System Architecture Chapter2 (Architecture Pattern)

System Architecture

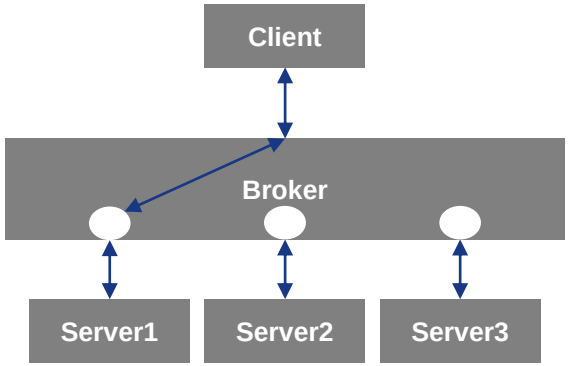
Dosung(Dave) Kim, Ph.D.

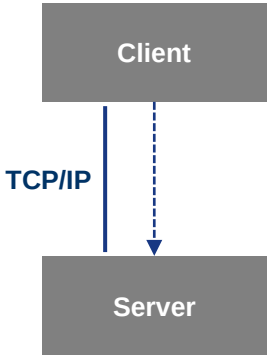
Table of Contents

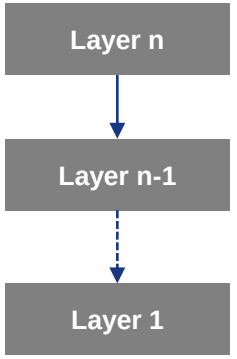
What is an architecture pattern?(1/5)

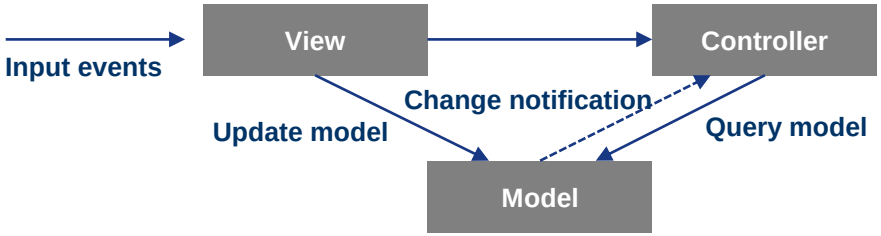
- **아키텍처 패턴(Architecture patterns)의 정의**
  - 어떤 주어진 상황속에서 소프트웨어 아키텍처에서 발생하는 일반적인 문제점들에 대해 재사용 가능 솔루션의 집합
  - 소프트웨어 시스템을 위한 검증된 구조적 도해(그림)
  - 어떻게 보면 소프트웨어 디자인패턴과 비슷하지만 더 큰 범주에 속함.

- **아키텍처 패턴의 유형 예제**









What is an architecture pattern?(2/5)

- **아키텍처 패턴 특징**
  - 수 많은 사례를 통해 검증이 된 패턴을 기반으로 하나의 집합체 정보를 제공해줌. 따라서 쉽게 정보시스템 개발을 위한 아키텍처 설계 및 구현에 적용을 할 수 있음.
  - 티어/계층기반의 패턴에서 Presentation/Business logic/Data등의 역할분담, MVC패턴기반의 Model-View-Controller의 기능역할분담을 통해 시스템 개발 및 운영관리의 효율성을 극대화 시킬 수 있음.
  - 기능 및 서비스별로 균형있게 구성되어 있는 아키텍처 패턴은 독립적으로 시스템 테스트 및 사용을 위한 긍정적 효과를 줄 수 있음.

구분	설명
패턴기반(Pattern basis)	서브 시스템과 그들의 역할을 사전정의해 놓은 하나의 집합임
분배(Distribution)	엄격한 규칙을 가지고 Entity들 사이에서 균형 있는 책임을 분배함 (역할 분담을 명확하게 하기 위해!)
테스트능력(Testability)	각각의 요소를 독립적으로 테스트할 수 있어야 함.
사용의 편리함(Easy of use)	사용자의 편리함은 코드의 길이에 비례함. 아키텍처 패턴 구현에 상당히 많은 유지보수 비용이 들어 감.

What is an architecture pattern?(3/5)

- **아키텍처 패턴과 디자인 패턴의 차이점**

구분	설명
아키텍처 패턴	프로그램내에서 큰 구조로 구성되어 다른 구성요소들을 관리하는 역할
디자인 패턴	어떤 특정유형의 문제를 해결하는 방법으로써 아키텍처 패턴보다는 조금 더 좁은 개념

- **패턴(Pattern)**
  - 서브 시스템과 그들의 역할을 사전에 정의해 놓은 하나의 Set(집합)
  - 만약에 하나의 예로Client ←→ Server 패턴의 시스템 구조에 의하면 두개의 서브 시스템으로 식별될 수 있음.
    - Client(여러 인스턴스가 존재할 수 있음)와 Server(하나의 존재)를 뜻함.
  - 시스템들 사이의 관계를 정리하기 위한 규칙과 지침을 설명함.
  - 경험이 많은 사용자(사람)에 의해 기록되고 형성되어 짐. 즉 검증이 된 사례가 많이 있어야 함.
  - 한 사용자(사람)에 의해 구성되어지지 않고, 많은 개발자들 경험의 산물임.

What is an architecture pattern?(4/5)

- **패턴사용의 효율성**
  - 자주 발생하는 문제를 해결하기 위한 일반화된 방식임.
  - 반복적 문제 발생 → 경험 많은 전문가들이 문제해결 및 패턴을 정의 → 문서화 → 검증된 방법으로 문제해결
  - 오랫동안 실용적 측면에서 사용해오고 있으므로 안정적인 분석의 틀을 제공함.
  - 다양한 종류의 시스템 개발 시 변경으로 인한 리스크를 최소화 할 수 있음.

구분	설명
기록된 패턴을 이용하여 해결책을 위한 재사용이 용이함	▪ 패턴은 일반적인 어휘와 설계 솔루션에 대한 이해를 제공함. ▪ 각 패턴의 이름은 이미 널리 사용되어지는 디자인(설계)의 언어가 됨. ▪ 패턴은 문제에 대한 쓸데없는 설명의 필요성을 줄여 줌. ▪ 정의된 패턴속성은 소프트웨어의 구현을 지원할 수 있음. ➢ 예) 서버←→클라이언트 패턴에서 서버는 클라이언트에게 통신요청을 하면 안됨.
많은 패턴은 명시적으로 소프트웨어 시스템에 대한 비기능적 요건을 다룸	▪ 사용자의 인터페이스 가변성을 지원 함. ➢ 예) MVC(Model-View-Controller) 패턴 ▪ 패턴으로 인해 복잡한 시스템은 질서정연한 블럭단위로 보임.

What is an architecture pattern?(5/5)

- **자주 사용하는 아키텍처 패턴**
  - 시스템 개발을 위해 자주 적용되는 아키텍처 패턴은 아래 표와 같음.
  - 계층화 패턴은 각 계층(Layer 또는 Tier)를 기능별로 나누어 개발 및 운영의 효율성을 모두 고려한 패턴이라고 할 수 있음.
  - 브로커 패턴은 Client 와 Server사이에서 매개역할을 하는 패턴구조. 시스템을 구축 시 프록시서버가 이에 해당 될 수 있음.
  - State-Logic-Display패턴은 계층화패턴과 매우 긴밀하게 연결될 수 있음. 즉 3계층 구조의 아키텍처 패턴임.
  - MVC패턴은 코드관리 및 개발의 효율성을 높이기 위해 3개영역의 Model-View-Controller기반으로 기능별로 구분 및 연결되는 패턴임.

구분	설명
계층화패턴 (Layered pattern)	▪ 가장 일반적으로 사용하는 아키텍처 패턴으로서 subtask들을 그룹으로 묶어 사용허가 관계를 표시하는 패턴
브로커패턴 (Broker pattern)	▪ 외부에서 분산된 컴포넌트를 호출하려고 할 때 Client 요청을 분석하여 Server측에 전달하고 그 결과값을 전달하는 역할 ➢ Proxy(Broker) → 보안 및 안정성을 높일 수 있는 패턴
State-Logic-Display 패턴	▪ 비즈니스 Application을 개발할 때 가장 일반적으로 사용되는 패턴, UI, Business logic and data로 구분함. 따라서 변경이 용이함. ➢ 게임, 웹 어플리케이션 등 많은 분야에 사용되고 있음.
Sense-Compute-Control 패턴	▪ 일정시간 별로 값을 읽어오는 Sense와 센서의 값을 계산하여 필요한 행위를 정의하는 Compute, 기능이나 행위로 이어지게끔 하는 Control로 모듈을 구분하는 패턴 ➢ Embeded software
MVC 패턴	▪ 모델, 뷰 및 컨트롤 3개의 컴포넌트로 Application을 구분함. ▪ Model: 기능 및 데이터 핸들링, View: 사용자의 화면표시(UI), Controller: 모델과 뷰의 관계를 기반으로 이벤트가 발생하면 모델과 뷰에 전달함.

Architecture pattern 종류(1/37)

- **관점을 기반으로 한 패턴특징**
  - 모든 아키텍처 패턴들은 시스템 개발을 위해 최적의 구성방법을 제시해줌.
  - 따라서 시스템 아키텍처를 어떠한 특징을 가지고 설계해야 하는지 명확하게 이해해야 함.

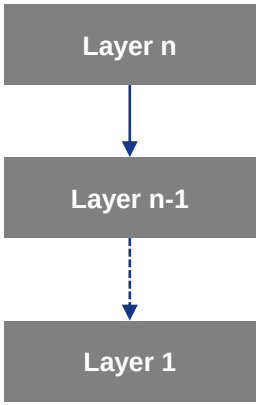
관점	아키텍처 패턴
데이터 공유	클라이언트-서버 패턴
	피어 투 피어 패턴
데이터 동기화 및 변환	파이프-필터 패턴
데이터 동기화 및 백업, 병렬처리	마스터-슬레이브 패턴
계층별 기능분리	계층화 패턴
분업(Model-View-Controller)	MVC패턴
네트워크(라우팅)	이벤트-버스 패턴(적절한 채널로 라우팅)
Shared 된 데이터 중심패턴	블랙보드 패턴
분산(컴퓨팅)	브로커 패턴
언어해석 (DB 질의어 및 통신프로토콜 정의)	인터프리터 패턴

Architecture pattern 종류(2/37)

- **계층화 패턴(Layered pattern or Tier pattern)**
  - n-패턴 또는 티어패턴 이라고도 함.
  - 하위 모듈(subtask)들의 그룹으로 나눌 수 있는 구조화된 시스템(프로그램)에서 사용할 수 있음.
  - 각 하위 모듈은 특정한 수준의 추상화를 제공함.
  - 일반적인 공통 3계층 또는 4계층으로 구성됨.

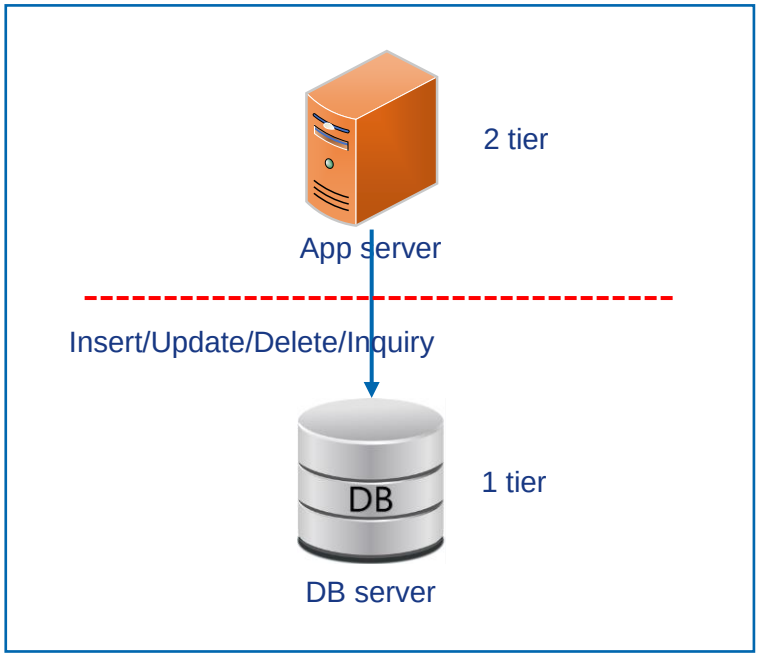
구분	설명
Presentation layer	사용자의 화면 즉 UI(User Interface)
Application layer	서비스 계층이라고 함.
Business logic layer	Backend 즉 도메인계층이라고 함.
Data access layer	영속계층이라고 함. 데이터가 저장되고 관리되는 계층

  - 적용 예)
    - General desktop application
    - E-commerce web application
    - OSI 7계층 구조



Architecture pattern 종류(3/37)

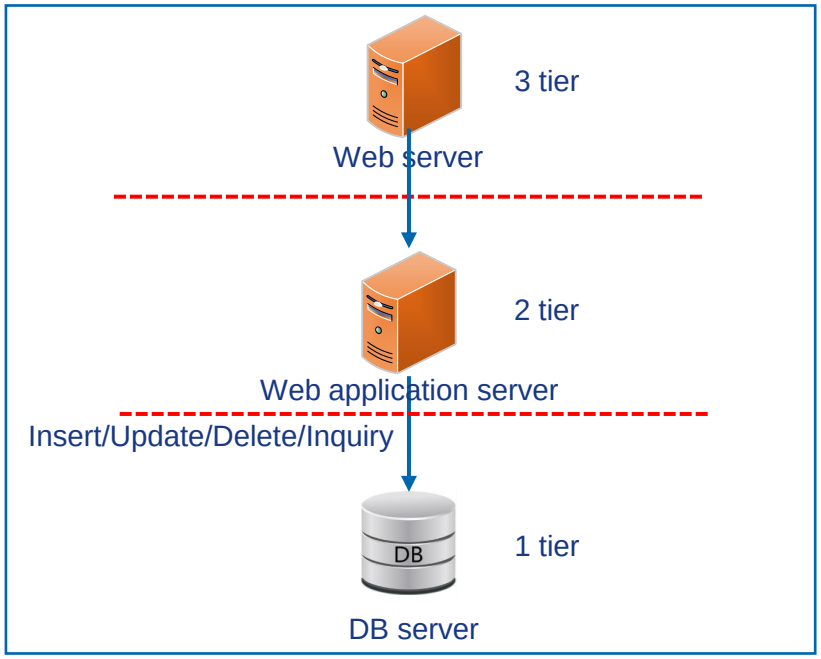
- **계층화 패턴(Layered pattern or Tier pattern)**
  - 예시
    - 일반적으로 Application개발을 위해 많이 적용되는 패턴으로 Native앱 및 웹환경에서 2티어 / 3티어 으로 개발을 많이 함.
    - 2티어 계층
      - AWS EC2(VM)인스턴스 웹서버 또는 Application서버가 DB에 연결된 형태를 말함.
      - 트래픽이 많지 않은 환경에서 자주 사용하는 패턴구조임.
      - 소규모 부서 또는 조직에서 적합함. 단일서버 및 배치업무분야에서 많이 사용함.



특징	설명
장점	- 프로그램개발의 기간 및 비용을 단출할 수 있음. - 데이터 프로세싱을 필요한 Native app에 적합함.
단점	- 트래픽이 늘어나면 성능이 급격하게 저하됨. - 확장 및 유연성이 떨어짐. - 보안 / 부하관리 / 장애관리 등이 힘듬.

Architecture pattern 종류(4/37)

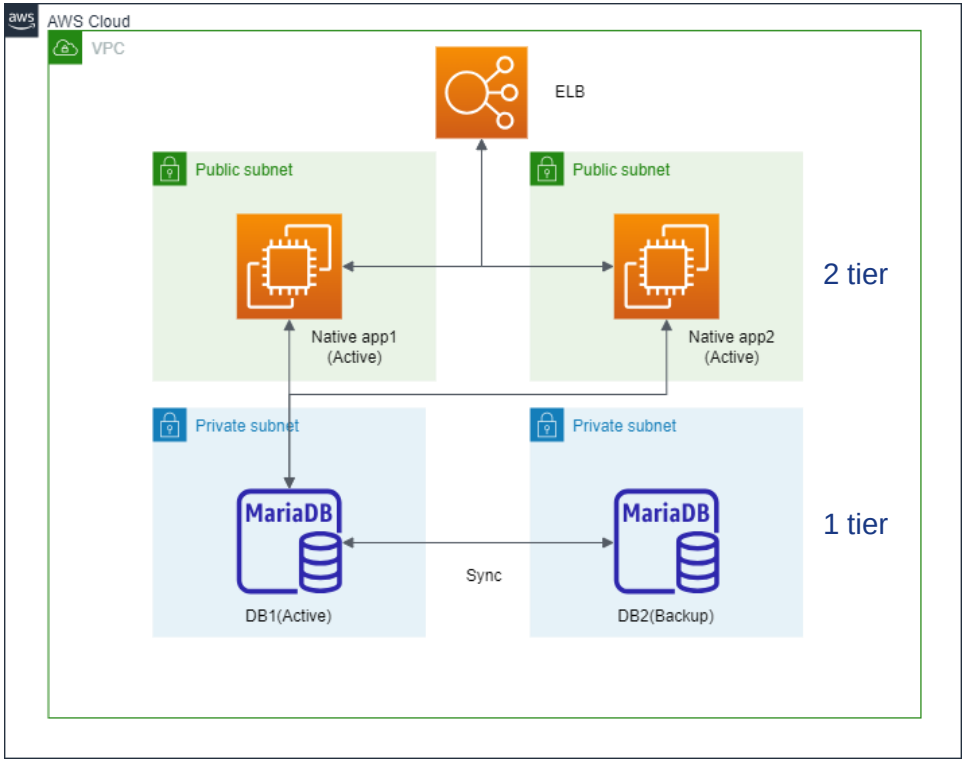
- **계층화 패턴(Layered pattern or Tier pattern)**
  - 예시
    - 일반적으로 Application개발을 위해 많이 적용되는 패턴으로 Native앱 및 웹환경에서 2티어 / 3티어 으로 개발을 많이 함.
    - 3티어 계층
      - 2티어 구조와 다르게 웹서버와 DB서버사이에 WAS(Web Application Server)가 구축되는 구조임.
      - 벡엔드에
      - 큰규모 및 다중서버시스템 환경에 적합함.
      - 대용량의 트래픽 발생 시스템에 적합함.



특징	설명
장점	- 이기종 시스템 및 DB연동이 가능 - 확장이 용이함. - 정보보안, 부하 및 장애관리에 유용함.
단점	- 시스템 개발에 대한 비용과 시간이 많이 소요됨. - 간단한 구조의 시스템 개발시에도 3티어 기능시스템 분리가 되어야 함.

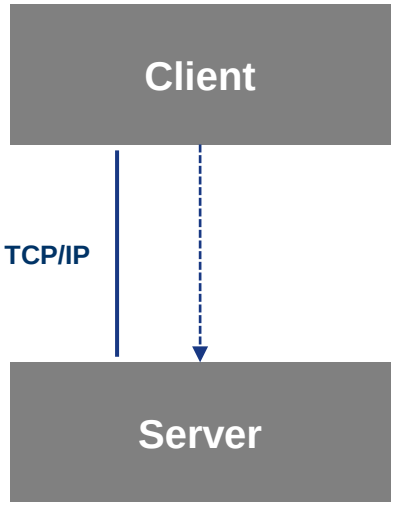
Architecture pattern 종류(5/37)

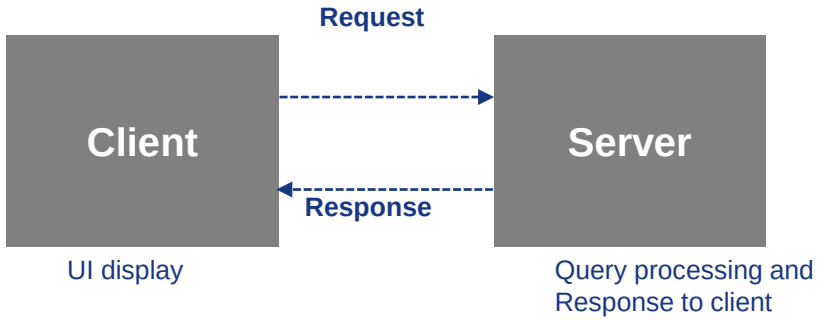
- **계층화 패턴(Layered pattern or Tier pattern)**
  - 예시
    - AWS 클라우드 기반으로 아래와 같이 3티어 뿐만이 아닌 2티어 구조의 시스템 개발을 할 수 있음. Native Android / IOS App개발도 보다 쉽게 할 수 있음.
    - 아래 그림을 보면 2티어 Public subnet영역에는 App서버가 존재, 1티어 Private subnet에는 DB서버가 존재하고 있음.



Architecture pattern 종류(6/37)

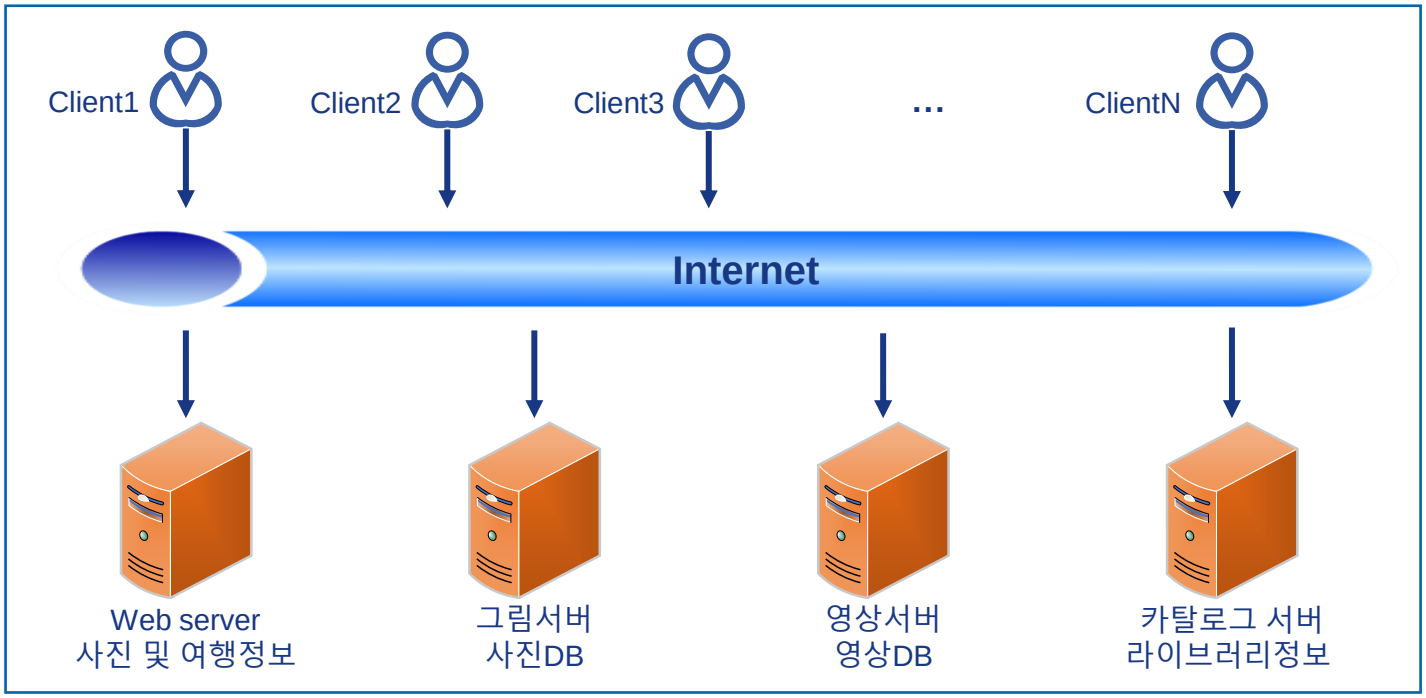
- **클라이언트-서버 패턴(Client-Server pattern)**
  - 다수의 클라이언트와 하나의 서버로 구분되어 구성 되어 짐.
  - 서버 → 클라이언트, 다수의 클라이언트들에게 서비스를 제공
  - 클라이언트 → 서버, 클라이언트들은 서비스를 제공받기 위해 서버에 요청
  - 클라이언트들끼리 서로 직접 통신하지 않음.
  - 독립적인 서버들에 의해 서비스가 제공되는 집합으로 표현할 수 있음.
  - 적용 예)
    - Email application
    - File sharing application
    - Bank application





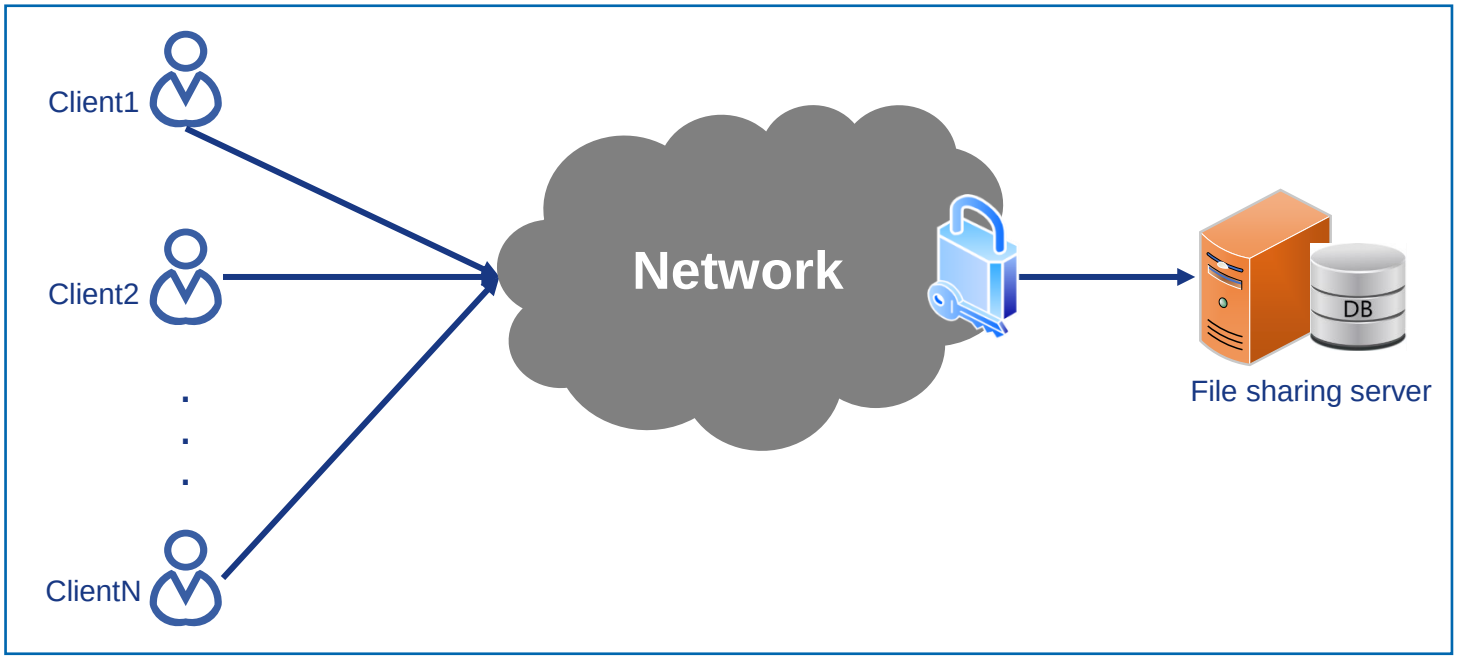
Architecture pattern 종류(7/37)

- **클라이언트-서버 패턴(Client-Server pattern)**
  - 예시
    - 모든 시스템에 기본으로 적용할 수 있는 패턴이라고 할 수 있음. 즉 여러 클라이언트들이 서버에 작업을 위한 요청을 하는 구조 임.
    - 아래 그림을 보면 전형적인 Client – Server 패턴 기반의 여행라이브러리 시스템임. 네트워크를 중심으로 사용자인 Client와 서비스 제공자인 Server로 나뉘어 있는 것을 볼 수 있음.
    - 클라이언트들은 수 없이 더 증가할 수 있음. 따라서 서버의 과부하 및 장애처리를 위해 네트워크 수준에서 로드벨런서 및 이중화 서비스가 필요함.



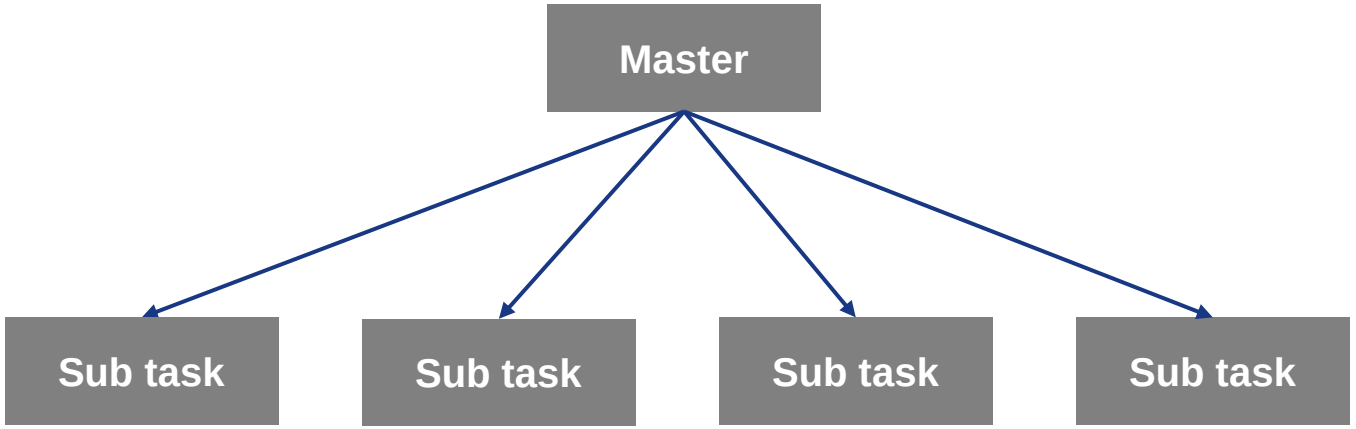
Architecture pattern 종류(8/37)

- **클라이언트-서버 패턴(Client-Server pattern)**
  - 예시
    - 또 하나의 예시를 보면 회사에서 흔히 사용하는 파일공유시스템을 볼 수 있음.
    - 단순하게 사용자들은 회사 사내망을 통해 파일이 저장되어 있는 서버에 접속하는 구조임.
    - 각 클라이언트들은 접속권한을 부여받아 파일공유서버에 저장되어 있는 폴더 및 파일에 접속할 수 있음.
    - 일반적으로 파일공유서버는 보안을 위해 사설망(내부망)에 구축됨.



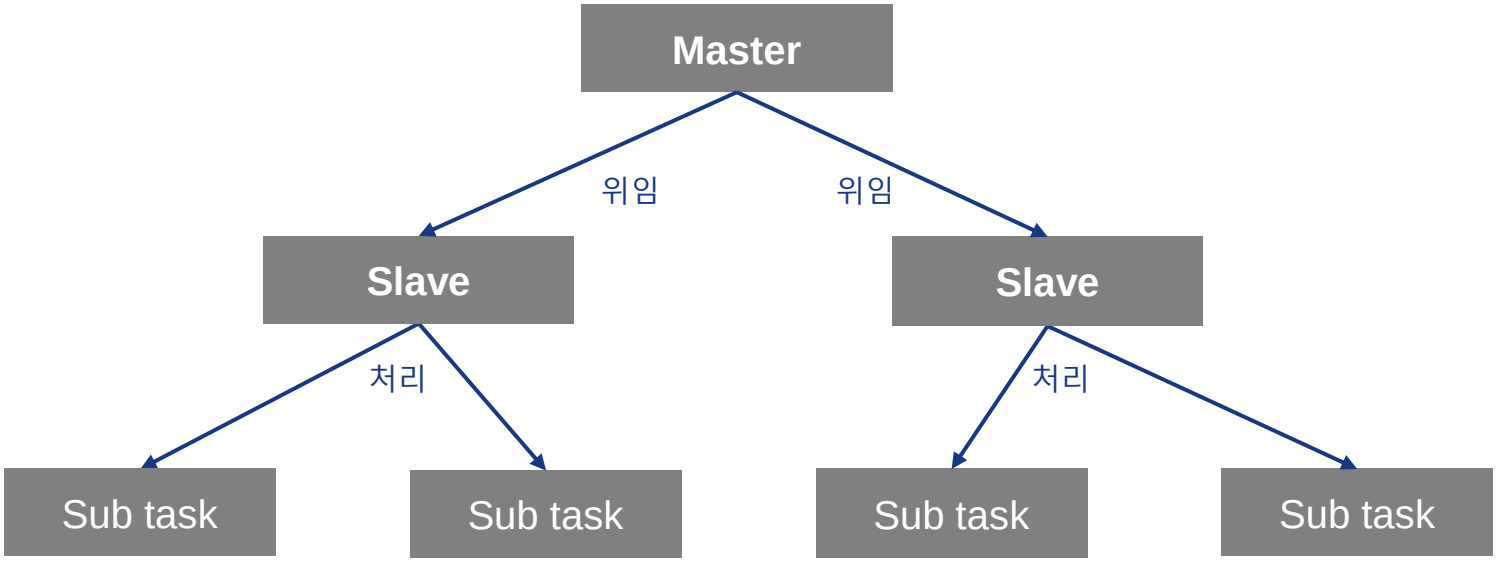
Architecture pattern 종류(9/37)

- **마스터-슬레이브 패턴(Master-Slave pattern)**
  - 크게 마스터와 슬레이브 두 부분으로 나눌 수 있음.
  - 마스터 컴포넌트는 동등한 구조를 지닌 슬레이브 컴포넌트들로 분선처리 함.
  - 마스터 컴포넌트는 슬레이브가 반환한 결과값을 이용하여 최종 결과값을 계산함.
  - 적용 예)
    - Database backup or Synchronization
    - 병렬컴퓨팅
    - 컴퓨터 시스템에서 버스와 연결된 주변장치(드라이버)
  - 아래 그림에서 보면 마스터 레벨에서 여러 개(4개)의 동일한 Task들을 분할함.
  - 슬레이브 없이 마스터가 직접 Task들을 컨트롤할 때



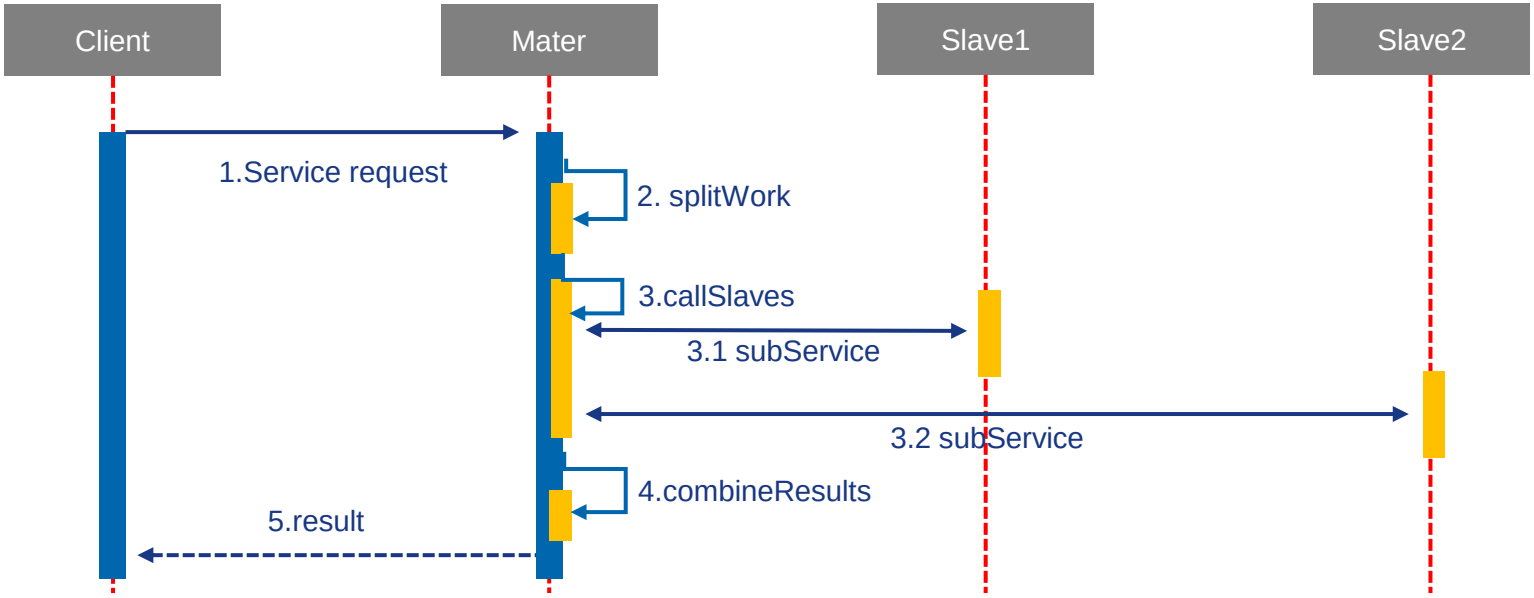
Architecture pattern 종류(10/37)

- **마스터-슬레이브 패턴(Master-Slave pattern)**
  - 아래 그림과 같이 마스터에서 분할 된 동등한 Task들은 슬레이브들에게 위임되고 각 슬레이브는 처리되는 반환하는 결과로부터 최종 결과를 산출하여 마스터에게 보내줌.
  - 결국 슬레이브들은 분할된 Task들을 최종적으로 처리하여 반드시 마스터에게 반환시켜 줘야 함.



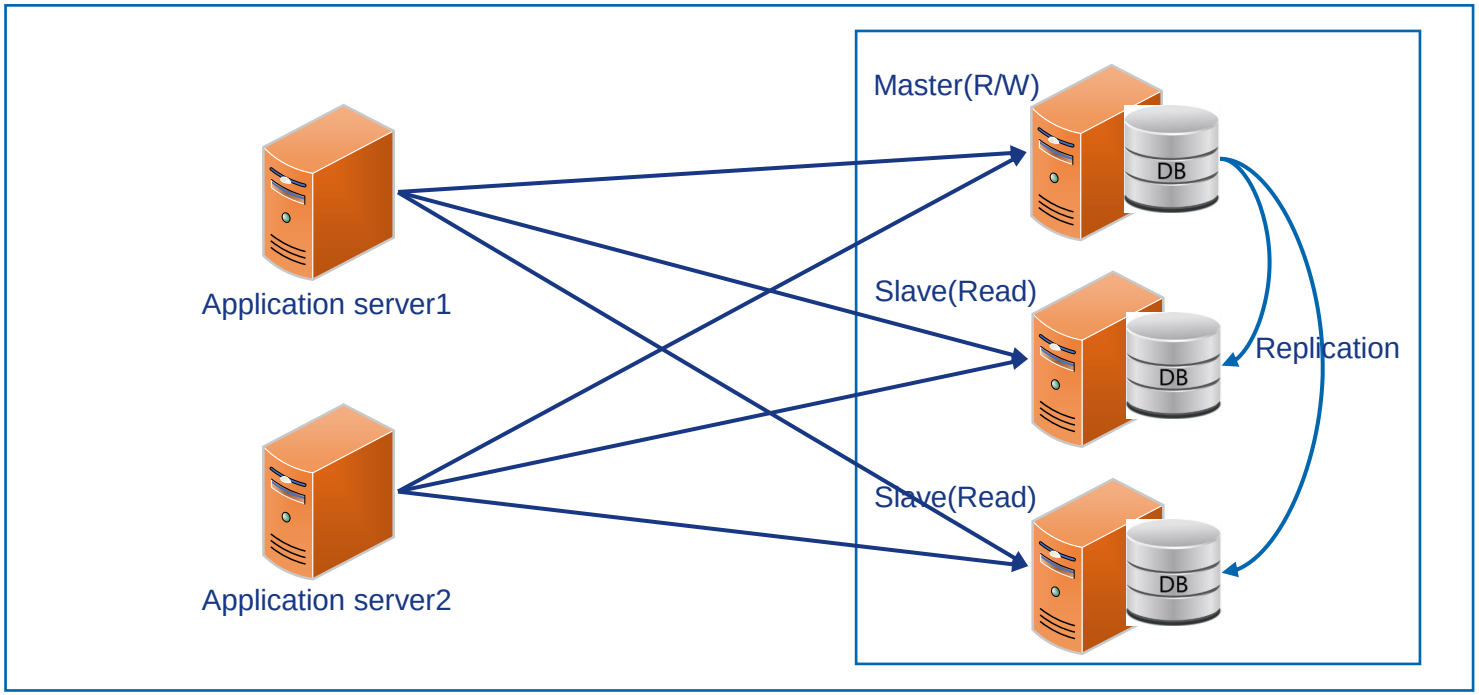
Architecture pattern 종류(11/37)

- **마스터-슬레이브 패턴(Master-Slave pattern)**
  - 마스터-슬레이브 동작설명 예시
    1. 클라이언트가 마스터에게 서비스 요청
    2. 마스터는 동등한 Sub task들을 N개의 task들로 분할됨.
    3. 마스터는 Sub task들을 슬레이브에 위임하고 실행함.
    4. 3.1-3.2 슬레이브들은 각 Sub task의 처리를 진행하고 그 결과를 마스터에게 반환함.
    5. 4. 마스터는 슬에이브들로 부터 받은 분할된 결과들을 취합하여 최종결과 산출을 진행함.
    6. 5. 마스터는 클라이언트에세 최종 결과를 반환함.



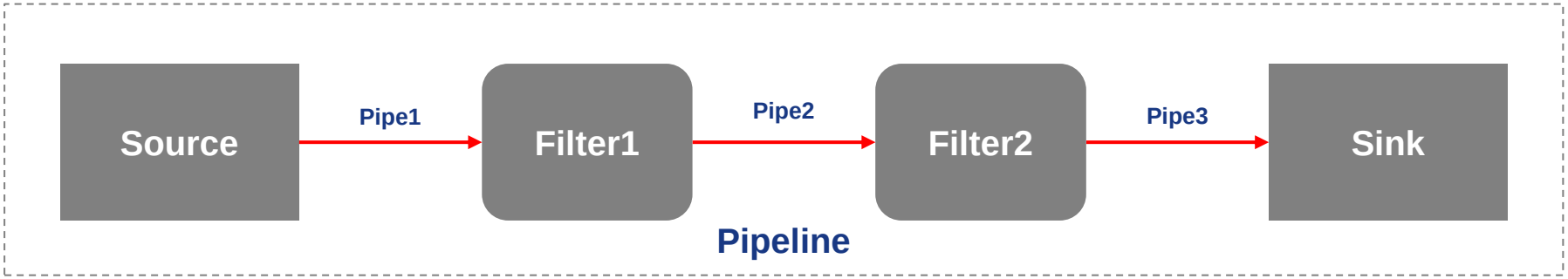
Architecture pattern 종류(12/37)

- **마스터-슬레이브 패턴(Master-Slave pattern)**
  - 예시
    - 마스터-슬레이브 패턴의 예시로 데이터 백업을 예로 들 수 있음.
    - 아래 그림과 같이 벡엔드 단의 시스템 중 데이터 저장소를 담당하는 DB서버는 DR(Disaster Recovery) 측면에서 장애대응에 많이 활용을 함.
    - MySQL/MariaDB의 구조에서 데이터 백업을 위해 마스터DB는 슬레이브DB들에게 역할을 위임함. 특히 부하분산을 위해 Read기능을 슬레이브DB에게 할당 함.
    - 실시간 데이터 백업을 위해 Replication기능을 활용함.



Architecture pattern 종류(13/37)

- **파이프-필터 패턴(Pipe-Filter pattern)**
  - 데이터 스트림을 생성하고 처리하는 시스템에서 활용될 수 있음.
  - 각 처리되는 과정은 필터(Filter)라는 컴포넌트에서 진행됨 이때 변환이 이루어짐.
  - 처리되는 데이터들은 파이프라인을 형성하여 처리됨.
  - 파이프라인상의 데이터는 각 변환작업을 통해 목적에 따라 개별 처리되거나 한번에 일괄처리가 될 수 있음.
  - 파이프라인은 버퍼링 또는 동기화 목적으로 사용될 수 있음.
  - 적용 예)
    - 컴파일러, 어휘분석, 파싱, 데이터 코드생성, 문맥분석등
    - 생물정보학에서의 Workflow
    - 일괄처리(배치) 시스템과 임베디드 시스템에 적합함.

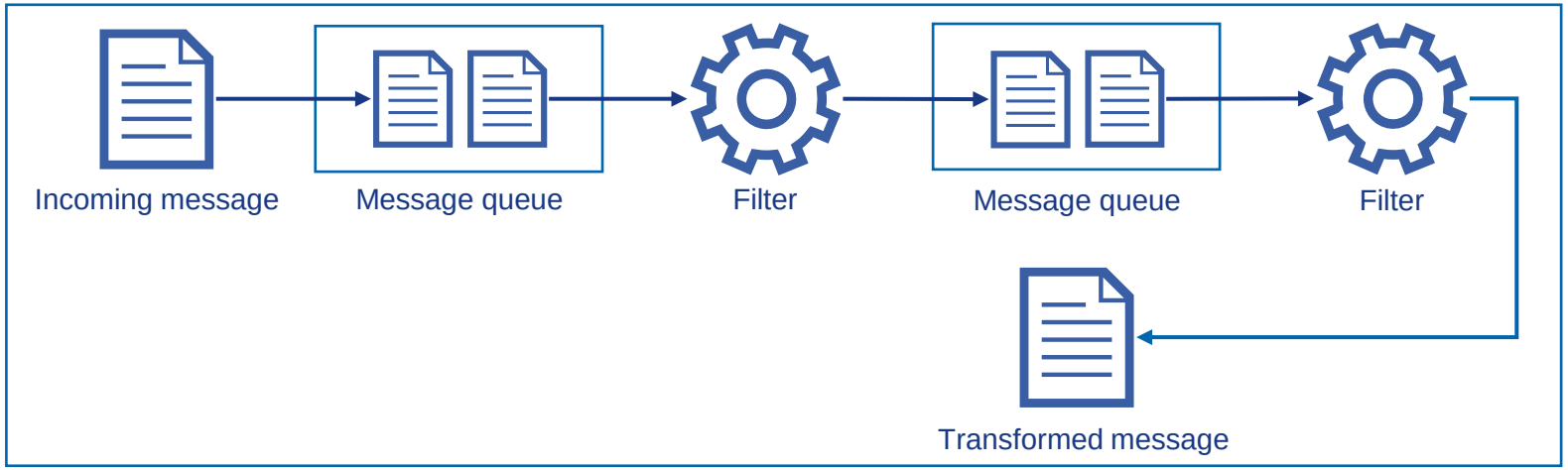


Architecture pattern 종류(14/37)

- **파이프-필터 패턴(Pipe-Filter pattern)**
  - 구현 및 적용 고려사항
    - 신뢰성
      - 파이프라인상 필터가 설치되는데 이때 처리되는 데이터가 손실되지 않는 인프라를 사용해야 함.
    - 복잡성
      - 여러 처리를 파이프라인상의 필터를 통해 처리하므로 유연성이 좋음. 따라서 여러 서버들에 걸쳐서 처리된다면 복잡성을 야기 시킬 수 있음.
    - 반복된 메시지
      - 파이프라인상에서 필터가 다음단계에 메시지를 게시한 후 실패하면 필터의 다른 인스턴스가 실행될 수 있음. 즉 동일한 메시지의 1개이상의 인스턴스가 다음 필터로 전달될 수 있다라는 의미임.
      - 파이프라인에서는 항상 중복메시지를 검색하고 제거해야 함.
    - 컨텍스트 및 상태
      - 파이프라인상의 각 필터는 요청받은 작업을 충분히 수행할 수 있는 컨텍스트를 가져야 함. 따라서 해당 컨텍스트는 많은 양의 상태정보를 가질 수 있음.
    - Application에서 수행되는 처리단계
      - 각 단계가 확장성을 가진 요구사항이고 서로 다른 경우 유용하게 적용할 수 있음.
      - 처리단계 순서를 재정렬하거나 어떠한 기능을 위해 추가/삭제할 수 있는 유연성 보장이 됨.

Architecture pattern 종류(15/37)

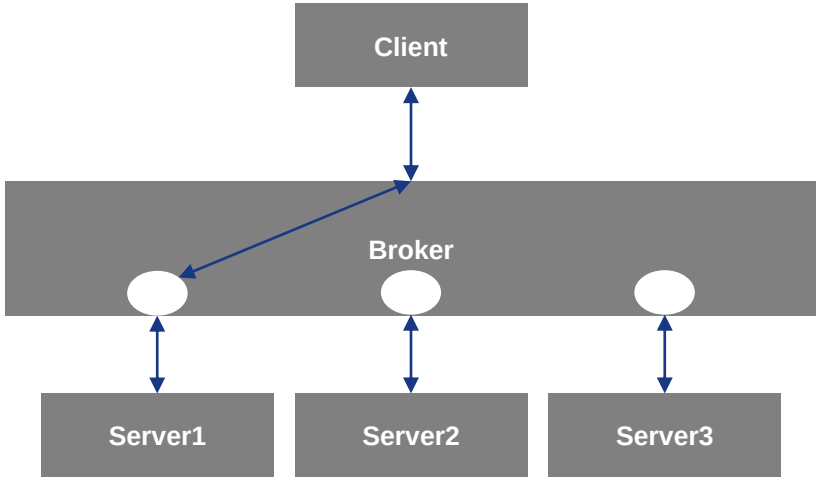
- **파이프-필터 패턴(Pipe-Filter pattern)**
  - 예시
    - 메시지 큐 시퀀스기반의 파이프라인을 구현하여 효율적인고 유연성 있는 큐 메커니즘을 제공함.



- 초기 Incoming message는 처리되지 않은 메시지 임. 지속적으로 메시지 큐에 쌓임.
- 필터작업을 구현된 구성요소는 처리되는 해당 큐의 메시지를 수신대기 시키고 작업을 진행한 다음 변환된 메시지를 시퀀스기반의 다음 큐에 게시함.
- 다른 필터작업은 완전히 변환된 데이터가 큐의 최종 메시지에 나타날 때까지 이 큐의 메시지를 수신대기함. 그리고 처리 및 결과를 다른 큐에 게시 함.
- 결국 정리하자면 파이프라인상의 큐에서 필터를 통해 입력 메시지를 수신하고, 처리하고, 결과를 다른 큐에 게시하는 것임.

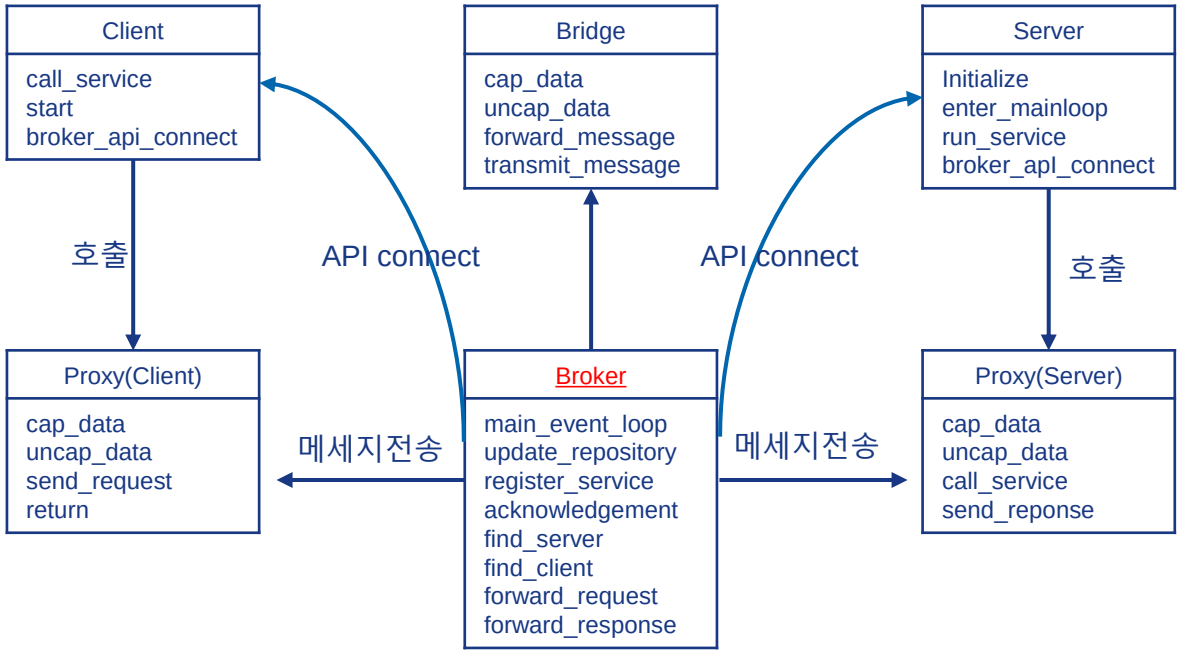
Architecture pattern 종류(16/37)

- **브로커 패턴(Broker pattern)**
  - 분리된 컴포넌트들로 구성된 분산시스템에 사용됨.
  - 분리된 컴포넌트들은 원격 서비스 실행을 통해 서로 상호작용을 할 수 있음.
  - 브로커 컴포넌트는 분산된 컴포넌트 간의 통신을 조정하는 역할을 함.
  - 서버는 자신의 기능들(서비스 및 특성)을 브로커에 전달하고 함.(Publish)
  - 클라이언트가 브로커에 서비스 요청을 하면 브로커는 클라이언트를 자신의 Registry에 있는 적합한 서비스로 Redirection함.
  - 적용 예)
    - ApacheMQ, Apache Kafka & RabbitMQ등



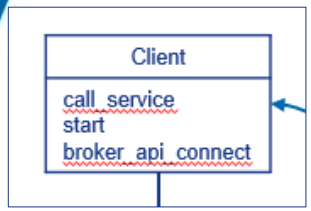
Architecture pattern 종류(17/37)

- **브로커 패턴(Broker pattern)**
  - 예시
    - 브로커 패턴은 결국 클라이언트와 서버를 제대로 분리하기 위해 브로커 컴포넌트를 도입하는 것임.
    - 아래 그림은 전형적인 브로커 패턴의 예제임.
    - 구현을 위해서 기본적으로 클라이언트, 서버, 브로커, 브리지, 클라이언트 프록시, 서버 프록시가 필요함.
    - 다음페이지부터 상세설명을 함.



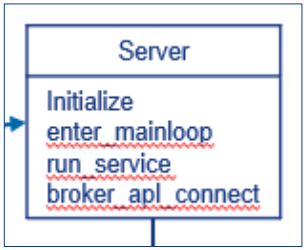
Architecture pattern 종류(18/37)

- **브로커 패턴(Broker pattern)**
  - 클라이언트



- 사용자의 기능구현을 담당하며 클라이언트의 프록시를 이용해 서버에 요청을 함.
- 하나 또는 그 이상의 서버에서 제공하는 서비스에 접속하는 Application이라고 할 수 있음.
- 원격서버에 서비스를 호출하기 위해서는 브로커를 통해 요청을 보내고 Operation이 실행됨. 그 이후 클라이언트는 브로커로부터 응답(response) 또는 예외(exception)의 피드백을 받음.
- “서버도 클라이언트 역할을 할 수 있다.”라는 서로와의 상호작용으로 동적모델을 형성할 수 있음.
- 구현관점에서 볼 때, 클라이언트는 Application, 서버는 Library로 간주 할 수 있음.

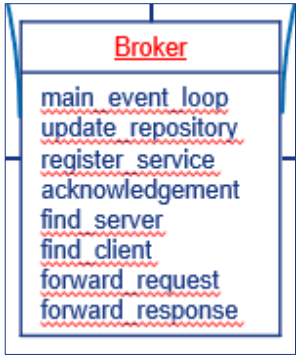
  - 서버



- 사용자를 위해 서비스를 구현하는 것을 담당함. 로컬브로커에 해다 서버를 등록함.
- 서버의 프록시를 통해 클라이언트에게 피드백 줄 응답과 예외를 전송함.
- Operation(동작) 과 Attribute(속성)으로 구성되어 있는 Interface를 통해 기능을 제공하는 객체를 구현
- 두 가지의 종류가 있음. 여러 Application 도메인들에 공통된 서비스들을 제공하는 서버, 하나의 Application 도메인이나 Task를 위해 특정 기능구현을 담당하는 서버로 나눌 수 있음.

Architecture pattern 종류(19/37)

- **브로커 패턴(Broker pattern)**
  - 브로커

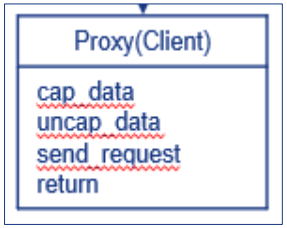


- 서비스를 제공하는 서버의 등록 및 제거를 위해 API를 제공함.
- 클라이언트 및 서버측 프록시 서버에 메시지를 전송함.
- 서비스를 제공 및 해당하는 서버의 위치를 찾음.
- 오류 복구를 함.
- Bridge컴포넌트를 통해 다른 브로커와 상호작용을 함

- 클라이언트에서 서버로 요청을 전송하는 역할만 하는 것이 아니라 그에 대한 피드백정보(응답/예외)를 다시 클라이언트에게 전송하는 역할도 함.
- 시스템 즉 서버의 식별을 위한 고유정보를 기반으로 요청을 받을 수신자의 위치를 파악하는 방법을 알고 있어야 함.
- 브로커는 서버를 등록하는 Operation(동작), 메서드를 호출하기 위한 Operation을 제공하는 API를 클라이언트와 서버에게 제공함.
- 로컬 브로커에 등록되어 있는 서버들 중에 요청작업을 처리할 서버를 찾고 그 요청작업을 서버에 직접 할당함.
- 만약에 해당서버가 활성화가 안되어 있다면 활성화 시킴. 서버의 서비스가 실행되면 발생하는 모든 응답과 예외들은 클라이언트에게 보내는 역할을 함.
- 만약에 서비스를 실행할 서버가 원격브로커로만 접근이 가능하다면, 로컬 브로커는 원격브로커의 접속경로를 찾아 그 정보를 이용해 접속 요청을 보냄.

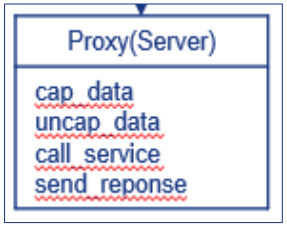
Architecture pattern 종류(20/37)

- **브로커 패턴(Broker pattern)**
  - 클라이언트측 프록시



- 어떤 특정 목적에 맞춰진 시스템으로 보안을 위해 캡슐화, 매개역할 등을 함.
- 클라이언트와 브로커 사이에서 중재(레이어)역할을 함.
- 중재역할을 위해 투명성을 제공해주고, 원격의 객체는 클라이언트 입장에서 보면 로컬 객체로 보일 수 있음.
- 프록시 시스템은 메모리 블록의 생성 및 제거 등 클라이언트의 일부 세부구현 정보를 숨김.
- 클라이언트와 브로커 간의 메시지 전송은 IPC(Inter-process communication: 메시지간의 통신) 메커니즘을 기반으로 처리됨.

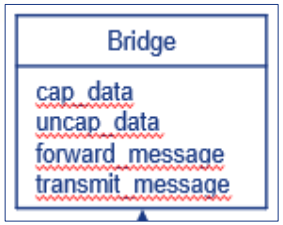
  - 서버측 프록시



- 클라이언트 측 프록시와 비슷함.
- 요청 받은 서버의 서비스 기능을 호출함. 또한 특정 시스템에 맞춰진 기능을 캡슐화 함.
- 서버와 브로커 사이에서 중재(레이어)역할을 함.
- 서비스 요청을 수신된 캡슐화 된 메시지를 풀고 매개변수를 Unmarshaling함. 그리고 적절한 서비스를 호출함.
- 응답을 위한 결과 및 예외를 클라이언트에게 보내기전에 Marshaling(한 객체의 메모리에서 표현방식 또는 처리 등을 적합한 다른 데이터 형식으로 변환하는 과정)을 수행 함.
- Marshaling처리가 된 데이터를 클라이언트측 프록시가 브로커로부터 수신하여 Unmarshaling한 후 클라이언트에게 전송함.

Architecture pattern 종류(21/37)

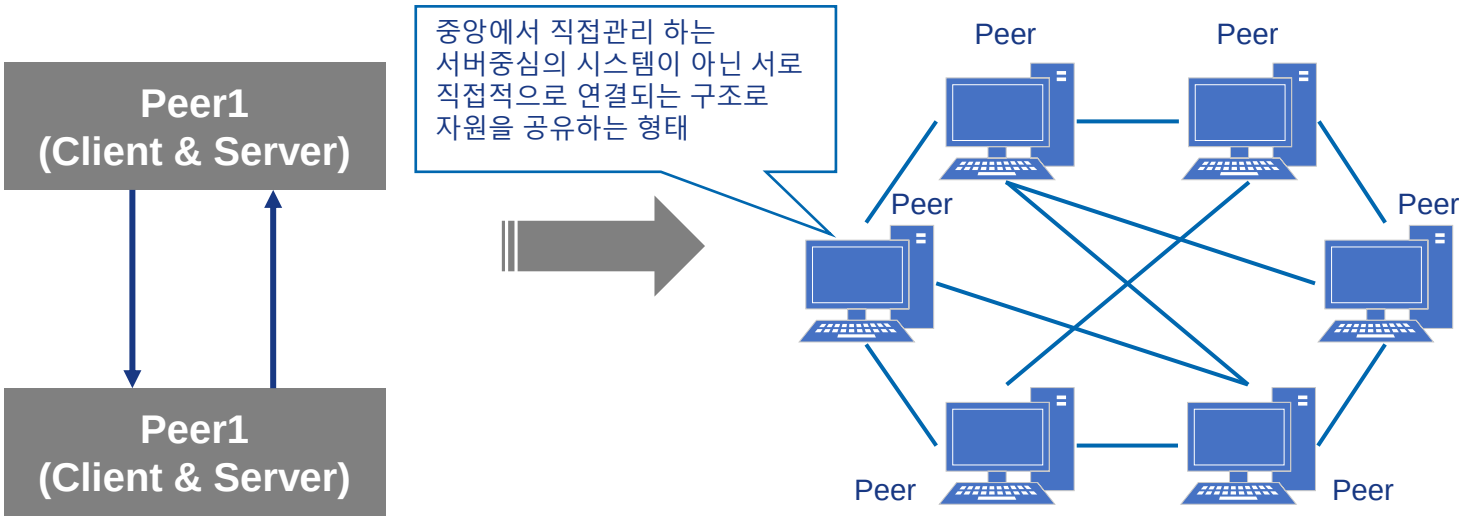
- **브로커 패턴(Broker pattern)**
  - 브리지



- 브리지는 네트워크 수준에서도 많이 활용하는 기술임. 즉 특정 네트워크에 맞춰진 기능을 캡슐화 함.
- 원격 브로커와 로컬 브로커의 사이를 중재하는 역할을 함.
- 브리지는 필수적인 컴포넌트는 아님. 원격과 로컬의 두 브로커가 상호운영적 일 때 세부정보 및 구현을 숨기기 위한 옵션으로 활용됨.
- 브로커시스템들이 서로 다른 특성을 가진 네트워크에서 실행되거나 조건이 있을 때, 브리지는 이를 위해 특정 시스템고 관련된 모든 세부구현 정보들을 캡슐화 한 중재(레이어)역할을 함.
  - 서비스 요청이 네트워크를 통해 요청이 되면 각 브로커는 처리에 있어 해당 네트워크 및 시스템 환경에 영향을 받지 말아야 함. 즉 독립적으로 문제없이 통신해야 한다는 의미임.

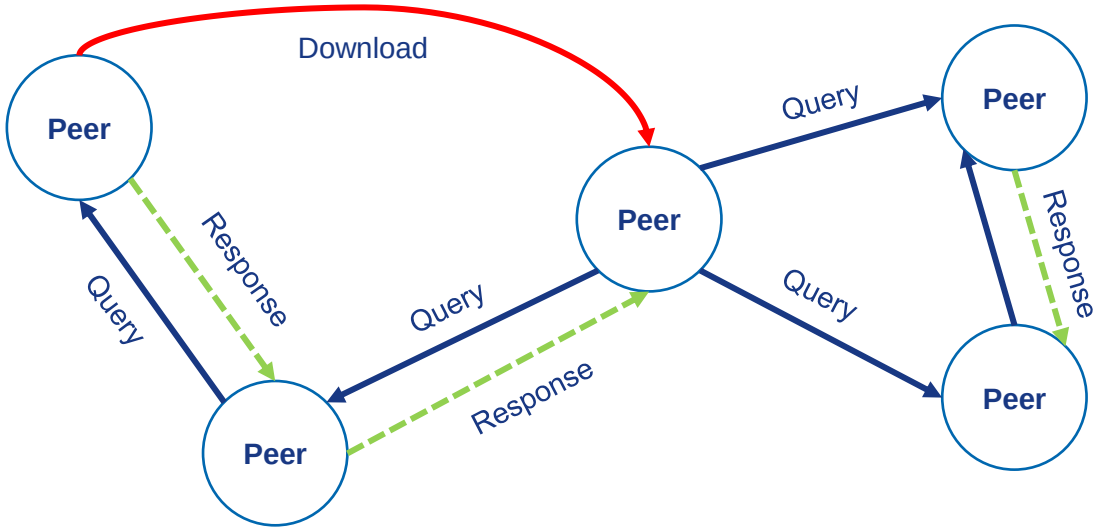
Architecture pattern 종류(22/37)

- **피어 투 피어 패턴(Peer-to-peer pattern)**
  - 각 endpoint 컴포넌트들을 피어라고 함.
  - 피어는 클라이언트 또는 서버가 될 수 있음. 다시 말해 하나의 컴포넌트로 간주함. 피어가 피어에게 서비스를 요청할 수 있고, 서버로서 각 피어에게 서비스를 제공할 수도 있음. 결국, 네트워크 수준에서 클라이언트와 서버의 개념이 없음.
  - 클라이언트-서버패턴과 다르게 항상 작동되고 있는 서버가 존재하지 않음.
  - 각 피어는 상황 및 시간이 지남에 따라 역할이 유동적으로 바뀔 수 있음.
  - 적용 예)
    - P2PTB or PDTP와 같은 멀티미디어 프로토콜
    - Spotify와 같은 독점적 멀티미디어 Application
    - Gnutella G2와 같은 파일 공유 네트워크 서비스



Architecture pattern 종류(23/37)

- **피어 투 피어 패턴(Peer-to-peer pattern)**
  - 예시
    - 분산 또는 중앙의 서버에 집중하기 보다는 네트워크에 참여한 시스템들의 성능 및 대역폭들에 의존하여 구성회는 형태임.
    - Gnutella P2P network architecture
      - 아래 그림은 P2P패턴기반의 Gnutella의 네트워크 아키텍처를 나타낸 것임. 각 피어들은 동등한 위치에서 쿼리(서비스 요청) 및 응답을 처리하고 있음.
      - 네트워크에 참여한 각 피어는 가지고 있는 서비스 자원들을 서로 공유하고 상호연결 및 작동하여 다운로드를 하고 있음.
      - 피어는 P2P 관계로 계속 늘어날 수 있으며 리소스 공유처리는 네트워크 및 여러 상황에 따라서 변화될 수 있음.

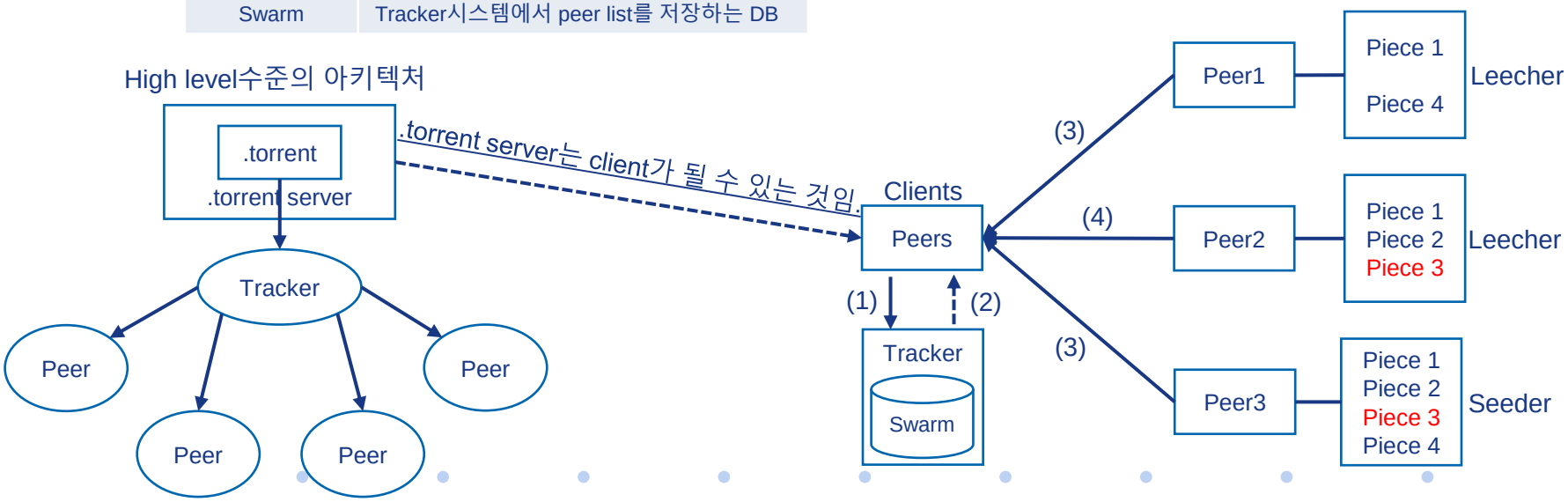


Architecture pattern 종류(24/37)

- **피어 투 피어 패턴(Peer-to-peer pattern)**
  - 예시
    - BitTorrent P2P architecture
      - 아래 그림은 P2P패턴기반의 대표적인 BitTorrent application시스템의 아키텍처임.
      - 오른쪽의 피어들의 리소스를 어떻게 공유하는지 아래와 같이 설명할 수 있음.

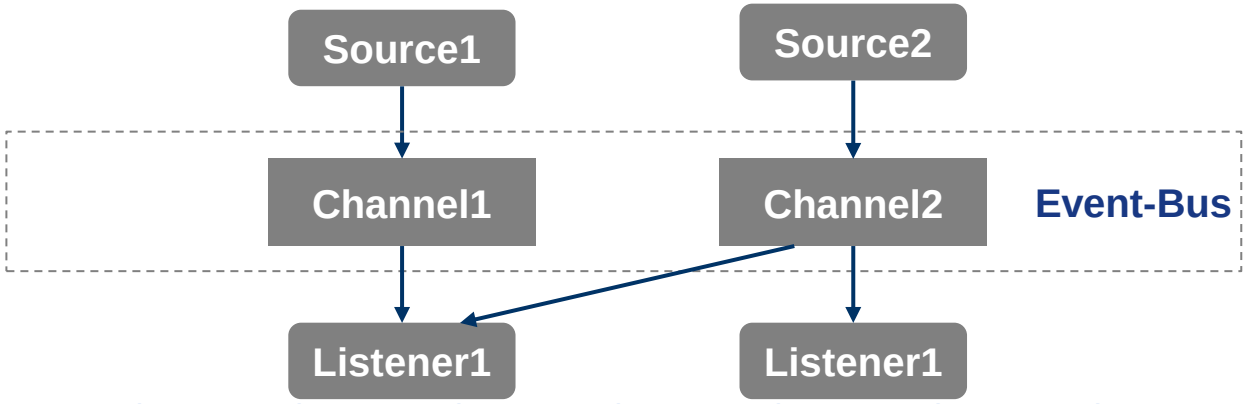
구분	설명
Piece	공유파일의 한 조각(256kb),
Peer	Piece들을 Upload / Download 하는 주체
Seeder	모든 piece정보들을 가지고 있는 피어
Leecher	일부 piece정도들을 가지고 있는 피어
Tracker	클라이언트에게 peer list를 전달하는 시스템
Swarm	Tracker시스템에서 peer list를 저장하는 DB

- BitTorrent의 작동원리
(1) 파일을 찾기위해 Hash기반의 A.torrent파일을 만들어 전송함.
(2) 찾고자 하는 피어정보를 리스터업 하여 요청 한 클라이언트에게 전송함.
(3) 해당 피어들의 정보들을 다운로드 함.
(4) Piece3를 업로드 / 다운로드 진행 함.



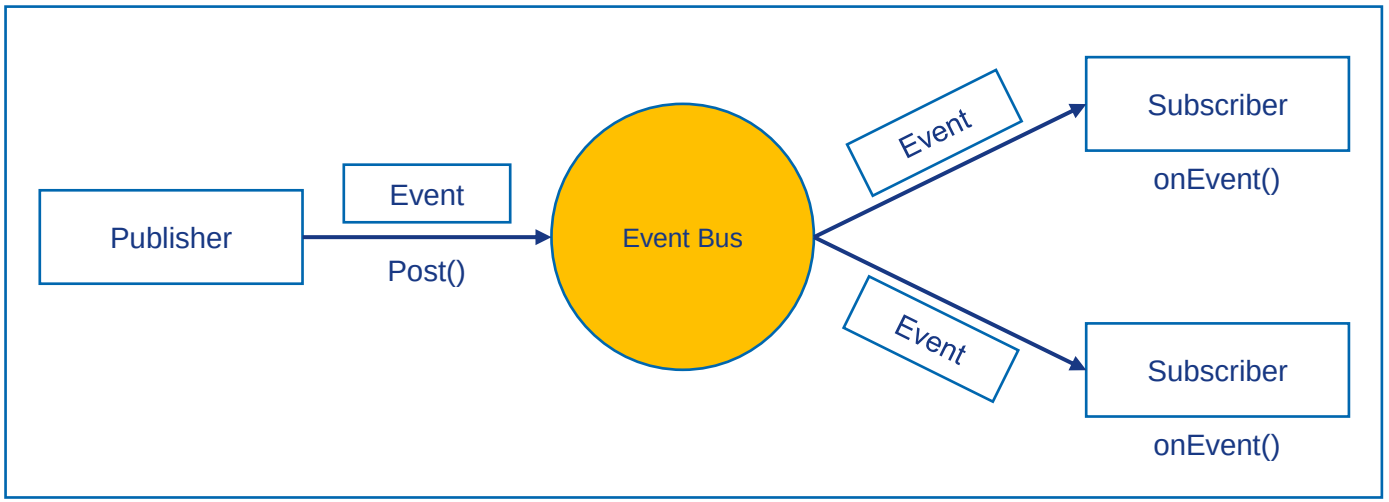
Architecture pattern 종류(25/37)

- **이벤트-버스 패턴(Event-bus pattern)**
  - 확장성을 고려한 반응형 application을 개발하기 위한 비동기식 분산 아키텍처 임.
  - 이벤트버스는 서로 다른 컴포넌트가 알지 못하는 상태에서 통신 할 수 있도록 하는 매커니즘임.
  - 4가지의 주요 컴포넌트로 구성됨.
    - 이벤트 처리기(Event source): 이벤트 생성자
    - 이벤트 리스너(Event listener):
    - 채널
    - 이벤트 버스(Event bus)
  - 소스는 이벤트 버스를 통해 특정채널로 메시지를 보냄.
  - 리스너는 특정채널에서 보내진 메시지를 읽음.
  - 적용 예)
    - Android 개발
    - 알림 서비스



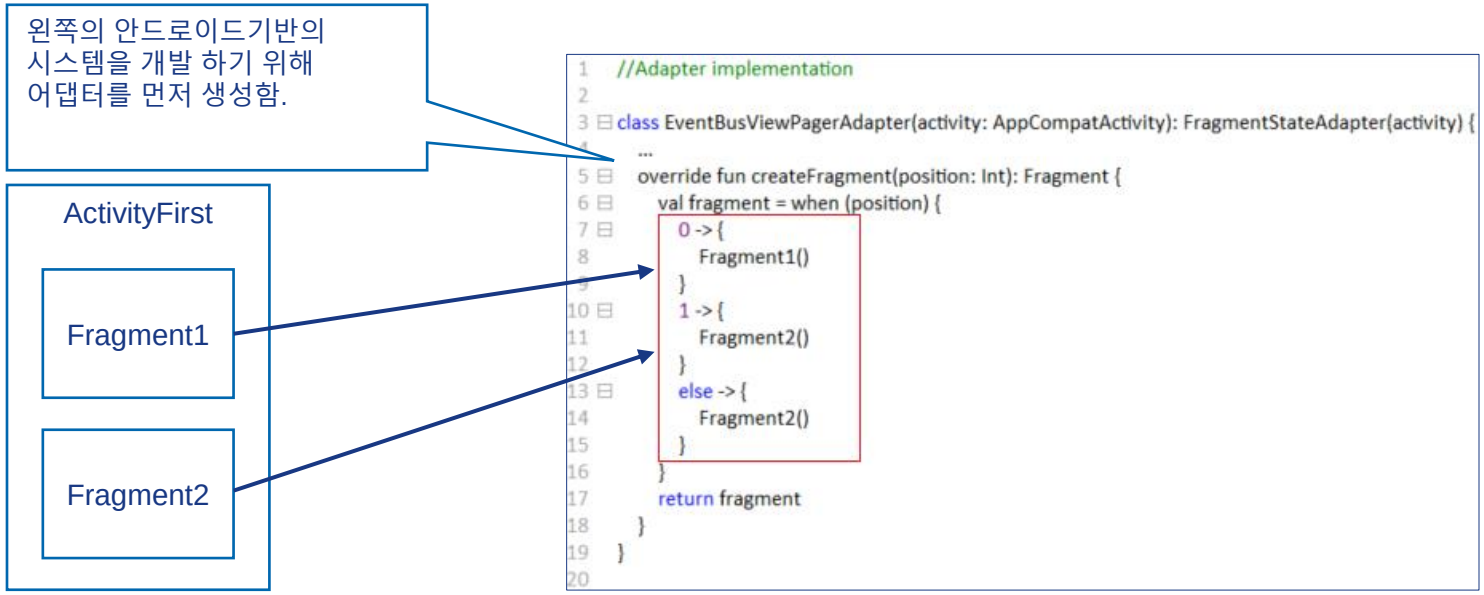
Architecture pattern 종류(26/37)

- **이벤트-버스 패턴(Event-bus pattern)**
  - 이벤트-버스 패턴을 사용 시 다음과 같은 이점이 있음.
    - 이벤트의 발신자 및 수신자를 완벽하게 분리할 수 있음.
    - 복잡하고 오류가 발생하기 쉬운 종속성 및 Life-cycle 문제를 방지할 수 있음.
    - 컴포넌트 간의 통신을 단순화 시킬 수 있음.
    - 안드로이드앱 개발을 위해 UI(액티비티, 프래그먼트) 그리고 백그라운드 스레드의 호환 및 작동이 잘 됨.
  - 예시
    - 아래 그림은 전형적인 게시-구독자 패턴과 이벤트-버스를 접목시킨 패턴임.
    - 이벤트를 보내는 게시자(Publisher)가 Post() 함수에 어떤 이벤트를 포함하여 Event Bus에 보냄.
    - Event Bus를 구독하고 있는 Subscriber들은 onEvent() 콜백을 통해 이벤트를 전송함.



Architecture pattern 종류(27/37)

- **이벤트-버스 패턴(Event-bus pattern)**
  - 예시
    - 일반적으로 안드로이드 앱 개발 시 Activity, Fragment, Service, Presenter, Listener 인터페이스, Adapter 등 여러 컴포넌트로 구성됨.
    - 여러 컴포넌트들은 긴밀하게 연결되어 있어 유연성 있는 통신이 어려움. 또한 리스너 인터페이스 기반으로 구현 시 긴밀한 커플링(결합) 이슈, 코드반복, 난해한 테스팅, 의존성 핸들링 이슈등이 발생함.
    - 상위 이슈사항들은 게시/구독 및 이벤트/버스 패턴을 사용하면 상당히 개선시킬 수 있는 효과를 얻을 수 있음. 다른 컴포넌트를 신경 쓸 필요없이 컴포넌트간의 효과적인 통신을 보장하기 때문임.
    - 아래 그림은 Event-bus패턴을 이용한 안드로이드 기반의 App개발 예시임.

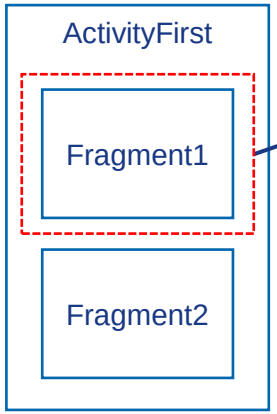


Architecture pattern 종류(28/37)

- **이벤트-버스 패턴(Event-bus pattern)**
  - 예시
    - 어댑터 생성 후 Fragment 2개를 생성해줌.

왼쪽의 Fragment1의 생성을 위한 로직임. 해당 Fragment에서 Subscriber로써 onCreate(), onStop()를 통해 Eventbus를 구독하고 해지하는 것을 볼 수 있음,

Fragment2도 같은 로직으로 생성함.

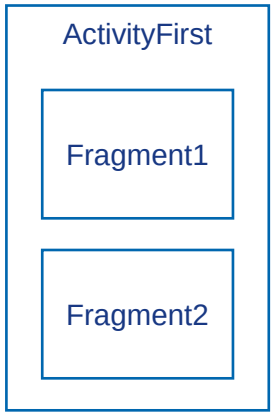



```
//Fragment implementation
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".eventbus.Fragment1">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Fragment1" />
</FrameLayout>
```

```
class EventBusFragment1 : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_event_bus_first, container, false)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        EventBus.getDefault().register(this)
        Log.e("프래그먼트1", "이벤트 등록")
    }

    @Subscribe(threadMode = ThreadMode.MAIN)
    fun getUser(user: User) {
        Toast.makeText(requireActivity(), "1번 프래그먼트에서 받은 user : ${user.name}", Toast.LENGTH_SHORT).show()
    }

    override fun onStop() {
        EventBus.getDefault().unregister(this)
        super.onStop()
        Log.e("프래그먼트1", "이벤트 해제")
    }
}
```

Architecture pattern 종류(29/37)

- **이벤트-버스 패턴(Event-bus pattern)**
  - 예시
    - EventBus를 통해 보낼 클래스를 생성함.
    - 그 다음 Activity를 생성함.



EventBus를 통해 보낼 이벤트 클래스 정의

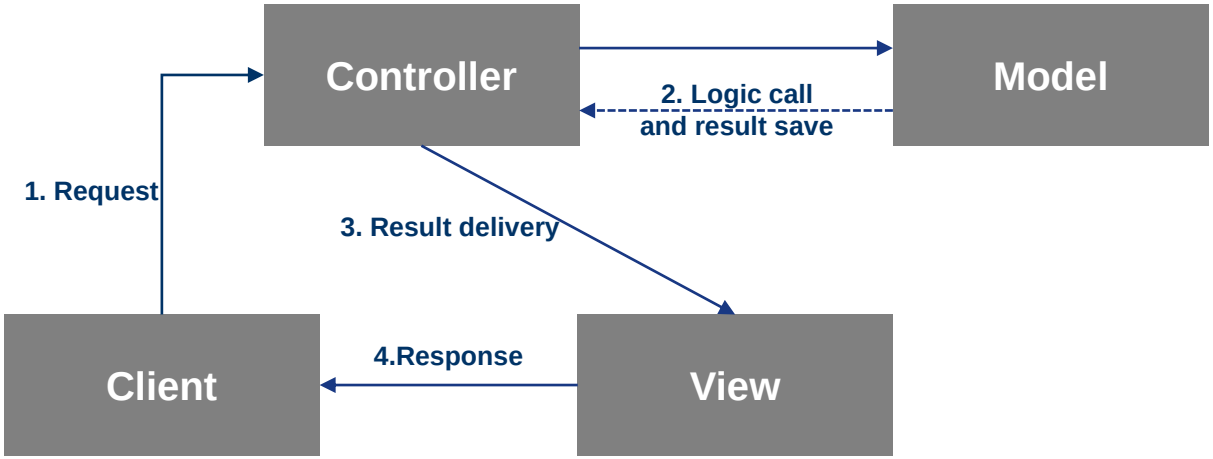
```
//Class and function definition through EventBus
...
@Override
public void onCreate() {
    super.onCreate();
    EventBus.getDefault().register(this);
}

@Override
public void onStop() {
    super.onStop();
    EventBus.getDefault().unregister(this);
}
```

Activity 생성

```
//Activity implementation
class EventBusTestActivity : AppCompatActivity() {

    private val binding: ActivityEventBusTestBinding by lazy {
        ActivityEventBusTestBinding.inflate(layoutInflater)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(binding.root)

        init()
    }

    private fun init() {
        binding.run {
            vpTest.run {
                offscreenPageLimit = 1
                adapter = EventBusViewPagerAdapter(this@EventBusTestActivity)
            }

            btnFirstFragment.setOnClickListener {
                EventBus.getDefault().post(User(name = "user"))
                vpTest.currentItem = 0
            }
            btnSecondFragment.setOnClickListener {
                EventBus.getDefault().post(User(name = "user2"))
                vpTest.currentItem = 1
            }
        }
    }
}
```

Architecture pattern 종류(30/37)

- **모델-뷰-컨트롤러 패턴(Model-view-controller pattern)**
  - 흔히 MVC패턴이라고도 함.
  - 대화형 어플리케이션(Interactive application)을 개발 시 아래와 같이 3부분으로 나눔.
    - 모델(Model): 핵심적인 기능 및 데이터를 포함하여 처리하는 영역임.
    - 뷰(View): UI(User Interface),사용자에게 정보를 표시함. HTML 및 CSS등을 통해 화면의 디자인을 처리함.
    - 컨트롤러(Controller): 뷰를 통해 전달받은 작업요청을 모델과 처리하여 다시 그 결과를 뷰에게 전달하는 역할을 함.
  - 각 컴포넌트를 분리하여 코드의 효율적인 재사용을 가능하게 한다.
  - 적용 예)
    - 일반적인 웹 어플리케이션 설계 아키텍처
    - Django / Rails



Architecture pattern 종류(31/37)

- **모델-뷰-컨트롤러 패턴(Model-view-controller pattern)**
  - 예시
    - 예를들어 회원정보저장구현을 위해 MVC패턴을 적용한다면 아래그림과 같이 할 수 있음.

Client측에서 보내는 요청사항과 같음.
Controller 클래스에 요청을 보내기 위해
MemberContoller 인스턴스를 생성함.

```
package MVCPattern;

public class MVCTest {

 public static void main(String[] args) {
    MemberModel      membermodel    = new MemberModel("dave.kim", "111-1111-1111");
    MemberView       memberview     = new MemberView();
    MemberController membercontroller = new MemberController(membermodel, memberview);
    membercontroller.updateView();

    membercontroller.setUserName("dave.kim");
    membercontroller.updateView();

    membercontroller.setPhoneNumber("111-1111-1111");
    membercontroller.updateView();

  }
}
```

Main함수에서 Memberview
인스턴스를 통해 printView를
호출하고 결과값을 보이게 함.

```
//View implementation
package MVCPattern;

public class MemberView {
  // State query
  public void printView(MemberModel membermodel) {
    System.out.println("User name: " + membermodel.getUserName());
    System.out.println("User phone number: " + membermodel.getPhoneNumber());
  }
}
```

MemberModel의 호출등을 위해 게터/세터로
데이터를 입력 및 저장을 인스턴스 변수로 생성함.
이를 위해 MemberModel의 인스턴스를 통해
setUserName, getUserName,
setUserPhoneNumber getUserPhoneNumber
함수를 정의함.

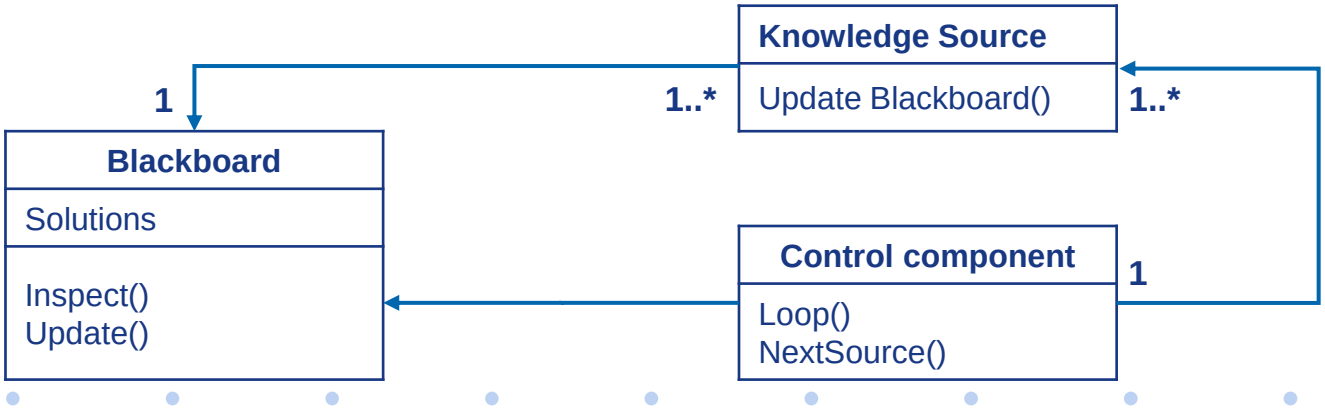
```
//Controller implementation
package MVCPattern;

public class MemberController {
  private MemberModel membermodel;
  private MemberView memberview;

  public MemberController(MemberModel membermodel, MemberView memberview) {
    this.membermodel = membermodel;
    this.memberview = memberview;
  }

  // Update from setter/getter for user name
  public void setUserName(String user_name) {
    membermodel.setName(user_name);
  }
  public String getUserName() {
    return membermodel.getUserName();
  }

  // Update from setter/getter for user phone number
  public void setPhoneNumber(String user_phon_number) {
    membermodel.setPhoneNumber(user_phon_number);
  }
  public String getPhoneNumber() {
    return membermodel.getPhoneNumber();
  }

  // View call
  public void updateView() {
    memberview.printView(membermodel);
  }
}
```

MemberModel 클래스를 생성하고
초기화를 위해 생성자 MemberModel이
생성됨. 데이터설정을 위해
세터/게터함수가 정의됨. 또한
MemberController에 정의된 인스턴스
변수로 전달됨.

```
//Model implementation
package MVCPattern;

public class MemberModel {
  private String user_name;
  private String user_phon_number;

  public MemberModel(String user_name, String user_phon_number) {
    this.user_name = user_name;
    this.user_phon_number = user_phon_number;
  }

  public String getUserName() {
    return user_name;
  }
  public void setUserName(String user_name) {
    this.user_name = user_name;
  }

  public String getPhoneNumber() {
    return user_phon_number;
  }
  public void setPhoneNumber(String user_phon_number) {
    this.user_phon_number = user_phon_number;
  }
}
```

Architecture pattern 종류(32/37)

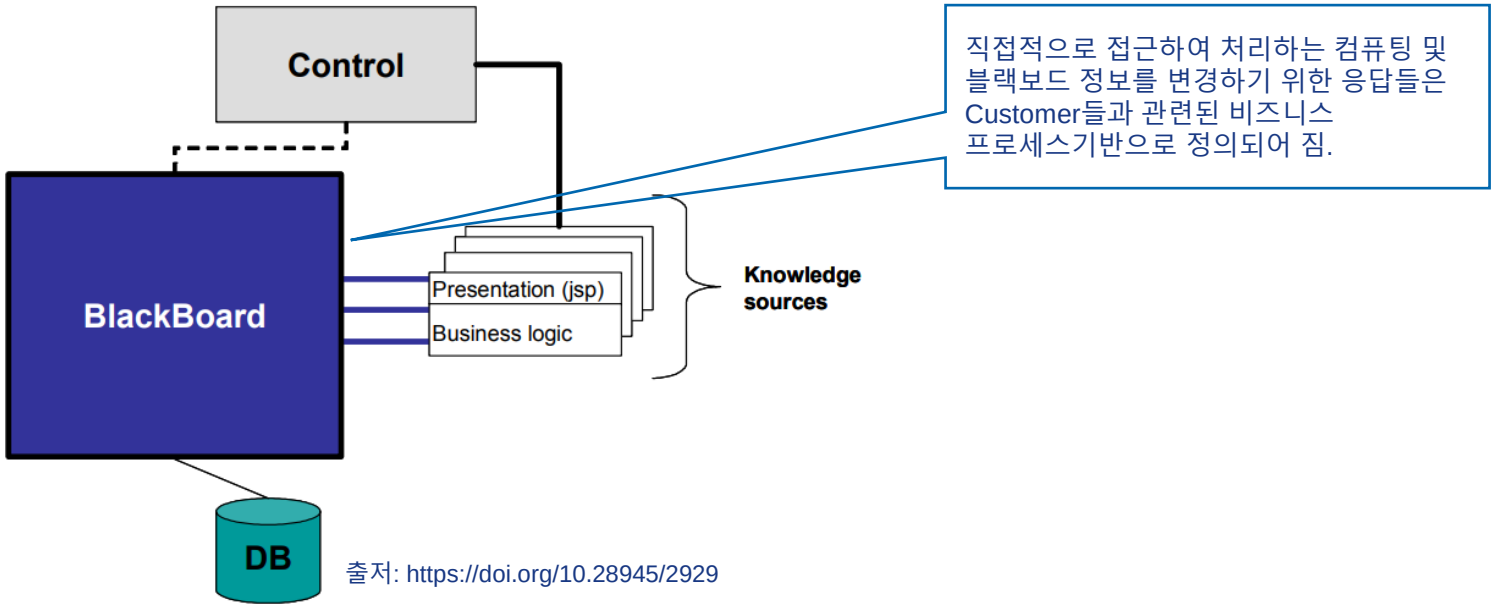
- **블랙보드 패턴(Blackboard pattern)**
  - 해결되지 않은 문제에 대해서 결정을 할 때 유용한 패턴
  - 3가지의 주요 컴포넌트로 구성됨.
    - 블랙보드(Blackboard): 솔루션의 객체를 포함하는 구조화된 전역메모리 또는 DB
    - 지식소스(Knowledge source): 자체 표현을 가진 특수 모듈
    - 제어 컴포넌트(Control component): 모듈선택, 설정 및 실행을 담당
  - 모든 컴포넌트는 블랙보드에 접근함.
  - 컴포넌트는 블랙보드에 추가되는 새로운 데이터 객체를 생성할 수 있음.
  - 컴포넌트는 특정 종류의 데이터를 찾기 위해 기존의 지식소스와 패턴매핑을 통해 처리 됨.
  - 적용 예)
    - 일음성인식
    - 차량식별 및 추적
    - 단백질 구조 식별
    - 수중 음파 탐지기 신호 해석등



Architecture pattern 종류(33/37)

- **블랙보드 패턴(Blackboard pattern)**
  - 예시
    - 아래 그림은 블랙보드기반의 웹어플리케이션 컴포넌트 구성을 나타낸 것임.
    - 시스템의 데이터 및 쿼리실행을 위해 메가클래스(수퍼클래스)가 사용되어 졌음. 메가클래스는 데이터베이스 운영 및 비즈니스 로직을 위한 새로운 기능추가 클래스 지원하는 서비스 수를 최소화 할 수 있음.
    - Control은 표현 및 비즈니스 로직계층 사이에서 logical 티어로 작동함. Control은 Knowledge sources(KS) 와 연결되는 jsp파일 및 비즈니스 로직객체들을 통제 및 관리하는 역할을 함. 이러한 측면에서 블랙보드는 계층화된 KS를 포함함.

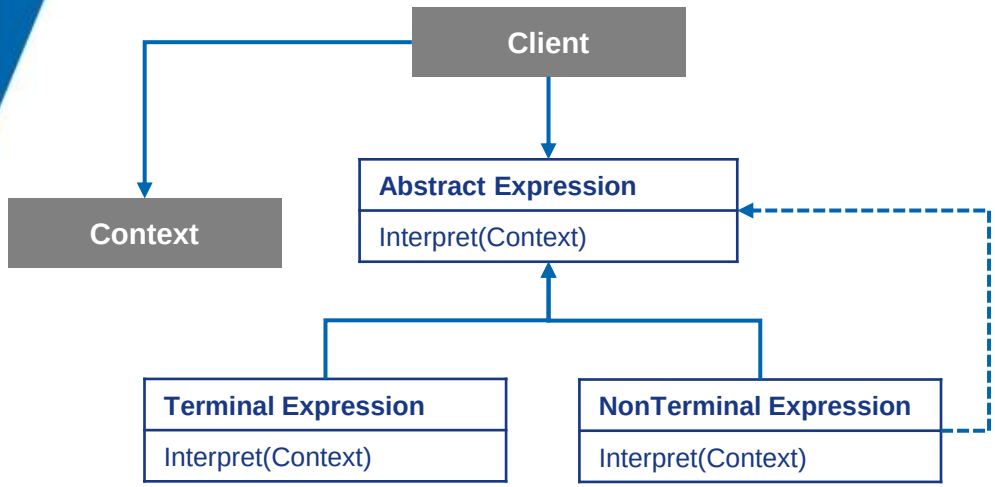
블랙보드 패턴 기반의 아키텍처 컴포넌트



출저: https://doi.org/10.28945/2929

Architecture pattern 종류(34/37)

- **인터프리터 패턴(Interpreter pattern)**
  - 특정언어로 작성된 프로그램을 해석하는 컴포넌트를 설계할 때 사용됨.
  - 특정언어로 작성(개발)된 프로그램의 각 라인을 수행하는 방법을 지정함.
  - 기본적인 아이디어는 언어의 각 기호에 대해 클래스를 만드는 것임.
  - 언어의 문법을 기술하는 규칙들에 대한 형식을 정의함. 그 규칙들은 클래스를 통해 구현 됨.
  - 클래스는 Context객체들을 공유하고 그 객체들로 변수에 값을 입력 받고 저장하는 작업들을 하게 됨
  - 인터프리터는 어떠한 웹 application을 개발 시 어떤 UX/UI디자이너가 무엇을 동작 시키고 싶을지에 대해서 추측이 불가능할 때 Java script같은 인터프리터 언어를 이용하여 개발자가 하지 않아도 그 동작을 구현할 수 있도록 하는 것임 .
  - 적용 예)
    - SQL과 같은 데이터베이스 쿼리 질의어
    - 통신 프로토콜을 정의하기 위한 언어



구분	설명
Client	언어로 정의한 특정문장을 나타내는 추상 구문트리 임. TE와 NTE클래스의 인스턴스로 구성됨. 또한 Interpret()를 호출함.
Context	인터프리터의 변수에 저장된 포괄적인 정보들을 의미함.
AbstractExpression (AE)	추상구문트리에 속한 모든 노드클래스들이 공통으로 가져야할 Interprete()연산을 추상화하여 정의 함.
TerminalExpression (TE)	문법을 기술한 규칙들과 같은 터미널 기호와 관련된 해석방법을 구현함.
NonTerminalExpression (NTE)	문법(규칙)의 오른쪽에 나타나는 모든 기호에 대해서 클래스를 정의함. AE와 연관이 될 수 있으니 재귀적으로 해석할 필요가 있음.

Architecture pattern 종류(35/37)

- **인터프리터 패턴(Interpreter pattern)**
  - 예시
    - 조회 SQL문을 구현하는 예제를 한번 보도록 함. 기본적으로 조회구문은 “SELECT * FROM WHERE”로 구성됨.
    - 우선 Interface를 통해 구문해석 하는 방법을 정의함. Interface란 여러 프로그램 또는 클래스에서 사용할 변수, 메서드들을 일관되게 구현하기 위한 기술명세임.

```
// Interface definition
public interface Printway
{
    List<String> interpret(Context ctx);
}
```

    - SELECT에 해당하는 클래스를 정의함.
    - 미리 정의된 Print인터페이스에서 상속받음. 사용될 열 이름 Column을 정의하고 From으로부터 또 다른 표현식의 생성자 from 매개변수를 가져옴.
    - 또한 재정의를 위해 interpret() 메서드에서 context를 변수로 받아 업데이트하고 해석한 결과를 2개이상의 다른 메서드들을 통해 전달함. 즉 NonTerminalExpression(비터미널표현)에 해당되는 것임.

```
// Select class definition
class Select implements Printway
{
    private String column;
    private From from;

    // 생성자
    @Override
    public List<String> interpret(Context ctx) {
        ctx.setColumn(column);
        return from.interpret(ctx);
    }
}
```

Architecture pattern 종류(36/37)

- **인터프리터 패턴(Interpreter pattern)**
  - 예시
    - FROM에 해당하는 클래스를 정의함.
    - SELECT클래스 원리와 같음. Where의 인스턴스 변수를 선언함. 또한 여기도 interpret() 메서드를 오버라이딩을 통해 재정의를 하고 Where 클래스를 통해 값이 있으면 컨텍스트(변수)값으로 조회하고 아니면 search()메서드를 실행하고 종료함.

```
// From class definition
class From implements Printway {
    private String table;
    private Where where;

    // 생성자
    @Override
    public List<String> interpret(Context ctx) {
        ctx.setTable(table);
        if (where == null) {
            return ctx.search();
        }
        return where.interpret(ctx);
    }
}
```

    - WHERE에 해당하는 클래스를 정의함.
    - 이 부분은 SQL문에서도 조건절로 존재할 수 도 있고 없을 수 도 있음. 따라서 터미널/비터미널표현 두 방식이 모두 해당 될 수 있음.
    - Filter의 인스턴스변수를 통해 검색에 필요한 필터를 생성하고 컨텍스트를 다시 업데이트 하여 search()메서드를 실행 함.

```
//Where class definition
class Where implements Printway {
    private Predicate<String> filter;

    // constructor

    @Override
    public List<String> interpret(Context ctx) {
        ctx.setFilter(filter);
        return ctx.search();
    }
}
```

Architecture pattern 종류(37/37)

- **인터프리터 패턴(Interpreter pattern)**
  - 예시
    - Context에 해당하는 클래스를 정의함.
    - 해당 부분은 DB Connect메서드를 통해 데이터를 가지고 오고 해당 리스트객체에 데이터를 입력해야 하는 부분임. 여기서는 list.add를 통해 두개의 필드를 입력하고 하위 클래스(Select, From, Where)에 의해 처리됨.

```
// Context definition
class Context
{
    private static Map<String, List<Row>> tables = new HashMap<>();

    static {
        List<Row> list = new ArrayList<>();
        list.add(new Row("Dave", "Kim"));
        list.add(new Row("Ga", "Chon"));

        tables.put("member", list);
    }

    private String table;
    private String column;
    private Predicate<String> whereFilter;

    // ..... 생략

    List<String> search() {

        List<String> result = tables.entrySet()
          .stream()
          .filter(entry -> entry.getKey().equalsIgnoreCase(table))
          .flatMap(entry -> Stream.of(entry.getValue()))
          .flatMap(Collection::stream)
          .map(Row::toString)
          .flatMap(columnMapper)
          .filter(whereFilter)
          .collect(Collectors.toList());

        clear();

        return result;
    }
}
```

검색을 완료되면 컨텍스트의 데이터가 지워지므로 기본값으로 설정이 됨.

해당 필터가 존재하면 CollectionMapping해서 처리하고 아니면 clear() 메서드를 실행하여 초기화 함.

마지막으로 해당 결과를 result를 통해 리턴함.

https://recordsoflife.tistory.com/1135

Architecture pattern 비교

- **10개의 아키텍처패턴 비교**

아키텍처	장점	단점
Layered	레이어 표준화가 쉽고 레이어 수준 정의도 쉬움. 레이어를 변경해도 다른 레이어에 영향을 미치지 않음.	광범위하게 적용이 어려움. 어떤 상황에서는 특정 레이어가 불필요할 수 있음.
Client-Server	client가 요청할 수 있는 일련의 서비스 모델링을 할 수 있음.	요청은 일반적으로 서버에서 스레드로 처리됨. 프로세스간의 통신은 클라이언트간 서로 다르게 표현되므로 오버헤드 발생 가능.
Master-Slave	정확성이 좋음. 서비스의 실행은 각기 다른 구현체를 가진 슬레이브들에게 전파됨.	슬레이브가 독립적이므로 공유되는 상태가 없음. 마스터-슬레이브사이에 Latency가 발생할 수 있음.
Pipe-filter	동시성 처리가 좋음. 시스템 확장성이 좋음. 파이프라인의 필터 재사용성 및 추가가 쉬움.	가장 느린 필터연산에 의해 효율성이 제한될 수 있음. 필터사이의 데이터 이동 시 변환에 따른 오버헤드 발생
Broker	객체의 동적인 추가,수정,삭제 및 개발자의 배포 투명성 보장	서비스표현에 대한 표준화가 필요함.
Peer-to-Peer	분산된 컴퓨팅을 지원함. 특정 노드장애 대응에 매우 강함. 리소스 및 컴퓨팅 성능면에서 확장성이 뛰어남.	노드들이 스스로 참여하기 때문에 서비스 품질에 대한 보장이 어려움. 보안 취약성 존재, 노드의 개수에 따라 성능이 좌우됨.
Event-Bus	새로운 발행자와 구독자의 추가연결이 쉬움. 고도로 분산화 된 Application에 효과적임.	모든 메시지가 동일한 이벤트버스를 통해 전달되기 때문에 확장성 문제가 발생할 수 있음.
MVC	동일한 모델에 여러 개의 뷰를 만들 수 있음. Runtime에 동적으로 연결 및 해제를 할 수 있음.	복잡성이 증가할 수 있으며, end-user의 행동에 대해 불필요한 업데이트가 많이 발생 할 수 있음.
Blackboard	새로운 Application을 쉽게 추가할 수 있음. 데이터 공간의 구조를 쉽게 확장할 수 있음.	모든 Application들이 영향을 받기 때문에 데이터 공간구조를 변경하기 어려움. 동기화 및 접근제어가 필요할 수 있음.
Interpreter	매우 동적인 설계를 할 수 있음. 최종사용자가 프로그래밍하기 쉬움. Interpreter를 쉽게 교체할 수 있기 때문에 유연성이 좋음.	Interpreter는 일반적으로 컴파일 언어보다 느리기 때문에 성능 문제가 발생할 수 있음.

Architecture pattern 기반 인프라 설계실습

- **10개의 아키텍처 기반으로 안정된 Application 아키텍처 설계실습**
  - **조건**
    - 기본적으로 Client ←→ Server의 큰 Frame으로 구성
    - 트래픽 분산을 고려해야 함.
    - 각 컴포넌트(Frontend, Backend & DB)는 이중화 구조로 되어야 함.
    - 각 컴포넌트간의 통신은 어떻게 이루어져야 하는지 명확하게 표시해야 함.
    - DB백업 및 통합이 이루어져야 하며 반드시 명확한 파이프라인을 통해 처리되어야 함.
    - 각 컴포넌트들의 계층은 목적과 특성(상위모듈-하위모듈)에 따라 명확하게 구분되어야 함.
    - 보안적 요소를 반드시 적용해야 함.
      - 예: 접근제어(Access control)등

패턴을 이해하고 고민~고민~해서 설계 해보세요~