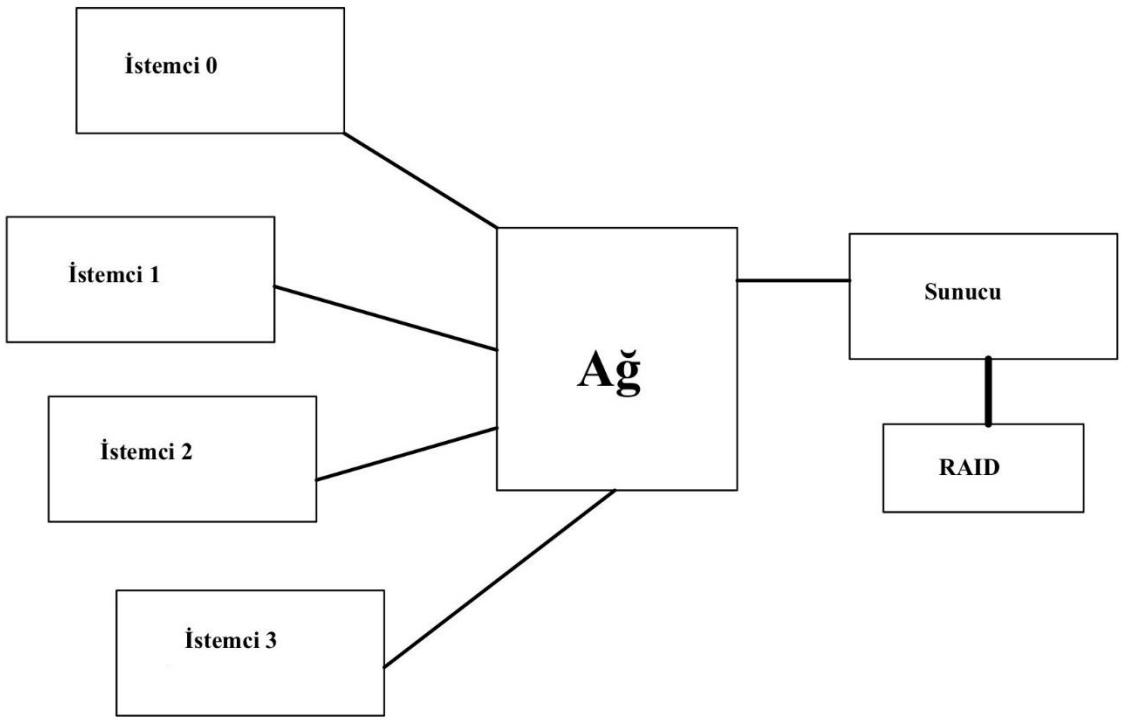
Sun'ın Ağ Dosya Sistemi (NFS)

Dağıtılmış istemci/sunucu bilgi işlemin ilk kullanımlarından biri, dağıtılmış dosya sistemleri alanındaydı. Böyle bir ortamda, bir dizi istemci makine ve bir (veya birkaç) sunucu vardır; sunucu, verileri disklerinde depolar ve istemciler, iyi biçimlendirilmiş protokol mesajları aracılığıyla veri ister. Şekil 49.1, temel kurulumu göstermektedir.



Şekil 49.1: Genel Bir İstemci/Sunucu Sistemi

Resimden de görebileceğiniz gibi, sunucunun diskleri vardır ve istemciler, bu disklerdeki dizinlerine ve dosyalarına erişmek için bir ağ üzerinden mesajlar gönderir. Neden bu düzenlemeyle uğraşıyoruz? (yani, neden istemcilerin yerel disklerini kullanmalarına izin vermiyoruz?) Öncelikle bu kurulum, verilerin istemciler arasında kolayca paylaşılmasını sağlar. Bu nedenle, bir makinedeki (İstemci 0) bir dosyaya erişir ve daha sonra başka bir makineyi (İstemci 2) kullanırsanız, aynı dosya sistemi görünümüne sahip olursunuz. Verileriniz doğal olarak bu farklı makineler arasında paylaşılır. İkincil bir fayda, **merkezi yönetimdir(centralized administration;);** örneğin, dosyaların yedeklenmesi çok sayıda istemci yerine birkaç sunucu makinesinden yapılabilir. Diğer bir avantaj güvenlik olabilir;

tüm sunucuların kilitli bir makine dairesinde olması, belirli türden sorunların ortaya çıkmasını önler.

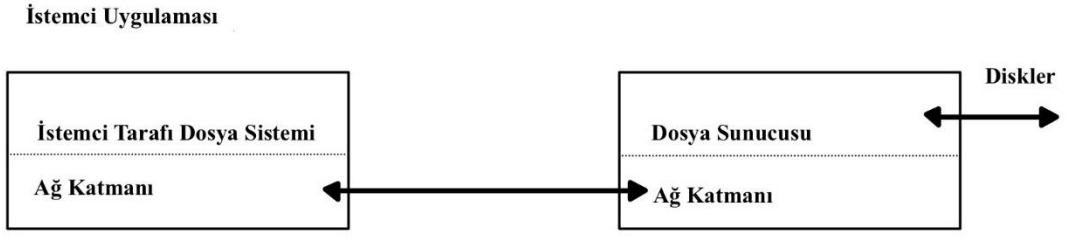
PÜF NOKTA : DAĞITILMIŞ DOSYA SİSTEMİ NASIL OLUŞTURULUR

Dağıtılmış bir dosya sistemi nasıl oluşturulur? Düşünülmesi gereken temel hususlar nelerdir? Yanlış yapmak kolay olan nedir? Mevcut sistemlerden ne öğrenebiliriz?

49.1 Temel Bir Dağıtılmış Dosya Sistemi

Şimdi basitleştirilmiş bir dağıtılmış dosya sisteminin mimarisini inceleyeceğiz. Basit bir istemci/sunucu dağıtılmış dosya sistemi, şu ana kadar incelediğimiz dosya sistemlerinden daha fazla bileşene sahiptir. İstemci tarafında, **istemci tarafı dosya sistemi(client-side file system)** aracılığıyla dosyalara ve dizinlere erişen istemci uygulamaları vardır. Bir istemci uygulaması, sunucuda depolanan dosyalara erişmek için istemci tarafı dosya sistemine (open(), read(), write(), close(), mkdir(), vb. gibi) **sistem çağrıları(system calls)** yapar. . Bu nedenle, istemci uygulamalarına, dosya sistemi, belki performans dışında, yerel (disk tabanlı) bir dosya sisteminden farklı görünmemektedir; bu şekilde, dağıtılmış dosya sistemleri dosyalara **şeffaf(transparent)** erişim sağlar, bu bariz bir hedeftir; ne de olsa, farklı bir API seti gerektiren veya başka bir şekilde kullanımı zahmetli olan bir dosya sistemini kim kullanmak isterdi?

İstemci tarafı dosya sisteminin rolü, bu sistem çağrılarına hizmet vermek için gereken eylemleri yürütmektir. Örneğin, istemci bir read() isteği gönderirse, istemci tarafı dosya sistemi, belirli bir bloğu okumak için **sunucu tarafı dosya sistemine(server-side file system)** (veya yaygın olarak adlandırıldığı şekliyle **dosya sunucusuna(file server)**) bir mesaj gönderebilir; dosya sunucusu daha sonra bloğu diskten (veya kendi bellek içi önbelleğinden) okuyacak ve istemciye istenen verileri içeren bir mesaj gönderecektir. İstemci tarafı dosya sistemi daha sonra verileri read() sistem çağrısına sağlanan kullanıcı arabelleğine kopyalayacak ve böylece istek tamamlanacaktır. İstemcide aynı bloğun bir sonraki okumasının() istemci belleğinde veya istemcinin diskinde **önbelleğe alınabileceğini(cached)** unutmayın; en iyi durumda, ağ trafiğinin oluşturulmasına gerek yoktur.



Şekil 49.2: Dağıtılmış Dosya Sistemi Mimarisi

Bu basit genel bakıştan, bir istemci/sunucu dağıtılmış dosya sisteminde iki önemli yazılım parçası olduğunu anlamalısınız: istemci tarafı dosya sistemi ve dosya sunucusu. Birlikte davranışları, dağıtılmış dosya sisteminin davranışını belirler. Şimdi belirli bir sistemi inceleme zamanı: Sun'ın Ağ Dosya Sistemi (NFS).

BİR KENARA: SUNUCULAR NEDEN Çöküyor?

NFSv2 protokolünün ayrıntılarına girmeden önce merak ediyor olabilirsiniz: sunucular neden çöküyor? Pekala, tahmin edebileceğiniz gibi, pek çok sebep var. Sunucular (geçici olarak) bir **elektrik kesintisinden(power outage)** muzdarip olabilir; sadece güç geri geldiğinde makineler yeniden başlatılabilir.Sunucular genellikle yüz binlerce **hatta(power outage)** milyonlarca kod satırından oluşur; bu nedenle, hataları vardır (iyi yazılımların bile yüz veya bin satır kod başına birkaç hatası vardır) ve bu nedenle, sonunda çökmelerine neden olacak bir hatayı tetiklerler. Ayrıca bellek sızıntıları da var; küçük bir bellek sızıntısı bile bir sistemin belleğinin bitmesine ve çökmesine neden olur. Ve son olarak, dağıtık sistemlerde, istemci ile sunucu arasında bir ağ vardır; ağ garip davranıyorsa (örneğin, bölümlenmişse(partitioned) ve istemciler ve sunucular çalışıyor ancak iletişim kuramıyorsa), uzaktaki bir makine çökmüş gibi görünebilir, ancak gerçekte şu anda ağ üzerinden erişilemez.

49.2 NFS'ye Açık

En eski ve oldukça başarılı dağıtılmış sistemlerden biri Sun Microsystems tarafından geliştirilmiştir ve Sun Network Dosya Sistemi (veya NFS) [S86] olarak bilinir. Sun, NFS'yi tanımlarken alışılmadık bir yaklaşım benimsedi: tescilli ve kapalı bir sistem

oluşturmak yerine, Sun bunun yerine istemcilerin ve sunucuların iletişim kurmak için kullanacakları tam mesaj biçimlerini belirten **açık bir protokol(open protocol)** geliştirdi. Farklı gruplar kendi NFS sunucularını geliştirebilir ve böylece bir NFS pazarında rekabet edebilir.
birlikte çalışabilirliği korurken. İşe yaradı: bugün NFS sunucuları satan birçok şirket var (Oracle/Sun, NetApp [HLM94], EMC, IBM ve diğerleri dahil) ve NFS'nin yaygın başarısı muhtemelen bu "açık pazar" yaklaşımına bağlanıyor.

49.3 Odak: Basit ve Hızlı Sunucu Çökmesinden Kurtulma

Bu bölümde, yıllardır standart olan klasik NFS protokolünü (sürüm 2, diğer adıyla NFSv2) tartışacağız; NFSv3'e geçişte küçük değişiklikler, NFSv4'e geçişte ise daha büyük ölçekli protokol değişiklikleri yapıldı. Ancak, NFSv2 hem harika hem de sinir bozucu ve bu nedenle odak noktamız olarak hizmet ediyor.

NFSv2'de protokol tasarımındaki ana hedef, basit ve hızlı sunucu çökmesinden kurtarmaydı. Çok istemcili, tek sunuculu bir ortamda bu hedef çok anlamlıdır; sunucunun çalışmadığı (veya kullanılamadığı) herhangi bir dakika, tüm istemci makineleri (ve kullanıcılarını) mutsuz ve verimsiz hale getirir. Böylece, sunucu gittiği gibi, tüm sistem de gider.

49.4 Hızlı Çökme Kurtarmanın Anahtarı: Vatansızlık

Bu basit hedef, NFSv2'de **durum bilgisi olmayan(stateless)** bir protokol olarak adlandırdığımız şeyi tasarlayarak gerçekleştirilir. Sunucu, tasarımı gereği, her istemcide neler olup bittiğiyle ilgili hiçbir şeyi takip etmez. Örneğin, sunucu hangi istemcilerin hangi blokları önbelleğe aldığını veya her istemcide o anda hangi dosyaların açık olduğunu veya bir dosya için geçerli dosya işaretçisinin konumunu vb. bilmez. Basitçe söylemek gerekirse, sunucu, istemcilerin ne yaptığıyla ilgili hiçbir şeyi izlemez; bunun yerine protokol, talebi tamamlamak için gerekli olan tüm bilgileri her protokol talebine iletmek üzere tasarlanmıştır. Şimdi değilse, aşağıda protokolü daha ayrıntılı olarak tartıştığımız için bu durum bilgisiz yaklaşım daha anlamlı olacaktır.

Durum bilgili(stateful) (durumsuz değil) bir protokol örneği için open() sistem çağrısını düşünün. Bir yol adı verildiğinde, open() bir dosya tanıtıcısı (bir tamsayı) döndürür. Bu

tanımlayıcı, bu uygulama kodunda olduğu gibi, çeşitli dosya bloklarına erişmek için sonraki okuma() veya yazma() isteklerinde kullanılır (uygun olduğuna dikkat edin). sistem çağrılarının hata kontrolü, alan nedeniyle ihmal edilmiştir):

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

Şekil 49.3: İstemci Kodu: Bir Dosyadan Okuma

Şimdi, istemci tarafı dosya sisteminin, sunucuya "'foo' dosyasını aç ve bana bir tanımlayıcı geri ver" diyen bir protokol mesajı göndererek dosyayı açtığını hayal edin. Dosya sunucusu daha sonra dosyayı kendi tarafında yerel olarak açar ve tanımlayıcıyı istemciye geri gönderir. Sonraki okumalarda istemci uygulaması, read() sistem çağrısını çağırmak için bu tanımlayıcıyı kullanır; istemci tarafı dosya sistemi daha sonra tanımlayıcıyı bir mesajla dosya sunucusuna iletir ve "tanımlayıcı tarafından atıfta bulunulan dosyadan bazı baytları okuyun, sizi buraya iletiyorum" der.

Bu örnekte, dosya tanıtıcı, istemci ile sunucu arasındaki **paylaşılan durumun(shared state)** bir parçasıdır (Ousterhout bu dağıtılmış durumu [O91] olarak adlandırır). Paylaşılan durum, yukarıda ima ettiğimiz gibi, kilitlenme kurtarmayı zorlaştırır. İlk okuma tamamlandıktan sonra, ancak istemci ikincisini yayınlamadan önce sunucunun çöktüğünü hayal edin. Sunucu tekrar çalışır duruma geldikten sonra, istemci ikinci okumayı yayınlar. Ne yazık ki, sunucunun fd'nin hangi dosyaya atıfta bulunduğu hakkında hiçbir fikri yok; bu bilgi geçiciydi (yani bellekte) ve bu nedenle sunucu çöktüğünde kayboldu. Bu durumla başa çıkmak için, istemci ve sunucunun, sunucuya bilmesi gerekenleri söyleyebilmek için (bu durumda) istemcinin belleğinde yeterli bilgiyi sakladığından emin olacağı bir tür **kurtarma protokolüne(recovery protocol)** dahil olması gerekir. , bu dosya tanıtıcı fd, foo dosyasına atıfta bulunur).

Durum bilgisi olan bir sunucunun istemci çökmeleriyle uğraşmak zorunda olduğu gerçeğini düşündüğünüzde durum daha da kötüleşir. Örneğin, bir dosyayı açan ve ardından

çöken bir istemci düşünün. open(), sunucuda bir dosya tanıtıcı kullanır; Sunucu, belirli bir dosyayı kapatmanın uygun olduğunu nasıl bilebilir? Normal çalışmada, bir istemci en sonunda close() işlevini çağırır ve böylece sunucuya dosyanın kapatılması gerektiğini bildirir. Bununla birlikte, bir istemci çöktüğünde, sunucu hiçbir zaman bir kapatma() almaz ve bu nedenle, dosyayı kapatmak için istemcinin kilitlendiğini fark etmesi gerekir.

Bu nedenlerden dolayı, NFS tasarımcıları durum bilgisi olmayan bir yaklaşım izlemeye karar verdiler: her istemci işlemi, isteği tamamlamak için gereken tüm bilgileri içerir. Süslü çarpışma kurtarma gerekmez; sunucu yeniden çalışmaya başlar ve bir istemci, en kötü ihtimalle, bir isteği yeniden denemek zorunda kalabilir

49.5 NFSv2 Protokolü

Böylece NFSv2 protokol tanımına ulaşıyoruz. Sorun bildirimimiz basit

> ÖNEMLİ NOKTA: DURUMSUZ BİR DOSYA PROTOKOLÜ NASIL TANIMLANIR
> Durumsuz çalışmayı etkinleştirmek için ağ protokolünü nasıl tanımlayabiliriz? Açıkçası, open() gibi durum bilgisi olan çağrılar tartışmanın bir parçası olamaz (çünkü sunucunun açık dosyaları izlemesini gerektirir); ancak, istemci uygulaması dosyalara ve dizinlere erişmek için open(), read(), write(), close() ve diğer standart API çağrılarını çağırmak isteyecektir. Bu nedenle, rafine bir soru olarak, protokolü hem durumsuz olacak hem de POSIX dosya sistemi API'sini destekleyecek şekilde nasıl tanımlarız?

NFS protokolünün tasarımını anlamanın anahtarı, **dosya tanıtıcısını(file handle)** anlamaktır. Dosya tutamaçları, belirli bir işlemin üzerinde çalışacağı dosya veya dizini benzersiz şekilde tanımlamak için kullanılır; bu nedenle, protokol isteklerinin çoğu bir dosya tanıtıcısı içerir.

Bir dosya tanıtıcısının üç önemli bileşeni olduğunu düşünebilirsiniz: bir birim tanımlayıcısı, bir inode numarası ve bir nesil numarası; birlikte, bu üç öğe, bir müşterinin erişmek istediği bir dosya veya dizin için benzersiz bir tanımlayıcı içerir. Birim tanımlayıcısı, isteğin hangi dosya sistemine atıfta bulunduğunu sunucuya bildirir (bir NFS sunucusu birden fazla dosya sistemini dışa aktarabilir); inode numarası, sunucuya, isteğin o bölümdeki hangi dosyaya eriştiğini söyler. Son olarak, bir inode numarası yeniden kullanılırken nesil numarası gereklidir; sunucu, bir inode numarası yeniden kullanıldığında

bunu artırarak, eski bir dosya tanıtıcısına sahip bir istemcinin yanlışlıkla yeni tahsis edilen dosyaya erişememesini sağlar.

İşte protokolün bazı önemli parçalarının bir özeti; protokolün tamamı başka bir yerde mevcuttur (NFS'ye [C00] mükemmel ve ayrıntılı bir genel bakış için Callaghan'ın kitabına bakın).

NFSPROC GETATTR	dosya tanıtıcısı döndürür: Öznitellikler
NFSPROC SETATTR	dosya tanıtıcısı, öznitelikler İadeler: -
NFSPROC LOOKUP	dizin dosya tanıtıcısı, aranacak dosya/dir adı döndürür: dosya tanıtıcısı
NFSPROC READ	dosya tanıtıcısı, ofset, sayım veriler, nitelikler
NFSPROC WRITE	dosya tanıtıcısı, ofset, sayım, veri Öznitelikler
NFSPROC CREATE	dizin dosya tanıtıcısı, dosyanın adı, nitelikler –
NFSPROC REMOVE	dizin dosya tanıtıcısı, kaldırılacak dosyanın adı –
NFSPROC MKDIR	dizin dosya tanıtıcısı, dizin adı, öznitelikler dosya tanıtıcısı
NFSPROC RMDIR	dizin dosya tanıtıcısı, kaldırılacak dizinin adı –
NFSPROC READDIR	dizin tanıtıcısı, okunacak bayt sayısı, tanımlama bilgisi döndürür: dizin girişleri, çerez (daha fazla giriş almak için)

Şekil 49.4: NFS Protokolü: Örnekler

Protokolün önemli bileşenlerini kısaca vurgularız. İlk önce, ARA protokol mesajı, daha sonra dosya verilerine erişmek için kullanılan bir dosya tanıtıcısı elde etmek için kullanılır.

İstemci, aranacak bir dosya tanıtıcısını ve bir dosyanın adını iletir ve bu dosyanın (veya dizinin) tanıtıcısı ve öznitelikleri sunucudan istemciye geri iletilir.

Örneğin, istemcinin zaten bir dosya sisteminin (/) kök dizini için bir dizin dosya tanıtıcısına sahip olduğunu varsayalım (aslında bu, istemciler ve sunucuların ilk olarak birbirine bağlanma şekli olan NFS **bağlama protokolü(mount protocol)** aracılığıyla elde edilir; biz bunu yapmayız. kısa olması için bağlama protokolünü burada tartışın). İstemcide çalışan bir uygulama /foo.txt dosyasını açarsa, istemci tarafındaki dosya sistemi sunucuya bir arama isteği gönderir ve sunucuya kök dosya tanıtıcısını ve foo.txt adını verir; başarılı olursa, foo.txt için dosya tanıtıcısı (ve öznitelikleri) döndürülür.

Merak ediyorsanız, öznitelikler yalnızca dosya sisteminin her dosya hakkında izlediği, dosya oluşturma zamanı, son değiştirilme zamanı, boyut, sahiplik ve izin bilgileri gibi alanlar dahil olmak üzere meta verilerdir, yani aynı türden bilgilerdir. Eğer geri alırsan bir dosyada stat() olarak adlandırılır.

Bir dosya tanıtıcı kullanılabilir olduğunda, müşteri sırasıyla dosyayı okumak veya yazmak için bir dosya üzerinde READ ve WRITE protokol mesajları verebilir. READ protokol mesajı, protokolün dosyanın dosya tanıtıcısı boyunca dosya içindeki ofset ve okunacak bayt sayısı ile birlikte geçmesini gerektirir. Sunucu daha sonra okumayı yayınlayabilir (sonuçta tanıtıcı, sunucuya hangi birimin ve hangi inode'dan okunacağını söyler ve ofset ve sayı ona dosyanın hangi baytlarını okuyacağını söyler) ve verileri istemciye döndürür (veya bir arıza varsa bir hata). YAZMA, verilerin istemciden sunucuya iletilmesi ve yalnızca bir başarı kodu döndürülmesi dışında benzer şekilde işlenir.

Son bir ilginç protokol mesajı GETATTR isteğidir; bir dosya tanıtıcısı verildiğinde, dosyanın son değiştirilme zamanı da dahil olmak üzere o dosyanın özniteliklerini getirir. Önbelleğe almayı tartışırken aşağıda NFSv2'de bu protokol isteğinin neden önemli olduğunu göreceğiz (nedenini tahmin edebiliyor musunuz?)

49.6 Protokolden Dağıtılmış Dosya Sistemine

Umarız artık bu protokolün istemci tarafı dosya sistemi ve dosya sunucusu genelinde nasıl bir dosya sistemine dönüştürüldüğüne dair bir fikir ediniyorsunuzdur. İstemci tarafı dosya sistemi açık dosyaları izler ve genellikle uygulama isteklerini ilgili protokol mesajları grubuna çevirir. Sunucu, her biri isteği tamamlamak için gereken tüm bilgileri içeren protokol mesajlarına yanıt verir.

Örneğin, bir dosyayı okuyan basit bir uygulamayı ele alalım. Diyagramda (Şekil 49.5), uygulamanın hangi sistem çağrılarını yaptığını ve istemci tarafı dosya sisteminin ve dosya sunucusunun bu tür çağrılara yanıt olarak ne yaptığını gösteriyoruz.

Şekil hakkında birkaç yorum. İlk olarak, istemcinin, tamsayı dosya tanıtıcısının bir NFS dosya tanıtıcısına eşlenmesi ve geçerli dosya işaretçisi dahil olmak üzere dosya erişimiyle ilgili tüm durumu nasıl izlediğine dikkat edin. Bu, istemcinin her bir okuma isteğini (fark etmiş olabileceğiniz gibi, okunacak uzaklığı açıkça belirtmemiş olabilirsiniz), sunucuya dosyadan tam olarak hangi baytların okunacağını söyleyen düzgün biçimlendirilmiş bir okuma protokolü mesajına dönüştürmesini sağlar. Başarılı bir okumanın ardından, istemci geçerli dosya konumunu günceller; sonraki okumalar aynı dosya tanıtıcısı ile ancak farklı bir uzaklık ile verilir.

İkinci olarak, sunucu etkileşimlerinin nerede gerçekleştiğini fark edebilirsiniz. Dosya ilk kez açıldığında, istemci tarafı dosya sistemi bir ARA istek mesajı gönderir. Aslında, uzun bir yol adının geçilmesi gerekiyorsa (örneğin,/home/remzi/foo.txt), müşteri üç LOOKUP gönderir: biri / dizininde evi aramak için, biri evde remzi'yi aramak için ve son olarak bir tane. remzi'de foo.txt dosyasını aramak için.

Üçüncüsü, her sunucu isteğinin, isteği bütünüyle tamamlamak için gereken tüm bilgilere nasıl sahip olduğunu fark edebilirsiniz. Bu tasarım noktası, şimdi daha ayrıntılı olarak tartışacağımız gibi, sunucu arızasından zarafetle kurtulabilmek için kritik öneme sahiptir; sunucunun isteğe yanıt verebilmesi için duruma ihtiyaç duymamasını sağlar.

Alıcı	**Sunucu**
fd = open(”/foo”, ...);	
LOOKUP (rootdir FH, ”foo”) gönder	
	LOOKUP isteğini alın, kök dizinde “foo” arayın, foo’nun FH+ niteliklerini döndürün.
ARAMA yanıtı al	
dosya açıklamasını açık dosya tablosunda tahsis et	
foo'nun FH'sini tabloda sakla	
geçerli dosya konumunu sakla (0)	
dosya tanıtıcısını uygulamaya döndür	

read(fd, arabellek, MAKS);
fd ile açık dosya tablosuna dizin
NFS dosya tanıtıcısını (FH) al
geçerli dosya konumunu ofset olarak kullan
READ gönder (FH, ofset=0, sayım=MAKS)

READ isteği al
birim/inode sayısını almak için FH'yi kullanın
diskten (veya önbellekten) inode oku
blok konumunu hesaplama (ofset kullanarak)
diskten (veya önbellekten) veri oku
verileri istemciye iade et

READ yanıtını al
dosya konumunu güncelle (+okunan bayt)
geçerli dosya konumunu ayarla = MAKS
uygulamaya veri/hata kodu döndür

READ(fd, arabellek, MAKS);
Offset=MAX ve set geçerli dosya konumu = 2*MAX dışında aynı

READ(fd, arabellek, MAKS);
Ofset=2*MAKS dışında aynı ve mevcut dosya konumunun ayarlanması = 3*MAKS

---

close(fd);
Sadece yerel yapıları temizlemeniz gerekiyor
Açık dosya tablosunda serbest tanımlayıcı ”fd”
(Sunucuyla konuşmaya gerek yok)

Şekil 49.5: Dosya Okuma: İstemci Tarafı ve Dosya Sunucusu Eylemleri

İPUCU: İDEMPOTENS GÜÇLÜDÜR

Bağımsızlık(Idempotency), güvenilir sistemler oluştururken yararlı bir özelliktir. Bir işlem birden çok kez düzenlenebildiğinde, işlemin başarısızlığını ele almak çok daha kolaydır; tekrar deneyebilirsin. Bir operasyon etkisiz değilse, hayat daha da zorlaşır.

49.7 Belirsiz İşlemlerle Sunucu Hatasını Ele Alma

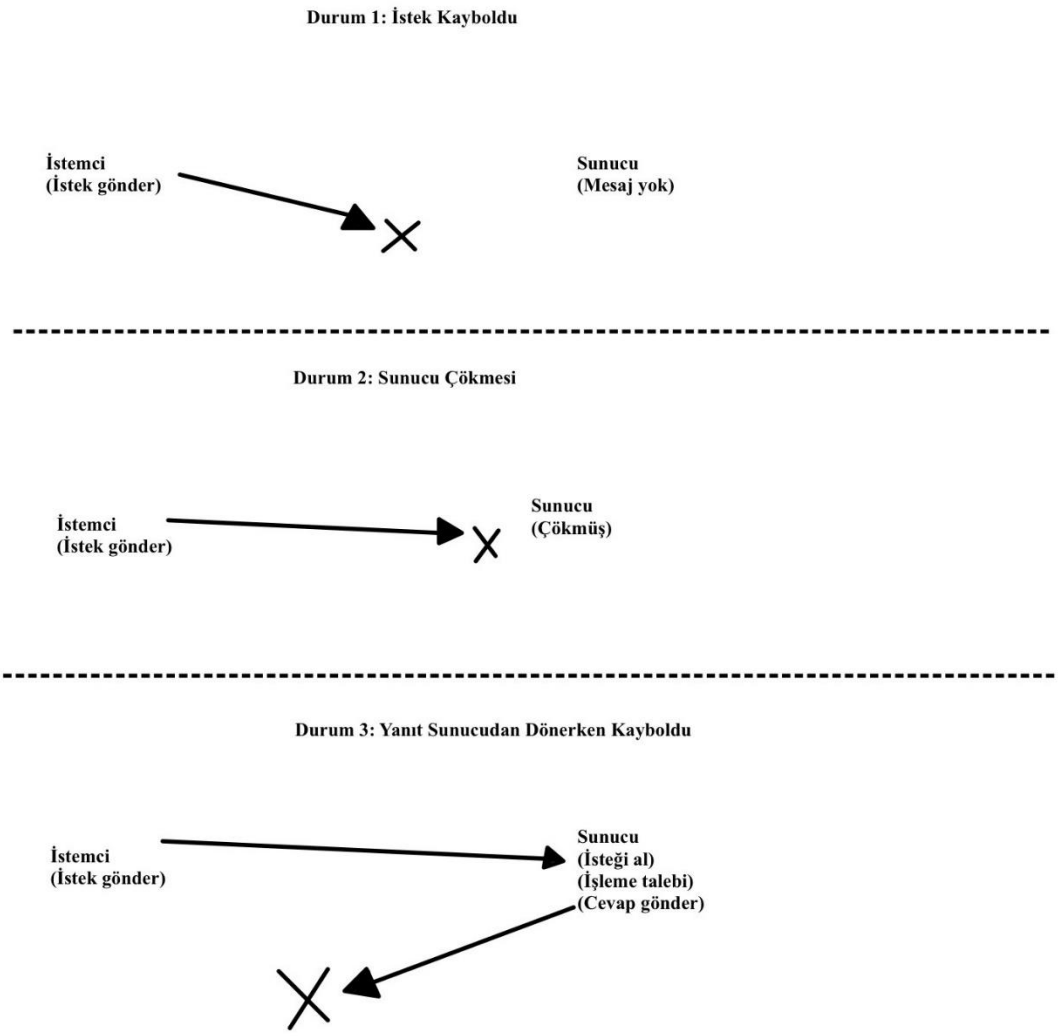
Bir istemci sunucuya bir mesaj gönderdiğinde, bazen bir yanıt almaz. Bu yanıt başarısızlığının birçok olası nedeni vardır. Bazı durumlarda, mesaj ağ tarafından bırakılabilir; ağlar mesajları kaybeder ve bu nedenle istek veya yanıt kaybolabilir ve bu nedenle müşteri hiçbir zaman yanıt alamaz.

Sunucunun çökmüş olması ve bu nedenle şu anda mesajlara yanıt vermemesi de mümkündür. Bir süre sonra sunucu yeniden başlatılacak ve yeniden çalışmaya başlayacaktır, ancak bu arada tüm istekler kaybolmuştur. Tüm bu durumlarda, müşterilere bir soru kalır: Sunucu zamanında yanıt vermediğinde ne yapmalıdırlar?

NFSv2'de bir istemci, tüm bu hataları tek, tek tip ve zarif bir şekilde ele alır: yalnızca isteği yeniden dener. Spesifik olarak, isteği gönderdikten sonra, müşteri belirli bir süre sonra kapanacak bir zamanlayıcı ayarlar. Zamanlayıcı kapanmadan önce bir yanıt alınırsa, zamanlayıcı iptal edilir ve her şey yolundadır. Ancak herhangi bir yanıt alınmadan zamanlayıcı kapanırsa, müşteri isteğin işlenmediğini varsayar ve yeniden gönderir. Sunucu yanıt verirse, her şey yolundadır ve istemci sorunu düzgün bir şekilde çözmüştür.

İstemcinin isteği yeniden deneyebilmesi (hatanın sebebi ne olursa olsun), çoğu NFS isteğinin önemli bir özelliğinden kaynaklanmaktadır: bunlar önemsizdir. İşlemi birden çok kez gerçekleştirmenin etkisi, işlemi tek bir kez gerçekleştirmenin etkisine eşdeğer olduğunda, bir işleme **etkisiz(idempotent)** işlem denir. Örneğin, bir değeri bir bellek konumuna üç kez kaydederseniz, bu, bir kez yapmakla aynıdır; böylece "değeri belleğe kaydet" etkisiz bir işlemdir. Bununla birlikte, bir sayacı üç kez artırırsanız, bu, yalnızca bir kez yapmaktan farklı bir miktarla sonuçlanır; bu nedenle, "artış sayacı" etkisiz değildir. Daha genel olarak, yalnızca verileri okuyan herhangi bir işlem açıkça önemsizdir; verileri güncelleyen bir işlemin, bu özelliğe sahip olup olmadığını belirlemek için daha dikkatli bir şekilde ele alınması gerekir.

NFS'de kilitlenme kurtarma tasarımının kalbi, en yaygın işlemlerin yetersizliğidir. ARAMA ve OKUMA istekleri, yalnızca dosya sunucusundan bilgi okudukları ve onu güncellemedikleri için önemsiz derecede önemsizdir. Daha da ilginci, YAZMA istekleri de önemsizdir. Örneğin, bir YAZMA başarısız olursa, istemci yeniden deneyebilir. YAZ mesajı, verileri, sayımı ve (önemli olarak) verilerin yazılacağı kesin ofseti içerir. Böylece, birden fazla yazmanın sonucunun tek bir yazmanın sonucuyla aynı olduğu bilgisi ile tekrar edilebilir.



Şekil 49.6: Üç Kayıp Türü

Bu şekilde, müşteri tüm zaman aşımlarını birleşik bir şekilde işleyebilir. Bir YAZMA isteği basitçe kaybedilirse (Yukarıdaki Durum 1), istemci bunu yeniden deneyecek, sunucu yazmayı gerçekleştirecek ve her şey yoluna girecek. İstek gönderilirken sunucu kapalıysa, ancak ikinci istek gönderildiğinde yedeklenip çalışıyorsa ve yine her şey istenildiği gibi çalışıyorsa aynı şey olur (Durum 2). Son olarak, sunucu aslında YAZMA isteğini alabilir, diskine yazma işlemini gerçekleştirebilir ve bir yanıt gönderebilir. Bu yanıt kaybolabilir (Durum 3), yine müşterinin isteği yeniden göndermesine neden olabilir. Sunucu isteği tekrar aldığında, tamamen aynı şeyi yapacaktır: verileri diske yazacak ve bunu yaptığını yanıtlayacaktır. İstemci bu kez yanıtı alırsa, her şey yeniden yolundadır ve bu nedenle istemci, hem mesaj kaybını hem de sunucu arızasını tekdüze bir şekilde ele almıştır. Düzenli!

Küçük bir yan bilgi: bazı operasyonları etkisiz hale getirmek zordur. Örneğin, zaten var olan bir dizini oluşturmaya çalıştığınızda, mkdir isteğinin başarısız olduğu konusunda bilgilendirilirsiniz. Bu nedenle, NFS'de, dosya sunucusu bir MKDIR protokol mesajı alır ve bunu başarılı bir şekilde yürütür ancak yanıt kaybolursa, istemci bunu tekrarlayabilir ve aslında işlem ilk başta başarılı olduğunda ve ardından yalnızca yeniden denemede başarısız olduğunda bu hatayla karşılaşabilir. Dolayısıyla hayat mükemmel değildir.

İPUCU: MÜKEMMEL İYİNİN DÜŞMANIDIR (VOLTAIRE YASASI)

Güzel bir sistem tasarladığınızda bile, bazen tüm köşe kasaları tam olarak istediğiniz gibi gitmez. Yukarıdaki mkdir örneğini ele alalım; mkdir farklı semantiklere sahip olacak şekilde yeniden tasarlanabilir, böylece onu önemsiz hale getirebilir (bunu nasıl yapabileceğinizi düşünün); Ancak, neden rahatsız? NFS tasarım felsefesi, önemli durumların çoğunu kapsar ve her şeyden önce sistem tasarımını arıza açısından temiz ve basit hale getirir. Bu nedenle, hayatın mükemmel olmadığını kabul etmek ve yine de sistemi kurmak, iyi mühendisliğin bir işaretidir. Görünüşe göre, bu bilgelik Voltaire'e atfedilir, çünkü "... akıllı bir İtalyan, en iyinin iyinin düşmanı olduğunu söyler" [V72] ve biz de buna **Voltaire Yasası(Voltaire's Law)** diyoruz.

49.8 Performansı Artırma: İstemci Tarafında Önbelleğe Alma

Dağıtılmış dosya sistemleri birkaç nedenden dolayı iyidir, ancak tüm okuma ve yazma isteklerini ağ üzerinden göndermek büyük bir performans sorununa yol açabilir: ağ genellikle, özellikle yerel bellek veya diskle karşılaştırıldığında o kadar hızlı değildir. Bu nedenle, başka bir sorun: Dağıtılmış bir dosya sisteminin performansını nasıl geliştirebiliriz?

Cevap, yukarıdaki alt başlıktaki büyük kalın sözcükleri okuyarak tahmin edebileceğiniz gibi, istemci tarafında **önbelleğe(cache)** almadır. NFS istemci tarafı dosya sistemi, sunucudan okuduğu dosya verilerini (ve meta verileri) istemci belleğinde önbelleğe alır. Bu nedenle, ilk erişim pahalı olsa da (yani, ağ iletişimi gerektirir), sonraki erişimlere müşteri belleğinden oldukça hızlı bir şekilde hizmet verilir.

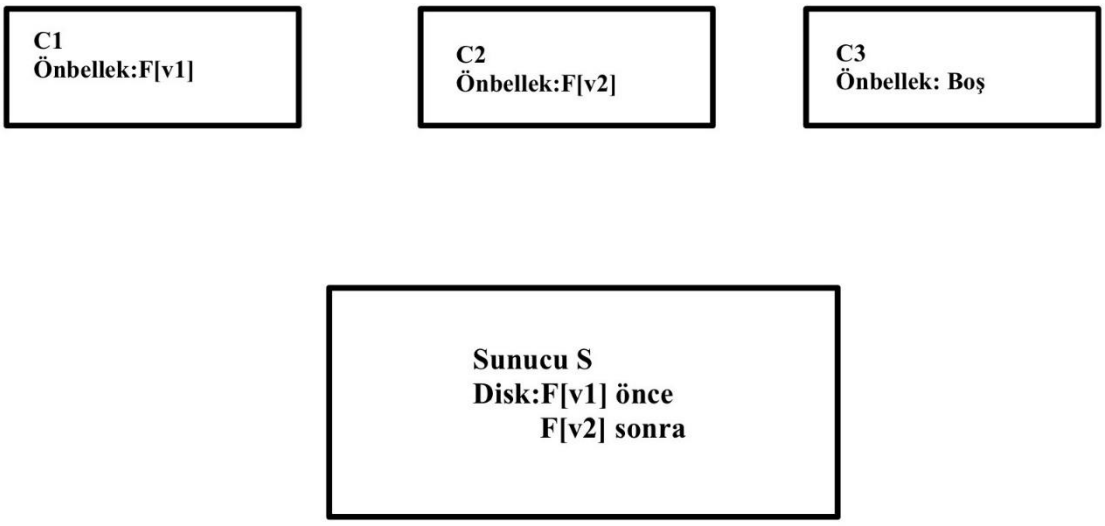
Önbellek ayrıca yazmalar için geçici bir arabellek görevi görür. Bir istemci uygulaması bir dosyaya ilk kez yazdığında, istemci, verileri sunucuya yazmadan önce istemci belleğindeki (dosya sunucusundan okuduğu verilerle aynı önbellekte) ara belleğe alır. Bu tür bir **yazma arabelleği(write buffering)** kullanışlıdır, çünkü uygulamanın write()

gecikmesini gerçek yazma performansından ayırır, yani uygulamanın write() çağrısı hemen başarılı olur (ve yalnızca verileri istemci tarafındaki dosya sisteminin önbelleğine koyar); ancak daha sonra veriler dosya sunucusuna yazılır.

Bu nedenle, NFS istemcileri verileri önbelleğe alır ve performans genellikle mükemmeldir ve işimiz bitti, değil mi? Ne yazık ki, tam olarak değil. Birden çok istemci önbelleği olan herhangi bir sisteme önbellek eklemek, **önbellek tutarlılığı sorunu(cache consistency problem.)** olarak adlandıracağımız büyük ve ilginç bir zorluğu beraberinde getirir.

49.9 Önbellek Tutarlılığı Sorunu

Önbellek tutarlılığı sorunu en iyi şekilde iki istemci ve tek bir sunucu ile gösterilmektedir. C1 istemcisinin bir F dosyasını okuduğunu ve dosyanın bir kopyasını yerel önbelleğinde tuttuğunu hayal edin. Şimdi farklı bir istemcinin, C2'nin F dosyasının üzerine yazdığını ve böylece içeriğini değiştirdiğini hayal edin; dosyanın yeni versiyonuna F diyelim



Şekil 49.7: Önbellek Tutarlılığı Sorunu

(sürüm 2) veya F[v2] ve eski sürüm F[v1] böylece ikisini ayrı tutabiliriz (ama tabii ki dosyanın adı aynı, sadece içeriği farklı). Son olarak, F dosyasına henüz erişmemiş olan üçüncü bir müşteri C3 vardır.

Yaklaşmakta olan sorunu muhtemelen görebilirsiniz (Şekil 49.7). Aslında iki alt problem var. Birinci alt problem, C2 istemcisinin yazmalarını sunucuya yaymadan önce bir süre

önbelleğinde arabelleğe alabilmesidir; bu durumda, F[v2] C2'nin belleğinde otururken, F'ye başka bir istemciden (C3 diyelim) herhangi bir erişim, dosyanın eski sürümünü (F[v1]) getirecektir. Bu nedenle, istemcide yazma işlemlerini arabelleğe alarak, diğer istemciler dosyanın eski sürümlerini alabilir ve bu istenmeyen bir durum olabilir; gerçekten de, C2 makinesinde oturum açtığınızı, F'yi güncellediğinizi ve ardından C3'te oturum açıp dosyayı okumaya çalıştığınızı, yalnızca eski kopyayı almak için hayal edin! Elbette bu sinir bozucu olabilir. Böylece, önbellek tutarlılık sorunu **güncelleme görünürlüğünün(update visibility;)** bu yönüne; Bir istemciden gelen güncellemeler diğer istemcilerde ne zaman görünür hale gelir?

Önbellek tutarlılığının ikinci alt problemi**, eski bir önbellektir(a stale cache;)**; bu durumda, C2 nihayet yazma işlemlerini dosya sunucusuna aktarmıştır ve bu nedenle sunucu en son sürüme (F[v2]) sahiptir. Ancak, C1'in önbelleğinde hala F[v1] vardır; C1 üzerinde çalışan bir program F dosyasını okursa, eski bir sürüm (F[v1]) alır ve en son kopyayı (F[v2]) almaz ki bu (genellikle) istenmeyen bir durumdur.

NFSv2 uygulamaları, bu önbellek tutarlılığı sorunlarını iki şekilde çözer. İlk olarak, güncelleme görünürlüğünü ele almak için, istemciler bazen kapatıldığında **hizalama(flush-on-close)** (diğer bir adıyla **açmaya yakın(close-to-open)**) tutarlılık anlambilimi olarak adlandırılan şeyi uygular; özellikle, bir dosya bir istemci uygulamasına yazıldığında ve daha sonra bu uygulama tarafından kapatıldığında, istemci sunucudaki tüm güncellemeleri (yani, önbellekteki kirli sayfalar) temizler. Kapatma sırasında hizalama tutarlılığıyla NFS, başka bir düğümden sonraki bir açmanın en son dosya sürümünü görmesini sağlar.

İkincisi, eski önbellek sorununu çözmek için, NFSv2 istemcileri önce bir dosyanın önbelleğe alınmış içeriğini kullanmadan önce değişip değişmediğini kontrol eder. Spesifik olarak, önbelleğe alınmış bir bloğu kullanmadan önce, istemci tarafı dosya sistemi, dosyanın özniteliklerini getirmesi için sunucuya bir GETATTR isteği gönderir. Nitelikler, daha da önemlisi, dosyanın sunucuda en son ne zaman değiştirildiğine ilişkin bilgileri içerir; değişiklik zamanı, dosyanın istemci önbelleğine getirildiği zamandan daha yeniyse, istemci dosyayı **geçersiz kılar(invalidates)**, böylece dosyayı istemci önbelleğinden kaldırır ve sonraki okumaların sunucuya gitmesini ve en son sürümü almasını sağlar dosyanın. Öte yandan, istemci dosyanın en son sürümüne sahip olduğunu görürse önbelleğe alınmış içeriği kullanmaya devam edecek ve böylece performans artacaktır.

Sun'daki orijinal ekip bu çözümü eski önbellek sorununa uyguladığında yeni bir sorunun farkına vardılar; NFS sunucusu aniden GETATTR istekleriyle doldu. İzlenecek iyi bir mühendislik ilkesi, ortak durum için tasarım yapmak ve onun iyi çalışmasını sağlamaktır;

burada, bir dosyaya yalnızca tek bir istemciden (belki tekrar tekrar) erişilmesi **yaygın bir durum(common case)** olsa da, istemcinin dosyayı başka hiç kimsenin değiştirmediğinden emin olmak için sunucuya her zaman GETATTR istekleri göndermesi gerekiyordu. Böylece bir istemci sunucuyu bombalar ve çoğu zaman kimsenin değiştirmediği halde sürekli "bu dosyayı değiştiren oldu mu?" diye sorar.

Bu durumu (bir şekilde) düzeltmek için her müşteriye bir **öznitelik önbelleği(attribute cache)** eklendi. Bir istemci, bir dosyayı erişmeden önce yine de doğrular, ancak çoğu zaman öznitelikleri getirmek için yalnızca öznitelik önbelleğine bakar. Belirli bir dosyanın öznitelikleri, dosyaya ilk erişildiğinde önbelleğe yerleştirildi ve ardından belirli bir süre sonra (3 saniye diyelim) zaman aşımına uğradı. Böylece, bu üç saniye boyunca, tüm dosya erişimleri, önbelleğe alınan dosyayı kullanmanın uygun olduğunu belirleyecek ve böylece sunucuyla hiçbir ağ iletişimi olmadan bunu yapacaktır.

49.10 NFS Önbellek Tutarlılığını Değerlendirme

NFS önbellek tutarlılığı hakkında son birkaç söz. "Anlamlı" olması için kapatıldığında aynı hizada davranışı eklendi, ancak belirli bir performans sorunu ortaya çıkardı. Spesifik olarak, bir istemcide geçici veya kısa ömürlü bir dosya oluşturulmuş ve kısa süre sonra silinmişse, yine de sunucuya zorlanır. Daha ideal bir uygulama, bu tür kısa ömürlü dosyaları silinene kadar bellekte tutabilir ve böylece sunucu etkileşimini tamamen ortadan kaldırarak belki de performansı artırabilir.

Daha da önemlisi, NFS'ye bir öznitelik önbelleğinin eklenmesi, bir dosyanın tam olarak hangi sürümünün alındığını anlamayı veya bu konuda akıl yürütmeyi çok zorlaştırdı. Bazen en son sürümü alırsınız; bazen, öznitelik önbelleğiniz henüz zaman aşımına uğramadığı ve bu nedenle istemci, istemci belleğindekileri size vermekten mutlu olduğu için eski bir sürümü alırsınız. Bu çoğu zaman iyi olsa da, bazen garip davranışlara yol açıyordu (ve hala da öyle!).

Ve böylece, NFS istemcisini önbelleğe almanın garipliğini tanımlamış olduk. Bir uygulamanın ayrıntılarının, tersi yerine kullanıcı tarafından gözlemlenebilir anlambilimi tanımlamaya hizmet ettiği ilginç bir örnek olarak hizmet eder.

49.11 Sunucu Tarafı Yazma Arabelleğine Etkileri

Şu ana kadar istemcileri önbelleğe almaya odaklandık ve ilginç sorunların çoğu da burada ortaya çıkıyor. Bununla birlikte, NFS sunucuları, çok fazla belleğe sahip iyi donanımlı makineler olma eğilimindedir ve bu nedenle önbelleğe alma sorunları da vardır. Veriler (ve meta veriler) diskten okunduğunda, NFS sunucuları bunu bellekte tutacak ve söz konusu verilerin (ve meta veriler) sonraki okumaları diske gitmeyecek, bu da performansta potansiyel (küçük) bir artış.

Daha ilgi çekici olan, yazma arabelleğe alma durumudur. NFS sunucuları, yazma kararlı depolamaya (örneğin, diske veya başka bir kalıcı aygıta) zorlanana kadar bir YAZMA protokolü isteğinde kesinlikle başarı döndürmeyebilir. Verilerin bir kopyasını sunucu belleğine yerleştirebilseler de, bir YAZMA protokolü isteğinde istemciye başarı döndürmek yanlış davranışa neden olabilir; nedenini anlayabilir misin?

Cevap, istemcilerin sunucu arızasını nasıl ele aldığına ilişkin varsayımlarımızda yatmaktadır. Bir müşteri tarafından verilen aşağıdaki yazma sırasını hayal edin:

```
write(fd, a_buffer, size); // ilk bloğu a ile doldur
write(fd, b_buffer, size); // ikinci bloğu b ile doldur
write(fd, c_buffer, size); // üçüncü bloğu c ile doldur
```

Bu yazma işlemleri, bir dosyanın üç bloğunun üzerine bir blok a'lar, sonra b'ler ve sonra c'ler yazar. Böylece, dosya başlangıçta şöyle görünüyorsa:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Bu yazmalardan sonra nihai sonucun x'ler, y'ler ve z'lerin üzerine sırasıyla a'lar, b'ler ve c'ler ile yazılmasını bekleyebiliriz.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccc
```

Şimdi bu üç istemci yazma işleminin sunucuya üç ayrı WRITE protokol mesajı olarak verildiğini örnek olarak kabul edelim. İlk WRITE mesajının sunucu tarafından alındığını ve diske gönderildiğini ve istemcinin başarı hakkında bilgilendirildiğini varsayalım. Şimdi, ikinci yazmanın bellekte arabelleğe alındığını ve sunucunun ayrıca bunu diske zorlamadan önce istemciye başarısını bildirdiğini varsayalım; ne yazık ki, sunucu diske yazmadan önce çöküyor. Sunucu hızlı bir şekilde yeniden başlatılır ve başarılı olan üçüncü yazma isteğini alır.

Böylece müşteri için tüm istekler başarılı oldu, ancak dosya içeriğinin şöyle görünmesine şaşırdık:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--
ccccccccccccccccccccccccccccccccccccccccc
Eyvah! Sunucu,

KENAR: İNOVASYON İNOVASYONU DOĞURUR

Birçok öncü teknolojide olduğu gibi, NFS'yi dünyaya getirmek, başarısını sağlamak için başka temel yenilikleri de gerektiriyordu. Muhtemelen en kalıcısı, farklı dosya sistemlerinin işletim sistemine [K86] kolayca takılmasına izin vermek için Sun tarafından sunulan **Sanal Dosya Sistemi (VFS) / Sanal Düğüm (vnode)(Virtual File System (VFS) / Virtual Node (vnode))** arabirimidir. VFS katmanı, bağlama ve ayırma, dosya sistemi çapında istatistikler alma ve tüm kirli (henüz yazılmamış) yazma işlemlerini diske zorlama gibi tüm bir dosya sistemine yapılan işlemleri içerir. Vnode katmanı, bir dosya üzerinde gerçekleştirilebilecek açma, kapama, okuma, yazma vb. tüm işlemlerden oluşur. Yeni bir dosya sistemi oluşturmak için bu "yöntemleri" tanımlamanız yeterlidir; çerçeve daha sonra sistem çağrılarını belirli dosya sistemi uygulamasına bağlayarak, tüm dosya sistemlerinde ortak olan genel işlevleri (örneğin önbelleğe alma) merkezi bir şekilde gerçekleştirerek ve böylece birden çok dosya sistemi uygulamasının aynı anda aynı sistem. Bazı ayrıntılar değişmiş olsa da, Linux, BSD varyantları, macOS ve hatta Windows (Yüklenebilir Dosya Sistemi biçiminde) dahil olmak üzere birçok modern sistemde bir çeşit VFS/vnode katmanı bulunur. NFS dünyayla daha az alakalı hale gelse bile, altındaki gerekli temellerden bazıları yaşamaya devam edecektir.

istemciye ikinci yazmanın diske kaydedilmeden önce başarılı olduğunu söylediği için, dosyada uygulamaya bağlı olarak felaketle sonuçlanabilecek eski bir parça kalır.

Bu sorunu önlemek için, NFS sunucuları, istemciye başarıyı bildirmeden önce her yazmayı kararlı (kalıcı) depolamaya işlemelidir; bunu yapmak, istemcinin bir yazma sırasında sunucu hatasını algılamasını ve böylece sonunda başarılı olana kadar yeniden denemesini sağlar. Bunu yapmak, yukarıdaki örnekte olduğu gibi iç içe geçmiş dosya içerikleriyle asla bitmeyeceğimizi garanti eder.

Bu gereksinimin NFS sunucu uygulamasında yol açtığı sorun, yazma performansının büyük bir dikkat göstermeden en büyük performans darboğazı olabilmesidir. Gerçekten de bazı şirketler (ör. Network Appliance) , yazma işlemlerini hızlı bir şekilde gerçekleştirebilen bir NFS sunucusu oluşturmak gibi basit bir amaçla ortaya çıktı; kullandıkları bir numara, yazma işlemlerini önce pil destekli bir belleğe yerleştirmek, böylece verileri kaybetme korkusu olmadan ve hemen diske yazma maliyeti olmadan YAZMA isteklerine hızlı bir şekilde yanıt vermeyi sağlamaktır; ikinci numara, en sonunda ihtiyaç duyulduğunda diske hızlı bir şekilde yazmak için özel olarak tasarlanmış bir dosya sistemi tasarımı kullanmaktır [HLM94, RO91].

49.12 Özet

NFS dağıtılmış dosya sisteminin tanıtımını gördük. NFS, sunucu arızası karşısında basit ve hızlı kurtarma fikri etrafında toplanmıştır ve bu amaca dikkatli protokol tasarımıyla ulaşır.Oper-

KENARA: ANAHTAR NFS ŞARTLARI

• NFS'de hızlı ve basit kilitlenme kurtarmanın ana hedefini gerçekleştirmenin anahtarı, durum bilgisi olmayan bir protokol tasarımıdır. Bir kilitlenmeden sonra, sunucu hızlı bir şekilde yeniden başlatılabilir ve istekleri yeniden sunmaya başlayabilir; istemciler istekleri başarılı olana kadar yeniden dener.

• İstekleri **geçersiz(idempotent)** kılma, NFS protokolünün merkezi bir özelliğidir. Bir işlemi birden çok kez gerçekleştirmenin etkisi, bir kez gerçekleştirmeye eşdeğer olduğunda, idempotenttir. NFS'de bağımsız olma özelliği, istemcinin endişelenmeden **yeniden denemesini(retry)** sağlar ve istemcinin kayıp ileti yeniden iletimini ve istemcinin sunucu çökmelerini nasıl ele aldığını birleştirir.
• Performans endişeleri, istemci tarafında **önbelleğe(cache)** alma ve **yazma arabelleğe(write buffering)** alma ihtiyacını belirler, ancak bir **önbellek tutarlılığı sorunu(cache consistency problem)** ortaya çıkarır.
• NFS uygulamaları, tutarlılığı birden çok yöntemle önbelleğe almak için bir mühendislik çözümü sağlar: kapatıldığında **temizle(flush-on-close) (açmaya yakın(close-to-open))** yaklaşımı, bir dosya kapatıldığında içeriğinin sunucuya zorlanmasını sağlayarak diğer istemcilerin güncellemeler. Öznitelik önbelleği, bir dosyanın değişip değişmediğini (GETATTR istekleri aracılığıyla) sunucuyla kontrol etme sıklığını azaltır.
• NFS sunucuları, başarıyı döndürmeden önce yazma işlemlerini kalıcı ortama işlemelidir; aksi takdirde veri kaybı meydana gelebilir
• Sun, işletim sistemine NFS entegrasyonunu desteklemek için **VFS/Vnode** arayüzünü sunarak birden fazla dosya sistemi uygulamasının aynı işletim sisteminde bir arada var olmasını sağladı.

asyonların yetersizliği esastır; Bir istemci başarısız bir işlemi güvenli bir şekilde yeniden oynatabileceğinden, sunucu isteği yürütmüş olsun ya da olmasın bunu yapmakta bir sakınca yoktur.

Ayrıca önbelleğe almanın çok istemcili, tek sunuculu bir sisteme dahil edilmesinin işleri nasıl karmaşık hale getirebileceğini de gördük. Özellikle, sistemin makul davranabilmesi için önbellek tutarlılığı sorununu çözmesi gerekir; ancak, NFS bunu biraz geçici bir şekilde yapar ve bu da bazen gözlemlenebilir şekilde garip davranışlara neden olabilir. Son olarak, sunucu önbelleğe almanın ne kadar zor olabileceğini gördük: sunucuya yazma işlemleri, başarı döndürmeden önce kararlı depolamaya zorlanmalıdır (aksi takdirde veriler kaybolabilir).

Kesinlikle alakalı olan diğer konulardan, özellikle güvenlikten bahsetmedik. İlk NFS uygulamalarında güvenlik oldukça gevşekti; bir istemcideki herhangi bir kullanıcının diğer kullanıcılar gibi görünmesi ve böylece neredeyse tüm dosyalara erişmesi oldukça kolaydı. Daha ciddi kimlik doğrulama hizmetleriyle (örneğin, Kerberos [NT94]) sonraki entegrasyon, bu bariz eksiklikleri gidermiştir.

Referanslar

[AKW88] "AWK Programlama Dili", Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. Pearson, 1988 (1. baskı). awk hakkında özlü, harika bir kitap. Bir zamanlar Peter Weinberger ile tanışma şerefine erişmiştik; kendini tanıttığında, "Ben Peter Weinberger, bilirsin, awk'deki 'W'?" Devasa awk hayranları olarak bu, tadına varılması gereken bir andı. Birimiz (Remzi) o zaman "Awk'u seviyorum! Her şeyi harika bir şekilde netleştiren kitabı özellikle seviyorum. Weinberger (hayal kırıklığına uğramış bir şekilde) yanıtladı, "Ah, kitabı Kernighan yazdı."

[C00] Brent Callaghan'ın "NFS Illustrated"ı. Addison-Wesley Professional Computing Series, 2000. Harika bir NFS referansı; protokolün kendisine göre inanılmaz derecede kapsamlı ve ayrıntılı.

[ES03] "Sistem Analizi için Yeni NFS İzleme Araçları ve Teknikleri", Daniel Ellard ve Margo Seltzer. LISA '03, San Diego, Kaliforniya. Pasif izleme yoluyla yapılan karmaşık, dikkatli bir NFS analizi. Yazarlar, yalnızca ağ trafiğini izleyerek, çok büyük miktarda dosya sistemi anlayışının nasıl elde edileceğini gösteriyor.

[HLM94] Dave Hitz, James Lau, Michael Malcolm tarafından yazılan "NFS Dosya Sunucusu Cihazı için Dosya Sistemi Tasarımı". USENIX Kış 1994. San Francisco, California, 1994. Hitz ve ark. günlük yapılı dosya sistemleri üzerindeki önceki çalışmalardan büyük ölçüde etkilenmiştir.

[K86] "Vnodes: Sun UNIX'te Çoklu Dosya Sistemi Tipleri İçin Bir Mimari", Steve R. Kleiman. USENIX Yaz '86, Atlanta, Georgia. Bu belge, bir işletim sisteminde esnek bir dosya sistemi mimarisinin nasıl oluşturulacağını ve birden fazla farklı dosya sistemi

uygulamasının bir arada var olmasını nasıl sağlayacağını gösterir. Artık hemen hemen her modern işletim sisteminde bir biçimde kullanılmaktadır.

[NT94] “Kerberos: Bilgisayar Ağları için Kimlik Doğrulama Hizmeti”, yazan B. Clifford Neuman, Theodore Ts'o. IEEE Communications, 32(9):33-38, Eylül 1994. Kerberos, eski ve çok etkili bir kimlik doğrulama hizmetidir. Muhtemelen bir ara bununla ilgili bir kitap bölümü yazmalıyız...

[O91] “Dağıtılmış Durumun Rolü”, John K. Ousterhout. 1991. Bu sitede mevcuttur: ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps. Dağıtılmış duruma ilişkin nadiren başvurulan bir tartışma; sorunlar ve zorluklar hakkında daha geniş bir perspektif.

[P+94] "NFS Sürüm 3: Tasarım ve Uygulama", Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz. USENIX Yaz 1994, sayfalar 137-152. NFS sürüm 3'ün altında yatan küçük değişiklikler.

[P+00] Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow tarafından yazılan "NFS sürüm 4 protokolü". 2. Uluslararası Sistem Yönetimi ve Ağ Kurma Konferansı (SANE 2000). Hiç şüphesiz NFS üzerine şimdiye kadar yazılmış en edebi makale.

[RO91] "Günlük Yapılı Dosya Sisteminin Tasarımı ve Uygulanması", Mendel Rosenblum, John Ousterhout. İşletim Sistemleri İlkeleri Sempozyumu (SOSP), 1991. Yeniden LFS. Hayır, asla yeterince LFS elde edemezsiniz.

[S86] "The Sun Network File System: Tasarım, Uygulama ve Deneyim", Russel Sandberg. USENIX Yaz 1986. Orijinal NFS makalesi; Biraz zorlu bir okuma olsa da, bu harika fikirlerin kaynağını görmeye değer.

[Sun89] “NFS: Ağ Dosya Sistemi Protokol Spesifikasyonu”, Sun Microsystems, Inc. Yorum Talebi: 1094, Mart 1989. Mevcut: http://www.ietf.org/rfc/rfc1094.txt.
Korkunç özellik; gerekirse okuyun, yani okumanız için para alıyorsunuz. Umarım, çok ödedi. Nakit para!

[V72] Francois-Marie Arouet namı diğer Voltaire'den “La Begueule”. 1772'de yayınlandı. Voltaire bir dizi zekice şey söyledi, bu sadece bir örnek. Örneğin Voltaire, “Ülkenizde iki din varsa, ikisi birbirinin boğazını keser; ama otuz dinin olursa, onlar selâmet içinde yaşarlar.” Buna ne dersiniz Demokratlar ve Cumhuriyetçiler?

Ödev (Ölçme)

Bu ödevde, gerçek izleri kullanarak biraz NFS iz analizi yapacaksınız. Bu izlerin kaynağı Ellard ve Seltzer'in çabasıdır [ES03]. Başlamadan önce ilgili BENİOKU'yu okuduğunuzdan ve ilgili tarball'ı OSTEP ödev sayfasından (her zamanki gibi) indirdiğinizden emin olun.

Sorular

1. İz analiziniz için ilk soru: İlk sütunda bulunan zaman damgalarını kullanarak izlerin alındığı süreyi belirleyin. Süre ne kadar? Hangi gün/hafta/ay/yıldı? (bu, dosya adında verilen ipucu ile eşleşiyor mu?) İpucu: Dosyanın ilk ve son satırlarını çıkarmak için head -1 ve tail -1 araçlarını kullanın ve hesaplamayı yapın.

strftime() gawk uzantısıdır, POSIX standardında belirtilmemiştir. Dosya adı anon-deasna-021016-1300.txt, aynı tarih.

```
alipekin@ubuntu2204:~/MasaüstÜ/dila$ touch anon-deasna-151222-1204.txt
alipekin@ubuntu2204:~/MasaüstÜ/dila$ awk -f q1.awk anon-deasna-151222-1204.txt
Period: 0,00 minutes
Start time: 01 01 1970
alipekin@ubuntu2204:~/MasaüstÜ/dila$
```

2. Şimdi işlem sayımı yapalım. İzlemede her işlem türünden kaç tane gerçekleşir? Bunları frekansa göre sıralayın; en sık hangi operasyon yapılır? NFS itibarını hak ediyor mu?

```
alipekin@ubuntu2204:~/MasaüstÜ/dila$  awk '{ if ($5 == "C3" && ($8 == "read" ||
 $8 == "write")) print $2, $8, $10, $12, $14 }' anon-deasna-151222-1204.txt | s
ort -u | wc -l
469427
0
469427: komut bulunamadı
```

```
alipekin@ubuntu2204:~/Masaüstü/dila$ awk '{ if ($5 == "C3" && ($8 == "read" ||
$8 == "write")) print $0 }' anon-deasna-151222-1204.txt | wc -l
838995
0
838995: komut bulunamadı
```

1 - 469427/838995 = %44,05 okuma ve yazma komutları yeniden deneme komutlarıdır.

3. Şimdi bazı özel işlemlere daha detaylı bakalım. Örneğin, GETATTR isteği, isteğin hangi kullanıcı kimliği için gerçekleştirildiği, dosyanın boyutu vb. dahil olmak üzere dosyalar hakkında birçok bilgi döndürür. İzleme içinde erişilen dosya boyutlarının dağılımını yapın; ortalama dosya boyutu nedir? Ayrıca, izlemedeki dosyalara kaç farklı kullanıcı erişir? Trafiğe birkaç kullanıcı mı hakim, yoksa daha mı dağınık? GETATTR yanıtlarında başka hangi ilginç bilgiler bulunur?

```
alipekin@ubuntu2204:~/Masaüstü/dila$ awk -f q3.awk anon-deasna-151222-1204.txt
| sort -nk4 -r
awk: q3.awk: line 27: function strtonum never defined
```

NFS Sürüm 3 Protokol Spesifikasyonunun 2. bölümüne bakın.

4. Belirli bir dosyaya yönelik isteklere de bakabilir ve dosyalara nasıl erişildiğini belirleyebilirsiniz. Örneğin, belirli bir dosya sırayla mı okunuyor veya yazılıyor? Yoksa rastgele mi? Cevabı hesaplamak için OKUMA ve YAZMA isteklerinin/yanıtlarının ayrıntılarına bakın.

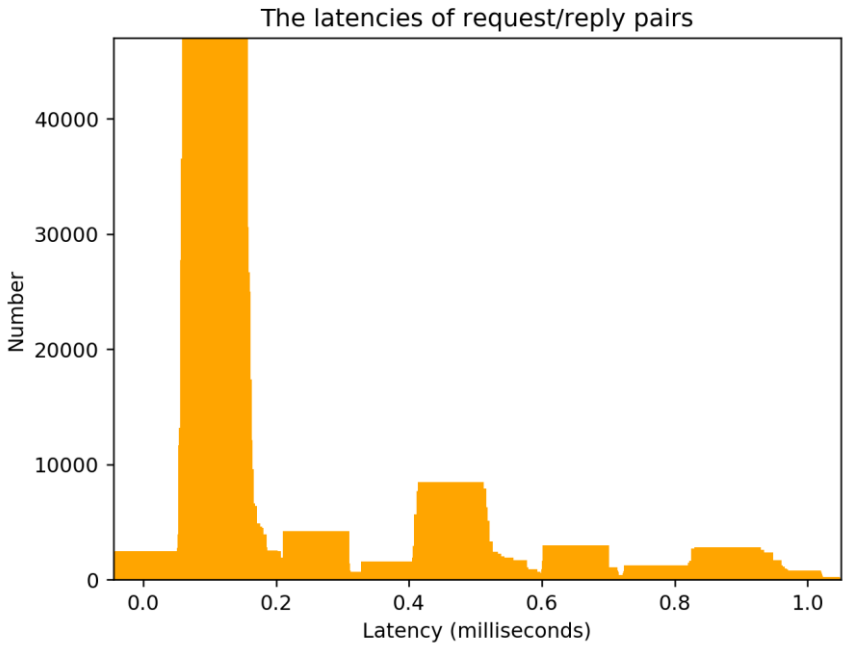
Yazma sıralıdır, ancak okuma sıralı değildir.

5. Trafik birçok makineden gelir ve bir sunucuya gider (bu izlemede). İzlemede kaç farklı istemci olduğunu ve her birine kaç istek/yanıt gittiğini gösteren bir trafik matrisi hesaplayın. Birkaç makine hakim mi yoksa daha dengeli mi?

```
alipekin@ubuntu2204:~/Masaüstü/dila$ awk -f q5.awk anon-deasna-151222-1204.txt
| sort -nk4 -r
0 clients
```

Yalnızca beş müşterinin on binden fazla isteği ve yanıtı vardır.

6. Zamanlama bilgileri ve istek/yanıt başına benzersiz kimlik, belirli bir istek için gecikmeyi hesaplamanıza izin vermelidir. Tüm istek/yanıt çiftlerinin gecikmelerini hesaplayın ve bunları bir dağılım olarak çizin. ortalama nedir? Maksimum? Asgari mi?



7. İstek veya yanıtı kaybolabileceği veya düşebileceği için bazen istekler yeniden denenir. İz örneğinde böyle bir yeniden denemeye dair herhangi bir kanıt bulabilir misiniz?

İkinci sorunun cevabını inceleyiniz.

8. Daha fazla analizle cevaplayabileceğiniz birçok başka soru var. Sizce hangi sorular önemli? Onları bize önerin, belki onları buraya ekleyeceğiz!

Attr önbellek süresini bulun.