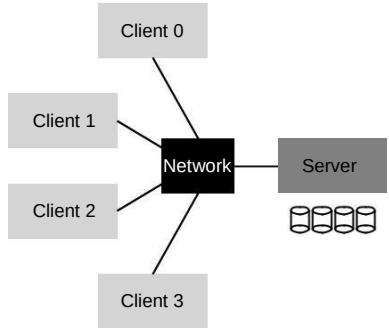
Sun Microsystems

Network File System (NFS)

Jedna od prvih upotreba klijent-server arhitekture bila je u oblasti distribuiranih fajl sistema. To je okruženje u kojem postoji nekoliko klijentskih mašina i jedna ili više serverskih. Server čuva podatke na svojim diskovima, a klijenti šalju zahteve za tim podacima preko nekog mrežnog protokola. Ovakvo ponašanje je opisano na slici ispod.



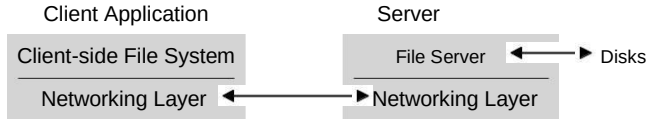
Slika 1: **A Klijent-server arhitektura**

Kao što se može videte sa slike, server ima nekoliko diskova, a klijenti šalju poruke preko mreže da bi pristupili svojim folderima i fajlovima na tim diskovima. Šta je problem ove arhitekture? Zašto ne bi smo dozvolili klijentima da koriste svoje lokalne diskove umesto serverskih? Prvi razlog je taj što ovako podešen sistem omogućava lako deljenje podataka. Podaci su već sami po sebi podeljeni između korisnika. Drugi razlog je centralna administracija. Backup fajlova je moguće uraditi samo na serverskim mašinama umesto na mnogo većem broju klijentskih mašina. Treći razlog je sigurnost podataka.

1 Pojednostavljeni distribuirani fajl sistem

Sada ćemo proučiti arhitekturu pojednostavljenog distribuiranog fajl sistema. Jednostavan klijent-server distribuirani fajl sistem ima mnogo komponenata. Na klijentskoj strani su klijentske aplikacije koje pristupaju folderima i fajlovima preko klijentskog fajl sistema. Klijentske aplikacije izdaju sistemske pozive ka klijentskom fajl sistemu (npr. Open, Read, Write, Close, mkdir...) da bi pristupile podacima na serverskim diskovima. Za klijentske aplikacije, fajl sistem ne mora biti ni malo drugačiji od lokalnog (disk-based) fajl sistema, osim po pitanju performansi. Distribuirani fajl sistemi obezbeđuju transparentan pristup fajlovima. Niko ne bi želeo da koristi fajl sistem koji bi bio težak za korišćenje.

Uloga klijentskog fajl sistema je da izvrši akcije koje su neophodne za obavljanje sistemskih poziva. Na primer, ako klijent izda zahtev za čitanje, klijentski fajl sistem može poslati poruku serverskom fajl sistemu (file server-u) da pročita određen memorijski blok. Serverski fajl sistem će pročitati taj blok sa diska i vratiti klijentu poruku sa podacima koje je zatražio. Klijentski fajl sistem će zatim kopirati te podatke u korisnički bafer koji je namenjen za sistemski poziv Read. Ovim se zahtev završava. Treba reći i da postoje situacije kada se podaci koje klijent zahteva već nalaze keširani u memoriji ili na klijentovom disku. Najbolji scenario je taj kada uopšte nema saobraćaja putem mreže.



Slika 2: **Arhitektura distribuiranog fajl sistema**

2 Prelazak na NFS

Jedan od prvih uspešnih distribuiranih fajl sistema razvijen je od strane Sun Microsystems-a pod nazivom Sun Network File System (NFS). Prilikom definisanja NFS-a Sun je iskoristio neobičan pristup. Umesto da razviju svoj i zatvoren sistem, oni su napravili otvoren protokol koji navodi tačne formate poruka koje razmenjuju klijent i server prilikom komunikacije. Različite kompanije mogu razviti svoje sopstvene NFS servere i sa njima se mogu takmičiti na tržištu NFS-a. Neke od velikih firmi koje su uradile ovako nešto su: Oracle, NetApp, EMC, IBM... Ovakav uspeh NFS-a je verovatno opravdan pristupom „otvorenog tržišta".

3 Jednostavan i brz oporavak od otkaza

U ovom poglavlju ćemo diskutovati o klasičnom NFS protokolu (verzija 2), koja je bila standard godinama. Za verziju 3 su napravljene manje promene, dok su malo veće napravljene za verziju 4.

U verziji 2 glavni cilj dizajniranja protokola bio je napraviti jednostavan i brz oporavak servera od otkaza. U okruženju gde imamo više klijenata i jedan server ovaj cilj je prilično jasan. Čim server otkaže, klijenti nemaju mogućnost rada. Onog trenutka kad se server oporavi, tada ceo sistem nastavlja dalje.

4 Ključ do brzog oporavka od otkaza: Protokol bez stanja

Ovaj jednostavan cilj je realizovan u NFSv2 projektovanjem nečega što se naziva „protokol bez stanja" (stateless protocol). Server je dizajniran tako da ne prati nikakva dešavanja na klijentskoj strani. Na primer server ne zna koji klijentu su keširali koji blok ili koji su fajlovi otvoreni kod klijenata... Ovaj protokol je dizajniran tako da svaki zahtev protokola (Read, Write...) dobije sve informacije koje su mu potrebne da bi taj zahtev uspešno završio.

Razmotrimo sistemski poziv Open kod protokola „sa stanjima". Sa prosleđenom putanjom do fajla, Open vraća fajl deskriptor (integer). Ovaj deskriptor se koristi u narednim Read pozivima da bi se pristupilo različitim blokovima fajla, kao što je dato u primeru ispod:

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);          // read MAX bytes from foo
...
read(fd, buffer, MAX);          // read MAX bytes from foo
close(fd);                      // close file
```

Slika 3: **Klijent vrši operacije čitanja fajla**

Sada zamislite da klijentski fajl sistem otvara fajl slanjem poruke (preko protokola) serveru: „Otvori fajl foo i vrati mi deskriptor nazad". Tada server otvara fajl kod sebe u lokalu, i vraća klijentu deskriptor. U narednim čitanjima, klijentska aplikacija koristi deskriptor da pozove sistemsku funkciju Read. Koristi ga tako što pošalje serveru poruku: „Pročitaj određen broj bajtova iz fajla koji je određen deskriptorom koji ti šaljem".

U ovom primeru, fajl deskriptor se može posmatrati kao deljeno stanje između klijenta i servera. Ovo deljeno stanje komplikuje oporavak od otkaza servera. Zamislite da server padne nakon što se prvi poziv funkcije Read završi, ali pre nego što klijent pošalje drugi. Server tada neće znati na koji se fajl *fd* odnosi. Ta informacija bila je zapamćena u memoriji, i obrisana nakon pada servera. Da bi rešili ovaj problem, klijent i server moraju napraviti neku vrstu protokola za oporavak, gde bi klijent bio siguran da sadrži dovoljno informacija u memoriji da bi mogao da kaže serveru sve što mu treba (u ovom slučaju je to da se *fd* odnosi na fajl foo).

Otažavajuća okolnost je i ta što kod protokola sa stanjima server mora da reši i slučajeve kada klijentska mašina otkaže. Zamislite na primer scenario u kome klijent otvara datoteku i zatim njegova mašina otkaže. Kako server da zna kada može da zatvori tu datoteku? U normalnim okolnostima, klijent bi poslao serveru zahtev Close, ali pošto je klijentova mašina otkazala, server nikad neće primiti taj zahtev. U ovoj situaciji, server bi morao da primeti da je klijentova mašina otkazala i da sam zatvori fajl.

Iz ovih razloga su se projektanti NFS-a odlučili za protokol bez stanja: svaka klijentska operacija sadrži sve informacije koje su potrebne da se zahtev izvrši do kraja. Nije potrebeban izvanredan oporavak od otkaza. Dovoljno je da se server startuje ponovo, a da klijent u najgorem slučaju ponovo pošalje isti zahtev serveru.

5 NFSv2 protokol

Dolazimo do definicije NFSv2 protokola. Jedan od ključnih pojmova za razumevanje NFS protokola je upravljač fajlom (file handle). Upravljači fajlova se koriste da jedinstveno opišu fajl ili folder nad kojim se radi određena operacija. Mnogi zahtevi protokola u sebi uključuju upravljač fajlovima.

Možete gledati na upravljač fajlova kao da ima 3 bitne komponente: identifikator fajl sistema, inode broj, i broj generacije fajla (volume identifier, inode number, generation number). Ove 3 stavke jedinstveno određuju fajl ili folder kojem klijent želi da pristupi. Identifikator fajl sistema određuje na koji se fajl sistem zahtev odnosi. Inode broj označava kojem fajlu se pristupa. Broj generacije je neophodan pri ponovnom pristupu inode-u. Njegovim uvećanjem prilikom svakog pristupa inode-u, server se osigurava da klijent sa starim upravljačem fajlova ne može pristupiti najnovijoj verziji fajla.

U nastavku možete videti nekoliko bitnijih delova protokola.

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

Slika 4: **NFS protokol - primeri**

Ukratko ćemo istaći važnije komponente protokola. Na primer, LookUp se koristi za dobijanje upravljača fajla, koji se nakon toga koristi za dobijanje podataka iz tog fajla. Klijent šalje upravljač fajla direktorijuma i ime fajla koji se traži. Server njemu vraća upravljač fajla traženog fajla, kao i atribute koji su vezani za njega.

Na primer, pretpostavimo da klijent već ima upravljač fajla za root direktorijum (/) (inače bi se to dobilo kroz NFS protokol). Ako klijentska aplikacija hoće da otvori fajl /foo.txt klijentski fajl sistem šalje LookUp zahtev serveru prosleđujući mu upravljač fajla za root direktorijum i ime fajla (foo.txt). Ako se sve završi uspešno, upravljač fajla za traženi fajl, kao i svi njegovi atributi, biće vraćeni nazad klijentu.

U slučaju da se pitate, atributi su samo metapodaci koje fajl sistem čuva za svaki fajl (npr. datum kreiranja fajla, vreme poslednje modifikacije, dozvole...).

Jednom kada upravljač fajla postane dostupan, klijent može da izvrši Read i Write zahteve nad njim. Read zahtevu se prosleđuje upravljač fajla za taj fajl, zajedno sa ofsetom (odakle čitanje treba da počne) i brojem bajtova koji treba da se pročita. Server će nakon toga biti u mogućnosti da izvrši traženu akciju (upravljač fajla govori serveru sa kog fajl sistema i koji fajl treba da pročita, a ofset i broj bajtova govore šta treba da pročita iz fajla) i vratiće klijentu podatke. Write radi slično, samo što će sada klijent morati da pošalje serveru podatke koje želi da upiše, a on mu vraća informaciju o tome da li je uspeo to da izvrši ili ne.

Još jedan interesantan sistemski poziv je GetAttr. Za prosleđeni upravljač fajlova vraćaju se svi atributi vezani za taj fajl (npr. vreme poslednje izmene). Videćemo zbog čega su ovi podaci važni kada budemo govorili o keširanju.

6 Od protokola do distribuiranog fajl sistema

Sada bi trebalo da dobijete neki osećaj o tome kako je ovaj protokol pretvoren u fajl sistem između klijentskog i serverskog fajl sistema. Klijentski fajl sistem prati otvorene datoteke i prevodi zahteve aplikacije u odgovarajuće poruke protokola (npr. Read, Write...). Server jednostavno odgovara na te zahteve (poruke protokola), od kojih svaki sadrži dovoljno informacija da se zahtev izvrši.

Razmotrimo na primer jednostavnu aplikaciju koja čita fajl. Na slici 5 je prikazano koje sistemske pozive aplikacija obavlja i šta klijentski i serverski fajl sistemi rade povodom toga.

Daćemo nekoliko komentara vezanih za sliku. Treba primetiti da klijent prati sva bitna stanja pristupa fajlu, uključujući mapiranje fajl deskriptora na NFS upravljač fajla, kao i pokazivač na fajl (dokle je fajl pročitan). Ovo omogućava klijentu da prevede svaki zahtev za čitanje u odgovarajući Read zahtev protokola koji precizno kaže serveru koje bajtove iz fajla da pročita. Nakon uspešno završenog čitanja, klijent postavlja nov pokazivač na fajl (ofset). Naredna čitanja se obavljaju sa istim upravljačem fajla, ali različitim ofsetom.

Sledeće što treba primetiti je gde se server uključuje u komunikaciju. Kada se fajl otvori prvi put, klijentski fajl sistem šalje LookUp zahtev. Ako je putanja do fajla dugačka npr. /home/remzi/foo.txt, klijent će poslati 3 LookUp zahteva (jedan za home, jedan za remzi unutar home direktorijuma, jedan za fajl foo.txt unutar remzi direktorijuma).

Poslednje što treba primetiti je to kako svaki zahtev serveru ima sve informacije koje su potrebne da bi se taj zahtev ispunio. Ovo je od ključne važnosti jer se na taj način server lako oporavlja od otkaza, što će biti detaljnije diskutovano u nastavku. To omogućava da server ne mora imati podatke o stanju klijenta da bi mogao da odgovori na njegove zahteve.

Klijent	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request look for "foo" in root dir return foo's FH + attributes
Receive LOOKUP reply allocate file desc in open file table store foo's FH in table store current file position (0) return file descriptor to application	
read(fd, buffer, MAX);	
Index into open file table with fd get NFS file handle (FH) use current file position as offset	
Send READ (FH, offset=0, count=MAX)	
	Receive READ request use FH to get volume/inode num read inode from disk (or cache) compute block location (using offset) read data from disk (or cache) return data to client
Receive READ reply update file position (+bytes read) set current file position = MAX return data/error code to app	
read(fd, buffer, MAX);	
Same except offset=MAX and set current file position = 2*MAX	
read(fd, buffer, MAX);	
Same except offset=2*MAX and set current file position = 3*MAX	
close(fd);	
Just need to clean up local structures Free descriptor "fd" in open file table (No need to talk to server)	

Slika 5: **Čitanje fajla: Klijentske i serverske akcije**

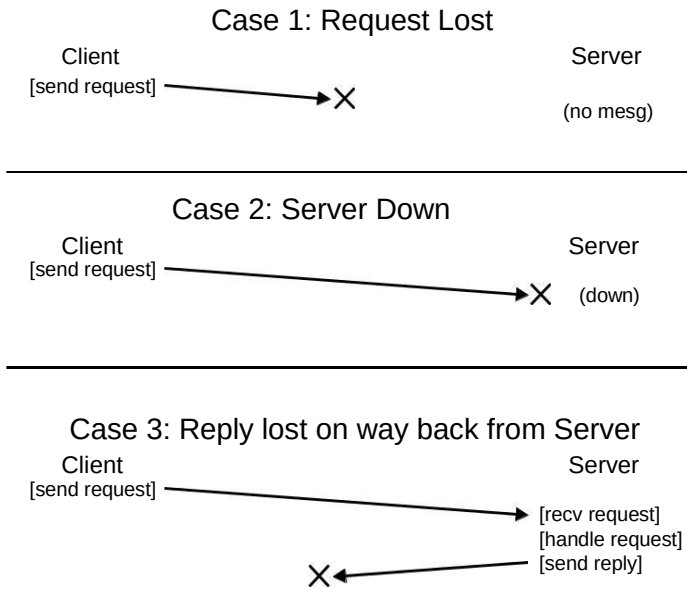
7 Upravljanje otkazom servera uz pomoć idempotentnih operacija

Kada klijent šalje poruku serveru, ponekad se desi da ne dobije odgovor. Postoji mnogo razloga zbog kojih se ovo dešava. U nekim slučajevima poruka može biti odbačena od strane same mreže. Takođe se može desiti da se i sam klijentov zahtev izgubi. Moguće je i da je server pao i da zbog toga nije u mogućnosti da odgovori na zahteve. Nakon nekog vremena, server će se restartovati i ponovo početi sa radom, ali u međuvremenu, svi zahtevi će biti izgubljeni. U svim ovim slučajevima klijentu ostaje dilema šta treba da radi kada server ne odgovara na zahteve u predviđenom vremenu.

Kod NFSv2, klijent rešava ove probleme vrlo jednostavno, ponovo pošalje zahtev. Nakon što klijent pošalje zahtev, pokreće tajmer. Ukoliko dobije odgovor od servera pre nego što tajmer istekne, znači da je sve prošlo kako treba. U suprotnom, ako tajmer istekne pre nego što klijent dobije odgovor, on će pretpostaviti da server nije obradio njegov zahtev, i poslaće ga ponovo.

Mogućnost klijenta da jednostavno pošalje ponovni zahtev (bez obzira na to šta je dovelo do greške) je ostvariva zbog važne osobine NFS zahteva: oni su idempotentni. Operaciju nazivamo idempotentnom ako je efekat izvršavanja te operacije više puta isto kao i efekat izvršavanja te operacije samo jednom. Na primer, ako zapamtimo neki broj u memoriji 3 puta, to je isto kao da smo ga zapamtili samo jednom. Ali ako bi smo uvećali taj broj 3 puta za 1, to nije isto kao da smo ga uvećali za 1 samo jednom. Operacije koje samo čitaju podatke su očigledno idempotentne. Za operacije koje menjaju podatke, treba razmotriti da li imaju ovu osobinu ili ne.

Srž projektovanja oporavka od otkaza kod NFS-a je idempotentnost većine operacija. LookUp i Read zahtevi su očigledno idempotentni jer oni čitaju podatke sa servera i ne menjaju ih. Zanimljivo je to da je i Write zahtev idempotentan. Ako bi na primer Write zahtev bio izgubljen, klijent može jednostavno da pošalje taj zahtev ponovo. Write zahtev u sebi sadrži podatke, broj bajtova koji treba da upiše, ali što je najbitnije, sadrži i ofset, odnosno odakle treba da počne sa upisivanjem tih podataka. Stoga ova operacija može biti ponavljana iznova i iznova, jer je ishod jednog Write zahteva isti kao i da je on ponovljen više puta.



Slika 6: **Tri vrste gubitka podataka/zahteva**

Zbog idempotentnosti kljent može da obradi sve greške na jedinstven način. Ako se Write zahtev izgubi u slučaju 1 na slici iznad, klijent će ponovo poslati taj zahtev i server će izvršiti pisanje. Slično će se desiti i kada server padne u trenutku slanja zahteva (slučaj 2). Poslednji slučaj je taj da server primi zahtev za pisanje, odradi pisanje i pošalje odgovor nazad klijentu. U slučaju da se taj odgovor izgubi (slučaj 3), klijent će ponovo poslati isti zahtev za pisanje. Kada server primi taj novi zahtev, jednostavno će ponovo izvršiti istu operaciju i poslati nazad odgovor klijentu. Ako ovaj put odgovor stigne do klijenta, sve se završilo kako treba.

Napomena: neke operacije nije lako napraviti da budu idempotentne. Na primer, ako pokušavamo da napravimo direktorijum koji već postoji, bićemo obavešteni da je došlo do greške prilkom pravljenja direktorijuma. U NFS-u, ako server dobije zahtev da napravi direktorijum i uspešno to izvrši, ali se odgovor klijentu izgubi u mreži, klijent će ponovo poslati isti zahtev, ali će dobiti odgovor o grešci, jer taj direktorijum već postoji.

8 Poboljšanje performansi: keširanje na strani klijenta

Distribuirani fajl sistemi su dobri iz mnogo razloga, ali slanje svih Read i Write zahteva kroz mrežu može dovesti do velikog problema u performansama. Mreža u globalu nije brza u poređenju sa memorijom ili diskom. Kako možemo rešiti ovaj problem performansi distribuiranih fajl sistema?

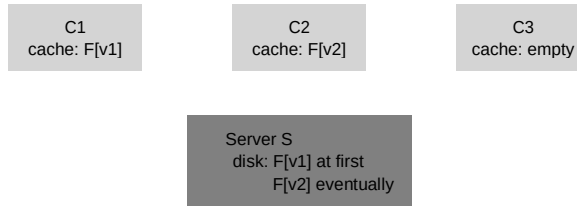
Odgovor je u tome, kao što ste verovatno već i primetili u naslovu, da se radi neka vrsta keširanja na klijentskoj strani. Kod NFS klijentskog fajl sistema se keširaju podaci i metapodaci, koji su dobijeni od strane servera. Iako je prvi zahtev skup (zahteva mrežnu komunikaciju), drugi zahtevi nisu toliko skupi.

Keš takođe služi kao privremeni bafer za pisanje. Kada klijentska aplikacija prvi put piše u fajl, ona te podatke smešta u svoju memoriju (onu istu memoriju u koje smešta i podatke koje dobije od servera) pre stvarnog pisanja tih podataka na serveru. Ovo baferovanje je korisno zbog toga što u trenutku kada klijent zahteva operaciju pisanja, odmah dobije odgovor da je uspešno izvšena, a u pozadini se ti podaci stave u bafer, i tek kasnije se stvarno upišu na server.

Dakle, klijent kešira podatke, i performanse su uglavnom odlične? To nije u potpunosti tačno. Dodavanje keširanja u bilo koji sistem sa više klijenata je veliki izazov, i taj problem nazivamo problem konzistentnosti keša.

9 Problem konzistentnosti keša

Problem konzistentnosti keša se najbolje demonstrira na primeru gde imamo dva klijenta i jedan server. Zamislite da klijent C1 čita fajl F i čuva kopiju tog fajla u svom lokalnom kešu. Neka sada klijent C2 prepiše taj isti fajl F (menja njegov sadržaj). Nazovimo prvu verziju fajla F sa Fv1, a drugu sa Fv2 (naravno i dalje je to samo jedan fajl F). Neka sada imamo i trećeg klijenta C3, koji još uvek nije pristupio fajlu F.



Slika 7: **Problem konzistentnosti keša**

Verovatno već možete naslutiti koji problem može da nastane. Postoje 2 potproblema. Prvi potproblem je taj da klijent C2 pamti svoju izmenu fajla u njegovom baferu, pre nego što je stvarno pošalje na server. U ovom slučaju, dok Fv2 stoji u memoriji klijenta C2, svaki pristup fajlu F od strane nekog drugog klijenta (C3) će vratiti Fv1 verziju fajla F. Na taj način, baferovanjem teksta za pisanje pre samog pisanja, ostali klijenti mogu dobiti staru verziju fajla, što može biti nepoželjno. Zamislite scenario u kome se prijavljujete na mašinu C2, menjate fajl, a zatim se odmah prijavite na mašinu C3 i dobijete staru verziju fajla. Ovaj problem konzistentnosti keša nazivamo vidljivost modifikacije (update visibility). Kada će drugi klijenti moći da vide promene od strane nekog klijenta?

Drugi potproblem konzistentnosti keša je zastareli keš. U ovom slučaju, recimo da je C2 završio sa svojim pisanjem i da se na serveru stvarno nalazi najnovija verzija fajla Fv2. U ovom slučaju, klijent C1 će u svojoj memoriji imati staru verziju fajla (Fv1), što je takođe nepoželjno.

Kod NFSv2 je problem konzistentnosti keša rešiv na 2 načina. Prvi je taj da u trenutku kada klijent završi sa pisanjem i zatvori datoteku, sve svoje promene šalje serveru. Ovaj način se zove flush-on-close. Njime se osiguravamo da će svaki naredni poziv za čitanje tog fajla dobiti najnoviju verziju.

Drugo, da bi se rešio problem zastarelog keša, klijenti jednostavno provere da li je bilo izmena na fajlu, pre nego što počnu da ga koriste iz svoje memorije. Kada se fajl otvara, klijentski fajl sistem će izdati serveru zahtev GetAttr. Kada mu ovaj vrati rezultate, on može videti u koje vreme je fajl bio poslednji put izmenjen na serveru. Ako je vreme modifikacije veće nego vreme kada je klijent dobio tu datoteku, klijent poništava svoju operaciju i uklanja fajl iz svog keša. Time se osigurava da će za naredno čitanje morati da ponovo preuzme datoteku sa servera, sa najnovijim sadžajem. Ako, sa druge strane, klijent vidi da ima najnoviju verziju fajla, on će je koristiti iz svoje memorije čime se dobija na performasama.

Kada su inženjeri u Sun-u implementirali ovakvo rešenje, dobili su novi problem: NFS server je bio preplavljen GetAttr zahtevima. Klijent je za svaki pristup fajlu morao da pošalje serveru zahtev kako bi ga pitao da li je neko menjao fajl u međuvremenu, iako je to retko bio slučaj.

Za ispravljanje ove situacije (donekle), svakom klijentu je dodat keš za atribute (attribute cache). Klijent će i dalje proveravati validnost fajla pre svakog pristupa, ali će u većini slučajeva odgovor dobiti iz svog keša. Kada se fajlu prvi put pristupi, njegovi atributi će biti smešteni u keš, a zatim će se odgovori na sve zahteve za atributima tog fajla u nekom vremenskom intervalu (3 sekunde) dobijati iz keša (i svi odgovori na zahteve u te 3 sekunde će biti uspešni).

10 Ocenjivanje konzistentnosti keša

Nekoliko završnih reči vezano za konzistentnost keša u NFS-u. Flush-on-close rešenje je uvelo dodatne probleme vezane za performanse. Ukoliko se na klijentu napravi neki privremeni fajl, koji će ubrzo biti obrisan, on takođe mora da postoji i na serveru. Bolja implementacija bi bila da takve fajlove klijent zadrži u svojoj memoriji i kasnije ih obriše, odnosno da potpuno izbaci komunikaciju sa serverom kada se radi o ovakvim fajlovima.

Što je još važnije, dodavanje keša za atribute je u mnogome otežalo tumačenje toga kojoj verziji fajla klijent pristupa. Nekada će dobiti poslednju verziju fajla, ali nekada neće. To se dešava zbog toga što u trenutku kada tajmer odbrojava, neki drugi klijent može da izmeni fajl, dok vi radite nad starim fajlom koji se nalazi u vašoj memoriji. Iako se u praksi retko dešavaju ove situacije, ne možemo da kažemo da je u potpunosti rešen taj problem.

11 Baferovanje poziva Write na serveru

Naš fokus do sada bio je keširanje na klijentskoj strani i tu se javlja većina zanimljivih pitanja. Međutim, NFS serveri su obično dobro hardverski opremljeni, imaju dosta memorije, zbog toga i keširanje na serverskoj strani može biti interesantno. Kada se podaci i metapodaci pročitaju sa diska, NFS server ih neće vratiti nazad na disk, već će ih zadržati u radnoj memoriji. Ovakav način može dovesti do potencijalnog poboljšanja performansi.

Još interesantniji slučaj je baferovanje pri pisanju. NFS serveri neće vratiti poruku o uspešnosti klijentu, sve dok se čitavo pisanje na neki trajni uređaj (disk) ne završi. Iako je server u mogućnosti da te podatke zapamti kod sebe u memoriji, pa da odmah zatim klijentu javi da je uspešno izvršio svoj posao i tek nakon toga upiše podatke na disk, on to neće uraditi. Zbog čega?

Odgovor leži u tome kako klijent obrađuje greške na serveru. Zamislite da su klijenti poslali serveru sledeće zahteve:

```
write(fd, a_buffer, size); // fill first block with a's
write(fd, b_buffer, size); // fill second block with b's
write(fd, c_buffer, size); // fill third block with c's
```

Ova pisanja će prepisati prva tri bloka datog fajla sa a-ovima, b-ovima i c-ovima respektivno. Ukoliko je prvobitna verzija fajla izgledala ovako:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Očkujemo da će efekat ova tri pisanja dovesti do sledećeg izgleda fajla:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Pretpostavimo sada, primera radi, da su ova tri pisanja klijenta izdata kao tri različita Write zahteva. Pretpostavimo da je prva Write poruka primeljena od strane servera, i da ju je on izvršio i vratio klijentu poruku o uspešnosti. Neka sada server primi zahtev od drugog klijenta, ali njega ne izvrši odma već ga samo baferuje, a drugom klijentu vrati poruku da je uspešno izvršio upisivanje. Ako sada server padne, nikada neće upisati podatke drugog klijenta na disk. Ako mu i treći klijent izda zahtev za pisanje i on ga uspešno izvrši, tada će fajl na serveru izgledati ovako (treba imati u vidu da su sva 3 klijenta primili poruku da je pisanje uspešno izvršeno):

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Pošto je server rekao drugom klijentu da je njegov zahtev uspešno obrađen, pre nego što je stvarno upisao podatke, došlo je do greške. U zavisnosti od aplikacije, ovaj izhod može biti fatalan. Da bi izbegli ovaj problem, NFS serveri moraju da potvrde da su podatke upisali na tvrdi disk, pre nego što jave klijentu da su to uspešno izvršili. Ovakvo ponašanje omogućava klijentu da detektuje greške na serverskoj strani, i ponavlja svoje zahteve sve dok mu server ne vrati da je to uspešno odradio. Na taj način se obezbeđujemo da nikada nećemo doći u situaciju u koju smo došli na prethodnom primeru.

Problem kod NFS-a je taj što zahtevi za pisanjem mogu biti veliki nosilac performasni. Neke kompanije kao što je Network Appliance su napravljene samo iz razloga da naprave NFS servere koji mogu da izvršavaju operacije pisanja jako brzo. Jedan od trikova koji su oni primenjivali je da kada dobiju Write zahtev, te podatke smeste na memoriju koja podržava baterije. To omogućava serveru da odmah klijentu vrati poruku da je uspešno obradio njegov zahtev, a da pritom ne mora da strahuje od pada servera, jer time neće izgubiti podatke. Drugi način kojim su rešili ovaj problem je pravljenje fajl sistema koji je dizajniran tako da jako brzo izvršava operacije pisanja.

12 Zaključak

Videli smo uvod u NFS distribuirani fajl sistem. NFS je usredsređen na ideju jednostavnog i brzog oporavka u slučaju otkaza servera, a ovaj cilj postiže pažljivim dizajnom protokola. Idempotentnost operacija je od ključne važnosti zato što klijent može sa sigurnošću da ponavlja neuspele operacije, i to je OK bilo da je server obradio te zahteve ili ne.

Takođe smo videli kako keširanje na više klijenata i jednom serveru može zakomplikovati stvari. Konkretno, sistem mora da reši problem konzistentnosti keša kako bi se mogao ponašati pouzdano. Međutim, NFS to radi na malo čudan način, što ponekad može dovesti do neočekivanog ponašanja. Konačno, videli smo kako keširanje na serveru može biti problematično: pisanje na serveru se mora odraditi tako da se podaci upišu u trajnu memoriju pa se nakon toga klijentu šalje poruka o uspešnosti (inače podaci mogu da se izgube).

Nismo govorili o drugim problemima koji mogu da budu bitni, posebno o sigurnosti. Sigurnost u ranijim NFS implementacijama bila je slaba. Tada se sa lakoćom jedan korisnik na klijentskoj strani mogao predstaviti kao neko drugi i na taj način pristupati njegovim fajlovima. Naredne implementacije NFS-a rešile su ove nedostatak.