2️⃣ Application Layer

태그	7주차	CSS

Electronic mail

- 전자 메일: 비동기적인 통신 매체

  → 사람들은 상대방 스케줄과 상관없이 자신이 편할 때 메시지를 보내거나 읽는다.
- 분배가 쉽고, 빠르고, 저렴하다.
- 현대의 전자메일: 첨부 메시지, 하이퍼링크, HTML 포맷 텍스트, 내장 사진 등의 특성을 지닌다.

인터넷 메일 시스템의 상위 레벨 개념

- 3개의 주요 요소

사용자 에이전트(user agent)

- 사용자가 메시지를 읽고, 응답하고, 전달하고, 저장하고, 구성하게 해준다.
- 마이크로소프트 아웃룩, 애플 메일, 웹 기반 지메일, 스마트폰용 지메일 앱 등

메일 서버(mail server)

- 전자메일 인프라스트럭처의 중심
- 각 수신자는 메일 서버 안에 메일박스를 갖고 있다.
- 메일박스(mailbox): 수신자에게 온 메시지를 유지하고 관리한다.
- 메시지 전송 경로
  - 송신자의 사용자 에이전트 → 송신자의 메일 서버 → 수신자의 메일 서버(메일박스에 저장됨)
- 전자메일 박스에 있는 메시지를 보려면
  - 메일 서버는 사용자 계정과 비밀번호를 이용하여 사용자를 인증한다.
- 송신자의 메일 서버는 수신자의 메일 서버 고장에 대처해야 한다.
  - 송신자 서버는 해당 메시지를 메시지 큐에 보관하여 30분마다 재전송 시도
  - 여러 날 시도해도 성공하지 못하면, 서버는 그 메시지를 제거
  - 송신자에게 전자메일로 이를 통보

SMTP(Simple Mail Transfer Protocol)

- 인터넷 전자메일을 위한 애플리케이션 계층 프로토콜
- 메일을 송신자의 메일 서버로부터 수신자의 메일 서버로 전송하는 데 TCP를 이용
  - 신뢰성 있는 데이터 전송 서비스를 이용

SMTP

구성

- 클라이언트(송신자 메일 서버)
- 서버(수신자 메일 서버)

→ SMTP의 클라이언트와 서버 모두가 메일 서버에서 수행됨

단점

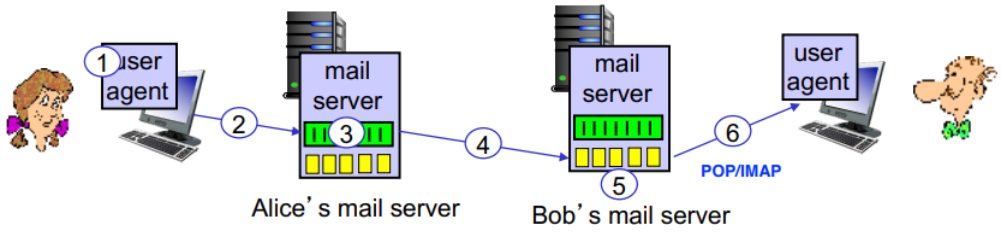
- HTTP보다 훨씬 오래된 낡은 특성을 가진 오래된 기술
- 모든 메일 메시지의 몸체는 단순한 7비트 ASCII여야 한다.

→ 전송 용량이 제한되고 커다란 첨부 파일이나 이미지, 오디오, 비디오 파일 보내기 힘듦

- SMTP를 통해 이진 멀티미디어 데이터를 전송 전에 ASCII로, 전송 후에 다시 원래 메시지로 변환해야 함

→HTTP는 전송 전에 멀티미디어 데이터를 ASCII로 변환하지 않아도 된다.

SMTP의 기본 동작



- 송신자: 앨리스, 수신자: 밥

1. 앨리스는 전자메일 사용자 에이전트를 수행하고 밥의 전자메일 주소를 제공하고, 메시지를 작성하고 사용자 에이전트에게 메시지를 보내라고 명령
2. 앨리스의 사용자 에이전트는 메시지를 앨리스의 메일 서버로 보내고 그곳에서 메시지는 메시지 큐에 들어감
3. 앨리스의 메일 서버에서 동작하는 SMTP의 클라이언트는 메시지 큐에 있는 메시지를 확인하여 밥의 메일 서버에서 수행되고 있는 SMTP 서버에게 TCP 연결을 설정
4. 초기 SMTP 핸드셰이킹 이후 SMTP 클라이언트는 앨리스의 메시지를 TCP 연결로 보냄
5. 밥의 메일 서버 호스트에서 SMTP의 서버는 메시지를 수신하고, 밥의 메일 서버는 그 메시지를 밥의 메일박스에 놓음
6. 밥은 자기가 가능한 시간에 그 메시지를 읽기 위해 사용자 에이전트를 이용함

SMTP 동작의 이해

- SMTP는 메일을 보낼 때 두 메일 서버가 먼 거리에 떨어져 있더라도 중간 메일 서버를 사용하지 않는다!
- 수신자의 메일 서버가 죽어 있다면, 메시지는 송신자의 메일 서버에 남아 새로운 시도를 기다린다.

SMTP의 구체적인 동작 과정

- 송신 메일 서버에서 수신 메일 서버로 어떻게 전송하는지 자세히

1. 클라이언트 SMTP(송신 메일 서버 호스트에서 수행됨)는 서버 SMTP(수신 메일 서버 호스트에서 수행됨)의 25번 포트로 TCP 연결 설정
   a. 서버가 죽어 있다면 클라이언트는 나중에 다시 시도
2. 연결이 설정되면, 서버와 클라이언트는 애플리케이션 계층 핸드셰이킹 수행
   a. SMTP 클라이언트와 서버는 정보를 전송하기 전에 서로 소개
   b. SMTP 클라이언트는 송신자와 수신자의 전자메일 주소를 제공
3. SMTP 클라이언트와 서버가 서로에게 소개를 마치면, 클라이언트는 메시지를 전송
   a. 서버에 오류 없이 메시지를 전달하기 위해 TCP에 의존
4. 서버에 보낼 다른 메시지가 있다면 1~3 과정을 반복
5. 더 이상 보낼 메시지가 없다면 TCP에 연결을 닫을 것을 명령

SMTP 클라이언트(C)와 SMTP 서버(S) 사이의 메시지 전달 과정

- 클라이언트 호스트: crepes.fr, 서버 호스트: hamburger.edu
- C: 클라이언트가 TCP 소켓으로 보내는 라인
- S: 서버가 TCP 소켓으로 보내는 라인
- 클라이언트의 5개 명령: HELO, MAIL FROM, RCPT TO, DATA, QUIT

```
S: 220 hamburger.edu
C: HELO crepes.fr // HELLO의 약자
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr> // 메시지 전송 시작
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA // 데이터를 보내겠다고 명시
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: . // 메시지의 끝을 나타냄
S: 250 Message accpeted for delivery
```

```
C: QUIT // 메시지 전송 종료
S: 221 hamburger.edu closing connection // TCP 연결을 끊음
```

메일 메시지 포맷

- 전자메일을 보낼 때 주변 정보가 포함된 헤더가 메시지 몸체 앞에 오게 된다.
  - 주변 정보는 일련의 헤더 라인에 포함되는데 RFC 5322에 정의되어 있다.
  - RFC 5322: 메일 헤더 라인에 대한 정확한 포맷과 그 의미에 대한 해석을 기술하고 있음
  - 헤더라인과 메시지 몸체는 빈 줄(CRLF)로 분리됨
- 각 헤더 라인은 키워드, 콜론, 값의 순서로 구성되고, 읽을 수 있는 텍스트를 포함
  - 키워드 중 반드시 필요한 것과 선택 사항이 있다.
  - 반드시 필요한 키워드 → From: 헤더 라인과 To: 헤더라인
  - 그 외 → Subject: 헤더라인 등
- SMTP 명령의 FROM, TO와는 다름
  - 여기서 From, To는 메일 메시지 자체의 일부
  - SMTP의 From, To는 SMTP 핸드셰이킹 프로토콜의 일부
- 일반 메시지 헤더 예
  - 메시지 헤더 다음에 빈 줄이 이어지고 메시지 몸체(ASCII 문자)가 나옴

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Searching for the meaning of life.
```

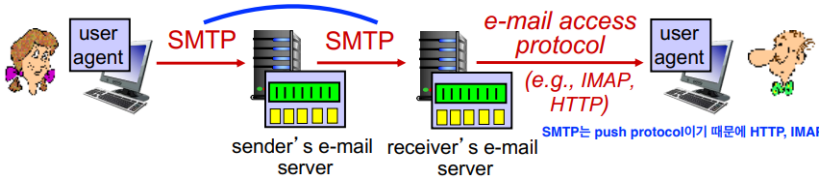
메일 접속 프로토콜

메일 서버의 위치

- 수신자의 메일 서버가 로컬 호스트에 있는 경우
  - 수신 호스트는 전자 메일을 수신하기 위해 항상 켜져 있고, 인터넷에 연결되어야 한다.
  - 메일 서버는 메일박스를 관리하고 SMTP의 클라이언트와 서버 둘 다 수행한다.

전자메일 메시지의 2단계 전송 경로

- 송신자 사용자 에이전트는 수신자의 메일 서버로 직접 대화하지는 않는다.



1. 자신의 메일 서버로 SMTP 또는 HTTP를 이용해 보냄
2. 송신자의 메일 서버는 SMTP를 사용하여 수신자 메일 서버로 메시지를 중계

→ 이렇게 하는 이유?

- 송신자의 메일 서버를 통해 중계하지 않으면, 송신자의 사용자 에이전트는 목적지 메일 서버에 도달할 수 없기 때문
- 송신자는 전자메일을 자신의 메일 서버에 먼저 저장하고, 그 메시지를 수신자의 메일 서버로 반복해서 보낸다. (수신자의 메일 서버가 동작할 때까지)

수신자가 자신의 ISP 내부의 메일 서버에 있는 메시지를 얻는 방법

- 수신자의 사용자 에이전트는 메시지를 얻기 위해 SMTP를 사용할 수 없다.
  - SMTP는 push 프로토콜이고, 메시지를 얻는 것은 pull 동작이기 때문
- 메일 서버로부터 전자 메일을 확인하는 대표적인 2가지 방법
  - 수신자의 메일 서버에 의해 유지되는 폴더를 관리

- 수신자는 자신의 메시지를 자신이 생성한 폴더로 이동, 삭제, 표기할 수 있음

1. HTTP를 사용
    a. 수신자가 웹 기반 전자메일, 스마트폰 앱을 사용하고 있는 경우
    b. 수신자 메일 서버가 송신자 메일 서버와 통신하기 위해 HTTP 인터페이스와 SMTP 인터페이스를 가져야 함
2. IMAP(Internet Mail Access Protocol, 인터넷 메일 접근 프로토콜) 사용
    a. 마이크로소프트 아웃룩과 같은 전형적인 메일 클라이언트를 사용하는 것

DNS: 인터넷의 디렉터리 서비스

호스트 이름(hostname)

- 호스트의 식별자 중 하나
    - www.facebook.com, www.google.com 같은 거 → 사용자가 기억하기 쉬움
    - 호스트 이름은 인터넷에서의 호스트 위치에 대한 정보를 거의 제공하지 않는다.
    - 호스트는 IP 주소(IP address)로도 식별됨 (라우터가 가변 길이의 알파뉴메릭 문자를 처리하는 데 어려움이 있어서)

DNS가 제공하는 서비스

Domain Name System이 필요한 이유?

- 사람-호스트 이름, 라우터-IP 주소 선호 차이를 절충하기 위해 호스트 이름을 IP 주소로 변환해 주는 서비스 필요

  → 이것이 DNS의 주요 임무

DNS란?

- DNS 서버들의 계층 구조로 구현된 분산 데이터베이스
- 호스트가 분산 데이터베이스를 질의하도록 허락하는 애플리케이션 계층 프로토콜
- DNS 서버는 주로 BIND(Berkeley Internet Name Domain) 소프트웨어를 수행하는 유닉스 컴퓨터
- DNS 프로토콜은 UDP상에서 수행되고 포트 번호 53을 이용
- 다른 애플리케이션 프로토콜들이 사용자가 제공한 호스트 이름(HTTP, SMTP, FTP 등)을 IP 주소로 변환하기 위해 사용

HTTP에서 DNS 수행 과정

- 사용자의 호스트에서 수행되는 브라우저(HTTP 클라이언트)가 URL을 요청할 때
    - 사용자의 호스트는 HTTP 요청 메시지를 웹 서버 www.someschool.edu로 보낼 수 있도록 www.someschool.edu의 IP 주소를 얻어야 함
- 수행 과정
    1. 같은 사용자 컴퓨터는 DNS 애플리케이션의 클라이언트 측을 수행
    2. 브라우저는 URL로부터 호스트 이름 www.someschool.edu를 추출하고 그 호스트 이름을 DNS 애플리케이션의 클라이언트 측에 넘김
    3. DNS 클라이언트는 DNS 서버로 호스트 이름을 포함하는 질의를 보냄
    4. DNS 클라이언트는 호스트 이름에 대한 IP 주소를 가진 응답을 받음
    5. 브라우저는 해당 IP 주소와 그 주소의 80번 포트에 위치하는 HTTP 서버 프로세스로 TCP 연결을 초기화

DNS 사용으로 인한 시간 지연

- DNS는 DNS를 사용하는 인터넷 애플리케이션에게 추가 지연을 준다.
- Fortunately, 원하는 IP 주소는 가까운 DNS 서버에 캐싱되어 있음

  → 평균 DNS 지연뿐만 아니라 DNS 네트워크 트래픽 감소에도 도움을 줌

DNS가 제공하는 중요한 추가 서비스

호스트 에일리어싱(host aliasing)

- 복잡한 호스트 이름을 가진 호스트는 하나 이상의 별명을 가질 수 있다.
- 예: relay1.west-coast.enterprise.com
    - enterprise.com or www.enterprise.com 같이 2개의 별명(대체로 더 기억하기 쉬움)
    - relay1.west-coast.enterprise.com을 정식 호스트 이름(canonical hostname)이라고 한다.
- DNS는 호스트의 IP 주소뿐만 아니라 별칭 호스트 이름에 대해서도 정식 호스트 이름을 얻기 위해 이용 가능

메일 서버 에일리어싱(mail server aliasing)

- 전자 메일 주소는 기억하기 쉬운 것이 좋음
  - 메일 서버의 호스트 이름은 더 복잡
- DNS는 별칭으로부터 정식 호스트 이름을 얻기 위해 메일 애플리케이션에 의해 수행
- MX 레코드는 기업의 메일 서버와 웹 서버가 같은 호스트 이름(별칭)을 갖는 것을 허용

부하 분산(load distribution)

- DNS는 중복 웹 서버 같은 여러 중복 서버 사이에 부하를 분산하기 위해서도 사용
  - 인기 사이트(cnn.com)는 여러 서버에 중복되어 있음

  → 각 서버가 다른 종단 시스템에서 수행되고 다른 IP 주소를 갖는다.
- 중복 웹 서버의 경우, 여러 IP 주소가 하나의 정식 호스트 이름과 연결됨
  - DNS 데이터베이스는 이런 IP 주소 집합을 갖고 있음
- 클라이언트가 주소 집합으로 매핑하는 호스트 이름에 대한 DNS 질의

  → 서버가 IP 주소 집합 전체를 가지고 응답
- DNS 순환 방식
  - 각 응답에서의 주소는 순환식으로 전송
  - DNS의 순환 방식은 중복 서버들 사이에서 트래픽을 분산하는 효과를 냄
    - 클라이언트는 주소 집합 내부의 첫 번째 IP 주소로 HTTP 요청 메시지를 보내기 때문
  - 해당 방식은 전자 메일에서도 사용되어 여러 메일 서버에 동일한 별칭을 가질 수 있음

DNS 동작 원리

- 호스트 이름 → IP 주소로 변환하는 서비스에 초점

호스트 이름을 IP 주소로 변환하는 과정

- 사용자 호스트에서 실행되는 어떤 애플리케이션이 호스트 이름을 IP 주소로 변환하려 한다고 가정하자.

순서

1. 그 애플리케이션은 변환될 호스트 이름을 명시하여 DNS 측의 클라이언트 호출
2. 사용자 호스트의 DNS는 네트워크에 질의 메시지를 보냄
   a. 모든 DNS 질의와 응답 메시지는 포트 53의 UDP 데이터그램으로 보내짐
3. 사용자 호스트의 DNS는 요청한 매핑에 해당하는 DNS 응답 메시지를 받음
   a. DNS 응답 메시지를 받기 전에 수 밀리초에서 수 초의 지연이 걸림
4. 호출한 애플리케이션으로 매핑이 전달됨

해석

- 사용자 호스트의 호출한 애플리케이션 관점에서 DNS는 간단하고 직접적인 변환 서비스를 제공하는 블랙박스와 같음
- 블랙박스 구성
  - DNS 서버뿐만 아니라, DNS 서버와 질의를 하는 호스트 사이에서 어떻게 통신하는지를 명시하는 애플리케이션 계층 프로토콜로 구성되어 있음
- DNS의 간단한 설계로 모든 매핑을 포함하는 하나의 인터넷 네임 서버를 생각할 수 있음
  - 중앙 집중 방식에서의 클라이언트와 DNS 서버의 동작
    - 클라이언트는 모든 질의를 단일 네임 서버로 보냄
    - DNS 서버는 질의 클라이언트에게 직접 응답
  - 간단하지만 수많은 호스트를 가진 현대 인터넷에 적합하지 않음

DNS 중앙 집중 방식의 단점

- 서버의 고장: 네임 서버가 고장 나면, 전체 인터넷이 작동하지 않음
- 트래픽양: 단일 DNS 서버가 모든 DNS 질의를 처리해야 함
- 먼 거리의 중앙 집중 데이터베이스: 물리적으로 서버와 먼 곳에서 보내는 질의는 느림 → 심각한 지연
- 유지관리: 단일 네임 서버는 모든 인터넷 호스트에 대한 레코드를 유지해야 함

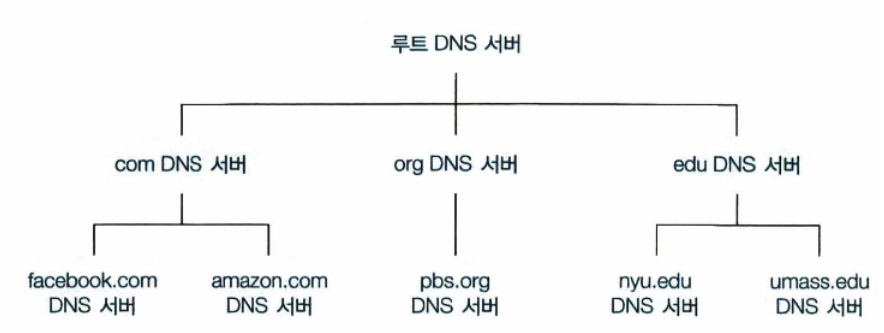
- 모든 새로운 호스트를 반영하기 위해 자주 갱신도 해줘야 함
- 호스트를 등록할 수 있도록 사용자에게 허용하는 것과 관련된 인증 문제

요약

- 단일 DNS 서버에 있는 중앙 집중 데이터베이스는 확장성이 없다!

  → DNS는 분산되도록 설계됨
- DNS는 분산 데이터베이스가 인터넷에서 어떻게 구현될 수 있는지를 보여주는 훌륭한 사례

분산 계층 데이터베이스



- 확장성 문제를 다루기 위해 DNS는 많은 서버를 이용하고, 이를 계층 형태로 구성하며 전 세계로 분산시킴
- 어떠한 단일 DNS 서버도 인터넷에 있는 모든 호스트에 대한 매핑을 갖지 않음
  - 대신 DNS 서버 사이에 분산됨

계층 DNS 서버 종류

- 루트 DNS 서버: 루트 네임 서버는 TLD 서버의 IP 주소들을 제공
  - 1000개 이상의 루트 서버 인스턴스가 전 세계에 흩어져 있음
  - 루트 서버들은 13개의 다른 루트 서버 복사체
    - 12개의 다른 기관에서 관리됨
    - 인터넷 할당 번호 관리기관에 의해 조정됨
- 최상위 레벨 도메인(TLD) 서버: 책임 DNS 서버에 대한 IP 주소를 제공
  - TLD를 지원하는 네트워크 인프라는 크고 복잡함
  - com, org, net, edu, gov 같은 상위 레벨 도메인, kr, uk, fr, ca, jp 같은 모든 국가의 상위 레벨 도메인에 대한 TLD 서버
    - 서버 클러스터라고도 함
- 책임 DNS 서버
  - 인터넷에서 접근하기 쉬운 호스트를 가진 모든 기관은 호스트 이름을 IP 주소로 매핑하는 공개적인 DNS 레코드를 제공해야 함
  - 기관의 책임 DNS 서버는 이 DNS 레코드를 가지고 있음
    - 이 레코드를 갖도록 자신의 책임 DNS 서버의 구현을 선택할 수 있고
    - 일부 서비스 제공자의 책임 DNS 서버에 이 레코드를 저장하도록 비용 지불
  - 대부분의 대학과 큰 기업들은 자신의 기본 책임 DNS 서버와 보조 책임 DNS 서버를 유지하고 구현

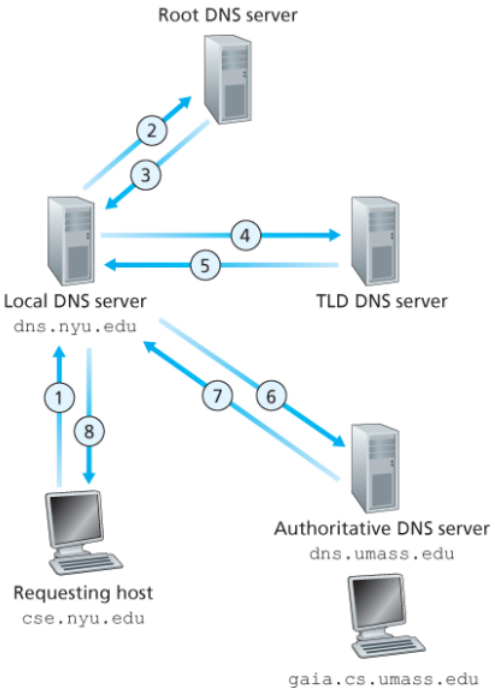
로컬 DNS 서버

- DNS의 또 다른 중요한 형태, DNS 구조의 중심
  - ISP들(대학이나 주거 지역 ISP)은 로컬 DNS 서버를 가짐
    - 로컬 DNS 서버로부터 IP 주소를 호스트에게 제공
- 기관 ISP에서 로컬 DNS 서버는 호스트와 같은 LAN상에 있을 수 있음
- 주거 지역 ISP는 전형적으로 몇 개의 라우터 범위 안에서 호스트로부터 떨어져 있음
  - 호스트가 DNS 질의를 보냄

  → 이 질의는 먼저 프록시로 동작하는 로컬 DNS 서버에 전달

→ 그 로컬 DNS 서버는 이 질의를 DNS 서버 계층으로 전달

DNS 서버 상호 작용 예시



가정

- cse.nyu.edu가 gaia.cs.umass.edu의 IP 주소를 원한다.
- dns.nyu.edu: cse.nyu.edu에 대한 NYU의 로컬 DNS 서버
- dns.umass.edu: gaia.cs.umass.edu에 대한 책임 DNS 서버

과정

1. 자신의 로컬 DNS 서버에 질의와 변환하고 싶은 호스트의 이름(gaia.cs.umass.edu)을 같이 보낸다.
2. 로컬 DNS 서버는 그 질의 메시지를 루트 DNS 서버에게 전달한다.
3. 루트 DNS 서버는 edu를 인식하고, edu에 대한 책임을 가진 TLD 서버의 IP 주소 목록을 로컬 DNS 서버에 보낸다.
4. 로컬 DNS 서버는 TLD 서버에 질의를 보낸다.
5. TLD 서버는 umass.edu를 인식하고, dns.umass.edu로 이름 지어진 책임 DNS 서버의 IP 주소로 응답한다.
6. 로컬 DNS 서버는 직접 책임 DNS 서버(dns.umass.edu)로 질의 메시지를 다시 보낸다.
7. 최종 gaia.cs.umass.edu의 IP 주소를 응답한다.
8. 호스트에 최종 IP 주소를 응답한다.

→ 총 8개의 DNS 메시지, 재귀적 질의와 반복적 질의 사용

- cse.nyu.edu로부터 dns.nyu.edu로 보내는 질의는 자신을 필요한 매핑을 대신하여 얻도록 dns.nyu.edu에 요구하므로 재귀적 질의이고, 나머지는 반복적 질의

질의 전송을 줄이기 위해 DNS 캐싱 방법을 이용

DNS 캐싱

- 실제로 DNS는 지연 성능 향상과 네트워크의 DNS 메시지 수를 줄이기 위해 캐싱 사용

아이디어

- 질의 사슬에서 DNS 서버가 DNS 응답을 받았을 때 로컬 메모리에 응답에 대한 정보를 저장할 수 있다.

예시

- 로컬 DNS 서버 dns.nyu.edu가 임의의 DNS 서버로부터 응답을 받을 때마다 응답에 포함된 정보를 저장할 수 있다.

    - 호스트 이름과 IP 주소 쌍 등이 정보로 저장됨
    - 이후, 다른 호스트 이름으로부터 같은 질의가 DNS 서버로 도착한다면?
        - DNS 서버는 호스트 이름에 대한 책임이 없어도 원하는 IP 주소 제공 가능
- 호스트 DNS와 IP 주소 사이의 매핑과 호스트는 영구적인 것은 아님
    - 일정 기간(일반적으로 2일) 이후에 저장된 정보 제거
- 로컬 DNS 서버는 또한 TLD 서버의 IP 주소를 저장
    - 로컬 DNS 서버가 질의 사슬에서 DNS 서버를 우회하게 (자주 일어나는 동작)

자원 레코드(resource record, RR)

- DNS 서버들은 호스트 이름을 IP 주소로 매핑하기 위해 자원 레코드를 저장
- 각 DNS는 하나 이상의 자원 레코드를 가진 메시지로 응답

자원 레코드의 구성

- 4개의 튜플로 구성
- {Name, Value, Type, TTL}
    - TTL(time to live): 자원 레코드의 생존 기간
        - 자원이 캐시에서 제거되는 시간을 결정

Type의 종류

- Name과 Value의 의미는 Type에 따름 (다음 예시에서는 TTL 필드를 무시)
- Type=A
    - Type A 레코드는 표준 호스트 이름의 IP 주소 매핑 제공
    - (relay1.bar.foo.com, 145.37.93.126, A)
- Type=NS
    - 도메인 내부의 호스트에 대한 IP 주소를 얻을 수 있는 방법을 아는 책임 DNS 서버의 호스트 이름
    - (foo.com, dns.foo.com, NS)
- Type=CNAME
    - 질의 호스트에게 호스트 이름에 대한 정식 이름을 제공
    - (foo.com, relay1.bar.foo.com, CNAME)
- Type=MX
    - 별칭 호스트 이름 Name을 갖는 메일 서버의 정식 이름
    - (foo.com, mail.bar.foo.com, MX)
    - 다른 서버의 정식 이름을 얻기 위해 DNS 클라이언트는 CNAME 레코드에 대한 질의를 한다.

DNS 서버와 레코드 종류의 관계

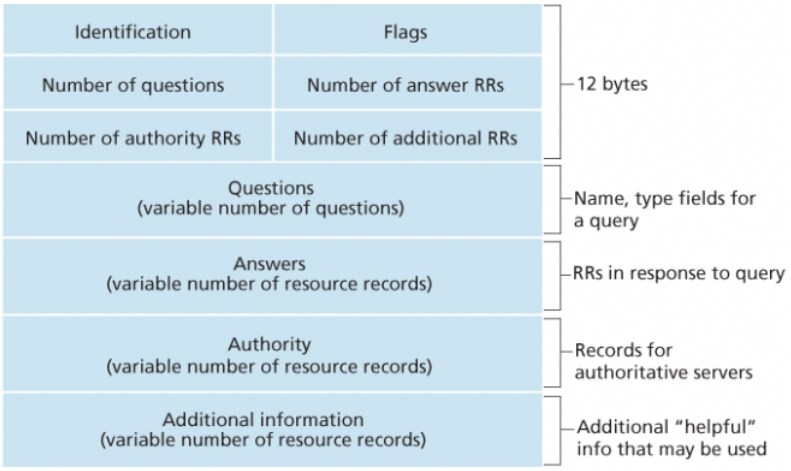
- 한 DNS 서버가 특별한 호스트 이름에 대한 책임 서버인 경우
    - 그 DNS 서버는 그 호스트 이름에 대한 Type A 레코드를 포함
    - 책임 서버가 아니더라도, 캐시에 Type A 레코드 포함 가능
- 서버가 호스트 이름에 대한 책임 서버가 아닌 경우
    - 그 서버는 호스트 이름을 포함하는 도메인에 대한 Type NS 레코드를 포함
    - 또한 Type A 레코드도 포함

      → NS 레코드의 Value 필드에 DNS 서버의 IP 주소를 제공하는 역할을 수행
- 예시 - 호스트 gaia.cs.umass.edu에 대한 책임 서버가 아닌 TLD 서버
    - (umass.edum, dns.umass.edu, NS)
    - Type A 레코드 → DNS 서버를 IP 주소로 매핑

DNS 메시지

- DNS 질의와 DNS 응답 메시지

DNS 메시지 포맷



- 헤더 영역(header section)
  - 처음 12 바이트
    - 식별자
      - 첫 필드, 질의를 식별하는 16비트 숫자
      - 질의에 대한 응답 메시지에 복사됨

      → 클라이언트가 보낸 질의와 수신된 응답 간의 일치를 식별
    - 플래그
      - 질의/응답 플래그(1비트): 메시지가 질의(0)인지 응답(1)인지
      - 책임 플래그(1비트): DNS 서버가 질의 이름에 대한 책임 서버일 때의 응답 메시지 결정
      - 재귀 요구 플래그(1비트): 재귀적 질의를 수행하기를 클라이언트가 원할 때 설정
        - DNS 서버가 레코드를 갖지 않을 때의 경우
        - 이때의 클라이언트는 호스트 혹은 DNS 서버
      - 재귀 가능 필드(1비트): DNS 서버가 재귀 질의를 지원하면 응답에 설정
- 질문 영역(question section): 현재 질의에 대한 정보를 포함
  - 이름 필드: 질의되는 이름
  - 타입 필드: 이름에 대한 문의되는 질문 타입(이름과 연관된 호스트 주소(A 타입) 혹은 이름에 대한 메일 서버(MX 타입) 등)
- 답변 영역(answer section): 질의된 이름에 대한 자원 레코드를 포함
  - 각 자원 레코드에 Type
  - 호스트 이름은 여러 개의 IP 주소를 가져서 응답으로 여러 개의 자원 레코드를 보낼 수 있음
- 책임 영역(authority section): 다른 책임 서버의 레코드를 포함
- 추가 영역(additional section): 다른 도움이 되는 레코드를 포함

DNS 데이터베이스에 레코드 삽입

- 처음에 어떻게 레코드를 데이터베이스에 넣는지

1. 도메인 네임을 등록기관에 등록
   a. 등록기관: 도메인 이름을 DNS 데이터베이스에 넣고, 요금을 받는 상업 기관으로 도메인 네임의 유일성을 확인해 줌
2. 등록기관에게 주책임 서버와 부책임 서버의 이름과 IP 주소를 제공
   a. 두 책임 DNS 서버 각각에 대해 등록기관은 Type NS와 Type A 레코드가 TLD com 서버에 서버에 등록되도록 확인
   b. 주책임 서버는 Type NS와 Type A 레코드를 DNS 시스템에 삽입
3. 웹 서버에 대한 Type A 자원 레코드와 메일 서버에 대한 Type MX 자원 레코드가 서버에 등록되는 것을 확인

DNS 서버의 취약점

DDoS 대역폭 플러딩 공격

- 공격자는 DNS 루트 서버로 다량의 패킷을 보내 다른 DNS 질의들이 응답을 받지 못하게 하려 한다.
- DNS 최상위 도메인 서버들을 공격하면 더 효과적

DNS 중간자 공격, 중독 공격

- 공격자는 호스트로부터 질의를 가로채어 DNS 서버로 가짜 응답을 보내 그 서버가 자신의 캐시에 가짜 레코드를 받아들이도록 한다.
- 이러한 공격에 대비하기 위해 DNS 보안 확정 프로토콜이 개발되어 사용 중

P2P 파일 분배

- P2P 구조는 항상 켜져 있는 인프라스트럭처 서버에 최소한으로 의존한다. (전혀 안 할 수도 있음(
- P2P는 간헐적으로 연결되는 피어(호스트 쌍)들이 서로 직접 통신
  - 피어는 사용자가 제어하는 데스크톱과 랩톱, 스마트폰이 보유
    - 서비스 제공자가 소유하는 것이 아님

자연적인 P2P 애플리케이션

- 커다란 파일을 한 서버에서 다수의 호스트(피어)로 분배한다.
  - 파일이 될 수 있는 것
    - 리눅스 운영체제의 새로운 버전
    - 기존 운영체제 혹은 애플리케이션을 위한 소프트웨어 패치
    - MP3 음악파일 혹은 MPEG 비디오 파일
- 클라이언트-서버 파일 분배에서 서버는 파일 복사본을 각 피어들에게 보내야 함
  - 이런 행위는 서버에게 큰 부하를 주고 많은 양의 서버 대역폭을 소비
  - P2P 파일 분배에서 각 피어는 서버의 분배 프로세스를 도울 수 있다.

  → 피어는 수신한 파일의 일부분을 다른 피어들에게 재분배할 수 있다.

P2P 구조의 확장성

한 파일을 고정된 수의 피어들에게 분배하는 양적 모델

- 서버와 피어들은 접속 링크로 인터넷에 연결되어 있다.

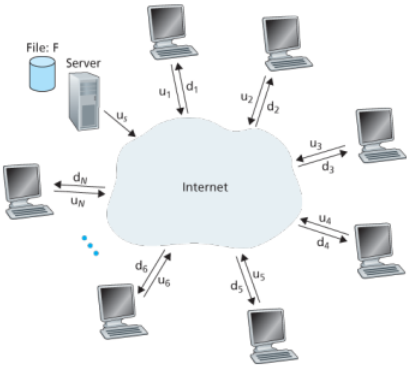


Figure 2.22 An illustrative file distribution problem

- U(s): 서버의 접속 링크 업로드 속도
- U(i): i번째 피어의 접속 링크 업로드 속도
- d(i): i번째 피어의 접속 링크 다운로드 속도
- 파일 F: 분배되는 파일의 크기(비트)
- N: 파일의 복사본을 얻고자 하는 피어들의 수
- 분배 시간(distribution time): 모든 N개의 피어들이 복사본을 얻는 데 걸리는 시간

클라이언트-서버 구조에 대한 분배 시간

가정

- 모든 업로드와 다운로드 접속 대역폭을 이 파일 분배에 모두 사용
- 인터넷 코어가 풍부한 대역폭을 갖고 있다.

분배 시간(D(cs)) 산출 과정

- 클라이언트-서버 구조에서는 어떤 피어도 파일을 분배하는데 도움을 주지 않는다.
- 서버는 파일 복사본을 N개의 피어 각각에게 전송해야 함
  - 서버는 NF 비트를 전송해야 함
  - 서버의 업로드 속도가 U(s)이기 때문에 파일 분배 시간 = NF/U(s)
- 가장 낮은 다운로드 속도를 가진 피어는 F/d(min)초보다 적은 시간에 파일의 모든 F 비트를 얻을 수 없다.
  - 따라서 최소 분배 시간은 적어도 F/d(min)
    - d(min): 가장 낮은 다운로드 속도를 가진 피어의 다운로드 속도

수식

D(cs) ≥ max{ NF/u(s), F/d(min }

- 충분히 큰 N에 대해 클라이언트-서버 분배 시간은 NF/u(s)로 주어진다.
- N에 따라 선형 증가한다.

P2P 구조 분배 시간

여기서는 각 피어들이 서버가 파일을 분배하는 데 도움을 줄 수 있다.

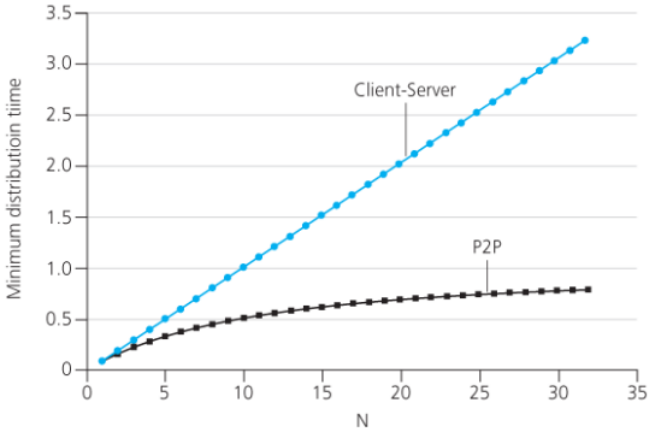
특히 한 피어가 파일 데이터 일부를 수신할 때, 피어는 그 데이터를 다른 피어들에게 재분배하는 데 자신의 업로드 용량을 이용할 수 있다.

- 분배가 시작되면 서버만이 파일을 갖고 있다.
  - 이 파일이 피어 커뮤니티에 도달할 수 있도록 하기 위해, 서버는 적어도 한 번 접속 링크로 파일의 각 비트를 보내야 한다.
  - 따라서 최소 분배 시간은 적어도 F/u(s)다.
    (서버가 한 번 보낸 비트는 서버가 다시 보낼 필요가 없는데, 이는 피어들이 그들 사이에서 재분배할 수 있기 때문이다.)
- 클라이언트-서버 구조와 마찬가지로 다운로드 속도가 가장 낮은 피어는 F/d(min)초보다 적은 시간 안에 파일의 모든 F 비트를 얻을 수 없다.
  - 따라서 최소 분배시간은 적어도 F/d(min)이다.
- 마지막으로, 시스템의 전체 업로드 용량은 전체적으로 서버의 업로드 속도와 각 피어들의 속도를 더한 것이다. 이를 u(total)이라 하자.
  - 시스템은 각 피어들 각각에게 F 비트를 전달해야 한다. 이는 u(total)보다 빠르게 할 수 없다.
  - 따라서 최소 분배 시간은 NF/u(total)이다.

즉, 분배 시간을 D(p2p)라고 하면 다음과 같은 수식을 얻을 수 있다.

D(p2p) ≥ max{ F/u(s), F/d(min), NF/u(total) }

하한값은 서버-클라이언트 구조에서 서버가 전송을 스케줄링 하거나, P2P 구조에서는 각 피어가 비트를 수신하자마자 그 비트를 재분배할 수 있다고 가정하면 식의 하한값을 최소 분배시간으로 채택할 수 있다.



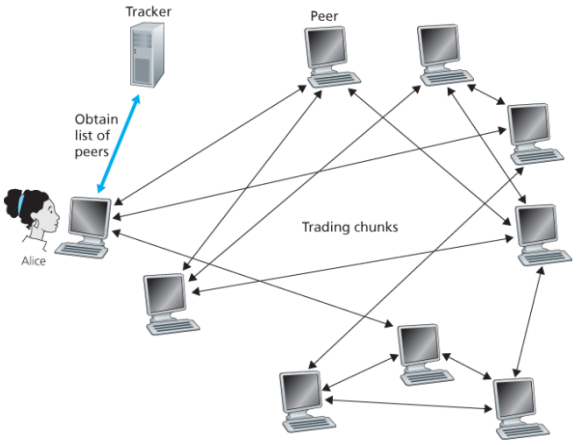
→ P2P 구조를 가진 애플리케이션은 자가 확장성을 갖는다.

비트토렌트(BitTorrent)

- 비트토렌트는 파일 분배를 위한 인기 있는 P2P 프로토콜

토렌트(torrent)

- 비트토렌트 용어로 특정 파일의 분배에 참여하는 모든 피어의 모임



토렌트에 참여하는 피어들은 서로에게서 같은 크기의 청크(chunk)를 다운로드(일반적으로 256KB)

- 처음으로 가입하면 그 피어에는 청크가 없지만, 시간이 지나면 점점 많은 청크를 쌓을 수 있다.
- 피어가 청크를 다운로드할 때 또한 청크를 다른 피어들에게 업로드한다.
- 일단 한 피어가 전체 파일을 얻으면 토렌트를 떠나거나, 토렌트에 남아서 다른 피어들로 청크를 계속해서 업로드할 수 있다.

트래커(tracker)

각 토렌트는 트래커라고 부르는 인프라스트럭처 노드를 갖고 있다.

한 피어가 토렌트에 가입할 때 트래커에 자신을 등록하고 주기적으로 자신이 아직 토렌트에 있음을 알려, 트래커는 토렌트에 있는 피어들을 추적할 수 있다.

가장 드문 것 먼저

- requesting chunks: rarest first

어느 임의의 시간 안에 앨리스는 청크의 일부를 가질 것이고, 이웃들이 어느 청크를 가지고 있는지를 알게 될 것이다.

앨리스는 이러한 정보를 바탕으로 다음 2가지 결정을 한다.

1. 이웃으로부터 어느 청크를 먼저 요구할 것인가?
2. 이웃들 중 어느 피어에게 청크를 요청할 것인가?

이때 rarest first(가장 드문 것 먼저) 기술을 사용한다.

갖고 있지 않은 청크 중에서, 이웃 가운데 가장 드문 청크를 결정하고 이를 먼저 요구하는 것이다.

이 방법을 통해 가장 드문 청크들은 더 빨리 재분배될 수 있어서 각 청크의 복사본 수가 대략적을 동일해질 수 있다.

현명한 교역

- sending chunks: tit-for-tat

어느 요청에 응답할 지 결정할 때 앨리스가 가장 빠른 속도로 그녀에게 데이터를 제공하는 이웃에게 우선순위를 주는 것이다.

- 특히, 계속해서 비트를 수신하는 속도를 측정하고 가장 빠르게 전송하는 4개의 피어를 결정하고, 이 4개의 피어에게 청크를 보냄으로써 보답한다.
- 이는 10초마다 계산하여 집합을 수정한다.

비트 토렌트 용어로 4개의 피어는 활성화되었다(unchoked)고 한다.

40초마다 위 피어를 제외하고 임의의 피어에게 청크를 보낸다.

비트 토렌트 용어로 이 피어는 낙관적으로 활성화되었다(optimisitically unchoked)고 한다.

- 즉, 이제 앨리스는 임의의 피어에게 활성화될 수 있고, 활성화가 된다면 임의의 피어도 앨리스에게 활성화될 수 있다.
- 임의의 선택을 통해 고정된 피어들과만 청크를 교역하는 것이 아니라 여러 피어와 교역할 수 있게 된다.

Video streaming

인터넷 비디오

이미지의 연속으로서 일반적으로 초당 24개에서 30개의 이미지로 일정한 속도로 표시된다.

압축되지 않은 디지털 인코딩된 이미지는 픽셀 단위로 구성되며, 각 픽셀은 휘도와 색상을 나타내는 여러 비트들로 인코딩된다.

비디오의 중요한 특징은 압축될 수 있다는 것

비디오의 품질과 비트 전송률은 서로 반비례(비트 전송률이 높을수록 이미지 품질이 좋다)

HTTP 스트리밍 및 DASH

- HTTP 스트리밍에서 비디오는 HTTP 서버 내의 특정 URL을 갖는 일반적인 파일로 저장된다.

1. 클라이언트는 서버에게 TCP 연결을 설립하고 해당 URL에 대한 HTTP GET 요청을 발생시킨다.
2. 서버는 기본 네트워크 프로토콜 및 트래픽이 허용되는 대로 HTTP 응답 메시지 내에서 비디오 파일을 전송한다.
3. 애플리케이션 버퍼에 전송된 바이트가 저장된다.
4. 버퍼의 바이트 수가 미리 정해진 임계값을 초과하면 재생을 시작한다.

특히, 버퍼에서 주기적으로 비디오 프레임을 가져와 프레임을 압축 해제한 다음 사용자의 화면에 표시한다.

문제점

가용 대역폭이 달라도 똑같이 인코딩된 비디오를 전송 받는다는 문제가 있다.

이 문제로 인한 HTTP 기반 스트리밍인 DASH(Dynamic Adaptive Streaming over HTTP)가 개발되었다.

Dash

비디오는 여러 가진 버전으로 인코딩 되며, 각 버전은 비트율과 품질 수준이 서로 다르다.

클라이언트는 동적으로 서로 다른 버전의 비디오를 몇 초 분량의 길이를 갖는 비디오 조각(청크) 단위로 요청한다.

(가용 대역폭이 충분하면 높은 비트율의 비디오 버전을 요청하고 가용 대역폭이 적으면 낮은 비트율의 비디오 버전을 요청한다.)

각 비디오 버전은 HTTP 서버에 서로 다른 URL을 가지고 저장된다.

HTTP 서버는 비트율에 따른 각 버전의 URL을 제공하는 매니페스트(manifest) 파일을 갖고 있다.

Content distribution network

- 다수의 지점에 분산된 서버들을 운영하며, 비디오 및 다른 형태의 웹 콘텐츠 데이터의 복사본을 이러한 분산 서버에 저장한다.

사용자는 최적의 서비스를 제공할 수 있는 CDN과 연결된다.

CDN 서버 위치의 철학 두가지

Enter Deep

서버 클러스터를 세계 곳곳의 접속 네트워크에 구축함으로써 ISP의 접속 네트워크로 깊숙이 들어가는 것

장점: 서버를 최대한 사용자 가까이 위치시켜 사용자와 CDN 서버 사이의 링크 및 라우터 수를 줄인다.

단점: 너무 분산되어서 유지보수가 힘들다.

Bring Home

핵심 지점에 큰 규모의 서버 클러스터를 구축하여 ISP를 Home으로 가져오는 것

장점: 클러스터 유지 및 관리 비용이 줄어듦

단점: 사용자가 느끼는 지연 시간과 처리율은 상대적으로 나빠짐

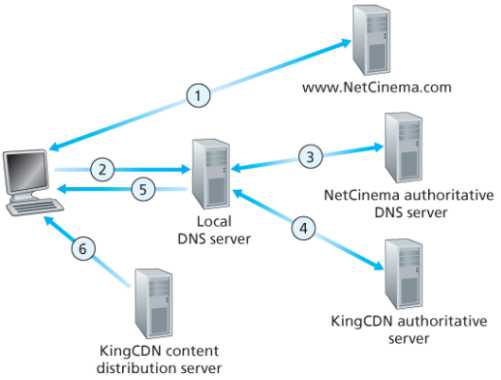
CDN은 콘텐츠의 복사본을 이들 클러스터에 저장하는데 모든 복사본을 유지하지는 않는다.

(어떤 비디오는 인기가 거의 없거나 특정 국가에서만 인기가 있을 수 있기 때문이다.)

실제로 CDN은 클러스터에 대해 사용자의 요청이 오면 중앙 서버나 다른 클러스터로부터 전송받아 사용자에게 서비스하는 동시에 복사본을 만들어 저장하는 pull 방식을 이용한다.

저장 공간이 가득 차게 되면 자주 사용되지 않는 비디오 데이터는 삭제된다.

CDN 동작



1. 사용자가 URL(www.NetCinema.com)을 입력한다.
2. 사용자의 호스트는 URL의 host name에 대한 질의를 로컬 DNS로 보낸다.
3. 로컬 DNS는 host name의 책임 DNS 서버로 질의를 전달한다. 책임 DNS 서버는 해당 질의를 CDN 서버로 연결하기 위해 CDN 서버의 책임 DNS 서버의 IP를 전달한다.
4. 로컬 DNS는 CDN 서버의 책임 DNS로 질의를 보내고, CDN 콘텐츠 서버의 IP 주소를 로컬 DNS 서버로 응답한다. 이때 클라이언트가 콘텐츠를 전송받게 될 서버가 결정된다.
5. 로컬 DNS 서버는 사용자 호스트에게 CDN 서버의 IP 주소를 알려준다.
6. 클라이언트는 호스트가 알게된 IP 주소로 HTTP 혹은 DASH 프로토콜을 통해 비디오를 받아온다.

클러스터 선택 정책

- 지리적으로 가장 가까운 클러스터 할당
  - 인터넷의 경로의 지연이나 가용 대역폭 변화는 무시하고 항상 같은 클러스터를 할당하게 됨
- 현재는 주기적으로 클러스터와 클라이언트 간의 지연 및 손실 성능에 대한 실시간 측정을 수행

사례

넷플릭스, 유튜브

Socket programming with TCP and UDP

UDP 소켓 프로그래밍

과정

1. 송신 프로세스가 데이터 패킷을 소켓 밖으로 내기 전에 먼저 패킷에 목적지 주소를 붙여 넣음
2. 패킷이 송신자의 소켓을 통과
3. 해당 소켓을 인터넷을 통해 수신 프로세스에 있는 소켓으로 라우트함
4. 패킷이 수신 소켓에 도착
5. 수신 프로세스는 소켓을 통해 해당 패킷을 추출하고 다음에 패킷의 콘텐츠를 조사 및 동작

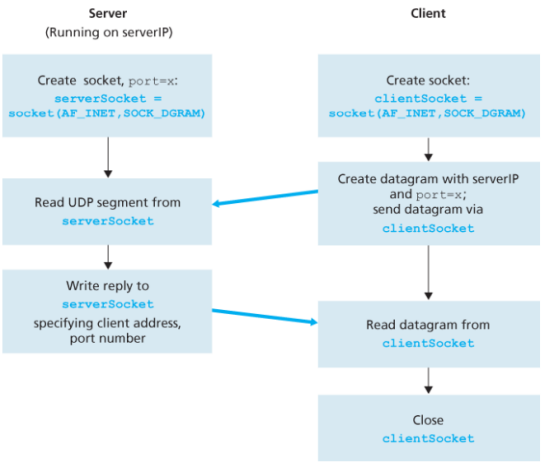
UDP 패킷 구성

- 목적지 주소
  - 목적지 호스트 IP 주소 + 소켓 포트 번호
    - 목적지 호스트 IP 주소

    → 이를 통해 인터넷의 라우터는 목적지 호스트로 인터넷을 통해 패킷을 라우트 가능
    - 소켓 포트 번호: 소켓이 생성될 때 소켓에 할당되는 식별자
      - 호스트는 여러 개의 소켓을 갖는 많은 네트워크 애플리케이션 프로세스 수행 가능

      → 소켓 식별이 필요

소켓 프로그래밍



1. 클라이언트는 키보드로부터 한 줄의 데이터를 읽고 그 데이터를 서버로 보냄
2. 서버는 그 데이터를 수신하고 문자를 대문자로 변환
3. 서버는 수정된 데이터를 클라이언트에게 보냄
4. 클라이언트는 수정된 데이터를 수신하고 그 줄을 화면에 나타냄

UDPClient.py

소켓을 생성할 때는 따로 소켓의 포트 번호를 명시하지 않아도 되며, 운영체제가 이 작업을 대신 수행한다.

```
# socket module이다. 이 module을 통해 소켓을 생성할 수 있다.
from socket import *

#서버의 IP 혹은 서버의 호스트 이름을 할당한다.
serverName = 'hostname'

# 목적지 port 번호를 나타낸다.
serverPort = 12000

# 클라이언트 소켓을 생성한다. AF_INET은 IPv4를 사용하고 있음을 나타내고, SOCK_DGRAM은 UDP 소켓임을 의미한다.
clientSocket = socket(AF_INET, SOCK_DGRAM)

# 보낼 메시지를 입력 받는다.
message = Input('Input lowercase sentence:')

# 소켓으로 바이트 형태를 보내기 위해 먼저 encode()를 통해 바이트 타입으로 변환한다.
# sendTo() 메서드는 목적지 주소를 메시지에 붙이고 그 패킷을 프로세스 소켓인 clientSocket으로 보낸다.
# 클라이언트 주소도 같이 보내지는데 이는 자동으로 수행된다.
clientSocket.sendto(message.encode(),(serverName, serverPort))

# 패킷 데이터는 modifiedMessage에 저장되고, 패킷의 출발지 주소(IP, port)는 serverAddress에 할당된다.
# recvfrom() 메서드는 2048의 버퍼 크기로 받아들인다.
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

# 출력
print(modifiedMessage.decode())

# 소켓 닫기
clientSocket.close()
```

UDPServer.py

while문을 통하여 한 번의 통신 이후에도, 계속 다음 UDP 패킷이 도착하기를 기다린다.

```
from socket import *

# 포트 번호
serverPort = 12000
```

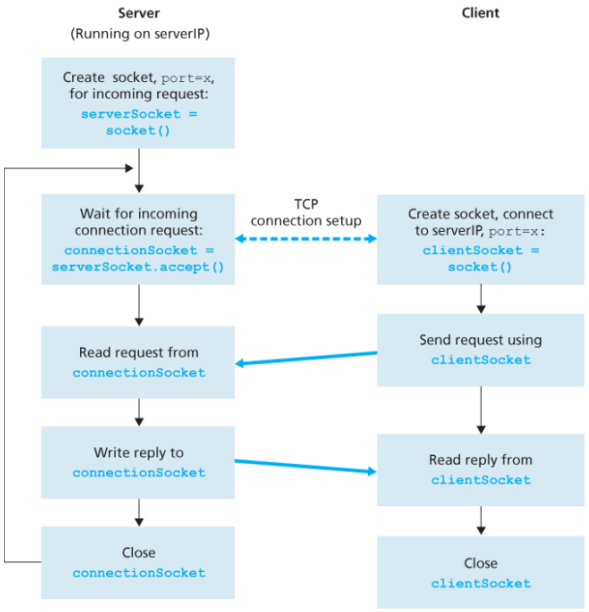
```
# UDP 소켓 생성
serverSocket = socket(AF_INET, SOCK_DGRAM)

# 12000 포트 번호를 소켓에 할당한다. 이를 통해 서버 IP 주소의 12000 포트로 패킷을 보내면 해당 소켓으로 패킷이 전달된다.
serverSocket.bind((’’, serverPort))

print(”The server is ready to receive”)

while True:
    # 패킷이 서버에 도착하면 데이터는 메세지에 할당되고 패킷의 출발지 주소는 clientAddress에 저장된다.
    # 해당 주소로 서버는 응답을 어디에 보내야할지 알 수 있다.
    message, clientAddress = serverSocket.recvfrom(2048)

    # 바이트 데이터를 decode()하고 대문자로 변환한다.
    modifiedMessage = message.decode().upper()

    # 클라이언트 주소를 대문자로 변환된 메시지에 붙이고, 그 결과로 만들어진 패킷을 서버에 보낸다.
    # 서버의 주소도 같이 보내지는데 이는 자동으로 수행된다.
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

TCP 소켓 프로그래밍

과정

1. 먼저 클라이언트는 서버로의 접속을 시도한다.→ 서버는 클라이언트의 초기 접속에 응대할 수 있도록 준비해야 함
   - 여기에는 2가지 의미가 존재
     1. TCP 서버는 클라이언트가 접속하기 전에 프로세스를 먼저 수행하고 있어야 함
     2. 클라이언트로부터의 초기 접속을 처리하는 특별한 출입문(소켓)을 가져야 한다.
2. 접속에 대한 시도로 클라이언트 프로그램에서 TCP 소켓을 생성한다.
   - TCP 소켓을 생성할 때, 서버에 있는 환영 주소를 명시한다.
     - 환영(welcome) 주소: 서버의 IP 주소와 소켓의 포트 번호
3. 소켓을 생성한 후, 클라이언트는 세 방향 핸드셰이크를 하고 서버와 TCP 연결을 설정
   - 세 방향 핸드셰이크는 트랜스포트 계층에서 일어남→ 클라이언트와 서버 프로그램은 전혀 인식하지 못함
4. 세 방향 핸드셰이크 동안, 클라이언트 프로세스는 서버 프로세스의 출입문을 두드림
5. 서버가 노크를 들으면, 서버는 해당 클라이언트에게 지정된 새로운 소켓을 생성
   - 연결을 시도하는 클라이언트에게 새로 지정된 소켓 → `connectionSocket`

소켓 프로그래밍



TCPClient.py

```
from socket import *

serverName = 'servername'
serverPort = 12000

# 클라이언트 소켓을 의미한다. SOCK_STREAM으로 TCP 소켓임을 명시했다.
# UDP 때와 마찬가지로 따로 출발지 주소를 명시하지 않는다. (운영체제가 대신 해준다.)
clientSocket = socket(AF_INET, SOCK_STREAM)

# 클라이언트가 TCP 소켓을 이용하여 서버로 데이터를 보내기 전에 TCP 연결이 먼저 클라이언트와 서버 사이에 설정되어야 한다.
# 해당 라인으로 TCP 연결을 시작하고, connect() 메서드의 파라미터는 연결의 서버 쪽 주소이다.
# 이 라인이 수행된 후에 3-way handshake가 수행되고 클라이언트와 서버 간에 TCP 연결이 설정된다.
clientSocket.connect((serverName, serverPort))

sentence = raw_input('Input lowercase sentence:')

# 클라이언트 소켓을 통해 TCP 연결로 보낸다. UDP 소켓처럼 패킷을 명시적으로 생성하지 않으며 패킷에 목적지 주소를 붙이지 않
는다.
# 대신 클라이언트 프로그램은 단순히 문자열에 있는 바이트를 TCP 연결에 제공한다.
clientSocket.send(sentence.encode())

# 서버로부터 바이트를 수신하기를 기다린다.
modifiedSentence = clientSocket.recv(1024)
print('From Server: ', modifiedSentence.decode())

# 연결을 닫는다. 이는 클라이언트 TCP가 서버의 TCP에게 TCP 메시지를 보내게 한다.
clientSocket.close()
```

TCPServer.py

```
from socket import *

serverPort = 12000

# TCP 소켓 생성
serverSocket = socket(AF_INET, SOCK_STREAM)

# 서버의 포트 번호를 소켓과 연관시킨다.
serverSocket.bind(('', serverPort))
```

```
# 연관시킨 소켓은 대기하며 클라이언트가 문을 두드리기를 기다린다.
# 큐잉되는 연결의 최대 수를 나타낸다.
serverSocket.listen(1)
print('The server is ready to receive')

while True:
    # 클라이언트가 TCP 연결 요청을 하면 accept() 메소드를 시작해서 클라이언트를 위한 연결 소켓을 서버에 생성한다.
    # 그 뒤 클라이언트와 서버는 핸드셰이킹을 완료해서 클라이언트의 소켓과 연결 소켓 사이의 TCP 연결을 생성한다.
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())

    # 응답을 보내고 연결 소켓을 닫는다. 그러나 환영소켓인 serverSocket이 열려있어 다른 클라이언트가 서버에 연결을 요청할
수 있다.
    connectionSocket.close()
```