Isti fajl se različito vidi

- Server vidi fajl mbox : */users/steen/mbox*
- Client A vidi fajl mbox : */remote/vu/mbox*
- Client B vidi fajl mbox : */work/me/mbox*

# Tri načina imenovanja (nast.)

\* **Totalna integracija komponenti fajl sistema - svi fajlovi u sistemu pripadaju jedinstvenom prostoru imena.**

- Imena su jedinstvena u celom sistemu
- Ime fajla je isto bez obzira sa kog čvora se pristupa fajlu
  - Ovo može da funkcioniše samo između sistema koji čvrsto sarađuju. (Berkeley Sprite, CMU Andrew)

# Semantika deljenja fajlova

## \* Fajl sistem deli mnogo korisnika

- Šta se dešava ako dva procesa pristupaju istom fajlu?

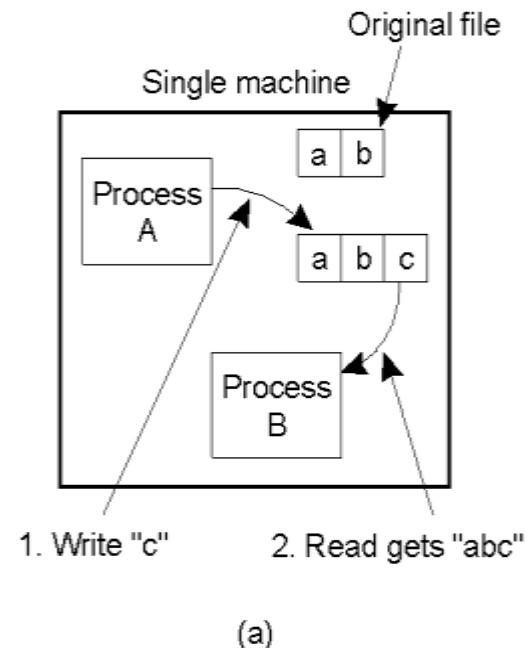
## \* Sekvencijalna semantika (unix semantika)

- Apsolutno vremensko uređenje svih operacija.
- Svaka read operacija nad fajлом vidi efekte prethodnih write operacija.
- Sekvencijalnu semantiku u DFS je lako ostvariti ako svakim fajлом upravlja jedan server i klijenti ne keširaju fajlove.
  - Tada redosled read i write pristupa određuje server.
- Ako se ne koristi keširanje na strani klijenta, javlja se problem sa performansama – klijenti će se obraćati serveru za svaku operaciju na fajlu (čak i čitanje jednog bajta)
- Problem performansi se može ublažiti ako se dozvoli keširanje na strani klijenta.
  - Javlja se problem konzistencije podataka: Ako klijent modifikuje podatak u kešu, a drugi klijent pročita podatke sa servera, dobiće ustajale podatke (sekvencijalna semantika više ne važi!)
  - Rešenje: koristiti write-through politiku upisa.
    - Ne rešava problem ustajalih kopija u keševima drugih klijenata – server mora da obavesti ostale klijente da imaju ustajale podatke.
  - Drugo rešenje – ubalažiti semantiku

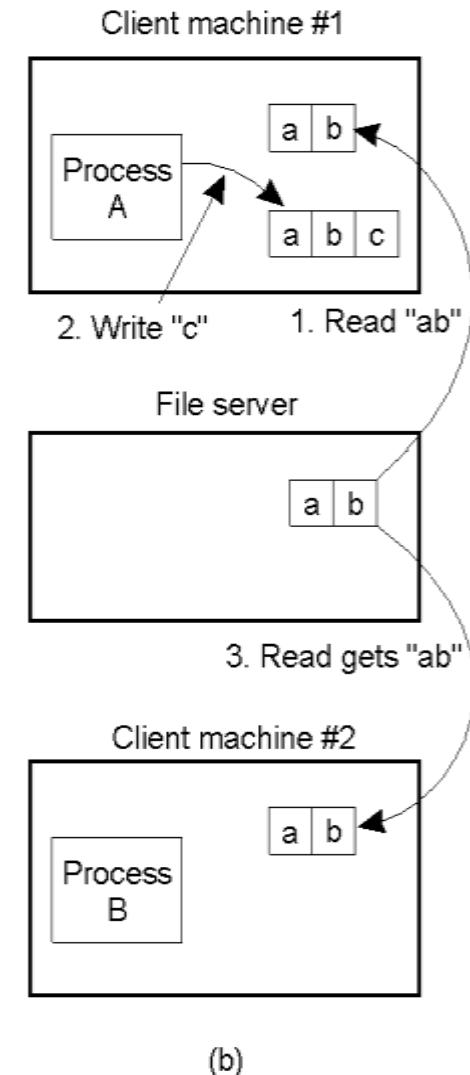
# Semantika deljenja fajlova

## \* Semantika sesije

- Sesija – serija pristupa fajlu između otvaranja i zatvaranja fja.
- Promene učinjene na fajlu su vidljive samo klijent procesu (moguće i drugim procesima na istoj mašini)
- Drugim procesima su promene vidljive nakon zatvaranja i ponovnog otvaranja fajla.



a) Sekvencijalna semantika



b) semantika sesije

# Keširanje

## \* Keširanje u memoriji servera

- Jednostavno i transparentno za korisnika
- Sve akcije nad fajlovima obavlja server
- Mrežni sobraćaj nije smanjen
- Server je potencijalno usko grlo

## \* Keširanje fajlova na strani klijenta

- Na klijentovom disku
  - Dovoljno prostora za pamćenje velikih fajlova, pouzdano
  - Brži pristup fajlu nego kroz mrežu
  - Server se samo povremeno kontaktira (rasterećen je, sistem je skalabilan)
- U klijentovom kešu
  - Uobičajeno rešenje
  - Brzi pristup podacima iz fajla
  - Nepouzdano – u slučaju otkaza klijenta podaci mogu biti izgubljeni
- Sa keširanjem na strani klijenta javlja se problem keš konzistencije!

# Politike upisa u keš

\* Kada modifikovani sadržaj keša treba da se prenese na server?

- *Write-through politika*

- Zajedno sa modifikacijom keša modificuje se i fajl na serveru.
- Prednost: pouzdanost, otkaz klijenta ne znači gubitak podataka.
- Nedostatak : upisi na server se obavljaju svaki put kada se modificuje keš.
- Prednost keširanja se koristi samo u slučaju read operacije.

- Politika zakašnjenog upisa

- Upis na server se obavlja nakon određenog vremena (npr. na svakih 30sec)
- Prednost: male i česte promene ne povećavaju mrežni saobraćaj.
- Nedostatak : manje pouzdano, osjetljivo na otkaz klijenta

- Upis u trenutku zatavaranja fajla.

- Redukuje se broj upisa na disk
- Nepouzdano – ako klijent otkaže podaci će biti izgubljeni

# Provera konzistentnosti podataka

\* Da li su podaci u lokalnom kešu konzistentni sa master kopijom (serverskom kopijom)?

- Provera inicirana od strane klijenta

- Klijent kontaktira server da bi utvrdio da li je njegova kopija konzistentna sa serverskom kopijom
- Provera se obavlja kod svakog pristupa fajlu ili samo prilikom otvaranja fajla
- Česta provera validnosti podataka u kešu može da anulira prednosti keširanja

- *Server-inicirana provera*

- Server beleži prisustvo fajlova u keševima klijenata
- Kada server detektuje potencijalnu nekonzistentnost, reaguje.

# HDFS-distribuirani fajl sistem

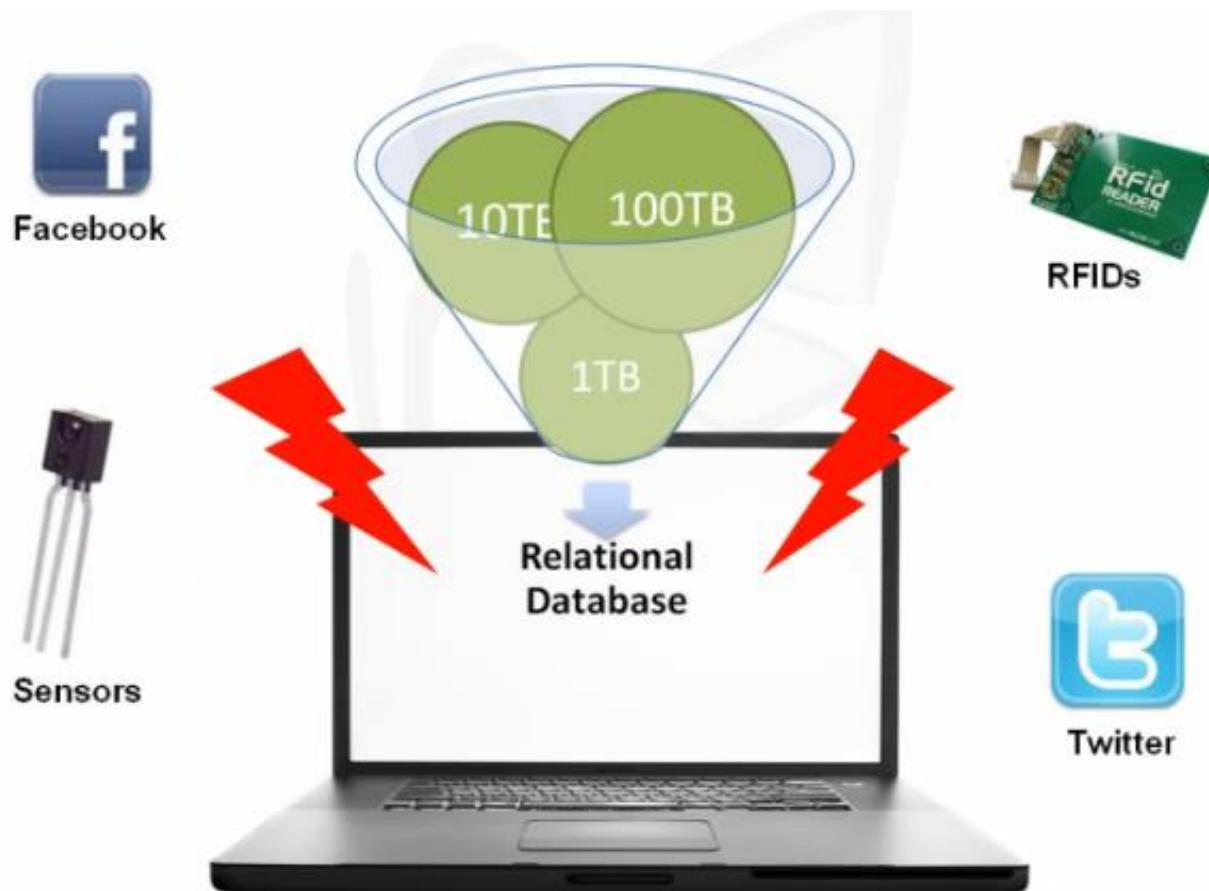
# Motivacija

- Pretpostavimo da imamo 1GB podataka koje treba obraditi
  - Podaci su smešteni u relacionoj BP na desktop računaru i računar nema problem da upravlja ovim podacima
- Vaša kompanija počne da raste i podaci koje treba obraditi dostignu 10 GB, a onda 100GB
  - Računar koji obrađuje podatke dostiže svoje granice

# Problemi

- Kupovina novog računara rešava problem samo nekoliko meseci, ako podaci počnu da rastu na 10 TB, pa na 100 TB
- Šta više, postoje zahtevi da vaša aplikacija koristi nestruktурне podatke koji potiču iz izvora kao što su Facebook, Twitter, RFID čitači, senzori i sl.
- Vaš menadžment ima zahtev za informacijama i iz relacione baze podaka i iz nestrukturnih podataka i to što pre.

# Zahtevi



# Motivacija

- Pretpostavimo da je moguće imati disk kapaciteta 500TB.
- Ovo je dovoljno za skladištenje više od 20 milijardi web strana (neka je prosečna veličina strane 20KB)
- U slučaju da samo želite da pregledate tih 500TB biće vam potrebno više od 5 meseci ako disk ima brzinu prenosa podataka sa diska u memoriju od 40 MB/sec.
- Zamislite vreme da potrebno obradu tih podataka!
- Potrebno je drugo rešenje!

# Motivacija

- Potreba za više CPU-a: Da bi se programi izvršavali brže
- Potreba za više diskova: Savremene aplikacije zahtevaju rad sa velikom količinom podataka, a sa više diskova I/O se može izvoditi u paraleli

# Eksplozija podataka

- Od 2012. godine svakog dana se kreira oko 2.5 eksabajta ( $2.5 \times 10^{18}$ ) podataka.
- Do 2020. godine, očekuje se da bude proizvedeno 40 zetabajta ( $40 \times 10^{21}$ ) podataka, 300 puta više od količine koja je postojala pre samo 9 godina.
- Odakle dolazi ovoliko podataka?

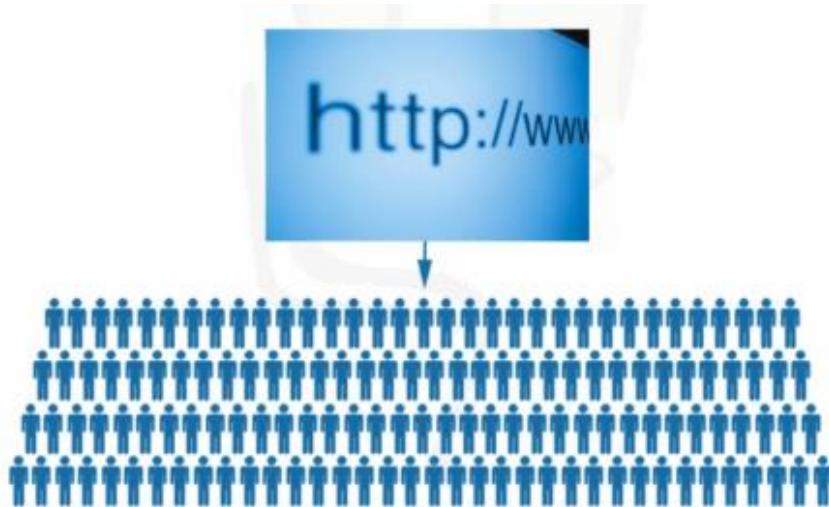
# Eksplozija podataka

- Skupovi podataka se prikupljaju sa jeftinih i mnogobrojnih uređaja koji poseduju određene senzore kao što su: RFID čitači, kamere, mikrofoni, senzori bežičnih mreža i sl. stoga smo svedoci eksplozije prikupljenih podataka širom sveta
- Ovi podaci mogu biti nestrukturni i da rastu tako brzo da je teško upravljati njima korišćenjem baza podataka



# Ljudi kao generatori podataka

- 2 milijarde Internet korisnika



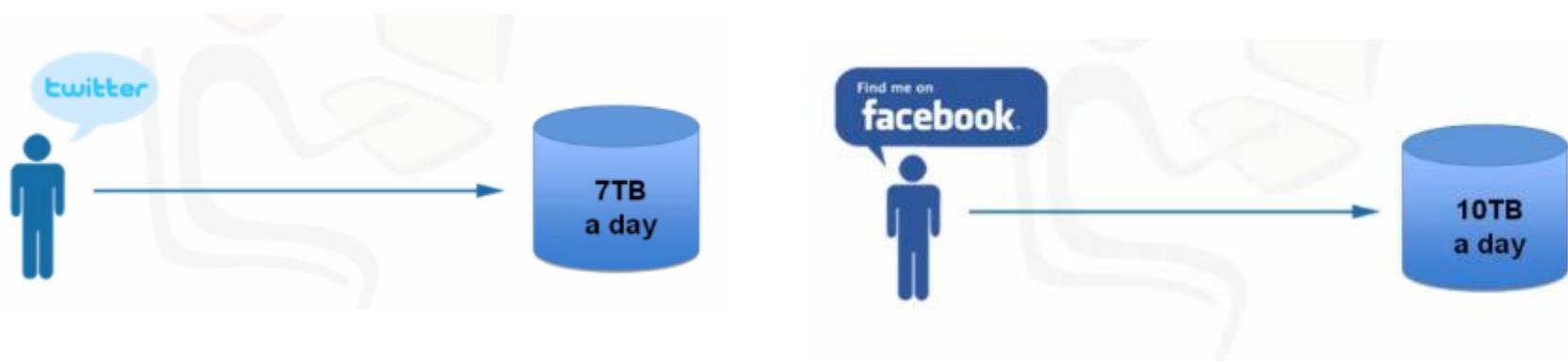
# Ljudi kao generatori podataka

- Osnovni razlog se može naći u činjenici da 6 od 7 miliardi ljudi na svetu poseduje mobilni telefon.
- Kako infrastruktura postaje dostupnija i jeftinija, upotreba mobilnih telefona (samim tim i količina podataka koja se generiše) raste eksponencijalno.

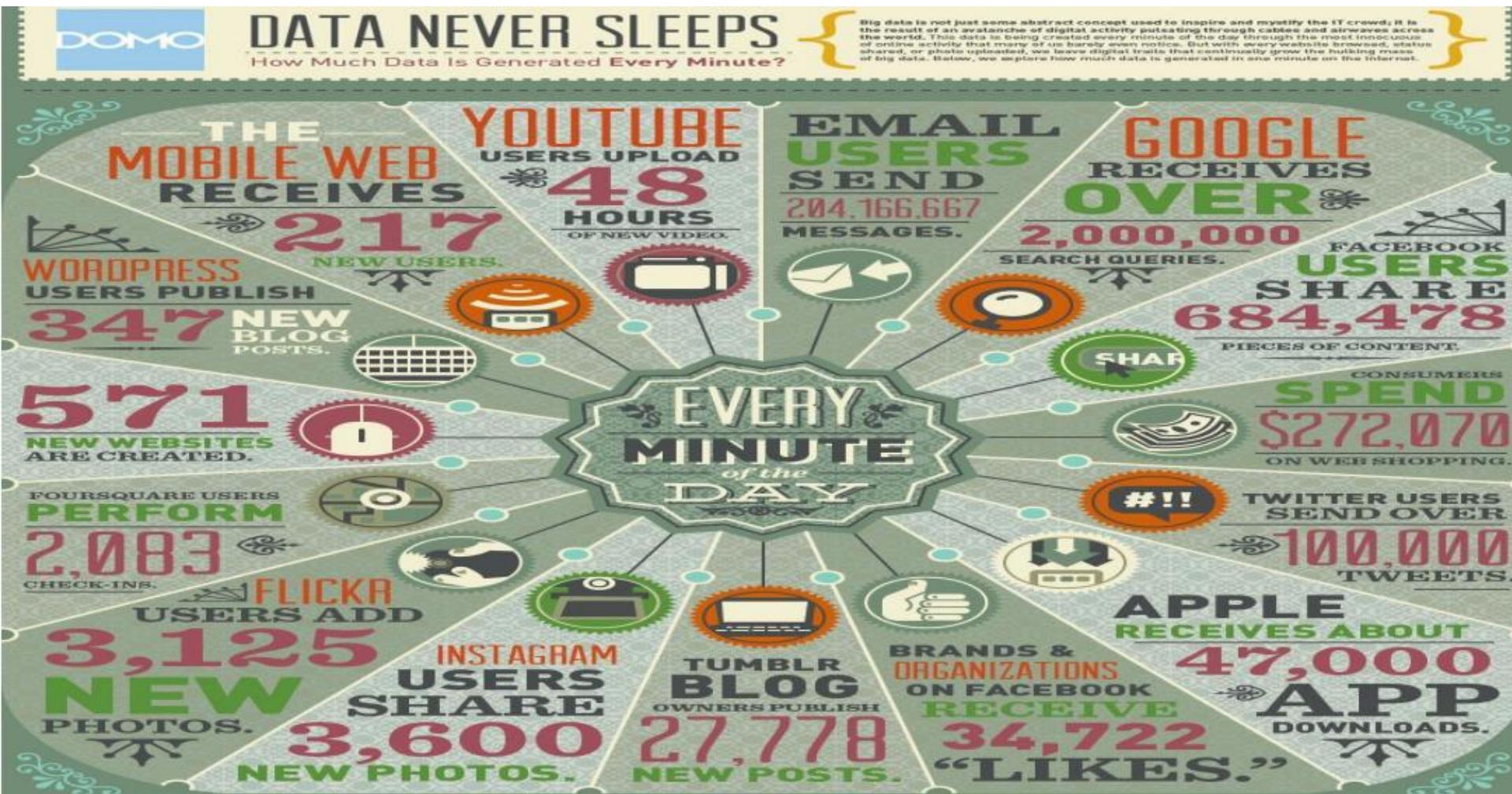


# Ljudi kao generatori podataka

- 7 TB podataka generisano na Twitteru svakog dana, 10 TB podataka generisano na FaceBooku svakog dana



# Ljudi kao generatori podataka



# Ljudi kao generatori podataka

- Zamislite sve podatke koje lično generišete svaki put kada izvršite plaćanje karticom, kada objavite nešto na socijalnim mrežama, vozite, posetite lekara i slično.
- Sada zamislite sve te podatke zajedno sa podacima svih ljudi, korporacija i organizacija na svetu.
- Preko zdravstva do socijalnih mreža, od biznisa do auto industrije, ljudi generišu više podataka nego ikada ranije.

# Ostali generatori podataka

- Veliki Hadron Collider akcelerator u Švajcarskoj proizvede 15 PB podataka za 1 godinu
- Avion Boing 737 generiše oko 240 TB podataka tokom jednog leta sa jednog na drugi kraj Sjedinjenih Američkih Država

# U skoroj budućnosti

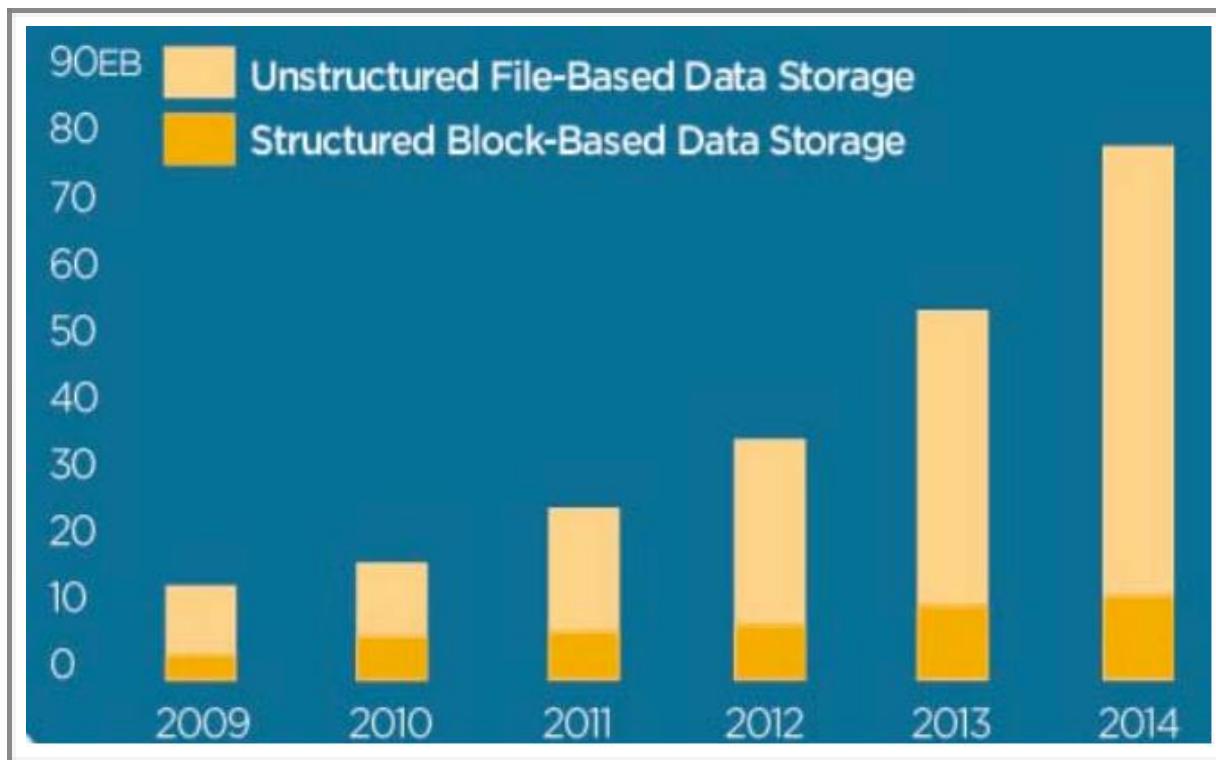
- “**IBM Research** and Dutch astronomy agency **Astron** work on new technology to handle **one exabyte of raw data per day**

that will be gathered by the world largest radio telescope, the **Square Kilometer Array**, when activated in **2024**”.

# Raznolikost podataka

- Podaci su veoma retko u obliku koji je pogodan za obradu. Čest slučaj u *Big Data* sistemima je da su podaci raznoliki, i ne mogu se skladištiti u relacionim bazama podataka.
- Podatke mogu predstavljati tekstovi sa socijalnih mreža, slike, ili *raw(sirovi)* podaci sa senzora.
- Nijedna od navedenih stavki nije u pogodnom obliku za integraciju sa aplikacijom, pa skladištenje podataka prevaziđa tradicionalni pristup gde su podaci smešteni u određeni okvir koji propagira relaciona šema i gde se unapred znaju apsolutno sve veze između podataka.

# Raznolikost podataka



# Zaključak

- Velika količina podataka je opšti naziv za skupove podataka toliko velike ili složene da se za rukovanje njima ne mogu koristiti tradicionalne aplikacije.
- Izazovi koji se javljaju obuhvataju prikupljanje, skladištenje, pretragu, analizu podataka, prenos podataka ili njihovu vizualizaciju.
- Analize velikih skupova podataka mogu doprineti određivanju trendova u poslovanju, predviđanju i sprečavanju katastrofa, borbi protiv kriminala i slično

# Rešenje

- Šta je rešenje?

- Grace Hopper (poznati naučnik u oblasti CS): “In pioneer days they used oxen for heavy pulling, and when one ox couldn’t budge a log, they didn’t try to grow a larger ox. We shouldn’t be trying for bigger computers, but for more systems of computers”.

**Scale-Up:** put more resources into the system  
to make it bigger and more powerful

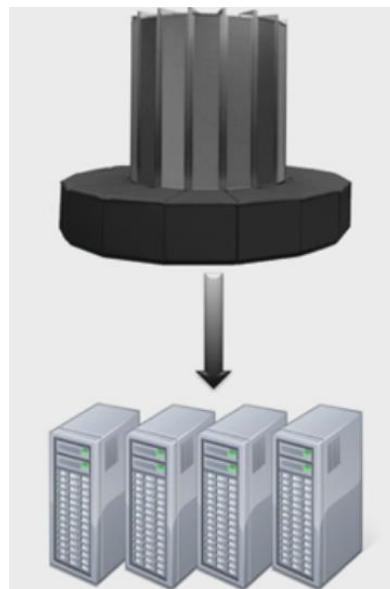


**Scale-Out:** connect a large number of “ordinary”  
machines and create a cluster

Scale-Out is **more powerful** than Scale-Up, and also  
**less expensive**

# Rešenje

- Šta je rešenje?
  - Umesto jednog velikog i moćnog računara probati korišćenje sistema koji se sastoji iz više računara (Distribuirani sistemi)



# Rešenje-izazovi, MapReduce

- Ovde takođe postoje izazovi:
  - Visok nivo programerske kompleksnosti:  
organizacija procesa i njihova sinhronizacija
  - Organizacija slanja velike količine podataka sa jednog računara na ostale računare
  - Greške u prisustvu velikog broja računara su neminovnost na koju se mora računati u DS i koja značajno utiče na ispravnost izvršenja aplikacije
- Za sve navedene probleme rešenje se može naći u Hadoop **MapReduce!**

# Nastanak Hadoopa

- Eksponencijalni rast podataka je na početku predstavljao izazov velikim kompanijama kao što su Google, Yahoo, Amazon i Microsoft.
- Ove kompanije imaju potrebu da prolaze kroz terabajte i petabajte podataka kako bi saznali koji web sajtovi su popularni, koje knjige se traže na internetu, kao i koje vrste reklama privlače pažnju ljudima.
- Postojeći alati su postali neadekvatni da bi se obrađivali skupovi podataka ovog obima.
- Google je bio prvi koji je publikovao MapReduce – sistem koji oni koriste kako bi obradili podatke tog obima.

# Nastanak Hadoopa

- 2003. Google objavljuje svoju klaster arhitekturu i svoj distribuirani fajl sistem (GFS)
- 2004. Google objavljuje svoj MapReduce programski model kao nadogradnju GFS
  - I GFS i MapReduce su pisani na C++ i zatvorenog su koda; sa Python i Java API-jem dostupnim samo Google programerima
- 2006. Apache & Yahoo objavljuju Hadoop i HDFS
  - Open-sorce (otvorenog koda), Java implementacija GFS i GoogleMR sa skupom API dostupnih svima

# Nastanak Hadoopa

- Januara 2008. godine, Hadoop na velika vrata ulazi na tržište kao vodeći projekat kompanije Apache i biva korišćen od strane drugih velikih kompanija u razvoju kao što su Last.fm, Facebook i giganta u svetu novinskih agencija kao što je New York Times.
- Aprila 2008. godine, sada već visoko razvijen Hadoop sistem obara tadašnji svetski rekord i postaje najbrži sistem u istoriji za sortiranje terabajta podataka. Sastavljen od 910 čvorova koji čine klaster, Hadoop uspešno obavlja sortiranje terabajta podataka za samo **209** sekundi.
- Novembra iste godine, Google objavljuje da njihov MapReduce sistem istu količinu podataka sortira za neverovatnih **68** sekundi.
- Nadmetanje se nastavlja, pa tako Hadoop ubrzo odgovara izazovu i ovo vreme smanjuje na **62** sekunde.

# Nastanak Hadoopa

- Od tada Hadoop sistem za pretraživanje i analiziranje ogromne količine podataka za kratko vreme kreće da sarađuje sa vodećim IT kompanijama poput IBM, Microsoft, Oracle i nastavlja da se širi u okviru svojih čerki kompanija: Cloudera, Hortonworks i MapR.
- Danas se Hadoop koristi kao platforma opšte namene za skladištenje i analizu velike količine podataka
  - Istraživanje i razvoj se aktivno nastavljaju...

# Hadoop

- Hadoop je open-source Apache Foundation projekat koji implementira MapReduce paradigmu
- To je programski okvir napisan u Java programskom jeziku
- Razijen od strane Douga Cuttinga i dobio ime po imenu igračke njegovog sina
- Hadoop koristi Google MapReduce i Google File System tehnologije kao osnovu

# Daug Cutting & Hadoop



# Hadoop-osobine

- Hadoop je optimizovan da:
  - rukuje velikom količinom podataka koji mogu biti strukturni i nestruktturni
  - uz korišćenje široko dostupnih i relativno jeftinih računara (commodity hardware)
  - sa dobitkom u performansama
  - i ugrađenom visokom pouzdanošću
- Hadoop replicira svoje podatke na više računara tako što, ako jedan računar otkaže, replicirani podaci će biti obrađeni na nekom drugom računaru

# Hadoop-osobine

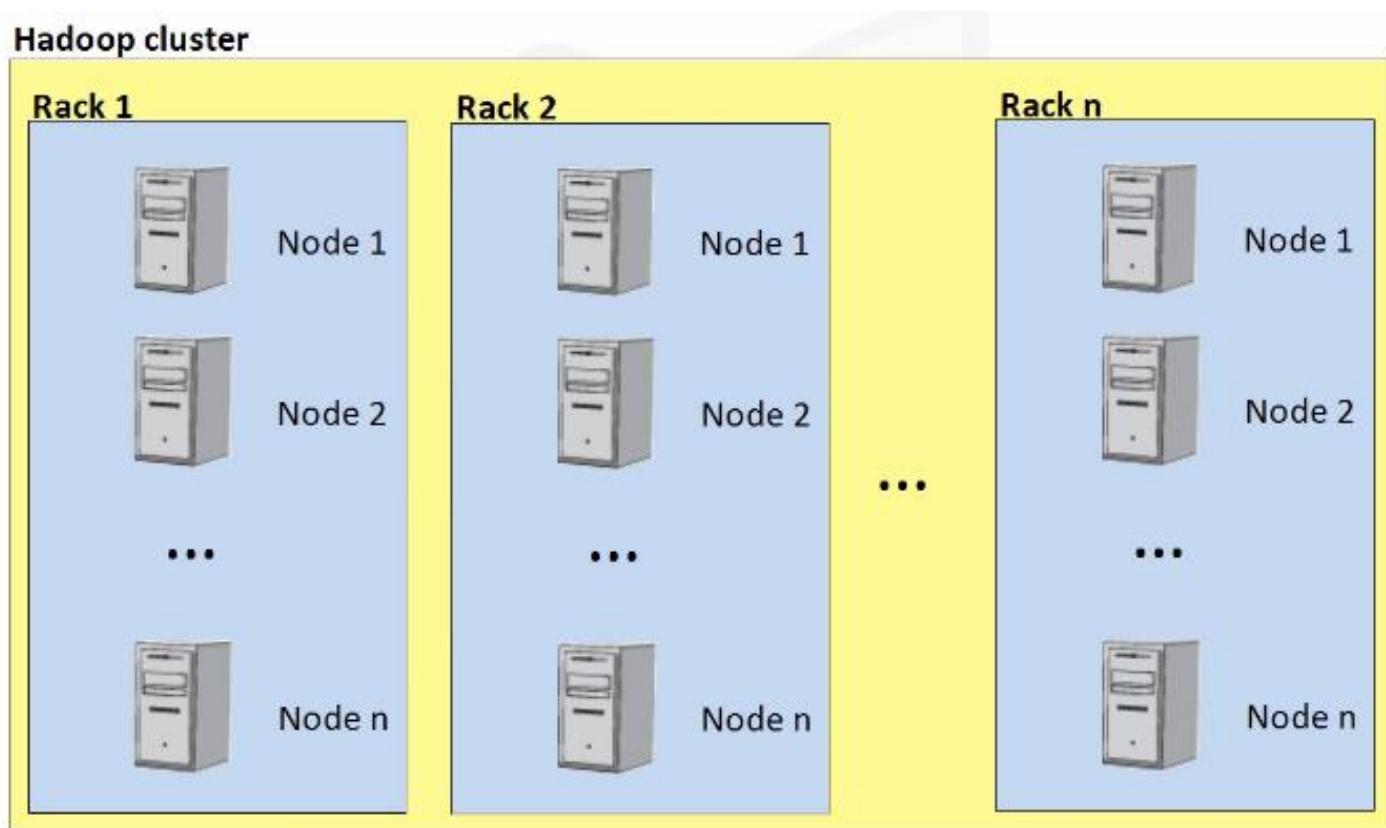
- *Dostupnost* – Hadoop radi na velikoj grupi računara ili u oblaku.
- *Otpornost* – Pošto je Hadoop predviđen da radi na velikom broju računara, on je dizajniran sa pretpostavkama da će često dolaziti do otkaza hardvera i veoma uspešno prevazilazi te probleme.
- *Skalabilnost* – Skaliranje Hadoop klastera se postiže jednostavnim dodavanjem čvorova u klaster.
- *Jednostavnost* – Hadoop omogućava programerima jednostavno pisanje paralelnog koda.

# Arhitektura

- Postoje dve glavne komponente Hadoop-a:
  - Distributed File System (DFS) komponenta
    - U slučaju Hadoop ima naziv HDFS i služi za skladištenje podataka na klasteru
  - Map-Reduce komponenta (MR) komponenta
    - Programski okvir za izvođenje izračunavanja nad podacima u HDFS
- Node (Čvor) je računar koji predstavlja commodity (standardni, uobičajen) hw koji sadrži podatke
- Rack (Rek) je kolekcija od 30-40 čvorova koji su fizički blizu jedan drugoga i povezani istim mrežnim switchem.

# Klaster

- Hadoop klaster je kolekcija rekova



# Distribuirani fajl sistem

- HDFS funkcioniše nad postojećim fajl sistemom na svakom čvoru Hadoop klastera
- Hadoop koristi blokove da sačuva fajl ili delove fajla
- Blokovi su veliki po defaultu 64MB (novije verzije Hadoopa 128 MB)
- Hadoop blok je fajl na fajl sistemu koji se nalazi u osnovi



# HDFS blokovi

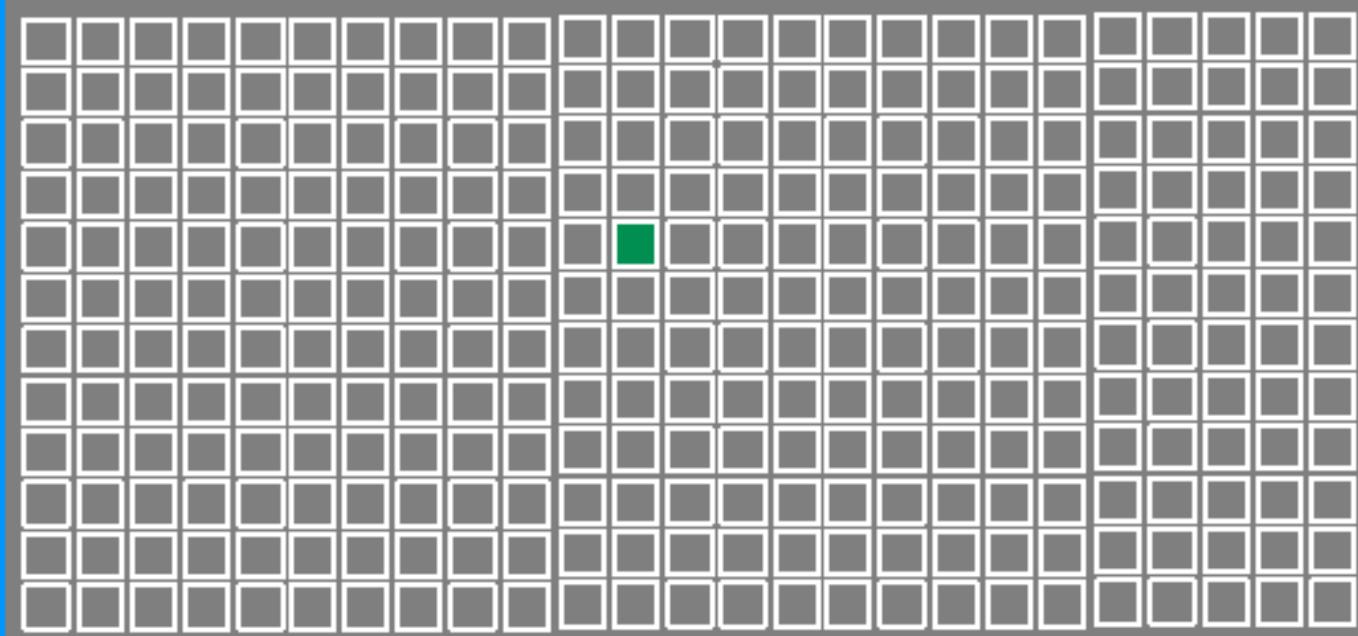


# HDFS blokovi

## Block IDs

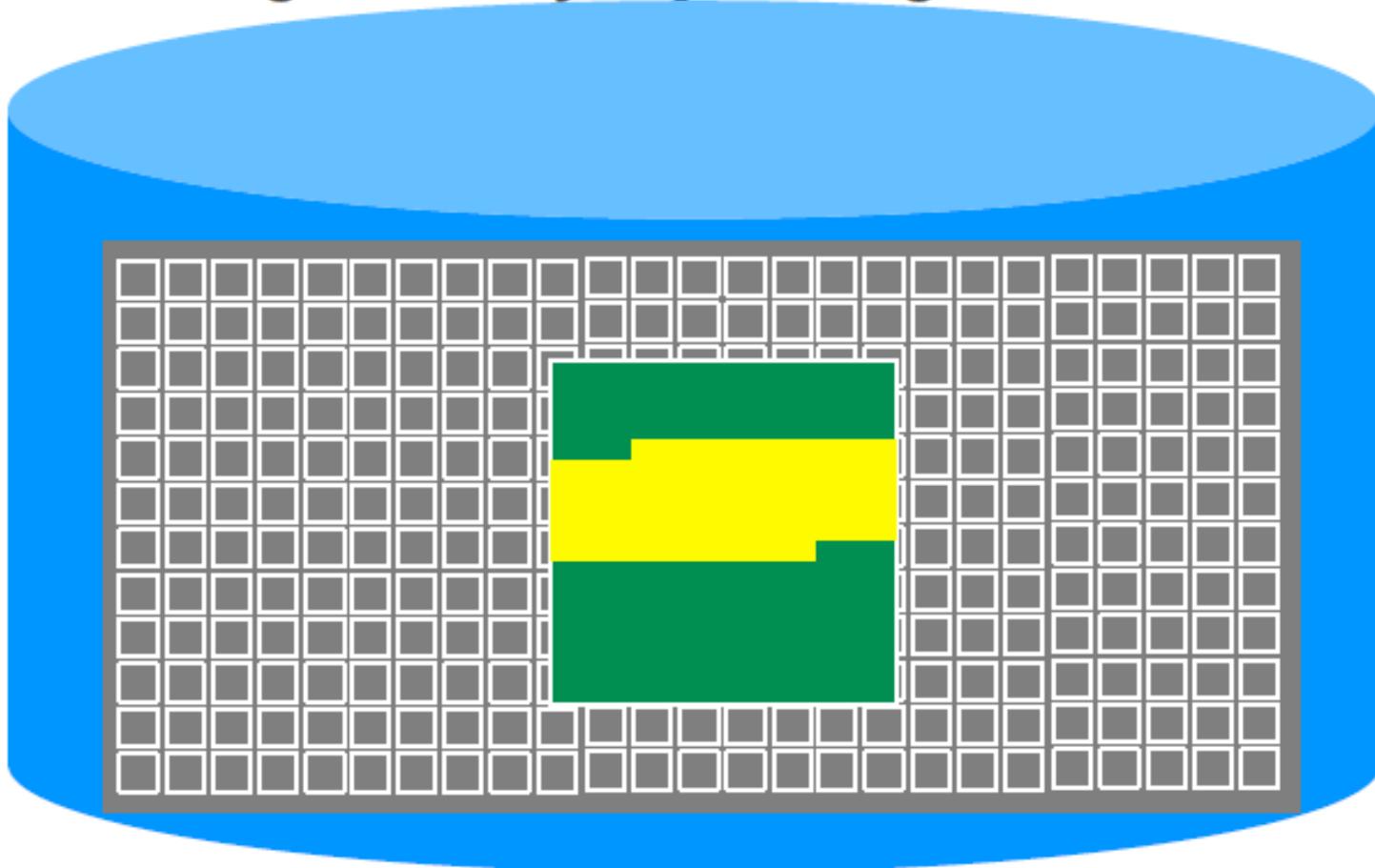
64 bits

e.g., 7586700455251598184



# HDFS blokovi

**Subblock granularity: Byte Range**

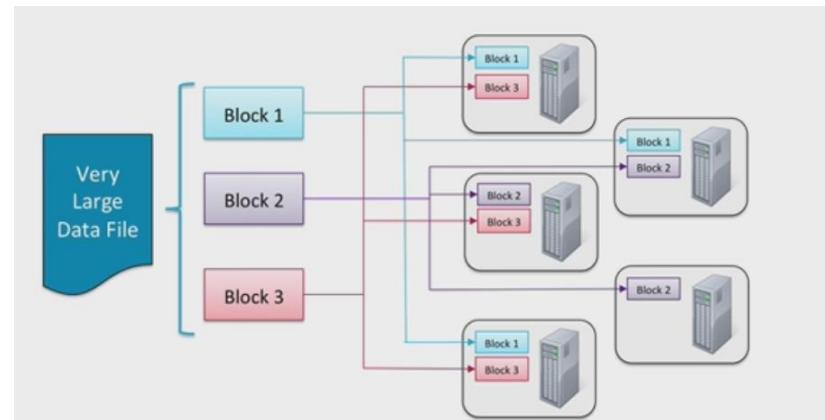


# Hadoop distribuirani fajl sistem

- Blokovi imaju sledeće prednosti:
  - Imaju fiksnu veličinu pa je lako izračunati koliko će stati na disk
  - Svojom raspodelom na više čvorova omogućavaju da fajl bude veći nego bilo koji pojedinačni disk u klasteru
  - Blokovi se repliciraju na više čvorova što omogućava HDFS da bude FT (Fault Tolerant)

# Hadoop distribuirani fajl sistem

- Ako čvor koji sadrži B1 otkaže, postoje još dva čvora koja sadrže taj blok i omogućavaju da se izračunavanje obavi bez problema
- Faktor replikacije se može promeniti u Hadoop konfiguraciji ili čak postaviti faktor za svaki individualni fajl

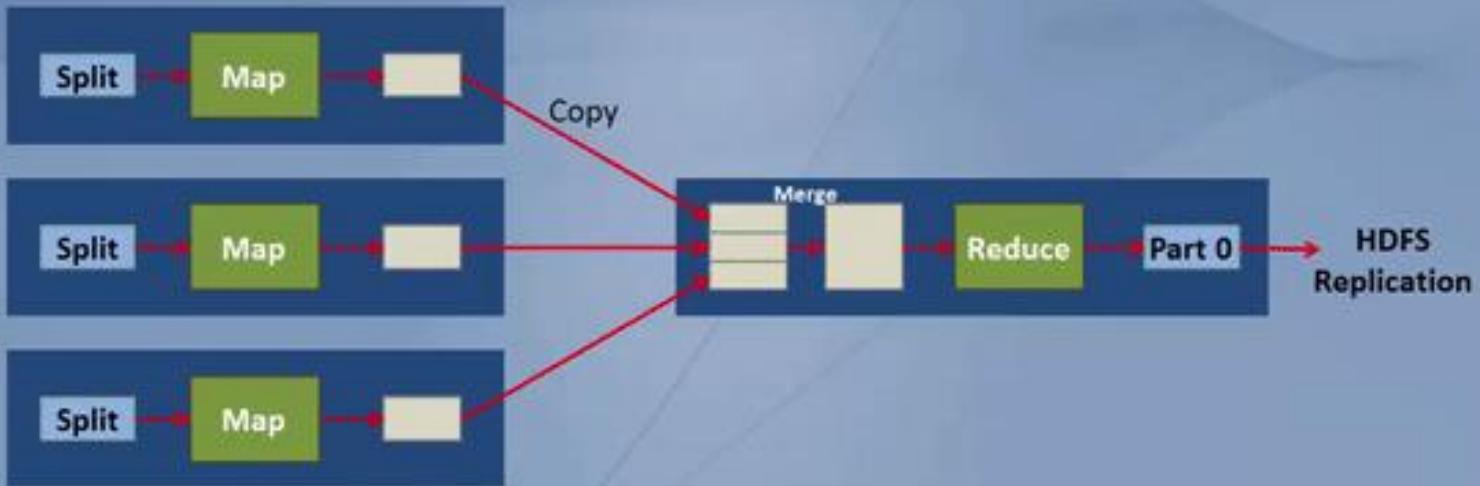


# MapReduce komponenta

- MapReduce program se sastoji iz dve vrste transformacija koje mogu biti primenjene više puta:
  - Map transformacija
  - Reduce transformacija
- MapReduce job izvršava MapReduce program koji je podeljen na Map taskove koji se izvršavaju paralelno i Reduce taskove koji se izvršavaju paralelno jedni sa drugima

# MapReduce komponenta

## MapReduce Data Flow



# Hadoop čvorovi

- Postoje tri vrste čvorova u klasteru u odnosu na uloge koje imaju: klijent čvor, *master* (gospodar) čvor i *slave* (sluga) čvor.
- Uloga Klijent čvora je da loaduje podatke u klaster, dostavlja MapReduce posao (job) koji opisuje kako treba obraditi podatke i dobije ili pogleda rezultate obavljenog posla po njegovom okončanju
- Na master čvoru se vrši nadgledanje i rukovodjenje dvema ključnim komponentama Hadoop-a: skladištenje velikog broja podataka (HDFS) i izvršavanje paralelnih izračunavanja nad svim tim podacima (MR).
- Slave čvorovi su čvorovi koji obavljaju skladištenje podataka i izvršavaju izračunavanja.

# Hadoop demoni

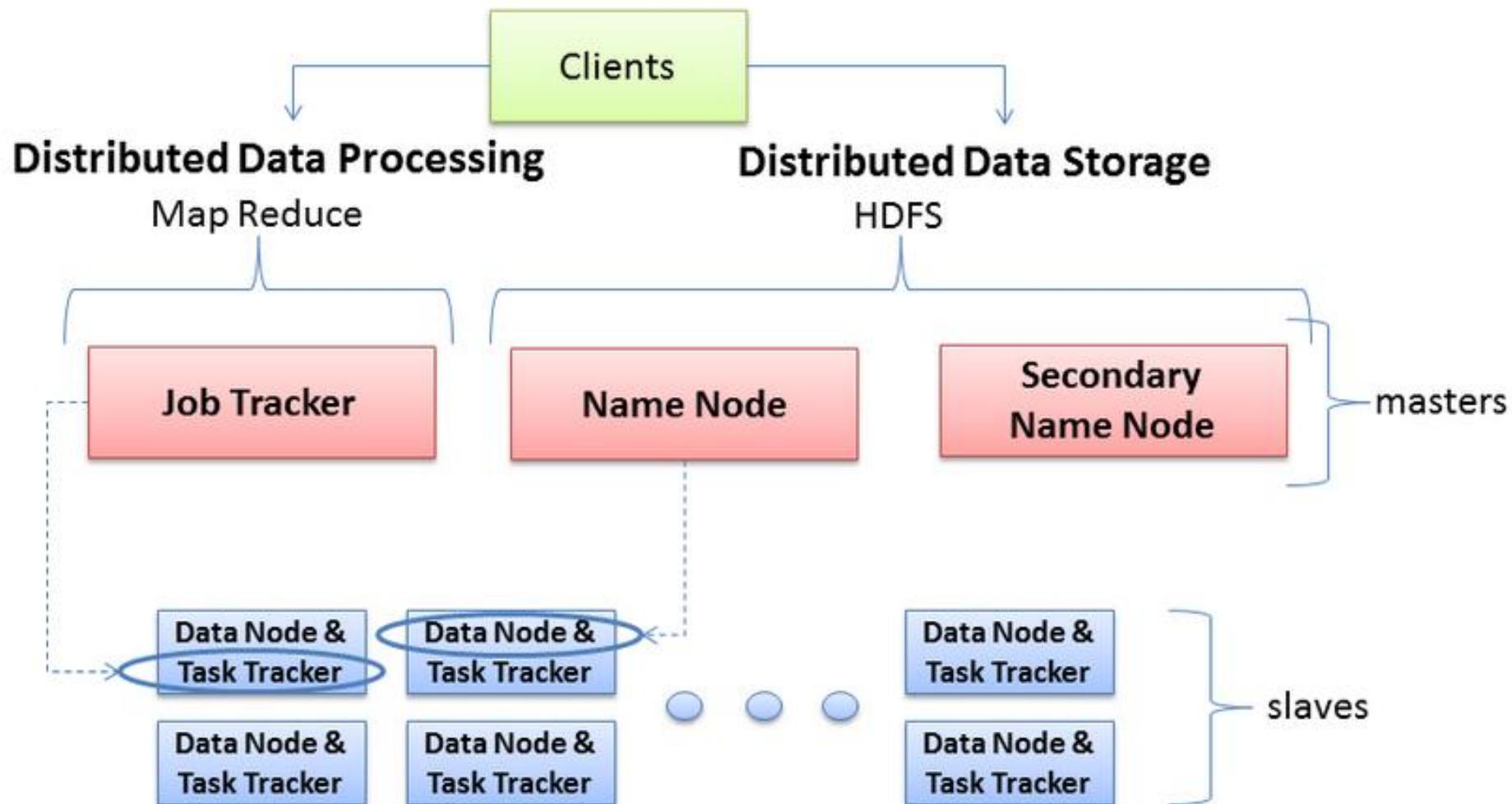
- Prilikom startovanja Hadoopa na čvorovima klastera se startuje skup demona (eng. deamon) koji ima specifične uloge.
- Postoje dve vrste demona Hadoop-a:
  - HDFS demoni
  - MapReduce demoni

# HDFS demoni

- U HDFS demone spadaju jedan NameNode (NN) demon i više DataNode (DN) demona
- Postoji samo jedan NameNode i više DataNode-ova u klasteru
- DataNode odgovoran za čuvanje podataka u blokovima, primanje naredbi od NN i davanje informacija NN-u.

# HDFS demoni

- Namenode deamon je demon koji se izvršava na glavnom ili master čvoru
- NameNode održava fajl sistem namespace (prostor imena) i svaka promena njegovih karakteristika je zapamćena od strane NameNode.
- NameNode čuva informacije o tome kako su fajlovi podeljeni na blokove, koji *slave* čvorovi čuvaju koje blokove i sam učestvuje u preslikavanju blokova u čvorove
- NameNode čuva informaciju i o vrednosti faktora replikacije



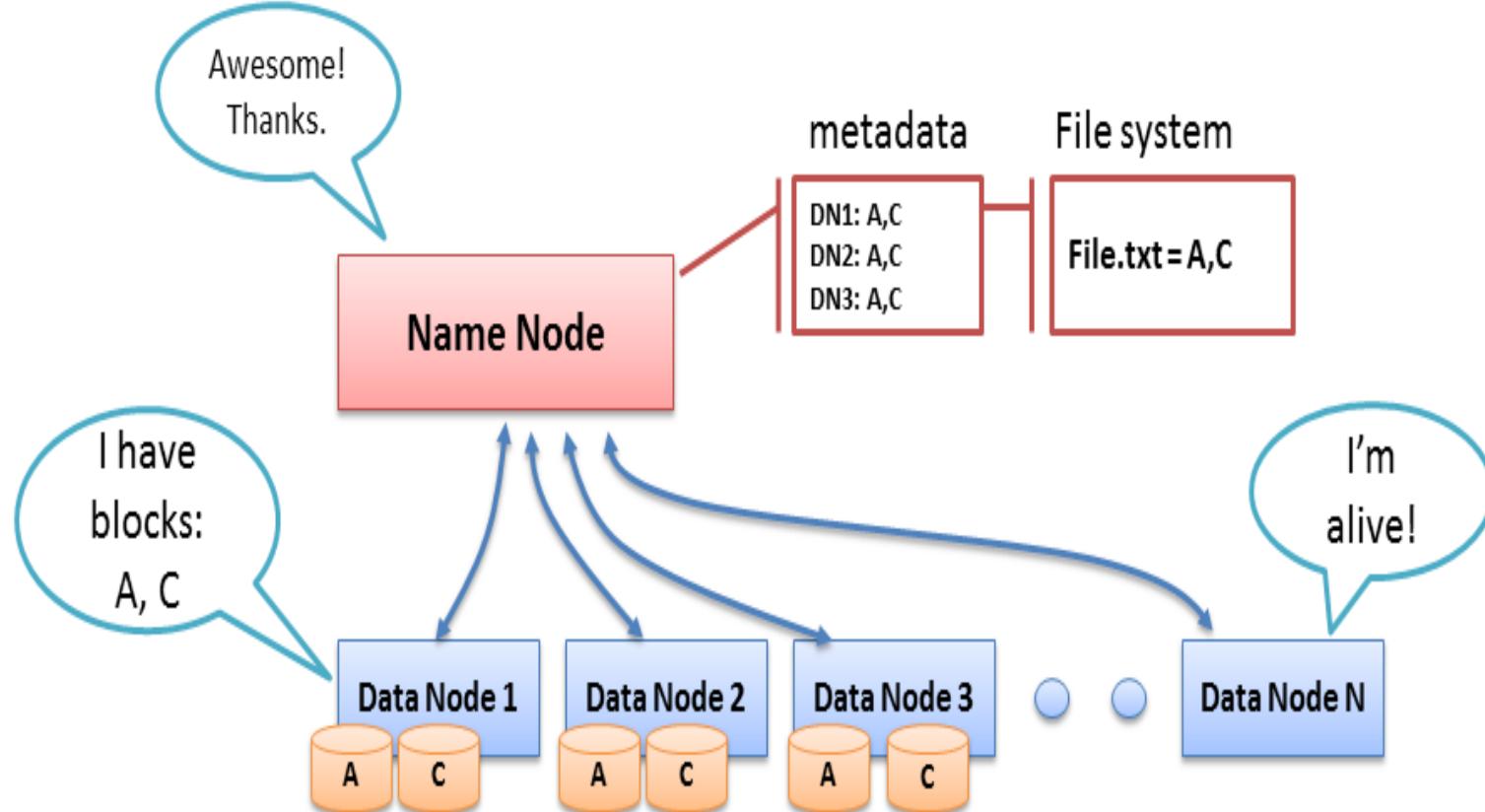
# NameNode

- NameNode je centralni kontroler HDFS-a
- NN čuva metapodatke fajl sistema, nadgleda ponašanje DN-a i koordiniše pristup podacima.
- NN ne čuva podatke nad kojima se vrši obrada , on samo ima informaciju o blokovima koji čine fajl i gde su ti blokovi locirani u klasteru
- Takođe vodi računa da svaki blok zadovolji definisani faktor replikacije

# NameNode

- Namenode je u direktnoj vezi sa ostalim čvorovima, i za komunikaciju sa DataNode demonima koristi paket određene strukture koji se naziva *heartbeat*
- Ovaj paket nosi takav simboličan naziv jer preko njega Namenode proverava da li je Datanode aktivan, i pored ovoga, ovaj paket prenosi identifikatore blokova podataka koji se nalaze na hostu Datanode demona.
- DataNodovi šalju *heartbeatove* svake tri sekunde i svaki deseti heartbeat je izveštaj o blokovima, gde DN obaveštava NN koje blokove čuva
- Ovi izveštaji služe NN da osiguraju da odgovarajući broj replika blokova (najčešće 3) postoji na različitim čvorovima, u različitim rekovima

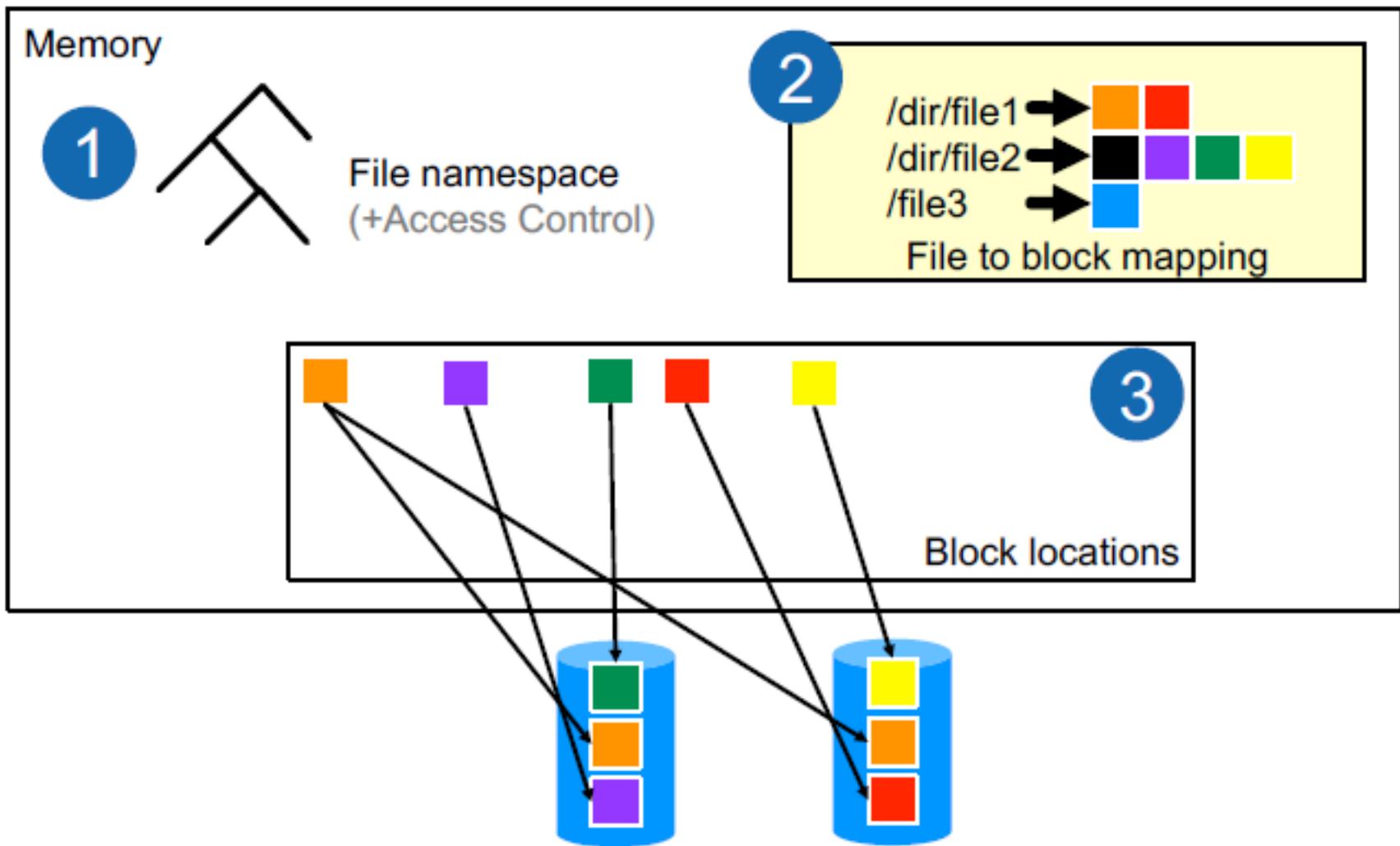
# Name Node



# NameNode

- NN je kritična komponenta HDFS-a
- Bez njega klijent ne bi mogao da upisuje i čita fajlove iz HDFS-a i bilo bi nemoguće upravljati i izvršavati MR poslove
- Iz tog razloga dobro je da NN bude neki jači server ali i bitno je obezbediti i dodatnu podršku u slučaju problema u radu NN
- Namenode čuva ove podatke u glavnoj memoriji, zato što se ovi podaci u velikim sistemima često menjaju.
- Ovakav pristup je omogućio Hadoop sistemima jednostavno proširenje klastera i nesmetan rad pri određenim tipovima otkaza.

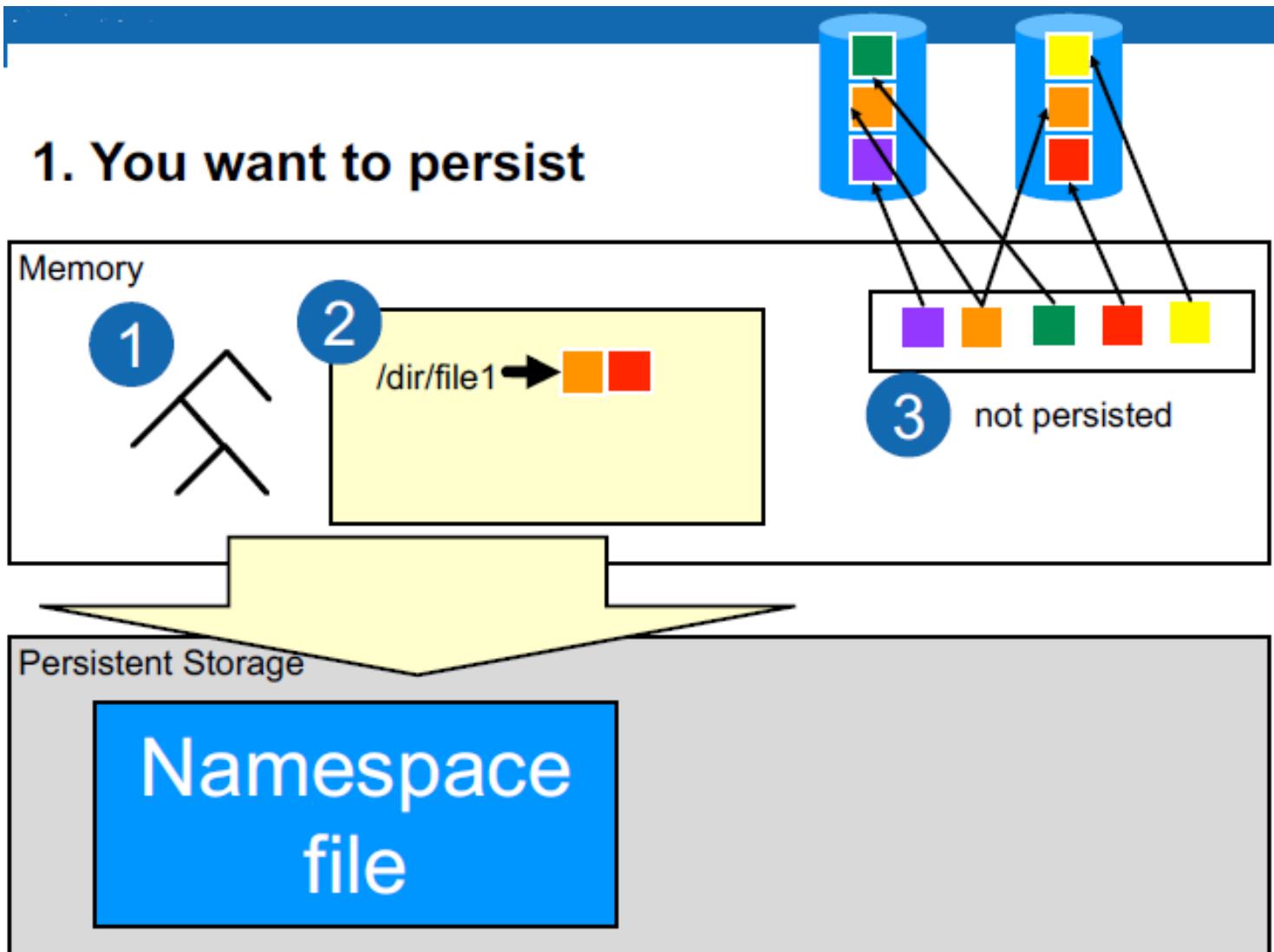
# NN glavna memorija



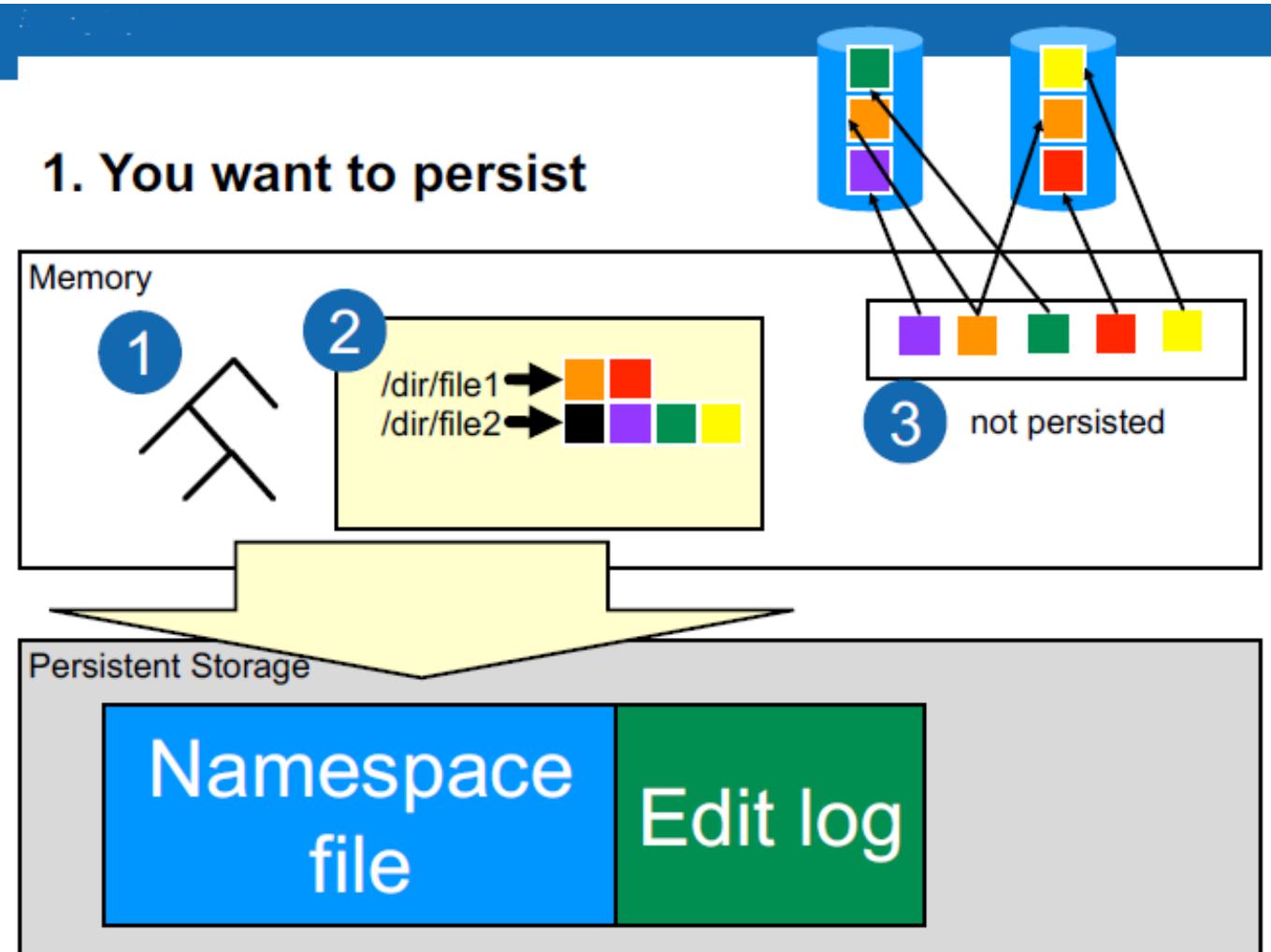
# Trajno čuvanje metapodataka NN-a

- Trajno čuvanje metapodataka NN-a na NN lokalnom file sistemu se naziva Checkpoint
- Fajl na lokalnom fajl sistemu NN-a u kome se trajno čuvaju ***promene*** koje se dešavaju u HDFS-u se zove ***edit log***
- Fajl *fsimage* (*Checkpoint image kod GFS*) je fajl u kome se čuva snapshot metapodataka fajl sistema (Namespace file)

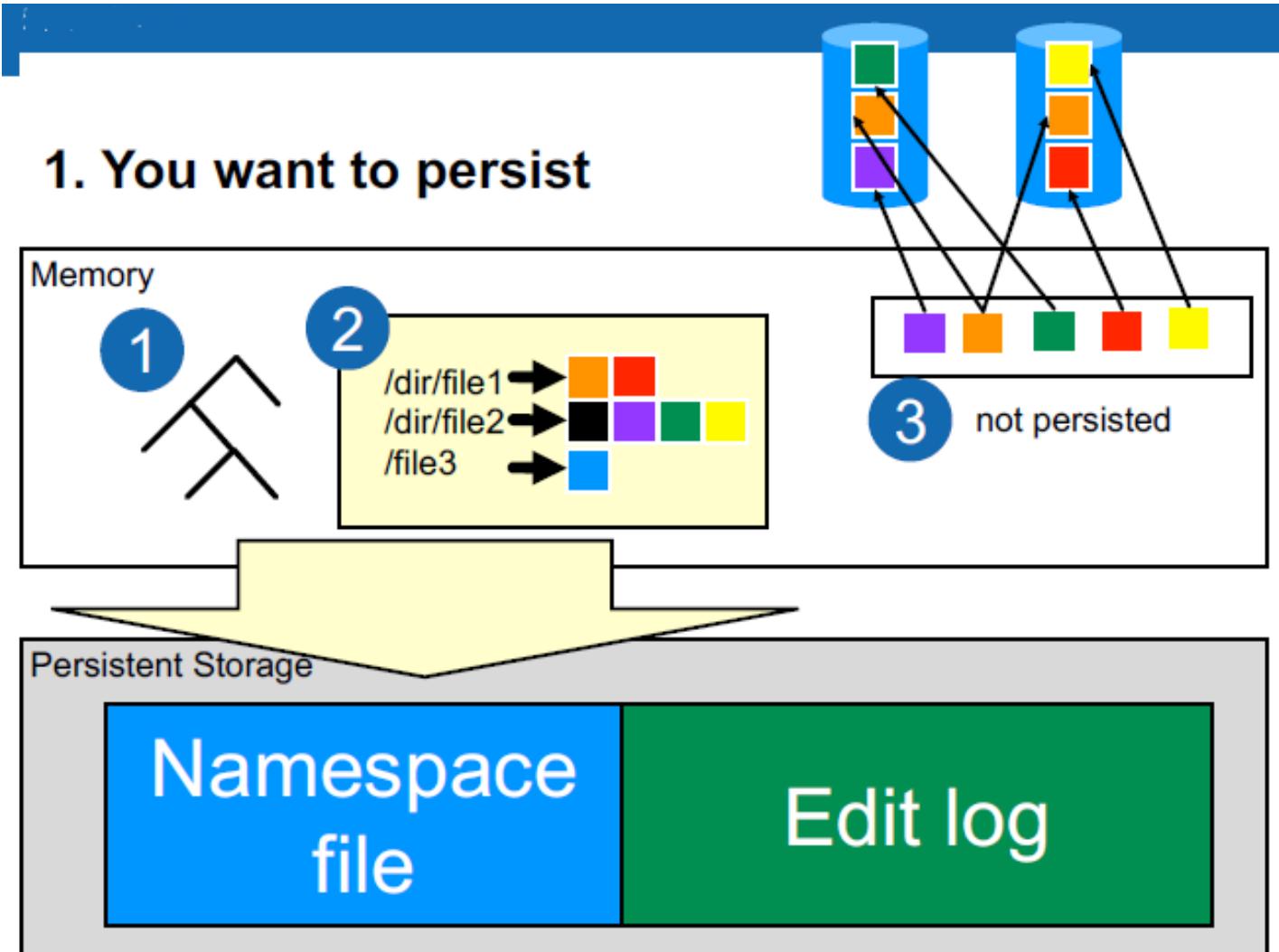
# Trajno čuvanje metapodataka NN-a



# Trajno čuvanje metapodataka NN-a



# Trajno čuvanje metapodataka NN-a

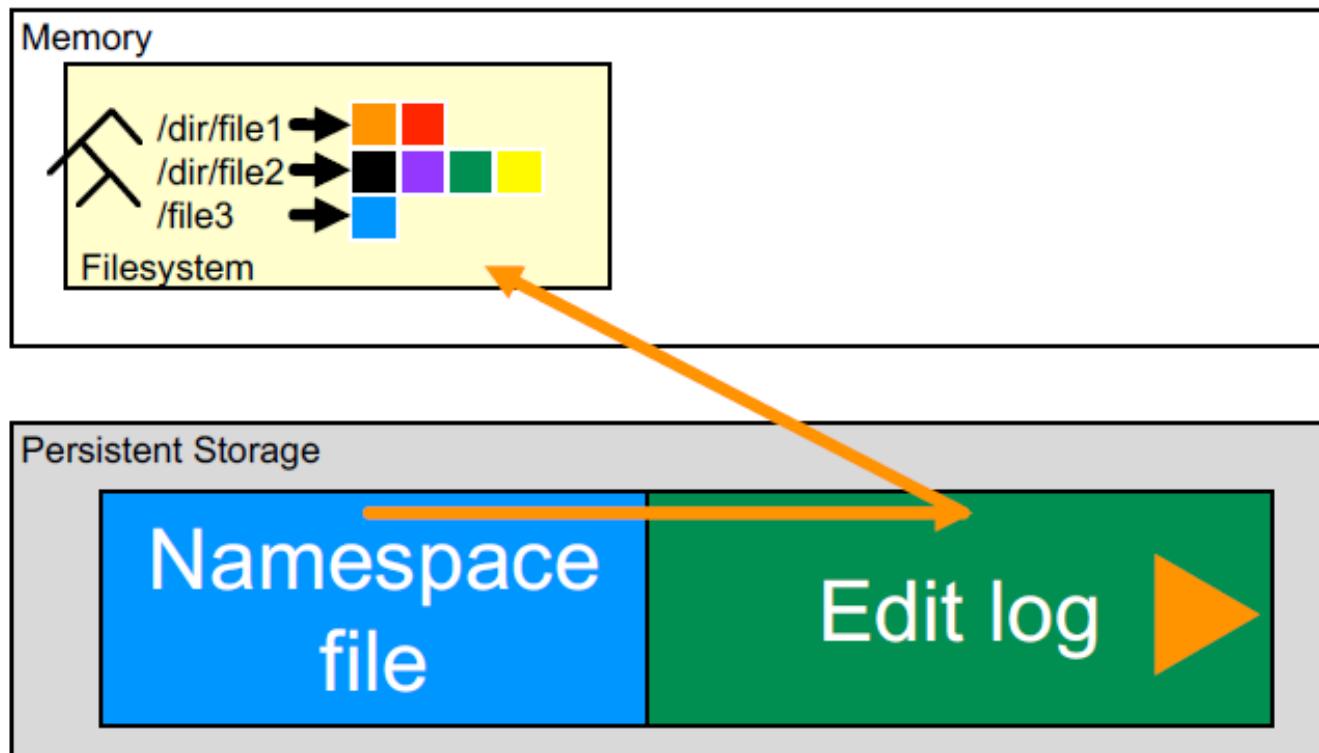


# NN Restart

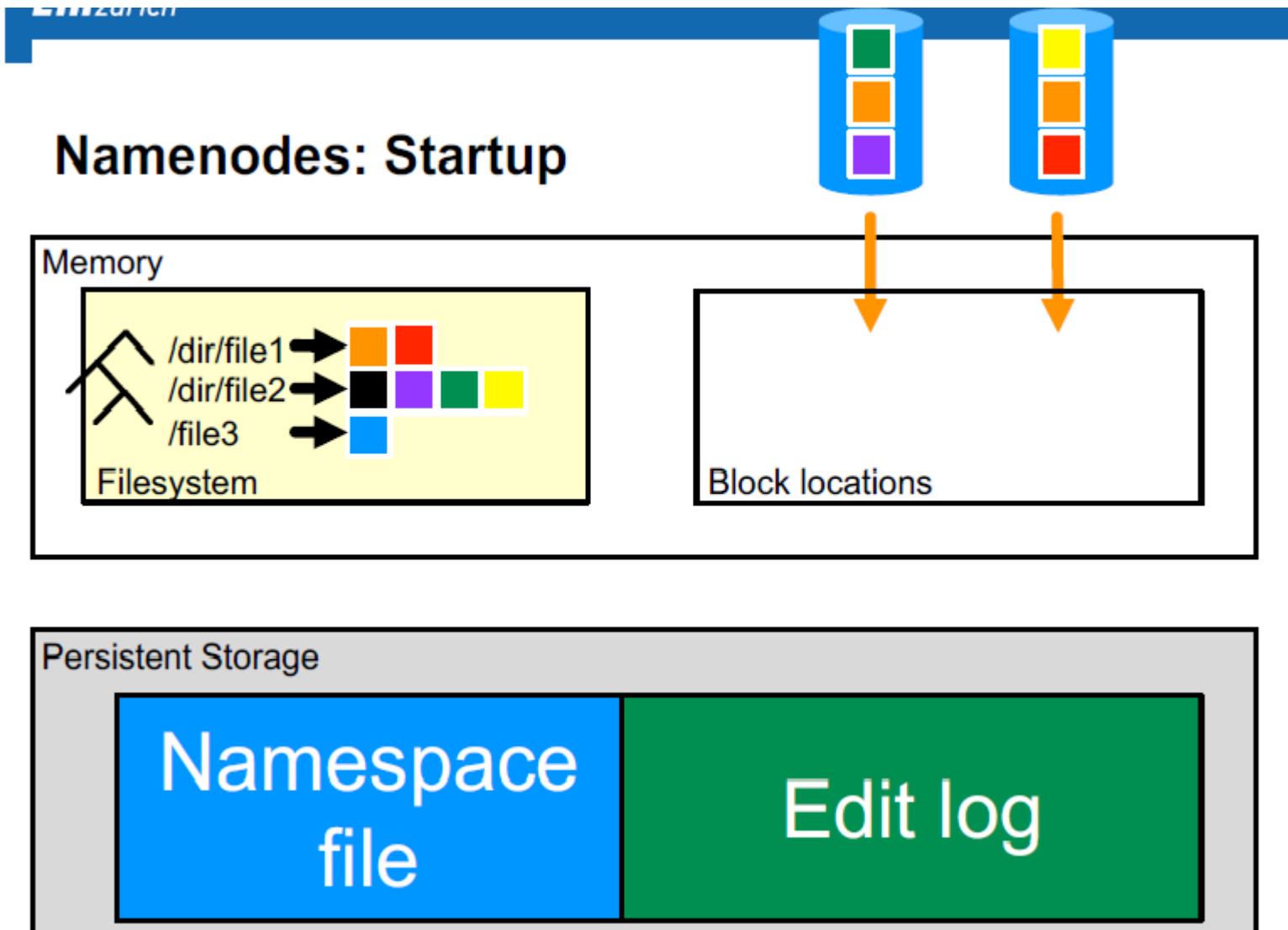
Samo u slučaju kada se restartuje NN, promene iz edit loga se primenjuju na poslednju verziju Namespace fajla(fajl fsImage) da bi se dobila nova verzija metapodataka fajl sistema.

Pošto se u klasterima retko vrši restart, edit logovi mogu narasti mnogo, tako da vreme za restart postaje mnogo dugo

## Namenodes: Startup

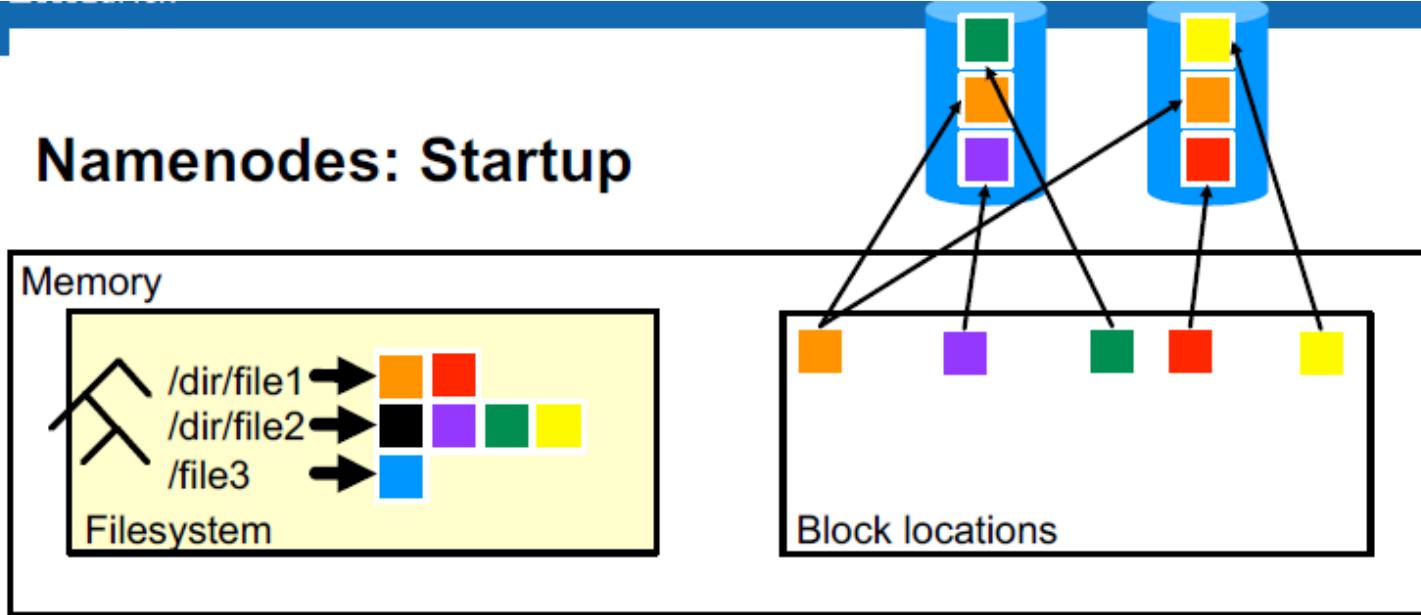


# NN Restart



# NN Restart

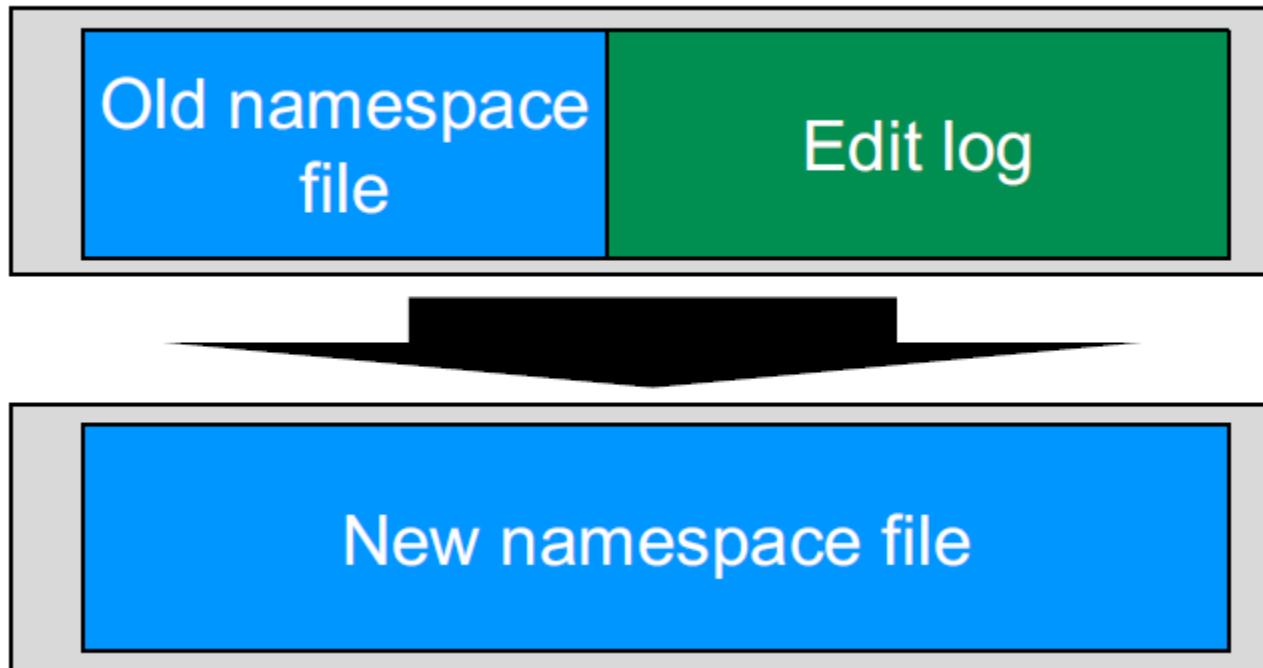
## Namenodes: Startup



# Sekundarni NN

- CP treba formirati što češće jer veliki edit log povećava vreme za restart NN (potrebni su sati za restart u slučaju edit log-a koji je formiran cele nedelje) a i u slučaju otkaza NN je bitno
- Da bi se prevazišao taj problem koristi se Secondary NN
- Secondary NN je startovan na serveru na kojem nije NN
- Secondary NN radi konkurentno sa NN kao pomoćni demon i on ne predstavlja NN backup

# Chackpoint na sekundarnom Name Node



- Secondary NN u regularnim intervalima preuzima edit log od NN, spaja ih lokalno (primenjuje promene iz edit loga na stari CP) i formira novi CP koji vraća NN.
- NN koristi ovaj CP prilikom sledećeg restarta

# Rack awareness

- Kao što je već rečeno, svaki blok podataka koji treba da se upiše biće repliciran na više mašina da bi se sprečilo da otkaz jedne mašine dovede do gubitka svih kopija podatka
- Šta bi se desilo kada bi sve kopije podataka bile locirane na mašinama u istom reku koji otkaže?
- Da bi se to izbeglo, potrebno je da se zna gde su DataNodovi locirani u mreži i tu informaciju iskoristiti u odlučivanju gde treba da budu ostale replike. To informaciju ima NameNode!

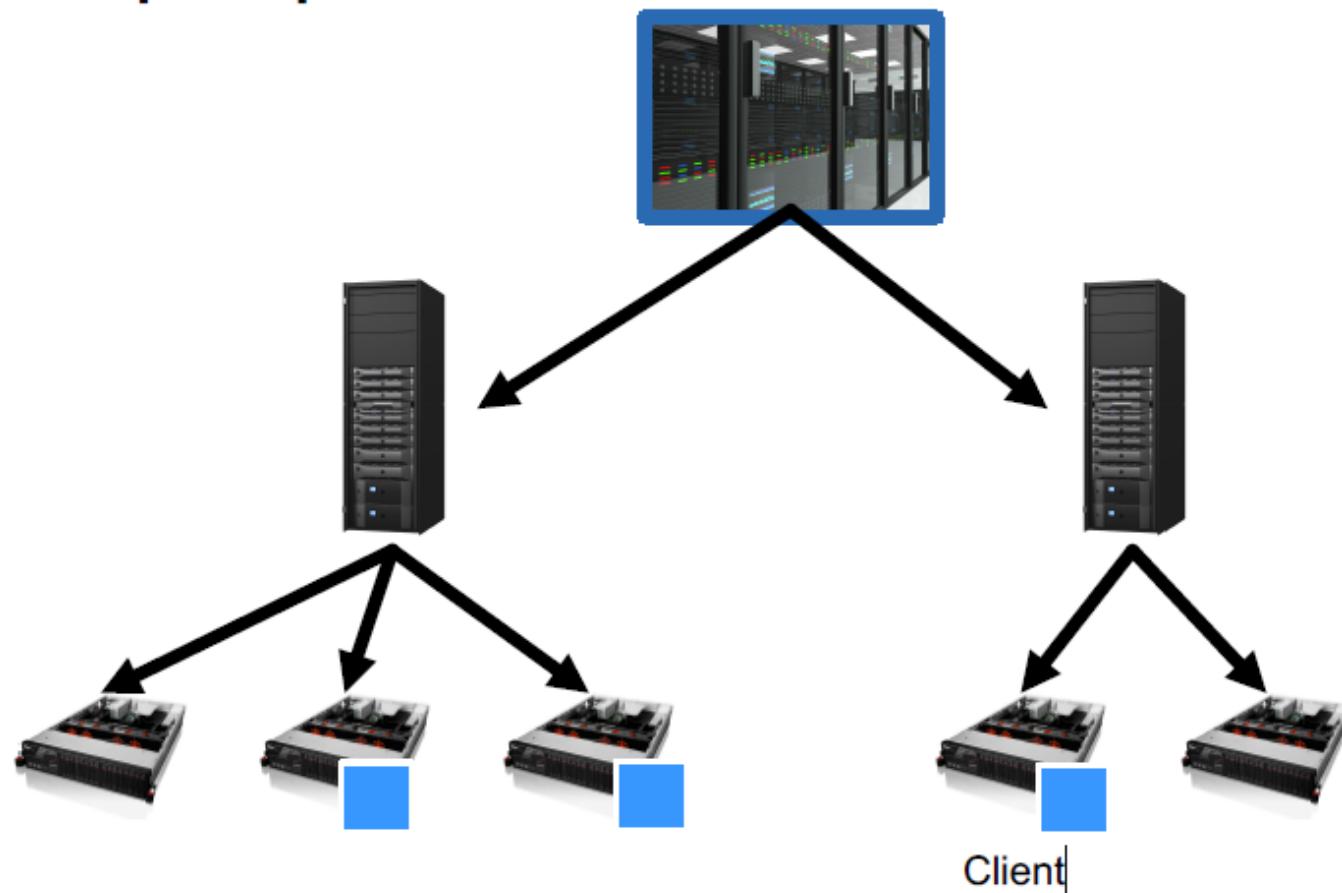
# Rack awareness



- Replica 1: same node as client (or random), rack A
- Replica 2: a node in a different rack B
- Replica 3: a node in same rack B
- Replica 4 and beyond: random, but if possible:
  - at most one replica per node
  - at most two replicas per rack

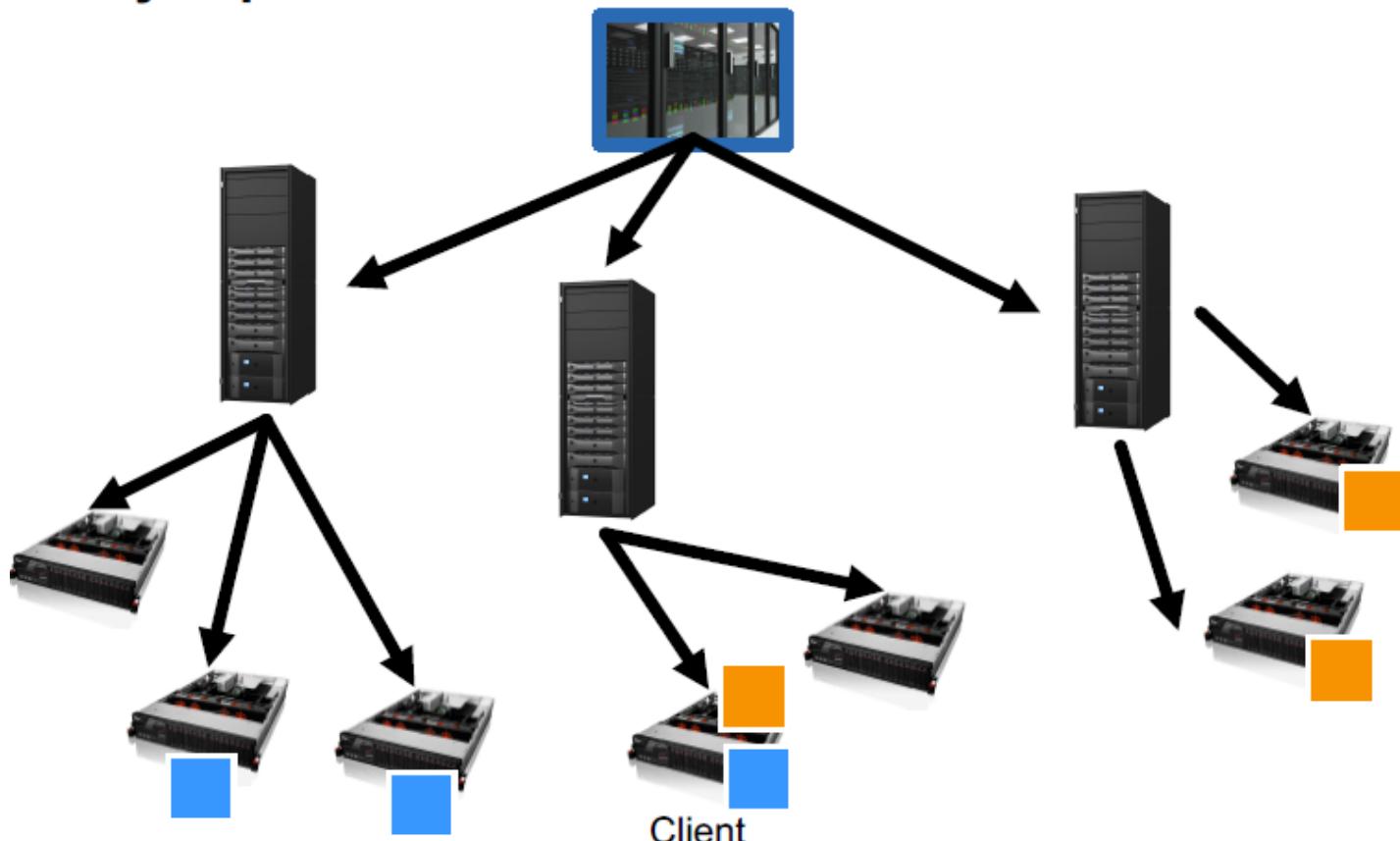
# Rack awareness

## Replica placement



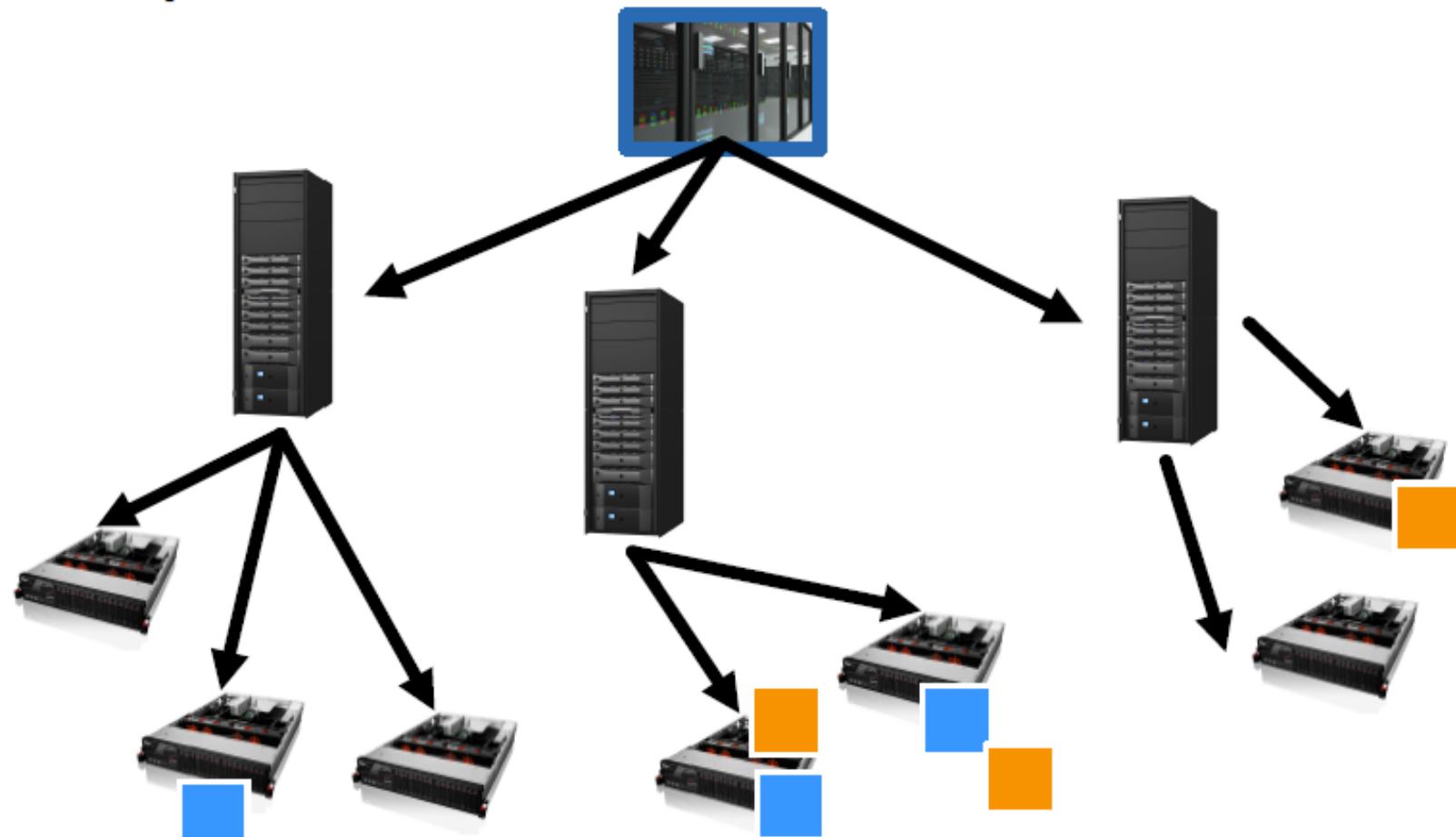
# Rack awareness

Why replicas 2+3 on other rack?



# Rack awareness

If replicas 1+2 were on same rack...



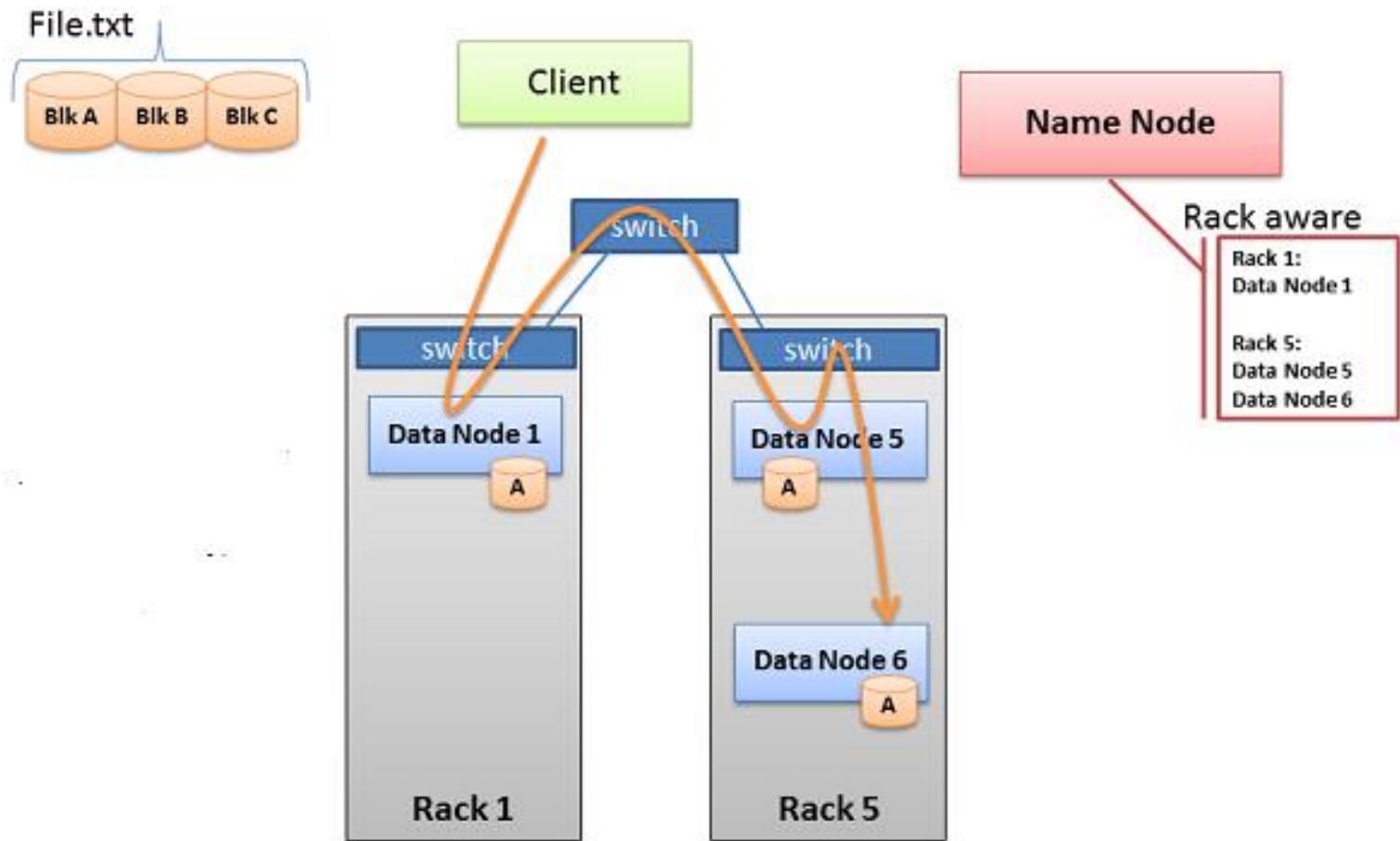
# HDFS upis

- Klijent je spremam da loaduje fajl File.txt u klaster (razbija ga u blokove A,B i C) počevši od bloka A.
- Klijent kontaktira Name Node, konsultuje ga, dobija dozvolu od NN i dobija listu od (3) DataNoda za svaki blok koji treba da bude upisan (jedinsvenu listu za svaki blok).
- NN koristi svoje Rack Awareness podatke da bi uticao na odluku koji DataNodovi će se naći na listi za pojedinačni blok
- Ključno pravilo je da za svaki blok podataka, jedna kopija se nalazi u jednom reku, a druge dve kopije u reku različitom od ovog s tim što su te dve kopije u istom reku.
- Sve liste koje dobija Klijent moraju da poštuju ovo pravilo
- Za blok A na listi su DN1, DN5 i DN6.

# HDFS upis

- Pre nego što Klijent upiše Blok A fajla File.txt u klaster, on želi da zna da su svi DataNodovi u koje treba da upiše spremni da prime blok.
- Nakon stizanja potvrde, od DN6 do DN5, od DN5 do DN1, i od DN1 do klijenta, Klijent je spreman da upiše blok A.
- Prilikom upisa DN-ovi formiraju pipeline, u redosledu koji minimizira rastojanje od klijenta do poslednjeg DN-a

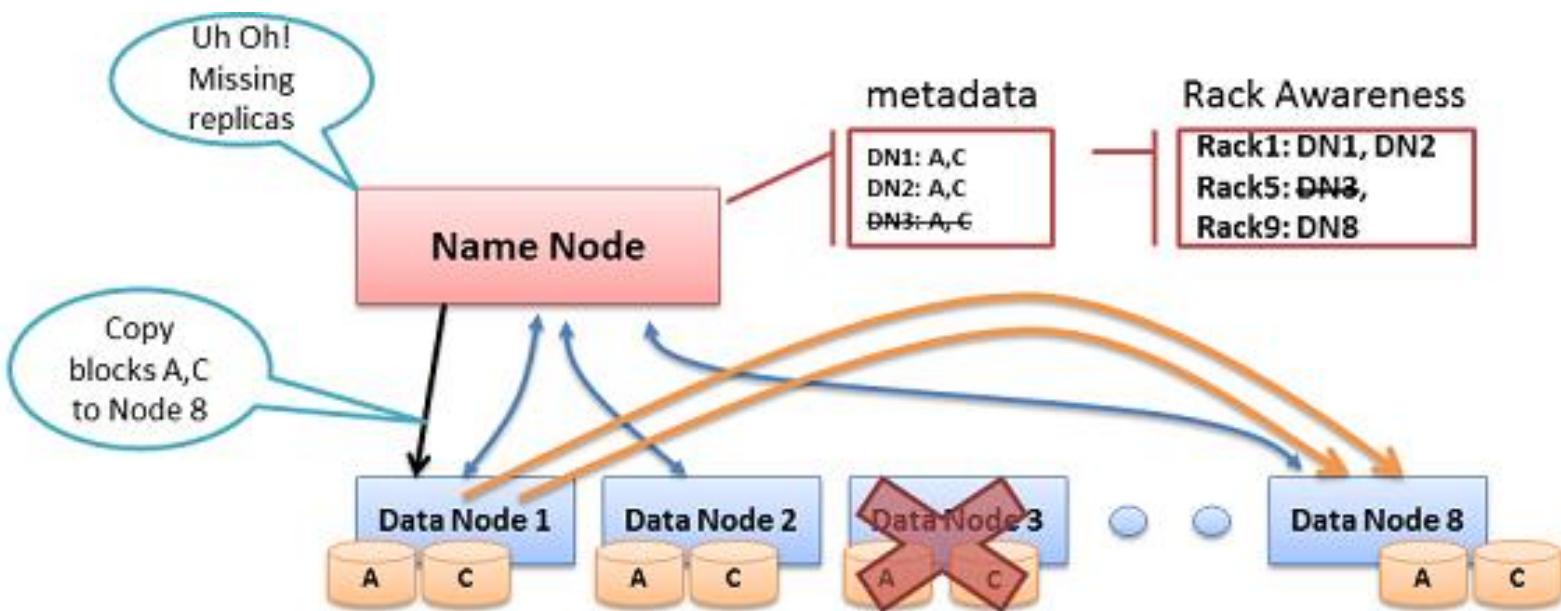
# HDFS Upis



# Rereplikacija blokova

- Ako NN prestane da prima *heartbeat*-ove od DN on prepostavlja da je računar “mrtav” i da su podaci na njemu nedostupni.
- Pošto je NN od ovog DN primao izveštaje on zna koji blokovi podataka su bili sačuvani na njemu i može doneti odluku o rereplikaciji blokova na druge DN.
- Pri tome će konsultovati Rack Awerness podatke da bi ispoštovao pravilo da je prva kopija u jednom reku a druge dve kopije u istom u reku, kada odlučuje koji DN će dobiti nove kopije blokova

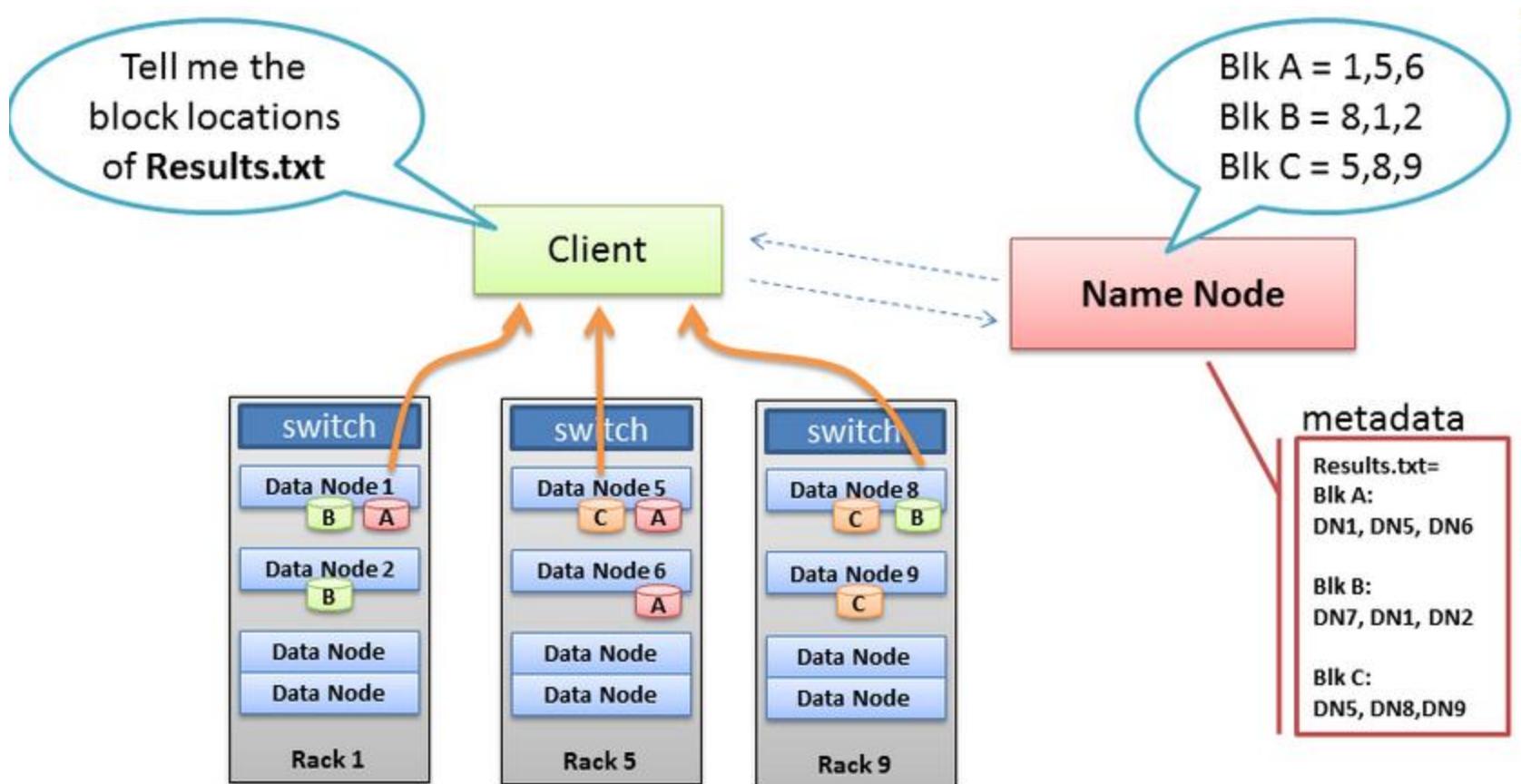
# Rereplikacija blokova



# Čitanje podataka iz HDFS-a

- Kad Klijent želi da pročita rezultujući fajl iz HDFS, konsultuje NN i pita za lokacije blokova u rezultujućem fajlu
- NN za svaki blok vraća listu DN-ova koji ga sadrže
- Klijent bira za svaki blok prvi DN iz liste (lokacije blokova su sortirane prema udaljenosti od klijenta) tj. najbližu repliku
- Klijent čita blokove sa odgovarajućih DN-a sekvencialno, jedan po jedan

# Čitanje podataka iz HDFS-a



# HDFS upis

- Kopiranje ulaznih fajlova sa direktorijuma lokalnog FS **input** u direktorijum **inputdfs** distribuiranog fajl sistema:
- natalija@natalija-VirtualBox:~/Apps/hadoop-1.2.1\$ **bin/hadoop fs -put ..../MaxTemperature/input inputdfs**

# Web interfejs za HDFS nakon kopiranja

The screenshot shows a web browser window with the following details:

- Address Bar:** localhost:50075/browseDirectory.jsp?dir=%2Fuser%2Fnatalija%2Finputdfs&namenodeInfoPort=50070
- Title Bar:** Contents of directory /user/natalija/inputdfs
- Form:** Goto :
- Text Link:** Go to parent directory
- Table:** A table displaying file information:

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
Sample.txt	file	4.59 KB	1	64 MB	2013-12-10 12:04	rw-r--r--	natalija	supergroup
- Text Link:** Go back to DFS home

# Kopiranje na lokalni FS

- Kopiranje izlaznih fajlova iz HDFS (outputdfs) u lokalni fajl sistem na folder output2:
- natalija@natalija-VirtualBox:~/Apps/hadoop-1.2.1\$ **bin/hadoop fs -get outputdfs output2**

# GFS (Google File System)

## GFS vs. HDFS

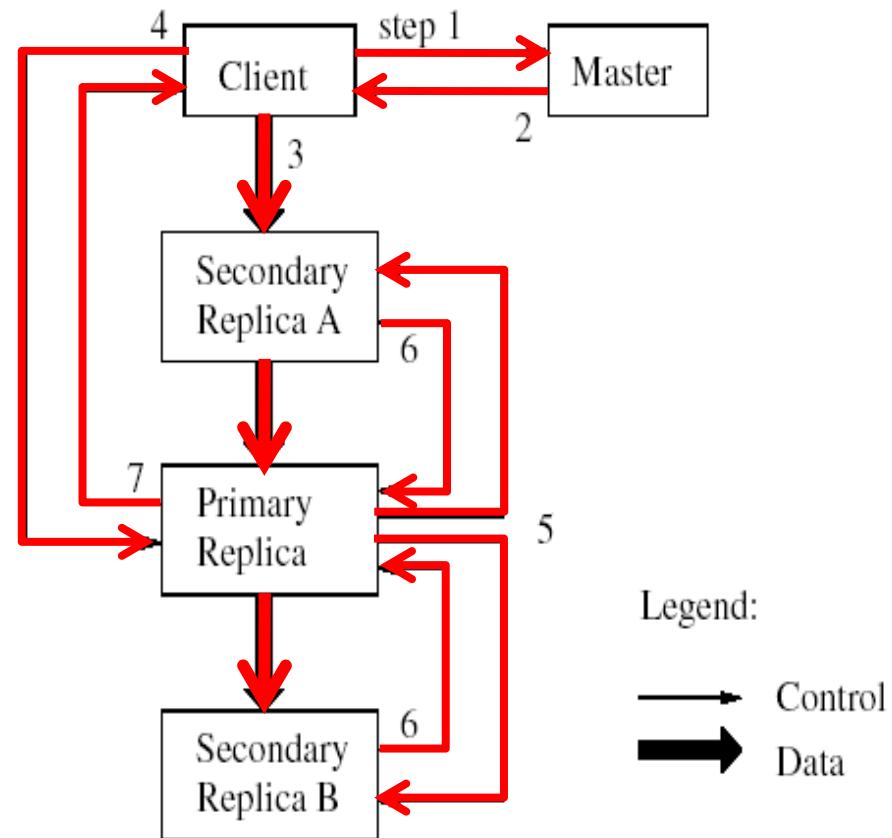
GFS	HDFS
Master	NameNode
chunkserver	DataNode
operation log	journal, edit log
chunk	block
<b>random file writes possible</b>	<b>only append is possible</b>
multiple writer, multiple reader model	single writer, multiple reader model
chunk: 64KB data and 32bit checksum pieces	per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp)
default block size: 64MB	default block size: 128MB

# Konzistentnost u GFS-u

- Pošto su chunk-ovi replicirani u GFS-u potrebno je održati konzistentnost replika u slučaju operacija koje modifikuju podatke(mutacija, promena)
- Kada se zahteva mutacija chunk-a, master dodeljuje “zakup” chunk-a (lease) čvoru jedne od replika, koja je sad imenovan kao primarni(primary)
- Ovo je od krucijalne važnosti kod konkurentnog upisa i ima ulogu u rasterećenju mastera tako što se upravljanje prenosi na chunk server(primary)
- Primary je odgovoran za obezbeđenje serijskog redosleda mutacija za sve konkurentne mutacije istog chunka koje trenutno čekaju
- Zakup koji je dobio Primary omogućava da napravi mutacije na svojom lokalnom kopijom podataka, ali i da kontroliše i primeni isti redosled mutacija i na ostalim (sekundarnim) kopijama

# Mutacije

1. Klijent “pita” master za lokacije replika i da dodeli čvoru na kojoj se nalazi jedna od njih ulogu primary (dodeljuje joj zakup)
2. Master odgovara na upit zahtevanim informacijama
3. Klijent šalje podatke, koji treba da se upišu u chunk, svim čvorovima sa replikama i oni bivaju smešteni u odgovarajući bafer svakog od čvora sa replikom i ne bivaju upisani dok ne dobiju nalog za upis
4. Klijent šalje nalog za upis primary-ju
5. Primary određuje serijski redosled konkurentnih upisa i primenjuje ga i na svoje lokalne podatke i na podatke sekundarnih replika
6. Čvorovi sa sekundarnim replikama odgovaraju Primary da su mutacije uspešno obavljene



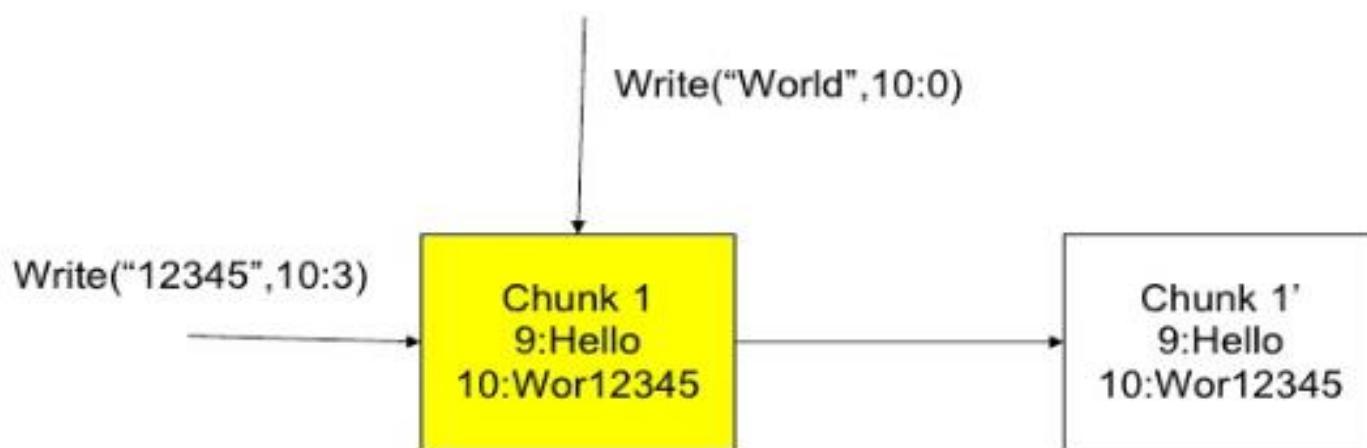
# Concurrent Write

- Ako dva klijenta konkurentno vrše upis u isti region fajla nešto od ove dve stvari se može desiti:
  - Preklapajući region može sadržati podatke samo jednog od ta dva upisa
  - Preklapajući region može sadžati miks podataka ova dva upisa

# Concurrent Write (primer)

## Multiple Writers

Consistent and Undefined



# Write/Concurrent Write

