

옴긴이의 글 .....	ix
지은이의 글 .....	x

---

## 1장 사용자 수에 따른 규모 확장성 1

---

단일 서버 .....	1
데이터베이스 .....	4
수직적 규모 확장 vs 수평적 규모 확장 .....	5
캐시 .....	11
콘텐츠 전송 네트워크(CDN) .....	13
무상태(stateless) 웹 계층 .....	17
데이터 센터 .....	20
메시지 큐 .....	22
로그, 메트릭 그리고 자동화 .....	23
데이터베이스의 규모 확장 .....	25
백만 사용자, 그리고 그 이상 .....	29
참고문헌 .....	30

---

## 2장 개략적인 규모 추정 33

---

2의 제곱수 .....	33
모든 프로그래머가 알아야 하는 응답지연 값 .....	34
가용성에 관계된 수치들 .....	36
예제: 트위터 QPS와 저장소 요구량 추정 .....	36
팁 .....	37
참고문헌 .....	38

3장	시스템 설계 면접 공략법	39
	효과적 면접을 위한 4단계 접근법 .....	41
4장	처리율 제한 장치의 설계	51
	1단계 문제 이해 및 설계 범위 확정 .....	52
	2단계 개략적 설계안 제시 및 동의 구하기 .....	53
	3단계 상세 설계 .....	67
	4단계 마무리 .....	73
	참고문헌 .....	74
5장	안정 해시 설계	77
	해시 키 재배치(rehash) 문제 .....	77
	안정 해시 .....	79
	마치며 .....	88
	참고문헌 .....	89
6장	키-값 저장소 설계	91
	문제 이해 및 설계 범위 확정 .....	92
	단일 서버 키-값 저장소 .....	92
	분산 키-값 저장소 .....	93
	요약 .....	114
	참고문헌 .....	114
7장	분산 시스템을 위한 유일 ID 생성기 설계	117
	1단계 문제 이해 및 설계 범위 확정 .....	117
	2단계 개략적 설계안 제시 및 동의 구하기 .....	118
	3단계 상세 설계 .....	122
	4단계 마무리 .....	124
	참고문헌 .....	125

8장	URL 단축기 설계	127
	1단계 문제 이해 및 설계 범위 확정	127
	2단계 개략적 설계안 제시 및 동의 구하기	128
	3단계 상세 설계	132
	4단계 마무리	138
	참고문헌	139
9장	웹 크롤러 설계	141
	1단계 문제 이해 및 설계 범위 확정	143
	2단계 개략적 설계안 제시 및 동의 구하기	145
	3단계 상세 설계	150
	4단계 마무리	161
	참고문헌	162
10장	알림 시스템 설계	165
	1단계 문제 이해 및 설계 범위 확정	166
	2단계 개략적 설계안 제시 및 동의 구하기	166
	3단계 상세 설계	174
	4단계 마무리	180
	참고문헌	181
11장	뉴스 피드 시스템 설계	183
	1단계 문제 이해 및 설계 범위 확정	184
	2단계 개략적 설계안 제시 및 동의 구하기	184
	3단계 상세 설계	188
	4단계 마무리	195
	참고문헌	196

12장	채팅 시스템 설계	197
	1단계 문제 이해 및 설계 범위 확정	197
	2단계 개략적 설계안 제시 및 동의 구하기	199
	3단계 상세 설계	210
	4단계 마무리	219
	참고문헌	220
13장	검색어 자동완성 시스템	223
	1단계 문제 이해 및 설계 범위 확정	224
	2단계 개략적 설계안 제시 및 동의 구하기	226
	3단계 상세 설계	228
	4단계 마무리	243
	참고문헌	244
14장	유튜브 설계	247
	1단계 문제 이해 및 설계 범위 확정	248
	2단계 개략적 설계안 제시 및 동의 구하기	251
	3단계 상세 설계	258
	4단계 마무리	273
	참고문헌	274
15장	구글 드라이브 설계	277
	1단계 문제 이해 및 설계 범위 확정	278
	2단계 개략적 설계안 제시 및 동의 구하기	280
	3단계 상세 설계	288
	4단계 마무리	298
	참고문헌	299

16장 배움은 계속된다 301

---

실세계 시스템들 ..... 301

회사별 엔지니어링 블로그 ..... 305

후기 ..... 307

# 1장

S y s t e m   D e s i g n   I n t e r v i e w

## 사용자 수에 따른 규모 확장성

수백만 사용자를 지원하는 시스템을 설계하는 것은 도전적인 과제이며, 지속적인 계량과 끝없는 개선이 요구되는 여정이다. 이번 장에서는 한 명의 사용자를 지원하는 시스템에서 시작하여, 최종적으로는 몇백만 사용자를 지원하는 시스템을 설계해 볼 것이다. 이번 장을 통해, 규모 확장성과 관계된 설계 문제를 푸는 데 쓰일 유용한 지식들을 마스터할 수 있을 것이다.

### 단일 서버

천 리 길도 한 걸음부터라는 말이 있듯, 복잡한 시스템을 만드는 것도 그와 크게 다르지 않다. 모든 컴포넌트가 단 한 대의 서버에서 실행되는 간단한 시스템부터 설계해 보자. 그림 1-1은 이 구성의 실제 사례다. 웹 앱, 데이터베이스, 캐시 등이 전부 서버 한 대에서 실행된다.

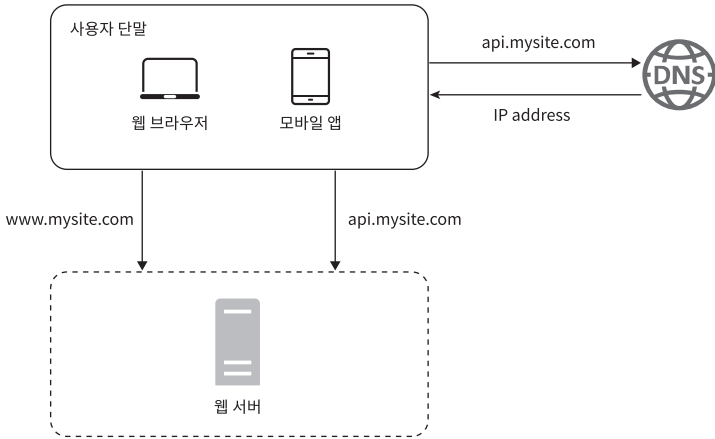


그림 1-1

이 그림의 시스템 구성을 이해하기 위해서는 사용자의 요청이 처리되는 과정과 요청을 만드는 단말에 대해서 이해할 필요가 있다. 사용자 요청 처리 흐름부터 살펴보자(그림 1-2).

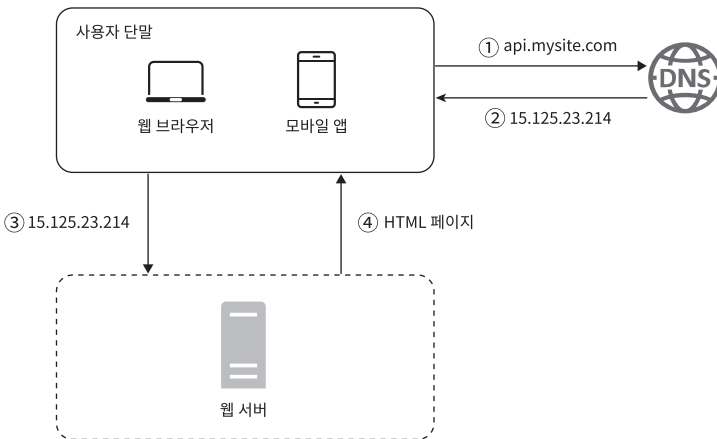


그림 1-2

1. 사용자는 도메인 이름(api.mysite.com)을 이용하여 웹사이트에 접속한다. 이 접속을 위해서는 도메인 이름을 도메인 이름 서비스(Domain Name Service, DNS)에 질의하여 IP 주소로 변환하는 과정이 필요하다. DNS는 보

통 제3 사업자(third party)가 제공하는 유료 서비스를 이용하게 되므로, 우리 시스템의 일부는 아니다.

2. DNS 조회 결과로 IP 주소가 반환된다. 여기 예제에서는 그 주소가 15.125.23.214라고 하겠다. 이 주소는 그림 1-2에 나온 웹 서버의 주소이다.
3. 해당 IP 주소로 HTTP(HyperText Transfer Protocol) 요청이 전달된다.
4. 요청을 받은 웹 서버는 HTML 페이지나 JSON 형태의 응답을 반환한다.

이제 실제 요청이 어디로부터 오는지를 살펴보자. 이 요청들은 두 가지 종류의 단말로부터 오는데, 하나는 웹 앱이고 다른 하나는 모바일 앱이다.

- 웹 애플리케이션: 비즈니스 로직, 데이터 저장 등을 처리하기 위해서는 서버 구현용 언어(자바, 파이썬 등)를 사용하고, 프레젠테이션용으로는 클라이언트 구현용 언어(HTML, 자바스크립트 등)를 사용한다.
- 모바일 앱: 모바일 앱과 웹 서버 간 통신을 위해서는 HTTP 프로토콜<sup>[1]</sup>을 이용한다. HTTP 프로토콜을 통해서 반환될 응답 데이터의 포맷으로는 보통 JSON(JavaScript Object Notation)이 그 간결함 덕에 널리 쓰인다. 아래는 그 예제다.

*GET /users/12 - id가 12인 사용자 데이터 접근*

```
{
  "id": 12,
  "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumbers": [
    "212 555-1234",
    "646 555-4567"
  ]
}
```



## 데이터베이스

사용자가 늘면 서버 하나로는 충분하지 않아서 여러 서버를 두어야 한다. 하나는 웹/모바일 트래픽 처리 용도로, 다른 하나는 데이터베이스용이다(그림 1-3). 웹/모바일 트래픽 처리 서버(웹 계층)와 데이터베이스 서버(데이터 계층)를 분리하면 그 각각을 독립적으로 확장해 나갈 수 있게 된다.

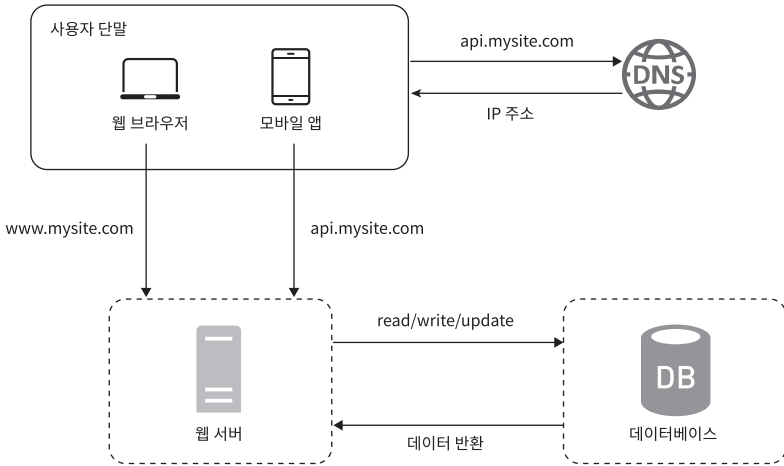


그림 1-3

### 어떤 데이터베이스를 사용할 것인가?

전통적인 관계형 데이터베이스(relational database)와 비-관계형 데이터베이스 사이에서 고를 수 있다. 그 차이를 알아보자.

관계형 데이터베이스는 관계형 데이터베이스 관리 시스템(Relational Database Management System, RDBMS)이라고도 부른다. RDBMS 가운데 가장 유명한 것으로는 MySQL, 오라클 데이터베이스, PostgreSQL 등이 있다. 관계형 데이터베이스는 자료를 테이블과 열, 칼럼으로 표현한다. SQL을 사용하면 여러 테이블에 있는 데이터를 그 관계에 따라 조인(join)하여 합칠 수 있다.

비 관계형 데이터베이스는 NoSQL이라고도 부른다. 대표적인 것으로는 CouchDB, Neo4j, Cassandra, HBase, Amazon DynamoDB 등이 있다.<sup>[2]</sup> NoSQL은 다시 네 부류로 나눌 수 있는데, 키-값 저장소(key-value store), 그래프 저

장소(graph store), 칼럼 저장소(column store), 그리고 문서 저장소(document store)가 그것이다. 이런 비-관계형 데이터베이스는 일반적으로 조인 연산은 지원하지 않는다.

대부분의 개발자에게는 관계형 데이터베이스가 최선일 것인데, 40년 이상 시장에서 살아남아 잘 사용되어 온 시스템이라서다. 하지만 여러분이 구축하려는 시스템에 적합하지 않은 경우에는 관계형 데이터베이스 이외의 저장소도 살펴보아야 한다. 아래와 같은 경우에는 비-관계형 데이터베이스가 바람직한 선택일 수 있다.

- 아주 낮은 응답 지연시간(latency)이 요구됨
- 다루는 데이터가 비정형(unstructured)이라 관계형 데이터가 아님
- 데이터(JSON, YAML, XML 등)를 직렬화하거나(serialize) 역직렬화(deserialize) 할 수 있지만 하면 됨
- 아주 많은 양의 데이터를 저장할 필요가 있음

## 수직적 규모 확장 vs 수평적 규모 확장

소위 ‘스케일 업(scale up)’이라고도 하는 수직적 규모 확장(vertical scaling) 프로세스는 서버에 고사양 자원(더 좋은 CPU, 더 많은 RAM 등)을 추가하는 행위를 말한다. 반면 ‘스케일 아웃(scale out)’이라고도 하는 수평적 규모 확장 프로세스는 더 많은 서버를 추가하여 성능을 개선하는 행위를 말한다.

서버로 유입되는 트래픽의 양이 적을 때는 수직적 확장이 좋은 선택이며, 이 방법의 가장 큰 장점은 단순함이다. 그러나 불행하게도 이 방법에는 몇 가지 심각한 단점이 있다.

- 수직적 규모 확장에는 한계가 있다. 한 대의 서버에 CPU나 메모리를 무한대로 증설할 방법은 없다.
- 수직적 규모 확장법은 장애에 대한 자동복구(failover) 방안이나 다중화(redundancy) 방안을 제시하지 않는다. 서버에 장애가 발생하면 웹사이트/앱은 완전히 중단된다.

이런 단점 때문에, 대규모 애플리케이션을 지원하는 데는 수평적 규모 확장법이 보다 적절하다.

앞서 본 설계에서 사용자는 웹 서버에 바로 연결된다. 웹 서버가 다운되면 사용자는 웹 사이트에 접속할 수 없다. 또한, 너무 많은 사용자가 접속하여 웹 서버가 한계 상황에 도달하게 되면 응답 속도가 느려지거나 서버 접속이 불가능해질 수도 있다. 이런 문제를 해결하는 데는 부하 분산기 또는 로드밸런서(load balancer)를 도입하는 것이 최선이다.

### 로드밸런서

로드밸런서는 부하 분산 집합(load balancing set)에 속한 웹 서버들에게 트래픽 부하를 고르게 분산하는 역할을 한다. 그림 1-4는 로드밸런서가 어떻게 동작하는지 보여주고 있다.

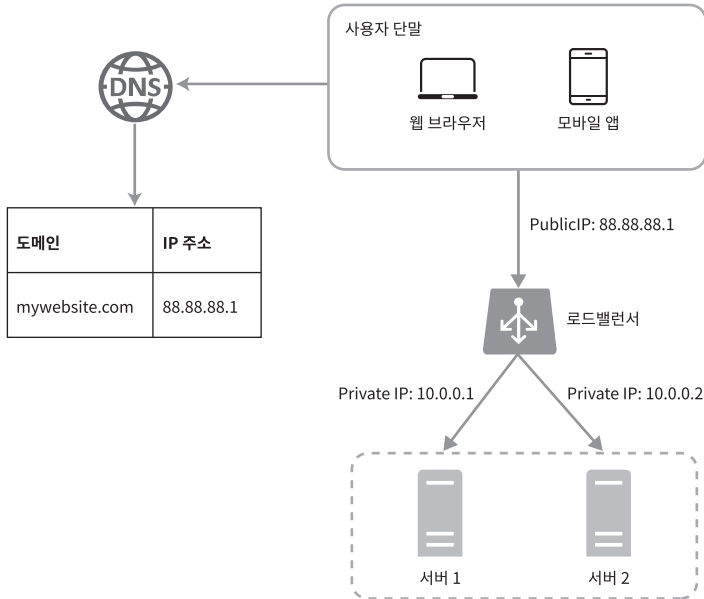


그림 1-4

그림 1-4와 같이, 사용자는 로드밸런서의 공개 IP 주소(public IP address)로 접속한다. 따라서 웹 서버는 클라이언트의 접속을 직접 처리하지 않는다. 더 나

은 보안을 위해, 서버 간 통신에는 사설 IP 주소(private IP address)가 이용된다. 사설 IP 주소는 같은 네트워크에 속한 서버 사이의 통신에만 쓰일 수 있는 IP 주소로, 인터넷을 통해서서는 접속할 수 없다. 로드밸런서는 웹 서버와 통신하기 위해 바로 이 사설 주소를 이용한다.

그림 1-4에 나온 대로, 부하 분산 집합에 또 하나의 웹 서버를 추가하고 나면 장애를 자동복구하지 못하는 문제(no failover)는 해소되며, 웹 계층의 가용성(availability)은 향상된다. 좀 더 구체적으로 살펴보면 다음과 같다.

- 서버 1이 다운되면(offline) 모든 트래픽은 서버 2로 전송된다. 따라서 웹 사이트 전체가 다운되는 일이 방지된다. 부하를 나누기 위해 새로운 서버를 추가할 수도 있다.
- 웹사이트로 유입되는 트래픽이 가파르게 증가하면 두 대의 서버로 트래픽을 감당할 수 없는 시점이 오는데, 로드밸런서가 있으므로 우아하게 대처할 수 있다. 웹 서버 계층에 더 많은 서버를 추가하기만 하면 된다. 그러면 로드밸런스가 자동적으로 트래픽을 분산하기 시작할 것이다.

이제 웹 계층은 괜찮아 보이는데, 그렇다면 데이터 계층은 어떤가? 현재 설계 안에는 하나의 데이터베이스 서버뿐이고, 역시 장애의 자동복구나 다중화를 지원하는 구성은 아니다. 데이터베이스 다중화는 이런 문제를 해결하는 보편적인 기술이다. 한번 살펴보자.

## 데이터베이스 다중화

위키피디아에 따르면, “많은 데이터베이스 관리 시스템이 다중화를 지원한다. 보통은 서버 사이에 주(master)-부(slave) 관계를 설정하고 데이터 원본은 주 서버에, 사본은 부 서버에 저장하는 방식이다.”<sup>[3]</sup>

쓰기 연산(write operation)은 마스터에서만 지원한다. 부 데이터베이스는 주 데이터베이스로부터 그 사본을 전달받으며, 읽기 연산(read operation)만을 지원한다. 데이터베이스를 변경하는 명령어들, 가령 insert, delete, update 등은 주 데이터베이스로만 전달되어야 한다. 대부분의 애플리케이션은 읽기 연산의 비중이 쓰기 연산보다 훨씬 높다. 따라서 통상 부 데이터베이스의 수가

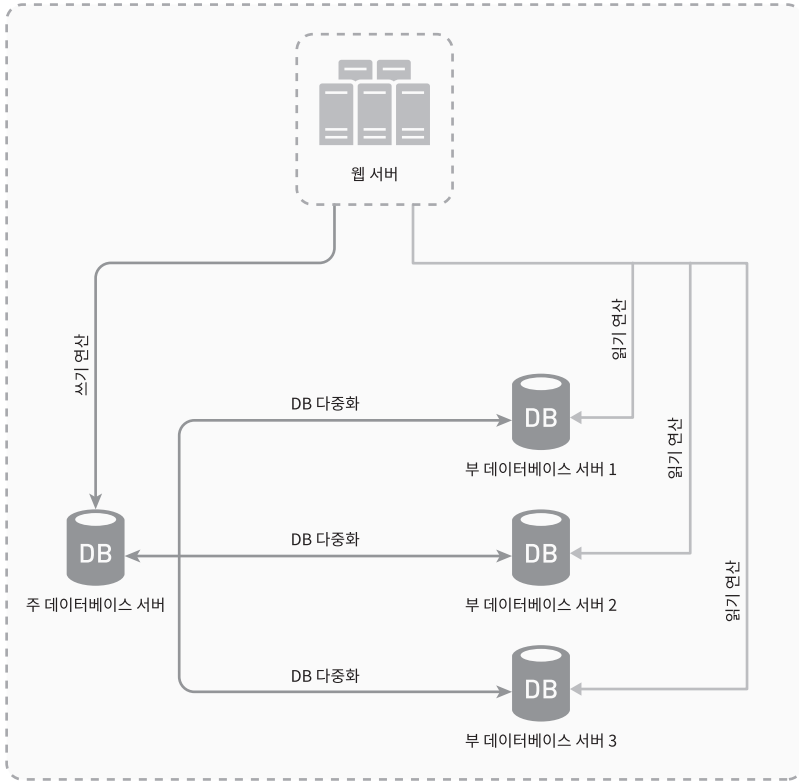


그림 1-5

주 데이터베이스의 수보다 많다. 그림 1-5는 이 구성을 보여주고 있다. 데이터베이스를 다중화하면 다음과 같은 이득이 있다.

- 더 나은 성능: 주-부 다중화 모델에서 모든 데이터 변경 연산은 주 데이터베이스 서버로만 전달되는 반면 읽기 연산은 부 데이터베이스 서버들로 분산된다. 병렬로 처리될 수 있는 질의(query)의 수가 늘어나므로, 성능이 좋아진다.
- 안정성(reliability): 자연 재해 등의 이유로 데이터베이스 서버 가운데 일부가 파괴되어도 데이터는 보존될 것이다. 데이터를 지역적으로 떨어진 여러 장소에 다중화시켜 놓을 수 있기 때문이다.

- 가용성(availability): 데이터를 여러 지역에 복제해 둬으로써, 하나의 데이터베이스 서버에 장애가 발생하더라도 다른 서버에 있는 데이터를 가져와 계속 서비스할 수 있게 된다.

앞 절에서 우리는 로드밸런서가 시스템 가용성을 어떻게 높이는지 살펴보았다. 데이터베이스에 대해서도 같은 질문을 던져보자. 데이터베이스 서버 가운데 하나가 다운되면 무슨 일이 벌어지는가? 그림 1-5에 제시한 설계는 이런 상황을 감당할 수 있다.

- 부 서버가 한 대 뿐인데 다운된 경우라면, 읽기 연산은 한시적으로 모두 주 데이터베이스로 전달될 것이다. 또한 즉시 새로운 부 데이터베이스 서버가 장애 서버를 대체할 것이다. 부 서버가 여러 대인 경우에 읽기 연산은 나머지 부 데이터베이스 서버들로 분산될 것이며, 새로운 부 데이터베이스 서버가 장애 서버를 대체할 것이다.
- 주 데이터베이스 서버가 다운되면, 한 대의 부 데이터베이스만 있는 경우 해당 부 데이터베이스 서버가 새로운 주 서버가 될 것이며, 모든 데이터베이스 연산은 일시적으로 새로운 주 서버상에서 수행될 것이다. 그리고 새로운 부 서버가 추가될 것이다. 프로덕션(production) 환경에서 벌어지는 일은 이것보다는 사실 더 복잡한데, 부 서버에 보관된 데이터가 최신 상태가 아닐 수 있기 때문이다. 없는 데이터는 복구 스크립트(recovery script)를 돌려서 추가해야 한다. 다중 마스터(multi-masters)나 원형 다중화(circular replication) 방식을 도입하면 이런 상황에 대처하는 데 도움이 될 수도 있지만 해당 구성은 훨씬 복잡하며 그에 대한 논의는 이 책에서 다룰 수 있는 내용의 범위를 넘어선다. 관심 있는 독자들은 참고 문헌 [4][5]를 살펴보기 바란다.

그림 1-6은 로드밸런서와 데이터베이스 다중화를 고려한 설계안이다.

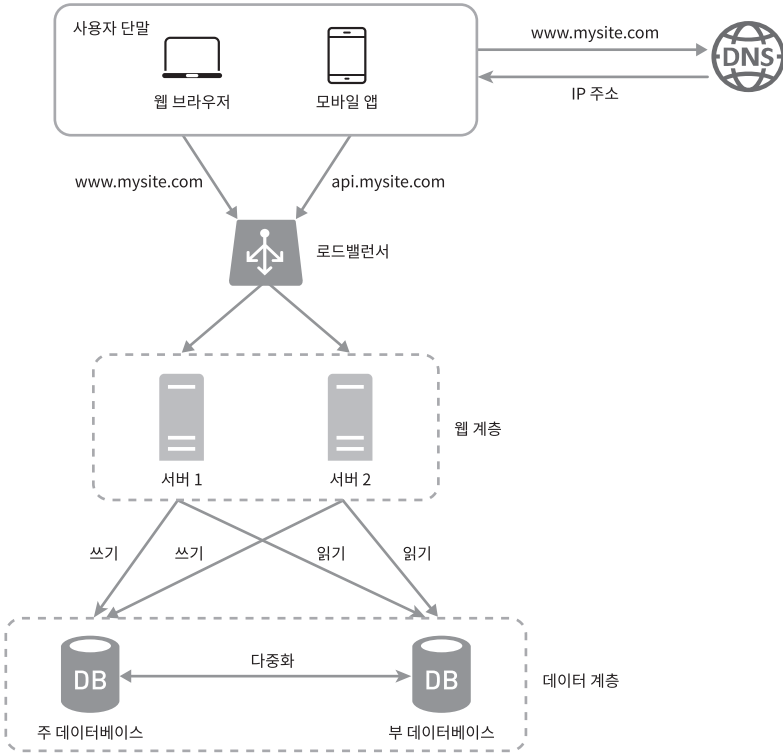


그림 1-6

이 설계안은 다음과 같이 동작한다.

- 사용자는 DNS로부터 로드밸런서의 공개 IP 주소를 받는다.
- 사용자는 해당 IP 주소를 사용해 로드밸런서에 접속한다.
- HTTP 요청은 서버 1이나 서버 2로 전달된다.
- 웹 서버는 사용자의 데이터를 부 데이터베이스 서버에서 읽는다.
- 웹 서버는 데이터 변경 연산은 주 데이터베이스로 전달한다. 데이터 추가, 삭제, 갱신 연산 등이 이에 해당한다.

이제 웹 계층과 데이터 계층에 대해 충분히 이해하게 되었으니, 응답시간(latency)을 개선해 볼 순서다. 응답 시간은 캐시(cache)를 붙이고 정적 콘텐츠를 콘텐츠 전송 네트워크(Content Delivery Network, CDN)로 옮기면 개선할 수 있다.

## 캐시

캐시는 값비싼 연산 결과 또는 자주 참조되는 데이터를 메모리 안에 두고, 뒤이은 요청이 보다 빨리 처리될 수 있도록 하는 저장소다. 그림 1-6에서 살펴본 바와 같이, 웹 페이지를 새로고침 할 때마다 표시할 데이터를 가져오기 위해 한 번 이상의 데이터베이스 호출이 발생한다. 애플리케이션의 성능은 데이터베이스를 얼마나 자주 호출하느냐에 크게 좌우되는데, 캐시는 그런 문제를 완화할 수 있다.

### 캐시 계층

캐시 계층(cache tier)은 데이터가 잠시 보관되는 곳으로 데이터베이스보다 훨씬 빠르다. 별도의 캐시 계층을 두면 성능이 개선될 뿐 아니라 데이터베이스의 부하를 줄일 수 있고, 캐시 계층의 규모를 독립적으로 확장시키는 것도 가능해진다. 그림 1-7은 캐시 서버를 두는 방법 중 하나다.

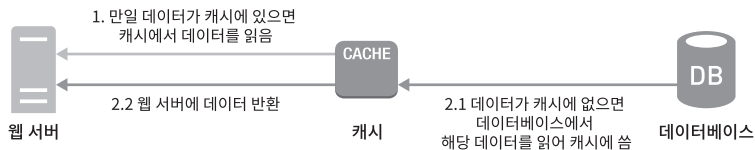


그림 1-7

요청을 받은 웹 서버는 캐시에 응답이 저장되어 있는지를 본다. 만일 저장되어 있다면 해당 데이터를 클라이언트에 반환한다. 없는 경우에는 데이터베이스 질의를 통해 데이터를 찾아 캐시에 저장한 뒤 클라이언트에 반환한다. 이러한 캐시 전략을 읽기 주도형 캐시 전략(read-through caching strategy)이라고 부른다. 이것 이외에도 다양한 캐시 전략이 있는데, 캐시할 데이터 종류, 크기, 액세스 패턴에 맞는 캐시 전략을 선택하면 된다. 캐시 전략에 대한 비교 연구 결과는 [6]을 참조하기 바란다.

캐시 서버를 이용하는 방법은 간단한데 대부분의 캐시 서버들이 일반적으로 널리 쓰이는 프로그래밍 언어로 API를 제공하기 때문이다. 다음 쪽 코드는 memcached API의 전형적 사용 예다.



```
SECONDS = 1
cache.set('myKey', 'hi there', 3600 * SECONDS)
cache.get('myKey')
```

## 캐시 사용 시 유의할 점

캐시를 사용할 때는 아래 사항들을 고려하여야 한다.

- 캐시는 어떤 상황에 바람직한가? 데이터 갱신은 자주 일어나지 않지만 참조는 빈번하게 일어난다면 고려해볼 만하다.
- 어떤 데이터를 캐시에 두어야 하는가? 캐시는 데이터를 휘발성 메모리에 두므로, 영속적으로 보관할 데이터를 캐시에 두는 것은 바람직하지 않다. 예를 들어, 캐시 서버가 재시작되면 캐시 내의 모든 데이터는 사라진다. 중요 데이터는 여전히 지속적 저장소(persistent data store)에 두어야 한다.
- 캐시에 보관된 데이터는 어떻게 만료(expire)되는가? 이에 대한 정책을 마련해 두는 것은 좋은 습관이다. 만료된 데이터는 캐시에서 삭제되어야 한다. 만료 정책이 없으면 데이터는 캐시에 계속 남게 된다. 만료 기한은 너무 짧으면 곤란한데, 데이터베이스를 너무 자주 읽게 될 것이기 때문이다. 너무 길어도 곤란한데, 원본과 차이가 날 가능성이 높아지기 때문이다.
- 일관성(consistency)은 어떻게 유지되는가? 일관성은 데이터 저장소의 원본과 캐시 내의 사본이 같은지 여부다. 저장소의 원본을 갱신하는 연산과 캐시를 갱신하는 연산이 단일 트랜잭션으로 처리되지 않는 경우 이 일관성은 깨질 수 있다. 여러 지역에 걸쳐 시스템을 확장해 나가는 경우 캐시와 저장소 사이의 일관성을 유지하는 것은 어려운 문제가 된다. 이에 대해서는 페이스북에서 내놓은 논문 <Scaling Memcache at Facebook><sup>[7]</sup>을 참고하기 바란다.
- 장애에는 어떻게 대처할 것인가? 캐시 서버를 한 대만 두는 경우 해당 서버는 단일 장애 지점(Single Point of Failure, SPOF)이 되어버릴 가능성이 있다. 위키피디아에 따르면 단일 장애 지점의 정의는 다음과 같다. “어떤 특정 지점에서의 장애가 전체 시스템의 동작을 중단시켜버릴 수 있는 경우, 우리

는 해당 지점을 단일 장애 지점이라고 부른다.<sup>16)</sup> 결과적으로, SPOF를 피하려면 여러 지역에 걸쳐 캐시 서버를 분산시켜야 한다.

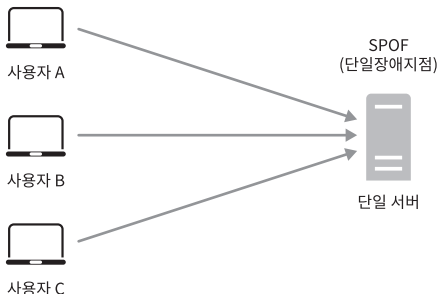


그림 1-8

- 캐시 메모리는 얼마나 크게 잡을 것인가? 캐시 메모리가 너무 작으면 액세스 패턴에 따라서는 데이터가 너무 자주 캐시에서 밀려나버려(eviction) 캐시의 성능이 떨어지게 된다. 이를 막을 한 가지 방법은 캐시 메모리를 과할당(overprovision)하는 것이다. 이렇게 하면 캐시에 보관될 데이터가 갑자기 늘어났을 때 생길 문제도 방지할 수 있게 된다.
- 데이터 방출(eviction) 정책은 무엇인가? 캐시가 꽉 차버리면 추가로 캐시에 데이터를 넣어야 할 경우 기존 데이터를 내보내야 한다. 이것을 캐시 데이터 방출 정책이라 하는데, 그 가운데 가장 널리 쓰이는 것은 LRU(Least Recently Used - 마지막으로 사용된 시점이 가장 오래된 데이터를 내보내는 정책)이다. 다른 정책으로는 LFU(Least Frequently Used - 사용된 빈도가 가장 낮은 데이터를 내보내는 정책)나 FIFO(First In First Out - 가장 먼저 캐시에 들어온 데이터를 가장 먼저 내보내는 정책) 같은 것도 있으며, 경우에 맞게 적용 가능하다.

## 콘텐츠 전송 네트워크(CDN)

CDN은 정적 콘텐츠를 전송하는 데 쓰이는, 지리적으로 분산된 서버의 네트워크이다. 이미지, 비디오, CSS, JavaScript 파일 등을 캐시할 수 있다.

동적 콘텐츠 캐싱은 상대적으로 새로운 개념으로서, 이 책에서 다룰 수 있

는 범위 밖이다. 간단하게만 요약하면, 요청 경로(request path), 질의 문자열(query string), 쿠키(cookie), 요청 헤더(request header) 등의 정보에 기반하여 HTML 페이지를 캐시하는 것이다. 이에 대해 더 자세히 알고 싶으면 [9]를 참고하기 바란다. 이 책에서는 CDN을 사용하여 정적 콘텐츠를 캐시하는 방법에만 집중할 것이다.

CDN이 어떻게 동작하는지를 개략적으로만 살펴보면 다음과 같다. 어떤 사용자가 웹사이트를 방문하면, 그 사용자에게 가장 가까운 CDN 서버가 정적 콘텐츠를 전달하게 된다. 직관적으로도 당연하겠지만, 사용자가 CDN 서버로부터 멀면 멀수록 웹사이트는 천천히 로드될 것이다. 예를 들어, CDN 서버가 샌프란시스코에 있다면 LA에 있는 사용자는 유럽 사용자보다 빠른 웹 사이트를 보게 될 것이다. 그림 1-9는 CDN이 사이트 로딩 시간을 어떻게 개선하는지를 보여주는 좋은 예다.

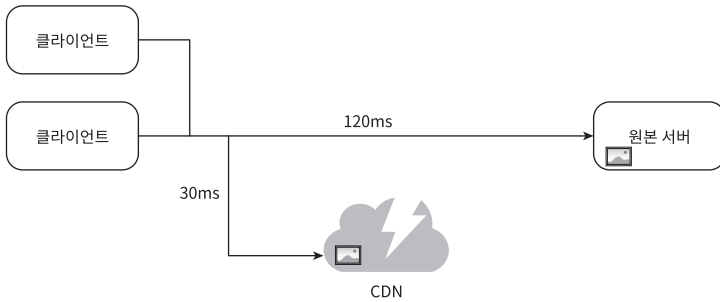


그림 1-9

그림 1-10은 CDN이 어떻게 동작하는지를 설명한다.

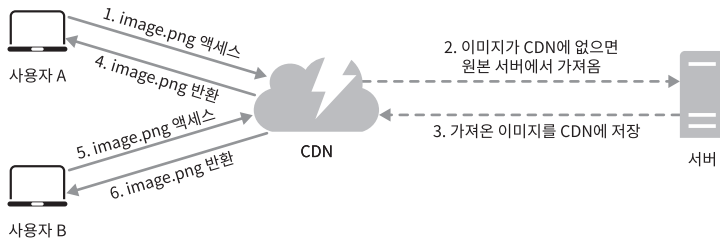


그림 1-10

1. 사용자 A가 이미지 URL을 이용해 image.png에 접근한다. URL의 도메인은 CDN 서비스 사업자가 제공한 것이다. 아래의 두 URL은 클라우드프론트(Cloudfront)와 아카마이(Akamai) CDN이 제공하는 URL의 예제다.
  - <https://mysite.cloudfront.net/logo.jpg>
  - <https://mysite.akamai.com/image-manager/img/logo.jpg>
2. CDN 서버의 캐시에 해당 이미지가 없는 경우, 서버는 원본(origin) 서버에 요청하여 파일을 가져온다. 원본 서버는 웹 서버일 수도 있고 아마존(Amazon) S3 같은 온라인 저장소일 수도 있다.
3. 원본 서버가 파일을 CDN 서버에 반환한다. 응답의 HTTP 헤더에는 해당 파일이 얼마나 오래 캐시될 수 있는지를 설명하는 TTL(Time-To-Live) 값이 들어 있다.
4. CDN 서버는 파일을 캐시하고 사용자 A에게 반환한다. 이미지는 TTL에 명시된 시간이 끝날 때까지 캐시된다.
5. 사용자 B가 같은 이미지에 대한 요청을 CDN 서버에 전송한다.
6. 만료되지 않은 이미지에 대한 요청은 캐시를 통해 처리된다.

### CDN 사용 시 고려해야 할 사항

- 비용: CDN은 보통 제3 사업자(third-party providers)에 의해 운영되며, 여러 분은 CDN으로 들어가고 나가는 데이터 전송 양에 따라 요금을 내게 된다. 자주 사용되지 않는 콘텐츠를 캐싱하는 것은 이득이 크지 않으므로, CDN에서 빼는 것을 고려하도록 하자.
- 적절한 만료 시한 설정: 시의성이 중요한(time-sensitive) 콘텐츠의 경우 만료 시점을 잘 정해야 한다. 너무 길지도 않고 짧지도 않아야 하는데, 너무 길면 콘텐츠의 신선도는 떨어질 것이고, 너무 짧으면 원본 서버에 빈번히 접속하게 되어서 좋지 않다.
- CDN 장애에 대한 대처 방안: CDN 자체가 죽었을 경우 웹사이트/애플리케이션이 어떻게 동작해야 하는지 고려해야 한다. 가령 일시적으로 CDN이 응

답하지 않을 경우, 해당 문제를 감지하여 원본 서버로부터 직접 콘텐츠를 가져오도록 클라이언트를 구성하는 것이 필요할 수도 있다.

- 콘텐츠 무효화(invalidation) 방법: 아직 만료되지 않은 콘텐츠라 하더라도 아래 방법 가운데 하나를 쓰면 CDN에서 제거할 수 있다.
  - CDN 서비스 사업자가 제공하는 API를 이용하여 콘텐츠 무효화
  - 콘텐츠의 다른 버전을 서비스하도록 오브젝트 버저닝(object versioning) 이용. 콘텐츠의 새로운 버전을 지정하기 위해서는 URL 마지막에 버전 번호를 인자로 주면 된다. 예를 들어, image.png?v=2와 같은 식이다.

그림 1-11은 CDN과 캐시가 추가된 설계다.

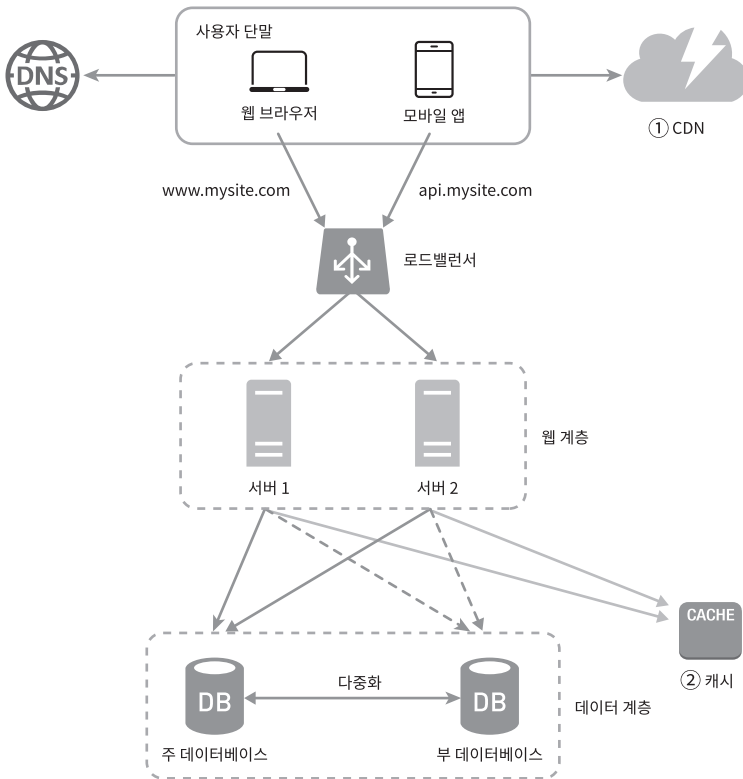


그림 1-11

변화된 부분은 다음과 같다.

1. 정적 콘텐츠(JS, CSS, 이미지 등)는 더 이상 웹 서버를 통해 서비스하지 않으며, CDN을 통해 제공하여 더 나은 성능을 보장한다.
2. 캐시가 데이터베이스 부하를 줄여준다.

## 무상태(stateless) 웹 계층

이제 웹 계층을 수평적으로 확장하는 방법을 고민해 볼 순서다. 이를 위해서는 상태 정보(사용자 세션 데이터와 같은)를 웹 계층에서 제거하여야 한다. 바람직한 전략은 상태 정보를 관계형 데이터베이스나 NoSQL 같은 지속성 저장소에 보관하고, 필요할 때 가져오도록 하는 것이다. 이렇게 구성된 웹 계층을 무상태 웹 계층이라 부른다.

### 상태 정보 의존적인 아키텍처

상태 정보를 보관하는 서버와 그렇지 않은 서버 사이에는 몇 가지 중요한 차이가 있다. 상태 정보를 보관하는 서버는 클라이언트 정보, 즉 상태를 유지하여 요청들 사이에 공유되도록 한다. 무상태 서버에는 이런 장치가 없다.

그림 1-12는 상태 정보 의존적인 아키텍처를 보여준다.

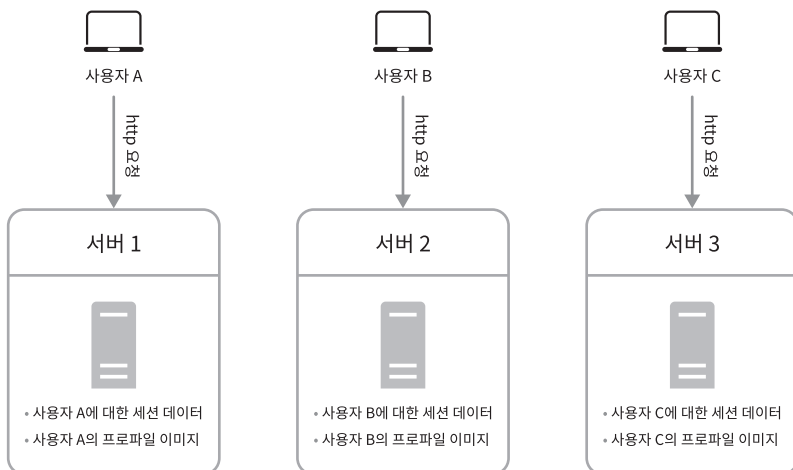


그림 1-12

그림 1-12에서 사용자 A의 세션 정보나 프로파일 이미지 같은 상태 정보는 서버 1에 저장된다. 사용자 A를 인증하기 위해 HTTP 요청은 반드시 서버 1로 전송되어야 한다. 요청이 서버 2로 전송되면 인증은 실패할 것인데, 서버 2에 사용자 A에 관한 데이터는 보관되어 있지 않기 때문이다. 마찬가지로, 사용자 B로부터의 HTTP 요청은 전부 서버 2로 전송되어야 하고, 사용자 C로부터의 요청은 전부 서버 3으로 전송되어야 한다.

문제는 같은 클라이언트로부터의 요청은 항상 같은 서버로 전송되어야 한다는 것이다. 대부분의 로드밸런서가 이를 지원하기 위해 고정 세션(sticky session)이라는 기능을 제공하고 있는데<sup>[10]</sup>, 이는 로드밸런서에 부담을 준다. 게다가 로드밸런서 뒷단에 서버를 추가하거나 제거하기도 까다로워진다. 이들 서버의 장애를 처리하기도 복잡해진다.

### 무상태 아키텍처

그림 1-13은 무상태 아키텍처를 보여준다.

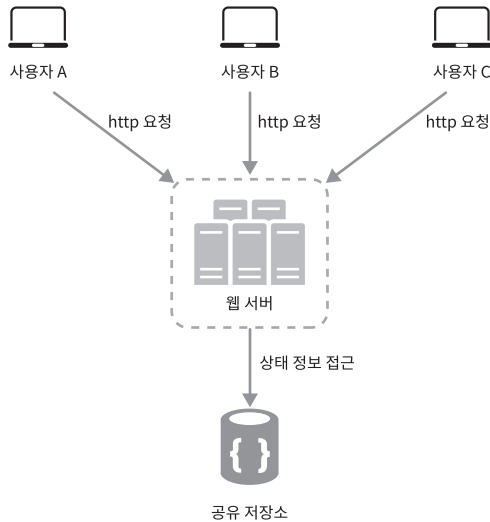


그림 1-13

이 구조에서 사용자로부터의 HTTP 요청은 어떤 웹 서버로도 전달될 수 있다. 웹 서버는 상태 정보가 필요할 경우 공유 저장소(shared storage)로부터 데이

터를 가져온다. 따라서 상태 정보는 웹 서버로부터 물리적으로 분리되어 있다. 이런 구조는 단순하고, 안정적이며, 규모 확장이 쉽다.

그림 1-14는 무상태 웹 계층을 갖도록 기존 설계를 변경한 결과다.

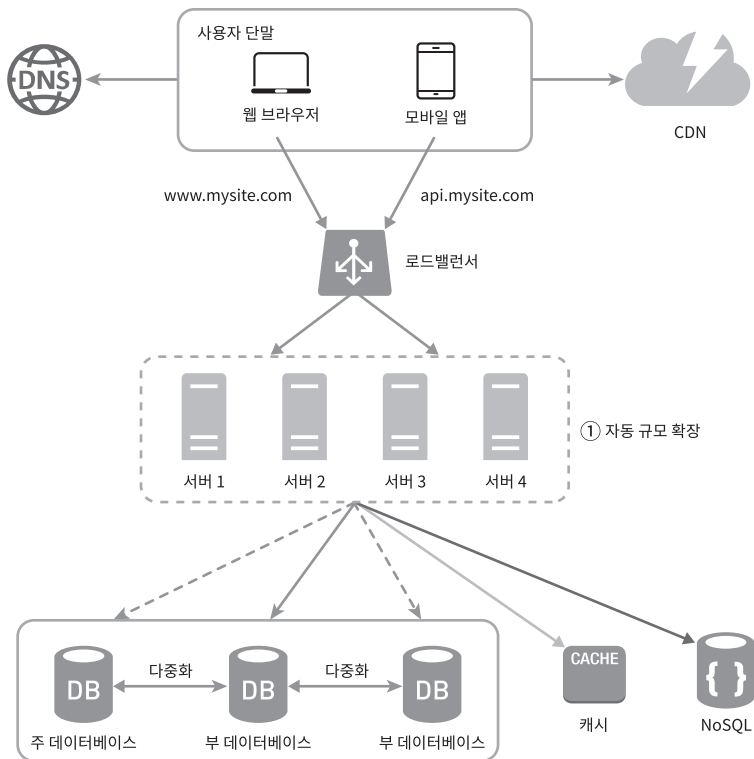


그림 1-14

그림 1-14에서 우리는 세션 데이터를 웹 계층에서 분리하고 지속성 데이터 보관소에 저장하도록 만들었다. 이 공유 저장소는 관계형 데이터베이스일 수도 있고, Memcached/Redis 같은 캐시 시스템일 수도 있으며, NoSQL일 수도 있다. 여기서는 NoSQL을 사용하였는데, 규모 확장이 간편해서다. ①의 자동 규모 확장(autoscaling)은 트래픽 양에 따라 웹 서버를 자동으로 추가하거나 삭제하는 기능을 뜻한다. 상태 정보가 웹 서버들로부터 제거되었으므로, 트래픽 양에 따라 웹 서버를 넣거나 빼기만 하면 자동으로 규모를 확장할 수 있게 되었다.

여러분의 웹사이트가 매우 빨리 성장하여 전 세계 사용자의 이목을 받는 시



점이 왔다고 해 보자. 가용성을 높이고 전 세계 어디서든 쾌적하게 사용할 수 있도록 하기 위해서는 여러 데이터 센터(data center)를 지원하는 것이 필요하다.

## 데이터 센터

그림 1-15는 두 개의 데이터 센터를 이용하는 사례다. 장애가 없는 상황에서 사용자는 가장 가까운 데이터 센터로 안내되는데, 통상 이 절차를 지리적 라우팅(geoDNS-routing 또는 geo-routing)이라고 부른다. 지리적 라우팅에서의 geoDNS는 사용자의 위치에 따라 도메인 이름을 어떤 IP 주소로 변환할지 결정할 수 있도록 해 주는 DNS 서비스다. 이 예제의 경우, 그 결과로 x% 사용자는 US-East 센터로, 그리고 (100-x)%의 사용자는 US-West 센터로 안내된다고 하자.

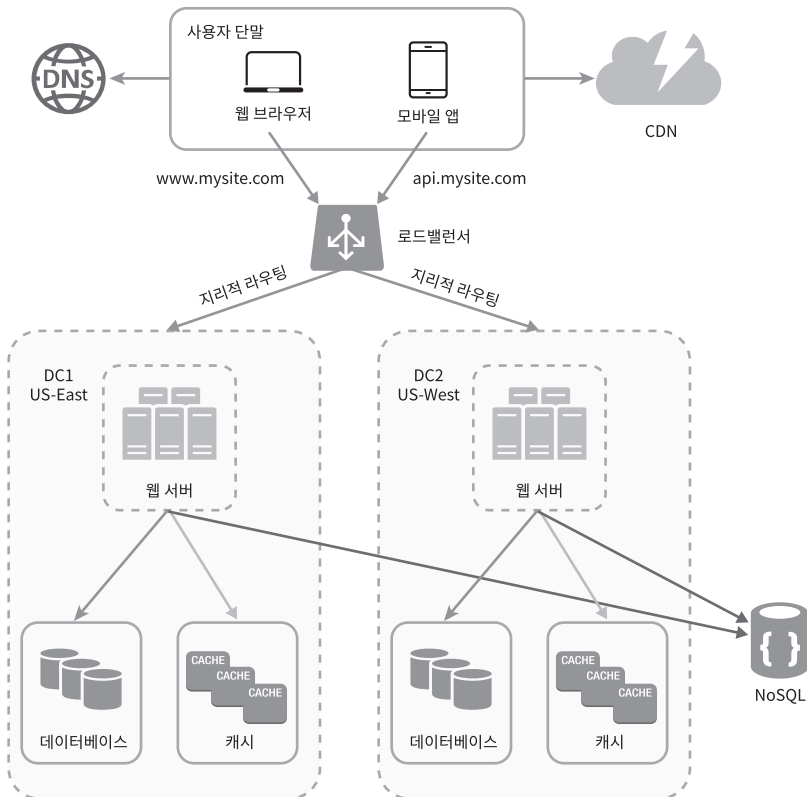


그림 1-15

이들 데이터 센터 중 하나에 심각한 장애가 발생하면 모든 트래픽은 장애가 없는 데이터 센터로 전송된다. 그림 1-16은 데이터센터2(US-West)에 장애가 발생하여 모든 트래픽이 데이터센터1(US-East)로 전송되는 상황을 보여준다.

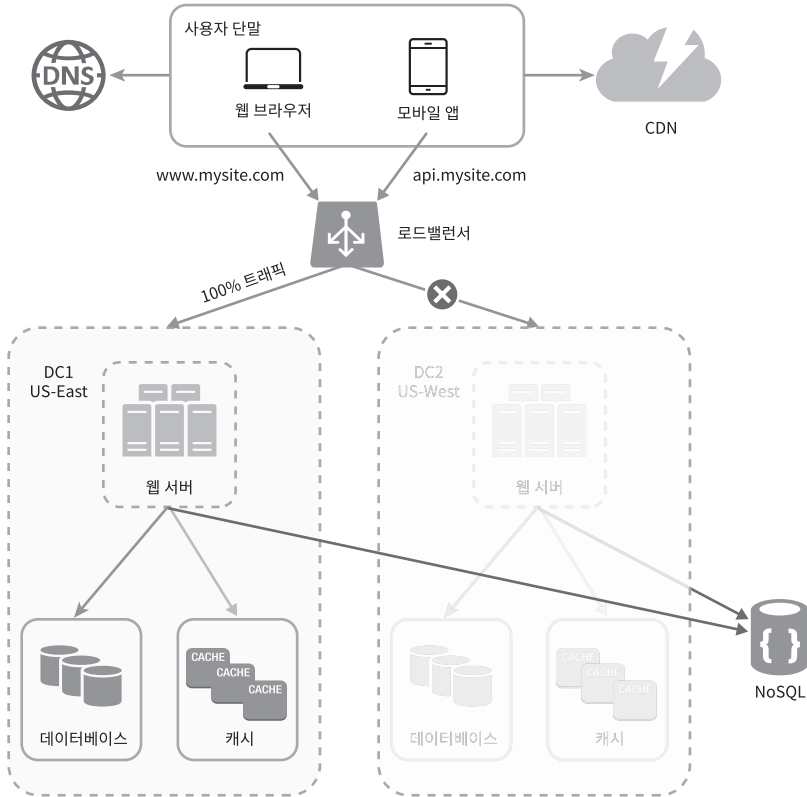


그림 1-16

이 사례와 같은 다중 데이터센터 아키텍처를 만들려면 몇 가지 기술적 난제를 해결해야 한다.

- 트래픽 우회: 올바른 데이터 센터로 트래픽을 보내는 효과적인 방법을 찾아야 한다. GeoDNS는 사용자에게서 가장 가까운 데이터센터로 트래픽을 보낼 수 있도록 해 준다.
- 데이터 동기화(synchronization): 데이터 센터마다 별도의 데이터베이스

를 사용하고 있는 상황이라면, 장애가 자동으로 복구되어(failover) 트래픽이 다른 데이터베이스로 우회된다 해도, 해당 데이터센터에는 찾는 데이터가 없을 수 있다. 이런 상황을 막는 보편적 전략은 데이터를 여러 데이터센터에 걸쳐 다중화하는 것이다. 넷플릭스(Netflix)가 여러 데이터센터에 걸쳐 데이터를 어떻게 다중화하는지 관심이 있다면 [11]을 읽어보기 바란다.

- 테스트와 배포(deployment): 여러 데이터 센터를 사용하도록 시스템이 구성된 상황이라면 웹 사이트 또는 애플리케이션을 여러 위치에서 테스트해 보는 것이 중요하다. 한편, 자동화된 배포 도구는 모든 데이터 센터에 동일한 서비스가 설치되도록 하는 데 중요한 역할을 한다.<sup>[11]</sup>

시스템을 더 큰 규모로 확장하기 위해서는 시스템의 컴포넌트를 분리하여, 각기 독립적으로 확장될 수 있도록 하여야 한다. 메시지 큐(message queue)는 많은 실제 분산 시스템이 이 문제를 풀기 위해 채용하고 있는 핵심적 전략 가운데 하나다.

## 메시지 큐

메시지 큐는 메시지의 무손실(durability, 즉 메시지 큐에 일단 보관된 메시지는 소비자가 꺼낼 때까지 안전하게 보관된다는 특성)을 보장하는, 비동기 통신(asynchronous communication)을 지원하는 컴포넌트다. 메시지의 버퍼 역할을 하며, 비동기적으로 전송한다. 메시지 큐의 기본 아키텍처는 간단하다. 생산자 또는 발행자(producer/publisher)라고 불리는 입력 서비스가 메시지를 만들어 메시지 큐에 발행(publish)한다. 큐에는 보통 소비자 혹은 구독자(consumer/subscriber)라 불리는 서비스 혹은 서버가 연결되어 있는데, 메시지를 받아 그에 맞는 동작을 수행하는 역할을 한다. 그림 1-17은 이 구조를 보여준다.



그림 1-17

메시지 큐를 이용하면 서비스 또는 서버 간 결합이 느슨해져서, 규모 확장성이 보장되어야 하는 안정적 애플리케이션을 구성하기 좋다. 생산자는 소비자 프로세스가 다운되어 있어도 메시지를 발행할 수 있고, 소비자는 생산자 서비스가 가용한 상태가 아니더라도 메시지를 수신할 수 있다.

사용 예로는 다음과 같은 것을 들 수 있다. 가령 이미지의 크로핑(cropping), 샤프닝(sharpening), 블러링(blurring) 등을 지원하는 사진 보정 애플리케이션을 만든다고 해 보자. 이러한 보정은 시간이 오래 걸릴 수 있는 프로세스이므로 비동기적으로 처리하면 편리하다. 그림 1-18에서 웹 서버는 사진 보정 작업(job)을 메시지 큐에 넣는다. 사진 보정 작업(worker) 프로세스들은 이 작업을 메시지 큐에서 꺼내어 비동기적으로 완료한다. 이렇게 하면 생산자와 소비자 서비스의 규모는 각기 독립적으로 확장될 수 있다. 큐의 크기가 커지면 더 많은 작업 프로세스를 추가해야 처리 시간을 줄일 수 있다. 하지만 큐가 거의 항상 비어 있는 상태라면, 작업 프로세스의 수는 줄일 수 있을 것이다.



그림 1-18

## 로그, 메트릭 그리고 자동화

몇 개 서버에서 실행되는 소규모 웹 사이트를 만들 때는 로그나 메트릭(metric), 자동화(automation) 같은 것은 하면 좋지만 꼭 할 필요는 없었다. 하지만 일단 웹 사이트와 함께 사업 규모가 커지고 나면, 그런 도구에 필수적으로 투자해야 한다.

- 로그: 에러 로그를 모니터링하는 것은 중요하다. 시스템의 오류와 문제들을 보다 쉽게 찾아낼 수 있도록 하기 때문이다. 에러 로그는 서버 단위로 모니터링 할 수도 있지만, 로그를 단일 서비스로 모아주는 도구를 활용하면 더

편리하게 검색하고 조회할 수 있다.

- **메트릭:** 메트릭을 잘 수집하면 사업 현황에 관한 유용한 정보를 얻을 수도 있고, 시스템의 현재 상태를 손쉽게 파악할 수도 있다. 메트릭 가운데 특히 유용한 것을 몇 가지 살펴보면 다음과 같다.
  - **호스트 단위 메트릭:** CPU, 메모리, 디스크 I/O에 관한 메트릭이 여기 해당한다.
  - **종합(aggregated) 메트릭:** 데이터베이스 계층의 성능, 캐시 계층의 성능 같은 것이 여기 해당한다.
  - **핵심 비즈니스 메트릭:** 일별 능동 사용자(daily active user), 수익(revenue), 재방문(retention) 같은 것이 여기 해당한다.
- **자동화:** 시스템이 크고 복잡해지면 생산성을 높이기 위해 자동화 도구를 활용해야 한다. 가령 지속적 통합(continuous integration)을 도와주는 도구를 활용하면 개발자가 만드는 코드가 어떤 검증 절차를 자동으로 거치도록 할 수 있어서 문제를 쉽게 감지할 수 있다. 이 외에도 빌드, 테스트, 배포 등의 절차를 자동화할 수 있어서 개발 생산성을 크게 향상시킬 수 있다.

### **메시지 큐, 로그, 메트릭, 자동화 등을 반영하여 수정한 설계안**

그림 1-19은 이들 도구들과 메시지 큐를 적용하여 수정한 설계다. 지면 관계상 하나의 데이터센터만 그림에 포함시켰다.

1. 메시지 큐는 각 컴포넌트가 보다 느슨히 결합(loosely coupled)될 수 있도록 하고, 결합에 대한 내성을 높인다.
2. 로그, 모니터링, 메트릭, 자동화 등을 지원하기 위한 장치를 추가하였다.