

# Uvod

## Definicija distribuiranih sistema

Distribuirani sistem je skup **nezavisnih** računara koji svojim korisnicima izgledaju kao jedan **koherentan** (povezan) sistem. Računari koji čine komponente distribuiranih sistema **komuniciraju isključivo slanjem poruka**.

## Prednosti DS u odnosu na centralizovane sisteme

- DS ima bolji odnos cena/perfomanse od velikih računara.
- DS ima veću ukupnu moć obrade od mainframe računara
- Ako 15% mašina otkáže, DS i dalje može da obavi posao, dok kod centralizovanog sistema ako računar otkáže aplikacija ne može da se izvršava
- DS se može postepeno povećavati dodavanjem novih računara
- DS omogućava da više korisnika pristupa zajedničkim bazama podataka i periferijama, kao i olakšanu komunikaciju između ljudi
- DS omogućava da se opterećenje celog sistema efikasno raspodeli među trenutno dostupnim računarima

## Mane DS u odnosu na centralizovani sistem

- DS je mnogo kompleksniji od centralizovanog sistema jer je teže razviti SW za DS nego za centralizovane sisteme
- Ako računari u DS-u nisu korektno povezani ceo sistem može da otkáže
- Dostupnost i pristup resursima je jednostavna i laka, pa se javljaju problemi bezbednosti

## Osnovne osobine DS-a

1. Heterogenost
2. Transparentnost
3. Otvorenost
4. Skalabilnost

## Heterogenost

Odnosi se na to da je DS sastavljen od heterogenog skupa računara. Računari mogu da se razlikuju po hardveru, po skupu instrukcija, po prezentaciji podataka i po OS-u.

## Transparentnost

Odnosi se na to da DS mora da skriva činjenicu da se njegovi procesi i resursi fizički distribuirani. Transparentnost omogućava da se DS koristi sa istom lakoćom kao i jednoprocesorski sistem. Primer je Web koji nam omogućava da pristupimo željenim informacijama samo jednim klikom, a da pri tim ne moramo da budemo svesni gde se ti podaci zaista nalaze.

### **1. Pristupna transparentnost**

Podacima i resursima se treba pristupati na jedinstven način, bez obzira da li su oni smešteni na udaljenom ili lokalnom računaru.

### **2. Lokacijska transparentnost**

Korisnik ne mora da zna gde se traženi resurs fizički nalazi u sistemu. To se postiže imenovanjem – primer je URL ime <http://www.prenhall.com/index.html> ne nagoveštava gde se nalazi glavni web server izdavačke kuće PrenHall, a korisnici sa bilo koje lokacije mu pristupaju na isti način.

### **3. Migraciona transparentnost**

Resurs može da promeni svoju lokaciju, a klijent to ne treba da primeti ni zna. Resurs može da menja svoju lokaciju, a da se ne menja njegovo ime.

### **4. Transparentnost konkurencije**

Omogućava da više konkurentnih procesa tj. da više procesa istovremeno koriste isti resurs, a da korisnici ne primete da se isti resurs koristi istovremeno. Ovakav istovremeni pristup istom resursu mora da ostavi resurs u konzistentnom stanju i to se postiže serijalizacijom deljenog resursa.

### **5. Transparentnost replikacije**

Omogućava postojanje većeg broja replikacija (kopija) istog resursa u DS-u da bi se povećale performanse i smanjilo komunikaciono kašnjenje. Kopija resursa se postavlja bliže mestu odakle se obavlja pristup tom resursu, a sve kopije imaju isto ime. Korisnici ne treba da budu svesni postojanja više kopija resursa.

### **6. Transparentnost na otkaze**

DS koji je otporan na otkaze mora da bude u stanju da izvrši realokaciju resursa sa dela koji je u kvaru na delove sistema koji korektno rade, tako da korisnik ne primeti da je došlo do kvara.

### **7. Transparentnost na paralelizaciju**

Paralelizacija procesa mora da se izvršava transparentno za aplikativnog programera i korisnika aplikacije. Korisnik ne treba da bude svestan da je i kako je aplikacija paralelizovana i gde se izvršavaju procesi.

## Otvorenost

Otvoreni DS dozvoljava dodavanje novih servisa i omogućava dostupnost servisa različitim klijentima. DS koji je otvoren pruža servise tj. usluge koje se definišu interfejsima. Interfejsi se opisuju pomoću posebnog jezika – IDL-a. Primer je Web servis kome može da se pristupi preko mnogo različitih klijenata – Chrome, Opera, Explorer itd.

## Skalabilnost

Skalabilnost ili proširljivost DS-a posmatra se kroz 3 dimenzije:

- **Skalabilnost u odnosu na broj korisnika i resursa**
- **Skalabilnost u odnosu na geografsku udaljenost resursa i korisnika**
- **Skalabilnost u odnosu na broj administrativnih domena (administrativna skalabilnost)**

Da bi skalabilnost bila izvodljiva koriste se **decentralizovani algoritmi**, koji imaju sledeće osobine:

- Ni jedna mašina nema kompletnu sliku i informaciju o stanju sistema
- Mašine donose odluke na osnovu lokalnih podataka
- Otkaz mašine ne narašava sistem
- Nema pretpostavke o globalnom časovniku, jer u DS-u nije moguće postoići tačnu sinhronizaciju časovnika

## Tehnike skaliranja

### 1. Skrivanje komunikacionog kašnjenja

- Koriste se asinhronne komunikacije umesto sinhronih, pa se klijent ne blokira dok čeka server, nego radi svoj posao, dok mu ne stigne prekid koji ga obaveštava da je dobio odgovor od servera
- Asinhronne komunikacije nisu od koristi kod interaktivnih aplikacija. Tada je rešenje download-ovati deo koda na klijentsku stranu da bi se ubrzala obrada.

### 2. Distribucija

- Komponente se dele na manje delove, a zatim se ti delovi distribuiraju na više mašina u sistemu. Primer je DNS i Web.

### 3. Replikacija

- Pomaže da se postignu bolje performanse. U geografski distribuiranim sistemima poželjno je da postoji kopija resursa blizu mesta korišćenja da bi se smanjilo komunikaciono kašnjenje.

Problem tehnika skaliranja:

- Postojanje više kopija dovodi do problema konzistencije, jer modifikacija jedne kopije dovodi do neslaganja sa ostalim kopijama. Da bi se kopije sinhronizovale potrebna je globalna sinhronizacija, koju je gotovo nemoguće postići u DS-u.

## Middleware

**Distribuirani OS** – operativni sistem koji upravlja skupom **nezavisnih** računara na različitim lokacijama (gde se podaci čuvaju i procesiraju) i čini da oni korisnicima izgledaju **kao jedan računar**. To je čvrsto spregnut OS za multiprocesore i **homogene** multi-računare.

**Mrežni OS** – specijalni OS za pružanje mrežnih funkcionalnosti. Pomaže u upravljanju podacima, korisnicima, grupama, bezbednošću i drugim mrežnim funkcijama. Slabo spregnut OS za **heterogene** multiračunare. Mrežni OS pruža mogućnost korisnicima da pristupe uslugama (servisima) koje su locirane na nekoj drugoj mašini.

**Middleware** – računarski SW koji pruža usluge (servise) višeg nivoa softverskim aplikacijama. Da bi se sakrila heterogenost platforme na kojoj je sistem izgrađen od same aplikacije, kao i da bi se sakrila komunikacija, mrežni OS se nadograđuje dodatnim SW slojem – middleware. Postoji dosta protokola opšte namene koji su od koristi za mnoge aplikacije, a ne mogu se kvalifikovati kao transportni protokoli. Ti protokoli spadaju u grupu middleware protokola. Neke od usluga koje middleware protokoli pružaju je **autentifikacija** i **autorizacija**. Middleware sistemi nude kompletan skup usluga aplikacija i ne dozvoljavaju korišćenje ničeg drugog do njihovih interfejsa prema uslugama.

Middleware komunikacioni protokoli pružaju viši nivo usluga i oslobađaju aplikativnog programera detalja vezanih za komunikaciju između procesa. Komunikacija je skrivena iza poziva **procedure** ili **metoda**. Udaljena procedura je *Remote Procedure Call* – **RPC**, a udaljena metoda – *Remote Method Invocation* – **RMI**.

## RPC komunikacioni middleware model

Model koji se zasniva na pozivu udaljenih **procedura**. Resursi se modeluju kao procedure (funkcije). Na ovaj način skriva se mrežna komunikacija, procesi mogu da pozivaju udaljene procedure čija je implementacija locirana na drugoj mašini. Proces koji poziva udaljenu proceduru ima utisak da je pozvao lokalnu proceduru i nije svestan cele komunikacije kroz mrežu.

## Objektno-orijentisan middleware model

Resursi se modeluju kao **objekti** koji sadrže skup podataka i funkcija nad podacima. Udaljenom resursu pristupa se kao objektu. Svaki objekat implementira interfejs koji skriva sve unutrašnje detalje objekta od korisnika.

Interfejs sadrži samo deklaraciju metoda koje objekat implementira i jedino što proces vidi jeste interfejs objekta. Distribuirani objekti na su uglavnom implementirani tako da je svaki objekat lociran na jednoj mašini, a interfejs je dostupan na mnogim drugim mašinama.

## Tipovi DS-a

- Na arhitekturnom nivou:
  - **Klijent server**
  - **Peer to peer**
- U odnosu na oblast primene:
  - **Distribuirani računarski sistemi** – klasteri i grid
  - **Distribuirani informacioni sistemi** – za obradu **transakcija** i za integraciju poslovnih aplikacija
- Ugrađeni (sveprisutni) distribuirani sistemi – mobilni računari, embeded sistemi, komunikacioni sistemi

## Distribuirani računarski sistemi

1. **Klasteri** – grupe sličnih računara na malom rastojanju i povezani su preko veoma brze lokalne mreže LAN-a. **Svaki računar ima isti OS**. Ključna osobina je **HOMOGENOST** jer svi računari imaju isti OS i svi su povezani istom mrežnom. Najčešći tip middleware-a kod klastera je **MPI** – *Message Passing Interface*.
2. **Grid** – sastoji se od grupe računarskih sistema koji mogu da budu locirani na velikom geografskom području, koji mogu da budu **HETEROGENI** u pogledu HW-a i SW-a, kao i mrežne tehnologije koju koriste.

## Sistemi za obradu transakcija

**Transakcija je 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 obavljene transakcije, sistem mora da bude u poznatom konzistentnom stanju, tako što obezbeđuje da se, ili sve operacije koje su međusobno zavisne izvrše, ili da se ni jedna od njih ne izvrši.

Primer je plaćanje platnim karticama – bankomat ili isporuči novac i umanju vrednost na računu – ili se obe ove operacije obave ili ni jedna.

### ACID osobine transakcija

Svaka transakcija mora da zadovolji ACID test pre nego što se dozvoli modifikacija sadržaja:

1. **A – Atomicity – Atomičnost** – transakcija mora kompletno da se obavi ili da se uopšte ne obavi
2. **C – Consistency – Konzistentnost** – transakcija ne ugrožava skup ograničenja sistema.
3. **I – Isolation – Izolacija** – transakcije koje su konkurentne moraju da ostave bazu podataka u istom stanju kao da su se izvršile sekvencijalno. Dakle, dve konkurentne transakcije moraju biti serijalizovane (da se izvršavaju u nekom redosledu).
4. **D – Durability – Trajnost** – kada se transakcija jednom obavi, ne može se poništiti i ostaće trajna čak i u slučaju otkaza sistema (nestanak struje).

### Ugrađeni (embedded) sistemi

Uređaji ove vrste su uglavnom mali, napajaju se pomoću baterija, mobilni su i imaju uglavnom bežične mreže. Ovde je bitno to da postoji odsustvo administrativnog upravljanja. Primer su kućni sistemi, elektronski uređaji za nadzor pacijenata, senzorske mreže (merenje temperature, vlažnosti itd).

# Komunikacija

## RPC – Remote Procedure Call

**RPC** je mehanizam za poziv procedura na udaljenim mašinama. Klijent je program u okviru koga se poziva procedura, a server je onaj koji nudi usluge (service) – procedure. Osnovna ideja RPC-a je da poziv udaljene procedure izgleda što sličnije pozivu lokalne procedure.

Client:

```
...  
bar = foo(a,b);
```

...

Server:

```
Int foo (int x, int y)  
{  
    if(x>100) return y-2;  
    else if(x>10) return y - x;  
    else return (x + y);  
}
```

Parametri a i b se upakuju i šalju putem mreže. Kada poruka stigne do servera, otpakuje se i izvuku se parametri u x i y. Izvrši se funkcija foo() i vrati neki rezultat koji se pakuje u poruku. Poruka preko mreže ide nazad u klijent. Klijent ne zna šta se desilo u pozadini, jer želi da samo pozove funkciju koja radi šta treba.

Prenos parametara može da se obavi:

- call-by-value (po vrednosti) – vrednosti parametara se kopiraju u stek, pa ako pozvana procedura modifikuje podatke, originalni podaci se ne modifikuju.
- call-by-reference (po referenci) – adrese parametara se smeštaju u stek, pa ako pozvana metoda modifikuje podatke, oni se zaista u originalu promene.
- **call-by-copy/restore** – vrednosti parametara se stavljaju na stek, kao kod poziva po vrednosti, a nakon završetka poziva procedure, modifikovane vrednosti se upisuju preko originalnih vrednosti parametara kao kod poziva po referenci.

Ono što se kod RPC-a koristi je copy/restore način prenosa parametara.

Da bi se obavio poziv udaljene procedure neophodno je locirati udaljeni host i odgovarajući proces na hostu. Postoje 2 tipa povezivanja:

- 1) **Statičko povezivanje** – klijent zna koji server treba da se kontaktira i adresa tog servera je u klijent stub-u. Kada klijent pozove odgovarajuću proceduru, klijent stub samo prosledi poziv te procedure serveru čiju adresu ima. Poseban program **PORTMAPPER** na udaljenom serveru pamti **preslikavanja imena programa, broja verzije i broj porta**.
- 2) **Dinamično povezivanje** – postoji centralizovana baza podataka smeštena u **name i directory serverima** i ona može locirati server koji obezbeđuje željeni servis. Name i Directory serveri vraćaju adresu traženog servera na osnovu potpisa procedure koja se poziva. Kada klijent

pozove udaljenu proceduru, klijent stub kontaktira Name server da bi dobio adresu servera koji izvršava tu udaljenu proceduru. Name server 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 Name serveru. Serverski stub registruje serversku proceduru u Name i Directory serveru prilikom startovanja servera.

## Sun RPC

Jedna od najkorišćenijih RC implementacija je **SunRPC**. Programski jezici ne podržavaju RPC, pa se koristi **SunRPC kompajler** koji generiše neophodne **stub-ove** sa klijent i server strane. Kompajler se zove **rpcgen**. SunRPC obezbeđuje jezik za definiciju interfejsa IDL, koji omogućava deklaraciju procedura slično deklaraciji prototipa funkcija u C-u. IDL fajl može da sadrži definicije tipova, deklaracije konstanti, deklaracije procedura i druge informacije potrebne za ispravno pakovanje/raspakovanje pramatara.

**Da bi poziv udaljene procedure bio moguć i izgledao kao poziv lokalne procedure, koriste se STUB FUNKCIJE. Stub funkcija ima isti interfejs kao i udaljena procedura.**

- **Stub-ovi sa klijent strane** strane su zaduženi za pakovanje parametara u mrežnu poruku i slanje preko mreže kao i za raspakovanje rezultata.
- **Stub-ovi sa serverske strane** su zaduženi za pakovanje rezultata u mrežnu poruku i slanje preko mreže kao i za raspakivanje parametara. Tom prilikom obavljaju konverzije iz lokalnog u standardni način predstavljanja podataka, a i obrnuto.

Klijent mora da zna ime udaljenog servera. Server registruje svoje usluge preko portmapper daemon procesa na serverskoj mašini, na portu 111.

### Klijent i server stub-ovi

Klijent stub je funkcija koja liči na lokalnu funkciju, međutim, ona stvarno ne sadrži kod koji treba da se izvrši u toj proceduri, već sadrži kod za slanje parametara i prijem rezultata kroz mrežu. Samo telo funkcije je i obavlja se na serveru.

SunRPC obezbeđuje jezik za definiciju interfejsa IDL i on omogućava deklaraciju procedura.

**RPC kompajler (rpcgen) generiše 3 fajla, komandom `rpcgen -C primer.x`:**

- 1) **Header fajl** – sadrži id interfejsa, definiciju tipova, konstanti i prototipova. Uključuje se i u server i u klijent kod.
- 2) **Klijent stub funkcija** – sadrži procedure koje će klijent program pozvati. Ove procedure su zadužene za pakovanje parametara u poruke, slanje poruka, prijem poruka, izvlačenje rezultata poziva procedure i prosleđivanje rezultata klijentu
- 3) **Server stub funkcija** – sadrži procedure koje se pozivaju kada stigne poruka do servera i koje zatim pozivaju odgovarajuću serversku proceduru.

Pored njih imamo i klijent i server program.



**Klijent program** client.c uspostavlja vezu sa serverom dobijanjem porta na kome se izvršava server program, na osnovu imena servera, broja programa i broja verzije. Nakon toga može da poziva udaljene procedure.

**Server program** server.c sadrži implementaciju odgovarajućih procedura. Procedure u serveru imaju ime imeprocedure\_brojverzije\_svc. Procedure u serveru uzimaju kao parametar pointer na podatak koji se prenosi, vraćaju pointer na rezultat. Ako se prenosi više parametara ili rezultata onda se to postiže preko struktura.

#### **Koraci kod poziva RPC-a:**

1. Klijent poziva lokalnu proceduru (klijent stub) i parametri se smeštaju u stek.
2. Klijent stub pakuje argumente za udaljenu proceduru i gradi jednu ili više mrežnih poruka i poziva lokalni OS (poziva SEND primitivu, a nakon toga RECEIVE primitivu i čeka dok ne dobije odgovor od servera).
3. Lokalni OS šalje poruku udaljenom OS-u pomoću transportnog protokola UDP ili TCP.
4. Mrežna poruka stiže do odredišta (servera), OS prosleđuje poruku serverskom stub-u i on prima poruku od svog lokalnog OS-a (izvršava RECEIVE primitivu), raspakuje je i izvlači argumente za poziv lokalne procedure.
5. Serverski stub poziva željenu serversku proceduru i predaje joj parametre koje je primio od klijenta tako što ih stavlja u stek.
6. Server izvršava proceduru i vraća rezultat serverskom stub-u.
7. Serverski stub pakuje rezultat u poruku i poziva lokalni OS (poziva SEND primitivu, pa RECEIVE da čeka na sledeće pozive klijenta)
8. Serverski OS šalje poruku klijentskom OS-u
9. Lokalni OS prosleđuje poruku klijent stub-u
10. Klijent stub izvlači rezultat iz poruke i vraća klijentskom procesu

Sve udaljene procedure definišu se kao deo udaljenog programa. Imena 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 nadodavanjem imena procedure iza kojeg sledi donja crtica i broj verzije [*imeprocedure\_brojverzije*]. Ove procedure u server programu imaju ime [*imeprocedure\_brojverzije\_svc*].

**Primer KALK.** Servis KALK za sabiranje i oduzimanje dva cela broja, ima dve procedure: SABERI(x, y) i ODUZMI(x, y). Pošto SunRPC poziv udaljene procedure može imati samo jedan argument, to ćemo uraditi na sledeći način:

```
struct operandi
{
    int x;
    int y;
}
```

```
program KALK
{
    version KALK_VERSION
    {
```

```

    int SABERI(operandi) = 1;    brojevi procedura
    int ODUZMI(operandi) = 2;
} = 1    broj verzije
} = 0x29999999    broj programa (32bit)

```

Svaka RPC procedura je definisana u okviru nekog programa i identifikovana je pomoću:

- Broja programa – HEX broj koji identifikuje grupu udaljenih procedura, od kojih svaka ima svoj broj
- Broja verzije – ako se napravi izmena u udaljenom servisu, programu se dodaje novi broj verzije
- Broja procedure – numeracija kreće od 1

```

program IME_PROGRAMA
{
    def_verzije = broj_proc1;
    def_verzije = broj_proc2;
} = 0xYYYYYYYY - 32 bit identifikator

```

```

version IME_VERZIJE
{
    def_procedure1
    def_procedure2
    ...
} = konstanta

```

```

definicija procedure
    tip ime_procedure(tip_argumenta) = konstanta;

```

#### Klijent strana:

```

#include <stdio.h>
#include <rpc/rpc.h> //standardna biblioteka za rpc
#include "primer.h" //fajl generisan pomovu rpcgen
int main(int argc, char *argv[])
{
    CLIENT *cln;
    char *server;
    int *zbir, *razlika; //povratne vrednosti iz udaljenih procedura
    operandi op; //struktura definisana u IDL fajlu
    if(argc<2)
    {
        exit(1);
    }
    server = argv[1];
    cln = clnt_create(server, KALK, KALK_VERSION, "udp");
    //kreira se socket i klijent handle i povezuje sa serverom
    if(cln==NULL)

```

```

{
    //konekcija se serverom nije uspostavljena
    exit(2);
}
op.x = atoi(argv[2]);
op.y = atoi(argv[3]);

zbir = saberi_1(&op, cln); //poziv udaljene procedure - poziva se stub fja
if(zbir == (int*)NULL)
{
    exit(3);
}
printf("suma je %d\n", *zbir);
razlika = oduzmi_1(&op, cln);
//poziv udaljene procedure, argument procedure je ptr na argument ciji je tip definisan u IDL
fajlu. Procedura mora da vrati ptr na rezultat
if(razlika == (int*)NULL)
{
    exit(4);
}
printf("razlika je %d\n", *razlika);
clnt_destroy(cln);
exit(0);
}

```

Pomoću rpcgen može se generisati template za serverski kod korišćenjem prethodno definisanog interfejsa: ***rpcgen -C -Ss primer.x > server.c***. Ime fajla u kome je zapamćen serverski kod je server-c. Sada rpcgen generiše sledeći kod:

```

#include primer.h
int *saber_1_svc(operandsi *argp, struct svc_req *rqstp)
{
    static int result;
    //serverski kod
    return &result;
}
int *oduzmi_1_svc(operandsi *argp, struct svc_req *rqstp)
{
    static int result;
    //serverski kod
    return &result;
}

```

Mi možemo da editujemo generisani kod za serversku stranu na sledeći način:

```

int *saber_1_svc(operandsi *a, struct svc_req *rqstp)
{
    static int zbir;
    zbir = a->x + a->y;
    return &zbir;
}

```

```

}

int *oduzmi_1_svc(operandsi *a, struct svc_req *rqstp)
{
    static int razlika;
    razlika = a->x - a->y;
    return &razlika;
}

```

**Kako izgleda poziv Sun RPC kompajlera? Koji fajlovi se dobijaju kao rezultat ovog kompajliranja i šta sadrže? Objasniti na primeru.**

Poziv kompajlera je rpcgen -C primer.x. fajlovi koji se dobijaju kao rezultat kompajliranja su:

- primer.h – header fajl
- primer\_clnt.c – klijent stub
- primer\_svc.c – server stub

Header fajl sadrži id interfejsa, definiciju tipova, konstanti i prototipova funkcija. On se sa include uključuje u klijent i server kod.

Klijent stub sadrži procedure tj. interfejse udaljenih procedura koje će klijent program da pozove. Zadužene su za pakovanje parametara u poruke, slanje poruke, prijem poruke, izvlačenje rezultata iz primljene poruke, prosleđivanje rezultata klijentu.

Server stub sadrži procedure koje se pozivaju sa udaljenog klijenta. Prima poruke koje stignu od klijenta, raspakuje parametre, prosleđuje ih serveru koji izvršava potreban kod na serveru, pakuje rezultat u poruku i šalje poruku ka klijent stubu.

Primer neka bude fact(int n):

Definicija programa:

```

program PRIMER_PROG
{
    version PRIMER_VERSION
    {
        int FACT(int) = 1;
    }=1;
}=0x31234567;

```

Ono što će sam kompajler da generiše na server strani je:

```

int* fact_1_svc(int *argp, struct svc*rgstp)
{
    static int result;
    //serverski kod
    return &result;
}

```

```
}
```

A mi modifikujemo kod da dobijemo ono što nam zapravo treba:

```
int* fact_1_svc(int *argp, struct svc*rgstp)
{
    static int result=1;
    int i;
    for(i=*argpli>0;i++)
    {
        result = result * i;
    }
    return &result;
}
```

Koje su prednosti SunRPC-a?

- aplikacija ne mora da vodi računa o dobijanju jedinstvene transportne adrese (broja porta)
- aplikacija ne mora da vodi računa o veličini poruke, fragmentaciji, reasembliranju
- aplikacija treba da zna samo jednu transportnu adresu – adresu **portmappera**

## DCE – Distribuirano računarsko okruženje (Distributet Computing Environment)

Middleware baziran na RPC-u koji predstavlja skup servisa i alata koji se može instalirati na vrhu postojećeg OS-a, koji služi kao platforma za gradnju distribuiranih aplikacija.

DCE je prvi middleware projektovan kao nivo apstrakcije između mrežnog OS i distribuirane aplikacije.

Programski model na kome se bazira DCE je klijent-server arhitektura. Sva komunikacija između klijenta i servera obavlja se pomoću RPC-a.

### Directory servis ili Name servis

Servis koji na osnovu imena objekta vraća informaciju o imenovanom. Ovaj servis se koristi da bi locirao server na kome je implementirana udaljena procedura. **DCE Cell Directory Servis (CDS)** je mehanizam koji omogućava korišćenje logičkih imena unutar **DCE ćelije** – grupa klijent-server mašina u okviru jednog LAN-a. CDS služi da locira CDS server u kome se nalazi preslikavanje logičkog imena u IP adresu servera na kome je implementirana potrebna udaljena procedura.

Dakle, za razliku od Sun RPC-a aplikacije identifikuju resurse po imenu, bez potrebe da znaju gde je resurs lociran.

DCE ćelije mogu međusobno komunicirati da bi locirali resurs koji je van lokalne DCE ćelije. Global Directory Servis (GDS) upravlja logičkim imenima van lokalne DCE ćelije. Sprega između CDS i GDS je GDA.

Definicija interfejsa (u IDL-u) je lepak koji sve drži na okupu u klijent-server sistemu baziranom na RPC-u. ključni element u svakom IDL fajlu je globalni id interfejsa, a pored toga IDL omogućava deklaraciju procedura, tipova, konstanti, itd.

### Pisanje klijent/server aplikacije:

1. Poziv uuidgen programa koji omogućava da se generiše protoisp IDL fajla koji će sadržati id interfejsa i garantuje da se isti id neće nigde pojaviti u DS-u generisanom sa uuidgen.  
**uuidgen imefajla.idl** → kreira se fajl **imefajla.idl** koji sadrži dobijen id i verziju i kreira se interfejs:  
  

```
interface INTERFACENAME
{
}
```
2. Sledeći korak je editovanje IDL fajla i popunjavanje imena udaljenih procedura i njihovih parametara. Menjamo INTERFACENAME sa imenom interfejsa, deklariramo procedure u okviru interfejsa – samo navodimo potpis procedure.

```
interface MATH
{
    long get_sum([in] long first, [in] long second);
}
```

Kada je IDL fajl potpun poziva se **IDL kompajler** sa **idl primer.idl**. Izlaz iz IDL kompajlera sastoji se od 3 fajla:

- **Header fajl** – sadrži id, definiciju tipova, konstanti i prototipova procedura.
  - **Klijent stub** – sadrži procedure tj. interfejse udaljenih procedura koje će klijent program pozivati. Ove procedure su zadužene za pakovanje parametara u poruke, slanje poruke ka serveru, prijem poruka sa servera, izvlačenje rezultata iz primljene poruke i slanje rezultata klijentu.
  - **Server stub** – sadrži procedure koje se pozivaju kada stigne poruka sa klijenta i koje pozivaju odgovarajuće procedure na serveru – to su udaljene procedure koje klijent zapravo poziva.
3. Sledeći korak je pisanje klijent i server programa, a zatim kompajliranje ovih programa i klijent i server stub-a i povezivanje sa odgovarajućim stub-om.

U opštem slučaju, DCE RPC programer piše 3 programa: *klijent kod, server kod kojim se server registruje u Directory servis, server operativni kod.*

Da bi klijent mogao da pozove server, server mora da bude registrovan i spreman da primi poziv. Registracija se obavlja pre nego što klijent poziva server. DCE klijent pronalazi server u 2 koraka:

- Lociranje serverske mašine
- Lociranje odgovarajućeg procesa na serverskoj mašini

Da bi klijent mogao da komunicira sa serverom on mora da zna broj porta na serverskoj mašini na koji može da šalje poruke. U DCE-u na svakoj serverskoj mašini, postoji tabela sa parovima [server, brojPorta] koju održava DCE daemon proces. Server se registruje u Directory serveru tako što dostavlja adresu serverske mašine i svoje interfejsa.

Povezivanje klijenta i servera

1. Registrovanje broja porta servera u DCE daemonu
2. Registrovanje servera u Directory serveru (adresa serverske mašine, ime servera i interfejsi koje implementira)
3. Klijent kontaktira Directory server da dobije IP adresu server mašine na kojoj se izvršava željeni server proces
4. Klijent kontaktira DCE daemon da bi dobio broj porta server procesa kome želi da pristupi
5. Poziv udaljene procedure

## Objektno-orijentisan pristup – poziv udaljenih metoda – RMI

To je RPC na OO način i omogućava pozivanje metoda objekata koji se nalaze na udaljenom računaru. *Najbitnije je da objekat skriva svoju unutrašnjost (podatke i metode) od spoljašnjeg okruženja uz pomoć dobro definisanog interfejsa.* Ovo je ključna osobina za krivanje heterogenosti u DS. Objekat može imati skup metoda koje se mogu pozivati sa udaljenih računara. Ovi metodi definisani su u REMOTE interfejs fajlu, sa tim da objekat može da ima i metode koje se pozivaju samo lokalno. Metodama koje su namenjene za udaljeni poziv može se pristupiti samo isključivo preko interfejsa. Interfejs sadrži samo definicije metoda, a server ima implementacije tih metoda. Zapravo, objekat nije distribuiran, već interfejs!

### Šta se dešava kada jedna mašina pozove metod druge mašine?

- **Proxy** je predstavnik server objekta u adresnom prostoru klijenta. Proxy implementira isti interfejs kao i server i proxy je analogan klijent stub-u. Proxy pakuje parametre u poruku i prosleđuje poruku serveru (poruka sadrži **referencu udaljenog objekta, id metode i parametre poziva**).
- Objekat se nalazi na serveru na kome se nalazi isti interfejs kao i na klijent strani
- Dolazni poziv se prvo prosleđuje serverskom stub-u (skeletonu), koji raspakuje poruku i poziva odgovarajući metod na serveru.

Kada klijent pozove metod udaljenog objekta, poziv se prosleđuje serverskom procesu u kome se nalazi udaljeni objekat. Ova poruka mora da specificira objekat čiji se metod poziva i to radi pomoću jedinstvenog id-a udaljenog objekta tj. preko reference udaljenog objekta (**remote object reference**). Referenca udaljenog objekta mora da garantuje svoju jedinstvenost u DS-u, pa se ona gradi ili kao:

[IP adresa računara + broj porta procesa koji je kreirao objekat + vreme kreiranja + lokalni broj objekta]

Ili može sadržati informacije o interfejsu udaljenog objekta kao što je njegovo ime.

### Kako klijent doznaje referencu udaljenog objekta?

Koristi se poseban distribuirani servis – **BINDER**. U binder-u postoji tabela sa preslikavanjem tekstualnog imena u referencu udaljenog objekta [imeObjekta, referencaObjekta]. Server registruje svoje udaljene objekte u binder-u i klijent koristi binder da pronade odgovarajuću referencu udaljenog objekta.

## Tipovi komunikacija

- 1) **Perzistentne (trajne)** – poruka se pamti u komunikacionom serveru koliko je potrebno da bi se isporučila odredištu
- 2) **Tranzijentne** – poruka se odbacuje ako komunikacioni server nije u stanju da je isporuči sledećem serveru ili odredištu
- 1) **Sinhrono** – pošiljalac se blokira dok se poruka ne zapamti u lokalnom baferu odredišnog hosta
- 2) **Asinhrono** – pošiljalac nastavlja sa radom odmah nakon što prosledi poruku za slanje

### Perzistentne komunikacije

Poruke koje se prenose se pamte u komunikacionim serverima sve dok se ne proslede prijemnom komunikacionom serveru. Izvor ne mora da bude aktivan nakon što postavi poruku komunikacionom serveru. Prijemnik ne mora da se izvršava u trenutku kada izvor šalje poruku.

Primer je e-mail sistem:

- na hostu se izvršava korisnički agent – mogu se kreirati, slati i primati poruke.
- Svaki host je povezan na tačno 1 mail server – komunikacioni server
- Kada korisnički agent prosledi poruku za slanje svom hostu, host je prosledi ka svom lokalnom mail serveru da je privremeno skladišti
- Mail server obradi poruku, pronađe adresu odredišnog mail servera i uspostavi vezu sa njim
- Odredišni server pamti primljenu poruku
- Ako je odredišni mail server nedostupan, lokalni mail server će u svom baferu nastaviti da čuva poruku
- Korisnički agent pristupa lokalnim mail serveru i kopira poruke iz mail box-a u lokalni bafer

### Tranzijentne komunikacije

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 da se odbaci. Ako se agenti ne izvršavaju, poruke se odbacuju. Komunikacioni server u ovom slučaju radi kao store-and-forward ruter.

### Perzistentne asinhrono

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.

### Perzistentne sinhrono

Poruka mora biti zapamćena u odredišnom hostu da bi se pošiljalac deblokirao.

### Tranzijentne asinhrono

Poruka se privremeno pamti u lokalnom baferu izvornog hosta, nakon čega aplikacija nastavlja sa izvršenjem.

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

RPC i RMI su sinhroni i klijent se blokira dok njegov zahtev ne bude obrađen. Nekada takav vid komunikacije nije dobar. Rešenje za to nude distribuirani sistemi orijentisani na razmeni poruka.



## MPI – Message Passing Interface

Middleware sistem koji **se bazira na prenosu poruka**. Namnjen je za podršku komunikacijama u paralelnim sistemima. Podržava **tranzijentne** komunikacije između procesa. Podržava komunikaciju između konkurentnih procesa i nudi **point-to-point** i **grupnu komunikaciju**. Implementiran je kao biblioteka funkcija, pa se može pozivati u okviru programskih jezika. Podrazumeva da se komunikacija obavlja u okviru neke grupe procesa, pa se svakoj grupi dodeljuje id, a svaki proces u grupi ima svoj jedinstven lokalni (na nivou te grupe) id. Par [id\_grupe, id:procesa] na jedinstven način identifikuje izvor/odredište poruke. U sistemu može da postoji više grupa procesa koje obavljaju izračunavanje i koje se izvršavaju u isto vreme.

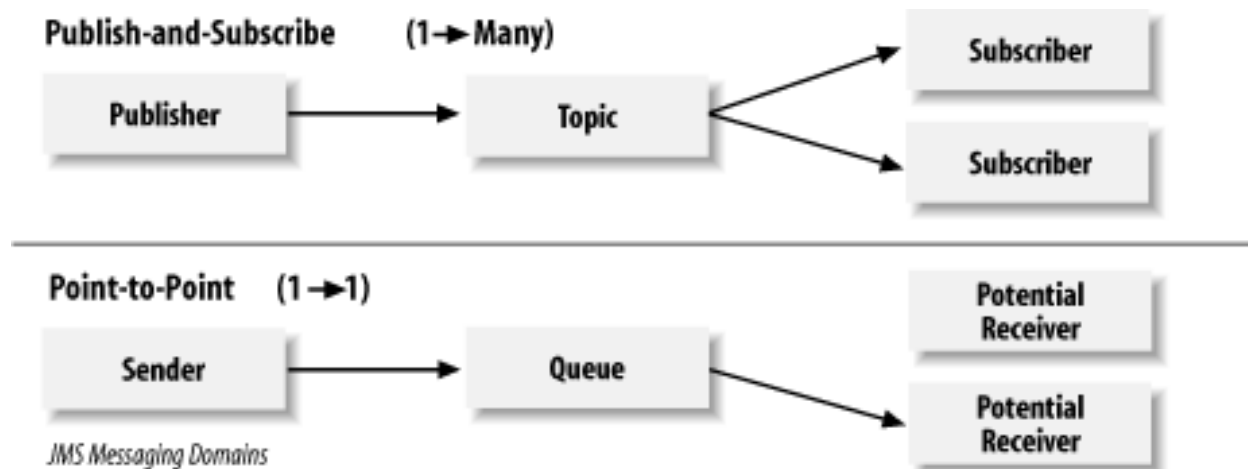
## Sistemi sa redovima poruka (QUEUE)

Middleware sistemi zasnovani na slanju poruka sa podrškom za perzistentne asinhronne komunikacije. Namereni su za aplikacije gde je dozvoljeno da prenos poruka traje i nekoliko minuta. Ne zahtevaju ni od izvora ni od odredišta da budu aktivni za vreme prenosa poruka.

Aplikacije komuniciraju stavljanjem poruka u redove (queue). Poruke

## Publish-Subscribe sistemi za razmenu poruka

Umesto pošiljaoca i primaoca imamo Publisher i Subscriber. Poruku može da primi više Subscriber-a. publisher kreira poruku i publish-uje je u okviru Topic-a (teme). Umesto QUEUE koristi se TOPIC. Više različitih Subscriber-a se može prijaviti za Topic i koristiti poruke koje su objavljene u tom Topic-u (ali samo one poruke koje su poslate nakon što se prijavio, ne i pre toga).



## Sinhronizacija

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

Procesi u DS-u pristupaju zajedničkim resursima pa je neophodno da se obezbedi da se pristup tim resursima obavlja isključivo od strane različitih procesa da bi resurs ostao u konzistentnom stanju.

Problem sinhronizacije u DS-u je težak problem. **Ne postoji pojam globalnog vremena, već svaka mašina ima svoj časovnik.** Proces koji se izvršavaju na različitim računarima, imaju različitu predstavu o vremenu, što dovodi do problema. DS treba da obezbedi striktnu sinhronizaciju fizičkih časovnika, ako se radi o aplikaciji u realnom vremenu. Nekada je potrebno uskalditi fizičke časovnike, a **nekada je dovoljno da svi časovnici imaju istu predstavu o relativnom vremenu** što zahteva sinhronizaciju logičkih časovnika (što je mnogo lakše od fizičkih).

## Sinhronizacija – Fizički časovnici

Sve metode sinhronizacije fizičkih časovnika svode se na razmenu vrednosti časovnika između računara. Tu je problem **komunikaciono kašnjenje**. Tokom vremena časovnici opet izađu iz sinhronizacije, pa se opet pokreće sinhronizacija. Poznavanje apsolutnog vremena nije neophodno u svakoj situaciji, **već je potrebno postići da se uzajamno zavisni događaji odvijaju u korektnom redosledu.**

### Timer

Svi računari imaju kolo koje beleži vreme – **TAJMER**. **Tajmer je kvarcni kristal** koji kad je pod naponom osciluje na stabilnoj frekvenciji, a brzina oscilacije zavisi od vrste kristala, načina sečenja kristala i veličine napona. Svakom tajmeru dodata su **2 registra: brojač i holding registar**. **Holding registar pamti inicijalnu vrednost za brojač**. Svaka oscilacija kristala dekrementira sadržaj brojača. Onog trenutka kada brojač postane 0 generiše se prekid i vrši se inkrementiranje lokalnog časovnika za 1 vremensku jedinicu. Onda se brojač opet postavlja na vrednost iz holding registra. Svaki prekid koji se generiše je jedan otkucaj časovnika.

Ako sistem ima N računara, svih N kristala u njima će oscilovati sa neznatno različitim brzinama. To će izazvati da softverski časovnici postepeno ispadnu iz sinhronizma i daju različite vrednost pri očitavanju vremena što dovodi do grešaka u radu sistema. 😞

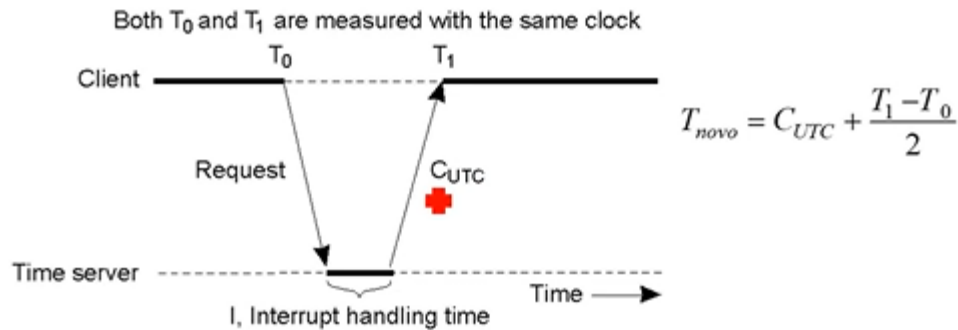
### GPS – Sistem za globalno pozicioniranje

Pomaže da se precizno odrede koordinate objekta na Zemlji. Neophodno je da se zna položaj izvora tačnog vremena, kao i objekta koji traži tačno vreme. Taj problem se rešava pomoću posebnog DS-a koji se zove GPS. Sastoji se od 29 satelita koji kruže u zemljinoj orbiti i svaki satelit ima 4 atomska časovnika koji se regularno kalibrišu sa specijalne stanice na Zemlji. Da bi se odredio položaj objekata na Zemlji neophodna su bar 4 satelita.

### Algoritmi za sinhronizaciju fizičkih časovnika u DS

1. **Kristijanov algoritam** – zahteva postojanje eksternog časovnika. Postoji jedna mašina koja je opremljena WWW prijemnikom – **server tačnog vremena** – koji prima signale sa radio stanice o UTC vremenu. *Cilj je da se sve ostale mašine sinhronizuju sa tom mašinom jer je to izvor tačnog vremena.* Svaki računar šalje poruku **serveru tačnog vremena** (koji ima **WWW prijemnik** koji

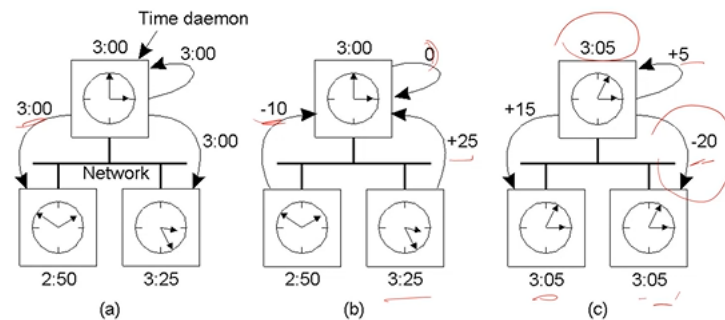
dobija tačno vreme) i traži od njega informaciju o tačnom vremenu. Server tačnog vremena je **pasivan**, samo daje info o tačnom vremenu.



Da bi klijent uskladio svoj časovnik on mora da uzme u obzir **propagaciono kašnjenje** između klijenta i servera tačnog vremena i to određuje razliku  $(T_1 - T_0)/2$ , gde je  $T_0$  trenutak kada je klijent uputio zahtev, a  $T_1$  trenutak kada je klijent primio odgovor od servera.

Da li da klijent sada postavi vrednost svog časovnika na dobijenu vrednost ili da to radi postepeno? **Vreme nikad ne sme da ide unazad!** Ako klijent dobije vreme **manje** od svog trenutnog onda se promena vremena vrši **postepeno**. Holding registar časovnika se povećava, tako da se praktično časovnik klijenta uspori dok ne dostigne željenu vrednost tj. onu koju je dobio od servera. Nema vremeplova 😊

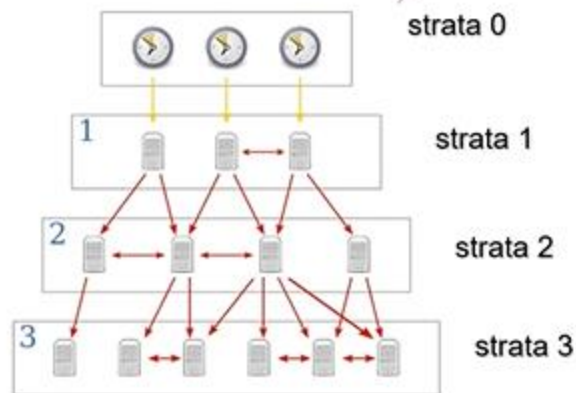
2. **Berklijev algoritam** – obezbeđuje internu sinhronizaciju mašina unutar DS-a bez potrebe za eksternim izvorom tačnog vremena. Imamo **server vremena** – proces koji se izvršava na jednom od računara – obavlja periodičnu prozivku svake mašine u DS-u da bi doznao koliko vremena da svakoj mašini. Na osnovu odgovora server izračunava srednje vreme i saopštava svim mašinama kako da podese svoje časovnike. Ovaj metod ne zahteva izvor tačnog vremena, već da se u okviru jednog DS-a da se usklade časovnici. To je sasvim zadovoljavajuće jer je bitno da računari jednog DS-a imaju istu predstavu o vremenu, ne mora to vreme da bude baš globalno tačno, bitno da je isto za sve u okviru DS-a.



- a) U 3:00 time daemon 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 podese svoje časovnike.

**Kristijanov algoritam i Berkley algoritam su prvenstveno projektovani da omoguće sinhronizaciju u privatnim mrežama.**

3. **NTP (Network Time protocol)** – Klijent-server protokol aplikativnog nivoa za sinhronizaciju u **Internetu**. Koristi mnogo različitih servera za sinhronizaciju časovnika u klijentima sa UTC vremenom. *Serveri su izvori informacija o tačnom vremenu. Klijent je mašina koja želi da uskladi svoje vreme sa tačnim vremenom.* To je protokol aplikativnog nivoa, a na transportnom koristi UDP protokol i osluškuje klijentske pozive na portu 123. Serveri vremena se hijerarhijski organizuju u slojeve tj. **stratume**. Numeracija kreće od 0 i na nivou 0 su **izvori tačnog vremena** – atomski časovnici. Na nivou 1 su serveri koji su **direktno** povezani na izvor tačnog vremena i oni



žute linije označavaju direktnu vezu  
crvene linije označavaju mrežnu vezu

imaju najeću preciznost. Na nivou 2 su serveri direktno mrežno povezani sa serverima nivoa 1, a mogu biti i međusobno povezani sa serverima na nivou 2 (što važi za sve nivoe ispod). Svi serveri zajedno čine sinhronizacionu mrežu:

Ovaj protokol definisan je standardom RFC 5905 i podržava 3 režima sinhronizacije časovnika:

- **Simetrični režim** – najpreciznija sinhronizacija i koristi se za sinhronizaciju *master* servera.
- **Klijent-server (RPC) režim** – slično kao Kristijanov algoritam. Klijent kontaktira server tačnog vremena i dobija info o tačnom vremenu pa to koristi da podesi svoj časovnik.
- **Multicast režim** – u brzim LAN mrežama. NTP server periodično emituje vreme koje ostali hostovi kroiste za sinhronizaciju.

Svaki protokol ima format poruke, kao i NTP protokol. Polja su sledeća:

- **LI** – 2 bit – da li će zadnji sat toga dana imati prestupnu sekundu (one se ubacuju 30.juna ili 31. Decembra)
- **VN** – 3 bit – verzija protokola
- **Mode** – 3 bit – režim rada (simetrični/klijent-server/broadcast)
- **Stratum** – 8 bit – na kom nivou (stratumu) je NTP server koji vraća odgovor klijentu
- **Poll** – 8bit – max interval između 2 uzastopne prozivke izražen u log sa osnovom 2
- **Precision** – 8 bit – preciznost časovnika u log sa osnovom 2 sekundi
- **Root delay** – 32 bit – koliko je kružno vreme propagacije do referentnog servera
- **Reference Id** – 32bit – identifikator servera koji se koristi kao referentni časovnik
- **Reference timestamp** – 32 bit – vreme kada je sistemski časovnik poslednji put bio postavljen/korigovan

- **Origin timestamp** – kada je poruka upućena od klijenta ka serveru
- **Receive timestamp** – kada je poruka stigla od klijenta na server
- **Transmit timestamp** – kada server šalje odgovor klijentu

**Klijent-server režim** - Klijent kontaktira NTP server da bi sinhronizovao svoj časovnik i u poruci navodi svoje lokalno vreme. U trenutku  $T_0$  klijent šalje zahtev serveru koji je obeležen *lokalnim vremenom klijentovog časovnika* – 231ms. Server beleži prijem zahteva u trenutku  $T_1=135$ ms, obrađuje zahtev i u trenutku  $T_2=137$ ms šalje odgovor klijentu o tačnom vremenu. Klijent u  $T_3=298$ ms beleži prijem poruke sa tačnim vremenom od servera.

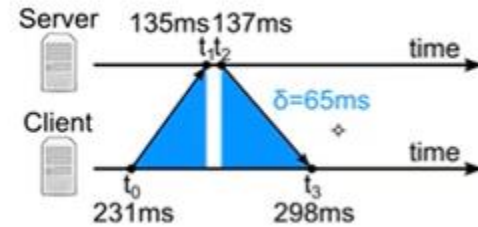
$\delta = (t_1 - t_0) + (t_3 - t_2)$  *kružno vreme propagacije*

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

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

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

*Tačno vreme:*  $t_3' = t_3 + \theta$



# Sinhronizacija – Logički časovnici

U mnogim programima nije bitno fizičko vreme nego redosled nekih događaja. Dakle, dovoljno je da se mašine koje ostvaruju međusobnu interakciju usaglasе oko relativnog sleda događaja. Relativno uređenje događaja bazira se na **logičkim časovnicima** tj. **Lamportove vremenske markice**. Fizički časovnici u DS-u vrlo brzo izlaze iz sinhronizma, ako je neophodno imati sinhronizaciju fizičkih časovnika, neophodna je neki od navedenih algoritama, što može biti skupo! Na sreću, nekada je dovoljna sinhronizacija logičkih časovnika (redosled događaja bude ispoštovan, nebitno je tačno vreme i sinhronizacija samog vremena).

## Lamportov algoritam sinhronizacije

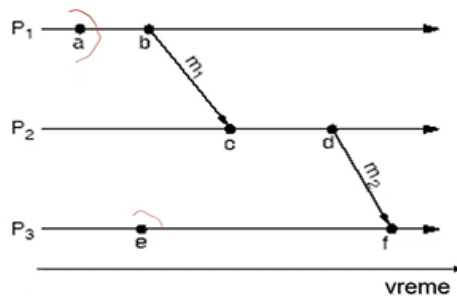
**Određuje redosled događaja, ali ne vrši sinhronizaciju časovnika.** Koristi se za sinhronizaciju logičkih časovnika. Definisana je relacija:

$$A \longrightarrow B \text{ čita se "A se desilo pre B"}$$

Što znači da je **događaj A nastupio pre događaja B**. Ova relacija može biti zadovoljena u 2 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 relacija  $a \rightarrow b$  tačna.

Ova operacija je **tranzitivna**! Ako važi  $a \rightarrow b$  i  $b \rightarrow c$  tada važi  $a \rightarrow c$ .



$P1, P2, P3$ : processes;  
 $a, b, c, d, e, f$ : events;

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

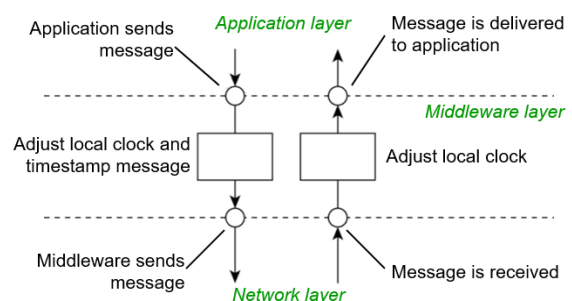
Ako su se događaji x i y desili u 2 procesa koji ne razmenjuju poruke (čak ni posredno) tada relacija  $x \rightarrow y$  ili  $y \rightarrow x$  ne važi, a za događaje x i y kaže se da su **konkurentni**.

Za događaje a i f možemo da tvrdimo da se a desilo pre f, dok za događaje a i e ne možemo to da tvrdimo.

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

## Gde se vrši sinhronizacija logičkih časovnika u DS-u?

Sinhronizacija logičkih časovnika (koja nam omogućava da utvrdimo koji događaj je nastupio pre ili posle nekog drugog događaja) odvija se u MIDDLEWARE sloju. Kod slanja poruke, middleware sloj beleži događaj i šalje poruku sa vrednošću logičkog časovnika tj. **Lamportove vremenske markice**. Kada se primi poruka, middleware koristi vremensku markicu te poruke da sinhronizuje lokalni časovnik pre nego što se poruka isporuči aplikaciji.



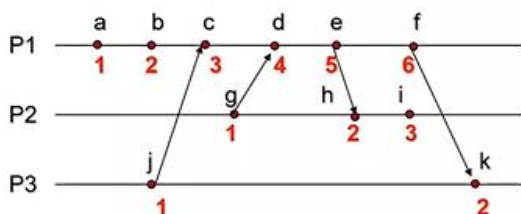
Ako je relacija  $a \rightarrow b$  tačna, tada je potrebno ostvariti sinhronizaciju logičkih časovnika tako da važi da je  $T(a) < T(b)$ .

$T(a)$  je vrednost logičkog časovnika za događaj  $a$ , tj. to je **vremenska markica događaja  $a$** . Dakle, **vremenska markica događaja  $a$  mora da bude manja od vremenske markice događaja  $b$** . Svakom događaju se dodeljuje vremenska markica i između svaka dva događaja sat mora da otkuca bar jednom. Sinhronizacija se ostvaruje korekcijom logičkih časovnika tako što se logičkom časovniku uvek dodaje neka vrednost, jer vreme ne može da ide unazad (nema vremepolova).

#### Pravila:

1. Neka je  $L_i$  logički časovnik (vremenska markica) u procesu  $P_i$ . Taj časovnik se **inkrementira** pre izvršenja bilo koje aktivnosti (slanje poruke kroz mrežu, isporuka poruke aplikaciji, itd) u procesu  $P_i$ .
2. Kada proces  $P_i$  šalje poruku  $m$  nekom drugom procesu  $P_j$ , on u nju ubacuje i vremensku markicu  $t = L_i$ .
3. Kada proces  $P_j$  primi poruku  $m$  on podešava svoj lokalni logički časovnik (vremensku markicu) tako što određuje  $L_j = \max(L_j, t)$  i primenjuje pravilo 1 (inkrementira svoj logički časovnik). Zatim prosleđuje poruku aplikaciji.

- \* Tri procesa  $P_1, P_2$  i  $P_3$  na tri mašine
- \* Događaji  $a, b, c, \dots$
- \* Lokalni brojač događaja na svakoj mašini
- \* Procesi povremeno razmenjuju poruke.



Ako posmatramo događaje  $e$  i  $h$ , vidimo da ako je  $e \rightarrow h$  ne znači da je  $T(e) < T(h)$ ! Dakle, nije ispoštovano pravilo! Prosto  $T(e)$  mora biti **MANJE** od  $T(h)$ ! Isto važi i za  $f$  i  $k$ .

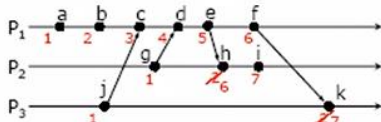
Postoji **rešenje**:

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.



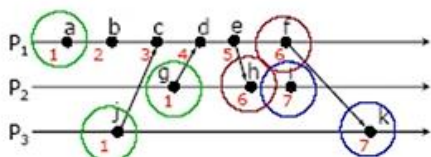
$e \rightarrow h, T(e) < T(h)$   
 $f \rightarrow k, T(f) < T(k)$

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



## Problemi sa Lamportovim algoritmom

Moguće je da se više konkurentnih događaja (nisu međusobno uslovljeni) imaju iste vremenske markice, što nije poželjno!



Ovo dovodi do konfuzije ako više procesa treba da donese odluku na osnovu vremenskih markica dva događaja.

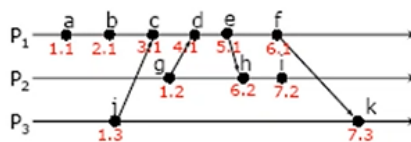
Postoji rešenje za ovo: Prisiliti da svaka markica bude jedinstvena. **Lamportovoj vremenskoj markici dodati još jedan identifikator koji predstavlja globalno jedinstveni id procesa u kome je nastao događaj (adresa hosta + proces ID).**

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

ako i samo ako

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

$$T_i = T_j, i < j$$



## Primer – potpuno uređena grupna komunikacija

Potpuno uređena grupna komunikacija odnosi se na komunikaciju u kojoj sve replike treba da prime isti skup poruka u istom redosledu tj. da sve replike budu u konzistentnom stanju. Komunikacija u kojoj se akcije obavljaju u istom redosledu. **Obezbediti potpuno uređenu grupnu komunikaciju znači obezbediti operaciju kojom se sve poruke isporučuju u istom redosledu svim prijemnicima.**

Replicirani bankovni računi u New York-u i San Francisco-u. Dve transakcije dešavaju se jednovremeno i obavlja se multicast. Tekuće stanje računa je 1000\$, u SF dodaje se 100\$, a u NY u isto vreme dodaje se 1% kamate. Nakon ažuriranja NY ima vrednost 1110\$, a SF 1111\$, jer u NY se prvo radi kamata od 1%, pa se doda 100\$, a u SF obrnuto. To ne sme da se desi!

**Mora da se obezbedi da se operacije ažuriranja obave u istom redosledu!**

### Algoritam:

- Poruka ažuriranja se obeležava logičkim vremenom izvora (proširene markice)
- Poruka ažuriranja se prosleđuje svima u grupi (pa i sebi)
- Kada poruka stigne:
  - Smešta se u lokalni queue čekanja
  - Poruke koje su prispele uređuju se po markicama
  - Poruka potvrde ACK se prosleđuje svima (pa i sebi)
- Poruke iz istog izvora u redosledu u kome su poslate (FIFO)
- Poruka se prosleđuje aplikaciji samo ako se nalazi na vrhu reda ili je potvrđena od strane svih procesa.
- Pi šalje potvrdu ACK za poruku koju je primio od Pj ako:
  - Pi nije poslao poruku ažuriranja
  - Pi id je veći od Pj id ( $i > j$ )
  - Pi zahtev za ažuriranjem je već obrađen



Neka je poruka **m** = „dodaj 100\$“, a poruka **n** = „dodaj kamatu od 1%“. Svaka poruka kada se pošalje sastoji se od same poruke i vremenske markice kada je zahtev za ažuriranje izdat. Za poruku **m** to je trenutak **1.1** (neka bude da je SF proces **P1**), a za poruku **n** to je **1.2** (NY je proces **P2**).

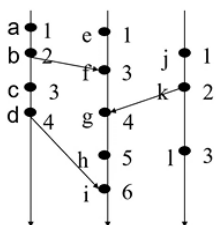
- Poruke se proslede svim procesima u grupi (uključujući i samog sebe)
- Nakon slanja poruka redovi čekanja u procesima sadrže:
  - P1: (m, 1.1), (n, 1.2)
  - P2: (n, 1.2), (m, 1.1)
- P1 će proslediti svima u grupi potvrdu ACK za poruku (m, 1.1), ali ne i za (n, 1.2)! Jer je identifikator procesa P1 niži od identifikatora P2 ( $1.1 < 1.2$ ). Kada P1 završi ažuriranje za m, on šalje svim procesima u grupi potvrdu za (n, 1.2).
- P2 će proslediti svima u grupi ACK i za (m, 1.1) i za (n, 1.2) jer je id P2 viši od id P1 ( $1.1 < 1.2$ ).

Šta da je postojao treći proces P3 koji je izdao zahtev za ažuriranjem (k, 1.3) skoro u isto vreme kad i procesi P1 i P2?

- P1 neće proslediti ostalim procesima u grupi potvrdu za k, dok se m ne obavi
- P2 neće proslediti ostalim procesima u grupi potvrdu za k, dok se n ne obavi
- P3 će proslediti ostalim procesima u grupi potvrdu za k
- Pošto operacija ne može da se obavi dok iz svih procesa ne stignu potvrde, operacija ažuriranja k se neće obaviti dok se operacije m i n ne obave!

## Sinhronizacija - Vektorski časovnici

Problem sa Lamportovim markicama je što se pomoću njih ne može utvrditi koji su događaji **međusobno uslovljeni**, a koji su **konkurentni**! Ako imamo dva događaja i znamo samo njihove vremenske markice i važi recimo  $T(a) < T(b)$  mi ne možemo da tvrdimo da se a desilo pre b, tj. ne možemo da tvrdimo da su događaji a i b međusobno uslovljeni!



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 događaja takav da ako važi da je npr.  $T(a) < T(b)$ , tada možemo da zaključimo da  $a \rightarrow b$

**Vektorski časovnik** je vektor koji ima onoliko celobrojnih elemenata koliko ima procesa u sistemu. Svaki proces ima svoj sopstven vektorski časovnik kojim beleži lokalne događaje. Kao i Lamportove markice, i vektorski časovnik se šalje sa svakom porukom.

### Pravila:

1. Vektor se inicijalizuje na 0 u svim procesima  $V[i][j] = 0$ , za  $i, j = 1, 2, \dots, N$ , gde je  $N$  broj procesa u sistemu.
2. Proces  $P_i$  inkrementira  $i$ -ti element svog vektora u lokalnom vektoru pre nego što obeleži lokalni događaj:  $V[i][i]++$ .
3. Kada  $P_j$  primi poruku on poredi svoj lokalni vektor sa primljenim, element po element, i postavlja element u lokalnom vektoru na veću od vrednosti

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

$V_j$  je vektor  $P_j$  procesa.

### Kako se porede vektori?

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

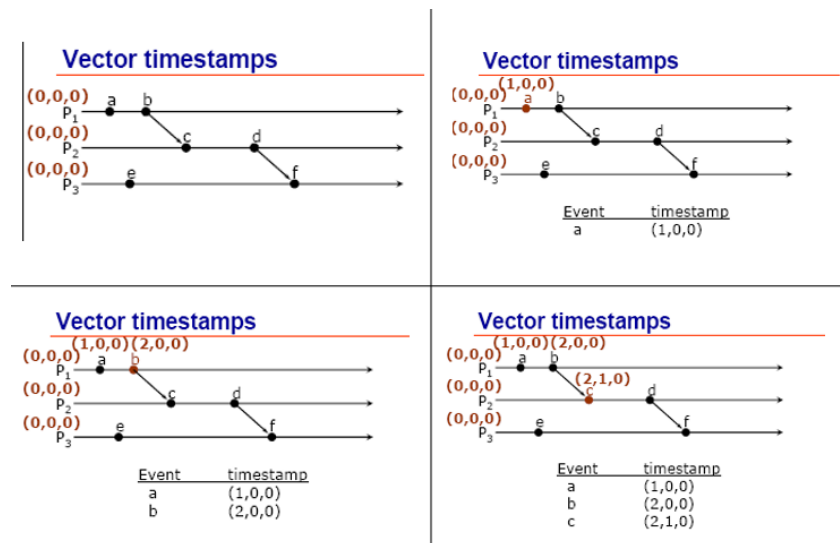
$V \leq V'$  ako je  $V[i] \leq V'[i]$ , za  $i = 1, \dots, N$

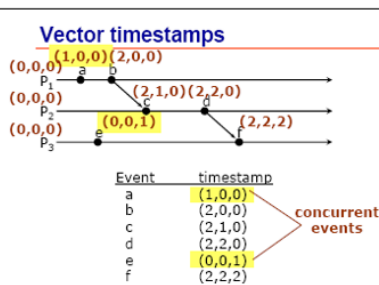
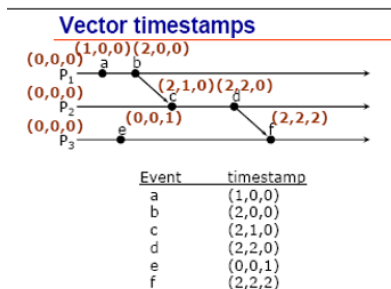
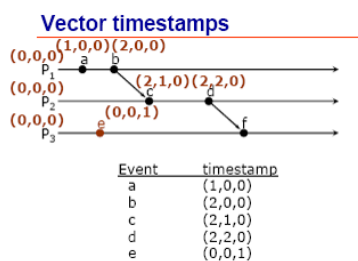
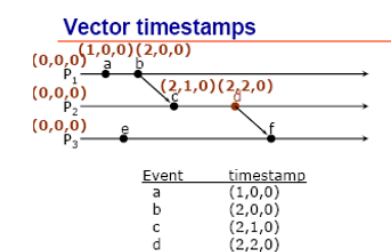
Za bilo koja 2 događaja  $e$  i  $e'$  važi sledeće:

- Ako je  $e \rightarrow e'$  tada važi  $V(e) < V(e')$
- Ako je  $V(e) < V(e')$  tada važi  $e \rightarrow e'$

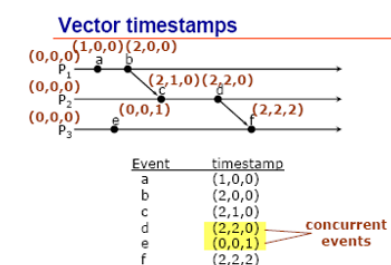
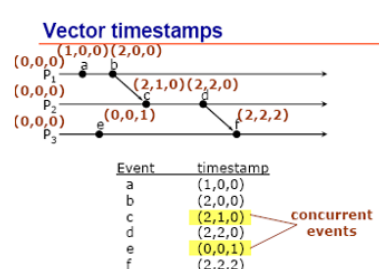
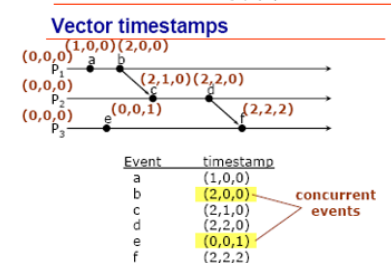
Ako se ne može uspostaviti relacija  $V(e) < V(e')$  ili  $V(e) \geq V(e')$ , tada su događaji **konkurentni** i **nisu međusobno uslovljeni**.

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

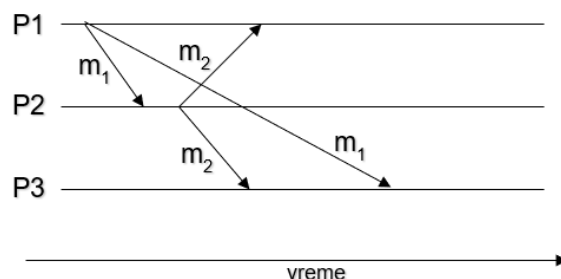




*Vektori a i e ne mogu da se uporede, pa su oni konkurentni!*



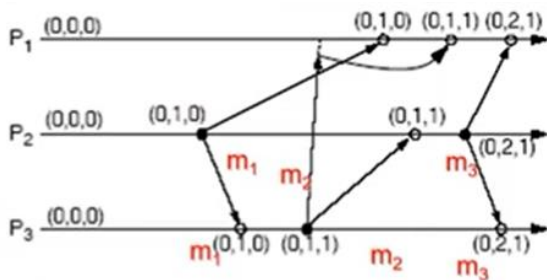
Ako se šalje poruka **m1**, može se desiti da proces P3 prvo primi njen odgovor **m2** pa tek onda poruku **m1**, što nije dobro, pa se mora obezbediti baferovanje poruke **m2** dok ne stigne **m1**. Vektorski časovnici se mogu iskoristiti za to, tako što se **časovnik inkrementira samo kod slanja poruke, ne i kod prijema!**



Pravila:

- Pre emisije poruke **m**, proces  $P_i$  inkrementira svoj vektorski časovnik:  $V_i[i]++$  i poruka **m** se šalje sa markicom **tm** =  $V_i$
- Na prijemnoj strani poruka **m** se ne prosleđuje procesu  $P_j$  dok se ne zadovolje sledeća dva uslova:
  - tm[i] = Vj[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
  - tm[k] ≤ Vj[k], za svako k različito od i**
    - obezbeđuje da proces  $P_j$  primi sve poruke koje je primio  $P_i$  pre slanja poruke **m**
- Kada se poruka prosledi procesu, njegov lokalni časovnik se ažurira

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



Poruka  $m_2$  iz  $P_3$  će biti primljena u  $P_1$  sa  $tm_2=(0,1,1)$ , a lokalni vektor u  $P_1$  je tada  $V_1=(0,0,0)$ . Proveravaju se uslovi algoritma:

- $tm_2[3] = 1, V_1[3] = 0$ , pa je  $tm_2[3] = V_1[3] + 1$  😊
  - $tm_2[2] = 1, V_1[2] = 0$ , pa je  $tm_2[2] > V_1[2]$ , 😞
- ⇒ Poruka  $m_2$  biće baferovana dok  $P_1$  ne primi  $m_1$ .

Kada stigne poruka  $m_1$  iz  $P_2$  sa  $tm_1=(0,1,0)$  u  $P_1$  gde je  $V_1=(0,0,0)$ , proveravaju se uslovi ponovo:

- $tm_1[2] = 1, V_1[2] = 0$ , pa je  $tm_1[2] = V_1[2] + 1$  😊
- $tm_1[0] \leq V_1[0], tm_1[3] \leq V_1[3]$  😊

Oba uslova su ispunjena, poruka  $m_1$  se prosleđuje procesu  $P_1$ , a lokalni časovnik se ažurira na  $V_1=(0,1,0)$ . Nakon ovoga, poruka  $m_2$  može biti isporučena procesu  $P_1$ .

# Sinhronizacija – uzajamno isključivanje

Pored usklađivanja časovnika postoji potreba za **sinhronizacijom pristupa deljivim resursima u DS-u**. Neophodno je da se deljivim resursima pristupa **uzajamno isključivo**. Pošto u DS-u ne postoji zajednička memorija, semafori/monitori se ne mogu koristiti! **Jedini način da se postigne uzajamno isključivanje je razmenom poruka!**

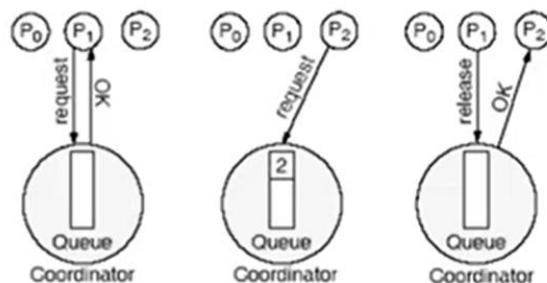
Algoritmi uzajamnog isključivanja u DS:

- **Centralizovani algoritam**
- **Distribuirani algoritam – Ricart Agrawala**
- **Bazirani na žetonima (tokenima)**

## 1. Centralizovani algoritam

Ovaj algoritam oponaša jednoprocesorski sistem i funkcioniše tako što je jedan proces u DS-u odabran za **koordinatora**. Koordinator određuje koji proces kada može da pristupi deljenom resursu. Kada neki proces želi da pristupi deljivom resursu on šalje zahtev koordinatoru, navodeći taj deljivi resurs na određen način.

Ako imamo procese P0, P1, P2 i P3 (koordinator), proces P1 šalje zahtev koordinatoru za korišćenjem resursa. Za svaki resurs postoji **FIFO red čekanja** u koji se smeštaju procesi koji upućuju zahtev za korišćenjem resursa. U prvom slučaju, red čekanja za taj resurs je prazan, pa koordinator odmah daje procesu P1 da koristi resurs. Ako se u drugom slučaju, proces P2 obrati koordinatoru za isti resurs, onda on ide u red čekanja i koordinator ne odgovara procesu P2. Kada proces P1 okonča pristup kritičnoj sekciji (deljivom resursu), on obaveštava koordinatora da je oslobodio pristup tom resursu, pa koordinator javlja procesu P2 da može da koristi taj deljivi resurs.



**Dobre strane algoritma:** jednostavan i lak za implementaciju, zahteva razmenu samo tri poruke (*request*, *OK* i *release*).

**Loše strane algoritma:** koordinator može postati **šć** sa stanovišta performansi, koordinator može i da otkáže (mora da se odabere novi koordinator tada) pa procesi ne mogu da razlikuju otkaz koordinatora od čekanja na oslobodjenje deljivog resursa.

Zbog navedenih loših strana ovaj algoritam može da se modifikuje tako da se razlikuje otkaz koordinatora od čekanja na oslobodjenje deljivog resursa.

## 2. Distribuirani algoritam: Ricart & Agrawala

Potpuna suprotnost centralizovanom algoritmu. Koristi logičke časovnike i grupnu komunikaciju. Kada proces želi da uđe u kritičnu sekciju tj. da pristupi deljivom resursu, on generiše **poruku** u kojoj se nalazi:

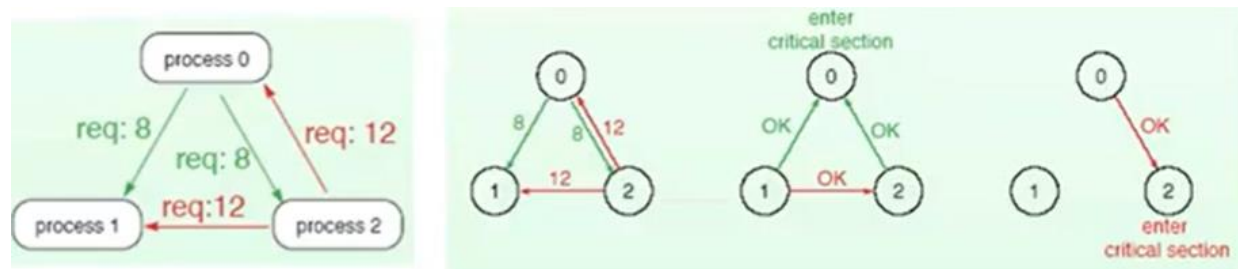
- **ID tog procesa,**
- **ime željenog resursa i**
- **logički časovnik.**

Taj zahtev se šalje svim procesima u grupi. Da bi se ušlo u kritičnu sekciju, čeka se na dozvolu od svih ostalih procesa iz grupe. Tek kada svi procesi iz grupe daju dozvolu procesu, onda on može da uđe u kritičnu sekciju i pristupi deljenom resursu.

Kada proces A primi zahtev za ulazak u kritičnu sekciju od nekog drugog procesa B:

- ako nije zainteresovan za korišćenje tog resursa, odmah mu šalje OK
- ako se proces A već nalazi u toj kritičnoj sekciji, ne odgovara procesu B i smešta njegov zahtev u svoj red čekanja
- ako je proces A upravo poslao zahtev za pristup istoj kritičnoj sekciji i čeka da se njegov zahtev razreši:
  - proces A poredi vremenske markice svog zahteva i primljenog zahteva od B. Proces sa **manjom** vremenskom markicom pobeđuje. Ako pobeđi B, proces A mu šalje OK. Ako pobeđi A, ne šalje potvrdu ka B, već smešta njegov zahtev u svoj red čekanja.
- Kada proces A okonča pristup kritičnoj sekciji, on šalje OK potvrdu **svim** zahtevima koji su bili u njegovom redu čekanja.

### Primer.



Neka proces 0 zahteva pristup kritičnoj sekciji i šalje req:8 ka svim procesima. Neka i proces 2 želi da pristupi istoj kritičnoj sekciji i šalje zahtev sa req:12. Proces 1 nije zainteresovan za taj resurs pa šalje OK i ka P0 i P2. Proces P2 je zainteresovan za korišćenje istog resursa, međutim njegov req ima veću vremensku markicu  $12 > 8$ , pa P2 šalje OK ka P0. Proces 0 ulazi u kritičnu sekciju, dok je P2 prikupio samo jednu OK potvrdu i čeka na ostale (zapravo čeka samo još na potvrdu od P0). Kada P0 okonča pristup kritičnoj sekciji on šalje OK ka procesu P2 i P2 sada ima sve dozvole i pristupa kritičnoj sekciji.

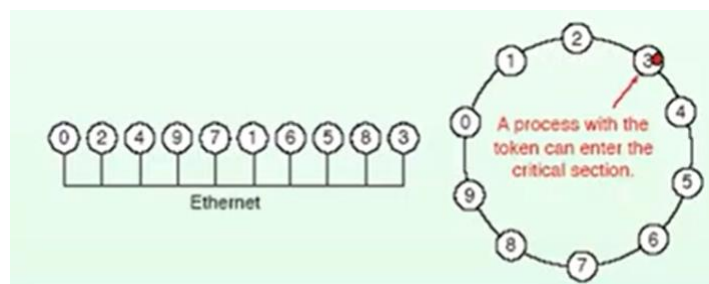
**PROBLEM:** Ovaj algoritam zahteva razmenu  $2(n-1)$  poruka za pristup kritičnoj sekciji ( $n-1$  zahtev i  $n-1$  potvrda). Da bi se dobilo pravo pristupa kritičnoj sekciji, potrebno je da se primi  $n-1$  OK potvrda od svih procesa osim tog koji traži pristup. Greška u bilo kom prenosu može da blokira ceo sistem. Šta ako jedan ili više procesa u grupi otkazu? Onda je nemoguć pristup deljenom resursu 😞

**REŠENJE:** Algoritam se može popraviti tako što će proces uvek slati odgovor na zahtev (dozvolu ili odbijanje). Proces koji šalje zahtev može da koristi timeout mehanizam i da ponovi zahtev dok ne dobije odgovor ili zaključi da je proces mrtav. Može se i ublažiti zahtev i da se dobije potvrda od većine procesa tj. više od  $n/2$  procesa. Ali to usložnjava ceo algoritam.

**Dakle, algoritam teorijski pokazuje da je moguće postići uzajamno isključivanje na ovaj način, ali ima puno problema i nije upotrebljiv.**

### 3. Algoritam sa prstenom i žetonima

Procesi u DS-u formiraju logički prsten koji nema veze sa fizičkom organizacijom procesa, već je softverski generisan. Proces u prstenu komunicira sa svojim logičkim susedom u smeru kazaljke na satu. Svaki proces ima svoj jedinstveni id. Za svaki deljivi resurs postoji **token** (žeton) i da bi proces mogao da pristupi kritičnoj sekciji (tačnije tom resursu) on mora da poseduje token. Token kruži po logičnom prstenu. Kreće od P0 i ide na dalje. Ako je procesu, kod koga je token trenutno, potreban deljeni resurs, on će pristupiti tom deljivom resursu i neće proslediti token svom susedu u prstenu sve dok ne završi pristup deljenom resursu. Kada završi pristup deljivom resursu on šalje token svom susedu. Proces koji nije zainteresovan za pristup resursu samo prosledi token svom susedu u pravcu kazaljke na satu.



Ovim algoritmom se može postići sinhronizacija – u jednom trenutku samo jedan proces može biti vlasnik žetona čime je zagarantovano međusobno isključivanje. Ali, redosled pristupa nije po potrebama, nego se čeka da token stigne do procesa kome treba kritična sekcija.

Ako se žeton izgubi, mora da se izvrši rekonfiguracija prstena da bi se isključio proces u kome se tokena zagubio. Token se regeneriše i ponovo se pušta u prsten.

### Algoritmi izbora koordinatora (election algorithms)

Polazne pretpostavke za izbor koordinatora su:

- **Svaki proces ima jedinstven id**
- **Svaki proces zna jedinstvene id-eve ostalih procesa, ali ne zna da li je odgovarajući proces živ**

Cilj algoritma izbora je da se pronađe aktivni proces sa najvećim id-em i proglasi se za koordinatora. Svi procesi moraju da se slože oko izbora koordinatora. Ovaj proces se obavlja u 2 faze:

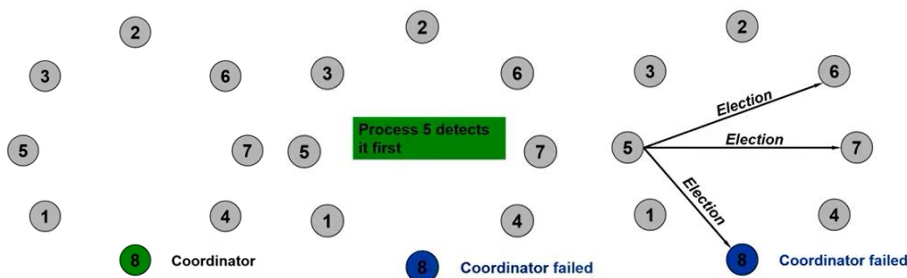
- Selekcija lidera sa najvećim brojem id-a
- Obaveštavanje svih procesa o pobedniku izbora tj. o tome ko je koordinator

## 1. Bully (siledžija) algoritam

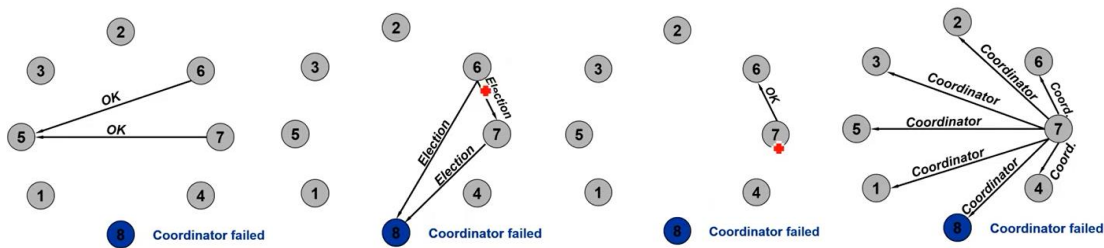
Pretpostavka je da svaki proces ima svoj jedinstven id, kao i da svaki proces zna id-eve ostalih procesa, ali ne zna da li je odgovarajući proces živ.

Ako proces Pi detektuje da koordinator ne odgovara na poruke, Pi startuje algoritam izbora novog koordinatora tako što šalje izbornu poruku svim procesima u sistemu koji imaju veći id od njega. Sad Pi čeka odgovor.

- Ako u okviru određenog vremena ne stigne odgovor ni od jednog procesa, to znači da ni jedan proces sa većim id-em nije aktivan i pobjednik je proces koji je inicirao izbore tj. proces Pi.
- Ako stigne odgovor na poruku izbora, to znači da u sistemu postoji proces sa većim id-em od Pi, pa se proces Pi povlači iz izbora i čeka da dobije poruku od novog koordinatora.



Neka se proces 8 – koordinator – ugasi. Neka proces 5 detektuje da koordinator ne odgovara na poruke. Onda proces 5 šalje poruku o izboru svim procesima koji imaju veći id od njega, a to su 6, 7 i 8.



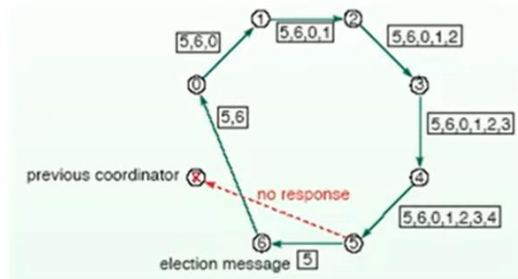
Sada proces 6 i proces 7 šalju OK potvrde ka procesu 5 i on zna da treba da se povuče iz izbora. Sada proces 6 šalje poruku o izboru procesu 7 i 8, a proces 7 šalje samo procesu 8. Proces 6 dobiće potvrdu OK od procesa 7 i udaljiće se iz izbora, a proces 7 neće dobiti nikakvu potvrdu od procesa 8 i to je znak da je novi koordinator proces 7. Sada proces 7 šalje svima poruku i kaže da je on novi koordinator.

## 2. Ring algoritam izbora

Pretpostavke su ovde iste kao i u prethodnom algoritmu. Procesi formiraju logički prsten kao kod algoritma sa tokenom. Ako neki proces detektuje da koordinator ne funkcioše, startuje algoritam izbora novog koordinatora:

- Šalje poruku izbora sa svojim ID-em svom susedu u prstenu (u smeru kazaljke na satu)
- 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 (na slici je to 5), on detektuje u poruci svoj ID. Na osnovu primljene poruke proces zaključuje ko su članovi prstena i šalje novu poruku KOORDINATOR sa brojem procesa koji ima najveći ID (to je 6) da obavesti ostale procese o novom koordinatoru.
- Postoji i druga varijacija ovog algoritma, gde ako proces primi poruku sa ID-em koji je veći od njegovog, on samo prosledi poruku, bez da dodaje svoj ID. Ako je njegov ID veći od dobijenog u poruci, onda on šalje poruku sa svojim ID-em, a onaj dobijeni briše. Tako da, ako proces 5 šalje poruku ka 6, proces 6 će proslediti ka procesu 0 samo 6, ne i 5. Sada će proces 0 uvideti da je 6 veće od 0, pa će sve do procesa 5 da kruži. Kada stigne u proces 5 poruka ima id 6. Kada stigne u proces 6 on će prepoznati da je njegov ID jednak onom u poruci i proglasiće se koordinatorom.

# Konzistentnost i replikacija

## Replikacija

Replikacija je **umnožavanje** i to je široko prihvaćena tehnika za povećanje pouzdanosti, performansi i skalabilnost sistema.

- **Pouzdanost** – ako je fajl sistem repliciran, moguće je da se nastavi sa radom i ako neka kopija bude narušena, jednostavnim komutiranjem na drugu kopiju
- **Performanse** – smanjuje se komunikaciono kašnjenje i smanjuje se vreme pristupa podacima
- **Skalabilnost** (proširljivost) – i u pogledu korisnika i u pogledu računara

Replikacija dovodi do niza novih problema. Najveći problem replikacije je **KONZISTENCIJA REPLIKA**. Ako neka kopija podataka bude modifikovana onda kopije nisu konzistentne! Da bi kopije bile usaglašene potrebno je svaki put kada se izvrši modifikacija jedne lokalne kopije izvrši i modifikacija svih ostalih udaljenih kopija. Međutim, to ažuriranje utiče loše na performanse posebno u velikim DS sistemima jer raste saobraćaj kroz mrežu. Potrebna je **GLOBALNA SINHRONIZACIJA** koja je **nemoguća** u DS-u.

**Kaže se da su replike međusobno konzistentne ako read operacija nad njima uvek vraća isti rezultat.**

Konzistencija je definisana u kontekstu **read** i **write** operacija nad **deljivim podacima**:

- **Read** operacija je konzumiranje (čitanje) nekog podatka
- **Write** operacija je bilo kakva modifikacija podatka
- Proces koji obavlja read operaciju očekuje da dobije vrednost koja predstavlja rezultat poslednje write operacije

**Deljivi podaci** su resursi kojima mogu pristupati svi procesi u DS sistemu. Svi podaci koji su replicirani nazivaju se **SKLADIŠTE PODATAKA**. Svaki proces ima lokalnu kopiju celokupnog skladišta podataka.

Kada kažemo da su kopije međusobno konzistentne?

- Ako **read** operacija nad bilo kojom postojećom kopijom vraća isti rezultat.

Da li vršiti replikaciju pa onda rešavati sve ove probleme? Da li ostaviti centralizovane podatke bez repliciranja? Ipak je bolji prvi prilaz – repliciranje procesa. Kada i kako se vrši ažuriranje kopija određuje cenu replikacije. Postoji mnogo **modela za održavanje konzistencije** replikacija.

*Problem održavanja konzistentnosti distribuiranih skladišta je vreme prenosa poruka (ažuriranje) tj. komunikaciono kašnjenje i nepostojanje globalnog časovnika.*

### Modeli konzistencije:

- 1) **Striktna**
- 2) **Sekvencijalna**
- 3) **Kauzalna**
- 4) **FIFO**

Svi slabiji modeli konzistencije sadržani su u jačem modelu. Najjači model je striktna konzistencija.

## Striktne konzistencije

[To je najjači model konzistencije i bazira se na postojanju **globalnog časovnika**.]

**Skladište podataka je striktno konzistentno ako bilo koja operacija nad operandom  $X$  vraća rezultat poslednje write operacije nad tim operandom  $X$ .**

Primer. Objasnenje zašto je ovo nemoguć model:

Proces  $P_i$  ažurira  $x$  u trenutku  $t_1$  sa 4 na 5, pa  $P_i$  sada mora da distribuira svim procesima ovo ažuriranje u sistemu. Neka proces  $P_j$  u trenutku  $t_2 > t_1$  i to za 1ns čita vrednost  $x$  i ono što se očekuje je vrednost 5 (pošto je reč o striktno konzistentnom modelu). Neka se procesi  $P_i$  i  $P_j$  izvršavaju na računarima povezanim optičkim kablom koji su udaljeni 3m. Prostiranje tog ažuriranja bi trebalo da je 10 puta veće od brzine svetlosti. Dakle, ovakav vid konzistencije nije moguće postići u DS-u! **Problem je što se model zasniva na apsolutnom globalnom vremenu.**

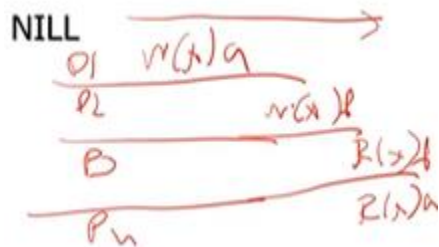
Oznake:

- **$W_i(x)a$**  – označava da proces  $P_i$  modifikuje podatak  $x$  nakon čega on ima vrednost  $a$
- **$R_i(x)b$**  – proces  $P_i$  čita podatak  $x$  i kao rezultat dobija vrednost  $b$

Svaki podatak je inicijalno NIL. Gledajmo deo pod b, to nije prihvatljivo ponašanje sa strane striktno konzistencije. Dakle, ne sme da se desi da proces  $P_2$  vidi staru vrednost resursa  $x$ !

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 striktno konzistencije.		



Isto i ovde, data su 4 procesa  $P_1, 2, 3, 4$  i ako  $P_1$  promeni  $W(x)a$ , pa  $P_2$  uradi  $W(x)b$ , pa  $P_3$  procita i dobije  $b - R(x)b$ , pa  $P_4$  procita  $a$  dobije  $a - R(x)a$ . To ne sme da se desi! To ne podržava striktno konzistencija.

**Ako je skladište podataka striktno konzistentno to znači da su svi upisi trenutno vidljivi, bez obzira koliko brzo se oni dešavaju.**

Zbog komunikacionog kašnjenja i nepostojanja globalnog časovnika ovaj model nije moguće implementirati u DS-u. Zato su stvoreni slabiji modeli konzistencije i oni nisu bazirani na apsolutnom vremenu.

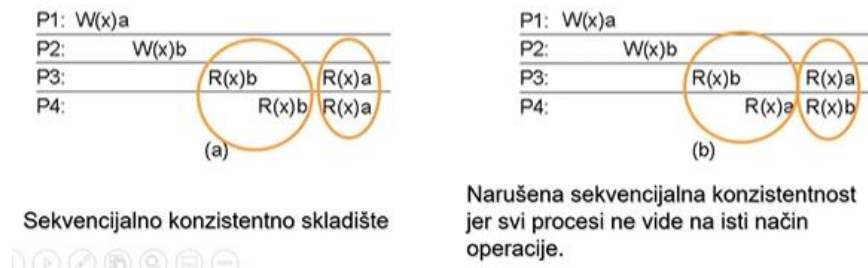
## Sekvencijalna konzistencija

[Slabiji model od striktno, ali je najjači model koji se može postići u DS-u. ]

Skladište podataka je sekvencijalno konzistentno ako:

- **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 sekvenci u redosledu koji je određen njihovim programom.**
- **Programski raspored svakog procesa mora biti ispoštovan!**

Definicija kaže da se procesi izvršavaju konkurentno ali je utisak kao da se izvršavaju sekvencijalno, kao da ne pristupaju jednovremeno podacima. Svi procesi vide dešavanja u DS-u na isti način.



Prvo skladište podataka nije striktno konzistentno, jer onda ne bi smelo da se vidi a, ali zato jeste sekvencijalno konzistentno.

### Primer1.

Vrednosti x, y, z su inicijalno 0.

Process P1	Process P2	Process P3	
x = 1; print ( y, z);	y = 1; print (x, z);	z = 1; print (x, y);	write read
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);	
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
(a)	(b)	(c)	(d)

- Sa 6 naredbi moguće je dobiti 6! (720) različitih preplitanja, mada neka od njih narušavaju programski redosled.
- Razmotrimo 5! (120) sekvenci koje počinju sa x=1.
  - Pola od njih ima print(x,z) pre y=1 i tako narušava sekvencijalno uređenje programa.
  - Takođe, pola od preostalih permutacija će imati print(x,y) pre z=1 i takođe narušava sekvencijalni redosled.
  - Samo ¼ 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!

Nisu sve binarne vrednosti moguće na izlazu, nije moguće 000000, niti 001001, jer mora da se ispoštuje redosled instrukcija po procesu! Mora x = 1 pre print(y,z)!

Da li je potpis **001001** validan?

Prva dva bita, 00, znače da su z i y oba bili 0 kada je P1 obavio štampanje. Ova situacija nastupa samo ako P1 izvrši obe svoje naredbe pre nego što P2 i P3 otpočnu sa izvršenjem. Sledeća dva bita, 10, označavaju da P2 mora da otpočne sa izvršenjem posle P1 ali pre P3. Poslednja dva bita, 01, znače da P3 mora da se okonča pre nego što

Četiri ispravna niza izvršenja naredbi za procese. Vertikalna osa je vreme.  
a) Proces se izvršavaju po redosledu P1, P2, P3.  
Sledeća tri primera demonstriraju različita, ali validna preplitanja 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 pretpostavkom sekvencijalne konzistencije.

P1 startuje, ali zbog prva dva bita (00) videli smo da P1 mora da staruje prvi! **Zbog toga potpis 001001 nije moguć!**

**Primer2.** Vrednosti za A i B su inicijalno 0.

\* 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 sekvencijalne konzistencije?

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

Oznake za naredbe bi bile P1.1 i P1.2, P2.1 i P2.2 i P3

Najlakše se može proveriti koje vrednosti za U,V i W nisu moguće na osnovu sledeće tabele:

U	V	W	
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	nije moguće jer bi značilo da P2 vidi prvo A=1 a onda A=0
1	0	1	
1	1	0	nije moguće jer bi značilo da P2 vidi prvo A=1 a onda A=0
1	1	1	

**0 0 0** – ako se izvrše P2, P3 pa onda P1

**0 0 1** – ako se izvršilo u = A, v = B, pa A = 1, pa w = A, sto je moguće jer je to redosled P2.1, P3, P1.1, P2.2.

**0 1 0** – ako se izvršilo sve iz P2, pa sve iz P1, pa P3

**0 1 1** – ako izvršilo se u = A, pa A = 1, pa B=1, pa w = A, pa v = B, sve je ispostovano

**1 0 0** – ako se izvršilo A=1, onda ce i u i w biti 1, a ovde je u = 1, a w = 0, sto nije moguće. W ne može da se dodeli na A pre u!

**1 0 1** – e ovo je moguće, A = 1, pa ceo P2 se izvrši

**1 1 0** – ne može u da bude 1, a w da bude 0

**1 1 1** – moguće, izvrši se lepo P1, P2, P3

## Uslovna (kauzalna) konzistencija

[Slabija od sekvencijalne konzistencije. ]

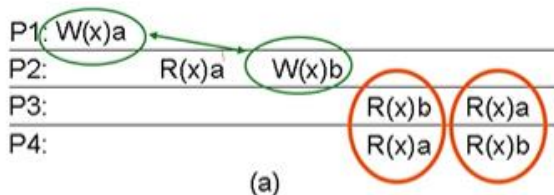
Skaldisite podataka je kauzalno konzistentno ako:

- **Upisi (write) koji su potencijalno uslovljeni moraju da se vide u svim procesima u istom redosledu.**
- **Konkurentni upisi se mogu videti u razlicitom redosledu u razlicitim procesima.**
- **Programski raspored svakog procesa mora biti ispostovan!**

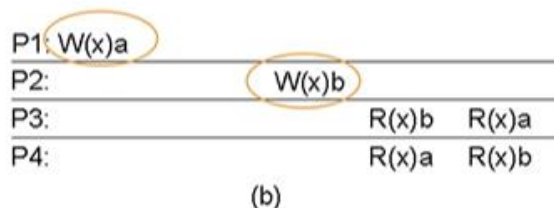
P1:	W(x)a		W(x)c	
P2:		R(x)a	W(x)b	
P3:		R(x)a		R(x)c
P4:		R(x)a		R(x)b

U procesu P1 upis **W(x)a** i u procesu P2 upis **W(x)b** su potencijalno uslovljeni upisi, jer je P2 koristio vrednost x sa R(x)a pre nego što je upisao x sa W(x)b.

Proces P3 prvo vidi  $R(x)c$ , pa onda  $R(x)b$ . Proces P4 prvo vidi  $R(x)b$ , pa onda  $R(x)c$ . Sa stanovišta sekvencijalne konzistencije ovo je nekorektno, kao i sa stanovišta stroge konzistencije, ali sa stanovišta uslovne (kauzalne) konzistencije, ovo je u redu **jer  $W(x)c$  i  $W(x)b$  nisu potencijalno uslovljeni. Bitno je da vide a pa b, jer oni jesu potencijalno uslovljeni.**



a) postoji potencijalna uslovljenost između P1 i P2 zbog  $W(x)a$ ,  $R(x)a$  i  $W(x)b$ . Pošto su potencijalno uslovljeni, oni moraju da se vide u datom redosledu, prvo a pa b. To ovde nije slučaj. Dakle, nije ispoštovana uslovna konzistencija! Ovo skladište podataka nije kauzalno konzistentno.



b) Da nije bilo čitanja  $R(x)a$  onda ne postoji uslovljenost između P1 i P2, pa redosled vidjenja a i b nije bitno, onda ovo jeste kauzalno konzistentno, jer ne postoji prekršena potencijalna uslovljenost.

Implementacija uslovne konzistencije obavlja se pomoću vektorskih časovnika.

Da je u primeru pod a i u P3 i u P4 bilo  $R(x)b$  i  $R(x)a$ , onda bi to bilo kauzalno konzistentno skladište podataka.

## FIFO konzistencija

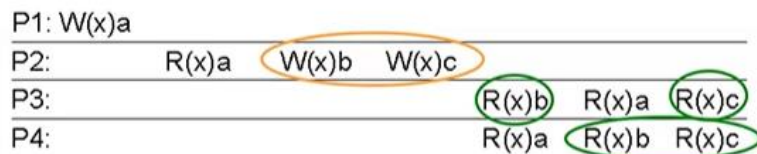
[Slabiji od kauzalne konzistencije.]

- **Upisi koje obavi jedan proces se vide od strane drugih procesa po redosledu po kome su izdati.**
- **Upisi različitih procesa mogu se videti u različitom redosledu u različitim procesima.**
- **Programski raspored svakog procesa mora biti ispoštovan!**

Ne postoje garancije o tome kako različiti procesi vide upise drugih procesa, osim što dva ili više upisu iz istog izvora moraju ići po redu.

Implementacija:

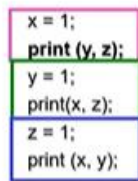
- Svaki proces svojoj poruci ažuriranja dodaje proces id i redni broj
- Svi ostali procesi primenjuju ažuriranja po redosledu u kome su ona obavljena u izvornom procesu



Postoji kauzalnost između P1 i P2 ali to ovde za FIFO nije bitno. Ono što jeste bitno je da P3 i P4 vide prvo b pa c, jer je P2 tako upisao  $W(x)b$  pa  $W(x)c$ .

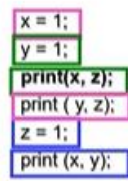
Ni jedan od prethodnih modela konzistencije ne dozvoljava ovakav redosled događaja

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



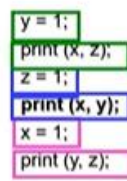
Prints: 00

(a)



Prints: 10

(b)



Prints: 01

(c)

Svaki od procesa na drugačiji način može da vidi redosled, jedino ograničenje je da se aktivnosti iz drugih procesa vide u korektnom redosledu što se svodi na: **Programski raspored svakog procesa mora biti ispoštovan!**

## Sekvencijalna naspram FIFO konzistencije

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

Process P1	Process P2
x = 1; if (y == 0) kill (P2);	y = 1; if (x == 0) kill (P1);

Pretpostavimo 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!

4. Nad objektom x obavljene su sledeće operacije u distribuiranom skladištu:

- P1: write(x)A
- P2: write(x)B, read(x) C
- P3: read(x) B, read(x) A, write(x)C
- P4: read(x) B, read(x) A

Nacrtati kako izgleda stvarni redosled događaja ako je skladište podataka sekvencijalno konzistentno.

P1 → 00, ako se izvrši x=1, pa print (y,z)



# Konzistentnost sa stanovišta klijenta (client-centric)

Modeli konzistencije iz prethodne oblasti u centru su imali skladište podataka. Govorili su o tome šta procesi koji pristupaju distribuiranom skladištu podataka mogu da očekuju. Ti modeli zovu se data-centric modeli. *Client-centric je slabiji model konzistencije od data-centric modela i lakše se može implementirati. Ovde je u centru pažnje samo jedan klijent koji potencijalno menja svoju lokaciju.*

Imamo klijenta koji u jednom trenutku pristupa DS skladištu podataka (jednoj od replika), čita ili modifikuje podatke. Može da promeni svoju lokaciju, da se diskonektuje i konektuje na nekoj drugoj lokaciji. Postavlja se pitanje: šta klijent može da očekuje kada čita replike istih podataka sa druge lokacije?

Razlikuju se 4 modela konzistencije:

- **Monotona čitanja** (*monotonic reads*)
- **Monotoni upisi** (*monotonic writes*)
- **Čitaj svoje upise** (*read your writes*)
- **Upiši nakon čitanja** (*writes follow reads*)

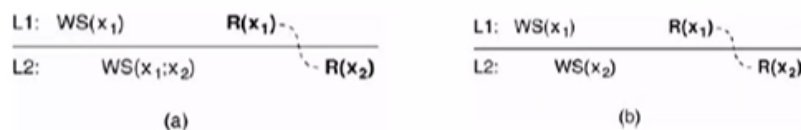
Oznake koje se koriste:

- $Xi[t]$  – verzija podatka  $X$  na lokaciji  $Li$  u trenutku  $t$
- $WS\ xi[t]$  – skup WRITE operacija koje su učinjene na lokalnoj kopiji na lokaciji  $Li$  u trenutku  $t$  i one su dovele do toga da  $X$  ima vrednost  $Xi$  u trenutku  $t$ .
- $WS(xi[t1], xj[t2])$  – ako se set operacija  $WS\ xi[t1]$ , koji se obavio na lokaciji  $Li$  u trenutku  $t1$ , obavi i na lokalnoj kopiji  $Lj$ , a zatim se na toj lokaciji  $Lj$  obavi i novo ažuriranje, to se označava sa  $WS(xi[t1], xj[t2])$

## Monotona čitanja (monotonic reads)

Ako proces pročita vrednost podatka  $x$  na jednoj lokaciji  $L1$ , a zatim promeni svoju lokaciju, šta može da očekuje, kakve podatke može da pročita?

**Proces podržava monotona čitanja ako svaki put kada se poveže na drugu repliku skladišta podataka on čitanjem dobija istu ili noviju vrednost te promenljive.**



- (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, proces promeni lokaciju na  $L2$ . On čita vrednost promenljive  $x$ , dobija vrednost  $x=x_2$ . Skup operacija koji je obavljen na  $L1$ , mora da se propagira i na lokaciju  $L2$  i to pre nego što se na lokaciji  $L2$  izvrše dodatna ažuriranja. To je označeno sa  $WS(x_1, x_2)$ . Dakle, prvo je izvršeno ažuriranje na lokaciji  $L1$ , a zatim je na lokaciji  $L2$  obavljeno dodatno ažuriranje. Čitanjem  $x$ -a na lokaciji  $L2$ , dobija se  $x_2$ .



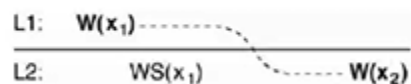
- (b) **Konzistencija monotonih čitanja nije zadovoljena.** Na lokaciji L1 klijent je obavio skup upisa (ažuriranja) koji su dali vrednost promenljivoj  $x$  da bude  $x_1$ . Kasnije imamo da je klijent promenio lokaciju na L2, na toj lokaciji obavio je ažuriranja i čita vrednost  $x = x_2$ . Međutim, ako ažuriranja sa L1 nisu prosleđena na L2 pre nego što je u L2 obavljeno ažuriranje, onda ovaj tip konzistencije nije zadovoljen.

## Monotoni upisi (monotonic writers)

**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 lokacija L1.**

Primer.

- Proces P obavi write operaciju na lokaciji L1,  $W(x_1)$
- Kasnije proces P obavi 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, dakle  **$WS(x_1)$  na lokaciji L2 pre obavljanja  $W(x_2)$**



Primer. Nije zadovoljen ovaj monotoni upis

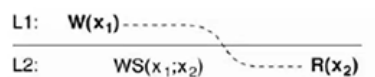


Nedostaje propagiranje ažuriranja  $W(x_1)$  sa lokacija L1 na lokaciju L2! Ovo ponašanje nije konzistentno sa stanovišta monotonih upisa.

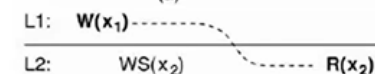
## Čitaj svoje upise (read your writes)

Vrlo blizak monotonom čitanju. Proces obezbeđuje ovaj vid konzistencije, ako je zadovoljeno sledeće:

- Efekat **write** operacije koju obavlja proces P nad podatkom  $x$  će uvek biti viđen kasnijim čitanjem podatka  $x$  od strane istog procesa



(a)



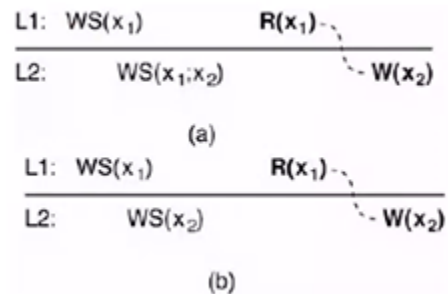
(b)

(a) klijent je pre promene lokacije obavio ažuriranja na lokaciji L1, nije čitao podatke. Promenio je lokaciju na L2 i na njoj je prvo obavio ažuriranje sa lokacija L1:  $x=x_1$ , pa ažuriranje za lokaciju L2:  $x=x_2$ .

(b) **Čitaj svoje upise nije zadovoljen** jer na lokaciji L2 nisu obavljena ažuriranja sa L1, tako da vrednost  $x=x_2$  ne mora da obuhvata u sebi i modifikacije sa L1.

## Upis sledi čitanje (writes follow reads)

Write operacija koju obavlja proces P nad podatkom X nakon prethodnog čitanja tog podatka se obavlja nad istom ili novijom vrednošću tog podatka.



- (a) Na lokaciji L1 obavljena je modifikacija nad podatkom X i pročitana je vrednost  $x=x_1$ , zatim je klijent promenio lokaciju L2 i obavio ažuriranje i na drugoj lokaciji, ali da bi bila ispoštovana ova konzistencija, moraju da se ažuriranja sa L1 distribuiraju i obave na lokaciji L2 –  $WS(x_1, x_2)$ . nakon toga na lokaciji L2 obavlja novu modifikaciju  $x=x_2$ .
- (b) Nije ispoštovan ovaj vid konzistencije.

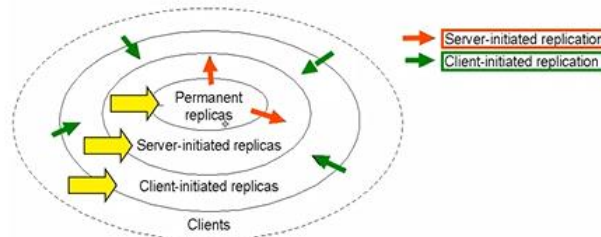
**SVI OVI MODELI SE PRIMENJUJU KOD APLIKACIJA GDE 1 PROCES OBAVLJA AŽURIRANJA! LAKŠE JE POSTIĆI OVAKVE VIDOVE KONZISTENCIJE OD DATA KONZISTENCIJA. DA BI SE IMPLEMENTIRALI MOGU SE KORISTITI LAMPORTOVE VREMENSKE MARKICE.**

# Vrste replika

Replikacija se koristi da bi se poboljšale performanse i pouzdanost DS-a tako što se replike skladišta podataka postavljaju bliže klijentima koji koriste te podatke. Ključno pitanje u DS-u koji podržava replikaciju je **gde, kada i ko može obaviti repliciranje** i **koji se mehanizmi koriste za održavanje konzistencije**.

Vrste replika:

- **Permanente replike**
- **Replike inicirane od strane servera**
- **Replike inicirane od strane klijenta**



## Permanente replike

Početni skup replika koje obrazuju distribuirano skladište podataka. Broj permanentnih replika je relativno **mali**. Ako govorimo o Web serverima distribucija Web sajtova se obavlja u 2 oblika:

- repliciranje fajlova koji čine web sajt na ograničenom broju servera na jednoj lokaciji (lokacija je klaster servera) i kada stigne neki zahtev do web servera on se prosleđuje jednom od servera po Round Robin principu.
- Koristiti usluge web hostinga i da se na udaljenim lokacijama naprave replike sadržaja određenih servera, tako da klijent može da pristupa tim mirror lokacijama, umesto glavnom serveru

## Replike inicirane od strane servera

Ove replike su privremene i postavljaju se u regione odakle dolazi veliki broj zahteva. Da bi se odredilo kada treba napraviti privremenu kopiju, svaki server vodi računa o broju pristupa određenom fajlu i odakle ti pristupi dolaze. Ako u jednom trenutku broj zahteva sa neke lokacije bude veliki dobro je instalirati privremene replike tamo gde su potrebne.

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 C1 i C2 imaju isti najbliži server P, svi zahtevi za pristup fajlu F u serveru Q od strane C1 i C2, se jedinstveno registruju u serveru Q tj. registruju se kao da dolaze od servera P. Ako je broj obraćanja fajlu F na serveru Q viši od praga replikacije donosi se odluka da se izvrši replikacija fajla F na server P.

## Replike inicirane od strane klijenta

**Lokalna memorija na strani klijenta** ili može biti na nekom **proxy serveru koji se nalazi na istom LAN-u** kao i klijent. **Na klijentskoj strani pravi se replika podataka koji su pročitani sa skladišta podataka. Server sa koga su kopije napravljene (server na kome je skladište podataka) nema nikakvu odgovornost da održava kopije konzistentnim.** Dakle, klijent je dužan da vrši održavanje svojih kopija, naravno ako želi konzistentne podatke.

Kada klijent pristupi nekoj **web stranici**, ta web stranica se **kešira** na **klijentskom disku** ili **proxy serveru** u klijentskoj LAN mreži. Ako klijent u kraćem vremenskom periodu ponovo pokuša pristup toj web stranici, pristup se obavlja na osnovu keša, a **ne pristupa se opet serveru na kome se stranica zapravo nalazi**. Ako klijent vodi računa o konzistentnosti podataka, on pri pristupu podacima koji su replicirani, u pozadini kontaktira server koji je vlasnik tih podataka i proveriti da li postoji nova verzija tih podataka. Ako postoji novija verzija, ta verzija se beleži tj. pribave se novi podaci sa servera, u suprotnom koriste se podaci koji su replikovani na klijentskoj strani.

### **Šta ako se neka kopija (replika) modifikuje?**

**Potrebno je obezbediti da se sve ostale replike tog skladišta podataka takođe modifikuju.** Šta se prosleđuje pri tom ažuriranju? Postoje 3 opcije (nama je cilj smanjenje saobraćaja kroz mrežu):

#### ➤ **Samo slanje obaveštenja o obavljenom ažuriranju**

Kada se na jednoj strani obavi ažuriranje, ostale kopije (tačnije klijenti) skladišta podataka se samo obaveste da je obavljeno ažuriranje. To je invalidacija ostalih kopija. Replika na kojoj se obavi modifikacija šalje poruku svim ostalim replikama da je modifikacija obavljena i one sada znaju da podatke koje poseduju nisu konzistentni. Ovaj vid ažuriranja je pogodan ako je odnos broja upisa naspram broja čitanja veliki. Tačnije, ako se na jednoj strani obavlja veliki broj modifikacija, a na svim ostalim replikama se obavlja mali broj čitanja, onda ovaj prilaz ima smisla.

#### ➤ **Slanje modifikovanih podataka**

Kada se na jednoj strani obavi ažuriranje podaci koji su ažurirani se proslede svim replikama. Ovo ima smisla ako je odnos čitanja podataka i pisanja podataka mnogo velik. Tačnije, kada je mnogo veći broj čitanja nego broj modifikacija, onda ovo ima smisla.

#### ➤ **Slanje operacija koje su izvršene nad modifikovanim podacima**

Umesto da se prosleđuju podaci koji su ažurirani, propagiraju se operacije koje treba da se obave da bi se izvršilo ažuriranje. Dakle, prosleđuju se operacije koje su izazvale ažuriranje, a ne sami podaci. Umesto da se prenosi veliki modifikovani fajl, prenosi se samo komanda koja će se obaviti na svim ostalim replikama. Ovaj način ažuriranja je poznat kao **AKTIVNA REPLIKACIJA** jer svaka replika obavlja aktivno izvršenje naredbi koje su dovele do nove verzije podatka. Zahteva više procesorskog vremena jer komande mogu biti kompleksne. Ažuriranja se često mogu obaviti uz minimalni saobraćaj.

### **Ko inicira ažuriranje?**

#### ➤ **Server (push prilaz)**

Ažuriranje se prenosi replikama na inicijativu samog skladišta podataka iako klijenti (replike) to nisu zahtevali. Ovo se koristi kod permanentnih i serverski kreiranih replika.

#### ➤ **Na zahtev klijenta (pull prilaz)**

Server nema nikakve obaveze o konzistentnosti podataka, već je sve ostavljeno klijentu da vodi računa da li su podaci konzistentni ili ne. Uobičajeno je da se na klijentskoj strani pri pristupu web stranici, ona pamti u kešu u slučaju ponovog pristupa istoj stranici. O tome je ranije već bilo reči.

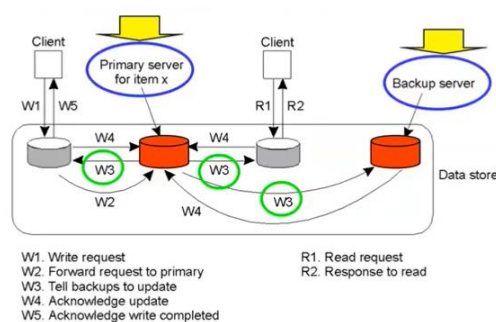
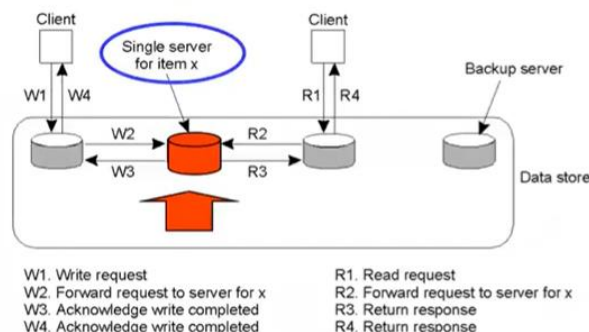
# Protokoli konzistencije

Opisuju implementaciju određenog modela konzistencije i postoje 3 vrste protokola:

- **Protokoli zasnovani na postojanju primarne kopije**  
Najjači vid konzistencije koji se može ostvariti u DS-u je **sekvencijalna konzistencija** i za njega se koriste protokoli zasnovani na postojanju primerne kopije. Postoje 2 varijante:
  - **Remote-write protokoli (protokoli sa udaljenim upisom)** – primar na kome mogu da se obavljaju modifikacije je fiksiran i nalazi se na jednoj lokaciji. Da bi se obavila modifikacija, mora se pristupiti toj lokaciji.
  - **Local-write protokoli (protokoli sa lokalnim upisom)** – primar privremeno **migrira** na lokalnu kopiju i tu se lokalno obavi modifikacija.
- **Protokoli sa više replikovanih replika (replirani write protokoli)** – write operacije može biti inicirana u bilo kojoj replici.
- **Keš koherentni protokoli** – inicirani od strane **klijenta**, a ne servera

## Remote-write protokoli (zasnovani na postojanju primarne kopije)

Najjednostavniji protokol baziran na postojanju primarne kopije i ovaj protokol je protokol kod koga se **sve read i write operacije obavljaju na udaljenom primarnom serveru gde je primarna kopija locirana**. Druge kopije koje postoje služe samo kao backup primarnog servera u slučaju otkaza. U suštini, podaci nisu uopšte replicirani, već se nalaze na jednom serveru odakle se ne mogu pomeriti. Svi zahtevi se upućuju primarnoj kopiji, a ostale kopije služe kao backup.



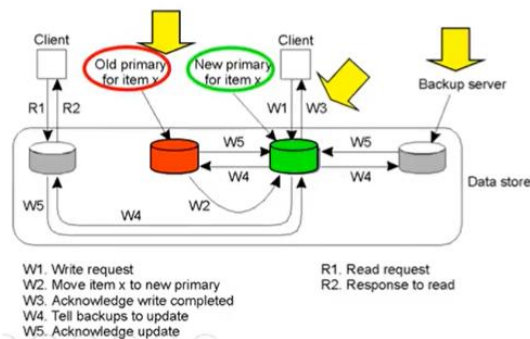
Sa stanovišta konzistencije mnogo su interesantniji **protokoli koji dozvoljavaju procesima da obave read na lokalno raspoloživoj kopiji, a operacije write na fiksnoj primarnoj kopiji**. Replike su najčešće locirane u LAN mreži. Klijent pristupa lokalnoj kopiji skladišta podataka ako želi da samo čita podatke, ali ako želi da modifikuje podatke, onda mora da kontaktira primar jer sve modifikacije se obavljaju na primarnoj kopiji. Na slici se vidi da krajnje levi klijent traži Write operaciju nad podacima (W1) što se šalje lokalnom serveru, nakon toga zahtev za modifikaciju

se prosleđuje primarnom serveru (W2), primarni server obavi modifikacije i šalje promene svim kopijama podataka (W3) (sivi lokalni server sa desne strane i narandžasti backup server sa krajnje desne strane). Kada se obavi ažuriranje svih kopija one primaru jave da su uspešno obavile ažuriranje (W4), tek nakon toga lokalni server dozvoljava klijentu koji je inicirao modifikaciju da nastavi sa radom (W5). **Primar može urediti sve dolazne zahteve za modifikacijom na globalno jedinstveni način tako da svi procesi vide sve write operacije u istom redosledu – što je pogodno za postizanje sekvencijalne konzistencije.**

## Local-write protokoli (zasnovani na postojanju primarne kopije)

**Dozvoljava se da primarna kopija privremeno migrira na stranu klijenta koji je inicirao ažuriranje.**

Kada je reč o **čitanju**, sva čitanja se obavljaju **nad lokalnim kopijama skladišta podataka**. Kada je u pitanju **modifikacija** nekog podatka u lokalnom skladištu, neophodno je da od **primara za dati podatak dobije dozvolu da izvrši modifikaciju**. To je simbolično prikazano kao da primar **migrira** u lokalno skladište podataka klijenta, ali zapravo samo daje dozvolu da se modifikacija izvrši.



**Zapravo, ono što se zaista dešava je da klijent koji želi da modifikuje neki podatak dobija privremenu dozvolu od primara da obavi tu modifikaciju.** Nakon obavljene modifikacije, vrši se ažuriranje svih ostalih kopija tog skladišta podataka. **U jednom trenutku samo jedan klijent može da obavi ovu modifikaciju.** Da bi se ovakav način rada ostvario, mora da se koristi neblokirajući protokol koji omogućava da se ažuriranje ostalih kopija obavi tek kada je primar obavio ažuriranje. Osnovna

prednost ovog prilaza je to što kada klijent dobije pravo da vrši modifikaciju podataka, može seriju modifikacija da obavi pre nego što se primar preseli na nekog drugog klijenta.

Klijent želi da izvrši neku modifikaciju i šalje zahtev ka svom lokalnom serveru (W1). Lokalni server dobija dozvolu od primara da može da vrši modifikacije (W2) i na serveru se izvrše modifikacije. Server obavesti klijenta da su modifikacije obavljene (W3), pa kontaktira primar i prosleđuje mu modifikaciju, kao i drugim serverima replika (W4). Onda mu svi serveri (i primar) odgovaraju da su obavili ažuriranja.

## Replicirani write protokoli

Dozvoljeno je da se **write** operacija obavi **na više replika**, umesto na jednoj kao kod protokola koji su zasnovani na primarnoj kopiji. Razlikujemo 2 varijante:

- **Koriste aktivnu replikaciju – operacija modifikacije** se prosleđuje svim replikama. Ažuriranje se obavlja prosleđivanjem **write komande/komandi**, a ne samih podataka. Svim replikama se šalje ista komanda. Sve replike su ravnopravne.  
**Problem:** operacije u svim replikama moraju da se obave u istom redosledu, a za to je potrebno implementirati mehanizam potpuno uređene grupne komunikacije.
  - Prva opcija za to je da se koriste **Lamportove vremenske markice**.
  - Druga opcija je **centralni koordinator**, pa bi svaka operacija prvo bila prosleđena centralnoj komponenti koja svakoj write operaciji dodeljuje **jedinstven id**, pa se onda operacija prosledi svim replikama. Zatim se na replikama operacije izvršavaju prema id-evima operacija. Tu se može javiti problem skaliranja jer centralni koordinator može postati **usko grlo**.
- **Zasnovani na kvorumu (većinskom glasanju)** – ako dođe do ažuriranja na jednoj kopiji ne zahteva se da sve ostale kopije budu ažurirane, nego da većina kopija bude ažurirana. Osnovna ideja je da klijent koji želi da **modifikuje/pročita podatke** (dakle i PROČITA, ne samo upis) treba da kontaktira **više od polovine ( $N/2+1$ ) od  $N$  servera** da bi mu dali dozvolu pre nego što obavi čitanje/upis repliciranih podataka. **Samo ako se odgovarajući broj servera složi, može se obaviti read ili write.**

- Da bi se fajl ažurirao klijent mora da kontaktira najmanje  $N/2 + 1$  server (većinu) i da dobije dozvole 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 da kontaktira najmanje  $N/2 + 1$  server i tražiti da mu pošalji broj verzije zajedno sa fajlom. Bar jedan server će sadržati novu verziju fajla.

#### Opšta šema glasi:

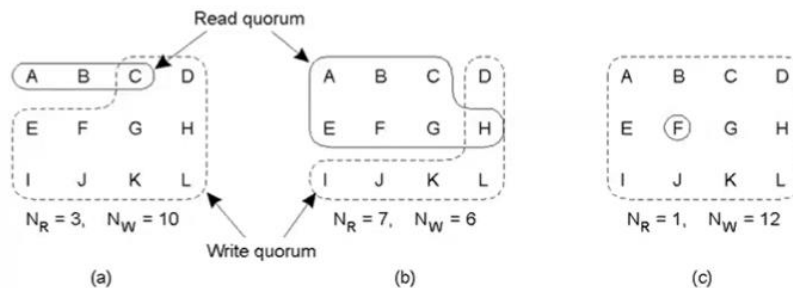
- Ako ima N replika, klijent mora da postigne **read kvorum** na skupu od najmanje **Nr** ili više servera.
- Ako ima N replika, klijent mora da postigne **write kvorum** od najmanje **Nw** ili više servera.

Pri čemu važe uslovi:

1.  $N_r + N_w > N$
2.  $N_w > N/2$

Prvi uslov sprečava *read-write konflikte*, a drugi *write-write konflikte*.

#### 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  $N_W=10$ . Zamislamo 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.



# Otpornost na greške (fault tolerance)

Greške u radu svakog sistema su neminovne, a razlozi za nastajanje grešaka su razni. Što je sistem složeniji, veće su šanse za greške.

**Tolerantnost sistema na defekte (fault tolerance) predstavlja osobinu sistema da korektno funkcioniše uprkos pojavi grešaka.**

- **Delimični otkaz** – nastaje kad jedna komponenta DS-a otkáže i otkaz te komponente može uticati i ne mora na druge komponente.
- **Totalni otkaz** – utiče na sve komponente sistema i dovodi do otkaza sistema.

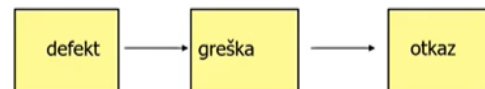
Cilj je projektovati DS tako da ume da razreši otkaze i da nastavi da radi na prihvatljiv način dok se oporavlja od greške. Sistem treba da toleriše greške i da nastavi korektno da radi. Sledeći pojmovi su usko vezani za toleranciju sistema na greške:

- **Raspoloživost** – sistem koji je spreman za korišćenje u trenutku kada je to potrebno
- **Pouzdanost** – osobina sistema da kontinualno radi (duže vreme) bez nastupanja grešaka

Razlika između raspoloživosti i pouzdanosti je u vremenu. Raspoloživost se odnosi na neki konkretan trenutak, a pouzdanost na dug vremenski period.

## Osnovni pojmovi

1. **Defekt** – trajni/privremeni **nedostatak** koji se javlja u nekoj HW ili SW komponenti
2. **Greška** – manifestacija defekta i to je odstupanje vrednosti podatka ili rada sistema od očekivanog rada. **Posledica defekta**. Defekt je uzrok pojave greške. **Defekt ne mora da dovede do greške, ali može.**
3. **Otkaz** – nastupa kada sistem više ne može da radi funkciju za koju je projektovan jer je nastala neka greška/greške u sistemu. **Greška ne mora da dovede do otkaza, ali može.**



**Sistem otporan na greške je sistem koji je u stanju da korektno funkcioniše čak i u prisustvu grešaka.**

**Primer:** U SW sistemu nekorektna instrukcija koja dekrementira vrednost umesto da je inkrementira je **defekt**. Kada se ova instrukcija izvrši ona generiše pogrešnu vrednost – **grešku**. Ako druge naredbe u programu koriste tu nevalidnu vrednost, ceo sistem će odstupati od željenog ponašanja – **otkazaće**.

## Kategorizacija grešaka

U odnosu na trajanje:

- **Prolazne greške** 😊 – pojave se i nestanu
- **Periodične greške** 😞 – greška se pojavi, pa nestane, pa se ponovo pojavi i nestane. Najnezgodniji tip grešaka.



- **Stalne greške** 😞 – postoje sve dok se komponenta ne zameni ispravnom.

Druga kategorizacija grešaka:

- **Mirna greška** 😊 – komponenta prestaje sa radom i ne generiše nikakav rezultat ili generiše poruku o grešci.
- **Vizantijska greška** 😞 – komponenta nastavlja sa radom i generiše pogrešan rezultat. Jako problematične greške. Nemamo jasnu naznaku da li je komponenta otkazala.

## Kako da se sistem učini otpornim na greške?

1. **Redundansa** – primenjuje se nanekoliko različitih nivoa:
  - a. **Informaciona redundansa** – otpornost se postiže repliciranjem ili kodiranjem podataka (**Hammingovi kodovi** – podatku koji se prenosi doda se određeni broj bitova da bi se na određitu detektovala i korigovala greška, **ABFT** – algoritam koji se primenjuje radi detekcije grešaka; **Erasure coding**)
  - b. **Vremenska redundansa (retransmisija)** – otpornost se postiže tako što se izvršenje neke operacije obavi više puta u vremenu. Koriste se *timeout*-i ili *retransmisija* poruka. **Ovaj tip nije od pomoći ako imamo stalne greške!** Koliko god puta ponovimo stalnu grešku ona će uvek biti ista 😊! Ali kod prolaznih ili periodičnih ova tehnika daje rezultate 😊
  - c. **Fizička redundansa** – prava replikacija, kao u normalnom životu (avion ima 4 motora, a dovoljna su 2, ima ih za rezervu; čovek ima 2 bubrega, a dovoljan je 1), na nivou:
    - i. **HW-ska** – komponente se repliciraju da bi se postigla otpornost na greške.
    - ii. **SW-ska** – replicirani procesi (kao kod DS-a)

## Kolika se otpornost na greške može postići?

**100% otpornost na greške se nikada ne može postići! Ali što smo bliže 100% to je sistem skuplji i složeniji. Što je sistem kompleksniji, veći je procenat da nastanu greške.**

Sistem je ***k-tolerantan*** ako može da podnese ***k*** grešaka, a da pri tome nastavi da korektno radi.

- Ako su to **mirne greške**, potrebno je ***k + 1*** komponenta da bi sistem i dalje radio (ako ***k*** otkaže, postoji još **1** koja korektno radi)
- Ako su to **Vizantijske greške**, potrebno je ***2k + 1*** komponenti da bi sistem i dalje radio. ***k*** može da generiše pogrešne rezultate, ali će ***k + 1*** korektno raditi i **većinskim glasanjem** izglasati korektan rezultat.

## Hardverska fizička redundansa

To je tehnika koja koristi fizičku HW redundansu. Najjednostavnija varijanta HW redundanse je **TROSTRUKA MODULARNA REDUNDANSA – TMR**. Tu se koriste 3 primerka komponente da bi se detektovala i korigovala **jednostruka** greška.

Ako komponenta  $i$  ima pouzdanost  $R_i$ , pouzdanost celog sistema je **PROIZVOD POUZDANOSTI SVAKE KOMPONENTE**:

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

U prevodu, verovatnoća da će sistem korektno funkcionisati  $R$  jednaka je verovatnoći da će  $R_1$  korektno funkcionisati, kao i  $R_2$  i  $R_3$  i  $R_4$  itd.

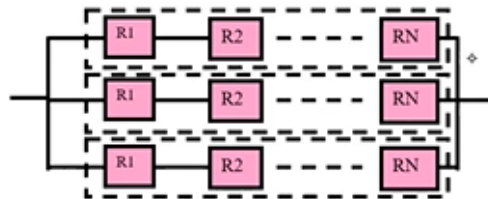
Npr. ako sistem ima 100 komponenti, otkaz bilo koje komponente će izazvati otkaz celog sistema. Ako svaka komponenta ima pouzdanost 0.999, pouzdanost celog sistema je

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

Pouzdanost sistema je **manja** od pojedinačne komponente, jer što je sistem veći veće su šanse za nastupanje greške!

### Trostruka modularna redundansa – TMR

Ako HW repliciramo 3 puta, pouzdanost sistema će se povećati



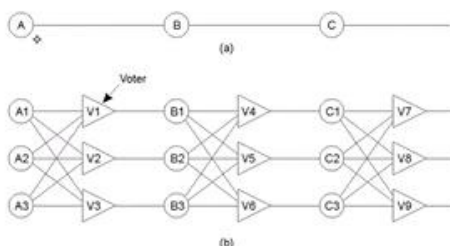
Sada postoje 3 paralelna puta. Ako je pouzdanost na jednom putu  $R_i$ , a **nepouzdanost**  $Q_i = 1 - R_i$ , tada je nepouzdanost celog sistema:

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

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

Sada nakon tripliciranja, pouzdanost je jednaka  **$R = 1 - (1 - 0.905)^3 = 0.9991$** , što je bolje!

## TMR sa glasačima



**Voter** ima funkciju da ako na ulazu dobije **bar dve jednake vrednosti**, onda to izbacuje na izlaz, ali ako dobije **3 (sve) različite vrednosti**, onda nema izlaz.

a) Sistem bez redundantnosti: signal prolazi kroz uređaje A, B i C sekvencijalno. 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.

### Ako element A2 otkáže,

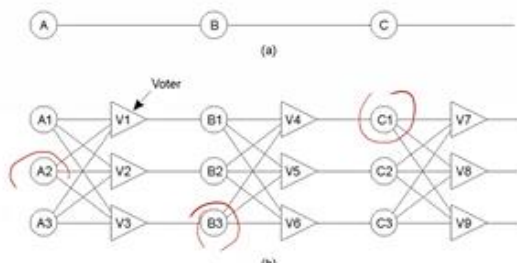
on će voterima V1,2,3 dostaviti svoju vrednost. Pošto A3 i A1 rade normalno, svaki voter će moći da izglasa korektnu vrednost, tako da će na B1, B2 i B3 da se jave vrednosti iz votera, kao da nema greške u A2! Greška u A2 je potpuno maskirana!

### Da li greška u 1 voteru može da se maskira?

Ako se V1 pokvari i izglasa pogrešnu vrednost, da li će sistem ipak da generiše korektnu vrednost. B1 dobija nekorektnu vrednost. Pošto B2 i B3 dobijaju korektne rezultate od svojih votera, opet bi se izglasala korektna vrednost i sistem bi opet radio normalno.

### Da li je ovakav sistem tolerantan samo na jednostruke greške?

PP. da su sada pored A2 otkazali i B3 i C1.



**To je trostruka greška.** Pa opet bi se sve sredilo, jer A2 bi se maskiralo, isto i B3 i C1. To znači da je ovaj sistem tolerantan na trostruke greške! Ili nije? **Trostruka greška znači da gde god nastanu 3 greške u sistemu, sistem će nastaviti da radi.** E pa ovde ne važi tako!

**Neka A1, A2 i A3 greše, gotovo je, sistem otkazuje! Dakle nije otporan na trostruke greške!**

**Sistem je otporan samo na jednostruke greške i to znači da gde god se javi jednostruka greška sistem će moći da nastavi da radi.**

## Informaciona redundansa – ABFT tehnika

Može da se primeni kod određene klase algoritama – **kada su u pitanju izračunavanja**, da se **tolerišu greške u računanju**. Tipično je množenje matrica, LU dekompozicija, inverzije itd.

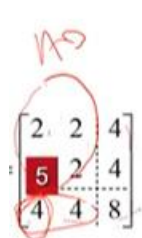
**Primer.** Zadate su dve kvadratne matrice  $A_{n \times n}$  i  $B_{n \times n}$ . Treba da se izračuna njihov **proizvod**  $C_{n \times n}$ .

- Matrica A se proširuje jednom vrstom čiji elementi predstavljaju sumu elemenata odgovarajuće kolone  $\rightarrow A_c$  dimenzija  $(n+1) \times n$
- Matrica B se proširuje jednom kolonom čiji elementi predstavljaju sumu elemenata odgovarajuće vrste  $\rightarrow B_r$  dimenzija  $n \times (n+1)$
- Rezultujuća matrica  $C_f$  dimenzija  $(n+1) \times (n+1)$

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

- **Detekcija grešaka** – kada se dobije  $C_f$ , onda se opet izračunaju sume po vrstama i kolonama za  $C_f$  i ako se poklapa sa dobijenim vrednostima u  $C_f$  onda je sve okej, ako se vrednosti ne poklapaju, onda je detektovana greška. Dakle, naći sume elemenata svake vrste i kolone –  $S_1$ , i uporediti ih sa dobijenim kontrolnim sumama –  $S_2$ . Ako postoje razlike detektovana je greška.
- **Lokacija greške** – u preseku vrste i kolone gde je otkriveno neslaganje kontrolnih suma i suma elemenata
- **Korekcija greške** –  $c = c' + (s_2 - s_1)$

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:  
 $C_{21} = 5 + (4 - 7) = 5 - 3 = 2$



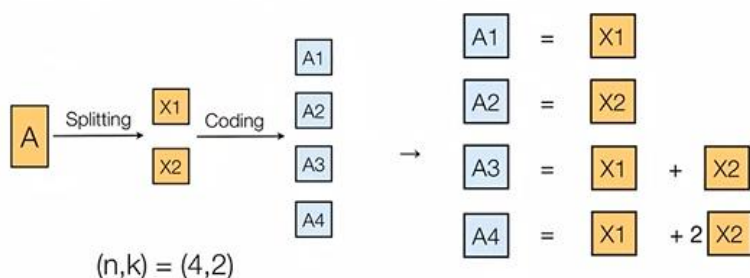
## Informaciona redundansa – Erasure coding

Tehnika za postizanje otpornosti na greške. Kroisti se kod HDF-a (**Hadoop file system**). Tehnika 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.

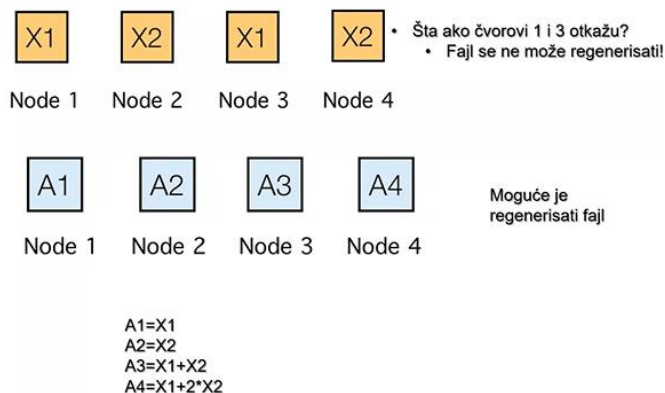
➤  $(n, k)$  erasure code omogućava tolerisanje  $n - k$  grešaka

**Primer.** Imamo fajl veličine  $M$  bajtova. Primenjujemo erasure coding. Delimo fajl na  $k$  blokova veličine  $M/k$ . Primenjujemo  $(n, k)$  code na prvih  $k$  blokova da bi se dobilo  $n$  blokova, svaki veličine  $M/k$ . Broj  $n$  mora biti veći ili jednak  $k$ . Fajl je sada proširen  $n/k$  puta. Ako je  $n$  jednako  $k$  fajl je samo podeljen na blokove, pa nema kodiranja nikakvog.

Neka je fajl podeljen na 2 bloka i formirajmo 4 nova bloka. Neka se 4 bloka pamte u 4 različita čvora.



Bolje je podeliti na 4 čvora nego na 2, jer šta ako čvorovi 1 i 3 otkazu? Fajl bi bio nauršen!

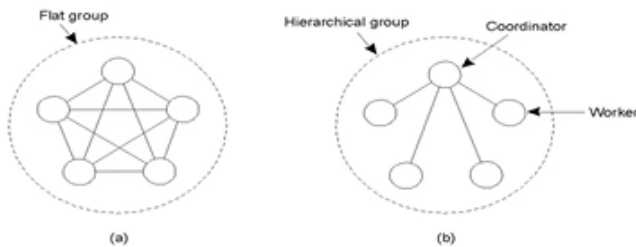


## Kako se otpornost na greške postiže u DS?

**Kroz replikaciju.** Nekoliko identičnih kopija procesa se organizuje u grupu. Kada se neka poruka pošalje grupi, svi članovi grupe je primaju. Ako jedan proces otkáže postoji drugi koji obavlja isti taj posao.

Replicirani procesi mogu biti organizovani u **grupe ravnopravnih procesa (RAVNE GRUPE)** ili u **grupe gde postoji 1 koordinator (HIJERARHIJSKE GRUPE)**:

- ravne grupe – svi procesi su jednaki
- hijerarhijske – postoji jedan koordinator



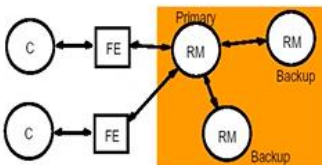
**Ravne grupe** – svi procesi imaju jednaku funkciju i kada se neka komanda/zahtev upiće od strane klijenta, on se prosleđuje svim procesima u grupi.

**Hijerarhijske grupe** – 1 koordinator, a svi ostali su backup procesori (radnici).

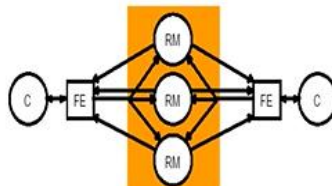
- **Pasivna replikacija** - Replikacija može da se bazira na primarne i backup kopije. Koristi se kod **hijerarhijski organizovanih grupa**. Primarni server (**koordinator**) obavlja ceo posao. Ako primar otkáže, jedan od backup servera preuzima posao. Otkaz primara treba da bude nevidljiv za aplikativne programe.  
Kako backup zna da je primar otkazao? Stalno se ispituje **heartbeat** od primara (koordinatora) – sekundari (radnici) šalju poruku primaru „**Da li si živ?**“. Ako se ne dobije odgovor od primara u okviru **timeout**-a, zna se da je primar mrtav. Sada se bira novi primar iz skupa backup servera.  
Kako se bira koji će backup server da bude primar? Vrš se **konsenzus**. Postizanje konsenzusa oko toga ko je novi koordinator (ili oko nečega drugog) može biti veoma teško i da zahteva razmenu ogromne količine poruka (loše performanse).
- **Aktivna replikacija** – koristi se kod **grupe ravnopravnih procesa**. Klijentski zahtev se prosleđuje svim procesima u grupi. Svi procesi u istom redosledu obrađuju klijentski zahtev – **potpuno uređena grupna komunikacija**.

### \* Pasivna replikacija

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



### \* Aktivna replikacija



Kod pasivne replikacije postoji primar i radnici i razlika u odnosu na aktivni je u tome što klijent upućuje komandu primaru, a primar je šalje backup serverima, da u slučaju otkaza backup server može da preuzme ulogu primara. Kod aktivne replikacije klijent zahtev upućuje svim replikama, sve replike obavljaju datu aktivnost i sve replike vraćaju odgovor. Klijentski interfejs koristi većinsko glasanje da prosledi tačan odgovor klijentu.

**Prednost pasivne replikacije** je jednostavniji dizajn, jer je teorijski dovoljan samo 1 backup server. Nema problema sa sinhronizacijom, jer se poruka šalje samo 1 serveru (primaru). Međutim, šta ako otkáže primar i taj 1 backup server? Onda nije dovoljan 1 backup server. Aktivna replikacija ima problem sinhronizacije.

**Mana** je što loše rade sa Vizantijskim greškama, primar može pogrešno da radi, a da greška ne bude otkrivena. To rešava aktivna replikacija.

## Usaglašavanje u prisustvu grešaka

Kako postići konsenzus ako neki procesi u grupi ne rade kako treba ili je KK nepouzdan?

Usaglašavanje je potrebno postići u konačnom broju koraka! Posmatraju se 2 odvojena scenarija:

- 1) Ispravni procesi, a nesavršeni komunikacioni kanali (**PROBLEM DVE ARMIJE**)
- 2) Nesavršeni procesi (mogu da greše), a pouzdani komunikacioni kanali (**PROBLEM VIZANTIJSKIH GENERALA**)

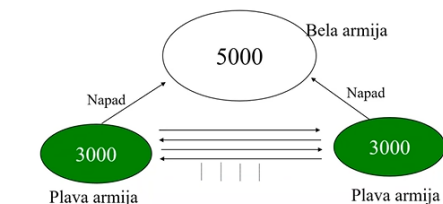
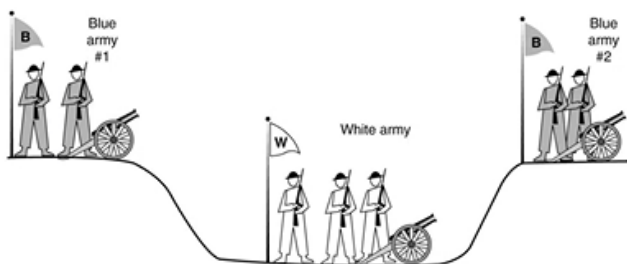
## Problem dve armije (problem višestrukih potvrda)

Procesi rade korektno, greške nastupaju u KK-u.

Dve divizije plave ramije, B1 i B2, treba da koordinišu napad na drugu armiju, W.

B1 i B2 imaju po 3000 vojnika, a armija W 5000.

- Ako se B1 i B2 dogovore da zajedno napadnu, mogu da pobeđu belu armiju (W)



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

Ako B1 sama krene na W, ili B2 krene sama na W neće pobediti jer W ima više vojnika od B1/B2. Ali ako se B1 i B2 dogovore da zajedno napadnu W, mogu da pobeđu, zajedno ih ima više.

Da bi se B1 i B2 dogovorili, koriste **kurira** koji šalje poruku tipa „*Hajde da napadnemo W u toliko i toliko sati*“. Kurir na putu prolazi kroz neprijateljsku teritoriju i rizikuje da bude zarobljen. Scenario bi bio da kurir krene iz B1 stigne do B2 i da poruku, B2 sada daje odgovor za B1. Kurir ide do B1. Sada B2 ne zna da li je B1 primio potvrdu, pa mu B1 šalje kurirom potvrdu da je dobio poruku, pa B2 odgovara i na to preko kurira. Sve to ide u nedogled. Kurir svaki put rizikuje da bude zarobljen.

Ovo pokazuje da ne postoji protokol koji može da reši problem nepouz danih kanala. Nikada ne možemo biti 100% sigurni da je poruka prenet a na drugu stranu.



## Problem vizantijskih generala

Nepouzdana procesa, a pouzdane komunikacije. Inspirirano je ratovima u Vizantijskom carstvu, jer niko nije mogao verovati nikome, bilo je puno zavera i izdajica.

Neka postoji  $n$  generala i svaki je sa svojom divizijom i treba da donesu odluku da li da napadnu neprijatelja ili ne. Komunikacije su pouzdane (radio/telefon). Međutim, od  $n$  generala, njih  $m$  su izdajnici. Oni pokušavaju da spreče ostale generale da postignu konsenzus šaljući im netačne i kontradiktorne informacije.

**Da li lojalni generali mogu postići konsenzus (da donesu istu odluku) uprkos svemu? Tj. koliko je moguće izdajnika tolerisati da bi se donela ispravna odluka?**

- Svaki general šalje svoj predlog – N (napad) ili P (povlačenje) – svim ostalim generalima, a i od svih ostalih prima predloge. Svaki general donosi odluku o tome da li će izvršiti N ili P na osnovu prikupljenih predloga od svih generala. Kako da svi lojalni generali izglasaju isti predlog?

### Lamportov algoritam za problem vizantijskih generala

Postoji Lamportov algoritam za ovaj problem i on radi pod određenim uslovima. Da bi moglo da se toleriše  **$m$  izdajnika** mora postojati najmanje  **$3m + 1$  generala ukupno**, odnosno mora biti barem  **$2m + 1$**  lojalnih generala. Dakle, više od  $2/3$  generala mora biti lojalno. Međutim, ovaj algoritam je veoma skup i nije efikasno koristiti ga.

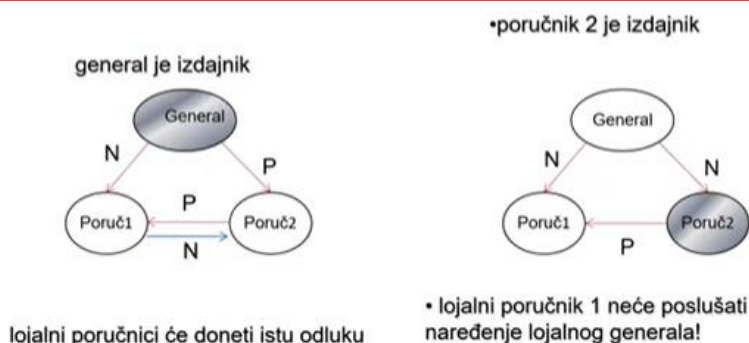
#### Dokaz.

Neka je 1 general i  $n-1$  poručnika. Od poručnika njih  $m$  mogu biti izdajice ili general i  $m-1$  poručnika.

Lojalni general šalje svim poručnicima istu poruku (N ili P). Svaki poručnik primljenu poruku prosleđuje ostalim poručnicima. Kada se sve poruke prikupe primenjuje se većinsko glasanje. **Default vrednost je P**, ako ništa ne može da se izglasa.

Želimo da postignemo sledeće:

1. Svi lojalni poručnici moraju da donesu istu odluku
2. Ako je general lojalan, tada svi lojalni poručnici poštuju njegovo naređenje



Ako imamo 3 generala i 1 izdajnika (dakle, ako nije ispoštovano pravilo od minimum  $2m + 1$  lojalni):

**Prvi sličaj sa slike:** neka je **general** izdajnik i šalje različite poruke različitim poručnicima – poručniku1 šalje N, a poručniku2 šalje P. Sledeće što se dešava je da poručnik1 i poručnik2 pošalju svoje primljene

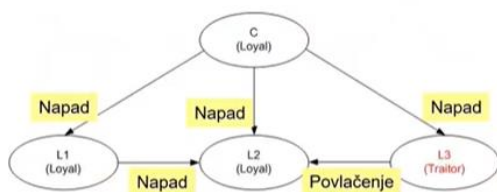


poruke svim ostalim poručnicima (a to su 1 i 2). Kada pogledaju sve poruke koje su primili, to su N i P. Da li mogu da donesu odluku? Naravno da ne. Zato se koristi default vrednost i daju zajednički odgovor P.

**Drugi slučaj:** neka je **poručnik2** izdajnik. U tom slučaju je general lojalan i šalje svim poručnicima istu poruku N. Poručnik1 prima N, kao i poručnik2. Poručnici sada šalju šta su dobili jedan drugom, sa tim da je poručnik2 izdajnik i šalje P umesto N. Šta će reći lojalni poručnik? Ne može većinski da izglasa odluku i reći će da je odluka P. ALI SVAKI LOJALNI PORUČNIK POŠTUJE GENERALOVO NAREĐENJE! General je rekao N, a on je zaključio P, dakle ne poštuje generalovo naređenje.

**Prema svemu do sada, jasno je da mora postojati najmanje  $2m + 1$  lojalnih generala i da je minimalni broj generala ukupno  $n = 3m + 1$ .**

**Primer.** Neka je sada ispoštovano pravilo. Imamo  $n=4$ ,  $m=1$ .



\*  $n=4$  generala;  $m=1$  izdajnik

\* L2 računa majority(Napad, Napad, Povlačenje) = Napad

General je lojalan i šalje svima isto - N. Izdajnik je general L3 pa će poslati ostalima P umesto N. Ali svi drugi šalju N kao što treba, jer su lojalni.

Ako gledamo L2 on je dobio sledeće: N, N i P. Da li može da izglasa većinski? Može. To je N, što je ispravna odluka.

Ako gledamo L1, on će dobiti N (general), N (L2) i P (L3). Dobiće N, što i treba.

**Dakle, lojalni generali doneće istu odluku 😊**

Ovaj algoritam je **rekurzivne** priorde, ali je problem što mora da se razmeni ogroman broj poruka da bi funkcionisao što je nepovoljno.

# Otpornost na greške – RPC

Cilj RPC-a je da sakrije komunikaciju tako što pokušava da poziv udaljene procedure izgleda kao poziv lokalne procedure. Ako nastane greška, nije uvek moguće sakriti razlike između udaljene i lokalne procedure.

Greške koje mogu nastati u RPC sistemu:

- Greška 1: Klijent ne može da locira server
- Greška 2: Zahtev upućen od klijenta ka serveru je izgubljen
- Greška 3: Došlo je do otkaza servera nakon prijema zahteva od klijenta
- Greška 4: Odgovor servera ka klijentu je izgubljen

Svaka vrsta greške izaziva različite probleme i zahteva različita rešenja.

## Greška 1: Klijent ne može da locira server

Razlog može biti:

- otkaz servera ili
- neodgovarajuća verzije klijentskog stub-a.

Kada nastane ovakva greška mora da se emituje **exception**. Programer piše posebnu proceduru koja se poziva kada se detektuje određena greška. Ovako se gubi transparentnost distributivnosti!

## Greška2: zahtev upućen od klijenta ka serveru je izgubljen

Ovo je greška koja se najlakše rešava – prosto se izvrši retransmisija zahteva.

- Klijent startuje timeout kada uputi zahtev
- Ako istekne timeout pre nego što stigne odgovor ili potvrda, vrši se retransmisija

Ako zahtev nije bio izgubljen, onda je ili došlo do **otkaza servera** ili je **izgubljen odgovor od servera**. Ako je ovaj drugi slučaj – izgubljen odgovor servera, onda **server mora da bude u stanju da prepozna retransmisiju**. Prva varijanta za to je da postoji poseban bit u zaglavlju RPC poziva koji se postavlja na 1 ako je reč o retransmisiji. Druga varijanta je da se numerišu pozivi udaljenih procedura i da, u slučaju da nije bio izgubljen klijentski zahtev nego serverski odgovor, server ume da prepozna da je u pitanju retransmisija.

## Greška 3: Otkaz servera nakon prijema zahteva od klijenta

Server može da otkaze u bilo kom trenutku, ali razlikujemo **2 moguće situacije**:

- 1) Server primi zahtev za izvršenje udaljene procedure, obavi tu proceduru i onda dođe do otkaza servera. Klijent nije primio dogovor od servera.
- 2) Server primi klijentski zahtev i onda dođe do otkaza servera. Opet klijent ne dobija nikakav odgovor.

Ove situacije deluju slično, ali zahtevaju sasvim drugačije akcije na strani servera. Klijent ne može da razlikuje ove dve situacije, jer on vidi samo istek timeout-a.

U prvom slučaju klijent samo treba da emituje exception, a u drugom slučaju klijent bi trebalo da obavi retransmisiju (ponovi svoj klijentski zahtev). Ali **problem je što klijent ne razlikuje ove dve situacije**. Ono što je idealno rešenje je **exactly one metoda**, da se udaljena procedura izvrši tačno jednom, ali to ne može da se garantuje.

#### **Postoje 2 rešenja za to:**

- **At least one:** Klijent šalje zahteve sve dok ne dobije odgovor (garantuje da će se udaljena procedura izvršiti bar jednom, a možda i više puta). Ova opcija je prihvatljiva kod idempotentnih funkcija (koje ne menjaju stanje sistema) – npr procedura koja vraća tačno vreme.
- **At most one:** Odmah signalizirati grešku (exception) nakon isteka timeout-a (udaljena procedura će se izvršiti najviše jednom, a možda ni jednom). Ova opcija je pogodna za procedure koje menjaju stanje sistema – procedura koja umanjuje stanje novca na računu.

**Primer.** Pretpostavimo da udaljena procedura zahteva štampanje teksta na udaljenom štampaču preko servera za štampanje. Kada klijent pošalje zahtev, štampač prima potvrdu da je zahtev prosleđen serveru. **Server ima 2 mogućnosti:**

- Čim dobije zahtev klijenta šalje potvrdu da je primio zahtev, pre nego što obavi štampanje
- Server odgovori klijentu tek nakon što obavi štampanje

Pretpostavimo da je **server otkazao** i zatim se **oporavio od otkaza** i **obaveštava klijenta da ponovo radi**. Problem je što klijent ne može da zna da li je njegov zahtev za štampu bio obavljen ili ne i postoje 4 strategije koje klijent može da primeni:

- 1) Klijent **nikada ne ponavlja zahtev**, po cenu da nikada ne dobije odštampan tekst
- 2) Klijent **uvek ponavlja zahtev**, što može da dovede do toga da se tekst odštampa više puta
- 3) Klijent **ponavlja zahtev ako nije primio potvrdu od servera**
- 4) Klijent **ponavlja zahtev samo ako je primio potvrdu za prethodno izdati zahtev**

Na strani servera mogu da nastanu 3 događaja:

- 1) Slanje poruke o okončanju zadatka – **M**
- 2) Štampanje teksta – **P**
- 3) Otkaz servera – **C**

Ovi događaji mogu da nastupe u 6 različitih redosleda (u zagradama [] je ono što se neće izvršiti):

- 1)  $M \rightarrow P \rightarrow C$ : otkaz servera se desio nakon slanja potvrde i štampanja teksta
- 2)  $M \rightarrow C [\rightarrow P]$ : otkaz servera nakon slanja potvrde, a pre štampanja teksta
- 3)  $P \rightarrow M \rightarrow C$ : otkaz servera nastao nakon štampanja i slanja potvrde
- 4)  $P \rightarrow C [\rightarrow M]$ : tekst odštampan, server otkazao pre slanja potvrde
- 5)  $C [\rightarrow P \rightarrow M]$ : server otkazao pre nego što je bilo šta uradio
- 6)  $C [\rightarrow M \rightarrow P]$ : server otkazao pre nego što je bilo šta uradio

Zaključak je da ne postoji ni jedna kombinacija klijent i server strategija koja će garantovati da će štampanje da se obavi samo jednom za svaku kombinaciju događaja na server strani, jer klijent nikada ne može da zna da li je tekst odštampan pre nego što je server otkazao ili ne.

**OK** – štampa se tačno jednom, **DUP** – duplirano, **ZERO** – ni jednom se ne štampa

Klijent	Server					
	Strategija $M \rightarrow P$			Strategija $P \rightarrow M$		
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

Levi (zeleni) deo slike je tabela koja pokazuje ponašanje klijenta, a desna (crvena) ponašanje servera.

**Objašnjenje prve tabele:** Strategija  $M \rightarrow P$  je strategija kada server po prijemu klijentskog zahteva odmah šalje potvrdu, pa tek onda štampa. Ovo su akcije koje je server mogao da uradi pre otkaza:

**MPC** – server je obavio sve 3 akcije, samim tim i štampanje

**MC(P)** – server je samo poslao potvrdu i otkazao

**C(MP)** – server je samo otkazao, ništa drugo nije uradio

- Ako klijent **uvek** kad dođe do otkaza servera **ponavlja zahtev**, a akcija servera pre otkaza je bila:
  - **MPC** – onda će doći do dupliranja, jer je server već odštampao pa je otkazao i sad opet dobija isti zahtev – **DUP**
  - **MC(P)** – prvi put ne štampa, ali šalje potvrdu, pa se javi klijentu da je dostupan opet i klijent opet šalje i ovaj put se tekst štampa – **OK**
  - **C(MP)** – prvi put je server otkazao pre prijema zahteva, kad se vrati on ne javlja klijentu da je dostupan, ali klijent svakako ponavlja zahtev zbog isteka timeout-a i sada se štampa – **OK**
- Ako klijent **nikad** ne ponavlja svoj zahtev, a akcija servera pre otkaza je bila:
  - **MPC** – server je već odštampao (tako da imamo jednom odštampano), pa je otkazao, pa se vratio i javio klijentu da je dostupan, ali klijent ne šalje ponovo zahtev – **OK**
  - **MC(P)** – server je otkazao pre štampanja, ali je primio zahtev i kada se vrati javi se klijentu, međutim on nikada ne ponavlja svoj zahtev tako da se tekst ne štampa uopšte – **ZERO**
  - **C(MP)** – isto kao i gore, s tim da server nije primio zahtev klijenta pa mu ni ne javlja – **ZERO**
- Ako klijent vrši ponavljanje zahteva **samo ako je primio potvrdu servera da je ponovo u funkciji**, a akcija servera pre otkaza je bila:
  - **MPC** – doći će do dupliranja jer je server već odštampao – **DUP**
  - **MC(P)** – doći će do jednog štampanja jer je prvi put otkazao pre štampanja – **OK**
  - **C(MP)** – neće se ništa odštampati, jer je server – **ZERO**

- Ako klijent vrši ponavljanje **samo ako nije primio potvrdu servera**, a akcija servera pre otkaza je bila:
  - **MPC** – onda će server poslati potvrdu, tekst će se jednom odštampati, pa server otkazuje, zbog te potvrde klijent ne šalje zahtev ponovo - **OK**
  - **MC(P)** – server šalje potvrdu, ali ne štampa, klijent ne šalje retransmisiju jer je primio potvrdu i tekst se nikad neće odštampati – **ZERO**
  - **C(MP)** – server odmah otkáže i ne šalje potvrdu, pa će klijent vršiti retransmisiju i server će odštampati tekst exacty one – **OK**

#### Objašnjenje druge tabele:

Strategija **P->M** je strategija servera koji prvo obavlja štampanje pa onda šalje potvrdu klijentu. Sve se vidi iz same tabele, neću dodatno objašnjavati.

### Greška 4: Odgovor servera za klijenta je izgubljen

Klijent ne zna koji je tip greške, ali zna da je istekao timeout. Nakon isteka timeout-a klijent može da izvrši retransmisiju (ponovi svoj zahtev) ili da ne ponovi svoj zahtev. Ako se radi o idempotentnim aktivnostima, onda je okej da se obavi retransmisija, ali ako se radi o akciji koja menja stanje sistema onda ne treba vršiti retransmisiju.

Pošto je server obavio svoj posao, ali se odgovor izgubio, i sad mu stigne retransmisija, kako da server zna da li je to novi zahtev ili je retransmisija već obrađenog zahteva?

- Jedno od rešenja je da se dodaju redni brojevi klijentskim zahtevima. To zahteva da server mora da vodi računa o rednim brojevima svih klijentskih zahteva da bi mogao da otkrije duplikat zahtev. Tako se dosta komplikuje stanje na serveru. Ako je **statefull server** i dođe do otkaza pa se posle vrati, stanje servera pre otkaza se beleži trajno na disku. Ako je **stateless server**, on ne vodi računa o stanju sistema, nakon otkaza, server nema stanje pre otkaza, samim tim ni redbe brojeve klijentskih zahteva.
- Drugi predlog je da se u zaglavlju zahteva postavi bit za retransmisiju da bi se razlikovali novi zahtevi od ponovljenih. To rešenje ne zahteva da server vodi računa o rednim brojevima klijentskih zahteva.

U oba slučaja, server mora da opet pošalje klijentu odgovor na zahtev jer će klijent vršiti retransmisiju sve dok ne dobije odgovor ili (ako ne radi retransmisiju) će se kod klijenta javiti **exception**.

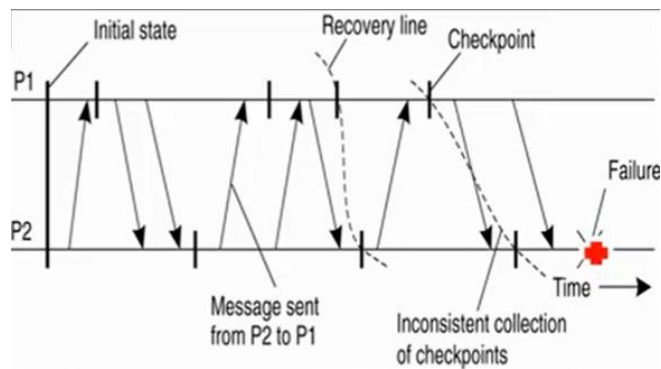
# Strategije za oporavak od greške

Kada nastupi greška mora da se izvrši oporavak od greške i da se vrati sistem u korektno stanje. Postoje 2 strategije za oporavljanje od greške:

- 1) **Backward Recovery** – Povratak u stanje pre nastupanja greške (vratiti sistem u neko prethodno korektno stanje pomoću tačaka provere – **checkpoints**). Nedostatak je što beleženje stanja ubacivanjem checkpoint-a može biti jako skupo sa stanovišta performansi, ali pored toga se ipak najčešće koristi u oporavku od greške.
- 2) **Forward Recovery** – Prelazak u novo stanje (prevesti sistem u korektno novo stanje, kao da nije nastala greška). Da bi mogla da radi ova strategija moraju da se identifikuju sve moguće greške koje mogu da nastupe u sistemu i da se shodno tome definiše kako bi izgledalo novo stanje. Tehnika koja se mnogo ređe koristi.

## Backward Recovery

Svaki proces u DS-u periodično kreira checkpoint-e u kojima beleži stanje sistema (vrednosti promenljivih, primljene/poslate poruke, stanje memorije itd). Ako nastupi greška u sistemu proces se vraća u stanje pre nastupanja greške na osnovu podataka u checkpoint-u.



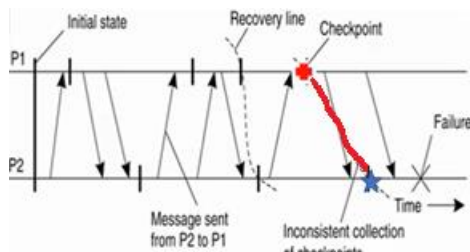
Ako u procesu P2 u obeleženom trenutku nastupila greška, neophodno je vratiti sistem u stanje pre nastupanja greške. Potrebno je vratiti i stanje u procesu P1, u trenutak pre nastupanja greške.

Da bi oporavak od greške bio moguć u DS-u, svi procesi moraju da se vrte stanje odakle je moguće izvršiti oporavak od greške – mora se naći **linija oporavka (konzistentni presek)**. **Konzistentni presek je skup tačaka provera (checkpoint-a) u različitim procesima koje omogućavaju da se od njih sistem restartuje i krene dalje sa radom.**

Skup checkpointa C je **konzistentan** ako za sve događaje u sistemu e i e' važi sledeće:

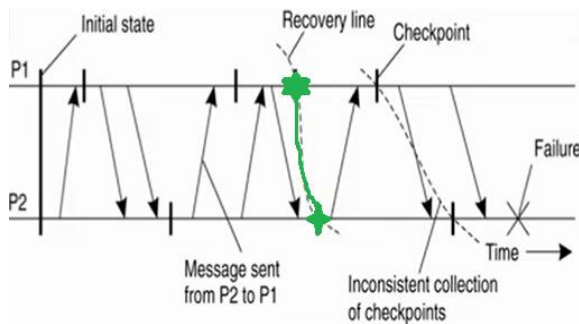
$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

ako je e događaj koji je zabeležen skupom checkpoint-a C i ako je događaj e' nastao pre događaja e, tada u skupu checkpoint-a mora da bude tačka koja je zabeležila e'. U prevodu, ako je zabeležen prijem poruke iz nekog procesa u skupu checkpoint-a, onda u njemu mora da bude i događaj koji je zabeležio slanje te poruke. Ne može da se desi da imamo checkpoint koji beleži prijem poruke, a da nije zabeleženo slanje te poruke iz nekog drugog procesa. Obrnuto može da se desi, da neki proces zabeleži slanje poruke, a da neki drugi proces u svom checkpoint-u nije zabeležio slanje te poruke.



Da li bi označena linija mogla da bude linija oporavka? Neka je proces P1 u **označenom** trenutku kreirao checkpoint i zabeležio je stanje sistema, poruke koje je primio i poslao. A neka je proces P2 u **označenom** trenutku ispod kreirao svoj checkpoint, da li bi onda ova dva checkpoint-a mogli da se iskoriste kao linija oporavka?

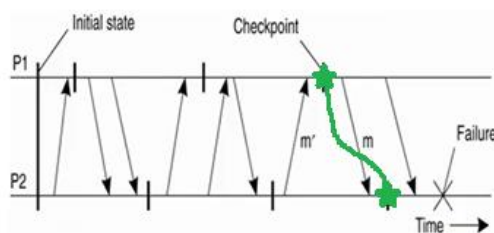
**Ovo nije validna linija oporavka, zašto? Checkpoint-om procesa P2 zabeležen je prijem poruke iz procesa P1, a checkpoint za proces P1 nije zabeležio slanje te poruke!**



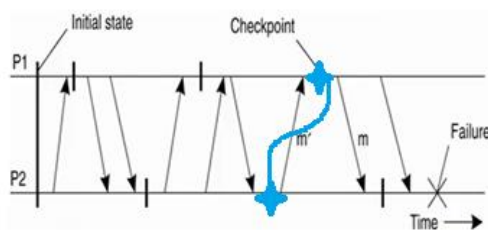
Da li bi **označena** linija mogla biti konzistentni presek? Da, ona jeste konzistentni presek (linija oporavka) jer tačke jesu konzistentne – proces P1 ima checkpoint nakon što je zabeležio slanje poruke, a za P2 nije bitno da li je zabeležio prijem, iako jeste.

### Backward Recovery - Domino efekat

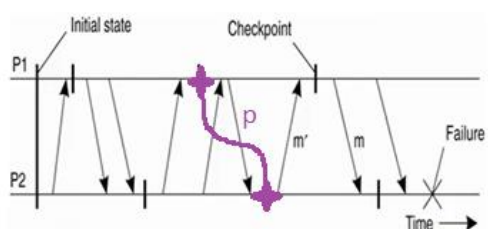
Ako procesi **nezavisno** jedan od drugog donose odluku gde će kreirati checkpointe, može biti vrlo teško naći liniju oporavka i često se javlja **domino efekat**. Tada sistem mora da se vrati u početno stanje. Evo jednog takvog primera:



Neka su procesi P1 i P2 kreirali svoje checkpoint-e i neka je u trenutku označenom na slici sa **Failure** došlo do greške. Da li poslednja **dva checkpoint-a pre greške** čine konzistentan presek? Ne, jer proces P1 nije zapamtio da je poslao poruku m, a proces P2 je zapamtio da je primio poruku m.



Da li možemo uzeti **sledeća dva checkpoint-a** kao konzistentni presek? Ne, jer proces P1 beleži primanje poruke m' dok proces P2 ne beleži slanje te poruke.



Da li ove **dve tačke** mogu da formiraju konzistentni presek? Opet ne, jer checkpoint iz P1 ne pamti slanje poruke p, a P2 checkpoint pamti prijem poruke p. I tako važi za sve checkpoint-e, sve do početnog stanja. Problem sa beleženjem stanja je taj što ako svaki proces za sebe donosi odluku kada kreira checkpoint, onda može doći do domino efekta. POSTOJI REŠENJE - **KOORDINATOR!**

Rešenje je da beleženje stanja bude **koordinisano**, za šta je potreban **centralni koordinator** koji definiše kada svaki proces treba da kreira svoj checkpoint. Primenuje se dvofazni blokirajući protokol gde **koordinator periodično svim procesima u grupi** šalje poruku **checkpoint\_REQUEST** kojom se zahteva kreiranje checkpoint-a od strane procesa. Kada proces primi ovakvu poruku, prestaje sa svojim aktivnostima, kreira svoj checkpoint i beleži svoja stanja. Onda javlja koordinatoru da je završilo kreiranje checkpoint-a sa **checkpoint\_ACK**. Kada koordinator primi **sve potvrde**, on opet šalje svim procesima **checkpoint\_DONE** čime se svi procesi **deblokiraju** i nastavljaju sa radom. Zapamćeno stanje u svim procesima je **konzistentno** i izbegnut je domino efekat. Jedini problem je postojanje koordinatora koji mora da vodi računa o svemu, ali je to jedini način.



# Peer-to-peer sistemi

Kada je reč o SW-skoj arhitekturi DS sistema, postoji nekoliko načina za organizaciju rada tih sistema:

- **Klijent-Server arhitektura**
- **Peer-to-peer (P2P) arhitektura**
- **Hibridna arhitektura (Klijent-server + P2P)**

Kod Klijent-server arhitektura u centru aplikacije je Server koji je stalno prisutan i ima stalnu IP adresu. Sa druge strane, klijenti se periodično povezuju u mrežu, svaki put kada se povežu mogu imati drugu IP adresu, komuniciraju sa serverom i nemaju mogućnost da direktno međusobno komuniciraju.

Kod Peer-to-peer arhitektura nema Servera, krajnji čvorovi mogu direktno da komuniciraju. Svaki čvor u sistemu može da bude i klijent i server i to istovremeno, dakle čvorovi su jednaki. Pošto su svi učesnici jednaki odatle sledi naziv **peer**. Peer-ovi se poveremeno konektuju na mrežu i svaki put mogu imati drugu IP adresu. Takve arhitekture su visoko skalabilne i proširljive su jer mogu obuhvatiti veći broj čvorova. Što je veći broj čvorova, performanse su bolje, međutim upravljanje je jako komplikovano jer ne postoji centralna komponenta koja upravlja radom.

- Aktivni čvorovi formiraju **overlay apstraktnu logičku mrežu** u kojoj imamo **čvorove** i **potege**. Važe sledeća pravila:
- Ako peer X ima TCP konekciju (na transportnom nivou) sa peer-om Y, tada na overlay mreži postoji **potez** između X i Y.
- Svi aktivni peer-ovi i potezi čine overlay mrežu.
- Potez ne predstavlja fizičku vezu.

U zavisnosti od toga kakva je ta overlay mreža po strukturi, razlikuju se **2 vrste P2P sistema**:

- 1) **Strukturirani P2P sistemi** – čvorovi formiraju overlay mrežu koja ima pravilnu strukturu (prsten, stablo ili slično)
- 2) **Nestrukturirani P2P sistemi** – overlay mreža koja nema pravilnu strukturu

Svaki čvor koji je član P2P sistema donira svoje resurse sistemu – deli fajlove, donira procesorsko vreme itd. Razlozi projektovanja P2P sistema: iskorišćenje ogromne računarske moći na obodu jedne računarske mreže, smeštajni kapaciteti i deljenje različitih informacija i resursa među čvorovima.

## Osnovni problem: Pronalaženje željenog sadržaja među peer-ovima

Osnovni problem kod P2P sistema je efikasno lociranje čvorova koji sadrže tražene podatke. Kako čvor u P2P sistemu zna koji čvor sadrži podatke koji su mu potrebni?

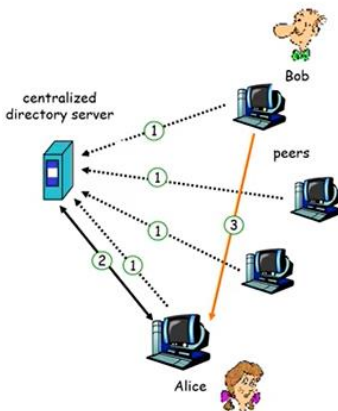
Mehanizmi za lociranje podataka mogu biti (ima ih 3):

1. **Generacija: Centralizovani mehanizam** – koristi se 1 ili više servera koji sadrže **centralne direktorijume** gde se registruju svi peer korisnici i gde se registruju sadržaji koje ti peer korisnici žele da dele sa drugima. Ovo nije pravi P2P sistem već hibridni, jer postoji server.
2. **Generacija: Decentralizovan nestrukturiran mehanizam** – svaki peer za sebe zadržava tabelu sa sadržajem koji je spreman da deli sa drugima. Overlay mreža je **nestrukturirana**, a da bi se pronašao željeni sadržaj u takvoj mreži koristi se **ograničena bujica**.

3. **Generacija: Decentralizovani struktuirani mehanizam** – overlay mreža je struktuirana i reč je o sistematskom pronalaženju sadržaja, ne koristi se bujica nego posebni algoritmi. Koriste se tablice slične tablicama rutiranja, međutim za pretraživanje se ne koriste IP adrese već ključevi koji se dobijaju primenom hash funkcija nad imenom fajla koji se pretražuje. Primer takvog sistema je **Chord** sistem.

## Centralizovan P2P

- **Napster** – najstarija app ovog tipa. Osnovna ideja je bila da se dele mp3 fajlovi. Pronalaženje fajlova je bilo centralizovano, a prenos fajlova je bio Peer-to-peer direktno između dva čvora u mreži.



Kada se peer konektuje, on centralnom serveru daje informacije: **svoju IP adresu i sadržaj koji želi da deli**. Da bi se ostvario prenos fajla između dva peer-a mora se kontaktirati centralni direktorijum odakle se dobija IP adresa peer-a koji sadrži željeni sadržaj, a nakon toga se uspostavlja direktna konekcija između ta dva čvora.

**1** – Bob se konektuje i daje svoj ID i sadržaj koji želi da deli centralnom serveru. Isto i Alisa to radi.

**2** – Alisa traži pesmu **Hey Jude** od centralnog servera jer on zna koji peer ima tu pesmu i koja je njegova IP adresa

**3** – Alisa sada zna IP adresu Boba i onda zahteva fajl od njega

Ono zbog čega je došlo do gašenja Napstera je bilo to što su se produkcijske kuće žalile da ugrožava autorska prava i sud je doneo odluku da Napster server mora da bude ugašen. Napster se branio da njihov server ne sadrži nikakve fajlove i tako ne krši autorska prava, ali zato sadrži informacije o tome ko sadrži fajlove od interesa, što je bilo dovoljno da se ceo sistem sudski zatvori. **Dakle, prenos fajlova je decentralizovan, ali je lociranje sadržaja visoko centralizovano!**

- **Skype** – centralni server za registrovanje korisnika i on služi za pronalaženje osobe sa kojom želimo da komuniciramo, a veza se uspostavlja direktno između peer-ova pri komunikaciji.

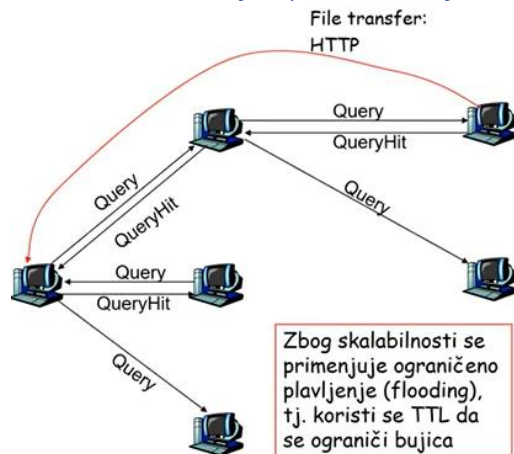
Kada aktivni peer dobije novi objekat ili ukloni objekat, on informiše centralni server koji zatim ažurira svoju bazu podataka – centralni direktorijum. Peer-ovi se povezuju u mrežu po svojoj volji, kao i što se diskonektuju po svojoj volji, pa zato server mora periodično da proverava da li su peer čvorovi još uvek prisutni u mreži. Ako server ustanovi da neki peer ne odgovara na heartbeat poruku „Jesi li živ?“ onda će ga server ukloniti, kao i njegov sadržaj.

Osnovni problem ovog prilaza je **slaba otpornost na greške – otkaz centralnog direktorijuma je fatalan za ceo sistem!** Centralni direktorijum može da postane usko grlo, jer broj korisnika može biti ogroman.

## Decentralizovan nestruktuiran P2P - potpuno distribuiran: *Gnutella*

Gnutella model je nastao kao odgovor na ukidanje Napstera. Nema centralnog servera, nema beleženja informacija na jednom mestu o tome ko šta poseduje. Aplikacija je besplatna i koristi poseban **Gnutella protokol**. Formira se **nestruktuirana overlay mreža**.

### Gnutella: lociranje i pretraživanje sadržaja



Peer šalje poruku sa upitom preko postojećih TCP konekcija. Svaki peer u Gnutella mreži sadrži tabelu sa sadržajima koje želi da deli sa drugima.

Da bi peer A pronašao željeni sadržaj u mreži, on šalje poruku sa upitom za pronalaženje željenog sadržaja. Poruka se prosleđuje prema peer-ovima sa kojima peer A ima uspostavljenu TCP konekciju tj. ima poteg. Svaki peer koji primi taj upit prosleđuje ga dalje peer-ovima sa kojima je "povezan". Primenuje se **ograničena bujica** – postoji polje **TimeToLive (TTL)** koji se dekrementira svaki put kada prođe kroz neki peer. Svaki peer koji u sebi nađe željeni sadržaj, odgovara peer-u A da postoji pogodak u

pretraživanju. Nakon što se okonča bujica, peer A odlučuje sa kojim peer-om će uspostaviti konekciju (ako je nema) da bi skinuo željeni fajl.

**MANE:** Gnutella koristi ograničenu bujicu za pronalaženje željenog sadržaja i generiše relativno veliki saobraćaj. **Zbog ograničene bujice, može se desiti da se željeni sadržaj ne pronađe iako postoji u sistemu.**

## Decentralizovan nestruktuiran hijerarhijski organizovan model: *KaZaA*

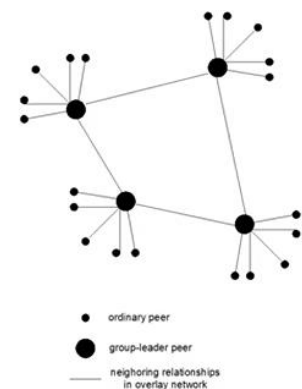
KaZaA je P2P aplikacija kod koje **peer-ovi nisu svi jednaki**, postoje **lideri grupa**. Koristi ideje i od Napstera i Gnutelle.

- Od Gnutelle uzima ideju za pronalaženje sadržaja, **ne koristi server**
- Od Napstera se inspiriše i pravi **lidere grupa – to su peer-ovi koji vode evidenciju o sadržajima svih članova svoje grupe peer-ova**

Peer-ovi ovde nisu jednaki – svaki peer je ili **lider grupe** ili **član grupe**. Svi članovi peer-ovi uspostavljaju TCP konekciju sa svojim liderom peer-om. Postoji TCP konekcija i između nekih lidera grupa.

KaZaA obezbeđuje deljenje informacija o sadržaju između različitih grupa tako što lider neke grupe može da preuzme bazu podataka od nekog drugog lidera i pridodati je svojoj bazi ili može da prosledi upit drugim liderima grupa da bi pronašao željeni sadržaj.

Svaki fajl je identifikovan **KLJUČEM** i **IMENOM FAJLA**, a opciono i **tesktualnim opisom**. Ključ se dobija primenom **hash funkcije** nad imenom fajla. Klijent šalje upit postavljanjem ključnih reči svom lideru grupe i ako lider pronađe uprivanje sa zadatim ključnim rečima, onda šalje peer-u odgovor koji sadrži KLJUČ za dati fajl i IP adresa peer-a koji ima taj sadržaj. Ako se poklapanje ne pronađe u okviru grupe, lider grupe može da prosledi zahtev drugim liderima grupa koji eventualno pronalaze sadržaj. Nakon što



se locira željeni sadržaj klijent selektuje sa kog peer-a će skinuti željeni fajl tako što prosleđuje HTTP zahtev (kao id se koristi dobijeni ključ).

KaZaA može da postavi ograničenja za broj jednovremenih uploadovanja sa svakog računara. Svaki peer može da kaže koliko jednovremenih privlačenja sa njegovog čvora može u jednom trenutku da postoji. Ako u jednom trenutku ima više zahteva nego što je dozvoljeno (za downloadovanjem sadržaja sa nekog peer-a), onda se ti zahtevi stavljaju u red čekanja.

Nudi i opciju paralelnog privlačenja fajla tako da peer može da zahteva različite delove fajla sa više peer čvorova paralelno.

## Decentralizovan struktuiran P2P sistem

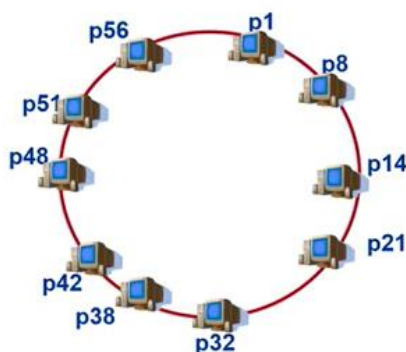
**Ne postoji centralni server, svi čvorovi su podjednako važni** (nema lidera). Problem sa 2. generacijom P2P sistema (Gnutella) je što se generiše veliki broj poruka u procesu pronalaženja željenog sadržaja, jer se koristi bujica. I ovde svi aktivni peer-ovi formiraju TCP konekciju i formiraju **overlay mrežu**.

U ovoj 3. generaciji P2P sistema, overlay mreža je **struktuirana** tj. mreža ima pravilnu strukturu (prsten, stablo). Informacije o podacima koji se traže su rasejani po čvorovima u mreži. Za pronalaženje željenog sadržaja koriste se **tabele** slične routing tabelama, ali se pretraživanje ne obavlja na osnovu IP adrese već na osnovu **KLJUČA**. Ne koristi se bujica, moguće je pronaći željeni sadržaj u konačnom broju koraka.

## Chord (žica)

**Protokol koji definiše kako se na osnovu ključa pronalazi željeni sadržaj, kako se vrši pridruživanje peer-a prstenu i kako se obavlja oporavak od otkaza nekog peer-a.**

Svim čvorovima i svim fajlovima dodeljuju se jedinstveni **m-bitni identifikatori** koji su raspoređeni na prstenu veličine  $2^m$  (ID-evi od 0 do  $2^m-1$ ). Na slici je dat primer kada je  $m = 6$  (64 identifikatora):



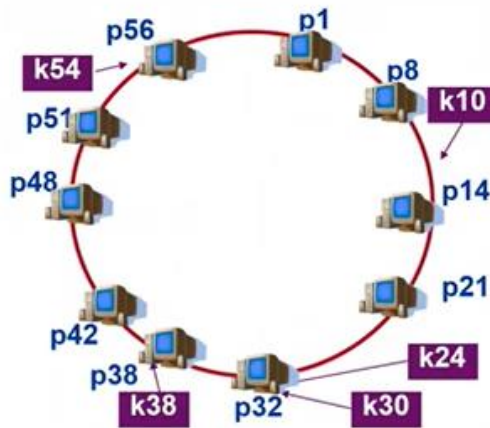
- Jedinstveni **ID čvora** dobija se primenom HASH funkcije na **IP adresu čvora** i na **broj porta**.  
$$ID = hash(IP, broj\ porta)$$
- Jedinstveni ID ili **KLJUČ** za **fajl** dobija se primenom iste HASH funkcije nad **imenom fajla** ili **sadržajem tog fajla**.

$$ključ = hash(fileName\ OR\ fileContent)$$

HASH funkcija vrši preslikavanje poruke proizvoljne dužine (fajl ili IP adresa + brojPorta) u blok fiksne veličine. Ima osobinu da je veoma randomizirana, pa će ID-evi biti ravnomerno raspoređeni po prstenu.

Kako se čvoru dodelje fajl za koji će on biti odgovoran?

Fajl sa ključem  $k$  se dodeljuje peer čvoru sa id-em  $ID$ , gde je  $ID \geq k$  (*gledano u smeru kazaljke na satu*). Dakle, prvi aktivni peer čvor koji se nalazi u Chord mreži i čiji je  $ID \geq k$  biće domaćin tog fajla. Taj peer čvor sa id-em  $ID$  je **sledbenik (successor)** za dati ključ  $k$  i označava se sa  $ID = succ(k)$ .



Recimo da se fajl sa  $k = 10$  dodeli prvom aktivnom peer čvoru (po kazaljci na satu) čiji je ID veći ili jednak od ključa  $k$ . To će biti peer sa  $ID = 14$ .

Ako gledamo  $k24$ , prvi na koga se nailazi, ako se ide u smeru kazaljke na satu, a da je ID veći ili jednak od 24 je peer 32.

Fajl sa ključem 30 biće dodeljen istom tom peer-u.

Za fajl 38 će biti odgovaran peer 38.

**Ako imamo fajl sa  $k = 60$ , kome će on biti dodeljen? Bio bi dodeljen peer-u 1, jer posle 56 je to jedina opcija.**

## Pronalaženje željenog sadržaja

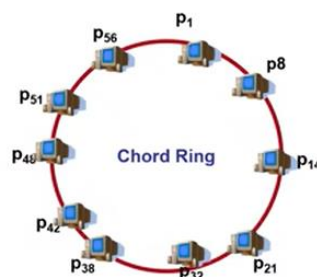
**Dovoljno je da svaki peer zna ID svog direktnog prethodnika i direktnog sledbenika.**

Sukcesivnim kontaktiranjem naslednika se uvek može pronaći željeni sadržaj (ako postoji). Svaki peer čvor zna i ko mu je direktni prethodnik, na taj način se vraća pronađen sadržaj tj. unazad (obrnuto od smera kazaljke na satu).

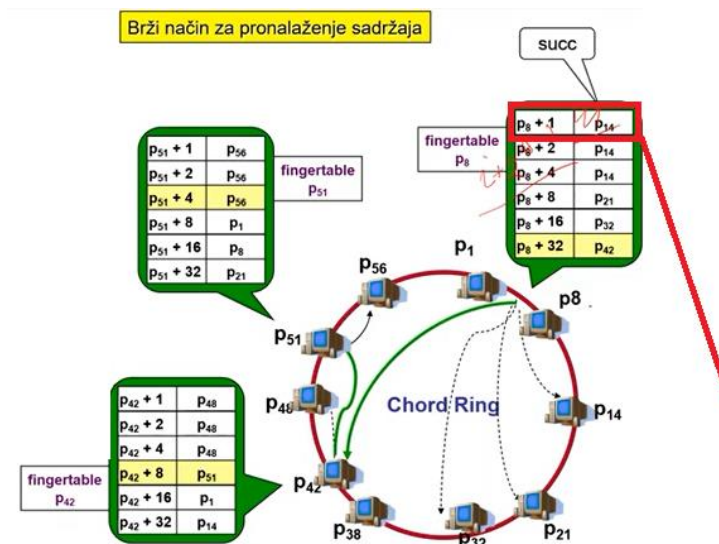
Broj koraka za pronalaženje željenog sadržaja je  $O(n)$ , pa što je  $n$  veće, složenost je veća. Da bi se smanjio broj koraka, svaki peer pamti ne samo ko su mu prethodnici i sledbenici nego ima i **FINGER TABELU** u kojoj pamti  $m$  najbližih suseda. Tako se broj koraka smanji na  $O(\log n)$ , što je dosta manje.

peer /sadrži tabelu rutiranja (finger table) sa najviše  $m$  vrsta.  
 $j$ -ta vrsta sadrži informaciju za naslednika (successora)  $i + 2^{j-1}$  ( $succ(i + 2^{j-1})$ )  
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



Svaki peer čvor  $P_i$  ima **finger tabelu** sa najviše  $m$  vrsta, koja sadrži ID-eve drugih čvorova koji predstavljaju **sledbenike** za ključeve čiji je  $ID = i + 2^{j-1}$ , pri čemu je  $i$  identifikator peer-a, a  $j$  je vrsta u tabeli i kreće se u granicama od 1 do  $m$ . Svaki čvor pored ove informacije (koja govori o tome koji peer je odgovoran za određeni ID), sadrži i IP adresu peer-a da bi mogao da bude kontaktiran i pronađen željeni sadržaj. Finger tabela čvora u opštem slučaju ne sadrži dovoljno informacija da se odmah u jednom koraku odredi naslednik za proizvoljni zadati ključ.



Prikazano je kako izgleda **Finger taberla** za čvorove **P8**, **P42** i **P51**. Pošto je u pitanju ring čija je veličina  $2^m$  tj. u ovom slučaju  $2^6$ , onda ID-evi peer čvorova idu od  $2^0$  do  $2^5$ .

Posmatrajmo finger tabelu za čvor P8. Finger tabela sadrži informacije o tome **koji čvor je sledbenik za fajl čiji je ključ jednak  $i + 2^{i-1}$** , što je u prvom redu za P8 jednako  $8 + 1 = 9$ . Sad je pitanje koji je čvor zadužen za fajlove sa ključem 9. **Prvi čvor čiji je ID veći ili jednak od 9 je čvor 14.**

Sledeća vrsta je  $8 + 2 = 10$ , pa je prvi čvor čiji je ID veći ili jednak od 10 čvor 14 opet. Sledeća vrsta je  $8 + 4 = 12$ , pa je opet čvor

14. Sledeća vrsta je  $8 + 8 = 16$ , pa je za to zadužen čvor 21. I tako dalje.

Dakle, svaki čvor kreira jednu ovakvu finger tabelu i ona omogućava da se željeni sadržaj nađe u mreži u  $O(\log n)$  konačnih koraka.

Kada se od čvora traži da pronađe lokaciju za zadati ključ fajla, on prvo pretraži svoju finger tabelu da nađe prvog **prethodnika** sa najvećim ID-em u odnosu na dati ključ. Ako se traži fajl sa ključem 10, traži se prethodnik od čvora sa ID-em 10, znači traži se čvor sa najvećim ID-em ali da je manji od 10.

Neka čvor P8 želi da nađe fajl sa ključem  $k=54$ . Prvo pokuša da vidi da li je njegov sledbenik (P14) slučajno veći ili jednak od ključa 54, pošto je ključ 54 veći od ID-a sledbenika 14, onda peer P8 pretražuje svoju tabelu i traži čvor čiji je ID najveći, ali manji od zadatog ključa. Nalazi da je to P42, pa P8 prosleđuje zahtev ka P42 i pita ga gde se nalazi fajl sa ključem 54. Sada P42 gleda svoju finger tabelu i traži peer čvor sa najvećim ID-em ali koji je manji od 54, a prvi takav je P51. Sad će P42 da prosledi upit ka P51 i pitaće ga gde se nalazi fajl 54. Peer P51 prvo pokuša da vidi da li je ključ manji ili jednak njegovom sledbeniku, a to je P56 i ustanovi da jeste, pa ne mora da pretražuje svoju finger tabelu. Peer P51 zna da se željeni sadržaj nalazi u čvoru P56, pa njemu upućuje zahtev i preko P42 odgovara procesu P8 gde je traženi fajl. Kada se dobije taj odgovor, onda može da se uspostavi direktna TCP konekcija da bi se izvršio pristup željenom fajlu.

### Pridruživanje čvora Chord prstenu

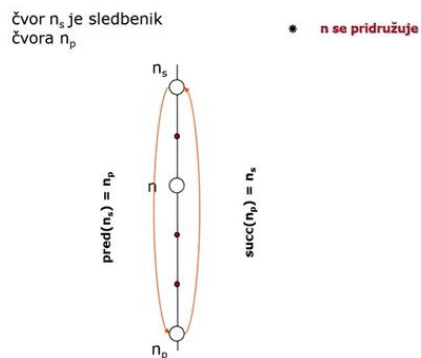
Čvorovi ne moraju stalno biti prisutni u mreži, mogu biti prisutni po svojoj volji. **Ono osnovno za funkcionisanje Chord ring-a je da svaki peer čvor ima validne pokazatelje na svog prethodnika i sledbenika.** Ako ulazi u finger tabeli nisu validni, to samo usporava pronalaženje sadržaja, ali pronalaženje je i dalje moguće pomoću sledbenika i prethodnika.

Da bi pointeri na prethodnika i sledbenika uvek bili validni, svaki čvor u Chord mreži periodično izvršava **protokol za stabilizaciju** da bi otkrio novopridružene čvorove ili one čvorove koji su izašli iz mreže.

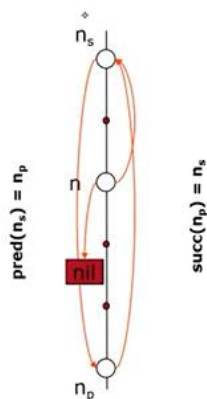
Neka je novi čvor dobio identifikator  $N$  i treba da se pridruži ring mreži iza peer čvora koji ima veći ID od njega. Da bi se to učinilo čvor koji se pridružuje izvršava funkciju **join( $N'$ )**, gde je  $N'$  bilo koji poznati čvor



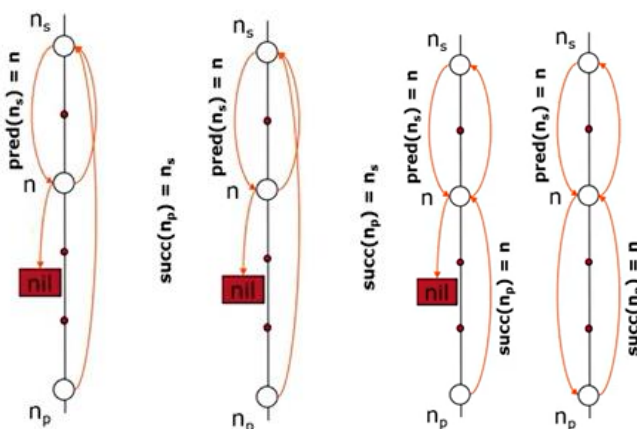
na Chord prstenu. Funkcija join traži od čvora  $N'$  da pronade neposrednog sledbenika za čvor  $N$ . Ova funkcija će obavestiti čvor  $N$  ko mu je sledbenik, a ostale čvorove ne obaveštava o prisustvu čvora  $N$ .



Neka postoji peer sa  $ID=N_p$  i peer sa  $ID=N_s$  i neka je  $N_s$  sledbenik čvora  $N_p$  (nalazi se iza  $N_p$ -a u ringu). Neka novi čvor  $N$  treba da se umetne između njih.

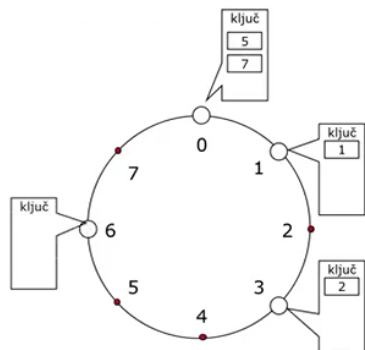


Novi čvor poziva funkciju  $\text{join}(N')$  i dobija kao odgovor sledbenika  $N_s$ . Čvor  $N$  sada obaveštava čvor  $N_s$  i kaže mu da je on sada njegov novi prethodnik.



Sada čvor  $N_p$  vrši **protokol za stabilizaciju** –  $N_p$  pita svog sledbenika (to je još uvek  $N_s$ ) „Ko je tvoj prethodnik?“. Na to pitanje  $N_s$  kaže da je njegov prethodnik čvor  $N$ . Tako  $N_p$  shvata da je dodat novi čvor u mreži, pa će čvoru  $N$  poslati poruku „Ja sam tvoj prethodnik“.

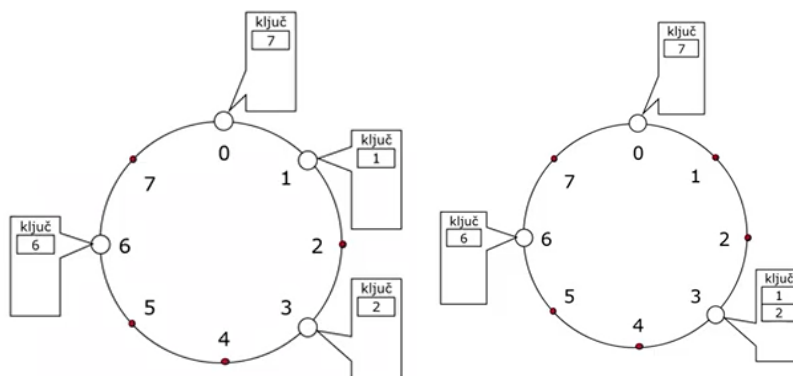
Kada se neki čvor pridruži mreži, neki ključevi koji su bili dodeljeni njegovom sledbeniku sada se dodeljuju čvoru N na čuvanje.



Ako je novi čvor N = 6, kao na slici, njegov sledbenik je aktivni čvor 0. Peer 0 ima ključeve 5 i 7, od njih se uzima ključ 5 (jer je on manji ili jednak od 6) i dodeljuje se peer-u 6.

### Napuštanje mreže

Kada čvor napušta mrežu, svi ključevi koji su mu bili dodeljeni se prebacuju njegovom sledbeniku. Neka je na donjoj slici peer 1 onaj koji će se isključiti iz mreže, njegove ključeve dobija sledbenik peer 3.



### Otkaz čvora

Ključni korak u oporavku od greške je održavanje korektna liste pointera na sledbenike. Ako čvor N primeti da je njegov neposredni sledbenik otkazao (ne odgovara na heartbeat poruku „Are you alive?“), on zamenjuje pointer pointerom prvog živog čvora u svojoj listi.



# Distribuirani fajl sistem

Jedan od najčešćih tipova distribuiranih sistema. Uobičajeno je da veliki broj računara može da deli pristup nekom skupu podataka ili programa. DS stvara iluziju da je deljivi disk pridružen svakom čvoru u sistemu, mada se nalazi na jednom ili nekoliko servera.

**Fajl** je imenovani skup podataka koji je trajno zapamćen na nekom disku/medijumu. Podaci zapamćeni u fajlu ne moraju imati značaj za operativni sistem, već za korisnika. Zadatak OS-a je da održava fajlove bez potrebe da ih razume.

**Direktorijum** je specijalna vrsta fajlova koja služi za organizovanje drugih fajlova i njihovi sadržaji mogu da ukazuju na druge direktorijume ili same fajlove.

**File system** je komponenta OS-a odgovorna za organizaciju, skladištenje, imenovanje, pretraživanje, deljenje i zaštitu fajlova. Obezbeđuje korisnički interfejs za rad sa fajlovima – operacija za rad sa fajlovima i operacije nad direktorijumima. ***Svi korisnici i svi fajlovi su na jednom istom računaru.***

Smeštanje fajlova na disku:

- **Kontinualno** – blokovi fajla su zapamćeni u sukcesivnim sektorima na disku, direktorijum ukazuje na lokaciju prvog bloka. Idealan za statične fajlove.
- **Ulančano** – blokovi jednog fajla mogu biti raseljeni na disku, kao lančane liste. Direktorijum ukazuje na lokaciju prvog bloka. Svaki blok ukazuje na sledeći blok.
- **Indeksirano** – poseban blok (**indeks blok**) sadrži pointere na sve blokove fajla. Direktorijum ukazuje na lokaciju indeks bloka. Svakom bloku se pristupa preko indeks bloka. Svaki indeks blok se pamti u specijalnoj lokaciji – **super blok**. Sam i-node pamti attribute fajla i pointer na blokove u kojima su zapamćeni podaci datog fajla. Ima 12 direktnih pointera, 1 indirektni, 1 dvostruki indirektni i 1 trostruki indirektni pointer.

**Distribuirani fajl sistem** radi u okruženju u kome su podaci rasejani na mnogo hostova u mreži, a i sami korisnici su distribuirani. Ovako se dobija više prostora za skladištenje podataka (nego na jednom hostu), dobija se i veća pouzdanost (korišćenjem replikacije) i otpornost na greške u sistemu. Implementacija distribuiranog fajl sistema DFS zahteva sve komponente tradicionalnog fajl sistema plus dodatne komponente za klijent-server komunikaciju, imenovanje i lociranje fajlova u DFS-u.

DFS se nalazi na više autonomnih mašina i omogućava da se pristup udaljenim fajlovima pristupa kao lokalnim fajlovima. Baziraju se na klijent-server modelu i pruža korisniku skup API-ja za pristup fajl sistemu. DFS se sastoji od:

- **Direktorijumskog servisa** – bavi se imenovanjem i lociranjem fajlova, vrši preslikavanje tekstualnog imena fajla u jedinstveni ID fajla na osnovu kog se on locira u DFS-u (ta imena su globalno jedinstvena u celom DFS-u). Može biti na odvojenoj mašini, na posebnom čvoru u mreži i sadrži informacije o indeks blokovima.
- **Fajl servisa** – bavi se održavanjem sadržaja fajlova i to se radi lokalno na svakom serveru na kome se fajl nalazi.

DFS treba da obezbedi:

- **Transparentnost pristupa** – istim skupom API-ja se pristupa udaljenim i lokalnim fajlovima
- **Lokacionu transparentnost** – na osnovu imena fajla se ne može zaključiti gde se fajl nalazi (da li je lokalan ili udaljen)
- **Transparentnost konkurencije** – fajlu može pristupati više korisnika, koji mogu i da modifikuju fajl pa se tada mora obezbediti da svi procesi imaju jedinstven pogled na stanje tog fajla (svi procesi vide izmene).
- **Transparentnost replikacije** – fajlovi mogu biti replicirani na više servera, ali klijent nije svestn replikacije
- **Migraciona transparentnost** – neki fajl može da promeni svoju lokaciju (sa jednog servera na drugi), a da to ne utiče na pristup fajlu od strane klijenta (promena lokacije nije zapažena od strane klijenta)
- **Heterogenost** – pristup fajlovima je moguć bez obzira na razlike u HW-u i SW-u klijenata.

Postoje 2 modela za pristup udaljenom fajlu:

- **Upload/download model**
- **Model udaljenog pristupa**

## Modeli pristupa udaljenom fajlu: Upload/download model

Jednostavan model koji ima dobre performanse, jer se operacije obavljaju lokalno, pa nema mrežnog saobraćaja. Jedini servisi za pristup udaljenom fajlu su komande **read** i **write**:

- **Read** – preuzima fajl sa servera (**download**) i pamti ga na klijent mašini, obavlja operacije nad fajlom lokalno, kada završi vraća fajl na server (**upload**).

Šta ako klijent nema dovoljno prostora da zapamti ceo fajl ili ako klijentu nije potreban ceo fajl, nego samo deo ili ako neki drugi klijent istovremeno želi da modifikuje fajl.

## Modeli pristupa udaljenom fajlu: Model udaljenog pristupa

Obezbeđuje da se sve operacije nad fajlom obavljaju na udaljenoj mašini (open, close, read, write, read byte, write byte). Operacije se izvršavaju na serveru i fajl se ne pomera sa servera. Sve se radi pomoću RPC mehanizma. Prednost je što je lakše implementirati deljenje fajlova, jer ako jedan klijent modifikuje fajl, ta promena je svima odmah vidljiva. Nedostatak je to što se sve vreme zahteva pristup serveru, može da nastane zagušenje u mreži i da server bude preopterećen (jer pristup serveru traje sve vreme dok se radi na fajlu). Performanse sa stanovišta brzine su gore od prethodnog modela.

## Stateless server

- Server ne pamti ništa o tome koji klijent pristupa fajlu i kom fajlu pristupa. Sve informacije potrebne da bi se opslužio klijent, mora da pruži sam klijent pri pristupu fajlu.
- Otporniji je na otkaz servera, jer se nikakve informacije neće izgubiti otkazom servera.
- Ako neki klijent modifikuje fajl, server nema načina da obavesti ostale klijente da je fajl modifikovan.
- Klijent vodi računa o konzistenciji i da li su podaci validni, jer server nema nikakve informacije. Server samo pamti fajl i održava fajl, ali ne vodi evidenciju o klijentskim zahtevima.

**Prednosti:** otporniji na otkaz servera jer ne pamti nikakve informacije o klijentskim zahtevima.

**Mane:** poruke koje se razmenjuju da bi se pristupilo fajlu su mnogo duže, u svakoj poruci klijent mora da navede ime fajla i komandu koja treba da se izvrši.

## Statefull server

- Suprotno od stateless servera, dakle pamti koji klijent je otvorio koji fajl.
- Bolje su performanse jer su kraće poruke koje klijent šalje serveru
- Lakše je ostvariti konzistenciju, jer server zna koji klijenti pristupaju kom fajlu i može da ih obavesti da je fajl promenjen.
- Može se izvršiti file locking – da se spreči jednovremena modifikacija istog fajla od više klijenata.

**Mana:** Teže se postiže otpornost na otkaze. Ako server otkáže, informacija o tome koji klijent je pristupio kom fajlu biće izgubljena. Pa je neophodno da se periodično pamti stanje servera, što može biti teško sa stanja performansi.

IMA JOŠ, NEGO ME MRZI VIŠE DA SLUŠAM OVO 😞

# Hadoop i HDFS

6. Šta omogućava uspešan restart servera na kom se izvršava NameNode demon? Koji se neophodni podaci za funkcionisanje klastera a koje poseduje NameNode, i gde i kako se oni skladište pre i nakon restarta servera?

Ono što je neophodno za restart servera je **TRAJNA MEMORIJA** – trajno čuvanje metapodataka (namespace file + promene koje će se desiti) NameNodea. Pri svakom restartu je potrebno da imamo podatke koji su se nalazili na HDFS-u prethodno. Kada se desi neka promena ona se opet beleži trajno u **edit log** – fajl na lokalnom fajl sistemu NameNode-a u kome se trajno čuvaju promene u HDFS-u. Promene iz edit loga- se primenjuju na namespace fajl i dobija se nova verzija namespace fajla.

NameNode čuva informacije o namespace-u, sadrži metapodatke fajl sistema i ima informacije o tome kako su fajlovi podeljeni na blokove i gde su ti blokovi locirani na klasteru. Od značaja su namespace fajl i edit log koji pamti promene u HDFS-u.

5. Koje vrste HDFS demona postoje i koja je uloga svakog od njih? Opišite postupak čitanja i postupak upisa u HDFS.

- **NameNode demon (samo jedan)** – izvršava se na glavnom ili **master** čvoru. On održava fajl sistem namespace i svaka promena njegovih karakteristika je zapamćena od strane NameNode-a. NameNode čuva informacije o tome kako su fajlovi podeljeni u blokove, koji **slave** čvorovi čuvaju koje blokove i sam učestvuje u preslikavanju blokova čvorove. NameNode je centralni kontroler HDFS-a.

- **DataNode deomoni (više)** – odgovoran za čuvanje podataka u blokovima, primanje naredbi od NameNode-a i davanje informacija NameNode-u. Šalju heartbeat-ove NameNode-u svake 3 sek i tako mu saopštavaju da su aktivni. Svaki 10. heartbeat je izveštaj o blokovima (koji blokovi su sačuvani na tom dataNode-u).

5. Koja je uloga NameNodea kod HDFS? Kako se postiže otpornost na greške (fault tolerance) kod HDFS?

NameNode održava fajl sistem namespace i pamti svaku promenu njegovih karakteristika. Čuva informacije o tome kako su fajlovi podeljeni u blokove. NameNode je centralni kontroler HDFS-a. Čuva metapodatke fajl sistema, nadgleda ponašanje DataNode-a i koordiniše pristup podacima. Ima informaciju o blokovima koji čine fajl sistem i gde su ti blokovi locirani u klasteru.

Otpornost na greške kod HDFS-a postiže se tako što se blokovi replikuju na više čvorova.

Rack Awareness