Armeria

A Microservice Framework Well-suited Everywhere

Trustin Lee, LINE
Oct 2019

@armeria_project line/armeria

A microservice framework, *again?*

Yeah, but for good reasons!

- Simple & User-friendly
- Asynchronous & Reactive
- 1st-class RPC support
  - … with better-than-upstream experience
- Unopinionated integration & migration
- Less points of failure

How simple is it, then?

Hello, world!

```
Server server = Server.builder()
    .http(8080)
    .https(8443)
    .tlsSelfSigned()
    .haproxy(8080)
    .service("/hello/:name",
             (ctx, req) -> HttpResponse.of("Hello, %s!",
                                           ctx.pathParam("name")))
    .build();
server.start();
```

Protocol auto-detection at 8080

Hello, world – Annotated

```
Server server = Server.builder()
    .http(8080)
    .annotatedService(new Object() {
        @Get("/hello/:name")
        public String hello(@Param String name) {
            return String.format("Hello, %s!", name);
        }
    })
    .build();
server.start();
```

- Full example:
  https://github.com/line/armeria-examples/tree/master/annotated-http-service

gRPC

```
Server server = Server.builder()
    .http(8080)
    .service(GrpcService.builder()
                        .addService(new GrpcHelloService())
                        .build())
    .build();

class GrpcHelloService
    extends HelloServiceGrpc.HelloServiceImplBase {
    ...
}
```

- Full example: https://github.com/line/armeria-examples/tree/master/grpc-service

Thrift

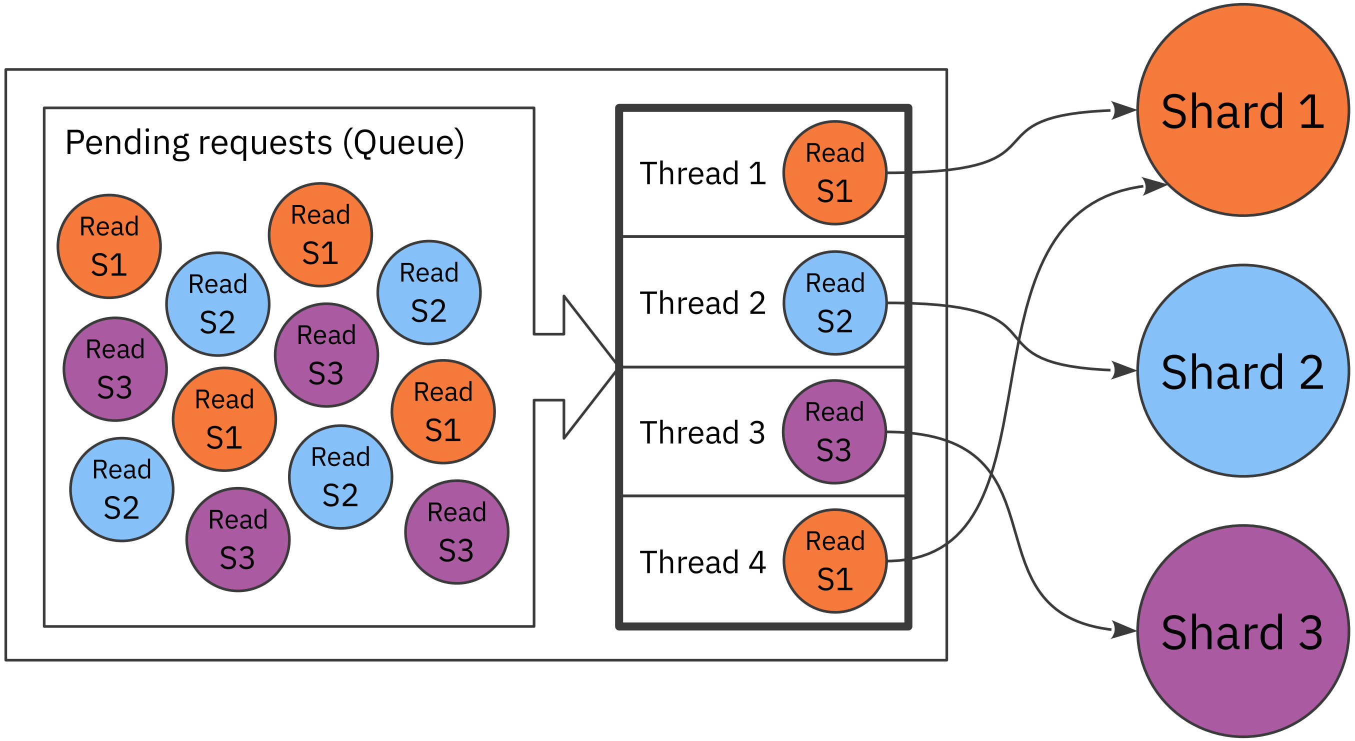
```java
Server server = Server.builder()
    .http(8080)
    .service("/hello",
             THttpService.of(new ThriftHelloService()))
    .build();

class ThriftHelloService implements HelloService.AsyncIface {
    ...
}
```

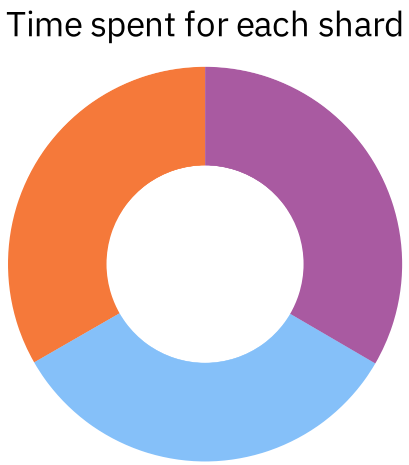
Mix & Match!

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

Why going asynchronous & reactive?

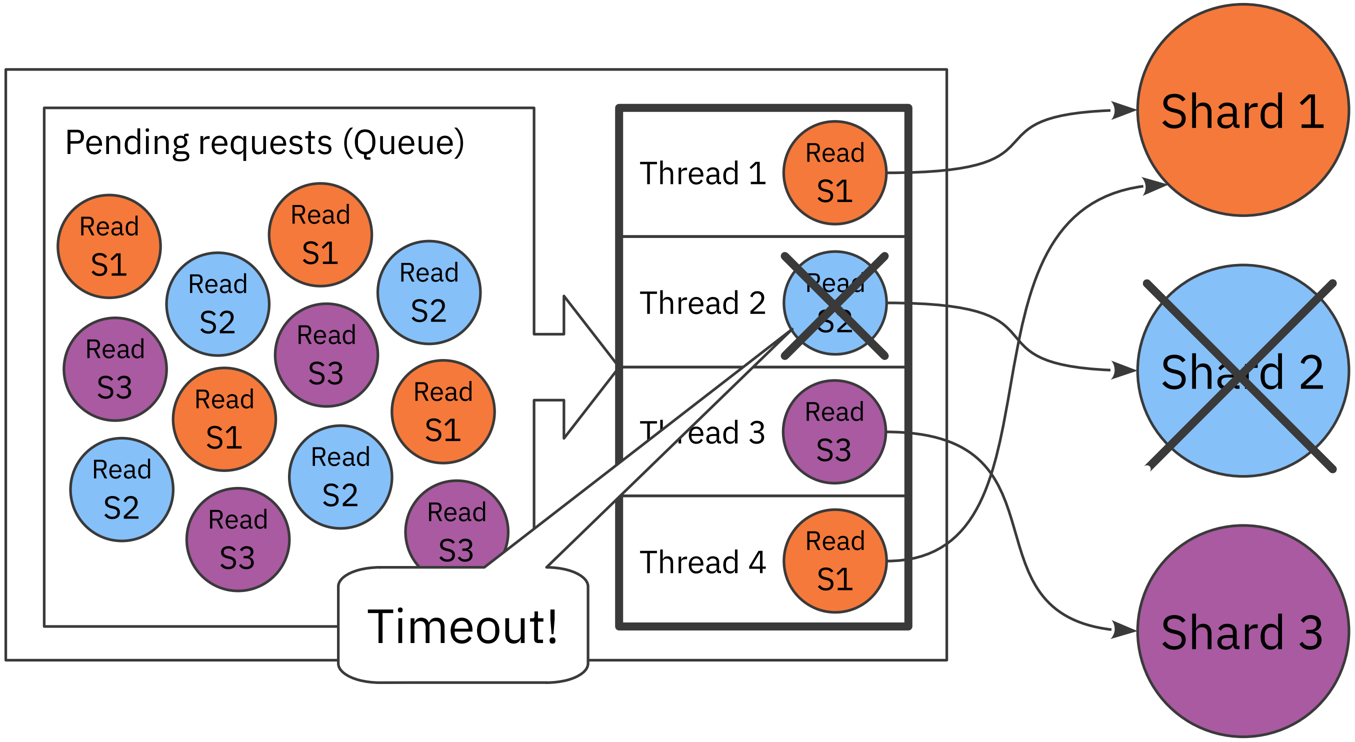
One fine day of a synchronous microservice



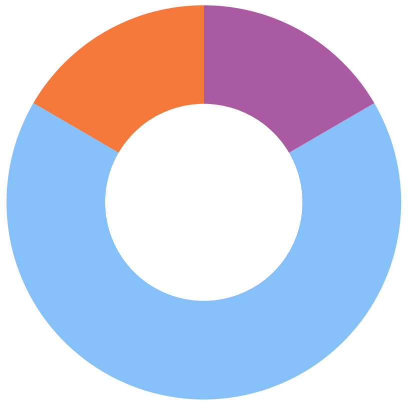


@armeria_project line/armeria

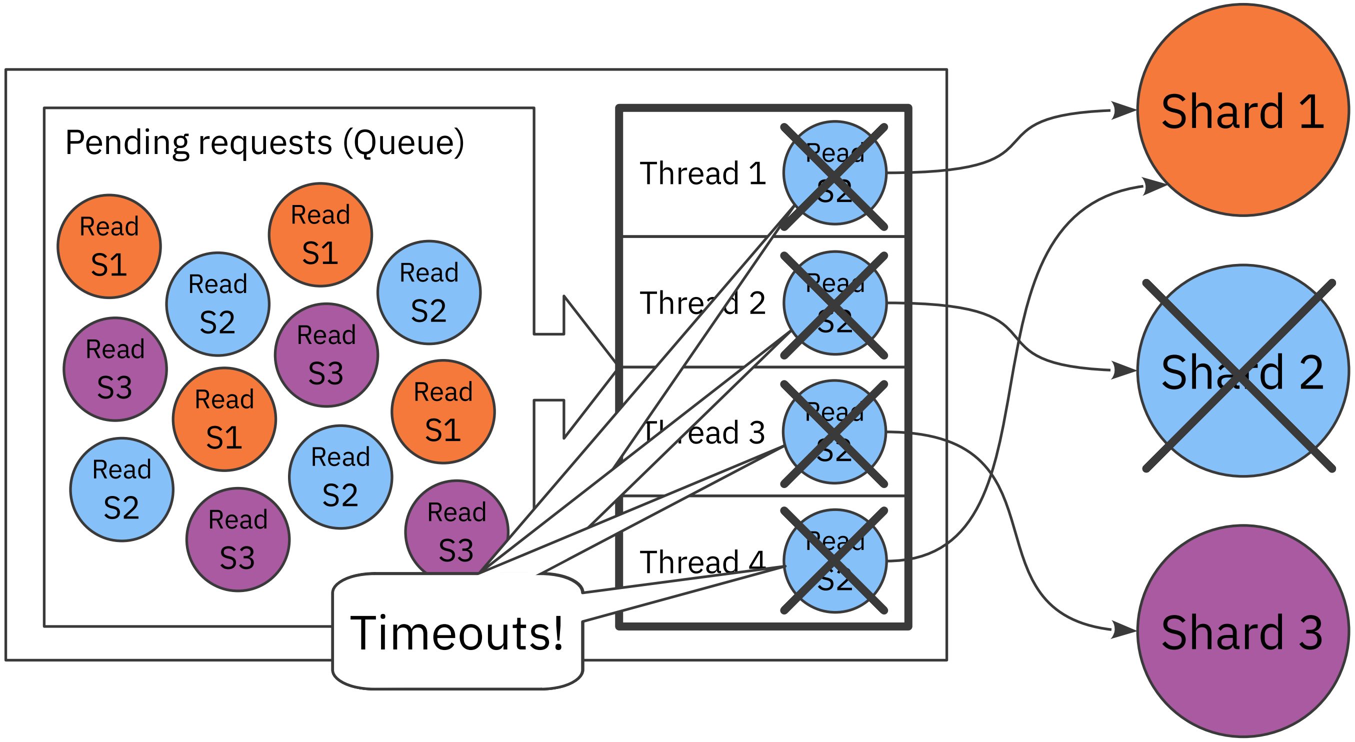
Shard 2 ruins the fine day...



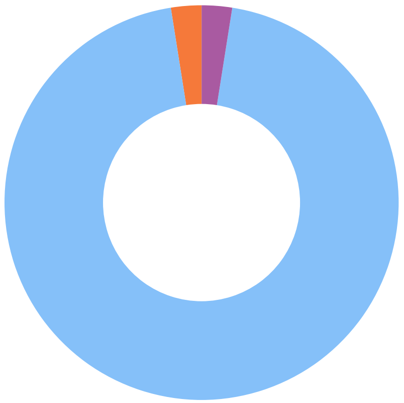
Time spent for each shard



Shard 1 & 3: Why are no requests coming? 🤔 Workers: We're busy *waiting* for Shard 2.

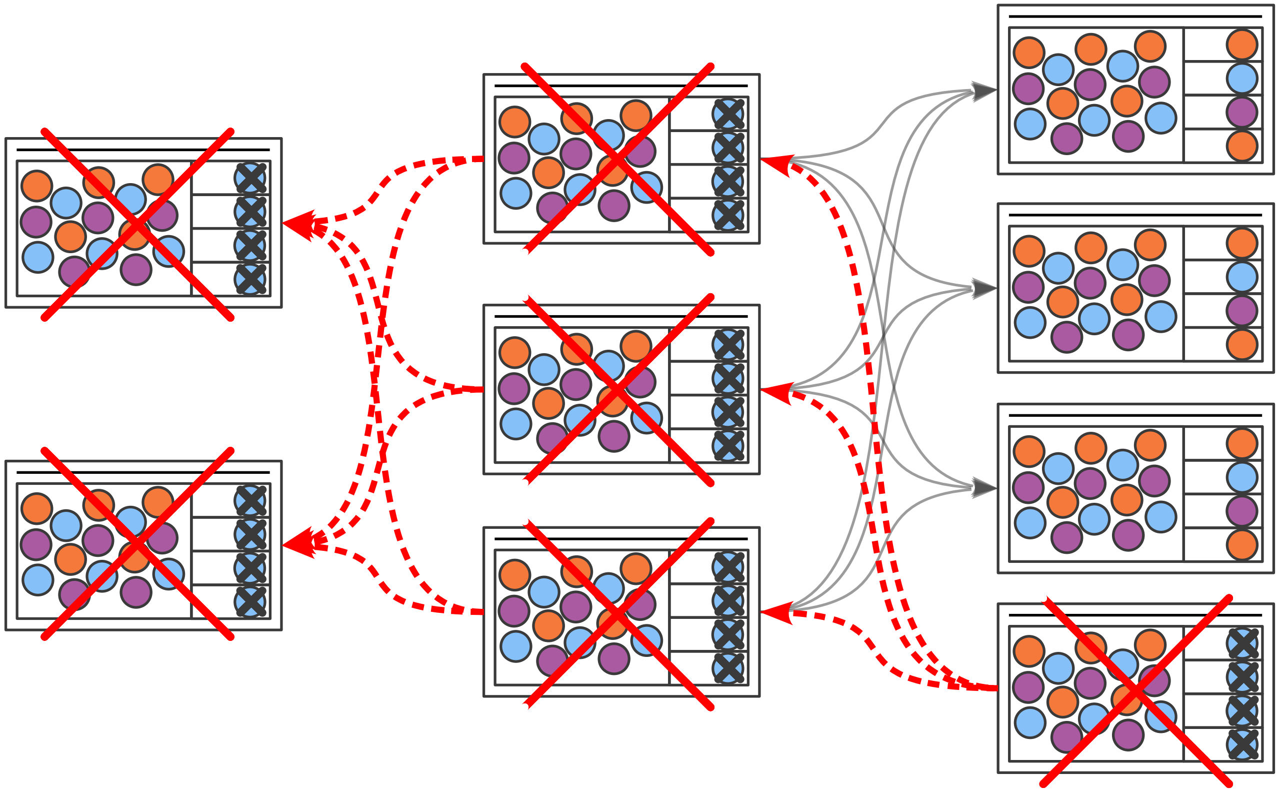


Time spent for each shard



@armeria_project line/armeria

... propagating everywhere! 🙀



How can we solve this?

- Add more CPUs?
  - They are very idle.
- Add more threads?
  - They will all get stuck with Shard 2 in no time.
  - Waste of CPU cycles & memory – context switches & call stack
- Result:
  - Fragile system that falls apart even on a tiny backend failure
  - Inefficient system that takes more memory and CPU

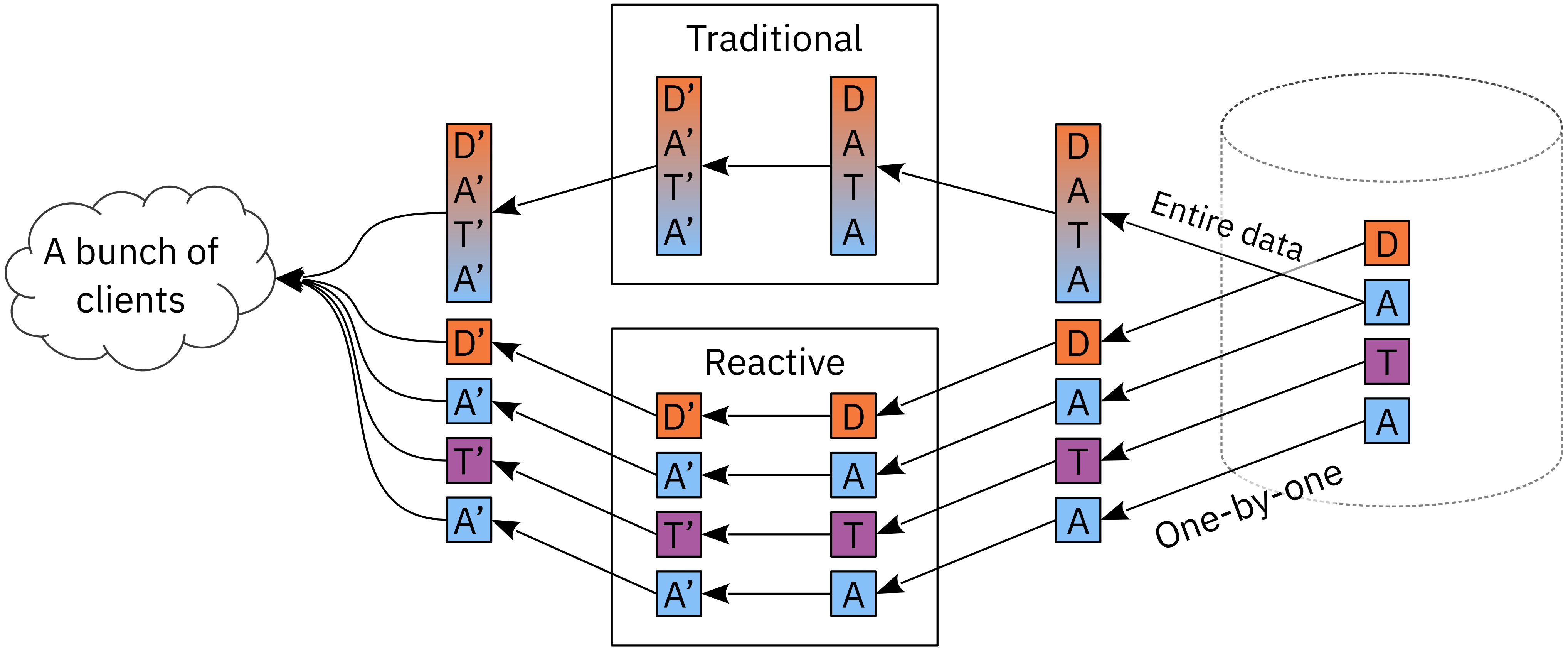
How can we solve this? (cont'd)

- Can work around, must keep tuning and adding hacks, e.g.
  - Increasing # of threads & reducing call stack
  - Prepare thread pools for each shard
- Shall we just go asynchronous, please?
  - Less tuning points
    - Memory size & # of event loops
  - Better resource utilization with concurrent calls + less threads

Problems with large payloads

- We solved blocking problem with asynchronous programming, but can we send *10MB* personalized response to *100K* clients?
  - Can't hold that much in RAM – 10MB × 100K = *1TB*
- What if we · they send too fast?
  - Different bandwidth & processing power
- We need '*just enough* buffering.'
  - Expect `OutOfMemoryError` otherwise.

Traditional vs. Reactive



Reactive HTTP/2 proxy in 6 lines

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

- Full example:
  https://github.com/line/armeria-examples/tree/master/proxy-server

1st-class RPC support

with better-than-upstream experience

RPC vs. HTTP impedance mismatch

- RPC has been hardly a 1st-class citizen in web frameworks.
  - Which method was called with what parameters?
  - What's the return value? Did it succeed?

```
POST /some_service HTTP/1.1
Host: example.com
Content-Length: 96

<binary request>
```

```
HTTP/1.1 200 OK
Host: example.com
Content-Length: 192

<binary response>
```

Failed RPC call

```
192.167.1.2 - - [10/Oct/2000:13:55:36 -0700]
"POST /some_service HTTP/1.1" 200 2326
```

Killing many birds with Structured Logging

- Timings
  - Low-level timings, e.g. DNS · Socket
  - Request · Response time
- Application-level
  - Custom attributes
    - User
    - Client type
    - Region, …
- HTTP-level
  - Request · Response headers
  - Content preview, e.g. first 64 bytes
- RPC-level
  - Service type
  - method and parameters
  - Return values and exceptions

First things first – Decorators

```
GrpcService.builder().addService(new MyServiceImpl()).build()
    .decorate((delegate, ctx, req) -> {
        ctx.log().addListener(log -> {
            ...
        }, RequestLogAvailability.COMPLETE);

        return delegate.serve(ctx, req);
    });
```

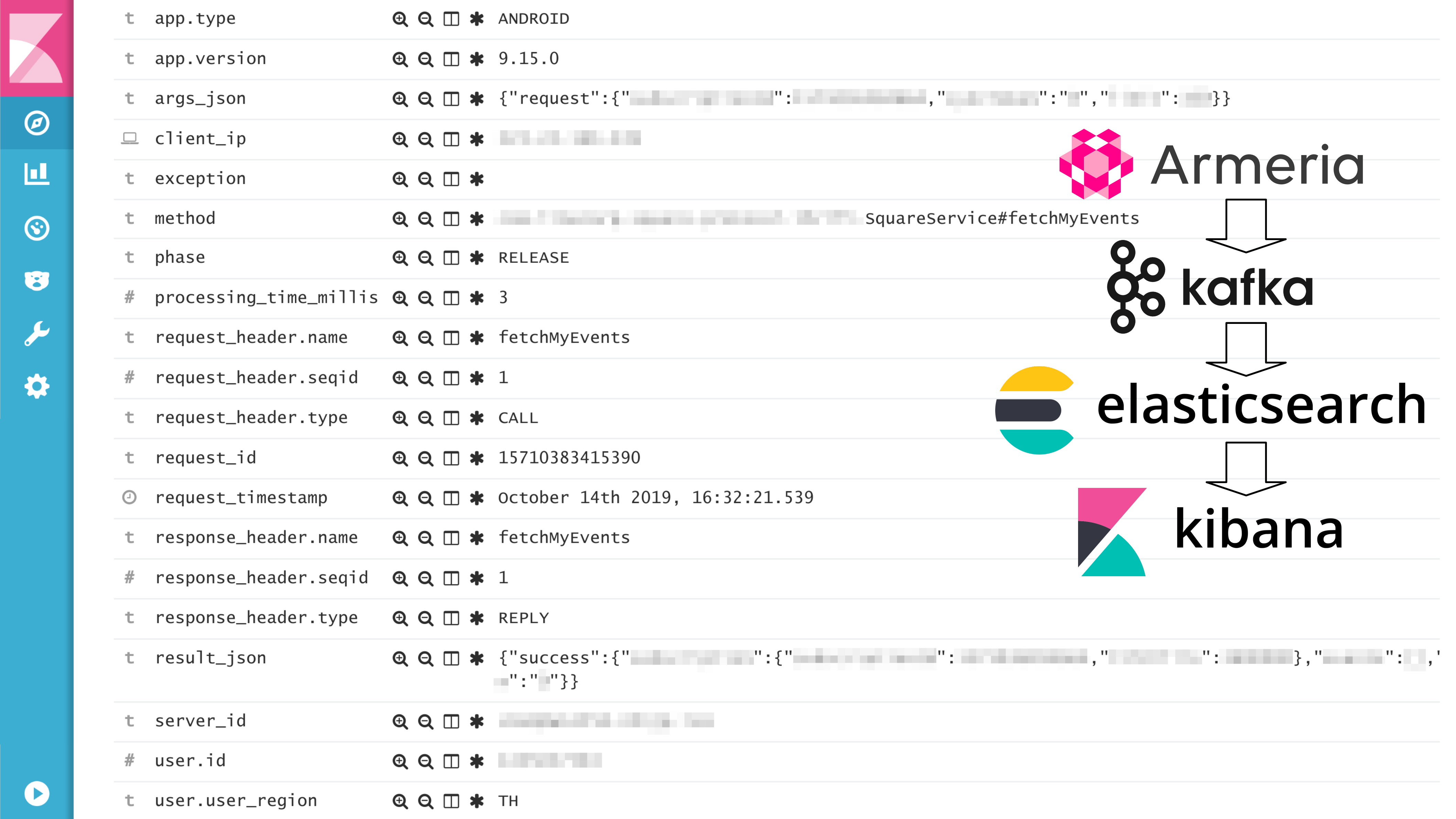
- Decorators are used everywhere in Armeria
  - Most features mentioned in this presentation are decorators.

Async retrieval of structured logs

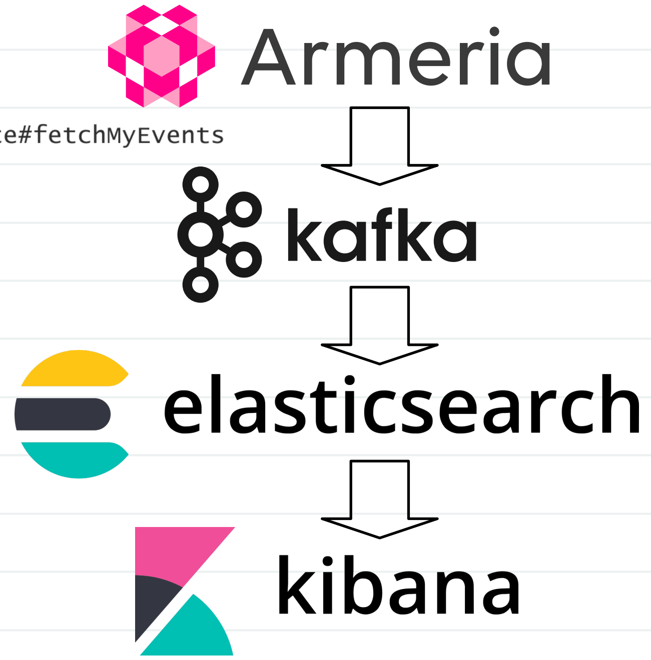
```
GrpcService.builder().addService(new MyServiceImpl()).build()
    .decorate((delegate, ctx, req) -> {
        ctx.log().addListener(log -> {
            ...
        }, RequestLogAvailability.COMPLETE);

        return delegate.serve(ctx, req);
    });
```

Async retrieval of structured logs (cont'd)

```
ctx.log().addListener(log -> {
    long reqStartTime = log.requestStartTimeMillis();
    long resStartTime = log.responseStartTimeMillis();

    RpcRequest rpcReq = (RpcRequest) log.requestContent();
    if (rpcReq != null) {
        String method = rpcReq.method();
        List<Object> params = rpcReq.params();

        RpcResponse rpcRes = (RpcResponse) log.responseContent();
        if (rpcRes != null) {
            Object result = rpcRes.getNow(null);
        }
    }
}, RequestLogAvailability.COMPLETE);
```



t	app.type	ANDROID
t	app.version	9.15.0
t	args_json	{"request":{"[redacted]":[redacted],"[redacted]":"[redacted]","[redacted]":[redacted]}}
💻	client_ip	[redacted]
t	exception	
t	method	[redacted] SquareService#fetchMyEvents
t	phase	RELEASE
#	processing_time_millis	3
t	request_header.name	fetchMyEvents
#	request_header.seqid	1
t	request_header.type	CALL
t	request_id	15710383415390
🕘	request_timestamp	October 14th 2019, 16:32:21.539
t	response_header.name	fetchMyEvents
#	response_header.seqid	1
t	response_header.type	REPLY
t	result_json	{"success":{"[redacted]":{"[redacted]":[redacted],"[redacted]":[redacted]},"[redacted]":[redacted],"[redacted]":"[redacted]"}}
t	server_id	[redacted]
#	user.id	[redacted]
t	user.user_region	TH

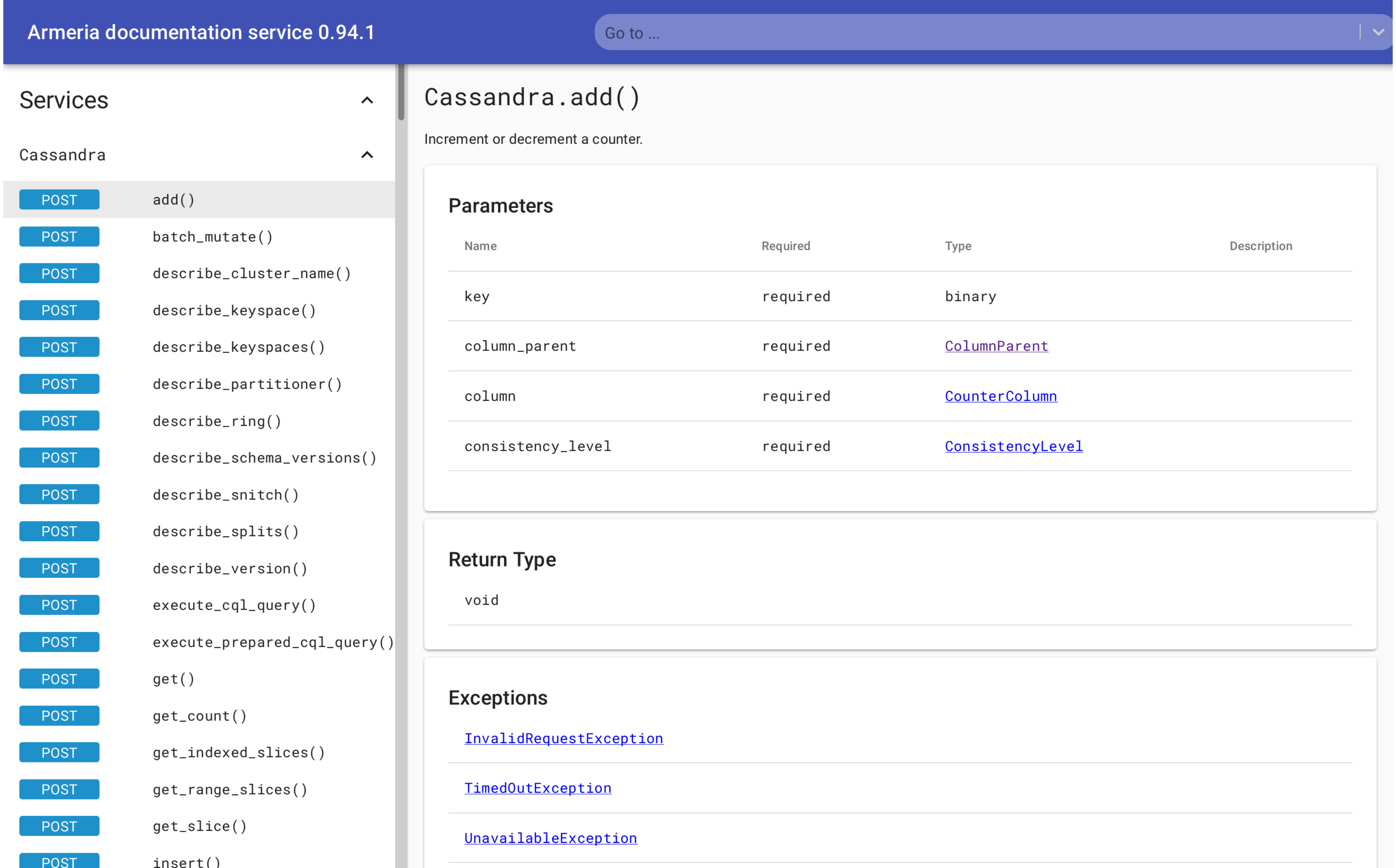


Making a debug call

- Sending an ad-hoc query in RPC is hard.
  - Find a proper service definition, e.g. `.thrift` or `.proto` files
  - Set up code generator, build, IDE, etc.
  - Write some code that makes an RPC call.
- HTTP in contrast:
  - cURL, `telnet` command, web-based tools and more.
- What if we build something more *convenient* and *collaborative*?

Armeria documentation service

- Enabled by adding `DocService`
- Browse and invoke RPC services in an Armeria server
  - No fiddling with binary payloads
  - Send a request without writing code
- Supports gRPC, Thrift and annotated services
- We have a plan to add:
  - Metric monitoring console
  - Runtime configuration editor, e.g. logger level



Armeria documentation service 0.94.1

Go to ...

Services

Cassandra

POST	add()
POST	batch_mutate()
POST	describe_cluster_name()
POST	describe_keyspace()
POST	describe_keyspaces()
POST	describe_partitioner()
POST	describe_ring()
POST	describe_schema_versions()
POST	describe_snitch()
POST	describe_splits()
POST	describe_version()
POST	execute_cql_query()
POST	execute_prepared_cql_query()
POST	get()
POST	get_count()
POST	get_indexed_slices()
POST	get_range_slices()
POST	get_slice()
POST	insert()

Cassandra.add()

Increment or decrement a counter.

Parameters

Name	Required	Type	Description
key	required	binary	
column_parent	required	ColumnParent	
column	required	CounterColumn	
consistency_level	required	ConsistencyLevel	

Return Type

void

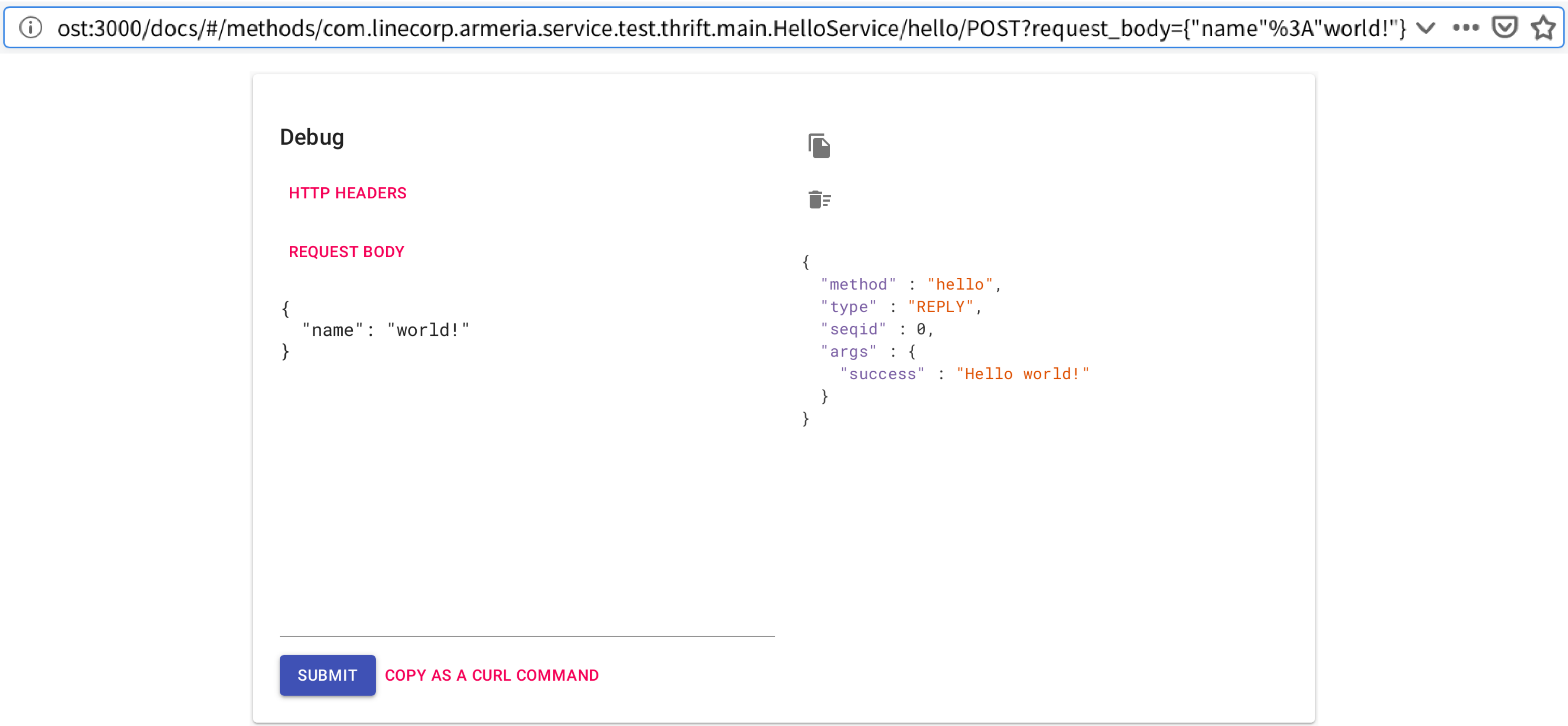
Exceptions

InvalidRequestException

TimedOutException

UnavailableException

- Share the URL to reproduce a call.



ost:3000/docs/#/methods/com.linecorp.armeria.service.test.thrift.main.HelloService/hello/POST?request_body={"name"%3A"world!"}

Debug

HTTP HEADERS

REQUEST BODY

```
{
  "name": "world!"
}
```

```
{
  "method" : "hello",
  "type" : "REPLY",
  "seqid" : 0,
  "args" : {
    "success" : "Hello world!"
  }
}
```

Cool features not available in upstream

- gRPC
  - Works on both HTTP/1 and 2
  - gRPC-Web support, i.e. can call gRPC services from JavaScript frontends
- Thrift
  - HTTP/2, TTEXT (human-readable REST-ish JSON)
- Can leverage Armeria decorators
  - Structured logging, Metric collection, Distributed tracing, Authentication
  - CORS, SAML, Request throttling, Circuit breakers, Automatic retries, ...

Cool features not available in upstream

- Can mix gRPC, Thrift, REST, Tomcat, Jetty, …
  - on a single HTTP port & single JVM
  - without any proxies
  - REST API
  - Exposing metrics
  - Traditional JEE webapps
  - Static files
  - Health-check requests from load balancers
- Share common logic between different endpoints!

Unopinionated integration & migration

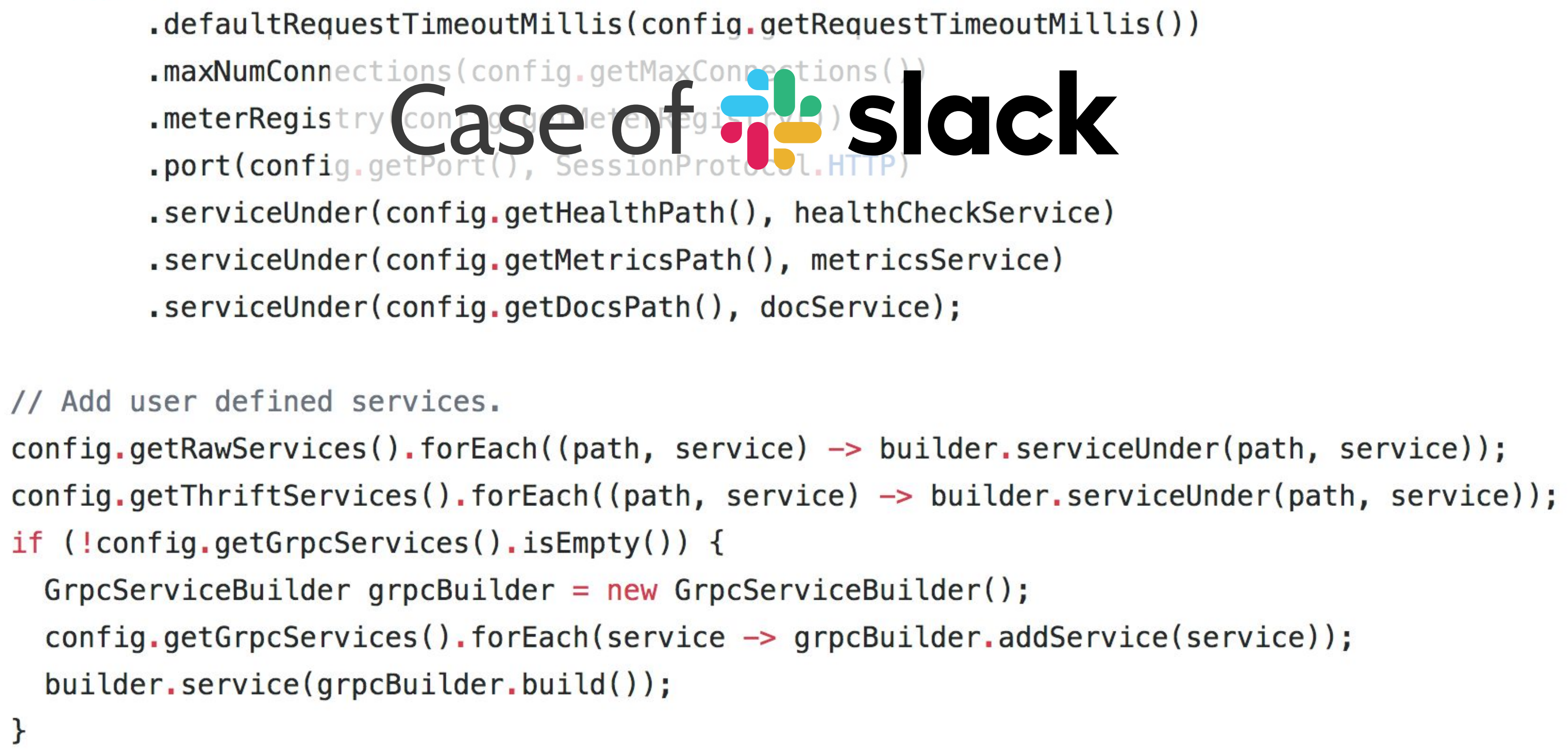
Armeria 😍 What You 😍

- Use your favorite tech, not ours:
  - DI – spring, Guice, Dagger, …
  - Protocols – gRPC, Thrift, REST, …
- Choose only what you want:
  - Most features are optional.
  - Compose and customize at your will.
    - Your application grows with you, not by its own.

Case of slack

- Using Thrift since 2015
- Migrated from Thrift to gRPC
  - Can run both while clients are switching
- Leverages built-in non-RPC services:
  - `PrometheusExpositionService`
  - `HealthCheckService`
  - `BraveService` – Distributed tracing with honeycomb
  - `DocService`

Case of slack

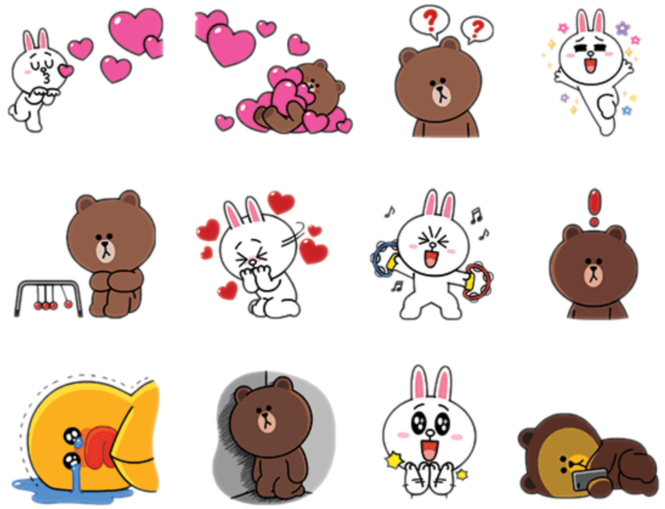


```
        .defaultRequestTimeoutMillis(config.getRequestTimeoutMillis())
        .maxNumConnections(config.getMaxConnections())
        .meterRegistry(config.getMeterRegistry())
        .port(config.getPort(), SessionProtocol.HTTP)
        .serviceUnder(config.getHealthPath(), healthCheckService)
        .serviceUnder(config.getMetricsPath(), metricsService)
        .serviceUnder(config.getDocsPath(), docService);

// Add user defined services.
config.getRawServices().forEach((path, service) -> builder.serviceUnder(path, service));
config.getThriftServices().forEach((path, service) -> builder.serviceUnder(path, service));
if (!config.getGrpcServices().isEmpty()) {
  GrpcServiceBuilder grpcBuilder = new GrpcServiceBuilder();
  config.getGrpcServices().forEach(service -> grpcBuilder.addService(service));
  builder.service(grpcBuilder.build());
}
```

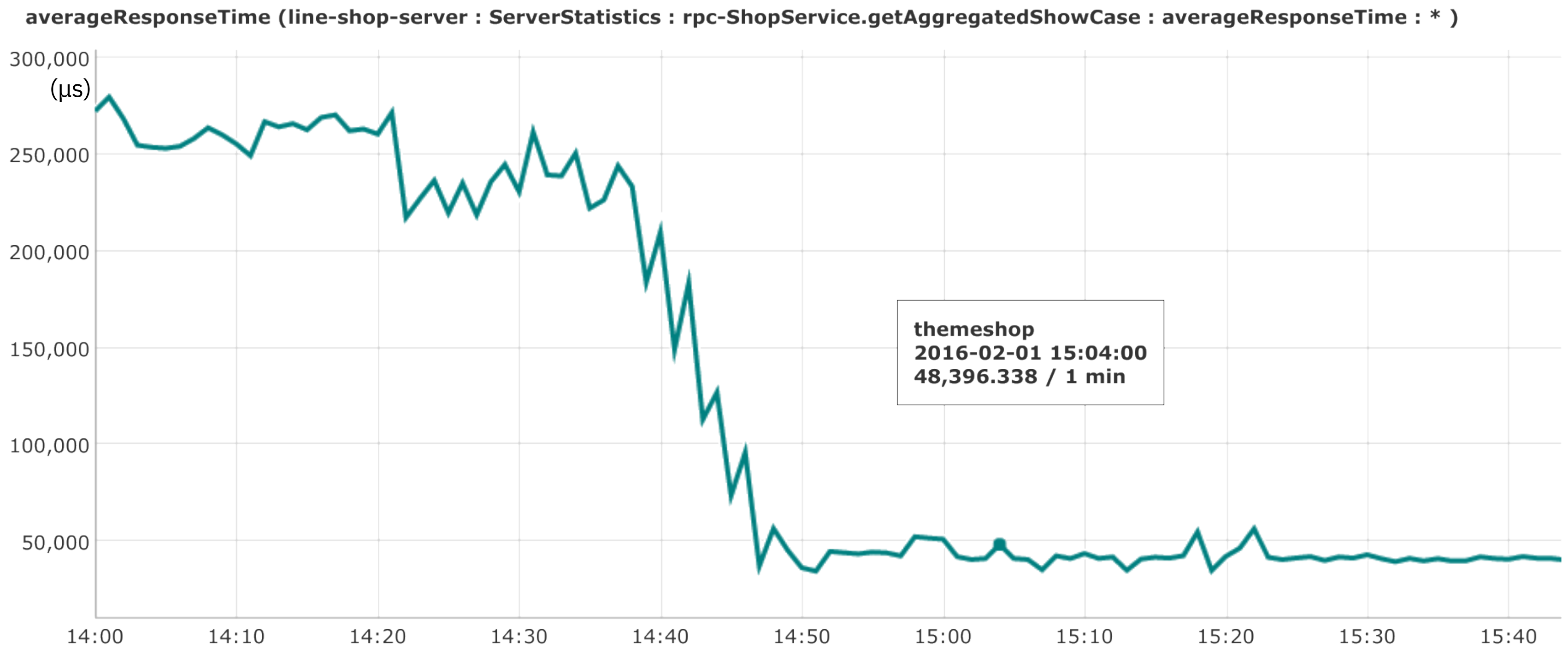
- Full migration story: https://sched.co/L715

Case of LINE

- In-app emoji · sticker store (50k-100k reqs/sec)
- Before:
  - Spring Boot + Tomcat (HTTP/1) + Thrift on Servlet
  - Apache HttpClient
- After – Migrate keeping what you love ✌️
  - Spring Boot + Armeria (HTTP/2)
  - Keep using Tomcat via `TomcatService` for the legacy
  - Thrift served directly & asynchronously = No Tomcat overhead
  - Armeria's HTTP/2 client w/ load-balancing



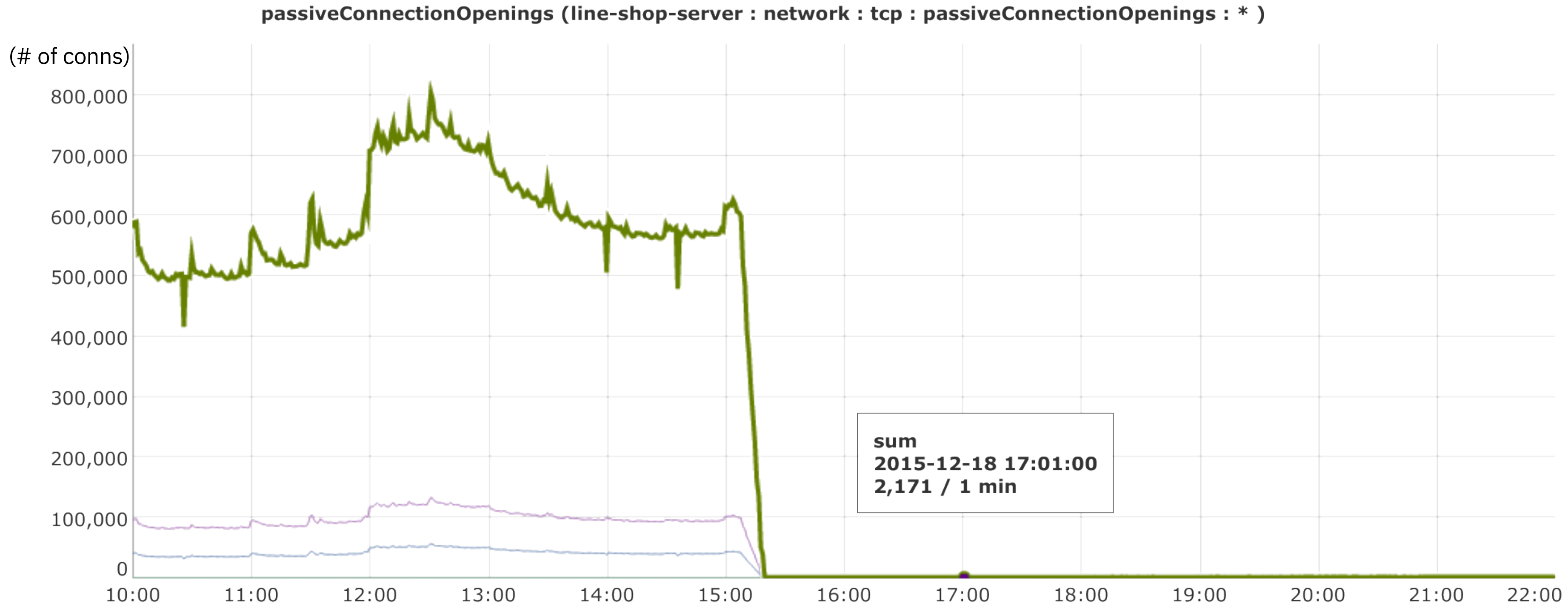
Case of LINE



- Asynchronification of 3 synchronous calls

@armeria_project line/armeria

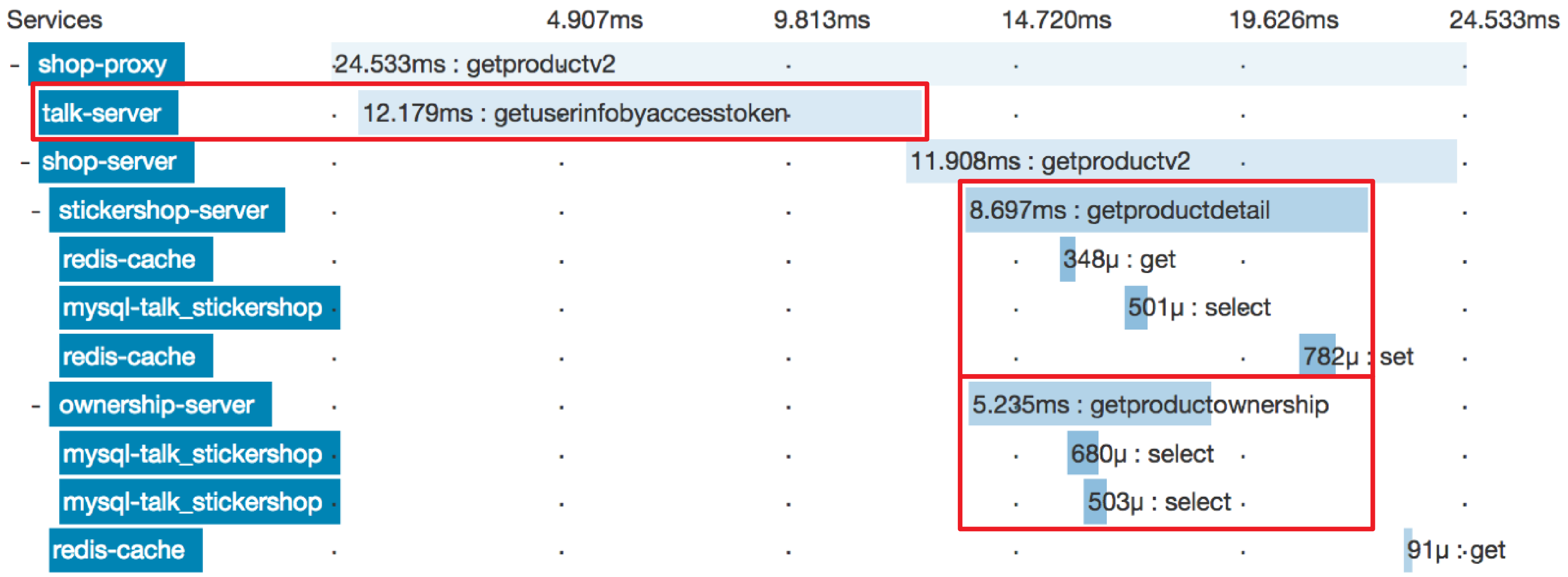
Case of LINE



- Significant reduction of inter-service connections

@armeria_project line/armeria

Case of LINE



Services		4.907ms	9.813ms	14.720ms	19.626ms	24.533ms
- shop-proxy	24.533ms : getproductv2					
talk-server	12.179ms : getuserinfobyaccesstoken					
- shop-server				11.908ms : getproductv2		
- stickershop-server				8.697ms : getproductdetail		
redis-cache				348μ : get		
mysql-talk_stickershop				501μ : select		
redis-cache					782μ : set	
- ownership-server				5.235ms : getproductownership		
mysql-talk_stickershop				680μ : select		
mysql-talk_stickershop				503μ : select		
redis-cache						91μ : get

- Distributed tracing with ZIPKIN by just adding `BraveService`
- Full story: https://www.slideshare.net/linecorp/line-zipkin

Case of kakaopay

- Firm banking gateway
  - Talking to Korean banks via VAN (value-added network)
- Kotlin + Armeria
  - Mostly non-null API
  - Using `@Nullable` annotation extensibly
- Spring WebFlux + gRPC
- Armeria Replaces Spring's network layer (`reactor-netty`)
- gRPC served directly = No WebFlux overhead

Less points of failure

Client-side load-balancing

Load balancers · Reverse proxies

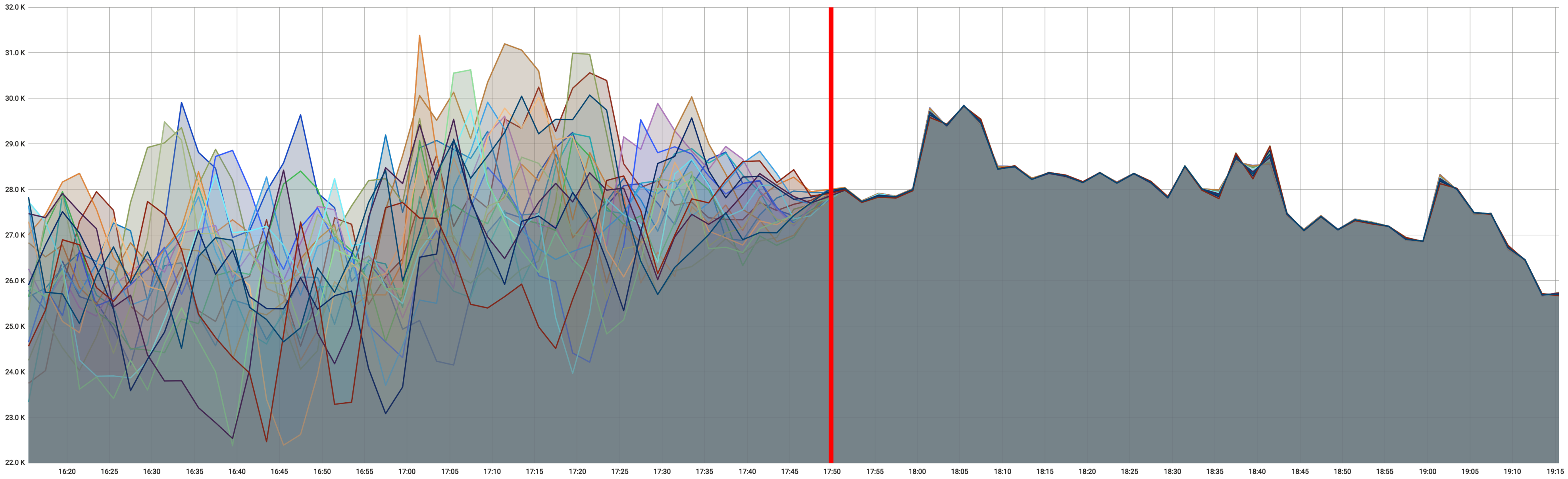
- Pros
  - Distributes load
  - Offloads TLS overhead
  - Automatic health checks
  - Service discovery (?)
- Cons
  - More points of failure
  - Increased hops · latency
  - Uneven load distribution
  - Cost of operation
  - Health check lags

Client-side load balancing

- Client-side load balancing
  - Chooses endpoints *autonomously*
  - Service discovery – DNS, **kubernetes**, APACHE ZooKeeper™, …
  - Near real-time health checks
  - Less points of failure
- Proxy-less Armeria server
  - OpenSSL-based high-performance TLS
  - Netty + `/dev/epoll`
  - Assemble your services into a single port + single JVM!

@armeria_project line/armeria

HTTP/2 load distribution at LINE



- Full migration story:
  https://speakerdeck.com/line_developers/lesson-learned-from-the-adoption-of-armeria-to-lines-authentication-system

Near real-time health check

- Leverage HTTP/2 + long-polling
  - Significantly reduced number of health check requests, e.g. every 10s vs. 5m
  - Immediate notification of health status
- Server considered unhealthy
  - On disconnection
  - On server notification, e.g. graceful shutdown, self-test failure
- Fully backwards-compatible
  - Activated only when server responds with a special header

Client-side load-balancing with auto-retry and circuit breaker in 8 lines

```java
// Kubernetes-style service discovery + long polling health check
EndpointGroup group = HealthCheckedEndpointGroup.of(
        DnsServiceEndpointGroup.of("my-service.cluster.local"),
        "/internal/healthcheck");
// Register the group into the registry.
EndpointGroupRegistry.register("myService", group, WEIGHTED_ROUND_ROBIN);
// Create an HTTP client with auto-retry and circuit breaker.
HttpClient client = HttpClient.builder("http://group:myService")
    .decorator(RetryingHttpClient.newDecorator(onServerErrorStatus()))
    .decorator(CircuitBreakerHttpClient.newDecorator(...))
    .build();
// Send a request.
HttpResponse res = client.get("/hello/armeria");
```

Future work

Consider joining us!

The road to 1.0 (and beyond)

- Currently at 0.95
- Hoping to release before the end of 2019
- API stabilization · clean-up
- Post-1.0
  - Kotlin · Scala DSL
  - Evolving `DocService` to `DashboardService`
  - More transports & protocols
    - Web Sockets, UNIX domain sockets, Netty handlers, …
  - More decorators
  - More service discovery mechanisms
    - Eureka, Consul, etcd, …
  - OpenAPI spec (`.yml`) generator
  - Performance optimization

Meet us at GitHub

github.com/line/armeria

line.github.io/armeria

@armeria_project line/armeria