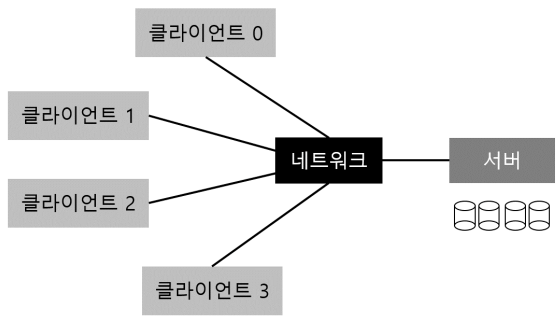
51

Sun 사의 네트워크 파일 시스템(NFS)

분산 클라이언트 서버 컴퓨팅이 처음 사용되기 시작한 영역 중 하나가 분산 파일 시스템이라는 영역이다. 이 환경에서는 다수의 클라이언트 기계와 하나(또는 적은 수)의 서버가 있으며, 서버는 데이터를 디스크에 저장하고 클라이언트는 약속된 포맷에 따라 메시지를 보내서 데이터를 요청한다. 그림 51.1은 그 기본적인 환경을 나타낸다.



〈그림 51.1〉 **일반적인 클라이언트/서버 시스템**

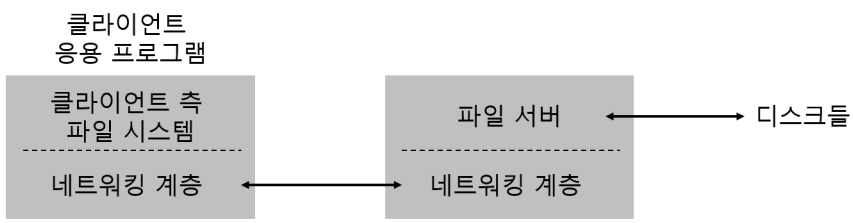
그림에서 볼 수 있는 것과 같이 서버는 디스크를 갖고 있고 클라이언트들은 해당 디스크에 있는 자신의 디렉터리와 파일에 접근하기 위해 메시지를 전송한다. 이러한 구조를 사용하는 이유가 뭘까? (즉, 클라이언트가 자기 로컬 디스크를 사용하지 않을까?) 그 이유는 클라이언트 간에 데이터를 **공유**하는 것이 쉬워지기 때문이다. 만약 한 기계(클라이언트 0)에 저장된 파일을 접근한 후에 다른 위치에 있는 (클라이언트 2) 파일을 사용할 때, 같은 파일 시스템에 있는 것처럼 접근할 수 있다. 데이터는 연결된 기계 사이에서 자연스럽게 공유된다. 부차적인 장점은 중앙 집중형 관리이다. 예를 들어, 파일을 백업할 때에 많은 클라이언트들에서 하는 것이 아니라 몇 안되는 서버에서만 수행하면 된다. 또, 다른 장점은 **보안**이다. 잠겨 있는 방에 서버들을 모아두면 특정 유형의 문제들이 발생하는 것을 막을 수 있다.

핵심 질문: 분산 파일 시스템은 어떻게 만드는가
분산 파일 시스템을 어떻게 만드는가? 고려해야 할 핵심적인 것들은 무엇이 있는가? 어떤 것들이 잘못되기 쉬운가? 기존의 시스템에서 무엇을 배울 수가 있는가?

51.1 기본적인 분산 파일 시스템

이제 단순화된 분산 파일 시스템의 구조에 대해서 살펴보도록 하자. 이 간단한 클라이언트/서버로 구성된 분산 파일 시스템은 지금까지 우리가 살펴보았던 파일 시스템들보다 구성 요소가 더 많다. 클라이언트 쪽에서는 클라이언트 응용 프로그램이 클라이언트 측 파일 시스템을 통해서 파일과 디렉터리를 접근한다. 클라이언트 프로그램이 서버에 있는 파일을 접근하려면 **클라이언트 측 파일 시스템**에 **시스템 콜**(`open()`, `read()`, `write()`, `close()`, `mkdir()` 등)을 호출한다. 분산 파일 시스템은 성능 부분을 제외한다면 응용 프로그램 입장에서는 로컬(디스크 기반) 파일 시스템과 전혀 다르지 않다. 분산 파일 시스템은 파일들을 로컬 파일 시스템과 동일(**transparent**)하게 접근할 수 있도록 하는 것이 목표이다. 누가 전혀 새로운 API나 사용이 불편한 파일 시스템을 쓰려고 하겠는가?

클라이언트 측 파일 시스템의 역할은 시스템 콜을 서비스하기 위해 필요한 동작들을 실행하는 것이다. 예를 들어, 만약 클라이언트가 `read()` 요청을 내렸다면 클라이언트 파일 시스템은 **서버 측 파일 시스템**(또는, 좀 더 흔하게는 **파일 서버**)에 메시지를 전송해서 해당 블럭을 읽도록 할 것이다. 그러면 파일 서버는 디스크(또는 자체가 갖고 있는 메모리 캐시)에서 블럭을 읽은 후에 클라이언트가 요청한 데이터와 함께 메시지를 전송할 것이다. 클라이언트 파일 시스템의 `read()` 시스템 콜은 사용자 버퍼에 데이터를 복사하는 것으로 요청을 완료한다. 클라이언트가 동일한 블럭을 `read()` 요청한다면 클라이언트측의 메모리 또는 디스크에 **캐시된** 이전 데이터를 전달하면 되기 때문에 최선의 경우 네트워크 트래픽이 발생하지 않을 수도 있다. 물론, 다른 클라이언트가 해당 블럭을 갱신했을 경우는 서버로 부터 다시 읽어들여야 한다. 간단치 않은 문제인것만은 사실이다.



〈그림 51.2〉 **분산 파일 시스템 구조**

간단한 예를 통해 클라이언트/서버 기반의 분산 파일 시스템에는 두 가지 핵심적인 소프트웨어가 있어야 한다는 것을 알았다. 그 두 가지는 클라이언트 측 파일 시스템과

여담: 서버는 왜 크래시되는가

NFSv2 프로토콜의 구체적인 내용을 다루기 이전에 서버가 크래시되는 이유가 궁금했을 것이다. 이미 예상했겠지만 수많은 이유들이 있다. (일시적으로) **정전**이 되어 서버가 꺼질 수도 있다. 전원이 다시 복구가 되면 기계가 재시작될 것이다. 서버들은 보통 수십만 또는 수백만 줄의 코드로 구성이 되어 있다. 그러므로 **버그**들이 있으며 (아주 잘 만들어진 소프트웨어도 백줄 또는 천줄의 코드마다 버그가 있다) 그렇기 때문에 서버가 크래시하게 만드는 버그를 만나게 된다. 서버에는 메모리 누설(memory leakage)이 있을 수 있다. 아무리 작은 메모리 누설이라도 시스템의 메모리를 고갈시킬 수 있으며 크래시가 나도록 만든다. 마지막으로 분산 시스템에는 서버와 클라이언트 사이에 네트워크가 있다. 만약 네트워크가 이상하게 동작한다면(예를 들어, 네트워크 구간이 분할(**partitioned**) 되어서 클라이언트와 서버가 동작은 하지만 서로 통신이 안되는 경우), 마치 원격의 기계가 크래시가 난 것처럼 보일 수 있다. 하지만, 실제로는 네트워크를 통해서 도달할 수 없을 뿐이다.

파일 서버이다. 이 둘의 동작이 분산 파일 시스템의 행동을 결정한다. 이제 구체적으로 Sun의 네트워크 파일 시스템(Network File System, NFS)을 살펴보도록 하자.

51.2 NFS에 대하여

성공적인 분산 시스템 중 하나는 Sun Microsystems가 개발한 Sun Network File System(NFS)이다 [San86]. Sun은 NFS의 상업화 과정에서 독특한 방식을 취하였다. 내부를 공개하지 않는 전형적인 독점 시스템을 만드는 대신 클라이언트와 서버 간의 통신 메시지 형식을 정의하고, 이를 공개하였다. **오픈 프로토콜** 이다 [Sun89]. 해당 메시지 형식과 교환 규약을 따르기만 하면 서로 다른 회사가 개발한 제품들이 서로 통신이 가능하였다. 많은 그룹들이 독자적인 NFS 서버를 출시하였다. 이 전략은 제대로 들어맞았다. NFS 서버 시장 자체의 규모가 증가하였고, NFS 프로토콜을 사용하는 프로그램들이 많이 개발되었다. 요즈음은 NFS 서버를 판매하는 회사들이 여럿 있다 (Oracle/Sun, NetApp [HLM94], EMC, IBM 등을 포함). NFS가 성공적으로 확산될 수 있었던 것은 "오픈 마켓" 전략으로 인한 것이었다.

51.3 핵심: 단순하고 빠른 서버 크래시 복구

이번 장에서는 수년 동안 표준이었고 이제는 고전이 된 NFS 프로토콜에 대해서 설명하겠다(버전 2, 다른 이름은 NFSv2). NFSv3 [Paw+94]에는 작은 변화들이 있었고 NFSv4 [Paw+00]에서는 대대적으로 프로토콜이 변경되었다. NFSv2는 잘 제작되었지만 개선의 여지도 있기 때문에 설명하려는 목적에 잘 부합한다.

NFSv2 프로토콜을 설계할 당시의 핵심 목표는 서버의 *간단하고 빠른 크래시 복구* 였다. 여러 클라이언트와 단일 서버 환경을 고려하면 이 목표는 상당히 수긍이 간다.

서버가 다운이 되면(또는 접속이 안되면) 모든 클라이언트들은 (그리고 접속한 사용자들) 일을 할 수 없게 된다. 서버가 사라지면 전체 시스템도 사라진다.

51.4 빠른 크래시 복구의 열쇠: 상태를 유지하지 않음

NFSv2가 크래시 복구라는 간단한 목표를 달성하기 위해 **상태를 유지하지 않는(stateless)** 프로토콜을 설계하였다. 상태라는 것이 무엇일까. 서버는 각 클라이언트에서 서버에서 발생하고 있는 일에 관해 어떤 정보도 저장하지 않는다. 예를 들어, 서버는 클라이언트가 어떤 블럭을 캐싱하고 있는지, 어떤 파일들이 열려 있는지도 모르며, 또한 파일의 포인터의 현재 위치도 알지 못한다. 간단하게 말하면, 서버는 클라이언트가 무엇을 하든 전혀 상관하지 않는다. 대신에 각 요청에 필요한 모든 정보를 담아서 전송하도록 프로토콜이 설계되었다. 지금은 잘 이해가 되지 않더라도 프로토콜에 대해서 좀 더 자세히 살펴보면 이해하게 될 것이다.

대조적으로 **상태를 유지(stateful)** 하는 프로토콜의 중요한 예로 `open()` 시스템 콜이 있다. `open()`은 경로명이 전달되면, 파일 디스크립터를 반환한다(int 타입). 이 디스크립터는 `read()`나 `write()`를 통해 파일을 읽거나 쓰는 데 사용된다(공간 제약상 시스템 콜 결과에 대한 적절한 에러 검사 문장들은 생략되었다). 파일 시스템은 파일 디스크립터와 관련하여 상태를 제공한다. 여기서의 가장 중요한 '상태' 정보는 다음 입출력을 시작할 위치이다. 우리는 이것을 파일 오프셋이라한다. `read()`나 `write()` 호출하면 파일의 '현재 오프셋'으로부터 입출력을 수행하고, 해당 '현재 오프세' 정보를 새로이 갱신한다. 파일의 읽기 또는 쓰기와 관련된 상태를 지속적으로 갱신하는 것이다.

```
char buffer[MAX];
int fd = open("foo", O_RDONLY);  // "fd" 디스크립터 얻기
read(fd, buffer, MAX);           // foo에서 MAX 바이트 읽기 (fd를 통해)
read(fd, buffer, MAX);           // foo에서 MAX 바이트 읽기
...
read(fd, buffer, MAX);           // foo에서 MAX 바이트 읽기
close(fd);                       // 파일 닫기
```

〈그림 51.3〉 **클라이언트 코드: 파일에서 읽기**

클라이언트 측 파일 시스템이 파일을 열기 위해 서버에게 "파일 'foo'를 열고 해당 파일 디스크립터를 전달 바랍니다."라고 프로토콜 메시지를 전송했다고 해 보자. 파일 서버는 로컬 저장 장치에 있는 파일을 열어서 클라이언트에게 파일 디스크립터를 전송한다. 그 뒤로 파일 서버에게 읽기 요청을 할 때 클라이언트 파일 시스템은 전달하는 메시지에 해당 파일 디스크립터를 포함시켜서 "지금 전달하는 파일 디스크립터가 가리키는 파일의 일정 바이트를 읽어 달라."고 한다.

이 예제의 파일 디스크립터는 클라이언트와 서버 사이에서 **공유되는 정보(shared state)**의 일부분이다. Ousterhout는 이것을 **분산된 상태(distributed state)**라고 불렀다 [Ous91]). 클아이언트와 서버간에 상태정보가 공유되면, 크래시 과정이 복잡하게 된다. 서버가 첫 번째 읽기를 종료한 후, 클라이언트가 두 번째 요청을 전송하기 전에

서버가 크래시 되었다고 해보자. 서버가 다시 동작하게 되면 클라이언트는 두 번째 읽기 요청을 전송하게 된다. 여기서 문제가 발생한다. 클라이언트의 두 번째 읽기요청이 서버에 도착했을 때, 서버는 클라이언트가 명시한 `fd`가 실제 어떤 파일을 가리키는지 알 수가 없다. 파일 디스크립터 테이블은 메모리에만 존재하는 자료구조이다. 크래시와 함께 손실된다. 이 상황을 해결하려면 클라이언트와 서버는 어떤 형태로든 **복구 프로토콜**(**recovery protocol**)을 사용해야 한다. 예를 들어, 클라이언트는 서버가 요청을 처리하는 데 필요한 모든 정보를 메모리에 보관한다. 위 예제의 경우에는, 클라이언트가 파일 디스크립터 `fd`가 파일 `foo`를 가리킨다는 정보를 보관해야 한다.

서버와 클라이언트간의 공유상태를 유지할 경우, 서버도 클라이언트 측의 크래시를 고려해야 한다. 상황이 더 복잡해진다. 예를 들어, 클라이언트가 파일을 연 후에 크래시 되었다고 해 보자. `open()`은 서버의 파일 디스크립터를 사용하였다. 이때 서버가 해당 파일을 언제 닫아야 할까? 일반적인 경우 클라이언트가 `close()`를 호출하면, 그때 파일을 닫으면 된다. 하지만, 클라이언트가 크래시가 되면 서버는 `close()`를 절대로 받지 못할 것이고, 그렇기 때문에 클라이언트가 크래시되었다는 것을 서버에게 알려서 해당 파일을 닫을 수 있도록 해야 한다.

이러한 이유들로 NFS의 설계자들은 상태를 유지하지 않는 방식을 사용하기로 하였다. 상태정보가 유지 되지 않는다는 것의 의미가 뭘까? 클라이언트가 서버에게 파일 입출력 연산을 요청할때, 그때그때 필요한 정보를 모두 전송하는 것이다. 예를 들어, 읽을 위치, 쓸 위치등이 그것이다. 복잡한 작업과정에 대해 고민할 필요가 없어졌다. 서버는 다시 시작하면 되고, 클라이언트는 최악의 경우에 요청을 재시도하면 된다.

51.5 NFSv2 프로토콜

이제 NFSv2 프로토콜을 정의해 보자. 우리가 해결해야 할 문제는 간단하다.

핵심 질문: 어떻게 상태를 유지하지 않는 프로토콜을 정의할까

어떻게 상태를 유지하지 않고도 연산이 가능하도록 네트워크 프로토콜을 정의할까? 분명한 것은 `open()`과 같은 상태를 유지하는 호출은 논의에서 제외되어야 한다(서버는 열려있는 파일을 관리해야 하기 때문이다). 하지만, 클라이언트 응용 프로그램은 `open()`, `read()`, `write()`, `close()`와 다른 표준 API 호출들을 사용하여 파일들과 디렉터리들을 접근하고 싶을 것이다. 그러므로 질문을 가다듬어보자. 상태를 유지하지 *않으면서* POSIX 파일 시스템 API를 지원하는 프로토콜을 정의하는 방법은 무엇인가?

NFS 프로토콜의 설계를 이해하기 위해 필요한 핵심 개념 중 하나는 **파일 핸들**(**file handle**)이다. 파일 핸들은 특정 연산을 수행할 파일이나 디렉터리를 고유하게 설명하는 데 사용된다. 디스크 기반 파일 시스템에서의 파일 디스크립터와 유사한 개념이다. 많은 프로토콜 요청들이 파일 핸들이 전달된다. 파일 핸들은 *볼륨 식별자*와 *아이노드 번호* 그리고 *생성 번호*의 세 가지 주요 인자로 구성된다. 서버는 이 세 가지 정보를 조합하여

클라이언트가 원하는 파일이나 디렉터리를 식별한다. 볼륨 식별자는 해당 요청이 어떤 파일 시스템을 대상으로 하고 있는지 서버에게 알려준다(NFS 서버는 하나 이상의 파일 시스템 파티션을 제공할 수 있다). 아이노드 번호는 그 파티션 내에서 해당 요청이 접근하고자 하는 파일을 나타낸다. 마지막으로 생성 번호는 아이노드 번호를 재사용하기 위해 필요한 것으로 아이노드 번호를 재사용할 때마다 증가한다. 서버는 생성 번호를 통해 오래된 파일 핸들을 사용하는 클라이언트가 실수로 새로 할당된 파일을 접근하는 경우를 방지한다.

프로토콜의 몇 가지 중요한 부분들을 정리하였다. 전체 프로토콜은 다른 자료를 참고하도록 하자(NFS에 대한 자세한 설명은 Callaghan의 책을 참고하자 [Cal00]).

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

〈그림 51.4〉 **NFS 프로토콜: 예제**

프로토콜의 중요한 구성 요소들을 간단하게 짚고 넘어가겠다. 먼저 LOOKUP 프로토콜 메시지는 파일 핸들을 얻기 위해 사용된다. 파일 핸들은 추후 파일 데이터를 접근하는 데 사용된다. 클라이언트는 원하는 대상의 파일 이름과 파일이 속한 디렉터리의 파일 핸들을 전달한다. 서버는 클라이언트에게 해당 파일(또는 디렉터리)의 핸들과 더불어 해당 파일의 속성 정보를 전달한다.

예를 들어 보겠다. 클라이언트가 파일 시스템의 루트 디렉터리(`/`)에 대한 디렉터리 파일 핸들을 이미 갖고 있다(정확히 말하자면, 루트 디렉터리의 정보는 NFS **마운트 프로토콜**(**mount protocol**)을 통해서 얻을 수 있다. 마운트 프로토콜은 클라이언트와 서버가 처음으로 연결되는 프로토콜로 논의를 간단하게 하기 위해 다루지 않겠다). 만약

클라이언트 프로그램이 `/foo.txt` 파일을 열고자 할 경우, 클라이언트 측 파일 시스템은 서버로 LOOKUP 요청을 전송하면서 파일 이름 `foo.txt`와 루트 파일 핸들을 같이 전달한다. 성공하면 `foo.txt`의 파일 핸들(그리고 속성 정보)이 리턴된다.

"속성"은 파일 시스템이 각 파일에 대하여 관리하고 있는 메타데이터로서 파일 생성 시간, 마지막 수정 시간, 크기, 소유와 권한 정보와 같은 것들을 포함한다. 즉, 어떤 파일에 대하여 `stat()`를 호출하여 얻을 수 있는 정보를 말한다.

파일 핸들이 준비되었다면 클라이언트는 각 파일을 읽거나 쓰기 위해 READ와 WRITE 프로토콜 메시지를 보낼 수 있다. READ 프로토콜 메시지는 파일의 핸들과 파일 내에서의 오프셋 그리고 읽을 바이트 길이를 같이 전달하도록 정의하고 있다. 그러면 서버가 읽기를 수행할 수 있으며(결국에는, 핸들이 서버에게 알려주는 정보도 어떤 볼륨에서 어떤 아이노드에 해당하는 파일을 읽어야 하며 해당 파일의 오프셋만큼 떨어진 위치에서 읽을 바이트의 수이기 때문이다) 데이터를 클라이언트에게 리턴할 수 있다(실패하였다면 에러를 전송한다). WRITE도 비슷하게 처리된다. 다른 점이 있다면 클라이언트에서 서버로 데이터가 전달되며 성공 시에 성공 코드만 리턴된다.

마지막으로 흥미로운 프로토콜 메시지는 GETATTR 요청이다. 이 요청은 어떤 파일 핸들에 대해 해당 파일의 마지막 수정 시간을 포함하는 속성 정보를 가져온다. NFSv2에서 이 프로토콜 요청이 왜 중요한지를 잠시 후 캐싱을 다룰 때 살펴보도록 하겠다(왜 그런지 생각해 보기 바란다.)

51.6 프로토콜에서 분산 파일 시스템으로

이제 이 프로토콜을 이용하여 어떻게 클라이언트 파일 시스템과 파일 서버를 연동하여 하나의 파일 시스템을 구성할 수 있는지에 대해 대충 감을 잡았기 바란다. 클라이언트 측 파일 시스템은 `open()`된 파일들을 관리하며, 응용 프로그램의 요청을 적절한 프로토콜 메시지들로 변환하는 일을 한다. 서버가 하는 일은 간단하다. 각 요청에 대해 답하는 것이다. 클라이언트가 보내는 요청은 수행에 필요한 모든 정보를 담고 있다.

예를 들어, 파일을 읽는 간단한 응용 프로그램을 생각해 보자. 응용 프로그램의 시스템 콜을 클라이언트 측 파일 시스템과 파일 서버가 처리하는 과정을 도식적으로 표현하고 있다(그림 51.5).

도표에서 몇가지 주요사항을 설명하겠다. 첫째, 클라이언트가 파일 연산의 "**상태**"를 어떻게 관리하는 지를 주의해서 보기 바란다. 클라이언트는 정수형으로 표현되는 파일 디스크립터와 이에 연결된 NFS 파일 핸들에 대한 정보, 그리고 현재의 파일 오프셋 등을 저장한다. 이러한 정보를 다 관리하고 있기 때문에 클라이언트는 각 읽기 요청(읽기의 오프셋을 명시적으로 정하지 않았다는 것을 알았을 것이다)을 적절하게 포맷된 읽기 프로토콜 메시지로 변환하여 서버가 파일의 정확히 어느 부분의 바이트를 읽어야 할지를 알 수 있도록 한다. 성공적으로 읽기가 처리되면 클라이언트는 현재 파일 위치를 갱신한다. 연이은 읽기가 같은 파일 핸들을 사용하지만 오프셋은 다른 값을 갖는다.

둘째로, 서버와 언제 상호작용하는지 알았을 것이다. 파일이 처음으로 열렸다면, 클라이언트 측 파일 시스템은 LOOKUP 요청 메시지를 전송한다. 정확히 말하자면,

클라이언트	서버
fd=open(‘/foo’,...);	
LOOKUP(rootdir FH, “foo”) 전송	
	LOOKUP 요청 수신
	루트 디렉터리에서 “foo” 검색
	foo의 FH + 속성 정보 리턴
LOOKUP 응답 수신	
열린 파일 테이블에 파일 디스크립터 할당	
foo의 FH를 테이블에 저장	
현재 파일 위치 (0) 저장	
응용 프로그램에게 파일 디스크립터 리턴	
read(fd, buffer, MAX);	
열린 파일 테이블에서 fd 인덱스 찾기	
NFS 파일 핸들(FH)을 얻음	
현재 파일 위치를 오프셋으로 사용	
READ(FH, ofsset=0, count=MAX) 전송	
	READ 요청 수신
	FH를 사용하여 볼륨/아이노드 번호 얻음
	디스크(또는 캐시)에서 아이노드를 읽음
	블럭 위치를 계산(오프셋을 활용)
	디스크(또는 캐시)에서 데이터 읽기
	클라이언트에게 데이터 리턴
READ 응답 수신	
파일 위치 갱신(+읽은 바이트)	
현재 파일 위치 = MAX로 설정	
데이터/에러 코드를 프로그램에게 전달	
read(fd, buffer, MAX);	
offset=MAX인 점만 제외하면 동일하며 현재 파일의 위치는 = 2*MAX로 설정	
read(fd, buffer, MAX);	
offset=2*MAX인 점만 제외하면 동일하며 현재 파일의 위치는 = 3*MAX로 설정	
close(fd);	
로컬 자료 구조들을 청소	
디스크립터“fd”를 열린 파일 테이블에서 해제	
(서버에게 알릴 필요 없음)	

〈그림 51.5〉 **파일 읽기 : 클라이언트 측과 서버 측의 동작**

팁: 멱등성(IDEMPOTENCY)은 강력하다

반복해서 연산을 수행하더라도 동일한 값이 유지되는 성질을 멱등성이라 한다[1]. **멱등성(idempotency)**은 시스템의 신뢰성을 담보할 수 있는 매우 유용한 성질이다. 연산이 성공할 때 까지 계속 연산을 요청하면 된다. 멱등성이 없는 연산이라면, 삶이 고달파진다.

아주 긴 경로명을 따라가야 한다면(예, `/home/remzi/foo.txt`), 클라이언트는 세 번의 LOOKUP 메시지를 보낼 것이다. / 디렉터리에서 home 디렉터리를 검색하기 위해서 한 번, `home`에서 `remzi`를 검색하기 위해서 한 번, 그리고 마지막으로 `remzi`에서 `foo.txt`를 탐색하기 위해서 한 번이다.

셋째로, 서버에게 보내는 요청에는 그 요청을 완료하는 데 필요한 모든 정보가 담겨 있다는 것을 알았을 것이다. 이제 상세하게 논의하겠지만, 서버가 실패했을 때 적절하게 복구할 수 있도록 만들어 주는 핵심 설계 요소이다. 서버가 상태 정보 없이도 요청에 응답할 수 있도록 한다.

51.7 서버의 고장을 멱등연산으로 처리하기

클라이언트가 서버로 메시지를 전송했을 때, 응답이 없는 경우가 있다. 여러 가지 원인이 있을 수 있다. 네트워크 상에서 메시지가 손실되었을 수 있다. 실제 환경에서 네트워크 상에서 메시지는 손실된다. 요청 메시지가 손실되었을 수도 있고, 응답 메시지가 손실되었을 수도 있다. 두 경우 모두, 클라이언트는 응답을 받지 못하게 된다.

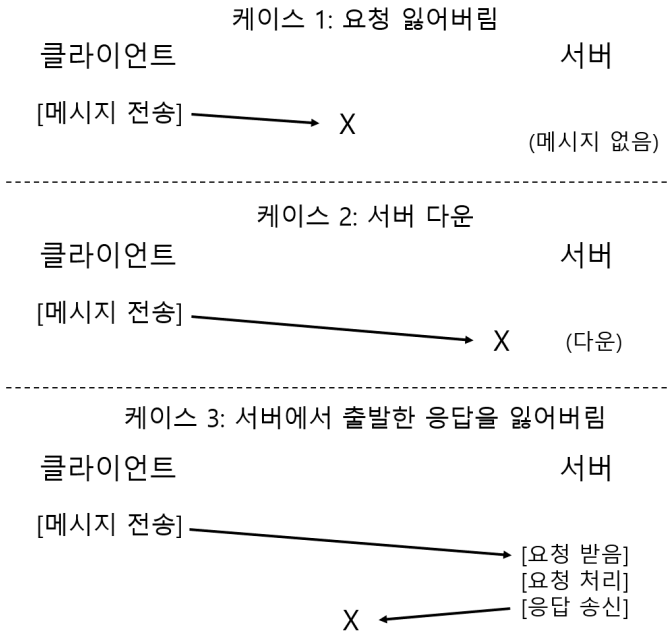
서버가 크래시되어 메시지에 응답하지 못할 수도 있다. 얼마 후 서버가 다시 시작되겠지만, 그동안 전송된 모든 요청들은 손실될 것이다. 이와 같은 모든 경우들에서 클라이언트 입장에서는 다음과 같은 질문을 할 수 있다. 서버가 제때에 응답을 하지 않으면 무엇을 해야 할까?

NFSv2에서 클라이언트는 모든 종류의 고장들을 하나의 일관성 있고 정교한 방식으로 대처한다. 요청을 *재전송* 한다. 구체적으로 클라이언트는 요청을 전송한 후 타이머를 설정한다. 타이머가 종료하기 전에 응답이 오면, 해당 타이머는 취소된다. 모든 것은 잘 되고 있다. 응답을 받기 *전에* 타이머가 종료한다면, 클라이언트는 요청 처리가 되지 않았다고 간주하고 재전송 한다. 만약 서버의 응답이 도착하면, 문제는 해결된 것이다.

재시도를 통해 문제를 쉽게 해결할 수 있는 근본 원인은 NFS 요청이 갖고 있는 중요한 성질에 기인한다. 바로 연산의 **멱등성(idempotent)**이다. 어떤 연산이 멱등연산이라고 불린다면 그 연산을 여러 차례 수행하여 얻는 결과가 한 번만 수행하였을 때 얻는 결과와 동일하다는 것을 뜻한다. 예를 들어, 메모리의 특정 위치에 어떤 값을 세 번 저장한다면, 그것은 한 번만 저장하는 것과 같은 결과를 가져온다. "메모리에 값을 저장"이라는 연산은 멱등연산이다. 반면 카운터를 세 번 증가하는 것은 한 번만 하는 것과 결과가 다르다. "카운터 증가"는 멱등연산이 아니다. 데이터를 읽기만 하는 연산은 당연하게

멱등연산이다. 데이터 갱신 연산의 경우에는 멱등연산인지 판단하기 위해 좀 더 세심한 주의를 기울여야 한다.

NFS에서 크래시 복구 설계의 핵심은 주요 연산들의 멱등성이다. LOOKUP과 READ 요청은 파일 서버에서 정보를 읽기만하고 갱신을 하지 않기 때문에 평범한 멱등연산이다. 좀 더 흥미로운 것은 WRITE 요청도 멱등연산이란 것이다. 예를 들어, WRITE가 실패했다면 클라이언트는 단순히 그 연산을 재시도하면 된다. WRITE 메시지는 데이터와 길이 그리고 데이터를 써야 할 (중요한) 정확한 오프셋을 포함한다. 그러므로 여러 번 쓰기 연산을 하더라도 한 번 수행한 결과와 동일하기 때문에 이 연산을 반복할 수 있다.



〈그림 51.6〉 **세 종류의 손실**

이와 같은 방식으로 클라이언트는 모든 타임아웃들을 일관된 방식으로 다룰 수 있다. WRITE 요청이 손실되었다면 (케이스 1), 클라이언트는 쓰기 요청을 재전송한다. 서버는 쓰기를 실행한다. 모든 것은 정상이 된다. 처음 요청을 전달한 시점에는 서버가 다운이었는데, 두 번째 요청 때는 다시 살아나서 실행 중인 경우에도 똑같이 모든 동작이 원하는 대로 수행된다(케이스 2). 마지막 경우로, 서버가 WRITE 요청을 받아 디스크로 쓰기 요청을 하였고 클라이언트도 응답을 전송하였다. 그런데 이 응답이 손실되었다면 (케이스 3) 클라이언트가 재요청 할 수도 있다. 서버가 이 요청을 다시 받게 되면 처음 받은 것처럼 똑같이 다시 수행한다. 서버는 데이터를 디스크에 쓰고 이전과 같이 다시 응답한다. 만약 클라이언트가 이번에는 응답을 받았다면, 모든 것이 정상이 된다. 이로써 클라이언트는 두 번의 메시지 손실과 서버 고장을 일관성있게 해결했다. 깔끔하다!

여담이기는 하지만, 어떤 연산들은 멱등연산으로 만들기 어렵다. 예를 들어, 이미

팁 : 완벽하게 만드는 것은 잘 만드는 것의 적이다.(Perfect is the enemy of the good, Voltaire's Law)

아름다운 시스템을 설계하였다 하더라도 때로는 모든 예외 상황들에서 원하는 대로 정확히 동작하지 않을 수 있다. 앞선 mkdir 예제를 돌이켜 보자. mkdir이 다른 시맨틱을 갖도록 재설계하여서 멱등연산이 가능하도록 만들 수도 있을 것이다(어떻게 설계할 수 있을지 생각해 보라). 그렇지만 왜 그렇게까지 해야 할까? NFS 설계 철학은 대부분의 중요한 케이스들을 다루며, 대체적으로 고장을 처리하기 위한 시스템 설계가 깔끔하고 간단하다. 그러므로 인생이 완벽하지 않다는 것을 인정하고 그런대로 시스템을 만드는 것이 좋은 엔지니어링의 시그널이다. Voltaire도 이렇게 언급한 적이 있다. "... 현명한 이탈리아인이 말하길, 모든 선의 적은 최고이다." 그래서 **Voltaire's Law**라고 한다 [aVol72].

존재하는 디렉터리를 만들려고 했는데 mkdir 요청이 실패했다는 정보를 받았다. NFS에서 파일 서버가 MKDIR 프로토콜 메시지를 받아서 성공적으로 처리하였지만 응답이 도중이 손실이 되었다. 그 사실을 모르는 클라이언트는 요청을 반복하지만 첫 번째 요청 때에 이미 성공적으로 디렉터리를 생성한 서버는 클라이언트에게 실패했다고 알린다. 하지만 사실은 첫 요청에 이미 처리를 성공적으로 하였기 때문에 재시도를 실패한 것뿐이다. 이와 같이, 인생은 완벽하지 않다.

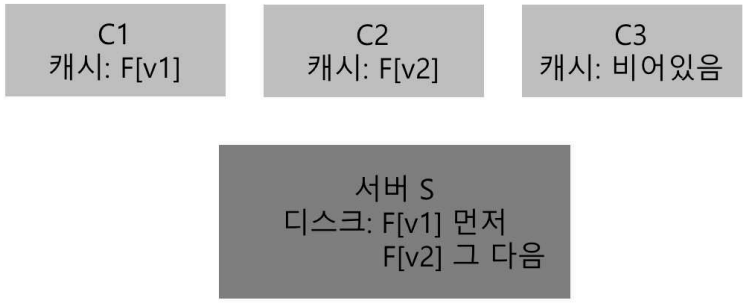
51.8 성능 개선하기 : 클라이언트 측 캐싱

분산 파일 시스템은 많은 장점이 있지만 모든 읽기와 쓰기 요청들을 네트워크를 통해 전송해야 한다는 것 때문에 성능 상의 문제가 있다. 일반적으로 로컬 메모리나 디스크에 비하면 네트워크는 빠르지 않다. 분산 파일 시스템의 성능을 어떻게 개선할 수 있을까?

해법은 제목에 굵은 글씨로 크게 써있는 것을 봐서 알겠지만 클라이언트가 **캐싱**을 하는 것이다. NFS 클라이언트 측 파일 시스템은 서버에서 읽은 파일 데이터(그리고 메타데이터)를 클라이언트 메모리에 캐싱한다. 첫 접근은 비싸더라도(즉, 네트워크 통신이 필요) 그 이후의 접근은 클라이언트 측 메모리에서 상당히 빠르게 서비스 될 수 있다.

캐시는 쓰기를 위한 임시 버퍼로서 사용되기도 한다. 클라이언트 응용 프로그램이 파일에 처음으로 쓴다면 클라이언트는 데이터를 서버로 내보내기 전에 클라이언트 측 메모리(파일 서버에서 받은 데이터를 읽을 때 사용한 동일한 캐시)에 저장한다. **쓰기 버퍼링(write buffering)**은 `write()`의 소요 시간과 실제 쓰기 소요 시간을 분리시키기 때문에 매우 유용하다. 즉, 응용 프로그램이 호출한 `write()`는 즉시 리턴하며(데이터를 클라이언트 측 파일 시스템의 캐시에 넣는다), 일정 시간 후에 파일 서버에 기록된다.

NFS 클라이언트는 데이터를 캐싱하기 때문에 성능이 대부분 좋게 나온다. 다된 것 같다, 그렇지 않은가? 불행하게도 아직은 아니다. 클라이언트 캐시를 갖고 있는 대다



〈그림 51.7〉 **캐시 일관성 문제**

수의 시스템은 **캐시 일관성**(**cache consistency**) 유지라는 거대하고 흥미로운 문제과 직면한다.

51.9 캐시 일관성 문제

캐시 일관성 문제를 두 개의 클라이언트와 하나의 서버의 구성예를 통해 설명하자. 클라이언트 C1이 파일 F를 읽고 그 파일의 사본을 로컬 캐시에 보관한다고 해 보자. 그리고 다른 클라이언트 C2가 그 파일 F를 덮어써서 내용을 변경하였다. 새로운 버전을 파일 F(version 2) 또는 F[v2]라 하고 이전 버전을 F[v1]라고 하여 두 개를 구분할 수 있도록 하자(하지만 이름만 같을 뿐 내용은 다르다). 마지막으로 세 번째 클라이언트 C3는 아직까지 파일 F를 접근한 적이 없다.

아마도 문제를 이미 예견했을 것이다(그림 51.7). 문제는 두 가지로 세분화 된다. 첫 번째 문제는 클라이언트 C2가 쓰기를 잠시 캐시에 보관하기 때문에 발생한다. F[v2]가 C2의 메모리에 있는 동안 다른 클라이언트(C3이라 하자)가 F를 접근하면 파일의 이전 버전(F[v1])을 가져가게 된다. 클라이언트가 쓰기 버퍼링을 사용하면 다른 클라이언트는 해당 파일의 예전 버전을 읽을 가능성이 있다. 당신이 클라이언트 C2에 접속하여 F를 갱신하고, 바로 C3에 접속하여 해당 파일을 읽을 경우, 이전 버전을 읽게 될 것이다! 좌절스런 경험이 될 것이다. 캐시 일관성 문제를 **갱신 가시성**(**update visibility**)이라고 한다. 한 클라이언트에서의 변경 결과가 언제 다른 클라이언트에게 보여질까?

두 번째 캐시 일관성 문제는 **오래된 캐시**(**stale cache**)이다. C2가 파일 서버로 쓰기를 내려 보내서 서버가 최신 버전(F[v2])을 보관하고 있다. 반면에 C1은 여전히 F[v1]을 캐시에 갖고 있다고 하자. C1에서 파일 F를 읽는다면 최신 버전인 F[v2]가 아닌 (대부분) 오래된 버전 F[V1]를 일게 될 것이다.

NFSv2는 이러한 캐시 일관성 문제를 두 가지 방법으로 해결하고 있다. 첫째로 갱신의 가시성을 해결하기 위해서, 클라이언트는 **닫을-때-내보냄**(**flush-on-close**, 다른 이름으로는 **열기-전에-닫음**(**close-to-open**)) 일관성 시맨틱이라는 것을 사용한다. 구체적으로 살펴보자. 응용 프로그램이 파일을 갱신하여 그 파일을 닫는 시점에, 갱신 내용(즉, 캐시의 더티 페이지들)을 서버에 보낸다. Flush-on-close 기법에서는, NFS는 어떤 파일을 닫았을 때, 그 이후에 다른 노드에서 해당 파일을 열면, 닫겨진 시점의 최신

파일 내용을 읽는다. 만약 클라이언트가 파일을 닫을 때 해당 파일 블럭들을 플러시하지 않는다면, 다른 클라이언트들은 `close()` 시점의 최신 내용들을 읽을 수 없다.

둘째로 오래된 캐시 문제를 해결하기 위해, NFSv2 클라이언트는 캐시에 보관된 내용을 사용하기 전에 파일의 변경 여부를 먼저 검사한다. 구체적으로는 다음과 같다. 클라이언트에서 파일을 열 때 GETATTR 요청을 서버로 전송하여 파일의 속성 정보를 가져온다. 속성에는 파일이 서버에서 마지막으로 갱신된 시간을 나타내는 정보가 포함되어 있다. 만약 갱신 시점이 파일이 클라이언트에 캐싱된 이후라면, 클라이언트는 캐시된 파일을 무효화(invalidate)한다. 클라이언트는 캐시에서 파일을 제거하고 서버로 읽기 요청을 전달하여 파일의 최신 버전을 가져온다. 만약 클라이언트 자신이 갖고 있는 파일이 최신 버전이라면 캐시에 있는 내용을 그대로 사용하여 성능을 높일 수 있다.

Sun의 개발 팀이 오래된 캐시 문제에 대한 해법을 개발하던 도중 새로운 문제를 발견하였다. NFS 서버에 갑자기 GETATTR 요청이 폭발적으로 증가한 것이다. 가장 바람직한 설계 원칙은 대표적인 경우(**common case**)를 잘 처리하는 방법을 만드는 것이다. 여기서 대표적인 경우란 하나의 클라이언트가 파일을(어쩌면 반복적으로) 사용하고 있더라도, 서버에 GETATTR 요청을 보내서 파일 변경 여부를 확인할 수밖에 없다는 것이다. 아무도 파일을 변경할 가능성이 없는 데도 불구하고, 클라이언트는 "누가 파일을 변경했어?"라고 계속적으로 서버에 질문하는 것이다.

이 상황을(어느 정도) 해결하기 위해 **속성 정보 캐시**(**attribute cache**)라는 것을 클라이언트에 추가하였다. 클라이언트는 여전히 파일을 접근하기 전에 그것이 최신본인지를 검사해야 하지만, 이를 위해 대부분의 경우 속성 정보를 가져오기 위해서 캐시되어 있는 속성 정보를 사용할 것이다. 파일의 속성 정보는 파일을 처음 접근할 때 캐시에 저장되며, 일정 시간이 지나면 캐시는 타임아웃된다(약 3초 정도). 그 3초 동안에는 캐시된 파일을 사용해도 괜찮다고 판단할 것이며, 서버와 통신은 발생하지 않는다.

51.10 NFS의 캐시 일관성 기법에 대한 평가

NFS 캐시 일관성 해결 기법에 대해 마지막으로 몇 가지 사항을 언급하고자 한다. 닫을-때-내보냄(flush-on-close) 방식은 이치에 맞기는 하지만 성능상 문제가 있다. 구체적으로는, 클라이언트에서 파일을 일시적으로 생성했다가 바로 삭제해도 그 파일 내용은 무조건 서버로 전달된다. 이상적으로는 그런 짧은 수명의 파일은 삭제되기 전까지 메모리에만 유치하고 애초에 서버와 통신을 하지 않도록 해서 성능을 개선할 수 있다.

더 중요한 것은 속성 정보 캐시로 인해 사용자는 자신이 어떤 버전의 파일을 읽고 있는지를 파악하는 것이 더욱 어렵게 되었다. 어떤 때는 최신 버전을 읽게 되고 또 어떤 때는 예전 버전을 읽을 수 있다. 캐시에 있는 속성정보의 유효성 때문이다. 타임아웃이 되지 않은 속성정보는 유용하다고 간주된다. 대부분의 경우엔 큰 문제가 없지만, 때때로 (꼭 일어난다!) 가끔씩 시스템이 오작동하는 원인이 된다.

NFS 클라이언트 캐싱이라는 개념을 설명하였다. 일반적으로는 시맨틱에 의해 구현 내용이 결정되는데, 이 경우는 구현의 세부 내용에 의해 시멘틱이 결정되는 예이다.

51.11 서버 측 쓰기 버퍼링의 의미

지금까지의 주 내용은 클라이언트 측의 캐싱이었으며, 흥미로운 문제들이 많이 발생했다. 하지만, NFS 서버들도 대용량 메모리를 갖고 있기 때문에 그 자체도 캐싱 문제를 갖고 있다. 디스크에서 데이터를(메타데이터와) 읽으면, NFS 서버는 그것을 메모리에 보관한다. 다음 번 해당 데이터(그리고 메타데이터)에 대한 읽기는 디스크에서 서비스되지 않는다. 캐싱으로 인해 잠재적인(약간의) 성능이 개선된다.

쓰기 버퍼링의 경우가 좀 더 흥미롭다. NFS 서버는 WRITE 프로토콜 요청을 받으면 저장 장치(예, 디스크 또는 다른 영속 저장 장치)에 완전히 쓴 후에 리턴한다. 데이터를 서버의 메모리에만 저장한 후 클라이언트에게 알릴 수도 있겠지만 그렇게 하면 잘못된 결과를 얻을 수 있다. 왜 그럴까?

그 해답은 클라이언트가 서버 측 실패를 다루는 방법에 대한 가정에 있다. 클라이언트가 다음과 같은 쓰기 문장들을 실행했다고 해 보자.

```
write(fd, a_buffer, size); // 첫 번째 블럭을 a로 채움
write(fd, b_buffer, size); // 두 번째 블럭을 b로 채움
write(fd, c_buffer, size); // 세 번째 블럭을 c로 채움
```

이 쓰기들은 파일의 첫 블럭을 a로 채우고 그 다음은 b로 그리고 그 다음을 c로만 덮어 쓴다. 처음에는 파일이 다음과 같았다고 하자.

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

세 번의 쓰기를 수행하고 나서 보이는 최종적인 모습은 다음과 같을 것이다. 원래의 x와 y 그리고 z가 a와 b 그리고 c로 덮어써졌다.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

클라이언트가 세 개의 쓰기 요청을 세 개의 쓰기 프로토콜로 전송했다 가정하자. 첫 번째 WRITE 메시지를 서버가 디스크로 요청하였으며 클라이언트가 성공했다는 답신을 받았다. 이제, 두 번째 쓰기 내용을 메모리 버퍼에 저장했고, 해당 내용을 디스크에 저장하기 *전에* 성공했다고 클라이언트에게 답신했다. 불행하게도, 디스크에 두 번째 요청 내용이 저장되기 전에 시스템이 크래시 되었다. 서버는 신속히 재시작하였고 세 번째 요청을 받아 성공적으로 처리를 하였다.

클라이언트는 모든 요청이 성공적으로 처리된 것으로 알았는데, 파일의 내용이 다음과 같아서 놀라게 된다.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <-- 저런
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

이런! 서버가 두 번째 쓰기 내용을 디스크에 완전히 저장하기 전에 디스크에 예전 내용이 있는 상황에서 클라이언트에게 답신을 보냈기 때문에 문제가 발생하였다. 응용 프로그램에 따라서는 큰 문제가 될 수도 있는 상황이다.

이러한 문제를 피하기 위해서 NFS 서버는 클라이언트에게 성공했다고 알리기 전에 *반드시* 각 쓰기를 안정적인(영속적인) 저장 장치에 커밋해야 한다. 그렇게 해야만 클라이언트가 쓰기 도중에 서버가 실패하는 경우를 파악할 수 있으며 성공할 때까지 재시도 할 수 있다. 앞의 예제처럼 파일에 이전 내용과 최신 내용이 무질서하게 섞여 있는 경우를 방지할 수 있다.

이러한 요구 조건을 만족시키도록 NFS 서버를 구현하면 쓰기 성능에 문제가 생긴다. 세심하게 다루지 않으면 중대한 성능 병목의 원인이 될 수도 있다. 사실, 어떤 회사들은 (예, NetApp) 쓰기를 빠르게 할 수 있는 NFS 서버를 간단히 구현하였다. 첫 번째는 메모리에 배터리를 연결하여 데이터를 디스크에 기록하지 않고도 손실 위험없이 빠르게 WRITE 요청에 응답할 수 있도록 하였다. 두 번째로는 쓰기 성능에 최적화된 파일 시스템을 개발하였다 [HLM94; RO91].

51.12 요약

NFS 분산 파일 시스템에 대해 살펴보았다. NFS는 서버 크래시에 대한 간단하고 신속한 복구에 중점이 맞춰져 있으며 그것을 달성하기 위해서 프로토콜을 정교하게 설계하였다. 연산의 멱등성이 핵심이다. 클라이언트가 실패한 연산을 재시도하기 때문에, 서버의 결과가 해당 요청을 실행했었는지의 여부와 무관해야 하기 때문이다.

클라이언트와 단일 서버로 구성된 시스템에 캐시를 사용하면 복잡한 일들이 생긴다는 것을 보았다. 시스템이 제대로 동작하기 위해서는 캐시 일관성 문제를 해결해야 한다. NFS는 이 문제를 상황 별로 즉흥적인 방식으로 해결을 하고 있기 때문에 가끔씩 이상한 결과가 발생한다. 마지막으로 서버 측의 캐싱도 까다롭다는 것을 살펴보았다. 서버는 요청받은 쓰기에 대해 성공했다고 알리기 전에 안정적인 저장 장치로 먼저 내려보내야 한다(그렇지 않으면 데이터를 잃어버릴 수도 있다).

관련된 모든 문제를 다 언급하지는 못했다. 중요한 예는 보안이다. 초기의 NFS 구현은 보안에 대해서는 허술하였다. 클라이언트 중 어느 사용자든 다른 사용자인 것처럼 위장하기가 너무나 쉬워서 거의 모든 파일에 대한 접근 권한을 얻을 수가 있었다. 이후에 좀 더 신중한 인증 서비스(예, Kerberos [NT94])와 접목이 되면서 이와 같은 부분들이 개선되었다.

참고 문헌

[aVol72] **"La Begueule"**
Francois-Marie Arouet a.k.a. Voltaire
볼테르는 수많은 재치 있는 말들을 했다. 이것은 그 중의 한 예일 뿐이다. 예를 들어, 볼테르가 한 번은 이렇게 말했다. "당신의 나라에 두 개의 종교가 있다면, 그 둘은 서로의 목을 따려고 할 것이다. 하지만 종교가 서른 개쯤 있으면 평안하게 살 것이다." 민주당과 공화당 사람들, 뭐라고 말하고 싶으신가?

[Cal00] **"NFS Illustrated"**
Brent Callaghan
Addison-Wesley Professional Computing Series, 2000
NFS에 대한 훌륭한 참고 문헌이다. 놀랍도록 자세하며, 또한 각 프로토콜에 대해서 상세하게 설명하였다.

[HLM94] **"File System Design for an NFS File Server Appliance"**
Dave Hitz, James Lau, and Michael Malcolm
USENIX Winter 1994, San Francisco, California, 1994
Hitz 등은 이전의 로그 기반의 파일 시스템에서 큰 영향을 받았다.

[NT94] **"Kerberos: An Authentication Service for Computer Networks"**
B. Clifford Neuman and Theodore Ts'o
IEEE Communications, 32(9):33-38, September 1994
커버러스(Kerberos)는 초기의 그리고 엄청나게 영향력 있는 인증 서비스이다. 언제가는 이 주제로 한 장을 써야 할 것 같다.

[Ous91] **"The Role of Distributed State"**
John K. Ousterhout
URL: `ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps`
분산 상태에 대한 설명이지만 흔하게 참조되는 글은 아니다. 문제와 도전거리에 대한 광범위한 시각을 보여준다.

[Paw+94] **"NFS Version 3: Design and Implementation"**
Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz
USENIX Summer 1994, pages 137-152
NFS 버전 3의 기반이 되는 작은 변경들에 대해 소개하고 있다.

[Paw+00] **"The NFS version 4 protocol"**
Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow
2nd International System Administration and Networking Conference (SANE 2000)
의심할 여지 없는 NFS에 대해 기록된 가장 문학적인 논문이다.

[RO91] **"The Design and Implementation of the Log-structured File System"**
Mendel Rosenblum and John Ousterhout
또 LFS다. 누구도 LFS는 이제 그만이라고 할 수 없다.

[San86] **"The Sun Network File System: Design, Implementation and Experience"**
Russel Sandberg
USENIX Summer 1986

NFS에 대한 원조 논문이다. 읽기에 조금 도전적인 자료이긴 하지만 훌륭한 개념들의 원전을 읽어볼 수 있기 때문에 가치가 있다.

[Sun89] **"NFS: Network File System Protocol Specification"**
Inc. Sun Microsystems
URL: `http://www.ietf.org/rfc/rfc1094.txt`
읽기 두려운 명세서이다. 꼭 읽어야 한다면 읽어보자. 즉, 누군가 읽으라고 돈을 주면 읽어보자. 엄청나게 많이 받길 바란다. 현금으로!