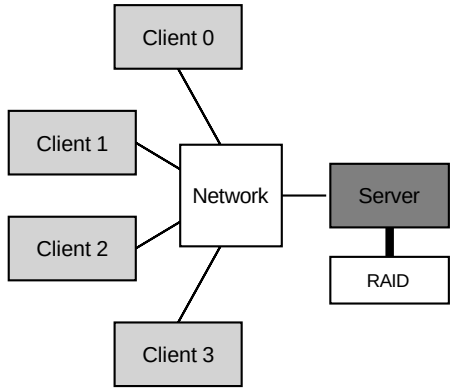
49

Sun 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 yoluyla 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 kullanmasına izin vermiyoruz?) Öncelikle bu kurulum, verilerin istemciler arasında kolayca **paylaşım (sharing)** yapılmasına olanak tanır. 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. İkinci bir fayda ise, **merkezi yönetim(centralized administration);** örneğin, dosyaların yedeklenmesi çok sayıda istemci yerine birkaç sunucu makinesinden yapılabilir. Diğer bir avantaj ise **güvenlik(security)** olabilir; tüm sunucuların kilitli bir makine dairesinde olması, belirli türden sorunların ortaya çıkmasını önler.

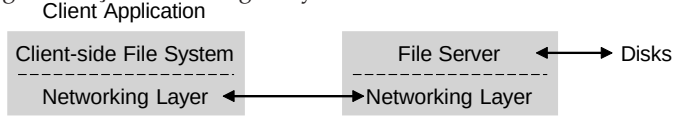
CAN ALICI NOKTA: DAĞITILMIŞ DOSYA SİSTEMİ NASIL OLUŞTURULUR?
Dağıtılmış bir dosya sistemi nasıl oluşturulur? Düşünülmesi gereken temel hususlar nelerdir? Yanlış yapmakta kolay olan nedir? Mevcut sistemlerden ne öğrenebiliriz?

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 (clientside file system)** aracılığıyla dosyalara ve dizinlere erişen istemci uygulamaları vardır. Bir istemci uygulaması, istemci taraflı dosya sistemine, sunucuda depolanan dosyalara erişmek için sistem çağrıları gönderir (open(), read(), write(), close(), mkdir(), vb.) 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 taraflı dosya sisteminin rolü, bu sistem çağrılarına hizmet vermek için gereken eylemleri yürütmektir. Örneğin, istemci bir read() isteği gönderirse, istemci taraflı 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 verilerle bir mesaj geri 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(cache)** alınabileceğini 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 TARAFTAN: 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 milyonlarca kod satırından oluşur; bu nedenle, **hataları(bugs)** 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, dağıtılmış 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

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 birlikte çalışabilirliği korurken bir NFS pazarında rekabet edebilir. Bu işe 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 Noktası: Basit ve Hızlı Sunucu Çökmesinden Kurtarma

Bu bölümde, 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 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: Durum Bilgisizliği

Bu basit hedef, NFSv2'de **durum bilgisi(stateless)** olmayan 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 işaretçisi 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 bilgisizliğine yaklaşım daha anlamlı olacaktır.

Durum bilgili(stateful) (durumsuz değil) bir protokol örneği için open() sistem çağrısını düşünün. Bir yol adı verildiğinde, open() bir tam sayı olan integer tipinde dosya tanıtıcısı döndürür. Bu tanımlayıcı, bu uygulama kodunda olduğu gibi, çeşitli dosya bloklarına erişmek için sonraki okuma() veya yazma() isteklerinde kullanılır (boşluk nedeniyle sistem çağrılarının uygun hata denetiminin yapılmadığını unutmayın)

```
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 taraflı 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 sistem çağrısını çağırmak için read() tanımlayıcısını kullanır; istemci taraflı 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 ve sunucu arasındaki **paylaşılan durumun(shared state)** bir parçasıdır. (Ousterhout bu **dağıtılmış durumu(distributed state)** [O91] olarak adlandırır.) Paylaşılan durum, yukarıda ima ettiğimiz gibi, kilitlenme kurtarmayı zorlaştırır. İlk okuma tamamlandıktan sonra, fakat 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 (yani bellekte) geçici bir bilgiydi ve bu nedenle sunucu çöktüğünde kayboldu. Bu durumla başa çıkmak için, istemci ve sunucunun, sunucuya bilmesi gerekenleri söyleyebilmesi için istemcinin belleğinde yeterli bilgiyi sakladığından emin olacağı bir tür **kurtarma protokolüne(recovery protocol)** dahil olması gerekir. (Bu durumda, fd dosya tanıtıcısı 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. Aşırı çarpışma kurtarması 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:

CAN ALICI 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, özel bir soru olarak, protokolü hem durumsuz olacak hem de POSIX dosya sistemi API'sini destekleyecek şekilde nasıl tanımlarız?

NFS protokolünün tasarımını anlamanın bir anahtarı, dosya tanıtıcısını anlamaktır. **Dosya tanıtıcıları(file handles**), 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: Birim tanımlayıcısı, bir inode(dosya indeksi) 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	file handle returns: attributes
NFSPROC_SETATTR	file handle, attributes returns: –
NFSPROC_LOOKUP	directory file handle, name of file/dir to look up returns: file handle
NFSPROC_READ	file handle, offset, count data, attributes
NFSPROC_WRITE	file handle, offset, count, data attributes
NFSPROC_CREATE	directory file handle, name of file, attributes –
NFSPROC_REMOVE	directory file handle, name of file to be removed –
NFSPROC_MKDIR	directory file handle, name of directory, attributes file handle
NFSPROC_RMDIR	directory file handle, name of directory to be removed –
NFSPROC_READDIR	directory handle, count of bytes to read, cookie returns: directory entries, cookie (to get more entries)

Şekil 49.4: NFS Protokolü: Örnekler

Protokolün önemli bileşenlerini kısaca vurguluyoruz. İlk olarak, **ARAMA(LOOKUP)** 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 aracılığıyla elde edilir; biz bunu yapmayız. Kısa olması için bağlama protokolünü burada tartışmıyoruz.) 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 vb. gibi alanlar dahil olmak üzere bir dosyada stat() işlevini çağırırsanız alacağınız bilgilerin aynısı olan verilerdir, yani aynı türden bilgilerdir.

Bir dosya tanıtıcısı kullanılabilir olduğunda, istemci sırasıyla dosyayı okumak veya yazmak için bir dosya üzerinde **READ(OKUMA**) ve **WRITE(YAZMA)** protokol mesajları verebilir. READ protokol mesajı, protokolün dosyanın dosya tanıtıcısı boyunca dosya içindeki ofset(Bilgisayar mühendisliğinde ve çevirme dili gibi alt seviye programlamada ofset, belirli bellek adresleme almak için taban adrese eklenen adres konumlarının sayısını gösterir.) ve okunacak bayt sayısı ile birlikte geçmesini gerektirir. Sunucu daha sonra okumayı yayınlayabilecektir (sonuçta tanıtıcı, sunucuya hangi birimin ve hangi inode'dan okunacağını söyler ve ofset ve sayı ona dosyanın hangi baytlarını okuyacağını söyler) ve verileri istemciye geri döndürür (veya bir başarısızlık durumunda hatayı). 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 taraflı dosya sistemi ve dosya sunucusu genelinde nasıl bir dosya sistemine dönüştürüldüğüne dair bir fikir ediniyorsunuzdur. İstemci taraflı 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(state)** 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 ofset (uzaklık) ile verilir. İkinci olarak, sunucu etkileşimlerinin nerede gerçekleştiğine dikkat edebilirsiniz. Dosya ilk kez açıldığında, istemci taraflı dosya sistemi bir **ARAMA(LOOKUP)** istek mesajı gönderir. Aslında, uzun bir yol adının geçilmesi gerekiyorsa (ör. /home/remzi/foo.txt), istemci üç **ARAMA(LOOKUP)** gönderir: biri / dizininde başlangıca(home) bakmak için, biri başlangıçta(home) remzi'ye bakmak için ve son olarak remzi'de foo.txt'ye bakmak için. Üçüncüsü, her sunucu isteğinin, isteği eksiksiz olarak tamamlamak için gereken tüm bilgilere nasıl sahip olduğuna dikkat edebilirsiniz. Bu tasarım noktası, şimdi daha ayrıntılı olarak tartışacağımız gibi, sunucu arızasından incelikle kurtulabilmek için kritik öneme sahiptir; sunucunun isteğe yanıt verebilmesi için duruma ihtiyaç duymamasını sağlar.

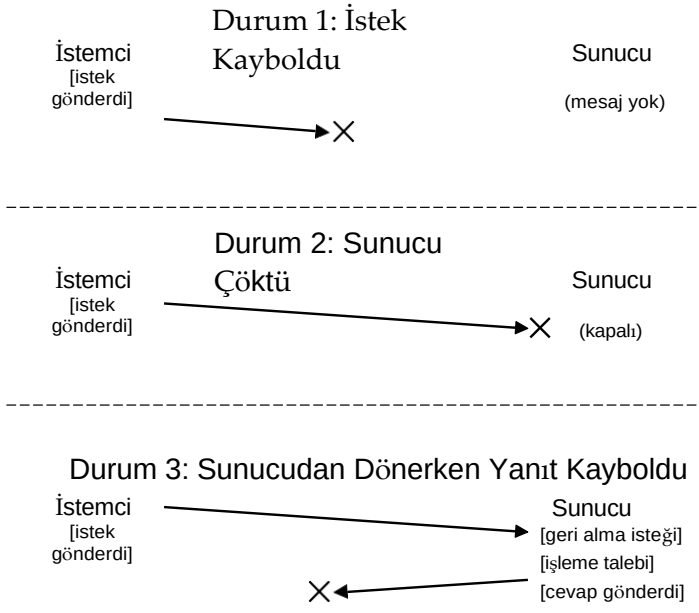
istemci	sunucu
fd = open("/foo", ...); Send LOOKUP (rootdir FH, "foo") (Arama gönder)	Ara isteğini al kök dizinde "foo "yu ara foo'nun FH + özniteliklerini döndür
ARA yanıtını al dosya tanımlayıcısını açık dosya tablosunda ayır foo'nun FH'sini tabloda sakla geçerli dosya konumunu sakla (0) dosya tanımlayıcısını uygulamaya döndür	
read(fd, buffer, MAX); fd ile açık dosya tablosunda dizin NFS dosya tanıtıcısı (FH) alır geçerli dosya konumunu ofset olarak kullan Send READ (FH, offset=0, count=MAX) (Yazma Gönder)	READ isteği alma birim/inode numarasını almak için FH kullanma diskten (veya önbellekten) inode okuma blok konumunu hesaplama (offset kullanarak) diskten (veya önbellekten) veri okuma veriyi istemciye döndürme
READ yanıtını al dosya konumunu güncelle (+bayt oku) geçerli dosya konumunu ayarla = MAX uygulamaya veri/hata kodu döndür	
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); Sadece yerel yapıları temizlemeniz gerekiyor Açık dosya tablosunda "fd" serbest tanımlayıcısı (Sunucuyla konuşmaya gerek yok)	

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

TIP: IDEMPOTENCY IS POWERFUL
Idempotency, güvenilir sistemler oluştururken faydalı bir özelliktir. Bir işlem birden fazla kez yapılabiliyorsa, işlemin başarısız olmasıyla başa çıkmak çok daha kolaydır; sadece yeniden deneyebilirsiniz. Eğer bir işlem idempotent değilse, hayat daha zor hale gelir.

49.7 Idempotent İşlemlerle Sunucu Arızasını Ele Alma

Bir istemci sunucuya bir mesaj gönderdiğinde, bazen bir yanıt alamaz. Bu yanıt alamama durumunun birçok olası nedeni vardır.Bazı durumlarda, mesaj ağ tarafından düşürülebilir; ağlar mesajları kaybeder ve böylece ya talep ya da cevap kaybolabilir ve Böylece istemci hiçbir zaman yanıt alamayacaktır. Sunucunun kapalı olması ve bu nedenle şu anda mesajlara yanıt vermemesi de mümkündür. Bir süre sonra sunucu yeniden başlatılacak ve tekrar çalışmaya başlayacaktır, ancak bu arada tüm istekler kaybolmuştur. Tüm bu durumlarda, istemciler bir soruyla baş başa kalırlar: Sunucu zamanında yanıt vermiyorsa ne yapmalıdırlar? NFSv2'de, bir istemci tüm bu hataları tek, tekdüze ve zarif bir şekilde ele alır: sadece isteği yeniden dener. Özellikle, isteği gönderdikten sonra, istemci belirli bir süre sonra kapanması için bir zamanlayıcı ayarlar. Zamanlayıcı kapanmadan önce bir yanıt alınırsa, zamanlayıcı iptal edilir ve her şey yolunda gider. Bununla birlikte, herhangi bir yanıt alınmadan önce zamanlayıcı kapanırsa, istemci isteğin işlenmediğini varsayar ve yeniden gönderir. Sunucu yanıt verirse, her şey yolundadır ve istemci sorunu düzgün bir şekilde halletmiştir. İstemcinin (başarısızlığa neyin neden olduğuna bakılmaksızın) isteği basitçe yeniden deneme yeteneği, çoğu NFS isteğinin önemli bir özelliğinden kaynaklanmaktadır: **denk güçlü olması (idempotent)** Bir işlemin etkisi şu durumlarda idempotent olarak adlandırılır: İşlemi birden çok kez gerçekleştirmenin etkisi, işlemi tek bir kez gerçekleştirmenin etkisine eşdeğerdir. Örneğin, bir değeri bir bellek konumuna üç kez depolarsanız, bunu bir kez yapmakla aynıdır; bu nedenle "değeri belleğe depola" bir idempotent işlemdir. Bununla birlikte, bir sayacı üç kez artırırsanız, bu, yalnızca bir kez yapmaktan farklı bir miktarda sonuçlanır; bu nedenle, "sayacı artır" idempotent değildir. Daha genel olarak, sadece veri okuyan herhangi bir işlem açıkça idempotenttir; veri güncelleyen bir işlemin bu özelliğe sahip olup olmadığını belirlemek için daha dikkatli düşünülmesi gerekir. NFS'de çökme kurtarma tasarımının kalbi, en yaygın işlemlerin idempotentliğidir. LOOKUP ve READ istekleri önemsiz derecede idempotenttir, çünkü yalnızca dosya sunucusundan bilgi okur ve güncelleme yapmazlar. Daha da ilginci, WRITE istekleri de idempotenttir. Örneğin, bir WRITE başarısız olursa, istemci basitçe tekrar deneyebilir. WRITE mesajı veri, sayı ve (daha da önemlisi) verinin yazılacağı tam ofseti içerir. Böylece, bu bilgilerle tekrarlanabilir. Birden fazla yazmanın sonucu tek bir yazmanın sonucuyla aynıdır.



Şekil 49.6: Üç Tür Kayıp

Bu şekilde, istemci tüm zaman aşımlarını birleşik bir şekilde ele alabilir. Eğer bir WRITE isteği basitçe kaybedilmişse (yukarıdaki Durum 1), istemci isteği yeniden deneyecektir. Sunucu yazma işlemini gerçekleştirecek ve her şey yolunda gidecektir. Aynı durum, istek gönderildiği sırada sunucunun kapalı olması, ancak ikinci istek gönderildiğinde tekrar çalışmaya başlaması ve yine her şeyin istendiği gibi çalışması durumunda da gerçekleşecektir (Durum 2). Son olarak, sunucu aslında WRITE isteğinde bulunur, diske yazma işlemini gerçekleştirir ve bir yanıt gönderir. Bu yanıt kaybolabilir (Durum 3), bu da istemcinin isteği yeniden göndermesine neden olur. Sunucu isteği tekrar aldığında, basitçe aynı şeyi yapacaktır: verileri diske yazacak ve bunu yaptığını yanıtlayacaktır. İstemci bu kez yanıtı alırsa, her şey yine yolunda demektir ve böylece istemci hem mesaj kaybını hem de sunucu arızasını tek tip bir şekilde ele almış olur. Harika! Küçük bir ayrıntı: Bazı işlemleri idempotent olarak yapmak zordur. Örneğin, zaten var olan bir dizini oluşturmaya çalıştığınızda, mkdir isteğinin başarısız olduğu konusunda bilgilendirilirsiniz. Dolayısıyla, NFS'de, dosya sunucusu bir MKDIR protokol mesajı alır ve başarıyla yürütürse, ancak yanıt kaybolursa, istemci işlemi tekrarlayabilir ve aslında işlem ilk başta başarılı olduğu ve daha sonra yalnızca yeniden denemede başarısız olduğu halde bu başarısızlıkla karşılaşabilir. Dolayısıyla, hayat mükemmel değildir.

IPUCU: MÜKEMMEL IYININ DÜŞMANIDIR (VOLTAIRE YASASI)
Güzel bir sistem tasarladığınızda bile, bazen tüm köşe durumları tam olarak istediğiniz şekilde çalışmaz. Yukarıdaki mkdir örneğini ele alalım; mkdir farklı anlamlara sahip olacak şekilde yeniden tasarlanabilir, böylece idempotent hale getirilebilir (bunu nasıl yapabileceğinizi düşünün), ama neden uğraşasınız ki? NFS tasarım felsefesi önemli durumların çoğunu kapsar ve genel olarak sistem tasarımını arıza açısından temiz ve basit hale getirir.Bu nedenle, hayatın mükemmel olmadığını kabul etmek ve sistemi yine de inşa etmek iyi mühendisliğin bir işaretidir. Görünüşe göre, bu bilgelik Voltaire'e atfediliyor, "... bilge bir İtalyan, en iyinin iyinin düşmanı olduğunu söyler" [V72] ve bu yüzden buna Voltaire Yasası diyoruz.

49.8 Performansı İyileştirme: İstemci Tarafı Önbelleğe Alma

Dağıtılmış dosya sistemleri birçok nedenden dolayı iyidir, ancak tüm okuma ve yazma isteklerini ağ üzerinden göndermek büyük bir performans sorununa yol açabilir: ağ genellikle o kadar hızlı değildir, özellikle de yerel bellek veya disk ile karşılaştırıldığında. Dolayısıyla, başka bir sorun ortaya çıkar: dağıtılmış bir dosya sisteminin performansını nasıl artırabiliriz?

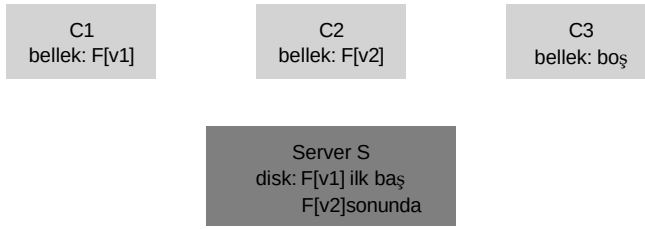
Cevap, yukarıdaki alt başlıktaki büyük kalın kelimeleri okuyarak tahmin edebileceğiniz gibi, istemci tarafı önbelleklemedir. NFS istemci tarafı dosya sistemi, sunucudan okuduğu dosya verilerini (ve meta verileri) istemci belleğinde önbelleğe alır. Böylece, ilk erişim pahalı olsa da (yani, ağ iletişimi gerektirir), sonraki erişimler istemci belleğinden oldukça hızlı bir şekilde sunulur.

Önbellek ayrıca yazmalar için geçici bir tampon görevi görür. Bir istemci uygulaması bir dosyaya ilk kez yazdığında, istemci verileri sunucuya yazmadan önce istemci belleğinde (dosya sunucusundan okuduğu verilerle aynı önbellekte) arabelleğe alır. Bu tür **yazma tamponlaması(write buffering)** kullanışlıdır çünkü uygulama write() gecikmesini gerçek yazma performansından ayırır, yani uygulamanın write() çağrısı hemen başarılı olur (ve yalnızca istemci tarafı dosya sisteminin önbelleğindeki veriler); ancak o zaman veriler dosya sunucusuna yazılır.

Bu nedenle, NFS istemcileri verileri önbelleğe alır ve performans genellikle mükemmeldir ve işimiz biter değil mi? Ne yazık ki, tam olarak değil. Birden fazla istemci önbelleği olan herhangi bir sisteme önbellek eklemek, **önbellek tutarlılık sorunu(cache consistency problem**) olarak adlandıracağımız büyük ve ilginç bir zorluk ortaya çıkarır.

49.9 Önbellek Tutarlılık Sorunu

Önbellek tutarlılığı sorunu en iyi iki istemci ve tek bir sunucu ile gösterilebilir. C1 istemcisinin bir F dosyasını okuduğunu ve dosyanın bir kopyasını yerel önbelleğinde tuttuğunu düşünün.Şimdi farklı bir istemci olan C2'nin F dosyasının üzerine yazdığını düşünün; F dosyasının yeni sürümünü (sürüm 2) veya F[v2] ve eski sürümünü F[v1] olarak adlandıralım, böylece ikisini birbirinden ayrı tutabiliriz (ancak elbette dosya aynı ada sahiptir, sadece içeriği farklıdır). Son olarak, henüz F dosyasına erişmemiş olan üçüncü bir istemci, C3, vardır.



Şekil 49.7: Önbellek Tutarlılık Sorunu

(Muhtemelen yaklaşmakta olan problemi görebilirsiniz (Şekil 49.7). Aslında iki alt problem vardır. İlk alt sorun, C2 istemcisinin yazdıklarını sunucuya yaymadan önce bir süre önbelleğinde tamponlamasıdır; bu durumda, F[v2] C2'nin belleğinde dururken, F'ye başka bir istemciden (diyelim ki C3) yapılacak herhangi bir erişim dosyanın eski sürümünü (F[v1]) getirecektir. Bu nedenle, istemcideki yazmaları arabelleğe alarak, diğer istemciler dosyanın eski sürümlerini alabilir, bu da istenmeyen bir durum olabilir; gerçekten de, C2 makinesinde oturum açtığınızı, F'yi güncellediğinizi ve ardından C3'te oturum açıp dosyayı okumaya çalıştığınızı ve yalnızca eski kopyayı aldığınızı düşünün! Bu kesinlikle sinir bozucu olabilir. Bu nedenle, önbellek tutarlılığı sorununun bu yönüne **güncelleme görünürlüğü(update visibility)** diyelim; bir istemciden gelen güncellemeler diğer istemcilerde ne zaman görünür hale gelir?

Önbellek tutarlılığının ikinci alt problemi **eski bir önbellektir(stale cache)**; bu durumda C2 sonunda yazdıklarını dosya sunucusuna aktarmıştır ve dolayısıyla sunucu en son sürüme (F[v2]) sahiptir. Ancak, C1'in önbelleğinde hala F[v1] vardır; C1'de çalışan bir program F dosyasını okursa, en son kopyayı (F[v2]) değil eski bir sürümü (F[v1]) alır ki bu (genellikle) istenmeyen bir durumdur.

NFSv2 uygulamaları bu önbellek tutarlılığı sorunlarını iki şekilde çözer. İlk olarak, güncelleme görünürlüğünü ele almak için, istemciler bazen flush-on-close (diğer adıyla close-to open) tutarlılık semantiği olarak adlandırılan şeyi uygular; özellikle, bir dosya bir istemci uygulaması tarafından yazıldığında ve daha sonra kapatıldığında, istemci tüm güncellemeleri (yani, önbellekteki kirli sayfaları) sunucuya temizler. Flush-on-close tutarlılığı ile NFS, başka bir düğümden yapılacak sonraki bir açmanın en son dosya sürümünü görmesini sağlar.

İkinci olarak, eski(bayat) önbellek sorununu ele almak için, NFSv2 istemcileri önbelleğe alınan içeriği kullanmadan önce bir dosyanın değişip değişmediğini kontrol eder. Özellikle, önbelleğe alınmış bir bloğu kullanmadan önce, istemci tarafı dosya sistemi dosyanın özniteliklerini almak için sunucuya bir GETATTR isteği gönderir. Öznitelikler, önemli olarak, dosyanın sunucuda en son ne zaman değiştirildiği bilgisini içerir; değişiklik zamanı dosyanın istemci önbelleğine getirildiği zamandan daha yeni ise, istemci dosyayı geçersiz kılar(invalides), böylece istemci önbelleğinden kaldırır ve sonraki okumaların sunucuya gitmesini ve dosyanın en son sürümünü almasını sağlar. Öte yandan, istemci dosyanın en son sürümüne sahip olduğunu görürse, devam edecek ve önbelleğe alınan içeriği kullanacak, böylece performansı artıracaktır.

Sun'daki orijinal ekip eski(bayat) bellek sorununa bu çözümü uyguladığında, yeni bir sorunun farkına vardılar; aniden NFS sunucusu GETATTR istekleriyle doldu. İzlenmesi gereken iyi bir mühendislik ilkesi, genel durum için tasarım yapmak ve bunun iyi çalışmasını sağlamaktır; burada, genel durum bir dosyaya yalnızca tek bir istemciden (belki de tekrar tekrar) erişilmesi olmasına rağmen, istemcinin dosyayı başka kimsenin değiştirmediğinden emin olmak için sunucuya her zaman GETATTR istekleri göndermesi gerekiyordu. Böylece bir istemci sunucuyu bombardımana tutarak sürekli "bu dosyayı değiştiren oldu mu?" diye sorar, oysa çoğu zaman kimse değiştirmemiştir.

Bu durumu (biraz) düzeltmek için her istemciye bir **öznitelik önbelleği(attribute cache)** eklendi. İstemci bir dosyaya erişmeden önce yine de doğrulama yapacak, ancak çoğu zaman öznitelikleri almak için öznitelik önbelleğine bakacaktı. Belirli bir dosyanın öznitelikleri, dosyaya ilk erişildiğinde önbelleğe yerleştirilir ve belirli bir süre sonra zaman aşımına uğrar(diyelim ki 3 saniye). Böylece, bu üç saniye boyunca, tüm dosya erişimleri önbellekteki dosyayı kullanmanın uygun olduğunu belirleyecek ve böylece sunucu ile ağ iletişimi olmadan bunu yapacaktır.

49.10 NFS Önbellek Tutarlılığının Değerlendirilmesi

NFS önbellek tutarlılığı hakkında son birkaç söz. Flush-on-close davranışı "mantıklı" olması için eklenmiştir, ancak belirli bir performans sorunu ortaya çıkarmıştır. Özellikle, geçici ya da kısa ömürlü bir dosya istemcide oluşturulduktan sonra kısa süre içinde silinirse, bu dosya sunucuya zorla gönderilmeye devam ederdi. 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 mantık yürütmeyi çok zorlaştırdı. Bazen en son sürümü alırdınız; bazen de eski bir sürümü alırdınız çünkü öznitelik önbelleğiniz henüz zaman aşımına uğramamıştı ve bu nedenle istemci size istemci belleğinde olanı vermekten mutluydu. Bu çoğu zaman iyi olsa da (ve hala da öyle!) zaman zaman garip davranışlara yol açıyor.

Ve böylece NFS istemci önbelleklemesinde olan tuhaflığı tanımlamış olduk. Bu, bir uygulamanın ayrıntılarının kullanıcı tarafından gözlemlenebilir semantikleri tanımlamaya hizmet ettiği ilginç bir örnektir.

49.11 Sunucu Tarafı Yazma Tamponlaması Üzerine Etkiler

Şimdiye kadar istemci önbelleğine odaklandık ve ilginç sorunların çoğu burada ortaya çıkıyor. Bununla birlikte, NFS sunucuları da çok fazla belleğe sahip iyi donanımlı makineler olma eğilimindedir ve bu nedenle önbelleğe alma endişeleri de vardır. Veriler (ve meta veriler) diskten okunduğunda, NFS sunucuları bu verileri bellekte tutacak ve söz konusu verilerin (ve meta verilerin) sonraki okumaları diske gitmeyecek, bu da performansta potansiyel (küçük) bir artış sağlayacaktır.

Daha ilgi çekici olan ise yazma tamponlama durumudur. NFS sunucuları, yazma işlemi sabit depolama alanına (örn. diske ya da başka bir kalıcı aygıta) zorlanana kadar bir WRITE protokolü isteğinde kesinlikle başarı döndürmeyebilir. Verilerin bir kopyasını sunucu belleğine yerleştirebilirken, bir WRITE protokolü isteğinde istemciye başarı döndürmek yanlış davranışa neden olabilir; nedenini bulabilir misiniz?

Cevap, istemcilerin sunucu arızasını nasıl ele aldığına ilişkin varsayımlarımızda yatmaktadır. Bir istemci tarafından yayınlanan aşağıdaki yazma dizisini hayal edin:

```
write(fd, a_buffer, size); // fill 1st block with a's
write(fd, b_buffer, size); // fill 2nd block with b's
write(fd, c_buffer, size); // fill 3rd block with c's
```

Bu yazımlar, bir dosyanın üç bloğunun üzerine a'lardan, sonra b'lerden ve sonra da c'lerden oluşan bir blok yazar. Böylece, eğer dosya başlangıçta şöyle görünüyorsa:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Bu yazımlardan sonra nihai sonucun şu şekilde olmasını bekleyebiliriz: x'ler, y'ler ve z'lerin üzerine sırasıyla a'lar, b'ler ve c'ler yazılacaktır.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccccc
```

Şimdi, örneklere göre, bu üç istemci yazma işleminin sunucuya üç ayrı WRITE protokol mesajı olarak verildiğini varsayalım. İlk WRITE mesajının sunucu tarafından alındığını ve diske verildiğini ve istemcinin bunun başarılı olduğu konusunda bilgilendirildiğini varsayalım. Şimdi ikinci yazma işleminin bellekte arabelleğe alındığını ve sunucunun diske yazmaya zorlamadan önce istemciye başarılı olduğunu bildirdiğini varsayalım; ne yazık ki sunucu diske yazmadan önce çöküyor. Sunucu hızla yeniden başlar ve üçüncü yazma isteğini alır, bu da başarılı olur.

Böylece, istemci için tüm istekler başarılı oldu, ancak dosya içeriğinin bu şekilde görünmesi bizi şaşırttı:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
cccccccccccccccccccccccccccccccccccccccccccccccc
```

Eyvah! Sunucu, diske işlemeden önce istemciye ikinci yazmanın başarılı olduğunu söylediği için dosyada eski bir yığın kalır ve bu da uygulamaya bağlı olarak felakete yol açabilir.

BIR TARAFTAN: YENİLİK YENİLİĞİ DOĞURUR
Birçok öncü teknolojide olduğu gibi, NFS'nin dünyaya getirilmesi de başarısını sağlamak için başka temel yenilikler gerektirmiştir. Muhtemelen en kalıcı olanı, farklı dosya sistemlerinin işletim sistemine kolayca takılabilmesini sağlamak için Sun tarafından sunulan **Sanal Dosya Sistemi (Virtual File System-VFS) / Sanal Düğüm (Virtual Node-vnode)** arayüzüdür [K86].
VFS katmanı, takma ve çıkarma, dosya sistemi çapında istatistikler alma gibi tüm bir dosya sistemine yapılan işlemleri içerir ve tüm kirli (henüz yazılmamış) yazmaları diske zorlar. Vnode katmanı, bir dosya üzerinde gerçekleştirilebilecek açma, kapatma, okuma, yazma ve benzeri tüm işlemlerden oluşur.
Yeni bir dosya sistemi oluşturmak için bu "yöntemleri" tanımlamak 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 fazla dosya sistemi uygulamasının aynı sistem içinde aynı anda çalışması için bir yol sağlayarak gerisini halleder.
Bazı detaylar değişmiş olsa da Linux, BSD varyantları, macOS dahil olmak üzere birçok modern sistem bir çeşit VFS/vnode katmanına sahiptir ve hatta Windows (Yüklenebilir Dosya Sistemi şeklinde). NFS dünya için daha az önemli hale gelse bile, altındaki bazı gerekli temeller yaşamaya devam edecektir.

Bu sorundan kaçınmak için NFS sunucuları, istemciyi başarı konusunda bilgilendirmeden önce her yazmayı sabit (kalıcı) depolama alanına işlemelidir; bunu yapmak, istemcinin bir yazma sırasında sunucu arızasını tespit etmesini ve böylece sonunda başarılı olana kadar yeniden denemesini sağlar. Bunu yapmak, yukarıdaki örnekte olduğu gibi dosya içeriklerinin asla birbirine karışmamasını sağlar.

Bu gereksinimin NFS sunucu uygulamasında ortaya çıkardığı sorun, yazma performansının, büyük bir özen gösterilmediği takdirde, başlıca performans engelidir. Gerçekten de,bazı şirketler (örneğin, Network Appliance) hızlı bir şekilde yazma işlemi gerçekleştirebilen bir NFS sunucusu oluşturmak gibi basit bir amaçla ortaya çıkmıştır. Kullandıkları ilk numara: Yazma işlemlerini pil destekli bir belleğe koyar, böylece verileri kaybetme korkusu olmadan ve diske hemen yazma maliyeti olmadan WRITE isteklerine hızlı bir şekilde yanıt vermeyi sağlar; ikinci numara, nihayetinde bunu yapmak gerektiğinde diske hızlı bir şekilde yazmak için özel olarak tasarlanmış bir dosya sistemi tasarımı kullanmaktır [HLM94, RO91].

49.12 ÖZET

NFS dağıtılmış dosya sisteminin tanıtıldığını gördük. NFS, sunucu arızası karşısında basit ve hızlı kurtarma fikrine odaklanır ve bu amaca dikkatli protokol tasarımı ile ulaşır. İşlemlerin isteğe bağlı olması esastır; bir istemci başarısız bir işlemi güvenli bir şekilde tekrarlayabildiğinden, sunucu isteği gerçekleştirmiş olsun ya da olmasın bunu yapmakta bir sakınca yoktur.

BİR TARAFTAN: ANAHTAR NFS TERIMLERI
NFS'de hızlı ve basit çökme kurtarma ana hedefini gerçekleştirmenin anahtarı, durumsuz(stateless) bir protokolün tasarlanmasıdır. Bir çökmeden sonra, sunucu hızlı bir şekilde yeniden başlatılabilir ve istekleri tekrar sunmaya başlayabilir; istemciler başarılı olana kadar istekleri yeniden dener(retry).
İstekleri boşta bırakma, NFS protokolünün merkezi bir özelliğidir. Bir işlemin birden fazla kez gerçekleştirilmesinin etkisi bir kez gerçekleştirilmesine eşdeğer olduğunda, o işlem idempotenttir(denk güçlük). NFS'de idempotentlik, istemcinin endişe duymadan yeniden denemesini sağlar ve istemci kayıp mesaj yeniden iletimini ve istemcinin sunucu çökmelerini nasıl ele aldığını birleştirir.
Performans kaygıları**, istemci tarafı önbelleğe(caching)** alma ve yazma arabelleğine alma-**yazma tamponlama (write buffering)** ihtiyacını belirler, ancak bir **önbellek tutarlılığı sorunu(cache consistency problem**) ortaya çıkarır.
NFS uygulamaları, önbellek tutarlılığı için birden fazla yolla mühendislik çözümü sunar: flush-on-close (close-to-open) yaklaşımı, bir dosya kapatıldığında içeriğinin sunucuya zorla gönderilmesini sağlayarak diğer istemcilerin dosyadaki güncellemeleri gözlemlemesine olanak tanır. Bir öznitelik önbelleği, bir dosyanın değişip değişmediğini sunucudan kontrol etme sıklığını azaltır (GETATTR istekleri aracılığıyla).
NFS sunucuları başarı döndürmeden önce kalıcı ortama yazma işlemi yapmalıdır; aksi takdirde veri kaybı yaşanabilir.
NFS'nin işletim sistemine entegrasyonunu desteklemek için Sun, VFS/Vnode (Sanal Dosya Sistemi -Sanal Düğüm) arayüzünü tanıtarak birden fazla dosya sistemi uygulamasının aynı işletim sisteminde bir arada bulunmasını sağladı.

Ayrıca çok istemcili, tek sunuculu bir sisteme önbellek eklemenin işleri nasıl karmaşıklaştırabileceğini gördük. Özellikle, sistemin makul bir şekilde davranabilmesi için önbellek tutarlılığı sorununu çözmesi gerekir; ancak NFS bunu biraz geçici bir şekilde yapar ve bu da zaman zaman gözlemlenebilir derecede garip davranışlara neden olur. Son olarak, sunucu önbelleğe almanın nasıl zor olabileceğini gördük: sunucuya yapılan yazmalar başarıya dönmeden önce sabit depolamaya zorlanmalıdır.(aksi takdirde veriler kaybolabilir)

Kesinlikle ilgili olan diğer konulardan, özellikle de güvenlikten bahsetmedik. İlk NFS uygulamalarında güvenlik oldukça gevşekti; bir istemcideki herhangi bir kullanıcının diğer kullanıcılar gibi davranması ve böylece neredeyse her dosyaya erişmesi oldukça kolaydı. Daha ciddi kimlik doğrulama hizmetleriyle (örneğin Kerberos [NT94]) daha sonraki entegrasyonlar bu bariz eksiklikleri gidermiştir.

References

[AKW88] "The AWK Programming Language" by Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. Pearson, 1988 (1st edition). *A concise, wonderful book about awk. We once had the pleasure of meeting Peter Weinberger; when he introduced himself, he said "I'm Peter Weinberger, you know, the 'W' in awk?" As huge awk fans, this was a moment to savor. One of us (Remzi) then said, "I love awk! I particularly love the book, which makes everything so wonderfully clear." Weinberger replied (crestfallen), "Oh, Kernighan wrote the book."*

[C00] "NFS Illustrated" by Brent Callaghan. Addison-Wesley Professional Computing Series, 2000. *A great NFS reference; incredibly thorough and detailed per the protocol itself.*

[ES03] "New NFS Tracing Tools and Techniques for System Analysis" by Daniel Ellard and Margo Seltzer. LISA '03, San Diego, California. *An intricate, careful analysis of NFS done via passive tracing. By simply monitoring network traffic, the authors show how to derive a vast amount of file system understanding.*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Winter 1994. San Francisco, California, 1994. *Hitz et al. were greatly influenced by previous work on log-structured file systems.*

[K86] "Vnodes: An Architecture for Multiple File System Types in Sun UNIX" by Steve R. Kleiman. USENIX Summer '86, Atlanta, Georgia. *This paper shows how to build a flexible file system architecture into an operating system, enabling multiple different file system implementations to coexist. Now used in virtually every modern operating system in some form.*

[NT94] "Kerberos: An Authentication Service for Computer Networks" by B. Clifford Neuman, Theodore Ts'o. IEEE Communications, 32(9):33-38, September 1994. *Kerberos is an early and hugely influential authentication service. We probably should write a book chapter about it sometime...*

[O91] "The Role of Distributed State" by John K. Ousterhout. 1991. Available at this site: `ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps`. *A rarely referenced discussion of distributed state; a broader perspective on the problems and challenges.*

[P+94] "NFS Version 3: Design and Implementation" by Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz. USENIX Summer 1994, pages 137-152. *The small modifications that underlie NFS version 3.*

[P+00] "The NFS version 4 protocol" by Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow. 2nd International System Administration and Networking Conference (SANE 2000). *Undoubtedly the most literary paper on NFS ever written.*

[RO91] "The Design and Implementation of the Log-structured File System" by Mendel Rosenblum, John Ousterhout. Symposium on Operating Systems Principles (SOSP), 1991. *LFS again. No, you can never get enough LFS.*

[S86] "The Sun Network File System: Design, Implementation and Experience" by Russel Sandberg. USENIX Summer 1986. *The original NFS paper; though a bit of a challenging read, it is worthwhile to see the source of these wonderful ideas.*

[Sun89] "NFS: Network File System Protocol Specification" by Sun Microsystems, Inc. Request for Comments: 1094, March 1989. Available: `http://www.ietf.org/rfc/rfc1094.txt`. *The dreaded specification; read it if you must, i.e., you are getting paid to read it. Hopefully, paid a lot. Cash money!*

[V72] "La Begueule" by Francois-Marie Arouet a.k.a. Voltaire. Published in 1772. *Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they will dwell in peace." What do you say to that, Democrats and Republicans?*

Ödev (Ölçme)

Bu ödevde, gerçek izler kullanarak biraz NFS iz analizi yapacaksınız. Bu izlerin kaynağı Ellard ve Seltzer'in çalışmasıdır [ES03]. Başlamadan önce ilgili README'yi okuduğunuzdan ve OSTEP ödev sayfasından (her zamanki gibi) ilgili tarball'unu indirdiğinizden emin olun.

Sorular

1. İz analiziniz için ilk soru: ilk sütunda bulunan zaman damgalarını kullanarak, izlerin alındığı zaman dilimini belirleyin. Dönem ne kadar uzun? 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.
2. Şimdi biraz işlem sayımı yapalım. İzde her bir işlem türünden kaç tane gerçekleşiyor? Bunları sıklığa göre sıralayın; en sık hangi işlem yapılıyor? NFS ününün hakkını veriyor mu?
3. Şimdi bazı özel işlemleri daha ayrıntılı olarak inceleyelim. Örneğin, GETATTR isteği, isteğin hangi kullanıcı kimliği için gerçekleştirildiği, dosyanın boyutu ve benzeri dahil olmak üzere dosyalar hakkında birçok bilgi döndürür. İzleme içinde erişilen dosya boyutlarının bir dağılımını yapın; ortalama dosya boyutu nedir? Ayrıca, izdeki dosyalara kaç farklı kullanıcı erişiyor? Trafiği birkaç kullanıcı mı domine ediyor, yoksa daha mı dağınık? GETATTR yanıtlarında başka hangi ilginç bilgiler bulunur?
4. Ayrıca belirli bir dosyaya yapılan isteklere bakabilir ve dosyalara nasıl erişildiğini belirleyebilirsiniz. Örneğin, belirli bir dosya sırayla mı okunuyor veya yazılıyor? Yoksa rastgele mi? Cevabı hesaplamak için OKU ve YAZ isteklerinin/cevaplarının ayrıntılarına bakın.
5. Trafik birçok makineden geliyor ve tek bir sunucuya gidiyor (bu izde). İzlemde kaç farklı istemci olduğunu ve her birine kaç istek/cevap gittiğini gösteren bir trafik matrisi hesaplayın. Birkaç makine mi baskın, yoksa daha dengeli mi?
6. Zamanlama bilgileri ve talep/cevap başına benzersiz kimlik, belirli bir talep için gecikme süresini hesaplamanıza izin vermelidir. Tüm istek/cevap çiftlerinin gecikme sürelerini hesaplayın ve bunları bir dağılım olarak çizin. Ortalama nedir? Maksimum? Minimum?
7. Bazen istekler yeniden denenir, çünkü istek veya yanıtı kaybolabilir veya düşebilir. İzleme örneğinde bu tür yeniden denemelere ilişkin herhangi bir kanıt bulabilir misiniz?
8. Daha fazla analiz yaparak cevaplayabileceğiniz başka birçok soru var. Sizce hangi sorular önemli? Bunları bize önerin, belki biz de buraya ekleriz!
   Yukarıdaki soruları çözebilmem için gereken Ellard ve Seltzer'in çalışmasını readme dosyası olarak githubda bulamadım internette araştırdım ama bulduğum analiz sorusu yine eksik bilgi içeriyordu ve çalışmadı dolayısıyla soruları çözemedim.