49

Sun'ın Ağ Dosya Sistemi (NFS)

Dağıtık istemci/sunucu bilgi işleminin ilk kullanımlarından biri dağıtık dosya sistemleri alanında olmuştur. Böyle bir ortamda, çok sayıda istemci makine ve bir (veya birkaç tane) sunucu vardır; sunucu verileri disklerinde depolar ve istemciler iyi biçimlendirilmiş protokol mesajları aracılığıyla veri talep eder. Şekil 49.1'de temel kurulum gösterilmektedir.

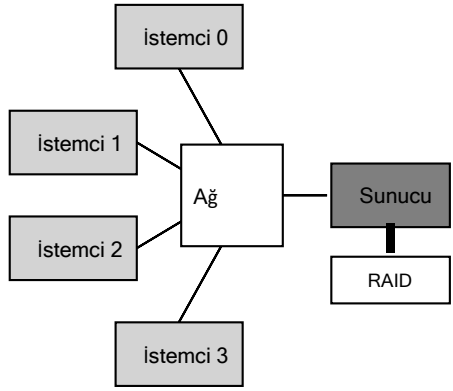


Figure 49.1: **Genel Bir İstemci/Sunucu Sistemi (A Generic Client/Server System)**

Resimden de görebileceğiniz gibi, sunucu disklere sahiptir ve istemciler bu disklerdeki dizinlerine ve dosyalarına erişmek için bir ağ üzerinden mesaj gönderirler. Neden bu düzenlemeyle uğraşıyoruz? (Yani, neden istemcilerin yerel disklerini kullanmalarına izin vermiyoruz?) Öncelikle bu kurulum, verilerin istemciler arasında kolayca **paylaşılmasını (sharing)** sağlar. Böylece, bir makinede (İstemci 0) bir dosyayı 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. Diğer bir avantaj **güvenlik (security)** olabilir; tüm sunucuların kilitli bir makine odasında olması, belirli türden sorunların ortaya çıkmasını engeller.

Kritik Nokta: Dağıtılmış Dosya Sistemi Nasıl Oluşturulur

Dağıtılmış bir dosya sistemi nasıl oluşturulur? Düşünülmesi gereken temel hususlar nelerdir? Neyi yanlış yapmak kolaydır? 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 (client- side file system)** aracılığıyla, dosya ve dizinlere erişen istemci uygulamaları. Bir istemci uygulaması, sunucuda depolanan dosyalara erişmek için istemci tarafı dosya sistemine **sistem çağrıları (system calls)** (`open()`, `read()`, `write()`, `close()`, `mkdir()`, gibi) gönderir. Böylece, istemci uygulamaları için dosya sistemi, belki performans dışında yerel (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, **sunucu tarafındaki dosya sistemine (server-side file system)** (ya da genel olarak adlandırıldığı şekliyle **dosya sunucusuna (file server)**) belirli bir bloğu okuması için bir mesaj gönderebilir; dosya sunucusu daha sonra bloğu diskten (ya da kendi bellek içi önbelleğinden) okuyacak ve istenen veriyle birlikte istemciye bir mesaj geri gönderecektir. İstemci tarafı dosya sistemi daha sonra verileri `read()` sistem çağrısına sağlanan kullanıcı arabelleğine kopyalayacak ve böylece istek tamamlanacaktır. İstemcide aynı bloğun, sonraki bir `read()` işleminin istemci belleğinde veya istemcinin diskinde bile **önbelleğe alındığını (cached)** unutmayın; böyle en iyi durumda, ağ trafiği oluşturulmasına gerek yoktur.

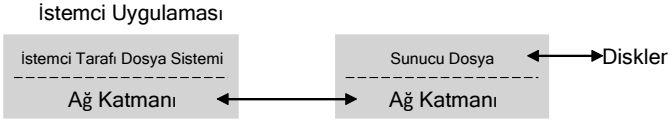


Figure 49.2: **Dağıtılmış Dosya Sistemi Mimarisi (Distributed File System Architecture)**

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

Bir Kenara: Sunucular Neden Çöker

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

49.2 NFS'ye Doğru

En eski ve oldukça başarılı dağıtılmış sistemlerden biri Sun Mikrosistemleri tarafından geliştirilmiştir ve Sun Ağ Dosya Sistemi (veya NFS) olarak bilinir [S86]. Sun, NFS'yi tanımlarken alışılmadık bir yaklaşım benimsedi: Sun, tescilli ve kapalı bir sistem oluşturmak yerine, istemcilerin ve sunucuların iletişim kurmak için kullanacağı tam mesaj biçimlerini 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 bağlı.

49.3 Odak: Basit ve Hızlı Sunucu Çökmesinden Kurtulma

Bu bölümde, uzun yıllar boyunca standart olan NFS protokolünü (Sürüm 2, diğer adıyla NFSv2), tartışacağız; NFSv3'e geçerken küçük değişiklikler yapıldı ve NFSv4'e geçerken daha büyük ölçekli protokol değişiklikleri yapıldı. Bununla birlikte, NFSv2 hem harika hem de sinir bozucu ve bu nedenle odak noktamız olarak hizmet ediyor.

NFSv2'de protokol tasarımındaki ana hedef *basit ve hızlı sunucu çökmesinden kurtarmaydı*. Çok istemcili, tek sunuculu bir ortamda bu hedef çok mantıklıdır; sunucunun kapalı olduğu (veya kullanılmadığı) dakika tüm istemci makineleri (ve kullanıcılarını) mutsuz ve verimsiz hale getirir.

49.4 Hızlı Çökme Kurtarmanın Anahtarı: Durumsuzluk

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

Durum bilgili (stateful) (durumsuz değil) bir protokol örneği için `open()` sistem çağrısını düşünün. Bir yol adı verildiğinde, `open()` bir dosya tanıtıcısı (bir tamsayı) döndürür. Bu tanımlayıcı, bu uygulama kodunda olduğu gibi, çeşitli dosya bloklarına erişmek için sonraki `read()` veya `write()` isteklerinde kullanılır (boşluk nedeniyle sistem çağrılarının uygun hata denetiminin yapılmadığını unutmayın):

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); //"fd" tanımlayıcıyı al
read(fd, buffer, MAX); //"fd" aracılığıyla foo'dan MAX'ı okuyun
read(fd, buffer, MAX); // MAX'i tekrar oku
...
read(fd, buffer, MAX); // MAX'i tekrar oku
close(fd);             // dosyayı kapat
```

Figure 49.3: **İstemci Kodu: Dosyadan Okuma (Client Code: Reading From A File)**

Şimdi, istemci tarafı dosya sisteminin, sunucuya "'foo' dosyasına 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 dosya sunucusuna bir mesajla ileterek "size burada ilettiğim tanımlayıcı tarafından atıfta bulunulan dosyadan bazı baytları okuyun" der.

Bu örnekte, dosya tanıtıcı, istemci ile sunucu arasındaki **paylaşılan durumun (shared state)** bir parçasıdır (Ousterhout bu **dağıtılmış durumu (distributed state)** [O91] olarak adlandırır). Paylaşılan durum, yukarıda da belirttiğimiz gibi, çökme kurtarmayı zorlaştırır. Sunucunun ilk okuma tamamlandıktan sonra, ancak istemci ikincisini yayınlamadan önce çöktüğünü düşünün. Sunucu tekrar çalışmaya başladıktan sonra, istemci ikinci okumayı gerçekleştirir. Ne yazık ki, sunucunun fd'nin hangi dosyayı 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 gerekecektir; burada istemci, sunucuya bilmesi gerekenleri (bu durumda, fd dosya tanımlayıcısının foo dosyasına atıfta bulunduğu) söyleyebilmek için hafızasında yeterli bilgi bulundurduğundan emin olacaktır.

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

Bu nedenden 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.

49.5 NFSv2 Protokolü

Böylece NFSv2 protokol tanımına ulaşıyoruz. Problem ifademiz basittir:

> **Kritik Nokta: Durumsuz bir Dosya Protokolü Nasıl Tanımlanır**
>
> Durumsuz çalışmayı etkinleştirmek için ağ protokolünü nasıl tanımlayabiliriz? Açıkçası, `open()` gibi durum bilgisi içeren çağrılar tartışmanın bir parçası olamaz (çünkü sunucunun açık dosyaları takip etmesini gerektirir); ancak istemci uygulaması dosya ve dizinlere erişmek için `open()`, `read()`, `write()`, `close()` ve diğer standart API çağrılarını çağırmak isteyecektir. Bu nedenle, rafine bir soru olarak, protokolü hem durumsuz olacak hem de POSIX dosya sistemi API'sini destekleyecek şekilde nasıl 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 tanıtıcısını üç önemli bileşene sahip olarak 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 sistemine başvurduğunu 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 döndürür: nitelikler
NFSPROC_SETATTR	dosya tanıtıcısı, nitelikler döndürür: –
NFSPROC_LOOKUP	dizin dosya tanıtıcısı, aranacak dosya/dizin adı döndürür: dosya tanıtıcısı
NFSPROC_READ	dosya tanıtıcısı, ofset, sayım verileri, nitelikler
NFSPROC_WRITE	dosya tanıtıcısı, ofset, sayım, veri nitelikleri
NFSPROC_CREATE	dizin dosya tanıtıcısı, dosya adı, nitelikler –
NFSPROC_REMOVE	dizin dosya tanıtıcısı, kaldırılacak dosyanın adı –
NFSPROC_MKDIR	dizin dosya tanıtıcısı, dizin adı, nitelikler dosya tanıtıcısı
NFSPROC_RMDIR	dizin dosya tanıtıcısı, kaldırılacak dizinin adı –
NFSPROC_READDIR	dizin tanıtıcısı, okunacak bayt sayısı, çerez döndürür: dizin girişleri, çerez (daha fazla giriş almak için)

Figure 49.4: **NFS Protokolü: Örnekler (The NFS Protocol: Examples)**

Protokolün önemli bileşenlerini kısaca vurguluyoruz. 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 dosyası 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 edilebilir; kısalık uğruna burada bağlama protokolünü tartışmıyoruz). İstemcide çalışan bir uygulama `/foo.txt` dosyasını açarsa, 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 dosya üzerinde READ ve WRITE protokol mesajları yayınlayabilir. READ protokol mesajı, protokolün dosya içindeki ofset ve okunacak bayt sayısı ile birlikte dosyanın dosya tanıtıcısını iletmesini gerektirir. Sunucu daha sonra okumayı gerçekleştirebilir (sonuçta, tanıtıcı sunucuya hangi birimin ve hangi inode'dan okunacağını söyler ve ofset ve sayı ona dosyanın hangi baytlarını okuyacağını söyler) ve verileri istemciye geri döndürebilir (veya bir başarısızlık 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 bu dosyanın özniteliklerini getirir. Bu protokol isteğinin NFSv2'de neden önemli olduğunu aşağıda önbelleklemeyi tartışırken göreceğiz (nedenini tahmin edebilir misiniz?).

49.6 Protokolden Dağıtılmış Dosya Sistemine

Umarım şimdi bu protokolün istemci tarafı dosya sistemi ve dosya sunucusu arasında nasıl bir dosya sistemine dönüştürüldüğünü anlamışsınızdır. 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 uygulama düşünelim. 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 ile ancak farklı bir ofsetle verilir.

İkinci olarak, sunucu etkileşimlerinin nerede gerçekleştiğini fark edebilirsiniz. Dosya ilk kez açıldığında, istemci tarafı dosya sistemi bir LOOKUP istek mesajı gönderir. Gerçekten de uzun bir yol adının geçilmesi gerekiyorsa (örneğin, `/home/remzi/foo.txt` istemci üç LOOKUP gönderecektir: biri / dizininde `home`'u aramak için, biri `home`'da `remzi`'yi aramak için ve son olarak biri `remzi`'de `foo.txt`'yi aramak için.

Üçüncü olarak, her sunucu talebinin, talebi 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 isteğe yanıt verebilmek için duruma ihtiyaç duymamasını sağlar.

Client	Server
fd = open("/foo", ...);	
LOOKUP gönder (kök dizini FH, "foo")	LOOKUP isteği al kök dizinde "foo" ara foo'nun FH + niteliklerini döndür
LOOKUP yanıtını al açık dosya Tablosunda dosya desc ayır foo'nun FH'sini tabloda sakla geçerli dosya konumunu saklar (0) dosya tanımlayıcısını uygulamaya döndür	
read(fd, buffer, MAX);	
fd ile açık dosya tablosuna indeks alın NFS dosya tanıtıcısı (FH) geçerli dosya konumunu ofset olarak kullan	
READ gönder (FH, offset=0, count=MAX)	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 istemciye döndürür
READ yanıtını alın dosya konumunu güncelle (+bayt oku) geçerli dosya konumunu ayarla = MAX uygulamaya veri/hata kodu döndür	
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 Serbest tanımlayıcı "fd" açık dosya tablosunda (Sunucuyla konuşmaya gerek yok)	

Figure 49.5: **Dosya Okuma: İstemci Tarafı ve Dosya Sunucusu Eylemleri (Reading A File: Client-side And File Server Actions)**

IPUCU: IDEMPOTANS GÜÇLÜDÜR

İdempotans (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.

49.7 Idempotent İşlemleri ile Sunucu Hatasını İşleme

Bir istemci sunucuya bir mesaj gönderdiğinde, bazen bir yanıt alamaz. Bu başarısızlığın pek çok olası nedeni vardır. Bazı durumlarda, mesaj ağ tarafından düşürülebilir; ağlar mesajları kaybeder ve bu nedenle ya istek ya da yanıt kaybolabilir ve böylece istemci asla bir yanıt alamaz.

Sunucunun çökmüş olması ve bu nedenle şu anda mesajlara yanıt vermiyor olması da 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, talebi 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 sona ererse, istemci isteğin işlenmediğini varsayar ve yeniden gönderir. Sunucu yanıt verirse, her şey yolundadır ve istemci sorunu düzgün bir şekilde halletmiştir.

İstemcinin isteği yeniden deneyebilmesi (hatanın sebebi ne olursa olsun), çoğu NFS isteğinin önemli bir özelliğinden kaynaklanmaktadır: bunlar **idempotent (idempotent)'**dir. İşlemin birden fazla kez gerçekleştirilmesinin etkisi, işlemin tek bir kez gerçekleştirilmesinin etkisine eşdeğer olduğunda, bir işlem idempotent olarak adlandırılır. 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 açıkça idempotenttir; veri güncelleyen bir işlemin bu özelliğe sahip olup olmadığını belirlemek için daha dikkatli düşünülmesi gerekir.

NFS'de çökme kurtarma tasarımının kalbi, en yaygın işlemlerin idempotensidir. LOOKUP ve READ istekleri, yalnızca dosya sunucusundan bilgi okudukları ve güncellemedikleri 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 bilgisiyle tekrarlanabilir.

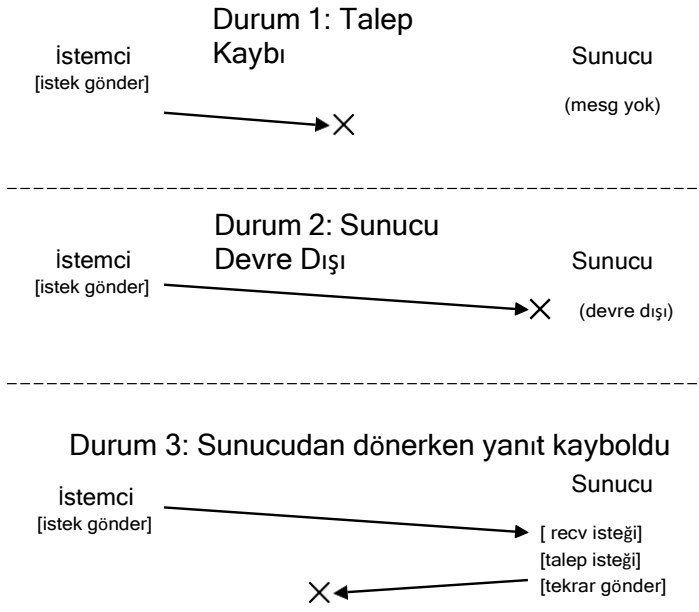


Figure 49.6: **Üç Tür Kayıp (The Three Types Of Loss)**

Bu şekilde, istemci tüm zaman aşımlarını birleşik bir şekilde ele alabilir. Bir WRITE isteği basitçe kaybedildiyse (yukarıdaki Durum 1), istemci 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 istendiği 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), bu da istemcinin isteği yeniden göndermesine neden olur. Sunucu isteği tekrar aldığında, basitçe aynı şeyi yapacaktır: verileri diske yazacak ve bunu yaptığını yanıtlayacaktır. İstemci bu kez yanıtı alırsa, her şey yine yolunda demektir ve böylece istemci hem mesaj kaybını hem de sunucu arızasını tek tip bir şekilde ele almış olur. Harika!

Küçük bir ek: bazı işlemleri boşta bırakmak 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 olmuşken ve daha sonra yalnızca yeniden denemede başarısız olmuşken 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 arıza açısından temiz ve basit hale getirir. Bu nedenle, 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 mükemmelin, iyinin düşmanı olduğunu söyler" [V72] ve bu nedenle buna **Voltaire Yasası (Voltaire's Law)** diyoruz.

49.8 Performansı İyileştirme: İstemci Tarafı Önbelleğe Alma

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

Yukarıdaki alt başlıktaki büyük ve kalın kelimeleri okuyarak tahmin edebileceğiniz gibi cevap, istemci tarafı **önbelleklemedir (caching)**. NFS istemci tarafı dosya sistemi, sunucudan okuduğu dosya verilerini (ve meta verileri) istemci belleğinde önbelleğe alır. Bu nedenle, ilk erişim pahalı olsa da (yani, ağ iletişimi gerektirir), sonraki 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) arabelleğe alır. Bu tür bir **ara belleğe yazma (write buffering)** kullanışlıdır çünkü uygulama `write()` gecikmesini gerçek yazma performansından ayırır, yani uygulamanın `write()` ağrısı 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.

49.9 Önbellek Tutarlılığı Sorunu

Önbellek tutarlılığı sorunu en iyi iki istemci ve tek bir sunucu ile gösterilebilir. C1 istemcisinin bir F dosyasını okuduğunu ve dosyanın bir kopyasını yerel önbelleğinde tuttuğunu düşünün. Şimdi farklı bir istemcinin, C2, F dosyasının üzerine yazdığını ve böylece içeriğini değiştirdiğini düşünün; F dosyasının yeni sürümüne

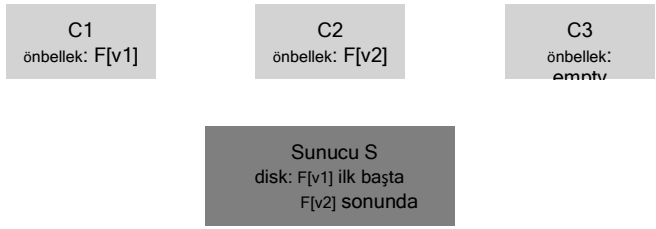


Figure 49.7: **Önbellek Tutarlılığı Sorunu (The Cache Consistency Problem)**

(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, henüz F dosyasına erişmemiş olan üçüncü bir istemci, C3, vardır.

Muhtemelen yaklaşmakta olan sorunu 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 (örneğin C3) yapılan herhangi bir erişim dosyanın eski sürümünü (F[v1]) getirecektir. Bu nedenle, istemcideki yazmaları arabelleğe alarak, diğer istemciler dosyanın eski sürümlerini alabilir, bu da istenmeyen bir durum olabilir; gerçekten de, C2 makinesinde oturum açtığınızı, F'yi güncellediğinizi ve ardından C3'te oturum açıp dosyayı okumaya çalıştığınızı ve yalnızca eski kopyayı aldığınızı düşünün! Bu kesinlikle sinir bozucu olabilir. Bu nedenle, önbellek tutarlılığı sorununun bu yönüne **güncelleme görünürlüğü (update visibility)** diyelim; bir istemciden gelen güncellemeler diğer istemcilerde ne zaman görünür hale gelir?

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

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

İkinci olarak, eski 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 özelliklerini almak için sunucuya bir GETATTR isteği gönderir. Nitelikler, önemli olarak, dosyanın sunucuda en son ne zaman değiştirildiği bilgisini içerir; değişiklik zamanı dosyanın istemci önbelleğine alındığı 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 edecek ve önbelleğe alınan içeriği kullanacak, böylece performansı artıracaktır.

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

Bu durumu (biraz) düzeltmek için her istemciye bir **öznitelik önbelleği (attribute cache)** eklendi. Bir istemci bir dosyaya erişmeden önce yine de doğrulama yapar, ancak çoğu zaman nitelikleri almak için öznitelik önbelleğine bakar. Belirli bir dosya için nitelikler, dosyaya ilk erişildiğinde önbelleğe yerleştirilir ve belirli bir süre sonra (örneğin 3 saniye) zaman aşımına uğrardı. Böylece, bu üç saniye boyunca, tüm dosya erişimleri önbelleğe alınmış dosyayı kullanmanın uygun olduğunu belirleyecek ve böylece sunucuyla ağ iletişimi olmadan bunu yapacaktır.

49.10 NFS Önbellek Tutarlılığını Değerlendirme

NFS önbellek tutarlılığı hakkında son birkaç söz. "Anlamlı" olması için kapatıldığında aynı hizada davranışı eklendi, ancak belirli bir performans sorunu ortaya çıkardı. Spesifik olarak, bir istemcide geçici veya kısa ömürlü bir dosya oluşturulur ve kısa süre sonra silinirse, bu dosya yine de sunucuya zorlanacaktır. 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 alırsınız; bazen de eski bir sürümü alırsınız çünkü öznitelik önbelleğiniz henüz zaman aşımına uğramamıştır ve bu nedenle istemci size istemci belleğinde olanı vermekten mutluluk duyar. Bu çoğu zaman iyi olsa da, zaman zaman garip davranışlara yol açardı (ve hala da açıyor!).

Ve böylece NFS istemci önbelleği olan tuhaflığı tanımlamış olduk. Bir uygulamanın ayrıntılarının, tam tersi yerine, kullanıcı tarafından gözlemlenebilir semantiği tanımlamaya hizmet ettiği ilginç bir örnek olarak hizmet eder.

49.11 Sunucu Tarafı Yazma Arabelleğine Etkileri

Şu ana kadar istemci önbelleğe alma konusuna odaklandık 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

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 arabelleğe alma durumudur. NFS sunucuları, yazma işlemi sabit depolama alanına (örneğin diske veya başka bir kalıcı aygıta) zorlanana kadar bir WRITE protokolü isteğinde kesinlikle başarı döndürmeyebilir. Verilerin bir kopyasını sunucu belleğine yerleştirebilirken, bir WRITE protokolü isteğinde istemciye başarı döndürmek yanlış davranışa neden olabilir; nedenini bulabilir misiniz?

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

```
write(fd, a_buffer, size); //1. bloğu a'larla doldurun
write(fd, b_buffer, size); //2. bloğu b'lerle doldurun
write(fd, c_buffer, size); //3. bloğu c'lerle doldurun
```

Bu yazmalar, bir dosyanın üç bloğunun üzerine önce a'lardan, sonra b'lerden ve sonra da c'lerden oluşan bir blok yazar. Böylece, dosya başlangıçta şu şekilde görünüyorsa:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

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

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccccc
```

Şimdi, örneğin hatırına, bu üç istemci yazma işleminin sunucuya üç ayrı WRITE protokol mesajı olarak verildiğini varsayalım. İlk WRITE mesajının sunucu tarafından alındığını ve diske verildiğini ve istemcinin bunun başarısı hakkında bilgilendirildiğini varsayın. Şimdi ikinci yazmanın sadece bellekte tamponlandığını ve sunucunun diske zorlamadan önce istemciye başarılı olduğunu bildirdiğini varsayalım; ne yazık ki sunucu diske yazmadan önce çöküyor. Sunucu hızla yeniden başlar ve üçüncü yazma isteğini alır, bu da başarılı olur.

Böylece, istemci için tüm istekler başarılı oldu, ancak dosya içeriğinin aşağıdaki gibi görünmesi bizi şaşırttı:

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

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

BİR KENARA: YENİLİK YENİLİĞİ DOĞURUR

Pek çok öncü teknolojide olduğu gibi, NFS'yi dünyaya getirmek de başarısını sağlamak için başka temel yenilikler gerektirdi. Muhtemelen en kalıcı olanı, farklı dosya sistemlerinin işletim sistemine kolayca takılabilmesini sağlamak için Sun tarafından sunulan **Sanal Dosya Sistemi (VFS)(Virtual File System (VFS))** / **Sanal Düğüm (vnode)(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 ile daha az ilgili hale gelse bile, altındaki bazı gerekli temeller yaşamaya devam edecektir.

Bu sorunu önlemek için NFS sunucuları, istemciye başarıyı bildirmeden önce her yazmayı kararlı (kalıcı) depolama alanına işlemelidir; bunu yapmak, istemcinin bir yazma sırasında sunucu arızasını tespit etmesini ve böylece sonunda başarılı olana kadar yeniden denemesini sağlar. Bunu yapmak, yukarıdaki örnekte olduğu gibi dosya içeriklerinin asla birbirine karışmamasını sağlar.

Bu gereksinimin NFS sunucu uygulamasında ortaya çıkardığı sorun, yazma performansının, büyük bir özen gösterilmediği takdirde, başlıca performans darboğazı olabilmesidir. Gerçekten de bazı şirketler (örneğin Network Appliance) yazma işlemlerini hızlı bir şekilde gerçekleştirebilen bir NFS sunucusu oluşturmak 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 nihayetinde bunu yapmak gerektiğinde diske hızlı bir şekilde yazmak için özel olarak tasarlanmış bir dosya sistemi tasarımı kullanmaktır [HLM94, RO91].

49.12 ÖZET

NFS dağıtılmış dosya sisteminin tanıtıldığını gördük. NFS, sunucu arızası karşısında basit ve hızlı kurtarma fikrine odaklanır ve bu amaca dikkatli protokol tasarımı ile ulaşır

BİR KENARA: ANAHTAR NFS TERİMLERİ

- NFS'de hızlı vebasit çökme kurtarma ana hedefini gerçekleştirmenin anahtarı, **durum bilgisi olmayan (stateless)** protokolün tasarlanmasıdır. Bir çökmeden sonra, sunucu hızlı bir şekilde yeniden başlatılabilir ve istekleri tekrar sunmaya başlayabilir; istemciler başarılı olana kadar istekleri **yeniden (retry)** dener.
- İstekleri **idempotent (idempotent)** hale getirmek NFS protokolünün merkezi bir özelliğidir. Bir işlem, birden fazla kez gerçekleştirilmesinin etkisi bir kez gerçekleştirilmesine eşdeğer olduğunda idempotenttir. NFS'de idempotens, 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 alma (caching)** ve **Arabelleği yazma (write buffering)**, ancak bir **önbellek tutarlılığı sorunu (cache consistency problem)** ortaya çıkarır.
- Performans kaygıları, istemci tarafı **önbelleğe alma (caching)** ve **Arabelleği yazma (write buffering)**, ancak bir **önbellek tutarlılığı sorunu (cache consistency problem)** ortaya çıkarır.
- NFS uygulamaları, önbellek tutarlılığı için birden fazla yolla mühendislik çözümü sunar: **kapatıldığında hizalama (flush-on-close)** (**açmaya yakın (close-to-open)**) yaklaşımı, bir dosya kapatıldığında içeriğinin sunucuya zorla gönderilmesini sağlayarak diğer istemcilerin dosyadaki güncellemeleri gözlemlemesine olanak tanır. Bir öznitelik önbelleği, bir dosyanın değişip değişmediğini sunucudan kontrol etme sıklığını azaltır (GETATTR istekleri aracılığıyla).
- NFS sunucuları başarı döndürmeden önce kalıcı ortama yazma işlemi yapmalıdır; aksi takdirde veri kaybı yaşanabilir.
- Sun, NFS'nin işletim sistemine entegrasyonunu desteklemek için **VFS/Vnode(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 yerine getirmiş olsun ya da olmasın bunu yapmakta bir sakınca yoktur.

Ayrıca, çok istemcili, tek sunuculu bir sisteme önbelleğe almanın işleri nasıl karmaşıklaştırabileceğini de 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 zaman gözlemlenebilir derecede garip davranışlara neden olabilir. Son olarak, sunucu önbelleğe almanın nasıl zor olabileceğini gördük: sunucuya yapılan yazmalar başarıya dönmeden önce sabit depolamaya zorlanmalıdır (aksi takdirde veriler kaybolabilir).

Güvenlik başta olmak üzere, konuyla kesinlikle ilgili olan diğer konulardan bahsetmedik. İlk NFS uygulamalarında güvenlik oldukça gevşekti; bir istemcideki herhangi bir kullanıcının diğer kullanıcılar gibi davranması ve böylece neredeyse her dosyaya erişmesi oldukça kolaydı. Daha sonra daha ciddi kimlik doğrulama hizmetleriyle (örneğin Kerberos [NT94]) entegrasyon bu bariz eksiklikleri gidermiştir with more serious authentication services (e.g., Kerberos [NT94]) have addressed these obvious deficiencies.

References

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

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

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

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

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

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

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

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

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

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

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

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

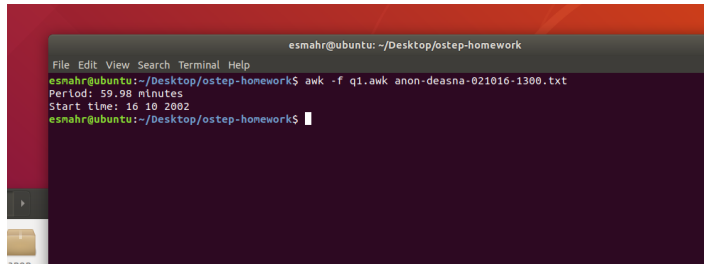
[V72] "La Begueule" by Francois-Marie Arouet a.k.a. Voltaire. Published in 1772. *Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each others throats; but if you have thirty religions, they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

Ödev (Ölçme)

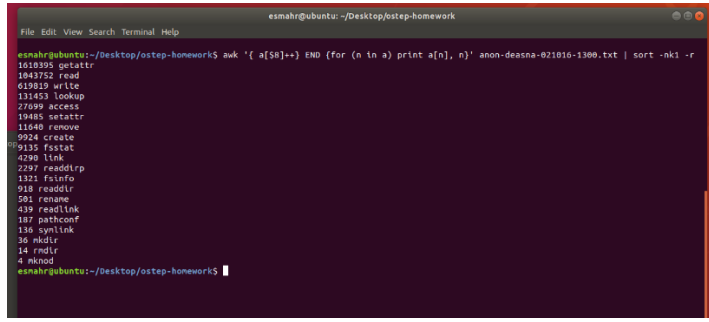
Bu ödevde, gerçek izleri kullanarak biraz NFS iz analizi yapacaksınız. Bu izlerin kaynağı Ellard ve Seltzer'in çabasıdır [ES03]. Başlamadan önce ilgili README'yi okuduğunuzdan ve ilgili tarball'ı OSTEP ödev sayfasından (her zamanki gibi) indirdiğinizden emin olun.

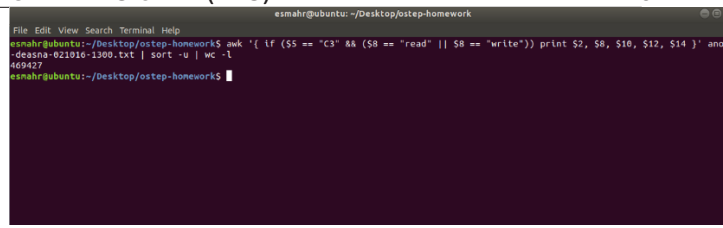
Sorular

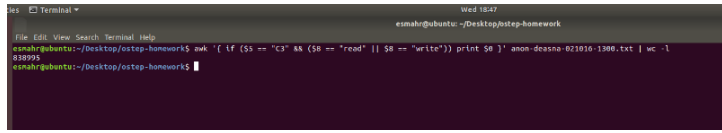
1. İz analiziniz için ilk soru: ilk sütunda bulunan zaman damgalarını kullanarak, izlerin alındığı zaman dilimini belirleyin. Dönem ne kadar uzun? Hangi gün/hafta/ay/yıldı? (bu, dosya adında verilen ipucu ile eşleşiyor mu?) İpucu: Dosyanın ilk ve son satırlarını çıkarmak için `head -1` ve `tail -1` araçlarını kullanın ve hesaplamayı yapın.
   strftime() gawk uzantısıdır, POSIX standardında belirtilmemiştir.
   Dosya adı anon-deasna-021016-1300.txt, bu aynı tarihdir. Çıktının ekran görüntüsü;



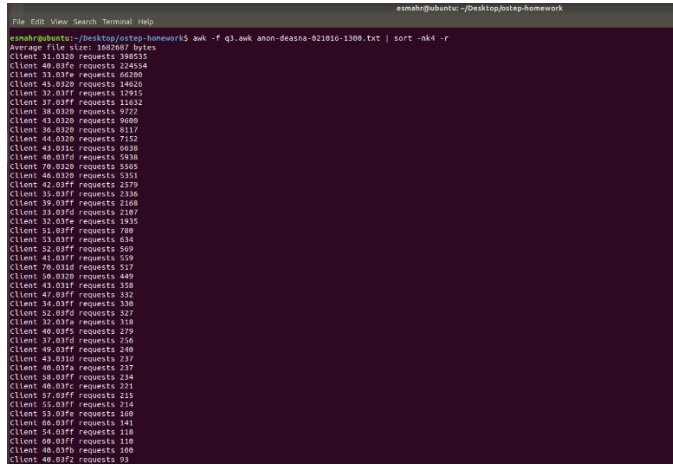
2. Şimdi biraz işlem sayımı yapalım. İzde her bir işlem türünden kaç tane gerçekleşiyor? Bunları sıklığa göre sıralayın; en sık hangi işlem yapılıyor? NFS itibarını koruyor mu?
   Aşağıda elde edilen çıktılara göre yapılan hesaplama:
   1 - 469427/838995 = %44,05 okuma ve yazma komutları yeniden deneme komutlarıdır.

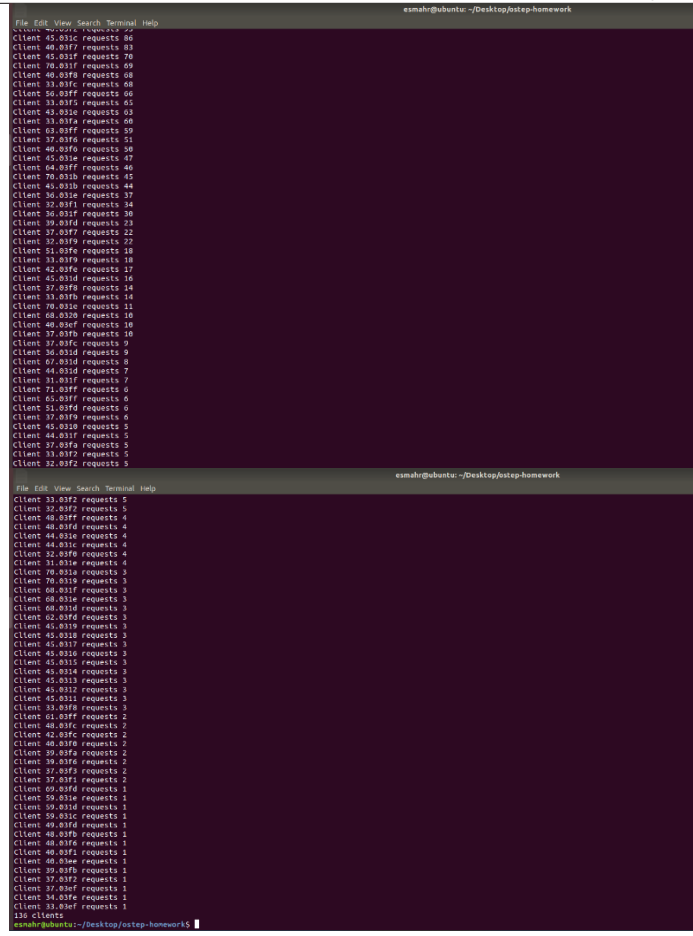






3. Şimdi bazı özel işlemlere daha detaylı bakalım. Örneğin, GETATTR isteği dosyalarla ilgili, isteğin hangi kullanıcı kimliği için gerçekleştirildiği, dosyanın boyutu vb. gibi birçok bilgiyi gönderir. İzleme içinde erişilen dosya boyutlarının dağılımını yapın; ortalama dosya boyutu nedir? Ayrıca, izlemedeki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





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 READ ve WRITE isteklerinin/cevaplarının ayrıntılarına bakın.

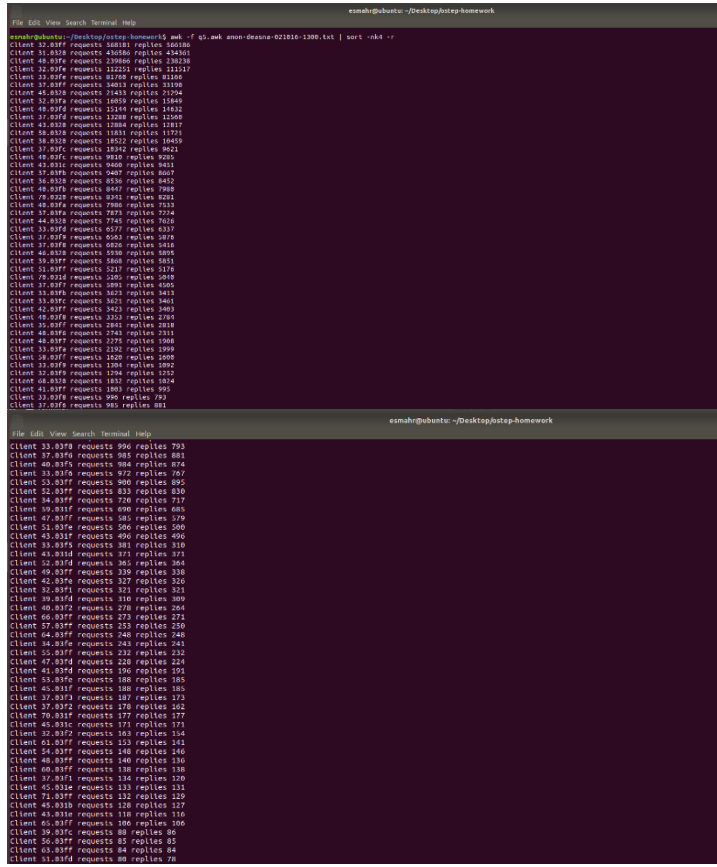
```
$ awk '{ if (($8 == "read" || $8 == "write") && $10 ==
"01122f009ead0d0120000000003669c1928f1001648600001122f009ead0d00" && $2 == "32.03fe") print
$0}' sample

$ awk '{ if (($8 == "read" || $8 == "write") && $10 ==
"01122f009ead0d0120000000003669d1c8a5100164860000011122f009ead0d00" && $2 == "32.03ff") print
$0}' sample

$ awk '{ if (($8 == "read" || $8 == "write") && $10 ==
"01122f009ead0d01200000000087cd5958811101648600001122f009ead0d00" && $2 == "32.03fa") print
$0}' sample
```

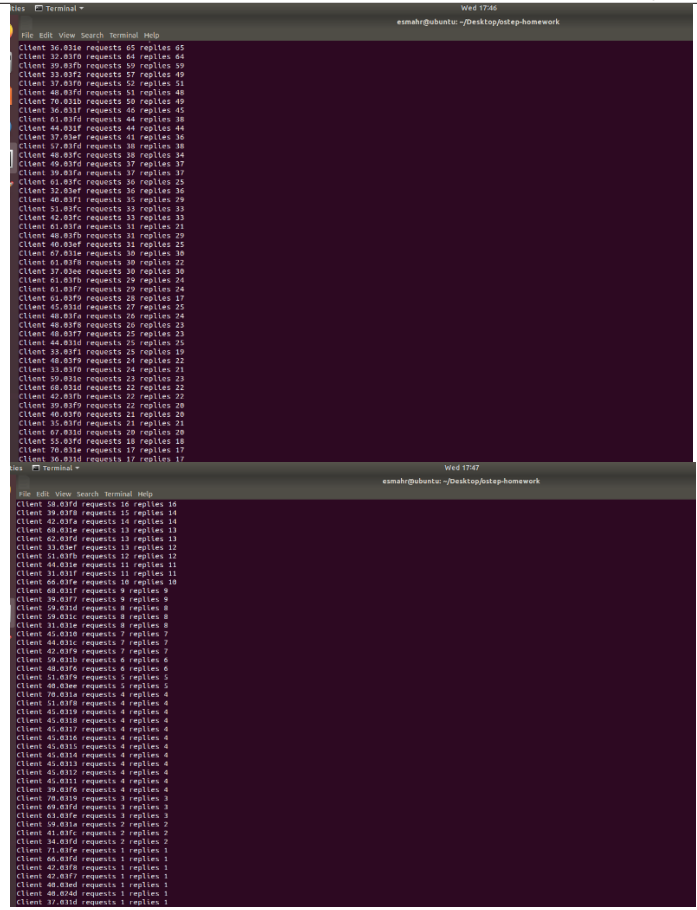
Yazma sıralıdır, ancak okuma sıralı değildir.

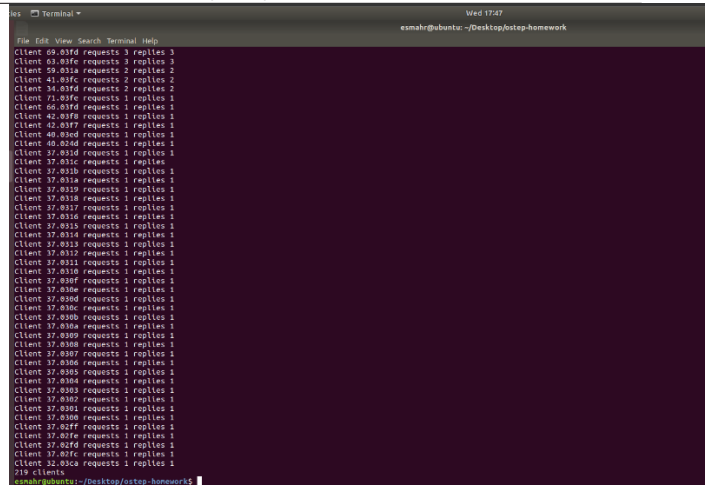
5. Trafik birçok makineden gelir ve tek bir sunucuya gider (bu izde).

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?

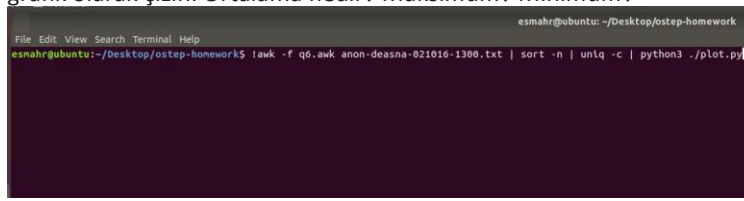
Yalnızca beş istemcinin on binden fazla isteği ve yanıtı vardır. Ekran görüntüleri:



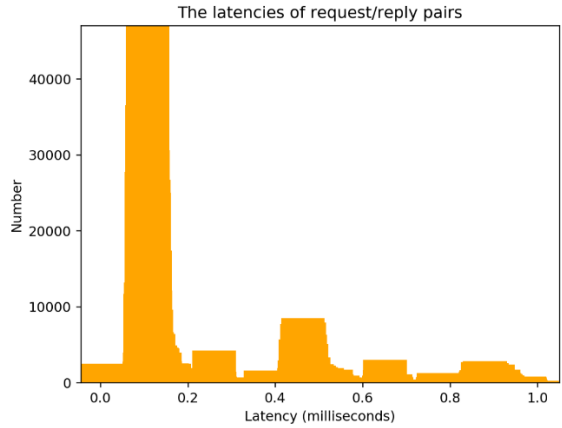

Terminal
Wed 17:46
esmahr@ubuntu: ~/Desktop/ostep-homework
File Edit View Search Terminal Help
Client 36.031e requests 65 replies 65
Client 32.03f0 requests 64 replies 64
Client 39.03fb requests 59 replies 59
Client 33.03f2 requests 57 replies 49
Client 37.03f0 requests 52 replies 51
Client 48.03fd requests 51 replies 48
Client 76.031b requests 50 replies 49
Client 36.031f requests 46 replies 45
Client 61.03fd requests 44 replies 38
Client 44.031f requests 44 replies 44
Client 37.03ef requests 41 replies 36
Client 57.03fd requests 38 replies 38
Client 46.03fc requests 38 replies 34
Client 49.03fd requests 37 replies 37
Client 39.03fa requests 37 replies 37
Client 61.03fc requests 36 replies 25
Client 32.03ef requests 36 replies 36
Client 46.03f1 requests 35 replies 29
Client 51.03fc requests 33 replies 33
Client 42.03fc requests 33 replies 33
Client 61.03fa requests 31 replies 21
Client 48.03fb requests 31 replies 29
Client 46.03ef requests 31 replies 25
Client 67.031e requests 30 replies 30
Client 61.03f8 requests 30 replies 22
Client 37.03ee requests 30 replies 30
Client 61.03fb requests 29 replies 24
Client 61.03f7 requests 29 replies 24
Client 61.03f9 requests 28 replies 17
Client 45.031d requests 27 replies 25
Client 48.03fa requests 26 replies 24
Client 46.03f6 requests 26 replies 23
Client 48.03f7 requests 25 replies 23
Client 44.031d requests 25 replies 25
Client 35.03f1 requests 25 replies 19
Client 48.03f9 requests 24 replies 22
Client 33.03f0 requests 24 replies 21
Client 59.031e requests 23 replies 23
Client 68.031d requests 22 replies 22
Client 42.03fb requests 22 replies 22
Client 39.03f9 requests 22 replies 20
Client 46.03f0 requests 21 replies 20
Client 35.03fd requests 21 replies 21
Client 67.031d requests 20 replies 20
Client 55.03fd requests 18 replies 18
Client 76.031e requests 17 replies 17
Client 36.031d requests 17 replies 17
Terminal
Wed 17:47
esmahr@ubuntu: ~/Desktop/ostep-homework
File Edit View Search Terminal Help
Client 58.037d requests 16 replies 16
Client 39.03f8 requests 15 replies 14
Client 42.03fa requests 14 replies 14
Client 68.031e requests 13 replies 13
Client 62.03fd requests 13 replies 13
Client 33.03ef requests 13 replies 12
Client 51.03fb requests 12 replies 12
Client 44.031e requests 11 replies 11
Client 31.031f requests 11 replies 11
Client 66.03fe requests 10 replies 10
Client 68.031f requests 9 replies 9
Client 39.03f7 requests 9 replies 9
Client 59.031d requests 8 replies 8
Client 59.031c requests 8 replies 8
Client 31.031e requests 8 replies 8
Client 45.0310 requests 7 replies 7
Client 44.031c requests 7 replies 7
Client 42.03f9 requests 7 replies 7
Client 59.031b requests 6 replies 6
Client 48.03f6 requests 6 replies 6
Client 51.03f9 requests 5 replies 5
Client 40.03ee requests 5 replies 5
Client 70.031a requests 4 replies 4
Client 51.03f8 requests 4 replies 4
Client 45.0319 requests 4 replies 4
Client 45.0318 requests 4 replies 4
Client 45.0317 requests 4 replies 4
Client 45.0316 requests 4 replies 4
Client 45.0315 requests 4 replies 4
Client 45.0314 requests 4 replies 4
Client 45.0313 requests 4 replies 4
Client 45.0312 requests 4 replies 4
Client 45.0311 requests 4 replies 4
Client 39.03f6 requests 4 replies 4
Client 70.0319 requests 3 replies 3
Client 69.03fd requests 3 replies 3
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



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



Komutunun sonucunca elde edilen matplotlib garafiği;



7. Bazen istek veya yanıtı kaybolabileceği veya düşebileceği için istekler yeniden denenir. İzleme örneğinde bu tür yeniden denemelere ilişkin

herhangi bir kanıt bulabilir misiniz?

8. Daha fazla analiz yaparak yanıtlayabileceğiniz başka birçok soru var. Sizce hangi sorular önemli? Bunları bize önerin, belki biz de buraya ekleriz!
    1. Attr önbellek süresini bulun.
    2. İstemci tarafı dosya sisteminin dosya okum işlemini gerçekleştirmesi için gerekli protokol mesajını yazın.