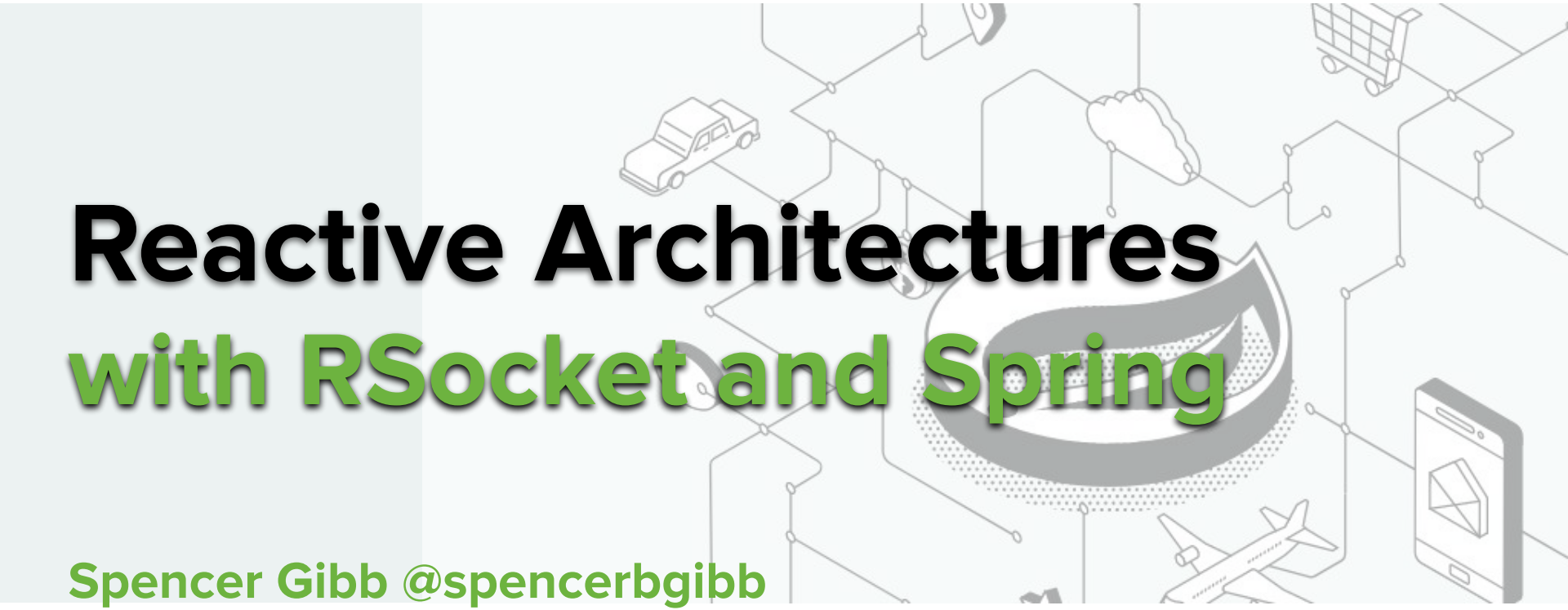
Reactive Architectures
with RSocket and Spring

Spencer Gibb @spencerbgibb
Cora Iberkleid @ciberkleid



Safe Harbor Statement

The following is intended to outline the general direction of VMware's offerings. It is intended for information purposes only and may not be incorporated into any contract. Any information regarding pre-release of VMware offerings, future updates or other planned modifications is subject to ongoing evaluation by VMware and is subject to change. This information is provided without warranty or any kind, express or implied, and is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions regarding VMware's offerings. These purchasing decisions should only be based on features currently available. The development, release, and timing of any features or functionality described for VMware's offerings in this presentation remain at the sole discretion of VMware. VMware has no obligation to update forward looking information in this presentation.

Agenda

- Reactive Architectures
- RSocket
- RSocket Routing Broker
  - Client
  - Broker
  - Anatomy of a Request
- Demo(s)

Reactive Architectures

Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

http://www.reactive-streams.org

Reactive Java Building Blocks

- Reactive Streams
  - **Standard** for async stream processing with non-blocking back pressure
  - **Publisher/Subscriber/Subscription/Processor**
- Project Reactor
  - **Implementation** of the Reactive Streams specification for the JVM
    - Adds **Flux** and **Mono** operators
    - Java 8 integration (Stream, CompletableFuture, Duration)

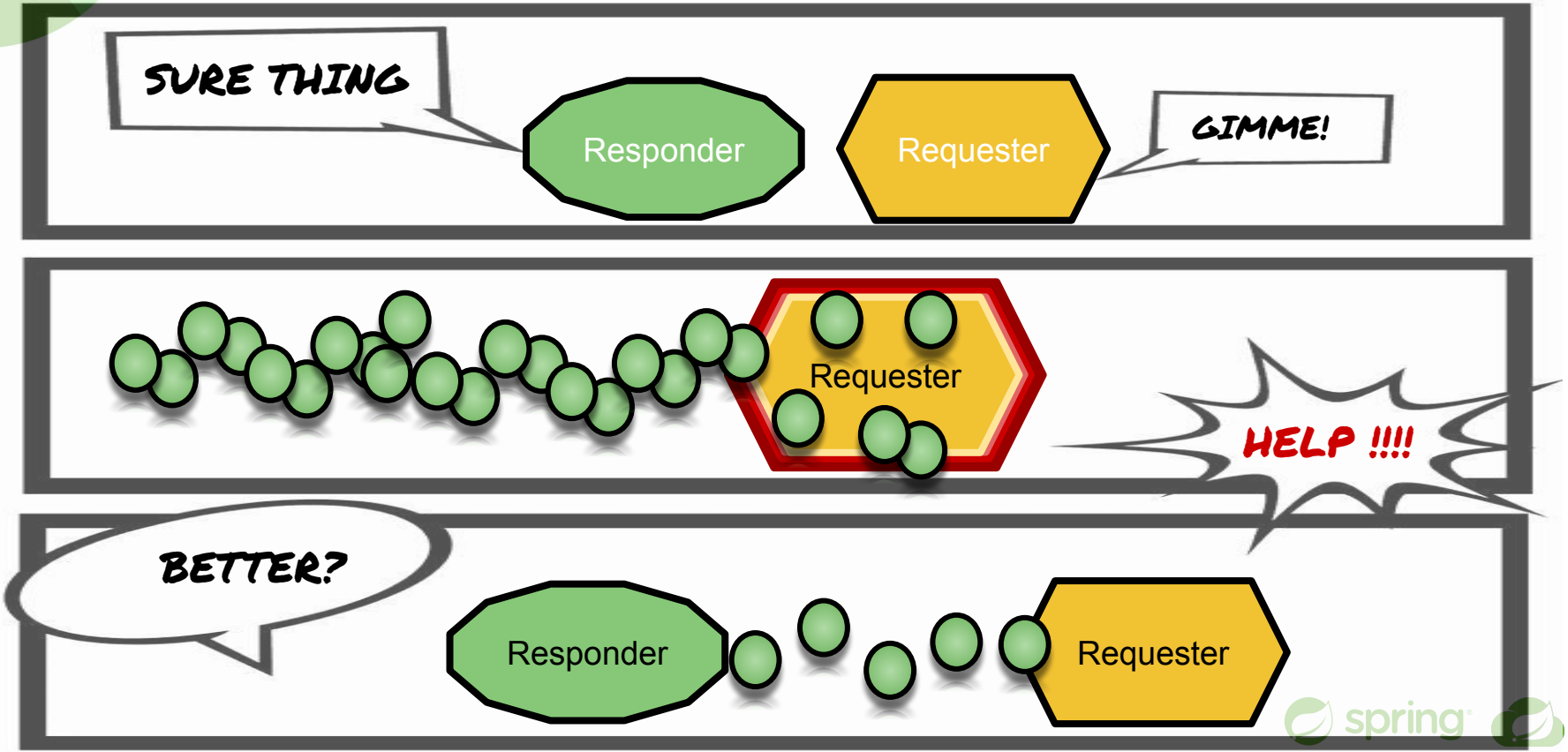
Roadblocks

- But there are still some barriers to using Reactive everywhere*

- Data Access
  - MongoDB, Apache Cassandra, and Redis
  - Relational database access (R2DBC)

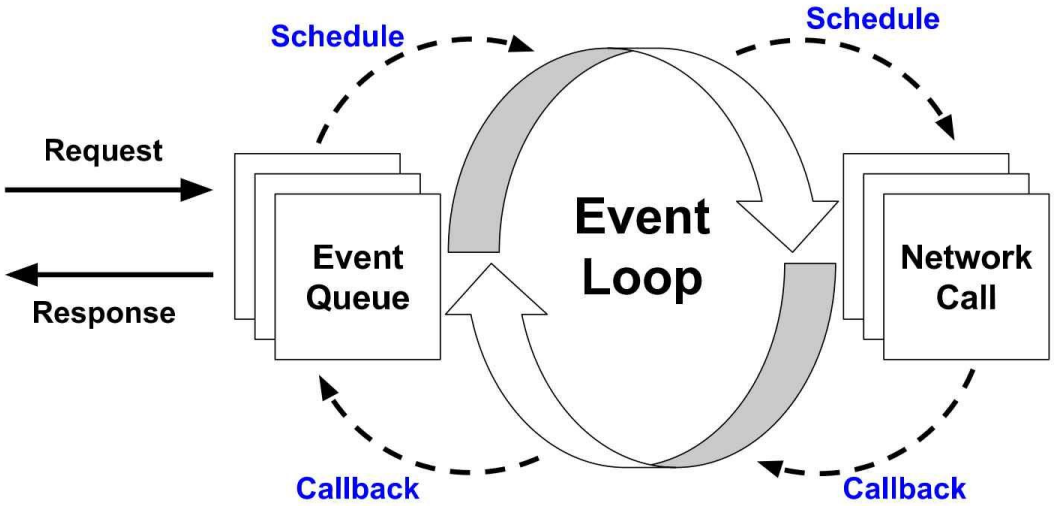
- Cross-process back pressure (networking)

Back Pressure



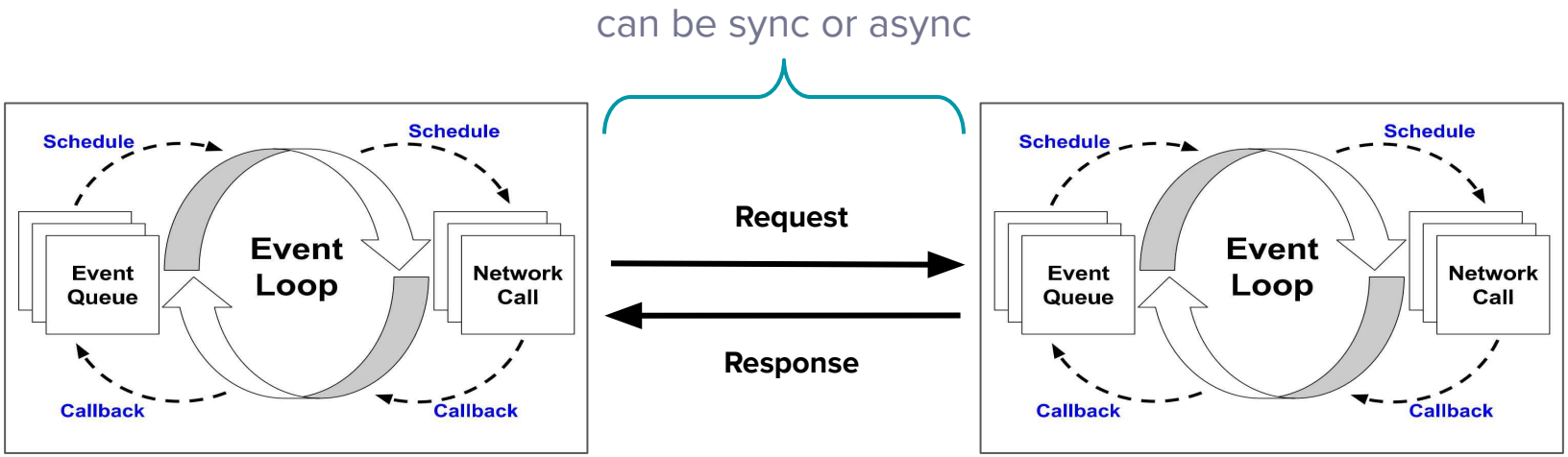
Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.



Reactive Inter-process Communication

- Reactive has no opinion on synchronous vs asynchronous
- Key differentiator is **back pressure** (Reactive pull/push)



http://rsocket.io

RSocket

RSocket is a bi-directional, multiplexed, message-based, binary protocol based on Reactive Streams back pressure

It provides out of the box support for four common interaction models

- Request-Response (1 to 1)
- Fire-and-Forget (1 to 0)
- Request-Stream (1 to many)
- Request-Channel (many to many)

Transport Agnostic: TCP, WebSocket, UDP, HTTP2 ...

RSocket vs HTTP - Key Differences

RSocket
Efficient and Responsive

- Single, shared long-lived connection
- Multiplexes messages
- Communicates back pressure
- Either party can initiate requests (flexible requester/responder roles)
- Supports canceling/resuming streams

HTTP
Slowly Improving

- New connection per request (HTTP 1.0)
- Pipelines messages (HTTP 1.1)
- Does not communicate back pressure
- Only client can initiate requests (fixed client/server roles)
- Does not support canceling/resuming streams

RSocket vs HTTP - Key Differences

RSocket Efficient and Responsive	HTTP Slowly Improving
• Single, shared long-lived connection	• New connection per request (HTTP 1.0)
• Multiplexes messages	• Pipelines messages (HTTP 1.1)
• Communicates back pressure	• Does not communicate back pressure
• Either party can initiate requests (flexible requester/responder roles)	• Only client can initiate requests (fixed client/server roles)
• Supports canceling/resuming streams	• Does not support canceling/resuming streams

RSocket vs HTTP - Key Differences

RSocket Efficient and Responsive	HTTP Slowly Improving
• Single, shared long-lived connection	• New connection per request (HTTP 1.0)
• Multiplexes messages	• Pipelines messages (HTTP 1.1)
• Communicates back pressure	• Does not communicate back pressure
• Either party can initiate requests (flexible requester/responder roles)	• Only client can initiate requests (fixed client/server roles)
• Supports canceling/resuming streams	• Does not support canceling/resuming streams

RSocket vs HTTP - Key Differences

RSocket Efficient and Responsive	HTTP Slowly Improving
• Single, shared long-lived connection	• New connection per request (HTTP 1.0)
• Multiplexes messages	• Pipelines messages (HTTP 1.1)
• Communicates back pressure	• Does not communicate back pressure
• Either party can initiate requests (flexible requester/responder roles)	• Only client can initiate requests (fixed client/server roles)
• Supports canceling/resuming streams	• Does not support canceling/resuming streams

RSocket vs HTTP - Key Differences

RSocket Efficient and Responsive	HTTP Slowly Improving
• Single, shared long-lived connection	• New connection per request (HTTP 1.0)
• Multiplexes messages	• Pipelines messages (HTTP 1.1)
• Communicates back pressure	• Does not communicate back pressure
• Either party can initiate requests (flexible requester/responder roles)	• Only client can initiate requests (fixed client/server roles)
• Supports canceling/resuming streams	• Does not support canceling/resuming streams

RSocket Routing

Reactive Building Blocks

RSocket Routing Broker

RSocket Routing Client

Spring Boot

Spring Framework
(depends on Project Reactor and RSocket Java)

RSocket Java

RSocket Javascript

RSocket Go

RSocket .NET

RSocket C++

RSocket Kotlin

Project Reactor
(Flux, Mono)

Java Reactive Streams
(Publisher, Subscriber, Subscription, Processor interfaces)

RSocket (specification)

Reactive Streams (specification)

spring

Vocabulary

- **Broker**: forwards RSocket requests to routable destinations
- **Routing**: mechanism for creating, and sharing routing tables
- **Routable Destination (Route)**: where requests can be forwarded
- **Address**: forwarding of a request to a routing destination
- **Tag**: a key/value pair which determines where to route data and for lookup in routing tables
- **Metadata**: a key/value pair used to represent metadata
- **Origin**: where a request starts

Roadmap

RSocket Routing Broker and Client are new projects under the Reactive Foundation. Initial contributions are from VMware and Netifi.

The RSocket Routing and Forwarding extension specification has not been finalized.

This work includes the Java implementation of the specification.

RSocket Routing Client

RSocket Routing Client

Provides utilities to construct the proper metadata to register with the Broker and send requests for any JVM language.

RSocket Routing Client

No incoming connections need be allowed.

RSocket Routing Client Spring

Provides autoconfiguration to auto connect to broker.

RSocket Routing Client

Clients send ROUTE_SETUP

RSocket Routing Client Spring

Provides Configuration Properties for broker connection and route setup

RSocket Routing Client Spring

Configures Spring Framework Messaging `RSocketRequester`.

RSocket Routing Client Spring

When making requests, automatically adds Address metadata via configuration properties.

RSocket Routing Broker

RSocket Routing Broker

Accepts connections from clients and other brokers.

RSocket Routing Broker

When clients connect, entries are made in routing table. Indexes are made with Roaring Bitmaps for fast lookup.

http://roaringbitmap.org

RSocket Routing Broker

Sends BROKER_INFO before remote brokers can forward to local connections

RSocket Routing Broker

Clients can be written in any language, they just need the correct metadata

RSocket Routing Broker

Local brokers forward ROUTE_SETUP to other brokers via ROUTE_ADD

RSocket Routing Broker

Broker's communicate using Spring Framework's RSocket `@MessageMapping` and `RSocketRequester` infrastructure.

RSocket Routing Broker

Can listen on any RSocket supported transport.

RSocket Routing Broker

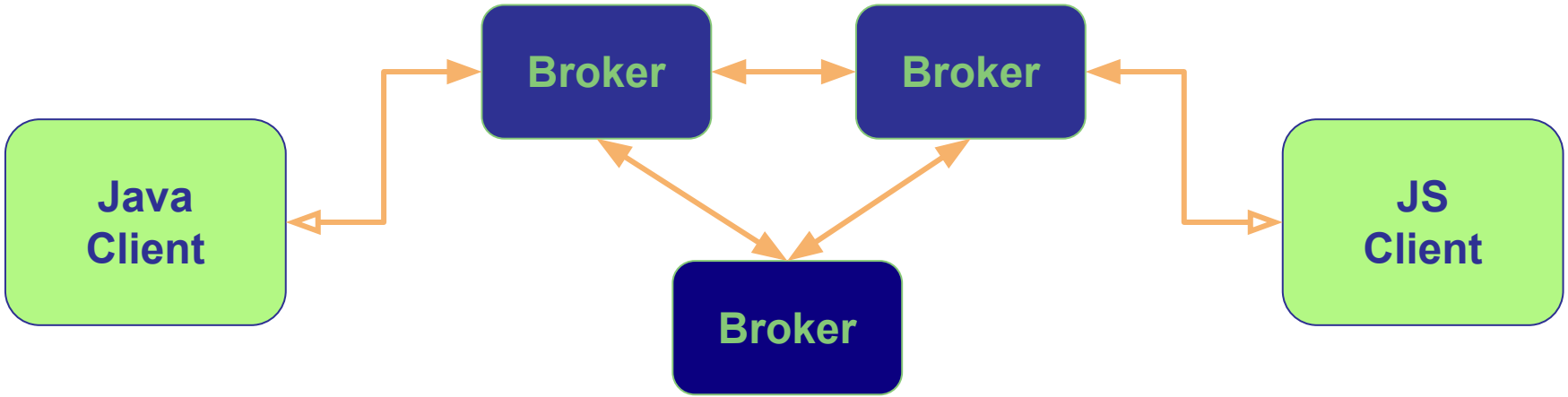
Security provided by Spring Security RSocket support

RSocket Routing Broker

Broker requires **Micrometer** for multiple levels of metrics: general, connection-specific, request specific.

Anatomy of a Request

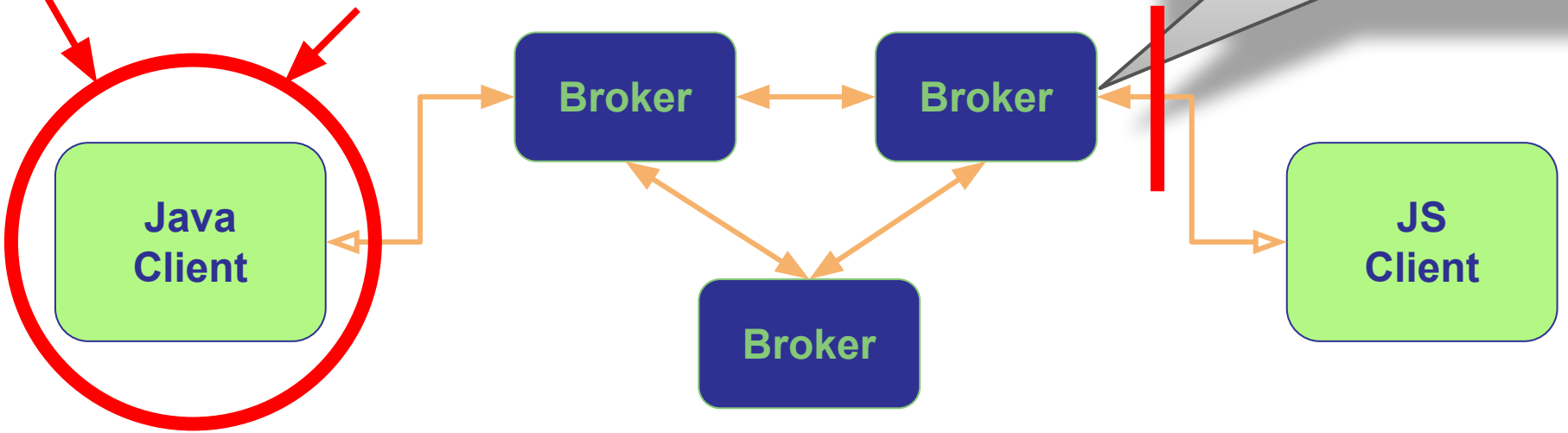
RSocket Routing Architecture



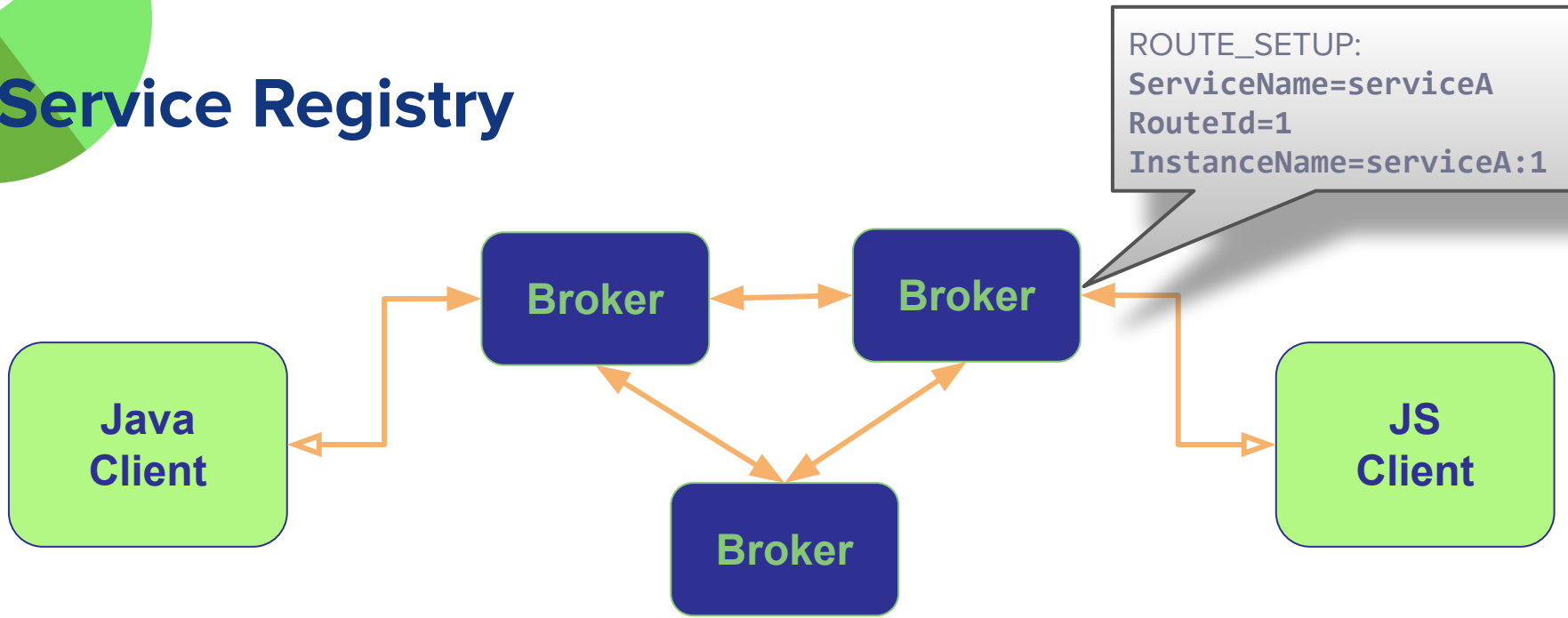
Connecting to RSocket Routing Broker

- Client makes connection to Broker Cluster
- Sends `ROUTE_SETUP` metadata on connect
  - Who am I?
    - `routeId`
    - `ServiceName`
    - other tags such as: `Region`, `InstanceName`, or `Version`

RSocket Routing Architecture



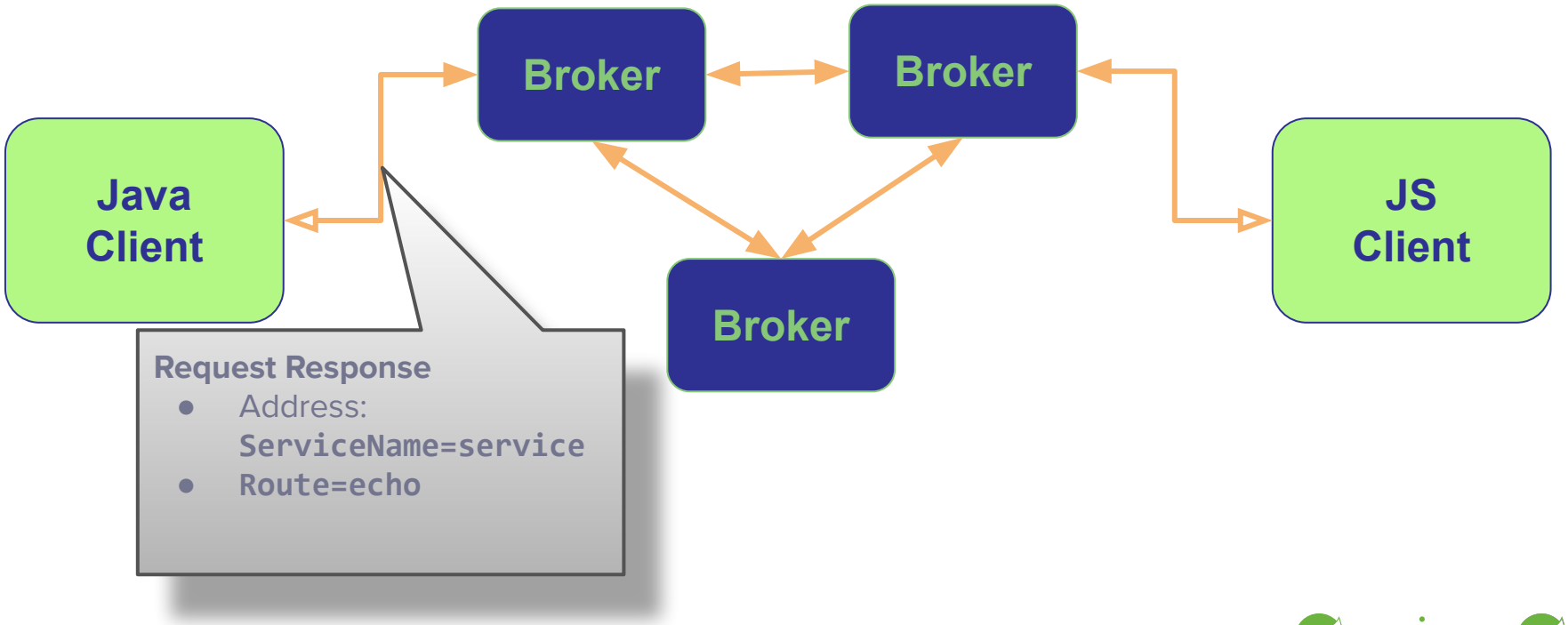
Service Registry



Making a Request

- Client already connected to Broker Cluster
- `Address` metadata
  - Who do I want to call?
    - `ServiceName` and/or other tags such as: `Region`, `InstanceName`, or `Version`
    - Routing metadata (such as ‘`add.user`’)
  - How to Route?
    - Unicast (default)
    - Multicast
    - Sharded

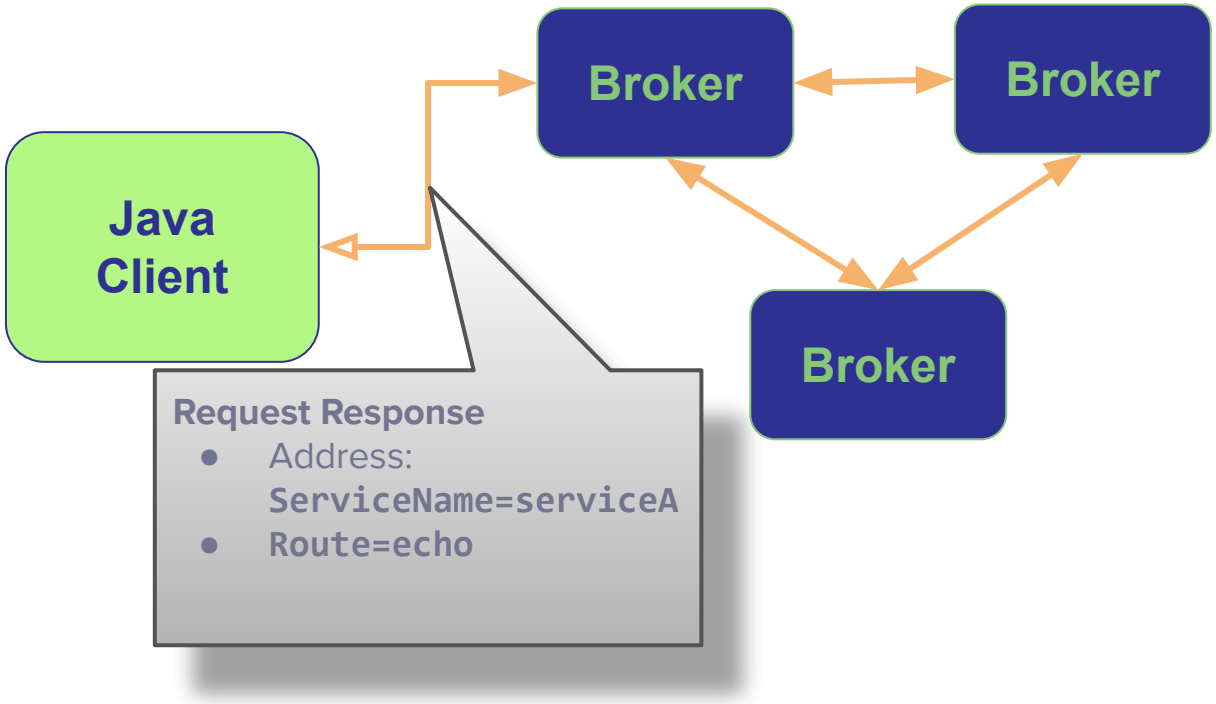
RSocket Routing Architecture



Making a Request

- No client side loadbalancer
- No sidecar
- No circuit breaker

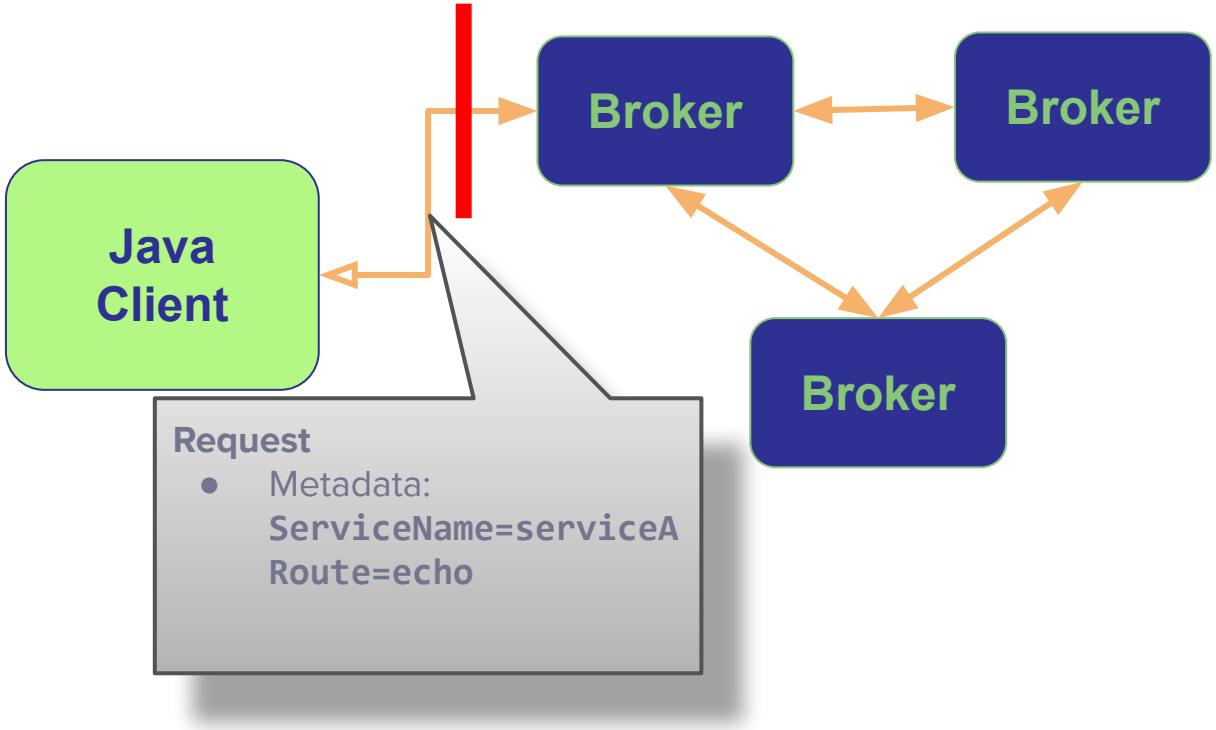
RSocket Routing Architecture



Requests to Non-existent Services

- Broker creates a placeholder
  - Applies 100% backpressure
- Avoids service startup ordering problems

RSocket Routing Architecture



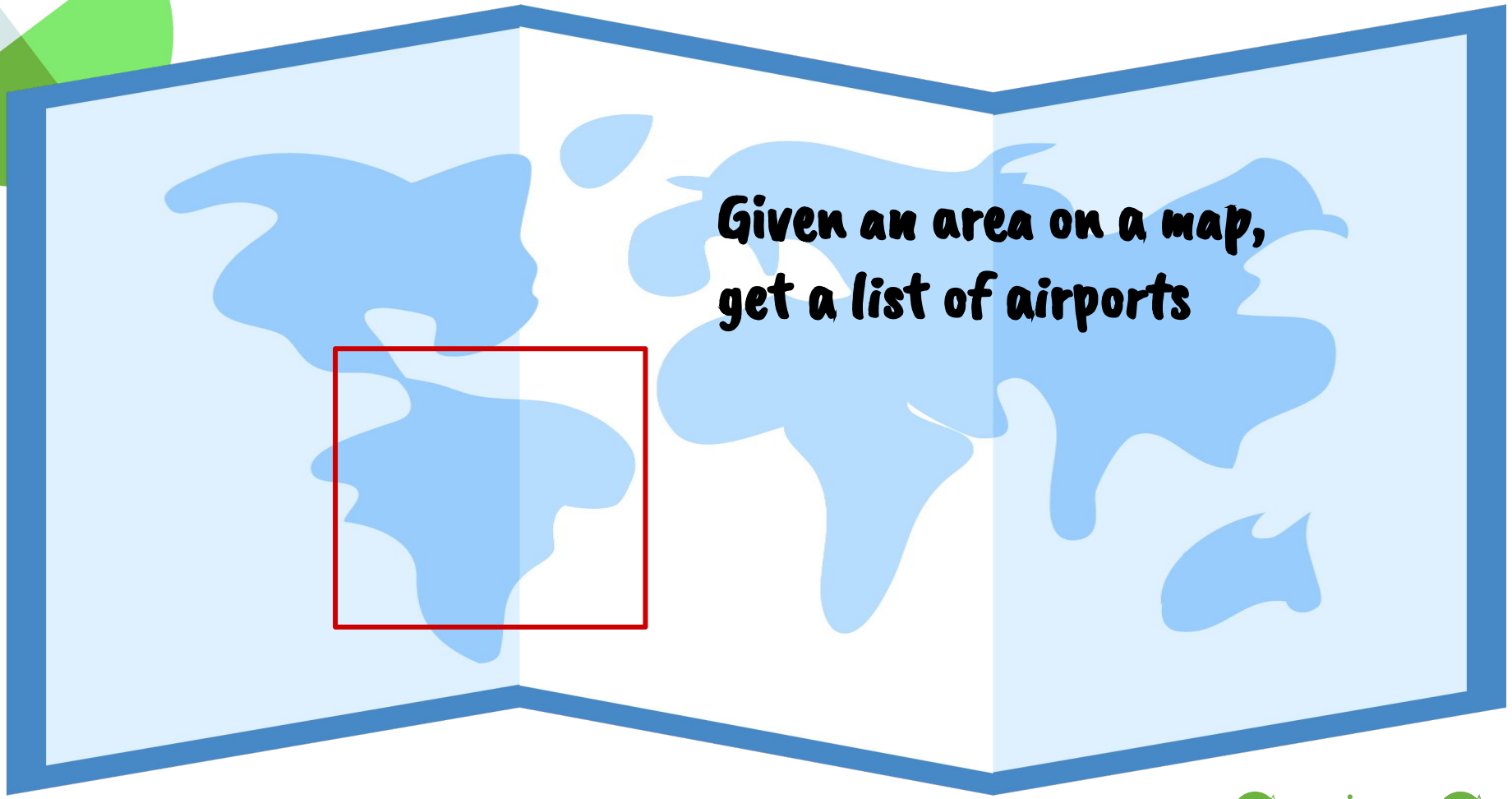
Requests are filtered

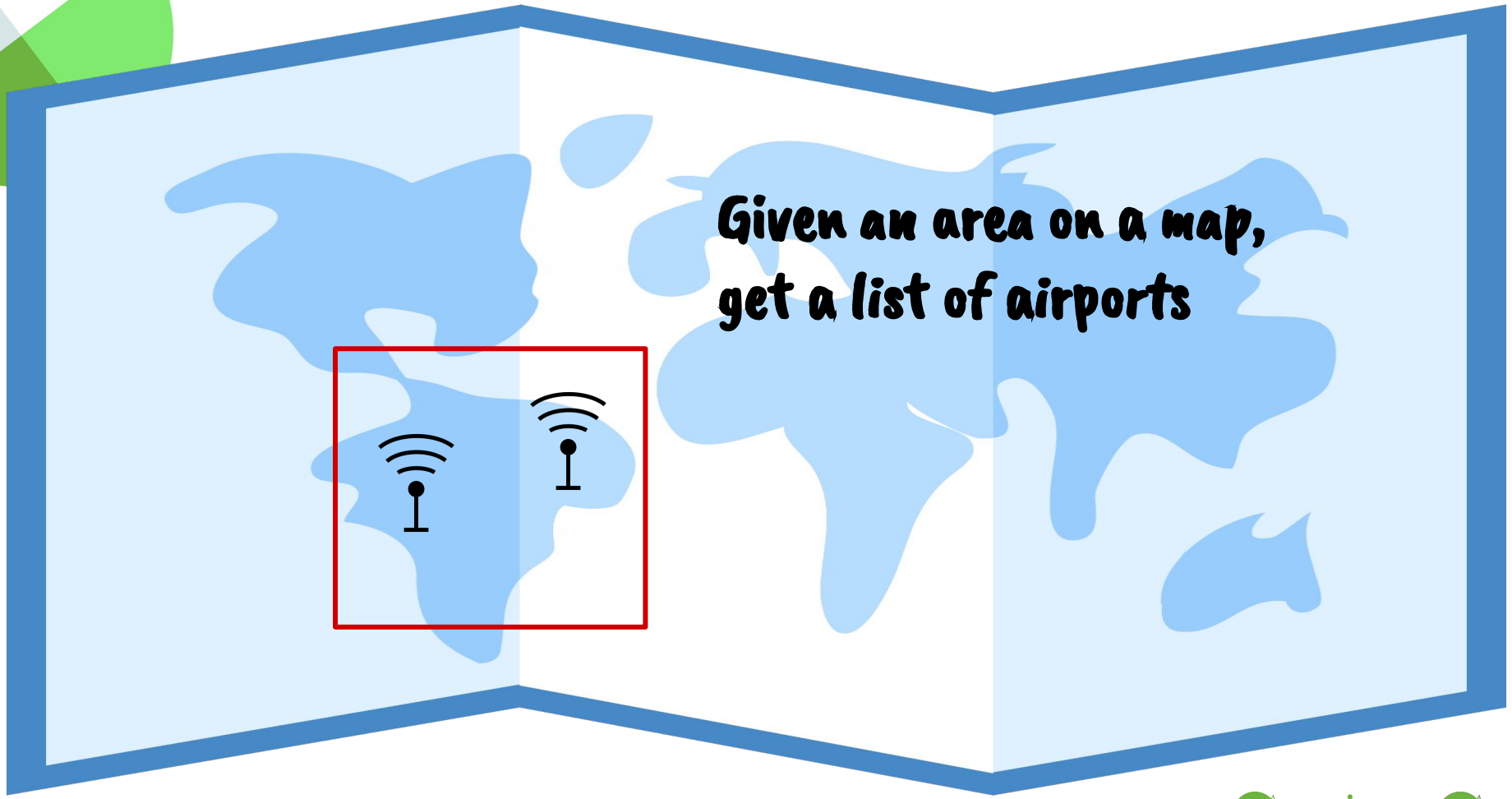
- Allows security at the request level
  - Is “Service A” allowed to talk to “Service B”
- Metrics collected at request level

To recap: Things you won't need...

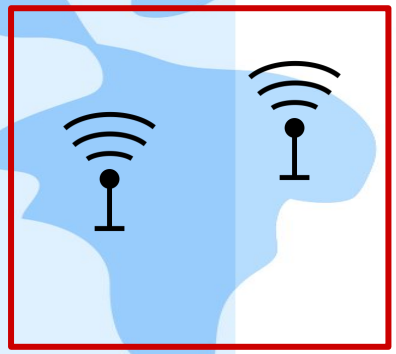
- Ingress permissions (except Broker)
- Separate Service Discovery
- Circuit Breaker
- Client-side load balancer
- Sidecar
- Startup ordering problems
- Special cases for warmup
- Message Broker

App Refactoring Demo


Given an area on a map,
get a list of airports


Given an area on a map,
get a list of airports

Given an area on a map, get a list of airports



Radar-Collector:

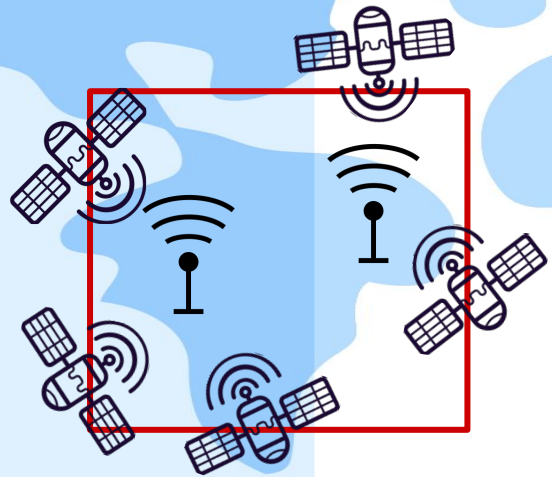
```
@MessageMapping("locate.radars.within")
Returns Flux<AirportLocation>
```

Flight-Tracker:

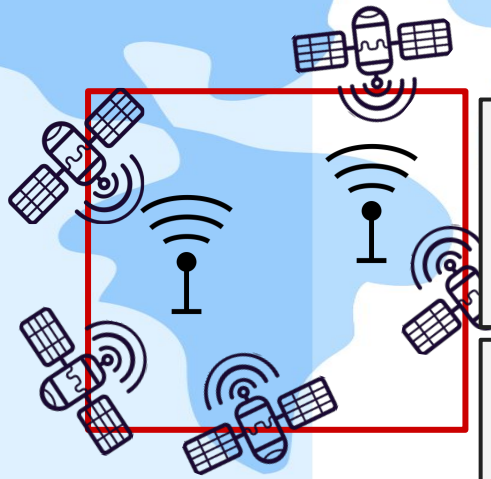
```
req.route("locate.radars.within")
     .data(box)
     .retrieveFlux(AirportLocation.class)
     .take(maxCount);
```

spring

For each airport (aka radar),
listen to the aircraft signals



For each airport (aka radar), listen to the aircraft signals



Radar-Collector:

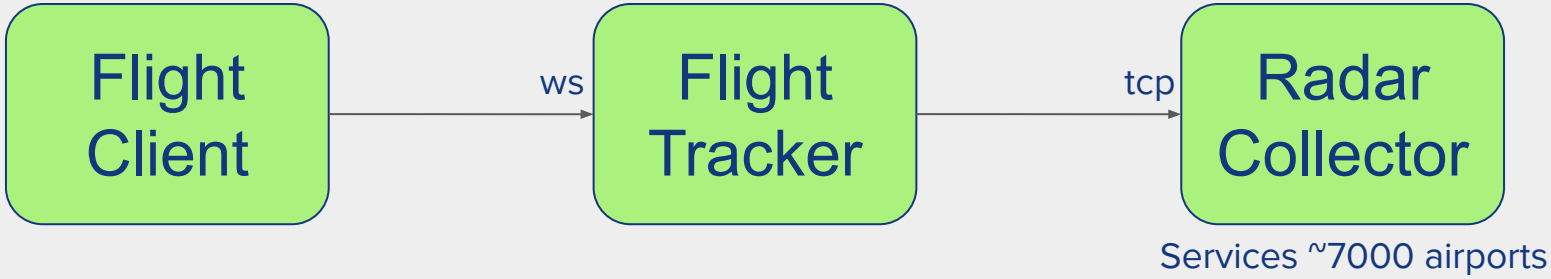
```
@MessageMapping("listen.radar.{type}.{code}")
Returns Flux<AircraftSignal>
```

Flight-Tracker:

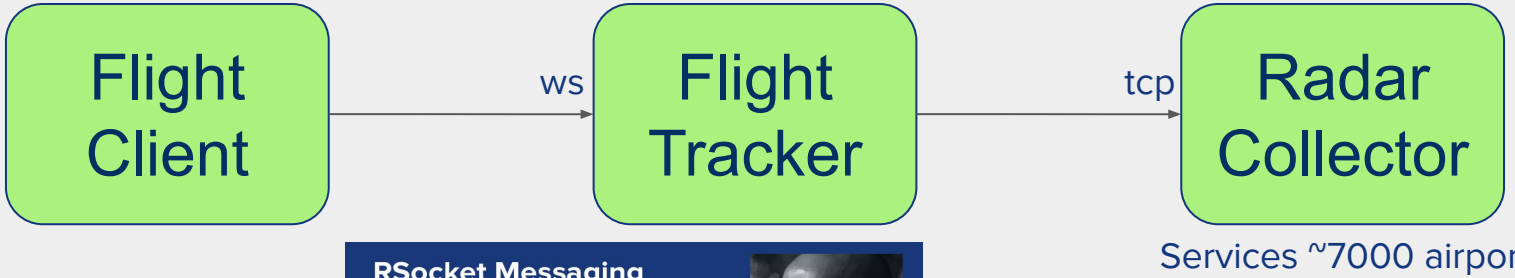
```
Flux.fromIterable(radars).flatMap(radar ->
req.route("listen.radar.{type}.{code}",
radar.getType(), radar.getCode())
     .retrieveFlux(AircraftSignal.class));
```

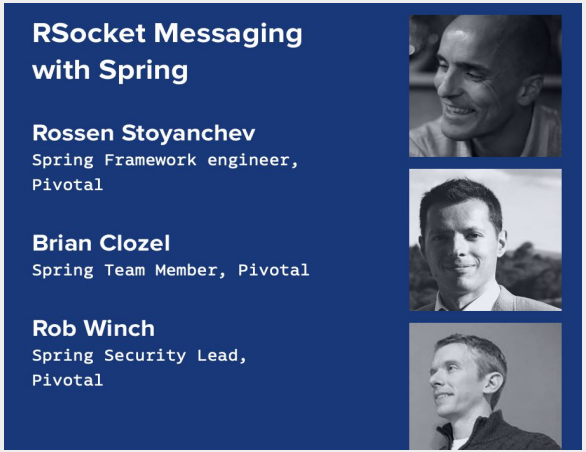
spring

Application Architecture

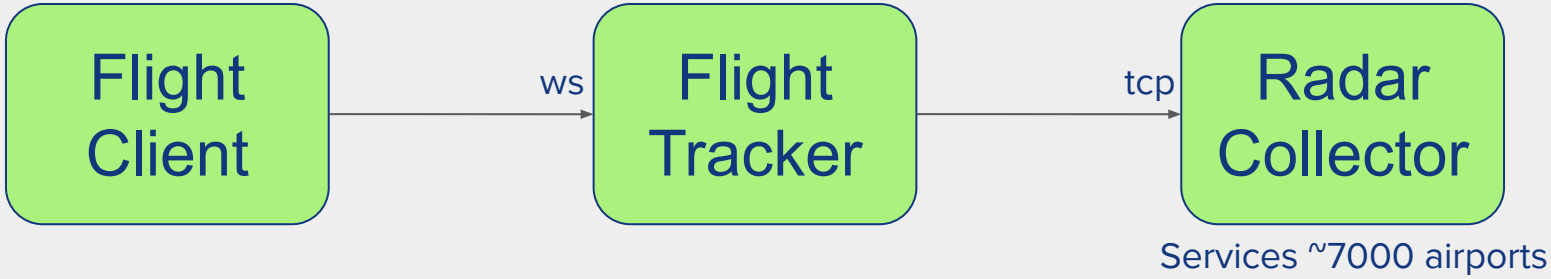


Application Architecture

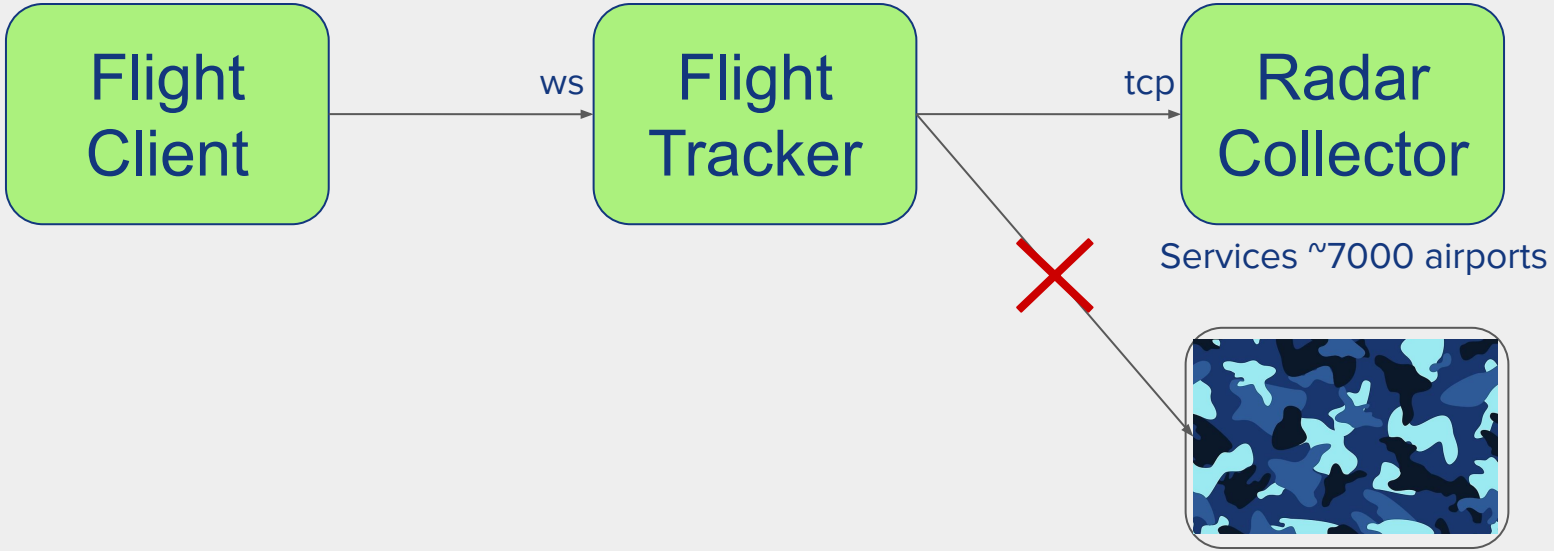




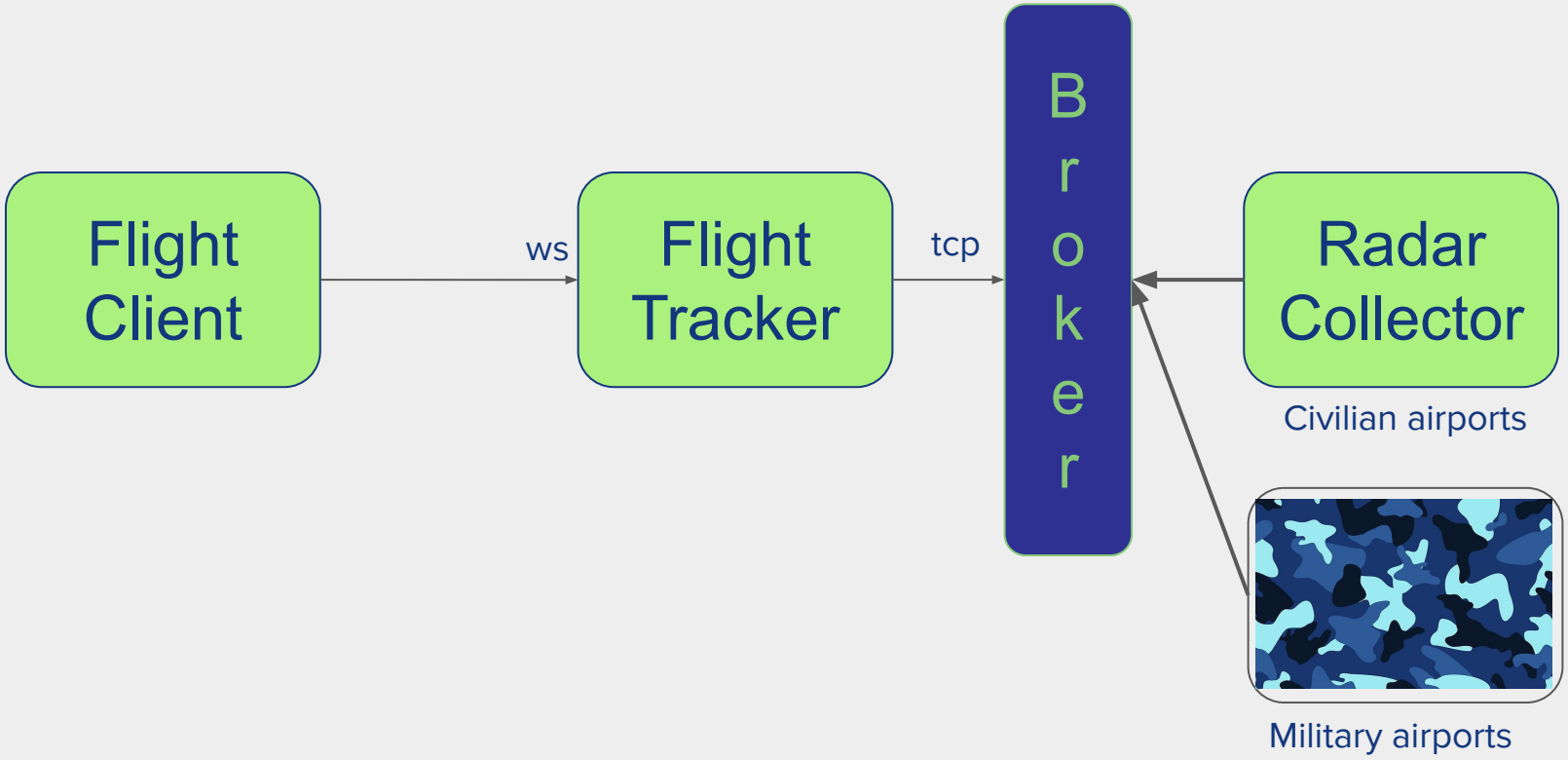
New Requirement:
Military Airports Require Isolation & Indirection



Challenge:
Direct Connection Has Disadvantages

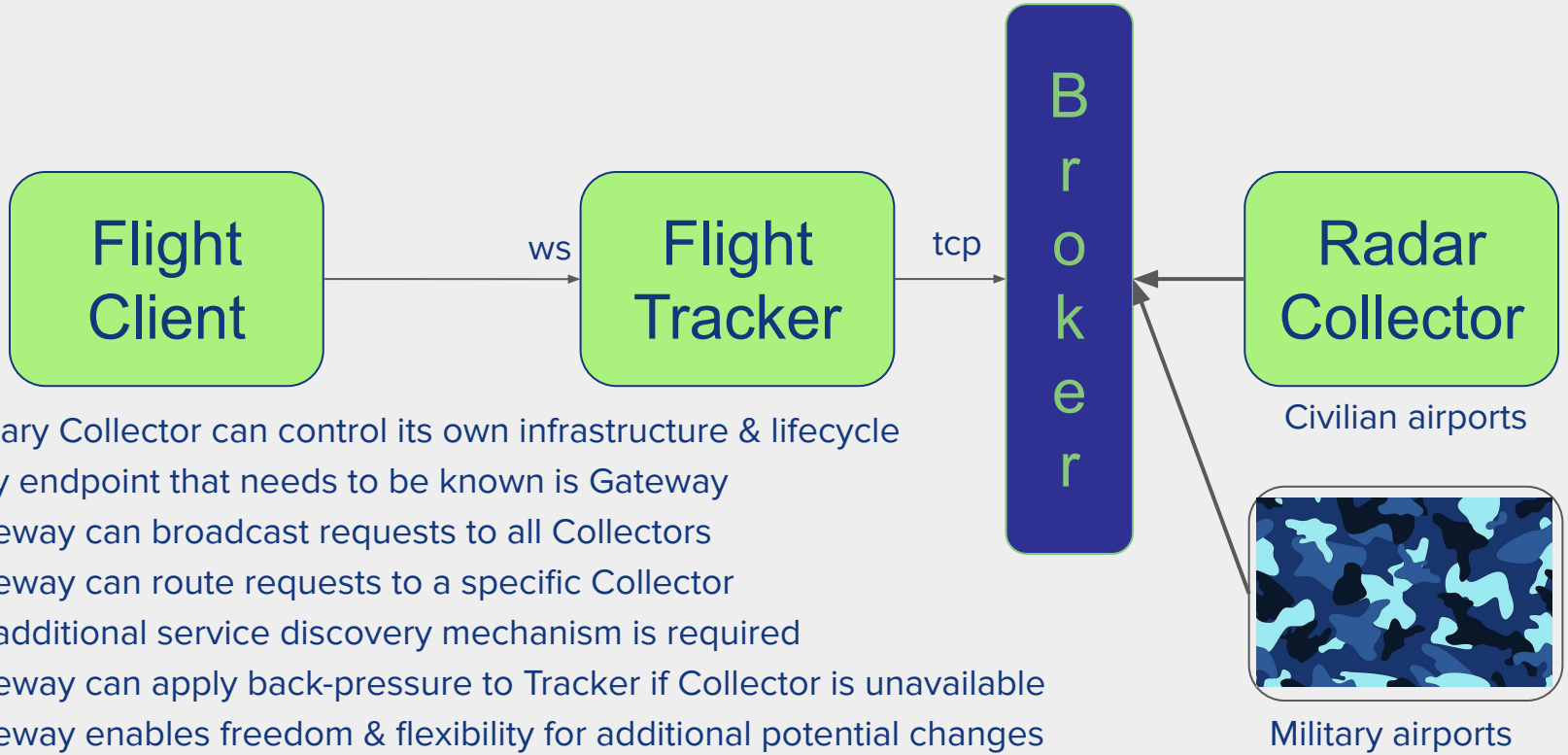


Gateway is a Perfect Solution



spring

Gateway is a Perfect Solution



- Military Collector can control its own infrastructure & lifecycle
- Only endpoint that needs to be known is Gateway
- Gateway can broadcast requests to all Collectors
- Gateway can route requests to a specific Collector
- No additional service discovery mechanism is required
- Gateway can apply back-pressure to Tracker if Collector is unavailable
- Gateway enables freedom & flexibility for additional potential changes

spring

Refactoring Step 1: Create a Gateway

```
spring:
  rsocket:
    server:
      port: 7002
  cloud:
    gateway:
      rsocket:
        enabled: true
        id: gateway
        route-id: 1
        service-name: gateway
        micrometer-tags:
          - component
          - gateway
```

```
dependencies:
  spring-cloud-gateway-rsocket-broker
  spring-boot-starter-actuator
```

Refactoring Step 2: Collector as Gateway Client

```
server:
  port: 0
spring.cloud.gateway.rsocket.client:
  broker:
    host: localhost
    port: 7002
  service-name: radar-collector
  route-id=10
  tags.instance-name=CIVILIAN
```

OR: route-id=11
instance-name=MILITARY

```
dependencies:
  spring-cloud-gateway-rsocket-client
  spring-boot-starter-actuator
  spring-boot-starter-webflux
  spring-boot-starter-rsocket
```

Refactoring Step 3: Tracker as Gateway Client

```
# Listen for Javascript websocket
client:
spring.rsocket.server:
  transport=websocket
  mapping-path=/rsocket

# Connect to gateway:
spring.cloud.gateway.rsocket.client:
  broker:
    host=localhost
    port=7002
  route-id=5
  service-name=flight-tracker
```

spring

Refactoring Step 4: Tracker as Gateway Client

```
@Component
public class RadarService {

	private RSocketRequester rSocketRequester;
	BrokerClient brokerClient;

	public RadarService(BrokerClient brokerClient) {
		this.brokerClient = brokerClient;
	}

	@EventListener
	public void getClient(PayloadApplicationEvent<RSocketRequester> event) {
		this.rSocketRequester = event.getPayload();
	}
	…
}
```

Refactoring Step 5: Multicast Request

```
// Leverages multi-casting feature in Gateway
public Flux<AirportLocation> findRadars(ViewBox box, int maxCount) {
      return rSocketRequester.route("locate.radars.within")
            .metadata(brokerClient.forwarding(builder -> builder
                        .serviceName("radar-collector")
                        .with("multicast", "true")))
            .data(box)
            .retrieveFlux(AirportLocation.class)
      .take(maxCount);
}
```

Refactoring Step 6: Request Routing Metadata

```
public Flux<AircraftSignal> streamAircraftSignals(List<Radar> radars) {

    return Flux.fromIterable(radars).flatMap(radar ->

        rSocketRequester.route("listen.radar.{type}.{code}", radar.getType(), radar.getCode())

        .metadata(brokerClient.forwarding(builder -> builder

                .serviceName("radar-collector")

                .with("INSTANCE_NAME", radar.getType())))

                .data(Mono.empty())

                .retrieveFlux(AircraftSignal.class));

}
```

Alternate Approach: Evrybady Canz Code

```
spring.cloud.gateway.rsocket.client:
  broker:
    host=localhost
    port=7002
  route-id=5
  service-name=flight-tracker
  forwarding:
    listen.radar.{type}.{code}:
      service_name: radar-collector
      instance_name: {type}
```

Refactoring Bonus Step: Deploy to Cloud (PAS)

Gateway application.properties

```
spring.rsocket.server:
  port: ${rsocket.broker.port:7002}
```

Collector and Tracker application.properties

```
# For connection to gateway:
spring.cloud.gateway.rsocket.client:
  broker:
    host: ${rsocket.broker.host:localhost}
    port: ${rsocket.broker.port:7002}
```

1. Parameterize the host/port assignments in the application properties

Refactoring Bonus Step: Deploy to Cloud (PAS)

manifest.yml

```
---
applications:
 - name: radar-gateway
   path: radar-gateway/build/libs/radar-gateway.jar
   routes:
     - route: tcp.clearlake.cf-app.com:1100
   env:
     RSOCKET_BROKER_PORT: 8080
```

1. Parameterize the host/port assignments in the application properties
2. Set the values of these parameterized properties in a Cloud Foundry manifest
3. cf push

```
---
applications:
 - name: radar-collector-civilian
   path: radar-collector/build/libs/radar-collector.jar
   no-route: true
   env:
     RSOCKET_BROKER_HOST: tcp.clearlake.cf-app.com
     RSOCKET_BROKER_PORT: 1100
     SPRING_PROFILES_ACTIVE: civilian
```

```
---
applications:
 - name: flight-tracker
   path: flight-tracker/build/libs/flight-tracker.jar
   routes:
     - route: flight-tracker.apps.clearlake.cf-app.com
   env:
     RSOCKET_BROKER_HOST: tcp.clearlake.cf-app.com
     RSOCKET_BROKER_PORT: 1100
```

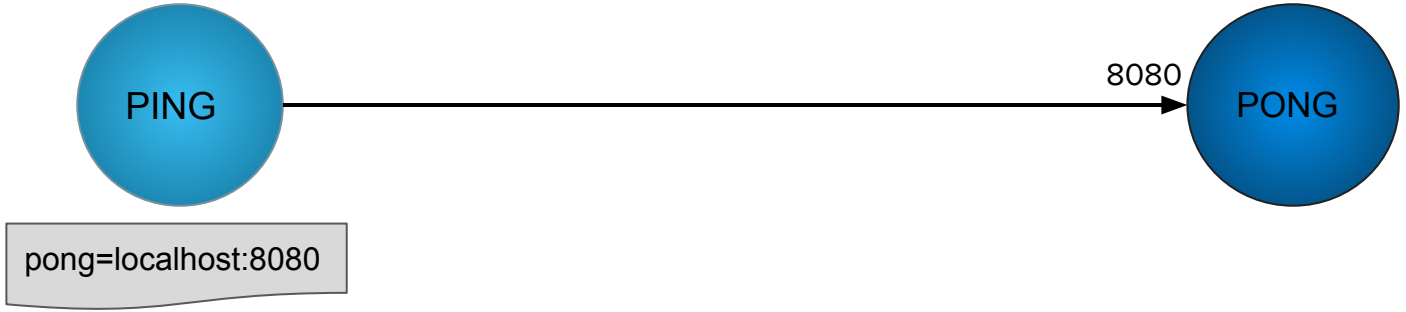
Deployed Apps on PAS

name	requested state	instances	memory	disk	urls
flight-tracker	started	1/1	1G	1G	flight-tracker.apps.clearlake.cf-app.com
radar-collector-civilian	started	1/1	1G	1G	
radar-collector-military	started	1/1	1G	1G	
radar-gateway	started	1/1	1G	1G	tcp.clearlake.cf-app.com:1100

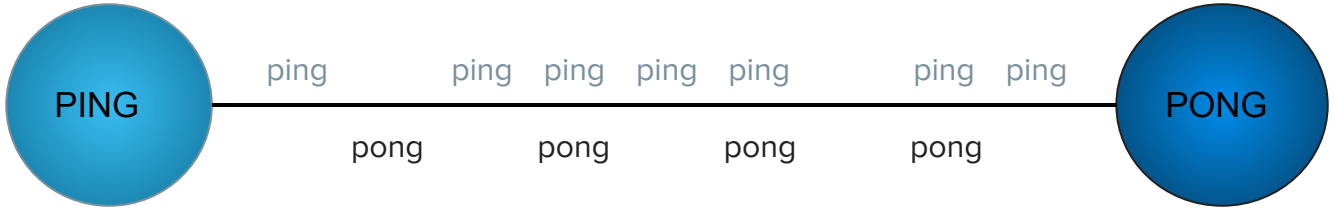
OR `radar-gateway.apps.internal`

RSocket Broker Demo

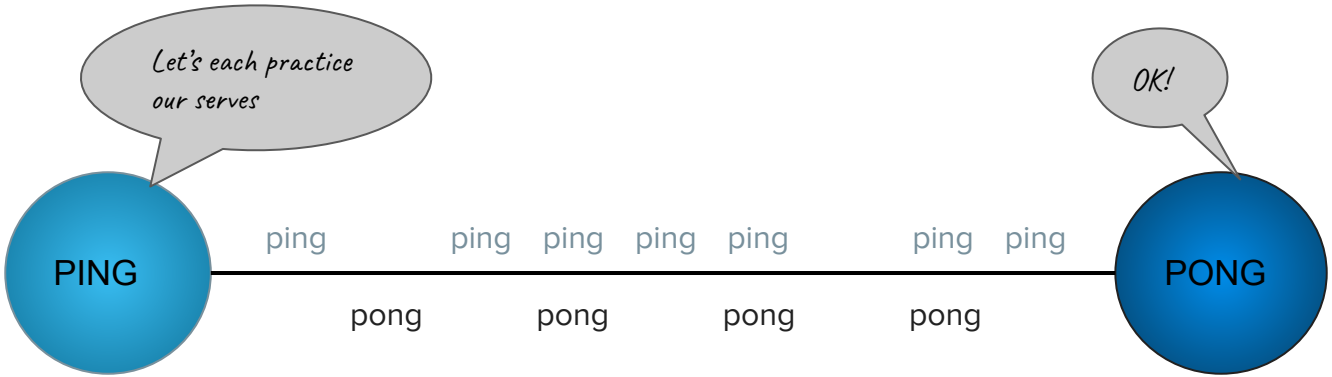
Ping initiates the connection to Pong



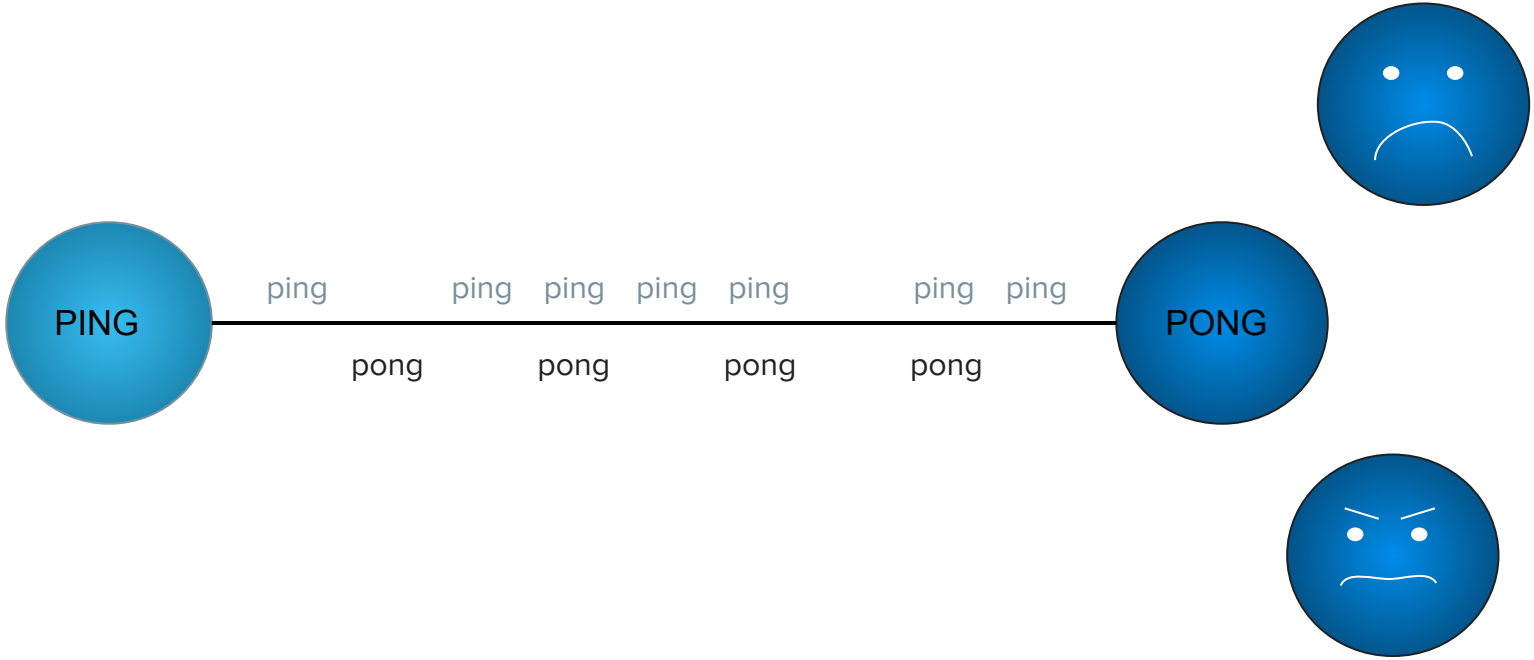
Either party can initiate requests/send messages



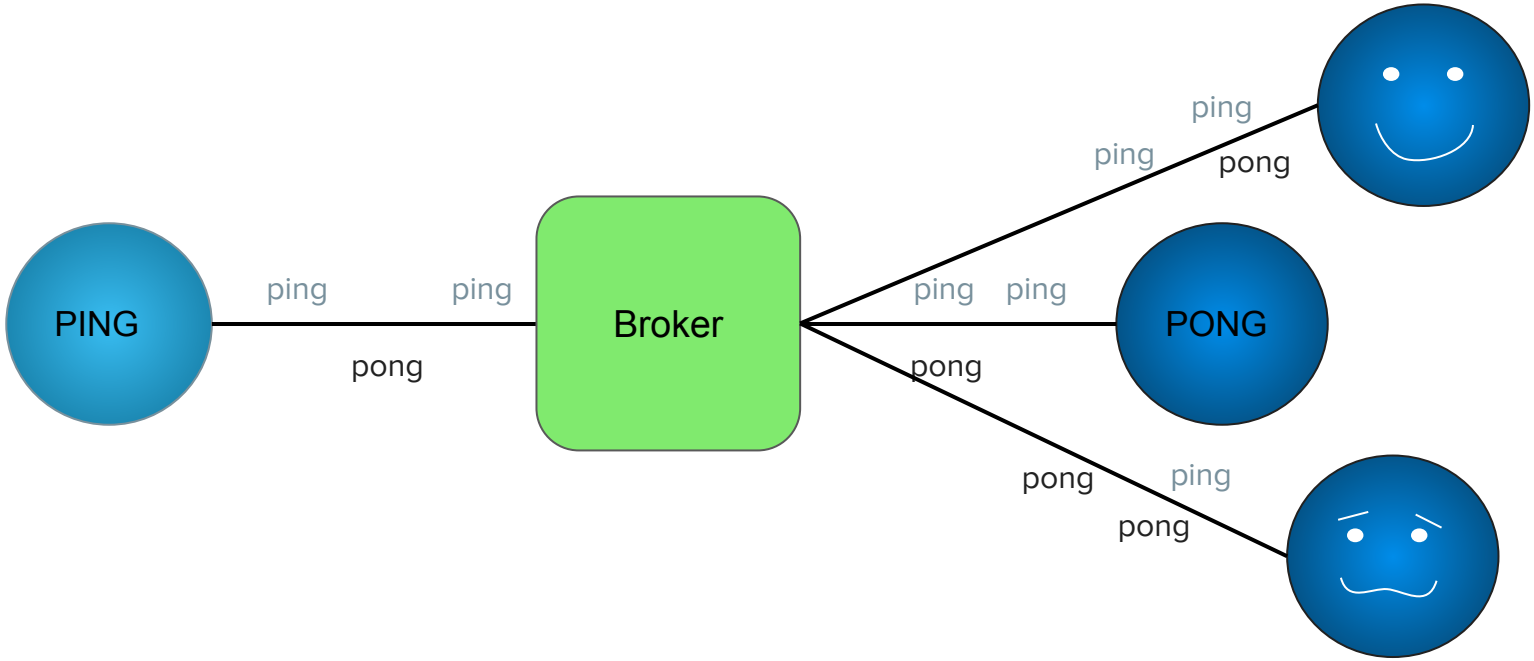
Demo illustrates “Request Channel” interaction



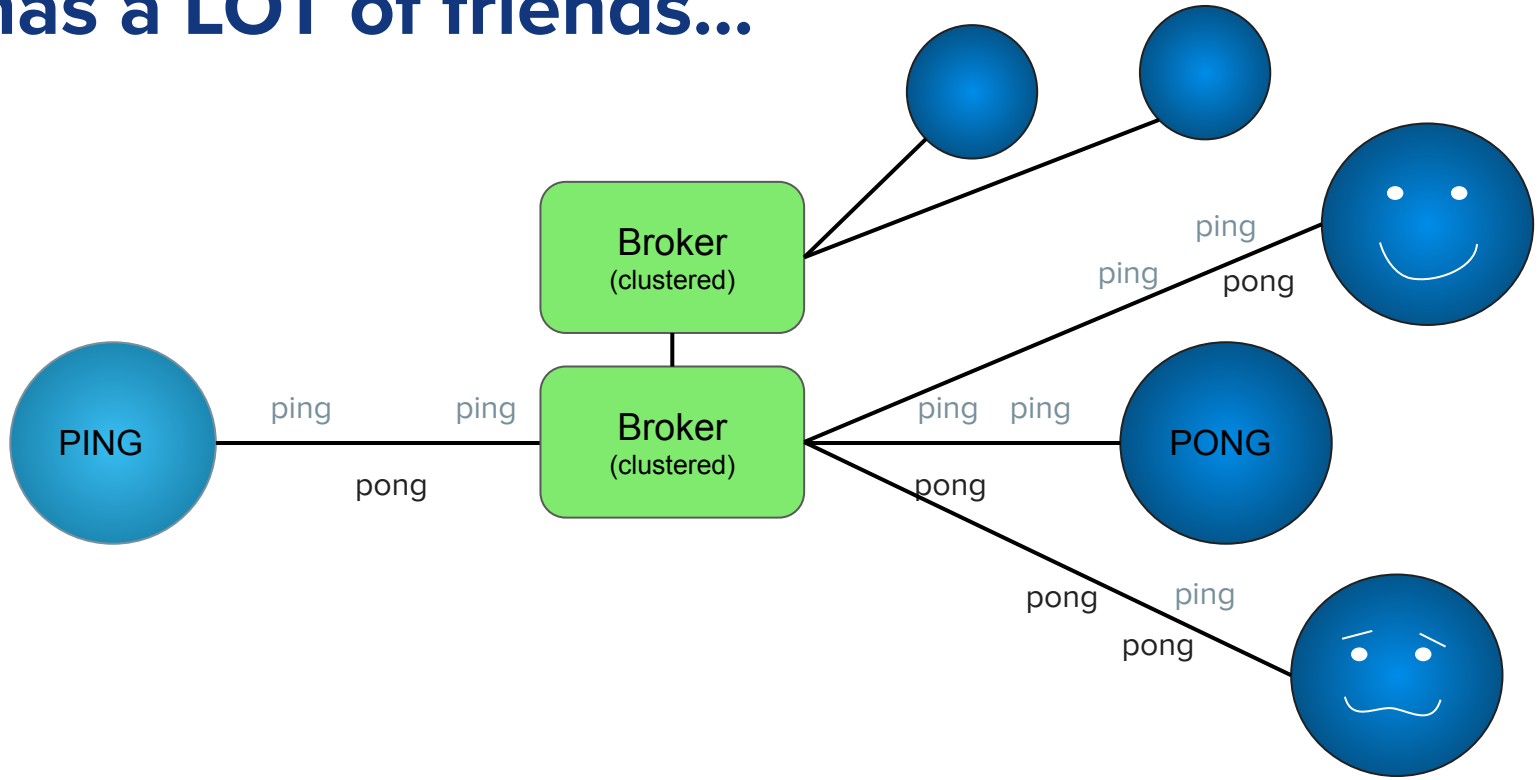
Pong has friends!



And the Broker knows where they live



Ping has a LOT of friends...



Roadmap

- Release will be under Reactive Foundation
- Full release after RSocket Routing spec is completed.
- Documentation
- Clustering Enhancements
- Tracing Integration
- Fault tolerance improvements
- Shard routing

Stay Connected.

https://github.com/spencergibb/rsocket-routing-sample

https://github.com/rsocket-routing/rsocket-routing-broker

https://github.com/rsocket-routing/rsocket-routing-client

https://github.com/rsocket-routing/rsocket-routing

Spencer Gibb @spencerbgibb

Stay Connected.

https://github.com/ciberkleid/spring-flights
https://github.com/spencergibb/spring-cloud-gateway-rsocket-sample
https://github.com/spring-cloud/spring-cloud-gateway

Cora Iberkleid @ciberkleid
Spencer Gibb @spencerbgibb