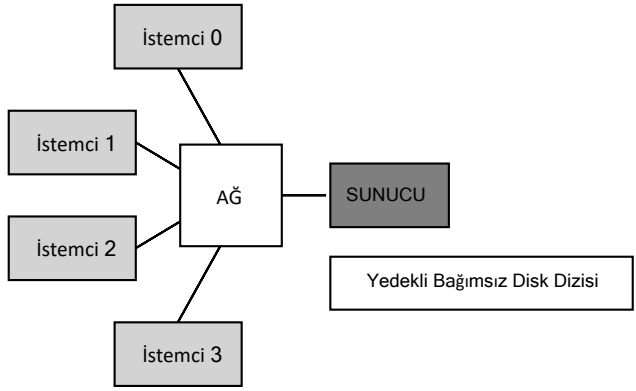
49

Sun'ın Ağ Dosya Sistemi (NFS)

Dağıtılmış istemci/sunucu bilgi işlemin ilk kullanımlarından biri, dağıtılmış dosya sistemleri alanındaydı. Böyle bir ortamda, birkaç istemci makine ve bir sunucu (veya birkaçı) vardır; sunucu verileri disklerinde depolar ve istemciler iyi biçimlendirilmiş protokol iletileri aracılığıyla veri ister. Şekil 49.1'de temel kurulum gösterilmektedir.



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

Resimden de görebileceğiniz gibi, sunucuda diskler vardır ve istemciler bu disklerdeki dizinlerine ve dosyalarına erişmek için bir ağ üzerinden ileti gönderir. Neden bu düzenlemeyle uğraşıyoruz? (yani, neden istemcilerin yerel disklerini kullanmalarına izin vermiyoruz?) Öncelikle, bu kurulum, istemciler arasında verilerin kolayca **paylaşılmasını (sharing)** sağlar. Bu nedenle, bir makinedeki bir dosyaya erişirseniz (İstemci 0) ve daha sonra başka bir dosya kullanırsanız (İstemci 2), dosya sisteminin aynı görünümüne sahip olursunuz. Verileriniz doğal olarak bu farklı makineler arasında paylaşılır. İkincil bir fayda **merkezi yönetimdir (centralized administration)**; örneğin, dosyaların yedeklenmesi çok sayıda istemci yerine birkaç sunucu makinesinden yapılabilir. Diğer bir avantaj **güvenlik (security)** olabilir; Tüm sunucuların kilitli bir makine odasında olması, belirli türdeki sorunların ortaya çıkmasını önler.

PÜF NOKTASI: Dağıtılmış Bir Dosya Sistemi Nasıl Oluşturulur

Dağıtılmış bir dosya sistemini nasıl oluşturursunuz? Düşünülmesi gereken temel hususlar nelerdir? Yanlış anlaşılması kolay olan nedir? Mevcut sistemlerden ne öğrenebiliriz?

49.1 Temel Dağıtılmış Dosya Sistemi

Şimdi basitleştirilmiş bir dağıtılmış dosya sisteminin mimarisini inceleyeceğiz. Basit bir istemci/sunucu dağıtılmış dosya sistemi, şimdiye kadar incelediğimiz dosya sistemlerinden daha fazla bileşene sahiptir. İstemci tarafında, **istemci tarafı dosya sistemi (clientside file system)** üzerinden dosyalara ve dizinlere erişen istemci uygulamaları vardır. Bir istemci uygulaması, sunucuda depolanan dosyalara erişmek için istemci tarafı dosya sistemine (`open()`, `read()`, `write()`, `close()`, `mkdir()`, `vb.`) **sistem çağrıları (system calls)** yapar. Bu nedenle, istemci uygulamaları için, dosya sistemi, belki de performans dışında, yerel (disk tabanlı) bir dosya sisteminden farklı görünmemektedir; Bu şekilde, dağıtılmış dosya sistemleri, açık bir amaç olan dosyalara **şeffaf (transparent)** erişim sağlar; Sonuçta, farklı bir API seti gerektiren veya başka bir şekilde kullanılması zor olan bir dosya sistemini kim kullanmak ister?

İstemci tarafı dosya sisteminin rolü, bu sistem çağrılarına hizmet vermek için gereken eylemleri yürütmektir. Örneğin, istemci bir `read()` isteği gönderirse, istemci tarafı dosya sistemi **sunucu tarafı dosya sistemine (server-side file system)** (veya yaygın olarak adlandırıldığı gibi **dosya sunucusuna (file server)**) belirli bir bloğu okumak için bir ileti gönderebilir; dosya sunucusu daha sonra bloğu diskten (veya kendi bellek içi önbelleğinden) okuyacak ve istenen verileri içeren istemciye bir ileti gönderecektir. İstemci tarafı dosya sistemi daha sonra verileri `read()` sistem çağrısına sağlanan kullanıcı arabelleğine kopyalar ve böylece istek tamamlanır. İstemcideki aynı bloğun sonraki bir `read()` öğesinin istemci belleğinde veya hatta istemcinin diskinde **önbelleğe (cached)** alınabileceğini unutmayın; böyle bir 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 olduğu hissine kapılmalısınız: istemci tarafı dosya sistemi ve dosya sunucusu. Davranışları birlikte dağıtılmış dosya sisteminin davranışını belirler. Şimdi belirli bir sistemi incelemenin zamanı geldi: Sun'ın Ağ Dosya Sistemi (NFS).

BİR KENARA: SUNUCULAR NEDEN ÇÖKÜYOR

NFSv2 protokolünün ayrıntılarına girmeden önce, merak ediyor olabilirsiniz: sunucular neden çöküyor? Tahmin edebileceğiniz gibi, bunun birçok nedeni var. Sunucular sadece **elektrik kesintisinden (power outage)** muzdarip olabilir (geçici olarak); yalnızca güç geri yüklendiğinde 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 tetikleyeceklerdir. Ayrıca hafıza sızıntıları var; küçük bir bellek sızıntısı bile sistemin belleğinin tükenmesine ve çökmesine neden olur. Ve son olarak, dağıtılmış sistemlerde, istemci ve sunucu arasında bir ağ vardır; ağ garip davranırsa (örneğin, **bölümlenirse (partitioned)** ve istemciler ve sunucular çalışıyor ancak iletişim kuramıyorsa), uzak bir makine çökmüş gibi görünebilir, ancak gerçekte şu anda ağ üzerinden erişilemez.

49.2 NFS'de

En eski ve oldukça başarılı dağıtılmış sistemlerden biri Sun Microsystems tarafından geliştirilmiştir ve Sun Ağ Dosya Sistemi (veya NFS) [S86] olarak bilinir. NFS'yi tanımlarken, Sun alışılmadık bir yaklaşım benimsedi: tescilli ve kapalı bir sistem oluşturmak yerine, Sun bunun yerine istemcilerin ve sunucuların iletişim kurmak için kullanacakları tam mesaj formatlarını belirten **açık protokol (open protocol)** geliştirdi. Farklı gruplar kendi NFS sunucularını geliştirebilir ve böylece birlikte çalışabilirliği korurken bir NFS pazarında rekabet edebilir. 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 atfediliyor.

49.3 Odak: Basit ve Hızlı Sunucu Çökmesi Kurtarma

Bu bölümde klasik NFS protokolünü tartışacağız (sürüm 2, uzun yıllardır standart olan NFSv2); nfsv3'e geçişte küçük değişiklikler yapıldı ve Nfsv4'e geçişte daha büyük ölçekli protokol değişiklikleri yapıldı. Bununla birlikte, NFSv2 hem harika hem de sinir bozucudur ve bu nedenle odak noktamız olarak hizmet eder.

NFSv2'de, protokolün tasarımındaki ana amaç basit ve hızlı sunucu çökmesi kurtarmasıydı. Çok istemcili, tek sunuculu bir ortamda, bu hedef çok mantıklıdır; sunucunun kapalı olduğu (veya kullanılamadığı) herhangi bir dakika, tüm istemci makinelerini (ve kullanıcılarını) mutsuz ve verimsiz hale getirir. Böylece, sunucu gittikçe, tüm sistem de gider.

49.4 Hızlı Çökme Kurtarmanın Anahtarı: Durumsuzluk

Bu basit hedef, NFSv2'de **durum bilgisi olmayan (stateless)** bir protokol olarak adlandırdığımız şeyi tasarlayarak gerçekleştirilir. Sunucu, tasarım gereği, her istemcide neler olup bittiği hakkında 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 izlemez; bunun yerine, protokol her protokol isteğinde isteği tamamlamak için gereken tüm bilgileri teslim etmek üzere tasarlanmıştır. Şimdi değilse, bu durum bilgisi olmayan yaklaşım, protokolü aşağıda daha ayrıntılı olarak tartıştığımız için daha mantıklı olacaktır.

Durum bilgisi olan (stateful) (durum bilgisi olmayan) bir protokol örneği için, `open()` sistem çağrısını göz önünde bulundurun. Bir yol adı verildiğinde, `open()` bir dosya tanımlayıcısı (bir tamsayı) döndürür. Bu tanımlayıcı, bu uygulama kodunda olduğu gibi çeşitli dosya bloklarına erişmek için sonraki `read()` veya `write()` isteklerinde kullanılır (sistem çağrılarının uygun hata denetiminin boşluk nedenleriyle atlandığını 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 tarafı dosya sisteminin, sunucuya "'foo' dosyasını aç ve bana bir tanımlayıcı geri ver" diyen bir protokol mesajı göndererek dosyayı açtığını hayal edin. Dosya sunucusu daha sonra dosyayı kendi tarafında yerel olarak açar ve tanımlayıcıyı istemciye geri gönderir. Sonraki okumalarda, istemci uygulaması `read()` sistem çağrısını çağırmak için bu tanımlayıcıyı kullanır; istemci tarafı dosya sistemi daha sonra tanımlayıcıyı bir iletiyle dosya sunucusuna geçirir ve "tanımlayıcı tarafından başvurulan dosyadan bazı baytları okuyun sizi buraya geçiriyorum" der.

Bu örnekte, dosya tanımlayıcısı istemci ile sunucu arasındaki **paylaşılan durumun (shared state**) bir parçasıdır (Çıkış bu **dağıtılmış durumu (distributed state**) [O91] olarak adlandırır). Paylaşılan durum, yukarıda ima ettiğimiz gibi, kilitlenme kurtarma işlemini zorlaştırır. İlk okuma tamamlandıktan sonra, ancak istemci ikincisini yayınlamadan önce sunucunun çöktüğünü düşünün. Sunucu yeniden çalışmaya başladıktan sonra, istemci ikinci okumayı yayınlar. Ne yazık ki, sunucunun fd'nin hangi dosyaya başvurduğu hakkında hiçbir fikri yoktur; Bu bilgi geçiciydi (yani bellekte) ve böylece sunucu çöktüğünde kayboldu. Bu durumu ele almak için, istemcinin ve sunucunun, sunucuya bilmesi gerekenleri söyleyebilmek için belleğinde yeterli bilgi tuttuğundan emin olacağı bir tür **kurtarma protokolüne (recovery protocol)** girmesi gerekir (bu durumda, bu dosya tanımlayıcısı fd dosya foo'sunu ifade eder.)

Durum bilgisi olan bir sunucunun istemci çökmeleriyle uğraşması gerektiğini düşündüğünüzde daha da kötüleşir. Örneğin, bir dosyayı açan ve sonra çöken bir istemci düşünün. `open()` sunucuda bir dosya tanımlayıcısı kullanır; sunucu belirli bir dosyayı kapatmanın uygun olduğunu nasıl bilebilir? Normal çalışmada, bir istemci sonunda `close()` öğesini çağırır ve böylece sunucuya dosyanın kapatılması gerektiğini bildirir. Ancak, bir istemci çöktüğünde, sunucu hiçbir zaman `close()` almaz ve bu nedenle dosyayı kapatmak için istemcinin çöktüğünü fark etmesi gerekir.

Bu nedenlerden dolayı, NFS tasarımcıları durum bilgisi olmayan bir yaklaşım izlemeye karar verdiler: her istemci işlemi isteği tamamlamak için gereken tüm bilgileri içerir. Süslü bir çarpışma kurtarmaya gerek yoktur; sunucu yeniden çalışmaya başlar ve en kötü ihtimalle bir istemcinin bir isteği yeniden denemesi gerekebilir.

49.5 NFSv2 Protokolü

Böylece NFSv2 protokol tanımına ulaşmış oluruz. Sorun ifademiz basittir:

> **Püf Nokta: Durum Bilgisi Olmayan Bir Dosya Protokolü Nasıl Tanımlanır**
>
> Durum bilgisi olmayan çalışmayı etkinleştirmek için ağ protokolünü nasıl tanımlayabiliriz? Açıkçası, `open()` gibi durum bilgisi olan çağrılar tartışmanın bir parçası olamaz (sunucunun açık dosyaları izlemesini gerektireceğinden); ancak istemci uygulaması, dosyalara ve dizinlere erişmek için `open()`, `read()`, `write()`, `close()` ve diğer standart API çağrılarını çağırmak isteyecektir. Bu nedenle, rafine edilmiş bir soru olarak, protokolü hem durum bilgisi olmayan hem de POSIX dosya sistemi API'sini destekleyecek şekilde nasıl tanımlarız?

NFS protokolünün tasarımını anlamanın bir anahtarı, **dosya tanıtıcısının (file handle)** altında durmaktır. 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 bileşene sahip olduğunu düşünebilirsiniz: birim tanımlayıcısı, düğüm numarası ve nesil numarası; Bu üç öğe birlikte, istemcinin erişmek istediği bir dosya veya dizin için benzersiz bir tanımlayıcı oluşturur. Birim tanımlayıcısı, sunucuya isteğin hangi dosya sistemine başvurduğunu bildirir (NFS sunucusu birden fazla dosya sistemini dışa aktarabilir); düğüm numarası, sunucuya isteğin bu bölümdeki hangi dosyaya eriştiğini söyler. Son olarak, bir düğüm numarasını yeniden kullanırken nesil numarasına ihtiyaç vardır; sunucu, bir düğüm numarası yeniden kullanıldığında bunu artırarak, eski bir dosya tanıtıcısına sahip bir istemcinin yeni ayrılan dosyaya yanlışlıkla erişememesini sağlar..

İşte protokolün bazı önemli parçalarının bir özeti; tam protokol 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, LOOKUP protokolü iletisi bir dosya tanıtıcısı elde etmek için kullanılır ve daha sonra dosya verilerine erişmek için kullanılır. İstemci, aramak için bir dizin dosyası tanıtıcısını ve dosyanın adını geçirir ve bu dosyaya (veya dizine) tanıtıcı ve öznitelikleri sunucudan istemciye geri gönderilir.

Örneğin, istemcinin bir dosya sisteminin (/) kök dizini için zaten bir dizin dosyası tanıtıcısına sahip olduğunu varsayalım (aslında, bu, istemcilerin ve sunucuların ilk önce birbirine nasıl bağlandığı NFS **bağlama protokolü (mount protocol**) aracılığıyla elde edilir; burada bağlama protokolünü kısalık uğruna tartışmıyoruz). İstemcide çalışan bir uygulama /foo.txt dosyasını açarsa, istemci tarafı dosya sistemi sunucuya kök dosya tanıtıcısını ve foo.txt adını ileten bir arama isteği gönderir; başarılı olursa, foo.txt için dosya tanıtıcısı (ve öznitelikler) döndürülür.

Merak ediyorsanız, öznitelikler, dosya oluşturma zamanı, son değişiklik zamanı, boyut, sahiplik ve izin bilgileri gibi alanlar da dahil olmak üzere dosya sisteminin her dosya hakkında izlediği meta verilerdir, yani bir dosyada `stat()` olarak adlandırdığınızda geri alacağınız aynı bilgi türüdür.

Bir dosya tanıtıcısı kullanılabilir olduğunda, istemci dosyayı okumak veya yazmak için bir dosyada sırasıyla READ ve WRITE protokol iletileri verebilir. READ protokolü iletisi, protokolün dosya içindeki uzaklık ve okunacak bayt sayısıyla birlikte dosyanın dosya tanıtıcısını iletmesini gerektirir. Sunucu daha sonra okumayı yayınlayabilir (sonuçta, tanıtıcı sunucuya hangi birimden ve hangi düğümden okunacağını söyler ve uzaklık ve sayım, dosyanın hangi baytlarının okunacağını söyler) ve verileri istemciye (veya bir hata varsa bir hataya) döndürebilir. WRITE, veriler istemciden sunucuya geçirilmediği ve yalnızca bir başarı kodu döndürüldüğü sürece benzer şekilde işlenir.

Son bir ilginç protokol mesajı GETATTR isteğidir; bir dosya tanıtıcısı verildiğinde, dosyanın son değiştirilme zamanı da dahil olmak üzere bu dosyanın özniteliklerini getirmesi yeterlidir. Bu protokol isteğinin neden NFSv2'de etkili olduğunu aşağıda önbelleğe almayı tartıştığımızda göreceğiz (nedenini tahmin edebiliyor musunuz?).

49.6 Protokolden Dağıtılmış Dosya Sistemine

Umarım şimdi bu protokolün istemci tarafı dosya sistemi ve dosya sunucusunda bir dosya sistemine nasıl dönüştürüldüğüne dair bir fikir ediniyorsunuzdur. İstemci tarafı dosya sistemi açık dosyaları izler ve genellikle uygulama isteklerini ilgili protokol iletileri kümesine dönüştürür. Sunucu, her biri isteği tamamlamak için gereken tüm bilgileri içeren protokol iletilerine 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 verirken ne yaptığını gösteriyoruz.

Şekil hakkında birkaç yorum. İlk olarak, istemcinin tamsayı dosya tanımlayıcısının bir NFS dosya tanıtıcısına ve geçerli dosya işaretçisine eşlenmesi de dahil olmak üzere dosya erişimi için ilgili tüm **durumu (state)** nasıl izlediğine dikkat edin. Bu, istemcinin her okuma isteğini (açıkça okunacak uzaklığı belirtmediğini fark etmiş olabilirsiniz), sunucuya dosyadan tam olarak hangi baytların okunacağını bildiren düzgün biçimlendirilmiş bir okuma protokolü iletisine dönüştürmesini sağlar. Başarılı bir okuma üzerine, istemci geçerli dosya konumunu günceller; sonraki okumalar aynı dosya tanıtıcısıyla, ancak farklı bir uzaklıkla verilir.

İkincisi, sunucu etkileşimlerinin nerede gerçekleştiğini fark edebilirsiniz. Dosya ilk kez açıldığında, istemci tarafı dosya sistemi bir LOOKUP istek iletisi gönderir. Gerçekten de, uzun bir yol adının geçilmesi gerekiyorsa (örneğin; `/home/remzi/foo.txt`), müşteri 3 tane LOOKUPs gönderir: bir tane LOOKUP `home` dizinine `/`, bir tane LOOKUP `remzi` in `home`, ve son LOOKUP `foo.txt` in `remzi`.

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

Alıcı	Sunucu
fd = open("/foo", ...);	
LOOKUP Gönder (rootdir FH, "foo")	LOOKUP isteğini alın kök dizinde "foo" arayın ve foo'nun FH + özniteliklerini döndürün
LOOKUP yanıtı al açık dosya tablosunda desc dosyasını ayırma foo'nun FH'ını masada sakla geçerli dosya konumunu sakla (0) dosya tanımlayıcısını uygulamaya döndür	
read(fd, buffer, MAX);	
fd ile açık dosya tablosuna dizin oluşturma NFS dosya tanıtıcısını edinin (FH) geçerli dosya konumunu offset olarak kullan	
READ Gönder (FH, offset=0, count=MAX)	READ isteği alın hacim/inode numarası almak için FH kullanın inode'u diskten (veya önbellekten) okuyun
OKUYUN yanıtı alın güncelleme dosyası konumu (+bayt okuma) geçerli dosya konumunu ayarla = MAX veri/hata kodunu uygulamaya döndürme	hesaplama bloğu konumu (offset kullanarak) diskten (veya önbellekten) veri okuma verileri istemciye döndürme
read(fd, buffer, MAX);	
Ofset hariç aynı = MAX ve geçerli dosya konumunu ayarlama = 2*MAX	
read(fd, buffer, MAX);	
Ofset hariç aynı =2 * MAX ve geçerli dosya konumunu ayarlayın = 3*MAX	
close(fd);	
Sadece yerel yapıları temizlemeniz gerekiyor Açık dosya tablosunda ücretsiz tanımlayıcı "fd" (Sunucu ile konuşmaya gerek yok)	

Figure 49.5: **Dosya Okuma: İstemci Tarafı ve Dosya Sunucusu Eylemleri**

İPUCU: AYNI GÜÇ GÜÇLÜDÜR

Aynı Güç (Idempotency), güvenilir sistemler oluştururken yararlı bir özelliktir. Bir işlem birden fazla kez verilebildiğinde, işlemin başarısızlığını ele almak çok daha kolaydır; sadece yeniden deneyebilirsiniz. Bir operasyon aynı güç etkili değilse, hayat daha da zorlaşır.

49.7 Aynı Güç İşlemlerle Sunucu Hatasını İşleme

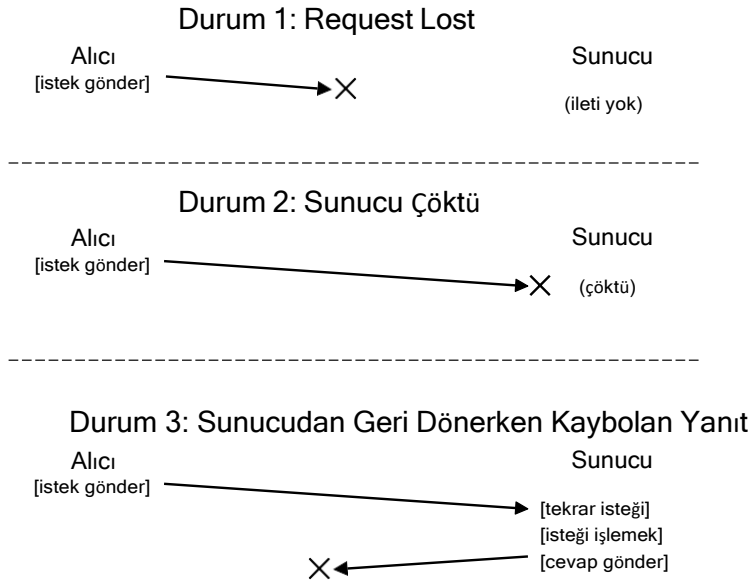
İstemci sunucuya bir ileti gönderdiğinde, bazen bir yanıtı yeniden almaz. Bu yanıt vermemenin birçok olası nedeni vardır. Bazı durumlarda, ileti ağ tarafından bırakılabilir; ağlar mesajları kaybeder ve bu nedenle istek veya yanıt kaybolabilir ve böylece istemci asla yanıt alamaz.

Sunucunun çökmüş olması ve bu nedenle şu anda iletilere 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 karşı karşıya kalır: sunucu zamanında yanıt vermediğinde ne yapmalılar?

NFSv2'de, istemci tüm bu hataları tek, tek tip ve zarif bir şekilde ele alır: yalnızca isteği yeniden dener. Özellikle, isteği gönderdikten sonra, istemci belirli bir süre sonra kapanacak 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, müşteri isteğin işlenmediğini varsayar ve yeniden gönderir. Sunucu yanıt verirse, her şey yolunda demektir ve istemci sorunu düzgün bir şekilde ele almıştır.

İstemcinin isteği basitçe yeniden deneme yeteneği (başarısızlığa neyin neden olduğuna bakılmaksızın), çoğu NFS isteğinin önemli bir özelliğinden kaynaklanmaktadır: bunlar **Aynı Güç (Idempotenttir).** İşlemi birden çok kez gerçekleştirmenin etkisi, işlemi tek bir kez gerçekleştirmenin etkisine eşdeğer olduğunda, işleme idempotent denir. Örneğin, bir değeri bir bellek konumuna üç kez depolarsanız, bunu bir kez yapmakla aynıdır; bu nedenle "değeri belleğe depolamak" idempotent bir işlemdir. Bununla birlikte, bir sayacı üç kez sokarsanız, bunu yalnızca bir kez yapmaktan farklı bir miktarla sonuçlanır; bu nedenle, "artış sayacı" idempotent değildir. Daha genel olarak, sadece verileri okuyan herhangi bir işlem açıkça idempotenttir; verileri güncelleyen bir işletme, bu özelliğe sahip olup olmadığını belirlemek için daha dikkatli düşünülmelidir.

NFS'de çarpışma kurtarma tasarımının kalbi, en yaygın operasyonların idempotensidir. LOOKUP ve READ istekleri, yalnızca dosya sunucusundaki bilgileri okudukları ve güncelleştirmedikleri için önemsiz derecede idempotenttir. Daha da ilginci, WRITE istekleri de idempotenttir. Örneğin, bir WRITE başarısız olursa, istemci bunu yeniden deneyebilir. WRITE iletisi verileri, sayımı ve (daha da önemlisi) verilerin yazılacağı tam offseti içerir. Böylece, birden fazla yazmanın sonucunun tek bir yazının sonucuyla aynı olduğu bilgisiyle tekrarlanabilir.



Şekil 49.6: **3 Tür Kayıp**

Bu şekilde, istemci tüm zaman aşımlarını birleşik bir şekilde işleyebilir. Bir WRITE isteği basitçe kaybolduysa (yukarıdaki Durum 1), istemci yeniden deneyecek, sunucu yazmayı gerçekleştirecek ve her şey yolunda gidecektir. Aynı şey, istek gönderilirken sunucu kapalıysa, ancak ikinci istek gönderildiğinde yedeklenip çalışıyorsa ve yine hepsi istenildiği gibi çalışıyorsa (Durum 2) gerçekleşir. Son olarak, sunucu aslında WRITE isteğini alabilir, diskine yazma işlemini yayınlayabilir ve bir yanıt gönderebilir. Bu yanıt kaybolabilir (Durum 3), istemcinin isteği yeniden göndermesine neden olabilir. Sunucu isteği tekrar aldığında, aynı şeyi yapacaktır: verileri diske yazın ve bunu yaptığını yanıtlayın. İstemci bu kez yanıtı alırsa, her şey yine yolunda demektir ve böylece istemci hem ileti kaybını hem de sunucu hatasını tek tip bir şekilde ele almıştır. Temiz!

Küçük bir kenara: Bazı işlemleri idempotent hale getirmek zordur. Örneğin, zaten var olan bir dizin oluşturmaya çalıştığınızda, mkdir isteğinin başarısız olduğu bildirilir. Bu nedenle, NFS'de, dosya sunucusu bir MKDIR protokol iletisi alır ve başarıyla yürütürse ancak yanıt kaybolursa, istemci bunu tekrarlayabilir ve aslında işlem ilk başta başarılı olduğunda ve daha sonra yalnızca yeniden denemede başarısız olduğunda bu hatayla karşılaşabilir. Bu nedenle, hayat mükemmel değildir.

İPUCU: MÜKEMMEL İYİ'NİN DÜŞMANIDIR (VOLTAIRE YASASI)
Güzel bir sistem tasarlarken bile, bazen tüm köşe durumları tam olarak istediğiniz gibi çalışmaz. Yukarıdaki mkdir örneğini ele alalım; biri mkdir'i farklı semantiklere sahip olacak şekilde yeniden tasarlayabilir, böylece onu idempotent hale getirebilir (bunu nasıl yapabileceğinizi düşünün); ancak, neden rahatsız oluyorsunuz? 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 hala sistemi inşa etmek iyi bir mühendisliğin işaretidir. Görünüşe göre, bu bilgelik Voltaire'e atfedilir, çünkü "... bilge bir İtalyan, en iyinin iyinin düşmanı olduğunu söyler" [V72] ve biz buna **Voltaire Yasası (Voltaire's Law)** diyoruz.

49.8 Performansı Artırma: İstemci Tarafı Önbelleğe Alma

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

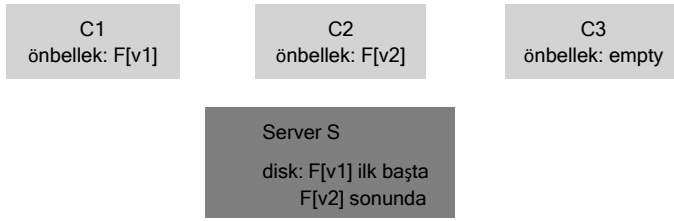
Cevap, yukarıdaki alt başlıktaki büyük kalın kelimeleri okuyarak tahmin edebileceğiniz gibi, istemci tarafı **önbelleğe (caching)** almadır. NFS istemci tarafı dosya sistemi, sunucudan okuduğu dosya verilerini (ve meta verileri) istemci belleğinden önbelleğe alır. Bu nedenle, ilk erişim pahalı olsa da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

Önbellek ayrıca yazmalar için geçici bir arabellek görevi görür. Bir istemci uygulaması bir dosyaya ilk kez yazdığında, istemci, verileri sunucuya yazmadan önce istemci üyesindeki verileri arabelleğe alır (dosya sunucusundan okuduğu verilerle aynı önbellekte). Bu tür **ara belleğe almayı yazma (write buffering**) yararlıdır, çünkü uygulama `write()` gecikmesini gerçek yazma performansından ayırır, yani uygulamanın `write()` çağrısı hemen başarılı olur (ve verileri istemci tarafı dosya sisteminin önbelleğine koyar); ancak daha sonra veriler dosya sunucusuna yazılır.

Bu nedenle, NFS istemcileri verileri önbelleğe alır ve performans genellikle harikadır ve işimiz bitti, değil mi? Ne yazık ki, tam olarak değil. Birden fazla istemci önbelleğine sahip herhangi bir sisteme önbelleğe alma eklemek, **önbellek tutarlılığı sorunu (cache consistency problem)** olarak adlandıracağımız büyük ve ilginç bir zorluk ortaya çıkarır.

49.9 Önbellek Tutarlılığı Sorunu

Önbellek tutarlılığı sorunu en iyi iki istemci ve tek bir sunucu ile gösterilir. C1 istemcisinin 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 hayal edin; F dosyasının yeni sürümünü arayalım



Şekil 49.7: **Önbellek Tutarlılığı Sorunu**

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

Muhtemelen yaklaşmakta olan sorunu görebilirsiniz (Şekil 49.7). Aslında, iki alt problem vardır. İlk alt sorun, C2 istemcisinin yazma işlemlerini sunucuya yaymadan önce bir süre önbelleğinde arabelleğe alabilmesidir; bu durumda, F[v2] C2'nin belleğinde otururken, başka bir istemciden (örneğin C3) F'ye herhangi bir erişim, dosyanın eski sürümünü (F[v1]) getirecektir. Bu nedenle, istemcideki yazmaları arabelleğe alarak, diğer istemciler dosyanın istenmeyen olabilecek eski sürümlerini alabilir; Gerçekten de, C2 makinesine giriş yaptığınızı, F'yi güncellediğinizi ve ardından C3'e giriş yaptığınızı ve yalnızca eski kopyayı almak için dosyayı okumaya çalıştığınızı hayal edin! Kesinlikle bu sinir bozucu olabilir. Bu nedenle, önbellek tutarlılığı sorununun bu yönüne **güncelleme görünürlüğü (update visibility)** diyelim; bir istemciden gelen güncelleştirmeler diğer istemcilerde ne zaman görünür hale gelir?

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

NFSv2 uygulamaları bu önbellek tutarlılığı sorunlarını iki şekilde çözer. İlk olarak, güncelleme görünürlüğünü ele almak için, istemciler bazen **flush-on-close (kapatırken yıkayın**) (diğer adıyla **close-to-open (açmak için kapat)**) tutarlılık semantiğini uygular; özellikle, bir dosya bir istemci uygulamasına yazıldığında ve daha sonra bir istemci uygulaması tarafından kapatıldığında, istemci tüm güncelleştirmeleri (yani, önbellekteki kirli sayfalar) sunucuya temizler. NFS, kapalı tutarlılık sayesinde, başka bir düğümden sonraki bir açılışın en son dosya sürümünü görmesini sağlar.

İkinci olarak, NFSv2 istemcileri eski önbellek sorununu gidermek için önce önbelleğe alınmış içeriğini kullanmadan önce bir dosyanın değişip değişmediğini denetler. Özellikle, önbelleğe alınmış bir bloğu kullanmadan önce, istemci tarafı dosya sistemi, dosyanın özniteliklerini getirmek için sunucuya bir GETATTR isteği gönderir. Öznitelikler, daha da önemlisi, dosyanın sunucuda en son ne zaman değiştirildiğine dair bilgileri içerir; değişiklik zamanı, dosyanın istemci önbelleğine getirildiği zamandan daha yeniyse, istemci dosyayı **geçersiz kılar (invalidates)**, böylece istemci önbelleğinden kaldırır ve sonraki okumaların sunucuya gidip dosyanın en son sürümünü almasını sağlar. Öte yandan, istemci dosyanın en son sürümüne sahip olduğunu görürse, devam eder ve önbelleğe alınmış içeriği kullanır, böylece performansı artırır.

Sun'daki orijinal ekip bu çözümü eski önbellek sorununa uyguladığında, yeni bir sorun fark ettiler; Birdenbire, NFS sunucusu GETATTR istekleriyle dolup taştı. İzlenmesi gereken iyi bir mühendislik ilkesi, ortak durum için tasarım yapmak ve iyi çalışmasını sağlamaktır; Burada, **yaygın durum (common case)** bir dosyaya yalnızca tek bir istemciden (belki de tekrar tekrar) erişilmesi olsa da, istemcinin dosyayı başka kimsenin değiştirmediğinden emin olmak için her zaman sunucuya GETATTR istekleri göndermesi gerekiyordu. Böylece bir istemci sunucuyu bombardımana tutar ve çoğu zaman kimsenin değiştirmediği zamanlarda sürekli olarak "bu dosyayı değiştiren oldu mu?" diye sorar.

Bu durumu düzeltmek için (biraz), her istemciye bir **öznitelik önbelleği (attribute cache)** eklenmiştir. İstemci bir dosyaya erişmeden önce dosyayı doğrulamaya devam eder, ancak çoğu zaman öznitelikleri getirmek için yalnızca öznitelik önbelleğine bakar. Belirli bir dosyanın öznitelikleri, dosyaya ilk erişildiğinde önbelleğe yerleştirildi ve belirli bir süre sonra (örneğin 3 saniye) zaman aşımına uğradı. Böylece, bu üç saniye boyunca, tüm dosya erişimleri önbelleğe alınmış dosyayı kullanmanın uygun olduğunu belirler ve böylece sunucuyla ağ iletişimi olmadan bunu yapar.

49.10 NFS Önbellek Tutarlılığını Değerlendirme

NFS önbellek tutarlılığı hakkında son birkaç söz. Flush-on-close davranışı "mantıklı" olarak eklendi, ancak belirli bir performans sorunu ortaya çıkardı. Özellikle, bir istemcide geçici veya kısa ömürlü bir dosya oluşturulduysa ve yakında silindiyse, yine de sunucuya zorlanır. Daha ideal bir uygulama, bu tür kısa ömürlü dosyaları silinene kadar bellekte tutabilir ve böylece sunucu etkileşimini tamamen ortadan kaldırabilir, belki de performansı artırabilir.

Daha da önemlisi, NFS'ye bir öznitelik önbelleğinin eklenmesi, bir dosyanın tam olarak hangi sürümünü aldığını anlamayı veya akıl yürütmeyi çok zorlaştırdı. Bazen en son sürümü alırsınız; Bazen eski bir sürümü elde edersiniz, çünkü öznitelik önbelleğiniz henüz zaman aşımına uğramamıştı ve bu nedenle istemci size istemci belleğinde ne olduğunu vermekten mutluluk duyuyordu. Bu çoğu zaman iyi olmasına rağmen, bazen garip davranışlara yol açacaktır (ve hala öyledir!).

Ve böylece NFS istemcisini önbelleğe alan tuhaflığı tanımladık. Bir uygulamanın ayrıntılarının, tersi yerine kullanıcı tarafından gözlemlenebilir semantikleri tanımlamaya hizmet ettiği ilginç bir örnek olarak hizmet eder.

49.11 Sunucu Tarafı Yazmayı Arabelleğe Alma Üzerindeki Etkileri

Şimdiye kadar odak noktamız istemciyi önbelleğe almak olmuştur ve ilginç sorunların çoğunun ortaya çıktığı yer burasıdır. Bununla birlikte, NFS sunucuları da çok fazla belleğe sahip iyi donanımlı makineler olma eğilimindedir ve bu nedenle önbelleğe alma özelliğine sahiptirler.

Veriler (ve meta veriler) diskten okunduğunda, NFS sunucuları bunları bellekte tutar ve söz konusu verilerin (ve meta verilerin) sonraki okumaları diske gitmez, bu da performansta potansiyel (küçük) bir artıştır.

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

Yanıt, istemcilerin sunucu hatasını nasıl ele aldığına dair varsayımlarımızda yatmaktadır. Bir müşteri tarafından verilen aşağıdaki yazma sırasını 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 yazılar, bir dosyanın üç bloğunun üzerine a's, sonra b's ve sonra c's bloğu ile yazar. Bu nedenle, dosya başlangıçta şöyle görünüyorsa:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Bu yazılardan sonraki nihai sonucun böyle olmasını bekleyebiliriz, x'ler, y'ler ve z'ler sırasıyla a'lar, b'ler ve c'ler ile yazılır.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccc
```

Şimdi örnek için, bu üç istemci yazmasının sunucuya üç ayrı WRITE protokolü mesajı olarak verildiğini varsayalım. İlk WRITE iletisinin sunucu tarafından alındığını ve diske verildiğini ve istemcinin başarısı hakkında bilgilendirildiğini varsayalım. Şimdi, ikinci yazmanın bellekte yalnızca arabelleğe alındığını ve sunucunun diske zorlamadan önce istemciye başarısını bildirdiğini varsayalım; ne yazık ki, sunucu diske yazmadan önce çöküyor. Sunucu hızlı bir şekilde yeniden başlatılır ve üçüncü yazma isteğini alır ve bu da başarılı olur.

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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- hata
cccccccccccccccccccccccccccccccccccccccccc
```

Eyvah! Sunucu istemciye ikinci yazmayı diske işlemeden önce başarılı olduğunu söylediğinden, dosyada uygulamaya bağlı olarak felaket olabilecek eski bir yığın kalır.

BİR KENARA: İNOVASYON İNOVASYONU DOĞURUR

Birçok öncü teknolojide olduğu gibi, NFS'yi dünyaya getirmek, başarısını sağlamak için başka temel yenilikler de gerektirdi. Muhtemelen en kalıcı olanı, farklı dosya sistemlerinin işletim sistemine kolayca takılmasını sağlamak için Sun tarafından tanıtılan **Sanal Dosya Sistemi (Virtual File System) (VFS)** / **Sanal Düğüm (Virtual Node) (vnode)** arayüzüdür [K86].

VFS katmanı, bağlama ve çıkarma, dosya sistemi genelinde istatistikler almak 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 vb. gibi tüm işlemlerden oluşur.

Yeni bir dosya sistemi oluşturmak için, basitçe bu "yöntemleri" tanımlamak gerekir; çerçeve daha sonra geri kalanını ele alır, sistem çağrılarını belirli bir dosya sistemi uygulamasına bağlar, tüm dosya sistemlerinde ortak olan genel işlevleri (örneğin, önbelleğe almak) merkezi bir şekilde gerçekleştirir ve böylece birden fazla dosya sistemi uygulamasının aynı sistem içinde aynı anda çalışması için bir yol sağlar.

Bazı ayrıntılar değişmiş olsa da, birçok modern sistemde Linux, BSD varyantları, macOS ve hatta Windows (Yüklenebilir Dosya Sistemi biçiminde) dahil olmak üzere bir VFS / vnode katmanı vardır. NFS dünyayla daha az alakalı hale gelse bile, altındaki gerekli temellerden bazıları yaşamaya devam edecektir.

Bu sorunu önlemek için, NFS sunucularının istemciyi başarı konusunda bilgilendirmeden önce her yazmayı kararlı (kalıcı) depolamaya işlemesi gerekir; bunu yapmak, istemcinin yazma sırasında sunucu hatasını algılamasını ve böylece sonunda başarılı olana kadar yeniden denemesini sağlar. Bunu yapmak, yukarıdaki örnekte olduğu gibi birbirine karışmış dosya içerikleriyle asla sonuçlanmayacağımızı garanti eder.

Bu gereksinimin NFS sunucusu uygulamasında yol açtığı sorun, yazma performansının büyük bir özen göstermeden en büyük performans darboğazı olabileceğidir. Gerçekten de, bazı şirketler (örneğin, Network Appliance), hızlı bir şekilde yazma işlemi gerçekleştirebilen bir NFS sunucusu oluşturmak gibi basit bir amaçla ortaya çıkmıştır; kullandıkları bir hile, ilk önce yazmaları pil destekli bir belleğe koymaktır, böylece verileri kaybetme korkusu olmadan ve hemen diske yazmak zorunda kalmadan WRITE isteklerine hızlı bir şekilde yanıt vermeyi sağlar; ikinci püf noktası, sonunda yapılması gerektiğinde diske hızlı bir şekilde yazmak için özel olarak tasarlanmış bir dosya sistemi tasarımı kullanmaktır [HLM94, RO91].

49.12 Özet

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

BİR KENARAR: ANAHTAR NFS TERİMLERİ

- NFS'de hızlı ve basit çarpışma kurtarmanın ana hedefini gerçekleştirmenin anahtarı, **durum bilgisi olmayan (stateless)** bir protokolün tasarımıdır. Bir çökmeden sonra, sunucu hızlı bir şekilde yeniden başlatılabilir ve istekleri yeniden sunmaya başlayabilir; istemciler yalnızca başarılı olana kadar istekleri **yeniden dener (retry)**.
- İstekleri **aynı güç (idempotent)** hale getirmek, NFS protokolünün merkezi bir yönüdür. Bir işlem, birden fazla kez gerçekleştirmenin etkisi, bir kez gerçekleştirmeye eşdeğer olduğunda denkgüçlüdür. NFS'de, idempotency istemcinin endişelenmeden yeniden denemesini sağlar ve istemci kayıp ileti yeniden iletimini ve istemcinin sunucu çökmelerini nasıl işlediğini birleştirir.
- Performans sorunları, istemci tarafında **önbelleğe (caching)** alma gereksinimini belirler ve **arabelleğe almayı yazar (write buffering)** ancak **önbellek tutarlılığı sorunu (cache consistency problem)** ortaya çıkarır.
- NFS uygulamaları, tutarlılığı birden fazla yolla önbelleğe almak için bir mühendislik çözümü sağlar: **Kapatırken yıka (flush-on-close) (açmak için kapat (close-to-open))** yaklaşımı, bir dosya kapatıldığında içeriğinin sunucuya zorlanmasını sağlayarak diğer istemcilerin sunucudaki güncellemeleri gözlemlemesini sağlar. Öznitelik önbelleği, bir dosyanın değişip değişmediğini (GETATTR istekleri aracılığıyla) sunucuyla kontrol etme sıklığını azaltır.
- NFS sunucuları, başarılı bir şekilde geri dönmeden önce kalıcı medyaya yazma işlemi yapmalıdır; Aksi takdirde veri kaybı meydana gelebilir.
- Sun, işletim sistemine NFS entegrasyonunu desteklemek için **VFS / Vnode (Sanal Dosya Sistemi / Sanal Düğüm**) arayüzünü tanıtarak birden fazla dosya sistemi uygulamasının aynı işletim sisteminde bir arada bulunmasını sağladı.

Operasyonların yaygınlığı esastır; İstemci başarısız bir işlemi güvenli bir şekilde yeniden yürütebildiğinden, sunucunun isteği yürütüp yürütmediğine bakılmaksızın bunu yapmak uygundur.

Ayrıca, önbelleğe almanın çok istemcili, tek sunuculu bir sisteme girmesinin işleri nasıl karmaşıklaştırabileceğini de gördük. Özellikle, sistem rea-sonable davranmak için önbellek tutarlılığı sorununu çözmelidir; Bununla birlikte, NFS bunu bazen gözlemlenebilir derecede garip davranışlarla sonuçlanabilecek biraz geçici bir şekilde yapar. Son olarak, sunucu önbelleğe almanın nasıl zor olabileceğini gördük: sunucuya yazılanlar, başarıya dönmeden önce kararlı depolamaya zorlanmalıdır (aksi takdirde veriler kaybolabilir).

Kesinlikle alakalı, kesinlikle güvenlik olmayan diğer konular hakkında konuşmadık. İlk NFS uygulamalarında güvenlik oldukça gevşekti; Bir istemcideki herhangi bir kullanıcının diğer kullanıcılar gibi davranması ve böylece neredeyse tüm dosyalara erişmesi oldukça kolaydı. Daha sonra daha ciddi kimlik doğrulama hizmetleriyle (örneğin, Kerberos [NT94]) entegrasyon bu bariz eksiklikleri gidermiştir.

Referanslar

[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 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!*

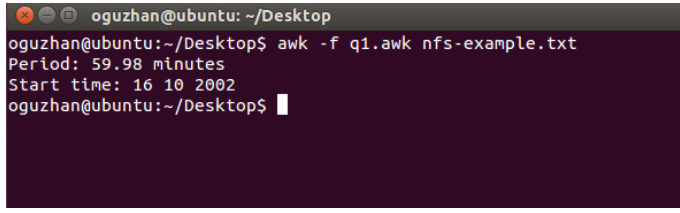
[V72] "La Begueule" by Francois-Marie Arouet a.k.a. Voltaire. Published in 1772. *Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they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

Ödev (Ölçüm)

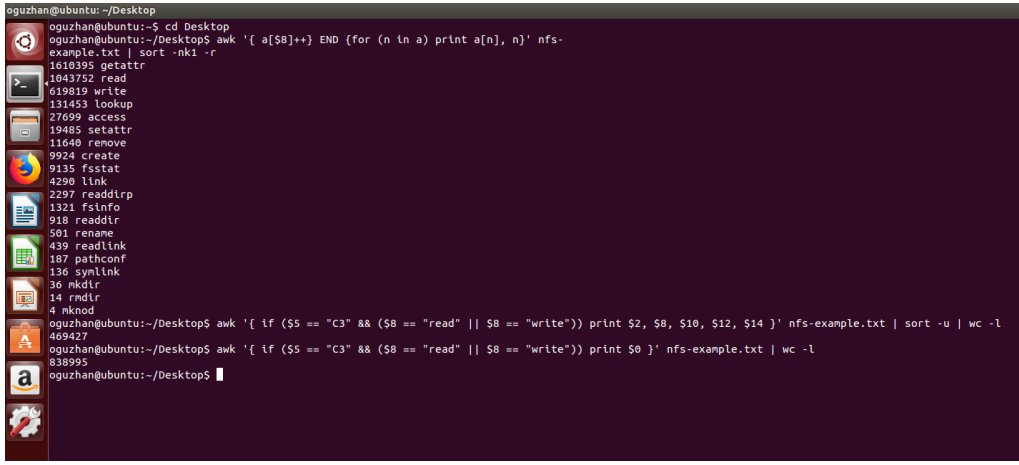
Bu ödevde, gerçek izleri kullanarak biraz NFS iz analizi yapacaksınız. Bu izlerin kaynağı Ellard ve Seltzer'in çabasıdır [ES03]. Başlamadan önce ilgili README'yi okuduğunuzdan ve ilgili tar. uzantılı OSTEP ev ödevi sayfasından (her zamanki gibi) indirdiğinizden emin olun.

Sorular

1. 1. İzleme analiziniz için ilk soru: İlk sütunda bulunan zaman damgalarını kullanarak, izlerin alındığı süreyi belirleyin. Süre ne kadardır? Hangi gün/hafta/ay/yıl idi? (Bu, dosya adında verilen ipucuyla eşleşiyor mu?) İpucu: Dosyanın ilk ve son satırlarını ayıklamak ve hesaplamayı yapmak için head -1 ve tail -1 araçlarını kullanın.

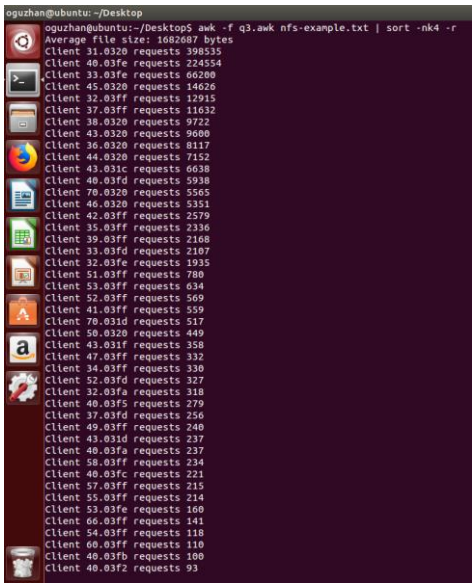


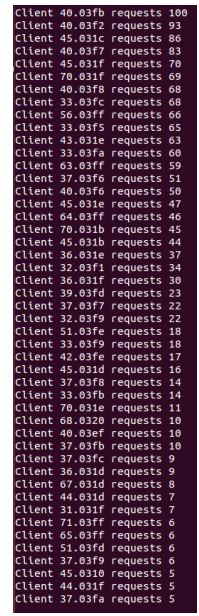
2. Şimdi, bazı işlem sayımları yapalım. İzde her bir işlem türünden kaç tanesi meydana gelir? Bunları frekansa göre sıralayın; en sık hangi işlemler yapılır? NFS adına uygun mu?

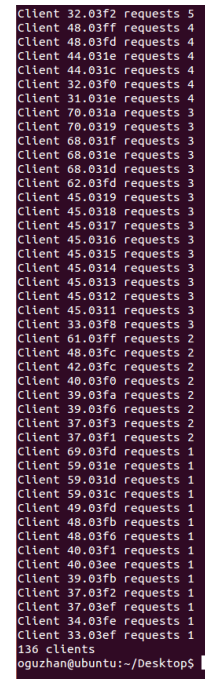


469427/838995 = %44, 05 okuma ve yazma komutları en sık görülen işlemlerdir.

3. Şimdi bazı özel işlemlere daha ayrıntılı olarak bakalım. Örneğin, GETATTR isteği, isteğin hangi kullanıcı kimliği için gerçekleştirildiği, dosyanın boyutu vb. dahil olmak üzere dosyalar hakkında birçok bilgi döndürür. İzleme içinde erişilen dosya boyutlarının dağıtımını yapın; ortalama dosya boyutu nedir? Ayrıca, izlemedeki dosyalara kaç farklı kullanıcı erişir? Birkaç kullanıcı trafiği doyuruyor mu, yoksa daha mı yayılıyor? GETATTR yanıtlarında başka hangi ilginç bilgiler bulunur?





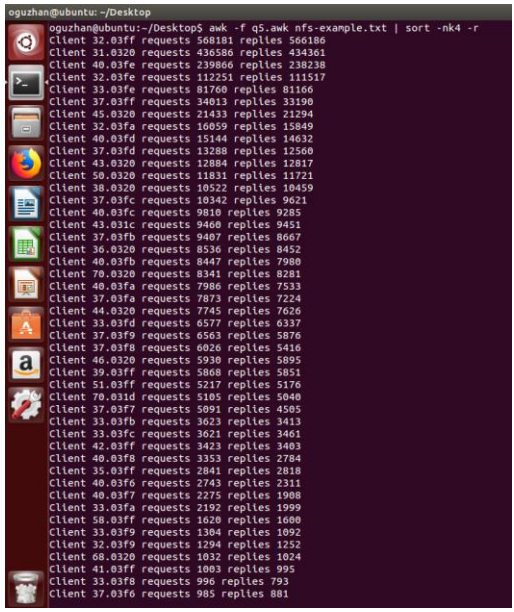


Ortalama dosya boyutu 1. Görselde görüldüğü üzere 1682687 baytdır.
İzlemedeki dosyalara 136 farklı kullanıcı erişir.
Verilerden anladığımız kadarıyla birkaç kullanıcı trafiği doyuruyor.
İlk satırlarda bulunan değerler ile son satırlarda bulunan değerler arasında büyük farklar vardır. Bundan dolayı birkaç kullanıcının trafiği doyurduğu sonucunu söyleyebiliriz.

4. Ayrıca, belirli bir dosyaya yönelik isteklere bakabilir ve dosyalara nasıl erişildiğini belirleyebilirsiniz. Örneğin, belirli bir dosya sırayla okunuyor veya yazılıyor mu? Yoksa rastgele mi? Cevabı hesaplamak için READ ve WRITE isteklerinin/yanıtlarının ayrıntılarına bakın.

Belirli bir dosya sırasıyla yazılıyor ancak okuması sırayla olmuyor.
Kısaca yazma sıralı iken okuma sıralı değildir.

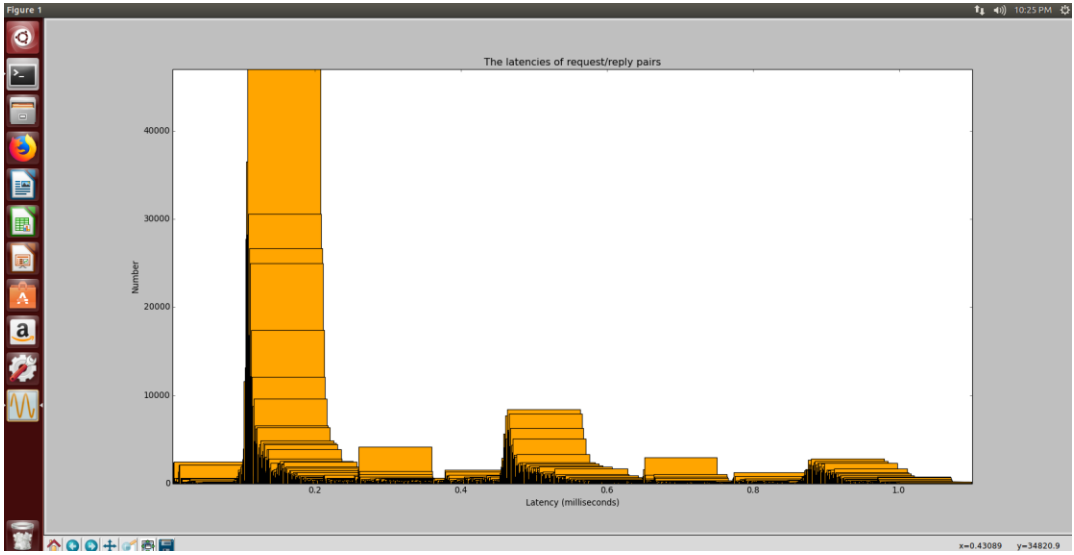
5. Trafik birçok makineden gelir ve bir sunucuya gider (bu izlemede). İzlemede kaç farklı istemci olduğunu ve her birine kaç istek/yanıt gittiğini gösteren bir trafik matrisi hesaplayın. Birkaç makine baskın mı, yoksa daha dengeli mi?



İzlemede 219 farklı istemci vardır.
Sadece 5 istemcinin 10 binden fazla isteği ve yanıtı olduğu için birkaç makinenin baskın olduğu sonucuna varabiliriz. Denge yoktur.

```
Client 63.03fe requests 3 replies 3
Client 59.031a requests 2 replies 2
Client 41.03fc requests 2 replies 2
Client 34.03fd requests 2 replies 2
Client 71.03fe requests 1 replies 1
Client 66.03fd requests 1 replies 1
Client 42.03f8 requests 1 replies 1
Client 42.03f7 requests 1 replies 1
Client 40.03ed requests 1 replies 1
Client 40.024d requests 1 replies 1
Client 37.031d requests 1 replies 1
Client 37.031c requests 1 replies
Client 37.031b requests 1 replies 1
Client 37.031a requests 1 replies 1
Client 37.0319 requests 1 replies 1
Client 37.0318 requests 1 replies 1
Client 37.0317 requests 1 replies 1
Client 37.0316 requests 1 replies 1
Client 37.0315 requests 1 replies 1
Client 37.0314 requests 1 replies 1
Client 37.0313 requests 1 replies 1
Client 37.0312 requests 1 replies 1
Client 37.0311 requests 1 replies 1
Client 37.0310 requests 1 replies 1
Client 37.030f requests 1 replies 1
Client 37.030e requests 1 replies 1
Client 37.030d requests 1 replies 1
Client 37.030c requests 1 replies 1
Client 37.030b requests 1 replies 1
Client 37.030a requests 1 replies 1
Client 37.0309 requests 1 replies 1
Client 37.0308 requests 1 replies 1
Client 37.0307 requests 1 replies 1
Client 37.0306 requests 1 replies 1
Client 37.0305 requests 1 replies 1
Client 37.0304 requests 1 replies 1
Client 37.0303 requests 1 replies 1
Client 37.0302 requests 1 replies 1
Client 37.0301 requests 1 replies 1
Client 37.0300 requests 1 replies 1
Client 37.02ff requests 1 replies 1
Client 37.02fe requests 1 replies 1
Client 37.02fd requests 1 replies 1
Client 37.02fc requests 1 replies 1
Client 32.03ca requests 1 replies 1
219 clients
oguzhan@ubuntu:~/Desktop$
```

6. Zamanlama bilgileri ve istek/yanıt başına benzersiz kimlik, belirli bir isteğin gecikme süresini hesaplamanıza izin vermelidir. Tüm istek/yanıt çiftlerinin gecikme sürelerini hesaplayın ve bunları bir dağıtım olarak çizin. Ortalama nedir? Maksimum? Minimum?



Grafikten anlayacağımız gibi istek/yanıt çiftlerinin gecikme sürelerinin ortalaması 0,2 milisaniyeye yakındır. Maksimum 1 milisaniye iken minimum 0 milisaniyedir.

7. İstek veya yanıtlar kaybolabileceğinden dolayı bazen istekler yeniden denenir. İzleme örneğinde böyle bir yeniden denemeye dair herhangi bir kanıt bulabilir misiniz?

2. Sorunun cevabında yeninden denemeye dair herhangi bir kanıt bulamadım.

8. Daha fazla analizle cevaplayabileceğiniz birçok başka soru var. Hangi soruların önemli olduğunu düşünüyorsunuz? Onları bize önerin belki de buraya ekleyeceğiz!

Kaybolan isteklerin sayısını veya attr önbellek süresini bulabiliriz.