BeanFS: 대규모 이메일 서비스를 위한 분산 파일 시스템

(BeanFS: A Distributed File System for Large-scale E-mail Services)

정 욱 † (Wook Jung) 이 대 우 † (Daewoo Lee) 박 은 지 † (Eunji Pak) 이 영 재 † (Youngjae Lee)
김 상 훈 † (Sanghoon Kim) 김 진 수 †† (Jinsoo Kim) 김 태 웅 ††† (Taewoong Kim) 전 성 원 ††† (Sungwon Jun)

요 약 저가의 하드웨어를 이용하는 분산 파일 시스템은 대용량의 저장 장치를 경제적으로 제공해주는 해법으로 많은 인터넷 서비스 업체에 의해 주목받고 있다. 본 논문에서는 대규모 이메일 서비스를 위한 분산 파일 시스템인 BeanFS의 설계와 구현에 대해 소개한다. BeanFS는 다음과 같이 이메일 서비스에 최적화되었다. 첫째, 이메일 서비스에서 이용되는 작고 많은 파일을 효과적으로 처리하기 위해서, 볼륨 기반의 복제 기법을 도입하여 중앙 서버의 병목현상을 완화시킨다. 둘째, 이메일 메시지의 단순한 접근 패턴을 고려하여 일관성 유지 기법을 경량화시킨다. 셋째, 재복제시에 발생하는 오버헤드를 줄이기 위해 일시적인 장애를 영구적인 장애와 분리하여 대처한다.

키워드 : 분산 파일 시스템, 이메일 시스템

Abstract Distributed file systems running on a cluster of inexpensive commodity hardware are being recognized as an effective solution to support the explosive growth of storage demand in large-scale Internet service companies. This paper presents the design and implementation of BeanFS, a distributed file system for large-scale e-mail services. BeanFS is adapted to e-mail services as follows. First, the volume-based replication scheme alleviates the metadata management overhead of the central metadata server in dealing with a very large number of small files. Second, BeanFS employs a light-weighted consistency maintenance protocol tailored to simple access patterns of e-mail message. Third, transient and permanent failures are treated separately and recovering from transient failures is done quickly and has less overhead.

Key words : Distributed file system, e-mail system

· 이 논문은 2009년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원을 받아 수행된 연구임(No. R01-2007-000-11832-0)

† 비 회 원 : KAIST 전산학과
wjung@camars.kaist.ac.kr
dwlee@camars.kaist.ac.kr
pakej@camars.kaist.ac.kr
yjlee@camars.kaist.ac.kr
sanghoon@camars.kaist.ac.kr
†† 종신회원 : 성균관대학교 정보통신공학부 교수
jinsookim@skku.edu
††† 비 회 원 : (주)nhn 가상화플랫폼개발팀
taewoong.kim@nhn.com
sung-won.jun@nhn.com

논문접수 : 2008년 8월 7일
심사완료 : 2009년 5월 11일

정보과학회논문지: 시스템 및 이론 제36권 제4호(2009.8)

1. 서 론

인터넷 사용자들의 이메일 서비스 사용은 매년 꾸준히 증가하고 있고, 이에 따라 이메일 서비스를 위한 스토리지 요구량도 지속적으로 증가하고 있다. 미국의 시장조사기관인 IDC(International Data Corporation)는 전세계 이메일 계정의 수가 1998년 2억 5300만개에서 2006년 16억 개로 증가하였다고 추산하였고, 2010년에는 20억 개까지 증가할 것으로 예측하였다[1]. 이메일 서비스 업체들은 개인에게 제공하는 메일 용량을 경쟁적으로 키우고 있고 이메일 사용량과 첨부파일의 크기는 계속 증가하는 추세이기 때문에, 이메일 서비스를 위한 스토리지 요구량의 증가는 당분간 계속될 것으로 전망된다.

저가의 하드웨어를 이용하는 분산 파일 시스템은 대용량의 저장 장치를 경제적으로 제공해주는 해법으로 많은 인터넷 서비스 업체에 의해 주목받고 있다. 구글(Google)은 자체적으로 구글 파일 시스템(GFS: Google File System)[2]을 개발하여 자사의 서비스에 이용 중이며, 아마존(amazon.com)은 Amazon S3(Amazon Simple Storage Service)[3]를 개발하여 스토리지 서비스를 제공하고 있다. 야후(Yahoo) 또한 오픈 소스 분산 파일 시스템인 하둡 분산 파일 시스템(HDFS: Hadoop Distributed File System)[4]을 자사의 서비스에 이용한다고 알려져 있다. 이러한 시스템들은 다수의 서버에 데이터를 분산시킴으로써 확장성(scalability)을 확보하는 동시에, 같은 데이터를 물리적으로 다른 서버에 복제본 형태로 저장하고 복제본 사이에 일관성(consistency)을 유지하는 방법으로 가용성(availability)을 보장한다. 또한 기존의 범용 파일 시스템과는 달리 특정 응용 프로그램에 특화된 기능만을 제공함으로서 단순하면서도 응용 프로그램에 최적화된 성능을 제공한다.

본 논문에서는 대규모 이메일 서비스를 위한 분산 파일 시스템인 BeanFS의 설계와 구현에 대해 소개한다. BeanFS는 저가 하드웨어를 이용하지만, 기존의 분산 파일 시스템과는 다르게 다음과 같이 이메일 서비스에 최적화되었다.

첫 번째로 다수의 작은 파일을 효과적으로 관리한다. 하나의 이메일 메시지를 하나의 파일로 저장하면 이메일 메시지에 대한 연산을 모두 파일 연산으로 처리할 수 있기 때문에 시스템 설계를 단순화하는데 효과적이다. 하지만 이렇게 하면 이메일 서비스의 특성상 다수의 작은 파일들의 생성 및 삭제가 빈번해지게 된다. 예를 들어, 수백만명의 사용자가 이메일 서비스에 가입되어 있고 평균적으로 하루에 10개의 메일을 받는다고 하면, 하루에 수천만개의 새로운 파일이 생성되고 삭제되게 된다. GFS나 HDFS와 같은 분산 파일 시스템은 하나의 중앙 서버가 네임스페이스(namespace)를 관리하는 집중형(centralized) 구조이기 때문에 메타데이터 연산이 증가하게 되면 중앙 서버에서 병목 현상(bottleneck)이 나타나게 된다. BeanFS는 이와 같은 병목 현상을 피할 수 있도록 볼륨(volume) 기반 복제 기법을 사용한다.

두 번째로 이메일 메시지의 단순한 접근 특성을 이용해서 복제본 간의 일관성 유지 기법을 경량화시켰다. 기존의 분산 파일 시스템은 범용의 어플리케이션에서 사용하기 위해 주복제본 기법(primary replica technique)[2,5,6]이나 쿼럼 프로토콜(quorum protocol)[7] 등의 강력한 일관성 유지 기법(strong consistency maintenance mechanism)을 사용한다. 이와 같은 기법들은 파일 내용 변경이나 같은 파일에 대한 동시 쓰기 연산이 존재하는 상황에서도 일관성을 유지하기 위한 과정을 포함하고 있다. 하지만 하나의 이메일 메시지를 하나의 파일에 저장하면 이러한 상황이 발생하지 않기 때문에, BeanFS는 단순하지만 이메일 메시지의 접근 패턴에는 충분한 상태 매칭(state matching) 기법을 이용하여 복제본 간의 일관성을 유지하였다.

세 번째로 일시적 장애에 대해 효율적인 복구를 수행한다. 일시적 장애란 비교적 짧은 시간에 해결이 가능하고 장애 이전의 데이터를 이용할 수 있기 때문에, 데이터가 손상되지 않기 때문에 장애가 지속된 시간 동안 누락된 연산을 적용하면 손쉽게 복구 가능한 장애를 의미한다. GFS나 HDFS와 같은 분산 파일 시스템에서는 일시적 장애를 별도로 분리하지 않고 장애가 발생하면 서버가 저장하던 데이터를 다른 서버로 복제하는 재복제(re-replication) 방식을 취한다. 하지만 다수의 작은 파일들이 저장된 경우에는 작은 크기의 빈번한 디스크 I/O에 의해 재복제 성능이 크게 저하된다. 또한 복구 과정을 조정하는 중앙 서버의 부하가 증가할 뿐 아니라 대량의 네트워크 트래픽이 발생된다. BeanFS는 일시적 장애의 복구시 전체 데이터에 대한 재복제를 수행하는 대신 로깅(logging) 기반의 볼륨 동기화(volume synchronization)를 사용하여 복구 시간을 단축한다.

본 논문의 구성은 다음과 같다. 제 2장에서는 본 연구의 배경에 대해서 설명하고, 제 3장에서는 BeanFS의 전체적인 아키텍처에 대해 기술한다. 제 4장에서는 BeanFS에서 발생하는 장애 유형에 대한 설명을 하고, 제 5장에서 일시적 장애와 영구적 장애로부터 복구하는 방법에 대해서 소개한다. 제 6장에서는 실험을 통해 BeanFS의 성능을 보여주며, 제 7장에서 관련 연구를 소개한 후, 제 8장에서 결론을 맺는다.

2. 연구 배경

2.1 목표 시스템 개요

그림 1은 BeanFS의 목표 시스템인 웹 기반 이메일

시스템의 구조를 보여준다. 전통적인 웹 기반 이메일 시스템은 다수의 메일 서버와 웹메일 클라이언트, 하나 또는 다수의 데이터베이스 서버, 그리고 이메일 메시지를 저장하기 위한 대용량의 스토리지 시스템으로 구성되어 있다. 메일 서버는 발송, 수신되는 SMTP(Simple Mail Transfer Protocol) 요청을 처리하고, 웹메일 클라이언트는 사용자에 의해 생성되는 메일 읽기, 쓰기, 삭제 요청을 처리한다. 스토리지 시스템은 보낸 메시지와 받은 메시지를 저장하고, 인덱스 서버는 송수신되는 메일 메시지의 메타데이터를 관리한다.

기존에는 이메일 메시지를 저장하기 위해 NAS(Network-Attached Storage)나 SAN(Storage Area Network) 같은 범용의 스토리지를 사용하였다. 하지만 이들은 단위 저장공간 당 비용이 높기 때문에 폭발적으로 증가하는 스토리지 요구량을 만족시키기에는 매력적인 해법이 아니며, 이들이 제공하는 POSIX 호환성이나 강력한 일관성 유지 기법들은 단순한 접근 패턴을 지니고 있는 이메일 메시지에는 불필요하다.

BeanFS는 기존의 웹 기반 이메일 시스템에서 이메일 메시지를 저장하는 용도로 쓰이는 보완적인 스토리지 시스템이다(그림 1의 점선). 이메일 서비스에서 대부분의 저장공간은 이메일 메시지를 저장하는데 사용되므로, 단순한 접근 패턴을 가지고 있는 이메일 메시지에 초점을 맞춰 적은 비용으로 확장성, 고가용성, 고성능을 갖는 저장 장치를 개발하는 것이 본 연구의 목표이다. 구글 사는 저가 하드웨어를 이용해서 어플리케이션에 최적화 된 저장 시스템을 구축하는 것이 경제적인 해법이라는 사실을 자신들의 서비스를 통해 이미 보여주었다[2].

2.2 이메일 메시지 접근 특성

BeanFS의 스토리지 요구사항을 파악하기 위해서 먼저 이메일 메시지 접근 특성을 조사하였다. 메일 서버와 웹 메일 클라이언트는 SMTP 요청을 처리하는 중에 이메일 메시지를 스토리지에 순차적으로 쓴다. 하나의 이메일 메시지는 하나의 파일로 저장되고, 인덱스 데이터베이스 서버가 각 이메일 메시지에 대해 유일한 파일 이름을 할당해 주기 때문에 같은 파일에 대한 동시 쓰기 요구는 없다. 읽기나 삭제는 사용자나 자동 스팸(spam) 검색기에 의해서 요청되며, 스팸 메일의 영향으로 인해 파일의 생성과 삭제가 빈번하다.

실제 저장되는 메시지의 크기 분포를 파악하기 위해서 대형 메일 시스템의 메일 서버로부터 507,306,028개의 SMTP 세션 로그를 수집해서 분석하였다. 그 결과, 대부분 수십 KB 수준의 작은 메시지를 다루고 있다는 것을 확인 할 수 있었다. 전체 이메일 메시지 중 95.0%는 55KB 보다 작았으며, 98.6%는 100KB 보다 작았다.

3. BeanFS 설계

3.1 목표

제 2장에서 언급한 이메일 시스템의 스토리지 요구사항을 만족시키기 위한 설계 목표는 다음과 같다.

- BeanFS는 단위 스토리지 용량 당 비용을 최소화한다. 웹 기반 이메일 서비스의 스토리지 요구량은 계속해서 증가하는 추세이기 때문에 스토리지 시스템의 경제성은 중요한 이슈가 된다. BeanFS는 고가의 특수한 하드웨어 대신 다수의 저가 하드웨어 사양의 서버를 이용하는 기존의 분산 파일 시스템의 설계 원칙을 따름으로써 경제성을 확보한다.
- BeanFS는 다수의 작은 파일을 효율적으로 지원한다. 설계 단순화를 위해 하나의 이메일 메시지를 하나의 파일에 저장하면 대부분의 파일이 수십 KB 수준이기 때문에 저장되는 파일 개수가 매우 많아지게 된다. 기존의 분산 파일 시스템이 수백 MB 이상 크기를 가지는 적은 수의 파일을 저장하는 것을 목표로 했다면, BeanFS는 기존의 분산 파일 시스템 보다 수 백 배에

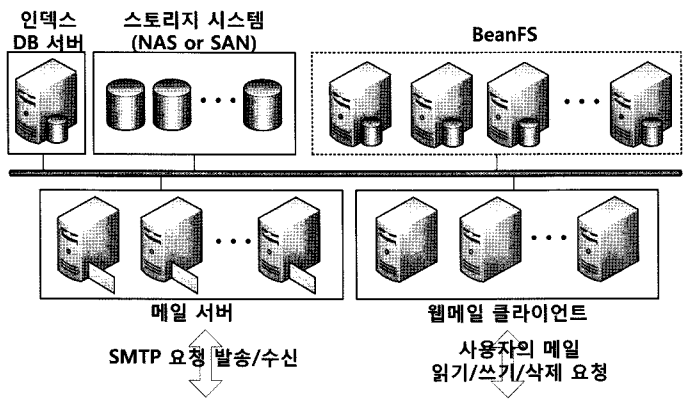


그림 1 웹기반의 이메일 시스템 구조

서 수 천 배 많은 파일을 효율적으로 저장 및 관리하는 것을 목표로 한다.

- BeanFS에서 사용하는 저가 하드웨어 서버의 장애 발생 빈도는 매우 높기 때문에[2] 구성 요소에 장애가 발생하는 환경에서도 높은 가용성을 유지한다. 따라서 BeanFS는 장애 발생 시 성능 저하 없이 스토리지 서비스를 제공하며, 장애 해소 후 빠른 시간 내에 정상 상태로 복구되도록 설계한다.
- 구성 요소의 수가 늘어나면서 관리 편의성 또한 중요한 이슈가 된다. 스토리지 사용량이 폭발적으로 증가하고 고장이 빈번한 환경에서 BeanFS는 스토리지 서버의 동적인 추가 및 제거 기능을 지원한다. 또한 많은 수의 서버 관리를 위해 서버 모니터링과 관리 도구, 자동화된 스토리지 사용량 균일화 기능을 통해 높은 관리 편의성을 제공한다.
- BeanFS는 POSIX 호환성을 제공할 필요가 없다. BeanFS는 범용의 응용 프로그램을 위한 시스템이 아니고 BeanFS를 이용하는 메일 서버나 웹메일 클라이언트는 서비스 업체에서 직접 개발하기 때문에, POSIX API 대신 BeanFS 전용의 API를 제공해도 무방하다.

3.2 전체 구조

BeanFS 클러스터는 그림 2와 같이 하나의 마스터와, 다수의 데이터 서버, 이에 접근하는 다수의 클라이언트로 구성된다. 마스터는 복제본 정보를 유지하고, 데이터 서버의 상태를 모니터링하며, 시스템 전체적인 작업을 주관한다. 실제 데이터는 데이터 서버에 저장되며, 각 데이터는 지정된 수(기본적으로 세 개)의 데이터 서버에 복제본 형태로 저장된다. 클라이언트는 어플리케이션의 파일 시스템 요청을 받아 처리하는 역할을 한다. 각 구성요소들 간의 통신은 분산 파일 시스템을 위해 개발된 FlexRPC[8]를 통해 이루어진다.

BeanFS의 전체 구조는 하나의 GFS[2]나 HDFS[4]처럼 하나의 중앙 서버를 가진 분산 파일 시스템과 흡사하다. 중요한 차이점은 BeanFS의 마스터는 파일 단위의 메타데이터가 아니라 볼륨 단위의 메타데이터만 관리하고, 파일 단위의 메타데이터는 실제 파일을 저장하는 데이터 서버의 로컬 파일 시스템이 관리한다는 점이다. 볼륨은 BeanFS의 복제 단위로써, 같은 디렉터리에 속한 파일의 집합을 의미한다. 따라서 마스터는 빈번하지 않은 디렉터리 메타데이터 연산만 처리하면 되며, 빈번한 파일 메타데이터 연산은 다수의 데이터 서버에서 나누어 처리되므로, 마스터에서의 메타데이터 연산의 오버헤드를 줄일 수 있다. 이러한 볼륨 기반의 복제 기법을 통해 BeanFS는 집중형 구조를 유지하면서 다수의 파일을 관리할 수 있다.

3.3 볼륨 관리

앞서 설명했듯이 볼륨은 BeanFS의 복제 단위로써 지정된 수의 데이터 서버에 복제본으로 저장된다. 각 볼륨은 64비트의 볼륨 식별자(vid: volume identifier)를 통해 구별되고, 파일 이름과 그 파일이 속한 볼륨의 볼륨 식별자를 통해 각 파일에 접근할 수 있다. 디렉터리 이름으로부터 볼륨 식별자를 알아내는 작업은 마스터에서 수행한다.

볼륨 단위의 복제 관리 기법은 AFS[9]와 Coda[10] 시스템에서 이미 사용된 개념이지만, BeanFS는 이에 비해 좀 더 작은 단위로 관리한다는 차이가 있다. AFS와 Coda의 볼륨은 특정 디렉터리의 서브트리(subtree)에 속하는 모든 파일을 포함하는 반면, BeanFS는 특정 파일이 속한 디렉터리를 기준으로 볼륨을 나누기 때문에 하나의 디렉터리가 하나의 볼륨에 대응되며, 특정 디

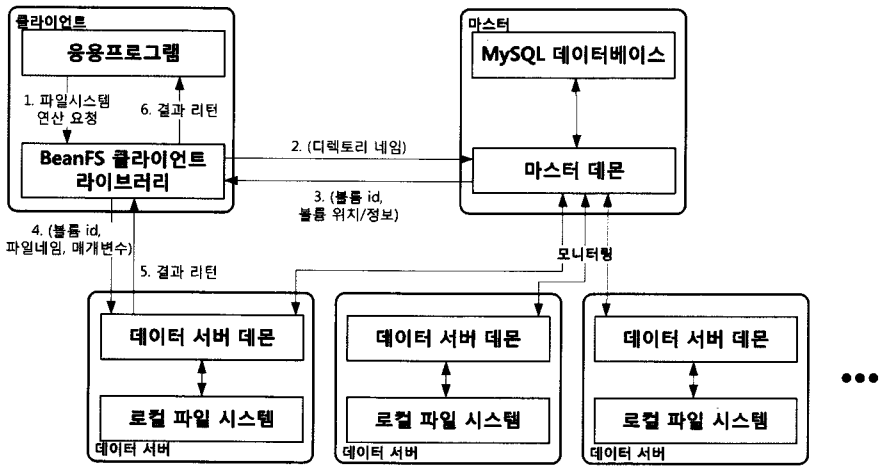


그림 2 BeanFS 전체 구조

렉터리에 속한 모든 파일들은 같은 볼륨에 속하게 된다. 웹 기반 이메일 시스템에서는 사용자가 네임스페이스를 관리하지 않기 때문에 디렉터리를 복제 단위로 사용해도 문제가 되지 않는다. 즉, 이메일 계정마다 하나의 디렉터리를 할당하고 수신된 모든 이메일 메시지를 해당 디렉터리에 저장하도록 한다.

3.4 BeanFS 구성요소

3.4.1 마스터

마스터는 볼륨 정보, 접근 제어 관리 등의 파일 시스템 전체 정보를 관리한다. 또한 주기적으로 현재 데이터 서버들의 CPU, 네트워크, 디스크 사용량을 모니터링 하며, 복구나 부하 재분배 등의 시스템 전체적인 작업을 관리한다.

하나의 마스터를 이용한 집중형 구조는 시스템 설계 및 메타데이터 관리를 쉽게 하지만, 장애에 취약하고 병목 현상이 발생하기 쉽다는 단점을 가지고 있다. 이를 위해 BeanFS의 마스터는 두 개의 대기 서버(stand-by server)를 가지고 있으며, 장애가 발생할 경우 자동으로 대기 서버가 마스터의 역할을 대신한다(fail-over). (4.2절 참고)

마스터가 시작할 때 데이터 서버로부터 마스터의 정보를 재구성하는 GFS나 HDFS와는 달리, BeanFS는 마스터가 관리하는 모든 데이터를 관계형 데이터베이스인 MySQL에 저장하도록 설계하였다. 이러한 방식은 마스터의 정보 업데이트 비용을 약간 증가시키지만, 데이터의 가공과 대기 서버들의 데이터베이스 복제를 편리하게 한다. 볼륨 정보는 크기가 작고 적은 빈도로 업데이트 되기 때문에, 이를 데이터베이스에 저장하는 오버헤드는 크지 않다. 일례로, 500만개 볼륨에 해당하는 MySQL 데이터베이스 용량은 인덱스 구조를 포함해서 2.2GB 정도 밖에 되지 않는다.

3.4.2 데이터 서버

데이터 서버는 볼륨 및 파일을 저장한다. 각 볼륨은 로컬 파일 시스템의 디렉터리로 만들어지고, 볼륨에 속한 파일은 해당 디렉터리에 저장된다. 따라서 빈번하게 접근되는 볼륨 및 파일은 로컬 파일 시스템의 버퍼 캐시에 저장되므로, 캐싱의 효과를 볼 수 있다. 또한 데이터 서버는 장애 발생 시 마스터의 제어를 받아 볼륨 동기화와 재복제를 수행한다. (5.2, 5.3절 참고)

3.4.3 클라이언트

클라이언트 모듈은 파일 시스템 요청을 어플리케이션으로부터 받아 처리하는 역할을 한다. 클라이언트 모듈은 라이브러리 형태로 제공되며, 어플리케이션은 BeanFS 전용의 API를 통해 BeanFS에 저장된 파일에 접근할 수 있다. 임의의 파일 시스템 요청이 클라이언트 모듈에 도착하면, 클라이언트는 먼저 마스터로부터 해당 볼륨의 볼륨 식별자와 담당하는 데이터 서버의 위치를 알아낸다. 읽기 연산의 경우에는 볼륨을 저장하고 있는 데이터 서버 중에 하나에 읽기 요청을 보내고, 쓰기 및 삭제와 같은 업데이트 연산의 경우에는 해당 볼륨을 저장하고 있는 모든 데이터 서버에 업데이트 요청을 보낸다. 또한 클라이언트에서는 일정 시간 동안 볼륨 정보를 캐싱하여 마스터로 가는 요청의 수를 줄인다.

3.4.4 RPC 계층

BeanFS를 개발하는 도중, 기존의 RPC 계층이 분산 파일 시스템이 필요한 많은 기능들이 부족하다는 것을 발견하였고, 이것은 분산 파일 시스템을 위한 RPC 계층인 FlexRPC[8]를 개발하는 계기가 되었다. FlexRPC는 클라이언트와 서버 양측에 완벽한 멀티스레드 환경을 제공하고, 다양한 호출 패턴을 지원한다. 또한 UDP와 TCP 프로토콜을 모두 지원하며 UDP 환경에서도 응답 캐시(response cache)를 통해 같은 호출이 서버에서 중복하여 실행되지 않도록(at-most-once semantics) 보장한다. 추가적으로 소켓 연결의 오버헤드를 줄이기 위해서 내부적으로 연결을 캐싱하고 CRC32를 이용해 무결성을 점검한다. 이러한 기능들은 BeanFS의 개발 복잡도와 비용을 줄이는데 크게 기여한다.

4. 장애 관리

구성 요소 일부에 장애가 발생하는 경우에도 고가용성을 유지하고 복제본 간의 일관성을 유지하는 동시에 장애 발생시 성능 저하를 최소화하는 것이 BeanFS의 장애 관리 목표이다. BeanFS는 장애가 발생한 구성 요소에 따라 다른 방식으로 장애에 대처한다.

4.1 클라이언트 장애

클라이언트는 동일한 복제본을 저장한 데이터 서버들과 직접적으로 통신하여 복제본을 업데이트 하기 때문에, 클라이언트 장애는 복제본 상태의 일관성을 손상시킬 수 있다. 이를 피하기 위해 분산 커미트 프로토콜(distributed commit protocol)[11]와 같은 기법을 사용하면 장애가 없는 상황에서의 파일 시스템 요청의 응답 시간까지 지연된다.

BeanFS는 목표 이메일 시스템의 특성을 고려하여 클라이언트 장애에 즉시 대응하지 않는다. 웹 메일 클라이언트에서 업데이트를 실패하면 다른 메일 서버나 다른 웹 메일 클라이언트가 실패한 세션을 다른 파일 이름으로 재시도하기 때문에, 실패한 업데이트에 대한 일관성이 요구되지 않기 때문이다. 클라이언트 장애로 인해 손상된 일관성은 백그라운드 작업을 통한 일관성 검사를 통해 차후에 해결된다.

4.2 마스터 장애

마스터는 BeanFS에서 가장 중요한 구성요소이기 때

문에 상대적으로 고가의 신뢰성 높은 하드웨어를 이용한다. 이러한 비용은 수많은 데이터 서버의 하드웨어 비용과 비교하면 무시할 만한 수준이다. 장애에 취약한 단점을 극복하기 위해 장애 시에는 자동으로 대기 서버가 마스터의 역할을 대신 하도록 설계하였다. 두 개의 대기 서버가 마스터에 데이지 체인(daisy chain) 형태로 연결되어 마스터의 데이터베이스를 실시간으로 복제한다. 마스터 혹은 첫 번째 대기 서버가 장애에 빠질 경우 체인의 뒤에 있는 서버가 장애가 난 서버의 역할을 대신하게 된다.

두 대기 서버는 체인의 앞의 있는 서버의 데이터베이스를 MySQL replication[12] 기능을 이용해서 복제한다. MySQL replication은 실제로 업데이트가 비동기적(asynchronous)으로 복제되기 때문에 체인의 앞에 있는 서버에서의 업데이트 성능은 빨라지지만, 장애 발생시 최신 업데이트가 체인의 뒤에 있는 서버에게 복제되지 못하는 경우가 발생할 수 있다. 이러한 문제를 해결하기 위해서 마스터와 두 대기 서버는 외부 저장장치를 공유해서 MySQL 로그를 저장한다. 장애가 발생하여 역할을 넘겨 받은 대기 서버는 로그를 통해서 복제하지 못한 업데이트를 반영하고 마스터 서비스를 넘겨받는다. 로그 저장을 위한 저장 공간은 수백 MB 정도면 충분하고, 외부 저장장치의 신뢰성은 RAID[13]를 통해 확보한다.

마스터가 모든 데이터를 데이터베이스를 통해서 저장하기 때문에, 대기 서버가 마스터의 역할을 넘겨 받는 작업은 RARP 프로토콜을 통해 ARP 테이블을 수정하는 것으로 완료된다. 따라서 클라이언트와 데이터 서버들은 마스터로 보내는 RPC 요청을 재시도 하기만 하면, 새로운 마스터가 요청을 받게 된다. 장애가 발생했던 서버는 복구가 완료되면 체인의 맨 끝에 붙어서 새로운 대기 서버가 된다.

4.3 데이터 서버 장애

BeanFS는 데이터 서버의 장애를 일시적 장애와 영구적 장애로 나누어 대처한다. 일시적인 네트워크 장애, 일시적인 하드웨어 오작동, 머신 재시작 등이 일시적 장애로 분류되며, 장애가 발생한 기간 동안 업데이트를 받지 못하기 때문에 장애가 발생했던 데이터 서버에 저장된 복제본의 일관성을 손상시킬 수 있다. 하드웨어 고장, 운영체제 이상 등 장애가 발생했던 서버의 볼륨 데이터를 사용할 수 없는 경우는 영구적 장애로 분류되며, 가용한 복제본의 수를 감소시키기 때문에 BeanFS의 가용성을 떨어뜨린다.

데이터 서버의 장애는 마스터의 주기적인 모니터링과 RPC 요청의 응답 여부를 통해 확인된다. 일반적으로 관리자의 개입이 없으면 일시적 장애와 영구적 장애를 판별하기가 어렵기 때문에, BeanFS는 모든 장애를 처음에는 일시적이라고 간주하였다가 일정 시간(30분)이지날 때까지 장애가 해소되지 않으면 영구적이라고 간주한다. 디스크 고장과 같이 영구적 장애가 분명한 경우에는 일정 시간이 지나기 전에 바로 영구적 장애로 간주한다. 데이터 서버 장애에 대한 복구 과정인 볼륨 동기화와 볼륨 재복제는 각각 5.2와 5.3에서 살펴보겠다.

5. 데이터 서버 복구

5.1 일관성 유지 프로토콜

복제본 중 일부는 장애 때문에 접근이 불가능할 수 있으며, 또 다른 일부는 복구가 진행 중이기 때문에 볼륨 데이터의 일관성이 보장되지 못하는 경우가 있다. 따라서 시스템 상에서 이러한 접근 불가능하거나 일관성이 보장되지 못한 복제본에 대한 접근을 제어해야 한다. 이를 위해 BeanFS에서는 복제본의 상태와 볼륨의 상태를 표 1처럼 정의하였다. 일반 복제본의 상태는 N으로 정의 된다. 데이터 서버에 장애가 발생하는 순간 그 서버에 속한 모든 복제본이 F 상태로 바뀌게 된다. 일시적 장애였다면 장애 해소 후 일관성 유지를 위한 볼륨 동기화가 시작되며, 볼륨 동기화 중인 복제본들은 R 상태로 바뀌게 된다. 영구적 장애였다면 가용성 보장을 위한 볼륨 재복제가 시작되며, 볼륨 재복제 중인 복제본들은 M 상태로 바뀌게 된다. 볼륨 동기화 및 볼륨 재복제를 마친 복제본들은 다시 N 상태로 돌아가게 된다. 그림 3은 복제본에 대한 상태 전이도를 보여준다.

BeanFS는 각 복제본에 대해 MO(Missed Operations)-로그와 DO(Delayed Operations)-로그를 유지한다. MO-로그는 F 상태의 복제본이 장애 기간 동안 받지 못했던 업데이트를 기록한 로그로, 일시적 장애였다면 해소 후 복제본에 재연되고 영구적 장애였다면 차후 제거된다. MO-로그는 해당 볼륨의 모든 N 상태 복제본에 중복해서 기록되며, 모든 업데이트 연산은 모든 N 복제본에 전달되므로 추가적인 통신이 발생되지 않는다. DO-로그는 R이나 M 상태의 복제본을 위한 로그이다.

표 1 복제본 및 볼륨 상태

복제본 상태	설명
State(v,d) = N	데이터 서버 d에 있는 볼륨 v에 속한 모든 파일의 상태가 정상
State(v,d) = R	데이터 서버 d가 볼륨 v의 파일을 복구 중에 있음
State(v,d) = F	데이터 서버 d에 있는 볼륨 v에 속한 파일의 접근이 불가능
State(v,d) = M	볼륨 v에 속한 파일들이 데이터 서버 d로 재복제 혹은 이동 중
볼륨 상태	
State(v) = <state(v,d_1), state(v,d_2), state(v,d_3)> d_1, d_2, d_3는 볼륨 v를 저장하고 있는 데이터 서버	

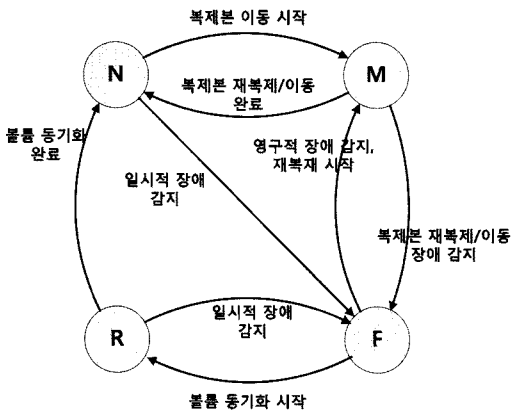


그림 3 복제본 상태 전이도

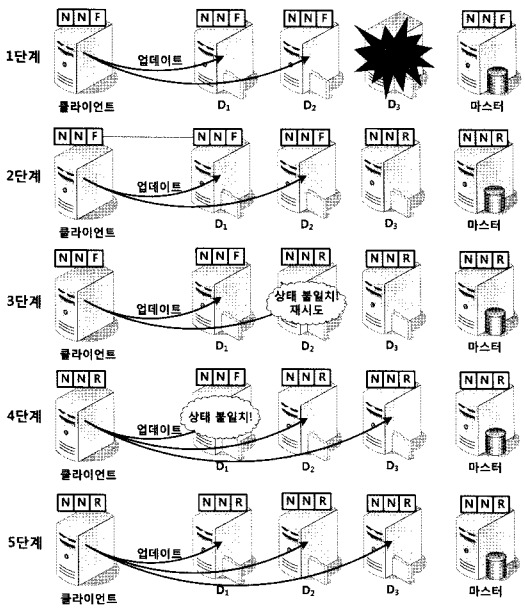


그림 4 상태 매칭 예제

R이나 M 상태의 복제본에 대한 업데이트는 볼륨 동기화나 볼륨 재복제 이후에만 수행 가능하므로, DO-로그에 기록해 두었다가 완료된 후에 반영한다.

볼륨 상태(볼륨에 속한 복제본들의 상태)는 클라이언트나 데이터 서버의 행동을 결정하는 중요한 정보이다. 클라이언트는 읽기 연산을 N 상태 복제본 중 하나에 보내야 하고, 업데이트 연산을 F 상태의 복제본을 제외한 모든 복제본에 보내야 한다. 업데이트를 받은 데이터 서버는 해당 볼륨에 F 상태의 복제본이 포함되어 있는지 보고 업데이트를 MO-로그에 기록할지를 결정해야 하며, 자신이 가지고 있는 복제본이 R 혹은 M 상태라면 업데이트를 DO-로그에 기록해야 한다. 따라서 클라이언트나 데이터 서버는 가장 최신의 볼륨 상태를 알아야만 한다. 하지만 데이터 서버가 자신이 저장하고 있는 복제본의 상태를 알아내는 것 이외에는 이를 보장하는 것이 사실상 불가능하다. 다른 데이터 서버에 저장된 복제본 상태를 알아낸다 하더라도 연산을 시도하는 동안 복제본 상태가 다시 바뀔 수 있기 때문이다.

BeanFS는 상태 매칭(state matching) 기법을 이용해서 이러한 문제를 해결한다. 상태 매칭 기법이란 파일 시스템 요청이 있을 때, 파일 시스템 요청에 참여한 클라이언트와 모든 데이터 서버들이 해당 볼륨 상태를 동일하게 알고 있는 경우에만 요청 처리를 허용하는 기법이다. 데이터 서버들은 자신이 저장하고 있는 복제본의 상태가 바뀔 때마다 마스터에게 알리고, 클라이언트는 마스터로부터 볼륨 상태를 얻어와서 파일 시스템 요청을 시작한다. 클라이언트로부터 요청을 받은 각 데이터 서버는 클라이언트가 보낸 볼륨 상태와 자신이 캐싱하고 있는 볼륨 상태가 같은 경우에만 요청을 처리한다. 만약 다르거나 캐싱되어 있지 않은 경우에는 마스터로부터 볼륨 상태를 얻어오고, 그것이 클라이언트가 보낸 볼륨 상태와 같은 경우에 요청을 처리한다. 여전히 다르다면 연산을 거부하여 클라이언트로 하여금 마스터로부터 볼륨 상태를 다시 얻어온 후 연산을 재시도하게 한다. 만일, 마스터 서버가 잘못된 볼륨 상태를 가지고 있다면, 파일 업데이트 연산은 클라이언트와 데이터 서버 간에 볼륨 상태가 일치되는 경우에만 수행되기 때문에 손실되는 파일 업데이트는 없으며, 마스터의 잘못된 볼륨 상태는 위 과정의 반복 속에서 클라이언트에 의해서 감지되거나 데이터 서버 모니터링을 통해 감지되어 올바르게 수정된다. 또한, 데이터 서버는 자신이 저장하고 있는 복제본 상태를 항상 확실하게 알 수 있기 때문에, 연산 중간에 볼륨 상태가 바뀐다 하더라도 위 과정을 반복하면 클라이언트와 데이터 서버들은 해당 볼륨 상태를 동일하게 알 수 있게 된다.

그림 4는 D_1, D_2, D_3에 복제되어 있는 볼륨에 대한 연산이 진행될 때, 상태 매칭 기법이 어떻게 진행되는지를 나타낸다. 1단계는 D_3이 장애에 빠져있고, 마스터와 클라이언트, 그리고 D_1, D_2 모두 최신의 볼륨 상태(N, N, F)를 캐싱하고 있는 상황을 나타낸다. D_3의 장애가 해결되는 2단계에서 D_3은 자신이 캐싱하고 있는 볼륨 상태를 (N, N, R)로 바꾸고 이를 마스터에게 알린다. 이 때, D_1, D_2는 볼륨 상태 변화를 인식하지 못하기 때문에, 2단계에서 요청 받은 클라이언트의 업데이트를 여전히 MO-로그에 기록한다. 3단계에서 D_3은 볼륨 동기화를 위해 D_2에게 MO-로그를 요청하며, 이 때 D_2는 D_3이 저장하고 있는 복제본의 상태 전이(F->R)를 인지하게 되어 이후 MO-로그 기록을 중지한다. 그리고 업

데이트를 받으면 상태 매칭 기법에 의해 이를 거부한다. 4단계에서 클라이언트는 거부된 연산을 돌려받았기 때문에, 마스터로부터 최신 볼륨 상태인 (N, N, R)을 알아내서 업데이트를 재시도한다. 이 때, D_1은 상태 매칭 기법에 의해 최신 볼륨 상태를 알게 된다. 결국에는 5단계처럼 클라이언트와 D_1, D_2, D_3 모두 정확한 볼륨 상태를 알게 되어서 연산을 진행하게 된다.

5.2 일시적 장애 복구

데이터 서버 D_{TF}에 발생한 일시적 장애가 해결되었을 경우, BeanFS는 D_{TF}이 저장하는 모든 복제본에 대해 볼륨 동기화를 시작하고, D_{TF}는 저장된 모든 복제본에 대해 다음 과정을 반복한다. D_{TF}는 먼저 복제본 상태 전이(F->R)를 마스터에게 알린 후, 해당 볼륨에 속한 N 상태의 복제본을 가지고 있는 데이터 서버 중 하나인 D_N에게 MO-로그를 요청한다. 이후 상태 매칭 기법에 의해 클라이언트는 업데이트를 D_{TF}에게도 보내게 된다. 다음으로 D_{TF}는 MO-로그를 재연하며, 이 과정에서 요청 받은 업데이트는 DO-로그에 기록된다. MO-로그 재연을 마친 후 D_{TF}는 DO-로그 재연을 하고, 마지막으로 복제본 상태 전이(R->N)를 마스터에게 알린다.

볼륨 동기화 중의 빈번한 복제본 상태 전이는 마스터에서 엄청난 양의 데이터베이스 트랜잭션을 유발하여, 볼륨 동기화 시간을 증가시키고 볼륨 동기화 중 마스터에 병목 현상을 야기한다. 이를 해결하기 위해 복제본 상태와 데이터 서버 상태가 동일하다고 간주하여 마스터에서의 데이터베이스 트랜잭션을 대폭 감소시켰다. 그리고 클라이언트나 데이터 서버가 상태 매칭 기법을 진행할 때 자신이 저장하고 있지 않은 복제본들의 상태 대신 데이터 서버 상태를 이용하도록 하였다. 데이터 서버는 N, R, F의 세 가지 상태로 표현되며, 저장된 모든 복제본들이 각각 모든 연산 가능, 업데이트만 가능, 모든 연산 불가능 상태임을 나타낸다. 5.1에서 설명한 복제본 상태 전이 과정과 데이터 서버 상태의 관계를 살펴보면, N 상태의 데이터 서버는 N, M 상태의 복제본만을, R 상태의 데이터 서버는 N, R 상태의 복제본만을, 그리고 F 상태의 데이터 서버는 F 상태의 복제본만을 가질 수 있음을 알 수 있다. 따라서 복제본 상태와 데이터 서버 상태가 동일하다고 간주할 수 없는 경우가 (복제본N, 데이터서버R)과 (복제본M, 데이터서버N)이다. (복제본N, 데이터서버R)의 경우는 문제가 되지 않는다. (복제본M, 데이터서버N)의 경우는 M 상태의 복제본이 읽기 요청을 받을 수 있다는 문제가 생기나, 단순히 이를 거부하여 클라이언트가 다른 데이터 서버에게 읽기 요청을 재시도하는 것으로 해결할 수 있다.

데이터 서버에 저장되는 볼륨의 개수가 늘어나면 모든 볼륨에 대해서 데이터 서버는 원격의 데이터 서버에 MO-로그가 존재하는지 확인해야 하는 오버헤드가 증가한다. 이러한 문제는 볼륨 그룹(volume group) 개념을 도입해서 해결할 수 있다. 같은 볼륨 그룹에 속한 볼륨은 같은 데이터 서버에 위치하고 볼륨 상태와 MO-로그를 공유하게 된다. 볼륨 그룹의 수를 볼륨 개수에 비해 상대적으로 적게 유지하면, MO-로그 존재 확인 횟수가 줄어들게 되고 볼륨 정보 캐싱에 필요한 메모리의 양을 줄일 수 있다.

5.3 영구적 장애 복구

장애가 영구적인 것으로 판별이 되면 마스터는 영구적 장애가 발생한 데이터 서버 D_{PF}에 대한 볼륨 재복제를 시작한다(그림 5). 모니터링 시스템이 영구적 장애를 감지하면 재복제 관리자에게 이를 알린다. 재복제 관리자는 D_{PF}에 저장된 모든 볼륨들을 우선순위 큐(priority queue)에 넣는다. 초기의 우선순위는 볼륨 내의 N 상태 복제본의 개수에 의해 결정되며, 볼륨이 큐에 오래 머무른 경우나 관리자가 우선 순위를 부여한 경우 변경될 수 있다. 재복제 스케줄러는 해당 볼륨을 새로 저장할 데이터 서버인 D_M과 현재 볼륨에 속한 N 상태의 복제본을 가지고 있는 데이터 서버인 D_N을 결정하고, D_M에 볼륨 재복제 요청을 보낸다. D_M은 D_N으로부터 복제본을 복사해오며, 볼륨 재복제 중 받은 업데이트 요청들은 DO-로그에 기록하여 복사가 끝난 후 처리한다.

볼륨 재복제는 몇 가지 고려해야 할 사항들이 있다. 첫 번째로 마스터는 시스템 내에 볼륨 재복제 중인 볼륨의 개수를 조절해야 한다. 볼륨이 다수의 작은 파일들로 이루어져 있기 때문에 볼륨 재복제는 작은 크기의 빈번한 디스크 I/O을 유발하므로, 많은 볼륨들에 대해 볼륨 재복제가 동시에 진행되면 성능 저하가 커지게 된다. 두 번째로 D_M과 D_N에 발생한 장애로 인해 볼륨 재복제가 실패할 수 있다는 것이다. 마스터는 재복제 실패를 인식하는 즉시, 재복제 관리자에게 알림으로써 해당

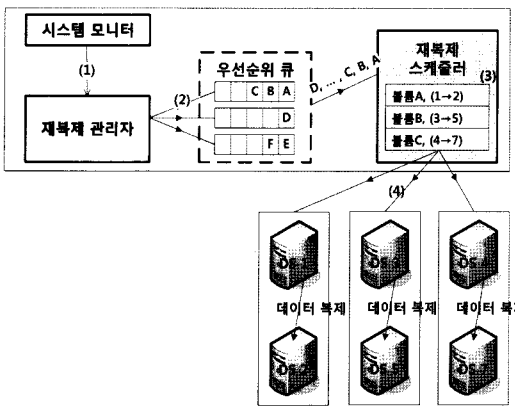


그림 5 볼륨 재복제

볼륨의 재복제를 재시도한다. 이전에 재복제 중이던 볼륨 정보는 추후 볼륨 동기화를 통해 삭제된다.

또한, 볼륨 재복제 기법은 영구적 장애가 발생하지 않아도 데이터 서버들 간의 스토리지 사용을 균일화하기 위해 사용된다. 데이터 서버가 새로 추가되거나 스토리지 사용량이 편중된다고 판단되는 경우나 별도의 관리자 개입이 있는 경우, BeanFS는 볼륨 재복제와 동일한 과정을 통해 데이터 서버들의 스토리지 사용량을 조절한다.

6. 성능 평가

실험에 사용된 BeanFS 클러스터는 하나의 마스터와, 최대 64대의 데이터 서버, 최대 32대의 클라이언트 장비로 구성되어 있다. 사용된 장비는 모두 두 개의 듀얼코어 Xeon 프로세서(2.0 GHz), 2 GB 메모리, 네 개의 500 GB SATA 하드 디스크, 온보드 기가빗 이더넷 컨트롤러를 가지고 있다.

6.1 마이크로 벤치마크(Micro-benchmark)

먼저 마이크로 벤치마크를 이용하여 파일 크기 및 클라이언트 수, 데이터 서버의 수를 각각 변화 시킬 때, 전체 클라이언트의 파일 시스템 대역폭 총합이 어떻게 변화하는지 측정하였다. 각 실험에서 읽고 쓰는 파일 데이터의 총합은 32 GB이며 하나의 볼륨에는 파일 크기와 상관 없이 총 64 MB의 파일 데이터가 저장되도록 하였다.

첫 번째 실험(그림 6)은 클라이언트와 데이터 서버의 수를 각각 8대, 64대로 고정한 후, 파일 크기를 변화시켰을 때 쓰기와 읽기 성능의 변화를 나타낸다. 파일 크기가 작은 경우에는 파일이 증가하면서 대역폭도 증가하는 모습을 보여주지만, 일정 크기를 넘어서면 증가가 정체 되는 것을 볼 수 있다. 쓰기의 경우, 읽기보다 빨리 정체되는 경향이 있다. 이는 쓰기의 경우 세 복제본 모두에게 요청을 보내기 때문에 실제로 사용하는 디스크와 네트워크 대역폭이 읽기의 세 배이기 때문이다. 실제로 읽기와 쓰기 대역폭이 모두 정체되는 시점인 512 KB 파일의 경우, 읽기의 대역폭은 쓰기의 대역폭의 세 배에 조금 못 미친다.

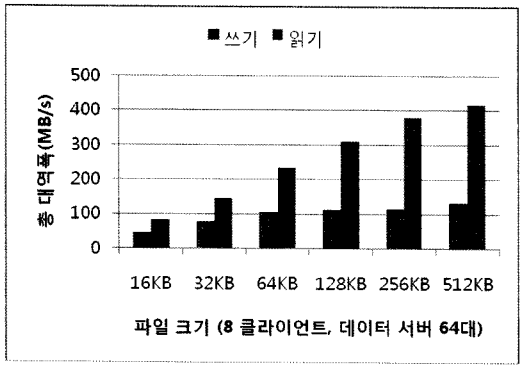


그림 6 파일 크기에 따른 성능 변화

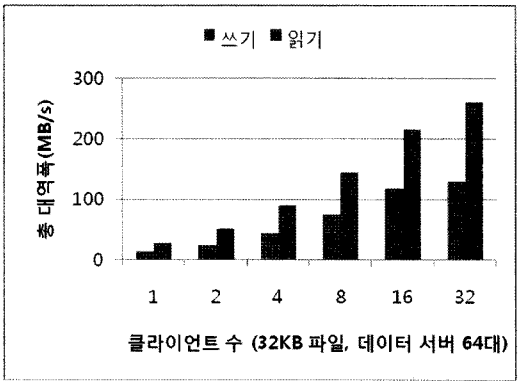


그림 7 클라이언트 수에 따른 성능 변화

두 번째 실험(그림 7)은 파일 크기와 데이터 서버의 수를 각각 32 KB와 64대로 고정한 후, 클라이언트 수를 변화시켰을 때 쓰기와 읽기 성능의 변화를 측정한 결과이다. 클라이언트의 수가 늘어나면서 대체로 성능이 증가하지만 클라이언트가 많아질수록 클라이언트 수에 정확하게 비례하여 증가하지는 않는 것을 확인 할 수 있다. 이는 클라이언트가 많아지면서 같은 디스크에 동시에 접근하는 경우가 빈번하게 되어 평균 대기시간이 증가하는 것이 원인으로 보인다.

마지막으로 파일 크기와 클라이언트 수를 각각 32 KB와 32대로 고정한 후, 데이터 서버의 수를 증가시켰을 때 쓰기와 읽기 성능의 변화를 측정하였다(그림 8). 역시 데이터 서버의 수가 증가함에 따라 성능이 증가함을 알 수 있다. 특징적인 것으로 데이터 서버가 64개일 때 읽기 성능이 급격히 증가하는 것을 볼 수 있다. 이는 데이터 서버 수의 증가에 따라 동시에 읽기를 처리할

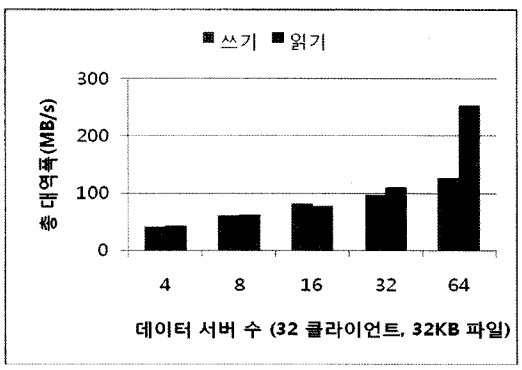


그림 8 데이터 서버 수에 따른 성능 변화

수 있는 가용 디스크의 수가 많아짐에 따라 찾기(seek)로 인해 발생되는 지연이 줄어들기 때문이다.

6.2 볼륨 동기화

볼륨 동기화의 성능을 측정하기 위하여, 2장에서 언급한 이메일 메시지 크기 분포를 이용하여 임의의 이메일 계정 1,000개를 각각 볼륨으로 만들어 데이터 서버 3대에 저장하였다. 그리고 하나의 데이터 서버에 장애를 발생시키고 특정 수의 업데이트를 수행한 후, 장애가 발생했던 데이터 서버에서의 볼륨 동기화 시간을 측정하였다. 그림 9에서와 같이 초기화를 제외한 로그 수신과 로그 재연 시간은 장애 기간 동안 발생한 업데이트 수, 즉 로그의 수에 비례하는 것을 확인 할 수 있었다.

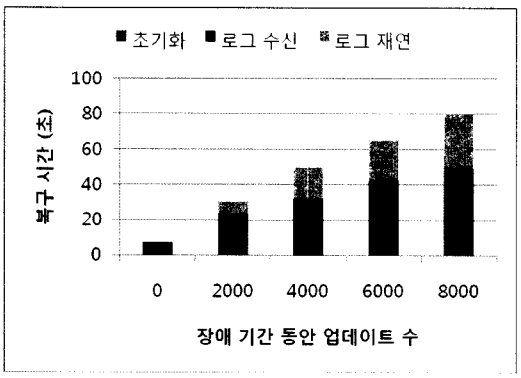


그림 9 볼륨 동기화 시간

6.3 볼륨 재복제

볼륨 재복제의 성능을 측정하기 위하여, 먼저 64대의 데이터 서버에 200 GB 정도의 데이터를 이메일 메시지 크기 분포대로 생성하였다. 그리고 재복제 동시성(동시에 수행되는 볼륨 재복제의 수)을 변화시키면서 하나의 데이터 서버가 저장하는 볼륨들을 모두 재복제 했을 때, 볼륨 재복제 대역폭을 측정하였다. 그림 10에서와 같이 동시성이 32가 되는 이후로 성능이 정체되는 것을 볼 수 있었다. 이는 데이터 서버 수의 절반에 해당하는 값으로, 하나의 데이터 서버에 동시에 두 개 이상의 볼륨 재복제에 관여하는 것은 성능에 이득을 주지 못함을 보여준다. 비록 각 데이터 서버가 네 개의 독립적인 디스크를 가지고 있지만 작고 많은 디스크 I/O 환경에서 디스크 대역폭의 총합은 정체된다.

동시성이 32일 때 데이터 서버 하나의 볼륨 재복제 대역폭은 약 11MB/s이다. 데이터 서버에 약 1TB(50%의 사용량)의 파일 데이터가 저장되어 있는 경우, 해당 서버에 대하여 볼륨 재복제를 수행하는데 약 25시간 정도의 시간이 소요된다. 이는 6.2에서 보여준 일반적인 환경에서 볼륨 동기화에 소요되는 시간에 비하여 매우

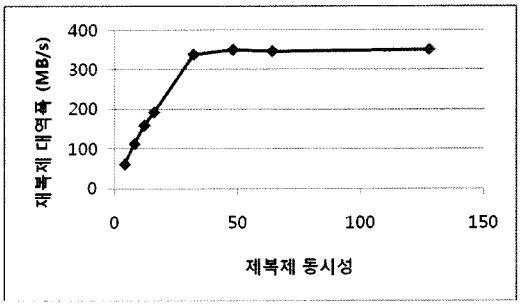


그림 10 재복제 성능

큰 값으로 영구적 장애와 같은 불가피한 경우에만 볼륨 동기화를 통해 데이터 서버를 복구하도록 하고 일시적 장애를 복구하는 데 있어서 log재연으로 복구하는 것이 보다 더 효율적임을 알 수 있다.

7. 관련 연구

기존의 저가 하드웨어 클러스터를 이용하는 분산 파일 시스템은 복제본을 관리하는 방법에 따라서 두 가지로 분류될 수 있다. GFS[2], Ceph[5], Harp[6] 모두 모든 복제본이 프라이머리 복제본에서 직렬화 되는 프라이머리 복제본 기법을 이용한다. 이것은 파일 시스템의 강력한 일관성 유지를 가능하게 하지만 일반적으로 낮은 가용성을 보여준다. 또한, 커밋 상태를 영구적으로 유지하기 위해서 복제본 사이의 통신과 동기화된 디스크 쓰기 등이 많이 필요하다.

한편, Dynamo[7], Procupine[14], Coda[10] 시스템에서는 낙관적 복제본 기법(optimistic replication technique)을 이용한다. 이 방법은 프라이머리 복제본 기법에 비해 약한 일관성을 유지하면서 좋은 가용성을 보여준다. 하지만 쓰기 충돌(write conflict)에 대비해서 응용 고유의 해결자(application-specific resolver)[7,10]나 약동기화된 클락(loosely synchronized clock)[14], 혹은 쿼럼 프로토콜[7]을 이용해야 한다. 쓰기 충돌이 없는 이메일 메시지의 경우에는 이러한 일관성 유지가 모두 불필요하다.

분산 파일 시스템은 메타데이터 관리 기법에 따라 다시 두 가지 분류로 나눌 수 있다. 중앙 집중형 기법[2,4]은 시스템의 모든 메타데이터를 하나의 서버에서 관리한다. 기본적으로 파일 시스템 디자인을 간단하고 유연하게 만들어 주지만, 중앙 서버의 병목 현상을 피할 수 없다. Ceph[5]나 Lustre[15]에서는 이 문제를 다수의 메타데이터를 클러스터링 해서 해결하려 했지만 부하 분산이 실패하거나 더 많은 메타데이터 연산이 몰릴 경우 여전히 병목현상을 피할 수 없다.

분산형 기법은 메타데이터 서버의 확장성 문제를 해

결하는 대안이 될 수 있다. Dynamo[7]는 P2P 방식과 유사하게 consistent hashing을 이용해서 파일을 담당하는 서버를 찾는다. Dynamo는 임의로 파일을 모든 서버에 분산시키지만 해시 값(hashing value)에 의존하기 때문에 실제로는 부하 불균형을 피할 수 없게 된다. Coda[10]에서는 복제본이 위치하는 서버의 정보인 볼륨 위치 정보(VLDB: Volume Location DataBase)를 모든 서버에 중복해서 저장한다. 이러한 방법은 볼륨의 수가 늘어날수록 VLDB의 크기가 증가하고 이를 모든 노드에서 동기화 하는 것이 어려워지게 된다.

8. 결 론

본 논문에서는 대규모 이메일 서비스를 위한 분산 파일 시스템인 BeanFS의 설계와 구현에 대해서 소개하였다. BeanFS는 기존의 범용의 분산 파일시스템과 달리, 단순한 접근 패턴을 가지고 있는 이메일 메시지를 효과적으로 저장하는 것을 목표로 개발되었다.

BeanFS의 확장성, 고가용성, 고성능을 위한 주요 특징들을 정리하면 다음과 같다. 첫째로, 볼륨기반의 복제 기법을 이용하여 중앙 서버의 메타데이터 관리 오버헤드를 줄였다. 마스터는 각 볼륨에 해당하는 위치정보와 상태 정보만을 관리한다. 둘째로, BeanFS는 복잡한 프라이머리 복제본 기법이나 쿼럼 프로토콜 없이 이메일 메시지에 맞추어 경량화된 일관성 유지 기술을 사용한다. 모든 분산된 구성요소들은 상태 매칭 기법을 통해 일관성 있는 볼륨 상태를 보장 받는다. 마지막으로 BeanFS는 볼륨 동기화를 통해서 일시적 장애를, 재복제를 통해서 영구적 장애를 복구한다. 이러한 기법들은 BeanFS의 가용성을 효과적으로 증가시킨다.

참 고 문 헌

[1] International Data Corporation (IDC), The diverse and exploding digital universe, 2007.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, The google file system, In *Proceeding of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pp.29-43, 2003.

[3] Amazon simple storage service (amazon s3), http://aws.amazon.com/s3.

[4] The hadoop opensource project, http://lucene.apache.org/hadoop/.

[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06), pp. 307-320, 2006.

[6] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira, Replication in the harp file system, In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pp.226-238, 1991.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, Dynamo: amazon's highly available key-value store, In *Proceedings of 21st ACMSymposium on Operating Systems Principles (SOSP'07)*, pp.205-220, 2007.

[8] S.-H. Kim, Y. Lee, and J.-S. Kim, Flexrpc: A flexible remote procedure call facility for modern cluster file systems, In *Proceeding of 9th IEEE International Conference on Cluster Computing (CLUSTER'07)*, pp.257-284, 2007.

[9] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, Andrew: a distributed personal computing environment, *Communications of the ACM (CACM)*, 29(3):184-201, 1986.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, Scale and performance in a distributed file system, *ACM Transactions on Computer System (TOCS)*, 6(1):51-81, 1988.

[11] D. Skeen and M. Stonebraker, A formal model of crash recovery in a distributed system, pp.295-317, 1987.

[12] Mysql reference manual, http://dev.mysql.com/doc/.

[13] D. A. Patterson, G. Gibson, and R. H. Katz, A case for redundant arrays of inexpensive disks (raid), In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*, pp.109-116, 1988.

[14] Y. Saito, B. N. Bershad, and H. M. Levy, Manageability, availability and performance in porcupine: a highly scalable, clusterbased mail service, In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pp.1-15, 1999.

[15] Cluster File Systems, Inc. Lustre: A Scalable, High-Performance File System, http://www.clusterfs.com.

정 욱

2004년 한국과학기술원 전자전산학과 전산학전공 학사. 2004년~현재 한국과학기술원 전자전산학과 전산학전공 석박사 통합과정 재학. 관심분야는 운영체제, 분산 시스템, 스토리지 시스템

이 대 우

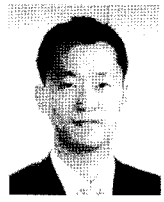
2002년 한국과학기술원 전자전산학과 전기 및 전자공학 전공 학사. 2003년~현재 한국과학기술원 전자전산학과 전산학전공 석박사통합과정 재학. 관심분야는 운영체제, 분산 시스템, 스토리지 시스템

박 은 지

2002년 한국과학기술원 전자전산학과 전산학전공 학사. 2004년 한국과학기술원 전자전산학과 전산학전공 석사. 2004년~현재 한국과학기술원 전자전산학과 전산학전공 박사과정 재학. 관심분야는 분산 시스템, 컴퓨터 구조

이 영 재

2005년 한국과학기술원 전자전산학과 전산학전공 학사. 2005년~현재 한국과학기술원 전자전산학과 전산학전공 석박사 통합과정 재학. 관심분야는 운영체제, 분산 시스템, 스토리지 시스템



김 상 훈

2002년 한국과학기술원 전자전산학과 전산학전공 학사. 2002년~2003년 (주)FnBC 전임 연구원. 2003년~2005년 (주)인젠 전임 연구원. 2006년~현재 한국과학기술원 전자전산학과 전산학전공 석박사통합과정 재학. 관심분야는 운영체제, 분산 시스템, 스토리지 시스템

김 진 수

1991년 2월 서울대학교 컴퓨터공학과 공학사. 1993년 2월 서울대학교 컴퓨터공학과 공학석사. 1999년 8월 서울대학교 컴퓨터공학과 공학박사. 1998년 5월~1999년 4월 IBM T. J. Watson Research Center, Academic Visitor. 1999년 9월~2002년 2월 한국전자통신연구원(ETRI) 선임연구원. 2002년 3월~2008년 8월 한국과학기술원(KAIST) 전산학과 부교수. 2007년 9월~2008년 8월 삼성전자 VD사업부 고문. 2008년 9월~현재 성균관대학교 정보통신공학부 부교수. 관심분야는 운영체제, 스토리지 시스템, 분산 시스템, 내장형 시스템

김 태 웅

1993년 2월 서울대학교 컴퓨터공학과 공학사. 1995년 2월 서울대학교 컴퓨터공학과 공학석사. 2000년 8월 서울대학교 전기,컴퓨터공학부 공학박사. 2000년 9월~2000년 12월 서울대학교 컴퓨터신기술공동연구소 특별연구원. 2000년 12월~2006년 9월 데이타코러스(주) 기술이사. 2006년 10월~현재 NHN㈜ 가상화플랫폼개발팀장. 관심분야는 분산파일시스템, 운영체제, 가상머신, 스토리지 가상화

전 성 원

1997년 한국과학기술원 전자전산학과 전산학전공 학사. 1999년 한국과학기술원 전자전산학과 전산학전공 석사. 현재 NHN 재직중. 관심분야는 분산시스템, 스토리지 시스템, 파일시스템