50

Andrew Dosya Sistemi (The Andrew File System) (AFS)

Andrew Dosya Sistemi (AFS) 1980'lerde Carnegie-Mellon Üniversitesi(CMU) tarafından tanıtıldı [H+88]. Carnegie-Mellon Üniversitesi'nden tanınmış Profesör M. Satya-narayanan (kısaca"Satya") liderliğindeki projenin ana amacı basitti: **Ölçek (scale)**. Özellikle, bir sunucu mümkün olduğu kadar çok istemciyi destekleyebilecek şekilde dağıtılmış bir dosya sistemini nasıl tasarlayabilir?

İlginçtir ki, tasarım ve uygulamanın ölçeklenebilirliğini etkileyen birçok sayıda bakış açısı vardır. En önemlisi istemciler ve sunucular arasında ki **protokolün(protocol)** tasarımıdır.Örneğin NFS'de protokol istemcileri önbelleğe alınmış içeriklerin değişip değişmediğini belirlemek için düzenli aralıklarla sunucu ile denetlemeye zorlar.Bu da ölçeklenebilirliğini kısıtlamış olur çünkü her denetim, sunucu kaynaklarını(CPU ve ağ bant genişliği dahil) kullandığından dolayı bunun gibi sık denetimler sunucunun yanıt verebileceği istemci sayısını sınırlamaktadır.

AFS ayrıca NFS'den farklıdır çünkü başından beri mantklı kullanıcı tarafından görülebilen davranış biçimi birinci sınıf bir endişe kaynağıdır.

NFS'de önbellek tutarlılığını açıklamak zordur bunun sebebi istemci tarafı, önbellek zaman aşımı aralıkları da dahil olmak üzere doğrudan düşük düzeyli uygulama ayrıntılarına bağlıdır. AFS'de önbellek tutarlılığı basittir ve kolayca anlaşılır çünkü dosya açıldığında bir istemci genellikle sunucudan en son tutarlı kopyayı almış olur.

50.1 AFS Sürüm(version) 1

AFS'nin iki versiyonunu tartışacağız [H+88,S+85]. İlk versiyon(AFSv1 olarak adlandıracağız fakat orijinal sistemde aslında ITC dağıtılmış dosya sistemi [S+85] olarak isimlendirilmiştir) bazı temel tasarımlara sahipti,ancak istenildiği gibi ölçeklendirilmediğinden yeniden tasarlanmasına ve son protokole(AFSv2 ya da AFS olarak çağıracağımız) [H+88] yol açmış oldu. Şimdi ilk versiyonunu tartışıyoruz.

[1]Başlangıçta "Carnegie-Mellon Üniversitesi" olarak anılmasına ragmen,CMU daha sonra kısa çizgiyi kaldırdı ve böylece modern biçim olan "Carnegie Mellon Üniversitesi" doğdu. AFS 80'lerin başındaki çalışmalardan türetildiği için,CMU'yu original tam tireli biçimde ifade ediyoruz.Eğer sıkıcı,önemsiz ayrıntılar içinde bulunmak istiyorsanız https://www.quora.com/When-did-Carnegie-Mellon-University-remove-the-hyphen-in-the-university-name daha fazla detaya bakabilirsiniz.

TestAuth	Bir dosyanın değişip değişmediğini sınama (önbelleğe alınmış girişleri doğrulamak için kullanılr)
GetFileStat	Bir dosyanın istatistik bilgilerini alma
Fetch	Dosyanın içeriğini getirme
Store	Dosyayı sunucuda depolama
SetFileStat	Bir dosyanın istatistik bilgilerini ayarlama
ListDir	Bir dizinin içeriğini listeleme

Şekil 50.1: **AFSv1 Protokolünün Önemli Noktaları** (**AFSv1 Protocol Highlights)**

AFS'nin tüm sürümlerinin temel ilkelerinden birisi, bir dosyaya erişen istemci makinenin **yerel diskinde (local disk)** tam **dosya önbelleğine (whole-file caching)** alınmasıdır. Bir dosyayı açtığınızda (`open()`) dosyanın tamamı (mevcut ise) sunucudan alınır ve yerel diskinizdeki bir dosyaya depolanır. Sonraki uygulama `okuma` ve `yazma` (`read()` and `write()`) işlemleri, dosyanın depolandığı yerel dosya sistemine yönlendirilir; bu işlemler ağ iletişimi gerektirmez bu nedenle hızlıdır. Son olarak `kapatma` üzerine (`close()`),dosya (değiştirilmişse) sunucuya geri gönderilir. *Blokları (blocks)* önbelleğe alan (NFS elbette tüm dosyanın her bloğunu önbelleğe alabilse de tüm dosyaları alamamaktadır) ve bunu istemci *belleğinde* (*memory*) (yerel disk değil) yapan NFS ile bariz zıtlıklarına dikkat ediniz.

Daha fazla detaya girecek olursak; Bir istemci uygulaması `open()` öğesini ilk kez çağırdığında AFS istemci-tarafı kodu (AFS tasarımcıları **Venus** olarak adlandırıyor) sunucuya protokolü getirme iletisini gönderir. Protokolü getirme iletisi istenen dosyanın tüm yol adını (örneğin; `/home/remzi/notes.txt`) dosya sunucusuna (**Vice** adını verdikleri grup) gönderir, yol adını geçtikten sonra istenen dosyayı bulur ve tüm dosyayı istemciye geri gönderir. İstemci-tarafı kodu daha sonra dosyayı istemcinin yerel diskinde önbelleğe alır (yerel diske yazarak). Yukarıda da belirttiğimiz gibi sonraki `okuma` (`read()`) ve `yazma(write())` sistem çağrıları AFS'de kesinlikle yereldir(sunucuyla iletişim kurulmaz); sadece dosyanın yerel kopyasına yönlendirilirler. `okuma` (`read()`) ve `yazma(write())` çağrıları tıpkı yerel bir dosya sistemine yapılan çağrılar gibi davrandığından, bir bloğa erişildikten sonra istemci belleğinde de önbelleğe alınabilir. Bu nedenle AFS, yerel diskinde bulunan blokların kopyalarını önbelleğe almak için istemci belleğini de kullanır. Son olarak iş bittiğinde AFS istemcisi dosyanın değiştirilip değiştirilmediğini kontrol eder(yani,yazma durumu için açılmıştır); Öyleyse, kalıcı depolama için tüm dosya ve yol adını sunucuya göndererek yeni sürümü bir mağaza(store) protokolü mesajı ile sunucuya geri gönderir.

Dosyaya bir sonraki erişimde, AFSv1 çok daha verimli bir şekilde çalışır. Özellikle, istemci-tarafı kodu, dosyanın değişip değişmediğini belirlemek için önce sunucuyla iletişim kurar(TestAuth protokol iletisini kullanarak). Aksi takdirde istemci yerel olarak önbelleğe alınan kopyayı kullanır ve böylece bir ağ aktarımından kaçınarak performansı artırır. Yukarıdaki şekil, AFSv1'deki bazı protokol mesajlarını göstermektedir.

Protokolün bu erken sürümünün yalnızca dosya içeriklerini önbelleğe aldığını unutmayın; örneğin, dizinler yalnızca sunucuda tutuluyordu.

İPUCU: ÖLÇÜN VE SONRA İNŞA EDİN PATTERSON YASASI (PATTERSON'S LAW)

Danışmanlarımızdan biri olan David Patterson, söz konusu sorunu çözmek için yeni bir sistem oluşturmadan önce bizi her zaman bir sistem ölçmeye ve bir sorunu göstermeye teşvik ederdi. İçgüdü yerine deneysel kanıtlar kullanarak, sistem kurma sürecini daha bilimsel bir çabaya dönüştürebilirsiniz. Bunu yapmak geliştirilmiş sürümünüzü yükseltmeden önce sistemi tam olarak nasıl ölçeceğinizi düşünmenizi sağlayan ek faydaları da vardır. Sonunda yeni sistem kurmaya başladığınızda sonuç(result) olarak iki durum daha iyidir: Birincisi, gerçek bir sorunu çözdüğünüzü gösteren kanıtlara sahipseniz; ikincisi, yeni sisteminizi yerinde ölçmenin en son teknolojiyi gerçekten iyileştirdiğini göstermenin bir yolunu buldunuz. Biz buna **Patterson Yasası (Patterson's Law)** diyoruz.

50.2 Sürüm(Version) 1 ile İlgili Sorunlar

AFS'nin ilk sürümüyle ilgili birkaç sorunu tasarımcıları tarafından dosya sistemlerini yeniden düşünmeye motive etmiş oldu. AFS tasarımcıları neyin yanlış olduğunu bulmak için mevcut prototipleri ölçmeye çok zaman harcadılar. Bu deneyim iyi bir şey çünkü **ölçüm (measurement)** sistemlerin nasıl çalıştığını ve bunların nasıl iyileştirileceğini anlamanın anahtarıdır. Bu nedenle somut ve iyi veri elde etmek sistem inşasının gerekli bir parçasıdır. Yazarlar çalışmalarında AFSv1 ile ilgili iki ana sorun buldular:

- **Yol geçiş maaliyetleri çok yüksek olması (Path-traversal costs are too high):** Bir Getir(Fetch) veya Sakla(Store) protokol isteği gerçekleştirilirken, istemci yol adının tamamını istemciye gönderir(örneğin, `/home/remzi/notes.txt`). Sunucu dosyaya erişmek için girilen yolun tamamını gezmek zorundadır. İlk olarak `home` olanı bulmak için önce kök dizinine bakar, sonra `remzi` 'yi bulmak için de `home`'a bakacaktır. Böylece istenen dosya bulunana kadar yolun sonuna kadar ilerlemesi gerekmektedir. Birçok istemcinin sunucuya aynı anda erişim sağlanmasıyla AFS tasarımcıları sunucunun işlemci(CPU) zamanının çoğunu sadece dizi yollarında yürümek için harcandığını fark ettiler.
- **İstemci çok fazla TestAuth protokol mesajı yayınlaması (The client issues too many TestAuth protocol messages)**: NFS ve GETATTR protokol mesajlarının fazlalığı gibi AFSv1 de bir yerel dosyasının (veya istatistik bilgilerinin) TestAuth protokol mesajıyla geçerli olup olmadığını kontrol etmek için yoğun bir trafik oluşturmuş olur. Bu nedenle sunucular zamanlarının çoğunu bir dosyanın önbelleğe alınmış kopyalarını kullanmanın daha uygun olup olmadığını söylemekle geçirdiler. Çoğu zaman

cevap dosyanın değişmediğiydi.

Aslında AFSV1 ile ilgili iki sorun daha vardı bunlar: İlk olarak yük, sunucular arasında dengelenmemişti ve sunucu her istemci başına tek bir ayrı işlem kullanıyordu, bu da içerik değiştirme ve diğer yüklere neden oluyordu. Yük dengesizlik sorunu, bir yöneticinin yükü dengelemek için sunucular arasında taşıyabileceği **birimler(volumes)** getirilerek çözüldü; İkini olarak bağlam anahtarı (context-switch) problemi, AFSv2'de sunucuyu işlemler(processes) yerine iş parçacıkları(Threads) oluşturarak çözdü. Bununla birlikte alan adına burada sistemin ölçeğini sınırlayan yukarıdaki iki ana protokol sorununa odaklanıyoruz.

50.3 Protokolün İyileştirilmesi

Yukarıdaki iki sorun AFS'nin ölçeklenebilirliğini sınırlamış oldu. Sunucu CPU'su sistemi darboğazı(bottleneck, çıkmaz yola girmek gibi) haline geldi ve her sunucu aşırı yüklenmeden(overloaded) yalnızca 20 istemciye hizmet verebildi. Sunucular çok fazla TestAuth mesajı alıyordu ve Getir(Fetch) veya Sakla(Store) mesajları aldıklarında dizin hiyerarşisinde gezinmek için çok fazla zaman harcıyorlardı. Böylece AFS tasarımcıları bir sorunla karşı karşıya kaldılar:

> Dönüm Noktası (The Crux) : ÖLÇEKLENEBİLİR BİR DOSYA PROTOKOLÜ NASIL TASARLANIR
>
> Sunucu etkileşimlerinin sayısını en aza indirek için protokol nasıl yeniden tasarlanmalıdır Yani TestAuth mesajlarının sayısı nasıl azaltılabilir? Ayrıca, bu sunucu etkileşimlerini verimli hale getirmek için protokolü nasıl tasarlayabilirler? Bu sorunların her ikisini de ele alan yeni bir protokol çok daha ölçeklenebilir bir AFS sürümüyle sonuçlanacaktır.

50.4 AFS Sürüm(Versiyon) 2

AFSv2, istemci/sunucu etkileşimlerinin sayısını azaltmak için **geri dönme (callback)** kavramını tanıttı. Geri arama (callback) sunucudan istemciye, istemcinin önbelleğe aldığı bir dosya değiştirildiğinde sunucunun istemciyi bilgilendireceğie dair basit bir sözdür. Bu **durumun (state)** sisteme eklenmesiyle, istemcinin önbelleğe(cached) bir dosyanın hala geçerli olup olmadığını öğrenmek için sunucuyla iletişime geçmesine gerek kalmaz. Bunun yerine sunucu aksini söyleyene kadar dosyanın geçerli olduğu varsayılır. **Yoklama(polling) ile kesmeler(interrupts)** arasındaki benzerliğe dikkat edin.

AFSv2 ayrıca bir istemcinin hangi dosyayla ilgilendiğini belirtmek için yol adları yerine bir **dosya tanımlayıcısı (file identifier (FID))** (NFS **dosya tanıtıcısına (file handle)** benzer) kavramını da tanıttı. AFS'deki bir dosya tanımlayıcısı (FID), bir dosya tanımlayıcısı ve bir

benzersizleştiriciden("uniquifier")(bir dosya silindiğinde birim ve dosya kimliklerinin yeniden kullanılmasını sağlamak için) oluşur. Böylece sunucuya tüm yol adlarını göndermek ve sunucunun istenen dosyayı bulmak için yol adını taramasına izin vermek yerine istemci yol adını her seferinde bir parça tarayacak, sonuçları önbelleğe(caching) alacak ve böylece sunucu üzerindeki yük azalacaktır.

Örneğin bir istemci `/home/remzi/notes.txt` dosyasına erişiyorsa ve `home` istemciye bağlı AFS dizini ise istemci önce `home` dizisinin içeriğini getirir, bunları yerel disk(local-disk) önbelleğine(cache) koyar ve `home` üzerinden bir geri arama(callback) yapar. Sonrasında, istemci dizini getirir(Fetch).

İstemci Client (C_1)	**Sunucu (Server)**
fd = open("/home/remzi/notes.txt", ...);	
Fetch Gönder (home FID, "remzi")	
	Getirme (Fetch) isteği al remzi'yi 'home dir' dizininde ara remzi'de geri dönüt (callback(C1)) oluştur ve remzi'nin içeriğini ile FID'yi (Dosya tanımlayıcısı) geri dönüt yap.
Getirme(Fetch) yanıtını al remzi'yi yerel disk önbelleğine yaz ve remzi'nin geri arama durummunu kaydet. Fetch Gönder (remzi FID, "notes.txt")	
	Getirme(Fetch) isteğini 'remzi dir' de notes.txt dosyasında ara bu dosyada geri dönüt (callback(C1)) oluştur ve notes.txt dosyasının içeriğini ve FID'yi (Dosya tanımlayıcısı) geri dönüt yap.
Getirme(Fetch) yanıtını al notes.txt'yi yerel disk önbelleğine yaz ve bu txt'yi geri arama durumunu kaydet. Önbelleğe alınan notes.txt'nin yerel `open()` işlevi ile dosya tanımlayıcısını uygulamaya döndür.	
read(fd, buffer, MAX);	
önbelleğe alınmış kopyada yerel okuma (`read()`) gerçekleştirme	
close(fd);	
dosya değiştiyse önbelleğe alınmış kopyada yerel kapatma (`close()`) yapın ve sunucuyu boşaltın.	
fd = open("/home/remzi/notes.txt", ...);	

Foreach dir (home, remzi)
if (callback(dir) == VALID)
arama için yerel kopyayı kullan (dir)
else
Getir (Fetch) (yukarıdaki gibi)
if (callback(notes.txt) == VALID)
önbelleğe alınmış yerel kopyayı aç dosya tanımlayıcısını ona döndür (return)
else
Getir (Fetch) (yukarıdaki gibi) sonrasında aç ve fd'yi geri dönder (return)	

Şekil 50.2: **Dosya Okuma : İstemci Tarafı ve Dosya Sunucusu Eylemleri (Reading A File: Client-side And File Server Actions)**

`remzi`, yerel dis önbelleğine yerleştirin ve `remzi` üzerinde bir geri arama(callback) yapın. Son olarak, istemci `notes.txt`'yi getirecek, bu normal dosyayı yerel diskte önbelleğe alacak, bir geri arama (callback) ayarlayacak ve son olarak çağıran uygulamaya bir dosya tanımlayıcısı döndürecektir. Özet için Şekil 50.2'ye bakınız. Bununla birlikte, NFS'den temel fark bir dizinin veya dosyanın her getirilmesinde, AFS istemcisinin sunucuyla bir geri arama oluşturmasdır.

BİR TARAFTAN: ÖNBELLEK TUTARLILIĞI HER SORUNA ÇÖZÜM DEĞİLDİR

Dağıtılmış dosya sistemlerini tartışırken, dosya sistemlerinin sağladığı önbellek sürekliliğinden çok şey yapılır. Ancak, bu temel tutarlılık, birden çok istemciden dosya erişimiyle ilgili tüm sorunları çözmez. Örneğin, birden fazla istemcinin kodu teslim etme ve teslim alma işlemlerini gerçekleştirdiği bir kod deposu oluşturuyorsanız, tüm işi sizin yerinize yapası için temeldeki dosya sistemine güvenemezsiniz. Bunun yerine, bu tür eşzamanlı erişimler gerçekleştirdiğinde "doğru" şeyin gerçelkleşmesini sağlamak için açık **dosya düzeyinde kilitleme (file-level locking**) kullanmanız gerekir. Aslında, eşzamanlı güncellemeleri gerçekten önemseyen herhangi bir uygulama, çakışmaların üstesinden gelebilmek için ekstra makineler ekleyecektir. Bu bölümde ve bir öncekinde açıklanan temel tutarlılık, öncelikle gündelik kullanım için yararlıdır örneğin, bir kullanıcı farklı bir istemcide oturum açtığında, dosyalarının makul bir sürümünün orada görünmesini bekler. Bu protokollerden daha fazlasını beklemek kendinizi başarısızlık, hayal kırıklığı ve gözyaşı dolu daha fazla kırgınlığa hazırlamaktadır.

Böylece sunucunun önbelleğe alınmış durumundaki bir değişikliği istemciye bildirmesini sağlar. Faydası açıktır: `/home/remzi/notes.txt` dosyasına *ilk erişim (first access*) birçok istemci-sunucu mesajı oluştursa da (yukarıda açıklandığı gibi), aynı zamanda tüm dizinler ve notes.txt dosyası için geri çağırımlar (callback) oluşturur ve böylece sonraki erişimler tamamen yereldir ve hiçbir sunucu etkileşimi gerektirmez. Bu nedenle, bir dosyanın istemcide önbelleğe alındığı yaygın durumda, AFS yerel disk tabanlı bir dosya sistemiyle neredeyse aynı şekilde davranır. Bir dosyaya bieden fazla kez erişiliyorsa, ikinci erişim yerel olarak bir dsoyaya erişmek kadar hızlı olmalıdır.

50.5 Önbellek Tutarlılığı (Cache Consistency)

NFS'den bahsederken, önbellek tutarlılığının göz önünde bulundurduğumuz iki yönü vardı: **Görünürlüğü güncelleme (update visibility)** ve **önbellek bayatlığı (cache staleness)** Güncelleme görünürlüğü(update visibility) ile ilgili soru şudur: Sunucu ne zaman bir dosyanın yeni bir sürümüyle güncelleştirilir? Önbellek bayatlığ(cache staleness) ile ilgili soru şu şekildedir: Sunucu yeni bir sürüme sahip olduğunda, istemciler önbelleğe alınmış eski bir kopya yerine yeni sürümü görmeleri ne kadar sürer?

Geri armalar(callback) ve tüm dosyayı önbelleğe alma nedeniyle, AFS tarafından sağlanan önbellek tutarlılığının tanımlanması ve anlaşılması kolaydır. Dikkate alınması gereken iki önemli durum vardır: *Farklı(different)* makinelerdeki süreçler arasındaki tutarlılık ve *aynı(same)* makinedeki süreçler(processes) arasındaki tutarlılık.

Farklı makineler arasında AFS, güncellemeleri sunucuda görünür hale getirir ve önbelleğe alınmış kopyaları aynı anda, yani güncellenmiş dosya kapatıldığında geçersiz kılar. İstemci bir dosyayı açar ve ardından ona yazar (belki de art arda). Sonunda kapatıldığında, yeni dosya sunucuya aktarılır(temizlenir) (ve böylece görünür olur). Bu noktada,sunucu önbelleğe alınmış kopyaları olan tüm istemciler için geri aramaları(callbacks) "keser"(break). Kesmek veya diğer anlamıyla mola(break), her bir istemciyle iletişime geçerek ve dosyadaki geri aramanın(callback) artık geçerli olmadığını bildirerek gerçekleştirilir.

İstemci(Client)$_1$			**İstemci Client$_2$**		**Server**	Yorumlar *Comments*
P_1	**P_2**	**Cache**	**P_3**	**Cache**	**Disk**	
open(F)		-		-	-	Dosya oluşturuldu
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Yerel süreçler(processes)
	read() → B	B		-	A	hemen yazıları gör
	close()	B		-	A	
		B	open(F)	A	A	Uzak işlemler yazmaları
		B	read() → A	A	A	görmeyin…
		B	close()	A	A	
close()		B		✚	B	… close() işlevine kadar
		B	open(F)	B	B	gerçekleşmiştir.
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		C	D	
		D	open(F)	D	D	Ne yazık ki P3 için
		D	read() → D	D	D	son vazan kazanır.
		D	close()	D	D	

Şekil 50.3: **Önbellek Tutarlılığı Zaman Çizelgesi** (**Cache Consistency Timeline)**

Bu adım, istemcilerin artık dosyanın eski kopyalarını okumasını sağlayacaktır; bu istemcilerde sonraki açmalar, dosyanın yeni sürümünün sunucudan yeniden getirilmesini gerektirecektir (ve ayrıca dosyanın yeni sürümünde yeniden bir geri arama oluşturmaya da

hizmet edecektir).

AFS, aynı makinedeki işlemler arasındaki bu basit modele bir istisna yapar. Bu durumda, bir dosyaya yazılanlar diğer yerel işlemler tarafından hemen görülebilir (yani,bir işlemin en son güncellemelerini görmek için bir dosyaya kapatılana kadar beklemesi gerekmez). Bu, tek bir makine kullanımının tam olarak beklediğimiz gibi davranmasını sağlar, çünkü bu davranış tipik bir UNIX semantiğine dayanır. Sadece farklı bir makineye geçtiğinizde daha genel AFS tutarlılık mekanizmasını tespit edebilirsiniz.

Daha fazla tartışmaya değer ilginç bir makineler arası durum vardır. Özellikle, farklı makinelerdeki işlemlerin bir dosyayı aynı anda değiştirdiği nadir durumlarda, AFS doğal olarak **son yazan kazanır(last writer wins)** yaklaşımı olarak bilinen yaklaşımı kullanır(belki de **son yaklaşan kazanır(last closer wins)** olarak adlandırılmalıdır). Özellikle, hangi istemci `close`() işlevini en son çağırırsa, sunucudaki tüm dosyayı en son güncelleyecek ve böylece kazanan olacaktır.

Dosya, yani başkalarının görmesi için sunucuda kalan dosya. Sonuç, bir istemci veya diğeri tarafından bütünüyle oluşturulmuş bir dosyadır. NFS gibi blok tabanlı bir protokolden farkına dikkat edin: NFS'de, her istemci dosyayı güncellerken ayrı blokların yazma işlemleri sunucuya aktarılabilir ve bu nedenle sunucudaki son dosya, her iki istemciden gelen güncellemelerin bir karışımı olarak sonuçlanabilir. Çoğu durumda, böyle karışık bir dosya çıktısı pek bir anlam ifade etmeyecektir, yani iki istemci tarafından parçalar halinde değiştirilen bir JPEG görüntüsü hayal edin bu durumda ortaya çıkan yazım karışımı muhtemelen geçerli bir JPEG oluşturmayacaktır.

Bu farklı senaryolardan birkaçını gösteren bir zaman çizelgesi Şekil 50.3'te gösterilmiştir. Sütunlar istemci1'deki(Client1) iki sürecin(P_1 ve P_2)(processes) ve önbellek durumunun, istemci2'deki(Client2) bir sürecin (P_3)(process) ve önbellek durumunun aynı zamanda sunucu davranışını göstermektedir. Bunların hepsi hayali olarak `F` diye adlandırılan tek bir dosya üzerinde çalışmaktadır. Sunucu için, şekil soldaki işlem tamamlandıktan sonra dsoyanın içeriğini gösterir. Okuyun ve her okumanın neden bu sonuçları verdiğini anlayıp anlamadığınıza bakın. Sağ taraftaki yorum alanı takıldığınız yerde size yardımcı olacaktır.

50.6 Kilitlenme(Bir diğer deyişle çökme) Kurtarması (Crash Recovery)

Yukarıdaki açıklamadan, kilitlenme kurtarmanın (crash recovery) işleminin NFS'ye göre daha karmaşık olduğunu düşünebilirsiniz. Haklısınız. Örneğin, bir sunucunun(S) bir istemciye (C1) iletişim kuramadığı kısa bir süre olduğunu düşünün, mesela C1 istemcisinin yeniden başlatılma durumu gibi. C1 mevcut değilken, S ona bir veya daha fazla geri çağırma(callback) mesajı göndermeyi denemiş olabilir; örneğin, C1'in F dosyasını kendi yerel diskinde önbelleğe aldığını ve ardından C2'nin (başka bir istemci) F'yi güncelleştirdiğini, böylece S'nin dosyayı önbelleğe alan tüm istemcilere yerel önbelleklerinden kaldırmak için ileti göndermesine neden olduğunu düşünün. C1 yeniden başlatılırken bu kritik mesajları kaçırabileceğinden, sisteme yeniden katıldığında C1 tüm önbellek içeriğini şüpheli olarak değerlendirmelidir. Bu nedenle, F dos-

yasına bir sonraki erişimde C1 önce sunucuya (TestAuth protokol mesajıyla) F dosyasının önbelleğe alınmış kopyasının hala geçerli olup olmadığını sormalıdır. Eğer öyleyse, C1 bunu kullanabilir; değilse, C1 yeni sürümü sunucudan getirmelidir.

Bir çökmeden sonra sunucu kurtarma daha da karışıktır. Ortaya çıkan sorun, geri çağırımların(callbacks) bellekte(memory) tutulmasıdır; bu nedenle, bir sunucu yeniden başlatıldığında, hangi istemci makinesinin hangi dosyalara sahip olduğu hakkında hiçbir fikri yoktur. Bu nedenle, sunucu yeniden başlatıldığında, sunucunun her istemcisi sunucunun çöktüğnü fark etmeli ve tüm önbellek içeriklerini şüpheli olarak değerlendirmeli (yukarıdaki gibi) aynı zamanda kullanmadan önce bir dosyanın geçerliliğini yeniden sağlamalıdır. Bu nedenle, sunucu çökmesi büyük bir olaydır, çünkü her bir istemcinin çökmeden zamanında haberdar olmasını sağlamak veya bir istemcinin eski bir dosyaya erişmesi riskini almak gerekir. Bu tür bir kurtarmayı uygulamanın birçok yolu vardır; örneğin, sunucu yeniden çalışmaya başladığında her istemciye bir mesaj ("önbellek içeriğinize güvenmeyin!") göndermesini sağlayarak veya istemcilerin sunucunun hayatta olup olmadığını periyodik kontrol etmesini sağlayarak (buna **kalp atışı(heartbeat)** mesajı denir). Gördüğünüz gibi, daha ölçeklenebilir ve mantıklı bir önbelleğe alma modeli oluşturmanın bir maliyeti vardır; NFS ile istemciler sunucunun çöktüğünü neredeyse hiç fark etmediler.

İş Yükü(Workload)	NFS	AFS	$\frac{\text{AFS}}{\text{NFS}}$
1. Küçük dosya, sıralı okuma	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Küçük dosya, sıralı yeniden okuma	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Orta boy dosya, sıralı okuma	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Orta boy dosya, sıralı yeniden okuma	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Büyük dosya, sıralı okuma	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Büyük dosya, sıralı yeniden okuma	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Büyük dosya, tekli okuma	L_{net}	$N_L \cdot L_{net}$	N_L
8. Küçük dosya, sıralı yazma	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Büyük dosya, sıralı yazma	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Büyük dosya, sıralı üzerine yazma	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Büyük dosya, tekli yazma	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Şekil 50.4: **Karşılaştırma (Comparison): AFS vs. NFS**

50.7 AFSv2'nin Ölçeği ve Performansı (Scale And Performance Of AFSv2)

Yeni protokol uygulandığında, AFSv2 ölçüldü ve orijinal sürümünden çok daha ölçeklenebilir olduğu bulundu. Gerçekten de, her sunucu yaklaşık 50 istemciyi destekleyebilir (sadece 20 yerine). Diğer bir fayda da istemci tarafı performansının genellikle yerel performansa oldukça yakın olmasıydı çünkü, genel durumda tüm dosya erişimleri yereldi; dosya okumaları genellikle yerel disk önbelleğine (ve potansiyel olarak yerel belleğe) gitmektedir. Yalnızca bir istemci yeni bir dosya oluşturduğunda veya mevcut bir dosyaya yazdığında sunucuya bir Depolanan mesajı (Store message) gönderilmesi ve böylece dosyanın yeni içeriklerle güncellemesi gerekiyordu. Ayrıca, yaygın dosya sistemi erişim senaryolarını NFS ile karşılaştırarak AFS performansına biraz bakış açısı kazandıralım Şekil 50.4

(sayfa 10). Nitelikleri karşılaştırmamızın sonuçlarını göstermektedir.

Şekilde, farklı boyutlardaki dosyalar için tipik okuma ve yazma modellerini analitik olarak inceliyoruz. Küçük dosyaların içinde N_s blokları vardır, orta boydaki dosyalar da N_m bloklarına sahiptir. Küçük ve orta ölçekli dosyaların bir istemcinin belleğine sığdığını varsayıyoruz; büyük dosyalar yerel diske sığıyor ancak istemci belleğine sığmıyor.

Ayrıca, analiz amacıyla bir dosya bloğu için ağ üzerinden uzak sunucuya erişimin L_{net} zaman birimi sürdüğünü varsayıyoruz. Yerel belleğe erişimi L_{mem} alır ve yerel diske erişim ise L_{disk} alır. Genel varsayım $L_{net} > L_{disk} > L_{mem}$ şeklindedir.

Son olarak, bir dosyaya ilk erişimin herhangi bir önbelleğe isabet etmediğini varsayıyoruz. İlgili önbellek dosyayı tutmak için yeterli kapasiteye sahipse, sonraki dosya erişimleri (yani, "yeniden okumalar") önbelleklerde isabet edeceğini varsayıyoruz.

Şeklin sutünları, belirli bir işlemin (örneğin, küçük bir dosya okuması) kabaca NFS veya AFS'de aldığı süreyi gösterir. En sağdaki sütun, AFS'nin NFS'ye oranını gösterir. Aşağıdaki gözlemleri yapıyoruz. Ilk Olarak, birçok durumda ger sistemin performansı kabaca eşdeğerdir. Örneğin, bir dosyayı ilk kez okurken (örneğin, İş Yükü 1,3,5), dosyayı uzak sunucudan getirme süresi baskındır ve her iki sistemde de benzerdir.

Dosyayı yerel diske yazması gerektiğinden AFS'nin bu durumda daha yavaş olacağını düşünebilirisiniz. Bu yazmalar yerel (istemci tarafı) dosya sistemi önbelleği tarafından arabelleğe alınır ve bu nedenle söz konusu maliyetler büyük olasılıkla gizlenmiş olur. Benzer şekilde, AFS'nin yerel önbelleğe alınmış kopyadan okumaların daha yavaş olacağını düşünebilirsiniz, çünkü AFS önbelleğe alınmış kopyayı diskte depolar. Ancak, AFS burada yine yerel dosya sistemi önbelleğe alma özelliğinden yararlanır. AFS'deki okumalar büyük olasılıkla istemci tarafı bellek önbelleğine ulaşacak ve performans NFS'ye benzer olacaktır.

İkinci olarak, büyük bir dosyanın sıralı olarak yeniden okunması (İş Yükü 6) sırasında ilginç bir fark ortaya çıkmaktadır. AFS büyük bir yerel disk önbelleğine sahip olduğundan, dosyaya yeniden erişildiğinde dosyaya oradan erişir. Buna karşın NFS, yalnızca istemci belleğindeki blokları önbelleğe alır; sonuç olarak, büyük bir dosya (yani, yerel bellekten daha büyük bir dosya) yeniden okunursa, NFS istemcisinin tüm dosyayı uzak sunucudan yeniden getirmesi gerekir. Bu nedenle, AFS bu durumda NFS'den $\frac{Lnet}{Ldisk}$ faktörü ile daha hızlıdır. Uzaktan erişimin gerçekten daha yavaş olduğunu varsayarsak yerel diskten daha fazla olmuş olur.

Bu durumda NFS'nin sunucu yükünü artırdığını ve bunun da ölçek üzerinde bir etkisi olduğunu da not ediyoruz.

Üçüncü olarak, sıralı yazma işlemlerin (yeni dosyaların) her iki sistemde de benzer şekilde gerçekleştirilmesi gerektiğini not ediyoruz (İş Yükleri 8,9). AFS, bu durumda dosyayı yerel önbelleğe alınış kopyaya yazacaktır; dosya kapatıldığında, AFS istemcisi protokole göre sunucuyu yazmaya zorlar. NFS, istemci tarafındaki bellek baskısı nedeniyle bazı blokları sunucuya zorlayarak istemci belleğindeki yazmaları arabelleğe alacaktır, ancak NFS kapatıldığında temizleme tutarlılığını korumak için dosya kapatıldığında bunları kesinlikle sunucuya yazacaktır. Tüm verileri yerel diske yazdığı için AFS'nin burada daha yavaş olacağını düş-

ünebilirsiniz. Ancak, yerel bir dosya sistemine yazdığının farkında olunuz; bu yazma işlemleri önce sayfa önbelleğine ve daha sonra (arka planda) diske işlenir ve bu nedenle AFS, performansı artırmak için istemci tarafı işletim sistemi belleğini önbelleğe alma altyapısının avantajlarından yararlanır.

Dördüncü olarak, AFS'nin sıralı dosya üzerine yazmada(İş Yükü 10) daha kötü performans gösterdiğini not ediyoruz. Şimdiye kadar yazan iş yüklerinin de yeni bir dosya oluşturduğunu varsaydık; bu durumda, dosya vardır ve üzerine yazılır. Üzerine yazma, AFS için özellikle kötü bir durum olabilir, çünkü istemci önce eski dosyayı bütünüyle getirir, sonra üzerine yazar. Buna karşılık NFS, basitçe blokların üzerine yazacak ve böylece ilk okuma(read[2]) olacak (işe yaramaz).

Son olarak, büyük dosyalar içindeki verilerin küçük bir alt kümesine erişen iş yükleri NFS'de AFS'ye göre çok daha iyi performans gösterir (İş Yükleri 7,11). Bu durumlarda, AFS protokolü dosya açıldığında dosyanın tamamını alır; ne yazık ki, yalnızca küçük bir okuma veya yazma işlemi gerçekleştirilir. Daha kötüsü, dosya değiştirilirse tüm dosya sunucuya geri yazılır ve performans etkisi iki katına çıkar.

Bi Taraftan: İş Yükünün Önemi

Herhangi bir sistemi değerlendirmenin bir zorluğu, **iş yükünün(workload)** seçimidir. Bilgisayar sistemleri çok farklı şekillerde kullanıldığından, aralarından sçim yapabileceğiniz çok çeşitli iş yükleri vardır. Makul tasarım kararları almak için depolama sistemi tasarımcısı hangi iş yüklerinin önemli olduğuna nasıl karar vermelidir?

AFS tasarımcıları, dosya sistemlerinin nasıl kullanıldığını ölçme konusundaki deneyimlerini göz önünde bulundurarak, belirli iş yükü varsayımlarında bulundular. Özellikle, çoğu dosyanın sık sık paylaşılmadığını ve dosyaların tamamına sıralı olarak erişildiğini varsaymışlardır. Bu varsayımlar göz önüne alındığında, AFS tasarımı mükemmel bir anlam ifade etmektedir.

Ancak bu varsayımlar her zaman doğru değildir. Örneğin, bilgileri periyodik olarak bir günlüğe ekleyen bir uygulama düşünün. Mevcut büyük bir dosyaya küçük miktarlarda veri ekleyen bu küçük günlük yazmaları AFS için oldukça sorunludur. Diğer birçok zor iş yükü de vardır örneğin, bir işlem veritabanında rastgele güncellemeler gibi.

Ne tür iş yüklerinin yaygın olduğu hakkında bilgi edinebileceğiniz bir yer, yapılan çeşitli araştırma çalışmalarıdır. AFS retrospektifi [H+88] de dahil olmak üzere, iş yükü analizinin iyi örnekleri için bu çalışmalardan herhangi birine bakın [B+91, H+11, R+00, V99].

[2]Burada NFS yazmalarının blok boyutlu ve blok hizalı olduğunu varsayıyoruz; Eğer böyle olmasaydı, NFS istemcisinin de önce bloğu okuması gerekirdi. Ayrıca dosyanın O TRUNC bayrağı ile açılmadığını varsayıyoruz; eğer öyle olsaydı, AFS'deki ilk açılış yakında kesilecek olan dosyanın içeriğini getirmezdi.

NFS, blok tabanlı bir protokol olarak, okuma veya yazma boyutuyla orantılı G/Ç gerçekleştirilir.

Genel olarak, NFS ve AFS'nin farklı varsayımlarda bulunduklarını ve bunun sonucunda şaşırtıcı olmayan bir şekilde farklı performans sonuçları elde ettiklerini görüyoruz. Bu farklılıkların önemli olup olmadığı, her zaman olduğu gibi bir iş yükü meselesidir.

50.8 AFS: Diğer İyileştirmeler (Other Improvements)

Berkeley FFS'nin (sembolik bağlantılar ve bir dizi başka özellik ekledi) tanıtımında gördüğümüz gibi, AFS'nin tasarımcıları sistemlerini oluştururken sistemin kullanımını ve yönetimini kolaylaştıran bir dizi özellik ekleme fırsatını yakaladılar. Örneğin, AFS istemcilere gerçek bir global isim alanı sağlar, böylece tüm dosyaların tüm istemci makinelerde aynı şekilde adlandırılmasını sağlamaktadır. NFS, aksine her istemcinin NFS sunucularını istedileri şekilde bağlamasına izin verir ve bu nedenle, dosyalar istemciler arasında yalnızca geleneksel (ve büyük bir idari çabayla) benzer şekilde adlandırılabilir.

AFS ayrıca güvenliği de ciddiye alır ve kullanıcıların kimliklerini doğrulamak aynı zamanda bir kullanıcı isterse bir dizi dosyanın gizli tutulabilmesini sağlamak için mekanizmalar içermektedir. Buna karşın NFS, uzun yıllar boyunca güvenlik için oldukça ilkel bir desteğe sahipti. AFS ayrıca kullanıcı tarafından yönetilen esnek erişim kontrolü için olanaklar da içerir. Bu nedenle AFS kullanırken, bir kullanıcı tam olarak kimin hangi dosyalara erişebileceği konusunda büyük bir kontrole sahiptir.

NFS çoğu UNIX dosya sisemi gibi bu tür paylaşım için çok daha az desteğe sahiptir.

Son olarak, daha önce de belirtildiği gibi AFS, sistemin yöneticileri için sunucuların daha basit bir şekilde yönetilmesini sağlayacak araçlar eklemektedir. AFS, sistem yönetimini düşünürken bu alanda ışık yılı ilerideydi.

50.9 Özet (Summary)

AFS bize dağıtılmış dosya sistemlerinin NFS ile gördüğümüzden oldukça farklı bir şekilde oluşturabileceğini göstermektedir. AFS'nin protokol tasarımı özellikle önemlidir; sunucu etkileşimlerini en aza indirerek (tüm dosyayı önbelleğe alma (cache) ve geri aramalar (callbacks) yoluyla), her sunucu birçok istemciyi destekleyebilir ve böylece belirli bir siteyi yönetmek için gereken sunuu sayısını azaltabilir. Tek isim alanı, güvenlik ve erişim kontrol listeleri de dahil olmak üzere diğer birçok özellik AFS'nin kullanımını oldukça güzel hale getirir. AFS tarafından sağlanan tutarlılık modelinin anlaşılması ve üzerinde akıl yürütmesi basittir. NFS'de bazen gözlemlenen gibi ara sıra ortaya çıkan tuhaf davranışlara yol açmamaktadır.

Ne yazıktır ki AFS muhtemelen düşüştedir. Çünkü NFS açık bir standart haline geldiğinden, birçok farklı satıcı bunu destekledi ve CIFS

(Windows tabanlı dağıtılmış dosya sistemi protokolü) ile birlikte NFS piyasada hakim oldu. Her ne kadar zaman zaman AFS kurulumları görülse de (Wisconsin dahil çeşitli eğitim kurumlarında olduğu gibi), tek kalıcı etki muhtemelen gerçek sistemin kendisinden ziyade AFS'nin fikirlerinden kaynaklanacaktır. Gerçekten de , NFSv4 artık sunucu durumu ekliyor (örneğin, "açık" bir protokol mesajı) ve bu nedenle temel AFS protokolüne giderek daha fazla benzerlik göstermektedir.

Referanslar (References)

[B+91] "Measurements of a Distributed File System" by Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout. SOSP '91, Pacific Grove, California, October 1991. *An early paper measuring how people use distributed file systems. Matches much of the intuition found in AFS.*

[H+11] "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications" by Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. SOSP '11, New York, New York, October 2011. *Our own paper studying the behavior of Apple Desktop workloads; turns out they are a bit different than many of the server-based workloads the systems research community usually focuses upon. Also a good recent reference which points to a lot of related work.*

[H+88] "Scale and Performance in a Distributed File System" by John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West. ACM Transactions on Computing Systems (ACM TOCS), Volume 6:1, February 1988. *The long journal version of the famous AFS system, still in use in a number of places throughout the world, and also probably the earliest clear thinking on how to build distributed file systems. A wonderful combination of the science of measurement and principled engineering.*

[R+00] "A Comparison of File System Workloads" by Drew Roselli, Jacob R. Lorch, Thomas E. Anderson. USENIX '00, San Diego, California, June 2000. *A more recent set of traces as compared to the Baker paper [B+91], with some interesting twists.*

[S+85] "The ITC Distributed File System: Principles and Design" by M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West. SOSP '85, Orcas Island, Washington, December 1985. *The older paper about a distributed file system. Much of the basic design of AFS is in place in this older system, but not the improvements for scale. The name change to "Andrew" is an homage to two people both named Andrew, Andrew Carnegie and Andrew Mellon. These two rich dudes started the Carnegie Institute of Technology and the Mellon Institute of Industrial Research, respectively, which eventually merged to become what is now known as Carnegie Mellon University.*

[V99] "File system usage in Windows NT 4.0" by Werner Vogels. SOSP '99, Kiawah Island Resort, South Carolina, December 1999. *A cool study of Windows workloads, which are inherently different than many of the* UNIX*-based studies that had previously been done.*

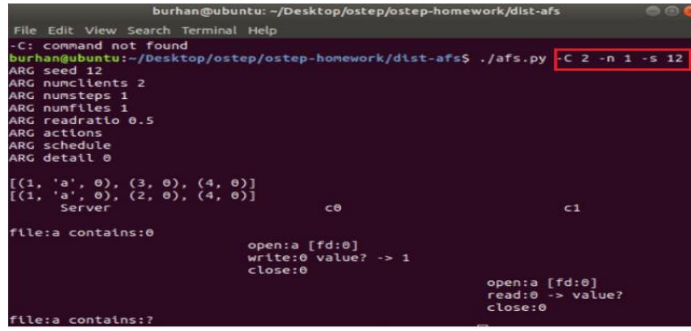
Ev Ödevi (Simülasyon) (Homework) (Simulation)

Bu bölümde, Andrew Dosya Sisteminin nasıl çalıştığına dair bilginizi pekiştirmek için kullanabileceğiniz basit bir AFS simülatörü olan afs.py'yi tanıtmaktadır. Daha fazla ayrıntı için README dosyasını okuyunuz.

Sorular (Questions)

1. İstemciler tarafından hangi değerlerin okunacağını tahmin edebileceğinizden emin olmak için birkaç basit servis talebi çalıştırın. Rastgele çekirdek bayrağını (-s) değiştirin ve hem ara değerleri hem de dosyalarda depolanan son değerleri izleyip tahmin edip edemeyeceğinizi görün. Ayrıca biraz daha zorlu hale getirmek için dosya sayısını (-f), istemci sayısını (-C) ve okuma oranını (-r,0 ile 1 arasında) değiştirin. Daha ilginç etkileşimler elde etmek için biraz daha uzun izler oluşturmak da isteyebilirsiniz, örneğin (-n 2 veya daha yüksek) gibi değerler. Basit servis talebi çalıştırılması istenmektedir.

   Yapılan işlemde 2 adet istemci (-C 2 diyerek) kullanıyoruz. (-n 1) diyerek de her istemcinin yapacağı işlem sayısını belirtiyoruz burada 1 belirlemiş olduk. İlk başta bizden çekirdek bayrağını (diğer bir deyişle tohum) değiştirerek tahmin etmemizi istediği için -s 12 diyerek tohum (seed) değerini 12 vermiş olduk. Çıktıyı inceleyecek olursak da sol tarafta bir sunucu (server) bulunmaktadır. Yan taraflarında da sütunlar boyunca girdiğimiz istemci (client) sayısı yazmaktadır (c0 ve c1 gibi). Bu izde görebileceğiniz gibi, f değerini değiştirmediğimiz için yalnızca bir dosya (a) vardır ve 0 değerini içerir (file: a contains :0).



Şekil 1.0

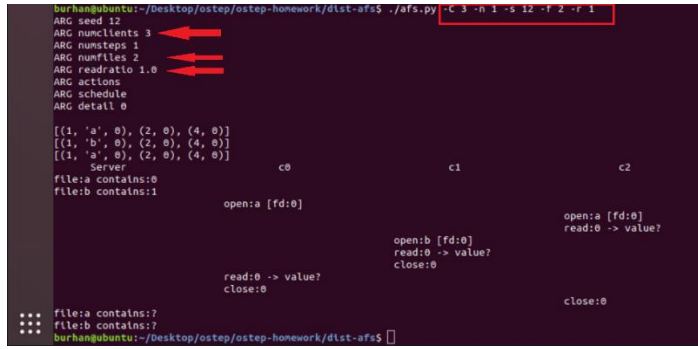
Aynı şekilde (-s) değerini 10 yaparsak da şu şekilde bir çıktı elde etmiş oluruz.

Bu çıktıda önce ikinci istemci (c1) açma okuma işlemleri yapıyor ardından birinci istemciye (c0) dönmüş oluyor.

Şekil 1.1

Soru bizden daha zorlu hale getirmemiz için dosya sayısını (-f), istemci (-C) ve okuma oranını (-r, 0 ile 1 arasında) değiştirmemizi istiyor bunu aynı değerlerin yanında istemci sayısını 3'e çıkararak (-C 3), dosya sayısını 2'ye (-f 2) ve okuma oranını 1 yaparak (-r 1) tekrar çalıştırıyoruz.

Bu şekilde a ve b olmak üzere 2 adet dosya (file) olduğunu görmüş olduk.Birincisine (a) 0 değeri yüklenirken ikincisine (b) 1 değeri yükleniyor. Aynı zamanda 3 adet istemci de eklenmiş oldu (c0, c1, c2).



Şekil 1.2

Daha ilginç etkileşimler elde etmek için istemcinin yapacağı işlemleri (-n) artırmamız istenmektedir. Normal şartlar altında varsayılan olarak 1 değerini yüklemektedir biz bunu 2 için denersek şu şekilde bir sonuç elde etmiş oluruz.

Normalde istemcilerimiz tek bir açmadan (open) sonra işlemleri yapıp bitirirken bu sefer işlem sayısını 2 yaptığımız için her bir istemci iki adet açma (open) işlemi yapmaktadır.

Şekil 1.3

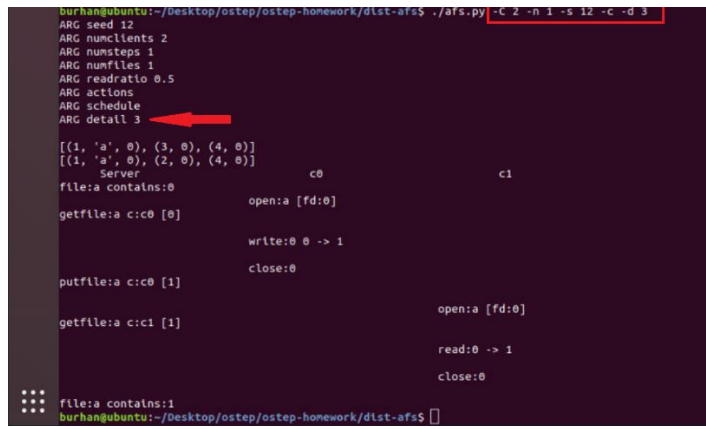
2. Şimdi aynı şeyi yapın ve AFS sunucusunun başlattığı her geri dönüşü (callback) tahmin edip edemeyeceğinize bakın. Farklı rastgele çekirdekleri (seeds) deneyin ve programın yanıtları sizin için hesaplamasını sağladığınızda (-c ile) geri aramaların (callback) ne zaman gerçekleştiğini görmek için yüksek düzeyde ayrıntılı geri bildirim (örneğin `-d 3`) kullandığınızdan emin olunuz. Her geri aramanın (callback) tam olarak ne zaman gerçekleştiğini tahmin edebiliyor musunuz? Birinin gerçekleşmesi için kesin koşul nedir?

Farklı rastgele çekirdekleri (seeds) oluşturmak için iki farklı durumu ele alıyoruz bunlar (-s 12) ve (-s 10) değerleridir. Bunları denediğimizde dosyaya yüklenen son değeri görebilmemiz için (-c) komutunu kullanmamız gerekiyor. Bu (-c) komutu ile (-s 12) ve (-s 10) arasındaki farkı inceleyelim. Her iki şekilde de (Şekil 2.0 ve Şekil 2.1) dosyaya 'a' değeri 0 ile yüklenmektedir (file: a contains: 0). Şekil 2.0'da ilk istemci (c0) 'a' dosyasını açıyor (open) üstüne 1 yazıyor (write) ve ardından kapatmaktadır (close). Sıra ikinci istemciye geldiğinde (c1) 'a' dosyasını açıyor (open) okumaya başlıyor (read) ve 1 değerini görmüş oluyor bunun sebebi birinci istemci (c0) close yapmış olsa bile AFS tutarlılığından dolayı 1 değerini yazdığında onu sunucuda güncellemiş oldu ve ardından ikinci istemci (c1) onu okuduğunda 1 değerini görmüş oldu. Şeklil 2.0'da sol alt kısımda bulunan (file: a contains: 1) yazısından da anlaşılmaktadır.
Şekil 2.1 için inceleyecek olursak burda da (-s 10) kullanmıştık. Bu kısımda ilk istemciden (c0) önce ikinci istemci (c1) ilk hamleyi yapmaktadır. İkinci istemci 'a' dosyasına en başında varsayılan olarak yüklü olan 0 değerini açıp (open) okumaktadır (read). İkinci istemci dosyayı kapattığında (close) ilk istemci (c0) işlemlerini yapmaya başlayacaktır. İlk istemci de sadece dosyayı açıp (open) ardından okuma (read) işlemi yaptığı için varsayılan olarak yüklenmiş olan 0 değerini okuyacaktır. Burada değişiklik olmamasının sebebi bir üzerine yazma (write) işlemi olmamasıdır. Şekil 2.1'de sol alt kısımda bulunan (file: a contains: 0) yazısından da anlaşılmaktadır.

Şekil 2.0

Şekil 2.1

Soru birde detaylı incelememizi istediği için ona da bakacağız bu da (-d) ile yapılmaktadır. Bu kısımda Şekil 2.2'de görüldüğü üzere (-d 3) alınmaktadır ve Şekil 2.0'da gösterilenlerin detaylandırılmış halidir.



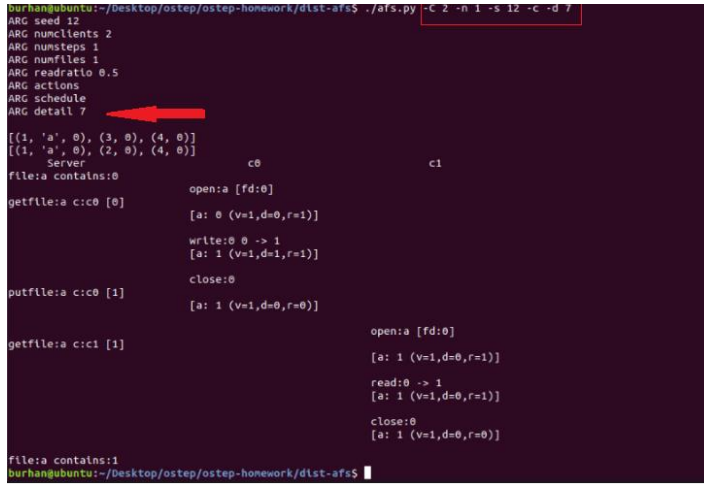
Şekil 2.2

Herbir geri aramanın (callback) tam olarak ne zaman gerçekleştiğini

tahmin edebiliyor musunuz diye sorulmuş bu kısımda detaylandırma kısmını (-d) daha fazla vererek bunu görebiliriz. Birinin gerçekleşmesi için kesin koşul da adım sayısını (-n) değerinin 0 dan büyük olması gerekmektedir yoksa işlem yapılamaz ve geri aramalar (callback) olmayacaktır.

3. Yukarıdakine benzer şekilde, bazı farklı rastgele çekirdeklerle (seeds) çalıştırın ve her adımda tam önbellek durumunu tahmin edip edemeyeceğinizi görün. Önbellek durumu `-c` ve `-d 7` ile çalıştıralarak gözlemlenebilir.

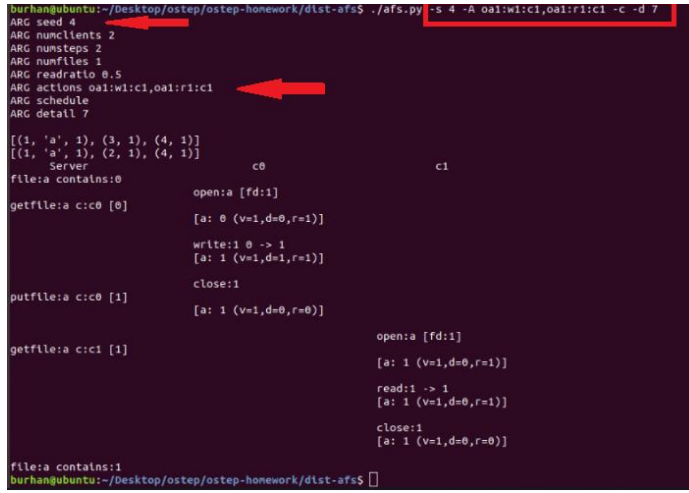
   İlk iki soruda rastgele çekirdeklerle (seeds) çalıştırıp önbellek durumunu tahmin etmiştik. Burada farklı bir durum olarak -c ve -d 7 şeklinde çalıştırılması istenmiş bunu deneyelim.



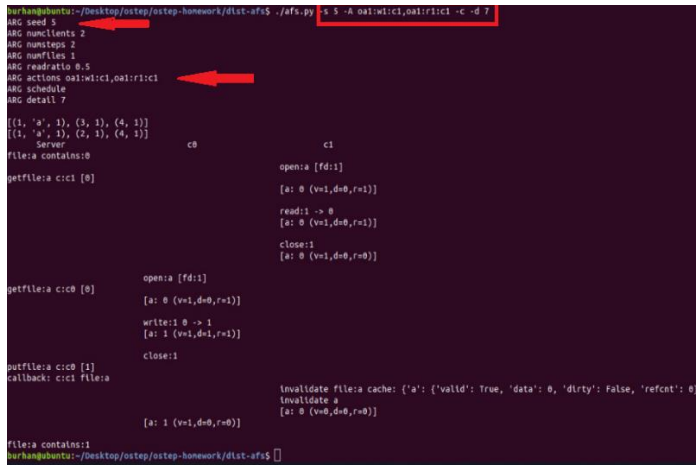
Şekil 3.0

Burada önbellek durumunu, geri aramları (callback) ve geçersiz kılmaları birkaç ekstra bayrakla görmüş olduk (-d 7).
İstemcilerin (c0 ve c1) açma işlemi yaptığında getfile işlemi yapması, putfile yapması gibi detaylı bilgilere erişmiş olmaktadır. Aynı zamanda detaylı bilgilerin içinde [a: 0 (v=1,d=0,r=1) gibi değerler de vardır bunlar şu anlama gelmektedir: Bu 'a' dosyasının 0 değeriyle (verdiğimiz örnek için 0) önbellekte olduğu ve bununla ilişkilendirilmiş üç bit durumu olduğu anlamına gelir [valid(v), dirty (d), reference count (r)]. Valid biti (geçerli bit), içeriğin geçerli olup olmadığını incelemektedir. Şu anda geçerlidir çünkü önbellek bir geri arama (callback) ile geçersiz kılmadı. Kirli bit (Dirty bit) dosyaya yazıldığında değişir ve dosya kapatıldığında sunucuya geri gönderilmelidir. Son olarak referans sayısı (reference count) dosyanın kaç kez açıldığını izlemektedir (Kapanmama (not close) şartı ile).

4. Şimdi bazı belirli iş yükleri oluşturalım. Simülasyonu `-A oa1:w1:c1,oa1:r1:c1` Bayrağı ile çalıştırın. Rastgele zamanlayıcı ile çalışırken `a` dosyasını okuduğunda istemci 1 tarafından gözlemlenen farklı olası değerler nelerdir (farklı sonuçları görmek için farklı rastgele çekirdekleri (seeds) deneyin)? İki istemcinin işlemlerinin tüm olası zamanlama aralarından kaç tanesi istemci 1'in 1 okunmasına ve kaçının 0 değerini okunmasına yol açar?



Şekil 4.0



Şekil 4.1

Burada ekstradan istemci sayısını belirtmek için (-C) veya istemcilerin gerçekleştireceği işlem sayısını belirtmeye (-n) ihtiyaç yoktur.

İstemci eylemlerini belirtmek için aksiyonları (actions) kullanıyoruz. (oa:1:w1:c1,oa:1:r1:c1) dediğimizde iki istemci belirtir; her biri a dosyası için istemci 0 tarafından yazılırken (write) istemci 1 tarafından okunur (read) ve ardından her ikisini de kapatacaktır (close). Şekil 4.0 ve Şekil 4.1 de aynı değerlerin farklı çekirdekleri (seeds) kullanılmıştır. Sorunun cevabına gelecek olursak da istemci 1'in 1 okunması hakkında; istemci 1, istemci 0 kapanmadan önce okumuş (read) veya açmışsa (open) 0 olacaktır aksi takdirde 1 okuyacaktır.

5. Şimdi bazı özel programlar oluşturalım. `-A oa1:w1:c1,oa1:r1:c1` bayrağı ile çalıştırırken, aşağıdaki programlarla da çalıştırın: `-S 01`, `-S 100011`, `-S 011100` ve aklınıza gelebilecek diğerleri. İstemci 1 hangi değeri okuyacaktır?
Bu soruda tek bir -A bayrağı değeri (oa1:w1:c1,oa:r1:c1) ile üç adet -S değerlerini çalıştıracağız bunlar sırasıyla Şekil 5.0, Şekil 5.1 ve Şekil 5.2'de gösterilmiştir.

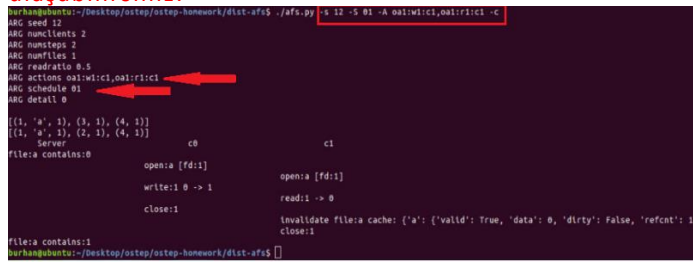
-A oa1:w1:c1,oa1:r1:c1 kısmından bir önceki sorudan bahsetmiştik ama şimdi özet geçecek olursak iki adet istemci kullandığımızı gösteriyor bunları virgülle ayırmış bulunmaktadır. Buradaki kısaltmalar oa1; ilk istemcinin a dosyayını açma (open file 'a'), w1; ilk istemcinin yazma işlemi (write), c1; ilk istemcinin kapatma (close) işlemi diyebiliriz. Virgülden sonraki kısıma ikinci istemci demiştik orda da farklı olarak r1 var o da ikinci istemcinin okuma işlemi (read) yapacağından bahsetmektedir.

-S bayrağı ile istemcilerin hangi sırada çalışacağını kontrol edebiliyoruz mesela -S 111000 dediğimiz zaman istemci 1 için çalışmasını sağlayacak ve ardından tekrar istemci 1 çalışacak en sonunda tekrar istemci 1 çalışacak sıfırlara geldiğimiz zamanda üç defa istemci 0 çalışacaktır.

Soruya bakalım -S 01 için Şekil 5.0'I inceleyecek olursak önce istemci 0 açma (open) işlemi yaparken ardından sıra c1'e geçer ve tekrardan c0 a geçer bu şekilde sırayla devam eder bunun sebebi 01 yazmamızdan kaynaklanıyor. 01 yazdığımız zaman önce istemci 0 için sonra istemci 1 için sonra tekrardan istemci 0 için işlemler yapacaktır ve bu böyle bitene kadar devam eder. Soru bizden istemci 1 için hangi değer okunacağını soruyordu bunun cevabı 0 olacaktır (read:1 -> 0).
Invalidate file: a cache hatası almamızın sebebi de istemci 0 da en son işlem olarak kapatma işlemi yaptığımızdan dolayı istemci 1'e geldiğinde tekrar kapatma yapacağı bir dosyayı önbellekte bulamamaktadır ondan dolayı bu hatayı gösterir bunun sorunun çözümü için bir engel oluşturmamaktadır. Bunun gözükmemesini istiyorsak -A

oa1:w1:c1,oa1:r1:c1 yerine oa1:w1,oa1:r1:c1 yazmamız gerekir bunun sebebi de istemci 0 kapatma işlemi yapmayacağından dolayı istemci 1'e kapatma işlemi yaparken hata almayacaktır.
-S 100011 için Şekil 5.1'i inceleyecek olursak önce istemci 1 için açma (open) işlemi yapacaktır bunun sebebi 1 ile başlamasındandır ardından ise üç adet 0 ile devam ettiği için istemci 0'a gelecek ve üç adet işlem yapacaktır

(open-write-close işlemleri). Sorunun cevabına gelecek olursak da Şekil 5.1den de gözükeceği üzere istemci 1 en son 0 değerini okuyacaktır (read: 1 -> 0).
Invalidate file uyarısını almaya devam ediyoruz yine istemci 0 tarafındaki kapatma (close) işlemini yapmazsak istemci 1 de böyle bir yazı görmeyiz. Bunu görmemizin sebebi zaten dediğimiz gibi istemci 0 da kapatılan dosyayı önbellekte istemci 1 tekrar bakıyor ve bunu okumaya (read) çalışıyor.
-S 011100 için Şekil 5.2'yi inceleyecek olursak önce istemci 0'ı açma (open) işlemi yapacaktır bunun sebebi 0 ile başlamasındandır ardından ise üç adet 1 ile devam ettiği için istemci 1'e gelecek ve üç adet işlem yapacaktır (open-read-close işlemleri). Sorunun cevabına gelecek olursak da Şekil 5.2'den de gözükeceği üzere istemci 1 en son 0 değerini okuyacaktır (read: 1 -> 0).
Invalidate file uyarısı burda da gözükmektedir önceki adımlarda bahsettiğim üzere sebep ve çözümüne ulaşabilirsiniz.



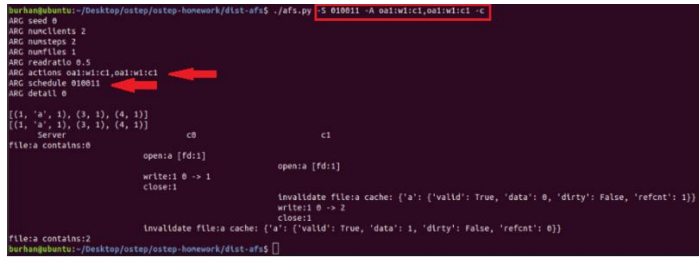
Şekil 5.0

Şekil 5.1

Şekil 5.2

6. Şimdi bu iş yüküyle çalıştırın: `-A oa1:w1:c1,oa1:w1:c1`, ve programları yukarıdaki gibi değiştirin. `-S 011100` değeri ile nasıl bir sonuç elde edilir? Peki ya `-S 010011` ile nasıl bir sonuç elde edersiniz? Dosyanın nihai değerinin belirlenmesinde önemli olan nedir?

Şekil 6.0



Şekil 6.1

Bu soruda verilmiş olan -A oa1:w1:c1,oa1:w1:c1 bayrağı diğer sorulardan farklı olarak iki istemcinin de yazma (write) işlemi yapmasını söylemektedir(w1). Bu bayrak sabitken bir de -S 011100 değerini inceleyecek olursak Şekil 6.0'da detaylı olarak verilmektedir.

Burada ilk başta istemci 0'da açma (open) işlemi yapmaktadır ardından üç defa 1 geldiği için istemci 1' e geçer ve üç adet işlem yapmaktadır (open-write-close). Ardından son iki 0 için tekrar istmeci 0'a geçerek yazma ve kapatma işlemlerini yapmaktadır.

-S 010011 değerini inceleyecek olursak Şekil 6.1'de detaylı olarak verilmektedir.

Burada ilk başta istemci 0'da açma (open) işlemi yapmaktadır ardından 1 geldiği için gidip istmeci 1'de de açma işlemi yapcaktır sonra gelen iki adet 0 değeri istemci 0'ın iki işlem yapmasını sağlayacaktır (write-

close).Ardından gelen son iki 1 değeri ise istemci 1' de yazma ve kapatma işlemi yapacaktır (write-close).

Her iki -S değerinde de Şekil 6.0 ve Şekil 6.1'de görüldüğü üzere dosyanın üzerine yazılan değer 0'dan 2'ye çıkmaktadır (file:a contains:2). Bunun sebebi hem istemci 0'da hem de istemci 1'de yazma (write) işlemi yapmasındandır.

Sorunun cevabına gelecek olursak dosyanın nihai değerinin belirlenmesinde önemli olan ilk yazma (write) istemcisi diğer istmeci kapandıktan (close) sonra sonra kapanırsa (close) son değeri 1 okuyacaktır.