

Practical Messaging

A 101 guide to messaging

Ian Cooper

X and Hachyderm: ICooper

Who are you?

- Software Developer for more than 25 years
 - Stuff I care about: Messaging, EDA, Microservices, TDD, XP, OO, RDD & DDD, Code that Fits in My Head, C#
 - Places I have worked: DTI, Reuters, Sungard, Beazley, Huddle, Just Eat Takeaway
- No smart folks
 - Just the folks in this room

Day One Messaging

- Distribution
- Integration Styles
- Request Driven Architectures
- Event Driven Architectures
- Reactive Architectures
- Messaging Patterns

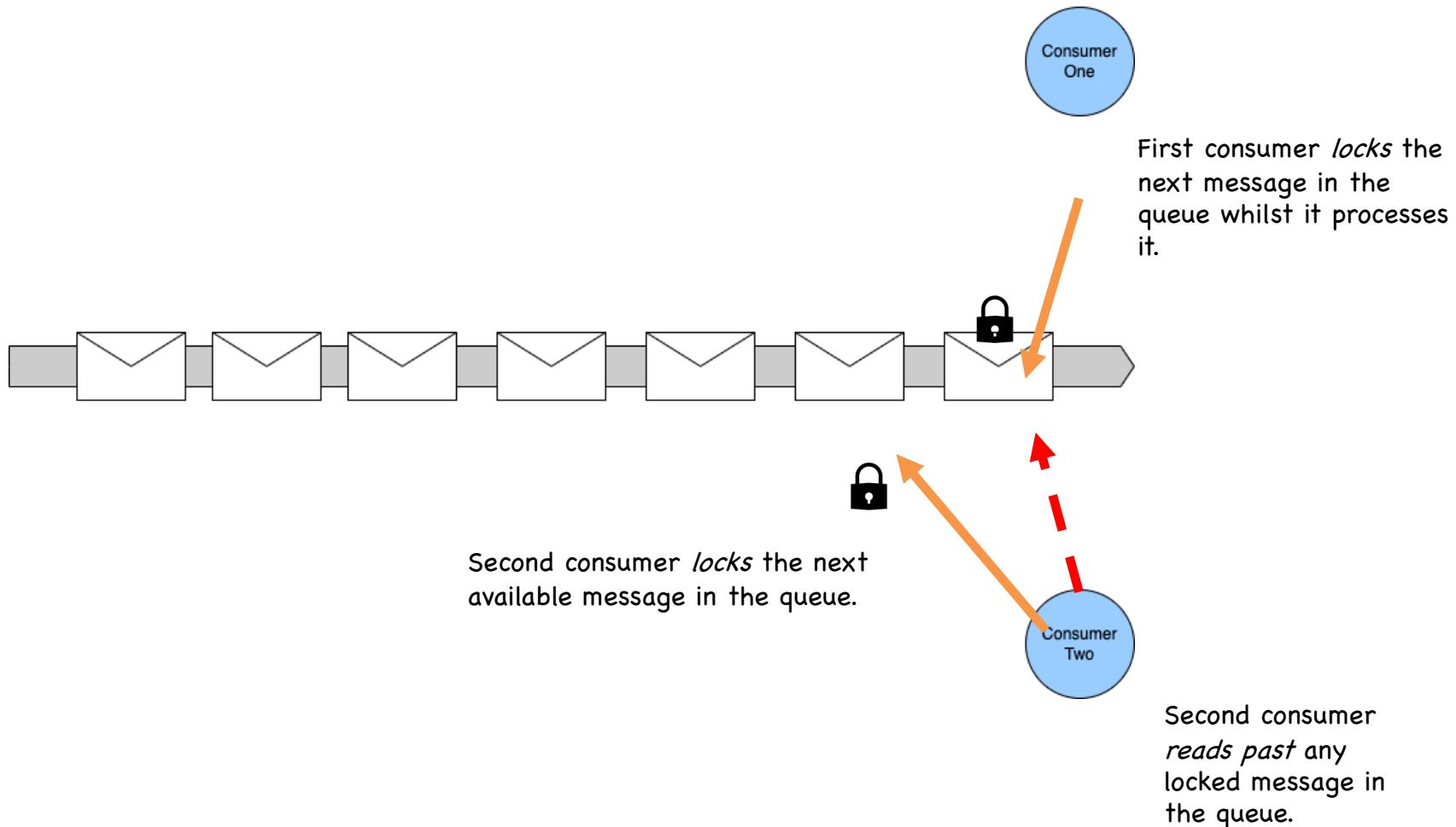
Day Two Conversations

- Queues and Streams
- Conversation Patterns
 - Activity and Correlation
 - Repair and Clarification
 - Reliable Messaging
 - Fat and Skinny
 - Conversations
- Reactive Architectures
 - Message Passing
 - Paper Based Flows
 - Flow Based Programming
- Managing Async APIs
 - Versioning
 - Documentation
 - Observability
- Next Steps

Day Two

QUEUES AND STREAMS

Queues

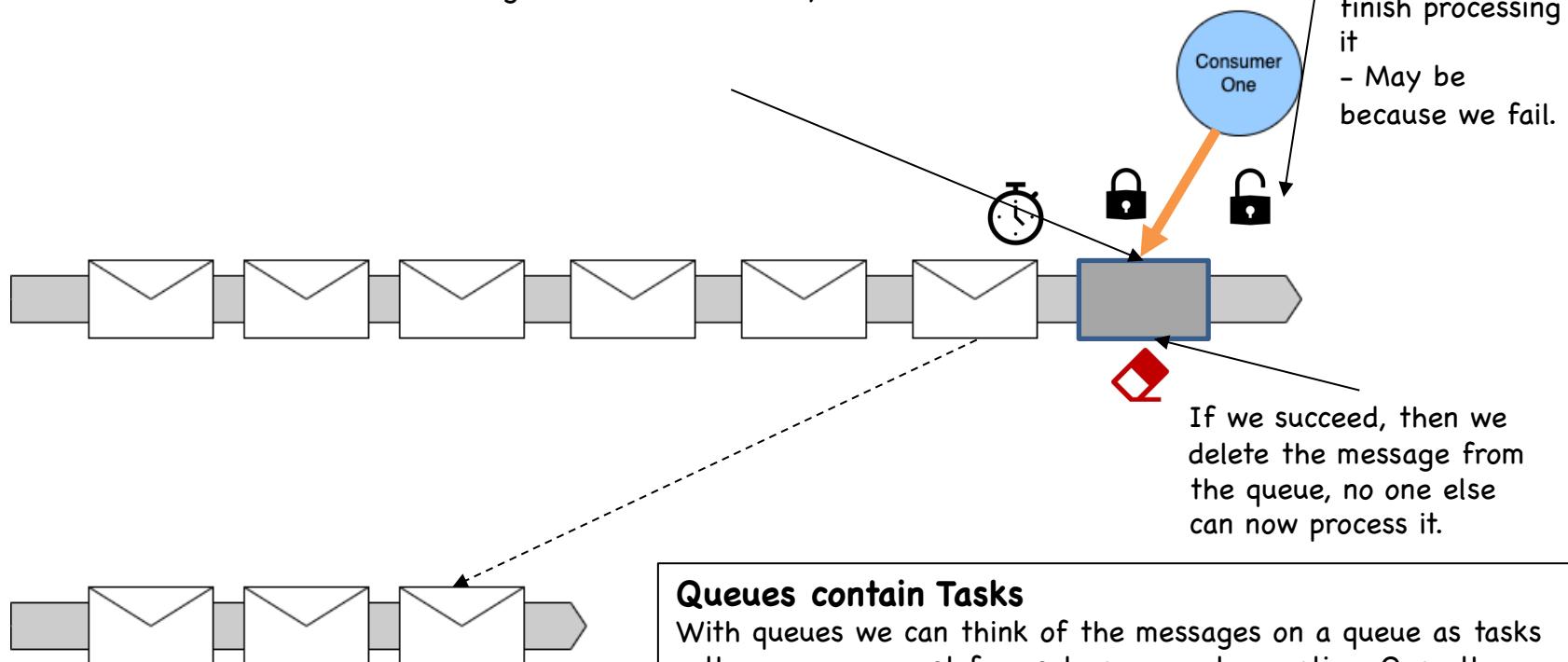


Queues

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.



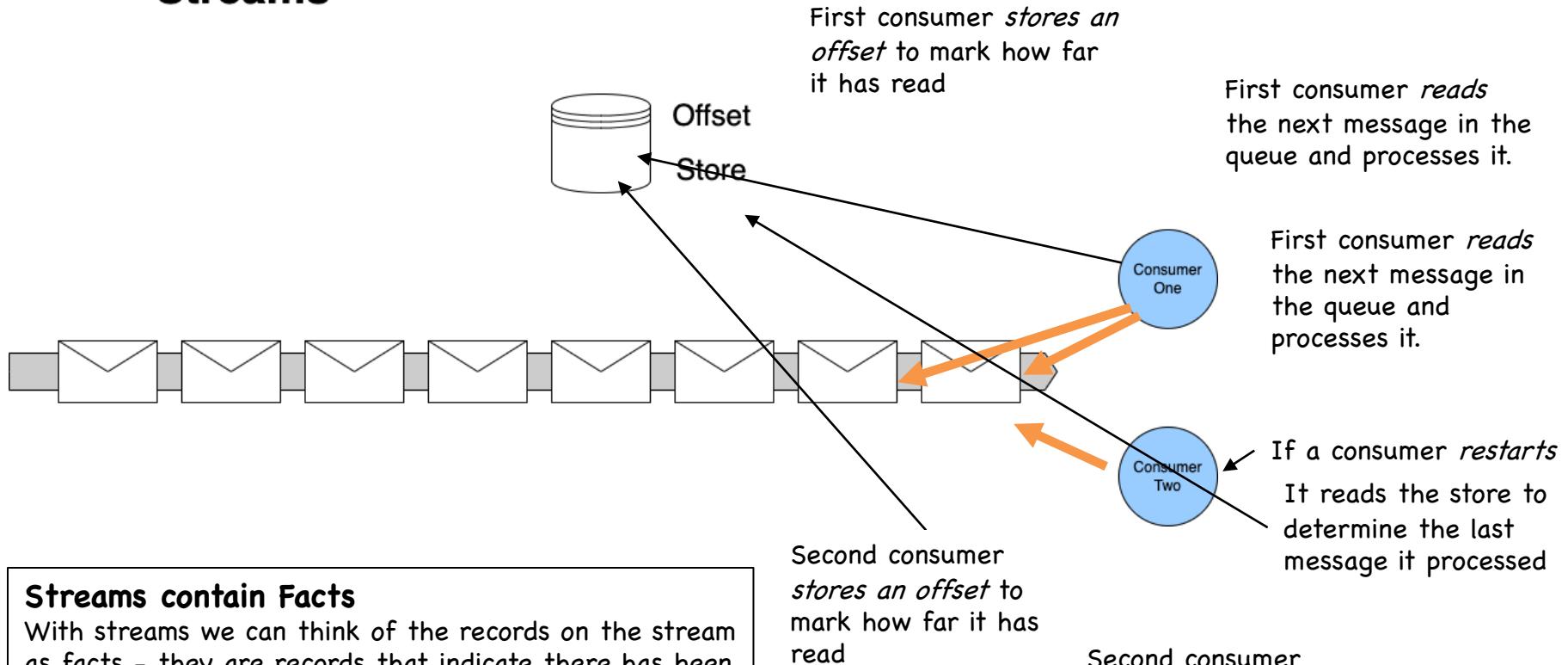
After a certain number of re-queues we may move the message to a dead-letter channel, it turns out that no one action the request within in a reasonable time frame

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



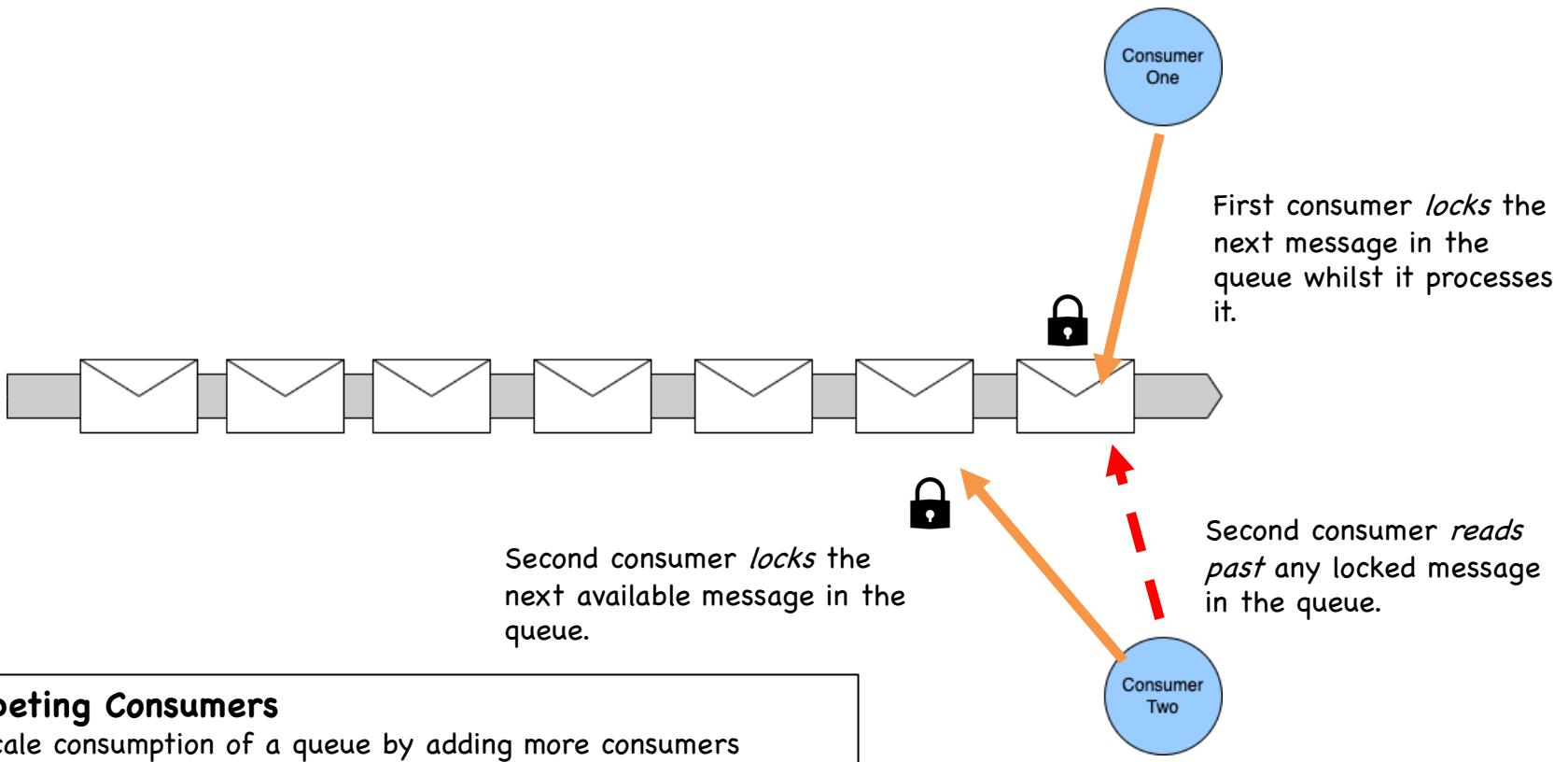
Streams contain Facts

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

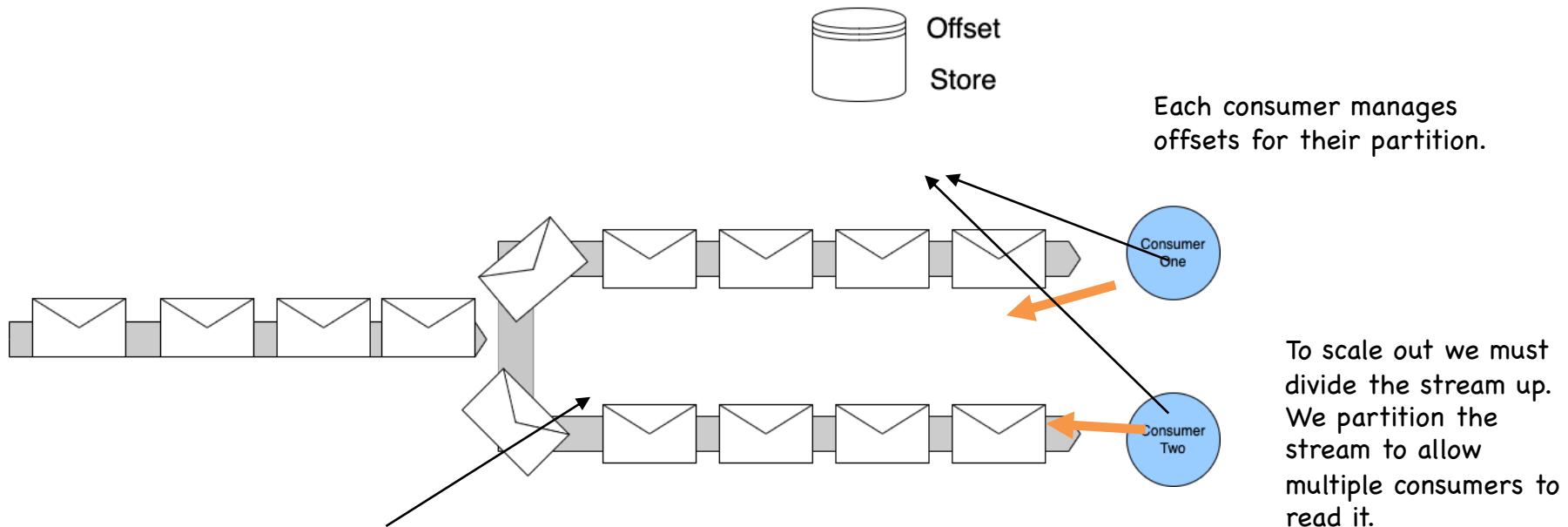
Scaling Queues and Streams

Queues



Streams

Partitions

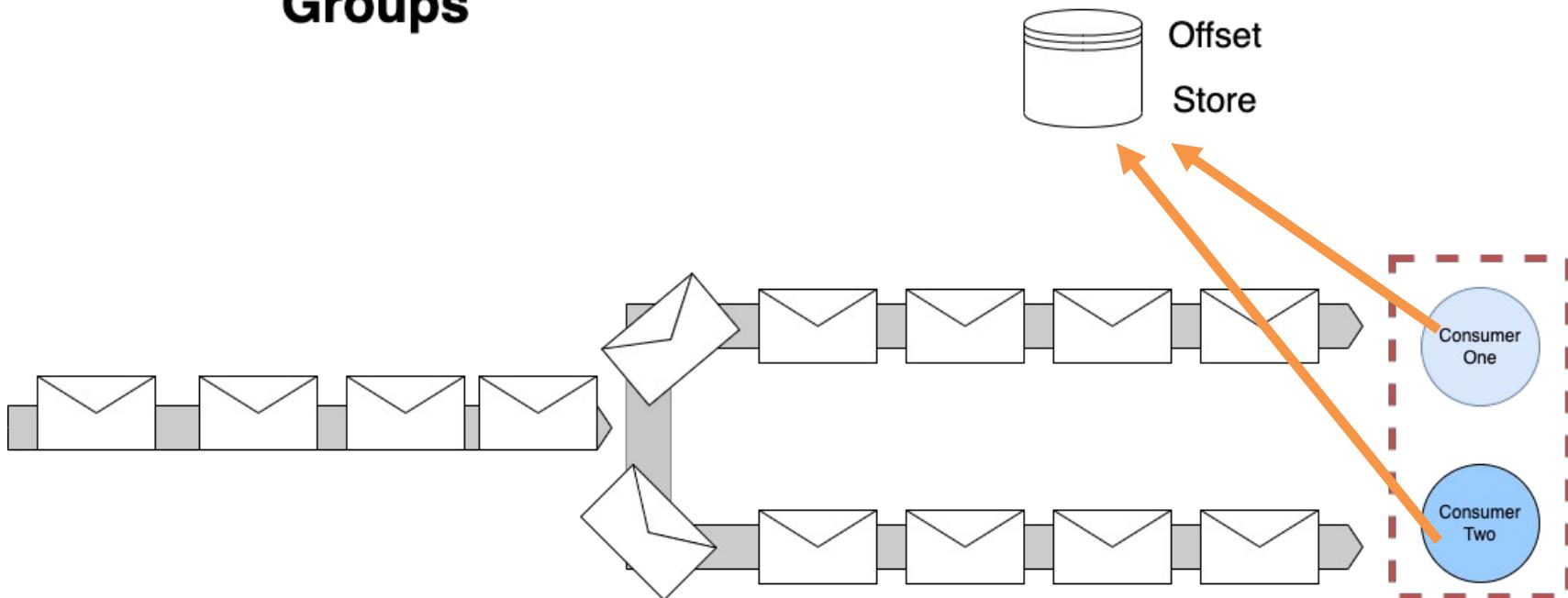


For any set of events that must be processed sequentially - all changes to one entity for example - we use consistent hashing to push messages with the same identifier to the same partition. This allows us to scale, whilst preserving our ordering.

Streams

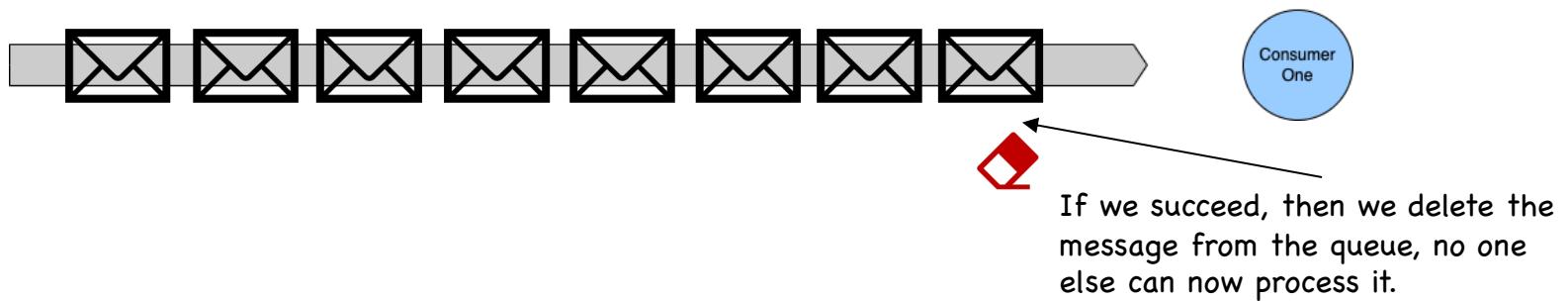
Consumer Groups

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

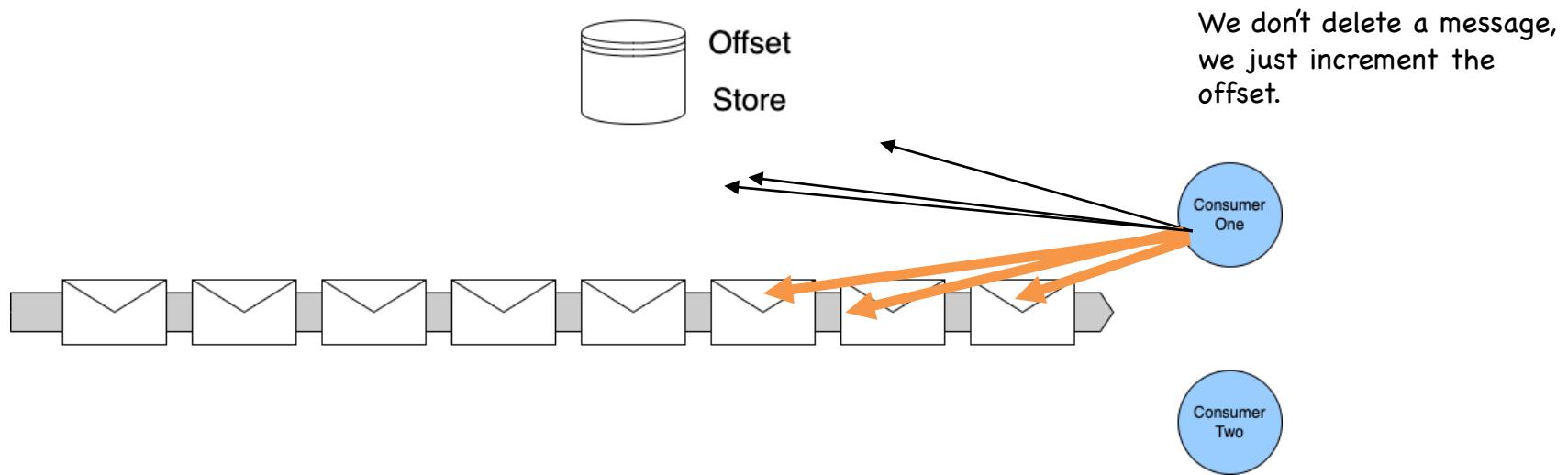
Queues



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



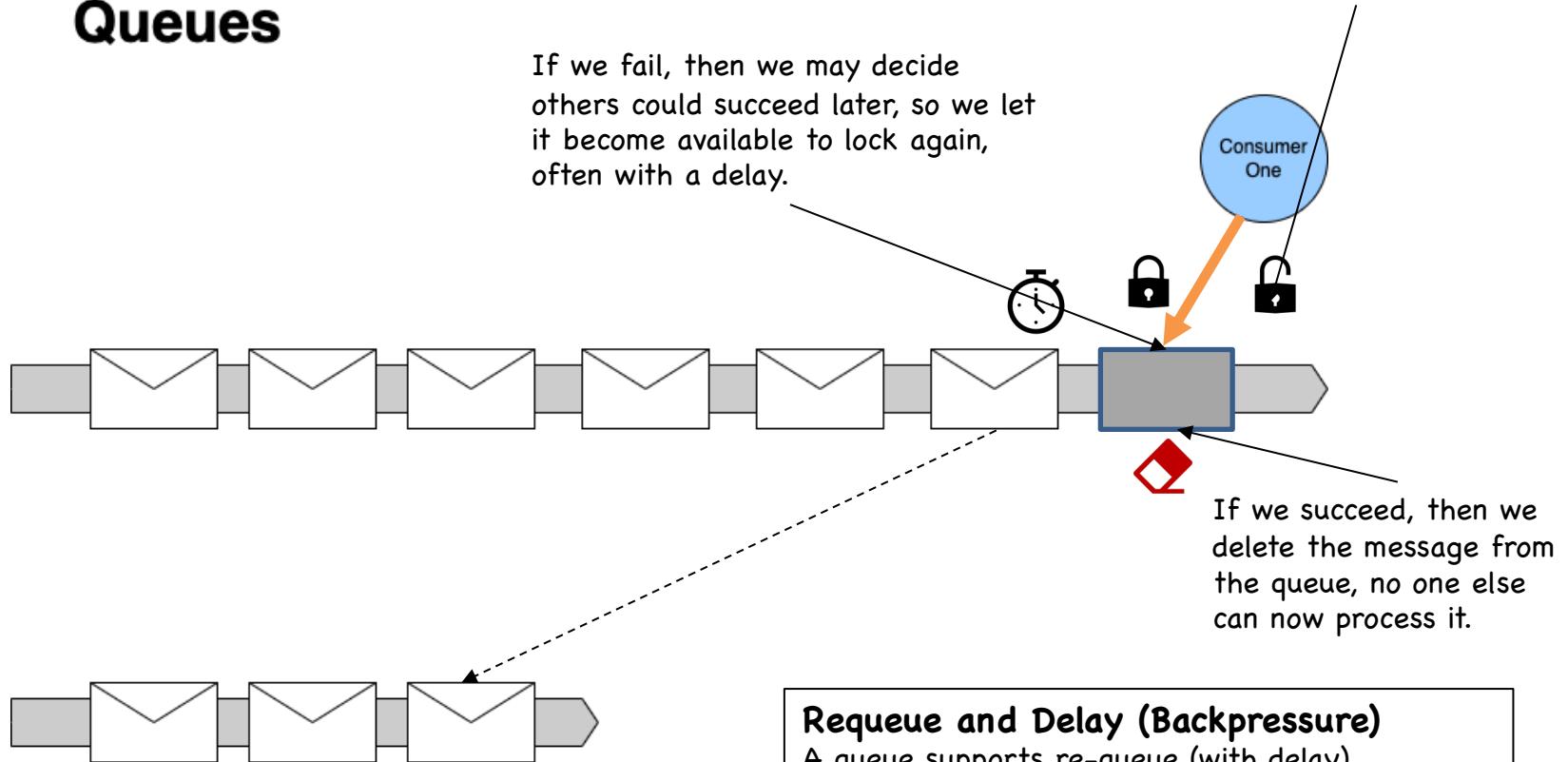
Archive and Replay

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

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



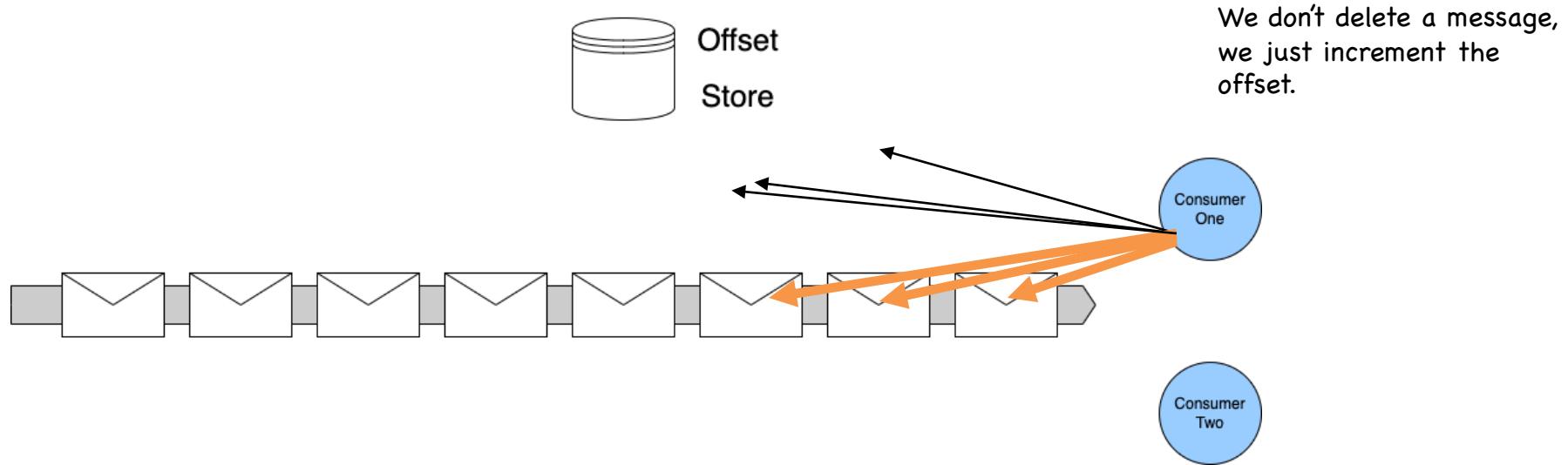
After a certain number of re-queues we may move the message to a dead-letter channel, it turns out that no one action the request within in a reasonable time frame

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 it pass

Streams



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

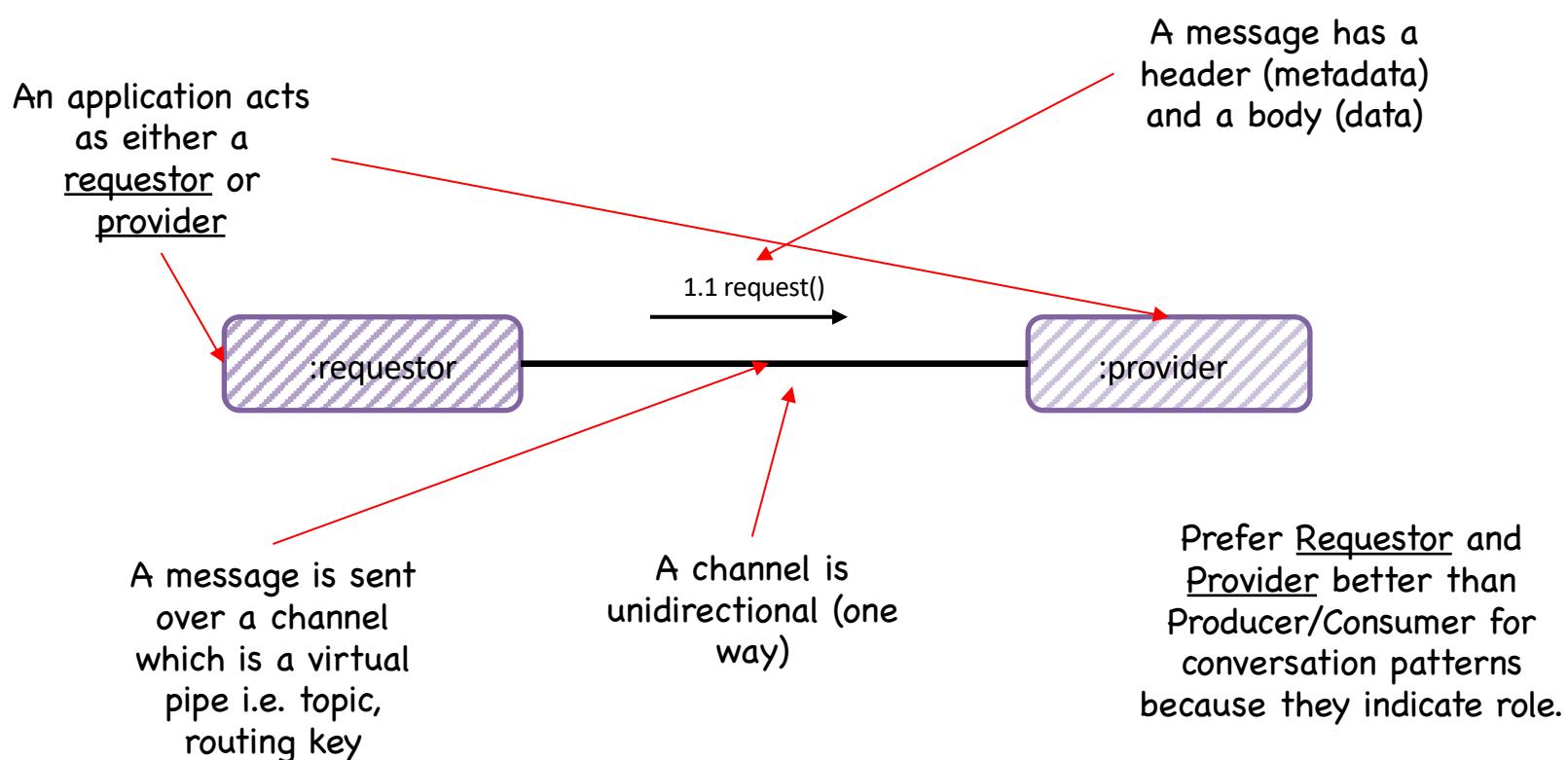
- Readme
- Slides



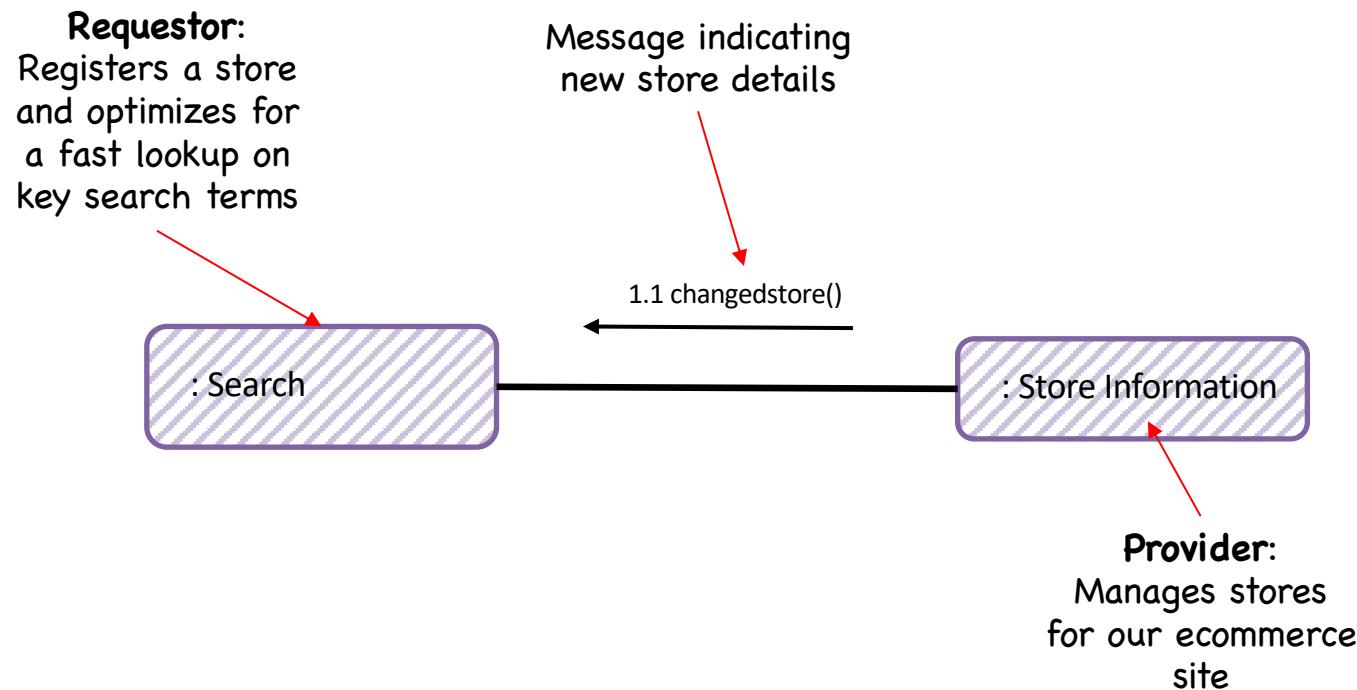
DON'T PANIC

CONVERSATION PATTERNS

Messaging Participants



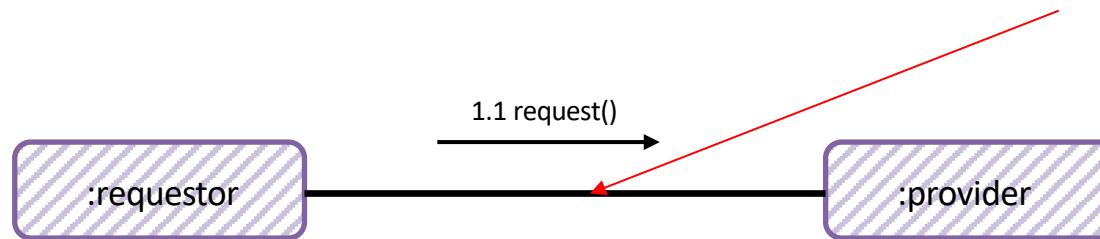
Messaging Participants



In-Only (Fire and Forget)

Typically we call this **fire-and-forget**.

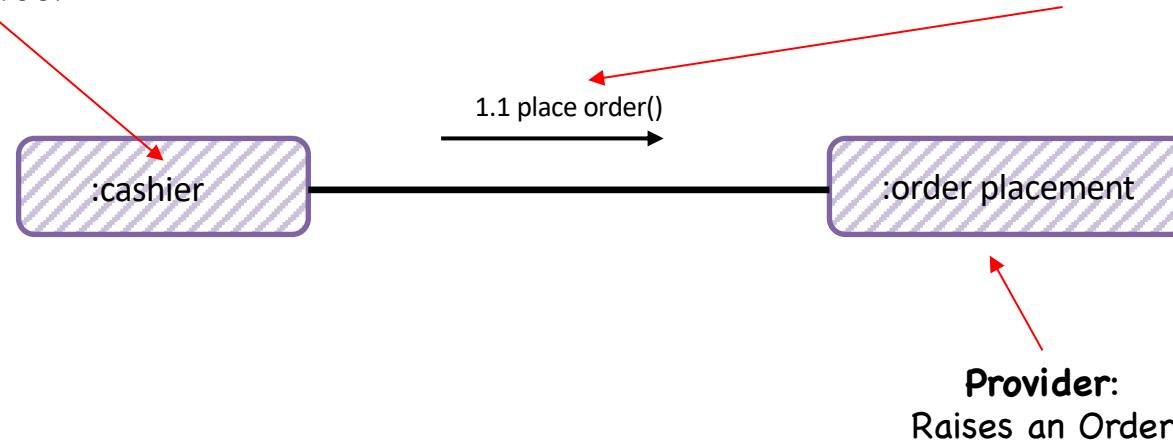
Under a In-Only pattern the requestor sends a request to the provider, but does not seek an acknowledgment of completion of the requested operation



In-Only (Fire and Forget)

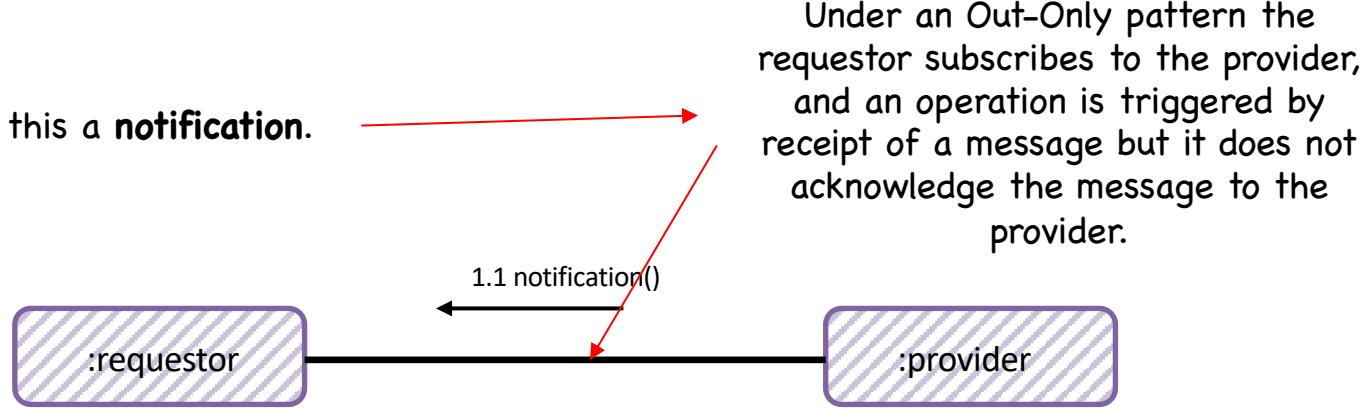
Requestor: Has a paid for basket it wants to turn into an order

Typically fire and forget is used where we are finished with our part in a flow, and transferring control. We need no response as we are done..

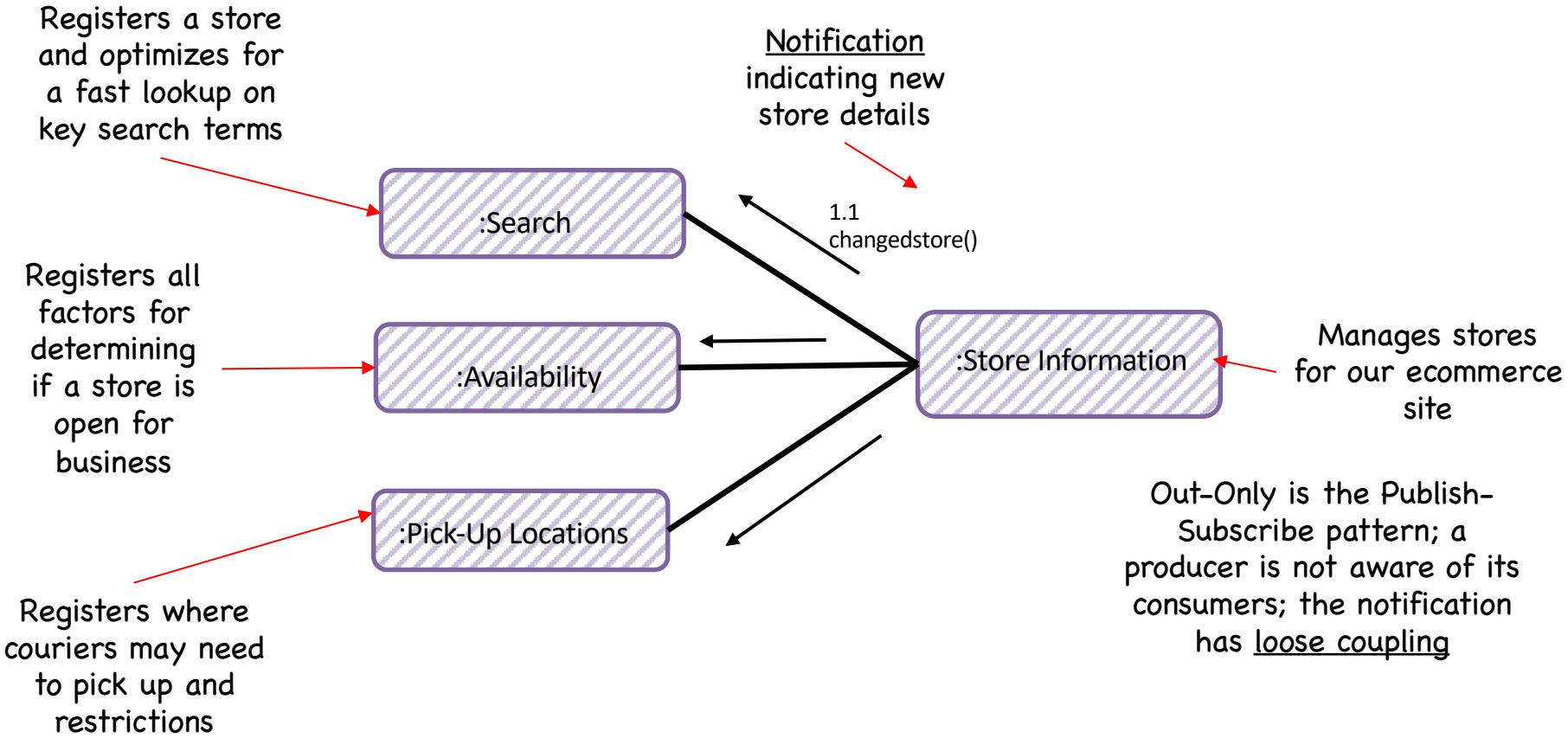


Out-Only (Notification)

Typically we call this a **notification**.

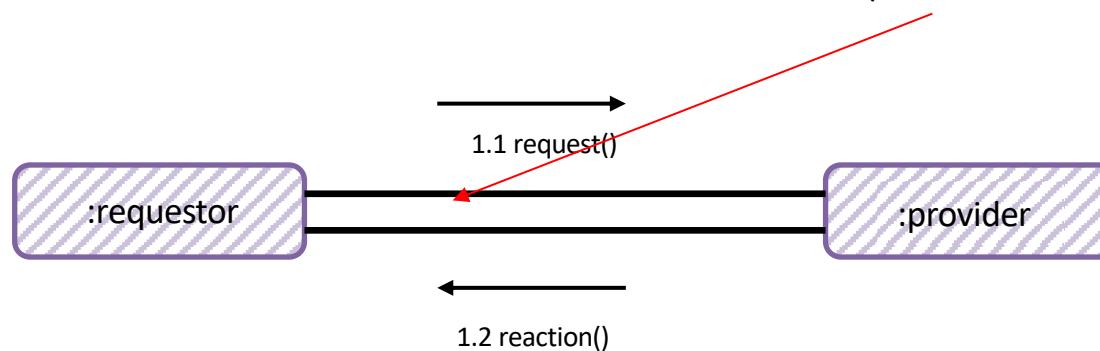


Out-Only (Publish-Subscribe)

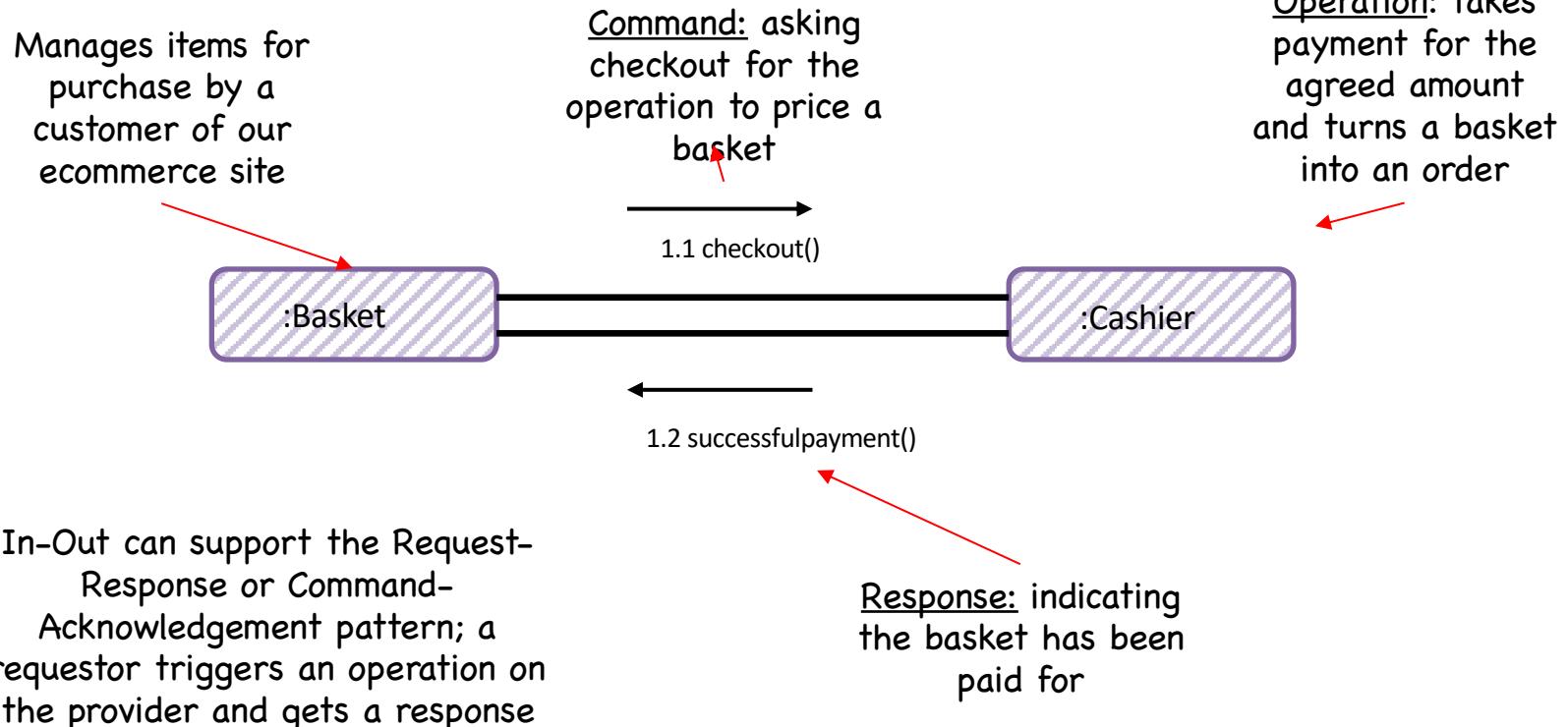


In-Out (Request-Reaction)

Under an In-Out pattern the provider receives a request from the requestor on one channel, and returns a response from the operation triggered by that message on a separate channel.



Request-Reaction (Command-Acknowledgement)



In-Out (Query-Result)

Manages items for purchase by a customer of our ecommerce site



Query asking for delivery fees for this basket

1.1 getdeliveryfees()



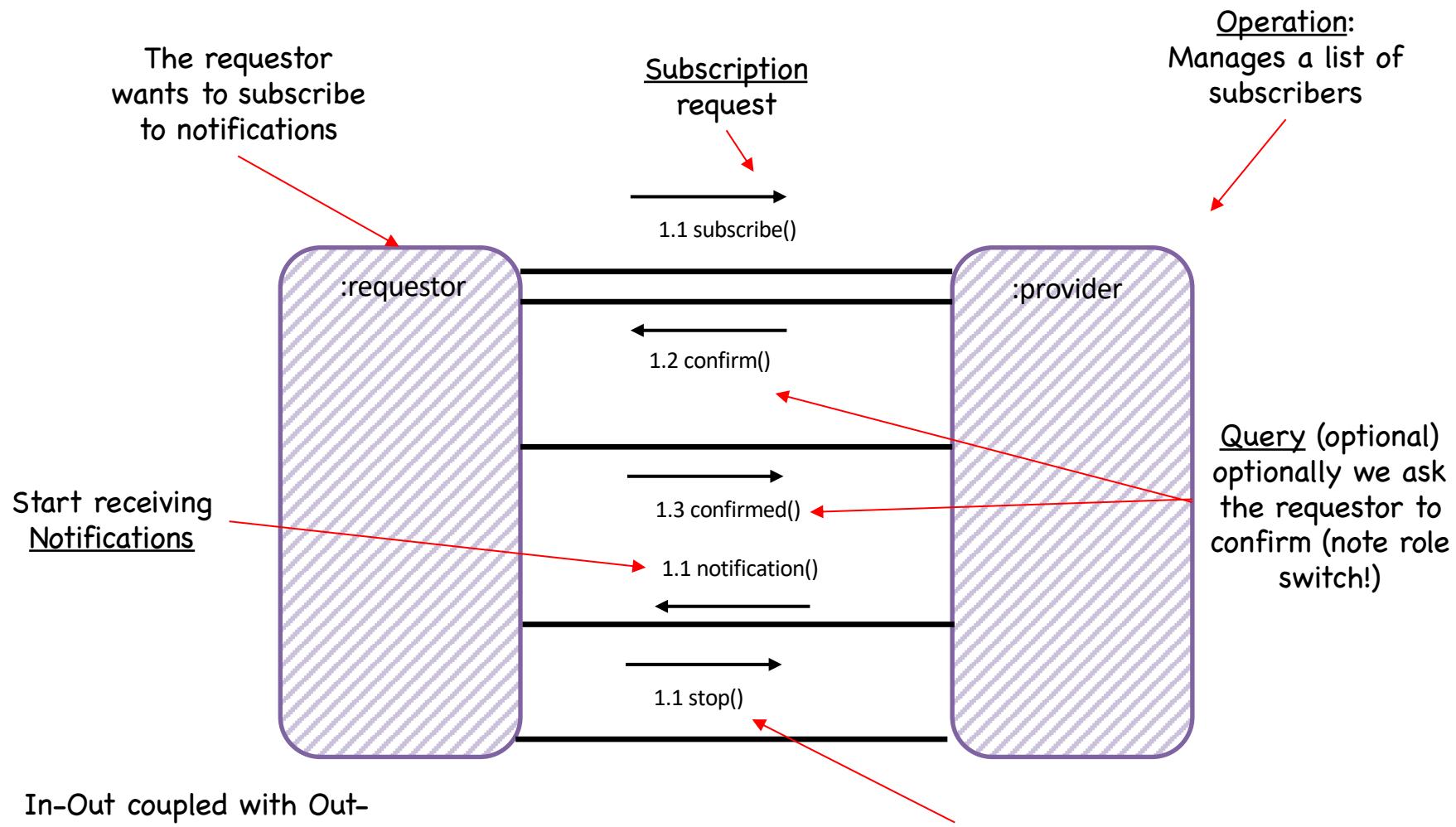
1.2 deliverfee()

Result indicating the fee for the current state of the basket

In-Out can support the Query-Result pattern; a requestor triggers a query on the provider and gets a result

Operation:
Determines the delivery fee based on store and items in the basket

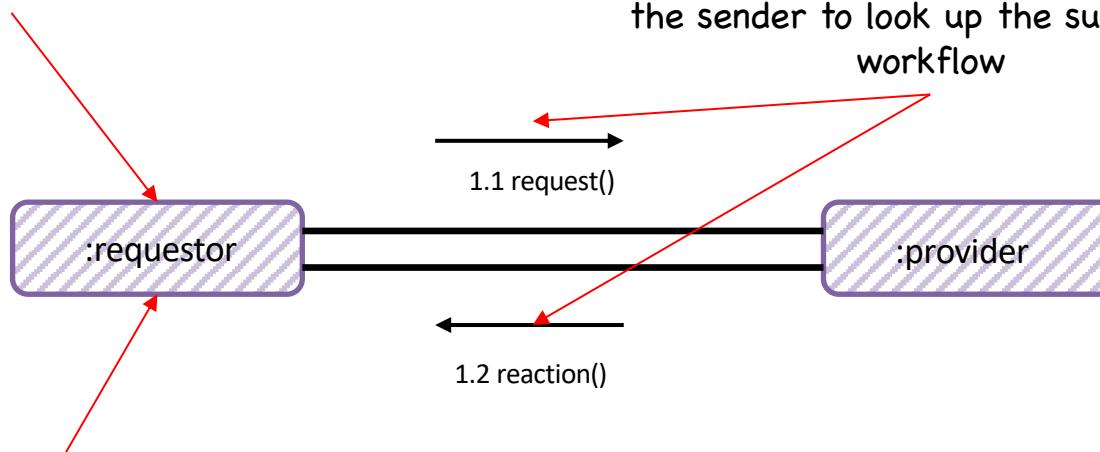
In-Out (Subscribe-Notify)



In-Out coupled with Out-
Only can support a
Subscribe-Notify pattern; a
requestor requests to
subscribe to a provider
(start/stop)

Blocking In-Out (Request-Reply)

In order to resume, we need a way to identify the suspended workflow.

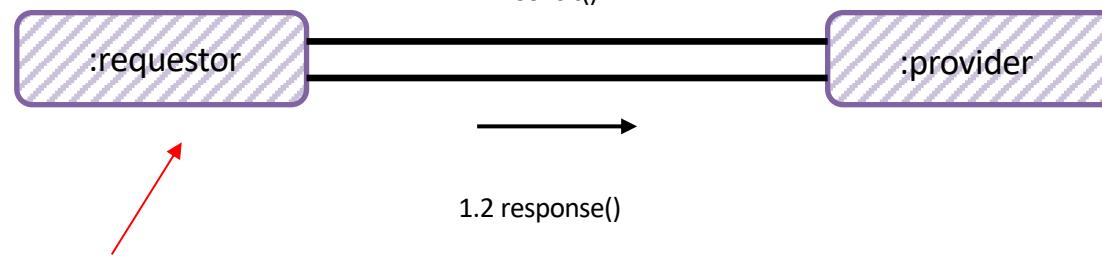


The requestor provides a unique **Reply To** channel (in the request message header/metadata) and the provider uses that channel for the response, allowing the sender to look up the suspended workflow

1.2 reaction()

Out-In (Solicit-Response)

Under an Out-In pattern a provider solicits a response from a subscriber, and await a confirmation

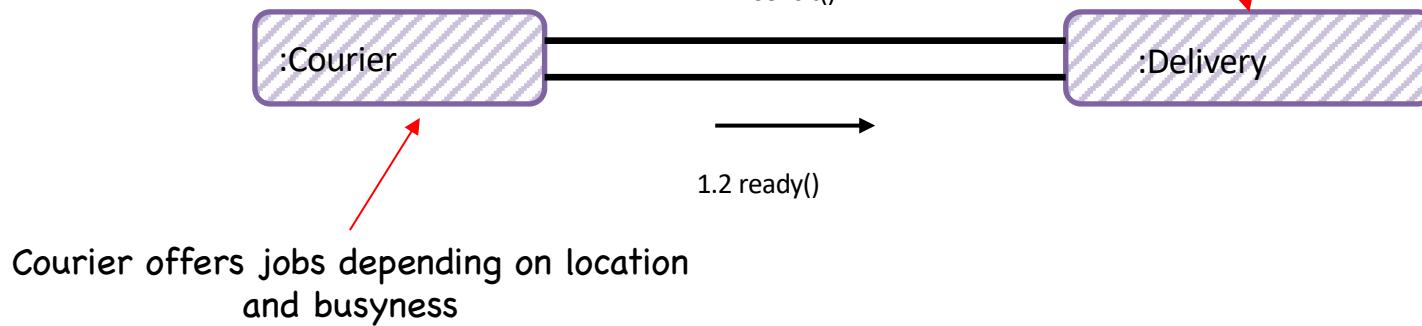


Under an Out-In pattern the subscriber confirms receipt of the provider's solicitation

Out-In (Solicit-Response)

Query to see if a courier is available to receive orders for delivery; usually Delivery service will follow with notifications of available work.

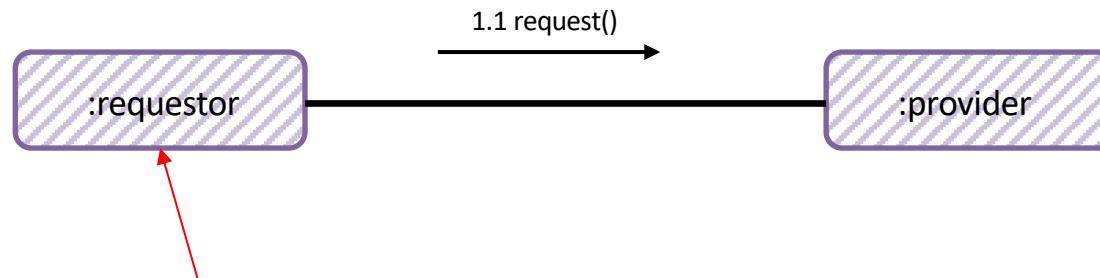
Delivery service assigns delivery request to drivers



How do we manage workflows in requestors and providers

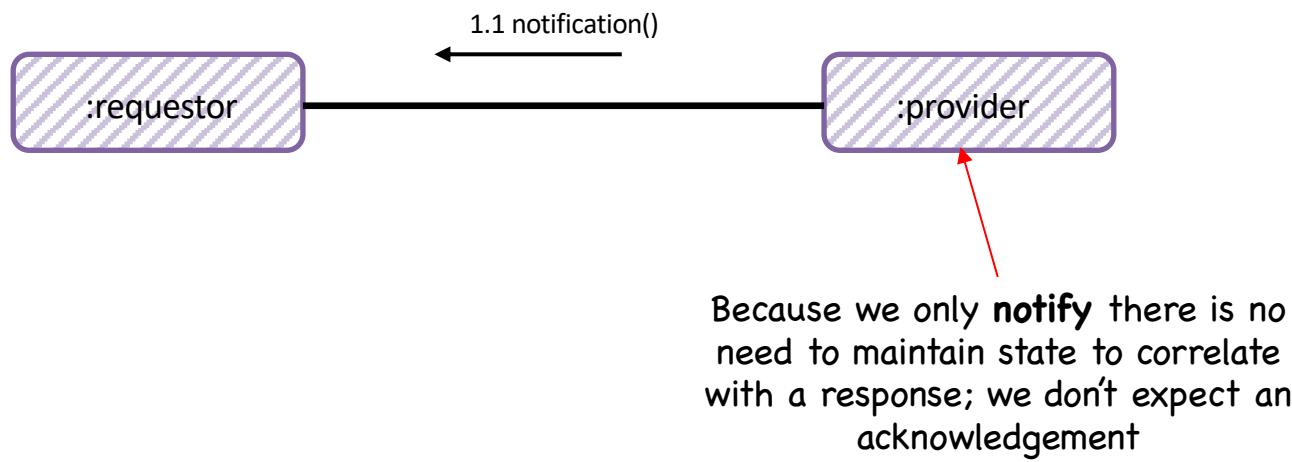
ACTIVITY AND CORRELATION

In-Only (Fire and Forget)



Because we are **fire-and-forget** there
is no need to maintain state to
correlate with a response; we don't
expect a response

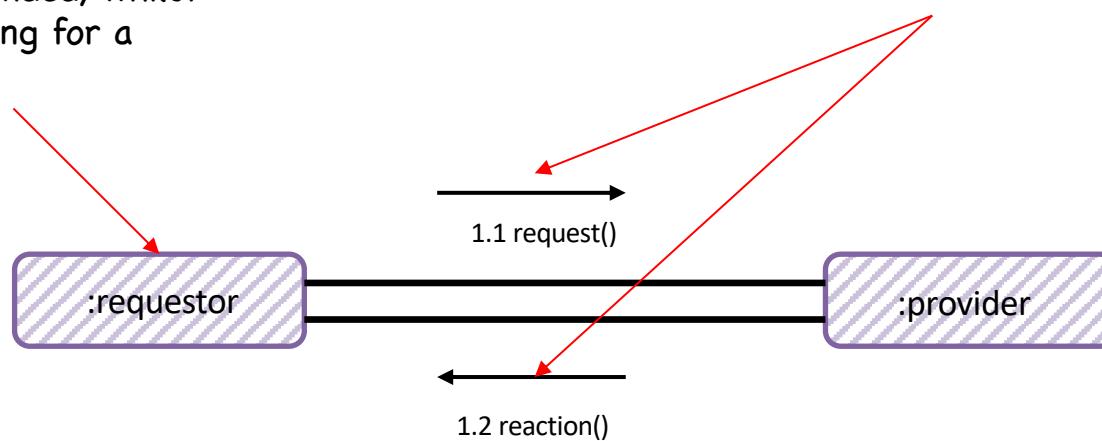
Out-Only (Notification)



In-Out (Request-Reaction)

We may resume a workflow
that we suspended, whilst
we were waiting for a
response

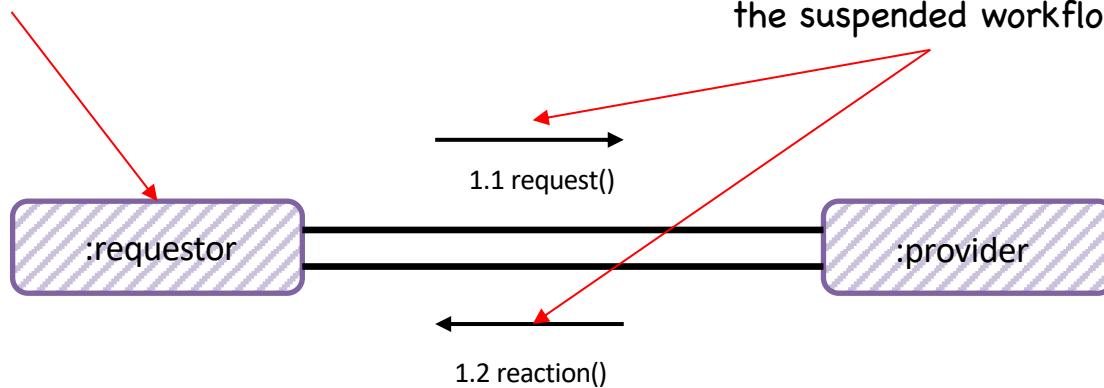
Because we expect a response to our
request we may need to save state which
we then correlate with the request



In-Out (Request-Reaction) Correlation Id

In order to resume, we need a way to identify the suspended workflow.

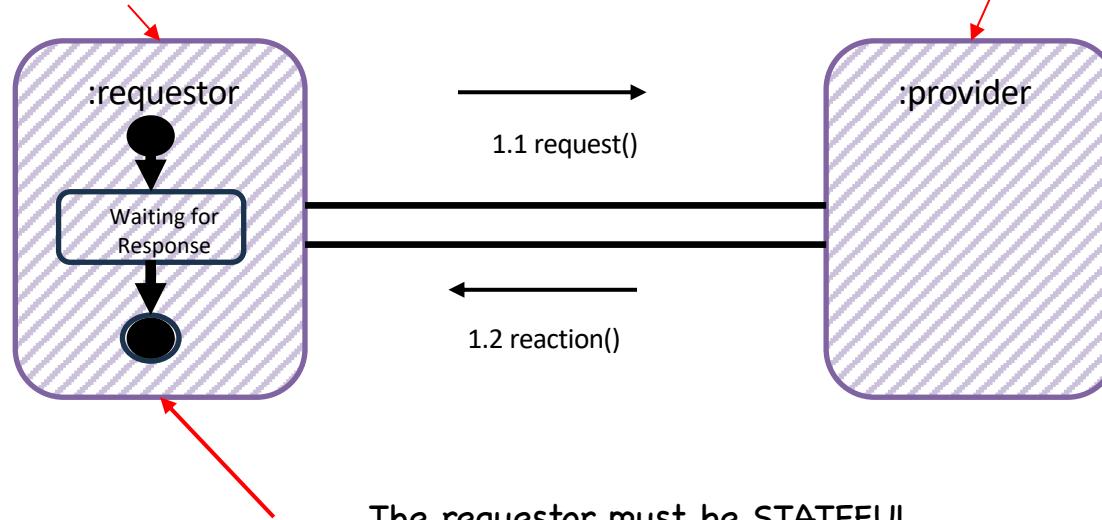
The requestor adds a **Correlation Id** (in the request message header/metadata) and the provider returns that in the response, allowing the sender to look up the suspended workflow



In-Out (Request-Reaction) Activity

Because the flow is asynchronous, the requestor enters a waiting for response state – as it cannot complete its own operation until it gets the response. Pat Helland calls this an **Activity**.

The provider can remain stateless as it only needs to return a response to the requestor

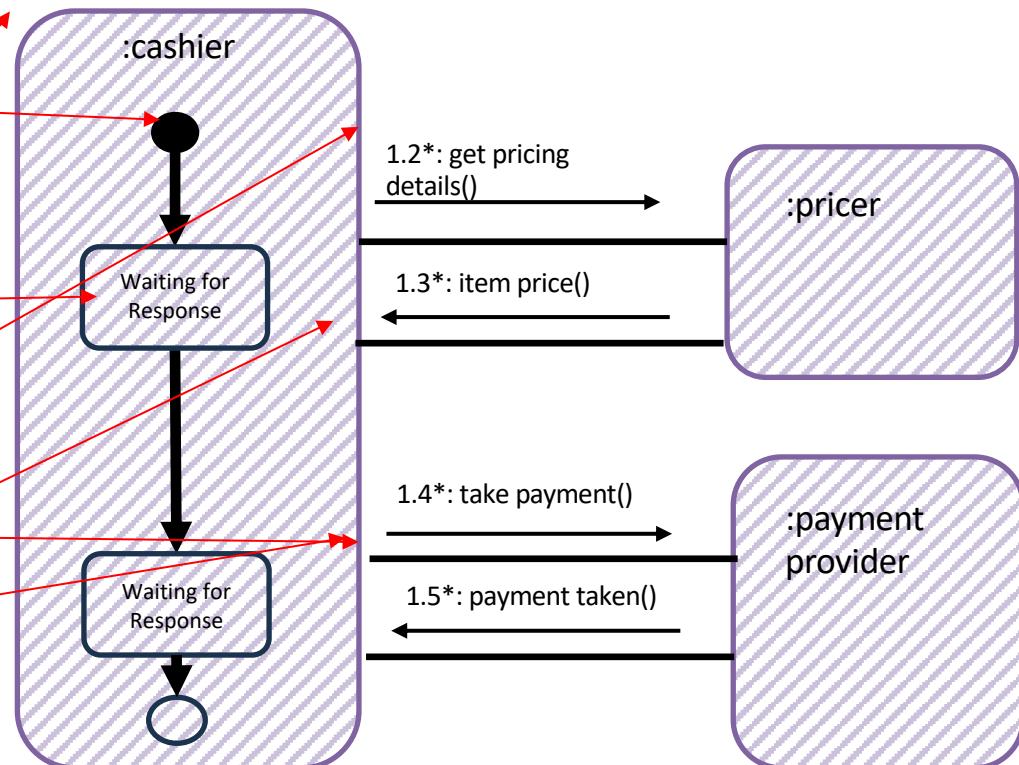


In-Out (Request-Reaction) Activity

An **activity** is the code that runs in response to receiving a message.

An activity also stores the state associated with the activity – what did we do in our last turn? It is keyed by a Correlation Id or Reply To channel.

An activity also knows the messages that it has sent and received, so it can be reconstructed.

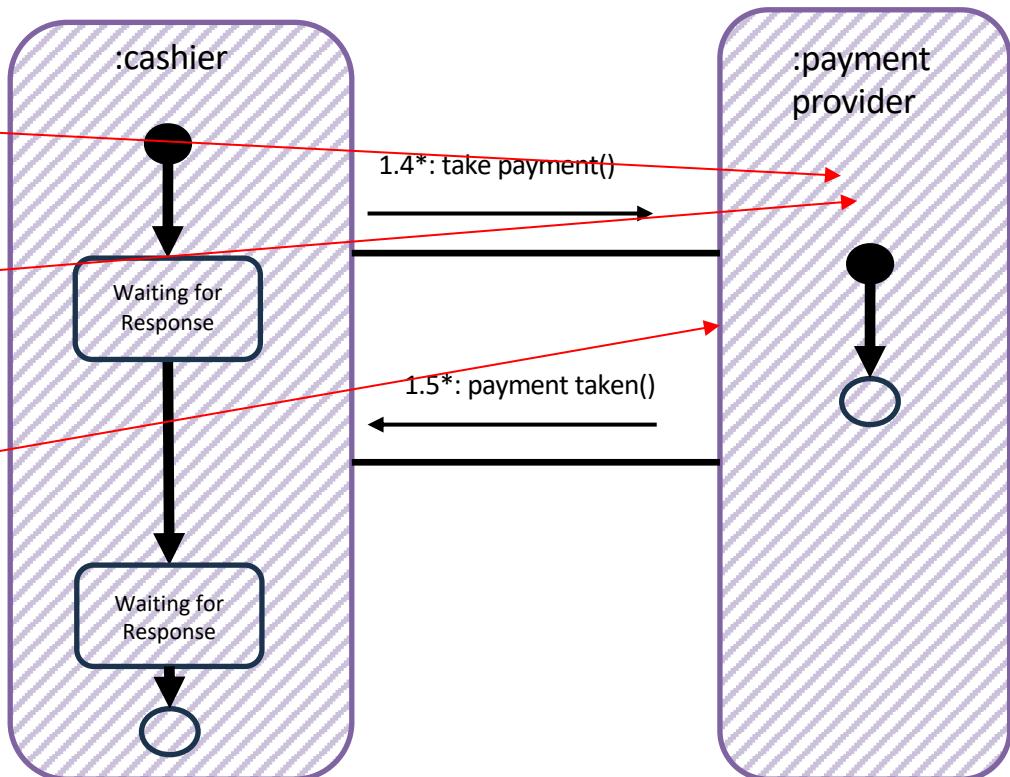


In-Out (Request-Reaction) Activity

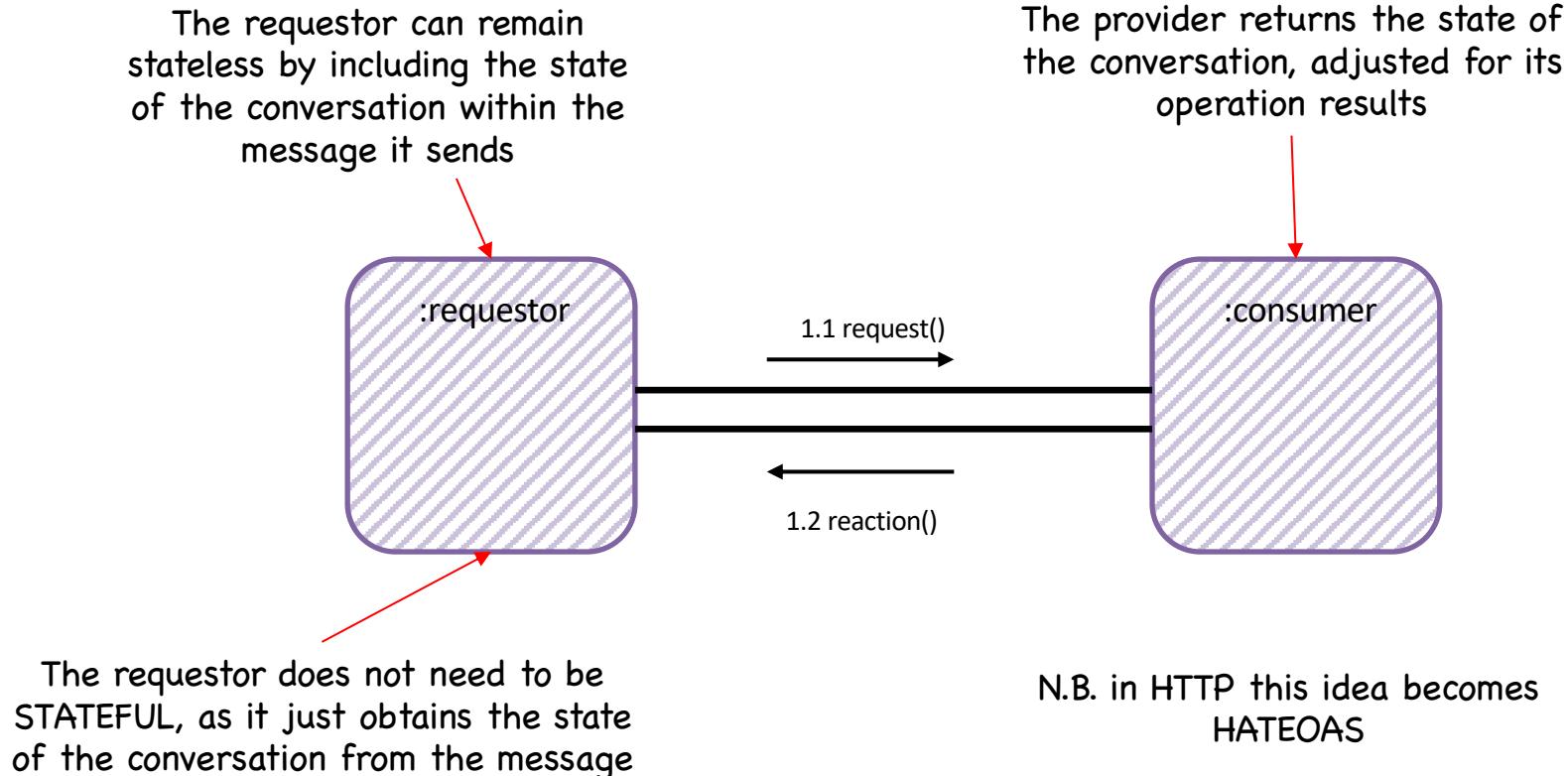
An **activity** is the code that runs in response to sending a message.

An activity *may* store state associated with the activity – but also may be *stateless* (Message as the Engine of State.)

Each activity also knows the messages that it has sent and received, so the conversation can be reconstructed.



In-Out (Request-Reaction) Message As Engine of Application State

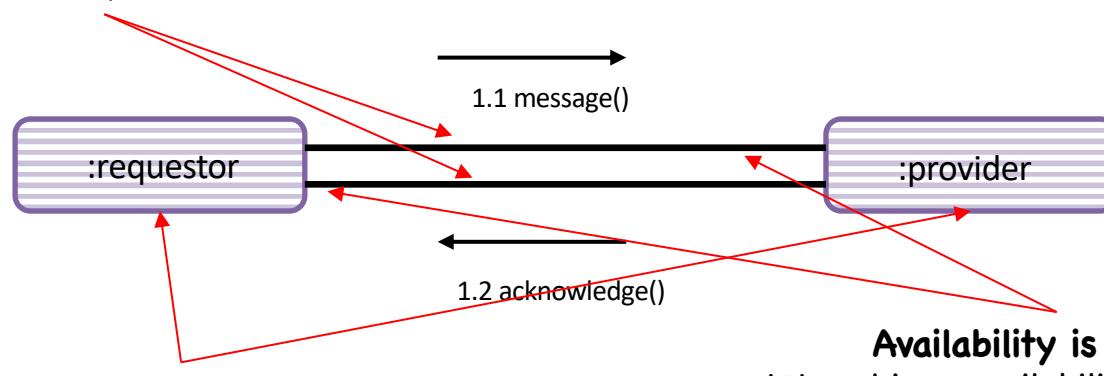


What happens when it all goes wrong

REPAIR AND CLARIFICATION

Failure Scenarios

Loss of messages, out of order messages, incorrect message schema, loss of broker
- a distributed system can fail in many ways.

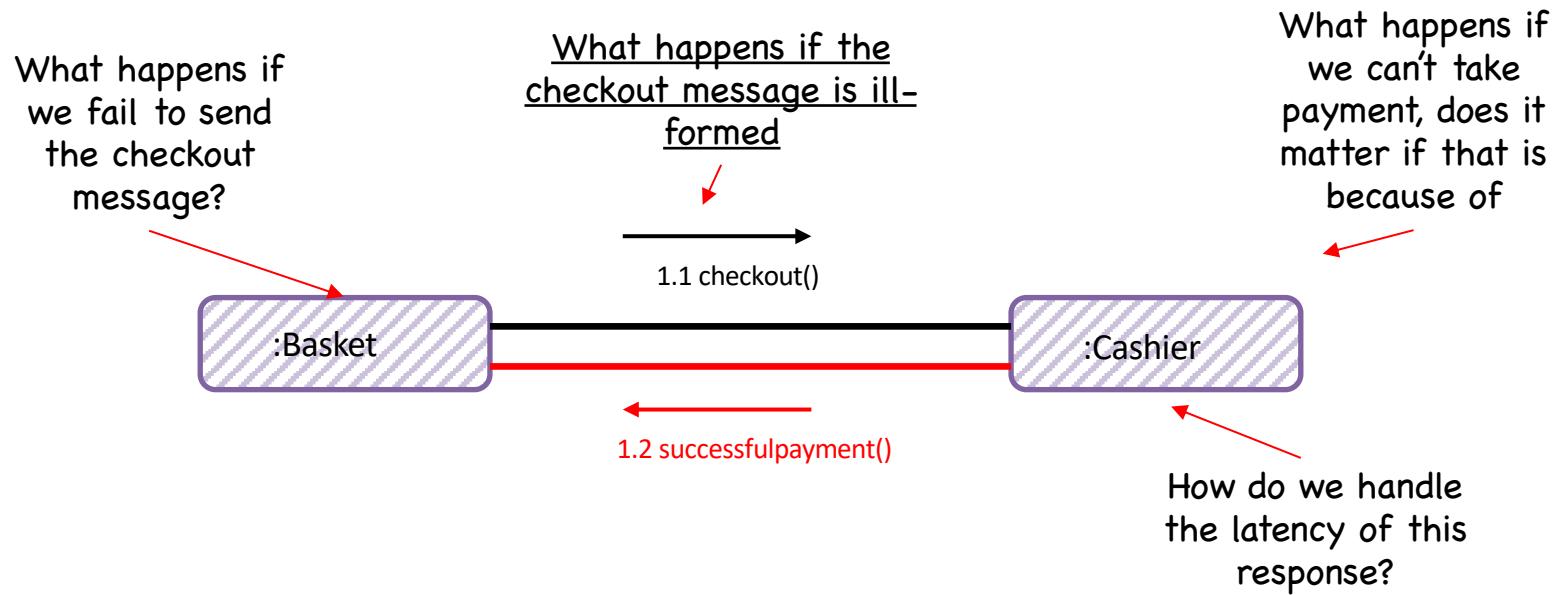


Consistency

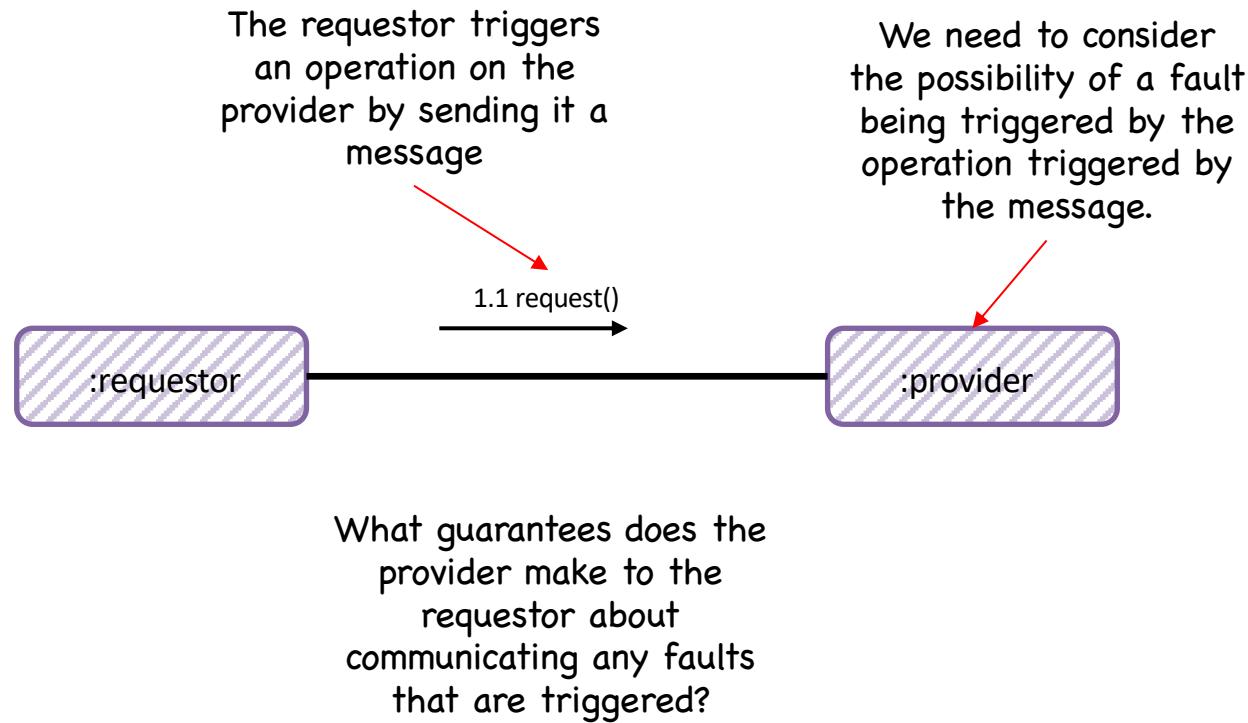
How can we know that a consumer received a message or processed it?

Availability is Latency

We achieve availability by temporal decoupling but this implies that we must cope with latency, even without failure.

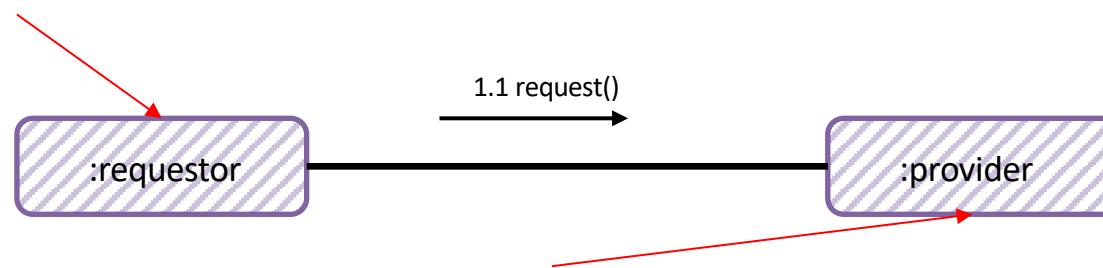


Repair and Clarification



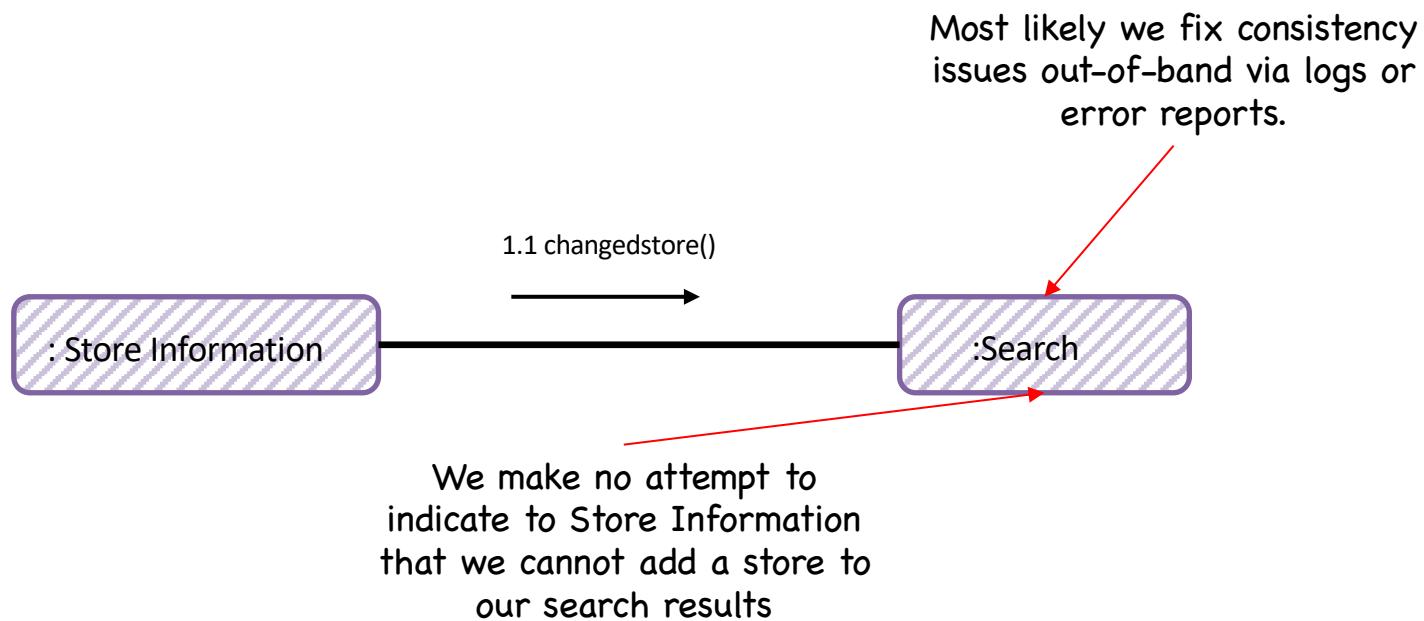
No Fault

From the requestor's perspective, faults are an application issue for the provider, and not its concern



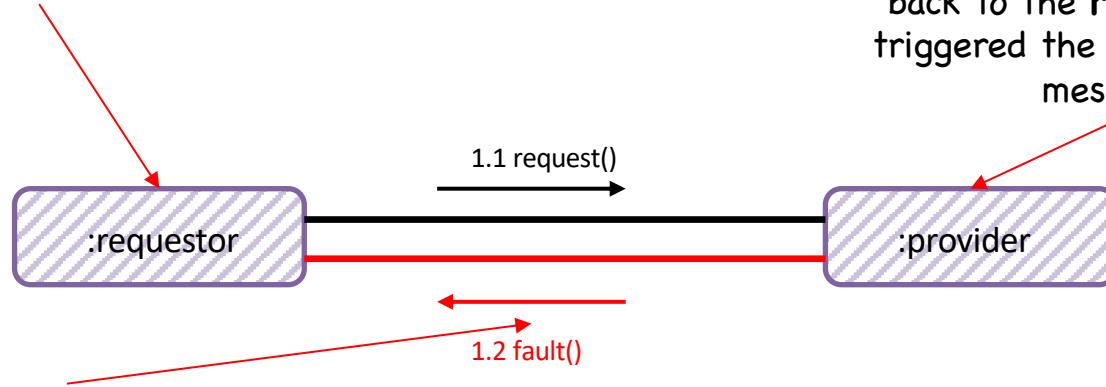
Under a No Fault pattern the provider makes no attempt to communicate triggered faults from the operation to the requestor

No Fault



Message Triggers Fault (Robust In-Only)

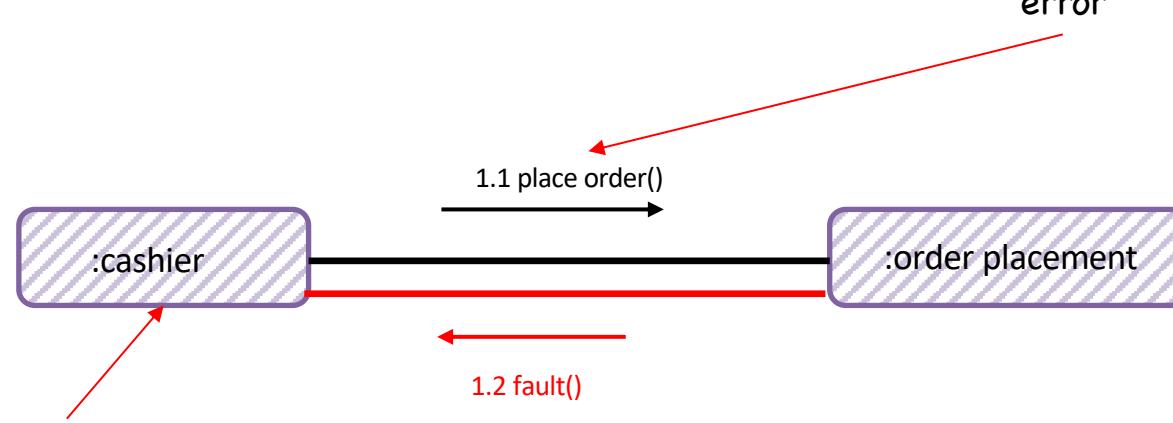
Assumption here is that the requestor can take action on receipt of the fault; but does not normally take a response.



The message must have the opposite direction and go back to the requestor

Under Message Triggers Fault pattern a **provider** propagates triggered faults from the operation back to the **requestor** that triggered the operation via a message

Message Triggers Fault

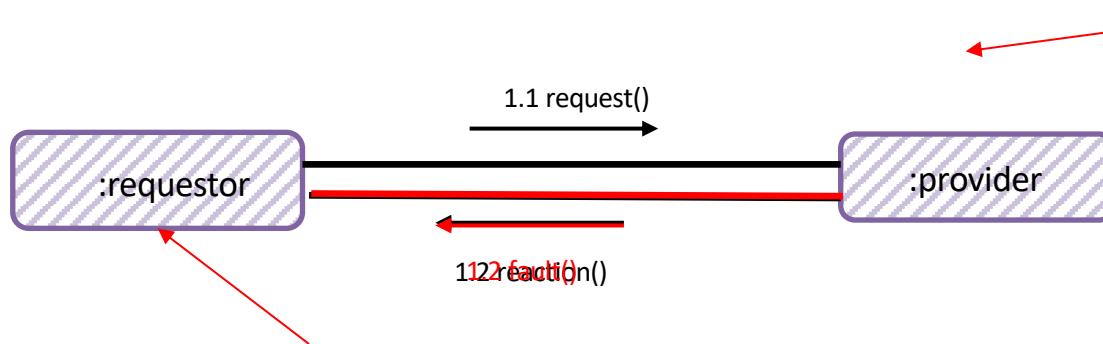


The requestor does not need to acknowledge success, but on a fault may need to take other action such as a refund

If Order Placement cannot place the order due to a fault we may raise an error

Fault Replaces Message

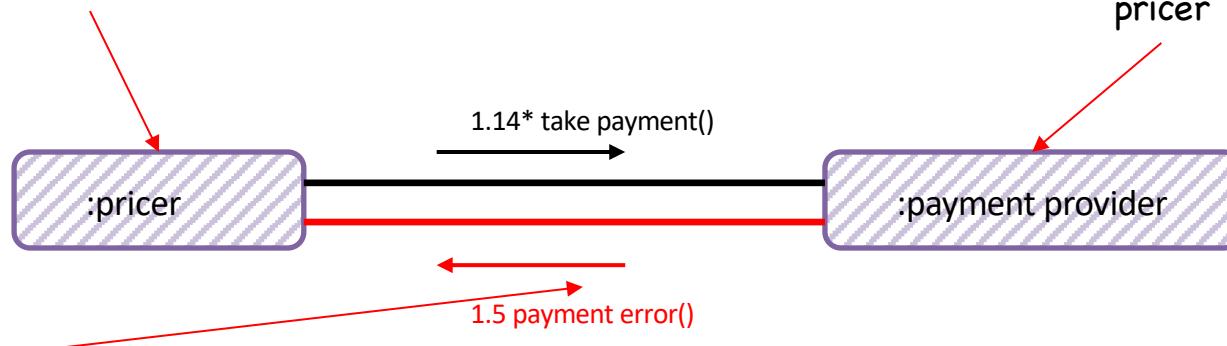
Under the Fault Replaces Message pattern a provider propagates faults triggered by an operation by switching to a fault flow, replacing any message subsequent to the first with a fault.



The requestor can handle the error; the error replaces the existing response

Fault Replaces Message (Robust In-Out)

Assumption here is that the pricer can orchestrate a new flow, such as asking for an alternate payment method, or cancel the order.

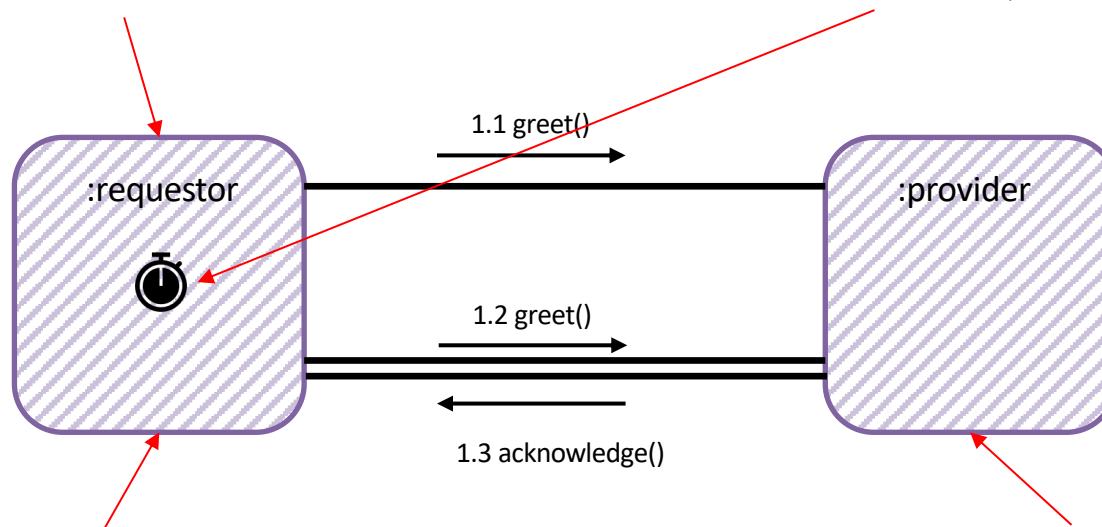


Under Fault Replaces Message pattern the payment **provider** would signal errors taking a card payment back to the pricer

The message should indicate why the payment failed. This might be an issue with the payment provider but it also might be an issue like an invalid card or insufficient funds

The requestor may not receive an expected response from a provider. What can it do?

The requestor can set a timeout within which to receive a response.



The requestor can choose to retry if it does not receive a response within that time window.

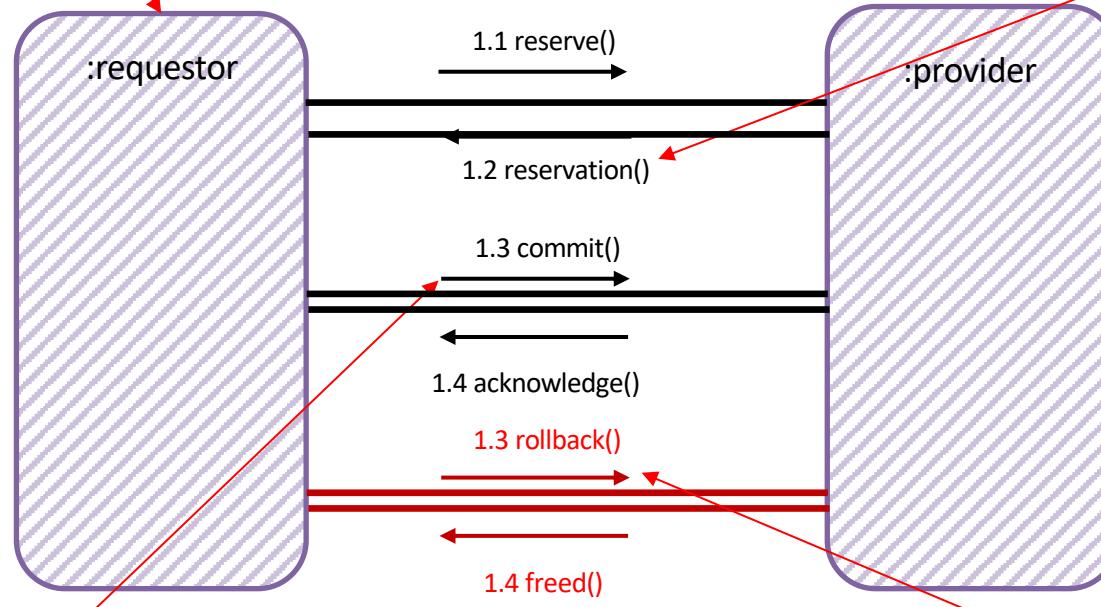
Because we might send a message twice, the operation on the consumer must be idempotent, or the consumer must de-duplicate already seen messages

Tentative Operations

The requestor may not know if the provider will be able to succeed, and might not want to proceed without knowing that

The requestor asks if the operation is possible, and reserves the resources to perform it.

The provider reserves the resource – usually with a timeout – during which time it awaits a message to commit or rollback



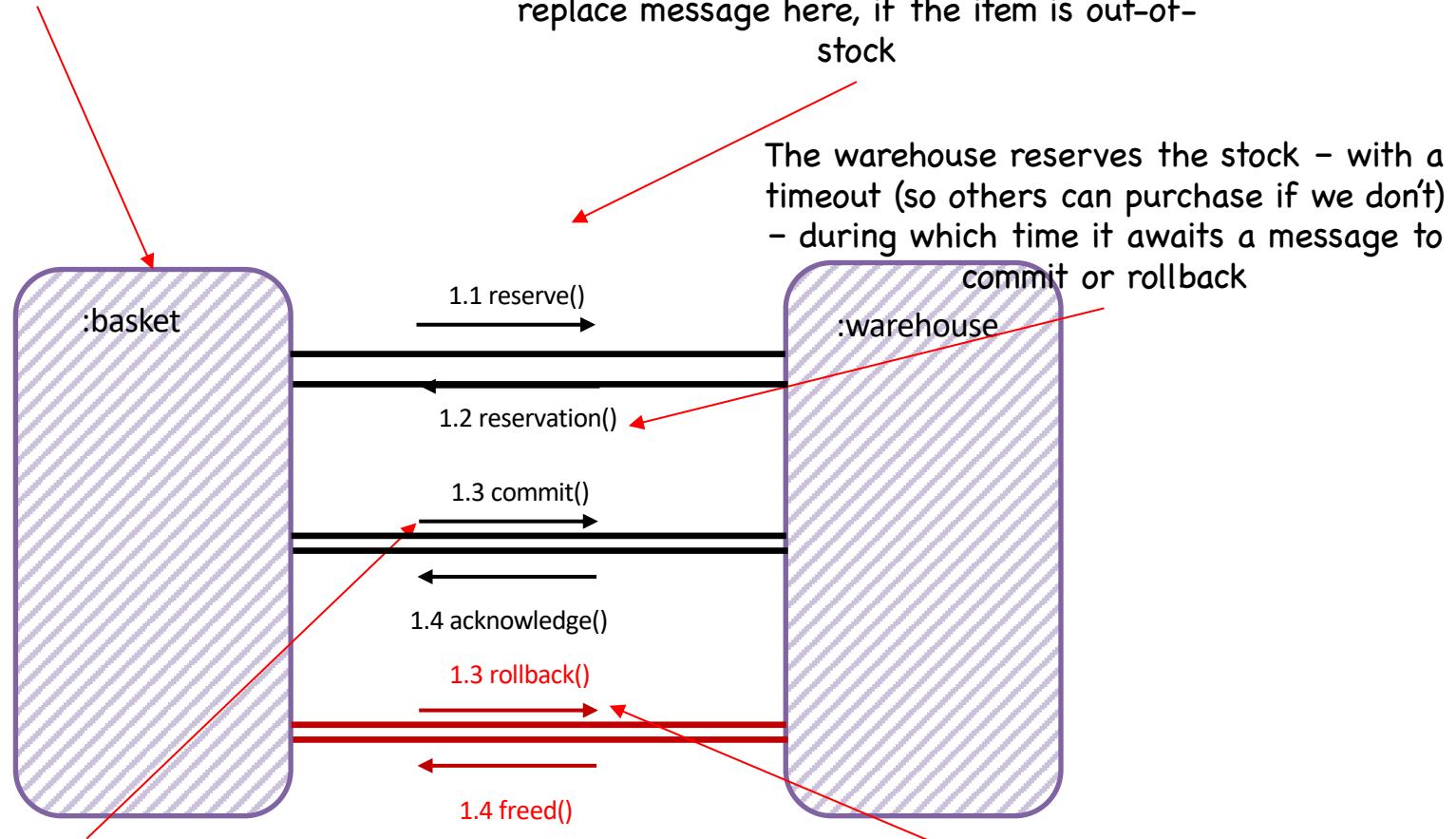
If the requestor completes its work, it then commits to ask the provider to allocate the reserved capacity.

If the requestor fails, it then rolls back to ask the provider to free the reserved capacity.

Tentative Operations

We don't want to purchase the item if the stock is not available

The basket asks the warehouse if there is stock, and if so reserves it. Fault could replace message here, if the item is out-of-stock



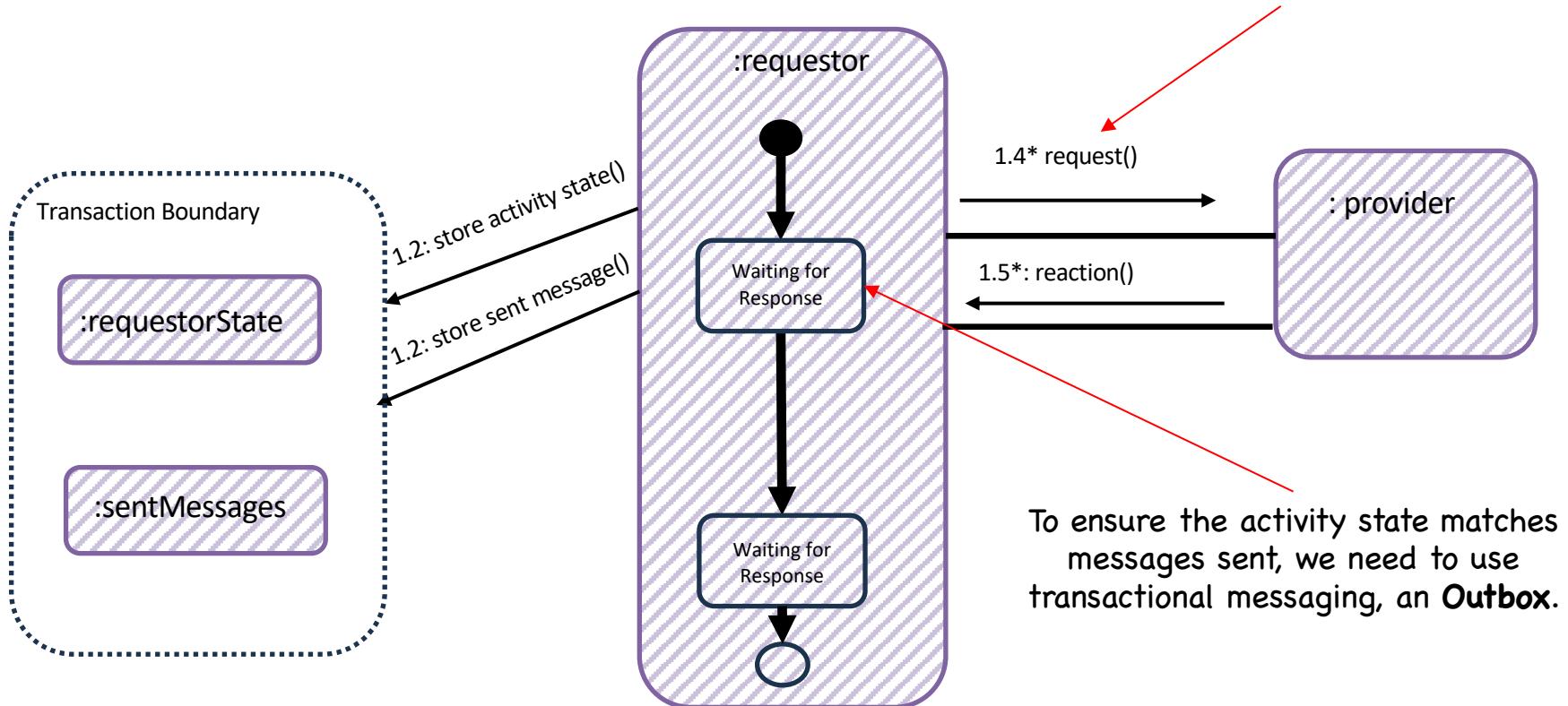
If the customer finishes shopping and pays for the basket before the time limit, then it commits to ask the warehouse to allocate the stock.

If the basket does not complete, then it rolls back to ask the warehouse to free the stock.

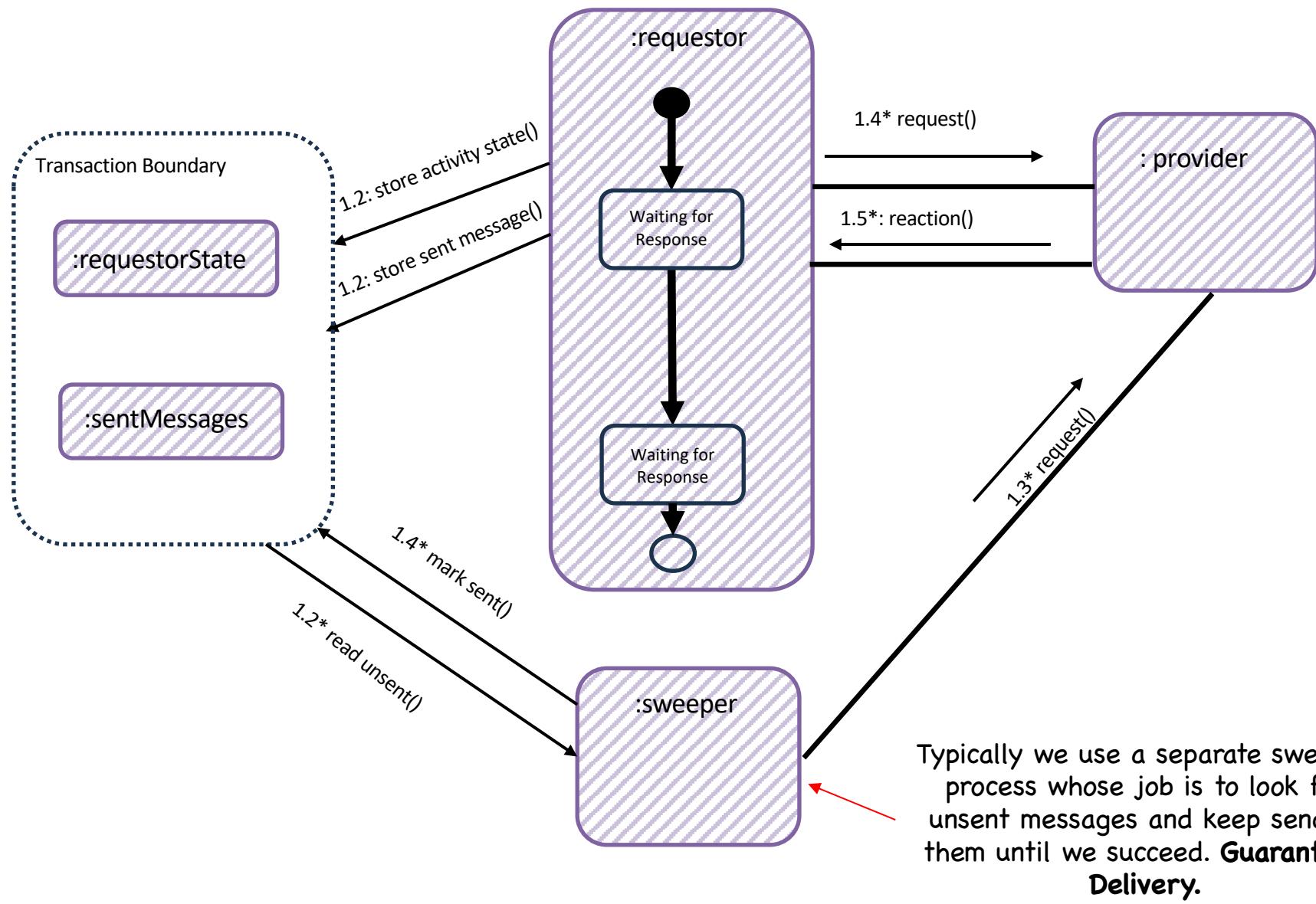
How do we ensure that our messaging is reliable?

RELIABLE MESSAGING

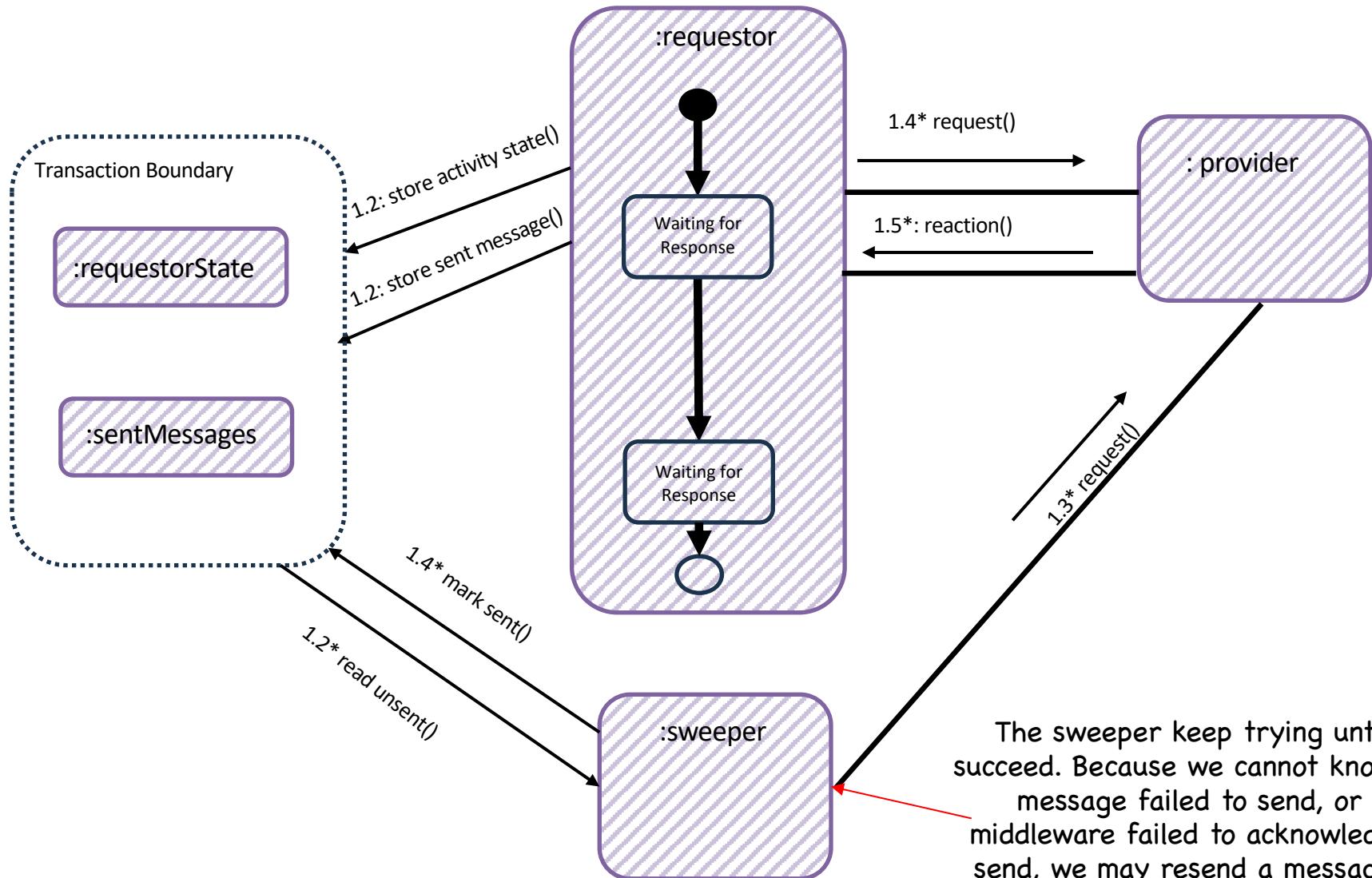
Reliable Messaging – Outbox



Reliable Messaging – Guaranteed Delivery

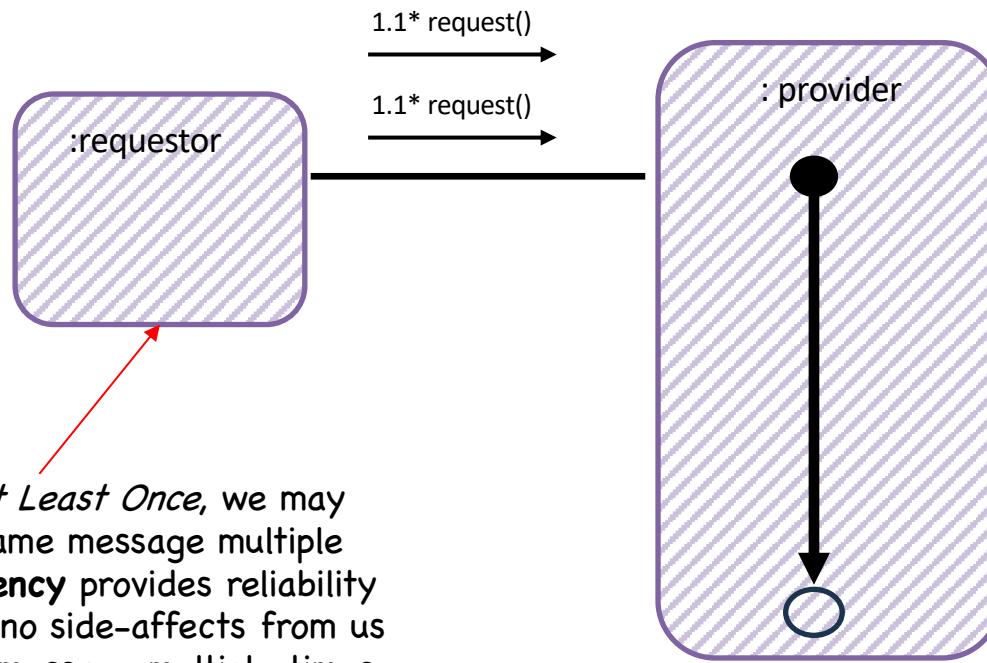


Reliable Messaging – At Least Once

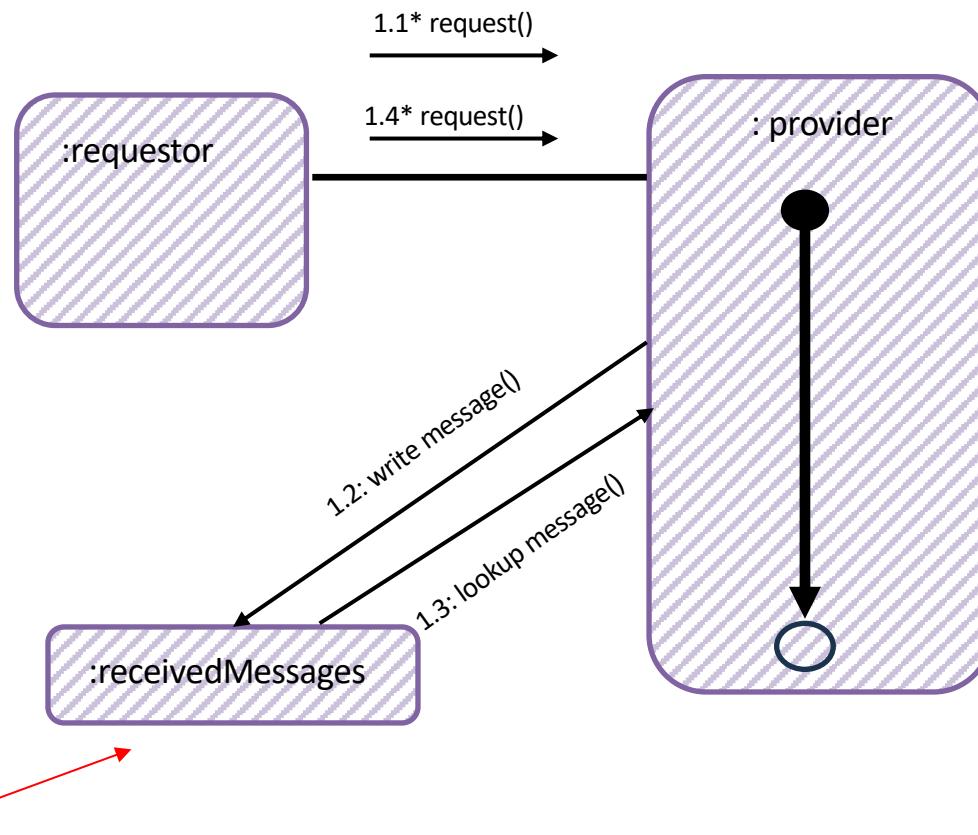


The sweeper keep trying until we succeed. Because we cannot know if the message failed to send, or the middleware failed to acknowledge our send, we may resend a message that has been sent. **At Least Once**.

Reliable Messaging - Idempotency

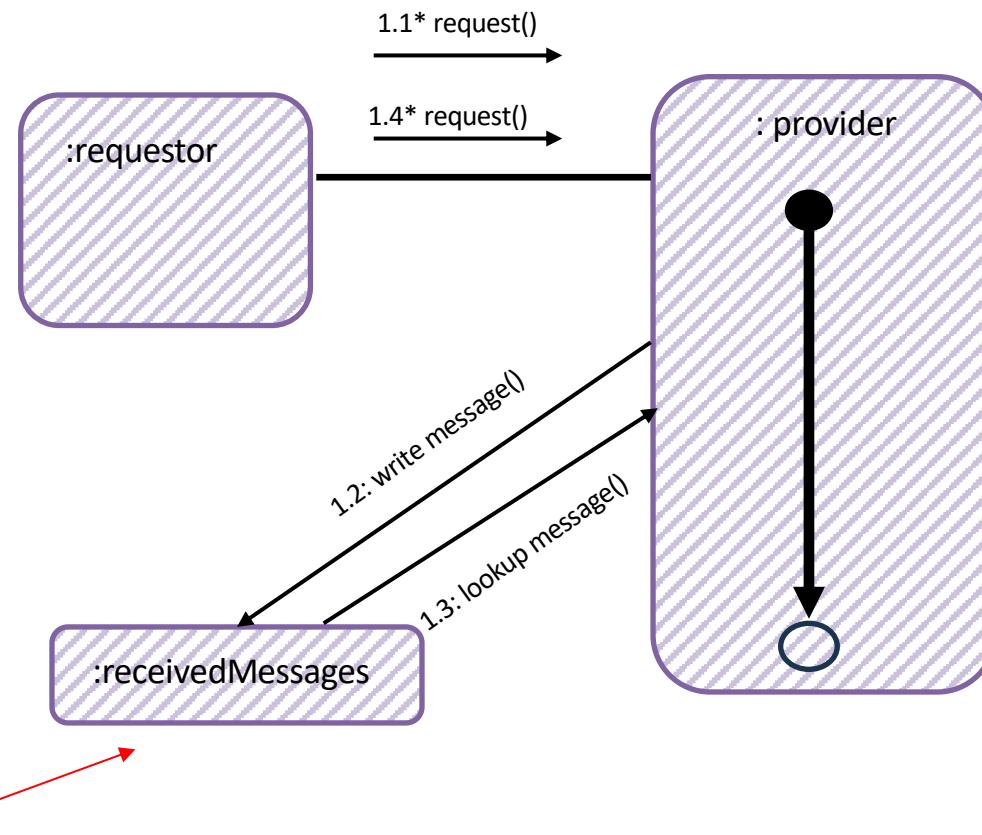


Reliable Messaging - Inbox



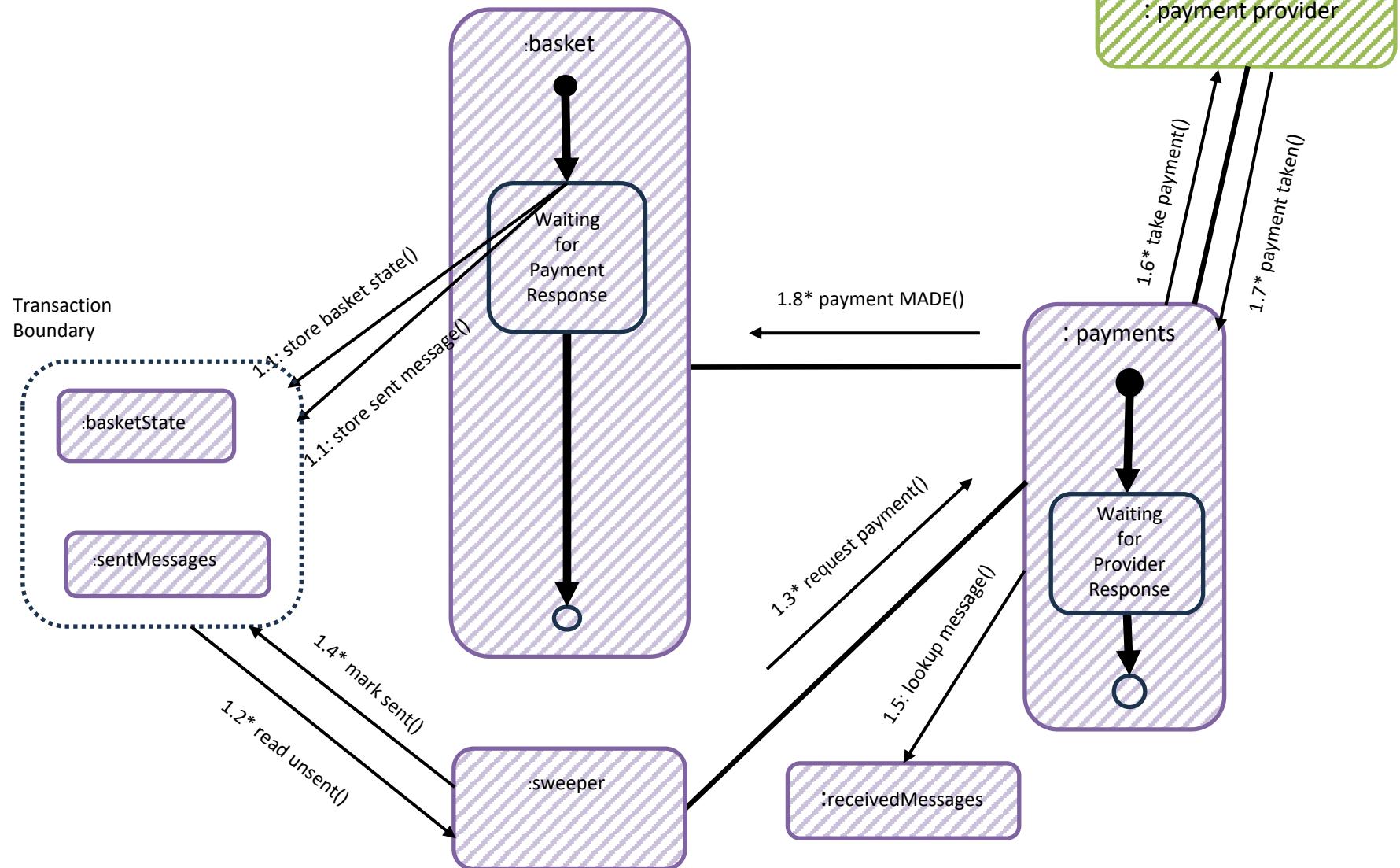
Because of *At Least Once*, to ensure that we have not already processed a message, we use a store seen messages, an **Inbox**.

Reliable Messaging – Once Only



Because an Inbox filters out duplicates,
we can turn *At Least Once* into **Once
Only**.

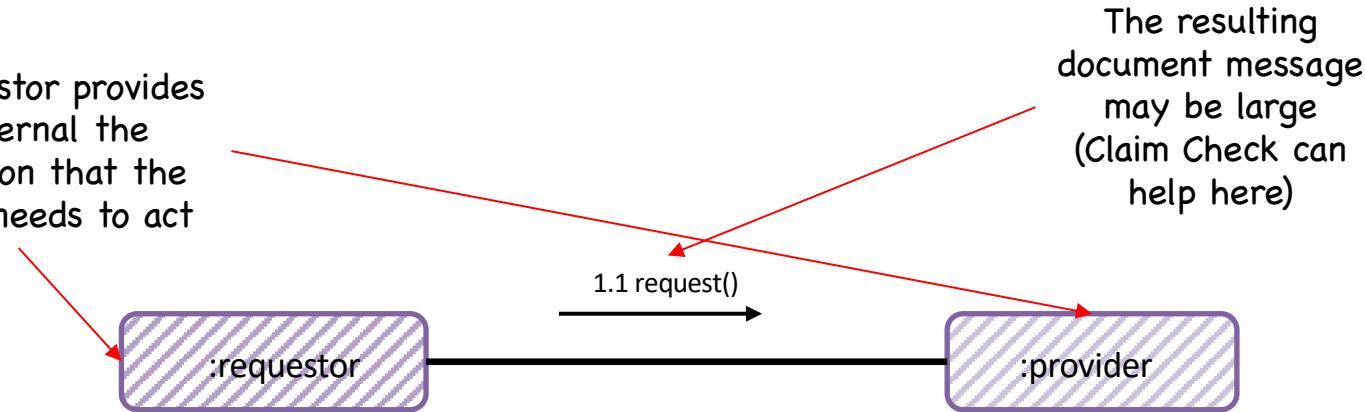
Reliable Messaging



FAT AND SKINNY

Fat Message

The requestor provides all external the information that the provider needs to act

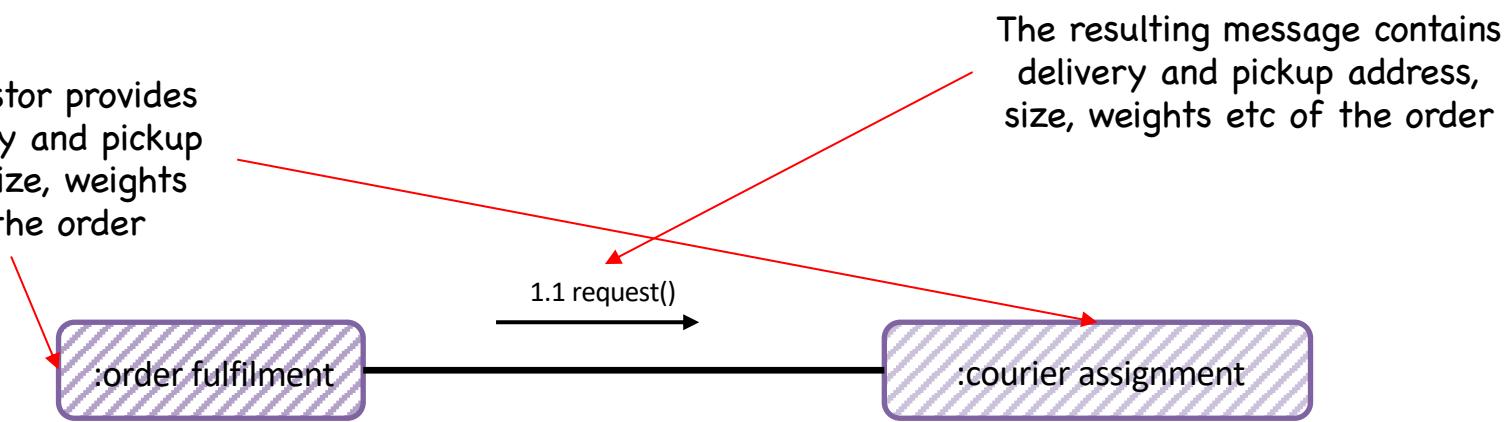


The resulting document message may be large
(Claim Check can help here)

With a **Fat Message** the requestor sends across all the external information a provider may need to perform the operation.

Fat Message Example

The requestor provides the delivery and pickup address, size, weights etc of the order



Fat Message, Transitive Dependencies

Purchase Order Message

OrderId	Customer First Name	Customer Last Name	Customer Post Code	Restaurant Name	Restaurant Post Code	Order Amount	Order items
12345	Jo	Doe	SW17 3NJ	Pizza 'R Us	SW17 5HK	1038p	...

This data has a lifetime of the message i.e. the purchase order and their schema changes if the purchase order changes

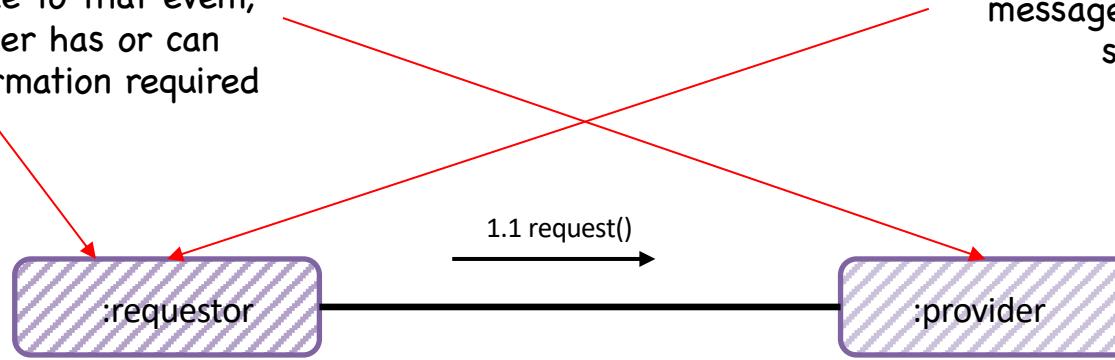
This data is a **transitive dependency**, it has a lifetime of the Customer and their schema changes if the Customer schema changes – which may force us to change the message for Customer changes

This data is a **transitive dependency**, it has a lifetime of the Restaurant and their schema changes if the Restaurant schema changes – which may force us to change the message for Restaurant changes

Skinny Request

The requestor provides only information unique to that event, assumes provider has or can obtain other information required

The resulting notification message is normally skinny

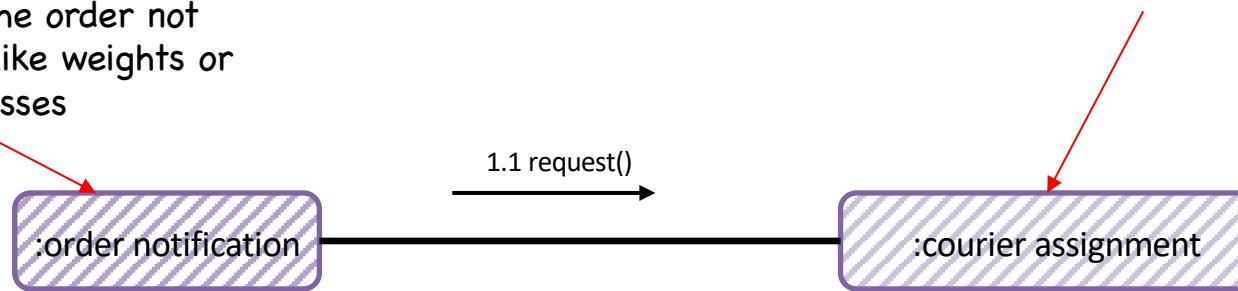


With a **Skinny Request** the requestor assumes the provider has necessary external information to perform the operation.

Skinny Request, Example

The requestor provides a notification that there is an order, but only inlines data that is unique to the order not reference data like weights or addresses

Courier assignment needs information that is not on the order: weights, addresses. It has to source that from elsewhere



Skinny Message, Normalized

Purchase Order Message

OrderId	Order Amount	Order items	CustomerId	RestaurantId
12345	1038p	...		

Customer

Customer First Name	Customer Last Name	Customer Post Code
Jo	Doe	SW17 3NJ

We use an Id where the data does not share the lifetime of the message; we assume the requestor obtains the data out-of-band

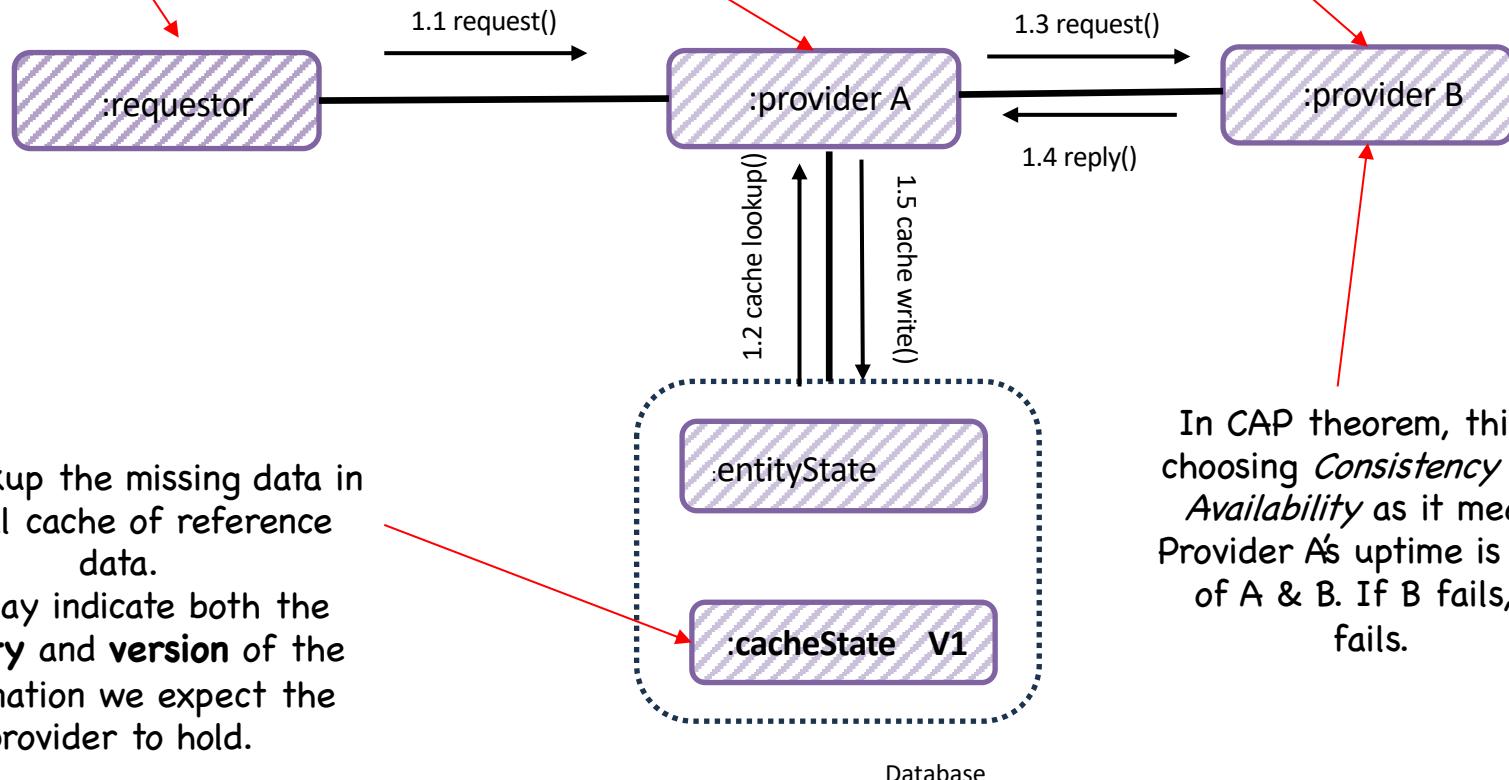
We need to lookup this id with other providers

Restaurant

Restaurant Name	Restaurant Post Code
Pizza 'R Us	SW17 5HK

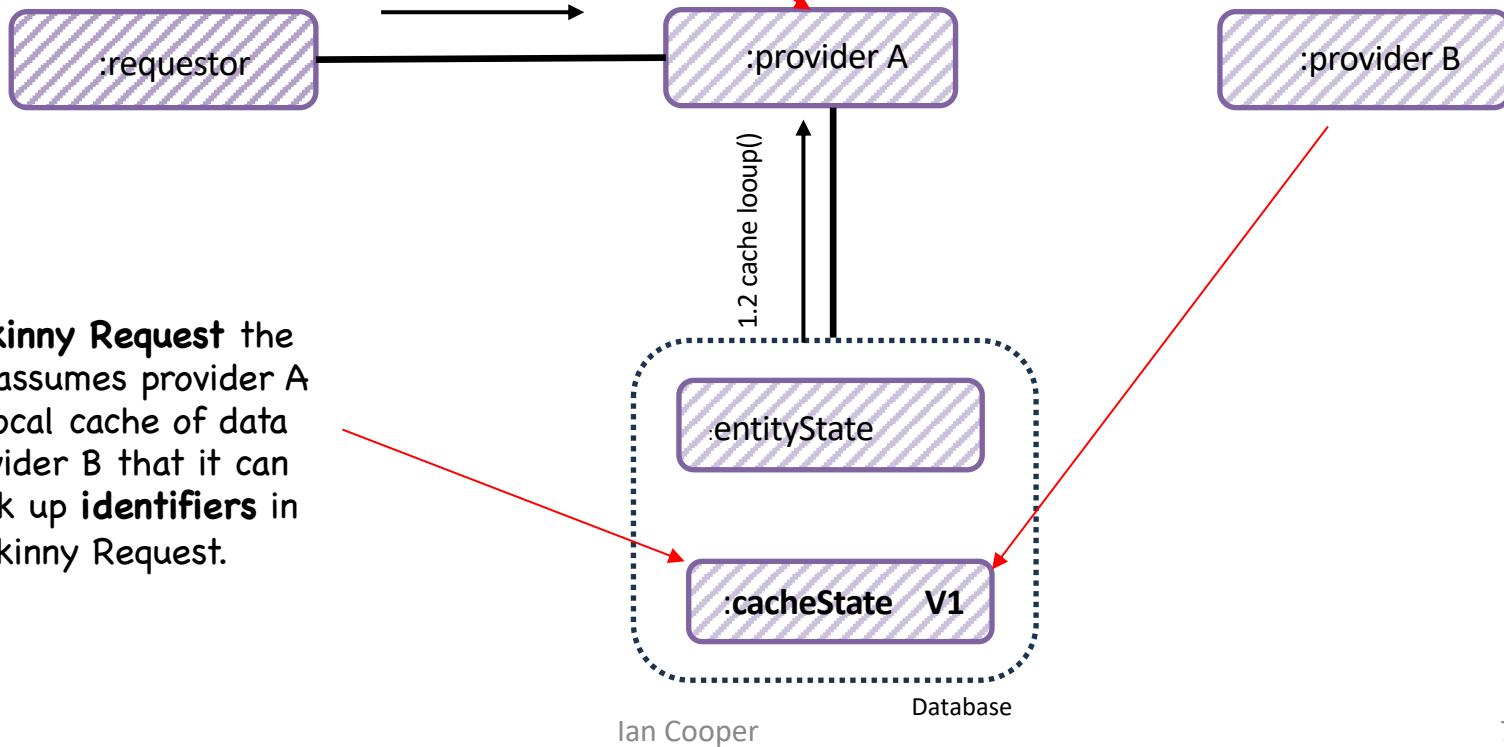
Skinny Request via REST/RPC

The requestor provides only information unique to that event, assumes provider A has or can obtain information from provider B



Skinny Request using Reference Data

The requestor provides only information unique to that event, assumes *provider A* has or can obtain information from *provider B*



With a **Skinny Request** the requestor assumes provider A B has a local cache of data from provider B that it can use to look up **identifiers** in the Skinny Request.

Data that leaves Provider B via an API is **Reference Data**.

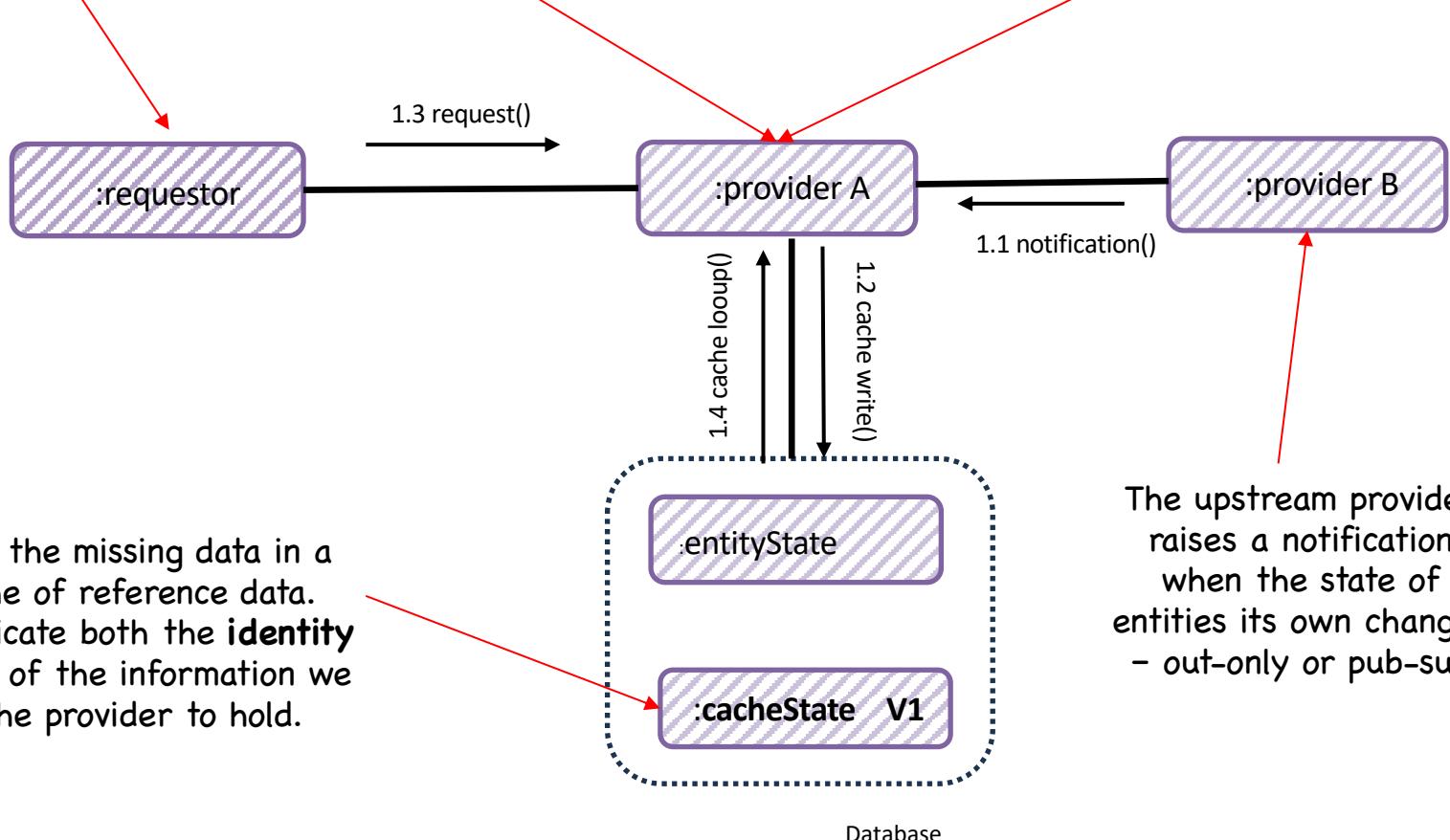
It is:

- *Immutable*
- *Versioned*
- *Stale*

It can be cached locally to provider A to prevent the need for frequent lookups

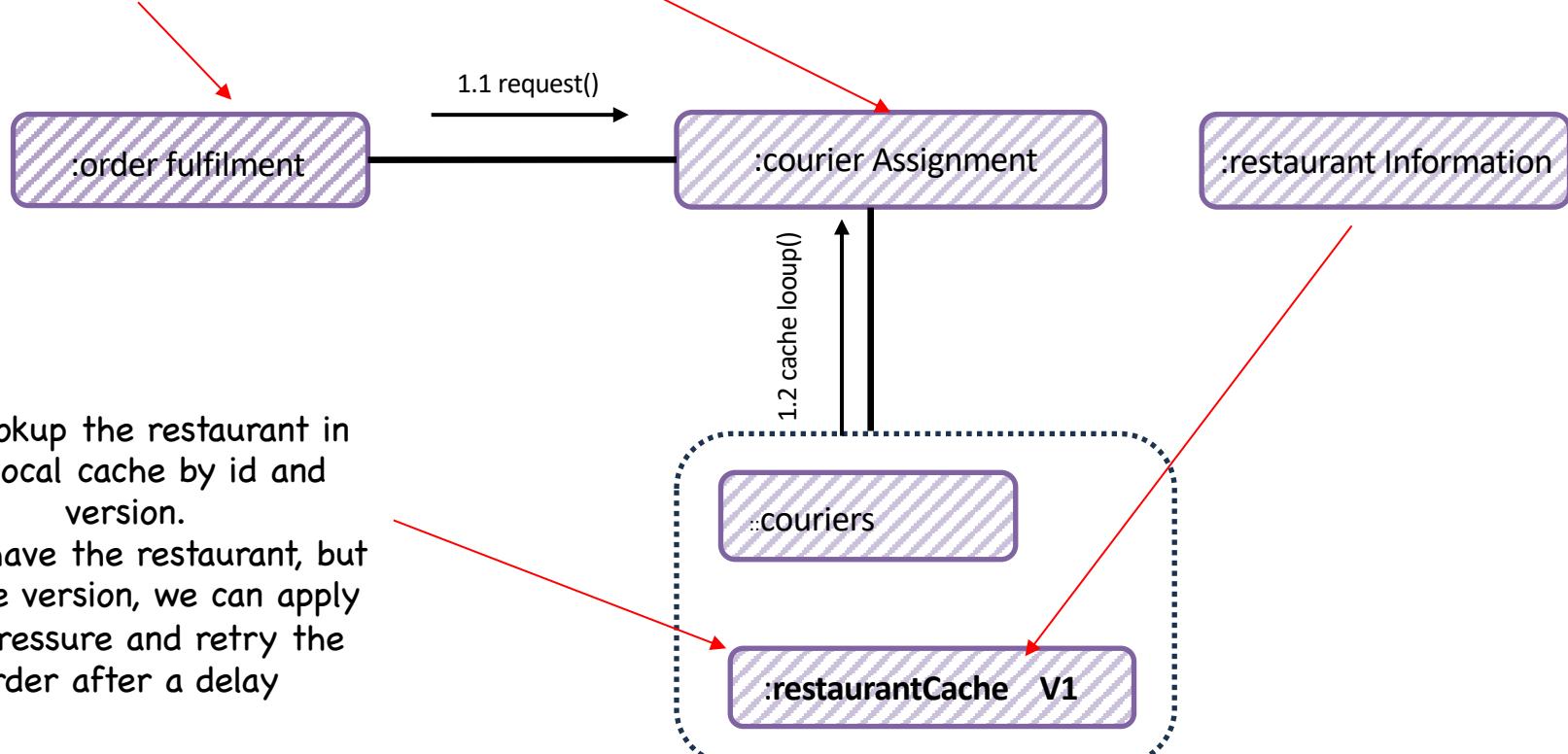
Skinny Request, Reference Data via ECST (Event Carried State Transfer)

The requestor provides only information unique to that event, assumes provider A has or can obtain information from provider B



Skinny Request using Reference Data

Order Fulfilment does not include address details for the restaurant for pickup, instead we assume that courier assignment has obtained it from restaurant information



We lookup the restaurant in the local cache by id and version.

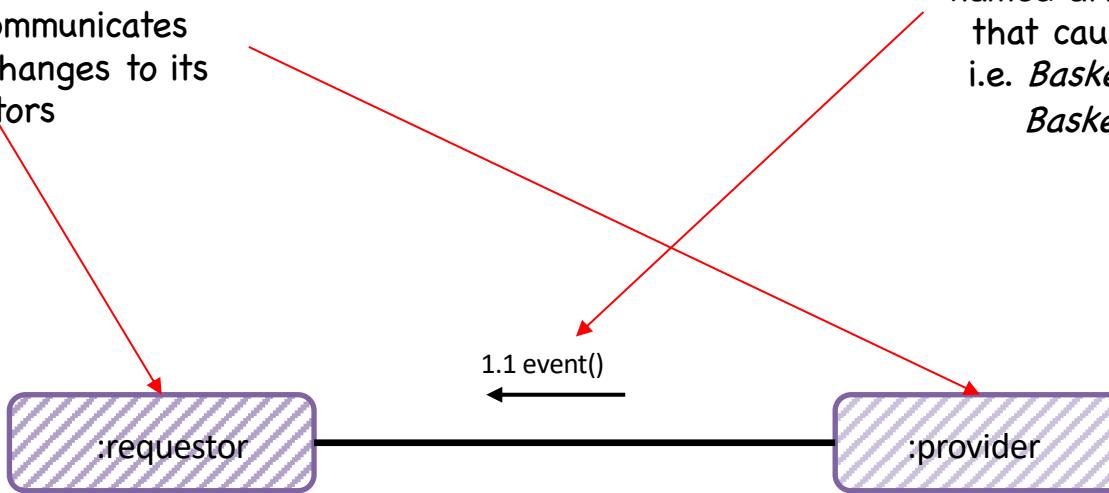
If we have the restaurant, but not the version, we can apply backpressure and retry the order after a delay

DOMAIN, SUMMARY, ORDERING

Domain Event

An approach to Pub-Sub where the provider communicates **granular** state changes to its requestors

The resulting message is usually a past participle named after the command that caused the change i.e. *BasketItemRemoved*, *BasketItemAdded*



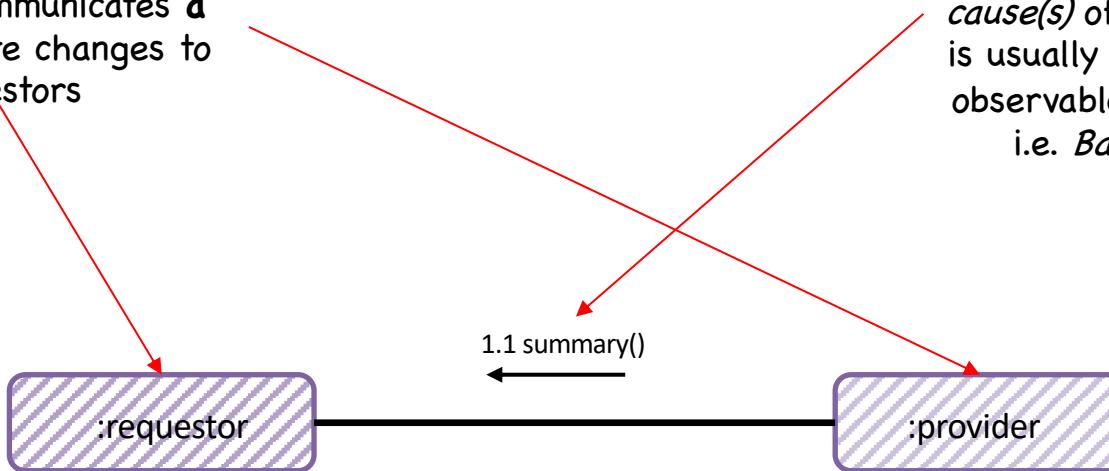
With a **Domain Event** the provider cannot use a *DataType Channel* as the domain events for the observable must be ordered on the same channel, and have a different schema.

This puts an additional burden on the requestor to multiplex handling the event

Summary Event

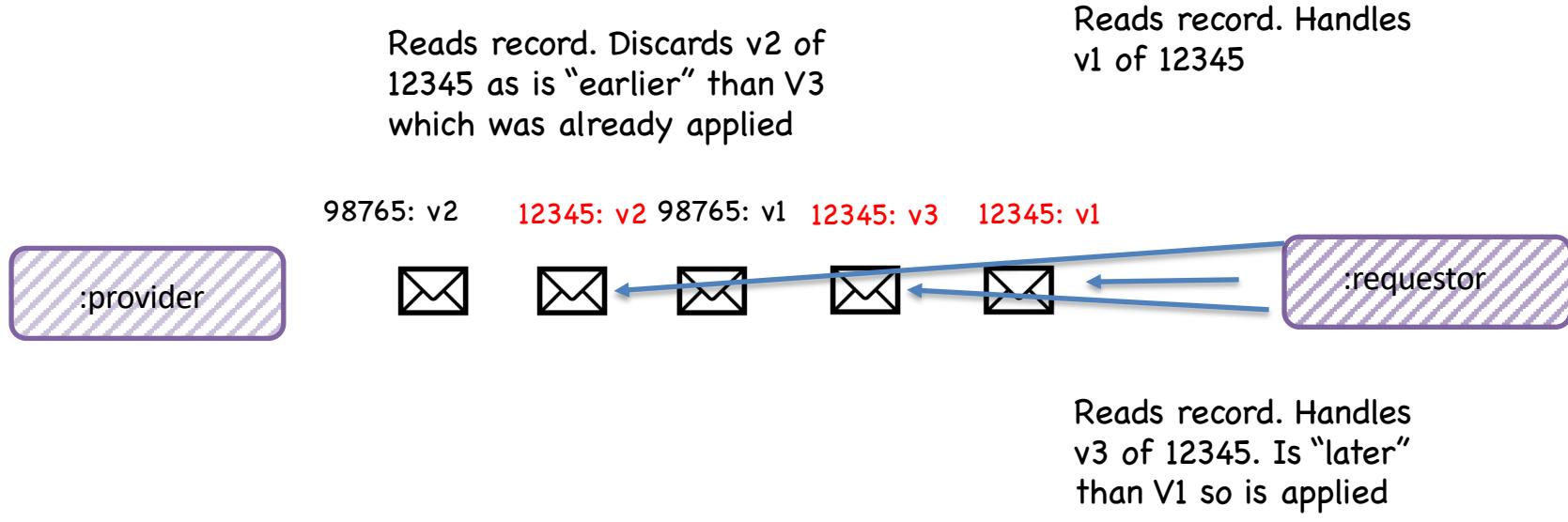
An approach to Pub-Sub where the provider communicates a **summary** of state changes to its requestors

The resulting message is **versioned** and contains *metadata describing the cause(s) of any changes*. It is usually named after the observable in the pub-sub i.e. *BasketChanged*



With a **Summary Event** the provider can use a *DataType Channel* as there is just one schema used to communicate a snapshot of the observable.

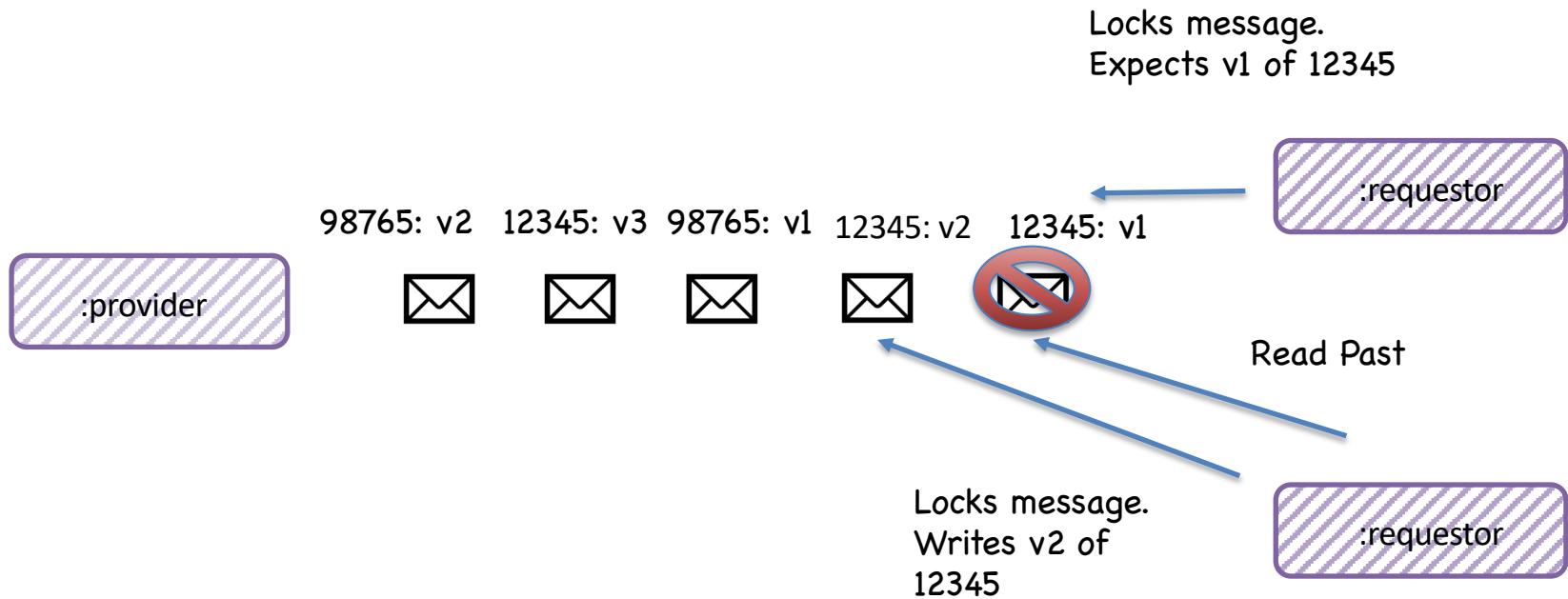
If Later, Stream



If Later allows us to use a versioned **Summary Event** to ignore ordering errors. Even on a stream, a non-blocking retry or guaranteed delivery via an outbox may result in out-of-order messages. We can also shed load

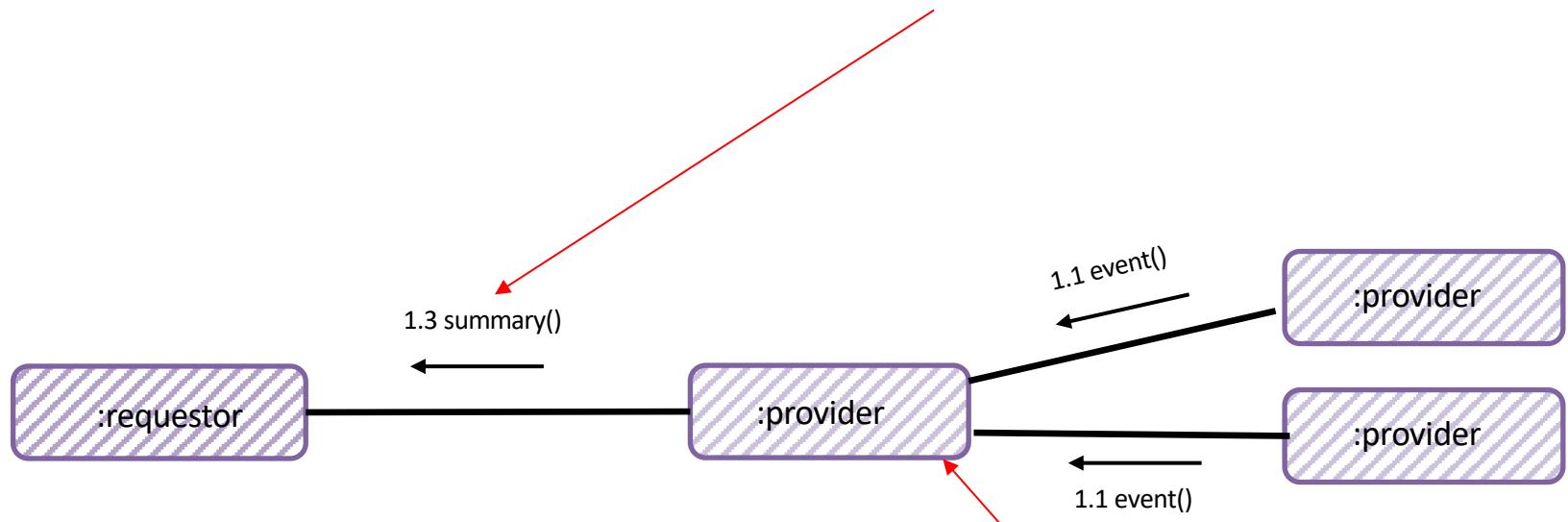
We cannot use If Later with a **Domain Event**, as they all must be applied. We can only use a blocking retry with a Domain Event and cannot shed load.

If Later, Queue



If later allows message to be processed out-of-order with a queue and competing consumers. A queue normally processes messages, and not events, so this only applies where we are using a queue.

If a message must be ordered, such as a series of commands, we must use requeue with delay, or a sequencer to re-order.



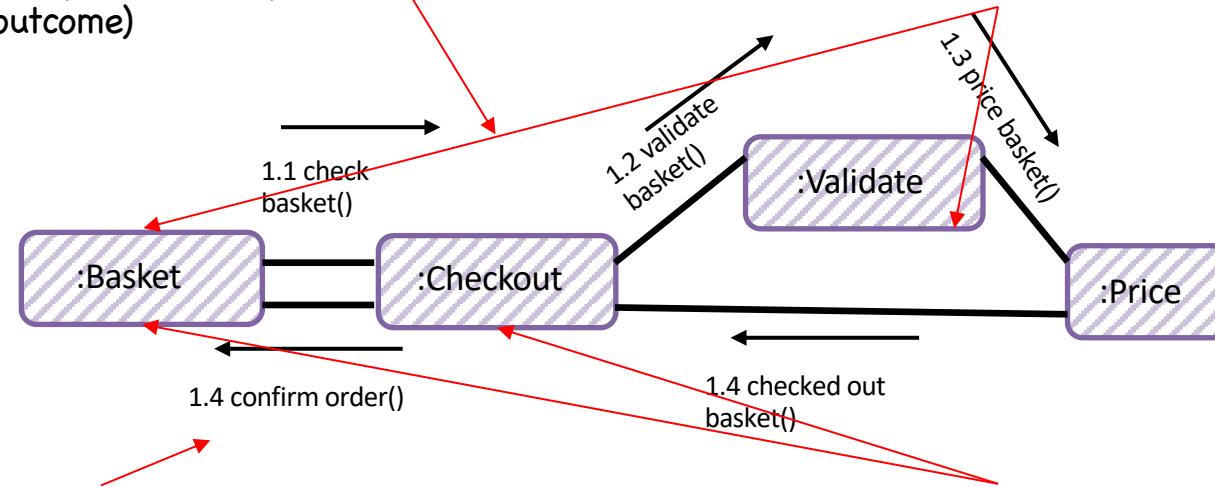
The resulting message is **versioned** and contains *metadata describing the cause(s) of any changes.*

An approach to Pub-Sub where a **public provider** communicates with collaborators in other domains via a *summary event of domain events* from **private providers**

CONVERSATIONS

Services are Processes

An ordering of activities with a beginning and end: it has inputs and outputs (outcome)

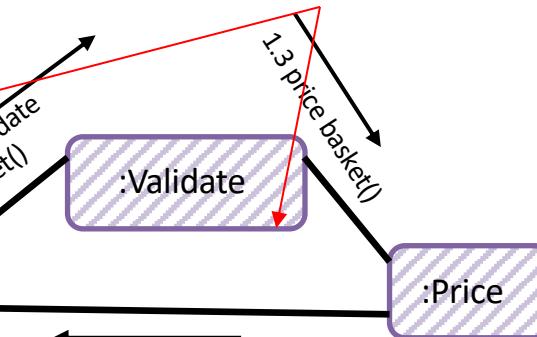


Command and Control

Who owns the process? Who informs us about its state? Who knows what step is next?

Lack of Distributed Transactions

Whilst 2PC is possible it is rarely used, how do we create an atomic workflow?

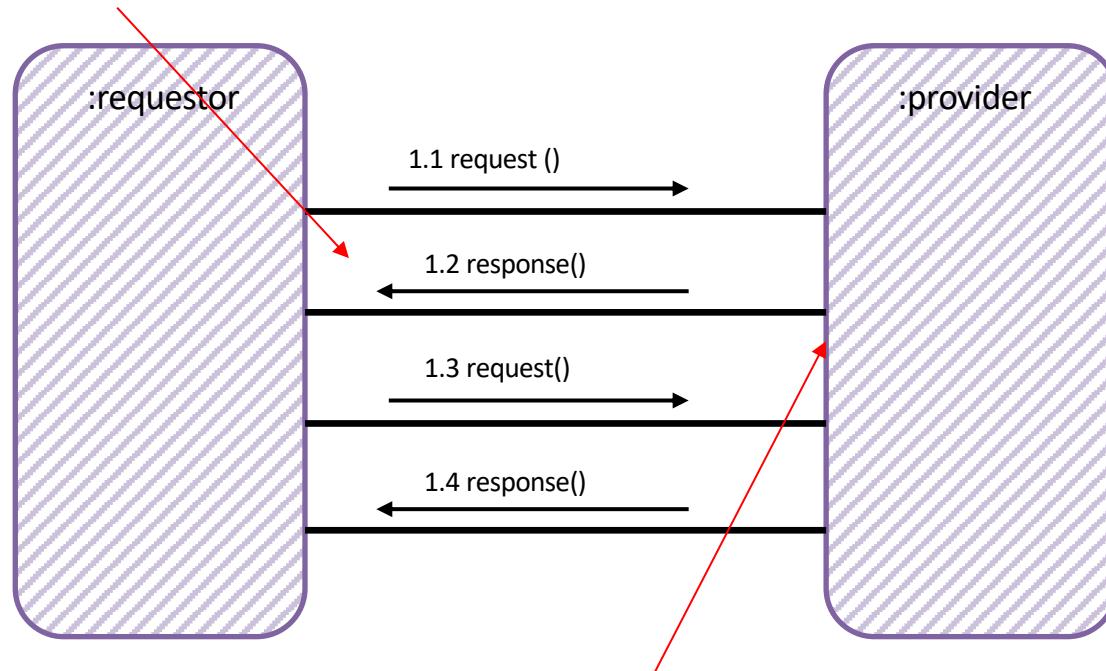


Coupling

Are we just coupled to data? Or are we coupled to behaviour?

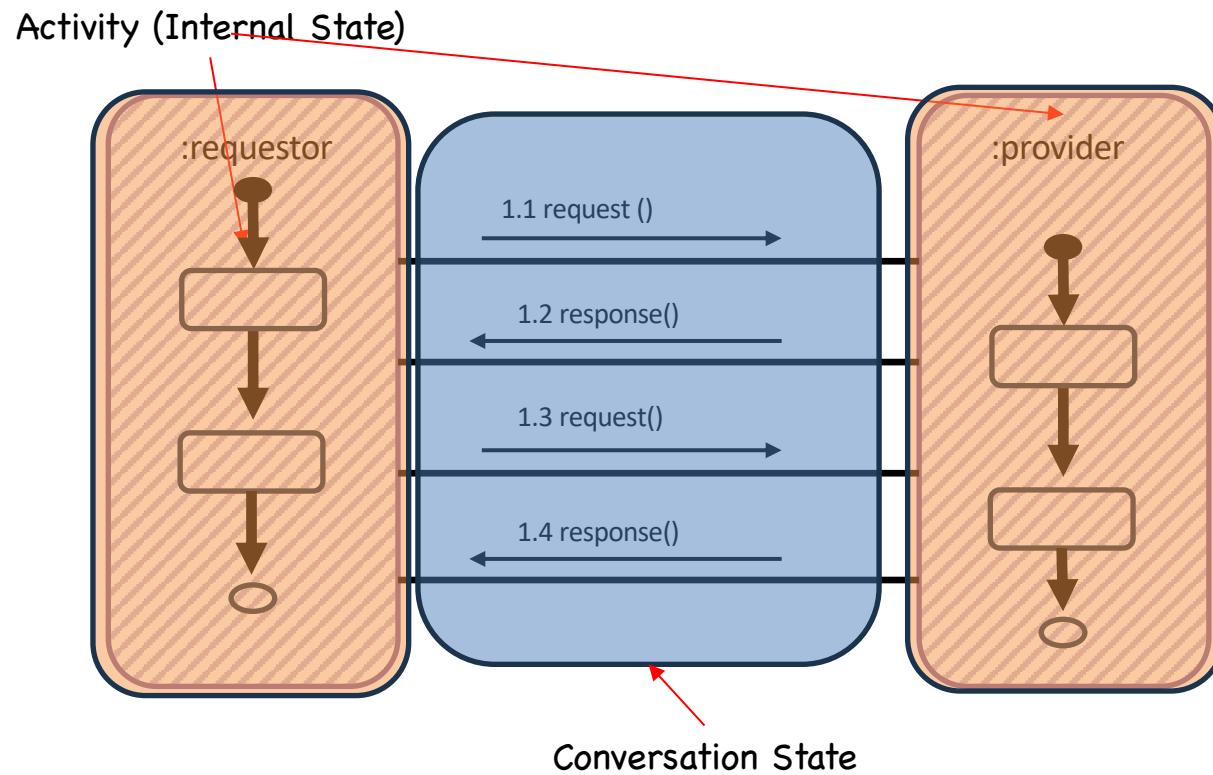
Turn Taking

Turn-taking consists of a set of in-out pairs



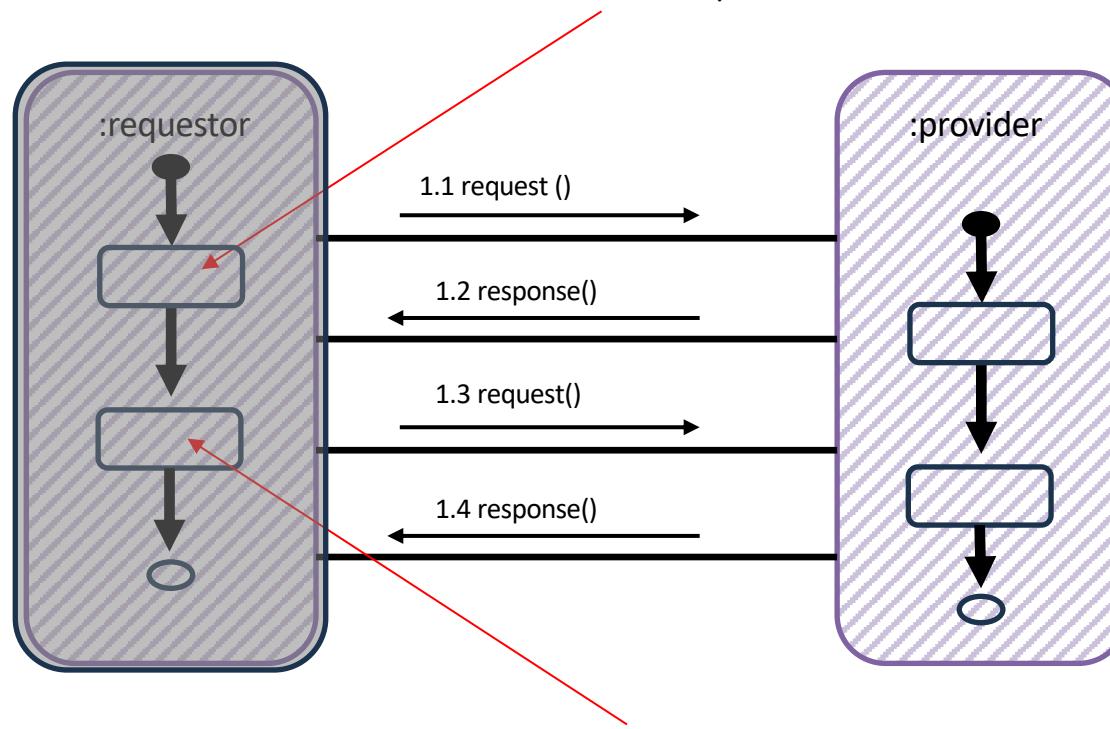
As each participant takes its turn, it takes control of the flow

Turn Taking



Turn Taking

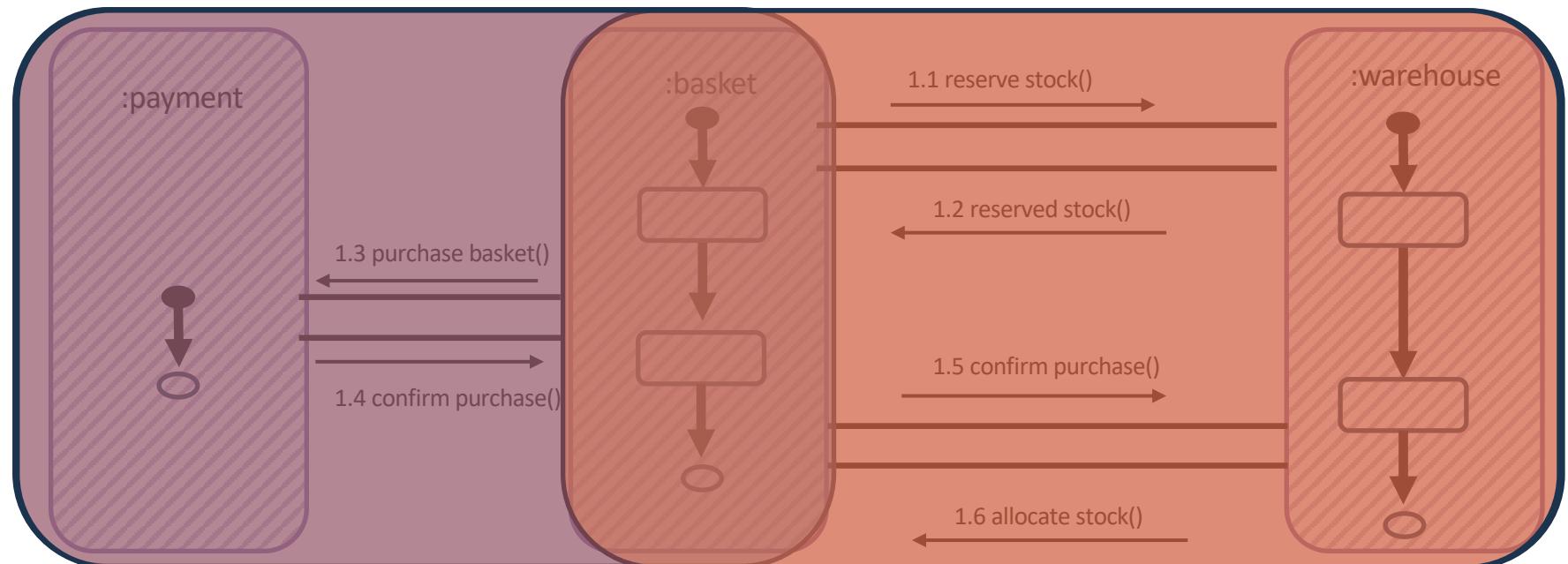
Orchestration: turn taking implies causality, and by implication the requestor's activity orchestrates the conversation with the provider.



Smart Endpoints, Dumb Pipes: we don't need a process manager/saga, as we can use turn-taking instead

Turn Taking

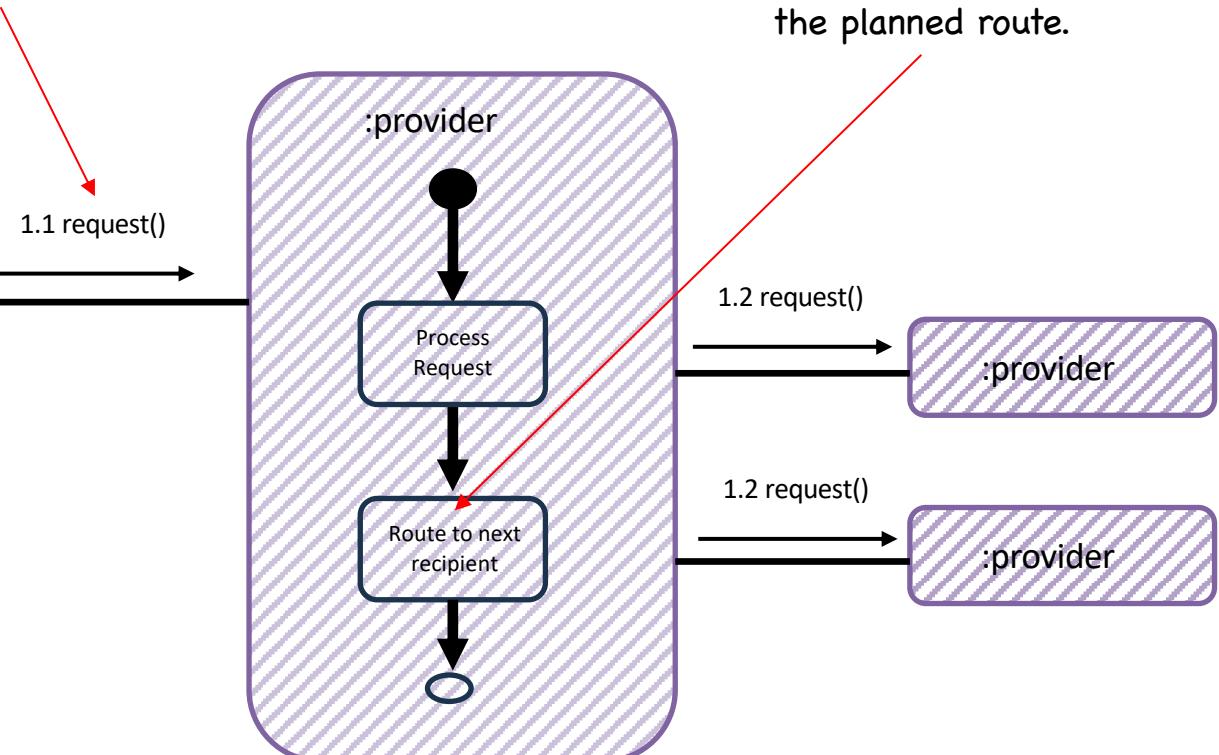
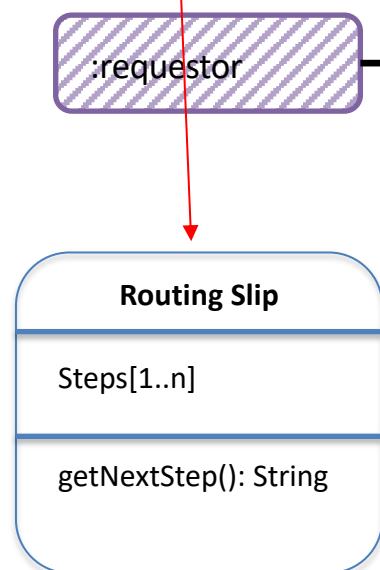
Choreography



Orchestration

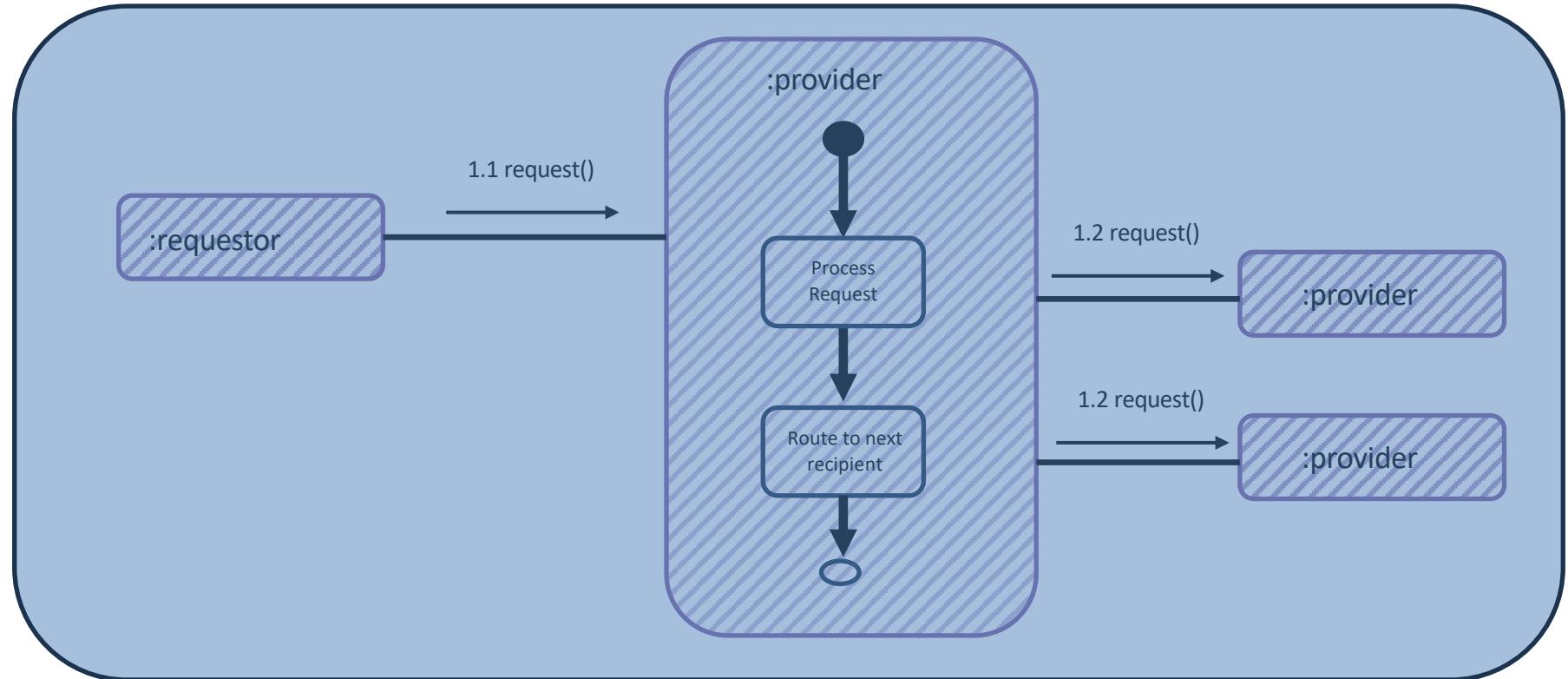
Routing Slip

The request is a **Routing Slip**, which contains the steps of the workflow. Normally these are the (topics) that the message should be forwarded to and an indicator a step is complete



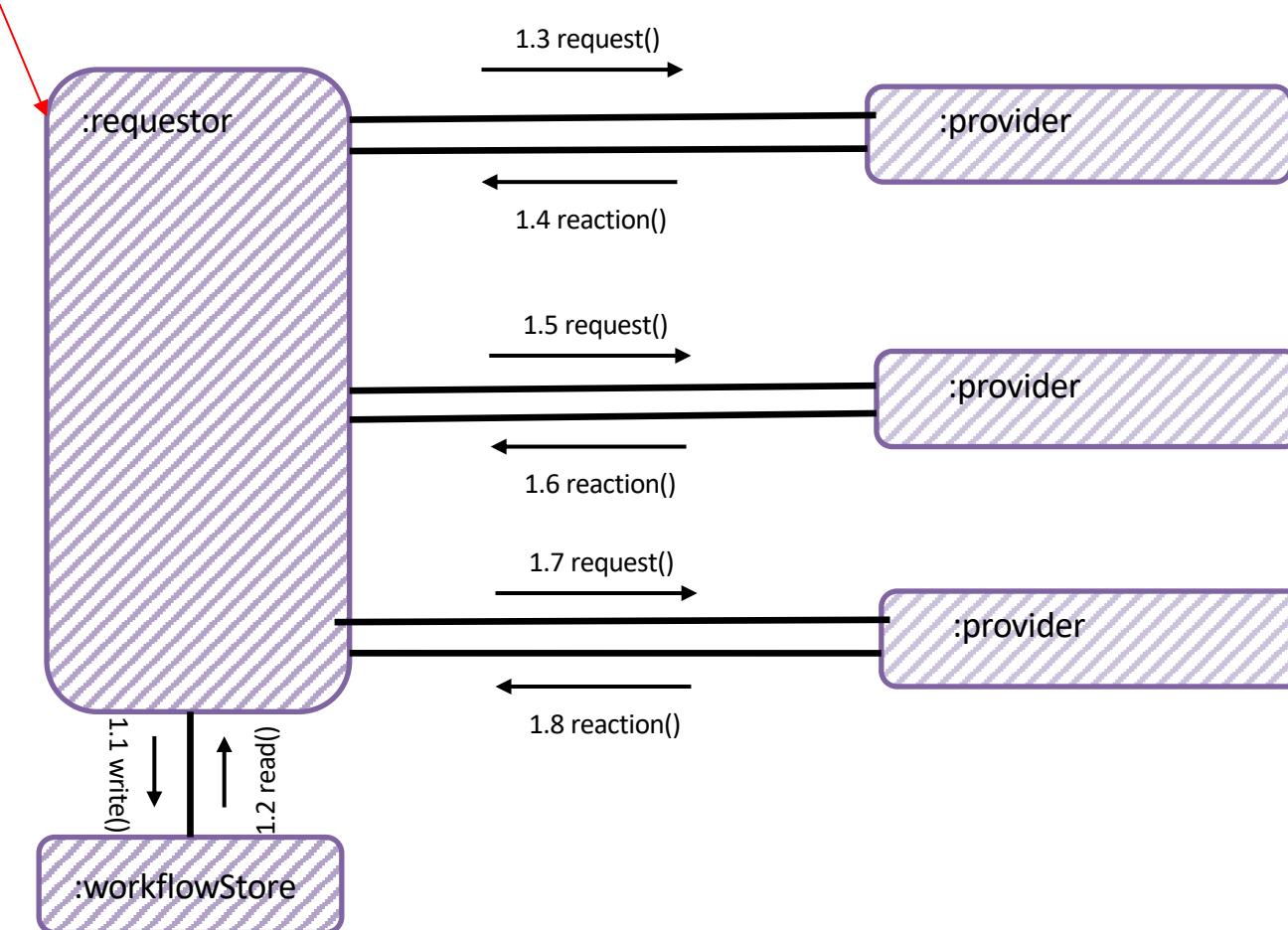
The requestor can choose the steps in the **Routing Slip**, and thus the flow. The point of the Routing Slip though is that the steps do not then make routing choices, but stick to the planned route.

Orchestration



Process Manager (Saga)

The requestor provides the delivery and pickup address, size, weights etc of the order



Turn Taking	Routing Slip	Process Manager (Saga)
Complex Flow	Simple Flow	Complex Flow
Rigid Flow	Dynamic Flow	Dynamic Flow
No central point of failure	No central point of failure	Central point of failure
Distributed	Distributed	Hub-and-Spoke
No central administration or reporting	Central administration but not reporting	Central Administration and Reporting

EXERCISE MATERIAL

Conversations

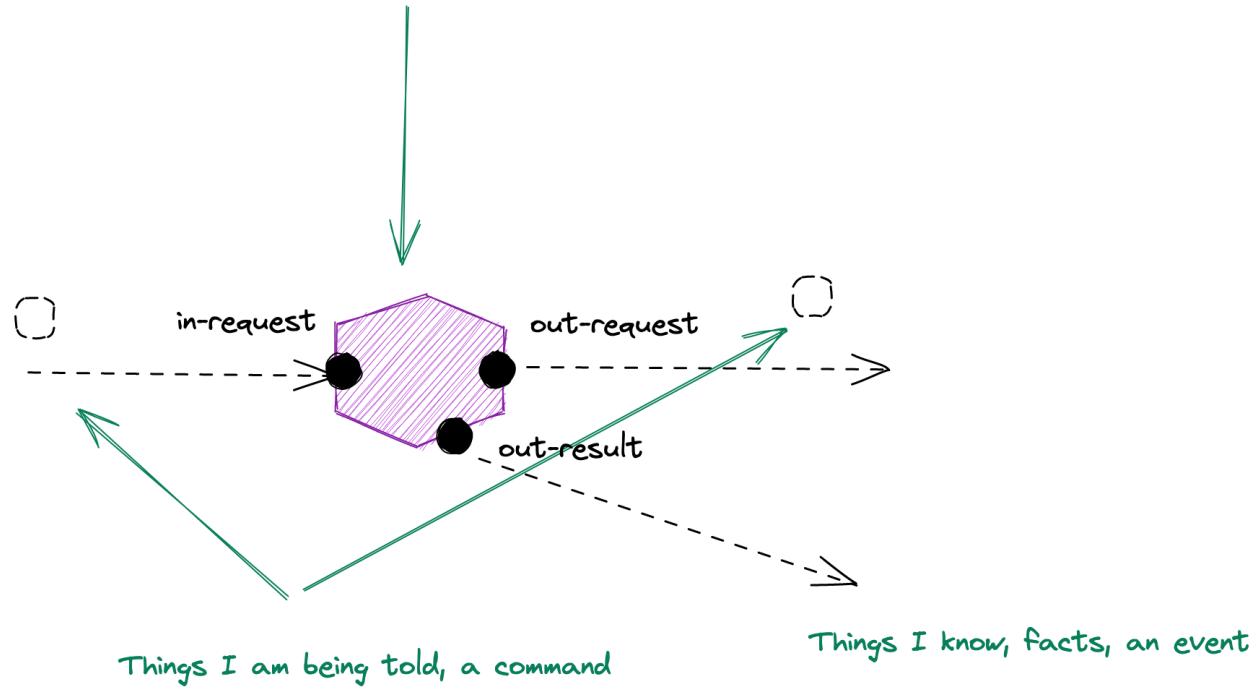
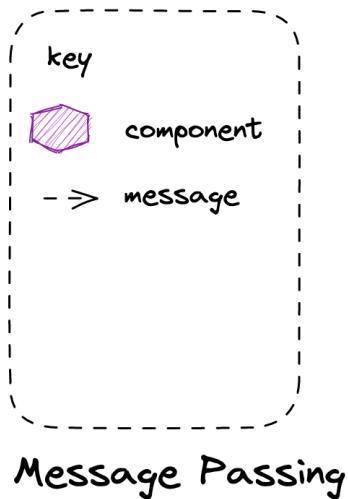
- Readme
- Slides



DON'T PANIC

REACTIVE ARCHITECTURES

Service should focus on a transformation to the data
Usually a verb + noun combination
Place Order or Onboard Restaurant



Paper Workflows

“My life looked good on paper - where, in fact, almost all of it was being lived.” - Martin Amis



Messaging/Discrete Event

Discrete, Immediately Actioned

Skinny



Series Event (Document)

Series, Supports Action

Fat





O E D I DIVISION
O E D I ROUTING SHEET

TITLE: EID-E15 (23 July 84)

REQD legend: I = Information
A = Action
R = Reference

Upon Completion, Return to **ETI Barmag**

Destroy
File

DPS/OE Form No. XXXXXXXXXX



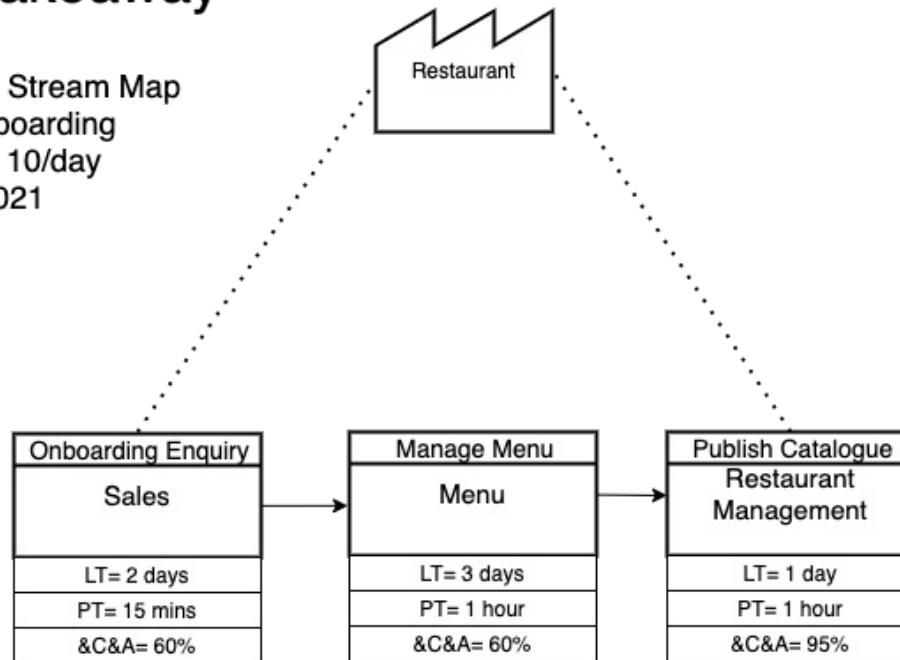
What is a microservice?

SOA is focused on business *processes*. These *processes* are performed in different steps (also called *activities* or *tasks*) on different systems. The primary goal of a **service** is to represent a “natural” step of business functionality. That is, according to the domain for which it’s provided, *a service should represent a self-contained functionality that corresponds to a real-world business activity.*

Josuttis, Nicolai M.. SOA in Practice: The Art of Distributed System Design . O'Reilly Media. Kindle Edition.

Just Paper Takeaway

Current State Value Stream Map
Restaurant Onboarding
Demand Rate 10/day
29 SEP 2021



1

Restaurant
Owner



Restaurant Onboarding



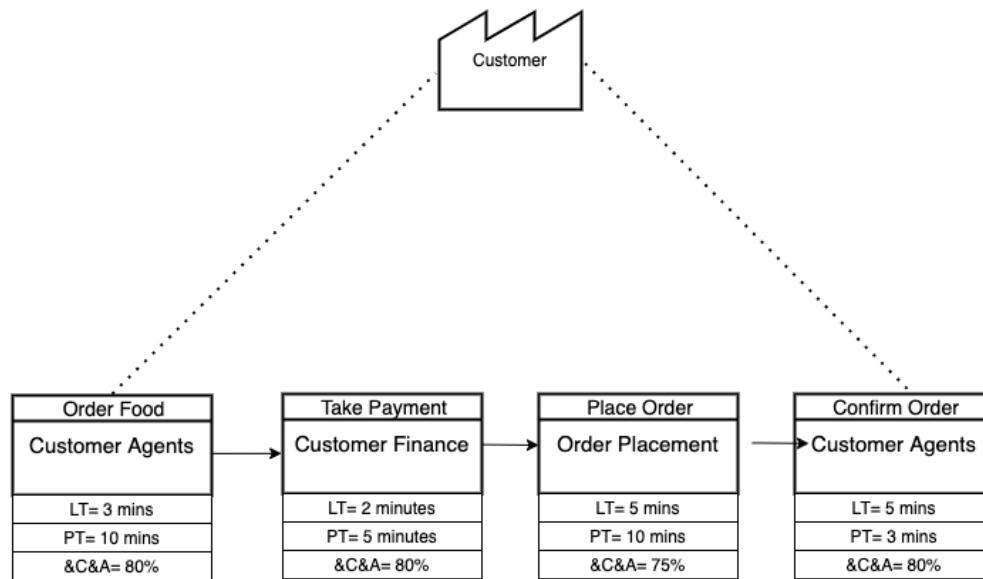
Just Paper Takeaway

Current State Value Stream Map

Order Flow

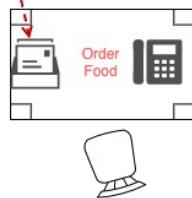
Demand Rate 10/day

29 SEP 2021





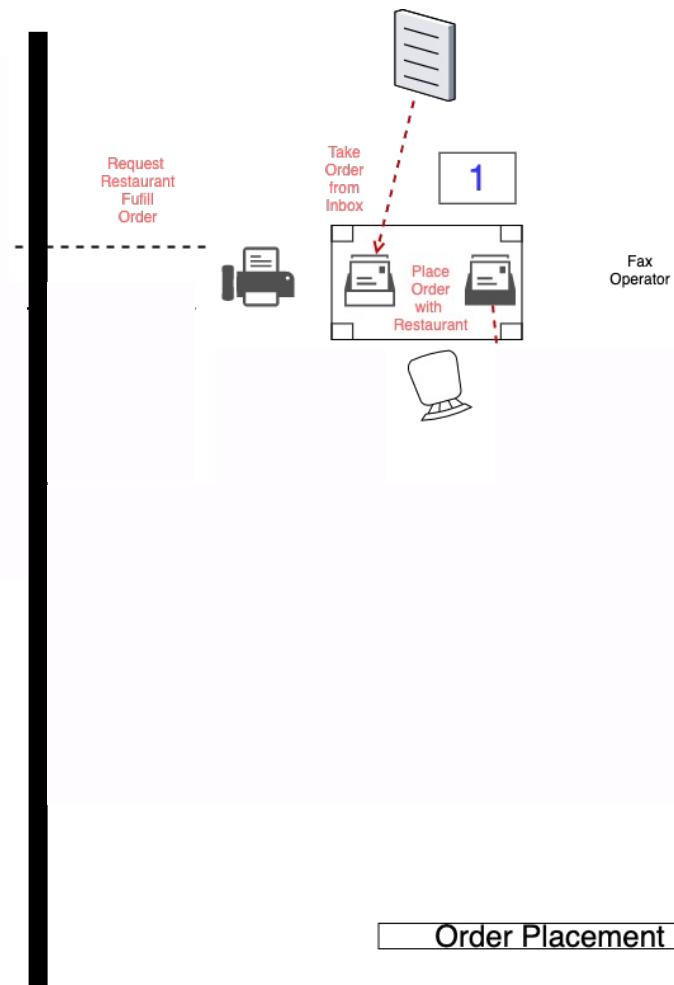
Hungry Customer

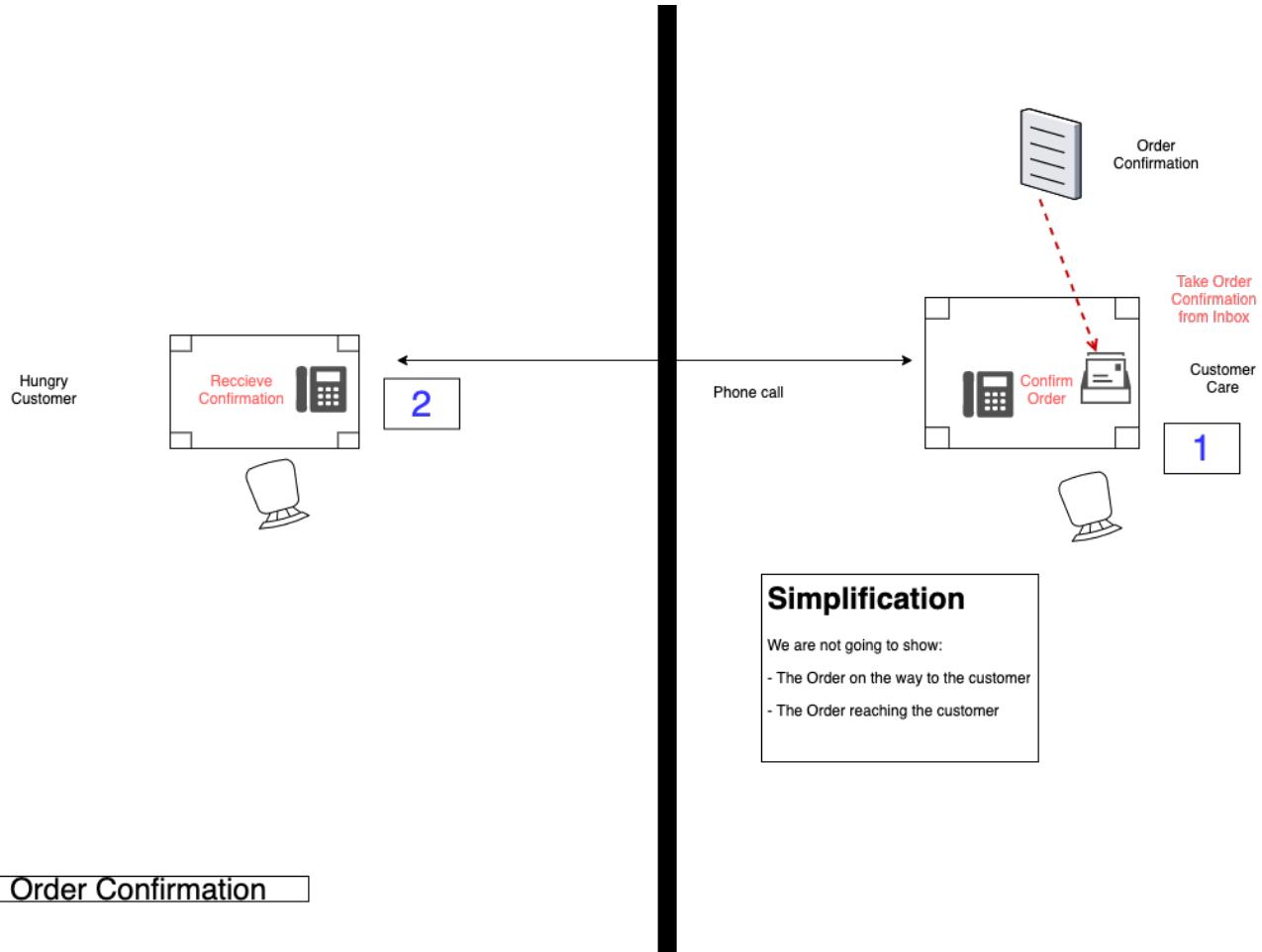


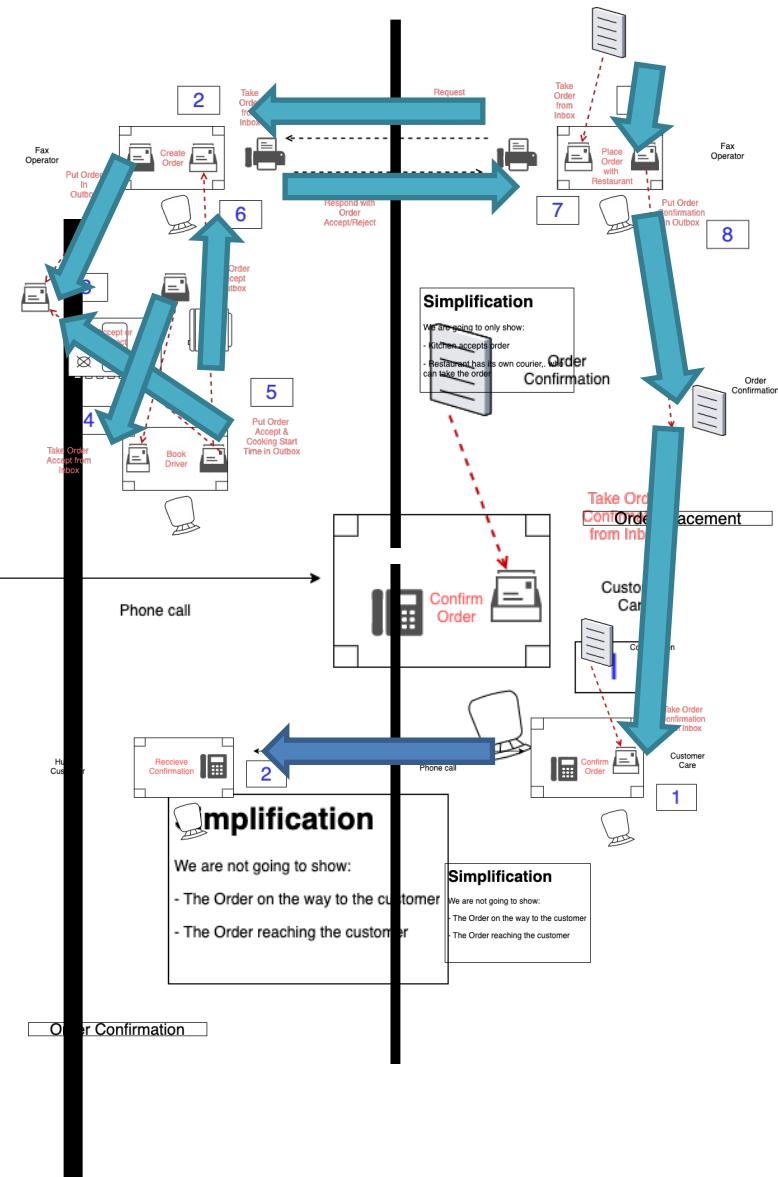
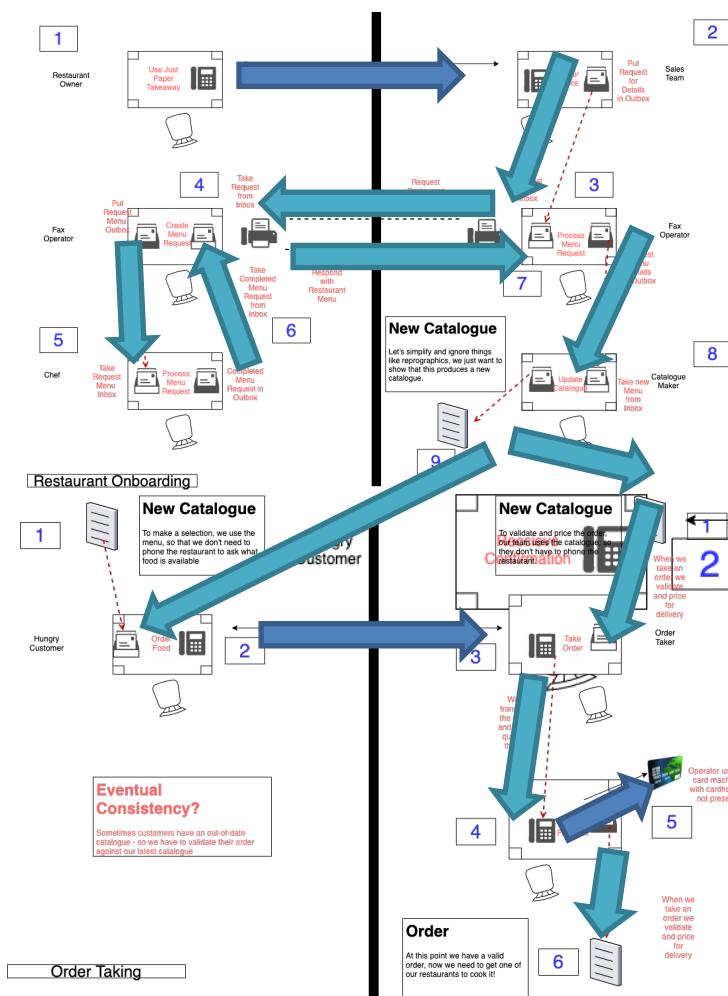
Order Taking

Order Wheel

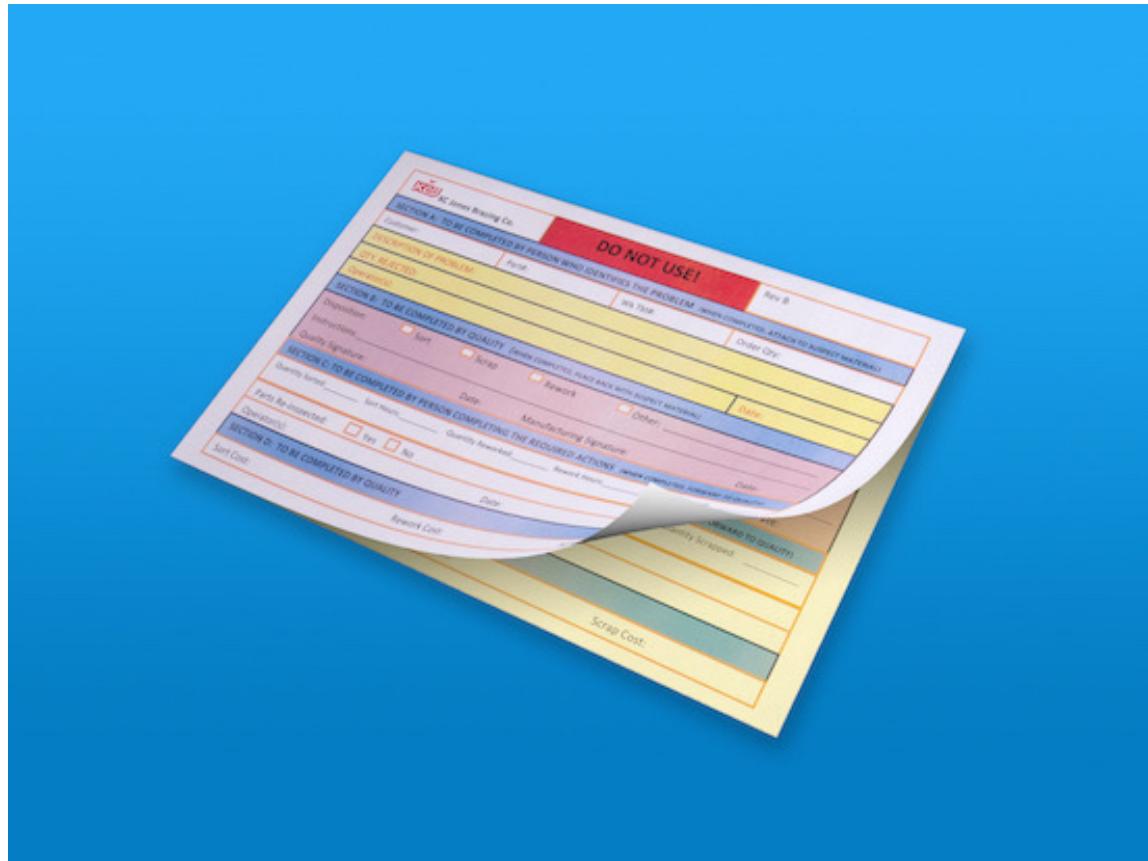


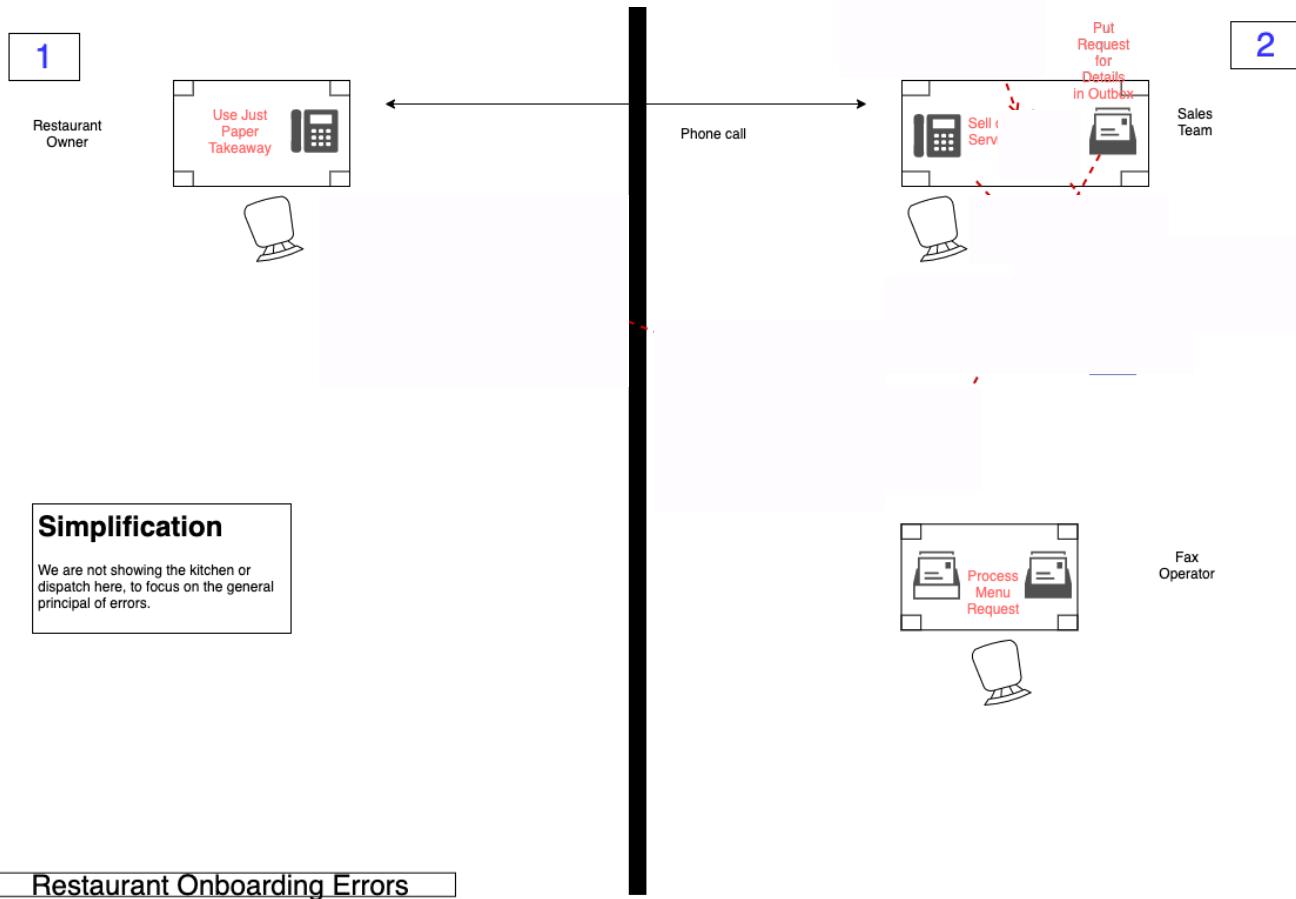






Compensation

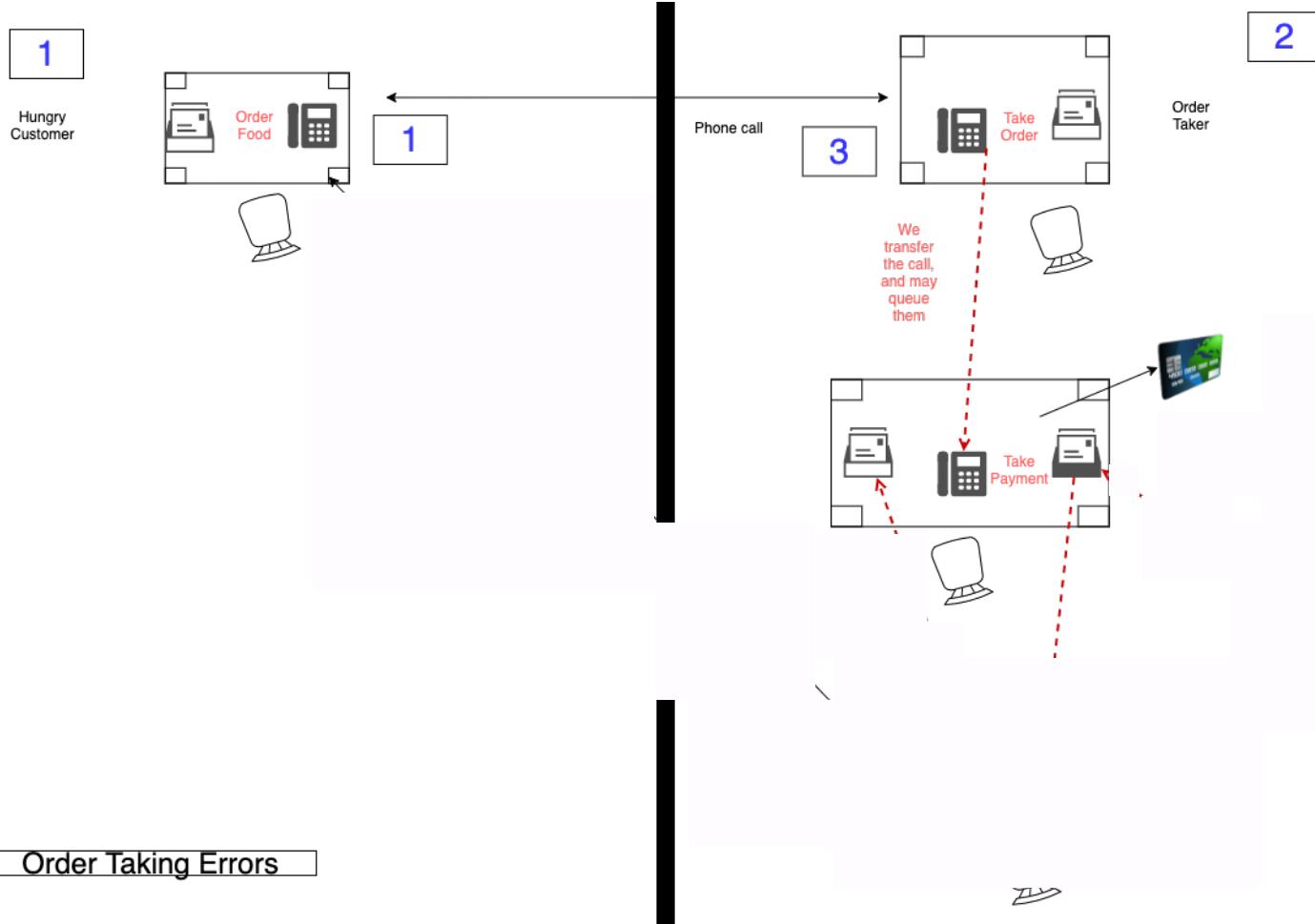


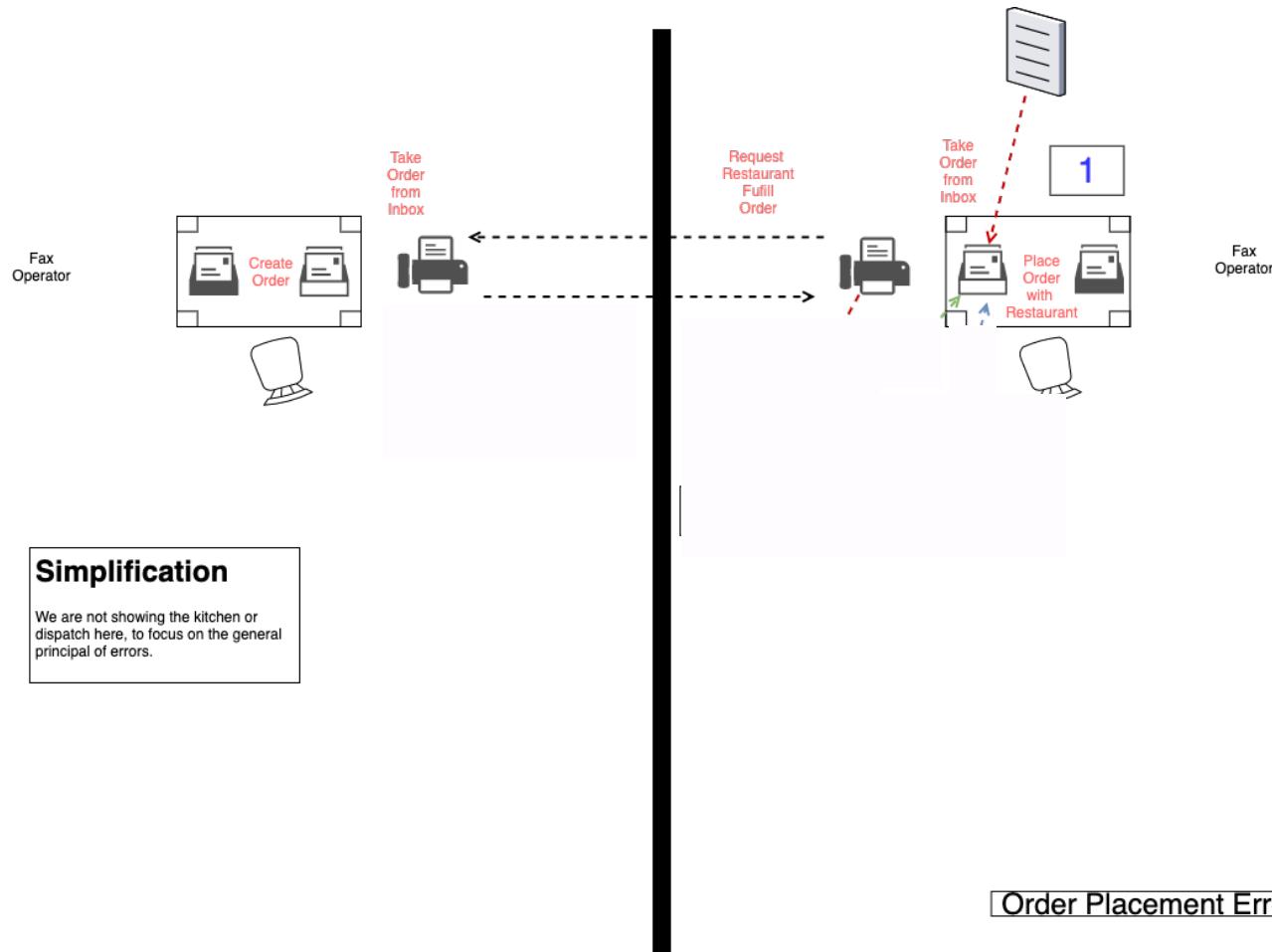


Fax Call Log

Monday, 2010-11-08 11:19

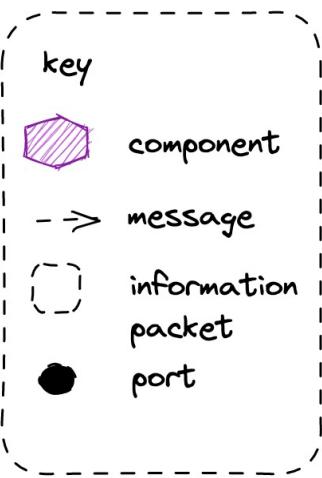
Date	Time	Type	Job #	Length	Speed	Station Name/Number	Pages	Status
2010-06-18	08:31	SCAN	92	0:25	28800	[REDACTED]	0	E-705 V.34 1M31
2010-03-22	11:39	RECV	59	0:20	26400	[REDACTED]	1	OK -- V.34 BM31
2010-03-22	11:45	RECV	60	0:20	26400	[REDACTED]	1	OK -- V.34 BM31
2010-03-22	12:29	RECV	61	0:21	26400	[REDACTED]	1	OK -- V.34 BM31
2010-09-14	14:46	SCAN	129	1:40	9600	[REDACTED]	2	E-606 V.29 AR30





Flow Based Programming

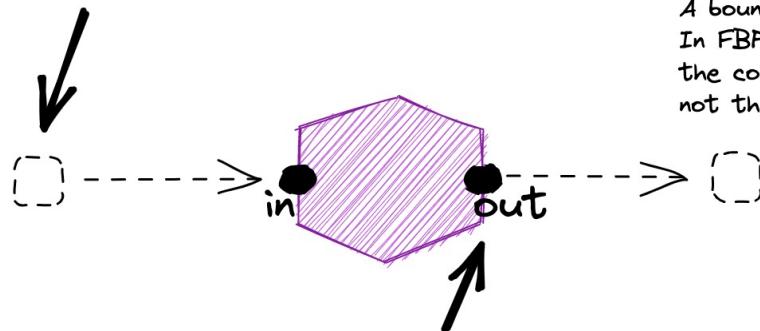
“Everything Flows and Nothing Stays.” - Heraclitus



Flow Based Programming (J. Paul Morrison)

Information Packet (IP)

An independent structured piece of information with a defined lifetime.



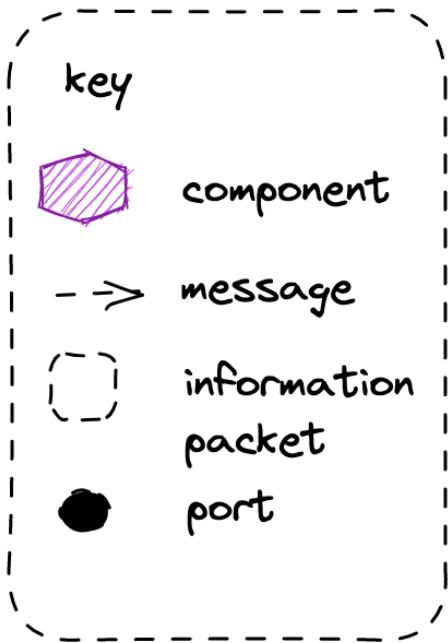
Port

The point where a connection makes contact with a process. A port is addressed by name.

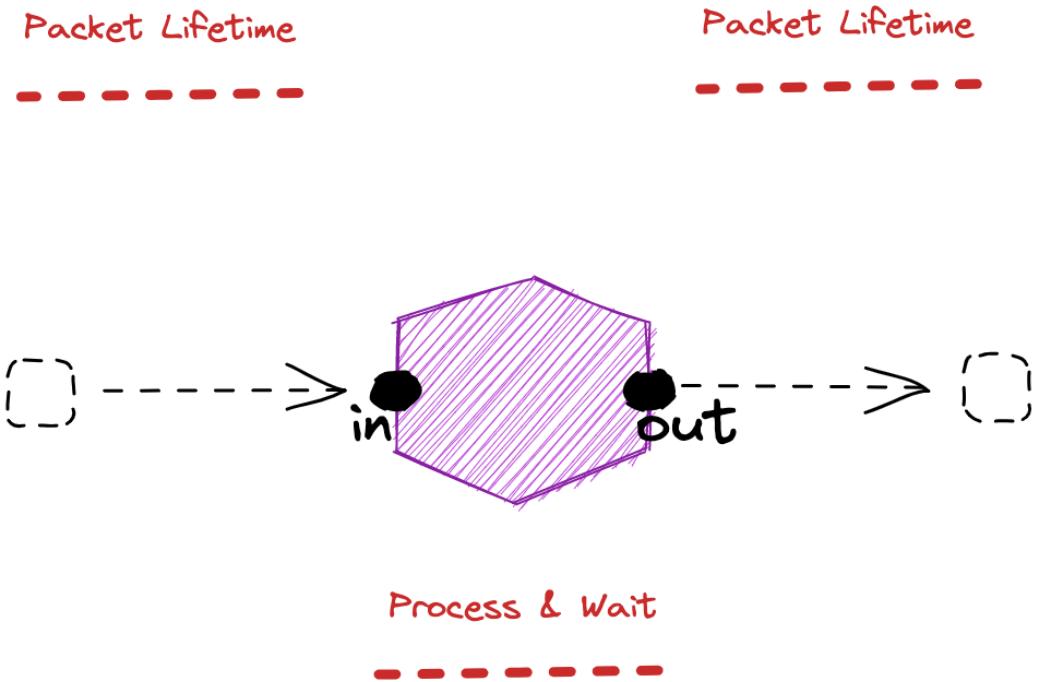
Flow-Based Programming (FBP) is a subclass of DFP. While DFP can be synchronous or asynchronous, FBP is always asynchronous. FBP allows multiple input ports, has bounded buffers and applies back pressure when buffers fill up.

Connector

A bounded pipe of information packets.
In FBP a connector is external - that is the component is only aware of the named port not the connector, so it could be swapped out



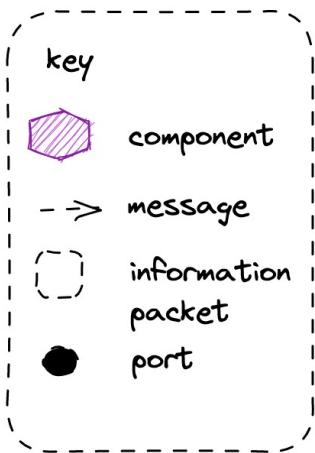
Flow Based Programming (J. Paul Morrison)



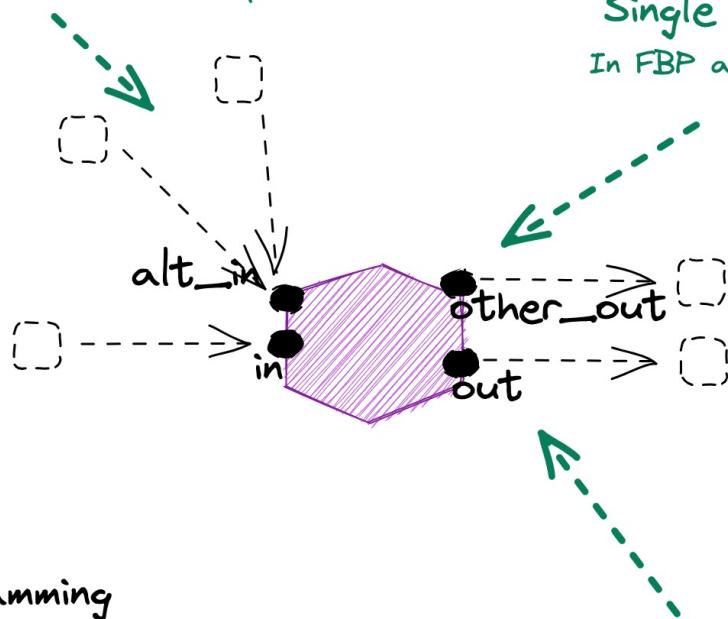
Node Lifetime: In flow base programming the a node can remain running whilst there is work on an input queue, can can suspend instead of terminate if there is no work on its connector.

Multiple Writers

In FBP an in port can have multiple writers



Flow Based Programming (J. Paul Morrison)



Single Writer

In FBP an out port is single writer

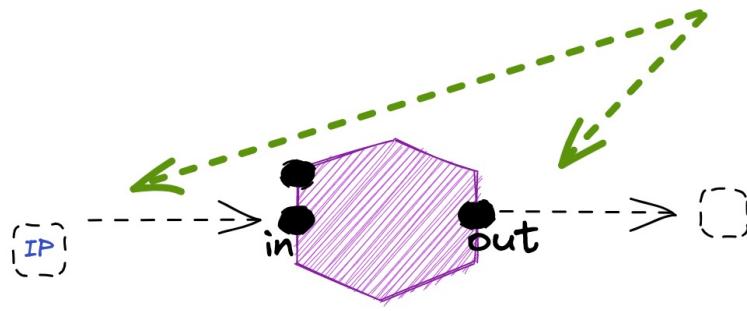
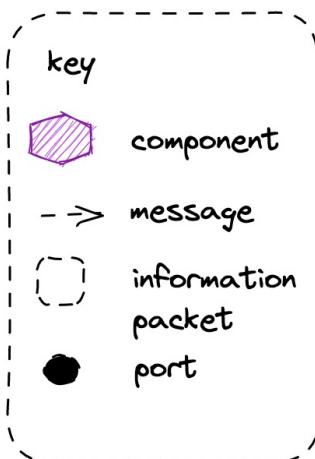
Multiple In/Out Ports

In FBP we can have multiple in or out ports

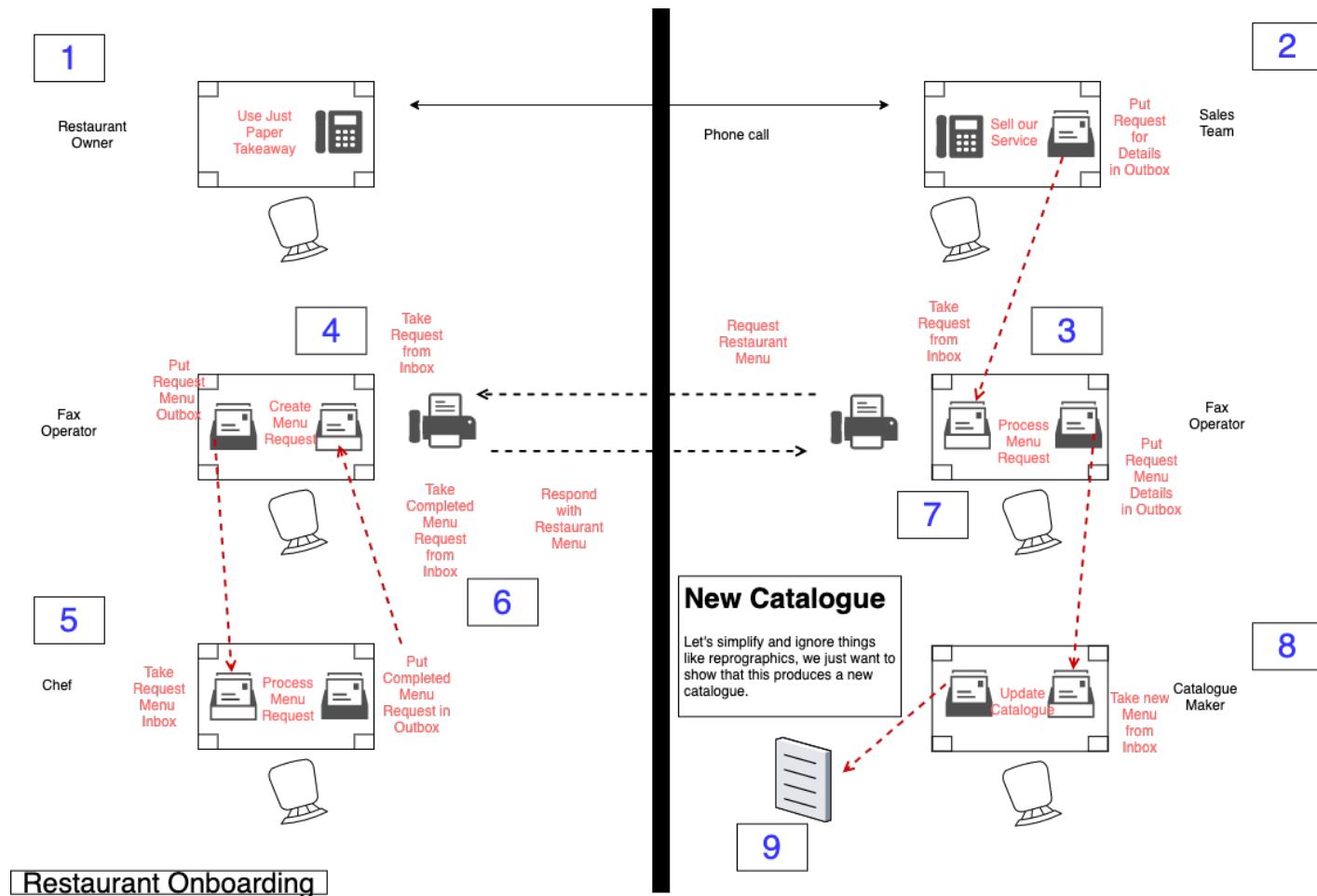
Capacity: In flow-based programming a component can test to see if it can send to out, and if not decide whether to halt or ignore.

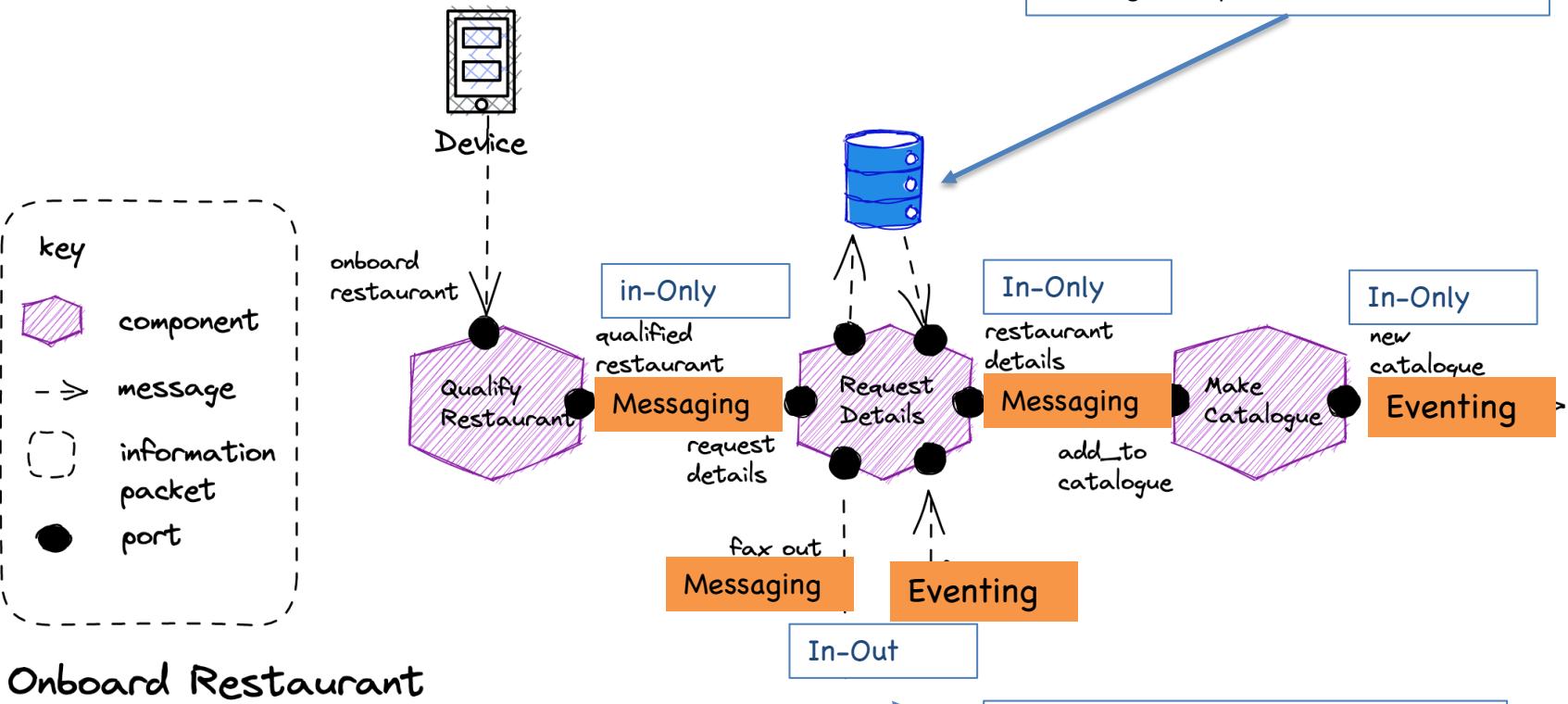
Message Oriented Middleware (MoM)

Because FBP allows for externally defined connectors in the 2010 update JPM outlined that it could be used in distributed systems. Nodes become processes and connectors use MoM.

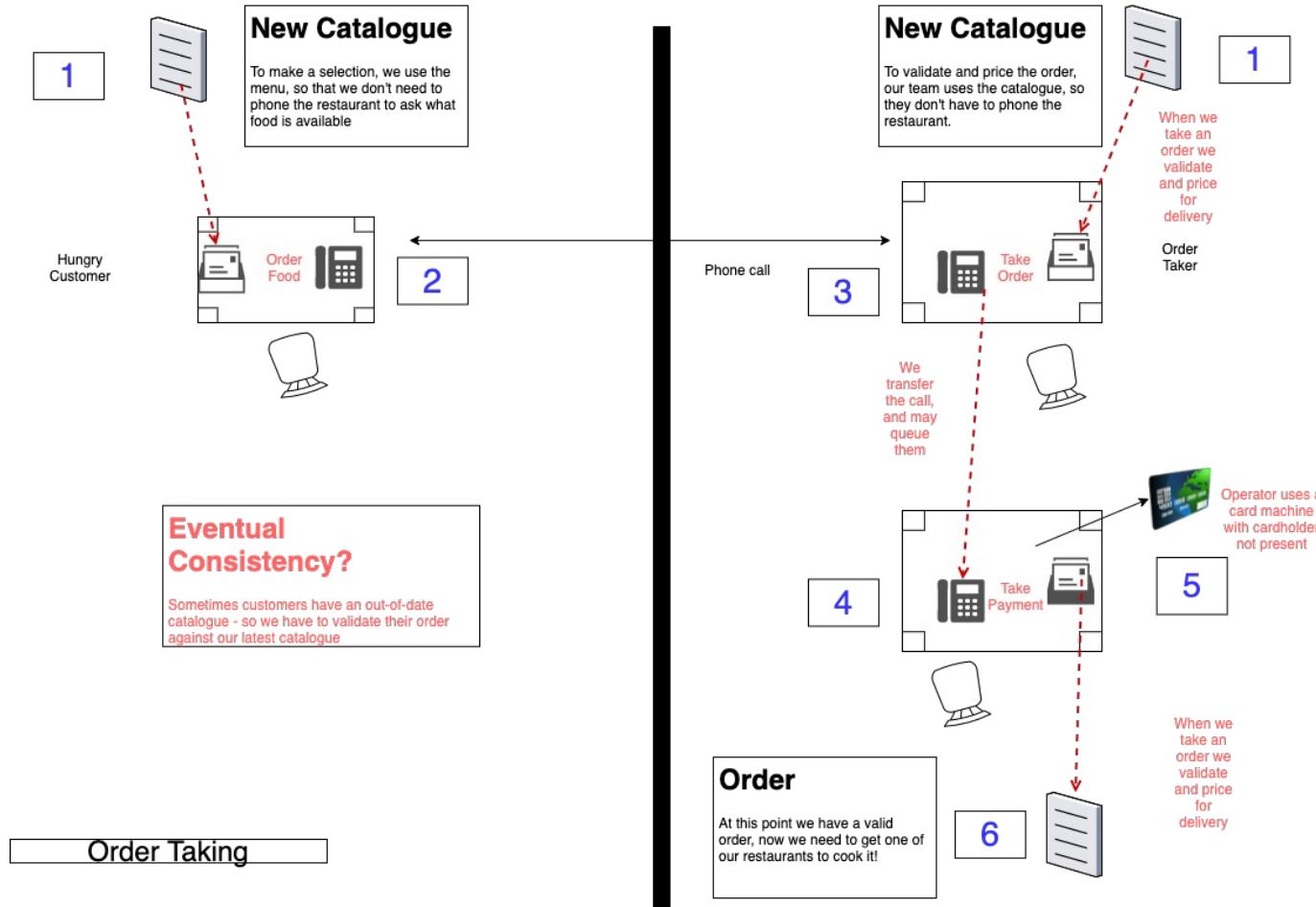


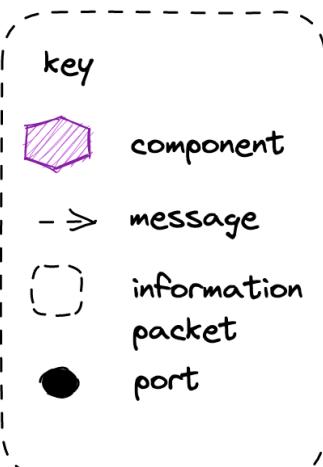
Flow Based Programming - 2010 Update
(J. Paul Morrison)



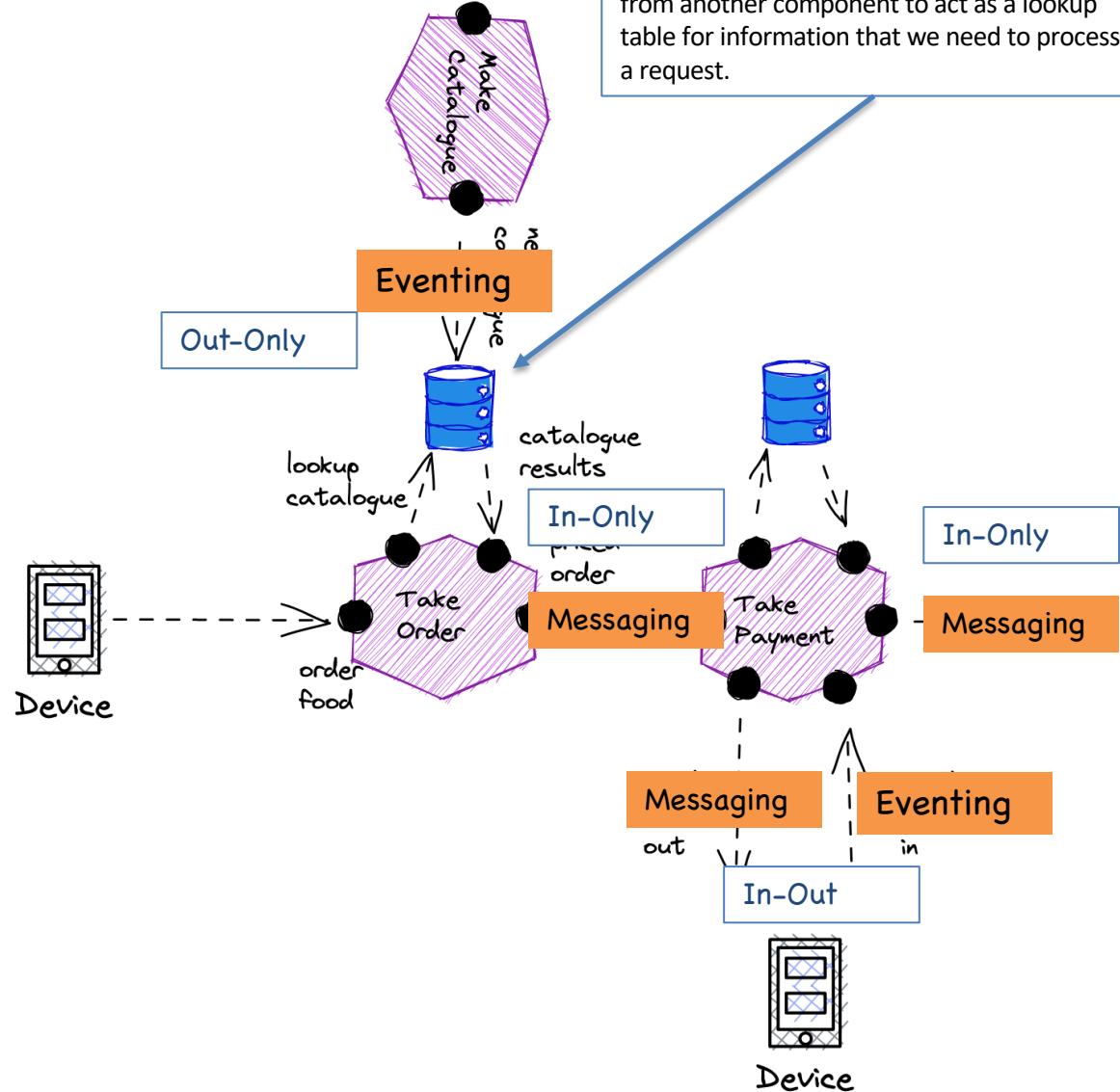


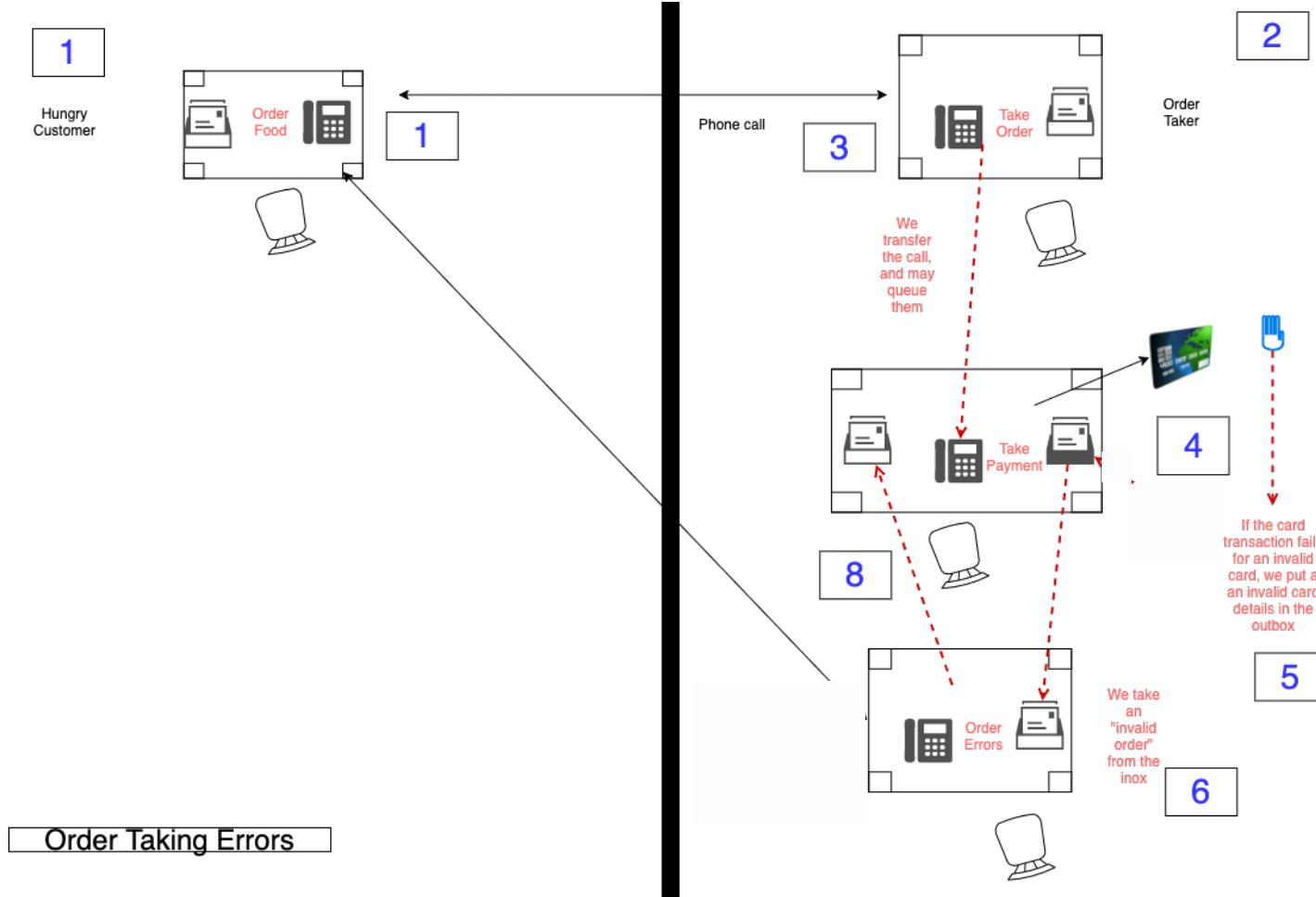
Storage: Request details may need storage to be reliable, it takes work from the **request_details** port, but cannot forward it to the **restaurant_details** port without waiting for an answer from **fax_in**. We store the state of the workflow in case we crash before receiving the response.

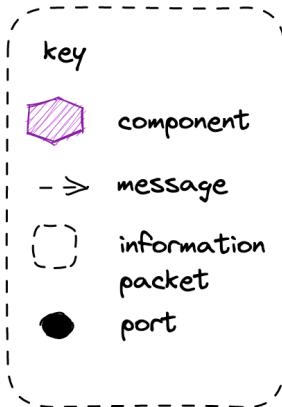




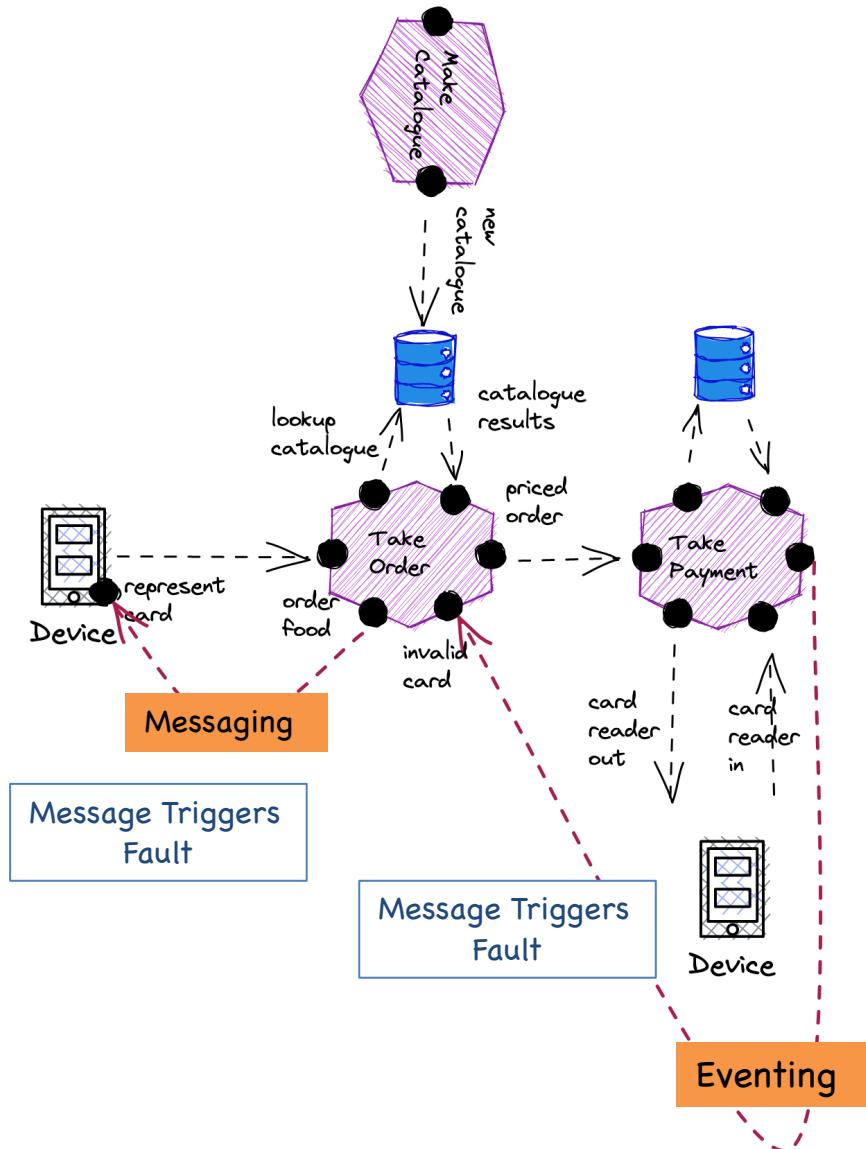
Order Food

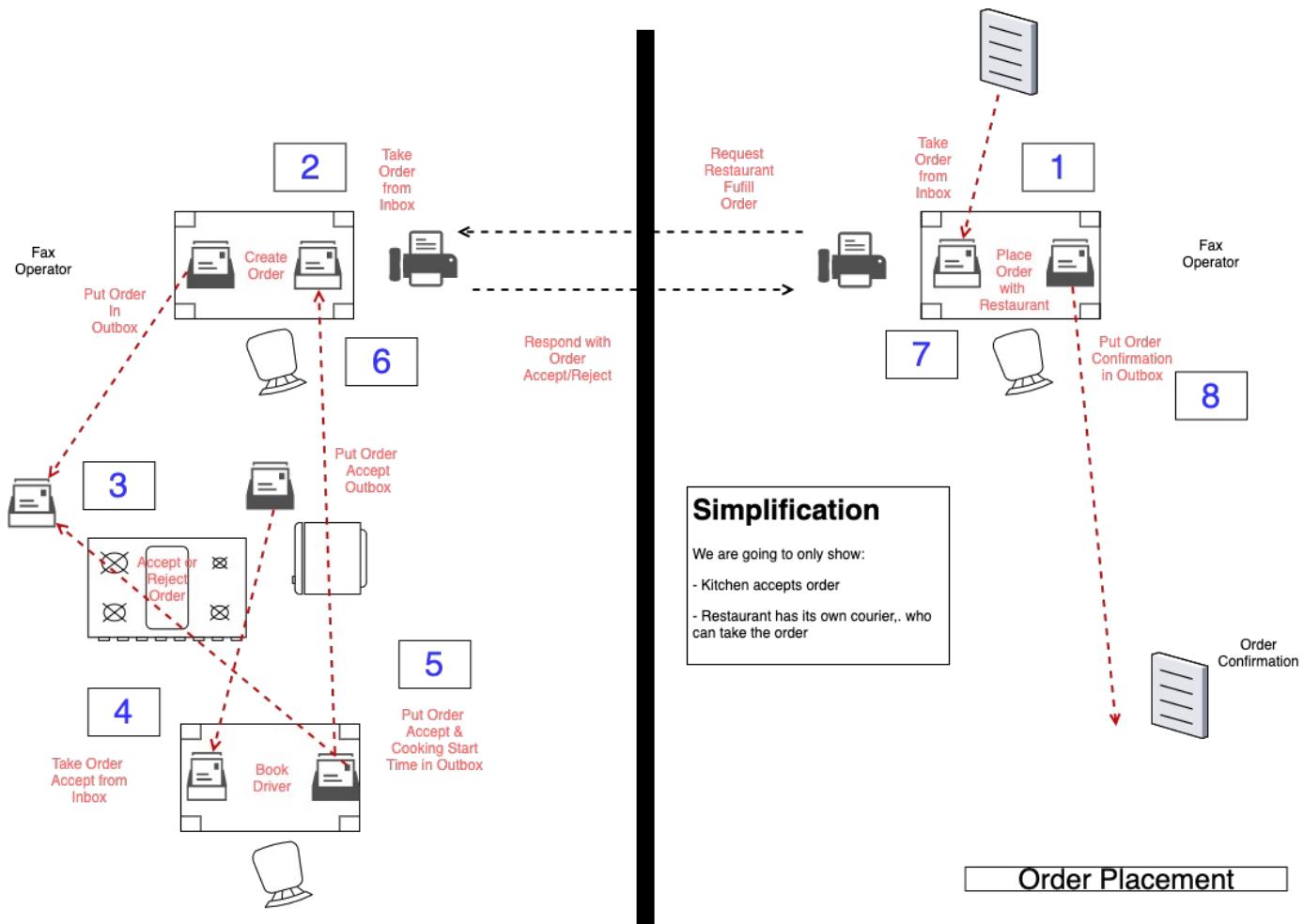


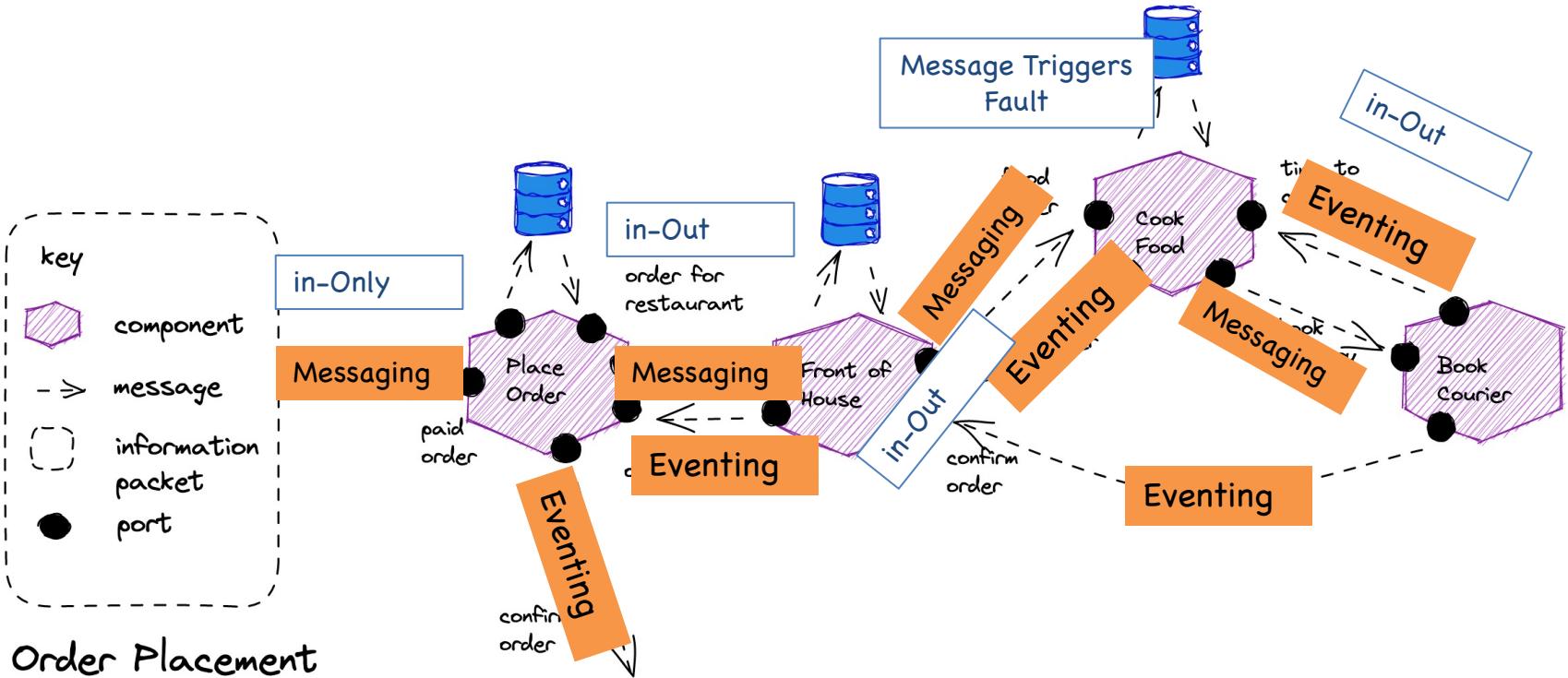


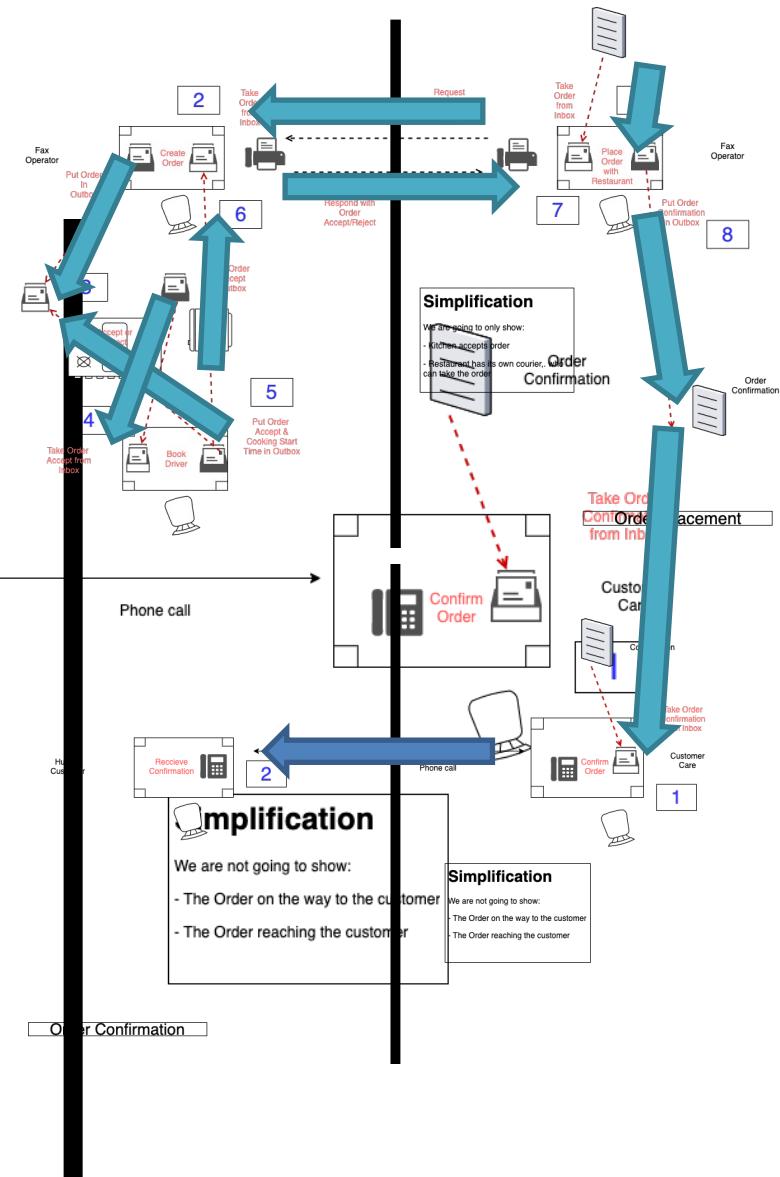
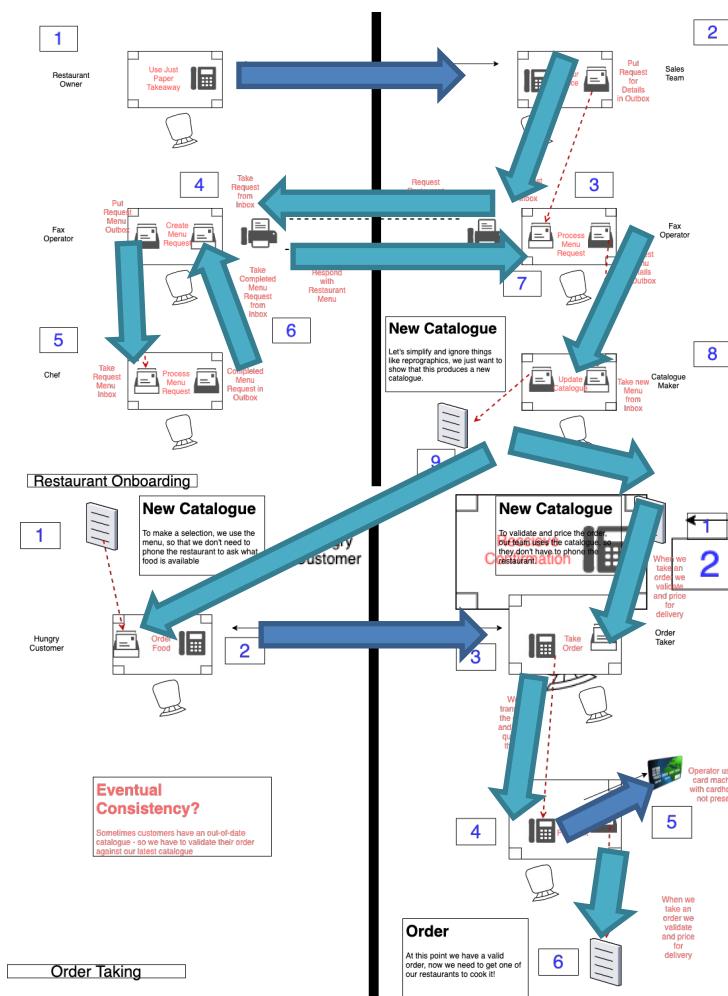


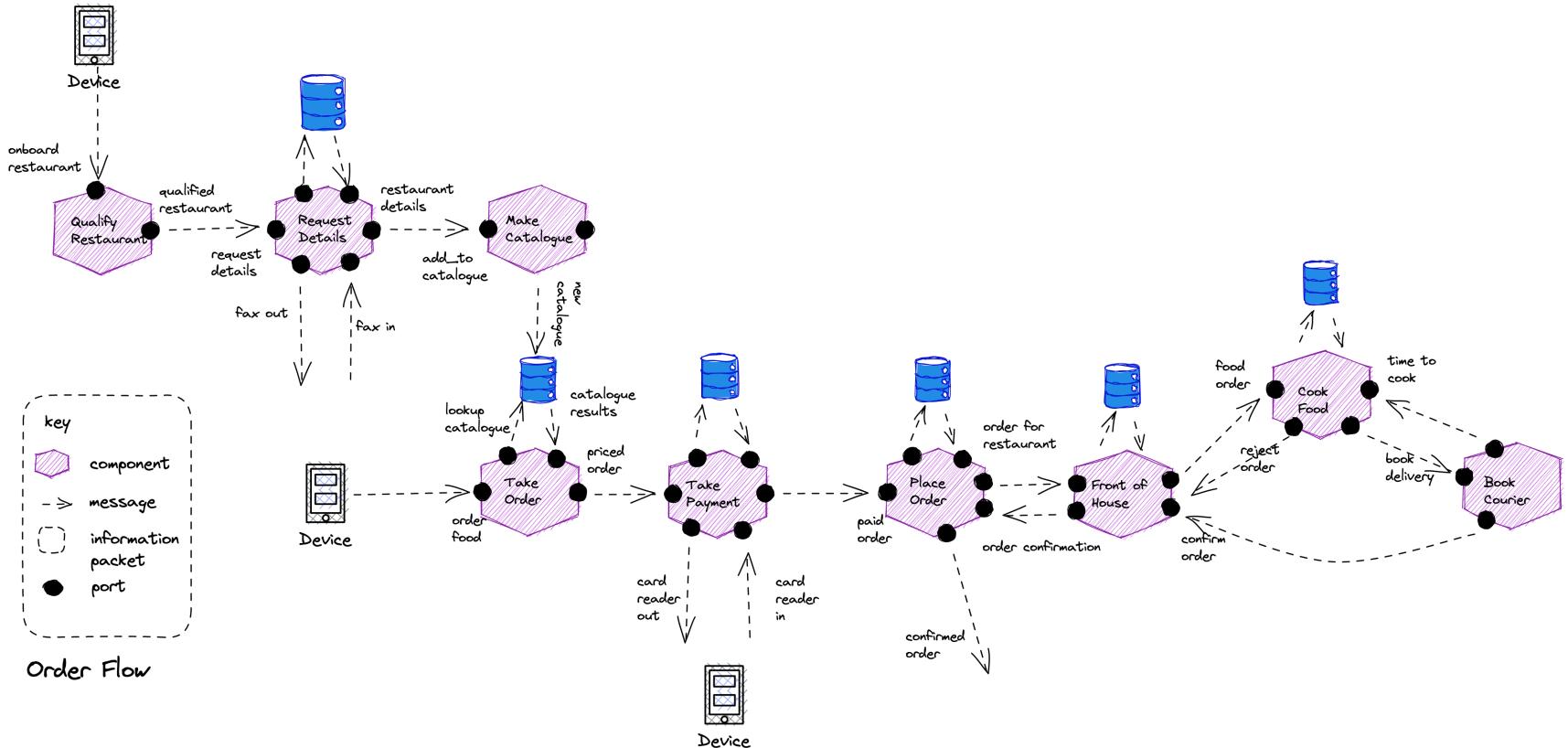
Order Errors











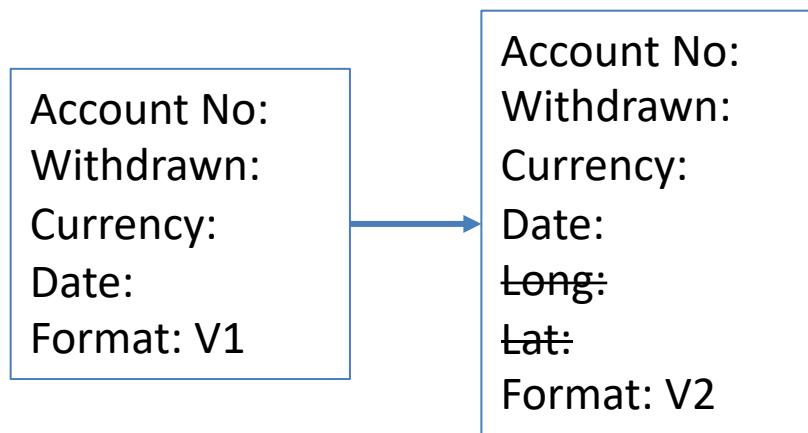
MANAGING ASYNCHRONOUS APIs

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.

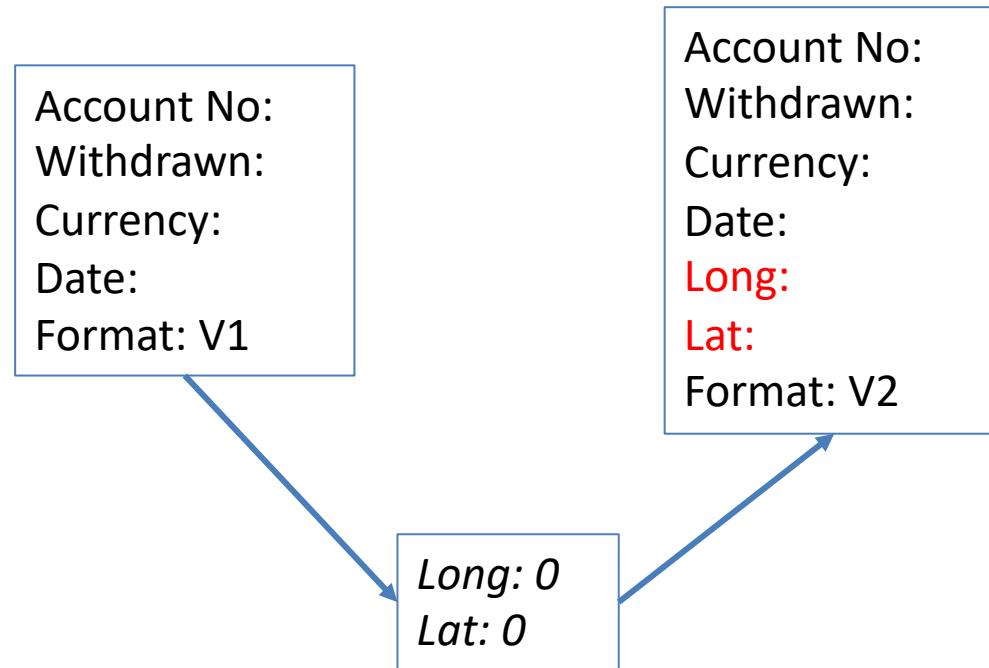
Robustness Principal or Postel's Law – Jon Postel RFC 1958

Tolerant Reader



Ignore New Fields

Tolerant Reader



Default Missing Fields

Breaking Change

Account No:
Withdrawn:
Currency:
Date:
Format: V1

Account.Withdrawal.Event

Account No:
New Balance:
Date:
Format: V2

Account.NewBalance.Event

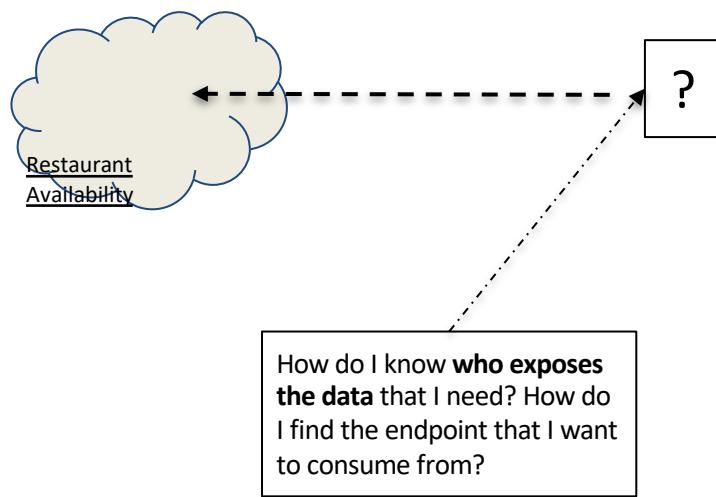
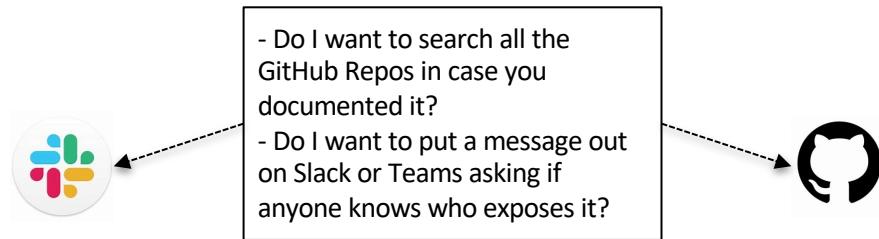
New Message

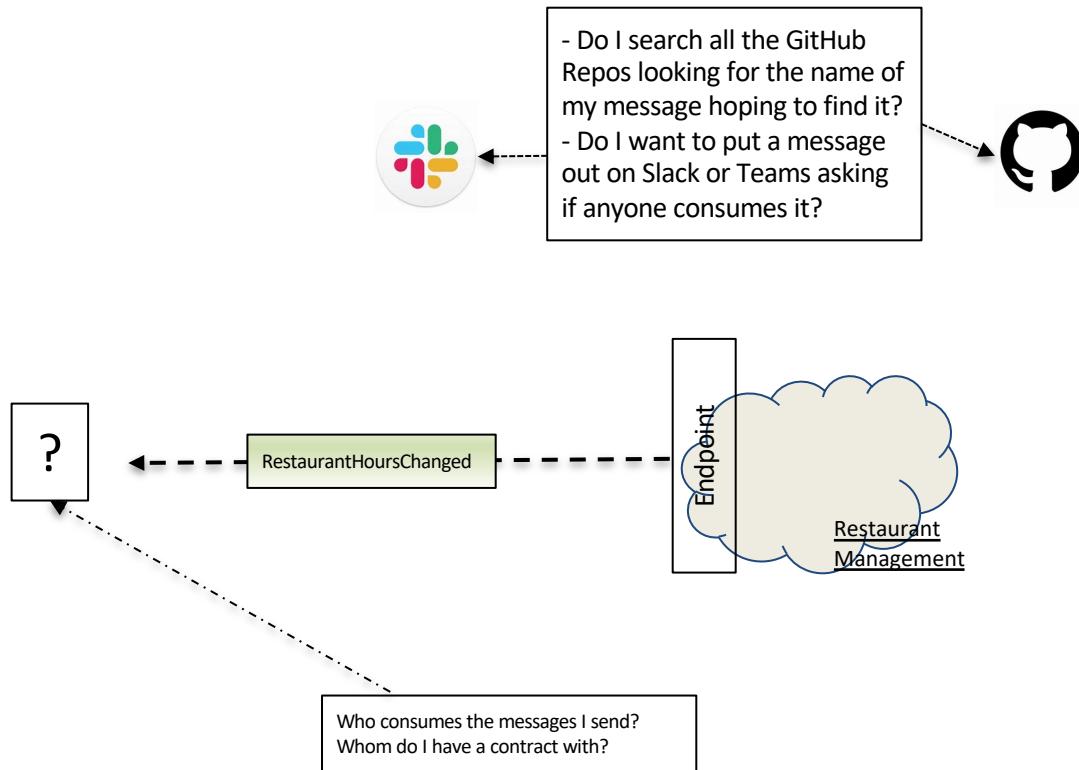
Documentation

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>

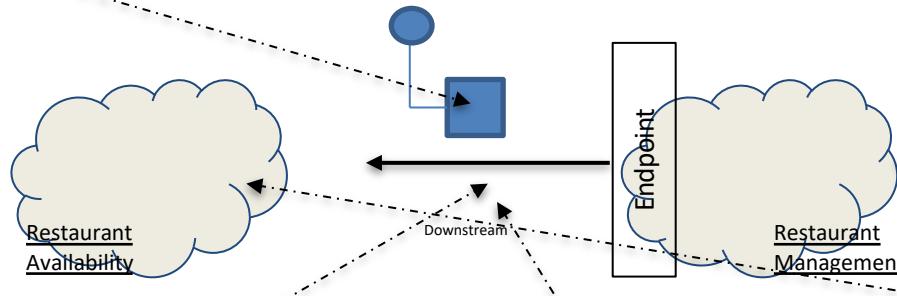




Asynchronous Endpoints

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

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

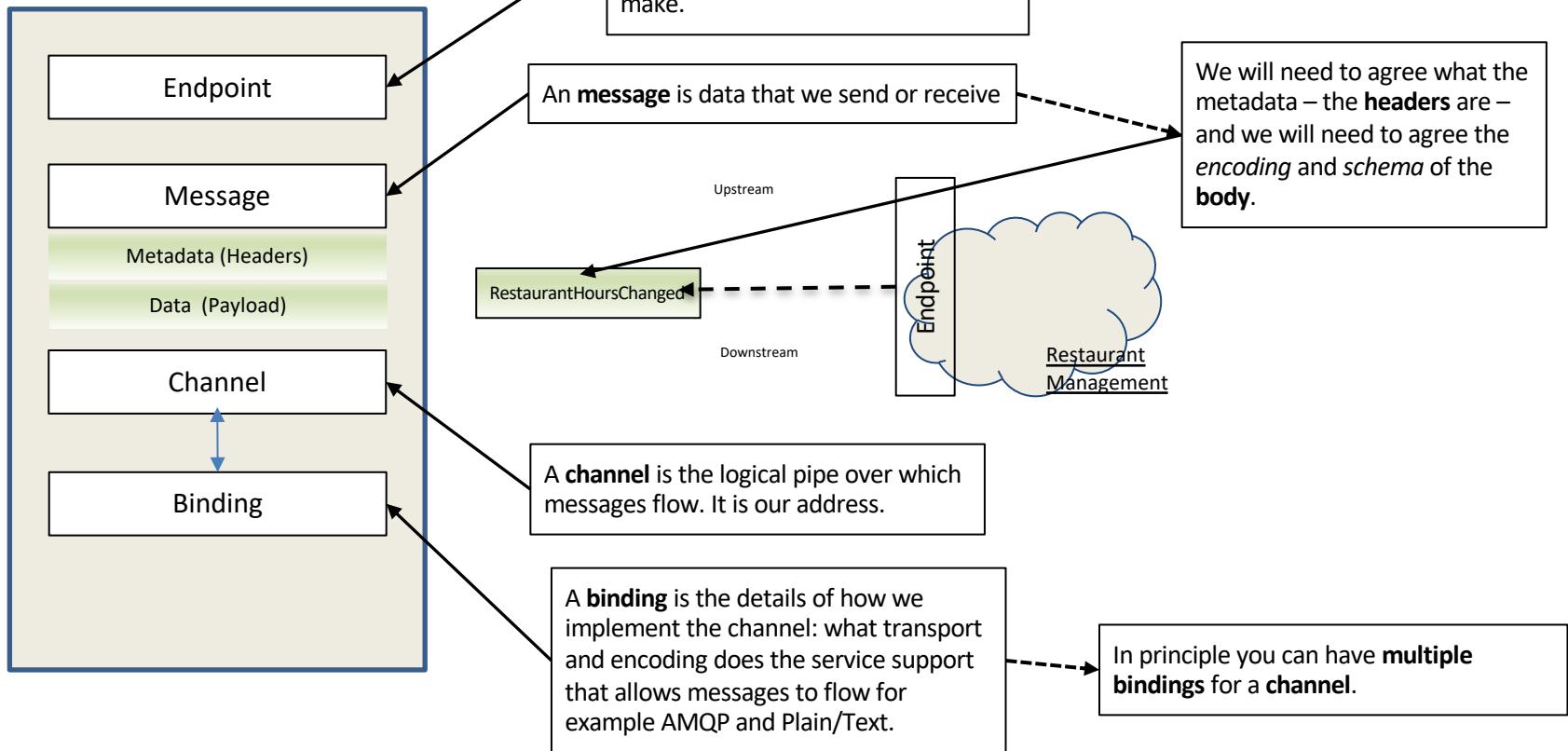


We need to document the channel so we know where the message is flowing

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



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](#)

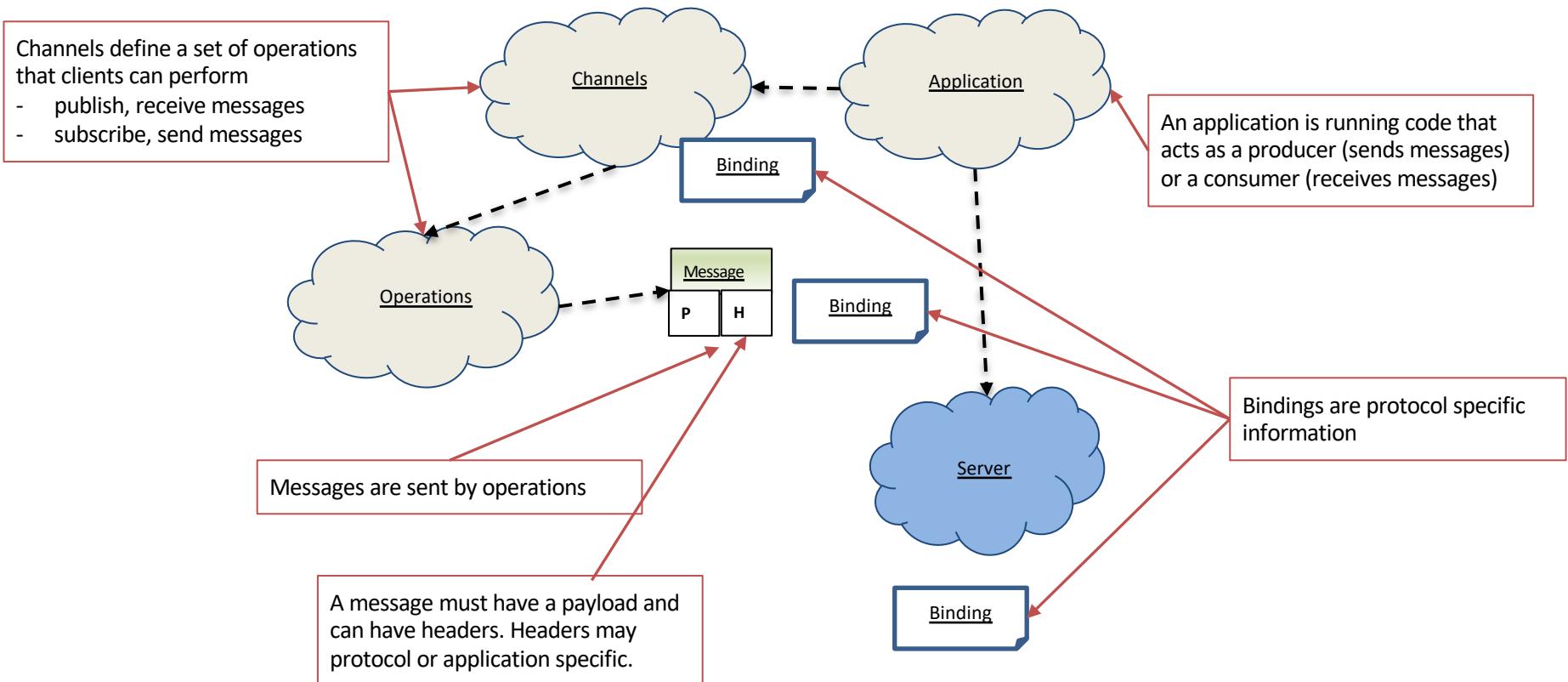
TSC
Members

And much more...

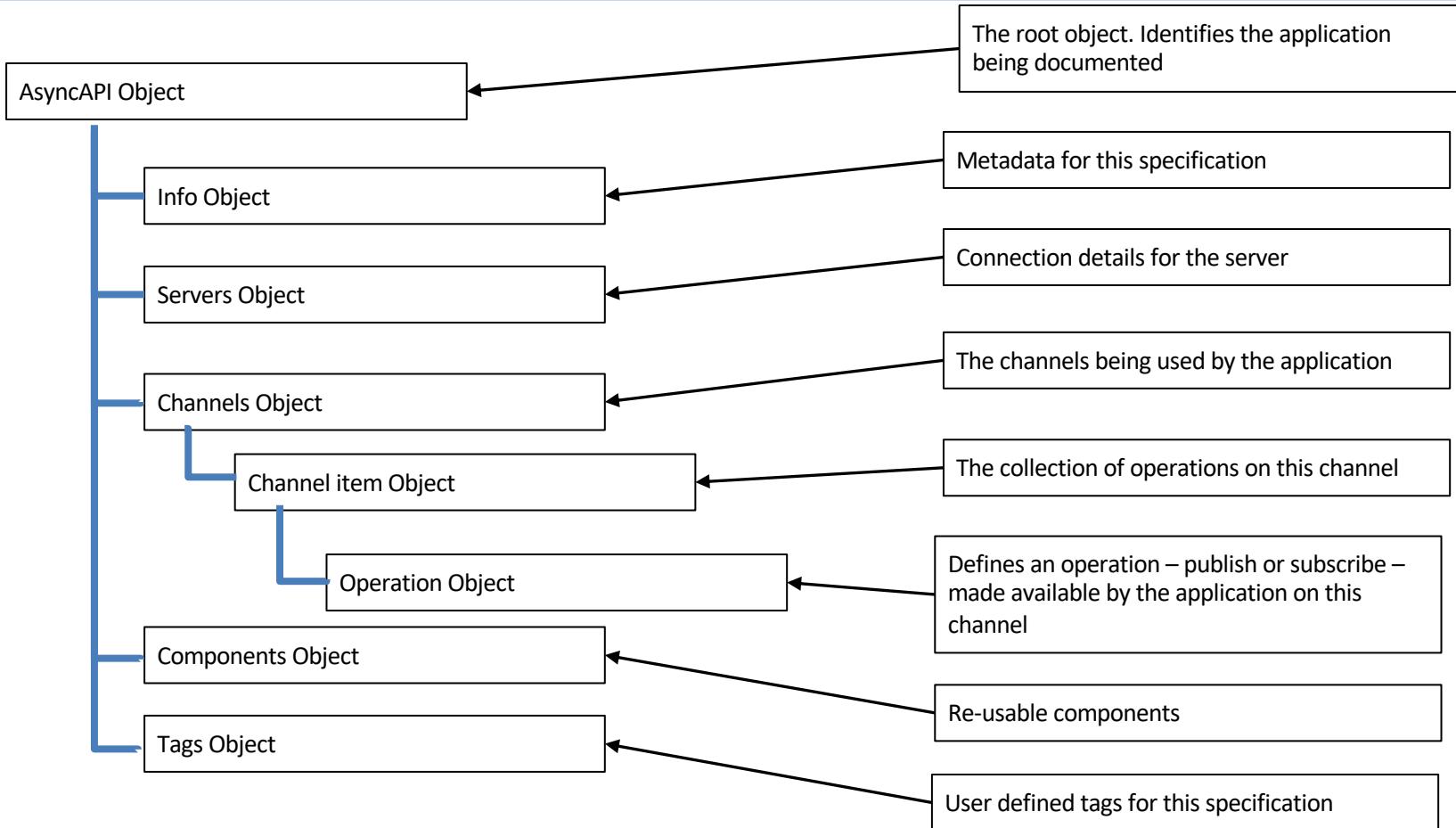
We have many different tools and welcome you to explore our ideas and propose new ideas to AsyncAPI.

[View GitHub Discussions](#)

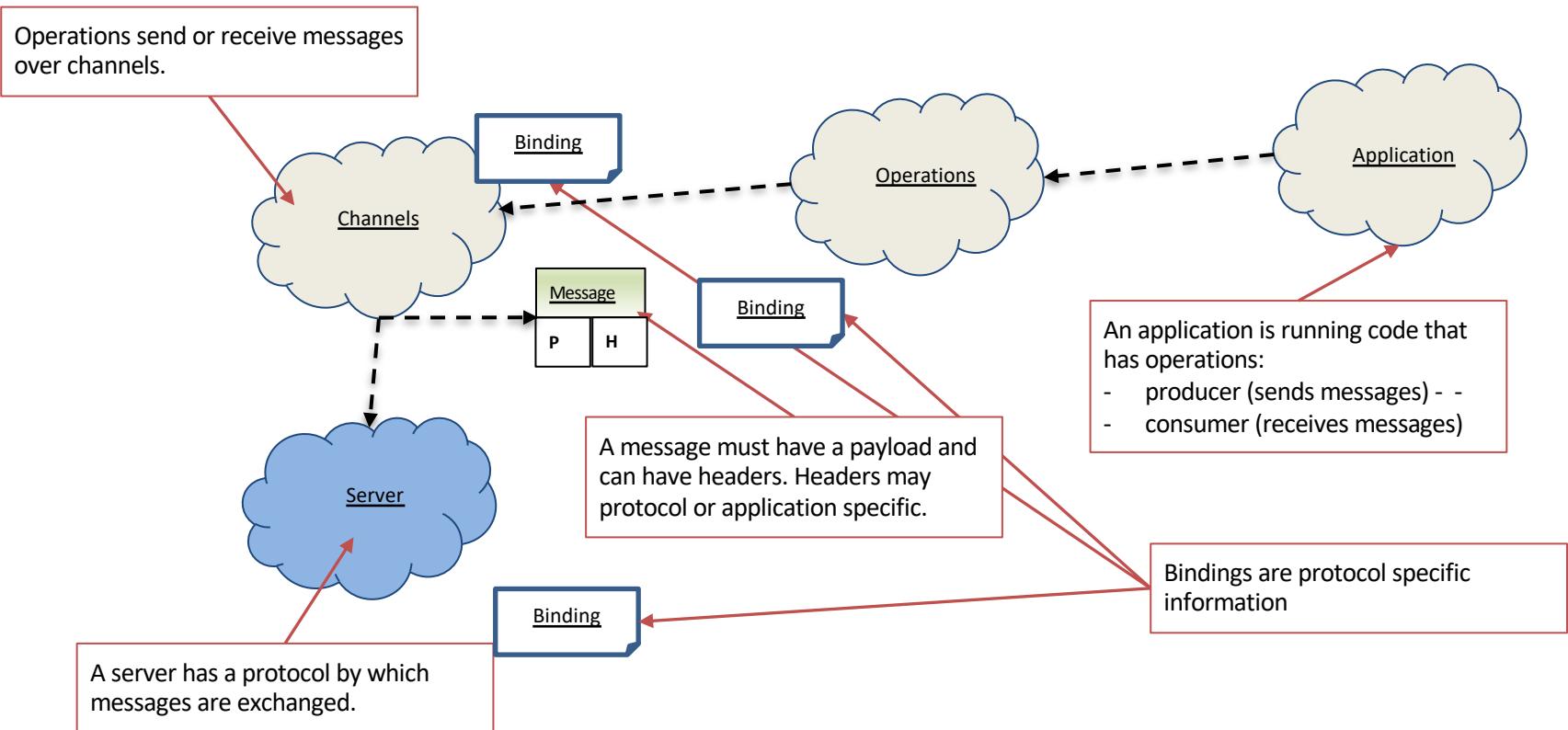
AsyncAPI Elements (V2)



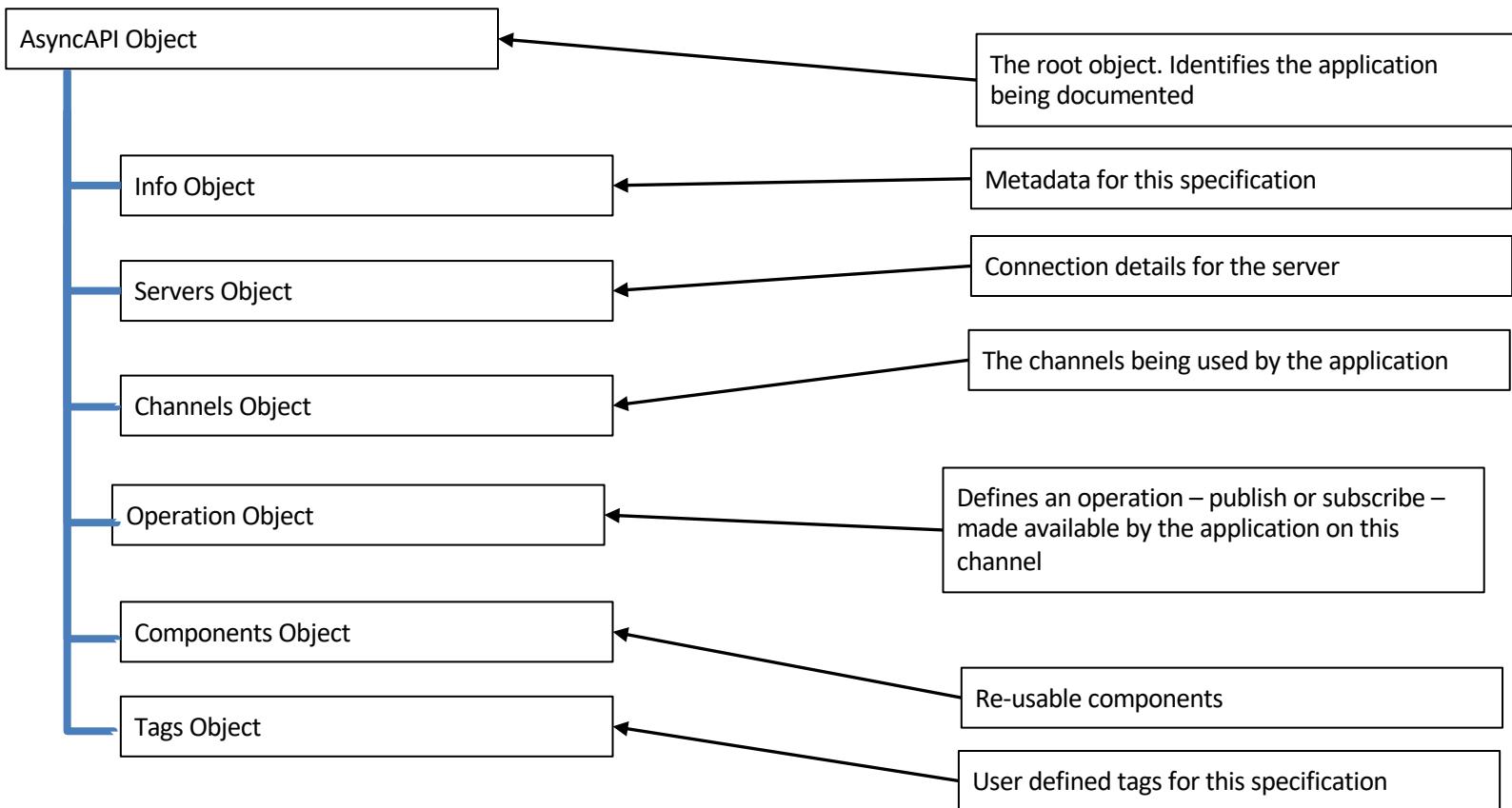
Document Structure (V2)



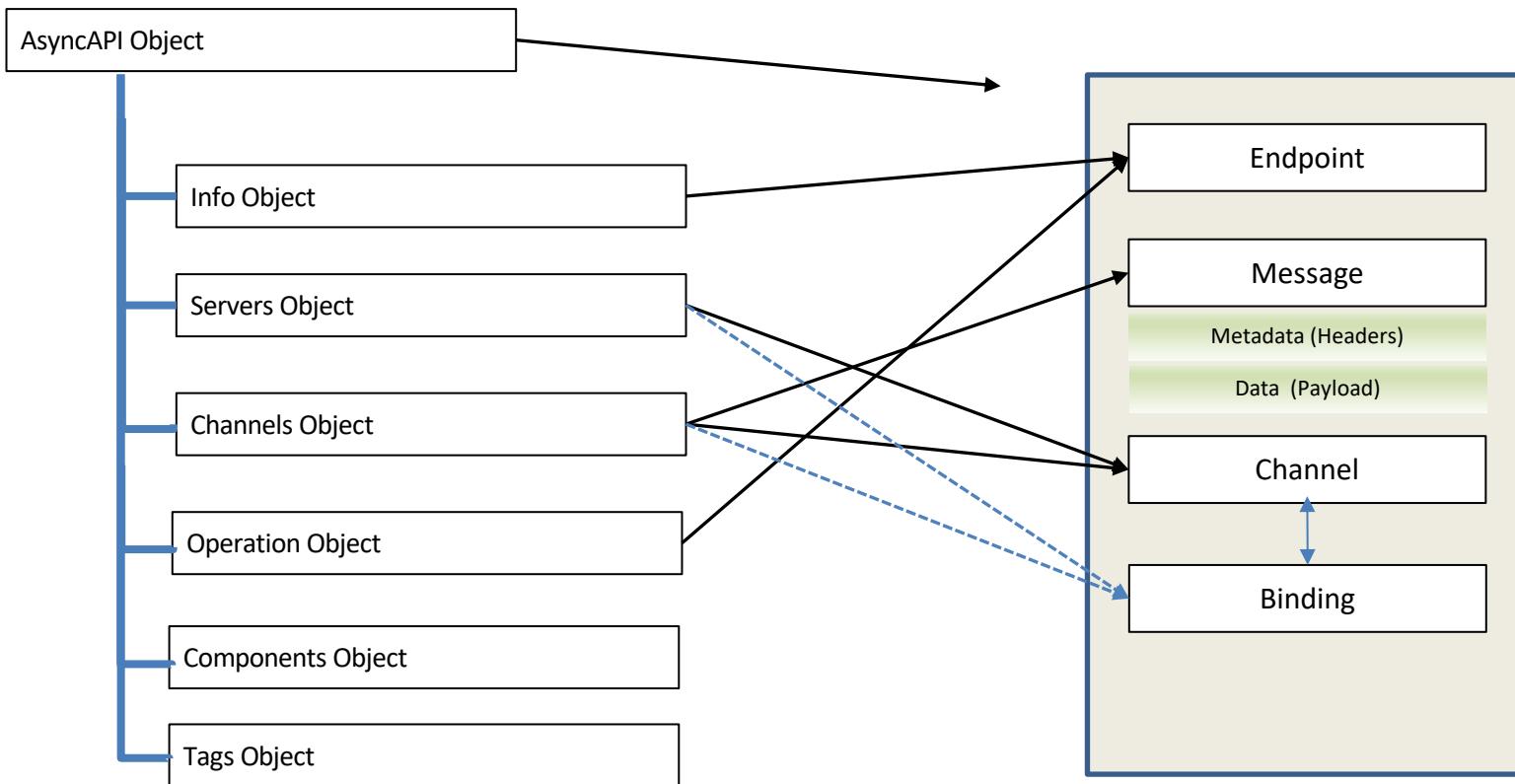
AsyncAPI Elements (V3)



Document Structure (V3)



AsyncAPI (V3) and Endpoint ABCs



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

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 (V2)

```
greeting:  
subscribe:  
    operationid: sendGreeting  
    summary: sends a greeting  
    description: This service lets you send the 'Hello World' greeting to another service.  
message:  
    $ref: "#/components/messages/greeting"  
bindings:  
    kafka:  
        partitions: 20  
        replicas: 3
```

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

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

Encodings:

JSON

Binary

Languages:

C

C++

C#

Java

Perl

Python

Ruby

Others...

```
syntax = "proto3";  
  
message Greeting {  
    string greeting = 1;  
}
```

Protobuf

Encodings:

Binary

Languages:

C++

C#

Dart

Go

Kotlin

Java

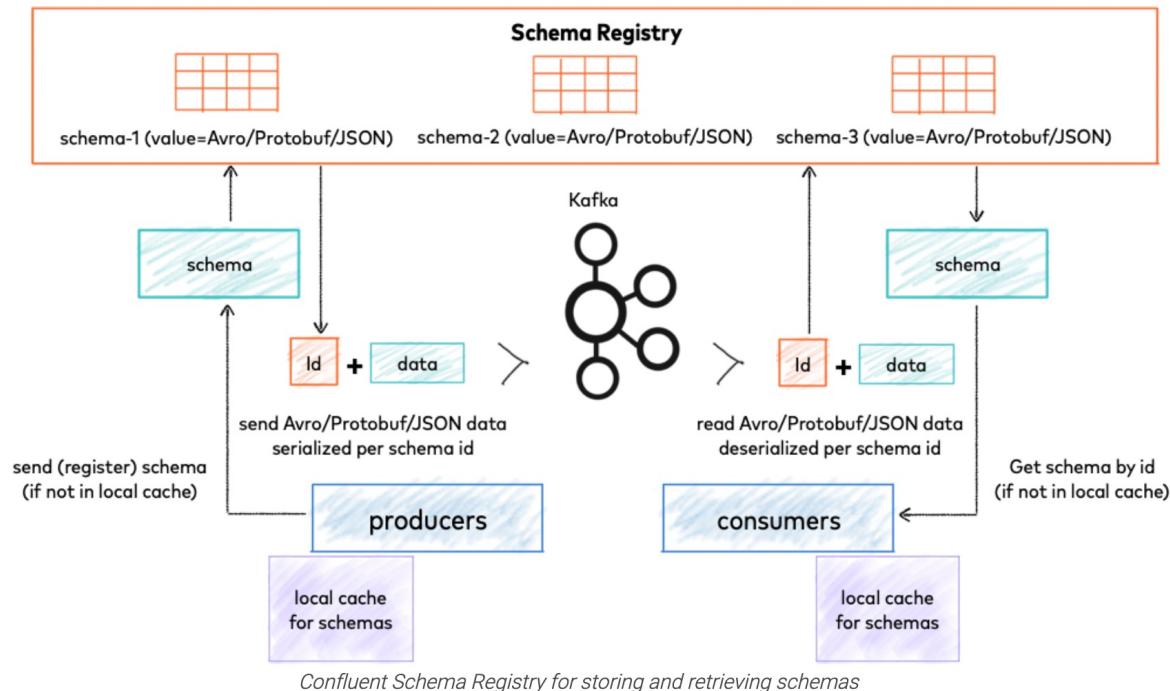
Objective-C

Python

Ruby

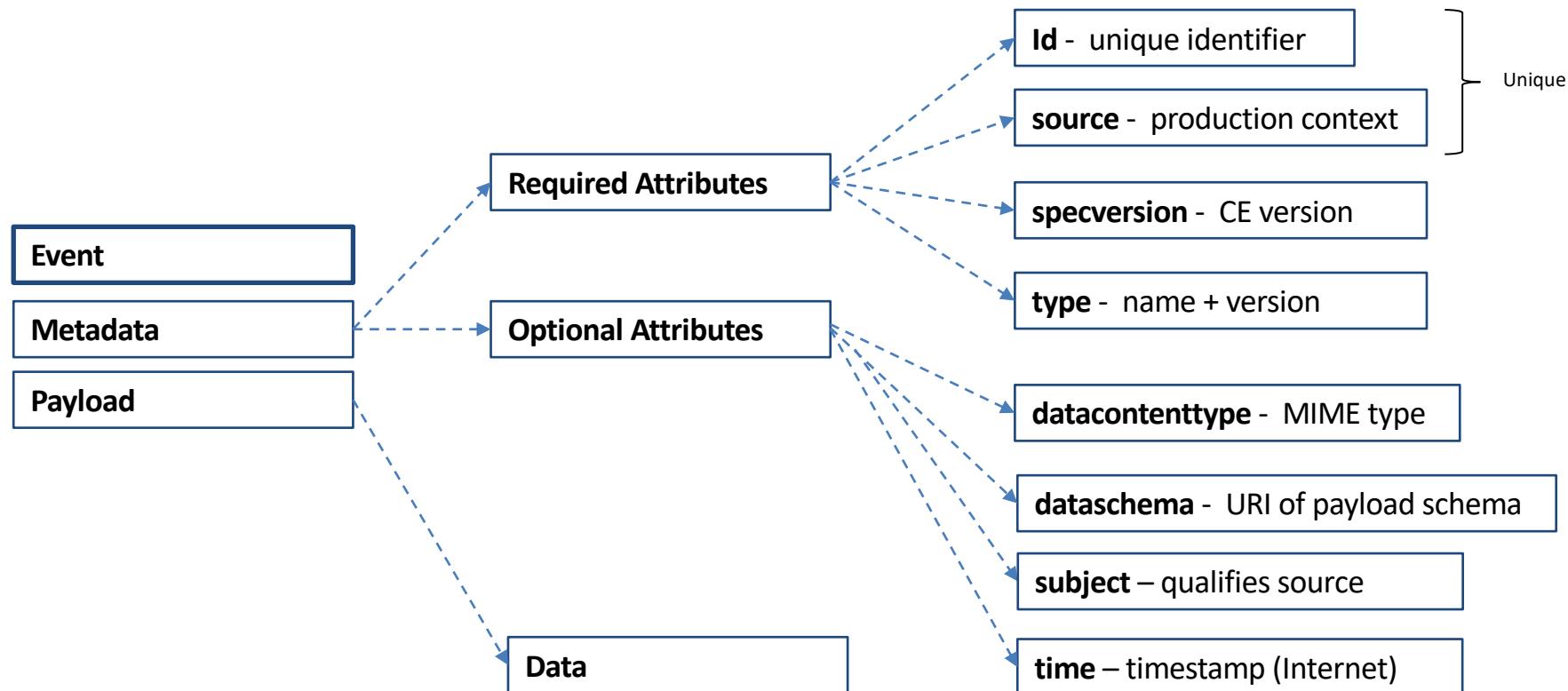
Others...

Schema Registry

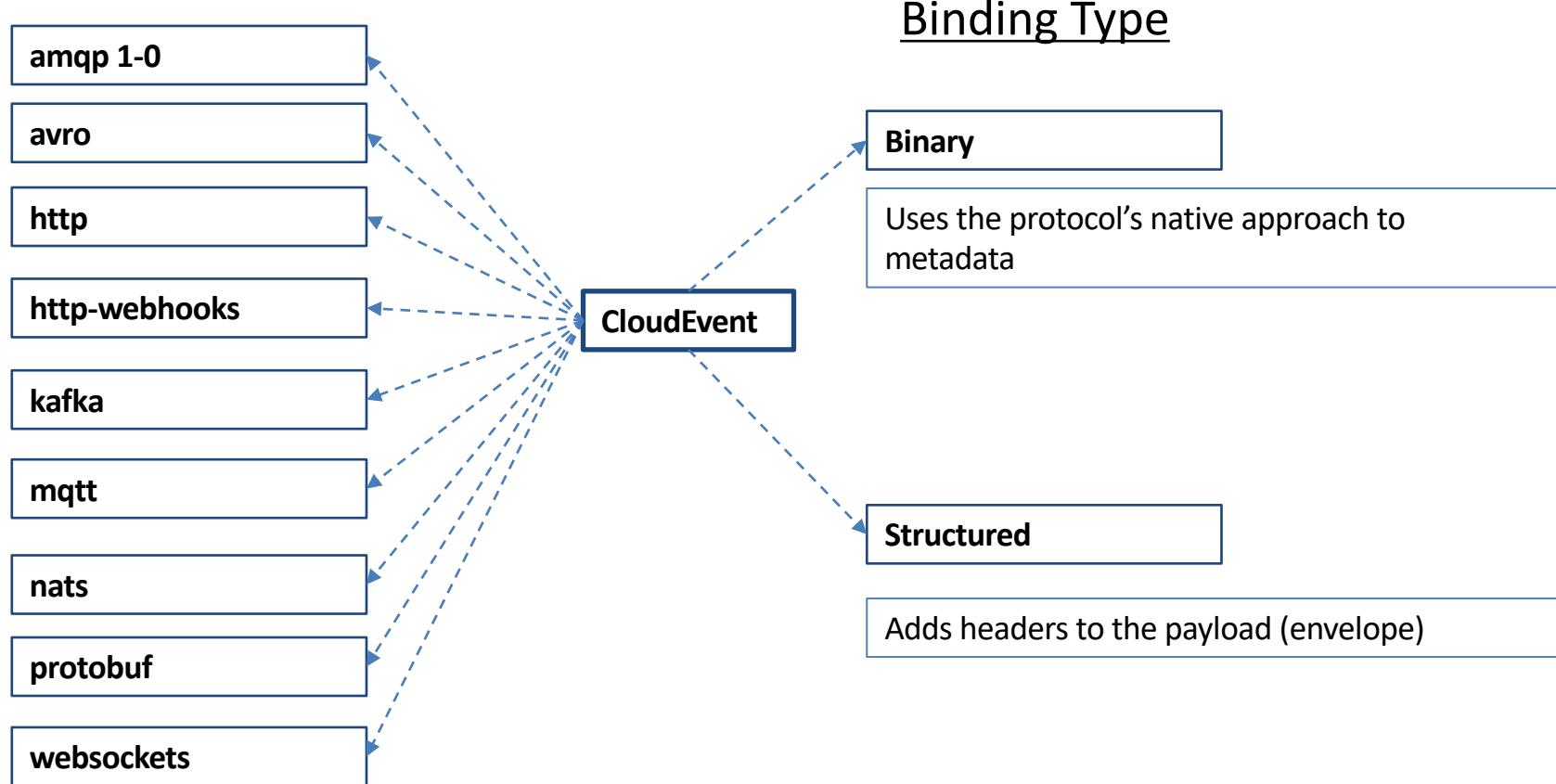


<https://docs.confluent.io/platform/current/schema-registry/index.html>

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 Studio

AsyncAPI studio beta

From localStorage

```
1  asyncapi: '2.0.0'
2  id: "https://github.com/brightercommand/greetings-sender/"
3  info:
4    contact:
5      name: Paramore Brighter
6      url: https://goparamore.io/support
7      email: support@goparamore.io
8    license:
9      name: Apache 2.0
10   url: https://www.apache.org/licenses/LICENSE-2.0.html
11   description: Demonstrates sending a greeting over a messaging
12     transport.
13   title: Brighter Sample App
14   version: 1.0.0
15   tags:
16     - name: brighter examples
17   servers:
18     localhost:
19       url: localhost:9092
20       protocol: kafka
21
22   channels:
23     greeting:
24       description: A channel for sending out greeting messages
25       subscribe:
26         description: This service lets you send the 'Hello World'
27           greeting to another service.
28         operationId: sendMessage
29         message:
30           $ref: "#/components/messages/greeting"
31
32   components:
33     messages:
34       greeting:
35         name: greeting
36         title: A salutation
37         summary: This is how we send you a salutation
38         contentType: application/json
39         traits:
40           - $ref: "#/components/messageTraits/commonHeaders"
41         payload:
42           $ref: "#/components/schemas/greetingContent"
43
44   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

VS Code

The screenshot shows a split-screen view of VS Code. On the left, the code editor displays the `brighter-greetings-sender.yaml` file, which defines a service for sending greetings over a Kafka transport. On the right, the generated AsyncAPI documentation is shown, featuring a title, contact information, and sections for Servers and Operations.

brighter-greetings-sender.yaml

```
! brighter-greetings-sender.yaml × Extension: AsyncAPI Preview ! brighter-greeting-rec ...
! brighter-greetings-sender.yaml > YAML > {} info > {} contact > url ...
1  asyncapi: '2.0.0'
2  id: "https://github.com/brightercommand/greetings-sender/"
3  info:
4    contact:
5      name: Paramore Brighter
6      url: https://goparamore.io/support
7      email: support@goparamore.io
8    license:
9      name: Apache 2.0
10   url: https://www.apache.org/licenses/LICENSE-2.0.html
11   description: Demonstrates sending a greeting over a messaging transpo
12   title: Brighter Sample App
13   version: 1.0.0
14   tags:
15     - name: brighter examples
16
17   servers:
18     localhost:
19       url: localhost:9092
20       protocol: kafka
21
22   channels:
23     greeting:
24       description: A channel for sending out greeting messages
25       subscribe:
26         description: This service lets you send the 'Hello World' greeting
27         operationId: sendMessage
28         message:
29           $ref: '#/components/messages/greeting'
30
31   components:
32     messages:
33       greeting:
34         name: greeting
35         title: A salutation
36         summary: This is how we send you a salutation
37         contentType: application/json
38         traits:
39           - $ref: '#/components/messageTraits/commonHeaders'
40         payload:
41           $ref: "#/components/schemas/greetingContent"
42
43     schemas:
44       greetingContent:
```

AsyncAPI - brighter-greetings-sender.yaml

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

Minikube -- NORMAL --

AsyncAPI Preview ✓ Spell 🔍 ⚡

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.
- `svncapi` - 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

Overall Shape Of An Entity

Substitutions In The Descriptor Format

Common to All Kinds: The Envelope

`apiVersion` and `kind` [required]

`metadata` [required]

`spec` [varies]

Common to All Kinds: The Metadata

`name` [required]

`namespace` [optional]

`title` [optional]

`description` [optional]

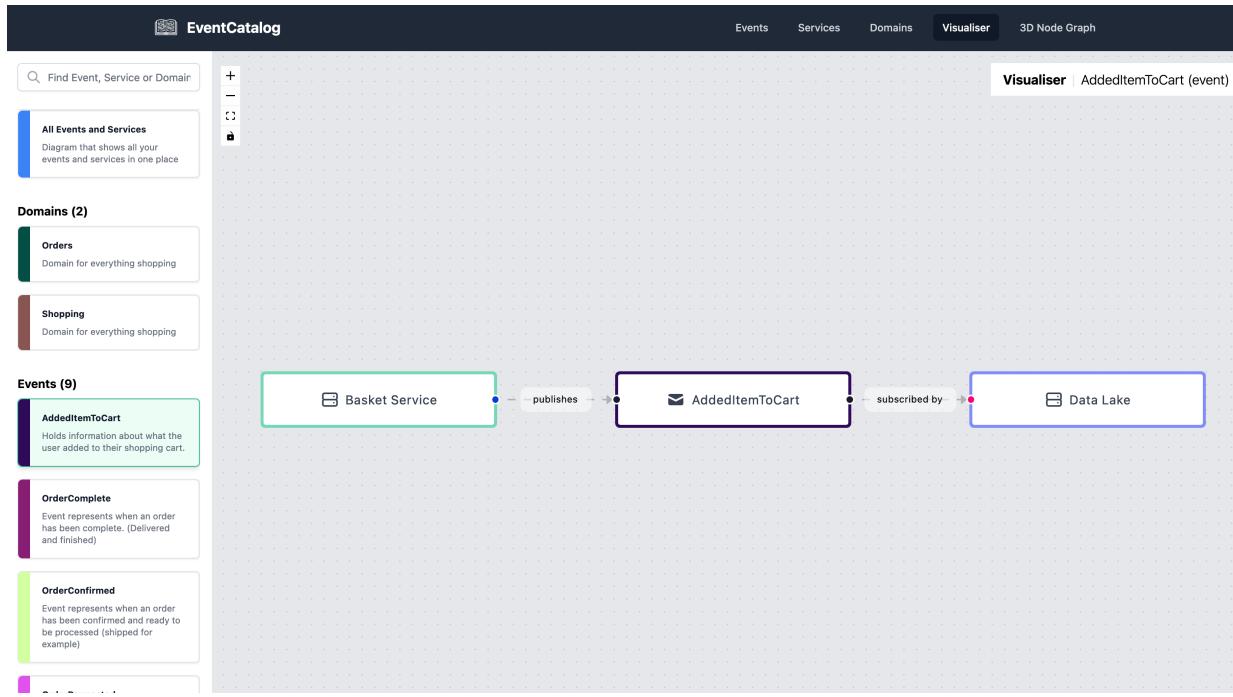
`labels` [optional]

`annotations` [optional]

`tags` [optional]

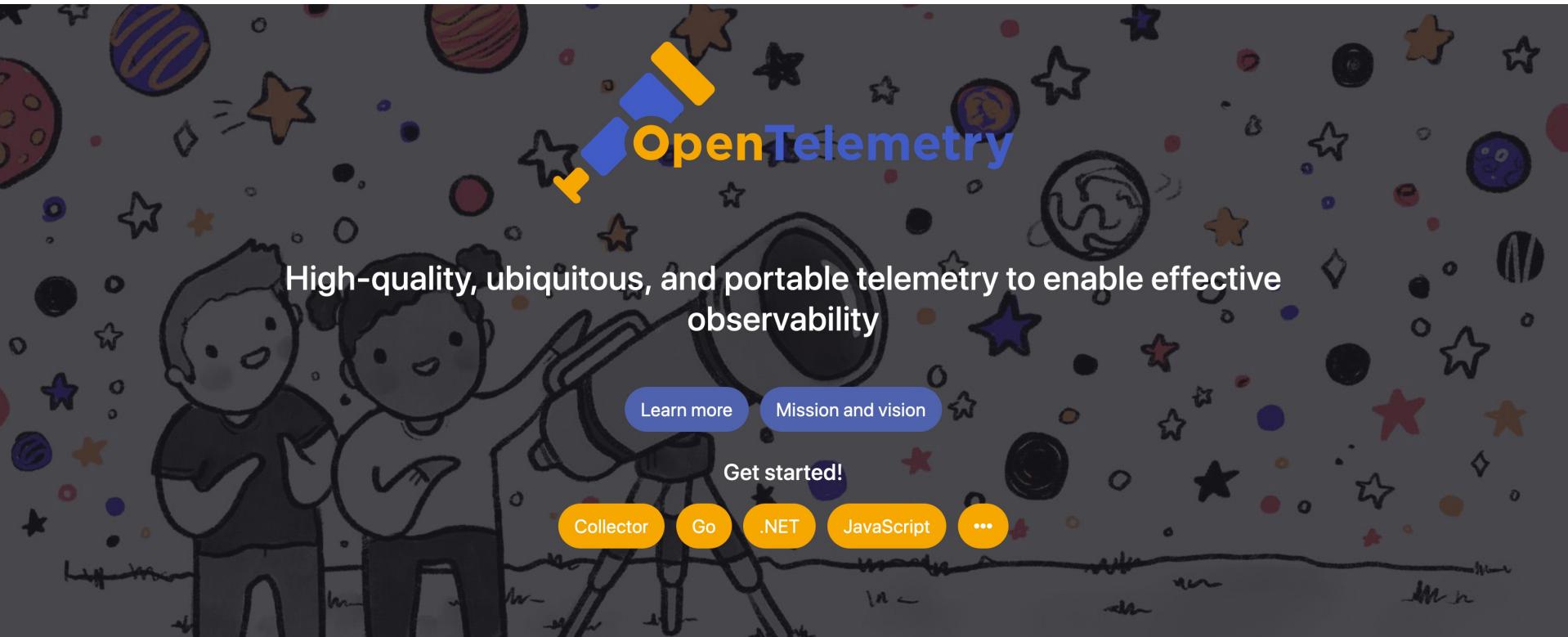
`links` [optional]

Event Catalog

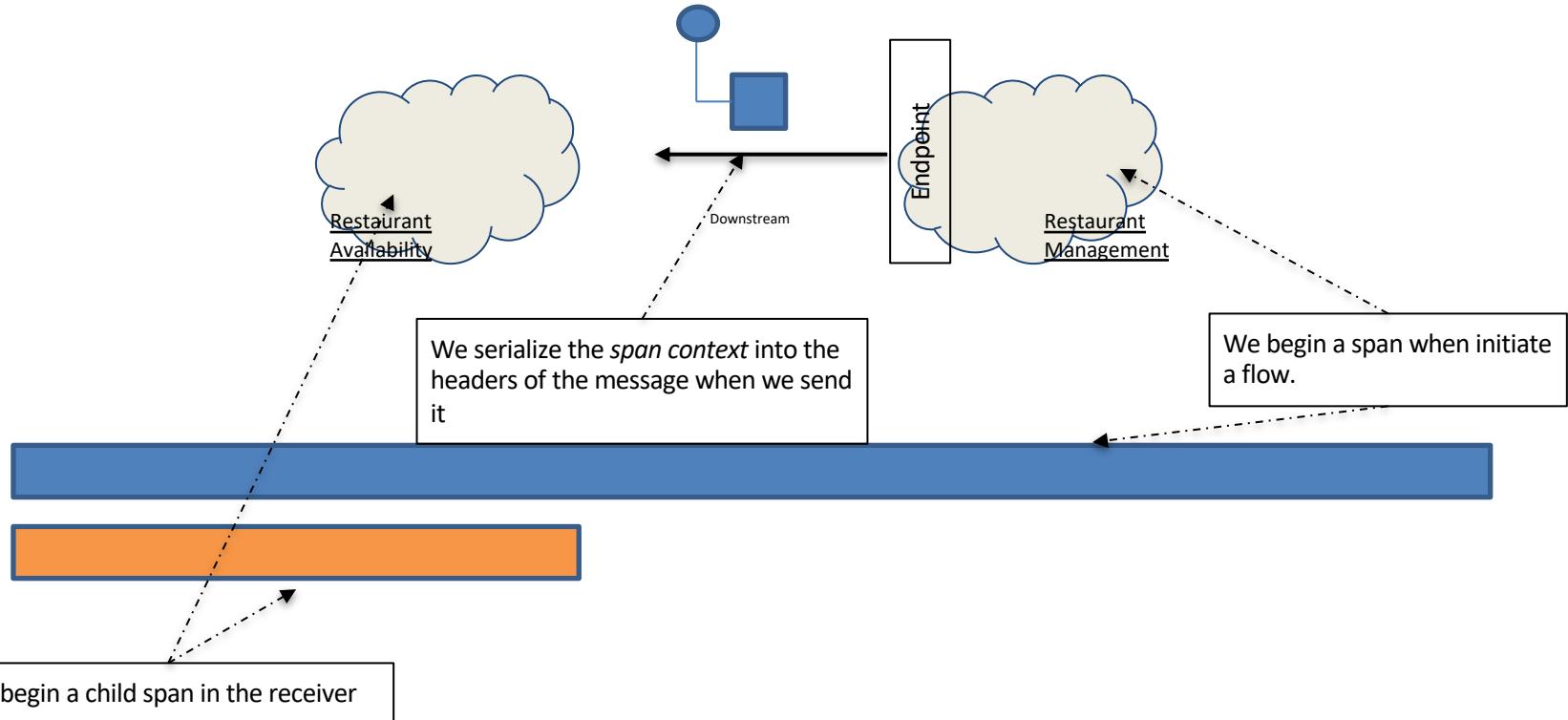


<https://github.com/boyney123/eventcatalog>

Observability



OpenTelemetry Tracing

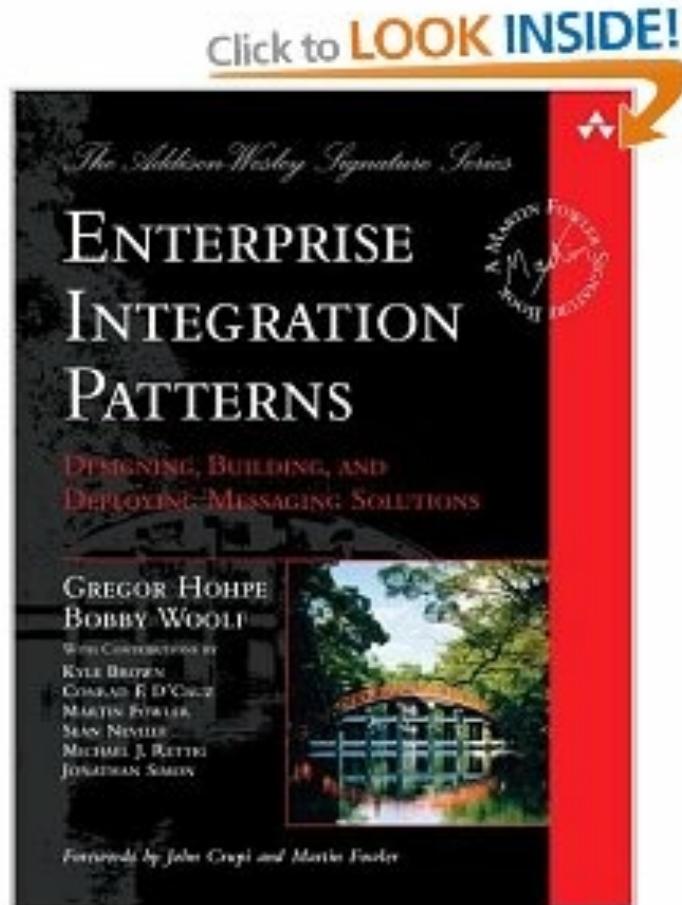


Spans

- Operation Name
- Start and Finish Timestamps
- Span Context (span_id, trace_id)
- Attributes (standard and defined)
- Events

NEXT STEPS

Further Reading



Further Reading

 Serverless Land Content ▾ Learn Code ▾ EDA ▾ Search Search

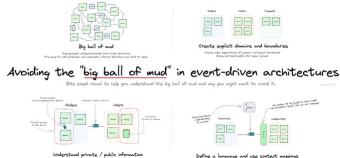
EDA VISUALS

Small bite sized visuals about event-driven architectures

Designs and thoughts from [@bogmey123](#)

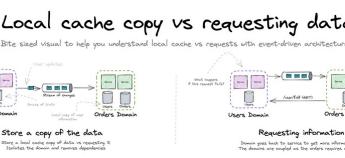
Avoiding the big ball of mud in event-driven architectures

Bite sized visual to help you understand the big ball of mud and why you might want to avoid it.



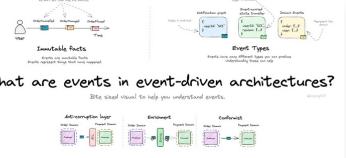
Local cache copy vs requesting data

Bite sized visual to help you understand local cache vs requests with event-driven architecture.



What are events in event-driven architectures?

Bite sized visual to help you understand events.

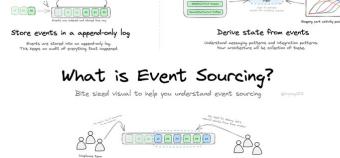


What are events?

Bite sized visual to help you understand events.

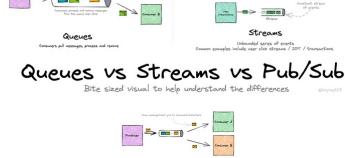
What is Event Sourcing?

Bite sized visual to help you understand event sourcing.



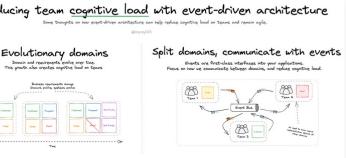
Queues vs Streams vs Pub/Sub

Bite sized visual to help understand the differences.



Reducing team cognitive load with event-driven architecture

Bite sized visual to help reduce cognitive load on teams and reduce risk.



Reducing team cognitive load with event-driven architectures

Bite sized visual to help reduce cognitive load.

How event-driven architecture can help reduce cognitive load.

<https://serverlessland.com/event-driven-architecture/visuals>

177

Q&A