# Usporedba distribuiranih sustava zasnovanih na objektima i datotekama

**Mavrinac, Nino**

Sveučilište u Rijeci
Filozofski fakultet u Rijeci
Odjel za informatiku
Rijeka

# Usporedba distribuiranih sustava zasnovanih na objektima i datotekama

DIPLOMSKI RAD

Mentor: dr. sc. Božidar Kovačić
Student: Nino Mavrinac
Smjer: informatika/povijest
Akademska godina: 2013./2014.

Rijeka, rujan 2014. god.

**Sadržaj**

# 1.Uvod

Distribuirani sustavi su oni sustavi koji omogućuju obradu podataka na više računala. Konkretno, ovdje ćemo razgovarati o distribuiranim sustavima zasnovanim na datotekama i objektima kao što su Globe, CORBA, Enterprise Java Beans, NFS i RPC. Razlike među njima su velike i u svakom je problem riješen na specifičan način. Mi ćemo proučavati različite načina rješavanja problema i kako ovi distribuirani sustavi komuniciraju, stvaraju replike, objekte i na kraju kako provode sigurnost.

Kroz ovaj diplomski rad nastojat ću pobliže objasniti razlike između distribuiranih sustava zasnovanih na objektima i datotekama. U početku ću definirati osnovne koncepte distribuiranih sustava kao što su objekti, imenovanje klijent-poslužitelj načina rada, te iznjeti neke osnovne razlike između Globe, CORBA i Enterprise Java Beans. Zatim ću opisati proces replikacije i iznjeti razne mogućnosti za provedbu sigurnosti u distribuiranim sustavima kako na objektima tako i na datotekama. Mnogobrojni primjeri koje sam naveo pokušat će što bolje približiti tematiku te uvidjeti pravi problem i rješenje.

# 2. Distribuirani objektno orijentirani sustavi

Za organizaciju distriburanih sustava postoje određena pravila po kojem su oni uređeni. Ta pravila, paradigme, nisu jedinstveno definirana i svaki sustav je uređen na specifičan način. Zato ćemo raspravljati o najboljem načinu korištenja tih paradigmi. Prva se paradigma sastoji od distribuiranih objekata. U distribuirano objektno orijentiranim sustavima pojam objekata igra ključnu ulogu u uspostavljanju distribucije transparentnosti. U principu, sve se tretira kao objekt i klijentima se nude usluge i resursi u obliku predmeta koji se mogu pozvati. Raspodjeljivanje objekata čine važnu paradigmu, jer je relativno lako skrivati distribucijske aspekte iza objekta sučelja. Objekt može biti gotovo sve i to je najmoćnija paradigma za izgradnju sustava. Nadalje, mi ćemo pogledati kako se načela distribuiranih sustava mogu primijeniti na nizu poznatih objektno orijentiranih sustava. Konkretno, u ovom radu, proučavamo CORBA, Globe i Java pokretačke sustave.

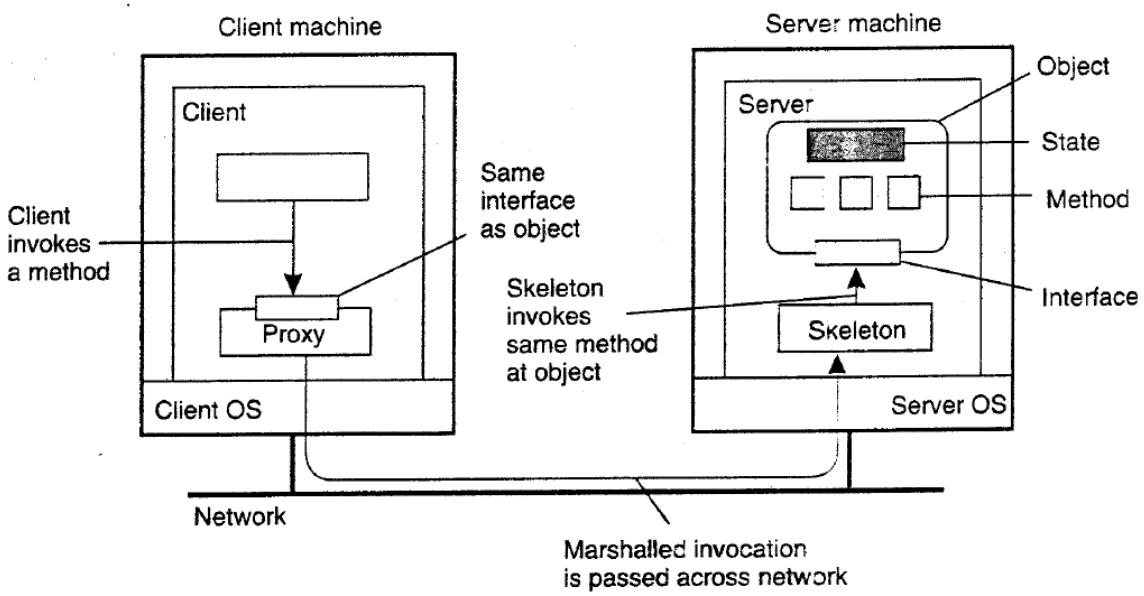
## 2.1. Arhitektura

Objekt orijentacije čini važnu paradigmu u razvoju softvera. Još od svog predstavljanja, on je uživao veliku popularnost. Ova popularnost proizlazi iz prirodne sposobnosti da se izgradi softver koji je dobro definiran sa više ili manje neovisnih komponenta. Programeri se mogu usredotočiti na provedbu specifičnih funkcionalnosti neovisno od drugih *developera*. Objekt orijentacije počeo se koristiti za razvoj distribuiranih sustava 1980-ih. Pojam neovisnog objekta kod domaćina od udaljenog poslužitelja za vrijeme postizanja visokog stupnja transparentnosti distribucije formirale su solidnu osnovu za razvoj nove generacije distribuiranih sustava. U ovom dijelu, prvo ćemo dobiti dublji uvid u opću arhitekturu objektno orijentiranih distribuiranih sustava, nakon čega možemo zaključiti kako su specifična načela raspoređena u tim sustavima.

### *2.1.1. Distribuirani objekti*

Ključno obilježje objekta je da uključuje podatke. Takvo uključivanje podataka zovemo stanje objekata, a operacije nad tim podacima zovemo metodama. Metode su dostupne kroz

sučelje koje korisnik vidi. Korisnik podatke ne doživljava kao objekte već kao informacije prikazane na sučelju. Važno je razumjeti da ne postoji "legalan" način u kojem proces može pristupiti ili manipulirati sa stanjem objekta osim da pozove metode koje su dostupne putem objekta sučelja. Objekt se može izvršiti kroz više sučelja. Isto tako, s obzirom na definiciju sučelja, postoji nekoliko objekata koji nude implementaciju za njega. Razdvajanje između sučelja i objekta koje implementiraju sučelja su presudni za distribuirane sustave. Strogo odvajanje nam omogućuje da stavimo sučelja na jedno računalo, dok se sam objekt nalazi na drugom računalu. Takva organizacijska struktura, koja je prikazana na slici 1, najčešće se spominje kao distribuirani objekti.




*Slika 1.*

Kad se klijent veže na distribuirani objekt implementira se objekt sučelja, koje se zove *proxy*, te je onda on učitan u klijentov adresni prostor. Na klijentovoj strani nalaze se samo sučelje dok se stvarni objekti nalaze na poslužitelj računalu, sučelja su identična i kod klijenta i kod poslužitelja. Poslužitelj i klijent komuniciraju preko *proxy-a* koji je od prije definiran. Poslužitelj pruža osnovna sredstva klijentu koji s tim sredstvima pristupa definiranim objektima. U praksi, ta sredstva su nepotpuni kodovi u obliku jezika posebne klase koje treba dodatno specijalizirati od strane programera, može i korisnik dodatno nadopuniti kod, ali to ovisi o operacijama na sučelju, jer često korisnik nije programer već

on samo uvjetuje stanje koje želi, a operacije koje se nalaze na sučelju predviđene su za generiranje koda i nadopunjuju kod te izvršavaju naredbe klijenta. Najvažnija karakteristika je da stanja distribuiranih objekata nisu raspoređena već je stanje uvijek jedno zbog rada na jednom računalu. Objekte na klijentovoj strani, koji su provedeni pomoću sučelja na klijentovoj strani, nazivamo udaljeni objekti. Generalno gledajući distribuirani objekt može biti fizički distribuiran na više računala, ali ova distribucija je također skrivena od klijenta i nalazi se iza objekta sučelja.

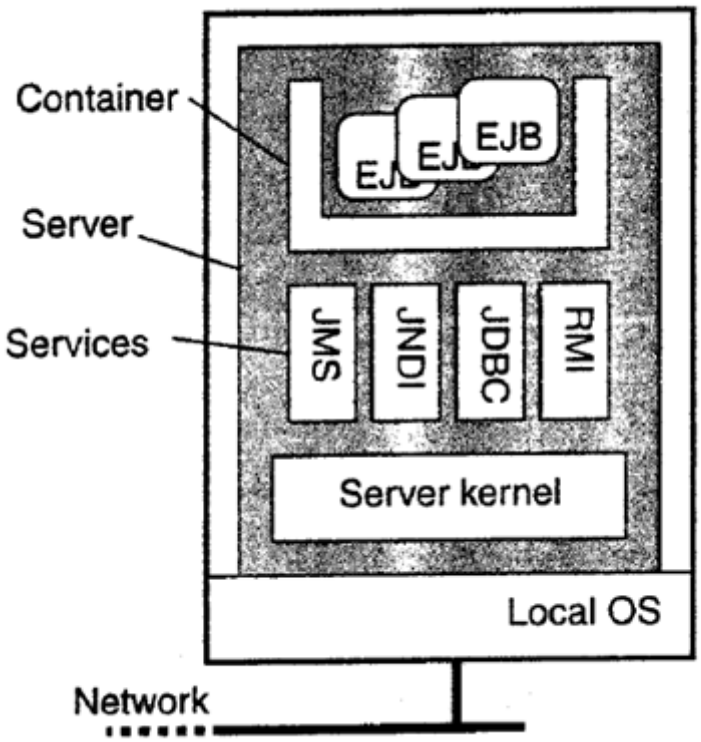
### *2.1.2. Trajni i prijelazni objekti*

Postoje objekti različitih tipova, pa tako oni koji su trajni i prijelazni. Trajni objekt nastavlja postojati čak i ako se trenutno ne nalazi u adresi poslužitelja. Trajni objekt nije ovisan o sadašnjem poslužitelju. U praksi to znači da poslužitelj koji trenutno upravlja trajnim objektom može pohraniti stanje objekta u sekundarnu memoriju. Kasnije novo pokrenuti poslužitelj može pročitati stanje trajnog objekta iz sekundarne memorije i spremiti u vlastiti adresni prostor. Nasuprot njemu je prijelazni objekt koji postoji samo dok je poslužitelj domaćin tom objektu. Iako danas postoje velike dvojbe koji objekti su bolji i dok se ne riješi rasprava o efikasnosti između ova dva objekta većina objektno orijentiranih sustava jednostavno podržava obje vrste.

### *2.1.3. Enterprise Java Beans (EJB)*

Java programski jezik i pripadajući model su temelj za brojne distribuirane sustave i aplikacije. Popularnosti Java jezika može se pripisati zbog jednostavne podrške za objektno orjentirane aplikacije u kombinaciji sa metodom udaljenog pozivanja. Još od predstavljanja Jave postojao je jak poticaj za razvoj distribuiranih aplikacija zbog toga jer nadilazi jezične podrške koje zahtjeva *runtime* okruženje i podržava tradicionalnu klijent poslužitelj arhitekturu.

EJB je zapravo Java objekt koji je organiziran kao poseban poslužitelj sa ponudom da na različite načine pozove objekte za udaljene klijente. Uključuju funkcije koje čuvaju objekte, spremaju, te puštaju objekte tako da budu dio transakcije itd.




*Slika 2.*

EJB je ugrađen u kontejner (slika 2) koji učinkovito pruža sučeljima temeljne usluge koji se provode primjerom poslužitelja. Spremnik ili kontejner može više ili manje automatski povezati EJB poslužitelj sa uslugama, što znači da su ispravne reference dostupne programeru. Tipične usluge EJB-a su: metode udaljenog pozivanja (Remore method invocation - RMI), pristup bazi podataka (database access - JDBC), imenovanje (JNDI) i slanje poruka (JMS). Korištenje ovih usluga su uglavnom automatizirane, ali zahtjeva da programer razlikuje četiri tipa EJB:

- Stateless session beans
- Stateful session beans
- Entity beans

- Message-driven beans

*Stateless session beans* je prijelazni objekt koji se jednom poziva, obavi ono za što je pozvan nakog čega odbacuje bilo kakve informacije povezane za obavljanje usluga koje su ponuđene klijentu. Na primjer, *Stateless session beans*, može se koristiti za implementaciju usluga s popisom top rangiranih knjiga. U ovom slučaju zrnca (*beans*) se obično trebaju sastojati od SQL upita. Rezultat će se staviti u poseban format kojeg klijent može preuzeti nakon kojeg bi njegov rad bio završen, a navedeni podaci izbrisani.

Nasuprot tome *Stateful session beans* održava *client-related* stanje. Tipičan primjer je kad je zrno implementirano za elektronsku trgovačku košaricu koje je najvjerojatnije dostavljeno za elektronsku trgovinu poput E-Bay-a. Zrno kojim upravlja korisnik, može pristupati bazi podataka, stavljati stvari u košaricu, brisati ih iz košarice, prikazati klijentu koliko još ima zaliha u skladištu te upravljati košaricom do elektronske blagajne. Zrno je ipak vremenski limitirano, te kada klijent završi sa operacijama zrno se uništi.

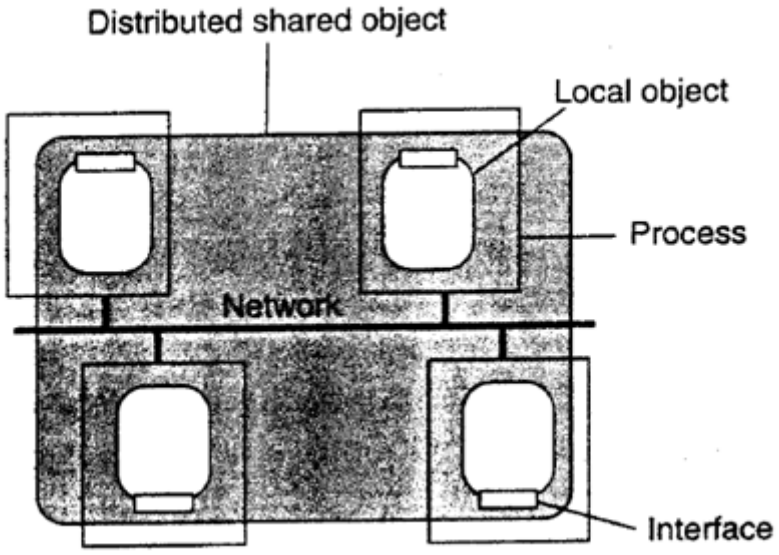
*Entity beans* se može smatrati kao dugovječni, trajni, objekti. Zato će *entity beans* biti spremljen u bazi podataka i bit će dio distribuiranih transakcija. Tako će *entity beans* sadržavati podatke koji će možda trebati sljedeći put kad određeni klijent pristupi poslužitelju. Na primjer, kako bi se što bolje elektronski reklamiralo *entity beans* sadržava podatke o korisniku, kao što su kućna adresa, adresa dostave, podaci sa kreditne kartice itd. Kada se korisnik prijavi automatski mu se dodjeljuje *entity beans* koji je jedinstven samo za njega, te se na ista zrna spremaju novi prikupljeni podaci o korisniku.

*Message-driven beans* koriste se za programske objekte koji bi trebali reagirati na dolazeće poruke, a isto tako i slati poruke. Ne može se izravno pozvati od klijenta nego se poziva javno, kada poslužitelj smatra da je to potrebno. Kada poruka dođe zrno se automatski poziva od strane poslužitelja. Zrno sadrži aplikacijski kod za rukovanje poruka, nakon koje poslužitelj otpusti zrno.

*2.1.4. Globe - raspoređenost zajedničkih objekata*

Globe je sustav u kojem skalabilnost igra središnju ulogu. Globe može podupirati sve aspekte koje se bave izgradnjom velikog, širokog područja sustava te koji može podupirati ogroman broj korisnika i objekata. Temelj sustava je način na koji su objekti sagledani. Kao i drugi objektno orijentirani sustavi od objekata u Globu se očekuje da enkapsuliraju stanje i operacije stanja. Važna razlika od ostalih objektno orijentiranih sustava je da objekti također enkapsuliraju implementaciju koja opisuje distribuciju stanja objekata na više strojeva. Znači svaki objekt određuje kako će se njegovo stanje distribuirati preko vlastite replike. Svaki objekt kontrolira vlastita stanja u ostalim područjima. Tako i veliki objekti u Globu odlučuju kada, kako i gdje će mjenjati svoje stanje. Ista tako odluči li objekt da će se njegovo stanje ponoviti tada on odlučuje da replika objekta na najbolji način zauzme njegovo mjesto.

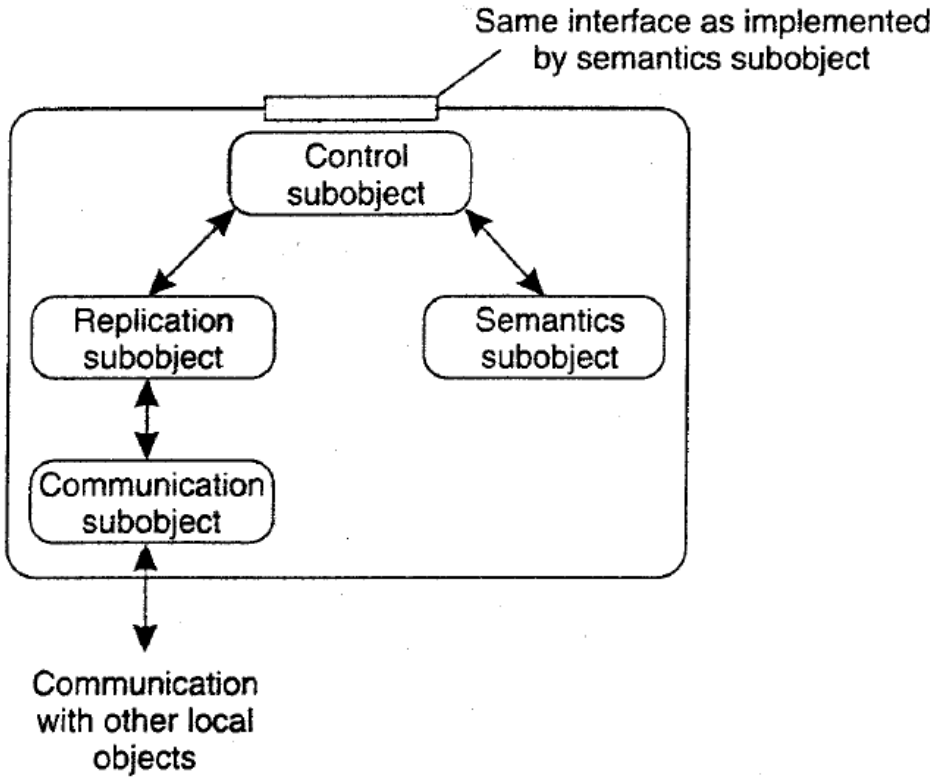
Za razliku od većine drugih objektno orijentiranih distribuiranih sustava, Globe nije usvojio prijenosne objektne modele. Umjesto toga objekti u Globu mogu biti fizički distribuirani, što znači da stanje objekta može biti distribuirano i replicirano kroz više procesa.




*Slika 3.*

Iz slike 3. vidimo da je objekt distribuiran kroz 4 različita procesa, od kojih se svaki odvija na svom računalu. Objekti u Globu se nazivaju distribuirano djeljeni objekti iz razloga da se naglasi da su objekti podjeljenji između nekoliko procesa. Proces koji je vezan za distribuirano-djeljenje objekte nudi lokalnu implementaciju sučelja od tog objekta. Takva lokalna implementacija se zove lokalni objekt. Sve implementacijske detalje objekta su skriveni iza sučelja dostupnih procesu. Jedino što je vidljivo izvan lokalnog objekta su njegove metode.

Globe poznaje dva tipa lokalnih objekata: jednostavan lokalni objekt i složeni lokalni objekt. Jednostavni lokalni objekt je lokalni objekt koji ne sadrži nikakve druge objekte, a za razliku od njega složeni lokalni objekt je objekt koji se sastoji od više objekata, a nekada mogu biti udruženi (slika 4.). Sustav se koristi za izradu lokalnih objekata koji je potreban za implementaciju distribuirano djeljenih objekata.




*Slika 4.*

Semantički podobjekt pokazuje funkcionalnost koja je potrebna za distribuirano djeljene objekte. U biti on odgovara običnim udaljenim objektima. Komunikacijski podobjekt se koristi kao standardno sučelje temeljne mreže. Može funkcionirati i bez spajanja na mrežu. Postoje i mnogo napredniji komunikacijski podobjekti koji se koriste za osiguravanje prikaza na više sučelja. Koriste se i za provedbu pouzdane komunikacije, dok ostali podobjekti nude nepouzdanu komunikaciju. Replikacijski podobjekat je temelj svakog distribuirano djeljenog sustava. Ovaj podobjekat provodi stvarnu strategiju distribuiranja objekata. Kao i kod komunikacijskog podobjekta sučelje mu je stadnardizirano. Replikacijski podobjekat je odgovoran za odlučivanje kada i koju metodu će koristiti semantički podobjekat. Na primjer, replikacijski podobjekat provodi aktivne replikacije, te će on osigurati da svi pozivi metoda se pozovu u isto vrijeme za svaku replikaciju. U tom slučaju podobjekti će morati komunicirati sa replikacijskim podobjektom u drugim lokalnim objektima koji čine distribuirano djeljene objekte. Kontrola subobjekta se provodi između korisničkog sučelja semantičkog podobjekta i standardnih sučelja replikacijskog podobjekta.

### 2.2. Objekt poslužitelj

Ključnu ulogu u objektno orijentiranim distribuiranim sustavima ima objekt poslužitelj koji je poslužitelj dizajniran kao domaćin distribuiranih objekata. Prilagođen je za potporu distribuiranih objekata. Glavna razlika između općih objekta poslužitelja i ostalih (tradicionalnih) poslužitelja je da objekt sam po sebi ne pruža posebnu uslugu. Posebne usluge se provode od strane objekata koji se nalaze na poslužitelju. U biti poslužitelj pruža odgovore samo na pozive koji su poslani sa udaljenih klijenata. Kao posljedica toga, usluge je lako promjeniti na način da se dodavaju ili uklanjaju objekti na poslužitelju. Objekt poslužitelj na taj način djeluje kao mjesto gdje se objekti nalaze.

Objekti se sastoje od dva dijela, jedan dio sadrži stanje u kojem se nalazi, a drugi sadrži kod za izvođenje metode. Objekt poslužitelj odlučuje da li će se dijelovi objekta odvojiti ili ostati zajedno, ako se metode izvode na više objekata tada neće sadržavati informaciju o objektnom stanju.

### *2.2.1. Alternative za pozivanje objekata*

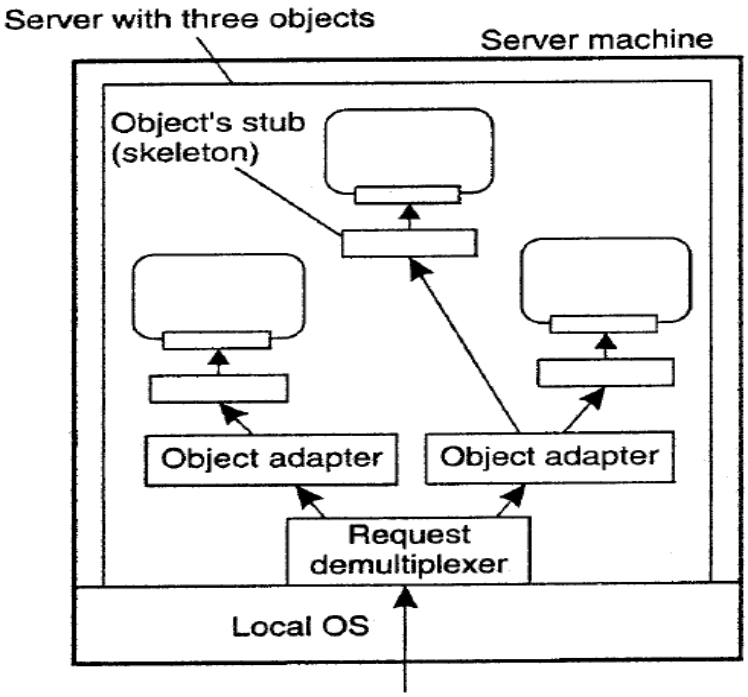
Da bi se objekt pozvao, objekt poslužitelj treba znati koji kod pozvati, s kojim podacima radi, da li treba pokrenuti zaseban slijed događaja i još mnogo drugih stvari. Jednostavan pristup je da pretpostavimo da svi objekti izgledaju isto i da postoji samo jedan način da se pozove objekt. Nažalost, takav pristup je nefleksibilan i često ograničava programere distribuiranih objekata.

Puno bolji način je da poslužitelj podržava više načina za pozivanje objekta. Na primjer prijelaznih objekata ima onoliko koliko i njegovih poslužitelja, no može se napraviti da se prijelazni objekt ugasi čim više nema zahtjeva od strane korisnika. Prednost ovakvog načina je što se štede resursi poslužitelja i objekt postoji samo koliko je to zaista potrebno. Negativna strana ovakvog poziva objekta je da kada ga se prvi put pozove treba mu neko vrijeme dok se implementacija objekta ne izvrši.

Slično tome poslužitelj bi mogao razmjestiti objekte tako da je svaki smješten u svom segmentu. Drugačije rečeno, objekti ne dijele niti kod niti podatke. Takva podjela je nužna kada objekti ne razlikuju kod i podatke ili kada objekti moraju biti odvojeni iz sigurnosnih razloga. Sukladno tome poslužitelj mora osigurati dodatnu potporu za sigurnost cijelog sustava, jer ne smije dozvoliti da se granice koje razdvajaju objekte prekrše, inače bi se mogao ugroziti cijeli sustav. Fizička podjela objekta stoga je najvažniji aspekt svakog poslužitelja. Drukčiji način je da se dozvoli objektima da dijele kod, tako bi se za objekte koji pripadaju jednoj klasi pozivala samo jedna implementacija, pozivajući implementaciju klase samo jednom na poslužitelju. Tako se najčešće pozivaju složeni objekti tj. oni procesi koji zahtjevaju puno informacija, na primjer umjesto da klijent poziva mnogo objekata on pozove samo jednu implementaciju koja u sebi ima nastavak za ostale implementacije, a s time i ostale objekte. Na taj način server ne mora odgovarati i slati objekte zasebno već pošalje cijeli niz objekata. Takvim pozivom štede se resursi poslužitelja i klijenta.

*2.2.2. Objekt adapter*

Odluke kako se pozivaju objekti se najčešće odnose na aktivacijska stanja. Prije nego što se pozovu objekti prvo moraju stići na poslužiteljev adresni prostor. Tu se grupiraju objekti po aktivacijskim stanjima, a sve to radi mehanizam koji se zove objekt adapter (slika 5.). Objekt adapter je softverska implementacija aktivacijskog stanja. Glavni problem je što dolazi kao opća komponenta za pomaganje programerima distribuiranih sustava za specifična stanja. Objekt adapter ima jedan ili više objekata pod svojom kontrolom. Poslužitelj je sposoban odjednom podržavati više objekata koji su potrebni za aktivacijska stanja i zato na poslužitelju mogu djelovati i nekoliko objekta adaptera.



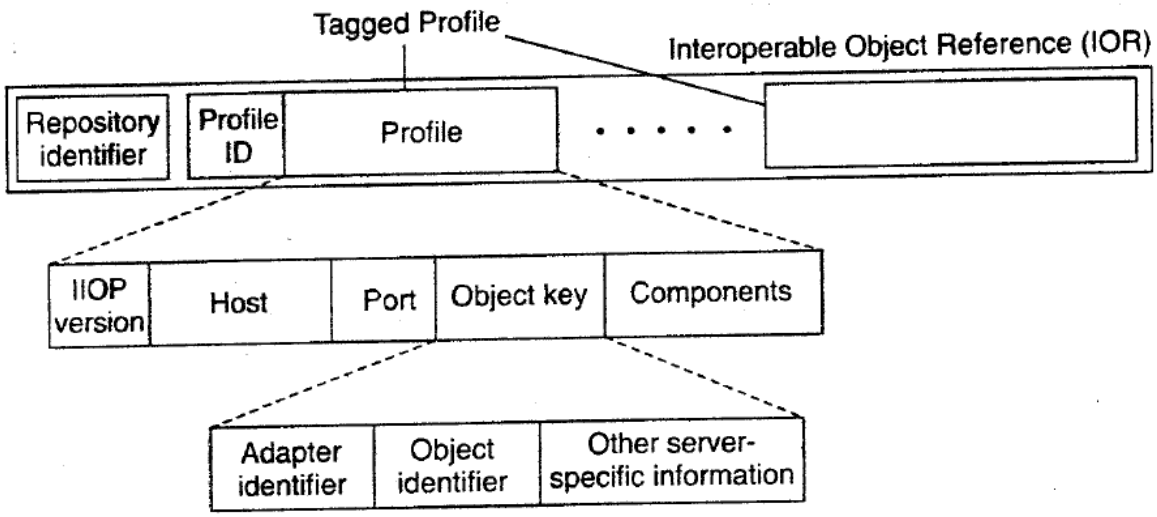

*Slika 5.*

## 2. 3. Imenovanje

### *2.3.1. CORBA objektne reference*

Imenovanje objektno orijentiranih distribucijskih sustava ovisi o tome na koji su način podržane objektne reference. Isto tako možemo vidjeti na CORBA objektnim referencama. Kada klijent drži objektnu referencu, može pozvati metode za implementaciju tih objektnih referenci. Važno je razlikovati objektnu referencu koja koristi klijentov proces za pozivanje metode i one koja je implementirana od RTS-a (*real time system*).

Proces, bilo da je od klijenta ili poslužitelja, može koristiti samo jezično definirane implementacije objektnih referenca. Referenca se ne može prosljediti iz procesa A u proces B, zato što postoji samo u adresnom prostoru procesa A. Umjesto toga proces A dodjeljuje pokazivač u procesno nezavisni prikaz. Operacija koja je zaslužena za to je odobrena od RTS-a. Jednom kad je dodjeljena može se prosljediti procesu B, koji je opet može uzeti kao svoju i briše se zapis da je ona prosljeđena. Trenutni CORBA sustavi podržavaju isti nezavisno jezični prikaz objektne reference koje se zovu *Interoperable Object Reference* ili IOR (slika 6.). CORBA implementacije ne moraju koristiti IOR unutar svog sustava, no IOR je potreban kada se prosljeđuju objektne reference između dva različita CORBA sustava i prosljeđuju se kao IOR. IOR prosljeđuje sve informacije za definiranje objekata.



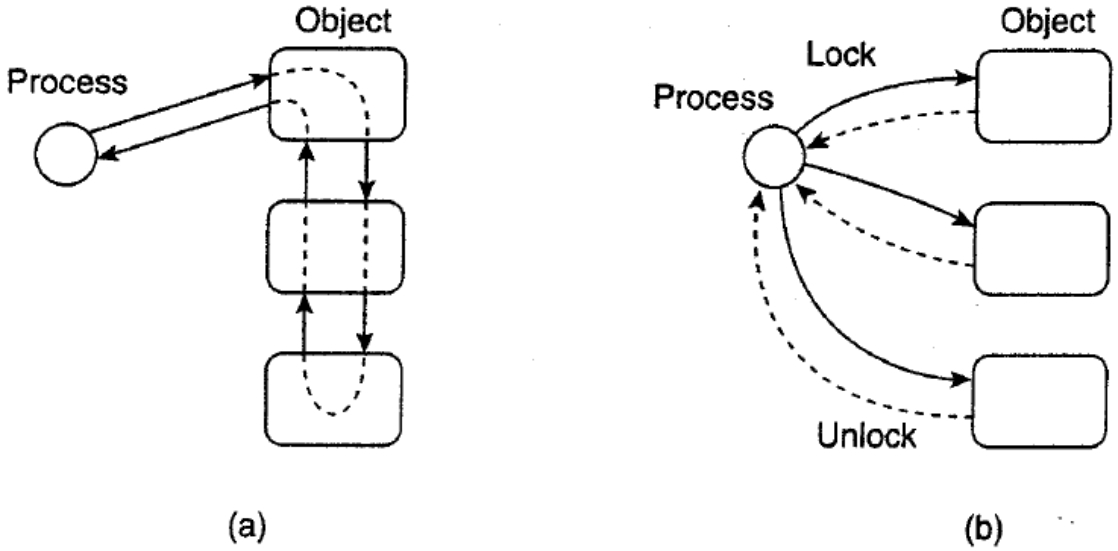

*Slika 6.*

### *2.3.2 Globove objektne reference*

U Globu, svaki distribuirano djeljenji objekt je obilježen kao globalni univerzalni objektni identifikator *(object identifier* - OID). Globov OID se odnosi samo na jedan distribuirano djeljeni objekt i nije nikada ponovo upotrebljen za novi objekt i svaki objekt ima najviše jedan OID. Globov OID se može koristiti samo za usporedbu objektnih referenca. Na primjer, pretpostavimo da procesi A i B su svaki vezani za svoj distribuirano vezani objekt. Svaki proces može pozvati OID objekta za koji su vezani. Ako su dva OID-a ista onde su proces A i proces B vezani za isti objekt. Za razliku od CORBA reference, Globovi OID-ovi se ne mogu koristiti kao izrazvan kontakt sa objektom. Umjesto toga, da bi se locirao objekt nužno je pogledati u adresni prostor službe koji je zadužen za lociranje objekata (*location service*). Služba vraća adresu i Globe može pronaći objekt. Adresa se sastoji od dva dijela. Prvi dio je adresni identifikator pomoću kojeg služba otkriva odgovarajući čvor na kojem se nalazi objekt. Osim toga na čvoru se mogu koristiti i operacije: umetni (*insert*), obriši (*delete*) i pozovi (*recall*). Drugi dio se sastoji od adresnih informacija, ali ove informacije su nevidljive službi. Za službu adrese su samo zapisi i ona ne razlikuje njihovu funkciju

## 2.4. Sinkronizacija

Znamo da su implementacijski detalji skriveni iza sučelja i to može stvoriti problem: kada proces pozove udaljeni objekt, on ne zna da li će poziv pozvati i ostale objekte. Kao posljedica toga možemo imati objekt koji će pozvati i ostale objekte, bez da proces zna za to. To se događa kada su objekti nezaštićeni od otključavanja, tj. nisu zaključani. (slika 7a). Za razliku od toga, radeći sa resursima podataka kao što su datoteke ili tablice baze podataka, koji su zaključani, nadležni proces će imati kontrolu toka. Tako će proces pozvati samo onaj objekt koji mu treba, a ostale koje je slučajno pozvao će biti zaključani i neće se pozvati. Jedina zamjerka ovom sustavu je da proces može pozivati ostale objekte, iako je već pozvao objekt koji mu treba, a pozivanje traje sve do isteka vremena pozivanja, jer on zapravo čeka ostale objekte koje je već odbacio. (slika 7b)




*Slika 7.*

Iz prethodnog primjera vrlo je važno znati kada primjeniti sinkronizaciju, jer nekada želimo pobuditi sve objekte u nizu, a nekada ne. Zato sinkronizacija treba postojatii na objektnom poslužitelju. Ako se istovremeno pozivaju više objekata, poslužitelj ima dovoljno informacija da bi prepoznao kojem procesu treba cijeli niz objekata, a kojem treba samo jedan objekt. Ukoliko je potrebno poslužitelj može zaključati objekte. Nekada se sinkronizacija obavlja i na klijentovoj strani. Na taj način se olakšava rad poslužitelja i smanjuje mogućnost rušenja procesa sa klijentove strane. Poslužitelj ne mora provjeravati i zaključavati objekte već pošalje cijeli niz objekata, a onda klijentova strana odlučuje da li će prihvatiti niz ili samo jedan objekt. Opet ovakav način sinkronizacije ima svojih poteškoća. Ako dva procesa istodobno pozovu metodu, samo će jednom procesu biti omogućeno izvršavanje metode, a drugi će biti blokiran. Još gore od toga je da prvi proces čeka da neki uvjet postane istinit, za to vrijeme drugi proces može pristupiti traženoj metodu, a na taj način oba procesa su blokirani i čekaju da istekne vrijeme poziva objekta (to se može ponoviti i nekoliko puta). Iz tog razloga sinkronizacija se obavlja na poslužitelju, što i nije savršeno rješenje ako se klijentov proces prekine dok traje pozivanje objekta na poslužitelju. Jedino su dizajneri Jave našli prikladno rješenje, tako da dijelovi istih procesa spriječe pozivanje istog udaljenog objekta, a djelovi različitih procesa ne sprječavaju pozivanje istog udaljenog objekta.

## 2.5. Dosljednost i replikacija

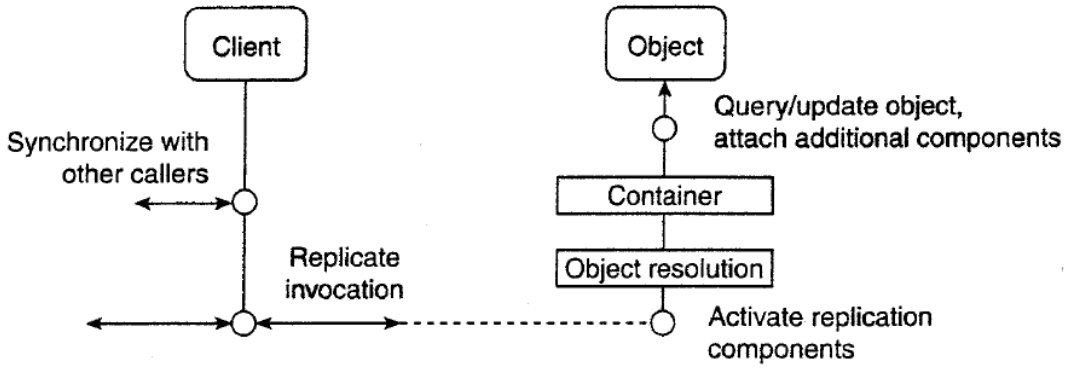
Mnogi distribuirano djeljeni sustavi sljede tradicionalni pristup prema repliciranim objektima, tretirajući ih kao skupove podataka sa njihovim specijalnim operacijama. Kako se objekti sastoje od podataka i operacija tih podataka jednostavno zaključavanje objekata tijekom pozivanja će ih zadržati dosljednim.

Ako ih želimo replicirati postoje dva glavna problema. Ako se jedna metoda objekta izvršava ne smijemo dopustiti niti jednoj drugoj metodi objekta da se izvrši. Takav način omogućava da su unutrašnji podaci objekta nepromjenjeni. Drugi problem je ukoliko repliciramo objekt moramo osigurati s time da sve promjene repliciranih stanja ostanu jednake. Drugačije rečeno, moramo osigurati da dvije nezavisne metode pozivanja zauzmu mjesto na različitim replikama u isto vrijeme. Takav način je jedino moguće riješiti tako da svaka replika pozivanje vidi na isti način.
Repliciranje se vrši na način da se koriste takozvani presretači za repliciranje Java zrnca (u konkretnom primjeru za Java J2EE poslužitelje). Ideja je da se pozivi objekta presretnu na tri različita mjesta:

- Na klijentovoj strani neposredno prije nego što poziv je prosljeđen
- Klijentovom dijelu gdje presjecanje je dio replikacijskog algoritma
- Na poslužiteljovoj strani, neposredno prije nego će objekt biti pozvan

Prvo presjecanje je potrebno kada se ispostavi da je poziv repliciran. U tom slučaju sinkronizacija sa drugim pozivima je potrebna da se vidi da pozivi nisu replicirani. Jednom kad se odluči da se poziv može izvesti presjecatelji koji su dijelovi replikacijskog algoritma odlučuju gdje će prosljediti zahtjev ili će zahtjev vratiti tako da replika ne može biti dosegnuta. U konačnici poslužitelj strana kontrolira presjecanje i upravlja pozivom. Zadnji presretači su podjeljeni na dva djela dijela. Prvi dio se izvodi nakon što je poziv došao i preuzima ga algoritam za repliciranje prije nego će ga predati objekt adapteru. Algoritam ispituje od koga je došao zahtjev te ga odobrava, a ako je potrebno aktivira bilo koji replikacijski objekt potreban da bi se dovršila replikacija. Drugi dio je upravo prije poziva

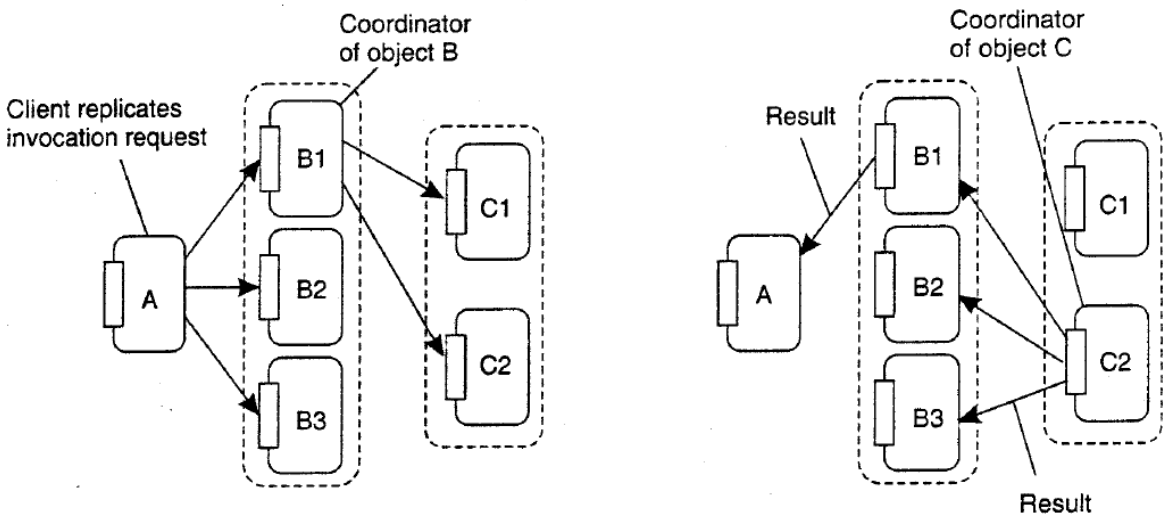
tj. buđenja objekta gdje se dozvoljava algoritmu da uzme i namjesti vrijednosti repliciranog objekta ili bilo kojeg drugog podatka vezanog za taj objekt (Slika 8.).




*Slika 8.*

*2.5.1. Odgovori na pozive*

Uzmimo na primjer da objekt A poziva objekt B, taj objekt B poziva objekt C. Ako je B repliciran, sva replika od objekta B će zasebno pozvati objekt C. Problem je dakle što se objekt C poziva više puta, umjesto samo jednom. Konkretno gledajući ako je objekt C transfer u iznosu 100 000 kn prije ili kasnije će se netko žaliti. Jedno od rješenja je da se jednostavno zabrani repliciranje poziva što ima smisla kada je izvedljivost upitna. Kao što vidimo u slici 9. drugo rješenje je da se uspostavi novi komunikacijski sloj na vrhu repliciranih objekta. Bit je da taj komunikacijski sloj upravlja pozivima i repliciranim pozivima. Kada replicirani objekt B pozove replicirani objekt C, pozivi će se prosljediti u isto vrijeme, ali je na svakoj replici B pridodan jedinstveni identifikator. Kordinator koji se nalazi u komunikacijskom sloju zatim upravlja replikacijama objekta B i dopušta samo jednom repliciranom objektu B1 da uputi poziv na objekt C dok ostali replicirani objekti B čekaju. Rezultat je samo jedan zahtjev koji je prosljeđen repliciranom objektu C, koji zatim odgovara na poziv, na isti način.




*Slika 9.*

## 2.6. Sigurnost

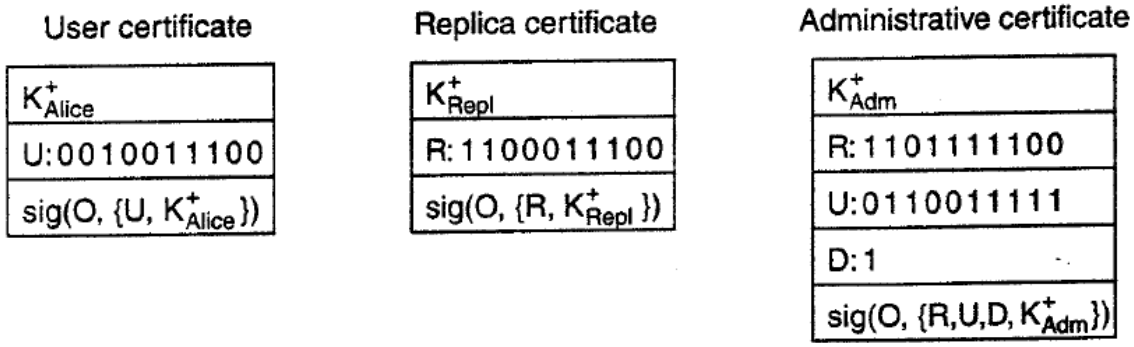
Sigurnost ima važnu ulogu u bilo kojem distribuiranom sustava pa tako i objektno orjentiranom. Činjenica je da su većina distribuiranih objekata, udaljeni objekti te se sve strukture sigurnosti zaštićuju na sličan način. Svaki objekt je zaštićen standardnim autentičnim i autoriziranim načinom.

### *2.6.1. Sigurnost u Globu*

Globe je jedan od rijetkih distribuiranih objektno orijentiranih sustava u kojem su stanja objekta fizički razdjeljena i replicirana na više računala. Takva arhitektura Globa donosi niz komplikacija kada je u pitanju sigurnost ali i specifična rješenja za takve probleme. Na primjer gledajući poziv za metodu udaljenog objekta barem su dva važnja pitanja što se tiče sigurnosti: Je li pozivatelj pozvao ispravan objekt i je li pozivatelju dozvoljeno da poziva tu metodu. U Globu je razvijena platforma sigurnost koja štiti objekte. Platforma na koju je kopiran objekt štiti objekt od zlonamjernog koda sadržanog u objektu (ako postoji takav kod) i brine se na način kako zaštititi objekt od zlonamjerne replike poslužitelja.

U Globu je dozvoljeno i kopiranje objekata drugim domaćinima što opet izaziva novi problem. Objekt poslužitelj koji je domaćin kopiranom objektu nije uvijek pouzdan i ne

smije mu se vjerovati, zato mora postojati mehanizam koji spriječava da svaki replicirani poslužitelj koji je domaćin objektima izvršava bilo koju objektnu metodu. Na primjer, vlasnik objekta želi ograničiti izvršavanje nekih metoda na mali broj repliciranih poslužitelja, a ostali poslužitelji imaju samo opciju za čitanje informacija s objekta, a ne za mjenjanje njegovog stanja. Samo pouzdan poslužitelj smije mjenjati stanje tog objekta. Postoje više načina rješavanja tog problema. Svaki objekt u Globu ima svoj privatni i javni ključ koji se naziva objektni ključ. Svatko tko zna privatan ključ objekta može pristupiti objektu i mjenjati stanje ovisno o uslugama klijenata i poslužitelja. Svaka replika ima svoj pripadajući replicirani ključ, koji se također sastoji od javnog i privatnog ključa. Takav ključ se generira od poslužitelja koji je trenutno domaćin određene replike. Svaki korisnik ima svoj jedinstveni javni i privatni ključ poznat kao korisnički ključ. Postoje tri razine ovlaštenja, a s time i tri razine sgurnosti: klijentova, replicirana i administracijska razina. Svako ovlaštenje je drugačije, tako će administracijski ključ imati najviše ovlasti, a replicirani najmanje. Slika 10. prezentira tri različita ključa: administrativni, korisnički i replicirani. Kao što vidimo administrativni ključ ima najveće ovlasti i može mjenjati bilo koji objekt. Korisnički i njemu replicirani ključ imaju manje ovlasti a s time manje mogućnosti u alternaciji objekta

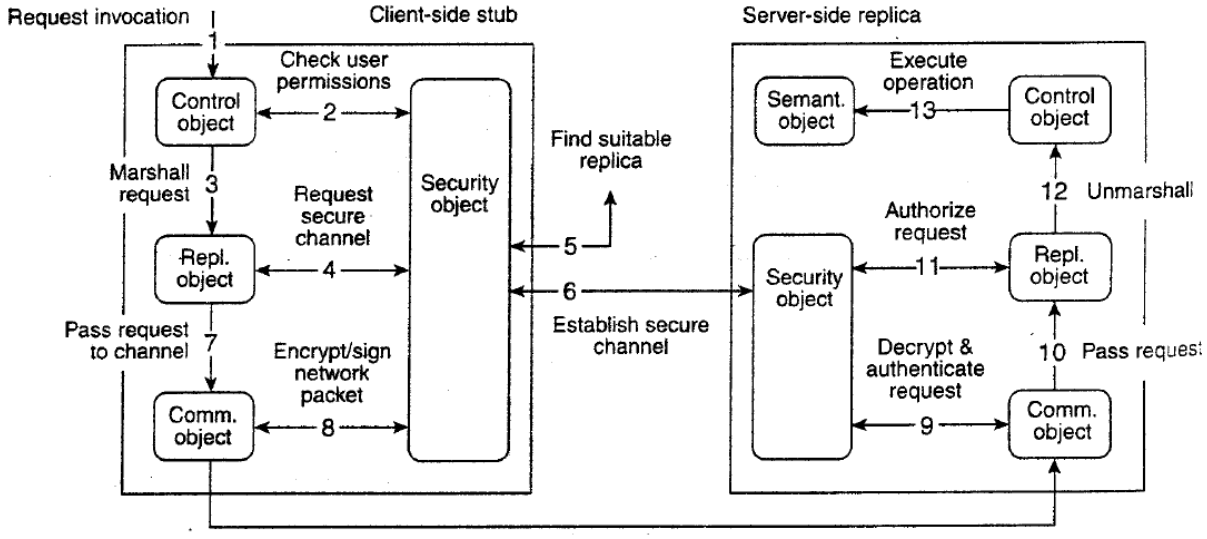



*Slika 10.*

### *2.6.2. Sigurnost kod metoda pozivanja objekata*

Sigurnosna metoda za pozivanje se izvodi u 13 koraka koji su prikazani na slici 11. Lijeva strana slike prikazuje operacije na korisničkoj strani, a desna na poslužitelj strani:

- Prvo aplikacija izdaje zahtjev za pozivanje, lokalno pozivajući metodu. (korak 1 na slici)
- Kontola podobjekata provjerava korisničke ovlasti zajedno sa informacijama koji su spremljeni na loklani objekt. (korak 2)
- Zahtjev je zapakiran i predan dalje. (korak 3)
- Replikacijski podobjekat zahtjeva od središnjeg sloja (middleware) da uspostavi sigurnosti kanal za odgovarajuću repliku. (korak 4)
- Sigurnosni objekt prvo pokreće repliku pretraživanja. Da bi se to postiglo, uzima bilo koju uslugu za imenovanje koja može pretraživati metode potrebne za izvršavanje zahtjeva. Globova služba (location service) je modificirana za takve potrebe i najčešće se koristi za ovaj korak. (korak 5)
- Jednom kada je nađena odgovarajuća replika, sigurnosni podobjekt može uspostaviti sigurnosni kanal. Na ovom koraku se i provjerava da li je replici dozvoljeno izvršavanje pozivanje objekata. (korak 6)
- Zahtjev je prosljeđen na komunikacijski podobjekat. (korak 7)
- Podobjekat šifrira i potpisuje zahtjev tako da može proći kroz sigurnosni kanal. (korak 8)
- Kada zahtjev pristigne, dešifrira se i potvrđuje se njegova autentičnost. (korak 9)
- Zahtjev je zatim predan na replicirani objekt kod poslužitelja(korak 10)
- Još jednom se provjerava autentičnost gdje se gledaju ovlasti klijenta i je li u mogućnosti izvršiti poziv i operaciju. (korak 11)
- Zahtjev se raspakira. (korak 12)
- Izvršava se operacija. (korak 13)


Request invocation
Client-side stub
Server-side replica
1
Check user permissions
Control object
2
Security object
Find suitable replica
Execute operation
Semant. object
13
Control object
Marshall request
3
Request secure channel
Repl. object
4
5
6
Establish secure channel
Security object
Authorize request
11
Repl. object
12
Unmarshall
Pass request to channel
7
Encrypt/sign network packet
Comm. object
8
Decrypt & authenticate request
9
10
Pass request
Comm. object

*Slika 11.*

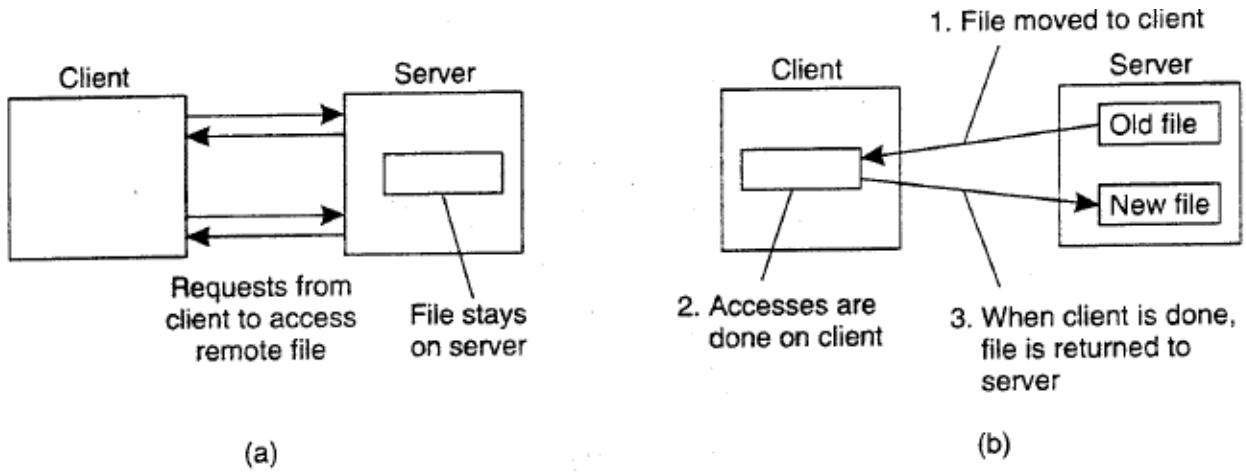
# 3. Distribuirani sustavi datoteka - arhitektura

Započinjemo našu raspravu o distribuiranim sustavima zasnovanim na datotekama gledajući kako su oni uglavnom organizirani. Većina sustava izgrađena je nakon tradicionalne klijent poslužitelj arhitekture, ali potpuno decentralizirana rješenja postoje, te ćemo ih nastojati objasniti u nastavku.

## 3.1. Klijent Poslužitelj Arhitektura

Mnogi distribuirani sustavi datoteka su organizirani po uzoru na klijent poslužitelj arhitekture sa *Network File System* (NFS) mikrosustava *Sun* kao jedno od najčešće korištenih za UNIX sustave. Uzet ćemo NFS kao kanonski primjer za distribuirane sustave datoteka temeljene na poslužiteljima. Konkretno, koncentrirati ćemo se na NFS v3, na široko rasprostranjenu treću inačicu NFS i NFS v4, najnoviju, četvrtu verziju. Razmotriti ćemo razlike između njih. Osnovna ideja je da svaki NFS poslužitelj datoteka pruža standardizirani pogled na svom lokalnom datotečnom sustavu.

Drugim riječima, to ne bi trebalo biti bitno kako je lokalni datotečni sustav implementiran; svaki NFS poslužitelj podržava isti model. Ovaj pristup je također usvojen za druge distribuirane sustava datotek. NFS dolazi s komunikacijskim protokolom koji omogućava klijentima pristup datotekama pohranjenim na poslužitelju, čime se omogućuje heterogena kolekcija procesa, koji možda rade na različitim operativnim sustavima i strojevima, kako bi se podijelio zajednički datotečni sustav. Model temeljnih NFS i sličnih sustava je onaj udaljenih usluga datoteka. U tom modelu, klijentima se nudi transparentan pristup datotečnom sustavu kojim se upravlja pomoću daljinskog poslužitelja. Međutim, klijenti su obično nesvjesni stvarnog položaja datoteke. Umjesto toga, njima je ponuđno sučelje za datotečni sustav koji je sličan sučelju koji nudi konvencionalnan lokalni sustav datoteka. Konkretno, klijentu se nudi samo sučelje koje sadrži razne operacije datoteka, ali poslužitelj je odgovoran za provedbu tih operacija. Ovaj model također se naziva model udaljenog pristupa.

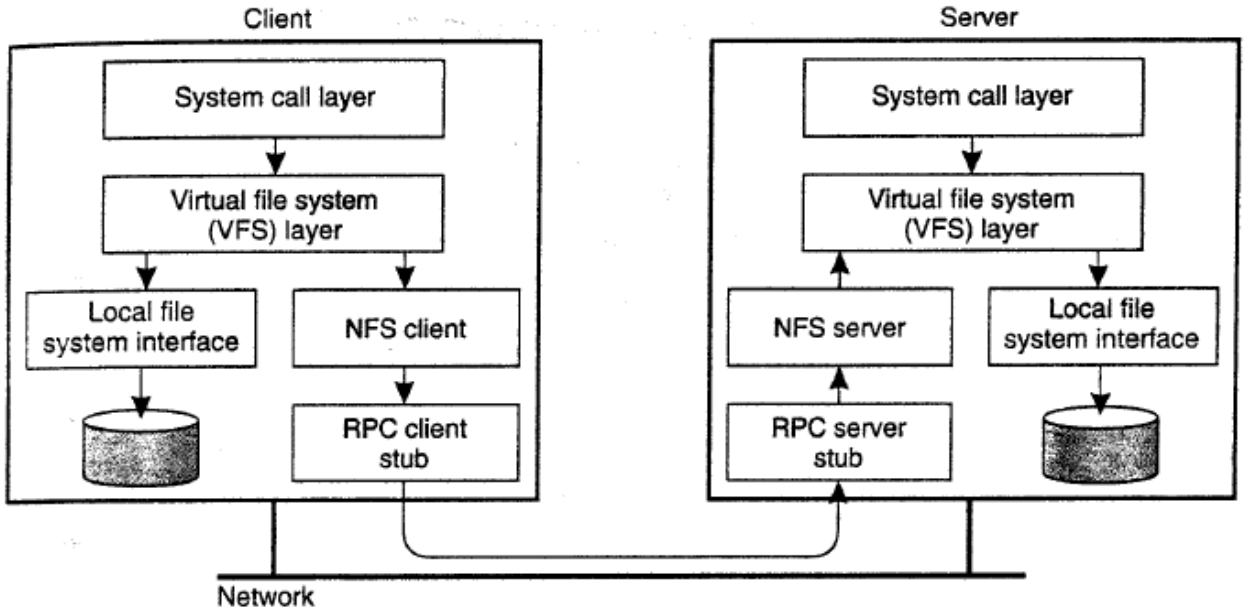



*Slika 12.*

Nasuprot tome, u *upload/download* modelu klijent pristupa datotekama na lokalnoj razini, nakon što ih skine s poslužitelja. Kada je klijent završio sa spisom, ponovno ga učita na poslužiteljea, tako da se može koristiti od strane drugog klijenta. FTP poslužitelj Internet može se koristiti na ovaj način kada klijent preuzima potpunu sliku, mijenja je, a zatim stavlja natrag. NFS je proveden na velikom broju različitih operativnih sustava a pogotovo prevladavaju na UNIX sustavima. Na gotovo svim modernim UNIX sustavima, NFS se po pravilu provodi nakon slojevite arhitekture prikazan na slici 13. Klijent pristupa sustavu datoteka pomoću poziva sustava koji mu omogućuje njegov lokalni operativni sustav. Međutim, lokalno sučelje UNIX sustav datoteka zamjenjuje se sučeljem *Virtual File System* (VFS), koja je do sada bio standard za povezivanje s različitim (distribuiranim) sustavima datoteka.

Gotovo svi suvremeni operativni sustavi pružaju VFS, a ako ga ne podržavaju svejedno prisiljava programere da uglavnom ponovno implementiraju ogromni dio operativnog sustava pri usvajanju nove strukture sustava datoteka. S NFS operacije na VFS sučelju su ili dodane na lokalnom sustavu datoteka, ili dodane na zasebnoj komponenti poznatoj kao NFS klijenta, koja se brine o rukovanju pristupnim datotekama pohranjenim na udaljenom poslužitelju. U N'FS-u sva komunikacija klijent poslužitelj se obavlja putem RPC-a. NFS klijent provodi operacije NFS datotečnog sustava kao RPCs na poslužitelj. Imajte na umu

da se operacije koje nudi VFS sučelje mogu razlikovati od onih koje nudi NFS klijent. Cijela ideja VFS-a je sakriti razlike između različitih datotečnih sustava.




*Slika 13.*

Na strani poslužitelja, vidimo sličnu organizaciju. NFS poslužitelj je odgovoran za upravljanje dolaznim zahtjevima klijenata. RPC stub konvertira zahtjeve i NFS poslužitelj ih pretvara u redovite VFS operacije datoteka koje su naknadno donesene na VFS. Opet, VFS je odgovoran za provedbu lokalnog sustava datoteka na kojem su pohranjene stvarne datoteke. Bitna prednost ovog sustava je u tome što NFS je uvelike neovisan o lokalnim sustavima datoteka. U principu, stvarno nije važno da li operacijski sustav na klijentu ili poslužitelju provodi UNIX sustav datoteka, Windows 2000 sustav datoteka, ili čak stari MS-DOS sustav datoteka. Jedino važno pitanje je da su ti sustavi datoteka u skladu s modelom sustav datoteka koji nudi NFS. Na primjer MS-DOS sa svojim kratkim imenima datoteka ne može se koristiti za provedbu NFS poslužitelja na potpuno transparentan način.

### *3.1.1.Model sustava datoteka*

Model sustava datoteka koji nudi NFS je gotovo isti kao i onaj koji nudi UNIX sustav. Datoteke se tretiraju kao neinterpretirani nizovi bajtova. Oni su hijerarhijski organizirani u graf imenovanja u kojem čvorovi predstavljaju direktorije i datoteke. NFS također podržava čvrste veze, kao i simboličke veze, kao bilo koji UNIX sustav datoteka. Datoteke se imenuju, ali im se pristupa pomoću UNIX datoteke. Drugim riječima, za pristup datoteci klijent prvo mora potražiti svoje ime unutar usluge imenovanja i dobiti pripadajući *handle* (broj) datoteke. Nadalje, svaka datoteka ima niz atributa čije vrijednosti se mogu pogledati i mijenjati. Slika 14. pokazuje opće operacije nad datotekama podržane od strane NFS verzija 3 i 4.

Operacija stvaranja se koristi za stvaranje datoteka, ali ima nešto različito značenje u NFS v3 i NFS v4. U verziji 3, operacija se koristi za stvaranje redovite datoteke. Posebne datoteke su stvorene pomoću zasebne operacije. Veza operacija se koristi za stvaranje čvrste veze. Simbolička veza se koristi za stvaranje simboličke veze. Mkdir (komanda koja služi za stvaranje direktorija) se koristi za stvaranje podmape. Posebne datoteke, kao što su datoteke uređaja (*device files*) stvorene su pomoću operacije mknod. Ova situacija se mijenja u potpunosti kod NFS v4, gdje se stvaranje koristi za stvaranje neredovitih datoteka, koje uključuju simboličke veze, direktorije, i posebne datoteke. Tvrdi linkovi još uvijek se stvaraju korištenjem zasebnih veze operacija, ali redovite datoteke se stvaraju pomoću otvorene operacije, što je novo za NFS i predstavlja veliku devijaciju od pristupa do rukovanja datotekama u starijim verzijama. Do verzije 4, NFS je osmišljen kako bi omogućio svojim poslužiteljima datoteka da budu *stateless*.

Datoteke se brišu pomoću operacije uklanjanja. U verziji 4, ova operacija se koristi za uklanjanje bilo koje vrste datoteke. U prethodnim verzijama, odvojena rmdir operacija je potrebna za uklanjanje poddirektorija. Datoteka je uklonjena po nazivu i ima učinak da se broj tvrdih linkova na njemu smanjuje za jedan. Ako broj linkova padne na nulu, datoteka može biti uništena. Verzija 4 omogućuje klijentima da otvaraju i zatvaraju (redovne) datoteke. Otvaranje nepostojeće datoteke ima nuspojavu da se stvara nova datoteka. Da

biste otvorili datoteku, klijent joj daje ime, zajedno s različitim vrijednostima atributa. Na primjer, klijent može odrediti da bi datoteka trebala biti otvorena za pisanje. Nakon što je datoteka uspješno otvorena, klijent može pristupiti toj datoteci pomoću svog *file handle-a*. Taj *handle* se također koristi kako bi zatvorili datoteku, čime klijent govori poslužitelju da više neće trebati imati pristup datoteci. Poslužitelj može objaviti bilo koje stanje koje je održavao kako bi osigurao da klijent pristupa datoteci.

| Operation | v3 | v4 | Description |
|---|---|---|---|
| Create | Yes | No | Create a regular file |
| Create | No | Yes | Create a nonregular file |
| Link | Yes | Yes | Create a hard link to a file |
| Symlink | Yes | No | Create a symbolic link to a file |
| Mkdir | Yes | No | Create a subdirectory in a given directory |
| Mknod | Yes | No | Create a special file |
| Rename | Yes | Yes | Change the name of a file |
| Remove | Yes | Yes | Remove a file from a file system |
| Rmdir | Yes | No | Remove an empty subdirectory from a directory |
| Open | No | Yes | Open a file |
| Close | No | Yes | Close a file |
| Lookup | Yes | Yes | Look up a file by means of a file name |
| Readdir | Yes | Yes | Read the entries in a directory |
| Readlink | Yes | Yes | Read the path name stored in a symbolic link |
| Getattr | Yes | Yes | Get the attribute values for a file |
| Setattr | Yes | Yes | Set one or more attribute values for a file |
| Read | Yes | Yes | Read the data contained in a file |
| Write | Yes | Yes | Write data to a file |

*Slika 14.*

Operacija pretraživanja se koristi za traženje *handle* datoteke za određeni put imenovanja. U NFS v3, operacije pretraživanja neće riješiti imenovanje, s obzirom da je točka montiranja katalog koji u suštini predstavlja link na poddirektorij u prostoru za strano ime. Na primjer, pretpostavimo da se ime */remote/vu*... odnosi na točku montiranja u imenovanju

grafa. Prilikom rješavanja imena */remote/vu/mbox*... pretraživanje operacija u NFSv3 će vratiti datoteke na točku montiranja */remote/vu* zajedno s ostatkom imenom (odnosno, mbox). Klijent je tada dužan da izričito montira sustav datoteke koji je potreban za dovršetak imena pretraživanja. Sustav datoteka u ovom kontekstu je skup više datoteka, atributa, direktorija i podatkovnih blokova koji se zajednički provode kao logičan blok uređaja.

U verziji 4, stvari su pojednostavljene. U ovom slučaju, traženje će pokušati rješiti cijelo ime, čak i ako to znači prelazak točke montiranja. Imajte na umu da je ovaj pristup moguć samo ako je sustav datoteka već montiran na točke montiranja. Klijent je u mogućnosti otkriti da je pređena točka montiranja uvidom u identifikator sustava datoteka koji se kasnije vratio kad pretraživanje završi. Tu je zasebna operacija Readdir kojom se čitaju unosi u direktorij. Ova operacija vraća popis (ime, *file handle*) parova zajedno s vrijednostima atributa koje klijent traži. Klijent također može odrediti koliko se unosa treba vratiti. Operacija vraća pomak koji se može koristiti u sljedećem pozivu na Readdir kako bi se čitali sljedeći nizovi prijava.

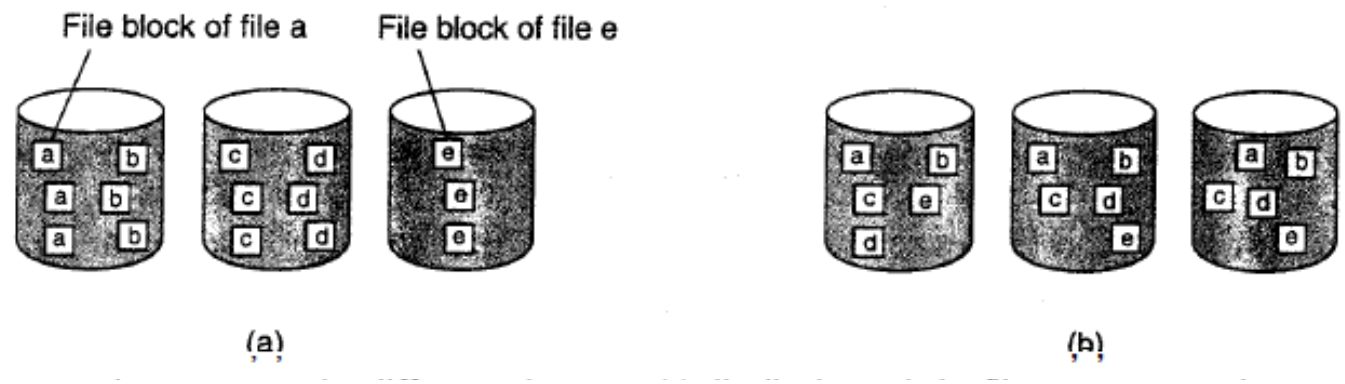
Operacija Readlink se koristi za čitanje podataka koji su povezani sa simboličkom vezom. Normalno, ovi podaci odgovaraju stazi lokaciji imena koja se može naknadno pronaći. Imajmo na umu da operacije pretraživanja ne mogu nositi simboličke veze. Umjesto toga, kada je postignuta simbolična veza, rješavanje imena se zaustavlja i klijent je dužan prvo pozvati readlink kako bi saznao gdje se rezolucija imena treba nastaviti. Datoteke imaju različite atribute povezane s njima. Opet, postoje značajne razlike između NFS verzije 3 i 4. Tipične osobine su tip datoteke (koji nam govori radi li se s direktorijom, simboličkom vezom, posebnom datotekom, itd), duljina datoteke, identifikator sustava datoteke koji sadrži datoteku, te identifikator kada je posljednji put datoteka mijenjana i prilagođena. Atributi datoteka mogu se pročitati i postaviti pomoću operacije Getattr i Setattr, odovojeno. Konačno, tu su i poslovi za čitanje podataka iz datoteke i pisanje podataka u datoteku. Čitanje podataka pomoću operacije Read je potpuno jasno.
Naručitelj navodi *offset* i broj bajtova koje treba pročitati. Klijentu se vraća stvarni broj bajtova koje su pročitani, zajedno s dodatnim informacijama o stanju (primjerice, da li je

postignuta datoteka *end-of*). Upisivanje podataka u datoteku se obavlja pomoću operacije pisanja. Klijent ponovno određuje poziciju u datoteci u kojoj pisanje treba početi, broj bajtova koji treba biti napisan, i podaci. Osim toga, to može uputiti poslužitelja kako bi se osiguralo da svi podaci moraju biti napisani na stabilno mjesto za pohranu. NFS poslužitelji moraju podržavati uređaje za pohranu koji mogu preživjeti kvarove napajanja, kvarove operativnog sustava i hardverske kvarove.

### *3.1.2. Distribuirani sustavi datoteka temeljeni na clusteru*

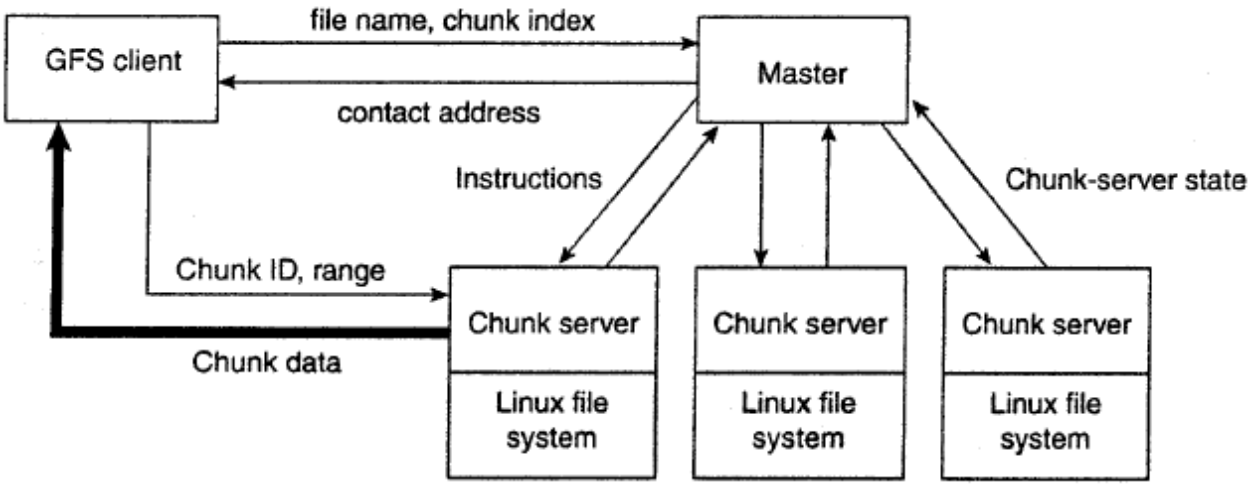
NFS je tipičan primjer za mnoge distribuirane sustave datoteka, koje su uglavnom organizirane prema tradicionalnim klijent poslužitelj arhitekturama. Ova arhitektura je često pojačana za poslužiteljske klastere s nekoliko razlika. S obzirom da se server klasteri često koriste za paralelne aplikacije, pa i ne čudi da su njihovi povezani sustavi datoteka sukladno tome prilagođeni. Jedna dobro poznata tehnika je da iskoristite tehnike file-iscrtavanje, kojim se jedna datoteka distribuira na više poslužitelja. Osnovna ideja je jednostavna: raspodjelom velikih datoteka na više poslužitelja, postaje moguće dohvatiti različite dijelove paralelno. Naravno, takva organizacija radi dobro samo ako se primjena organizira na takav način da paralelni pristup podacima ima smisla. To obično zahtjeva da podaci pohranjeni kao datoteke imaju vrlo pravilne strukture, na primjer gusta matrica.

Za opće namjene, ili one s nepravilnim ili više različitim vrstama struktura podataka, podijela datoteke možda i nije učinkovita. U tim slučajevima često je više prikladno podijeliti sustav datoteka i jednostavno pohraniti različite datoteke na različitim poslužiteljima, ali ne podijeliti jednu datoteku na više poslužitelja (slika. 15.). Zanimljiviji su slučajevi organiziranja distribuiranih sustava datoteka za velike podatkovne centre poput onih koje se koriste u tvrtkama kao što su Amazon i Google. Te tvrtke nude usluge web klijentima što je rezultiralo čitanjem i ažuriranjem golemog broja datoteka distribuiranih preko desetka tisuća računala. U takvim sredinama, tradicionalne pretpostavke vezane za distribuirane sustave datoteka više ne vrijede. Na primjer, možemo očekivati da će u svakom trenutku biti neko računalo koje je neispravno.




*Slika 15.*

Za rješavanje tih problema, Google je, primjerice, razvio svoj Google sustav datoteka (GFS – *Google File System*). Google datoteke znaju biti jako velike, obično u rasponu do više gigabajta, gdje svaka sadrži puno manje objekte. Štoviše, ažuriranja datoteke obično se rade dodavanjem podataka umjesto zamjene dijelova datoteke. Ova zapažanja, zajedno s činjenicom da su server kvarovi norma, a ne iznimka, dovode do izgradnje klaster servera kao što je prikazano na slici. 16




*Slika 16.*

Svaki GFS klaster se sastoji od jednog *mastera* zajedno s poslužiteljima s više *chunkova* (komada). Svaka GFS datoteka se dijeli na *chunkove*, 64 megabajt svaki, nakon čega se *chunkovi* distribuiraju preko onoga što se zove komad poslužitelji (*chunk servers*). Važna

primjedba je da je GFS *master* spojen samo za informacije metapodataka. Konkretno, GFS klijent prolazi naziv datoteke i *chunk* indeksa na *mastera*, očekujći kontakt adresu za *chunk*. Kontakt adresa sadrži sve podatke za pristup ispravnom *chunk* poslužitelju kako bi dobili potrebnu *chunk* datoteku. U tu svrhu, GFS *master* u biti održava prostor za ime, uz mapiranje iz naziva datoteke na *chunk*ove. Svaki *chunk* ima pridruženi identifikator koji će omogućiti *chunk* poslužitelju da ga traži. Osim toga, *master* prati gdje se nalazi *chunk*. *Chunkovi* se repliciraju na *handle* neuspjehe, ali ništa više od toga.

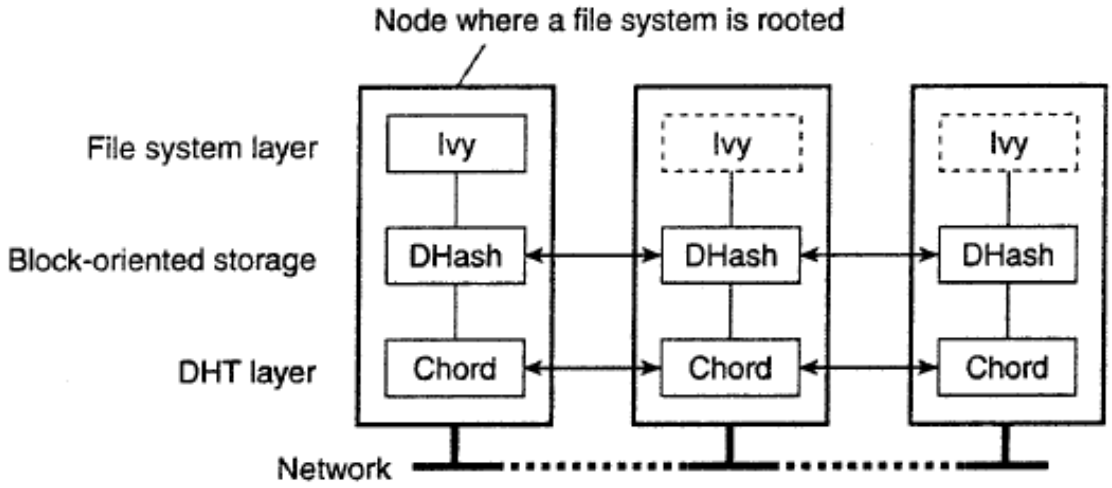
Zanimljiva značajka je da GFS *master* ne pokušava držati točne evidencije o *chunk* mjestima. Umjesto toga, on povremeno kontaktira *chunk* servere kako bi vidjeli koji *chunkovi* su pohranjeni. Prednost ovog sustava je jednostavnost. Imajmo na umu da je *master* u kontroli dodjele *chunkova* na *chunk* poslužiteljima. Osim toga, *chunk* poslužitelji vode evidenciju o tome što su pohranili. Kao posljedica toga, kada je *master* dobio *chunk* lokaciju, ima točnu sliku gdje se pohranjuju podaci. Međutim, stvari će postati kompliciranе ako je takvo viđenje moralo biti dosljedno cijelo vrijeme. Na primjer, svaki put kada se *chunk* poslužitelj ruši ili kad poslužitelj dodaje, *master* bi trebao biti informiran. Umjesto toga, mnogo je jednostavnije da osvježi svoje podatke iz trenutnog skupa *chunk* poslužitelja. GFS klijenti jednostavno znaju za koje *chunk* poslužitelje *master* vjeruje da skladište tražene podatke.

Budući da su *chunkovi* svakako replicirani, postoji velika vjerojatnost da je *chunk* dostupan na barem jednom od *chunk* poslužitelja. Važno pitanje dizajna je to što *master* u velikoj mjeri ima sve pod kontrolom. Dvije važne vrste mjere su poduzete kako bi se smjestila skalabilnost. Prva, i daleko najvažnija, je da većinu stvarnog rada obavlja *chunk* poslužitelj. Kada klijent treba pristupiti podacima, kontaktira *mastera* kako bi saznao koji *chunk* poslužitelj sadržava podatke. Nakon toga, on komunicira samo s *chunk* poslužiteljima. *Chunkovi* se repliciraju prema primarnoj *backup* shemi. Kada klijent izvodi operaciju ažuriranja, on kontaktira najbliži *chunk* poslužitelj koji sadržava te podatke, i gura svoje ažuriranje na tom poslužitelju. Ovaj poslužitelj će gurnuti nadopune na sljedeći najbliži poslužitelj koji također sadrži podatke, i tako dalje.

Nakon što su sva ažuriranja propagirana, klijent će kontaktirati primarni *chunk* poslužitelj, koji će zatim dodijeliti redni broj na ažuriranju rada i prenijeti ga na *backup*. U međuvremenu, *master* je zadržan izvan petlje. Hijerarhijski naziv prostora za datoteke se provodi pomoću jednostavne tablice na jednoj razini, u kojoj je put imena mapiran na metapodacima (kao što je ekvivalent indeks u tradicionalnim sustavima datoteka). Štoviše, cijela ova tablica se održava u glavnoj memoriji, uz mapiranje datoteke na *chunkove*. Ažuriranja na tim podacima prijavljena su na trajnu pohranu. Kad zapisnik postane prevelik, napravi se prijelaz kojim se pohranjuju podaci glavne memorije i to na takav način da se može odmah mapirati natrag u glavnu memoriju. Kao posljedica toga, intenzitet od GFS *mastera* je smanjen. Ova organizacija omogućuje da jedan *master* kontrolira nekoliko stotina *chunk* poslužitelja, što je značajna veličina za jedan klaster. Ako naknadno organiziramo uslugu, kao što je Google manje usluge preslikava na klastere, nije teško zamisliti da veliki skup klastera može raditi zajedno.

### *3.1.3. Simetrična arhitektura*

Naravno, potpuno simetrične organizacije koje se temelje na *peer-to-peer* tehnologiji također postoje. Svi trenutni prijedlozi koriste sustav DHT (Distributed Hash Table) za distribuciju podataka, u kombinaciji s ključnim preglednim mehanizmom. Bitna razlika je da li će se izgraditi sustav datoteka na vrhu distribuiranog sloja za pohranu, odnosno hoće li se cijela datoteka pohranjivati na čvorove koji sudjeluju. Primjer prve vrste sustav datoteka je Ivy, distribuirani sustav datoteka koji je izgrađen korištenjem sustava Chord DHT. Njihov sustav u osnovi sastoji se od tri zasebna sloja, kao što je prikazano na slici. 17. Najniži sloj se formira na Chord sustavu koji osigurava osnovne decentralizirane *lookup* objekte. U sredini je potpuno distribuirano orjentalni blok za pohranu. Konačno, na vrhu se nalazi sloj provedbe sustav datoteka nalik NFS-u.




*Slika 17.*

Pohrana podataka u Ivy ostvaruje se pomoću temeljnih Chord-ova, blokova orijentiranih distribuiranih sustava pohrane koji se zovu DHash. U biti, DHash je vrlo jednostavan. DHash zna za blokove podataka, svaki blok obično ima veličinu od 8 KB. Ivy koristi dvije vrste podatkovnih blokova. Sadržaj DHash bloka ima pridruženu tipku, koja se izračunava kao siguran *hash* sadržaja bloka. Na taj način, kad god se traži blok, klijent može odmah utvrditi da li se pretražuje pravi blok, ili je povraćena neka druga ili pokvarena verzija.

Nadalje, Ivy također omogućuje korištenje javnih ključ blokova. Oni su blokovi koji koriste javni ključ kao ključ pregleda. Kako bi se povećala dostupnost, DHash replicira svaki blok *B* u *k* neposrednih nasljednika poslužitelja odgovornog za pohranjivanje B. Osim toga, pretraženi blokovi također su spremljeni na ruti na kojoj stigao zahtjev za pretragom. Datoteke se implementiraju kao zasebne strukture podataka na vrhu DHash. Da bi se postigao ovaj cilj, svaki korisnik zadržava dnevnik operacija koje obavlja na datotekama. Radi jednostavnosti, pretpostavimo da postoji samo jedan korisnik po čvoru, tako da će svaki čvor imati svoj dnevnik. Dnevnik je povezan popis nepromjenljivih zapisa, gdje svaki zapis sadrži sve informacije vezane uz operaciju na Ivy sustav datoteka. Svaki čvor dodaje zapise samo na vlastitu, lokalnu, prijavu. Samo *log* glava je promjenjiva, i ukazuje na zadnje priloženi zapis. Svaki zapis je pohranjen u zasebnom sadržaju *hash* bloka.

Postoje različite vrste zapisa, koje otprilike odgovaraju različitim operacijama koje podržava NFS. Na primjer, ako izvodite operaciju ažuriranja na datoteci, zapis Pisati je stvoren, koji sadrži identifikator datoteke uz odmak za pokazivača i podacima koji se pišu. Isto tako, postoje zapisi za stvaranje datoteke (tj, dodavanjem nove inode), manipuliranje direktorijima, itd. Da bismo stvorili novi sustav datoteka, čvor jednostavno stvara novi dnevnik zajedno s novom inodom koji će služiti kao korijen. Ivy raspoređuje ono što je poznato kao NFS povratni poslužitelj koji je samo lokalni poslužitelj na korisničkoj razini koji prihvaća NFS zahtjeve lokalnih klijenata. U slučaju Ivy, NFS poslužitelj podržava ugradnju novonastalog sustava datoteka koji omogućuje aplikacijama da mu pristupe kao bilo koji drugi NFS sustava datoteka. Kada izvodite operaciju čitanja, lokalni Ivy NFS poslužitelj prikuplja podatake iz tih evidencija koje predstavljaju operacije pisanja na istom bloku podataka, omogućujući mu da dohvati nedavno pohranjene vrijednosti.

Imajte na umu da, jer svaki zapis se pohranjuje kao DHash blok, višestruki upiti diljem preklopne mreže mogu biti potrebni za preuzimanje odgovarajuće vrijednosti. Umjesto korištenja zasebnog sloja za pohranu, alternativni dizajn predlaže distribuciju cijele datoteke umjesto podataka blokova. Programeri Kosha predlažu distribuiranje datoteke na određenom nivou direktorija. U svom pristupu, svaki čvor ima točku montiranja direktorija nazvanu */kosha* koja sadrži datoteke koje će se distribuira pomoću sustava DHT. Distribucija datoteke na razini direktorija 1 znači da će sve datoteke u poddirektoriju */kosha/a* biti pohranjeni na istom čvoru. Isto tako, distribucija na razini 2 implicira da su sve datoteke pohranjene u direktorij */kosha/aa* pohranjene na istom čvoru.

Potencijalni nedostatak ovog pristupa je da čvor može ostati bez diskovnog prostora za pohranu svih datoteka pohranjenih u poddirektoriju za koji je odgovoran. Opet, jednostavno rješenje je pronađeno u stavljanju ogranka tog poddirektoriju na drugom čvoru i stvaranje simboličke veze na kojoj je sada pohranjena grana.

### *3.1.4. Procesi*

Kada je riječ o procesima, distribuirani sustavi datoteka nemaju neobična svojstva. U mnogim slučajevima, tu su različite vrste kooperativnih procesa: poslužitelji za pohranu datoteka i upravitelji datoteka, baš kao što je gore opisano za različite organizacije. NFS je dobar primjer koji ilustrira ustupke. Jedno od njegovih dugotrajnih razlikovnih obilježja (u usporedbi s drugim distribuiranim sustavima datoteka), bila je činjenica da su poslužitelji *stateless*. Drugim riječima, NFS protokol ne zahtjeva da poslužitelji održavaju bilo kakvo klijent stanje. Ovaj pristup je slijedio u verzijama 2 i 3, ali se nije zadržao na verziji 4.

Glavna prednost *stateless* pristupa je jednostavnost. Na primjer, kada dođe do pada *stateless* poslužitelja, tu u suštini nema potrebe za ulazak u fazu oporavka kako bi se poslužitelj doveo u prethodno stanje. Međutim, kao što smo objasnili, još uvijek treba uzeti u obzir da klijent ne može dati nikakva jamstva da li je zahtjev zaista obavljen ili nije. *Stateless* pristup u NFS protokolu nije mogao da se slijedi u potpunosti praktičnim implementacijama. Na primjer, zaključavanje datoteke ne može se lako učiniti od strane *stateless* poslužitelja. U slučaju NFS-a, odvojeni upravitelj zaključavanja se koristi da riješi ovakav problem.

Isto tako, neki autentifikacijski protokoli zahtjevaju da poslužitelj održava stanje na svom klijentu. Ipak, NFS poslužitelji općenito mogu biti dizajnirani na način da je potrebno vrlo malo podataka o klijentima da se on održava. Za veći dio, sustav je radio na odgovarajući način. Počevši s verzijom 4, *stateless* pristup je napušten, iako je novi protokol osmišljen na način da poslužitelj ne treba održavati više podataka o svojim klijentima. Osim onih što smo spomenuli, postoje i drugi razlozi za izbor *stateful* pristupa. Važan razlog je što se od NFS verzije 4 očekuje se da će također raditi preko širokog područja mreže.

To zahtjeva da klijenti mogu učinkovito iskoristiti catch, a zauzvrat zahtjevaju učinkovit cache protokol dosljednosti. Takvi protokoli često rade najbolje u suradnji s poslužiteljima koji održavaju neke informacije o datotekama. Na primjer, poslužitelj može povezati zakup sa svake datoteke koju preda klijentu, obećavajući da će dati klijentu ekskluzivni pristup

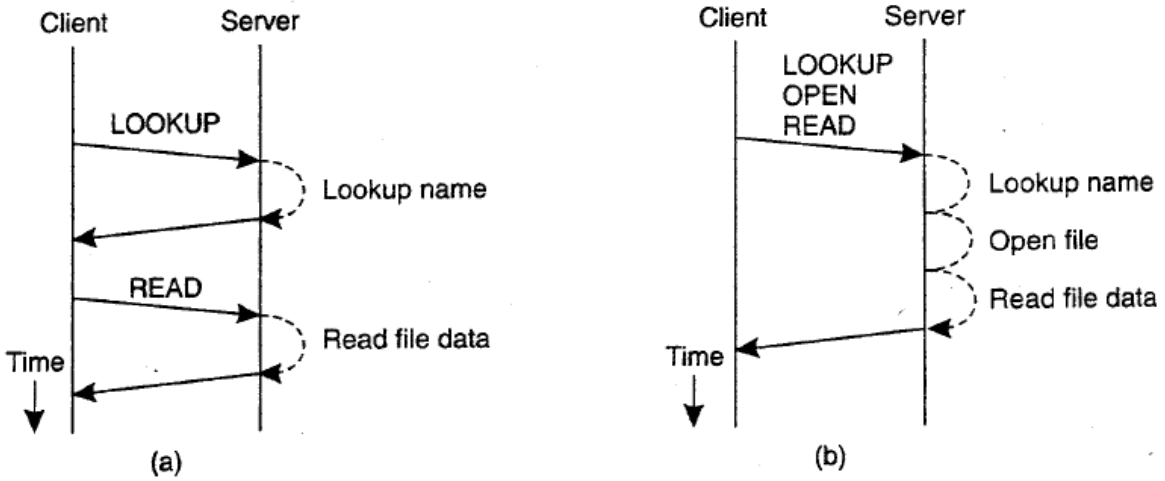
čitanju i pisanju dok najam ne istekne ili se obnovi. Najočitija razlika u odnosu na prethodne verzije je podrška za rad na otvorenom. Osim toga, NFS podržava povratni poziv procedure kojeg poslužitelj može napraviti prema RPO (Recruitment Process Outsourcing) klijentu. Jasno, uzvratni pozivi također zahtjevaju da poslužitelj pratiti svoje klijente. Slično razmišljanje je utjecalo na oblikovanje drugih distribuiranih sustava datoteka. Uglavnom, ispada da održavanje potpunog stateless dizajna može biti vrlo teško, često dovodi do izgradnje stateful rješenja kao poboljšanja, kao što je slučaj s NFS zaključavanjem datoteka.

### *3.1.5. Komunikacije*

Kao i kod procesa, ne postoji ništa posebno ili neobično kada je u pitanju komunikacija u raspodjeljenim sustavima datoteka. Mnogi od njih se temelje na daljinskim proceduralnim pozivima (RPC - *remote procedure call*), iako su neka zanimljiva poboljšanja napravljena za podršku posebnih slučajeva. Glavni razlog za odabir RPC mehanizma jest da je sustav neovisan od temeljnog operativnog sustava, mreža i transportnih protokola.

### *3.1.6. RPC u NFS-u*

Na primjer, u NFS-u, sva komunikacija između klijenta i poslužitelja nastavlja se duž *Open Network Computing* RPC (ONC RPC) protokola. ONC RPC je sličan drugim RPC sustavima. Svaka NFS operacija može se provoditi kao jedan poziv za daljinski postupak na poslužitelj datoteka. U stvari, sve do NFS v4, klijent je bio odgovoran da učini život poslužitelja što je moguće jednostavnijim držeći zahtjeve relativno jednostavne. Na primjer, kako bi čitali podatke iz datoteke po prvi put, klijent bi obično prvo morao pogledati *handle* datoteke pomoću operacije pretraživanja, nakon čega se može izdati zahtjev za čitanje, kao što je prikazano na slici. 18 (a). Ovaj pristup zahtijeva dva uzastopna RPC-a. Nedostatak je postao jasan kada se u obzir uzme korištenje NFS-a u sustavu širokog područja. U tom slučaju, dodatno kašnjenje od drugog RPO dovelo je do degradacije performansi. Kako bi izbjegli takve probleme, NFS v4 podržava složene procedure po kojoj se nekoliko RPC-a mogu svrstati u jedan zahtjev, kao što je prikazano na slici. 18 (b).




*Slika 18.*

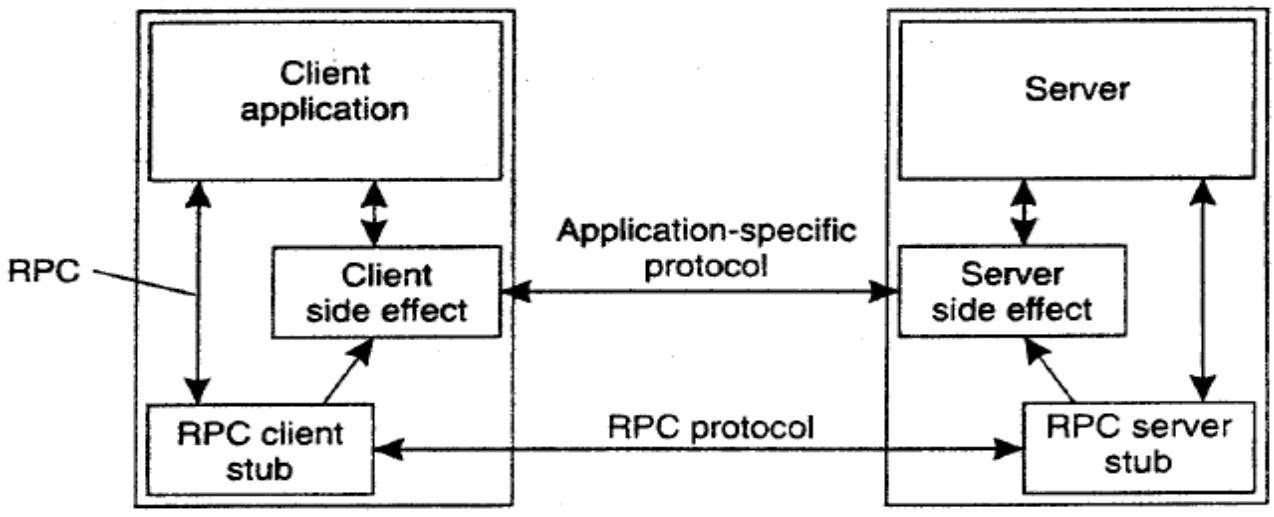
U našem primjeru, klijent spaja pretraživanje i učitava zahtjev u jedan RPO. U slučaju verzije 4, također je potrebno otvaranje datoteke prije nego se počne čitanje. Nakon što je *handle* datoteka pretražena, dodana je na otvorenu operaciju, nakon čega poslužitelj nastavlja s operacijom čitanja. Ukupni učinak u ovom primjeru je da se samo dvije poruke moraju razmjeniti između klijenta i poslužitelja. Nema transakcijske semantike povezane sa složenim postupcima. Poslovanje koje je grupirano zajedno u složenoj proceduri jednostavno se obrađuje po redu kako je zatraženo. Ako postoje istovremene operacije od drugih klijenata, onda se ne poduzimaju mjere kako bi se izbjegli sukobi. Ako operacija ne uspije iz bilo kojeg razloga, onda se ne izvršavaju daljnje radnje, a rezultat koji je dobiven do sada se vraća klijentu.

### *3.1.7. RPC2 Podsustav*

Još jedno zanimljivo poboljšanje RPC-a je razvijeno kao dio Coda sustava datoteka. RPC2 je paket koji nudi pouzdane RPC ispred (nepouzdanih) UDP (*User Datagram Protocol*) protokola. Svaki put kada se naziva daljinski postupak, RPC2 klijent kod počinje novu temu koja šalje zahtjev pozivanja poslužitelja te ga blokira dok ne stigne odgovor. Kako obrada zahtjeva zahtjeva znatno vrijeme za dovršetak, poslužitelj redovito šalje poruke natrag klijentu da mu da do znanja da i dalje radi na zahtjevu. Ako poslužitelj istekne, prije ili kasnije, ova tema će primijetiti da su poruke prestale i i prijaviti kvar u aplikaciji koja

poziva. Zanimljiv aspekt RPC2 je njegova podrška za nuspojave. Nuspojava je mehanizam kojom klijent i poslužitelj mogu komunicirati pomoću protokola za pristup aplikaciji. Razmotrimo, na primjer, klijent otvara datoteku na video serveru.

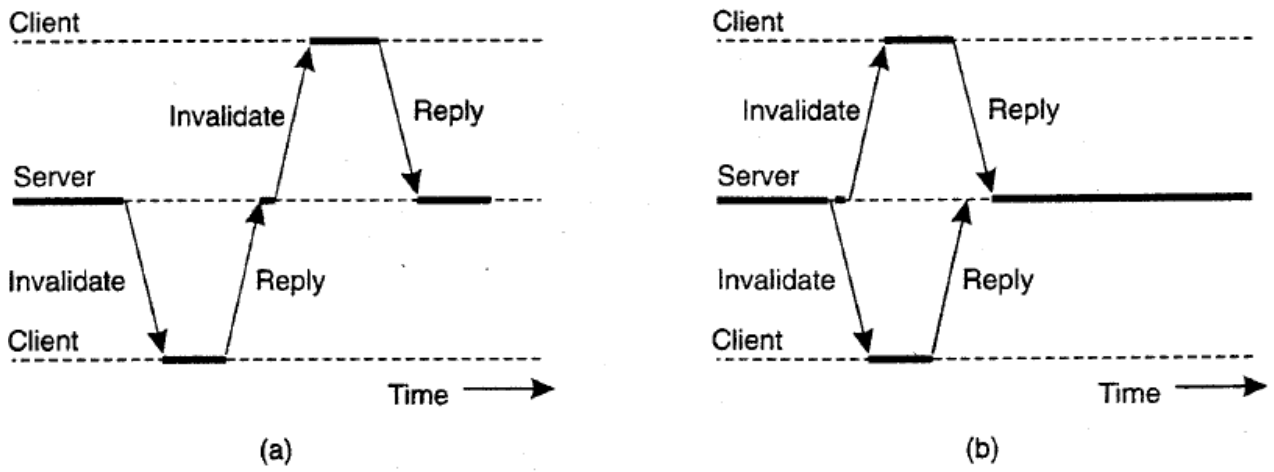
Ono što je potrebno u ovom slučaju je da klijent i poslužitelj postave kontinuirani tok podataka s izosinkronim načinom prijenosa. Drugim riječima, za prijenos podataka s poslužitelja na klijenta je zajamčeno da će u roku od minimalno i maksimalno stići do *end-to-end* odgode. RPC2 omogućuje klijentu i poslužitelju postaviti zasebnu vezu za prijenos video podataka na klijentu za vrijeme trajanja slanja podataka. Povezivanje se obavlja kao nuspojava od RPC poziva na poslužitelju. U tu svrhu, RPC2 *runtime* sustav pruža sučelje za efekt rutine koja će se provoditi od strane aplikacije *developer*. Na primjer, postoje rutine za uspostavu veze i rutine za prijenos podataka. Ove rutine automatski se izvode po RPC2 *runtime* sustavu na klijentu i poslužitelju ali njihova provedba je inače potpuno neovisana o RPC2. Ovaj princip nuspojava prikazan je na slici. 19




*Slika 19.*

Važno pitanje dizajna u Codi je da poslužitelji drže nuspojave u Coda RPC2 sustavu za koje klijenti imaju lokalnu kopiju datoteke. Kada je datoteka izmijenjena, poslužitelj poništava lokalne kopije obavještavajući odgovarajuće klijente putem RPO. Jasno, ako server može obavijestiti samo jednu stranku u isto vrijeme, poništavanje svih klijenta može

potrajati neko vrijeme, kao što je prikazano na slici. 20 (a). Problem je uzrokovan činjenicom da RPC povremeno neće uspjeti. Poništavanje datoteke strogim redoslijedom može biti značajno odgođeno, jer poslužitelj ne može doći do eventualno srušenog klijenta, ali će odustati od tog klijenta tek nakon relativno dugog vremena isteka. U međuvremenu, drugi klijenti i dalje će čitati iz svojih lokalnih primjeraka.




*Slika 20.*

Alternativa (i bolje rješenje) je prikazana na slici. 20 (b). Ovdje, umjesto da se poništi svaka kopija jedna po jedna, poslužitelj šalje poruku poništenja svim klijentima u isto vrijeme. Kao posljedica toga, svi klijenti koji nisu doživjeli neuspjeh su obaviješteni u isto vrijeme.

Paralelni RPC se provode putem MultiRPC sustava, koji je dio paketa RPC2. Važan aspekt MultiRPC je da paralelno zazivanje RPC-a je potpuno transparentno za onoga koji poziva. Drugim riječima, prijemnik MuitiRPC poziva ne može raspoznati taj poziv s normalnog RPO-a. Sa strane pozivatelja, paralelno izvršenje je također u velikoj mjeri transparentno. Na primjer, semantika MultiRPC u prisutnosti kvarova je uvelike ista kao kod onog normalnog RPC-a. Slično, mehanizmi nuspojava se mogu koristiti na isti način kao i prije. MultiRPC se provodi uglavnom izvršenjem više RPC-a paralelno.

To znači da pozivatelj izričito šalje RPC-u zahtjev za svakog primatelja. Međutim, umjesto da se odmah čeka odgovor, on odgađa blokiraje dok se svi zahtjevi ne upute. Drugim

riječima, pozivatelj priziva brojne jednosmjerne RPC-ove, nakon čega se blokira dok se svi odgovori ne prime od strane primatelja koji nije doživio kvar. Alternativni pristup paralelnom izvršavanju RPC-a u MultiRPC-u osigurava postavljanje *multicast* skupine, i slanjem RPC svim članovima skupine pomoću *IF multicast.*

### *3.1.8. Komunikacijski plan 9*

Na kraju, valja spomenuti jedan potpuno drugačiji pristup rukovanja komunikacijama u raspodjeljenim datotečnim sustavima. Plan 9 nije toliko distribuirani datotečni sustav, nego distribuirani sustav baziran na datotekama. Svim resursima se pristupa na isti način, odnosno s datotečnom sintaksom kao i operacijama, uključujući čak i resurse, kao što su procesi i mrežna sučelja. Ova ideja je došla od UNIX-a, koji također nastoji ponuditi sučelja poput datoteka resursima, ali to je istraženo mnogo više i dosljednije u planu 9. Da ilustriramo, mrežna sučelja zastupljena su od strane sustava datoteka, koja se u ovom slučaju sastoje od zbirka posebnih datoteka. Taj pristup je sličan UNIX-u, iako su mrežna sučelja na UNIX-u predstavljena pločicama, a ne sustavima datoteka. (Imajte na umu da je sustav datoteka u ovom kontekstu opet blok uređaj koji sadrži sve podatke i metapodatke koji obuhvaćaju zbirku datoteka.) U Planu 9, na primjer, pojedina TCP veza zastupa poddirektorij koji se sastoji od datoteke prikazan na slici. 21

| File | Description |
|---|---|
| ctl | Used to write protocol-specific control commands |
| data | Used to read and write data |
| listen | Used to accept incoming connection setup requests |
| local | Provides information on the caller's side of the connection |
| remote | Provides information on the other side of the connection |
| status | Provides diagnostic information on the current status of the connection |

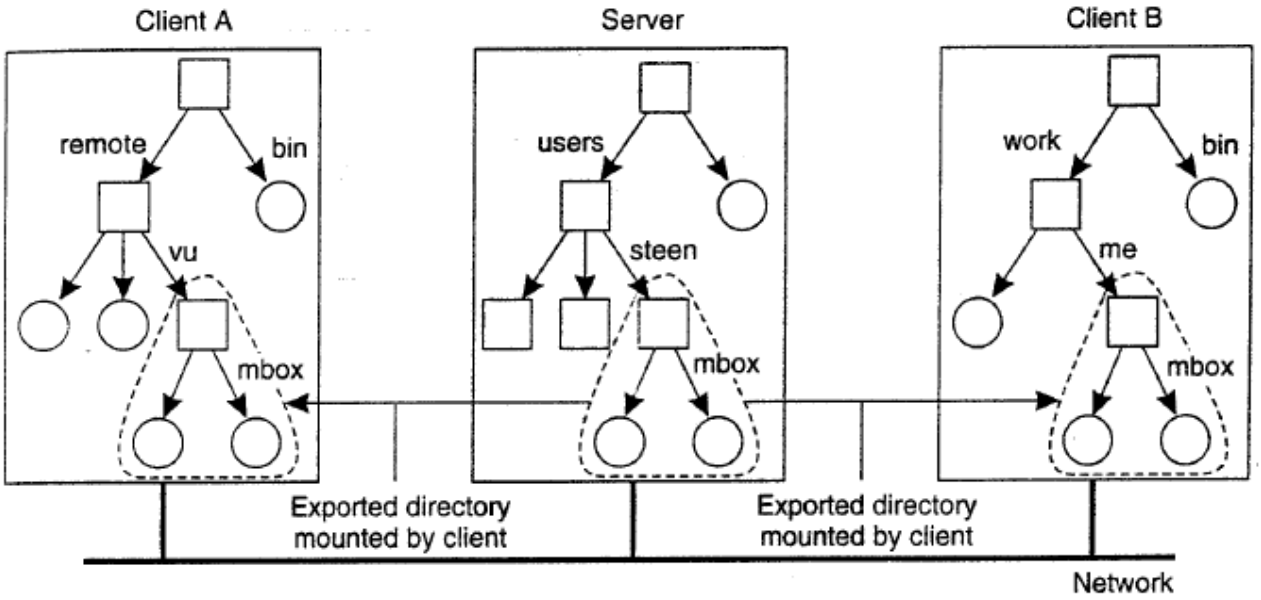
*Slika 21.*

Datoteka *ctl* se koristi za slanje upravljačkih naredbi na vezi. Na primjer, da biste otvorili Telnet sesije na stroju s IP adresom 192.31.231.42 pomoću porta 23, to zahtjeva da pošiljatelj piše tekstualno niz "spojiti 192.31.231.42! 23" datoteci *ctl*. Prijamnik će prethodno napisati niz "objaviti 23" na vlastitu *ctl* datoteku, što znači da može prihvatiti dolazne zahtjeve sesije. Datoteke podataka koriste se za razmjenu podataka jednostavnim izvođenjem operacija čitanja i pisanja. Ove operacije slijede uobičajenu UNIX semantiku za operacije datoteke.

Na primjer, za pisanje podataka na vezu, proces jednostavno zaziva operaciju *res = write* gdje je *f* datoteka deskriptor koji se vraća nakon otvaranja datoteke podataka, *buf* je kazaljka na spremniku koji sadrži podatke za upisivanje, a *nbytes* je broj bajtova koji bi trebao biti izvađen iz *buffera*. Broj bajtova koji je zaista napisan vraća se i pohranjuje u promjenjivom *res*. Datoteka Slušati koristi se za čekanje zahtjeva za postavljanje veze. Nakon što je proces najavio svoju spremnost da prihvati nove veze, može učiniti blokiranje čitanja na datoteci slušati. Ako zahtjev dođe, poziv vraća deskriptor datoteku na novu c*tl* datoteku koja odgovara novonastaloj vezi direktorija.

### *3.1.9. Imenovanje u NFS-u*

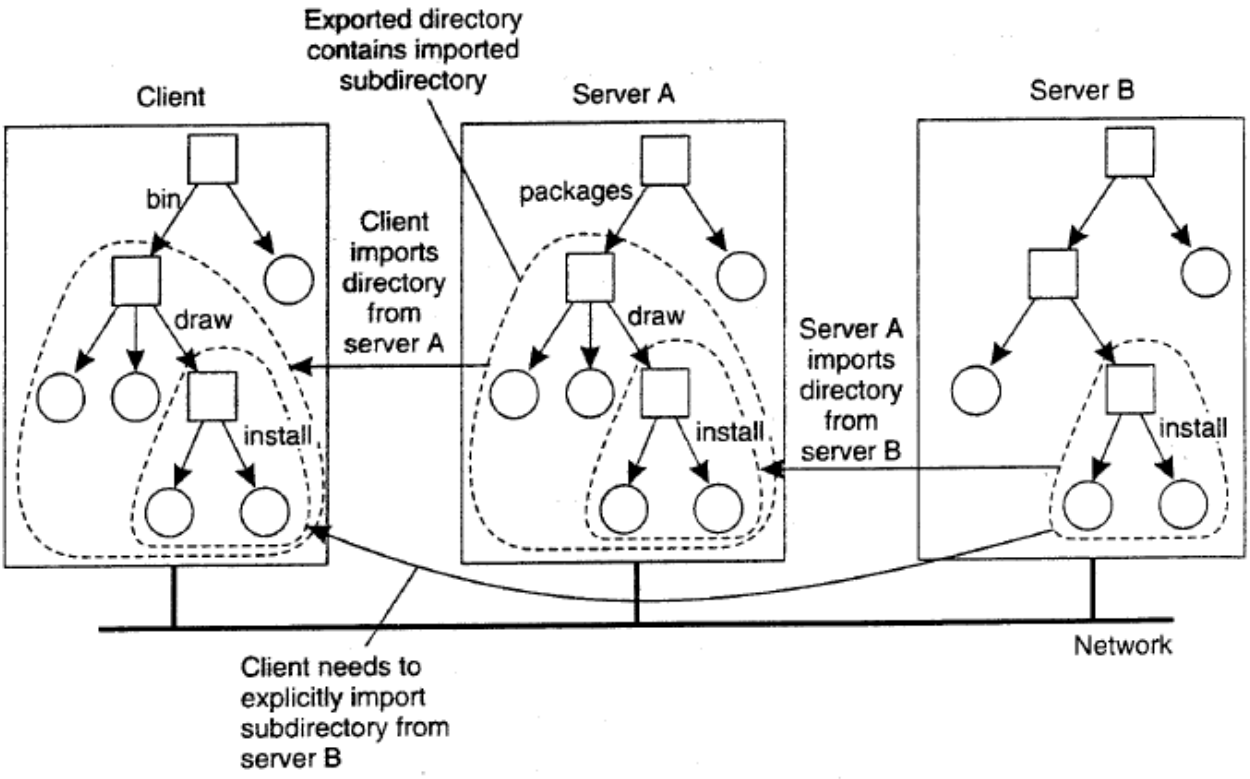
Temeljna ideja za NFS imenovanje modela je pružiti klijentima potpuni transparentan pristup udaljenom sustavu datoteka koje održava poslužitelj. Ta transparentnost se postiže dopuštajući klijentu da montira daljinski sustav datoteka u vlastiti lokalni sustava datoteka, kao što je prikazano na slici. 22. Umjesto montaže cijelog sustava datoteka, NFS omogućuje klijentima da montiraju samo dio sustava datoteka. Poslužitelj izvozi imenik, kada taj imenik i njegovi zapisi postanu dostupni klijentima. Izvezeni direktorij može biti montiran na lokalni prostora naziv klijenta.




*Slika 22.*

Ovaj dizajn pristup ima ozbiljnu implikaciju: U principu, korisnici ne dijele mjesta s imenima. Kao što je prikazano na slici. 22, datoteka pod nazivom *remote/vu/mbox* na klijentu A se zove */work/me/mbox* na klijentu B. Ime datoteke, dakle, ovisi o tome kako klijenti organiziraju svoj lokalni naziv prostor, i gdje su izvozni direktoriji montiranja. Nedostatak ovog pristupa u distribuiranom sustavu datoteka je da dijeljenje datoteka postaje mnogo teže.

Na primjer, svaki klijent možda koristiti lokalni telefonski imenik */usr/bin* za montiranje sustava datoteka koji sadrži standardnu kolekciju programa koji su dostupni svima. Isto tako, katalog */local* može se koristiti kao standard za ugradnju lokalnog sustav datoteka koji se nalazi na klijentu domaćina. NFS poslužitelj može sam montirati direktorije koji se izvoze po drugim poslužiteljima. Međutim, ne smije izvoziti te direktorije vlastitim klijentima. Umjesto toga, klijent će morati izričito montirati takav katalog s poslužitelja koji ga održava, kao što je prikazano na slici. 23. Ovo ograničenje dolazi dijelom iz jednostavnosti.




*Slika 23.*

Kako bismo objasnili ovu točku detaljnije, pretpostavimo da je poslužitelj A domaćin sustav datoteka FSA iz kojeg se izvozi imenik */packages*. Ovaj katalog sadrži folder */draw* koji djeluje kao točka montiranja za sustav datoteka *F* koja je izvezena od strane poslužitelja B i montiran od strane A.

Ako je razlučivost imena iterativna (kao što je slučaj u NFSv3) za rješavanje imena */bin/remi/install*, klijent kontaktira poslužitelja A kada je na lokalnoj razini rješeno */bin* i traži da se vrati na datoteke za katalog. Rezolucija imena u NFSv3 (i starijim verzijama) je strogo iterativna u smislu da se samo jedan naziv datoteke može gledati u isto vrijeme. Drugim riječima, rješavanje imena, kao što je */bin/remi/install* zahtjeva tri zasebna poziva na NFS poslužitelja. Osim toga, klijent je u potpunosti odgovoran za provedbu rezolucije staze imena.

Postoji još jedna posebnost s NFS dohvatom imena koji je riješen s verzijom 4. Ako u obzir uzmemo poslužitelj datoteka koji je domaćin nekoliko sustavava datoteka uz strogu

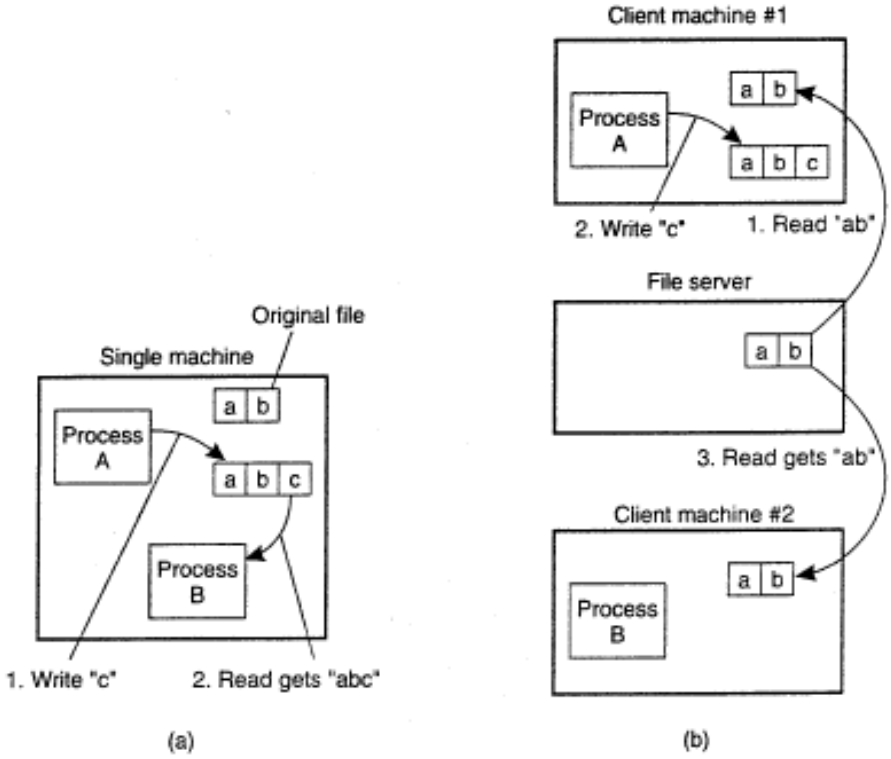
iterativnu rezoluciju imena u verziji 3, kad god napravimo pretrage za direktorij na kojemu je montiran drugi sustav datoteka, pretraživanje će vratiti *handle* datoteku. Nakon čitanja taj imenik će vratiti svoj izvorni sadržaj.

### 3.2. Sinkronizacija

Sada ćemo nastaviti našu raspravu s naglaskom na pitanjima sinkronizacije u distribuiranim sustavima datoteka. Postoje razni problemi koji zahtjevaju našu pažnju. Na prvom mjestu, sinkronizacija za sustave datoteka ne bi bila problem da datoteke nisu podijeljene. Međutim, u distribuiranim sustavima, semantika dijeljenja datoteka postaje malo nezgodna kada su problemi s performansama u pitanju. U tu svrhu, različita rješenja su predložena od kojih ćemo spomenuti samo najvažnije.

### 3.3. Semantika dijeljenja datoteka

Kada dva ili više korisnika dijele istu datoteku u isto vrijeme, potrebno je definirati semantiku čitanja i pisanja upravo kako bi izbjegli probleme. U jednoprocesorskim sustavima koji omogućuju procese za dijeljenje datoteka, kao što su UNIX, semantika obično navodi da, kada operacija čitanja slijedi postupke pisanja, čitanje vraća tek napisanu vrijednost, kao što je prikazano na slici. 24. Slično tome, kada se dva pisanja dogode u brzom slijedu, nakon čega slijedi čitanje, vrijednost čitanja je vrijednost pohranjena do posljednjeg pisanja. U stvari, sustav provodi apsolutno vrijeme naručivanja na svim operacijama i uvijek vraća najnoviju vrijednost. Mi ćemo se odnositi prema ovom modelu kao UNIX semantici.




*Slika 24.*

Ovaj model je lako razumljiv i jednostavan za implementaciju. Na jednom procesoru kad čitanje slijedi iza pisanja, vrijednost koja se vrati od čitanja je tek napisana vrijednost. Kod distribuiranog sustava sa *cachingom*, zastarjele vrijednosti mogu biti vraćene. U distribuiranom sustavu, UNIX semantika se može postići jednostavno sve dok postoji samo jedna datoteka poslužitelj i klijenti ne hvataju datoteke. Sva čitanja i pisanja idu izravno na poslužitelj datoteke, koja ih obrađuje strogo sekvencijalno. Ovaj pristup daje UNIX semantiku. U praksi, međutim, izvedba distribuiranog sustav u kojem svi zahtjevi datoteka moraju ići na jednog poslužitelja je često loša.

Ovaj problem se često rješava tako što se klijentima dopušta održavanje lokalne kopije jako korištenih datoteka u svojim privatnim (lokalnim) spremnicima. Iako ćemo razgovarati o pojedinostima datoteke *caching* u nastavku, za sada je dovoljno istaknuti da, ako klijent lokalno mijenja spremljenu datoteku i ubrzo nakon toga drugi klijent čita datoteku s poslužitelja, drugi klijent će dobiti zastarjele datoteke, kao što je prikazano na slici. 24 (b). Jedan izlaz iz ovog problema je da se odmah proširuju sve promjene spremljene datoteke na poslužitelja. Iako konceptualno jednostavan, ovaj pristup je neučinkovit.

Umjesto zahtjeva za čitanjem kako bi se vidjeli učinci svih prethodnih zapisa, možemo dobiti novo pravilo koje kaže: promjene na otvorenoj datoteci su inicijalno vidljive samo za proces (ili možda stroj) koji modificiraju datoteku. Samo kada je datoteka zatvorena promjene koje su napravljene vidljive su drugim procesima (ili strojevima). Usvajanje takvog pravila ne mijenja ono što se događa na slici. 24 (b). ali to ne redefinira stvarno ponašanje

Ovo pravilo se široko primjenjuje i poznato je kao sesija semantike. Većina distribuiranih sustava datoteka implementira sesije semantike. To znači da, iako u teoriji oni slijede daljinski pristup modela, većina implementacije koristi lokalne spremnike, učinkovito provodeći *upl oad/download* modele. Korištenjem sesije semantike postavlja se pitanje što će se dogoditi ako se dva ili više korisnika istodobno mijenjaju istu sliku. Jedno od rješenja je da kako se svaka datoteka zatvora naizmjenice, njegova vrijednost se šalje natrag poslužitelju, tako da konačni rezultat ovisi o tome čiji zahtjev za zatvaranje je nedavno obrađen od strane poslužitelja.

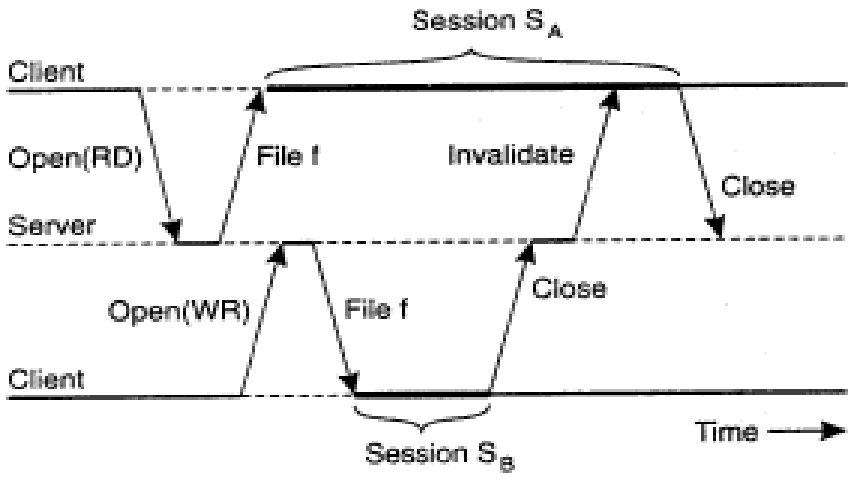
### 3.4. Dijeljenje datoteka u Codi

Sesija semantike u NFS-u nalaže da će posljednji proces koji zatvara datoteku imati njene izmjene propagirane na poslužitelju, ali u ranijim sesijama će biti izgubljene. Bolje je provesti nešto suptilniji pristup. Za smještaj dijeljenja datoteka, na Coda sustav datoteka koristi se posebna shema za dodjelu sredstava koja nosi neke sličnosti podijeljenim rezervacijama u NFS-u. Da biste razumjeli kako sustav funkcionira, sljedeće je važno.

Kada klijent uspješno otvara datoteke 1, cijeli primjerak *f* prenosi se na klijentov stroj. Poslužitelj bilježi da klijent ima kopiju *f*. Do sada, ovaj pristup je sličan otvorenoj delegaciji u NFS-a. Sada pretpostavimo da je klijent otvorio datoteku *f* za pisanje. Kad drugi klijent B želi otvoriti *f*, to neće uspjeti napraviti. Taj neuspjeh je uzrokovan činjenicom da je poslužitelj zabilježio da je klijent možda već promijenio *f*. S druge strane, da je klijent A otvorio *f* za čitanje, pokušaj klijenta B da dobije kopiju s

poslužitelja za čitanje također bi uspio. Pokušaj klijenta B da otvori za pisanje će uspjeti također. Sada razmislimo o tome što se događa kada je nekoliko primjeraka *f* pohranjeno lokalno na raznim klijentima.

S obzirom na ono što smo upravo rekli, samo jedan klijent će moći mijenjati *f*. Ako ovaj klijent mijenja *f.*, te potom zatvara datoteku, datoteka će se prenijeti na poslužitelja. Međutim, svaki drugi klijent može nastaviti čitati svoju lokalnu kopiju, unatoč činjenici da je zapravo ta kopija zastarila. Razlog za ovo nedosljedno ponašanje je da se sesija tretira kao transakcija u Codi. Uzmimo u obzir sliku. 25, koji pokazuje vremensku crtu za dva procesa, A i B. Pretpostavimo da je A otvorio *f* za čitanje, što je dovelo do prijave SA. Klijent B otvorio je *f* za pisanje, prikazan je kao *session* S8 na slici 25. Transakcijsko ponašanje u dijeljenju datoteka u Codi. Kad B zatvara *session* S8 prenosi ažuriranu verziju *f* na poslužitelja, koji će potom poslati poruku poništenje A. A će sada znati da je to čitanje iz starije verzije *f*. Međutim, iz transakcijske točke gledišta, to stvarno ne smeta, jer prijava SA može se smatrati da su na rasporedu prije *session* S8.




*Slika 25.*

### *3.4.1. Tolerancija grešaka*

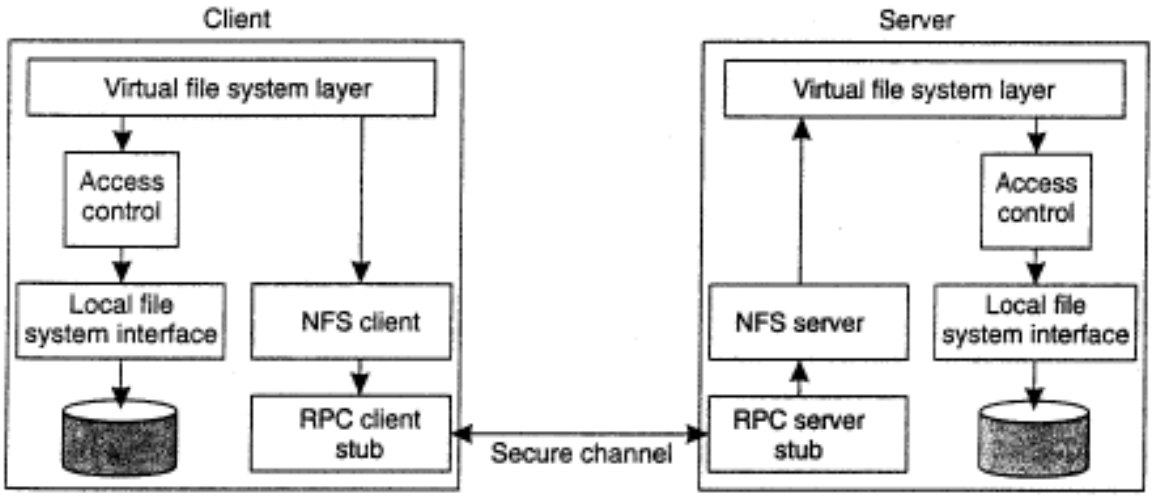
Tolerancija u distribuiranim sustavima datoteka se rješava u skladu s načelima o kojima smo razgovarali u ranije. Kao što smo već spomenuli, u mnogim slučajevima, odgovor je angažiran za izradu tolerancije greške skupine poslužitelja.

## 3.5. Sigurnost

Mnoga od sigurnosnih načela koja smo imali priliku vidjeti u ranijim poglavljima primjenjuju se izravno na distribuiranim sustavima datoteka. Sigurnost u distribuiranim sustavima datoteka organiziranih uz klijent poslužitelj arhitekturu je da imaju *handle* autentičnosti i kontrolu pristupa poslužitelja. To je jednostavan način obavljanja sigurnosti, pristup koji je usvojen, primjerice, u sustavima kao što su NFS. U takvim slučajevima, zajedničko im je da imaju zasebne provjere autentičnosti usluga, kao što su Kerberos, dok će poslužitelji datoteka jednostavno nositi odobrenje. Glavni nedostatak ove sheme je da to zahtjeva centraliziranu administraciju korisnika, što može ozbiljno ometati skalabilnost. U nastavku, prvo ćemo kratko razgovarali o sigurnosti u NFS-a kao primjer tradicionalnog pristupa, nakon čega ćemo obratiti pozornost na alternativne pristupe.

### *3.5.1. Sigurnost u NFS-u*

Kao što smo već spomenuli, osnovna ideja iza NFS-a jeste da bi daljinski sustav datoteka trebao biti predstavljen za klijente kao da je lokalni sustav datoteka. U tom pogledu, to bi trebalo biti iznenađenje da se sigurnost u NFS uglavnom usredotočuje na komunikaciju između klijenta i poslužitelja. Osim osiguravanja RPC-a, potrebno je kontrolirati datoteke pristupa. koje su obrađene pomoću kontrole pristupa atribute datoteka u NFS-u. Poslužitelj datoteka je zadužen za provjeru prava pristupa svojih klijenata, kao što ćemo objasniti u nastavku. U kombinaciji sa sigurnim RPC-om, NFS sigurnosna arhitektura je prikazana na slici. 26




.

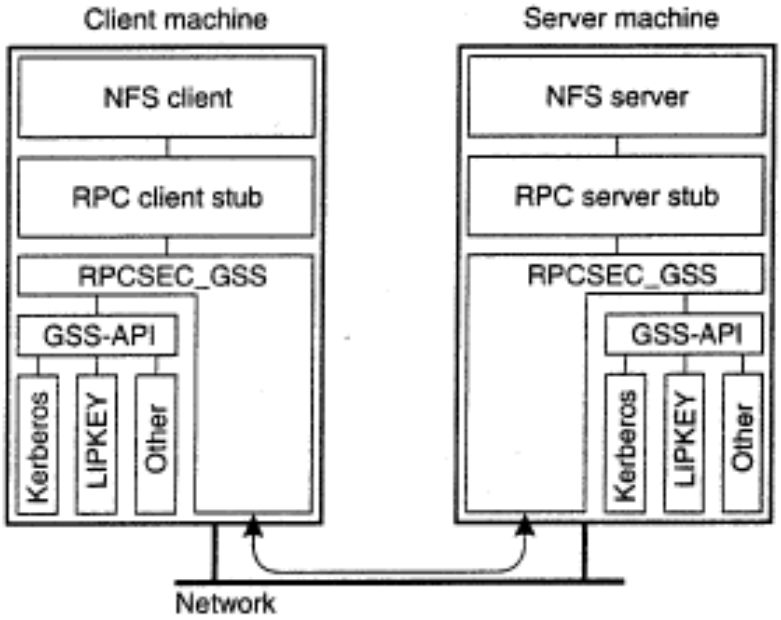
*Slika 26.*

*3.5.2. Sigurnost RPC*

S obzirom da je NFS na vrhu RPC sustava, postavljanje sigurnog kanala u NFS svodi se na uspostavu sigurnog RPC-a. Do NFSv4, siguran RPC značio je da je samo provjeru autentičnosti. Postojale su tri načina za autentifikaciju. Najčešće korištena metoda, koja je zapravo teško bilo kada zahtjeva autentifikaciju je poznata kao sustav za provjeru autentičnosti. U tom UNIX baziranoj metodi klijent jednostavno prolazi njegov korisnički ID i ID grupe na poslužitelju, zajedno s popisom grupa za koje tvrdi da je član. Ovi podaci se šalju na server kao nepotpisani tekst. Drugim riječima.poslužitelj nema načina da provjeri da li su ID od korisnika i grupa zapravo povezani s pošiljateljem. U biti, poslužitelj pretpostavlja da je klijent prošao propisane procedure za prijavu, te da se može vjerovati klijentima stroja. Druga metoda provjere autentičnosti u starijim verzijama NFS koristi Diffie-Hellman ključnu razmjenu kako bi se uspostavio ključ sesije, što je dovelo do onoga što se zove siguran NFS.

Ovaj pristup je puno bolji od autentičnosti sustava, ali je složeniji. Zbog čega se provodi rjeđe. Diffie-Hellman može se promatrati kao kriptosustav. U početku, nije bilo načina da se osigura distribuirati javni ključ poslužitelja, ali to je kasnije ispravljeno s uvođenjem usluge sigurnog imena. Točka kritike je uvijek bila uporaba razmjerno malih javnih ključeva, koji su samo 192 bita u NFS-u.

Treći protokol provjere autentičnosti je Kerberos, koji smo također opisali ranije. S uvođenjem NFSv4, sigurnost je poboljšana podrškom za RPCSEC_GSS. RPCSEC_GSS je opći sigurnosni okvir koji može podržati bezbroj sigurnosnih mehanizama za uspostavu sigurnih kanala. Konkretno, to ne samo da pruža mogućnosti za različite sustave za autorizaciju, već također podržava integritet poruke i povjerljivosti, dvije značajke koje nisu podržane u starijim verzijama NFS-a. RPCSEC_GSS se temelji na standardnom sučelju za sigurnosne usluge, a to su GSS-API-ja. RPCSEC_GSS se nanese na vrh ovog sučelja, što dovodi do organizacije koja je prikazana na slici 27. Za NFSv4, RPCSEC_GSS bi trebao biti konfiguriran s podrškom za Kerberos V5. Osim toga, sustav mora također podržati postupak poznat kao LIPKEY. LIPKEY je sustav javnog ključa koji omogućuje klijentima da autentificiraju koristeći lozinku, a poslužitelji mogu biti autentificirani koristeći javni ključ.




*Slika 27.*

Važan aspekt sigurnog RPC-a u NFS-u je da su dizajneri odabrali da ne daju svoje vlastite sigurnosne mehanizme, nego samo da pruže standardni način za rukovanje sigurnosti. Kao posljedica toga, dokazani sigurnosni mehanizmi kao Kerberos, mogu biti uključeni u provedbu NFS bez utjecaja na ostale dijelove sustava. Isto tako, ako postojeći sigurnosni mehanizme ispadne manjkav (kao što je u slučaju Diffie-

Hellman kada koristite male tipke), to se lako može zamijeniti. Treba napomenuti da je zbog činjenice da je RPCSEC_GSS implementiran kao dio RPC sloja koja je u podlozi NFS protokola može se koristiti za starije izvedbe.

# 4. Zaključak

Za organizaciju distriburanih sustava postoje određena pravila po kojem su oni uređeni. Ta pravila, paradigme, nisu jedinstveno definirana i svaki sustav je uređen na specifičan način. Zato smo raspravljali o najboljem načinu korištenja tih paradigmi. Kroz distribuirane sustave zasnovane na objektima i datotekama vidjeli smo mnogobrojna rješenja za općenite probleme s kojima se ostali distribuirani sustavi suočavaju. Upoznali smo se sa tematikom i terminologijom distribuiranih sustava te pobliže objasnili, po meni, najvažniju stvar distibuiranih sustava, a to je sigurnost. Sigurnost koja nije bila pretjerano naglašena u prijašnjim verzijama distribuiranih sustava kao što je NFS, za razliku od modernih distribuiranim sustava kojima je sigurnost najvažnija stavka. Kroz sigurnost možemo promatrati ubrzan rast distribuiranih sustava i zato sam je sa posebnom dozom naglasio.
Distribuirani sustavi su veoma napredni i pružaju učinkovita rješenja za mnoge probleme prvenstveno u današnjem svijetu interneta. Distribuirani sustavi zasnovani na objektima i datotekama daju dobru osnovu za ostale distribuirane sustave pružajući dobra rješenja za dane probleme.

# 5. Literatura


- CALLAGHAN, B.: *NFS Illustrated.* Reading, MA: Addison-Wesley, 2000.
- KLEIMAN, S.: *Vnodes: an Architecture for Multiple File System Types in UNIX. Proc.Summer Techn. Con!* USENIX, 1986. pp. 238-247.
- MONSON-HAEFEL, R., BURKE, B., and LABOUREY, S.: *Enterprise Java Beans.* Sebastopol,CA: O'Reilly & Associates, 4th ed., 2004.
- NEUMAN, C., YU, T., HARTMAN, S., and RAEBURN, K.: *The Kerberos Network*
- *Authentication Service.* RFC 4120, July 2005
- STEVENS, W. and RAGO, S.: *Advanced Programming in the UNIX Environment.* Reading, MA: Addison-Wesley, 2nd ed., 2005
- TANENBAUM, A.S., MULLENDER, S.J., AND VAN RENESSE, R.: *Using Sparse Capabilities in a Distributed Operating System, Proc. 6th Int'l Conf, on Distr. Comp. Syst.* , IEEE, pp. 558-563, 1986.
- TANENBAUM, A.S., VAN RENESSE, R., *Research Issues in Distributed Operating Systems*. Computing in High Energy Physics. L.O. Hertzberger and W. Hoogland (ed.). Amsterdam. December 1986.
- TANENBAUM, A.S., VAN STEEN, M., *Distributed Systems, Principles and Paradigms,* (2nd ed.)
- TANENBAUM, A. and WOODHULL, A.: *Operating Systems, Design and Implementation.* Englewood Cliffs, NJ: Prentice Hall, 3rd ed., 2006.
- TEL, G.: *Introduction to Distributed Algorithms.* Cambridge, UK: Cambridge University Press, 2nd ed., 2000.
- TUREK, J. AND SHASHA, S.: *The Many Faces of Consensus in Distributed Systems. IEEE Computer,* (25)6:8-17, 1992.