# Interprocess Communication Patterns II

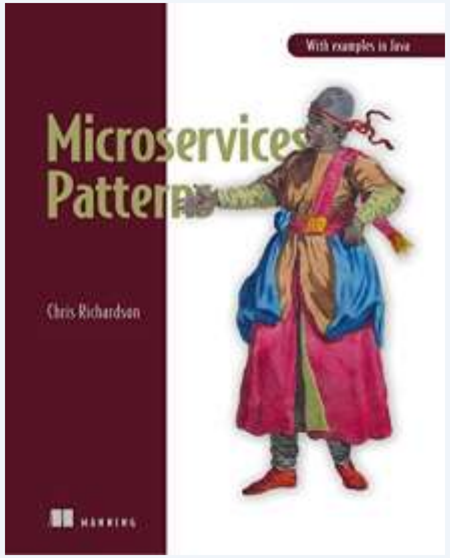
## Asynchronous Communication

# Objectives

- Communication between services
- Messaging Pattern:

  Context & Problem
  Forces
  Solutions
  Resulting Context
  Issues

- Message structure, messaging architecture, issues




Owerya RESOURCING

# Remember this?

| | One –to-one | One-to-many |
|---|---|---|
| Synchronous | Request/Response<br>• Response in timely fashion (might event block)<br>• Tight coupling<br>One-way notifications | - |
| Asynchronous | Async Request/Response<br>• Client doesn't block<br>• Loose coupling<br><br>One-way notifications | Publish/subscribe<br>• Client publishes a notification message, one or more servers consume<br>Publish/async response<br>• Client publishes request, waits a certain time for responses |

# Pattern: Asynchronous Messaging

Asynchronous interprocess communication

# Context (Same as RPI)

- The Microservices Architecture Pattern has been applied
- The services must handle requests from external clients and services
  - This requires service collaboration which means inter-process communication

# Forces (Same as RPI)

- Services often need to collaborate
- Synchronous communication means tight runtime coupling
  - Both client and server must be available for the duration of the request

# Solution

- Use asynchronous messaging for inter-service communication
  - Services communicate by exchanging messages over messaging channels

Messaging is inherently asynchronous, but we could implement this by blocking until a reply is received

- Several different styles:
  - Request/response – a request is sent to a recipient; a reply message expected promptly
  - Notifications – a message is sent to a recipient; no reply is expected or sent.
  - Request/asynchronous response - a request is sent to a recipient; a reply message expected eventually
  - Publish/subscribe – a message is published to zero or more recipients
  - Publish/asynchronous response – a request is published to one or more recipients; some of whom reply.

# Message Structure

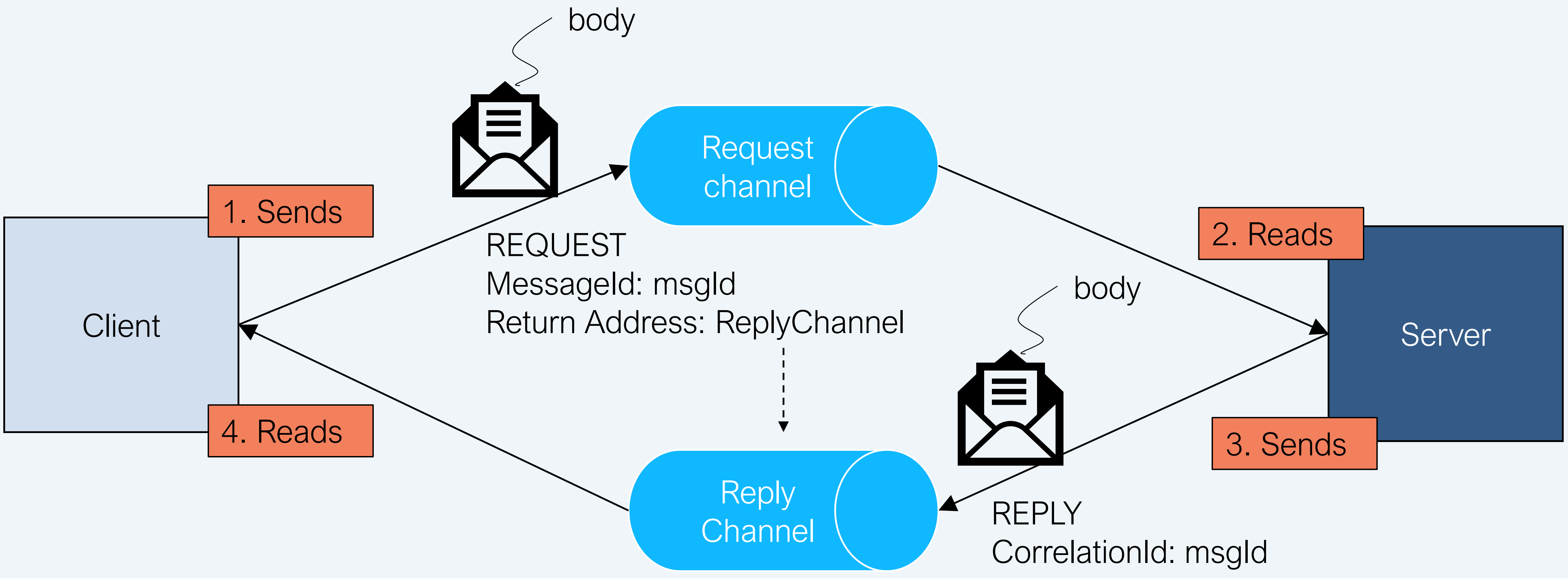
- All messaging systems will require a header/body structure

- Header – key-value pairs giving metadata identifying the message and routing
  - message id (must be unique)
  - (optional) message source
  - (optional) return address
  - (optional) content type
  - …

# Message Structure

- Body – the message being sent in text or binary format. This could contain:
  - A **Document** – generic message containing only data, receiver decides how to interpret.
    - E.g. a response to a command
  - A **Command** – specifies the operation to invoke and its parameters
  - An **Event** – indication something notable has happened

# Asynchronous Request/Response Message Style



# Group Exercise

## Synchronous and Asynchronous Communication

# Messaging Protocols

- For communication to occur both client and server (or peers) need to agree on the messaging protocol.
- There are a lot of these around, two of the more popular ones:
  - AMQP (Advanced Message Queueing Protocol)
  - JMS (Java Message Service)

# Messaging Protocols: AMQP

- Developed by JPMorgan Chase.
- Designed for very large volumes of data, the onus is on the consumer to ensure all messages are processed.
- It is a wire-level protocol, published standard. This is like an API, so anyone implementing it can interface with other AMQP equipped tech.
- Works with binary data only.
- Supports point-to-point and publish subscribe models.

# Messaging Protocols: JMS

- Developed by Sun Microsystems.
- Designed for smaller message volumes in which confirmation of delivery is key.
  - Sender is responsible for confirming delivery
- An API specification, does not guarantee interoperability between implementations.
- Is primarily a Java technology and doesn't normally interface well with other techs.
- Supports multiple message types including text, stream and serialized object.
- Supports point-to-point and publish/subscribe models.

# Messaging Brokers

- AMQP based brokers (tends to support multiple platforms)
  - RabbitMQ
  - Amazon MQ
  - Apache Qpid

- JMS based brokers (Java based)
  - ActiveMQ

- Apache Kafka
  - An Open-Source distributed event-streaming platform with its own event protocol, but has bridges for AMQP and JMS sources.

- Cloud provider native, e.g. AWS SQS / SNS, Gcloud Firebase, Azure Service bus, event grid etc

# Resulting Context / Consequences

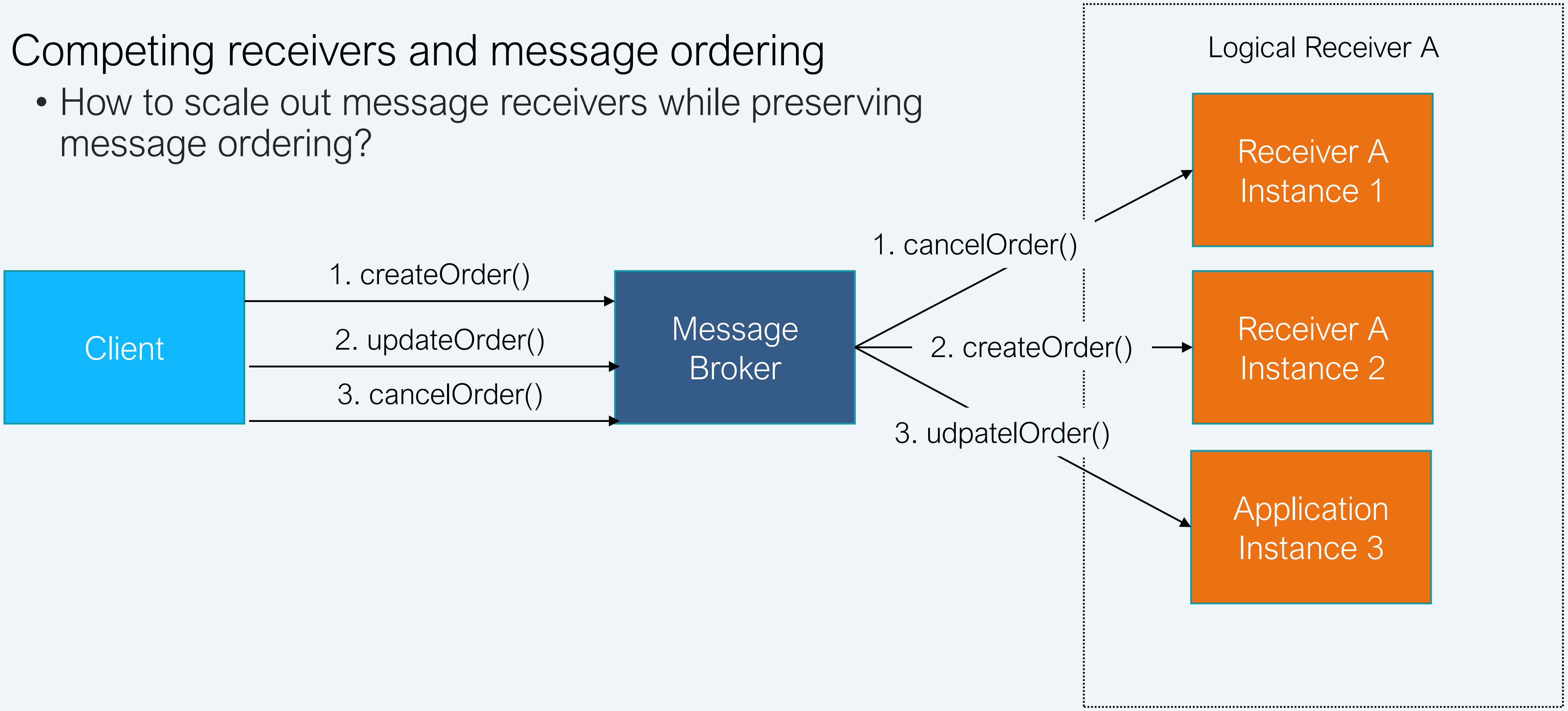
- Benefits
  - Loose runtime coupling
    - Decouples message sender from message consumer
  - Improved availability
    - Message broker buffers messages until the consumer is able to process them
  - Support for a variety of communication patters
- Drawbacks
  - Message broker introduces additional complexity – must be highly available
- Issues
  - Request/reply style communication is more complex

# Related Patterns

- Remote Procedure Invocation is an alternative
- The Saga pattern and CQRS pattern (see later) use messaging

# Messaging Design Issues

- Competing receivers and message ordering
  - How to scale out message receivers while preserving message ordering?

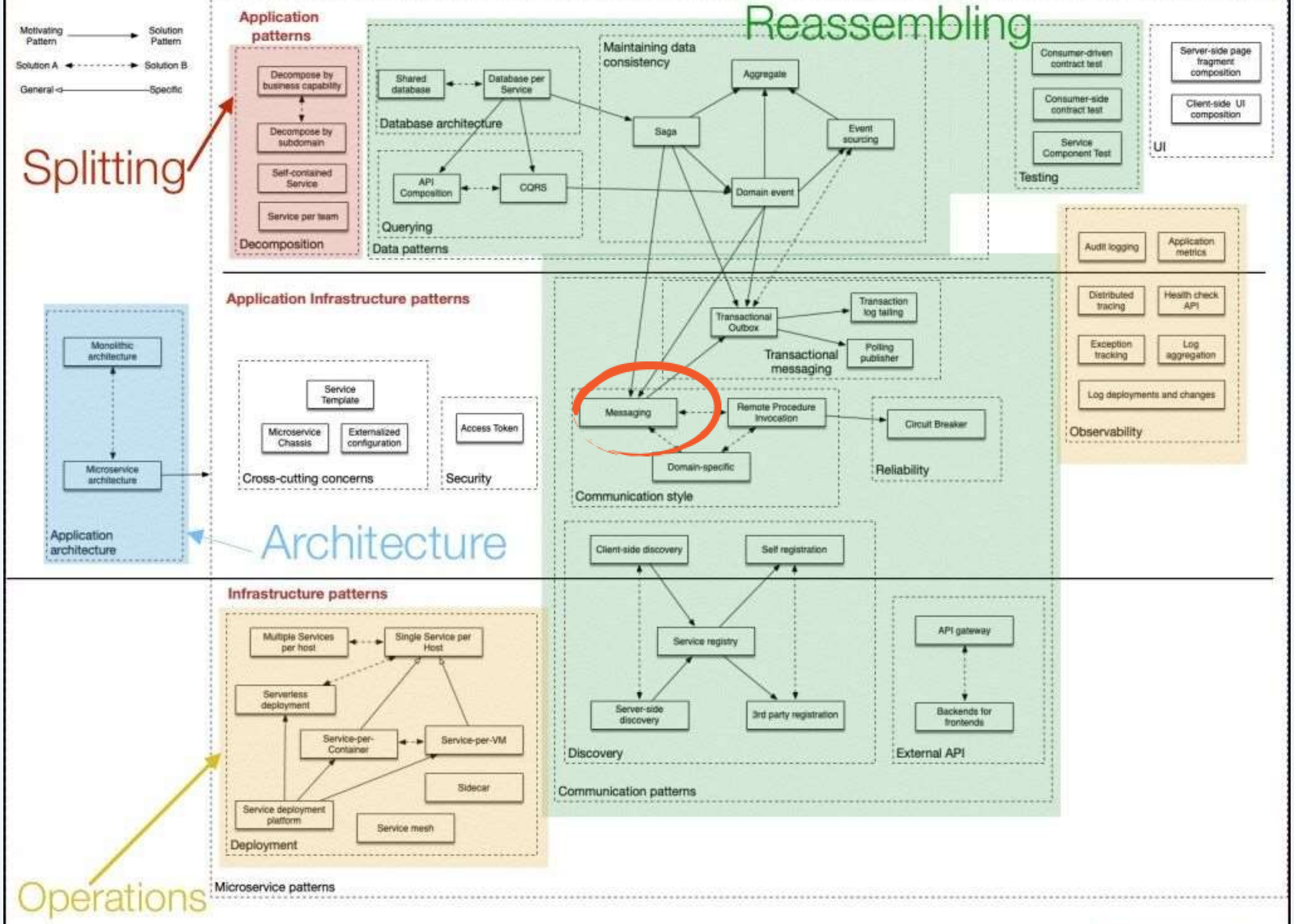


# Sharding

- A common solution (Apache Kafka uses this) is *sharding*
- The receiving service is scaled by partitioning into shards
- The sender specified a shard key in the message header
- The broker assigns each shard to a single receiver, so one receiver handles all the messages with the same shard key
- The broker reassigns shards when the receivers come up or go down.

# Messaging Design Issues – Duplicate Messages

- Most message brokers guarantee only to deliver each message *at least* once.
- This means with any given message there is a chance it may be delivered multiple times.
- This raises the issue of idempotency
  - Multiple applications of the same action do not result in changes beyond the first application.
  - E.g. deleteOrder(id=get_rid_of_this_one) is idempotent because the order id will be unique, so it will only be deleted once.
- But many actions are not inherently idempotent.
  - E.g. adding a new order
- You must design your messaging system with this in mind!



Motivating Pattern
Solution Pattern
Solution A
Solution B
General
Specific
Splitting
Application patterns
Decompose by business capability
Decompose by subdomain
Self-contained Service
Service per team
Decomposition
Reassembling
Shared database
Database per Service
Database architecture
API Composition
CQRS
Querying
Data patterns
Maintaining data consistency
Aggregate
Saga
Event sourcing
Domain event
Consumer-driven contract test
Consumer-side contract test
Service Component Test
Testing
Server-side page fragment composition
Client-side UI composition
UI
Application Infrastructure patterns
Monolithic architecture
Microservice architecture
Application architecture
Service Template
Microservice Chassis
Externalized configuration
Cross-cutting concerns
Access Token
Security
Transactional Outbox
Transaction log tailing
Polling publisher
Transactional messaging
Messaging
Remote Procedure Invocation
Domain-specific
Communication style
Circuit Breaker
Reliability
Audit logging
Application metrics
Distributed tracing
Health check API
Exception tracking
Log aggregation
Log deployments and changes
Observability
Architecture
Infrastructure patterns
Multiple Services per host
Single Service per Host
Serverless deployment
Service-per-Container
Service-per-VM
Sidecar
Service deployment platform
Service mesh
Deployment
Client-side discovery
Self registration
Service registry
Server-side discovery
3rd party registration
Discovery
API gateway
Backends for frontends
External API
Communication patterns
Operations
Microservice patterns

# Summary

- Communication between services
- Messaging Pattern:
  - Context & Problem
  - Forces
  - Solutions
  - Resulting Context
  - Issues
- Message structure, messaging architecture, issues

# Questions or Comments?

Owerya RESOURCING