Practical Messaging

A 101 guide to messaging

Ian Cooper
X, BlueSky and Hachyderm: ICooper

Who are you?

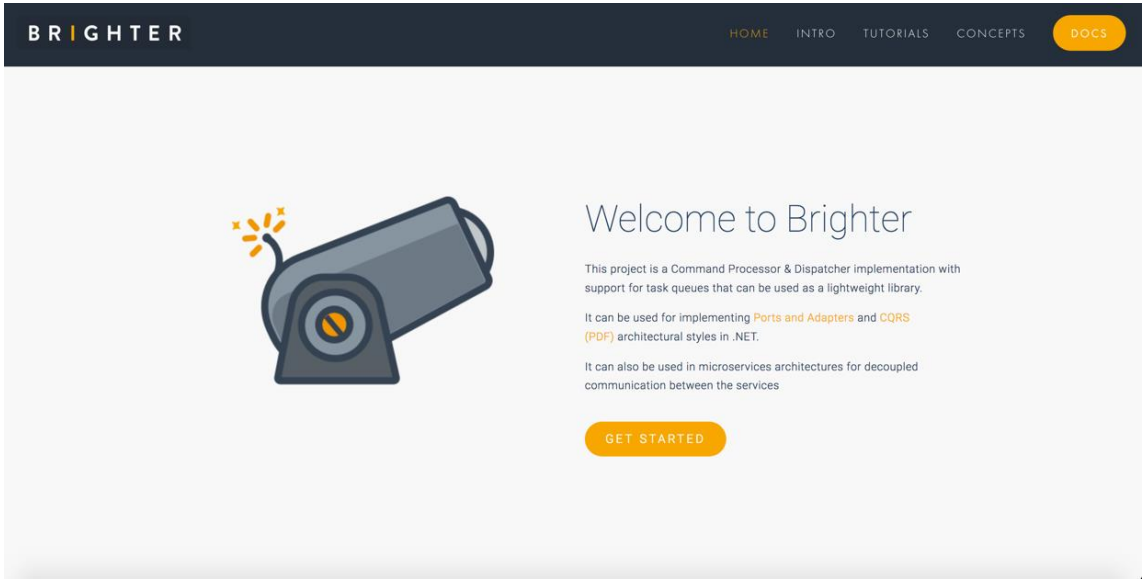
www.linkedin.com/in/ian-cooper-2b059b

I am a polyglot coding architect with over 20 years of experience delivering solutions in government, healthcare, and finance and ecommerce. During that time I have worked for the DTI, Reuters, Sungard, Misys, Beazley, Huddle and Just Eat Takeaway delivering everything from bespoke enterprise solutions, 'shrink-wrapped' products for thousands of customers, to SaaS applications for hundreds of thousands of customers.

I am an experienced systems architect with a strong knowledge of OO, TDD/BDD, DDD, EDA, CQRS/ES, REST, Messaging, Design Patterns, Architectural Styles, ATAM, and Agile Engineering Practices

I am frequent contributor to OSS, and I am the owner of: https://github.com/BrighterCommand. I speak regularly at user groups and conferences around the world on architecture and software craftsmanship. I run public workshops teaching messaging, event-driven and reactive architectures.

I have a strong background in C#. I spent years in the C++ trenches. I dabble in Go, Java, JavaScript and Python.


BRIGHTER
HOME
INTRO
TUTORIALS
CONCEPTS
DOCS
Welcome to Brighter
This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.
It can be used for implementing Ports and Adapters and CQRS (PDF) architectural styles in .NET.
It can also be used in microservices architectures for decoupled communication between the services
GET STARTED

Day One Messaging

- Distribution
- Integration Styles
- Messaging Patterns
- Queues and Streams
- Managing Asynchronous Architectures

Day Two Conversations

Conversations

Flow

Process Automation

Prerequisites

We will use Rabbit MQ and Kafka for examples. You should have Docker (or an equivalent) installed on your machine, as exercises provide a Docker Compose file to spin up RMQ and Kafka.

You will need to be able to write code with an editor/IDE of your choice.

You can choose from: C#; Java; Python; Go; JavaScript

Course Content

https://github.com/iancooper/practical-messaging

Exercise Code

https://github.com/iancooper/Practical-Messaging-Sharp
https://github.com/iancooper/Practical-Messaging-Python
https://github.com/iancooper/Practical-Messaging-JavaScript
https://github.com/iancooper/Practical-Messaging-Go
https://github.com/iancooper/Practical-Messaging-Java

Day One

DISTRIBUTED SYSTEMS

What is driving messaging

Why Distribute?

Performance and Scalability

Availability

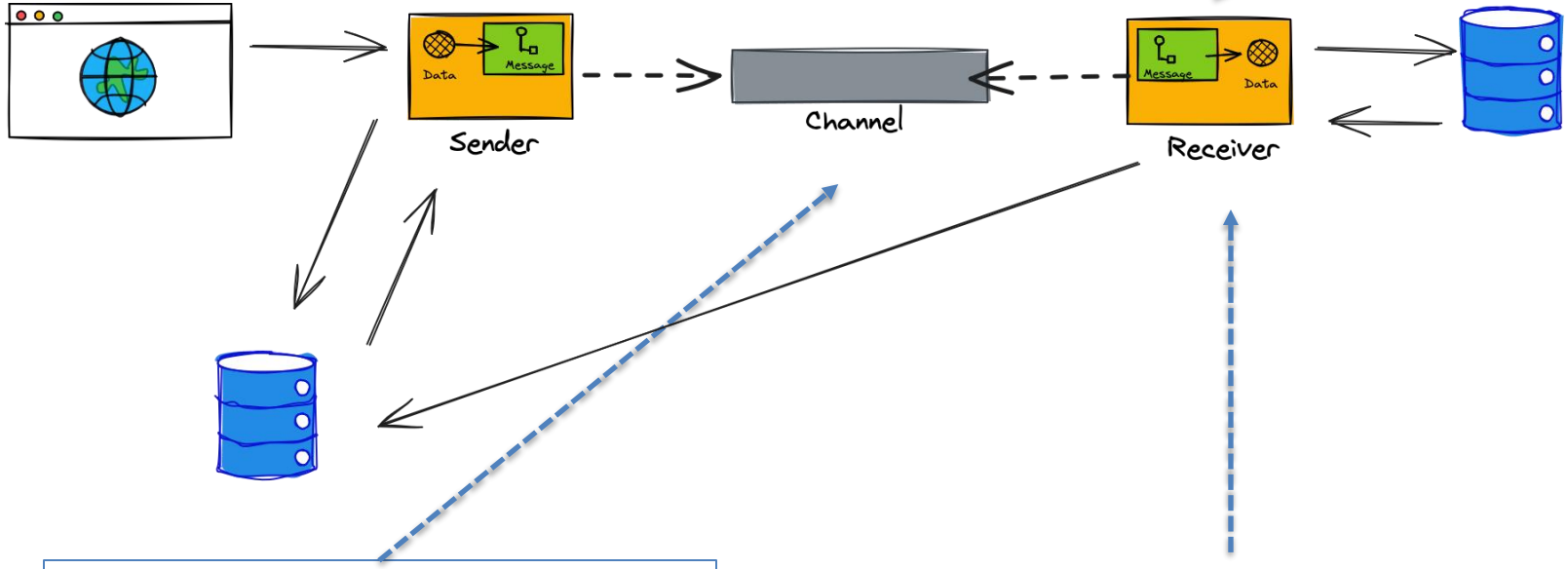
Maintainability

Inherent Distribution

Example: Task Queues

The web server puts the work on a queue

A backend application can perform long-running or CPU intensive work, allowing the web server to service new requests.



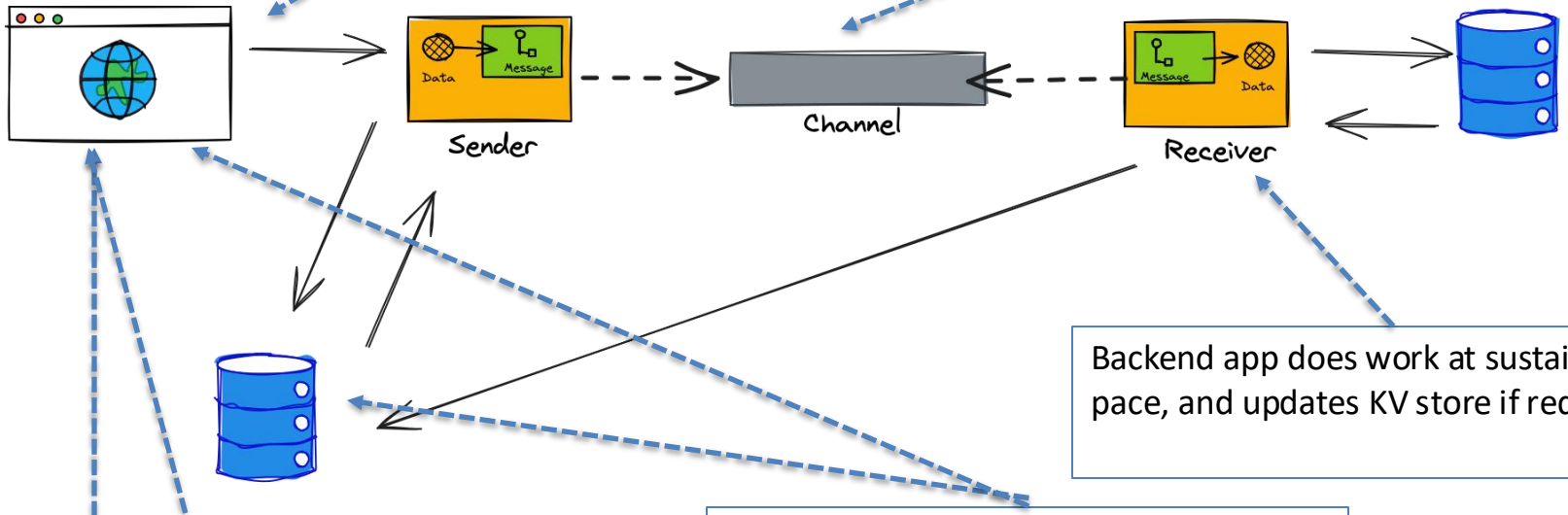
The queue stores work until we are ready to consume it. We can *throttle* to prevent surges.

We can scale out the backend services using a competing consumers approach, to ensure the queue does not backup.

Task Queue

We return 202 Accepted – we have your work request, and won't lost it.

We enqueue a work item for the request.



Backend app does work at sustainable pace, and updates KV store if required.

Link to a progress page, backed by a KV store where we note progress

We return a Location header, so you can monitor progress.

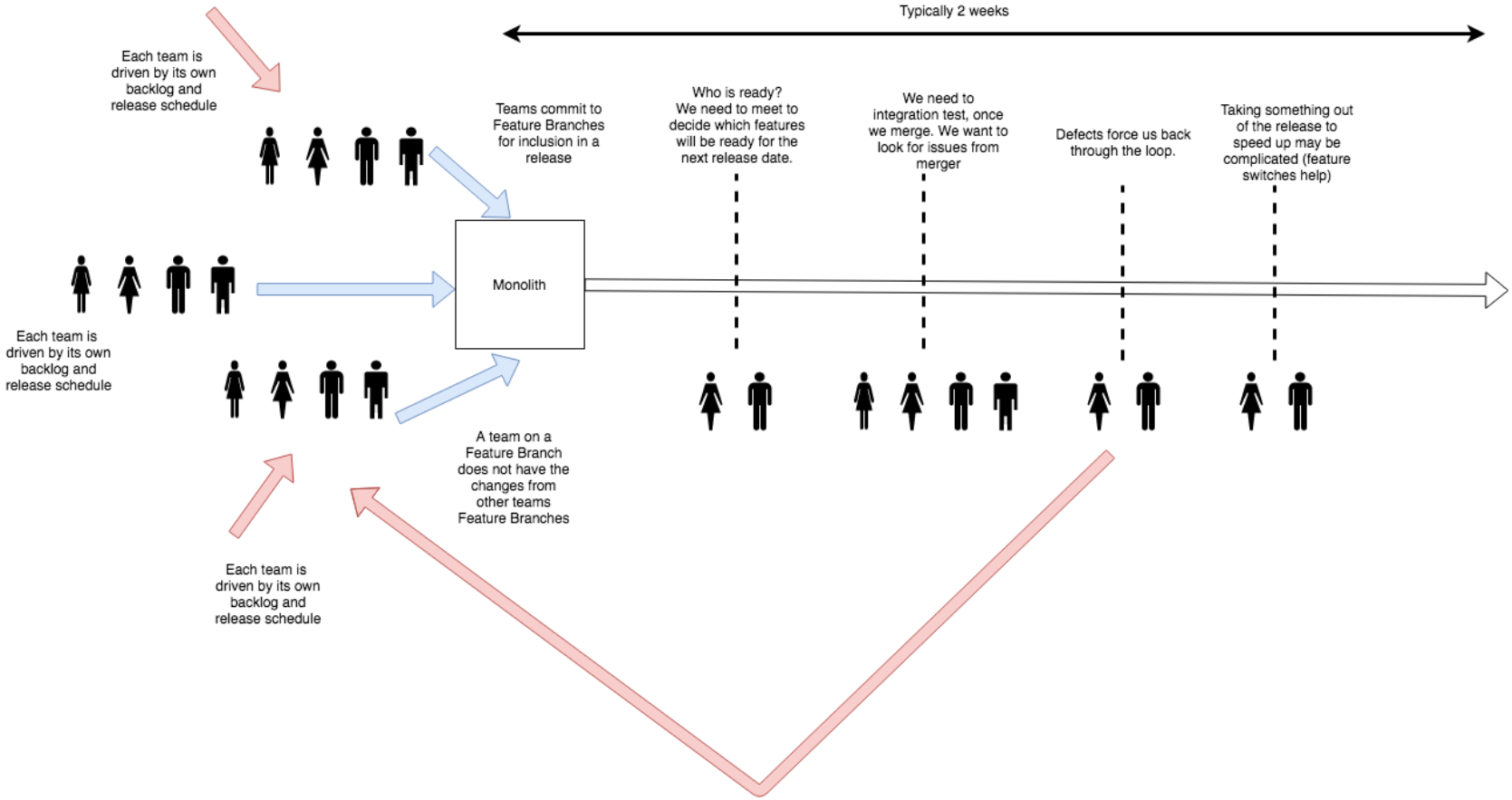
Link to Resource – 404 until created

Example: Microservices

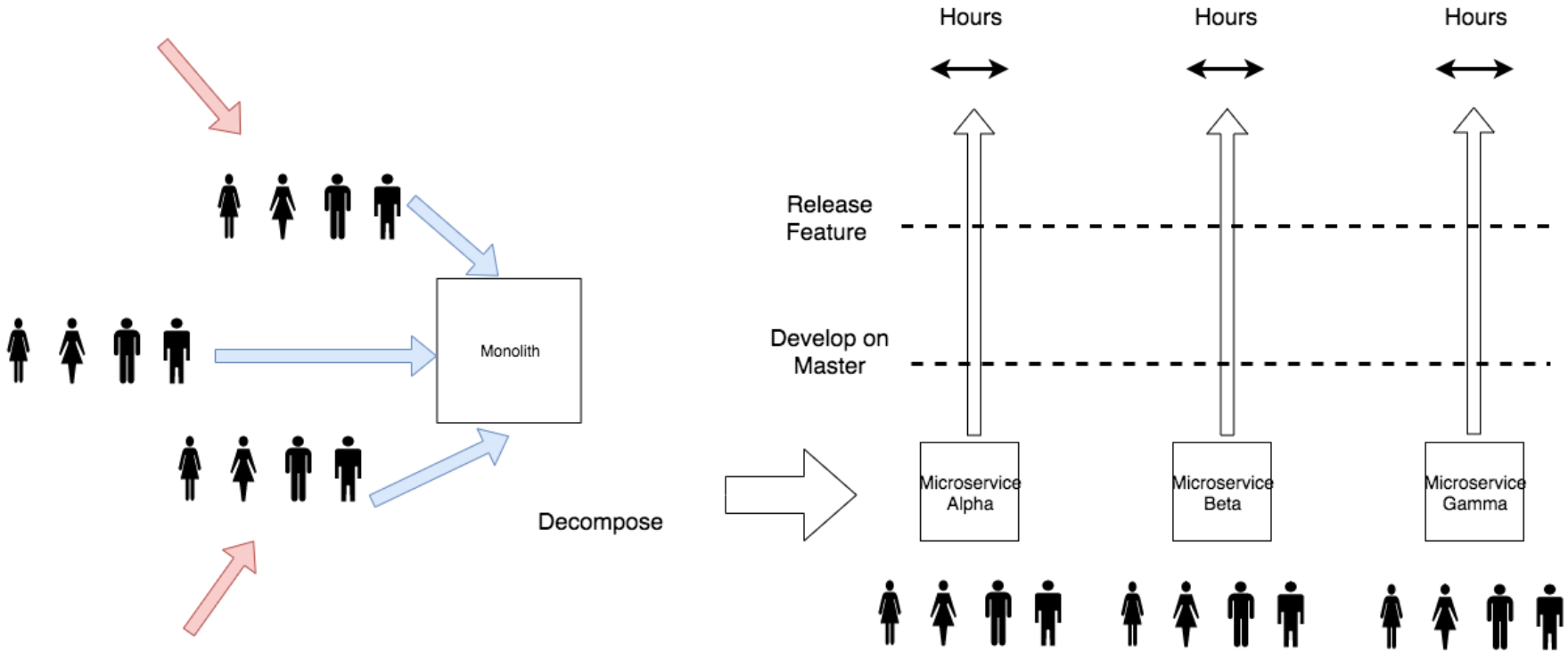
It’s all about velocity!!!

- “Speed wins in the marketplace”
    - Adrian Cockcroft, former lead architect at Netflix

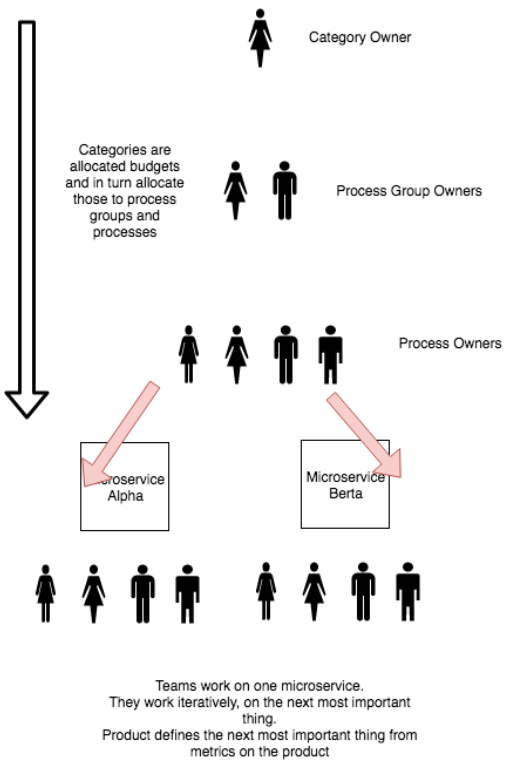
Monoliths Do Not Scale To Many Teams!



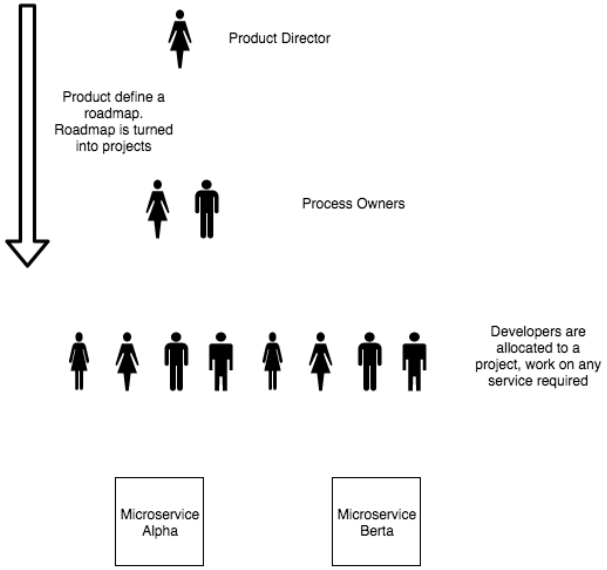
Microservices let us scale an organisation



Product Mode



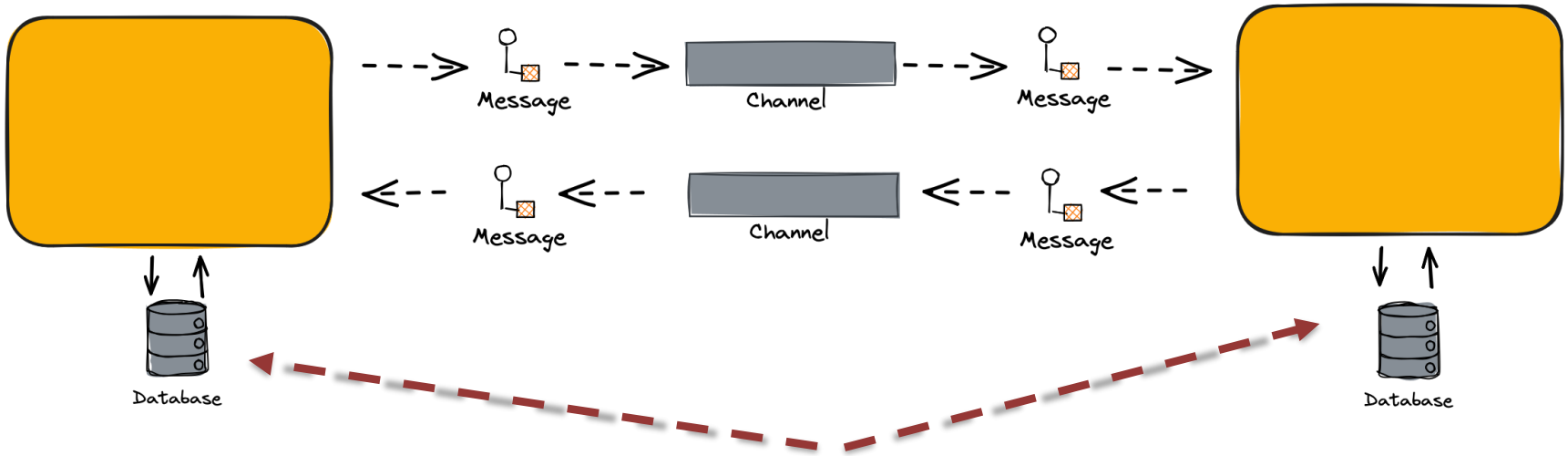
Product Mode



Projects

Microservice

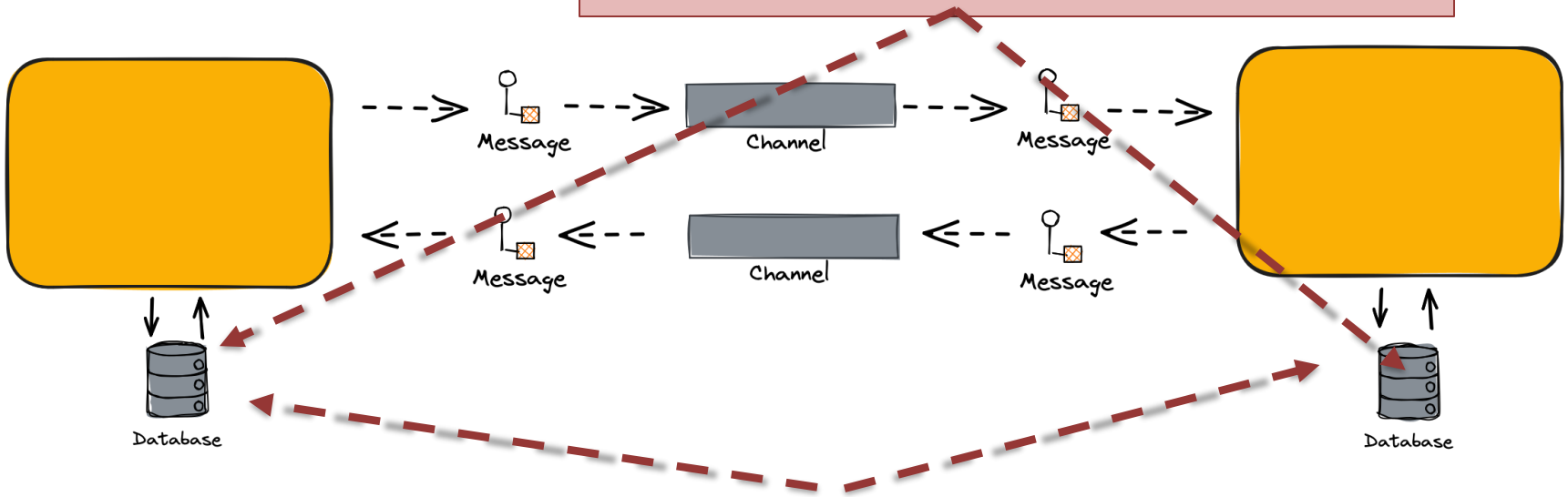
The only way to complete tasks within a service is by sending a message. Each service has its own accepted message types and specific data requirements for partners submitting work.



Encapsulated within the service is private data.
Requests to the service do not describe the shape of internal data

Microservice

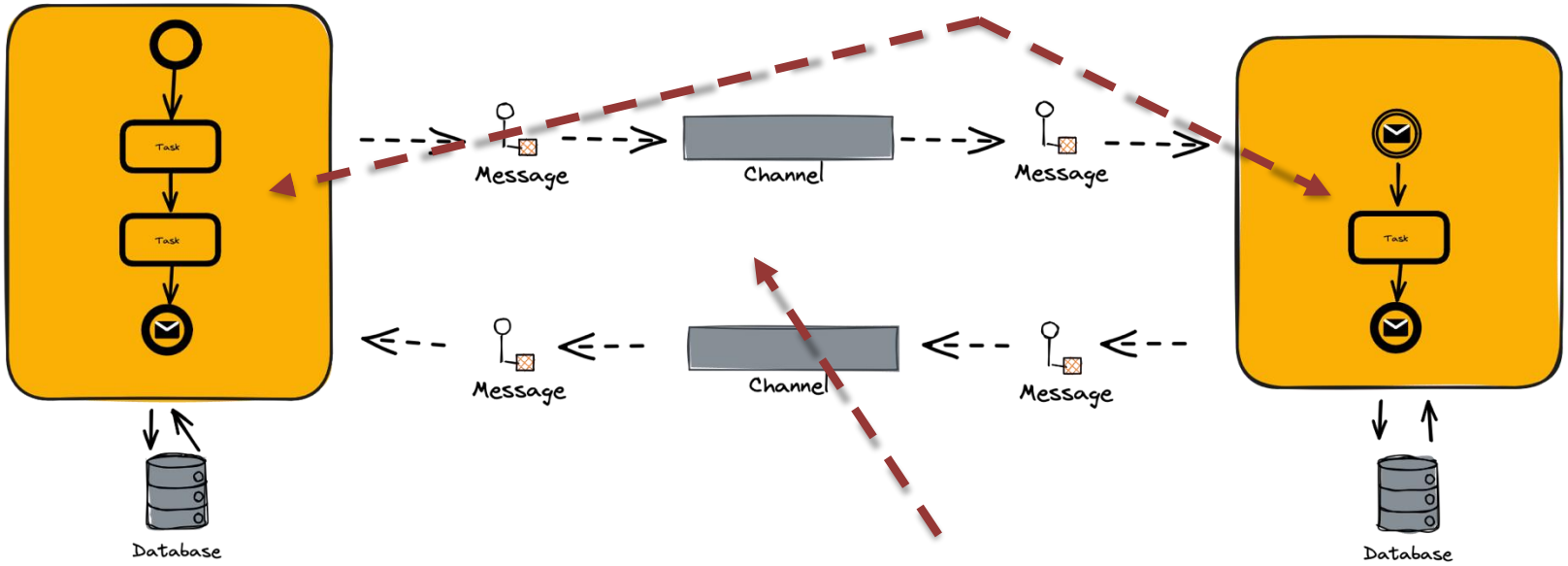
Transactions cannot occur between different services. If you operate independently from your business partners, you won't exchange transactions with them. Cross-organizational transactions are avoided to prevent potential lockup of YOUR database if the OTHER organization makes a mistake. Without transactions, you need to communicate through multiple messages over time.



Transactions, including 2PC, may occur within fiefdoms

Collaboration

As services are independent of each other, a collaboration comprises orchestrations (handlers, sagas, or workflows) within the services



A collaboration is also the flow of messages, the choreography, between our services

The Price of Distribution

Fallacies of Distributed Computing

The network is reliable.

Latency is zero.

Bandwidth is infinite.

The network is secure.

Topology doesn't change.

There is one administrator.

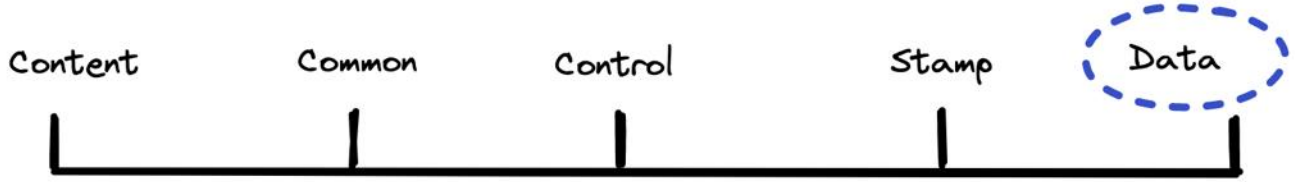
Transport cost is zero.

The network is homogeneous.

COUPLING

What architectural risks do we face from interoperability?

Coupling



Tight

More Interdependency

More Coordination

More Information Flow

Loose

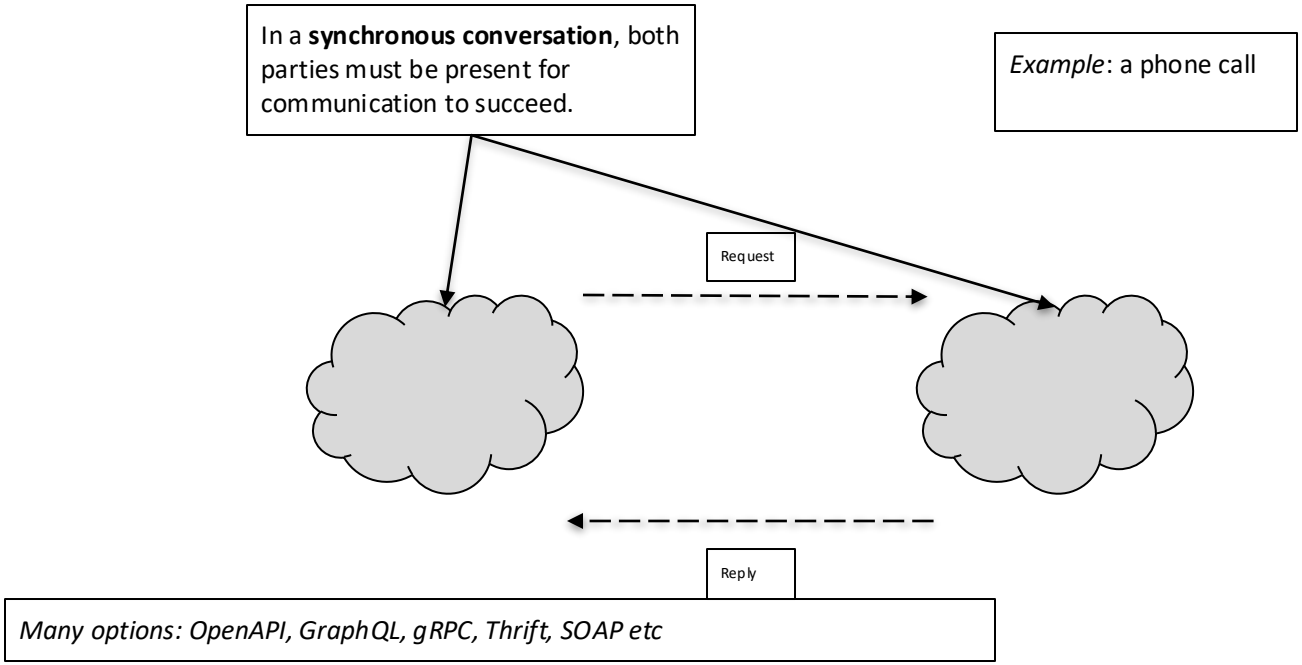
Less Interdependency

Less Coordination

Less Information Flow

Temporal Coupling

Synchronous Conversation

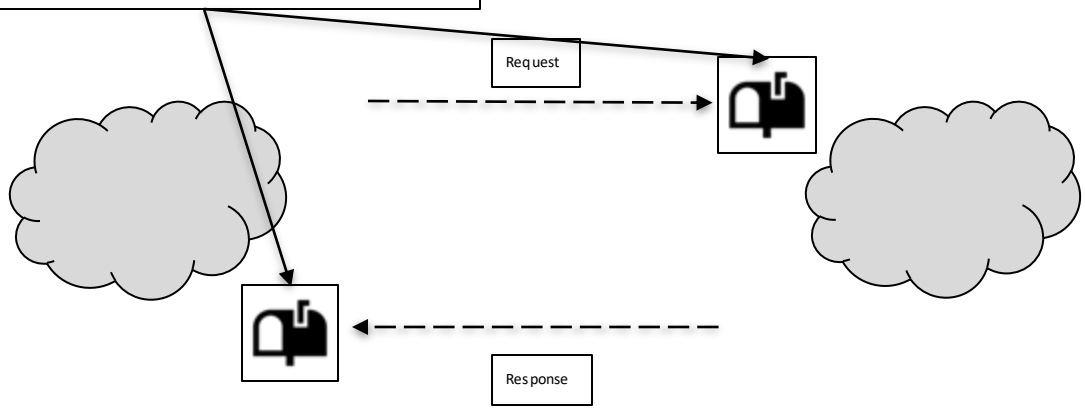


Temporal Coupling

Asynchronous Conversation

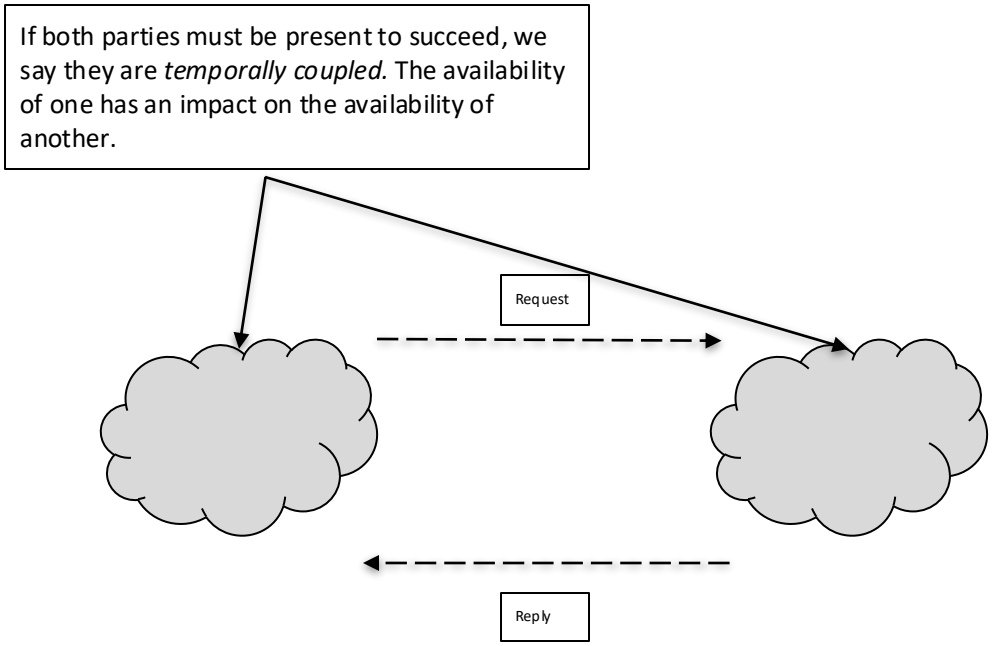
In an a**synchronous conversation**, the *receiver* does not need to be present at the time the *sender* communicates with them, using **store and forward** to pick up the message later.

Example: snail mail



Many options: SQS, Kafka, AMQP 0-9-1 (RMQ), AMQP 1-0, MQTT, S3

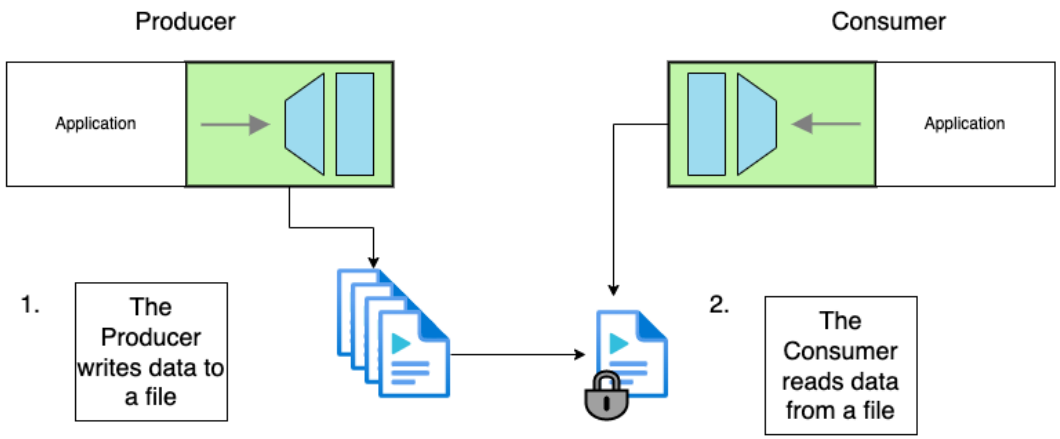
Temporal Coupling



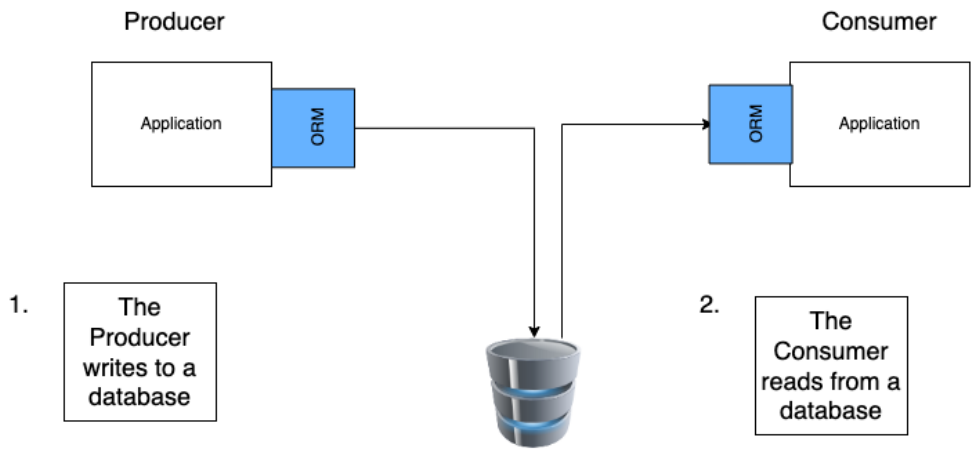
INTEGRATION STYLES

How do we communicate between microservices?

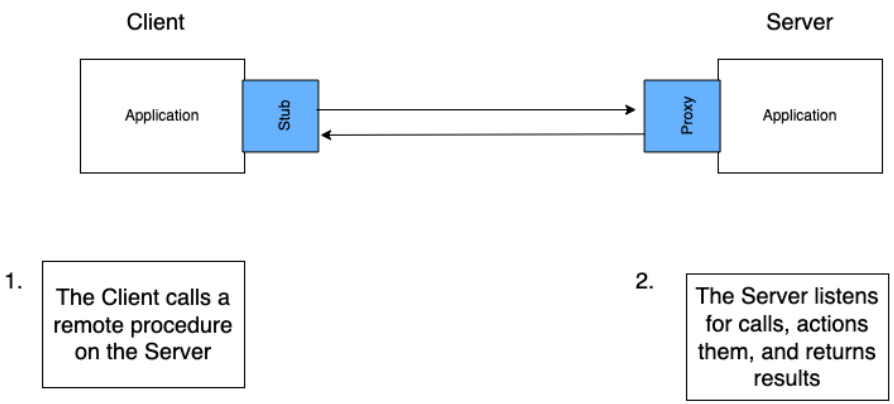
File Transfer



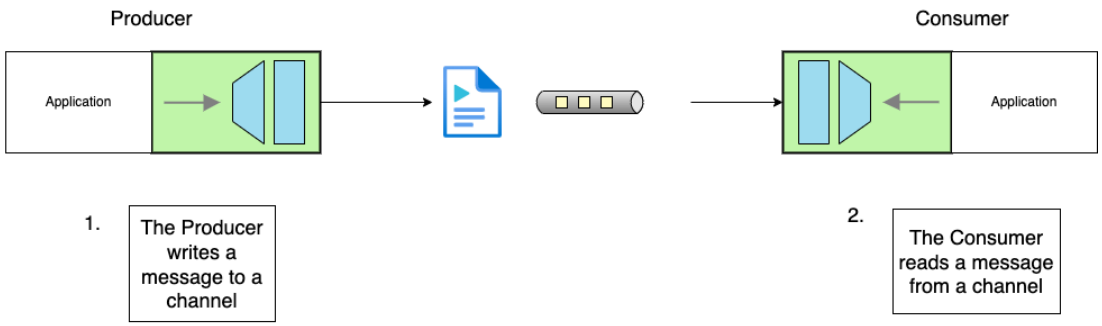
Shared Database



Remote Procedure Call

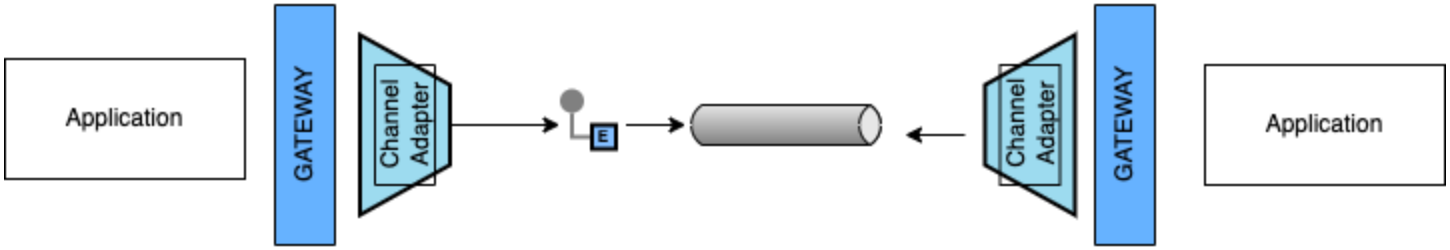


Messaging


Producer
Consumer
Application
Application
1.
The Producer writes a message to a channel
2.
The Consumer reads a message from a channel

MESSAGING PATTERNS

Integrating using events


Application
GATEWAY
Channel Adapter
E
Channel Adapter
GATEWAY
Application

A MESSAGE

What is a message?

Message Construction

A message has a header and body

The body contains data for the consumer

The header contains metadata for any *filter* in the pipeline.

The header should indicate the format of the body

Break a large message into pieces as a Message Sequence or use a Claim Check

MESSAGING AND EVENTS

Message Types

Messaging

Has Intent

Request An Answer
(Query)
Transfer of Control
(Command)
Transfer of Value

Part of a Workflow
Part of a Conversation

Concerned with
the Future

Eventing

Provides Facts

Things you Report On

No Expectations

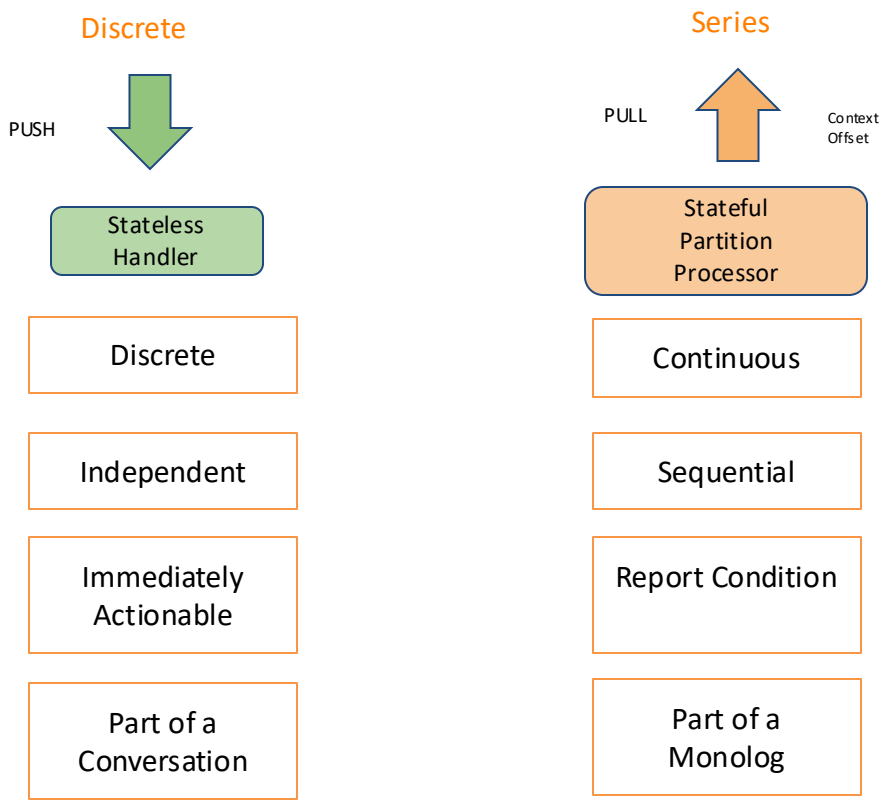
History

Context

Concerned with the Past

After Clemens Vasters https://youtu.be/ITrlLErsqzY

Eventing Types



After Clemens Vasters: https://skillsmatter.com/skillscasts/10191-keynote-events-data-points-jobs-and-commands-the-rise-of-messaging

See also: https://en.wikipedia.org/wiki/Discrete_time_and_continuous_time

Message Types

Messaging	Eventing
Command	Event (Notification)
	Document

See Gregor Hohpe: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Message.html

Command Message

Use a Command Message to reliably invoke a procedure in another application

Uses the well-established pattern for encapsulating a request as an object. The Command pattern [GoF] turns a request into an object that can be stored and passed around.

Document Message

Use a Document Message to reliably transfer a data structure between applications.

The receiver decides what, if anything, to do with the data

Event Message

Use an Event Message for reliable, asynchronous event notification between applications.

The difference between an Event Message and a Document Message is a matter of timing and content. An event's contents are typically less important.

EXERCISES

Self-paced material

EXERCISE MATERIAL

Introduction to Exercises

- Readme
- Videos
- Scripts & Slides

Introduction to RMQ

CHANNELS

Channels

A virtual pipe that connects producer and consumer

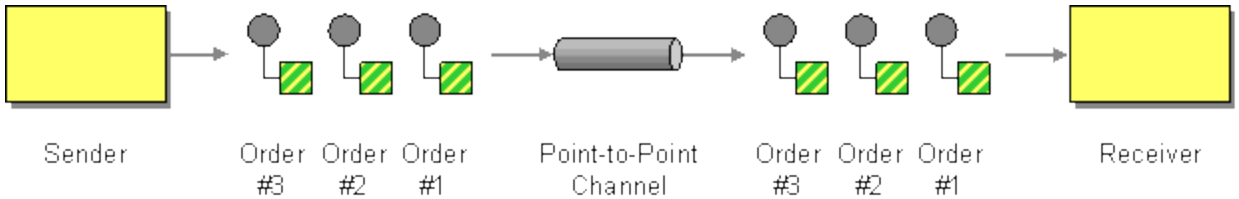
Logical Address (Topic or Routing Key)

Unidirectional

One-to-One or One-to-Many

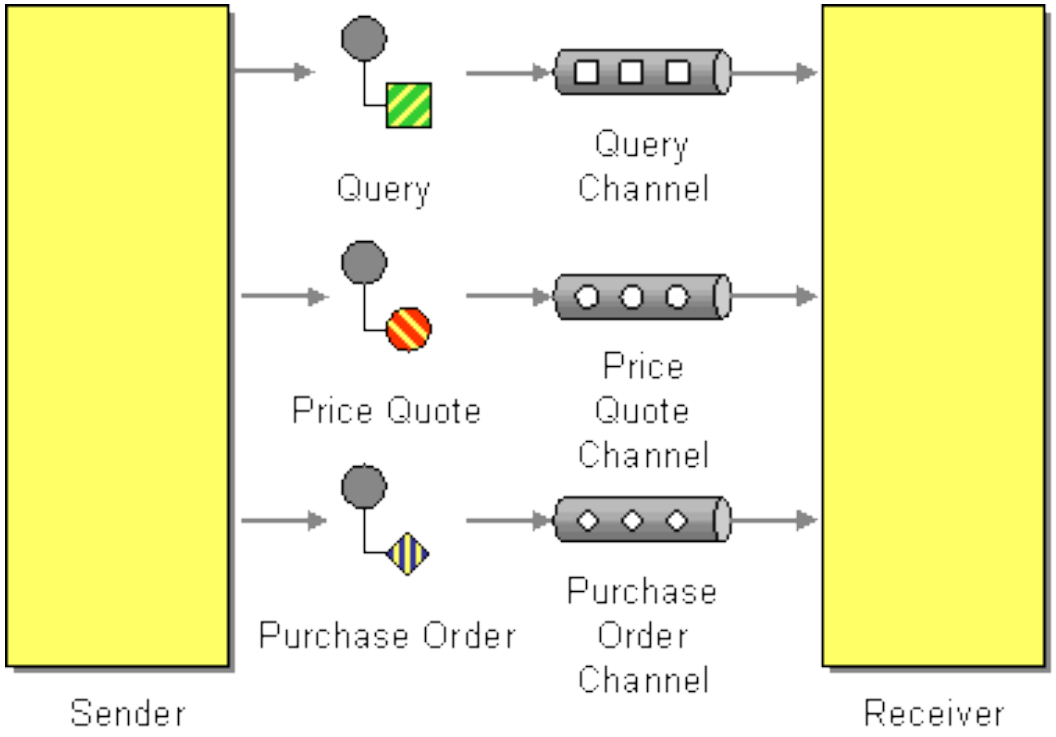
Messaging is a ‘pipe’ not a ‘bucket’.

Point-to-Point Channel



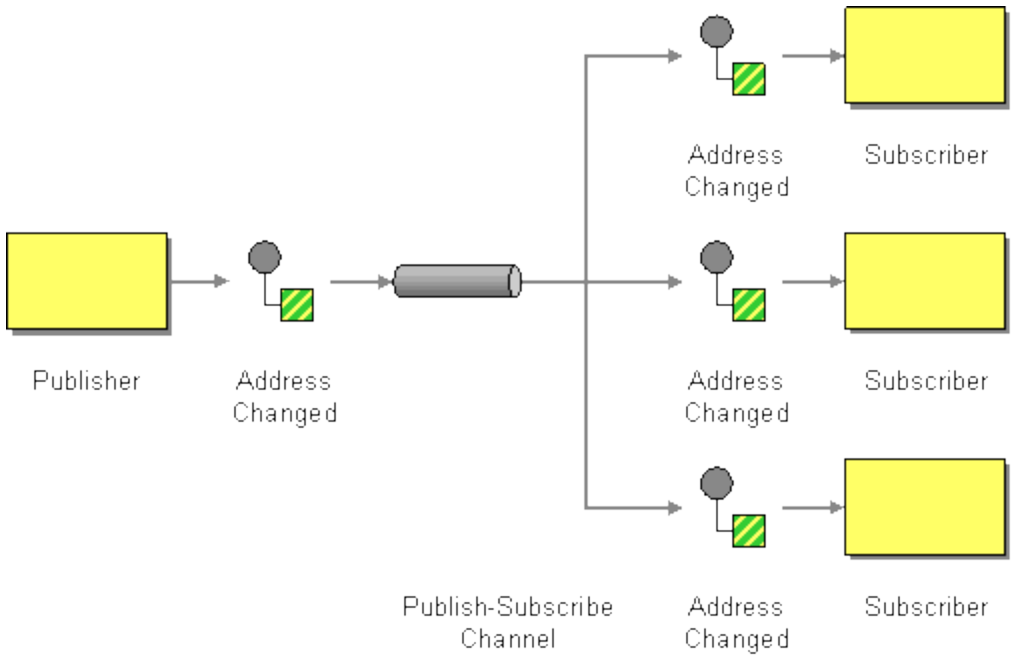
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html

Datatype Channel



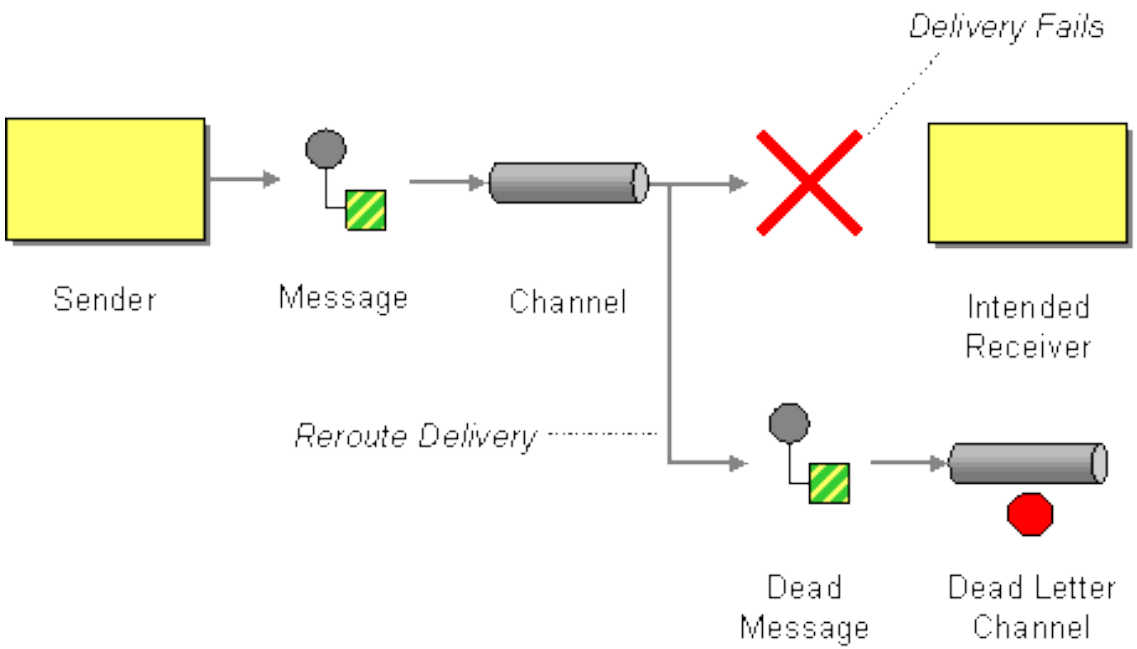
http://www.enterpriseintegrationpatterns.com/patterns/messaging/DatatypeChannel.html

Publish-Subscribe Channel



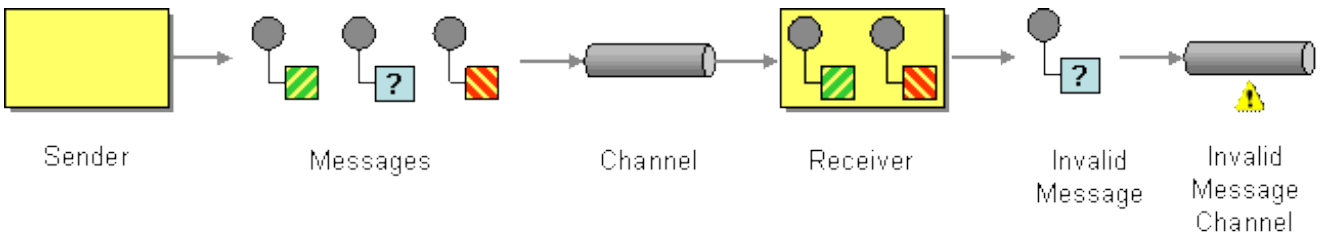
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html

Dead Letter Channel



http://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html

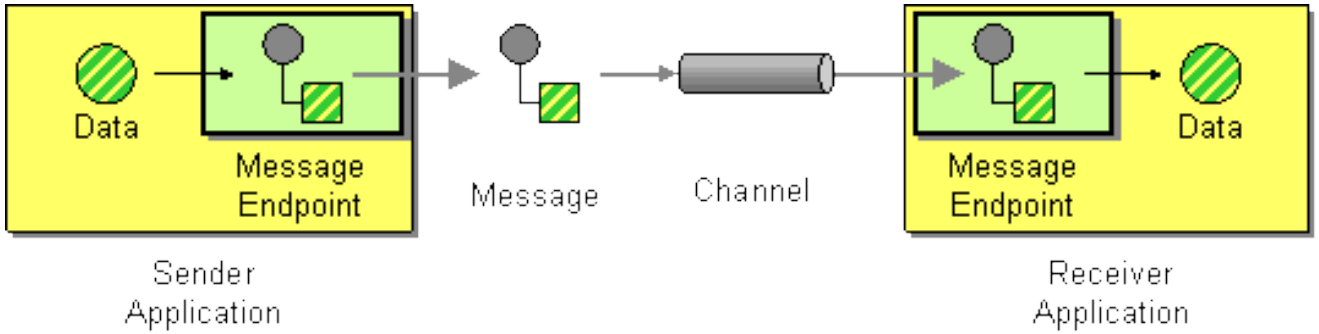
Invalid Message Channel



http://www.enterpriseintegrationpatterns.com/patterns/messaging/InvalidMessageChannel.html

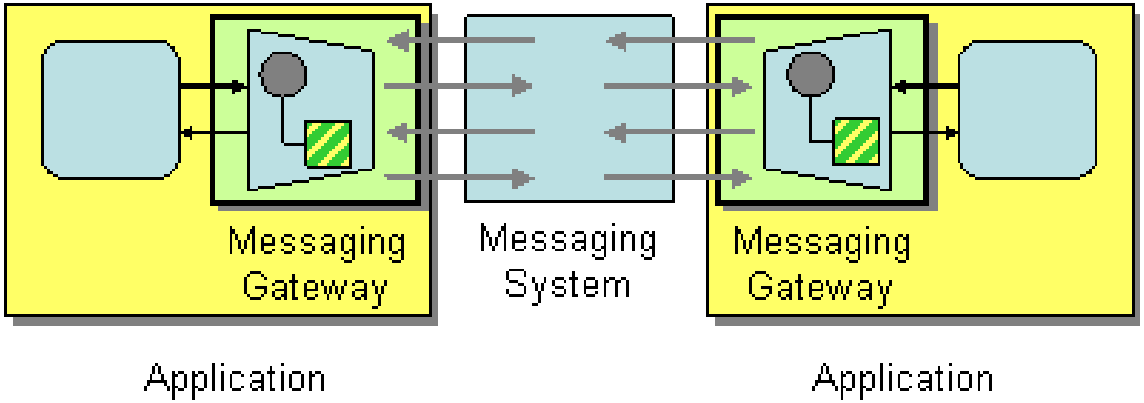
ENDPOINTS

Message Endpoint



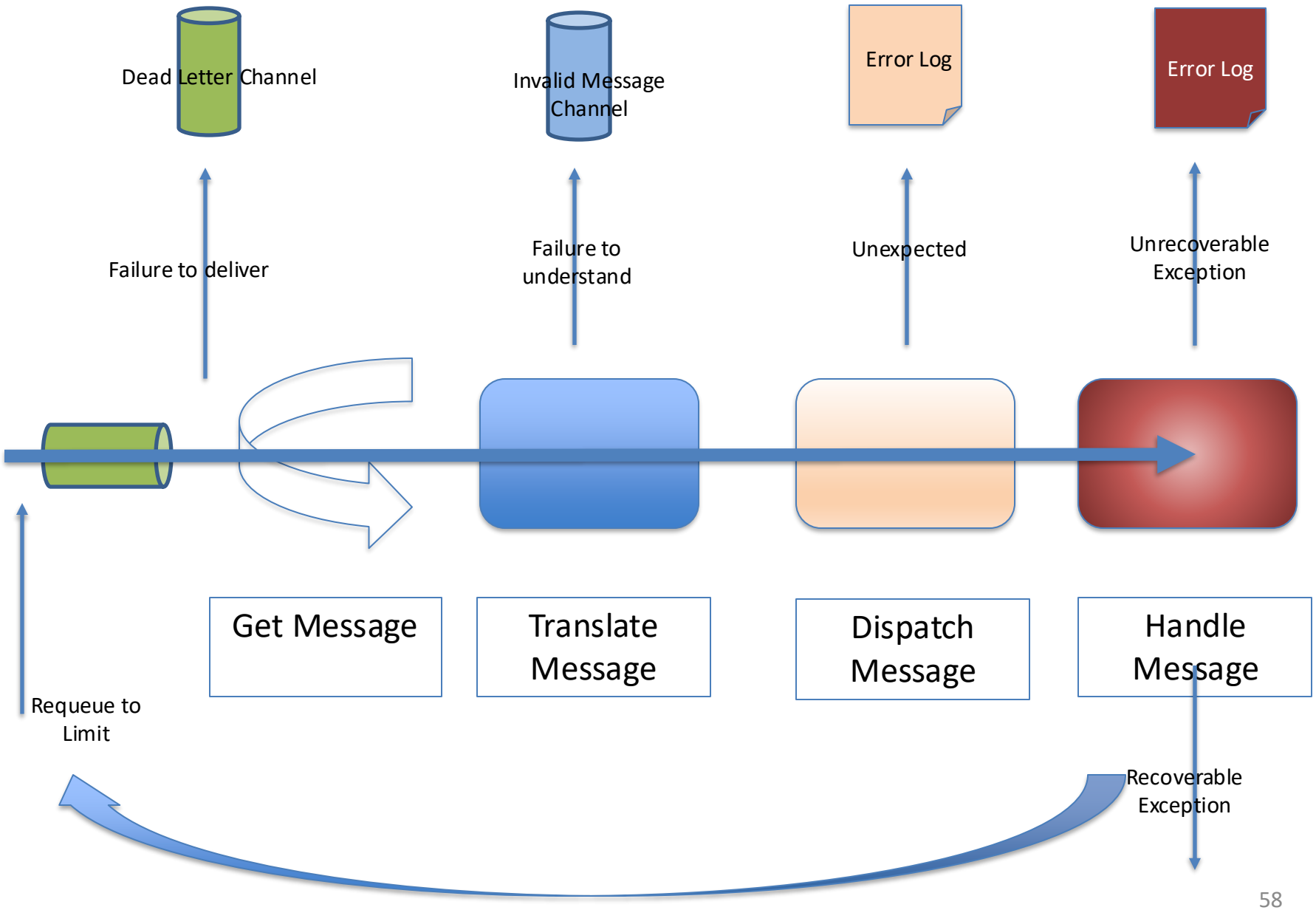
http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html

Messaging Gateway

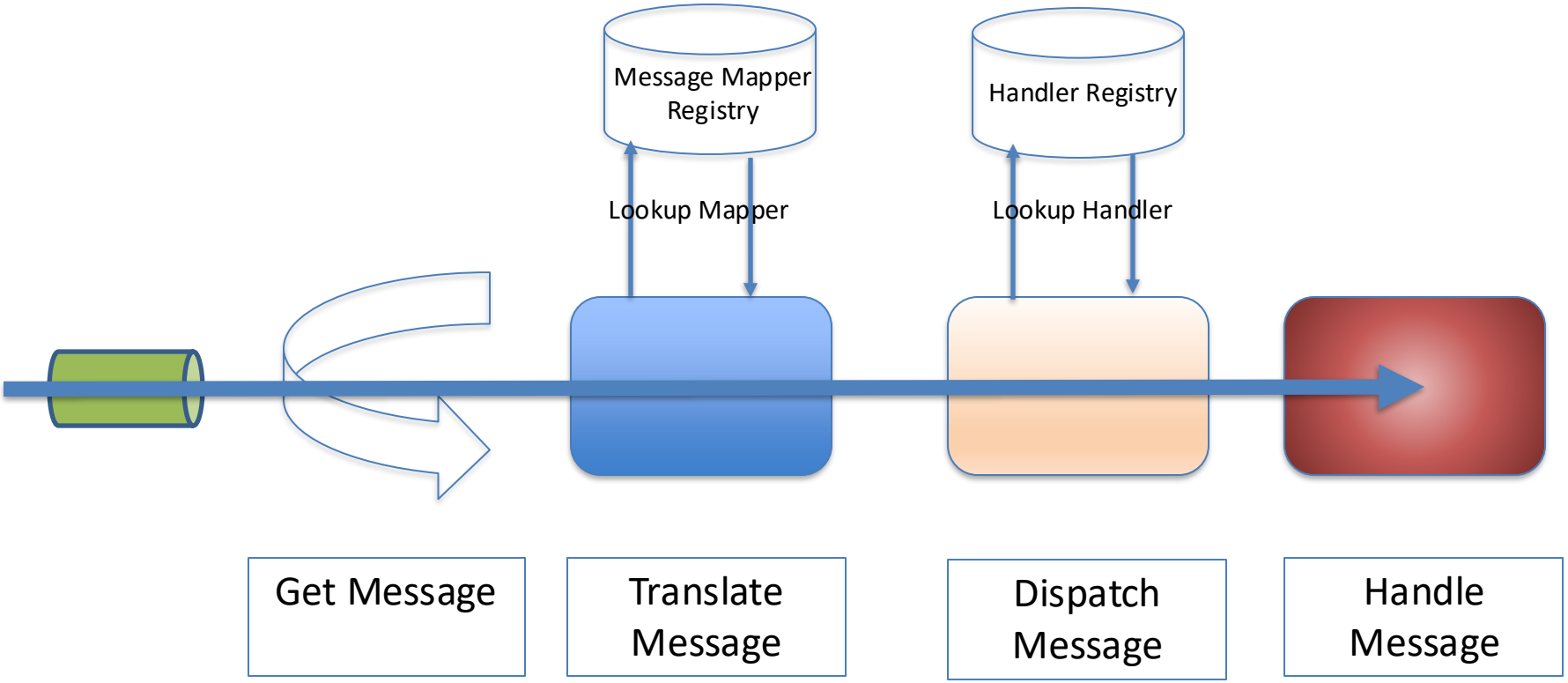


http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingGateway.html

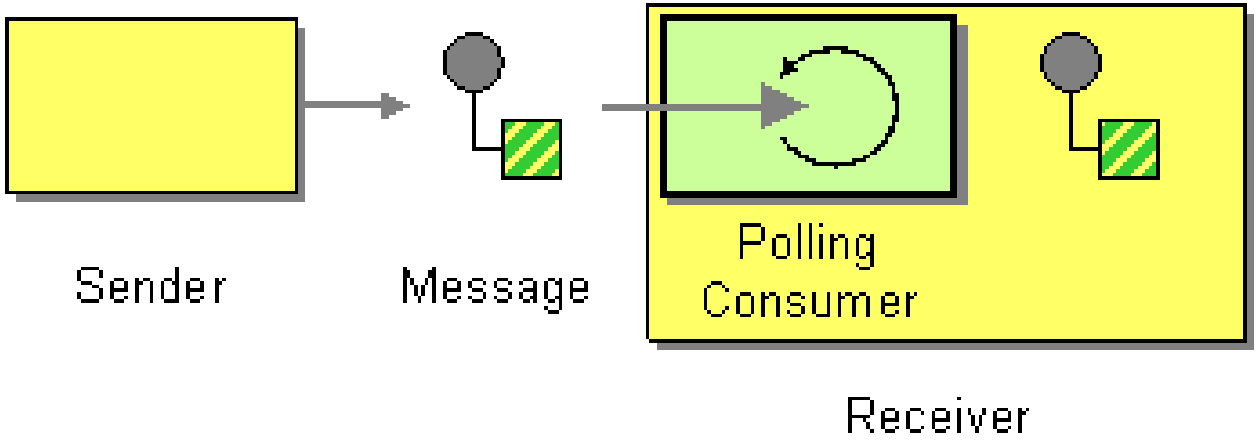
THE MESSAGE PUMP


Dead Letter Channel
Invalid Message Channel
Error Log
Error Log
Failure to deliver
Failure to understand
Unexpected
Unrecoverable Exception
Get Message
Translate Message
Dispatch Message
Handle Message
Requeue to Limit
Recoverable Exception

Translate and Dispatch

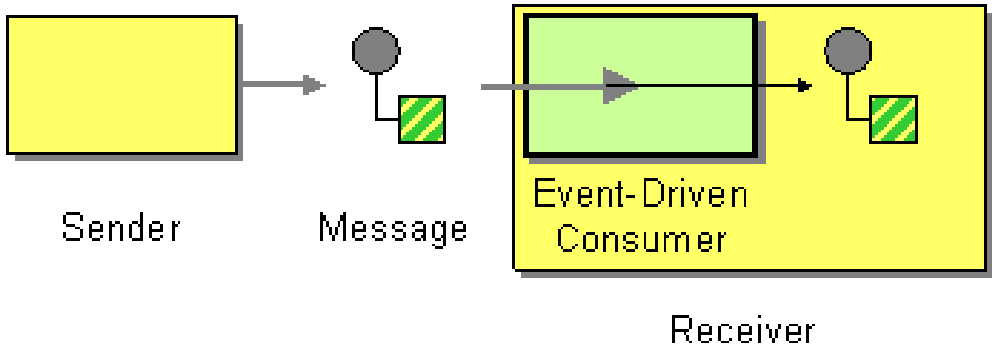


Polling Consumer



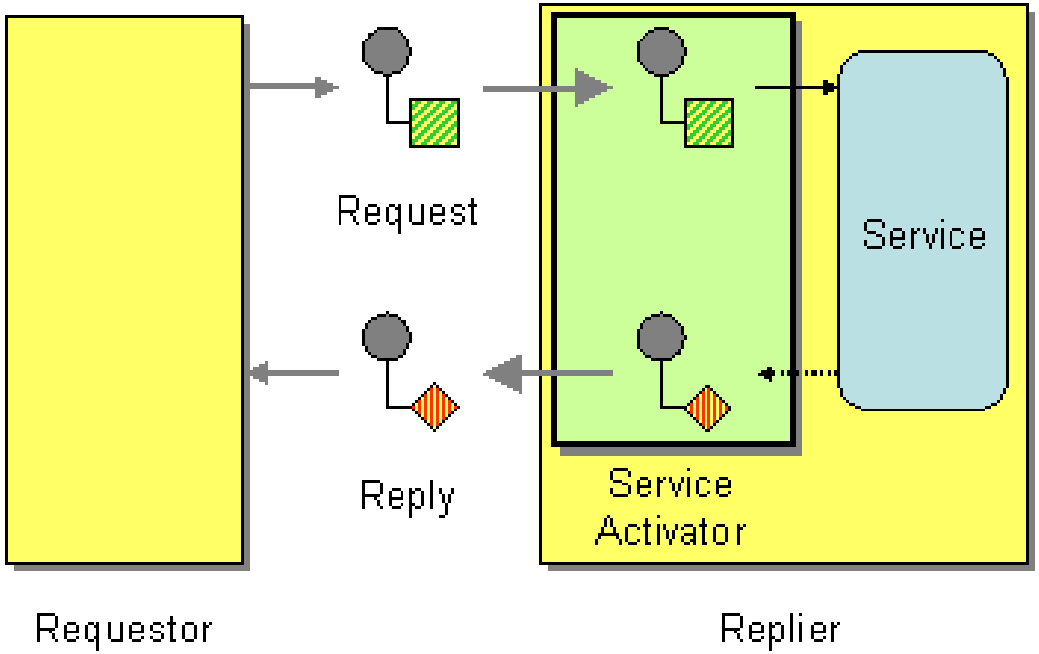
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PollingConsumer.html

Event Driven Consumer



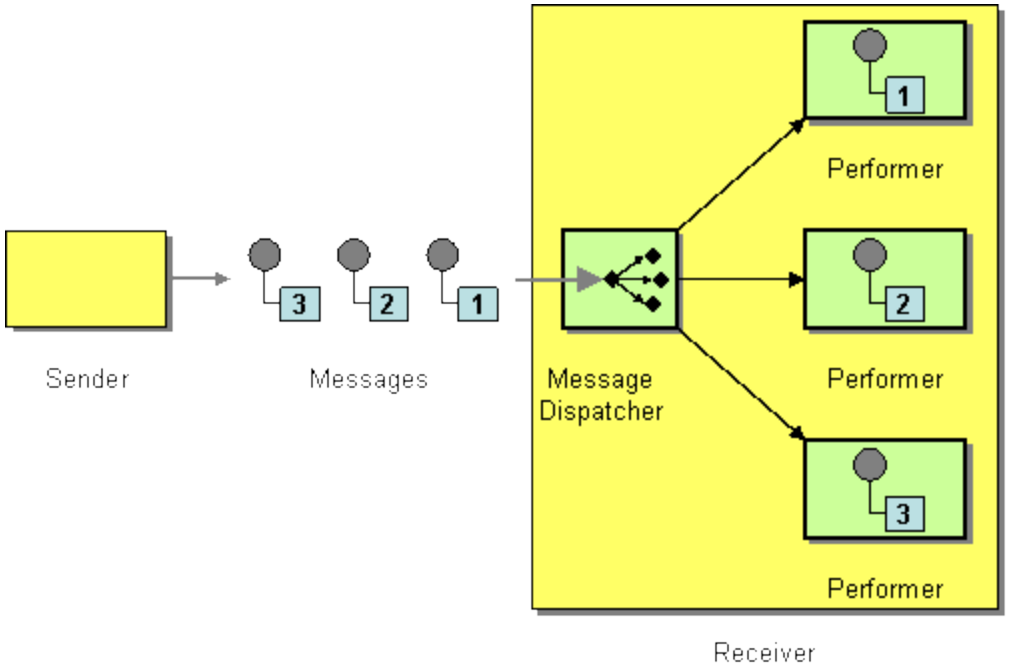
http://www.enterpriseintegrationpatterns.com/patterns/messaging/EventDrivenConsumer.html

Service Activator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingAdapter.html

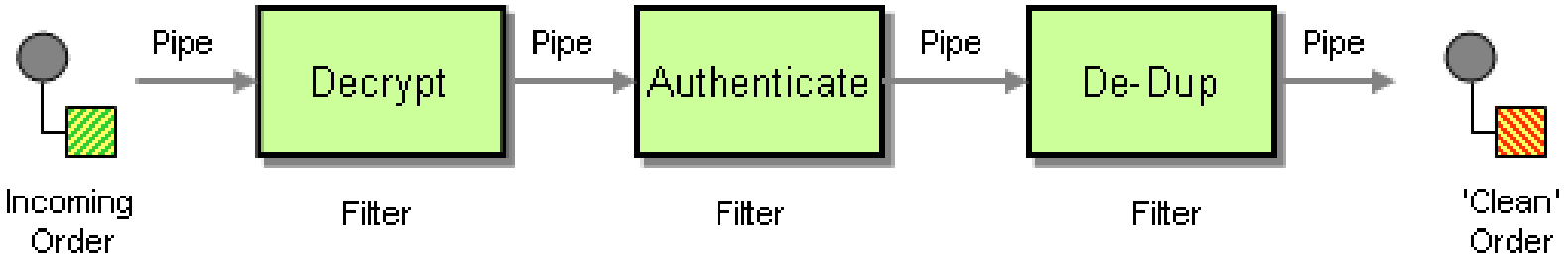
Competing Consumers



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageDispatcher.html

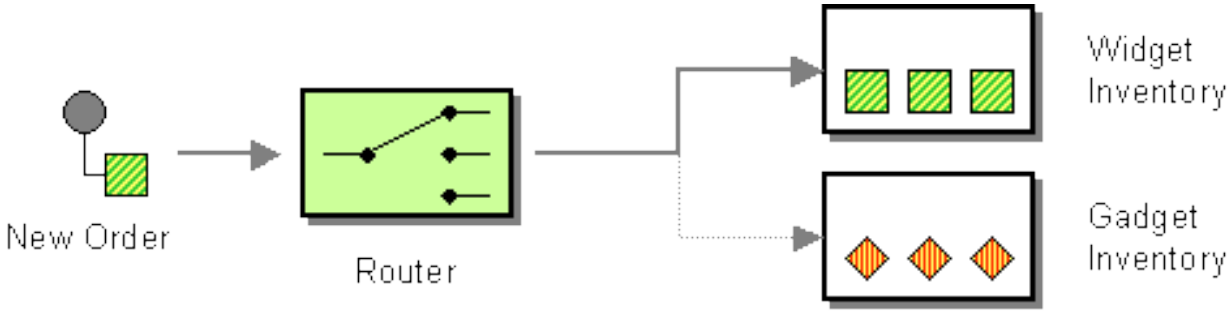
PIPELINES

Pipes and Filters



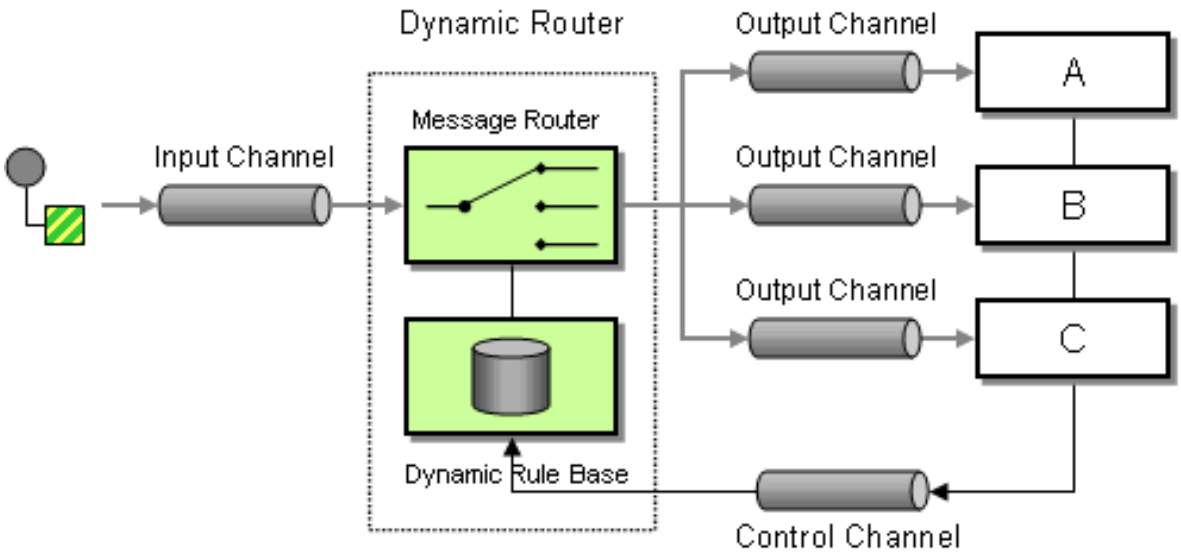
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html

Content Based Router



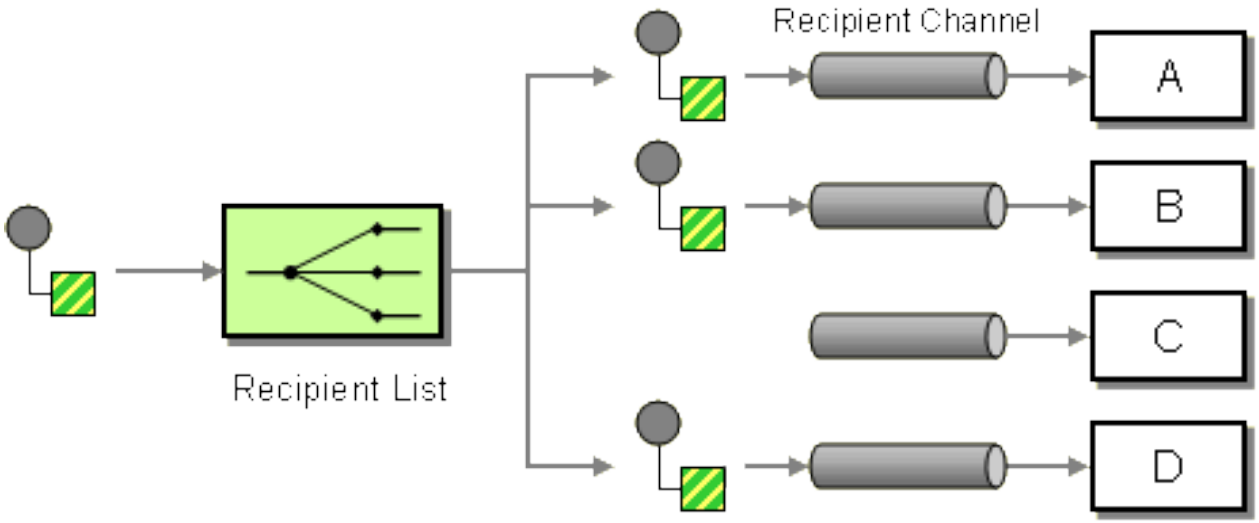
http://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html

Dynamic Router



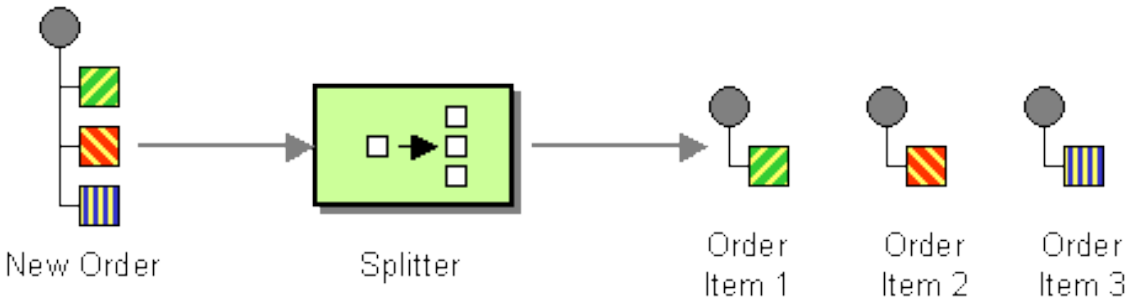
http://www.enterpriseintegrationpatterns.com/patterns/messaging/DynamicRouter.html

Recipient List



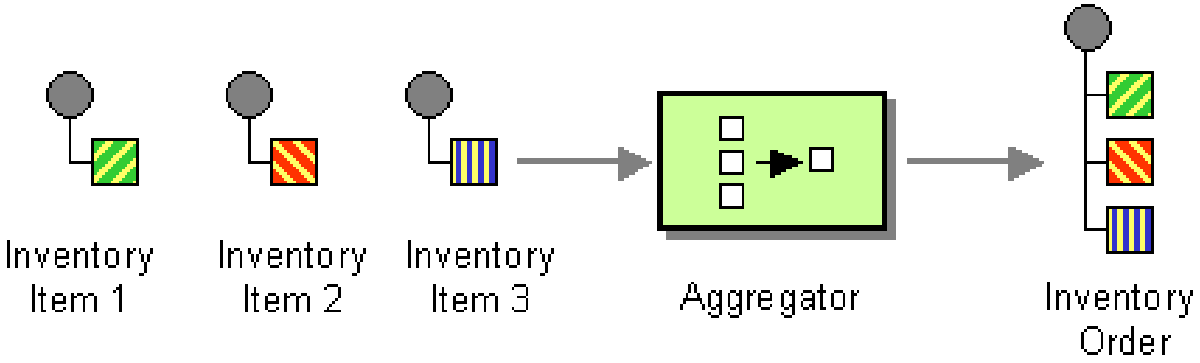
http://www.enterpriseintegrationpatterns.com/patterns/messaging/RecipientList.html

Splitter



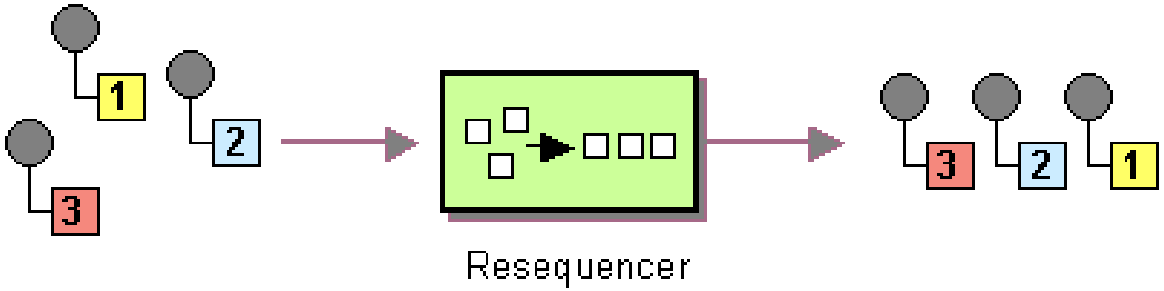
http://www.enterpriseintegrationpatterns.com/patterns/messaging/Sequencer.html

Aggregator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/Aggregator.html

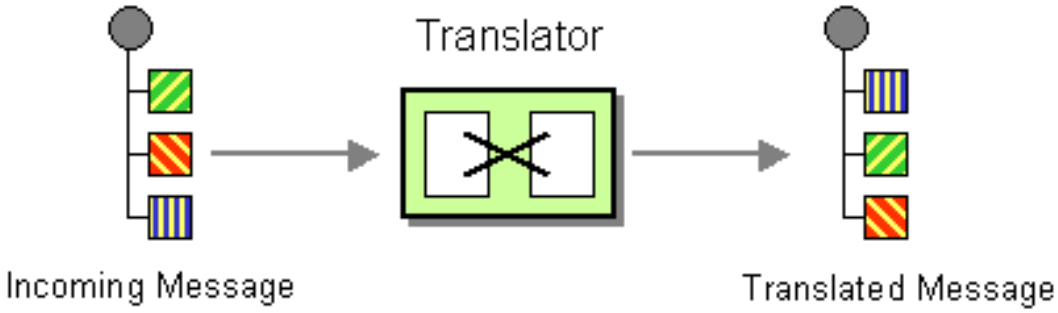
Resequencer



http://www.enterpriseintegrationpatterns.com/patterns/messaging/Resequencer.html

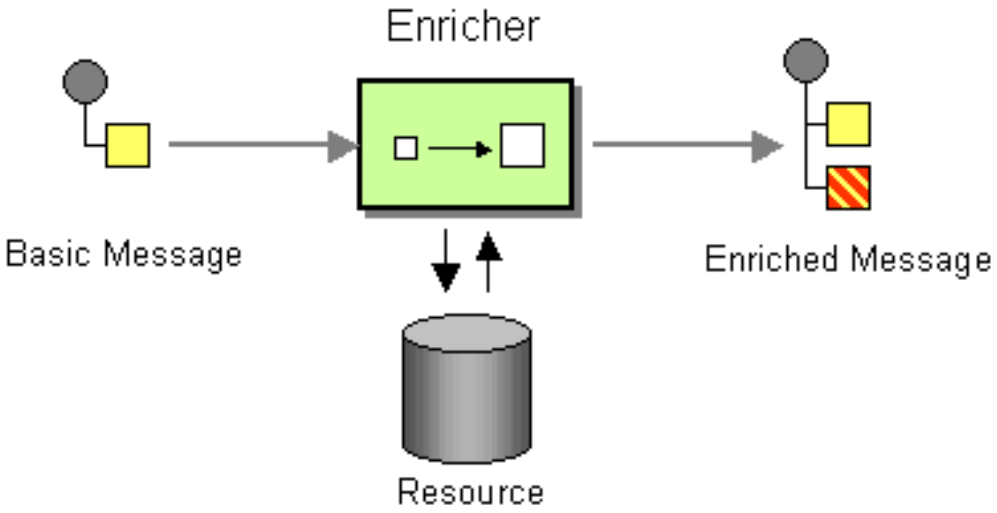
TRANSFORMATION

Message Translator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html

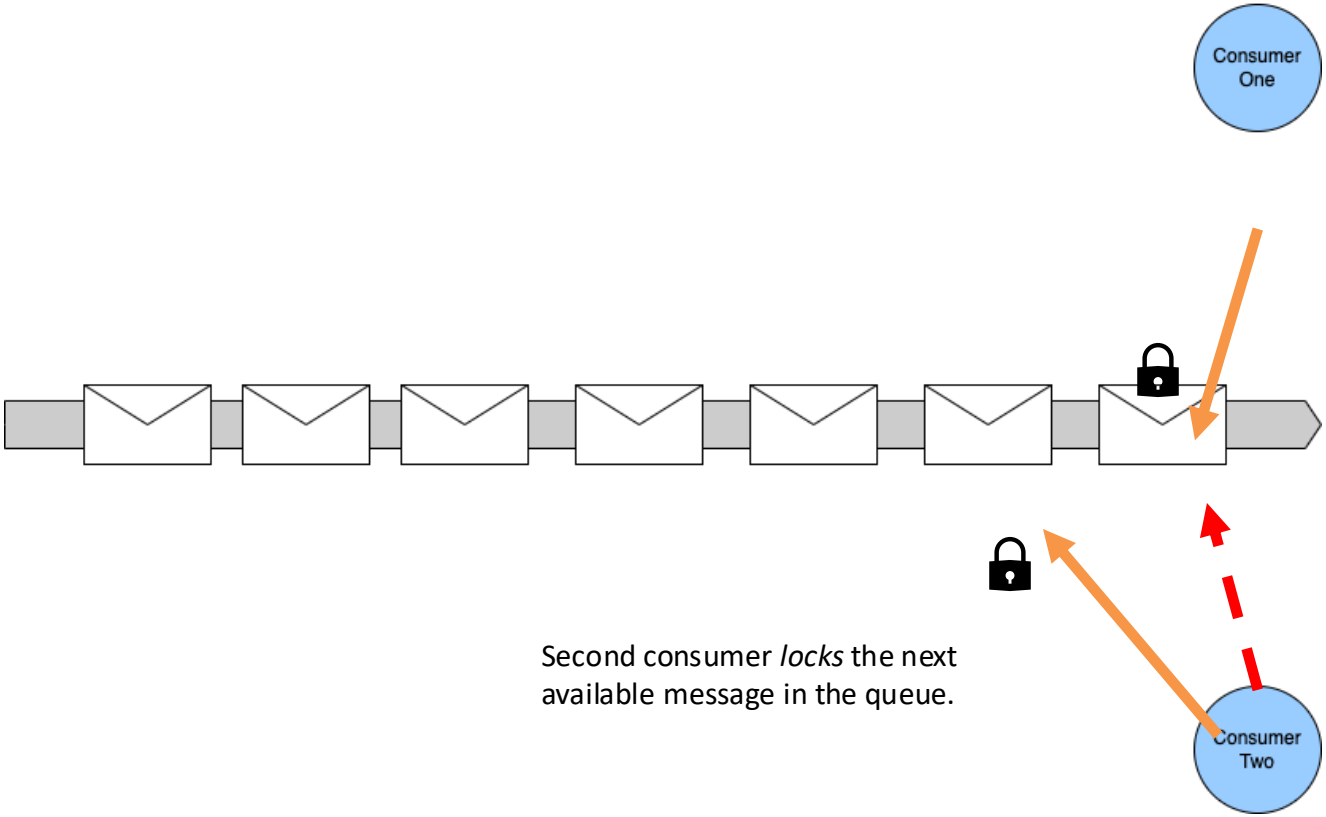
Content Enricher



http://www.enterpriseintegrationpatterns.com/patterns/messaging/DataEnricher.html

QUEUES AND STREAMS

Queues



First consumer *locks* the next message in the queue whilst it processes it.

Second consumer *locks* the next available message in the queue.

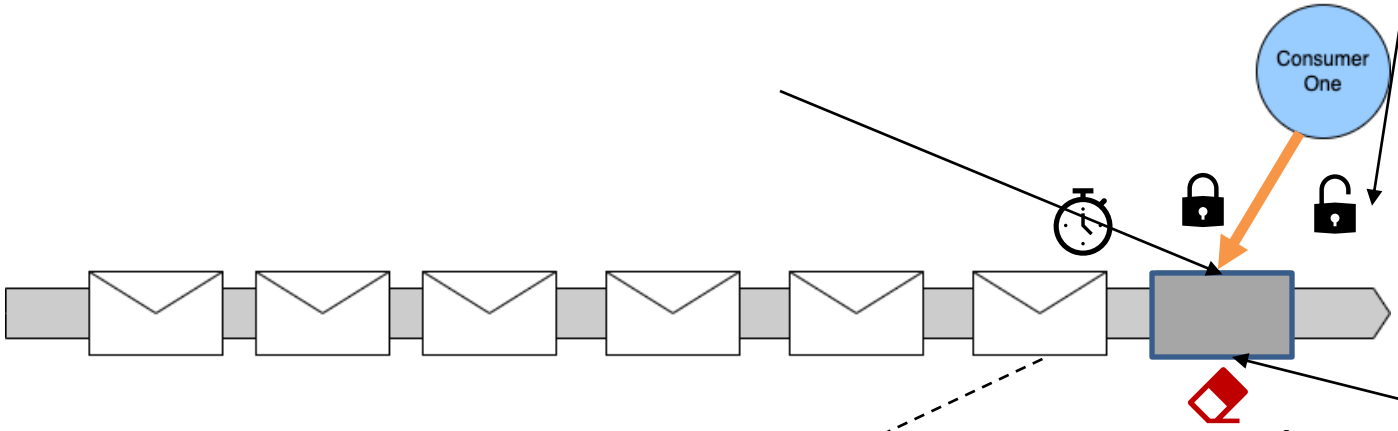
Second consumer *reads past* any locked message in the queue.

Queues

If we fail, then we may decide others could succeed later, so we let it become available to lock again, often with a delay.

When we are done processing, we unlock it.

- Usually because we finish processing it
- May be because we fail.



If we succeed, then we delete the message from the queue, no one else can now process it.

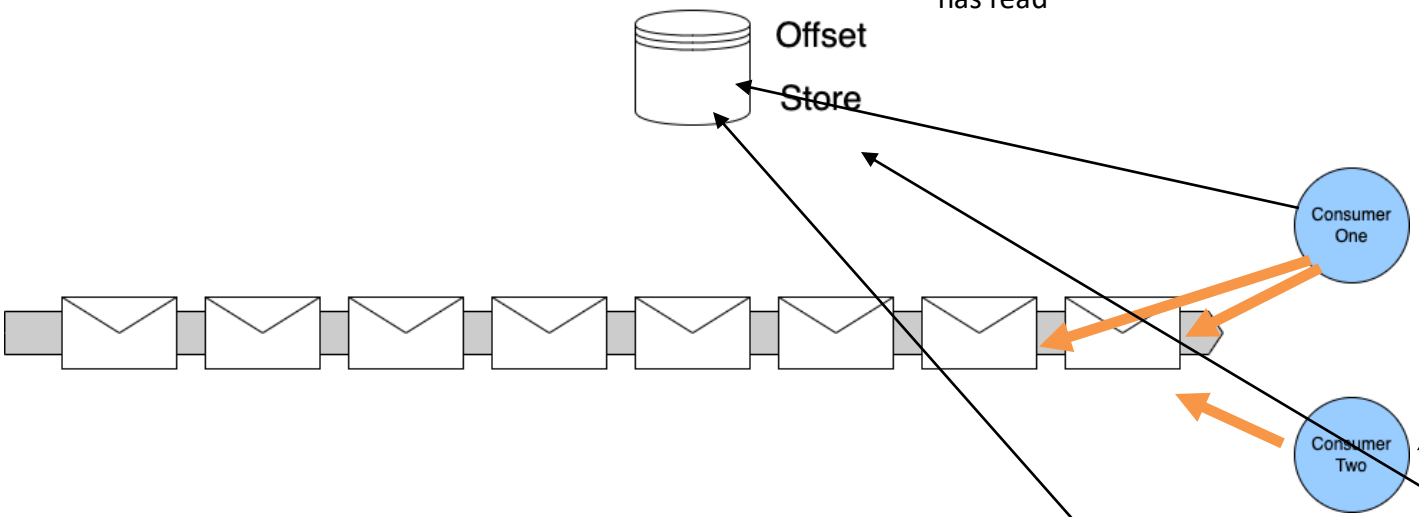
After a certain number of re-queues we may move the message to a dead-letter channel, it turns out that no one action the request within in a reasonable time frame

Queues contain Tasks

With queues we can think of the messages on a queue as tasks - they are a request for us to carry out an action. Once the action is done, we can delete the task.

- We don't anyone else to action it, it's already been done.
- Someone receiving a done task will have to discard it.
- If we can't action it, someone else will need to action it.

Streams



First consumer *stores an offset* to mark how far it has read

First consumer *reads* the next message in the queue and processes it.

First consumer *reads* the next message in the queue and processes it.

If a consumer *restarts*

It reads the store to determine the last message it processed

Second consumer *stores an offset* to mark how far it has read

Second consumer *reads* the next message in the queue and processes it.

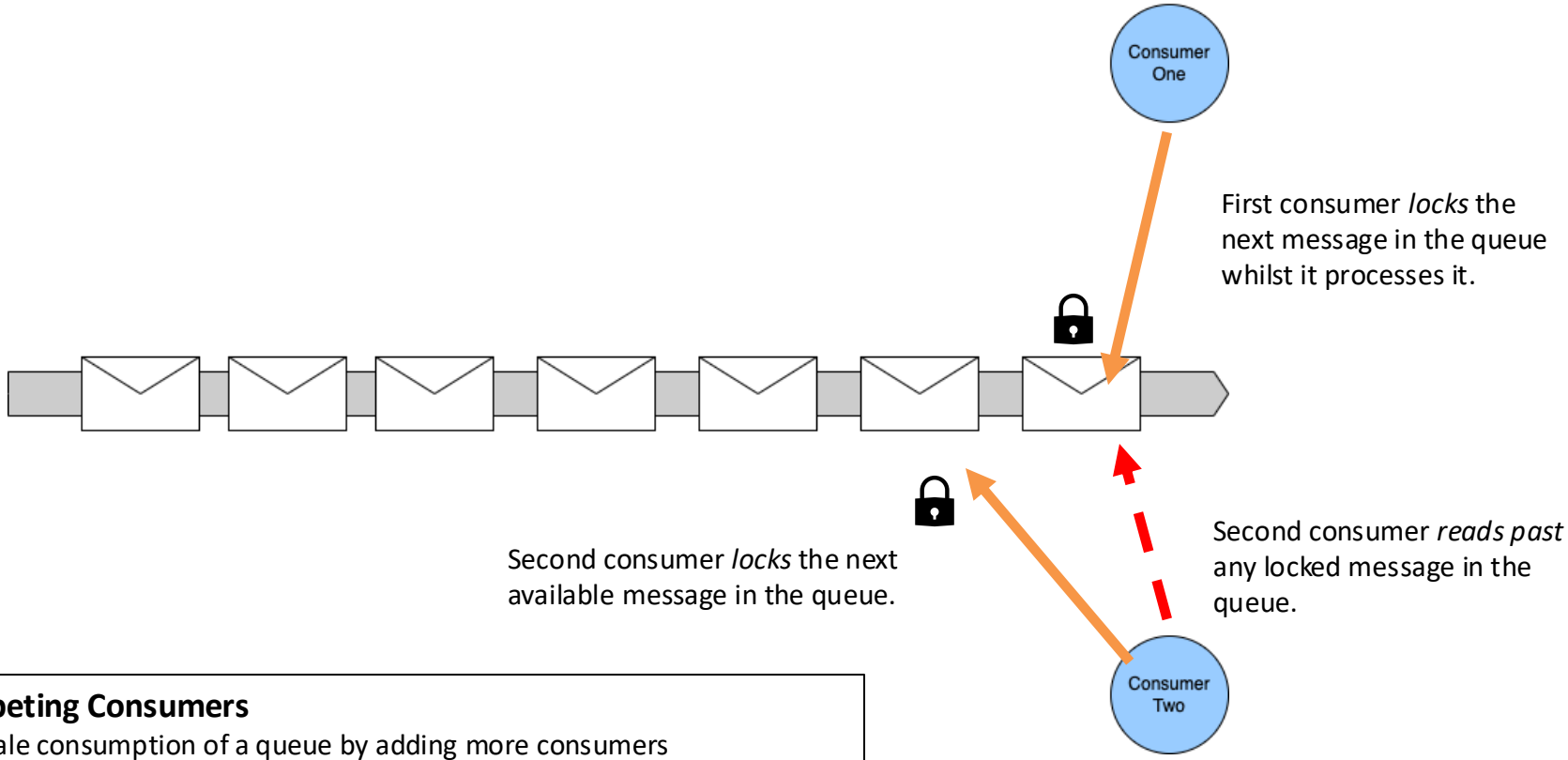
Streams contain Facts

With streams we can think of the records on the stream as facts - they are records that indicate there has been a change in state.

- We can view facts as an 'inverse database' they represent how current stat is arrived at
- We can navigate offsets to calculate a position at a 'point in time'
- We don't consume facts by reading them, they persist

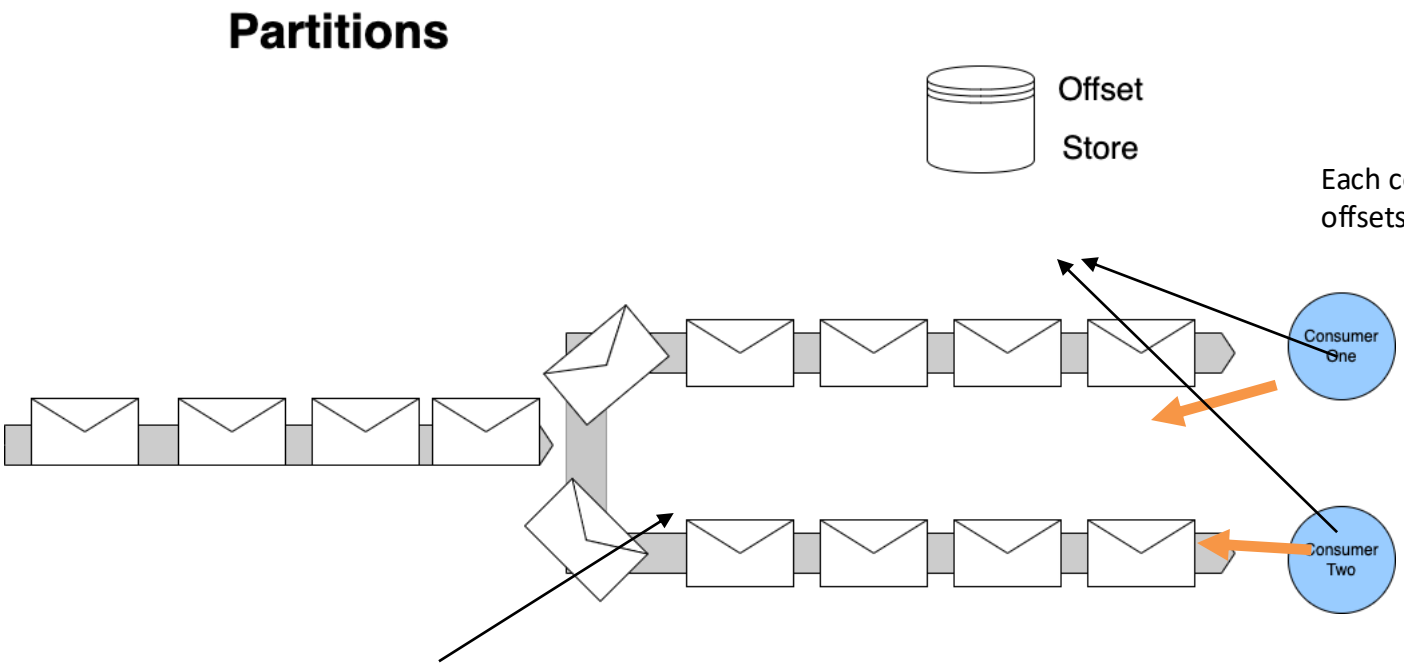
Scaling Queues and Streams

Queues



Competing Consumers
We scale consumption of a queue by adding more consumers

Streams



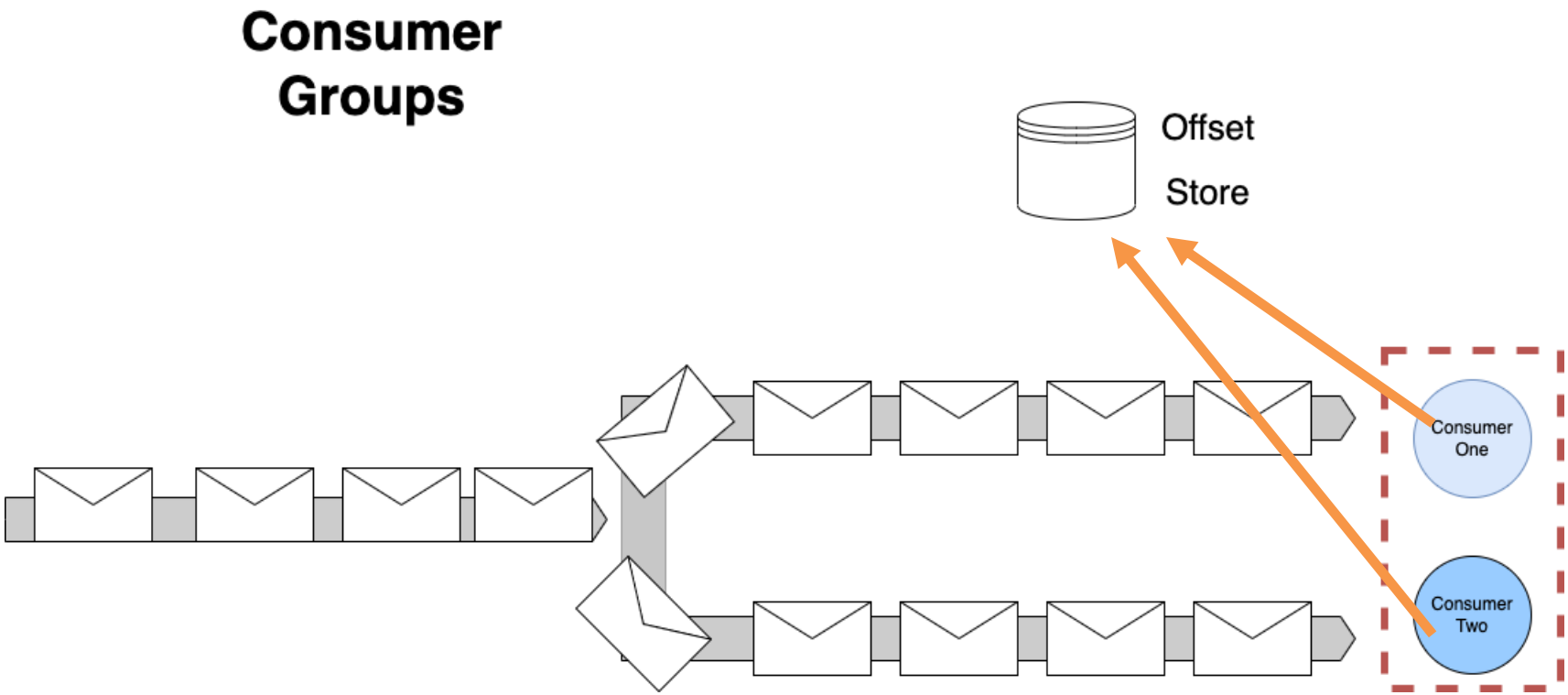
Each consumer manages offsets for their partition.

To scale out we must divide the stream up. We partition the stream to allow multiple consumers to read it.

For any set of events that must be processed sequentially - all changes to one entity for example - we use consistent hashing to push messages with the same identifier to the same partition. This allows us to scale, whilst preserving our ordering.

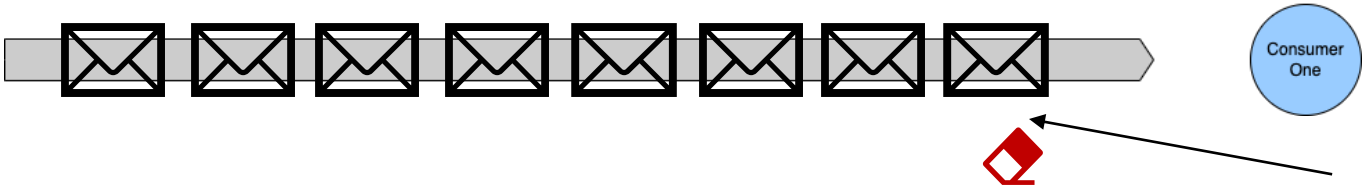
Streams

To provide availability – only one consumer in a group can read from a partition at a time – but a consumer in a group may read from more than one of the partitions owned by that group



Archive and Replay

Queues

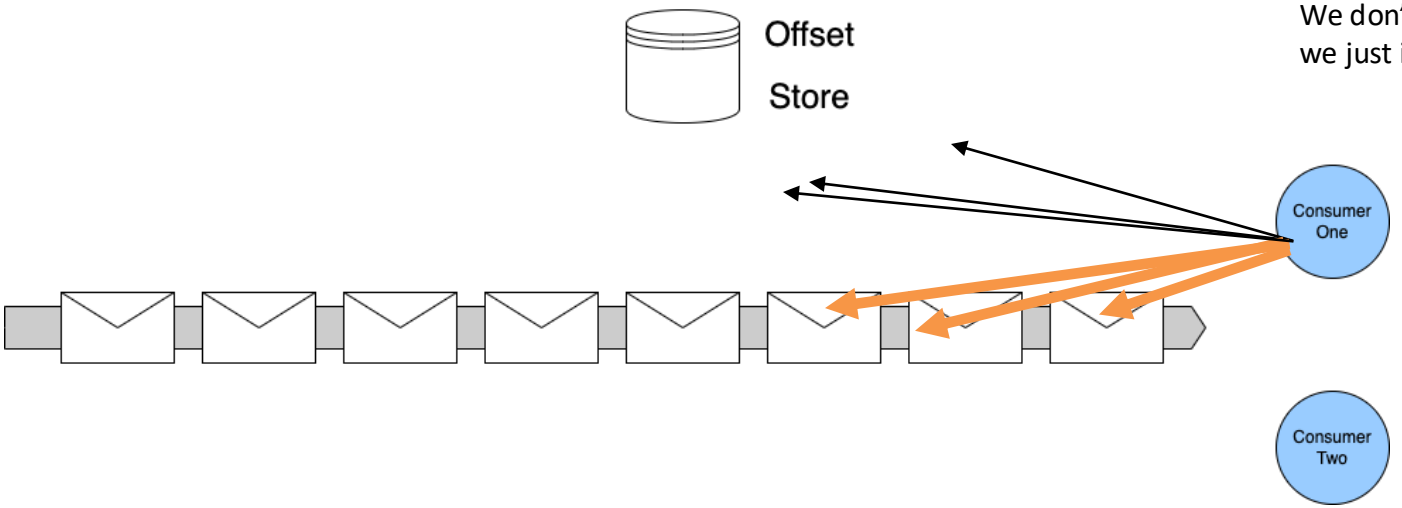


If we succeed, then we delete the message from the queue, no one else can now process it.

No Archive and Replay

With queues we delete a message once we have completed the associated action. That means we have no way to replay the request for work. Our only option is to ask the producer to resend their request.

Streams



We don't delete a message,
we just increment the offset.

Archive and Replay
Archive and Replay is straightforward as nothing is deleted. We simply reset the consumer's offset to re-read the stream

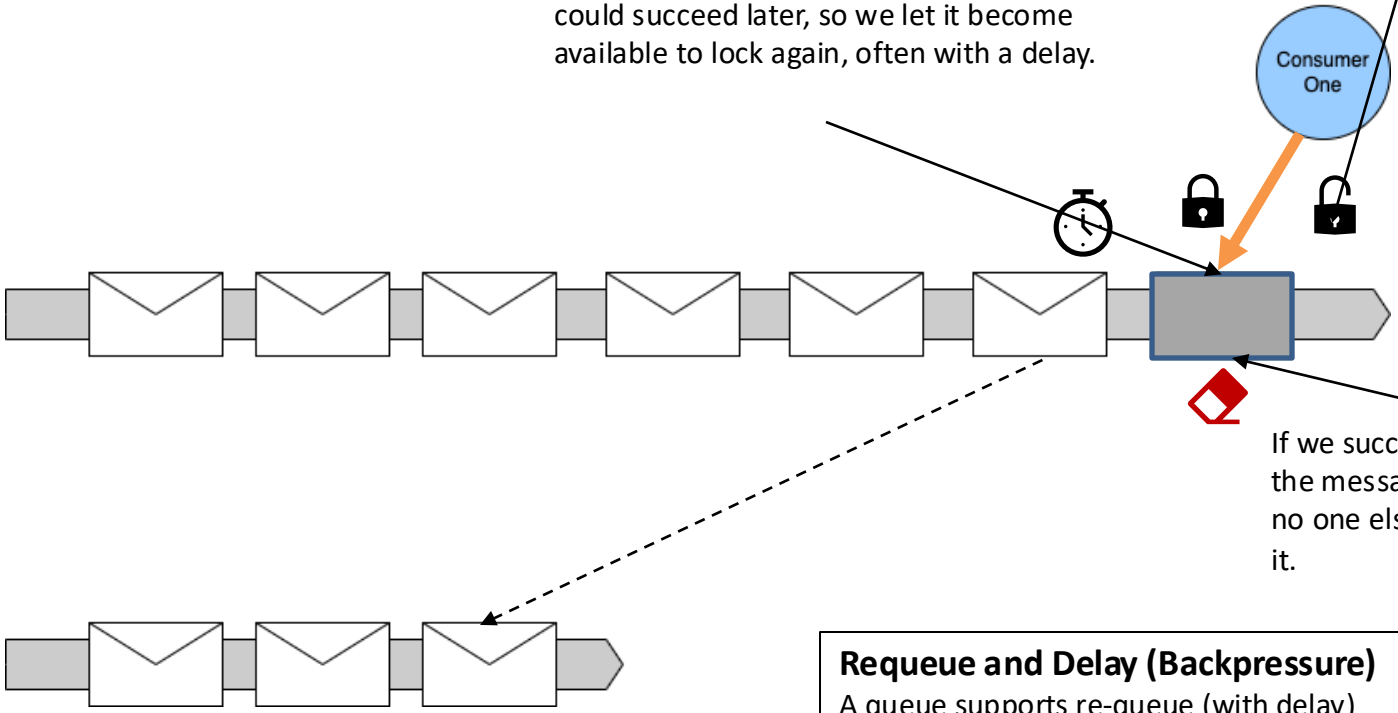
Requeue and Delay (Backpressure)

Queues

When we are done processing, we unlock it.

- Usually because we finish processing it
- May be because we fail.

If we fail, then we may decide others could succeed later, so we let it become available to lock again, often with a delay.



If we succeed, then we delete the message from the queue, no one else can now process it.

After a certain number of re-queues we may move the message to a dead-letter channel, it turns out that no one action the request within in a reasonable time frame

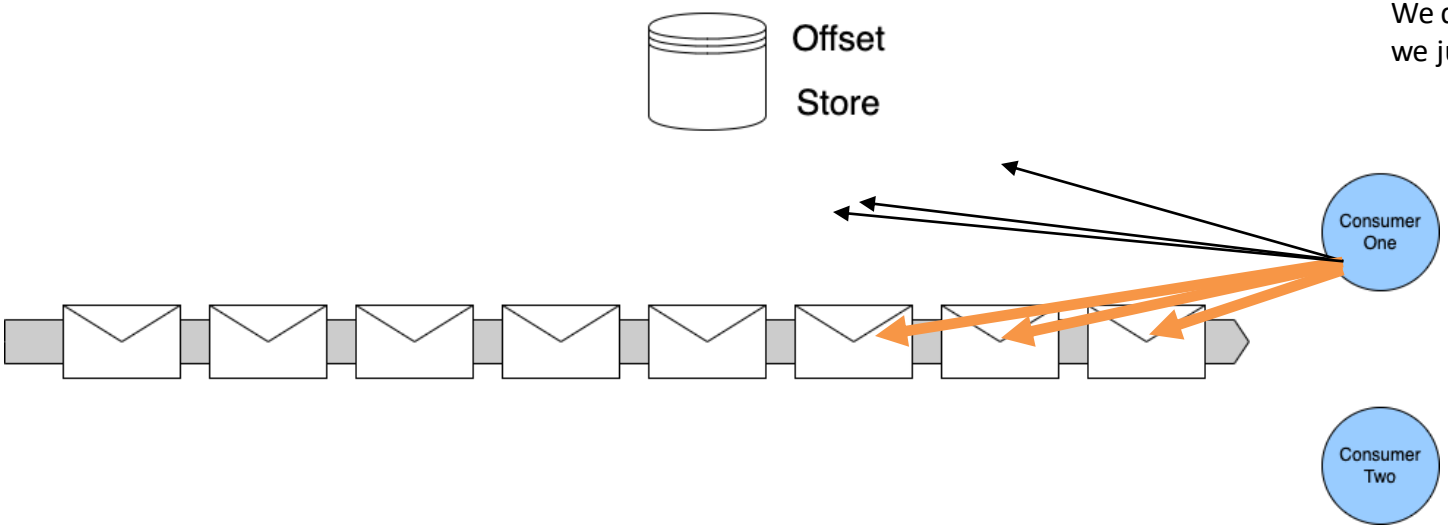
Requeue and Delay (Backpressure)

A queue supports re-queue (with delay)

- If the work is not done/acked, just make it available again to the next consumer
- If the work could not be done because of a transient issue, delay to let is pass

Streams

We don't delete a message,
we just increment the offset.



No Requeue or DLQ

Because we do not lock items, we do not requeue items, including requeue with delay. Your strategy is:

- Ignore and Continue (Load Shedding)
- Retry (Backpressure)
- Copy to another stream (a delay or DLQ stream)

	Messaging	Discrete Event	Series Event
Queue	✓	✓	✗
Stream	✗	✓	✓

	Ordering	Archive and Replay	Requeue with Delay
Queue	✓ ✗	✗	✓
Stream	✓	✓	✗

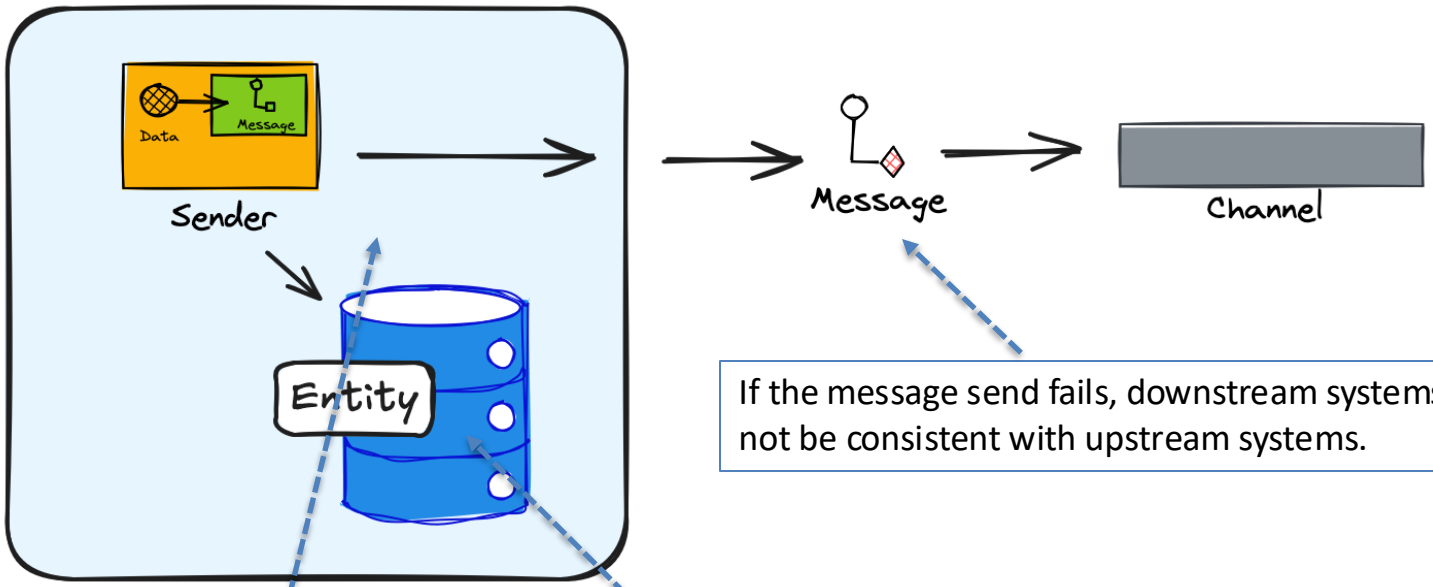
EXERCISE MATERIAL

Introduction to Kafka

- Readme
- Slides

Transactional Messaging

Transactional Messaging

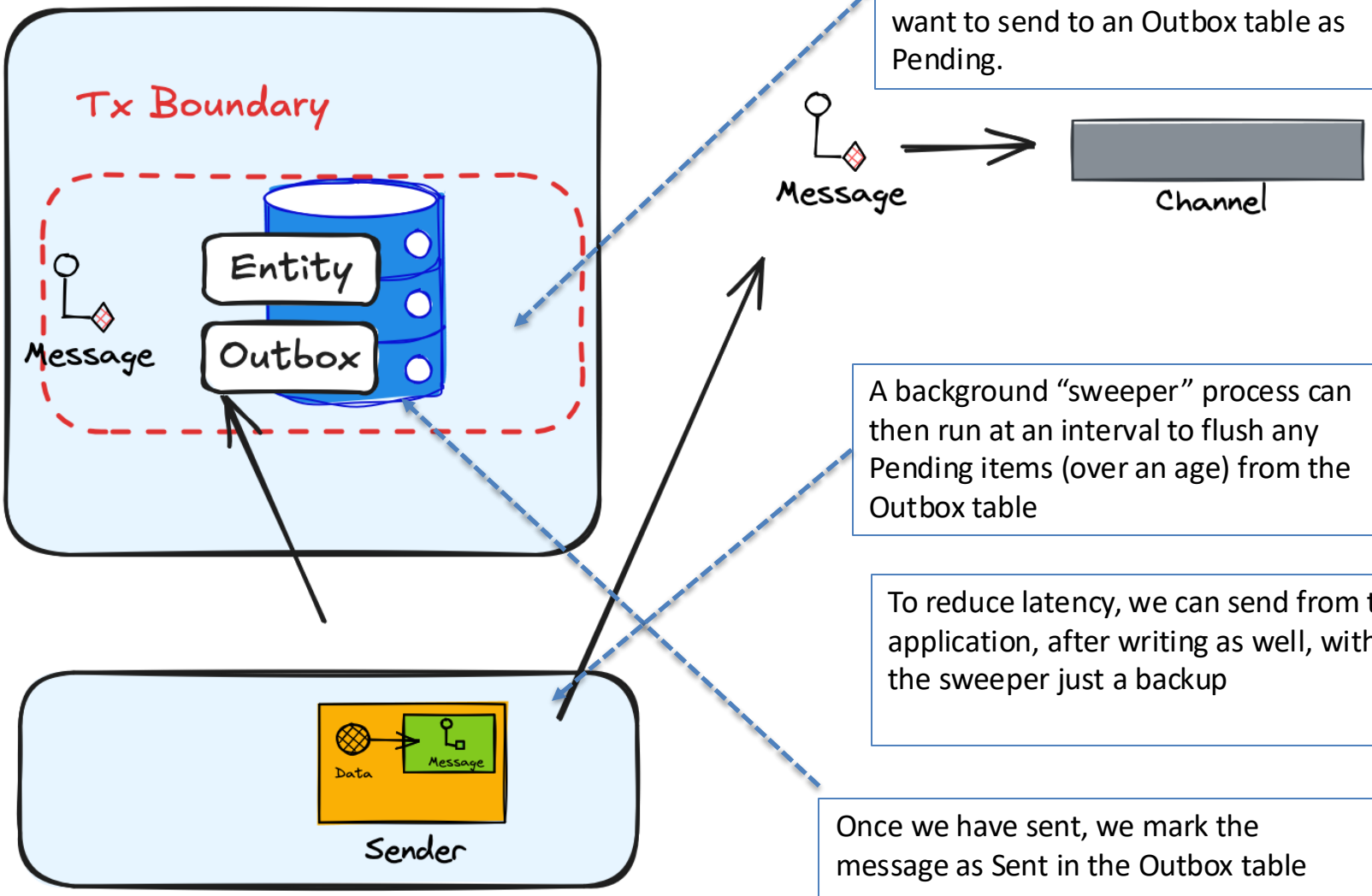


If the message send fails, downstream systems will not be consistent with upstream systems.

No transaction spans both my write to the Db for an entity, and my sending of a message.

If we reverse the operations and the message is sent but the Db write fails, the upstream is inconsistent.

Outbox



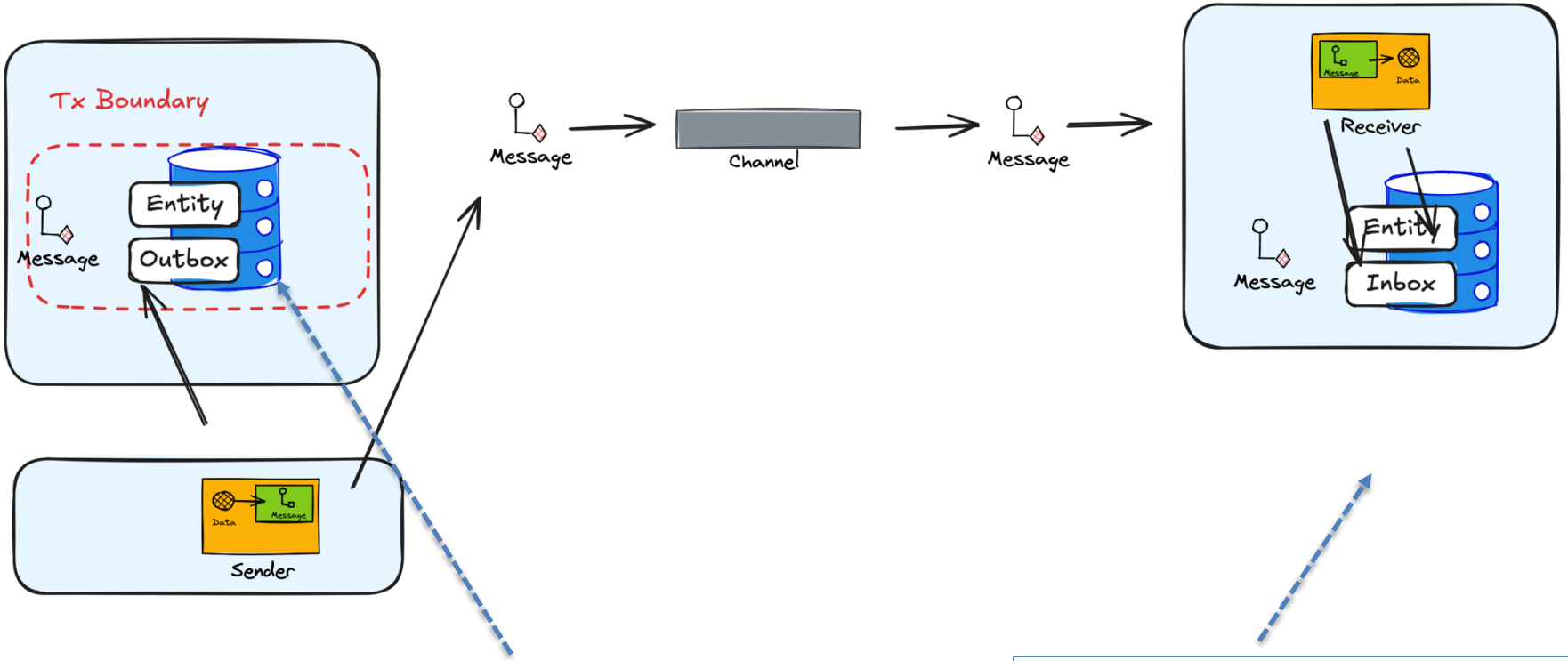
Instead of writing directly to the channel, we open a transaction, update our entity, and write the message we want to send to an Outbox table as Pending.

A background “sweeper” process can then run at an interval to flush any Pending items (over an age) from the Outbox table

To reduce latency, we can send from the application, after writing as well, with the sweeper just a backup

Once we have sent, we mark the message as Sent in the Outbox table

Inbox

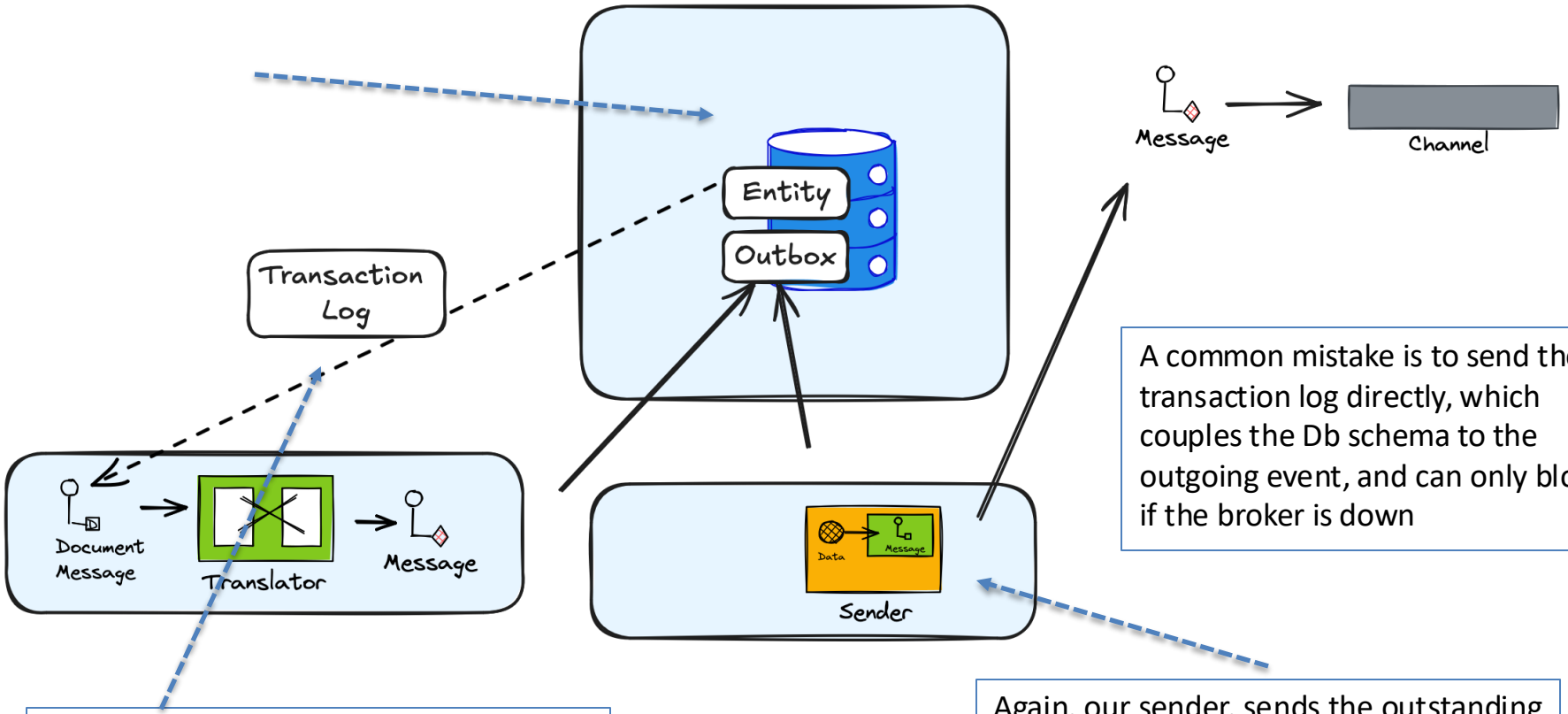


We cannot guarantee that we fail not mark dispatched after a send, so we may send the same message twice

If the message is not idempotent (side-effect free), then we use an Inbox to record messages seen (working on) and processed (might fail)

Log Tailing

We may not be using a database that supports ACID transactions, across Outbox and Entity tables



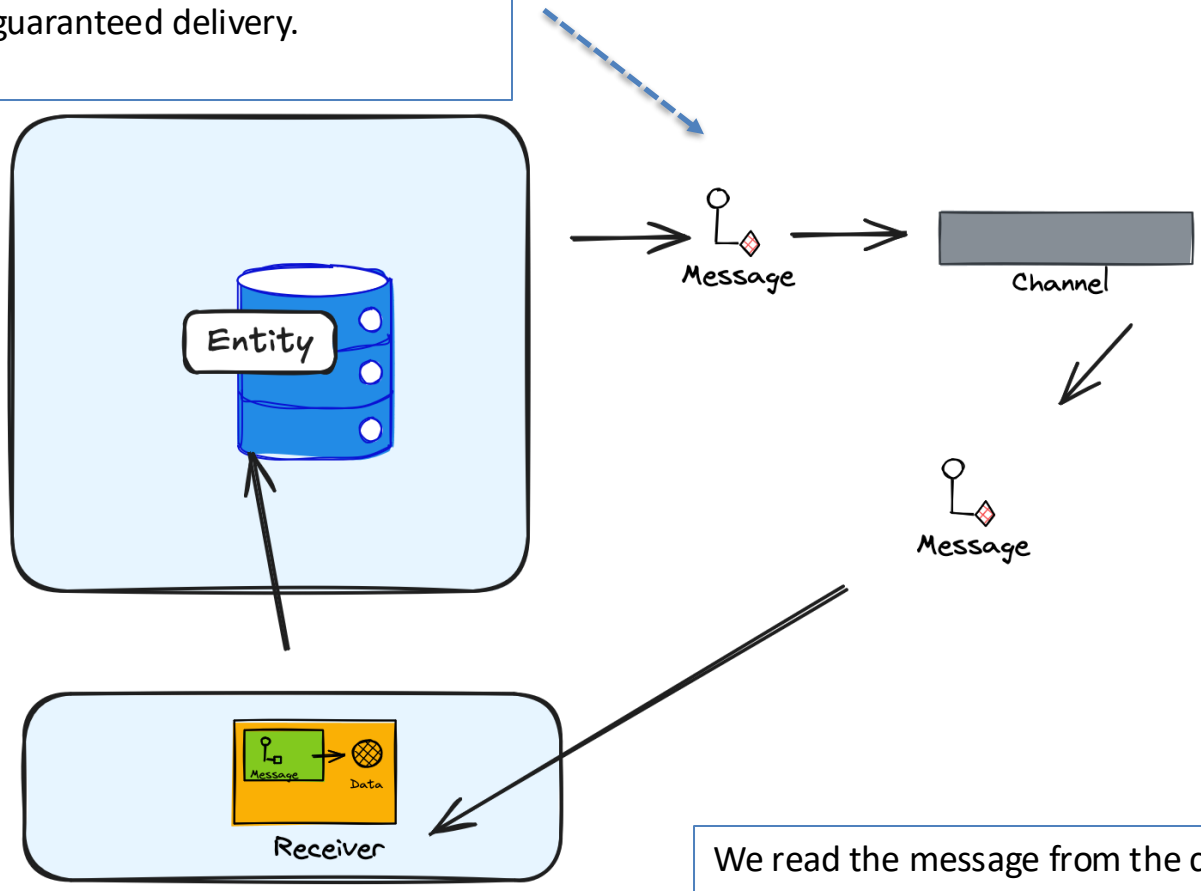
A common mistake is to send the transaction log directly, which couples the Db schema to the outgoing event, and can only block if the broker is down

In this case we can tail the transaction log, translate the log (via domain code) into a message and store in the Outbox

Again, our sender, sends the outstanding Outbox entries and marks them as sent in the Outbox

State Change Capture

A low-cost alternative is to send the message before we write to the entity. If the message sends, we have guaranteed delivery.



We read the message from the channel and then update the entity. The main issue is that we must be able to cope with EVENTUAL CONSISTENCY, which is not always simple.

MANAGING ASYNCHRONOUS APIS

Versioning

Be strict when sending and tolerant when receiving.
Implementations must follow specifications precisely when sending to the network, and tolerate faulty input from the network.

Robustness Principal or Postel's Law – Jon Postel RFC 1958

Tolerant Reader

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "orderid": {
      "type": "string"
    },
    "customerName": {
      "type": "string"
    },
    "addressLineOne": {
      "type": "string"
    },
    "postCode": {
      "type": "string"
    },
    "pinCode": {
      "type": "string",
      "pattern": "^[0-9]+$"
    }
  },
  "required": ["orderid", "customerName", "addressLineOne",
"postCode", "pinCode"],
  "additionalProperties": false
}
```

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "orderid": {
      "type": "string"
    },
    "customerName": {
      "type": "string"
    },
    "addressLineOne": {
      "type": "string"
    },
    "postCode": {
      "type": "string"
    },
    "pinCode": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  },
  "required": ["orderid", "customerName", "addressLineOne",
"postCode", "pinCode"],
  "additionalProperties": false
}
```

Ignore New Fields

Tolerant Reader

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "orderid": {
      "type": "string"
    },
    "customerName": {
      "type": "string"
    },
    "addressLineOne": {
      "type": "string"
    },
    "postCode": {
      "type": "string"
    },
    "pinCode": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  },
  "required": ["orderid", "customerName", "addressLineOne",
"postCode", "pinCode"],
  "additionalProperties": false
}
```

Default Latitude: 0
Default Longitude: 0

Note: not required

Default Missing Fields

Breaking Change

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "orderid": {
      "type": "string"
    },
    "firstName": {
      "type": "string"
    },
    "surName": {
      "type": "string"
    },
    "addressLineOne": {
      "type": "string"
    },
    "pinCode": {
      "type": "string",
      "pattern": "^[0-9]+$"
    },
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  },
  "required": ["orderid", "firstName", "surName",
"addressLineOne", "pinCode", "latitude", "longitude"],
  "additionalProperties": false
}
```

We might be able to write code to deal with this change, but we have to know that a required field is missing and we have new fields instead

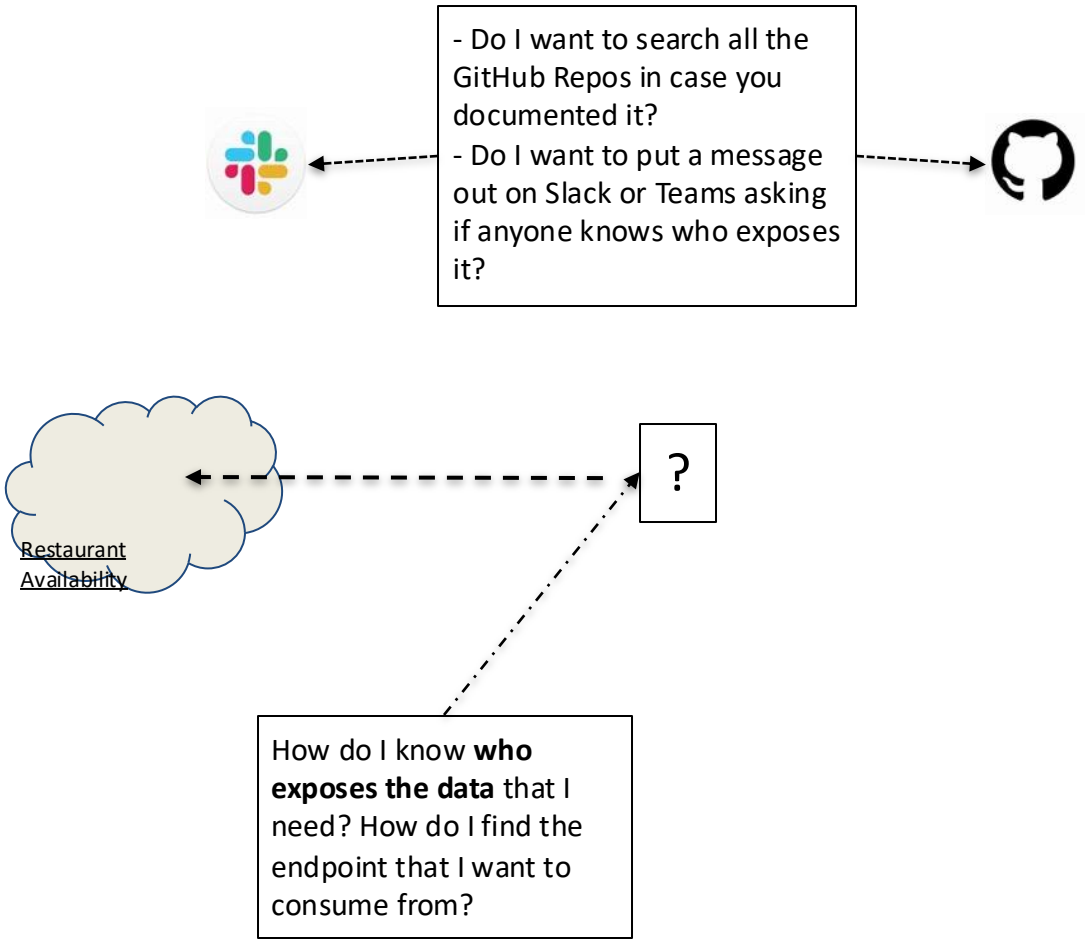
For this we need to rely on a version in the header, and the ability to process messages with this new version, alongside old ones to allow us to run out the old until new replaces it.

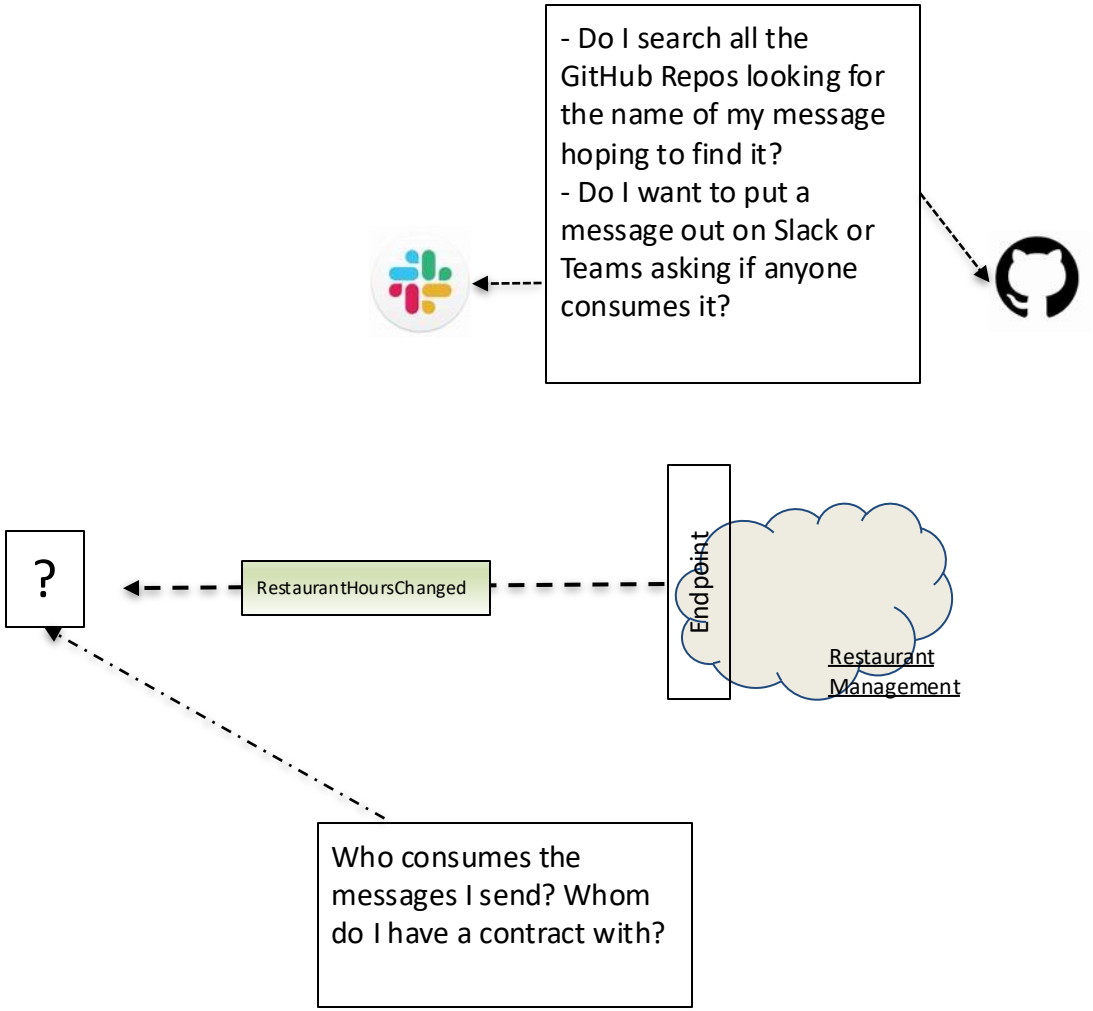
Documentation

Endpoints are **places where messages are sent or received** (or both), and they define all the information required for the message exchange.

An *endpoint* describes in a standard-based way **where messages should be sent, how they should be sent, and what the messages should look like**.

https://docs.microsoft.com/en-us/dotnet/framework/wcf/fundamental-concepts

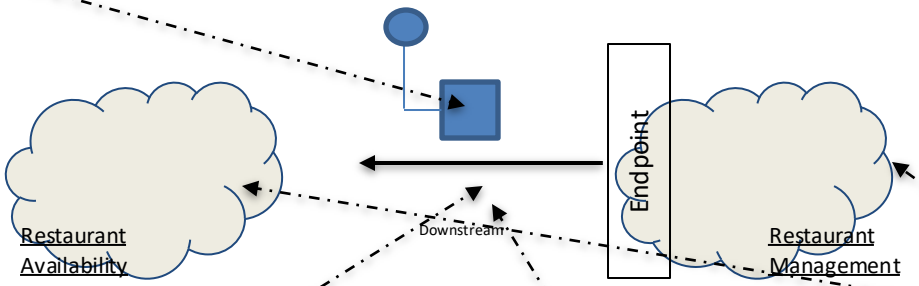

- Do I want to search all the GitHub Repos in case you documented it?
- Do I want to put a message out on Slack or Teams asking if anyone knows who exposes it?
Restaurant Availability
?
How do I know **who exposes the data** that I need? How do I find the endpoint that I want to consume from?


- Do I search all the GitHub Repos looking for the name of my message hoping to find it?
- Do I want to put a message out on Slack or Teams asking if anyone consumes it?
?
RestaurantHoursChanged
Endpoint
Restaurant Management
Who consumes the messages I send? Whom do I have a contract with?

Asynchronous Endpoints

Our Asynchronous APIs need documenting just like any other API (HTTP etc).

We need to document the message, because it is the contract

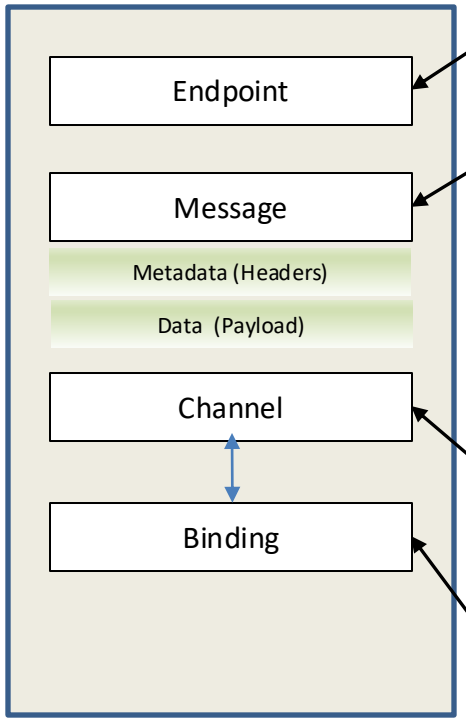


We need to document the channel so we know where the message is flowing

We need to document the protocol so we know how to send-receive

We need to document who sends and receives to understand flow

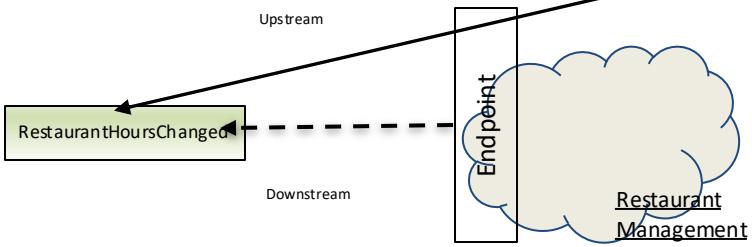
Endpoint – Documenting the Contract



An **endpoint** is where **messages should be sent, how they should be sent, and what the messages should look like**. What *promises* do we make.

An **message** is data that we send or receive

We will need to agree what the metadata – the **headers** are – and we will need to agree the *encoding* and *schema* of the **body**.



A **channel** is the logical pipe over which messages flow. It is our address.

A **binding** is the details of how we implement the channel: what transport and encoding does the service support that allows messages to flow for example AMQP and Plain/Text.

In principle you can have **multiple bindings** for a **channel**.

Why AsyncAPI?

Improving the current state of Event-Driven Architectures (EDA)

Specification

Allows you to define the interfaces of asynchronous APIs and is protocol agnostic.

Documentation

Document APIs

Use our tools to generate documentation at the build level, on a server, and on a client.

HTML Template React Component

Code Generation

Generate documentation, Code (TypeScript, Java, C#, etc), and more out of your AsyncAPI files.

Generator Modelina

Community

We're a community of great people who are passionate about AsyncAPI and event-driven architectures.

Join our Slack

Open Governance

Our Open-Source project is part of Linux Foundation and works under an Open Governance model.

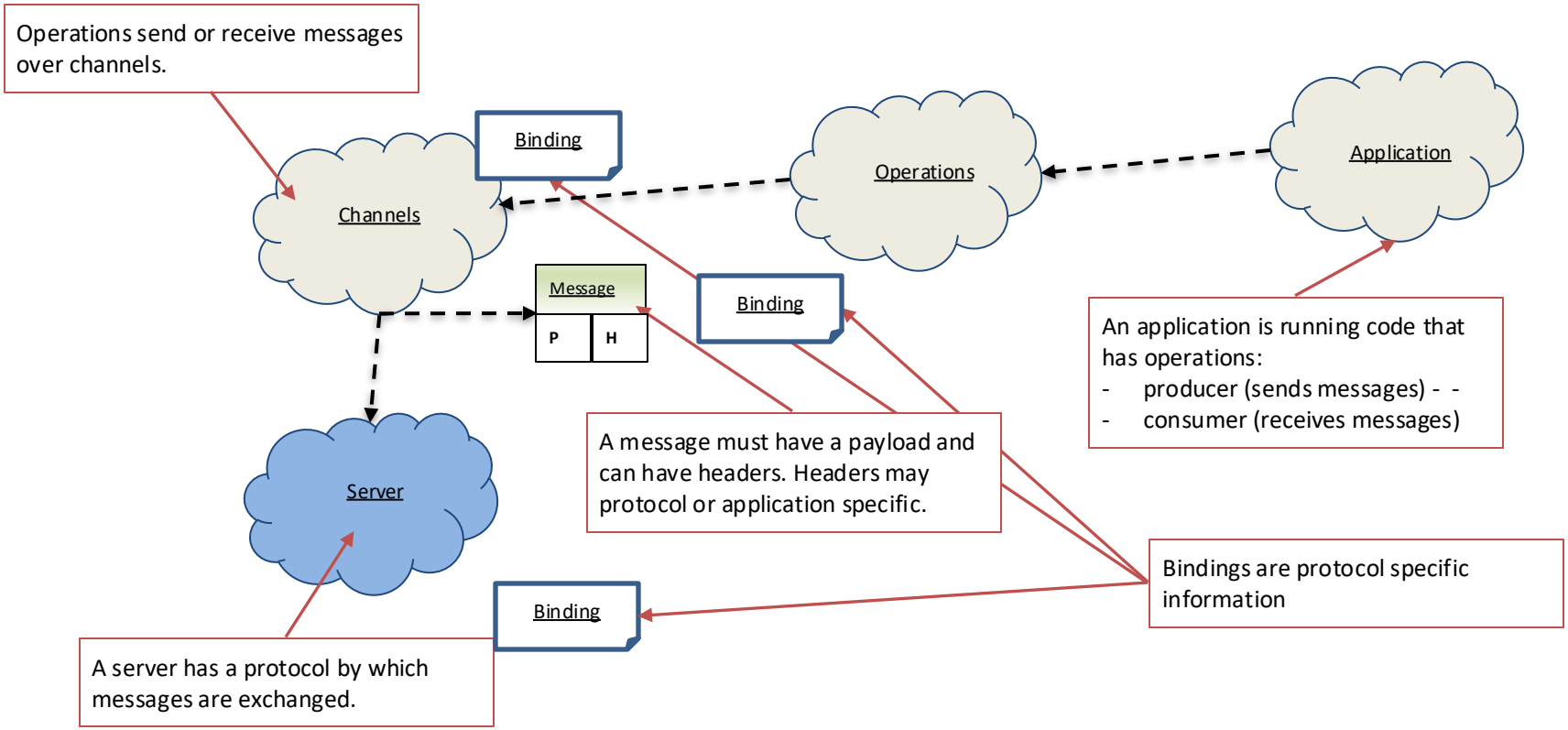
Read more about Open Governance TSC Members

And much more...

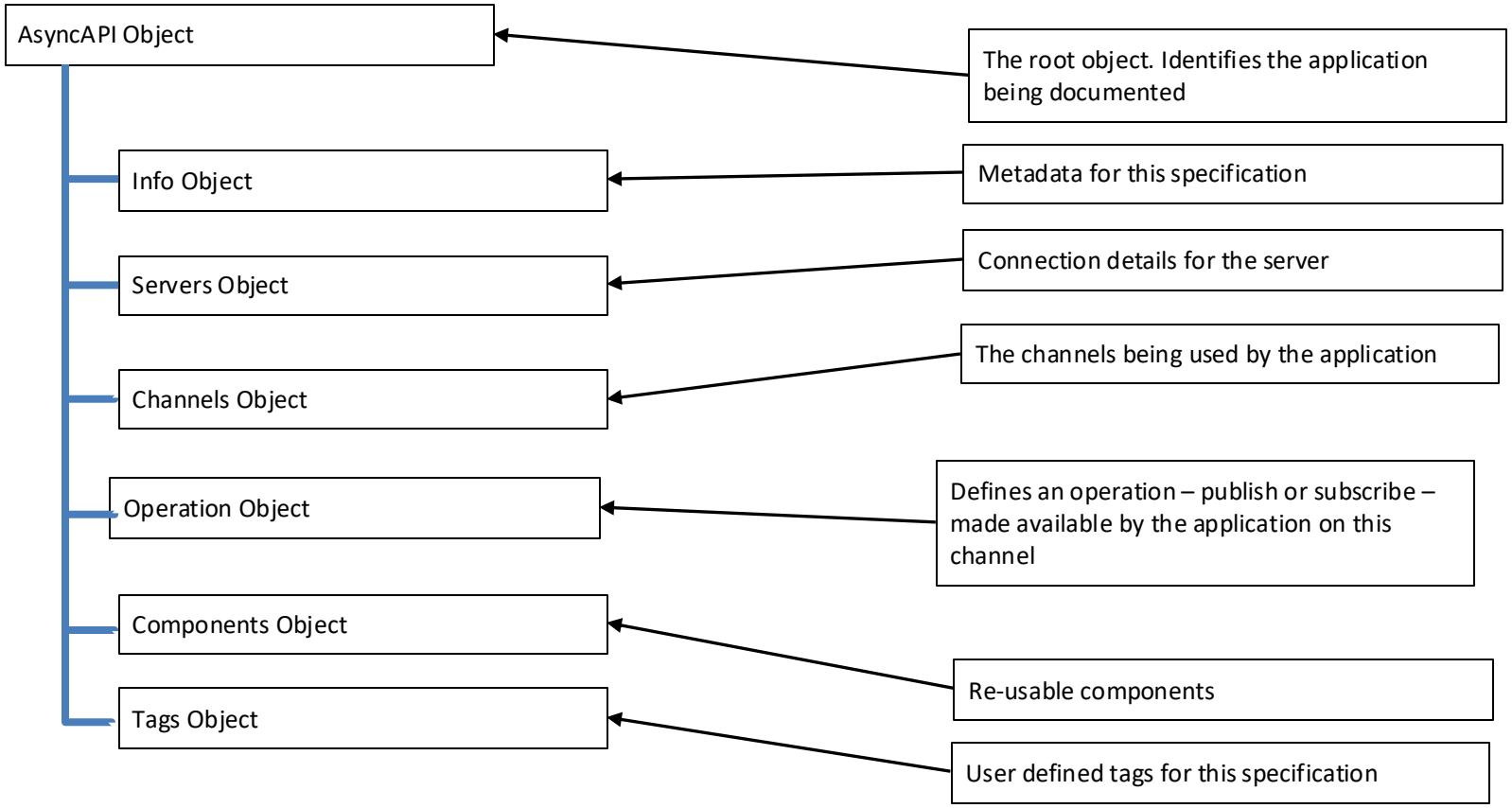
We have many different tools and welcome you to explore our ideas and propose new ideas to AsyncAPI.

View GitHub Discussions

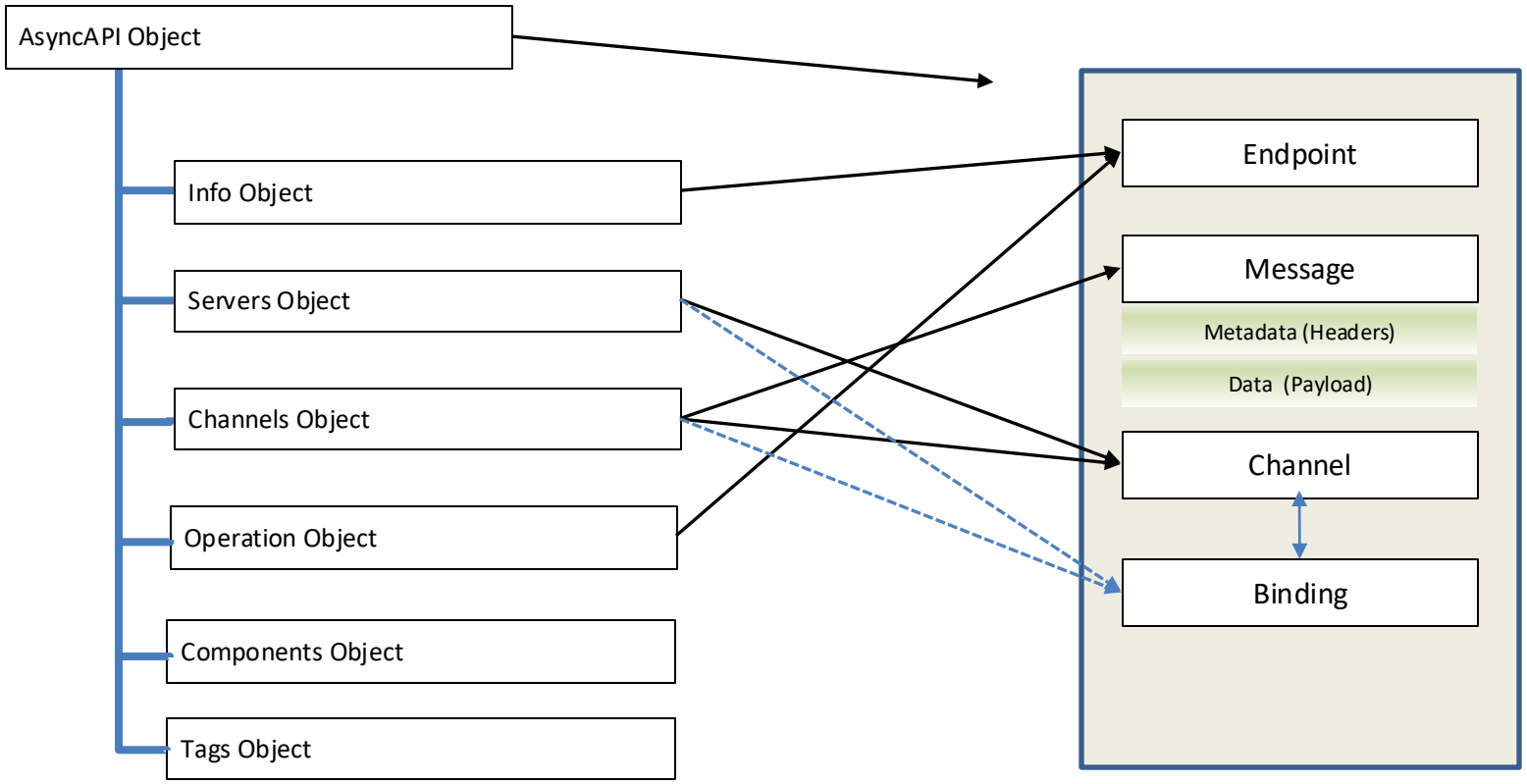
AsyncAPI Elements (V3)



Document Structure (V3)



AsyncAPI (V3) and Endpoint ABCs



Info Object

```
info:
  contact:
    name: Paramore Brighter
    url: https://goparamore.io/support
    email: support@goparamore.io
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  description: Demonstrates sending a greeting over a messaging
transport.
  title: Brighter Sample App
  version: 1.0.0
tags:
  - name: brighter examples
```

Servers Object

```
development:
  description: A Kafka broker for local development
  url: localhost:9092
  protocol: kafka
```

Channels (V3)

```
greeting:
  address: ‘goparamore.io.greeting’
    summary: For sending greetings
    description: This channel contains greeting messages
    servers:
      - $ref: '#/servers/development’
    messages:
        greeting:
          $ref: "#/components/messages/greeting”
    bindings:
       kafka:
          partitions: 20
          replicas: 3
```

Operations (V3)

```
sendGreeting :
  action: send
  summary: sends a greeting
  description: The application sends a greeting to a consumer.
  channel:
    $ref: "#/channels/greeting”
  bindings:
     kafka:
        partitions: 20
        replicas: 3
```

Components

```
components:
  messages:
    greeting:
      name: greeting
      title: A salutation
      summary: This is how we send you a salutation
      contentType: application/json
      traits:
        - $ref: '#/components/messageTraits/commonHeaders'
      payload:
        $ref: "#/components/schemas/greetingContent"

  schemas:
    greetingContent:
      type: object
      properties:
        greeting:
          type: string
          description: The salutation you want to send
  …
```

JSON Schema (AsyncAPI Schema Object)

$schema

$id

title

description

type

properties

```
{
  ”$schema” : "https://json-schema.org/draft/2020-12/schema",
  “$id” : “https://goparamore.io/greeting.schema.json”,
  “title” : “greeting”,
  “description” : “A greeting message”,
  “type” : “object”,
  “properties” : {
      “greeting” : {
        “description” : “the salutation”
        “type” : string
       }
  }
}
```

Avro

Complex Types: **records** | **enums** | **arrays** | **maps** | **unions** | **fixed**

Records:

- **name**
- **namespace**
- **doc**
- **alias**
- **fields**
  - **name**
  - **doc**
  - **type**
  - **default**

Avro

```
{
  "type" : "record",
  "name" : "greeting,"
  "title" : "greeting",
  "fields" : [
    {"name" : "greeting", "type" : "string"}
  ]
}
```

Avro

Encodings:

- **JSON**
- **Binary**

Languages:

- **C**
- **C++**
- **C#**
- **Java**
- **Perl**
- **Python**
- **Ruby**
- **Others…**

Protobuf

```
syntax = "proto3";

message Greeting {
   string greeting = 1;
}
```

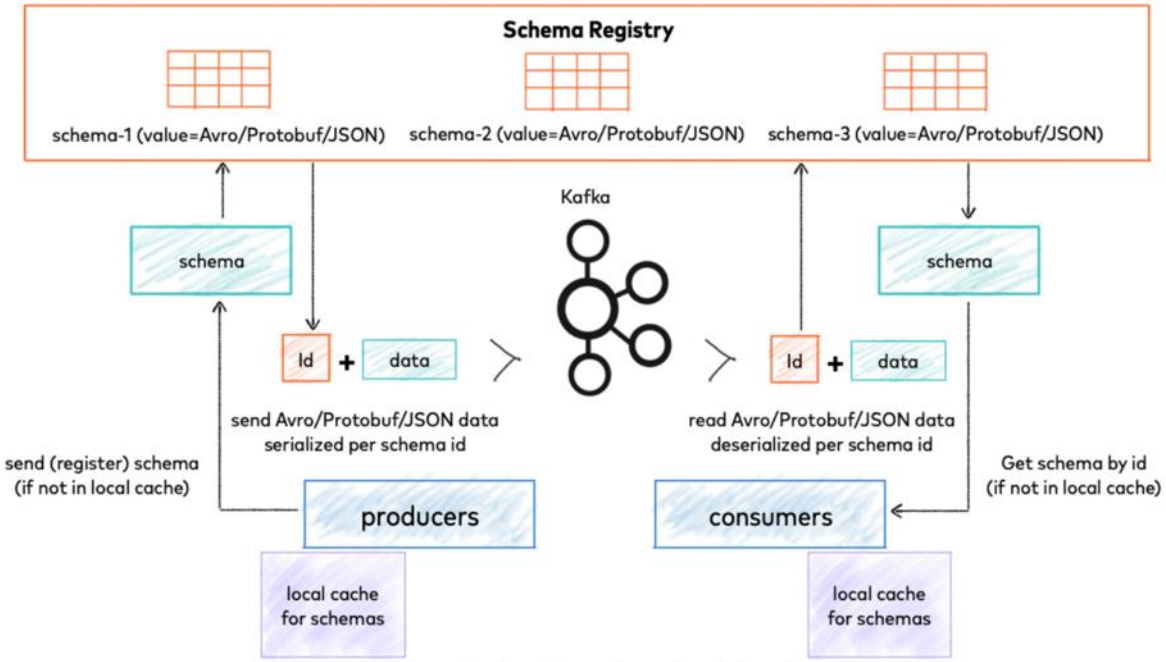
Protobuf

Encodings: **Binary**

Languages:

- **C++**
- **C#**
- **Dart**
- **Go**
- **Kotlin**
- **Java**
- **Objective-C**
- **Python**
- **Ruby**
- **Others…**

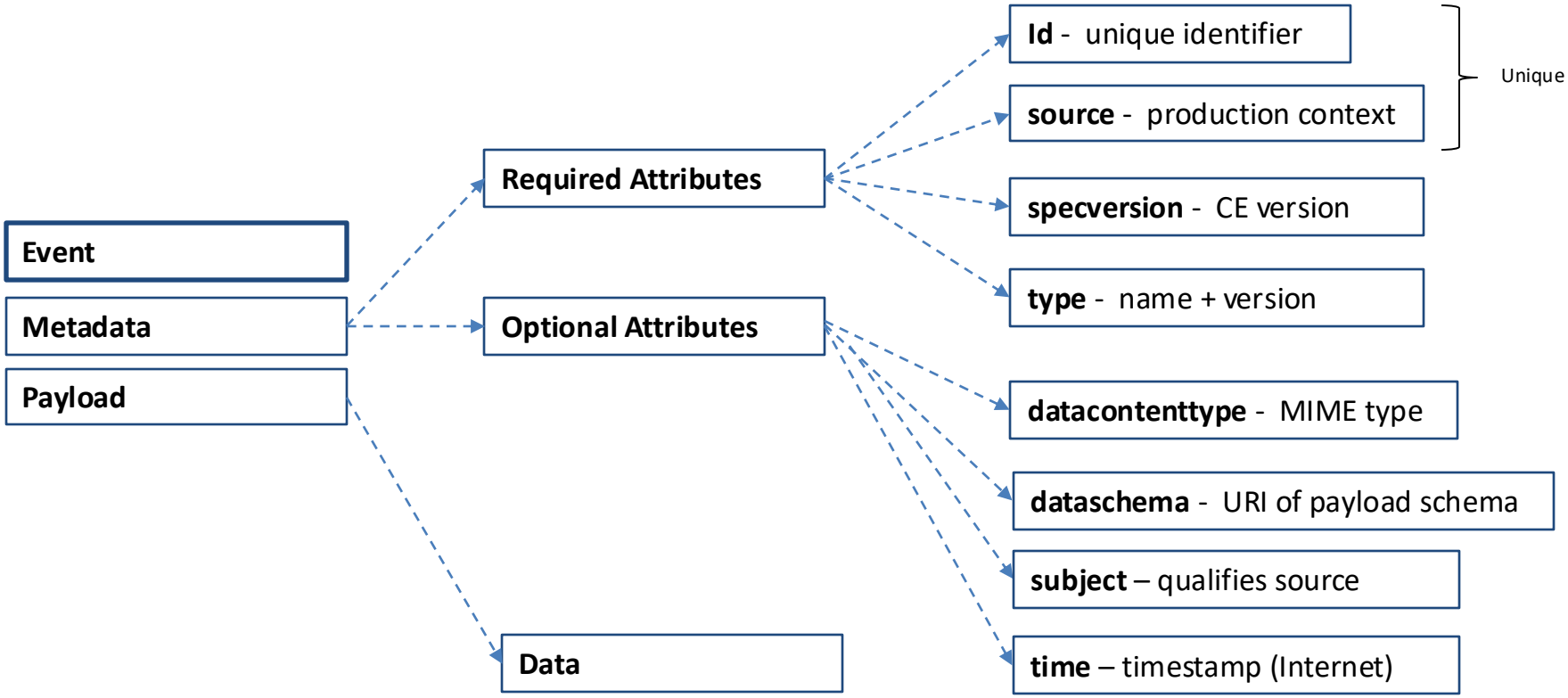
Schema Registry



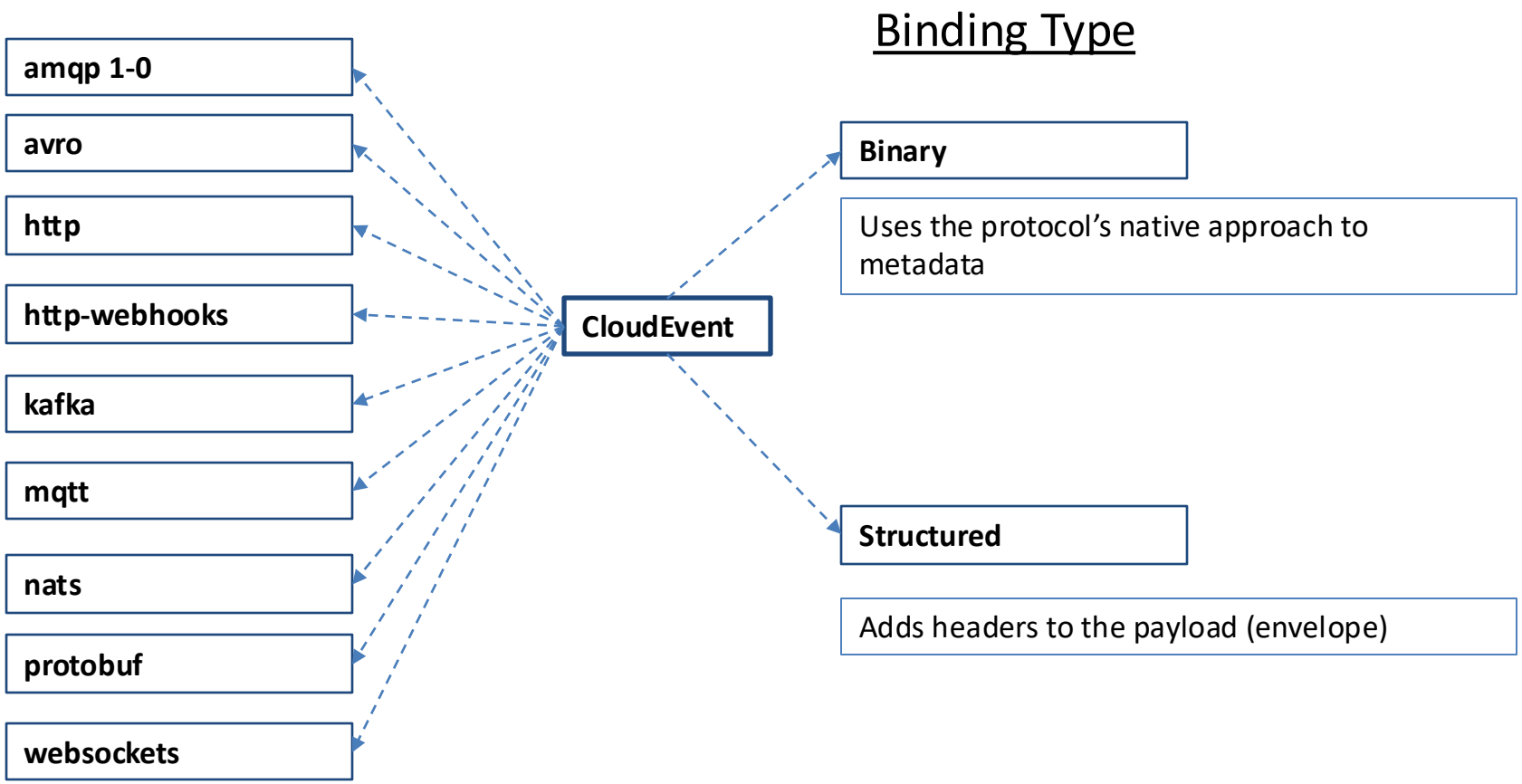
Confluent Schema Registry for storing and retrieving schemas

https://docs.confluent.io/platform/current/schema-registry/index.html

Cloud Events



Protocol Binding



Protocol Binding

Binary

```
---------------- Message -----------------
Topic Name: mytopic
 ----------------- key --------------------
Key: mykey
---------------- headers -----------------
ce_specversion: "1.0"
ce_type: "com.example.someevent"
ce_source: "/mycontext/subcontext"
ce_id: "1234-1234-1234"
ce_time: "2018-04-05T03:56:24Z"
content-type: application/avro
----------------- value -------------------
 ... application data encoded in Avro ...
-------------------------------------------
```

Structured

```
---------------- Message -----------------
Topic Name: mytopic
 ----------------- key --------------------
Key: mykey
---------------- headers -----------------
content-type: application/cloudevents+json; charset=UTF-8
----------------- value -------------------
 {
        "specversion" : "1.0",
        "type" : "com.example.someevent",
        "source" : "/mycontext/subcontext",
        "id" : "1234-1234-1234",
        "time" : "2018-04-05T03:56:24Z",
        "datacontenttype" : "application/json",
        "data" :        {
                ... application data encoded in JSON ...
        }
 }
-------------------------------------------
```

AsyncAPI Object

id: https://github.com/brightercommand/greetings/

We use a specification file for an app, which is a producer or consumer, and identify them by id

VS Code

brighter-greetings-sender.yml > YAML > {} info > {} contact > url

```yaml
asyncapi: '2.0.0'
id: "https://github.com/brightercommand/greetings-sender/"
info:
  contact:
    name: Paramore Brighter
    url: https://goparamore.io/support
    email: support@goparamore.io
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  description: Demonstrates sending a greeting over a messaging transpo
  title: Brighter Sample App
  version: 1.0.0
tags:
  - name: brighter examples

servers:
  localhost:
    url: localhost:9092
    protocol: kafka

channels:
  greeting:
    description: A channel for sending out greeting messages
    subscribe:
      description: This service lets you send the 'Hello World' greetin
      operationId: sendMessage
      message:
        $ref: '#/components/messages/greeting'

components:
  messages:
    greeting:
      name: greeting
      title: A salutation
      summary: This is how we send you a salutation
      contentType: application/json
      traits:
        - $ref: '#/components/messageTraits/commonHeaders'
      payload:
        $ref: "#/components/schemas/greetingContent"

  schemas:
    greetingContent:
```

AsyncAPI - brighter-greetings-sender.yml

Brighter Sample App 1.0.0

APACHE 2.0

Demonstrates sending a greeting over a messaging transport.

Contact link: PARAMORE BRIGHTER Contact email: SUPPORT@GOPARAMORE.IO

Servers

localhost:9092 KAFKA

Operations

SUB greeting

A channel for sending out greeting messages

This service lets you send the 'Hello World' greeting to another service.

Accepts the following message:

A salutation greeting

This is how we send you a salutation

Backstage

Backstage

GITHUB DOCS PLUGINS BLOG DEMOS NEWSLETTER Search

- Overview
- Getting Started
- Local Development
- Core Features
  - Software Catalog
    - Overview
    - The Life of an Entity
    - Catalog Configuration
    - System Model
    - YAML File Format
    - Entity References
    - Well-known Annotations
    - Well-known Relations

`spec.type` **[required]**

The type of the API definition as a string, e.g. `openapi`. This field is required.

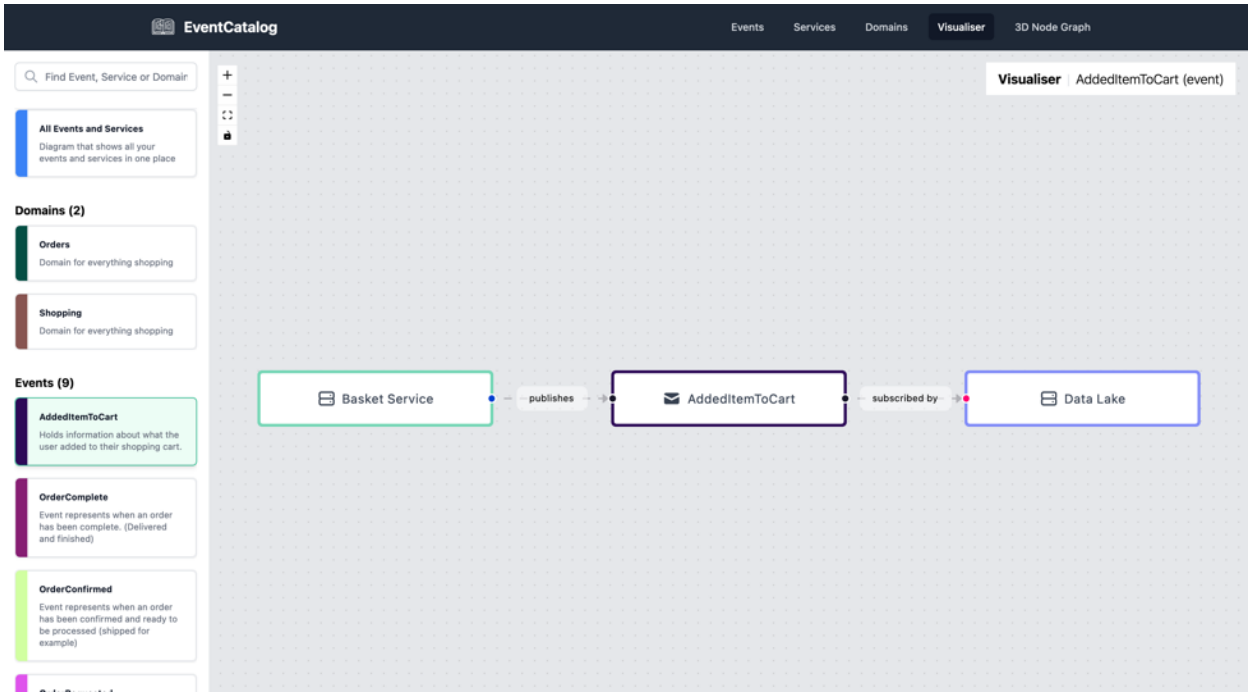
The software catalog accepts any type value, but an organization should take great care to establish a proper taxonomy for these. Tools including Backstage itself may read this field and behave differently depending on its value. For example, an OpenAPI type API may be displayed using an OpenAPI viewer tooling in the Backstage interface.

The current set of well-known and common values for this field is:

- `openapi` - An API definition in YAML or JSON format based on the OpenAPI version 2 or version 3 spec.
- `asyncapi` - An API definition based on the AsyncAPI spec.
- `graphql` - An API definition based on GraphQL schemas for consuming GraphQL based APIs.
- `grpc` - An API definition based on Protocol Buffers to use with gRPC.

Contents

Event Catalog



https://github.com/boyney123/eventcatalog

AsyncAPI Studio

AsyncAPI studio beta

Information

Servers

KAFKA localhost

Operations

SUB greeting

Messages

greeting

Schemas

greetingContent

From localStorage

```yaml
asyncapi: '2.0.0'
id: "https://github.com/brightercommand/greetings-sender/"
info:
  contact:
    name: Paramore Brighter
    url: https://goparamore.io/support
    email: support@goparamore.io
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  description: Demonstrates sending a greeting over a messaging transport.
  title: Brighter Sample App
  version: 1.0.0
tags:
  - name: brighter examples

servers:
  localhost:
    url: localhost:9092
    protocol: kafka

channels:
  greeting:
    description: A channel for sending out greeting messages
    subscribe:
      description: This service lets you send the 'Hello World' greeting to another service.
      operationId: sendMessage
      message:
        $ref: '#/components/messages/greeting'

components:
  messages:
    greeting:
      name: greeting
      title: A salutation
      summary: This is how we send you a salutation
      contentType: application/json
      traits:
        - $ref: '#/components/messageTraits/commonHeaders'
      payload:
        $ref: "#/components/schemas/greetingContent"

  schemas:
```

PROBLEMS 0 VALID NOT LATEST YAML

Brighter Sample App 1.0.0

APACHE 2.0 PARAMORE BRIGHTER SUPPORT@GOPARAMORE.IO

ID: HTTPS://GITHUB.COM/BRIGHTERCOMMAND/GREETINGS-SENDER/

Demonstrates sending a greeting over a messaging transport.

#brighter examples

Servers

localhost:9092 KAFKA LOCALHOST

Security:

SECURITY.PROTOCOL: PLAINTEXT

Operations

SUB greeting

A channel for sending out greeting messages

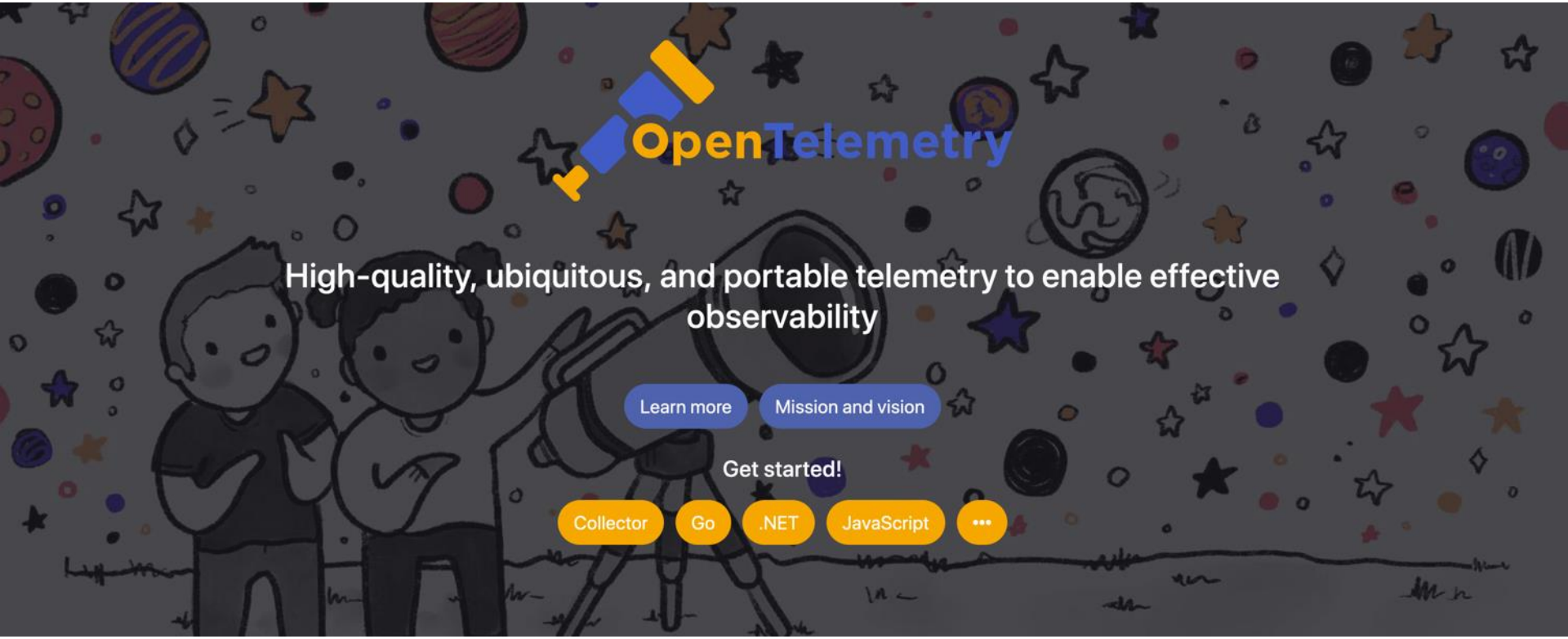
This service lets you send the 'Hello World' greeting to another service.

Operation ID sendMessage

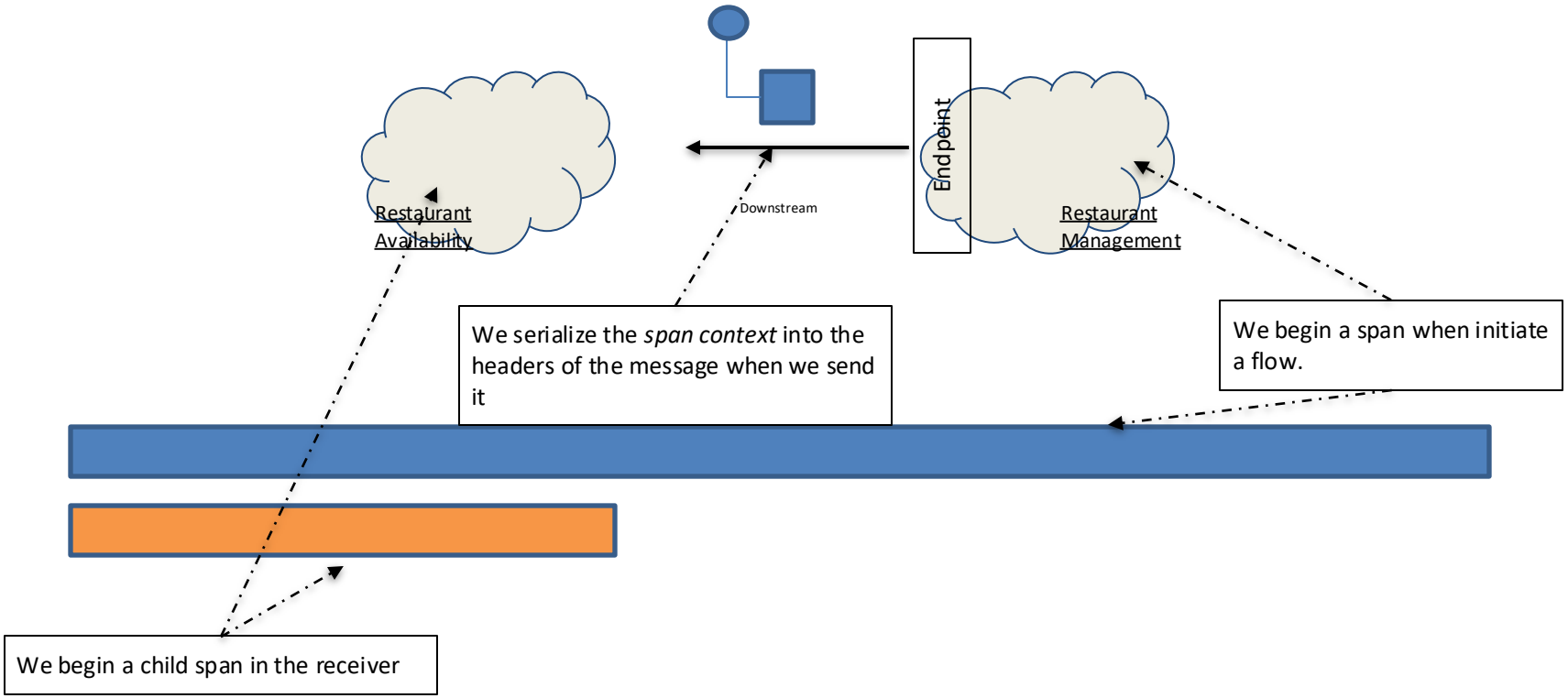
Accepts the following message:

A salutation greeting

OBSERVABILITY


OpenTelemetry
High-quality, ubiquitous, and portable telemetry to enable effective observability
Learn more
Mission and vision
Get started!
Collector
Go
.NET
JavaScript
•••

OpenTelemetry Tracing



open-telemetry / semantic-conventions

Code | Issues 275 | Pull requests 42 | Actions | Projects 5 | Security

main | semantic-conventions / docs / messaging / messaging-spans.md

crossoverJie and joaopgrassi messaging.system support pulsar (#1099)

Preview | Code | Blame | 609 lines (459 loc) · 35.7 KB ·

Semantic Conventions for Messaging Sp

Status: Experimental

ıame SHOULD only be used for the span name if it is known to be of low cardinality (cf. ge
med if it is statically derived from application code or configuration. Wherever possible, th
or aliased names SHOULD be used. If the destination name is dynamic, such as a conversa
der, it SHOULD NOT be used for the span name. In these cases, an artificial destination na
generic, static fallback like `"(anonymous)"` for anonymous destinations SHOULD be used

`s publish`
`s subscribe`
`s settle`
`publish`
`nack`
`spaces process`
`tionRequest-Conversations settle`
`) send` (`(anonymous)` being a stable identifier for an unnamed destination)

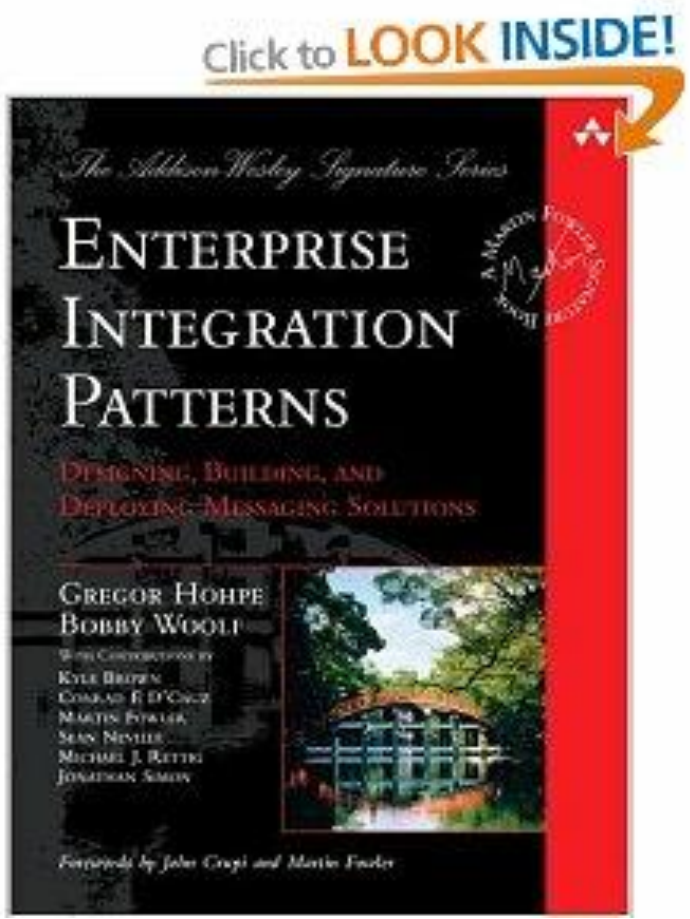
m specific adaptions to span naming MUST be documented in semantic conventions for sp

ɔes

eration types related to messages are defined for these semantic conventions:

Description
A message is created or passed to a client library for publishing. "Create" spans always ı are used to provide a unique creation context for messages in batch publishing scenario:
One or more messages are provided for publishing to an intermediary. If a single messag the "Publish" span can be used as the creation context and no "Create" span needs to b
One or more messages are requested by a consumer. This operation refers to pull-based explicitly call methods of messaging SDKs to receive messages.
One or more messages are delivered to or processed by a consumer.
One or more messages are settled.

Further Reading



Q&A