컴네 퀴즈

Application layer

- OSI에서는 7계층, IP에서는 5계층에 Application layer가 존재한다.
  - Presentation layer, Session layer가 합쳐져있다.
- 이번 장에서는 Principle of network applications를 중점적으로 살펴보게 된다.

Principle of network applications

- 네트워크 app은 단순히 네트워크를 사용하는 프로그램이다.
  - 가장 오래된 email부터 웹 브라우저, sns, 게임, 유튜브, Skype 등이 모두 속한다.
- Internet Protocol Stack를 되짚어 보자.
  - Application layer는 프로그램이 실제 구현할 수 있는 프로토콜을 제공하여 프로그램들을 지원한다.
    - HTTP, SMTP, IMAP, FTP 등이 여기에 속한다.
  - Transport layer는 실제로 다른 호스트 내의 프로세스 간 통신, 종단 시스템 간의 통신을 담당한다.
    - TCP, UDP가 여기에 속한다.
  - Network layer는 데이터그램을 출발지 -> 목적지로 라우팅하고, 전송, Addressing을 한다.
    - IP, Routing protocol이 속한다.
  - Link layer는 프레임을 실제로 유, 무선을 통해 연결된 이웃 네트워크 사이에 전송한다.
    - Ethernet과 WiFi(802.11)이 여기에 속한다.
  - Physical layer는 선으로 비트를 전송한다.
    - Optical fiber, Twisted Pair 등이 이를 사용한다.
- IP를 거쳐내려오며 전송할 메시지에 헤더를 붙이는 과정을 캡슐화(Encapsulation)이라고 칭한다.
  - 각 단계의 메시지는 Payload라고 칭한다.
  - Application layer → Message

    - Transport layer → H1 + Message = segment
    - Network layer → H2 + Segment → Datagram
    - Link layer → H3 + Datagram ⇒ Frame
- 응용 프로그램은 크게 두 가지 구조를 가진다.

> 실제 통신은 네트워크 코어가 아닌, 엣지들에서 이뤄진다.
> 네트워크 코어는 프로그램이 실행되는게 아니라, 패킷을 전달하는 게 전부이다.

    - Client-Server 구조
        - 서버
            - 항상 호스트, 항상 연결되어있는 상태이다.
            - 영구적인 IP 주소를 가진다.
            - 처리할 데이터의 규모에 따라 바뀌는 데이터 센터
        - 클라이언트
            - 서버를 통해서만 간헐적으로 소통한다
            - 유동적인 IP 주소를 가진다.
            - 클라이언트끼리는 절대 연결하지 않는다.
        - Peer-to-Peer(P2P) 구조
            - 항상 켜져있을 필요가 없다.
            - 소통시 무작위로 연결된다.
            - 자가 확장성 - 피어가 늘어날 수록 성능이 좋아진다.
            - 동적인 IP를 가지고, 변경될 수 있다.
- 프로세스 통신
    - 프로세스는 호스트 내부에서 실행되는 프로그램이다.
        - Client 프로세스는 서버에 무언가를 요청하는, 통신을 시작하는 프로세스이다.
        - Server 프로세스는 요청을 기다리는, 통신을 기다리고 있는 프로세스이다.
    - 프로세스는 다른 프로세스와 통신한다.
        - 같은 호스트 내부에서는 Inter Process Communication을 통해 통신한다.

        - 서로 다른 호스트의 프로세스는 Message를 주고받으며 통신한다.
            - 이때 서로 다른 프로세스는 Socket을 통해 메시지를 주고받게 된다.
- 소켓(Socket)
    - 프로세스가 메시지를 주고받을 때 사용하는 도구이다.
        - 프로세스 하나당 하나씩 적어도 두 개의 소켓이 필요하다.
    - Transport layer와 Application layer 사이에 위치하여, 개발자가 통제하는 문이다.
        - 그 아래 계층은 OS가 통제한다.
    - 보내는 프로세스가 소켓에 데이터를 넣으면, 받는 프로세스가 소켓에 있는 데이터를 건져간다.
- 주소(Addressing)
    - 메시지를 주고받기 위해서는 프로세스가 각자의 식별자(Identifier)를 가져야한다.
    - 호스트 디바이스를 구분하는 식별자 = IP
        - IP 주소는 32비트(IPv4)로 이루어져 있다.
        - 집 주소와 비슷한 기능을 한다.
    - 데이터를 받을 프로세스를 구분하는 식별자 = 포트 번호(Port number)
        - 포트 번호는 정수이며, 오래된 기술일 수록 낮은 번호를 가진다.
        - HTTP 서버는 80, 메일 서버는 25를 가진다.
        - 집 내부의 방 번호와 비슷한 기능을 한다.
- Application layer protocol
    - 프로토콜은 여러 정보를 담고 있어야한다.
        - 주고 받을 메시지의 타입
            - request? response?
        - 메시지 문법(Syntax)
            - Header, 메시지 내부에 어떤 필드가 있는지
        - 메시지 시멘틱(Semantic)
            - 메시지의 의미, 출발지와 도착지 ip, 포트 번호 등
        - 규칙(rules)

            - 언제 어떻게 메시지를 보내고 받을지.
    - 프로토콜은 크게 두 종류가 있다.
        - Open protocol
            - TCP/IP 등 RFC 표준으로 지정된 프로토콜
        - Proprietary protocol
            - Skype 등 특정 기업이 만든 프로토콜
- Application layer가 필요로 하는(기대하는) Transport layer의 서비스(역할)
    - 서로 붙어있는 두 계층은 상호작용하기 때문에, 상부상조해야한다.
    - Data Integrity(데이터 보전)
        - 데이터를 손실 없이 전송해줬으면 좋겠다(보안과는 전혀 관련 없다).
        - Reliable data transfer = 내가 보낸 데이터가 100% 도착해야 함.
        - 그러나 이는 프로그램의 종류에 따라 변화할 수 있음.
            - 메일이나 웹 통신은 100% 보장되어야하지만, 오디오 등은 조금의 손실이 있어도 괜찮다.
    - Timing
        - Delay와 반대되는 말로, Transport layer가 적은 딜레이로 데이터를 주고받기를 원한다.
    - Throughput(처리량)
        - 사실상 최종 목적이며, 높을 수록 좋겠다.
        - less delay = more throughput
            - 일반적으로 패킷 손실 = delay가 되는 경우가 많다.
        - 하지만 이 처리량도 elastic(탄력적)일 수 있다.
            - 실시간 멀티미디어는 최소치가 정해져 있으나, 문서 전송, 메일 등의 경우에는 조금 천천히 와도 받기만 하면 된다.
    - Security(보안)
    - 이 4가지는 아주 중요하나, 모든 프로그램에서 이 4가지를 완전히 만족시키지 못하므로 적절히 타협이 필요하다.

TCP와 UDP

- 4 계층, Transport 프로토콜에는 TCP와 UDP의 두 종류가 있다.
- TCP
  - Transmission control protocol이며, 신뢰적인 전송을 추구한다.
  - 송수신자 간의 신뢰적인 전송을 보장한다.
  - 흐름 제어 = 송, 수신측의 데이터 처리 속도 차이를 해결한다.
    - 송신자의 데이터 처리 속도를 수신자보다 낮춘다.
  - 혼잡 제어 = 송신측의 데이터 전송과 네트워크의 데이터 처리 속도 차이 해결 (Bottleneck)
    - 라우터의 버퍼를 늘리거나, 버퍼가 넘치지 않게 송신자의 속도를 낮춘다.
  - 데이터 보전을 제공, 다른 건 전혀 신경쓰지 않는다.
  - 서버-클라이언트 간 연결 설정 단계를 거친다.
- UDP
  - User datagram protocol이며, 단순한 전송을 추구한다.
  - 비신뢰적인 전송 방식이다.
  - 그 어떤 것도 제공해주지는 않지만, 데이터를 빠르게 전송한다.(real-time streaming = fast!)
- TCP가 대부분 UDP를 대체했으나, 아직 UDP를 쓰는 프로그램도 존재한다.
  - 멀티미디어 스트리밍, 인터넷 전화 등은 loss-tolerant하므로 UDP를 사용하기도 한다.

Socket Interface

- 1980년도 ARPA에 의해 투자, UC 버클리에서 개발되었다.
  - 목적 : TCP/IP 소프트웨어를 UNIX로 전송하자.
- 1981년도 UNIX 사이의 통신을 위한 generic interface로 소개되었다.

Socket

- 그래서 소켓이 뭐냐?

> 네트워크를 통해 서로에게 연결(plug into)될 수 있는 인터페이스.

- 많은 프로토콜(TCP, UDP)등의 일반적인 인터페이스가 된다.
    - Application layer와 Transport layer 사이에 위치한다.
    - 프로세스와 end-end-transport protocol 사이의 문의 역할을 한다.
- 프로세스간 식별을 위해 두 가지의 식별자를 가진다.
    - IP: 32비트로, 호스트의 식별자가 된다.
    - Port: 16비트로, 궁극적인 목적지를 식별한다.

▼ 웹과 HTTP

- Application layer = os의 7번째 계층
- Web = 거미줄처럼 촘촘히 전 세계를 연결하자.
    - 연결하는 것과 이에 액세스하는 것은 다른 것이다.
- Web page는 여러 오브젝트(객체)들로 구성되어 있다.
    - 객체들은 HTML파일, 이미지, 오디오 등이 있으며, 각각 다른 웹 서버에 저장될 수 있다.
    - 이 각각의 객체들은 URL 주소를 가지고 있다.
    - Web page는 HTML 파일 기반으로 구성되어 있으며, 이 HTML 파일은 URL 주소를 통해 참조되는 여러 객체들이 존재한다.
- web의 app layer protocol = HTTP (HyperText Transfer Protocol)
    - Client-server model
        - 클라이언트가 서버에게 요청하면
        - 서버는 클라이언트에 응답한다.
        - asymmetric한 관계이다. = 서로 동격이 아니다.
    - HTTP는 TCP를 사용한다.
        - TCP는 모든 경우에 사용가능하나, UDP는 그렇지 않기 때문이다.
        - 클라이언트가 서버에 TCP연결을 소켓을 사용해 initiate한다. (포트 80)
        - 서버는 클라이언트의 TCP연결을 받아들인다.
        - HTTP 메시지(app-layer protocol message)가 브라우저(클라이언트)와 웹서버(HTTP 서버) 사이에 오간다.
        - TCP 연결이 닫힌다.(일반적으로 클라이언트가 한다)

- HTTP는 상태가 없다(stateless)
  - 서버는 과거에 클라이언트가 했던 요청을 기억하지 않는다.
  - 어떤 프로토콜이 상태를 가지는 것은 복잡하다.
    - 상태를 저장하는 것 자체가 오버헤드가 된다.
    - 만약 서버나 클라이언트가 크래시되면, 서버, 클라이언트의 상태에 대한 뷰가 비일관적이게 될 수 있다.
- HTTP 연결은 크게 두 종류가 있다.
  - 차이점은 하나의 연결에 보낼 수 있는 객체의 개수이다.
  - Non-persistent HTTP(HTTP 1.0)
    - TCP 연결이 열린다.
    - 많아봐야 하나의 객체가 TCP 연결에 보내진다.
    - TCP 연결이 닫힌다.
    - 즉, 여러 객체를 다운로드하려면 여러 연결을 만들어야한다. = 굉장히 비합리적이다.
    - 초기 HTTP가 이 방식을 사용했다.
  - Persistent HTTP
    - TCP 연결이 서버에 열린다.
    - 한 TCP 연결에 여러 객체가 보내질 수 있다.
    - TCP 연결이 닫힌다.
    - 현재 사용하는 방식.
  - Non-persistent http의 예시를 들어보면
    - 1a : 클라이언트가 연결 시작.
    - 1b : 서버가 받음
    - 2 : 클라이언트가 요청
    - 3: 서버가 응답.
    - 4: 서버가 TCP 연결을 닫음
    - 5: 응답 받은 HTML 파일에 10개의 JPEG가 있음을 알아냄.
    - 6: 10개의 각 JPEG에 대해 1 ~ 5 까지의 과정을 총 10번 반복함.

        - Non-persistent HTTP의 response time
            - RTT(Round Trip Time) : 작은 패킷 하나가 클라이언트에서 서버로 갔다가 다시 클라이언트로 돌아오는 데 걸린 시간.
            - file transmission time = 굵은 선으로 표기됨.
            - response time = 2 * RTT + file transmission time
                - 1 RTT = TCP 연결 맺는데 걸리는 시간.
                - 1 RTT = HTTP 요청과 응답.
                - file 전송 시간.
            - 보내야 하는 파일 마다 2 * RTT + file 전송 시간이 걸린다 = 겁나게 비효율.
            - 각 연결에 OS 지연이 발생한다.
            - 브라우저는 여러 참조 객체를 가져오기 위해 TCP 연결을 병렬적으로 열게 된다. = 많은 지연.
            - HTTP 1.0이 바로 이 방식이다.
        - Persistent HTTP
            - 서버가 응답을 보낸 후에도 연결을 계속 유지함.
            - 앞에 단계에서 4번째 단계를 없애버림.
            - 각 모든 객체에 대해 최소 1 RTT + 파일 전송 시간 사용해서 전송이 가능.
            - 2 * RTT + 각 파일 전송 시간 합.
    - User/server state를 유지 = 쿠키(Cookies)
        - HTTP 요청, 응답은 상태가 없음. = 프로토콜 자체를 바꿀 수는 없다.
            - 하나의 트랜잭션을 위해 다단계 HTTP 메시지의 교환이 필요하다는 개념은 없다.
            - 즉, 1번 요청하고 다음 2번 요청, 3번 요청을 진행하는 것이 아니다.
            - 모든 HTTP 요청은 독립적이다.
            - 따라서 트랜잭션이 도중에 실패해 상태를 회복할 필요도 없다.
        - 프로토콜을 새로 설계하는 것 보다는 어플리케이션 내부에서 상태를 저장하게 만들자.

        - 웹 사이트와 브라우저들은 트랜잭션들 사이의 상태를 유지하기 위해 쿠키를 사용한다.
    - 쿠키는 4가지 구성 요소를 가진다.
        - HTTP 응답 메시지 앞에 들어있는 쿠키 헤더 라인.
        - 다음 HTTP 요청 메시지 앞에 쿠키 헤더 라인을 붙인다.
        - 쿠키 파일은 유저 호스트에 저장되고, 유저의 브라우저를 통해 관리됨.
        - 웹 사이트의 백엔드 데이터베이스
    - susan이 어떤 웹쇼핑 사이트에 처음 접속하면
        - 웹사이트는 최초의 HTTP 요청을 받으면
            - susan에 대한 고유 id = 쿠키를 만들고 돌려준다.
            - id 를 통해 백엔드 DB 입장
                - 백엔드 DB에 쿠키가 저장된다.
        - 부가적인 HTTP 요청을 할때 처음 받아온 쿠키를 헤더로 사용하여 요청을 보낸다.
            - 쿠키 헤더는 susan을 식별하는데 사용된다.
    - 쿠키가 할 수 있는 것들.
        - Authorization
        - shopping carts
        - recommendations
        - user session state(Web e-mail)
    - 과제 : 어떻게 상태를 유지할 수 있겠는가?
        - 프로토콜 종점 : 여러 트랜잭션에 대한 송수신자의 상태를 유지한다.
        - 쿠키: HTTP 메시지들이 상태를 carry하게 한다.

▼ Web caches (Proxy servers)

- 목표: Origin server를 관여시키지 않고 클라이언트 요청을 만족시키자!
    - 아무리 빨라도 Origin server의 메모리에 접근하면 지연이 많이 발생함
    - 자주 찾는 데이터는 따로 저장해놓자! = 캐시!!!!!!!

- 유저(클라이언트)는 웹 캐시에다가 HTTP 요청을 함.
    - 캐시에 요청받은 객체가 있으면(Hit) ⇒ 웹 캐시가 응답함.
    - 캐시에 요청받은 객체가 없으면(Miss) ⇒ 웹 캐시는 origin 서버에 객체를 요청하고, 받아와서 다시 클라이언트에 응답함.
- 따라서 캐시는 클라이언트와 서버의 역할을 동시에 수행함!
- 일반적으로 캐시는 ISP에 의해 설치된다.
- 그럼 웹 캐시를 외씀???
    - 클라이언트 요청에 대한 응답 시간을 줄일 수 있다 = 캐시는 클라이언트와 조금 더 가까움.
    - 어떤 집단의 액세스 링크에 대한 트래픽을 줄일 수 있음.
    - 인터넷은 캐시들로 밀집해있음
        - 느린 서버라도 빠르게 컨텐츠를 제공할 수 있음.
- 캐싱 예
    - Access link rate (망)= 1.54 Mbps
    - RTT = 2sec
    - Web object size = 100K bits
    - Avg request rate : 15 / sec
        - Avg data rate to browser = 1.5Mbps = Web obj * Avg req
    - LAN utilization = .0015
        - 1.5mbps / 1 gbps
    - Access link utilization = .97 ⇒ 높은 사용률은 높은 딜레이를 만든다.
        - 1.5mbps / 1.54 mbps
    - end-end delay = internet delay + Access link delay + Lan delay = 2 sec + minutes + usecs.
- 그럼 Access link를 새로 깔아서 154Mbps로 바꾸자.
    - 돈 준나게 많이 듦. = 사기꾼이다 이거.

▼ Caching

- bottleneck이 발생 = Access link의 대역폭이 너무 낮음.

    - access link utilization이 0.97이나 됨!
    - .0097로 만들면 지연을 msecs 단위로 줄일 수 있음.
    - 그럴 경우 종단간 지연이 (2 sec +minutes + usecs) 에서 (2 sec + msecs + usecs)로 변화함.
- 그러면 더 비싼 망을 사면 되는거 아님?
    - 100배 빠른 망으로 바꾸는데 비용이 천문학적으로 발생!
    - T-1망이 1.54Mbps
    - t1 → t3(45mbps) 로 바꾸는데도 돈이랑 시간이 미친듯이 걸림
- Web Cache를 사용해서 100배 빠르게 하자!
    - 캐시는 컴구에서 배웠던 바로 그것.
    - 비용을 거의 쓰지 않고 속도를 빠르게 만들 수 있다.
    - = 프록시 서버!
- 캐시 히트율이 0.4라고 가정하자.
    - 40%의 요청은 캐시 선에서 처리하고, 나머지 60%는 기점 서버까지 왕복한다.
    - 즉, 모든 요청의 60%만 Access link를 사용하면 된다.
    - 총 요청 1.5Mbps에서 60%인 0.9 Mbps만 링크를 사용한다.
        - 즉 0.9 / 1.54 Mbps로 링크 사용율은 .58이 된다.
    - 평균적인 종단간 지연
        - 0.6 * 기점 서버까지 왕복하는 지연(2 sec + msec + usec)
        - + 0.4 * 웹 캐시에서 반응하는 지연(~msecs)
        - = ~1.2 secs
        - 이는 154Mbps link를 새로 까는 것 보다도 훨씬 빠름!!

▼ HTTP/2 (RFC 7540)

- 목표 : 여러 객체에 대한 HTTP 요청의 지연을 감소시키자.
- HTTP 1.1의 후속작임
- HTTP 1.1은 persistent TCP 연결을 지원했음.
    - 여러 파일을 하나의 연결만 사용해서 보낼 수 있지만,

    - 한번의 요청-응답에 파일 하나만 보낼 수 있었음.
    - 즉, 다수의 요청을 FCFS으로 pipelined하여 처리함.
- 그럼 HTTP 1.1이 뭐가 문제였냐?
    - HOL(head on line) blocking
    - 큰 객체가 FCFS의 앞에 존재하면, 뒤에 있는 작은 객체들은 큰 객체가 처리될 때 까지 한 없이 기다려야됨.
    - 특히나 TCP 세그먼트가 손실되어 재전송하는 loss recovery등이 객체 전송에 의해 stall됨.
- HTTP/2의 핵심 개념 (2015)
    - 클라이언트에 객체를 전송하는 서버의 유연성을 증가시킨다.
        - 메소드나 상태 코드, 헤더 필드등은 대부분 HTTP 1.1을 그대로 사용한다.
            - 메소드 = GET, POST, , PUT, DELETE, HEAD
            - 상태 코드 = 200, 400, 404, 505...
        - 요청 받은 객체의 전송 순서는 클라이언트가 정한 우선순위에 기반한다 (꼭 FCFS일 필요는 없다!)
        - 객체를 프레임 단위로 쪼개서, 각 프레임을 스케줄한다.
            - HOL 블로킹을 완화한다.
    - FCFS를 유지하면서 RR(Round Robin) 사용
        - 요청을 프레임 단위로 조개서 처리하자.
    - 전체 처리 시간이 줄어드는게 아니라, 반응성이 높아진다.
    - 패킷 로스에 대한 회복은 여전히 모든 객체 전송을 stall시킨다.
        - HTTP 1.1처럼 브라우저는 여러 병렬 TCP 연결을 통해 스톨링을 줄이고 전체적인 처리율을 올릴 수 있다.
    - 기본 TCP 연결에서는 어떤 보안도 제공하지 않는다.
- HTTP/3
    - HTTP/3는 보안을 추가하고, UDP를 통한 혼잡제어(Congestion control)과 오브젝트당 에러를 통제한다.
        - 혼잡제어는 송신측의 데이터 전송과 네트워크의 데이터 처리 속도가 차이나는 것을 해결한다.

        - 라우터의 버퍼를 키우거나, 송신측의 속도를 낮춘다.
    - HTTP = TCP(+ TLS)라고 생각했는데 UDP가 등장
    - HTTP/3 = UDP(+ QUIC)
        - IP는 Network 계층의 기반 (모든 HTTP에서 공통으로 씀)
        - QUIC = UDP based Mux and secure transport
            - RFC 9000
            - 새롭게 등장한 프로토콜

▼ 복습

- 1.54 Mbps가 bottleneck이 되므로, 100배 빠른 154mpbs로 바꾸려면 돈이 많이 든다.
- 따라서 캐시에 해당하는(Webcache) 서버를 하나 둠으로써 옛날 네트워크를 그대로 사용해도 속도가 개선됨.
- LLM → LMM

▼ DNS

- Domain name System.
- 80년대 개발됨.
- ip주소와 호스트의 이름을 사람이 알아보기 쉽게 도메인으로 맵핑한 분산 데이터베이스.
    - ip 주소는 dotted decimal (154.00.00.00)
    - 소켓 프로그래밍에서는 ip주소는 무조건 32 bit binary로 변환함.
    - 많은 name servers(.org, .edu...) 들의 계층 구조를 통해 구현된 분산 데이터베이스.
    - Application-layer protocl = 호스트와 name server는 이름을 확인하기 위해 통신한다 (Address - name translate)
        - 도메인 이름을 IP 주소로 변환시켜 준다.
- application layer protocol이지만, 응용 프로그램을 위한 프로토콜이 아니다.
    - core internet function이 app-layer protocol로 구현된다.
    - 네트워크의 엣지에 구현한다.

        - 네트워크 엣지의 복잡함이 여기서 나온다.
- ICANN → 도메인 주소(Ip 주소 + 도메인 이름)를 할당함.
    - 우리나라에선 인터넷 진흥원 → KISA?
- DNS services
    - 호스트 이름을 IP 주소로 변환시켜 준다.
    - 호스트 앨리어싱?
        - 도메인 이름도 복잡할 경우, 별명만 사용해서 도메인 이름을 알게 해준다.
    - 메일 서버 앨리어싱
        - 메일 서버도 복잡한 경우, 별명을 통해 정확한 메일 서버를 알려준다.
    - Load distribution
        - 접속자가 많아 여러 서버가 존재할 때 = 도메인 네임과 ip주소가 1 : 1이 아닌 경우 클라이언트의 요청을 분산시킨다.
- 그럼 왜 중앙화(Cetralized) 하지 않고 분산시킨 여러 DNS를 쓰는가?
    - single point of failure : 하나의 DNS가 동작하지 않을 경우 모든 서버가 동작하지 않음.
    - traffic volume : 트래픽 양이 너무 많음.
    - distant centralized database : 하나의 데이터베이스만 있으면, 먼 거리에서 통신할 때 너무 많은 지연이 생김.
    - maintenance : 유지 보수가 어렵고
    - 가장 중요한 것은 확장할 수 없기 때문.
- 일종의 트리 구조 = 분산되고, 계층화된 구조
    - 루트 DNS 서버가 루트에 위치한다.
    - Top level domain (.org, .com, .edu 서버들 + kr, uk, jp 등) = TLD
    - Authoritative servers (google.com, korea.edu...)
- 루트에서 직접 IP 주소를 알려주는 것이 아니고, 하위 계층의 서버를 알려주고, 그 서버는 또 하위 계층의 서버를 알려준다 (포워딩)
    - root → TLD → Authoritative
- DNS는 DB에 저장되어있다.
    - 쿼리를 통해 가져온다.

    - 클라이언트는 루트 서버에 쿼리하고 루트 서버는 TLD 서버를 알려준다.
    - 다시 클라이언트는 TLD 서버에 쿼리하고, TLD는 Authoritative server를 알려준다.
    - 다시 클라이언트는 Auth server에 최종 주소의 IP를 알려달라고 쿼리한다.
- DNS : root server
    - 이름을 알 수 없는 서버들의 최종 종착지이다.
    - 인터넷의 작동에 있어 엄청나게 중요한 역할을 한다.
        - 없으면 인터넷이 작동하지 않음!
        - DNSSEC = DNS에 security를 제공한다.
    - ICANN(Internet Corporation for Assigned Names and Numbers)에서 root DNS domain을 관리한다.
        - 총 13개의 루트 서버가 전 세계에 존재하며
        - 이 서버들은 복제되어 사용된다.
- TLD(Top-Level Domain) servers
    - .com, .org, .net, .edu 등의 도메인과
    - .kr, .jp 등의 국가 도메인들을 담당한다.
- Authoritative DNS servers:
    - 각 조직이 가지는 DNS 서버가 된다.
    - 서비스 제공자나 조직이 관리한다.
- Local DNS name servers:
    - 꼭 계층구조에 포함될 필요는 없다.
    - 각 ISP (통상적 ISP, 회사나 대학) 등이 하나씩 가진다.
        - default name server라고도 한다.
    - 호스트가 DNS 쿼리를 만들면, 이 쿼리는 먼저 Local DNS Server로 보내진다.
        - 최근 발생한 유효한 이름-주소 변환 쌍을 저장하는 로컬 캐시의 역할을 수행한다.
        - 없으면, 계층구조로 포워딩한다.
- IP 주소 결정 방식

    - 반복적인 쿼리(iterated query) = 책임 회피
        - 야. 난 모르겠는데 저 서버에 물어보는건 어때?
        - 호스트 → 로컬 DNS → Root → 로컬 → TLD → 로컬 → Auth → 로컬 → 호스트로 응답.
    - 재귀적인 쿼리(recursive query) = 책임 짊어지기
        - 알았어 내가 찾아올게.
        - 호스트 → 로컬(내가 찾아볼게) → root(내가 찾을게) → TLD → Auth(야 찾았어) → TLD → root → 로컬 → 호스트
        - 상위 레벨의 서버에 많은 부담이 가해짐 = 보통은 반복적인 쿼리를 사용함.
- 어떤 서버든, 한번 매핑을 진행하면, 이를 캐싱한다.
    - TTL 만큼의 시간이 지나면 이 캐시는 없어진다.
    - TLC 서버들은 일반적으로 local name server에 캐시되어있다.
        - 따라서 root 서버를 많이 방문하지 않아도 된다.
- 그런데 TTL이 지나기 전에 맵핑이 바뀌면(name-ip address) 캐싱이 쓸모없어진다.
    - RFC 2136 에는 이를 해결하는 업데이트/알림 표준이 제정되어있다.
- DNS 보안
    - DDoS 공격
        - 루트 서버에 엄청난 트래픽을 쏟아붓는 경우
            - 지금까지 성공한 적이 없다.
            - 트래픽 필터링
            - local DNS가 TLD를 캐싱하고 있으면, 루트 서버를 우회할 수 있기 때문에
        - TLD에 대한 DDoS
            - 이게 조금 더 위험할 수 있다.
    - Redirect 공격
        - DNS 쿼리를 도중에서 가로챈다.
        - DNS를 오염시킨다.
    - Exploit DNS for DDoS

        - 가짜 주소로 대상 IP에 쿼리를 전송한다.
    - 위 두 개는 DNSSEC (RFC 4033) 통해 방지한다.

▼ 소켓 프로그래밍

- 소켓의 역사
    - 소켓은 한마디로 API라고 할 수 있겠다.
    - 1981년 TCP/IP가 처음으로 표준으로 등장하면서 소켓도 네트워크에서 UNIX to UNIX 사이의 통신을 위해 등장했다.
        - UNIX BSD 4.2
            - BSD = UC 버클리 버전, 무료버전?
        - IPC
    - 1985년 SunOS 라는 UNIX기반 OS가 시장을 장악했다.
        - 소켓 대신 사용할 NFS, RPC가 등장했다.
    - 후에 MS에서는 Winsock을 만들어 사용했다.
- W. Richard Stevens = 바이블을 만들었다.
- 목적 : 클라이언트 / 서버 어플리케이션들이 서로 소켓을 통해 소통할 수 있게 하자.
- application-layer와 Transport layer 중간에서 작동하는 유저 인터페이스.
    - 어플리케이션 프로세스와, 종단간 전송 프로토콜 사이의 문.
    - UC 버클리에서 82년도 실질적으로 UNIX간 통신을 위해 개발.
- 32비트 IP 주소와 16비트 포트 넘버를 통해 프로세스를 식별한다.
- 인터넷 연결 (TCP/IP)
    - 32 비트 IP 주소로 machine을 주소 할당
    - 16 비트 포트 넘버로 프로세스를 주소 할당
        - 웹 서버의 포트 넘버는 80으로 정해놓았다.
        - 클라이언트의 포트 넘버는 임의로 지정할 수 있다.
    - IP + port = socket-address가 된다.
- API들이 필요한 것?
    - 연결 없는 UDP
        - 소켓 만들기

        - 데이터 보내고 받기
        - 소켓 식별하기
        - 소켓 닫기
    - 연결을 먼저 맺어야하는(Connection-Oriented) TCP
        - 소켓 만들기
        - 연결을 정립하기
            - 클라이언트를 서버에 연결
            - 서버는 클라이언트의 요구를 수용
        - 데이터 보내고 받기
        - 소켓 식별하기
            - local 주소/포트를 묶어서 식별하게 만듦.
        - 소켓 닫기.
- IP datagram format
    - TCP = 20bytes, IP = 20bytes
    - 32 bit source IP 주소와
    - 32 bit destination IP 주소만 정해주면 됨.
    - 이 안에 TCP나 UDP 세그먼트가 들어가있음(data)
        - UDP 는 32비트 세그먼트
            - source port #, dest port #만 정해주면 됨.
        - TCP 도 32비트 세그먼트지만
            - 더럽게 복잡함!!!
            - 일단은 source port와 dest port 넘버만 지정해주면 된다!
- UDP 소켓
    - Unreliable datagram(바이트 그룹)
    - UDP 자체가 간단하므로 단순한 소켓으로 구현 가능
    - 클라이언트와 서버간의 연결이 없음(no handshaking = no connection)
    - 단순하게 출발, 도착지의 IP 주소와 포트번호만 정해주면 된다.
    - 송신한 데이터가 손실될 수 있고, 순서가 어긋날 수도 있다.

        - = 클라 서버간 비신뢰적인 통신을 진행한다.
    - 서버는 계속 실행되며 서버 소켓을 계속 대기하고 있는다.
        - bind를 통해 소켓에 서버 포트를 묶어준다.
        - AF_INET = 인터넷 프로토콜 패키지(IPv4).
        - SOCK_DGRAM = UDP, datagram을 받는다.
        - while True를 통해 계속 전송을 받는다.
            - 전송받은 데이터그램에서 클라이언트의 IP주소를 알아내서 응답한다.

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)        # create UDP socket
serverSocket.bind(('', serverPort))               # bind socket to local port number 12000
print ("The server is ready to receive")
while True:                                       # loop forever
    message, clientAddress = serverSocket.recvfrom(2048)   # Read from UDP socket into message, getting client's address (client IP and port)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),          # send upper case string back to this client
                        clientAddress)
```

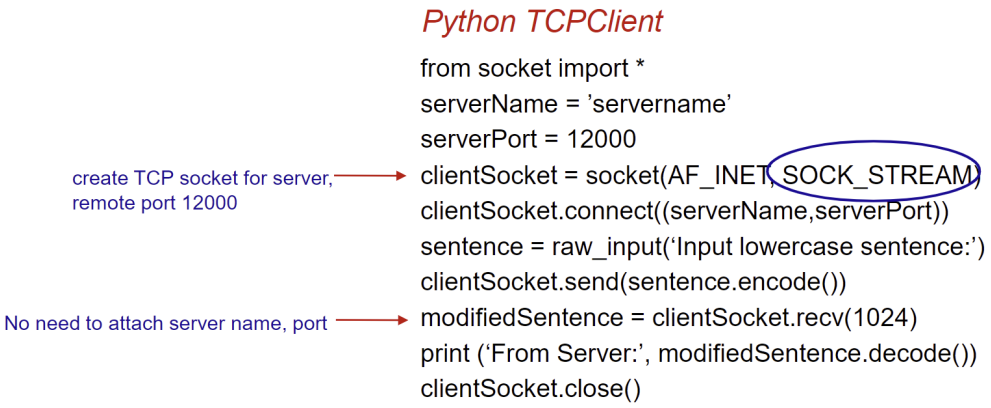
    - 클라이언트는 클라이언트 소켓을 통해 서버에 데이터를 보낸다.
        - 클라이언트 소켓은 bind 없이 목적 포트에 보낸다.
        - sendto로 보낸다.
        - recvfrom으로 받는다.

Python UDPClient

```
from socket import *                              # include Python's socket library
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET,                    # create UDP socket for server
                      SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')  # get user keyboard input
clientSocket.sendto(message.encode(),             # attach server name, port to message; send into socket
                    (serverName, serverPort))
modifiedMessage, serverAddress =                  # read reply characters from socket into string
                    clientSocket.recvfrom(2048)
print modifiedMessage.decode()                    # print out received string and close socket
clientSocket.close()
```

- TCP 소켓

    - reliable, byte stream-oriented
        - 보내면 확실히 간다.
        - 보내질 때 까지 재전송한다.(바이트 단위로)
    - TCP 자체가 복잡한 기술이므로 상대적으로 조금 더 복잡한 소켓이 된다.
    - Creating
        - protocolFamily = PF_INET (= AF_INET)
        - type = SOCK_STREAM = 연결 지향형, 바이트 기반의 신뢰성 있는 전송.
        - protocol = IPPROTO_TCP
    - Destroying
        - 소켓을 닫는다. close()
- TCP 클라이언트
    - 소켓을 만든다.
    - 연결을 설립(Establish)한다.
        - int connect(int socket, struct sockaddr *foreignAddress, unsigned int addressLength)
        - 서버가 accept하면 이제 연결이 성립된다.
        - 3-way handshaking
    - 통신한다.
        - int send(int socket, const void *msg, unsigned int msgLength, int flags)
        - int recv(int socket, void *rcvBuffer, unsigned int bufferLength, int flags)
    - 연결을 닫는다.



- TCP 서버
  - 소켓을 만든다.
  - 소켓에 포트를 할당한다.
    - int bind(int socket, struct sockaddr *localAddress, unsigned int addressLength)
  - 들을 소켓을 정한다.(Listen)
    - int listen(int socket, int queueLimit)
    - accept와 다른 것, 연결이 들어오기 전까지 대기하고 있는것이다.
      - 물건 팔기 전에 가게 열어놓는 느낌.
  - 반복한다.
    - 새로운 연결을 accept한다.
      - int accept(int socket, struct sockaddr *clientAddress, unsigned int *addressLength)
    - 통신한다.
    - 연결을 닫는다.

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                        encode())
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

- TCP를 통한 소켓 프로그래밍
  - 클라이언트는 무조건 서버와 접속해야한다.
    - 즉 서버는 항상 가동중 이여야함!
    - 클라이언트랑 접속하면 서버도 새로운 소켓을 만들어서 통신한다.
      - 여러 클라이언트와 동시에 서버가 통신할 수 있다.
      - source port number = 클라이언트를 구분하기 위해 사용한다.
  - 클라이언트는 서버랑 접속하기 위해
    - TCP 소켓을 만들고, IP 주소를 지정하고, 서버 프로세스의 포트 넘버를 지정한다.
    - 클라이언트가 소켓을 만들었으면, 서버 TCP 소켓과 연결을 설립한다.
  - TCP를 통한 통신은 신뢰성을 보장한다.
    - 클라이언트와 서버 사이에 일종의 파이프를 연결한다.
      - in-order-byte-stream transfer
    - 손실이 없고 순서대로 데이터가 도착한다.

▼ 추가 내용

- IP 주소 = Dotted-Decimal
  - 32비트를 8비트씩(Octet = byte) 쪼개서 10진수로 바꾼다.
  - 따라서 각 10진수는 최대 255까지만 가질 수 있다.

        - ex) 255.255.255.255
    - 도메인 네임 ≠ IP 주소
        - IP 주소는 binary로 얘기하자.
- IP 주소 Conversion
    - Domain name → IP addr(numeric)
        - gethostbyname() →
        - gethostbyaddr() ←
    - IP addr(numeric) → Dotted decimal(atomic)
        - ← inet_addr() or inet_aton()
        - → inet_ntoa()
        - dennis ritchie, inet.h에 박아놓음.
            - atomic to numeric
            - numeric to atomic
            - 각 10진수들이 atomic하다 = 더이상 쪼갤 수 없다. = 각 10진수에는 고유한 역할이 있다.
- big-endian
    - MSB, MSD(digit)
    - MSB가 먼저 온다 = big endian = Sun, HP, SGI
        - 즉 낮은 주소에 높은 바이트를 기록한다
    - LSB가 먼저 온다 = little endian = intel, DEC(Digital Equipment Corporation)
    - 그런데 네트워크 바이트는 big endian을 따른다.
        - ntohl() : network-to-host byte order를 바꾼다, long(32비트)
        - htonl() : host-to-network로 byte order를 바꾼다, Long
        - ntohs(), htons() : short integer(16 bits)를 바꾼다.
    - DEC, Intel, Sun, HP, Motorola, SGI, Cray..?
        - Intel과 HP 빼고 다 사망.

▼ 챕터 2 요약

- 일반적인 요청/응답 메시지 교환
    - 클라이언트는 정보나 서비스를 요청함
    - 서버는 데이터와 상태 코드를 통해 응답함.
- 메시지 포맷
    - 헤더 = 메타데이터, 데이터에 대한 정보를 전달.
    - 데이터 = 보내고자 하는 정보
- 중요한 테마
    - 중앙화 VS 탈중앙화
        - 장단
    - 무상태 VS 상태
    - 확장성
    - 신뢰적 VS 비신뢰적 메시지 전송
    - 네트워크 엣지의 복잡함.