24시간 365일

서버/인프라를 지탱하는 기술

무중단 서비스를 위해 지금 당장 할 수 있는 것은 무엇인가?

확장성 고성능 효율성

이토 나오야, 카츠미 유키, 다나카 신지, 히로세 마사아키, 야스이 마사노부, 요코가와 카즈야 공저 / 진명조 옮김

Jpub 제이펍

“24 JIKAN 365 NICHI”
SERVER/INFRA O SASAERU GIJUTSU

24시간 365일 **서버/인프라를 지탱하는 기술**

1쇄 발행 2009년 4월 22일
7쇄 발행 2022년 3월 31일

지은이 이토 나오야, 카츠미 유키, 다나카 신지, 히로세 마사아키, 야스이 마사노부, 요코가와 카즈야
옮긴이 진명조
펴낸이 장성두
펴낸곳 주식회사 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 회동길 159 3층 / **전화** 070-8201-9010 / **팩스** 02-6280-0405
홈페이지 www.jpub.kr / **원고투고** submit@jpub.kr / **독자문의** help@jpub.kr / **교재문의** textbook@jpub.kr

편집부 김정준, 이민숙, 최병찬, 이주원, 송영화
소통기획부 이상복, 송찬수, 배인혜 / **소통지원부** 민지환, 김수연 / **총무부** 김유미

진행 및 교정 · 교열 장성두 / **내지디자인** 북아이 / **표지디자인** Aowa & Arowana
용지 신승지류유통 / **인쇄** 해외정판사 / **제본** 일진제책사

ISBN 978-89-962410-0-3 (13000)
값 25,000원

차례

CHAPTER 01

서버/인프라 구축 입문
다중화/부하분산의 기본

CHAPTER 02

한 단계 높은 서버/인프라 구축
다중화, 부하분산, 고성능 추구

CHAPTER 06

서비스의 무대 뒤
자율적인 인프라, 다이나믹한 시스템 지향

한국어판 서문

한국의 독자 여러분!

이번에, 저희 집필진이 쓴 이 책이 한국어로 번역된다는 말을 듣고 영광으로 생각함과 동시에 한국의 독자 여러분께 도움을 드릴 수 있게 되어 집필진 모두는 매우 기쁘게 생각하고 있습니다.

이 책은, 일본에서 웹 서비스를 제공하고 있는 ㈜Hatena와 KLab㈜의 엔지니어가 「확장성」, 「고성능」, 「운용의 효율화」라는 3가지를 키워드로 하여, 집필진의 경험을 바탕으로 한 노하우를 정리한 것입니다.

양사는 웹 서비스를 제공하고 있다는 공통점은 있지만 협업을 하고 있는 것은 아니므로, 서로의 인프라 시스템에 관해서는 잘 알지 못 했습니다. 그러나, 이 책을 집필하는 과정에서 양측의 시스템을 알게 되면서, 최종적인 구현 모습은 다를지라도 거기에 이르는 설계사상에는 유사한 점이 많아서 놀랐던 기억이 납니다.

저는 한국어판 책에서도 동일한 노하우를 전할 수 있으리라 생각합니다. 즉, 살고 있는 나라나 말하는 언어가 다르더라도 「오픈소스 소프트웨어와 필수 하드웨어로 만든, 웹 서비스를 위한 인프라」라는 공통점이 있다면, 분명 그 노하우는 국가나 언어의 벽을 넘어서 통할 것이라고 저는 생각합니다.

틀림없이, 다양한 상황이나 환경이 있을 것이므로, 이 책에 쓰여져 있는 내용이 독자 여러분의 시스템에 있는 그대로 적용할 수는 없을지도 모릅니다. 하지만, 여러분과 집필진 사이에는 「서버/인프라」라는 공통점이 있습니다. 분명, 우리의 노하우가 담긴 이 책 속에서 여러분은 새로운 발견과 식견을 얻을 수 있으리라 믿습니다. 그리고, 바다 건너에 있는 여러분에게 도움을 드릴 수 있어서 집필진은 매우 기쁘게 생각합니다.

끝으로, 이런 종류의 기술서는 영문 원서를 번역한 것이 많지만, 이렇게 좋은 기회를 준 번역 관계자 여러분, 그리고 이 책을 지금 손에 들고 있는 여러분에게 진심으로 감사드립니다.

"당신의 엔지니어 인생이 늘 행복하길 기원하며…"

2009년 4월

도쿄 록본기에서

저자들을 대표해서 **히로세 마사아키**

옮긴이 서문

학습, 연구 목적으로 또는 상용 서비스를 제공하기 위해 기본적인 네트워크, 서버를 구축하고 나름대로의 보안성과 성능을 유지하고 최대한 장애가 발생하지 않도록 안정적으로 운용, 관리하는 것이 시스템 관리자가 일차적으로 수행해야 할 임무다. 이를 위해 다양한 설치, 운용관련 기술문서와 서적을 검색, 탐독하고 수많은 시행착오를 겪으면서 운영체제, 서비스 프로그램 등의 설치, 환경설정, 삭제 등을 반복하게 될 것이며, 지금 이 책을 읽고 있는 독자라면 아마도 이미 이러한 과정을 지나왔을 것이라 생각한다. 그리고, 이제는 한 단계 높은 레벨의 서버/인프라 관리방안을 모색하고 있으리라 51% 확신하는 바이다.

"24시간 365일 무중단 서비스를 위해 지금 무엇을 할 수 있는가?"

당신의 질문이 위와 같다면 이 책이 그 해답이 될 것이다. 이 책에서는 이미 널리 알려진 혹은 저자들이 직접 제작하여 실무에 적용해온 오픈소스 소프트웨어를 이용하여 중단 없는 서비스를 운용하기 위한 '다중화', '성능향상', '확장성', '운용효율성'에 대해 설명하고 있다. 아울러, 이 책의 저자들이 소속되어 있는 '일본의 구글'이라 칭송받는 ㈜Hatena와 모바일 플랫폼 서비스를 제공하는 KLab㈜의 서버/인프라 구성환경, 운용도구에 대한 소개를 비롯해 운용사례 및 차후 개선방향까지도 언급하고 있어, '다중화', '확장성', '효율적 운용방안' 등에 관한 이론과 함께 실례를 접할 수 있다.

이 책을 읽으면서 서버/인프라 관련 기술을 접하게 되면 자연스레 그 기술의 정점에 있는 '구글'을 떠올리게 된다. 구글의 서버/인프라 구성에 관한 바탕을 이루는 생각은 비용절감, 저가형 하드웨어, 스마트한 소프트웨어에 집중하는 것이라고 한다. 기능에 적합하게 튜닝된 리눅스 운영체제를 이용하고, 서버, 랙, 데이터센터 레벨의 장애가 일어나더라도 데이터의 손실이나 전체 시스템 다운이 방지되도록 '다중화' 되어 있으며, 시스템 규모 확장시에는 플러그인 방식에 의

해 세팅과 설정이 가능할 정도로 '확장성'과 '운용이 표준화, 효율화' 되어 있다. 구글은 이러한 기술들을 우수한 인력을 이용해 자체 개발하여 운용하고 있다. 물론, 우리가 다루는 서버/인프라 환경은 구글의 그것과는 많이 다르겠지만, 중단 없이 서비스해야 한다는 목표와 필요로 하는 기술은 유사하리라 생각한다.

모든 시스템 관리자에게 '서버/인프라를 지탱하는 기술'이 필요하지는 않겠지만, 더 나은 서비스를 위해 필수적인 것만은 사실이다. 한 단계 높은 레벨의 서버/인프라 관리방안에 대해 고민해본 적이 있다면 이 책에 있는 기술들을 꼭 한 번 적용해보길 바란다.

오늘을 함께 살아가고 있는 가족, 친구 그리고 회사동료들에게 고마운 마음을 전한다. 그리고 부족한 저를 믿고 제이펍의 첫 책의 번역을 맡겨주신 장성두 실장님께, 지면을 통해 제이펍의 첫 출간 축하드리고 감사드린다. '0과 1로 밥먹고 사는 사람들을 위해' 좋은 책 많이 내주시기를...

2009년 4월

진명조

진명조 … 고려대학교 재료공학부를 졸업하고 ㈜오늘과내일 연구소에서 근무 중이다. 『입문자를 위한 루비』(2009), 『Binary Hacks : 해커가 전수하는 테크닉 100선』(2007), 『C언어로 배우는 알고리즘 입문』(2004) 등을 번역하였다. IT 개발자의 삶 속에서 작은 보람을 찾고자 오늘도 주어진 업무에 최선을 다하고 있다.

지은이 서문

SNS나 블로그, 쇼핑 사이트를 시작으로 다양한 웹서비스, 메일이나 채팅 등의 커뮤니케이션 툴 등, 이미 인터넷은 우리 생활에 없어서는 안 될 인프라라고 해도 과언이 아니다. 역시나 필자도 예외 없이 매일 인터넷을 사용하고 있다. 보다 정확히 표현하면, 공과 사를 가릴 것 없이 완전히 인터넷의 바다에 빠져 나날을 보내고 있다.

그런데 필자의 경우에는 – 분명, 지금 이 책을 보고 있는 여러분도 똑같으리라 생각하지만 – 「사용자」와는 다른 측면을 가지고 있다. 바로 서비스 「제공자」라는 측면이다. 필자의 경우는 네트워크나 서버 구축, 운용관리를 직업으로 하고 있다.

예전에는 네트워크나 서버라고 하면 기자재가 고가여서 그리 간단하게 접할 수 없는 분야였다. 그러나 요즘은 리눅스Linux나 FreeBSD를 시작으로 PC UNIX가 보급되고, 하드웨어 가격이 많이 저렴해졌으며, 네트워크에 항상 접속할 수 있는 덕분에 가정에서 네트워크를 구성하고 서버를 구축해서 이용하는 사람도 많아졌다.

이러한 상황과 맞물려 인프라 관련 정보도 자주 접할 수 있게 되었다. 특히, 설치방법이나 아파치Apache 등의 데몬 설정 등 이른바 하우투How-To의 내용이 사뭇 놀랄 만큼 충실해져서 초보 엔지니어에게는 편리한 세상이 되었다고 생각한다.

한편, 그 다음 단계인 운용관리작업의 효율화, 서비스의 다중화나 확장성과 같은 주제의 기술적인 정보나 노하우는 아직도 모자란 느낌이다.

필자의 경우도 몇 대의 서버로 구성된 시스템을 구축, 운용하는 것부터 시작했지만 수십 대에서 수백 대 규모로 장비가 증가하면서 느끼는 가장 커다란 벽은 「다중화」, 「확장성」에 관한 정보였다. 당시 필자는 「다중화」나 「확장성」에 관한 지식이나 경험도 없었으며, 어디서부터 손을 대어야 좋을지조차도 모르는 상태

였다. 또한, 이를 실현하기 위해서 고가의 상용제품을 이용해야만 한다는 잘못된 믿음을 가지고 있어서 테스트를 해보려 해도 좀처럼 만져볼 수 없었다.

지금 생각해보면 당시의 이런 생각은 잘못된 것이었다. 사실은 오픈소스 소프트웨어OSS와 일반적인 도구로 「다중화」와 「확장성」을 겸비한 시스템을 구축할 수 있기 때문이다. 그렇다면 돌이켜서 무엇이 원인이었는지 생각해보면, 단순히 '그런 게 있을 줄은 몰랐어.', '그런 게 가능할 줄은 몰랐어.' 라는 반응으로 그칠 것으로 생각한다.

바로 여기에 이 책을 집필한 동기가 있다. 즉, 이 책의 목표는 「다중화」되고 「확장성」도 있는 인프라를 구축하기 위한 힌트를 여러분에게 전하기 위해서다.

이 책에는 다소 의도적으로 오픈소스를 사용하고 있다. ㈜Hatena와 KLab㈜의 엔지니어 그룹이 허황되지 않은, 실제 가동중인 시스템에 관련된 보다 실천적인 정보를 전달한다. 시스템이란 「계系」다. '계' 라는 것은 각각의 요소가 서로 연관되어 구성된 것이다. 이 책에서는 각각의 요소기술에 대해 상세히 설명하고 상호 연관성이나 흐름, 연결관계를 중요시한 내용이 될 수 있도록 힘을 쏟았다. 또한 이 책은 하우투how-to 책이 아니다. 따라서 꼼꼼하게 설치순서를 설명해주지는 않으며, 책에 쓰여진 대로 명령을 실행한다고 해서 뭔가 이루어지지도 않는다.

이 책에 기술되어 있는 것은 실제 현장에서 우리가 어떻게 생각하고 고민하고 연구해왔는지, 그 궤적과 성과의 결과물이다. 「독자 여러분이 이후에 인프라를 설계, 구축, 운용관리할 때, 이 책이 조금이나마 의지가 되고 지식이 되었으면…」하는 바람을 담아 이 책을 집필했다.

2008년 7월

저자들을 대표해서 **히로세 마사아키**

이 책의 구성

이 책은 총 6장으로 구성되어 있다.

1장 서버/인프라 구축 입문 …… 다중화/부하분산의 기본

2장 한 단계 높은 서버/인프라 구축 …… 다중화, 부하분산, 고성능 추구

3장 무중단 인프라를 향한 새로운 연구 …… DNS서버, 스토리지 서버, 네트워크

1 ~ 3장에 걸친 일관된 테마는 「다중화」와 「확장성」을 겸비한 인프라 디자인이다.

각 장의 절은 각각 독립된 주제지만, 「소규모 시스템을 출발점으로 어떻게 인프라를 정비해 나아갈까」라는 스토리 내에서 서로 연관되어 있다. 우선은 흐름을 파악하기 위해 1 ~ 3장 전체를 대략적으로 훑어보고 그 다음 관심 있는 절로 돌아가 차분히 읽어가는 방법을 추천한다.

4장 성능향상, 튜닝 …… 리눅스 단일 호스트, 아파치, MySQL

4장의 테마는 「성능향상」이다.

서버를 나열해서 로드밸런싱하고 시스템 전체의 성능향상을 꾀한다는 작전에는 그 구성요소인 개별서버의 튜닝도 빼놓을 수 없다. 4장에서는 특히 개별 성능향상에 관해 다루고 개별서버의 능력을 발휘하기 위해 필요한, 병목의 특징이나 튜닝에 대해 서술한다.

5장 효율적인 운용 …… 안정된 서비스를 향해

5장은 감시나 관리와 같은 「운용」이 테마다.

만일 서버 대수가 증가함에 따라 운용비용도 증가한다면 장래에는 운용비용

이 병목이 되어 생각처럼 인프라를 확대할 수 없을 가능성이 있다. 다른 좋은 방법을 쓰면 얼마나 운용을 효율화할 수 있는지가 확장성 있는 인프라를 키워내는 열쇠가 된다고 할 수 있다. 5장에는 집필진의 운용환경에서 어떻게 효율적인 연구를 수행하는지, 그 사례를 소개한다.

6장 서비스의 무대 뒤 …… 자율적인 인프라, 다이나믹한 시스템 지향

마지막 6장에는 ㈜Hatena와 KLab㈜에서 운용 중인 DSAS의 여러 기능에 대해, 그리고 실제로 가동 중인 네트워크, 서버 인프라에 관련된 얘기를 한다.

집필진은 인프라팀 내에서도 핵심 엔지니어들이다. 내용은 테크니컬한 얘기와 함께 지금까지의 각 장에서는 너무 사소해서 소개할 수 없는 것이나 오늘에 이르기까지의 경위, 역사, 인프라 계열 엔지니어의 모티베이션이나 마인드와 같은 주제도 포함시킴으로써 읽을거리로서도 재미있게 구성하였다.

집필담당 및 출처

절	집필담당
1.1 다중화의 기본	야스이 마사노부 (KLab㈜)
1.2 웹 서버의 다중화 DNS 라운드로빈	야스이 마사노부
1.3 웹 서버의 다중화 IPVS를 이용한 로드밸런서	야스이 마사노부
1.4 라우터 및 로드밸런서의 다중화	야스이 마사노부
2.1 리버스 프록시 도입 아파치 모듈	이토 나오야 (㈜Hatena)
2.2 캐시서버 도입 Squid, emcached	이토 나오야

절	집필담당
2.3 MySQL 리플리케이션 단시간에 장애복구하기※1	히로세 마사아키 (KLab㈜)
2.4 MySQL 슬레이브 + 내부 로드밸런서 활용 예※2	히로세 마사아키
2.5 고속, 경량의 스토리지 서버 선택	야스이 마사노부
3.1 DNS서버의 다중화	야스이 마사노부
3.2 스토리지 서버의 다중화 DRBD로 미러링 구성	야스이 마사노부
3.3 네트워크의 다중화 Bonding 드라이버, RSTP	카츠미 유키 (KLab㈜)
3.4 VLAN 도입 유연한 네트워크 구성	요코가와 카즈야 (KLab㈜)
4.1 리눅스 단일 호스트 부하의 진상규명	이토 나오야
4.2 아파치 튜닝	이토 나오야
4.3 MySQL 튜닝의 핵심※3	히로세 마사아키
5.1 서비스의 가동감시 Nagios	다나카 신지 (㈜Hatena)
5.2 서버 리소스 모니터링 Ganglia※4	히로세 마사아키
5.3 서버관리의 효율화 Puppet	다나카 신지
5.4 데몬의 가동관리 daemontools	히로세 마사아키
5.5 네트워크 부트의 활용 PXE, initramfs	카츠미 유키
5.6 원격관리 관리회선, 시리얼 콘솔, IPMI	카츠미 유키
5.7 웹 서버 로그관리 syslog, syslog-ng, cron, rotatelogs	카츠미 유키
6.1 Hatena의 내부	다나카 신지
6.2 DSAS의 내부	야스이 마사노부

출처 : ※ 1 『WEB+DB PRESS』 (Vol.22)의 특집2 「MySQL 전환 안내」, 2장 「현장 지향의 리플리케이션 해설」

※ 2 『WEB+DB PRESS』 (Vol.38)의 연재, [보이리라! 장인의 기술] 확장성 있는 웹 시스템 연구 「제1회 : 다양한 형태의 로드밸런스」

※ 3 「5분 만에 하는 MySQL 메모리 관련 튜닝!」

URL http://dsas.blog.klab.org/archives/50860867.html

※ 4 『WEB+DB PRESS』 (Vol.40)의 연재, [보이리라! 장인의 기술] 확장성 있는 웹 시스템 연구 「제3회 : 그 밖에 모니터링 관련」

용어 정리

이 책에서 다루는 내용은 네트워크부터 애플리케이션에 이르기까지 범위가 넓고 다양한 용어가 나온다. 우선, 자주 사용하는 용어를 정리해보자.

AP서버(Application Server)

애플리케이션 서버. 동적 콘텐츠를 반환하는 서버를 말함. 예를 들면, Apache + mod_perl이 동작하는 웹 서버나 Tomcat과 같은 애플리케이션 컨테이너가 동작하는 서버.

CDN(Content Delivery Network)

콘텐츠를 전송하기 위한 네트워크 시스템. 전송 성능향상과 가용성 향상을 목적으로 한다. Akamai 등 몇몇 상용 서비스가 존재하며, 전 세계에 존재하는 캐시 서버 중에 클라이언트에 보다 가까운 캐시 서버를 선택해서 전송함으로써 성능향상을 실현하는 것이 구성상 특징임.

IPVS(IP Virtual Server)

LVS[Linux Virtual Server] 프로젝트의 성과물로, 로드밸런서에 불가결한 「부하분산」 기능을 실현함.

➡ 「LVS」 참조.

LVS(Linux Virtual Server)

리눅스에서 확장성이 있고 가용성이 높은 시스템을 만드는 것을 목표로 하고 있는 프로젝트. 그 성과물 중 하나로 리눅스 로드밸런서를 위한 IPVS가 있다. 본래는 프로젝트명이지만 관례적으로 「LVS」를 「리눅스로 만든 로드밸런서」라는 의미로 쓰기도 한다.

URL http://www.linuxvirtualserver.org/

Netfilter

리눅스 커널 상에서 네트워크 패킷을 조작하기 위한 프레임워크. 패킷 필터링 등을 수행하는 iptables나 로드밸런스를 실현하기 위한 IPVS도 Netfilter의 기능을 이용하고 있다.

NIC(Network Interface Card)

본래는 네트워크 기능을 추가하기 위한 카드를 가리키는 용어지만, 확장카드나 온보드On-board를 가리지 않고 네트워크 인터페이스를 총칭해서 사용되기도 한다. LAN카드, 네트워크 카드, 네트워크 어댑터라고도 한다.

OSI 참조 모델

데이터통신을 위한 네트워크 계층을 설명한 모델. 7개 계층(레이어layer)으로 되어 있다.

자주 접하는 레이어는 다음과 같다.

- **레이어7 (애플리케이션 계층)** : HTTP나 SMTP와 같은 통신 프로토콜
- **레이어4 (트랜스포트 계층)** : TCP나 UDP
- **레이어3 (네트워크 계층)** : IP나 ARP, ICMP
- **레이어2 (데이터링크 계층)** : 이더넷Ethernet 등

또한 「L2스위치」와 같이 「Layer n」은 「Ln」으로 표기하기도 한다. OSI는 Open Systems Interconnection의 약자.

VIP(Virtual IP Address)

물리적인 서버나 NIC가 아니라 유동적인 서비스나 역할에 할당된 IP주소를 말함. 예를 들면, 로드밸런서의 경우에는 클라이언트의 요청을 받아들이는 IP주소를 VIP라고 한다. 왜냐하면 이 IP주소는 HTTP 등의 서비스에 관련된 것이기 때문이며, 또한 다중화를 위해 Active/Backup 구성을 할 경우에는 유일한 마스터가 되는 Active측의 로드밸런서가 이 IP를 인계하기 때문이다. 가상주소, 가상

IP주소라고도 한다.

가용성(Availability)

시스템을 정지시키지 않음을 뜻함. 「가용성이 높다」라고 하면 「해당 서비스는 거의 멈추지 않는다」라는 의미다. 또한, 문맥에 따라서는 「가동률이 높다」거나 「연중 가동시간이 길다」라는 의미로도 사용된다.

다중화(Redundancy)

시스템의 구성요소를 여러 개 배치해서 하나가 고장 나서 정지해도 바로 교체해서 서비스가 멈추지 않도록 하는 것을 말한다. RAID(Redundant Arrays of Inexpensive Disks)가 그 전형적인 예. 이중화라고도 한다.

네트워크 부트(Network Boot)

네트워크를 통해 부팅에 필요한 부트로더나 커널 이미지 등을 전달받아 기동하는 것. 5.5절에서 소개하는 PXE는 네트워크 부트를 실현하기 위한 사양 중 하나.

네트워크 세그먼트(Network Segment)

브로드캐스트 패킷이 전달되는 범위의 네트워크를 말함. 「충돌 도메인(collision domain)」과 동일한 의미였지만, 전 이중화 구성에서는 충돌이 발생하지 않으므로 「네트워크 세그먼트 = 충돌 도메인」이라고 하기 어려워졌다.

단일장애점(Single Point of Failure)

장애가 발생하면 시스템 전체가 정지해버리는 곳. 말하자면 시스템의 급소. SPOF(Single Point of Failure)라고도 한다. 예를 들면, RAID나 전원과 같은 서버 내의 요소를 아무리 다중화하더라도 모든 서버가 한 대의 스위칭 허브에 연결되어 있다면, 시스템 전체를 볼 때 그 스위치는 단일장애점이 된다.

데몬(Daemon)

백그라운드에서 지속적으로 실행되면서 특정 작업을 수행하는 프로그램. 예를 들어, httpd나 bind 등.

데이터센터(data center)

서버 등의 기기를 수용하기 위해 만들어진 전용시설의 명칭. 공조, 정전대책, 소화, 지진대책과 같이 24시간 365일 서비스를 수행하기 위해 필요한 설비가 갖춰져 있다.

라운드로빈(Round Robin)

여러 개의 노드에 대해 순서대로 할당하거나 분산하는 것.

예를 들면, 하나의 FQDN[Fully Qualified Domain Name](전체 주소 도메인명)에 복수의 A레코드(IP주소)를 할당해서 액세스를 분산하는 「DNS 라운드로빈」이나, 복수의 서버에 순차적으로 요청을 분산하는 로드밸런서의 밸런스 알고리즘 등이 있다.

레이어(Layer)

➡「OSI 참조 모델」 참조

로드밸런서(Load Balancer)

클라이언트와 서버 사이에 위치해서 클라이언트로부터의 요청을 백엔드[backend]의 여러 서버로 적절하게 분산하는 역할을 하는 장치. 다르게 표현하면, 여러 서버를 묶어서 하나의 고성능 가상서버에 준하는 성능을 내기 위한 장치. 부하분산기라고도 한다.

리소스(Resource)

CPU나 메모리, 하드디스크 등, 서버가 지닌 하드웨어적인 자원.

예를 들면, CPU사용률이 높은 상태를 「리소스를 잡아먹고 있다」라고 표현한다.

메모리 파일시스템(Memory File System)

하드디스크와 같은 영구기억장치가 아닌 메모리상에 만든 파일시스템. 디스크상의 파일시스템과 동일하게 사용할 수 있으나 메모리상에 있기 때문에 재부팅하면 데이터가 사라지는 반면, 읽고 쓰기를 고속으로 수행할 수 있다는 장점이 있다.

부하(Load)

「부하」는 여러 종류가 있는데 크게 「CPU 부하」와 「I/O 부하」로 나눌 수 있다. 부하를 계산하기 위한 지표는 Load Average 등 몇 가지가 있다. 또한, 부하를 계측하기 위한 명령어도 top이나 vmstat 등 몇 가지가 있다. 자세한 것은 4.1절을 참조.

병목(Bottleneck)

시스템 전체의 성능을 떨어뜨리는 원인이 되는 지점.

블록되다(Blocked)

읽기 또는 쓰기처리가 완료되기를 기다리기 위해 다른 처리를 할 수 없는 상태를 「I/O 대기로 블록되어 있다」라고 한다. 주로 디스크 I/O나 네트워크 I/O에 대해 사용되는 용어지만 입출력 처리 일반에서도 사용되기도 한다.

서버팜(server farm)

수많은 서버가 모여서 구성된 인프라 시스템을 말한다. 문맥에 따라서는 데이터센터와 같은 시설을 나타내는 의미로 사용되기도 한다.

스위칭 허브(Switching Hub)

현재 시장에 있는 거의 모든 「허브Hub」는 리피터 허브Repeater Hub가 아니라 브리지 기능을 지닌 스위칭 허브다. L2스위치 또는 그냥 스위치라고도 한다.

스케일 아웃(Scale-out)

서버를 여러 대 두고 분산함으로써 시스템 전체의 성능을 향상시키는 것. 예를 들면, 로드밸런서 하위의 웹 서버의 대수를 두 배로 늘리는 것.

스케일 업(Scale-up)

단일 서버의 성능을 높임으로써 시스템 전체의 성능을 향상시키는 것. 예를 들면, 서버의 메모리를 증설하거나 보다 고성능의 기종으로 교체하는 것.

스테이징 환경(Staging Environment)

실 서비스에 투입하기 전에 최종적인 동작을 확인하기 위한 환경을 말함.

➡「프로덕션 환경」 참조

장애극복(Failover)

다중화된 시스템에서 Active인 노드(서버나 네트워크 기기 등)가 정지했을 때 자동적으로 Backup 노드로 전환되는 것. 페일오버. 아울러, 자동이 아닌 수동으로 전환되는 것은 일반적으로 「스위치오버Switchover」라고 한다.

전송량(Throughput)

네트워크와 같이 데이터통신 측면에서 사용할 경우, 단위시간당 데이터 전송량을 의미한다. 예를 들어 말하면, 「같은 자동차라도 F1 머신보다 버스가 승차가능인원이 많으므로 '전송량'이 크다」라고 할 수 있다.

➡「지연시간」 참조

지연시간(Latency)

네트워크와 같이 데이터통신 측면에서 사용할 경우, 데이터가 도달할 때까지의 시간을 의미한다. (➡「전송량Throughput」 참조)

예를 들어 말하면, 「같은 자동차라도 버스보다 F1 머신이 속도가 빠르므로 지연시간이 작다」라고 할 수 있다.

콘텐츠(Contents)

웹서비스와 관련해서 사용할 경우, 브라우저와 같은 클라이언트로 반환하는 HTML이나 이미지 데이터를 의미한다. 특히, 정적 콘텐츠라고 하면 내용이 변화하지 않는 HTML이나 이미지 등을 가리키고, 동적 콘텐츠는 매 요청마다 내

용이 다른 데이터를 가리킨다. 또한, 데이터 자체가 아니라 동적인 데이터를 출력하는 서버측 프로그램을 일컬어 「동적 콘텐츠」라고 하기도 한다.

패킷(Packet)

주로 IP(Internet Protocol)에서 데이터의 최소단위 묶음을 의미한다. IP패킷이라고도 한다.

페일백(Failback)

Active 노드가 정지한 후 장애극복된 상태에서 원래의 정상상태로 복귀하는 것.

프레임(Frame)

주로 이더넷에서 데이터의 최소단위 묶음을 의미한다. 이더넷 프레임이라고도 한다.

프로덕션 환경(Production Environment)

실제 서비스를 하고 있는 환경

➡ 「스테이징 환경」 참조

확장성(Scalability)

이용자나 규모가 증대됨에 따라 시스템을 확장해서 대응할 수 있는 능력의 정도를 나타낸다.

헬스체크(Health Check)

감시대상이 정상인 상태에 있는지 여부를 확인하는 것. 예를 들면, 웹 서버에 대해 ping이 가는지, TCP 80번 포트로 접속할 수 있는지, HTTP 응답이 있는지 등을 확인하는 것. 대부분의 경우 헬스체크에 실패하면 관리자에게 감시실패 경고가 전달되도록 하고 있다. 사활감시라고도 한다.

CHAPTER

01

서버/인프라 구축 입문

다중화/부하분산의 기본

1.1 다중화의 기본

다중화란

다중화Redundancy란, 장애가 발생해도 예비 운용장비로 시스템의 기능을 계속할 수 있도록 하는 것을 말한다. 예를 들면, 공장이나 병원 등에서는 정전에 대비해서 자가발전장치를 갖추고 있고, 공공 교통기관에서는 만일에 대비해서 여러 브레이크 계통을 갖추고 있는 것을 말한다.

웹서비스를 제공하는 네트워크나 서비스 시스템도 가용성을 확보하기 위해서 다중화하는 것은 새삼스러운 일이 아니다. 이 절에서는 시스템을 다중화하기 위해서 최소한으로 알아두어야 할 것을 설명한 후에 간단한 예를 소개한다.

다중화의 본질

시스템의 다중화란 다음의 단계를 실천하는 것이다.

1. 장애를 상정한다.
2. 장애에 대비해서 예비 운용장비를 준비한다.
3. 장애가 발생했을 때 예비 운용장비로 교체할 수 있는 운용체제를 정비한다.

각 단계를 따라가면서 작업의 흐름을 간단하게 살펴보자.

❶ 장애를 상정한다.

다중화의 첫걸음은 장애를 상정하는 것에서 시작한다. 그 예로 그림 1.1.1과 같이 간단한 구성에서 생각해보자.

우선은, 그림 1.1.1의 시스템에서 발생할 수 있는 장애를 상정해보자.

- 라우터 장애로 서비스가 정지한다.
- 서버 장애로 서비스가 정지한다.

그림 1.1.1에서는 어떤 장비에 장애가 발생하더라도 서비스가 정지해버린다.

❷ 예비 운용장비를 준비한다.

다음으로 장애에 대비해서 예비 운용장비를 도입한다. 그림 1.1.1에 예비 운용장비를 추가한 것이 그림 1.1.2다. 여기서는 아직 예비 라우터와 서버가 네트워크에 연결되어 있지 않다.

❸ 운용체제의 정비 …… 장애발생시 예비 운용장비로 교체한다.

이제 운용체제를 정비한다. 운용체제 정비는 상기 ❶, ❷단계의 어디에 어떻게 장애가 발생할지, 어떤 장비로 어떻게 구성할지에 따라 다양한 대응을 생각해야 한다.

그러면, 먼저 ❶단계에서 상정한 라우터 장애와 웹 서버 장애를 예로 운용체제 정비의 기본과 다중화에서의 기본 용어들을 설명하겠다.

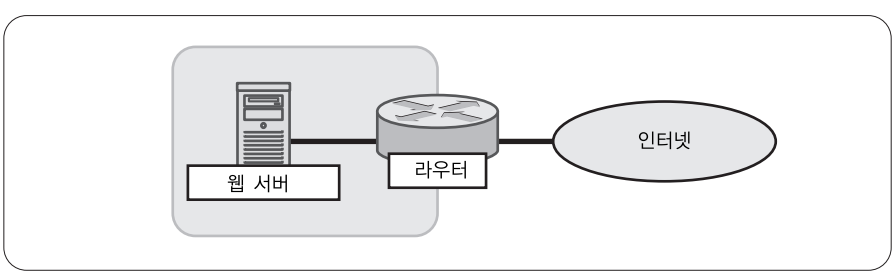

그림 1.1.1 가장 간단한 서버 시스템

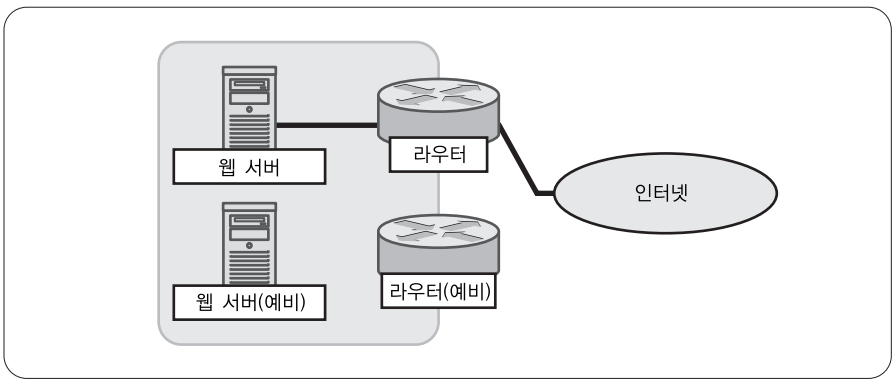

그림 1.1.2 예비 운용장비 도입

라우터 장애시의 대응

그림 1.1.1과 같은 상태에서는 라우터에 장애가 발생하면 서비스가 멈춰버리지만, 그림 1.1.2에서 예비 운용장비를 도입함에 따라 라우터 장애가 생기더라도 그림 1.1.3과 같이 회선을 변경해서 연결하기만 하면 간단히 복구할 수 있게 된다.

Cold Standby

그림 1.1.2 ➡ 그림 1.1.3의 예와 같이, 예비 운용장비는 보통 사용하지 않고 현재 운용장비에 장애가 발생하면 예비 운용장비를 연결하는 운용체제를 「Cold Standby」라고 한다.

여기서 주의해야 할 점은 현재 운용장비와 예비 운용장비의 설정은 동일하게 해두어야 한다는 점이다. 다중화된 시스템에서는 「현재 운용장비와 예비 운용장비의 구성을 항상 같은 상태로 해두는 것이 정석」이다.

라우터와 같은 네트워크 장비라면 운용 중에 빈번히 설정을 변경할 일도 없고, 저장해두어야 할 데이터도 그다지 많지 않으므로 Cold Standby로의 운용은 현실적인 선택방법 중 하나다.

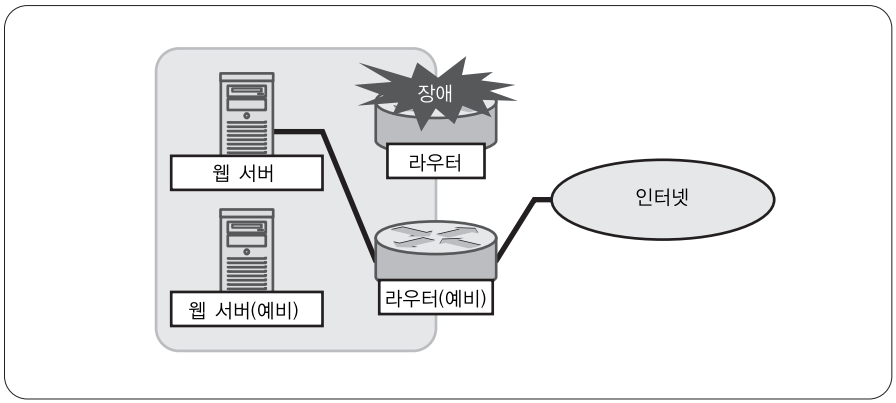

그림 1.1.3 라우터 장애가 발생한 경우의 대응

웹 서버 장애시의 대응

다음으로 웹 서버 장애가 발생했을 때의 대응을 생각해본다. 웹 서버 장애시의 대응으로는 라우터의 경우와 마찬가지로, 그림 1.1.4와 같이 예비 운용장비로 교체하는 방법을 생각해볼 수 있으나, 여기에는 문제가 있다.

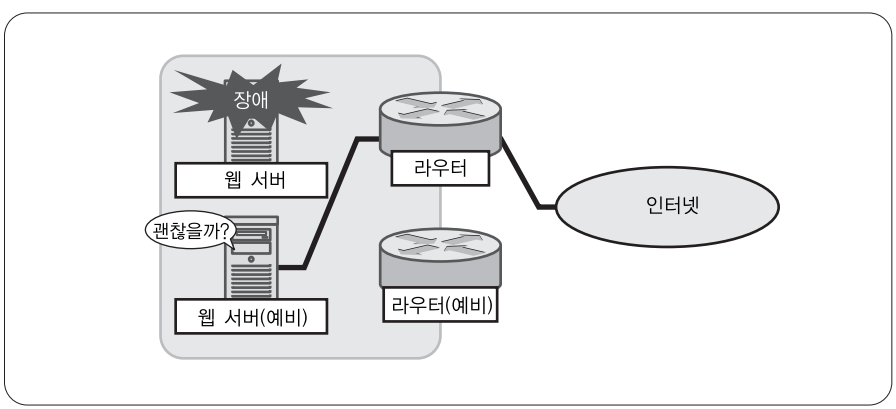

그림 1.1.4 서버 장애시의 대응

Hot Standby

앞서 말한 대로 다중화된 시스템에서는 「현재 운용장비와 예비 운용장비의 구성을 항상 같은 상태로 해두는 것이 정석」이다.

웹 서버의 경우, 사이트의 내용은 매일 갱신될 것이고 애플리케이션이나 운영체제의 버전업 등도 빼놓을 수 없을 것이다. 이렇게 다양한 갱신작업을 보통 중지되어 있는 예비 운용장비에 계속 수행하는 것은 실제로 매우 곤란하다. 만일의 경우에 예비 운용장비를 기동했을 때 컨텐츠의 내용이 오래됐거나 애플리케이션의 버전이 이전 버전이라면 큰 문제가 될 수 있다.

따라서, 웹 서버의 예비 운용장비는 항상 전원을 켜두고 네트워크에 연결해두는 것이 좋을 것이다. 그리하여 현재 운용장비의 내용을 갱신할 때에는 예비 운용장비에도 동일하게 갱신될 수 있도록 운용한다(그림 1.1.5).

이와 같이, 두 대의 서버를 항상 가동시켜 두고 늘 같은 상태로 유지해두는 운용형태를 「Hot Standby」라고 한다. Cold Standby의 경우는 물리적으로 회선연결을 변경하거나 전원을 투입해야 하기 때문에 장애시의 다운타임이 길어지기 쉽지만, Hot Standby라면 즉시 교체하는 것이 가능하다.

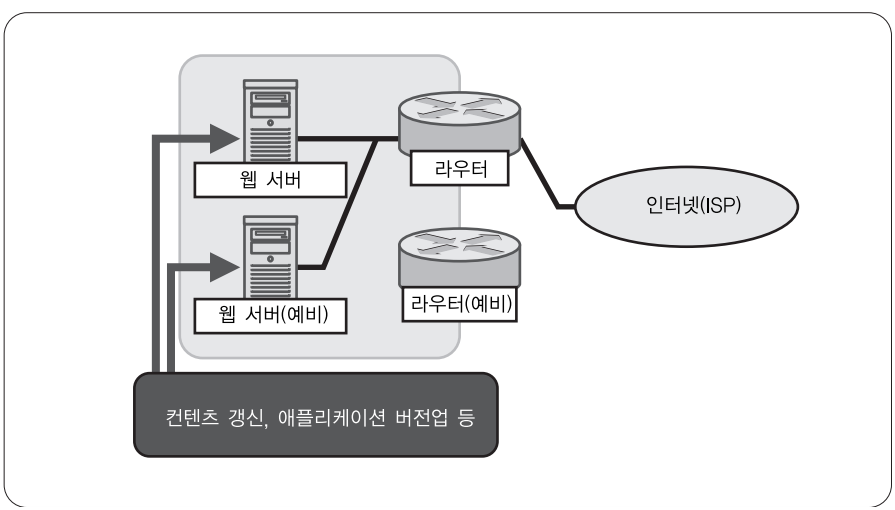

그림 1.1.5 Hot Standby 구성의 운용

장애극복

현재 운용장비에 장애가 발생했을 때 자동적으로 예비 운용장비로 처리를 인계하는 것을 장애극복Failover이라고 한다.

서버를 장애극복하기 위해서는 「가상 IP주소Virtual IP Address」(이하 VIP)와 「IP주소 인계引繼」를 이용한다.

VIP

그림 1.1.6은 VIP를 이용한 Active/Backup 구성의 예다. 현재 운용장비인 Web1에는 자신의 IP주소와는 별개로 「VIP」(10.0.0.1)을 할당해두고 웹서비스는 「VIP」로 제공하도록 한다.

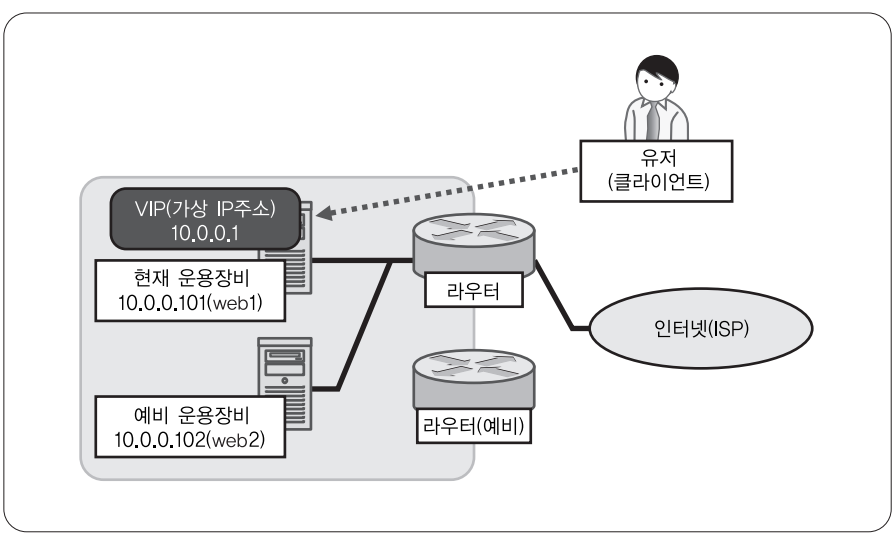

그림 1.1.6 VIP를 이용한 Active/Backup 구성

IP주소 인계

현재 운용장비에 장애가 발생했을 때에는 그림 1.1.7과 같이 예비 운용장비가 VIP를 인계한다. 이에 따라 사용자는 예비 운용장비인 Web2로 접속하게 된다.

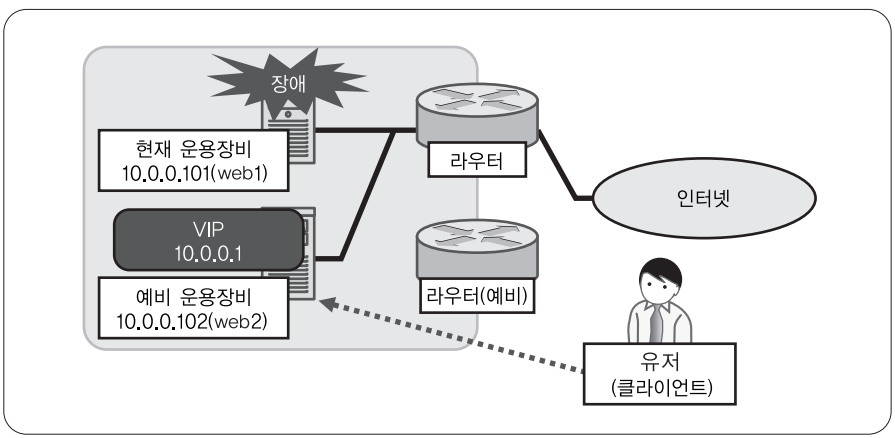

그림 1.1.7 IP주소 인계

장애검출 …… 헬스체크

정상적으로 장애극복하기 위해서는 현재 운용장비에서 장애가 발생하고 있음을 검출하는 방법이 필요하다. 이 방법을 헬스체크Health Check라고 한다. 헬스체크에는 다양한 종류가 있어서 용도에 따라 적절한 것을 선택해야 한다. 주로 이용되는 것은 다음과 같다.

- **ICMP 감시 (Layer 3)**

  ICMP 감시는 ICMP주1의 echo 요청을 보내서 응답이 돌아오는지를 체크한다. 가장 간단하고

주1 Internet Control Message Protocol. 이상발생시에 에러와 에러정보를 통지하는 프로토콜.

가벼운 헬스체크지만, 웹서비스가 다운된 경우(아파치가 중지한 경우 등)은 감지할 수 없다.

- **포트 감시 (Layer 4)**

  포트 감시는 TCP로 접속을 시험해서 접속할 수 있는지 여부를 체크한다. 웹서비스가 다운된 것은 감지할 수 있지만, 과부하 상태로 응답할 수 없다거나 에러를 반환하는 것은 감지할 수 없다.

- **서비스 감시 (Layer 7)**

  실제로 HTTP 요청 등을 보내서 정상적인 응답이 돌아오는지를 체크한다. 대부분의 이상을 감지할 수 있지만 경우에 따라서는 서버에 부하를 유발할 수도 있다.

웹 서버의 헬스체크

이 장의 첫머리에 그림 1.1.1의 구성에서 발생할 수 있는 장애를 두 가지 상정했다.

그 중에 하나인 「서버 장애에 의한 서비스 중지」를 정상적으로 검출하기 위해서는 상기의 헬스체크 방법 중에 「서비스 감시」를 이용한다. 서비스 감시를 이용하는 이유는 서버의 전원이 켜져 있어서 ICMP 응답이 돌아오더라도 웹서비스(아파치 등)가 정상적으로 동작하고 있다고는 할 수 없기 때문이다. 웹 서버의 장애를 검출하기 위해서는 실제로 HTTP로 요청을 보내서 응답이 있는지를 확인하는 것이 가장 확실한 방법이다.

라우터의 헬스체크

「라우터 장애에 의한 서비스 정지」를 검출하기 위해서는 「ICMP 감시」를 이용할 수 있다. 다만, 라우터에 대해 ICMP 감시를 하는 것은 아니다. 여기서 확인하고자 하는 것은 「라우터가 확실히 패킷을 전송할 수 있는가」이므로 인터넷상의 호스트에서 웹 서버에 대해 감시하는 것이 좋을 것이다. 요컨대, 웹 서버가 인터넷과 통신할 수 있는 상태인지를 확인할 수 있으면 된다.

* * *

웹 서버, 라우터에 한하지 않고 헬스체크를 할 때에는 「무엇을 확인하고자 하는가」를 명확히 하는 것이 가장 중요하다.

Active/Backup 구성 만들기

그러면 실제로 셸스크립트를 이용해서 앞에 나온 그림 1.1.6의 구성을 만들어보자. Web1과 Web2에는 자신의 IP주소만을 할당해두자. 리스트 1.1.1은 VIP에 대해 1초마다 ping 검사를 수행해서 실패하면 서버에 VIP를 할당하는 스크립트다.

우선 Web1에 리스트 1.1.1의 스크립트를 실행하기 바란다. 그러면 「fail over!」라는 문자열을 출력하고 종료한다. 이로써 Web1에 VIP가 할당되었다. 다음으로 Web2에 리스트 1.1.1을 실행하기 바란다. 그러면 이번에는 「health ok!」라는 문자열이 1초마다 계속 표시된다.

여기서 클라이언트에서 VIP에 대해 ping 감시를 하면서 Web1을 shutdown해본다. Web1을 shutdown하면 Web2에서 실행 중인 스크립트의 헬스체크가 실패하면서 VIP를 인계한다.

클라이언트의 ping 감시는 그림 1.1.8의 결과대로 Web1을 shutdown한 후 3초 정도 만에 IP주소가 인계되는 모습을 확인할 수 있다.

리스트 1.1.1 failover.sh

```
#!/bin/sh
VIP="10.0.0.1"
DEV="eth0"

healthcheck() {
  ping -c 1 -w 1 $VIP >/dev/null
  return $?
}

ip_takeover() {
  MAC=`ip link show $DEV | egrep -o '([0-9a-f]{2}:){5}[0-9a-f]{2}' |
head -n 1 | tr -d :`
  ip addr add $VIP/24 dev $DEV
  send_arp $VIP $MAC 255.255.255.255 ffffffffffff ←❶
}

while healthcheck; do
```

```
  echo "health ok!"
  sleep 1
done
echo "fail over!"
ip_takeover
```

※ Debian/GNU Linux 4.0, bash 3.1에서 동작확인

그림 1.1.8 장애극복의 동작확인

```
~$ ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=2.46 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=1.86 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=5.06 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=2.64 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.453 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=3.73 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=3.91 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.418 ms ←Web1을 shutdown
64 bytes from 10.0.0.1: icmp_seq=11 ttl=64 time=3.20 ms ←Web2로의 인계 완료
64 bytes from 10.0.0.1: icmp_seq=12 ttl=64 time=1.69 ms
64 bytes from 10.0.0.1: icmp_seq=13 ttl=64 time=1.48 ms
```

IP주소를 인계하는 원리

「IP주소 인계」란 단순히 「IP주소를 바꿔서 설정하는 것뿐」만이 아니다. 시험 삼아 두 대의 서버에 같은 IP주소를 할당하고 다른 머신에서 ping을 계속 날리면서 LAN케이블을 차례로 빼고 꽂아본다. 그러면 아무리 케이블을 바꿔 꽂아도 어느 한 서버에만 ping이 전송됨을 확인할 수 있다.

LAN(Ethernet)의 세계에서는 IP주소가 아닌 NIC(Network Interface Card)에 고정적으로 할당되어 있는 MAC주소(Media Access Control Address)를 사용해서 통신한다. 다른 서버에 패킷을 보낼 때에는 MAC주소를 얻기 위해 ARP(Address Resolution Protocol)라는 프로토콜을 이용한다.

ARP는 IP주소를 지정해서 MAC주소를 조회하기 위한 프로토콜이다. 그러나 통신할 때마다 조회를 하는 것은 효율이 좋지 않으므로, 한번 얻은 MAC주소는 ARP테이블에 저장해서 일정시간 캐싱한다. 따라서, 다른 서버에 같은 IP주소가 할당되더라도 ARP테이블이 갱신될 때까지는 그 서버와 통신할 수 없다. 즉, IP주소를 인계하기 위해서는 다른 서버의 ARP테이블을 갱신해주어야만 한다.

그 방법으로 gratuitous ARP[GARP]가 있다. 통상의 ARP 요청은 「이 IP주소에 대응하는 MAC주소를 알려주기 바람」과 같이 질의하기 위한 것이지만, gratuitous ARP는 「내 IP주소와 MAC주소는 이것이다」라고 다른 서버에 통지하기 위한 것이다. 리스트 1.1.1(failover.sh)에서는 ①에서 send_arp 명령을 사용해서 gratuitous ARP를 전송하고 있다.

서버를 효과적으로 활용하자 …… 부하분산

이상과 같은 Active/Backup 구성에서는 현재 운용장비만 접속을 처리하고 있고 예비 운용장비는 얌전히 대기하고 있지만, 잘 생각해보면 아까운 상황이다. 두 대의 서버를 이용해 서비스를 제공하는 것이 가능하다면 사이트 전체의 처리성능은 배가 될 것이기 때문이다.

여러 대의 서버에 처리를 분산시켜 사이트 전체의 확장성[scalability]을 향상시키는 방법을 부하분산[Load Balance](로드밸런스)이라고 한다. 웹 서버를 부하분산 구성으로 하면 앞으로 접속수가 늘어나서 서버의 처리가 따라가지 못하더라도 서버를 증설함으로써 대응할 수 있게 된다. 고성능인 서버를 구입해서 교체할 필요가 없으므로 낡은 서버가 남아돌거나 쓸모없게 될 일이 없게 된다.

이어서 1.2절, 1.3절에서는 웹 서버를 부하분산하는 구체적인 구축사례를 소개하도록 한다.

CHAPTER

02

한 단계 높은 서버/인프라 구축

다중화, 부하분산, 고성능 추구

2.1 리버스 프록시 도입 아파치 모듈

리버스 프록시 입문

1장에서의 로드밸런서 도입으로 웹 서버의 부하분산은 가능해졌지만, IPVS(LVS)와 같은 로드밸런서는 L4 레벨에서 패킷을 전송하기만 한다. 웹 서버가 클라이언트 애플리케이션으로부터의 요청에 직접 응답하는 구성이라는 점은 변함이 없다.

여기서 로드밸런서와 웹 서버 사이에 리버스 프록시Reverse Proxy라고 하는 역할의 서버를 넣음으로써 보다 유연하게 부하를 분산할 수 있게 된다. 리버스 프록시는 아파치에 mod_proxy나 mod_proxy_balancer를 내장함으로써 구축할 수 있다. 아파치 이외에도 lighttpd나 Squid(이후에 모두 설명) 등으로도 이용 가능하다.

리버스 프록시는 클라이언트로부터의 요청을 받아서(필요하다면 주위에서 처리한 후) 적절한 웹 서버로 요청을 전송한다. 웹 서버는 요청을 받아서 평소처럼 처리를 하지만, 응답은 클라이언트로 보내지 않고 리버스 프록시로 반환한다. 요청을 받은 리버스 프록시는 그 응답을 클라이언트로 반환한다.

그림 2.1.1과 같이 클라이언트와 웹 서버 사이에 놓여서 요청을 대리로 처리하는 것이 리버스 프록시의 역할이다. 통상 프록시 서버는 LAN ➡ WAN의 요청을 대리로 수행하지만, 리버스 프록시는 WAN ➡ LAN의 요청을 대리한다. 이로 인해 「리버스Reverse」라는 이름이 되었다.

리버스 프록시를 이용하면 클라이언트로부터의 요청이 웹 서버로 전달되는 도중의 처리에 끼어들어서 다양한 전후 처리를 시행할 수가 있게 된다. 이것이 리버스 프록시 도입의 장점이다. 보다 구체적인 이점/기능에는 다음을 들 수 있다.

- HTTP 요청의 내용에 따라 시스템의 동작 제어(L7 스위치가 하는 역할과 비슷하다)
- 시스템 전체의 메모리 사용효율 향상
- 웹 서버가 응답하는 데이터의 버퍼링 역할
- 아파치 모듈을 이용한 처리의 제어

다음 절부터 이 내용들에 대해 설명할 것이다.

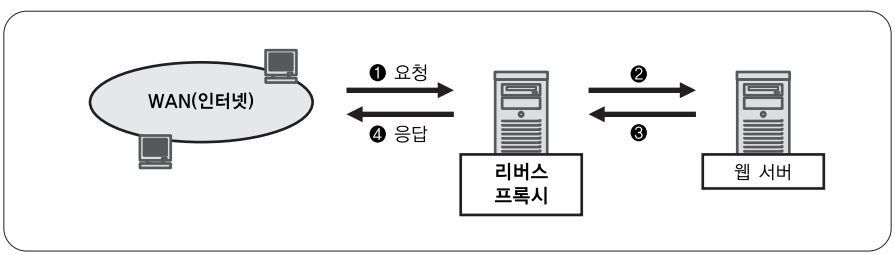

그림 2.1.1 리버스 프록시

HTTP 요청 내용에 따른 시스템의 동작 제어

IPVS는 L4이므로 클라이언트로부터 요구된 HTTP 요청의 내용에 따라 처리를 분배하는 일은 할 수 없다. 여기에 리버스 프록시가 있으면, 예를 들어 HTTP 요청 내에서 URL을 보고,

- 클라이언트로부터 요구된 URL이 /images/logo.jpg이면 이미지용 웹 서버로
- 클라이언트로부터 요구된 URL이 /news이면 동적 컨텐츠를 생성하는 웹 서버로

최종적인 처리를 각기 다른 서버에 분배하는 제어가 가능하다(그림 2.1.2).

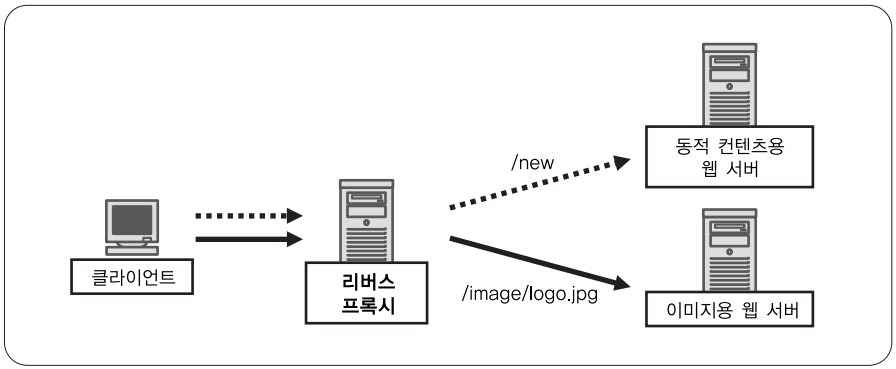

그림 2.1.2 리버스 프록시에 의한 분배

아파치로 리버스 프록시를 구축할 경우, 이러한 분배는 mod_rewrite의 RewriteRule 기능을 이용하게 된다. mod_rewrite로 제어할 수 있다면 거의 무엇이든 가능하다고도 할 수 있다. 예를 들면, 다음과 같은 처리가 가능하다.

- 클라이언트의 IP주소를 보고 특정 IP주소만 서버로의 접속을 허가한다.
- 클라이언트의 User-Agent를 보고 임의의 User-Agent로부터의 요청을 특별한 웹 서버로 접속되도록 유도한다.
- /hoge/foo/bar라는 URL을 /hoge?foo=bar라는 URL로 변경해서 웹 서버로 요청한다.

각각의 예가 어떤 경우에 유효한지 좀 더 생각해보자.

IP주소를 이용한 제어

예를 들어 IP주소에 따른 제어는 악의가 있는 호스트로부터의 요청을 차단할 목적에도 이용할수 있다. 또한, 관리자 전용 페이지가 포함된 사이트에서 IP주소와 URL에 의한 제어를 조합해서 관리자 전용 페이지에는 특정 IP주소에서만 접속 가능하도록 제한할 수도 있다.

User-Agent에 의한 제어

User-Agent에 의한 제어는 Googlebot이나 Yahoo! Slurp 등의 검색엔진 로봇에의 대응에 이용할 수 있다.

예를 들면, 사용자에 대해서는 캐싱하기가 어려운 동적인 페이지(사용자에 맞게 사용자명이 표시되는 페이지 등)가 있다고 하자. 로봇에게는 사용자명을 표시할 필요가 없을 경우, 그 페이지를 캐싱할 수가 있다. 여기서 User-Agent를 보고 로봇의 User-Agent인 경우는 캐시서버를 경유해서 웹 서버로 접속되도록 제어하는 일이 가능하다.

URL 다시쓰기

요즘에는 사이트 전체의 계층구조를 쉽게 알 수 있게 하려는 등의 이유로 사용자에게 웹사이트의 URL을 깔끔하게 보이고자 할 경우도 있다. 「쿨한 URL」[주1]를 실현하려면 본래 웹 애플리케이션 측에서 처리를 해야 하지만, 레거시 시스템을 반드시 이용해야 하는 경우가 있다. 그럴 때에는 리버스 프록시로 요청 URL을 분해해서 레거시 시스템이 이해할 수 있는 URL로 변경해서 웹 서버로 전송하는 것도 하나의 방법이다.

시스템 전체의 메모리 사용효율 향상

동적 컨텐츠를 반환하는 웹 서버(AP서버라고도 함)에는 통상 애플리케이션이 이용하는 프로그램을 메모리에 상주시킴으로써, 애플리케이션 기동시의 오버헤드를

주1 예를 들면, http://b.hatena.ne.jp/bookmark.cgi?user=naoya&tag1=perl&tag2=2=cpan이라는 URL보다도 http://b.hatena.ne.jp/naoya/perl/cpan이라고 하는 편이 쉽고 깔끔하다. 자세한 것은 다음을 참조하기 바란다.

URL http://www.w3.org/Provider/Style/URI.html

회피할 수 있게 설계되어 있다. 예를 들면, 자바Java로 작성된 프로그램은 기동할 때 상당한 시간이 소요되지만, 한번 메모리에 상주되면 이후는 기동시간을 줄여서 동작시킬 수 있다. mod_perl이나 mod_php로 펄perl이나 PHP를 웹 서버에 내장해서 이용하면 애플리케이션의 처리가 고속화되는 것도 같은 원리다. 또한 FastCGI도 거의 마찬가지 원리로 애플리케이션을 고속화시킨다.

AP서버는 이런 상황에서 대량의 메모리를 필요로 한다. 정적 컨텐츠만을 반환하는 웹 서버에 비해 동적 컨텐츠를 반환하는 AP서버에서는 수 배에서 수십 배의 메모리를 소비하는 것도 드물지 않다.

통상 AP서버는 클라이언트의 하나의 요청에 대해 하나의 프로세스 또는 하나의 쓰레드를 할당해서 처리하는 방식을 취하고 있다. 각각의 프로세스/쓰레드는 다른 프로세스/쓰레드와는 독립적으로 동작한다. 이로 인해 애플리케이션 개발자는 리소스 경합을 신경쓰지 않고 프로그램을 개발할 수 있으므로, 애플리케이션 설계가 쉽고 편해진다는 장점을 얻을 수 있다.

그러나 AP서버가 하나의 요청에 대해 하나의 프로세스/쓰레드로 응답할 경우, 이미지나 자바스크립트JavaScript, CSS와 같은 정적 컨텐츠를 반환하는, 즉 파일에 쓰인 내용을 그대로 반환하기만 하면 될 경우도 동일한 방식으로 반환하게 된다.

예 : 동적 페이지에서의 요청의 상세

예를 들면, 동적으로 생성된 하나의 HTML 페이지 내에 이미지가 30개 정도 사용되고 있는 경우를 생각해보자. 가령, Hatena의 메인페이지주2와 같은 페이지다. 이 페이지는 동적으로 생성되고 있다.

이 페이지에 대한 요청은 첫 요청 한 번만 동적 컨텐츠를 요구하게 된다. 첫 요청으로 HTML이 동적으로 생성되고 이 HTML은 클라이언트인 브라우저에 의해 다운로드된다. 다음으로 브라우저는 HTML을 해석해서 필요한 이미지 파일이나 스크

주2 URL http://www.hatena.ne.jp/

립트 파일을 서버에 요청한다. 결과적으로 동적 요청 1회 + 정적 요청 30회가 된다.

모두 AP서버에서 응답할 경우

위 1 + 30회의 요청을 모두 AP서버에서 응답할 경우, 전체적으로는 정적 컨텐츠를 반환하는 게 주된 일이지만 단 1회 동적 요청을 처리할 뿐임에도 나머지 30회의 정적인 요청에 대해 응답할 때에도 대량으로 메모리를 소비하게 된다. 이는 이미지든 동적 컨텐츠든 똑같이 하나의 요청에 대해 하나의 프로세스/쓰레드로 응답할 필요가 있기 때문이다(그림 2.1.3).

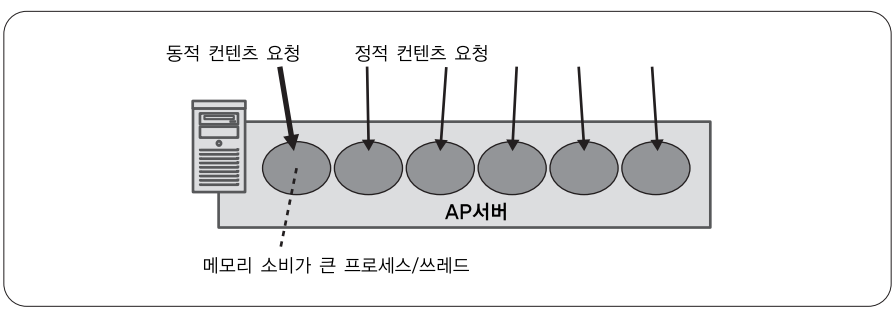

그림 2.1.3 모두 AP서버에서 응답할 경우

서버를 분할할 경우

그러면 정적인 파일을 반환하는 웹 서버와 동적 컨텐츠를 생성하는 AP서버를 각기 다른 서버로 나눠보자(그림 2.1.4). 이렇게 하면 정적 컨텐츠는 메모리 소비량이 적은 웹 서버가 응답하고 동적 컨텐츠만 애플리케이션으로 응답하는 형태의 구성이 가능해진다. 시스템 전체를 보면 메모리 사용효율이 높아져 동시에 처리할 수 있는 요청수가 향상된다.

서버를 2대로 분할하는 것이 좋다고 할 때, 어떻게 정적 컨텐츠, 동적 컨텐츠에 대한 요청을 각각의 서버로 할당할 수 있을까? 바로 이 때문에 리버스 프록시가 필요한 것이다.

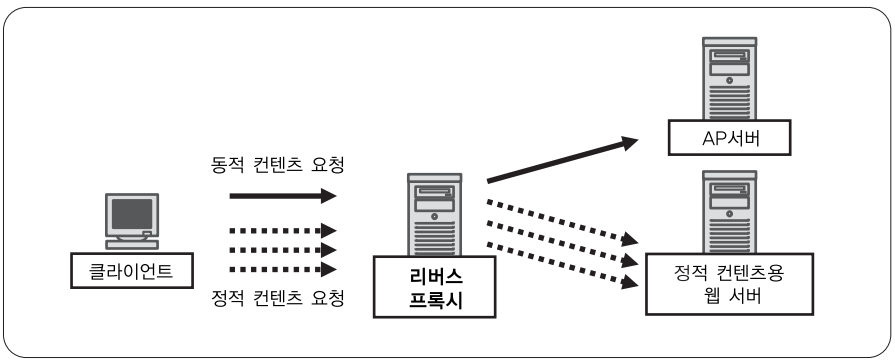

그림 2.1.4 서버를 둘로 분할한 경우

- 요청된 URL이 /images 이하이거나 CSS와 같이 정적 컨텐츠를 대비한 경로 이하인 경우는 웹 서버로
- 그 밖의 URL인 경우는 동적 컨텐츠 요구이므로 AP서버로

이와 같이 URL의 내용을 보고 요청을 할당할 곳을 변경한다. 리버스 프록시의 이 같은 동작은 L7 스위치에 상당하는 처리를 수행하는 것으로 볼 수 있다.

이 때, 리버스 프록시 자신도 웹 서버라는 특징을 살려서(정적 컨텐츠를 반환하기 위해 웹 서버를 별도로 준비하는 것이 아니라) 정적 컨텐츠는 리버스 프록시 자신이 반환하는 형태의 구성이 일반적이다(그림 2.1.5).

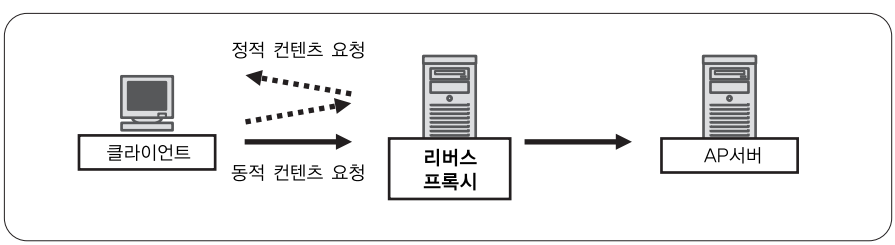

그림 2.1.5 일반적인 구성

웹 서버가 응답하는 데이터의 버퍼링의 역할

리버스 프록시는 AP서버의 바로 앞에 위치해서 AP서버의 버퍼로서의 역할을 수행한다는 점도 중요하다. 특히 HTTP의 Keep-Alive기능을 이용하고자 할 경우에 이 점에서 리버스 프록시의 존재가 중요해지게 되었다.

HTTP의 Keep-Alive

HTTP에는 Keep-Alive라는 사양이 있다. 특정 클라이언트가 한 번에 다수의 컨텐츠를 동일한 웹 서버로부터 얻고자 할 경우(예를 들면, 앞서 본 30개의 이미지가 이용되고 있는 HTML페이지가 좋은 예다), 다수의 HTTP요청마다 서버와 접속하고 끊고를 반복하는 것은 비효율적이다. 최초 요청시에 연결된 서버와의 접속을 해당 요청이 종료한 후에도 접속을 끊지 않고 유지한 채로 이어지는 요청에 해당 접속을 계속 사용함으로써, 하나의 접속으로 다수의 요청을 처리할 수 있다.

이를 실현하는 것이 Keep-Alive다. 서버측이 「Keep-Alive OK」라는 지시를 HTTP헤더로 브라우저에 알리면, 브라우저는 서버와의 접속을 계속 유지해서 Keep-Alive 사양을 따라 하나의 접속으로 여러 파일을 다운로드한다. 실제, Keep-Ailve가 OFF인 서버보다 Keep-Alive가 유효한 서버로부터 파일을 다운로드하는 것이 체감속도면에서도 빠르게 느껴진다.

Keep-Alive는 한번 연결된 접속을 당분간 유지하는 특성상, 웹 서버에 다소 부하를 야기한다. 구체적으로는 특정 클라이언트로부터 요청을 받은 프로세스/쓰레드는 그 시점으로부터 일정 시간 동안 해당 클라이언트로의 응답을 위해서 점유되는 것을 들 수 있다[주3].

주3 lighttpd와 같은 이벤트 모델을 채용하고 있는 웹 서버는 해당되지 않는다.

예 : 메모리 소비와 Keep-Alive의 ON/OFF

메모리 소비 관점에서 이 상황을 생각해보자. 하나의 프로세스당 메모리 소비량이 많은 AP서버에서는 하나의 호스트 내에 실행될 수 있는 최대 프로세스 수는 많아야 50 ~ 100개 정도다. 이 때 리버스 프록시 없이 Keep-Alive를 유효화한 경우 50 ~ 100개의 프로세스 중 다수가 Keep-Alive의 접속 유지를 위해 소비된다(그림 2.1.6).

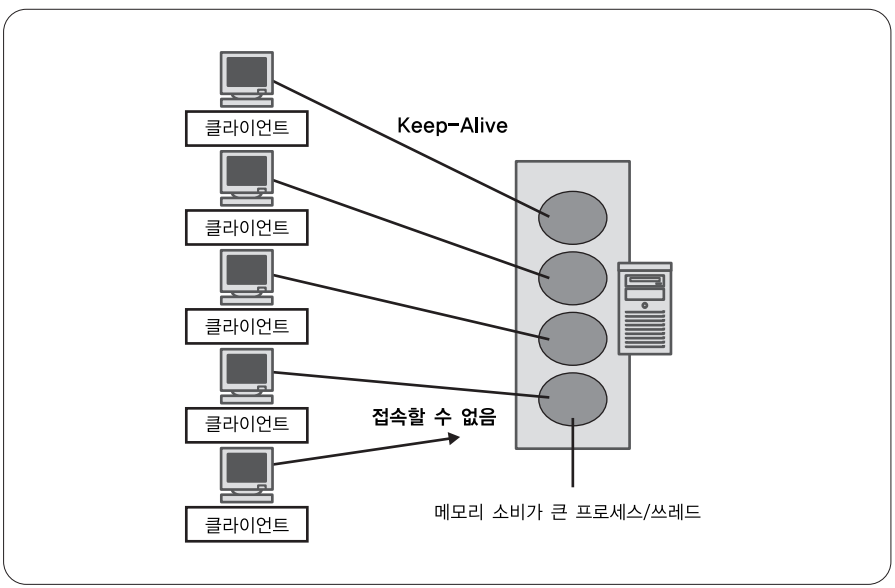

그림 2.1.6 프로세스가 Keep-Alive인 접속 유지를 위해 소비됨

그렇다면 Keep-Alive를 OFF로 한다면 어떨까? 이 경우는 클라이언트에서 볼 때의 체감속도가 저하된다. 이는 원하는 결과가 아니다.

여기에 리버스 프록시를 도입한 경우를 생각해보자. 일반적으로 리버스 프록시 역할을 하는 웹 서버는 프로세스당 메모리 소비량이 그다지 많지 않으므로 하나의 호스트 내에 1,000 ~ 10,000 프로세스를 실행할 수도 있다. 이 경우에는 일부 프로세스가 Keep-Alive 연결을 유지하기 위해 소비된다고 해도 문제가 되지 않는다.

그리고 클라이언트와 리버스 프록시 사이에만 「Keep-Alive ON」으로 하고, 리버스 프록시와 백엔드인 AP서버 사이는 「Keep-Alive OFF」로 한다(그림 2.1.7).

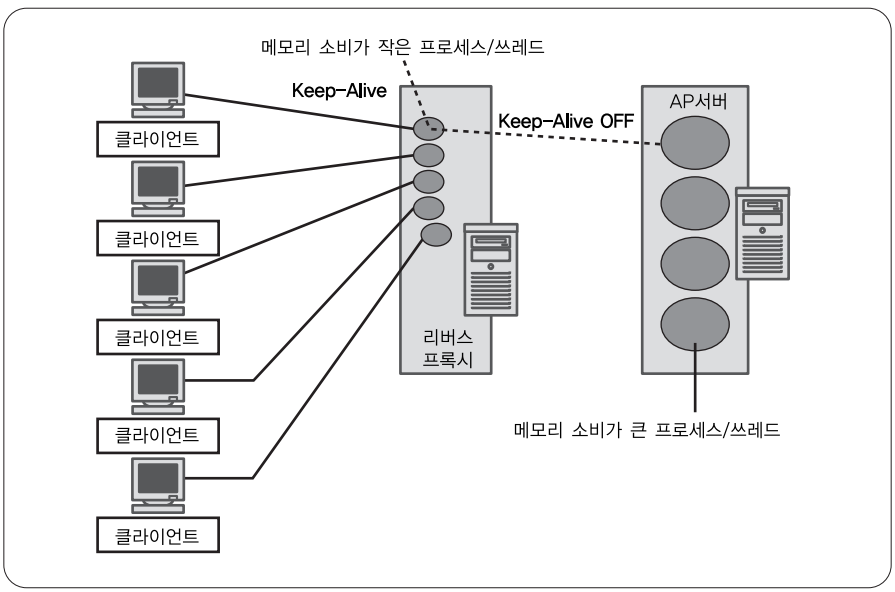

그림 2.1.7 Keep-Alive의 ON/OFF

이렇게 하면 AP서버 측은 프로세스 수가 적더라도 하나의 요청이 종료되면 곧바로 그 후에 다른 요청에 응답할 수 있다. 전체적으로 동시에 다룰 수 있는 클라이언트의 수는 많아지고 또한 전송량도 향상된다. 클라이언트와의 접속 유지를 리버스 프록시가 담당하고 메모리 소비량이 많은 AP서버에서는 그 책무를 지지 않아도 되는, 두 마리 토끼를 잡는 시스템을 구축할 수가 있는 것이다.

아파치 모듈을 이용한 처리의 제어

리버스 프록시로 아파치를 선택한 경우, 해당 리버스 프록시에 아파치 모듈을 내장해서 HTTP 요청의 전처리/후처리로 임의의 프로그램을 실행시킬 수가 있다.

예를 들면, 아파치 2.2에서 소스에 기본 포함되어 있는 mod_deflate는 컨텐츠를 gzip 압축하는 아파치 모듈이다. 이를 리버스 프록시에 내장함으로써 백엔드인 AP서버로부터 수신한 HTTP 응답을 클라이언트에 압축해서 보낼 수가 있다(그림 2.1.8). 마찬가지로 mod_ssl을 이용하면 AP서버로부터의 응답을 SSL로 암호화할 수가 있다.

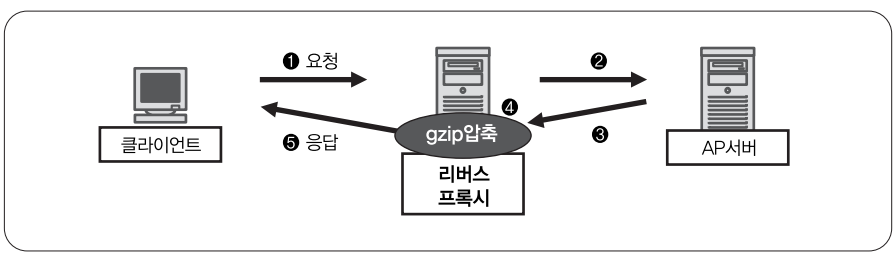

그림 2.1.8 mod_deflate를 내장한 경우

또한, mod_dosdetector[주4]는 아파치 2.2용 Dos공격 대책용 모듈로, 특정 클라이언트로부터 과도한 접속이 있을 경우 일시적으로 차단하는 등의 기능을 하는 모듈이다. 이를 리버스 프록시에 내장함으로써 백엔드인 AP서버가 과도한 접속으로 인해 과부하가 발생하는 것을 막을 수 있다.

아파치 이외에도 lighttpd 등, 써드파티 제품 모듈/플러그인을 내장할 수 있는 웹서버가 몇 종류 있으며, 이를 리버스 프록시로 이용하면 유사한 이점을 얻을 수 있다.

주4 URL http://sourceforge.net/projects/moddosdetector/

리버스 프록시 도입 결정

이와 같이 동적 컨텐츠를 송신하는 AP서버를 이용할 경우, 리버스 프록시의 유무에 따라 시스템의 유연성이 크게 차이가 날 수 있다. 예를 들어, 물리적인 호스트가 한 대밖에 없는 경우에도 동일 호스트 내에 리버스 프록시와 AP서버를 실행해서 「정적 컨텐츠 송신 역할과 백엔드인 AP서버」라는 역할 분담을 분명히 함으로써 서버 리소스의 이용효율을 높일 수 있다.

리버스 프록시를 도입하지 않아야 할 이유는 어디서도 찾을 수가 없는 것이다.

리버스 프록시의 도입

지금부터는 아파치를 이용한 리버스 프록시 구축방법과 각종 설정 예에 대해 설명하도록 한다.

아파치 2.2 사용

리버스 프록시를 구축하기에는 안정버전인 아파치 2.2를 이용하면 좋을 것이다[주5]. 또한 리버스 프록시는 가능하면 다수의 클라이언트를 동시에 처리하는 것이 바람직하므로, 하나의 클라이언트에 대해 하나의 프로세스를 할당하는 prefork 모델보다는 하나의 클라이언트를 하나의 쓰레드로 처리하는 「worker 모델」이 더 효율적이다.

worker로 아파치를 기동

Red Hat Enterprise Linux 5나 CentOS 5에는 표준 패키지로 아파치 2.2가 포함

주5 이 절에서는 CentOS 4.4, 아파치 2.2.4를 사용했다.

되어 있으므로 이를 설치하면 된다. 이 Red Hat에 패키지로 포함된 아파치는 prefork/worker(멀티 프로세스 모델/멀티 프로세스 + 멀티 쓰레드 복합 모델) 중에 사용할 모델은 기동할 때 선택할 수가 있다. /etc/sysconfig/httpd에서,

```
HTTPD=/usr/sbin/httpd.worker
```

라고 하면, httpd를 worker 모델로 기동할 수가 있다.

httpd.conf 설정

아파치를 리버스 프록시로서 작동시키기 위한 최소한의 설정을 나타낸다. 한편, 아파치는 DSO Dynamic Shared Object 가능하도록 컴파일된 것을 사용한다.

최대 프로세스/쓰레드 수 설정

우선은 worker 프로세스, 쓰레드 수에 관한 설정이다.

```
StartServers          2
MaxClients          150
MinSpareThreads      25
MaxSpareThreads      75
THreadsPerChild      25
MaxRequestsPerChild  0
```

디폴트로는 httpd.conf의 Global Directive에 위와 같이 설정되어 있지만, 이는 상당히 줄여서 나타낸 설정이다. 리버스 프록시는 부하가 높을 때에도 백엔드인 AP 서버의 방패 역할을 해주어야 하므로, 좀 더 리소스를 사용해서 처리 가능한 동시접속수를 늘릴 수 있도록 설정하는 편이 좋을 것이다.

위 설정 중에도 중요한 지표는 MaxClients와 ThreadsPerChild다. worker 모델의 경우, 아파치는 여러 자식 프로세스를 실행시키고 각각의 프로세스 내에 여러 쓰레드를 생성해서, 결국 「프로세스 수 × 프로세스 당 쓰레드 수」만큼의 요청을 동시에 처리할 수 있게 된다.

여기서 프로세스 당 쓰레드 수를 제어하는 것이 바로 ThreadsPerChild다. 자식 프로세스의 최대값은,

```
 MaxClients
----------------
 ThreadsPerChild
```

로 결정된다. MaxClients는 동시에 처리할 수 있는 클라이언트의 총수가 된다. 따라서 앞에서 본 설정의 경우는 다음과 같은 설정이 된다.

- **최대 프로세스 수** : 6
- **프로세스 당 최대 쓰레드 수** : 25
- **동시에 처리할 수 있는 클라이언트 수** : 6 × 25 = 150

메모리를 2GB~4GB 정도 탑재하고 있는 서버라면, 동시접속수는 1,000~10,000 정도를 처리하는 것도 가능하다. 예를 들어,

- **최대 프로세스 수** : 32
- **프로세스 당 최대 쓰레드 수** : 128
- **동시에 처리할 수 있는 클라이언트 수** : 32 × 128 = 4096

위와 같이 설정할 경우에는 다음과 같이 하면 된다.

```
StartServers         2
ServerLimit         32  ←신규
ThreadLimit        128  ←신규
MaxClients        4096  ←변경
MinSpareThreads     25
MaxSpareThreads     75
ThreadsPerChild    128  ←변경
MaxRequestsPerChild  0
```

신규로 ServerLimit와 ThreadLimit을 설정했다. ServerLimit/ThreadLimit은 MaxClients나 ThreadsPerChild와 병행해서 프로세스/쓰레드의 최대 생성수를 결정하는 또 다른 설정항목이다. ServerLimit는 디폴트로 16, ThreadLimit는 64로

되어 있으므로 그 이상의 프로세스/쓰레드 수를 설정할 경우에는 이 두 항목을 명시적으로 지정할 필요가 있다.

ServerLimit/ThreadLimit와 메모리의 관계

그런데 MaxClients나 ThreadsPerChild가 프로세스/쓰레드 수의 상한선을 결정하는 것임에도 불구하고 비슷한 파라미터로서 ServerLimit/ThreadLimit 항목이 존재한다는 점에 의구심이 들 것이다. MaxClients나 ThreadsPerChild는 서버의 동적인 리소스 소비에 관련된 설정항목으로, 이 값이 높든 낮든 서버가 최소한으로 소비하는 리소스 소비량에는 영향을 미치지 않는다. 한편으로, ServerLimit/ThreadLimit은 아파치가 확보하는 공유 메모리의 크기에 영향을 준다. 이 값에 필요 이상으로 높은 값을 설정하면 그만큼 아파치는 쓸데없이 공유 메모리를 소모하게 되는 것이다. 따라서, 설정하고자 하는 프로세스/쓰레드 수의 상한선이 내장된 값인 ServerLimit 16, ThreadLimit 64를 초과할 경우에만 그 값에 맞게 설정된다.

한편, 프로세스/쓰레드당 메모리 사용량은 내장된 모듈의 종류 등에 의존한다. 또한, OS가 아파치에 메모리를 얼마나 할당할지는 환경에 따라 다르므로 설정값을 단언할 수는 없다. 위 설정은 어디까지나 참고만 하길 바란다. 실제로 사용 중인 환경하에서 어느 정도로 상한선을 정해야 좋을지는 프로세스/쓰레드 당 메모리 사용량을 고려해서 산정해야 한다. 자세한 것은 4장에서 설명할 것이다.

특정 웹 서버상에서의 최대 프로세스/쓰레드 수는 리소스가 상한선에 이르렀을 때 스왑이 발생하지 않을 정도, 즉,

- OS나 웹 서버 이외의 소프트웨어가 항시 이용하는 메모리량
- 웹 서버의 프로세스/쓰레드 수가 최대수에 이르렀을 때 서버가 소비하는 합계 메모리량

위 두 가지를 합해서 탑재된 물리적 메모리의 범위 내에 들 정도로 튜닝하는 것이 적합하다. 프로세스/쓰레드 당 메모리 사용량의 판별방법에 대해서도 4장에서 자세히 설명할 것이다.

Keep-Alive 설정

앞서 설명한대로 리버스 프록시에서는 Keep-Alive를 ON, AP서버에서는 Keep-Alive를 OFF로 하는 것이 정석이다. 아파치에서 Keep-Alive를 유효하게 하려면 Global Directive에서 다음과 같이 설정한다.

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 5
```

위 설정의 내용은 다음과 같다.

- Keep-Alive를 유효하게
- Keep-Alive 상태로 처리할 수 있는 최대 요청수는 100건
- Keep-Alive 타임아웃(클라이언트와 접속을 계속 유지하는 시간)은 5초

KeepAliveTimeout은 디폴트로는 15초로 되어 있지만, 통상의 웹사이트에서 클라이언트로의 응답은 5초 이내로 이루어진다. 이 값을 크게 하면 그만큼 프로세스/쓰레드가 Keep-Alive를 위해 점유하는 시간이 길어지므로, 서버의 리소스 소비량이 커진다. 디폴트인 15초보다도 작은 값을 설정해도 문제는 없을 것이다.

필요한 모듈 로드

다음으로 설정해야 하는 것은 필요한 모듈을 로드하는 것이다. 리버스 프록시를 구축하기 위해 최소한으로 필요한 모듈은 다음과 같다.

- mod_rewrite
- mod_proxy
- mod_proxy_http

이와 함께 공개 디렉토리의 앨리어스를 정의할 수 있는 「mod_alias」도 유효하게 해두면 편리하다.

```
LoadModule alias_module modules/mod_alias.so
LoadModule rewrite_module modules/mod_rewrite.so
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

이러한 모듈을 로드함에 따라 RewriteRule이나 RewriteRule 내에서 Proxy, Alias 등의 Directive를 사용할 수 있게 된다.

한편, 패키지로 설치된 아파치의 httpd.conf에서는 디폴트로 다수의 모듈이 내장되어 있지만, 필요없는 모듈을 내장하면 그만큼 아파치의 메모리 사용량이 늘어나게 되므로 이용하지 않는 모듈은 최대한 제거해서 가급적 메모리를 절약하는 편이 좋을 것이다.

RewriteRule 설정

ServerRoot나 로그 등의 설정을 마쳤다면[주6], 끝으로 RewriteRule 설정을 한다. 이것이 바로 리버스 프록시 구축시의 핵심이다.

다음과 같은 설정에 대해 생각해보도록 하자.

- /images는 이미지를 전송하는 경로로, 이 URL은 리버스 프록시에서 직접 전송. 한편, 모든 이미지는 리버스 프록시와 동일한 호스트 내의 /path/to/images/ 아래에 두기로 한다.
- /css, /js도 마찬가지
- 그 밖의 URL은 동적 컨텐츠 전송. AP서버 192.168.0.100으로 요청을 프록시한다.

설정은 리스트 2.1.1과 같다.

RewriteRule의 내용에 주목하기 바란다. RewriteRule은 요청된 URL에 패턴매치를 수행해서 매치되면 해당 URL에 임의의 처리를 할 수 있는 Directive다. RewriteRule에서는 정규표현을 이용할 수 있다. 리스트 2.1.1의 ①은 다음과 같은 설정을 의미한다.

주6 아파치의 기본적인 설정에 대해서는 아파치 매뉴얼을 참고하기 바란다.

/images/, /css/, /js/ 중 하나에 URL이 매치될 경우, 별다른 처리 없이([L]은 RewriteRule의 패턴매치를 여기서 마친다는 의미), 디폴트 컨텐츠 핸들러로 컨텐츠를 반환한다.

디폴트 컨텐츠 핸들러는 정적인 파일을 반환하는, URL의 경로에 따라 파일을 찾아서 클라이언트로 반환하는, 아파치의 일반적인 동작을 수행한다.

예를 들어, 클라이언트로부터의 요청이 /images/profile/naoya.png인 경우, 이 설정에 의해 로컬의 /path/to/images/profile/naoya.png가 클라이언트로 반환된다.

이어서, 리스트 2.1.1의 ②는 다음과 같은 내용의 설정이 된다.

모든 URL에 요청을 192.168.0.100으로 프록시한다.

리스트 2.1.1 RewriteRule 설정

```
Listen 80
<VirtualHost *:80>
  ServerName naoya.hatena.ne.jp

  Alias /images/ "/path/to/images/"
  Alias /css/    "/path/to/css/"
  Alias /js/     "/path/to/js/"

  RewriteEngine on
  RewriteRule ^/(images|css|js)/ - [L]  ←①
  RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L]  ←②
</VirtualHost>
```

이로써 설정은 완료되었다. 리버스 프록시가 바인드한 포트로 브라우저를 통해 접속하면 실제 AP서버로부터 응답해오는 것처럼 192.168.0.100이 반환하는 컨텐츠가 표시될 것이다.

진보된 RewriteRule의 설정 예

보다 진보된 RewriteRule의 설정 예를 살펴보도록 하자.

특정 호스트로부터 요청 금지

예를 들면, 특정 IP주소로부터의 요청에 대해 접근금지를 뜻하는 상태코드 403을 반환할 경우는, 리스트 2.1.2와 같이 설정한다.

AP서버로 프록시하기 전에 조건판정을 하는 RewriteCond Directive로 REMOTE_ADDR을 보고 특정 주소인 경우에는 403을 반환[주7]하고 종료한다.

리스트 2.1.2 설정 예 1

```
RewriteEngine on

# 192.168.0.200으로부터의 요청에 대해 403을 반환하고 종료
RewriteCond %{REMOTE_ADDR} ^192\.168\.0\.200$
RewriteCond .* - [F,L]

# 리버스 프록시 설정
RewriteRule ^/(images|css|js)/ - [L]
RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L]
```

로봇으로부터의 요청에 대해 캐시서버 경유

가령, Squid로 구축한 HTTP 캐시서버가 192.168.0.150에 위치해 있다고 하자. 로봇으로부터의 요청, 즉 특정 User-Agent로부터의 요청만 캐싱된 내용을 반환하고자 할 경우는 리스트 2.1.3과 같이 설정한다.

mod_setenvif를 사용해서 User-Agent 문자열로부터 로봇인지 판정해서 로봇

주7 RewriteRule의 플래그인 [F,L]을 사용한다.

으로 판정된 경우에는 캐시서버로 프록시한다.

이와 같이, 아파치의 mod_rewrite는 다른 모듈과 조합해서 유연한 설정이 가능하다는 점이 장점이다. RewriteRule로 작성할 수 있는 조건으로 적합한 경우라면 얼마든지 다른 서버로 요청을 프록시할 수 있다.

리스트 2.1.3 설정 예 2

```
# SetEnvIf Directive를 유효하게 하기 위해 mod_setenvif를 로드
LoadModule setenvif_module modules/mod_setenvif.so

# User-Agent에 "Yahoo! Slurp" 혹은 "Googlebot"이
# 포함된 경우 환경변수 ISRobot을 참으로 설정
SetEnvIf User-Agent "Yahoo! Slurp" IsRobot
SetEnvIf User-Agent "Googlebot" IsRobot

RewriteEngine on

# 환경변수 IsRobot이 참인 경우 캐시서버로 프록시
RewriteCond %{ENV:IsRobot} .+
RewriteCond ^/(.*)$ http://192.168.0.150/$1 [P,L]

# 그 밖에는 통상적으로 프록시
RewriteRule ^/(images|css|js)/ - [L]
RewriteRule ^/(.*)$ http://192.168.0.100/$1 [P,L]
```

mod_proxy_balancer로 여러 호스트로 분산하기

그렇다면 백엔드인 AP서버가 여러 대인 경우 어떻게 구성할지에 대한 의문이 생기게 된다. 여기에는 몇 가지 방법을 생각해볼 수가 있다.

① 리버스 프록시와 AP서버는 항상 일대일로 갖춘다. 하나의 프록시로부터 하나의 AP서버로 요청을 전송한다.

② mod_proxy_balancer를 이용해서 하나의 리버스 프록시로부터 여러 AP서버로 분산한다(그림 2.1.9).

③ 리버스 프록시와 AP서버 사이에 LVS를 넣는다.

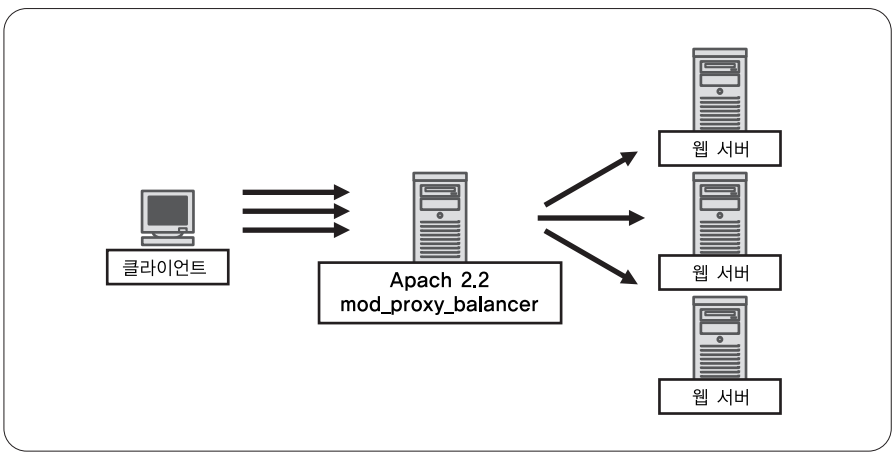

그림 2.1.9 mod_proxy_balancer 이용

이 중에 ①은 그다지 현명한 선택이라 할 수 없다. 리버스 프록시에서는 기본적으로 AP서버보다도 리소스 소비량이 적고, 게다가 리소스 소비량이 적더라도 맡겨진 일을 잘 수행할 것으로 기대된다. 일반적인 시스템이라면 리버스 프록시는 다중화를 고려해서 2대만 있으면 충분하다. 한편, 부하 상황에 따라서는 백엔드인 AP서버는 2대만으로는 충분치 않은 경우도 많다. 예를 들면, Hatena 북마크[주8]는 집필하고 있는 시점(2008년 2월)에서 리버스 프록시 2대에 대해 AP서버가 11대로 구성되어 있다.

리버스 프록시와 AP서버는 리소스 소비가 불균형적이므로 일대일 구성으로는 리버스 프록시측의 리소스가 쓸데없이 남아돌게 된다.

②의 아파치 mod_proxy_balancer는 리버스 프록시를 수행함에 있어서 프록시될 호스트가 여러 대 있는 경우라도 각 호스트로 요청을 분산해서 할당할 수 있도록

주8 URL http://b.hatena.ne.jp/

처리해주는 모듈이다. 또한, 프록시될 호스트가 특정 이유로 응답할 수 없는 경우 장애극복을 해서 분산할 호스트 목록에서 해당 호스트를 분리하고, 해당 호스트가 요청 가능해졌을 때 페일백FailBack하는 기능도 지니고 있다. 이를 이용하는 것도 하나의 방법이다.

또 하나의 방법으로, 리버스 프록시와 AP서버 사이에 LVS + keepalived를 넣는 것이 ③이다. 이것이 가장 확실한 방법이라 할 수 있다. 필자가 개인적으로 이용해 본 느낌으로는, mod_proxy_balancer의 장애극복 기능은 LVS의 그것에 비해 신뢰성은 그다지 높지는 않다고 생각한다. 또한, LVS + keepalived는 mod_proxy_balancer에 비해 부하분산 로직을 조정하기 쉽고 명령줄에서 수행하기 때문에 관리하기도 편리하다.

다만, LVS + keepalived는 준비하기에 약간 수고스럽고 추가적인 서버가 필요하다. 간편하게 부하분산을 수행하고자 할 경우를 고려해서 「mod_proxy_balancer」를 이용하는 방법을 설명하도록 한다.

mod_proxy_balancer 이용 예

mod_proxy_balancer를 이용한 리버스 프록시 구축은 간단하다.

- mod_proxy_balancer를 로드한다[주9].
- BalancerMember Directive로 분산할 호스트 목록을 정의한다.
- RewriteRule로 리버스 프록시를 설정한다. 이 때 balancer:// scheme을 이용한다.

AP서버가 3대, 192.168.0.100~102까지 있다고 하자. 이 경우 httpd.conf 설정은, 예를 들면 리스트 2.1.4와 같이 된다.

리스트 2.1.4의 ①에 주목하기 바란다. 앞서의 예에서는 http://192.168.0.100/$1과 AP서버의 URL을 직접 기술했었지만, 이번에는 「balancer://」라는 scheme의 URL을 사용하고 있다. balancer://backend라고 기술하면, 요청마다 위에 정의되

주9 mod_proxy_balancer는 아파치 2.2에 표준으로 내장되어 있다.

어 있는 BalancerMember 중 한 대가 선택되어 전개된다. 어떤 서버가 전개될지는 BalancerMember에 정의되어 있는 loadfactor값에 의존한다. loadfactor값이 클수록 할당될 확률이 커진다. 리스트 2.1.4와 같이 loadfactor를 모두 동일한 값으로 설정하면 거의 균일하게 요청이 분산되게 된다.

mod_proxy_balancer에는 loadfactor 이외에도 몇몇 파라미터가 있다. 사이트 구성에 맞게 적절하게 설정을 추가하면 좋을 것이다.

리스트 2.1.4 mod_proxy_balancer에 의한 리버스 프록시 구축 설정 예

```
# mod_proxy_balancer 로드
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so

# 프록시될 호스트 목록 정의
<Proxy balancer://backend>
  BalancerMember http://192.168.0.100 loadfactor=10
  BalancerMember http://192.168.0.101 loadfactor=10
  BalancerMember http://192.168.0.102 loadfactor=10
</Proxy>

Listen 80
<VirtualHost *:80>
  ServerName naoya.hatena.ne.jp

  Alias /images/ "/path/to/images/"
  Alias /css/    "/path/to/css/"
  Alias /js/     "/path/to/js/"

  # 리버스 프록시 설정
  RewriteEngine on
  RewriteRule ^/(images|css|js)/ - [L]
  RewriteRule ^/(.*)$ balancer://backend/$1 [P,L]  ←①
</VirtualHost>
```