SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

Dr. sc. EUGEN MUDNIĆ

UVOD U DISTRIBUIRANE INFORMACIJSKE SUSTAVE

Split, 2008

SADRŽAJ:

LITERATURA:

Distributed Systems, Concepts and Design, Fourth Edition
George Coulouris, Jean Dollimore and Tim Kindberg
Addison Wesley/Pearson Education
2005
ISBN 0-321-263545

Distributed Computing – Concepts and Applications
M. L. Liu
Addison-Wesley, Inc.
2004
ISBN 0-321-21817-5

1. PREGLED, MODELI I KONCEPTI DISTRIBUIRANIH SUSTAVA

Distribuirani sustav je takav računalni sustav u kojemu komponente sustava locirane na umreženim računalima komuniciraju i koordiniraju svoje djelovanje isključivo prosljeđivanjem poruka.

Ova definicija nas vodi do slijedećih osnovnih karakteristika distribuiranih sustava:

- istodobno zajedničko djelovanje komponenti,
- nedostatak globalnog sata (global clock),
- neovisno otkazivanje komponenti.

Primjeri distribuiranih sustava:

- Internet,
- intranet (dio Interneta pod upravljanjem lokalne organizacije),
- mobilno ili sveprisutno računalstvo.

Glavna motivacija za konstrukciju i korištenje distribuiranih sustava je dijeljenje resursa. Resursima upravljaju serveri, a koriste ih klijenti.

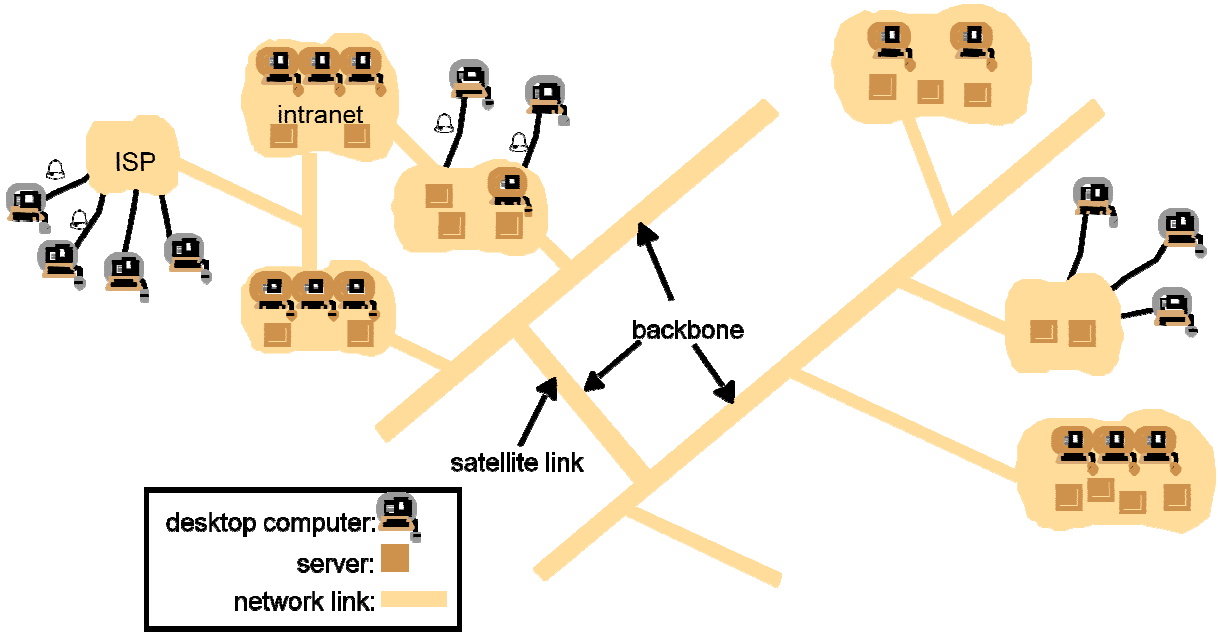
Glavni izazovi u konstrukciji distribuiranih sustava su:

- heterogenost komponenti sustava,
- otvorenost – mogućnost dodavanja ili zamjene komponenti,
- sigurnost,
- skalabilnost – mogućnost dobrog rada sustava povećavanjem broja korisnika,
- upravljanje ispadima dijelova sustava,
- konkurentnost komponenti,
- transparentnost.

1.1 Primjeri distribuiranih sustava

1.1.1 Internet

Internet je niz širom svijeta međusobno spojenih, javno dostupnih računalnih mreža koje prenose podatke korištenjem paketne komunikacije zasnovane na Internet Protokolu (IP).



Slika 1.1 Dio Interneta

Aplikacije na računalima koja su umrežena u Internet komuniciraju korištenjem zajedničke komunikacijske podloge (Internet protokoli).

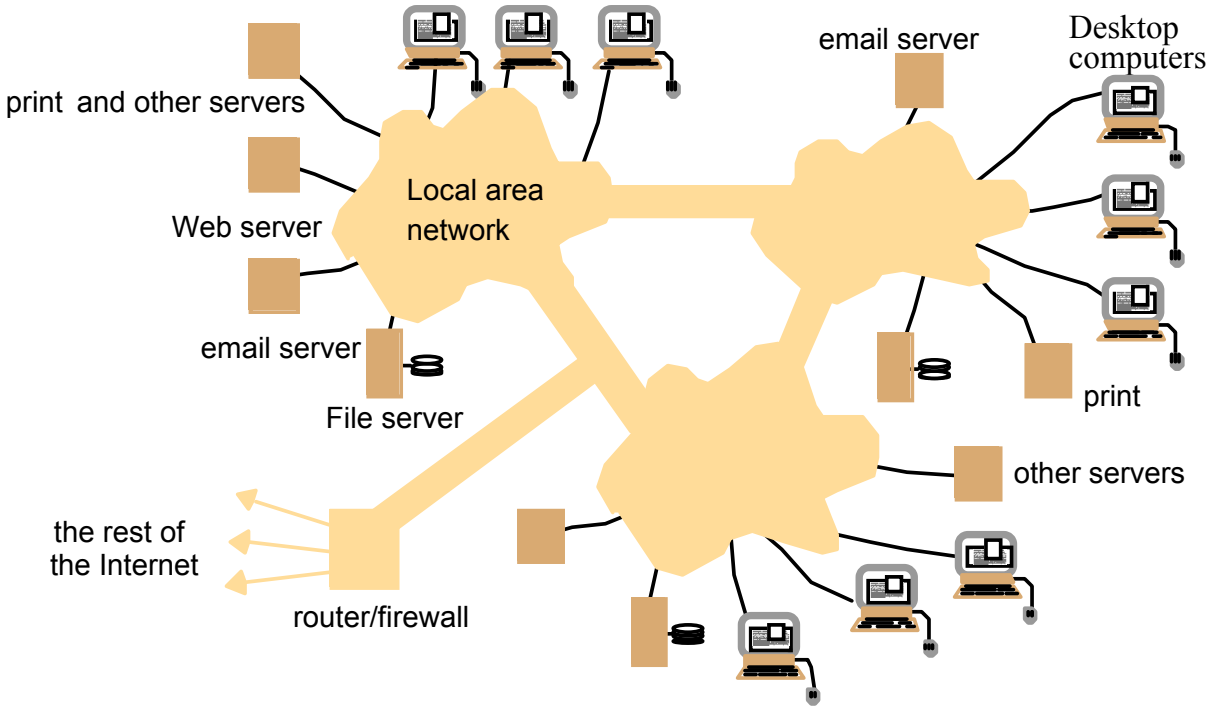
Internet predstavlja veoma veliki distribuirani sustav. On omogućava korisnicima da neovisno o položaju koriste servise poput WWW, elektronske pošte (email) i prijenosa datoteka. Set servisa je otvoren – može biti proširen dodavanjem novih poslužitelja ili novih tipova servisa.

Slika 1.1 ilustrira jedan odsječak Interneta. Možemo je gledati kao kolekciju intraneta – podmreža kojim upravljaju poduzeća ili druge organizacije. ISP (Internet Service Providers) su organizacije koje pružaju usluge konekcije te usluge npr. elektronske pošte ili web-hosting individualnim korisnicima ili manjim organizacijama.

Intraneti su međusobno povezani mrežnim vezama visokog kapaciteta (backbone) realiziranim odgovarajućim sklopovljem (optički kabeli, satelitske veze i ostali mrežni hardware). Internet može osigurati i prijenos multimedijalnih sadržaja iako postoje poteškoće jer osnovni Internet protokoli ne podržavaju rezervaciju kapaciteta za pojedinačne tokove podataka.

1.1.2 Intranet

Intranet je dio Interneta koji je pod posebnom administracijom i ima granice koje mogu biti konfigurirane da nameću određene sigurnosne mjere (security policy). Slika 1.2 pokazuje tipičan intranet.



Slika 1.2 Primjer intraneta

Sastoji se od nekoliko lokalnih mreža (LAN) povezanih s glavnim vezama. Sama konfiguracija intraneta varira od jedne LAN mreže na jednoj lokaciji , do niza LAN-ova koji se mogu rasprostirati i na različite zemlje.

Intranet je spojen na Internet korištenjem usmjernika (router). Usmjernik omogućava da Intranet korisnici koriste servise izvan intraneta (Web, e-pošta) ili da se servisi unutar intraneta stave na raspolaganje vanjskim korisnicima (ostatak Interneta).

Zadatak vatrozida (firewall) je zaštita intraneta. Vatrozid sprječava da neautorizirane poruke ulaze ili izlaze iz intraneta. Pri tome se koristi filtriranje poruka zasnovano na parametrima poruka poput oznake izvora ili cilja poruke.

Pojedine organizacije mogu svoj intranet ili dio u potpunosti izolirati od Interneta (npr. vojne organizacije).

Glavne pitanja u dizajniranju komponenti intraneta su:

- Potreba za datotečnim servisima koji omogućavaju dijeljenje podataka između različitih korisnika.
- Podešavanje vatrozida da ne predstavljaju zapreku korištenja legitimnih servisa.
- Troškovi instalacije i održavanja softvera.

1.1.3 Mobilno ili sveprisutno računalstvo

Tehnološki napredak u minijaturizaciji uređaja i bežične mreže doveli su do povećane integracije niza malih i prenosivih računalnih uređaja unutar distribuiranih sustava.

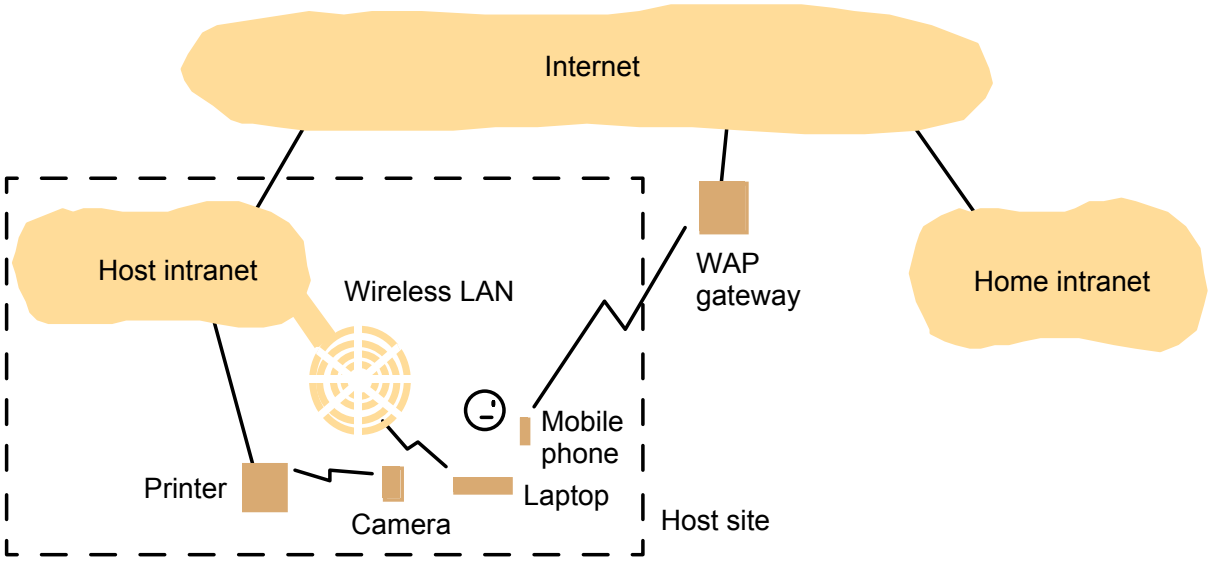
To su uređaji poput:

- prijenosna računala,
- ručna računala (PDA, mobilni telefoni, digitalne kamere,...),
- uređaji ugrađeni u druge uređaje poput digitalnih snimača, satelitskih prijemnika, strojeva za pranje, itd.

Mobilno računalstvo podrazumijeva da korisnik izvršava računalne zadatke na istom uređaju, ali na različitim lokacijama ili čak u pokretu. Pri tome može koristiti servise Interneta, pristupati svome vlastitome intranetu i koristiti resurse bliske trenutnoj lokaciji (npr. printeri).

Sveprisutno računalstvo podrazumijeva ugrađivanje računalnih uređaja u veliki broj objekata okoline. Time se korisniku otvara mogućnost korištenja niza servisa na lokaciji na kojoj se nalazi.

Slika 1.3 prikazuje situaciju kad korisnik posjećuje neku lokaciju. Korisnik sa slike koristi tri forme bežične mreže:



Slika 1.3 Primjer mobilnog korisnika

- veza prijenosno računalo – lokalni LAN ,
- mobilni telefon – telefonski operater (Internet),
- digitalni fotoaparat – printer.

1.2 Dijeljenje resursa i Web

Korisnici su se toliko navikli na dijeljenje resursa poput printera, datoteka, tražilica, itd. tako da sve manje primjećuju hardversko/softversku podlogu dijeljenih sustava. Dijeljenje hardvera npr. printera je lako sagledivo, ali dijeljenje podataka poput web-tražilica ili drugih dijeljenih baza podataka je daleko kompleksniji problem.

Uzorci dijeljenja resursa mogu se razmatrati po:

- opsegu dijeljenih resursa,
- povezanosti korisnika resursa.

Primjeri : Web tražilica , sustav za upravljanje dokumentima.

Pojmovi ◊ Servis, server, klijent, poruka

Servis je zaseban dio računalnog sustava koji upravlja kolekcijom resursa i prezentira njihovu funkcionalnost korisnicima i aplikacijama. Npr. *file service*, *printing service.*

Pristup servisu odvija se kroz operacije koje servis eksportira. Npr. file service – *read, write, delete*,itd.

Restrikcija pristupa preko prikladnog skupa operacija je uobičajena praksa u softverskom inženjerstvu.

Server je proces (program u izvršavanju) na umreženom računalu koji prihvaća i odgovara na zahtjeve izvršavanja servisa od drugih programa. Procesi koji zahtijevaju servis nazivaju se **klijenti**.

Zahtjevi i odgovori se šalju u formi **poruka**. Klijent poziva operaciju na serveru. Kompletna interakcija od točke poziva operacije servera do točke kad klijent primi odgovor servera naziva se **udaljeni poziv** (remote invocation).

Isti proces može biti i klijent i server, posebno što serveri često pozivaju operacije na drugim serverima. Stoga pojmovi klijent i server su **uloge procesa u jednom pozivu**.

Klijenti su aktivni serveri su pasivni. Server je u stalnom izvršavanju, a klijent samo dok je aktivna aplikacija čiji je dio.

1.2.1 The World Wide Web (WWW)

The World Wide Web (www.w3.org) ili skraćeno Web (Tim Berners-Lee, napisao prvi web klijent i server, 1990) je evoluirajući sustav za objavu i pristup resursima širom Interneta. Kroz Web preglednike korisnici mogu ostvarivati interakciju s vrlo velikim skupom servisa.

Web je nastao u CERN-u kao sustav za razmjenu dokumenata između fizičara na računalima spojenim u Internet. Ključna osobina Web-a je *hypertext* struktura između dokumenata koja omogućava organizaciju dokumenata. To podrazumijeva postojanje *link-ova (hyperlink)* unutar dokumenata - reference na druge dokumente pohranjene na Web-u.

Hipertekst strukture su stare 50-tak godina (Bush 1945), ali tek pojava Interneta dovela je upotrebu na opću razinu (zemlje, orbite, mjeseca,...).

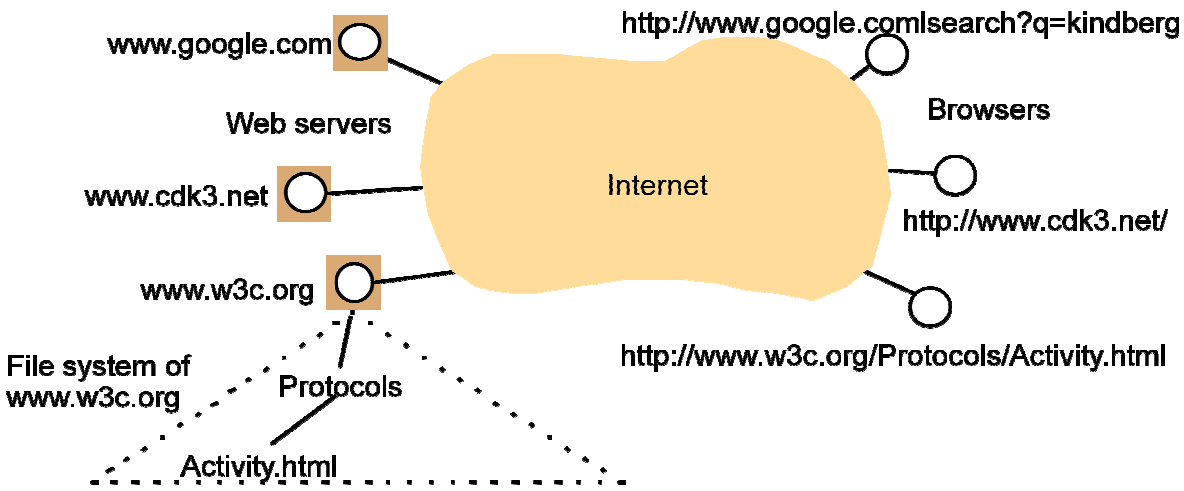
Web je ***otvoren*** sustav. To znači da može biti nadograđivan bez remećenja postojane funkcionalnosti. zasniva se na standardima komunikacije i dokumenata koji su javno objavljeni i implementirani u širokom opsegu. Različite implementacije Web preglednika i web servera slijede standarde i postoji interoperabilnost istih.

Web je otvoren prema tipovima resursa koji mogu biti publicirani i dijeljeni. U slučaju da se uvede novi resurs, npr. novi format pohrane podataka, svi elementi sustava mogu biti modularno nadograđeni za rad s istim.

Web je zasnovan na tri tehnološka standarda:

- HyperText Markup Language (HTML) – jezik za specificiranje sadržaja i izgleda stranica koje se prikazuju u web pregledniku.
- Uniform Resource Locators (URLs) – način identifikacije dokumenata i ostalih resursa kao dijelova Web-a.
- Klijent-server arhitektura – standardizirana pravila za interakciju po kojemu web preglednici i drugi klijenti pribavljaju dokumente i druge resurse od web servera.

Slika 1.4 prikazuje osnovnu strukturu Web sustava.



Slika 1.4 Osnovna struktura Web-a

HTML

HyperText Markup Language koristi se za specificiranje sadržaja stranice (tekst,slike), rasporeda elemenata stranice, veza (links) na druge resurse te njihovih tipova.

Tipični HTML kod izgleda ovako:

```
<IMG SRC = http://www.cdk3.net/WebExample/Images/earth.jpg>
<P>
Welcome to Earth! Visitors may also be interested in taking a look at the
<A HREF = "http://www.cdk3.net/WebExample/moon.html>Moon</A>.
</P>
```

Ovaj tekst se pohranjuje u datoteku koja mora biti dostupna web serveru, npr. datoteku earth.html. Korisnik upisom URL adrese stranice:
(http://www.cdk3.net/WebExample/earth.html) zahtijeva sadržaj datoteke od web servera lociranom na računalu www.cdk3.net .

Preglednik zatim čita sadržaj datoteke i prezentira njen sadržaj. Samo preglednik, dakle ne i server interpretira HTML tekst.

URL (Unifrom Resource Lokator)

Namjena URL-a je identifikacija resursa. Često se koristi i izraz URI (Uniform Resource Identificator) . Svaki URL u potpunoj formi ima dvije komponente:

scheme: scheme-specific-identifier

Npr:

mail:emudnic@fesb.hr identificira mail adresu

Najčešće se koristi HTTP URL. Opća forma HTTP URL-a je

http://servername[:port][/pathName][?query][#fragment]

, gdje su izrazi u uglatim zagradama opcionalni. Značenje pojedinih elemenata je:

- server name : IP adresa ili DNS (Domain Name System) naziv servera,
- port : broj porta na kojem se server prima zahtjeve (pretpostavljeno 80),
- pathName : staza do resursa,
- query : parametri za program ako pathname označava program,
- fragment : identifikator oznake fragmenta u html dokumentu.

HTTP

HyperText Transfer Protocol definira načine na koje preglednici i ostali tipovi klijenata vrše interakciju s web serverom. Glavne osobine protokola su:

- *zahtjev-odgovor interakcija (request-reply)* ◊ Klijent šalje pouku zahtjeva za resursom serveru naznačenom u URL-u. Server vraća sadržaj resursa ili poruku da nije našao resurs.
- *tipovi sadržaja (content types)* ◊ Preglednici ne moraju znati upravljati s bilo kojom vrstom sadržaja. Preglednik može poslati preferirani tip sadržaja (npr. gif format slike umjesto jpeg). Server onda može vratiti odgovarajući sadržaj. Server uključuje oznaku tipa sadržaja u poruku odgovora. Stringovi koji označavaju tip sadržaja se nazivaju MIME tipovi i standardizirani su. Npr. *text/html*, *application/zip* oznaka tipa. Set akcija koje preglednik izvršava za određeni tip podataka je konfigurabilan.
- *jedan resurs po zahtjevu* ◊ Ako web stranica sadrži npr. 5 slika, preglednik će web pregledniku uputiti 6 odvojenih zahtjeva za resursom. Obično su to konkurentni zahtjevi.
- *jednostavna kontrola pristupa* ◊ Pretpostavljeno je da korisnik s mrežnim pristupom web serveru može pristupiti svim objavljenim sadržajem. U slučaju da se želi ograničiti pristup sadržaju, moguće je konfigurirati server da izvrši provjeru identiteta svakog korisnika koji pristupa resursu (password).

Dinamičke stranice ◊ Primjenjivost web-a bi bila ograničena ako bi omogućavao samo prikaz unaprijed definiranih, statičnih stranica. Stoga često URL označava *program* na serveru, a ne datoteku. Npr. stranica može imati polja (web form) koja korisnik popuni i nakon toga pošalje (submit) sadržaj serveru. Tada se sadržaj forme šalje programu na serveru. Ako je kratak uobičajeno se šalje kao query komponenta URL-a.

Npr. za http://www.google.com/search?q=moon

search program će producirati HTML tekst kao svoj izlaz i poslati ga pregledniku.

Dakle preglednik uvijek prima HTML tekst.

Program koji web preglednik izvršava za generaciju HTML teksta se često naziva Common Gateway Interface (CGI) program. Može imati bilo kakvu internu funkcionalnost, ali eksterno gledano takav program razlaže (parse) primljeni sadržaj, a kao izlaz producira sadržaj (obično HTML tekst).

Povučeni kod ◊ Preglednik može povući i kod (npr. Javascript, Java Applet) i onda ga izvršavati te sadržaj prikazivati u okviru stranice. Npr. Javascript kod može izvršavati provjeru ispravnosti unosa podataka u polja web-forme.

Prednosti i nedostaci Web-a:

Uspjeh zasnovan na slijedećim elementima:

- Relativna jednostavnost objave sadržaja.
- Prikladnost hypertext strukture za organizaciju velikog broja različitih sadržaja.
- Otvorenost sistemske arhitekture.

Problemi dizajna :

- Hypertext model : linkovi mogu pokazivati ni na što, ako se ukloni pripadni sadržaj (ublaženo pretraživačima).
- Skaliranje: povećanje broja korisnika istog sadržaja uvjetuje distribuciju istog sadržaja na više računala. Ovisno o tehnici rasprostiranja sadržaja mogu se javiti problemi s ažuriranjem sadržaja na različitim pozicijama.
- Web stranica nije u svim slučajevima zadovoljavajuće sučelje. Dodavanje sadržaja ili raznih widgeta odužuje povlačenje stranice sa servera.

1.3 Izazovi dizajniranja distribuiranih sustava

Mnogi izazovi u dizajniranju distribuiranih sustava su uspješno riješeni, ali budući dizajneri moraju biti svjesni istih i uzeti ih u obzir prilikom korištenja ili nadogradnje sustava.

1.3.1 Heterogenost

Heterogenost se odnosi na slijedeće:

- mreže,

- računalni hardver
- operacijske sustave,
- programske jezike,
- implementacije različitih developera.

Heterogenost mreže riješena je korištenjem Internet protokola. Stoga nije bitna podloga tj.da li je u pitanju Ethernet ili FibreChanell ili iSCSI dok su implementirani Internet protokoli.

Računalni hardver može imati različite implementacije pohrane podataka (npr. poredak byte-ova). To se rješava na nivou softverskog međusloja gdje se uvode izvedeni tipovi podataka.

Operacijski sustavi implementiraju Internet protokole, ali API (Application Programming Interface) se razlikuje.

Različiti programski jezici mogu imati različite implementacije tipova podataka te nizova i struktura. Potreban je međusloj softvera koji rješava problem komunikacije programa pisanih u različitim programskim jezicima.

Različiti developeri trebaju koristi standarde koji omogućavaju međusobnu komunikaciju programa.

Heterogenost i mobilni kod ◊ Mobilni kod je kod koji može biti poslan s jednog računala da se izvršava na drugom. Međutim zbog heterogenosti operacijskih sustava nije garantirano njegovo izvršavanje na svakom od računala. Pristup upotrebom *virtualnog stroja* omogućava način da se omogući izvršavanje koda na bilo kojem stroju (Java virtualni stroj).

1.3.2 Otvorenost

Otvorenost računalnog sustava je karakteristika koja određuje da se sustav može proširiti i ponovo implementirati na različite načine. Prvenstveno je određena stupnjem u kojem se mogu dodati novi servisi dijeljenja resursa.

Otvorenost zahtijeva da se javno specificiraju i dokumentiraju sučelja ključnih komponenti sustava. Proces je srodan standardizaciji, ali često i zaobilazi oficijelne standardizacijske procedure. Npr. Internet protokol je specificiran preko niza javno dostupnih numeriranih dokumenata nazvanih RCF (request for comments). Dokumenti sadrže diskusije i same specifikacije protokola. Slično CORBA je publicirana kroz niz javno dostupnih dokumenata.

Sustavi koji su izgrađeni na temelju otvorenih protokola nazivamo četo otvoreni distribuirani sustavi. Otvoreni sustavi bi prema tome trebali biti neovisni od individualnih isporučitelja kako programske opreme tako i softvera.

1.3.3 Sigurnost

Često informacijski resursi koji se nalaze u distribuiranom sustavu su od velikog značaja za korisnike. Stoga je sigurnost takvih podataka vrlo bitna. Sigurnost informacijskih resursa posjeduje tri komponente:

- tajnost (confidentiality): zaštita od pristupa neautoriziranih korisnika,
- integritet: zaštita od izmjene i kvarenja,
- dostupnost: zaštita od uskraćivanja pristupa legalnim korisnicima.

Razvoj enkripcijskih tehnika značajno je povećao sigurnost, ali neki sigurnosni problemi još nisu zadovoljavajuće riješeni:

- *Napadi uskraćivanjem servisa* (Denial of Service Attcaks): određeni servis se može bombardirati s nizom besmislenih zahtjeva te time spriječiti ili značajno usporiti normalna upotreba servisa
- *Sigurnost mobilnog koda:* Mobilni kod poput priloga e-pošte ili koda koji se treba izvršavati unutar web preglednika može biti značajna prijetnja računalu.

1.3.4 Skalabilnost

Distribuirani sustavi efikasno izvršavaju zadatke u različitim razmjerima, počevši od malog intraneta do Interneta. Sustav smatramo skalabilnim ako ostaje upotrebljiv u slučaju porasta količine resursa i korisnika. Internet je primjer distribuiranog sustava s naglim porastom količine resursa i broja korisnika. Tablice koje slijed pokazuju porast broja računala i web servera u 24 godine razvoja Interneta.

Tabela 1-1 Porast broja računala uključenih u Internet

Date	*Computers*	*Web servers*
1979, Dec.	188	0
1989, July	130,000	0
1999, July	56,218,000	5,560,866
2003, Jan.	171,638,297	35,424,956

Tabela 1-2 Porast broja Web servera

Date	*Computers*	*Web servers*	*Percentage*
1993, July	1,776,000	130	0.008
1995, July	6,642,000	23,500	0.4
1997, July	19,540,000	1,203,096	6
1999, July	56,218,000	6,598,697	12
2001, July	125,888,197	31,299,592	25
		42,298,371	

Dizajn skalabilnih sustava postavlja slijedeće izazove.

Kontrola troškova fizičkih resursa ◊ Kako zahtjevi za resursima rastu treba postojati mogućnost da se uz razumne troškove sustav proširi. Npr. povećan broj pristupa (frekvencija) datotekama resursa može biti prevelika za jedan poslužitelj te zahtijevati dodavanje novih poslužitelja u sustav. Općenito sustav za *n* korisnika smatramo skalabilnim ako je potrebna količina fizičkih resursa za podršku reda najviše *O(n)*, tj. proporcionalna broju korisnika. Npr. ako jedan server podržava 20 korisnika onda bi dva trebala podržavati 40 korisnika. Iako zvuči razumljiv, taj zahtjev nije uvijek i ostvarljiv.

Kontrola gubitka performansi ◊ Razmotrimo održavanje skupa podataka čija veličina je proporcionalan broju korisnika (npr. DNS). Za pristup skupovima podataka velikih sustava koriste se hijerarhijske strukture jer se skaliraju bolje od linearnih. Međutim i s hijerarhijskim strukturama imamo gubitak u performansama. Vrijeme pristupa podatku u hijerarhijskoj strukturi je *O(log n)* , gdje je n veličina skupa podataka. Da bismo sustav smatrali skalabilnim maksimalan gubitak performansi treba biti manji od navedenog reda veličine.

Prevencija iscrpljivanja softverskih resursa ◊ Primjer nedostatka skalabilnosti je iscrpljivanje broja IP adresa (32 bitni). Nova verzija protokola s 128-bitnim adresama je prihvaćena, ali zahtijeva modifikaciju velikog broja softverskih komponenti. Teško je predvidjeti zahtjeve za desetljeća unaprijed. Ponekad je manja greška izvršiti prilagodbu nego koristiti predimenzionirani sustav.

Idealno, sistemski i aplikativni softver se ne bi trebao mijenjati povećanjem opsega sustava, ali to je često teško ostvarivo. Tehnike skaliranja sustava su vrlo značajan element u dizajnu distribuiranih sustava. Primjeri tehnika su:

- sustavi replikacije i keširanja podataka,
- korištenje višestrukih servera,
- korištenje specijaliziranih hardverskih komponenti.

1.3.5 Upravljanje pogreškama u radu sustava

Računalni sustavi ponekad su podložni greškama u radu. Pogreška se može dogoditi ili u hardveru ili u softveru te rezultirati u nekorektnim rezultatima ili prekidu rada, odnosno nekompletiranjem operacije. Pogreške u distribuiranim sustavima su parcijalne tj. pojedine komponente otkažu dok ostale nastavljaju raditi. Upravljanje greškama u radu sustava je izuzetno težak problem. Koriste se slijedeće tehnike.

Detekcija pogrešaka ◊ Neke pogreške je moguće detektirati. Npr. neispravan prijenos podataka može se detektirati korištenjem kontrolne sume (checksum). Teže je detektirati npr. ispad udaljenog servera.

Maskiranje pogrešaka ◊ Detektirane pogreške se ponekad mogu otkloniti npr. zahtjevom za retransmisiju ili korištenjem alternativnog izvora podataka.

Toleriranje pogrešaka ◊ Klijent može jednostavno obavijestiti korisnika o nastaloj pogrešci npr. nedostupnost stranice u pregledniku. Može se i isporučiti sadržaj s greškama tj. umanjene kvalitete, npr. multimedijalni sadržaj.

Oporavak od grešaka ◊ Dizajniranje softvera na način da nekompletne ili pogrešno izvedene operacije ne uzrokuju nekonzistentnost sustava.

Redundancija ◊ Servisi mogu biti konstruirani tako da toleriraju pogreške korištenjem redundantnih komponenti. Korisni primjeri su:

- korištenje bar dvije staze (routes) između bilo koja dva usmjernika (routera) u Internetu.
- Replikacija svake tabele u DNS sustavu bar na dva različita servera.
- Replikacija baze podataka na više servera na način da podaci ostaju dostupni nakon otkaza jednog ili više servera.

1.3.6 Konkurentnost

Često korisnici sustava pristupaju i žele mijenjati podatke približno u isto vrijeme. Sustav im može dopuštati pristup servisu "jedan po jedan", ali to značajno limitira propusnost sustava. Stoga se sami servisi moraju konstruirati na način da dopuštaju konkurentan rad više korisnika. Cilj je sačuvati konzistentnost podataka (sinkronizacija;semafori).

1.3.7 Transparentnost

Transparentnost je definirana kao prikrivanje odvojenosti komponenti distribuiranog sustava kako bi se sustav jasnije sagledao i koristio kao cjelina. Postoji više formi transparentnosti i ovdje je nabrojano osam ponešto modificiranih formi preuzetih iz ANSA reference manuala (1989)

- *Transparentnost pristupa*: omogućavanje korištenja identičnih operacija na lokalnim i udaljenim resursima.
- *Transparentnost lokacije*: omogućavanje pristupa resursima bez obzira na njihov fizički smještaj ili smještaj u mreži (npr. zgrada ili IP adresa).
- *Transparentnost konkurentnosti*: omogućavanje da se više procesa izvršavaju istovremenu korištenjem istih resursa, a bez nepoželjne interferencije.
- *Transparentnost repliciranja*: omogućavanje da se višestruke replike resursa koriste bez da korisnici i aplikacijski programeri moraju biti svjesni njihovog postojanja.
- *Transparentnost pogrešaka*: omogućavanje da i nakon pogrešaka u radu sustava korisnici i aplikacijski programeri kompletiraju započete zadatke.
- *Transparentnost mobilnosti*: omogućavanje pokretnosti klijenata i resursa u sustavu bez utjecaja na rad korisnika ili programa.
- *Transparentnost performansi*: omogućavanje rekonfiguracije sustava kao prilagodbu na opterećenje.
- *Transparentnost skaliranja*: omogućavanje skaliranja sustava i aplikacija bez utjecaja na strukturu sustava i aplikacijske algoritme.

Dva najznačajnija oblika su transparentnost pristupa i lokacije i često se zajedno nazivaju *mrežna transparentnost*.

2. MODELI SUSTAVA

2.1 Uvod

Različiti tipovi distribuiranih sustava imaju zajedničku podlogu te iz toga proizlaze i zajednički problemi dizajna. U ovom poglavlju zajedničke karakteristike distribuiranih sustava predstavljene su u formi opisnih modela. Svaki model koristi se za apstraktan, pojednostavljen, ali konzistentan opis relevantnog aspekta distribuiranog sustava.

Model arhitekture sustava definira način međusobne interakcije komponenti sustava i način na koji su te interakcije mapirane na podlogu sustava tj. mrežu računala.

Poglavlje 2.2 opisuje slojevitu strukturu softvera distribuiranog sustava i glavne modele arhitekture koji opisuju lokaciju i interakciju komponenti sustava. Razmatraju se varijante klijent-poslužitelj modela uključivši modele koji se pojavljuju zbog upotrebe mobilnog koda.

Poglavlje 2.3 uvodi tri temeljna modela koji nam omogućavaju razmatranje ključnih problema koji se postavljaju pred dizajnera sustava. Temeljni modeli daju apstraktan poglede na upravo ona svojstva distribuiranog sustava koja utječu na njihove karakteristike koje proistječu iz međuovisnosti komponenti sustava: ispravnost, pouzdanost i sigurnost.

2.2 Modeli arhitekture

Model arhitekture distribuiranog sustava prvo pojednostavljuje i apstrahira funkciju pojedinih komponenti sustava te zatim razmatra:

- smještaj komponenti unutar mreže računala – traženje korisnih uzoraka za distribuciju podataka i računalnog opterećenja,
- međurelacije između komponenti – njihove funkcionalne uloge i uzorke međusobne komunikacije.

Početno pojednostavljenje ostvaruje se klasifikacijom procesa na:

- poslužiteljski proces,
- klijent proces,
- peer proces (procesi s simetričnom komunikacijom).

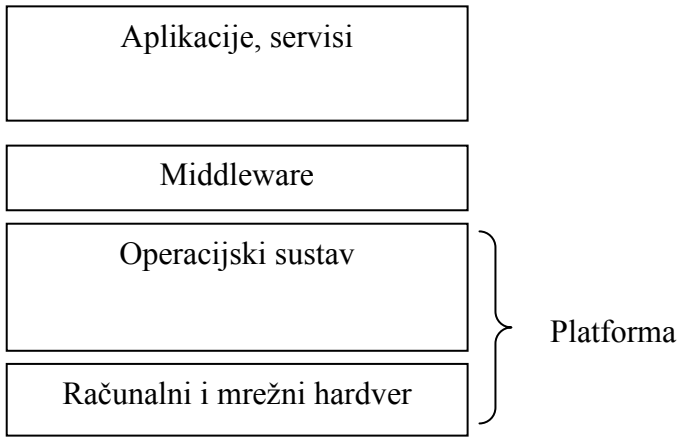
Dinamičniji sustavi mogu biti građeni kao varijacije klijent-poslužitelj modela:

- Mogućnost pomicanja koda između procesa omogućava procesu delegiranje; npr. klijent može povući kod sa servera i izvršavati ga lokalno. Cilj je minimizirati kašnjenje u pristupu i mrežni promet.

- Neki distribuirani sustavi imaju mogućnost bešavnog priključenja računala koja onda otkrivaju dostupne servise ili nude svoje servise ostatku sustava.

2.2.1 Softverski slojevi

Softver razmatramo ne kao slojeve modula u jednom računalu, već kao ponuđene ili tražene servise koje procesi locirani na istom ili različitim računalima nude ili traže. Dakle pogled je usmjeren na procese i servise koji oni nude. Slijedeća slika pokazuje osnovnu podjelu slojeva servisa.



Slika 2.1 Slojevi softverskih i hardverskih slojeva u distribuiranom sustavu

Server je proces koji prihvaća i obrađuje zahtjeve drugih procesa. *Distribuirani servis* može biti osiguran s jednim ili više server procesa koji međusobnom interakcijom i interakcijom s klijentom odražavaju funkcionalnost servisa na nivou distribuiranog sustava. Npr. servis mrežnog vremena NTP (Network Time Protocol) implementiran je kao veliki broj server procesa na različitim host računalima. Oni osiguravaju klijentu točno vrijeme, a vrše korekciju svog vlastitog vremena u međusobnoj interakciji.

Dva su posebno važna izraza: *platforma* i *middleware*.

Platforma ◊ Najniži hardverski i softverski slojevi nazivaju se *platformom* za distribuirane sustave i aplikacije. Osiguravaju servise za gornje slojeve do razine koja osigurava komunikaciju i koordinaciju među procesima. Najčešće platforme su: x86/MS Windows, 86_64/MS Windows, x86/Linux, x86_64/Linux, Sun Sparc/Solaris, Intel/Solaris, HP 9000,HP-UX 111,DEC Alpha,Digital Unix 4.0,IBM RS-6000/AIX 4.3,Intel/Mac Os X ,SGI/IRIX2.

Middleware ◊ Middleware je softverski sloj čiji je zadatak da maskira heterogenost sustava te osigura ugodan programski model za programere aplikacija. Middleware osigurava osnovne građevne blokove za izgradnju softverskih komponenti. Najčešće osigurava podizanje nivoa komunikacije među komponentama podržavajući apstrakcije poput:

- remote procedure calling (RPC), remote method invocation (RMI),

- notifikacija događaja (events notification),
- pohrana i korištenje dijeljenih podataka,
- replikacija dijeljenih podataka , itd.

Danas se najčešće koristi objektno orijentirani middleware poput:

- CORBA,
- Java RMI,
- Microsoft DCOM (Distributed Component Object Model),
- .NET Remoting ,
- Web servisi,

Ograničenja middlewarea

Aplikacija tipa klijent-poslužitelj , npr. baza podataka može se u potpunosti oslanjati na middleware koji samo pruža mogućnost poziva udaljenih metoda.

Međutim uvijek se postavlja pitanje koji dio komunikacijskog servisa postaviti u middleware, a koji dio realizirati na nivou aplikacije ?

Primjer: transfer mail poruke s velikim prilogom od mail hosta pošiljaoca do mail hosta primatelja.

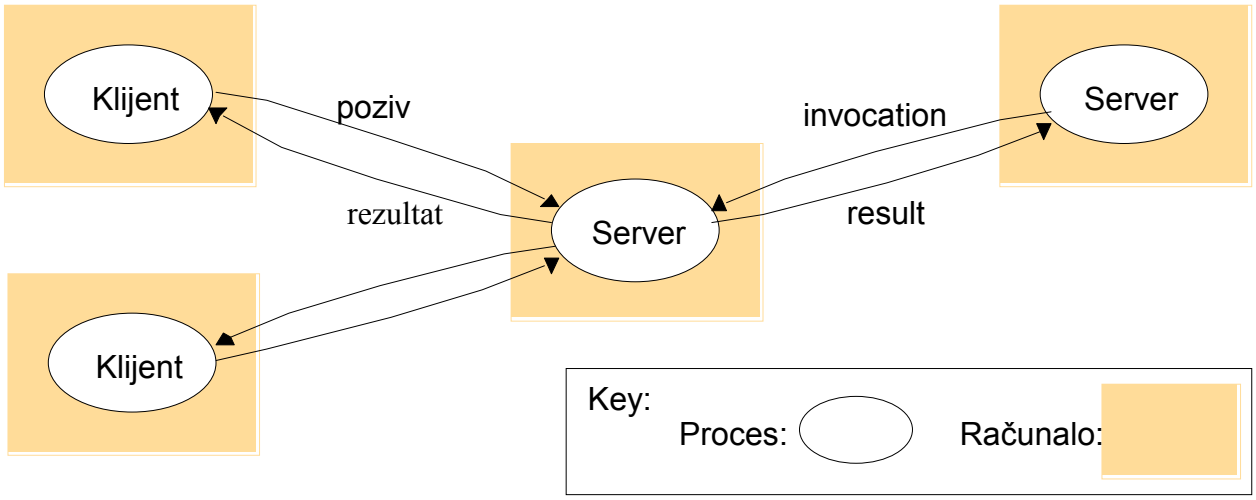
Na prvi pogled prijenos mail poruke je aplikacija koja se oslanja na TCP protokol. Međutim u slučaju nepouzdane mreže TCP protokol ne osigurava u potpunosti prijenos poruke. TCP protokol osigurava određenu detekciju pogrešaka i korekciju, ali ne može se oporaviti u slučaju većih prekida u mreži. Stoga aplikacija mail servera dodaje još jedan nivo tolerancije na greške, na način da nadgleda konekciju i u slučaju prekida koristi novu TCP konekciju.

U dizajnu distribuiranih sustava korisna je slijedeća izjava (Saltzer, Reed and Clarke 1984.)

Neke funkcije vezane za komunikaciju mogu se kompletno i pouzdano realizirati samo uz pomoć aplikacije koja stoji na krajevima komunikacijskog sustava. Zbog toga osiguravati tu funkcionalnost kao sastavni dio komunikacijskog sustava nije uvijek razumno.

2.2.2 Arhitekture sustava

Raspodjela funkcionalnosti između komponenti sustava i smještaj komponenti unutar mreže su najevidentniji aspekti dizajna distribuiranog sustava. Ima glavni utjecaj na performanse, pouzdanost i sigurnost rezultirajućeg sustava. Ovdje ćemo se usredotočiti na smještaj procesa koristeći označavanje gdje su procesi (elipse) smješteni u računala (kvadrati).



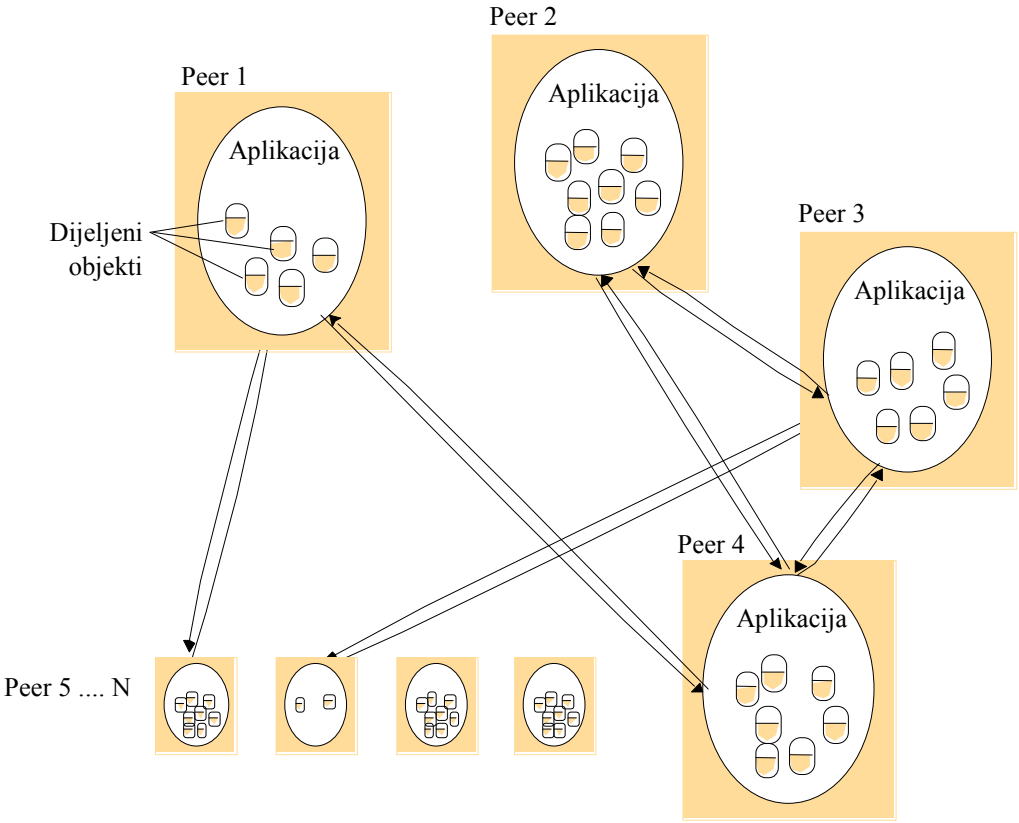
Slika 2.2 Klijent - server arhitektura

Klijent-server arhitektura ◊ Najviše spominjana arhitektura sustava koja je i u najširoj primjeni (Slika 2.2). Klijent proces inicira interakciju s individualnim serverom kako bi pristupio resursima pod upravljanjem servera. Npr. Web Server može biti u klijent lokalnog datotečnog servera. Većina Internet servisa je klijent DNS servera.

Peer-to-peer arhitektura ◊ U ovoj arhitekturi svi procesi uključeni u zadatak igraju slične uloge. U svojoj interakciji ostvaruju servis bez razvrstavanja u klijent ili server procese i bez obzira na računala na kojima se izvršavaju. Nastali su kao odgovor na loše skaliranje sustava zasnovanih na klijent-server modelu, a koje je posljedica kombinacije centralizacije servisa i ograničenja mrežne propusnosti.

Cilj peer-to-peer arhitekture je iskorištenje resursa (hardverski resursi, podaci) velikog broja učestvujućih računala za ispunjenje zadataka. Na tisuće računala mogu biti tako organizirana da dijele kolektivne resurse (podaci, mrežna propusnost, računalna snaga,...).

Slika 2.3 ilustrira oblik peer-to-peer aplikacije.



Slika 2.3 Peer-to-peer arhitektura aplikacije

Aplikacija je sastavljena od niza međusobno kooperativnih peer procesa.

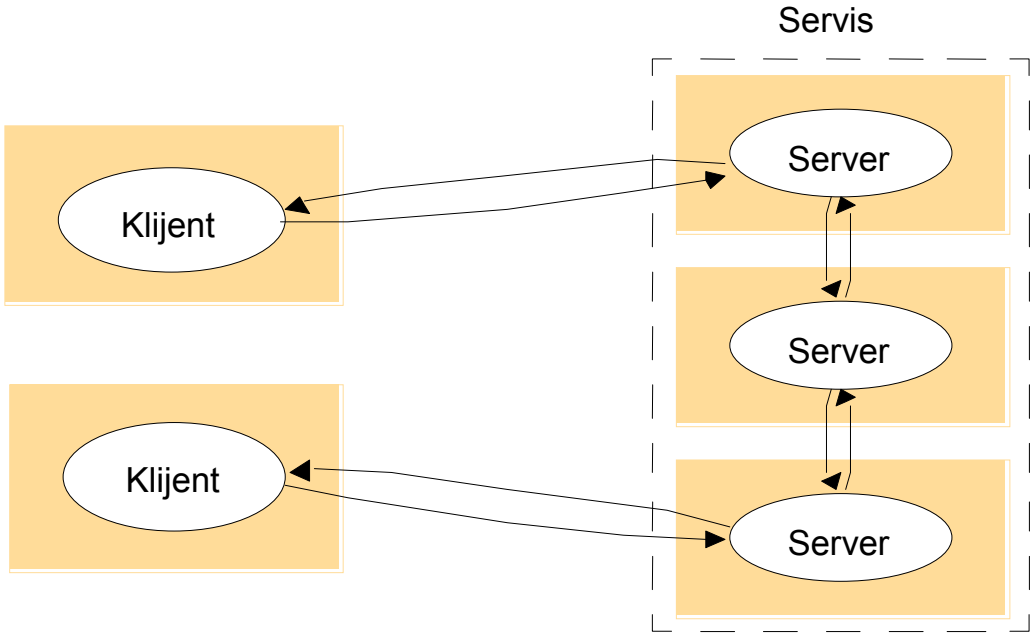
Objekti s podacima ne nalaze se samo na jednom računalu nego su replicirani tako da svako računalo sadržava samo dio podataka cjelokupnog sustava. Time se omogućava da se prilikom pristupa objektima koriste njihove višestruke kopije unutar sustava te time mrežno i procesno opterećenje distribuira na niz mrežnih veza i računala. Osim toga sustav postaje otporniji na ispad pojedinih računala. Ovakva arhitekturu sustava ima daleko veće zahtjeve na održavanje softvera od standardne klijent-server arhitekture.

2.2.3 Varijacije

Iz prije navedenih sustava se može izvesti niz varijacija ako uzmemo u obzir slijedeće faktore:

- Upotreba višestrukih servera te međumemorija zbog povećanja performansi i otpornosti na ispade.
- Upotreba mobilnog koda i mobilnih agenata.
- Potrebu korisnika za klijent računalima s jednostavno održavanim hardverom i softverom.
- Potrebu za jednostavnim spajanjem i odspajanjem mobilnih uređaja.

Servisi koji koriste višestruke servere ◊ Servisi mogu biti implementirani kao više serverskih procesa na odvojenim host računalima koji interakcijom omogućavaju određeni servis prema klijentu.



Slika 2.4 Servis realiziran višestrukim serverima

Distribucija podataka je moguća na način da serveri imaju razdijeljen (partitioned) skup podataka ili se radi o repliciranim kopijama.

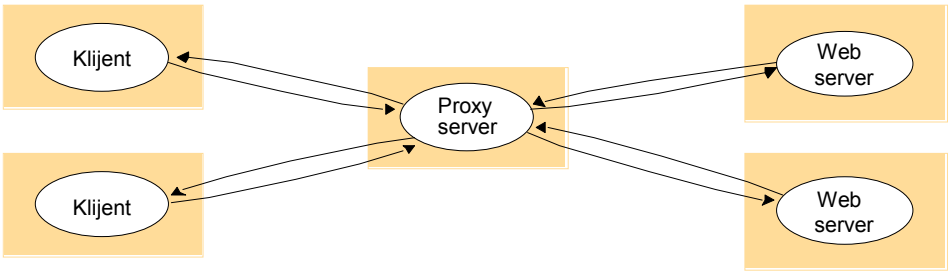
Primjer razdijeljenog skupa je Web. Svaki web server održava svoj skup resursa. Klijent – preglednik može pristupiti resursu na bilo kojem od servera.

Primjer repliciranog skupa podataka je Sun NIS (Network information service) koji se koristi prilikom logiranja na računalo ili LAN mrežu. Svaki od NIS servera ima svoju vlastitu repliku datoteke s passwordima.

Arhitekturu višestrukih servera s jačim međusobnim vezama nazivamo klaster (cluster) ili grozd. Koristi se npr. za pretraživače ili velike on-line prodavaonice. Mogu imati raspodijeljene ili particionirane servise.

Međumemorija (cache) i proxy serveri ◊ Međumemorija je skladište kopija nedavno korištenih objekata podataka koji su dostupniji nego originali objekata. Nalaze se na putanji izvor-cilj objekata. Kad je neki objekt potreban klijentu, on prvo provjerava njegovu egzistenciju u međumemoriji. Ako je traženi objekt u međumemoriji onda se povlači iz iste, a ako nije povlači se iz originalne lokacije s tim da se u međumemoriju pohranjuje njegova kopija. Ako je međumemorija puna iz nje se izbacuje najstarija, najrjeđe korištena ili neka druga kopija objekta.

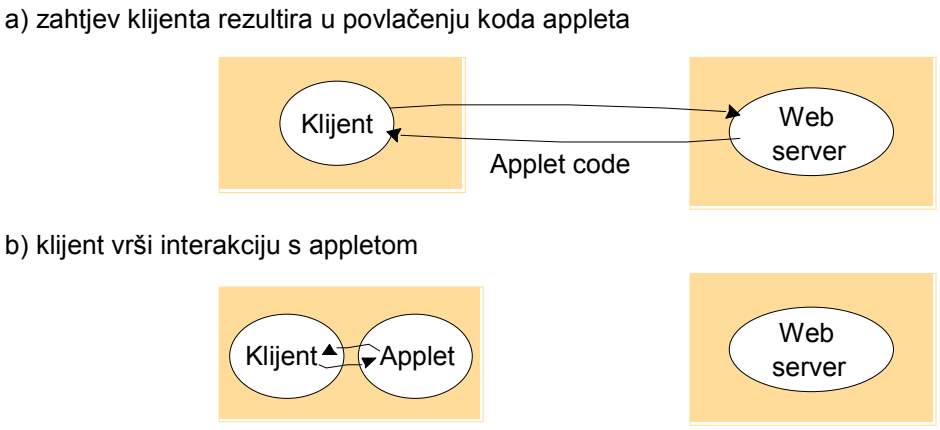
Web preglednici koriste međumemoriju prilikom pristupa stranicama s web servera (posebni HTTP zahtjev za provjeru ažurnosti kopije stranice).



Slika 2.5 Web proxy server

Web proxy serveri se nalaze u blizini klijenata i drže u međumemoriji sadržaje kojima pristupaju web preglednici. Proxy server može se koristiti i za pristup udaljenih web servera kroz vatrozid.

Mobilni kod ◊ Primjer mobilnog koda su appleti.



Slika 2.6 Izvršavanje mobilnog koda appleta

Razlog korištenja mobilnog koda može biti davanje bolje i ujednačenije reakcije na korisnikove akcije zbog izbjegavanja mrežnih kašnjenja. Npr. korisnik web preglednika je onaj koji uobičajeno inicira komunikaciju s web serverom. Međutim u nizu primjena web server bi trebao obavijestiti klijenta o promjenama u sadržaju podataka. Npr. web stranica za praćenja stanja cijena dionica može biti realizirana na način da se uz nju povuče kod koji će neprestano ažurirati područje prikaza cijena dionica na web stranici.

Mobilni agenti ◊ Mobilni agent je program u izvršavanju (i kod i podaci) koji putujući između računala izvršava zadatak zadan od strane korisnika istoga. Zadatak može biti npr. skupljanje podataka i vraćanje rezultata. Mobilni agenti mogu umanjiti komunikacijske troškove i skratiti vrijeme izvršavanja korištenjem lokalnih poziva umjesto udaljenih.

Mobilni agent može biti korišten npr. za instaliranje i održavanje softvera unutar neke organizacije. Može se koristiti npr. u Grid računalstvu kad može izvršavati proračune na čvorovima klastera.

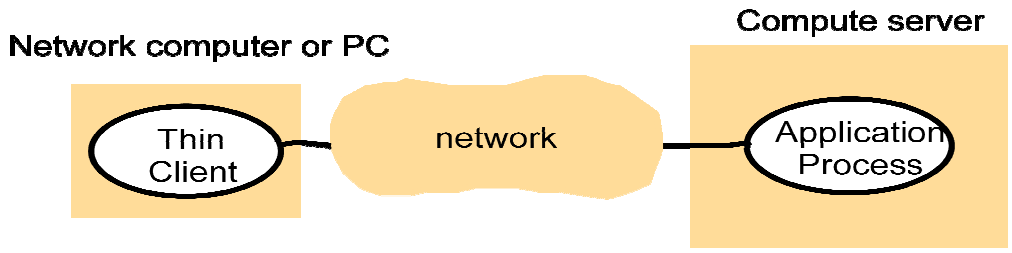
Nedostaci mobilnih agenata su potreba za povećanim administriranjem sustava u vidu potrebe identifikacije pošiljaoca mobilnih agenata te određivanja prava pristupa mobilnih agenata na različitim lokacijama izvršavanja. Pojačanjem mrežne propusnosti ponekad je jednostavnije koristiti udaljene pozive.

Mrežna računala ◊ Operacijski sustav i aplikacije koje se pokreću na tipičnom klijentu u klijent-server arhitekturi zahtijevaju veliku količinu aktivnog programskog koda i podataka na lokalnom disku klijenta. Njegovo održavanje zahtijeva velike resurse.

Mrežno računalo bi trebalo biti odgovor na taj problem. Ono povlači i operacijski sustav i aplikacije s udaljenog datotečnog servera. Aplikacije se izvršavaju lokalno, ali se datoteke održavaju na udaljenom serveru.

To dodatno omogućava da korisnik se lako migrira unutar mreže i da se koristi pojednostavljena računala. Mrežno računalo može imati i lokalni tvrdi disk, ali njegova funkcija je samo kao međumemorija.

Arhitektura mrežnog računala prikazana je na Slika 2.7.



Slika 2.7 Arhitektura mrežnog računala

Tanki klijenti ◊ Pojam *tanki klijent* odnosi se na softverski sloj koji na lokalnom računalu korisnika omogućava prozorsko grafičko sučelje prema aplikacijama koje se izvršavaju na udaljenom računalu. Takva arhitektura omogućava sličnu redukciju troškova sa strane klijenta kao korištenje mrežnih računala. Razlika je u tome da se umjesto povlačenja koda i podataka na klijent računala, aplikacije izvršavaju na udaljenom računalu. Zbog toga što se na njemu može paralelno izvršavati niz aplikacija, udaljeno računalo je najčešće u multiprocesorskoj izvedbi ili je u pitanju klaster. Glavni nedostatak ove arhitekture je da najčešće nije pogodna za aplikacije s intenzivnom grafičkom interakcijom. Mrežne latencije mogu uzrokovati neprihvatljivu interaktivnost aplikacija. Primjeri tankih klijenata su X11, VNC (Virtual network computer), WinFrame, NX.

Mobilni uređaji i spontana interoperabilnost ◊ Za razliku od mobilnih agenata koji predstavljaju softver koji migrira s računalo na računalo, mobilni uređaju predstavljaju migrirajući hardver s pripadnim softverom. Mobilni uređaji migriraju s lokacije na lokaciju i pri tome mijenjaju svoju poziciju u mreži. Primjeri uređaja su prijenosna računala, mobilni telefonski uređaji, PDA , .. Mobilni uređaji mogu koristiti različite mreže npr. WiFi, GSM, 3G, Bluetooth,...

Mobilni uređaj mogu sadržavati i klijente i servere, a uobičajeno je da su u ulozi klijenata.

Mobilna transparentnost je često problem kod ovakve arhitekture jer korisnici žele da sustav automatski funkcionira bez obzira na lokaciju i tipove resursa prisutnih na lokaciji.

Drugi problem je varijabilni spoj na mrežu jer uređaj u pokretu mijenja mjesto spoja i čak i tip mreže, a može i biti privremeno odspojen s mreže.

Konačno, to nas vodi do pojma *spontana interoperabilnost*, koji se odnosi na klijent-server arhitekturu u kojoj se podrazumijeva česta promjena uvjeta spojne veze klijent-server. Pri tome uređaj treba otkriti i koristiti resurse dostupne na nekoj lokaciji, a isto tako biti u mogućnosti dijeliti svoje resurse.

2.3 Temeljni modeli

Prije navedeni modeli sustava, iako različiti, dijele mnoge temeljne karakteristike. Cilj definiranja i korištenja temeljnih modela je:

- Eksplicitno definiranje svih relevantnih pretpostavki na kojima se modelira sustav.
- Generalizacija u smislu što je moguće, a što ne uz dane pretpostavke. Forme generalizacije su:
    - uopćeni algoritmi funkcioniranja sustava,
    - garantirana svojstva sustava.

Generalizacija se obavlja logičkom analizom ili matematičkim dokazima.

Cilj je dobiti odgovor da li će dizajn sustava zadovoljavajuće funkcionirati u određenoj okolini.

U temeljnim modelima želimo obuhvatiti aspekte koji nam omogućavaju diskusiju i zaključivanje o slijedećem:

Interakcija ◊ Model interakcije mora reflektirati činjenicu da se komunikacija obavlja uz kašnjenja koja su često značajnog trajanja, i da je koordinacija procesa limitirana tim kašnjenjima kao i nemogućnošću pouzdanog održavanja istog vremena u svim računalima distribuiranog sustava.

Pogreške ◊ Korektno djelovanje sustava je ugroženo kad god se dogodi hardverska ili softverska pogreška unutar sustava. Model treba definirati i klasificirati takve pogreške te dati osnovu za upravljanje pogreškama.

Sigurnost ◊ Modularna priroda sustava i otvorenost čini ih podložnima napadima. Model treba definirati i klasificirati forme napada te dati osnovu za zaštitu sustava.

2.3.1 Model interakcije

Ponašanje i stanje distribuiranog sustava opisujemo *distribuiranim algoritmom* – definicija svih koraka koje obavljaju sastavni procesi, *uključujući međusobni prijenos poruka*.

Poruke se prenose radi izmjene podataka i međusobne koordinacije.

Svaki proces se nalazi u određenom *stanju* koje se sastoji od skupa podataka kojima pristupa i upravlja.

Poteškoće:

- Brzina izvođenja procesa i vremena vezana za izmjenu poruka nije lako predvidjeti.
- Vrlo je teško opisati sva stanja distribuiranog algoritma zbog mogućih grešaka u izvođenju procesa i mogući grešaka u prijenosu poruka.

Dva najznačajnija faktora koji utječu na interakciju procesa u distribuiranom sustavu su:

- ograničenost komunikacijskih kanala,
- nemogućnost održavanja jedinstvenog globalnog vremena.

Ograničenost komunikacijskih kanala ◊ Komunikacijski kanali se realiziraju na korištenjem različitih softverskih i hardverskih podloga. Komunikacija korištenjem računalnih mreža posjeduje slijedeće karakteristike vezane za performanse:

- *Latencija* : kašnjenje između početka prijenosa poruke od strane jednog procesa i početka prijema od strane drugog procesa. Latencija uključuje:
    - vrijeme potrebno za prijenos kroz mrežnu infrastrukturu (npr. za FESB-CERN oko. 8ms),
    - kašnjenje u pristupu na mrežu (npr. ethernet protokol čeka na oslobađanje mrežnog resursa),
    - kašnjenje obrade poruke od strane komunikacijskih servisa i procesa pošiljaoca i procesa primatelja (varira npr. ovisno od opterećenja operacijskog sustava).
- *Širina propusnog opsega (bandwidth)* : Ukupna količina informacije koja može biti prenesena preko mreže u određenom vremenu. Komunikacijski kanali obično dijele dostupni propusni opseg.
- *Jitter* je varijacija u za vremena isporuke niza poruka (relevantno za multimediju).

Računalni satovi i vremenski događaji ◊ Svako računalo u distribuiranom sustavu ima svoj sat kojega koriste lokalni procesi. Stoga procesi mogu asocirati oznake vremena (timestamps) uz događaje. Međutim iako možemo inicijalno namjestiti vremena na svim računalima na neko referentno vrijeme, nakon nekog vremena javit će se određeno odstupanje.

Postoji nekoliko načina korekcije vremena na računalima. Npr korištenjem GPS sustava (Global Positioning System) moguće je namjestiti vrijeme lokalnog stata na razini 1 mikrosekunde. Međutim zasad ne možemo garantirati da svako računalo i na svakom mjestu može koristiti GPS za namještanje vremena.

Umjesto toga računalo s točnim izvorom vremena poput GPS-a može ostalima slati podatak točnog vremena, ali s utjecajem varijabilnih komunikacijskih kašnjenja.

Dvije varijante interakcijskog modela ◊ U distribuiranom sustavu teško je postaviti vremenska ograničenja na izvršenje procesa, isporuku poruka i pomak satova. Dva jednostavna modela koriste oprečne pozicije. Jedan ima stroge pretpostavke o vremenu, a drugi ga ne uzima u obzir.

Sinkroni distribuirani sustavi:

- Vrijeme izvršavanja svakog koraka procesa ima poznate gornje i donje granice.

- Svaka poruka prenesena preko komunikacijskog kanala ima poznatu gornju vremensku granicu.
- Svaki proces ima sat lokalnog vremena čije odstupanje ima poznatu gornju granicu.

Unatoč vrlo velikim zahtjevima moguće je sagraditi sinkroni distribuirani sustav. Međutim u sustavu trebaju postojati mehanizmi za rezervaciju i dodjelu željenih procesnih i komunikacijskih resursa te satovi s ograničenim odstupanjem.

Asinkroni distribuirani sustavi:

- Nema granica na brzinu izvršavanja procesa. Svaki korak može biti izvršen u proizvoljnom vremenu.
- Nema granica na kašnjenja u isporuci poruka.
- Nema granica na odstupanje sata.

Asinkroni model egzaktno opisuje Internet. Većina današnjih distribuiranih sustava su asinkroni jer **potreba za dijeljenjem resursa uvjetuje niz vremenskih varijabilnost**.

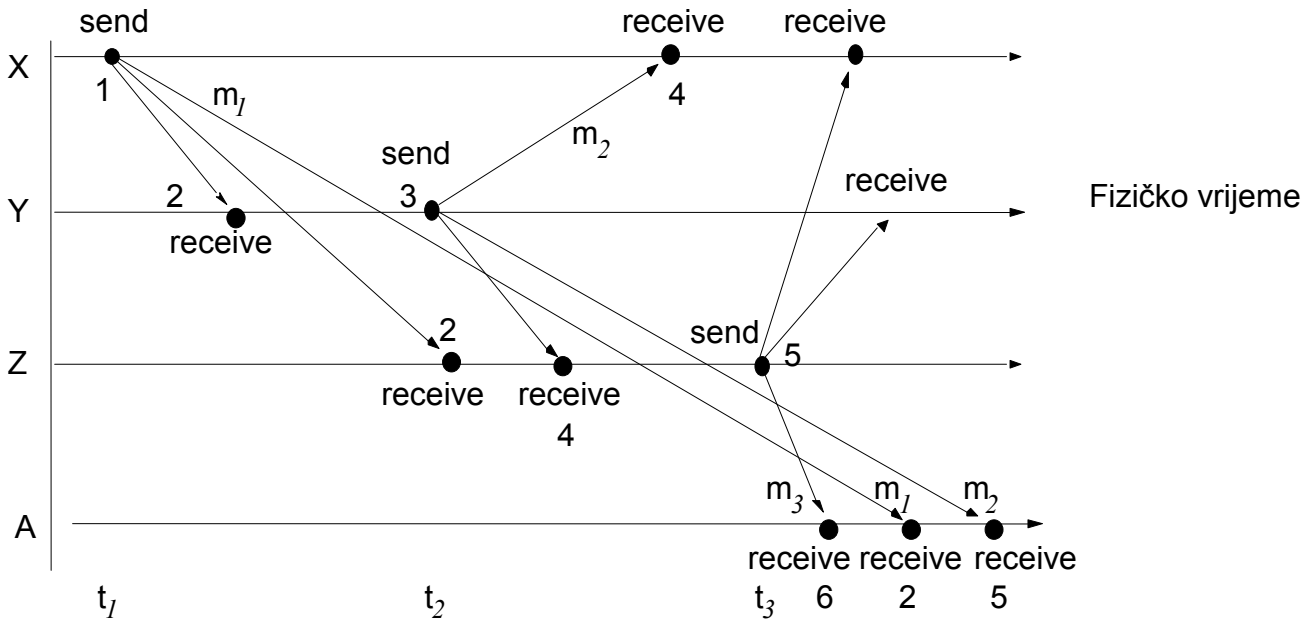
Poredak događaja ◊ Često je potrebno znati da li se neki događaj (npr. slanje ili primanje poruke) jednog procesa dogodio prije, poslije ili istodobno s drugim događajem u drugom procesu. Izvršavanje u sustavu da se opisati u formi događaja i njihovog poretka, unatoč nedostatku globalnog sata.

Primjer: izmjena mail poruka između korisnika mail liste (poslana poruka ide svim ostalim korisnicima liste) označenih kao X,Y,Z i A.

1. Korisnik X šalje poruku s subject-om : *Meeting*
2. Korisnici Y i Z odgovaraju na poruku s porukom sa subject-om: *Re:Meeting*

U realnom vremenu X' poruka poslana je prva, Y je čita i odgovara ; Z čita i X' poruku i Y' odgovor te šalje odgovor koji referencira i X' i Y' poruku.

Zbog neovisnih kašnjenja u isporuci poruka isporuka poruka može biti kao na slijedećoj slici.



Slika 2.8 Slijed poruka

Korisnik A vidi pogrešan poredak poruka ! :

Inbox:

Item	From	Subject
23	Z	Re:Meeting
24	X	Meeting
25	Y	Re:meeting

Ako se satovi na X' Y' i Z' računalu daju sinkronizirati, onda svaka poruka može nositi i oznaku vremena lokalnog računala koja kaže kad je poslana. Čak i u slučaju malo grublje sinkronizacije poruke će se pojaviti u korektnom redoslijedu.

Može se koristiti imodel *logičkog vremena* (Lamport,1978). Cilj nam je odrediti redoslijed prezentacije poruka bez korištenja sata. Koristi se numeracija slijeda događaja i iz dodijeljenog broja zaključuje se pravilan raspored.

2.3.2 Model pogrešaka (failures)

U distribuiranom sustavu i proces i komunikacijski kanal mogu biti uzrok pogreške. Pogreškom smatramo odstupanje od korektnog tj. neprihvatljiv rad sustava. Model pogreške definira načine na koje pogreška nastaje s ciljem da sagledamo utjecaj pogreške na rad sustava.

Pogreške možemo podijeliti na pogreške procesa i pogreške komunikacijskog kanala.

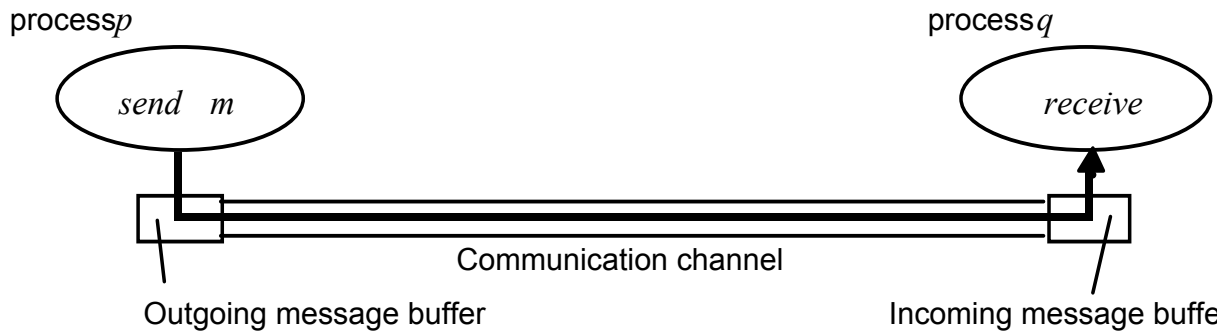
Dalje se pogreške mogu klasificirati na pogreške propusta (omission failures), proizvoljne pogreške (arbitrary failures) i pogreške odziva (timing failures).

Pogreška propusta ◊ Pogreška propusta događa se kad proces ili komunikacijski kanal ne izvrši akciju koju pretpostavljamo da treba učiniti.

Pogreška propusta procesa: Glavna pogreška propusta procesa je njegovo rušenje (crash). Proces je zaustavljen u izvršavanju i ne izvršava nikakve korake. Dizajn servisa koji može preživjeti rušenje nekog procesa može biti znatno pojednostavljen ako se pouzda u pretpostavku da proces funkcionira ili ispravno ili nikako.

Proces se naziva *fail-stop* ako drugi procesi mogu sa sigurnošću detektirati njegovo rušenje. Često se koristi detekcija rušenja na osnovu *timeouta.* Problem detekcije rušenja procesa zna biti vrlo složen.

Pogreška propusta komunikacijskog kanala: Događa se kad se ne prenese poruka iz spremnika pošiljaoca u spremnik primatelja ili između spremnika i procesa. Postoji niz mehanizama za detekciju takvog tipa pogrešaka.



Slika 2.9 Komunikacijski kanal

Proizvoljne pogreške ◊ Opisuje teško predvidive scenarije pogrešaka u kojima pogreška može biti bilo kakvog tipa. Npr. proces može čitati krive vrijednosti iz memorije ili slati krive podatke u porukama. Takve pogreške je vrlo teško detektirati.

Komunikacijski kanali mogu isto uzrokovati proizvoljne pogreške, ali mehanizmi detekcije (npr. kontrolna suma) uspijevaju uspješno detektirati i otkloniti takve pogreške.

Pogreške odziva ◊ Odnose se na sinkrone elemente distribuiranog sustava gdje imamo definirana vremenska ograničenja na izvršavanje nekog procesa ili na isporuku poruke. U asinkronom ustavu teško možemo govoriti o pogreškama odziva jer tu nemamo garancije odziva. Postoje posebni operacijski sustavi tzv. sustavi za rad u realnom vremenu, ali samo za specifične i kritične primjene. Odziv je bitan za multimedijalne aplikacije.

Maskiranje pogrešaka ◊ Općenito svaka komponenta u distribuiranom sustavu konstruirana je od kolekcije drugih komponenti. Moguće je konstruirati pouzdane servise na osnovu komponenti koje ispoljavaju pogreške. Npr. višestruki poslužitelji mogu držati replike podataka i nastavljati pružati servis i kad jedan od njih prestane s ispravnim radom.

Poznavanje prirode pogrešaka komponente na koju se servis oslanja može nam omogućiti dizajniranje servisa koji je u stanju maskirati pogreške komponenti na koje se oslanja.

Servis maskira pogrešku :

- Skrivanjem : npr. nakon detekcije pogreške traži se retransmisija poruke.
- Konverzijom u prihvatljiv tip pogreške: npr. provjerom kontrolne sume može se proizvoljna pogreška konvertirati u pogrešku ispada.

Pouzdanost komunikacije jedan na jedan ◊ Iako komunikacijski kanali ispoljavaju pogreške u radu moguće je ostvariti pouzdanu komunikaciju. Pojam *pouzdana komunikacija* definiran je u pogledu neospornosti (validity) i integriteta podataka.

- Neospornost – svaka poruka iz spremnika odlaznih poruka je prije ili kasnije isporučena u spremnik dolaznih poruka.
- Integritet – primljena poruka je identična poslanoj i nema dupliciranja poruka.

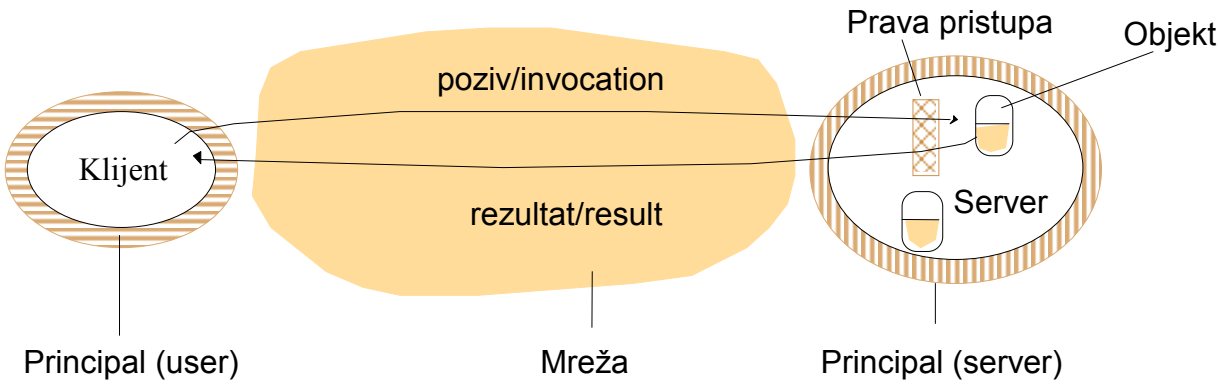
2.3.3 Model sigurnosti

Dijeljenje resursa je glavni motivacijski faktor distribuiranih sustava. Procesi enkapsuliraju objekte s podacima i omogućavaju kontroliran pristup drugim procesima.

Sigurnost distribuiranog sustava postiže se :

- osiguravanjem procesa i komunikacijskih kanala,
- zaštitom objekata podataka od neautoriziranog pristupa.

Zaštita objekata (resursa bilo kojeg tipa) ◊ Slijedeća slika prikazuje server koji upravlja nizom objekata za račun određenih korisnika.



2.10 Zaštita objekata

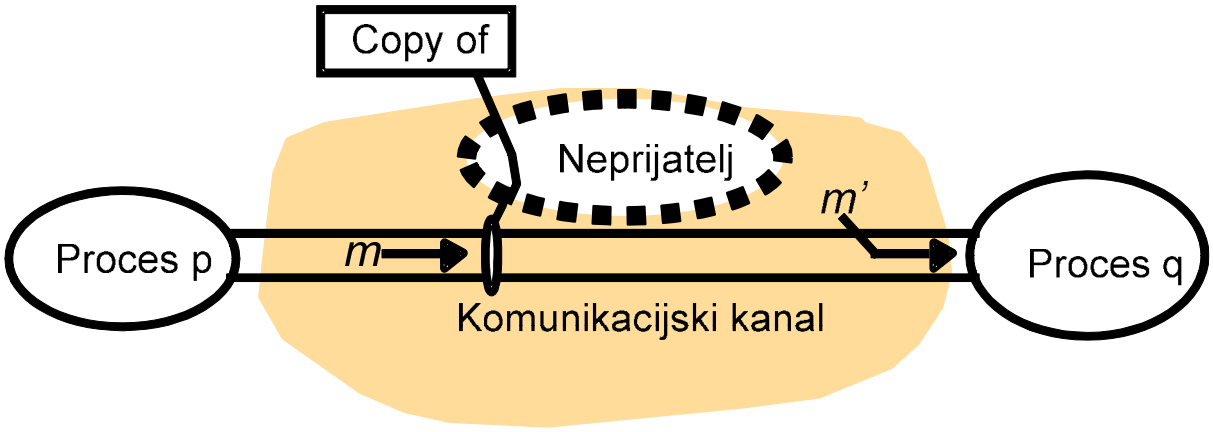
Prava pristupa (access rights) objektima razlikuju se za različite korisnike. Prava pristupa određuju operacije koje pojedini korisnik može izvršiti nad nekim objektom. Stoga korisnike moramo uključiti u model.

To radimo na način da svaki poziv i rezultat asociramo s autoritetom na čiji se račun radi. Takav autoritet nazivamo principal (upravitelj). Principal može biti korisnik ili proces.

Server je odgovoran za identifikaciju principala koji stoji iza svakog poziva te provjeru da li isti ima prava za obavljanje operacije nad određenim objektom. Klijent može provjeravati identitet principala koji stoji iza odgovora servera da bi se osigurao da odgovor dolazi od pozvanog servera.

Osiguravanje procesa i njihove interakcije kroz komunikacijske kanale ◊ Serveri i peer procesi izlažu svoja sučelja omogućavajući pozive od drugih procesa. Poruke kojima procesi komuniciraju izložene su napadima zato što koriste otvorene mrežne servise.

Uvodimo pojam **neprijatelja** čija aktivnost je čitanje/kopiranje poruka kojim procesi komuniciraju ili slanje poruka procesima.

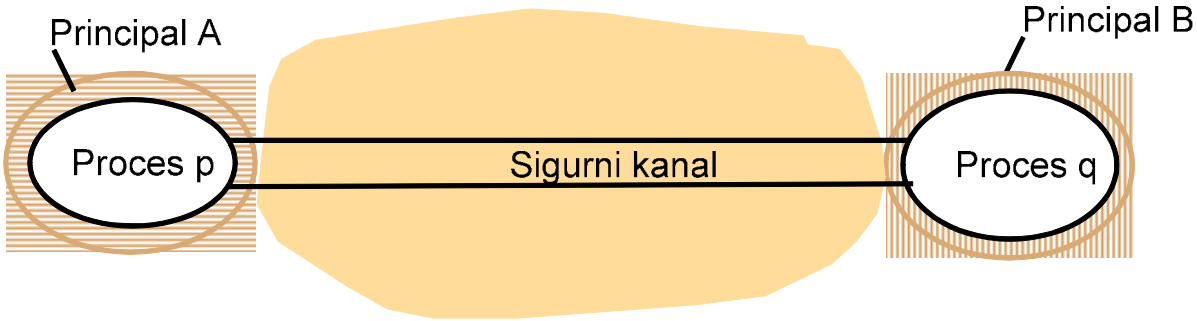


2.11Napad kroz komunikacijski kanal

Neprijatelj djeluje s legalno ili ilegalno mrežno priključenog računala te koristi programe kojima prati, čita ili generira poruke.

Prijetnja procesima: Proces može primiti poruku, a da nužno ne odredi pravi identitet pošiljaoca. Komunikacijski protokoli poput IP protokola uključuju adresu izvora u svakoj poruci, ali nije velika teškoća da neprijatelj generira poruku s lažnom adresom izvora.

Prijetnja komunikacijskim kanalima: Neprijatelj može kopirati, izmijeniti ili injektirati poruku dok ona putuje komunikacijskim kanalom. To predstavlja prijetnju integritetu poruke kao i cijelog sustava. Takve prijetnje rješavaju se upotrebom sigurnih kanala koji se zasnivaju na kriptografiji i autentikaciji.



2.12 Upotreba sigurnog kanala

Sigurni kanali ◊ Predstavljaju servisni sloj na vrhu postojećih komunikacijskih slojeva. To je komunikacijski kanal koji spaja par procesa, a koji djeluju u korist svojih principala. Sigurni kanal ima slijedeća svojstva:

- Svaki od procesa pouzdano zna identitet principala koji stoji iza drugih procesa. To omogućava pouzdanu primjenu prava pristupa. Isto tako klijent je siguran u to da odgovor dolazi od željenog servera.
- Osiguranje privatnosti i integriteta podataka koji se prenose preko komunikacijskog kanala.
- Svaka poruka posjeduje logičku ili fizičku oznaku vremena što sprječava promjenu redoslijeda ili ponovno slanje poruke.

Primjeri tehnologija su VPN (virtual private networks) i SSL (Secure socket layers).

Ostale neprijateljske prijetnje

Denial of Service (DoS) : Forma napada u kojem neprijatelj interferira s legalnim aktivnostima ostalih korisnika na način da šalje besmislene i prekomjerne pozive s ciljem da izazove preopterećenje fizičkih resursa sustava. Time može izazvati vrlo tešku dostupnost servisa ili čak ga potpuno onemogućiti. Posebno je opasno ako je time pogođen servis koji nadgleda sigurnost drugih servisa (npr. nadzorni sustav banke).

Mobilni kod: Postoji veliki broj formi napada mobilnim kodom. Posebno je opasno to što mobilni kod u većini slučajeva obavlja korisne funkcije i što se nivo pristupa mobilnog koda resursima sustava često regulira odobrenjem (često nesmotrenih) korisnika.

Upotreba sigurnosnih modela

Može izgledati da je upotreba sigurnosnih modela prilično jednostavna ako se upotrijebe sigurni kanali i autentifikacija, ali to nije općeniti slučaj. Upotreba tehnika enkripcije i kontrole pristupa donosi značajno povećane troškove procesiranja i upravljanja.

Osim toga česta je pojava da se upotrebom tehničkih aspekata sigurnosti zanemari ljudski faktor odnosno da čovjek kao korisnik griješi ,bilo nenamjerno ili zlonamjerno.

Stoga je potrebno za sustav načiniti listu svih modela napada kojima je sustav izložen te napraviti procjenu rizika i posljedica te troškova za svaki od modela.

3. SINKRONIZACIJA I KOORDINACIJA

3.1 Vrijeme i satovi

Pitanje točnog vremena u distribuiranim sustavima je od velike praktične važnosti. Npr. želimo da računala daju konzistentne oznake vremena transakcijama plaćanja.

Svako računalo u distribuiranom sustavu ima svoj fizički sat. Satovi uvijek odstupaju, a ne možemo ih ni savršeno sinkronizirati.

Ovdje ćemo razmotriti algoritme koji omogućavaju da se izvrši aproksimativna sinkronizacija satova u distribuiranom sustavu, a zatim ćemo objasniti pojam logičkih satova, uključujući i vektorske satove.

Nedostatak globalnog fizičkog vremena onemogućava nam određivanje stanja distribuiranih programa u izvršavanju. Često je potrebno znati u kojem se stanju dva ili više procesa na različitim računalima u istom vremenskom trenutku, ali je pitanje s kojom pouzdanošću se možemo osloniti na lokalne satove fizičkog vremena.

Da bismo znali kojim redom su se odvijali pojedini događaji na različitim računalima potrebno je sinkronizirati njihov sat s nekim autoritativnim , vanjskim izvorom vremena. Npr. transakcija e-trgovine uključuje događaje na računalu trgovca i na bankovnom računalu. Za potrebe revizije potrebno je da ti događaju posjeduju točnu oznaku vremena.

Niz algoritama u distribuiranim sustavima oslanja se na sinkroniziranost satova:

- održavanje konzistentnosti distribuiranih podataka,
- provjera autentičnosti zahtjeva poslanih na server,
- eliminacija procesiranja dupliciranih ažuriranja,
- garbage–collection kod distribuiranih objekata,...

3.1.1 Satovi, događaji i stanja procesa

Cilj razmatranja je kako dati vremensku oznaku pojedinim događajima u sustavu.

Neka se distribuirani sustav u izvršavanju je kolekcija $\mathcal{P}$ koja se sastoji od N procesa p_i, $i=1,2,... N$.

Svaki proces se odvija na jednom procesoru (jezgri) i procesori ne dijele memoriju. Svaki proces p_i nalazi su u stanju s_i . Izvršavanje procesa vrši transformaciju stanja. Stanje uključuje sadržaj svih varijabli koje ga sačinjavaju. Stanje procesa može sadržavati i vrijednosti objekata operacijskog sustava poput datoteka. Podrazumijevamo da procesi ne mogu komunicirati na bilo koji drugi način osim slanjem poruka kroz mrežu.

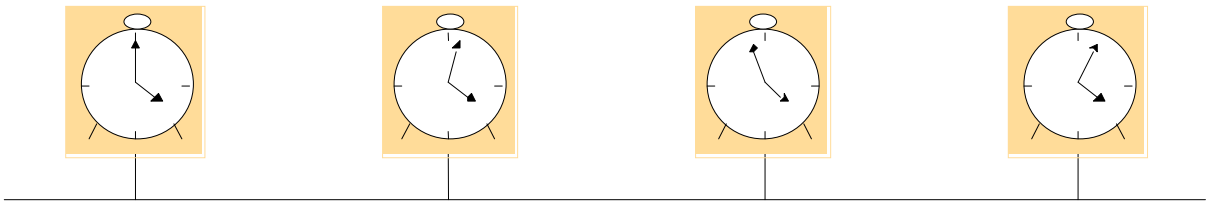
Kako se svaki proces p_i izvršava on poduzima niz akcija . Akcije mogu biti slanje ili primanje poruka ili promjena stanja sustava s_i. Događajem označavamo pojedinačnu akciju procesa u izvršavanju – ili komunikacijsku akciju ili transformaciju stanja.

Sekvencu događaja unutar jednog procesa p_i možemo poredati o pojedinačni redoslijed koji označavamo relacijom $\rightarrow_i$ između događaja.

Stoga $e \rightarrow_i e'$ znači da se događaj e dogodio prije e' u procesu p_i. Ovaj poredak je dobro definiran bez obzira da li je proces višenitan jer imamo ograničenje izvršavanja na jednom procesoru (jezgri). Sada možemo definirati povijest procesa p_i kao seriju događaja unutar istoga, poredanih pomoću relacije $\rightarrow_i$:

$$history(p_i)=h_i= <e_i^0, e_i^1, e_i^2,... >$$

Sat ◊ Vidjeli smo kako poredati događaje u sustavu, ali kako im dati vremenske oznake (timestamp). Svako računalo sadržava vlastiti fizički sat.



Slika 3.1 Prikaz odstupanja satova u distribuiranom sustavu

Operacijski sustav čita vrijednost hardverskog sata računala $H_i(t)$ i nakon operacija skaliranja i dodavanja pomaka producira softversko vrijeme $C_i(t)=\alpha H_i(t)+\beta$ koji daje aproksimativni iznos fizičkog vremena. $C_i(t)$ se često bilježi kao 64-bitni broj i označava broj nanosekundi koje su protekle od nekog referentnog trenutka. Sukcesivni događaji će imati različite oznake vremena samo ako je *rezolucija sata* $C_i(t)$ – vrijeme ažuriranja softverskog vremena – manja od vremenskog intervala između događaja.

Odstupanje sata ◊ Računalni satovi , kao i svi drugi satovi nisu međusobno usklađeni (Slika). Razlika istovremenih očitanja sata naziva se odstupanjem (skew). Računalni satovi su zasnovani na kvarcnim kristalima i podložni su odstupanjima takta i s vremenom divergiraju. Čak isti sat ima različitu mjeru odstupanja zbog npr. promjene temperature.

Bez obzira koliko točno namjestili dva sata oni će nakon dužeg vremena (veliki broj oscilacija) imati primjetna i značajna odstupanja. Normalni kvarcni sat ima odstupanje od 1 sekunde svakih 1000000 sekundi, odnosno 11.6 dana.

Koordinirano univerzalno vrijeme ◊ UTC (francuska skraćenica) (Coordinated Universal Time). Predstavlja međunarodni standard zasnovan na atomskom vremenu. Signali UTC vremena su sinkronizirani i emitiraju se ili preko zemaljskih ili satelitskih odašiljača (GPS). Računalo se s posebnim uređajem može sinkronizirati svoje interne satove na točnost od 0.1-10 ms preko zemaljskih stanica ili telefonskih linija (USA) do oko 1us za signale s GPS sustava.

3.1.2 Sinkronizacija fizičkih satova

Da bismo odredili redoslijed događaja u kolekciji procesa tj. sustavu $\mathcal{P}$ moramo izvršiti sinkronizaciju satova C_i s nekim autoritativnim, vanjskim izvorom vremena. Takvu sinkronizacija nazivamo *eksternom sinkronizacijom*.

Za istu svrhu možemo i satove C_i sinkronizirati jedan s drugim s nekim stupnjem odstupanja. To je *interna sinkronizacija*.

Preciznija definicija oba tipa sinkronizacije za zadani vremenski interval realnog vremena *I* glasi:

Eksterna sinkronizacija: unutar granica *D>0* gdje je *S* izvor UTC vremena

$$|S(t) - C_i(t)| < D, za\ i = 1,2, \dots N\ i\ za\ svaki\ t\ iz\ I$$

tj. satovi su točni unutar granica D.

Interna sinkronizacija: unutar granica *D>0*

$$\left|C_i(t) - C_j(t)\right| < D, za\ i,j = 1,2, \dots N\ i\ za\ svaki\ t\ iz\ I$$

tj. satovi se slažu unutar granica D.

Interno sinkronizirani satovi nisu nužno eksterno sinkronizirani.

Iz definicija proizlazi da unutar granica *D* eksterno sinkronizirani sustav $\mathcal{P}$ je istovremeno i interno sinkroniziran unutar granica *2D*.

Korektnost sata je najčešće vezana uz pojmove:

- monotonost $t' > t \Rightarrow C(t') > C(t)$
- odstupanje unutar granica sinkronizacijskih točaka

Npr. sat koji ide prebrzo može biti korigiran tako da se vrati njegovo vrijeme. Međutim tada se može poremetiti označavanje vremena. Npr. *make* naredba oslanja se na vremena modifikacije izvorne datoteke u odnosu na objektnu datoteku. Vraćanje vremena sata može dovesti da je vrijeme modificirane izvorne datoteke starije od vremena objektne datoteke prethodnog prevođenja. U tom slučaju make neće prevesti modificiranu izvornu datoteku.

3.1.2.1 Sinkronizacija satova u sinkronom sustavu

Razmatramo najjednostavniji slučaj interne sinkronizacije sata u dva procesa. U sinkronom sustavu poznati su slijedeći parametri:

- granice otklona lokalnih satova (drift)
- minimalno *(min)* i maksimalno (*max)* vrijeme T_{trans} prijenosa poruka
- vrijeme izvođenja svakog koraka procesa

Nesigurnost u vremenu slanja poruke je *u=(max-min)*.

Slanjem poruke *m* od strane jednog procesa šalje se u okviru poruke i vrijeme lokalnog sata.

Proces primatelj tada može minimizirati pogrešku postavljanjem vremena sata na :

t+(max+min)/2,

pri tome je maksimalno odstupanje u/2.

Općenito za sinkroni sustav u kojemu se sinkronizira *N* satova optimalna granica odstupanja koja se može postići iznosi : *u(1-1/N)* (Lunedelius i Lynch 1984).

Međutim većina sustava nije sinkrona . Npr. Interenet nije sinkroni sustav i vrijeme prijenosa je $T_{trans} = min + x$, *gdje je* $x \geq 0$.

3.1.3 The Network Time Protocol (NTP)

Network Time Protocol [Mills 1995] definira arhitekturu vremenskog servisa i protokol za distribuciju informacije točnog vremena preko interneta.

Osnovni ciljevi dizajna i osobine NTP-a su:

Omogućiti servis koji će omogućiti klijentima širom interneta točno sinkroniziranje na UTC vrijeme : Unatoč velikim i varijabilnim kašnjenjima poruka koja se javljaju u Internet komunikaciji. NTP koristi statističke tehnike za filtriranje podataka vremena i određivanje kvalitete podataka vremena dobivenog od različitih servera.

Omogućavanje pouzdanog servisa koji može preživjeti dugotrajne prekide konekcija: Postoje redundantni serveri i redundantne staze između servera. Serveri se mogu rekonfigurirati tako da nastave pružati servis i nakon ispada pojedinih servera.

Omogućavanje da se klijenti dovoljno čestu osvježavaju kako bi se odstupanja satova klijenata držala na prihvatljivoj razini. Servis je dizajniran da se skalira na veliki broj servera i klijenata.

Omogućavanje autentikacije između dijelova sustava kako bi se spriječilo slučajno ili namjerno korištenje netočnih ili nepouzdanih izvora točnog vremena.

NTP servis realiziran je korištenjem mreže servera raspoređenih širom interneta.

Serveri su spojeni u logičku hijerarhiju koja se naziva *sinkronizacijska podmreža* , a pojedini nivoi te mreže nazivaju se *stratum* (sloj). Svaki slijedeći stratum sinkronizira se sa serverima prethodnog sloja. Serveri zadnjeg sloja izvršavaju se na korisničkim radnim stanicama.

Primarni serveri – time serveri (stratum 1) spojeni su direktno na izvore točnog vremena (UTC) *(stratum 0)*. *Sekundarni serveri (stratum 2)* sikronizirani su s primarnim serverima i koriste NTP algoritam kako bi u izmjeni poruka s drugim serverima istog nivoa osigurali najbolju procjenu točnog vremena.

Satovi koji pripadaju serverima s većim stratum brojem jamče manju točnost od servera s nižim stratum brojem. Razlog su greške koje se uvode sa svakim stupnjem sinkronizacije.

Upotrijebljeni algoritmi omogućavaju da se korištenjem višestrukih izvora reduciranja jitter i detektiraju nepouzdani serveri. Sistemski satovi se discipliniraju korištenjem phase-locked loop tehnike..

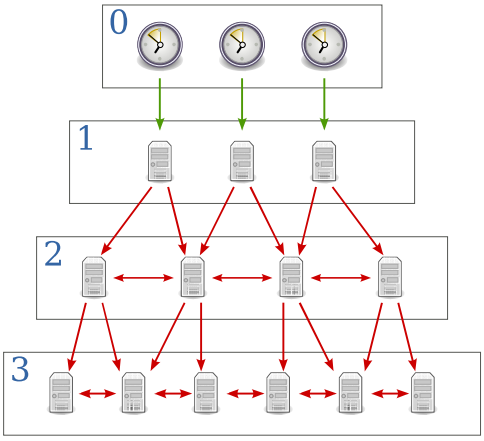
Međusobna sinkronizacija servera i klijenata obavlja se u tri moda: multicast, klijent-serever i simterični mod (peer). Poruke se šalju nepouzdanim UDP Internet transportnim protokolom.

Multicast mod koristi se u LAN mrežama visoke brzine. Jedan ili više servera multicastira [1] vrijeme ostalim računalima unutar LAN-a, koji onda podešavaju svoje satove podrazumijevajući malo kašnjenje. Na ovaj način postižu se male točnosti, ali dovoljne za niz primjena.

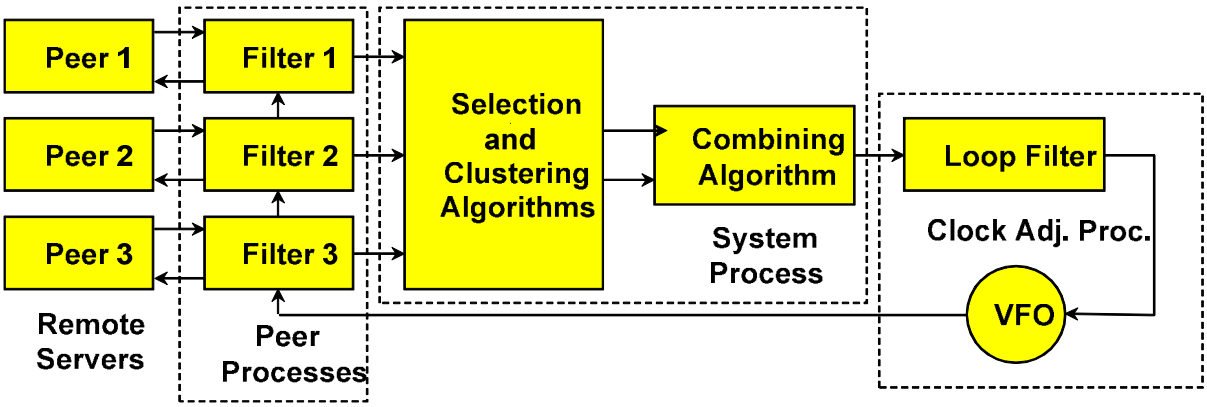
U *klijent-server modu* jedan server prima zahtjeve od ostalih računala na koje odgovara s porukom s uključenom oznakom vremena. Ovaj način omogućava preciznije sinkroniziranje nego multicast mod.

Simetrični mod (peer) se koristi za najveću točnost. Niz (min 4 za dobru procjenu) servera koji operiraju u simetričnom modu izmjenjuju poruke s oznakama točnog vremena. Točno vrijeme sadržano je asocijacijama između servera.

Moguća točnost sinkronizacije za stratum 1 je oko 200uS za LAN , a za WAN od 1-10ms.



Slika 3.2 Organizacija NTP servera korištenjem stratuma



Slika 3.3 Sinkronizacija peer servera

Slika 3.3 prikazuje sinkronizaciju više servera u simetričnom (peer) modu:

[1] Način istovremenog slanja poruka grupi primatelja na način da jednu kopiju poruke istovremeno prima grupa primatelja priključena na isti mrežni segment.

Djelovanje niza algoritama može se svesti na slijedeće grupe operacija:

- svaki peer proces u određenim intervalima komunicira s drugim peer procesima
- interval je ovisan o uočenom mrežnom jitteru i stabilnosti lokalnog sata
- korekcija (discipliniranje) sata se obavlja svake sekunde.

3.1.4 Logičko vrijeme i logički satovi

Sa stanovišta promatranja jednog procesa događaji u distribuiranom sustavu su poredani u jedinstvenom rasporedu koji se može označiti vremenima lokalnog sata procesa. Međutim kako ne možemo idealno sinkronizirati satove u distribuiranom sustavu, ne možemo općenito vremenske oznake dodijeljene od strane procesa koristiti za određivanje redoslijeda događaja.

Međutim možemo koristiti shemu koja je slična fizičkoj kauzalnosti da bismo poredali događaje koji se odvijaju u različitim procesima.

Takvo određivanje redoslijeda zasnovano je na dvije očite postavke:

- Ako su se dva događaja zbila unutar istoga procesa p_i, $i=1,2,... N$, onda se događaju u redu koji p_i može registrirati.
- kada god pošaljemo poruku između procesa, događaj slanja poruke je prije događaja primanja poruke

Lamport [1978] je parcijalni poredak dobiven s uopćavanjem ove dvije relacije nazvao *dogodio-se-prije (happened before)* relacijom. Nekad se naziva i *kauzalnim poretkom*.

Relacija *happened-before* se označava s $\rightarrow$, a formalno je definirana kako slijedi:

HB1: Ako postoji takav proces pi : $e \rightarrow_i e'$, tada $e \rightarrow e'$

HB2: Za svaku poruku m, $send(m) \rightarrow receive(m)$ – gdje je $send(m)$ događaj slanja poruke, a $receive(m)$ događaj primanja poruke

HB3: Ako su e,e' i e'' takvi događaji takvi da je $e \rightarrow e'$ i $e' \rightarrow e''$ tada je i $e \rightarrow e''$.

Ilustracija za događaje u tri procesa p_1,p_2,p_3:

- točno je da je $a \rightarrow b$ jer za proces p_1 ,$a \rightarrow_1 b$, slično za $c \rightarrow d$
- točno je da je $b \rightarrow c$ jer $m_1, b \rightarrow c$, slično $d \rightarrow f$
- kombinacijom prve dvije, $a \rightarrow c$

Međutim nije točno : $a \rightarrow e$ ili $e \rightarrow a$ pošto se odvijaju u različitim procesima , a bez lanca poruka između njih. Za događaje a i e možemo kazati da su konkurentni i to pišemo a ‖ b.

Treba pripaziti da događaje mogu inicirati i korisnici koji imaju sustav komunikacije van mrežnog slanja poruka (npr. obična glasovna komunikacija). Tada nije moguće koristiti navedene relacije.

Logički satovi ◊ Lamport je uveo jednostavan mehanizam kojim se dogodilo-se-prije redoslijed može izraziti numerički, nazvan *logički sat*. Lamportov logički sat je softverski brojač s monotonim porastom, čije vrijednosti nemaju nikakvu određenu relaciju s bilo kojim

fizičkim satom. Svaki proces p_i održava svoj vlastiti logički sat L_i , koji koristi da bi događajima dodijelivao tzv. Lamport vremenske oznake (timestamps).

Lamport vremensku oznaku događaja *e* u procesu p_i označavamo s $L_i(e)$. S $L(e)$ označavamo vremensku oznaku događaja *e* bez obzira na proces u koje me događa.

Da bi se obuhvatila relacija → (dogodio-se-prije) procesi ažuriraju svoje logičke satove i transmitiraju ih u porukama na slijedeći način:

LC1: L_i je se inkrementira prije izvođenja svakog događaja u procesu p_i : $L_i := L_i+1$

LC2: (a) Kada proces p_i pošalje poruku *m* onda poruka m nosi sa sobom vrijednost $t= L_i$.

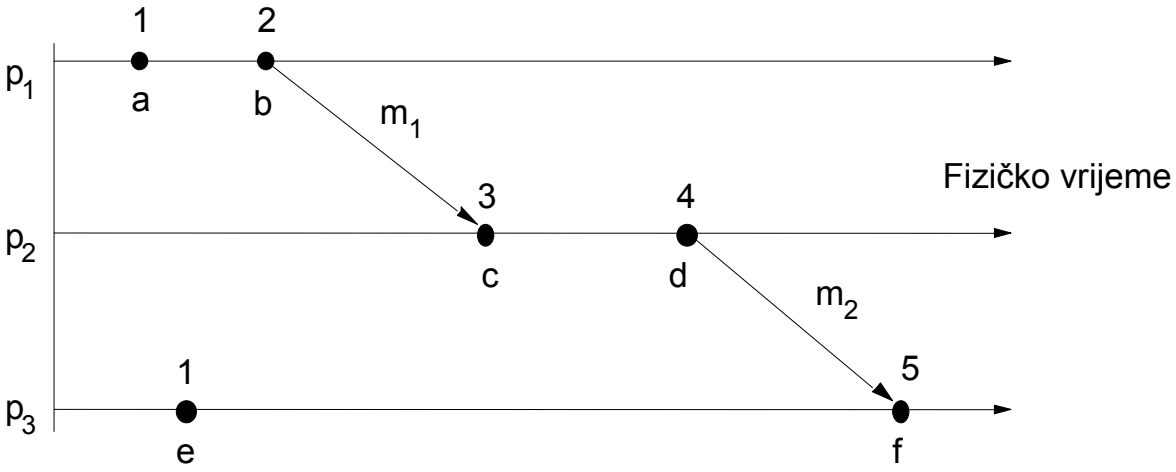
(b) Na prijemu (m,t), proces p_j računa $L_j := max(L_j , t)$, a nakon toga primjenjuje LC1 prije davanja vremenske oznake događaju *receive(m)*.

Iako se sat uobičajeno inkrementira za 1 moguća je upotreba bilo koje pozitivne vrijednosti.

Može se pokazati da vrijedi za bilo koju sekvencu događaja : $e \rightarrow e' \Rightarrow L(e) < L(e')$

Međutim ne možemo tvrditi obrnuto tj. nije točno $L(e) < L(e') \Rightarrow e \rightarrow e'$

Međutim s Lamport satom možemo ipak odrediti redoslijed slanja poruka !



Slika 3.4 Korištenje logičkog sata

Slika 3.4 ilustrira primjenu logičkog sata. Svi procesi p_1,p_2 i p_3 na početku inicijaliziraju svoje logičke satove na 0. Primjenom pravila ažuriranja i transmisije vremenskih oznaka dobivamo numeraciju kao na slici. Primjetite da je $L(e)<L(b)$, ali $e \parallel b$.

Vektorski satovi ◊ Mattern [1989] and Fidge [1991] razvili su vektorske satove kako bi otklonili nedostatak Lamportova sata tj. nemogućnost zaključivanja $L(e) < L(e') \Rightarrow e \rightarrow e'$.

Vektorski sat za sustav od *N* procesa je niz od *N* cjelobrojnih vrijednosti. Svaki proces održava svoj vlastiti vektorski sat V_i, koji koristi za označavanje lokalnih događaja. Slično kao kod Lamporta , poruka nosi sa sobom vektorsku vremensku oznaku.

Jednostavna pravila za ažuriranje sata su:

VC1: Inicijalno $V_i[j]=0$, za $i,j=1,2...N$.

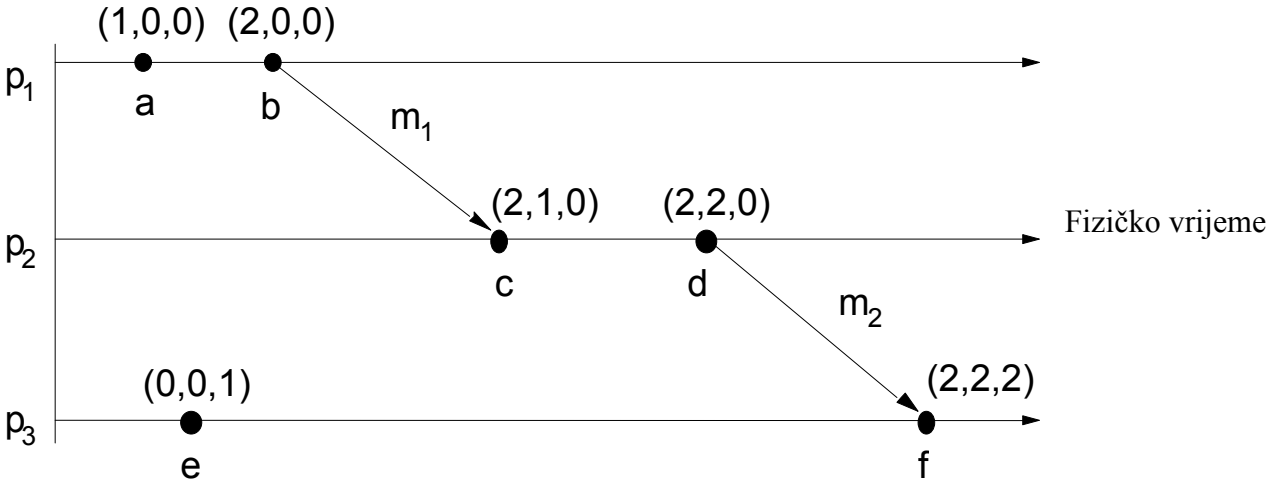
VC2: Neposredno prije nego što p_i označi događaj postavlja se $V_i[i] := V_i[i] +1$

VC3: p_i uključuje vrijednost $t= V_i$ *u svaku poslanu poruku.*

VC4: Kada p_i primi oznaku vremena t u poruci , postavlja $V_i[j]:=max(V_i[j],t)$ (maksimum po komponentama vektora)

Za vektorski sat V_i , $V_i[i]$ odgovara broju događaja koje je proces p_i označio vremenskom oznakom.

Razmotrimo primjer označavanja :



Slika 3.5 Korištenje vektorskog sata

Vektorske oznake uspoređujemo na slijedeći način: (↔ znači onda i samo onda)

$V=V' \leftrightarrow V[j]= V'[j]$, *za* $j=1,2,...,N$

$V \leq V' \leftrightarrow V[j] \leq V'[j]$, *za* $j=1,2,...,N$

$V<V' \leftrightarrow V \leq V'$ *i* $V \neq V'$

vrijedi : $V(e)< V(e') \Rightarrow e \rightarrow e'$ i $e \rightarrow e' \Rightarrow V(e)< V(e')$.

Analiza primjera sa slike:

- $a \rightarrow f$ slijedi iz V(a)<V(f)
- $c \parallel e$ slijedi jer ne vrijedi ni $V(c) \leq V(e)$, *nit* $V(e) \leq V(c)$

Nedostatak vektorskog sata je povećana upotreba memorije i transfera podataka u odnosu na Lamport logički sat.

3.2 Koordinacija

U ovom poglavlju bavimo se temama vezanima za pitanja koordinacije procesa u pristupu dijeljenim resursima u okviru distribuiranog sustava unatoč mogućim greškama u dijelovima sustava.

Razmatraju se algoritmi kojima se za kolekciju procesa osigurava uzajamno isključenje (mutual exclusion) i koordinacija pristupa dijeljenim resursima. Zatim se razmatra pojam izbora (election) kod distribuiranih sustava, tj. kako se grupa procesa može složiti oko izbora novog koordinatora (ili vođe) nakon otkaza (greške) prethodnog koordinatora. Zadnji dio vezan je za pojam konsenzusa i problem njegovog postizanja u distribuiranom sustavu.

3.2.1 Uzajamno isključivanje u distribuiranim sustavima (mutual exclusion)

Ako kolekcija procesa dijeli resurs ili kolekciju resursa, često je potrebno uzajamno isključivanje u pristupu kako bi se održala konzistentnost. Ovaj problem vežemo za pojam *kritične sekcije.* U operacijskim sustavima i na jednom računalu ovaj problem rješavamo pomoću dijeljenih varijabli ili drugih mehanizama jezgre operativnog računala. Međutim u distribuiranom sustavu nemamo niti dijeljene varijable niti jedinstvena jezgra OS. u distribuiranim sustavima uzajamno isključivanje potrebno je realizirati izmjenom poruka.

U pojedinim slučajevima server može imati mehanizme kojima osigurava uzajamno isključenje klijenata u pristupu resursima. Međutim koordinacija dijeljenja resursa od strane peer procesa zahtijeva dodatne mehanizme.

Algoritam za uzajamno isključivanje

Razmatramo sustav od N procesa p_i, $i=1,2,... N$ koji ne dijele varijable. Procesi pristupaju zajedničkim resursima, ali u samo u kritičnim sekcijama. Radi pojednostavljenja promatra se samo slučaj s jednom kritičnom sekcijom. Dalje, pretpostavljamo asinkroni sustav s procesima koji ne ispadaju i da je prijenos poruka pouzdan tj. da se svaka poslana poruka prije ili kasnije dostavlja s neizmijenjenim sadržajem u jednom primjerku.

Protokol na nivou aplikacije za izvršavanje kritične sekcije je:

```
enter()             // uđi u kritičnu sekciju  - blokiraj ostale procese
resourceAcesses()   // pristupaj dijeljenim resursima
exit()              // napusti kritičnu sekciju – drugi procesi sad mogu ući
```

Osnovni zahtjevi za uzajamno isključivanje su:

ME1: (*zaštita*) Samo jedan proces se može izvršavati u kritičnoj sekciju (CS) u isto vrijeme

ME2: (*liveness*) Zahtjevi za ulaz i izlaz iz kritične sekcije se odobravaju

Uvjet ME2 podrazumijeva i odsustvo mrtvih točaka (deadlocka) i skapavanja (starvation). Mrtve točke nastaju kad jedan proces ne može izaći iz kritične sekcije jer ovisi o blokiranom procesu (međusobna ovisnost). Skapavanje znači da neki proces neprestano čeka na ulaz u kritičnu sekciju tj. nikako ne dolazi na red. Odsustvo skapavanja nazivamo stanje nepristranosti (fairness). Nepristranost je vezana i za redoslijed kako procesi ulaze u kritičnu sekciju. Nije moguće koristiti redoslijed zasnovan na vremenu kad je proces tražio ulazak u

kritičnu sekciju zbog nedostatka globalnog sata. Međutim za pristigle poruke kojima se traži pristup dijeljenom resursu moguće je koristiti redoslijed zasnovan na relaciji dogodilo-se-prije (logički i vektorski sat).

ME3: (→ *poredak*) Redoslijed ulaska u kritičnu sekciju određen je time da prednost ima zahtjev koji se *dogodio-prije* .

Ako se poštuje ME3 i ako između svih zahtjeva postoji relacija → , onda se ne može dogoditi da jedan proces uđe u kritičnu sekciju dva puta za vrijeme dok drugi proces čeka na ulazak.

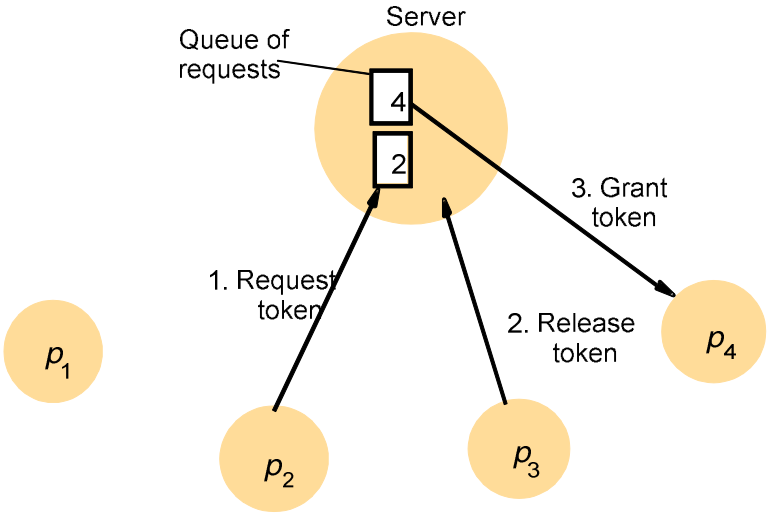
Višenitni proces može nastaviti s radom dok jedna njegova nit čeka dozvolu pristupa. Tijekom toga on može *poslati poruku* drugom procesu, koji nakon toga također čeka na ulazak u kritičnu sekciju. ME3 specificira da će prvi proces dobiti pristup prije drugog.

Performanse algoritama za uzajamno isključivanje evaluiramo po slijedećim kriterijima:

- konzumirani bandwidth – proporcionalan broju poruka poslanih u svakoj *entry* ili *exit* operaciji
- po kašnjenju klijenta (client delay) uzrokovanom procesom u svakoj *entry* ili *exit* operaciji
- efektu *algoritma* na propusnost sustava – sinkronizacijsko kašnjenje. Procesi gube dio vremena raspoloživosti dijeljenog resursa na sinkronizaciju izlaza/ulaza u kritičnu sekciju .

Ne uzimamo u obzir samo implementaciju pristupa resursu koja se izvodi unutar CS. Međutim uzimamo u obzir da klijent procesi u nekom konačnom vremenu obavljaju operacije unutar CS.

Algoritam centralnog servera ◊ Najlakši način rješavanja uzajamnog isključivanja je primjena servera koji daje dopuštenje ulaska u kritičnu sekciju. Slika 3.6 prikazuje upotrebu takvog servera.



Slika 3.6 Algoritam centralnog servera

Da bi ušao u kritičnu sekciju proces šalje poruku serveru i čeka na odgovor. Konceptualno odgovor se sastoji od tokena koji označava dopuštenje ulaska u kritičnu sekciju. Ako nijedan drugi proces nema token u vrijeme zahtjeva, onda server odgovara odmah i odobrava token. Ako token drži drugi proces onda stavlja zahtjev u red čekanja. Na izlasku iz kritične sekcije proces šalje poruku serveru i vraća mu token.

Ako red čekanja procesa nije prazan, onda server bira zahtjev s najstarijim ulaskom , skida ga iz reda i obavještava odgovarajući proces te mu šalje token.

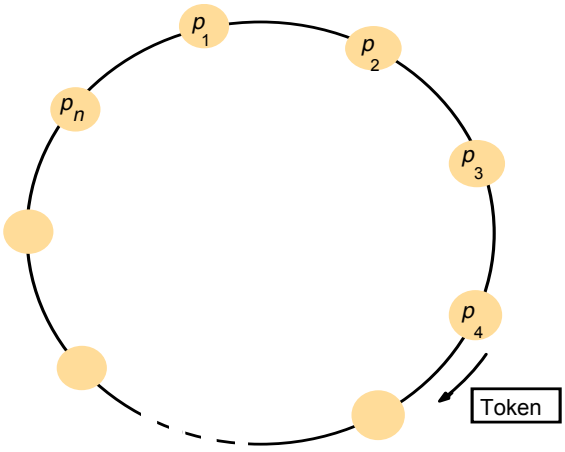
Ovakav način dodjeljivanja prioriteta (tokena) poštuje pravila ME1 i ME2 (zaštita i skapavanje) , ali ne i pravilo ME3 !

Performanse algoritma : ulazak u kritičnu sekciju uzima dvije poruke (zahtjev-odobrenje, *request-grant*) čak iako nema procesa na čekanju. Uvedeno kašnjenje je round-trip vrijeme poruka.

Izlazak iz kritične sekcije je slanje jedne *otpusti* (*release*) poruke. Podrazumijevajući asinkrono slanje poruka to ne ometa proces da nastavi s radom tj. ne uvodi dodatno sinkronizacijsko kašnjenje.

Ovakav server može postati usko grlo sustava.

Kružni algoritam (ring-based algorithm) ◊ Jedan od najjednostavnijih načina postizanja uzajamnog isključivanja između N procesa bez potrebe dodatnog procesa (servera) je da se procesi poredaju u logički krug. To zahtijeva samo da svaki proces p_i ima komunikacijski kanal prema drugom procesu $p_{(i+1) \bmod N}$ u krugu.Ideja je da se isključivanje postiže slanjem poruke s tokenom u krug u jednome smjeru – npr. kretanja sata. Toplogija prstena ne mora biti vezana za fizičke interkonekcije sustava.



3.7 Kružni algoritam

Ako proces koji primi token ne zahtijeva ulazak u kritičnu sekciju, onda odmah prosljeđuje token svome susjedu u krugu. Ako proces zahtjeva ulazak u kritičnu sekciju onda zadržava token, ulazi u kritičnu sekciju i predaje token tek nakon izlaska iz kritične sekcije.

lako je potvrditi da su zahtjevi ME1 i ME2 postignuti s ovim algoritmom, ali da se token ne dodjeljuje po dogodilo-se-prije redoslijedu (procesi mogu izmjenjivati poruke neovisno o rotaciji tokena).

Ovaj algoritam konstantno troši mrežnu propusnost (osim kad je proces unutar kritične sekcije). Sinkronizacijsko kašnjenje ulaska u CS je od 0 (upravo je dobio token) do N poruka (upravo otpustio token). Izlazak iz kritične sekcije zahtijeva jednu poruku.

Algoritam koji koristi multicast i logičke satove ◊ Ricart i Agrawala [1981] razvili su algoritam za implementaciju uzajamnog isključenja između N peer procesa koji je zasnovan na multicastu. Osnovna ideja je da proces koji zahtijeva ulazak u kritičnu sekciju multicastira poruku zahtjeva, a može ući tek kada svi procesi odgovore na tu poruku. Uvjet pod kojim svi procesi odgovaraju dizajniran je da osigura zahtjeve ME1-ME3.

Procesi $p_1,p_2,...,p_N$ označeni su jedinstvenim brojčanim oznakama. Pretpostavlja se da posjeduju međusobne komunikacijske kanale i da svaki proces p_i ima svoj Lamport logički sat koji ažurira prema pravilima LC1 i LC2. Poruke koje zahtijevaju ulaz su u formi <*T,pi*> gdje je T vremenska oznaka pošiljaoca, a pi njegova identifikacija.

Svaki proces zapisuje svoje stanje bivanja van *CS* kao *RELEASED* , stanje čekanja ulaza kao *WANTED* i bivanja unutar kritične sekcije kao *HELD* u varijabli *state.* Slijedi prikaz kompletnog protokola za pojedini proces (za proces p_j):

On initialization

state := RELEASED;

To enter the section

state := WANTED; |

Multicast *request* to all processes; | request processing deferred here

T := request's timestamp; |

Wait until (number of replies received = (*N* – 1));

state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$

if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))

then

queue *request* from p_i without replying;

else

reply immediately to p_i;

end if

To exit the critical section

state := RELEASED;

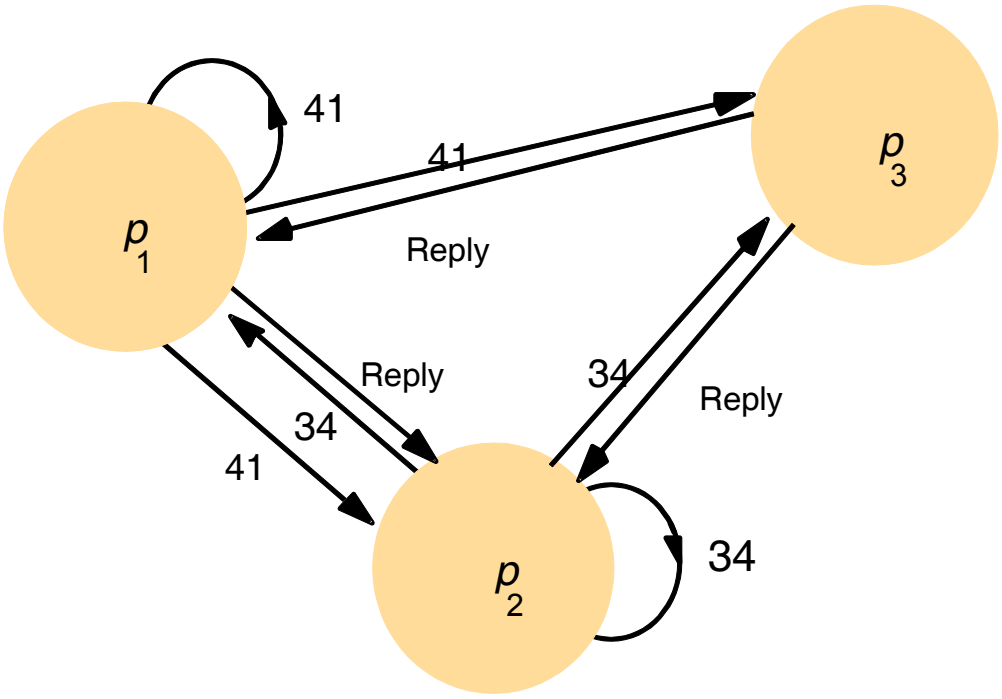
reply to any queued requests;

Ako proces zahtijeva ulaz i stanja svih ostalih procesa je *RELEASE*, onda će svi procesi odgovoriti procesu koji zahtijeva ulaz i on će dobiti odobrenje za ulaz. Ako su neki procesi u stanju *HELD* , tada ti procesi neće odgovoriti procesu koji je zahtijevao ulaz u kritičnu sekciju i isti u nju neće moći ući.

Ako dva procesa traže u isto vrijeme pristup kritičnoj sekciji tada će proces koji sadržava najnižu vremensku oznaku prvi sakupiti N-1 odgovora.

Ako zahtjevi imaju istu Lamport oznaku prednost se dodjeljuje na osnovu identifikatora procesa. Treba primijetiti: kad proces ulazi u postupak zahtjeva za ulaskom u CS on odlaže obradu zahtjeva drugih procesa sve dok ne završi odašiljanje svog zahtjeva i ne pohrani *T* vremensku oznaku.

Algoritam je ilustriran na situaciji s tri procesa p_1,p_2,p_3 kako je to prikazano na slijedećoj slici.



Slika 3.8 Izmjena poruka za Ricart i Agrawala algoritam

Pretpostavimo da *p3* nije zainteresiran za ulazak u kritičnu sekciju, a da *p1* i *p2* istovremenu traže ulazak. Vremenska oznaka od *p1* je 41, a od *p2* 34. kada *p3* dobije njihove zahtjeve, odmah će odgovoriti.Kada p2 dobije zahtjev od p1, naći će da njegov zahtjev ima nižu vremensku oznaku i neće odgovoriti. Kada p1 dobije zahtjev od p2 naći će da je p2-ova vremenska oznaka niža, te će odmah odgovoriti. Na primitku drugog odgovora , p2 ulazi u CS. Kada p2 izađe iz kritične sekcije, odgovorit će na p1 zahtjev (drži ga u čekanju) i time mu omogućiti ulaz u CS.

Dobivanje odobrenja pristupa traje 2(N-1) poruke tj. N-1 za multicast i N-1 za odgovore. Međutim ako je multicast hardverski podržan onda je potrebna samo jedna poruka zahtjeva, dakle ukupno N poruka. Ovaj algoritam generira veliki mrežni promet. Međutim, sinkronizacijsko kašnjenje zahtjev-odobrenje je samo jedan RTT (bez kašnjenja multicasta).

Performanse algoritma se mogu ubrzati na način da proces koji nije dobio ni jednu poruku zahtjeva od drugih procesa može ponovo ući iz CS iz koje je upravo izašao.

Otpornost na greške ◊ Glavne točke koje trebamo razmotriti za navedene algoritme u evaluaciji otpornosti na greške su:

- Što se događa ako se poruka izgubi
- Što se događa ako se proces sruši

Nijedan od prije navedenih algoritama ne tolerira gubitak poruke. Kružni algoritam ne tolerira pad bilo kojeg pojedinačnog procesa. Algoritam s centralnim serverom može tolerirati rušenje procesa koji ne drži niti zahtijeva token. Ricart i Argawala algoritam može se adaptirati da tolerira rušenje procesa na način da za takav proces uzimaju da implicitno daje dozvole svim procesima.

3.2.2 Izbori vođe (leader election)

Algoritam kojim biramo jedinstven proces za vođenje određene uloge nazivamo *izborni algoritam* (election alghoritm). Ovaj algoritam koristima kako bismo odabrali koji od procesa će igrati ulogu servera. Bitno je da se svi ostali procesi slažu u izboru vođe. Ako se proces koji igra ulogu servera odluči na umirovljenje, provode se novi izbori.

Kaže se da proces *raspisuje izbore* (calls the election) ako poduzima akciju koja inicira pozivanje algoritma izbora. Individualni algoritam raspisuje samo jedne izbore u jednom vremenskom trenutku, ali N procesa može u principu raspisati N konkurentnih izbora. U svakom vremenskom trenutku proces p_i je *učesnik* (participant) izbora, tj. uključen je izborni algoritam, ili *nije učesnik* (non-participant), tj. nije uključen u niti jedan izborni postupak.

Bitno je da rezultat izbora bude jedinstven, bez obzira koliko procesa istovremeno raspisalo izbore. Npr. može se dogoditi da dva procesa istovremeno zaključe da proces koordinator ne funkcionira ispravno i da oba dva raspišu izbore.

Jedan od zahtjeva je da se za proces server bira onaj s najvećim identifikatorom izbora. Pri tome identifikatori moraju biti jedinstveni i s određenim poretkom. Npr. može se birati proces s najmanjim opterećenjem , tako da se za identifikator koristi $<1/opterećenje,i>$. Pri tome opterećenje>0, a i je npr. PID procesa koji nam služi za poredak procesa s istim opterećenjem.

Svaki proces p_i, $i=1,2,\ldots N$,.posjeduje varijablu $elected_i$, koja sadržava identifikator izabranog procesa. Kada proces prvi put učestvuje u izborima ta varijabla postavlja se na nedefiniranu vrijednost označenu kao $\perp$.

Zahtjevi na svako pokretanje algoritma izbora su:

E1: (sigurnost izbora-safety)

Proces učesnik p_i ima $elected_i = \perp$ ili $elected_i = P$, gdje je P proces koji na kraju izbora nije u stanju rušenja i koji ima najveći identifikator.

E2: (živost-liveness)

Svi procesi p_i učestvuju u izboru i eventualno postavljanju $elected_i \neq \perp$ ili se ruše.

Treba primjetiti da mogu postojati procesi p_i koji nisu učesnici, a koji u varijabli $elected_i$ drže vrijednost identifikatora prethodno izabranog procesa.

Performanse algoritma izbora mjere se njegovim korištenjem mrežne propusnosti (proporcionalno broju poslanih poruka) i kompletnom vremenu izvršenja algoritma.

Kružni izborni algoritam (ring-based) ◊ Jedan od kružnih algoritama je Chang i Roberts [1979] algoritam. Svaki proces pi posjeduje komunikacijski kanal prema slijedećem procesu u $p_{(i+1) \bmod N}$ u krugu i sve poruke se šalju u smjeru kazaljke na satu. Podrazumijevamo da se ne događaju pogreške u radu i da je sustav asinkron. Cilj algoritma je izbor jednog procesa s najvećim identifikatorom - proces *koordinator*.

Inicijalno svaki proces se označava kao *ne-učesnik* u izborima. Bilo koji proces može inicirati izbore. Nakon iniciranja sebe označava kao *učesnika* i stavlja vrijednost svoga identifikatora u *izbornu poruku* koju šalje prvom susjedu u smjeru kretanja kazaljke na satu.

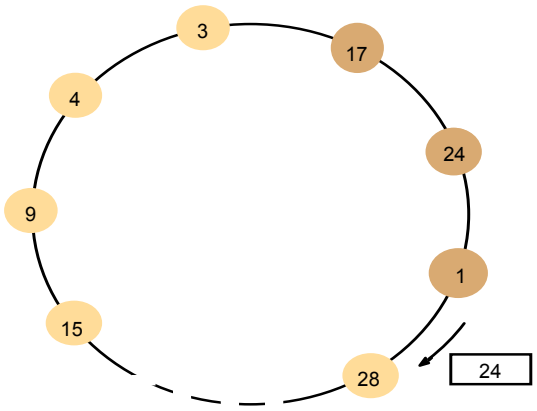
Kada proces primi *izbornu poruku* on uspoređuje primljeni identifikator sa svojim vlastitim.

- Ako je primljeni identifikator veći onda samo proslijedi poruku slijedećem susjedu.
- Ako je pristigli identifikator manji:

- ako primatelj nije učesnik onda on u poruku smješta vlastiti identifikator i šalje poruku dalje
- ako je već učesnik ne šalje poruku dalje.

U bilo kojem slučaju prosljeđivanja poruke, proces samog sebe označava kao *učesnika*.

Ako je primljeni identifikator zapravo vlastiti identifikator, onda je to najveći identifikator i taj proces postaje *koordinator*. Koordinator sebe označava kao ne-učesnik i šalje poruku *izabran ,* koja sadržava njegov identitet prvom susjedu. Kada proces p_i primi poruku *izabran* , prima informaciju o novom koordinatoru i sebe označava kao *ne-učesnik*. Proces p_i šalje dalje istu poruku osim ako nije novi *koordinator*.



3.9 Kružni izbor koordinatora (proces 17 pokrenuo izbor – učesnici zatamnjeni)

E1 zahtjev je ispunjen jer se osigurava da proces s najvećim identifikatorom bude izabran za vođu. E2 je također ispunjen (ako nema grešaka u slanju poruka).

Ako samo jedan proces inicira selekciju slučaj s najgorim performansama je ako najveći identifikator ima prvi susjed suprotno kretanju kazaljke na satu.

Tada je potrebni N-1 poruka da se pristupi tom susjedu, koji neće oglasiti selekciju sve dok poruke ne naprave još jedan krug (mora primiti vlastiti identifikator!), tj dodatno N poruka. Poslije toga poruka *izabran* šalje se N puta. Ukupan broj poruka je 3N-1. Vrijeme slanja je također proporcionalno 3N-1 jer se poruke šalju sekvencijalno.

Iako je dobar za ilustraciju kružni izbor je od male praktične važnosti jer ne tolerira greške u slanju poruka ni srušene procese.

Bully algoritam ◊ (Garcia-Molina 1982) dozvoljava rušenje procesa tijekom izbora, iako podrazumijeva pouzdan prijenos poruka među procesima. Ovaj algoritam podrazumijeva da je sistem sinkron tj. koristi timeout za detekciju srušenog procesa. Razlika između kružnog i bully algoritma je i slijedeća:

- kružni: svaki proces *apriori* zna za svog susjeda u smjeru kazaljke na satu, a ne zna ništa o identifikatorima drugih procesa
- bully: svaki proces zna za procese s većim identifikatorom (koji i kako s njima komunicirati)

Ovaj algoritam koristi tri tipa poruka. Poruka *izbori* šalje se za oglašavanje izbora. Poruka *odgovor* šalje se kao odgovor na poruku *izbori*. Poruka *koordinator* šalje se za objavu identiteta izabranog procesa – novog *koordinatora*.

Proces šalje poruku izbora kada pomoću mehanizma timeouta primijeti nedostatak u radu koordinatora. To može otkriti više procesa istovremeno.

Kako je sustav sinkron moguće je konstruirati pouzdan detektor grešaka.

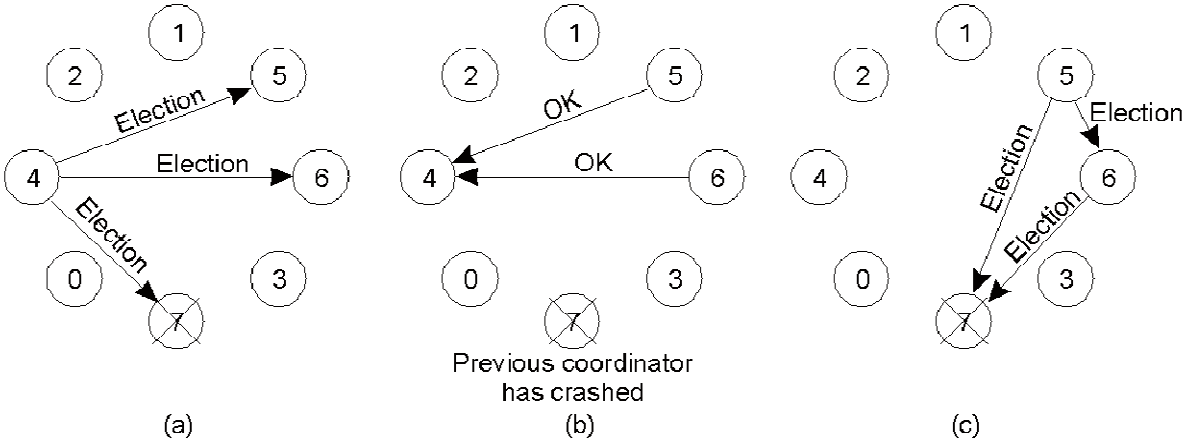
Zadaje se maksimalno kašnjenje prijenosa T_{trans} i maksimalno vrijeme procesiranja poruke $T_{process}$. Stoga je vrijeme $T=2\ T_{trans} + T_{process}$ je gornja vremenska granica od slanja poruke drugom procesu do njenog prijema. Ako ne stigne odgovor u vremenu T onda lokalni detektor pogreške može objaviti pogrešku u radu primatelja poruka.

Opis algoritma:

Algoritam počinje kada jedan proces p_i detektira da proces koordinator ne odgovara na njegove poruke (timeout). Tada pi inicira izbore uz slijedeća pravila:

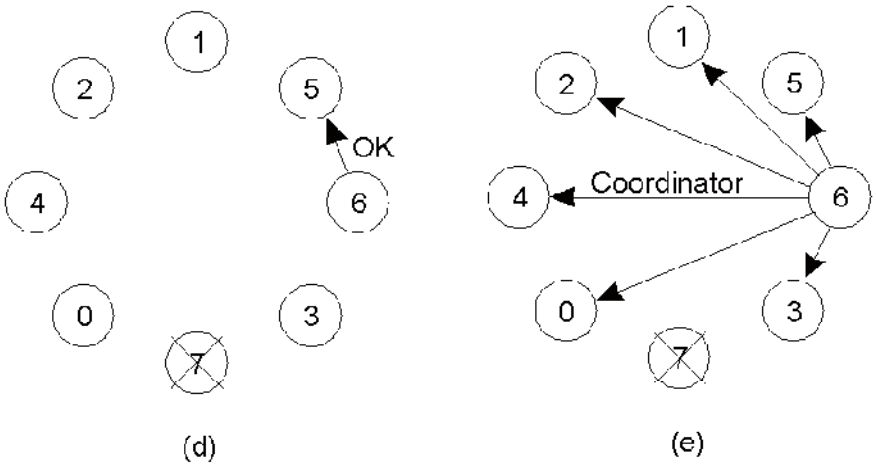
1. p_i šalje poruku *izbori* svakom procesu s većim id
2. Ako nakon timeout vremena T nitko ne odgovori , p_i dobiva izbore i postaje koordinator -> šalje poruku *koordinator* svakom procesu.
3. Ako dobije poruku *odgovor* od nekog procesa , pi čeka daljni interval vremena T' da dobije poruku *koordinator* od novog koordinatora. Ako je ne primi započinje novi postupak izbora
4. Kad proces pi primi poruku *koordinator* postavlja svoju varijablu $elected_i$ i prihvaća novog koordinatora
5. Ako proces primi poruku *izbori*, vraća poruku *odgovor* i započinje nove izbore ako već i sam to nije učinio.
6. Ako se prethodni proces koordinator oporavi i ako zaključi da ima veći id od svih procesa on šalje svima poruku *koordinator* (bully)

Kompleksnost poruka : najbolji slučaj n-2; najgori slučaj: $O(n^2)$.



Slika 3.10 Bully algoritam - ilustracija 1/2

(a) Kada proces "zapazi" da trenutni koordinator ne odgovara na poruke (4 zaključi da je 7 srušen), šalje poruku *izbori* svakome procesu s višom numeracijom. Ako• proces s većom numeracijom (5,6) odgovori *(b)* na poruku *izbori* za 4 izbor je poništen i proces s višom numeracijom saziva *(c)* svoje izbore (5,6) – (bully ponašanje).



Slika 3.11 Bully algoritam - ilustracija 2/2

(d) 6 odgovara 5 i isti poništava svoje izbore. 6 pobjeđuje na izborima

Ako originalni koordinator (7) se povrati u život, on jednostavno šalje svima poruku da on postaje koordinator jer on zna da ima najvišu numeraciju.

4. KONSENZUS I VEZANI PROBLEMI

Gledano na nivou procesa problem se sastoji u tome kako se procesi mogu *sporazumjeti* u odabiru neke vrijednosti nakon što je ista predložena od jednog ili više procesa.

Npr. kako se računala koja kontroliraju motore svemirskog broda mogu složiti da svi odluče "nastavi" ili da svi izvedu "prekini", nakon što su sva računala predložila neku od tih akcija. Ili, u izborima vodećeg procesa svi se procesi moraju složiti oko izbora vodećeg procesa.

Postoje razni protokoli za postizanje sporazuma, a u ovom poglavlju obradit će se karakteristični problemi vezani za postizanje sporazuma

4.1 Problem konsenzusa

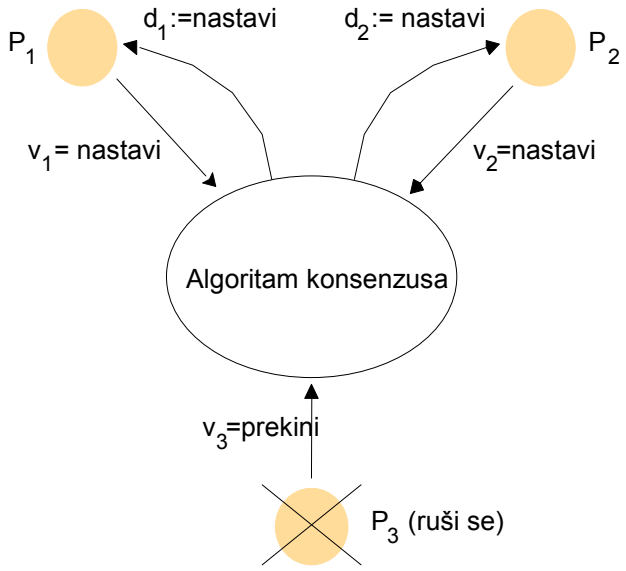
Promatramo distribuirani sustav u izvršavanju koji je kolekcija $\mathcal{P}$ koja se sastoji od N procesa p_i, $i=1,2,... N.$ i koji komuniciraju izmjenom poruka.

Jedan od važnih zahtjeva u mnogim praktičnim situacijama je da se sporazum treba postići i uz prisustvo pogrešaka. Za početak možemo pretpostaviti pouzdanu komunikaciju, ali da proces može biti neispravan. Proces se može srušiti ili uzrokovati proizvoljne pogreške (bizantske pogreške). Ukupno f od N procesa može biti neispravno.

U slučaju proizvoljnih pogrešaka bitna je činjenica da li procesi mogu digitalno potpisivati poruke koje šalju. Zasad pretpostavimo da to nije slučaj.

Definicija problema konsenzusa ◊ Da bi se postigao sporazum, svaki proces p_i počinje u stanju *neodlučeno* i *predlaže* jednu vrijednost v_i iz skupa D.

Procesi komuniciraju jedan s drugim izmjenjujući vrijednosti. Nakon toga svaki proces postavlja vrijednost svoje *varijable odluke* d_i. Kad to učini nalazi se u stanju *odlučeno* u koje više ne mijenja stanje *di*. Slika 4.1 prikazuje primjer tri procesa uključena u algoritam konsenzusa. Dva procesa predlažu "nastavi", a treći "prekini" i nakon toga se sruši. Dva procesa koja su nastavila ispravno raditi odlučuju se na odluku "nastavi".



Slika 4.1 Konsenzus za tri procesa

Zahtjevi na izvršavanje algoritma konsenzusa je da se mora pridržavati slijedećih zahtjeva:

Terminacija: Svaki ispravni proces prije ili kasnije postavlja svoju varijable odluke

Sporazum: Vrijednost varijable odluke je za sve ispravne procese ista

Integritet: Ako su svi korektni procesi predložili istu vrijednost, tada svaki proces u stanju *odlučeno* mora imati tu vrijednost

Manje zahtjevna definicija integriteta bi bila da svi procesi imaju vrijednost koju je predložio neki od ispravnih procesa (pogodna za neke tipove aplikacija).

Razmatranje jednog algoritma za sustav u kojemu procesi ne mogu biti neispravni je slijedeće:

Promatramo grupu od *N* procesa. Svaki proces pouzdano multicastira svim procesima svoj prijedlog odluke. Svaki proces čeka dok ne primi *N* poruka (uključujući i svoju). Tada evaluira funkciju većina(v1,...,vn) koja vraća najučestaliju vrijednost ili specijalnu vrijednost $\perp$, ako ne postoji većinska vrijednost.

Terminacija je garantirana pouzdanošću multicast operacije. Sporazum i integritet garantirani su definicijom funkcije većine i svojstvom integriteta pouzdanog multicasta. Svaki proces prima isti skup vrijednosti i svaki proces izračunava istu funkciju nad tim vrijednostima. *Većina* je samo jedna od mogućih funkcija. Ako su vrijednosti *vi* poredane onda se može koristiti i funkcija *minimum* ili *maksimum*.

Ako se procesi mogu rušiti onda zbog problematične detekcije srušenog procesa nije odmah jasno da li se algoritam konsenzusa može terminirati. Za asinkroni sustav moguće je da se ne terminira.

Problem bizantinskih generala ◊ Neformalna definicija problema bizantinskih generala [Lamport i ostali,1982] glasi: tri ili više generala trebaju se složiti da napadnu ili uzmaknu. Jedan od njih je "zapovjednik" tj. izdaje naređenje.

Potreban nam je algoritam koji omogućava slijedeće:

- Svi ispravni podređeni generali moraju odlučiti za istu naredbu (unisono djelovanje)
- Ako je zapovjednik ispravan tada svi ispravni podređeni generali izvršavaju zadanu naredbu

Međutim, jedan od generala može biti "izdajnik"- neispravan. Ako je zapovjednih izdajnik on će jednom generalu dati npr. zapovijed napada, a drugom uzmaka. Ako je podređeni general izdajnik, on svom jednom peer-u kaže da mu je zapovjednik naredio napad, a drugome da mu je zapovjednik naredio uzmak. Problem bizantinskog generala razlikuje se od konsenzusa u tome što istaknuti proces daje vrijednost za koju se ostali trebaju složiti (umjesto da svi predlažu). Zahtjevi za algoritam su:

Terminacija: svaki ispravni proces prije ili kasnije postavlja svoju varijable odluke

Sporazum: Vrijednost varijable odluke je za sve ispravne procese ista

Integritet: Ako je zapovjednik ispravan, tada svi korektni procesi odabiru vrijednosti koju je predložio zapovjednik.

Primijetite da integritet implicira sporazum kada je zapovjednik ispravan, ali zapovjednik i ne mora biti ispravan.

Interaktivna konzistencija ◊ Predstavlja jednu od varijanti konsenzusa u kojoj svaki proces predlaže jednu vrijednost. Cilj algoritma je da se postigne konsenzus oko vektora vrijednosti, tj. vrijednosti za svaki od procesa (vektor odluke). Npr. cilj može biti da svi procesi dobiju istu informaciju oko njihovog stanja. Zahtjevi za interaktivnu konzistenciju su:

Terminacija: Svaki ispravni proces prije ili kasnije postavlja svoju varijable odluke

Sporazum: Vrijednost vektora odluke je za sve ispravne procese ista.

Integritet: Ako je pi ispravan, tada svi procesi odlučuju da je vi i-ta komponenta vektora odluke.

Postoje relacije kojima je moguće C – konsenzus , BG – bizantinski generali, IC –interaktivni konsenzus izvesti jedno iz drugoga. Time se ponekad daju smanjiti troškovi implementacije.

4.2 Konsenzus u sinkronom sustavu

Ovo poglavlje prikazuje jedan od algoritama koji koristi multicast protokol za rješavanje konsensuza u sinkronom sustavu. Algoritam podrazumijeva da je do f od N procesa srušeno. Da bi se postigao konsenzus svaki ispravni proces sakuplja vrijednosti od drugih procesa. Algoritam treba $f+1$ rundi te u svakoj korektni procesi B – multicastiraju međusobno vrijednosti. Najviše f procesa može se srušiti, u najgorem slučaju tijekom rundi. Međutim algoritam garantira da nakon kraja svih rundi svi ispravni procesi su u stanju složiti se oko odluke.

Slika 4.2 prikazuje jedan od algoritama za konsenzus u sinkronom sustavu [Dolev, Strong 1983, prezentacija Attiya i Welch - 1998].

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round *r*)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Slika 4.2 Algoritam za konsenzus u sinkronom sustavu

Diskusija algoritma ◊ Varijabla $Values^r_i$ sadržava skup predloženih vrijednosti poznatih procesu *pi* na početku runde *r*. Svaki proces multicastira skup vrijednosti koje nije poslao u prethodnim rundama. U istoj rundi prima multicastom vrijednosti od drugih procesa te registrira sve nove vrijednosti. Nije prikazano na slici, ali trajanje svake runde ograničeno je na maksimalno vrijeme koje je potrebno da ispravni proces multicastira poruku. Nakon *f+1* rundi svaki proces bira minimalnu vrijednost od svih vrijednosti koje je primio kao njegovu vrijednost odluke.

Terminacija je očigledna iz razloga što je sustav sinkron. Kako bi provjerili korektnost algoritma potrebno je pokazati da svaki proces sadrži iste vrijednosti nakon finalne runde. Sporazum i integritet se postiže korištenjem funkcije *minimuma*.

Korektnost ćemo pokušati pokazati osporavanjem suprotne tvrdnje: dva procesa razlikuju se u finalnim vrijednostima varijabli.

Na kraju svih rundi neki korektni proces *pi* posjeduje vrijednost *v* koju neki drugi proces *pj* (i≠j) ne posjeduje. Jedino objašnjenje za prethodno je da je neki treći proces *pk* uspio vrijednost poslati procesu *pi*, ali se srušio prije nego što je poslao procesu *pj*. Isto tako, bilo koji proces koji je slao *v* u prethodnoj rundi mora da se srušio da bi se objasnilo zašto *pk* posjeduje vrijednost *v*, a *pj* je nije primio. Nastavljajući tako trebamo pretpostaviti bar jedno rušenje procesa po svakoj prethodnoj rundi. Međutim pretpostavili smo najviše *f* rušenja, a imamo *f+1* rundi. Došli smo do kontradikcije.

Pokazano je da bilo koji algoritam koji želi postići konsenzus unatoč do *f* rušenja procesa, zahtijeva *f+1* rundi izmjena poruka bez obzira kako je konstruiran. (Dolev, Strong 1983). To se da primijeniti i za slučaj bizantinskih pogrešaka (Fischer i Lynch, 1982).

4.3 Problem Bizantinskih generala u sinkronom sustavu

Za razliku od algoritma za određivanje konsenzusa opisanog u prethodnom poglavlju, ovdje pretpostavljamo da proces može pokazivati proizvoljno pogrešan rad. Tj. proces može poslati bilo koju poruku s bilo kojim sadržajem u bilo koje vrijeme ili ne poslati nikakvu poruku.

Ukupno do *f* od *N* procesa može biti neispravno. Ispravni procesi mogu detektirati odsustvo poruke kroz timeout, ali ne mogu pretpostaviti da je zbog toga neki proces pao jer možda je bio šutljiv neko vrijeme, a onda ponovo počne slati poruke.

Pretpostavljamo privatnost komunikacijskih kanala među parom procesa (jedan proces ne može nadgledati poruke između druga dva procesa, te neispravni proces ne može injektirati poruke u komunikacijski kanal između duga dva procesa).

Lamport et al [1982] razmatrali su slučaj tri procesa koji šalje nepotpisane poruke jedan drugome. Pokazali su da ne postoji rješenje za problem bizantinskih generala ako je dozvoljeno da ijedan proces nekorektno radi. Generalizacijom rezultata pokazali su da ne postoji rješenje ako je $N \leq 3f$. Dalje su dali algoritam koji rješava problem bizantinskih generala u sustavu ako je $N > 3f+1$, za nepotpisane poruke.

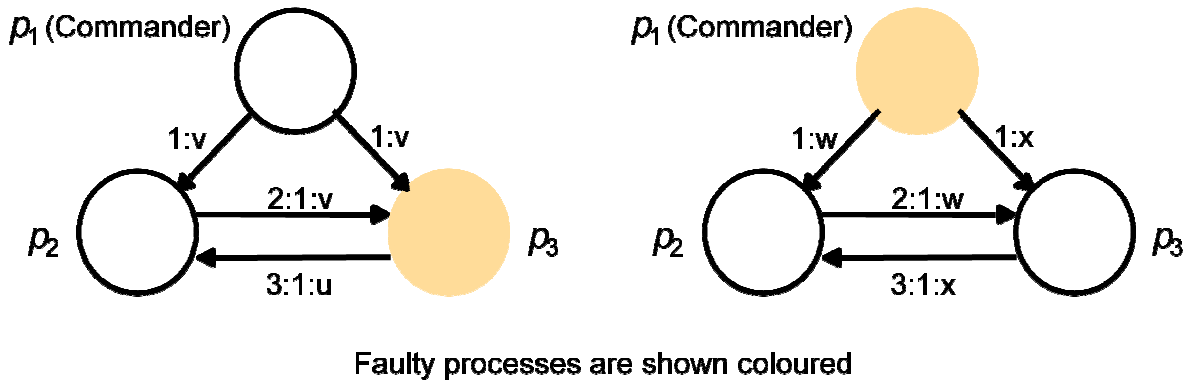
Nemogućnost s tri procesa ◊ Slika 4.3 prikazuje dva scenarija u kojima je tek jedan od tri procesa neispravan. U lijevom je jedan od podređenih generala je neispravan (p3), a u desnom je zapovjednik neispravan (p1).

Svaki scenarij pokazuje dvije runde poruka; poruke koju šalje zapovjednik i poruke koje zamjenici šalju jedan drugome. Npr oznaka 3:1:v se čita kao 3 kaže da 1 kaže *v*.

U lijevom scenariju zapovjednik p1 šalje korektne poruke v svakome od zamjenika p2,p3. p2 korektno to prosljeđuje p2, ali p3 šalje p2 vrijednost u≠v. Sve što p2 zna u tom trenutku je da je primio različite poruke, tj. ne zna koju je primio od zapovjednika.

U desnom scenariju zapovjednik je neispravan i šalje različite vrijednosti zamjenicima. Nakon što je p3 korektno proslijedio poruku x koju je primio , p2 se nalazi opet u istoj situaciji kao u prethodnom scenariju.

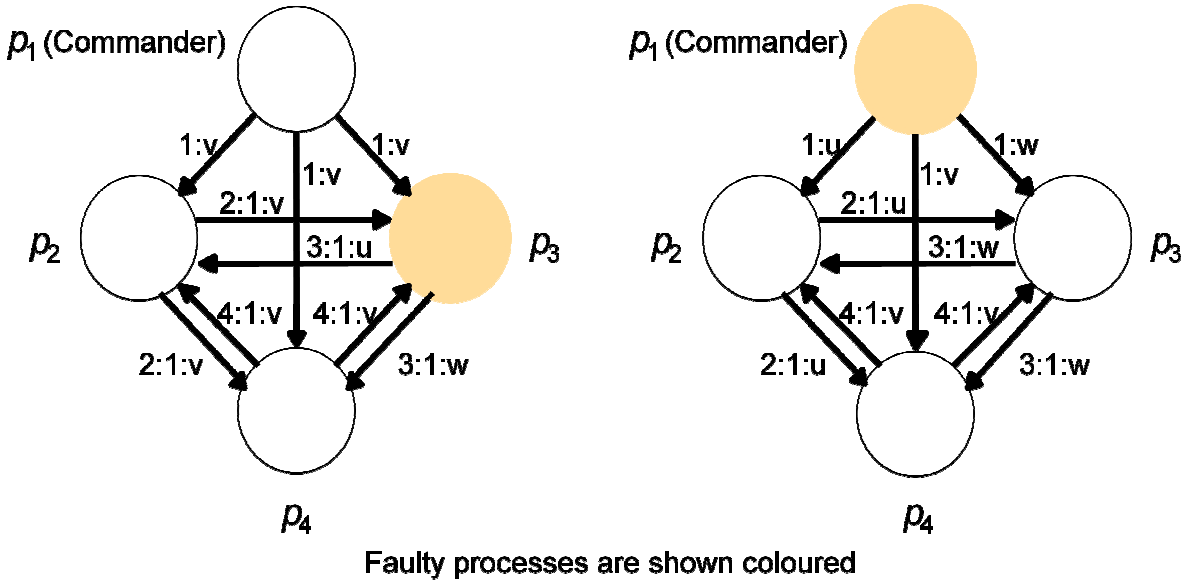
Nema načina da p2 zaključi tko je neispravan; da li je u pitanju p1 ili p3 ?



Slika 4.3 Nemogućnost s tri procesa

Rješenje za N=4, f=1 ◊ Ispravni generali postići će sporazum u dvije runde poruka.

- u prvoj rundi zapovjednik šalje poruke zamjenicima.
- u drugoj rundi zamjenici šalju poruke procesima istog ranga



Slika 4.4 Korektan rad algoritma za N=4, f=1

Svaki zamjenik prima vrijednost od zapovjednika, plus N-2 poruke od procesa istog ranga.

Može se pokazati da u ovom slučaju (i svim drugim s N>3f+1) da zamjenici mogu primjenom jednostavne funkcije većine postići sporazum.

U lijevom slučaju dva ispravna procesa odlučuju se za v na osnovu:

p2 - većina (v,u,v) = v

p3 - većina (v,v,w) = v

U desnom slučaju (neispravan zapovjednik)

p2 - većina (u,v,w) = ⊥

p3 - većina (u,v,w) = ⊥

p4 - većina (u,v,w) = ⊥

Ako proces ne primi poruku u nekom zadanom vremenu (timeout) pretpostavlja da je primio poruku ⊥.

Diskusija ◊ U diskusiji treba razmotriti efikasnost rješenja problema bizantinskih generala ili bilo kojeg drugog problema vezanog za sporazum odgovorima na slijedeća pitanja:

- Koliko rundi slanja poruka je potrebno za okončanje algoritma
- Koliko se poruka šalje i koje veličine

Fisher i Lynch [1982] su dokazali da bilo koji algoritam konsenzusa za nepotpisane poruke treba najmanje f+1 rundu . U svakoj rundi proces šalje ostalima podskup poruka koje je primio u prethodnoj rundi. Algoritam je veoma skup: zahtijeva slanje $O(N^{f+1})$ poruka. Efikasnost se da povećati jedino sa smanjenjem kompleksnosti poruka.

Postoje algoritmi (npr. Dolev i Strong [1983]) koji koriste digitalno potpisane poruke i koji opet završavaju u f+1 rundu, ali broj poslanih poruka je samo $O(N^2)$.

Veliki trošak navedenih algoritama dovodi do slijedećih praktičnih uputa u primjeni:

- Ako je izvor prijetnje neispravni hardver onda je povoljna činjenica što hardver rijetko otkazuje na proizvoljan način pa je moguće koristiti algoritam koji će se osloniti na poznavanje modela otkazivanja
- Ako su izvor prijetnje korisnici onda je svako rješenje bez digitalnog potpisa nepraktično

4.3.1 Nemogućnost u asinkronom sustavu

Prethodno navedeni algoritmi podrazumijevaju sinkroni sustav:

- izmjena poruka ide u rundama
- timeout za slanje poruka, tj. pretpostavka da je proces neispravan ako ne pošalje poruku u predviđenom vremenu

Fischer i ostali [1985] pokazali su da nijedan algoritam ne može garantirati postizanje konsenzusa u asinkronom sustavu. Dokaz je složen, ali svodi se na : zbog toga što procesu u asinkronom sustavu je dozvoljen odgovor u bilo koje vrijeme nije moguće razlikovati spori proces od onoga koji se srušio.

Međutim iako se ne može *garantirati* , konsenzus se određenom vjerojatnošću može postići i u asinkronom sustavu.

Postoje razne tehnike kojima se može pokušati postići konsenzus u asinkronom sustavu i ovdje su navedene dvije.

Maskiranje pogrešaka ◊ Možemo maskirati rušenje bilo kojeg procesa. Npr. transakcijski sustavi koriste perzistentnu memoriju za zapis stanja procesa. Ako proces se sruši, on se može ponovo pokrenuti na način da sa stanovišta drugih procesa se radi samo o usporenju.

Konsenzus korištenjem detektora kvara ◊ Procesi se mogu složiti da će držati neispravnim proces koji ne odgovori u nekom zadanom vremenu. Taj proces možda i nije neispravan, ali svi ostali procesi odbacuju njegove daljnje poruke. Efektivno smo asinkroni sustav pretvorili u sinkroni. Međutim oslanjamo se na pouzdanost detektora kvara.

Da bismo imali pouzdan detektor moramo često koristiti duge timeout vrijednosti čime smanjujemo efikasnost sustava. Jedno od rješenja je voditi statistiku vremena odgovora u sustavu i prema tome podešavati timeout vrijednosti.

5. KONKURENTNO PROGRAMIRANJE: PROCESI I NITI, SINKRONIZACIJA, UPRAVLJANJE NITIMA

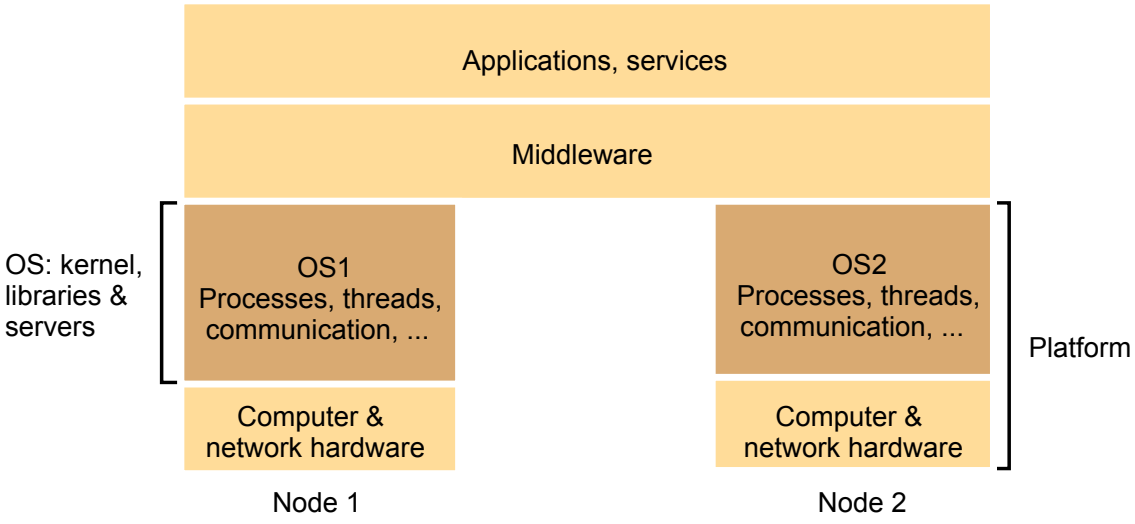
Današnji operacijski sustavi koji se koriste za realizaciju distribuiranih sustava su mrežni operacijski sustavi (Windows, razne varijante UNIX sustava,...) . To znači da imaju ugrađene funkcije za rad na mreži i da mogu pristupati različitim resursima u mreži. Mrežni pristup je najčešće, ali ne i uvijek mrežno-transparentan.

Mrežni operacijski sustavi imaju jednu vrlo bitnu karakteristiku. Oni kao čvorovi u distribuiranom sustavu zadržavaju autonomiju u upravljanju svojim resursima procesiranja. dakle svaki čvor je jedna zasebna instanca operacijskog sustava. takav operacijski sustav upravlja vlastitim procesima, ali ne određuje raspored (scheduling) procesa na drugim čvorovima.

Nasuprot prethodnome, mogli bismo zamisliti takav sustav koji bi imao kontrolu nad svim čvorovima sustava i koji bi pokretao i raspoređivao procese po svim čvorovima sustava. Takav sustav bi mogli nazvati distribuiranim operacijskim sustavom. (Tanenbaum i van Renesse, 1985).

Međutim još nema takvih sustava u široj upotrebi. Korisnicima odgovara određeni stupanj autonomije, a problem je i raznolikost platformi. Međutim pojava efikasnih virtualizacija računala vodi nas prema mogućim rješenjima u pogledu distribuiranih operacijskih sustava. Posebno bitna značajka virtualizacije je mogućnost pokretanja istovrsnih instanci operacijskih sustava na heterogenoj podlozi različitog sklopovlja na način da svaki čvor može pokrenuti istovremeno različite OS.

U ovom poglavlju obradit ćemo osnovne elemente aktivnosti OS-a tj. procese i niti te prikazati njihovu ulogu u ostvarenju aktivnosti distribuiranih sustava. Slika 5.1 prikazuje slojeve distribuirane arhitekture u kojima sastavne elemente čine mrežni operacijski sustavi na platformama čvorova.



Slika 5.1 Slojevi distribuirane infrastrukture

5.1 Procesi i niti

Tradicionalni pojam procesa iz 80-tih bio je vezan za izvršenje jedne aktivnosti i ubrzo je postao nedostatan bilo sa strane zadataka vezanih za distribuirane sustave bilo za složene aplikacije koje zahtijevaju internu konkurentnost.

Danas, pojam procesa podrazumijeva višestruku aktivnost. Proces je prema tome izvršna okolina u kojoj se izvršava jedna ili više niti (threads). Nit je apstrakcija izvršavanja na nivou operacijskog sustava. Izvršna okolina procesa se primarno sastoji od:

- adresnog prostora
- resursa za sinkronizaciju niti (semafori) i komunikaciju (utičnice-sockets)
- resursa više razine: otvorene datoteke, prozori,...

Izvršnu okolinu procesa skupo je kreirati i održavati, ali više niti može dijeliti jednu okolinu – tj. može dijeliti zajedničke resurse okoline.

Glavni razlog korištenja višestrukih niti je maksimiziranje stupnja konkurentnog izvršavanja među operacijama, tj. omogućavanje konkurentnog izvršavanja procesorskih operacija i U/I operacija, te omogućavanje korištenja konkurentnog procesiranja na višeprocesorskim sustavima.

Izvršna okolina pretpostavljeno štiti niti od utjecaja drugih niti van iste okoline tj. da niti ne mogu pristupati resursima drugog procesa. Međutim, pojedini OS omogućavaju kontrolirani među-procesni pristup.

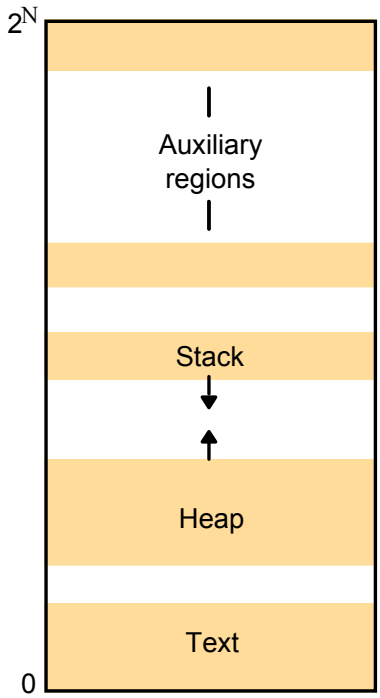
5.1.1 Adresni prostor

Adresni prostor je jedinica za upravljanje virtualnom memorijom procesa. To je veliki prostor od 2^{32} ili 2^{64} bajtova, koji je podijeljen u više područja (regions), koji su međusobno odijeljeni nepristupačnim dijelovima virtualne memorije. Područja se ne preklapaju.

Svako područje je specificirano sa slijedećim svojstvima:

- opseg – najniža virtualna adresa i veličina
- dozvole pisanja/čitanja/izvršavanja za niti procesa
- da li može rasti prema dolje ili prema gore

Za razliku od standardnog poimanja UNIX adresnog prostora koji ima tri područja (fixed, heap, stack), koristimo reprezentaciju adresnog prostora kao neodređenog skupa nepreklapajućih područja.



Slika 5.2 Adresni prostor

To je motivirano nizom faktora. Prvi je potreba da svaka nit ima svoj odvojeni stog (stack). Ako se svakoj niti da odvojeno područje onda je lako detektirati prekoračenje dozvoljenog podrčja detektiranjem pristupa nealociranoj virtualnoj memoriji. Drugo je potreba za mapiranjem datoteka u adresni prostor (omogućava pristup datoteci poput bajtova u memoriji). Slijedeći faktor je da se omoguće dijeljena memorijska područja između više različitih procesa ili između procesa i kernela. Procesi tada pristupaju identičnim memorijskim lokacijama za dijeljena područja, a ostala područja ostaju zaštićena.

Upotreba dijeljenih područja uključuje slijedeće namjene:

Biblioteke (libraries): Kod biblioteka može biti vrlo velik i bilo bi ga nerazumno učitavati za svaki proces koji ga koristi. Umjesto toga samo jedna kopija koda se učitava u fizičku memoriju, a onda koristi preko dijeljenih područja.

Kernel: najčešće su kod kernela i njegovi podaci mapirani u svaki adresni prostor na isti način. Stoga nije potrebno mijenjati mapiranje adresnog prostora prilikom sistemskih poziva.

Dijeljenja podataka i komunikacija: Dva procesa ili proces i kernel mogu dijeliti podatke prilikom izvršavanja nekog zadatka. To može biti daleko efikasnije upotrebom dijeljenog područja nego izmjenom poruka.

5.1.2 Kreiranje novog procesa

Kreiranje procesa je obično nedjeljiva operacija operacijskog sustava. Poznato nam je djelovanje Unix naredbi fork i exec. Za distribuirane sustave postupak kreiranja procesa treba uzeti u obzir neki kriterij pokojemu se odabire lokacija kreiranja novog procesa (npr. opterećenje). Stoga kreiranje dijelimo u dva neovisna aspekta:

- odabir ciljnog domaćina (host)
- Stvaranje izvršne okoline (i inicijalne niti izvršavanja)

Odabir domaćina procesa ◊ Politika odabira kreće se od stalnog pokretanja procesa na čvoru na kojem je zadano pokretanje do dijeljenja procesnog opterećenja između niza računala (load-sharing). Eager i ostali [1986] razlikuju slijedeće politike odabira,

Politika transfera određuje da li će se proces izvršavati lokalno ili udaljeno. To može npr. ovisiti trenutnom opterećenju čvora.

Politika lokacije određuje tko će biti domaćin za proces koji je određen za transfer. To može biti određeno prema opterećenju čvora, prema arhitekturi ili prema resursima koje posjeduju.

Politike lokacije mogu biti statičke ili adaptivne. Prve djeluju bez obzira na promjene stanja sustava iako mogu biti kreirane prema očekivanom dugoročnom ponašanju sustava. Mogu biti deterministične (čvor A uvijek transferira proces u čvor B) ili probabilističke (čvor A transferira proces na čvorove B-E prema nekoj statističkoj razdiobi).

Adaptivne politike uzimaju u obzir trenutne vrijednosti promjenjivih veličina (opterećenje čvorova, mreže) i odluke donose na osnovu heurističkih pravila.

Sustavi s dijeljenjem opterećenja mogu biti centralizirani, hijerarhijski ili decentralizirani. U prvom slučaju imamo komponentu sustava koju nazivamo upravitelj opterećenja (load manager) , u drugoj ih imamo više organiziranih u hijerarhiju. U decentraliziranom sustavu pojedini čvorovi razmjenjuju informacije i na osnovu toga donose odluke o transferu procesa.

Sustavi s dijeljenjem opterećenja mogu imati i mogućnost migracije procesa tj. transfera procesa u izvršavanju s jednog čvora na drugi. Ovaj pristup ima vrlo tešku praktičnu realizaciju.

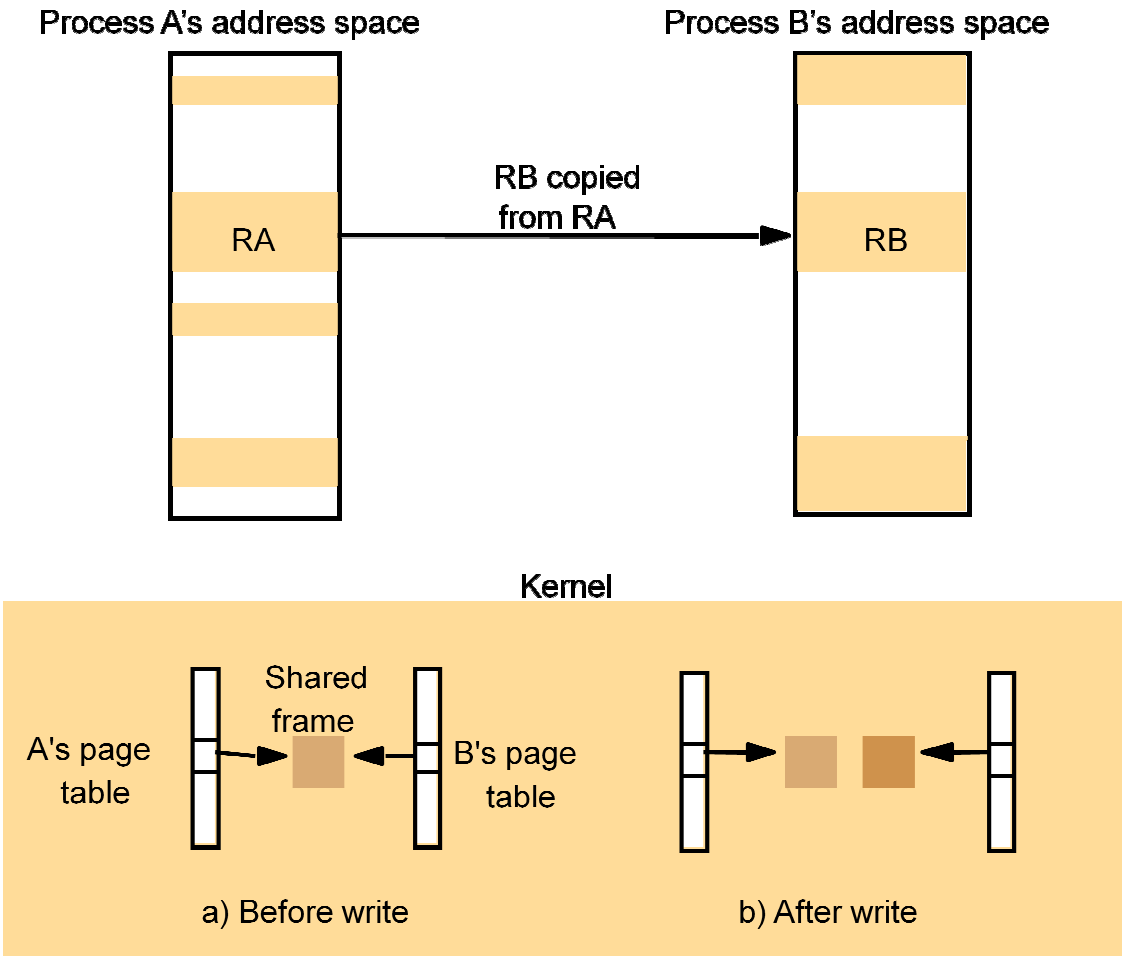
Eager i ostali [1986] proučavali su razne pristupe u konstrukciji sustava da dijeljenje resursa i došli su do zaključka da je jednostavnost jedno od najbitnijih svojstava bilo koje sheme dijeljenja resursa. Razlog je da prikupljanje informacija o detaljnom stanju sustava uvodi troškove koji najčešće premašuju moguće dobiti.

Kreiranje nove izvršne okoline ◊ Jednom kad je odabran računalo koje će biti domaćin procesa potrebno je kreirati izvršnu okolinu procesa.

Postoje dva načina u kreiranju adresnog prostora novokreiranog procesa. Prvi je da se adresni prostor kreira iz statički definiranog formata definiranog u izvršnoj datoteci.

Alternativno, adresni prostor može se kreirati uzevši u obzir tekuću izvršnu okolinu. Npr. u semantici UNIX fork naredbe, novokreirani proces fizički dijeli text područje roditelja, a ima heap i stack područja koja su kopije roditeljskog. Ova shema je uopćena na način da svako područje roditeljskog procesa može biti naslijeđeno ili izostavljeno iz procesa djeteta. naslijeđeno područje može biti dijeljeno ili kopirano iz roditeljskog područja.

Interesantna je optimizacijska shema copy-on-write koju primjenjuju neki OS. Područje koje se treba kopirati sastoji se od okvira koji odgovaraju stranicama virtualne memorije računala. Kad se područje početno kopira ne dolazi do fizičke kopije. Fizička kopija pojedine stranice se izvršava tek kad neki od procesa pokuša mijenjati sadržaj stranice. tad on dobiva svoju vlastitu kopiju. Slika 5.3 ilustrira navedenu tehniku.



Slika 5.3 Copy-on-write

5.2 Niti

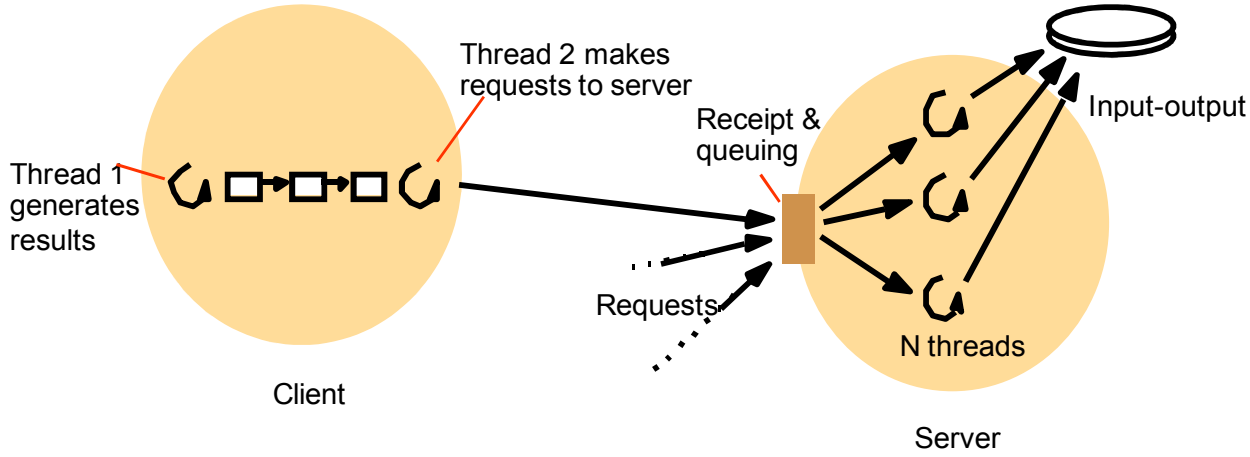
Ovo poglavlje ispituje koje su to prednosti upotrebe višenitnosti za klijent i server procese. Podrazumijevamo da su nam poznate prednosti kreiranja dodatnih niti u odnosu na dodatne procese tj. zašto iste zadatke ne realiziramo s više procesa umjesto s više niti.

Slika 5.4 prikazuje klijent i server s nitima. Uzmimo u razmatranje server. Server posjeduje pul (pool) s jednom ili više niti od kojih svaka ponovljeno uzima iz reda zahtjeve te ih procesira. Trenutno se ne zamaramo prijemom zahtjeva i njihovim postavljanjem u red. Također pretpostavljamo da svaka nit primjenjuje istu proceduru u obradi zahtjeva. Npr. uzmimo da svaki zahtjev treba 2ms procesiranja i 8ms I/O kašnjenja za čitanje s diska. Pretpostavljamo i jednoprocesorsko računalo servera.

Slijedi analiza propusnosti mjerena u broju obrađenih zahtjeva klijenta po sekundi.

Ako jedna nit obrađuje sve zahtjeve onda je prosječna brzina obrade jednog zahtjeva 8+2=10ms, tj 100 zahtjeva u sekundi.

Ako imamo dvije niti i to takve da se jedna nit može izvršavati dok druga je blokirana zbog I/O operacije, tada očekujemo povećanje propusnosti. Međutim u našem slučaju brzina je ograničena diskom i tada je za slučaj slijednog izvođenja disk operacija propusnost jednaka 1000/8=125 zahtjeva u sekundi.



Slika 5.4 Klijent i server s nitima

Pretpostavimo sada uvođenje međumemoriju za blokove diska. Server čuva podatke koje čita u spremnicima u adresnom prostoru. Nit koja čita podatke prvo podatke traži u međumemoriji, a ako ih ne nađe tamo onda traži na disku. Ako pretpostavimo 75% pogodaka u međumemoriji (hit rate) onda se srednje vrijeme pristupa podacima reducira na (0.75*0+0.25*8)=2ms, tj. maksimalna teoretska propusnost se povećava na 500 zahtjeva u sekundi. Međutim zbog operacija s međumemorijom može doći do podizanja prosječnog procesorskog vremena na npr. 2.5 ms po zahtjevu pa procesor postaje usko grlo tj. propusnost je 400 zahtjeva u sekundi.

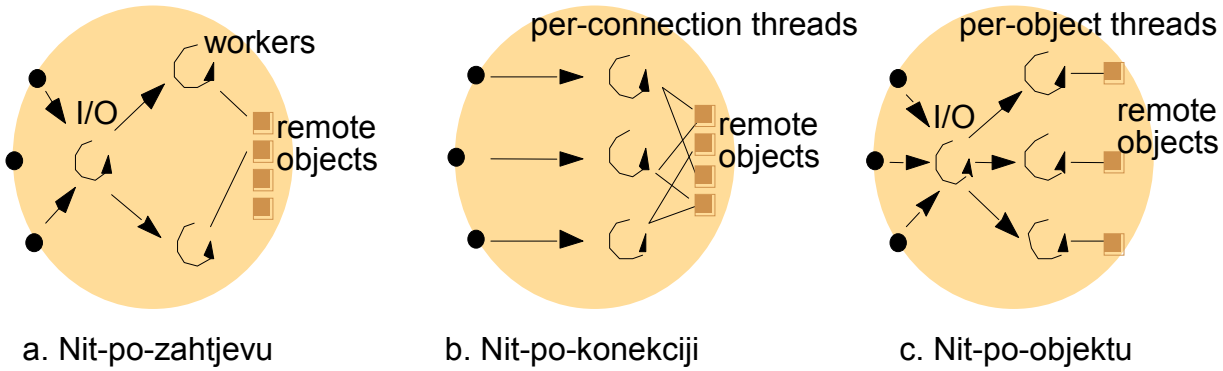
Propusnost se može povećati uvođenjem multiprocesora s dijeljenom memorijom. Tada za dvije niti koje procesiraju zahtjeve + jedna za U/I operacije propusnost raste na 444 zahtjeva u sekundi, a za 3 i više niti koje procesiraju zahtjeve propusnost je 500 zahtjeva u sekundi (sve uz 2.5ms procesorskog vremena po obradi zahtjeva). Pokušajte to izračunati !

Arhitekture za višenitne servere ◊ Za opis različitih načina mapiranja zahtjeva na niti koristit ćemo terminologiju koje je koristio Schmidt [1998] u opisu višenitnih arhitektura CORBA Object Request Brokera (ORB). ORB procesira zahtjeve koji stižu preko skupa vezanih socketa.

Slika 5.4 pokazuje primjer jedne moguće višenitne arhitekture, tzv. *worker pool* arhitekture. U njenoj najjednostavnijoj formi server kreira fiksan pul worker niti koje procesiraju zahtjeve. Modul prijem i red sa prethodne slike implementiran je s jednom 'U/I' niti koja zaprima zahtjeve i postavlja ih na dijeljeni red zahtjeva gdje su spremni za preuizimanje od strane worker niti.

Ako zahtjeve moramo razvrstavati po prioritetu onda možemo uvesti višestruke redove čekanja s različitim prioritetima. Worker niti tada pregledavaju redove po njihovom padajućem prioritetu.

Nedostatak predstavljene arhitekture je da fiksan broj worker niti ponekad može biti nedostatan za povećanom frekvencijom pristizanja zahtjeva. Osim toga može doći do visoke učestalosti preklapanja između U/I niti i worker niti u pristupu dijeljenom redu čekanja.



Slika 5.5 Alternativne višenitne server arhitekture

Slika 5.5a prikazuje arhitekturu nit-po-zahtjevu koja pokreće dodatne worker niti po zahtjevu, a na kraju procesiranja worker nit uništava samu sebe. Prednost ove arhitekture (i slijedeće dvije) je da se niti ne bore za pristup dijeljenom redu čekanja, a i propusnost se potencijalno može povećati jer se može stvoriti nit za svaki od zahtjeva. Nedostatak je u dodatnom opterećenju zbog kreiranja i uništavanja niti.

Slika 5.5b prikazuje arhitekturu *nit-po-konekciji*. U ovoj arhitekturi se svakoj konekciji dodijeljuje jedna nit. Server kreira novu worker nit kada klijent napravi konekciju, a uništava je pri zatvaranju konekcije. U međuvremenu klijent može preko konekcije zahtjevati pristup jednom ili više udaljenih objekata.

Slika 5.5c prikazuje arhitekturu *nit-po-objektu* gdje se nit asocira sa svakim udaljenim objektom. Jedna U/I nit prima zahtjeve i stavlja ih u red čekanja (na nivou objekta) za worker niti.

Ove zadnje dvije arhitekture imaju prednost zbog smanjenih zahtjeva što se tiče upravljanja nitima (kreiranje/uništavanje). Nedostatak je da klijenti mogu čekati jer se događa da jedna nit ima više čekajućih zahtjeva dok druga nije uposlena.

Niti unutar klijenta ◊ Niti mogu biti korisne i za klijente. Slika 5.4 pokazuje klijent proces sa dvije niti. Prva nit priprema podatke koje treba uputiti serveru prilikom poziva udaljene metode, ali ne zahtjeva rezultate. Klijent proces koristi drugu nit koja poziva udaljenu metodu što istu blokira, dok prva nit može raditi na pripremi podataka za drugi poziv. Prva nit podatke postavlja u spremnik kojeg prazni druga nit. Prva nit je blokirana samo ako je spremnik prepunjen. (primjer višenitnog klijenta je Web preglednik).

5.3 Programiranje s nitima

Programiranje s nitima se tradicionalno proučava u okviru predmeta vezanih za operacijske sustave. Pojmovi koji se vežu uz ovo polje su: stanje natjecanja (race condition), kritična sekcija, monitor, uvjetna varijabla i semafor.

Najšire prihvaćeni standard za rad s nitima u C/C++ jezicima je POSIX standard IEEE 1003.1c-1995 poznat kao *pthreads*.

Neki jezici, a među njima i Java imaju direktnu podršku za rad s nitima. Java posjeduje metode za stvaranje, uništavanje i sinkronizaciju niti.

Slijede Java konstruktor i metode za upravljanje nitima:

Thread(ThreadGroup group, Runnable target, String name)

Kreira novu nit u SUSPENDED stanju, koja pripada grupi *group*; identificirana je s nazivom *name*;izvršava *run()* metodu objekta *target*.

setPriority(int newPriority), getPriority()

Postavlja i vraća prioritet niti

run()

Nit izvršava *run()* metodu ciljnog objekta ako isti ima takvu; inače izvršava vlastitu run() metodu, (Thread implementira Runnable).

start()

Mijenja stanje niti iz SUSPENDED u RUNNABLE.

sleep(int millisecs)

Uzrokuje da nit ulazi u SUSPENDED stanje u trajanju specificiranog vremena.

yield()

Ulazi u READY stanje i poziva scheduler.

destroy()

Uništava nit.

Metode za sinkronizaciju su:

thread.join(int millisecs)

Blokira pozivnu nit za maksimalno do specificiranog vremena i čeka na uništenje niti *thread*.

thread.interrupt()

Prekida nit *thread ()*: šalje signal prekida koji uzrokuje preikadanje niti u nekom od metoda blokiranja, kao što je npr. sleep.

object.wait(long millisecs, int nanosecs)

Blokira pozivnu nit sve do poziva notify() ili notifyAll() ne probudi nit ili je nit prekinuta ili je specificirano vrijeme prošlo.

object.notify(), object.notifyAll()

Budi jednu ili sve niti koje su za *object* pozvale wait().

Životni ciklus niti ◊ Nova nit se kreira na podlozi Java virtualnog stroja (JVM) kao i nit koja daje nalog za kreiranje. Početno je u SUSPENDED stanju. S pozivom metode start() stavlja se u RUNNABLE stanje, i počinje s izvršavanje run() metode objekta određenog parametrom konstruktora. Nitima može biti dodijeljen prioritet ako to podržava JVM implementacija za

određenu platformu. Nit završava povratkom iz run() metode ili kad se pozove njena destroy() metoda.

Programi mogu upravljati nitima organiziranima u grupe. Niti jedne grupe ne mo gu upravljati nitima druge grupe. Npr. grupa jedne aplikacije ne može prekidati niti vezane za upravljanje sustavom prozora (AWT niti).

Sinkronizacija niti ◊ Programiranje višenitnog procesa zahtjeva veliku pažnju. Glavne poteškoće su u dijeljenju objekata i tehnike koje se koriste za kordinaciju i suradnju niti. Primjer je pristup porukama u redu redu čekanja iz prethodnih primjera. Prethodno nabrojane metode prikazuju dio mogućnosti jave u koordinaciji niti.

Upravljanje nitima ◊ Najbitnije razlikovanje u upravljanju nitima je između preemptivnog i ne-preemptivnog rasporeda izvođenja niti. U preemptivnom rasporedu izvođenja nit može biti suspendirana u bilo kojoj točki kako bi prepustila izvođenje drugoj niti. U ne-preemptivnom načinu nit se izvodi sve dok ne napravi poziv prema sustavu za upravljanje nitima. Tada je sustav može prekinuti i dati mogućnost izvođenja drugim nitima.

Prednost ne-preemptivnog izvođenja je da je svaka sekcija koja ne sadrži sistemski poziv prema suatavu za upravljanje nitima, ujedno i kritična sekcija. Na taj način se jednostavno izbjegavaju situacije natjecanja niti. S druge strane ne-preemtivne niti ne mogu u potpunosti iskoristiti multiprocesore, pošto se izvršavaju u eksluzivnom modu. Mora se paziti na duže sekcije koda koje ne sadrže pozive sustavu za upravljanje nitima.Programer mora ubacivati specijalni yield() poziv čiji je jedini zadatak da se drugim nitima da vremena za izvođenje. Sve ovo vodi da je potebno razmotriti kako pojedini sustavi implementiraju podršku za upravljanje nitima.

Implementacija unutar OS ◊ Veliki broj jezgri operacijskih sustava podržava rad s višenitnim procesima, npr. Windows,Linux,Solaris,Mach i Chorus. Upravljanje nitima se u takvim sustavima postiže sistemskim pozivima jezgre (kernel threads).

Neki drugi OS podržavaju jednonitne procese. Ipak, upotrebom biblioteka funkcija koje se povežu s aplikacijom moguće je kreirati višenitne aplikacije (user threads) . Nedostaci ovakvog pristupa su:

- Niti unutar procesa ne mogu koristiti prednosti multiprocesora
- Nit koja pristupa blokirajućim funkcijama jezgre OS, blokira sve ostale niti
- Kernel ne može upravljati s rasporedom izvođenja pojedinačnih niti.

Međutim postoje i značajne prednosti korištenja biblioteka funkcija za upravljanje s nitima (user threads) kao što su:

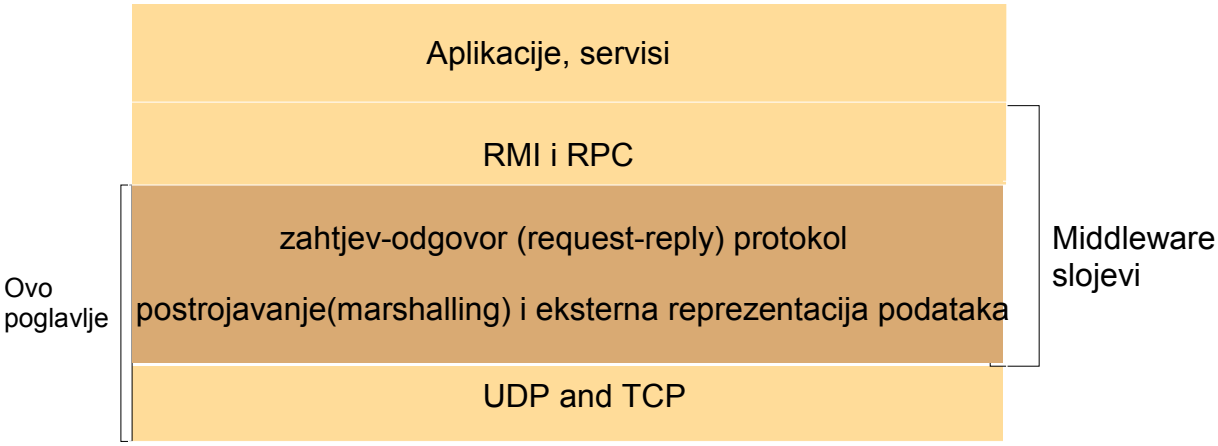
- Neke operacije s nitima imaju manji trošak izvođenja. Npr. prebacivanje između niti procesa tada ne zahtijeva sistemski poziv jezgri OS.
- Kako je modul za upravljanje rasporedom izvođenja niti realiziran izvan jezgre OS, tada ga je moguće prilagoditi prema tipu aplikacije.
- Moguće je postići podršku za veći broj niti nego što to je prihvatljivo za jezgru OS.

Neki OS podržavaju oba načina korištenja niti (Solaris). Tada se oba sustava koriste na hijerarhijski način pri čemu su moguće različite implementacije.

6. INTERPROCESNA KOMUNIKACIJA I MIDDLEWARE

Chapters 4, 5, & 20 of Distributed Systems: Concepts and Design, Fourth Edition, G. Coulouris, J. Dollimore and T. Kindberg, Addison-Wesley 2005.

Ovo poglavlje bavi softverskim slojem koji pripada dijelom u middleware, a dijelom transportnim protokolima.



6.1 Softverski slojevi

Sloj iznad bavi se integracijom komunikacije u paradigmu programskog jezika. Sloj ispod definira transportne protokole Interneta UDP i TCP, bez obzira na njihov način korištenja unutar middlewarea ili aplikacija.

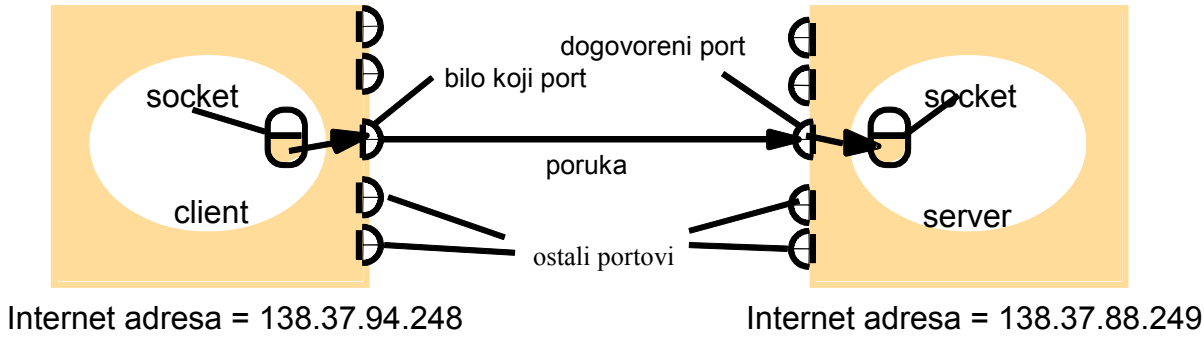
Aplikacijsko programsko sučelje (API) za UDP pruža nam apstrakciju u vidu *slanja poruka (message passing)* – najjednostavniji način međuprocesne komunikacije. Paketi kojima se šalju poruke su neovisni i nazivaju se *datagrami*. Java i UNIX API koristi *socket* kao indirektnu referencu na određeni port na koji se šalje poruka.

API za TCP pruža nam apstrakciju dvosmjernog *toka (stream)* između para procesa. Informacija se prenosi bez obzira na granice poruka. Jedan proces proizvodi podatke, a drugi ih konzumira. Poslani podaci se spremaju u bufferu na strani konzumenta sve dok ih konzument ne prihvati. Proces proizvođač mora čekati ako je buffer konzumenta prepunjen.

Bitan zadatak sloja kojim se ovdje bavimo je kako objekte i strukture podataka koje koristimo u aplikacijama prevesti u formu pogodnu za slanje preko mreže, a da se uzme u obzir da različita računala mogu imati različite implementacije za zapis podataka. Poseban je problem i reprezentacija referenci na objekte u distribuiranom sustavu.

6.1 Sockets

Obe forme komunikacije (UDP i TCP) koriste apstrakciju *socket*.za označavanje krajnjih točaka komunikacije između procesa. Socketi potječu od BSD UNIX-a, a prisutni su u većini suvremenih OS poput raznih verzija UNIX-a (uključivši i Linux), kao i u Windowsima i Machintosh OS. Slika 6.2 ilustrira slanje poruke između dva socketa, jednog na strani pošiljaoca, a drugog na strani primatelja.



6.2 Socketi i portovi

Poruke poslane na određenu Internet adresu i port može primiti samo proces koji je vezan za taj port i računalo. Proces može isti port koristiti za slanje i primanje poruka. Svaki proces može koristiti više portova, ali procesi na istom računalu ne mogu dijeliti isti port (osim procesa koji koriste IP multicast). Svako računalo ima na raspolaganju veliki broj (2^{16}) portova. Svaki socket je asociran s jednim od protokola – ili UDP ili TCP.

.

6.2 UDP datagram komunikacija

Ovdje su predstavljene osnovne karakteristike komunikacije zasnovane na UDP protokolu. Datagram koji se šalje od procesa pošiljaoca ka procesu primatelju, šalje se bez potvrde (acknowledgement) i bez ponovnih pokušaja. Ako se dogodi pogreška u komunikacijskom kanalu, poruka možda neće stići. Pošiljatelj može slati s proizvoljno odabranog slobodnog porta na točno određen port primatelja (servera). *Receive* metoda primatelju osim poruke, daje i Internet adresu i port pošiljaoca , tako da to omogućava primatelju slanje odgovora.

Bitne su slijedeće karakteristike datagram komunikacije:

Veličina poruke: Proces primatelj specificira međuspremnik (buffer) točno određene veličine za prijem poruke. Ako je pristigla poruka prevelika ona se jednostavno krati. IP protokol koji je podloga ograničava dozvoljenu veličinu poruke na 2^{16} bajtova. Međutim većina okolina nameće strože ograničenje od 8kiB. Svaka aplikacija mora poruke veće od ograničenja podijeliti u više manjih dijelova.

Blokiranje: Socketi normalno rade na način da omogućavaju neblokirajuće *send* metode, a blokirajuće *receive* metode (neke implementacije imaju neblokirajući receive kao opciju). Pristigla poruka se sprema u red ako je socket vezan za neki proces, inače se odbacuje. Tu će je pokupiti već pozvana ili buduće pozvana receive metoda. Pozvana receive metoda blokira

sve dok se ne primi datagram ili ne istekne timeout ako je isti postavljen. Ako proces treba obavljati neki rad za vrijeme dok receive čeka, potrebno je koristiti dodatne niti.

Timeout: receive koji blokira prikladan je za upotrebu za server kada on čeka na poruku nekog klijenta. Međutim ponekad proces koji je trebao slati poruku biva srušen ili se poruka izgubi u komunikacijskom sloju. Zbog toga je kad očekujemo takve pojave moguće na receive postaviti timeout. Izbor prikladnog vremena timeouta je težak i najčešće se upotrebljava timeout vrijeme koje je značajno veće nego vrijeme potrebno da se poruka prenese u normalnim uvjetima.

Receive from any: receive metoda ne specificira izvor poruke. Međutim, moguće je izvršiti dediciranu konekciju s određenim portom na određenom računalu pri čemu socket može samo primati i slati poruke na tu adresu.

Model pogrešaka ◊ UDP komunikacija je podložna slijedećim tipovima pogrešaka:

Pogreške propusta : Poruke mogu biti odbačene bilo zbog greške kontrolne sume ili zbog nedostatka mjesta u nekom od buffera.

Pogreška redoslijeda: Poruke mogu biti dostavljene po redoslijedu koji nije redoslijed slanja.

Aplikacije koje koriste UDP protokol trebaju osigurati vlastite mehanizme detekcije i otklanjanja grešaka. Povrh UDP protokola mogu će kreirati pouzdane zahtjev-odgovor protokole korištenjem raznih tehnika komunikacije npr. potvrda (aknowledgements).

Upotreba UDP protokola◊ Za neke servise prihvatljiva je upotreba protokola koji je podložan povremenim pogreškama propusta (npr. DNS, VOIP-voice over IP). razlog korištenja je to što UDP protokol nema dodatnih opterećenja koja su inače vezana uz garantiranu isporuku poruka.

6.3 Komunikacija s TCP tokom

API TCP protokola koji potječe s BSD 4.x UNIX sustava, ostvaruje apstrakciju toka (stream) u koji možemo pisati i iz kojega možemo čitati podatke. Apstrakcija toka skriva slijedeće karakteristike mrežne komunikacije:

Veličina poruka: Aplikacija sama bira koliko će podataka upisati u tok ili čitati iz toka u pojedinoj operaciji. Implementirana TCP podloga sama odlučuje koliko će se podataka nakupiti dok ne budu poslani s jednim ili više IP paketa. Ako je potrebno, aplikacija može naložiti trenutno slanje podataka.

Izgubljene poruke: TCP protokol upotrebljava shemu s potvrdama primitka (acknowledgement). Koristi se složeni algoritam koji se zasniva na prepoznavanju nedostatka potvrde i nakon toga zahtjeva za retransmisijom.

Kontrola toka: TCP protokol nastoji upariti brzine čitanja i pisanja procesa na krajevima toka. To znači da može blokirati i slanje, ako proces koji čita ne stiže pročitati poslane podatke.

Dupliciranje ili određivanje redoslijeda poruka. Svaki paket ima identifikator poretka koji omogućava da primatelj odbaci duplicirane poruke ili služi u pravilan redoslijed poruke koje stižu u redoslijedu različitom od redoslijeda slanja.

Odredišta poruke: Par procesa treba uspostaviti konekciju prije nego što počnu slati poruke kroz tok. Uspostavljanje komunikacije uključuje *connect()* zahtjev od strane klijenta i *accept()* zahtjev od strane servera. Za vrlo kratke poruke to može biti značajno dodatno opterećenje.

Iako API ta komunikaciju upotrebom TCP toka zahtjeva prilikom komunikacije podjelu uloga na klijent/server proces, nakon uspostave procesi mogu biti u različitim ulogama (peer).

Serverska uloga uključuje kreiranje socketa vezanog za određeni port koji nadalje osluškuje zahtjeve klijenata. Obično se nakon *connect()* zathjeva klijenta , izvršava *accept()* sa strane servera i kreira se novi socket za komunikaciju s tokom, dok se na prvotnom socketu i dalje osluškuju zahtjevi za konekcijom od strane drugih klijenata.

Istaknimo i slijedeće značajke komunikacije upotrebom tokova:

Podudaranje formata pojedinih podataka: Dva procesa koji komuniciraju preko toka trebaju se složiti oko formata podataka koji se prenose kroz tok. Npr ako pošiljalac šalje *int* podataka kojeg slijedi *double* podatak, onda se moraju čitati po istom formatu. Najmanji propust može dovesti do krive interpretacije podataka.

Blokiranje: Ako u prijemnom spremniku nema dovoljno podataka proces koji čita ostat će blokiran. Isto tako proces koji piše može ostati blokiran ako su spremnici prepunjeni.

Niti: Kada server prihvati novu konekciju , uobičajeno je da kreira novu nit u kojoj komunicira s klijentom. Stoga, ako jedna nit i blokira ostale mogu koristiti resurse računala.

Model pogrešaka ◊ Da bi se zadovoljio zahtjev za integritetom podataka u komunikaciji, TCP tokovi koriste kontrolnu sumu za detektiranje i odbacivanje iskvarenih podataka, te slijedne (sekvence) brojeve za odbacivanje dupliciranih paketa. Da bi se osigurala valajnost prenesenih podataka TCP tokovi koriste timeout i retransmisiju. Stoga imamo garanciju da će poruka biti prenesena unatoč tome što neki od paketa može biti izgubljen.

Međutim u slučaju bilo kakvog poremećaja u mreži koji izazove gubljenje paketa u granicama u kojima TCP protokol ne može izvršiti korekciju , TCP će deklarirati prekid konekcije. Stoga TCP ne omogućava potpuno pouzdanu komunikaciju. Kada je konekcija prekinuta proces koji pokuša čitati ili pisati bit će upozoren. To ima slijedeće posljedice:

- proces koji koristi konekciju ne može zaključiti da li je prekid konekcije nastao zbog rušenja procesa na drugoj strani ili zbog problema u mreži
- proces koji je poslao poruku, nema potvrdu o prijemu skoro poslanih poruka

Upotreba TCP tokova ◊ Veliki broj često upotrebljavanih servisa koristi TCP tokove na rezerviranom broju porta. Npr. HTTP, FTP, SSH, Telnet, SMTP.

6.4 Eksterna prezentacija podataka i postrojavanje (marshalling)

Informacija pohranjena u pokrenutim programima nalazi se u formi struktura podataka – npr. kao skup povezanih objekata, dok se informacija u porukama nalazi u formi niza bajtova. Bez obzira na formu komunikacije, strukture podataka se moraju prije slanja konvertirati u niz bajtova. Podaci se sastoje od različitih tipova, a računala uključena u komunikaciju ne moraju ih imati implementirane na isti način. Npr:

- big-endian i little endian reprezentacija cijelih brojeva
- različiti formati zapisa brojeva u pokretnom zarezu
- način zapisivanja karaktera (8-bit, UNICODE, ...)

Dva su osnovna načina da se računala sporazume u izmjeni binarnih (ne-tekstualnih) tipova podataka:

- Eksterna prezentacija podataka: vrijednosti se konvertiraju u dogovoreni eksterni format prije same transmisije . Ako su dva računala istog tipa može se dogovoriti prijenos bez konverzije.
- Vrijednosti se šalju u formatu koji podržava pošiljatelj, zajedno s indikacijom korištenog formata. Primatelj konvertira podatke ako je to potrebno.

Postrojavanje (marshalling) je proces kojim se strukturirana kolekcija pojedinih podataka prevodi u formu pogodnu za transmisiju unutar poruka. Izdvajanje (unmarshalling) je suprotan postupak čiji je cilj na strani primatelja iz poruka proizvesti ekvivalentnu strukturu podataka onoj poslanoj od strane pošiljaoca.

Razmatramo tri alternativna pristupa:

- CORBA reprezentacija podataka. Odnosi se na eksternu reprezentaciju strukturiranih i primitivnih tipova podataka. Koristi se u raznim programskim jezicima i platformama.
- Java serijalizacija objekata. Odnosi se na transformaciju u niz bajtova jednog ili stabla objekata koji tako mogu biti prenošeni u porukama ili pohranjeni na disk. Koristi se samo u Javi.
- XML (Extensible Markup Language). Definira tekstualni format za reprezentaciju strukturiranih podataka. Naširje je zastupljen i koristi se u raznim programskim jezicima i platformama.

Prve dvije forme obavljaju se kroz odgovarajući middleware, a bez uključenja programera aplikacije. Čak i u slučaju XML koji se može ručno kodirati, za sve platforme i jezike postoji odgovarajući middleware.

Prve dvije forme koriste binarni zapis što u odnosu na XML koji koristi tekstualni zapis, daje kompaktniji sadržaj.

CORBA reprezentacija podataka sadržava samo prenesene vrijednosti, a bez informacije o tipu podataka. Java serijalizacija i XML prenose i informaciju o tipu podataka s tip da XML može se u prijenosu referirati na eksterni set podataka koji opisuju prenesene podatke.

6.4.1 CORBA Common Data Representation (CDR)

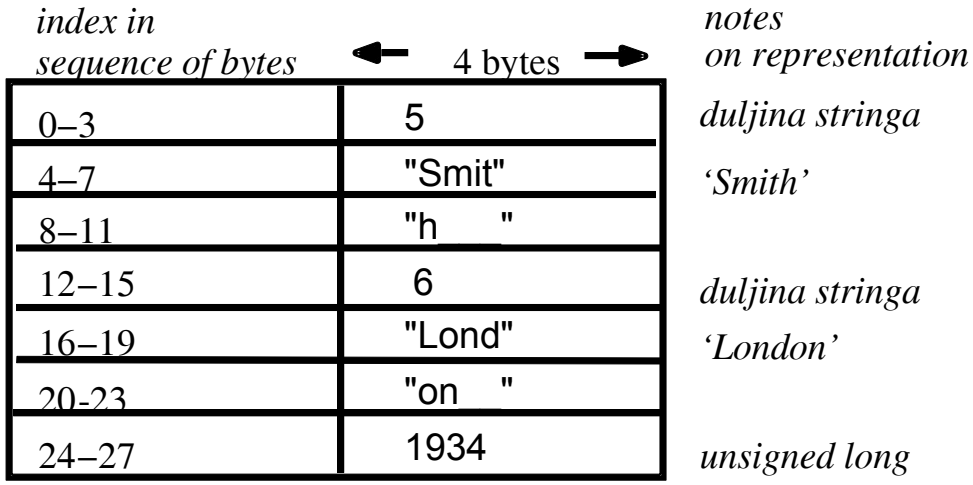
CORBA CDR je eksterna reprezentacija podataka definirana s CORBA 2.0 [OMG 2004a]. CDR reprezentira sve tipove podataka koji mogu biti korišteni kao argumenti ili povratne vrijednosti u CORBA udaljenim pozivima. Sastoji se od 15 primitivnih tipova podataka te niza kompozitnih tipova podataka.

Operacije postrojavanja/izdvajanja (marshalling/unmarshalling) se generiraju automatski iz specifikacije tipova podataka u formi CORBA IDL.

Npr. u CORBA IDL možemo opisati slijedeću strukturu podataka:

```
struct Person{
        string name;
        string place;
        unsigned long year;
}
```

To se prevodi u byteove kako je to pokazano na slici:



6.3 CORBA poredana reprezentacija podataka

6.4.2 Java serijalizacija objekata

U java RMI i objekti i primitivne vrijednosti mogu biti proslijeđeni kao argumenti i rezultati poziva metoda. Objekt je instanca klase. Java klasa ekvivalentna strukturi Person izgleda ovako:

```
public class Person implements serializable {
        private String name;
        private String place;
        private int year;
        public Person(String aName,String aPlace, String aYear){
                name=aName;
                place=aPlace;
                year=aYear;
        }
}
```

Gornja klasa implementira sučelje Serializable koje nema metoda. Navođenje da klasa implementira sučelje Serializable ima efekt da je dozvoljena serijalizacija instanci klase.

Pojam serializacija u Javi odnosi se na aktivnost prevođenja objekta ili vezanog niza objekata u formu (niz bajtova) pogodnu za prijenos u porukama (npr. kao argument RMI) ili za pohranu na disk. deserijalizacija je obnova sadržaja (stanja) objekata iz serijalizirane forme. Pri tome se pretpostavlja da proces koji obavlja deserijalizaciju nema prethodna saznanja o tipovima objekata koji se nalaze u serijaliziranoj formi. Stoga neka informacija o klasi objekata mora biti sadržana u serijaliziranoj formi.

Informacija o klasi sastoji se od :

- naziv klase
- broj verzije klase

Broj verzije klase se mijenja u slučaju značajnijih promjena u sadržaju klase. Verziju može promijeniti programer ili se može postaviti automatsko računanje kao hash naziva klase njenih varijabli instance, metoda i sučelja. Proces koji vrši deserijalizaciju tada može znati da li učitava korektnu verziju klase.

Java objekti mogu sadržavati reference na druge objekte. Kada se objekt serijalizira potrebno je serijalizirati i objekte vezane na reference. Kako više referenci može referirati na jedan objekt, sustav za serijalizaciju vodi računa i o tome.

Primitivni brojčani tipovi i boolean tip se zapisuju u binarnom formatu, a sadržaj Stringova i tipa character zapisuju se u UTF-8 formatu (Universal Transfer Format). UTF-8 format nam omogućava da se ASCII znakovi zapisuju jednim byteom, a Unicode karakteri s više bajtova.

Java klasa ObjectOutputStream posjeduje metodu writeObject koja kao argument može primiti bilo koju klasu koja implementira sučelje Serializable. (vidi i klasu ObjectInputStream i metodu readObject).

Serijalizacija i deserijalizacija argumenata u udaljenim pozivima (RMI) obavlja se automatski u middlewareu.

Prednosti Java serijalizacije su:

- jednostavnost implementacije
- kompaktan zapis u odnosu na tekstualne reprezentacije podataka (XML)

Nedostaci su:

- format upotrebljiv isključivo iz Java aplikacija
- nečitljiv iz vanjskih editora

6.4.3 Extensible markup language (XML)

XML je jezik s oznakama (markup) koji je kao i HTML izveden iz vrlo opsežnog SGML jezika (Standardized Generalized Markup Language).

XML je dizajniran s namjerom da se koristi za zapis strukturiranih dokumenata u web okružju. XML podaktovni članovi označeni su tekstualnim oznakama koje opisuju logičku

strukturu podataka kao i atribute samih podataka. Specifikacija XML jezika nije sadržaj ovih predavanja. Upotreba XML jezika je vrlo široka npr:

- prijenos podataka te definicija sučelja web servisa
- arhivska pohrana podataka aplikacija (arhive su veće od binarnih, ali su čitljive)
- konfiguracijske datoteke i definicija sučelja u operacijskim sustavima

Definicija Person struktura u XML jeziku bi izgledala ovako:

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

Jedna od najbitnijih osobina XML je dakle samoopisnost što omogućava upotrebu u istih poruka u različitim aplikacijama u različite svrhe. Aplikacija može isčitati samo one informacije koje su joj potrebne, preskačući informacije namijenjene za druge aplikacije.

Mogućnost da se XML čita/editira direktno u tekstualnom formatu u nekom tekst editoru vrlo je korisna kod testiranja programa i otklanjanja grešaka.

Tekstualni format XML-a izbjegava i nekompatibilnosti koje uvode različite platforme.

Najveći nedostatak, a to je povećanje veličine poruka, može se dijelom izbjeći korištenjem sažimanja koja su već standardno uključena u neke komunikacijske protokole (npr. HTTP 1.1)

6.4.4 Referenca udaljenog objekta

Ovaj dio odnosi se na jezike poput Jave ili CORBA middlewarea koji podržavaju model s distribuiranim objektima. Ne odnosi se na XML.

Kada klijent poziva udaljenu metodu on treba specificirati za koju instancu tj. objekt se ta metoda poziva. *Referenca udaljenog objekta* je identifikator objekta koji ima valjanu vrijednost unutar razmatranog distribuiranog sustava.

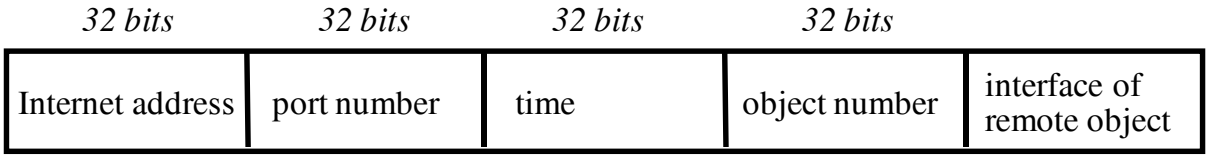
Stoga reference udaljenih objekata moraju biti generirane na način koji garantira njihovu prostornu i vremensku unikatnost. U svakom trenutku može postojati niz procesa pokrenutih na nizu računala u distribuiranom sustavu i svi trebaju koristiti unikatne oznake za objekte. Štoviše, nakon što je neki objekt asociran s nekom referencom izbrisan, vrlo je važno da se vrijednost reference ne koristi ponovo. Stoga svaki poziv na izbrisani objekt treba proizvesti grešku, umjesto da se eventualno spoji na novokreirani objekt.

Postoji nekoliko načina osiguranja jedinstvenosti referenci udaljenog objekta. Jedan način je da se povežu slijedeći podaci:

- Internet adresa računala
- broj porta na koji je spojen proces koji je kreirao objekt

- Vrijeme kreacije objekta
- lokalni broj objekta (inkrementiran prilikom kreacije svakog objekta)

Slika 6.4 prikazuje predloženi format.



6.4 Prijedlog formata reference udaljenog objekta

Međutim, postoje sustavi u kojima se objekt može realocirati na različiti proces na drugom računalu. U tom slučaju treba koristiti reference udaljenog objekta čija vrijednost nije ovisna o lokaciji.

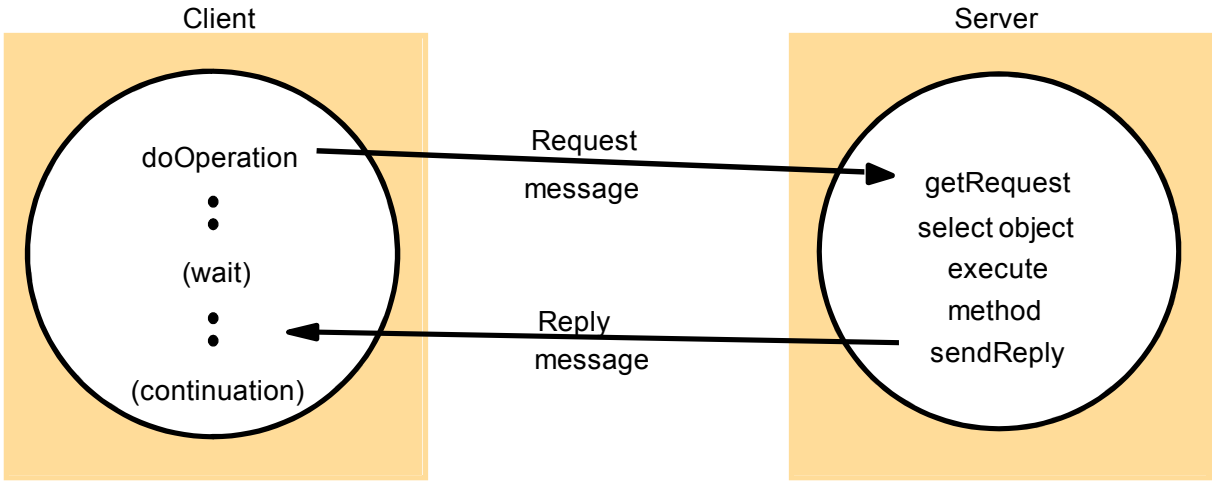
Zadnje polje na prethodnoj slici predstavlja informaciju o sučelju objekta npr. naziv sučelja.

6.5 Klijent server komunikacija

Ova vrsta komunikacije omogućava uloge i izmjenu poruka u tipičnim klijent-server interakcijama. Uobičajeno je da se radi o sinkronoj komunikaciji jer klijent proces blokira dok ne primi odgovor od servera. Također takva komunikacija može biti pouzdana jer odgovor od servera efektivno predstavlja potvrdu klijentu.

U slijedećem tekstu klijent-server izmjena poruka opisana je upotrebom *send* i *receive* metoda Java API-ja za UDP datagrame iako mnoge implementacije koriste TCP tokove. Klijent server protokol izgrađen na osnovi UDP protokola izbjegava suvišne troškove koje donose TCP tokovi odnosno:

- potvrde u TCP tokovima su suvišne jer svaki zahtjev zahtjeva odgovor
- uspostava komunikacije zahtjeva dva dodatna para poruka
- kontrola toka je suvišna za većinu poziva metoda jer se prenose relativno male poruke argumenata i rezultata



6.5 Zahtjev-odgovor komunikacija

Protokol zahtjev-odgovor ◊ Slika 6.5 prikazuje protokol koji se zasniva na tri komunikacijske operacije : *doOperation, getRequest* i *sendReply*. Većina RMI i RPC protokola o osnovi sadrže navedeni protokol. RMI protokol poziva metodu udaljenog objekta preko njegove reference.

Detaljniji opis operacija:

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

Šalje poruku zahtjeva udaljenom objektu i vraća odgovor pozivatelju. Argumenti specificiraju referencu udaljenog objekta, metodu koja se poziva i argumente poziva

public byte[] getRequest ();

prima zahtjev klijenta preko serverskog porta

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

vraća odgovor klijentu u obliku reply poruke na njegovu Internet adresu i port

Podrazumijeva se da klijent koji poziva *doOperation* izvršava postrojavanje (marshalling) argumenata prilikom slanja i prijema poruka.

Slika 6.6 prikazuje format *Request i* reply poruka. Prvo polje određuje da li se radi o zahtjevu ili odgovoru. Drugo polje generira doOperation prilikom svakog novog zahtjeva, a server kopira isti sadržaj u svaki odgovor. Treće polje je jedinstvena referenca na objekt. Četvro polje može biti redni broj metode u korištenom sučelju. Peto je niz bajtova koji predstavljaju argumente.

Ako s obe strane stoji isti jezik koji podržava refleksiju (mogućnost introspekcije programa u izvršavanju; JavaJava) onda je moguće da se umjesto id metode prenese i kompletan sadržaj metode !

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

6.6 Struktura zahtjev-odgovor poruke

Identifikatori poruka ◊ Svaka shema koja ima zahtjeve za pouzdanom isporukom poruka zahtjeva jedinstvenu identifikaciju poruka. Identifikator poruke sastoji se od dva dijela:

- *requestId* koji je uzima iz rastuće sekvence cijelih brojeva pridružene procesu koji šalje poruku
- identifikatora procesa pošiljaoca; npr. njegov port i Internet adresa

Kada vrijednost *requestId* pređe maksimalnu (npr. 2^{32}-1) onda se resetira na nulu. Treba paziti da je duljina života identifikatora mnogo manja od vremena iscrpljivanja opsega identifikatora.

Model pogrešaka request-reply protokola ◊ Ako se prije navedene tri primitivne funkcije implementiraju preko UDP protokola, tada će imati iste komunikacijske pogreške (propust slanja poruke, redoslijed primanja koji ne odgovara redoslijedu slanja).

Protokol može imati i pogrešku zbog rušenja procesa.

Zbog svega navedenog *doOperation* koristi timeout kada čeka na odziv servera. Akcija koja se poduzima nakon isteka timeouta ovisi o stupnju garancija isporuke poruka.

Timeouti ◊ Postoje različite opcije što doOperation može raditi poslije timeouta. Najjednostavnija opcija je izlazak iz doOperation s indikacijom klijentu da doOperation nije uspio. To nije uobičajeni pristup zbog mogućnosti da se dogodio gubitak poruke pa se obično nakon timeouta šalju ponovljeni zahtjevi serveru. zahtjevi se prekidaju nakon određenog broja ponavljanja kad se može zaključiti da nisu uzrokovani gubitkom poruke.

Odbacivanje dupliciranih poruka zahtjeva ◊ U slučaju retransmisije poruke može se dogoditi da server primi poruku istog zahtjeva više puta. Npr. server može primiti prvu poruku i zbog opterećenja može se dogoditi da se prije dogodi timeout nego što server obradi poruku i odgovori klijentu. Stoga svaka poruka mora biti označena s identifikacijskim brojem kako bi server odbacio duplicirane poruke, tj. da ne bi obrađivao više puta isti zahtjev.

Izgubljena poruka odgovora ◊ Ako je server poslao poruku odgovora i ona se izgubi, nakon timeouta klijenta primit će dupliciranu poruku zahtjeva. Server će morati ponovo obraditi poruku osim ako nije pohranio rezultat originalne obrade. Obrada nekih zahtjeva je takva da se imaju isti efekt na stanje servera i daju isti rezultat ako se uzastopno izvršavaju. Takve operacije nazivaju se idempotentne operacije (npr dodavanje elementa u skup elemenata). Međutim neke operacije nisu idempotentne (dodavanje elementa na kraj liste). Serveri koji

imaju samo idempotentne operacije ne trebaju poduzimati dodatne mjere za obradu dupliciranih zahtjeva.

Povijest izvršavanja ◊ Za servere koji zahtjevaju retransmisiju poruka bez potrebe za ponovnim izvršavanjem operacija može se koristiti povijest izvršavanja. Povijest izvršavanja je struktura podataka koja sadrži transmitirane poruke odgovora. Jedan zapis povijesti sadrži identifikator zahtjeva, poruku i dentifikator klijenta koji je poslao zahtjev. povijest izvršavanja ima memorijski trošak pa server mora posjedovati mehanizam koji će određivati koje poruke nisu više potrebne za retransmisiju.

Kako klijent upućuje zahtjeve jedan za drugim, server može uzimati pojavu novog zahtjeva kao potvrdu njegovog odgovora na prethodni zahtjev. Stoga povijest izvršavanja treba sadržavati samo zadnju poruku odgovora za svakog klijenta. Volumen poruka i dalje može biti problem u slučaju velikog broja klijenata. Na kraju kod terminacije klijent procesa neće biti potvrđen zadnji odgovor servera, pa odgovarajuća poruka u povijesti izvršavanja treba biti odbačena nakon nekog predefiniranog vremena.

RPC protokol ◊ Postoje u osnovi tri protokola koji se različito ponašaju u slučaju komunikacijskih pogrešaka, a služe za implementaciju različitih tipova RPC. Identificirao ih je Spector[1982]:

- zahtjev (request) (R) protokol;
- zahtjev-odgovor (request-reply) (RR) protokol;
- zahtjev-odgovor-potvrda (request-reply-acknowledge) (RRA) protokol;

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

6.7 RPC protokoli izmjene poruka

Slika 6.7 prikazuje poruke koje se izmjenjuju u navedenim protokolima. Primjer *R* protokola je slanje UDP datagrama te posjeduje sve komunikacijske nedostatke.RR protokol je dovoljan za većinu klijent-server implementacija. Posebna potvrda nije potrebna ako se uzme da je slanje odgovora klijentu zapravo potvrda prijema zahtjeva klijenta i da je slijedeći zahtjev klijenta zapravo potvrda prijema odgovora od servera.

RRA protokol koristi potvrdu klijenta o prijemu odgovora i slanje potvrde ne blokira klijent. Potvrdu klijenta server može koristiti za efikasnije upravljanje s sadržajem povijesti izvršavanja.

6.6 Grupna komunikacija

Izmjena poruka u paru nije najbolji način za komunikaciju jedne grupe procesa s drugom grupom procesa ili unutar grupe procesa. U tom slučaju *multicast* način komunikacije je prikladniji. U tom načinu komunikacije jedan proces šalje jednu poruku grupi procesa pri čemu proces pošiljatelj ne vodi računa o tome tko su članovi grupe (članstvo je transparentno za proces pošiljatelj). Postoji niz načina u poželjnom djelovanju multicast komunikacije. Najjednostavniji multicast način komunikacije ne pruža nikakve garanciju u pogledu isporuke i redoslijeda isporuke.

Multicast poruke predstavljaju korisnu infrastrukturu za konstrukciju distribuiranih sustava sa slijedećim karakteristikama:

1. *Replicirani servisi otporni na greške.* Replicirani servis sastoji se od grupe servera. Zahtjevi klijenta se multicastiraju na sve članove grupe i svi obavljaju identične operacije. Iako neki od servera može omanuti, klijent će ipak dobiti uslugu.
2. *Traženje servera za otkrivanje (discovery server) u spontanom umrežavanju.* Multicast poruke mogu slati i serveri i klijenti da bi locirali servise otkrivanja (resursa) prilikom spontanog umrežavanja.
3. *Poboljšanje performansi korištenjem repliciranih podataka:* Podaci su replicirani kako bi se povećale performanse servisa. Svaki put kada se podatak promijeni podaci su multicatirani procesima koji upravljaju replikama.
4. *Propagacija upozorenja o događajima (event notification):* Npr. kad stigne poruka na neku news grupu svi korisnici mogu biti obaviješteni multicast porukom.

Grupna komunikacija je u osnovi podržana IP multicast-om. IP multicast pruža određene mogućnosti koje u nekim tipovima distribuiranih aplikacija mogu biti nedovoljne pa tada je potrebno implementirati dodatna svojstva.

6.6.1 IP multicast – implementacija grupne komunikacije

IP multicast je izgrađen na osnovu Internet protokola, IP. IP protokol u adresiranju upotrebljava IP adresu računala – a pojam *porta* se uvodi na sloju TCP ili UDP protokola. IP multicast omogućava slanje jednog IP paketa na skup računala koji formiraju multicast grupu. Pošiljaoc nije svjesan identiteta individualnih primatelja i veličine grupe. *Multicast* grupa se specificira korištenjem D klase internet adresa, tj adresa koje imaju prva 4 bita jednako 1110 (224-239 - IPv4).

Članstvo u multicast grupi je dinamičko, tj. članovi mogu ulaziti ili napuštati grupu u bilo koje vrijeme i mogu biti članovi više grupa istovremeno. Moguće je slati datagrame grupi, a da niste član grupe.

Na stupnju aplikativnog programiranja IP muliticast je dostupan samo posredstvom UDP. Proces obavlja multicast na način da šalje UDP datagrame s multicast adresom i uobičajenim brojevima portova. Proces se može postati članom određene multicast grupe na način da svoj socket pridruži toj grupi. NA IP stupnju, računalo postaje član multicast grupe kada jedan ili više njegovih socketa pripada toj grupi. Kada multicast poruka stigne na računalo ona se prosljeđuje na sve sockete koji su vezani za specificiranu multicast adresu i port (više socketa iz različitih procesa vezano na jedan port !).

Splijedeći detalji su specifični za IPv4:

Multicast routeri: IP paketi se mogu multicastirati i na lokalnu mrežu i na širi Internet. Lokalni multicast koristi se multicast sposobnostima lokalne mreže, npr. Etherneta. za multicastiranje na razini Interneta potrebna je podrška multicast usmjernika. Kako bi se limitirala udaljenost propagacije multicast datagrama, moguće je specificirati broj dozvoljenih prolaza kroz router (TTL time to live).

*Multicast alokacija adresa:*Multicast adrese mogu biti stalne ili privremene. Stalne grupe egzistiraju i kada nema članova. Njihove adrese su u području 224.0.0.1 do 224.0.0.255. Prva adresa odnosi se na sve multicast hostove.

Ostatak adresa koristi se za privremene grupe. Privremena grupa se kreira neposredno prije korištenja i prestaje postojati nakon što je napuste svi članovi. Kada se kreira privremena grupa ona zahtjeva trenutno nekorištenu multicast adresu. IP multicast protokol ne rješava taj problem. Kada se radi o lokalnoj komunikaciji moguće je uz mali TTL izbjeći korištenje adrese koju koristi neka druga grupa unutar TTL opsega. Programi koji koriste multicast na nivou Interneta trebaju rješenje za taj problem posredstvom drugih aplikacija.

Model pogrešaka za multicast datagrame ◊ Datagrami multicastirani preko IP multicast protokola imaju iste pogreške kao i UDP datagrami. Stoga imamo pojam *nepouzdanog multicasta* koji podrazumijeva da nemamo garanciju isporuke poruke ni za jednog člana grupe.Postoji i nadgradnja IP multicast protokola koja daje garancije isporuke tzv. *pouzdani multicast.*

Java API za IP multicast ◊ Java API posjeduje sučelje prema IP multicast kroz klasu MulticastSocket , koja je subklasa klase DatagramSocket. Dodano svojstvo je mogućnost pridruženja i slanja poruka određenoj multicast grupi. Posjeduje dva konstruktora koji omugućavaju bilo vezanje na specificirani port ili korištenje slobodnog porta dodijeljenog od strane operacijskog sustava.

Slijedi kod Java programa koji koristi multicast:

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
      public static void main(String args[]){
       // args give message contents & destination multicast group (e.g.
"228.5.6.7")
      MulticastSocket s =null;
       try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                  new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
```

```
        // this figure continued on the next slide
    // get messages from others in group
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) {
                DatagramPacket messageIn =
                  new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
            System.out.println("Received:"+new
            String(messageIn.getData()));
            }
            s.leaveGroup(group);
          }catch (SocketException e){System.out.println("Socket: " +
e.getMessage());
         }catch (IOException e){System.out.println("IO: " +
e.getMessage());}
      }finally {if(s != null) s.close();}
    }
}
```

Osim metoda navedenih u kodu Java dozvoljava i postavljanje TTL pomoću metode *setTimeToLive*. Pretpostavljena vrijednost je 1, što dozvoljava propagaciju samo kroz lokalnu mrežu. Aplikacije koje koriste multicast mogu koristiti više portova, npr. jedan za izmjenu podataka, a drugi za kontrolne podatke.

6.6.2 Pouzdanost i redoslijed multicast protokola

Već smo vidjeli da IP multicast trpi propuste isporuke poruka. Svaki od primatelja poruka može ispustiti poruku zbog prepunjenosti međuspremnika ili poruka može biti zadržana u usmjerniku ,itd. S druge strane distribuiranih algoritama zasniva se na pretpostavci pouzdanog multicasta.

Redoslijed poruka je drugi problem. IP paketi poslani ne moraju stići u redoslijedu kako su poslani. To ima slijedeće efekte:

- niz poslanih datagrama od strane jednog pošiljaoca mogu biti primljeni u različitim rasporedima od strane primaoca
- datagrami poslani od dva različita procesa ne moraju stići u istom redoslijedu svim članovima grupe.

Primjeri efekata zbog pouzdanosti i redoslijeda ◊ Na već korištenim primjerima korištenja multicast protokola možemo razmotriti efekte modela pogrešaka:

1. *Replicirani servisi otporni na greške.* Ako razmotrimo replicirani servis koji se sastoji grupe servera koja počinje u istom inicijalnom stanju i obavljaju operacije u istom redoslijedu onda možemo očekivati da su u međusobno konzistentnim stanjima. Ovakva implementacija zahtjeva da poruku rebaju primiti svi članovi grupe ili nijedan. Svako djelomično slanje izaziva nekonzistentnost stanja. U većini implementacija bitan je redoslijed poruka.
2. *Traženje servera za otkrivanje (discovery server) u spontanom umrežavanju.* U ovom slučaju šroces koji želi otkriti resurs periodično multicastira šalje poruke, pa nije bitno da li se neka poruka izgubi. Aplikacije mogu direktno koristiti IP multicast protokol.
3. *Poboljšanje performansi korištenjem repliciranih podataka:* Razmatramo slučaj kada se replicirani podaci multicastiraju (ne operacije nad podacima). U tom slučaju pojedine replike mogu biti neažurne i s različitim redoslijedom podataka. Ovisno o tipu servisa koji se postiže repliciranim podacima to može biti bitno ili ne . Npr. u slučaju newsgroup-a može se dopustiti da pojedine replika su privremeno neažurne ili u različitim redoslijedima.
4. *Propagacija upozorenja o događajima (event notification):*Ovisi o aplikaciji. Npr. lookup servisi mogu objavljivati svoje postojanje korištenjem osnovnog IP multicasta.

BIBLIOGRAFIJA

[1]