Joker<?>

# Microservices design philosophy with ServiceTalk.io

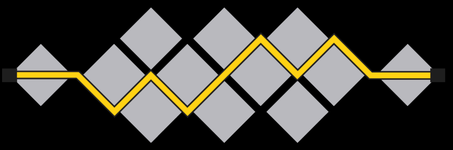
Idel Pivnitskiy
November 27, 2020

@idelpivnitskiy
idel.pivnitskiy@gmail.com

# Speaker


Netty

- Whole career around networks
- OSS fan
- Frameworks/libraries developer
- Contributor to:
  - ServiceTalk.io
  - Netty.io
  - AeroGear
- RFC8030 — Generic Event Delivery Using HTTP Push

I E T F

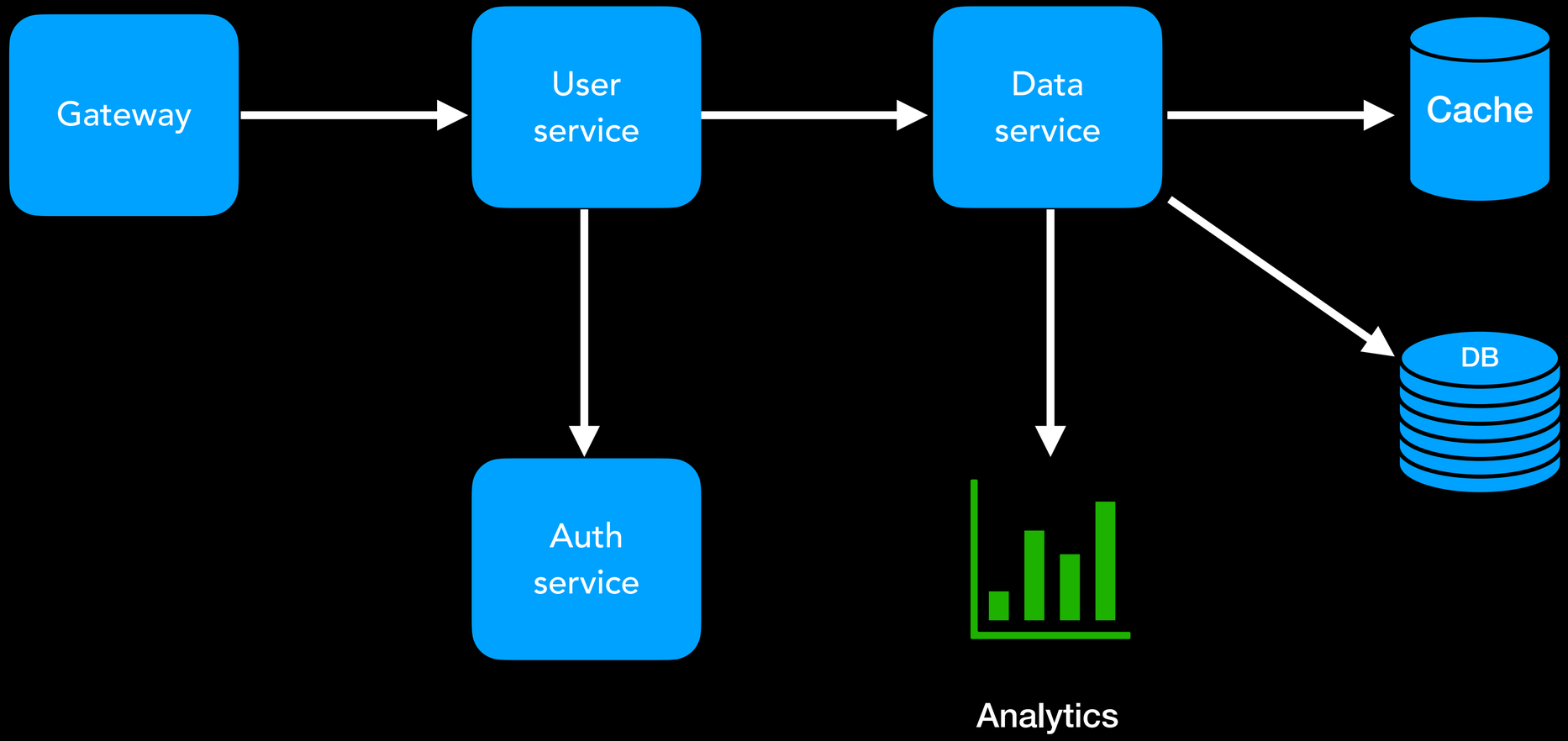
# Agenda

- Microservices world
- Frameworks grows and foundation
- Why do I need your Reactive Streams?
- ServiceTalk design principals
- Microservice evolution with ServiceTalk
- How to get started

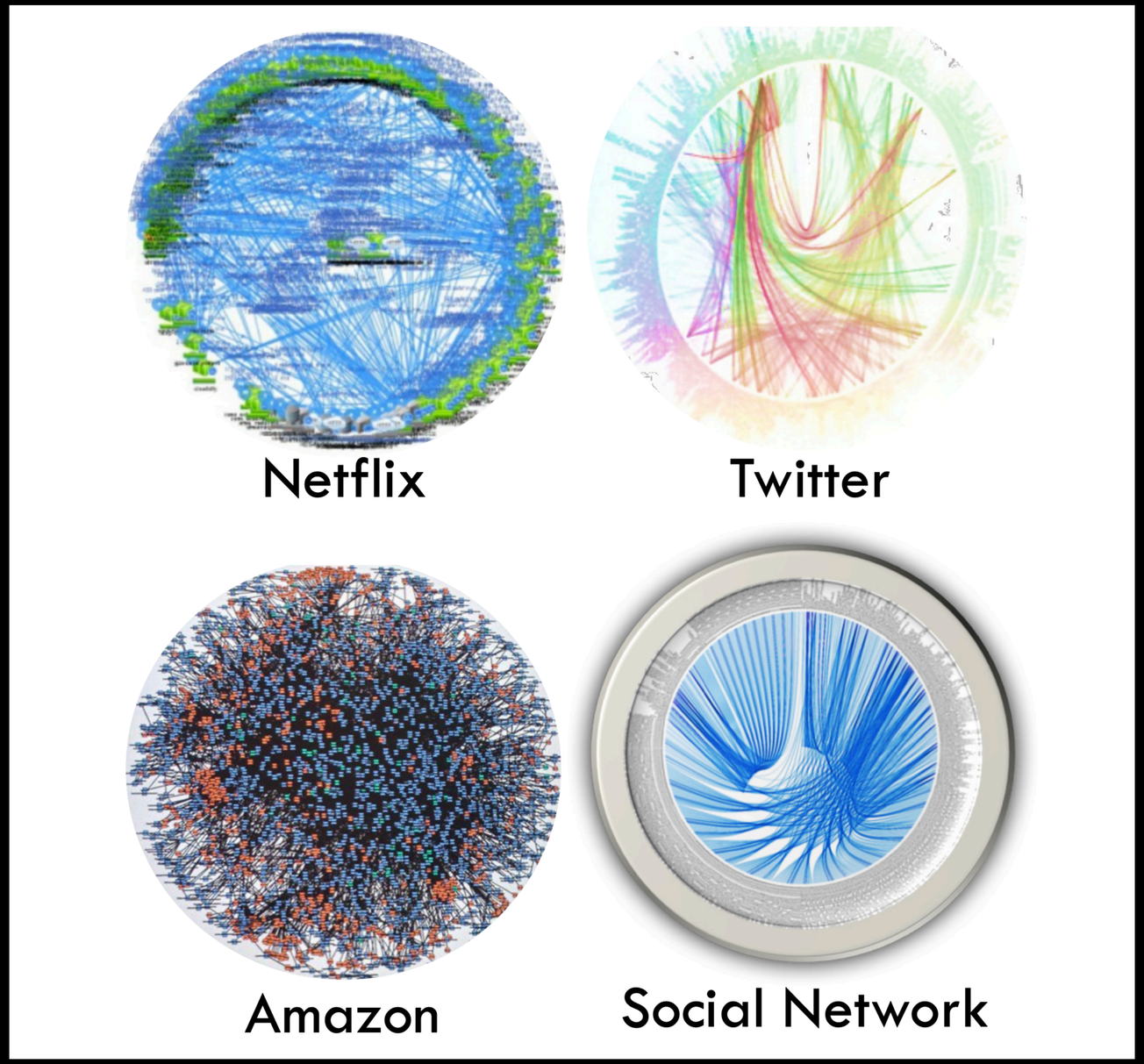
# Microservices world


https://medium.com/@_JuanGallo/dismantling-a-monolith-8486e6baaf69

# Microservices world, A.D.



# A few years later…




"An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems" Gan et al., ASPLOS'19

Whom should I talk to?

There is a failure!!!

Where is the failure?

I’m still processing…

What’s going on???

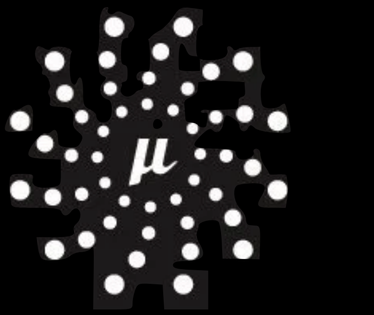
Take all these documents!

Why are you guys so slow?

# Frameworks grows and foundation

Reactor


Netty




VERT.X

QUARKUS

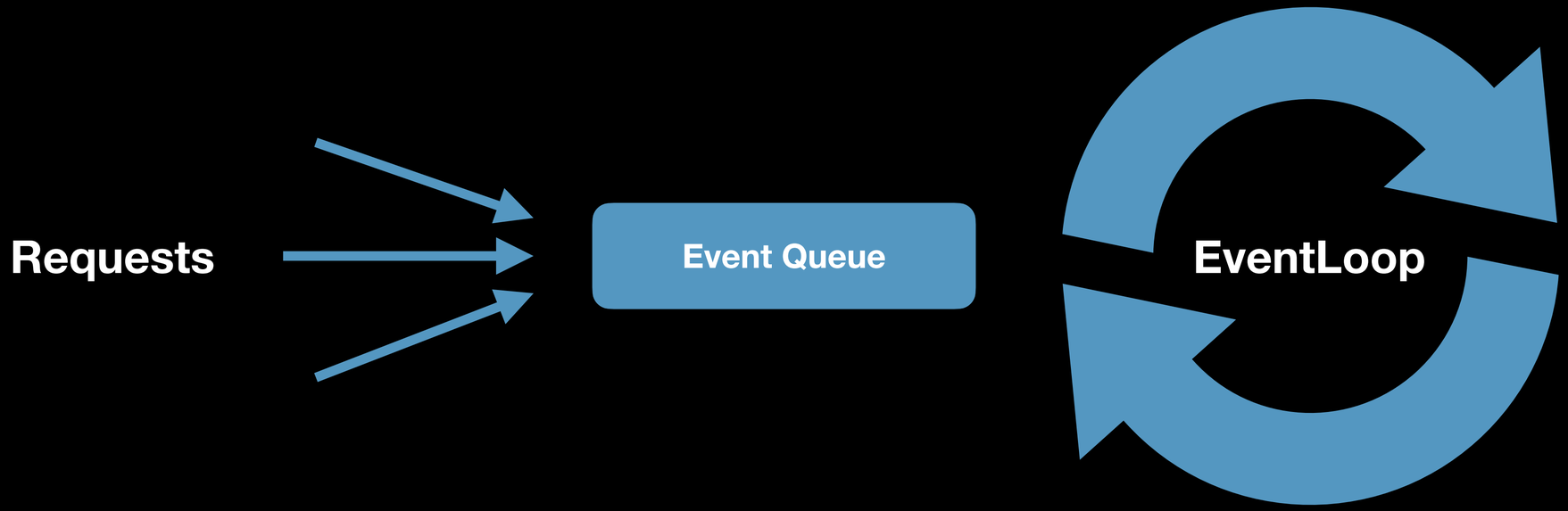
helidon.io

"Netty has emerged as the de-facto standard for network programming in Java"

*- Undertow Team*

http://undertow.io/blog/2019/04/15/Undertow-3.html

# Netty core

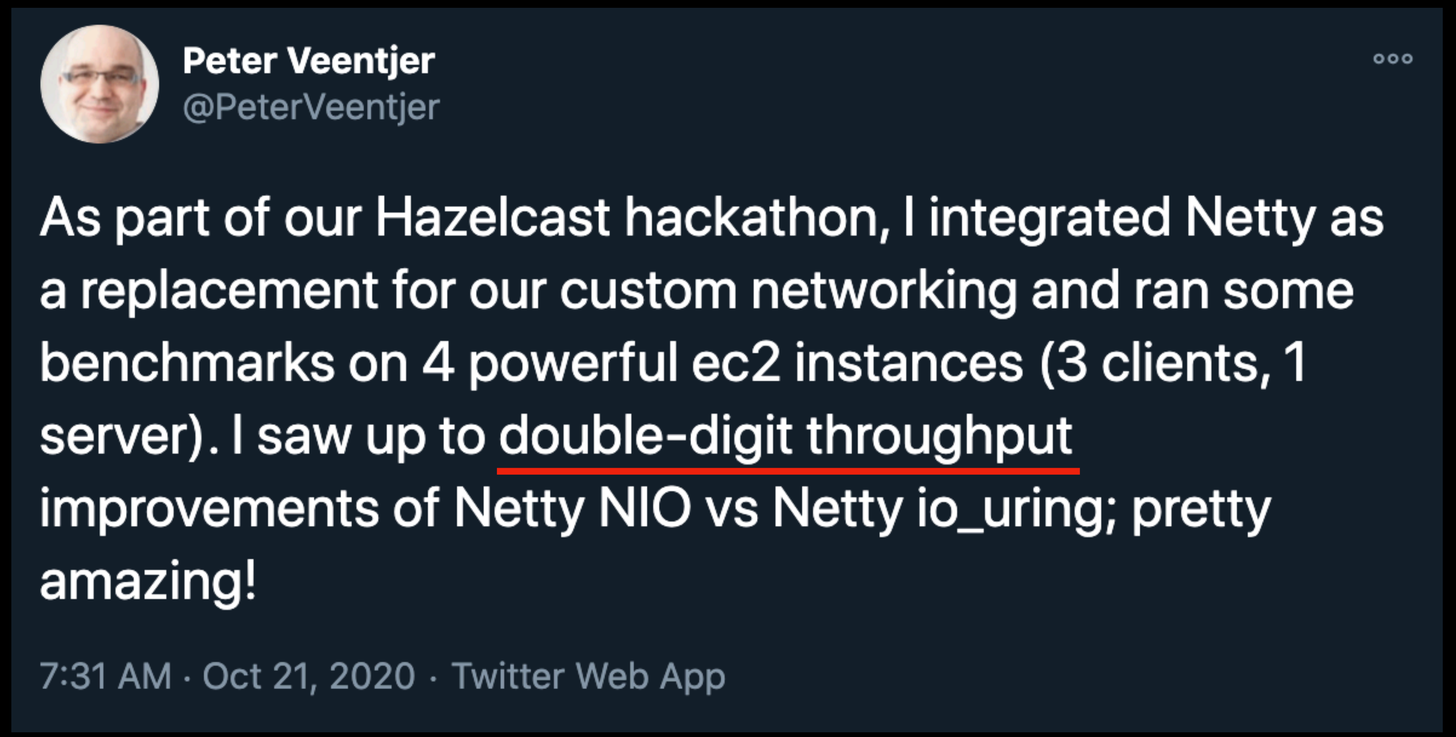



- Low number of threads
- No context switches
- High RPS
- Low latency

Fully asynchronous and event-driven

# Netty features

- Efficient memory allocation and control
- Highly optimized and feature rich native SSL/TLS engines
- Optimized transport
  - ~~Java NIO~~
  - Epoll / Kqueue
  - io_uring is coming
  - UnixDomainSocket




"Optimizations everywhere: Netty internals" by @normanmaurer, 2018

https://speakerdeck.com/normanmaurer/netty-internals-optimizations-everywhere

# Why not just to use Netty?

- Too low level
- Memory reference counting
- Manual back pressure management
- Requires deep knowledge of networking protocols
- Associating requests with responses, correlating errors
- No connection pooling
- No L7 (OSI) connection management

# What is ServiceTalk?

- Developed by Netty team
  - Gives best out of Netty
  - Hides complex
- Accumulates years of experience
  - OSS'ed in Nov 2019
- Proven to work under huge scale

# Why do I need your Reactive Streams?

# Reactive Streams

Standard API specification

```
package org.reactivestreams;

public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}

public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

www.reactive-streams.org

**Module** java.base
**Package** java.util.concurrent
**Class** Flow

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

# Reactive Streams

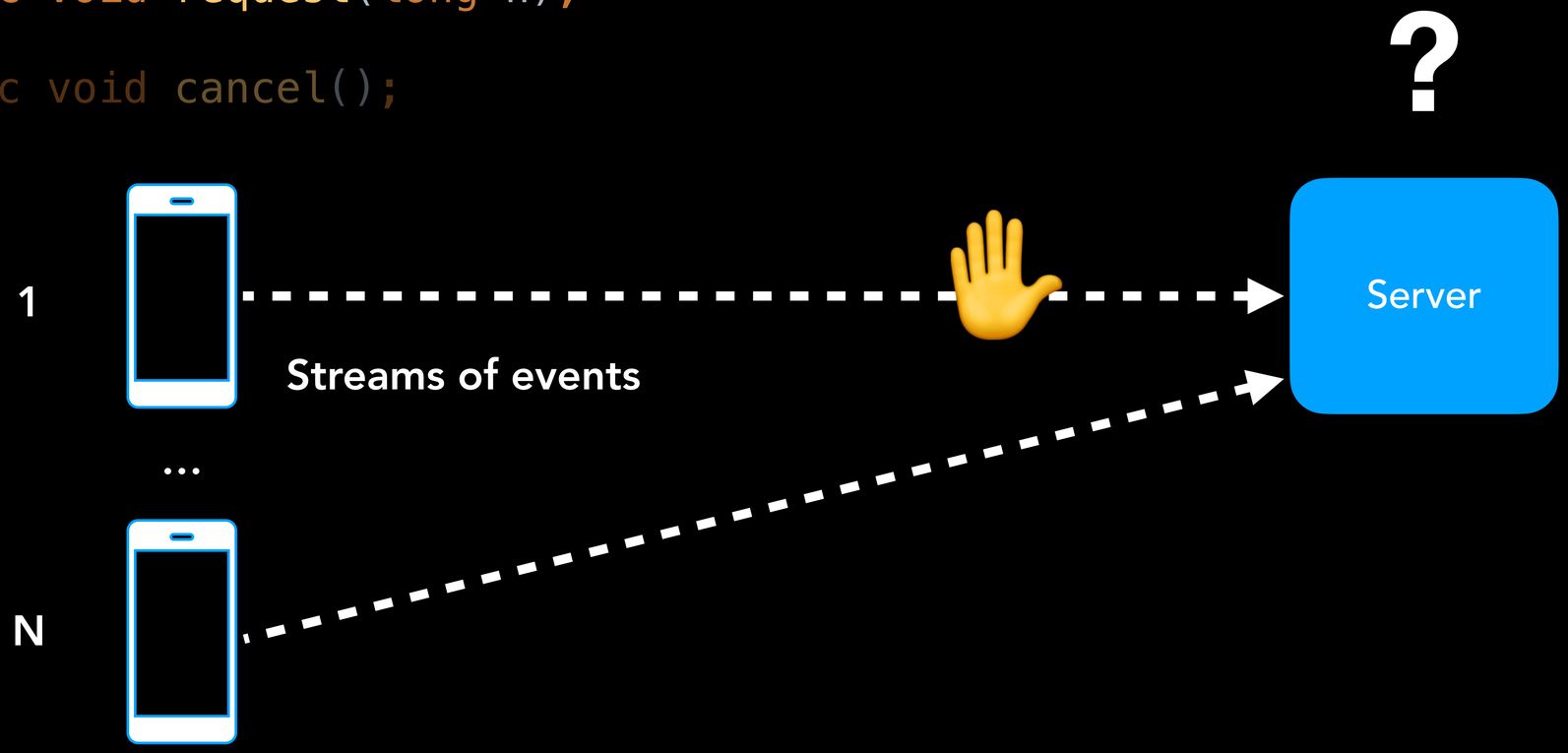
Standard API specification

Asynchronous parallel execution

Back pressure

# Back pressure

```
public interface Subscription {

    public void request(long n);

    public void cancel();
}
```



# Reactive Streams

Standard API specification

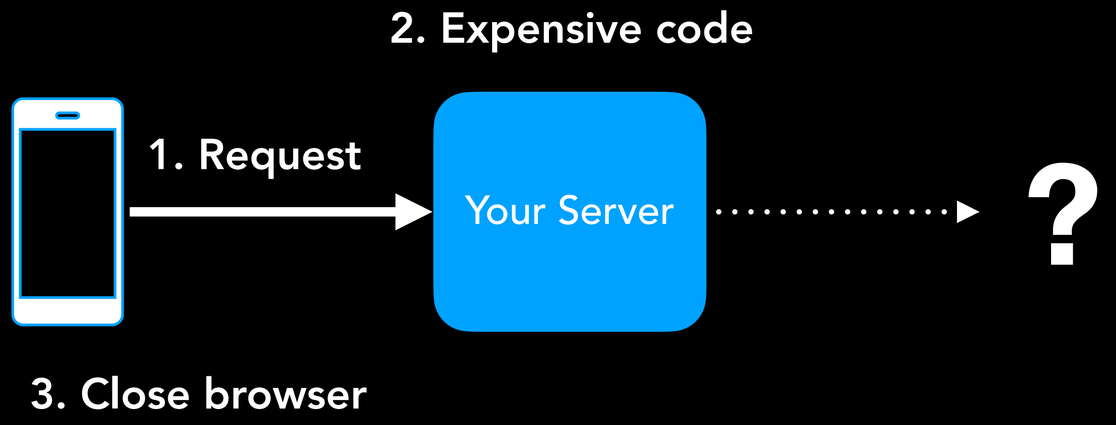
Asynchronous parallel execution

Back pressure

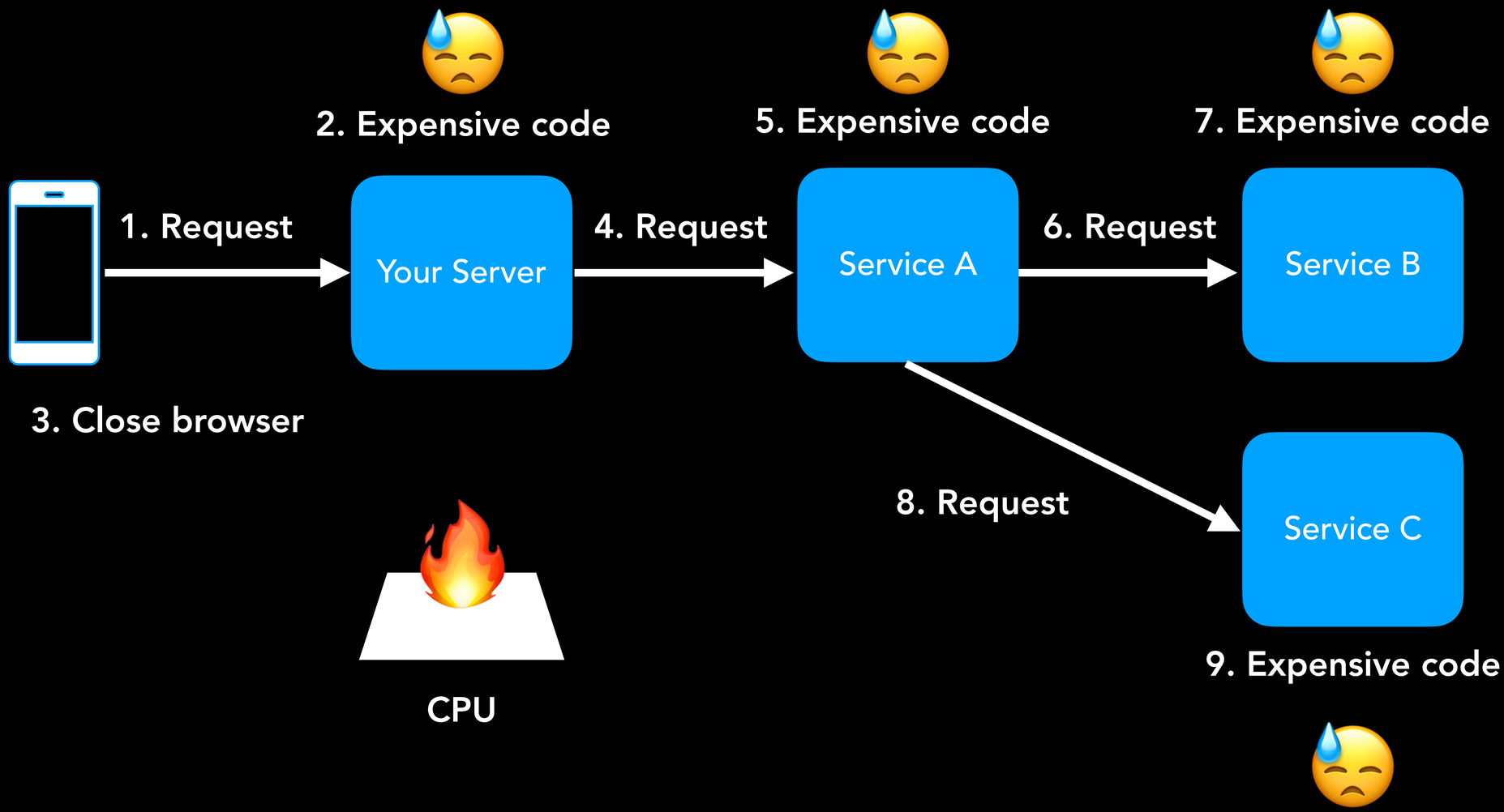
Cancellation

# Cancellation

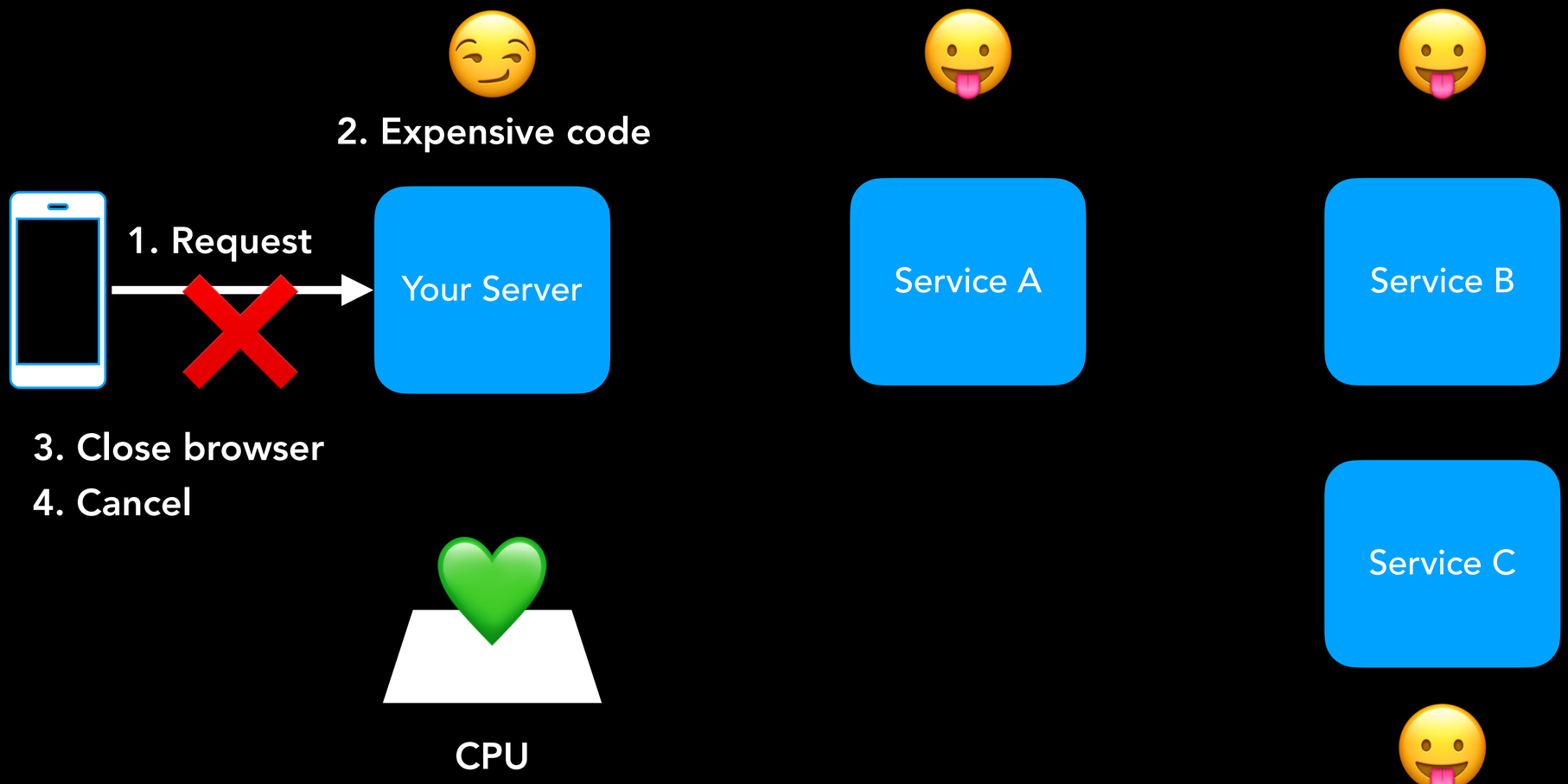
```
public interface Subscription {

    public void request(long n);

    public void cancel();
}
```



# Cancellation



# Cancellation



# Reactive Streams

Standard API specification

Asynchronous parallel execution

Back pressure

Cancellation

"Adventures with Reactive Streams" by @NiteshKant and @jovoordeckers, 2018

https://speakerdeck.com/jayv/adventures-with-reactive-streams

# Reactive Streams Challenges

Function composition

Not everything is a stream

Blocking safety

Debuggability

"Reactive programming: lessons learned" by @tnurkiewicz, 2019

https://2019.jpoint.ru/talks/57wq8ab5ytwmw7pdjvxcjd/

# The biggest challenge

Learning new concepts
and changing development style

# ServiceTalk design principals

# Main principles

Low barrier
to entry


## Reactive Streams

```
Single<Pojo> pojo = client.request(req)
    .map(resp -> {
        if (resp.status() != OK)
            throw error("Not OK");
        return resp.payloadBody(deserializer);
    });
```

## Imperative

```
HttpResponse resp = client.request(req);
if (resp.status() != OK)
    throw error("Not OK");
Pojo pojo = resp.payloadBody(deserializer);
```

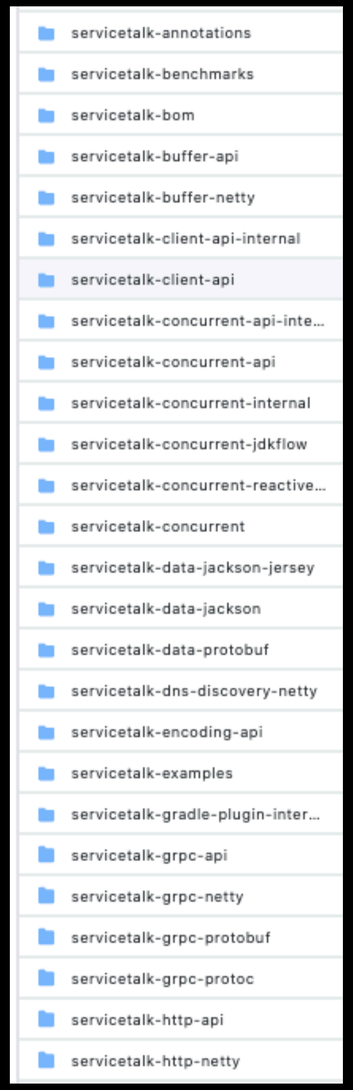
# Main principles

Low barrier
to entry

Cross-protocol
API symmetry

Extensible
modular core

# Project split into 50 modules

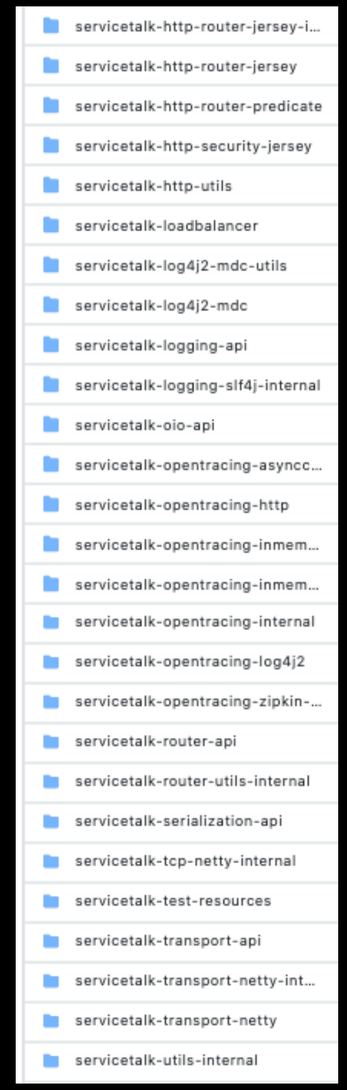



-api: 11 modules

-impl-name: 16 modules

-internal: 11 modules

other: 12 modules



# Main principles

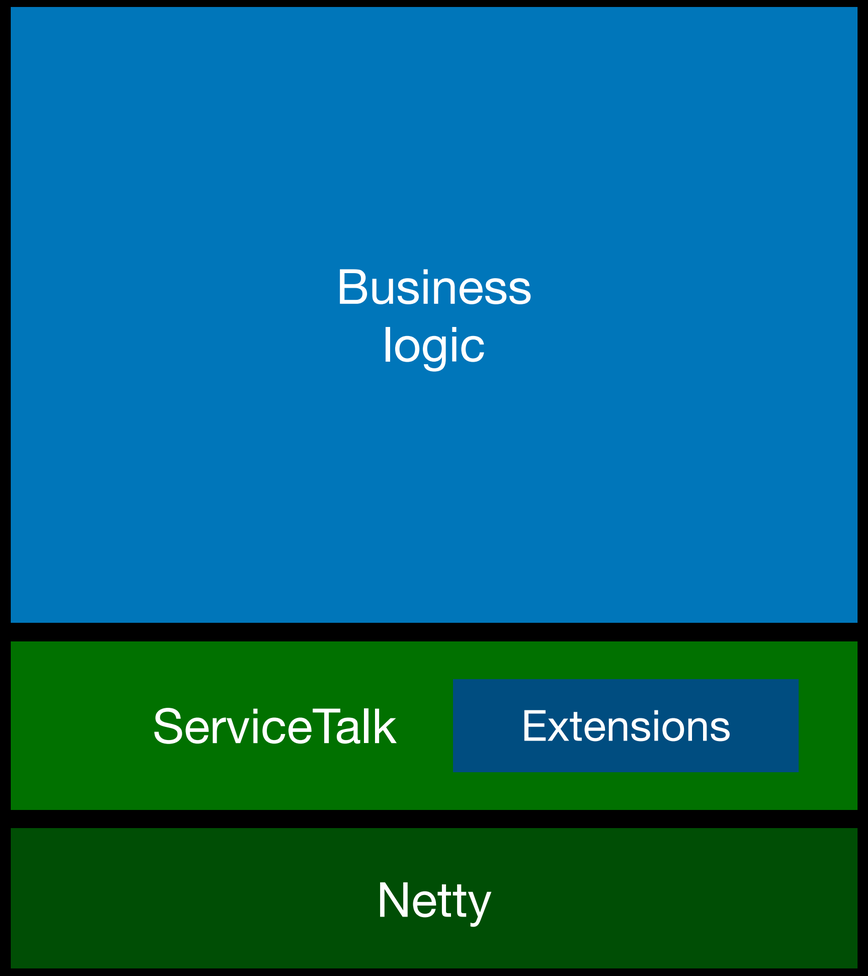
Low barrier
to entry

Cross-protocol
API symmetry

Extensible
modular core

Evolves with
your application

# ServiceTalk




Multiple protocols:

- HTTP/1.x, HTTP/2.0, gRPC, TCP.

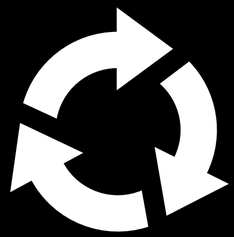
Multiple programming paradigms:

- Blocking / Asynchronous
- Aggregated / Streaming

Enables microservices:

- Service Discovery
- Load Balancing
- Observability
- Auto-retries
- Extensibility

Reactive core



# API variance

| | **Aggregated** | **Streaming** |
|---|---|---|
| **Blocking** | Servlet<br>Apache HTTP Client | Servlet<br>Apache HTTP Client |
| **Asynchronous** | Finagle, Reactor<br>CompletionStage | Netty<br>Reactor |

# Service as a Function

```
(ctx, request, responseFactory) ->
        responseFactory.ok().payloadBody("Hello World!", textSerializer())
```

# Service as a Function

Learn more about programming paradigms

Blocking aggregated

```
(ctx, request, responseFactory) ->
        responseFactory.ok().payloadBody("Hello World!", textSerializer())
```

Blocking streaming

```
(ctx, request, response) -> {
        try (HttpPayloadWriter<String> payloadWriter =
                     response.sendMetaData(textSerializer())) {
            payloadWriter.write("Hello ");
            payloadWriter.write("World!");
        }
}
```

Async aggregated

```
(ctx, request, responseFactory) ->
        Single.succeeded(responseFactory.ok()
                .payloadBody("Hello World!", textSerializer()))
```

Async streaming

```
(ctx, request, responseFactory) ->
        Single.succeeded(responseFactory.ok()
                .payloadBody(Publisher.from("Hello ", "World!"),
                        textSerializer()))
```

# Unified API

API variants share:

- HttpMetaData (version, headers, etc.)
- Conversion between APIs

Client and server share:

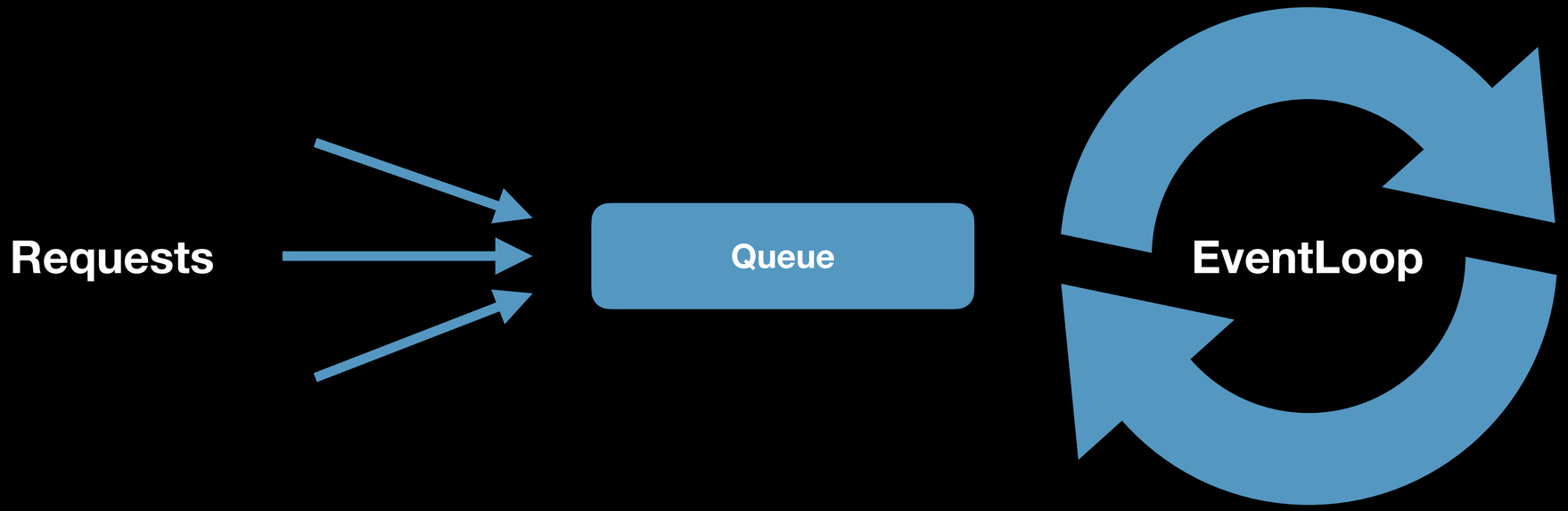
- The same HTTP classes (request / response):
  - Reusable objects
- IO and worker executors

Protocols share:

- Operators
- Filters
- LoadBalancer
- ServiceDiscoverer
- Programming paradigms (API variants)

# Offloading or
# Can I block right here?

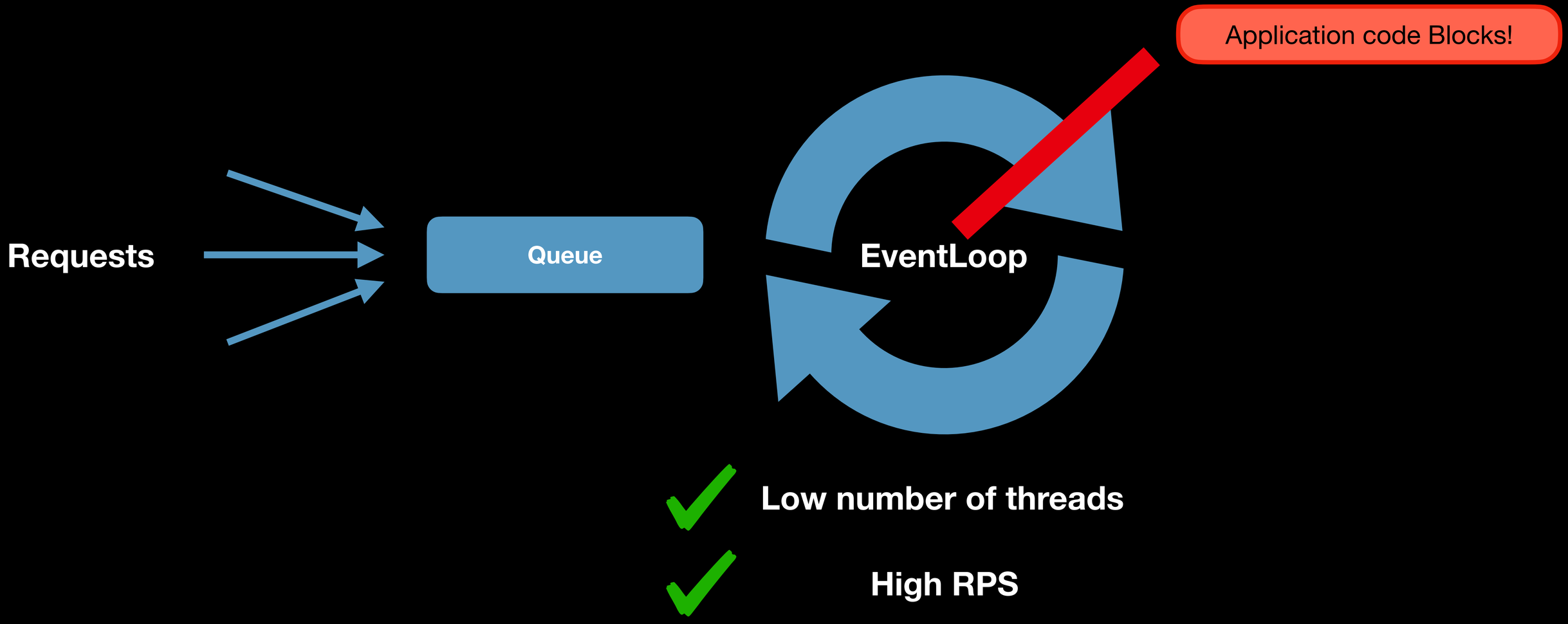
# Asynchronous execution




✓ Low number of threads

✓ High RPS

# Asynchronous execution




✓ Low number of threads

✓ High RPS

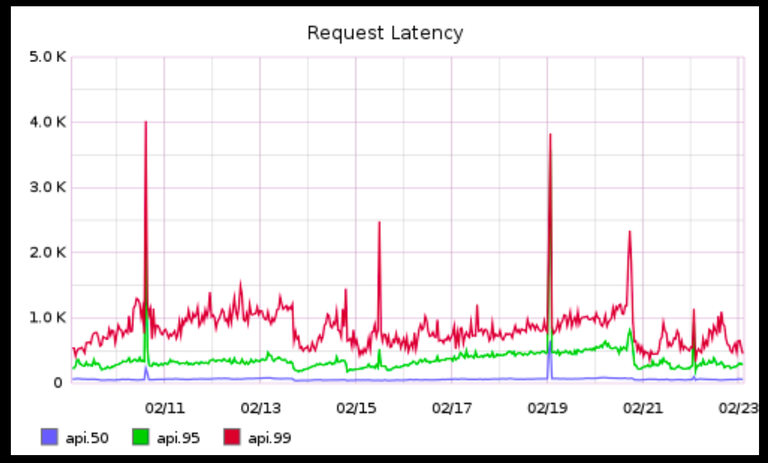
# Asynchronous execution

Application code Blocks!

Requests

Queue

EventLoop

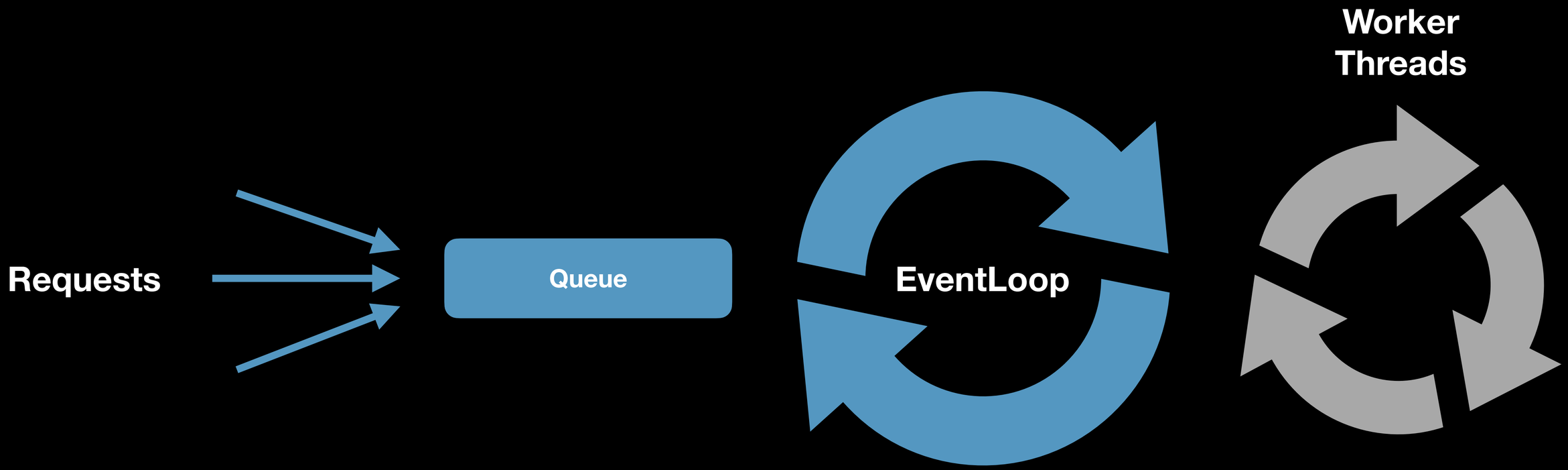



✓ Low number of threads

✓ ~~High RPS~~

✗ Latency spikes

# Safe execution

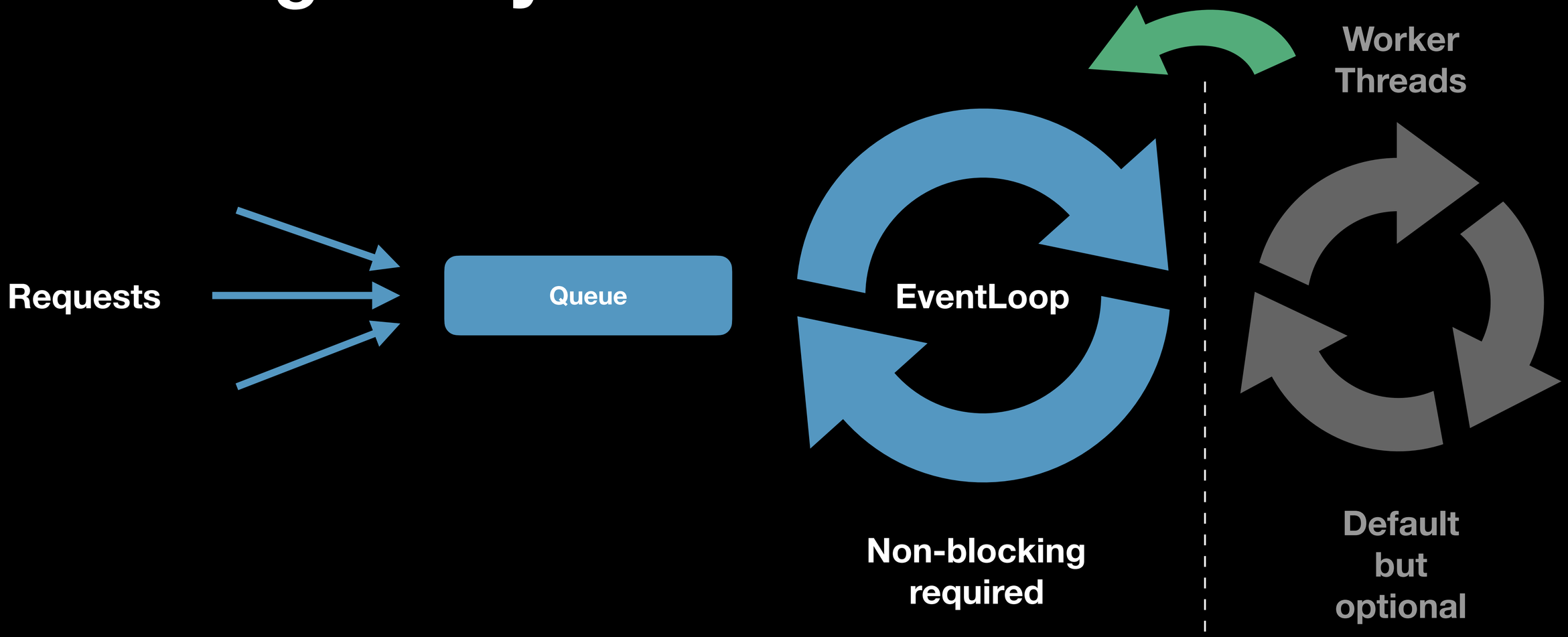



ServiceTalk.io
Jetty
Tomcat

**Safe to block by default** 💥

Docs: Blocking safe by default

# Evolving to asynchronous




Docs: Evolving to asynchronous

# Execution strategy control

```
public final class HttpExecutionStrategies {

   public static HttpExecutionStrategy defaultStrategy() {
        return Builder.DEFAULT;
    }

    public static HttpExecutionStrategy noOffloadsStrategy() {
        return NO_OFFLOADS_NO_EXECUTOR;
    }

    public static Builder customStrategyBuilder() {
        return new Builder();
    }
}
```

```
public interface HttpExecutionStrategyInfluencer {

    HttpExecutionStrategy influenceStrategy(
            HttpExecutionStrategy strategy);
}
```

ServiceTalk computes optimal ExecutionStrategy based on:

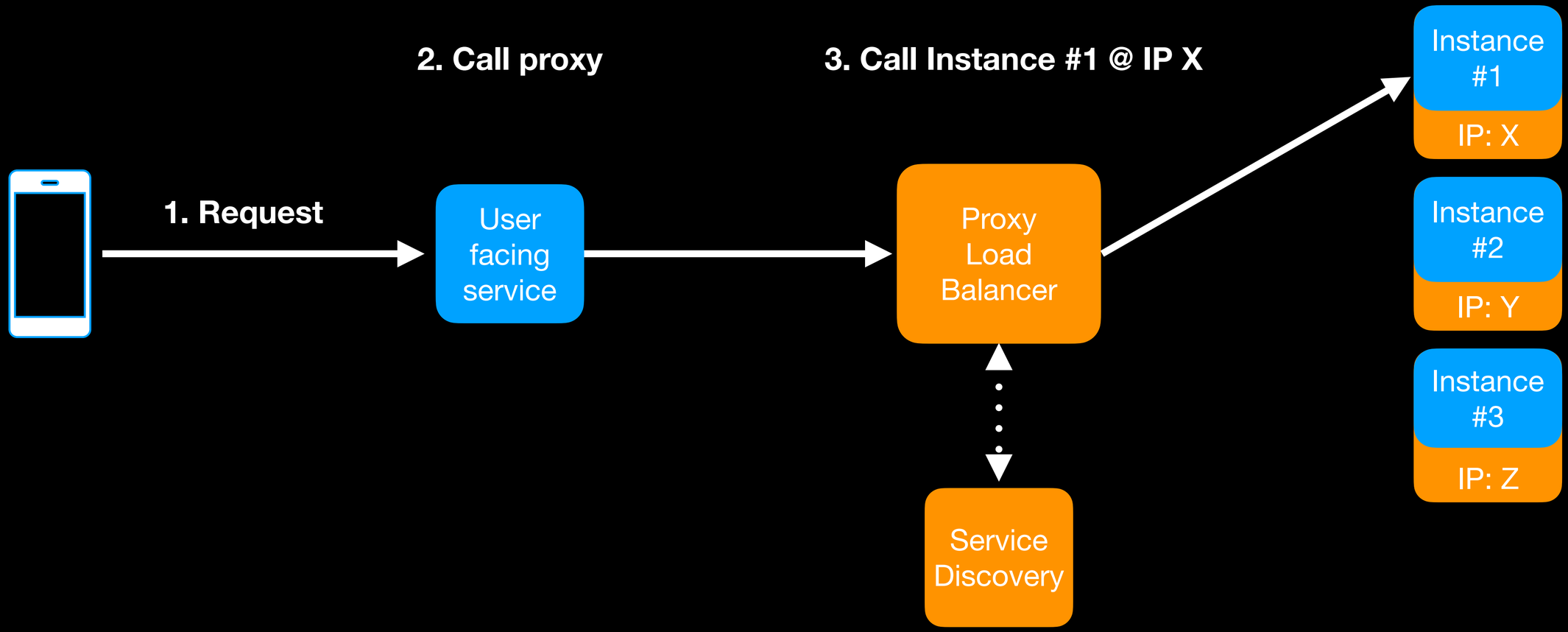
- used API variant
- inserted filters and extensions
- per-route configuration

Blocking safe by default

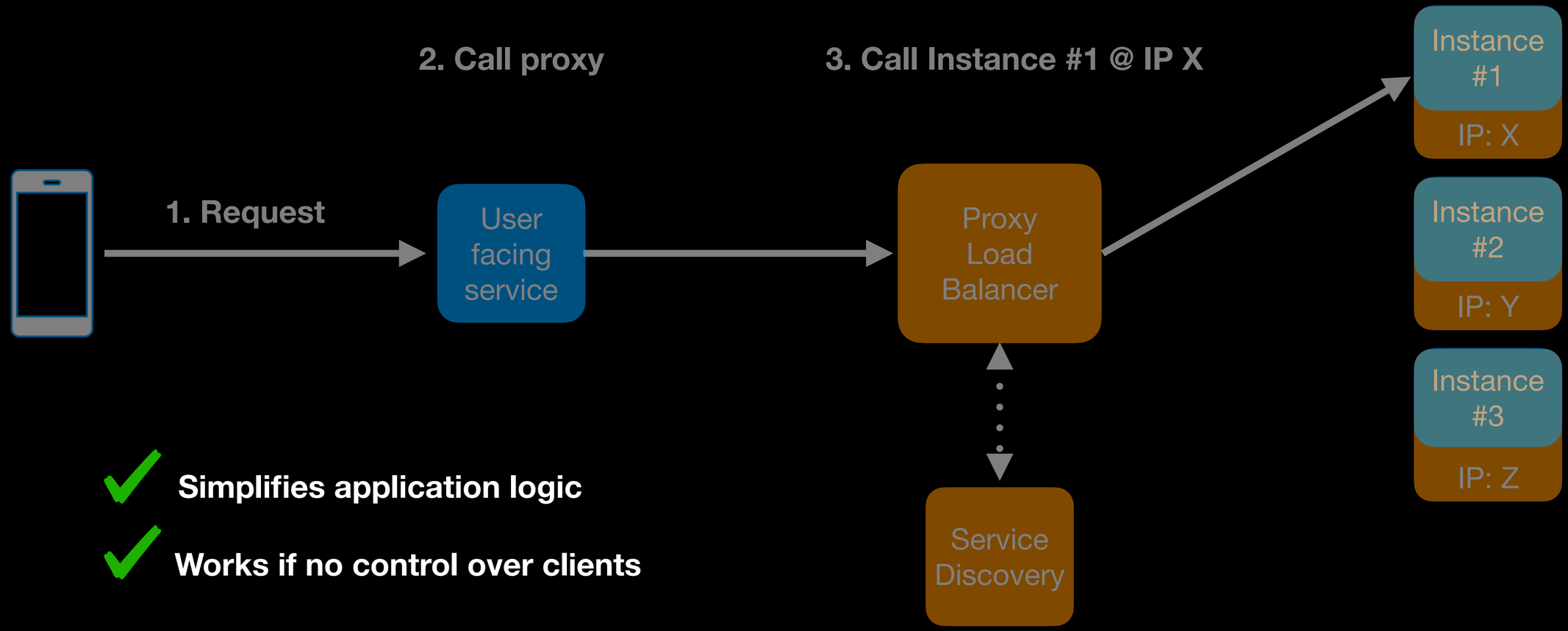
Evolving to asynchronous

# Smart-client:
# Load balancing & Service Discoverer

# Load balancing approach 1: Proxy

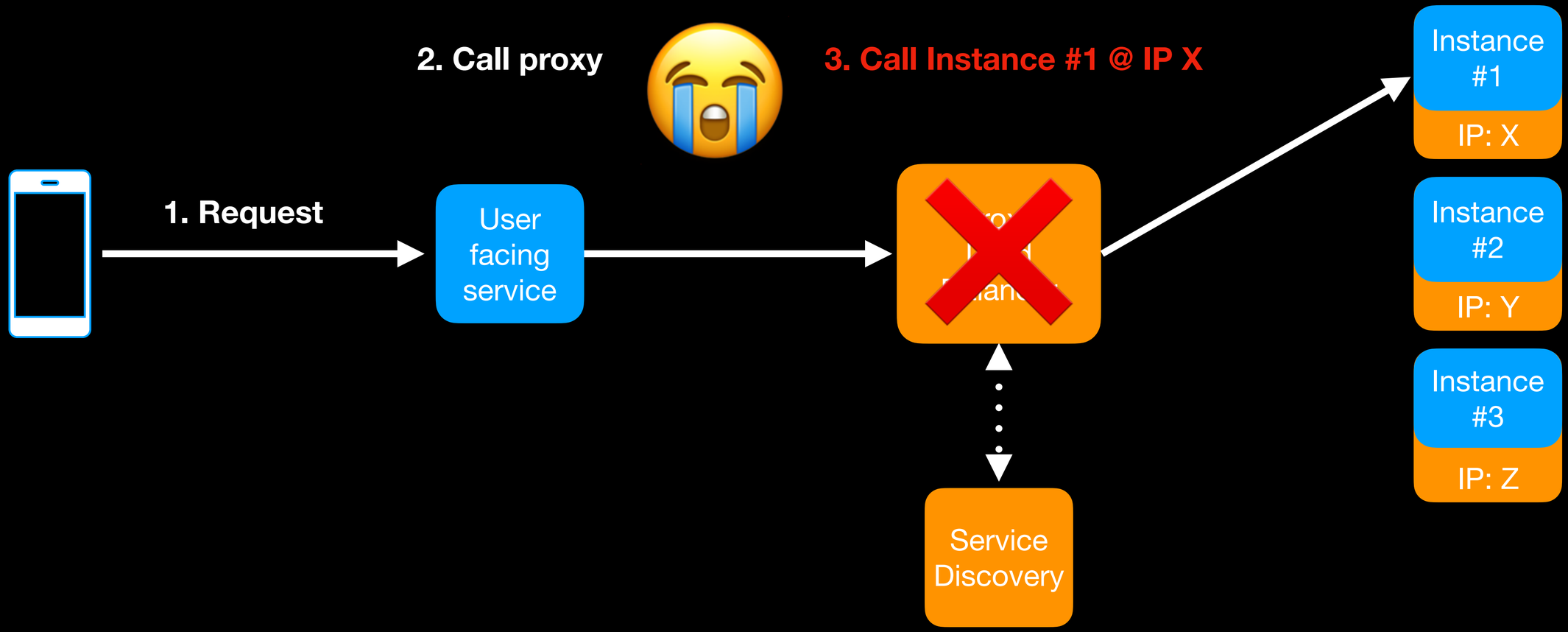


# Load balancing approach 1: Proxy

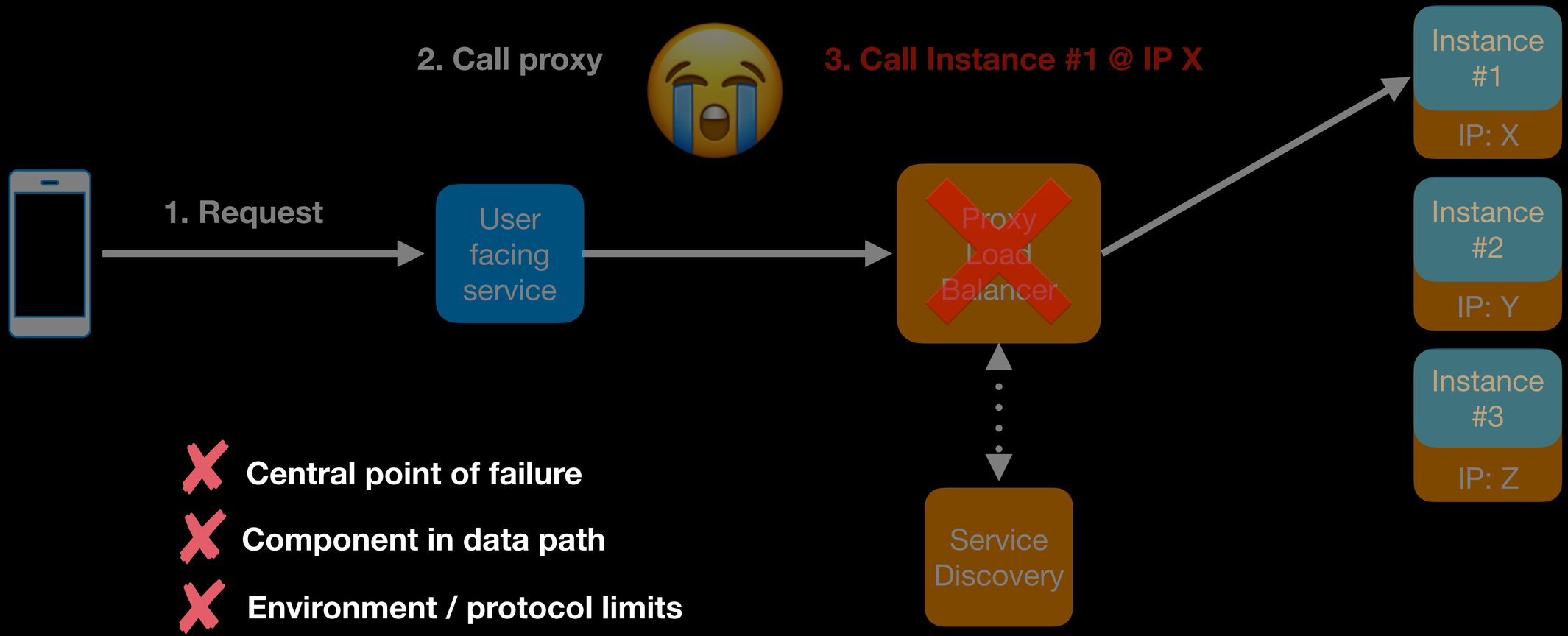



- ✓ **Simplifies application logic**
- ✓ **Works if no control over clients**

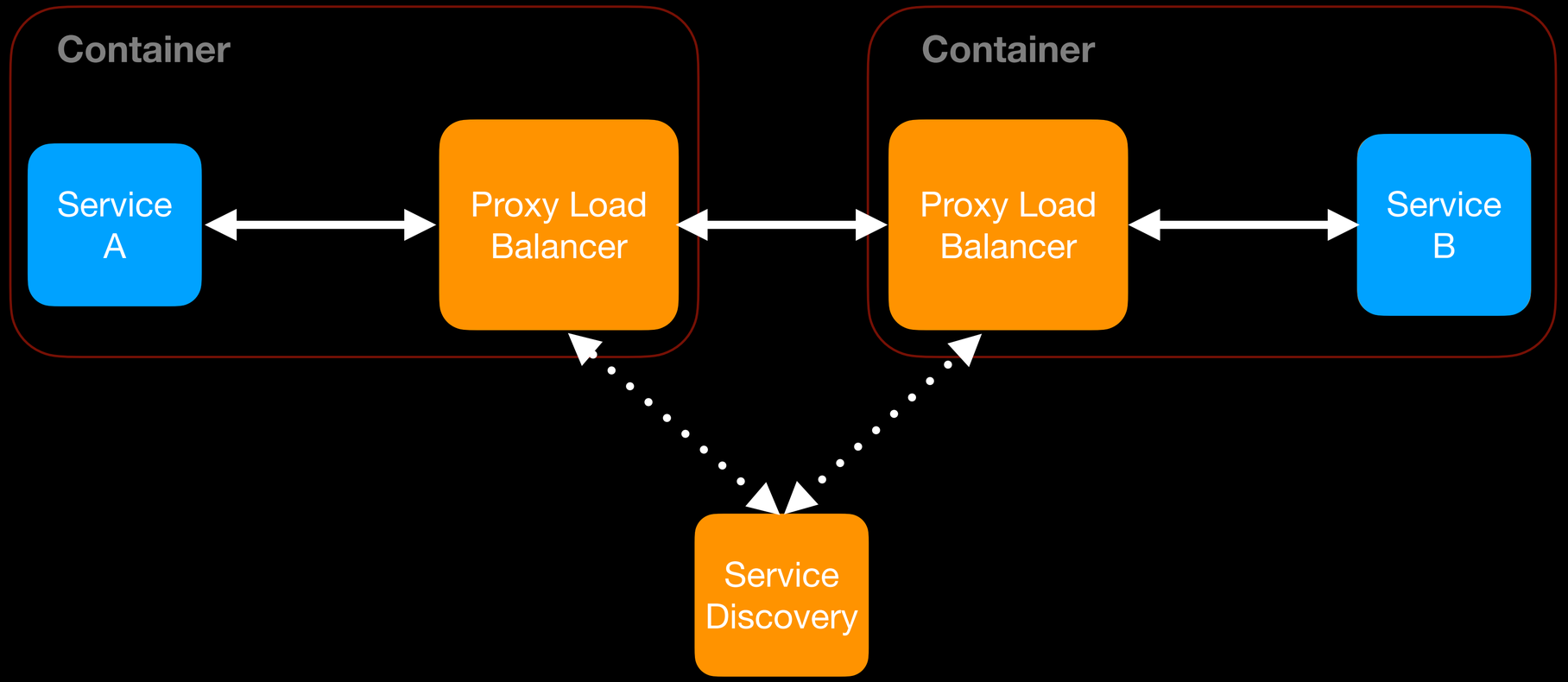
# Load balancing approach 1: Proxy



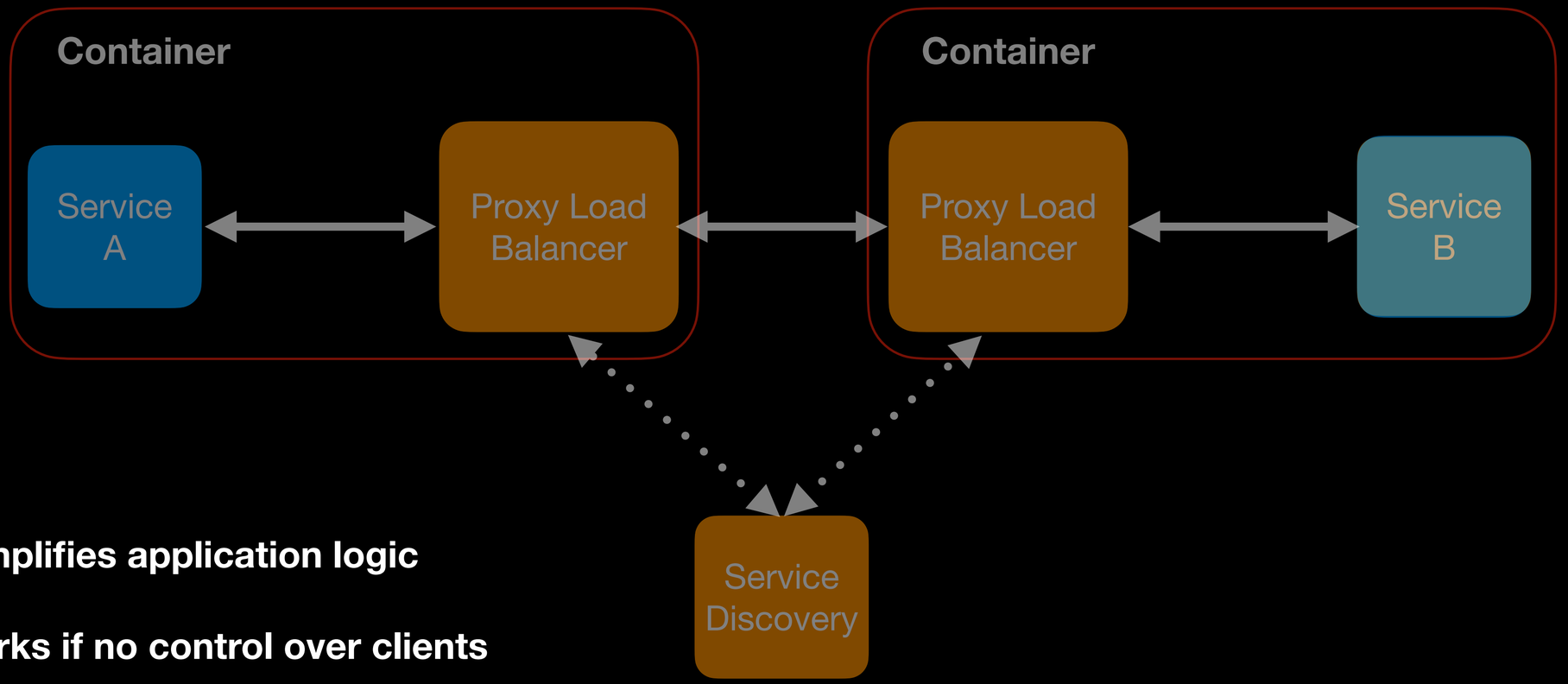
# Load balancing approach 1: Proxy



# Load balancing approach 2: Service mesh

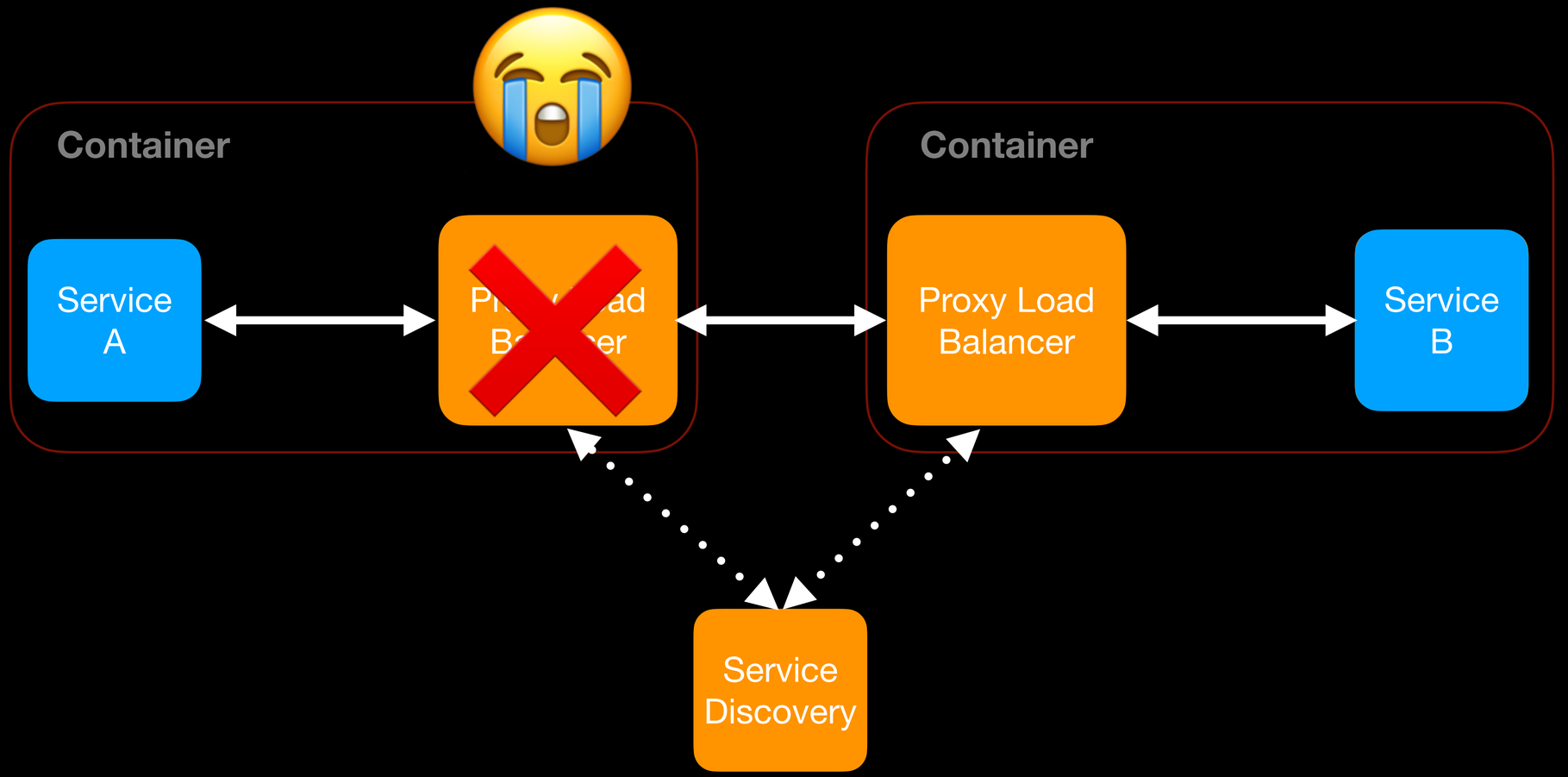


# Load balancing approach 2: Service mesh

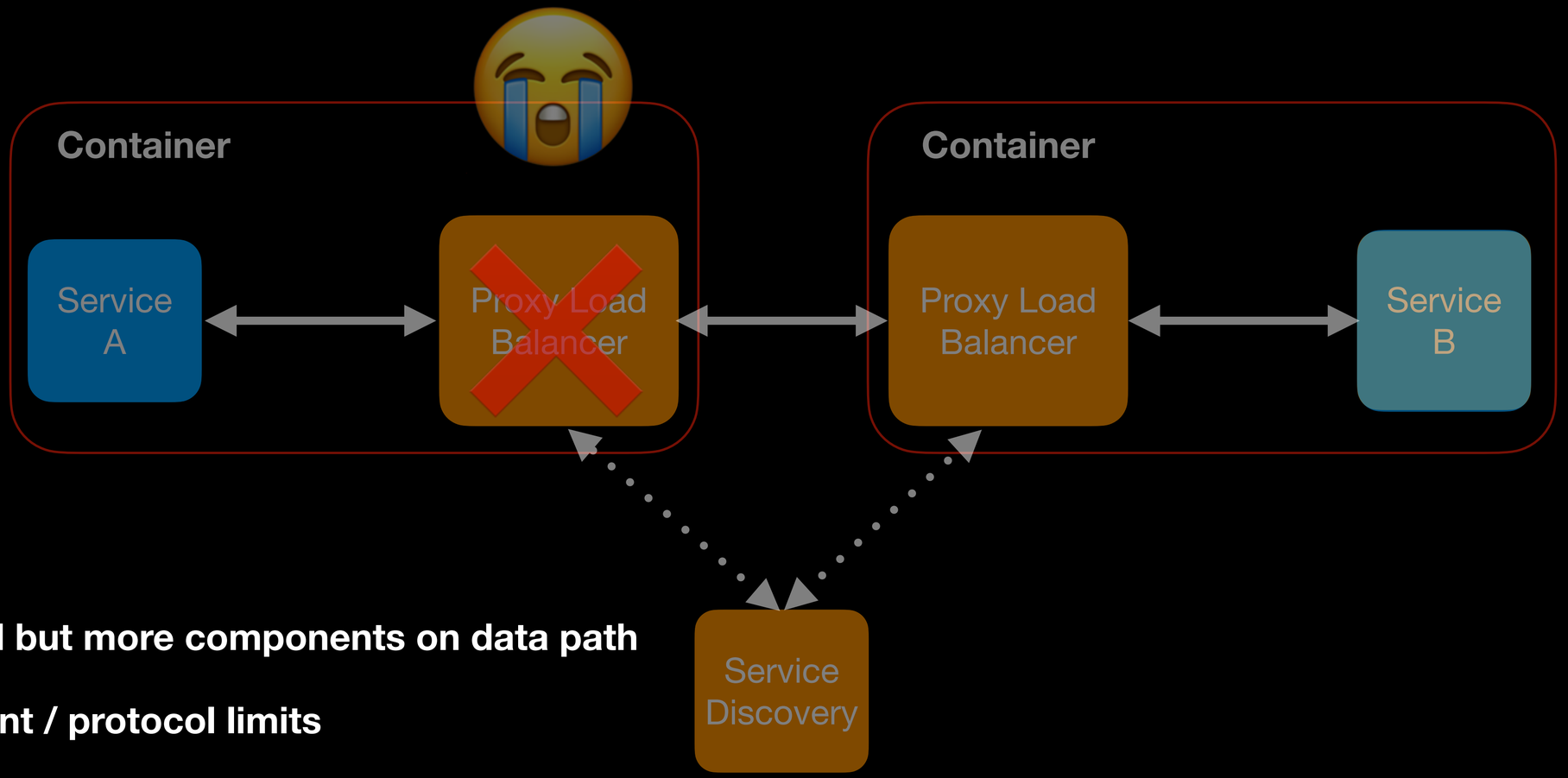



- ✓ **Simplifies application logic**
- ✓ **Works if no control over clients**
- ✓ **No central point of failure**

# Load balancing approach 2: Service mesh

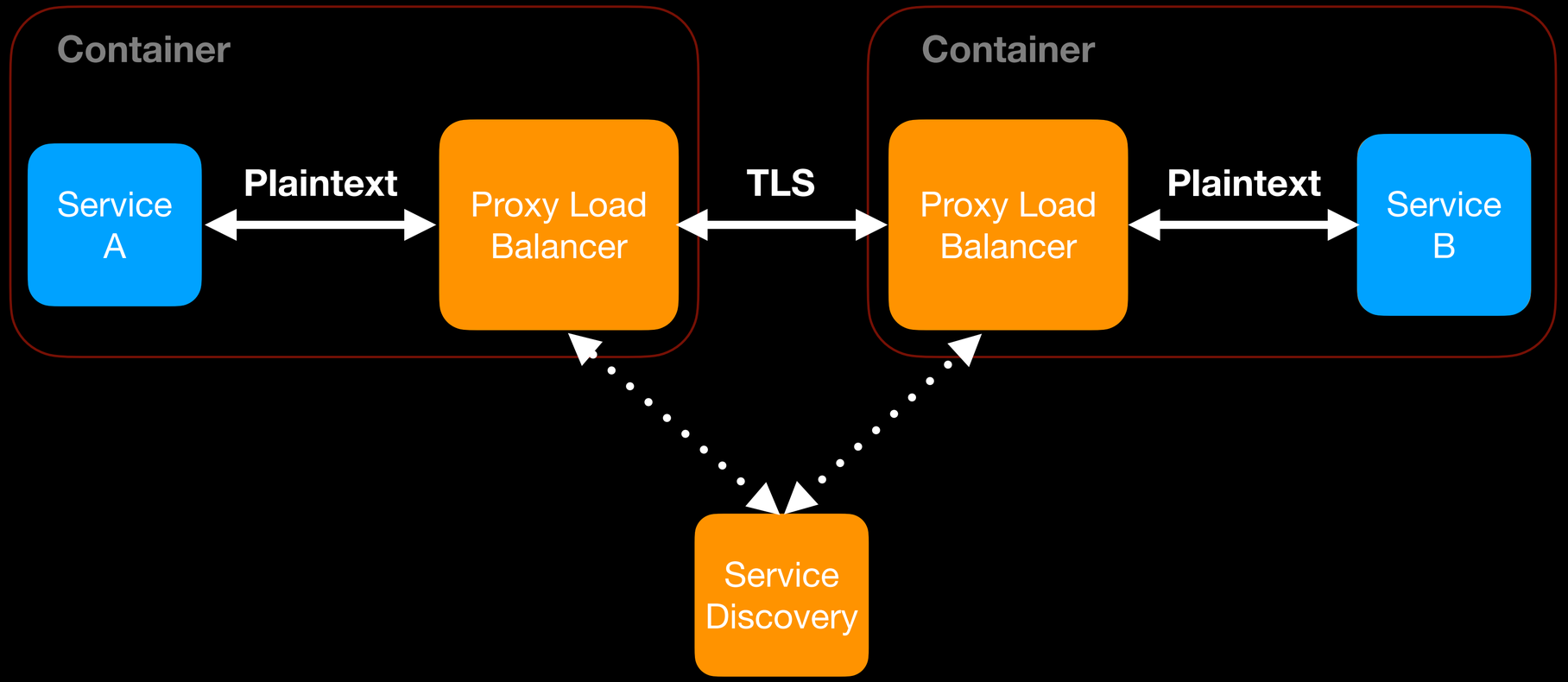


# Load balancing approach 2: Service mesh

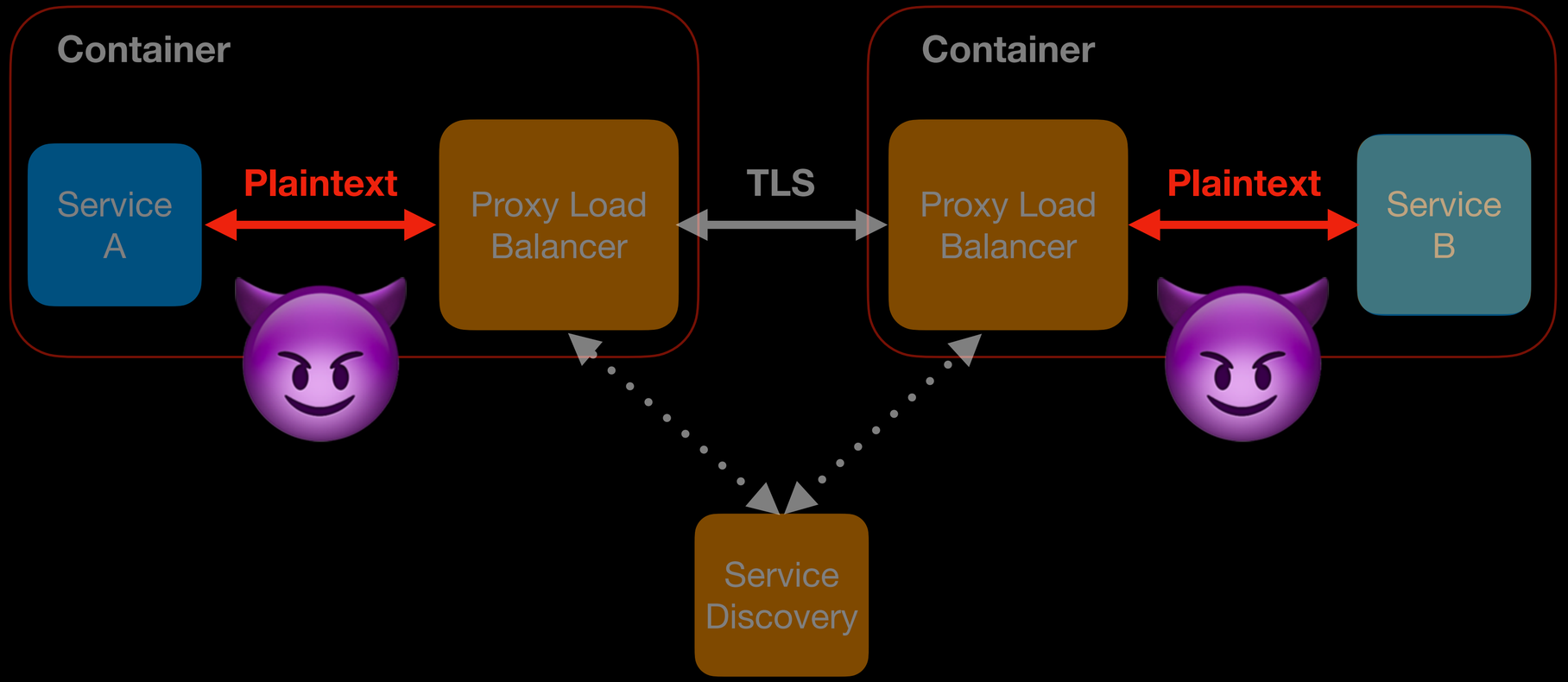



- ✗ **Distributed but more components on data path**
- ✗ **Environment / protocol limits**

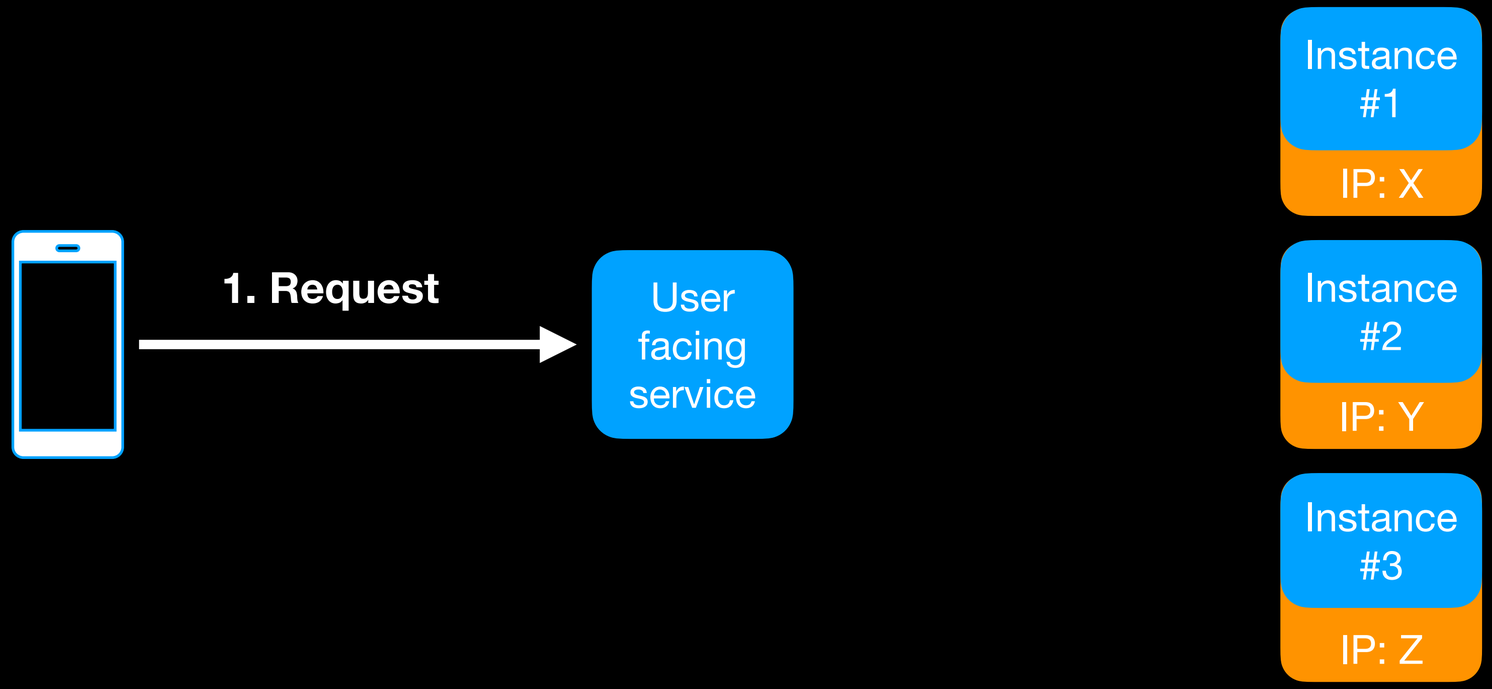
# Load balancing approach 2: Service mesh



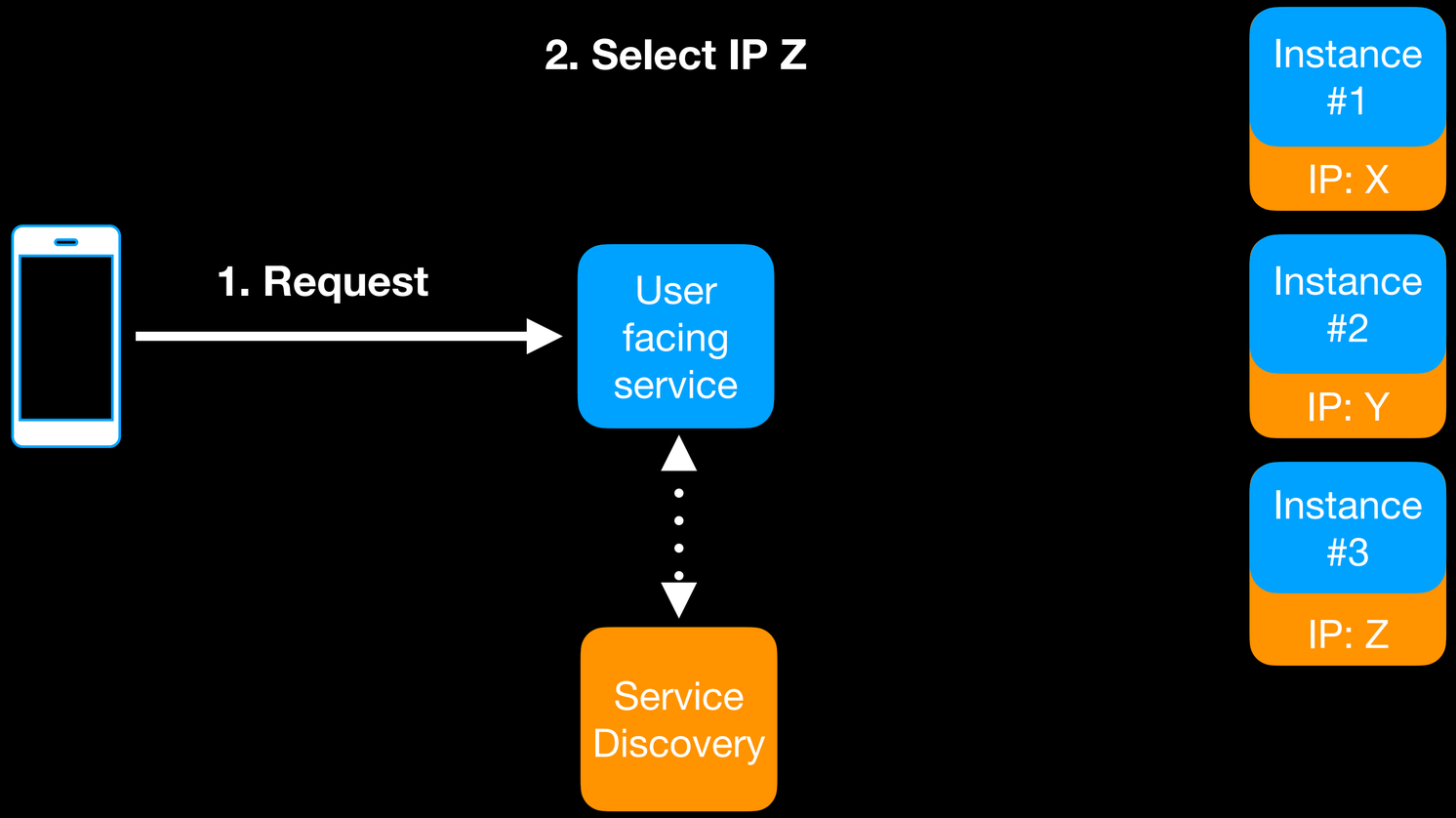
# Load balancing approach 2: Service mesh



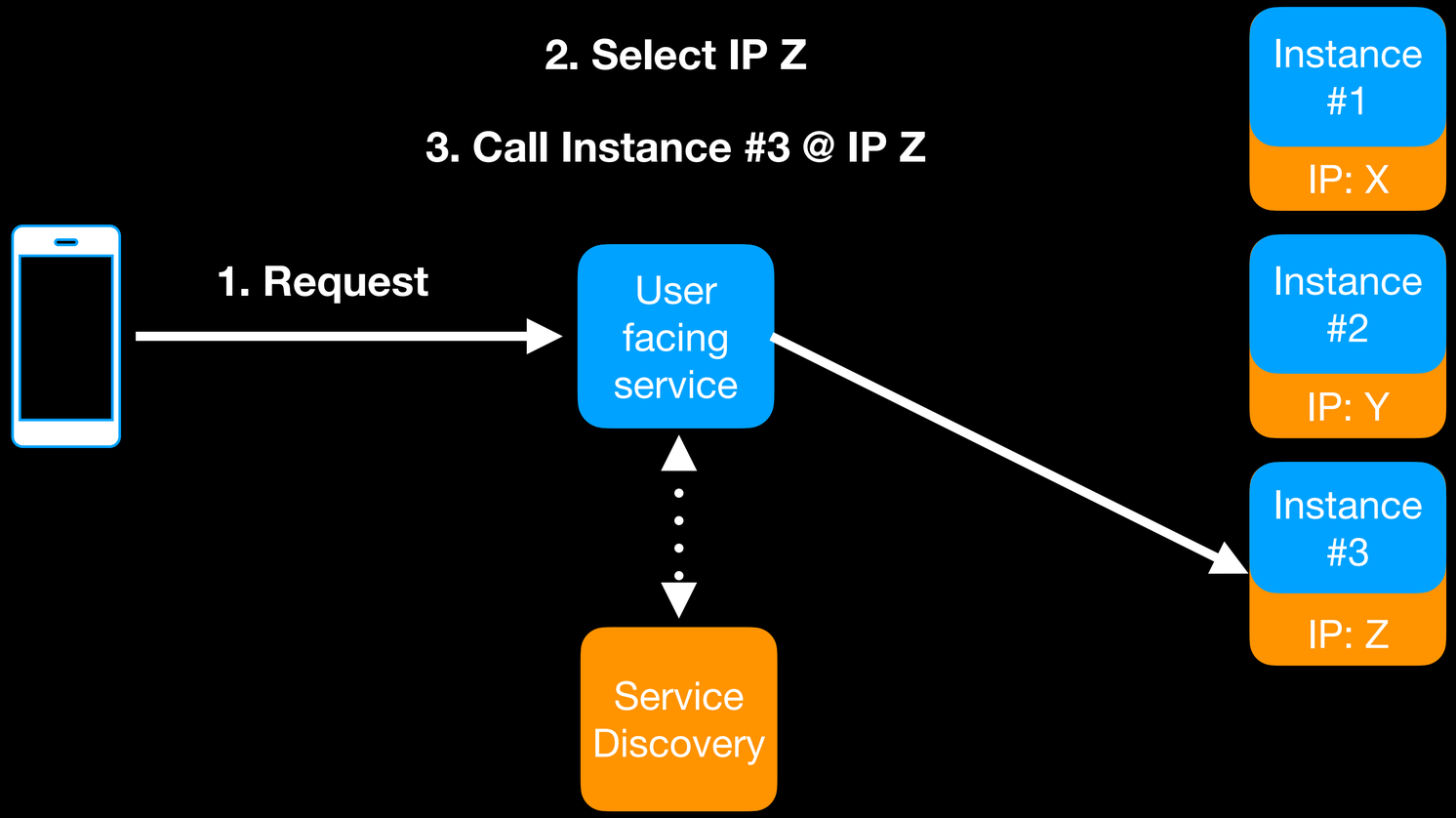
# ServiceTalk: Client-side load balancing



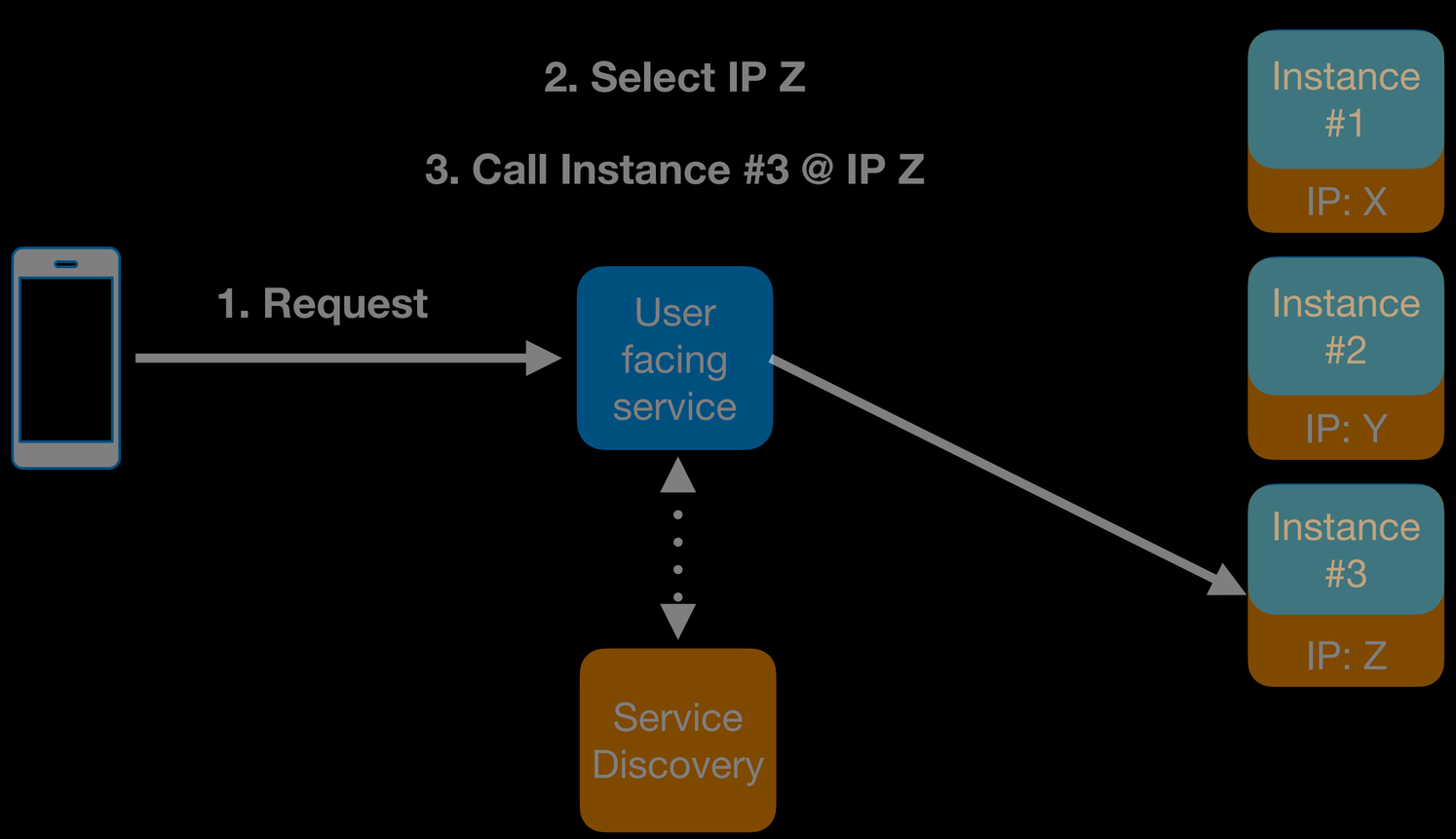
# ServiceTalk: Client-side load balancing



# ServiceTalk: Client-side load balancing



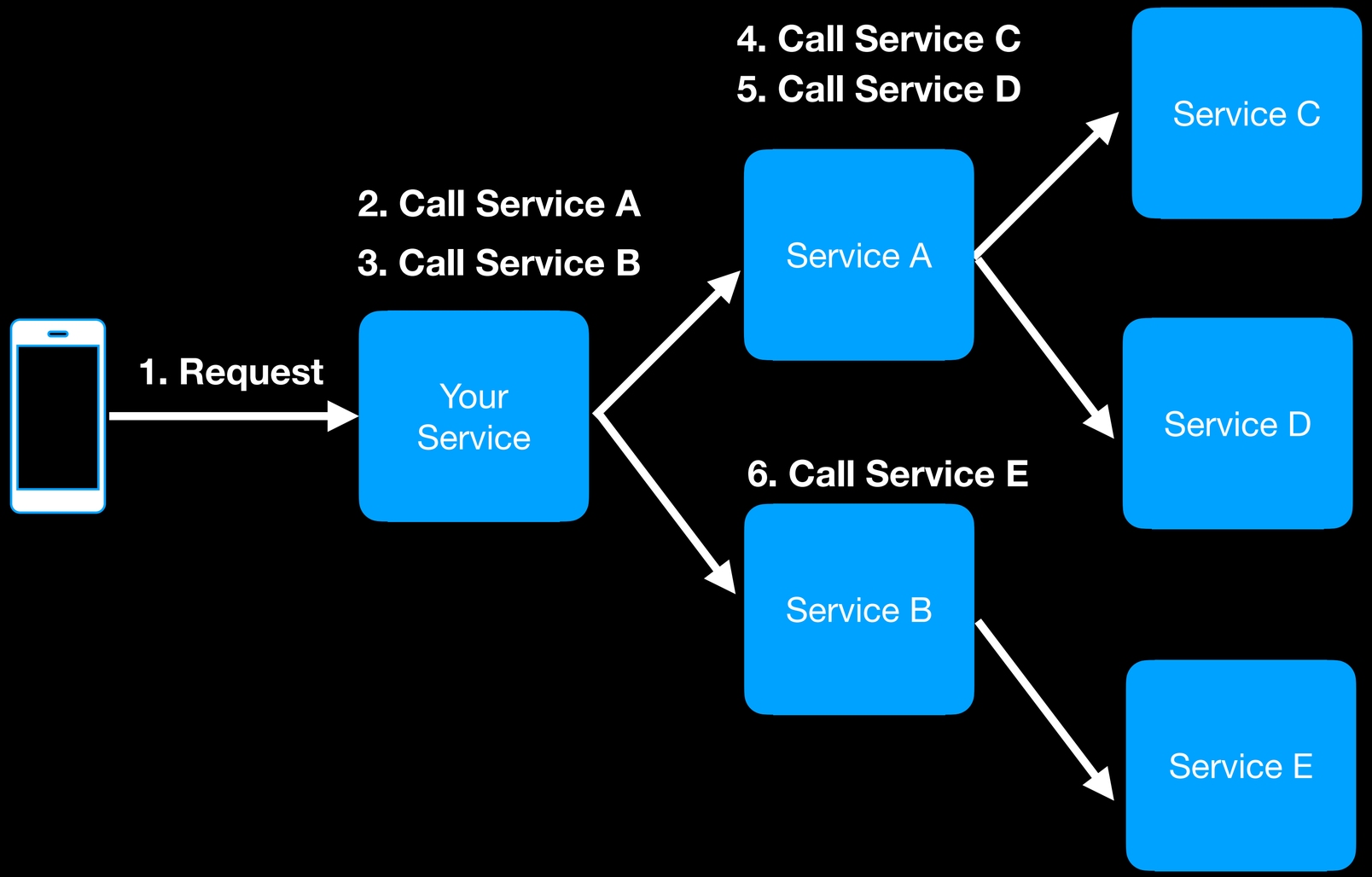
# ServiceTalk: Client-side load balancing



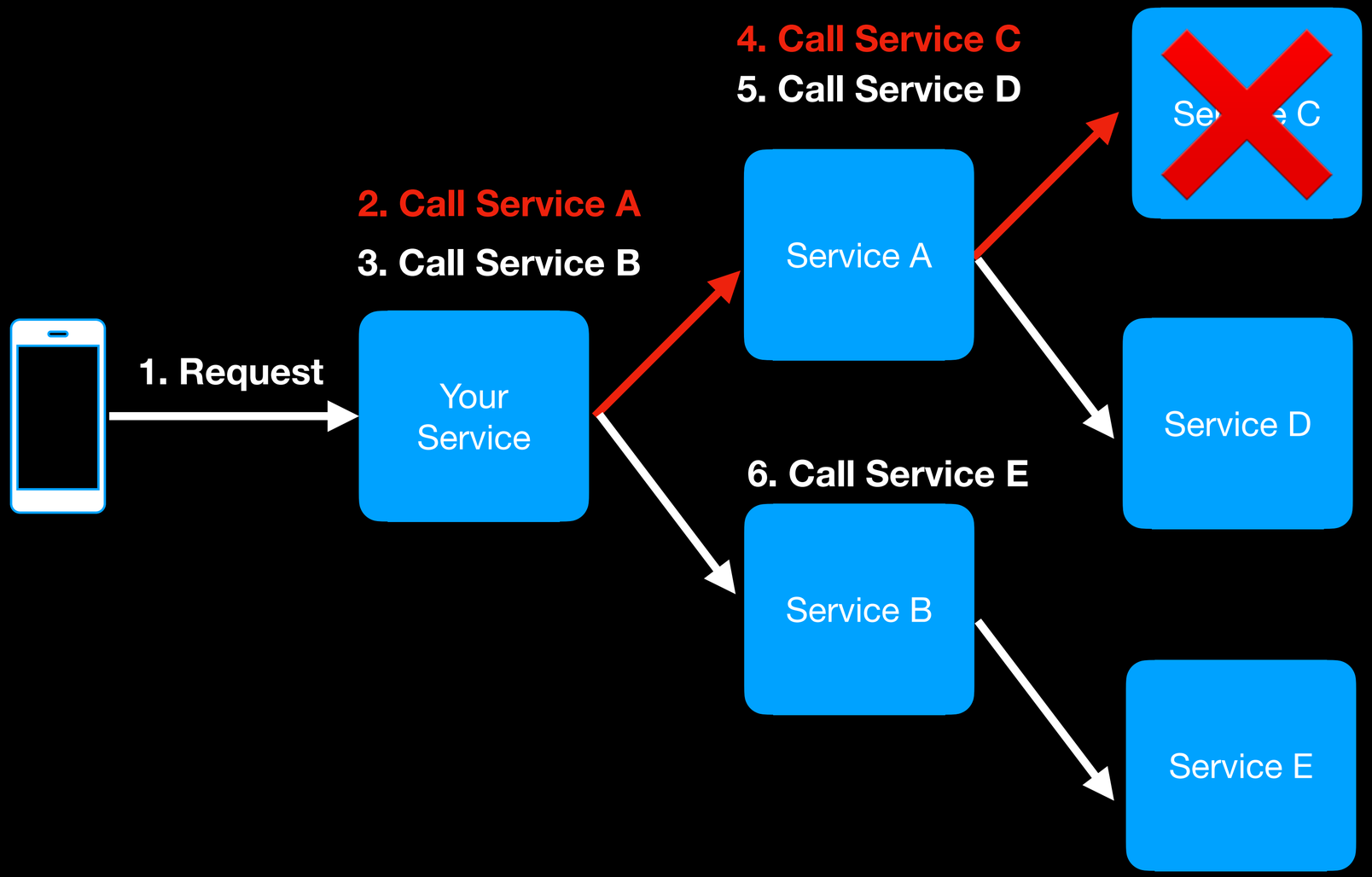

- Reduce tail latency
- No central point of failure
- Less on the data path
- Request level visibility
- Proactive Circuit Breaking
- TLS in application

# Monitoring / Observability / Debuggability

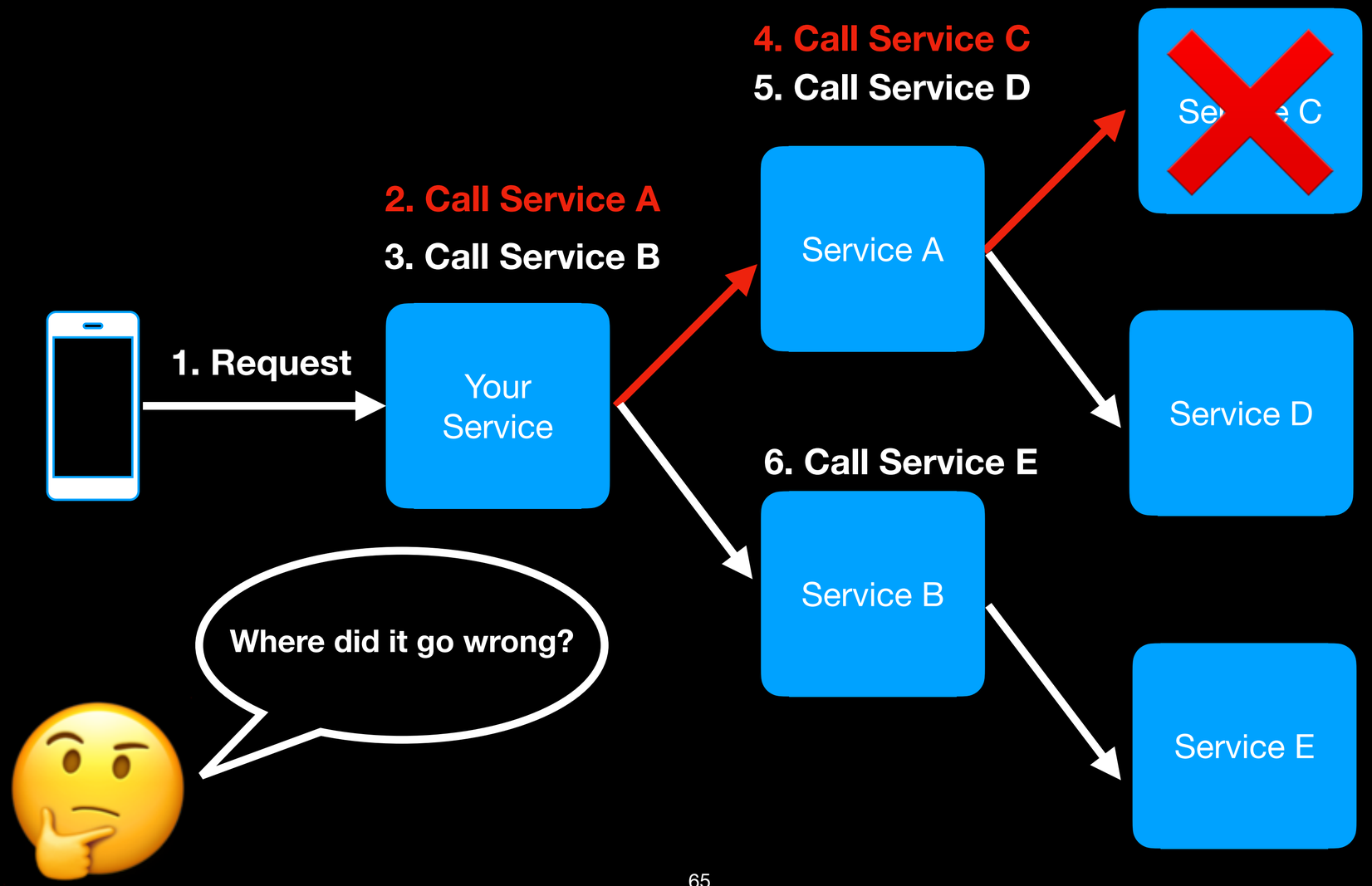
# Why Distributed Tracing?



# Why Distributed Tracing?

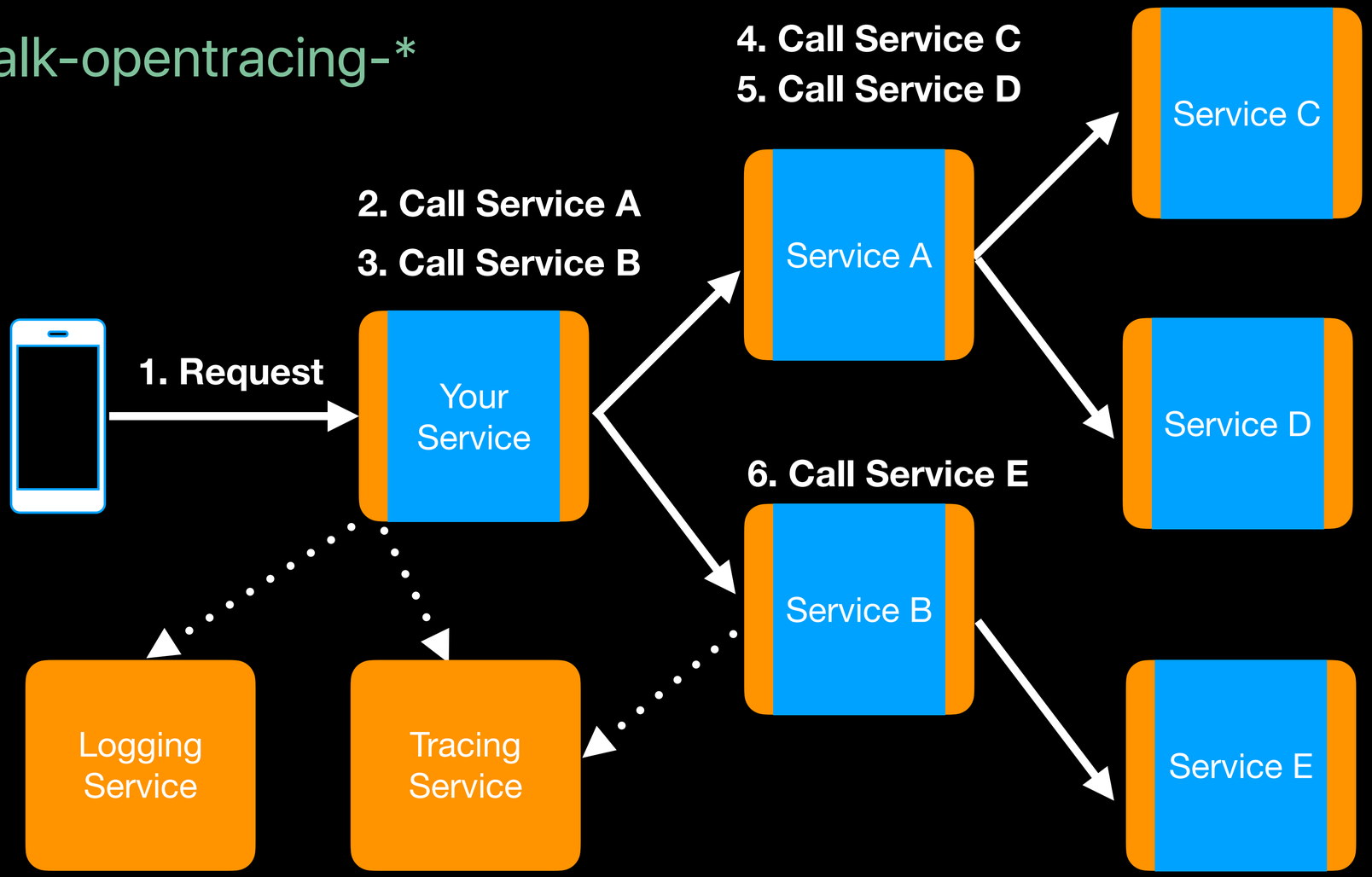


# Why Distributed Tracing?



# Distributed Tracing

servicetalk-opentracing-*

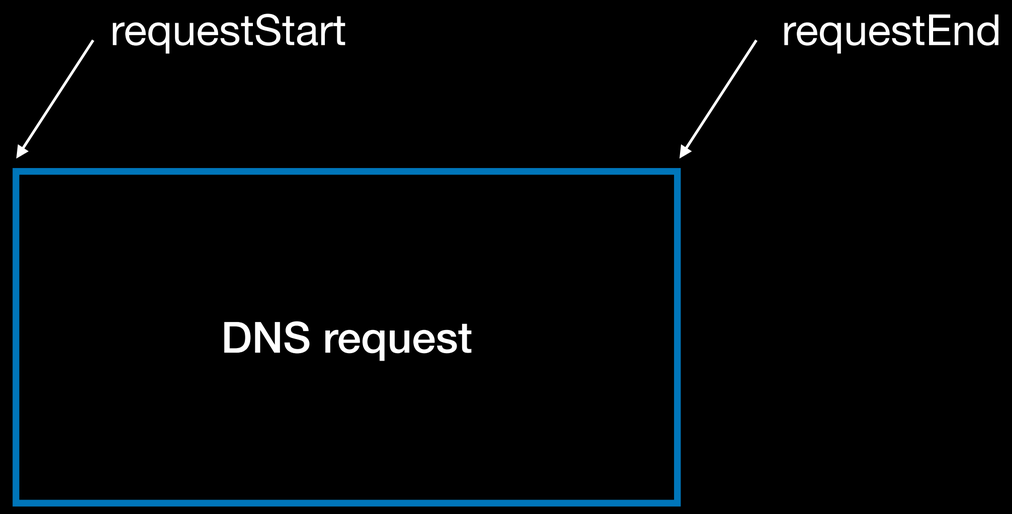


# Request is taking longer than expected?

# Or encountered an unexpected error?

# DNS ServiceDiscoverer Observer

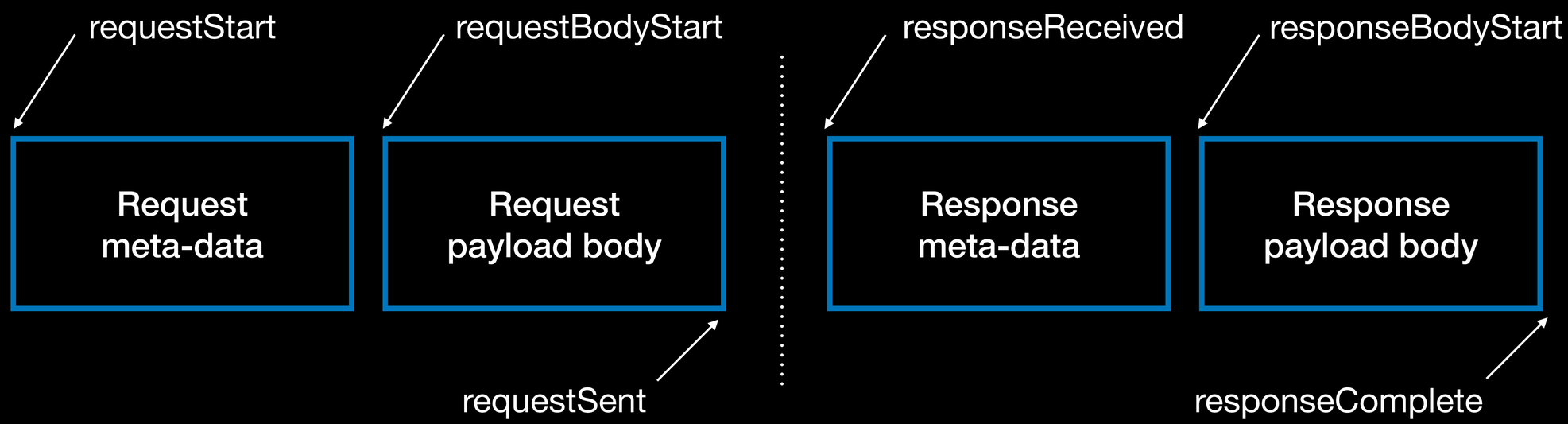
- Decoupled from request-response lifecycle
- Notifies a client when servers are discovered




- ✓ **How many resolutions do we make?**
- ✓ **Did any resolution fail?**
- ✓ **How long did it take?**
- ✓ **What was the result of the resolution?**
- ✓ **How this result affected our client?**
- ✓ **Do we update results on-time?**

# Request-response lifecycle

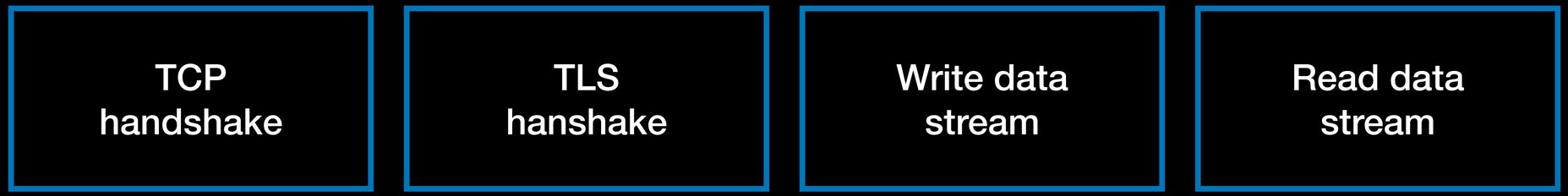
Filters enable request-response interception:




- ✓ **Latency**
- ✓ **Failures**
- ✓ **Connection reuse ratio**
- ✓ **Requests for auth**
- ✓ **Number of retries**
- ✓ **Number of redirects**


WE NEED TO GO DEEPER
DEEPERER
memegenerator.net

# L4 TransportObserver




- Total number of bytes read/written
- Flushing frequency
- SSLSession or failure cause
- Connection type and info
- Streams tracking
- Connection/stream closure reason

- Write/read boundaries
- Writability of the channel
- Flush strategy
- Thread hop time
- Read demand and availability
- Write/read operation result

https://servicetalk.io/blob/main/servicetalk-transport-api/src/main/java/io/servicetalk/transport/api/TransportObserver.java

# Understandable exceptions

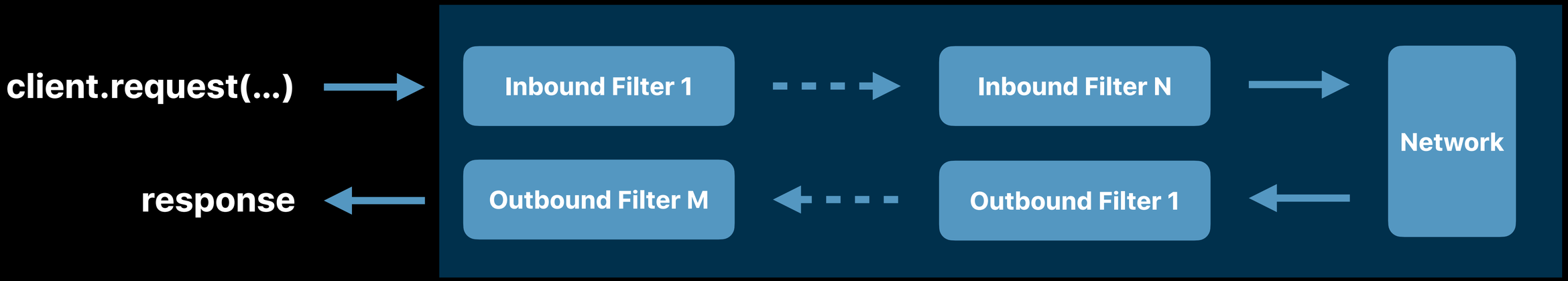
- Decoder tells exactly what was wrong with the message
- java.nio.channels.ClosedChannelException with more info:
  - which side initiated the closure?
  - was the closure expected at protocol level?
  - what event initiated the closure?
- RetryableException marker interface

# Wire logging for full transparency

```
2020-11-27 00:17:51,750 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0] REGISTERED
2020-11-27 00:17:51,753 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0] CONNECT null ->
localhost/127.0.0.1:8080
2020-11-27 00:17:51,758 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] ACTIVE
2020-11-27 00:17:51,848 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] WRITE 48B
         +-------------------------------------------------+
         |  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------+----------------+
|00000000| 47 45 54 20 2f 73 61 79 48 65 6c 6c 6f 20 48 54 |GET /sayHello HT|
|00000010| 54 50 2f 31 2e 31 0d 0a 68 6f 73 74 3a 20 6c 6f |TP/1.1..host: lo|
|00000020| 63 61 6c 68 6f 73 74 3a 38 30 38 30 0d 0a 0d 0a |calhost:8080....|
+--------+-------------------------------------------------+----------------+
2020-11-27 00:17:51,849 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] WRITE 0B
2020-11-27 00:17:51,850 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] USER_EVENT io.servicetalk.transport.netty.internal.CloseHandler$OutboundDataEndEvent
2020-11-27 00:17:51,852 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] WRITE 0B
2020-11-27 00:17:51,853 servicetalk-global-io-executor-1-2 [INFO ] client - [id: 0xfa1e8ae0, L:/127.0.0.1:65530 -
R:localhost/127.0.0.1:8080] FLUSH
```

# Additional features

# Extensibility using filtering




- Logging and metrics
- Authentication
- Retries and error handling
- Redirects
- Concurrency control
- Payload transformations

# Pluggable routing

JAX-RS router

```
ServerContext serverContext =
        HttpServers.forPort(8080)
    .listenStreamingAndAwait(
        new HttpJerseyRouterBuilder()
        .buildStreaming(new JaxRsApplication()));
```

```
@Path("/greetings")
public class JaxRsResource {
    @GET
    @Path("/hello")
    @Produces(TEXT_PLAIN)
    public String hello(@DefaultValue("world")
                        @QueryParam("who")
                        final String who) {
        return "Hello " + who;
    }
}
```

servicetalk-http-router-jersey

See docs

# Pluggable routing

JAX-RS router

```
ServerContext serverContext =
        HttpServers.forPort(8080)
    .listenStreamingAndAwait(
        new HttpJerseyRouterBuilder()
        .buildStreaming(new JaxRsApplication()));
```

servicetalk-http-router-jersey

Predicate router

```
ServerContext serverContext =
        HttpServers.forPort(8080)
    .listenStreamingAndAwait(
            new HttpPredicateRouterBuilder()
        .whenPathEquals("/route1")
            .thenRouteTo(service1)
        .whenPathStartsWith("/route2")
            .thenRouteTo(service2)
        .buildStreaming());
```

servicetalk-http-router-predicate

# Automatic recovery from failures

- Auto-retry recoverable failures
  - When if’s safe to do so
  - When users configure additional conditions

```
AutoRetryStrategyProvider

RetryingHttpRequesterFilter

RetryableException
```

- Load Balancer is capable to do circuit breaking

# Context propagation

```
AsyncContext.put(key, value);
AsyncContext.get(key);
```

See docs

servicetalk-log4j2-mdc

servicetalk-opentracing-asynccontext

# Netty API is hidden

- ServiceTalk does not expose Netty API to the users
- Provides tuned, safer, and more secure default Netty settings
- It took Twitter years to upgrade Finagle from Netty 3 to Netty 4
- Netty 5 is coming

# Microservice evolution with ServiceTalk

# 1. Start simple with what you know

```
HttpServers.forPort(8080)
    .listenStreamingAndAwait(
        new HttpJerseyRouterBuilder()
        .buildStreaming(new JaxRsApplication()))
    .awaitShutdown();
```

```
public class JaxRsApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return singleton(JaxRsResource.class);
    }
}
```

```
@Path("/api")
public class JaxRsResource {

    @GET
    @Path("/user")
    @Produces(APPLICATION_JSON)
    public User user(@QueryParam("id")
            String id) {
        return findUser(id);
    }

    @GET
    @Path("/users")
    @Produces(APPLICATION_JSON)
    public List<User> users(@QueryParam("loc")
            String loc) {
        return findUsers(loc);
    }
}
```

# 2. Learn Reactive Streams safely

```
HttpServers.forPort(8080)
    .listenStreamingAndAwait(
        new HttpJerseyRouterBuilder()
        .buildStreaming(new JaxRsApplication()))
    .awaitShutdown();
```

```
public class JaxRsApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return singleton(JaxRsResource.class);
    }
}
```

**Learning new programming style**

```
@Path("/api")
public class JaxRsResource {

    @GET
    @Path("/user")
    @Produces(APPLICATION_JSON)
    public User user(@QueryParam("id")
            String id) {
        return findUser(id);
    }

    @GET
    @Path("/users")
    @Produces(APPLICATION_JSON)
    public Publisher<User> users(@QueryParam("loc")
        String loc) {
    return findUserIds(loc).map(this::findUser);
    }
}
```

# 3. Make a route fully asynchronous

```
HttpServers.forPort(8080)
    .executionStrategy(noOffloadsStrategy())
    .listenStreamingAndAwait(
        new HttpJerseyRouterBuilder()
        .buildStreaming(new JaxRsApplication()))
    .awaitShutdown();
;
```

```
public class JaxRsApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return singleton(JaxRsResource.class);
    }
}
```

**Significantly reduced latency**

```
@Path("/api")
public class JaxRsResource {

    @GET
    @Path("/user")
    @Produces(APPLICATION_JSON)
    public User user(@QueryParam("id")
            String id) {
        return findUser(id);
    }

    @GET
    @Path("/users")
    @Produces(APPLICATION_JSON)
    @NoOffloadsRouteExecutionStrategy
    public Publisher<User> users(@QueryParam("loc")
        String loc) {
    return findUserIds(loc)
            .flatMapMergeSingle(
                    userService::findUserAsync);
    }
}
```

# 4. Avoid Jersey overhead for a critical route

```
HttpServers.forPort(8080)
    .executionStrategy(noOffloadsStrategy())
    .listenStreamingAndAwait(new HttpPredicateRouterBuilder()
        .whenPathEquals("/api/users")
            .executionStrategy(noOffloadsStrategy())
            .thenRouteTo(new UsersStreamingService())
        .whenPathStartsWith("/api")
            .thenRouteTo(new HttpJerseyRouterBuilder()
                .buildStreaming(new JaxRsApplication()))
        .buildStreaming())
```

```
public class JaxRsApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        return singleton(JaxRsResource.class);
    }
}
```

**Avoid expensive routing: reflection, OIO, etc.**

```
@Path("/api")
public class JaxRsResource {

    @GET
    @Path("/user")
    @Produces(APPLICATION_JSON)
    public User user(@QueryParam("id")
                     String id) {
        return findUser(id);
    }
```

```
public final class UsersStreamingService
        implements StreamingHttpService {

    @Override
    public Single<StreamingHttpResponse> handle(
            HttpServiceContext ctx,
            StreamingHttpRequest request,
            StreamingHttpResponseFactory responseFactory) {
        return succeeded(responseFactory.ok()
                .payloadBody(findUserIds(
                    request.queryParameter("loc"))
                .flatMapMergeSingle(userService::findUserAsync),
                    serializer.serializerFor(User.class)));
    }
}
```

# 5. Avoid Jersey for all routes

```
HttpServers.forPort(8080)
    .executionStrategy(noOffloadsStrategy())
    .listenStreamingAndAwait(new HttpPredicateRouterBuilder()
        .whenPathEquals("/api/users")
            .executionStrategy(noOffloadsStrategy())
            .thenRouteTo(new UsersStreamingService())
        .whenPathEquals("/api/user")
            .thenRouteTo(new UserService())
        .buildStreaming())
    .awaitShutdown();
```

**Avoid Jersey at all, reduce dependencies**

```
public static final class UserService
        implements BlockingHttpService {

    @Override
    public HttpResponse handle(HttpServiceContext ctx,
            HttpRequest request,
            HttpResponseFactory responseFactory) {
        return responseFactory.ok()
                .payloadBody(findUser(
                request.queryParameter("id")),
                serializer.serializerFor(User.class));
    }
}

public final class UsersStreamingService
        implements StreamingHttpService {

    @Override
    public Single<StreamingHttpResponse> handle(
            HttpServiceContext ctx,
            StreamingHttpRequest request,
            StreamingHttpResponseFactory responseFactory) {
        return succeeded(responseFactory.ok()
                .payloadBody(findUserIds(
                    request.queryParameter("loc"))
                .flatMapMergeSingle(userService::findUserAsync),
                    serializer.serializerFor(User.class)));
    }
}
```

# 6. Convert blocking route to async

```
HttpServers.forPort(8080)
    .executionStrategy(noOffloadsStrategy())
    .listenStreamingAndAwait(new HttpPredicateRouterBuilder()
        .whenPathEquals("/api/users")
            .executionStrategy(noOffloadsStrategy())
            .thenRouteTo(new UsersStreamingService())
        .whenPathEquals("/api/user")
            .executionStrategy(noOffloadsStrategy())
            .thenRouteTo(new UserService())
        .buildStreaming())
```

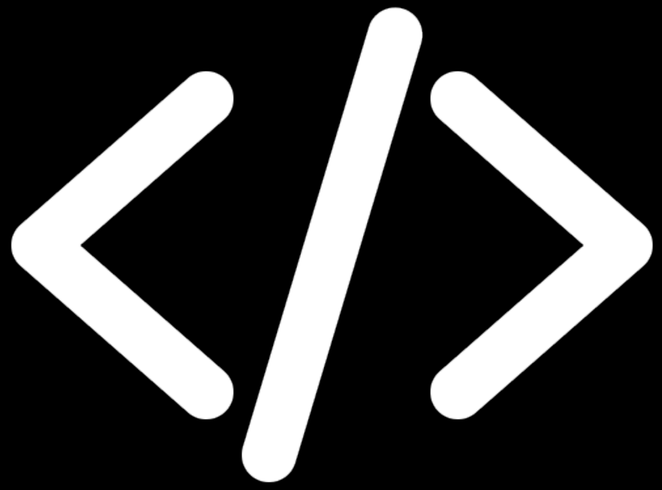
**Evolve to fully asynchronous execution**

```
public static final class UserService
        implements HttpService {

    @Override
    public Single<HttpResponse> handle(HttpServiceContext ctx,
            HttpRequest request,
            HttpResponseFactory responseFactory) {
        return userService.findUserAsync(
                request.queryParameter("id"))
                .map(user -> responseFactory.ok()
                    .payloadBody(user,
                        serializer.serializerFor(User.class)));
    }
}
public final class UsersStreamingService
        implements StreamingHttpService {

    @Override
    public Single<StreamingHttpResponse> handle(
            HttpServiceContext ctx,
            StreamingHttpRequest request,
            StreamingHttpResponseFactory responseFactory) {
        return succeeded(responseFactory.ok()
                .payloadBody(findUserIds(
                    request.queryParameter("loc"))
                .flatMapMergeSingle(userService::findUserAsync),
                    serializer.serializerFor(User.class)));
    }
}
```

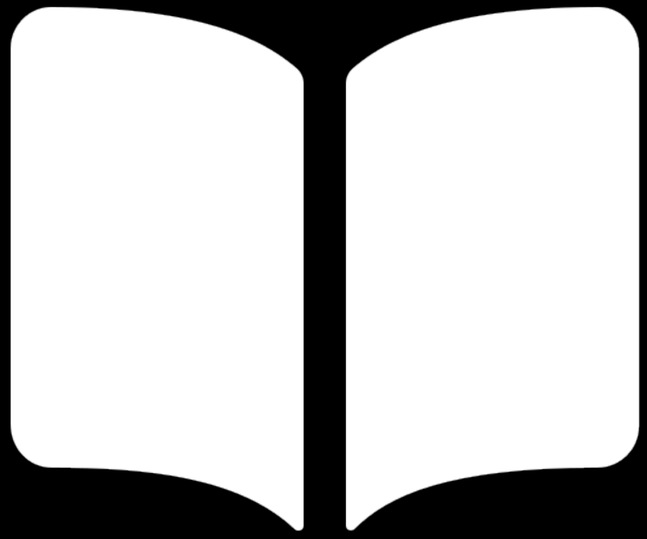
# 7. Continue evolving as you go

- Add filters for logging, metrics, auth, tracing
- Configure ConnectionAcceptor
- Configure ConcurrencyController
- Enable TransportObserver
- Tune performance
- Evolve on the client-side: error handling, ServiceDiscoverer, LoadBalancer, etc.

# How to get started



https://servicetalk.io

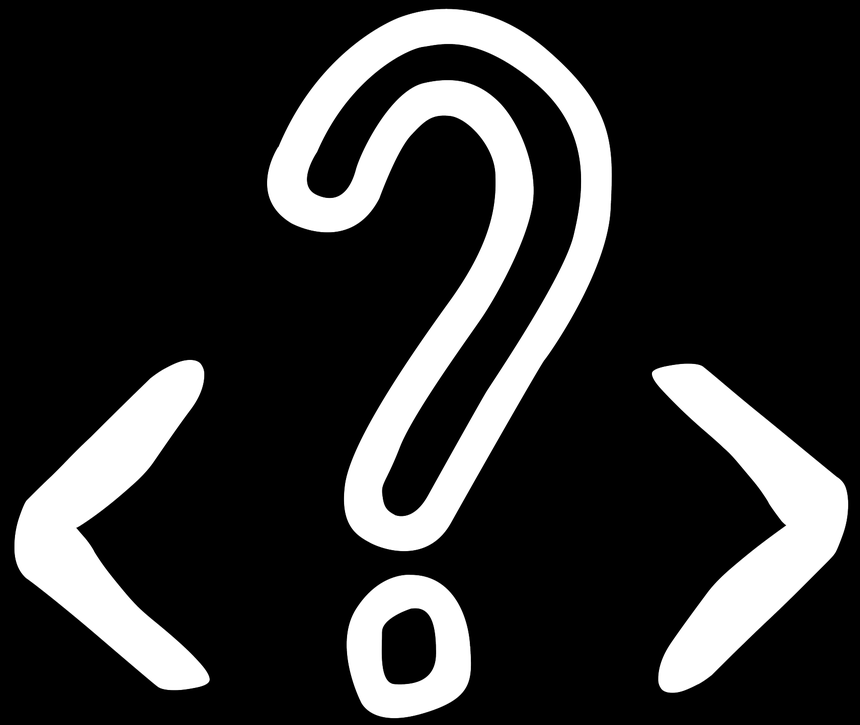


https://docs.servicetalk.io

http://ionicons.com

# Show me something real

https://servicetalk.io/tree/main/servicetalk-examples



https://stackoverflow.com/questions/tagged/servicetalk

https://jokerconf.com/assets/images/logo_white.svg

# ServiceTalk

# contributions

Start a discussion
Report a bug
Look for "good first issue" tag in the tracker

# Conclusion

- Easy to start with API you already know
- Highly extensible framework on top of Netty
- Reactive core, but safe for typical blocking Java code
- Evolves with your application
  - Transition between APIs and protocols
  - On-demand optimizations
- Transparent execution (monitoring, observability, debuggability)

Joker<?>

# Microservices design philosophy with ServiceTalk.io

Idel Pivnitskiy
November 27, 2020

@idelpivnitskiy
idel.pivnitskiy@gmail.com