대규모 분산 파일 시스템 환경의 메타 데이터 관리

Metadata Management Techniques in a Large Distributed File System Environment

차명훈 (M.H. Cha) 저장시스템연구팀 연구원
이상민 (S.M. Lee) 저장시스템연구팀 선임연구원
김 준 (J. Kim) 저장시스템연구팀 책임연구원
김영균 (Y.K. Kim) 저장시스템연구팀 책임연구원
김명준 (M.J. Kim) 인터넷서버그룹 그룹장

목 차

Ⅰ. 서론
Ⅱ. 메타 데이터 서버 클러스터링 방법
Ⅲ. 메타 데이터 클러스터링 분석
Ⅳ. 결론

메타 데이터와 데이터의 처리 경로를 독립시킨 분산 파일 시스템 구조는 입출력 성능 향상 및 확장성 용이라는 측면에서 현재 주도적인 아키텍처로 사용되고 있다. 이러한 환경에서 클라이언트 및 데이터 서버의 수가 계속 확장되어 전체 시스템 규모가 페타(peta) 바이트급 이상 처리가 가능한 대규모로 진화될 경우 필연적으로 메타 데이터 서버에 병목 현상이 발생하게 된다. 본 고에서는 이러한 문제를 처리하기 위한 아키텍처로서 메타 데이터 서버들의 클러스터링을 고려하며, 이를 위해 제안된 다양한 기술들의 동작 원리 및 장단점 등을 분석하고 고찰해 보기로 한다.

I. 서론

인터넷 이용자가 급격하게 증가하고 다량의 데이터를 신속하게 처리할 필요성이 높아짐에 따라 대용량 데이터를 저장하고 처리하기 위한 플랫폼에 대한 요구가 높아지고 있다. 전통적으로 이와 같은 요구가 상존하였던 슈퍼 컴퓨팅 분야뿐만 아니라 범용 컴퓨팅 분야에서도 대량의 데이터 처리 필요성이 점점 요구되고 있다.

하나의 서버 차원에서 CPU, 메모리, 디스크 성능을 개선하기 위한 시도가 물리적인 한계에 직면함에 따라 다수 컴퓨터들을 네트워크로 결합하여 전체적인 성능 향상을 추구하는 흐름이 생기게 되었고, 이러한 과정을 통하여 분산 파일 시스템 기술이 대용량 스토리지 및 확장성(scalability)을 제공하는 수단으로 인정받고 있다.

분산 파일 시스템은 네트워크 기술을 이용하여 다수 컴퓨터들의 저장 장치들을 논리적으로 결합함으로써 사용자에게 통합된 저장 공간을 제공해 주는 파일 시스템이다. 분산 파일 시스템의 예로는 Lustre[1], Panasas[2], OASIS[3] 등이 있으며, 이러한 시스템들의 아키텍처는 크게 대칭형(symmetry) 구조와 비대칭형(asymmetry) 구조로 구분할 수 있다.

대칭형 구조에서는 모든 노드들이 클라이언트와 서버 기능을 동시에 보유하고 있는 반면, 노드들 사이의 통신 비용이 전체 클러스터의 확장성을 제약하는 것이 단점이다. 비대칭형 구조에서는 메타 데이터 서비스를 제공하는 메타 데이터 서버가 별도로 독립되어 있는 것이 특징이며, 메타 데이터와 데이터의 처리 경로가 독립됨으로써 클라이언트, 데이터 서버 등을 대규모로 운영하는 경우에 입출력 성능 및 확장성이 증가하는 장점이 있다. 현재 분산 파일 시스템의 주도적인 아키텍처는 (그림 1)과 같은 비대칭형 구조이다.

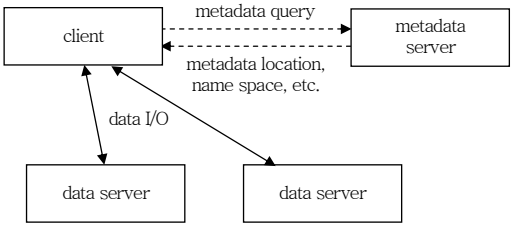
(그림 1)과 같은 구조에서는 클라이언트가 파일의 위치 및 사용 권한 등을 파악하기 위한 용도로 메타 데이터 서버에게 질의를 하며, 실제 데이터 입출력 작업은 메타 데이터 서버를 통하지 않고 클라이언트와 데이터 서버 사이에서 직접 처리될 수 있다. 이 방법은 대량의 클라이언트와 서버들을 요구하는 환경에서의 통합 성능을 현저하게 높일 수 있다. 그러나, 메타 데이터를 요청하는 클라이언트들의 수가 수십~수백 개 규모를 초과하여 수천~수만 개 이상으로 늘어나고, 필요한 스토리지 양이 페타 바이트 이상 규모인 환경을 고려하였을 때는 다수 클라이언트의 요청을 하나의 메타 데이터 서버가 정상적으로 처리할 수 있는 한계를 초과하게 된다. 따라서, 이러한 병목 현상 문제를 유연하게 해결할 수 있도록 아키텍처를 설계할 필요가 있다.

메타 데이터 서버의 병목 현상을 해결하기 위한 시도로 메타 데이터 서버들의 클러스터를 고려할 수 있다. 메타 데이터 서버 클러스터를 구성하여 효율적으로 운영하기 위해서는 하나의 메타 데이터 서버에게 과부하가 걸리지 않도록 클러스터를 구성하고 있는 메타 데이터 서버들 간에 부하를 균등하게 분산시키는 방법을 찾는 것이 가장 중요한 설계 포인트가 된다.

본 고에서는 메타 데이터 서버들의 클러스터를 운영하는 데 있어서 성능의 저하를 방지하고 효율적인 부하 분산(load balancing)을 유지하기 위한 다양한 시도들을 분석하고 고찰해 보고자 한다.

● 용 어 해 설 ●

메타 데이터: 데이터에 관한 데이터를 의미하는 것으로서, 그 역할은 데이터를 사용, 관리, 이해하는 데 편의성을 제공하는 것이다. 파일 시스템 환경에서 어떤 파일이 저장되는 예를 고려하면 그 파일이 저장된 위치, 그 파일의 크기, 생성 시간, 사용 권한 등이 그 파일에 대한 메타 데이터라고 할 수 있다.



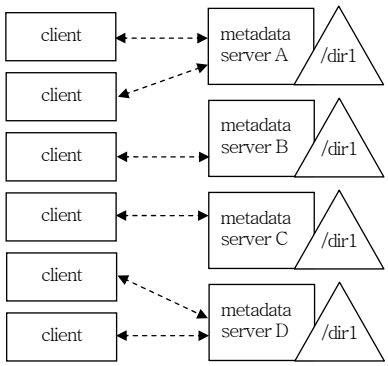
(그림 1) 비대칭형 분산 파일 시스템 구조

Ⅱ. 메타 데이터 서버 클러스터링 방법

1. 복제

하나의 메타 데이터 서버에 부하가 집중되는 것을 방지하기 위하여 동일한 메타 데이터 이미지를 가진 별도의 메타 데이터 서버를 유지함으로써 읽기 작업에 관한 부하를 경감시킬 수 있다. 예를 들어, 메타 데이터 서버 A가 /dir1에 대한 이름 공간 및 각종 권한을 관리하고 있을 경우에 수백 대의 클라이언트가 동시에 /dir1에 대한 정보를 획득하기 위하여 서버 A에 접속하게 되면 A가 처리할 수 있는 자원의 한계를 넘어설 수 있게 되어 정상적인 처리를 수행하지 못할 수도 있다. 이러한 경우에, (그림 2)와 같이 A와 동일한 이미지를 가진 메타 데이터 서버 B, C, D 등을 유지하고 있다면 클라이언트들의 요구가 A, B, C, D 등에 분산됨으로써 각각의 메타 데이터 서버는 주어진 요구량에 대해 정상적인 처리를 할 수 있는 확률이 높아지게 된다.

복제(replication) 방식에 의하여 메타 데이터 서버들의 클러스터를 구성하게 되면 읽기 작업에 대해서는 부하 분산이 이루어지는 반면, 쓰기 작업에 대해서는 더 많은 시간과 비용이 들게 된다[4]. 그 이유는 기록하고자 하는 메타 데이터가 메타 데이터 서버 A 뿐만 아니라 B, C, D 등에도 모두 동일하게 저장되어야만 해당 메타 데이터의 쓰기 트랜잭션이 종료하기 때문이다.



(그림 2) 동일한 메타 데이터가 복제된 구조

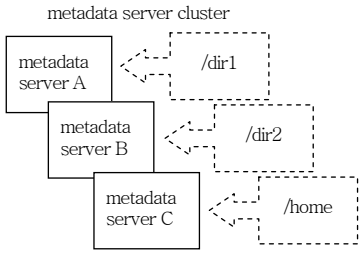
게다가 관리하고자 하는 메타 데이터 자체의 총 규모가 페타 바이트급 이상의 대용량 크기일 경우에는 각각의 메타 데이터 서버가 차지하는 막대한 스토리지 공간을 또다시 몇 개 이상으로 복제한다는 것은 공간적으로 상당한 부담이 될 수 있다. 따라서, 메타 데이터 자체가 대용량일 경우를 처리하기 위하여 각각의 메타 데이터 서버가 관리하는 스토리지 공간들이 해당 클러스터 내에서 공유될 수 있는 방법을 적용하거나, Ⅱ장 2절에서 소개할 정적 서브 트리 분할 방법의 문맥 안에서 핫스폿(hot spot)이 발생하는 메타 데이터만 별도로 타 메타 데이터 서버에 복제하는 방법 등도 고려해 볼 수 있다[5].

2. 정적 서브트리 분할

정적 서브트리 분할(static subtree partition)은 관리자가 이름 공간(name space)을 수동으로 적절히 분할하고, 각 공간에 대하여 전담 처리를 수행하는 메타 데이터 서버를 할당함으로써 메타 데이터 관리를 수행하는 방법이다[5],[6].

예를 들어, (그림 3)과 같이 /dir1 이하의 모든 서브 디렉토리, 파일 등에 관한 정보는 메타 데이터 서버 A가 관리하고, /dir2 이하에 대해서는 메타 데이터 서버 B가 관리하고, /home 이하는 서버 C가 관리하도록 관리자가 정적으로 메타 데이터 서버들의 클러스터를 구성할 수가 있다.

정적 서브트리 분할 방법은 디렉토리 계층 구조(directory hierarchy)를 유지하며, 참조 지역성(reference of locality)을 제공한다. 따라서, 검색하



(그림 3) 메타 데이터를 서브트리별로 분리하여 전담 메타 데이터 서버를 운영하는 구조

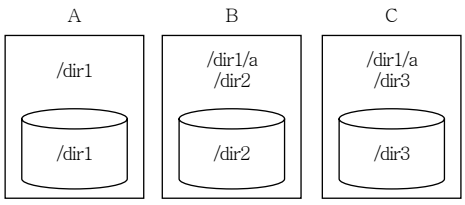
고자 하는 파일이 속한 전체 경로(path)를 순회(traversal)하면서 권한을 체크해야 할 필요가 있으나, 이에 필요한 비용은 동일 디렉토리 내의 파일들이 지속적으로 사용되는 접근 패턴에 의거하여 상쇄된다. 예를 들어, 클라이언트가 /dir1/subdir1/subsubdir1/data1과 /dir1/subdir1/subsubdir1/data2, /dir1/subdir1/subsubdir1/data3라는 파일들에 차례대로 접근하고자 할 경우에 dir1, subdir1, subsubdir1, data1, data2, data3에 관한 각각의 정보가 모두 동일한 메타 데이터 서버 A에 저장되어 있다면 클라이언트가 /dir1/subdir1/subsubdir1/data1의 정보를 처음으로 조회하기까지는 메타 데이터 서버 A를 여러 번 접속해야 하지만, 그렇게 하는 과정에서 동일한 디렉토리에 있는 다수의 파일들에 대한 정보를 미리 읽어 오는(prefetching) 운영 등을 통하여 data2 및 data3에 대한 접근 비용이 상쇄되도록 만들 수 있다[5].

정적 서브트리 분할 방법을 사용하는 시스템들의 예에는 NFS[7], AFS[8], Coda[9] 등이 있으며, 관리하고자 하는 메타 데이터의 총 크기가 적정 규모 이내일 경우에는 비교적 관리하기가 용이하다. 그러나, 이름 공간을 최초에 관리자가 어떻게 분할하는가에 따라 향후 성능이 좌우되게 되며, 본질적으로 부하 분산이 불균등하게 이루어진다.

만일, 하나의 파일 또는 디렉토리, 기타 특정 디렉토리의 서브트리에 부하가 몰리게 될 경우 심각한 병목 현상을 겪게 되며, 메타 데이터 자체가 대용량으로 확대되고 메타 데이터 정보를 요구하는 클라이언트들의 수가 대규모로 늘어나게 되면 특정 메타 데이터 서버에 부하가 집중될 확률이 높아지게 됨으로써 전체 성능이 저하된다. 또한, 클러스터를 구성하는 메타 데이터 서버 중 일부가 손상되면 그 서버가 관리하고 있는 이름 공간에 대한 질의가 실패하게 되어 신뢰성(reliability) 문제가 야기되게 된다.

정적 서브트리 분할 방법이 적용된 메타 데이터 클러스터에 복제 방법을 추가함으로써 성능 저하 및 신뢰성 부족을 회피하는 변형적인 모델도 시도해 볼 수 있다. 예를 들어, 메타 데이터 서버 클러스터가 A, B, C라는 3대의 서버들로 구성되어 있을 경우에 A', B', C'라는 3대의 메타 데이터 서버들을 추가하고, A', B', C'에 각각 A, B, C의 복제 이미지를 유지한다면 읽기 성능과 신뢰성을 높일 수 있다. 그러나, 쓰기 작업의 성능이 저하되고 메타 데이터 서버들 간의 일관성 유지 문제를 해결해야 하는 부담이 존재한다.

정적 서브트리 분할에 복제 방법을 보강한 또 다른 예는 (그림 4)에 표현되어 있다. (그림 4)에서는 /dir1, /dir2, /dir3를 메타 데이터 서버 A, B, C에 각각 정적으로 할당한 후 운영하다가 /dir1/a라는 파일에 관한 정보를 클라이언트 100대가 동시에 요구하게 되어 핫스폿이 발생하면, /dir1/a의 메타 데이터를 B, C 서버의 캐시에 복제해 놓고 각 클라이언트들이 최초 접속하는 메타 데이터 서버를 무작위로 선택하도록 함으로써 각 메타 데이터 서버는 /dir1/a에 대해 요구되는 총 부하의 약 1/3씩만 부담하면 된다. 그러나, 위와 같은 특정 경우를 고려하도록 복제 방법을 구현한다면 각 메타 데이터 서버들의 입장에서는 클러스터 내에서 부하가 집중되는 서버를 실시간으로 파악할 수 있어야 하며, 이를 위한 상호 간의 통신 수단 확보, 메타 데이터 서버들 간에 메타 데이터를 이동할 수 있는 메커니즘 구현, 클러스터 내의 일관성 유지라는 상당한 추가 부담을 지게 된다.



(그림 4) 정적 서브트리 분할과 복제를 결합한 예

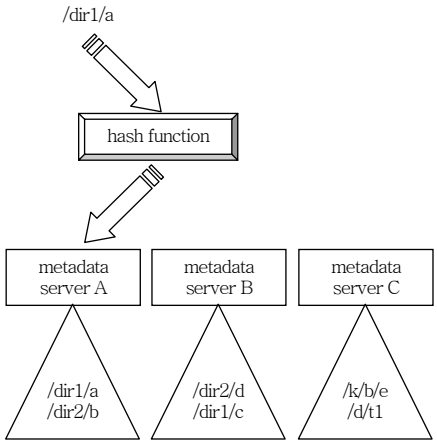
3. 해싱

정적 서브트리 분할 방법에서 가장 큰 문제는 사람이 이름 공간을 할당하기 때문에 메타 데이터 서버들 상호 간에 부하가 불균형하다는 점이다. 이러한 문제를 보완하기 위해 클라이언트는 정적으로 설정된 메타 데이터 서버를 접촉하는 대신, 해시 함수

(hash function)를 이용하여 메타 데이터 서버를 선택할 수 있다[5],[6]. 예를 들어, (그림 5)와 같이 어떤 클라이언트가 /dir1/a라는 파일에 대한 메타 데이터 정보를 얻기 위하여 모든 클라이언트들이 공유하는 해시 함수에 디렉토리 이름(dir1), 혹은 파일 이름(a), 혹은 전체 경로 이름(/dir1/a) 등을 입력값으로 주고 해시 함수가 반환하는 값을 이용하여 메타 데이터 서버를 선택할 수 있다.

해싱(hashing)에서는 클러스터를 구성하는 각 메타 데이터 서버들 상호 간에 균등하게 부하를 할당함으로써 평균적인 경우의 성능을 향상시킬 수 있으며, 이것이 해싱의 본질적인 특징이다. 그러나, 개별 파일에 대한 핫스폿 현상은 처리할 수가 없다. 즉, 다수 클라이언트들이 동시에 특정 파일을 접속하는 경우에는 결국 하나의 메타 데이터 서버에 모든 부하가 집중되는 것이기 때문에 성능의 저하를 막을 수 없게 된다.

메타 데이터의 실제 위치는 해시 함수에 의하여 결정되기 때문에 디렉토리 계층과는 아무런 연관성이 없으며 그 결과 참조 지역성이 보존되지 못한다[5]. 또한, 리눅스 ls 명령어 등과 같이 디렉토리 계층 구조의 문맥을 유지하기 위하여 또는 해당 파일의 경로명을 구성하는 각 디렉토리의 접근 권한(permission)을 확인하기 위하여 필연적으로 경로의 순환 작업이 필요하게 되어 해싱이 본래 의도했던 부하 분산의 장점은 상당부분 퇴색된다[6]. 예를 들어, /a/b/c/d/file에 대한 메타 데이터 정보를 얻고자 할 경우, 디렉토리 a에 대한 접근 권한을 확인하기 위하여 a에 관한 정보를 가지고 있는 메타 데이터 서버를 한 번은 접촉해야 하며, b에 대한 정보를 가지고 있는 메타 데이터 서버도 마찬가지의 이유로 접촉해야 한다. c와 d, file에 대한 정보를 가지고 있는 메타 데이터 서버도 이와 마찬가지이다.



(그림 5) 해싱을 이용하여 메타 데이터 서버를 결정하는 구조

해싱을 적용하게 되면 메타 데이터 서버들의 클러스터 구성이 변화되는 경우를 처리하기가 어렵다[6]. 만일, 클러스터에 새로운 메타 데이터 서버가 추가되거나, 기존의 메타 데이터 서버가 클러스터에서 삭제된다면 해시 함수의 출력 범위가 변경된다. 따라서, 해시 함수 자체를 수정해야 하며 클러스터 내의 균등한 부하 분산을 위하여 기존에 저장되어 있던 메타 데이터들의 상당량을 이동시켜야 할 필요가 있다. 이에 따라, 메타 데이터 서버들 상호 간에 대량의 메시지 전송 및 데이터의 이동이 발생하며 그에 따른 일관성 유지는 메타 데이터 서버 클러스터 관리에 있어서 상당한 부담이 된다.

Lustre[1], RAMA[10], Vesta[11] 등은 메타 데이터 정보를 획득하기 위하여 경로명 혹은 기타 식별자 정보를 활용하는 해싱을 채택한 시스템들이다.

해싱은 입력값을 어떻게 사용하는가에 따라 디렉토리 해싱과 파일 해싱 등으로 구분할 수 있다[6].

디렉토리 해싱은 해시 함수의 입력값으로 부분적인 디렉토리 정보를 주는 것이다. 예를 들어, 클라이언트가 /dir1/subdir1/file1에 대한 메타 데이터를 획득하기 위하여 해시 함수의 입력값으로 “dir1” 혹

● 용 어 해 설 ●

해시 함수: 테이블에 저장되어 있는 특정 레코드를 신속하게 탐색하기 위한 목적으로 사용되며, 어떤 키(key)를 입력으로 받아서 테이블 내의 원하는 레코드 위치를 찾기 위한 특정값을 산출하는 수식으로 구성된다. 해시 함수를 잘 정의해 두면 특정 레코드를 저장하거나 검색할 때 테이블 상의 모든 레코드들을 조회할 필요 없이 한 번의 해시 함수 호출만으로도 원하는 위치를 발견할 수 있다.

은 “subdir1” 혹은 “/dir1/subdir1”을 사용할 수 있다. 즉, 전체 경로 중에서 파일명인 file1을 제외한 나머지 디렉토리의 이름을 이용함으로써 특정 디렉토리 내에서의 참조 지역성을 유지하고자 하는 해싱 방법이다. 그러나, 이 방법은 디렉토리 내에서의 지역성은 보존 가능하지만 그 디렉토리 이하의 서브트리에 대해서는 지역성을 보존할 수는 없다. 또한, 해당 디렉토리 자체가 핫스폿이 되는 경우는 처리할 수 없는 문제가 존재한다.

파일 해싱은 해시 함수의 입력값으로 파일명 혹은 파일명을 포함한 완전한 경로명을 주는 것이다. 이러한 해싱 방법을 사용하면 메타 데이터 서버들 상호 간의 메타 데이터 분포가 평균적으로 균등하게 되는 장점이 있으나, 어떠한 지역성도 활용할 수 없게 된다. 또한, 경로명을 구성하는 디렉토리 등에 대한 접근 권한 정보를 얻기 위하여 필연적으로 경로를 순회해야만 하기 때문에 성능 저하를 유발하게 된다.

4. Lazy Hybrid

Lazy Hybrid (LH)[6],[12] 방법은 해싱과 정적 서브트리 분할 방법의 개념을 결합한 방법이다. LH에서는 특정 파일에 관한 메타 데이터를 획득하기 위하여 그 파일의 전체 경로명을 해시 함수의 입력값으로 준다. 따라서, 해싱의 장점인 메타 데이터 서버들 상호간의 균등한 부하 분산이 이루어지게끔 의도한다.

또한, 해싱에서의 이러한 장점을 상쇄시켰던 경로 순회 문제를 해결하기 위하여 접근 제어 리스트(ACL)를 유지한다. 그 결과, 특정 파일에 대한 권한 정보를 조회하기 위해서는 그 파일의 전체 경로명에 대한 해시 함수의 반환값에 의거하여 특정 메타 데이터 서버에 한번 접촉하면, 그 파일과 관련된 ACL을 직접 조회할 수 있기 때문에 그 파일이 속한 경로명에 포함된 모든 디렉토리 정보를 획득하기 위한 목적으로 그 디렉토리들에 대한 정보를 포함하고 있는 해당 메타 데이터 서버들을 순회할 필요가 없게 된다. 물론 ACL을 처음 생성할 때는 예외이다.

LH는 ls 명령어를 처리하기 위한 예에서와 같이 표준 디렉토리 개념과 문맥을 지원하기 위하여 계층적인 디렉토리 구조를 가정하고 운영된다. 그리고, 파일이나 디렉토리 이름, 또는 권한이 변경되는 경우와 메타 데이터 서버들의 클러스터 구성에 변화가 발생하는 경우를 효율적으로 처리하기 위하여 지연 수정(lazy update) 정책을 취한다.

본 장에서는 LH 방법을 구성하는 주요 개념들에 대하여 소개하고자 한다.

LH에서는 해시 함수의 결과값을 직접 사용하는 대신 메타 데이터 검색 테이블(MLT)의 인덱스로서 사용한다.

<표 1>에서 살펴볼 수 있는 것처럼 MLT는 모든 클라이언트들과 메타 데이터 서버들이 공유하는 전역 테이블로서 클라이언트들과 메타 데이터 서버들 상호 간의 간접적인(indirect) 관계 수준을 높인다. 따라서, 메타 데이터 서버가 클러스터에 추가되거나 삭제되는 변경 상황이 발생되면 MLT를 간단하게 수정함으로써 해시 함수의 수정 등 시스템 전체에 미치는 파급 효과를 줄인다.

앞에서 언급했듯이 해싱을 사용하더라도 찾고자 하는 파일이 포함된 전체 경로명의 각 구성 요소에 해당되는 권한 정보를 확인해야 할 필요가 있기 때문에 각 구성 요소에 대응하는 권한 정보를 가지고 있는 모든 메타 데이터 서버들을 순회하게 됨으로써 해싱의 장점인 균등한 부하 분산 효과가 퇴색되는 것이 일반적이다. LH에서는 이러한 디렉토리 순회 문제를 방지하기 위하여 ACL을 구성하는데 그 구성 방법은 다음과 같다.

모든 파일 및 디렉토리는 파일 권한(file permission)과 경로 권한(path permission)을 나타내는 2개의 ACL을 가진다. 파일 권한은 해당 파일이나 디

〈표 1〉 메타 데이터 검색 테이블의 예[6]

해시 함수 반환값 범위	메타 데이터 서버 식별자
0~1000	0
1000~2000	1
2000~3000	2

렉토리 자체가 가지고 있는 권한을 의미하며, 본질적으로 경로 권한은 부모 디렉토리의 경로 권한을 해당 요소의 파일 권한과 교집합한 것이다. 경로 권한은 파일을 처음 생성할 때 같이 만들어지며 /a/b/c/d라는 파일을 생성할 때 d의 권한이 rw-r--r--이면, d의 부모 디렉토리인 c의 경로 권한을 조회한 후, c의 경로 권한이 rwx-r-xr-x이면 d의 경로 권한은 최종적으로 rw-r--r--로 계산된다.

LH에서는 ACL을 사용함으로써 경로 순회 문제점을 상당부분 해결하였지만, 기존의 계층 구조를 그대로 유지할 경우에만 그 성능을 유지할 수 있다. 만일, 경로 중의 특정 디렉토리의 권한이나 이름이 변경되거나 삭제되고, 메타 데이터 서버가 클러스터에 추가되거나 삭제된다면 그 후속 처리를 위하여 대량의 메타 데이터를 이동하는 등의 높은 비용을 소모하게 된다.

이러한 후속 처리 비용을 줄이기 위하여 LH에서는 권한이나 이름이 변경된 경로 상의 구성 요소에 대하여 대량의 메타 데이터를 이동하는 등의 고비용 조치를 취하지 않고 일단은 해당 요소에 대하여 무효화(invalidation) 혹은 향후 메타 데이터가 이동될 것이라는 표시만 한다. 그리고, 그 요소가 추후에 처음 사용되는 시점에서 약간의 디렉토리 순회를 통하여 ACL을 수정하거나, 메타 데이터를 이동시킨다. 따라서, 평균적으로 보면 디렉토리 권한이나 이름 변경에 따른 초기 작업은 매우 신속하게 처리되며, 수정되어야 할 모든 메타 데이터 요소들은 향후에 각기 처음 사용되는 시점에서 약간의 비용 부담을 더 부담함으로써 메타 데이터 이동에 따른 부담을 감소시킨다.

이와 같은 정책에 따른 처리 예제는 다음과 같다. /a/b/c/d라는 경로를 가지는 파일 d가 있는 상황에서 a에 대한 메타 데이터 및 권한 정보는 메타 데이터 서버 A가 가지고 있고, b에 대해서는 메타 데이터 서버 B, c에 대해서는 메타 데이터 서버 C, d에 대해서는 메타 데이터 서버 D가 대응되도록 해시 함수가 결정하였다고 가정한다. 만일, 디렉토리 이름 c가 k로 변경되면 메타 데이터 서버 C는 다른 서버들인 A, B, D에게 디렉토리 c의 이름이 변경되었기 때문에 향후 메타 데이터가 이동될 것이라는 메시지를 보내고 이들 메시지는 A, B, D의 로그에 기록되고 초기 작업은 신속하게 종결된다.

향후 어떤 클라이언트가 d에 관한 정보를 얻기 위하여 /a/b/k/d라는 전체 경로명을 해시 함수의 입력으로 주었을 때 해시 함수가 그 결과값으로 메타 데이터 서버 A라는 잘못된 값을 반환하였다고 가정하자. 그러면, 클라이언트는 d의 정보를 얻기 위하여 메타 데이터 서버 A에게 접속하였으나 실제로 A에는 d의 정보가 없다는 사실을 확인하게 된다. 또한 메타 데이터 서버 A에서 디렉토리 c가 k로 변경되었다는 로그 레코드를 검색함으로써 찾고자 하는 정보가 실제로는 다른 메타 데이터 서버에 있다는 것을 알게 된다. 따라서, A는 과거에 d의 부모 디렉토리였던 c에 대한 정보를 얻기 위하여 메타 데이터 서버 C에게 접속하여 C가 관리하고 있는 내용을 확인함으로써 d는 메타 데이터 서버 D에 있다는 것을 알게 된다. 그러면, 다시 메타 데이터 서버 D에 접속한 후, D에 저장되어 있던 d의 정보를 메타 데이터 서버 A에게 이동시킨다. 이때부터, 해시함수가 /a/b/k/d라는 입력에 대하여 반환하는 값인 A가 정확한 의미를 지니게 된다.

5. 동적 서브트리 분할

동적 서브트리 분할(dynamic subtree partition) 방법[5]은 메타 데이터 서버 클러스터가 관리하는 이름 공간을 관리자가 개입하여 정적으로 최초 분할함으로써 각 이름 공간의 파티션에 대응하는 전용 메타 데이터 서버를 지정한다. 그러나, 부하가 특정 기준 이상으로 올라가게 되면 동적으로 파티션을 재배치하고 이에 따라 해당되는 메타 데이터를 이동시키도록 하는 방법이다.

동적 서브트리 분할 방법이 가지고 있는 기능은 다음과 같다.

첫째, 이름 공간을 서브트리별로 분할한다. 이를 통하여 참조 지역성 향상을 의도하며, LH 방법과 비

교하면 특정한 서브 트리에 대한 권한 변경 비용이 낮다.

둘째, 각 메타 데이터 서버들끼리는 협력적 캐싱(collaborative caching)을 유지한다. 이러한 기능을 통하여 특정 데이터를 다른 메타 데이터 서버에 복제할 수 있으며, 클라이언트들이 메타 데이터 서버들의 클러스터를 접근할 때 좀 더 유연한 대응을 할 수 있게 된다.

예를 들어, 읽기 작업의 부하 분산을 위해 특정 메타 데이터가 여러 개의 메타 데이터 서버들에게 복제되어 있다면, 그 메타 데이터를 필요로 하는 클라이언트가 자신에게 할당된 전용 메타 데이터 서버에게 요청을 보내는 대신, 무작위(random)로 메타 데이터 서버를 선택하도록 하는 것이 가능해진다. 즉, 클라이언트가 최초 접속하는 메타 데이터 서버가 항상 무작위로 결정된다면 네트워크상의 통신량은 일부 증가하겠지만 부하 분산은 더 용이해진다. 또한, 메타 데이터 서버는 자신이 관리하고 있지 않은 데이터를 요구하는 요청 작업에 대하여 그 요청을 다른 메타 데이터 서버로 포워딩(forwarding)시킬 수도 있다.

셋째, 부하 분산 및 핫스폿 제어를 위한 기능을 제공한다. 이를 위하여, 메타 데이터 서버들끼리는 주기적으로 하트비트(heartbeat) 메시지를 주고 받다가, 특정 서버가 관리하는 특정 서브 트리 혹은 파일에 과부하가 걸리게 되면 부하가 적은 서버 혹은 클러스터 내의 모든 서버들에게 해당 메타 데이터를 이동시킨다. 따라서, 어떤 클라이언트가 특정 메타 데이터에 대한 정보를 가지고 있는 메타 데이터 서버를 선택하기 위해서 처음에는 정적으로 배정된 메타 데이터 서버를 선택하다가, 그 메타 데이터를 요구하는 클라이언트 수가 특정 기준 이상을 초과하면 해당 메타 데이터가 메타 데이터 서버들의 클러스터 내로 고루 분산이 되고 해싱이 적용되어 최종 메타 데이터 서버를 선택할 수 있는 것이다.

이러한 방법을 사용할 때 정적 서브 트리 분할 방법을 버리고 해싱을 선택할 시점을 결정하는 것이 굉장히 어려운 문제이다. 그리고, 한 때 부하가 집중되었던 메타 데이터에 대한 요구가 점점 감소한다면 해싱을 버리고 기존의 정적 서브트리로 전환을 하는 시점을 파악하는 것과 이를 수행하기 위한 구체적인 방법을 결정하는 것도 풀기 어려운 문제이다.

참고로, 메타 데이터 서버는 동적인 부하 분산 및 핫스폿 제어를 위한 데이터를 수집하기 위하여 각 메타 데이터에 관하여 얼마나 요구가 집중되는지 결정하기 위한 모니터링 수단을 유지해야 한다. 예를 들어, 메타 데이터와 클라이언트의 요구 횟수를 유지하는 카운터를 연계할 수 있으며, 메타 데이터 서버가 클라이언트에게 전송하는 정보에 메타 데이터의 분포에 관한 내용을 피기백(piggyback)함으로써 클라이언트가 메타 데이터 서버들에 관한 좀더 정확한 정보를 파악하게끔 만들 수 있다.

넷째, 메타 데이터와 관련된 모든 변경 사항들은 신속하게 영구 저장 장치에 기록되어야 한다. 그러나, 성능의 향상을 위하여 각 메타 데이터 서버는 해당 변경 사항들을 영구 저장하기 전에 임시로 로그를 사용할 수 있다. 이러한 로그를 기록하기 위한 용도로 객체기반 저장장치(OSD)를 사용할 수 있으며, 그 결과 해당 로그들을 복제하는 등의 작업을 연계시킴으로써 메타 데이터의 신뢰성을 확보하기가 용이해진다.

6. 메타 데이터 관리를 위한 기타 전략

본 장에서는 지금까지 언급하였던 복제, 정적 서브트리 분할, 해싱, 동적 서브트리 분할 방법 이외에 메타 데이터 관리와 관련하여 특별히 언급해 볼 만한 사항들을 살펴보기로 한다.

첫째, 메타 데이터를 여러 가지 관점으로 분류한 후, 전체적인 처리 효율을 높이기 위한 다양한 조합을 고려해 볼 수 있다.

예를 들어, RAMA[10]에서는 메타 데이터 자체를 두 가지 성격으로 구분하여 저장하는 위치를 각기 다르게 취급하고 있다. 메타 데이터 중 실제 데이터를 찾기 위한 위치 정보를 가지고 있는 것을 위치적 메타 데이터(positional metadata)라고 하고, 그 외

에 파일의 크기, 소유자, 접근 시간, 수정 시간 등 일반적으로 메타 데이터로 취급되는 모든 것을 본질적 메타 데이터(intrinsic metadata)라고 구분한다[6].

RAMA는 해싱을 사용함으로써 위치적 메타 데이터를 획득하며, 본질적 메타 데이터는 데이터와 함께 저장하는데 실제 데이터가 저장되는 위치의 앞부분에 배치시킨다. 이러한 방식으로 2원화하여 메타 데이터를 관리함으로써 메타 데이터의 유지, 관리가 용이해진다. 그러나, ls 명령어와 같이 본질적 메타 데이터를 요구하는 디렉토리 관련 명령어 등을 처리하기 위해서는 메타 데이터를 획득하기까지 2회에 걸친 통신 비용이 소모되며 그 결과 시스템의 반응 시간이 느려진다.

Google 파일 시스템[13]에서는 메타 데이터를 다음과 같이 3가지 형태로 관리한다. 파일 및 청크(chunk) 이름 공간에 관련된 메타 데이터, 파일을 어떤 청크에 매핑할 것인지에 대한 정보, 그리고 각 청크의 복제본에 대한 위치 정보가 그에 해당된다. Google의 메타 데이터 서버는 앞의 2가지 정보, 즉 이름 공간과 매핑 정보는 영구적으로 저장하지만, 청크의 위치 정보는 영구적으로 저장하지 않는다. 그 대신, 청크의 위치 정보를 파악하기 위하여 메타 데이터 서버가 기동될 때 모든 청크 서버에게 청크 정보를 질의함으로써 그 내용을 파악하여 메모리에 보관한다.

Google에서 이러한 방식으로 구현한 이유는 메타 데이터 서버와 청크 서버 상호 간의 동기화를 맞출 필요가 없도록 단순하게 관리하기 위해서이다. 그 결과 청크 서버들의 추가 및 삭제, 이름 변경, 장애가 생겨서 재시작하는 경우 등 수많은 청크 서버들로 구성된 시스템에서 상시적으로 발생하는 오류로 인해 야기되는 다양한 경우에 대하여 메타 데이터 서버와의 동기화를 맞추기 위해 전체 시스템을 복잡하게 관리하고 구현할 필요성이 없어진다.

Hadoop의 DFS[14]도 Google 파일 시스템의 배치 방법을 따르고 있다.

이처럼 메타 데이터의 성격을 구분하여 별도 저장하거나, 시스템의 사용 방식에 따라 재구성 가능한 메타 데이터는 영구 저장하지 않고 필요할 때마다 질의하여 그 내용을 동적으로 구성함으로써 시스템의 복잡도를 적절하게 조정할 수 있게 된다.

둘째, 메타 데이터 관리와 관련하여 핫스폿 현상이 발생할 가능성이 높은 장소를 식별함으로써 핫스폿이 발생할 가능성을 사전에 예방하는 조치를 취할 수 있다. 예를 들어, 다음에 언급하는 장소들은 핫스폿이 발생할 가능성이 높은 곳이다.

디렉토리 계층 구조 상에서 / 혹은 /에 근접한 디렉토리는 어떤 파일을 접근하기 전에 다수의 클라이언트들이 거의 항상 요구할 확률이 높다. 그리고, 유닉스나 리눅스에 접속할 때 권한 관리를 위하여 항상 조회되는 /etc/passwd 등과 같이 특정한 시스템을 사용하기 위해 반드시 조회해야 하는 특정한 파일들 역시 핫스폿이 발생할 가능성이 높다. 여기에는 웹서버에서 클라이언트들의 로그인 과정을 처리하는 절차를 포함한 php 파일 등과 같은 것도 그 예가 된다. 또한, /tmp 디렉토리와 같이 동일한 디렉토리 내에서 많은 사용자들이 파일 생성, 삭제 작업을 일상적으로 수행하는 공용 디렉토리도 핫스폿이 발생할 가능성이 높은 곳이다.

이와 같은 파일이나 디렉토리는 메타 데이터 서버 클러스터 전체에 걸쳐서 고루 복제해 놓는 선행 작업을 수행함으로써 핫스폿 현상을 상당부분 통제할 수 있다.

셋째, 단일 디렉토리에 생성되는 파일들이 과도하게 많아지면 부하 분산 작업을 수행할 필요가 있다. 그러나, 이 작업의 주체를 메타 데이터 서버가 수행하는 것이 아니라 응용 프로그램 차원에서 수행할 수도 있다.

예를 들어, 특정 디렉토리에 생성되는 파일들이 어떤 기준을 초과하면 부하 분산을 위하여 응용 프로그램이 파일 생성 작업을 잠정적으로 연기함으로써 파일 시스템에 부담을 주지 않을 수도 있고, 부하가 적은 다른 디렉토리를 선택하여 파일을 생성할 수도 있는 것이다. 이러한 복잡한 작업을 응용 프로그램에게 위임하게 되면 파일 시스템 자체를 구현하기가 더욱 간단해진다.

넷째, 전통적인 파일 시스템들이 제공하던 기능들 예를 들어, POSIX 인터페이스 혹은 하나의 디렉토리마다 관리하는 자료 구조 등을 제공하지 않음으로써 구현할 때 필요한 많은 오버헤드를 줄일 수 있다. 예를 들어, Google 파일 시스템에서는 POSIX 인터페이스를 모두 제공하는 대신 파일의 생성 및 삭제 등 일부 인터페이스만 제공함으로써 전체 시스템의 구현 복잡도를 상당히 제거하였다. 또한, 전통적인 유닉스 시스템에서처럼 하나의 디렉토리마다 대응되는 자료 구조를 유지하는 대신, 파일의 전체 경로명을 입력으로 주면 대응되는 메타 데이터를 바로 찾아주는 매핑 함수를 제공함으로써 특정한 아키텍처에서의 효율성을 추구하였다[13].

다섯째, 메타 데이터 자체의 기록을 위하여 2계층 접근 방식을 취할 수 있다[5]. 즉, 메타 데이터를 기록할 필요가 있을 때 이를 영구 저장 장치에 바로 기록하는 것은 디스크 속도 차이 등의 문제 때문에 메타 데이터들의 잠금(lock) 문제, 동시성 수준, 메타 데이터의 신뢰성 등 다양한 측면에 부정적으로 작용할 수 있다. 따라서, 메타 데이터는 일차적으로 로그에 기록하고 그 기록 동작을 종료한다. 그리고 나서, 향후에 로그에 저장되는 총량이 특정 기준을 초과한다 등의 조건에 따라 영구 저장 장치에 실제로 기록을 시작한다. 이렇게 함으로써 기록 매체의 특성에 따른 시간 차이 등을 상쇄시킴으로써 효율적인 운영이 가능해진다. 물론, 메타 데이터의 신뢰성을 위하여 로그를 다른 메타 데이터 서버에 복제하는 것도 고려해야 할 사항이다.

여섯째, 효율적인 캐싱 관리를 위하여 각 메타 데이터 서버들이 독립적으로 캐싱을 유지하는 것이 좋을지 협력적 캐싱을 지원하는 것이 좋을지를 고려하여야 한다. 근본적으로 메타 데이터 서버들이 메타 데이터를 캐싱하는 것은 응답속도를 빠르게 하기 위하여 필수적으로 지원되어야 하는 기능이다. 그러나, 메타 데이터를 어떠한 방식으로 메타 데이터 서버들의 클러스터에 분할했는지에 따라 협력적 캐싱이 필요할 수도 있고 필요하지 않을 수도 있다. 예를 들어, 정적 서브트리 분할 방식으로 메타 데이터들을 분할해 놓은 경우 각 메타 데이터 서버들은 기본적으로 독립적 캐싱만 지원하면 된다. 그러나, 정적 서브트리 분할 방식일 경우에도 복제 개념이 추가된다면 협력적 캐싱이 일부 지원될 필요성이 존재한다.

Ⅲ. 메타 데이터 클러스터링 분석

메타 데이터 클러스터링을 크게 분류하면 서브트리 기반으로 분할하는 방법과 해시 함수를 제공하여 메타 데이터를 관리하는 방법으로 구분할 수 있다.

서브 트리 기반 분할 방법은 메타 데이터 서버들 상호간의 독립성이 유지되고 참조 지역성을 활용할 수 있는 점이 해싱에 비하여 우월하다. 그러나, 해싱과 비교하여 본질적으로 불균등한 부하 분산이 이루어지는 것이 단점이며, 이러한 문제를 극복하기 위하여 동적인 부하 분산이 이루어지도록 서브트리 기반 분할 방식을 개조할 수 있으나 이의 구현은 상당히 복잡하고 동적인 서브트리 분할을 실제 적용하기 위한 기준값을 파악하는 것이 난해한 문제이다.

해싱은 평균적인 응용 환경에서의 균등한 부하 분산을 유지할 수 있는 것이 최대의 장점이나 메타 데이터 서버들의 구성에 변화가 있거나 디렉토리 이름 등이 변경되는 경우 등을 처리하기 위한 비용이 많이 드는 것이 단점이다. 또한, 해싱 방법을 채용할 경우에는 파일 시스템 구조에 따르는 미묘한 정보를 이용하기 어렵다. 예를 들어, 서브 트리 분할 방법 하에서는 데이터 아카이빙(archiving) 작업보다 다른 작업에 우선 순위를 두는 정책을 설정함으로써 디렉토리를 다른 메타 데이터 서버로 이동하여 부하 분산을 시도할 수 있다. 이를 위하여, 계층 구조에 따른 성능 기준치를 정의하고 이를 반영하여 메타 데이터를 관리하는 것이 가능하다.

해싱을 기반으로 한 변형으로 LH 방법이 제시되어 있다. LH는 찾고자 하는 파일이 포함된 전체 경로를 구성하는 각 디렉토리들의 권한을 확인하기 위하여 해당 디렉토리를 순회해야 하는 필요성을 최소화시켰으나, 경로 상의 이름 혹은 권한이 변경되거

나 메타데이터 서버 클러스터의 구성이 변경되는 경우를 처리하는 비용이 크고 그 구현이 복잡하다.

한편, 서브트리 기반 혹은 해싱 기반의 메타 데이터 분할 방법에 복제 개념을 결합함으로써 읽기 용도에 자주 사용되는 메타 데이터를 모든 메타 데이터 서버에 복제하는 식으로 효율적인 부하 분산을 의도하는 방법도 존재한다[4].

지금까지 다수의 메타 데이터 서버들 중에서 특정 서버를 선택하기 위한 방법과 그에 따른 장단점들을 비교하였다. 한편, 이러한 관점 이외에 메타 데이터 서버 클러스터링을 위해 고려할 만한 다른 관점들도 존재한다.

근본적으로 메타 데이터 자체를 효율적으로 분류함으로써 관리의 효율성을 추구하는 전략이 존재한다. 즉, 메타 데이터 서버에 실제 저장해야 하는 메타 데이터와 저장할 필요가 없는 메타 데이터를 분류하는 것이다. 또한, 저장해서 관리하지는 않지만 다른 수단을 통하여 동적으로 구축 가능한 메타 데이터를 식별하는 작업이 필요하다. 이와 관련된 결정 사항은 시스템의 구조와 성능, 신뢰성 등에 절대적인 영향을 미치기 때문에 많은 시간을 투입하여 숙고해야 할 항목이다. 이 밖에도 부하 분산을 효율적으로 추구하기 위하여 예방적 차원에서 수행할 수 있는 작업, 응용 프로그램에게 위임 가능한 작업들의 분류, 이와 연관하여 구현의 복잡성을 감소시키기 위한 설계 방식도 일관성 있게 결정해야 할 사항이다.

마지막으로, 메타 데이터 서버에 장애가 생길 경우를 대비하여 로그를 도입한 2계층 구조의 저장 방식을 지원해야 하는지 여부를 결정하여야 한다. 이와 더불어 메타 데이터 서버 클러스터 내에서 이러한 로그 내용을 상호 간에 복제함으로써 고장 회복이 가능하도록 조치하는 방법 이외에 더 향상된 다른 방법이 존재하는지도 고민해야 할 문제이다.

Ⅳ. 결론

지금까지 메타 데이터 서버들의 클러스터링을 위해 제안된 다양한 방법들을 살펴보았다. 이러한 방법들은 소규모 분산 시스템이 아닌 클라이언트, 서버들이 대규모로 통합된 환경을 가정하고 있기 때문에 이러한 환경에서의 정상적인 메타 데이터 서비스를 위해서는 메타 데이터 서버들을 다수 유지하여 클러스터로 관리하는 것이 필연적이다.

메타 데이터 서버들의 클러스터는 효율적인 부하 분산, 높은 확장성, 손쉬운 관리 등이 가능하도록 전체 시스템이 제공하고자 하는 서비스 범위를 손상시키지 않는 한에서 아키텍처, 입출력 성능, 알고리듬 등을 되도록 간단하게 설계하는 것이 유리하다.

결론적으로, 메타 데이터 서버들을 효율적으로 관리하기 위해서는 부하의 분산, 핫스폿 제어, 효율적인 캐싱 관리, 메타 데이터 자체의 신뢰성 유지, 낮은 탐색 오버헤드, 참조 지역성 제공, 용이한 확장성, 메타 데이터 종류별로 다양한 처리 메커니즘 제공 등 여러 가지 측면을 고려하여 전체 아키텍처를 설계하고 그 운영 방식을 구현해야 할 것이다.

약 어 정 리

ACL	Access Control List
LH	Lazy Hybrid
MLT	Metadata Lookup Table
OSD	Object-based Storage Device
POSIX	Portable Operating System Interface for Computer Environments

참 고 문 헌

[1] Lustre, “Lustre: A Scalable High Performance File System,” Cluster File System, Inc., http://www.lustre.org/docs/whitepaper.pdf, Nov. 2002.

[2] D. Nagle, D. Serenyi, and A. Matthews, “The Panasas ActiveScale Storage Cluster-Delivering Scalable High Bandwidth Storage,” *In Proc. of the ACM/IEEE SC2004 Conf.*, Pittsburgh, PA, USA, Nov. 6-12, 2004.

[3] Y.K. Kim, H.Y. Kim, S.M. Lee, J. Kim, and M.J. Kim, “OASIS: Implementation of a Cluster File System

Using Object-based Storage Devices," *In Proc. of the Int'l Conf. on Computational Science and Its Applications*, Glasgow, UK, May 2006.

[4] Q. Huang, W. Zheng, and M. Shen, "TH-CluFS: An Open Platform Cluster File System," *In Proc. of the Fifth Int'l Conf. on Algorithms and Architectures for Parallel Proc.*, 2002.

[5] S.A. Weil, K.T. Pollack, S.A. Brandt, and E.L. Miller, "Dynamic Metadata Management for Petabyte-scale File Systems," *In Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, Nov. 2004.

[6] S.A. Brandt, L. Xue, E.L. Miller, and D.D.E. Long, "Efficient Metadata Management in Large Distributed File Systems," *In Proc. of IEEE/11th NASA Goddard Conf. on Mass Storage Systems and Technologies*, Apr. 2003, pp.290-298.

[7] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS Version 3: Design and Implementation," *In Proc. of the Summer 1994 USENIX Technical Conf.*, 1994, pp.137-151.

[8] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, Vol.29, No.3, Mar. 1986, pp.184-201.

[9] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, Vol.39, No.4, 1990, pp.447-459.

[10] E.L. Miller and R.H. Katz, "RAMA: An Easy-to-Use, High-performance Parallel File System," *Parallel Computing*, Vol.23, No.4, 1997, pp.419-446.

[11] P.F. Corbett and D.G. Feitelso, "The Vesta Parallel File System," *ACM Transactions on Computer Systems*, Vol.14, No.3, 1996, pp.225-264.

[12] K.T. Pollack and S.A. Brandt, "Efficient Access Control for Distributed Hierarchical File System," *In Proc. of the 22nd IEEE/13th NASA Goddard Conf. on Mass Storage Systems and Technologies*, Apr. 2005.

[13] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google File System," *In Proc. of the 19th ACM Symp. on Operating Systems Principles, Bolton Landing*, NY, Oct. 2003.

[14] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design," http://lucene.apache.org/hadoop/hdfs_design.html, 2005.