49

Sun'ın Ağ Dosya Sistemi (NFS)

Dağıtık istemci/sunucu bilişiminin ilk kullanım alanlarından biri dağıtık dosya sistemleri olmuştur. Böyle bir ortamda, çok sayıda istemci makine ve bir sunucu (ya da birkaçı) vardır; sunucu verileri disklerinde depolar ve istemciler iyi biçimlendirilmiş protokol mesajları aracılığıyla veri talep eder. Şekil 49.1'de temel kurulum gösterilmektedir.

Ağ

RAID

Sunucu

İstemci 3

İstemci 2

İstemci 1

İstemci 0

**Şekil 49.1: Genel Bir İstemci/Sunucu Sistemi**

Resimden de görebileceğiniz gibi, sunucu disklere sahiptir ve istemciler budisklerdeki dizinlerine ve dosyalarına erişmek için bir ağ üzerinden mesaj gönderirler. Neden bu düzenlemeyle uğraşıyoruz? (yani, neden istemcilerin kendi yerel disklerini kullanmalarına izin vermiyoruz?) Öncelikle bu kurulum, istemciler arasında verilerin kolayca paylaşılmasını **(sharing)** sağlar. Böylece, bir makinede (İstemci 0) bir dosyaya erişirseniz ve daha sonra başka bir makineyi (İstemci 2) kullanırsanız, dosya sisteminin aynı görünümüne sahip olursunuz. Verileriniz doğal olarak bu farklı makineler arasında paylaşılır. İkincil bir fayda ise merkezi yönetimdir**(centralized administration)**; örneğin, dosyaların yedeklenmesi çok sayıda istemci yerine birkaç sunucu makineden yapılabilir. Bir başka avantaj da güvenlik**(security)** olabilir; tüm sunucuların kilitli bir makine odasında olması belirli türden sorunların ortaya çıkmasını engeller.

1

ÖNEMLİ NOKTA: DAĞITILMIŞ DOSYA SİSTEMİ NASIL OLUŞTURULUR

Dağıtılmış bir dosya sistemini nasıl oluşturursunuz? Düşünülmesi gereken kilit noktalar nelerdir? Neleri yanlış yapmak kolaydır? Mevcut sistemlerden ne öğrenebiliriz?

## Temel Bir Dağıtılmış Dosya Sistemi

Şimdi basitleştirilmiş bir dağıtık dosya sisteminin mimarisini inceleyeceğiz. Basit bir istemci/sunucu dağıtık dosya sistemi, şimdiye kadar incelediğimiz dosya sistemlerinden daha fazla bileşene sahiptir. İstemci tarafında, istemci tarafı dosya sistemi **(client- side file system)** aracılığıyla dosya ve dizinlere erişen istemci uygulamaları vardır. Bir istemci uygulaması, sunucuda depolanan dosyalara erişmek için istemci tarafı dosya sistemine sistem çağrıları (**system calls)** (open(), read(), write(), close(), mkdir(), vb. gibi) gönderir. Böylece, istemci uygulamaları için dosya sistemi, belki performans dışında, local (disk tabanlı) bir dosya sisteminden farklı görünmez; bu şekilde, dağıtılmış dosya sistemleri dosyalara şeffaf **(transparent)** erişim sağlar, bu da bariz bir hedeftir; sonuçta, kim farklı bir API seti gerektiren veya başka bir şekilde kullanımı zahmetli olan bir dosya sistemi kullanmak ister ki?

İstemci tarafı dosya sisteminin rolü, bu sistem çağrılarına hizmet etmek için gereken eylemleri gerçekleştirmektir. 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 geri gönderecektir. İstemci tarafındaki dosya sistemi daha sonra verileri read() sistem çağrısına verilen kullanıcı arabelleğine kopyalayacak ve böylece istek tamamlanacaktır. İstemcide aynı bloğun sonraki bir read() çağrısının istemci belleğinde veya istemcinin diskinde bile önbelleğe alınabileceğini **(cached)** unutmayın; böyle en iyi durumda, ağ trafiği oluşturulmasına gerek yoktur.

Müşteri Uygulaması

Disks

Client-side File System

File Server

Networking Layer

Networking Layer

**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. İkisinin birlikte davranışı dağıtık dosya sisteminin davranışını belirler. Şimdi belirli bir sistemi incelemenin zamanı geldi: Sun'ın Ağ Dosya Sistemi (NFS).

AYRICA: SUNUCULAR NEDEN ÇÖKER?

NFSv2 protokolünün ayrıntılarına girmeden önce merak ediyor olabilirsiniz: sunucular neden çöker? Tahmin edebileceğiniz gibi bunun pek çok nedeni vardır. Sunucular basitçe (geçici olarak) elektrik kesintisine **(power outage)** uğrayabilir; yalnızca elektrik geri geldiğinde makineler yeniden başlatılabilir. Sunucular genellikle yüz binlerce hatta milyonlarca satır koddan oluşur; bu nedenle hataları **(bugs)** vardır (iyi yazılımlarda bile her yüz veya bin satır kodda birkaç hata bulunur) ve bu nedenle eninde sonunda çökmelerine neden olacak bir hatayı tetikleyeceklerdir. Ayrıca bellek sızıntıları da vardır; küçük bir bellek sızıntısı bile bir sistemin belleğinin tükenmesine ve çökmesine neden olur. Ve son olarak, dağıtık sistemlerde, istemci ve sunucu arasında bir ağ vardır; ağ garip davranırsa (örneğin, bölümlere ayrılırsa **(partitioned)** ve istemciler ve sunucular çalışıyor ancak iletişim kuramıyorsa), uzak bir makine çökmüş gibi görünebilir, ancak gerçekte o anda ağ üzerinden erişilemez.

## NFS'ye Giriş

En eski ve oldukça başarılı dağıtık sistemlerden biri Sun Microsystems tarafından geliştirilmiştir ve Sun Network File Sys- tem (ya da NFS) olarak bilinir [S86]. Sun, NFS'yi tanımlarken alışılmadık bir yaklaşım benimsedi: Sun, tescilli ve kapalı bir sistem kurmak yerine, istemcilerin ve sunucuların iletişim kurmak için kullanacakları mesaj formatlarını basitçe belirten açık bir protokol **(open protocol)** geliştirdi. Farklı gruplar kendi NFS sunucularını geliştirebilir ve böylece birlikte çalışabilirliği korurken NFS pazarında rekabet edebilirlerdi. 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 bağlı.

## Odak Noktası: Basit ve Hızlı Sunucu Çökmesi Kurtarma

Bu bölümde klasik NFS protokolünü (sürüm 2,a.k.a. NFSv2) ele alacağız, uzun yıllar boyunca standarttı; NFSv3'e geçerken küçük değişiklikler yapıldı ve NFSv4'e geçerken daha büyük ölçekli protokol değişiklikleri yapıldı. Bununla birlikte, NFSv2 hem harika hem de sinir bozucudur ve bu nedenle odak noktamız olarak hizmet vermektedir.

NFSv2'de, protokolün tasarımındaki ana hedef basit ve hızlı sunucu çökmesi kurtarmasıydı. Çok istemcili, tek sunuculu bir ortamda, bu hedef çok anlamlıdır; sunucunun kapalı olduğu (veya kullanılamadığı) her dakika, tüm istemci makineleri (ve kullanıcılarını) mutsuz ve verimsiz hale getirir. Dolayısıyla, sunucu çöktüğünde tüm sistem de çöker.

## Hızlı Kaza Kurtarmanın Anahtarı: Durumsuzluk

Bu basit hedef, NFSv2'de durum bilgisi olmayan **(stateless)** bir protokol olarak adlandırdığımız bir tasarımla gerçekleştirilmiştir. Sunucu, tasarım gereği, her bir istemcide neler olup bittiğine dair hiçbir şeyi takip etmez. Örneğin, sunucu hangi istemcilerin hangi blokları önbelleğe aldığını veya her istemcide hangi dosyaların açık olduğunu veya bir dosya için geçerli dosya işaretçisi konumunu vb. bilmez. Basitçe söylemek gerekirse, sunucu istemcilerin ne yaptığı hakkında hiçbir şey takip etmez; bunun yerine, protokol her protokol isteğinde isteği tamamlamak için gereken tüm bilgileri iletmek üzere tasarlanmıştır. Eğer şimdi değilse, protokolü aşağıda daha ayrıntılı olarak tartıştığımızda bu durumsuz yaklaşım daha anlamlı hale gelecektir.

Durumlu **(stateful)** (durumsuz değil) bir protokol örneği için open() sistem çağrısını düşünün. Bir yol adı verildiğinde, open() bir dosya tanımlayıcısı (bir integer) döndürür. Bu tanımlayıcı, bu uygulama kodunda olduğu gibi çeşitli dosya bloklarına erişmek için sonraki read() veya write() isteklerinde kullanılır (sistem çağrılarının uygun hata denetiminin yer nedeniyle atlandığına dikkat edin):

char buffer[MAX];

int fd = open("foo", O\_RDONLY); // "fd" tanımlayıcısını alın

read(fd, buffer, MAX); // "fd" aracılığıyla foo'dan MAX'ı okuyun

read(fd, buffer, MAX); // MAX'i tekrar okuyun

...

read(fd, buffer, MAX); // MAX'i tekrar okuyun

close(fd); // dosyayı kapat

**Şekil 49.3: İstemci Kodu: Bir Dosyadan Okuma**

Şimdi istemci tarafındaki dosya sisteminin sunucuya "foo dosyasını aç ve bana bir tanımlayıcı ver" şeklinde bir protokol mesajı göndererek dosyayı açtığını düşünün. 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 dosya sunucusuna bir mesajla iletir ve "size burada ilettiğim tanımlayıcı tarafından atıfta bulunulan dosyadan bazı baytları okuyun" der.

Bu örnekte, dosya tanımlayıcısı istemci ve sunucu arasında paylaşılan bir durum **(shared state)** parçasıdır (Ousterhout buna dağıtılmış durum **(distributed state)** demektedir [O91]). Yukarıda ima ettiğimiz gibi paylaşılan durum, çökme kurtarmayı zorlaştırır. Sunucunun ilk okuma tamamlandıktan sonra, ancak istemci ikinci okumayı yapmadan önce çöktüğünü düşünün. Sunucu tekrar çalışır hale geldikten sonra, istemci ikinci okumayı gerçekleştirir. Ne yazık ki, sunucunun fd'nin hangi dosyaya atıfta bulunduğuna dair hiçbir fikri yoktur; bu bilgi geçicidir (yani bellektedir) ve dolayısıyla sunucu çöktüğünde kaybolur. Bu durumla başa çıkmak için, istemci ve sunucunun bir tür kurtarma protokolüne **(recovery protocol)** girmesi gerekir; bu protokolde istemci, sunucuya bilmesi gerekenleri (bu durumda, fd dosya tanımlayıcısının foo dosyasına atıfta bulunduğunu) söyleyebilmek için belleğinde yeterli bilgi tuttuğundan emin olur.

Durum bilgisine sahip bir sunucunun istemci çökmeleriyle uğraşmak zorunda olduğunu düşündüğünüzde durum daha da kötüleşir. Örneğin, bir dosya açan ve sonra çöken bir istemci düşünün. open() işlevi sunucuda bir dosya tanımlayıcısı kullanır; sunucu belirli bir dosyayı kapatmanın uygun olduğunu nereden bilebilir? Normal işleyişte, istemci eninde sonunda close() işlevini çağırır ve böylece sunucuya dosyanın kapatılması gerektiğini bildirir. Ancak, bir istemci çöktüğünde, sunucu asla bir close() çağrısı almaz ve bu nedenle dosyayı kapatmak için istemcinin çöktüğünü fark etmesi gerekir.

Bu nedenlerden dolayı, NFS tasarımcıları durumsuz bir yaklaşım izlemeye karar verdiler: her istemci işlemi, isteği tamamlamak için gereken tüm bilgileri içerir. Süslü çökme kurtarma işlemlerine gerek yoktur; sunucu yeniden çalışmaya başlar ve bir istemci en kötü ihtimalle bir isteği yeniden denemek zorunda kalabilir.

## NFSv2 Protokolü

Böylece NFSv2 protokol tanımına ulaşıyoruz. Problem durumumuz basittir:

KRİTİK NOKTA: DURUMSUZ BİR DOSYA PROTOKOLÜ NASIL TANIMLANIR

Durum bilgisiz çalışmayı mümkün kılmak için ağ protokolünü nasıl tanımlayabiliriz? Açıkçası, open() gibi durumlu çağrılar tartışmanın bir parçası olamaz (çünkü sunucunun açık dosyaları izlemesini gerektirir); ancak, istemci uygulaması dosya ve dizinlere erişmek için open(), read(), write(), close() ve diğer standart API çağrılarını çağırmak isteyecektir. Dolayısıyla, daha rafine bir soru olarak, protokolü hem durumsuz olacak hem de POSIX dosya sistemi API'sini destekleyecek şekilde nasıl tanımlayabiliriz?

NFS protokolünün tasarımını anlamanın anahtarlarından biri dosya tanıtıcısını **(file handle)** anlamaktır. Dosya tanıtıcıları, belirli bir işlemin üzerinde çalışacağı dosya veya dizini benzersiz bir 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; bu üç öğe birlikte, bir istemcinin erişmek istediği bir dosya veya dizin için benzersiz bir tanımlayıcı oluşturur. Birim tanımlayıcısı sunucuya isteğin hangi dosya sistemiyle ilgili olduğunu bildirir (bir NFS sunucusu birden fazla dosya sistemini dışa aktarabilir); inode numarası sunucuya isteğin bu bölüm içindeki hangi dosyaya eriştiğini söyler. Son olarak, bir inode numarası yeniden kullanıldığında nesil numarasına ihtiyaç duyulur; bir inode numarası yeniden kullanıldığında bunu artırarak, sunucu eski bir dosya tanıtıcısına sahip bir istemcinin yanlışlıkla yeni tahsis edilen dosyaya erişememesini sağlar.

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

NFSPROC GETATTR dosya tanıtıcısı

geri döndürür: nitelikler

NFSPROC SETATTR dosya tanıtıcısı, nitelikler

geri döndürür: –

NFSPROC LOOKUP dizin dosya tanıtıcısı, aranacak dosya/dizin adı

geri döndürür: dosya tanıtıcısı

NFSPROC READ dosya tanıtıcısı, ofset, sayım

veri, nitelikler

NFSPROC WRITE dosya tanıtıcısı, ofset, sayım,veri

nitelikler

NFSPROC CREATE dizin dosya tanıtıcısı, dosya adı, öznitelikler

–

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, belirteç

döndürür: dizin girişleri, belirteç (daha fazla giriş almak için)

**Şekil 49.4: NFS Protokolü: Örnekler**

Protokolün önemli bileşenlerini kısaca vurgulayacağız. İlk olarak, LOOKUP protokol mesajı bir dosya tanıtıcısı elde etmek için kullanılır ve bu tanıtıcı daha sonra dosya verilerine erişmek için kullanılır. İstemci bir dizin dosya tanıtıcısını ve aranacak dosyanın adını iletir ve bu dosyanın (veya dizinin) tanıtıcısı artı öznitelikleri sunucudan istemciye geri iletilir.

Örneğin, istemcinin zaten bir dosya sisteminin kök dizini (/) için bir dizin dosya tanıtıcısına sahip olduğunu varsayalım (aslında bu, istemcilerin ve sunucuların ilk olarak birbirine nasıl bağlandığı NFS bağlama protokolü **(mount protocol)** aracılığıyla elde edilir; kısalık uğruna burada bağlama protokolünü tartışmıyoruz). İstemci üzerinde çalışan bir uygulama dosyayı açarsa

/foo.txt, istemci tarafı dosya sistemi sunucuya bir arama isteği göndererek kök dosya tanıtıcısını ve foo.txt adını iletir; başarılı olursa, foo.txt için dosya tanıtıcısı (ve öznitelikleri) döndürülür.

Merak ediyorsanız, öznitelikler, dosya sisteminin her dosya hakkında izlediği, dosya oluşturma zamanı, son değişiklik zamanı, boyut, sahiplik ve izin bilgileri gibi alanları içeren meta verilerdir, yani bir dosyada stat() işlevini çağırdığınızda geri alacağınız bilgilerle aynı türdendir.

Bir dosya tanıtıcısı mevcut olduğunda, istemci dosyayı okumak veya yazmak için sırasıyla READ ve WRITE protokol mesajlarını verebilir. READ protokol mesajı, protokolün dosyanın dosya tanıtıcısını, dosya içindeki ofset ve okunacak bayt sayısı ile birlikte iletmesini gerektirir. Sunucu daha sonra okuma işlemini gerçekleştirebilir (sonuçta, tanıtıcı sunucuya hangi birimden ve hangi inode'dan okunacağını söyler ve ofset ve sayı da dosyanın hangi baytlarının okunacağını söyler) ve verileri

istemciye geri gönderebilir (veya bir hata varsa bir hata). WRITE da benzer şekilde işlenir, ancak veri istemciden sunucuya aktarılır ve sadece bir başarı kodu döndürülür.

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

## Protokolden Dağıtılmış Dosya Sistemine

Umarım şimdi bu protokolün istemci tarafı dosya sistemi ve dosya sunucusu arasında nasıl bir dosya sistemine dönüştürüldüğüne dair bir fikir edinmişsinizdir. İstemci tarafı dosya sistemi açık dosyaları izler ve genellikle uygulama isteklerini ilgili protokol mesajları kümesine aktarır. 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 sistemi ve dosya sunucusunun bu çağrılara yanıt verirken ne yaptığını gösteriyoruz.

Şekil hakkında birkaç yorum. İlk olarak, istemcinin, tamsayı dosya tanımlayıcısının bir NFS dosya tanıtıcısına eşlenmesi ve geçerli dosya işaretçisi de dahil olmak üzere dosya erişimi için ilgili tüm durumu **(state)** nasıl izlediğine dikkat edin. Bu, istemcinin her okuma isteğini (okunacak ofseti açıkça belirtmediğini fark etmişsinizdir) 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 ancak farklı bir ofset ile verilir.

İkinci olarak, sunucu etkileşimlerinin nerede gerçekleştiğini fark edebilirsiniz. Dosya ilk kez açıldığında, istemci tarafı dosya sistemi bir ARA istek mesajı gönderir. Gerçekten de, uzun bir yol adının geçilmesi gerekiyorsa (örn,

/home/remzi/foo.txt), istemci üç ARA mesajı gönderir: bir tanesi / dizininde home dosyasını aramak için, bir tanesi home dizininde remzi dosyasını aramak için ve son olarak bir tanesi de remzi dizininde foo.txt dosyasını aramak için.

Üçüncü olarak, her sunucu isteğinin, isteği tamamlamak için gereken tüm bilgileri bütünüyle nasıl içerdiğini fark edebilirsiniz. Bu tasarım noktası, şimdi daha ayrıntılı olarak tartışacağımız gibi, sunucu arızasından zarif bir şekilde kurtulabilmek için kritik öneme sahiptir; sunucunun talebe yanıt verebilmek için duruma ihtiyaç duymamasını sağlar.

**İstemci Sunucu**

**fd = open(”/foo”, ...);**

Send LOOKUP (rootdir FH, ”foo”)

LOOKUP yanıtını al

açık dosya tablosunda dosya desc ayır foo'nun FH'sini tabloda sakla geçerli dosya konumunu saklar (0)

dosya tanımlayıcısını uygulamaya döndür

ARA isteğini al kök dizinde "foo "yu ara foo'nun FH + özniteliklerini döndür

**read(fd, buffer, MAX);**

fd ile açık dosya tablosunda dizin NFS dosya tanıtıcısı (FH) alır

geçerli dosya konumunu ofset olarak kullan

READ gönder (FH, offset=0, count=MAX)

OKU yanıtını alın

dosya konumunu güncelle (+bayt oku) geçerli dosya konumunu ayarla = MAX uygulamaya veri/hata kodu döndür

READ isteği alma

birim/inode numarasını almak için FH kullanın diskten (veya önbellekten) inode okuyun blok konumunu hesaplayın (ofseti kullanarak) diskten (veya önbellekten) veri okuyun

verileri müşteriye döndürür

**read(fd, buffer, MAX);**

Ofset=MAX dışında aynıdır ve geçerli dosya konumunu = 2\*MAX olarak ayarlar

**read(fd, buffer, MAX);**

Ofset=2\*MAX ve geçerli dosya konumunu = 3\*MAX olarak ayarlamak dışında aynı

**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

IPUCU: ETKİSİZLİL GÜÇLÜDÜR

Etkisizlik **(Idempotency)**, güvenilir sistemler oluştururken yararlı 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.

## Etkisiz 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 bu nedenle ya istek ya da cevap kaybolabilir ve böylece istemci asla bir yanıt alamaz.

Sunucunun çökmüş 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 vermediğinde 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 kapanmak üzere 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: etkisizdirler **(idempotent)**.Bir işlemin birden fazla kez gerçekleştirilmesinin etkisi, işlemin tek bir kez gerçekleştirilmesinin etkisine eşdeğer olduğunda, bu işleme idempotent denir. Örneğin, bir değeri bir bellek konumuna üç kez depolarsanız, bunu bir kez yapmakla aynı şeydir; bu nedenle "değeri belleğe depola" bir idempotent işlemdir. Bununla birlikte, bir sayacı üç kez artırırsanız, bunu yalnızca bir kez yapmaktan farklı bir miktarla sonuçlanır; bu nedenle, "sayacı artır" idempotent değildir. Daha genel olarak, sadece veri okuyan herhangi bir işlem açık bir şekilde 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, dosya sunucusundan sadece bilgi okudukları ve güncelleme yapmadıkları için önemsiz bir şekilde idempotenttir. Daha da ilginci, WRITE istekleri de idempotenttir. Örneğin, bir WRITE başarısız olursa, istemci basitçe yeniden deneyebilir. WRITE mesajı veri, sayı ve (daha da önemlisi) verinin yazılacağı tam ofseti içerir. Böylece, birden fazla yazmanın sonucunun tek bir yazmanın sonucuyla aynı olduğu bilinerek tekrarlanabilir.

Client

[send request]

## Case 1: Request Lost



Server

(no mesg)

Client

[send request]

## Case 2: Server Down

Server

(down)

## Case 3: Reply lost on way back from Server

Client

[send request]

Server

[recv request] [handle request] [send reply]



**Şekil 49.6: Üç Tip Kayıp**

Bu şekilde, istemci tüm zaman aşımlarını birleşik bir şekilde ele alabilir. Eğer bir WRITE isteği basitçe kaybolursa (yukarıdaki Durum 1), istemci isteği yeniden deneyecek, 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 istenildiği gibi çalışması durumunda da gerçekleşecektir (Durum 2). Son olarak, sunucu aslında WRITE isteğini alabilir, diskine yazma işlemini gerçekleştirebilir ve bir yanıt gönderebilir. Bu yanıt kaybolabilir (Durum 3) ve 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 ek: bazı işlemleri idempotent yapmak 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 başarıyla yürütürse, ancak yanıt kaybolursa, istemci işlemi tekrarlayabilir ve aslında işlem ilk başta başarılı olmuş ve daha sonra yalnızca yeniden denemede başarısız olmuşsa bu başarısızlıkla karşılaşabilir. Dolayısıyla, hayat mükemmel değildir.

İPUCU: MÜKEMMEL İYİNİN DÜŞMANIDIR (VOLTAİRE YASASI)

Güzel bir sistem tasarladığınızda bile, bazen tüm köşe durumları tam olarak istediğiniz gibi çalışmaz. Yukarıdaki mkdir örneğini ele alalım; mkdir farklı anlamlara sahip olacak şekilde yeniden tasarlanabilir, böylece idempotent hale getirilebilir (bunu nasıl yapabileceğinizi düşünün); ancak, neden uğraşasınız ki? NFS tasarım felsefesi önemli durumların çoğunu kapsar ve genel olarak sistem tasarımını başarısızlık açısından temiz ve basit hale getirir. Dolayısıyla, hayatın mükemmel olmadığını kabul etmek ve yine de sistemi inşa etmek iyi mühendisliğin bir işaretidir. Görünüşe göre, bu bilgelik Voltaire'e atfedilir, çünkü "... bilge bir İtalyan en iyinin düşmanı olduğunu söyler" [V72] ve bu nedenle buna Voltaire Kanunu (**Voltaire’s Law)** diyoruz.

## Performansı İyileştirme: İstemci Tarafı Önbelleğe Alma

Dağıtık 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 diskle karşılaştırıldığında. Dolayısıyla, başka bir sorun: dağıtılmış bir dosya sisteminin performansını nasıl artırabiliriz?

Cevap, yukarıdaki alt başlıktaki büyük kalın kelimeleri okuyarak tahmin edebileceğiniz gibi, istemci tarafı önbelleğe almadır **(caching)**. 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 servis edilir.

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) tamponlar. Bu tür bir yazma tamponlaması **(buffering)** kullanışlıdır çünkü uygulama write() gecikmesini gerçek yazma performansından ayırır, yani uygulamanın write() çağrısı hemen başarılı olur (ve verileri istemci tarafı dosya sisteminin önbelleğine koyar); veriler ancak daha sonra dosya sunucusuna yazılır.

Böylece, 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 tutarlılığı sorunu **(cache consistency problem)** olarak adlandıracağımız büyük ve ilginç bir zorluk ortaya çıkarır.

## Önbellek Tutarlılık Problemi

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 istemcinin, C2'nin, F dosyasının üzerine yazdığını ve böylece içeriğini değiştirdiğini düşünün; F dosyasının yeni sürümüne

C1

önbellek: F[v1]

C2

önbellek: F[v2]

C3

önbellek: boş

Sunucu S Disk: İlk başta F[v1]

F[v2] sonunda

Figure 49.7: **The Cache Consistency Problem**

(sürüm 2) veya F[v2] ve eski sürümüne F[v1] diyelim, böylece ikisini ayrı tutabiliriz (ancak elbette dosya aynı ada sahiptir, sadece farklı içeriklere sahiptir). Son olarak, F dosyasına henüz erişmemiş olan üçüncü bir istemci, C3, vardır.

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 tamponlayarak, 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, ancak eski kopyayı aldığınızı hayal edin! 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 bayat 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 üzerinde çalışan bir program F dosyasını okursa, en son kopyayı (F[v2]) değil, eski bir sürümü (F[v1]) alır, bu da (genellikle) istenmeyen bir durumdur.

NFSv2 uygulamaları bu önbellek tutarlılığı sorunlarını iki şekilde çözer. İlk olarak, güncelleme görünürlüğünü ele almak için, istemciler bazı zamanlar **flush-on-close** (a.k.a., **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 açılan bir alt dosyanın en son dosya sürümünü görmesini sağlar.

İkinci olarak, bayat önbellek sorununu ele almak için, NFSv2 istemcileri önbellekteki 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 **invalidates**, 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 eder ve önbelleğe alınan içeriği kullanır, böylece performansı artırır.

Sun'daki orijinal ekip stale- cache sorununa bu çözümü uyguladığında, yeni bir sorunun farkına vardılar; aniden NFS sunucusu GETATTR istekleri ile doldu. İzlenmesi gereken iyi bir mühendislik ilkesi, genel durum **common case** için tasarım yapmak ve bunun iyi çalışmasını sağlamaktır; burada, genel durum bir dosyaya yalnızca tek bir istemciden (belki de tekrar tekrar) erişilmesi olsa da, istemci dosyayı başka kimsenin değiştirmediğinden emin olmak için sunucuya her zaman GETATTR istekleri göndermek zorundaydı. Böylece bir istemci sunucuyu bombardımana tutarak sürekli "bu dosyayı kimse değiştirdi mi?" diye sorar, oysa çoğu zaman kimse değiştirmemiştir.

Bu durumu (biraz) düzeltmek için her istemciye bir öznitelik önbelleği **attribute cache** eklendi. Bir istemci bir dosyaya erişmeden önce yine de dosyayı doğrulayacak, ancak çoğunlukla öznitelikleri almak için öznitelik önbelleğine bakacaktı. Belirli bir dosyanın öznitelikleri, dosyaya ilk erişildiğinde önbelleğe yerleştiriliyor ve belirli bir süre sonra (örneğin 3 saniye) zaman aşımına uğruyordu. Böylece, bu üç saniye boyunca, tüm dosya erişimleri önbellekteki dosyayı kullanmanın uygun olduğunu belirleyecek ve böylece sunucu ile ağ iletişimi olmadan bunu yapacaktı.

## 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 oluşturulup kısa süre sonra silinirse, bu dosya sunucuya zorla gönderilmeye devam eder. 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, zaman garip davranışlara yol açardı (ve hala da açıyor!).

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

## 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 bunları 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 (örneğin diske ya da başka bir kalıcı aygıta) zorlanana kadar bir WRITE protokol isteğinde kesinlikle başarı döndürmeyebilir. Verilerin bir kopyasını sunucu belleğine yerleştirebilseler de, bir WRITE protokol 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:

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

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.

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb cccccccccccccccccccccccccccccccccccccccccccc

Şimdi örneğin hatırına, bu üç istemci yazma işleminin sunucuya üç farklı 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 sadece bellekte tamponlandığını ve sunucunun diske göndermeden 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:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops cccccccccccccccccccccccccccccccccccccccccccc

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 KENARA: YENILIK YENILIĞI 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 (VFS) / Sanal Düğüm (vnode) **Virtual File System** (**VFS**) / **Virtual Node** (**vnode**) arayüzüdür [K86].

VFS katmanı, takma ve çıkarma, dosya sistemi çapında istatistikler alma ve tüm kirli (henüz yazılmamış) yazmaları diske zorlama gibi tüm dosya sistemine yapılan işlemleri içerir. 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 ve hatta Windows (Yüklenebilir Dosya Sistemi şeklinde) dahil olmak üzere birçok modern sistem bir çeşit VFS/vnode katmanına sahiptir. 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 kon- tentlerinin asla birbirine karışmamasını sağlar.

Bu gereksinimin NFS sunucusu uygulamasında ortaya çıkardığı sorun, yazma performansının büyük bir özen gösterilmediği takdirde en önemli performans darboğazı olabilmesidir. Gerçekten de bazı şirketler (örneğin Network Appliance) yazma işlemlerini hızlı bir şekilde gerçekleştirebilen bir NFS sunucusu inşa etmek gibi basit bir amaçla ortaya çıkmıştır; kullandıkları hilelerden biri yazma işlemlerini önce pil destekli bir belleğe yerleştirmektir, böylece veriyi kaybetme korkusu olmadan ve diske hemen yazma maliyeti olmadan WRITE isteklerine hızlı bir şekilde yanıt vermek mümkün olur; ikinci hile ise nihayet ihtiyaç duyulduğunda diske hızlı bir şekilde yazmak için özel olarak tasarlanmış bir dosya sistemi tasarımı kullanmaktır [HLM94, RO91].

## Özet

NFS dağıtılmış dosya sisteminin tanıtıldığını gördük. NFS, sunucu arızası karşısında basit ve hızlı kurtarma fikri etrafında şekillenmiştir ve bu amaca dikkatli protokol tasarımı ile ulaşmaktadır.

AYRICA: ANAHTAR NFS TERİMLERİ

* + - 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 **idempotent** bir kez gerçekleştirilmesine eşdeğer olduğunda, o işlem idempotenttir. 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 tamponla-

ması **write buffering**, ancak bir önbellek tutarlılığı sorunu **cache consistency problem** ortaya çıkarır.

* + - NFS uygulamaları, önbellek tutarlılığı için birden fazla yolla mühendislik çözümü sunar: flush-on-close (kapat-aç) **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 başarıya ulaşmadan önce kalıcı ortama yazma işlemi gerçekleştirmelidir; aksi takdirde veri kaybı ortaya çıkabilir.
    - Sun, NFS'nin işletim sistemine entegrasyonunu desteklemek için VFS/Vnode arayüzünü geliştirerek birden fazla dosya sistemi uygulamasının aynı işletim sisteminde bir arada var olmasını sağladı.

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.

Ayrıca çok istemcili, tek sunuculu bir sisteme önbelleğin eklenmesinin işleri nasıl karmaşıklaştırabileceğini gördük. Özellikle, sistemin gerçekçi bir şekilde davranabilmesi için önbellek tutarlılığı sorununu çözmesi gerekir; ancak NFS bunu biraz geçici bir şekilde yapar ve bu da zaman gözlemlenebilir garip davranışlara neden olabilir. Son olarak, sunucu önbelleğinin nasıl aldatıcı olabileceğini gördük: sunucuya yapılan yazmalar başarıya dönmeden önce sabit depolamaya zorlanmalıdır (aksi takdirde veri 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 hemen 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 Neu- man, 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 some- time...*

[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 dis- cussion 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 Rosen- blum, 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.](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?*

# Ev Ödevi (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 tar topunu 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!