Welcome to the Reactive Revolution: RSocket & Spring Cloud Gateway

Spencer Gibb, Co-founder and Lead of Spring Cloud Core
@spencerbgibb

Contributions from
Ben Hale @nebhale, Rossen Stoyanchev @rstoya05, and Cora Iberkleid

Safe Harbor Statement

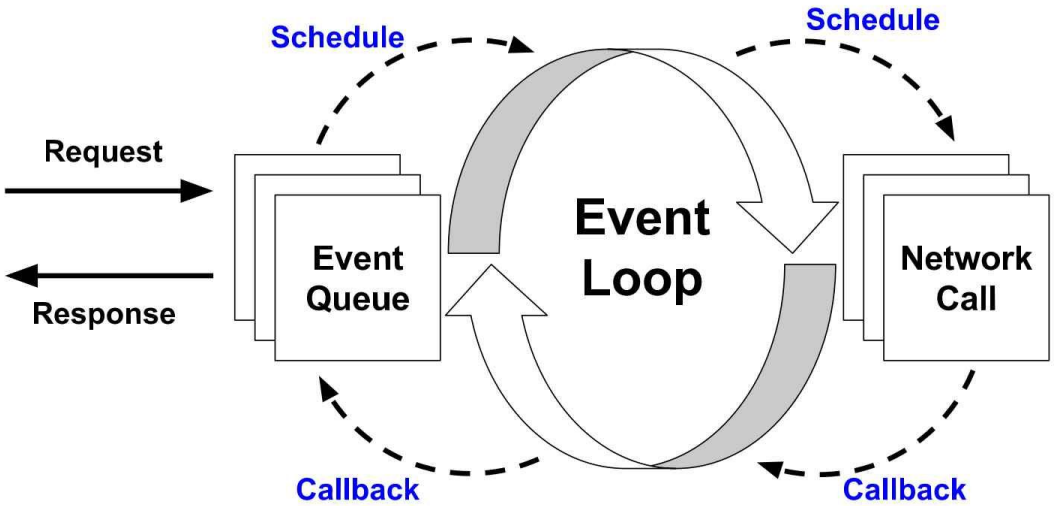
The following is intended to outline the general direction of Pivotal's offerings. It is intended for information purposes only and may not be incorporated into any contract. Any information regarding pre-release of Pivotal offerings, future updates or other planned modifications is subject to ongoing evaluation by Pivotal 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 Pivotal'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 Pivotal's offerings in this presentation remain at the sole discretion of Pivotal. Pivotal has no obligation to update forward looking information in this presentation.

Agenda

- Introduction
  - Reactive Architecture
  - Reactive Communication
  - RSocket Protocol
- Spring Cloud Gateway RSocket
- Demo
- Recap

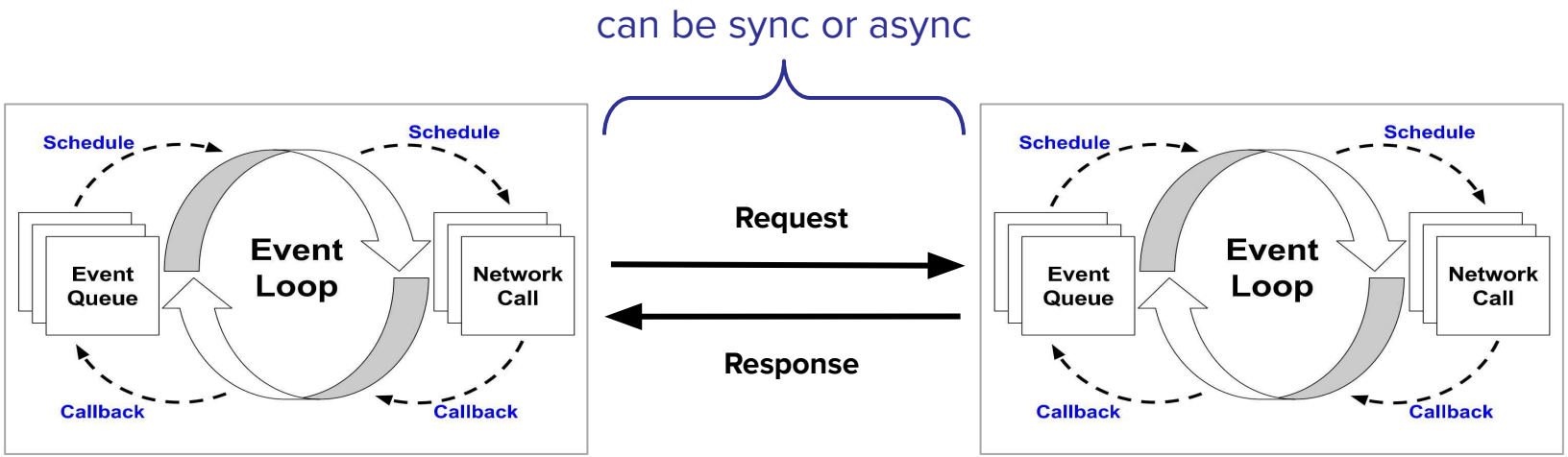
Reactive Architecture

Highly Efficient and Fundamentally Non-blocking

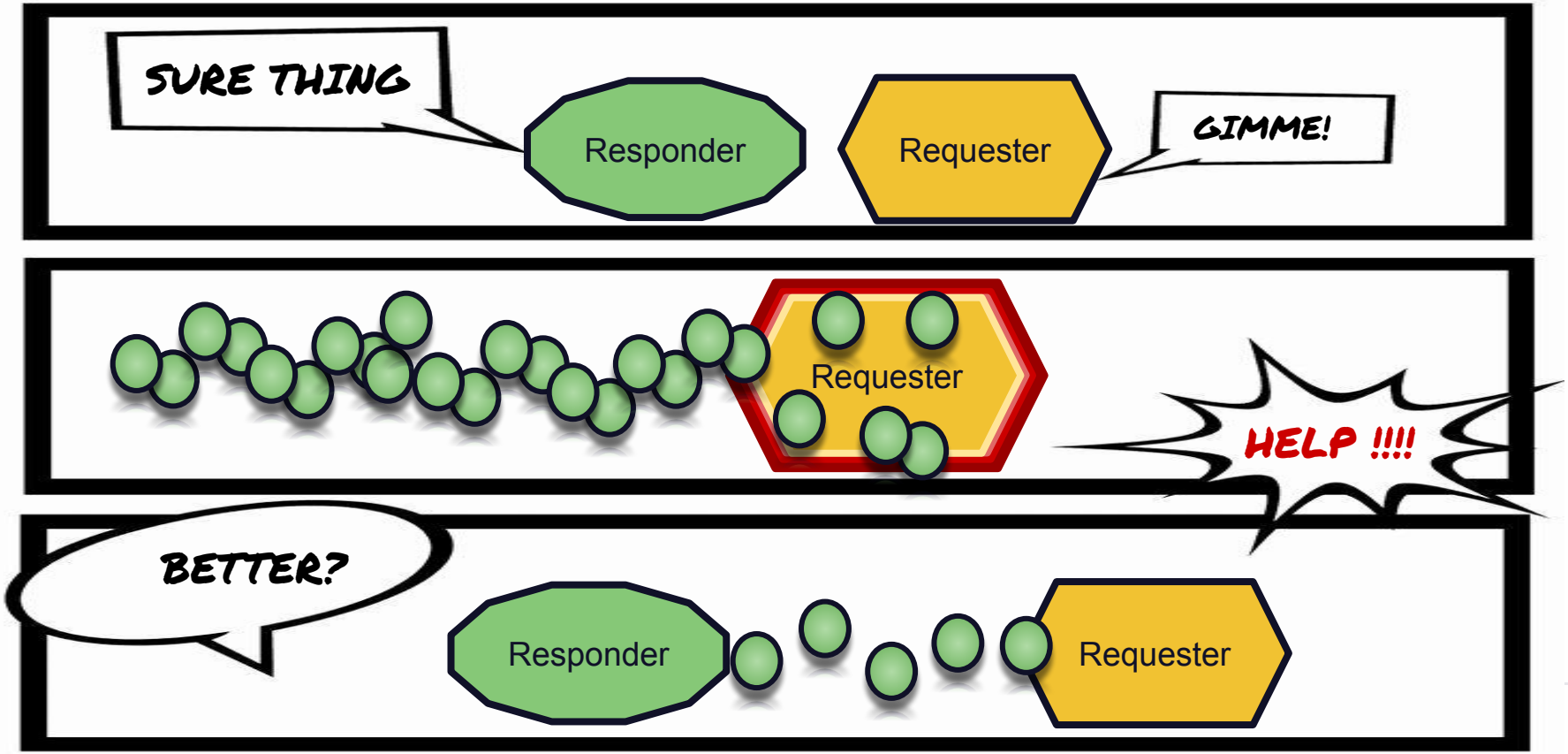


Reactive Inter-process Communication

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



Back Pressure



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)

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

Multiplexed

- Connections that are only used for a single request are massively inefficient (HTTP 1.0)
- Pipelining (ordering requests and responses sequentially) is a naive attempt solving the issue, but results in head-of-line blocking (HTTP 1.1)

- Multiplexing solves the issue by annotating each message on the connection with a *`stream id`* that partitions the connection into multiple "logical 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

Reactive Streams Back Pressure

- Network protocols generally send a single request, and receive an arbitrarily large response in return
- There is nothing to stop the responder (or even the requestor) from sending an arbitrarily large amount of data and overwhelming the receiver
- In cases where TCP back pressure throttles the responder, queues fill with large amounts of un-transferred data

- Reactive Streams (pull-push) back pressure ensures that data is only materialized and transferred when receiver is ready to process it

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

Bi-Directional

- Many protocols (notably not TCP) have a distinction between the client and server for the lifetime of a connection
- This division means that one side of the connection must initiate all requests, and the other side must initiate all responses
- Even more flexible protocols like HTTP/2 do not fully drop the distinction
  - Servers cannot start an unrequested stream of data to the client

- Once a client initiates a connection to a server, both parties can be requesters or responders to a logical stream

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

Message Driven Binary Protocol

- Requester-Responder interaction is broken down into frames that encapsulate messages
- The framing is binary (not human readable like JSON or XML)
  - Massive efficiencies for machine-to-machine communication
  - Downsides only manifest rarely and can be mitigated with tooling

- Payloads are bags of bytes
  - Can be JSON, XML, Protobuf, CBOR, etc... (it's all just 1's and 0's)

Metadata and Data in Frames

- Each Frame has an optional metadata payload
- The metadata payload has a MIME-Type but is otherwise unstructured

- Very flexible
  - Can be used to carry metadata about the data payload
  - Can be used to carry metadata in order to decode the payload
  - More specific announcement and routing metadata extensions forthcoming
- Generally means that payloads can be heterogenous and each message decoded uniquely

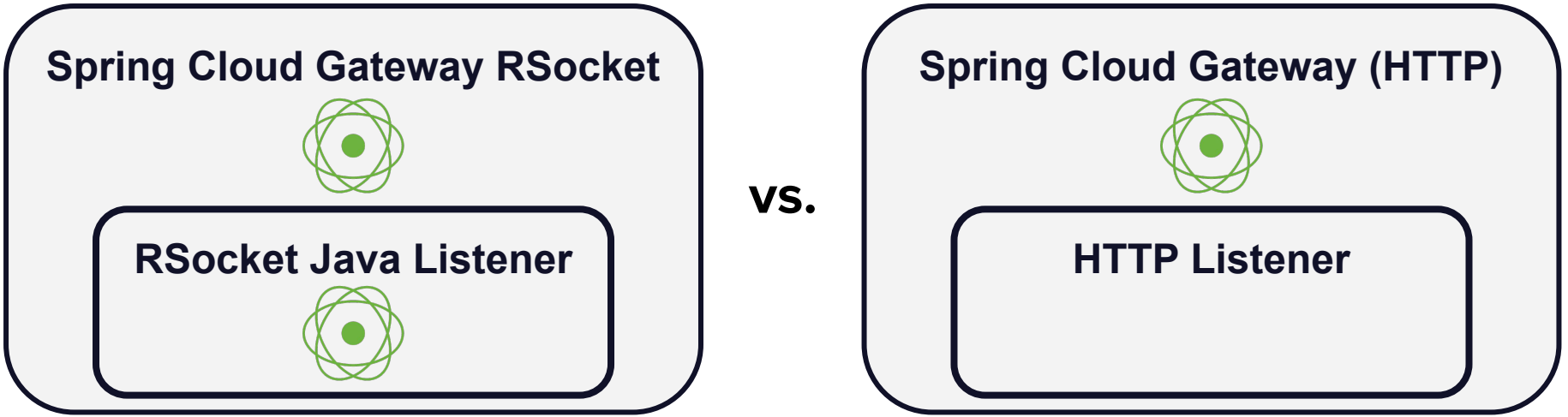
Spring Support

- Spring Framework 5.2 Messaging Support
- Spring Boot Auto-configuration
- Spring Security (future)

Spring Cloud Gateway RSocket

Spring Cloud Gateway RSocket

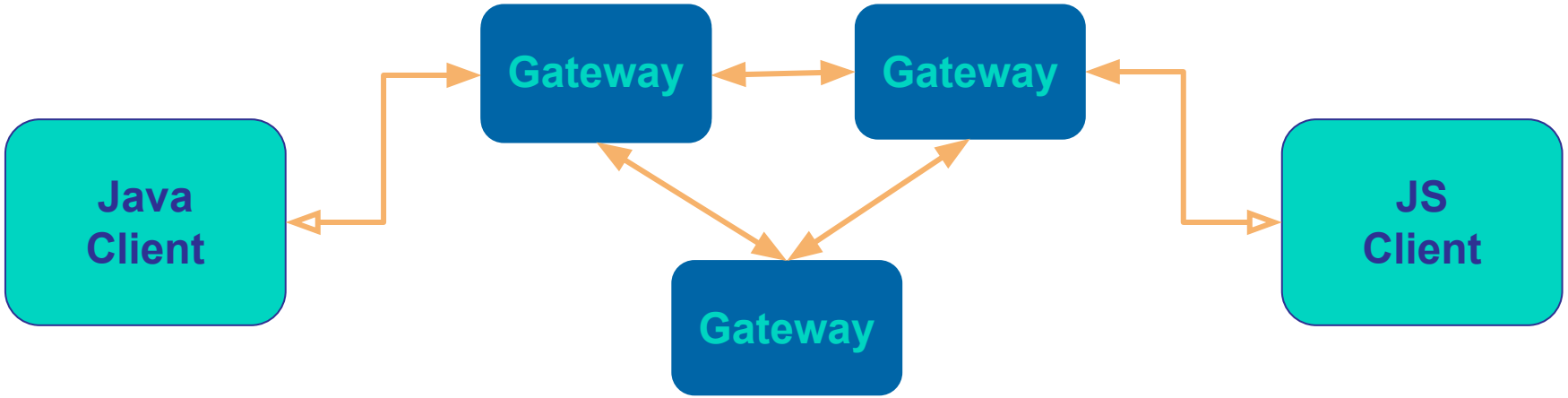
Reactive Runtime + Reactive Network Protocol



- With RSocket Java, the network layer is also built with Project Reactor
- Spring Boot auto-configures the RSocket listener

PROJECT REACTOR

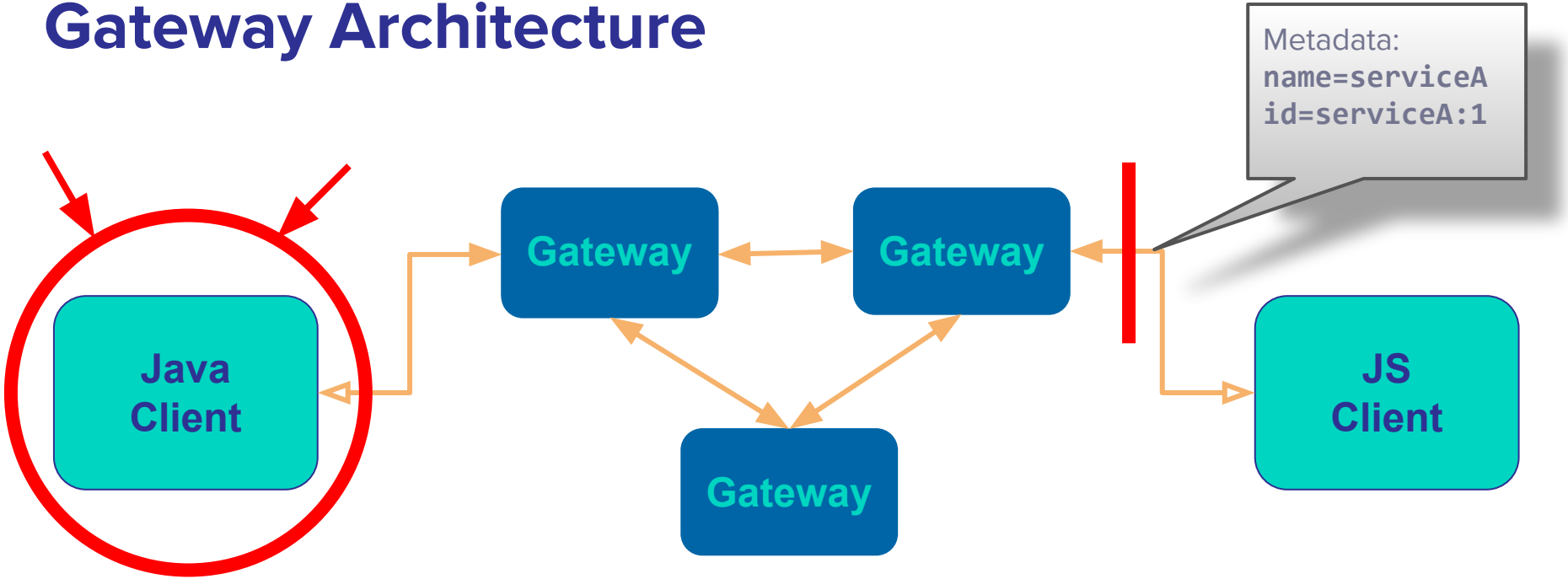
Gateway Architecture



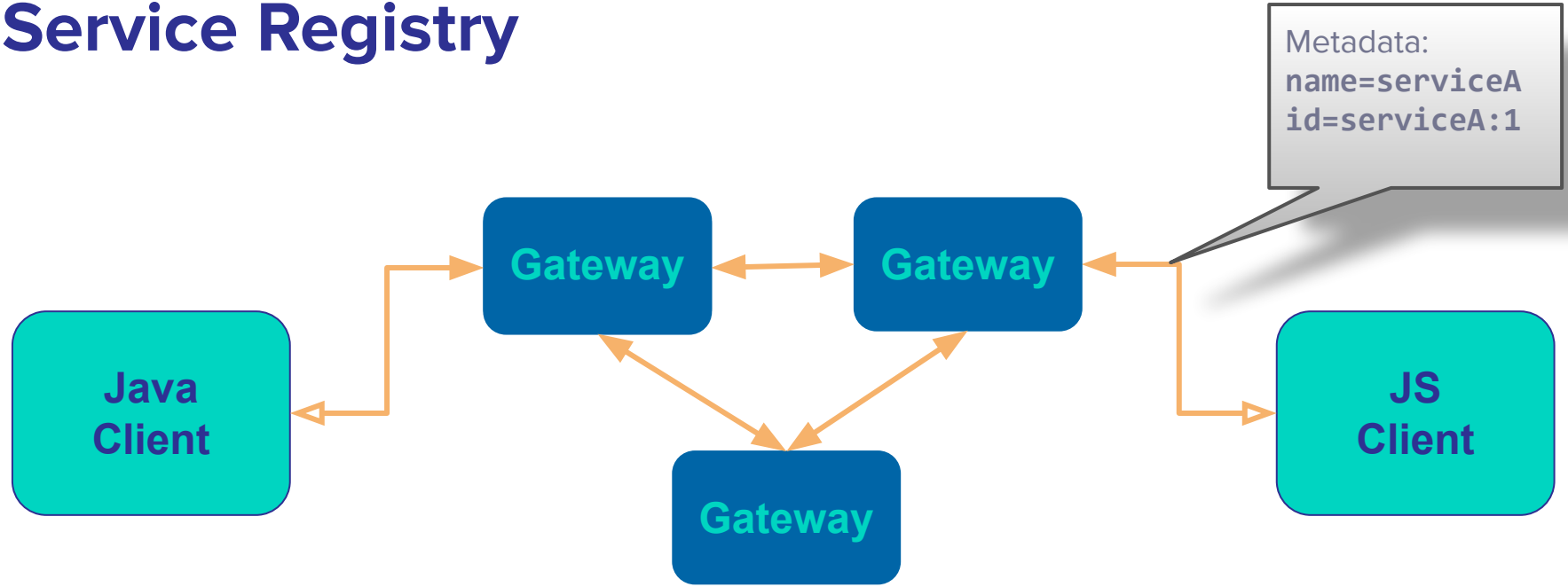
Connecting to Gateway RSocket

- Client makes connection to Gateway Cluster
- Sends connection level metadata
  - Who am I? Name and `Id`

Gateway Architecture



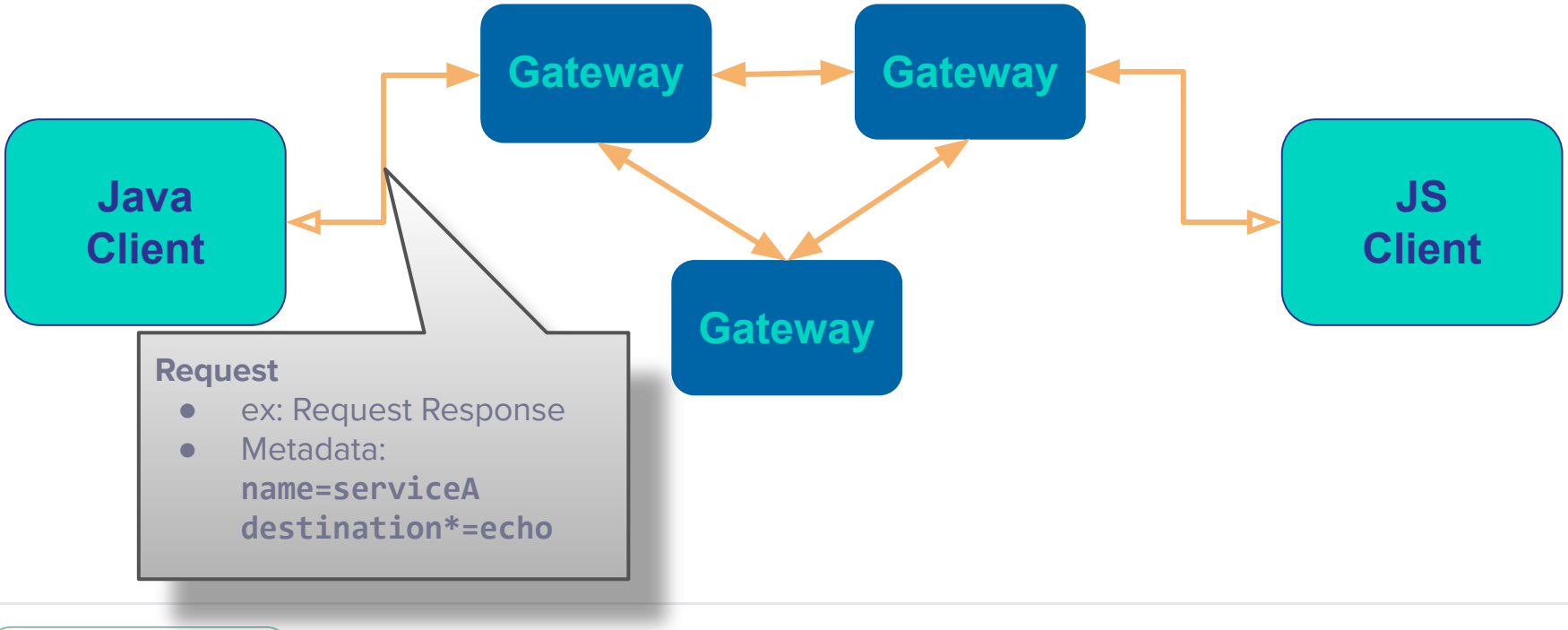
Service Registry



Making a request

- Client makes connection to Gateway Cluster
- Request level metadata
  - Who do I want to call? Name and Destination*

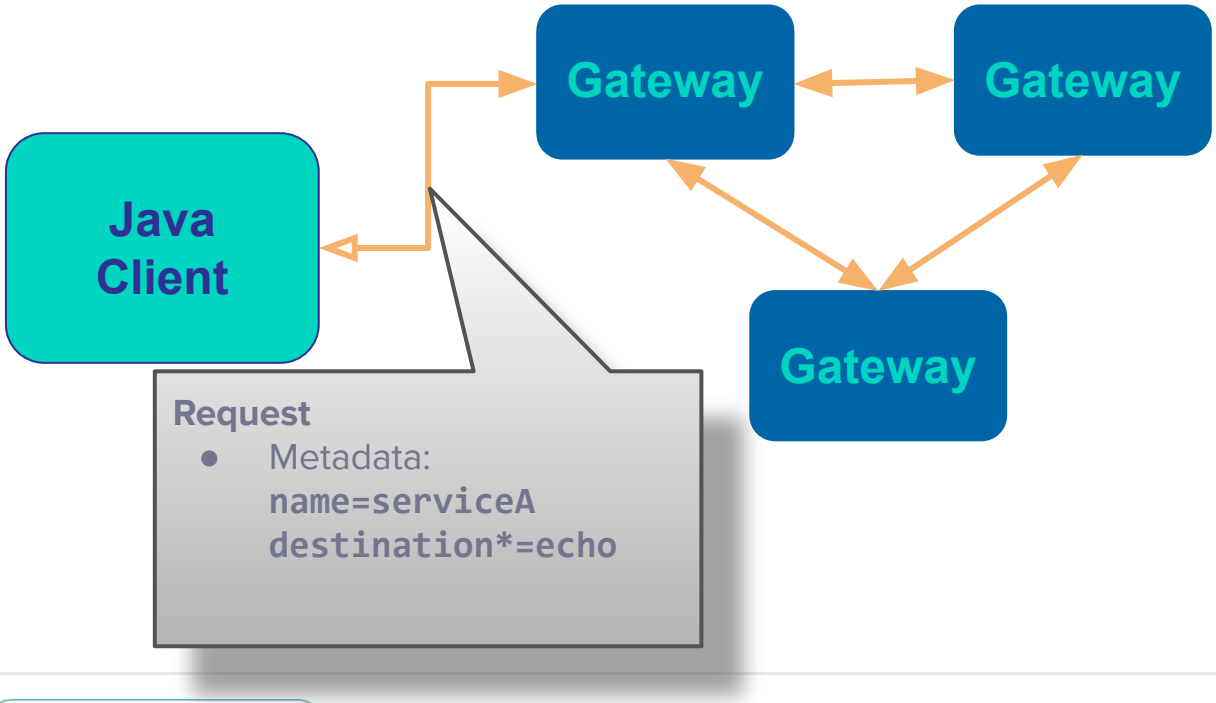
Gateway Architecture



Making a request

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

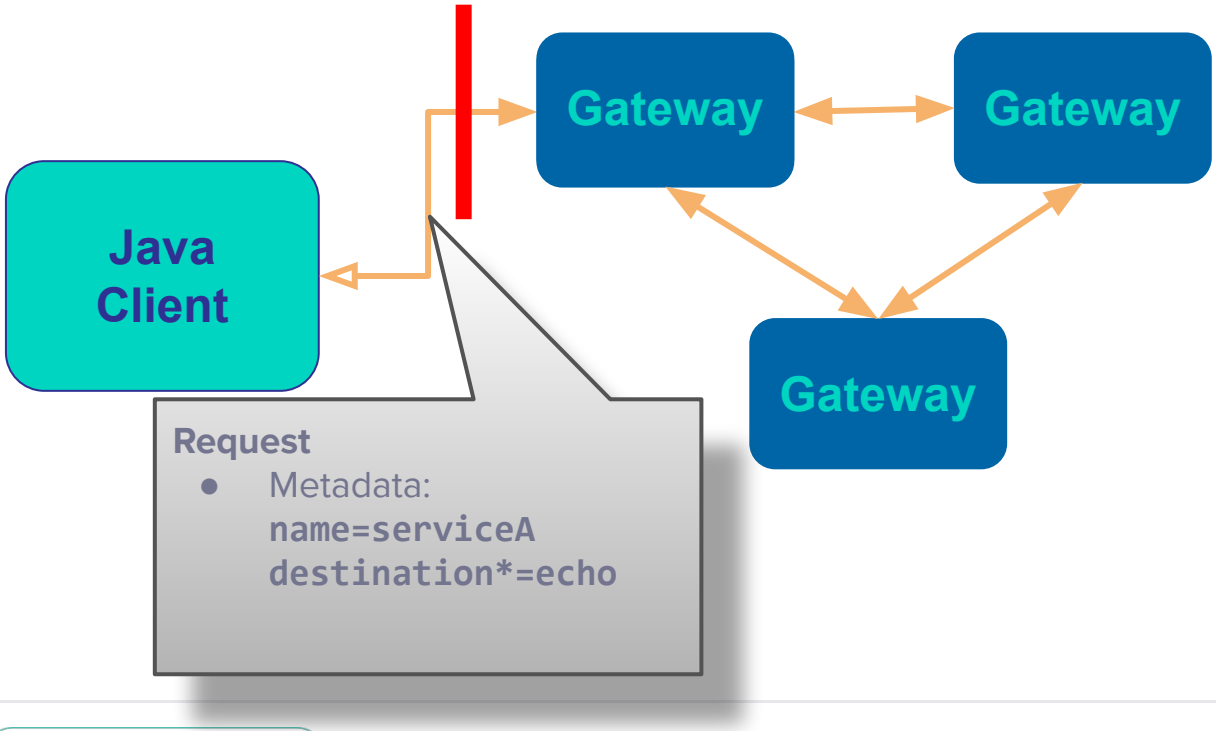
Gateway Architecture



Requests to non-existant services

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

Gateway Architecture

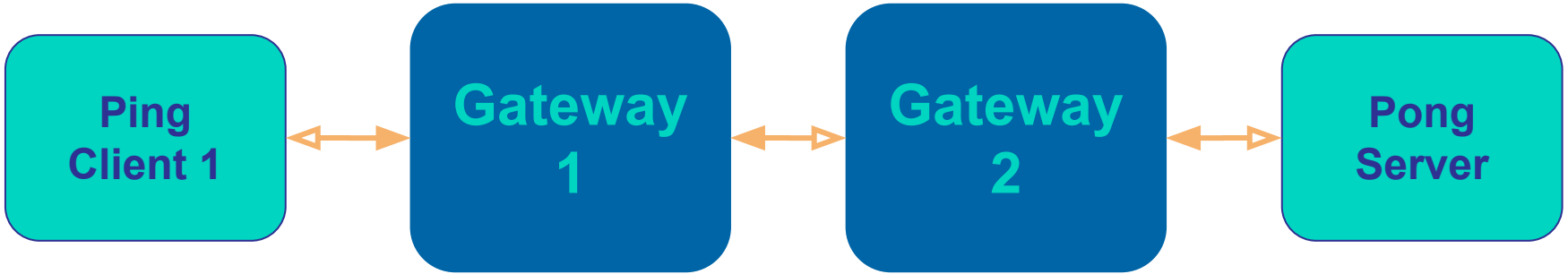


Requests are filtered

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

Demo

Demo setup



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

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

To recap: Additional Benefits

- via RSocket
  - persistent, multiplexed connections
  - multiple transports (TCP, Websockets, Http/2, etc...)
  - polyglot
    - only need standardized connection and request metadata
- Metrics via Micrometer

Roadmap

- Clustering Enhancements
- Messaging Semantincs (topic vs queue)
- Tracing integration
- Routing optimization
- Failure tolerance improvements
- Release
  - Builds on Spring Framework 5.2 & Spring Boot 2.2
  - Part of Spring Cloud Hoxton
  - Targets Q3 2019

Questions?

http://rsocket.io
https://github.com/spring-cloud/spring-cloud-gateway
https://github.com/spencergibb/spring-cloud-gateway-rsocket-sample
@spencerbgibb