

A Top-Down Approach

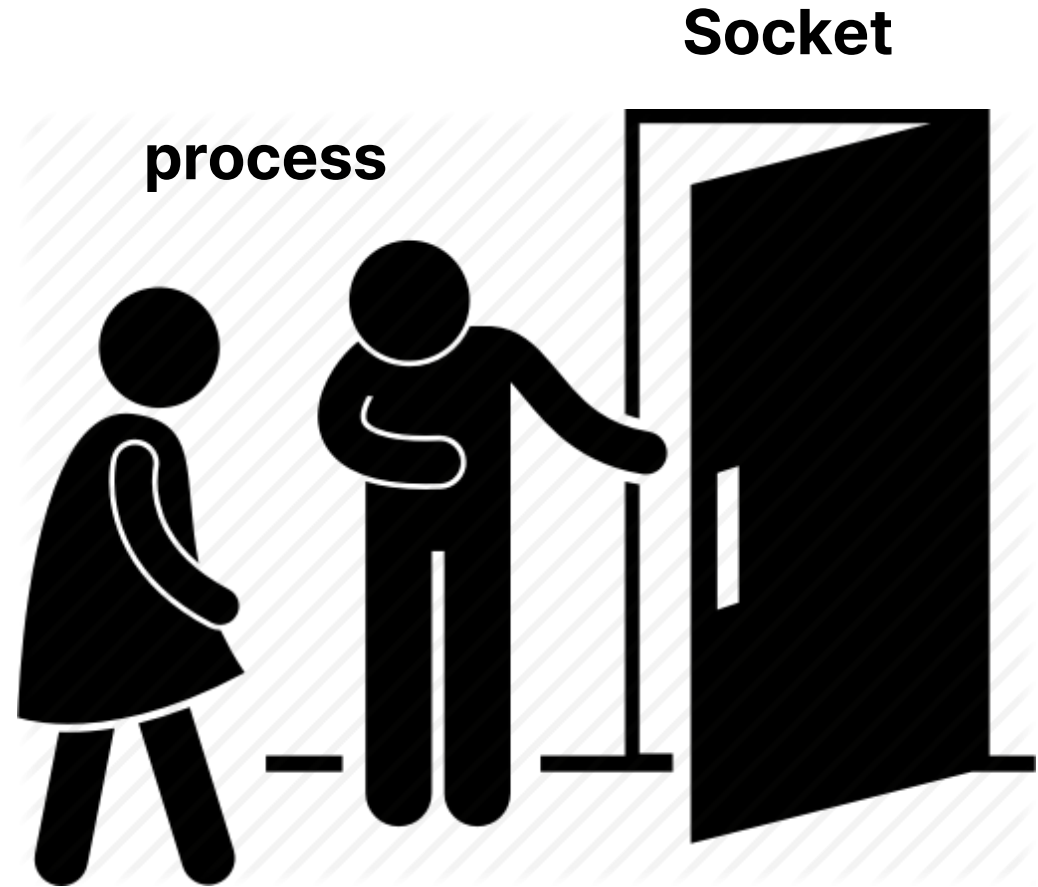
Chapter 2.7

2.7 Socket Programming: Creating Network Application

- Network application client와 server로 나누어 진다.
- 각각 client process, server process가 생성되어 socket을 통해 데이터를 읽고 쓴다.
- 1. RFC에 프로토콜의 표준이 명시되어 있는 것과 같이 "OPEN" 구현 방식
- 2. 개발자가 직접 정의한 프로토콜을 사용하는 "CLOSED" 구현 방식

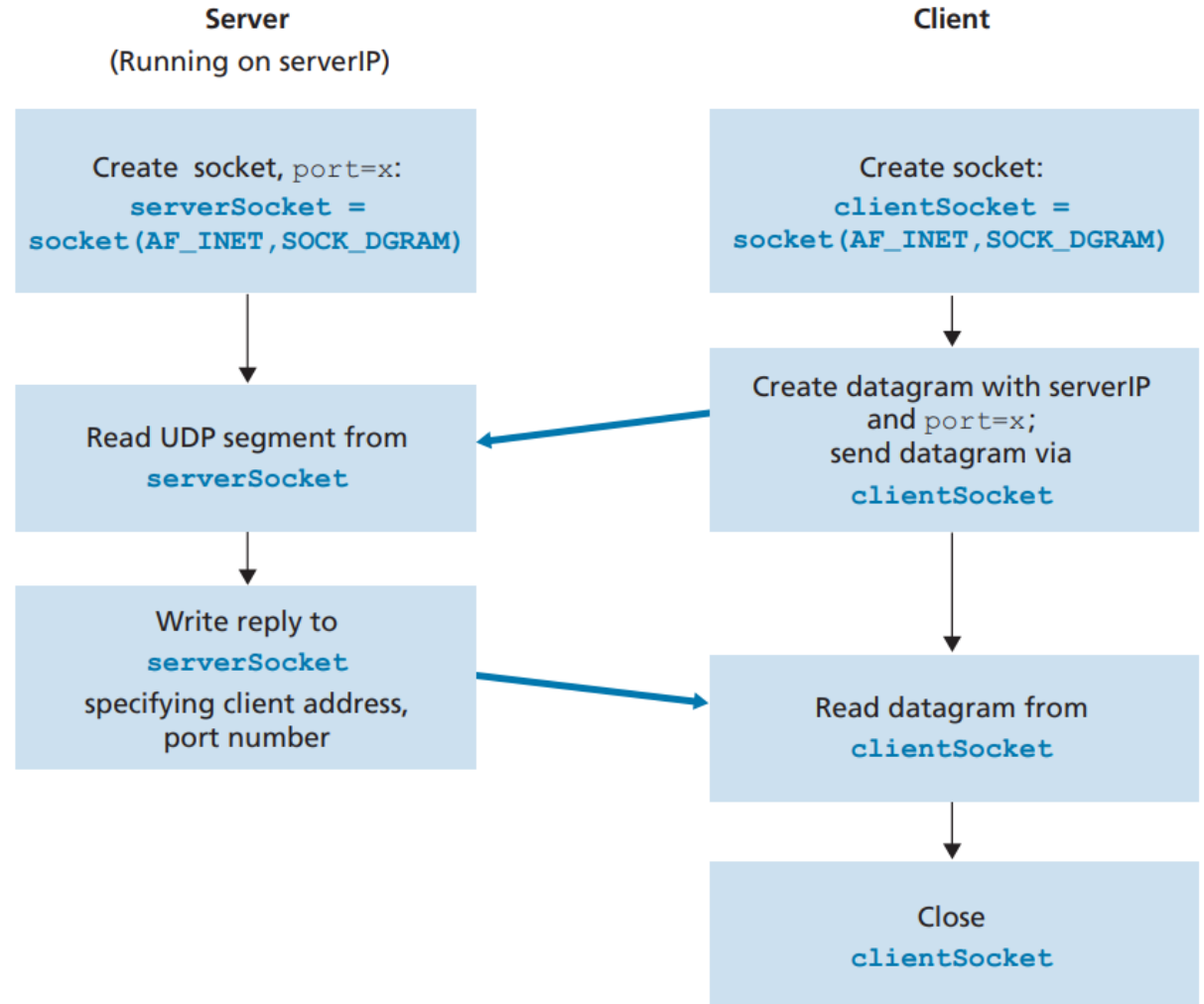
2.7 Socket Programming: Creating Network Application

- 각 프로세스는 집이고, socket은 문이다.
- 전송계층은 문밖이다.
- 개발자는 집만 수리할 수 있다.



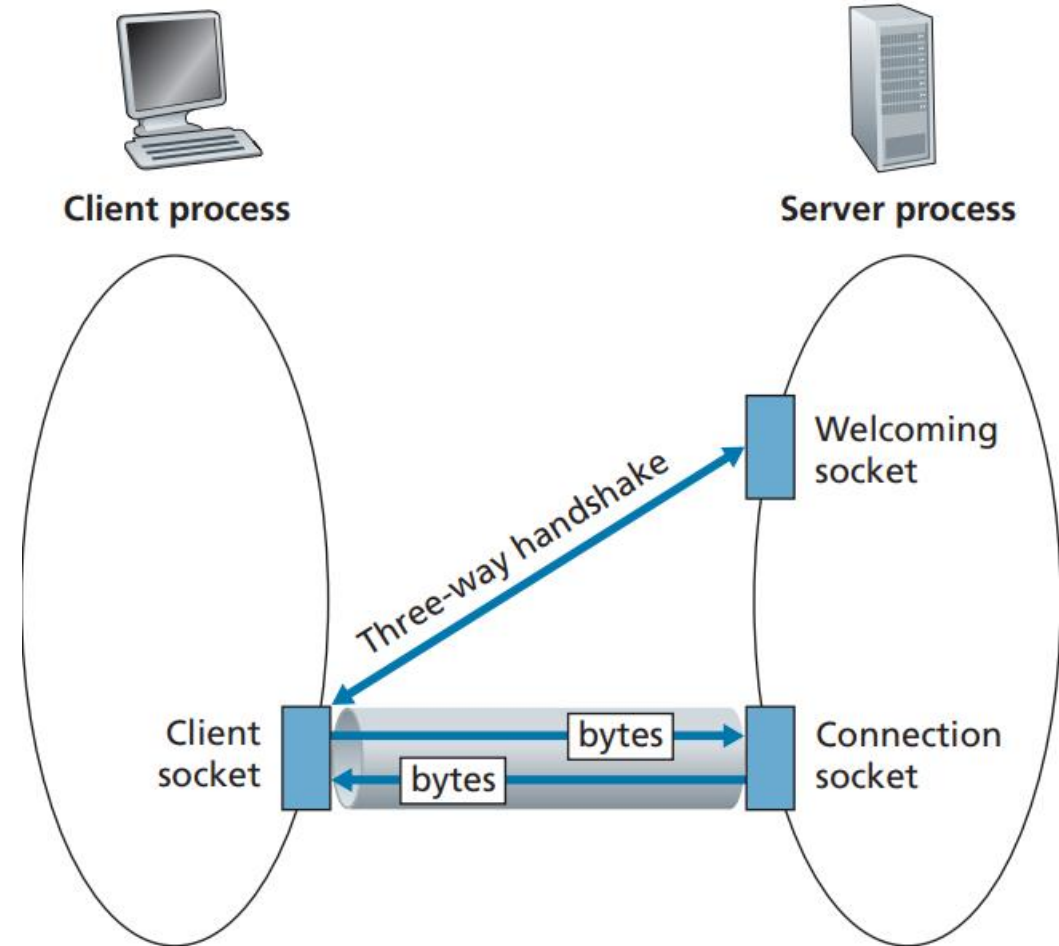
2.7 Socket Programming: Creating Network Application

- UDP의 경우 구현이 단순하다.
- Host는 여러 socket에 대해 식별해야한다
- Port number가 식별자가 된다



2.7 Socket Programming: Creating Network Application

- TCP는 연결 지향성 프로토콜이다.
- TCP connection establish
- Welcoming socket에서 Connection과정을 수행한다.



2.7 Socket Programming: Creating Network Application

- **Welcome socket(server socket)**
- **두 종류의 상태를 제어해야 한다.**
- **1. socket에 대한 연결 요청을 관리(listen)**
- **2. client socket에 대한 정보(bind)**



The reason is that TCP has two different kinds of state that you want to control, whereas UDP has only one.

23



When listening for TCP connections on a port, the networking stack needs to keep track of the port number and interface(s) you are listening on with that socket, as well as the backlogged list of incoming TCP connection-requests for that socket, and it keeps that state in an internal data structure that is associated with the socket you pass to `listen()`/`bind()`/`accept()`. That data structure will be updated for as long as the socket exists, and will be discarded when you `close()` that socket.



Once you have accepted a TCP connection, on the other hand, the new TCP socket returned by `accept()` has its own connection-specific state that needs to be tracked separately -- this state consists of the client's IP address and source port, TCP packet ID sequences, TCP window size and send rate, the incoming data that has been received for that TCP connection, outgoing data that has been sent for that TCP connection (and may need to be resent later if a packet gets dropped), etc. All of this is stored in a separate internal data structure that is associated specifically with that new socket, which will be updated until that new socket is `close()`'d, at which point it will be discarded.

Note that the lifetimes of these two types of state are very much independent of each other -- for example, you might decide that you don't want to accept any more incoming TCP connections and therefore you want to close the first (connections-accepting) socket while continuing to use the second (TCP-connection-specific) socket to communicate with the already-connected client. Or you might do the opposite, and decide that you don't want to continue the conversation with that particular client, so you close the second socket, but you do want to continue accepting more TCP connections, so you leave the first socket open. If there was only a single socket present to handle everything, closing down just one context or the other would be impossible; your only option would be `close()` the single socket and with it lose all of the state, even the parts you actually wanted to keep. That would be awkward at best.

UDP, on the other hand, has no notion of "accepting connections", so there is only one kind of state, and that is the set of buffered sent and/or received packets (regardless of their source and/or destination). As such, there is generally no need to have more than one UDP socket.

2.7 Socket Programming: Creating Network Application

```
public static void main(String[] args) throws IOException {  
    int port = DEFAULT_PORT;  
    ExecutorService service = Executors.newFixedThreadPool(DEFAULT_THREAD_NUM);  
  
    if (args.length != 0) {  
        port = Integer.parseInt(args[0]);  
    }  
  
    // TCP 환영 소켓  
    try (ServerSocket welcomeSocket = new ServerSocket(port)){  
  
        // 연결 소켓  
        Socket connection;  
        while ((connection = welcomeSocket.accept()) != null) {  
            // 스레드에 작업 전달  
            service.submit(new RequestHandler(connection));  
        }  
    }  
}
```

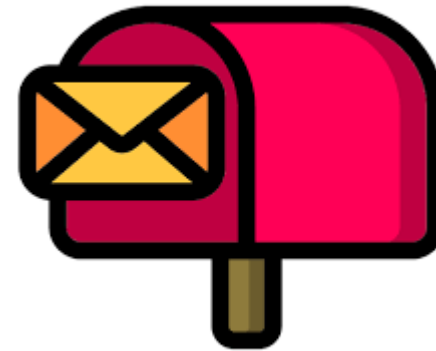
3. Transport Layer

- Transport layer는 핵심 layer이다.
- Transport layer와 network layer의 관계
- TCP, UDP ...
- 전송속도 제어로 혼잡성 제어
- TCP에서 혼잡성 제어



3.1 Introduction and Transport-Layer Service

- Logical communication
- Message: 편지 내용
- Process: 사촌
- Host: 집
- Transport-layer protocol: Ann and Bill
- Network-layer protocol: 우편 서비스



3.1 Introduction and Transport-Layer Service

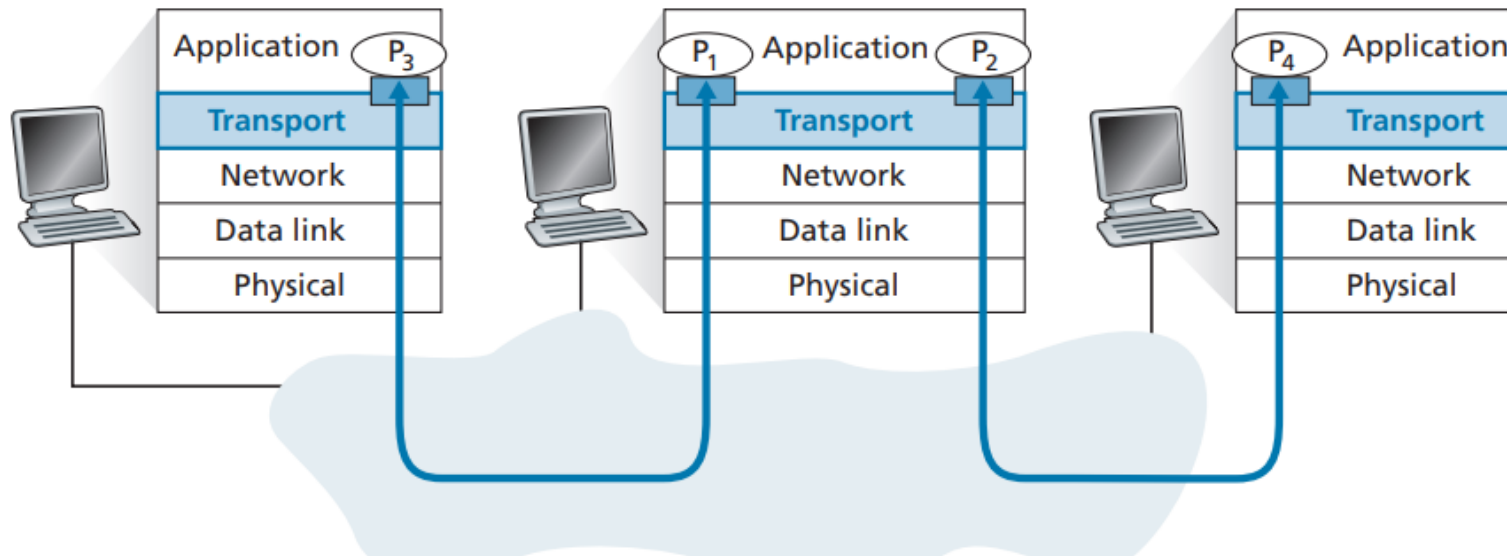
- 각 layer들은 할 일을 한다.
- 서로에게 영향을 주는 경우가 있다(지연 시간, 대역폭 보장)
- 서로 영향을 주지 않는 경우(신뢰성 통신)
- Demultiplexing(역 다중화)

3.1 Introduction and Transport-Layer Service

- Remind...
- UDP(데이터 무결성)
- TCP(데이터 무결성+신뢰성 통신+혼잡 제어)
- 3장 내내 등장할 예정

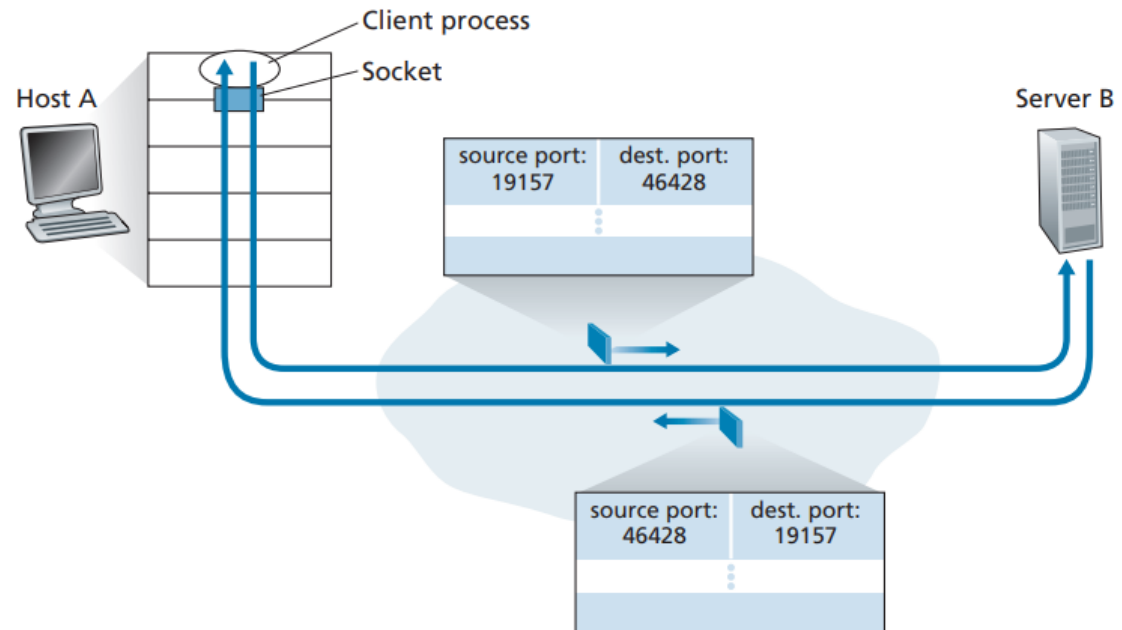
3.1 Introduction and Transport-Layer Service

- 다중화와 역다중화
- 상위 계층에서 어떻게 사용될까..



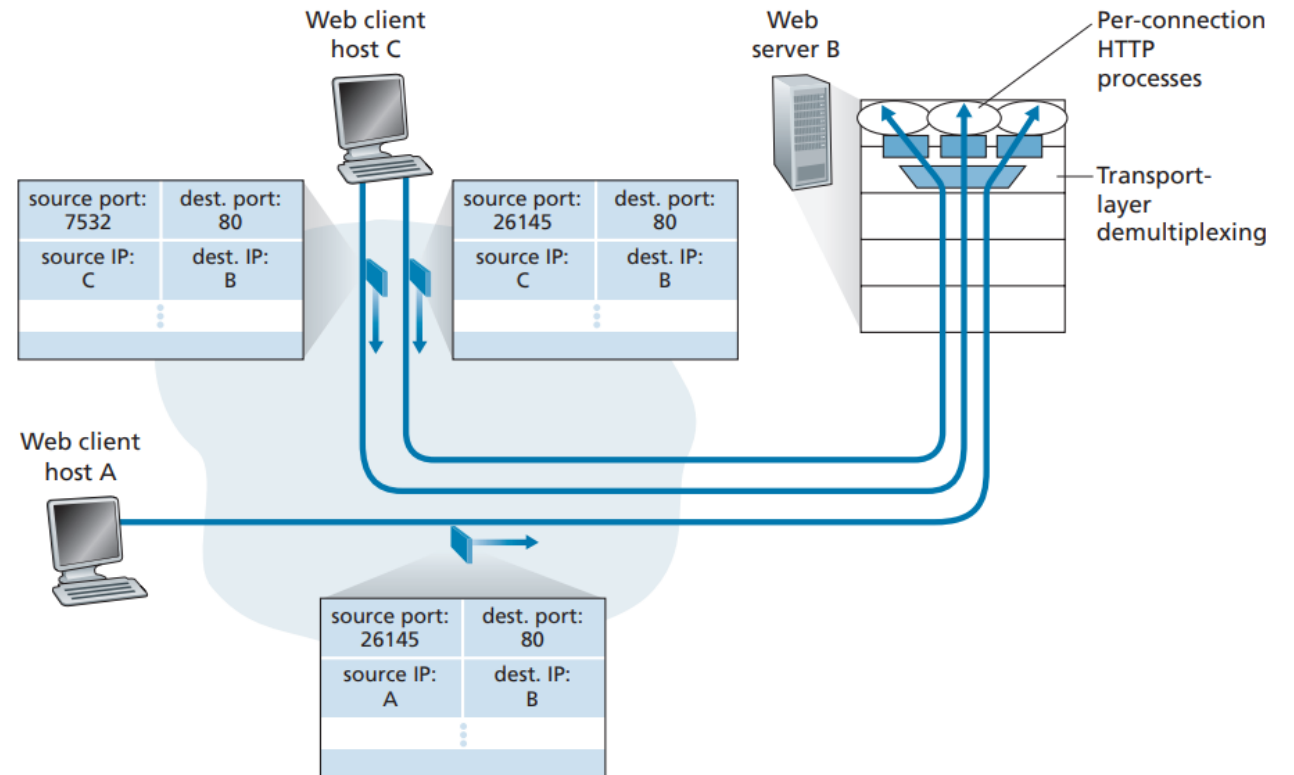
3.1 Introduction and Transport-Layer Service

- UDP에서의 역다중화
- Ip address를 남기지 않으면
- 어떻게 reply 하지?



3.1 Introduction and Transport-Layer Service

- TCP에서의 역다중화

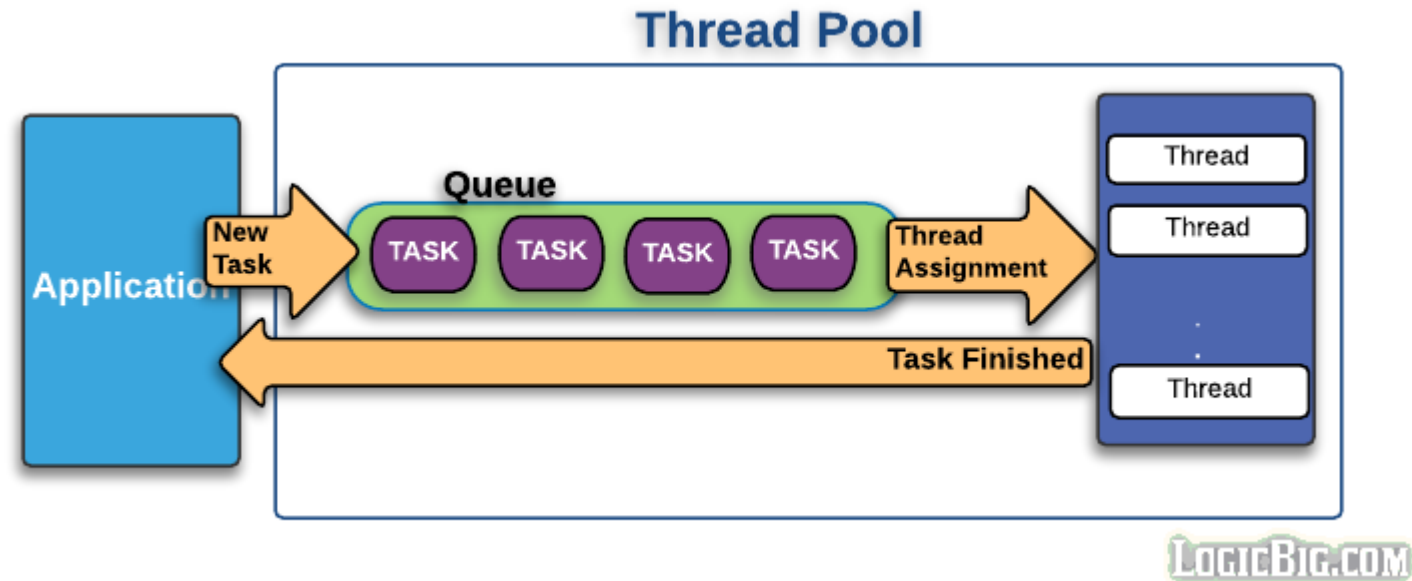


3.1 Introduction and Transport-Layer Service

- 요새의 web server는 성능 향상을 위해 하나의 process를 사용하고 socket connection을 thread와 연결한다.
- 자바 기준 server socket에서 segment header를 통해 역다중화 진행
- 적절한 server thread에 접근

3.1 Introduction and Transport-Layer Service

- Thread pool 에 thread를 관리(재사용 합니다)
- Os thread 와 1 : 1
- Socket을 매번 생성

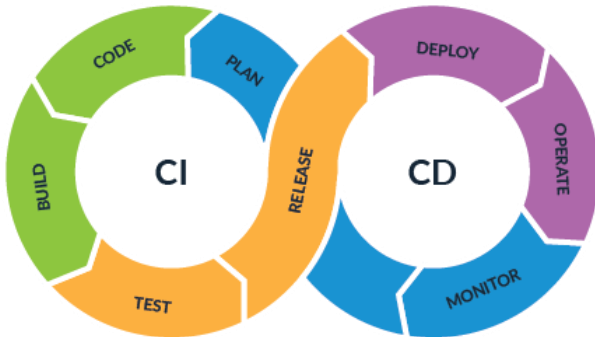


번외 CI CD **무작정** 구현하기

- 발표자는 컨테이너에 대한 개념을 잘 모른다.
- Docker, Kubernetes 모른다.
- 때론 먼저 해보는게 좋을 수 있다...

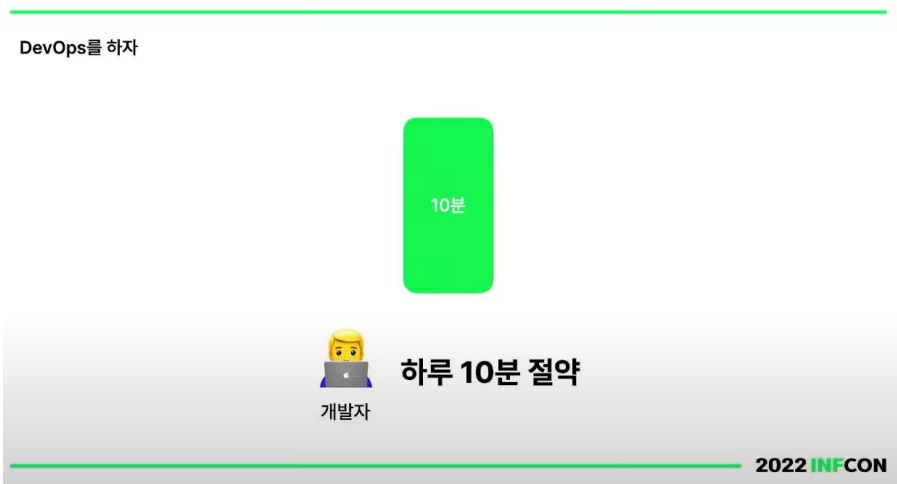
번외 CI CD 무작정 구현하기

- CI : 지속적인 통합
- CD : 지속적인 배포
- SCM(git)으로 통합을 하기에 버겁다.
- 배포하는데 생각보다 많은 리소스(시간)이 든다.



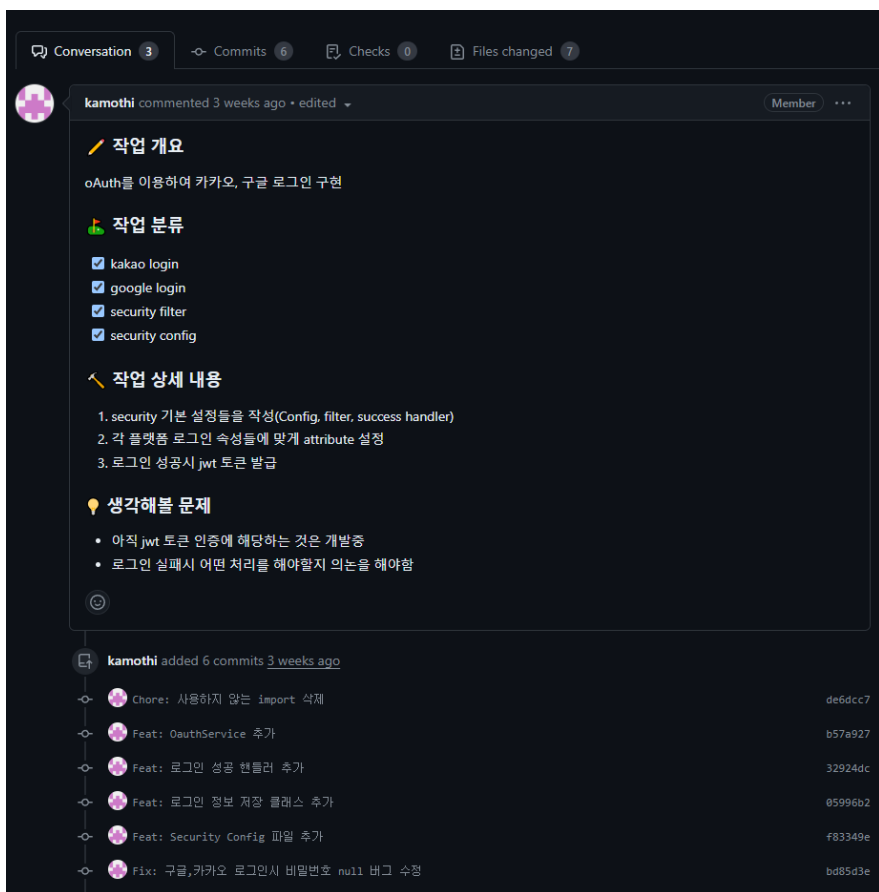
번외 CI CD 무작정 구현하기

- 자원 절약 측면에서 DevOps의 필요성

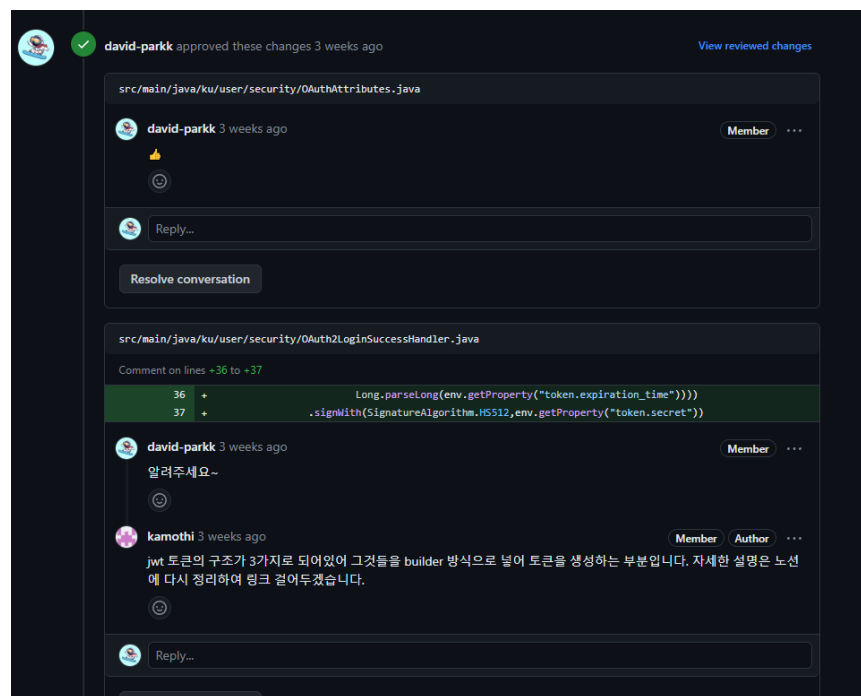


번외 CI CD 무작정 구현하기

• 깃허브 PR을 통한 협업 방법



The screenshot shows a GitHub Pull Request conversation. At the top, there are tabs for Conversation (3), Commits (6), Checks (0), and Files changed (7). The conversation is initiated by user 'kamothi' 3 weeks ago. The title of the PR is '작업 개요' (Task Overview). The description states: 'oAuth를 이용하여 카카오, 구글 로그인 구현' (Implement Kakao and Google login using OAuth). Under '작업 분류' (Task Classification), there are four checked items: 'kakao login', 'google login', 'security filter', and 'security config'. Under '작업 상세 내용' (Task Details), there are three numbered items: 1. 'security 기본 설정들을 작성(Config, filter, success handler)', 2. '각 플랫폼 로그인 속성들에 맞게 attribute 설정', and 3. '로그인 성공시 jwt 토큰 발급'. Under '생각해볼 문제' (Things to think about), there are two bullet points: '아직 jwt 토큰 인증에 해당하는 것은 개발중' (Still in development for jwt token authentication) and '로그인 실패시 어떤 처리를 해야할지 의논을 해야함' (Need to discuss what to do when login fails). At the bottom, there is a list of commits added by 'kamothi' 3 weeks ago: 'Chore: 사용하지 않는 import 삭제' (de6dcc7), 'Feat: OAuthService 추가' (b57a927), 'Feat: 로그인 성공 핸들러 추가' (32924dc), 'Feat: 로그인 정보 저장 클래스 추가' (05996b2), 'Feat: Security Config 파일 추가' (f83349e), and 'Fix: 구글, 카카오 로그인시 비밀번호 null 버그 수정' (bd85d3e).



The screenshot shows a GitHub Pull Request review. At the top, there is a green checkmark and the text 'david-parkk approved these changes 3 weeks ago'. There is a link 'View reviewed changes'. The review is for the file 'src/main/java/ku/user/security/OAuthAttributes.java'. The reviewer 'david-parkk' has left a comment: '알려주세요~' (Please let me know~). The review is for the file 'src/main/java/ku/user/security/OAuthLoginSuccessHandler.java'. The reviewer 'david-parkk' has left a comment: '알려주세요~'. The reviewer 'kamothi' has responded: 'jwt 토큰의 구조가 3가지로 되어있어 그것들을 builder 방식으로 넣어 토큰을 생성하는 부분입니다. 자세한 설명은 노션에 다시 정리하여 링크 걸어두겠습니다.' (The structure of the jwt token is 3 types, so I will use the builder pattern to generate the token. Detailed explanation is in Notion, I will link it after reorganizing it.)

발표자: 박지원

번외 CI CD 무작정 구현하기

- LGTM(looks good to me)
- 실제로 merge후 문제가 발생할 여지가 많다.
- Merge 전에 코드가 정상적으로 작동하는지 확인해야 한다.
- 적어도 build와 test code 정도 확인해보면 어떨까?

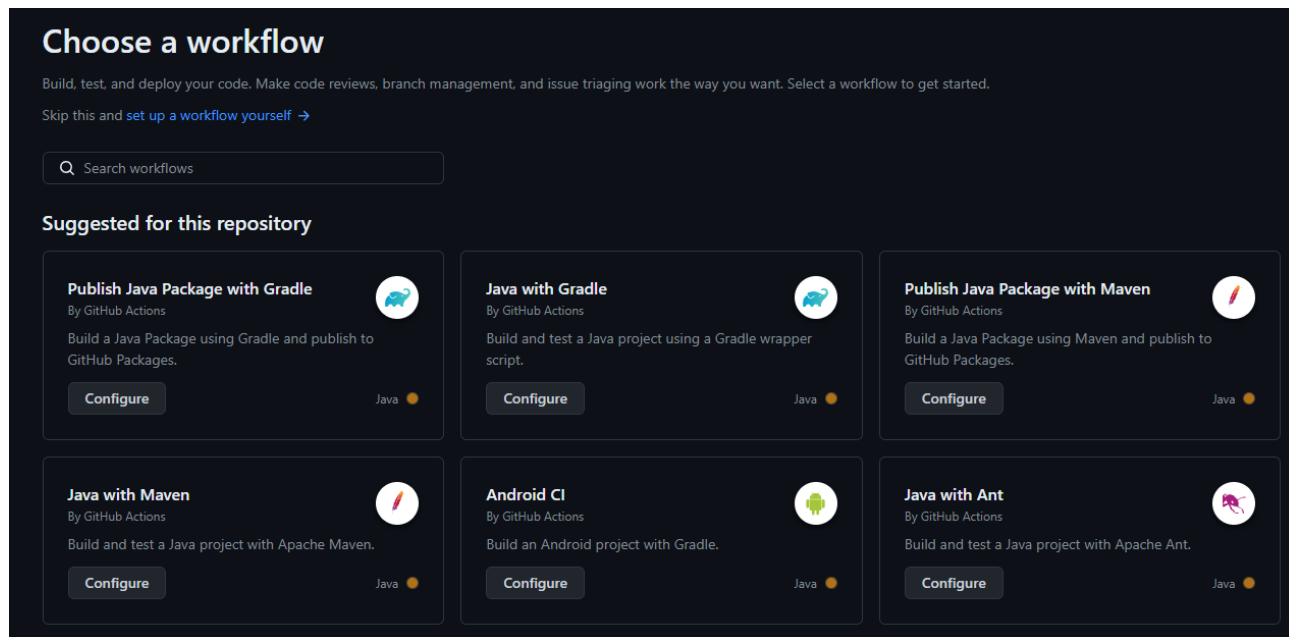
번외 CI CD 무작정 구현하기

- Github action
- Github에서 workflow를 통한 CI CD 파이프라인을 제공한다.
- 다른 tool 보다 상대적으로 빠르게 구성할 수 있다.
- 블로그글 많다...



번외 CI CD 무작정 구현하기

- `.github/workflows` 디렉토리에 `yml` 파일로 `workflow`을 추가할 수 있다.



번외 CI CD 무작정 구현하기

- Workflow에 행동을 명세하여 구현한다.
- `.github/workflows` 디렉토리에 `yml` 파일로 `workflow`을 추가할 수 있다.

```
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions 🚀
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "🎉 The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "👤 This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "🔗 The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v4
      - run: echo "💡 The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "🖨️ The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "🍏 This job's status is ${{ job.status }}."
```


번외 CI CD 무작정 구현하기

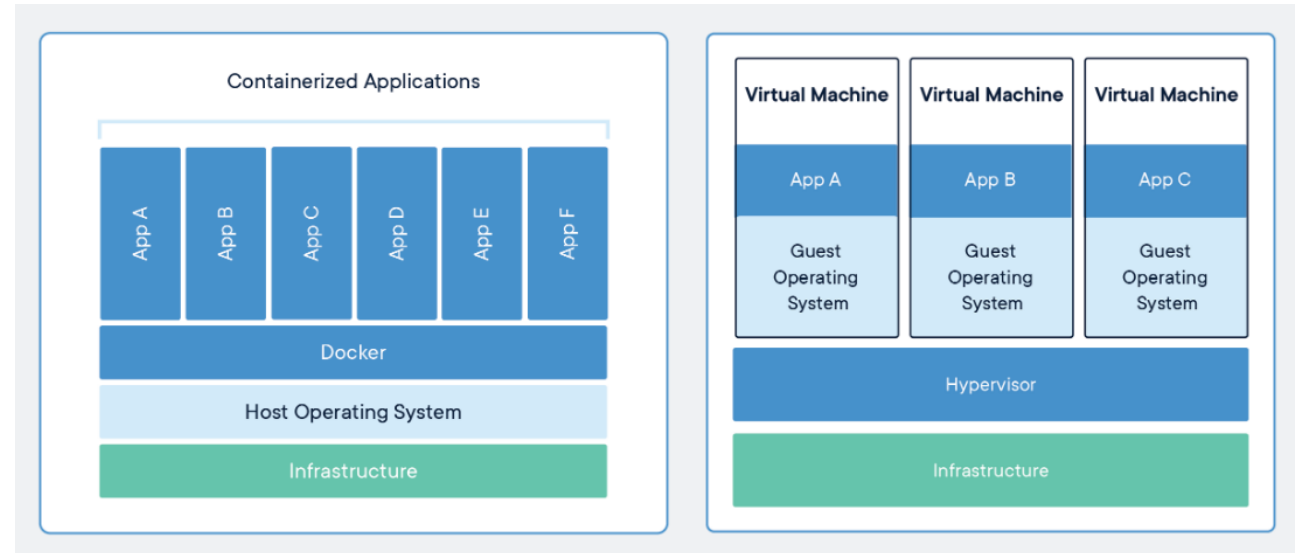
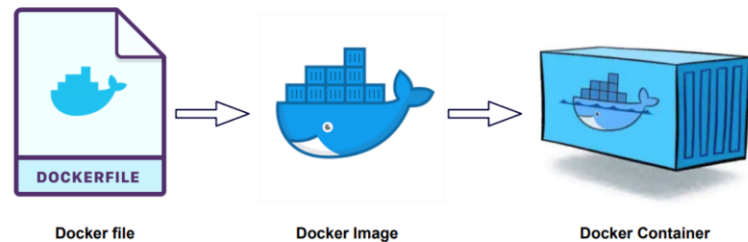
- **Event:** workflow을 실행할 event정의 (일반적으로 push or PR)
- **Workflow:** event에 대해 실행하는 일련의 flow, workflow에는 job이 있다.
- **Job:** 각각의 task, 동일 workflow에서 job은 병렬적으로 실행된다.
- **Action:** 행동(특정 명령어 형태)
- **Runner:** 실제 job을 실행하는 컨테이너

번외 CI CD 무작정 구현하기

- 1. code deploy + S3 + EC2
- 2. docker + EC2
- 2 번 방법을 채택

번외 CI CD 무작정 구현하기

- 컨테이너 기술
- 이미지를 컨테이너에 실행하여 애플리케이션에 실행



번외 CI CD 무작정 구현하기

- 이미지를 생성

```
FROM openjdk:17-jdk
```

```
# build가 되는 시점에 JAR_FILE이라는 변수 명에 build/libs/*.jar 선언
```

```
# build/libs - gradle로 빌드했을 때 jar 파일이 생성되는 경로
```

```
ARG JAR_FILE=build/libs/*.jar
```

```
# JAR_FILE을 app.jar로 복사
```

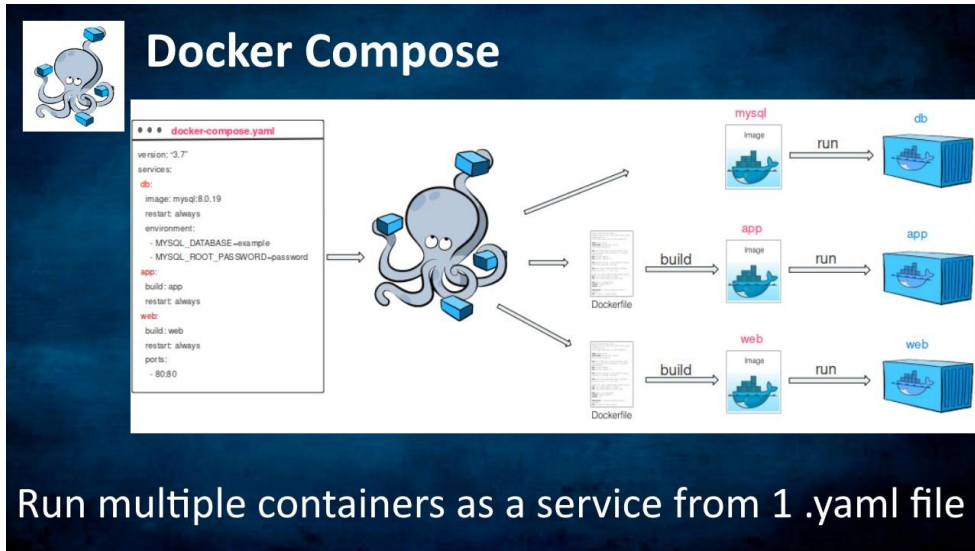
```
COPY ${JAR_FILE} app.jar
```

```
# 운영 및 개발에서 사용되는 환경 설정을 분리
```

```
ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=prod", "/app.jar"]
```

번외 CI CD 무작정 구현하기

- 도커 컴포즈로 여러 컨테이너를 한번에 실행한다.
- 여러가지 애플리케이션을 동시에 사용해야할 경우(springboot+db)



번외 CI CD 무작정 구현하기

- 각각 이미지를 실행하고 port를 설정한다

```
version: '3'
services:
  redis:
    image: redis
    ports:
      - 6379:6379
  springbootapp:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 8080:8080
    depends_on:
      - redis
```

번외 CI CD 무작정 구현하기

- 추후에 계속...

출처

- 테코톡 thread pool