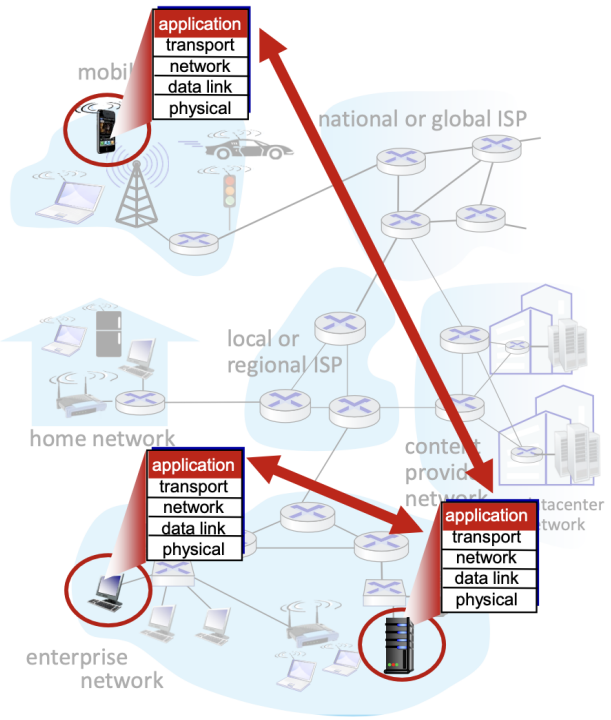
chapter2

Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols
  - HTTP
  - SMTP, IMAP
  - DNS
- programming network applications
  - socket API

- **client-server: 서버와 서버에 접속하는 게 확실하게 구분**
- **peer-to-peer(p2p): 서버이자 클라이언트**



end system(host) 에는 모든 layer stack이 다 올라가

인터넷 5계층 ⇒ application, transport, network, data link, physical

osi 7계층 ⇒ application, presentation, session, transport, network, data link, physical

Client-server paradigm(HTTP, IMAP, FTP 사용)

server

1. 모든 기능, 데이터를 가지고 있어

2. IP 를 함부로 못 바꿈 (IP 영구적)

clients

1. 유동 IP로 사용 (server가 clients의 IP를 알면 안되니까 IP가 바뀌어도 상관없어)
2. client 간에는 통신하지 않음

Peer-peer architecture

1. server 와 client 가 없어
2. 그럼? server 이자 동시에 client

example

1. 용량: 10, 사용자: 10
2. 사용자(client) → 100명으로 증가 ⇒ server의 용량도 100으로 증가

결론: client request, server 용량 모두 증가

client process: process that **initiates** communication

server process: process that **waits** to be contacted

Processes communicating

process: 현재 수행중인 program

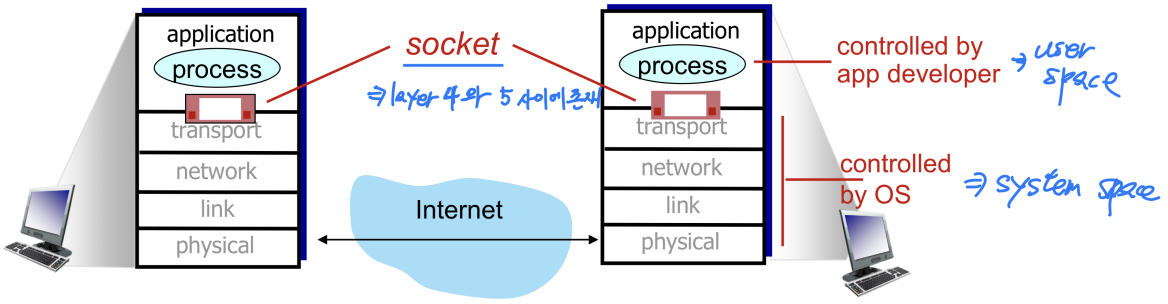
client는 communication을 요청(request)

↔ server는 communication을 수락(accept)

Sockets

socket: process 가 message 를 송수신할 때 사용하는 것 (집에서 현관문 같은 역할)

+ layer4 와 layer5 사이에 존재(transport와 application)



application 계층은 user space

그 밑으로는 다 system space

Addressing processes

identifier(식별자): IP 주소(IP address)와 port 번호(port number)까지 필요

⇒ 기숙사를 예로 들어보면 1생활관 이라고만 하는게 아닌 1생활관 301호 이런식으로 써야 정확한 위치 파악이 가능하기 때문

IP address: 128.119.245.12 ⇒ 기숙사 주소

port number: 80 ⇒ 방번호

Application-layer protocol defines

An application-layer protocol defines:

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

protocol의 종류가 2개

1. open protocols
   - RFC에 의해서 정의됨
   - 상호운용성(interoperability) ⇒ 표준을 따르기 때문에 가능 (생각해보면 모두가 같은 rule을 따르면 대화가 가능하니까)
2. proprietary protocols
   - 독자적(자체) 프로토콜 == 자기들만 쓰는 프로토콜

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

security

- encryption, data integrity, …

data integrity(데이터 유실 관련해서 어느정도까지 허용되는지)

- 그래서 어떤 데이터는 반드시 100% 전송을 요구하는 반면
- 어떤 데이터는 조금 손실이 있어도 괜찮아
- 이거에 대해서 간단하게 생각해보면 radio, audio 같은건 잠시 끊겨도 마저 들을 수 있는 반면에
- 파일 같은 친구들은 손실되면 읽지를 못하니깐

timing(delay를 뜻함)

- delay를 어느정도까지 허용할 수 있는지
- discord 나 zoom 같은 거 delay 되면 뭐... 망했죠?
- 반대로 email 이 칼같이 갈 이유가 있나? 조금 delay 가 있어도 괜찮겠지 뭐

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

위에 나온 data loss(data integrity) 와 throughput(timing) 에 관련된 정보

굳이 자세히 외울 필요는 없을듯...

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

TCP 와 UDP 관련해서 설명이 나오는데

일단 UDP 에 관련해서는 나중에 배움 (일단 필요하다 라고만 알고 넘어가기)

TCP service

- reliable transport
  - 위에서 언급됐듯이 reliable transport는 data loss가 없는걸 말함
- flow control && congestion control

- 얘네는 개념이 조금 다르긴 한데 사람들이 이 둘 개념을 섞어 쓰는 경향이 있음 (알아서 잘 파악하란 소리)
- flow control: receiver 가 받을 수 있는 속도에 맞춰서 sender 가 data 를 보내는 것
- congestion control: sender, receiver 둘 다 문제가 없지만 network에 문제가 발생한 경우 ⇒ 네트워크 환경에 맞춰서 sender가 전송량 조절

UDP service

- multiplexing 기능만 지원

Internet transport protocols services

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 7320]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

1. TP ⇒ transport protocol 이걸 기억해야함
2. FTP ⇒ File Transport Protocol
3. SMTP ⇒ Simple Mail Transport Protocol
4. HTTP ⇒ Hyper Text Transport Protocol
5. SIP ⇒ Session Initiation Protocol
6. RTP ⇒ Real time Transport Protocol

Securing TCP

Securing TCP

Vanilla TCP & UDP sockets:

- no encryption ⇒ presentation layer에서 함
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication ⇒ 상대방의 실제 id

TSL implemented in application layer

- apps use TSL libraries, that use TCP in turn

TLS socket API

- cleartext sent into socket traverse Internet *encrypted*
- see Chapter 8

Transport Layer Security

- TCP 연결에서 암호화 해서 전송
- data integrity(데이터 무결성) == 손실되는 데이터가 없다 이소리
- end-point authentication ⇒ 상대방의 실제 id

HTTP connections: two types

HTTP connections: two types

Non-persistent HTTP

⇒ 연결이 계속 지속되지 않는다.

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

Persistent HTTP

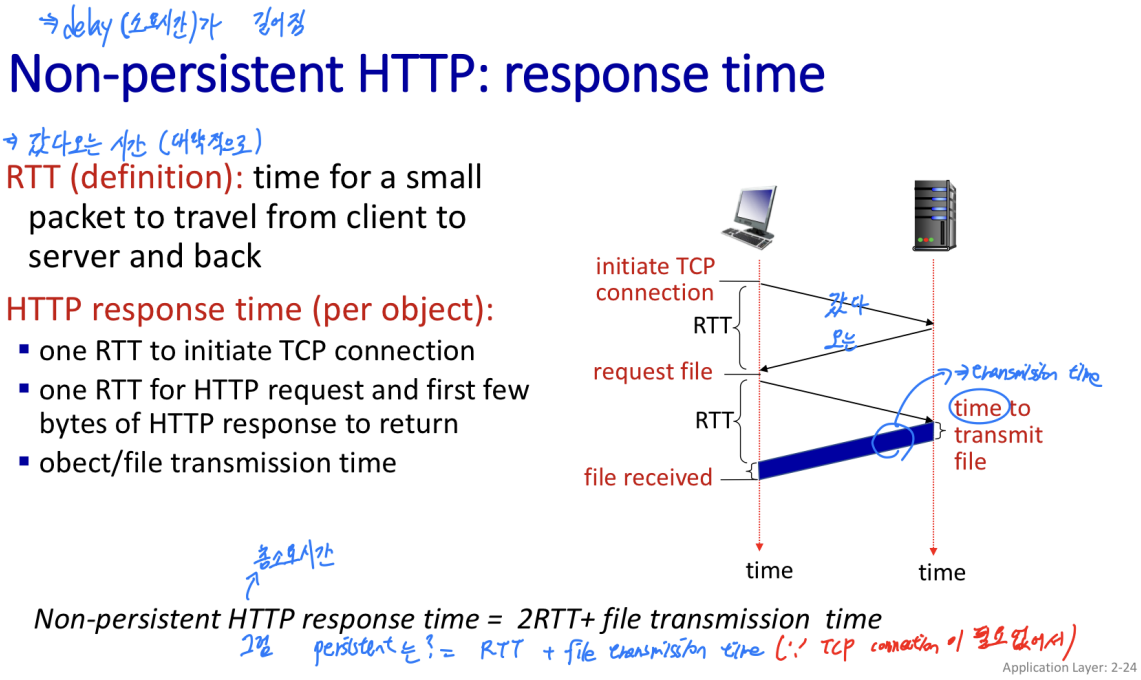
⇒ 연결이 지속됨

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

1. Non-persistent HTTP (연결이 계속 지속되지 않는다)
   - 하나 보낼때마다 연결 열고 보내고 나서 연결끊고
   - 당연히 open-close 를 반복하니 connection을 여러번 하겠지
2. Persistent HTTP (연결이 지속된다)

- 보낼거 다 보내고 나서 연결 끊기
- 이 짓 하니까 당연히 connection 한번이면 됨

Non-persistent HTTP



우선 연결이 지속되지 않으니 하나 보낼때마다 갔다 오는 시간이 있겠지?

갔다오는 시간이 RTT(Round Trip Time)

그러면 파일 1개 보내려면

1. TCP Connection 을 요청 ⇒ 1RTT
2. 파일 전송 ⇒ 1RTT + file transmission time
3. 합계: 2RTT + file transmission time

↔ 그럼 반대로 Persistent 는?

1. TCP Connection 을 요청할 필요가 없지 (서버에 연결이 돼있을거니까)
2. 파일 전송 ⇒ 1RTT + file transmission time
3. 합계: 1RTT + file transmission time

Persistent HTTP

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel

Persistent HTTP (HTTP1.1):

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects (cutting response time in half)

파일을 반복적으로 많이 보내거나 가져올수록 유리하다

근데 이걸 보낼 때 response를 받고 다시 보내는게 아니라, 보낼 수 있을 때 계속 보냄

Other HTTP request messages

POST method:

- web page often includes form input
- user input sent from client to server in entity body of HTTP POST request message

GET method (for sending data to server):

- include user data in URL field of HTTP GET request message (following a '?'):

```
www.somesite.com/animalsearch?monkeys&banana
```

HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.

PUT method:

- uploads new file (object) to server
- completely replaces file that exists at specified URL with content in entity body of POST HTTP request message

method: 그 HTML 에서 form 태그 안에 적는 그 method 맞음

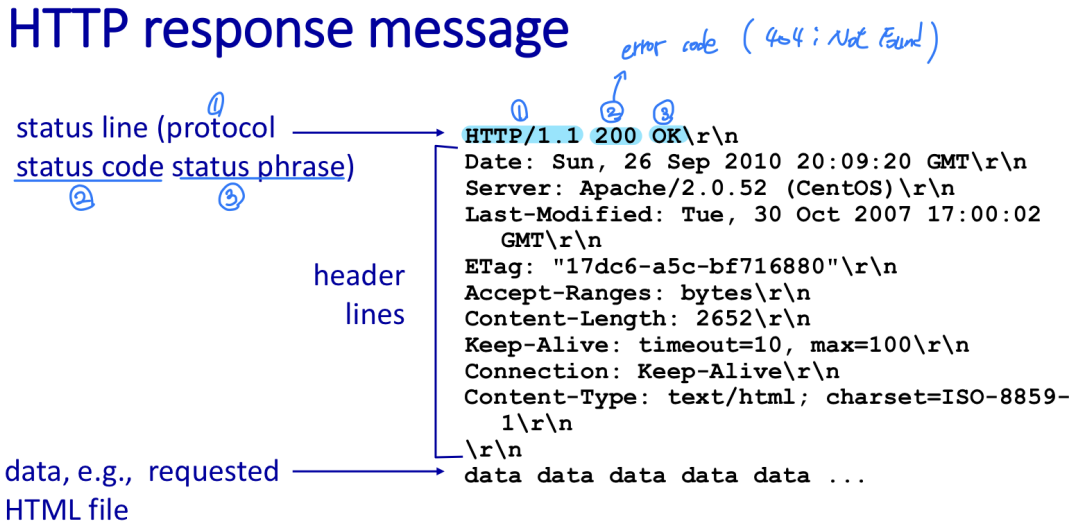
POST ⇒ URL 에 내가 요청한 정보 숨김

GET ⇒ URL 에 내가 요청한 정보가 나옴

HEAD ⇒ 디버깅 용도로 만든거, Body 영역을 통으로 부르면 트래픽이 너무 올라가니까

PUT ⇒ GET 이랑 반대 이건 upload 할때 요청하는거

HTTP response message



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Application Layer:

HTTP response status code

저기 보면 error code 가 있는데 (200) 번호 대 별로 의미가 달라

- 200 번대: 정상적으로 도착
- 300 번대: page가 주소를 이동한 경우 (redirection)
- 400 번대: client 측 에러
- 500 번대: server 측 에러

Trying out HTTP (client side) for yourself

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

- opens TCP connection to port 80 (default HTTP server port) at gaia.cs.umass. edu.
- anything typed in will be sent to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

- by typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. look at response message sent by HTTP server!

(or use Wireshark to look at captured HTTP request/response)

그냥 한번 읽어보면 될듯

Maintaining user/server state: cookies

Http의 stateless를 보완하기 위해 등장

매번 방문할 때마다 직접 로그인 하기 불편하니까 생긴거

생각해보면 사이트 잠시 나갔다가 들어와도 로그인 돼있는 경우 그거 생각하면 됨

stateful protocol

1. client가 server에 record에 lock 요청
2. server가 response로 ok
3. client가 update 요청
4. server가 response로 ok
5. client가 unlock 요청
6. server에서 lock 해제

Maintaining user/server state: cookies

Web sites and client browser use *cookies* to maintain some state between transactions

four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

local browser disk에 기록이 될

Example:

- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

HTTP cookes: comments

HTTP cookies: comments

사가 session을 만듦

What cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

Challenge: How to keep state:

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: HTTP messages carry state

단 그 웹서버 내에서만 알수있어

쿠키가 내가 뭘 검색했는지 알수있어

aside

cookies and privacy:

- cookies permit sites to *learn* a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

그럼 A 와 B가 정보공유하면?

뭐 망했다고 봐야지

보안 쪽에 우려가 있을 수 있는데, 그나마 막아놓은 선이 그 웹서버 내에서만 알 수 있게끔

근데 다른 웹서버끼리 공유한다? 그럼 개인정보 유출 되는거지 뭐

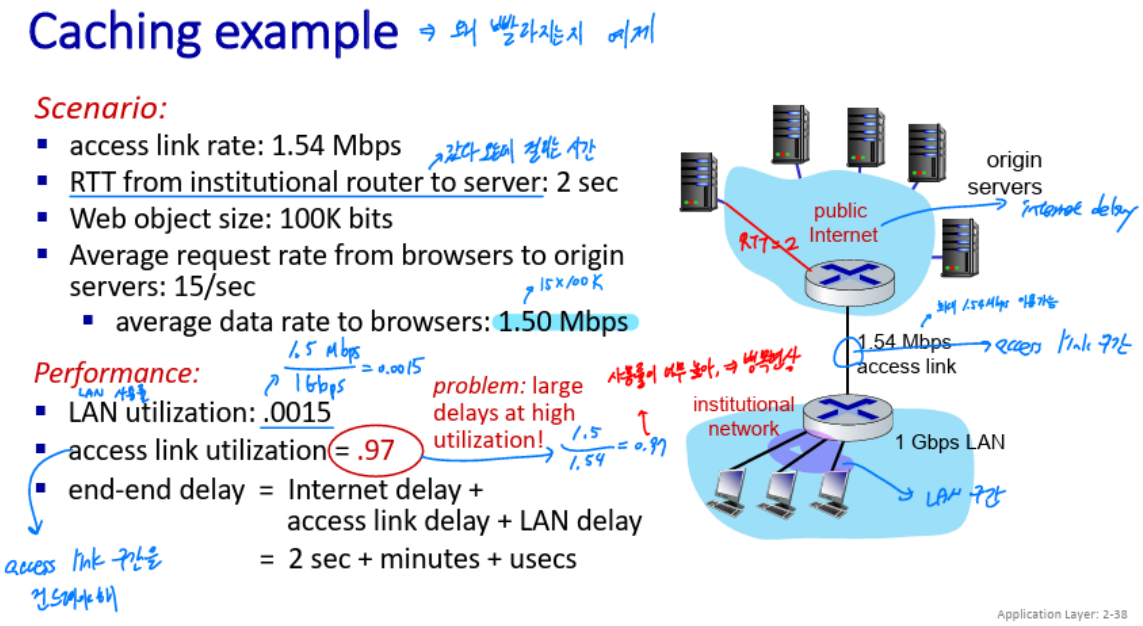
Web caches(proxy servers)

일단 이름에서부터 알 수 있듯이 속도를 향상하기 위해 만든거

그래서 local 쪽에 server를 하나 둬서 실제 서버와 통신하는 횟수를 줄이고 가능하면 proxy server와 통신하도록

근데 이렇게 하다보면 문제가 proxy server의 데이터와 실제 서버의 data가 일치하지 않는 문제가 발생(cache coherence problem)

Caching example



상황이 주어지는데

access link rate(server와 local 사이의 통로)

RTT: 2sec

data rate to browsers: server에서 보내고자 하는 데이터 양(Web object size * request rate)

Performance

Lan 사용률: 0.15% (강의 중에는 1.5%라고 했는데 0.15가 맞는 거 같은데)

⇒ access link가 1.54Mbps까지 허용하니까 1.50 Mbps까진 전송 가능

access link 이용률 = 97% (1.5 / 1.54)

이렇게만 보면 괜찮은거 아닌가? 라고 할 수도 있지만,

데이터를 순차적으로 하나씩만 보내는게 아니라 연속적으로 보내기 때문에 문제 있어

해결책 1

가장 무식한 해결책 ⇒ access link 크기를 늘리는 방법

근데 이렇게 하다보니 비용이 너무 비싸

해결책 2

web cache 사용

이거 사용하면 access link 이용없이 server에 접근 가능

그래서 상황을 가정해보면

cache hit rate: 0.4, 40%

access link에서는 60%의 request만 받는다고 가정

⇒ 전체 데이터 전송률 1.5Mbps * 0.6 = 0.9Mbps (40%는 캐시에 존재)

access link utilization = 0.9 / 1.58 = 0.58

그래서 이거 계산해보면 1.2초 정도 나오는데

이게 access link 10배 늘린것 만큼의 시간과 동일

Conditional GET

request, response message를 통해 마지막으로 수정된 날짜를 확인

만약 cache가 최신이면 캐시 데이터에서 그대로 받아옴

아니면? 서버에서 데이터 받아다가 캐시 갱신 해줘야돼

⇒ if-modified-since: <date> 로 마지막 갱신된 시점 확인

E-mail

user agents: client

mail servers: server

simple mail transfer protocol: SMTP

⇒ 이걸 활용해서 메일 전송하는거

근데 SMTP 뭔지 몰라서 여기까지...