3-1 서버 설계

서버를 개발하기에 앞서서 대략적인 설계를 해보자. 모바일 게임 서버를 만드는 것을 가정하고 생각해본다. 초기 스마트폰 게임들은 옛날 핸드폰 게임들과 크게 다른 점이 없었다. 게임을 플레이한 유저의 데이터는 전부 폰 안에 저장되어 있었다. 패키지 게임처럼 처음에 한번 게임을 구입하면 그 이후에는 더 이상의 비용 지출이 필요 없었다. 그러다 보니 구매를 해야 한다는 진입 장벽이 있었고, 이 때문에 시장의 크기는 더 이상 커지지 않았다.

스마트폰의 성능이 좋아지면서 인터넷에 쉽게 접속되자 양상은 달라졌다. 스마트폰 게임들이 온라인 게임화되기 시작한 것이다. 최근에 나오는 모바일 게임들은 대부분 무료로 다운 받아서 설치할 수 있다. 게임 개발 회사들은 게임 플레이 중에 발생하는 인앱 결제In-App Purchase를 통해 수익을 얻는다. 게임을 구입해야 설치가 가능한 게임들은 매출 순위가 낮고, 인앱 결제 방식의 부분 유료화 게임들이 매출 순위의 상위권을 차지하고 있다.

참고

인앱 결제

앱을 이용하면서 특정 기능이나 서비스에 대해 결제를 통해 사용할 수 있도록 한 시스템이다. 예를 들어 무료로 플레이할 경우에는 10분에 한번 게임을 플레이할 수 있지만, 그 보다 더 많이 하고 싶을 경우에는 돈을 지불해야 하는 것이다.

이렇게 되면서 서버의 중요성이 매우 커졌다. 게임의 데이터가 스마트폰에 있을 경우에는 유저가 마음대로 데이터 조작이 가능하기 때문에 굳이 인앱 결제를 통해 돈을 지불하지 않아도 유저가 원하는 대로 플레이가 가능한 것이다. 이를 막기 위해서는 서버에서 데이터를 가지고 있어야 하고 서버가 게임의 플레이를 제어하고 검증해야 한다.

온라인 게임 서버와 모바일 게임 서버의 다른 점은 유저수의 차이이다. 온라인 게임과 다르게 모바일 게임은 잘될 때는 유저수가 매우 빠르게 급증하고, 안될 때는 매우 빠르게 떨어진다. 온라인 게임에서는 초기에 10만명의 가입자가 들어온다면 매우 잘된 케이스지만, 모바일 게임에서의 10만명은 아주 기본적인 수치다.

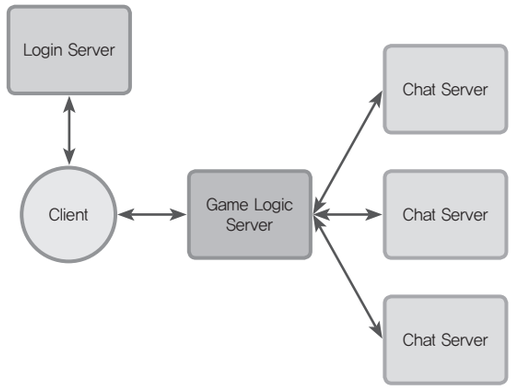
또 다른 점은 네트워크이다. 온라인 게임은 PC 기반의 이더넷Ethernet 혹은 LAN이라는 안정적인 환경만 고려하면 되므로 네트워크 플레이에 큰 문제가 없다. 모바일 게임은 Wi-Fi일 경우는 그나마 덜하지만 3G/LTE와 같은 환경에서는 접속이 끊어지는 것이 빈번하다. 온라인 게임에서 접속이 끊어질 경우 게임에서 로그아웃LOGOUT되는 것이 일반적이지만, 모바일 게임에서 접속이 끊어진다고 로그아웃 된다면 이동 중에서는 거의 게임을 즐길 수가 없을 것이다. 그리고 전송지연 또한 심하다. LAN의 경우 100ms[1] 이내의 반응 속도가 나오는데 3G/LTE의 경우는 500ms 이상의 경우도 많다. 서버에 대한 반응 속도가 빠르지 않아도 문제없이 게임이 동작하도록 구현해야 한다.

모바일 게임들의 장르적인 특성도 고려해야 할 항목이다. 스마트폰의 사양이 PC에 비해서 열악한 이유도 있고, 작은 포터블 기기의 특성상 모바일 게임들은 PC 게임에 비해서는 단순하다. 온라인 게임들의 유저당 분당 패킷 수가 수백, 수천 개씩 된다면, 모바일 게임은 1분에 패킷이 10번 정도면 많은 편이다. 패킷 수가 적다 보니 패킷의 구조는 최적화가 필요 없다. 다만 패킷의 암호화는 필수적이다. 패킷 구조가 쉬울수록 제3자가 마음대로 패킷을 수정하기도 쉬워진다. 암호화를 하면 패킷을 가로채더라도 수정된 패킷을 만들기가 어렵다.

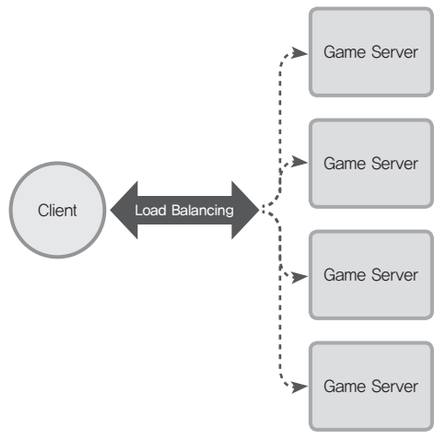
서버를 기능별로 분리하는 방법도 추천하지 않는다. 예를 들어 로그인 기능만 처리하는 로그인 서버와 채팅 기능만 있는 채팅 서버 등 이렇게 기능별로 나눌 경우, 어느 곳에서든 병목현상Bottleneck이 발생할 수 있고, 상황에 따라서 병목이 발생하는 부분이 수시로 변할 수 있기 때문에 신속하게 대응하기가 더욱 힘들어진다. 실 서비스 시에는 물리적인 서버 구성 또한 중요해지는데, DB 서버는 게임과 분리된다고 하더라도 게임의 기능들까지 분리될 경우 관리하기가 복잡하다.

가장 좋은 방법은 물리적 서버 한대에 설치된 서버 프로그램 안에 모든 게임의 기능이 다 포함되어 있어서, 사용자가 많아질 경우 별도의 복잡한 프로그램 구성을 생각할 필요 없이 똑같은 물리적 서버만 계속 추가하는 방식으로 구성하는 것이다. 훨씬 자동화하기도 쉽다.

1 millisecond(밀리세컨드 – 1/1000 초)



[그림 3-1] 모바일 서버에서는 추천하지 않는 서버 구성도
- 기능별로 분리된 서버. 특정 기능을 수행할 때마다 특정 서버에서 처리한다.



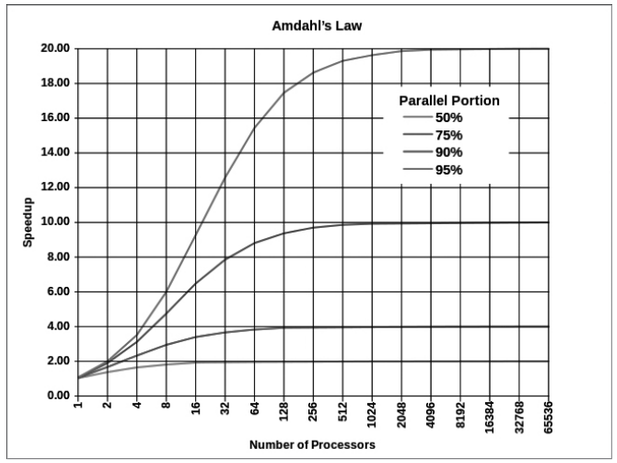
[그림 3-2] 추천하는 서버 구성도 - 모든 기능이 통합된 하나의 서버
- 어느 서버에서 명령을 수행해도 같은 결과를 얻을 수 있다. 자동적으로 서버를 늘리고 줄이기가 쉽다.

유저가 많아져서 서버에 과부하가 심하다면, Game Server를 하나 더 늘리거나, 혹은 Game Server의 사양을 더 좋은 장비로 교체하는 방법이 있다. 전자의 방식, 즉 대수를 하나 더 늘리는 것을 스케일 아웃Scale Out이라 하며, Game Server 자체의 사양을 업그레이드 하는 것을 스케일 업Scale Up이라 한다. 서버 대수를 늘리는 것은 서버 프로그램이 병렬적으로 유연하게 동작하도록 잘 설계되어 있어야만 가능하다. 대다수의 서버는 단일 서버를 가정하고 만들어져 있다. 그래서 두 가지 중 더 간단한 성능 향상 법은 스케일 업이다. 돈 더 주고, 더 좋은 서버로 바꾸면 된다, 라고 생각했을 것이다.

그런데 서버 사양만 올리면 정말 성능이 올라갈까? 요즘의 CPU 업계는 CPU Core 자체의 성능 개선은 한계에 다다랐다고 볼 수 있다. CPU Core의 Clock speed는 더 이상 올라가지 않고, 듀얼코어, 쿼드코어, 하는 식으로 주로 Core의 개수를 늘린 멀티 프로세서Multi-Processor 제품들이 나오고 있다.

CPU Core가 많으면 어떻게든 OS가 알아서 빠르게 처리해 주지 않을까? 그냥 사양만 올리면 되는 거 아니야? 라고 생각한 사람이 있을지도 모르겠다. 이에 대해서는 암달의 법칙Amdahl's Law이 대답이 될 것이다. 시스템의 일부를 개선하면 전체 시스템에서 봤을 때는 성능 향상이 얼마나 있는지를 계산할 때 사용되는 법칙인데, 이를 가지고 멀티 프로세서 환경에서 CPU 개수가 늘어남에 따라 얼마나 성능이 향상될지를 계산하는 데도 사용된다.

아래 그래프를 보면, 시스템에서 병렬로 동작하는 부분Parallel Portion이 50%일 때 최대로 낼 수 있는 성능 향상의 수치는 2배가 한계다. CPU 1개에서 시작해서 16개가 되었을 때 성능 향상이 2배가 되며, 그 이후로는 CPU 개수가 65,536개로 늘어나더라도 최대 2배에서 증가하지 않는다. 하지만 병렬로 동작하는 부분이 95%까지 되면 성능향상은 이론적으로 최대 20배까지 늘어난다. 95% 이상일 때는 CPU 개수와 비슷하게 증가한다.



[그림 3-3] http://en.wikipedia.org/wiki/Amdahl's_law, 암달의 법칙 페이지에서 발췌

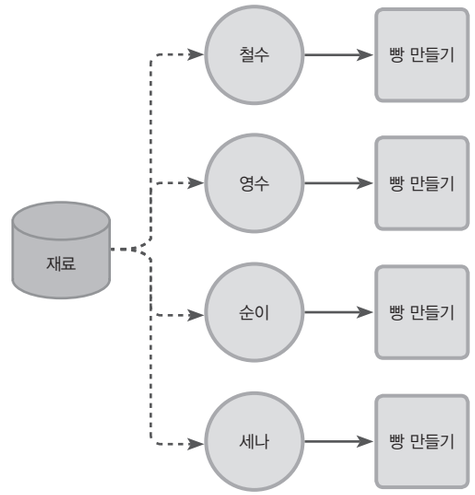
결국 프로그램의 성능을 개선하려면 **프로그램을 멀티 프로세서에 적합하도록** 만들어야 한다는 뜻이고, 그 말은 곧 프로그램의 각 기능들이 동시성Concurrency에 충족되도록 동작해야 하며, 병렬화Parallel가 가능한 구조여야 한다는 뜻이 된다.

3-1-1 Concurrency와 Parallelism

이처럼 Concurrency(동시성, 병행성)와 Parallelism(병렬성)은 서버 설계에서 매우 중요하다. 이 두 개의 개념은 비슷하지만 서로 다른 개념이다. 많은 사람들이 헷갈려 하므로 이번 기회에 명확하게 이해하고 넘어가도록 하자.

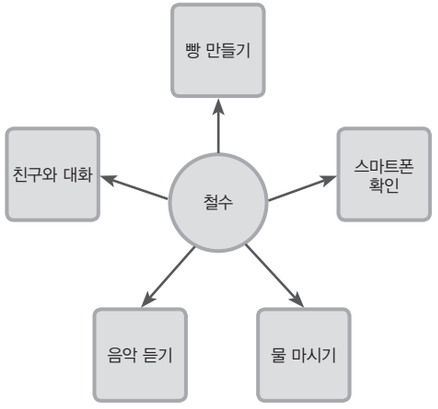
예를 들어 철수가 빵을 만든다 라고 하자. 빵을 만들 수 있는 재료는 아주 많이 쌓여 있는데, 철수 혼자서 빵을 만드니 속도가 느리다.

철수는 더 빠르게 빵을 만들기 위해서 친구들을 불러 각각 빵을 만들도록 하였다. 4명이서 빵을 만들기 시작하니 혼자서 만드는 것보다 거의 4배의 속도로 빵을 만들 수 있었다.

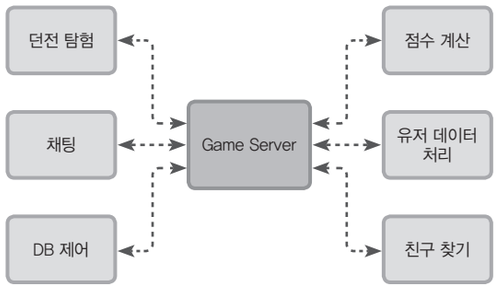


이렇게 여럿이 한 가지 작업을 동시에 진행하는 것을 병렬성Parallelism 혹은 병렬구조라고 한다. 위에서 설명한 서버 구성도와 비슷하게 보이지 않는가? **게임서버라는 하나의 기능을 수행하는 여러 대의 서버는 병렬적인 구성**인 것이다.

이제 철수 한 명에게 집중해보자. 철수는 빵을 만든다. 실제 삶에서 빵을 만든다고 생각했을 때, 철수는 빵을 만드는 일 하나만 하지는 않을 것이다. 영수가 물어보면 대답해 주기도 하고, 스마트폰에 문자가 오면 잠깐 빵 만들다가 확인하기도 한다. 빵을 만들면서 지루하니까 음악을 듣고 있고, 더워서 가끔씩 물도 마신다.



이 모든 행동은 동시에 이루어지고 있다. 빵을 다 만들고 스마트폰을 확인하는 것이 아니라 스마트폰이 울리면 바로 반응해야 한다. 이렇게 여러 가지의 일을 동시에 처리하는 것이 Concurrency다. 게임 서버의 입장에서 보자면 **Game Server 내부에서 동시에 실행되는 많은 기능들을 Concurrency**라고 생각할 수 있다.



어떤 유저는 채팅 기능을 요청하고, 어떤 유저는 친구 찾기를 요청했을 때 두 요청 모두 동시에 실행되어야 한다.

Concurrency에서 가장 중요한 것은 동시에 수행되는 기능들의 빠른 반응이다. 최근의 운영체제는 Concurrency를 잘 지원하고 있다. 파일을 다운 받으면서 웹 서핑도 하고 게임도 할 수 있다. 운영체제의 핵심인 커널kernel에서 프로세스 스케줄러가 선점형Preemption으로 동작하느냐가 관건이다(선점형에 대해서는 뒤이어 설명한다). 선점형이 아니더라도 프로세스가 동시에 실행되는 것처럼 보이도록 속임수를 쓰기도 하지만, 비선점형은 반응 속도 면에서 느릴 수밖에 없다.

얼랭은 운영체제가 프로세스들을 동시에 실행되도록 스케줄링 하듯이 얼랭 내부에 실행시킬 수 있는 수많은 프로세스들을 선점형으로 스케줄링 하고 있다. 얼랭의 설명 중에서 얼랭은 실시간 시스템Soft Real-time system이라 한 것을 기억할 것이다. 실시간 시스템의 가장 큰 장점은 빠른 반응 시간이다. 현재의 작업을 하는 와중에 더 중요한 일이 생기면 즉각 바로 중요한 일을 실행하는 것이다. 얼랭이 인기를 얻으면서 얼랭과 비슷한 방식으로 여러 개의 프로세스를 내부적으로 생성하는 모델Actor Model을 구현한 도구들[2]이 있지만 얼랭처럼 선점형 스케줄러를 갖고 실시간 시스템으로 동작하는 것은 얼랭이 유일하다. 이런 점은 기능은 같더라도 반응 속도의 차이를 가져온다.

선점형 스케줄링에 대해 간단히 알아보자. CPU의 자원은 한정되어 있는 데 반해 작업 진행이 필요한 프로세스들은 여러 개가 생길 수 있다. 이때 CPU의 자원을 어떻게 분배하느냐가 스케줄링이 필요한 이유다. 컴퓨터에서 사용자가 파일을 다운로드 받을 때 파일을 다운로드 받는 프로세스가 CPU의 자원을 할당 받아서 작업을 진행하게 된다. 이때 사용자가 뮤직 플레이를 실행시켜서 또 다른 프로세스가 생성되었다고 해보자. 비선점형Non-preemptive 스케줄링일 경우 파일을 다운로드 받는 프로세스 작업이 끝나야 음악을 플레이하는 프로세스가 CPU의 자원을 할당 받아서 작업을 진행할 수 있다. 즉 파일 다운로드가 완료될 때까지 음악이 나오지 않을 것이다. 선점형 스케줄링의 경우에는 스케줄러가 강제로 그 즉시 음악을 플레이하는 프로세스에게 자원을 할당해 주고, 상황에 따라 적당히 CPU의 자원을 분배하여 두 개의 프로세스가 작업을 진행하는 데 문제가 없도록 하게 된다. 물론 자세히 들어가면 비선점형 스케줄링에도 여러 가지 방법을 통해서 동시에 실행되는 것처럼 보이게 할 수 있다.

예를 들어 Node.js, Python Twisted[3] 같이 협력형Cooperative 스케줄링을 사용하는 경우에는 자원을 작은 시간 단위로 쪼개서 서로 양보하면서 자원을 할당받을 수 있게

2 하나만 예를 들면, 최근 Scala 기반의 Akka toolkit이 유명하다. http://akka.io/

3 https://twistedmatrix.com/trac/

하고, 구글이 개발한 Go[4]나 Haskell(GHC)[5]의 경우도 비슷한 방법을 이용해서 선점형이 아니면서도 동시에 실행될 수 있도록 하고 있다. 하지만 얼랭처럼 선점형 스케줄링을 사용한 것보다는 빠른 반응 속도를 얻을 수 없는 방식이다.

Concurrency를 구현하기 위한 방법은 여러 가지가 있다. 가장 전통적인 Low-Level 방식인 Thread와 Lock 방식은 하드웨어적으로는 가장 근본적인 방법이라고 할 수 있지만, 프로그래머가 구현하기에는 복잡하고, 버그가 발생할 수 있는 여지가 매우 많다. 얼랭도 C로 작성된 내부 소스코드를 뜯어보면 mutex나 spinlock 등이 존재한다. 하지만 얼랭을 사용하는 우리들은 신경 쓸 필요가 없다.

참고

Low-level은 수준이 낮다는 의미가 아니라, 해당 기술이 좀더 기계(CPU 등)에 가깝다는 뜻이다. 깊이가 깊을수록 좀 더 OS와 Kernel, CPU에 가까운 기술이고, 높을수록 사람에 친숙한 기술이다. Low-level일수록 좀더 복잡하고, 컴퓨터의 본질에 근접한 기술이라고 볼 수 있다.

3-1-2 얼랭의 방식

얼랭의 탄생은 전화 교환 시스템의 여러 문제점들을 효과적으로 해결하기 위함이었다. 그러기 위해서는 굉장히 많은 수의 동시 작업들을 수행할 수 있는 시스템Concurrency이어야 했고, 하드웨어의 결함이나 소프트웨어의 에러에 대응할 수 있는 결함 방지 능력Fault Tolerance이 필요했다. 특별한 프로그래밍 언어를 만들고자 함이 아니라, 여러 가지 문제를 해결하기 위한 실무적인 목적으로 만들어진 도구인 것이다.

얼랭을 최초로 만든 사람들은 최대한 가벼운 Concurrent System을 만들고자 했다. 독립적으로 동작하는 수많은 프로세스를 다룰 수 있는 시스템이면서, 서비스의 중단 없이 시스템의 업그레이드가 가능하며, 에러를 확실하게 핸들링 할 수 있는 시스템을 목표로 하였다. 그리고 이 모든 것을 쉽게 제어할 수 있도록 하이 레벨High Level 언어를 개발하는 것이 목표였다.

4 http://golang.org

5 http://www.haskell.org/ghc/

얼랭이라는 이름은 파스칼Pascal[6]이나 오캄Occam[7]과 같은 언어처럼 덴마크의 수학자 Anger Krarup Erlang[8]에서 따왔다. 얼랭 팀의 리더였던 Joe Armstrong은 얼랭을 개발할 때 패턴 매칭Pattern matching과 문법Syntax은 Prolog에서 영향을 받았고, 메세징 부분은 Smalltalk에서 영감을 받았다고 한다. 그 밖에도 영향을 받은 언어들은 ML, Miranda, Lisp, Ada, Modula, Chill 등 다양하다.

얼랭에 대해 이야기하는 사람들 중에는 액터 모델Actor model을 언급하는 경우가 있다. 얼랭이 액터 모델을 구현한 것도 아니고 실제 구현에서도 얼랭과 조금 다른 부분도 있지만[9] 얼랭의 프로세스 개념을 이해하는 데 도움이 될 것 같아서 소개하겠다. 액터 모델은 작업을 Actor들에게 나누어서 작업하는 방식이다. 아시다시피 Actor는 배우라는 뜻이다. 수많은 배우들이 연극을 공연하는 것을 상상해보자. 각자 맡은 역할에 따라서 서로 유기적으로 대사를 주고 받으며 시나리오를 진행한다.

Actor는 자기만의 작업을 가지고 있고, 자신의 작업 영역은 다른 Actor들이 침범할 수 없다. 다만 서로 메시지 교환을 통해서 정보를 주고 받는다. 앞서 설명한 철수와 친구들이 빵을 만드는 것에 비유하면 철수와 영희 같은 사람들을 Actor라고 볼 수 있다. 철수는 자기의 생각(메모리)과 업무를 가지고 있고, 영희의 생각(메모리)과 업무에 침범할 수 없다. 다만 대화(메시지)를 통해서 이런 저런 업무를 지시할 수는 있을 것이다.

얼랭에서 Actor 개념과 대응되는 것은 초경량의 프로세스light-weight process이다. 이 프로세스는 운영체제에서 말하는 프로세스와 다르게 매우 가벼운 프로세스이다. 상황에 따라서 수백, 수천, 수백만 개 이상의 프로세스를 생성하여 작업들을 처리할 수 있다. 얼랭의 프로세스는 자기만의 메모리를 가지고 있고, 다른 프로세스의 메모리 영역에 침범할 수 없다. 데이터를 전달하려면 상대방 프로세스에게 메시지를 보내고,

6 Blaise Pascal (1623~1662) 프랑스의 수학자, 철학자

7 William of Occam (1287~1347) 영국의 신학자, 철학자

8 Anger Krarup Erlang (1878~1929) 덴마크의 수학자

9 Actor model을 만든 Carl Hewitt는 2014년 논문에서 Actor model을 설명하면서 Erlang에 대한 항목에 자신이 만든 이론과 일치하는 부분과 맞지 않는 면들을 언급하였다. Actor model은 컴퓨팅 이론일 뿐이라서 실제 성능으로 생각했을 때 효율적이라고 볼 수는 없기 때문에 큰 의미는 없다. 더 자세히 알고 싶은 사람은 http://arxiv.org/ftp/arxiv/papers/1008/1008.1459.pdf를 참조하라.

해당 프로세스의 메일박스mailbox에 메시지가 도착하면 프로세스가 받아서 처리한다. 이것이 얼랭의 Concurrent Programming 방법이다. C나 Java로 구현할 때와 다르게 공유 메모리에 Lock을 걸고 풀고를 신경 쓸 필요가 없다. C++로 서버를 만든다면 Worker Thread 몇 개를 띄워놓고 모든 유저의 접속 처리를 위해 커다란 루프loop를 돌면서 처리된 데이터를 저장하는 공유 메모리 관리에 신경을 써야 하겠지만, 얼랭에서는 유저 접속자 한 명 당 프로세스 하나씩 혹은 그 이상을 띄워놓고 프로세스마다 각자의 메시지 처리 루프를 만들어 처리하게 된다.

그렇게 되면 유저 접속 숫자만큼의 프로세스가 생성되는 거 아니냐, 컨텍스트 스위칭context switching 부하가 엄청날 텐데, 라고 놀라는 분이 있을 수도 있겠다. 하지만 얼랭 내부의 프로세스는 OS의 프로세스가 아니라 초경량의 프로세스이다. 초경량의 프로세스들은 수만, 수십만 개 이상 생성하고 동작하는 데 아무런 문제가 없다. 컨텍스트 스위칭도 빠르고, 가볍게 동작한다.

얼랭으로 구현하는 Concurrency는 우리의 실제 행동과 흡사하다. 얼랭에서는 모든 것이 프로세스에서 동작한다. 프로세스를 사람에 대응하거나 게임 캐릭터에 대응해봐도 좋다. 캐릭터 하나하나 NPC 하나하나 프로세스를 생성하여 아이템을 교환할 때도 실제로 사람이 말을 주고 받고 물건을 교환하듯이 프로세스끼리 아이템 메시지를 교환하게 하여 구현할 수 있다. 게임 로직 그대로를 프로그램 코드로 구현할 수 있는 것이다. 프로그램을 작성하다 보면 실제 문제와 그것을 해결하는 코드를 작성하는 것과 아주 큰 괴리감이 있을 때가 많은데, 얼랭으로 작성하면 코드가 하는 일이 훨씬 더 명확하게 와닿는 것을 느낄 수 있다. 실제로 Erlang process를 다루는 것은 6장 유저 세션에서 공부하게 될 것이다.

얼랭의 첫번째 핵심이 Concurrency라면 두 번째는 **Fault tolerance**다. 얼랭에는 "Let it crash"라는 철학이 있다. 100% 에러가 없는 프로그램은 존재하지 않는다. 그렇다면 에러를 감추려고 하지 않고 오히려 에러가 발생하도록 놔두는 것이 버그를 찾는데 더 쉽지 않겠는가? 그래서 얼랭은 에러를 복구하는 시스템이 잘 갖추어져 있다. 어떤 프로세스에서 발생한 에러로 인해 프로세스가 죽으면, 다른 프로세스가 이를 복구하는 방식이다. 이것은 Exception을 의미하는 것은 아니다. Exception은 프로그래머가 어떻게 처리할지 이미 예상한 이벤트이고 에러가 아니다. 에러는 프로그래머가 예상하지 못한 버그이다.

얼랭의 Fault tolerance는 마치 분산 서버 시스템에서 한 서버가 죽었을 경우 다른 서버가 대체하고, Crash로 인해 죽은 서버는 빠르게 죽고, 빠르게 다시 살리는 것과 같다. 얼랭에는 시스템에 에러가 발생하는지 지켜보고 있는 프로세스가 있다. 프로세스는 물리적으로 같은 서버에 있을 수도 있고, 다른 서버에 있을 수도 있다. 여러분은 얼랭의 Fault tolerance를 통해서 영원히 다운되지 않는 서비스의 개발에 가까워질 수 있다.

3-2 기본 모바일 서버 만들기

실제 개발 작업에서 클라이언트와 연동을 위해 먼저 해야 할 것은 프로토콜Protocol을 정의하는 것이다. 여러분은 서버 프로그래머를 지망하고 있으므로, TCP/IP 스택stack에 대해서는 이미 알고 있다고 가정하겠다. 프로토콜 정의 이후에는 그에 맞추어 프로그램 코드를 작성하고 클라이언트와 기본적인 통신을 성공하면 이번 장의 임무는 완료된다.

3-2-1 컴퓨터 네트워크

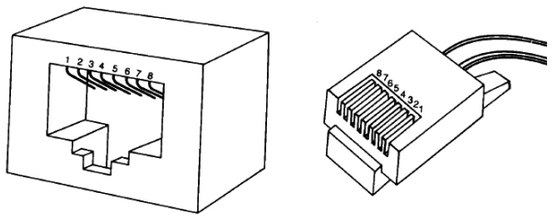
프로토콜 정의에 앞서서 네트워크에 대해 간단하게 설명하고 넘어가겠다. 컴퓨터 네트워크는 넓게 보면 컴퓨터와 컴퓨터 사이를 통신망으로 연결한 모든 것을 의미하지만, 일반적으로는 대중적으로 가장 많이 사용하고 있는 인터넷Internet 즉 TCP/IP라 불리는 네트워크를 의미한다.

OSI Model	TCP/IP	예
Application	Application	HTTP, SMTP, DNS
Presentation		JPEG, MIME
Session		Sockets
Transport	Transport	TCP, UDP
Network	Internet	IP, ICMP
Data link	Network Access	Ethernet
Physical		구리선, 광케이블

OSI 7 Layer는 네트워크 이론 수업에서 한 번쯤은 접해본 단어일 것이다. OSI 각각의 7개 계층은 위 표와 같이 TCP/IP의 4개 계층으로 분류할 수 있다.

OSI 7 Layer에서 가장 아랫단의 Physical layer(물리 계층)와 Data Link layer(데이터 링크 계층)는 구리선 혹은 광케이블을 통해 전달된 디지털 신호를 랜카드 혹은 NIC(Network Interface Card)라고 불리는 장치를 통해 컴퓨터 메모리에 패킷으로 저장하는 것까지를 의미한다.

이 부분은 IEEE 802 Network Standards에 잘 정의되어 있다. 최초에는 10Mbps에 불과했던 속도도 현재는 100Mbps, 1Gbps, 10Gbps, 40Gbps, 100Gbps까지 나온 상태이다.



[그림 3-4] Cable connector - IEEE 802.3 발췌

Network(네트워크 계층)와 Transport(전송 계층)는 OS에서 구현한 TCP/IP 스택을 뜻한다. NIC Driver에서 처리한 패킷을 실제의 TCP/IP 데이터로 처리하는 역할을 한다. 그 이후 계층은 유저가 만든 어플리케이션의 프로토콜 영역이다. TCP/IP 기반의 다양한 어플리케이션이 이 영역에 속한다.

3-2-2 프로토콜 정의

프로토콜Protocol은 전송규약이다. 클라이언트와 서버간의 데이터 즉 패킷Packet을 어떤 모양으로 주고 받을지 정하는 것이 프로토콜이다. 우리는 인터넷 환경에서 통신을 하게 되므로 TCP/IP 기반에서 동작하는 프로토콜을 작성해야 한다.

우선 IP layer에서 프로토콜을 구현하는 것을 생각해보자. Raw socket을 이용하는 방식인데, 불가능한 것은 아니지만 모바일 게임과는 맞지 않고, 가능하더라도 해야 할 일이 너무 많다. 그럼 IP layer 상단의 Transport layer에서 프로토콜을 구현해야 한다. 다시 말하면 TCP를 사용할 것인지 UDP를 사용할 것인지가 될 것이다.[10]

TCP(Transmission Control Protocol)는 connection 기반의 프로토콜로 data stream을 순서대로 전송하기 위해 만들어졌다. TCP Header를 보면 Sequence number가 있는데, Length가 없는 것을 알 수 있다. TCP는 data stream의 분실은 방지하지만 stream 기반이다 보니 하나의 메시지를 구분할 수가 없다는 것을 알 수 있다. IP의 Length만 가지고 패킷이 조립되는 것이다. 따라서 메시지를 정확히 구분할 필요가 있다면 TCP 위의 Application layer에서 따로 Length를 만들어서 관리해 주어야 한다. UDP Header에는 Length가 있어서 따로 프로그래밍 하지 않아도 메시지를 분리해서 받을 수 있다. 하지만 UDP는 connection 기반이 아니다 보니 패킷의 순서를 보장할 수 없고, 분실이 발생할 수 있다.

모바일 게임의 경우 클라이언트와 서버간의 패킷의 신뢰성을 보장해야 하므로 TCP를 선택하는 것이 맞다. UDP에 비해서 Latency가 조금 높을 수는 있지만 특별한 경우가 아니라면 큰 문제는 없을 것이다.

10 다른 프로토콜도 있겠지만 범용성을 생각하면 TCP와 UDP 밖에 없다.

참고

Latency란?

응답속도 혹은 반응속도, 지연시간 등을 의미하여, 높으면 높을수록 사용자 입장에서는 속도가 느리게 인식된다. 세분화하면 Ping latency, Connection latency, First-Byte latency 등으로 나뉠 수 있으며 서버의 성능을 측정할 때 자주 언급된다.

Transport layer에서는 TCP를 기반으로 작성하기로 하였고, 그럼 이제 Application layer에서는 어떤 프로토콜을 사용하는 것이 좋을까? 성능을 중시한다면 자체적인 프로토콜을 작성하는 것을 추천한다. 무엇을 만드느냐에 따라서 최적화된 프로토콜을 만들 수 있다. 혹은 이미 기존에 존재하는 프로토콜을 조금 변형하는 것도 좋은 방법이다.

예전에 텍스트 머드Mud[11]게임은 전부 텔넷Telnet 프로토콜 기반으로 만들어졌었다. 텔넷 프로토콜은 RFC 854[12]에 정의되어 있다. 옛날 PC 통신도 전부 Telnet을 기반으로 하고 있었다. 아주 작은 문자를 주고 받는 데 알맞지만, 데이터가 크면 효율적이지 못하고, 모바일처럼 네트워크가 자주 끊어질 수 있는 상황에서는 맞지 않다.

모바일 서버에 어울리는 프로토콜은 HTTP이다. 무엇보다도 프로토콜이 텍스트Text 기반이라서 사람이 이해하기가 쉽다. 그리고 RESTRepresentational State Transfer 같은 인터페이스 구조가 등장하면서 각종 웹 서비스의 연동에 많이 쓰인다. 구글, 애플, 페이스북, 네이버, 다음, 카카오톡 등 IT 업체들의 각종 서비스를 연동하기 위한 API들이 모두 HTTP(REST) 기반의 오픈 API로 제공되고 있다.

모바일 게임 서버는 오픈 API를 제공하기 위함이 아니므로, REST 인터페이스 구조를 따를 필요는 없다. 다만 HTTP의 장점은 그대로 이용해서 사용할 수 있을 것이다.

11 Multi User Dungeon

12 http://tools.ietf.org/html/rfc854

3-2-3 HTTP

HTTP는 Hypertext Transfer Protocol의 약자로 1.1 버전이 RFC 2616[13]에 정의되어 있다. 클라이언트는 TCP를 이용해서 해당 포트에 접속하여 요청을 보낸다, 서버는 요청에 대한 응답 코드와 데이터를 전송하고 접속을 끊는다.

웹 브라우저의 동작은 모두 HTTP를 기반으로 하고 있다.

〈클라이언트에서 웹서버로의 요청〉

```
GET /index.html HTTP/1.1
```

일반적인 브라우저의 요청은 이런 식이다. 웹 서버에 index.html의 내용을 보내 달라고 요청하는 것이다. 더 자세히 본다면 Host나 Content-Type 등의 Header가 붙어있을 것이다.

〈웹서버에서 클라이언트로의 응답〉

```
HTTP/1.1 200 OK
Date: Mon, 20 May 2014 21:00:34 GMT
Server: Apache/2.0 (Linux)
Last-Modified: Wed, 08 Jan 2014 20:00:55 GMT
ETag: "3f80a-1b2-4e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 131
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

13 http://tools.ietf.org/html/rfc2616

HTTP/1.1 다음에 200이 바로 응답 코드이다. 웹 서버는 index.html이라는 파일이 있는지 확인하고, 있다면 그 내용을 웹 브라우저로 전송한다. 파일이 없었다면 응답 코드로 404를 보냈을 텐데 아무 이상이 없었으므로 200이 전송되었다. 그 밑에 붙어 있는 Date, Server 등이 Header 부분이고, `<html>`로 시작하는 부분이 Body이다.

우리가 만들 모바일 서버는 특정한 파일을 요청 받고 그 파일을 보내주는 것이 아니라 우리가 원하는 대로 정의해서 행동하도록 할 것이다.

〈Hello World API〉

```
GET /hello/world
```

이런 요청 값이 웹 서버로 들어왔다면 /hello/world라는 파일로 접근하려고 하겠지만, 우리는 JSON JavaScript Object Notation 형식의 문자열을 전송하도록 하자. JSON은 XML보다 사이즈가 작으면서도 사람이 보기에 편한 구조로 되어 있다. 또한 JavaScript의 구문이라 웹 환경과 연동하는 데 편리하다. 최근에 많은 프로젝트에서 JSON을 기본 데이터 구조로 사용하고 있다.

여기서는 우선 응답 값으로,

```
{result: Hello world!}
```

라고 전송하도록 구현하겠다.

3-2-4 얼랭의 HTTP Server

HTTP Server로 가장 유명한 것은 Apache와 IIS가 있다. 그것들을 사용하지 않고 Erlang으로 작성된 HTTP Server(Web Server)를 사용하는 것은 어떤 장점이 있을까?

예를 들어 Apache의 경우 유저의 요청 작업을 몇 개의 worker process(thread)에서 처리한다. 메모리를 공유하기 때문에 특정 에러는 전체 웹 서버에 영향을 미칠 수 있다.

Erlang의 경우는 일반적으로 유저 한 명 당 프로세스 하나가 담당하여 처리한다. 프로세스 하나하나가 모두 웹 서버이다. 각각의 프로세스가 독립적으로 동작하기 때문에 결국 웹 서버가 독립적으로 동작한다는 의미이고, 어디서 에러가 난다고 해서 다른 프로세스에게 영향을 미치지 않는다. 동시접속자수가 수천명이라면 유저 하나의 세션을 담당하는 웹서버들이 수천 개 존재한다. **유저의 세션 하나당 웹 서버가 하나씩 동작하는 것이 상상이 가는가? 이것이 얼랭의 방식이다.**[14]

얼랭에는 자체적으로 HTTP Server가 내장되어 있다. 얼랭 모듈인 inets의 http server[15]인데, 간단한 기능만 제공하기 때문에 우리는 사용하지 않겠다. 대신 오픈소스 프로젝트 중에서 하나를 선택할 것이다.

얼랭으로 만들어진 HTTP Server는 매우 많이 있지만, 그 중 몇 개만 언급한다.

- Cowboy
  홈페이지 http://ninenines.eu/
  소스코드 https://github.com/ninenines/cowboy
- Yaws
  홈페이지 http://yaws.hyber.org/
  소스코드 https://github.com/klacke/yaws
- Mochiweb
  소스코드 https://github.com/mochi/mochiweb

Cowboy는 성능 면에서 가장 뛰어나며, Yaws는 기능이 가장 많다. Mochiweb은 단순한 것이 장점이다. 세 프로젝트 모두 엔터프라이즈 급 서버 운영에 쓰이고 있기 때문에, 어느 것을 선택해도 큰 문제는 없지만, 여기서는 향후 지속적으로 발전 가능성이 높은 Cowboy를 이용하도록 한다.

14 물론 개념적으로 그렇다는 것이지 웹 서버가 OS 차원에서 무식하게 정말 여러 개 뜨는 것이 아니다. 메모리에 웹 서버는 하나이다.

15 http://www.erlang.org/doc/apps/inets/http_server.html