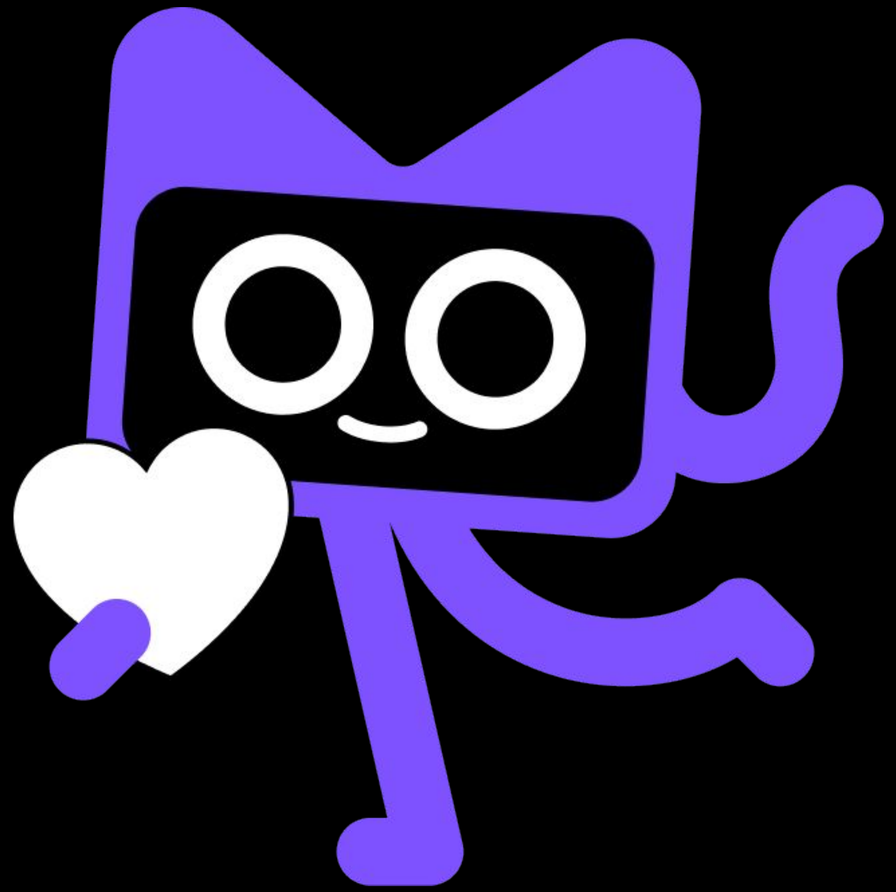
Kotlin User Groups Seoul

Kotlin Backend Meetup

Spring Webflux Overview with Armeria



Sunghoon Park

현) Staff Backend Engineer
Retail Engineering Team of the Coupang
전) 인증플랫폼 of Naver

Slipp Community

Email: monorisk@gmail.com
LinkedIn: https://www.linkedin.com/in/chupin-park
Youtube: https://www.youtube.com/@chupin-park

누구를 위한 발표인가요?

1. Spring Webflux의 내부 구현을 알고 싶은 분
2. 성능을 위해 Spring Webflux로 개발하시는 분
3. Async/Non-blocking System에 대해서 궁금하신 분

누구를 위한 발표인가요?

1. Spring Webflux의 내부 구현을 알고 싶은 분
2. 성능을 위해 Spring Webflux로 개발하시는 분
3. Async/Non-blocking System에 대해서 궁금하신 분

오늘은 Reactive Streams 보다는 Async/Non-blocking에 집중

왜 Spring Webflux를 이용하나요?

1. Event base Async/Non-blocking
2. Publish/Subscribe
3. Backpressure
4. 선언적 프로그래밍

등등의 Reactive Streams의 철학을 바탕으로 Web Application을 개발하기 위해

Async/Non-blocking을 통한 Thread 사용 효율화

Servlet API의 문제점

Spring Webflux 서문에 Servlet API의 문제가 서술
- Servlet API는 Async/Non-blocking 사용이 어려워서 Reactive Stack이 필요

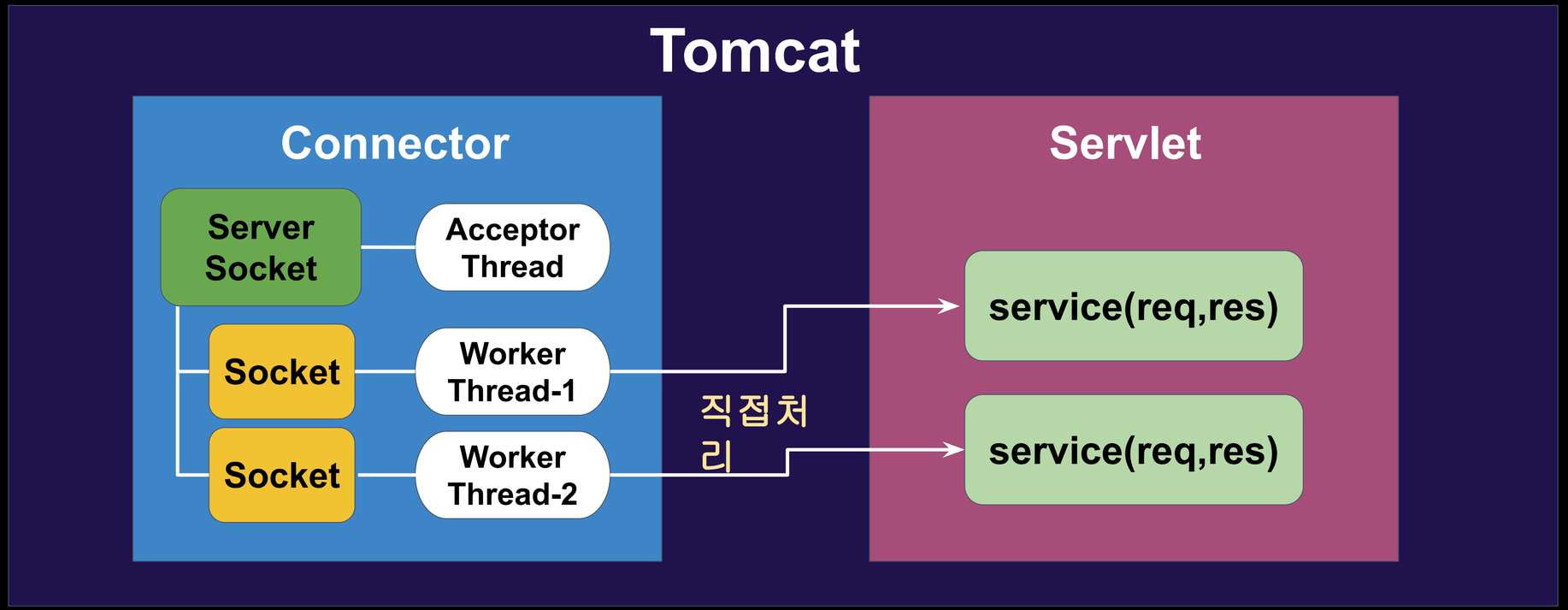
Overview

Why was Spring WebFlux created?

Part of the answer is the need for a non-blocking web stack to handle concurrency with a small number of threads and scale with fewer hardware resources. Servlet non-blocking I/O leads away from the rest of the Servlet API, where contracts are synchronous (`Filter`, `Servlet`) or blocking (`getParameter`, `getPart`). This was the motivation for a new common API to serve as a foundation across any non-blocking runtime. That is important because of servers (such as Netty) that are well-established in the async, non-blocking space.

Tomcat BIO Connector

Connection 요청당 하나의 Worker Thread가 할당 됨
할당된 Worker Thread가 Servlet 처리 수행

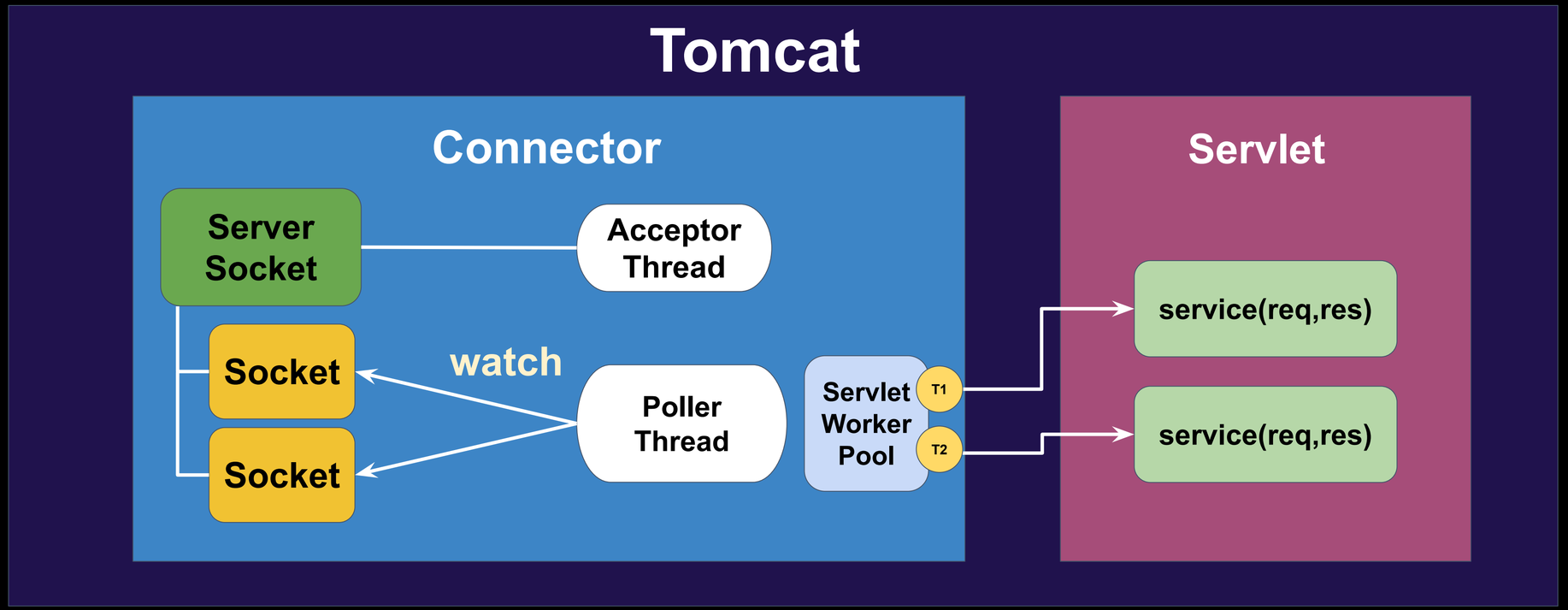


하지만 BIO Connector는 잘 사용되지 않음

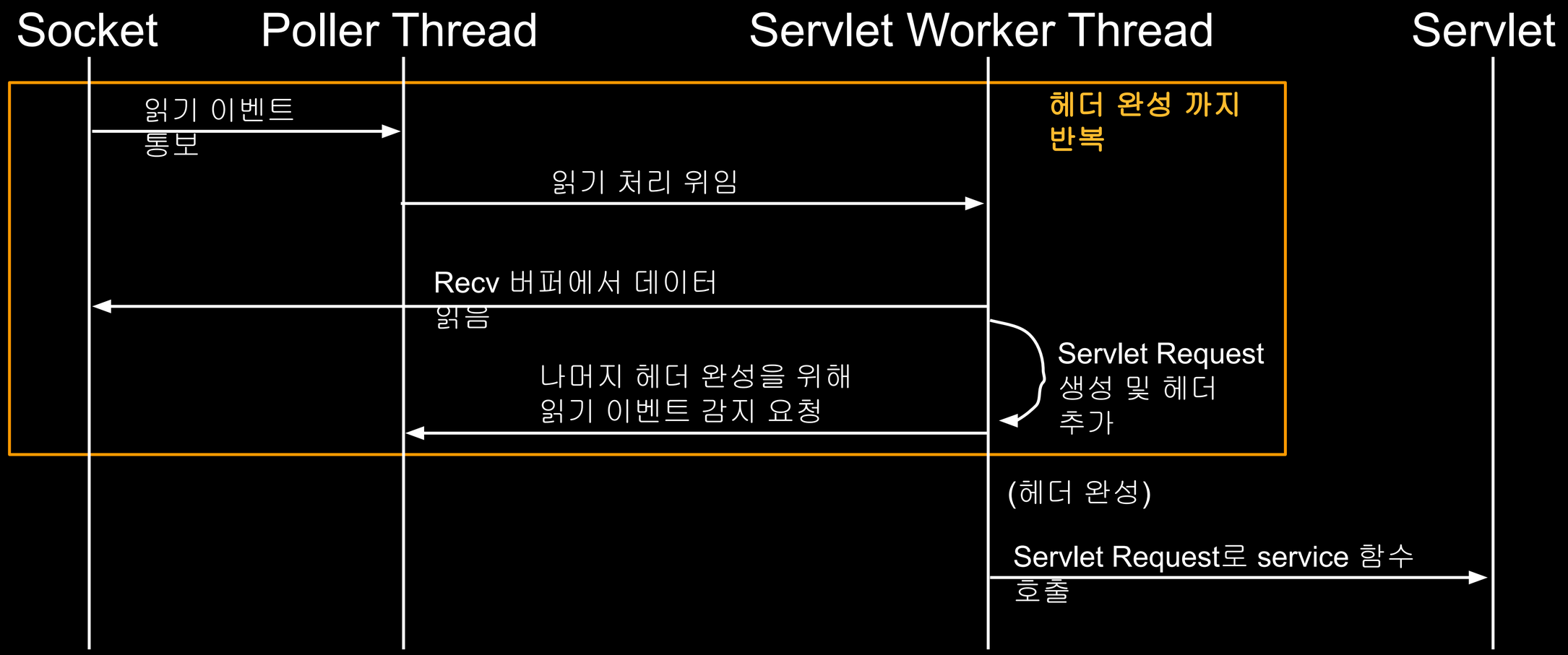
Tomcat 8에서 NIO로 대체됨

Tomcat NIO Connector

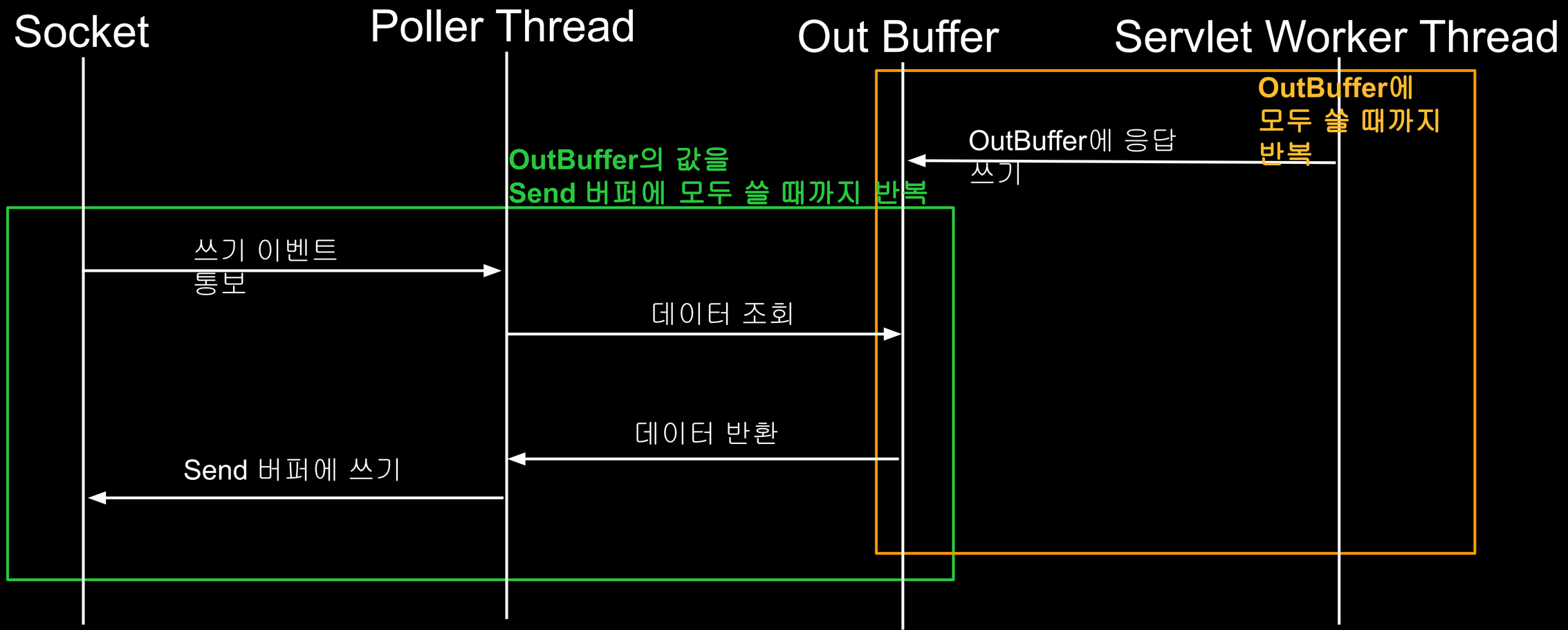
- Tomcat 6버전부터 NIO Connector를 지원
- Network I/O를 하기위한 Thread 자원을 효율적으로 사용할 수 있음



Tomcat NIO Connector - Read

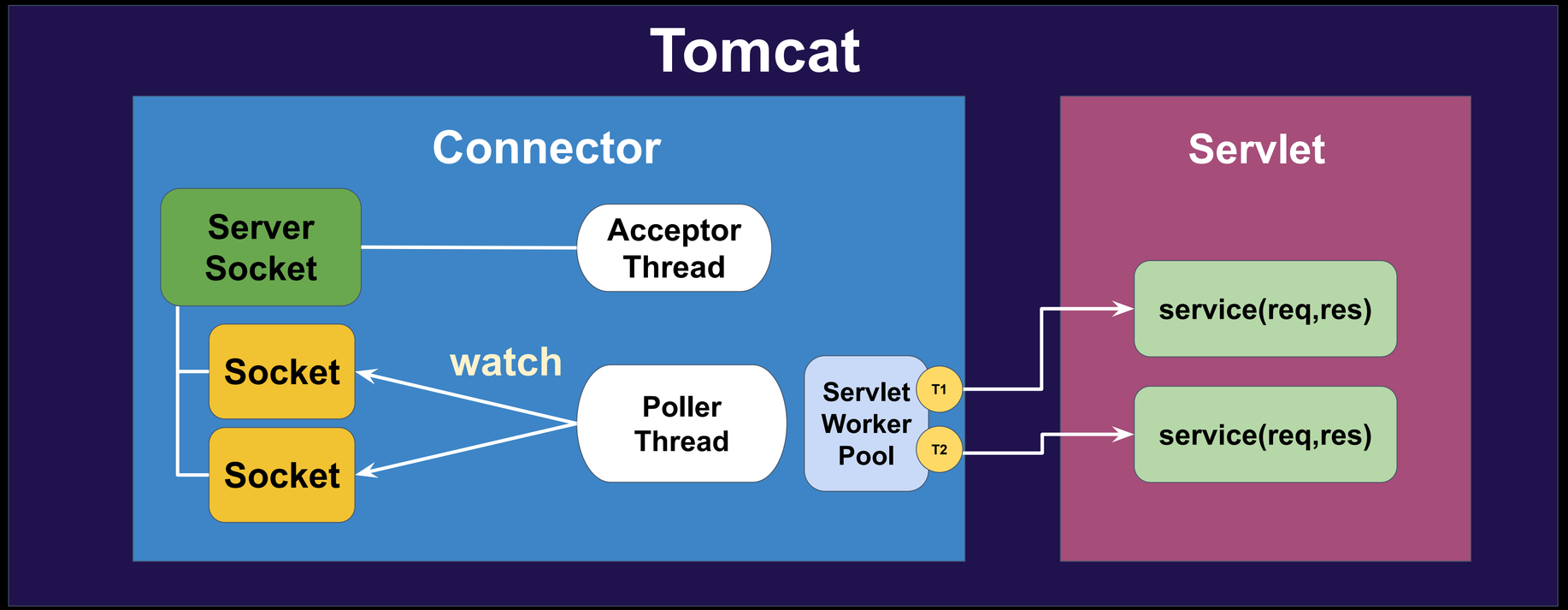


Tomcat NIO Connector - Write



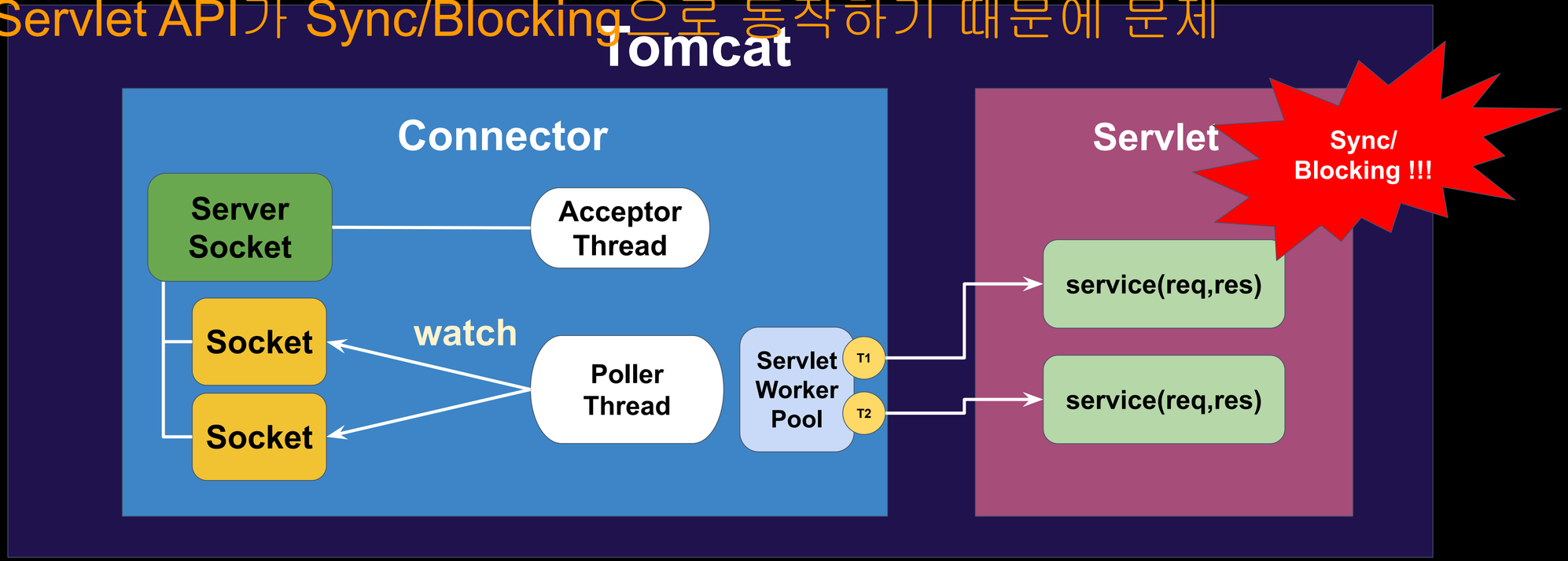
Tomcat NIO Connector

- Tomcat 6버전부터 NIO Connector를 지원
- Network I/O를 하기위한 Thread 자원을 효율적으로 사용할 수 있음



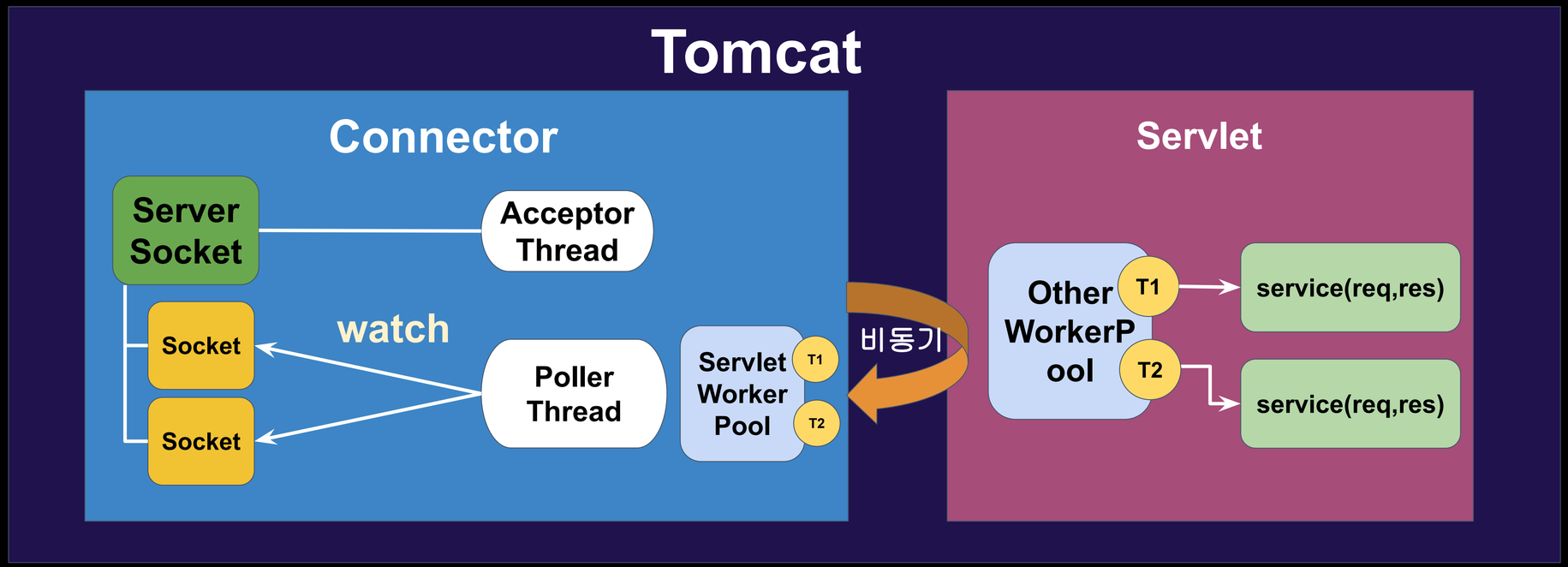
Servlet API 문제

- NIO Connector를 통해 부분적으로 Async/Non-blocking으로Network I/O를 처리해도
  결국 Servlet API가 Sync/Blocking으로 동작하기 때문에 문제



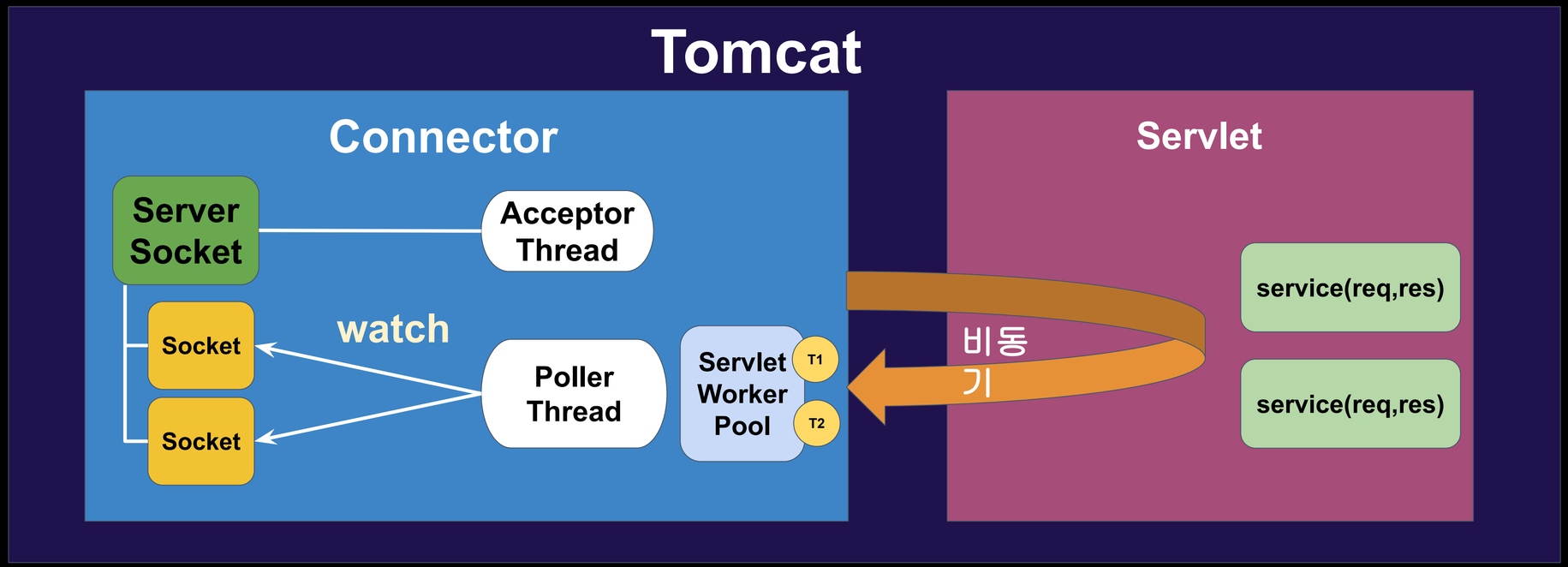
Servlet 3.0 Async API

- Servlet의 service(req, res) 함수 처리를 다른 Thread에게 위임하여 네트워크 I/O를 담당하는 Worker Thread를 더 많이 확보
- 하지만 결국 Servlet의 service 함수 수행을 위한 Thread들이 필요함



Servlet 3.1 Async/Non-blocking I/O

- Servlet의 write/read 기능에서 Async/Non-blocking을 지원
- service 함수 내에서 다른 Blocking이 없다면 Fully Async/Non-blocking 가능

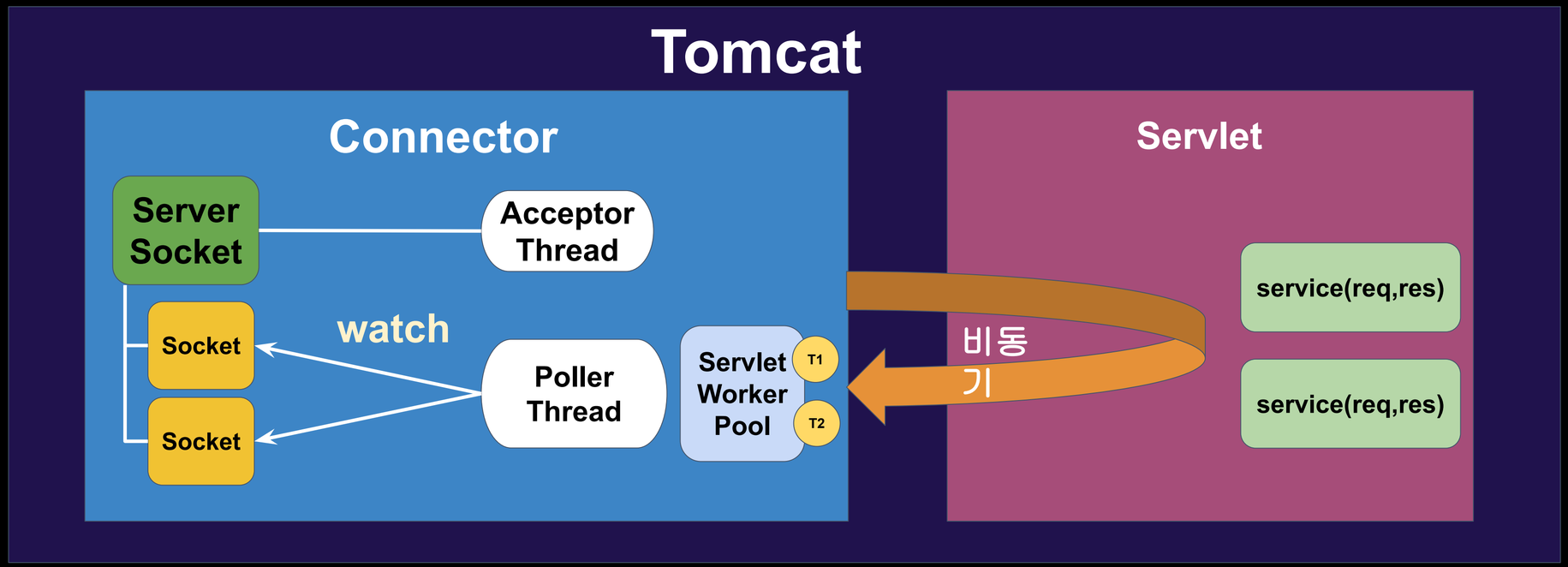


Servlet 3.1 Code Sample

```
final StringBuilder requestBody = new StringBuilder();

// InputStream에 리스너 추가 가능
inputStream.setReadListener(new ReadListener() {

    // 데이터 읽기가 준비 되면 호출됨
    @Override
    public void onDataAvailable() throws IOException {
        final byte[] buffer = new byte[1024];
        int byteRead;

        // inputStream에 데이터가 있을 때까지 읽어서 requestBody에 추가
        while (inputStream.ready() && (byteRead = inputStream.read(buffer)) != -1) {
            requestBody.append(new String(buffer, 0, byteRead, StandardCharsets.UTF_8));
        }
    }
});
```

Servlet 3.1 Async/Non-blocking I/O

- Servlet의 write/read 기능에서 Async/Non-blocking을 지원
- service 함수 내에서 다른 Blocking이 없다면 Fully Async/Non-blocking 가능



문제 해결?

Servlet API Blocking Methods

아직 문제가 존재

일부 Method들은 Async/Non-blocking I/O 기반으로 동작하지 않고
호출 되는 즉시 Sync/Blocking I/O을 수행하여 Thread를 Blocking 상태에 빠트림
- getParameter(), getPart()

결국 Servlet API로는 어려움

Spring Webflux

- Servlet API를 사용하지 않는 Spring Webflux 탄생
- Non-blocking과 잘 동작하는 새로운 API들
- 여러 Async/Non-blocking Server들을 이용할 수 있음

우리가 살펴볼 조합과 버전

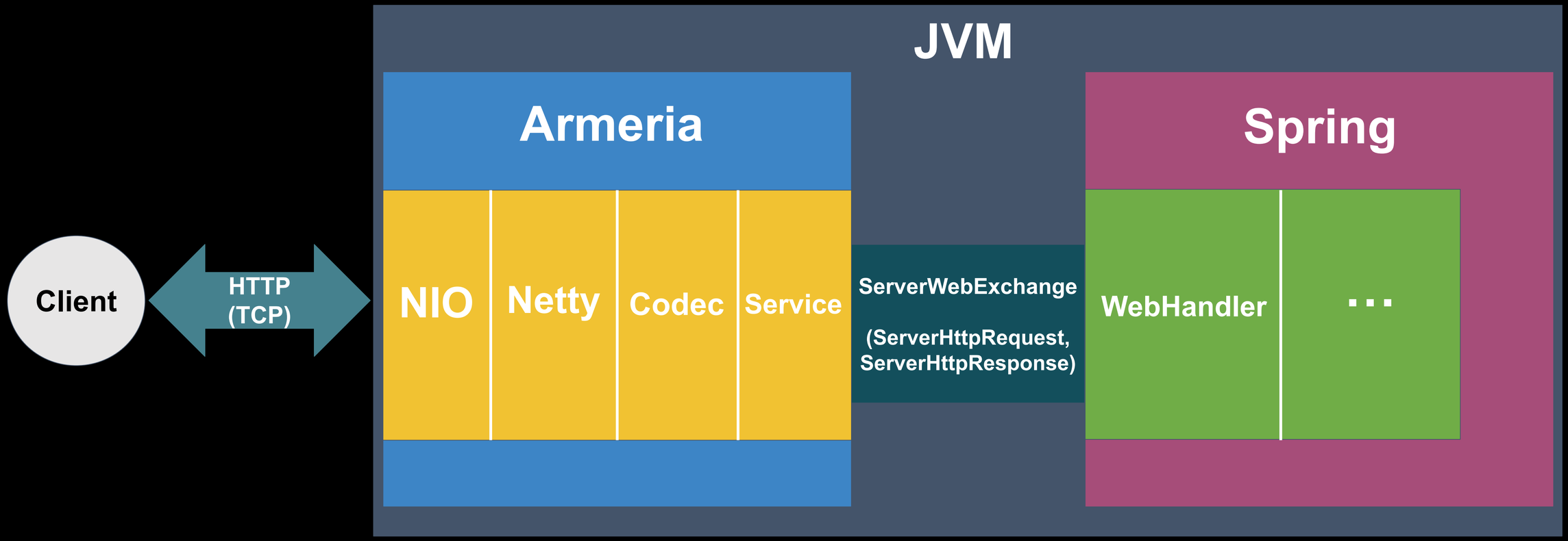
- Spring Boot 3.4.2
  - Spring-Webflux 6.2.2
- Armeria 1.31.3
  - Netty 4.1.117 Final
    - NIO Selector
- HTTP 1.1

대상 Spring Controller

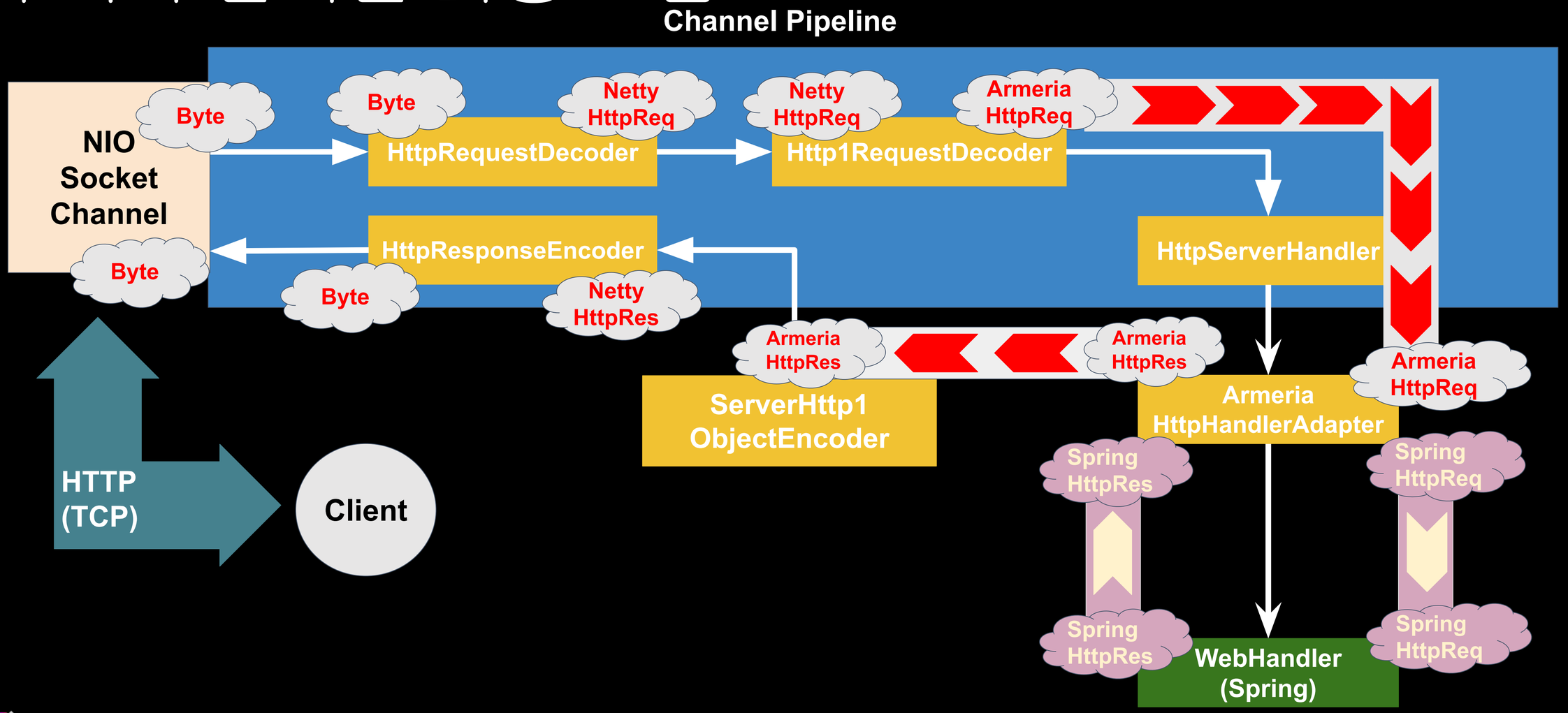
```
// POST http://localhost:8080/test
@PostMapping("/test")
public Mono<String> echo(@RequestBody final Mono<String> requestBody) {
    return requestBody.map(body -> "Echo: " + body);
}
```

Spring Webflux Overview

Spring Webflux Overview



우리가 살펴볼 최종 모습


Channel Pipeline
NIO Socket Channel
Byte
Byte
Netty HttpReq
Netty HttpReq
Armeria HttpReq
HttpRequestDecoder
Http1RequestDecoder
HttpServerHandler
HttpResponseEncoder
Byte
Byte
Netty HttpRes
Armeria HttpRes
Armeria HttpRes
Armeria HttpReq
ServerHttp1 ObjectEncoder
Armeria HttpHandlerAdapter
Spring HttpRes
Spring HttpReq
Spring HttpRes
Spring HttpReq
WebHandler (Spring)
HTTP (TCP)
Client

NIO

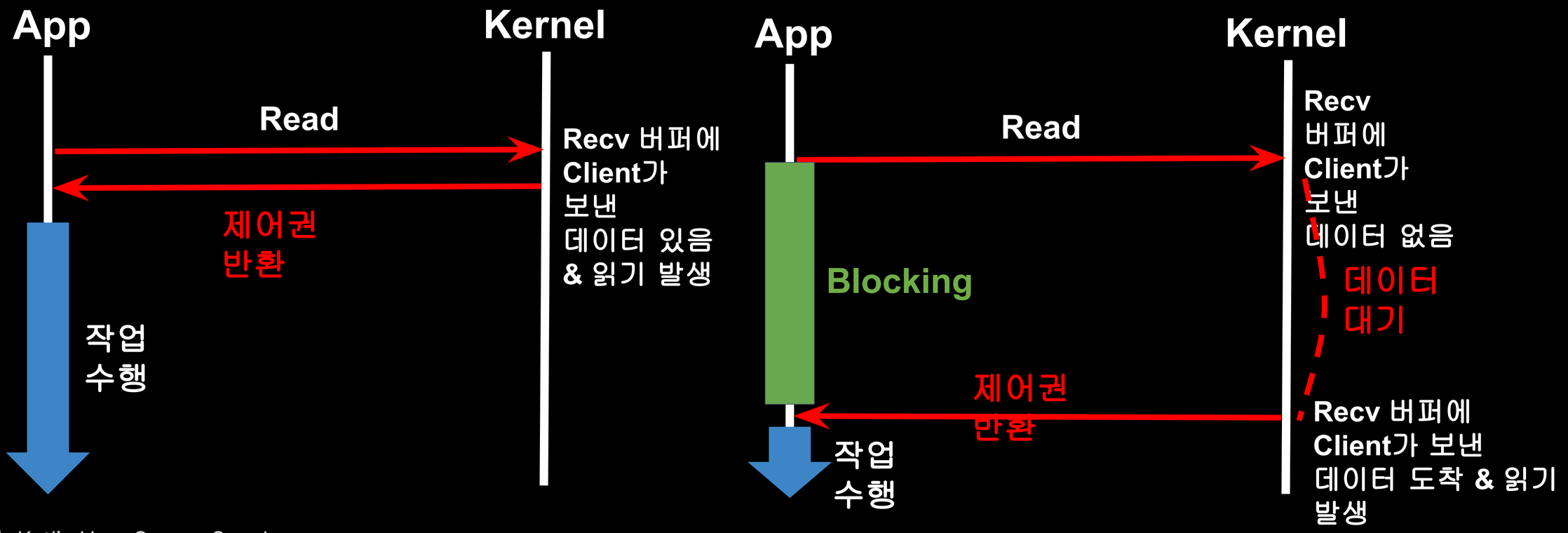
NIO(New I/O)

- Async/Non-blocking을 지원하는 Java I/O API
- Blocking I/O 모델을 극복하기 위해 Java 4에서 추가
- NIO2도 있으나, 이번에는 다루지 않음

OIO는 왜 Blocking이 발생하나?

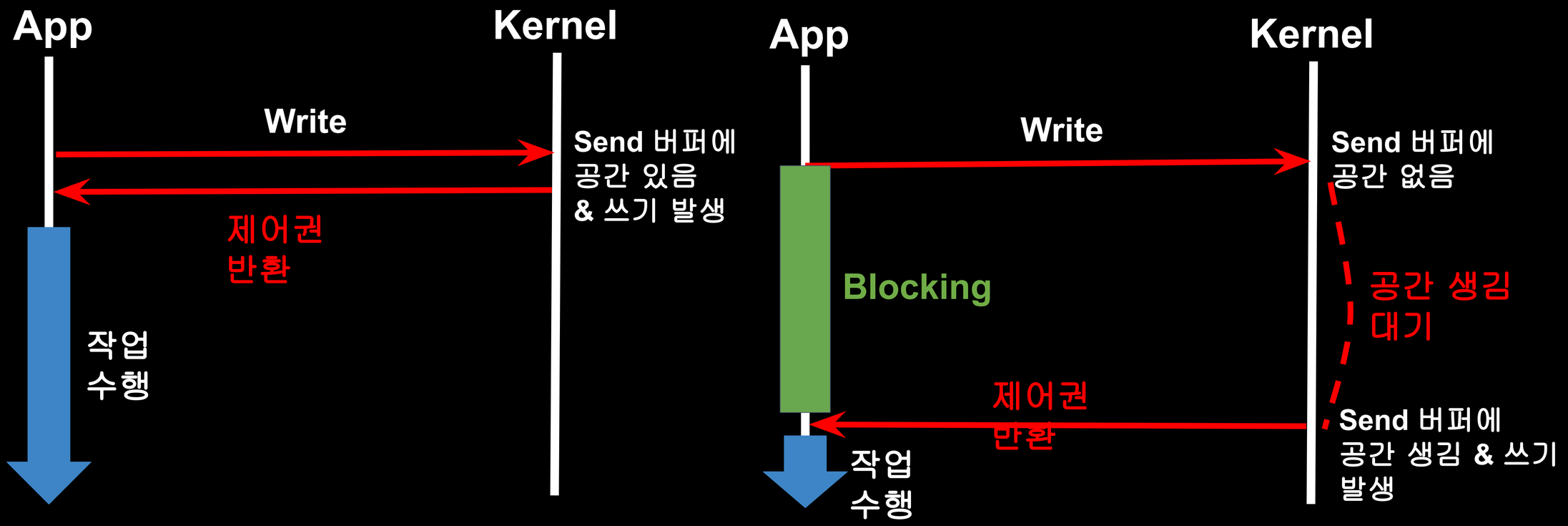
OIO Read

- read(buffer) 호출 시 kernal의 recv buffer에서 데이터 읽기 시도



OIO Write

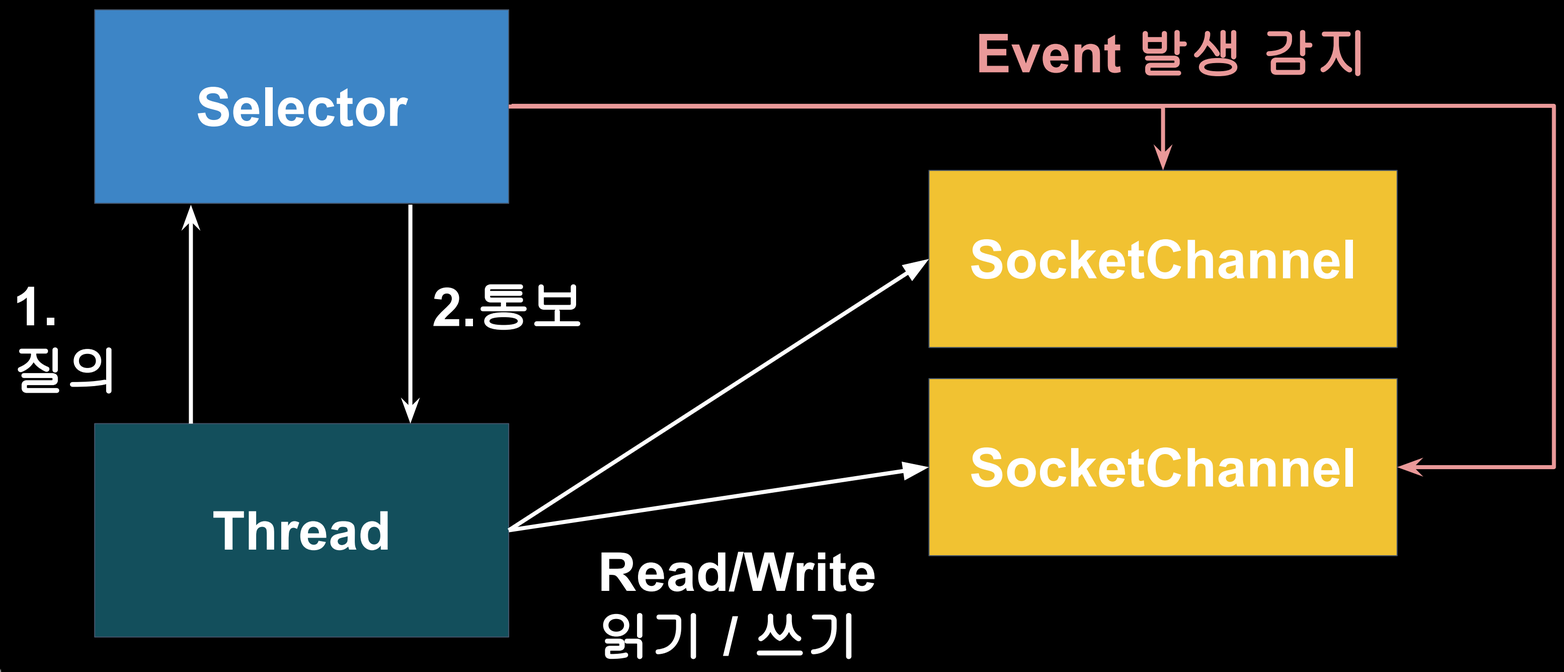
- write(buffer) 호출 시 kernal의 send buffer에 데이터 쓰기 시도



Buffer 상태를 알면 Blocking을 피할 수 있다

NIO Overview

- Selector를 통해 SocketChannel의 상태를 감지
- 연산이 준비 되었다는 Event를 감지하고 알림



NIO Selector

- SocketChannel의 Event를 감지
  - CONNECT, ACCEPT, READ, WRITE
  - 한 번에 복수개의 SocketChannel의 여러 종류 Event 감지 가능
- SocketChannel들에 Event가 발생하면 그 SocketChannel들과 발생한 Event 종류를 반환
- **READ/WRITE가 준비 되었을 때만 I/O를 수행하므로 Blocking이 발생하지 않음**

NIO Example

```
// 서버가 종료될 때까지 반복
while (true) {
    // SocketChannel에서 이벤트가 발생할 때까지 Blocking
    final Set<SelectionKey> selectedKeys = selector.selectedKeys();
    final Iterator<SelectedKey> iterator = selectedKeys.iterator();

    while (iterator.hasNext()) {
        final SelectionKey key = iterator.next();

        // READ 이벤트 준비 여부 확인
        if (key.isReadable()) {
            // READ 이벤트 준비됨!!!!
            // 읽기 작업 수행!
        }

        // ... 기타 이벤트 확인 및 처리
    }
}
```

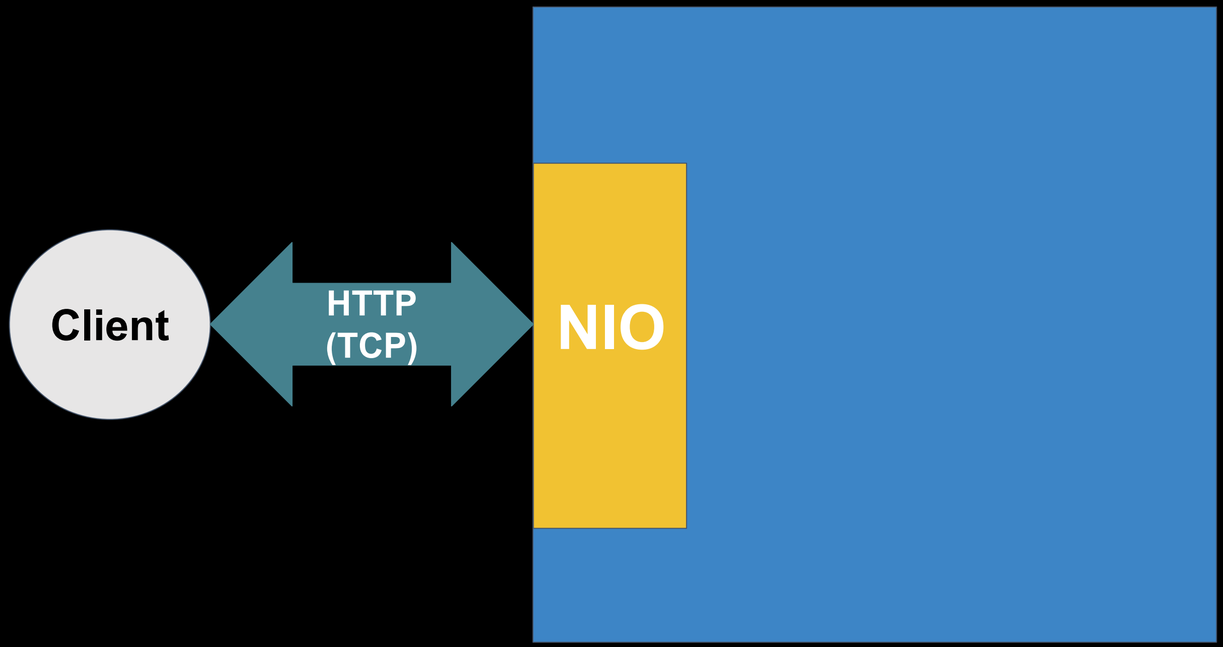
NIO Worker Size

- 병렬성을 올리기 위해 CPU 논리 Core 개수에 비례해서 Event 처리 Thread 생성
  - Netty의 경우 CPU 논리 Core 개수 X 2

NIO ETC

- NIO에는 다른 중요한 개념들도 많으나 오늘은 생략
- 추후에 찾아보는 것들을 추천

NIO 마무리



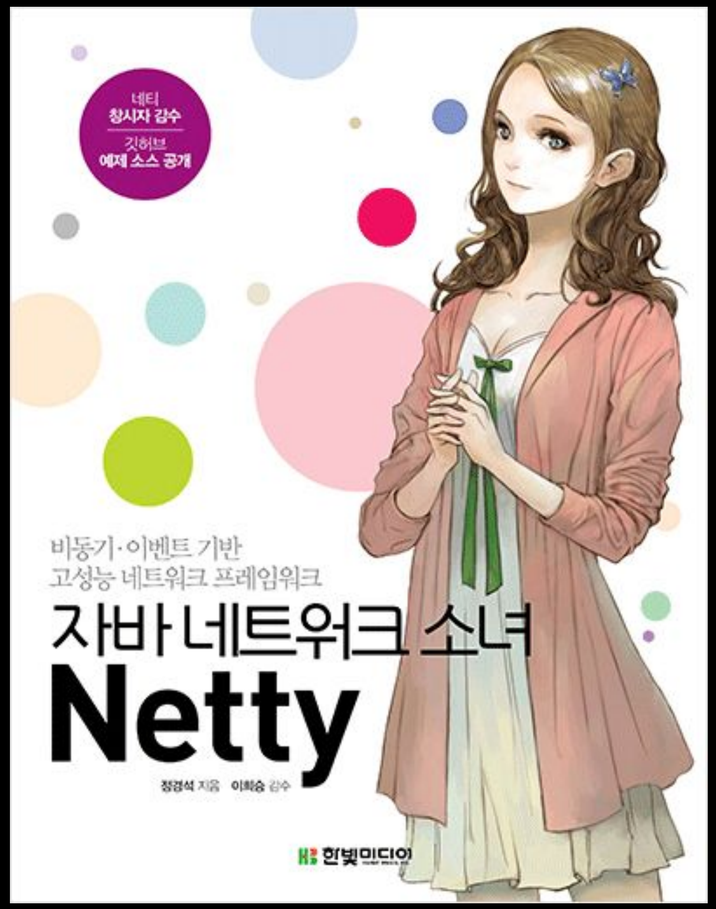
Netty

Netty

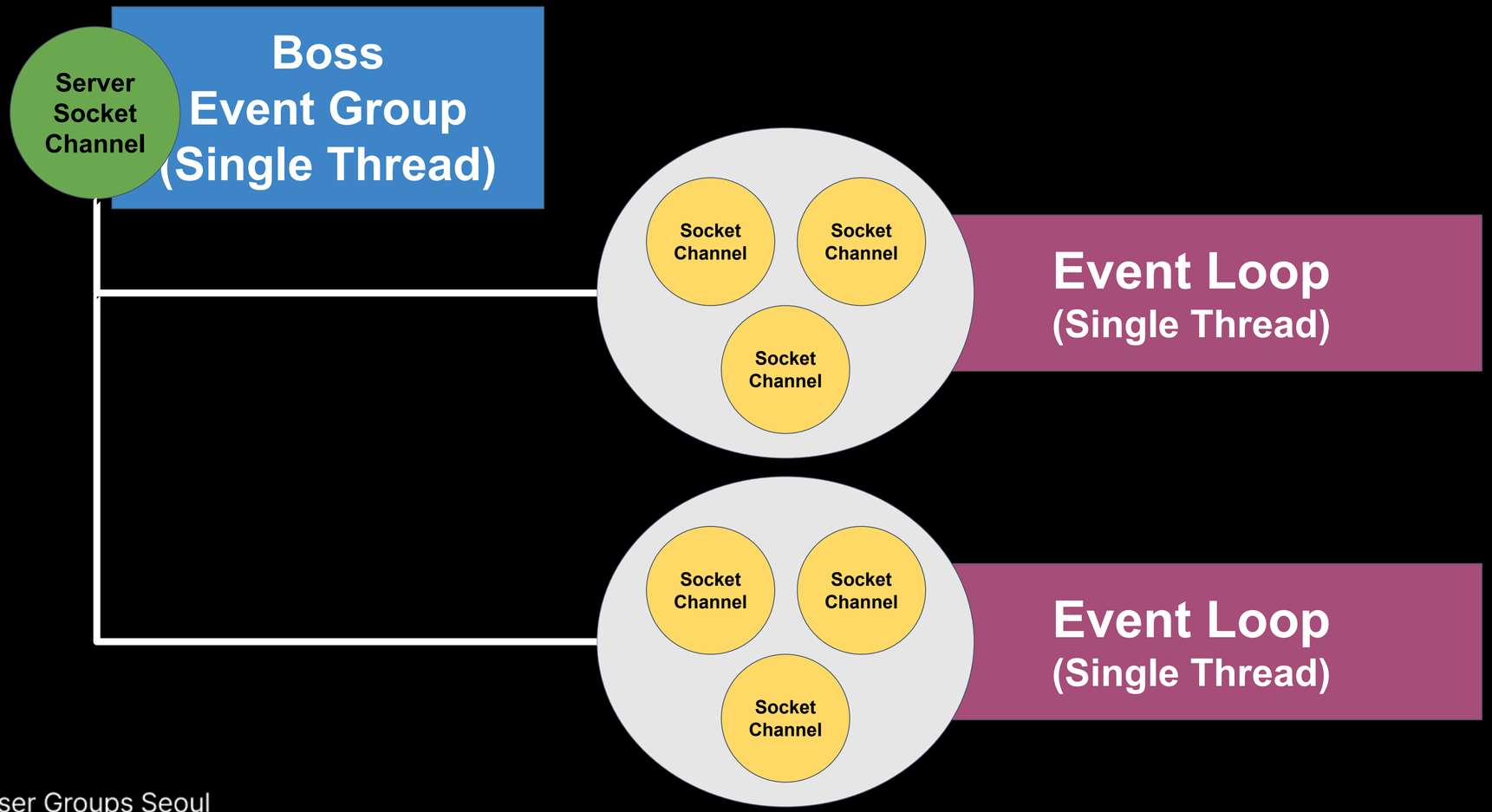
- Async/Non-blocking을 지원하는 Java Network Framework
- Java 진영의 많은 프로젝트들이 사용
- 많은 Network I/O 기술 & 프로토콜을 지원
- Line의 이희승님께서 창시

Netty

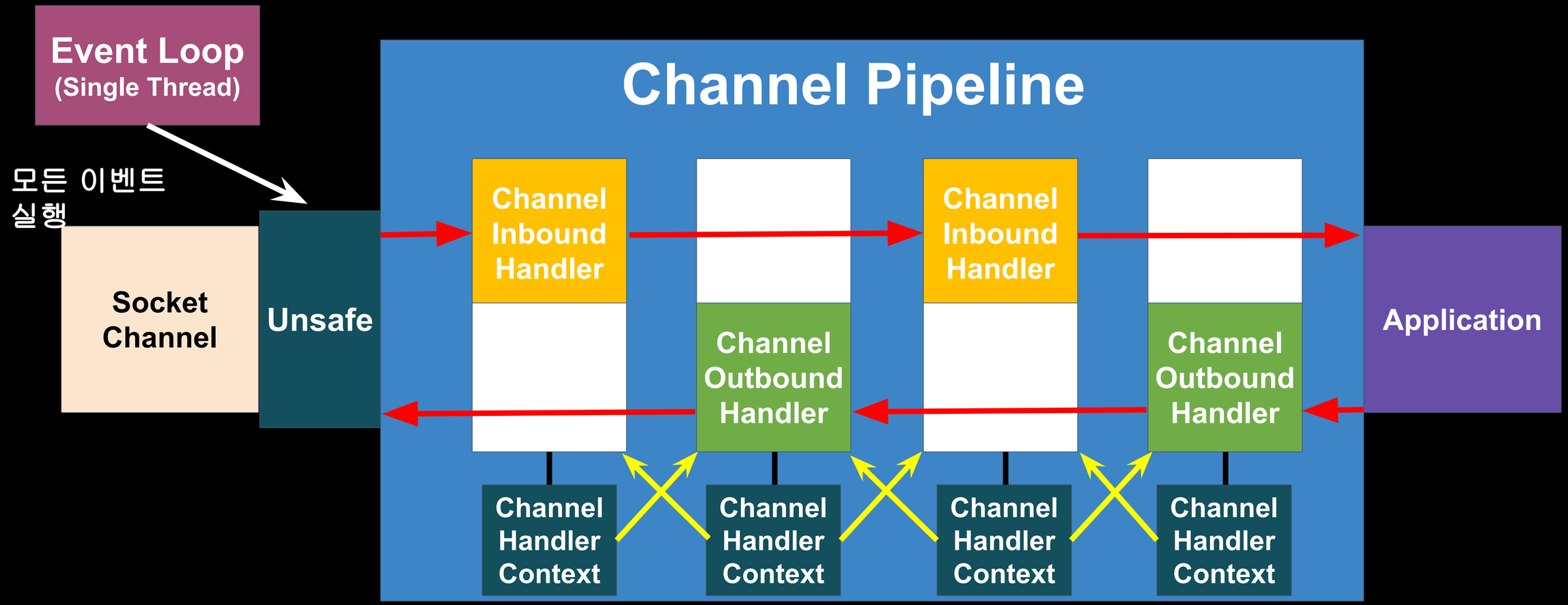
- 전설적인 책 표지 & 브로마이드



Netty Overview - Event Loop



Netty Overview - Channel

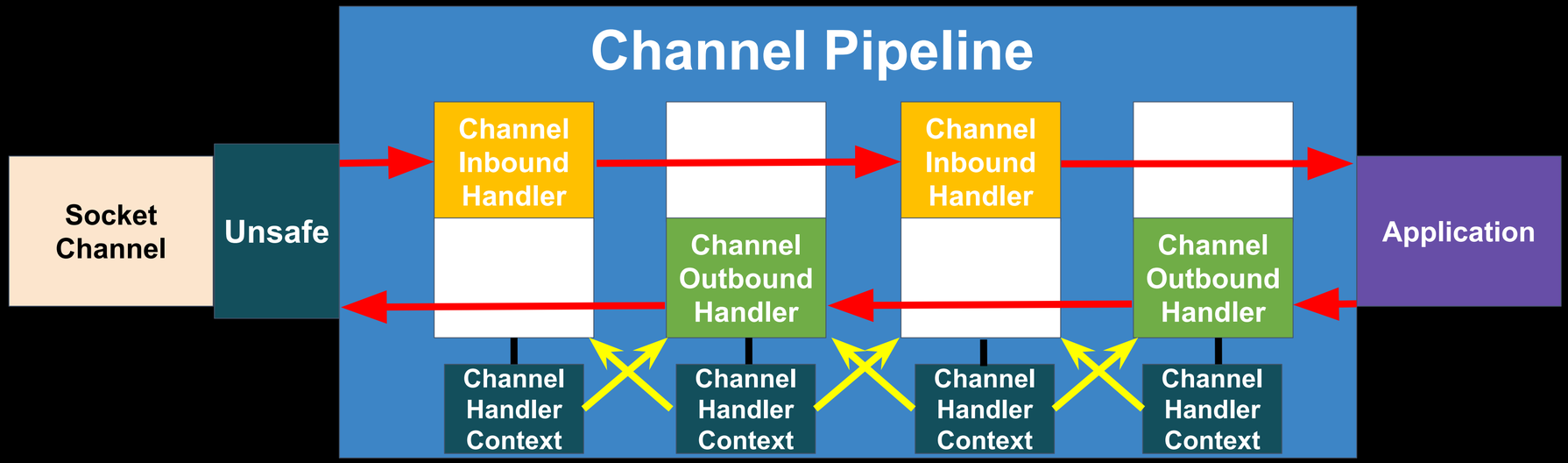


Netty SocketChannel

- ServerSocketChannel이 Client의 CONNECT 요청을 ACCEPT 하면 생성됨
- 우리 예제의 경우 NIOSocketChannel이 생성됨
- Network Read/Write가 NIOSocketChannel에 의해 발생함
  - NIO를 이용

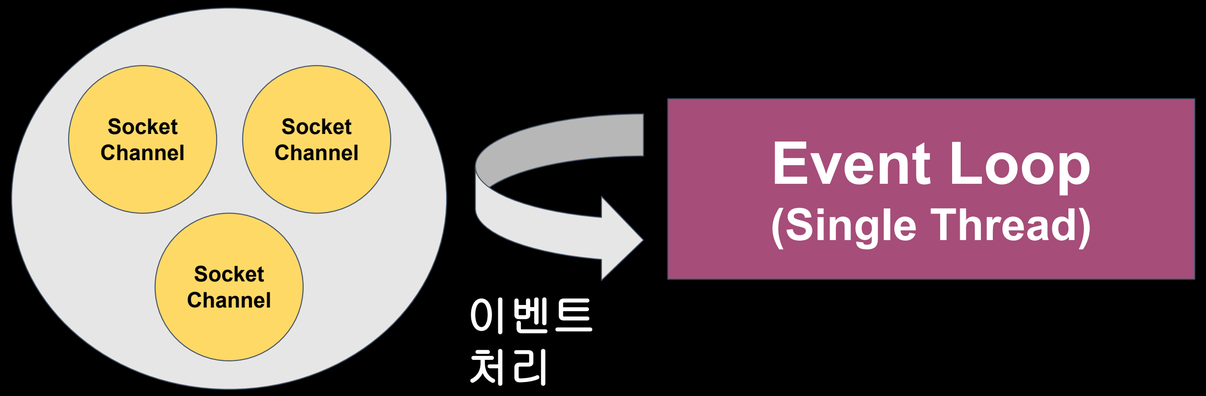
Netty ChannelPipeline

- Channel이 생성될 때 자신만의 ChannelPipeline을 생성
- Channel과 생명주기 함께함
- 처음 생성 시 Head/Tail을 기본으로 등록
- Head는 Unsafe와 협력하여 Network I/O를 처리하는 특별한 ChannelHandler

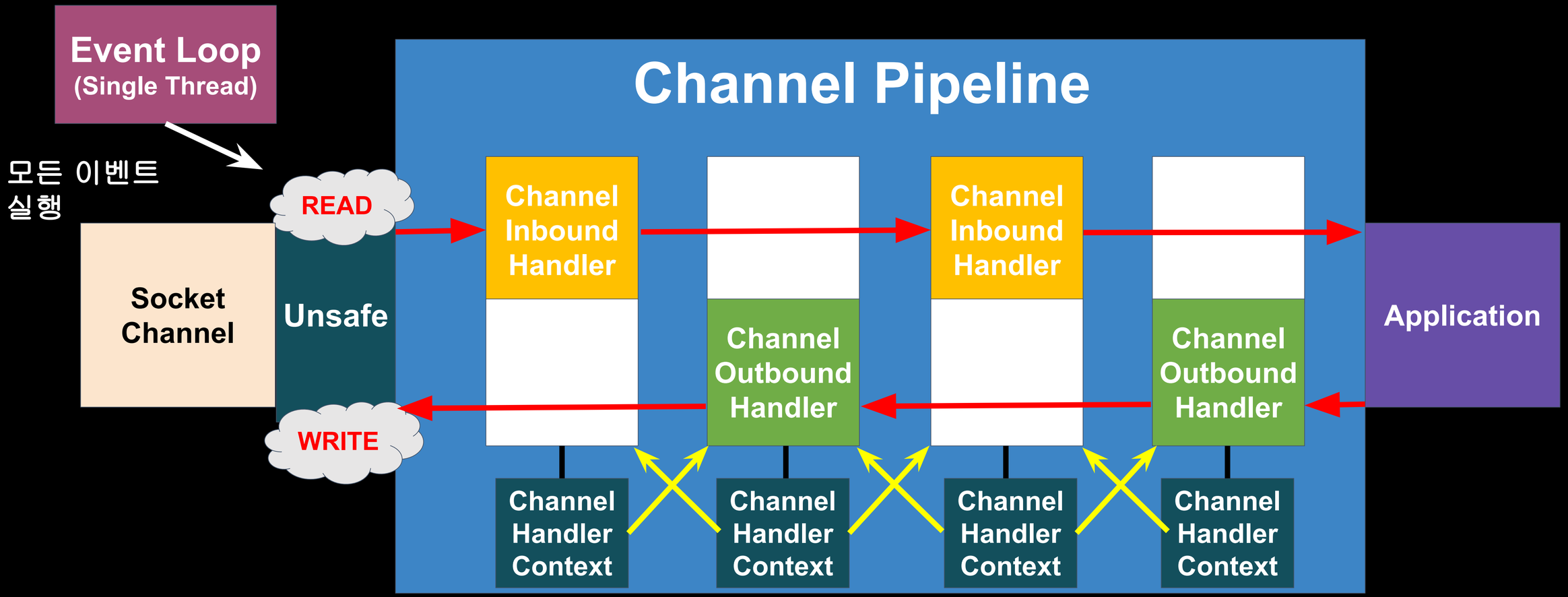


Netty EventLoop

- EventLoop는 자신에게 할당된 N개의 Channel들의 이벤트를 처리
- Single Thread로 이뤄져 있기 때문에 경합 상태 없음
- 우리의 경우는 NIOEventLoop
- NIOEventLoop는 NIO Selector에서 감지한 READ/WRITE 이벤트도 처리

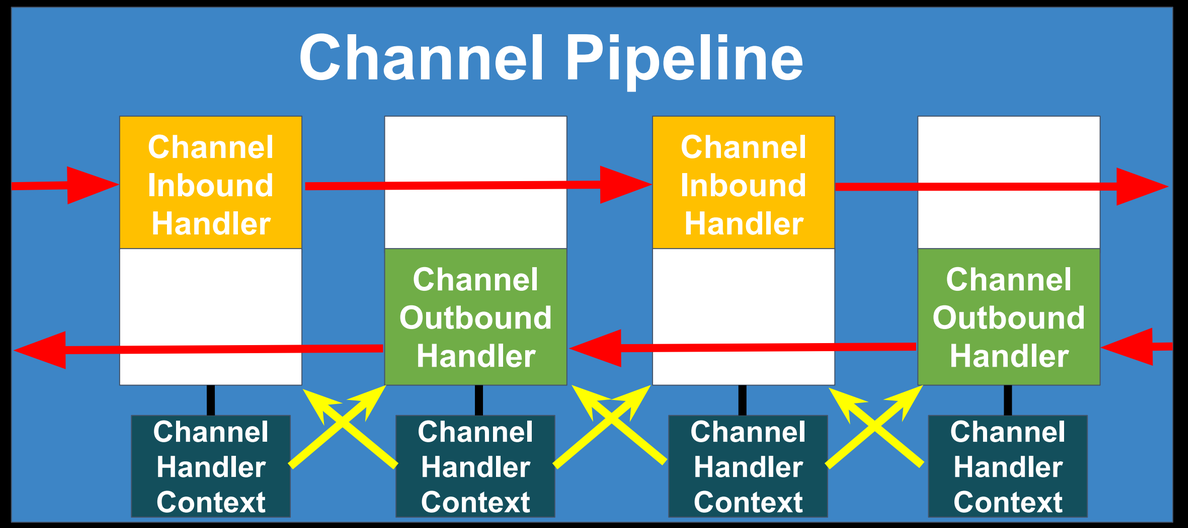


Netty EventLoop + Channel

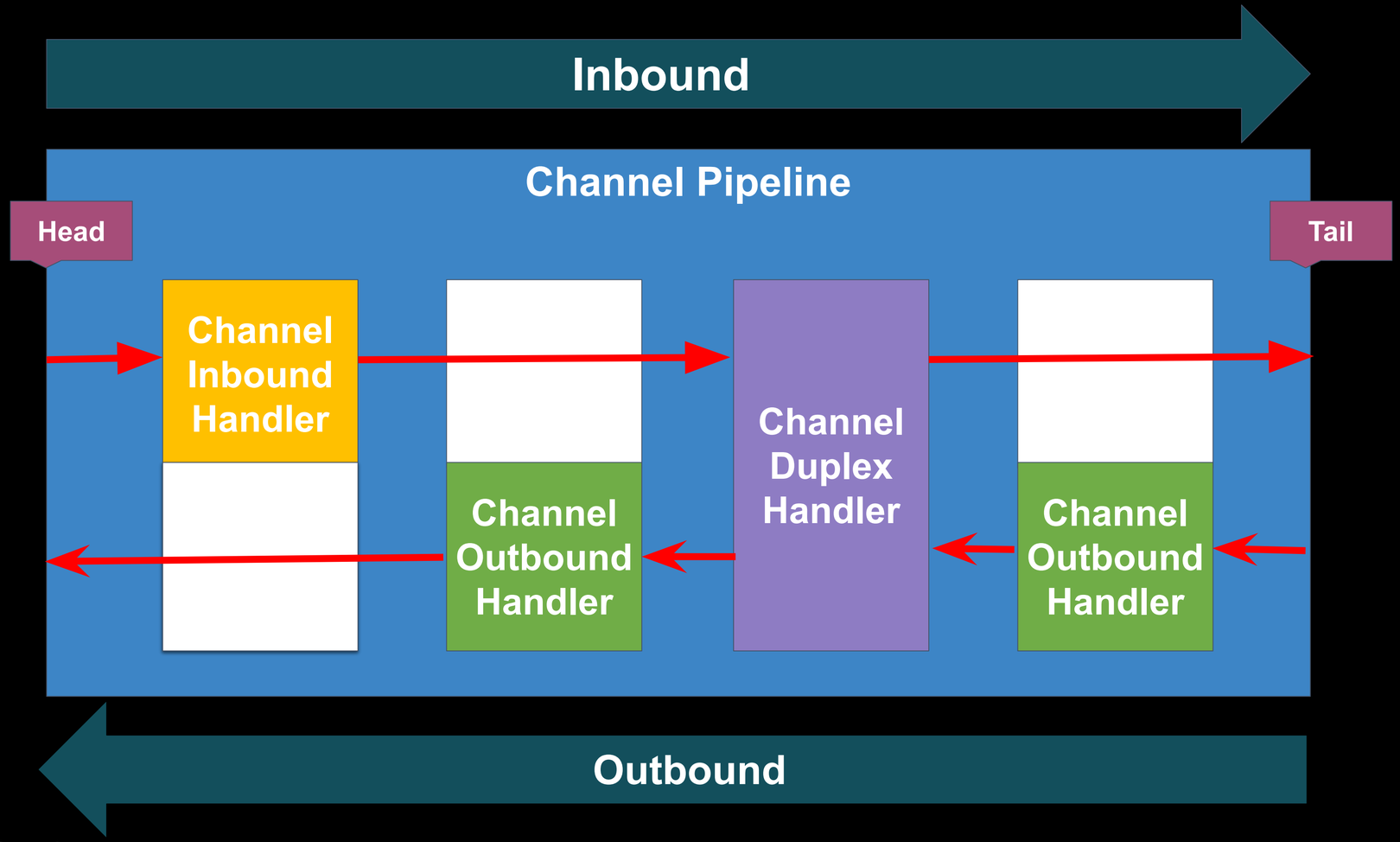


Netty ChannelHandler - 1

- ChannelHandler는 Decorator Pattern으로 이뤄진 Event Handler
- Event의 방향에 따라 논리적으로 Inbound/Outbound로 나뉨
- 런타임에 동적으로 추가/제거 가능
- 통상적으로 각각의 Channel을 위해 객체가 매번 생성 되어 등록됨

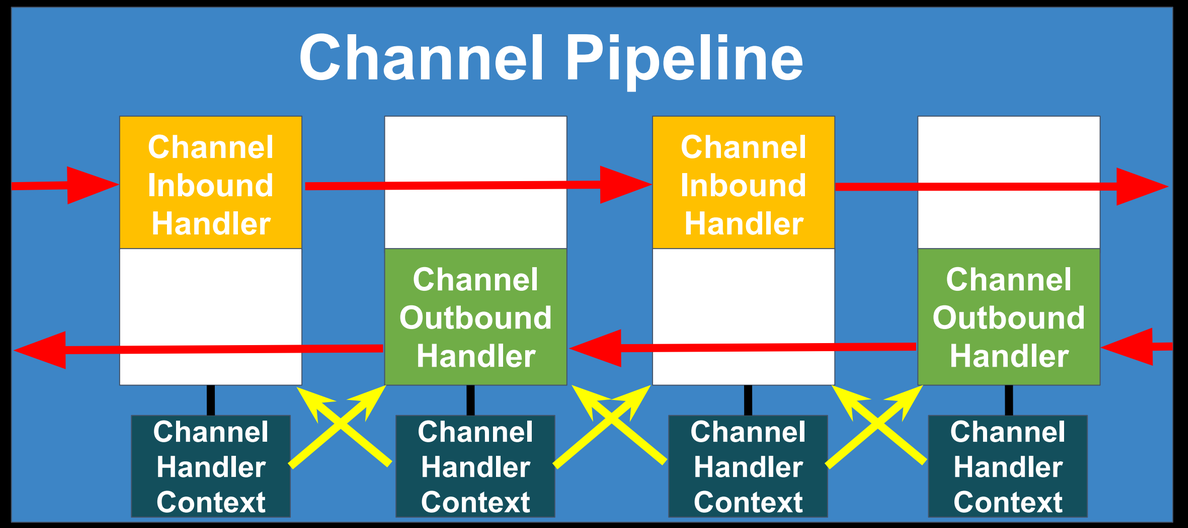


Netty Event 방향 & ChannelHandler



Netty ChannelHandler

- ChannelHandler는 Decorator Pattern으로 이뤄진 Event Handler
- Event의 방향에 따라 논리적으로 Inbound/Outbound로 나뉨
- 각 ChannelHandler마다 자신의 ChannelHandlerContext 존재
- 런타임에 동적으로 추가/제거 가능
- 통상적으로 각각의 Channel을 위해 객체가 매번 생성 되어 등록됨



Netty ChannelHandler - 2

```
/*
    LoggingHandler.java
    SocketChannel에서 읽은 msg를 로그에 남기는 ChannelInboundHandler
*/
@Override
public void channelRead(final ChannelHandlerContext ctx, final Object msg) throws Exception {
    // 로깅 수행
    logger.log(internalLevel, format(ctx, "READ", msg));

    // 다음 ChannelInboundHandler로 msg 전파
    // 만약 호출 하지 않는다면 이벤트 전파 중단
    ctx.fireChannelRead(msg);
}
```

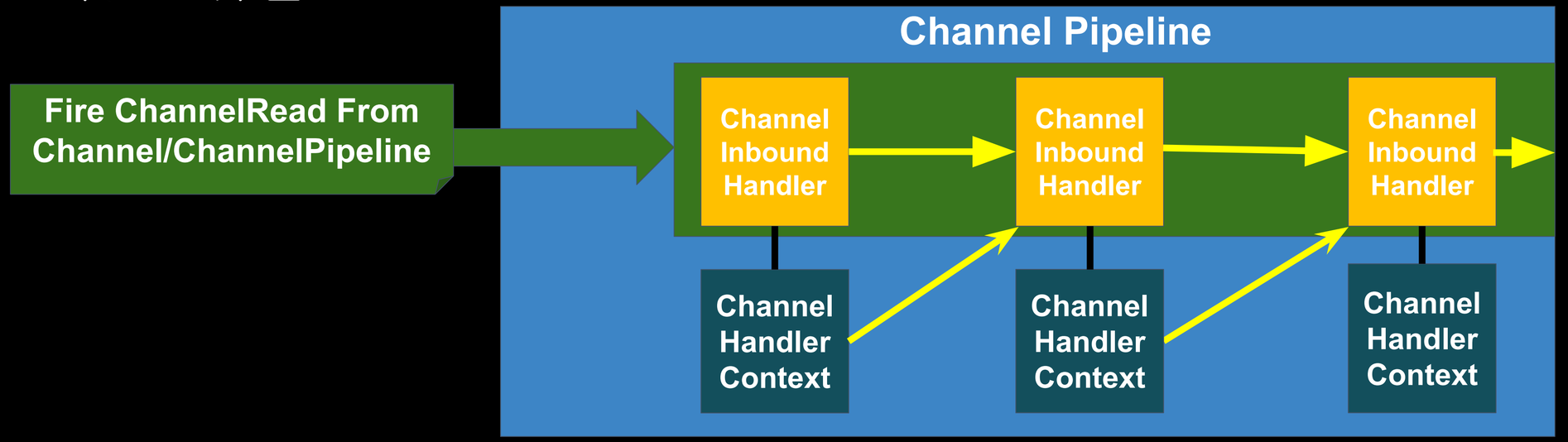
Netty Event 종류

- Channel과 관련된 다양한 이벤트 존재
- 사용자 정의 이벤트도 존재

이벤트 이름	방향	의미
channelRead	Inbound	메시지 읽음
channelReadComplete	Inbound	메시지 읽기 완료 현재 전송된 데이터 기준으로 완료
write	Outbound	버퍼에 메시지 쓰기
flush	Outbound	버퍼에 써진 메시지를 방출 이 시점에 Network Write가 발생

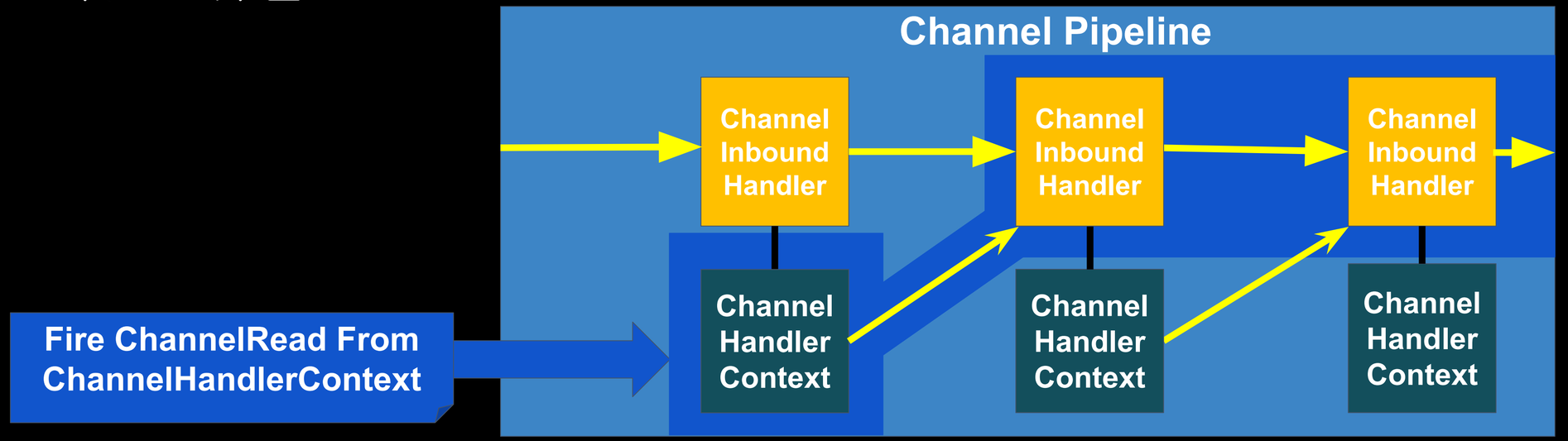
Netty Event Fire

- Channel/ChannelPipeline과 ChannelHandlerContext에 차이점 존재
- Channel/ChannelPipeline은 항상 HEAD/TAIL로부터 이벤트 유발
- ChannelHandlerContext는 현재 ChannelHandler 기준 이전/다음으로 이벤트유발



Netty Event Fire

- Channel/ChannelPipeline과 ChannelHandlerContext에 차이점 존재
- Channel/ChannelPipeline은 항상 HEAD/TAIL로부터 이벤트 유발
- ChannelHandlerContext는 현재 ChannelHandler 기준 이전/다음으로 이벤트유발

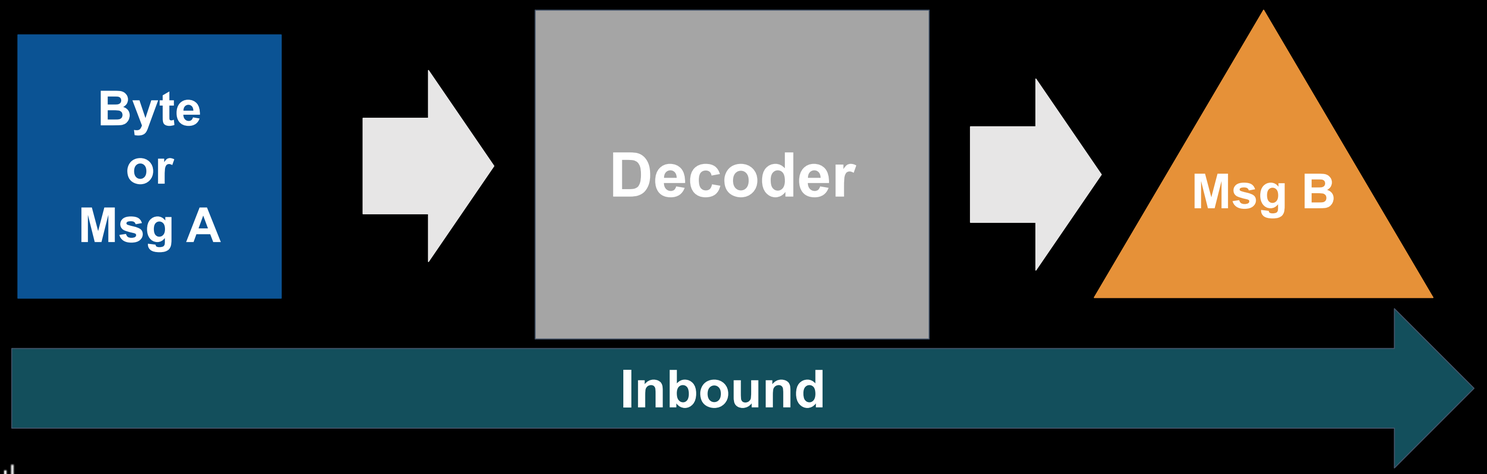


Netty Decoder/Encoder

- 특별한 ChannelHandler
- SocketChannel은 Byte만 읽고 쓸 수 있음
- 하지만 순수한 Byte 데이터는 프로그래밍 논리를 구현하기 어려움
- Byte 데이터를 적절한 책임을 가지는 클래스의 객체로 변환 필요

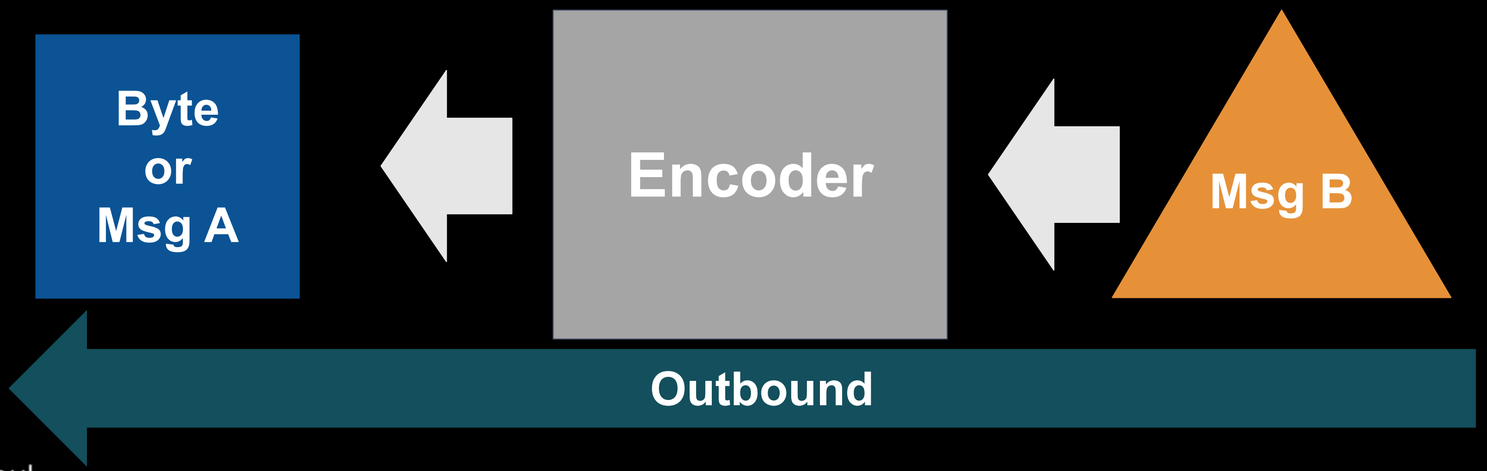
Netty Decoder

- Decoder는 Inbound 이벤트 처리에 사용
- Byte → Msg
  - 주로 Channel에서 읽은 데이터를 변환
- Msg A → Msg B
  - Msg는 다른 타입 Msg의 Source일 수 있음

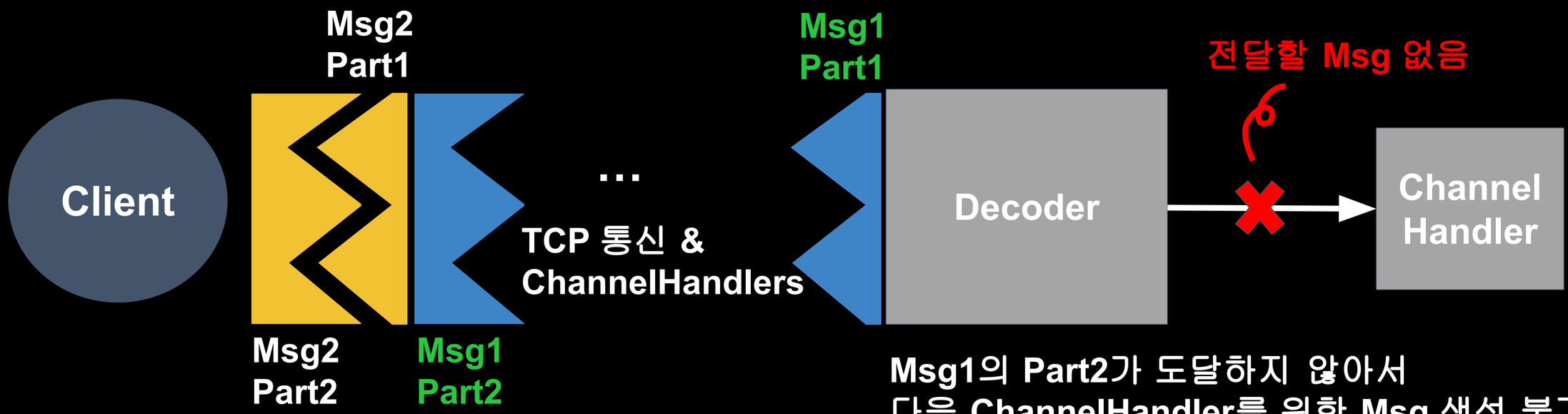


Netty Encoder

- Encoder는 Outbound 이벤트 처리에 사용
- Msg → Byte
  - 주로 Channel에 쓸 데이터를 변환
- Msg B → Msg A
  - Msg는 다른 타입 Msg의 Source일 수 있음



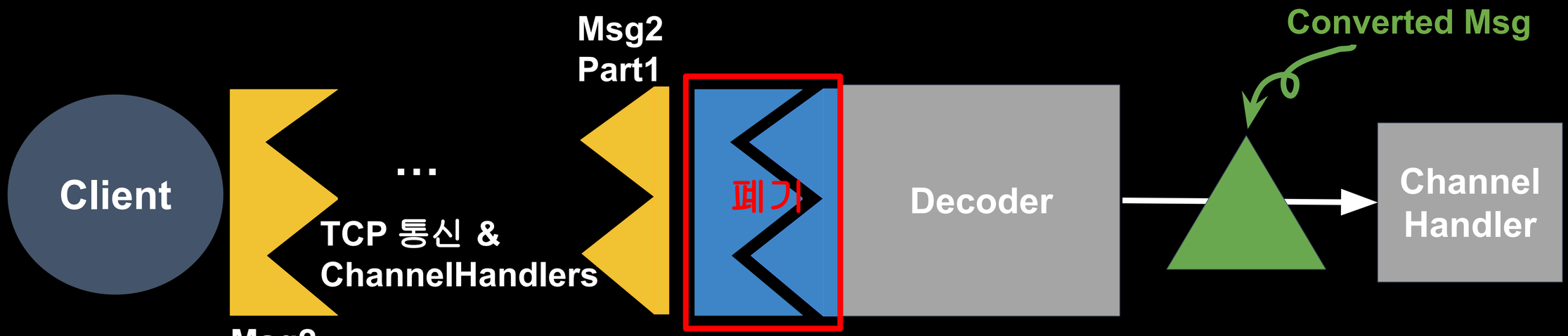
Netty Decoder 데이터 축적1



Msg1의 Part2가 도달하지 않아서
다음 ChannelHandler를 위한 Msg 생성 불가

Msg1 Part1을 Decoder의 멤버변수로 저장 해둠

Netty Decoder 데이터 축적2



Msg1의 Part2가 도착하면 Msg를 완성하여 ConvertedMsg를 만들고
다음 ChannelHandler에게 전달

ConvertedMsg를 만드는데 사용된 Msg1은 폐기
아직 ConvertedMsg를 만들 수 없는 Msg2의 Part1은 변수에 저장

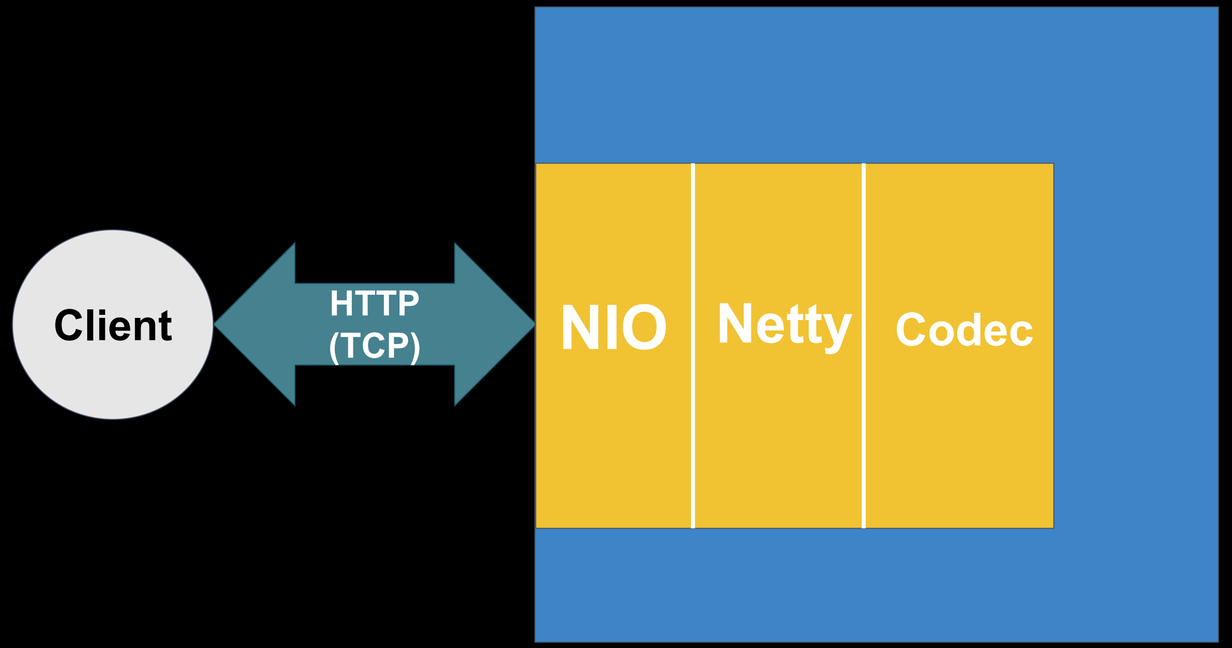
Netty EventLoop Execution

- 모든 이벤트는 EventLoop의 Thread에서 수행됨
- 이벤트 처리 함수 내부의 코드를 통해 통제

```
if (executor.inEventLoop()) {
    next.invokeChannelRead(msg);
} else {
    executor.execute(new Runnable() {
        @Override
        public void run() {
            next.invokeChannelRead(msg);
        }
    });
}
```

```
public boolean inEventLoop(Thread thread) {
    // EventLoop와 동일 Thread이면 TRUE
    return thread == this.thread;
}
```

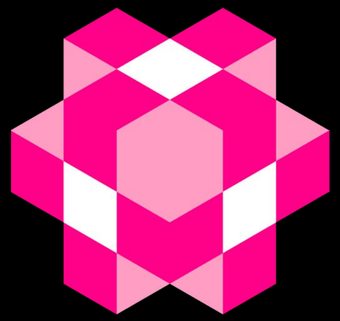
Netty 아우리



Armeria

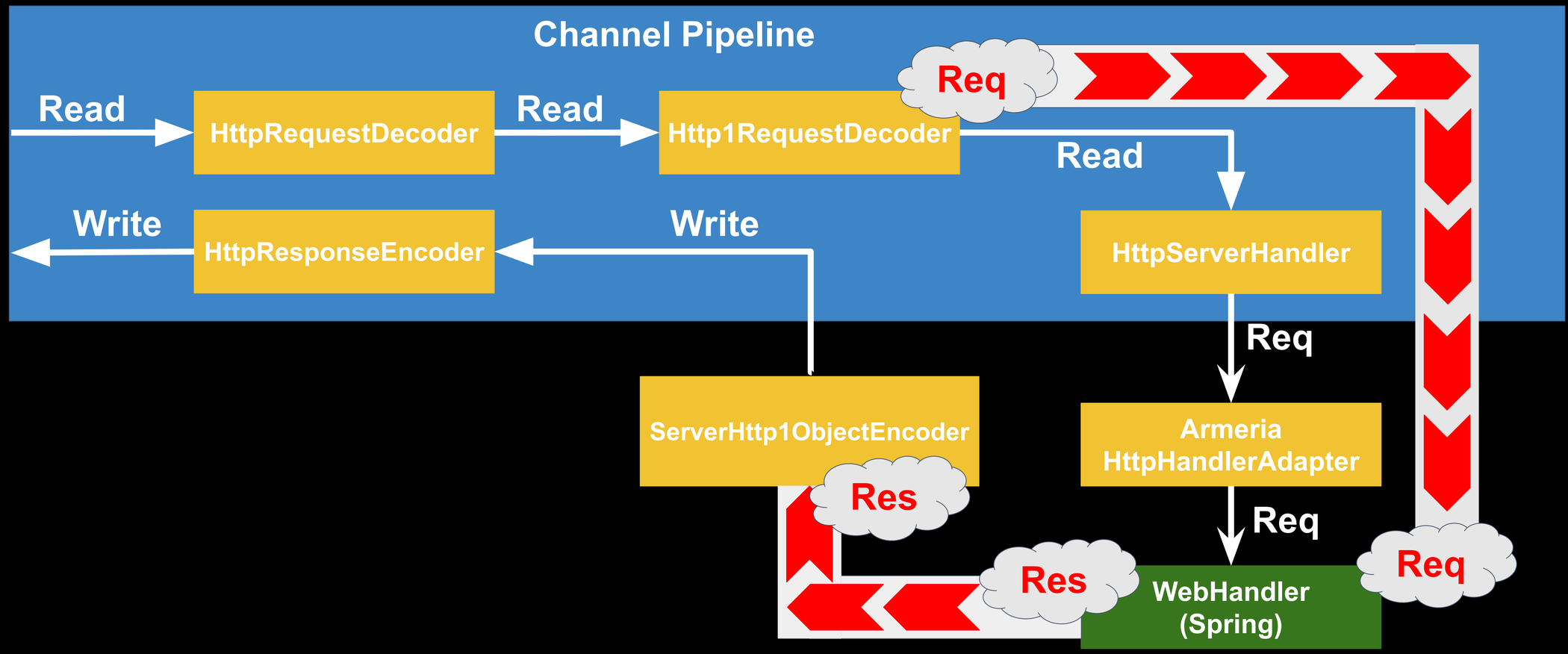
Armeria 소개

- Line의 오픈소스
- Reactive Streams를 지원하는 Async/Non-blocking 서버
- 단일 포트로 멀티 프로토콜 지원
- 다양한 프로토콜과 기능 및 Spring 연동 제공
- 이희승님이 리딩



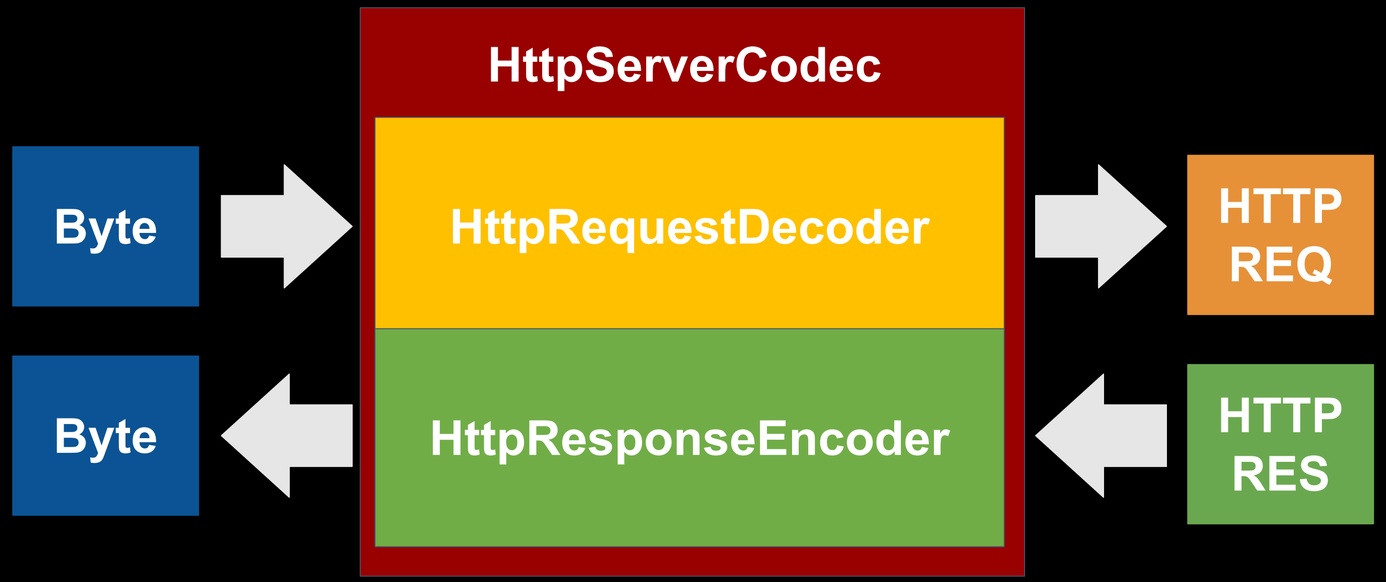
Armeria

Armeria Overview for Spring Webflux



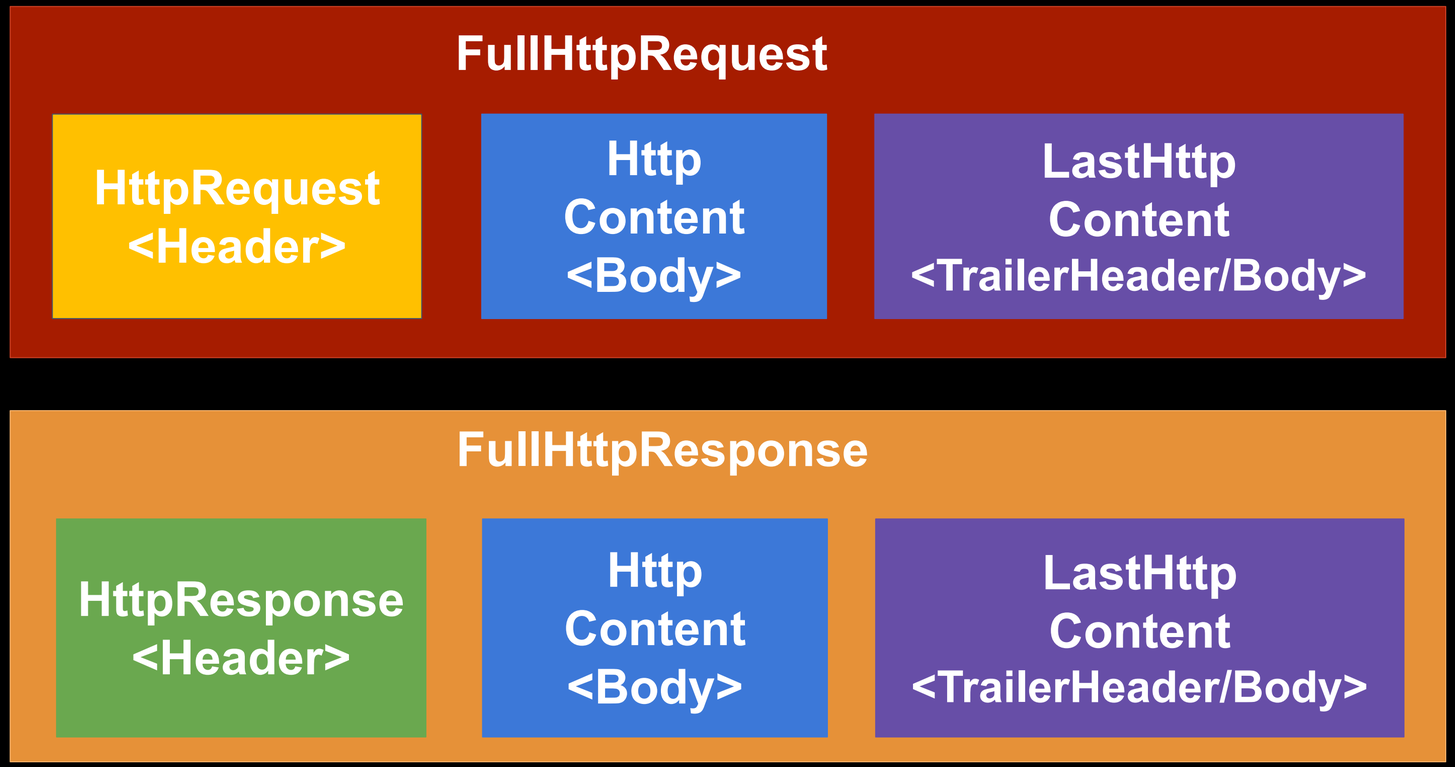
Armeria HttpServerCodec

- Netty의 HttpRequestDecoder/HttpResponseEncoder의 확장 이용
- Byte와 Netty의 HTTP 관련 객체 사이에서 Decoding/Encoding 수행



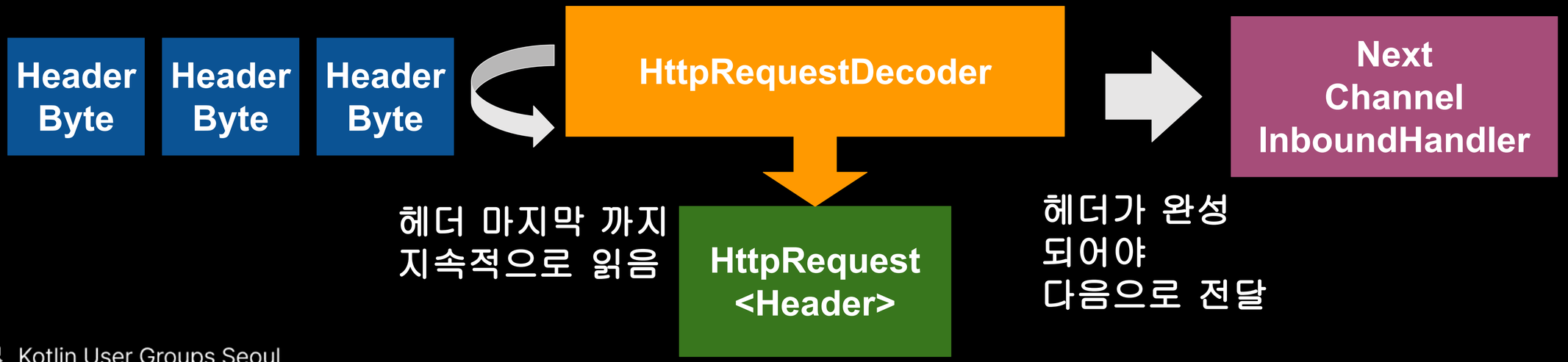
Armeria Netty HTTP Class

- Netty 6가지 HTTP 관련 Class들



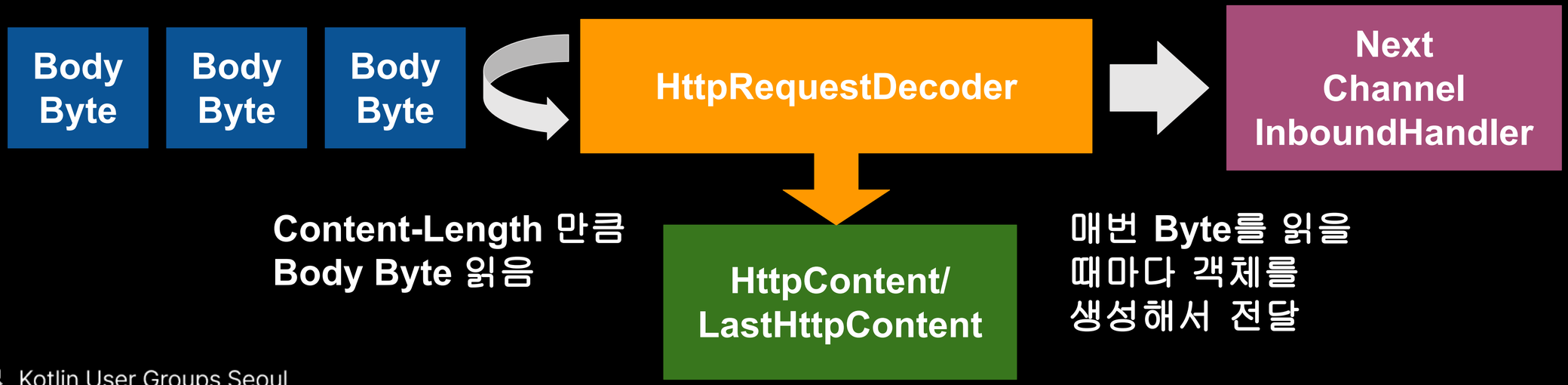
Armeria HttpRequestDecoder Header

- Byte를 읽어들여 헤더를 의미하는 HttpRequest 객체 생성 및 멤버 변수로 저장
- Header의 끝까지 계속 Byte를 읽어가며 HttpRequest에 헤더를 추가
- Header를 모두 읽으면 HttpRequest를 다음 ChannelInboundHandler에게 전달



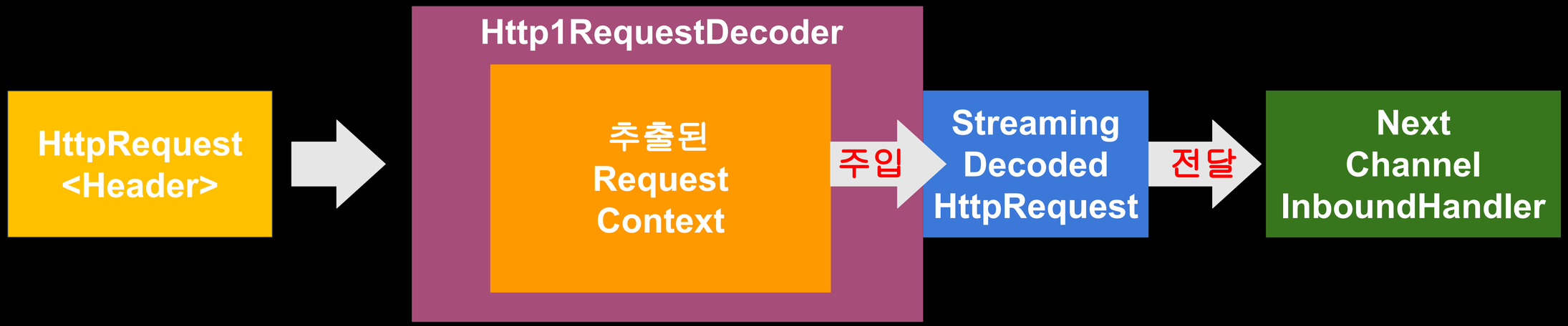
Armeria HttpRequestDecoder Body

- Body의 Byte를 읽어들여 HttpContent로 변경
- Content-Length 만큼 Body의 Byte를 모두 읽어들이면 LastHttpContent 생성
  - Last/HttpContent는 값으로 Byte를 가지고 있음
- Msg를 생성할 때마다 다음 ChannelInboundHandler로 전달



Armeria Http1RequestDecoder - 1

- Netty의 HTTP 관련 Class를 이용하여 동작
- 프로토콜 규약에 맞춰 요청 검사
- HttpRequest의 정보를 바탕으로 어떤 Armeria Service에게 전달할지 결정
- HttpRequest 객체를 요청 문맥과 함께 Armeria의 StreamingDecodedHttpRequest로 변환 및 멤버 변수로 저장

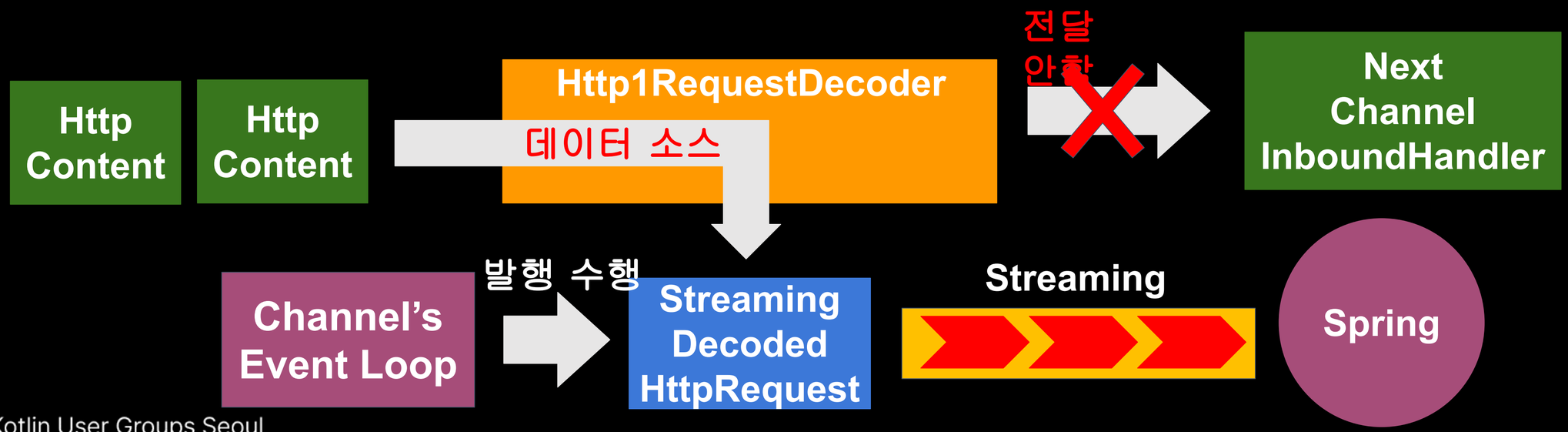


Armeria StreamingDecodedHttpRequest

- Reactive Streams의 Publisher 구현
- 추후에 Last/HttpContent가 도착하면 발행 가능
- 최종 구독자는 Spring Webflux
  - Spring Controller 함수의 Mono/Flux로 데이터 발행

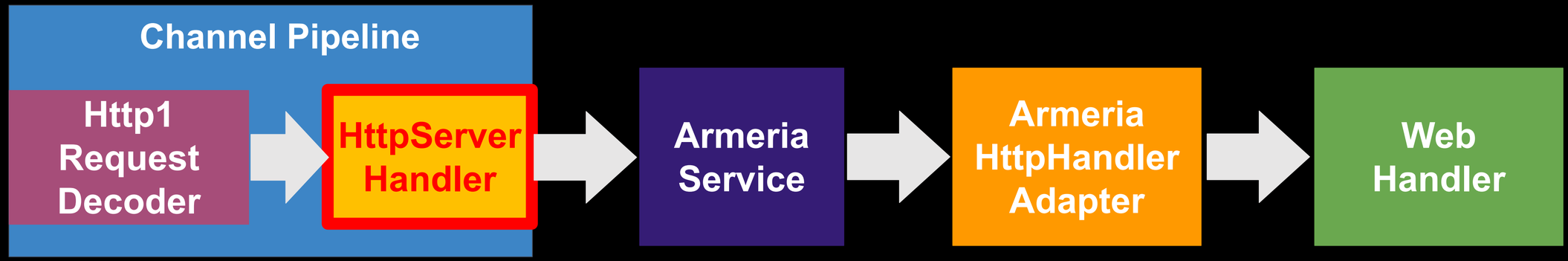
Armeria Http1RequestDecoder - 2

- StreamingDecodedHttpRequest를 다음 ChannelHandler로 전달
- 다음 도착하는 Last/HttpContent는 다음 ChannelHandler에게 넘기지 않고 StreamingDecodedHttpRequest에게 발행
- 발행 Thread는 Channel의 EventLoop 이용



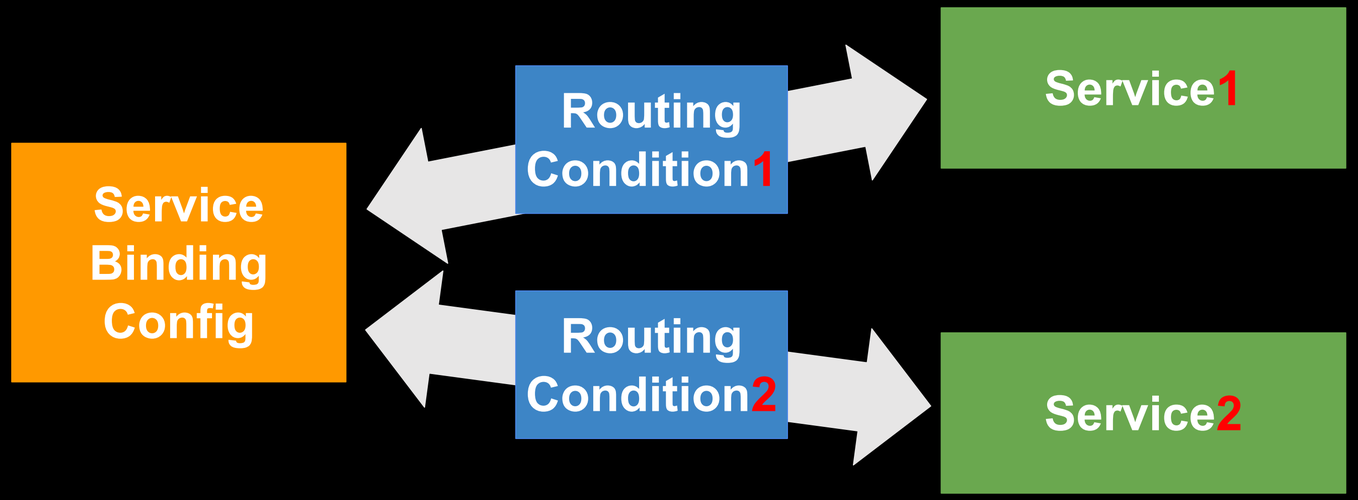
Armeria HttpServerHandler - 1

- 마지막 ChannelInboundHandler
- HTTP 요청, 서버 설정, 요청 문맥을 Armeria Service의 인자인 ServiceRequestContext로 감싸서 선택된 Armeria Service에게 위임
- Service는 ArmeriaHttpHandlerAdapter에게 다시 위임
- Adapter가 Spring Webflux의 WebHandler에게 위임



Armeria Service

- Armeria의 Service.java의 구현체
- ChannelPipeline 밖에 존재
- 요청의 Host Name, Port, Path 등등의 많은 값을 이용하여 수신 서비스 결정
- 우리 예제의 모든 요청은 Spring Webflux에게 위임하기 위한 Service로 결정됨

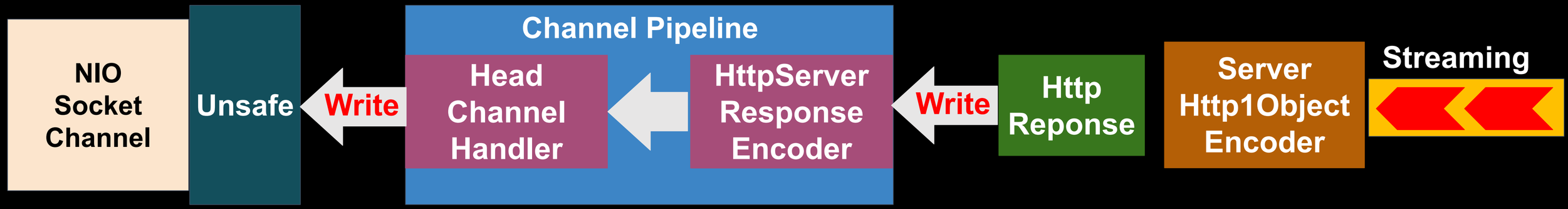


Armeria HttpServerHandler - 2

- Service가 구독 가능한 DefferedHttpResponse를 반환
- Response는 Reactive Streams의 Publisher 구현
- Response를 구독하며 전달 되는 HTTP 응답을 ServerHttp1ObjectEncoder에게 전달
  - 응답의 발행자는 Spring Webflux
- Encoder는 Encoding 수행 뒤 ChannelPipeline에 write 수행

Armeria Http Response

- ChannelPipeline에서 write 발생
- HttpServerResponseEncoder를 거쳐 NIOSocketChannel에 전달
- NIOSocketChannel에서 Network write I/O 발생



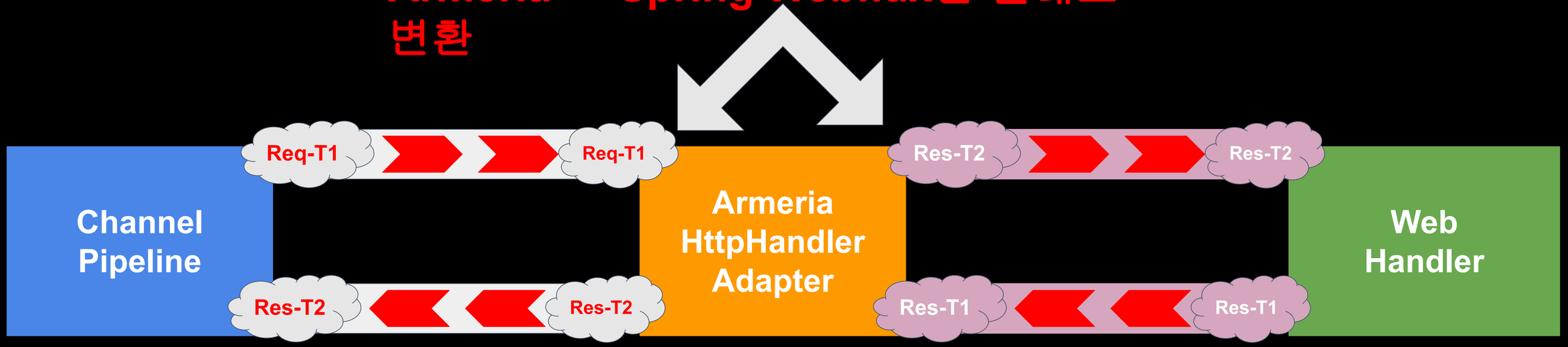
Armeria Spring Webflux

- Adapter에서 WebHandler의 함수에 알맞게 객체 변경
  - ServerWebExchange.java
- WebHandler가 반환한 Mono를 구독하면 Armeria ↔ Spring Webflux의 Reactive Streams Pipeline에서 구독/발행 시작

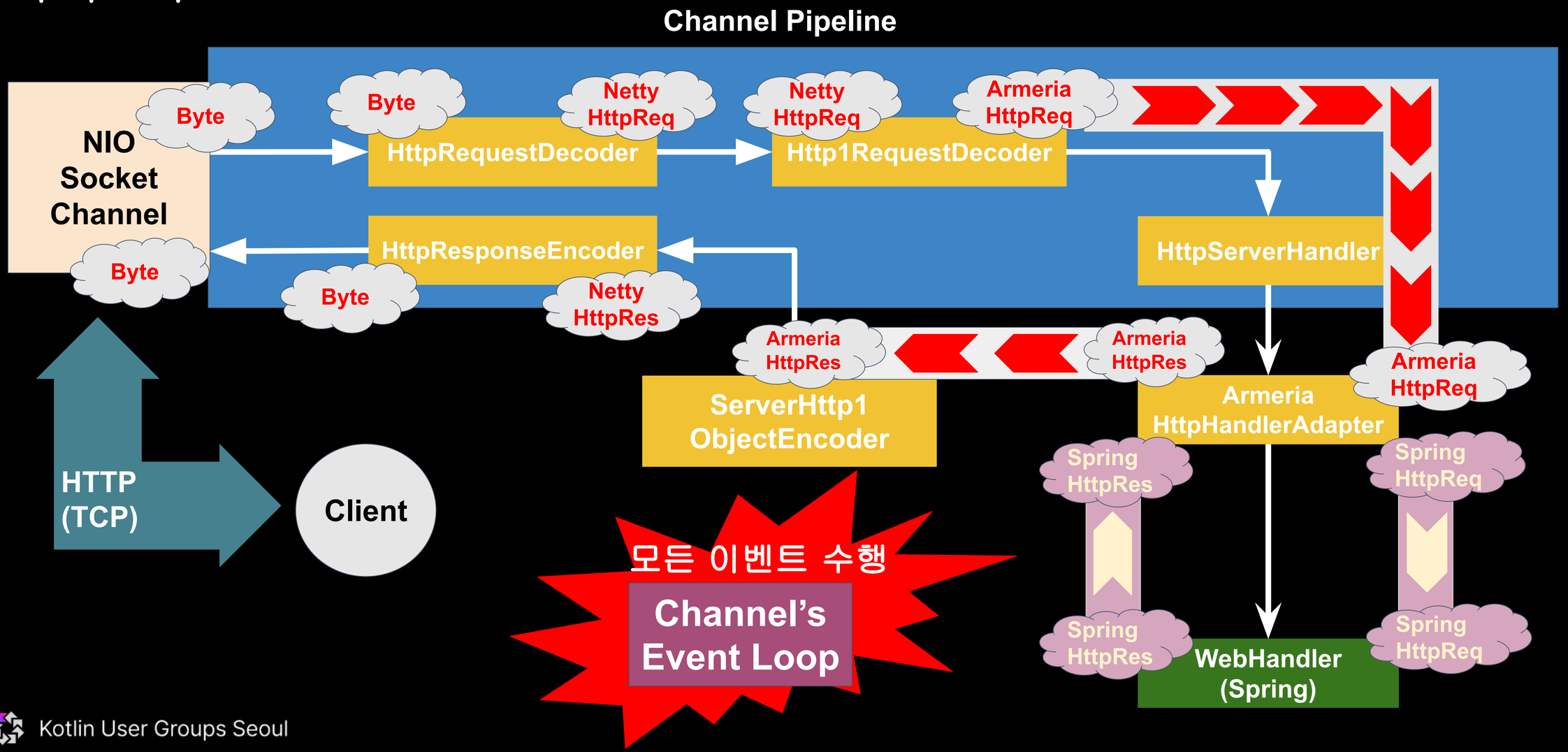
```
public interface WebHandler {
    /**
     * Handle the web server exchange.
     * @return {@code Mono<Void>} to indicate when request handling is completed
     */
    Mono<Void> handle(ServerWebExchange exchange);
}
```

Armeria Adapter Layer

- Spring Webflux의 함수 선언에 맞춰 요청/응답 클래스 변경 필요
- ArmeriaServerHttpRequest/Response가 수행



아무리


Channel Pipeline
NIO Socket Channel
Byte
Byte
Netty HttpReq
Netty HttpReq
Armeria HttpReq
HttpRequestDecoder
Http1RequestDecoder
HttpServerHandler
HttpResponseEncoder
Byte
Byte
Netty HttpRes
Armeria HttpRes
Armeria HttpRes
Armeria HttpReq
ServerHttp1 ObjectEncoder
Armeria HttpHandlerAdapter
HTTP (TCP)
Client
모든 이벤트 수행
Channel's Event Loop
Spring HttpRes
Spring HttpReq
Spring HttpRes
Spring HttpReq
WebHandler (Spring)

감사합니다!

Special Thanks to 정승주

츄핀테크

발표 영상 다시 보기

https://www.youtube.com/@chupin-park