**CS230 – Skripta**

**L01\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Koja je razlika između centralizovanog, decentralizovanog i distribuiranog sistema? Nacrtati grafik i opisati razlike.**

Centralizovani racunarski sistem je arhitektura u kojoj je sav softver, podaci i kontrola koncentrisana u jednom centralnom racunaru ili serveru. Svi korisnici pristupaju resursima preko tog centralnog cvora.

U decentralizovanim sistemima postoji više centralnih čvorova (servera) koji međusobno dele odgovornost za upravljanje podacima i resursima. Klijenti se mogu povezivati sa bilo kojim od ovih centralnih čvorova.

Distribuirani racunarski sistem je arhitektura u kojoj se procesi i resursi racunarskog sistema nalaze na vise fizicki razdvojenih racunara koji su povezani mrezom. Osnovna karakteristika distribuiranih sistema je transparentnost, tj. sposobnost da se korisnicima i aplikacijama sakrije da se procesi i resursi nalaze na razlicitim racunarima.

1. **Koje su bitne razlike između projektovanja decentralizovanog i distribuiranog sistema? Objasniti integrativni i ekspanzivni pristup.**

**Decentralizovani sistem:**

* **Arhitektura:** Sastoji se od više centralnih čvorova (servera), gde svaki server može imati grupu klijenata. Centralni čvorovi mogu međusobno komunicirati kako bi razmenjivali informacije i resurse.
* **Topologija:** Postoji nekoliko ključnih čvorova sa kojima su povezani klijenti. Ovi ključni čvorovi mogu biti povezani u mrežu, ali svaki centralni čvor je ključna tačka za svoje klijente.

**Distribuirani sistem:**

* **Arhitektura:** Svi čvorovi su ravnopravni i nema centralnog čvora. Svi čvorovi mogu obavljati funkcije servera i klijenta.
* **Topologija:** Svi čvorovi su povezani u mrežu bez centralne tačke, što omogućava direktnu komunikaciju između bilo koja dva čvora.

#### **Integrativni pristup**

* **Opis:** Integrativni pristup fokusira se na integraciju različitih komponenti u kohezivnu celinu. Ovaj pristup je često primenjivan u decentralizovanim sistemima gde je potrebno da centralni čvorovi efikasno sarađuju i koordiniraju resurse i podatke.
* **Primena:** Koristi se za postizanje bolje koordinacije i upravljanja između centralnih čvorova. Može uključivati centralizovane metode sinhronizacije podataka, upravljanje zajedničkim resursima i koordinaciju mrežnog saobraćaja.
* **Prednosti:** Poboljšana koordinacija i efikasnost, lakša implementacija kontrolnih mehanizama.
* **Izazovi:** Složenost u implementaciji i održavanju sinhronizacije između više centralnih čvorova.

**Ekspanzivni pristup**

* **Opis:** Ekspanzivni pristup naglašava širenje i skalabilnost sistema, omogućavajući dodavanje novih čvorova i resursa bez značajnog povećanja složenosti upravljanja. Ovaj pristup je često primenjivan u distribuiranim sistemima.
* **Primena:** Fokusira se na dizajniranje sistema koji može lako rasti dodavanjem novih čvorova, sa minimalnim uticajem na postojeću infrastrukturu. Koristi tehnike poput decentralizovanih algoritama za konsenzus, distribuirane baze podataka i P2P (peer-to-peer) mreža.
* **Prednosti:** Visoka skalabilnost, fleksibilnost i otpornost na otkaze. Jednostavno dodavanje novih čvorova.
* **Izazovi:** Složenost u postizanju konzistentnosti i pouzdanosti, potreba za naprednim algoritmima za distribuciju podataka i resursa.

1. **Šta predstavlja svojstvo transparentnosti distribuiranog sistema?**

Transparentnost distribuiranog sistema odnosi se na sposobnost sistema da sakrije složenost svojih distribuiranih procesa i resursa od krajnjih korisnika i aplikacija. To znači da korisnici i aplikacije ne moraju biti svesni činjenice da su resursi i procesi fizički raspoređeni na različitim lokacijama, često na velikim udaljenostima. Krajnji korisnici imaju osećaj da rade sa jedinstvenim, integrisanim sistemom, iako su resursi i procesi zapravo distribuirani.

1. **Koji su tipovi transparentnosti distribuiranog sistema? Nabrojati i opisati svih sedam tipova.**

**Transparentnost pristupa**

Transparentnost pristupa se bavi sakrivanjem razlika u načinima predstavljanja podataka i načina na koji se objektima može pristupiti. Korisnici i aplikacije ne moraju biti svesni različitih konvencija imenovanja, operacija sa datotekama ili komunikacionih protokola koje koriste različiti sistemi u mreži.

**Transparentnost lokacije**

Transparentnost lokacije omogućava korisnicima da ne znaju gde se fizički nalazi resurs u sistemu. Resursi su imenovani logički, bez informacija o njihovoj fizičkoj lokaciji, omogućavajući pristup resursima kao da su lokalni.

**Transparentnost relokacije**

Transparentnost relokacije odnosi se na pomeranje resursa ili procesa unutar sistema bez znanja korisnika. Ova transparentnost je ključna za sisteme računarstva u oblaku, gde resursi mogu biti dinamično premješteni radi optimizacije performansi ili resursa.

**Transparentnost migracije**

Transparentnost migracije omogućava mobilnost procesa i resursa iniciranu od strane korisnika, bez uticaja na tekuće operacije ili komunikaciju. Primeri uključuju kontinuiranu komunikaciju preko mobilnih telefona ili praćenje robe u transportu.

**Transparentnost replikacije**

Transparentnost replikacije sakriva postojanje više kopija resursa ili procesa od korisnika. Sistem automatski upravlja replikama kako bi osigurao konzistentnost i raspoloživost, često podržavajući i transparentnost lokacije kako bi se omogućio pristup replikama na različitim lokacijama.

**Transparentnost konkurencije**

Transparentnost konkurencije omogućava korisnicima da ne primećuju istovremeni pristup istim resursima od strane drugih korisnika. Konzistentnost resursa se održava kroz mehanizme kao što su zaključavanje, koji osiguravaju ekskluzivni pristup deljenim resursima kada je to potrebno.

**Transparentnost otkaza**

Transparentnost otkaza omogućava sistemu da maskira kvarove i automatski se oporavlja bez uticaja na korisnike. Korisnici ne primećuju otkaze jer sistem preusmerava zahteve i nastavlja rad koristeći dostupne resurse. Međutim, maskiranje kvarova je složeno i ponekad nemoguće zbog teškoća u razlikovanju između mrtvih i sporih procesa.

1. **Pored transparentnosti, distribuirani sistem ima za cilj da bude otvoren i pouzdan. Opisati značenje ovih ciljeva.**

**Otvorenost** distribuiranog sistema se odnosi na sposobnost sistema da omogući integraciju i upotrebu njegovih komponenti u druge sisteme, kao i da sam koristi komponente izvan sopstvenog okvira. Otvoreni sistemi su fleksibilni, skalabilni i omogućavaju lako prilagođavanje i zamenu komponenti bez ometanja ukupnog funkcionisanja sistema.

**Pouzdanost** se odnosi na sposobnost računarskog sistema da funkcioniše kako se očekuje, čak i u prisustvu grešaka ili otkaza. U distribuiranim sistemima, pouzdanost je posebno složena zbog mogućnosti delimičnih otkaza, gde određene komponente mogu zakazati dok sistem kao celina i dalje funkcioniše, iako možda sa smanjenim performansama.

1. **Nabrojati tehnike skaliranja računarskog sistema.**

**Horizontalno skaliranje** podrazumeva dodavanje više servera u postojeći sistem kako bi se povećao njegov ukupni kapacitet.

* Dodavanje novih servera:

Uključivanje dodatnih servera u mrežu kako bi se raspodelilo opterećenje između više mašina.

* Balansiranje opterećenja (Load Balancing):

Korišćenje load balancera za raspodelu zahteva između više servera kako bi se izbeglo preopterećenje jednog servera i poboljšale performanse sistema.

* Klasterizacija:

Organizovanje servera u klastere koji rade zajedno kao jedinstveni sistem, pružajući visoku dostupnost i bolju skalabilnost.

**Vertikalno skaliranje** podrazumeva povećanje kapaciteta postojećih resursa u sistemu.

* Povećanje memorije (RAM):

Dodavanje više RAM memorije serveru kako bi mogao da obradi veći broj zahteva i poboljša performanse aplikacija koje koriste puno memorije.

* Nadogradnja CPU-a:

Instaliranje bržih ili više jezgara procesora kako bi se povećala brzina obrade podataka i poboljšala ukupna performansa servera.

* Zamena mrežnih modula:

Nadogradnja mrežnih adaptera ili prebacivanje na brže mrežne tehnologije (npr. prelazak sa 1Gbps na 10Gbps mreže) kako bi se povećao protok podataka i smanjila latencija.

1. **Nabrojati i opisati česte propuste pri projektovanju sistema. Dati i primer za propust.**

Računarska mreža je pouzdana

Računarska mreža je bezbedna

Računarska mreža je homogena

Topologija mreže se ne menja

Kašnjenje je jednako nuli

Propusni opseg je beskonačan

Cena prenosa podataka je jednaka nuli

Postoji samo jedan administrator

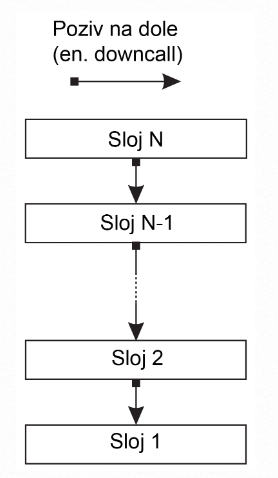
**L02 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Opisati slojevite sisteme koji su jednosmerni. Nacrtati dijagram i objasniti tok prenosa poruka (zahteva/odgovora).**

Jednosmerne arhitekture u distribuiranim aplikacijama mogu biti od dva tipa: jednosmerne arhitekture bez preskakanja slojeva i jednosmerne arhitekture sa preskakanjem slojeva.

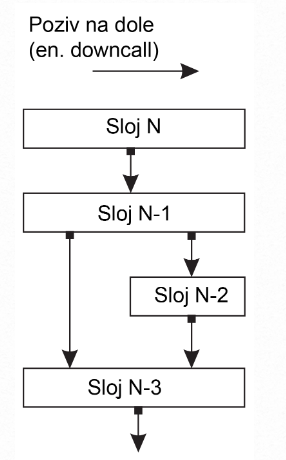
**Jednosmerne arhitekture bez preskakanja slojeva**

U ovim arhitekturama, podaci i zahtevi putuju kroz sve slojeve aplikacije redom, od gornjeg (aplikativnog) sloja do donjeg (npr. podatkovnog) sloja, bez preskakanja.

****

**Jednosmerne arhitekture sa preskakanjem slojeva**

U ovim arhitekturama, podaci i zahtevi mogu preskakati slojeve kako bi se optimizovala performansa ili efikasnost aplikacije.



* 1. **Opisati slojevite sisteme koji su jednosmerni, ali koji mogu preskakati slojeve. Nacrtati dijagram i objasniti tok prenosa poruka (zahteva/odgovora).**

Ima u proslom pitanju

* 1. **Opisati slojevite sisteme koji su dvosmerni. Nacrtati dijagram i objasniti tok prenosa poruka (zahteva/odgovora). Dati primer ovakvog sistema.**

Dvosmerna arhitektura u kontekstu distribuiranih sistema odnosi se na situaciju kada se komunikacija između slojeva odvija u oba smera, tj. i od nižeg ka višem sloju (downcall) i od višeg ka nižem sloju (upcall).

Karakteristike dvosmerne arhitekture:

**Downcall (od nižeg ka višem sloju):**

* Ova komunikacija se odvija kada niži slojevi aplikacije pozivaju funkcije ili usluge koje pružaju viši slojevi.
* Primeri downcall-a uključuju zahteve za obradom podataka ili pristup resursima koji su u višim slojevima.

**Upcall (od višeg ka nižem sloju):**

* Upcall predstavlja situaciju kada viši slojevi aplikacije pozivaju funkcije ili usluge koje pružaju niži slojevi.
* Tipični primeri upcall-a uključuju signalizaciju događaja ili obaveštenja o promenama stanja koje niži slojevi trebaju da obrade ili reaguju na njih.

U primeru operativnog sistema, niži slojevi mogu generisati događaje kao što su pritisak tastera ili dolazak mrežnog paketa. Ti događaji se zatim šalju višim slojevima putem upcall-a, gde se može izvršiti dalja obrada ili reakcija na te događaje. Na primer, aplikacija može reagovati na pritisak tastera za pokretanje određene funkcije ili na dolazak mrežnog paketa za obradu podataka.

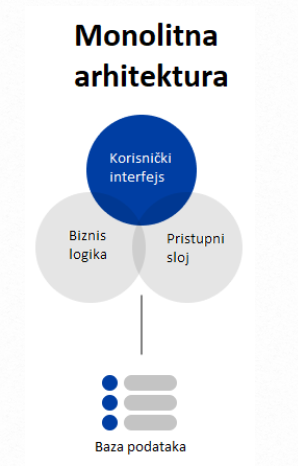
* 1. **Opisati prednosti i nedostatke razvoja monolitne aplikacije. Nacrtati dijagram.**

### Prednosti monolitne arhitekture:

1. **Jednostavnost početka:** Monolitna arhitektura omogućava jednostavan početak projekta jer se sve komponente nalaze unutar jednog aplikacijskog procesa. Ovo olakšava razvoj i prvo lansiranje aplikacije.
2. **Centralizovana baza podataka:** Svi delovi aplikacije dele istu bazu podataka, što pojednostavljuje dizajn, organizaciju podataka i održavanje konzistentnosti podataka.
3. **Jednostavno raspoređivanje:** Deployment (raspoređivanje) monolitne aplikacije je relativno jednostavno jer se svi delovi aplikacije objedinjuju u jedan deployable unit (jedinica za raspoređivanje).
4. **Manje kompleksnosti pri početku:** Ne postoji potreba za upravljanjem i održavanjem komunikacije između različitih mikroservisa ili komponenti, što smanjuje početnu kompleksnost.

### Mane monolitne arhitekture:

1. **Skalabilnost:** Monolitne aplikacije često imaju problema sa skaliranjem. Skaliranje se može postići samo vertikalnim dodavanjem resursa (npr. više RAM-a, jači procesor), što može postati skupo i ograničeno.
2. **Održavanje i rizik promena:** Svaka promena u kodu može imati potencijal da utiče na sve delove aplikacije. Ovo može povećati rizik od grešaka i otežati održavanje aplikacije kako raste.
3. **Teškoće u razdvajanju koda:** Kako aplikacija raste, teže je održavati čist kod i organizovati ga pod kontrolom. Ovo može dovesti do kompleksnosti i teškoća pri dodavanju novih funkcionalnosti.
4. **Fleksibilnost i inovacija:** Monolitne arhitekture mogu ograničiti fleksibilnost i inovacije jer sve komponente su usko povezane. Uvođenje novih tehnologija ili rad sa različitim timovima može biti izazovnije.

****

1. **Opisati prednosti i nedostatke razvoja mikroservisne aplikacije. Nacrtati dijagram.**

### Prednosti mikroservisne arhitekture:

**Razdvajanje dužnosti**: Mikroservisna arhitektura omogućava jasno razdvajanje funkcionalnosti aplikacije na manje, specijalizirane servise. Svaki mikroservis je odgovoran za obavljanje određene poslovne logike ili funkcionalnosti, što olakšava razumijevanje, održavanje i nadogradnju aplikacije. Razvojni timovi mogu se fokusirati na razvoj samo jednog mikroservisa, što smanjuje kompleksnost i olakšava upravljanje kodom.

**Manje projekte za rad**: Zbog modularne prirode mikroservisne arhitekture, razvojni timovi mogu raditi na manjim, samostalnim projektima umjesto na jednom velikom kodnom monolitu. Ovo smanjuje rizik od prekida u razvoju, omogućava brži razvojni ciklus i olakšava upravljanje verzijama i promjenama.

**Više opcija za skaliranje i raspoređivanje**: Svaki mikroservis može se skalirati neovisno prema potrebama. To znači da je moguće optimizirati resurse i skalirati samo one dijelove aplikacije koji zahtijevaju više kapaciteta ili brži odziv. Također, mikroservisi se mogu raspoređivati na različite servere ili kontejnere, što omogućava bolje iskorištenje infrastrukturnih resursa i poboljšava otpornost na kvarove.

### Nedostaci mikroservisne arhitekture:

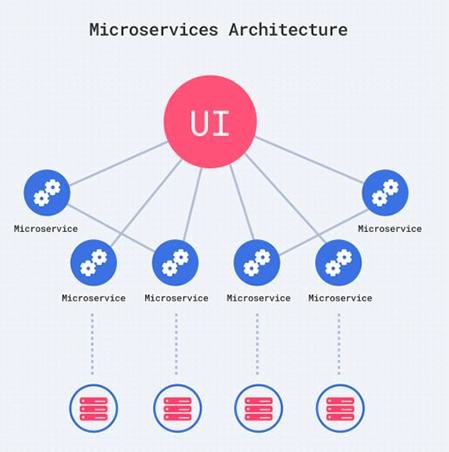
**Nelogično razdvajanje**: Postoji rizik da se aplikacija podijeli na mikroservise na način koji nije optimalan ili logičan, što može dovesti do prekomjerne komplikacije ili nepotrebnog mrežnog opterećenja. Pravilno planiranje granularnosti mikroservisa ključno je za efikasnu i skalabilnu arhitekturu.

**Više mrežne interakcije**: Svaka komunikacija između mikroservisa odvija se preko mreže, što može dovesti do povećane latencije i kompleksnosti u upravljanju. Potrebno je pažljivo optimizirati i nadgledati mrežne pozive kako bi se osigurala performantnost i stabilnost aplikacije.

**Skladištenje i dijeljenje podataka**: Mikroservisna arhitektura zahtijeva pažljivo razmatranje kako skladištiti i dijeliti podatke između mikroservisa. Svaki servis može imati svoju bazu podataka ili koristiti zajednički skup podataka, što može dovesti do složenosti u održavanju dosljednosti i integriteta podataka.

**Problemi kompatibilnosti**: Svaki mikroservis može biti razvijen na različitim tehnologijama ili verzijama, što može uzrokovati probleme u kompatibilnosti kada se integriraju. Potrebno je pažljivo upravljati verzioniranjem i kompatibilnošću među servisima kako bi se izbjegli problemi prilikom integracije i nadogradnji.

**Testiranje**: Testiranje mikroservisne arhitekture zahtijeva složenije strategije za integracijsko testiranje, kako bi se osiguralo da svi mikroservisi rade zajedno kao cjelina. Potrebno je testirati ne samo funkcionalnost pojedinačnih servisa, već i njihovu međusobnu komunikaciju i integraciju.

****

1. **Nabrojati karakteristike mikroservisnih aplikacija.**

**Karakteristike mikroservisa** uključuju:

* Mikroservisi su najčešće procesi koji komuniciraju putem mreže da izvrše zadatke koristeći protokole koji nisu vezani za određeni tip tehnologije poput HTTP-a. Ipak, servisi mogu da koriste i druge mehanizme inter-procesne komunikacije poput deljene memorije. Servisi, takođe, mogu biti pokrenuti u okviru istog procesa, na primer, OSGi paketi.
* Servisi u mikroservis arhitekturi se mogu posebno isporučivati.
* Servisi su lako zamenjivi.
* Servisi su organizovani oko funkcionalnosti, npr. front-end korisnički interfejs, preporuke, logistika, naplata, itd.
* Servisi se mogu implementirati korišćenjem različitih programskih jezika, baza, hardverskih i softverskih okruženja, u zavisnosti šta od toga najviše odgovara za implementaciju.
* Servisi su mali veličinom, omogućena je razmena poruka, povezani su sadržajem, nezavisno razvijani, decentralizovani, kao i bildovani i izdavani pomoću automatskih procesa.

1. **Detaljno opisati prednosti razvoja mikroservisnih aplikacija.**

**Skalabilnost**: Mikroservisne aplikacije omogućavaju granularnu skalabilnost. Svaki mikroservis može se skalirati nezavisno, što znači da se resursi mogu efikasnije raspoređivati prema potrebama aplikacije. Na primjer, ako je potreban veći kapacitet za servis koji obrađuje korisničke zahteve, samo taj servis može biti skaliran, dok drugi ostaju nepromenjeni.

**Fleksibilnost i agilnost u razvoju**: Svaki mikroservis može se razvijati, testirati i implementirati nezavisno od ostalih servisa. To omogućava timovima da brže iteriraju i isporučuju nove funkcionalnosti. Osim toga, različiti timovi mogu raditi na različitim mikroservisima istovremeno bez međusobnog ometanja.

**Tehnološka nezavisnost**: Svaki mikroservis može biti implementiran i upravljan pomoću različitih tehnologija, što omogućava timovima da odaberu najbolje alate i tehnologije za specifične zahteve servisa. Ovo doprinosi većoj fleksibilnosti u arhitekturi aplikacije.

**Jednostavnije održavanje**: Svaki mikroservis je manji i fokusiran na specifičnu funkcionalnost. To olakšava održavanje jer je lakše identifikovati, izolovati i rešavati probleme u određenom mikroservisu bez potrebe za kompletnim restartovanjem aplikacije.

**Otpornost na greške**: Mikroservisna arhitektura omogućava da se aplikacija i dalje delimično funkcioniše čak i ako jedan mikroservis ne radi ispravno. Ovo je zato što servisi komuniciraju putem mreže, a ne direktno, i mogu biti dizajnirani tako da se greške izoliraju i kontrolišu.

**Lakša integracija i kontinuirana dostava**: Mikroservisna arhitektura je pogodna za kontinuiranu integraciju i dostavu (CI/CD) zbog nezavisnosti servisa i mogućnosti automatskog implementiranja svake promene u servisu bez uticaja na ostale delove aplikacije.

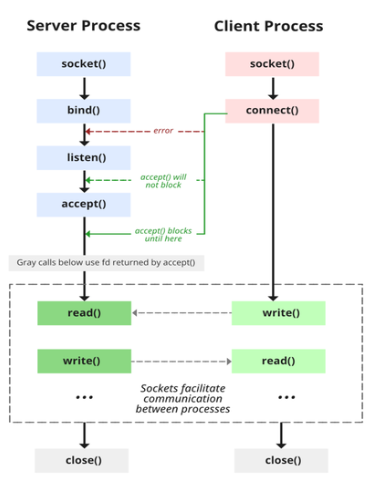
**Poboljšana skalabilnost timova**: Razdvajanje aplikacije na manje, samostalne servise omogućava timovima da se organizuju prema funkcionalnim oblastima ili servisima, što može dovesti do efikasnijeg upravljanja projektima i bolje skalabilnosti tima.

**Bolja upravljivost**: Mikroservisna arhitektura omogućava bolje upravljanje različitim aspektima aplikacije kao što su performanse, sigurnost, dostupnost i sl. Svaki servis može imati posebne metrike i pravila za upravljanje.

1. **Opisati princip mrežnog (socket) programiranja. Nacrtati (okvirni) dijagram toka pri komunikaciji serverske i klijentske komponente u mrežnom programiranju.**

**Princip mrežnog (socket) programiranja:**

1. **Kreiranje soketa**: Na obe strane komunikacije (serveru i klijentu) se kreira socket. Socket je endpoint koji služi za komunikaciju između dva računara ili procesa.
2. **Povezivanje (server-side)**:
   * Server kreira socket i čeka dolazne zahteve za konekciju od klijenata.
   * Kada klijent želi da uspostavi vezu, šalje zahtev serveru.
3. **Povezivanje (client-side)**:
   * Klijent takođe kreira socket i zatim se povezuje sa serverom koristeći IP adresu i port servera.
4. **Komunikacija**:
   * Nakon uspostavljanja veze, server i klijent mogu slati i primati podatke jedan od drugog.
   * Podaci se mogu slati u oba smera (od servera ka klijentu i obrnuto) u vidu sekvenci bajtova.
5. **Zatvaranje veze**:
   * Kada je komunikacija završena, ili kada jedna strana odluči da prekine vezu, soketi se zatvaraju.



**L03 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Opisati ključne karakteristike RESTful arhitekture, kao i operacije koje se koriste.**

**Ključne karakteristike RESTful arhitekture:**

1. **Identifikacija resursa kroz jedinstvenu šemu imenovanja**:
   * Resursi u RESTful arhitekturi se identifikuju putem Uniform Resource Identifier (URI). Svaki resurs ima jedinstveni URI koji omogućava klijentima da pristupe resursu direktno preko mreže. Na primer, resurs korisnika može biti identifikovan sa http://example.com/users/123.
2. **Uniformni interfejs**:
   * RESTful arhitektura koristi uniformni interfejs za interakciju sa resursima. Ovo podrazumeva da svi servisi nude isti interfejs koji se sastoji od četiri osnovne operacije: POST, GET, PUT, i DELETE. Ove operacije omogućavaju kreiranje, preuzimanje, ažuriranje i brisanje resursa.
3. **Potpuno opisane poruke**:
   * Poruke koje se šalju između klijenta i servera su samodokumentujuće. To znači da svaka poruka nosi sve potrebne informacije za razumevanje i obradu zahteva. U HTTP-u, ovo uključuje zaglavlja, telo poruke, metode, statusne kodove i URI-e.
4. **Izvršenje bez stanja (stateless execution)**:
   * U RESTful arhitekturi, server ne čuva nikakve informacije o stanju klijenta između zahteva. Svaki zahtev od klijenta ka serveru mora sadržati sve potrebne informacije za server da obradi taj zahtev. Ovo pojednostavljuje skaliranje servera i smanjuje potrebu za kompleksnim mehanizmima za upravljanje stanjem.

**Operacije koje se koriste u RESTful arhitekturi:**

**POST**

Koristi se za kreiranje novog resursa na serveru. Na primer, slanje POST zahteva na http://example.com/users sa podacima o novom korisniku kreira novog korisnika.

**GET**

Koristi se za preuzimanje resursa sa servera. Na primer, slanje GET zahteva na http://example.com/users/123 preuzima podatke o korisniku sa ID-jem 123.

**PUT**:

Koristi se za ažuriranje postojećeg resursa na serveru. Na primer, slanje PUT zahteva na http://example.com/users/123 sa izmenjenim podacima o korisniku ažurira informacije o korisniku sa ID-jem 123.

**DELETE**:

Koristi se za brisanje resursa sa servera. Na primer, slanje DELETE zahteva na http://example.com/users/123 briše korisnika sa ID-jem 123 sa servera.

* 1. **Opisati preplatničke arhitekture distribuiranih sistema - navesti sve tipove prema spregnutosti.**

Pretplatničke arhitekture (poznate i kao publish-subscribe arhitekture) su dizajn paradigme u distribuiranim sistemima koje omogućavaju efikasnu i fleksibilnu komunikaciju između različitih komponenti sistema. Ove arhitekture se koriste za smanjenje čvrste sprege između procesa, omogućavajući im da komuniciraju na način koji je manje direktan i manje zavistan od konkretnih identifikatora ili vremenskih uslova.

Osnovni koncepti pretplatničkih arhitektura

* Publikacija (Publish): Proces koji šalje informacije (publikuje poruke) sistemu.
* Pretplata (Subscribe): Proces koji prima informacije (pretplati se na poruke) iz sistema.
* Broker (posrednik): Središnji entitet koji posreduje između izdavača i pretplatnika, omogućavajući filtriranje i isporuku poruka.

Pretplatničke arhitekture (publish-subscribe) omogućavaju različite vrste spregnutosti između izdavača i pretplatnika. Ove vrste spregnutosti se mogu klasifikovati prema vremenskim i referencijalnim karakteristikama, slično kao u opštim distribuiranim sistemima.

**Vremenska spregnutost**

* Vremenski spregnuti sistemi:

Sinhrona komunikacija: Izdavači i pretplatnici moraju biti aktivni u isto vreme da bi se poruka isporučila.

Primer: Real-time streaming aplikacije kao što su video konferencije, gde svi učesnici moraju biti prisutni u isto vreme.

* Vremenski nespregnuti sistemi:

Asinhrona komunikacija: Izdavači i pretplatnici ne moraju biti aktivni u isto vreme. Poruke se čuvaju i isporučuju kasnije.

Primer: Email sistemi ili push notifikacije gde poruke mogu biti primljene i pročitane u različito vreme.

**Referencijalna spregnutost**

* Referencijalno spregnuti sistemi:

Direktno referenciranje: Izdavači i pretplatnici moraju eksplicitno znati jedni za druge da bi komunicirali.

Primer: Tradicionalni klijent-server sistemi gde klijent mora znati adresu servera za slanje zahteva.

* Referencijalno nespregnuti sistemi:

Indirektno referenciranje: Izdavači i pretplatnici ne moraju znati jedni za druge. Komunikacija se vrši preko posrednika (brokera).

Primer: Tipične publish-subscribe arhitekture kao što su Apache Kafka ili RabbitMQ, gde izdavači šalju poruke brokeru, a pretplatnici ih primaju od brokera bez direktnog znanja o izdavačima.

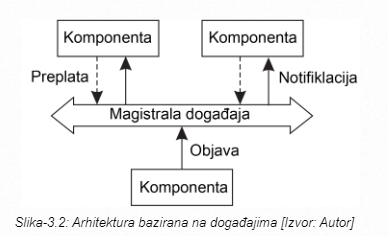
* 1. **Detaljno opisati direktnu i tzv. mailbox arhitekturu. Nacrtati dijagrame.**

Direktna arhitektura u distribuiranim sistemima odnosi se na koordinaciju između procesa koja zahteva da procesi budu vremenski i referencijalno spregnuti. Ovi sistemi omogućavaju neposrednu i sinhronu komunikaciju između procesa, što znači da su oba procesa aktivna u isto vreme i eksplicitno znaju jedni za druge.

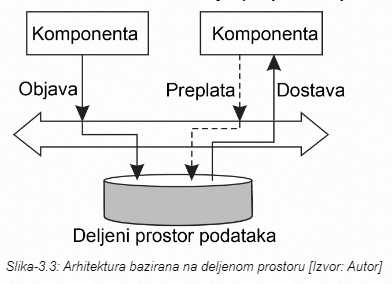
Mailbox (poštansko sanduče) arhitektura predstavlja način koordinacije među procesima u distribuiranim sistemima gde su procesi vremenski odvojeni, ali referencijalno spregnuti. Ova arhitektura omogućava komunikaciju putem poruka koje se stavljaju u poštansko sanduče, a ne kroz direktnu interakciju između procesa.

* 1. **Detaljno opisati koordinaciju baziranu na događajima i koordinaciju sa deljenim prostorom podataka. Nacrtati dijagrame.**

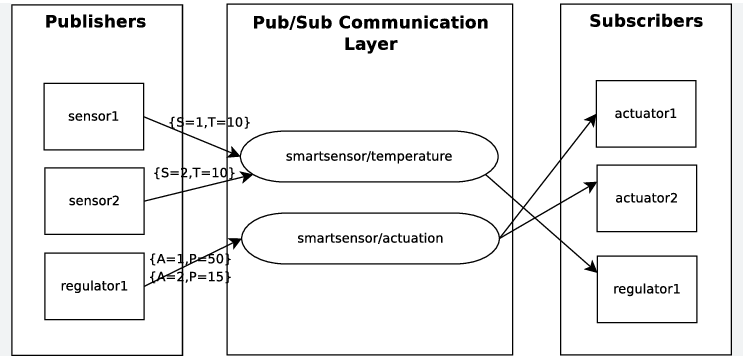
Koordinacija zasnovana na događajima (event-based coordination) je model komunikacije u distribuiranim sistemima gde procesi komuniciraju putem događaja bez potrebe za eksplicitnim poznavanjem jedni drugih. Ovo se postiže objavljivanjem obaveštenja o događajima na koje se drugi procesi mogu pretplatiti.

****

Sistem sa deljenim prostorom podataka omogućava procesima da komuniciraju i koordiniraju aktivnosti putem zajedničkog prostora u kojem se razmenjuju strukturisani podaci nazvani torke (tuples). Ovaj model koordinacije koristi kombinaciju referencijalno i vremenski odvojenih procesa.

****

* 1. **Nacrtati dijagram pretplatničkih arhitektura prema temi (en. topic-based publish-subscribe systems) i opisati način rada**

****

Topic-based publish-subscribe (pub-sub) sistemi su model komunikacije u distribuiranim sistemima gde se informacije razmenjuju putem "tema" (topics). Ovaj model omogućava asinkronu i skalabilnu komunikaciju između izdavača (publishers) i pretplatnika (subscribers) bez potrebe za direktnim poznavanjem jedni drugih.

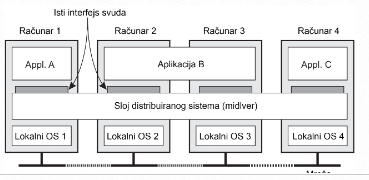
* 1. **Objasniti pojam midlvera: čemu služi, koje usluge nudi i gde se nalazi u slojevima distribuiranog sistema (nacrtati dijagram)**

Midlver (middleware) je ključna komponenta u distribuiranim sistemima koja služi kao posrednik između aplikacija i operativnog sistema računara. Njegova uloga je da olakša integraciju i komunikaciju između različitih aplikacija i servisa koji se izvršavaju na različitim računarima u mrežnom okruženju. Evo detaljnijeg objašnjenja i njegovog položaja u slojevima distribuiranog sistema:

**Uloga midlvera**

1. **Posrednik između aplikacija i operativnog sistema:**
   * Midlver pruža apstrakciju i usluge koje omogućavaju aplikacijama da se fokusiraju na svoje funkcionalnosti, nezavisno od specifičnosti hardvera i operativnih sistema računara na kojima se izvršavaju.
2. **Upravljanje resursima:**
   * Pruža mehanizme za efikasno deljenje resursa kao što su memorija, procesorsko vreme, mrežni resursi i druge komponente sistema.
3. **Podrška za komunikaciju:**
   * Omogućava komunikaciju između aplikacija koje se izvršavaju na različitim računarima u mreži, često kroz mehanizme kao što su RPC (Remote Procedure Call), CORBA (Common Object Request Broker Architecture), Web servisi, ili modernije pristupe poput REST (Representational State Transfer) arhitekture i GraphQL.
4. **Usluge zaštite:**
   * Pruža sigurnosne mehanizme za zaštitu podataka i komunikacije između aplikacija, uključujući autentifikaciju, autorizaciju i enkripciju.
5. **Maskiranje i oporavak od grešaka:**
   * Midlver može sakriti specifične greške ili problematične situacije od aplikacija, nudeći mehanizme za oporavak od grešaka i rešavanje problema bez potrebe da aplikacije direktno rukuju sa tim situacijama.

**Položaj midlvera u slojevima distribuiranog sistema**



**Aplikacijski sloj:** Ovde se nalaze same aplikacije koje pružaju svoje funkcionalnosti.

**Midlver:** Posrednički sloj koji se nalazi između aplikacija i operativnog sistema. Pruža usluge za komunikaciju, upravljanje resursima, zaštitu i druge funkcionalnosti.

**Operativni sistem (OS):** Ovde se nalazi operativni sistem računara koji pruža osnovne operativne resurse kao što su upravljanje memorijom, procesima, mrežom itd.

**Hardver:** Fizički računarski resursi kao što su CPU, memorija, mrežne kartice itd.

* 1. **Opisati ogranizaciju midlvera kroz wrapper. Nacrtati i opisati dijagram. Kako raste broj wrappera sa brojem aplikacija?**

Organizacija middleware-a kroz omotače (wrappers) je ključan koncept u arhitekturi distribuiranih sistema, posebno kada je potrebno integrisati različite komponente sa nekompatibilnim ili različitim interfejsima. Omotači su oblik dizajniranja softvera koji omogućava da se postojeće komponente prilagode tako da budu kompatibilne sa zahtevima klijentskih aplikacija, čime se olakšava integracija i interoperabilnost.

Broj omotača (wrappers) u distribuiranom sistemu obično raste sa porastom broja aplikacija i potrebe za integracijom različitih komponenti.

**Povećanje heterogenosti sistema:**

* Sa povećanjem broja aplikacija u sistemu, verovatnoća da će te aplikacije koristiti različite tehnologije, protokole ili interfejse takođe raste. Kako bi se omogućila komunikacija između ovih raznovrsnih aplikacija, može biti potrebno implementirati više omotača koji će posredovati u njihovoj komunikaciji.

**Potreba za prilagođavanjem interfejsa:**

* Svaka nova aplikacija koja se uvodi u sistem može zahtevati specifičan pristup ili interfejs za integraciju sa postojećim komponentama. Omotači se koriste kako bi se prilagodili interfejsi postojećih sistema ili servisa, čineći ih kompatibilnim sa zahtevima novih aplikacija.

**Različite tehnologije i arhitekture:**

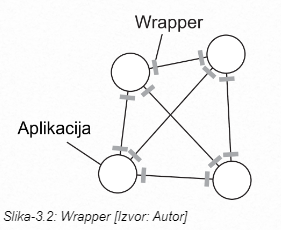
* Aplikacije koje koriste različite tehnologije (npr. RESTful servisi, SOAP servisi, GraphQL, legacy API-ji) ili koje su razvijene na različitim arhitekturnim principima (npr. mikroservisi, monolitne aplikacije) zahtevaju različite pristupe integraciji. Svaka od ovih tehnologija može zahtevati implementaciju specifičnih omotača radi uspešne integracije.

**Skaliranje i rast sistema:**

* Kako sistem raste i skalira se, može biti potrebno da se dodaju nove aplikacije koje pružaju dodatne funkcionalnosti ili usluge. Svaka nova aplikacija može zahtevati specifičan omotač kako bi se osiguralo da se nova funkcionalnost integriše bez narušavanja postojećih sistema.

**Fleksibilnost i evolucija sistema:**

* Omotači omogućavaju sistemima da budu fleksibilni i evolutivni, pružajući mehanizme za prilagođavanje i poboljšanje komunikacije između komponenti sistema. Kako se zahtevi i tehnologije menjaju tokom vremena, mogu biti potrebni novi omotači kako bi se podržali novi interfejsi ili arhitekture.

****

* 1. **Opisati ogranizaciju midlvera kroz broker. Nacrtati i opisati dijagram. Kako raste broj interfejsa brokera sa brojem aplikacija?**

Broker u kontekstu middleware-a je centralizovana komponenta ili sistemski entitet koji olakšava komunikaciju i koordinaciju između različitih aplikacija ili komponenti u distribuiranom sistemu. Glavna uloga brokera je da deluje kao posrednik između aplikacija tako što prima zahteve od jedne aplikacije, obrađuje ih i prosleđuje odgovore ili informacije drugoj aplikaciji ili više njih.

**Direktna komunikacija sa svakom aplikacijom:**

* U početku, kada postoji mali broj aplikacija, svaka aplikacija može direktno komunicirati sa svakom drugom aplikacijom u sistemu. Ovo znači da ne postoji potreba za posebnim interfejsima brokera jer se sve komunikacije obavljaju direktno.

**Dodavanje novih aplikacija:**

* Sa dodavanjem novih aplikacija u sistem, direktna komunikacija postaje neefikasna i teško održiva. Svaka nova aplikacija bi morala da uspostavi direktnu komunikaciju sa svim ostalim aplikacijama, što može dovesti do brzog rasta kompleksnosti i smanjenja skalabilnosti sistema.

**Potreba za centralizacijom i posredovanjem:**

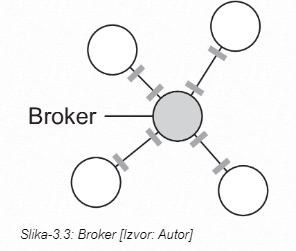
* Kako se broj aplikacija povećava, počinje da raste i potreba za centralizacijom komunikacije. Brokeri se uvode kako bi posredovali u komunikaciji između aplikacija, obezbeđujući jednostavan interfejs za slanje zahteva i primanje odgovora.

**Broj interfejsa brokera:**

* Sa povećanjem broja aplikacija, broj interfejsa brokera obično raste linearno. Svaka nova aplikacija može zahtevati jedan ili više interfejsa u brokeru kako bi se omogućila komunikacija sa ostalim aplikacijama u sistemu.

**Efikasnost i skalabilnost:**

* Korišćenjem brokera, sistem postaje efikasniji i skalabilniji u upravljanju komunikacijom između aplikacija. Umesto da svaka aplikacija održava direktnu vezu sa svim drugim aplikacijama, broker pruža centralizovanu tačku za upravljanje ovim vezama.



1. **Opisati kako midlver organizovan kao interceptor. Nacrtati dijagram.**

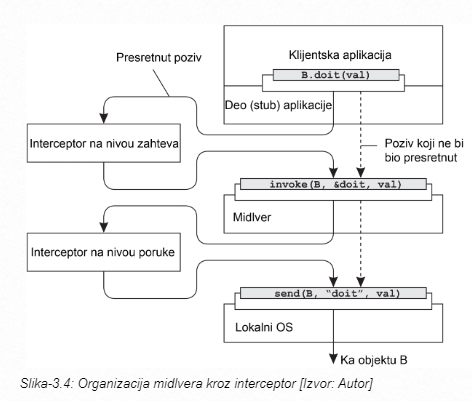
Interceptori su ključni konstrukt u organizaciji middleware-a jer omogućavaju prilagođavanje i proširivanje funkcionalnosti middleware-a prema specifičnim potrebama aplikacija. Evo kako se middleware može organizovati koristeći interceptore:

**Organizacija middleware-a kroz interceptore**

1. **Centralni modul za interceptore:**
   * Middleware obično ima centralni modul ili servis koji upravlja interceptorskim lancem ili nizom interceptora. Ovaj modul je odgovoran za registraciju, upravljanje i izvršavanje interceptora prema potrebi.
2. **Registracija interceptora:**
   * Aplikacije ili komponente mogu registrovati interceptore kod middleware-a kako bi reagovali na određene događaje ili zahteve. Registracija se obično vrši putem API-ja ili konfiguracionih fajlova.
3. **Prekidanje i izvršavanje koda:**
   * Kada middleware primi zahtev ili događaj, interceptorski modul prekida uobičajeni tok kontrole i omogućava izvršenje interceptora. Svaki interceptor može da izvrši specifičnu logiku pre ili posle obrade zahteva, uključujući validaciju, logovanje, transformaciju podataka ili slanje dodatnih zahteva.
4. **Lanac interceptora:**
   * Interceptor može biti deo lanca interceptora, gde svaki interceptor u lancu može obraditi zahtev i proslediti ga sledećem interceptoru ili završiti obradu. Ovo omogućava modularnost i fleksibilnost u tome kako se obrada zahteva može proširiti ili prilagoditi.
5. **Podrška za različite aplikativne potrebe:**
   * Interceptori su dizajnirani da podrže različite aplikativne potrebe bez potrebe za direktnim menjanjem osnovne logike middleware-a. Na primer, interceptor može dodati sigurnosne provere, sprečiti dupliranje zahteva, ili dodati dodatne metapodatke za praćenje performansi.

**Primer primene interceptora**

U objektno-orijentisanim distribuiranim sistemima, interceptori se često koriste za realizaciju udaljenih poziva objekata. Na primer, middleware može imati interceptor koji hvata pozive lokalnog interfejsa aplikacije, pretvara ih u generičke pozive objekata koji se šalju preko mreže, i zatim ih interpretira na udaljenom računaru kako bi izvršio odgovarajuću logiku. Ovo omogućava aplikacijama da se ponašaju kao da pozivaju lokalne objekte, bez obzira na to gde su stvarno implementirani.

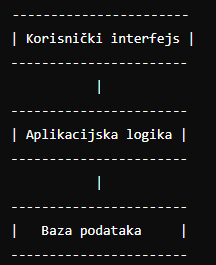


1. **Opisati sve modele separacije korisničkog interfejsa, aplikacije (logike) i baze podataka (nacrtati dijagram). Dati po primer za svaki model separacije.**

### Model baziran na monolitnoj arhitekturi

Monolitna arhitektura predstavlja tradicionalni pristup gde se svi slojevi (korisnički interfejs, aplikaciona logika i baza podataka) implementiraju kao jedna celina. Sve komponente dele isti prostor za izvršavanje i izvršavaju se unutar istog procesa.

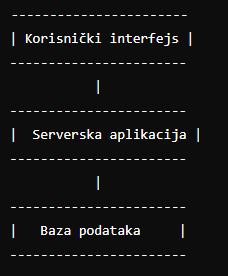
**Primer:** Na primer, jednostavna web aplikacija može imati HTML/CSS/JavaScript kao korisnički interfejs, serverski kod (npr. PHP, Java, Node.js) kao aplikacijsku logiku, i MySQL ili MongoDB kao bazu podataka.



### Model baziran na klijent-server arhitekturi

Klijent-server arhitektura razdvaja korisnički interfejs (klijent) od serverske aplikacijske logike i baze podataka (server). Klijent šalje zahteve serveru koji obrađuje te zahteve i odgovara sa odgovarajućim podacima.

**Primer:** Moderne web aplikacije koriste ovaj model, gde korisnički interfejs (browser) komunicira sa serverskom aplikacijskom logikom (backend) putem HTTP zahteva. Na primer, React kao klijentski interfejs, Node.js kao serverska aplikacija, i PostgreSQL kao baza podataka.

****

### Model baziran na mikroservisima

**Opis:** Mikroservisi razbijaju aplikaciju na više malih servisa, svaki sa svojom specifičnom funkcionalnošću. Svaki mikroservis može imati sopstveni korisnički interfejs, aplikacijsku logiku i bazu podataka. Komunikacija između mikroservisa često se ostvaruje preko API-ja.

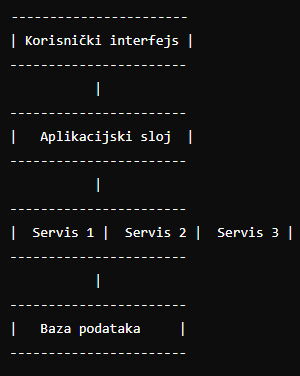
**Primer:** Amazon-ova arhitektura, gde svaki servis (npr. korpa, preporuke, naplata) može imati svoj frontend, backend i bazu podataka (npr. DynamoDB, Aurora).



### Model baziran na arhitekturi sa servisima

**Opis:** Arhitektura sa servisima slična je mikroservisnoj arhitekturi, ali servisi mogu biti veći i obuhvatiti više funkcionalnosti. Oni takođe mogu koristiti zajednički frontend ili aplikacijski sloj.

**Primer:** ERP sistem gde različiti moduli (npr. finansije, ljudski resursi, prodaja) imaju svoje backend servise i delimično deljeni frontend interfejs.



**L04 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Šta predstavlja proces, a šta nit?**

**Proces** je instanca programa u izvršavanju. To uključuje sam kod programa, kao i sve resurse koje program koristi tokom izvršavanja, kao što su otvoreni fajlovi, memorijski resursi, i podaci o stanju procesa. Procesi se koriste za izvršavanje aplikacija i obezbeđuju izolaciju između različitih aplikacija. Operativni sistem koristi procese za upravljanje i raspoređivanje izvršavanja programa.

**Nit** je lakši entitet unutar procesa, i predstavlja najmanju jedinicu raspoređivanja koja se može izvršavati. Niti unutar istog procesa dele isti prostor adresa i resurse, što omogućava brzu i efikasnu komunikaciju između niti.

* 1. **Šta predstavlja kontekst procesa?**

Operativni sistem kreira virtuelne procesore na kojima moguda se izvrsavaju razliciti programi. Da bi pratio ove virtuelne procesore, OS ima tabelu procesa koja sadrzi podatke za cuvanje vrednosti registara CPU-a, mapa memorije, privilegija I slicno. Zajedno ovi unosi formiraju kontekst procesa.

* 1. **Šta predstavlja interprocesna komunikacija (IPC) i kako se može postići?**

**Interprocesna komunikacija (IPC)** je skup metoda i mehanizama koji omogućavaju razmenu podataka i koordinaciju između različitih procesa koji se izvršavaju u istom operativnom sistemu. IPC je ključna komponenta u dizajnu distribuiranih sistema, kao i u tradicionalnim sistemima, jer omogućava procesima da dele informacije i sinhronizuju svoje aktivnosti.

**Soketi (Sockets)**

* **Mrežni soketi (Network Sockets):** Omogućavaju komunikaciju između procesa na različitim računarima preko mreže. Mogu koristiti različite protokole kao što su TCP i UDP.
  + **Unix primena:** socket(), bind(), listen(), accept(), connect(), send(), recv().
* **Unix domen soketi (Unix Domain Sockets):** Omogućavaju komunikaciju između procesa na istom računaru.
  + **Unix primena:** socket(), bind(), listen(), accept(), connect(), send(), recv().
  1. **Koje su prednosti implementacije niti umesto procesa?**

Niti omogućavaju izvršavanje blokirajućih sistemskih poziva bez blokiranja celog procesa. Ova karakteristika je posebno korisna u distribuiranim sistemima jer olakšava održavanje više logičkih veza istovremeno, čime se poboljšava komunikacija i performanse sistema.

* 1. **Opisati višenitnog klijenta na primeru veb pretraživača.**

Višenitni klijenti, kao što je veb pretraživač, koriste se efikasno za obradu i prikazivanje veb stranica koje sadrže različite elemente poput teksta, slika, ikona i drugih resursa. Ključna prednost višenitnih pristupa leži u sposobnosti da se sakriju komunikaciona kašnjenja i poboljša korisničko iskustvo.

Kada korisnik zahteva veb stranicu, veb pretraživač prvo preuzima glavni HTML dokument. Ovaj proces može biti blokirajući zbog uspostavljanja TCP/IP veze i čitanja podataka sa servera. Međutim, da bi se izbeglo dugotrajno čekanje, veb pretraživač može započeti prikazivanje dostupnog dela HTML-a odmah nakon što se deo dokumenta preuzme, umesto da čeka da se cela stranica učita. Ovo se postiže pokretanjem posebnih niti koje upravljaju preuzimanjem drugih elemenata stranice, kao što su slike, paralelno sa prikazivanjem već preuzetog sadržaja.

Korišćenje višenitnog pristupa omogućava veb pretraživaču da otvori više konekcija sa istim ili različitim serverima istovremeno. Na primer, može se uspostaviti više veza sa različitim replikama servera koji hostuju iste veb resurse. Ovo omogućava da se podaci paralelno preuzimaju sa više izvora, čime se ubrzava prikazivanje kompletnog veb dokumenta korisniku.

Ukratko, višenitni klijenti poput veb pretraživača efikasno upravljaju paralelnim zadacima preuzimanja i prikazivanja veb stranica, čime se smanjuje percepcija kašnjenja i poboljšava korisničko iskustvo pri interakciji sa vebom.

* 1. **Opisati višenitni server na primeru dispečerske i radne niti.**

Višenitni serveri predstavljaju ključan koncept u distribuiranim sistemima jer omogućavaju efikasno upravljanje višestrukim zahtevima od klijenata, poboljšavajući istovremeno performanse i skalabilnost sistema. Evo kako funkcionišu višenitni serveri, koristeći primer dispečerske i radne niti:

1. **Organizacija servera**: Višenitni server se organizuje sa dve glavne vrste niti - dispečerskom i radnim nitima. Dispečerska nit ima ulogu da čeka na dolazne zahteve od klijenata. Kada zahtev stigne na server, dispečer ga uzima i prosleđuje jednoj od slobodnih radnih niti na obradu.
2. **Radne niti**: Radne niti su zadužene za izvršavanje zahteva koji su im prosleđeni od strane dispečerske niti. Svaka radna nit može obavljati blokirajuće operacije, kao što je čitanje ili pisanje podataka na disk. Kada je radna nit suspendovana zbog čekanja na disk, dispečerska nit može izabrati drugu slobodnu radnu nit kako bi nastavila obradu novih dolaznih zahteva. Ovo omogućava da se procesiranje zahteva odvija paralelno, umesto da se čeka na završetak svakog pojedinačnog zahteva pre nego što se nastavi sa sledećim.
3. **Prednosti višenitnog pristupa**: Višenitni serveri donose značajne prednosti u performansama i responzivnosti. Nasuprot jednonitnim serverima koji često moraju da čekaju na završetak operacija nad diskom pre nego što mogu nastaviti sa obradom drugih zahteva, višenitni serveri omogućavaju istovremeno izvršavanje više zahteva. Ovo je ključno za postizanje visokih performansi čak i na višejezgarnim ili višeprocesorskim sistemima, jer svaka nit može biti dodeljena različitom procesoru ili jezgru radi paralelne obrade.
4. **Model asinhronog programiranja**: Pored višenitnog pristupa, alternativa može biti korišćenje modela asinhronog programiranja. Umesto blokirajućih operacija nad diskom, server može zakazati asinhronu operaciju čitanja ili pisanja podataka, što omogućava niti da ne blokiraju svoje resurse dok čekaju na završetak operacije. Ovaj pristup zahteva posebno rukovanje stanjem za svaki zahtev kako bi se pravilno upravljalo tokom asinhronih operacija.
5. **Skalabilnost**: Višenitni pristup je ključan za skalabilnost servera u distribuiranim sistemima, jer omogućava efikasno upravljanje velikim brojem istovremenih zahteva. Kako se opterećenje povećava, može se dodati više radnih niti ili se prilagoditi strategija raspoređivanja zahteva, čime se održava performanse sistema.

**L05 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* + 1. **Šta podrazumeva virtuelizacija?**

Virtuelizacija podrazumeva apstrakciju računarskih resursa i omogućava deljenje jednog fizičkog računarskog sistema na više virtuelnih okruženja, poznatih kao virtuelne mašine (VM). Ovaj koncept omogućava da se resursi kao što su procesorsko vreme, memorija, skladišni prostor i mrežni resursi podele i alokiraju različitim VM-ovima, čime se povećava efikasnost korišćenja hardverskih resursa.

* + 1. **Šta predstavlja *host*, a šta *guest*mašina? Dati primer.**

Host mašina je osnovni računar na kojem se vrši virtuelizacija. To je fizički računar koji poseduje hardverske resurse poput procesora, memorije, diskova i mrežnih interfejsa. Host mašina je odgovorna za izvršavanje hipervizora ili virtuelizacijskog softvera koji omogućava kreiranje, upravljanje i izvršavanje virtuelnih mašina.

Guest mašina, s druge strane, je virtuelna mašina koja radi unutar host mašine. To je emulirano ili virtualizovano okruženje koje se ponaša kao samostalan računar sa svojim operativnim sistemom, aplikacijama i korisnicima. Guest mašina koristi resurse host mašine, kao što su CPU, memorija, diskovi i mreža, ali to radi putem slojeva virtualizacije koji su implementirani od strane hipervizora.

**Primer:**

* **Host mašina:** Neka je host mašina fizički računar koji koristi operativni sistem Windows 10 Pro. Na ovoj host mašini instaliran je VMware Workstation, koji je popularan hipervizor za desktop računare. VMware Workstation omogućava host mašini da kreira, upravlja i izvršava virtuelne mašine.
* **Guest mašina:** Kreiramo virtuelnu mašinu (guest) unutar VMware Workstation-a. Za potrebe primera, guest mašina može biti instaliran Ubuntu Linux operativni sistem. Ova virtuelna mašina koristi resurse host mašine kao što su CPU, RAM, disk i mrežni interfejs, ali radi u izolovanom okruženju unutar VMware Workstation-a.
  + 1. **Koja je razlika između emulacije i virtuelizacije?**

**Emulacija:** Emulacija podrazumeva simulaciju hardverske arhitekture na kojoj radi guest mašina. To može biti sporije jer zahteva dodatne slojeve za prevođenje i emulaciju instrukcija.

**Virtualizacija:** Virtualizacija izoluje i deli resurse host mašine između virtuelnih mašina. Ovo može biti efikasnije jer se koristi postojeća arhitektura hosta, što omogućava bolje performanse.

* + 1. **Ukratko opisati oblike virtuelizacije.**

Ima previse mrzi me

* + 1. **Opisati virtualizaciju servera i koje su glavne prednosti iste.**

Virtualizacija servera omogućava jednom fizičkom računaru da pokreće više virtuelnih mašina (VMs), pri čemu svaka VM može imati sopstveni operativni sistem i aplikacije. Ova tehnologija transformiše način na koji se serveri koriste i administriraju u IT infrastrukturi. Evo nekoliko ključnih aspekata i prednosti virtualizacije servera:

1. **Konsolidacija resursa:** Umesto da svaka aplikacija ili usluga zahteva zaseban fizički server, virtualizacija omogućava više aplikacija da se izvršavaju na istom fizičkom hardveru u vidu različitih VM-ova. Ovo smanjuje potrebu za više fizičkih servera, što vodi smanjenju troškova nabavke hardvera, prostora u data centru, potrošnje energije i administrativnih troškova.
2. **Povećana iskorišćenost resursa:** Tradicionalni fizički serveri često imaju nisku iskorišćenost resursa, jer je svaki posvećen samo jednoj aplikaciji. Virtualizacija omogućava da se više resursa (CPU, RAM, skladište) deli između VM-ova u skladu sa njihovim zahtevima. Ovo može povećati iskorišćenost hardvera sa tipičnih 10-15% na preko 80%, što je ekonomičnije i efikasnije.
3. **Fleksibilnost i agilnost:** Kreiranje, kopiranje, premestanje i brisanje virtuelnih mašina je brzo i jednostavno. Ovo omogućava IT timovima da brzo odgovore na promenjene zahteve i optimizuju resurse prema trenutnim potrebama aplikacija. Takođe, omogućava testiranje novih konfiguracija ili aplikacija bez uticaja na proizvodnu infrastrukturu.
4. **Visoka dostupnost i oporavak od katastrofe:** Virtuelizacija olakšava implementaciju visokodostupnih rešenja kao što su klasteri, replikacija VM-ova i brzo obnavljanje (recovery) u slučaju havarije. Automatizovani alati za upravljanje omogućavaju da se VM-ovi automatski migriraju ili restartuju u slučaju problema sa fizičkim hardverom.
5. **Optimizacija resursa:** Virtualizacija omogućava podešavanje resursa za svaku virtuelnu mašinu prema specifičnim zahtevima aplikacija (CPU, RAM, skladište). Ovo dovodi do bolje optimizacije resursa i efikasnijeg korišćenja kapaciteta servera.
6. **Zaštita okoline:** Smanjenje broja fizičkih servera u data centru smanjuje potrošnju električne energije, potrebu za hlađenjem i emisiju ugljen-dioksida, što doprinosi smanjenju ekološkog uticaja organizacije.
   * 1. **Opisati virtuelizaciju računarske mreže, koji su slučajevi korišćenja i koje su prednosti ovog tipa virtuelizacije.**

Virtuelizacija računarske mreže predstavlja tehniku koja omogućava deljenje i upravljanje mrežnim resursima na način koji je nezavisan od fizičke mrežne infrastrukture. Ova tehnologija se primenjuje kako bi se omogućilo efikasnije upravljanje mrežnim resursima, pojednostavila administracija, i poboljšala fleksibilnost i skalabilnost mrežnih sistema. Postoje dva glavna tipa virtuelizacije mreže: eksterna i interna.

**Eksterna virtuelizacija mreže**

Eksterna virtuelizacija mreže se primenjuje na nivou šire mrežne infrastrukture, obično u okviru velikih korporativnih ili data centar mreža. Glavni slučajevi korišćenja uključuju:

1. **Virtualne LAN mreže (VLANs):** VLAN tehnologija omogućava da se jedna fizička mreža podeli na više logičkih segmenata. Ovo omogućava administratorima da grupišu uređaje u različite virtuelne mreže na osnovu njihovih funkcionalnih potreba, bez obzira na njihovu fizičku lokaciju u mreži. Na primer, različiti departmani u kompaniji mogu biti smješteni na istoj fizičkoj mreži, ali na različitim VLAN-ovima radi boljeg upravljanja i sigurnosti.
2. **Virtualne privatne mreže (VPN):** VPN tehnologija omogućava kreiranje sigurnih mrežnih veza preko javnih ili nezaštićenih mreža kao što je internet. Ovo je posebno korisno za povezivanje udaljenih lokacija ili za pristup mrežnim resursima van tradicionalne mrežne infrastrukture.
3. **Software Defined Networking (SDN):** SDN je arhitektura koja omogućava centralizovano upravljanje mrežom putem programabilnih kontrolnih ravnina, što omogućava veću fleksibilnost, efikasnost i skalabilnost u upravljanju mrežnim resursima.

**Interna virtuelizacija mreže**

Interna virtuelizacija mreže se primenjuje na nivou pojedinačnog računara ili servera. Ovde se koristi za kreiranje virtuelnih mreža i interfejsa unutar jednog fizičkog servera ili među više virtuelnih mašina na istom fizičkom hostu. Glavni slučajevi korišćenja uključuju:

1. **Kontejnerizacija:** Koristi se za izolaciju i upravljanje aplikacijama i njihovim zavisnostima unutar virtuelnih kontejnera. Kontejneri dele resurse operativnog sistema, što omogućava efikasniju upotrebu resursa i brže pokretanje aplikacija u poređenju sa tradicionalnim virtuelnim mašinama.
2. **Virtuelne mreže na nivou operativnog sistema:** Ovo je korisno za simulaciju i testiranje mrežnih postavki, kao i za razvoj i implementaciju kompleksnih mrežnih arhitektura bez potrebe za fizičkom infrastrukturom.

**Prednosti virtuelizacije računarske mreže**

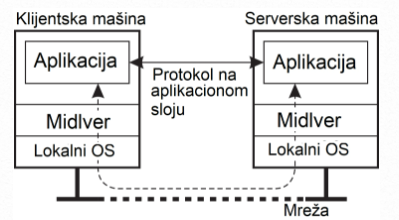
* **Fleksibilnost:** Omogućava dinamičko prilagođavanje mrežnih resursa prema potrebama aplikacija i korisnika.
* **Efikasnost u korišćenju resursa:** Povećava iskorišćenost mrežnih resursa kroz bolje upravljanje i deljenje.
* **Skalabilnost:** Omogućava jednostavno skaliranje mrežnih resursa prema rastu potreba organizacije.
* **Sigurnost:** Pruža veći nivo kontrole i sigurnosti, posebno kroz tehnologije kao što su VLANs i VPNs.
* **Jednostavnost upravljanja:** Centralizovano upravljanje i automatske konfiguracije olakšavaju administraciju mreže.

**L06 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Šta predstavlja umreženi korisnički interfejs. Nacrtati dijagram.**

Umreženi korisnički interfejsi (networked user interfaces) su sistemi gde se većina obrade i skladištenja podataka obavlja na udaljenim serverima, dok korisničke mašine (kao što su računari, pametni telefoni ili tableti) služe kao terminali za interakciju sa tim serverima. Klijentske mašine pružaju sredstva korisnicima za pristup i interakciju sa udaljenim resursima, a svi podaci i većina funkcionalnosti se obrađuju i čuvaju na serveru. Postoje dva glavna načina podrške za ovu interakciju:

1. **Klijent sa pandanom za udaljenu uslugu:** Svaka usluga na serveru ima odgovarajući pandan na klijentskoj mašini. Na primer, aplikacija za kalendar na pametnom telefonu sinhronizuje podatke sa udaljenim serverom koji sadrži deljeni kalendar. Ova sinhronizacija se obavlja putem aplikacionog protokola.
2. **Direktan pristup udaljenim uslugama:** Klijentska mašina funkcioniše kao terminal koji omogućava direktan pristup udaljenim uslugama putem korisničkog interfejsa, bez lokalnog skladištenja podataka. Na ovaj način, sve funkcionalnosti i podaci se nalaze na serveru, a klijentska mašina služi samo za prikazivanje informacija i interakciju.



* 1. **Opisati umreženi korisnički interfejs na primeru X11 (*X Window System)*.**

X Window System, obično poznat kao X ili X11, je mrežni korisnički interfejs koji omogućava korisnicima da kontrolišu bit-mapirane terminale koji uključuju monitor, tastaturu, i uređaje za pokazivanje kao što je miš. X11 je jedan od najstarijih, ali i dalje široko korišćenih mrežnih korisničkih interfejsa. Koristi se za različite uređaje, od tradicionalnih desktop računara do modernih tableta i pametnih telefona sa ekranima osetljivim na dodir.

X Window System koristi klijentsko-server model gde X kernel funkcioniše kao server, dok aplikacije koje koriste Xlib funkcionišu kao klijenti. Interesantan aspekt ovog modela je da X kernel i X aplikacije ne moraju biti smešteni na istom računaru. Aplikacije mogu biti pokrenute na jednom računaru i koristiti X protokol za komunikaciju sa X kernelom na drugom računaru. Ovo omogućava fleksibilnost i omogućava korisnicima da pristupaju grafičkim interfejsima na udaljenim sistemima.

Korisnik na svom lokalnom računaru (terminalu) može koristiti X Window System da pokrene grafičku aplikaciju koja se zapravo izvršava na udaljenom serveru. Na primer, korisnik može pokrenuti softver za grafički dizajn na moćnom serveru, dok se grafički interfejs prikazuje na lokalnom računaru. X protokol omogućava komunikaciju između lokalnog terminala i udaljenog servera, prenoseći događaje poput pokreta miša i unosa sa tastature.

* 1. **Opisati pojam tankog klijenta. Kada se ovakvi sistemi koriste?**

**Tanki klijent** (engl. thin client) je uređaj ili softverski program koji zavisi od servera za većinu svoje obrade, skladištenja i upravljanja podacima. Ovaj koncept se često koristi u umreženim okruženjima gde se obrade i skladištenje podataka obavljaju centralizovano na serveru, dok klijentski uređaji služe kao terminali za pristup tim resursima. Tanki klijent obavlja minimalne lokalne obrade i oslanja se na udaljene servere za glavne funkcionalnosti.

**Primeri Tankih Klijenata**

1. **Zero Client:** Vrsta tankog klijenta koja nema lokalni operativni sistem ili skladište. Sve operacije se obavljaju na serveru.
2. **VDI (Virtual Desktop Infrastructure):** Korisnički desktop se virtualizuje i pokreće na centralizovanom serveru. Korisnici pristupaju svojim desktopima putem tankih klijenata.
3. **Web Browser:** Web browser može funkcionisati kao tanki klijent kada koristi web aplikacije koje obavljaju obradu na serveru, kao što su Google Docs ili Office 365.

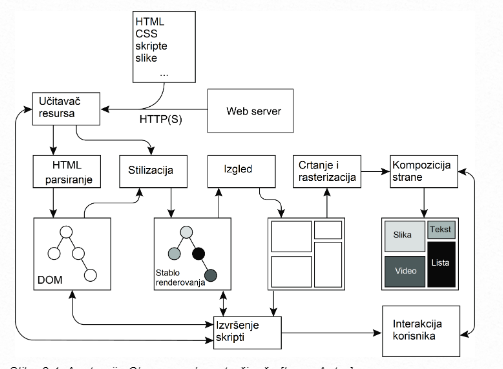
Tanki klijenti su važan deo modernih IT infrastruktura, posebno u okruženjima gde je centralizovana kontrola, sigurnost i efikasnost ključna. Omogućavaju efikasno korišćenje resursa, smanjuju troškove i poboljšavaju upravljivost sistema.

Tanki klijenti se obično koriste u različitim situacijama i okruženjima gde centralizovano upravljanje, sigurnost i efikasno korišćenje resursa imaju ključnu važnost.

1. **Opisati pojam virtuelnog desktop okruženja.**

**Virtuelno desktop okruženje** (engl. Virtual Desktop Environment - VDE) je tehnologija koja omogućava korisnicima pristup desktopu i aplikacijama koje se ne izvršavaju lokalno na njihovim računarima, već na udaljenim serverima u oblaku. Ovaj koncept je postao popularan sa rastom računarstva u oblaku i omogućava korisnicima pristup svojim radnim okruženjima sa bilo kojeg uređaja povezanog na internet.

1. **Nacrtati dijagram komponenti savremenog veb pretraživača poput Google Chrome-a. Opisati ulogu:**
   1. **Resource loader-a**
   2. **DOM-a**
   3. **Render Tree-a**

****

**Resource Loader** je ključna komponenta Chrome veb pretraživača koja je odgovorna za preuzimanje sadržaja sa web servera. Kada korisnik unese URL ili klikne na link, resource loader pokreće proces učitavanja resursa. Glavne funkcije resource loader-a uključuju:

1. **Preuzimanje HTML datoteke:** Resource loader prvo preuzima osnovnu HTML datoteku sa servera.
2. **Identifikacija resursa:** Parsiranjem HTML datoteke identifikuje dodatne resurse kao što su CSS, JavaScript, slike, video zapisi itd., koje takođe treba preuzeti.
3. **Paralelno preuzimanje:** Da bi ubrzao učitavanje stranice, resource loader može paralelno preuzimati više resursa koristeći više niti.
4. **Praćenje zavisnosti:** Resource loader upravlja zavisnostima između različitih resursa i osigurava da se resursi preuzimaju i izvršavaju u ispravnom redosledu.

**DOM (Document Object Model)**

**Document Object Model (DOM)** je strukturni model dokumenta koji predstavlja HTML ili XML datoteku kao hijerarhijsko stablo. Svaki čvor u stablu predstavlja deo dokumenta, kao što su elementi, atributi i tekstualni sadržaji. DOM omogućava skriptama da dinamički pristupaju i menjaju sadržaj, strukturu i stil dokumenta. Glavne funkcije DOM-a uključuju:

1. **Struktura dokumenta:** DOM predstavlja dokument kao stablo čvorova, gde svaki čvor odgovara određenom delu HTML-a.
2. **Manipulacija sadržajem:** DOM omogućava dinamičke promene sadržaja stranice pomoću JavaScript-a, uključujući dodavanje, uklanjanje ili promenu elemenata.
3. **Interaktivnost:** DOM omogućava korisničku interakciju sa stranom putem događaja kao što su klikovi, unosi na tastaturi i drugi korisnički unosi.
4. **Podrška za stilizaciju:** Informacije iz CSS-a se dodaju DOM-u kako bi se odredile stilizacije elemenata.

**Render Tree**

**Render Tree** je struktura koja kombinuje informacije iz DOM-a i CSS-a kako bi se odredio izgled stranice na ekranu. Render tree sadrži vizuelne elemente koji treba da budu prikazani, zajedno sa stilskim informacijama. Glavne funkcije render tree-a uključuju:

1. **Kreiranje čvorova render tree-a:** Prilikom parsiranja DOM-a i CSS-a, pretraživač stvara čvorove render tree-a koji sadrže stilizovane elemente.
2. **Određivanje izgleda:** Render tree određuje tačne pozicije i dimenzije svakog elementa na stranici, uključujući raspored teksta, slika, okvira itd.
3. **Proračuni geometrije:** Render tree uključuje proračune geometrijskih regiona, poput prelomljenja linija i pozicioniranja elemenata, uzimajući u obzir karakteristike fonta i jezičke specifičnosti.
4. **Priprema za rasterizaciju:** Nakon što je render tree kreiran i stilizovan, on se prosleđuje komponenti za slikanje koja konstruiše operacije slikanja potrebne za prikaz stranice na ekranu.
5. **Opisati tok podataka kroz komponente veb pretraživača od učitavanja datoteka do prikaza na ekranu.**

Tok podataka kroz komponente veb pretraživača može se sumirati kroz sledeće korake:

1. Učitavanje resursa (Resource Loader)
2. Parsiranje HTML-a (HTML Parser)
3. Izgradnja DOM-a (DOM Construction)
4. Parsiranje CSS-a (CSS Parser)
5. Konstrukcija Render Tree-a (Render Tree Construction)
6. Layout (izračunavanje rasporeda)
7. Painting (slikanje)
8. Rasterizacija (Rasterization)
9. Kompozitovanje (Compositing)
10. Prikaz na ekranu (Display)
11. **Šta predstavlja konkurentni, a šta iterativni server?**

**Iterativni server** je tip servera koji obrađuje zahteve jedan po jedan. Ovaj server čeka da primi zahtev od klijenta, obrađuje taj zahtev, vraća odgovor klijentu, a zatim prelazi na sledeći zahtev. Ključna karakteristika iterativnog servera je da on ne prihvata novi zahtev dok se trenutni zahtev ne obradi u potpunosti. Ovaj pristup je jednostavan i koristi se kada zahtevi stižu relativno retko ili kada obrada zahteva traje veoma kratko. Primeri iterativnih servera uključuju jednostavne HTTP servere ili FTP servere.

**Konkurentni server** je dizajniran da može rukovati sa više zahteva istovremeno. Umesto da obrađuje zahteve jedan po jedan, konkurentni server prihvata novi zahtev i zatim prosleđuje taj zahtev zasebnoj niti ili novom procesu. Na taj način, server može odmah prihvatiti sledeći zahtev dok se prethodni zahtev obrađuje u pozadini. Postoje dva glavna pristupa implementaciji konkurentnih servera:

1. **Višenitni (multithreaded) server:**
   * Kreira novu nit za svaki novi zahtev.
   * Sve niti dele isti memorijski prostor, što omogućava efikasnu komunikaciju između njih.
2. **Višeporcesni (multiprocessing) server:**
   * Kreira novi proces za svaki novi zahtev.
   * Svaki proces ima svoj memorijski prostor, što pruža bolju izolaciju, ali može povećati troškove kreiranja i upravljanja procesima.
3. **Opisati dva pristupa zahteva klijenta ka serveru kada klijent ne zna endpoint.**

1. Dinamički dodeljene krajnje tačke uz pomoć daemona

**Opis:**

* **Krajnje tačke (endpoints):** Servisi koji ne koriste unapred dodeljene, fiksne krajnje tačke mogu koristiti dinamičke krajnje tačke koje im dodeljuje lokalni operativni sistem.
* **Daemon proces:** Daemon je poseban proces koji se pokreće na svakoj mašini gde su implementirani serveri. On sluša na dobro poznatoj krajnjoj tački i prati trenutne krajnje tačke svakog lokalno postavljenog servisa.

**Kako funkcioniše:**

1. **Klijent kontaktira daemona:** Klijent prvo šalje zahtev daemonu putem dobro poznate krajnje tačke.
2. **Daemona odgovara:** Daemon proverava trenutne krajnje tačke servisa i odgovara klijentu sa tačnom krajnjom tačkom specifičnog servera kojeg klijent traži.
3. **Klijent kontaktira server:** Klijent koristi dobijenu krajnju tačku da direktno kontaktira željeni server.

2. Superserver (npr. inetd daemon u Unix-u)

**Opis:**

* **Superserver:** Umesto da svaka usluga ima svoj zaseban pasivni proces, superserver sluša na svim krajnjim tačkama povezanim sa određenim servisima.

**Kako funkcioniše:**

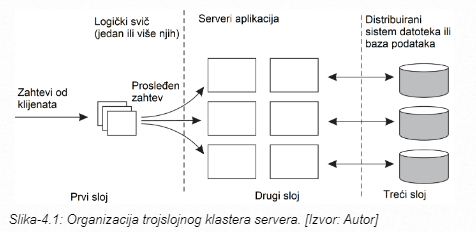
1. **Superserver sluša:** Superserver (kao što je inetd u Unix-u) sluša na nekoliko dobro poznatih portova za razne servise.
2. **Klijent kontaktira superserver:** Kada klijent pošalje zahtev za određeni servis, taj zahtev prvo dolazi do superservera.
3. **Superserver delegira:** Superserver zatim klonira (pokreće) novi proces ili nit koja će obraditi taj specifični zahtev.
4. **Obrada zahteva:** Novi proces ili nit obrađuje zahtev klijenta i odgovara klijentu. Nakon završetka, taj proces ili nit se zatvara.
5. **Šta predstavljaju out-of-band podaci i koja je njihova uloga? Dati primer.**

Out-of-Band (OOB) podaci su specijalna vrsta podataka koji se prenose odvojeno od glavnog toka podataka, omogućavajući prioritetnu obradu određenih informacija. Oni služe kao kontrolni mehanizam između klijenta i servera, koristeći se za specifične funkcije kao što su prekidanje aktivnosti, hitne notifikacije ili slanje kontrolnih komandi. Na primer, kada korisnik otprema veliki fajl na FTP server i želi da prekine transfer, može poslati OOB podatke sa zahtevom za prekidanje trenutne aktivnosti. Server, primajući te podatke, odmah prekida transfer i preduzima odgovarajuće akcije, poput otkazivanja transfera i oslobađanja resursa.

OOB podaci se takođe koriste za hitne notifikacije, gde klijent može brzo obavestiti server o promeni stanja, omogućavajući serveru da prilagodi svoje ponašanje. Takođe, administrativne komande, kao što su restartovanje ili resetovanje servera, mogu biti poslate kao OOB podaci, čime server odmah prepoznaje i izvršava te komande bez čekanja na redovni tok podataka.

Funkcionisanje OOB podataka može se ostvariti na dva načina. Prvi način je da server osluškuje na posebnoj krajnjoj tački namenjenoj isključivo za OOB podatke. Kada klijent pošalje OOB podatke na tu tačku, server ih odmah prepoznaje i obrađuje pre regularnih podataka. Drugi način je slanje OOB podataka preko iste veze koja se koristi za regularne podatke, ali sa zastavicom za hitne podatke, kao što je to moguće u TCP protokolu. Kada server primi hitne podatke, generiše se prekid (na primer, signal u Unix sistemima), i server ih odmah prepoznaje i obrađuje.

1. **Nacrtati dijagram klastera servera podeljenih u tri sloja i ukratko opisati svaki sloj.**

****

Klaster servera može se logički organizovati u tri nivoa: nivo rutiranja zahteva, nivo aplikacione obrade, i nivo obrade podataka.

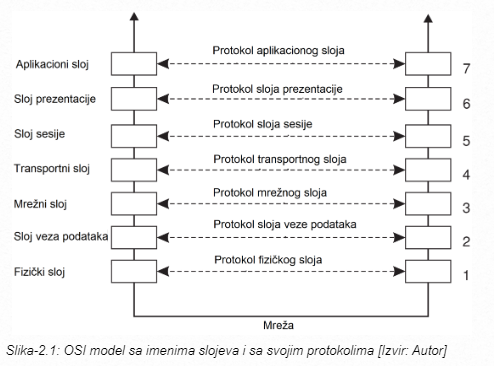
Prvi nivo, nivo rutiranja zahteva, sastoji se od logičkog prekidača koji služi za distribuciju dolaznih zahteva klijenata ka odgovarajućim serverima u klasteru. Ovaj prekidač može biti transportni prekidač, koji prihvata dolazne TCP zahteve za konekciju i prosleđuje ih jednom od servera u klasteru. Alternativno, može biti Web server koji prihvata dolazne HTTP zahteve, prosleđuje ih aplikacionim serverima za dalju obradu, sakuplja rezultate od tih servera i vraća HTTP odgovore klijentima. Ovaj nivo funkcioniše kao prvi kontakt tačka između klijenata i klastera, omogućavajući raspodelu zahteva i balansiranje opterećenja.

Drugi nivo, nivo aplikacione obrade, sastoji se od servera koji su posvećeni izvršavanju specifičnih aplikacija. Ovi serveri mogu biti serveri visokih performansi namenjeni za računarske operacije, ili serveri srednje klase u slučajevima gde računarska snaga nije usko grlo. Ovaj nivo je odgovoran za izvršavanje aplikacionih logika, obradu poslovnih pravila, i druge zadatke vezane za primenu specifičnih poslovnih potreba. U višeslojnim arhitekturama, aplikacioni serveri igraju ključnu ulogu u obradi podataka pre nego što se rezultati vrate klijentima ili proslede na dalju obradu.

Treći nivo, nivo obrade podataka, sastoji se od servera za skladištenje i obradu podataka, kao što su serveri za fajlove i baze podataka. Ovi serveri su specijalizovani za upravljanje velikim količinama podataka i omogućavanje brzog pristupa tim podacima za aplikacione servere. Skladišni serveri pružaju pouzdano i efikasno čuvanje podataka, dok serverske baze podataka omogućavaju kompleksne upite i transakcije. Ovaj nivo je ključan za pružanje podrške aplikacionim serverima, obezbeđujući neophodne resurse za obradu i skladištenje podataka, čime se omogućava nesmetano funkcionisanje celokupnog klastera servera.

**L07 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Nabrojati sve slojeve OSI referentnog modela i opisati tok enkapsulacije podataka od aplikativnog do fizičkog sloja.**
* Fizički sloj: Bavi se standardizacijom načina kako su dva računara povezana i kako se 0 i 1 reprezentuju.
* Sloj veze podataka: Pruža sredstva za detektovanje i možda ispravljanje grešaka u prenosu, kao i protokole za održavanje sinhronizacije između pošiljaoca i primaoca.
* Mrežni sloj: Sadrži protokole za rutiranje poruke kroz računarsku mrežu, kao i protokole za upravljanje zagušenjem.
* Transportni sloj: Glavni sadrži protokole koji direktno podržavaju aplikacije, kao što su oni koji uspostavljaju pouzdanu komunikaciju ili podržavaju realno vreme tokom prenosa podataka.
* Sloj sesije: Pruža podršku za sesije između aplikacija.
* Sloj prezentacije: Propisuje kako se podaci predstavljaju na način koji je nezavisan od računara na kojima se izvršavaju aplikacije koje komuniciraju.
* Aplikacioni sloj: U osnovi, sve ostalo: protokoli za e-poštu, protokoli za pristup vebu, protokoli za prenos datoteka, i tako dalje.

****

Enkapsulacija podataka u OSI referentnom modelu uključuje transformaciju podataka kroz različite slojeve mrežnog modela, od aplikativnog do fizičkog sloja. Tok enkapsulacije može se opisati na sledeći način:

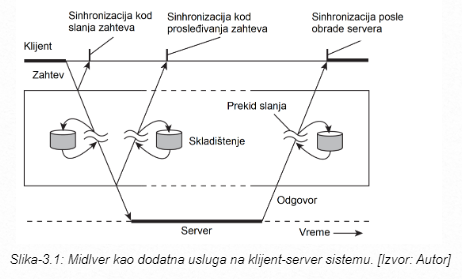
1. **Aplikativni sloj (Application Layer)**: Ovo je najviši sloj OSI modela, gde korisničke aplikacije komuniciraju sa mrežom. Podaci koje korisnik unosi (npr. HTTP zahtevi ili e-mail poruke) generišu se i prosljeđuju ka nižem sloju.
2. **Prezentacioni sloj (Presentation Layer)**: Ovaj sloj se brine o pravilnoj interpretaciji podataka između različitih sistema. Funkcije prezentacionog sloja uključuju enkripciju, dekripciju, kompresiju i prevođenje podataka u format koji niži slojevi mogu razumeti. Podaci se enkapsuliraju dodavanjem odgovarajućih informacija za prevođenje i kompresiju.
3. **Sesijski sloj (Session Layer)**: Sesijski sloj upravlja komunikacionim sesijama između aplikacija. Ovaj sloj uspostavlja, održava i raskida veze (sesije) između aplikacija. Dodaje informacije o sesiji koja omogućava održavanje sesije u toku cele komunikacije.
4. **Transportni sloj (Transport Layer)**: Transportni sloj osigurava pouzdanu isporuku podataka između aplikacija. Koristi protokole kao što su TCP (Transmission Control Protocol) za pouzdanu komunikaciju ili UDP (User Datagram Protocol) za manje pouzdane ali brže prenose. Enkapsulacija na ovom sloju uključuje dodavanje zaglavlja koja sadrže informacije kao što su brojevi portova, kontrolne informacije, i broj sekvence za praćenje paketa.
5. **Mrežni sloj (Network Layer)**: Mrežni sloj upravlja rutiranjem podataka kroz mrežu. Koristi IP adrese za identifikaciju izvora i odredišta podataka. Na ovom sloju se podacima dodaje zaglavlje koje sadrži informacije o IP adresi, kao i kontrolne informacije za rutiranje.
6. **Sloj podatkovne veze (Data Link Layer)**: Ovaj sloj je odgovoran za pouzdanu komunikaciju između čvorova na istom fizičkom segmentu mreže. Enkapsulacija uključuje dodavanje zaglavlja (i ponekad podnožja) koje sadrži MAC (Media Access Control) adrese, kontrolne informacije za otkrivanje grešaka, i kontrolu pristupa medijumu.
7. **Fizički sloj (Physical Layer)**: Fizički sloj se bavi stvarnim prenosom bitova preko fizičkog medijuma, kao što su bakarne žice, optički kablovi, ili radio talasi. Na ovom sloju se podaci pretvaraju u električne, optičke, ili radio signale i šalju preko mrežnog medijuma. Nema dodatne enkapsulacije na ovom sloju jer se prenose sirovi bitovi.

**2. Ukoliko koristimo model sa četiri sloja, dati primere protokola na midlver sloju**

Ako koristimo model sa četiri sloja, midlver može biti predstavljen kao zaseban sloj između aplikacionog i transportnog sloja. U ovom modelu, midlver sloj obuhvata protokole i usluge koje pružaju opšte funkcionalnosti nezavisne od specifičnih aplikacija. Evo nekoliko primera protokola i usluga koje pripadaju midlver sloju:

1. **DNS (Domain Name System)**: DNS je distribuirana usluga koja se koristi za mapiranje simboličkih imena (kao što su veb adrese) na IP adrese. DNS protokol se može smatrati delom midlvera jer pruža osnovnu uslugu koja je nezavisna od specifičnih aplikacija.
2. **LDAP (Lightweight Directory Access Protocol)**: LDAP se koristi za pristup i održavanje distribuiranih direktorijuma informacija, kao što su korisnički podaci i resursi mreže. LDAP pruža opštu uslugu autentifikacije i autorizacije, koja se može integrisati u različite aplikacije.
3. **Kerberos**: Kerberos je mrežni protokol za autentifikaciju koji koristi tajne ključeve za omogućavanje sigurnog komuniciranja između korisnika i servisa. Pruža opštu uslugu autentifikacije i stoga pripada midlver sloju.
4. **OAuth**: OAuth je protokol za autorizaciju koji omogućava trećim stranama da pristupe resursima korisnika bez deljenja lozinki. Koristi se za sigurno delegiranje pristupa, što ga čini opštom uslugom midlvera.
5. **SOAP (Simple Object Access Protocol)**: SOAP je protokol za razmenu informacija u decentralizovanim i distribuiranim okruženjima. Koristi XML za formatiranje poruka i može se koristiti za komunikaciju između različitih aplikacija i servisa, pružajući opštu uslugu komunikacije.
6. **REST (Representational State Transfer)**: REST je arhitektonski stil koji koristi HTTP protokol za komunikaciju između klijenata i servera. Pruža opštu uslugu za izgradnju web servisa, što ga svrstava u midlver sloj.
7. **JMS (Java Message Service)**: JMS je API koji omogućava komunikaciju između aplikacija putem poruka. Pruža opštu uslugu razmene poruka u distribuiranim sistemima, što ga čini delom midlvera.
8. **Distributed Commit Protocols (npr. Two-Phase Commit)**: Ovi protokoli osiguravaju da u distribuiranom sistemu, sve transakcije ili budu uspešno izvršene ili uopšte ne budu izvršene, pružajući uslugu atomičnosti. Oni su opšti i nezavisni od specifičnih aplikacija, pripadajući midlver sloju.
9. **Distributed Locking Protocols**: Ovi protokoli omogućavaju kontrolu pristupa resursima u distribuiranim sistemima, sprečavajući simultani pristup od strane više procesa. Kao opšta usluga za sinhronizaciju, ovi protokoli su deo midlvera.
10. **Opisati model tzv. uporne (en. persistant) komunikacije između klijenta i servera. Nacrtati i opisati dijagram.**

Uporna (persistant) komunikacija je model u kojem se poruka koja je poslata za prenos čuva od strane komunikacionog sistema sve dok se ne dostavi primaocu. To znači da sistem garantuje isporuku poruke čak i ako primalac trenutno nije aktivan ili ako postoje privremeni prekidi u mreži. Ovaj model je koristan u aplikacijama gde je važno da se poruke ne izgube, kao što su sistemi za dostavu e-pošte, sistemi za upravljanje porukama i drugi slični sistemi.



1. **Opisati model tzv. prolazne (en. transient) komunikacije između klijenta i servera. Nacrtati i opisati dijagram.**

Prolazna (transient) komunikacija je model u kojem se poruka čuva u komunikacionom sistemu samo dok su aplikacije za slanje i primanje aktivne. U ovom modelu, ako komunikacioni sistem ne može da dostavi poruku zbog privremenog prekida ili zato što primalac trenutno nije aktivan, poruka će biti odbačena. Ovaj model je često korišćen u aplikacijama gde je brzina važnija od pouzdanosti, kao što su usluge u realnom vremenu ili direktne mrežne aplikacije.

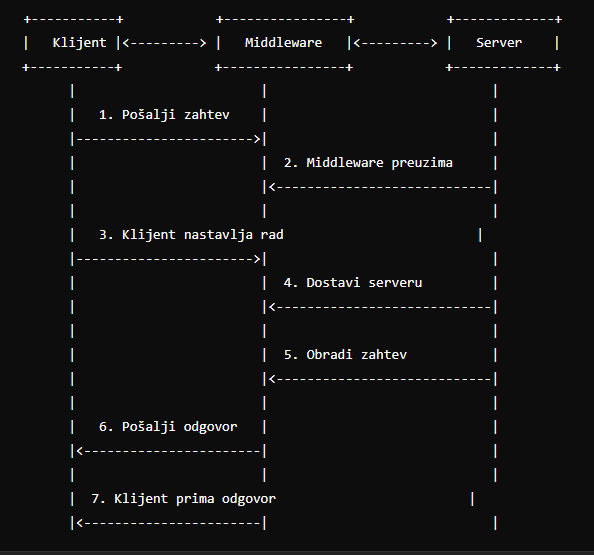


1. **Opisati model asinhrone komunikacije između klijenta i servera. Nacrtati i opisati dijagram.**

U asinhronoj komunikaciji, pošiljalac odmah nastavlja sa radom nakon što pošalje poruku. Middleware privremeno čuva poruku i kasnije je dostavlja serveru.

**Model asinhrone komunikacije:**

1. **Klijent šalje zahtev**: Klijent šalje zahtev ka serveru preko middleware-a.
2. **Middleware preuzima zahtev**: Middleware prima zahtev i potvrđuje prijem klijentu.
3. **Klijent nastavlja rad**: Klijent ne čeka odgovor i nastavlja sa daljim radom.
4. **Zahtev se dostavlja serveru**: Middleware dostavlja zahtev serveru kasnije.
5. **Server obrađuje zahtev**: Server obrađuje zahtev i priprema odgovor.
6. **Server šalje odgovor**: Server šalje odgovor nazad klijentu preko middleware-a.
7. **Klijent prima odgovor**: Klijent prima odgovor kada je spreman za prijem.

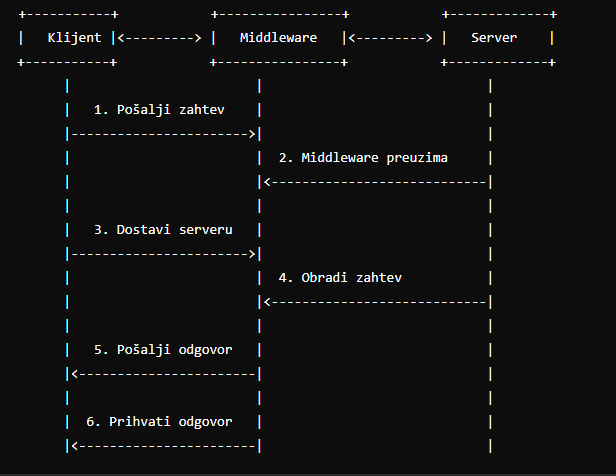


1. **Opisati model sinhrone komunikacije između klijenta i servera. Nacrtati i opisati dijagram. Gde su tačke sinhronizacije?**

U sinhronoj komunikaciji, pošiljalac se blokira i čeka na odgovor dok se poruka ne dostavi i obradi. Sinhrona komunikacija podrazumeva da pošiljalac mora čekati na različitim tačkama dok se njegov zahtev ne procesuira.

**Model sinhrone komunikacije:**

1. **Klijent šalje zahtev**: Klijent šalje zahtev ka serveru preko middleware-a.
2. **Middleware preuzima zahtev**: Middleware prima zahtev i potvrđuje prijem klijentu.
3. **Zahtev se dostavlja serveru**: Middleware dostavlja zahtev serveru.
4. **Server obrađuje zahtev**: Server obrađuje zahtev i priprema odgovor.
5. **Server šalje odgovor**: Server šalje odgovor nazad klijentu preko middleware-a.
6. **Klijent prima odgovor**: Klijent prima odgovor i nastavlja sa daljim radom.

****

1. **Šta predstavlja udaljeni poziv procedure (RPC)? Opisati pojednostavljeni način rada RPC-a sa klijentske i serverske strane.**

Kada proces na mašini A (pozivaoc) želi da izvrši proceduru koja se nalazi na mašini B (poslužilac), RPC omogućava da se taj poziv tretira kao lokalni poziv procedure. Evo kako to funkcioniše:

1. **Pozivanje procedure:** Proces na mašini A poziva udaljenu proceduru kao što bi pozvao bilo koju lokalnu proceduru.
2. **Suspendovanje pozivaoca:** Proces na mašini A je suspendovan dok se procedura izvršava na mašini B.
3. **Prenos parametara:** Parametri poziva se prenose sa mašine A na mašinu B.
4. **Izvršavanje procedure:** Procedura se izvršava na mašini B koristeći prenete parametre.
5. **Vraćanje rezultata:** Rezultati izvršavanja procedure vraćaju se sa mašine B na mašinu A.
6. **Nastavak rada:** Proces na mašini A nastavlja sa radom nakon što dobije rezultate.

**Klijentska strana:**

1. **Klijentski stub (Client Stub):**
   * Kada klijent želi da pozove udaljenu proceduru na serveru, on koristi klijentski stub.
   * Klijentski stub je lokalni deo koda na klijentskoj mašini koji pruža interfejs sličan interfejsu samih procedura koje se žele pozvati na serveru.
   * Klijentski stub pakuje parametre poziva (npr. argumente funkcije) u oblik koji je pogodan za prenos preko mreže (serijalizacija).
   * Nakon pakovanja parametara, klijentski stub poziva operaciju send za slanje poruke serveru preko mreže.
   * Nakon slanja poruke, klijentski stub obično poziva operaciju receive i blokira se dok ne stigne odgovor od servera.
2. **Mrežni prenos:**
   * Poruka sa pakovanim parametrima putuje preko mreže do servera.

**Serverska strana:**

1. **Serverski stub (Server Stub):**
   * Kada server prima poruku od klijenta, operativni sistem servera prosleđuje tu poruku serverskom stub-u.
   * Serverski stub je lokalni deo koda na serverskoj mašini koji prima dolazne poruke, raspakuje parametre iz poruke (deserijalizacija) i poziva stvarnu proceduru (koju je klijent želeo da pozove).
   * Serverski stub obavlja lokalne pozive procedura, kao da je sama procedura pozvana direktno na serveru.
2. **Izvršavanje procedure:**
   * Nakon što serverski stub izvrši proceduru, on priprema rezultate (ako postoje) za povrat klijentu.
   * Rezultati se pakuju u odgovornu poruku koja se šalje nazad klijentu preko mreže.
3. **Slanje odgovora:**
   * Serverski stub poziva operaciju send kako bi poslao odgovor klijentu.
   * Nakon slanja odgovora, serverski stub obično poziva receive i čeka sledeći zahtev od klijenta.

**Klijentska strana (nastavak):**

1. **Odgovor klijentu:**
   * Kada klijentski stub primi odgovor od servera, on ga raspakuje (deserijalizacija) i vraća ga klijentskoj aplikaciji kao rezultat poziva funkcije.
   * Klijentska aplikacija dobija rezultat kao da je funkcija lokalno pozvana, bez obzira na to da li je izvršenje procedure bilo udaljeno.
2. **Koji problemi mogu da se jave kod RPC-a?**

Kada se koristi RPC (Remote Procedure Call), mogu se pojaviti neki specifični problemi i izazovi zbog prirode distribuirane komunikacije i razlika između lokalnog i udaljenog izvršavanja. Evo nekih glavnih problema koji se mogu javiti kod RPC-a:

1. **Marshalling i Unmarshalling:**
   * **Problem:** Prilikom prosleđivanja parametara preko mreže, potrebno je upakovati (marshalling) podatke u format koji je nezavisan od mašine i mreže, kao i kasnije raspakovati (unmarshalling) te podatke na strani primaoca.
   * **Izazovi:** Razlike u redosledu bajtova (endianness) između različitih arhitektura računara mogu izazvati probleme prilikom interpretacije podataka. Takođe, treba voditi računa o tome da li su tipovi podataka kompatibilni između klijenta i servera.
2. **Prenosnost referenci i pokazivača:**
   * **Problem:** Referenca ili pokazivač na memoriju lokalnog računara nema značenje na udaljenom računaru. Potrebno je pronaći način da se adresni prostor lokalnog računara "preslika" na udaljeni računar.
   * **Izazovi:** U nekim slučajevima, prenos pokazivača nije moguć ili je složen. Moraju se koristiti alternativni načini za upravljanje deljenim resursima kao što su globalne reference ili upravljanje fajlovima.
3. **Konzistentnost i atomičnost:**
   * **Problem:** Ukoliko se RPC koristi za operacije koje menjaju stanje podataka (npr. upis u bazu podataka), važno je osigurati konzistentnost i atomičnost operacija.
   * **Izazovi:** Komunikacija preko mreže može biti nepouzdana. Treba razmotriti scenarije kao što su gubici poruka, redosled poruka i ponovno slanje u cilju održavanja konzistentnog stanja.
4. **Mrežni problemi:**
   * **Problem:** Kao i svaka mrežna aplikacija, RPC je podložan problemima kao što su kašnjenja, prekidi veze i zagušenja mreže.
   * **Izazovi:** Potrebno je implementirati mehanizme za upravljanje greškama, rukovanje izuzecima i ponovno slanje zahteva kako bi se obezbedila robusnost i pouzdanost sistema.
5. **Sigurnost:**
   * **Problem:** Komunikacija preko mreže može biti podložna napadima i neovlašćenom pristupu.
   * **Izazovi:** Potrebno je implementirati sigurnosne mehanizme kao što su enkripcija podataka, autentifikacija korisnika i autorizacija pristupa kako bi se zaštitili podaci i resursi u distribuiranom okruženju.
6. **Performanse:**
   * **Problem:** Efikasnost i performanse mogu biti pogođene zbog dodatnog vremena potrebnog za pakovanje, slanje i raspakivanje poruka preko mreže.
   * **Izazovi:** Treba balansirati potrebu za brzim odzivom sistema sa potrebom za efikasnim upravljanjem mrežnim resursima i protokom.
7. **Transparentnost i kompleksnost:**
   * **Problem:** Klijentska aplikacija obično želi da RPC bude što transparentniji, ali kompleksnost mehanizama koji stoje iza RPC-a može otežati dijagnostiku i otklanjanje problema.
   * **Izazovi:** Potrebno je pružiti dovoljno alata i logike za otkrivanje i rešavanje problema u distribuiranom okruženju bez kompromitovanja transparentnosti i jednostavnosti upotrebe RPC-a.

Kada se implementira RPC, važno je razmotriti ove faktore i prilagoditi arhitekturu i implementaciju tako da odgovara specifičnim zahtevima aplikacije i okruženja u kojem se koristi.

1. **Šta predstavlja asinhroni, a šta multicast RPC?**

Asinhroni RPC omogućava klijentu da nastavi rad nakon slanja zahteva serveru, čak i pre nego što dobije konačan odgovor. Kada klijent pošalje asinhroni RPC zahtev, server odmah šalje potvrdu klijentu da je zahtev primljen. Nakon toga, server lokalno poziva traženu proceduru i nastavlja sa obradom. Kada server završi izvršavanje RPC-a, može poslati odgovor klijentu, ali klijent može i dalje raditi bez blokiranja čekanja na odgovor.

Multicast RPC se koristi za slanje istog RPC zahteva grupi servera istovremeno. Klijent šalje jedan RPC zahtev i serveri koji su deo grupe nezavisno i paralelno obrađuju taj zahtev. Svaki server može odgovoriti klijentu sa svojim rezultatom ili klijent može čekati sve rezultate pre nego što nastavi sa radom.

Asinhroni RPC je koristan kada klijent ne mora odmah dobiti rezultate RPC-a ili kada je potrebno izvršiti nekoliko RPC poziva paralelno. Klijent može kasnije provjeriti ili pričekati rezultate prethodno poslatih asinhronih zahteva.

Multicast RPC se koristi kada je potrebno paralelno obraditi isti zahtev na više servera radi povećanja otpornosti na greške ili za bolje iskorišćenje resursa. Može se koristiti i za distribuirane sisteme gde je potrebno brzo dobijanje odgovora iz više izvora.

**L08 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Šta podrazumeva ekskluzivni pristup resursu od strane procesa?**

Ekskluzivni pristup resursu od strane procesa podrazumeva da samo jedan proces može pristupiti tom resursu u bilo kom trenutku, dok su svi drugi procesi blokirani ili u stanju čekanja. Ovo je važno kako bi se izbegli konflikti, nekonzistentnost podataka ili druge nepoželjne posledice koje mogu nastati kada više procesa istovremeno pokušava da pristupi i modifikuje isti resurs.

* 1. **Zašto postoji problem sinhronizacije časovnika u distribuiranim sistemima?**

Postoji problem sinhronizacije časovnika u distribuiranim sistemima iz nekoliko razloga:

1. **Različiti lokalni časovnici:** Svaki čvor u distribuiranom sistemu ima svoj lokalni časovnik koji može biti različito podešen i može otkucavati različitom brzinom. To dovodi do toga da časovnici na različitim čvorovima nisu sinhronizovani.
2. **Mrežna kašnjenja:** Komunikacija između čvorova u distribuiranom sistemu prolazi kroz mrežu, što uvodi kašnjenja koja nisu konstantna i mogu varirati. Ova kašnjenja otežavaju tačno merenje i usklađivanje vremena između čvorova.
3. **Drift časovnika:** Časovnici u računarima nisu savršeni i s vremenom mogu driftovati, tj. njihova tačnost može se smanjivati zbog razlika u frekvenciji kristala koji se koriste za mjerenje vremena. Ovaj drift može uzrokovati dodatnu nesinhronizaciju među čvorovima.
4. **Topologija mreže:** U distribuiranim sistemima, topologija mreže može biti složena i može uključivati različite rute za komunikaciju između čvorova. To znači da putanja koju poruka pređe od jednog čvora do drugog može varirati, što dodatno otežava preciznu sinhronizaciju vremena.
5. **Ograničenja protokola za sinhronizaciju:** Protokoli koji se koriste za sinhronizaciju vremena, kao što su NTP (Network Time Protocol) ili PTP (Precision Time Protocol), imaju svoja ograničenja i mogu uvoditi greške u sinhronizaciji zbog varijabilnih mrežnih kašnjenja i drugih faktora.
   1. **Ukratko opisati način rada fizičkog časovnika u računarskom sistemu.**

Fizički časovnik u računaru je obično baziran na kvarcnom kristalu koji oscilira na preciznoj frekvenciji pod tenzijom. Ovaj kristal je povezan sa brojačem i registrom za čuvanje. Svaka oscilacija kristala smanjuje vrednost brojača. Kada brojač dostigne nulu, generiše se prekid koji se koristi za održavanje vremena u računaru. Softver koji upravlja časovnikom zatim obnavlja brojač iz registra za čuvanje, omogućavajući časovniku da nastavi sa održavanjem vremena. Na ovaj način, računar može precizno pratiti prolazak vremena bez obzira na druge aktivnosti sistema. Važno je napomenuti da se fizički časovnik ne zasniva na "pravom" vremenu (kao što je vreme mereno na zidnom satu), već na internom sistemu brojanja oscilacija kvarcnog kristala.

* 1. **Šta predstavljaju Lampartovi časovnici u računarskom sistemu?**

Lamportovi satovi su algoritam koji omogućava procesima u distribuiranom sistemu da usaglase vremenski red događaja, iako svaki proces ima svoj lokalni časovnik čiji takt varira. Ovaj algoritam je nazvan po Leslie Lamportu, koji ga je prvi put opisao 1978. godine.

**Funkcionisanje Lamportovih satova:**

1. **Softverski brojači događaja:** Umesto da koriste apsolutno vreme kao što je trenutak na zidnom satu, Lamportovi satovi koriste brojač događaja. Svaki proces ima svoj lokalni brojač koji se inkrementira kada se dogodi lokalni događaj ili kada se pošalje poruka.
2. **Odnos "desi se pre" (happens-before):** Centralna ideja Lamportovih satova je da, ako događaj A mora da se desi pre događaja B, tada se vreme događaja A mora smatrati manjim od vremena događaja B. Ovaj odnos se zasniva na uzajamnom opažanju događaja u distribuiranom sistemu.
3. **Inkrementiranje lokalnog sata:** Kada proces pošalje poruku, on u poruci uključuje trenutno vreme svog lokalnog sata. Kada poruka stigne na drugi proces, taj proces ažurira svoj lokalni sat: uzima maksimum između svog trenutnog sata i vremena koje je dobilo u poruci, plus jedan. Ovaj korak osigurava da se očuva "desi se pre" odnos među događajima.
4. **Primer primene:** Recimo da proces P1 šalje poruku m1 procesu P2 u trenutku kada njegov sat pokazuje 6. Kada poruka stigne u P2, ako P2-ov sat pokazuje 16, P2 će ažurirati svoj sat na max(16, 6) + 1 = 17. Ovaj mehanizam garantuje da se "stvarni redosled" događaja održava čak i ako lokalni satovi imaju različite taktove.
5. **Preventivne mere:** Lamportovi satovi ne garantuju apsolutnu tačnost vremena izvan konteksta lokalnog sistema. Njihov glavni cilj je omogućiti korektno upravljanje redosledom događaja u distribuiranom okruženju, gde su vremenske varijacije među procesima neizbežne.
   1. **Opisati algebarsku relaciju *a → b*(en. happens-before).**

Algebarska relacija a→ba \rightarrow ba→b (ili happens-before) u Lamportovom algoritmu predstavlja osnovu za uspostavljanje redosleda događaja u distribuiranim sistemima. Ova relacija se definiše na sledeći način:

1. **Isti proces:** Ako su događaji aaa i bbb u istom procesu i aaa se dešava pre bbb, tada je a→ba \rightarrow ba→b istinito. Ovo znači da lokalni redosled događaja unutar jednog procesa određuje relaciju "desi se pre".
2. **Poruke između procesa:** Ako je aaa događaj slanja poruke od jednog procesa, a bbb događaj prijema iste te poruke od drugog procesa, tada je a→ba \rightarrow ba→b istinito. Ova situacija reflektuje stvarni redosled u komunikaciji između procesa.
   1. **Opisati Lampartov algoritam (konkretan primer)**

Lamportov algoritam, nazvan po Lesli Lamportu koji ga je prvi put opisao 1978. godine, predstavlja jednostavan način za uspostavljanje redosleda događaja u distribuiranim sistemima. Ovaj algoritam se oslanja na koncept "logičkih satova" ili "satova događaja" (event clocks), umesto da koristi apsolutno sinhronizovano vreme.

**Ključne ideje Lamportovog algoritma:**

1. **Logički satovi:** Svaki proces u sistemu ima svoj lokalni logički sat (event clock) koji se inkrementira svaki put kada se desi lokalni događaj ili kada se pošalje poruka. Ovaj sat se ne zasniva na stvarnom vremenu, već na broju događaja ili poruka.
2. **Relacija "desi se pre" (happens-before):** Osnovna relacija koju Lamportov algoritam koristi je →, koja označava "događaj a se dešava pre događaja b". Postoji nekoliko pravila koja definišu ovu relaciju:
   * Ako su a i b događaji u istom procesu i a se dešava pre b, tada važi a→ba → ba→b.
   * Ako je a događaj slanja poruke, a b je događaj prijema iste te poruke, tada važi a→ba \rightarrow ba→b.
3. **Ažuriranje lokalnog sata:** Kada proces pošalje poruku, on u poruci uključuje trenutno vreme svog lokalnog sata. Kada poruka stigne na drugi proces, taj proces ažurira svoj lokalni sat na max(svog trenutnog sata, vreme koje je dobilo u poruci) + 1. Ovaj korak osigurava da se očuva "desi se pre" odnos među događajima.

**Primer primene Lamportovog algoritma:**

Razmotrimo jednostavan primer sa tri procesa: P1, P2 i P3.

1. **Inicijalizacija satova:**
   * P1 ima sat koji se inkrementira za 1 za svaki lokalni događaj.
   * P2 ima sat koji se inkrementira za 2 za svaki lokalni događaj.
   * P3 ima sat koji se inkrementira za 1 za svaki lokalni događaj.
2. **Redosled događaja:**
   * P1 šalje poruku m1 procesu P2. Pre nego što pošalje poruku, P1 inkrementira svoj sat sa 6 na 7 (pretpostavimo da je trenutno vreme 6).
   * P2 prima poruku m1. P2 ažurira svoj sat na max(2, 7) + 1 = 8, jer je P2-ov sat bio 2 kada je primio poruku.
   * P2 šalje poruku m2 procesu P3. Pre slanja, P2 inkrementira svoj sat sa 8 na 10.
   * P3 prima poruku m2. P3 ažurira svoj sat na max(1, 10) + 1 = 11, jer je P3-ov sat bio 1 kada je primio poruku.
   1. **Šta predstavlja pojam međusobnog isključenja?**

Pojam međusobnog isključenja se odnosi na koncept u računarstvu koji osigurava da konkurentni procesi ili niti ne istovremeno pristupaju istom deljenom resursu. Deljeni resurs može biti promenljiva, datoteka, hardverski uređaj ili bilo koji drugi resurs koji se može koristiti u više procesa ili niti.

**Primer:**  
Na primer, ako imamo dva procesa koji žele pristupiti deljenoj promenljivoj x, međusobno isključenje se postiže korišćenjem mutex-a. Procesi P1 i P2 moraju zaključati mutex pre nego što pristupe promenljivoj x. Ako P1 prvi zaključa mutex, P2 mora čekati da P1 oslobodi mutex pre nego što može nastaviti sa svojim izvršavanjem. Ovaj pristup sprečava da se P1 i P2 mešaju pri pristupu promenljivoj x, čime se održava integritet podataka.

* 1. **Opisati centralizovani algoritam međusobnog isključenja.**

U centralizovanom algoritmu za međusobno isključenje, proces P1, kada završi sa korišćenjem resursa, šalje poruku koordinatoru sistema kako bi oslobodio ekskluzivni pristup resursu. Koordinator zatim obradjuje listu odloženih zahteva i šalje poruku odobrenja prvom procesu koji je na redu za pristup resursu. Ovaj proces se obično naziva zahtevanje i oslobađanje zaključavanja

* 1. **Opisati distribuirani algoritam međusobnog isključenja.**

Distribuirani algoritam koji koristi Lamportove logičke časovnike omogućava kontrolu pristupa deljenim resursima u sistemu. Evo ključnih tačaka kako algoritam funkcioniše:

1. **Poruke sa zahtevom:** Kada proces želi pristupiti deljenom resursu, šalje poruku svim ostalim procesima. Poruka sadrži ime resursa, broj procesa i trenutno logičko vreme (timestamp) tog procesa.
2. **Pouzdanost poruka:** Pretpostavlja se da je slanje poruka pouzdano, što znači da nijedna poruka neće biti izgubljena ili izmenjena tokom prenosa.
3. **Obrada zahteva:**
   * Kada proces primi zahtevnu poruku od drugog procesa, postoji nekoliko mogućih akcija u zavisnosti od trenutnog stanja:
     + Ako primalac ne pristupa resursu i ne želi mu pristupiti, šalje nazad poruku "OK" pošiljaocu.
     + Ako primalac već ima pristup resursu, ignoriše zahtev i stavlja ga u red čekanja.
     + Ako primalac takođe želi pristupiti resursu, ali ga trenutno nema, upoređuje timestamp dolazne poruke sa svojim vlastitim timestamp-om.
       - Ako dolazna poruka ima niži timestamp od vlastitog, primalac šalje nazad poruku "OK" pošiljaocu.
       - Ako vlastiti timestamp ima nižu vrednost, primalac stavlja dolazni zahtev u red i ne šalje odgovor.
4. **Jednoznačnost redosleda događaja:** Korišćenjem Lamportovih logičkih časovnika, algoritam osigurava da za bilo koji par događaja (npr. poruka) postoji jedinstven redosled događaja. To znači da procesi mogu odrediti koji se događaj zaista desio prvi, iako rade asinhrono.
5. **Implementacija i konkurentnost:** Algoritam omogućava konkurentan pristup resursima, ali uz mehanizam koji garantuje da nijedan proces neće oštetiti integritet resursa jer se međusobno isključuju prilikom pristupa.
6. **Opisati algoritam žetona (token ring) međusobnog isključenja.**

Algoritam prstena je distribuirani pristup međusobnom isključivanju u sistemima sa više procesa, koji koristi koncept logičkog prstena kako bi deterministički upravljao pristupom deljenim resursima. Evo ključnih tačaka o tom algoritmu:

1. **Inicijalizacija prstena:** Prsten se inicijalizuje dodeljivanjem žetona (tokena) procesu P0. Token zatim cirkuliše redosledom procesa u prstenu.
2. **Kruženje tokena:** Token putuje od jednog procesa do drugog u prstenu, koristeći poruke tačka-do-tačka između susednih procesa. Na primer, ako postoji N procesa, token se prenosi od procesa Pk do P(k+1) mod N.
3. **Pravo na pristup resursu:** Kada proces dobije token od svog suseda, proverava da li ima potrebu za pristupom deljenom resursu. Ako je proces zainteresovan, pristupa resursu, obavlja operacije koje su mu potrebne i zatim oslobađa resurse.
4. **Dalje prosleđivanje tokena:** Ako proces ne treba resurs, jednostavno prosleđuje token svom narednom susedu u prstenu. Ovo osigurava da token uvek cirkuliše i da se ne zadržava kod procesa koji trenutno ne treba resurs.
5. **Jedinstvenost pristupa:** Budući da samo jedan proces uvek poseduje token u bilo kom trenutku, samo taj proces može pristupiti resursu. Ovo eliminše potrebu za centralizovanim koordinatorom i omogućava deterministički mehanizam za međusobno isključivanje.
6. **Prednosti algoritma:** Algoritam prstena garantuje da će svaki proces imati priliku da pristupi resursu u dobro definisanom redosledu, sprečavajući potencijalne probleme kao što su trkaći uslovi ili blokiranje procesa.

**L09 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Šta predstavlja pojam razrešenja imena (en. name resolution).**

Razrešenje imena je proces prevođenja imena ili identifikatora u odgovarajuće adrese. Postoje dva glavna pristupa:

1. **Tabela parova (ime, adresa)**:
   * Održava se tabela parova (ime, adresa).
   * Primer: DNS sistem pretražuje tabele da bi našao IP adresu za dati URL.
2. **Postepeno usmeravanje zahteva**:
   * Zahtev se postepeno usmerava ka odgovarajućoj adresi.
   * Primeri: Peer-to-peer sistemi, Information-centric networking (ICN).

Ovi pristupi omogućavaju korisnicima da koriste razumljiva imena umesto numeričkih adresa za pristup mrežnim resursima.

* 1. **Ukratko opisati ravne sisteme imenovanja komponenti.**

Ravni sistemi imenovanja koriste nestruktuirane identifikatore, koji su slučajni nizovi bitova bez informacija o lokaciji entiteta. Ovi identifikatori su jednostavni za jedinstveno predstavljanje entiteta. Kada koristimo ravna imena, moramo pronaći način da lociramo entitet koristeći samo taj identifikator.

Difuzno emitovanje (broadcasting) je jedan način za lociranje entiteta. Ovaj metod uključuje slanje zahteva svim čvorovima u mreži, što je efikasno u malim, lokalnim mrežama. Drugi način je korišćenje pokazivača za prosleđivanje (forwarding pointers). Svaki čvor u mreži čuva pokazivač ka sledećem čvoru bližem cilju, što je korisno za praćenje pokretnih entiteta u lokalnim mrežama.

Ova dva pristupa su jednostavna i često dovoljno efikasna za lokalne mreže, ali nisu prikladna za veće, distribuirane sisteme.

* 1. **Opisati razlike između entiteta, pristupne tačke, adrese i identifikatora.**

**Entitet** je bilo koji resurs ili objekat u distribuiranom sistemu, kao što su računari, štampači, datoteke, procesi, korisnici, mrežne veze, itd. Entiteti su objekti nad kojima se mogu izvršavati razne operacije, poput štampanja na štampaču ili slanja podataka putem mrežne veze.

**Pristupna tačka** je specifična lokacija ili interfejs preko kojeg se može pristupiti entitetu. To je "ulaz" putem kojeg se komunicira sa entitetom. Na primer, telefonski broj je pristupna tačka za kontaktiranje osobe, a IP adresa i port su pristupne tačke za pristup serveru na mreži.

**Adresa** je ime koje identifikuje pristupnu tačku entiteta. Adrese su specijalne vrste imena koje direktno omogućavaju pristup entitetima. Na primer, IP adresa zajedno sa brojem porta može biti adresa servera. Telefonski broj je adresa putem koje se može kontaktirati određena osoba.

**Identifikator** je jedinstveni niz bitova ili karaktera koji se koristi za jednoznačno predstavljanje entiteta unutar sistema. Identifikatori su često nestruktuirani i ne pružaju informacije o lokaciji entiteta. Njihova svrha je da jedinstveno označe entitet, kao što je serijski broj uređaja ili jedinstveni korisnički ID.

**Razlike između entiteta, pristupne tačke, adrese i identifikatora:**

* **Entitet** je stvarni resurs ili objekat u sistemu koji pruža određene funkcionalnosti.
* **Pristupna tačka** je specifična lokacija ili interfejs kroz koji se pristupa entitetu.
* **Adresa** je ime koje identifikuje pristupnu tačku i omogućava pristup entitetu.
* **Identifikator** je jedinstveni niz koji jednoznačno označava entitet, ali ne daje informacije o tome kako mu pristupiti.

Na primer, osoba (entitet) može imati više telefonskih brojeva (pristupne tačke), pri čemu svaki telefonski broj (adresa) omogućava pozivanje te osobe. Identifikator za osobu može biti njen jedinstveni lični broj (npr. JMBG), koji jednoznačno označava tu osobu, ali ne pruža informacije o tome kako joj se može pristupiti.

* 1. **Dati primer protokola difuznog emitovanja za razrešenje imena.**

Jedan od najpoznatijih primera protokola difuznog emitovanja za razrešenje imena je **Address Resolution Protocol (ARP)**. ARP se koristi u IPv4 mrežama za mapiranje IP adresa na fizičke (MAC) adrese. Kada uređaj treba da komunicira sa drugim uređajem na istoj lokalnoj mreži, a ne zna njegovu MAC adresu, koristi ARP da je pronađe. Evo kako funkcioniše:

1. **ARP Request (zahtev)**: Kada uređaj (A) želi da komunicira sa drugim uređajem (B) i zna samo njegovu IP adresu, A šalje ARP zahtev. ARP zahtev je difuzna (broadcast) poruka koja se šalje svim uređajima na lokalnoj mreži. Poruka sadrži IP adresu B i pita: "Ko ima ovu IP adresu?"
2. **ARP Response (odgovor)**: Svaki uređaj na mreži prima ARP zahtev. Samo uređaj sa odgovarajućom IP adresom (u ovom slučaju, uređaj B) odgovara na ARP zahtev. B šalje ARP odgovor kao unicast poruku direktno uređaju A. ARP odgovor sadrži MAC adresu uređaja B.
3. **Keširanje**: Nakon što uređaj A primi ARP odgovor, on kešira (čuva) IP-MAC par u svojoj ARP tabeli za buduću upotrebu. Ovo omogućava da se budući paketi šalju direktno, bez potrebe za dodatnim ARP zahtevima.
   1. **Opisati protokol pokazivača za prosleđivanje (en. forwarding pointers).**

**Protokol pokazivača za prosleđivanje (Forwarding Pointers)**

Protokol pokazivača za prosleđivanje koristi se za lociranje entiteta koji se može kretati unutar distribuiranog sistema. Ovaj protokol koristi pokazivače koji ukazuju na novu lokaciju entiteta kada se on premesti. Proces funkcionira na sledeći način:

1. **Premještanje entiteta**:
   * Kada se entitet premesti iz lokacije A u lokaciju B, on ostavlja pokazivač na lokaciji A koji ukazuje na njegovu novu lokaciju u B. Ovaj pokazivač sadrži informacije o adresi B.
2. **Pretraživanje entiteta**:
   * Kada klijent želi pristupiti entitetu, započinje pretragu na poznatoj početnoj lokaciji (u ovom slučaju A).
   * Ako entitet više nije na lokaciji A, klijent će pronaći pokazivač koji ukazuje na novu lokaciju B.
3. **Praćenje pokazivača**:
   * Klijent zatim koristi pokazivač da se premesti na lokaciju B.
   * Ako entitet više nije na lokaciji B, pronaći će novi pokazivač koji ukazuje na sledeću lokaciju, recimo C.
   * Ovaj proces se ponavlja sve dok se ne pronađe trenutna lokacija entiteta.
   1. **Opisati način rada mobilnog IP-a.**

Mobilni IP (Mobile IP) je protokol koji omogućava mobilnim uređajima (hostovima) da se slobodno kreću između različitih mreža dok zadržavaju stalnu IP adresu. Ovo omogućava kontinuiranu komunikaciju bez prekida veze zbog promene mrežnih lokacija.

Komponente Mobilnog IP-a uključuju mobilni čvor (MN), Home Agent (HA), Foreign Agent (FA) i Care-of Address (CoA). Mobilni čvor je uređaj koji se kreće između mreža, kao što je mobilni telefon ili laptop. Home Agent je entitet na home mreži mobilnog čvora koji prati njegovu trenutnu lokaciju i odgovoran je za preusmeravanje paketa ka mobilnom čvoru kada se on nalazi izvan svoje home mreže. Foreign Agent je entitet na stranoj mreži koja trenutno služi mobilnom čvoru, može dodeliti privremenu adresu mobilnom čvoru i prosleđivati pakete između njega i Home Agenta. Care-of Address je privremena IP adresa koju mobilni čvor koristi dok je na stranoj mreži, može biti dodeljena direktno mobilnom čvoru ili preko Foreign Agenta.

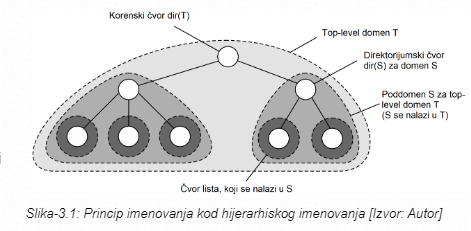
Kada se mobilni čvor poveže na stranu mrežu, dobija privremenu adresu (Care-of Address, CoA) i obaveštava svog Home Agenta o svojoj novoj Care-of Address adresi. Kada pošiljalac pošalje paket mobilnom čvoru, paket se najpre šalje na Home Agent koristeći stalnu IP adresu mobilnog čvora. Home Agent proverava gde se mobilni čvor trenutno nalazi. Ako je na stranoj mreži, Home Agent preusmerava (tuneluje) paket na Care-of Address adresu mobilnog čvora. Home Agent upakuje originalni IP paket unutar novog IP paketa sa destinacijom Care-of Address i šalje ga ka stranoj mreži. Kada paket stigne na stranu mrežu, Foreign Agent (ili direktno mobilni čvor ako nema Foreign Agenta) otpakovava originalni paket i isporučuje ga mobilnom čvoru. Kada mobilni čvor odgovara, može direktno komunicirati sa pošiljaocem koristeći Care-of Address kao izvor IP adrese. Pošiljalac može naučiti Care-of Address mobilnog čvora i koristiti je za direktnu komunikaciju, smanjujući opterećenje na Home Agentu.

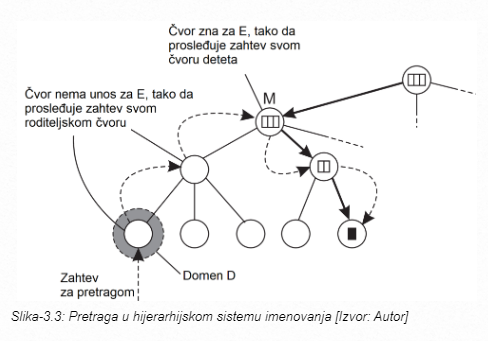
* 1. **Ukratko opisati sisteme hijerarhijskog imenovanja. Nacrtati dijagram.**

U hijerarhijskom sistemu imenovanja, mreža je organizovana u domene koji su strukturirani u hijerarhiju. Najviši nivo ove hijerarhije je top-level domen koji obuhvata celu mrežu. Ovaj domen se može dalje podeliti na poddomene, a najniži nivo domena, nazvan domen lista, obično odgovara lokalnoj mreži ili ćeliji.

Svaki domen ima svoj direktorijumski čvor koji prati entitete u tom domenu. Direktorijumski čvor top-level domena, nazvan korenski čvor, zna za sve entitete u mreži. Kada se entitet premesti ili replikira, zapisi o njegovoj lokaciji ažuriraju se u odgovarajućim direktorijumskim čvorovima. Najniži direktorijumski čvorovi sadrže tačne adrese entiteta, dok viši nivo direktorijumski čvorovi čuvaju pokazivače na niže čvorove gde se entitet nalazi.

Kada klijent želi da locira entitet, započinje pretragu u listnom domenu. Ako direktorijumski čvor nema zapis o lokaciji entiteta, zahtev se prosleđuje roditeljskom čvoru na sledećem višem nivou. Ovaj proces se ponavlja sve dok se ne pronađe direktorijumski čvor koji ima zapis o lokaciji entiteta. Zahtev se tada prosleđuje naniže kroz hijerarhiju sve do listnog čvora koji ima tačnu adresu entiteta. Na taj način, pretraga u hijerarhijskom sistemu imenovanja omogućava efikasno i organizovano lociranje entiteta u distribuiranim sistemima.



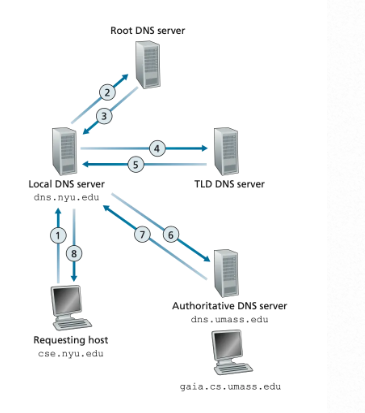
* 1. **Opisati način pretrage entiteta kod hijerarhijskog imenovanja. Nacrtati dijagram.**

U hijerarhijskom sistemu imenovanja, pretraga se odvija korak po korak kroz strukturu hijerarhije domena kako bi se pronašla tačna lokacija entiteta. Evo detaljnog opisa kako se odvija pretraga:

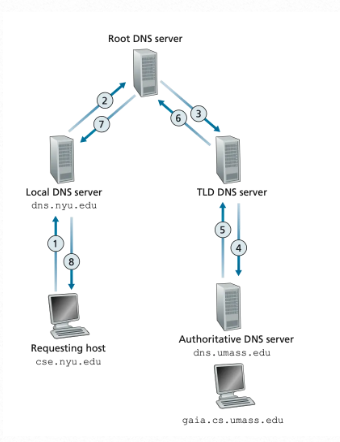
1. **Početak pretrage:**
   * Klijent koji želi da locira određeni entitet započinje pretragu u listnom domenu u kojem se nalazi.
2. **Provera direktorijumskog čvora:**
   * Klijent šalje zahtev za pretragu direktorijumskom čvoru listnog domena.
   * Direktorijumski čvor proverava svoj zapis o lokaciji entiteta. Ako pronađe zapis, vraća adresu entiteta klijentu.
3. **Prosleđivanje zahteva roditeljskom čvoru:**
   * Ako direktorijumski čvor nema zapis o lokaciji entiteta, prosleđuje zahtev roditeljskom čvoru na sledećem višem nivou u hijerarhiji.
4. **Ponavljanje procesa:**
   * Postupak se ponavlja sve dok zahtev ne stigne do direktorijumskog čvora koji ima zapis o lokaciji entiteta.
   * Svaki roditeljski čvor predstavlja veći domen u odnosu na svoje potomke. Ako ni roditeljski čvor nema zapis o lokaciji, zahtev se dalje prosleđuje naviše u hijerarhiji.
5. **Pronalaženje odgovarajućeg čvora:**
   * Kada zahtev stigne do direktorijumskog čvora koji čuva zapis o lokaciji entiteta, zna se da se entitet nalazi negde u tom domenu.
6. **Konačno lociranje entiteta:**
   * Direktorijumski čvor koji ima zapis o lokaciji entiteta prosleđuje zahtev naniže ka odgovarajućem poddomenu, gde se entitet tačno nalazi.
7. **Vraćanje adrese entiteta:**
   * Na kraju, direktorijumski čvor u listnom domenu koji tačno sadrži adresu entiteta vraća tu adresu klijentu koji je inicirao pretragu.
   1. **Opisati kako radi iterativni DNS upis. Nacrtati dijagram.**

Iterativni DNS upit je proces pretrage i dobijanja IP adrese (ili drugog zapisa) za određeni domen putem iterativnog slanja upita kroz hijerarhiju DNS servera. Evo detaljnog opisa kako radi iterativni DNS upit:

1. **Inicijacija upita:**
   * Klijent šalje DNS upit svom lokalnom DNS resolveru sa zahtevom za određenim domenom (npr. [www.example.com](http://www.example.com)).
2. **Lokalni DNS resolver:**
   * Lokalni DNS resolver prima upit od klijenta. Umesto da odmah odgovori ili izvrši rekurzivnu pretragu, lokalni resolver šalje iterativne upite drugim DNS serverima.
3. **Upit korenskim DNS serverima:**
   * Lokalni DNS resolver započinje upit tako što šalje zahtev korenskim DNS serverima. Korenski DNS serveri su vrhovni autoriteti za ceo DNS sistem i održavaju informacije o top-level domenima (TLD).
4. **Dobijanje IP adresa TLD DNS servera:**
   * Korenski DNS serveri odgovaraju lokalnom DNS resolveru sa IP adresama TLD (top-level domain) DNS servera koji su nadležni za određene TLD-ove (npr. .com, .net, .org).
5. **Upit TLD DNS serverima:**
   * Lokalni DNS resolver bira jedan od dobijenih TLD DNS servera i šalje mu upit. Upit sadrži zahtev za autoritativnim DNS serverima koji su nadležni za konkretan domen (npr. example.com).
6. **Dobijanje IP adresa autoritativnih DNS servera:**
   * TLD DNS server odgovara lokalnom DNS resolveru sa IP adresama autoritativnih DNS servera za domen example.com.
7. **Upit autoritativnim DNS serverima:**
   * Lokalni DNS resolver zatim šalje upit jednom od autoritativnih DNS servera za konkretan domen (npr. ns1.example.com).
8. **Dobijanje traženih informacija:**
   * Autoritativni DNS server za domen example.com odgovara lokalnom DNS resolveru sa traženim informacijama, kao što su IP adresa za [www.example.com](http://www.example.com).
9. **Odgovor klijentu:**
   * Lokalni DNS resolver kada dobije odgovor od autoritativnog DNS servera, vraća odgovor klijentu koji je inicirao upit. To može uključivati IP adresu zahtevanog domena (npr. [www.example.com](http://www.example.com)).

****

1. **Opisati kako radi rekurzivni DNS upit. Nacrtati dijagram.**

****

Rekurzivni DNS upit je proces pretrage i dobijanja IP adrese (ili drugih zapisa) za određeni domen koji se izvršava kroz rekurzivno traženje od lokalnog DNS resolvera. Evo detaljnog opisa kako radi rekurzivni DNS upit:

1. **Inicijacija upita:**
   * Klijent šalje DNS upit svom lokalnom DNS resolveru sa zahtevom da razreši naziv domena (npr. [www.example.com](http://www.example.com)) u IP adresu.
2. **Provera keš memorije lokalnog DNS resolvera:**
   * Lokalni DNS resolver prvo proverava svoju keš memoriju da vidi da li već ima tražene informacije (IP adresu za [www.example.com](http://www.example.com)). Ako lokalni resolver već ima odgovarajući zapis u kešu, odgovorit će klijentu direktno iz keša.
3. **Upit korenskim DNS serverima:**
   * Ako lokalni DNS resolver nema tražene informacije u kešu, on pokreće proces rekurzivne pretrage tako što šalje upit korenskim DNS serverima.
   * Korenski DNS serveri su najviši nivo DNS hijerarhije i održavaju informacije o svim top-level domenima (TLD) (.com, .net, .org itd.).
4. **Dobijanje IP adresa TLD DNS servera:**
   * Korenski DNS serveri odgovaraju lokalnom DNS resolveru sa IP adresama TLD DNS servera koji su nadležni za traženi TLD domen (npr. .com).
5. **Upit TLD DNS serverima:**
   * Lokalni DNS resolver bira jedan od dobijenih TLD DNS servera i šalje mu upit, tražeći autoritativne DNS servere za naredni nivo domena (npr. example.com).
6. **Dobijanje IP adresa autoritativnih DNS servera:**
   * TLD DNS server odgovara lokalnom DNS resolveru sa IP adresama autoritativnih DNS servera za domen example.com.
7. **Upit autoritativnim DNS serverima:**
   * Lokalni DNS resolver zatim šalje upit jednom od autoritativnih DNS servera za konkretan domen (npr. ns1.example.com), tražeći tražene informacije (npr. IP adresu za [www.example.com](http://www.example.com)).
8. **Dobijanje traženih informacija:**
   * Autoritativni DNS server za domen example.com odgovara lokalnom DNS resolveru sa traženim informacijama, kao što su IP adresa za [www.example.com](http://www.example.com).
9. **Odgovor klijentu:**
   * Lokalni DNS resolver kada dobije odgovor od autoritativnog DNS servera, vraća odgovor klijentu koji je inicirao upit. To može uključivati IP adresu zahtevanog domena (npr. [www.example.com](http://www.example.com)).
10. **Opisati kako se formira ime koristeći LDAP.**

U LDAP-u (Lightweight Directory Access Protocol), ime svakog unosa je globalno jedinstveno i formira se koristeći niz atributa imenovanja koji se nazivaju relativno različita imena (Relative Distinguished Names, RDNs). Ovi atributi imenovanja zajedno čine jedinstveno ime za svaki unos u direktorijumu. Evo kako se formira ime u LDAP-u koristeći konvencionalne skraćenice za atribute imenovanja:

1. **Primena atributa imenovanja:**
   * Uzmimo primer koristeći atribute Country (C), Organization (O), i OrganizationalUnit (OU):

/C=NL/O=VU University/OU=Computer Science

1. **Pojasnjenje:**
   * C=NL označava zemlju (Netherlands).
   * O=VU University označava organizaciju (VU University).
   * OU=Computer Science označava organizacionu jedinicu (Computer Science).
2. **Globalno jedinstveno ime:**
   * Kombinacija ovih atributa čini globalno jedinstveno ime za unutar LDAP direktorijuma. Ova struktura ima sličnost sa DNS imenom, gde se koristi hijerarhijski niz imena za identifikaciju resursa.
3. **Stablo informacija direktorijuma (DIT):**
   * Imenski graf ili stablo informacija direktorijuma (DIT) formira se na osnovu ovih globalno jedinstvenih imena. Svaki čvor u DIT-u predstavlja jedan direktorijumski unos.
4. **Hijerarhija i odnosi:**
   * Hijerarhijski odnos između čvorova u DIT-u odražava se u načinu formiranja imena. Na primer, u datom primeru, OU=Computer Science je poddirektorijum O=VU University, a O=VU University je poddirektorijum C=NL.

**L10 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

* 1. **Opisati razloge za replikaciju podataka.**

Replikacija podataka u distribuiranim sistemima se vrši iz dva osnovna razloga: povećanje pouzdanosti sistema i poboljšanje performansi.

**Povećanje pouzdanosti sistema:**

1. **Održavanje dostupnosti:** Kada su podaci replicirani, sistem može nastaviti s radom čak i ako jedna ili više replika postanu nedostupne. U slučaju pada jedne replike, sistem se može prebaciti na neku od drugih dostupnih replika, čime se povećava pouzdanost i dostupnost sistema.
2. **Zaštita od oštećenih podataka:** Više kopija podataka omogućava bolju zaštitu od gubitka ili oštećenja podataka. Na primer, ako postoje tri kopije datoteke, sistem može koristiti mehanizme glasanja da bi odredio tačnu verziju podataka. Ako se jedna kopija pokvari, druge dve mogu biti korišćene za vraćanje tačnih podataka.
3. **Kontinuirani rad:** U slučaju greške ili pada sistema, replikacija omogućava da sistem nastavi s radom bez prekida, osiguravajući kontinuitet poslovanja i smanjujući gubitke uzrokovane prekidom rada.

**Poboljšanje performansi:**

1. **Skaliranje veličine sistema:** Kada se broj korisnika ili procesa koji pristupaju podacima povećava, performanse se mogu poboljšati replikacijom servera. Deljenje opterećenja među replikama omogućava većem broju korisnika simultan pristup podacima bez usporavanja sistema.
2. **Geografsko skaliranje:** Postavljanjem replika podataka bliže korisnicima koji ih koriste, vreme pristupa podacima se smanjuje. Ovo je posebno korisno u globalno distribuiranim sistemima gde korisnici pristupaju podacima iz različitih geografskih lokacija.
3. **Lokalna dostupnost:** Replikacija podataka na više lokacija omogućava brži pristup korisnicima jer mogu pristupiti najbližoj kopiji podataka, smanjujući latenciju i poboljšavajući ukupne performanse sistema.
   1. **Šta predstavlja konzistentnost podataka?**

Konzistentnost podataka u distribuiranim sistemima predstavlja stanje u kojem sve kopije podataka u sistemu imaju iste vrednosti u svakom trenutku. To znači da svaki korisnik ili aplikacija koja pristupi podacima vidi najnoviju i tačnu verziju tih podataka, bez obzira na to koju repliku koristi. Konzistentnost podataka je ključna za tačnost, pouzdanost i integritet sistema.

* 1. **Opisati model sekvencijalne konzistentnosti.**

### Definicija sekvencijalne konzistentnosti

Sekvencijalna konzistentnost zahteva da rezultat bilo koje operacije (čitanje ili pisanje) bude isti kao da su sve operacije izvršene u nekom sekvencijalnom redosledu. Pritom, operacije svakog pojedinačnog procesa moraju se pojaviti u tom nizu u redosledu koji je specificiran programom tog procesa.

### Kako radi sekvencijalna konzistentnost

Pretpostavimo da imamo više procesa koji pristupaju deljenom skladištu podataka. Svaka operacija čitanja ili pisanja koju izvrši neki proces se evidentira, a svi procesi vide ove operacije u istom redosledu. Redosled operacija ne mora biti vremenski striktan, ali mora postojati neki sekvencijalni redosled koji svi procesi vide isto.

* 1. **Opisati model uslovne konzistentnosti.**

Model uslovne konzistentnosti (causal consistency) je slabiji model konzistentnosti od sekvencijalne konzistentnosti, a njegova ključna karakteristika je pravljenje razlike između događaja koji su potencijalno uzročno povezani i onih koji nisu.

**Osnovni koncepti uslovne konzistentnosti**

1. **Uzročno povezani događaji:** Ako događaj bbb zavisi od ili je uzrokovan događajem aaa, onda svi procesi moraju prvo videti aaa, a zatim bbb. To znači da redosled događaja mora biti isti za sve procese ako su događaji uzročno povezani.
2. **Konkurentni događaji:** Ako su dva događaja nezavisna jedan od drugog (nisu uzročno povezana), oni se nazivaju konkurentni događaji. U tom slučaju, različiti procesi mogu videti ove događaje u različitim redosledima.

**Uslov za uslovnu konzistentnost**

Da bi skladište podataka bilo smatrano uslovno konzistentnim, mora ispunjavati sledeći uslov:

* **Uzročno povezani upisi:** Upisi koji su potencijalno uzročno povezani moraju biti viđeni od strane svih procesa u istom redosledu.
* **Konkurentni upisi:** Konkurentni upisi mogu biti viđeni u različitim redosledima na različitim mašinama.
  1. **Šta predstavlja kritična sekcija?**

Kritična sekcija (critical section, CS) predstavlja deo koda u kojem se izvršavaju operacije čitanja i pisanja na zajedničkim (deljenim) podacima, koje moraju biti zaštićene od konkurentnog pristupa kako bi se izbegli problemi sa konzistentnošću podataka.

* 1. **Opisati model konzistentnosti zasnovanih na podacima kroz primer konzistentnosti na nivou objekata.**

Konzistentnost na nivou objekata je model konzistentnosti koji se primenjuje na nivou pojedinačnih objekata u distribuiranom sistemu. Ovaj model osigurava da sve kopije određenog objekta ostanu konzistentne bez obzira na to koliko je instanci tog objekta replicirano i gde su te instance smeštene.

**Primer konzistentnosti na nivou objekata**

Pretpostavimo da imamo distribuirani sistem koji upravlja zalihama proizvoda u globalnoj mreži prodavnica. Svaka prodavnica ima lokalnu kopiju baze podataka sa informacijama o proizvodima, uključujući stanja zaliha. Svaka kopija baze podataka sadrži objekat "Proizvod" sa atributima kao što su "ID proizvoda", "naziv proizvoda", "cena", i "stanje zaliha".

**Operacije na objektu "Proizvod"**

1. **Čitanje (Read):** Klijent (prodavnica ili aplikacija) može čitati trenutne informacije o proizvodu, uključujući stanje zaliha.
2. **Pisanje (Write):** Klijent može ažurirati stanje zaliha, na primer, nakon prodaje ili dopune zaliha.
   1. **Opisati model monotonog čitanja.**

Monotono čitanje je model konzistentnosti koji se fokusira na ponašanje operacija čitanja u distribuiranim sistemima. Ovaj model garantuje da ako proces pročita određenu vrednost podatka, svaka sledeća operacija čitanja tog podatka od strane istog procesa nikada neće vratiti stariju verziju tog podatka.

Drugim rečima, jednom kada proces vidi neku verziju podatka, neće nikada videti verziju koja je starija od one koju je već pročitao. Ovo obezbeđuje da čitanja podataka uvek slede hronološki redosled kako su se podaci ažurirali.

* 1. **Opisati model "upisi koje prate čitanja".**

Model konzistentnosti "upisi koja prate čitanja" garantuje da će operacija pisanja koja sledi nakon operacije čitanja biti izvršena na kopiji podatka koja je ažurirana sa vrednošću koja je najskorije pročitana od strane tog procesa. Drugim rečima, svaka operacija pisanja koja sledi nakon operacije čitanja će se izvršiti na istoj ili novijoj vrednosti podatka koja je pročitana od strane tog procesa.

Primer:

Pretpostavimo da korisnik čita status poruke na društvenoj mreži, a zatim odgovara na tu poruku.

1. **Korisnik čita poruku:**
   * Čitanje trenutne verzije poruke.
   * Operacija čitanja: R(message\_id) -> current\_status.
2. **Korisnik odgovara na poruku:**
   * Pisanje odgovora na trenutni status poruke.
   * Operacija pisanja: W(response\_id) -> reply\_based\_on\_current\_status.

Ako sistem pruža konzistentnost "upisi koji prate čitanja", korisnikov odgovor će biti zasnovan na trenutnom statusu poruke koji je pročitan u prvom koraku

* 1. **Šta predstavljaju permanente replike? Dati primer.**

Permanentne replike predstavljaju osnovni skup kopija skladišta podataka koje su trajno raspoređene na nekoliko servera. Ove replike su statično konfigurisane i njihov broj je relativno mali. One su ključne za održavanje visoke dostupnosti i pouzdanosti podataka u distribuiranim sistemima.

#### Vrste distribucije

1. **Centralizovana replikacija na jednoj lokaciji:**
   * Fajlovi koji čine web sajt se replikovani na ograničen broj servera smeštenih na jednoj fizičkoj lokaciji.
   * Zahtevi korisnika se prosleđuju jednom od tih servera, često koristeći strategiju rotacije (engl. round-robin).
2. **Ogledanje (mirroring):**
   * Web sajt se kopira na nekoliko servera, poznatih kao ogledala (engl. mirrors), koji su geografski raspoređeni širom interneta.
   * Klijenti biraju jedno od dostupnih ogledala sa liste ili se transparentno prosleđuju jednom od ogledala.
   1. **Opisati replikaciju iniciranu serverom.**

Replikacija inicirana serverom (engl. server-initiated replication) je koncept u distribuiranim sistemima gde se kopije podataka automatski kreiraju ili ažuriraju na osnovu inicijative samog servera ili centralizovanog sistema za upravljanje. Ova vrsta replikacije se često koristi radi optimizacije performansi, smanjenja opterećenja ili poboljšanja dostupnosti podataka u distribuiranim okruženjima.

### Kako funkcioniše replikacija inicirana serverom?

1. **Inicijativa servera:** Serveri ili centralni sistem za upravljanje identifikuju potrebu za kreiranjem novih replika ili ažuriranjem postojećih na osnovu dinamičnih faktora kao što su opterećenje, geografska blizina korisnika ili potreba za skaliranjem.
2. **Automatsko upravljanje:** Replike se kreiraju ili ažuriraju automatski, bez potrebe za direktnom interakcijom od strane korisnika ili drugih sistema. Ovo može biti pokrenuto na osnovu predefinisanih pravila ili algoritama za održavanje performansi i dostupnosti.
3. **Prilagodljivost:** Sistemi za upravljanje mogu dinamički prilagođavati broj i lokaciju replika u skladu sa trenutnim zahtevima i uslovima u mreži. Na primer, tokom vrhunca saobraćaja, server može automatski kreirati privremene replike u blizini korisnika radi smanjenja latencije.

### Primeri primene

* **Web hosting:** Veb server može dinamički inicirati replike sadržaja kako bi efikasno odgovorio na promene u saobraćaju ili zahtevima za geografski raspoređenim sadržajem.
* **Baze podataka:** Centralizovani serveri baza podataka mogu automatski replikovati podatke na dodatne servere radi poboljšanja performansi i održavanja redundantnosti.
* **Mreže za dostavu sadržaja (CDN):** CDN mreže često koriste automatsku replikaciju kako bi distribuirale statički sadržaj na globalnom nivou, povećavajući brzinu isporuke i smanjujući opterećenje na centralne servere.
  1. **Opisati replikaciju iniciranu klijentom.**

Replikacija inicirana klijentom (eng. client-initiated replication) je koncept u distribuiranim sistemima gde klijent samostalno pokreće proces kopiranja podataka radi poboljšanja performansi ili dostupnosti. Ova vrsta replikacije se često koristi kako bi se omogućilo brže pristupanje podacima ili smanjilo opterećenje na centralne servere.

### Funkcionisanje replikacije inicirane klijentom

1. **Inicijativa klijenta:** Klijent prepoznaje potrebu za brzim ili čestim pristupom određenim podacima, te inicira proces kopiranja tih podataka na lokalni keš ili drugu lokalnu mrežu radi bržeg pristupa.
2. **Lokalno skladište (keš):** Klijent čuva kopiju podataka koje je nedavno preuzeo u lokalnom kešu. Ovo može biti na istoj mašini kao klijent ili na drugoj mašini u istoj lokalnoj mreži.
3. **Upravljanje kešem:** Klijent je odgovoran za upravljanje svojim kešem, što uključuje odlučivanje o tome koje podatke treba keširati, kako dugo ih čuvati i kako ih ažurirati ili brisati.
4. **Sinhronizacija sa izvorom:** Klijent povremeno može sinhronizovati svoj keš sa izvornim skladištem podataka kako bi osigurao ažurnost podataka. Ovo može biti automatski ili na zahtev klijenta.
5. **Optimizacija performansi:** Korišćenjem lokalnog keša, klijent može smanjiti latenciju prilikom čitanja podataka jer nema potrebe da se podaci preuzimaju sa udaljenih servera svaki put kada su potrebni.

### Primeri primene

* **Offline rad:** Aplikacije koje omogućavaju offline rad mogu koristiti klijentski keš kako bi omogućile korisnicima pristup podacima čak i kada nisu povezani sa mrežom.
* **Smanjenje opterećenja servera:** Kopiranje podataka na lokalni keš može smanjiti opterećenje na centralni server, posebno u slučajevima kada postoji veliki broj klijenata koji zahtevaju iste podatke.
* **Poboljšanje performansi:** Klijentski keš može značajno poboljšati performanse aplikacija, posebno ako se radi o aplikacijama koje često pristupaju istim podacima.
  1. **Opisati push-based ažuriranje. Dati primer.**

Push-based ažuriranje (protokoli zasnovani na guranju) je koncept u distribuiranim sistemima gde se ažuriranja automatski propagiraju na druge replike ili keševe bez čekanja na zahtev od strane tih replika ili keševa. Ovaj pristup se često koristi za održavanje konzistentnosti podataka u distribuiranim okruženjima gde je brza i konzistentna dostupnost podataka ključna.

### Funkcionisanje push-based ažuriranja

1. **Inicijacija ažuriranja:** Ažuriranje se inicira na centralnom serveru ili izvornom skladištu podataka.
2. **Automatska propagacija:** Nakon što se ažuriranje izvrši na centralnom serveru, ono se automatski šalje ili "gura" na sve druge replike ili keše u sistemu.
3. **Primanje na replikama:** Replike koje primaju ažuriranje ne moraju aktivno tražiti ili inicirati ažuriranje. Umesto toga, ažuriranje se automatski dostavlja na njih.
4. **Konzistentnost podataka:** Push-based pristupi su efikasni jer omogućavaju brzo širenje ažuriranja na sve potrebne lokacije odmah nakon izvršenja ažuriranja na izvornom mestu. Ovo osigurava da sve replike imaju najnovije podatke gotovo u realnom vremenu.
5. **Efikasnost i performanse:** Ovaj pristup je posebno koristan u okruženjima gde postoji visok broj čitanja u odnosu na ažuriranja, jer svako ažuriranje koje se "gura" može biti odmah korisno za jednog ili više čitaoca. Takođe, omogućava efikasno održavanje konzistentnosti podataka bez potrebe za aktivnim zahtevima za ažuriranjem od strane svake replike ili keša.

### Primer primene push-based ažuriranja

* **Distribuirani web sajtovi:** Kada se ažuriranje izvrši na centralnom serveru, svi mirror-i (ogledala) web sajta automatski dobijaju ažurirane podatke putem push operacije. Ovo osigurava da posetioci sajta uvek vide najnoviju verziju bez obzira na to koja ogledala koriste.
* **Veliki sistemi za upravljanje sadržajem:** U sistemima kao što su CMS (Content Management Systems), push-based pristupi omogućavaju da se promene na sadržaju brzo šire na sve servere koji hostuju taj sadržaj, osiguravajući da korisnici vide iste informacije bez obzira na server koji koriste.
  1. **Opiati pull-based ažuriranje. Dati primer.**

Pull-based ažuriranje (protokoli zasnovani na povlačenju) se koristi u distribuiranim sistemima kada replike ili keševi aktivno traže ažuriranja od izvornog skladišta podataka ili centralnog servera. Ovaj pristup je suprotan push-based ažuriranju, gde se ažuriranja automatski šalju na replike bez njihovog zahteva.

### Funkcionisanje pull-based ažuriranja

1. **Inicijacija zahteva:** Replike ili keševi periodično ili po potrebi šalju zahtev za ažuriranjem na izvorno skladište podataka ili centralni server.
2. **Pregled dostupnih ažuriranja:** Izvorno skladište podataka ili server pregleda zahteve i šalje sve ažurirane podatke koji su dostupni.
3. **Primena ažuriranja:** Replike ili keševi prihvataju primljene ažurirane podatke i primenjuju ih lokalno.
4. **Konzistentnost podataka:** Pull-based pristupi su efikasni u okruženjima gde se podaci retko menjaju ili gde je odnos ažuriranja prema čitanjima nizak. Oni osiguravaju da se podaci ažuriraju samo kada je to potrebno, smanjujući opterećenje mrežnih resursa i resursa na skladištu podataka.
5. **Efikasnost i performanse:** Ovaj pristup je koristan u situacijama gde je potrebno minimizirati mrežni promet ili kada je izvorno skladište podataka preopterećeno ažuriranjima. Međutim, može doći do kašnjenja u primeni ažuriranja na replike, posebno u slučaju čestih promena podataka.

### Primer primene pull-based ažuriranja

* **Veb keševi:** Kada veb kešira stranice sa veb servera, može se koristiti pull-based pristup za proveru da li keširani podaci i dalje važe. Na primer, veb keš može periodično ili pri svakom zahtevu proveravati da li su podaci na veb serveru ažurirani od poslednjeg keširanja. Ako jesu, keš povlači novu verziju podataka i ažurira svoj lokalni keš pre nego što odgovori na zahtev korisnika.
* **Sinhronizacija baza podataka:** U distribuiranim bazama podataka, pull-based pristup može se koristiti za sinhronizaciju replika u različitim data centrima. Replike periodično ili po potrebi povlače ažurirane podatke sa centralnog servera ili master baze kako bi održale konzistentnost i integritet podataka.