18.11.15 이진명

Network

하…다신 술먹고 자료 만드는 그런 무책임한 짓 하지 않을게,,,… 미안…. 개소리 적어도 이해해줘… ❤️

Web server vs. Web application server

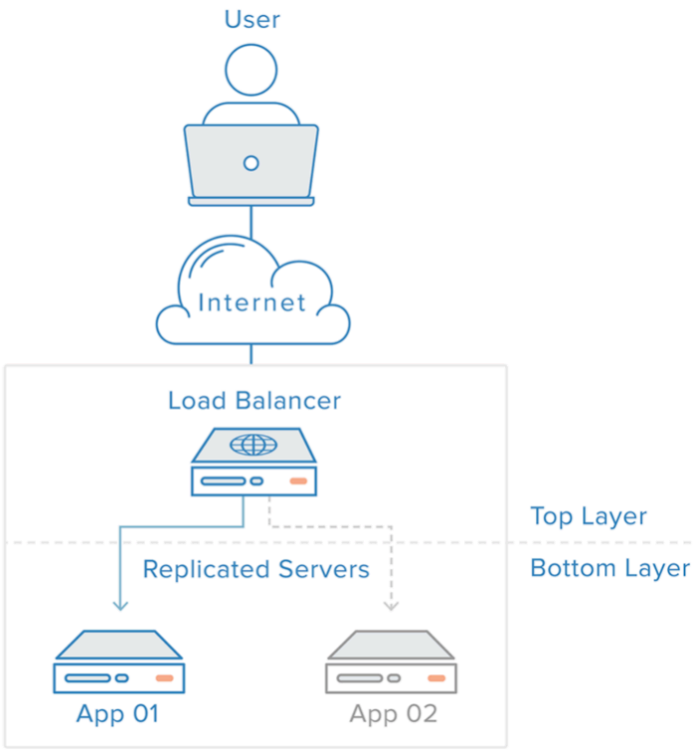
web server: 클라이언트가 서버에 페이지 요청을 하면 요청을 받아서 <정적> 컨텐츠(html, css)를 제공. 클라이언트에서 요청이 오면 가장 앞에서 요청에 대한 처리를 함. 요청이 정적 데이터인 경우 응답을 해준다.
정적 contents가 아닌 경우 WAS에 처리를 넘긴다.

Web Application Server: 동적 contents를 제공하기 위해서 만들어진 application server(DB 조회, js를 써서 하는 흔히 알고있는 그런 응답)
웹 서버로부터 요청이 오면 컨테이너가 받아서 처리 -> 컨테이너는 web.xml으르 참조해서 thread를 생성해서 response를 위한 객체를 생성하여 전달 -> 컨테이너가 서블릿 호출 -> 서블릿이 doPost(), doGet()을 호출 -> 컨테이너가 웹서버에 HTTPResponse 형태로 바꿔 웹서버로 전달하고 생성되었던 thread 종료.

그러니까! 가장 큰 차이는 동적인 처리를 할 수 있냐 없느냐! 그 차이다…!

Load Balancing

당연히 하나의 컴퓨터에 서버를 구축해서 사용하면 큰 트래픽을 감당하지 못하겠지. 그래서 컴퓨터의 하드웨어 성능을 좋게 하자! 이것이 Scale-up! 아니다. 컴퓨터를 여러개 사서 연결하자! 이것이 Scale-out.



현실적으로 Scale-up을 통해서 엄청 큰 트래픽을 감당하기 어렵고 가성비가 그렇게 좋지 않다. 그래서 Scale-out을 많이 사용하며 Scale-out을 하기 위해서는 request가 오면 알맞는 노드로 분산시켜주는 <Load-Balancer>가 필수적이게 된다.
주요 기능:
NAT(Network Address Translation): 사설 IP를 공인 IP로 바꿔줌
DSR(Dynamic Source Routing protocol): 서버에서 클라이언트로 되돌아가는 경우 목적지 주소를 스위치의 IP 주소가 아닌 클라이언트 IP 주소로 전달해서 네트워크 스위치를 거치지 않고 바로 클라이언트로 찾아감.
Tunneling: 데이터를 캡슐화해서 연결된 상호 간에만 패킷을 구별해 캡슐화를 해제할 수 있음. 통로 같은 것!

Load-Balancer의 주 목적은 동시에 오는 수많은 커넥션을 처리하고 해당 커넥션이 노드 중 하나로 전달될 수 있게 하는 것이다. 그리고 단순하게 노드를 추가하는 것으로 확장성을 가질 수 있게 하는데 있다.

load를 balancing하는 여러 알고리즘이 있는데 가장 단순한 방법은 Round-Robin이다. 예에 랜덤게임 짱.
단순하고 하나의 노드를 select하는데 overhead가 크지않기 때문에 생각보다 효율이 좋다.

여기서 조금 더 발전한 것이 Weighted Round-Robin이다. 기기의 성능에 따라 weight를 주어서 select를 할 때 더 성능이 좋은 기기에 connection을 많이 할 수 있게 해주는 것이다.

여기서 조금 더 발전하면 Least-Connection이 있다.
각 노드에 얼마나 많은 connection이 있는 지 확인하고 가장 적은 connection이 있는 노드로 새로운 connection을 할당해주는 것이다. 근데 이게 계산하는데 또 오래걸리고 그래서…흠. 여튼 그렇다.

조금 다른 접근: Source의 IP를 hashing하여 분배 -> 사용자는 항상 같은 서버로 연결되는 것을 보장받음.

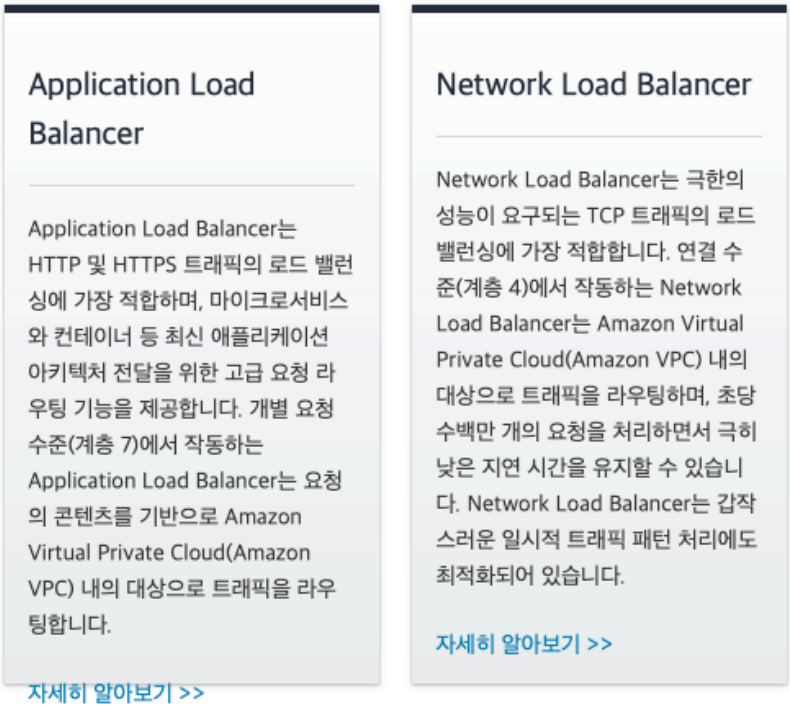
노드 몇 개만 있는 경우라면 RR이 합리적. 로드 밸런서 자체의 비용이 높기도 하고 complexity를 증가시키기 때문. 대규모의 시스템의 경우 RR과 복잡한하고 다양한 알고리즘과 스케쥴링을 사용하고 있다. 어떤 노드가 응답이 지속적으로 없을 경우 해당 노드를 제거하는 등의 신뢰성 관련한 기능을 제공하기도 한다. 그러나 이것은 또 부하가 굉장히 높은 상황에서 다른 노드의 상황을 악화시키는 등의 반작용을 일으키기도 한다.
그래서 사장 중요한 것이 모니터링을 잘하는 것이다.

오픈소스 load balancer 중 많이 쓰는게 HAProxy(http://www.haproxy.org)

종류:
L2: mac 주소 기반
L3: IP주소 기반

L4: Trasport layer(IP, port) 기반. TCP, UDP -> 스위치 가상서버가 존재하며 서버로 들어오는 패킷을 리얼 서버로 균일하게 트래픽을 분산시킴.
L7: Application layer기반. HTTP, FTP -> 웹페이지를 담당하는 웹서버와 특정 서비스에 대한 API 웹서버를 따로 운용하고 트래픽 요청에 따라서 로드 밸런싱을 해줌. L4보다 복잡한 방식이지만 그만큼 더 효율적으로 로드 밸런싱이 가능하다.

<AWS 에서 확인할 수 있는 설명>



Socket
서버나 클라이언트의 구체적인 주소를 표현하기 위해서는 주소체계, IP, Port 3개가 필요함.
이것을 소켓 주소라고 하고 소켓 주소를 담을 구조체를 sockaddr이라는 struct로 정의되어있다.

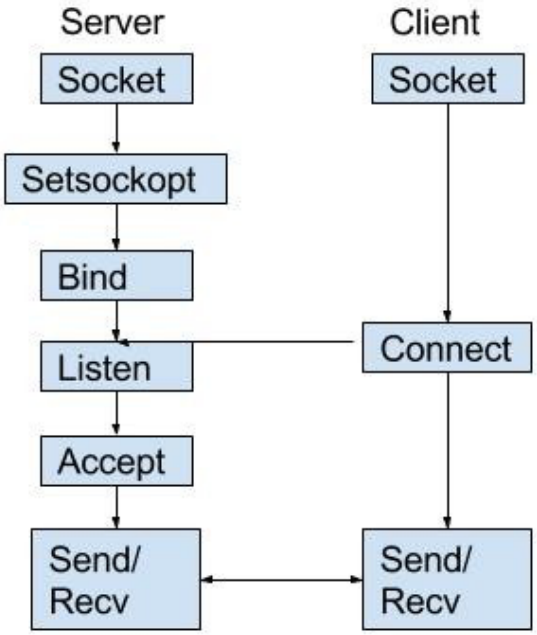
```
struct sockaddr {
        u_short    sa_family;    // address family
        char    sa_data[14];    // IP address + Port number
};
```

여기서 주소체계라는 것은 IPV4, IPV6이런 것들을 의미한다. 그런데 IP, Port를 구분하기 불편하니깐 sockaddr_in이라는 구조체를 대신 사용한다.

```
struct in_addr {
        u_long s_addr;    // 32비트 IP 주소를 저장 할 구조체
};

struct sockaddr_in {
        short    sin_family;            // 주소 체계
        u_short sin_port;               // 16 비트 포트 번호
        struct    in_addr sin_addr;     // 32 비트 IP 주소
        char      sin_zero[8];          // 전체 크기를 16 비트로 맞추기 위한 dummy
};
```

<어떻게 통신을 하는지 보여주는 그림>



Server:

```
int main(int argc, char **argv)
{
    int server_sockfd, client_sockfd;
    int state, client_len;
    int pid;
    struct sockaddr_in clientaddr, serveraddr;

    char buf[255];
    char line[255];

    // internet 기반의 소켓 생성 (INET)
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0)
    bzero(&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(atoi(argv[1]));

    state = bind(server_sockfd , (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
    state = listen(server_sockfd, 5);

    while(1)
    {
        client_sockfd = accept(server_sockfd, (struct sockaddr *)&clientaddr,
                               &client_len);
        while(1)
        {
            rewind(fp);
            memset(buf, '0', 255);
            if (read(client_sockfd, buf, 255) <= 0)
            {
                close(client_sockfd);
                break;
```

```
            }

            while(fgets(line,255,fp) != NULL)
            {
                if (strstr(line, buf) != NULL)
                {
                    write(client_sockfd, line, 255);
                }
                memset(line, '0', 255);
            }
            write(client_sockfd, "end", 255);
        }
    }
    close(client_sockfd);
}
```

Client:

```
int main(int argc, char **argv)
{

    int client_len;
    int client_sockfd;

    FILE *fp_in;
    char buf_in[255];
    char buf_get[255];

    struct sockaddr_in clientaddr;

    client_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    clientaddr.sin_family = AF_INET;
    clientaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    clientaddr.sin_port = htons(atoi(argv[1]));

    client_len = sizeof(clientaddr);

    connect(client_sockfd, (struct sockaddr *)&clientaddr, client_len)
    while(1)
    {
        printf("지역이름 입력 : ");
        fgets(buf_in, 255,stdin);

        buf_in[strlen(buf_in) - 1] = '0';
        write(client_sockfd, buf_in, 255);
        while(1)
        {
            read(client_sockfd, buf_get, 255);
            if (strncmp(buf_get, "end", 3) == 0)
                break;

            printf("%s", buf_get);
        }
    }

    close(client_sockfd);
    exit(0);
}
```