

* application 계층

application

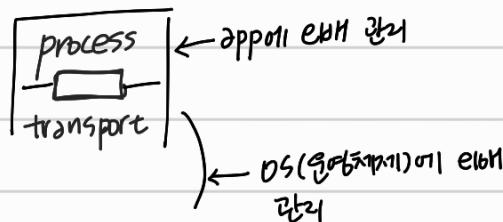
transport

network → 이 계층에서 통신?

data link

physical

① Socket : 메시지를 내보내고 받는 동로 → 메시지 아님!



1) TCP / UDP 차이점.

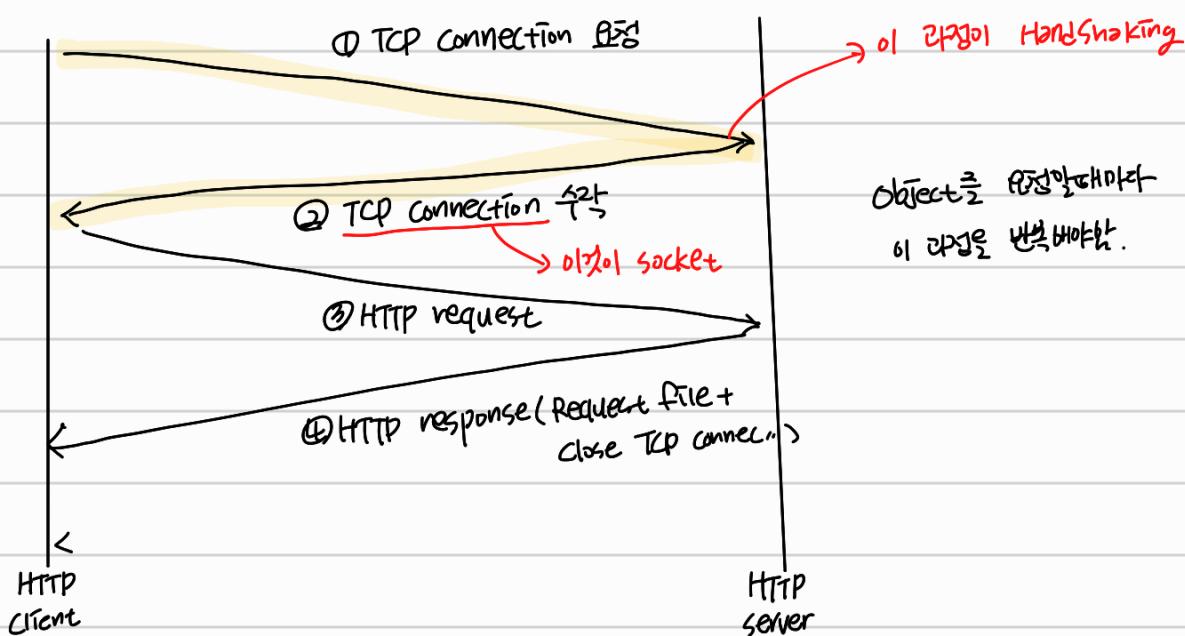
- TCP : 신뢰할만한 통신서비스 (loss, delay)가 거의 없음 → 손실이 있어서는 안되는 email, web ..
- UDP : 신뢰X인 " , 복수X → 손실보다 속도가 더 중요한 YouTube ..

* SSL : 암호화된 TCP connection을 제공함.

2) HTTP : application 계층에서 browser(HTTP Client)와 Web server(HTTP Server) 사이에 교환하는 메시지

→ 서버는 객체 요청을 기억하지 않음.

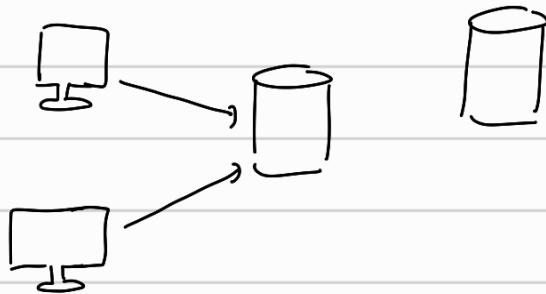
- Non-persistent HTTP : Client와 server 사이에 메시지를 보낼 때마다 TCP connection을 재생성함.



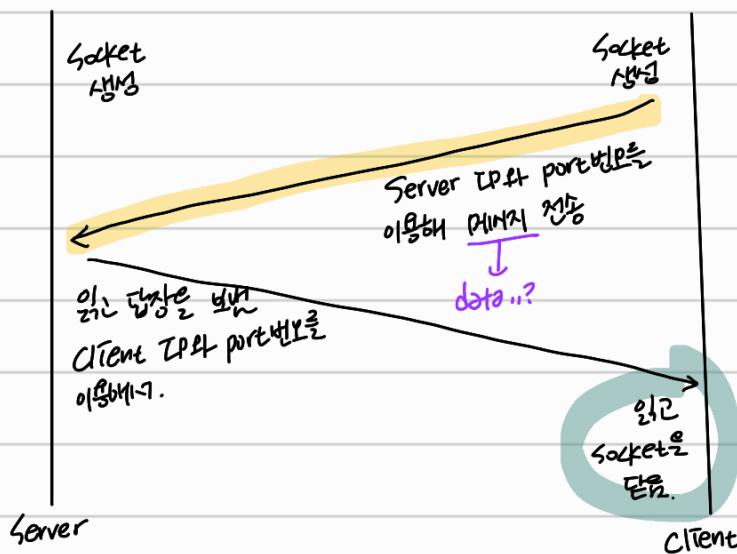
- persistent HTTP : Client와 Server 사이에 한 번의 TCP connection을 여러 메시지를 보낼 수 있음.
 → 한 번의 TCP socket

* HTTP Request · Response Message : header + body

* Web Cache : CPU 업로드 저하에 있음 → 시장단축 · Network Traffic 감소



3) Socket 통신과정 (UDP)



- Python Socket Code

- UDP Client

```
from socket import * → Socket은 OS에서 기본 제공
```

```
serverName = 'hostname' = 오스드 도메인
```

```
serverPort = 12000 → IP4, AF_INET은 IPv4, UDP는 UDP
```

```
ClientSocket = socket(AF_INET, SOCK_DGRAM) → 소켓 생성
```

메시지

```
message = .. → 보낼 때 암호화된 문자열을 bytes.
```

전송

```
ClientSocket.sendto(message.encode(), (serverName, serverPort)) → Socket에 보낼 정보를 부착
```

받기

```
modifiedMessage, serverAddress = ClientSocket.recvfrom(2048) → 2048 bytes를 최대한 받아온다.
```

Socket 닫음

```
modifiedMessage.decode()
```

```
ClientSocket.close()
```

→ Socket에서 최대 2048 bytes를 읽겠다.

- UDP Server

```
from Socket import *
```

```
serverPort = 12000
```

serverSocket = Socket (AF_INET, SOCK_DGRAM) → 소켓생성

serverSocket.bind ("", serverPort)

while True :

message, ClientAddress = ServerSocket.recvfrom(1024)

modifiedMessage = message.decode().upper()
byte를 문자로.

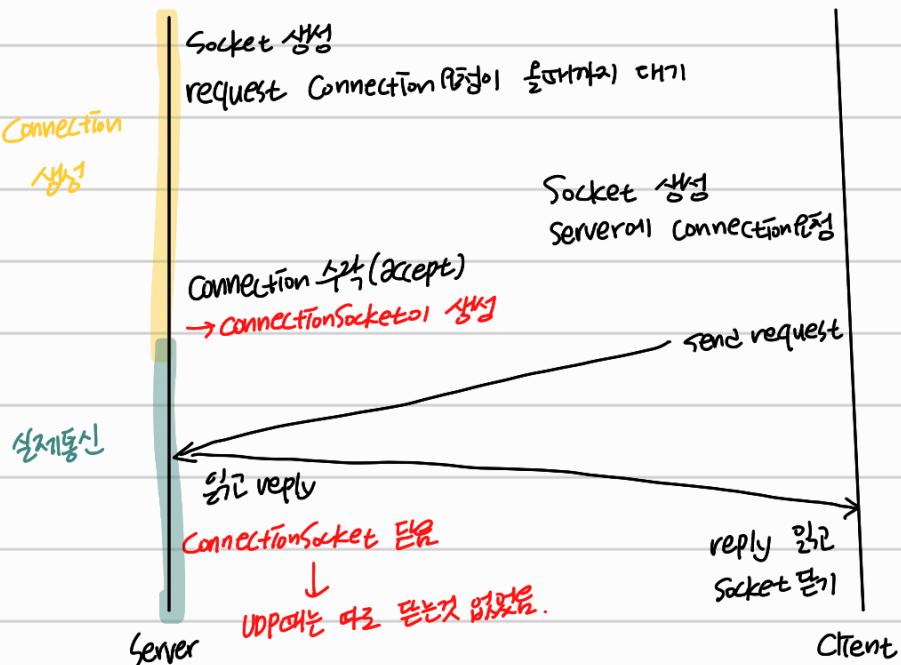
serverSocket.sendto (modifiedMessage.encode(), ClientAddress)

받은 메시지 인의 주소를 그대로.

이부분
다른곳에서
공부해야하는듯

데이터를
받고 있는
답장

4) Socket 통신관련(TCP)



- Python Socket Code

- Client

```
from Socket import *
```

```
ServerName = 'Servername'
```

```
ServerPort = 12000
```

TCP 방식

```
ClientSocket = Socket (AF_INET, SOCK_STREAM)
```

```
ClientSocket.connect ((servername, serverPort))
```

ClientSocket.send (sentence.encode()) → 이부분에는 sendto가 아니
ServerName, ServerPort 필요

```
modifiedSentence = ClientSocket.recv(1024)
```

```
ClientSocket.Close()
```

Connection
생김

통신관련

- Server

```
from Socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(("", serverPort))
```

serverSocket.listen(1) → 미리 TCP connection 풀을 기다림

while True :

```
connectionSocket, addr = serverSocket.accept() → Connection 풀을 수령  
serverSocket가 ConnectionSocket을 미리 대기!
```

```
sentence = connectionSocket.recv(1024).decode()
```

```
connectionSocket.send(sentence.encode())
```

connectionSocket.close() → ConnectionSocket을 해제.

2] WebSocket

- Server : Node.js로 연결? → local

- Client : Android? + EMULAT? + Ubuntu? + docker?

외부접속이 되나?

AP 외장이 안된다.

→ 2단계 Server - Client 연결 만들어두고 Client, Client는 2단계로!

→ HTTP 기본 통신

→ 이미 자주 쓰이는 듯..

WebSocket : Node.js에서 socket.io 라이브러리나 ws 라이브러리를 사용해 가능

Socket programming : Node.js에서 net 모듈을 사용해 가능

* Svelte는 Client의 역할밖에 하지 못함! → 나중에 Client는 Android로 구현할 거 때문이!

Svelte - Node.js는 코드만 한번 짜본다! 추가.

1) Svelte 기본 설정 명령어

npx dgit sveltejs/template svelte-app → npm install -g : npm을 지역으로 설치하는 명령어

npx : npm 자체에서 실행 도구, npm이 전역으로 설치되어 있지 않더라도 npm이 바로 실행되게 하는-

dgit : git과 같은 git 저장소에서 프로젝트의 초기 템플릿을 복사하는 도구

→ Svelte의 공식 템플릿인 sveltejs/template을 복사해온다.

svelte-app : 만들 파일 directory 이름.

npm create vite@latest frontend -- --template svelte

npm create : 프로젝트 스키마팅을 위한 명령어
↳ 프로젝트를 시작할 때 기본적인 구조와 설정을 자동으로 생성해주는 명령

vite@latest : 최신버전의 vite 패키지를 가져온다.

*vite : 빌드 도구, 프레임워크의 초기설정과 템플릿을 가져와주는 도구.

vite@latest 프로젝트 이름 -- --template 가져온 프레임워크 이름.

frontend : 디렉토리 이름

template svelte : 사용할 프레임워크 이름

⇒ 둘 다 Svelte의 초기설정을 하는 명령어!, 각각 상관없음!

2) 라이브러리

- express : 웹 서버 프레임워크

- socket.io : WebSocket 라이브러리

3) WebSocket - Node.js의 Socket.io 라이브러리 + express 프레임워크

- Server

모듈을 가져오기 다른 파일이나 모듈 가져오는 변수.
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');

HTTP 서버 Application이라는 개념, 여기서 get, post... 같은 HTTP 요청을 처리하는 함수를 가지고 있음.
const app = express();
const server = http.createServer(app);
const io = socketIo(server);
HTTP server Instance가 필요함
→ WebSocket은 초기연결은 HTTP로 수락한 후
비대칭 연결을 업그레이드하여 지속적인 연결 유지·데이터 교환
serverSocket → Application 개념을 기반으로 하는 http server.
= request handler
= connection event handler = ~~Client에게~~? Client에게 io.connect 메시지를 통해 소켓연결을
설정되는 이벤트 handler.

이 부분은
connect 부분이고
다른 기능은
구현되어야 함
io.on('connection', (socket) => {
 console.log('New client connected');

socket.on('message', (message) => {

io.emit('message', message);

}); 모든 Client의 message를 보냄. ⇒ 즉, 모든 Client의 message event를 전송.

```
socket.on('disconnect', () => {
  console.log('Client disconnect');
});
```

server.listen(4000, () => console.log('Listening on port 4000'));
→ 4000번 port에서 대기 중.

- Client

<Script>

```
import { onMount } from 'velte';
let messages = [];
let newMessage = '';
```

→ 템포리지가 로딩되면 바로 실행되는 힘

onMount(async () => {

초기
데이터를
가져온다

const response = await axios('/messages');

const messages = response.data;

server.js에서 만들어야하는 기능

app.get('/messages', (req, res) => {

res.json(messages);

messages 배열은 JSON 형태로
변환한 후 client에게 전달

const socket = io.connect("http://localhost:4000");

이전에 X

이후 모든 message 이벤트가 발생하는 때마다

socket.on("message", (message) => {

messages.push(message);
 });
 newMessage = '';
});

messages.push(message);
 → 배열에 메시지 추가하기 newMessage 초기화

인가?

newMessage = '';

});

});

function sendMessage() {

const message = {

text: newMessage,

→ 새로운 메시지

timestamp: new Date().toISOString(),

});

axios.post('/messages', message); 또는 axios({

url: '/messages',
body: message

method: 'post',

url: '/messages',

header: {

'Content-Type': 'application/json'

},

data: message

</script>

모든 Client의
message 이벤트를
수행.

app.post('/messages', (req, res) => {

const message = req.body;

messages.push(message);

io.emit('message', message);

res.status(201).end();

});

* res.end()

- res.end() : Express.js에서 응답 process를 종료하는 메서드, Client에게 어떤 data도 보내지 않고 단순히 응답 process를 종료.

ex) post 메시지에서 data를 브로드캐스팅해 응답을 종료해야하는 경우

- res.status() : 인자로 받은 값을 status code로 설정하는 것.

* res.json()

(res.end()를 사용하고 싶은 경우)
- res.json() : 인자로 받은 값을 JSON 형태로 변환 후 Client에게 전달

- res.send() : 인자로 디양한 타입을 받을 수 있고 (ex. 문자열, 객체, 배열...), Client에게 전달

⇒ Node.js가 아니라 FastAPI에서 python을 이용해서 한번 해보기!

Server

4) 기능구현 : REST API 엔드포인트 같은 HTTP 기반 응답 or Socket.io로만 구현

① REST API 엔드포인트

```
app.post('/post-message', (req, res) => {
  const reqMessage = req.body;
  messages.push(reqMessage);
  To.emit('message', reqMessage);
  res.status(201).end();
});
```

② Socket.io

```
socket.on('message', (reqMessage) => {
  message.push(reqMessage);
  To.emit('message', reqMessage);
});
```

= 같은 기능임

Client에게 정의된 message event를 발생시킴

- Client에서 호출

① REST API 엔드포인트

```
axios.post("http://localhost:8000/post-message", message);
```

② Socket.io

```
const socket = Io("http://localhost:8000");
socket.emit('message', message);
```

event 호출 함수

* on: event 수신 함수

→ 일반적으로는 REST API와 Socket.io를 같이 사용함.

* RestAPI GET 부분 → Server에서 서버 메시지가 올 때마다 Client의 message event를

받을 때까지 서버 메시지를 전달하고 Client에게 배열에 추가하고

이면 내용에 있는
Client는 어떤 내용을
알지 못함.

마음에 드는 때문에 굳이 Server에서 messages 배열을 보내는
API가 필요하지 않음.

그렇다고 RestAPI가
아직도 어떤 내용이 불러와야 할지 알 수...

근데 현재 불러와야 하는지 어떤가?

아직 어떤 내용인가...

- 미친 모드 ㅋㅋ

① server.js - message = ['안녕하세요'];

→ Client에서 message 배열안의 text, timestamp 항목을 나열하기 때문에!

배열안에 text, timestamp가 없으면 안됨!

② Client.svelte - const messages = response.data

→ response.data에는 배열이 있음!, response.data[0] 를 보면 제이슨 dictionary 나옴!

→ 배열끼리 합치는건 for (messages in ..) 이런거 앤하고

messages = [...messages, ...response.data] 이런식으로 한번 만번에 합쳐짐

3 | Socket Programming

* 프로그래밍언어 개념정리

. 웹 프레임워크 : 웹 사이트나 웹 서비스를 개발하는 소프트웨어 프레임워크

→ HTTP 요청 응답, 세션관리, 라우팅

. 웹 애플리케이션 프레임워크 : 웹 애플리케이션에 필요한 기능하는 프레임워크.

→ 데이터베이스 연결, 템플릿 엔진 ..

. 웹 서버 프레임워크 : HTTP 요청 응답에 초점을 맞춘 프레임워크

→ Web Framework 와 동일한 의미.

⇒ 서버가 용어를 구별하는 쓰기보다는 같은 의미로 사용함!

- Server

- FastAPI : 웹 프레임워크로 Socket 통신을 구현하기에는 별로 적절하지 않은
→ 애도 Client의 역할
- Python : Socket 라이브러리 + 멀티스레딩을 사용해 구현
- Java : Socket, ServerSocket 라이브러리 + 멀티스레딩 ..

이 개념과 사용방법은 학습이 해야하는데..

- Client - Android Studio

1) Thread (= Multi Thread) : 여러 작업을 동시에 수행하기 위해 사용하는 개념
→ 프로세스 안에서 동작하는 작은 단위.

```
new Thread() {
    public void run() {
        // 작업할 코드
    }
}.start();
```

2) Server Code - Java

- Myserver.java → 클라이언트가 연결될 때마다 새로운 thread를 생성하는 서버코드.

```
import java.io.IOException; → 오류처리
import java.io.PrintWriter; → getOutputStream()을 처리
import java.net.ServerSocket; → ServerSocket
import java.net.Socket; → ConnectionSocket
import java.util.ArrayList; → ArrayList 정의된 List
```

```
public class Myserver {
    public static ArrayList<PrintWriter> m_outputList; → 각 클라이언트의 출력스트림을 저장하는 List.
    public static void main(String[] args) {
        m_outputList = new ArrayList<PrintWriter>();
    }
}
```

try 1

ServerSocket 생성, Listen() 상태
ServerSocket s-socket = new ServerSocket(2000);

while(true){

Socket c-socket = s-socket.accept(); Connection 요청을 받아들이면
Connection Socket을 생성.

Client Manage Thread c-thread = new ClientManageThread(c);
별도 Thread 생성

c-thread.setSocket(c-socket);

제거됨

m-outputList.add(new PrintWriter(c-socket.getOutputStream()));

c-thread.start();

Thread 실행

Socket 객체의 출력스트림을 가진다는 메서드
텍스트 출력 기반 매핑
: 텍스트 데이터를 쉽게 사용할 수
있게 만든다

: 소켓을 통해 데이터를 보내는 방식
스트림 (여기서는 Server → Client)

* Stream : 연속적으로 단방향으로 끌어가는 것 → 데이터가 지나가는 통로
→ 단방향이기에 하나의 Stream으로 입출력이 가능.

ex) Java

OutputStream : 데이터를 보낼 때

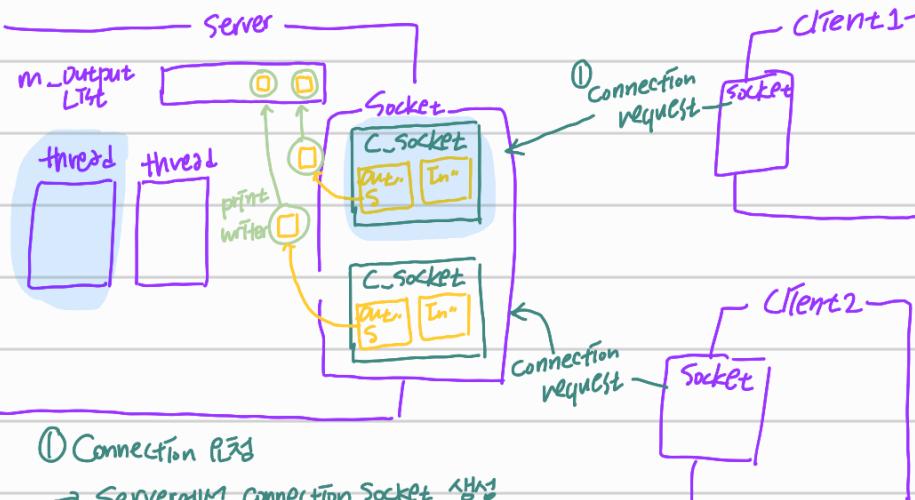
InputStream : 데이터를 수신할 때

C-thread : 클라이언트와의 통신을 관리

→ 해당 Client로부터 데이터를 읽어들여서 연결된 모든 Client와 데이터를 전송.

⇒ 그동안에 출력스트림을 List에 추가하고 클라이언트 관리 Thread가 시작되어야 함.

? 그럼 왜 상태가 이렇게 되는가?



Q1. OutputStream은 서버에서 Client에게
보내는 메시지가 담겨 있는데 예전에는
문제가 있었나? Client가 전송한
메시지?

: OutputStream은 데이터를 보내기 위한 통로.

m-outputList에는 각 클라이언트의

OutputStream이 저장된다! thread 마다

이 List에 OutputStream을 통해 연결된

Client와 메시지를 전송함.

② Client와 연결될 때마다 둘이 반세트!

Connection Socket과 Connection(?) Thread가 생성됨

→ 각각의 Client 연결은 별도의 socket과 thread에 의해 관리.

Q2. Client와 연결될 때마다 C-socket이
생기는데, C-thread도 마찬가지인가?

: oo, Client와 연결될 때마다 C-socket과

C-thread가 생성됨.

```
}] catch (IOException e) {
```

```
    e.printStackTrace();
```

```
}
```

Socket 연결해..

이때 buffer에 저장이되고
바로 전송이 안될 수 있음
(buffering방식의 표본)

```
ex) println(String s)
```

→ 문자열 S에 개별문자 (/n)을

Q3. PrintWriter는 텍스트기반 출력 매핑인데

getOutputStream은 데이터를 보내기 위한 통로인데

왜 같이 사용되는가?

: 컴퓨터네트워크로 data를 전송할 때는
byte 형태로 전송되어야 하기 때문에 PrintWriter는

문자열을 byte로 바꿔주는 역할을 한다.

* Java의 System.out.println()에서

System.out은 PrintStream 객체이다.

PrintWriter와 비슷하지만 여기서는

Console과 연결되어있음.

추가한 후 byte로 변환 후

OutputStream에게 전달

→ OutputStream은 들어온 S를

Client에게 전달.

PrintWriter는 생성 시 인자로 OutputStream을

받아서 변환한 byte data를 OutputStream에

보낸다. 즉 print

Server → PrintWriter → OutputStream →

Network → Client 순서로 data가 전송됨.

flush()

flush를 통해
buffer에 저장된 것을

→ PrintWriter의 버퍼에 저장됨

바로 OutputStream에 모든 data를 OutputStream으로 즉시 전송하는 버퍼드 전달.

→ OutputStream은 data를 모아두었다가 한번에 전송하는 버퍼링(buffering)을 사용.

그리기에 flush를 사용해 즉시 모두 전송.

- ClientManagerThread.java

```
import java.io.BufferedReader;
```

```
import java.io.IOException;           → IOException
```

```
import java.io.InputStreamReader;
```

```
import java.io.PrintWriter;          → List안의 PrintWriter 객체의 메소드 사용하려고
```

```
import java.net.Socket;             → Socket
```

```
public class ClientManagerThread extends Thread{
```

```
    private Socket m_socket;          → Connection Socket을 담기 위한 Socket 변수
```

```
    private String m_ID;              →
```

① Override

```
    public void run(){
```

try {

 BufferedReader in = new BufferedReader(new InputStreamReader(m_socket.getInputStream()));
 String text;
 // 텍스트를 버퍼링.
 // 한번에 하나의 문자가 아니라
 // 여러개의 문자를 한번에 읽어서
 // 버퍼에 저장.

 while (true) {

 text = in.readLine(); // 입력에서 한줄 (\n을 끝내는 text)을 데이터를 얻음.

 if (text != null) {

 // OutputStream이 저장되어 있는 List

 for (int i = 0; i < Myserver.m_outputList.size(); i++) {

 Myserver.m_outputList.get(i).println(text); // 현재 Client의 OutputStream

 Myserver.m_outputList.get(i).flush();

 // text를 byte 형태로 변환 후 전송하거나
 // buffer에 저장

 } // 다른 Client의 OutputStream

 printWriter가져와 buffer에 저장된 것을 바로 전송.

 }

 } catch (IOException e) {

 e.printStackTrace();

 }

}

 public void setSocket(Socket _socket) {

 m_socket = _socket;

 // Connection socket을 저장할 변수.

 }

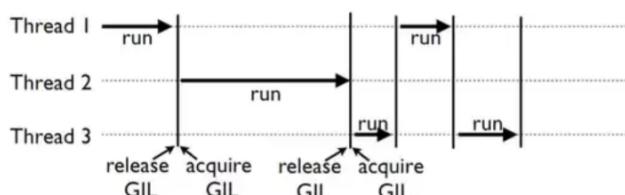
}

C-thread.setSocket(c-socket);

3) Server Code - python ~~쓰레드는 python이 수동적이다~~

* GIL (Global Interpreter Lock) : Python에서 하나의 스레드만 하나의 바이트코드를 실행시킬 수
 있기 때문에 lock.

→ 이것은 때문에 Python에서 멀티스레드의 성능은 좋지 않으면.



	Java	Python
데이터 입/출력	OutputStream API PrintWriter 라이브러리 필요 InputStream API InputStreamReader, BufferedReader 라이브러리 필요	Socket 객체 자체에서 recv, send를 통해 데이터를 읽고 쓸 수 있음.
비동기		GIL 때문에 멀티스레드의 성능이 제대로 나타나지 않음.
멀티스레드	멀티스레드를 이용해 구현 가능	비동기라이브러리 asynio를 사용해야함.
멀티프로세싱		asynio에서는 코루틴과 핸들러, async와 await를 사용해 함수를 비동기로 실행한다.

* 비동기 프로그래밍 vs 멀티스레드

⇒ 개념과 방법은 다르지만 둘다 병렬 처리가 가능함.

- 비동기 프로그래밍 : 하나의 스레드 안에서 여러 작업을 동시에 처리함.
 → I/O 작업 (네트워크 요청 등)의 작업을 기다리는 데 많은 시간이 소요되기 때문

멀티스레드 : 실제로 여러개의 스레드가 동시에 실행돼 각 작업을 처리함.
 → CPU 작업 (복잡한 계산)의 경우, CPU의 여러 코어를 활용 가능.

* 코루틴 : 일반 함수와 다르게 실행을 중단하고 나중에 계속할 수 있는 기능을 가지고 있는 함수.

→ yield를 사용해 중단점을 설정하거나 python 3.5 이상에서는 async def를 사용.

↳ Unity에서는 yield를 사용해보자

yield return new WaitForSeconds(3); 이렇게!!

* asynio : 네트워크 입출력을 위한 스트림을 제공하는 라이브러리

→ 스트림은 내부에 소켓을 포함하고 있으며 소켓을 읽고 쓰는 작업을 용이하게 함.



① 이벤트루프

```
loop = asyncio.get_event_loop()
```

loop.run_until_complete(코루틴 함수 이름) → 인자로 받은 코루틴 객체의 소행이 끝날 때까지 기다림.
loop.close()

코루틴 함수를 노출하면 코루틴 객체가 생성됨.

② await : 해당 객체가 끝날 때까지 기다린 뒤 결과를 반환, 코루틴 안에서만 사용이 가능.

- 사용 예시

- [변수 = await 코루틴 객체] + await asyncio.sleep(0.5)
- [변수 = await 퓨처 객체] ↗ asyncio에서 제공하는 클래스
- [변수 = await 태스크 객체]

③ Client - asyncio.open_connection(host, port) → 서버에 Connection을 요청

Server - asyncio.start_server(handle_conn, host, port) → 서버에 생성, bind, listen 작업

→ 연결을 처리하는 핸들러 함수 - 비동기 코루틴 함수 / 인자로 StreamReader, StreamWriter를 받음.

④ 스트림 → 이를 직접 생성하기보다는 open_connection이나 start_server 등의 코루틴 함수로

내부에 자동 생성된 스트림을 핸들러가 사용하는 것이 좋음.

• 출력스트림 - asyncio.StreamReader

- [read(n) : 최대 n byte 만큼 읽고 일어들인 byte를 반환]
- [readline() : 개행문자 (\n) 까지 읽은 후 byte를 반환] ↗ 코루틴에서도 await와 함께 사용.

• 입력스트림 - asyncio.StreamWriter

- [write(data) : data를 기록한다. → 끝에 drain() 를 꼭 실행해야 함]
- [drain() : 스트림을 다시 사용할 수 있을 때까지 기다린다.]
- [close(), is_closing() : 스트림을 닫는다, 스트림이 닫혔는지를 확인]
- [wait_closed() : 스트림이 닫힐 때까지 기다림.]
- [get_extra_info('name') : socket의 정보를 불러온다. (내부에 다른게 올수도 있지만 여기서는 socket만 다룬다.)]

 [peername : socket이 연결된 원격 주소]

 [socket : socket 인스턴스 (=socket 객체)]

 [sockname : socket 자체의 주소]

Import asyncio

Clients = [] Dictionary 형태로 ClientName : writer 정보를 담음

ol handler는 Client가 연결될 때마다 생성, Client의 연결을 깊숙이 관리
async def handler-client(reader, writer) → 인자로 자동으로 reader, writer 객체가 생성됨
이 인자는 각 Client의 reader, writer 객체임.

data = await reader.read(1024) 사용자에게 data를 받아옴

nickname = data.decode().strip() data를 String으로 바꾸고 strip()으로 공백, 개별문자를 제거.

Clients[nickname] = writer → Clients dictionary에 nickname과 writer 객체 저장

Writer 객체를 저장해야 Client에게 메시지를 전송할 수 있음.

await broadcast_message("[" + nickname + " joined the chat")

= f"[{nickname} joined the chat"]

while True : 여기서 무한으로 도연서
Client가 통신하는

try :

2명

data = await reader.read(1024)

message = data.decode().strip()

if message :

이 핸들러는 Client로부터 받을 때마다
파문에 굳이 dictionary에 각 객체를
저장할 필요가 없음.

broadcast_message("[" + nickname + " : " + message)"))

except ConnectionResetError : Client가 연결을 끊으면

del Clients[nickname] dictionary에서 Clients 삭제

await broadcast_message(f"[{nickname} left the chat")

break

except Exception as e :

print(f"Error Broadcasting message : {e}")

Client에게 메시지를 보내는 함수.

async def broadcast_message(message) :

print(message)

for client in Clients.values() :
→ writer 객체

try :

Client.write((message + '\n').encode()) Client에게 메시지를 보냄.

await client.drain() write와 반대로
작동되는 read 속도에서 각 메시지를
새로운 줄에 메시지를 보내기 때문에.

except Exception as e :

print(f"error : {e}")

스트림을 다른 사용할 수
있을 때까지 기다림.

server & handler 쌍

```

async def start_server(host, port):
    server = await asyncio.start_server(handler_client, host, port)
    
```

connection socket 생성, bind, listen

이 번들러자는 쿠루틴, 스케줄, 프로세스도 될 수 있음.
여기서는 쿠루틴!

```

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    print("server start")

```

try :

```

        loop.run_until_complete(start_server('localhost', 8000))
    
```

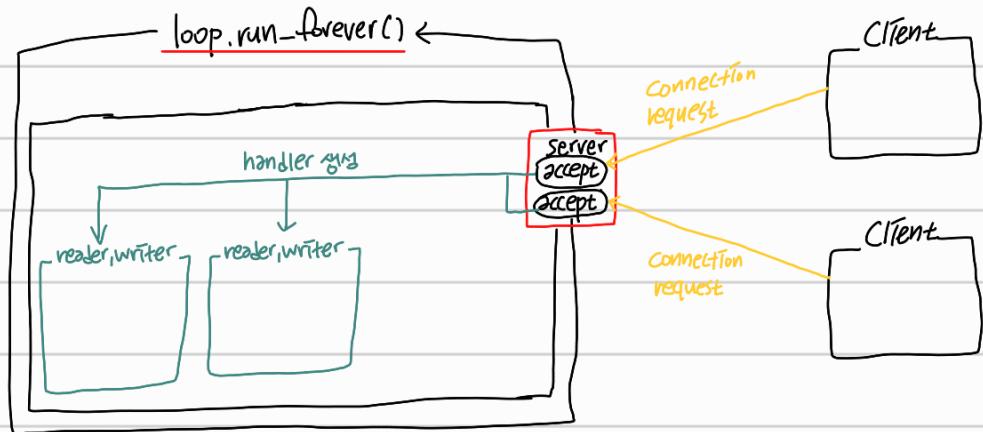
loop.run_forever() 미벤드로프를 시작하고 계속 실행하는 역할 → 계속 새로 시작하는 것이 아니라
서로운 이벤트를 생길 때까지 대기,
부에 오류가 되면 블랙홀이나 쿠루틴 실행

except KeyboardInterrupt :

```
print("Shutting down server")
```

여기서는 생성된 서버 socket, server가
listen 상태로 기다릴 수 있게 함.
client 요청이 오면 핸들러가 생성

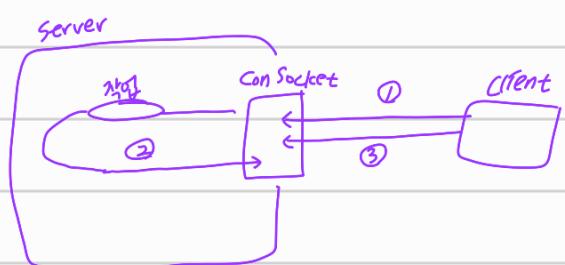
: listen 셋팅 초기



test-client.py 이 코드가 문제였는데

Q1. Client에서 receive가 안된다고 왜 데이터를 전송하는 것까지 막혀버림? ~~이것땜에 3시간걸림~~

추측1. reader와 writer는 같은 connection socket을 사용하기 때문에..



①이 문제가 생겨서 ②이 안되는건가..?

②가 막혔다? ②가 입구를.. 막고있나..?

Socket 내부의 버퍼가 차웠나..? Socket 내부버퍼?

Socket의 스트리밍이 차렸나? ???

gather로 대로 input이 불로깅연산에 receive가 막혀있었는데..

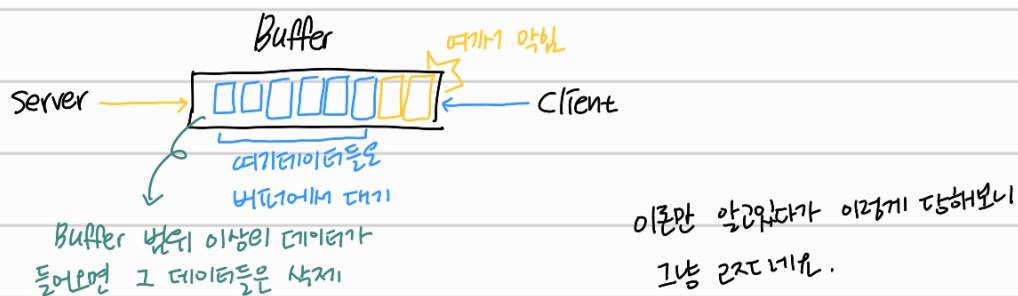
이 블로그가.. 근데 그럼 입력값을 대 대로 풀려야되는데 아닌가?

불고기 문제였던가 맞지만 직접적인 원인은 아님
무엇이...

⇒ TCP/IP 프로토콜의 끌제어(flow Control) → 이것이 UDP와 차이점이기도 함!

- 송신서버와 수신클라이언트의 데이터처리속도 차이로 인한 수신버퍼 overflow를 해결하기 위한 기법
- TCP/IP 프로토콜은 양방향 통신을 지원, Client가 Input() 함수에 의해 막히면 서버가 보낸 데이터를 받을 수 없고 네트워크상의 버퍼가 가득차면 데이터전송을 일시정지하여 buffer overflow를 방지.

따라서 데이터수신을 중단하면 서버쪽에서도 전송된 데이터를 받아들일 공간이 없어져서 데이터전송차단이다.



- 사용하는 클래스

① InetSocketAddress : IP주소를 나타내는 클래스

→ 도메인 이름 시스템(DNS)을 조회하는 메서드 제공

getByName(String host) : 도메인 이름이나 IP주소 받아 해당 호스트의 InetSocketAddress 반환
getHostAddress() : InetSocketAddress 객체가 가진 IP주소 문자열 반환
getHostName() : InetSocketAddress 객체가 가진 호스트명 반환
굵이 이 객체만고 String은 안되나
→ 되긴 하는데 InetAddress이 좀 더 많은 기능을 제공함.
이미 Socket이 인자에 사용
DNS, IPv4, IPv6 모두 지원.
일반적 인터페이스 제공.

② Socket vs ServerSocket

Client Code - Java에서
이미 사용됨

차이점
Socket : Client용 socket, 서버에 연결을 요청하고 Socket을 이용해 데이터 송수신
ServerSocket : Server용 socket, Client의 요청을 기다리고 소켓과 동시에 connection socket 생성, 이를 이용해 송수신

- new Socket(String host, int port) → 호스트명과 포트번호로 Socket 생성

InetAddress address, int port → InetAddress 객체와 포트번호로 Socket 생성

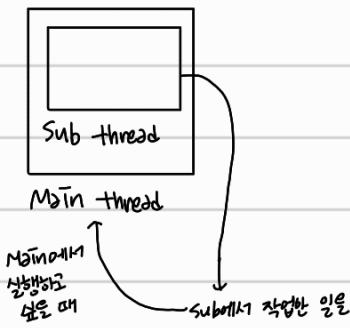
* host명 [domain]
IP 주소

③ Handler : thread간에 코드를 실행하도록 예약하는데 사용하는 클래스

→ 메시지나 Runnable 객체를 MessageQueue에 넣어 다른 thread에서 실행될 수 있게 한다.

- 사용되는 예시, 사용 이유

* thread는
별도의 공간이기에
저걸 넘보이며도
다른 공간임!



* MainThread : main() 함수가 실행되는 thread.

→ 모든 Android는 기본적으로 하나의 MainThread를 가지고 있음.

주로 UI 작업들이 실행됨.

★ MainThread에서 onCreate() 메서드가 호출되는 것임!

그렇기에 onCreate() 메서드 안에서 오래 걸리는 작업은 X!

• 보통 UI 업데이트와 관련된 작업을 할 때 사용.

→ 안드로이드의 UI 업데이트는 오직 Main thread에서만 가능하기 때문에

다른 thread에서 계산한 결과를 외부에 반영하는 등의 목적으로 사용.

보통 네트워크 작업이나 파일 입출력 등 오래 걸리는 작업

→ MainThread에서 시간이 오래 걸리는 작업을 하면 UI가 멈추거나 응답하지 않기 때문에

시간이 오래 걸리는 작업들은 별도의 background thread에서 실행해야함.

- API

post (Runnable r) : Runnable 객체를 현재 Handler의 MessageQueue에 추가.

postDelayed (Runnable r, long delayMillis) : Runnable 객체를 지정된 시간만큼 지연시킨 후
Runnable 객체 또는 Message 객체 messageQueue에 추가.

* Runnable : 인터페이스, 별도의 thread에서 실행되어야 하는 코드블럭 / run() 메서드 하나만을 가지고 있음.

ex)

```
public void onCreate ( " " ) {
```

Main Thread

```
    Handler handler = new Handler();  
    // handler는 기본적으로 자신이  
    // 만들었던 thread와 연결  
    // → 여기서는 MainThread
```

```
    new Thread(new Runnable () {
```

Sub Thread

```
        @Override  
        public void run () {
```

오래 걸리는 작업

```
            handler.post (new Runnable () {  
                @Override  
                public void run () {  
                    // MainThread에서 실행될 작업  
                }  
            });  
        }  
    }  
}
```

+> handler 말고 runOnUiThread 메서드도

Sub Thread에서 Main Thread로 작업 전달 가능

```
}).start();
```

→ Runnable 클래스를 상속하는 다른 클래스도 사용 가능

- 코드

① override

Socket을 종료하는 메서드

```
protected void onStop() {
    super.onStop();
    // 오류 있을까?
```

```
try {
    sendWriter.close();
    socket.close();
}
```

```
} catch ( IOException e) {
```

```
    e.printStackTrace();
}
```

```
}
```

```
}
```

② override

```
protected void onCreate() ~ }
```

```
super.onCreate(savedInstanceState);
```

```
setContentView(R.layout. ~ );
```

mHandler = new Handler(); 백그라운드 스레드에서 코드를 메인 스레드에서 실행할 수 있게 하는 handler.

Intent intent = getIntent(); → Intent로 변수를 전달할 수 있음.

userID = intent.getStringExtra("username"); Intent.putExtra("username", username)
startActivity(~)

Sub Thread 생성 - 솔직생성, 높여. 입력스트림 정의, 메시지 수신과 대기에 프리기(Handler)

```
new Thread() {
```

```
    public void run() {
```

```
        try {
```

Socket
생성

```
        InetAddress serverAddr = InetAddress.getByName(ip);
```

```
        Socket = new Socket(serverAddr, port);
```

여기서는 InetAddress 객체를 사용, 실제로는 String ip로됨.

높여, 입력
스트림 정의

문자열을 byte로! 가장자리 출력스트림(데이터 전송)
sendWriter = new PrintWriter(socket.getOutputStream());

BufferedReader Input = new BufferedReader(new InputStreamReader
 한번에 여러개의 문자를
byte를 문자열로!
(socket.getInputStream()));

메시지를 수신하고
백그라운드에서
前线에 표기!

```
while (true) {
```

```
    read = input.readLine();
```

코드집에서는
Log.d("message", read)

↳ System.out.println("메시지 수신 : " + read)

If(read != null) { 수신한 메시지는 null이 아닐 때만

mHandler.post(new msgUpdate(read));

메인스레드에서
Runnable 객체로
run() 실행되게

인자로 read를 넣어
Runnable 객체 생성

}

} catch (IOException e) {

e.printStackTrace();

} }.start();

Sub Thread 실행

메시지 전송

ChatButton.setOnClickListener (new View.OnClickListener() {

@Override

public void onClick(View v) {

SendMsg = message.getText().toString(); 사용자가 입력한 메시지

Sub Thread

new Thread() {

@Override

public void run() {

try {

OutputStream을

이용해 서버에
메시지 전송

sendWriter.println(userID + ":" + sendMsg);

text를 bytes로 변환 후 전송하거나 Bufferon

sendWriter.flush(); 저장

Buffer에 있는 것을 바로 전송

message.setText("");

} catch (IOException e) {

e.printStackTrace();

}

} Start();

}

인터페이스는 상속한 때 implements!

class msgUpdate implements Runnable {

private String msg;

클래스와 이름이 같은 메서드는 생성자!, 개체가 만들어질 때 실행되는 부분이자
조건!

public msgUpdate (String str) { this.msg; }

@Override

public void run() {

TextView에
내용을 넣는
Runnable 인터페이스를
상속하는 클래스.

chatView.setText(chatView.getText().toString() + msg + "\n"));

}

기존의 내용에 새 문자 추가