Let's play Reactive Streams with Armeria

Ikhun Um, LINE

Agenda

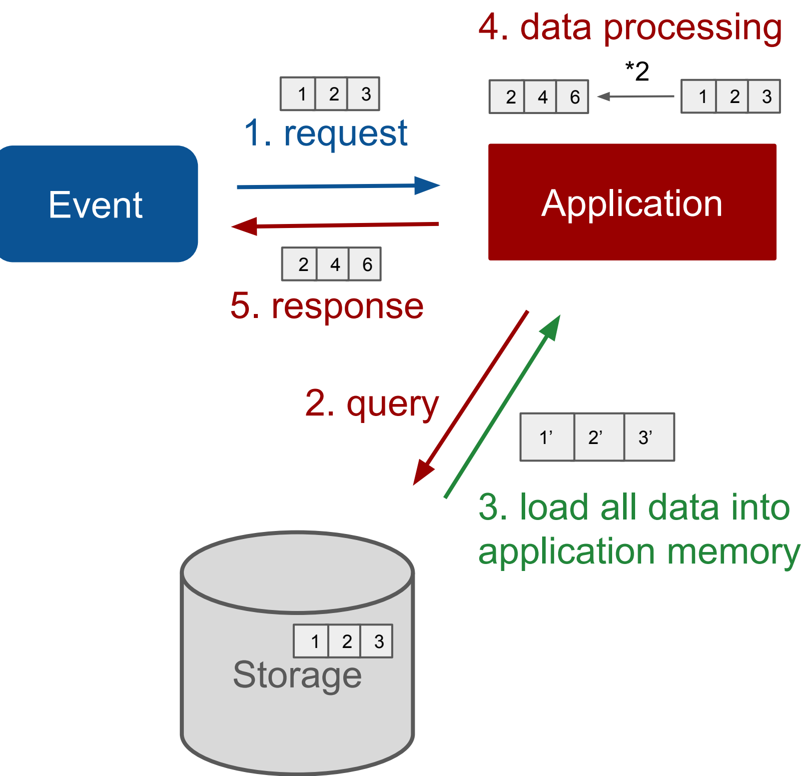
- Reactive Streams principles
- Reactive Streams API
- Reactive Streams interop
- What’s Armeria?
- Reactive Streams integration with Armeria
- gRPC stream support

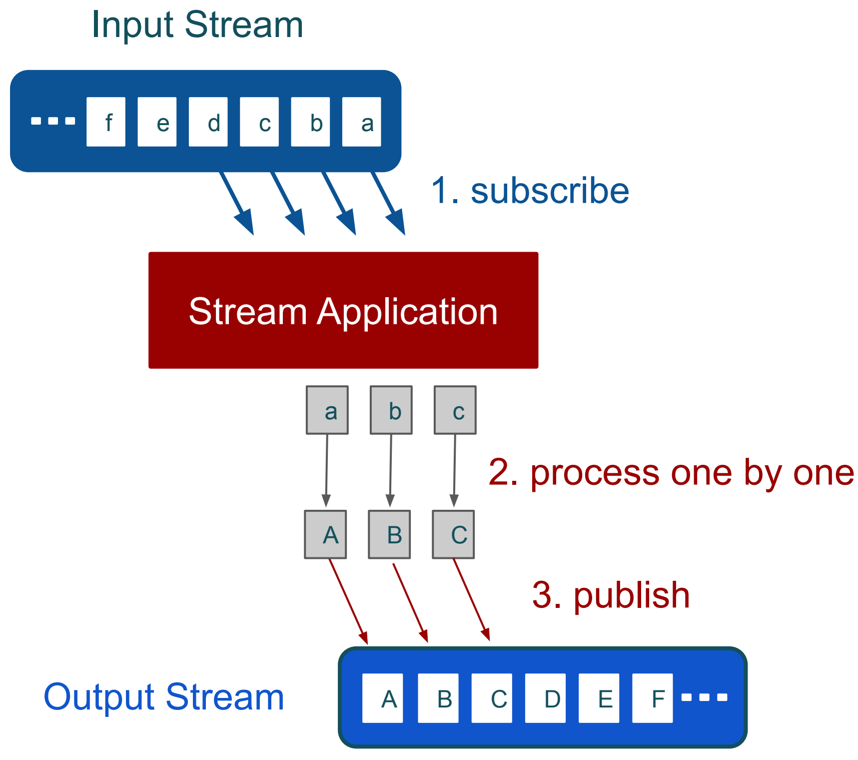
Reactive Streams

Reactive Streams is a standard for asynchronous data processing in a streaming fashion with non-blocking backpressure.

Reactive Streams

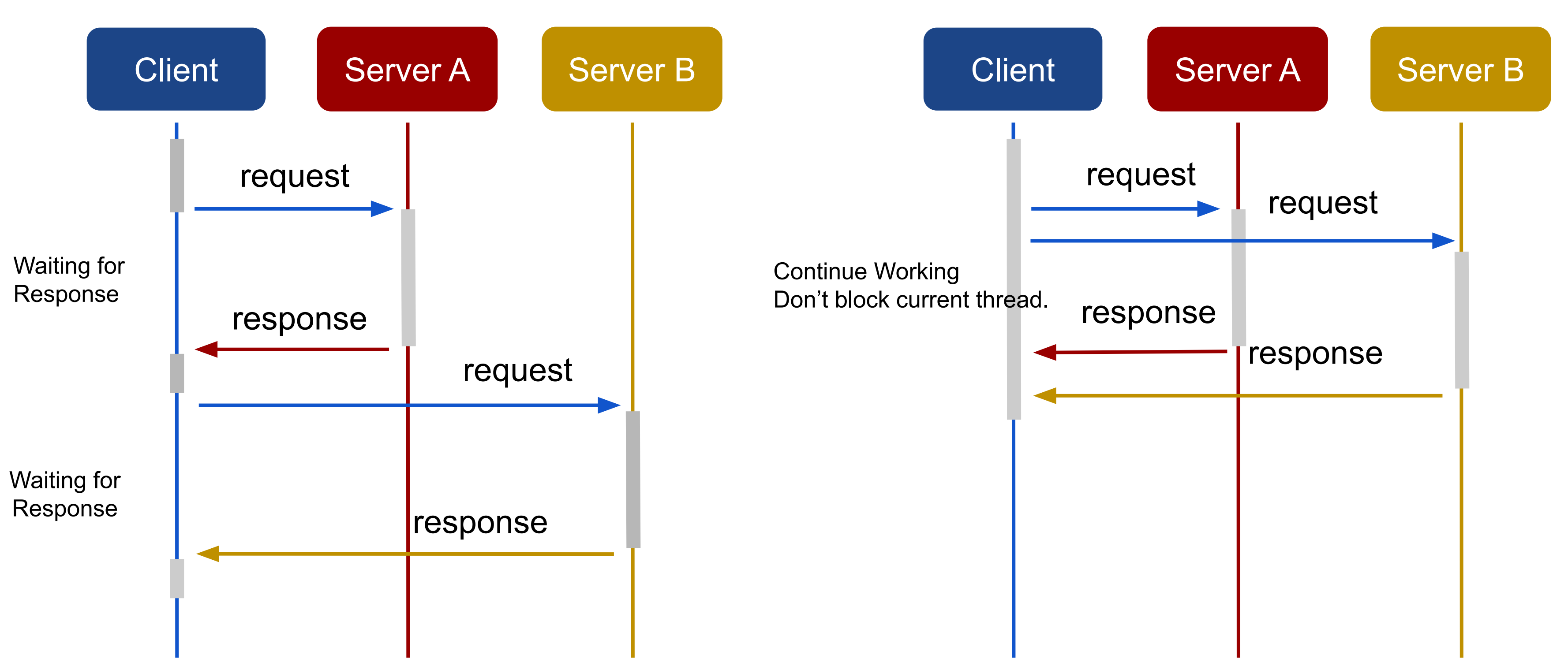
Reactive Streams is a standard for asynchronous data processing in a streaming fashion with non-blocking backpressure.


4. data processing
*2
2 4 6
1 2 3
1 2 3
1. request
Event
Application
2 4 6
5. response
2. query
1' 2' 3'
3. load all data into application memory
1 2 3
Storage
Traditional Data Processing


Input Stream
f e d c b a
1. subscribe
Stream Application
a b c
2. process one by one
A B C
3. publish
Output Stream
A B C D E F
Stream Processing

Reactive Streams

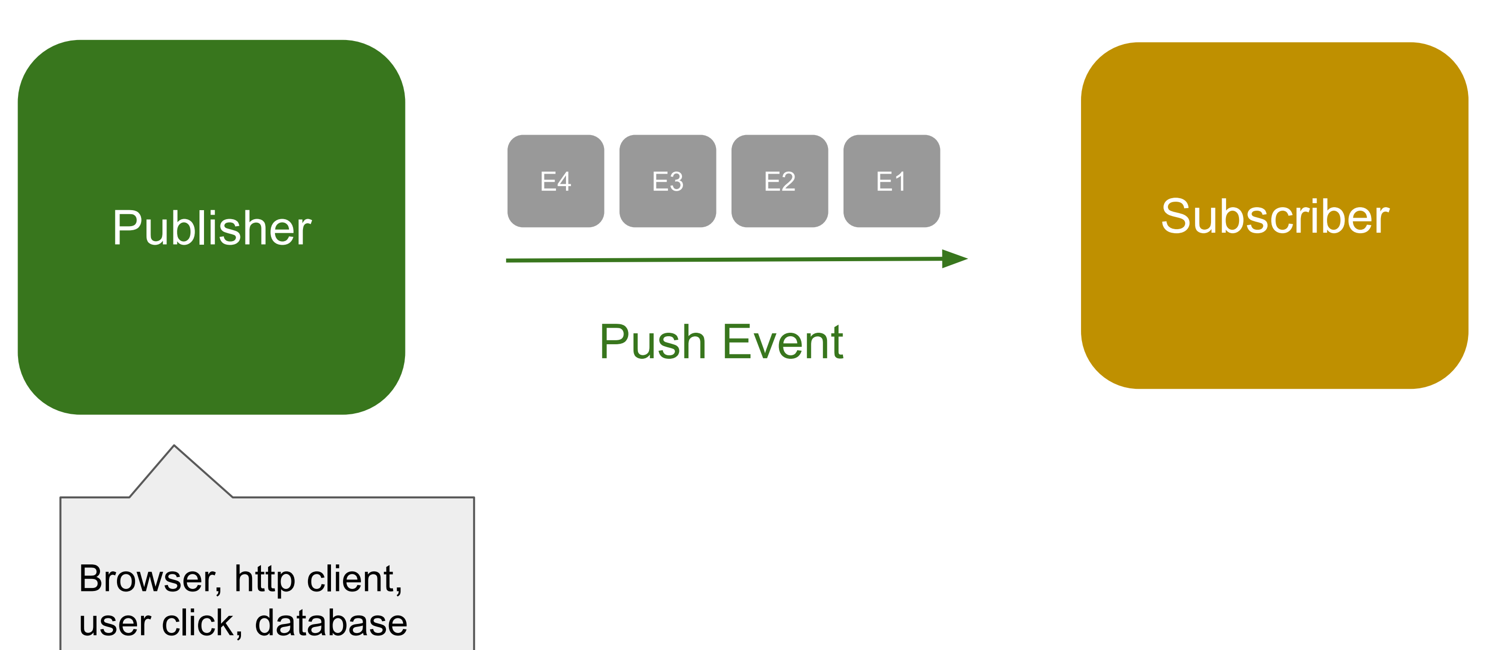
Reactive Streams is a standard for **asynchronous** data processing in a streaming fashion with non-blocking backpressure.


Synchronous
Asynchronous
Client
Server A
Server B
request
Waiting for
Response
response
request
Waiting for
Response
response
Client
Server A
Server B
request
request
Continue Working
Don’t block current thread.
response
response

Reactive Streams

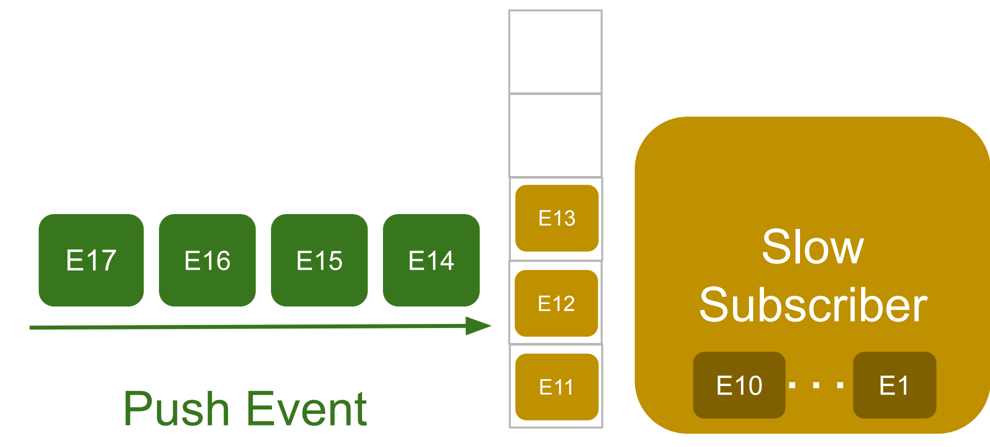
Reactive Streams is a standard for asynchronous data processing in a streaming fashion with non-blocking **backpressure**.

Push Model



Push Model

use buffer for pending events



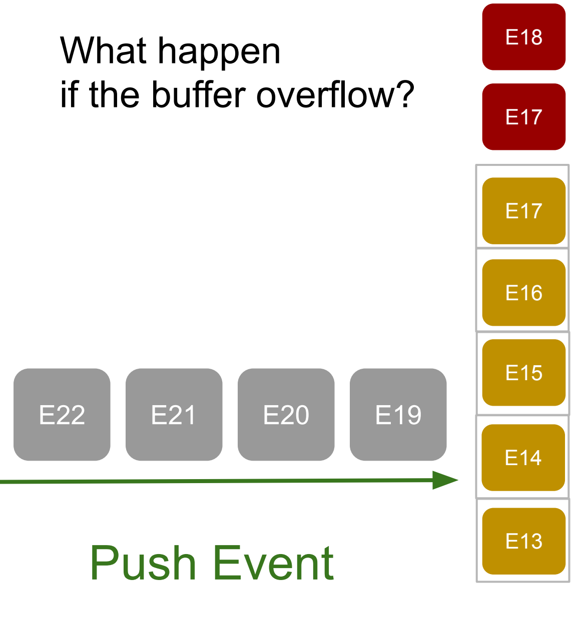
100 ops/sec

push too many events to subscriber

10 ops/sec

Push Model

100 ops/sec



Slow
Subscriber

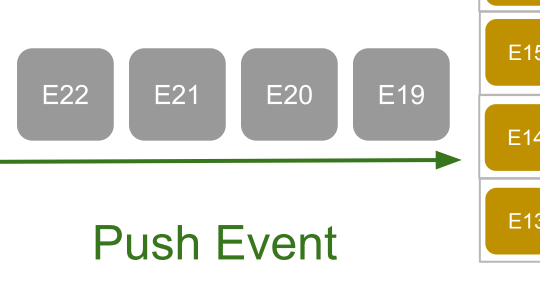
E12 . . . E3

10 ops/sec

Push Model

If use bounded buffer, drop messages

100 ops/sec



10 ops/sec

Push Model

If use unbounded buffer, out of memory

Fast Publisher

E22 E21 E20 E19

Push Event

E17
E16
E15
E14
E13

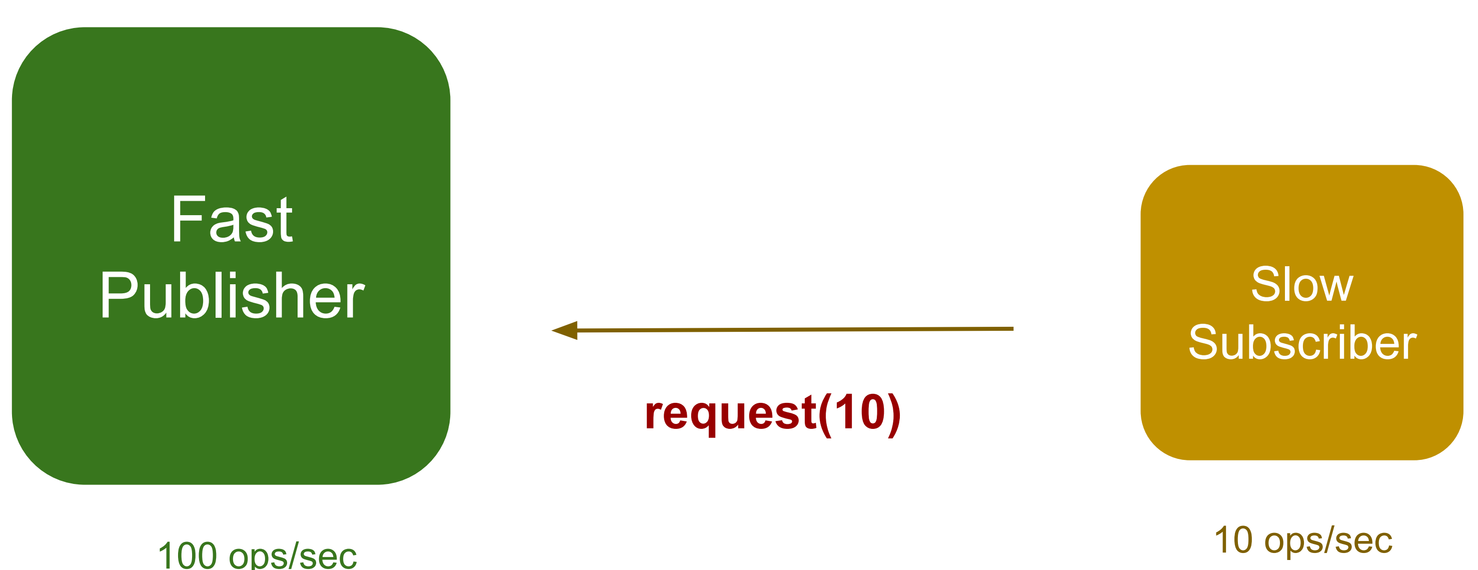
Slow Subscriber

E12 · · · E3

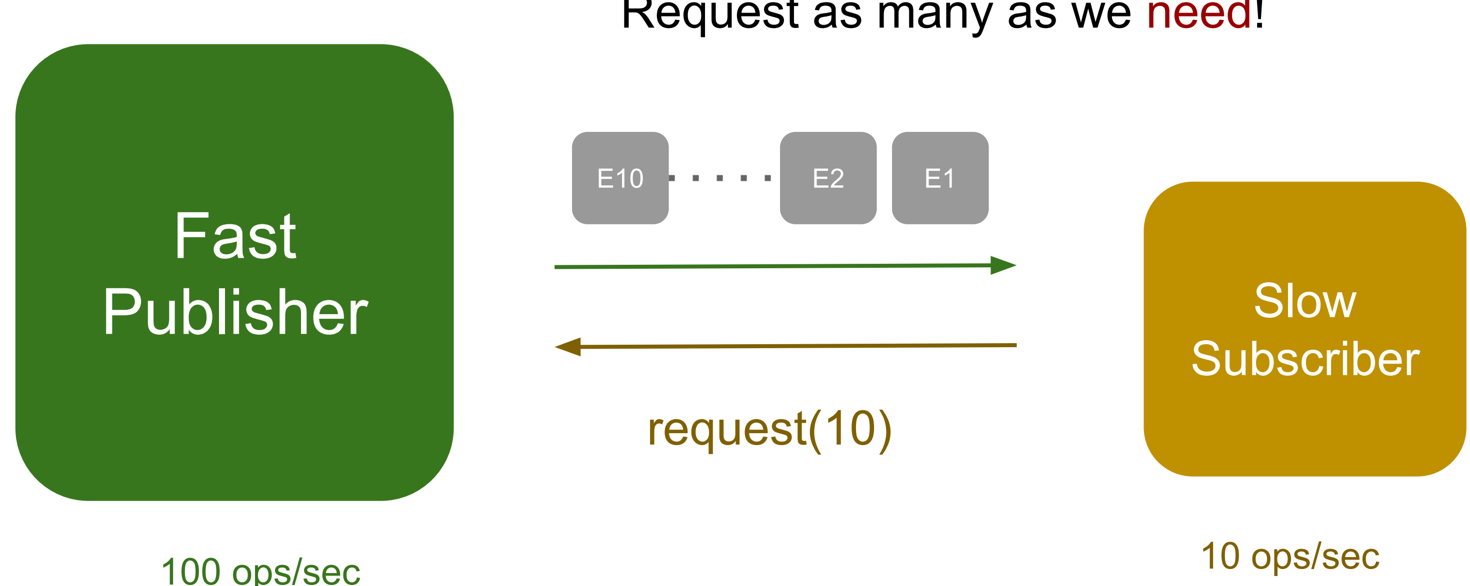
100 ops/sec

10 ops/sec

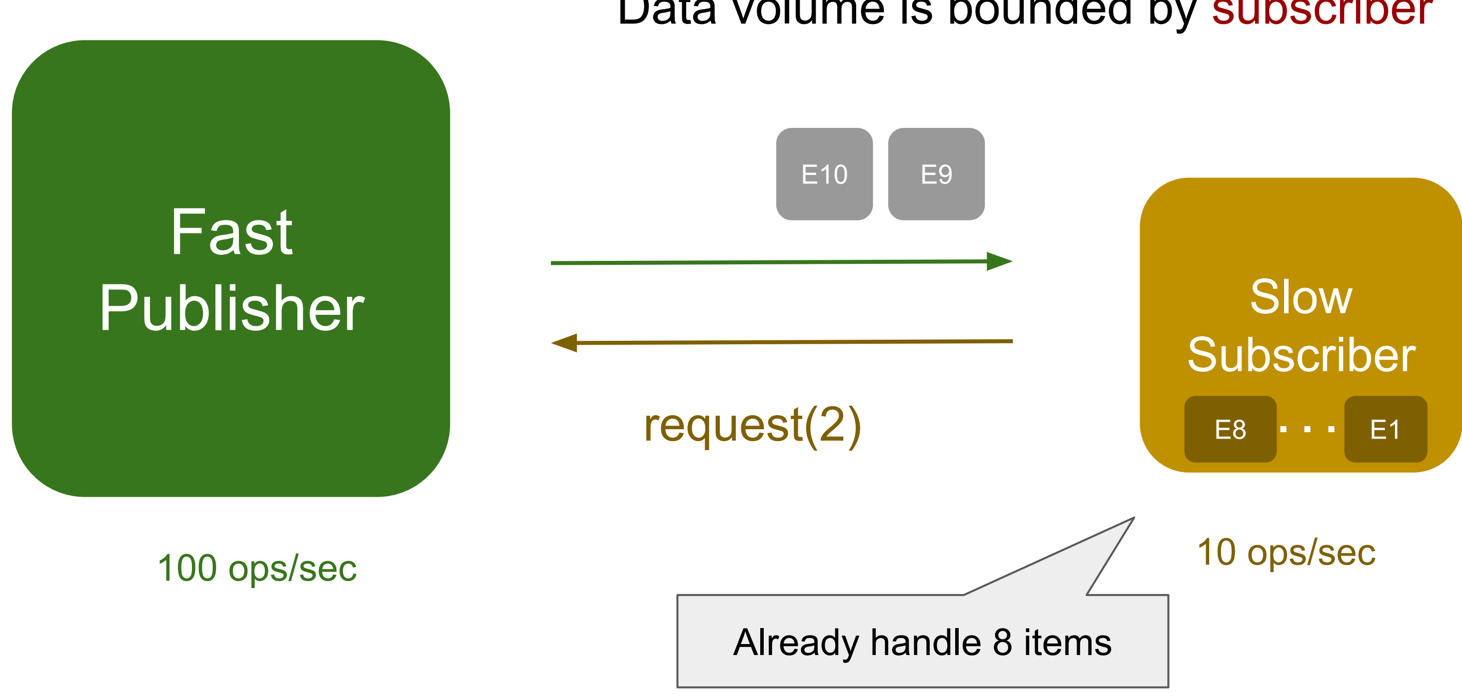
Pull Model



Dynamic Pull - request(10)

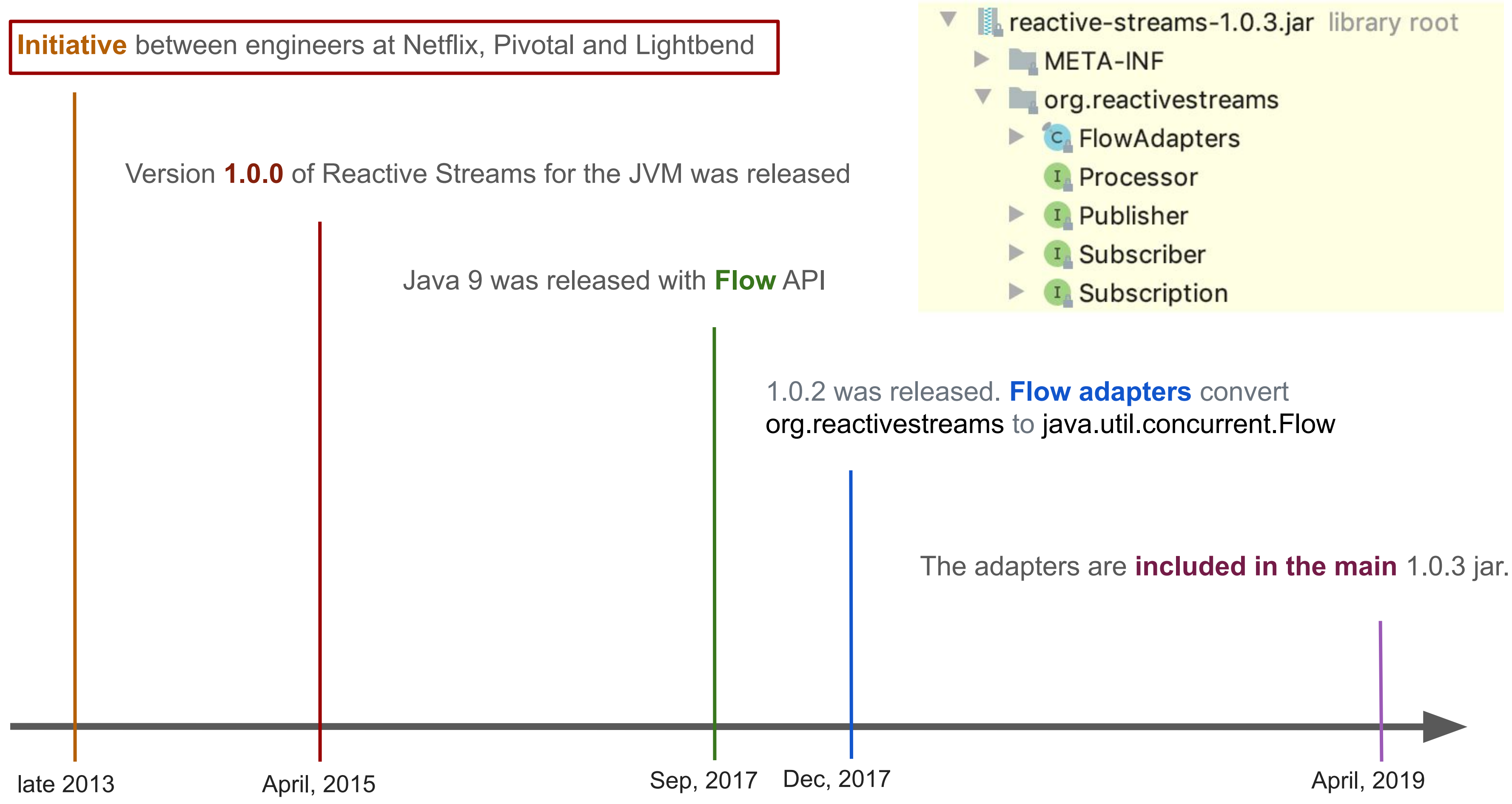


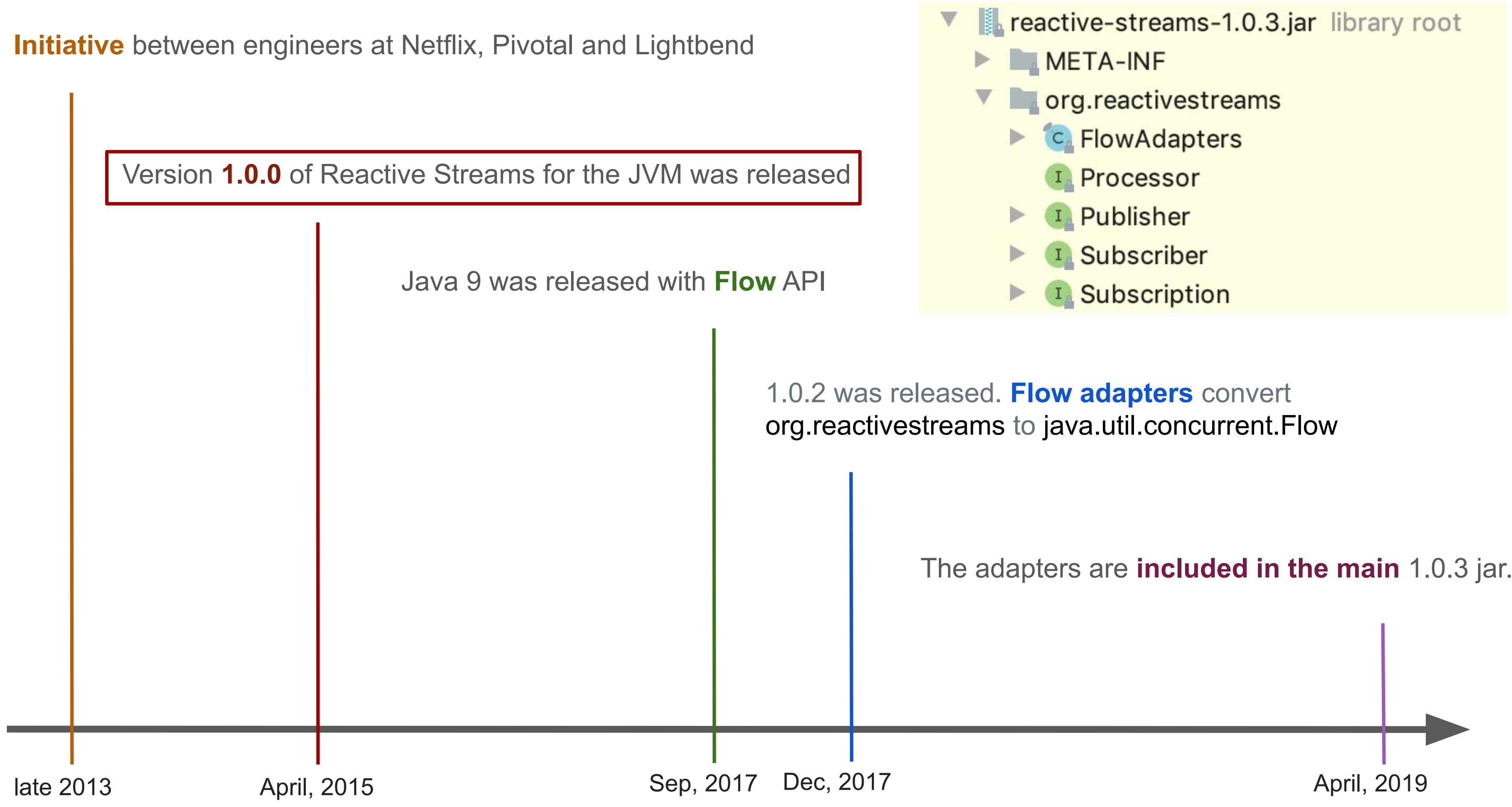
Dynamic Pull - request(2)

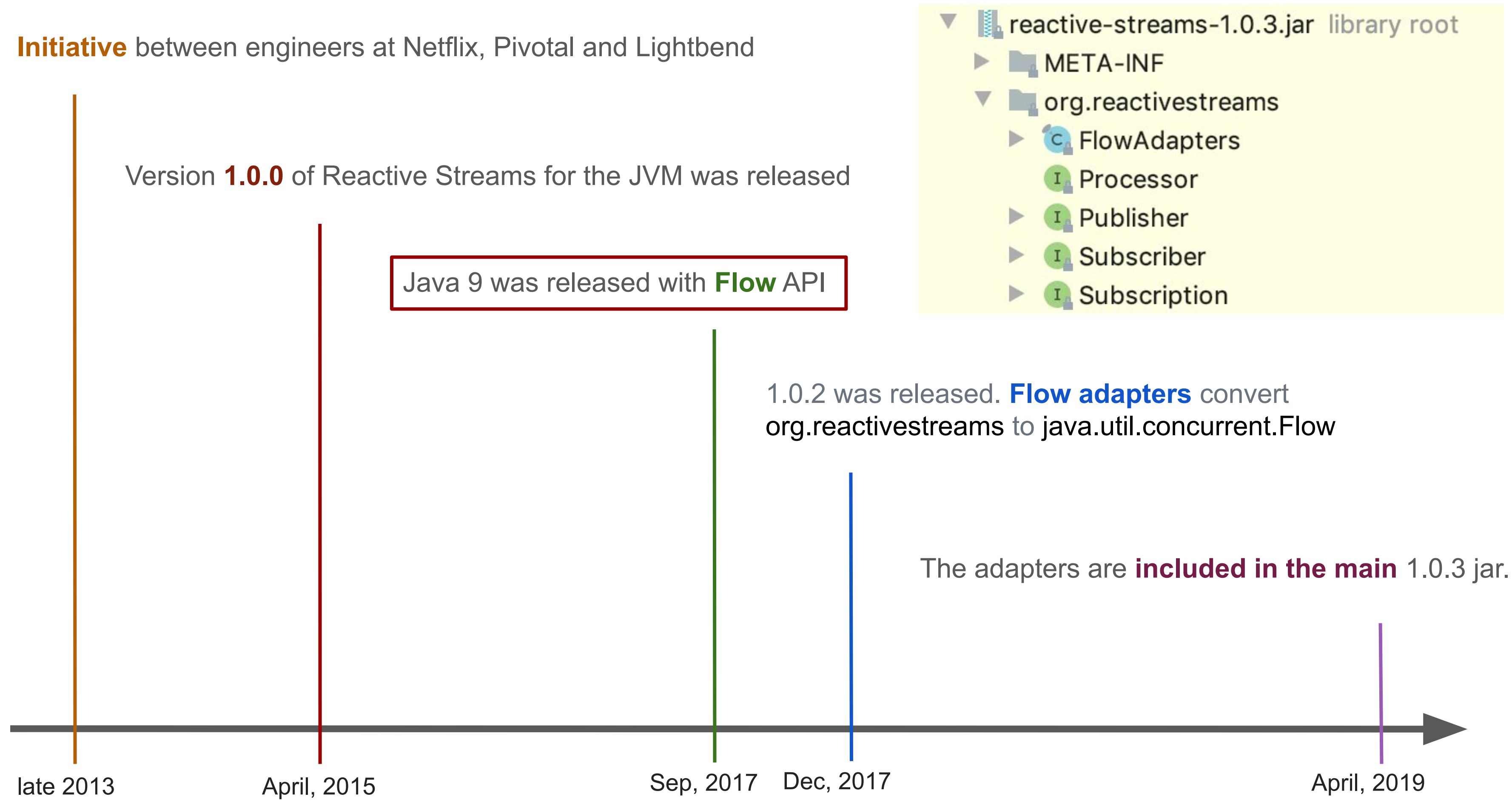


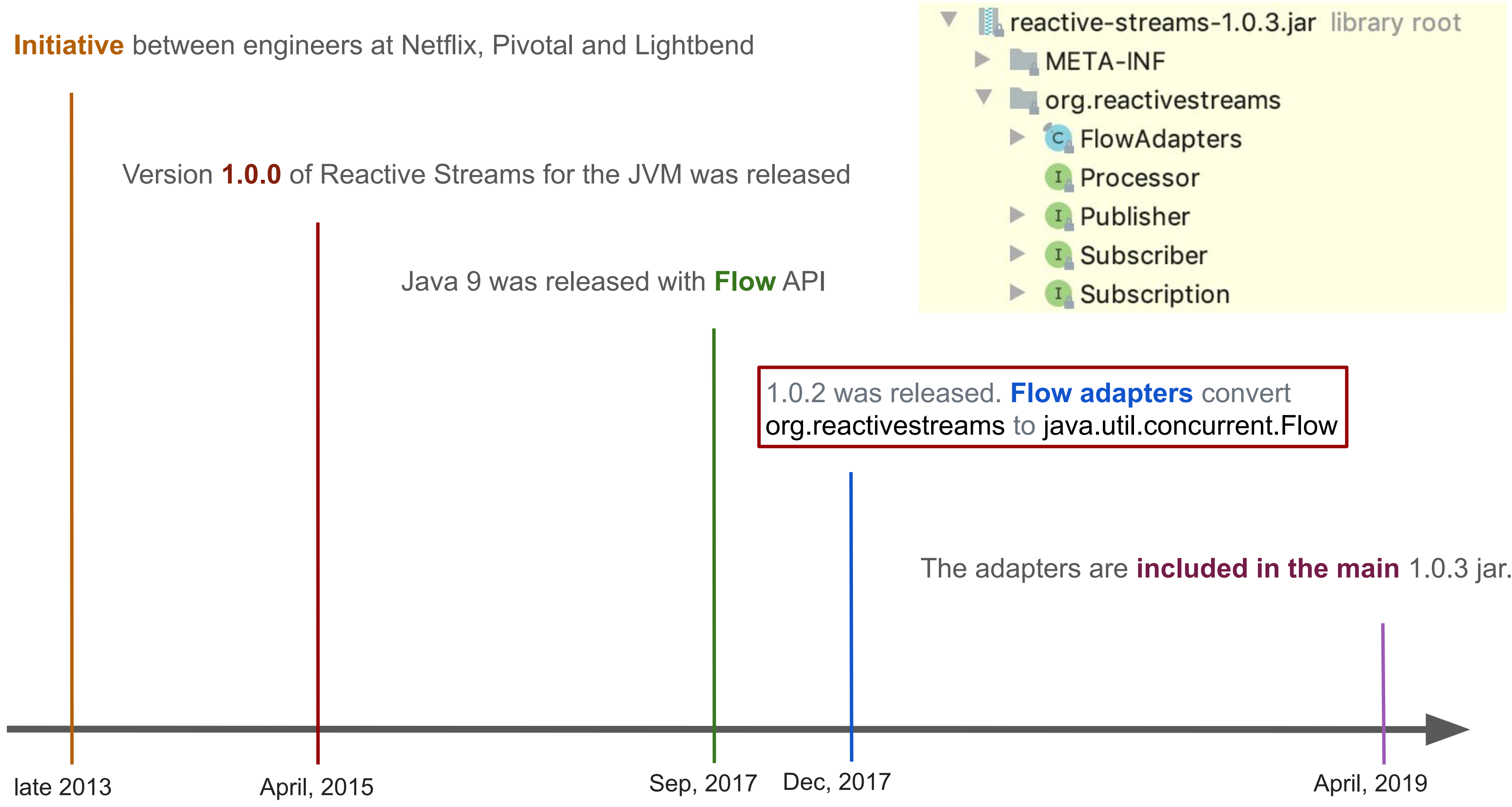
Reactive Streams

Reactive Streams is a standard for asynchronous data processing in a streaming fashion with non-blocking backpressure.

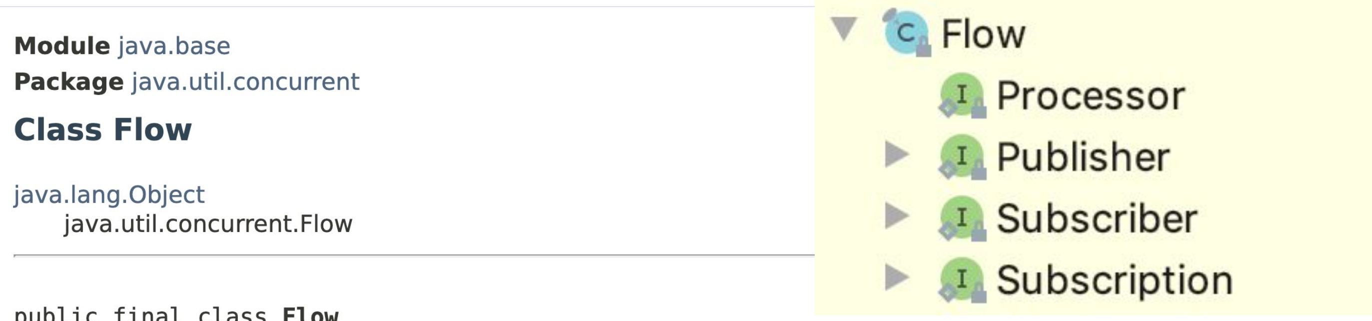

Initiative between engineers at Netflix, Pivotal and Lightbend
Version 1.0.0 of Reactive Streams for the JVM was released
Java 9 was released with Flow API
1.0.2 was released. Flow adapters convert
org.reactivestreams to java.util.concurrent.Flow
The adapters are included in the main 1.0.3 jar.
late 2013
April, 2015
Sep, 2017
Dec, 2017
April, 2019
reactive-streams-1.0.3.jar library root
META-INF
org.reactivestreams
FlowAdapters
Processor
Publisher
Subscriber
Subscription


Initiative between engineers at Netflix, Pivotal and Lightbend
Version 1.0.0 of Reactive Streams for the JVM was released
Java 9 was released with Flow API
1.0.2 was released. Flow adapters convert org.reactivestreams to java.util.concurrent.Flow
The adapters are included in the main 1.0.3 jar.
reactive-streams-1.0.3.jar library root
META-INF
org.reactivestreams
FlowAdapters
Processor
Publisher
Subscriber
Subscription
late 2013
April, 2015
Sep, 2017
Dec, 2017
April, 2019


Initiative between engineers at Netflix, Pivotal and Lightbend
Version 1.0.0 of Reactive Streams for the JVM was released
Java 9 was released with Flow API
1.0.2 was released. Flow adapters convert
org.reactivestreams to java.util.concurrent.Flow
The adapters are included in the main 1.0.3 jar.
reactive-streams-1.0.3.jar library root
META-INF
org.reactivestreams
FlowAdapters
Processor
Publisher
Subscriber
Subscription
late 2013
April, 2015
Sep, 2017
Dec, 2017
April, 2019


Initiative between engineers at Netflix, Pivotal and Lightbend
Version 1.0.0 of Reactive Streams for the JVM was released
Java 9 was released with Flow API
1.0.2 was released. Flow adapters convert org.reactivestreams to java.util.concurrent.Flow
The adapters are included in the main 1.0.3 jar.
reactive-streams-1.0.3.jar library root
META-INF
org.reactivestreams
FlowAdapters
Processor
Publisher
Subscriber
Subscription
late 2013
April, 2015
Sep, 2017
Dec, 2017
April, 2019

Starting from Java 9, they have become a part of the JDK in the form of the java.util.concurrent.Flow.* interfaces.



Module java.base
Package java.util.concurrent

Class Flow

java.lang.Object
    java.util.concurrent.Flow

```
public final class Flow
extends Object
```

Interrelated interfaces and static methods for establishing flow-controlled components in which `Publishers` produce items consumed by one or more `Subscribers`, each managed by a `Subscription`.

These interfaces correspond to the reactive-streams specification. They apply in both concurrent and distributed asynchronous settings: All (seven) methods are defined in `void` "one-way" message style. Communication relies on a simple form of flow control (method `Flow.Subscription.request(long)`) that can be used to avoid resource management problems that may otherwise occur in "push" based systems.

Reactive Streams API

```java
public interface Publisher<T> {
   public void subscribe(Subscriber<? super T> s);
}
```

```java
public interface Subscription {
   public void request(long n);
   public void cancel();
}
```

```java
public interface Subscriber<T> {
   public void onSubscribe(Subscription s);
   public void onNext(T t);
   public void onError(Throwable t);
   public void onComplete();
}
```

```java
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

```java
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

```java
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

```java
public interface Publisher<T> {
   public void subscribe(Subscriber<? super T> s);
}
```

```java
public interface Subscription {
   public void request(long n);
   public void cancel();
}
```

```java
public interface Subscriber<T> {
   public void onSubscribe(Subscription s);
   public void onNext(T t);
   public void onError(Throwable t);
   public void onComplete();
}
```

Subscribe Streams

1. subscribe

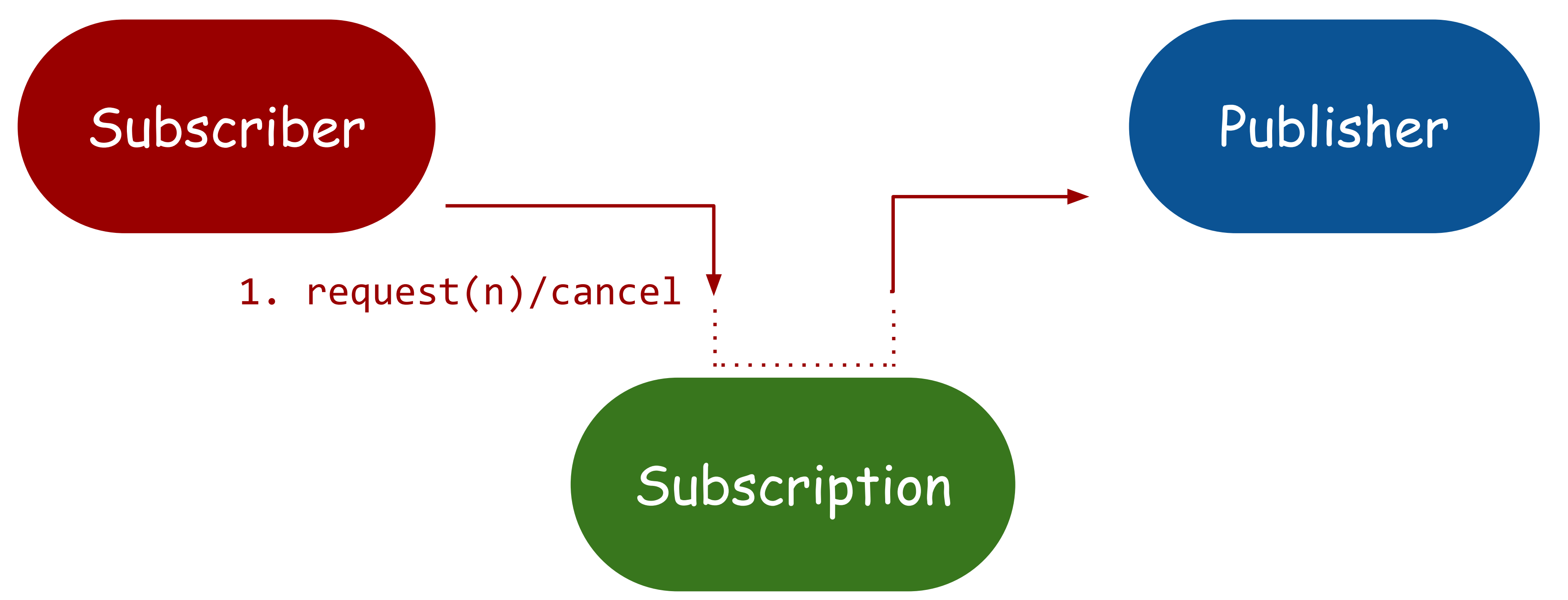
Subscriber

Publisher

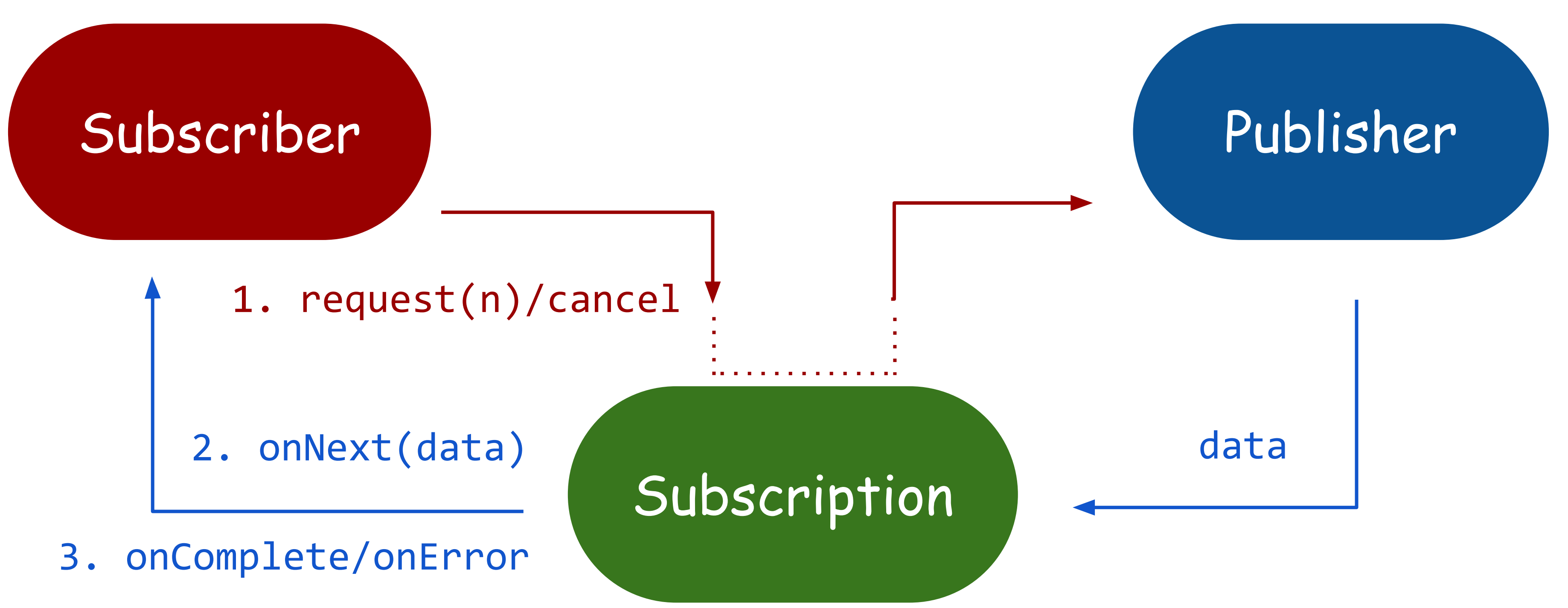
Subscription

Subscribe Streams

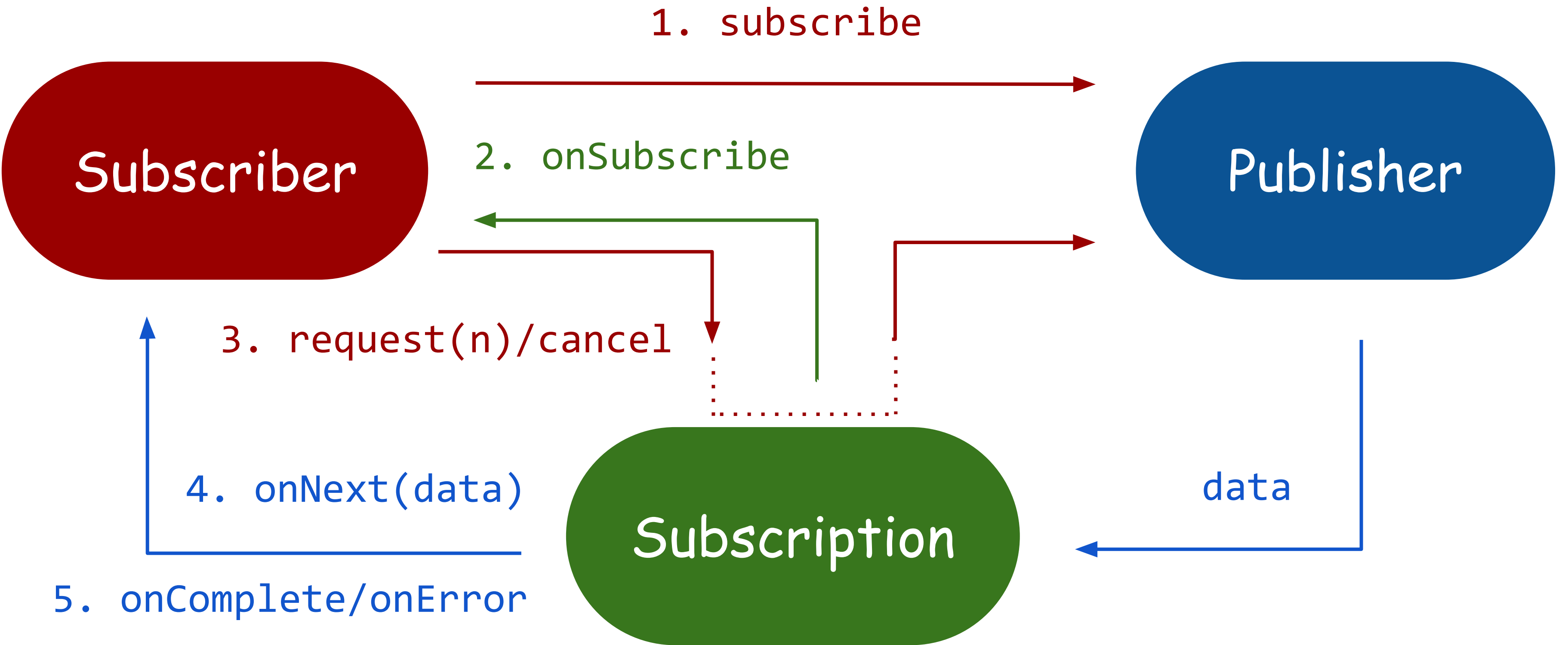
Request(n) Streams



Request(n) Streams



Put them all together



How to make Publisher?

```
Subscriber<Event> subscriber = ...;
Publisher<Event> publisher = new Publisher<Event>() {
    @Override
    public void subscribe(Subscriber<? super Event> subscriber) {
        Subscription subscription = ...;
        // 2. give subscription to subscriber
        subscriber.onSubscribe(subscription);
    }
};
// 1. subscribe stream events
publisher.subscribe(subscriber);
```

1. subscribe

Subscriber

Publisher

2. onSubscribe

Publisher should give Subscription to Subscriber via onSubscribe method

Subscription

Reactive Streams Specification

1. Publisher (Code)

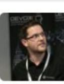
```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

API is simple, but it has lots of rules.

Reactive Streams Technology Compatibility Kit(https://github.com/reactive-streams/reactive-streams-jvm/tree/master/tck) helps Reactive Streams library implementers to **validate** their implementations against **the rules**.

ID	
1	The total number of `onNext`´s signalled by a `Publisher` to a `Subscriber` MUST be less than or equal to the total number of elements requested by that `Subscriber`´s `Subscription` at all times.
💡	*The intent of this rule is to make it clear that Publishers cannot signal more elements than Subscribers have requested. There's an implicit, but important, consequence to this rule: Since demand can only be fulfilled after it has been received, there's a happens-before relationship between requesting elements and receiving elements.*
2	A `Publisher` MAY signal fewer `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError`.
💡	*The intent of this rule is to make it clear that a Publisher cannot guarantee that it will be able to produce the number of elements requested; it simply might not be able to produce them all; it may be in a failed state; it may be empty or otherwise already completed.*
3	`onSubscribe`, `onNext`, `onError` and `onComplete` signaled to a `Subscriber` MUST be signaled serially.
💡	*The intent of this rule is to permit the signalling of signals (including from multiple threads) if and only if a happens-before relation between each of the signals is established.*
4	If a `Publisher` fails it MUST signal an `onError`.
💡	*The intent of this rule is to make it clear that a Publisher is responsible for notifying its Subscribers if it detects that it cannot proceed—Subscribers must be given a chance to clean up resources or otherwise deal with the Publisher´s failures.*
5	If a `Publisher` terminates successfully (finite stream) it MUST signal an `onComplete`.
💡	*The intent of this rule is to make it clear that a Publisher is responsible for notifying its Subscribers that it has reached a terminal state—Subscribers can then act on this information; clean up resources, etc.*
6	If a `Publisher` signals either `onError` or `onComplete` on a `Subscriber`, that `Subscriber`'s `Subscription` MUST be considered cancelled.
💡	*The intent of this rule is to make sure that a Subscription is treated the same no matter if it was cancelled, the Publisher signalled onError or onComplete.*
7	Once a terminal state has been signaled (`onError`, `onComplete`) it is REQUIRED that no further signals occur.

Closed **Custom Publisher and potential flatMap bug** #482
ikajinic opened this issue on Mar 16, 2017 · 5 comments



simonbasle on Mar 16, 2017 • edited · Contributor

Rule number 1 of the Reactor club: don't write your own `Publisher`. Even though the **interface** is simple, the set of rules about interactions between all these reactive streams interface **is not**.

Here for example, there are several issues:

1. your publisher never completes (might be intentional, but I don't think so)
2. your publisher short-circuits the subscription process. it should call `onSubscribe` on the `Subscriber` and pass it a `Subscription` the subscriber can use to request data or cancel
3. without the subscription, or if the publisher ignores the subscription's `request()` calls, the publisher ignores backpressure.
4. your test `expected` is too broad, it could be accepting any kind of exception. better to narrow it inside a `try { ...; fail(); } catch { ... }` block.

....

The `ErrorCallbackNotImplemented` comes from the fact that in Handler your subscribe doesn't define what to do with each item and in case of errors.

In Conclusion: Do not roll your own `Publisher`

I'd recommend that you look at `Flux.create()` (if you want to bridge an existing API into a `Flux`) or various `Processors` (if you just want to feed values to something manually and have these values represented as a `Flux`).

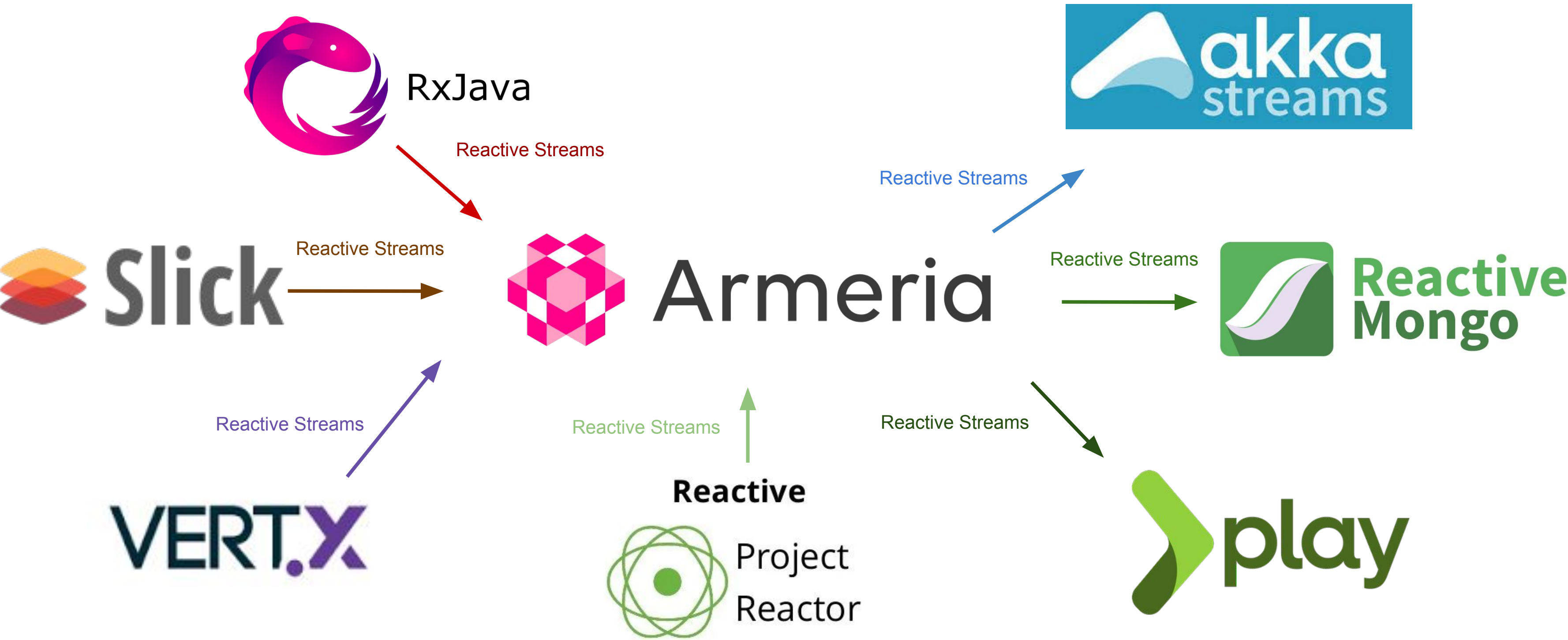
1 3 1

https://github.com/reactor/reactor-core/issues/482

Reactive Streams Implementations

See more implementations at https://en.wikipedia.org/wiki/Reactive_Streams#Adoption

Reactive Streams Interop



Reactive Streams make different implementations communicate with each other

Reactive Streams Interop by Example

```java
// Initiate MongoDB FindPublisher
FindPublisher<Document> mongoDBUsers = mongodbCollection.find();

// MongoDB FindPublisher -> RxJava Observable
Observable<Integer> rxJavaAllAges =
    Observable.fromPublisher(mongoDBUsers)
              .map(document -> document.getInteger("age"));
```

MongoDB

↓

RxJava

Reactive Streams Interop by Example

```
// Initiate MongoDB FindPublisher
FindPublisher<Document> mongoDBUsers = mongodbCollection.find();

// MongoDB FindPublisher -> RxJava Observable
Observable<Integer> rxJavaAllAges =
    Observable.fromPublisher(mongoDBUsers)
              .map(document -> document.getInteger("age"));

// RxJava Observable -> Reactor Flux
Flux<HttpData> fluxHttpData =
    Flux.from(rxJavaAllAges.toFlowable(BackpressureStrategy.DROP))
        .map(age -> HttpData.ofAscii(age.toString()));
```

MongoDB

↓

RxJava

↓

Reactor

Reactive Streams Interop by Example

```
// Initiate MongoDB FindPublisher
FindPublisher<Document> mongoDBUsers = mongodbCollection.find();

// MongoDB FindPublisher -> RxJava Observable
Observable<Integer> rxJavaAllAges =
    Observable.fromPublisher(mongoDBUsers)
              .map(document -> document.getInteger("age"));

// RxJava Observable -> Reactor Flux
Flux<HttpData> fluxHttpData =
    Flux.from(rxJavaAllAges.toFlowable(BackpressureStrategy.DROP))
        .map(age -> HttpData.ofAscii(age.toString()));

// Reactor Flux -> Armeria HttpResponse
HttpResponse.of(Flux.concat(httpHeaders, fluxHttpData));
```

MongoDB

↓

RxJava

↓

Reactor

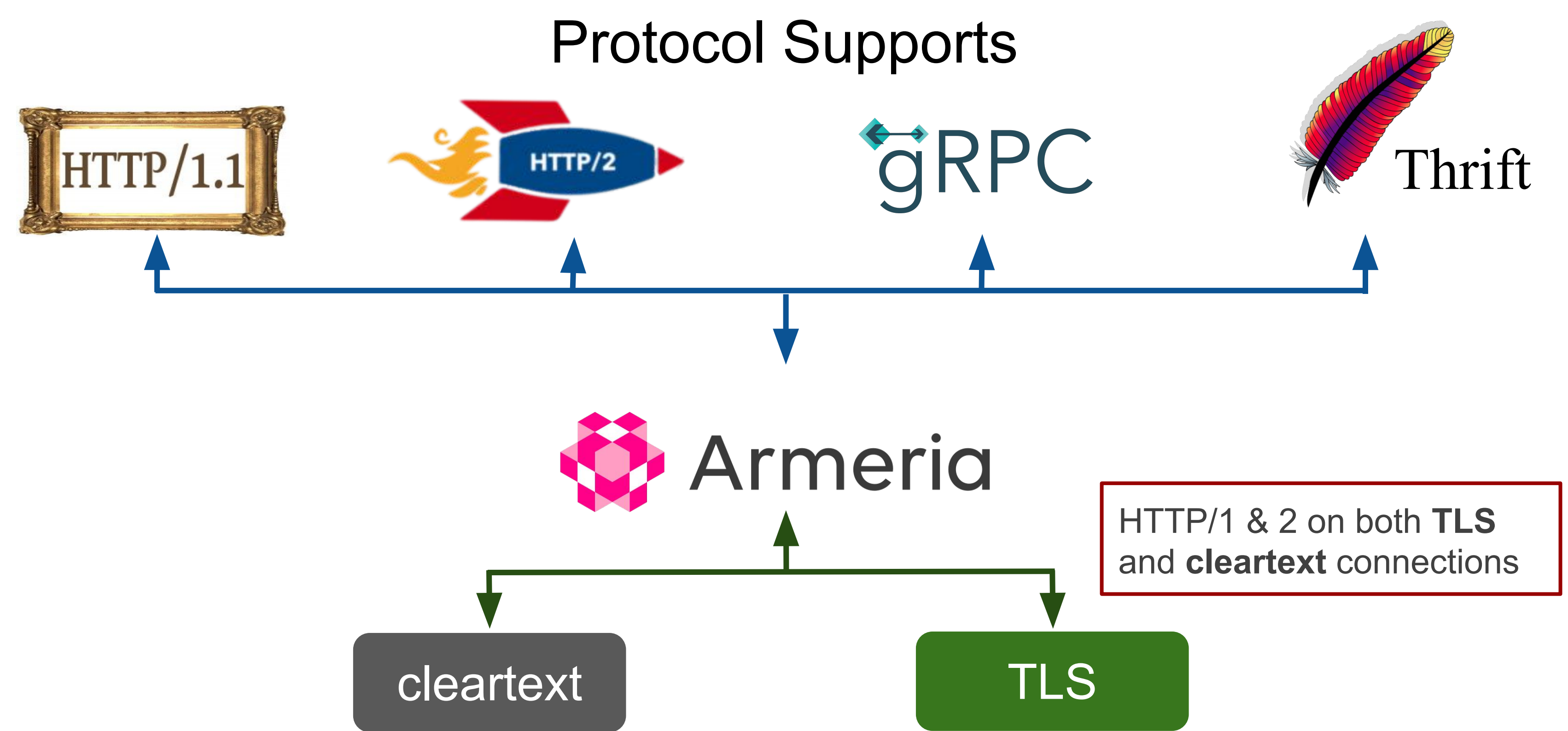
↓

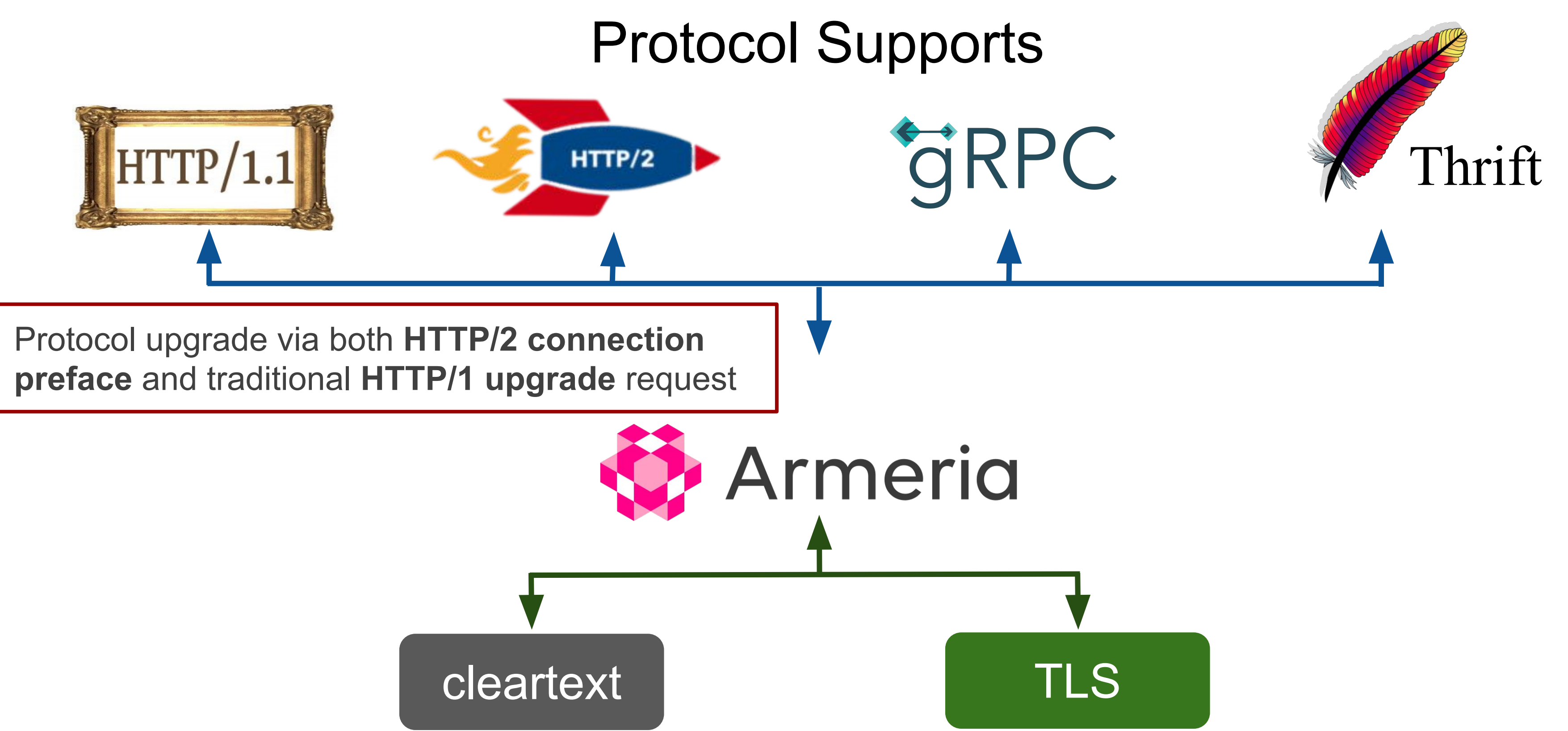
Armeria

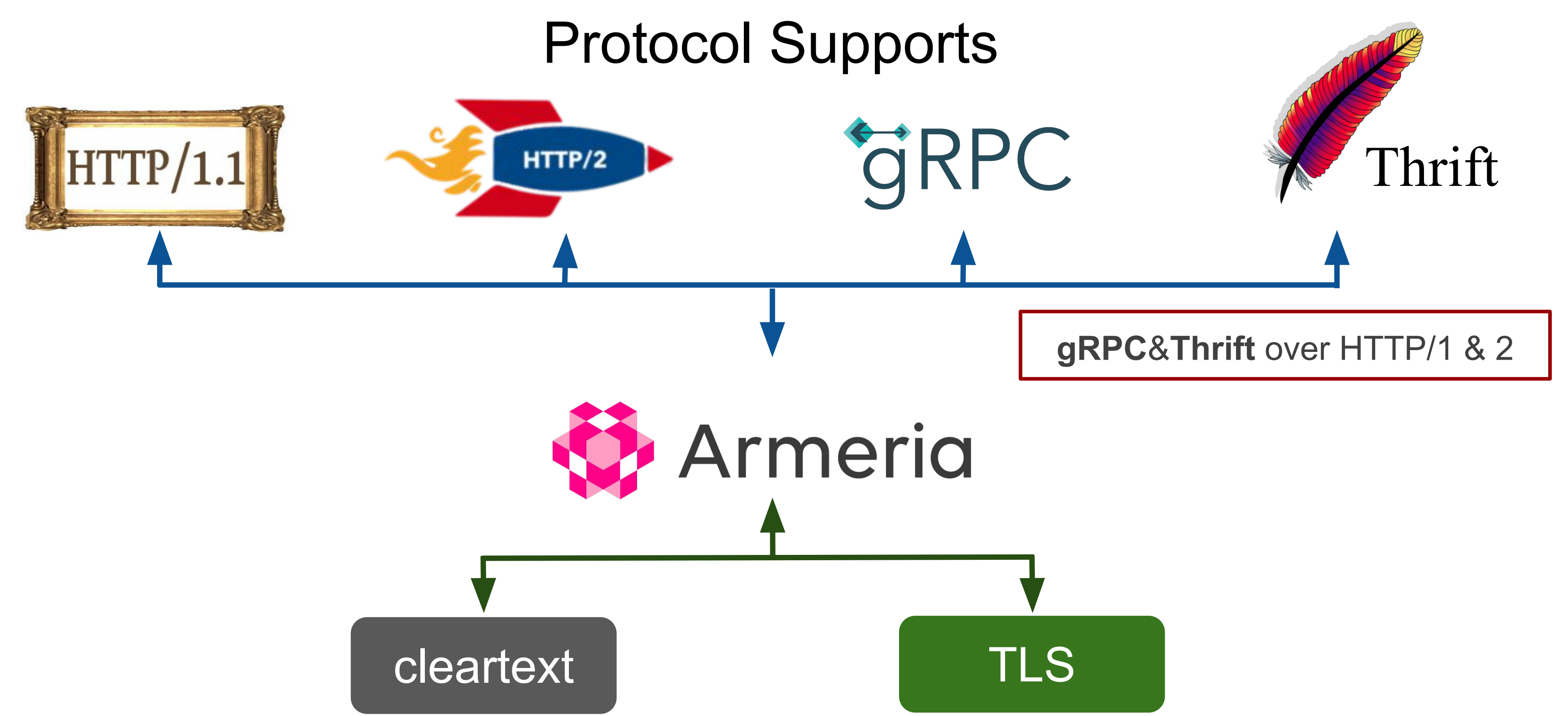
What's Armeria?

Armeria is an open-source asynchronous HTTP/2 RPC/REST client/server library built on top of Java 8, Netty, Thrift and gRPC.

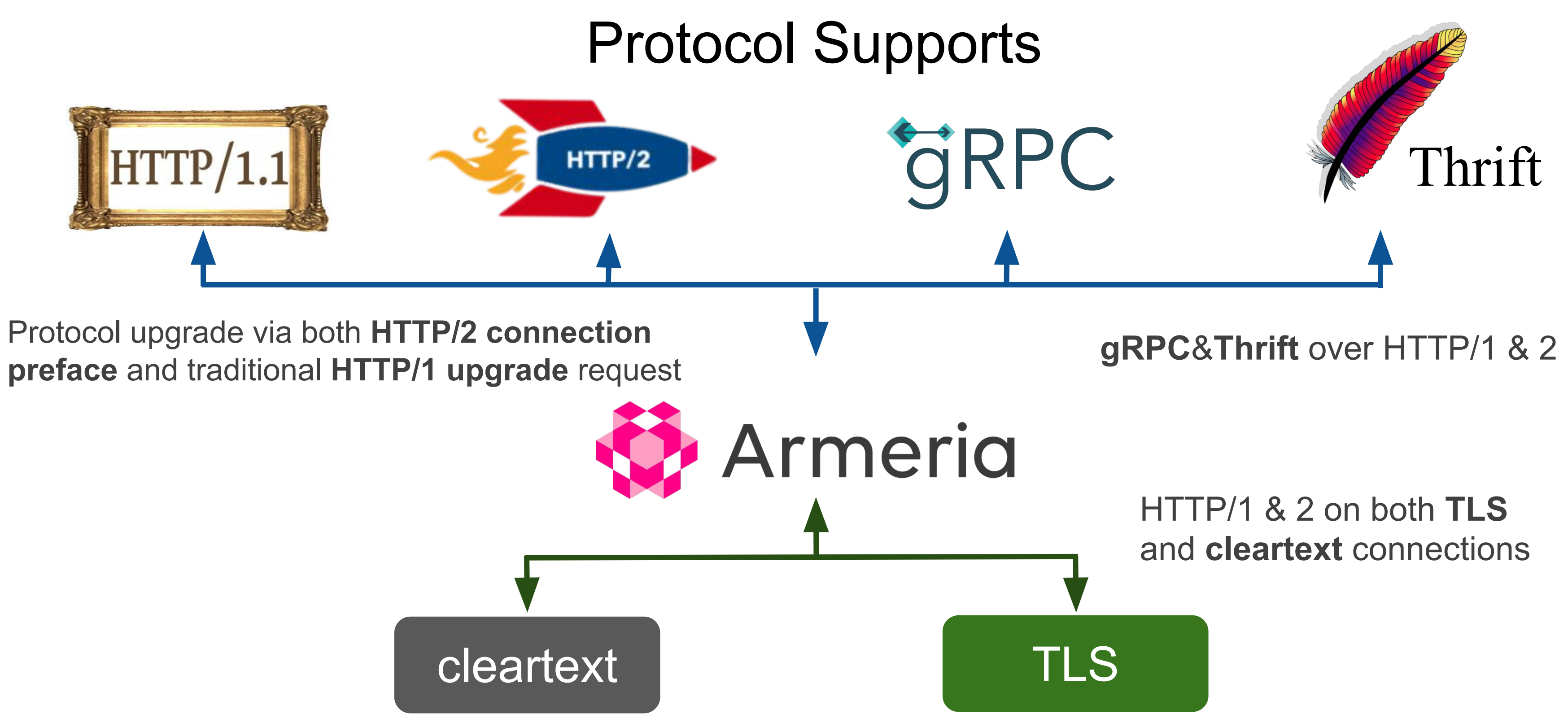
@armeria_project line/armeria


Protocol Supports
HTTP/1.1
HTTP/2
gRPC
Thrift
Armeria
HTTP/1 & 2 on both TLS and cleartext connections
cleartext
TLS


Protocol Supports
HTTP/1.1
HTTP/2
gRPC
Thrift
Protocol upgrade via both HTTP/2 connection preface and traditional HTTP/1 upgrade request
Armeria
cleartext
TLS


Protocol Supports
HTTP/1.1
HTTP/2
gRPC
Thrift
gRPC&Thrift over HTTP/1 & 2
Armeria
cleartext
TLS

Protocol Supports



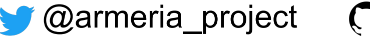
JNI-based socket I/O, **BoringSSL**-based TLS connections on Linux - **higher performance**

Learn Armeria by Example

Easy

```
// Build your own server under 5 lines.
Server server = Server.builder()
        .http(8080)
        .service("/", (ctx, req) -> HttpResponse.of("Hello, World!"))
        .build();

server.start();
```

Armeria is a light-weight microservice framework



Simple

```
// Don’t need additional sidecar like nginx to support HTTP2 or HTTPS
Server server = Server.builder()
        .http(8080)
        .https(8443)
        .tlsSelfSigned()
        .service("/", (ctx, req) -> HttpResponse.of("Hello, World!"))
        .build();
server.start();
```

Supports HTTP/2 on both TLS and cleartext connections

```java
import com.linecorp.armeria.server.annotation.Get;
import com.linecorp.armeria.server.annotation.Param;
import com.linecorp.armeria.server.annotation.PathPrefix;

// Use @annotations for mapping path and parameter injection
@PathPrefix("/hello")
class HelloService {
    @Get("/:name")
    public String hello(@Param String name) {
        return String.format("Hello, %s!", name);
    }
}
```

Armeria built-in annotations

@armeria_project line/armeria

Mix & Match! 🤞

```
Server server = Server.builder()
    .http(8080)
    .service("/hello/rest", (ctx, req) -> HttpResponse.of("Hello, world!"))
    .service("/hello/thrift", THttpService.of(new ThriftHelloService()))
    .service("/hello/grpc", GrpcService.builder()
                                      .addService(new GrpcHelloService())
                                      .build())
    .build();
```

Composable

```
// Write your service
UserService userService = ...;

// Write your decorating service
class AuthService extends SimpleDecoratingHttpService {
    @Override
    public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) {
        if (!authenticate(req))
            return HttpResponse.of(HttpStatus.UNAUTHORIZED);
        return delegate().serve(ctx, req);
    }
    ...
}
// Compose userService with auth & logging
Service<HttpRequest, HttpResponse> userAuthLoggingService =
    myService.decorate(AuthService::new)
             .decorate(LoggingService.newDecorator());
```

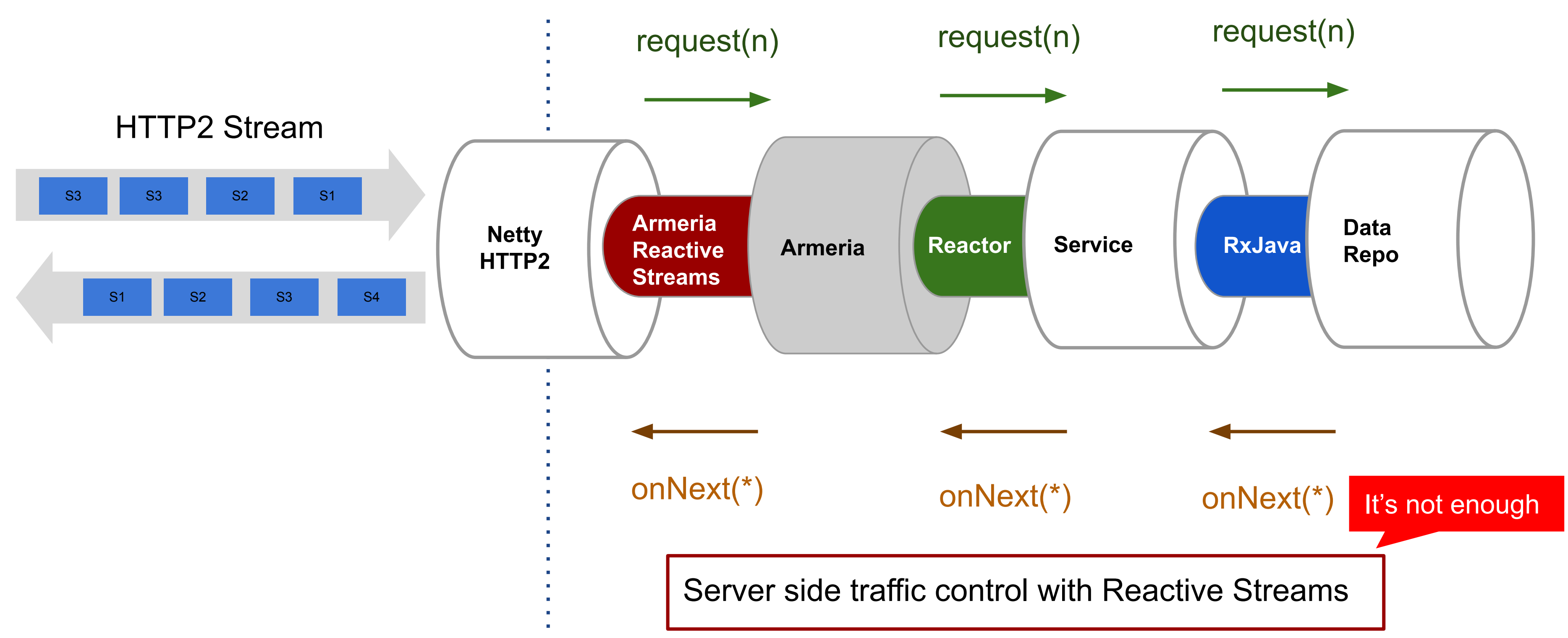
service
decorators
UserService
AuthService
LoggingService
compose
decorated service
LoggingService
AuthService
request
UserService
response

Composable

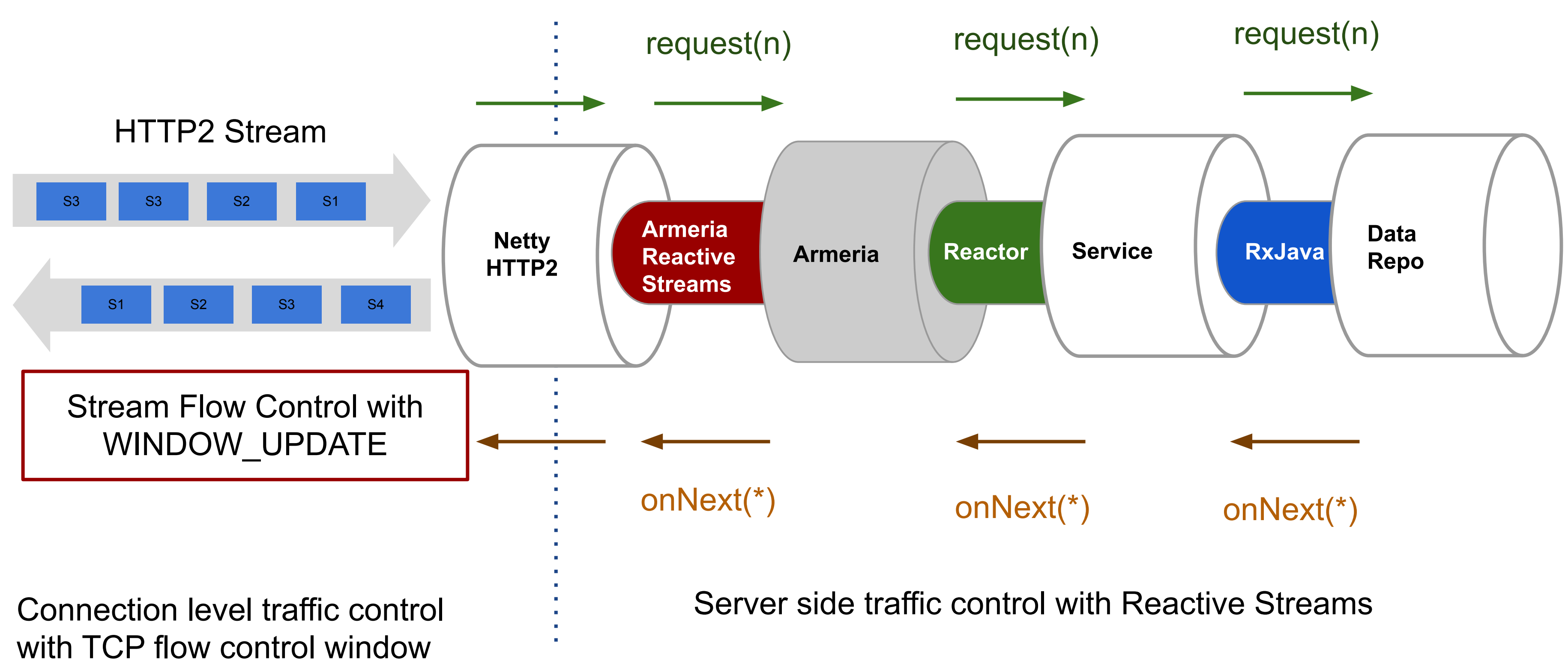
```
// Write your service
UserService userService = ...;

// Write your decorating service
class AuthService extends SimpleDecoratingHttpService {
    @Override
    public HttpResponse serve(ServiceRequestContext ctx, HttpRequest req) {
        if (!authenticate(req))
            return HttpResponse.of(HttpStatus.UNAUTHORIZED);
        return delegate().serve(ctx, req);
    }
    ...
}
// Compose userService with auth & logging
Service<HttpRequest, HttpResponse> userAuthLoggingService =
    myService.decorate(AuthService::new)
             .decorate(LoggingService.newDecorator());
```

service
decorators
UserService
AuthService
LoggingService
compose
decorated service
LoggingService
AuthService
request
UserService
response

HTTP2 Stream on top of Reactive Streams



HTTP2 Stream on top of Reactive Streams



Reactive HTTP/2 proxy in 6 lines

```
// Use Armeria's async & reactive HTTP/2 client.
var client = HttpClient.of("h2c://backend");
var server = Server.builder()
    .http(8080)           // Forward all requests reactively
    .service("prefix:/", (ctx, req) -> client.execute(req))
    .build();
```

Roll your own Reactive Streams Server with Armeria

Publisher to Armeria Response Basics

a | b | ...

```
// 1. Fetch data from Reactive Streams Publisher
Observable<String> dataStream = Observable.just("a", "b", "c", "d", "e");
```

Publisher to Armeria Response Basics

```java
// 1. Fetch data from Reactive Streams Publisher
Observable<String> dataStream = Observable.just("a", "b", "c", "d", "e");

// 2. Convert string to Armeria HttpData
Observable<HttpData> httpDataStream = dataStream.map(HttpData::ofUtf8);
```

a | b | ...

HTTP Data | HTTP Data | ...

convert to HTTP data

Publisher to Armeria Response Basics

```
// 1. Fetch data from Reactive Streams Publisher
Observable<String> dataStream = Observable.just("a", "b", "c", "d", "e");

// 2. Convert string to Armeria HttpData
Observable<HttpData> httpDataStream = dataStream.map(HttpData::ofUtf8);

// 3. Prepare response headers
ResponseHeaders httpHeaders = ResponseHeaders.of(HttpStatus.OK);
```

Publisher to Armeria Response Basics

```
// 1. Fetch data from Reactive Streams Publisher
Observable<String> dataStream = Observable.just("a", "b", "c", "d", "e");

// 2. Convert string to Armeria HttpData
Observable<HttpData> httpDataStream = dataStream.map(HttpData::ofUtf8);

// 3. Prepare response headers
ResponseHeaders httpHeaders = ResponseHeaders.of(HttpStatus.OK);

// 4. Concat http header and body stream
Observable<HttpObject> responseStream = Observable.concat(Observable.just(httpHeaders), httpDataStream);
```

HTTP Data | HTTP Data | ...

+

HTTP Header

Publisher to Armeria Response Basics

```
// 1. Fetch data from Reactive Streams Publisher
Observable<String> dataStream = Observable.just("a", "b", "c", "d", "e");

// 2. Convert string to Armeria HttpData
Observable<HttpData> httpDataStream = dataStream.map(HttpData::ofUtf8);

// 3. Prepare response headers
ResponseHeaders httpHeaders = ResponseHeaders.of(HttpStatus.OK);

// 4. Concat http header and body stream
Observable<HttpObject> responseStream = Observable.concat(
httpDataStream);

// 5. Convert Observable to Armeria Response Stream
HttpResponse response = HttpResponse.of(responseStream.toFlowable(BackpressureStrategy.BUFFER));
```

HTTP Response

HTTP Header | HTTP Data | HTTP Data | ...

Convert to Reactive Streams

Shortcut - Built in JSON Text Sequences Publisher

```
// 1. Fetch data from Reactive Streams Publisher
Publisher<String> dataStream = Flux.just("a", "b", "c", "d", "e");

// 2. Convert Publisher to JSON Text Sequences with Armeria HttpResponse
HttpResponse httpResponse = JsonTextSequences.fromPublisher(dataStream);
```

Generate **JSON Text Sequences**(RFC 7464) with “application/json-seq” MIME type

Shortcut - Built in Server-sent Events Publisher

```java
// 1. Fetch data from Reactive Streams Publisher
Publisher<String> dataStream = Flux.just("a", "b", "c", "d", "e");

// 2. Convert Publisher to Server-sent Events with Armeria HttpResponse
HttpResponse httpResponse = ServerSentEvents
        .fromPublisher(dataStream, SeverSentEvent::ofData);
```

Generate **Server-sent Events**(HTML5 standard) with “**text/event-stream**” MIME type

```java
import io.reactivex.Observable;
import com.linecorp.armeria.server.annotation.ProducesJsonSequences;

class RxJavaService {
  @Get("/json")
  @ProducesJsonSequences
  // Just return RxJava Observable!
  public Observable<String> json() {
      return Observable.just("a", "b", "c");
  }
});
```

Generate **JSON Text Sequences**

No more explicit conversion.
Armeria works for you 😉

JSON streaming over HTTP/1 & 2

```java
import com.linecorp.armeria.server.streaming.JsonTextSequences;

Server jsonStreamingServer = Server.builder()
    .service("/users-streaming", (ctx, req) -> {
        Publisher<User> userPublisher = Flux.from(userA, userB, ...);
        JsonTextSequences.fromPublisher(userPublisher);
    })
    .build();
```

Reactive Streams works on both HTTP/1 & 2

RS	JSON Text	LF
RS	JSON Text	LF
RS	JSON Text	LF

⋮

HTTP/1

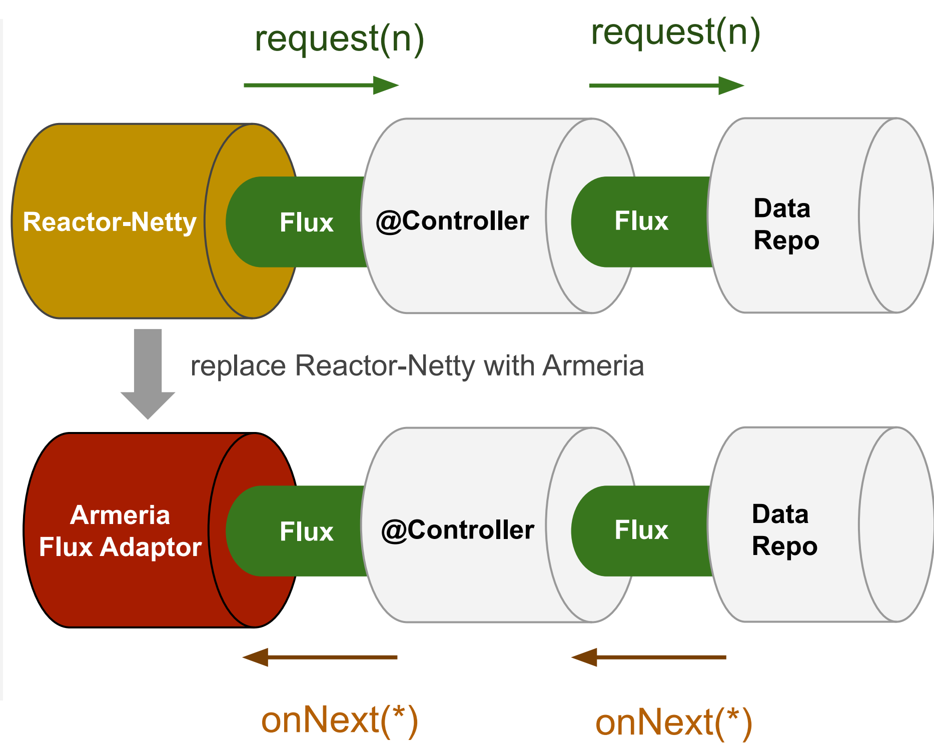
```
$ curl --http1.1 -vv http://127.0.0.1:8080/users-streaming
...
< HTTP/1.1 200 OK
< content-type: application/json-seq
< transfer-encoding: chunked
<
{"id":0,"name":"User0","age":27}
{"id":1,"name":"User1","age":25}
{"id":2,"name":"User2","age":82}
{"id":3,"name":"User3","age":29}
...
```

HTTP/2

```
$ nghttp -v http://127.0.0.1:8080/users-streaming
...
recv (stream_id=13) :status: 200
recv (stream_id=13) content-type: application/json-seq
recv HEADERS frame <length=28, flags=0x24, stream_id=13>
        ; END_HEADERS | PRIORITY
        (padlen=0, dep_stream_id=0, weight=16, exclusive=0)
        ; First response header
{"id":0,"name":"User0","age":3}
recv DATA frame <length=33, flags=0x00, stream_id=13>
{"id":1,"name":"User1","age":50}
recv DATA frame <length=34, flags=0x00, stream_id=13>
{"id":2,"name":"User2","age":25}
...
```

Spring Webflux Integration

```
// build.gradle
plugins {
  id "org.springframework.boot" version "2.1.8.RELEASE"
}

dependencyManagement {
  imports {
    mavenBom 'com.linecorp.armeria:armeria-bom:0.94.0'
    mavenBom 'io.netty:netty-bom:4.1.42.Final'
  }
}

dependencies {
  // Configure Armeria as the HTTP server for WebFlux
  compile 'com.linecorp.armeria:armeria-spring-boot-webflux-starter'
}
```



Armeria 💖 Spring Webflux

You can now **run your Spring Webflux code** on top of **Armeria Reactive Server with zero modification**

```
@Controller
class ReactiveController {
    @GetMapping("/")
    Mono<String> index() {
        return webClient.get()
                        .uri("/hello")
                        .retrieve()
                        .bodyToMono(String.class);
    }
}
```

Customize Armeria for Spring WebFlux

```java
@Configuration
public class ArmeriaConfiguration {
    // Configure the server by providing an ArmeriaServerConfigurator bean.
    @Bean
    public ArmeriaServerConfigurator armeriaServerConfigurator() {
        // Customize the server using the given ServerBuilder. For example:
        return builder -> {
            // Add DocService that enables you to send gRPC and Thrift requests from web browser.
            builder.serviceUnder("/docs", new DocService());
            // Log every message which the server receives and responds.
            builder.decorator(LoggingService.newDecorator());
            // Write access log after completing a request.
            builder.accessLogWriter(AccessLogWriter.combined(), false);
            // You can also bind asynchronous RPC services such as Thrift and gRPC:
            builder.service(THttpService.of(...));
            builder.service(GrpcService.builder()...build());
        };
    }
}
```

gRPC stream basics

```
syntax = "proto3";
package users;
option java_package = "users";

service UserService {
  // Returns all user information
  rpc getAllUsers(UserRequest) returns (stream User) {}

  // Save all given user stream
  rpc saveUsers(stream User) returns (Result) {}
}
```

Publish gRPC stream

Subscribe gRPC stream

Implement service - Publisher to gRPC StreamObserver

```java
// Implement interfaces generated by gRPC
public final class UserServiceImpl extends UserServiceImplBase {

    @Override
    public void getAllUsers(UserRequest request, StreamObserver<User> responseObserver) {
        final Flux<User> userPublisher = userRepo.findAll();
        publisher.subscribe(responseObserver::onNext,
                            responseObserver::onError,
                            responseObserver::onCompleted);

    }

}
```

Convert Flux(Publisher) to StreamObserver

Implement service - gRPC StreamObserver to Processor

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}
```

```
@Override
public StreamObserver<User> saveUsers(StreamObserver<Result> responseObserver) {
    Processor<User, User> processor = EmitterProcessor.create();
    Publisher<User> publisher = processor;
    Subscriber<User> subscriber = processor;
    // save logic ...

    return new StreamObserver<User>() {
        // subscribe user data
        @Override public void onNext(User user) { processor.onNext(user); }
        @Override public void onError(Throwable throwable) { processor.onError(throwable); }
        @Override public void onCompleted() {
            responseObserver.onNext(Result.newBuilder().setStatus(200).build());
            responseObserver.onCompleted();
        }
    };
}
```

Processor can be Publisher and Subscriber

give data to processor(subscriber)

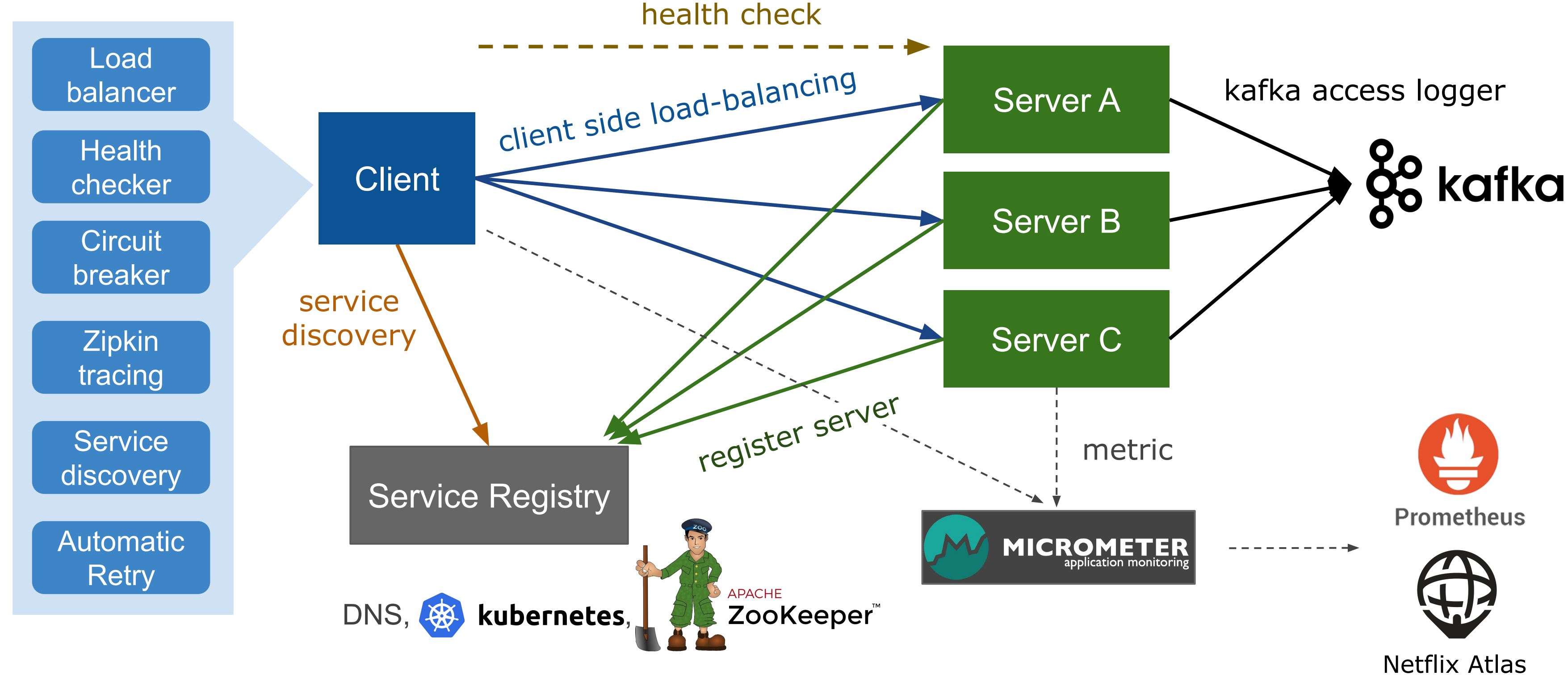
Run your service on Armeria

```
import com.linecorp.armeria.server.Server;
import com.linecorp.armeria.server.grpc.GrpcService;

// Add your grpc service to Armeria GrpcService
var grpcService = GrpcService.builder()
                             .addService(new UserServiceImpl())
                             .build();

var server = Server.builder()
                   .http(8080)
                   .serviceUnder("/grpc", grpcService)
                   .serviceUnder("/docs", new DocService())
                   .build();
```

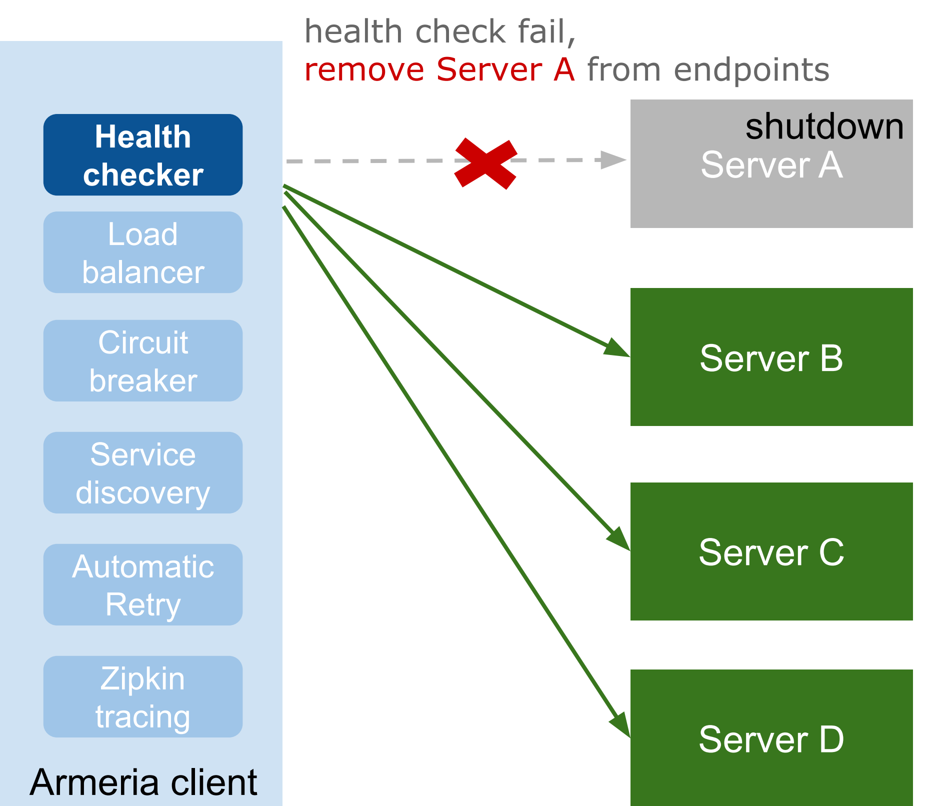
gRPC service is registered to Armeria Server

Armeria built in web console for gRPC, Thrift & REST

Essential Features for Microservices

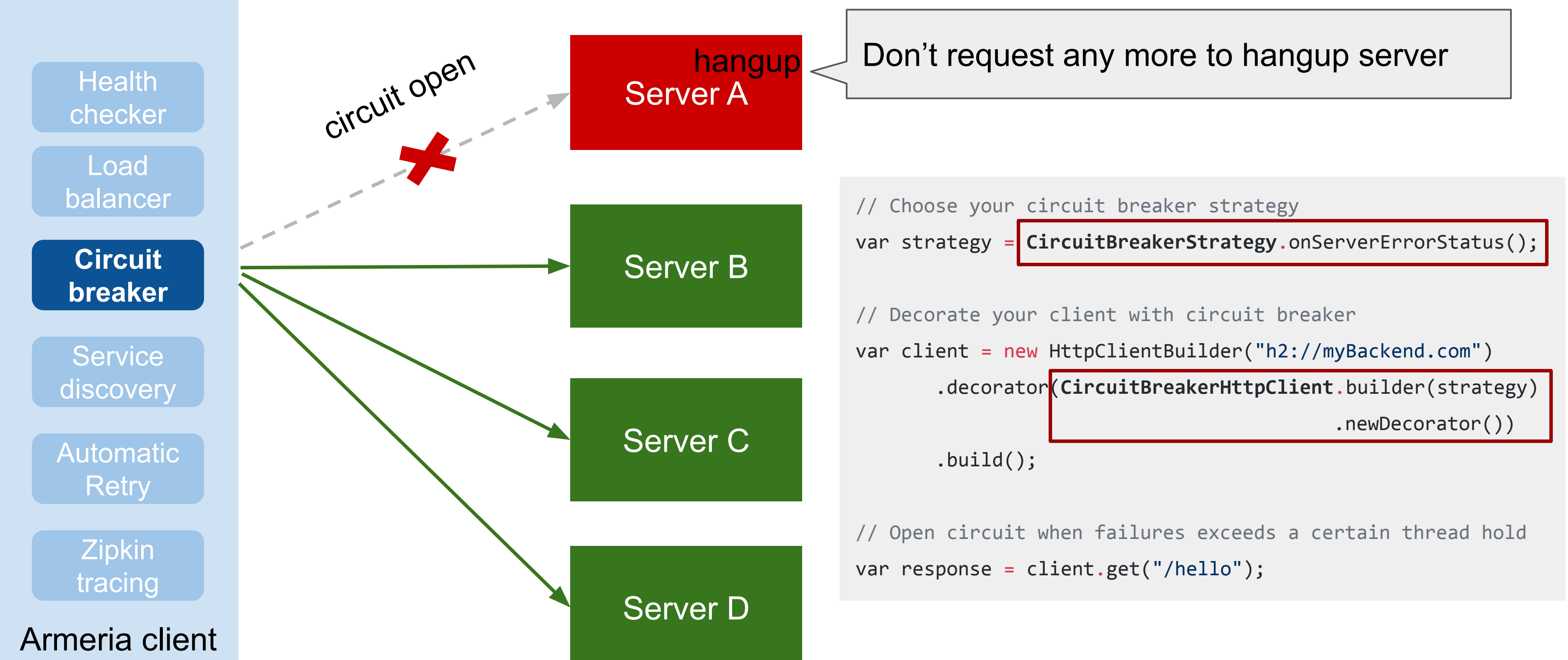


Disaster resilience - Health check

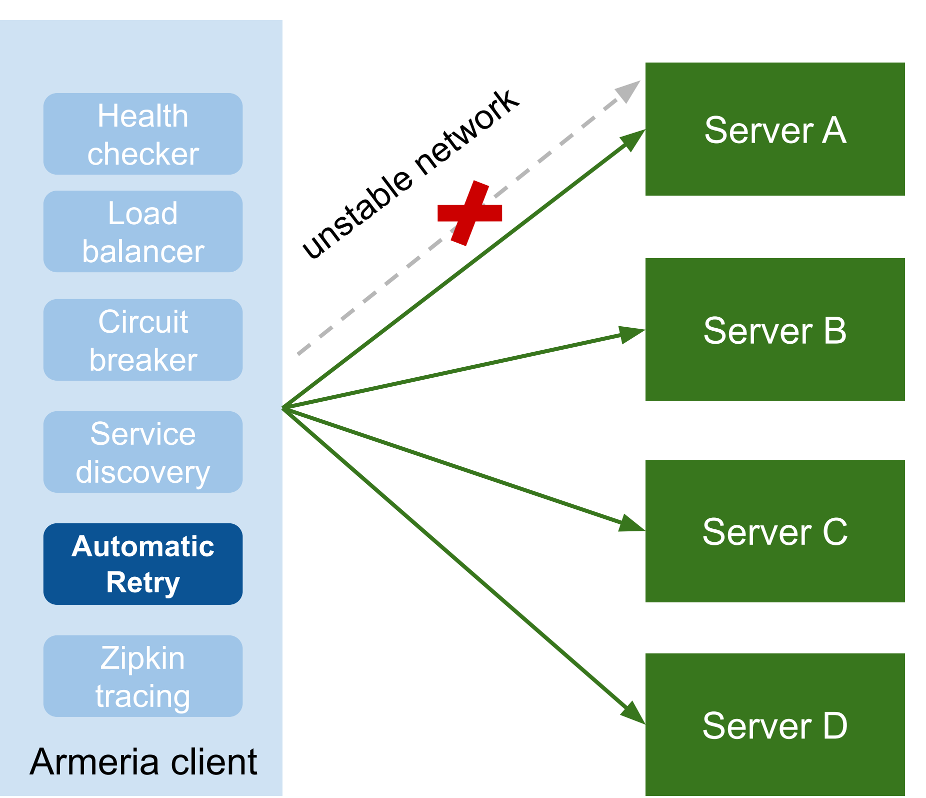



```
var endpoint = Endpoint.of("myServiceHost", "8080");
// Create endpoint group with health chekcker
var endpointGroup = HealthCheckedEndpointGroup
        .builder(EndpointGroup.of(endpoint), HEALTH_CHECK_PATH)
        .build();

// Register health checked endpoint group
EndpointGroupRegistry.register("myService", endpointGroup,
        EndpointSelectionStrategy.WEIGHTED_ROUND_ROBIN);

var client = HttpClient.of("http://group:myService");
// Only request to healthy endpoints
var response = client.get("/hello");
```

Disaster resilience - Circuit breaker



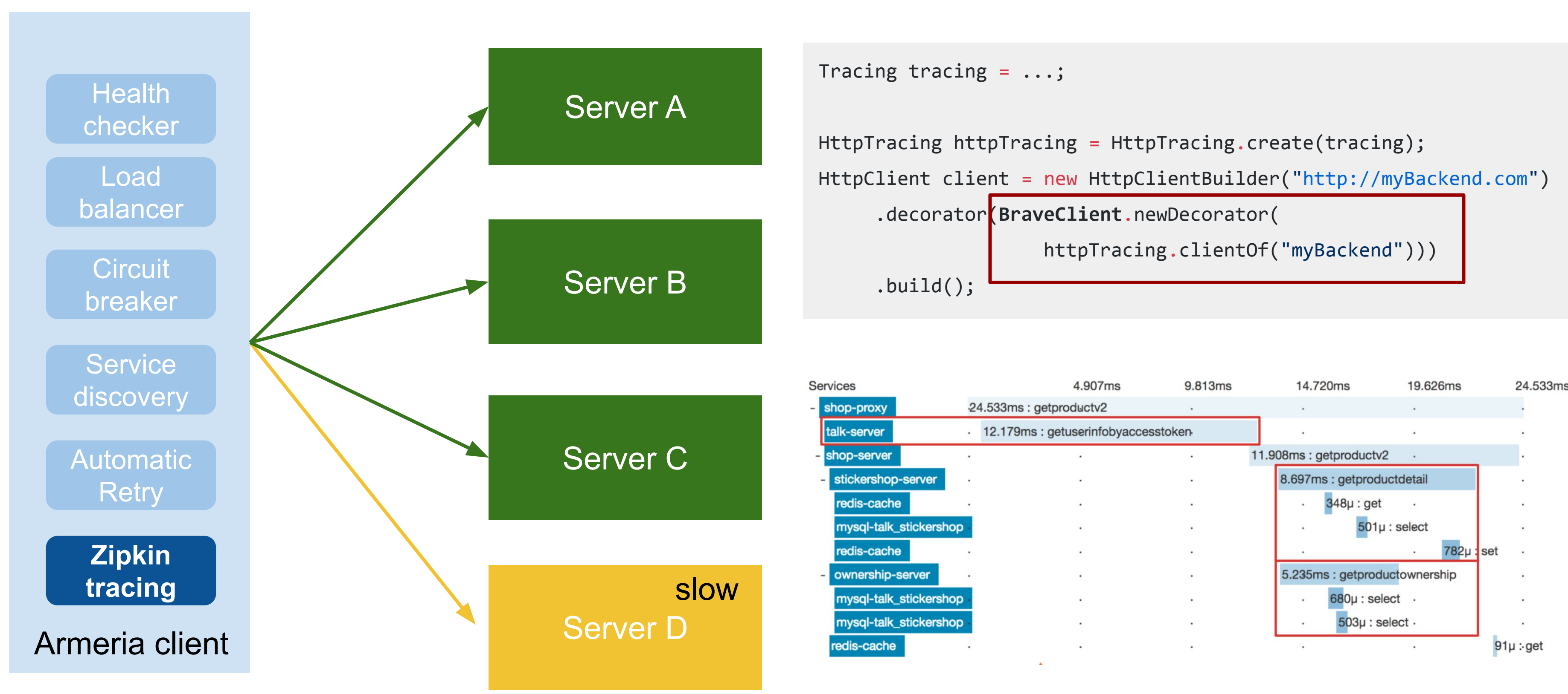
Disaster resilience - Automatic retry



Automatic retry on unstable network or choose your own strategy

```
// Choose your retry strategy
var strategy = RetryStrategy.onUnprocessed();

// Decorate your client with automatic retry client
var client = new HttpClientBuilder("h2://myBackend.com")
        .decorator(RetryingHttpClient.builder(strategy)
                                     .newDecorator())
        .build();

// Automatic retry with backoff
var response = client.get("/hello");
```

Disaster resilience - Distributed tracing



distributed tracing with zipkin

Got interest?

@armeria_project line/armeria

Let's build Armeria *together*!

- Give it a try.
- Ask questions.
- Request new features.
- Tell us what rocks and sucks.
- Consider joining the effort.

@armeria_project   line/armeria

Meet us at Github

github.com/line/armeria

line.github.io/armeria

@armeria_project line/armeria