DEVIEW 2020

비동기 서버 그까이꺼, Request Scoping만 알면 끝!

Minwoo Song LINE PLUS

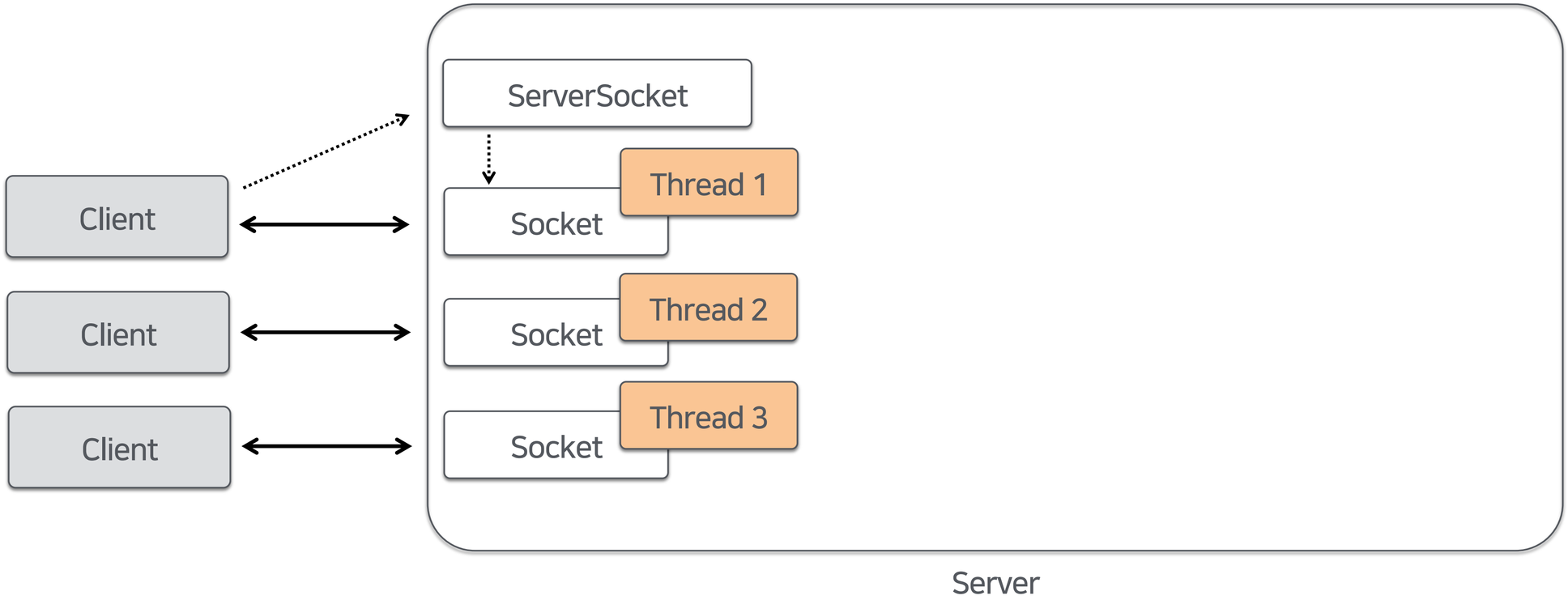
Managing and providing information
on which request the logic was derived by
when executing a specific logic

CONTENTS

1. How an asynchronous server works?

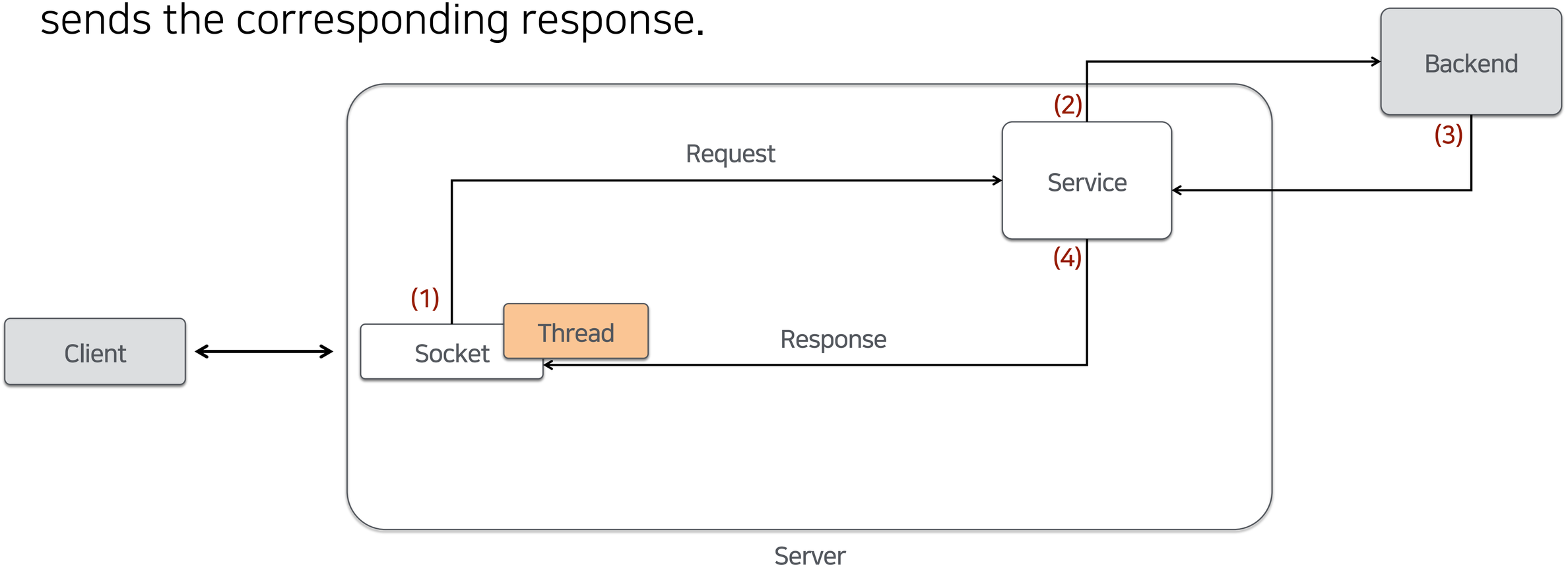
Syncronous server

- Each thread handles one socket.



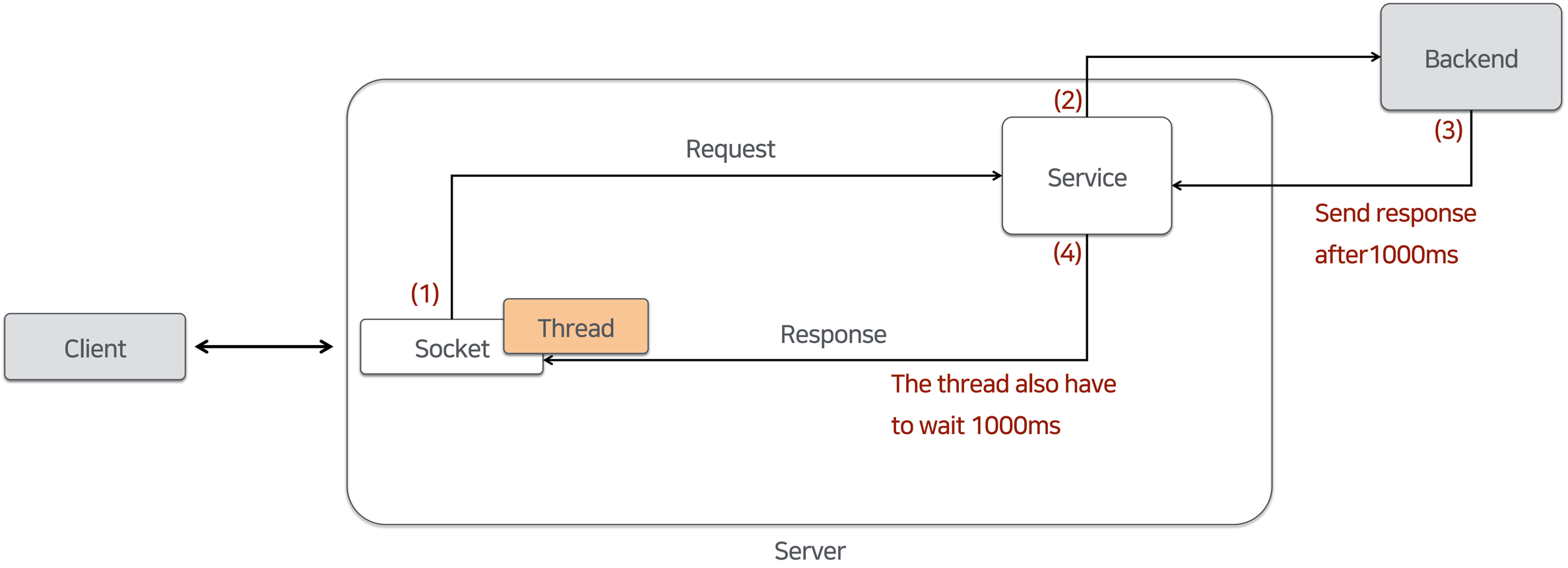
Syncronous server

- Once the thread receives a request it can't do anything until it sends the corresponding response.



Syncronous server

- Even when it takes longer to get a response from the backend.



Syncronous server

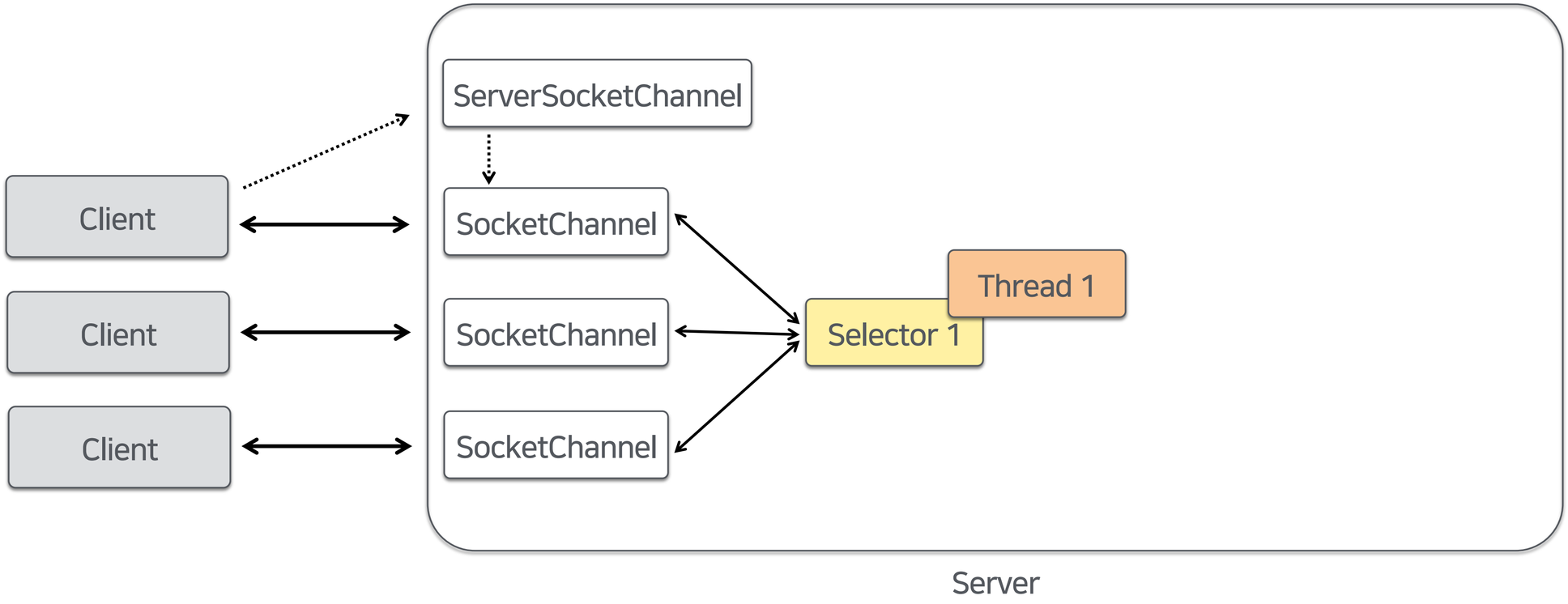
- Simpler code
- Can’t solve C10k problem

Asyncronous server

- `EventLoop`
- Wrapper

Asyncronous server

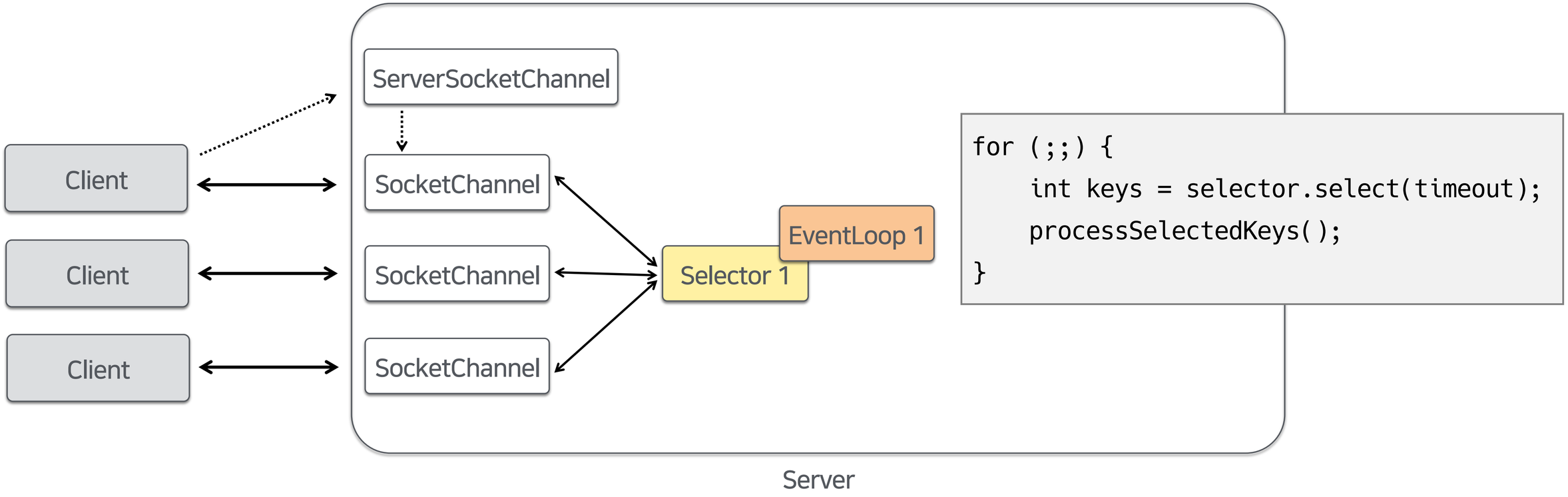
- Each thread handles many socket channels.




```
socketChannel.register(selector, SelectionKey.OP_READ, ...);
```

EventLoop

- A single thread selector loop



EventLoop

- A single thread selector loop
  - epoll, io_uring(Linux), kqueue(FreeBSD)
- Relatively few threads handle all IO.
  - Usually the number of core * 2
  - Tomcat uses 200 threads by default.

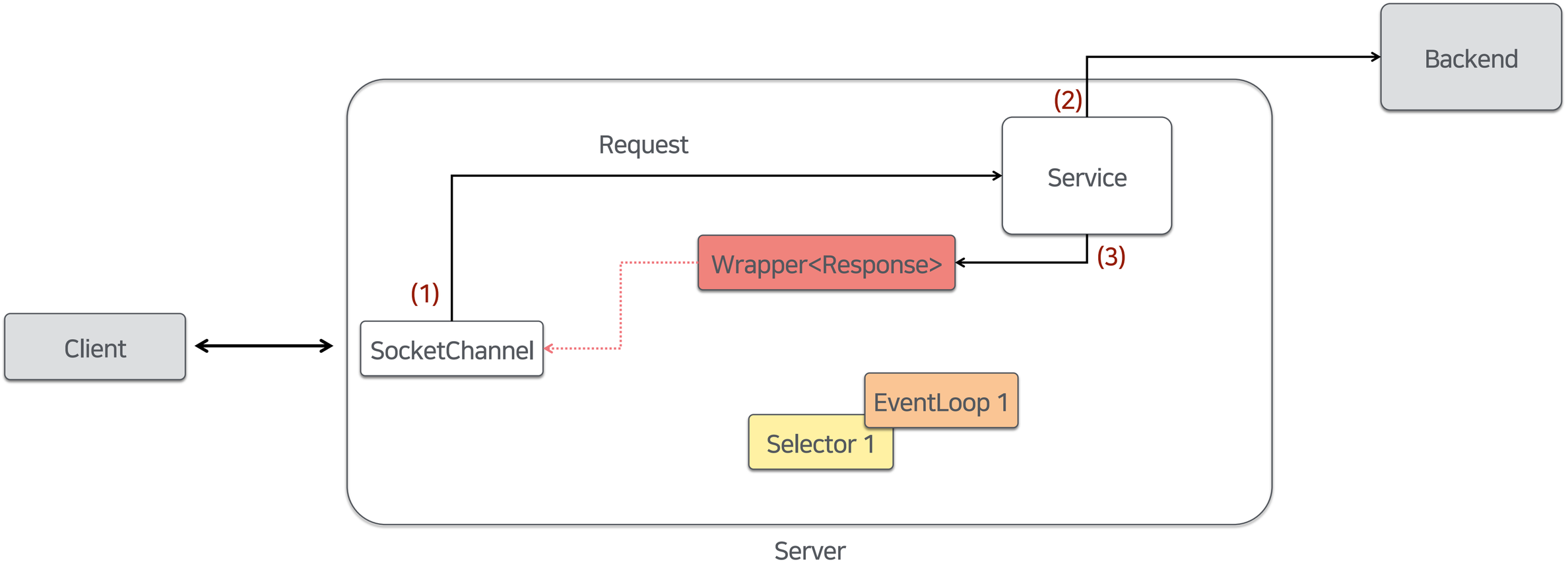
- What's the rational size of a thread pool?

$$Ncpu * Ucpu * (1 + {}^{W}/_{C})$$

Goetz, Brian. *Java concurrency in practice*. Addison-Wesley, 2006.

Wrapper

- The service returns a wrapper instead of the actual response.



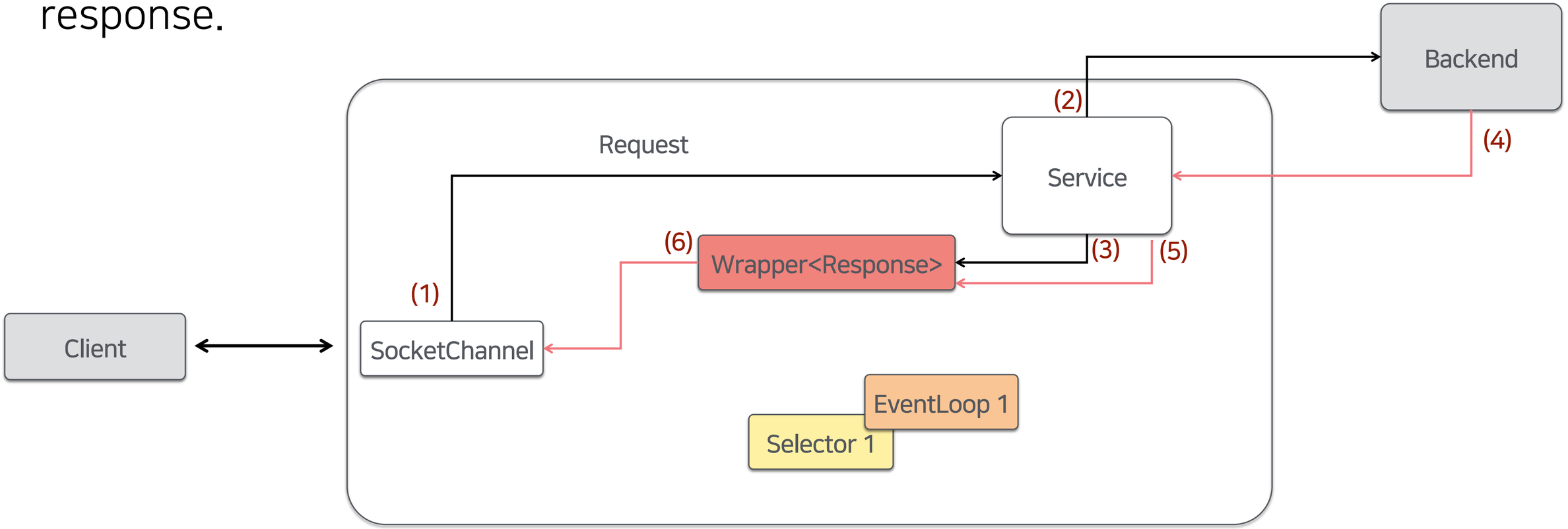
Wrapper

DEVIEW 2020

- A wrapper that will have one or more instances now or in the future
  - e.g. `Future`, `Promise`, `Publisher`, etc,
- Should use asynchronous callback (No blocking operation)
- The request could also be a wrapper because it might be composed of multiple frames. (e.g. headers and body)

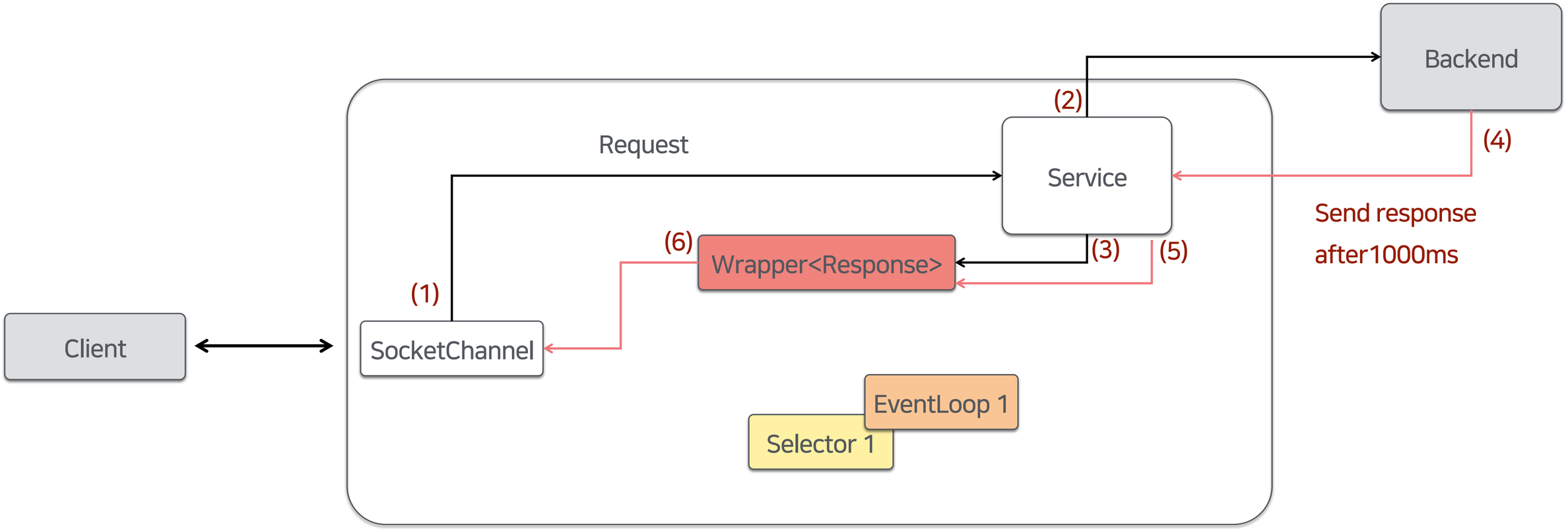
Wrapper

- The actual response will be sent later when the Backend sends the response.



Asynchronous server

- The `EventLoop` does not wait for the response from the backend.

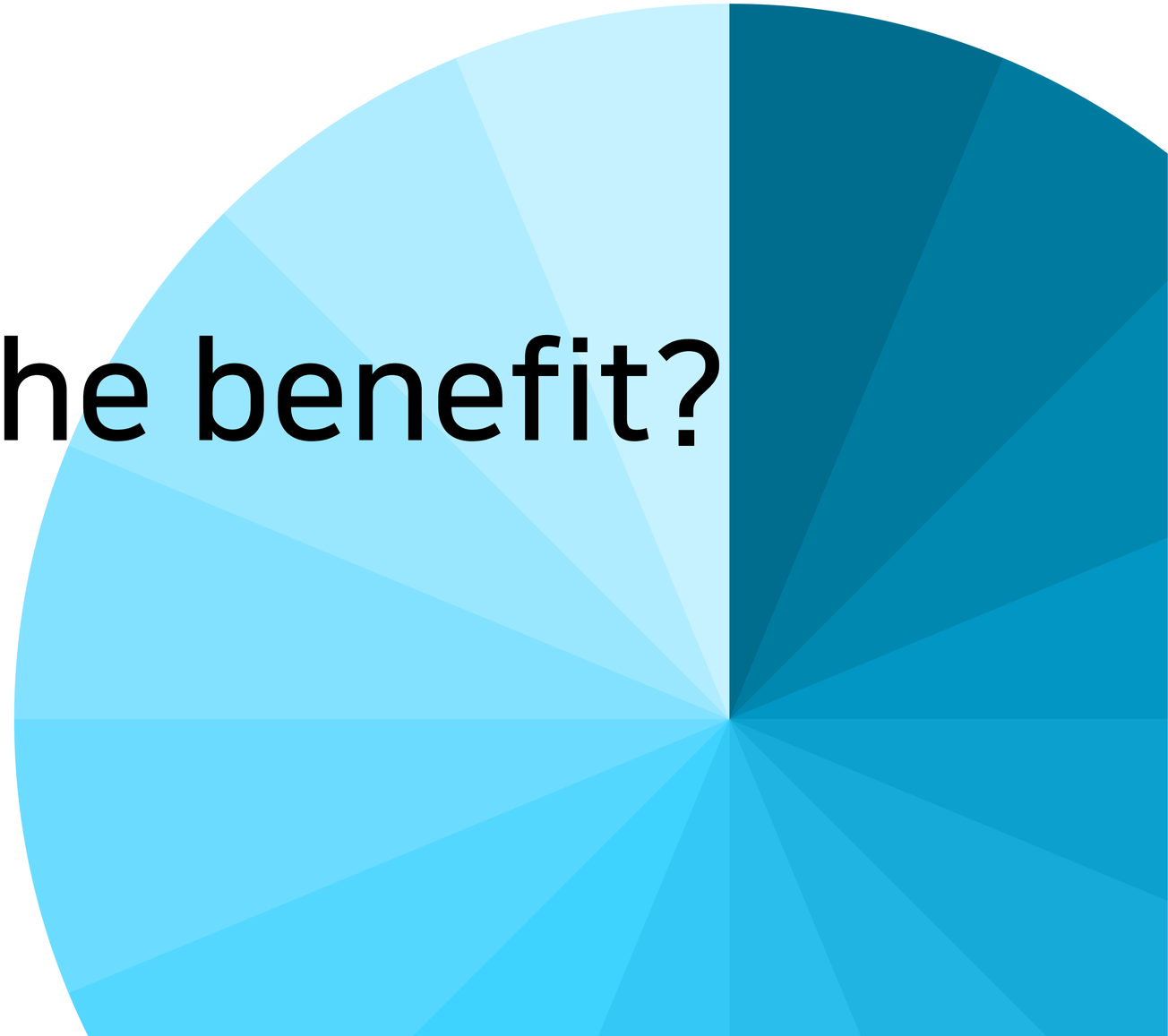


Asyncronous server

DEVIEW 2020

- Full example: https://github.com/minwoox/deview2020-requestscoping

2. What’s the benefit?



Performance

DEVIEW 2020

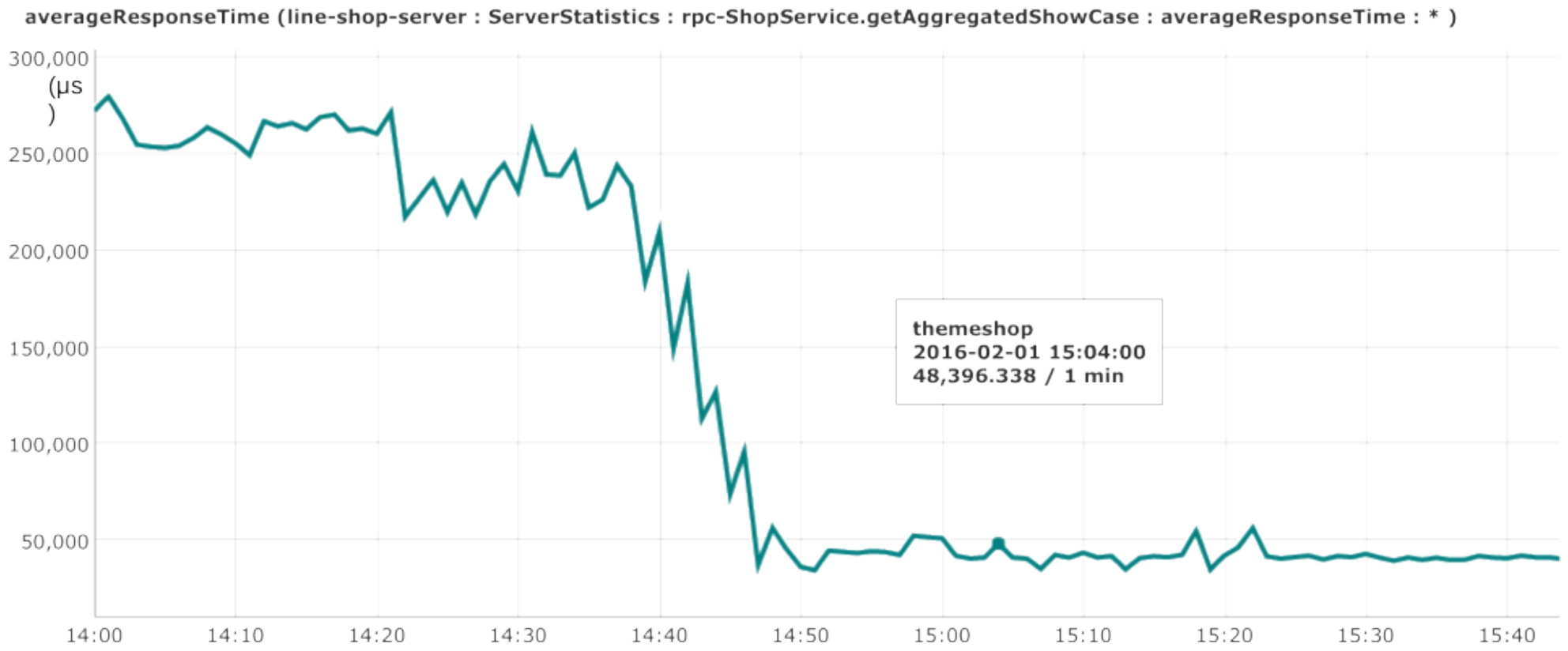
- Can even achieve 1M concurrent connections on a powerful machine

Case of LINE

DEVIEW 2020

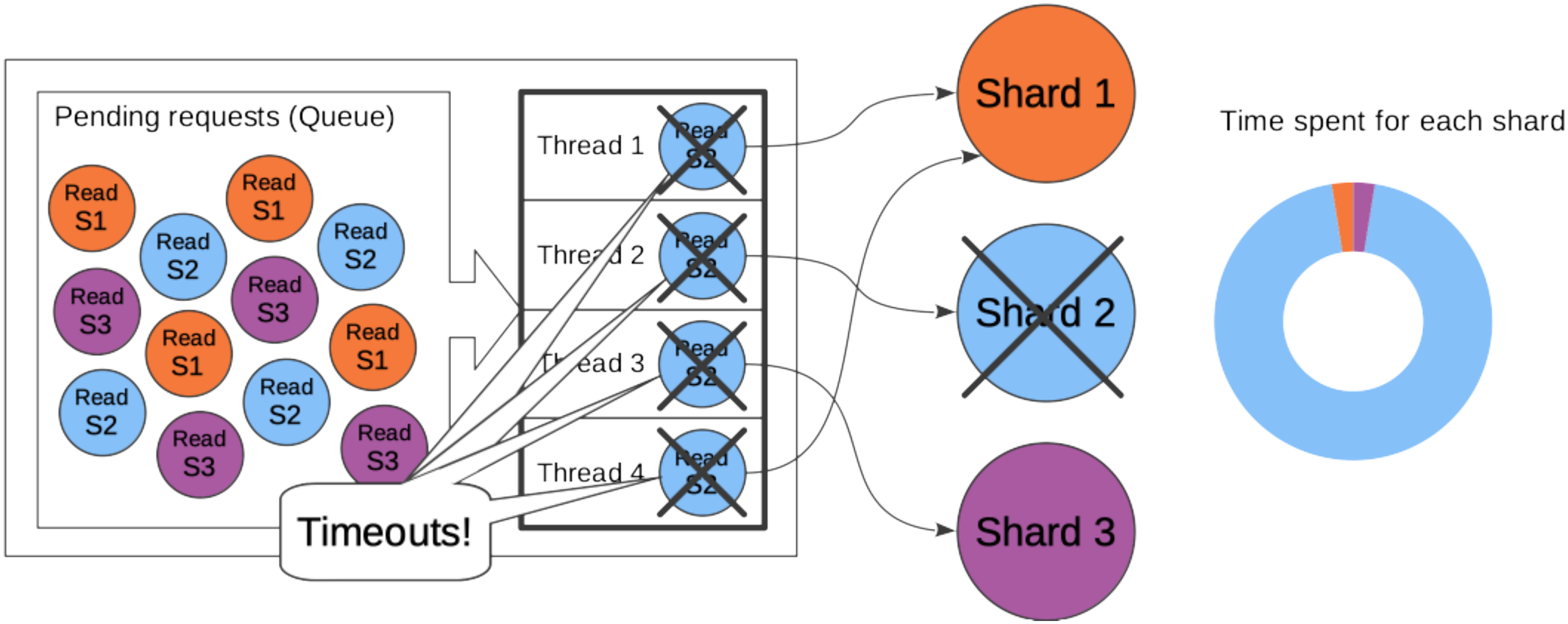
- LINE sticker store (50k -100k RPS)
  - Before: Spring Boot + Tomcat(HTTP/1.1)
  - After: Spring Boot + an async server(HTTP/2)

Case of LINE



- Full story: https://www.slideshare.net/linecorp/a-9-line-shop-powered-by-armeria

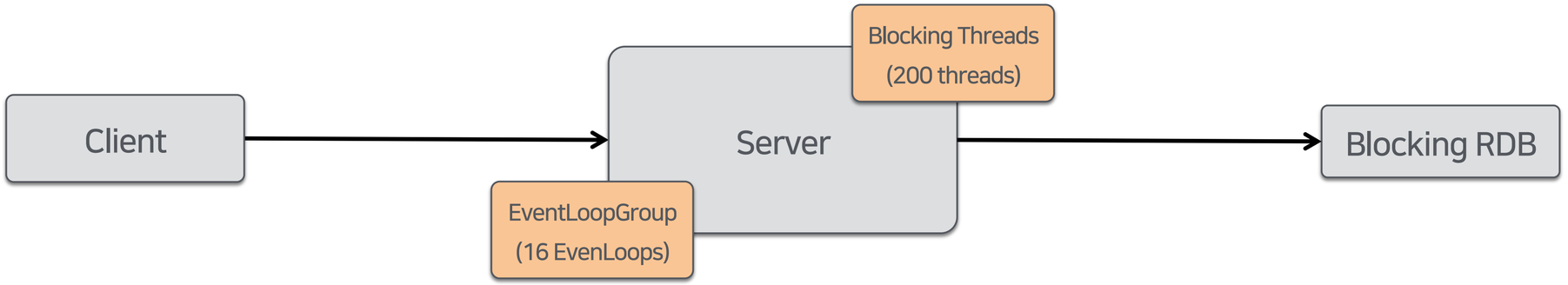
Works even on a shard failure



- Full story: https://tv.naver.com/v/11267414

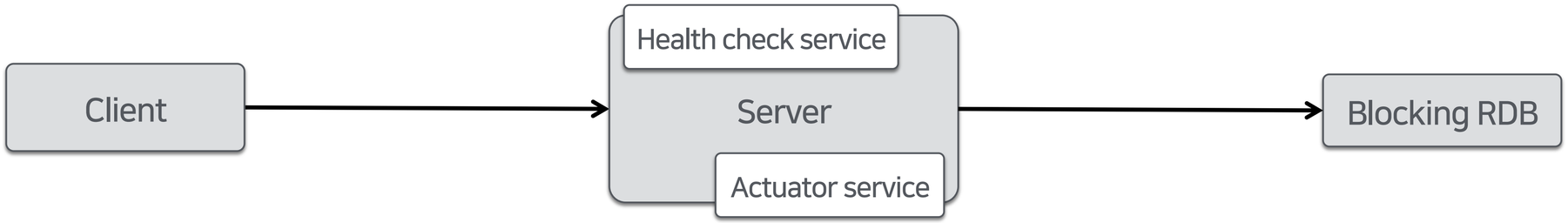
I'm just using blocking RDB!

- Blocking calls happen by delegating to the blocking threads.
  - 200 threads by default
- In addition to the blocking threads, the asynchronous server needs 16 more threads! (if the machine has 8 cores.)



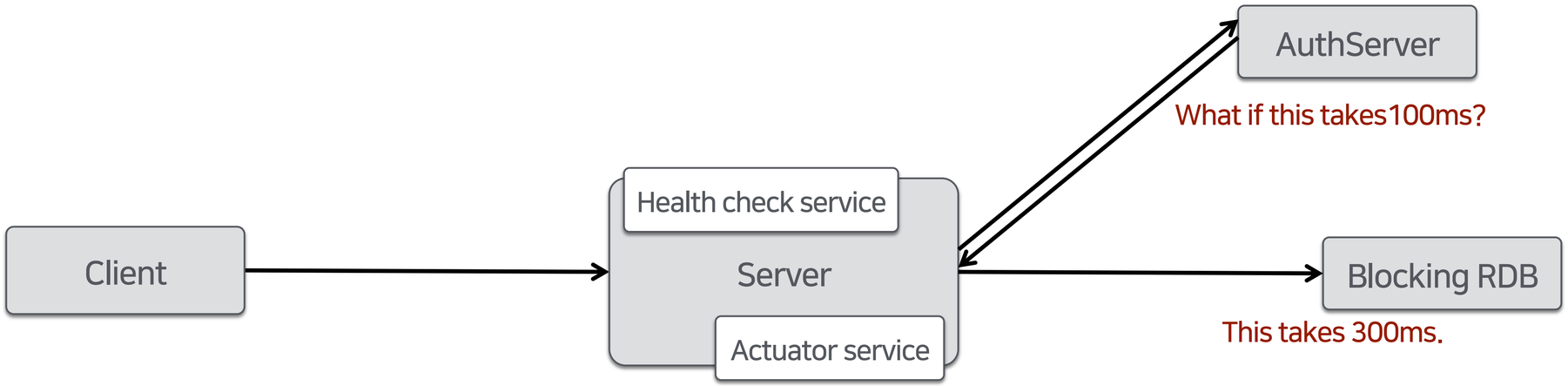
I'm just using blocking RDB!

- What if there's a health check service?
- What if there's an actuator service?



I'm just using blocking RDB!

- What if there's a health check service?
- What if there's an actuator service?
- What if requests have to go to the authentication server before querying?
- What if it's microservice?



3. Is it always good?

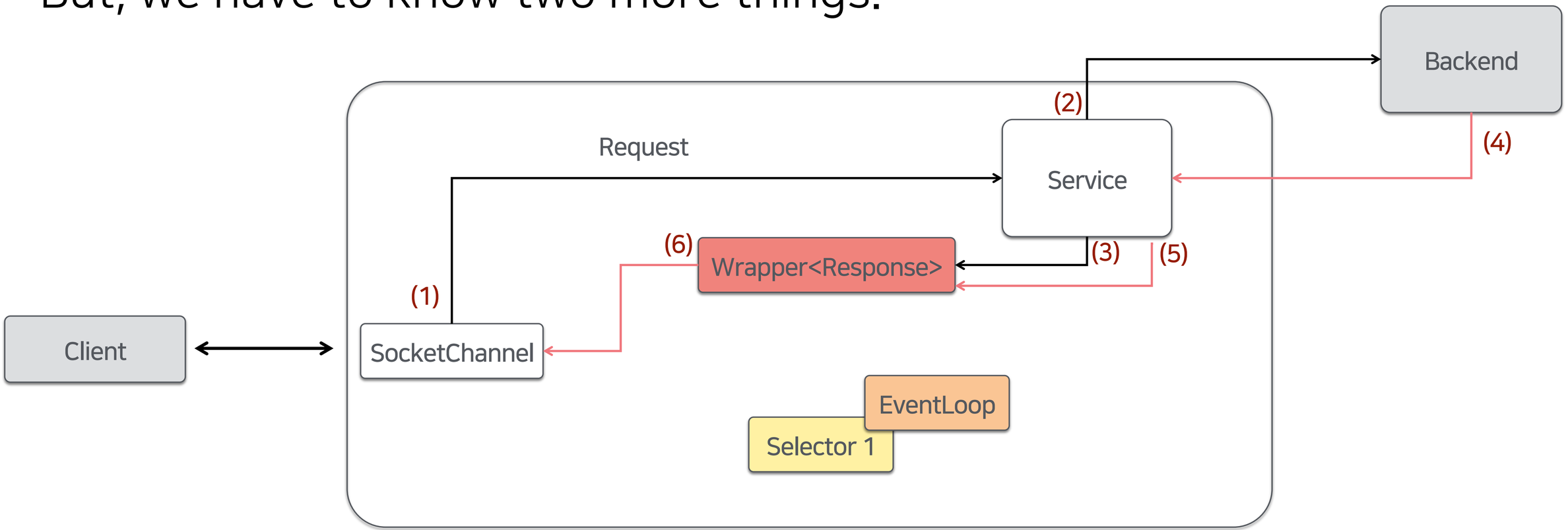
Drawbacks?

DEVIEW 2020

- Not easy to understand the control flow
- Callback hell
- Coloring problem
- Lost context

Control flow

- We've already learned it.
- But, we have to know two more things.



Quiz

- What's the result?

```
1:  // Executed by the thread-A.
2:  CompletableFuture<String> future = new CompletableFuture<>();
3:  Executor threadB = …; // The name of this executor is thread-B.
4:  threadB.execute(() -> {
5:      future.complete("foo")
6:  });
7:
8:  future.thenAccept(value -> {
9:      System.err.println("Executed by: " + Thread.currentThread.getName());
10: });
```

Which thread executes the Callback?

DEVIEW 2020

- The thread that calls callback
  (The wrapper is already complete.)

```
// Executed by the thread-A.
completedFuture.thenAccept(value -> {
    // Executed by the same thread-A.
});
```

- The thread that completes the wrapper
  (The wrapper is not complete yet.)

```
// Executed by the thread-A.
incompleteFuture.thenAccept(value -> {
    // Executed by a thread named
    // thread-B which completes the
    // incompleteFuture.
});
```

- The thread that is specified when calling callback
  (No matter the wrapper is complete or not.)

```
future.thenAcceptAsync(value -> {
    // Executed by the executorC.
}, executorC);
```

Blocking operation

- Blocks the current thread to wait for some event
  - e.g. `future.get()`, `future.join()`, `flux.blockFirst()`, etc.
- Never use blocking operation in asynchrony world except:
  - In test code
  - When starting(stopping) a server, or in the process of it
  - In a dedicated blocking thread
- Otherwise, you might be doing like this:

```
// Executed by thread-A
CompletableFuture<String> future = new CompletableFuture<>();
String fooString = future.get(); // Never complete.
…
future.complete("foo"); // The thread waiting above is the one who completes the future.
```

Reactive Streams

DEVIEW 2020

- Most modern JVM frameworks use it for stream processing.
- A standard for asynchronous stream processing with non-blocking backpressure

Reactive Streams

- Most modern JVM frameworks use it for stream processing.
- ~~A standard for asynchronous stream processing with non-blocking backpressure~~
- Just 7 methods in 3 interfaces
  - Not easy to implement in multi threading
  - Not hard to understand the flow
- Good material: https://engineering.linecorp.com/ko/blog/reactive-streams-with-armeria-1/

Callback hell

DEVIEW 2020

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

- https://anexsoft.com/javascript-callbacks-vs-promise-vs-asyncawait

Callback hell

- What if the programmer makes the code in synchronous way?
- Not a problem of asynchrony but the lack of peers who can properly do the code review

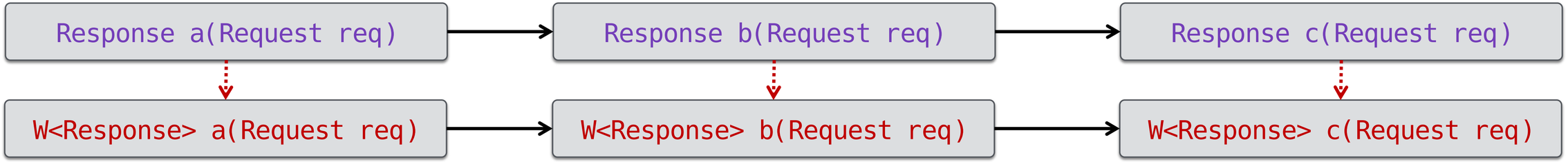
Coloring problem

- Can’t call asynchronous methods in a synchronous method.

```
1:  // Asynchronous method
2:  CompletableFuture<BackendResponse> getResponse(…) {…}
3:
4:  // Synchronous method
5:  Response helloService(Request req) {
6:      CompletableFuture<BackendResponse> future =
7:          getResponse(…);
8:      BackendResponse res = future.get(); // Should call get()!!
9:      return makeRespone(res);
10: }
```

Coloring problem

- Have to change all methods in the call stack as asynchronous methods.

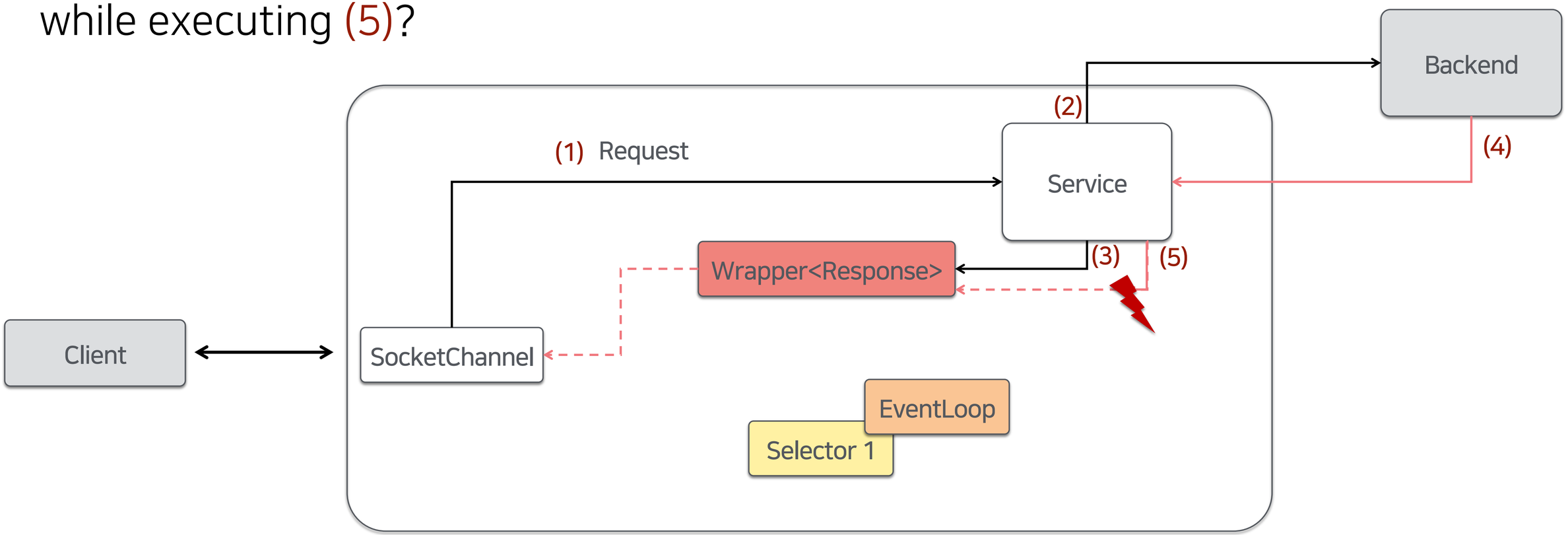


- But, what if you build your server with asynchrony from the first?
  - Just use a blocking thread at the end.

```
1: W<Response> c(Reqeust req) {
2:     W<Response> wrapper = …;
3:     blockingTaskExecutor.execute(() -> {
4:         // Do some blocking job!
5:         wrapper.complete(…);
6:     });
7:     return wrapper;
8: }
```

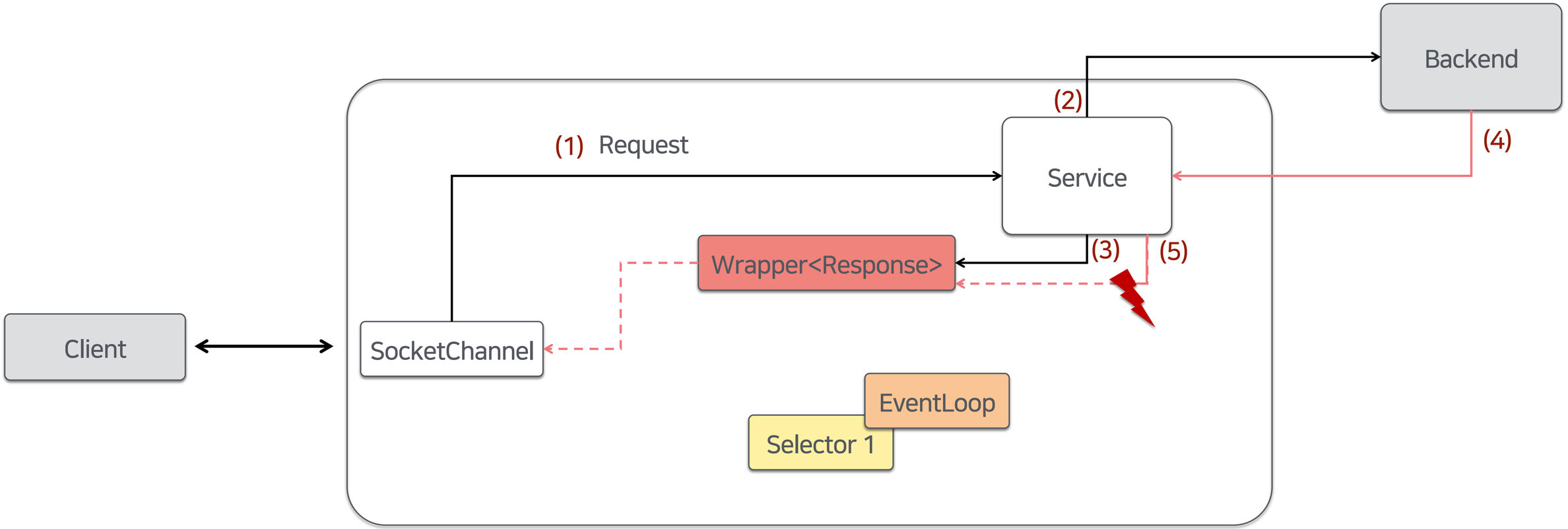
Lost Context

- Does the stack trace show (1), (2) and (3) if an exception is raised while executing (5)?



Lost Context

- We have to designate that (5) is derived by the (1) request.



4. Request scoping

Request scoping

- Managing and providing information on which request the logic was derived by when executing a specific logic
- Don’t know how others are doing, but I know how **Armeria** is doing.

Armeria

- Your go-to microservice framework for any situation. You can build any type of microservice leveraging your favorite technologies, including gRPC, Thrift, Kotlin, Retrofit, Reactive Streams, Spring Boot and Dropwizard.

Request scoping

DEVIEW 2020

- Let’s put aside the stack trace thing for now, but focus on more realistic example.

Zipkin

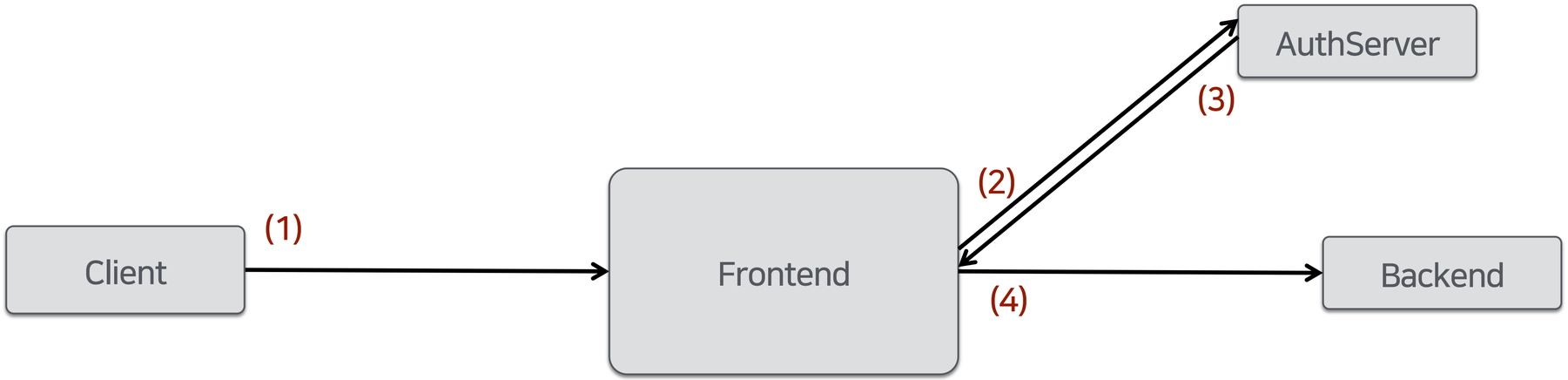
- Great open source for Distributed tracing
  - A method used to profile and monitor applications, especially those built using microservice architecture
- Can do distributed tracing with just one decorator.

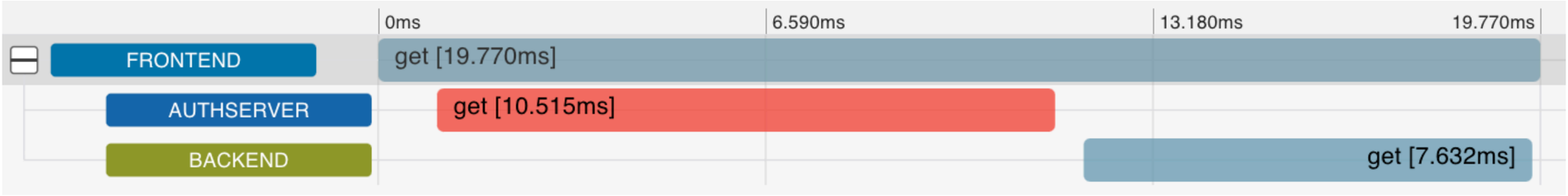
```
ServerBuilder sb = …;
sb.decorator(BraveService.newDecorator(tracing));
```

- Zipkin also uses Armeria!

Distributed tracing

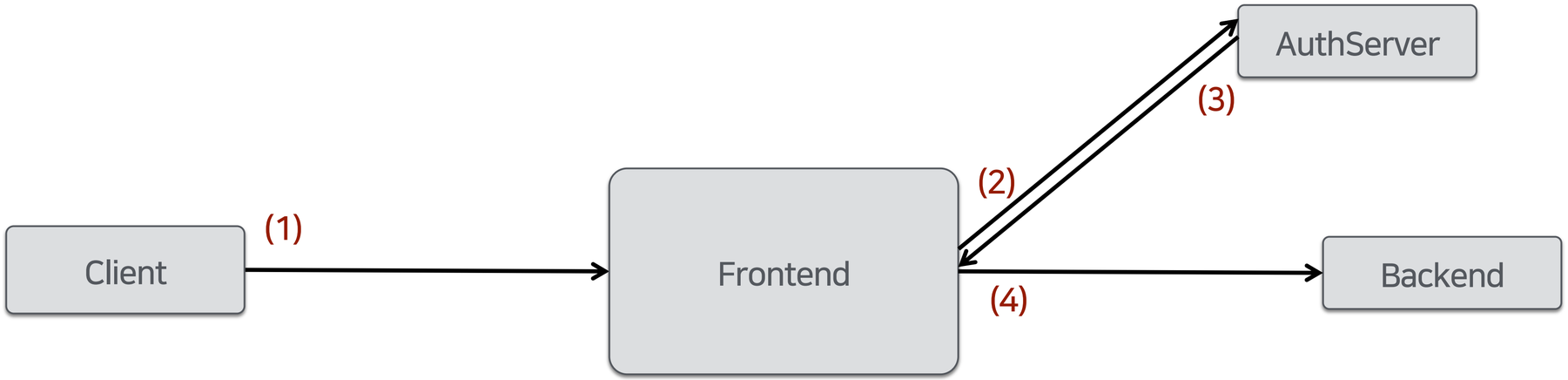
- Perfect request scoping!

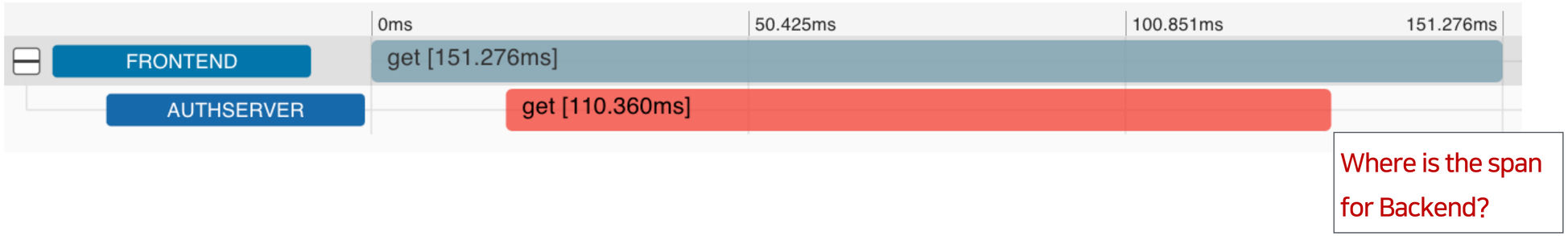




Distributed tracing

- What if it fails to do the request scoping?





Where is the span for Backend?

Request scoping

DEVIEW 2020

```
1:  ServerBuilder sb = …;
2:  sb.service("/api", (ctx, req) -> {
3:        CompletableFuture<HttpResponse> future = new CompletableFuture<>();
4:        // Request scoping is done by Armeria.
5:        HttpResponse authResponse = authClient.execute(req.headers());
6:        CompletableFuture<AggregatedHttpResponse> aggregated = authResponse.aggregate();
7:        aggregated.thenAccept(aggregatedRes -> {
8:            // You should do the request scoping by yourself because this thread
9:            // does not know which request this logic was derived by.
10:           HttpResponse response = backendClient.execute(req);
11:           future.complete(response);
12:       });
13:       return HttpResponse.from(future);
14:   );
```

How to check?

- A `ServiceRequestContext` is created whenever the server receives a request.
- The context is pushed to the thread-local to indicate the request scope.
- If `RequestContext.currentOrNull()` returns `null`, request scoping hasn't done.

How to check?

DEVIEW 2020

```
1:  sb.service("/api", (ctx, req) -> {
2:        CompletableFuture<HttpResponse> future = new CompletableFuture<>();
3:        // Request scoping is done by Armeria.
4:        assert RequestContext.currentOrNull() != null;
5:        HttpResponse authResponse = authClient.execute(req.headers());
6:        CompletableFuture<AggregatedHttpResponse> aggregated = authResponse.aggregate();
7:        aggregated.thenAccept(aggregatedRes -> {
8:            // You should do the request scoping by yourself.
9:            assert RequestContext.currentOrNull() == null;
10:           HttpResponse response = backendClient.execute(req);
11:           future.complete(response);
12:       });
13:       return HttpResponse.from(future);
14:   );
```

Do it manually


```
1:  ServerBuilder sb = …;
2:  sb.service("/api", (ctx, req) -> {
3:        CompletableFuture<HttpResponse> future = new CompletableFuture<>();
4:        HttpResponse authResponse = authClient.execute(req.headers());
5:        authResponse.aggregate().thenAccept(aggregatedRes -> {
6:            try (SafeCloseable ignored = ctx.push()) {
7:                HttpResponse response = backendClient.execute(req);
8:                future.complete(response);
9:            }
10:       });
11:       return HttpResponse.from(future);
12:   );
```

ContextAwareFuture

DEVIEW 2020

```
1:  ServerBuilder sb = …;
2:  sb.service("/api", (ctx, req) -> {
3:        CompletableFuture<HttpResponse> future = new CompletableFuture<>();
4:        HttpResponse authResponse = authClient.execute(req.headers());
5:        CompletableFuture<AggregatedHttpResponse> aggregated = authResponse.aggregate();
6:        CompletableFuture<AggregatedHttpResponse> contextAwareFuture =
7:            ctx.makeContextAware(aggregated);
8:        contextAwareFuture.thenAccept(aggregatedRes -> {
9:            HttpResponse response = backendClient.execute(req);
10:           future.complete(response);
11:       });
12:       return HttpResponse.from(future);
13:   );
```

@armeria_project line/armeria

What? Hold on.

DEVIEW 2020

- Can’t just return the context aware future from `authResponse.aggregate()`?

```
1:  CompletableFuture<AggregatedHttpResponse> aggregated = authResponse.aggregate();
2:  CompletableFuture<AggregatedHttpResponse> contextAwareFuture =
3:      authResponse.aggregate();
4:  contextAwareFuture.thenAccept(aggregatedRes -> {
5:      HttpResponse response = backendClient.execute(req);
6:      future.complete(response);
7:  });
```

What? Hold on.

DEVIEW 2020

- We have decided not to because we can’t automatically do that in all situations.

```
1:  CompletableFuture<AggregatedHttpResponse> backend1 = …;
2:  CompletableFuture<AggregatedHttpResponse> backend2 = …;
3:  CompletableFuture<Void> allOfFuture =
4:      CompletableFuture.allOf(backend1, backend2);
5:  // We can’t make the allOfFuture context aware.
6:  allOfFuture.thenRun(() → {…});
```

What? Hold on.

- Well-suited everywhere
  - A user should be able to use the library what he/she wants.
- If we can't fully automate request scoping, then we'd better crash loudly in the early stages and let people know and understand what's going on.

Executor

DEVIEW 2020

```
1:  ServerBuilder sb = …;
2:  sb.service("/api", (ctx, req) -> {
3:        CompletableFuture<HttpResponse> future = new CompletableFuture<>();
4:        HttpResponse authResponse = authClient.execute(req.headers());
5:        CompletableFuture<AggregatedHttpResponse> aggregated = authResponse.aggregate();
6:        Executor contextAwareExecutor = ctx.eventLoop();
7:        aggregated.thenAcceptAsync(aggregatedRes -> {
8:            HttpResponse response = backendClient.execute("/api");
9:            future.complete(response);
10:       }, contextAwareExecutor);
11:       return HttpResponse.from(future);
12:   );
```

- Full example: https://github.com/line/armeria/tree/master/examples/context-propagation

Best practice

- Use callbacks that takes an executor which automatically does the request scoping for you.

```
ServiceRequestContext ctx = …;
future.thenAcceptAsync(…, ctx.eventLoop());

// for Reactor
Flux flux = …;
flux.subscribeOn(Schedulers.fromExecutor(ctx.eventLoop()));
```

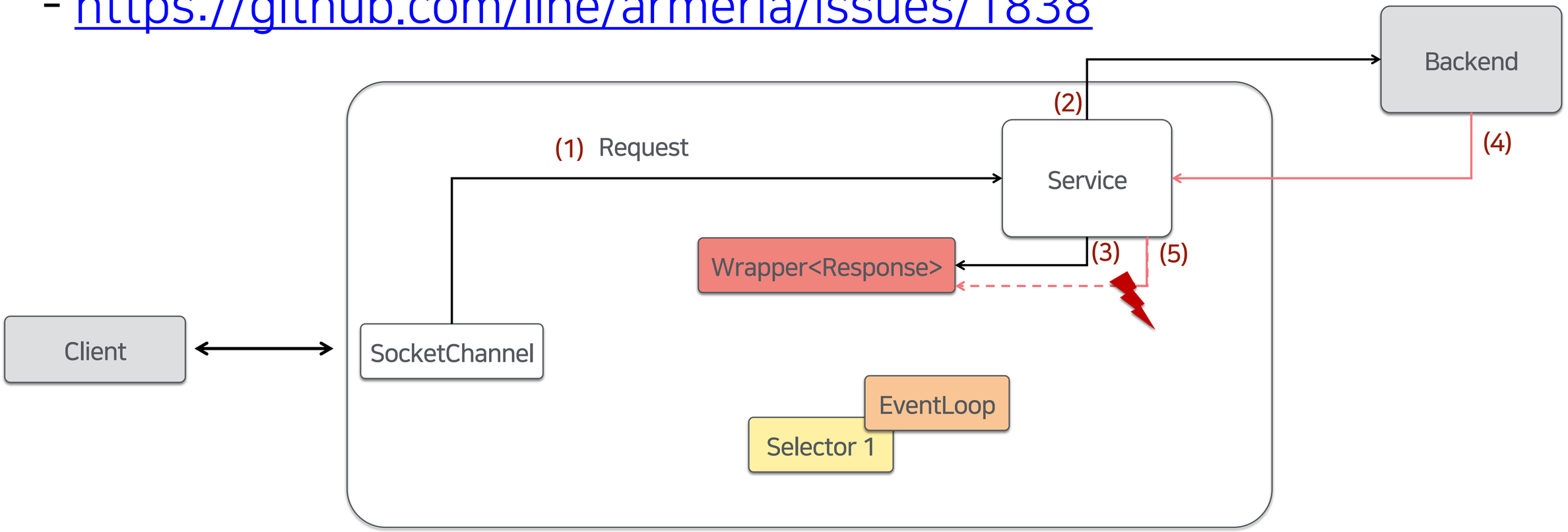
Coroutine

DEVIEW 2020

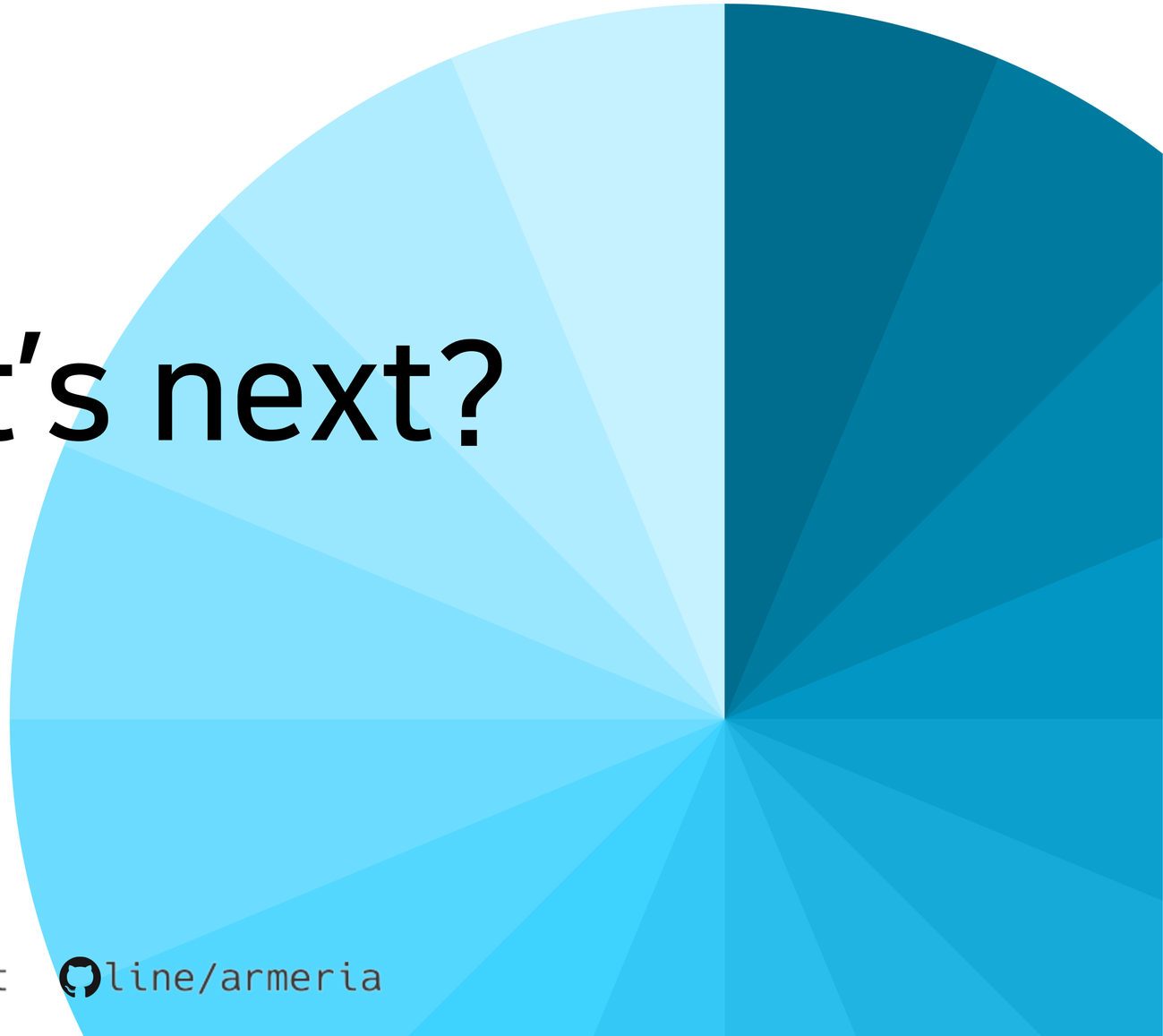
```
1:  val response = GlobalScope.future(ctx.eventLoop().asCoroutineDispatcher()) {
2:      val authResponse = authClient.execute(req.headers()).aggregate().await()
3:      require(httpResponse.status() == HttpStatus.OK)
4:      backendClient.execute(req)
5:  }
6:  return HttpResponse.from(response)
```

Stack trace

- We still do not know the stack trace because we haven't implemented it yet.
  - https://github.com/line/armeria/issues/1838

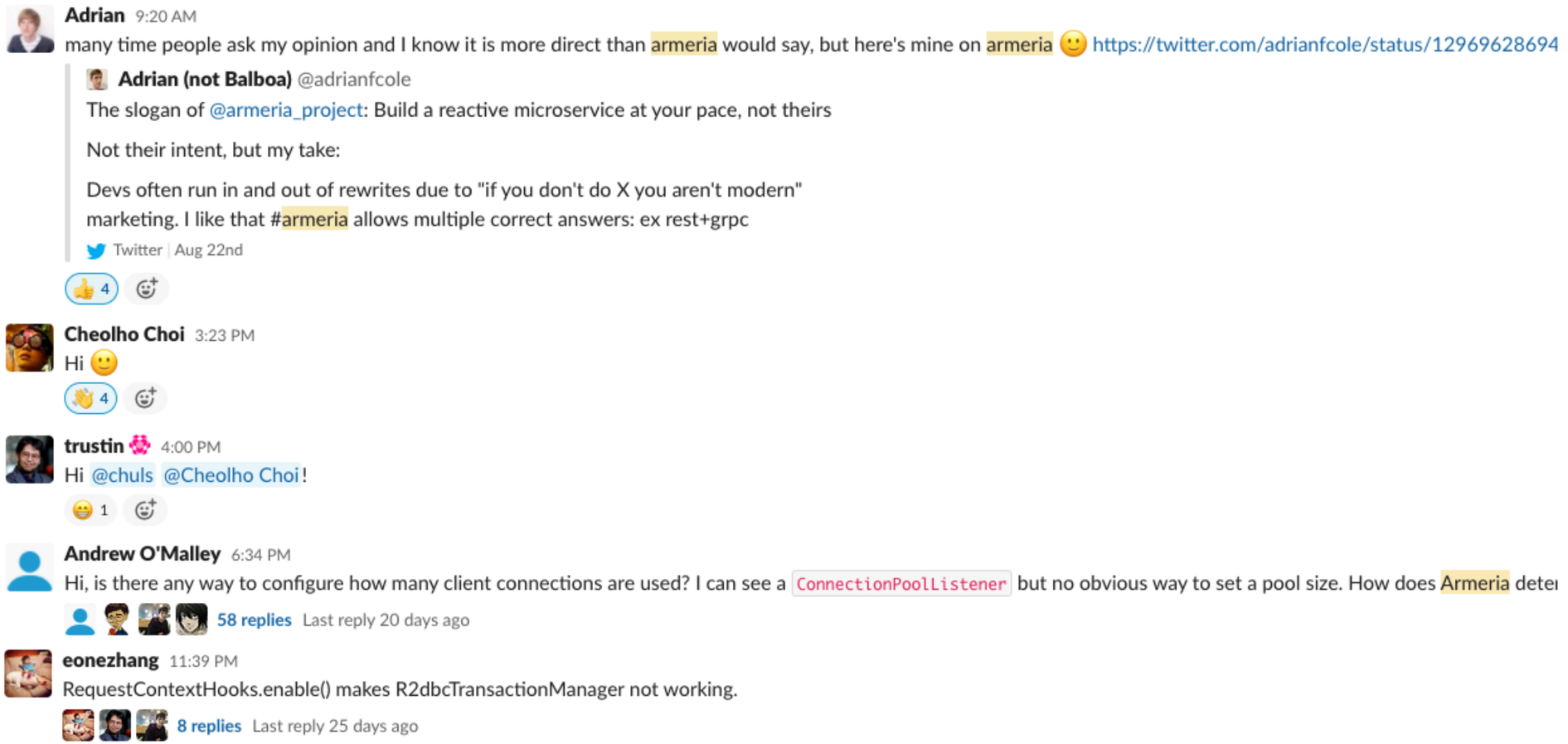


5. What's next?



We got our Slack!

DEVIEW 2020



@armeria_project line/armeria

We got our Slack! (Korean)

dino 4:01 PM
안녕하세요 아르메리아 적용을 위해서 문서를 보고 있습니다.
보던 중, CB 관련해서 문의 드립니다.
아르메리아의 Circuit Breaker 는 객체 단위로 설정이 되는걸로 봤는데요
Thread pool 없이 라이프 싸이클이 Request Thread 내에서만 동작하고 소멸되는 형태로
이해를 하면 되는지.. 궁금합니다. 써본 CB 라이브러리가 Hystrix 다보니
다소 생소해서 문의 드립니다 🙏
9 replies Last reply 1 month ago

eonezhang 5:19 PM
joined #general-ko along with 4 others.

Thursday, September 24th

Joonhaeng Lee 5:40 PM
혹시 클라이언트에서 Content-Encoding: gzip헤더붙여서 날아오는 gzip 인코딩된 post 본문데이터를 armeria+spring webflux 에서 받아서 디코딩하는 편한 방법이 있을까요?ㅎㅎ (edited)
199 replies Last reply 5 days ago

Friday, September 25th

sehajyang 11:34 AM
joined #general-ko along with ryan.

@armeria_project line/armeria

What's next

- A message the team leader got

  "...

  Finally, thank you so much for making Armeria. This saved development time so that I could make time for warm dinners with my family. Thank you again."

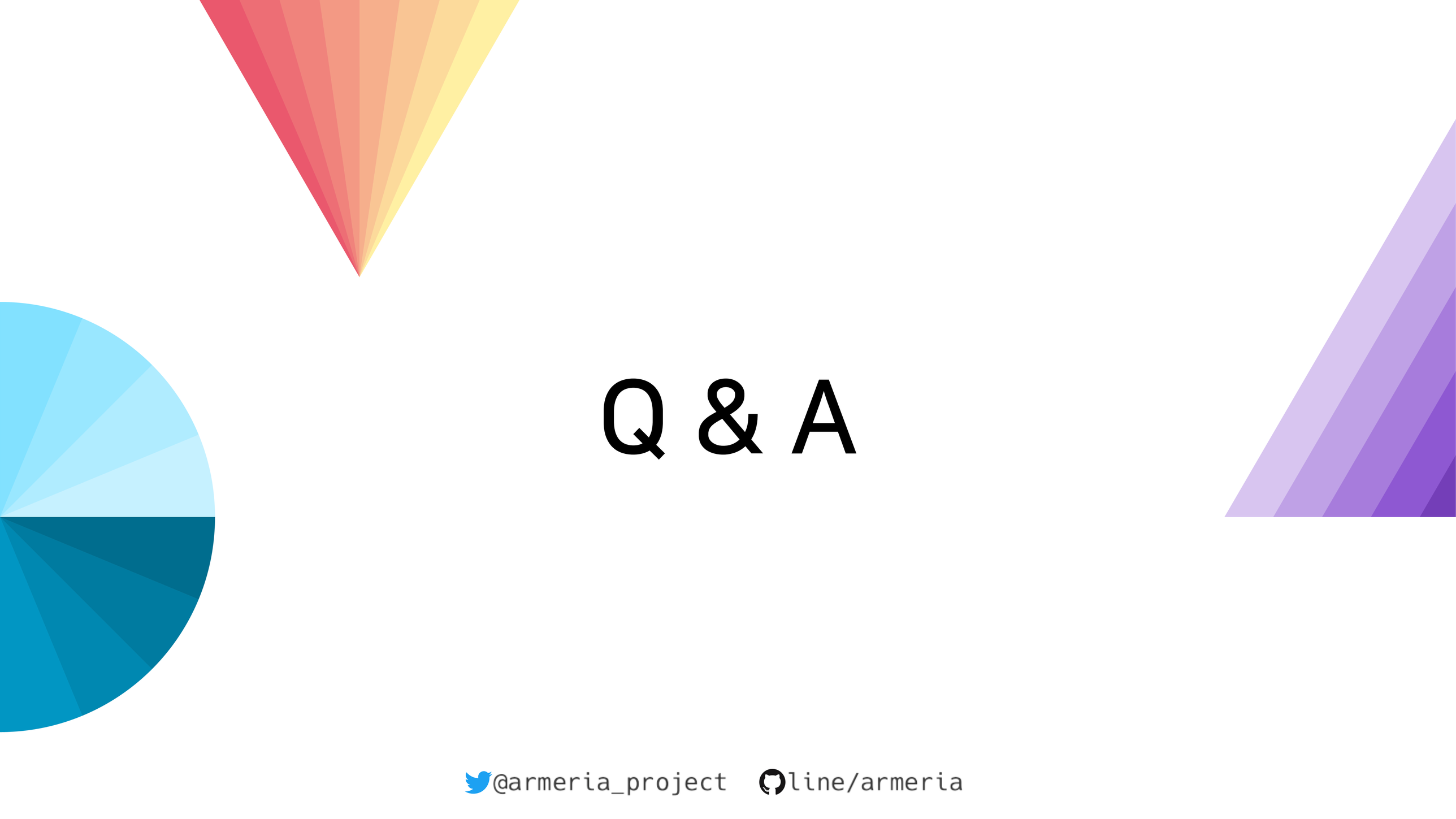
Meet us at GitHub & Slack

DEVIEW 2020

github.com/line/armeria

armeria.dev

@armeria_project line/armeria


Q & A

Thank You

@armeria_project line/armeria

Reactor and RxJava

DEVIEW 2020

- Armeria provides the global hooks.

```
// For Reactor
RequestContextHooks.enable();

// For RxJava
RequestContextAssembly.enable();
```

Mixed request scope

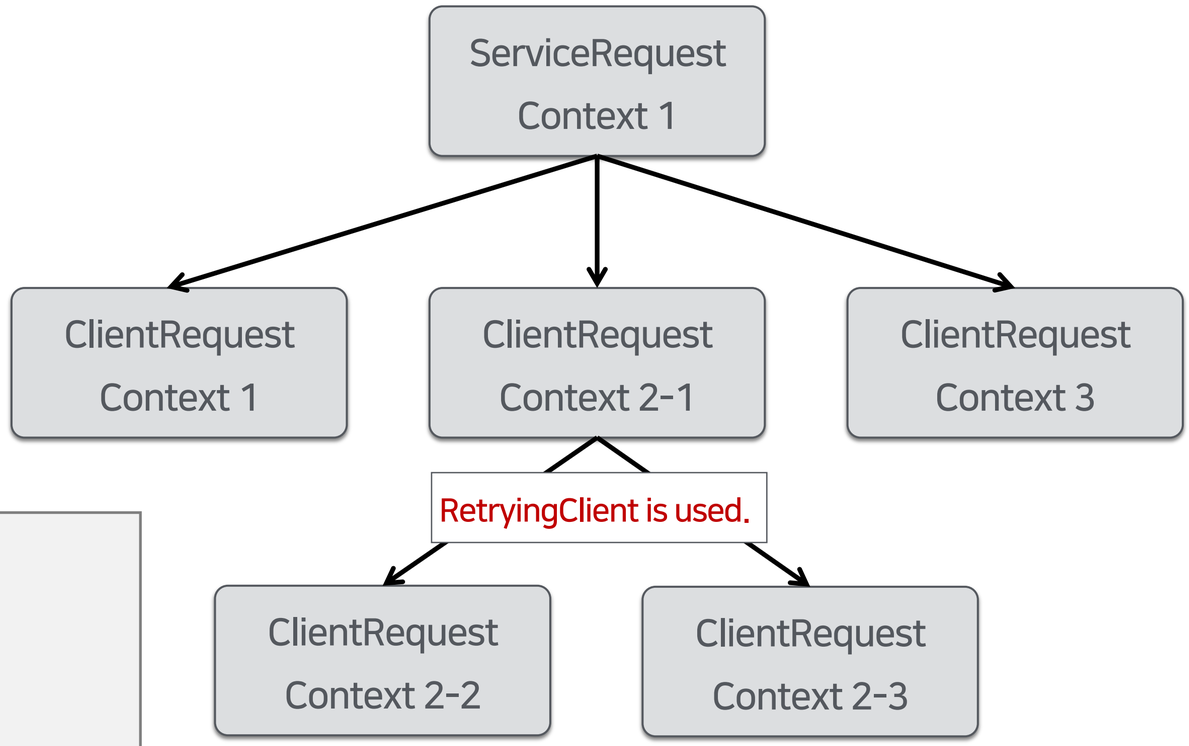
DEVIEW 2020

- Each request must have its own scope.
- If the request scope is mixed which is caused by context leak or coding mistakes, an `IllegalStateException` is raised.

```
ServiceRequstContext ctx1 = …;
try (SafeCloseable ignored = ctx1.push()) {
    …
    ServiceRequestContext ctx2 = …;
    try (SafeCloseable ignored = ctx2.push()) { // IllegalStateExcepion!
        …
    }
}
```

Mixed request scope

- We mange a request scope as a tree!
- A `ClientRequestContext` is created when the client sends a request to another backend.
- The `root()` of a `ClientRequestContext` is the `ServiceRequestContext` which the request is derived by.
- `ctx.push()` is valid if the `root()` of the old ctx and the new ctx are the same.




```
ServiceRequstContext serviceCtx1 = …;
try (SafeCloseable ignored = serviceCtx1.push()) {
    …
    ClientRequestContext clientCtx1 = …;
    assert clientCtx1.root() == serviceCtx1;
    try (SafeCloseable ignored = clientCtx1.push()) { // Valid!
        …
    }
}
```