52

Andrew 파일 시스템(AFS)

Andrew 파일 시스템은 1980년대에 Carnegie Mellon University(CMU)의 연구진들에 의해 개발되었다 [How+88]. Carnegie Mellon University의 유명한 M. Satyanarayanan (짧게 "Satya"라 줄여서 부른다) 교수가 이끈 이 프로젝트의 핵심 목적은 **확장성**(**scale**)이다. 가능한 많은 클라이언트를 지원하려면 어떻게 분산 파일 시스템을 설계해야 할까?

파일 시스템의 확장성을 결정하는 데에는 많은 설계와 구현 요소들이 영향을 준다. 그 중 가장 중요한 것은 클라이언트와 서버 간의 **프로토콜**을 설계하는 것이다. 예를 들어, NFS 프로토콜은 클라이언트가 캐시된 내용의 변경 여부를 확인하기 위해 주기적으로 서버를 검사해야 한다. 검사할 때마다 서버의 자원(CPU와 네트워크 대역폭을 포함하는)을 사용하기 때문에 자주 검사하게 되면 서버가 응답할 수 있는 클라이언트의 수를 제한하게 되므로 확장성에 한계가 생긴다.

AFS의 최우선 관심사는 사용자에게 상식적으로 이해가능한 서비스를 제공하는 것이다. NFS에서 캐시 일관성을 설명하기 어려웠던 이유는 그 정책이 클라이언트 측의 캐시 타임아웃 간격 등 하부의 구현 내용과 깊은 연관을 갖기 때문이다. AFS의 캐시 일관성은 간단하고 쉽게 이해할 수 있다. 서버는 클라이언트에게 항상 최신 파일이 open 되는 것을 보장한다.

52.1 AFS 버전 1

우리는 두 버전의 AFS를 설명할 것이다 [How+88; Sat+85]. 첫 번째 버전(AFSv1이라고 부를테지만, 사실 최초의 시스템은 ITC 분산 파일 시스템 [Sat+85]이라고 불렀다)은 기본적인 기능은 다 갖추고 있었지만 확장성이 좋지 않았다. 재설계를 통해 최종적인 프로토콜(AFSv2 또는 AFS라고 부름)이 탄생하였다. 먼저 첫 번째 버전을 설명하도록 하겠다.

모든 버전의 AFS의 기본 원칙 중 하나는 클라이언트가 파일을 접근하면 **로컬 디스크**에 **파일 전체를 캐싱**(**whole-file-caching**)하는 것이다. 파일을 `open()`하면 전체 파일(존재한다면)을 서버에서 가져와서 로컬 디스크에 파일로 저장하는 것이다. 프로그램이 연이어 `read()`와 `write()` 요청을 하면 그 요청은 파일이 저장되어 있는 로컬 파일 시스템에서 서비스 된다. 그러므로 이 연산들은 네트워크 통신이 발생하지 않으므로 매우 빠르다. 마지막으로 `close()`를 호출하면 해당 파일은(변경이 있었으면)

서버로 전송된다. 여기서 NFS와의 명백한 차이점에 유의해야 한다. NFS의 캐싱 단위는 *블럭*이며(전체 파일이 아니기는 하지만 NFS가 파일의 모든 블럭을 캐시할 수도 있다) 클라이언트의 *메모리*(로컬 디스크가 아님)에 해당 블럭을 저장한다.

좀 더 자세하게 살펴보도록 하자. 클라이언트 측 응용 프로그램이 `open()`을 호출하면, AFS 클라이언트(AFS 설계자들은 비너스(Venus)라고 부른다)은 Fetch 프로토콜 메시지를 서버로 전송한다. Fetch 프로토콜 메시지를 통해 파일의 전체 경로명(예, `/home/remzi/notes.txt`)을 파일 서버(**바이스**(**Vice**)라고 부름)로 전송한다. 서버는 경로명을 따라가서 파일을 찾은 후에 클라이언트에게 전체 파일을 전송한다. 클라이언트의 로컬 디스크에 파일을 캐싱한다(로컬 디스크에 쓴다). 위에서 언급했듯이, AFS에서는 연이은 `read()`와 `write()` 시스템 콜은 엄격하게 *로컬*에서만 일어난다(서버와의 통신은 발생하지 않는다). 해당 호출은 로컬의 파일 사본으로 향하게 될 뿐이다. `read()`와 `write()` 호출은 마치 로컬 파일 시스템에 호출하는 것과 같이 동작하기 때문에 접근된 블럭은 클라이언트 메모리에 캐시 될 수도 있다. 그러므로 AFS는 클라이언트의 메모리에 로컬 디스크의 블럭들의 사본을 캐싱한다. 작업이 종료되면 AFS의 클라이언트는 파일의 변경 여부를 검사한다(만약 파일을 쓰기 위해 열었다면). 만약 변경되었으면 새로운 버전을 서버로 Store 프로토콜 메시지와 함께 전송한다. 이때 전체 파일과 경로명이 서버로 전달되어 영속 저장 장치에 기록된다.

AFSv1은 그 뒤에 해당 파일이 접근되면 좀 더 효율적으로 처리할 수 있다. 클라이언트는 먼저 서버에 파일의 변경 여부를 확인한다(TestAuth 프로토콜 메시지를 사용). 변경이 없었다면, 클라이언트의 로컬에 캐시된 버전을 사용하면 되기 때문에 네트워크 전송이 불필요해지고 성능을 개선할 수 있다. 그림 49.1은 AFSv1의 프로토콜 메시지 중 몇 개를 나타낸다. 초기 버전의 프로토콜에서는 파일 내용만 캐싱하였다는 것에 유의하자. 디렉터리 등은 서버에 저장하였다.

`TestAuth`	파일의 변경 여부를 검사 (캐시 항목이 유효한지 확인하는 데 사용됨)
`GetFileStat`	파일의 stat 정보를 얻어옴
`Fetch`	파일의 내용을 로컬 디스크로 가져옴
`Store`	파일을 서버에 저장함
`SetFileStat`	파일의 stat 정보 설정함
`ListDir`	디렉터리의 내용을 보여줌

〈그림 52.1〉 **AFSv1 프로토콜의 하이라이트**

52.2 버전 1의 문제점

첫 번째 버전 AFS의 문제점을 해결할 필요가 있다. AFS의 설계자들은 문제를 정확히 파악하고 자세히 연구하기 위해 상당한 시간을 들여 기존의 프로토타입의 동작 시간을

팁: 측정을 먼저 하고 만들어라(Patterson's Law)

우리의 지도 교수님 중 한 분이셨던 David Patterson (RISC와 RAID로 유명하심)은 어떤 문제를 해결하는 새로운 시스템을 만들기 *전에* 해당 시스템을 측정하고 실제 문제를 증명해 보이는 것을 항상 장려하셨다. 직감을 따르는 대신 실험적 증거들을 활용할 때 시스템 개발의 과정을 좀 더 과학적인 노력으로 변화시킬 수 있다. 이 방법을 사용하였을 때 얻게 되는 부가적인 혜택은 개선된 버전이 개발되기 이전에 정확하게 시스템을 측정하는 방법을 미리부터 갖게 된다는 것이다. 마침내 새로운 시스템을 만들게 되면 결과적으로 두 가지 이득을 얻는다. 하나는 실제적인 문제를 해결했다는 증거를 얻은 것이고, 두 번째는 최신의 기술이 적용된 새로운 시스템을 측정할 수 있는 방법을 이미 갖고 있다는 것이다. 그래서 이것을 **Patterson's Law**라고 부른다.

측정하였다. 이런 류의 실험은 좋은 것이다. **측정**은 시스템이 어떻게 동작하는지와 개선 방향을 이해하는 데 핵심이다. 측정 데이터는 직관력을 갖게 해주며, 시스템 분석 능력을 함양한다. 그들의 연구에 따르면 저자들은 AFSv1에서 두 가지 주요 문제를 발견하였다.

- **경로명을 따라가는 것은 매우 비싼 작업이다**: Fetch 또는 Store 프로토콜에 따라 요청하면 클라이언트는 전체 경로명(예, `/home/remzi/notes.txt`)을 서버에게 전달한다. 해당 파일을 접근하기 위해 서버는 먼저 루트 디렉터리에서 `home`을 찾고 `home`에서 `remzi`를 찾아 나가는 식으로 원하는 파일을 찾을 때까지 경로명 전체를 따라간다. 많은 클라이언트들이 서버를 동시에 접근할 경우, 대부분의 서버의 CPU 시간이 단순히 디렉터리 경로명을 따라가는 것에 사용되고 있다는 것을 알게 되었다.
- **클라이언트가 TestAuth 프로토콜 메시지를 너무 많이 요청한다**: GETATTR 프로토콜 메시지가 너무나 많았던 NFS와 비슷하게 AFSv1도 TestAuth 프로토콜 메시지를 사용하여 로컬의 파일(또는 stat 정보)이 유효한지를 검사하려고 엄청난 트래픽을 생성하였다. 그로 인해 서버들은 클라이언트의 캐시에 저장되어 있는 파일 사본의 사용 여부를 알려주느라 대부분의 시간을 낭비하였다. 대부분의 경우 그 대답은 파일은 변경되지 않는다는 것이었다.

사실 AFSv1에는 두 개의 또 다른 문제가 있었다. 서버들 간에 오버헤드가 적절히 분산되지 않았으며 서버는 클라이언트마다 각기 다른 프로세스를 할당하여 사용하였기 때문에 문맥 교환 비용 등의 다른 오버헤드를 유발하였다. 관리자가 오버헤드의 불균형 문제를 해결할 수 있도록 여기저기로 이동할 수 있는 "**볼륨(Volume)**" 개념을 도입하였다. AFSv2는 프로세스 대신에 쓰레드를 사용하여 문맥 교환 문제를 해결하였다. 하지만 여기서는 지면 관계 상 시스템의 확장성을 제한하는 앞서 언급한 두 개의 주요 프로토콜 문제들만 집중하기로 한다.

52.3 프로토콜의 개선

앞서 언급한 두 문제는 AFS의 확장성을 제한하였다. 서버 CPU가 시스템의 병목이 되었으며 각 서버는 과부하 문제로 인해 20개의 클라이언트만 서비스할 수 있었다. 서버들은 TestAuth 메시지를 너무 많이 받고 있었으며, Fetch 또는 Store 메시지들을 받으면 디렉터리를 구분하여 경로명을 따라가는 데에 너무 많은 시간을 쓰고 있었다. AFS 설계자들은 다음과 같은 문제에 직면하였다.

핵심 질문: 어떻게 확장 가능한 파일 프로토콜을 설계할까
서버와의 상호작용의 수를 최소화하려면 프로토콜을 어떻게 재설계해야 할까? 즉, TestAuth 메시지의 수를 어떻게 하면 줄일 수 있을까? 더 나아가, 서버와의 상호작용을 효율적으로 만들려면 프로토콜을 어떻게 설계해야 할까? 이 두 개의 문제들을 동시에 대처할 수 있다면 AFS를 확장 가능한 버전으로 만들 수 있다.

52.4 AFS 버전 2

AFSv2는 클라이언트와 서버 간의 상호작용의 횟수를 줄이기 위해서 **콜백**(**callback**)이라는 개념을 도입하였다. 간단하게 콜백은 서버가 클라이언트에게 캐싱된 파일의 변경 사실을 알려주겠다고 하는 서버의 약속이다. 클라이언트에 캐쉬된 파일들에 있어 상태(state) 정보가 추가된다. 클라이언트는 캐시된 파일의 유효성 검사를 위해 서버에 접근할 필요가 없다. 서버가 변경 여부를 알려주기 전까지 클라이언트는 파일이 유효하다고 가정한다. **폴링**(**polling**)과 **인터럽트**(**interrupt**)의 비교를 연상하여 생각해 보면 이해가 쉬워진다.

AFSv2는 경로명 대신에 **파일 식별자**(**file identifier**, **FID**, NFS의 **파일 핸들**(**file handle**)과 유사)라는 개념을 사용하여 클라이언트가 원하는 파일의 위치를 표현한다. AFS의 FID는 볼륨 식별자와 파일 식별자 그리고 "uniquifier"(파일이 삭제되었을 때 볼륨과 파일 ID가 재사용이 가능하도록 한다)로 구성되어 있다. 전체 경로명을 서버로 전달하여 서버가 그 경로명을 따라가도록 하는 대신, 클라이언트가 경로명을 한 번에 하나씩 따라가면서 그 결과를 캐싱한다.

클라이언트가 파일 `/home/remzi/notes.txt`를 접근한다고 가정하자. AFS 디렉터리 `home`이 `/`에 마운트되어 있다고 하자(즉, `/`가 로컬 루트 디렉터리이며 home과 그 이하의 디렉터리는 AFS에 있는 것이다). 클라이언트는 먼저 `home` 디렉터리의 내용을 Fetch로 가져와서 로컬 디스크 캐시에 넣은 후에 `home`에 대해서 콜백을 설정한다. 그 후, 디렉터리 `remzi`를 Fetch로 가져와서 로컬 디스크 캐시에 넣은 후 서버의 `remzi`에 대해서도 콜백을 설정한다. 마지막으로 `notes.txt`를 Fetch로 가져온 후에 로컬 디스크에 이 일반 파일을 캐싱하고 콜백 설정 후, 최종적으로 호출한 응용 프로그램에게 파일 디스크립터를 리턴한다. 정리한 내용은 그림 52.2에 나와 있다.

클라이언트(C_1)	서버
fd=open(“/home/remzi/notes.txt”,...);	
Fetch(home FID, “remzi”) 전송	
	Fetch 요청 수신
	home 디렉터리에서 remzi를 검색
	remzi에 콜백(C_1)을 설정
	remzi의 내용과 FID를 전송
Fetch 응답 수신	
remzi를 로컬 디스크 캐시에 쓰기	
remzi의 콜백 상태를 기록	
Fetch(remzi FID, “notes.txt”) 전송	
	Fetch 요청 수신
	remzi 디렉터리에서 notes.txt를 검색
	notes.txt에 콜백(C_1)을 설정
	notes.txt의 내용과 FID를 전송
Fetch 응답 수신	
notes.txt를 로컬 디스크 캐시에 쓰기	
notes.txt의 콜백 상태를 기록	
캐시된 notes.txt를 로컬 `open()`으로 열기	
응용 프로그램에게 파일 디스크립터 리턴	
read(fd, buffer, MAX);	
캐시된 사본에 대해 로컬 `read()` 수행	
close(fd);	
캐시된 사본에 로컬 `close()` 수행	
파일 변경 시, 서버로 내려보냄	
fd=open(“/home/remzi/notes.txt”,...);	
Foreach dir(home, remzi)	
if(callback(dir)==VALID)	
lookup(dir)에 로컬 사본 사용	
else	
Fetch(위와 같음)	
if(callback(notes.txt)==VALID)	
로컬의 캐시된 사본을 열기	
파일 디스크립터를 리턴	
else	
Fetch(위와 같음) 그리고 열고 fd 리턴	

〈그림 52.2〉 **파일 읽기 : 클라이언트와 파일 서버 측의 동작**

여담: 캐시 일관성이 만병통치약은 아니다

분산 파일 시스템을 설명할 때, 파일 시스템의 캐시 일관성 기능에 많은 부분을 할애한다. 하지만, 기본적인 일관성으로는 다수의 클라이언트들이 파일에 접근할 때 발생하는 모든 문제를 해결할 수는 없다. 예를 들어, 여러 클라이언트가 코드를 체크인하고 체크아웃하는 코드 저장소를 만든다고 해 보자. 이때에는 파일 시스템에 전적으로 의지하면 안 된다. 동시 접속이 가능한 상황에서는 **파일 수준의 락(file-level locking)**을 사용하여 반드시 "제대로 된" 일만 발생토록 해야 한다. 실제로, 어느 응용 프로그램이든 동시 갱신에 주의를 기울인다면, 충돌을 피하기 위한 기법을 추가할 것이다. 이번 장과 이전 장에 설명된 기본적인 일관성은 일반적인 상황에서 유용하다. 즉, 사용자가 다른 클라이언트에서 로그인 할 때 자신의 파일이 제대로 된 버전이길 기대한다. 이러한 프로토콜에서 무엇인가를 더 바란다는 것은 실패와 실망 그리고 좌절감만 있을 뿐이다.

여기서 NFS와의 핵심적인 차이점은 디렉터리 또는 파일을 가져오는 각 과정에서 AFS의 클라이언트는 서버에 콜백을 설정한다는 것이다. 서버는 클라이언트가 캐시하고 있는 데이터의 상태가 변경되면 클라이언트에게 반드시 알려준다. 이로 인한 장점은 분명하다. `/home/remzi/notes.txt`를 *처음*으로 접근할 때는 많은 클라이언트-서버 메시지(위에서 설명한 것과 같이)가 발생하겠지만, 모든 디렉터리와 파일 `notes.txt`에 대해서 콜백을 설정해 놓았기 때문에 이후에는 모든 접근이 로컬에서만 발생하며 서버와의 상호작용은 전혀 필요없다. 클라이언트에 파일이 캐시되어 있는 경우 AFS는 로컬 디스크 기반의 파일 시스템과 거의 동일하게 동작한다. 파일을 한 번 이상 접근한다면 두 번째 접근은 로컬에서 접근하는 것처럼 빠르게 접근할 수 있다.

52.5 캐시 일관성

NFS의 캐시 일관성 두 가지는 **갱신 가시성(update visibility)**과 **오래된 캐시(cache staleness)** 문제였다. 갱신 가시성에서는 새로운 파일 버전이 서버로 갱신되었는지가 문제였다. 오래된 캐시는 다음과 같은 질문을 하였다. 서버가 새로운 버전을 갖게 되었을 때 클라이언트의 캐시 사본이 새 버전으로 갱신될 때까지 얼마나 걸릴까?

AFS가 제공하는 캐시 일관성은 콜백과 전체 파일 캐싱을 사용하기 때문에 쉽게 설명되며 이해하기도 쉽다. 다만, 고려해야 할 두 가지 경우가 있는데, *다른* 기계들 간의 일관성 그리고 *같은* 기계 내에 있는 프로세스들 간의 일관성이다.

AFS는 서버에서 내용이 변경되는 시점과 다른 기계에 캐시된 사본이 무효화가 되는 시점이 같다. 그 시점은 갱신된 파일이 닫히는 순간이다. 클라이언트는 파일을 연 후에 쓴다(어쩌면 반복적으로). 클라이언트가 파일을 닫으면, 새로운 파일은 서버로 보내진다. 서버는 캐시된 사본을 갖고 있는 클라이언트들과 콜백을 모두 끊어서 클라이언트들이 더 이상 오래된 파일 사본을 캐시에 갖고 있지 않도록 한다. 콜백이 끊긴 클라이언트들이 그 파일을 다시 열려면 파일을 서버로부터 다시 최신 버전을 갖고 오게 된다.

AFS는 같은 기계 내의 프로세스들 사이에서는 위의 일관성을 유지기법을 적용하

지 않는다. 파일에 대한 쓰기 결과는 즉시 다른 로컬 프로세스들에게 보여진다(즉, 프로세스가 파일을 닫지 않아도 갱신된 최신의 내용이 보인다). 단일 기계에서의 동작 방식은 상식적으로 기대하는 바와 같다. UNIX의 일반적인 시맨틱에 기반하여 동작하기 때문이다. 다른 기계들 간에 일관성 유지 문제의 경우에만 AFS의 좀 더 강력한 기능을 보게 된다.

한 가지 흥미로운 기기 간의 일관성 유지 사례를 살펴보자. 아주 드문 경우이기는 하지만 서로 다른 기기의 프로세스들이 동시에 한 파일을 수정한다고 하면, AFS는 **마지막 기록자가 승리(last writer wins)**라는 방식을 사용한다(또는 **마지막으로 닫는 자가 승리(last closer wins)**라고 불려야 할지도 모르겠다). `close()`를 마지막으로 부르는 클라이언트가 서버에 마지막으로 파일의 전체를 갱신하는 것이고 해당 파일이 "승자" 파일이 된다. 즉, 다른 이들이 보게 될 파일이다. 결과는 사용자가 갱신한 전체 파일이다. 이때 블럭 기반의 프로토콜을 갖는 NFS와 다른 점에 유의해야 한다. 클라이언트가 파일을 갱신할 때 NFS는 개별 블럭에 쓰고 서버로 전송된다. 서버의 최종 파일은 양쪽 클라이언트의 갱신 내용이 혼합되어 있다. 혼합된 파일은 대부분 앞뒤가 맞지 않는다. 두 클라이언트가 JPEG 그림을 부분적으로 수정한 경우를 생각해 보자. 쓰기 결과가 섞여 있다면 그 JPEG 파일은 깨진 것과 다름없다.

그림 52.3에서 몇 가지 다른 시나리오들을 볼 수 있다. 각 열은 클라이언트 1 상의 두 프로세스들(P1과 P2)의 동작과 캐시 상태, 클라이언트 2 상의 프로세스(P3)와 캐시 상태 그리고 서버를 나타낸다. 이 모두는 하나의 가상의 파일인 `F`에 연산을 하고 있다. 그림에서 서버 열은 단순하게 좌측 열들의 연산이 완료된 시점의 파일 내용을 나타낸다. 읽어보고 각 읽기가 왜 그런 결과를 리턴하는지를 이해해 보라. 우측 컬럼의 설명이 도움이 될 것이다.

52.6 크래시 복구

이제까지의 설명으로 판단컨데, 크래시 복구 문제가 NFS 때보다 더 복잡할 것이라 느꼈을 것이다. 그렇다. 예를 들어보자. 클라이언트(C1)가 재부팅을 하느라 잠시 서버(S)와의 교신이 두절되었다. 이 기간 중에 서버 S가 C1에게 콜백 메시지를 보내려 했을 수 있다. 예를 들어, C1이 파일 F를 로컬 디스크 캐시에 가지고 있었고, C2(또 다른 클라이언트)가 F를 갱신하였다. S는 파일 F를 캐싱한 모든 클라이언트에게 로컬 캐시에서 파일을 제거하라는 메시지를 보냈다. C1은 재부팅하느라 이 메시지를 못 받았다. 문제가 발생한다. 해결책이 필요하다. 이를 위해서 재부팅되면 C1은 모든 캐시 내용을 검사한다. 다음 번에 파일 F를 접근할 때 C1은 먼저 서버에게 자신이 캐시한 파일 F의 사본의 유효 여부를 확인해야 한다. (TestAuth 프로토콜 메시지를 사용). 유효하면 C1은 파일을 그대로 사용하고, 그렇지 않다면 C1은 서버에서 새로운 버전을 가져온다.

서버가 크래시한 경우 복구는 좀 더 복잡하다. 문제는 컬백 관련 정보를 메모리에 보관한다는 사실이다. 서버가 크래시 된후, 재부팅되면, 어떤 클라이언트가 어떤 파일을 갖고 있었는지 알 길이 없다. 서버가 재시작하면 각 클라이언트는 서버가 크래시되었다는 사실을 인지하고, 자신의 캐시 내용의 유효성을 다시 검사해야 한다. 즉, (위에서 한

클라이언트$_1$			클라이언트$_2$		서버	해설
P_1	P_2	캐시	P_3	캐시	디스크	
`open(F)`		-		-	-	파일 생성
`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	로컬 프로세스들은
	`read()->B`	B		-	A	즉시 기록된 결과를 봄
	`close()`	B		-	A	
		B	`open(F)`	A	A	원격 프로세스는
		B	`read()->A`	A	A	기록된 내용 못 봄
		B	`close()`	A	A	
`close()`		B		~~A~~	B	... `close()`가
		B	`open(F)`	B	B	완료될 때까지
		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	P3에겐 불행하게도
		D	`read()->D`	D	D	최종 기록자가 승리
		D	`close()`	D	D	

〈그림 52.3〉 **캐시 일관성 시간 흐름표**

것처럼) 사용하기 전에 파일의 유효성을 재확인해야 한다. AFS에서 서버 크래시는 대형사고다. 각 클라이언트가 서버가 크래시되었다는 것을 제 때 파악할 수 있게 설계해야 한다. 그렇지 않으면 오래된 파일을 접근할 수도 있다. 이에 대한 여러 가지 복구 방법들이 있다. 예를 들어서, 서버가 살아나서 재실행되면 각 클라이언에게 메시지("갖고 있는 캐시 내용을 믿지 마라!"라고 알림)를 전송한다거나 주기적으로 서버가 살아있는지를 검사(**heartbeat** 메시지라고 부른다)하는 방법이 있다.

52.7 AFSv2의 확장성과 성능

새로운 프로토콜을 추가함으로써, AFSv2이 성능이 확장성 측면에서 원래의 버전보다 훨씬 더 우수한 것으로 판명되었다. 실제로, 하나의 서버가 약 50개의 클라이언트(단 20개가 아니라)를 지원할 수 있다. 더욱 바람직한 성질은 클라이언트 성능이 대부분의 경우 로컬 성능에 거의 가깝게 나온다는 것이다. 파일들을 로컬에서 접근하기 때문이다. 특히 대부분의 읽기 요청이 로컬 디스크 캐시(실질적으로 많은 경우 로컬 메모리)에서 처리된다. 새로운 파일을 생성하거나 기존의 것에 쓰는 경우에만 서버에 Store 메시지를 전달할 필요가 있고, 서버의 해당 파일이 새로운 버전으로 갱신된다.

AFS 성능을 NFS와 비교하여 보자. 그림 52.4가 비교 결과를 나타낸다.

워크로드	**NFS**	**AFS**	**AFS/NFS**
1. 작은 파일, 순차 읽기	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. 작은 파일, 순차 다시읽기	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. 중간 파일, 순차 읽기	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. 중간 파일, 순차 다시읽기	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. 큰 파일, 순차 읽기	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. 큰 파일, 순재 다시읽기	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. 큰 파일, 한 번 읽기	L_{net}	$N_L \cdot L_{net}$	N_L
8. 작은 파일, 순차 쓰기	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. 큰 파일, 순차 쓰기	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. 큰 파일, 순차 덮어쓰기	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. 큰 파일, 한 번 쓰기	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

〈그림 52.4〉 **AFS 대 NFS**

다양한 크기의 파일들에 대해 대표적인 읽기와 쓰기 패턴에서의 성능을 분석해 보았다. 작은 파일들은 N_s 개의 블럭들로 이루어져 있으며 중간 크기 파일들은 N_m 개의 블럭들로 그리고 큰 파일들은 N_L 블럭들로 이루어져 있다. 작은 파일들과 중간

크기의 파일들은 클라이언트의 메모리에 저장이 가능하다고 가정하고, 큰 파일들은 로컬 디스크에는 저장 가능하지만 클라이언트 메모리에는 저장이 안 된다고 하자.

분석을 위해 네트워크를 통해 원격 서버의 파일 블럭을 접근하는 것은 L_{net} 단위 시간만큼 소요된다고 하자. 로컬 메모리를 접근하는 것은 L_{mem} 의 시간이 소요되고 로컬 디스크를 접근하는 것은 L_{disk} 의 시간이 소요된다. 일반적으로 $L_{net} > L_{disk} > L_{mem}$ 의 식이 성립된다.

마지막으로, 처음으로 파일을 접근하는 경우에는 캐시에서 가져올 수 있는 것은 없다. 캐시에 파일 전체를 저장할 만큼 충분한 공간이 있을 때에만 반복되는 파일 접근(즉, "다시읽기")들이 캐시에서 히트된다.

그림의 각 줄은 특정 작업(예, 작은 파일을 순차 읽기)을 처리할 때 NFS 또는 AFS에서 걸리는 대략적인 시간을 나타낸다. 가장 우측은 AFS와 NFS의 비율을 나타낸다.

다음과 같은 특성을 관찰할 수 있다. 첫 번째, 대부분의 경우 각 시스템의 성능은 대체적으로 비슷하다. 예를 들어, 파일을 처음 읽는 경우(예, 워크로드 1, 3, 5), 원격 서버에서 파일을 가져오는 시간이 대부분을 차지한다. 두 시스템 모두 비슷한 시간이 소요된다. 이 경우에 파일을 로컬 디스크에 써야 하기 때문에 AFS가 더 느리지 않을까 생각할 수도 있다. 해당 쓰기는 로컬(클라이언트 측) 파일 시스템의 캐시에 버퍼가 되기 때문에 그 비용은 감춰지는 경향이 있다. 마찬가지로, AFS는 캐시된 사본을 디스크에 저장하기 때문에 로컬에 캐시된 사본을 읽는 것이 느릴 것이다라고 생각할 수도 있다. 하지만, AFS는 로컬 파일 시스템 캐시의 혜택을 받는다. AFS에서 읽을 때 클라이언트 측 메모리 캐시에서 히트가 발생할 가능성이 많기 때문에 NFS의 성능과 비슷하게 될 것이다.

둘째로, 흥미로운 차이점은 큰 파일을 순차적으로 다시읽기를 시도할 때 나타난다(워크로드 6). AFS는 큰 로컬 디스크 캐시를 갖고 있기 때문에 파일을 다시 읽을 때에는 디스크에서 파일을 읽을 것이다. 대비적으로 NFS는 클라이언트 메모리에 블럭들만 캐시할 수 있다. 그 결과 큰 파일(즉, 로컬 메모리보다 큰 파일)이 다시읽기 되었다면 NFS 클라이언트는 원격 서버로부터 전체 파일을 다시 가져와야 한다. 그러므로 실제로 원격 접속이 로컬 디스크보다 느리다고 한다면, AFS는 NFS 보다 $\frac{L_{disk}}{L_{net}}$ 배만큼 더 빠르다. 이 경우에 NFS 서버의 오버헤드도 증가하기 때문에, 다수의 클라이언트를 서비스하는 데(확장성) 영향을 준다.

셋째로, 순차 쓰기 동작은 (새로운 파일들에 대한) 양 시스템에서 비슷한 성능을 보인다는 것(워크로드 8, 9). AFS는 파일을 로컬의 캐시된 사본에 쓸 것이다. 파일이 닫히면 AFS 클라이언트는 프로토콜에 정의된 바와 같이 해당 파일을 서버로 전송한다. NFS는 쓰기 결과를 클라이언트의 메모리 버퍼에 보관한다. 클라이언트 측 메모리가 고갈될 경우 블럭들이 강제로 서버에 보내질 수도 있겠지만, NFS는 닫을-때-내보냄 일관성을 유지하기 위해서 분명하게 파일이 닫힐 때에만 서버에 쓰도록 하고 있다. 이 부분에서 AFS는 모든 데이터를 디스크에 쓰기 때문에 AFS가 더 느리다고 생각할 수도 있다. 하지만, 로컬 파일 시스템에 쓴다는 것을 기억해야 한다. 쓰기는 먼저 페이지 캐시에 커밋이 되고 얼마 후에(백그라운드로) 디스크로 내려간다. AFS는 클라이언트 측 운영체제의 메모리 캐싱 부분의 도움으로 성능을 향상시킨다.

넷째로, AFS는 파일 순차 덮어 쓰기에서 성능이 아주 안좋은 것을 관찰하였다 (워크로드 10). 지금까지는 새로운 파일을 생성하는 워크로드만을 가정하였다. 하지만 이 경우에는 이미 존재하는 파일을 갱신한다. AFS에서는, 특히 덮어 쓰기 경우에 안 좋다. 그 이유는 클라이언트가 파일 전체를 먼저 가져온 뒤에, 덮어 쓰기를 수행하기 때문이다. 그에 반하여 NFS의 클라이언트는 블럭들을 무조건 덮어 쓰기 때문에, 파일을 클라이언트로 읽어[1]올 필요가 없다.

마지막으로 큰 파일의 일부 작은 데이터를 접근하는 워크로드에서는 NFS가 AFS 보다 훨씬 더 좋은 성능을 보인다(워크로드 7, 11). AFS 프로토콜은 파일이 열리면 파일 전체를 가져오지만, 불행하게도 작은 크기의 읽기 또는 쓰기만 수행되는 경우이다. 만약 파일이 갱신되었다면 전체 파일을 서버로 다시 보내야 하기 때문에 오버헤드가 두 배가 된다. 블럭 기반의 프로토콜을 따르는 NFS는 읽기와 쓰기 크기에 비례하여 I/O를 수행한다.

NFS와 AFS는 서로 다른 목적으로 설계되었다. 그 결과 서로 다른 성능 특성을 보인다는 사실을 알 수 있다. 이 차이가 의미가 있느냐는, 언제나 그렇듯이, 워크로드에 달려 있다.

52.8 AFS: 그 외의 개선점들

Berkeley FFS의 소개에서 보았던 것처럼(심볼릭 링크와 다른 몇 가지 기능을 더 갖고 있었다), AFS의 설계자들은 시스템 사용과 관리가 용이하도록 몇 가지 기능을 추가하였다. 예를 들어, AFS는 진정한 의미의 전역적 이름 공간(global name space)을 클라이언트에게 제공한다. 모든 파일들이 모든 클라이언트 기계들에서 반드시 같은 이름을 갖도록 하였다. NFS는 각 클라이언트가 자신이 원하는 대로 NFS 서버들을 마운트할 수 있다. 클라이언트들 간에서 파일이 같은 이름을 갖기 위해서는 동일한 마운트 관례를 따르도록 해야 한다(그리고 관리자의 엄청난 노력도 필요하다).

AFS는 보안을 강조하였다. 사용자 인증 방법들을 포함하며 원하면 사용자의 파일들을 사용자 자신만 볼 수 있도록 하였다. NFS는 그와 달리 수년 동안 기초 수준의 보안만을 제공하였다.

AFS는 사용자가 유연하게 접근 제어를 관리할 수 있는 기능들을 갖고 있다. AFS에서는 어떤 사용자가 무슨 파일을 접근할 수 있는지를 사용자가 자유롭게 제어할 수 있다. NFS는 대부분의 Unix 파일 시스템들과 같이 이러한 종류의 공유에 대한 지원이 매우 약하다

마지막으로 앞에서 언급했던 것처럼, AFS는 시스템 관리자들을 위해서 서버들을 간단하게 관리할 수 있는 도구들을 추가하였다. 시스템 관리라는 측면에서 생각한다면 AFS는 이 분야에서 수 광년을 앞서 있었다.

1) NFS는 블럭단위로 읽으며, 읽기의 크기는 블럭 크기에 정렬이 되어 있다고 가정한다. 그렇지 않다면 NFS 클라이언트도 마찬가지로 블럭을 먼저 읽어야 할 것이다. 또 하나, 파일 오픈시 O_TRUNC 플래그는 사용하지 않는다고 가정한다. 만약 이 플래그를 사용한다면, AFS에서는 파일 오픈을 위해 곧 잘려나갈 파일의 내용을 가져와야 한다.

여담: 워크로드의 중요성

어떤 시스템이든지 평가할 때의 중요한 사안은 **워크로드**의 선택이다. 컴퓨터 시스템들은 매우 다양하게 사용되기 때문에 선택할 수 있는 워크로드가 너무나 많다. 설계가 합당한지를 판단하기 위해서 저장장치 시스템 설계자들은 무슨 워크로드가 중요한지를 어떻게 정해야 할까?

AFS의 설계자들은, 자신들이 관찰한 파일 시스템 사용 패턴을 기반으로, 워크로드 특성을 결정하였다. 구체적으로, 대부분의 파일들은 공유 빈도가 낮으며 접근이 될 때는 그 전체를 순차적으로 접근한다고 가정하였다. 이와 같은 가정을 고려하면 AFS의 설계가 완벽하게 이해된다.

이러한 가정이 항상 옳지는 않다. 예를 들면, 응용 프로그램이 정보를 주기적으로 로그에 추가하는 경우를 생각해 보자. 이러한 작은 로그 쓰기들은 기존의 큰 파일에 작은 양의 데이터를 추가하는데 AFS에 있어서는 이것은 상당한 골치였다. 그 외에도 다른 많은 어려운 워크로드가 있다. 일례로, 데이터베이스 트랜젝션이 임의의 위치에 갱신하는 그런 워크로드이다.

어떤 종류의 워크로드가 일반적인지에 대한 정보를 원한다면 지금까지 진행된 많은 연구들을 읽어 봐야 할 것이다. 워크로드를 아주 잘 분석한 보고를 원한다면 AFS에 대한 회고 [How+88]와 다음 논문들을 [Bak+91; Har+11; RLA00; Vog99] 살펴보기 바란다.

52.9 요약

AFS는 NFS와는 상당히 다른 방식으로 분산 파일 시스템을 개발하였다. AFS에서 특히 중요한 부분은 프로토콜의 설계이다. 서버와의 통신을 최소화하여 서버당 클라이언트를 증가시켰고 이를 통해 특정 사이트를 관리하는 데 필요한 서버의 수를 감소시켰다. AFS는 단일 이름 공간과 보안 그리고 접근 권한 목록(access control list) 등의 많은 유용한 기능들을 갖고 있다. AFS의 일관성 모델은 이해하기 쉽고 NFS에서는 가끔 나타났던 예외적인 상황이 발생하지 않는다.

그렇지만 불행하게도 AFS는 점차 사라지고 있다. NFS가 오픈 표준이 되어서 수많은 제조사가 지원하고 있으며, CIFS(Windows 기반의 분산 파일 시스템 프로토콜)와 더불어 시장을 주도하고 있기 때문이다. AFS를 가끔 볼 수 있지만(Wisconsin 대학을 포함하는 다양한 교육 기관에서), 여전히 영향이 남아 있는 것은 실제 시스템 자체가 아니라 AFS의 개념들 때문이다. 이제는 NFSv4에 서버 상태(예, "open" 프로토콜 메시지) 개념이 추가되어 AFS 프로토콜과 점점 더 유사해지고 있다.

참고 문헌

[Bak+91] **"Measurements of a Distributed File System"**
Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout
SOSP '91, Pacific Grove, California, October 1991
사람들이 분산 파일 시스템을 사용하는 방식을 조사한 초기 논문이다. AFS의 초기의 직관들이 많은 부분 맞음을 보여준다.

[Har+11] **"A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications"**
Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
SOSP '11, New York, New York, October 2011
Apple 데스크탑 워크로드를 분석한 우리의 논문이다. 시스템 연구 그룹들이 일반적으로 집중하는 서버 기반의 워크로드들과는 상당히 다른 것을 보였다. 최근의 참고 문헌을 포함하는 많은 수의 관련 연구를 소개하고 있다.

[How+88] **"Scale and Performance in a Distributed File System"**
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J.West
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
유명한 AFS 시스템에 대한 긴 학술지 버전으로 세계 도처 여러 곳에서 아직도 사용 중이다. 그리고 아마도 분산 파일 시스템의 개발 방법에 대해 명확하게 사고한 가장 초기의 논문이다. 공학 원칙과 과학적 측정이 아주 적절히 조합된 경우의 예를 보여준다.

[RLA00] **"A Comparison of File System Workloads"**
Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson
USENIX '00, San Diego, California, June 2000
Baker의 논문[Bak+91]에 비해서 좀 더 최신의 트레이스 자료로서 몇 가지 흥미로운 그리고 예상치 못한 내용이 있다.

[Sat+85] **"The ITC Distributed File System: Principles and Design"**
M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, and M.J. West
SOSP '85, Orcas Island, Washington, December 1985
조금 오래된 분산 파일 시스템에 대한 논문이다. 대부분의 AFS의 기본 설계가 이 오래된 시스템에 구현되어 있었지만 확장성을 위한 개선은 되어있지 않다.

[Vog99] **"File system usage in Windows NT 4.0"**
Werner Vogels
SOSP '99, Kiawah Island Resort, South Carolina, December 1999
Windows 워크로드의 훌륭한 연구로서 과거에 진행되었던 많은 Unix 기반의 연구들과는 근본적으로 다르다.

숙제

이 절에서는 afs.py라고 하는 간단한 AFS 시뮬레이터를 사용하여 Andrew 파일 시스템의 동작을 이해해 보자. 사용 방법은 README 파일을 참고하자.

문제

1. 클라이언트가 어떤 값들을 읽게 될지 예측할 수 있는지 확인하기 위하여 간단한 경우들로 실험해 보자. 랜덤 시드 플래그(`-s`)를 변경하여 흐름을 따라갈 수 있는지 확인해 보고 또한 파일에 저장될 중간 값과 최종 값을 예측할 수 있는지 보자. 파일의 개수(`-f`)를 조정해 보고 클라이언트의 수(`-C`)와 읽기 비율(`-r`, 0부터 1 사이의 값)을 변경하여 좀 더 도전적인 상황을 만들어 보자. 더 흥미로운 상호작용을 보기 원한다면 좀 더 긴 트레이스를 생성해 보는 것도 좋은 방법이다. 예로 `-n 2` 또는 그 이상의 값을 사용해 보자.
2. 이번에는 위와 같은 동작에서 AFS 서버가 몇 번의 콜백할 때 마다 예측할 수 있는지 보자. 랜덤 시드를 변경하여 실험해 보자. 그리고 해답을 실행해 볼 때는(`-c`를 사용) 좀 더 상세한 피드백을 제공해도록 상세 피드백 옵션을 설정(예, `-d 3`)하여 콜백이 언제 발생하는지 확인해 보자. 정확히 언제 각 콜백이 발생하는지 알 수 있겠는가? 콜백이 발생하는 정확한 조건은 무엇인가?
3. 위와 비슷하게 또 다른 랜덤 시드를 사용하여 각 단계 마다 정확한 캐시 상태를 예측할 수 있는지 실험해 보자. 캐시 상태는 `-c`와 `-d 7` 옵션을 주어서 확인할 수 있다.
4. 이번에는 구체적인 워크로드를 만들어 보자. 다음과 같이 `-A oa1:w1:c1,oa1:r1:c1` 플래그를 사용하여 시뮬레이션을 실행해 보자. 파일 `a`에서 읽을 수 있는 값들은 무엇인가?(다른 결과를 만들어 내기 위하여 다른 랜덤 시드를 실험해 보라) 두 클라이언트 연산의 가능한 모든 스케줄을 고려할 때, 클라이언트 1이 1을 읽는 경우는 몇 번이고, 0을 읽는 경우는 몇 번인가?
5. 이번에는 다음과 같은 구체적인 스케줄에 따라 동작시켜 보자. 프로그램을 `-A oa1:w1:c1,oa1:r1:c1` 플래그를 주고 실행할 때 `-S 01`, `-S 10011`, `-S 011100` 및 생각할 수 있는 다른 스케줄링 옵션을 추가해 보자. 이때, 클라이언트 1은 어떤 값을 읽는가?
6. 이번에는 `-A oa1:w1:c1,oa1:w1:c1`의 옵션으로 워크로드를 설정한 후에 위에서 사용한 스케줄 옵션을 적용하여 실행해 보자. `-S 011100`의 옵션을 사용하면 어떻게 동작하는가? `-S 010011`의 옵션을 준 경우는 어떤가? 파일의 최종 값을 결정하는 데 있어서 중요한 것은 무엇인가?