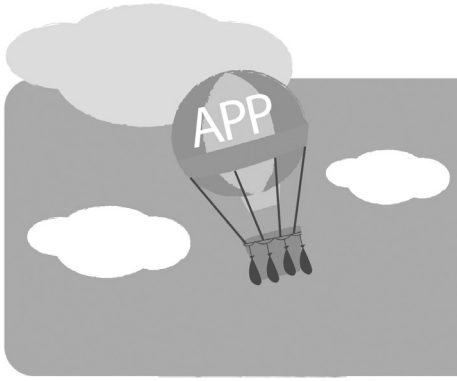


CHAPTER 00

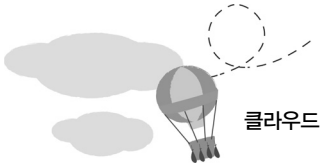
소개

클라우드 기반의 애플리케이션 개발에 대한 설명에 앞서서 클라우드를 이해하기 위한 몇 가지 핵심 개념을 먼저 소개한다. **클라우드 컴퓨팅(cloud computing)**의 개념은 최근 몇 년의 짧은 시간에 발전한 것이긴 하지만, 사용되는 개념과 패턴은 컴퓨터의 역사가 시작되는 시점부터 계속 회자되어 왔던 것들이다.

클라우드 컴퓨팅이란 대체 무엇인가?

클라우드에 대한 정의는 많지만, 이 정의들 대부분은 클라우드를 만들었던 사람들 외에는 이해하기가 다소 모호한 것들이 대다수다. 많은 기업들은 가상 호스팅으로 구성된 환경을 **클라우드(cloud)**라고 부르는데, 그 저변에는 가상화가 컴퓨팅 파워, 스피드, 확장성을 제공한다고 생각하기 때문이다. 하지만 클라우드는 무한대에 가까운 확장성을 가지는 컴퓨팅과 스토리지의 클러스터 이상의 개념을 내포하고 있는데, 이 중 주요한 개념들 중 하나는 대다수의 클라우드 서비스들이 사용한 만큼 과금하는 형태를 띠고 있다는 점이다.

클라우드 서비스는 보통 비용이 저렴하다는 장점을 가지고 있는데, 이는 사업 초기에 서비스에 필요한 자원들을 구매하지 않아도 된다는 점과 이런 자원을 관리하기 위한 IT 관련 인력을 고용할 필요가 없기 때문이다. 클라우드 컴퓨팅의 세상으로 뛰어들게 될 때 누릴 수 있는 축복은 비용뿐만이 아니다. 클라우드 컴퓨팅은 제대로 된 디자인패턴을 가진 애플리케이션의 즉각적인 확장 역시 가능하게 하며, 지난 오랜 세월 동안 관리자가 해왔던 관리에 필요한



시간과 노력을 획기적으로 줄여주는 이점도 제공한다. 기존의 서비스는 비즈니스의 요구에 따라 하드웨어의 장애, 신규증설 등을 위해 숙련된 관리자가 오랜 시간 일을 해야 했었다. 하지만 클라우드 기반 플랫폼에서는 단순히 서비스 공급자에게 더 많은 서버와 서버들에 대한 사용시간을 요청하기만 하면 즉시 해결된다. 이런 것들이 가능한 이유는 보통 클라우드 서비스 제공 기업이 클라우드 서비스를 이용하는 기업보다 규모가 크고, 클라우드 플랫폼을 항상 최상의 상태로 만들기 위해 필요한 모든 인력을 확보하여 발생하는 문제에 즉각 대응하기 때문이다.

여러분이 만약 한밤중이나 새벽 시간대의 서비스 중단을 경험한 적이 있다면, 이걸 고치는데 얼마나 많은 사람의 고통과 노력이 수반되어야 하는지 알고 있을 것이다. 따라서 물리적으로 사용하던 서버를 클라우드에 변환하여 구성함으로써 시스템 관리에 대한 걱정을 덜고 애플리케이션 개발에 매진할 수 있을 것이다.

모든 사업가들에게 폭발적으로 증가하는 고객의 비즈니스 요구를 수용 가능하도록 인프라를 구성하는 것은 중요한 목표다. 하지만 고객의 요청이 언제, 얼마큼 증가할지도 모르는데 미리 이 수요를 예측하여 많은 수의 서버를 미리 구성할 만한 비용을 확보하고 있지도 않을 것이며, 그렇게 해서도 안 된다. 더군다나 이제 사업을 시작한 사람이라면 그 사업이 성공할지 아닐지에 대한 확신도 없다. 하지만 사업이 어느 순간 갑자기 성공하게 된다면, 20명의 고객이 사용하던 웹 서비스용 애플리케이션이 동작할 수 있는 인프라를 20,000명의 클라이언트를 수용 가능하도록 야간에 확장해야 할 필요가 있을 것이다.

이러한 상황에서 직접 서버를 소유하거나 호스팅 서비스를 사용하는 기존의 방법에서는 비즈니스 요구사항을 수용하기 위해 필요한 서버군을 하룻밤 안에 물리적으로 확장 가능한 방법이 없다. 또한 가능한 최단 시간 안에 구매해서 새로 설치한 서버들이 제 기능을 잘 해주기를 기도할 수밖에 없다. 이런 지경이 되면 이제 재미와 관심 있었던 주제를 위해 즐기던 여가의 시간들을 데이터와 클러스터의 관리 및 유지에 쏟아 부어야 한다. 하지만 만약 클라우드에 서비스를 구성하여 사용하고 있다면, 클라우드 서비스를 제공하는 기업의 웹페이지 또는 그들의 서비스에 간단한 API 호출을 날려서 40대의 새로운 서버를 단지 몇 분 안에 준비할 수 있게 된다. 더군다나 고객이 200명으로 줄어드는 때가 온다면, 필요 없는 서버들은 그저 종료시켜 버리면 그만이다. 이러한 서비스의 사용 형태는 클라우드를 사용하는 것이 어떻게 비용을

줄여 주는지 아주 노골적으로 보여주는 사례다.

클라우드 컴퓨팅의 진화

필자는 클라우드 컴퓨팅과 새로운 소프트웨어 아키텍처가 과거의 디자인과 대체 어디가 어떻게 다르며, 사실 우리가 과거로 회귀하고 있는 게 아닌가 하는 논의를 한 적이 있다. 하지만 과거에 했던 서비스의 디자인과 현대의 디자인이 얼마간의 유사성을 띄고 있을지는 몰라도, 실질적인 디자인 기술이나 팁은 제법 다른 부분이 많이 있다.

아래에 소개되는 애플리케이션 개발에 대한 역사를 살펴보자.

메인프레임

컴퓨팅 역사의 시작은 메인프레임(mainframe)과 함께 하였다. 메인프레임이란 건 당대에 돈으로 살 수 있는 가장 빠른 프로세서를 탑재한 거대한 슈퍼컴퓨터 제품이다. 이 컴퓨터는 너무나도 거대하고 많은 전력과 열을 식히기 위한 냉각 비용이 너무나도 비싸서 대기업조차도 직원 한 명 한 명에게 할당할 수 없었다. 그래서 기업은 이런 메인프레임을 구매하여 적절한 위치에 두고 직원에게는 키보드와 모니터 정도만 달린, 거의 아무런 기능이 없는 **덤 단말기(dumb terminal)**를 메인프레임과 연결하여 업무 환경에 제공해 주었다(그림 I.1). 세월이 흐르고 결국 이런 메인프레임도 소형화되었지만, 여전히 단말기와 같은 씬 클라이언트(thin client)를 사용하는 환경은 크게 변하지 않았다.

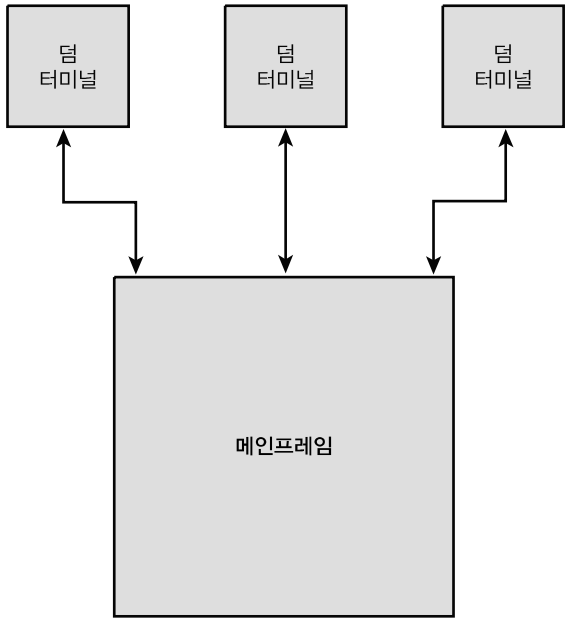


그림 I.1 메인프레임

메인프레임은 한 가지의 일을 매우 빠른 속도로 하도록 설계되었다. 따라서 한꺼번에 여러 개의 일을 하려면 시간을 나누어서 일(job)을 처리하도록 해야 한다. 일을 하다가 다른 일을 하는 방법은 각각 정해진 시간을 주고 이 시간이 끝나면 다른 일로 전환을 해 주어야 하기 때문에 시간과 일을 관리하는 스케줄러의 구조가 복잡할 수밖에 없었다. (**옮긴이_** 이러한 개념에서 발생한 것이 바로 시분할 시스템이다. 자세한 개념은 Context Switching에 대해 검색하면 알 수 있으며, 불과 십수 년 전의 도스(DOS)만 해도 이런 스케줄러를 포함하지 않았기에 한 번에 하나의 작업만이 가능했다. 도스는 메인프레임에서 사용되지는 않았다.)

이런 상황에서 클라이언트-서버 모델이 등장하게 되고, 한 번에 하나만 할 수 있었던 기존의 시스템에 획기적인 발전을 가져오게 된다. 클라이언트-서버 모델은 단말기가 **씬 클라이언트(thin client)**로 대체된 뒤에도 널리 사용된다. 씬 클라이언트라는 건 일반적인 컴퓨터에서 구동되도록 디자인된 소프트웨어로서 단순히 메인프레임에 연결된 단말기보다 많은 기능을 사용할 수 있게 되어 있는 장치를 말한다. 이런 씬 클라이언트의 대표적인 예는 현대의 거의 모든 사람이 사용하는 인터넷 브라우저다. (**옮긴이 _** 씬 클라이언트의 개념은 최근 많은 곳에 사용된

다. 이는 클라우드가 발전함에 따라 기존 PC의 컴퓨팅 파워를 예전의 메인프레임처럼 클라우드에서 처리함으로써 PC조차도 씬 클라이언트로 만들어버리게 된다. 이러한 라이프 사이클은 마치 패션이 돌고 돌듯이 컴퓨팅 환경에서도 동일한 현상이 나타난다고 볼 수 있다.)

PC의 혁명

최초의 메인프레임과 같이 최초의 PC는 요즘의 디지털 손목시계보다 보잘 것 없었다. 하지만 기술이 발전함으로 인하여 가정에서 일반적으로 사용할 수 있을 만한 크기의 컴퓨터에서 오래 전의 메인프레임과 비슷한 컴퓨팅 파워를 가지게 된다. 이 새로운 컴퓨터의 등장은 개발자에게 기존의 메인프레임 환경을 버리고 스프레드시트 프로그램과 같이 PC에서 동작하는 애플리케이션을 개발하는 환경을 제공한다. 이런 변화는 과거처럼 메인프레임과 연결되는 네트워크의 속도가 더 이상 문제가 되지 않게 하는 동시에, 사용자에게 PC가 이전의 메인프레임보다 훨씬 느리다는 사실을 깨닫게 하였다. 시간이 흐름에 따라 PC 사용자의 증가하는 요구를 반영한 소프트웨어들은 점점 더 무거워지고, 이러한 소프트웨어를 구동하기 위한 PC는 지속적으로 빠르게 발전했다. 그 결과, PC는 일반적으로 사용하는 많은 소프트웨어를 구동하기에 그 성능이 차고 넘치게 되었다. 이러한 빠른 속도의 PC 발전은 한꺼번에 여러 개의 일을 수행할 수 있는 멀티태스킹 기술을 가져온다. 멀티태스킹 이전에 소프트웨어의 속도를 빠르게 처리할 수 있는 방법은 그저 메인프레임에서와 같이 스케줄러를 더 똑똑한 상위 버전으로 업그레이드하는 방법만이 존재했었다. 하지만 이런 오래된 방법은 하드웨어 스레딩과 여러 개의 프로세서를 가진 멀티프로세서 시스템으로 빠르게 교체된다.

당시의 많은 개발자들은 멀티태스킹을 꺼려했지만, 소수의 개발자들은 이 혁신적인 방법을 사용해 소프트웨어를 개발했으며, 이들은 하드웨어의 성능을 최대로 이끌어 낼 수 있는 무지막지하게 확장 가능한 시스템을 개발하기에 이른다. 이런 상황은 또 하나의 거대한 흐름, 바로 고속 인터넷의 시대를 열게 된다. (**옮긴이 _** 십수 년 전부터 컴퓨팅 환경을 접해 온 사람들이 이미 알고 있듯이, 이전에는 현대의 플래시 게임을 구동하기조차 벅찬 PC들이 대부분이었다. 주로 게임에서의 요구와 같이 PC 사용자들은 보다 좋은 그래픽과 빠른 응답성을 소프트웨어에 요구해 왔으며, 이러한 소프트웨어를 구동하기 위해 하드웨어도 눈부신 발전을 하게 된다. 이런 발전은 이제 PC에서도 유닉스 비슷한 시스템, 즉 시분할 스케줄러를 가진 OS의 등장을 불러오게 되고, 이는 높아진 PC의 사용성과 함께 서버와 클라이언트 모두에서 필요로 하는 높은 대역폭의 네트워크 성능을 자연히 요구하는 상황을

초래하게 된다.)

고속 인터넷의 시대

네트워크의 지연이나 대역폭과 같은 고민거리들은 통신기업(Telco)들이 사업을 장악하기 시작하면서 해소되었다. 이전에는 네트워크를 통해 몇 분씩 걸리던 것들이 이제는 몇 초 또는 채 1초도 소요되지 않는다. 이제껏 없었던 대규모의 시스템이 출현하면서 인터넷이 탄생하였고, 이런 인터넷은 충분한 대역폭과 처리량을 가지고 공급된다. 이런 인터넷의 발전은 개발자들로 하여금 기존 시스템들의 아키텍처에 대해 다시 생각하게 하는 계기를 만든다. 이런 고민은 많은 사람들에게 소프트웨어는 스레드화되어야 한다는 인식을 품게 하며, 한 걸음 더 나아가 **클러스터링(clustering)**을 개발하기에 이른다.

클러스터링이란 건 병렬 프로세싱에 대한 새로운 아이디어다. 한 대의 컴퓨터에서 병렬 프로세싱을 하는 대신, 여러 대의 컴퓨터(multiple machine)에서 병렬로 프로세싱하도록 구성하는 것이 바로 클러스터링이다(그림 I.2). 그림에 있는 모든 PC들은 일반적으로 사용하는 것보다 훨씬 뛰어난 컴퓨팅 파워를 가지고 있으며, 이들은 거의 대부분의 시간 동안 아이들(idle)의 상태로 동작한다. (**옮긴이 _** idle 상태라는 건 기본적으로 자동차를 시동만 걸고 악셀 페달을 밟지 않은 상태로 이해하면 쉽다. 시스템에서의 idle이란 모든 서비스에 필요한 애플리케이션이 동작하는 상태에서 어떠한 리소스에도 부하가 걸리지 않은 상태를 의미한다.)

클러스터링의 한 가지 예를 들면, 몇 년 전 그래픽을 디자인하는 회사가 고비용의 소량 서버 구매를 고려하게 된다. 예나 지금이나 그래픽이라는 건 통상 복잡한 수의 연산을 위해 보통 업계 최고 성능의 높은 컴퓨팅 파워를 요구한다. 때문에 서버의 로컬에서 이런 그래픽 관련 작업을 한다는 건 많은 시간이 걸려서 이 회사는 색다른 방법을 모색하게 된다. (**옮긴이 _** 영화 쿵푸팬더나 아바타를 생각하면 쉽다. 영상은 초당 대략 29프레임을 가지는데(표준에 따라 다르다) 3D 그래픽으로 제작된 콘텐츠를 하나의 프레임, 즉 사진으로 만드는 연산과정을 렌더링이라 한다. 이러한 렌더링은 콘텐츠의 품질이 높으면 높을수록 더 많은 연산을 요구하게 되는데, 일반적으로 쿵푸팬더 한 프레임의 렌더링을 위해 PC를 사용하게 된다면 심한 경우 하루 이상도 걸릴 수 있다. 더군다나 이러한 하나의 프레임은 보통 수십 메가에서 수백 메가의 크기를 가지게 되므로, 90분 영상 기준으로 생성하고 처리해야 하는 양은 실로 어마어마하다고 할 수 있다.)

이 색다른 솔루션이란 건 결국 사용하지 않는 로컬 시스템의 컴퓨팅 파워를 그래픽 연산에 사용하자는 것이었다. 잡(job)을 받을 수 있는 큐, 그리고 큐에 저장된 잡을 꺼내서 처리할 수 있는 간단한 클라이언트 인터페이스를 디자인함으로써 사무실의 모든 PC가 사용되지 않는 동안 그래픽 연산을 수행하도록 하였다.

이는 쉽게 말해 사용하지 않는 PC의 자원을 그래픽 연산에 이용하자는 것이다. 그래픽 연산을 위한 잡(job)을 받을 수 있는 간단한 큐(queue)를 만들고, 사무실의 모든 PC에 이 애플리케이션을 배포하여 PC가 사용되지 않는 시간에 연산을 처리하도록 하는 기법을 구현한 것이다. 이는 더 많은 서버를 구매하는 비용을 줄이는 획기적인 방법이 된다. 이러한 분산 컴퓨팅의 아이디어는 소프트웨어를 개발하는 데 새로운 방법을 제시하게 된다. (**옮긴이 _** 저자가 설명하는 클러스터링은 일종의 분산 컴퓨팅을 모델로 하는데, 이를 설명하기 위한 가장 좋은 모델은 보잉(BOINC)을 사용한 일반 PC를 사용하는 모델이다. 이는 천체의 시뮬레이션과 같은 어마어마한 계산을 잘게 쪼개어 처리할 수 있도록 하고, 일반 PC 사용자들에게 인터넷을 통해 협력을 구하는 형태로 사용자의 동의하에 의해 계산 전용의 애플리케이션이 설치되면, PC의 프로세싱 파워 중 아주 일부분을 계산에 사용하는 구조였다. 인터넷에 연결된 컴퓨터라면 어디서든 이 클라이언트의 설치가 가능하며, 이런 식으로 모인 전 세계의 수많은 컴퓨터를 통한 컴퓨팅 파워는 천문학적인 계산이 가능할 정도로 가히 놀라운 것이었다.)

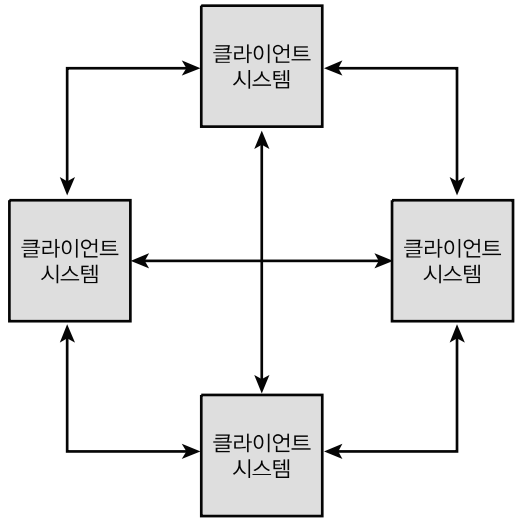
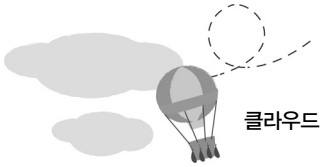


그림 1.2 클러스터 시스템



클라우드

아마존(Amazon)은 클라우드 컴퓨팅의 개념을 만들어냈다. 클라우드의 개념은 클러스터 컴퓨팅의 개념과 매우 유사하지만, 직원들의 PC를 계산에 사용하지는 않는다(그림 I.3). 고전적인 클러스터링과는 다르게 클라우드는 항상 프로세싱을 위한 자원을 확보해 두어야 한다. 비용이 조금 더 비싸기는 하지만, 필요할 때 필요한 만큼의 (가상화된) 서버를 미리 생성해둔 풀(pool) 안에서 그때그때 할당하여 사용하기 때문에 추가적인 시스템을 구매하기 위한 자본금이 필요하지는 않다. 이런 클라우드의 등장은 또 다시 사람들이 소프트웨어를 새로운 개념으로 디자인해야 한다는 것을 일깨운다. 그런데 이런 방식의 접근은 어디서 많이 본 듯하지 않은가? (**옮긴이** _ 일반적으로 pool이란 리소스의 집합을 말한다. 모뎀 시절의 전화를 사용한 컴퓨터 통신이 좋은 예인데, 당시의 아날로그 신호를 주고받는 전화선은 일단 통신을 시작하게 되면 한 회선에 하나의 통신 연결만이 가능했다. 따라서 전화선을 사용해 서비스를 공급하는 하이텔(HITEL)과 같은 서비스 공급자는 이러한 전화 통신을 받기 위해 수많은 전화회선을 집적한 모뎀 풀을 운영하여 신규로 발생하는 통신의 요청에 대응하게 된다. 따라서 이러한 풀의 처리량을 넘어가는 접속 요청이 발생하게 되면 사용자는 연결 끊김이나 끊김 이후의 재접근 시 통화 중과 비슷한 형태의 응답을 받게 된다. 따라서 하이텔은 그 이름과 특정시간대의 사용자 폭주로 인한 끊김 현상을 빗대어 '안녕전화'라는 별명을 얻기도 했다. 현대의 인프라에서는 팜(farm)이라는 용어와 혼용되는 경향이 있다.)

그렇다. 바로 메인 프레임의 구조다. 클라우드 외부에서 접근하는 클라이언트의 입장에서 볼 때 이건 메인프레임인지 클라우드인지 분간하기 힘들 정도로 비슷하다. 다만 클라우드와 메인프레임의 다른 가장 큰 차이는 한 대가 아닌 항시 컴퓨팅에 필요한 여러 대의 서버를 가진 구조라는 점이며, 이 많은 서버들의 프로세싱 파워를 어떻게 결합하느냐 하는 부분이 가장 주요한 점이 된다.

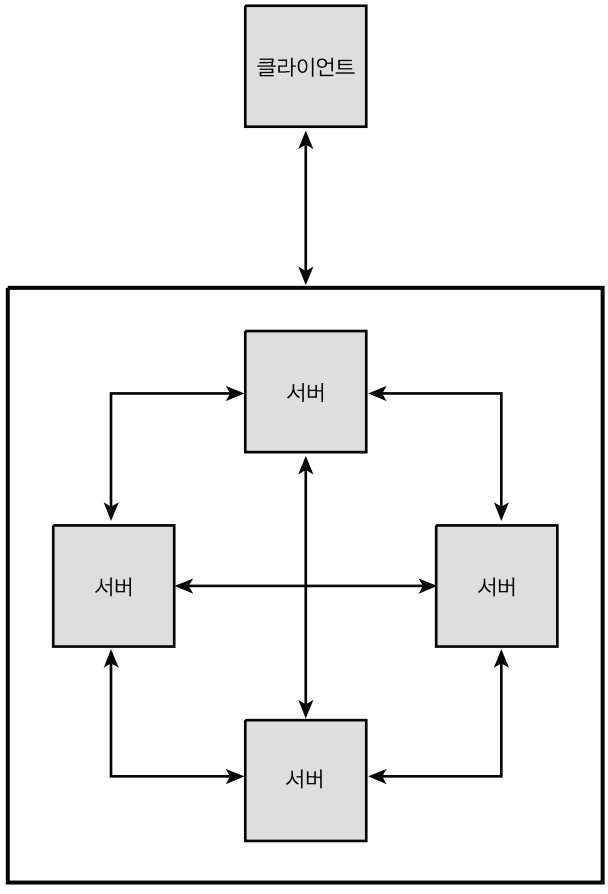
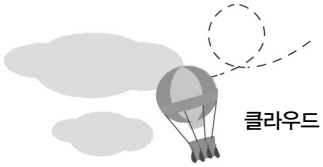


그림 1.3 클라우드 시스템

HTML5와 로컬 스토리지

하지만 이런 클라우드의 방법이 진짜 좋은 것인지, 혹은 두 대의 시스템을 하나로 묶는 새로운 기술이 존재하는 것은 아닌지 등의 의문이 있을 수 있다.

클라이언트-서버 모델에서 클라이언트와 서버의 상호작용은 웹 애플리케이션을 통해 이루어진다. 하지만 HTML5 이전의 웹 애플리케이션의 가장 큰 문제점은 서버에 대한 클라이언트



요청을 통해 모든 것이 이루어진다는 점이다. 서버가 필요할 때 클라이언트를 호출할 방법은 없었으며, 따라서 클라이언트는 로컬에서 프로세스를 처리할 방법이 전무했고, 데이터를 원하는 포맷으로 클라이언트의 로컬에 저장할 만한 공간 역시 충분치 않았다. (**옮긴이 _** HTML5 이전 웹의 이러한 특성을 가장 잘 이해할 수 있는 방법은 바로 브라우저의 '새로고침' 기능이다. 서버는 서버에 이미지나 동적으로 생성된 페이지가 업데이트되었을 때 이를 클라이언트에 알려줄 방법이 없기 때문에, 이러한 새로운 업데이트를 확인하기 위해서는 클라이언트로부터의 요청이 반드시 필요하게 된다. 따라서 주식 변동 그래프와 같이 실시간성이 요구되는 콘텐츠를 기존의 웹에 표시하기 위해서는 클라이언트로 전달하는 코드에 일정한 시간이 지나게 되면 '새로고침'을 자동으로 수행하도록 구현하는 경우가 많았다. 또한 브라우저의 저장공간이라는 것은 주로 페이지를 표시하기 위해 웹으로부터 다운받은 리소스들을 저장하거나 쿠키 등의 세션 정보를 저장하는 용도 이외의 사용성은 부여받기가 힘들었다. 이는 만약 서버에서 클라이언트로의 푸시가 불가능한 상황에서 클라이언트 로컬 저장공간에 실행코드를 동작할 수 있게 하는 경우, 보안상의 이슈가 매우 많이 발생하는 부분도 크게 작용했기 때문이다.)

서버로부터의 푸시가 가능한 HTML5의 등장과 함께 **로컬 스토리지 옵션(local storage option)**의 기능이 애플리케이션의 정보를 저장하기 위해 브라우저에 추가되었다. 이런 신기술은 서버와 클라이언트 간에 신뢰성이 필요한 데이터의 분산 처리를 가능하게 해 주었다. 웹 이외의 이런 기술을 보여주는 가장 좋은 애플리케이션은 분산 버전 컨트롤 시스템인 **머큐리얼(Mercurial)**이다.

머큐리얼은 소스 코드 저장소(repository)를 위해 특별히 서버가 필요하지 않다. 서버의 저장소를 사용하는 대신, 개발에 참여하는 사람의 로컬 컴퓨터 저장소에 모든 코드 정보를 보관하여 체크인(check-in), 리버트(revert), 업데이트(update), 머지(merge), 브랜치(branch) 등과 같은 모든 종류의 버전 컨트롤 커맨드를 오프라인에서 사용 가능하도록 지원한다. 모든 액션은 클라이언트의 로컬 저장소에서 수행되며, 다른 작업자와 결과물을 공유하려면 업데이트된 파일을 수동으로 전송해도 되고, 동기화(sync)를 처리하는 서버를 사용해도 된다. 변경된 코드의 푸시(push)나 풀(pull)을 위해서 코드를 저장하는 서버를 두어도 되지만, 이건 머큐리얼을 사용하는 데 있어 필수사항이 아니다. 로컬 소스 저장소에 아무런 문제없이 개발자가 원하는 어떠한 대상과도 push와 pull을 수행할 수 있다. 이런 모델에서 서버는 단지 인증 및 인증을 통해 허가된 사용자로부터 작성된 코드만을 받는 용도의 간단한 처리를 위해 사용한다.

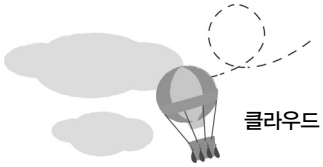
머큐리얼의 이러한 구성은 개발자가 소스 코드를 개발하고 공유하기 위해서 별도의 많은 컴퓨

팅 파워를 필요로 하지 않아도 되는 환경을 제공한다. 기존의 SVN과 같은 모델은 저장소를 가진 서버에 문제가 발생하여 접근이 불가능하게 되는 경우, 변경된 코드의 push나 pull 등이 불가능한 상태가 되어 개발을 지속할 수 없게 된다. 따라서 이러한 중앙 서버 집중적이지 않은 분산화된 처리 방식은 서버와 클라이언트 모두에게 상당한 서비스 가용성을 제공하게 되고, 클라이언트와 서버 모두에서 앞서 클러스터링에서 이야기했던 그래픽 프로세싱의 수행도 가능한 것이다. 이는 바꾸어 말하면, 머큐리얼의 코드의 변경이나 추가 내용을 포함한 데이터베이스의 조회나 검색과 같은 기능을 서버에 접근하지 않고서도 클라이언트에서 모두 그 기능을 활용할 수 있다는 것이다!

새로운 시대의 애플리케이션의 고민은 각각의 기능을 어디에 넣을 것인가 하는 것이다. 이에 대한 모범답안은 테스트와 에러를 통해 알아내는 것이다. 보안을 제외한 모든 이슈들은 어디서 어떻게 처리되어야 하는지에 대한 고정된 해법이 없다. 이건 클라이언트에서 무엇을 할 수 있는가에 달려 있기 때문이다. 예를 들어, XSLT가 네이티브하게 처리될 수 있도록 구현된 브라우저가 있는가 하면, 지원하지 않는 브라우저도 있다. 고객들에게 XSLT를 지원하는 브라우저만 사용하도록 권고할 수 있다면 좋겠지만, 모든 브라우저에 동일한 사용 환경을 제공하려면 이 기능을 서버에서 처리하도록 만들어야 할 것이다. (**옮긴이 _** 이런 서버가 필요 없는 분산 기술은, 바꾸어 말하면 클라이언트를 포함한 모든 서비스의 리소스를 서버화한다는 말과 비슷하다. 개발된 코드를 버저닝 시스템을 통해 공유하는 것은 은행에서 사용하는 애플리케이션의 계좌 동기화만큼 즉각적인 반응을 요구하는 것이 아니기 때문에 이러한 유연성을 제공한다. 이에 대한 개념은 이 책에서 나중에 보다 세부적인 이론과 함께 설명된다.)

모바일 장치들의 출현

왜 우리는 all-client 인프라에서 탈피하려고 하는 것일까? 답변은 간단한데, 그건 바로 애플(Apple)사 때문이다. 애플 이전에 블랙베리를 포함한 많은 스마트폰은 오로지 메일 정도만 확인하는 데 사용되는 대기업들의 전유물이라는 인식이 지배적이었다. 하지만 이제 모바일 스마트폰과 모바일 터치패드란 두 가지 다른 종류의 제품이 존재한다. 이런 장치들은 클라이언트-서버의 상호 관계에 대한 인식의 변화를 가져오게 된다. 이는 사용자가 원하는 순간에 언제든 아이폰(iPhone)을 놓고 아이패드(iPad)를 통해 동일한 서비스에 접근이 가능해졌기 때문이다.



영상을 서비스하는 넷플릭스(Netflix)를 살펴보자. 넷플릭스는 최근에 아이폰과 아이패드 두 장치 모두를 지원하겠다고 발표했다. 더군다나 넷플릭스는 그들이 제공하는 콘솔게임 역시 이 두 플랫폼 모두에서 지원하도록 할 것이며, 심지어는 일부 TV를 사용한 모델에서조차 가능하게 될 것이라 했다. 넷플릭스는 TV나 아이폰, 아이패드 모두 새로 구매할 의사가 없는 사용자들을 위해 일종의 스트리밍 장치를 제공하기도 한다. (**옮긴이 _** 네트워크가 연결된 셋탑박스. 하나TV나 메가TV와 비슷한 제품으로 생각하면 된다.) 넷플릭스가 제공하는 콘텐츠들의 가장 좋은 점은 이들 중 어느 장치에서든 게임이나 영화를 즐기다가 일시중지 하더라도 또 다른 장치에서 이어서 플레이가 가능하다는 점이다. 공항에서 집으로 오는 길에 스마트폰을 사용해 영화를 보다가 중지하더라도 집에 도착해서 TV를 통해 보다가 중지했던 부분부터 다시 시청할 수 있는 것이다! (**옮긴이 _** 국내에 소개된 Hoppin의 '이어플레이' 서비스와 유사한 N-스크린 개념이라 보면 된다.)

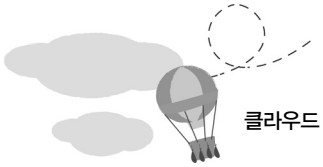
이게 무엇을 의미하냐면, 고객은 PC나 모바일 제품 중 어떤 장치를 사용하던 그 제품들에서 사용하던 콘텐츠가 부드럽게 다른 장치에서 이어진 상태로 제공받기를 원한다는 것이다. 고객은 사용하는 방법이 완전히 다른 제품을 원하는 것이 아니라 인터페이스만 다른 서로 다른 제품을 원하는 것이다. 데스크톱을 벗어나 노트북이나 아이패드를 사용하게 되었을 때도 하던 작업을 계속 하기를 원한다. 이런 방식의 작업을 위해 고객은 파일을 복사하거나 하는 작업을 서비스가 알아서 해주기를 바란다. 더군다나 인터넷이 불가능한 오프라인의 상황에서도 동일하게 사용되길 원하며, 하던 걸 계속할 수 있기를 바라고, 동기화가 가능할 때 사용자의 복잡한 조작 없이 서비스에서 간단하게 처리하기를 원한다. 이런 고객의 요구사항은 모든 개발자가 서비스를 개발할 때 항상 염두에 둬야 할 중요한 사항이다.

스레딩, 병렬 프로세싱, 그리고 병렬 컴퓨팅

거대한 규모의 애플리케이션을 개발할 때 자주 사용되는 방법은 하나의 프로세스를 여러 부분으로 쪼개서 처리하도록 하여 속도를 높이곤 한다. 통상적으로 이런 개발은 스레딩으로 가능한데, **스레딩(threading)**이란 코드를 여러 조각으로 나누어 처리하도록 구현된 언어 내부에서 제공하는 모든 기능을 지칭한다. 스레딩은 공유메모리를 사용하며, 각 스레드 간에 이 공유된

메모리 영역은 동일하게 참조되어야 한다. 이런 동일한 메모리 자원에 서로 다른 스레드가 접근할 때 고려되어야 하는 것이 바로 락(locking)과 세마포어(semaphores) 또는 비슷한 다른 방법들인데, 이것들은 모두 동일한 자원에 다수의 스레드가 접근할 때 발생하는 문제를 처리하는 기법들이다. 두 개의 스레드가 하나의 변수의 값이 0으로부터 순차적으로 증가하는 코드를 작성한다고 가정해 보자. 만약 각 스레드가 동시에 데이터에 접근하여 동일한 값을 가져다가 서로 하나씩 증가시키고 다시 저장하게 된다면, 원래 개발자가 기대했던 2라는 결과 대신 1이라는 결과를 얻는 상황이 발생하게 된다. 이는 스레드가 변수에 접근할 때 락을 정상적으로 처리하지 않을 때 발생하는 문제인데, 락이란 건 결국 데이터(변수)에 접근하여 참조하거나 변경하고자 할 때 변경이 끝날 때까지 다른 스레드가 접근하지 못하도록 방지하는 방법이다. 이런 동시접근에 대한 문제는 일반적인 코딩에서도 자주 발생하는 이슈이지만, 스레드에서는 보통 락을 처리하기 위한 방법이 컴파일러에 내장되어 있다. 스레딩은 통상 사용하는 언어(language)를 통해 코드가 빌드될 때 적용되는데, 이는 일종의 스케줄링으로서 하나의 스레드가 실행될 때 동시에 다른 스레드가 실행되도록 하는 것이 아니라, 하나가 실행되고 나서 다른 스레드가 실행될 때까지 아주 약간의 시간차를 두는 것이다. (**옮긴이 _** 스레드는 일반적으로 하나의 프로세서에서 데이터를 프로세싱할 때 병렬로 처리할 수 있도록 하는 방법이다. 만약 아파치와 같은 웹서버에서 스레드와 같은 형태로 클라이언트의 요청을 처리하지 않고 하나의 프로세스가 하나의 요청만을 처리하게 되는 경우를 생각해 보자. 이런 경우에 웹페이지에 나보다 먼저 접근한 사용자의 요청이 모두 처리되고 나서야 비로소 나의 순서가 돌아오게 된다. 또는 먼저 접근한 사용자의 요청을 처리하다가 잠깐 쉬고 내가 요청한 내용을 처리하는 식의 고전적인 방법을 사용해야 한다. 스레드는 이런 처리 대신, 클라이언트의 요청을 여러 개의 스레드에서 돌아가면서 처리하도록 하여 이전보다 많은 사용자를 처리할 수 있게 한다. 이런 방법에서 중요한 건 각 사용자의 요청을 정확히 분배해서 각 스레드에 할당해야 하는데, 만약 중복되게 되면 동일한 사용자의 요청을 중복해서 처리하는 낭비를 낳게 된다. 이런 클라이언트의 요청이 중첩되지 않도록 막는 기법을 저자는 소개하고 있다.)

또 다른 프로세싱 방법인 병렬 프로세싱에서는 스레드에서처럼 메모리를 공유하지 않는다. 언어에 내장된 스케줄러를 사용하는 스레드의 방식과는 다르게, 병렬 프로세싱은 하드웨어적으로 내장된 다수의 CPU를 사용할 수 있는 OS의 능력에 의존한다. 이런 방법의 차이로 인해 병렬 프로세싱은 단순히 변수를 공유할 수는 없으며, 파일과 같은 서버의 다른 리소스를 사용하여 데이터를 공유하는 방법으로 리소스를 공유한다. 세마포어나 락과 같은 세련된 방법을 사용할 수는 없지만, 병렬 프로세싱에서는 파일에 대한 락(file-lock)과 같은 네이티브 파일



시스템에서 제공하는 기능을 사용하여 비슷한 기능을 구현한다. 이런 방법이 비록 시간차를 두어 비동기 기법의 형태로 구현하는 스레딩보다는 조금 나을지언정, 대량의 데이터를 처리하기를 원하거나 보다 좋은 성능을 얻기 위해서는 다수의 CPU가 장착된 대형화된 슈퍼컴퓨터와 같은 컴퓨터가 필요하게 된다. 이는 한계가 있는데, 요즘의 하드웨어가 아무리 발전했더라도 12개의 버추얼코어(Virtual Core) 제품만이 가용하기 때문이다.

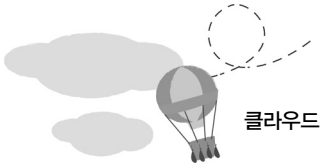
대량의 데이터를 처리하기 위한 가장 복잡한 방법은 병렬 컴퓨팅이다. 로컬 머신에서 단순히 작업을 처리하는 프로세서를 지정하고 파일 시스템을 사용하여 작업을 나누어 처리하는 대신, 메모리가 남아도는 개별 컴퓨터 여러 대를 묶어 거대한 컴퓨터로 탈바꿈하여 처리하는 것이 바로 병렬 컴퓨팅이다. 로컬 시스템에서 여분의 RAM을 저장공간의 확장에 사용할 수 있을지라도, 이러한 로컬 저장소는 병렬 컴퓨팅의 구성에서는 프로세스 간의 데이터 공유나 각 프로세스에서 장기간 보존해야 하는 데이터를 저장하도록 구성하면 안 된다. 당연히 프로세스는 각 시스템에서 독립적으로 구동되기 때문에 기존과는 다른 데이터의 공유 방법을 고안해 내야 한다. 대부분의 경우에 이런 문제는 데이터베이스를 사용하거나 공유 파일시스템을 사용하여 해결된다. 이러한 병렬 컴퓨팅의 제한사항이란 건 그저 동시에 부팅해서 사용 가능한 시스템의 수량이나 공유해야 하는 데이터를 처리할 수 있는 데이터베이스의 성능이나 대역폭 정도이기 때문에, 이런 부분을 잘 처리할 수 있다면 병렬 컴퓨팅의 확장성은 무한대에 가까울 수 있다. 병렬 컴퓨팅의 방법을 사용하게 됨으로써 비로소 슈퍼컴퓨터가 필요 없어지긴 했지만, 대신 소규모의 프로세싱 노드를 클러스터의 개념으로서 계속적으로 추가하여 확장해야 한다. 이런 방법은 서비스의 요구 또는 프로세싱해야 할 데이터의 양이 변경됨에 따라서 노드의 숫자를 줄이거나 늘리는 방법을 사용하여 유연성을 제공하며, 바로 이러한 개념이 모든 클라우드 기반 애플리케이션에서 사용되는 디자인이며, 왜 클라우드 기반에서 사용 가능한 디자인 패턴이 클러스터 컴퓨팅의 패턴과 유사한지를 나타내 준다. (**옮긴이 _** 저자는 스레드를 단일 시스템에서의 병렬 처리 모델로, 멀티 프로세싱을 병렬 컴퓨팅의 모델로 소개했는데, 이 중간에 있는 것이 프리포크(prefork) 모델이라고 할 수 있다. 이 모든 방법을 사용한 대표적인 애플리케이션은 바로 아파치에서 찾아 볼 수 있는데, 아파치의 경우 이 세 가지 모델을 모두 지원하기 때문이다. 한 대의 서버 안에서 동작할 때 아파치는 리퀘스트가 수신된 큐에 들어온 웹 리퀘스트를 처리하기 위해 스레드로서 처리가 가능할 수도 있으며, 이런 경우 리퀘스트는 하나의 단일 프로세서 내에서 공유메모리에 위치하게 되며, 현재 유효한 스레드에게 해당 요청의 처리를 넘기게 된다. 또 하나의 다른 방식은 프리포크라고 통칭되는 방식인데, 이는 아파치의 데몬을 미리 여러 개를 띄워두고, IPC(Inter Process Communication)라

불리는 방법으로 부모 프로세스로부터 자식 프로세스를 복제 생성하여 리퀘스트를 처리하기 위한 데이터를 공유한다. 마지막으로, 병렬 처리에서는 다수의 시스템에 아파치를 설치해 두고 이러한 시스템의 앞단에서 요청 자체를 분산해서 시스템에 한 번에 하나씩 분배해 주는 방식으로 서비스를 처리할 수 있게 된다. 저자는 스레딩에서 세마포어만을 설명했지만, 세마포어는 이러한 공유메모리에 대한 락 알고리즘을 지원하는 여러 가지 중 단 하나의 방법이며, 이런 공유 방법에 대해서는 리처드 스티븐(Richard Steven)의 IPC 관련 서적을 보면 보다 자세하게 참조가 가능하다. 이러한 프리포크는 일종의 스레드와 병렬 프로세싱의 중간 단계 정도에 위치하고 있으며, 스레드에서 논의되었던 락이나 세마포어 등 메모리 레벨에서의 프로세스 간 데이터 공유 기법을 사용한다.)

클라우드에서의 개발과 일반적인 개발의 차이점

일반적인 서버에서 사용되는 애플리케이션과는 다르게, 클라우드 기반 애플리케이션에서는 다른 모든 애플리케이션에서 사용되는 컴퓨팅 파워와 스토리지를 어떻게 분산해서 원하는 데이터를 처리하느냐가 관건이 된다. 데이터는 또 몇 가지의 서로 다른 레벨로 구분할 수 있는데, 바로 공유 데이터와 공유되지 않는 데이터로 나눈다. 공유되지 않는 데이터의 대표적인 사례는 바로 RAM이다. RAM은 작고, 빠르며, 접근이 가능하지만 아무때나 지워질 수 있으며, 별도의 방법을 사용하지 않는 한 서로 다른 시스템 사이에서의 공유가 쉽지 않다. 클라우드 컴퓨팅 환경이 인스턴스를 종료할 때 모든 데이터를 날려버리는 것은 아니지만, 각 인스턴스 로컬의 모든 자원은 RAM과 같다고 생각하는 방식의 접근은 굉장히 유익하다. 그리고 인스턴스에서 특정 작업의 수행이 끝난 뒤에 데이터는 아무것도 남아 있지 않을 거라고 추정하는 것이 좋다. 공유해야 하거나 또는 지속적으로 남겨두기를 원하는 데이터는 데이터베이스나 공유된 스토리지 영역에 넣는 것이 바람직하다.

각 노드에서 동작하는 프로세스가 서로 메시지를 주고받기를 원하는 경우에 로컬 파일시스템은 고려의 대상이 될 수 없다. 동작 중인 새로운 프로세스를 종료하거나 기존 프로세스의 상태를 메시지를 통해 변경을 가하고 싶은 경우에는 전체 시스템의 어딘가에 파일을 남겨두는 방법보다는 메시지 큐를 사용하는 식의 방법을 써야 한다. 만약 사용 중인 클라우드 서비스에 공급자가 제공하는 큐 서비스가 없다면(**옮긴이 _** 아마존 웹 서비스의 Simple Queue Service와 같은) 데이터베이스를 사용하여 비슷한 방법을 구현할 수도 있다.



RAM이나 각 로컬 파일시스템에서 존재하는 데이터는 일반적으로 인터넷 애플리케이션에서 사용하는 세션 데이터와 같은 형태일 가능성이 높다. 이는 기본적으로 클라우드 서비스는 인터넷을 사용하지 않는 서비스가 거의 없기 때문이다. 브라우저를 사용하지 않는 애플리케이션을 개발하는 경우에도 인터넷에서 사용되는 프로토콜에 대한 고려는 해야 한다. 세션 데이터에 대해서는 매우 중요하게 고려되어야 하는데, 만약 세션 유지를 반드시 해야 하는 경우라면 이 세션 데이터는 각 서버의 로컬이 아닌 공유 영역에 저장해야 한다. 클라우드에서 서비스를 구성할 때 세션을 각 서버의 로컬 영역에 저장한다면, 수많은 웹서버 중 어느 서버에 세션이 위치하는지 알 길이 없을 뿐더러 한 번 로그인했던 고객이 매번 동일한 서버로 접근한다는 보장도 없으며, 시스템이 계속 확장되면 누가 어느 프로세스로 접근해야 하는지 알 길이 없다. 심지어 그 프로세스가 어디서 동작하고 있는지도 모르기 때문이다. 따라서 이런 세션 유지가 필요하다면 필요한 서비스 사이의 공유 영역을 사용해야 한다. 만약 세션을 유지할 필요가 전혀 없다면 세션 상태를 유지하지 않는 방법으로 서비스를 구현해야 한다.

클라우드가 비용을 절감시켜 주고 기존 코드의 사용이 가능하더라도, 개발자는 기존의 코드를 그대로 사용하는 것보다는 클라우드로 이전하면서 변경하고 싶을 것이다. 이런 변경을 고려할 때 클라우드를 서비스 공급자가 제공하는 공유 데이터베이스, 큐 시스템, 컴퓨팅 시스템들과 같은 솔루션들을 함께 이용하겠다는 전략은 바람직하다. 모든 클라우드 서비스 제공자가 컴퓨트 클라우드(compute cloud)를 제공하며, 많은 경우에 데이터베이스와 스토리지도 함께 제공한다. 어떤 클라우드 서비스는 큐 시스템같이 프로세스들 간에 메시지를 전달할 수 있는 인프라를 제공하기도 한다.

클라우드 기반 애플리케이션을 개발할 때 또 다른 중요한 개념은 장애는 필연적으로 발생한다는 것이다. 문제가 발생한 경우에 왜 발생했는지를 서비스 정상화를 미룬 채 많은 시간을 투자해서 파악해 내기보다는 그냥 서버를 바꿔버리면 된다. 클라우드에서 새로운 서버를 준비하는 데는 채 몇 분이 걸리지 않으며, 서비스 구조 설계상에 문제가 없다면 데이터를 잃어버릴 염려도 없기 때문에 서비스에는 아주 작은 영향만 미칠 뿐이다. 이런 디자인과 테크닉을 어떻게 실제 서비스에 적용하는지에 대해서는 파트 3의 "프로젝트"에서 확인할 수 있을 것이다.

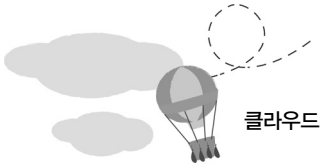
클러스터 기반 애플리케이션 개발 경험이 있다면 클라우드 기반의 개발을 위한 많은 기본지식을 가지고 있는 셈이다. 클라우드 기반의 개발과 클러스터 기반의 개발의 가장 주요한 차이점

은 완전히 새로운 것을 만드는 게 아니라 기본에 존재하는 시스템 위에서 개발한다는 것이다. 이미 프락시(Proxy)와 같은 시스템을 보유하고 있다 하더라도 많은 클라우드 서비스 공급자는 이런 기본적인 시스템을 함께 제공하고 있으므로 개발에 필요한 시간과 비용을 절감하여 줄 것이다. (**옮긴이 _** 만약 여러분이 베오울프 프로젝트와 같이 MPICH 같은 라이브러리를 사용한 클러스터 시스템이나 애플리케이션을 작성한 경험이 있다면, 동일한 시스템과 애플리케이션을 클라우드에 손쉽게 마이그레이션할 수 있을 것이다. 이 책 전체를 통틀어 클러스터링에 대한 개념이 있다면 이해가 매우 쉬울 것이나 없더라도 너무 걱정할 필요는 없다.)

피해야 할 것들

"클라우드에" 애플리케이션을 구성할 수 있다는 이야기를 듣는 순간, 사람들이 쉽게 오해하는 것들이 있다. 애플리케이션은 별다른 수정 없이 클라우드에서 안정적으로 동작할 것이고, 엄청난 확장성을 가질 것이며, 손쉽게 비용 절감을 이룰 수 있다고 믿는다. 하지만 클라우드에서 애플리케이션을 아무런 수정 없이 "구동"하는 것이 안정성과 확장성을 가져다주진 않는다. 안정성과 확장성을 위해서는 클라우드 인프라에 주어진 자원들을 사용하여 애플리케이션을 그에 맞게 재구성해야만 한다. 대부분의 클라우드 사업자가 제공하는 플랫폼은 일반 사용자도 쉽게 구할 수 있는 저렴한 수준의 컴퓨터가 클러스터의 형태로 구성되어 있다. 이는 데스크톱과 비슷한 확률로 서비스에 문제가 생길 수 있음을 시사한다. 장애에 대비하라. 클라우드 컴퓨팅의 목적은 장애를 회피하는 것이 아니라 장애 발생에 대비하고 발생한 경우 이를 빠르게 교체하는 데에 있다.

클라우드에서 가장 나쁜 구조는 한 지점의 장애가 전체의 장애를 유발하는 형태의 구조다. 이런 구조로 대표적인 것들은 서비스에 사용되는 주요한 데이터 또는 리퀘스트(request)의 분배를 한 대의 서버에서 처리하는 경우다. 또한 애플리케이션을 만들 때 가장 많이 고민해 볼 것은 부하의 집중을 어떻게 처리할 것인가 하는 부분이다. 클라우드 이전의 과거에 가장 큰 병목(bottleneck) 현상이 발생하는 지점은 물리적 서버이거나 대역폭, 아니면 비용이었다. 하지만 클라우드 컴퓨팅 시대에서는 이런 것들이 제한이 되지 않으므로 다른 어떤 부분이 병목을 유발할 수 있는지 생각해야 한다. 클라우드에서의 병목은 프로세스가 스레드 방식으로 처리될 수 없을 때, 순차적으로 증가하는 숫자를 생성할 때 발생하는 문제와 같이 주요한



부분을 단일 리소스로 중복 처리하게 되면 발생할 가능성이 높아진다. ID를 생성하는 데 있어서 절대로 순차적인 번호를 사용하지 않도록 한다. 대신에 랜덤하게 생성할 수 있는 UUID를 사용하도록 하며, 이런 UUID는 절대로 중복될 가능성이 없다. 단일 스레드가 지연을 유발하는 순차적 구조에서 고유한 ID를 생성하는 구조로의 변경은 어느 한 부분의 장애가 서비스 전체로 번지는 것을 피하고, 병목이 되는 부분을 애초에 없애는 것과 같다. 이런 구조는 요청이 증가하는 경우, 이 요청을 더 많은 스레드와 프로세스에서 처리가 가능하므로 결과적으로 무한대로 확장을 가능하게 한다. (**옮긴이 _** 이런 개념은 주의해서 이해해야 하는데, 서비스에는 순차적인 처리가 필요한 부분과 "그렇지 않아도 되는 부분"이 있다. 데이터를 전송하기 위해 쪼개진 패킷을 다시 정렬하기 위해 순번을 매겨야 할 필요는 있지만, 이 패킷이 궁극적으로 하나의 큐에 도달할 수 있다면 어느 경로를 통해서든 수신되는 순서는 중요하지 않다. 만약 수신되는 순서까지 주요하게 처리해야 한다면, 1~10까지의 패킷 중 세 번째가 누락되는 경우 이전에 도착된 4번부터 10번까지의 데이터는 무효 처리하고 재전송해야 하는 경우가 발생된다. 이러한 재전송이 바로 지연시간을 유발하며, 병목구간이 되는 것이다. 따라서 애플리케이션에서 처리되는 각 데이터의 처리 방법이나 형태에 대한 분석이 선행되어야 하며, 책의 이 부분에서 소개되는 내용들에 대해서는 시스템 수준의 다소 거시적인 관점이 일부 필요할 수 있다.)

클라우드 시작하기

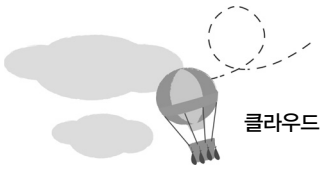
클라우드를 사용할 때 가장 먼저 해야 하는 것은 인스턴스에 사용될 이미지의 생성이다. 인스턴스에 필요한 기능을 추가하고 이를 이미지로 만드는 것은, 클라우드 기반 애플리케이션이 구동될 수 있는 가장 중요한 환경을 준비하고 이를 필요할 때마다 재사용하여 필요한 수의 서버를 만들어 낼 수 있는 클라우드의 가장 기본이며 핵심인 부분이다. 이러한 이미지에 이것저것 많이 넣는 것은 이미지를 보다 크게 만들어 인스턴스를 생성하고 시작할 때 더 느려질 수 있으나, 필요한 것을 전부 넣지 않으면 인스턴스를 시작하고 난 다음에 설정하는 데 시간을 더 소비하게 될 수 있으므로 그 구성에 신경 써야 한다. (**옮긴이 _** 이미지라는 건 일종의 템플릿으로서, 서비스를 구성하기 위해 필요한 적절한 OS와 인스턴스의 사이즈를 AMI에서 선택하여 구동시킨 이후, 여기에 서비스를 위한 기본적인 애플리케이션이나 컴포넌트를 추가로 설치하여 본인의 서비스에 맞도록 설정한다. 이를 이미지로 생성하면, 이후부터는 필요한 때에 독자적인 원본 이미지를 사용하여 인스턴스를 시작함으로써 서비스로 추가될 수 있는 시간을 최대한 줄이고 매번 생성될 인스턴스에 대한 동질성을 확보할 수 있다.) 코어 이미지에 apt나 yum, rpm과 같은 패키지 관리 시스템만

준비해 두어 이미지가 부팅하는 동안 패키지를 설치하여 인스턴스가 설정되도록 하는 방법도 있다. 또는 서비스에 필요한 인스턴스의 종류별로 이미지로 만들어 두고, 이를 각각 개별적으로 업데이트해서 관리할 수도 있다. 대부분의 클라우드 서비스 공급자는 이미지 생성 기능을 제공하며, 이미지를 통해 시작된 인스턴스가 하나만 동작하고 있을 때 특정한 인스턴스 레벨의 데이터를 설정하도록 지원함으로써 이미지를 통해 인스턴스를 생성하는 데 유연성을 부여할 수 있다. 이런 기능은 추가적인 패키지를 로드하거나, 업데이트를 수행하거나, 패스워드와 같은 보안 관련 정보를 이미지에 포함하지 않도록 설정하고 싶은 경우에도 매우 유용하다.

아마존 웹 서비스에서 이 인스턴스 데이터는 일반적으로 AWS(Amazon Web Service) 인증정보를 저장하는 데 사용된다. 따라서 각 인스턴스에서 아마존이 제공하는 다른 서비스로 접근할 때 사용이 가능하다.

다음 단계는 공유 데이터를 저장할 방법을 모색한다. 만약 프로세스가 끝난 데이터를 공유 스토리지로 업로드하기를 원하는 경우, AWS에서는 이런 용도로 사용 가능한 공유 스토리지를 제공하는데, 이 서비스가 바로 Simple Storage Service(S3) 서비스다. S3는 큰 데이터를 저장하는 데 사용이 가능하며, 역시 사용한 만큼 비용을 지불한다. 아마존은 SimpleDB(SDB) 서비스도 제공하는데, 이 서비스는 공유되어야 하는 간단한 데이터의 저장과 검색에 유용하다. 관계형 데이터베이스가 필요하다면 SDB 서비스 대신 RDS(Relational Database Service)를 사용하여 MySQL을 클라우드에서 구성하는 것도 가능하다.

데이터를 저장하는 방법을 결정했다면 이제 이 데이터들이 언제 어떻게 처리해야 하는지 구성해야 할 필요가 있다. 이는 **메시징(messaging)**의 방법으로 처리가 가능한데, 만약 Objective-C와 같은 관점지향(Aspect-Oriented) 언어를 다루어 본 경험이 있다면, 이런 백그라운드에서 동작하는 부분에 대한 프로그래밍을 이미 염두에 두고 있을 것이다. 또한 함수 호출과 같은 특정한 프로세스 요청의 전송을 위해서나, 데이터 소스에 대한 공유 처리를 위해 락을 지원하도록 **메시지 큐(message queue)**를 사용하여 다른 프로세스들이 이 큐를 참조하여 실행하도록 구성할 수 있다. 프로세스가 큐에서 메시지를 뽑아낼 때 다른 프로세스들이 중복하여 처리하지 못하도록 구현하고, 이 프로세스가 종료되면 결과값을(또는 데이터를) 다른 데이터 소스에 푸시하고 난 후에 메시지 큐에서 해당 작업을 삭제함으로써 동일한 작업이 수행되는 것을 방지해야 한다. 사용 중인 클라우드 서비스에서 이러한 부분을 지원하지 않는다면, MySQL이나



MSSQL과 같은 데이터베이스를 사용하여 간단하게 구현이 가능하다.

메시지를 어떻게 주고받는지에 대해 개념이 정리되었으면, 수많은 인스턴스에서 실행되는 프로세스들이 동일한 메시지를 수행하지 않도록 하는 락을 구현하는 것이 필요하다. MySQL이나 MSSQL과 같은 데이터베이스를 사용한다면, 테이블 또는 해당 Row를 락(lock)하는 등의 데이터베이스에서 제공하는 방법이 사용 가능하다. 락을 사용할 때는 매우 주의해야 하는데, 이는 어떤 프로세스가 공유자원에 락을 수행한 이후 문제가 발생해서 hang이 되거나 프로세스가 죽어버렸다면, 락이 풀리지 않은 상태가 되어 데드락(dead-lock) 상태에 빠지게 된다. 데이터베이스에서 이런 데드락을 제어하는 방법은 몇 가지가 있는데, 사용 중인 데이터베이스의 매뉴얼을 참조하면 된다.

관계형 데이터베이스가 아닌 대부분의 경우 락을 사용하는 것은 위험할 수 있는데, 이는 비교적 덜 정교한 로직으로 구현되었을 때 락을 사용하면 프로세싱에 엄청난 지연을 유발할 가능성이 있기 때문이다. SDB와 같은 **최종적 일관성(eventual consistency)**의 구조를 사용하는 데이터베이스를 락의 구현에 사용하려는 것은 잘못된 생각이다. 일반적으로 데이터베이스를 사용하여 락을 구현하려는 경우 데이터베이스에 요청한 락이 해당 프로세스가 요청한 락이 확실한지, 혹은 다른 프로세스가 제어하고 있는 것은 아닌지 등의 확인을 위해 다른 프로세스들은 일정 시간 동안 대기(wait)하도록 해야 한다. 이때 SDB와 같은 특성을 가진 데이터베이스의 큰 문제는 대기를 하게 되더라도 이 대기시간에 대한 제한을 둘 수가 없기 때문에 락이 안전하게 구현되지 않을 가능성이 매우 높다. 이러한 이유로 인해 즉각적인 일관성이 떨어지는 데이터베이스를 락의 구현에 사용하는 것은 심각한 문제를 유발할 수 있다. 하지만 다행스럽게도 SimpleDB 서비스는 이제 즉각적인 읽기/쓰기를 지원하는 옵션을 제공한다. 하지만 SDB에서 이 기능을 사용하게 되면 최종적 일관성이 제공하는 장점을 잃게 될 것이다. (**옮긴이 _** 여기서 설명되는 eventual consistency에 대해서는 다음 장의 CAP 이론 부분에서 자세하게 설명된다.)

클라우드 패턴의 선택

클라우드의 장점을 십분 활용 가능한 애플리케이션을 개발하기 위해서는 올바른 클라우드 패턴의 선택이 필요하다. 이는 과학이라기보다는 사실 예술에 가까운 작업으로, 이 책에 소개된 것들 중 하나를 콕 집어 선택하기 전에 각각의 패턴을 읽어 볼 것을 추천한다. 아마도 애플리케이션에는 하나가 아닌 여러 가지의 패턴이 복합적으로 필요할 것이므로, 꼭 맞는 한 가지만 고르느라 미리부터 고생할 필요는 없겠다. 파트 2에서 소개된 패턴들을 참조하면서 여러분의 애플리케이션을 개발하는 데 있어 유사한 패턴들을 확인하는 것이 좋다. 이러한 시도는 개발을 시작하는 데 좋은 시작점이 될 것이다. 또한 파트 3의 내용을 참조하여 이러한 패턴들이 실제 애플리케이션에서 어떻게 사용되는지, 어떤 게 우리의 서비스와 맞는지를 고심해 볼 수 있을 것이다.

애플리케이션의 패턴을 최대한 적용하여 분산 처리하려는 시도를 해 보는 것이 필요하다. 대부분의 애플리케이션은 7장에 소개된 클러스터링의 방법을 적용했을 때 장점을 가지게 되는데, 이는 대부분의 클라우드 애플리케이션이 7장에 보인 웹의 패턴을 따르고 있기 때문이다. 비동기 프로세싱을 필요로 하는 데이터를 다루어 본 경험이 있다면 6장을 참조한다. 클라우드의 외부에서 클라우드로 접근해야 하는 애플리케이션을 개발할 필요가 있을 때는 5장을 참조한다. 이제 막 클라우드 환경에서의 개발을 시작해서 인스턴스 및 이미지의 생성이 궁금하다면 4장을 읽는 것이 좋다.

클라우드 플랫폼의 구현

패턴을 선택했다면 선택한 패턴들을 연습해 보아야 한다. 각 패턴의 가장 마지막 부분에서는 상세한 설명과 아마존 웹 서비스에서 사용 가능한 예제 코드가 첨부되어 있고, 이 책에 소개된 코드는 거의 모든 클라우드 기반 시스템에서 쉽게 사용이 가능하다. 클라우드 서비스 공급자들은 지속적으로 성장하고 있으며, 클라우드 관련 서비스들을 결합하여 하나로 묶어 서비스를 제공하려는 시도도 계속되고 있다. boto 파이썬 라이브러리는 AWS, 구글 스토리지, 그리고 유칼립투스를 지원한다. 랙스페이스의 클라우드에도 적용하기 위해 개발이 완료되었으며, 여기에는 오픈 소스인 **오픈스택(Open Stack)** 라이브러리가 사용되었다. 책에 사용된 예제 코드는 모두 파이썬과 boto를 사용했기 때문에 수정하지 않거나 아니면 약간의 수정만으로도 수많은 클라우드 플랫폼에서 사용이 가능할 것이다.