# Practical Messaging

A 101 guide to messaging

Ian Cooper

X and Hachyderm: ICooper

# Who are you?

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.

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

# Day One Messaging

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

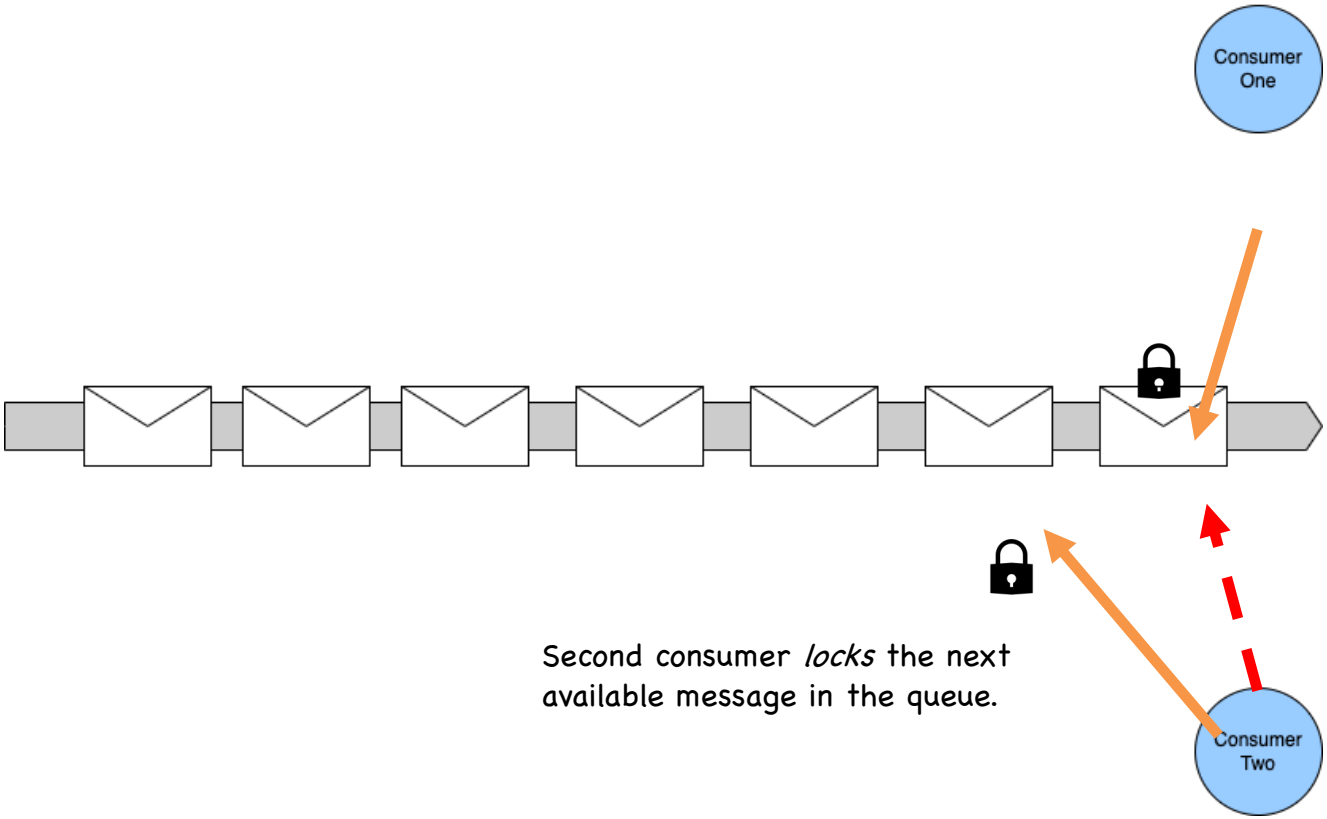
# Day Two Conversations

- 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

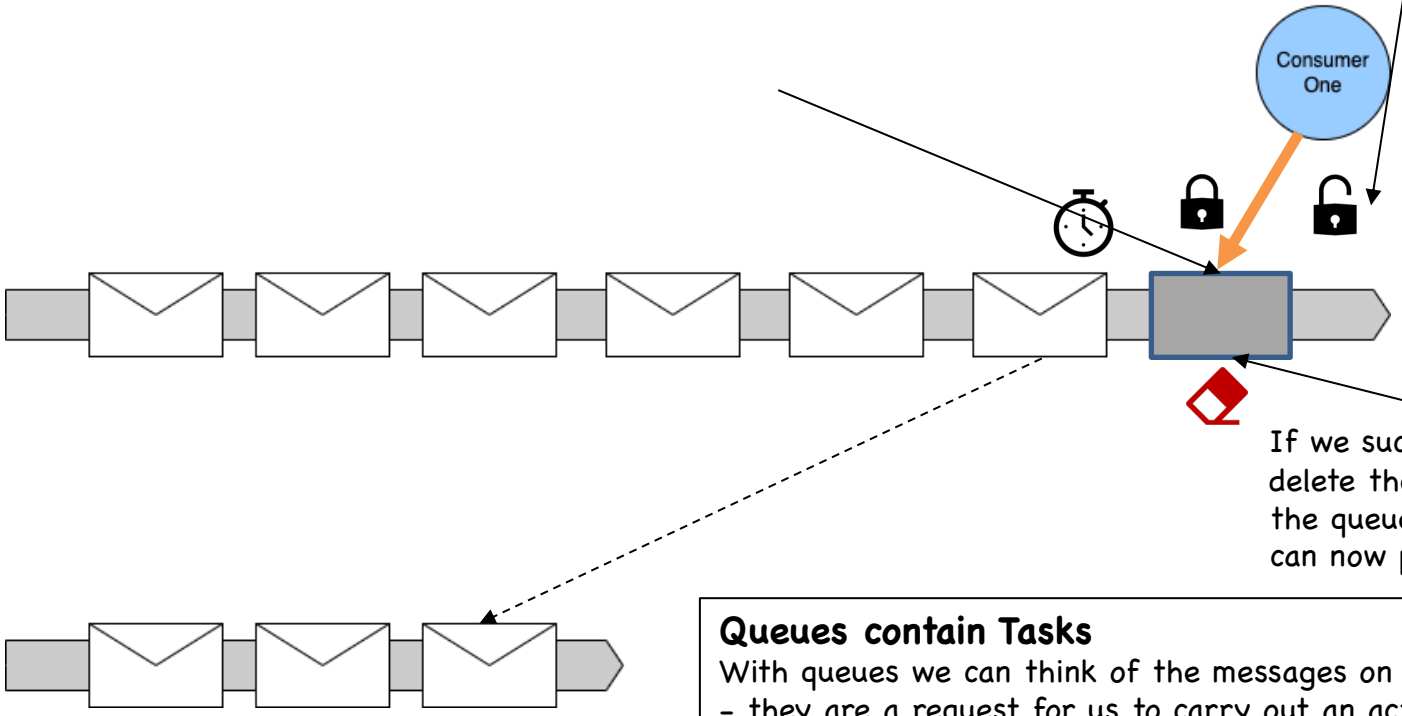

Consumer One
First consumer *locks* the next message in the queue whilst it processes it.
Second consumer *locks* the next available message in the queue.
Consumer Two
Second consumer *reads past* any locked message in the queue.

# 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.

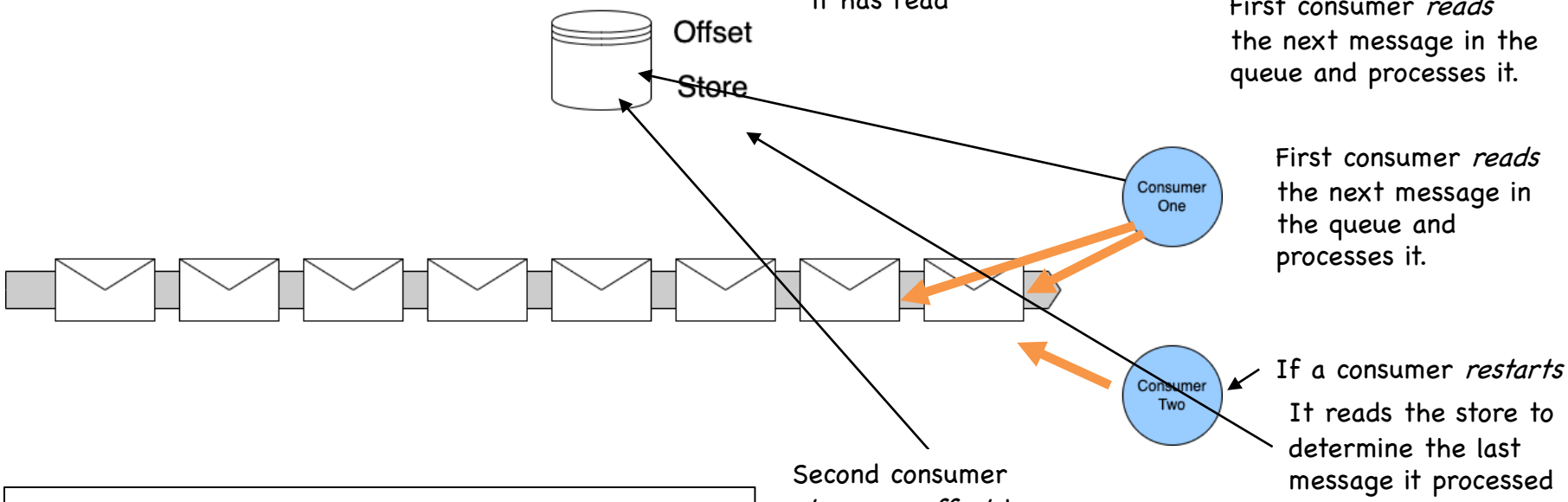
After a certain number of re-queues we may move the message to a dead-letter channel, it turns out that no one action the request within in a reasonable time frame

**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



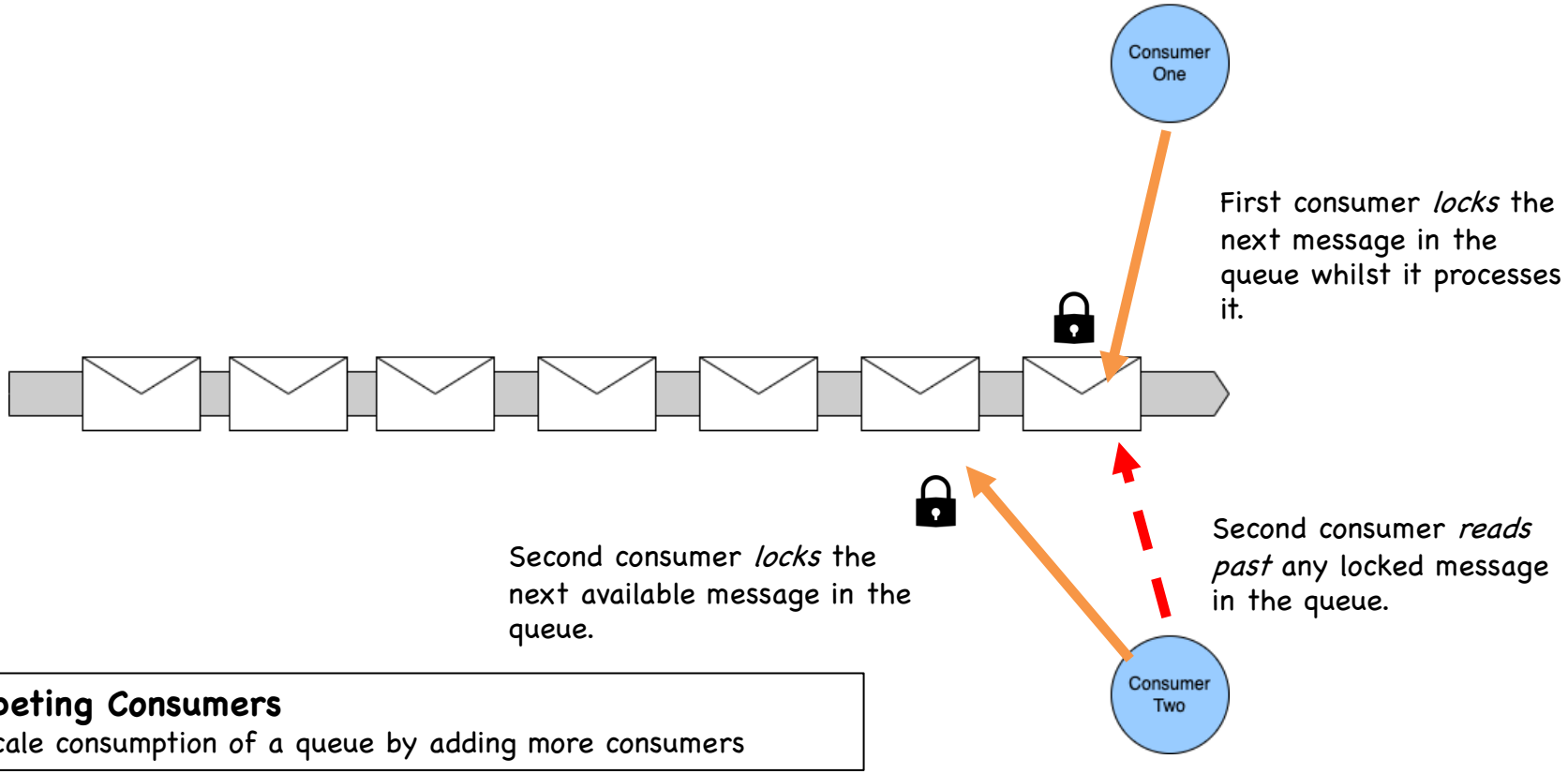

**Streams contain Facts**

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

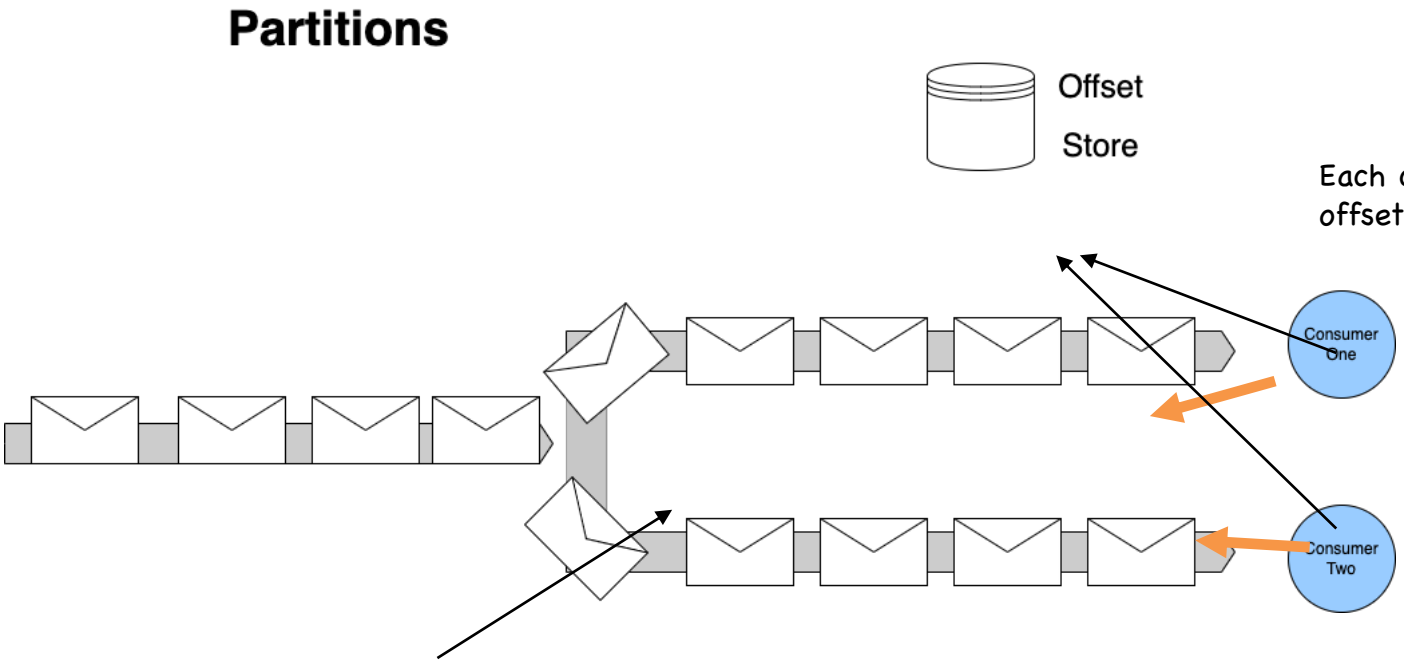
# Scaling Queues and Streams

# Queues




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

# Streams



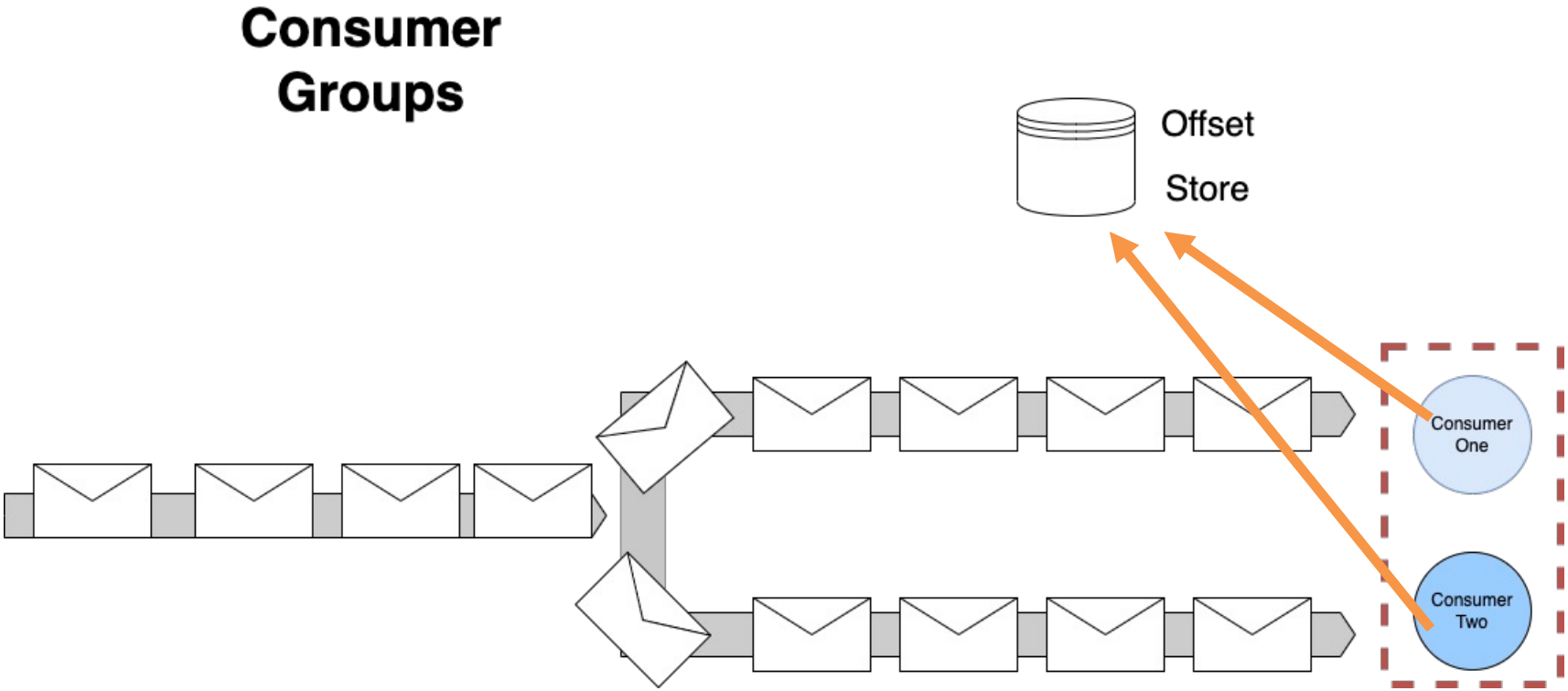

Each consumer manages offsets for their partition.

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.

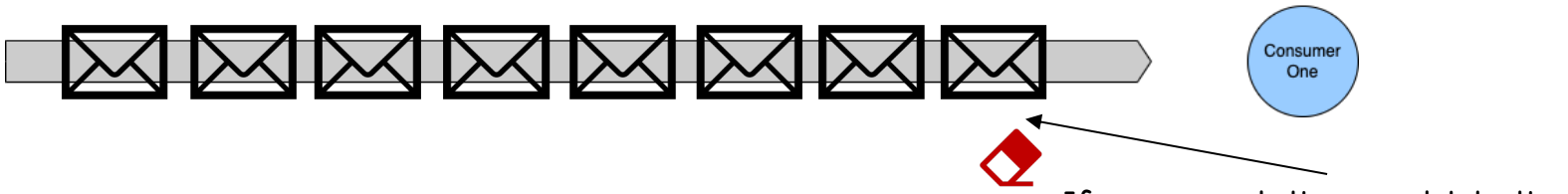
# Streams

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

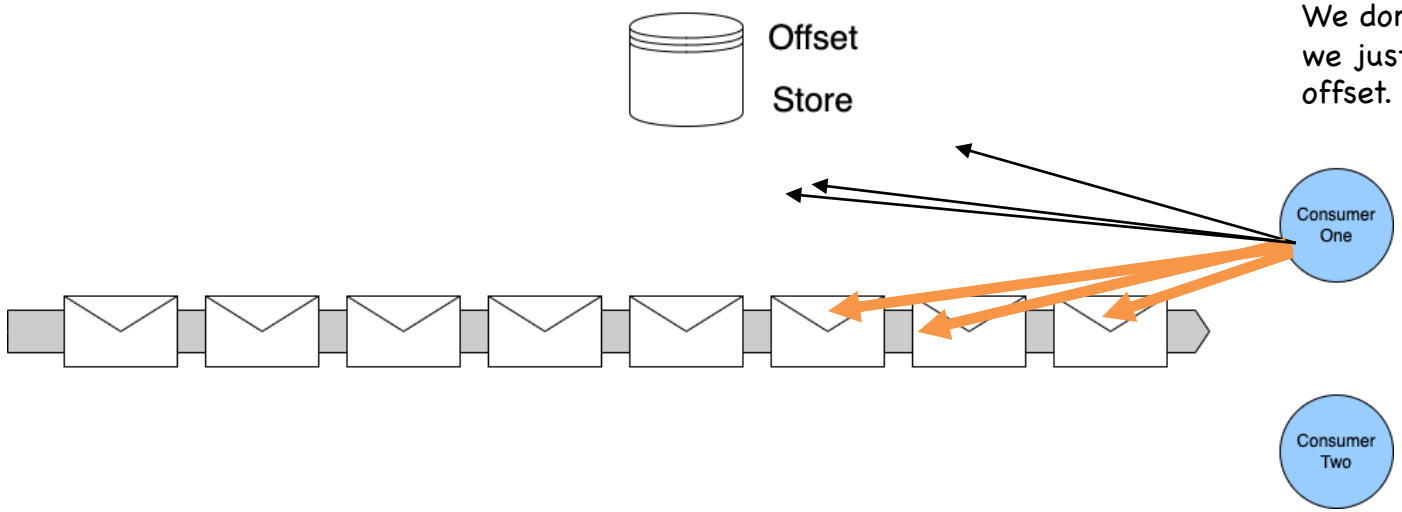



If we succeed, then we delete the message from the queue, no one else can now process it.

**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

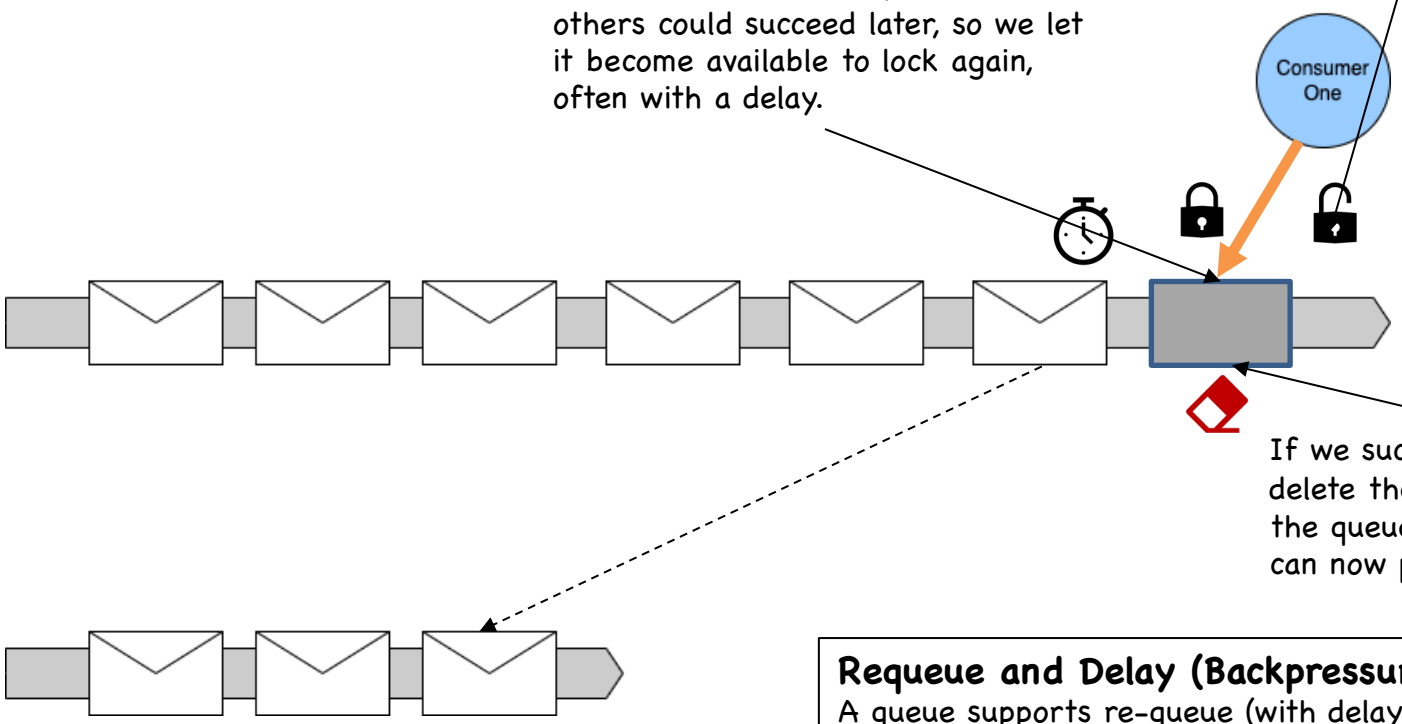
# Requeue and Delay (Backpressure)

# 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.



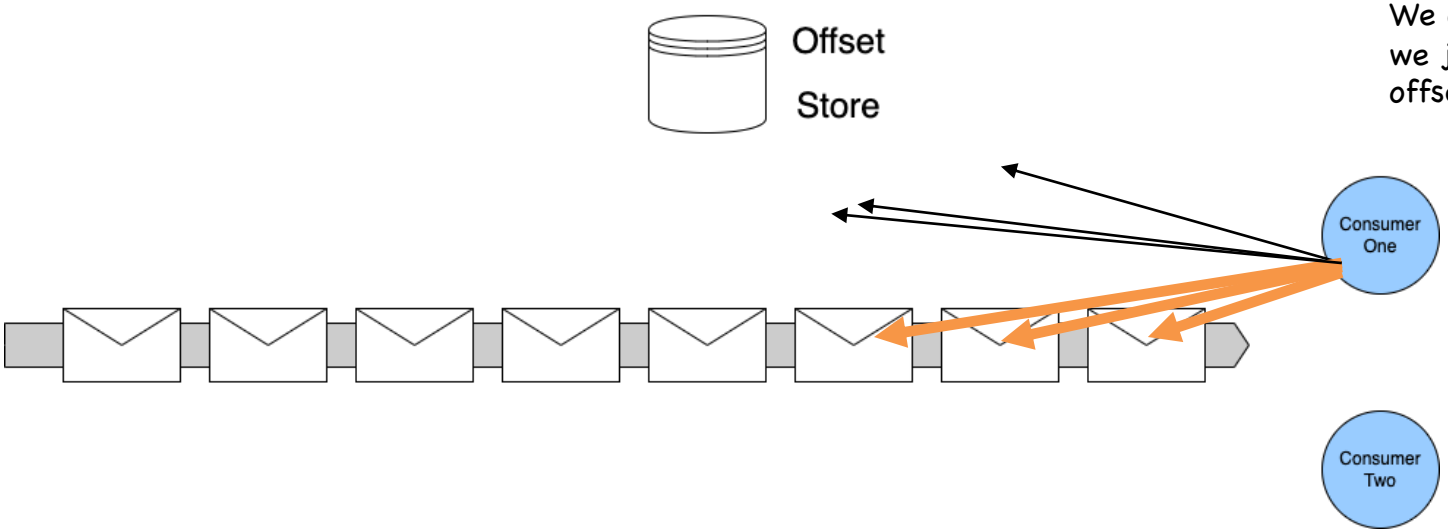

If we succeed, then we delete the message from the queue, no one else can now process it.

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 | ✓ | ✓ | ✗ |

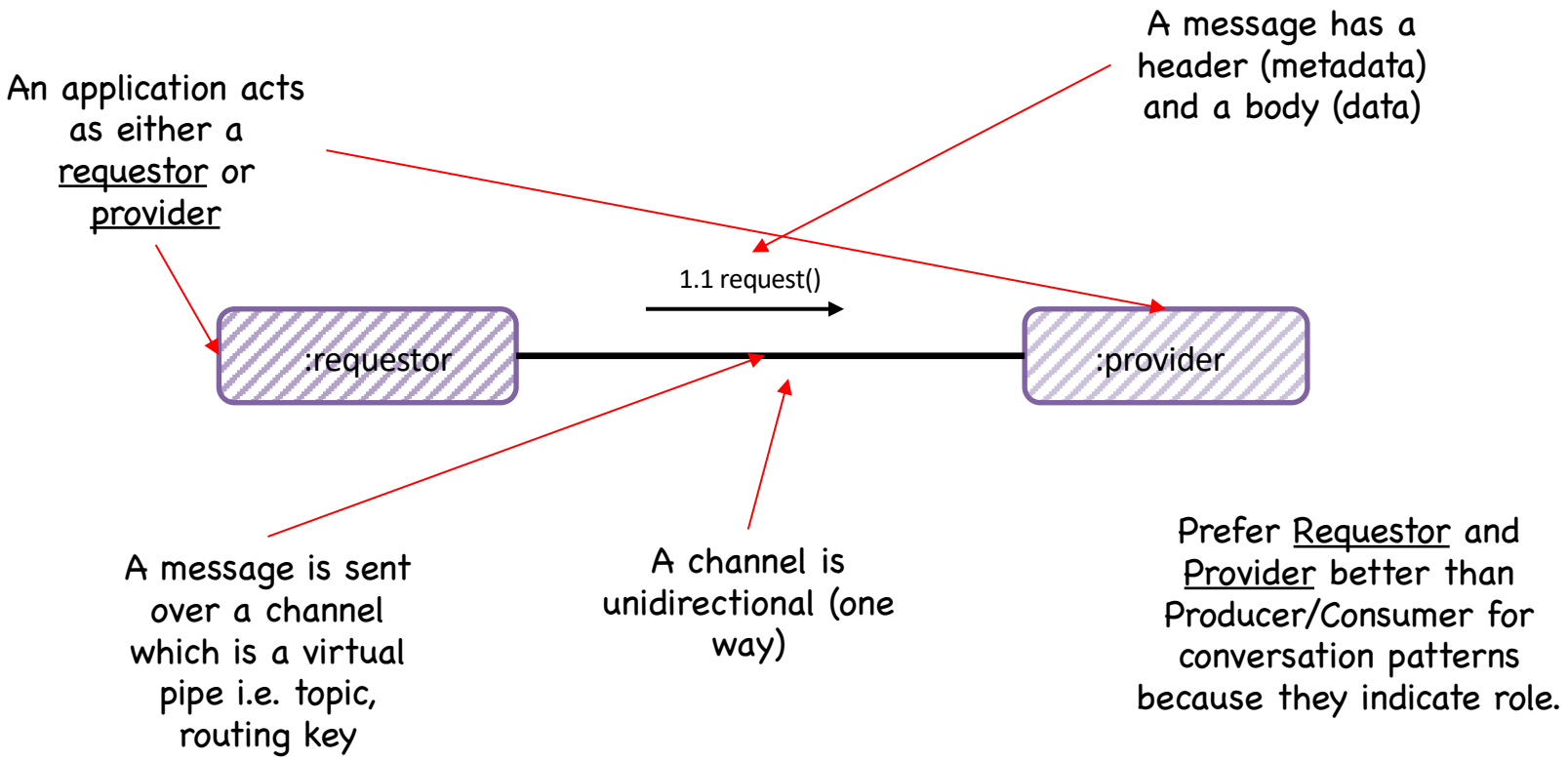
# EXERCISE MATERIAL

Introduction to Kafka

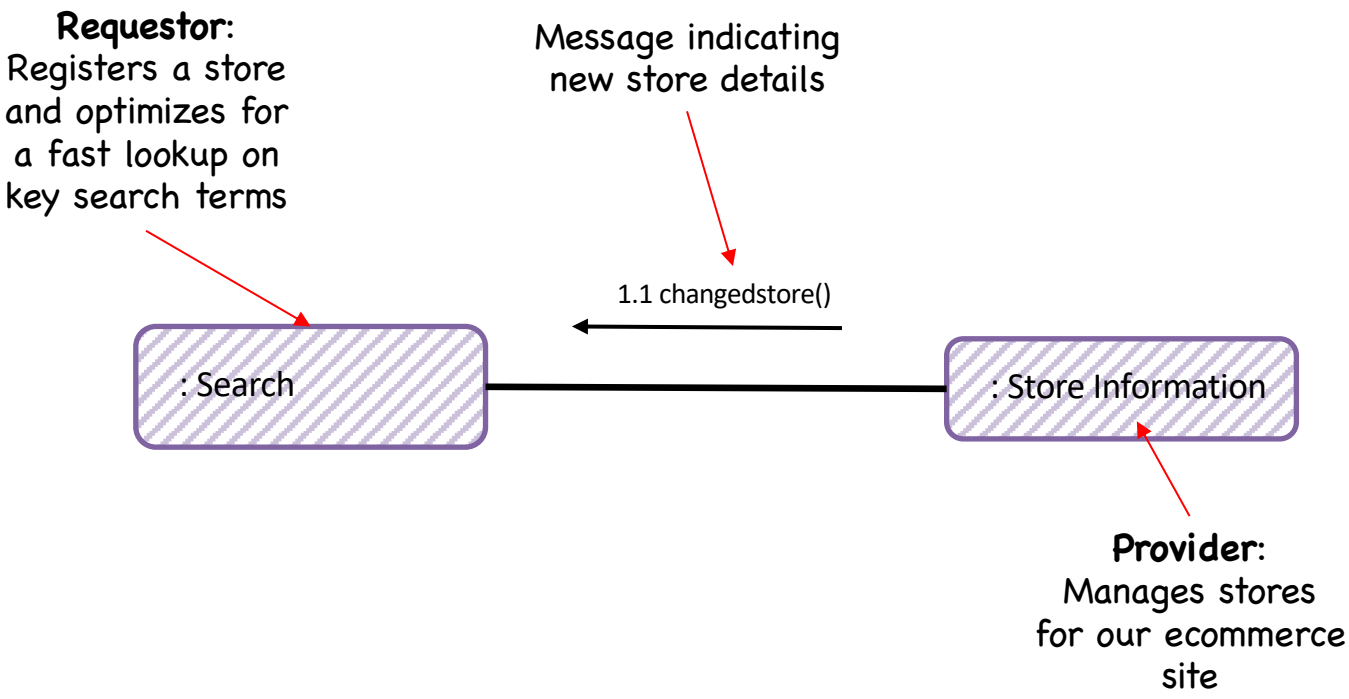
- Readme
- Slides

# CONVERSATION PATTERNS

# Messaging Participants



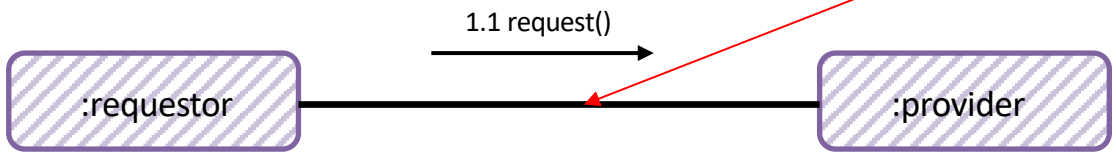
# Messaging Participants



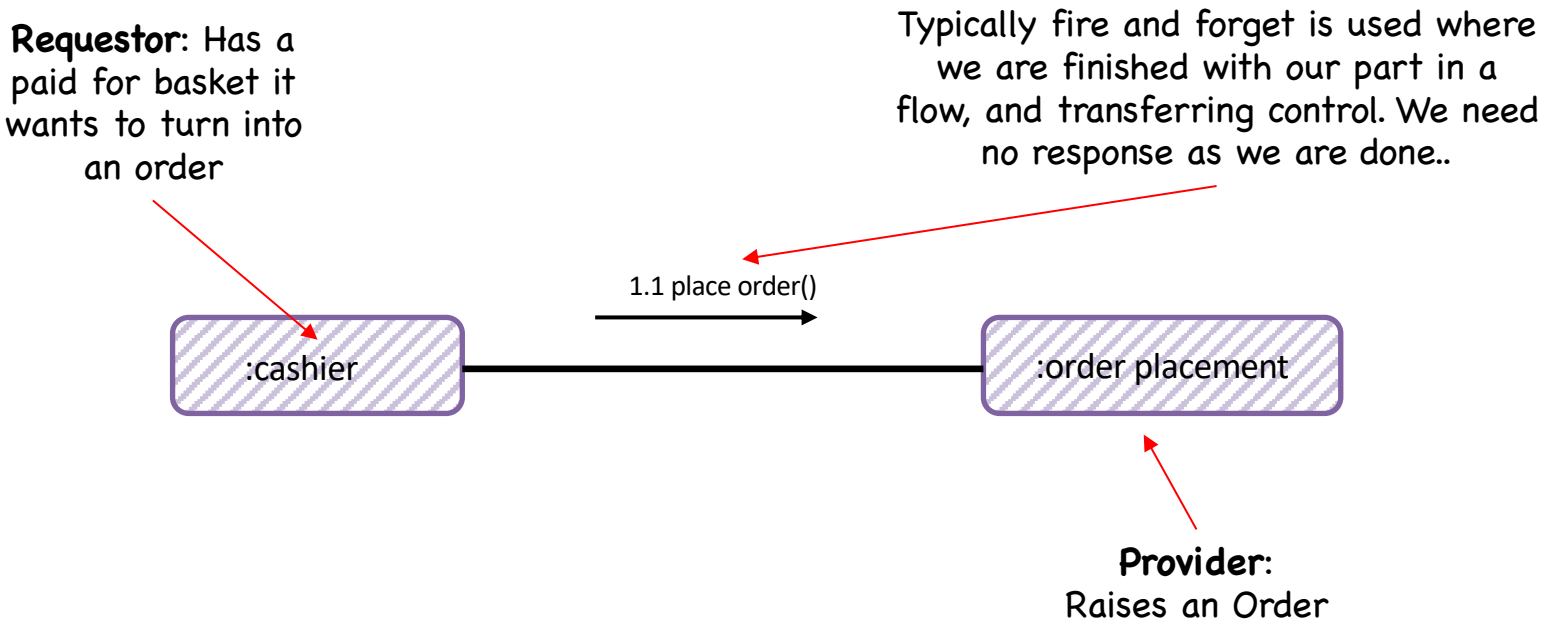
# In-Only (Fire and Forget)

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



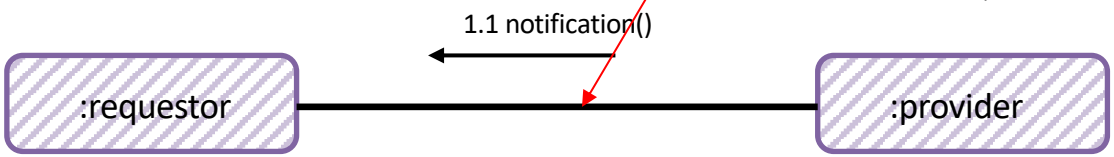
# In-Only (Fire and Forget)



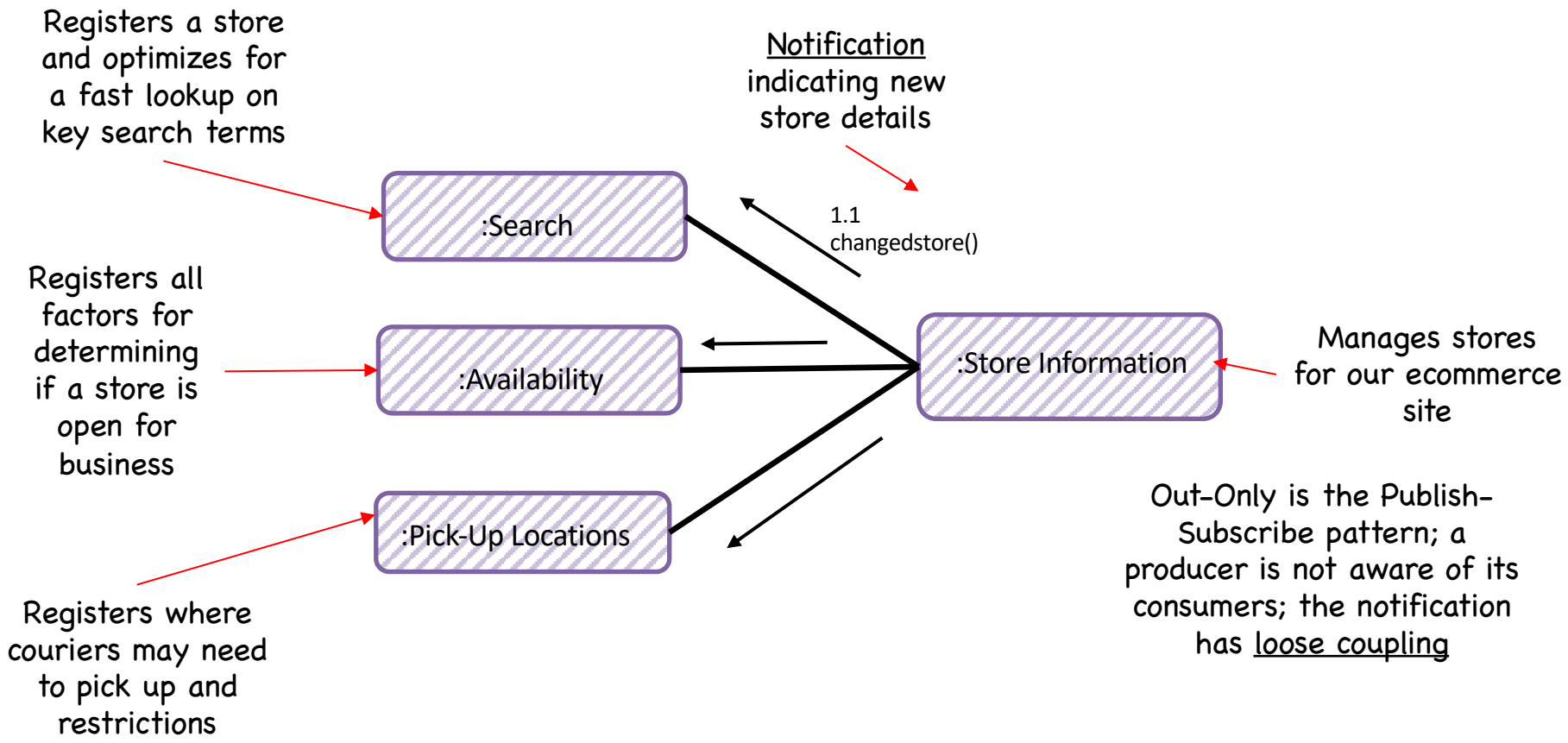
# Out-Only (Notification)

Typically we call this a **notification**.

Under an Out-Only pattern the requestor subscribes to the provider, and an operation is triggered by receipt of a message but it does not acknowledge the message to the provider.

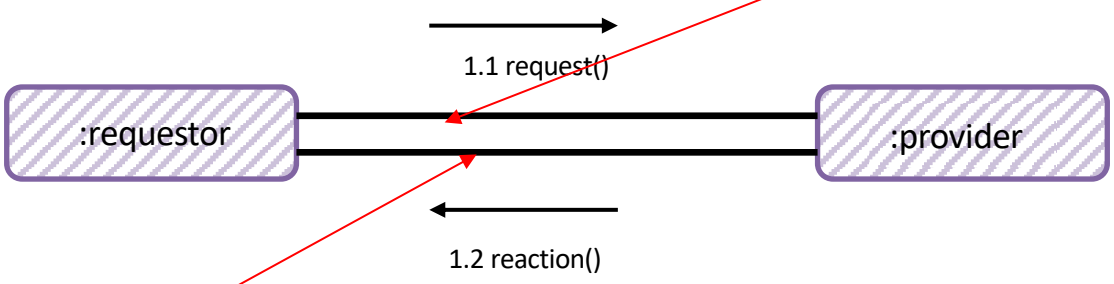


# Out-Only (Publish-Subscribe)



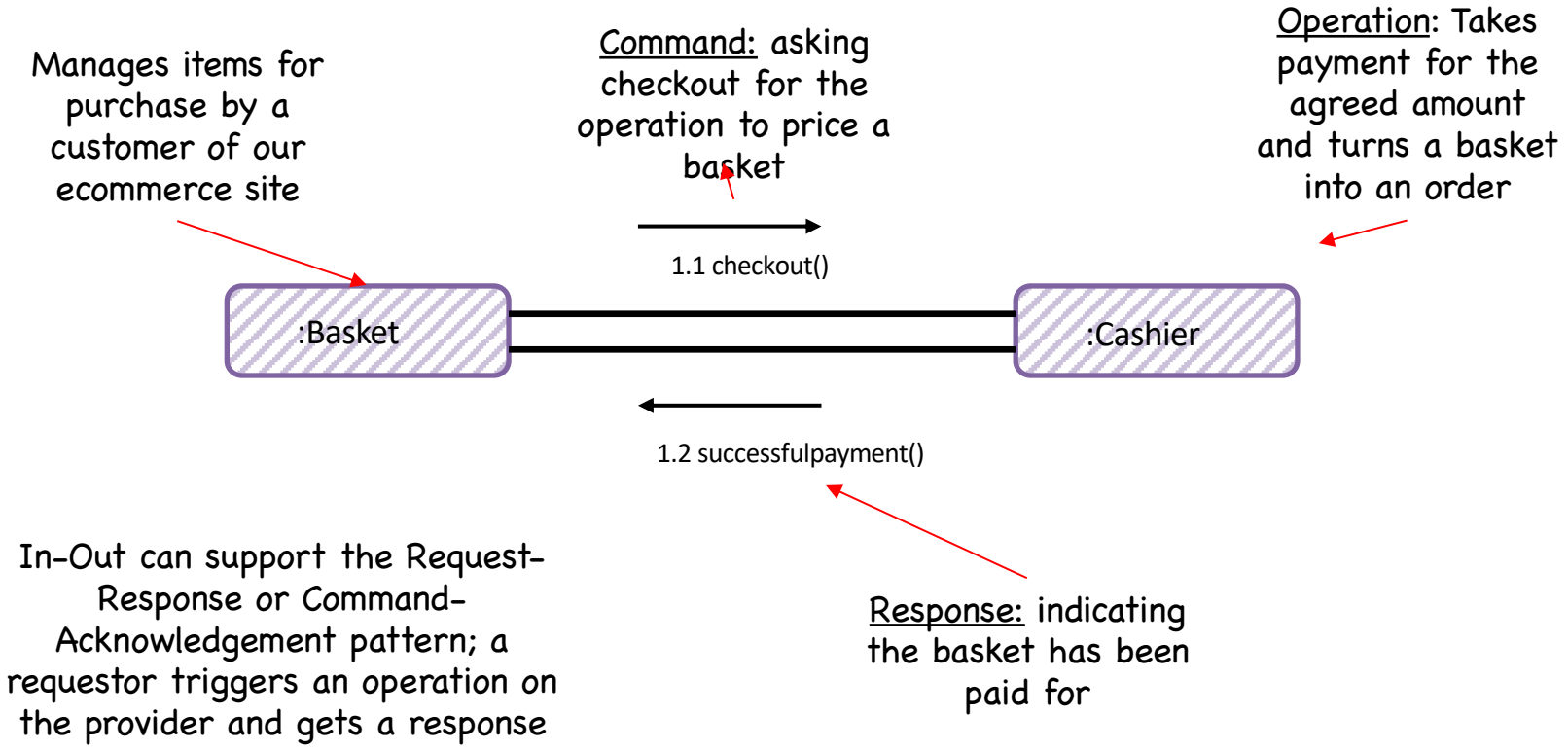
# In-Out (Request-Reaction)

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.

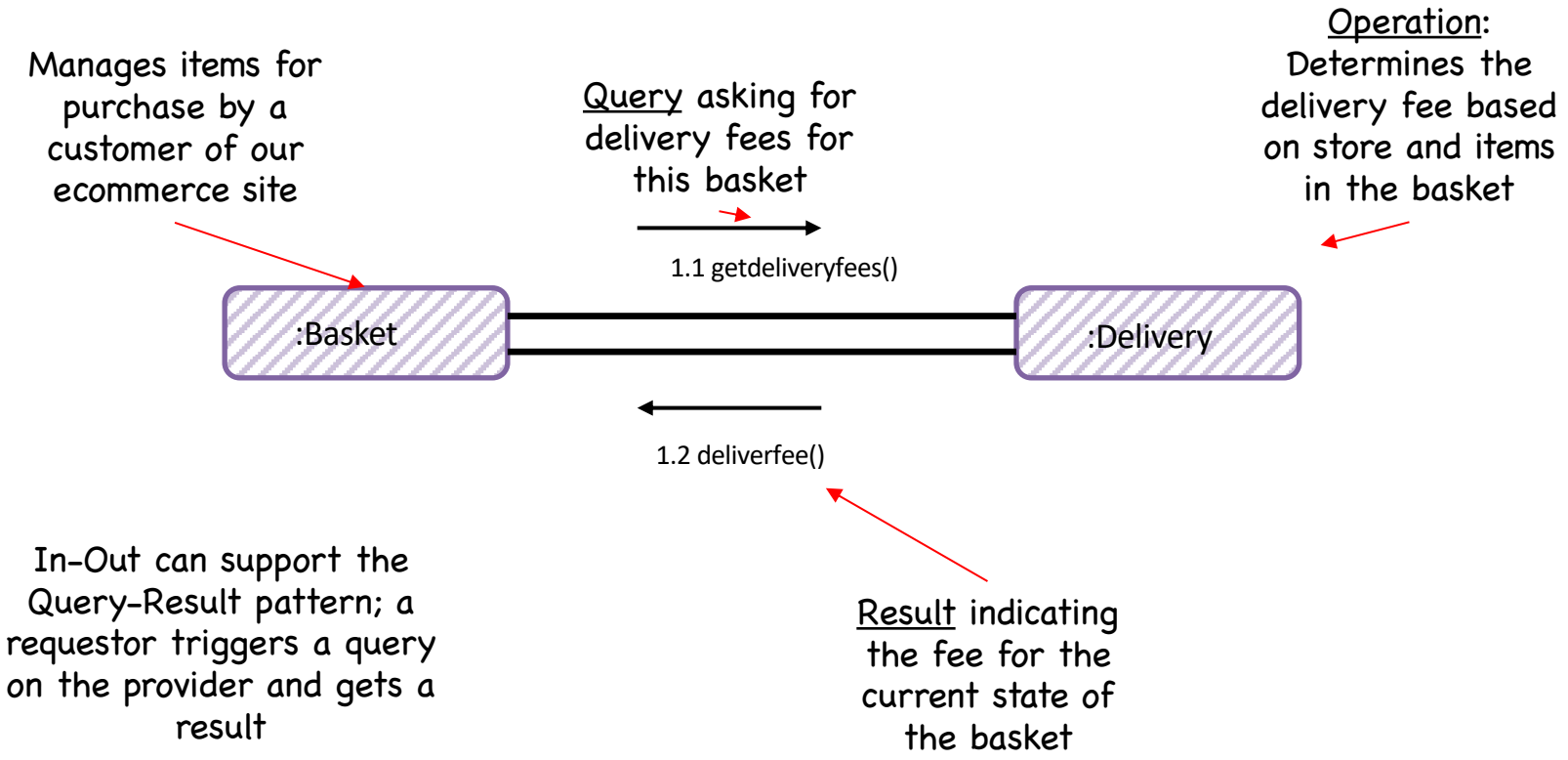



When using an In-Out pattern the provider sets a **correlation id** (conversation id) in the header of the request and the provider returns that id in the header of the response so that we can correlate replies sent over a separate channel.

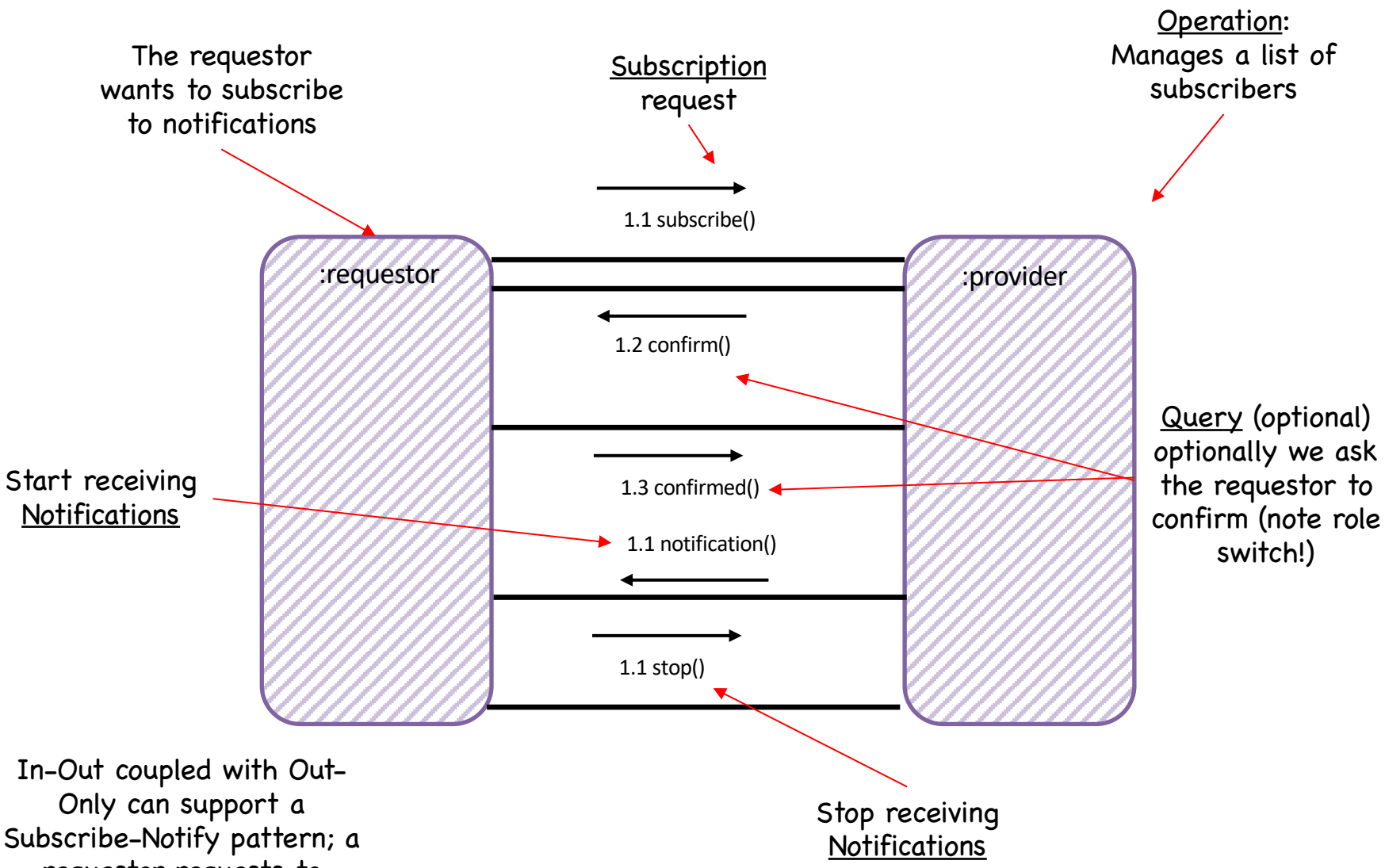
# Request-Reaction (Command-Acknowledgement)



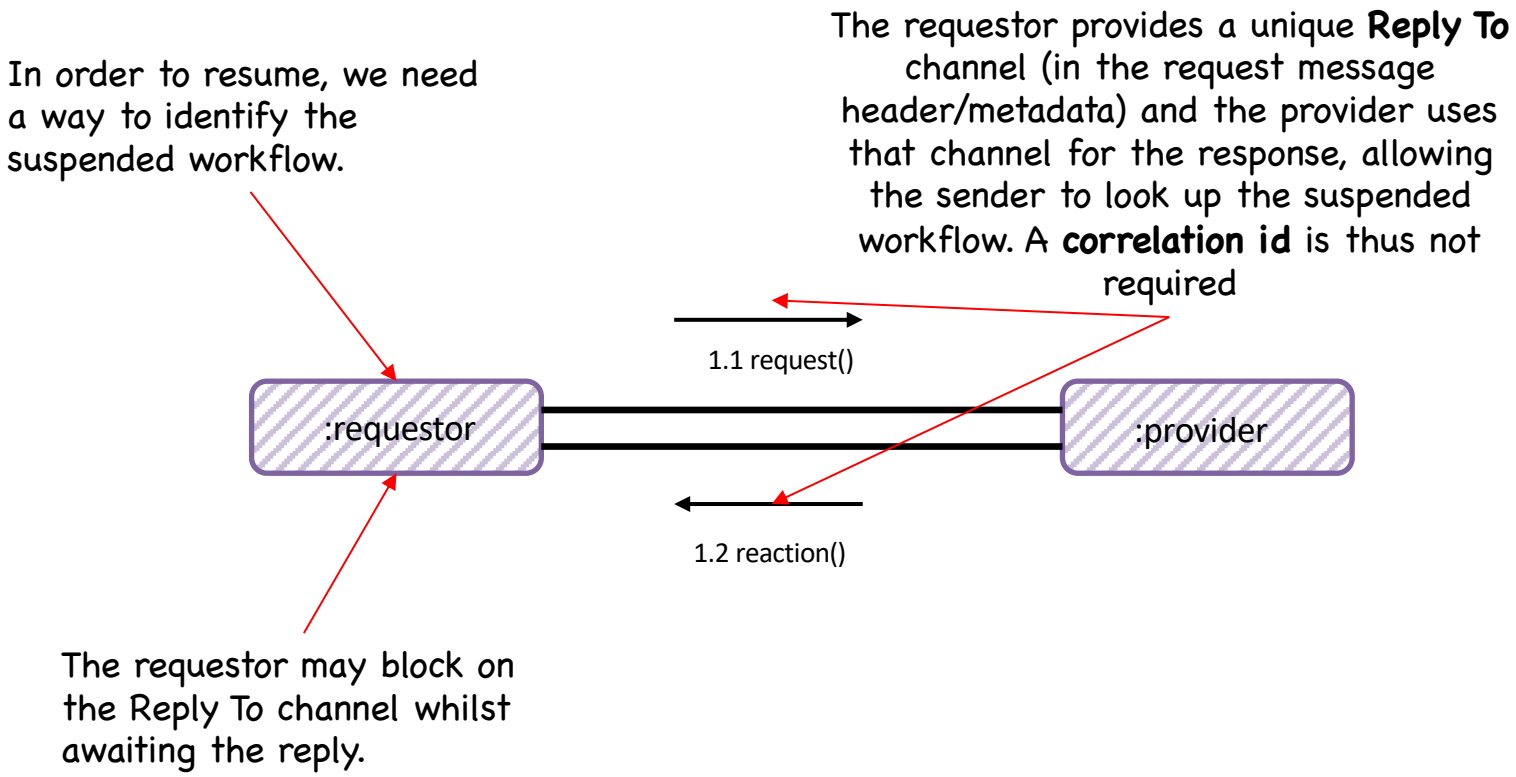
# In-Out (Query-Result)



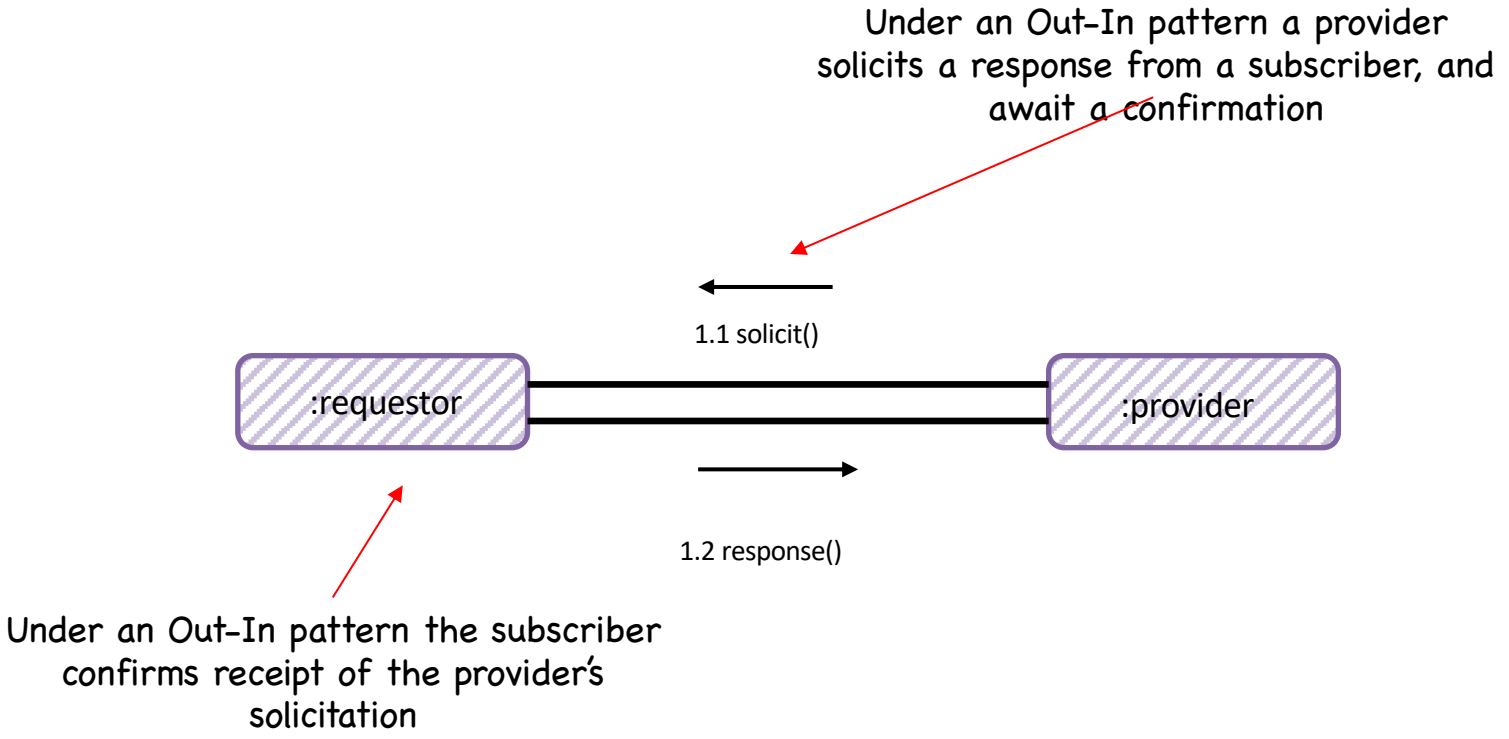
# In-Out (Subscribe-Notify)



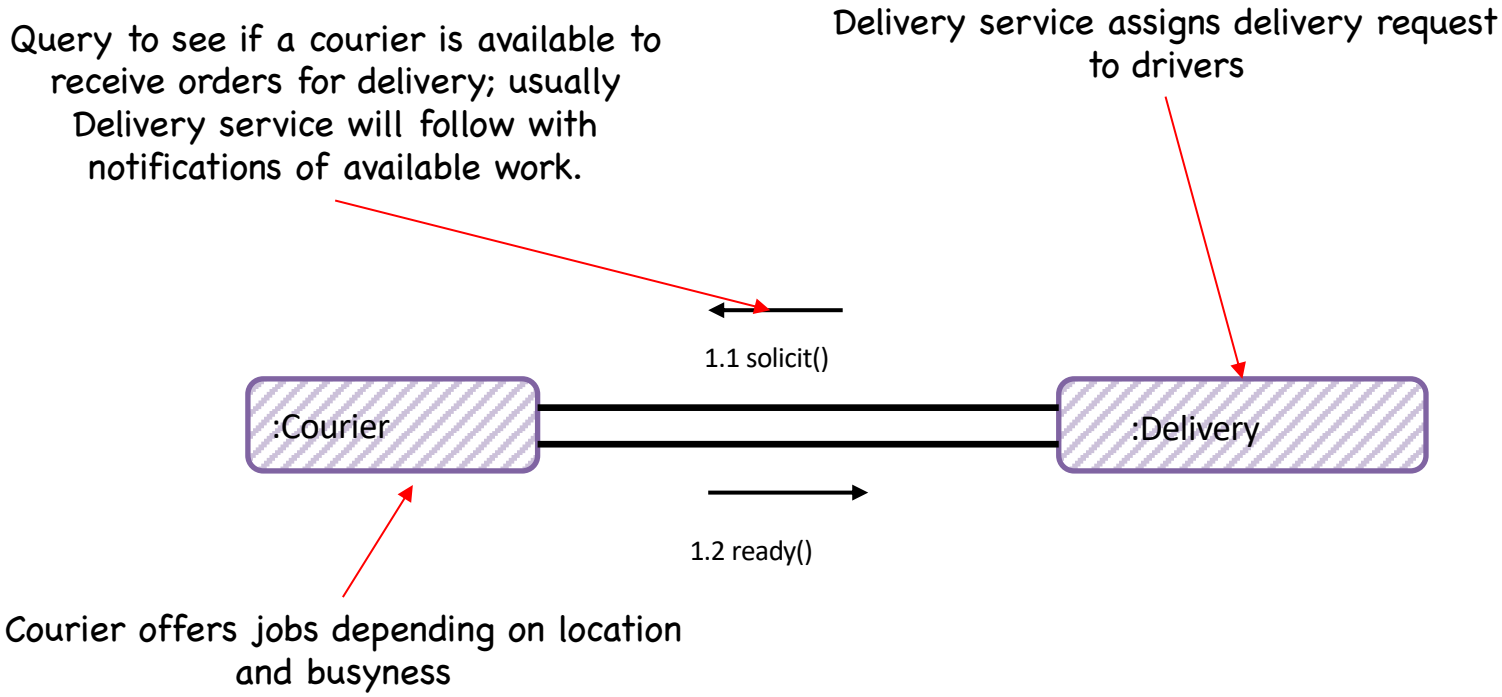
# Blocking In-Out (Request-Reply)



# Out-In (Solicit-Response)



# Out-In (Solicit-Response)



# Messaging

Has Intent

Request An Answer (Query)

Transfer of Control

(Command)

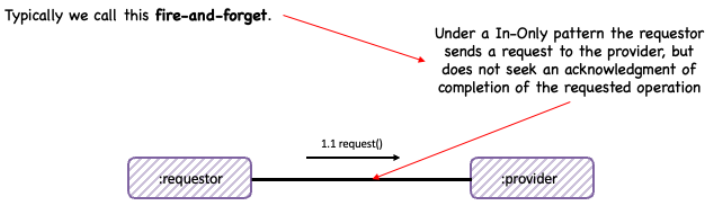
Transfer of Value

Part of a Workflow

Part of a Conversation

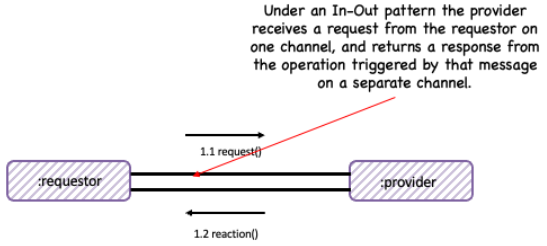
Concerned with the Future

In-Only (Fire and Forget)




Ian Cooper 25

In-Out (Request-Reaction)

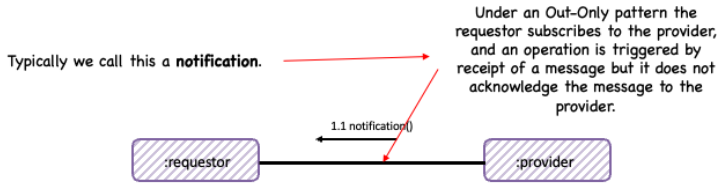



Ian Cooper 29

# Eventing

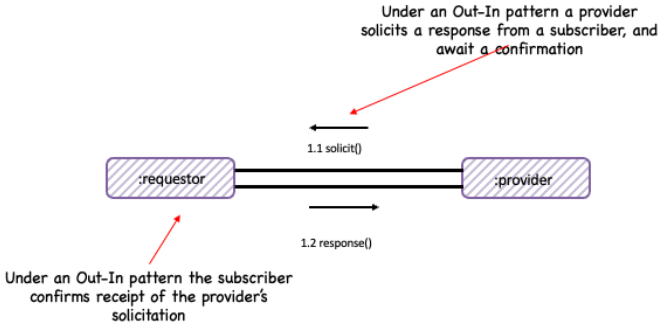
- Provides Facts
- Things you Report On
- No Expectations
- History
- Context
- Concerned with the Past

Out-Only (Notification)




Ian Cooper 27

Out-In (Solicit-Response)

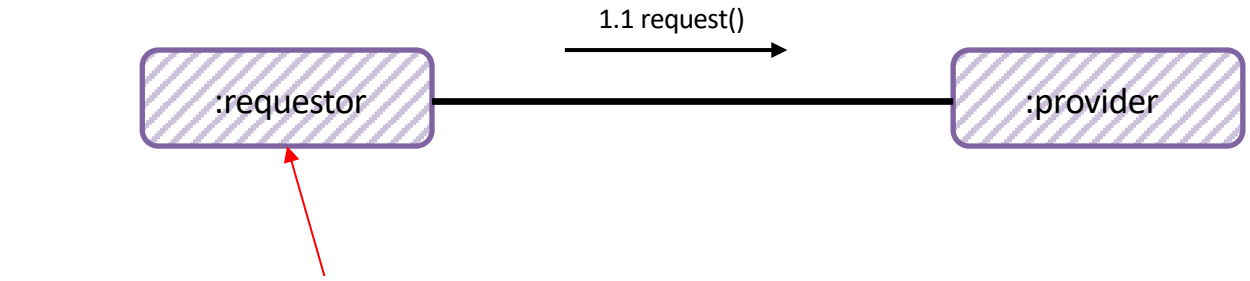



Ian Cooper 34

How do we manage workflows in requestors and providers

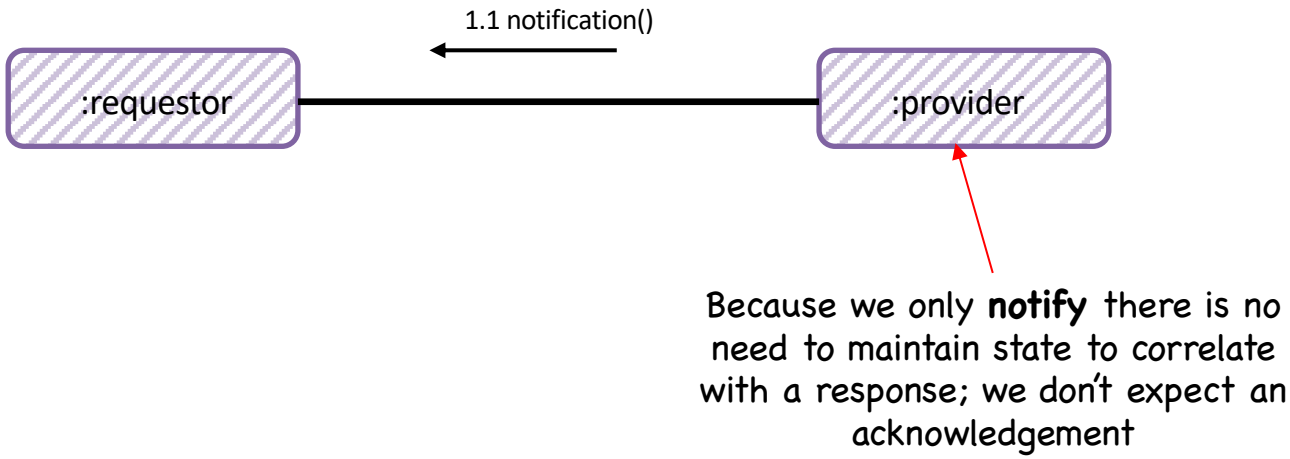
# ACTIVITY AND CORRELATION

# In-Only (Fire and Forget)

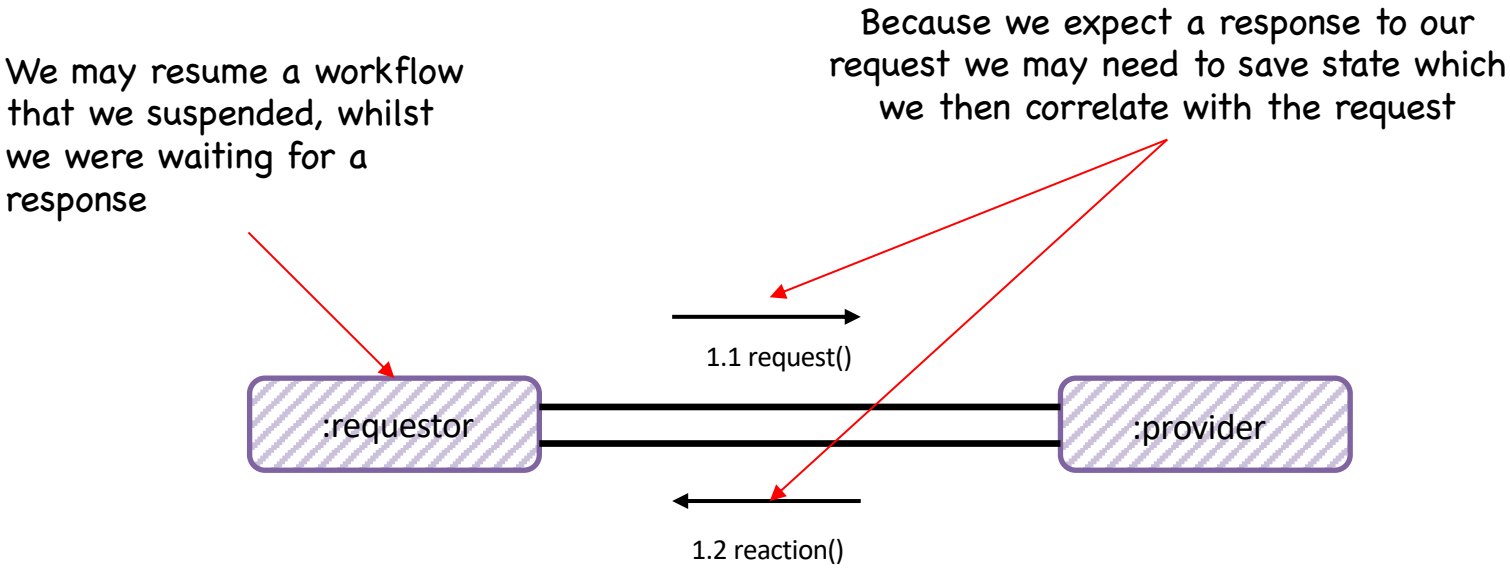



Because we are **fire-and-forget** there is no need to maintain state to correlate with a response; we don't expect a response

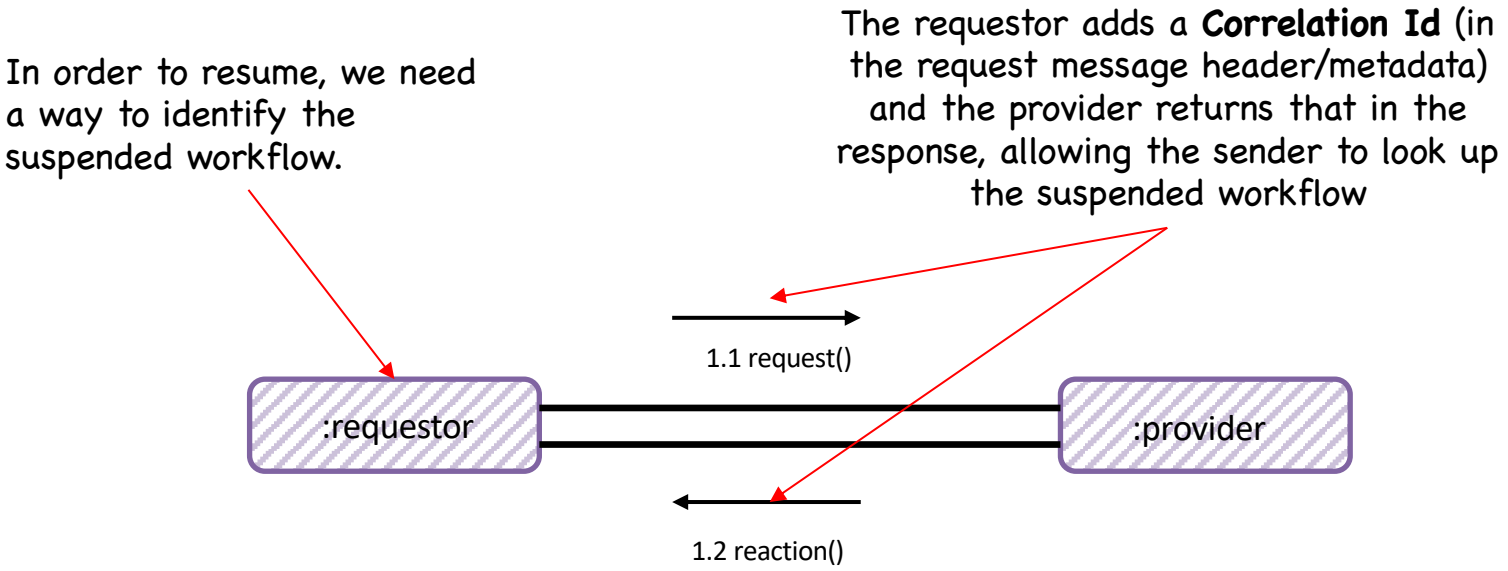
# Out-Only (Notification)



# In-Out (Request-Reaction)



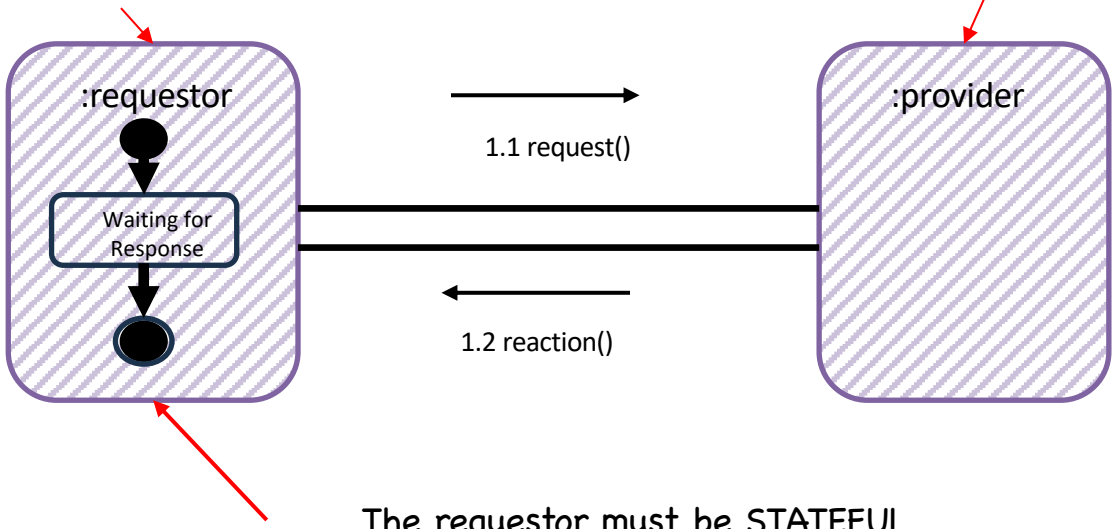
# In-Out (Request-Reaction) Correlation Id



# 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



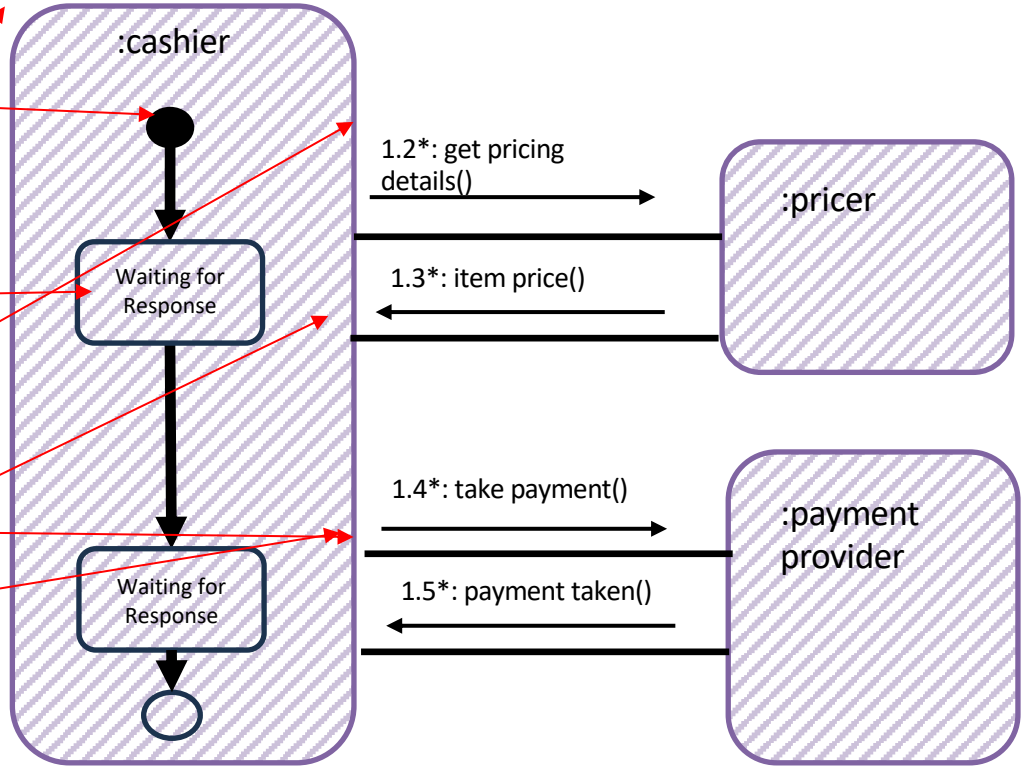

The requestor must be STATEFUL

# 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 is has sent and received, so it can be reconstructed.

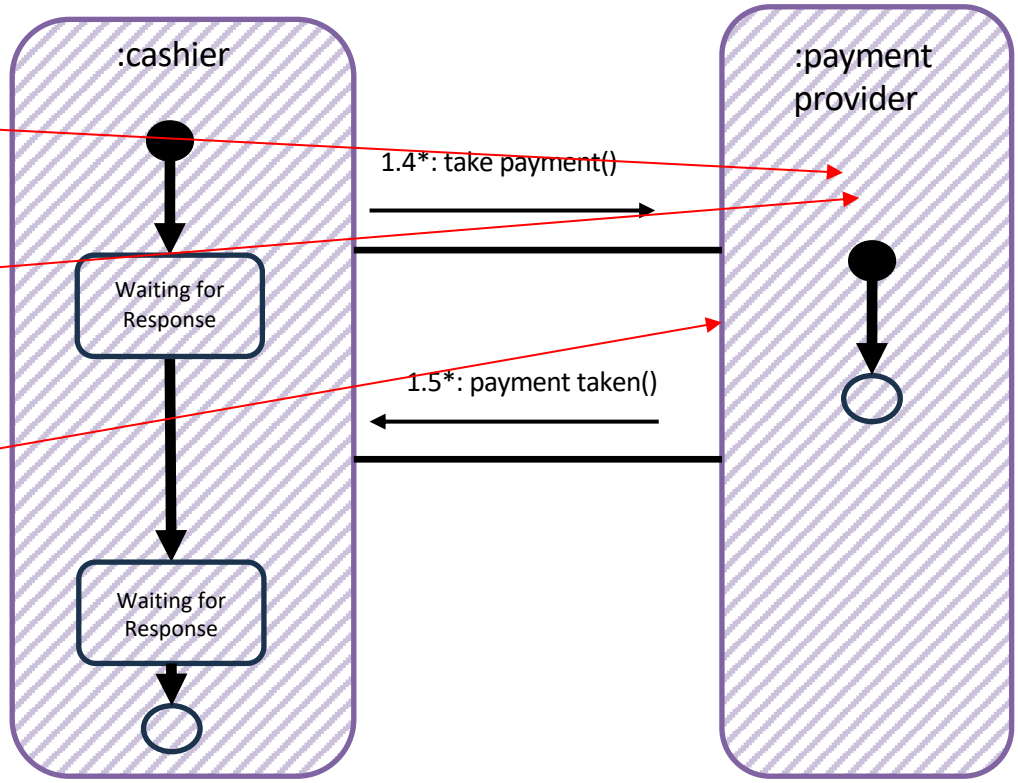


# In-Out (Request-Reaction) Activity

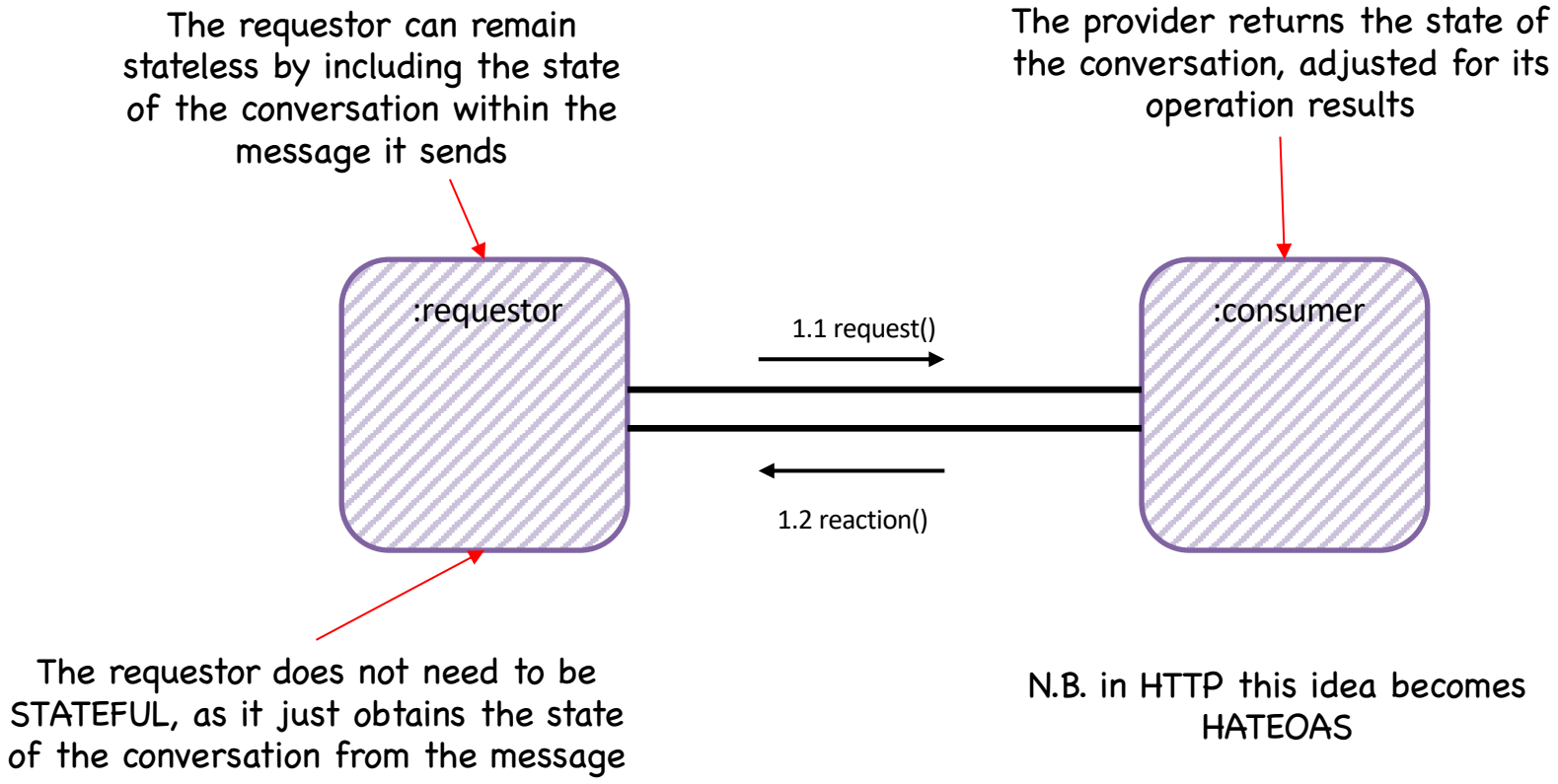
An **activity** is the code that runs in response to sending a message.

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 is has sent and received, so the conversation can be reconstructed.



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



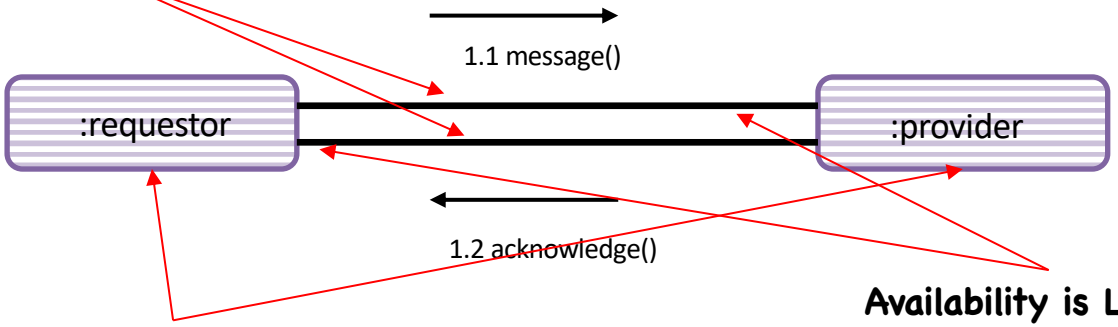
What happens when it all goes wrong

# REPAIR AND CLARIFICATION

# Failure

**Failure Scenarios**

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



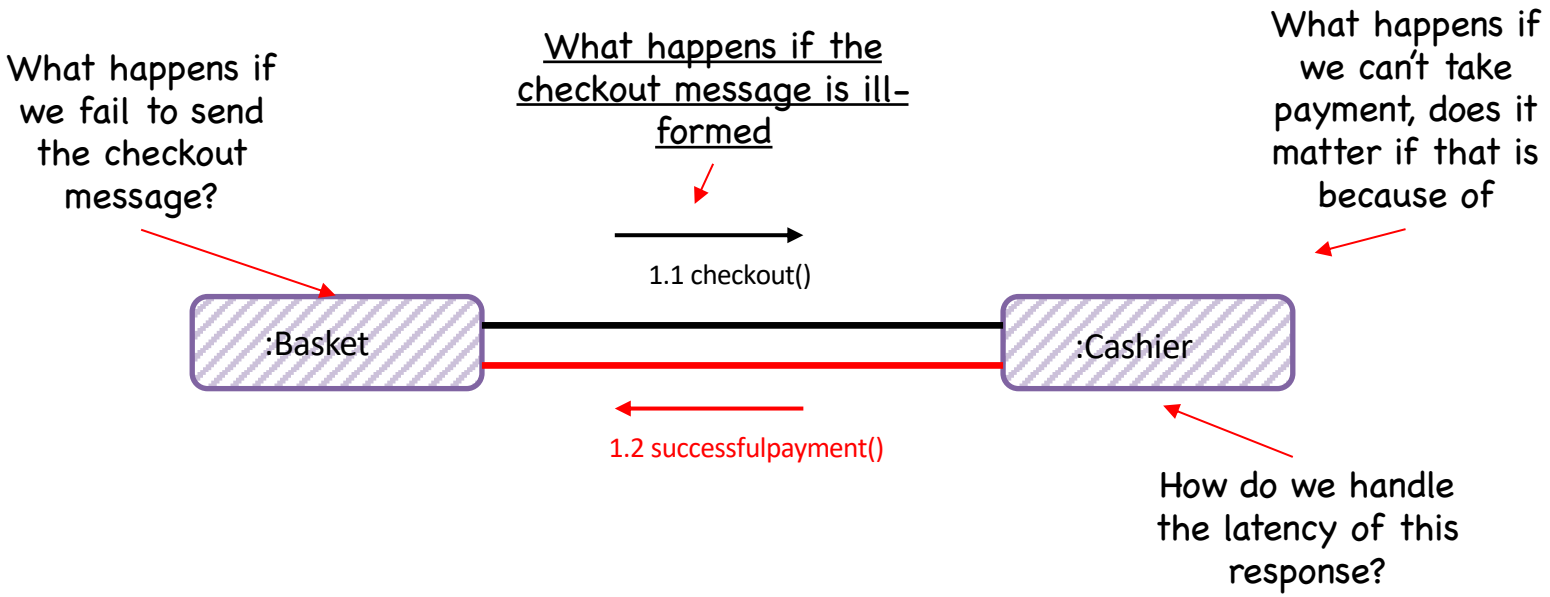

**Consistency**

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

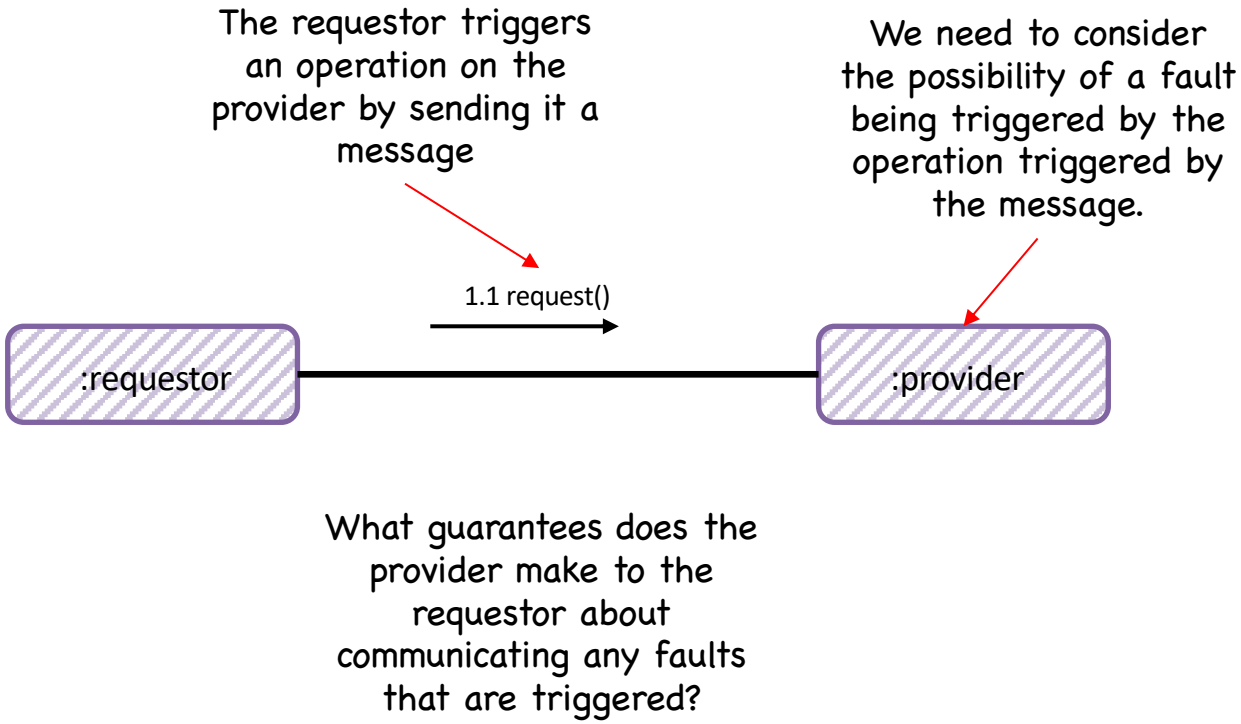
**Availability is Latency**

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

# Failure

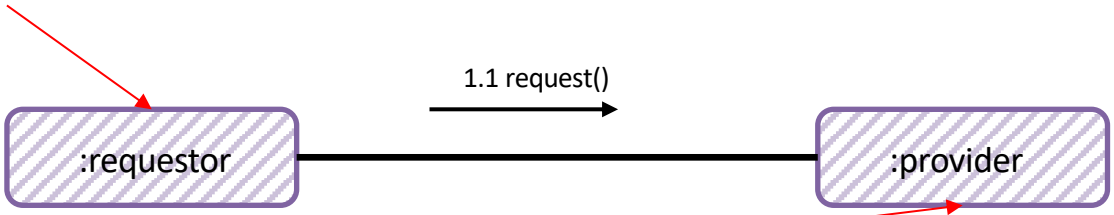

What happens if we fail to send the checkout message?
What happens if the checkout message is ill-formed
What happens if we can't take payment, does it matter if that is because of
1.1 checkout()
:Basket
:Cashier
1.2 successfulpayment()
How do we handle the latency of this response?

# Repair and Clarification



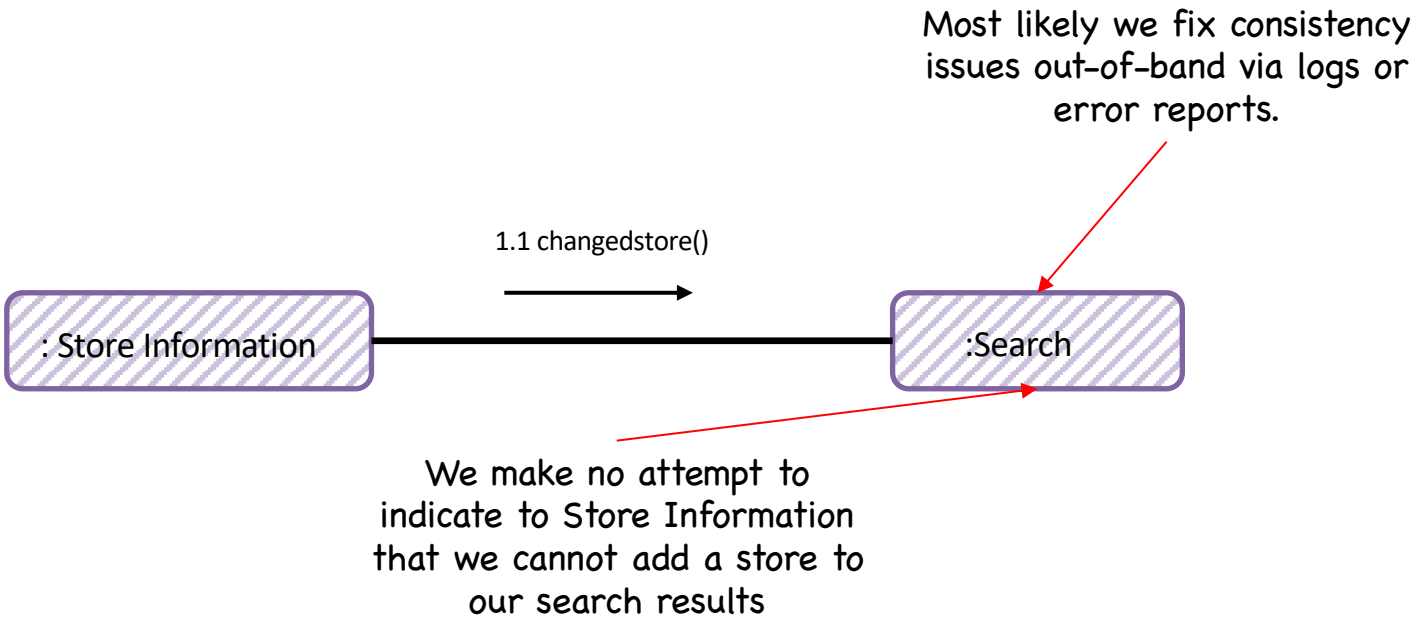
# No Fault

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

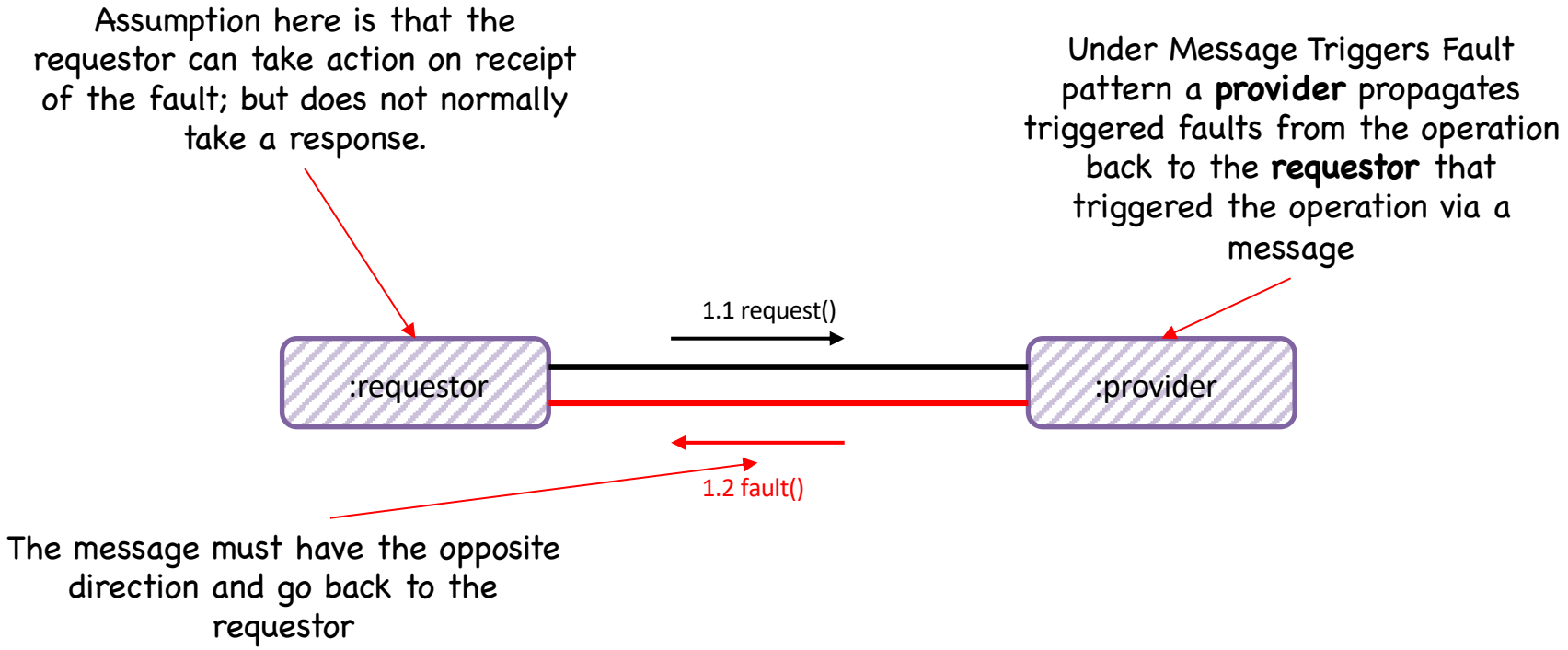



Under a No Fault pattern the provider makes no attempt to communicate triggered faults from the operation to the requestor

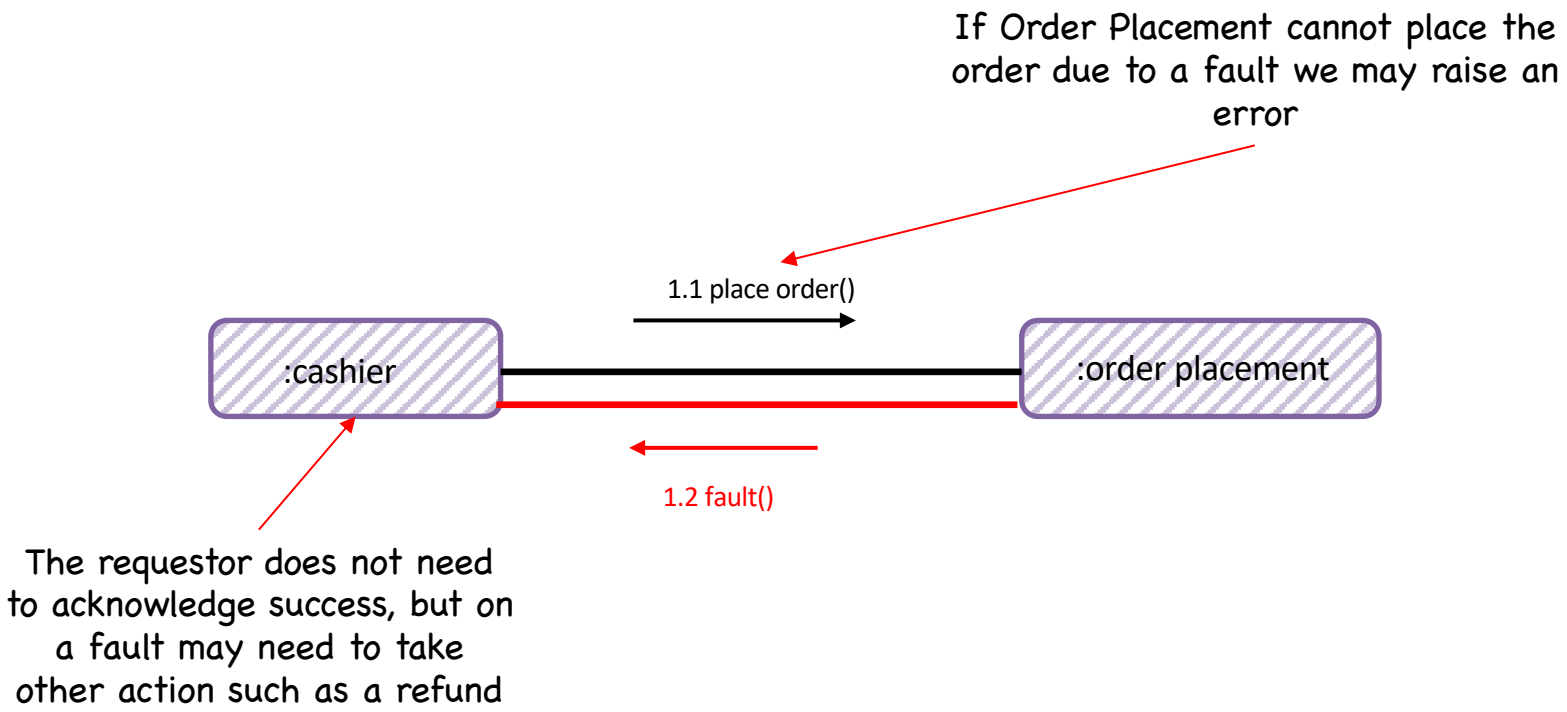
# No Fault


Most likely we fix consistency issues out-of-band via logs or error reports.
1.1 changedstore()
: Store Information
:Search
We make no attempt to indicate to Store Information that we cannot add a store to our search results

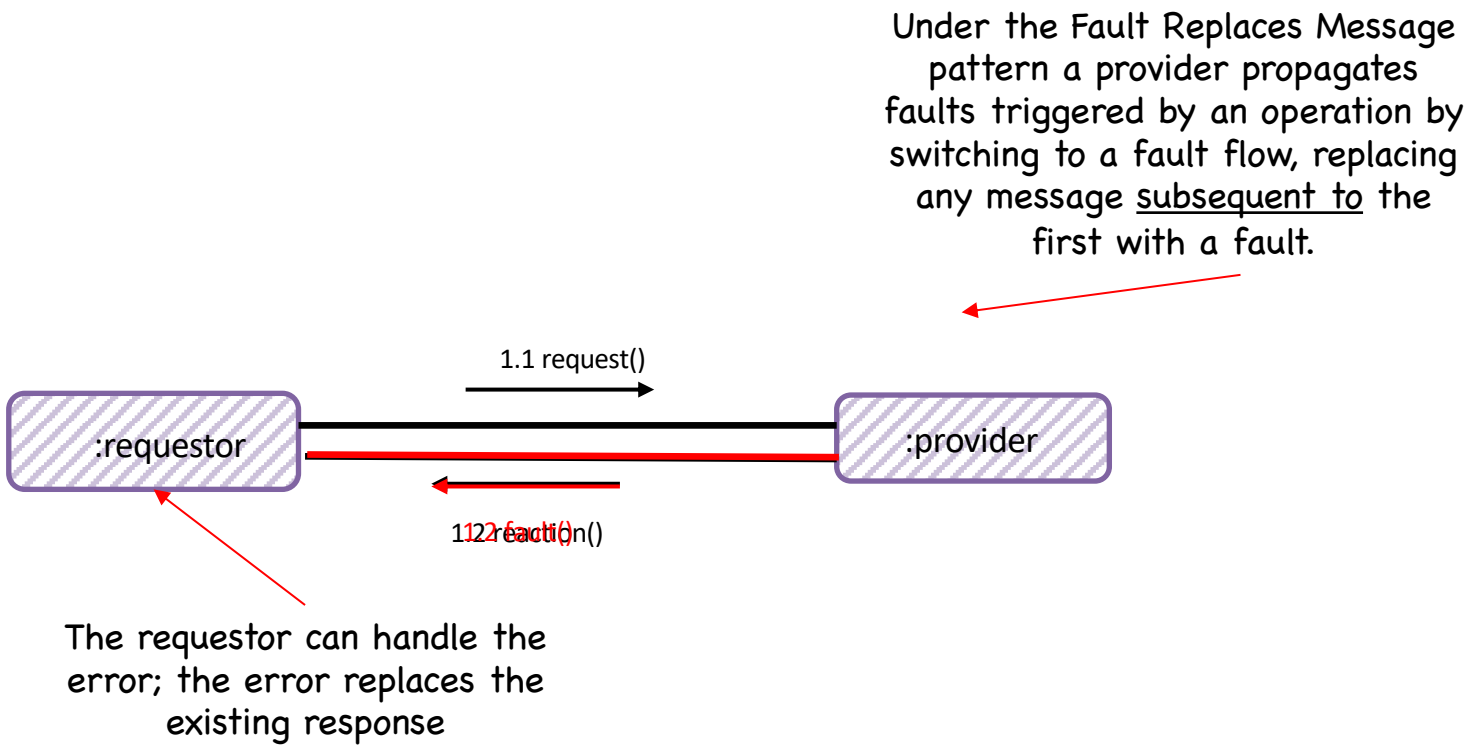
# Message Triggers Fault (Robust In-Only)



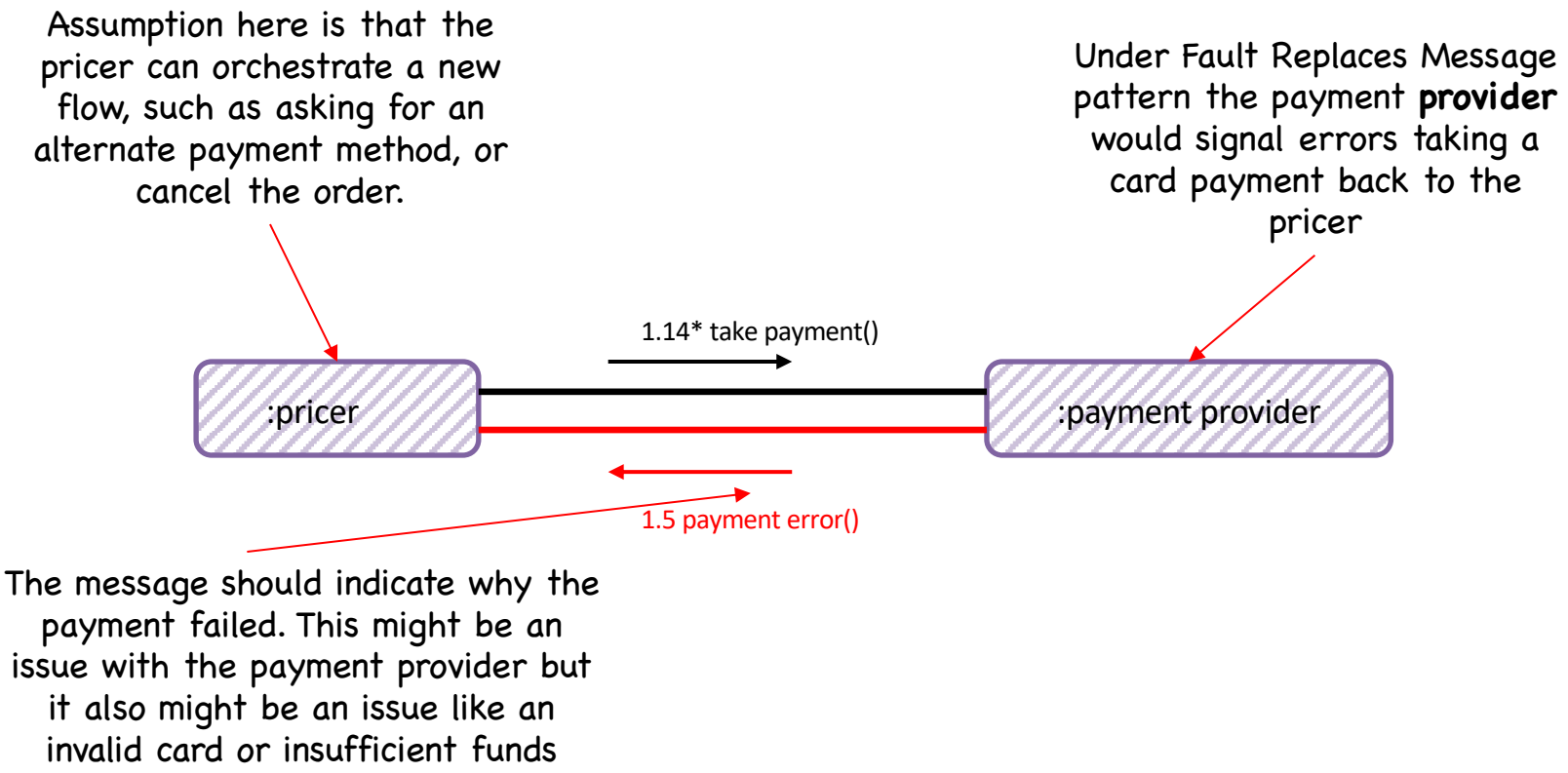
# Message Triggers Fault



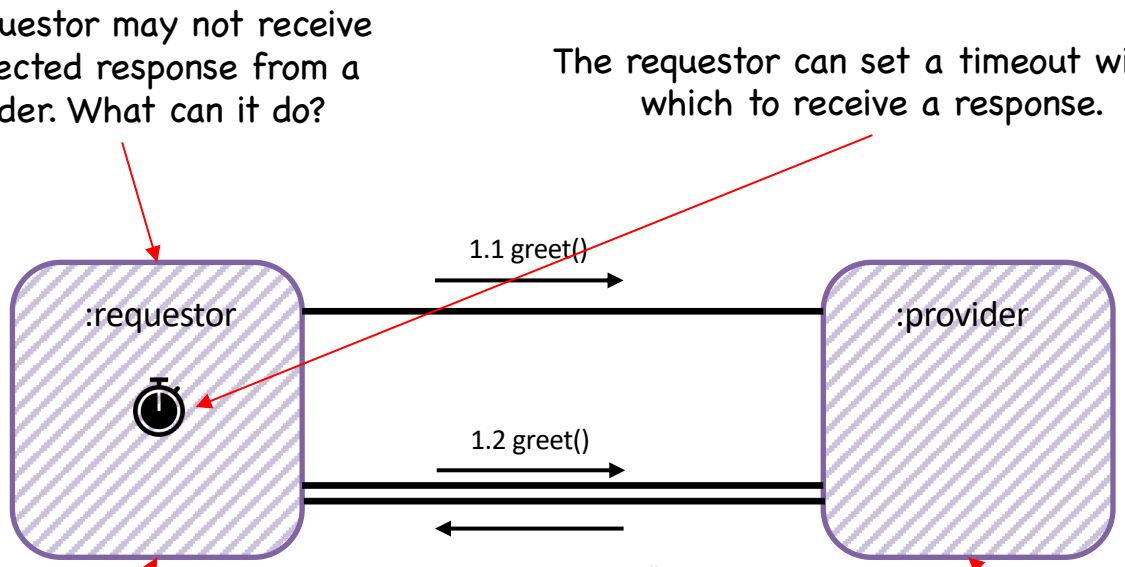
# Fault Replaces Message



# Fault Replaces Message (Robust In-Out)



# In-Out-Retry

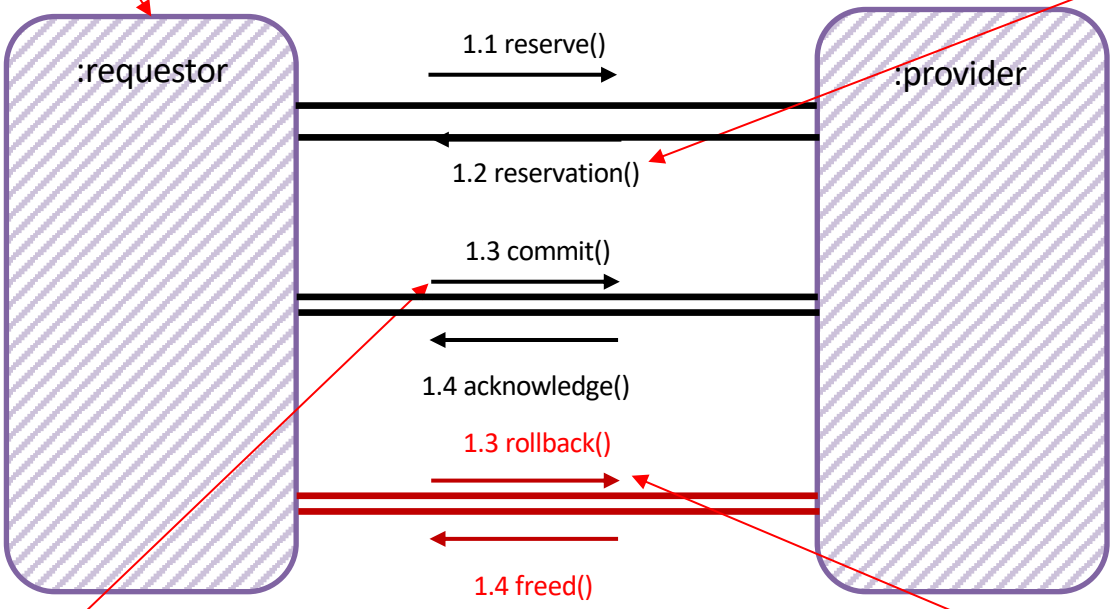

The requestor may not receive an expected response from a provider. What can it do?
The requestor can set a timeout within which to receive a response.
1.1 greet()
:requestor
:provider
1.2 greet()
1.3 acknowledge()
The requestor can choose to retry if it does not receive a response within that time window.
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.

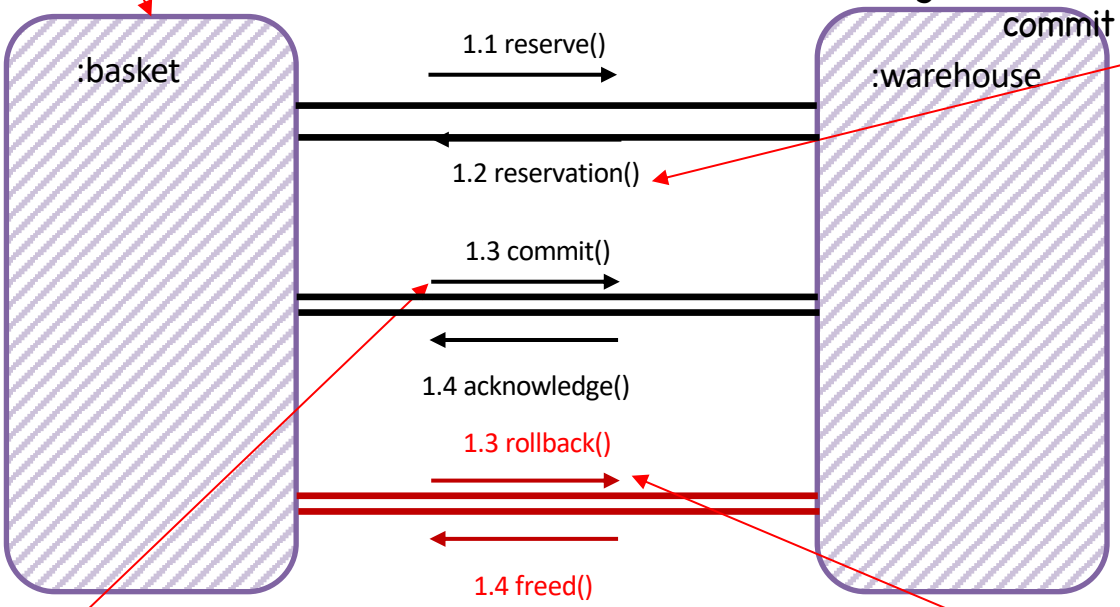
If the requestor fails, it then rolls back to ask the provider to free the reserved capacity.

# 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

The warehouse reserves the stock – with a timeout (so others can purchase if we don't) – during which time it awaits a message to commit or rollback



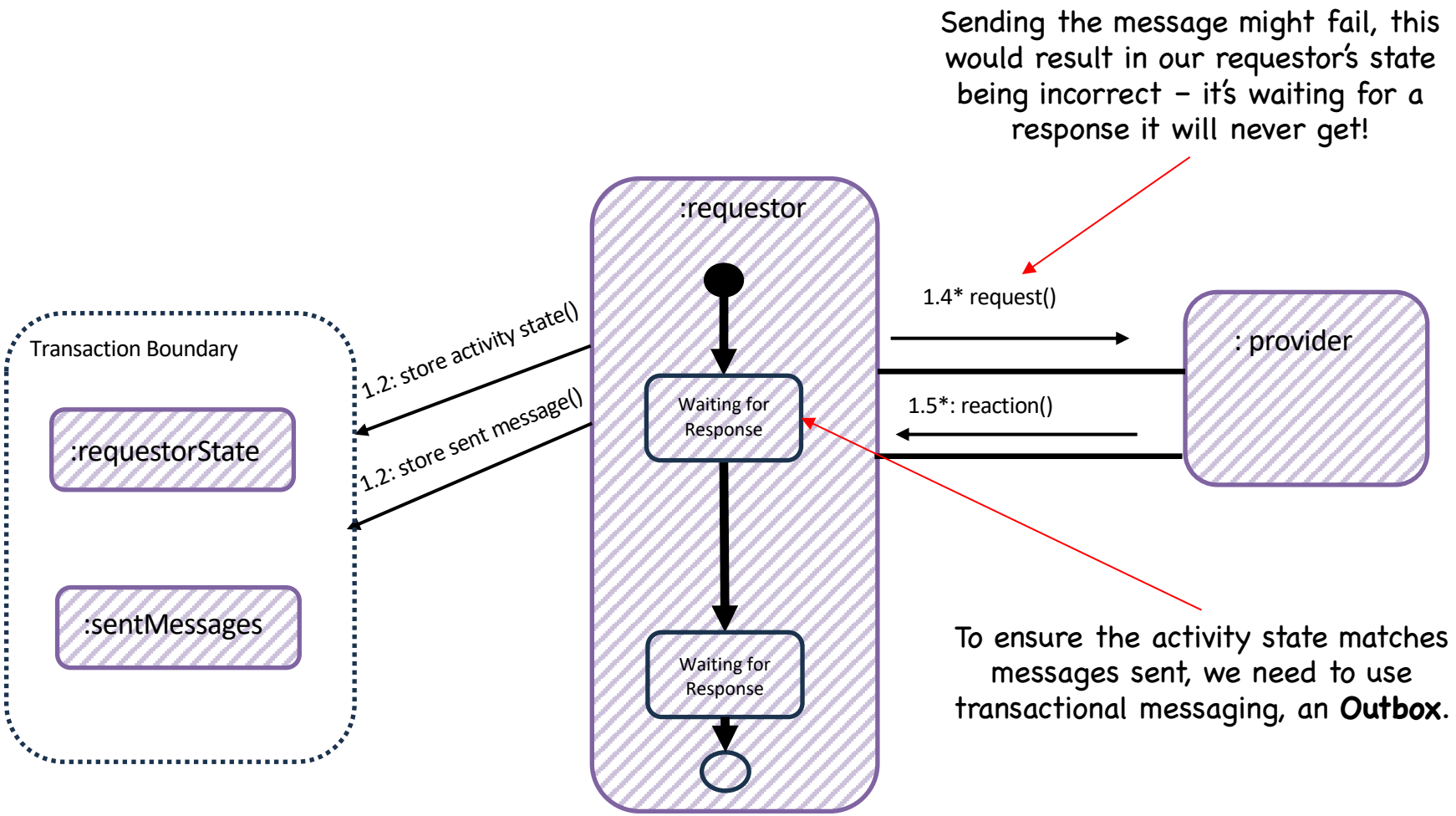

If the customer finishes shopping and pays for the basket before the time limit, then it commits to ask the warehouse to allocate the stock.

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

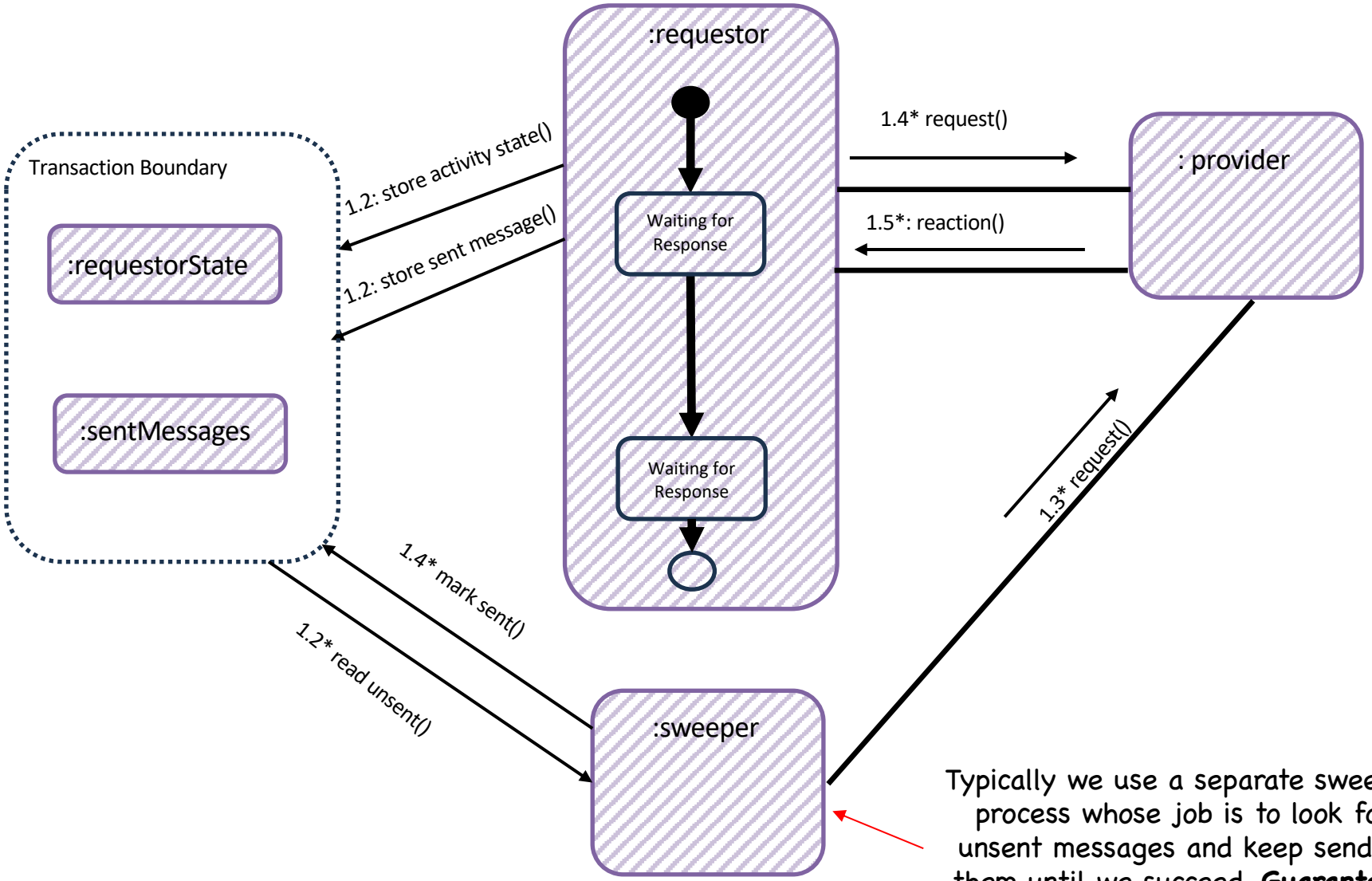
How do we ensure that our messaging is reliable?

# RELIABLE MESSAGING

# Reliable Messaging – Outbox

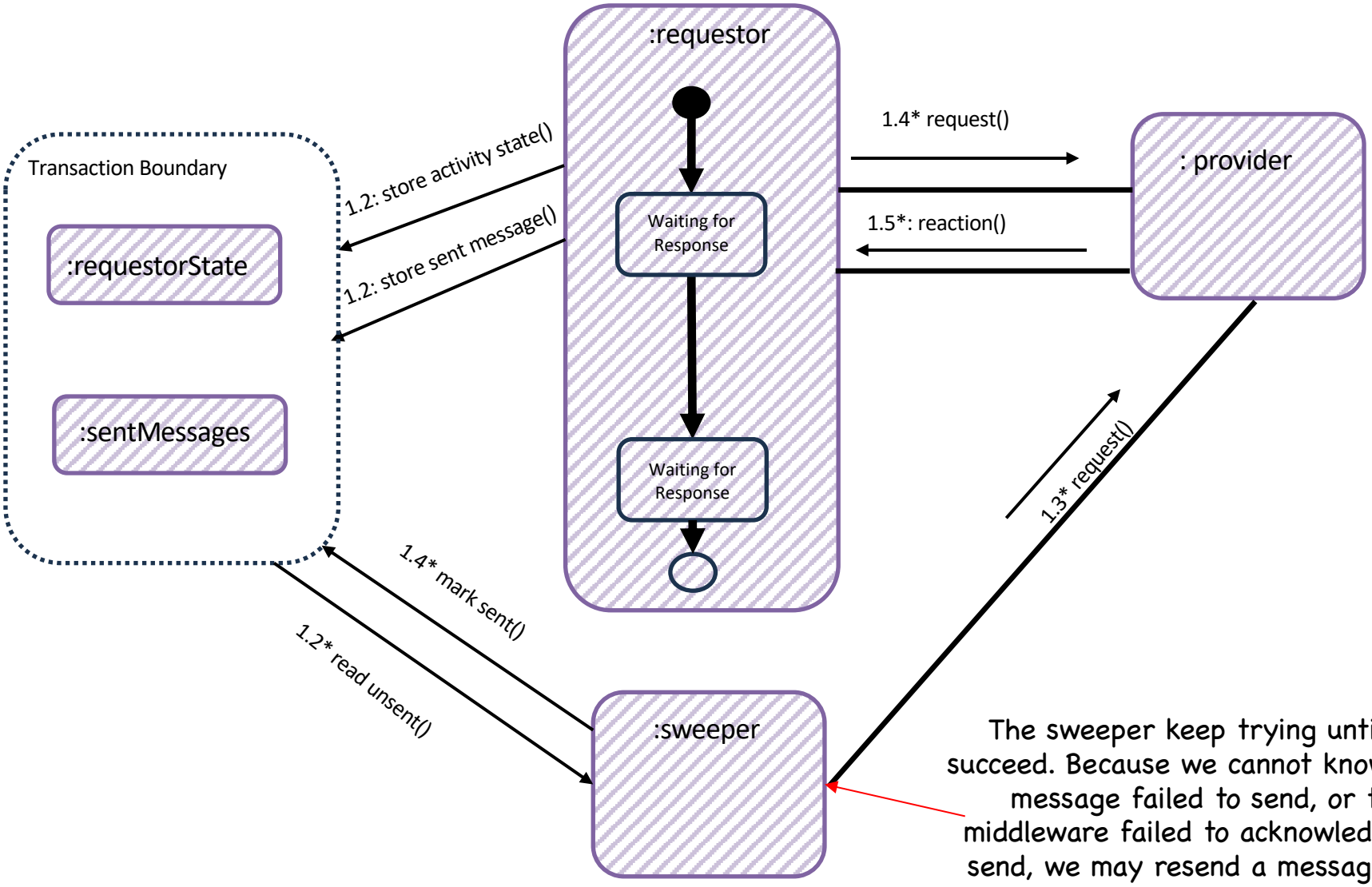


# Reliable Messaging – Guaranteed Delivery



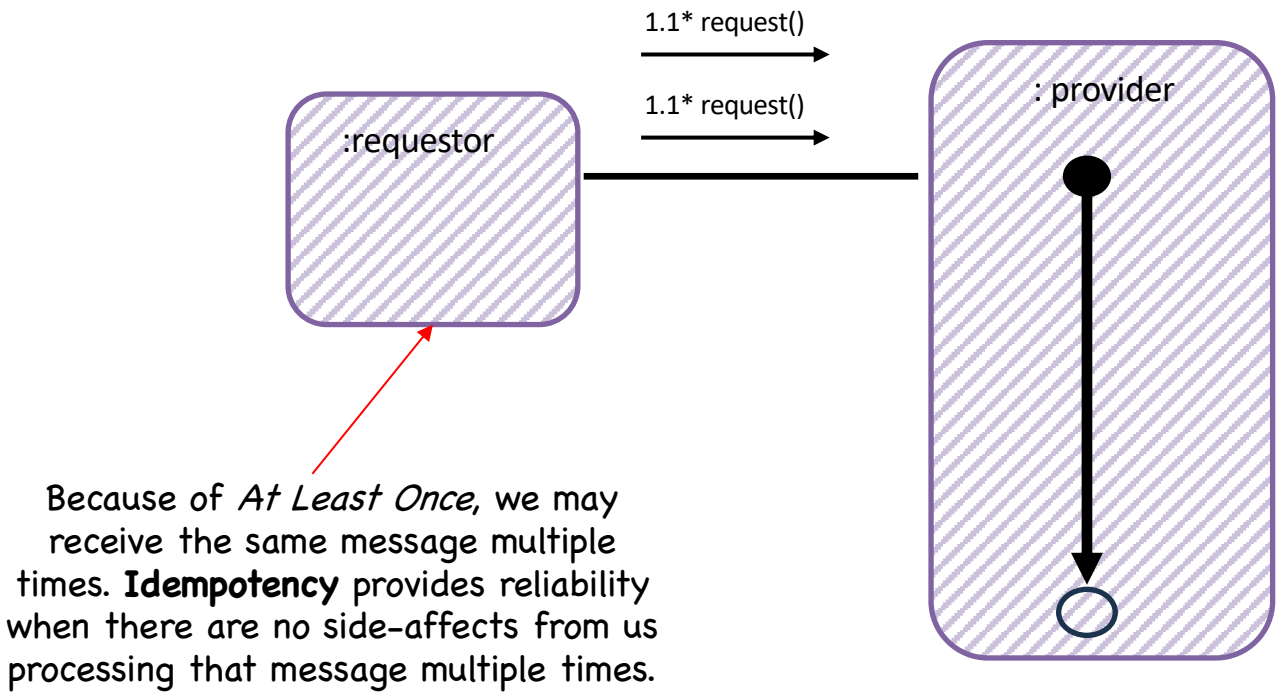

Typically we use a separate sweeper process whose job is to look for unsent messages and keep sending them until we succeed. **Guaranteed Delivery.**

# Reliable Messaging – At Least Once

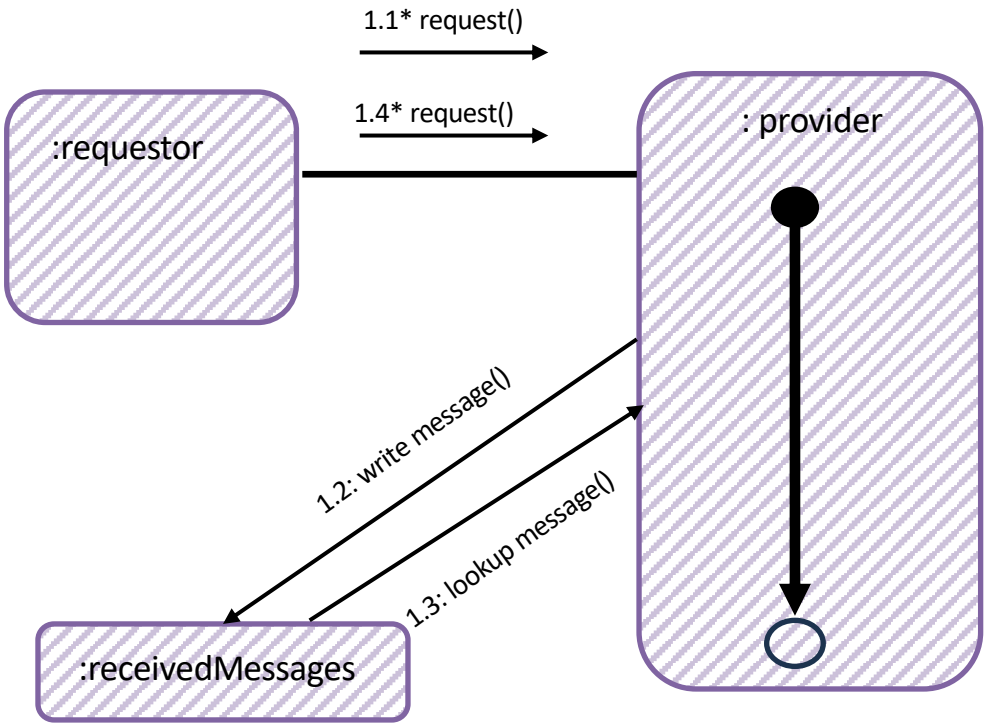



The sweeper keep trying until we succeed. Because we cannot know if the message failed to send, or the middleware failed to acknowledge our send, we may resend a message that has been sent. **At Least Once.**

# Reliable Messaging - Idempotency

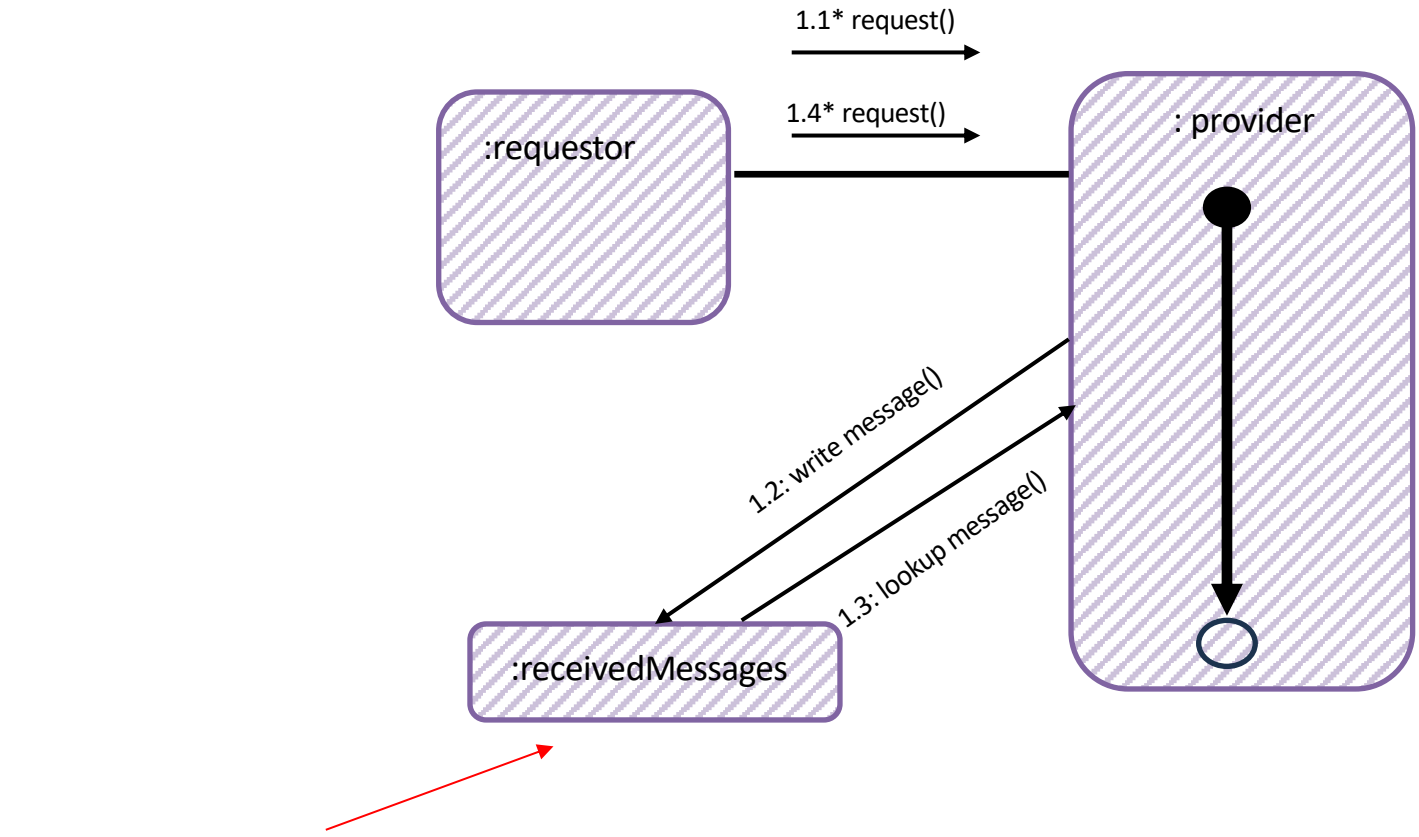


# Reliable Messaging - Inbox



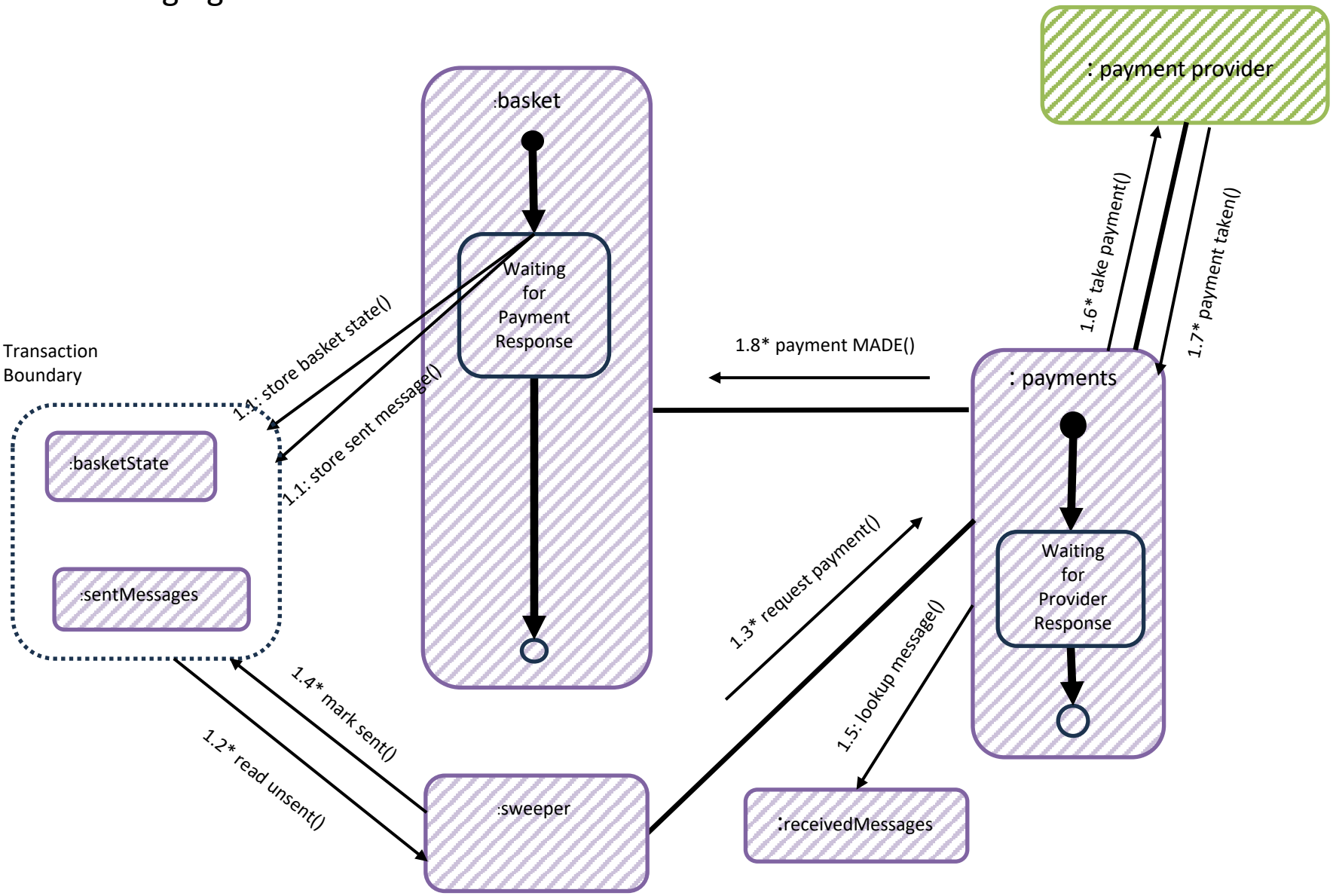

Because of *At Least Once*, to ensure that we have not already processed a message, we use a store seen messages, an **Inbox**.

# Reliable Messaging – Once Only



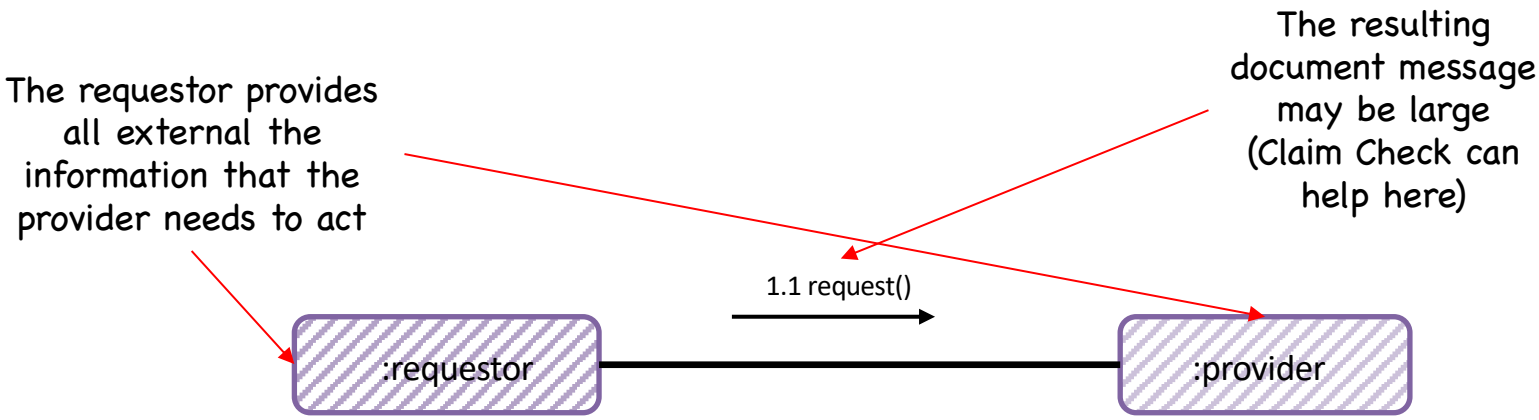

Because an Inbox filters out duplicates, we can turn *At Least Once* into **Once Only**.

# Reliable Messaging



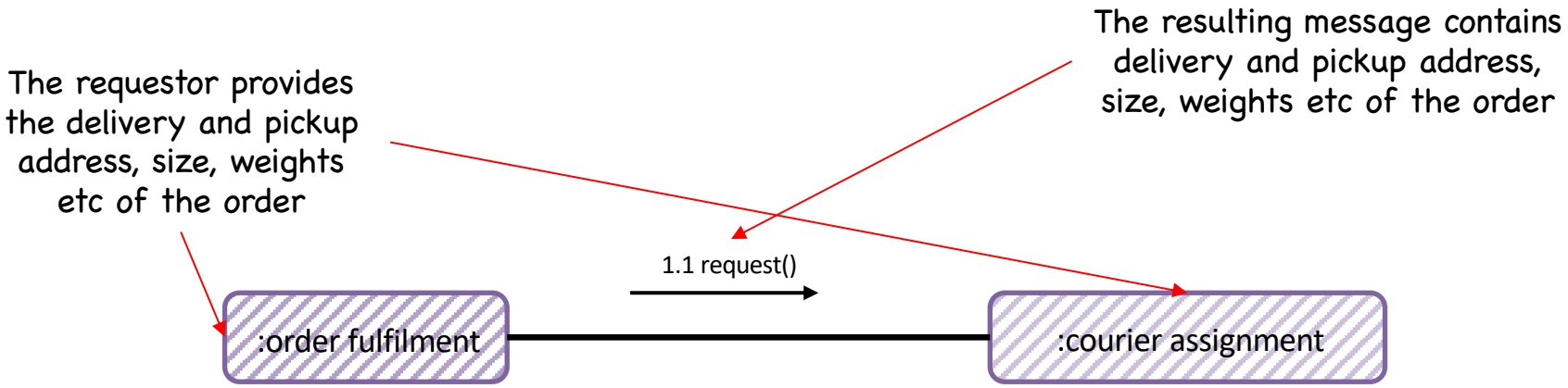
# FAT AND SKINNY

# Fat Message



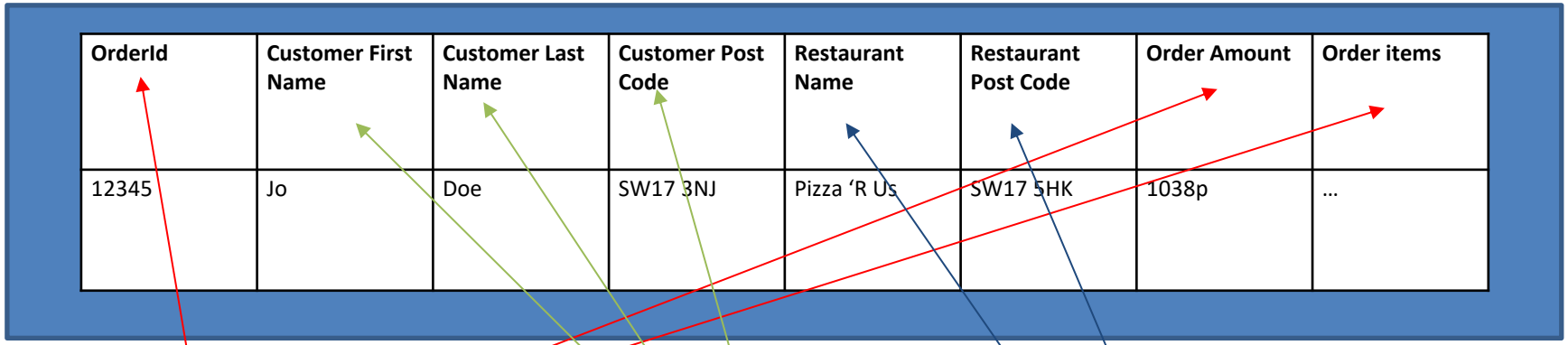

With a **Fat Message** the requestor sends across all the external information a provider may need to perform the operation.

# Fat Message Example



# Fat Message, Transitive Dependencies

Purchase Order Message



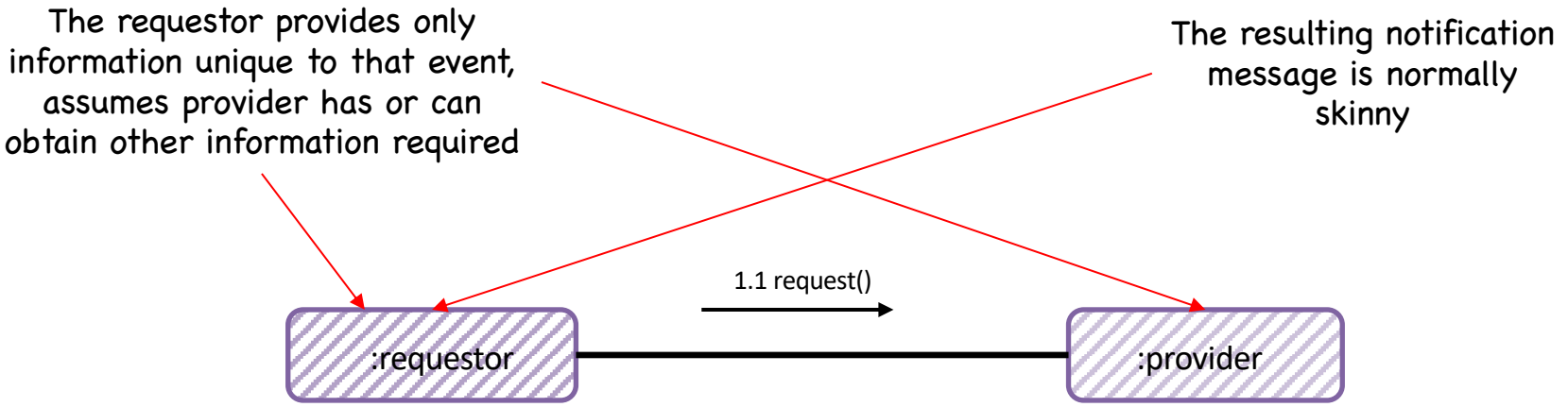
| OrderId | Customer First Name | Customer Last Name | Customer Post Code | Restaurant Name | Restaurant Post Code | Order Amount | Order items |
|---|---|---|---|---|---|---|---|
| 12345 | Jo | Doe | SW17 3NJ | Pizza 'R Us | SW17 5HK | 1038p | … |

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

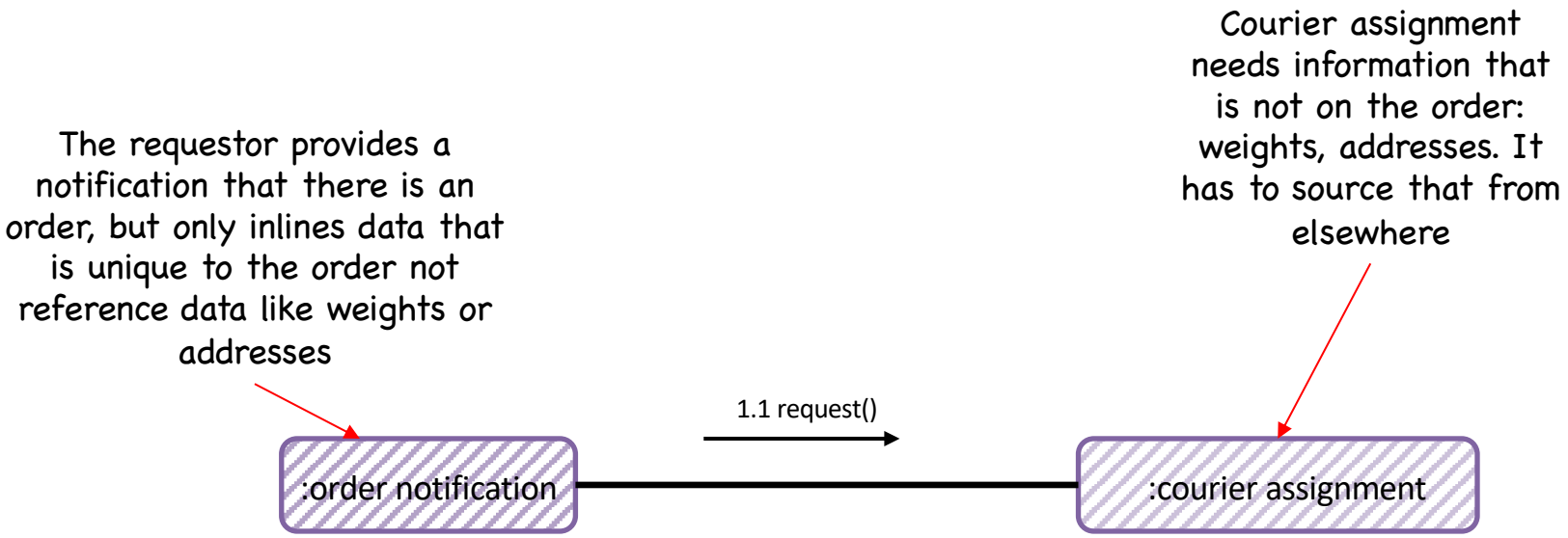
This data is a **transitive dependency**, it has a lifetime of the Restaurant and their schema changes if the Restaurant schema changes – which may force us to change the message for Restaurant changes

# Skinny Request




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

# Skinny Request, Example



# 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

Restaurant

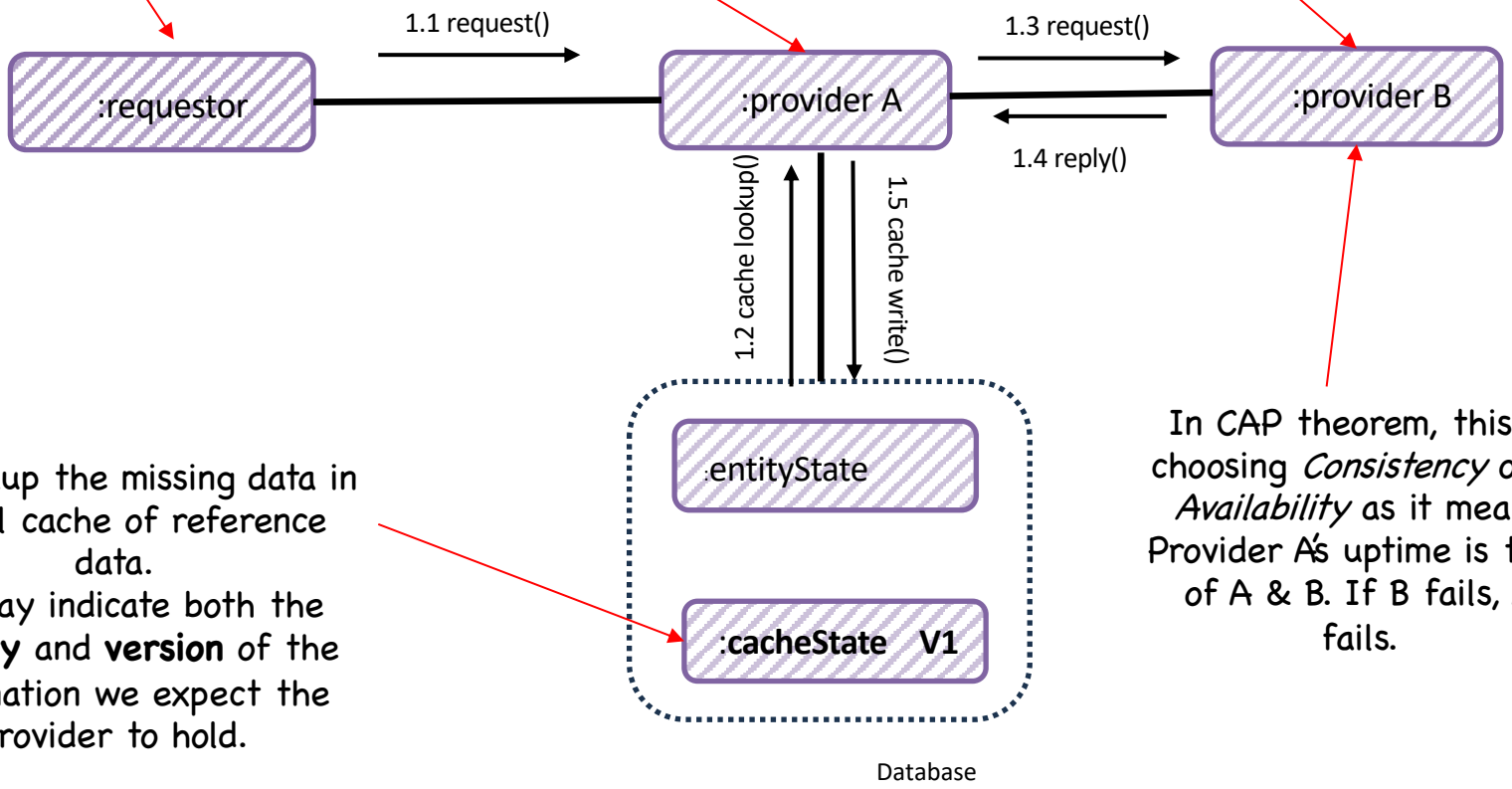
| Restaurant Name | Restaurant Post Code |
|---|---|
| Pizza 'R Us | SW17 5HK |

We need to lookup this id with other providers

# 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*

On a cache miss we request the data from provider B, and store it in the cache. The cache is choosing *Availability* over *Consistency*, **except** on a miss.




We lookup the missing data in a local cache of reference data.
We may indicate both the **identity** and **version** of the information we expect the provider to hold.

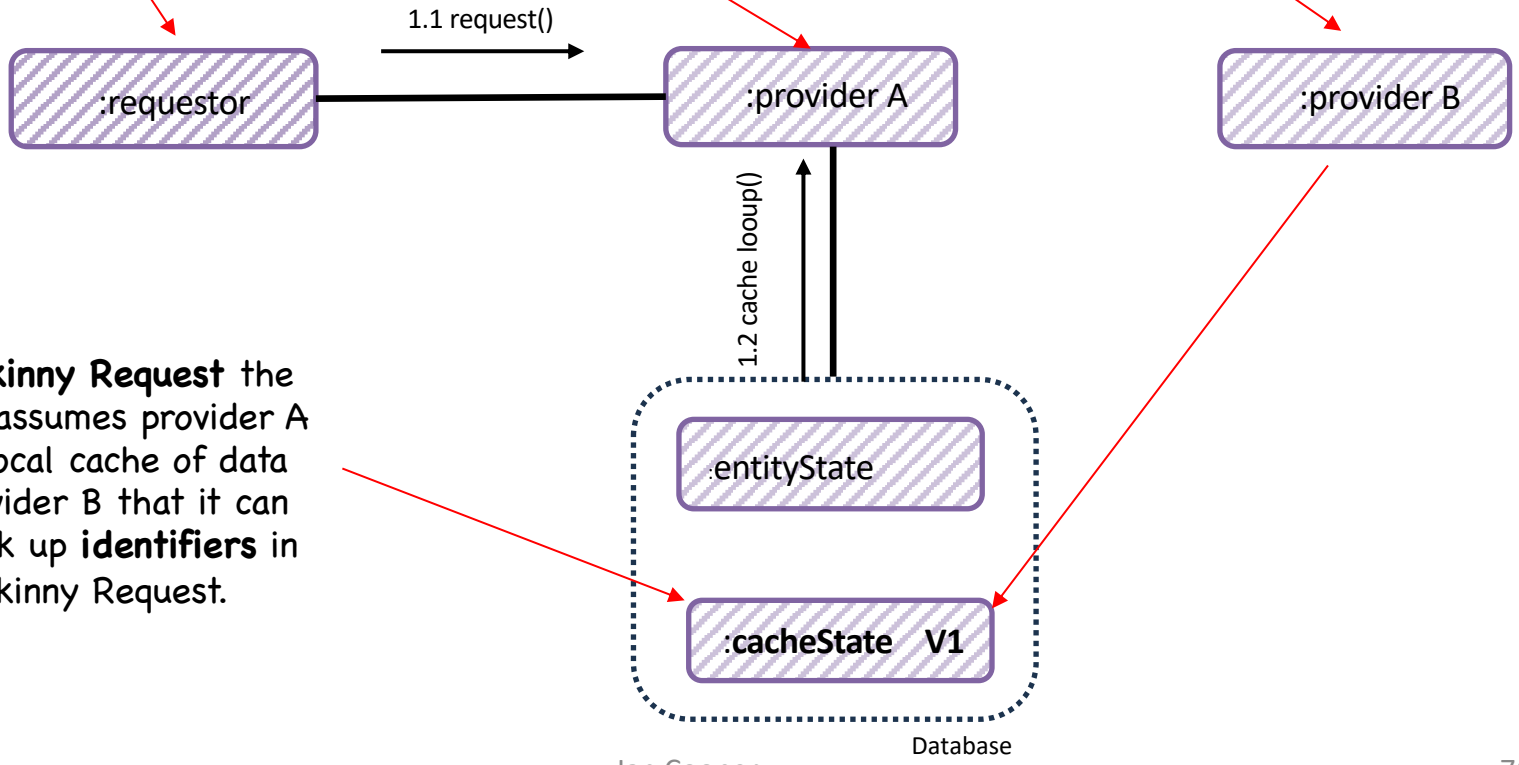
In CAP theorem, this is choosing *Consistency* over *Availability* as it means Provider A's uptime is that of A & B. If B fails, A fails.

# Skinny Request using Reference Data

Data that leaves Provider B via an API is **Reference Data**.
It is:
- *Immutable*
- *Versioned*
- *Stale*

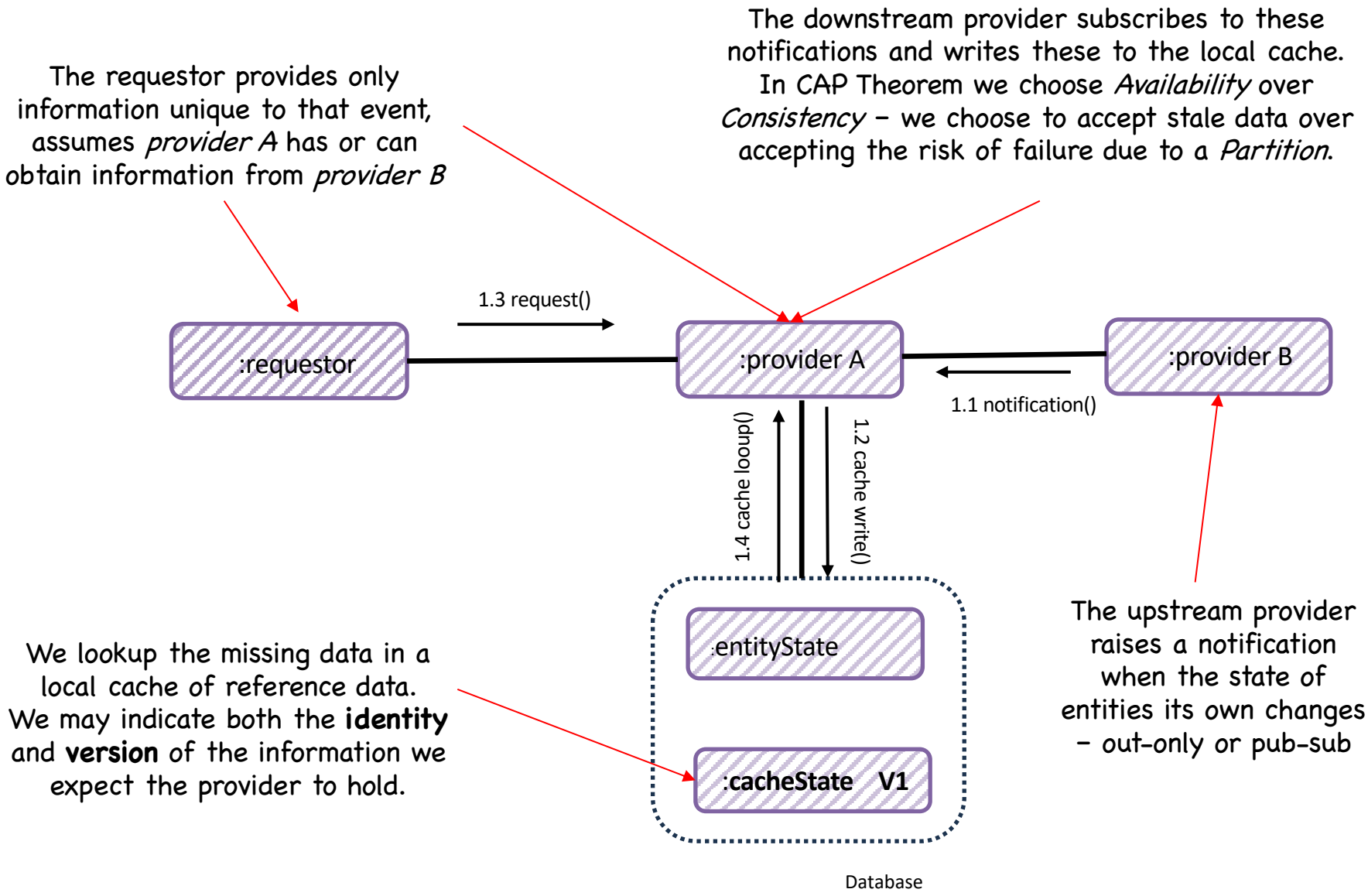
It can be cached locally to provider A to prevent the need for frequent lookups

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

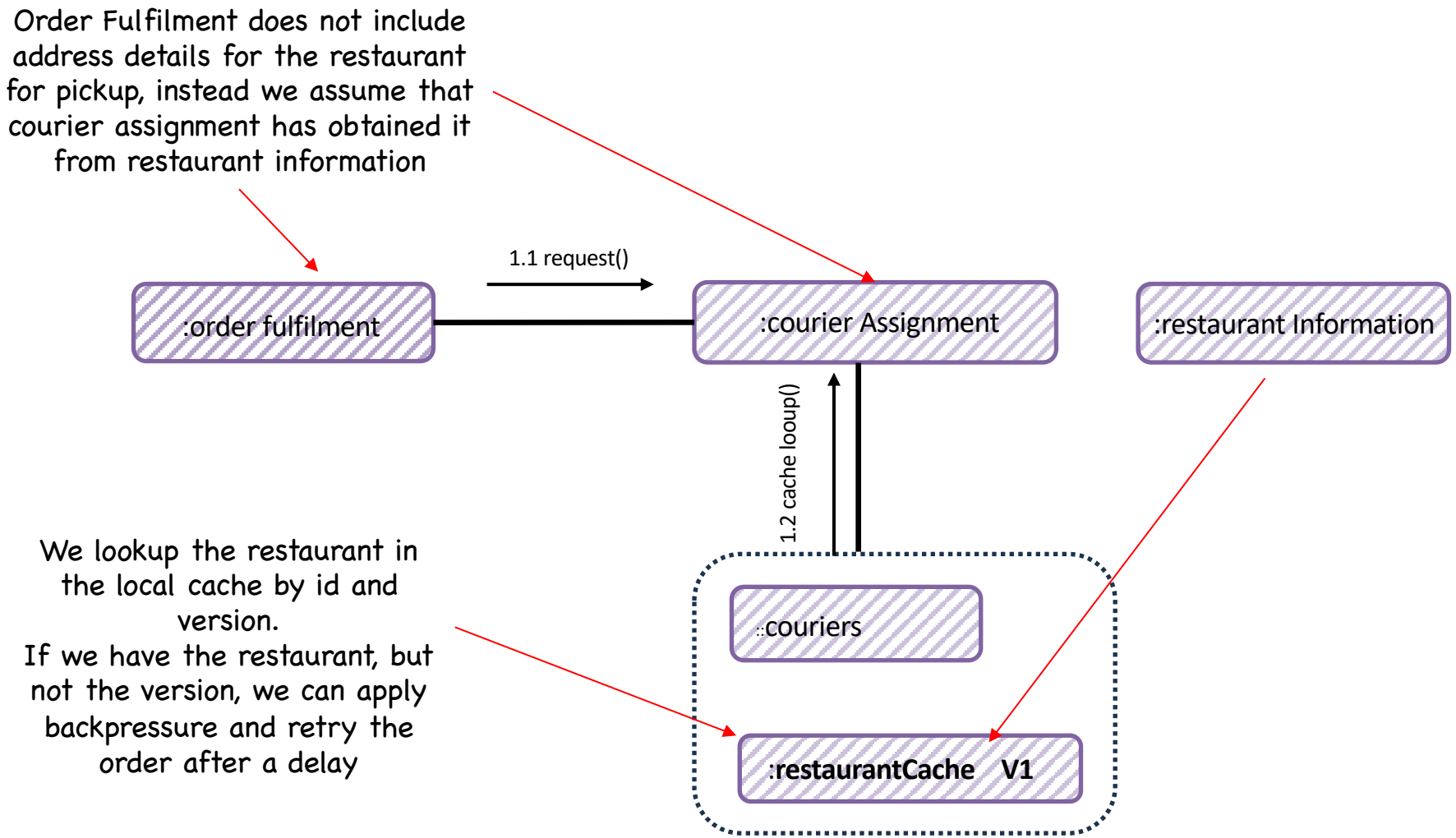



With a **Skinny Request** the requestor assumes provider A B has a local cache of data from provider B that it can use to look up **identifiers** in the Skinny Request.

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



# Skinny Request using Reference Data

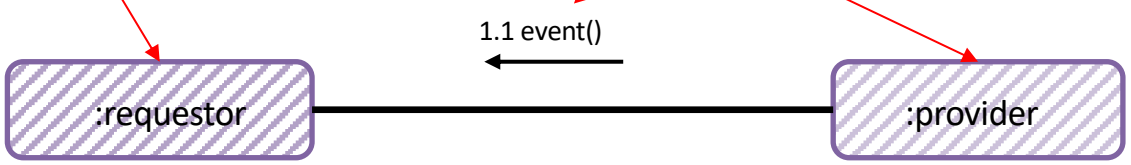


# DOMAIN, SUMMARY, ORDERING

# 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*



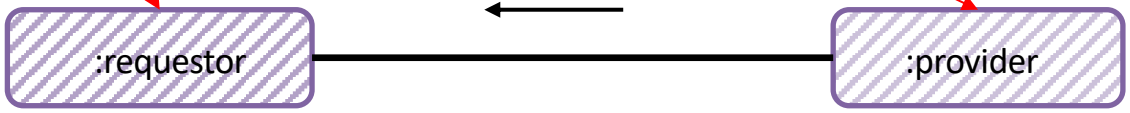

With a **Domain Event** the provider cannot use a *DataType Channel* as the domain events for the observable must be ordered on the same channel, and have a different schema.

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

# Summary Event

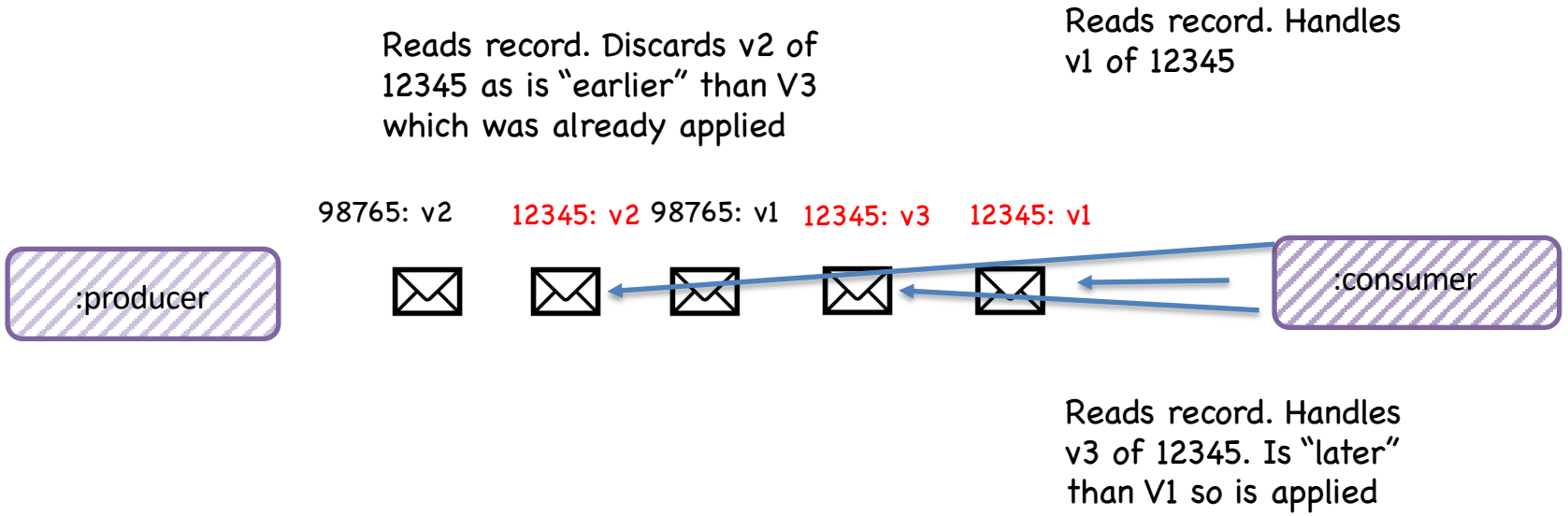
An approach to Pub-Sub where the provider communicates **a summary** of state changes to its requestors

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 uses a *DataType Channel* as there is just one schema used to communicate a snapshot of the observable.

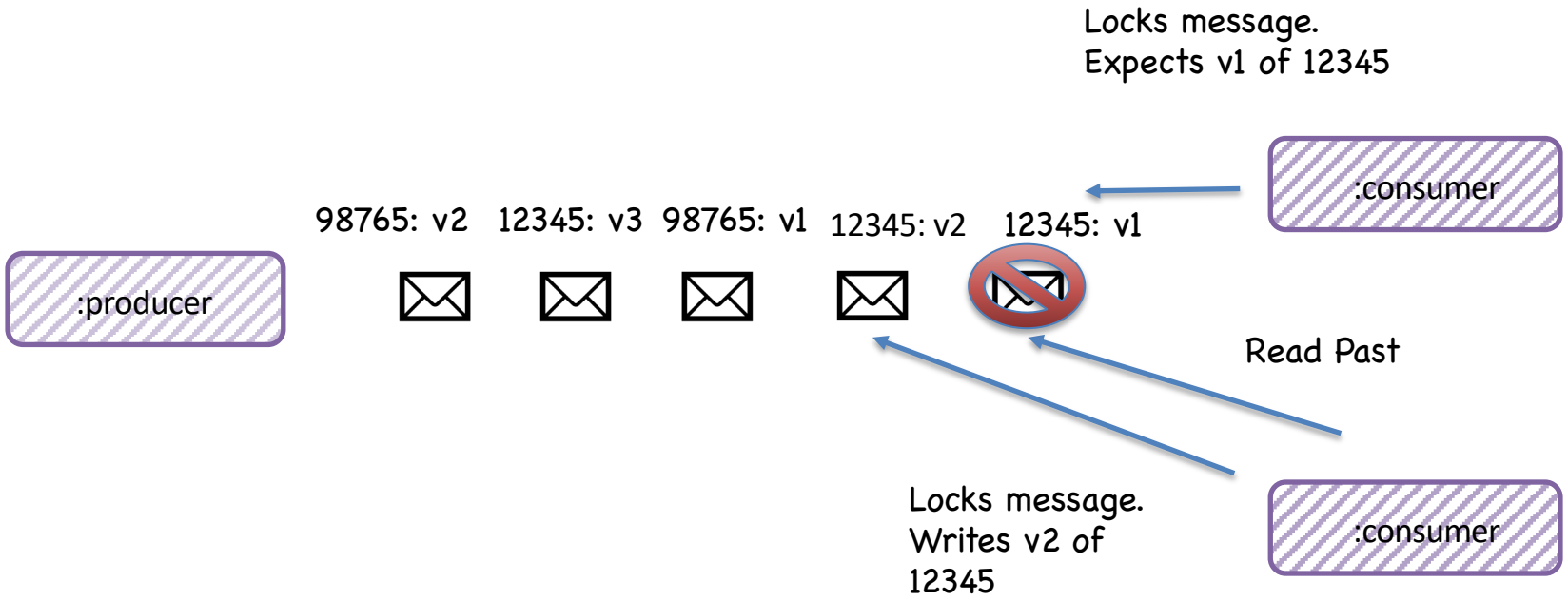
# If Later, Stream




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.

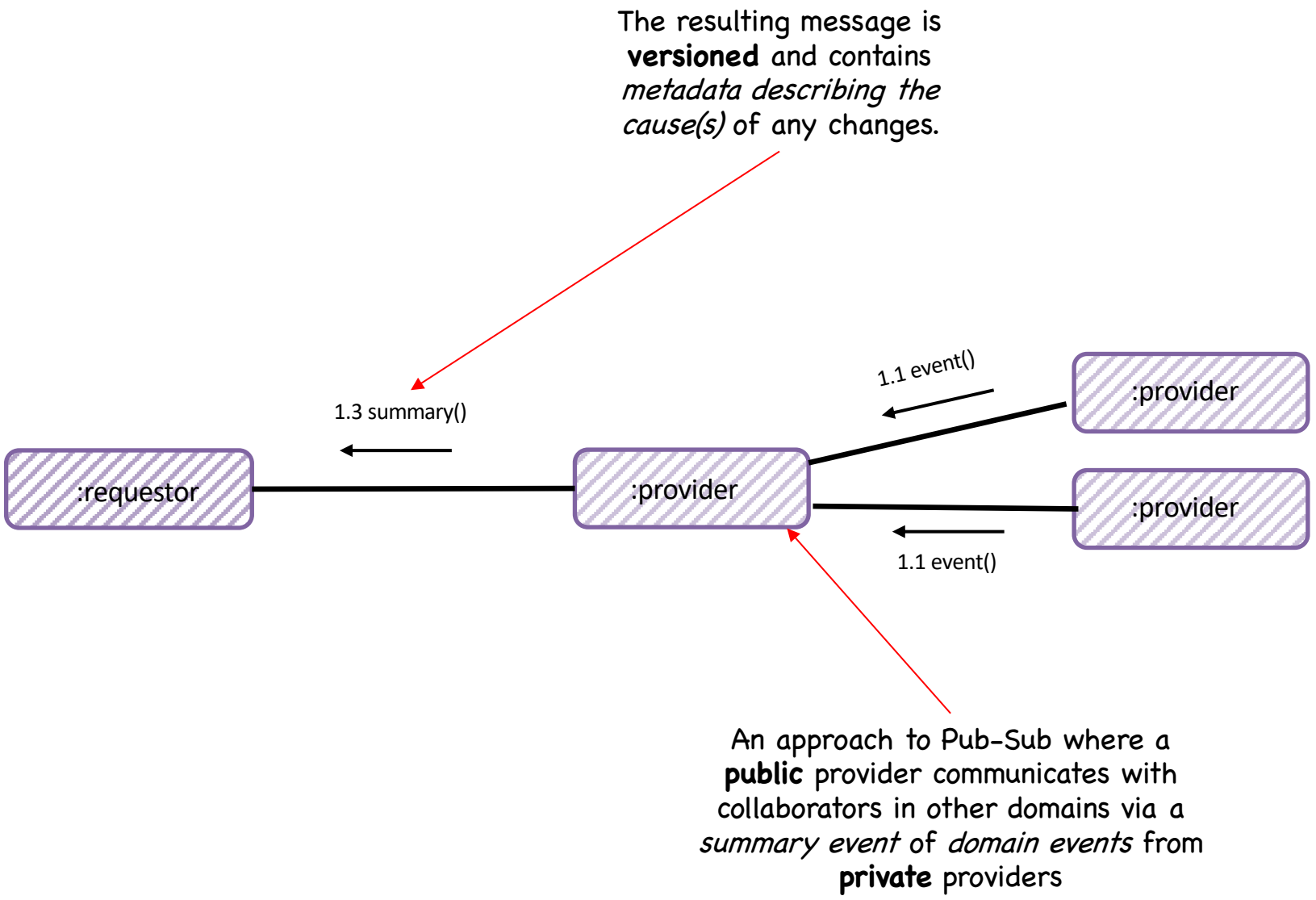
# If Later, Queue




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

# Public and Private Providers

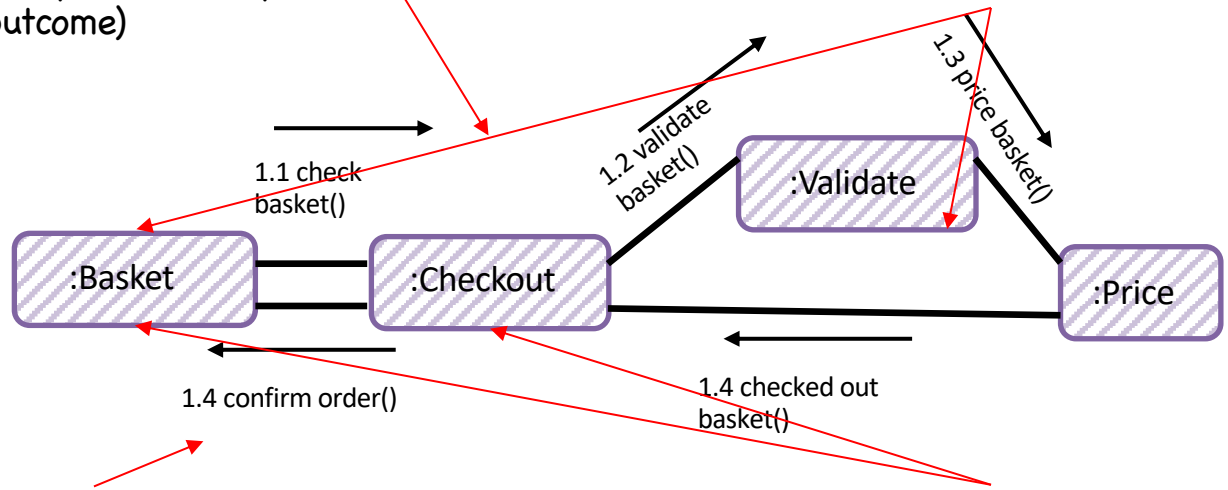


# CONVERSATIONS

# Conversations

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

**Lack of Distributed Transactions**
Whilst 2PC is possible it is rarely used, how do we create an atomic workflow?

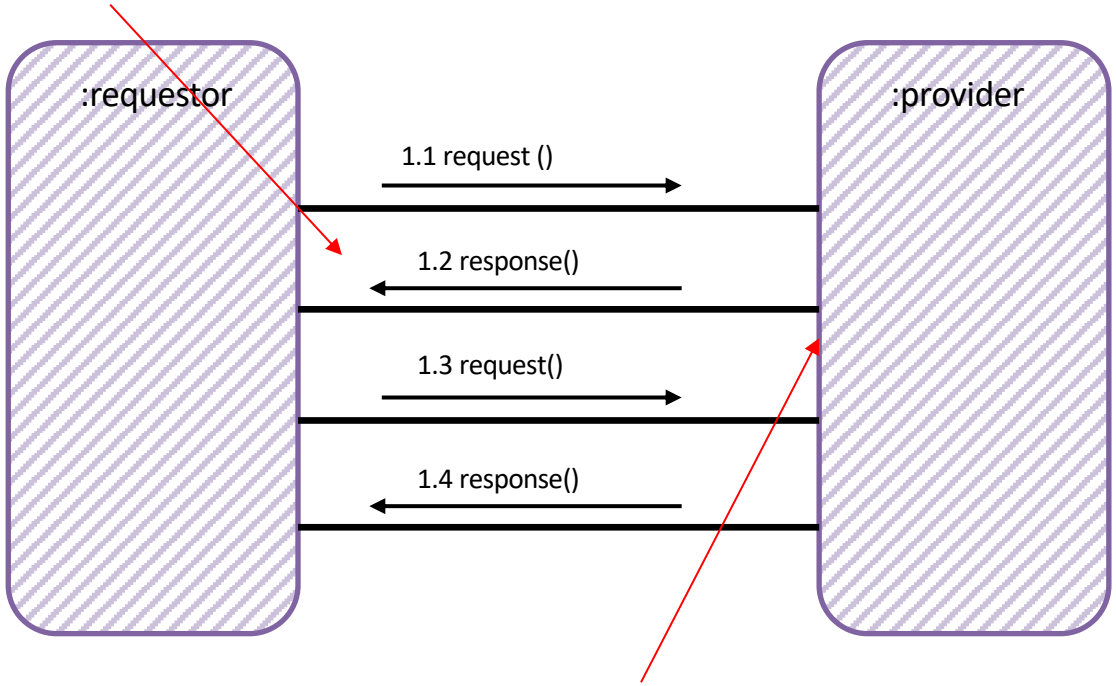



**Command and Control**
Who owns the process? Who informs us about its state? Who knows what step is next?

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

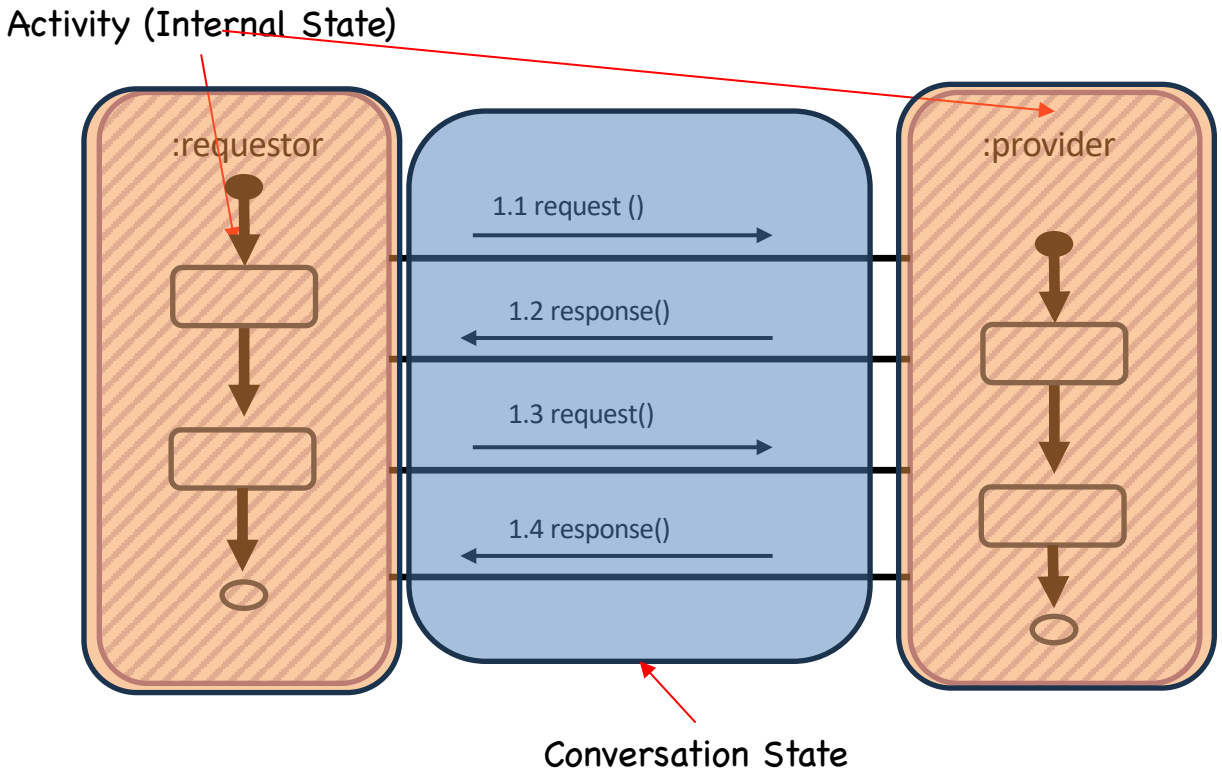
# Turn Taking

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



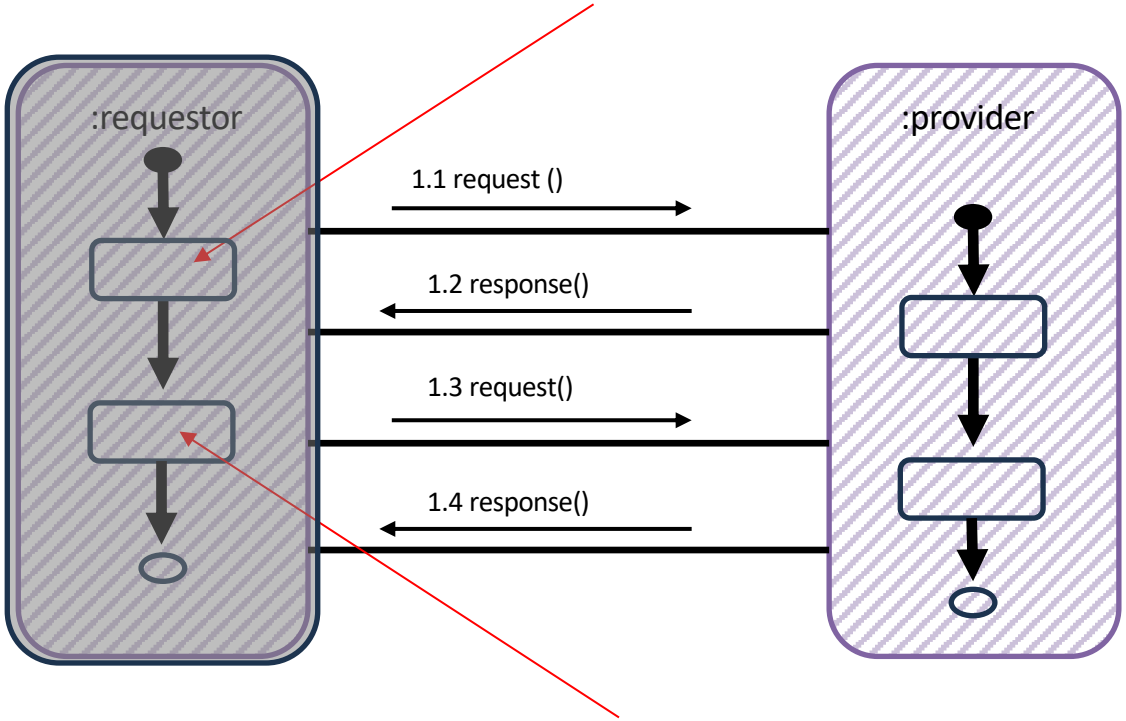

As each participant takes its turn, it takes control of the flow

# Turn Taking


Activity (Internal State)
:requestor
:provider
1.1 request ()
1.2 response()
1.3 request()
1.4 response()
Conversation State

# Turn Taking

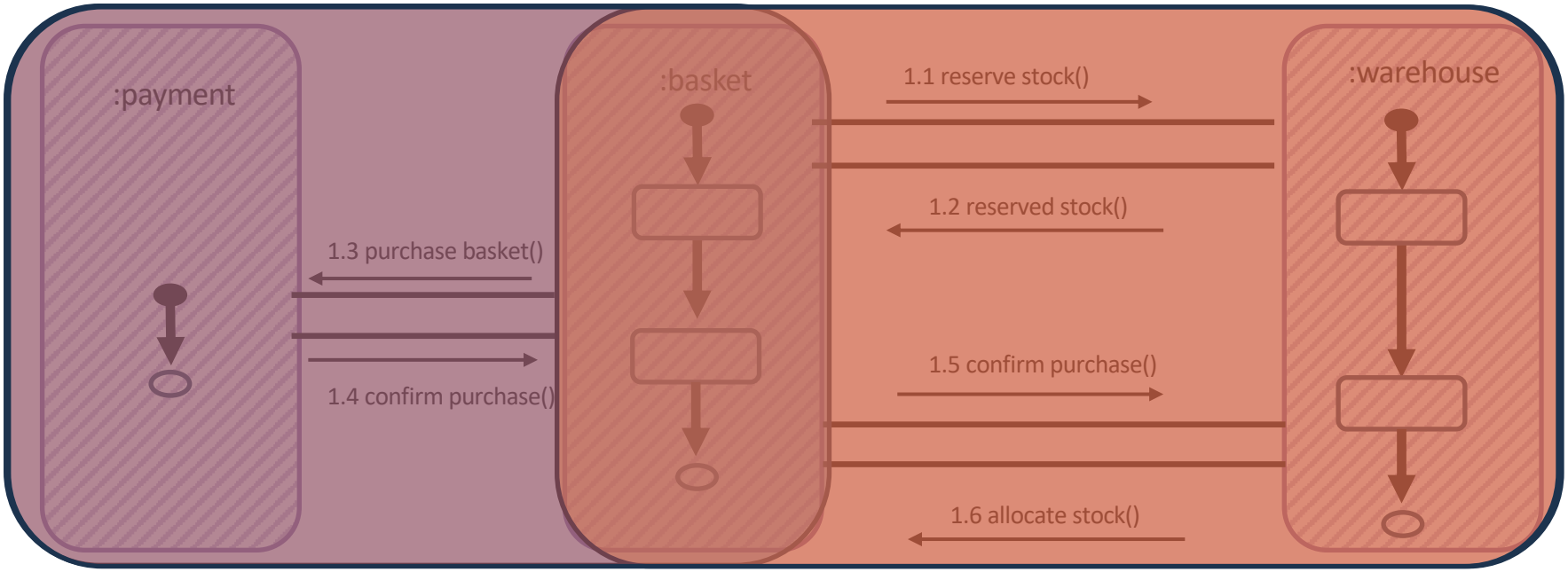
Orchestration: turn taking implies causality, and by implication the requestor's activity orchestrates the conversation with the provider.




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

# Turn Taking

Choreography

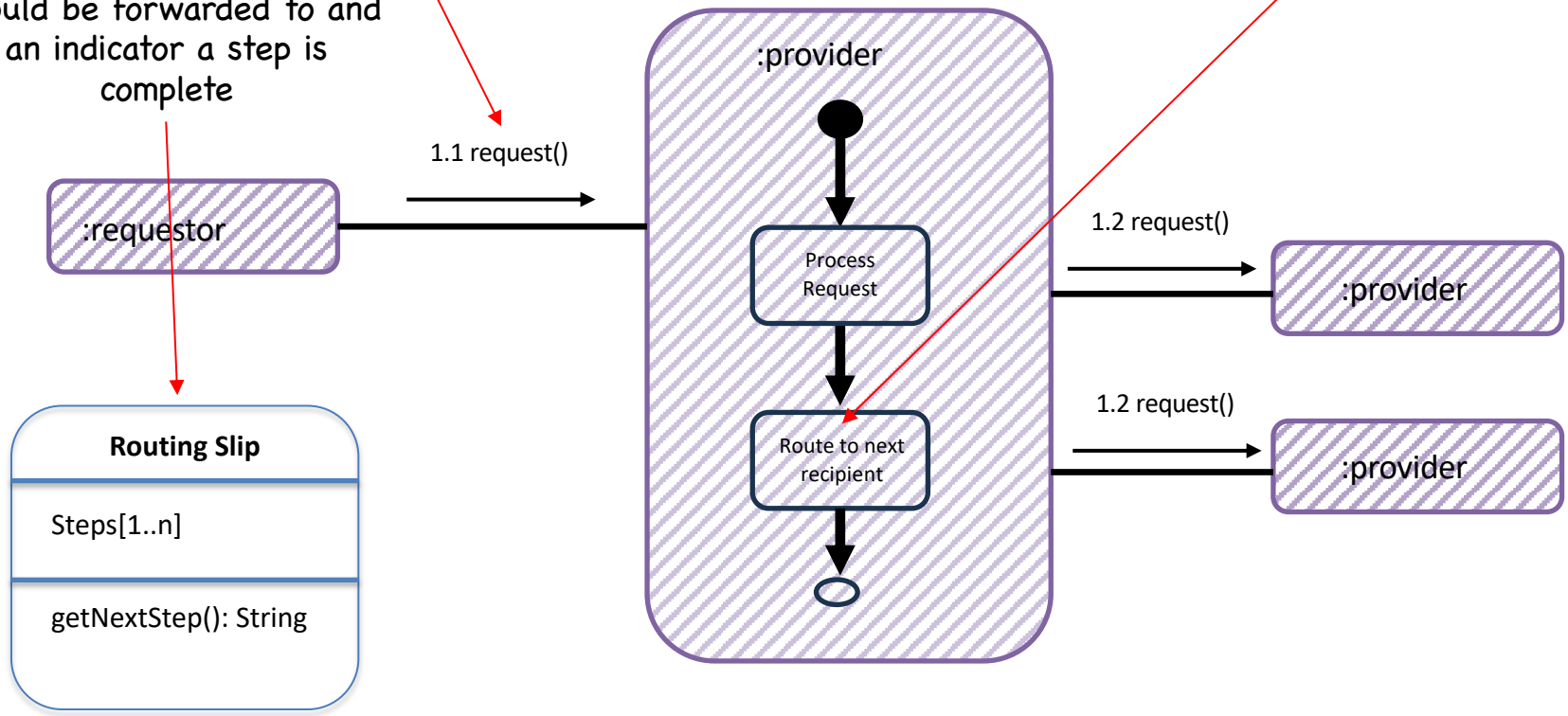



Orchestration

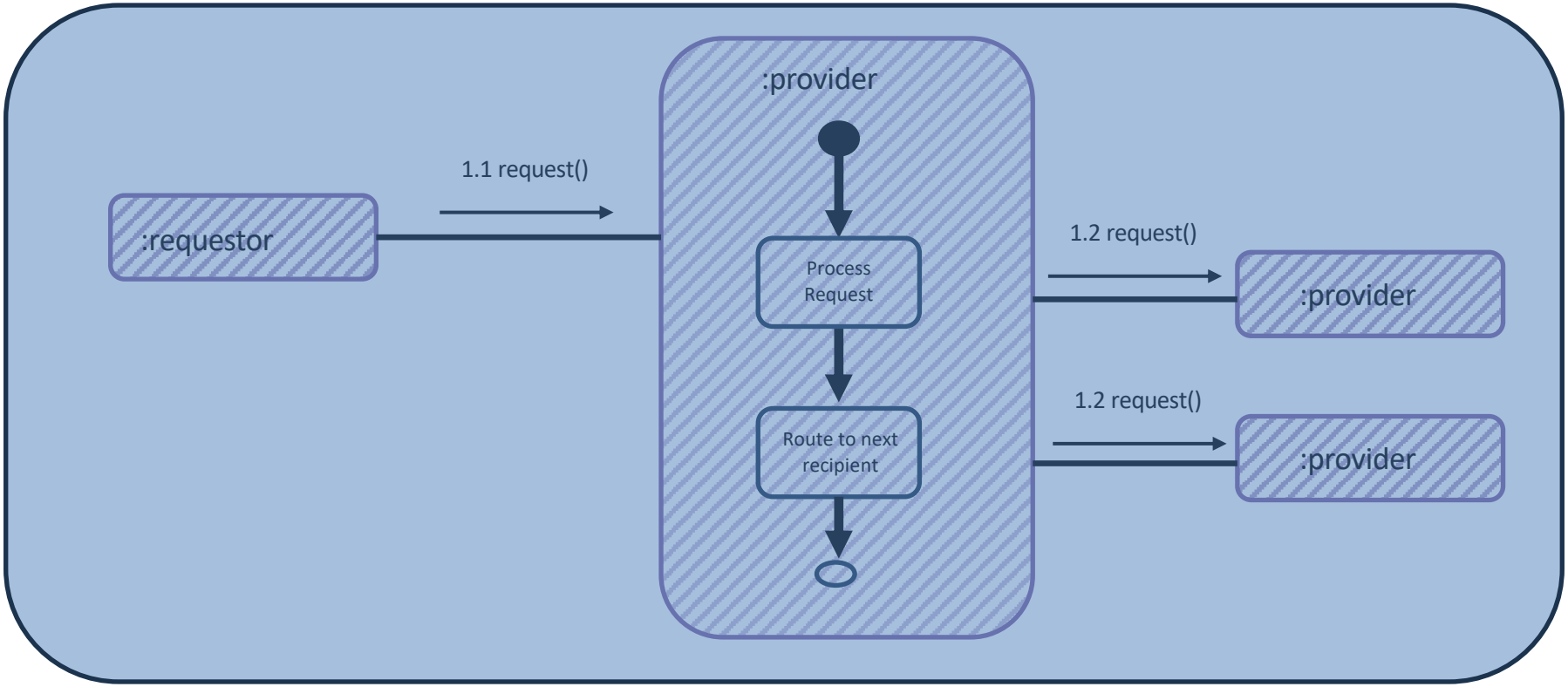
# Routing Slip

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.

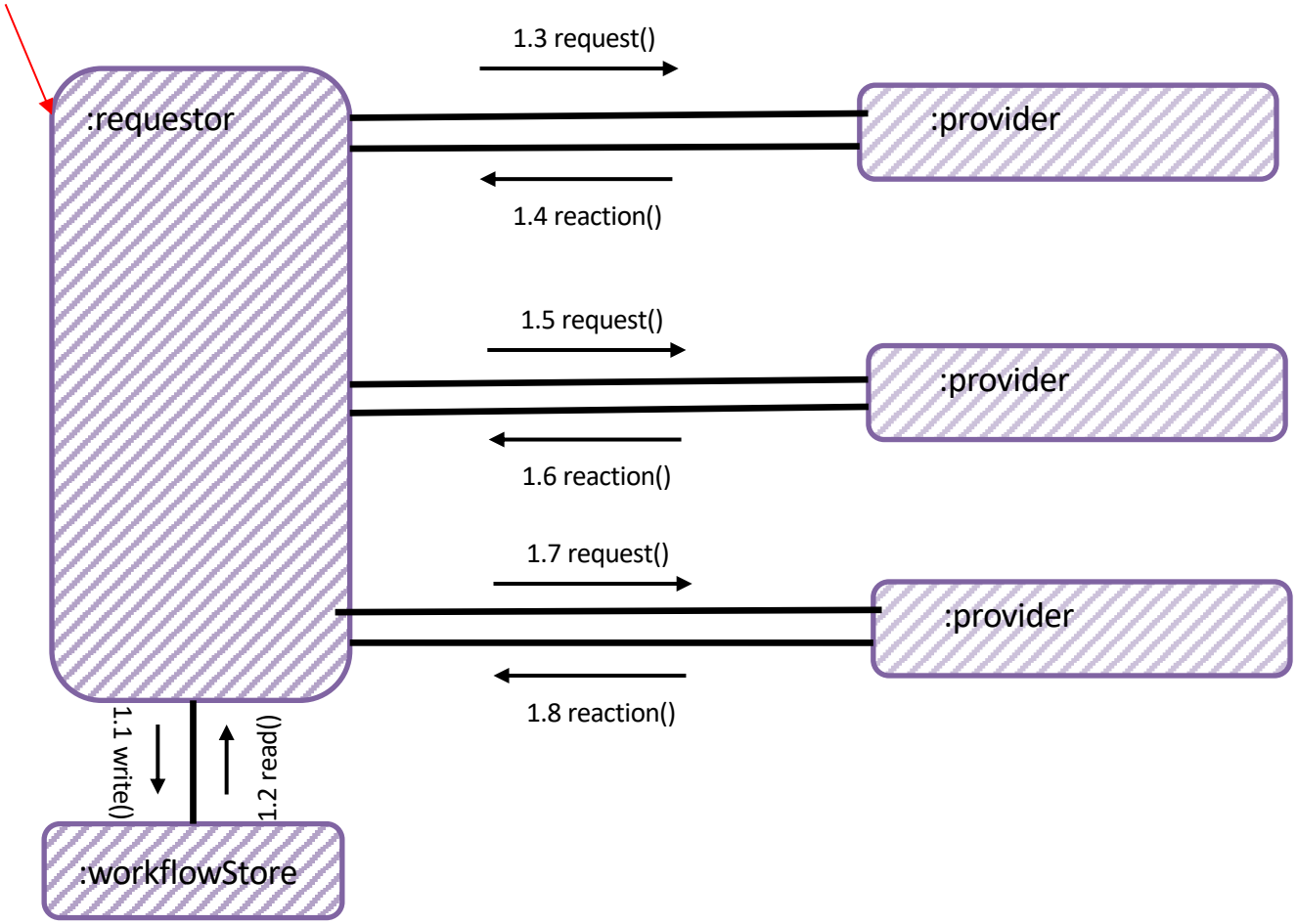


## Orchestration

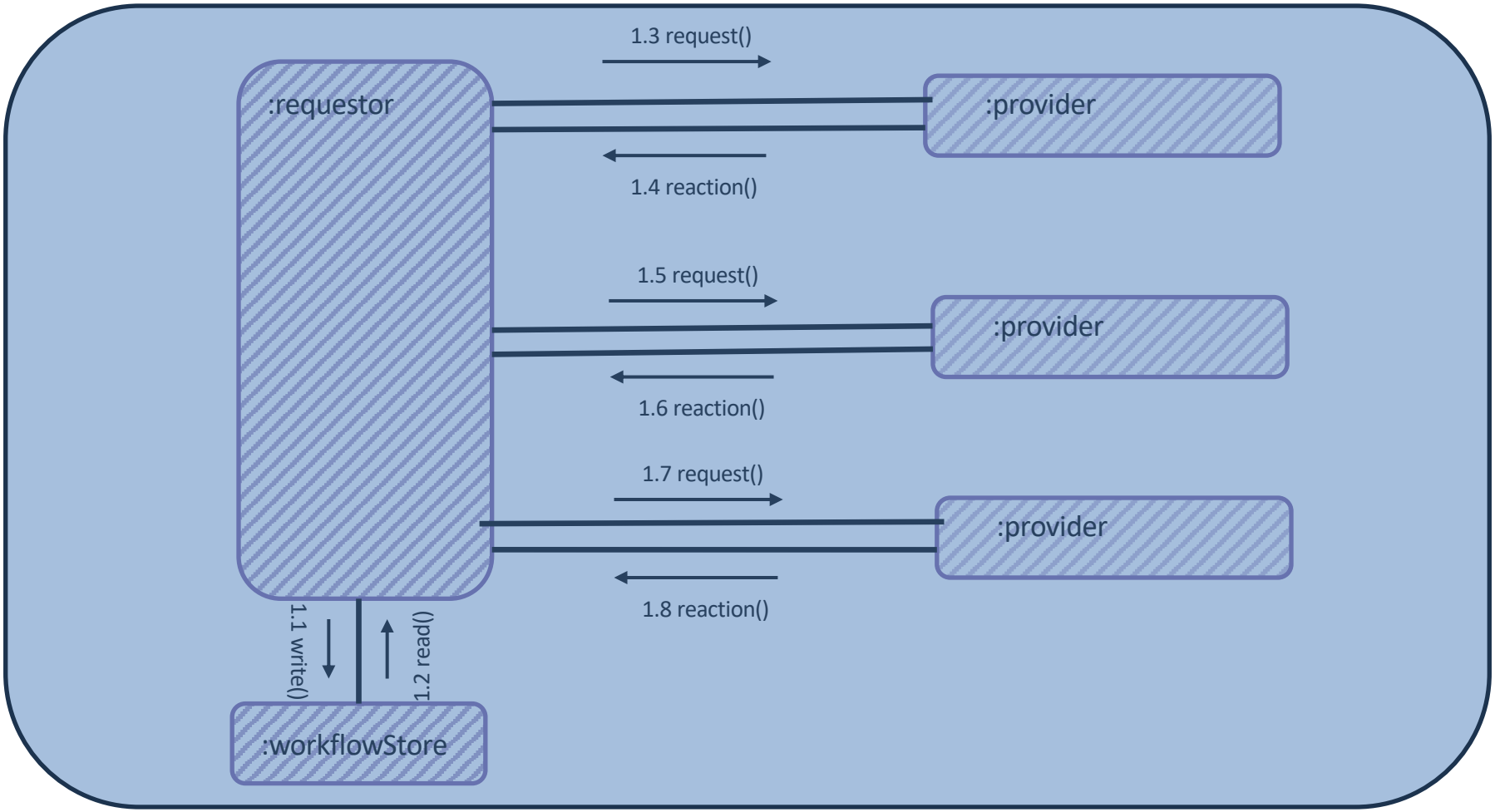


# Process Manager (Saga)

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



## Orchestration



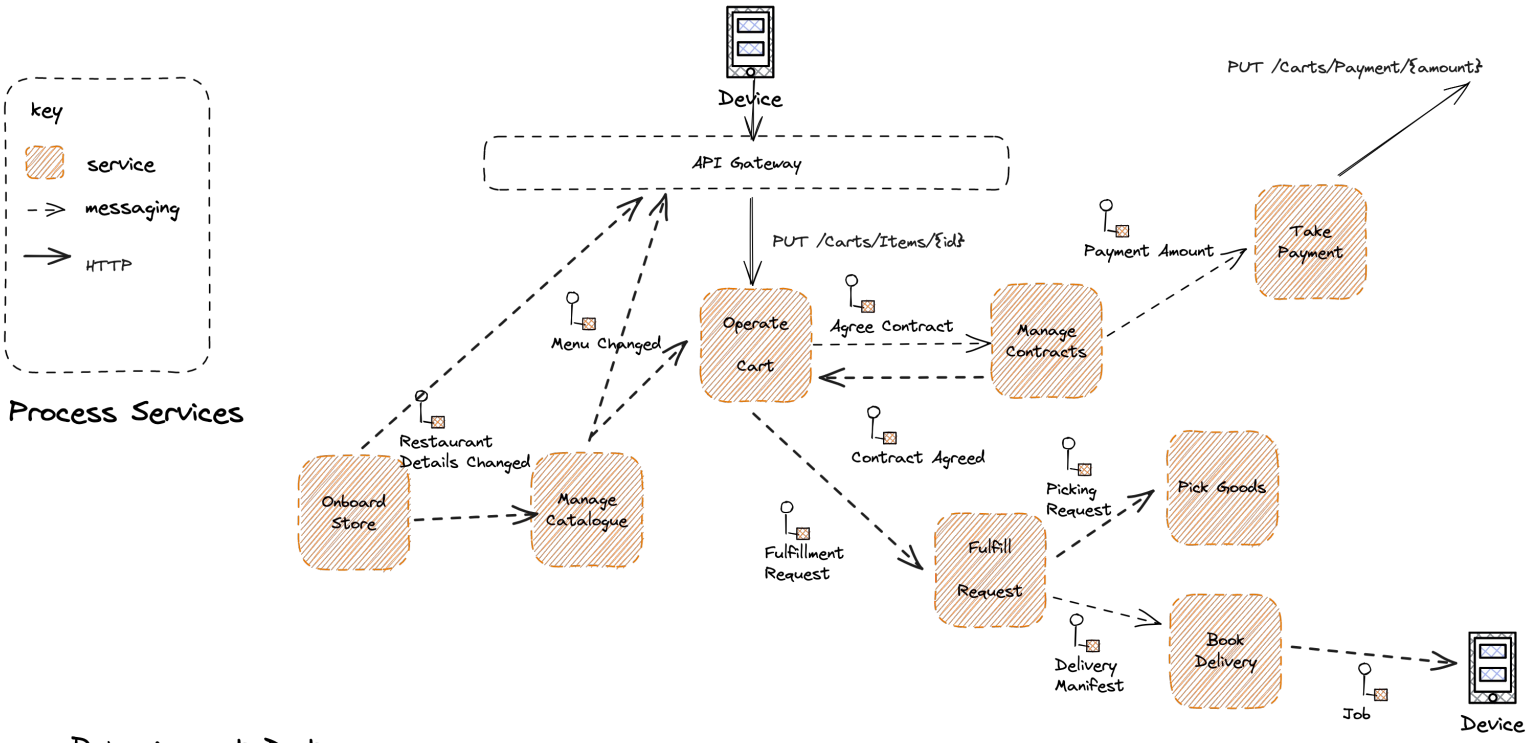
| Turn Taking | Routing Slip | Process Manager (Saga) |
|---|---|---|
| Complex Flow | Simple Flow | Complex Flow |
| Rigid Flow | Dynamic Flow | Dynamic Flow |
| No central point of failure | No central point of failure | Central point of failure |
| Distributed | Distributed | Hub-and-Spoke |
| No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

# EXERCISE MATERIAL

Conversations

- Readme
- Slides

# REACTIVE ARCHITECTURES




## Focus on Behavior not Data

I recommend thinking in terms of the service's responsibilities. (And don't say it's responsible for knowing some data!)
Does it apply policy? Does it aggregate a stream of concepts into a summary?
Does it facilitate some kinds of changes? Does it calculate something? And so on.
Notice how moving through the business process causes previous information to become effectively read-only?

- Michael Nygard

# Paper Workflows

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


Messaging/Discrete Event

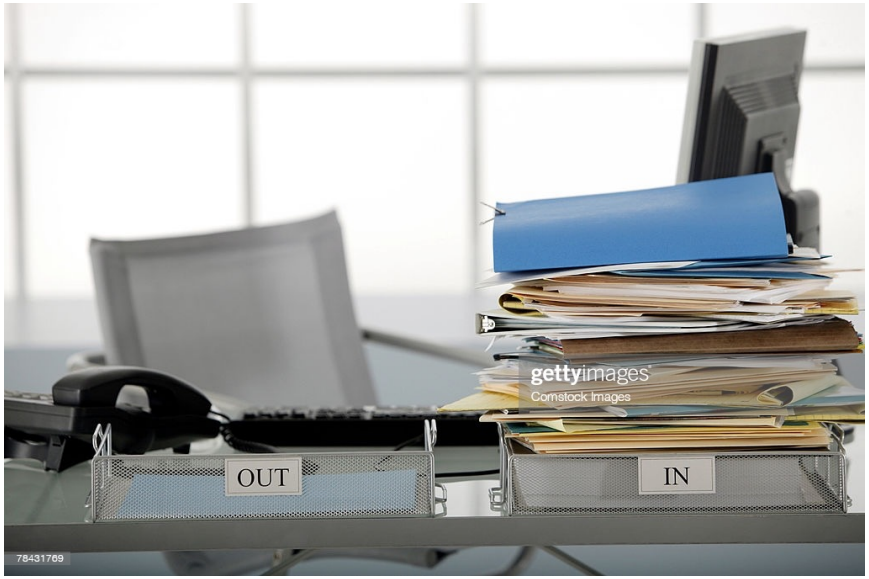
Discrete, Immediately Actioned

Skinny

Series Event (Document)

Series, Supports Action

Fat


OUT
IN
gettyimages
Comstock Images
78431769


Walmart.com
J. Clark
12-5-12
J Rostberg 12/17
J Riley
12/17

OE DIVISION
OEØ1 ROUTING SHEET

TITLE: EID-E15 (23 July 84)

| NAME | REQD | DATE IN | DATE OUT | INITIAL |
|---|---|---|---|---|
| ET1 BARMAN | R | | 8/12/84 | JFB |
| ET2 SWEET | I | | | |
| ET2 BALABUSHKA | A | | | |
| ET2 HURLBUTT | I | | | |
| ET2 JENNINGS | I | | | |
| ET3 MORGAN | I | | | |
| ET3 JONES | A | | | |
| ET3 JACKSON | I | | | |
| ET3 MYERS | I | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

REQD legend; I - Information
A - Action
R - Retain

Upon Completion, Return to ET1 Barman

Destroy
File

OPS/OE Form No. XXXXXXXXX


FAX
brother

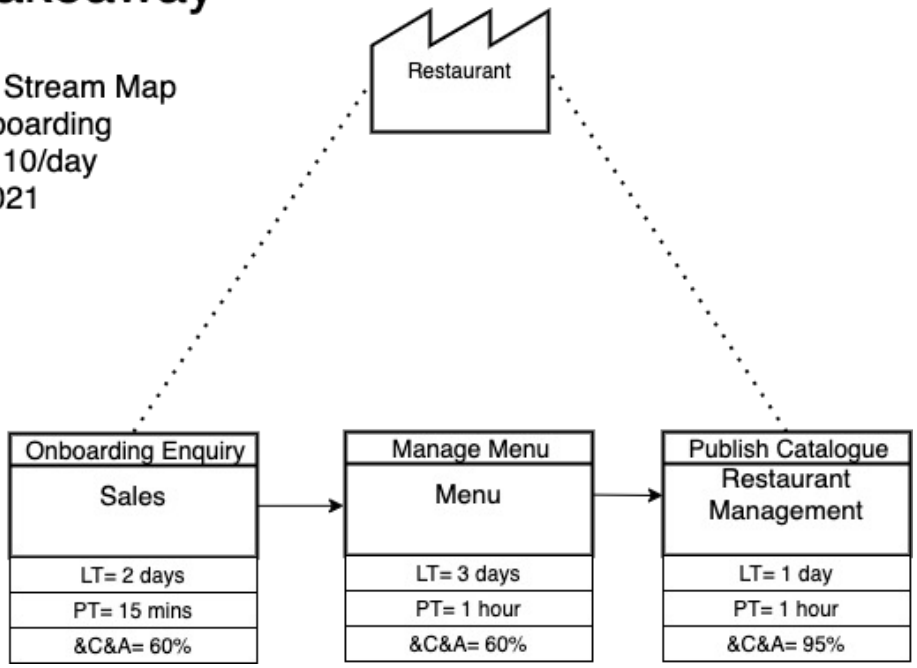
# What is a microservice?

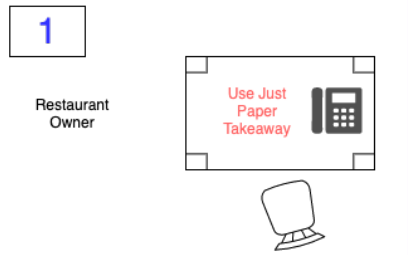
SOA is focused on business *processes*. These *processes* are performed in different steps (also called *activities* or *tasks*) on different systems. The primary goal of a **service** is to represent a "natural" step of business functionality. That is, according to the domain for which it's provided, *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

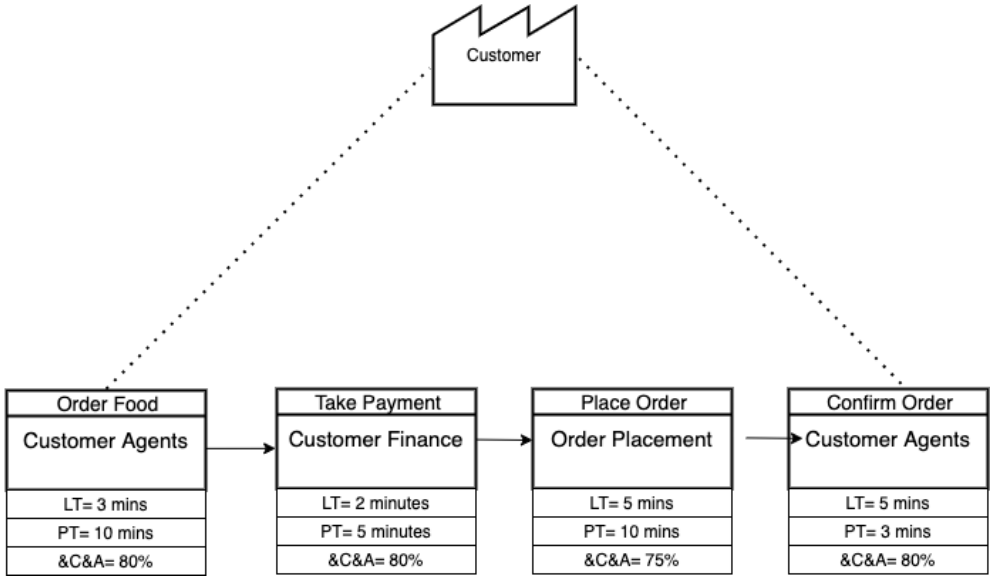


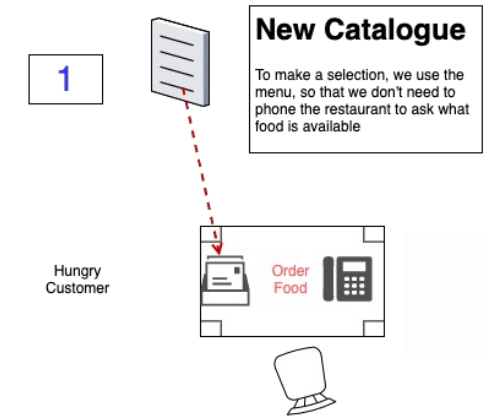



Restaurant Onboarding

# Just Paper Takeaway

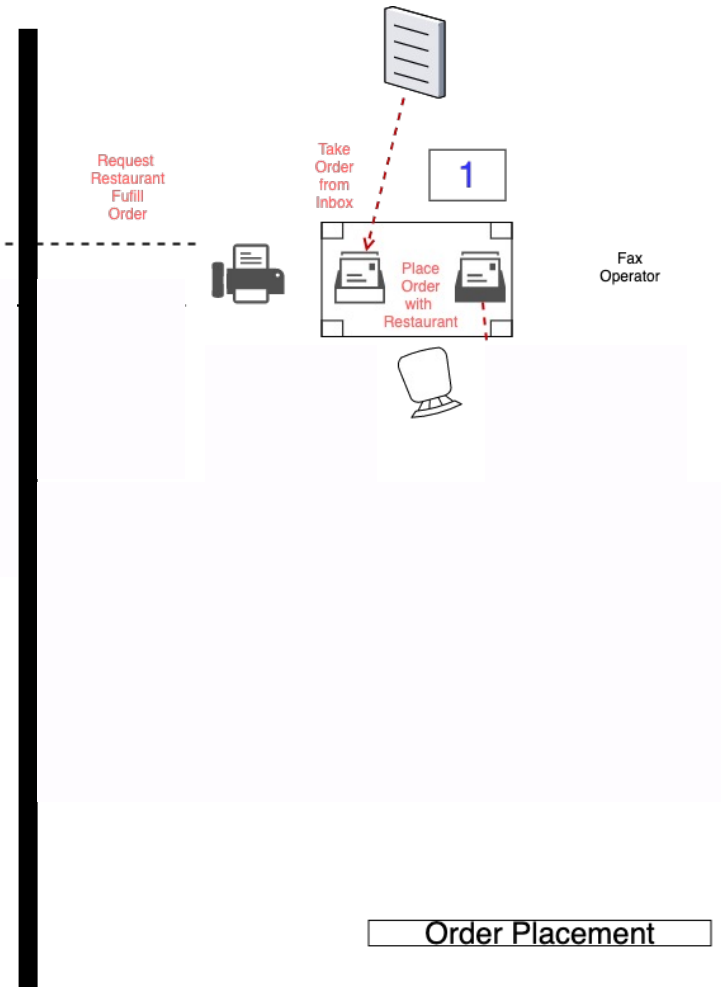
Current State Value Stream Map
Order Flow
Demand Rate 10/day
29 SEP 2021

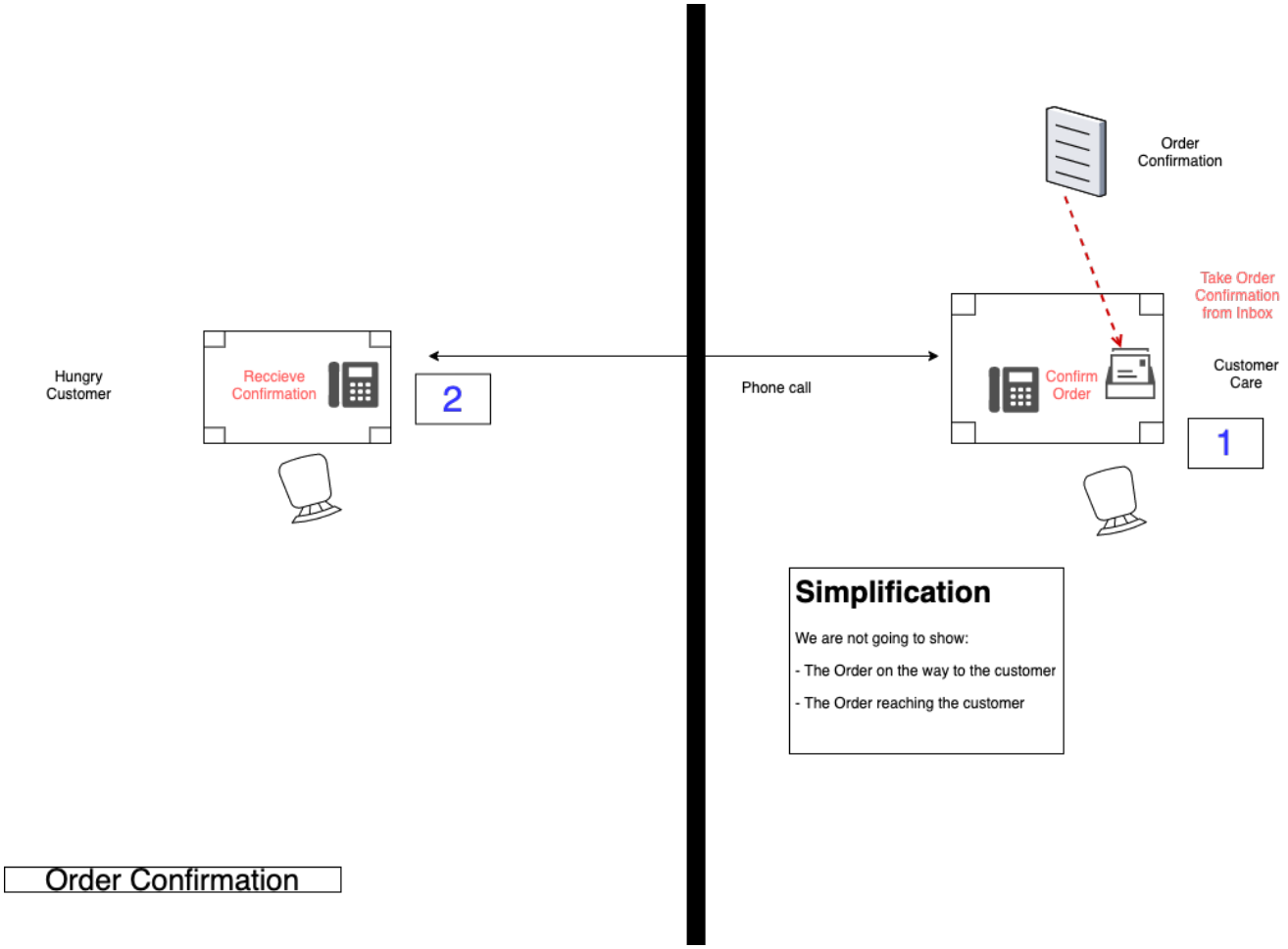


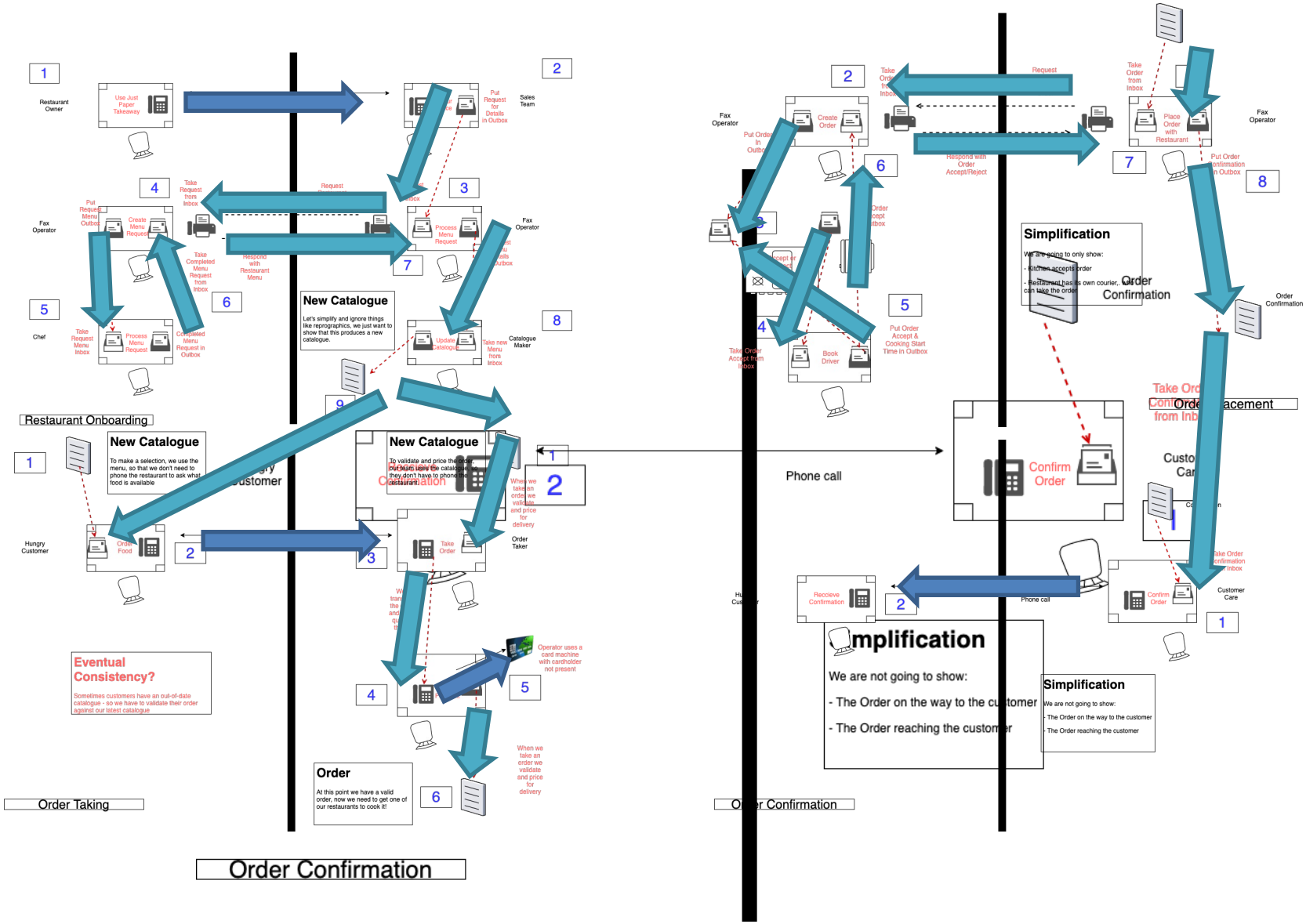



Order Taking

Order Wheel

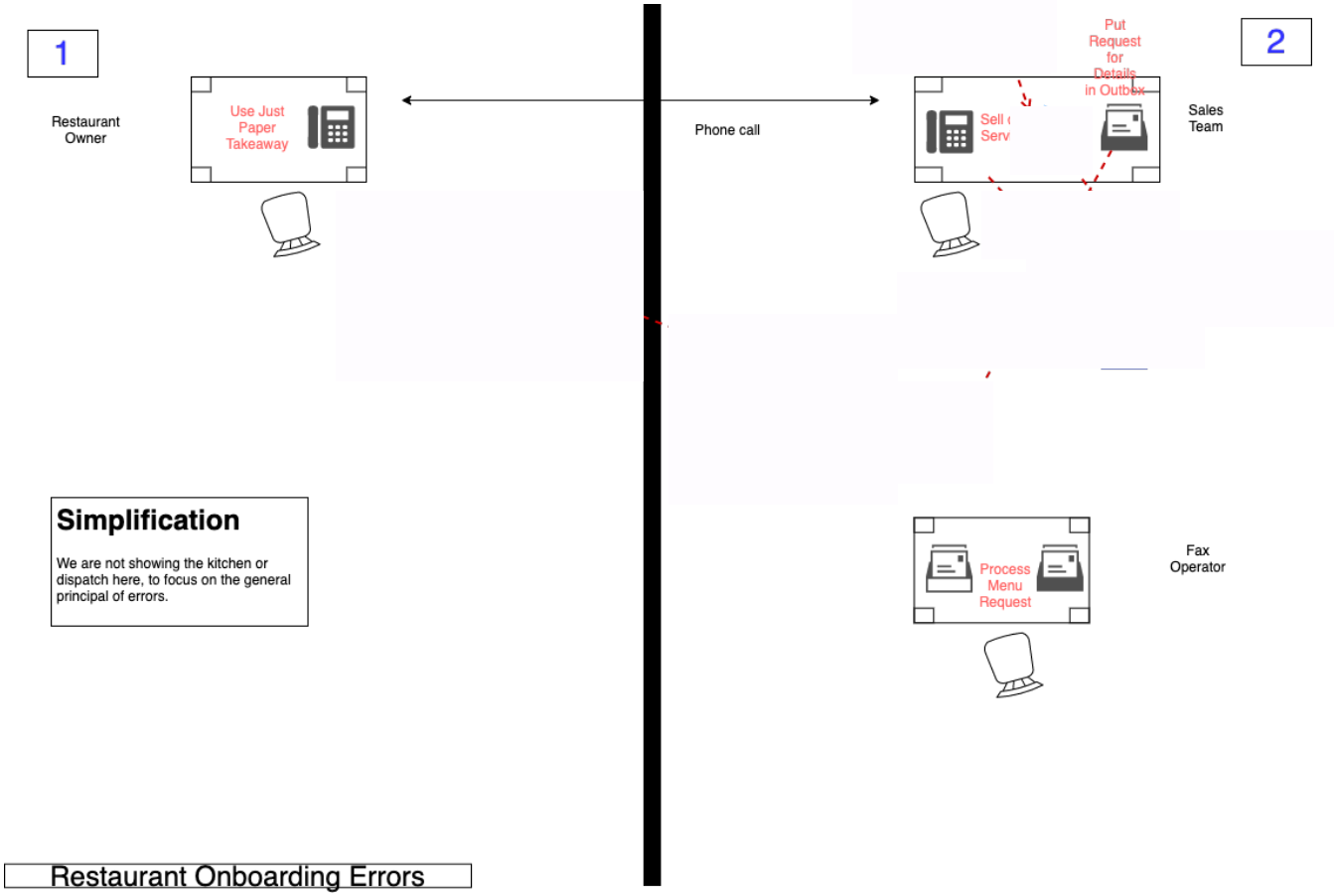

Request Restaurant Fufill Order
Take Order from Inbox
1
Place Order with Restaurant
Fax Operator
Order Placement


Order Confirmation
Take Order Confirmation from Inbox
Hungry Customer
Reccieve Confirmation
2
Phone call
Confirm Order
Customer Care
1
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation


Restaurant Owner
Use Just Paper Takeaway
Sales Team
Put Request for Details in Outbox
Fax Operator
Put Request Menu Outbox
Create Menu Request
Take Request from Inbox
Request
Process Menu Request
Respond with Restaurant Menu
Take Completed Menu Request from Inbox
Chef
Take Request Menu Inbox
Process Menu Request
Completed Menu Request in Outbox
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
Catalogue Maker
Update Catalogue
Take new Menu from Inbox
Restaurant Onboarding
New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
Hungry Customer
Order Food
New Catalogue
To validate and price the order, we take use the catalogue so they don't have to phone the restaurant
Take Order
Order Taker
When we take an order we validate and price for delivery
Operator uses a card machines with cardholder not present
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
When we take an order we validate and price for delivery
Order Taking
Order Confirmation
Phone call
Fax Operator
Create Order
Put Order In Outbox
Request
Take Order from Inbox
Respond with Order Accept/Reject
Place Order with Restaurant
Put Order Confirmation in Outbox
Order Accept Outbox
Accept or Reject
Book Driver
Take Order Accept from Inbox
Put Order Accept & Cooking Start Time in Outbox
Simplification
We are going to only show:
- Kitchen accepts order
- Restaurant has its own courier, who can take the order
Order Confirmation
Take Order Confirmation from Inbox
Confirm Order
Customer Care
Receive Confirmation
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation

# Compensation

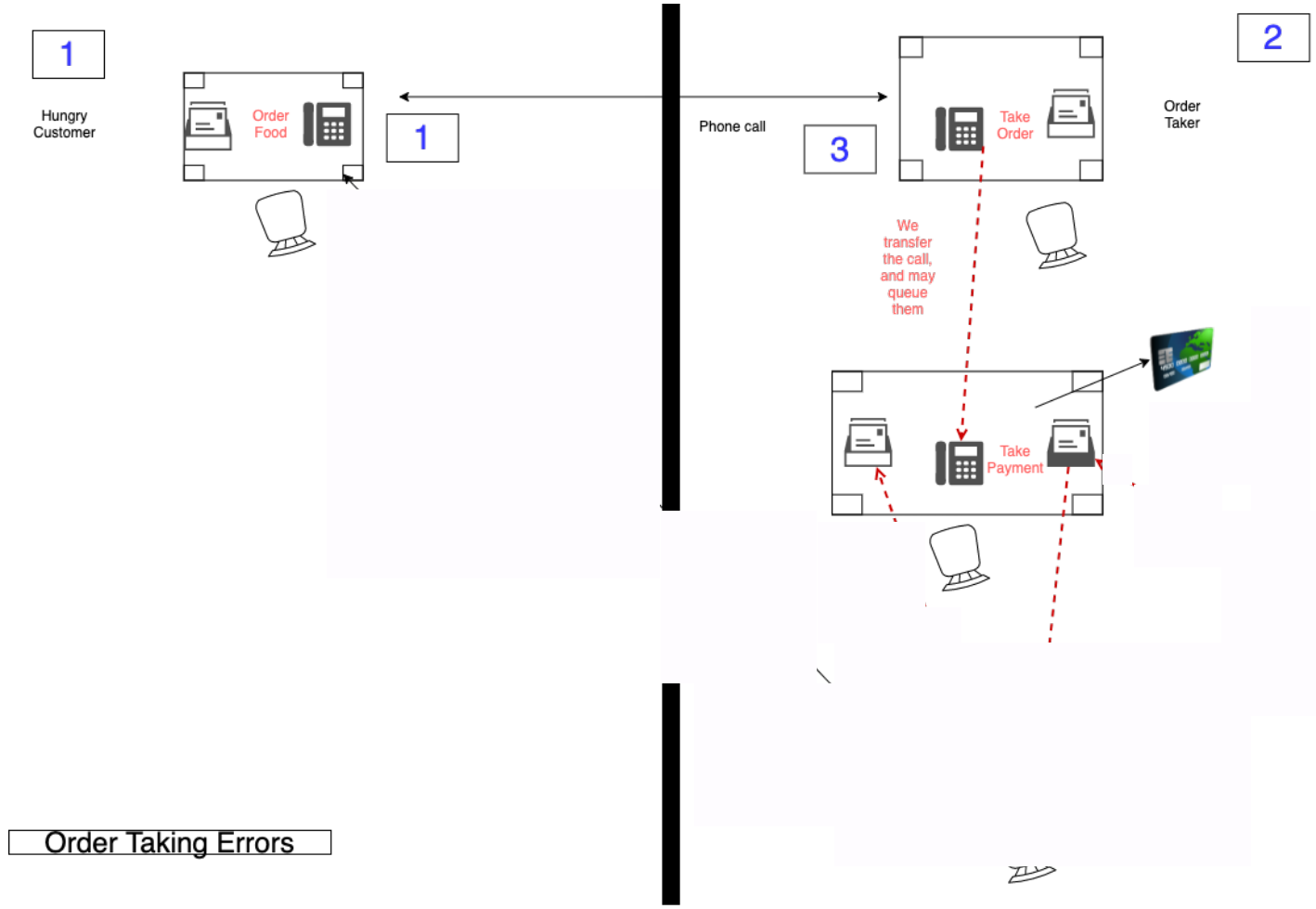

DO NOT USE!
SECTION B - TO BE COMPLETED BY QUALITY
SECTION C - TO BE COMPLETED BY PERSON COMPLETING THE REQUIRED ACTIONS
SECTION D - TO BE COMPLETED BY QUALITY
Scrap Cost:

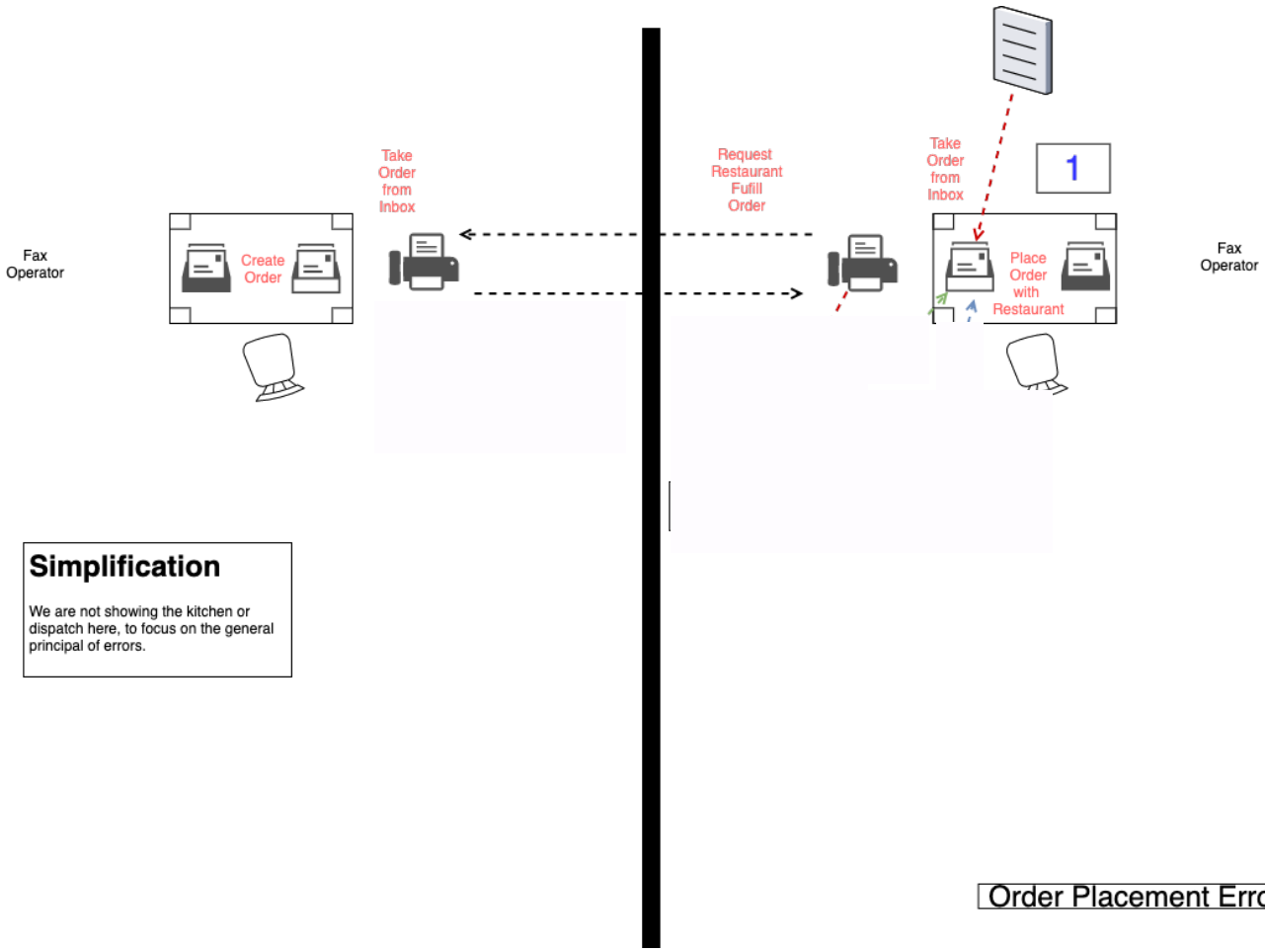

1
Restaurant Owner
Use Just Paper Takeaway
Phone call
Put Request for Details in Outbox
2
Sales Team
Simplification
We are not showing the kitchen or dispatch here, to focus on the general principal of errors.
Process Menu Request
Fax Operator
Restaurant Onboarding Errors

# 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 | [illegible] | 0 | E-705 V.34 1M31 |
| 2010-03-22 | 11:39 | RECV | 59 | 0:20 | 26400 | [illegible] | 1 | OK -- V.34 BM31 |
| 2010-03-22 | 11:45 | RECV | 60 | 0:20 | 26400 | [illegible] | 1 | OK -- V.34 BM31 |
| 2010-03-22 | 12:29 | RECV | 61 | 0:21 | 26400 | [illegible] | 1 | OK -- V.34 BM31 |
| 2010-09-14 | 14:46 | SCAN | 129 | 1:40 | 9600 | [illegible] | 2 | E-606 V.29 AR30 |

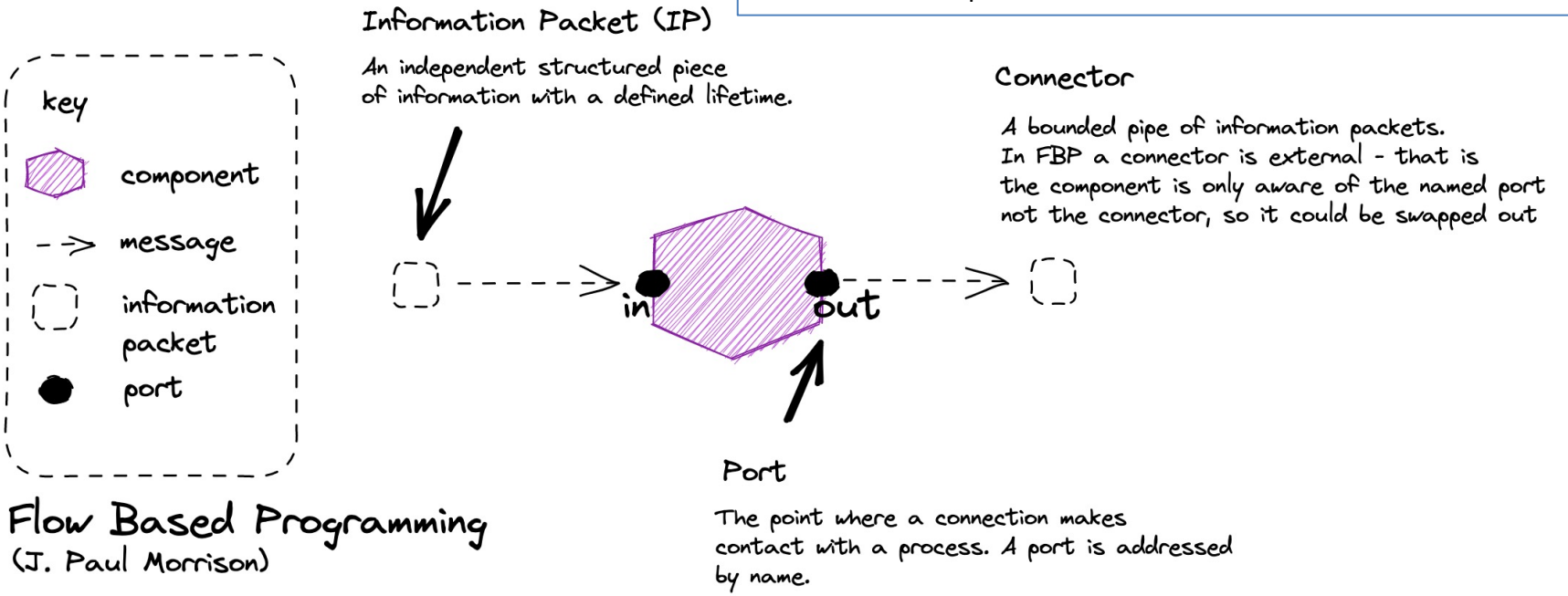

1
Hungry Customer
Order Food
1
Phone call
3
2
Take Order
Order Taker
We transfer the call, and may queue them
Take Payment
Order Taking Errors

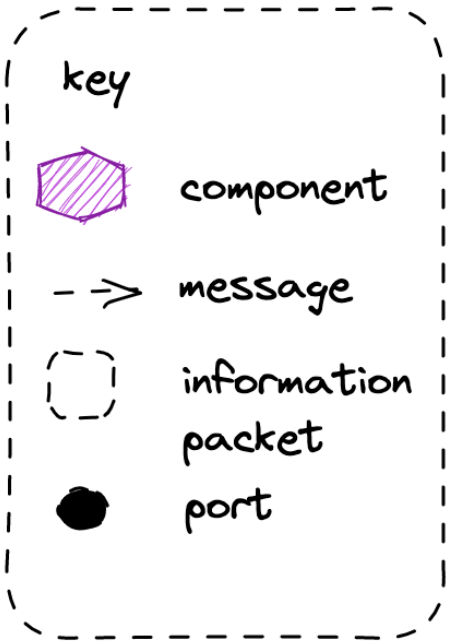

Fax
Operator
Create
Order
Take
Order
from
Inbox
Request
Restaurant
Fufill
Order
Take
Order
from
Inbox
1
Place
Order
with
Restaurant
Fax
Operator
Simplification
We are not showing the kitchen or dispatch here, to focus on the general principal of errors.
Order Placement Errors

# Flow Based Programming

"Everything Flows and Nothing Stays." - Heraclitus

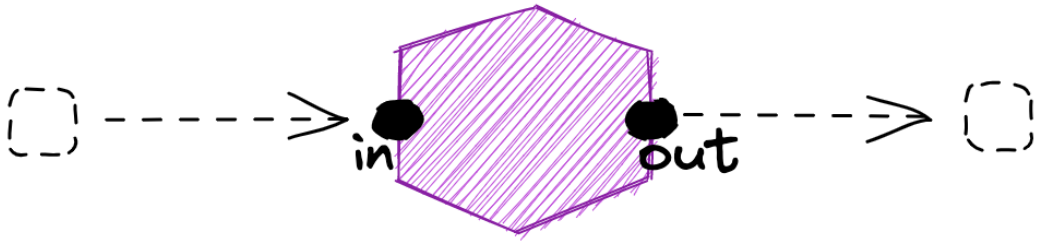
**Flow-Based Programming (FBP)** is a subclass of DFP. While DFP can be synchronous or asynchronous, FBP is always asynchronous. FBP allows multiple input ports, has bounded buffers and applies back pressure when buffers fill up.






Packet Lifetime

Packet Lifetime

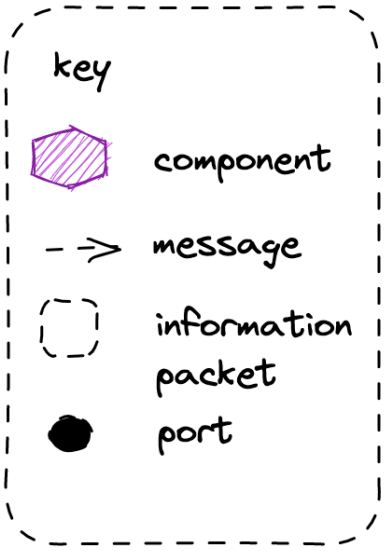



Process & Wait

# Flow Based Programming

(J. Paul Morrison)

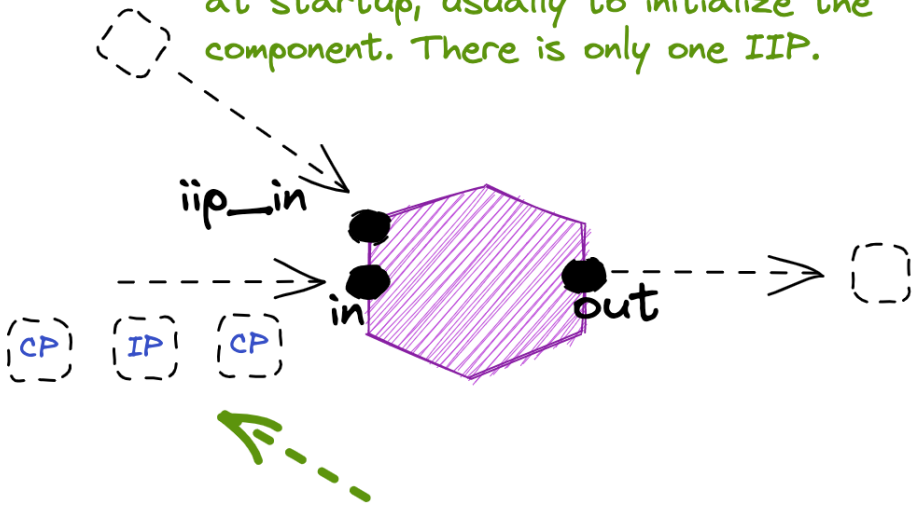
**Node Lifetime**: In flow base programming the a node can remain running whilst there is work on an input queue, can can suspend instead of terminate if there is no work on its connector.




Flow Based Programming
(J. Paul Morrison)

## Initial Information Packet

A packet that the component reads at startup, usually to initialize the component. There is only one IIP.




## Control Packets

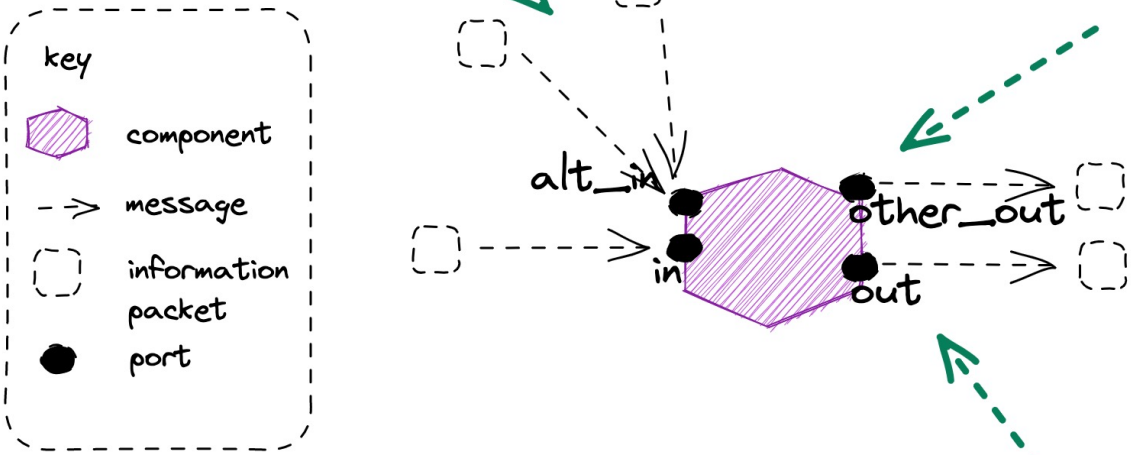
Control Packets can be used to 'bracket' groups

## Multiple Writers

In FBP an in port can have multiple writers

## Single Writer

In FBP an out port is single writer

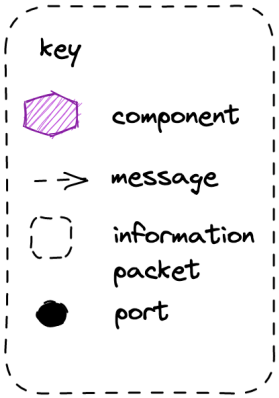



## Multiple In/Out Ports

In FBP we can have multiple in or out ports

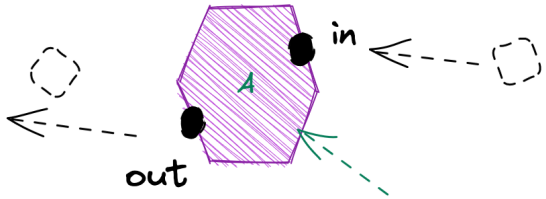
Flow Based Programming
(J. Paul Morrison)

**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.

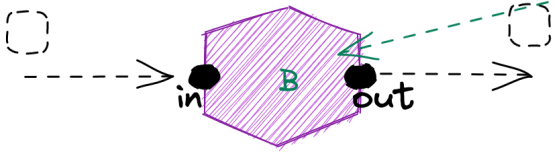



Flow Based Programming

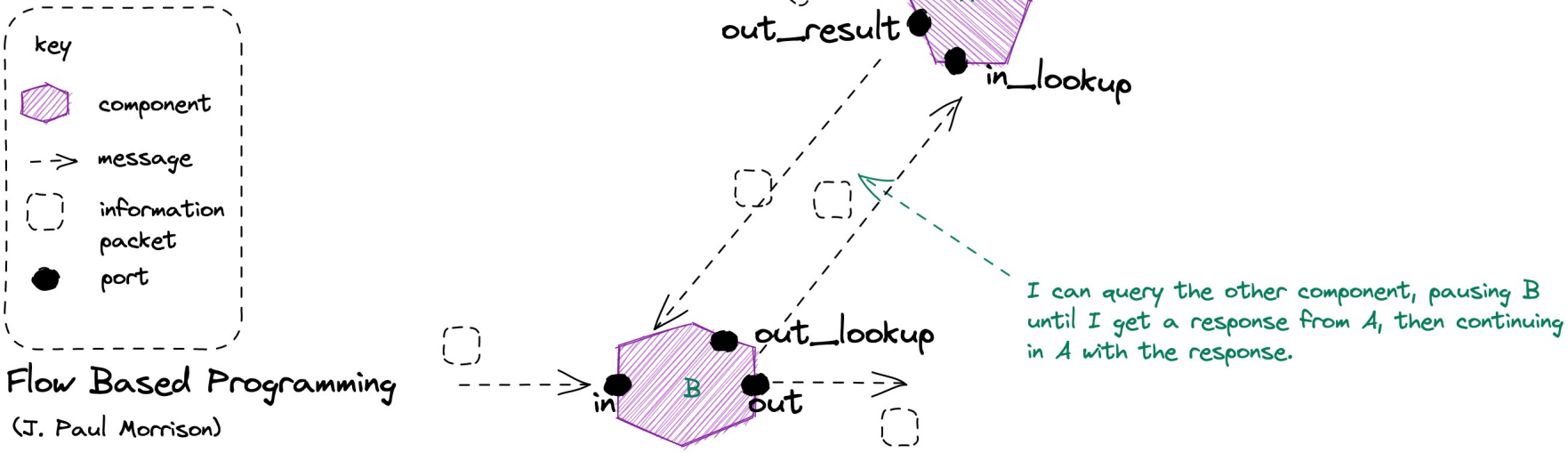
(J. Paul Morrison)

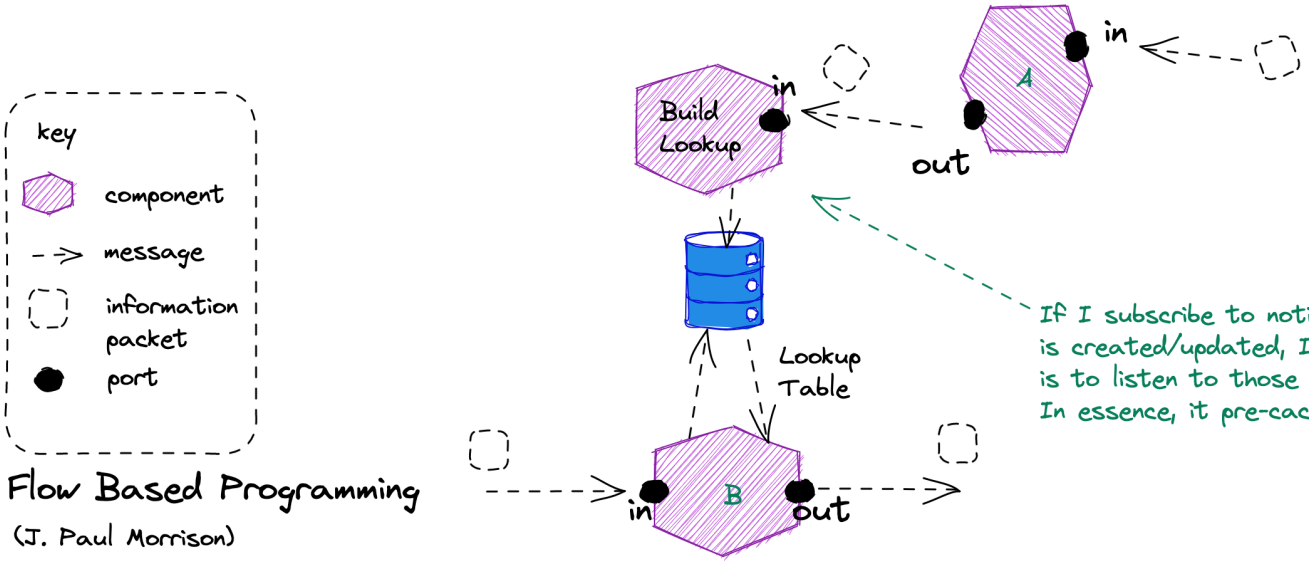



What happens if I need data from one node, to react in another node?



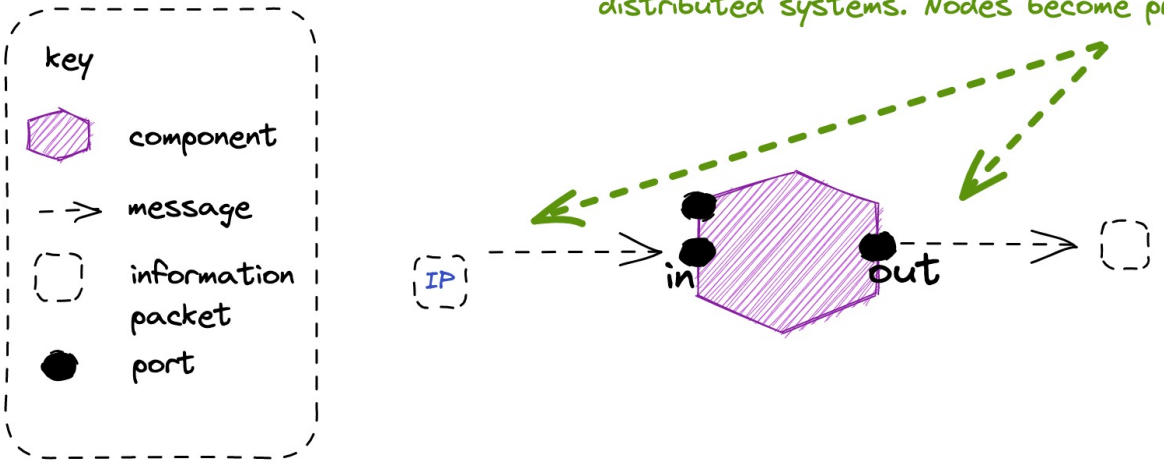
**Walk of Shame**: The trouble is that we may keep asking, so it makes sense to cache this somehow.



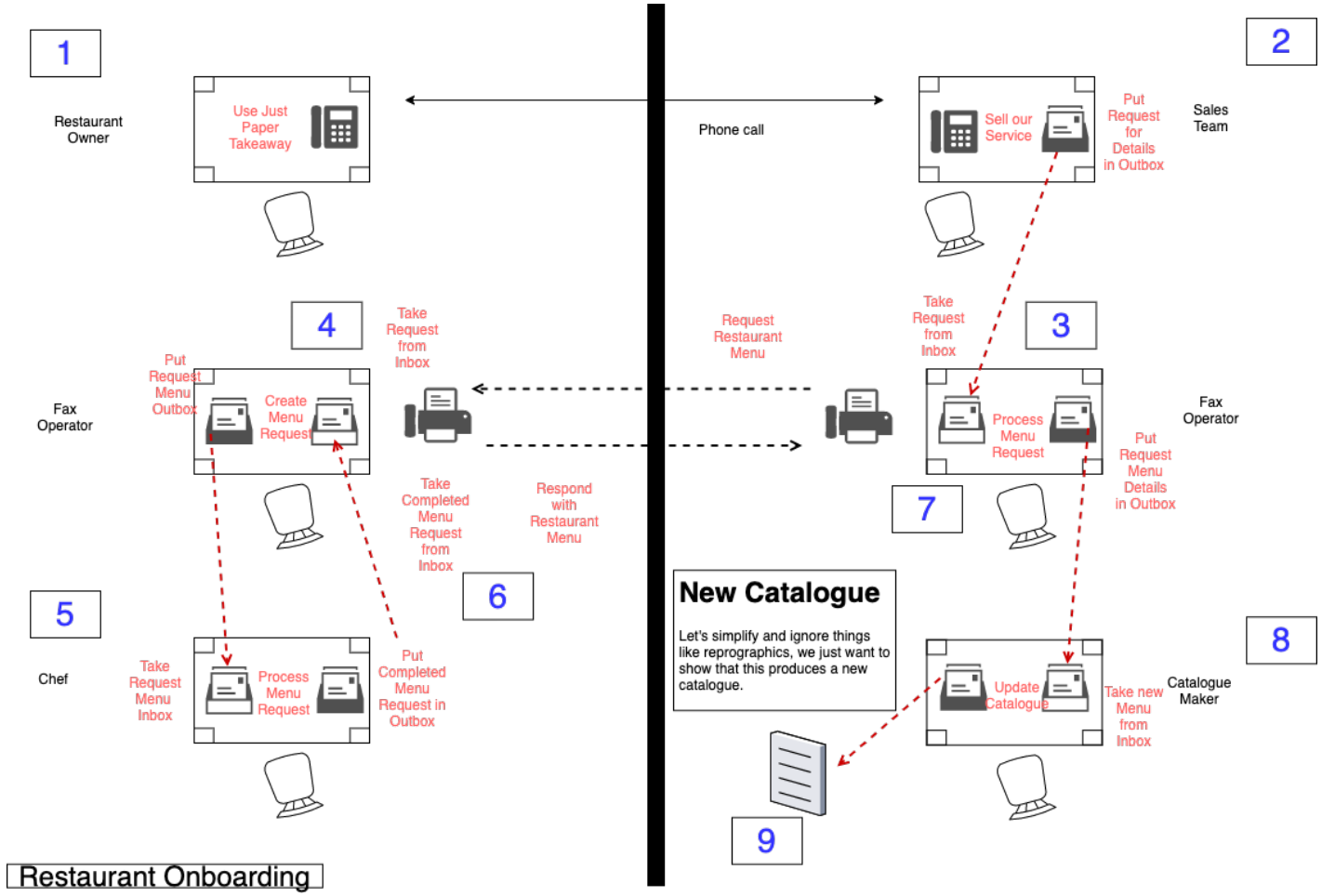

key
component
message
information packet
port
Flow Based Programming
(J. Paul Morrison)
Build Lookup
in
out
A
in
Lookup Table
in
B
out
If I subscribe to notifications from A, as new data is created/updated, I can write a node whose responsibility is to listen to those messages and builds a lookup table for B. In essence, it pre-caches the results for B.

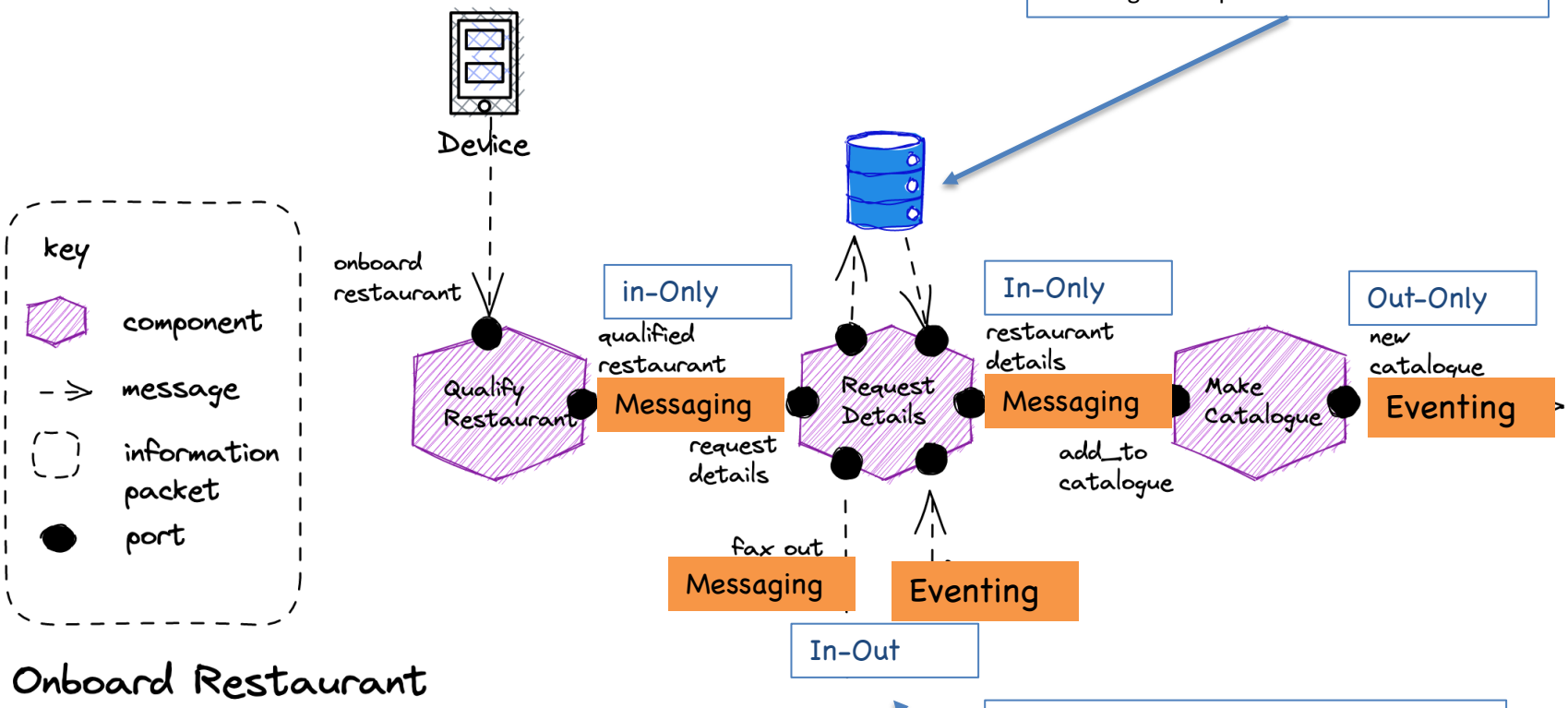
## Message Oriented Middleware (MoM)

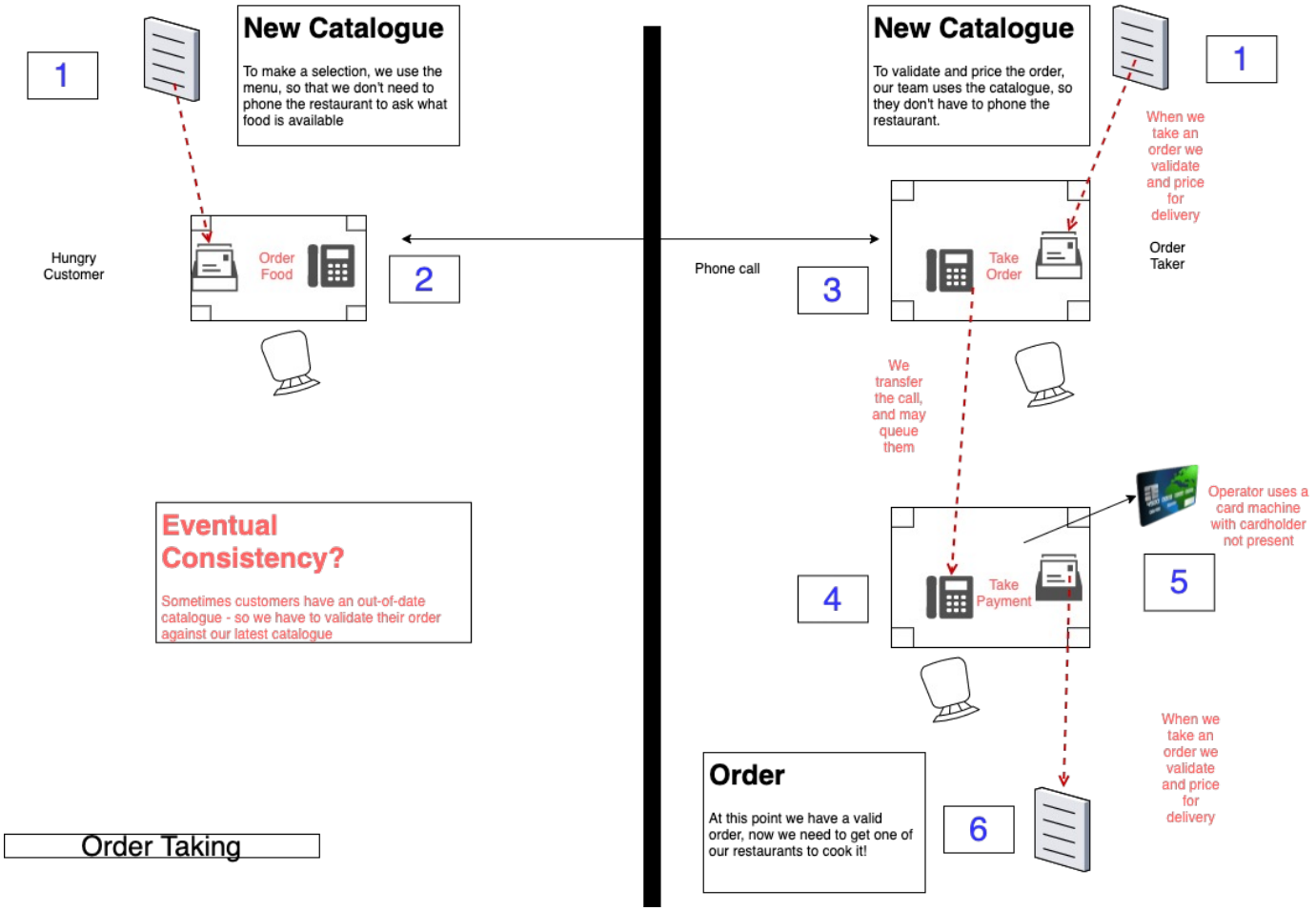
Because FBP allows for externally defined connectors
in the 2010 update JPM outlined that it could be used in
distributed systems. Nodes become processes and connectors use MoM.

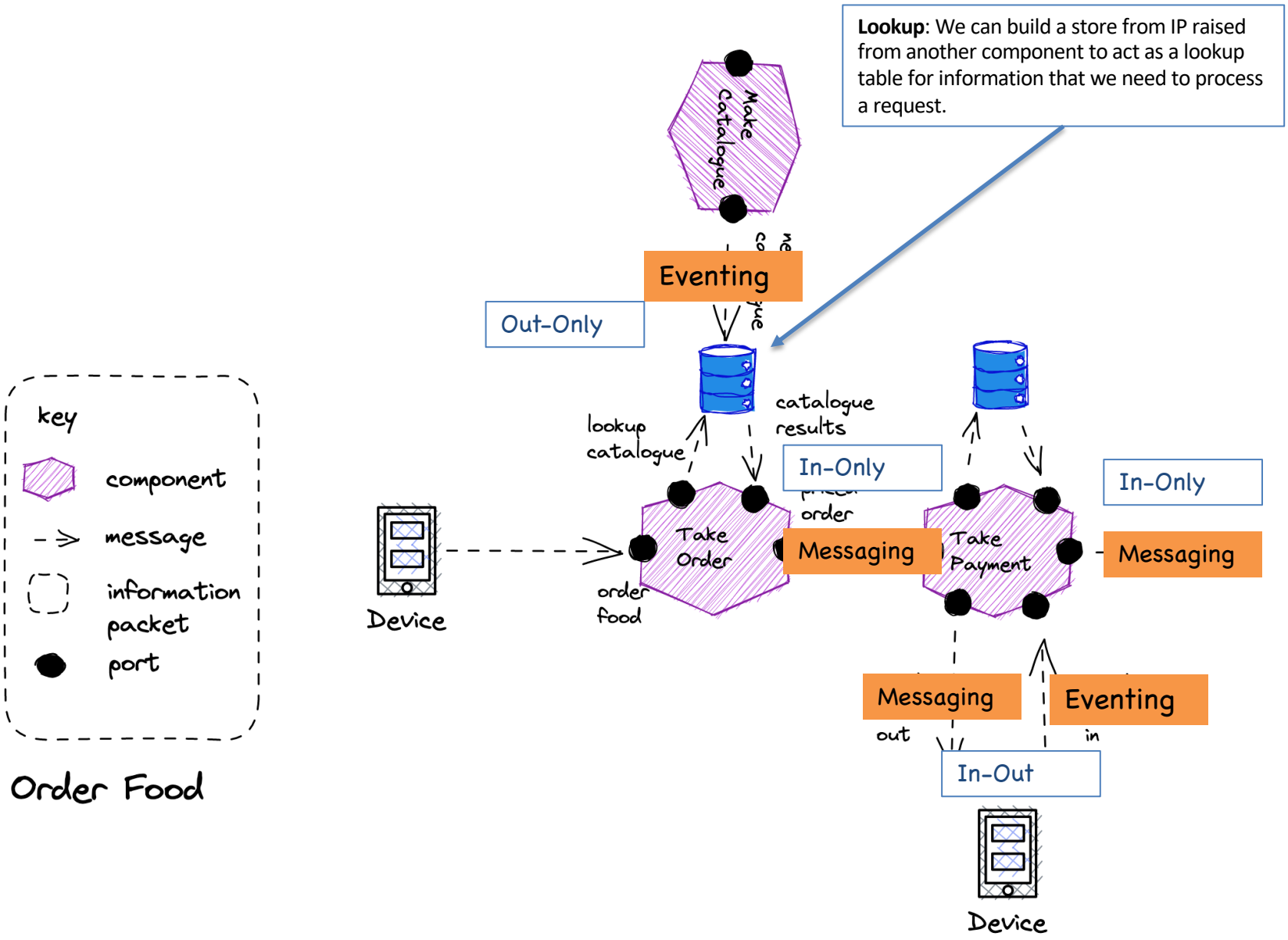


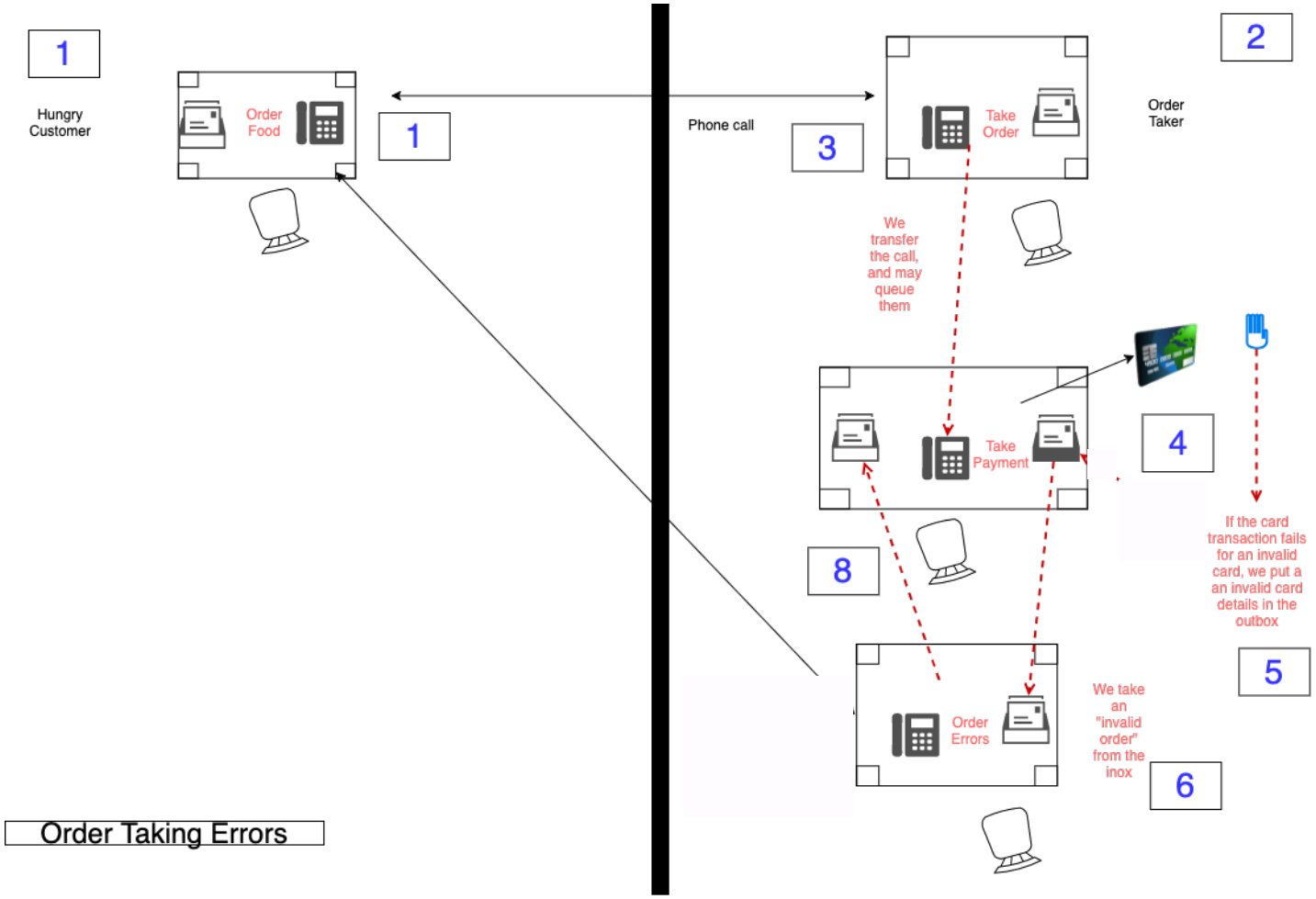

Flow Based Programming - 2010 Update
(J. Paul Morrison)

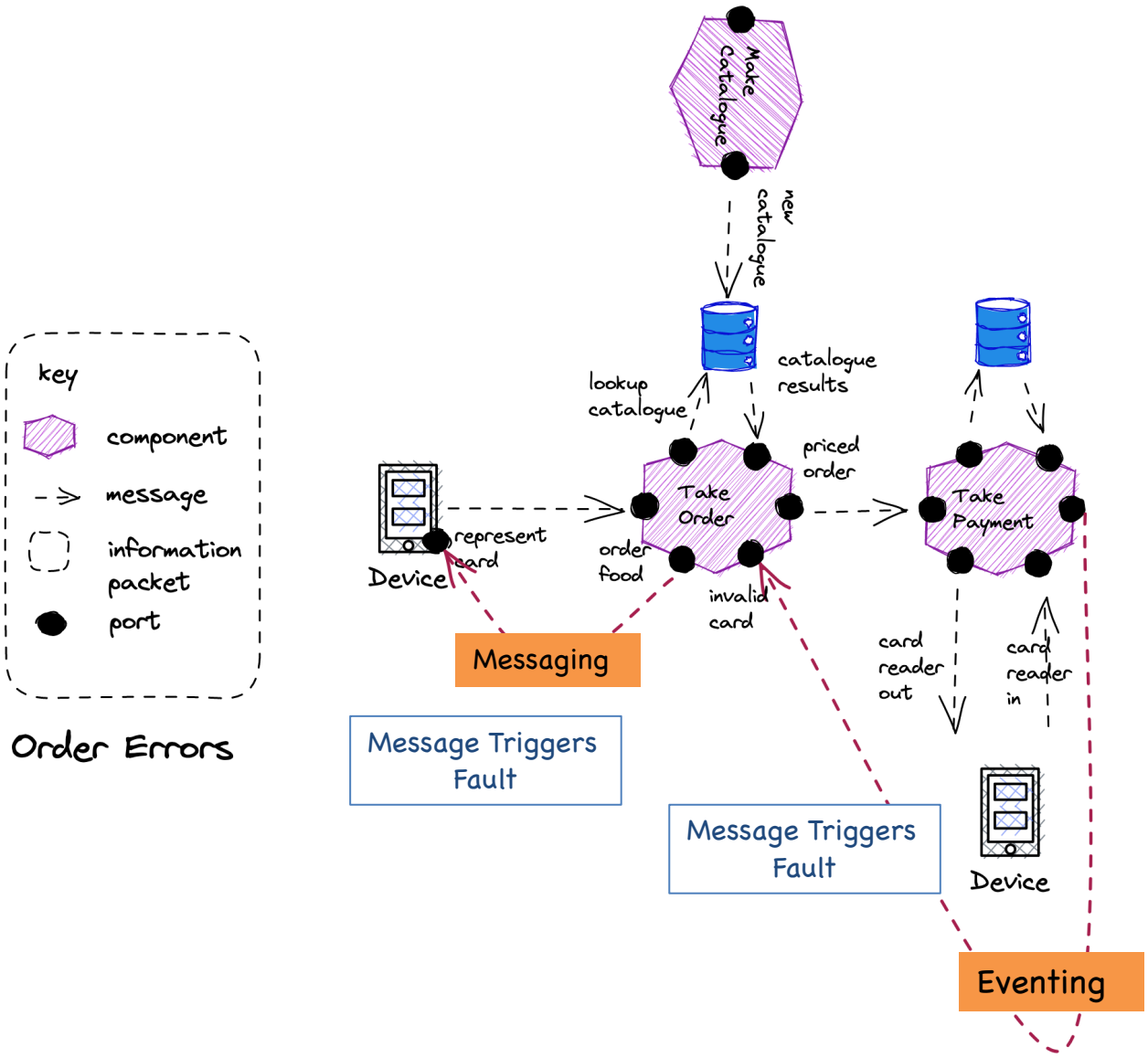

1
Restaurant Owner
Use Just Paper Takeaway
Phone call
2
Sell our Service
Put Request for Details in Outbox
Sales Team
4
Take Request from Inbox
Put Request Menu Outbox
Create Menu Request
Fax Operator
Request Restaurant Menu
Take Request from Inbox
3
Process Menu Request
Fax Operator
Put Request Menu Details in Outbox
7
Take Completed Menu Request from Inbox
Respond with Restaurant Menu
6
5
Chef
Take Request Menu Inbox
Process Menu Request
Put Completed Menu Request in Outbox
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
8
Update Catalogue
Take new Menu from Inbox
Catalogue Maker
9
Restaurant Onboarding

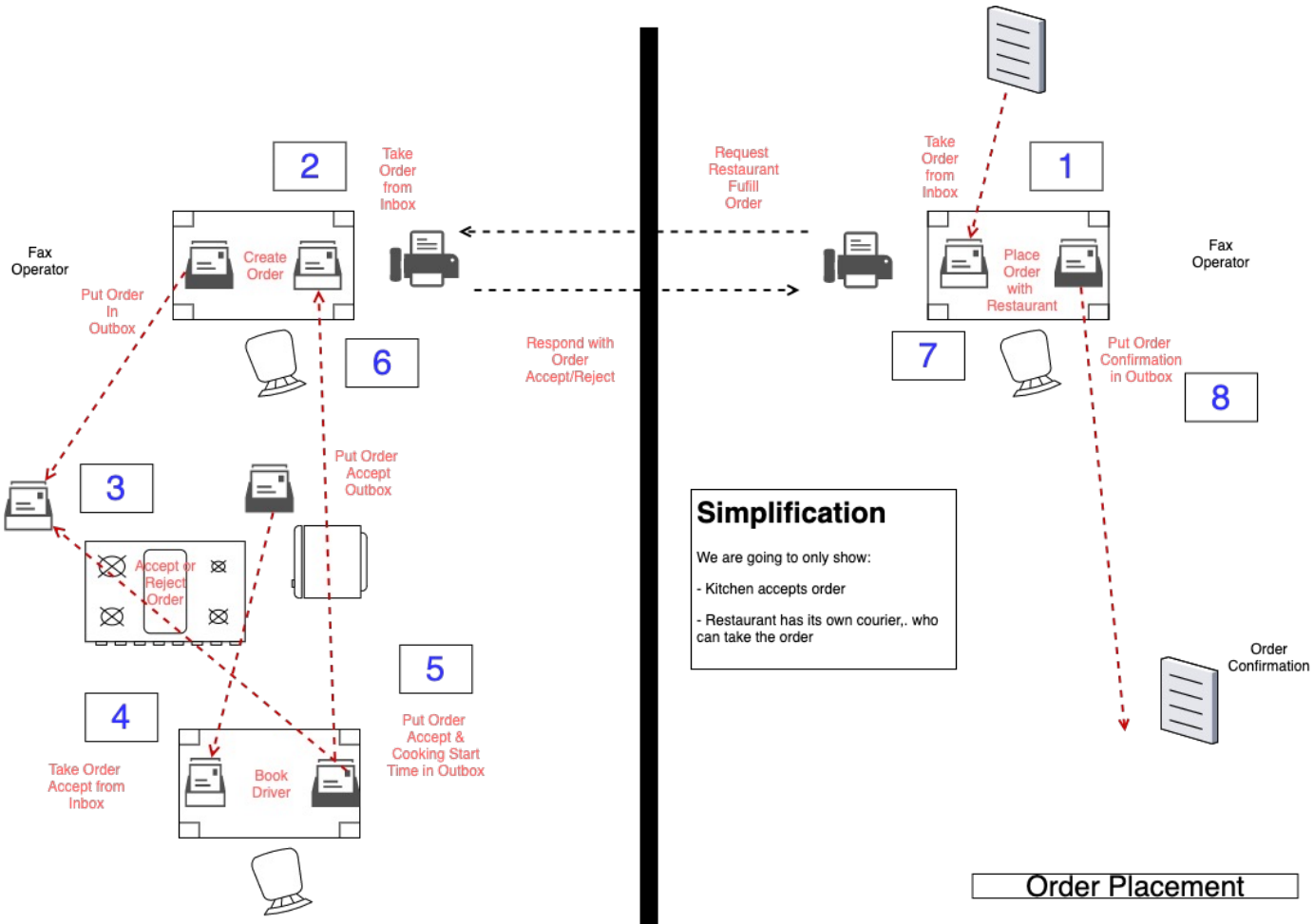

Storage: Request details may need storage to be reliable, it takes work from the request_details port, but cannot forward it to the restaurant_details port without waiting for an answer from fax_in. We store the state of the workflow in case we crash before receiving the response.
Device
key
component
message
information packet
port
onboard restaurant
in-Only
qualified restaurant
Qualify Restaurant
Messaging
request details
Request Details
In-Only
restaurant details
Messaging
add_to catalogue
Make Catalogue
Out-Only
new catalogue
Eventing
fax out
Messaging
Eventing
In-Out
Onboard Restaurant
Correlation: To allow us to match the response to the saved state we use a correlation id. We send it on the ip to fax_out, and the restaurant includes it on the ip from fax_in. We use the ip to look up the workflow we stored.

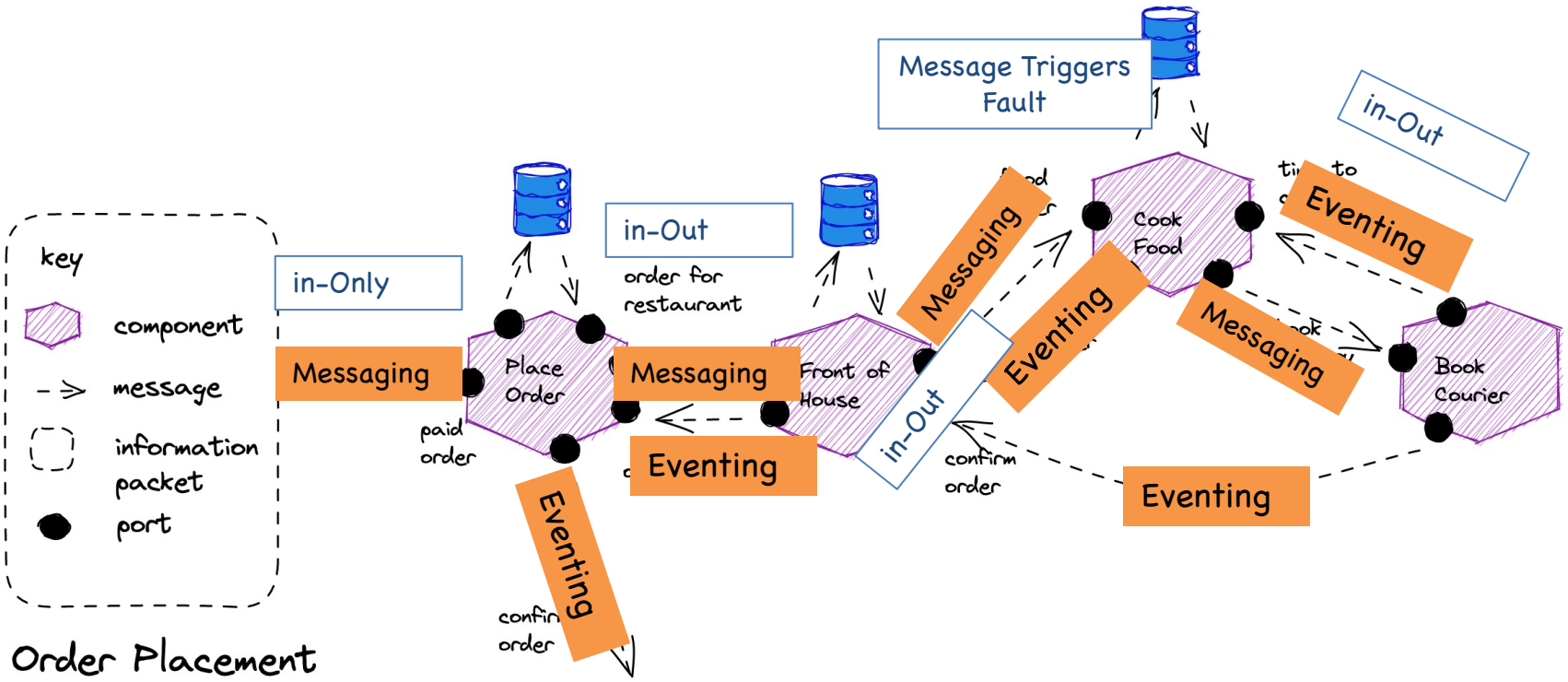

New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
1
Hungry Customer
Order Food
2
Phone call
3
New Catalogue
To validate and price the order, our team uses the catalogue, so they don't have to phone the restaurant.
1
When we take an order we validate and price for delivery
Order Taker
Take Order
We transfer the call, and may queue them
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
4
Take Payment
5
Operator uses a card machine with cardholder not present
When we take an order we validate and price for delivery
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
6
Order Taking

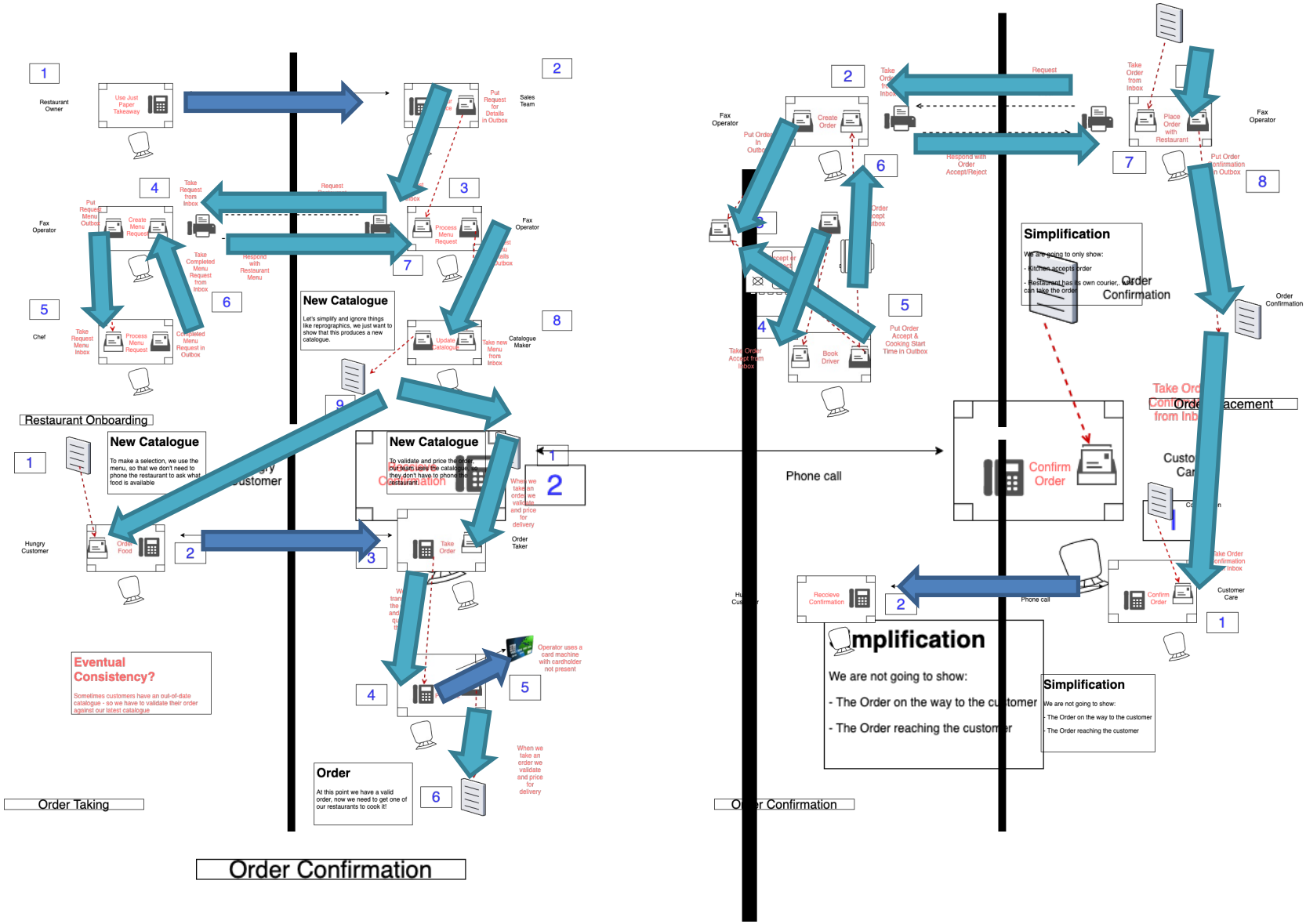

Lookup: We can build a store from IP raised from another component to act as a lookup table for information that we need to process a request.
Make Catalogue
Eventing
Out-Only
catalogue results
lookup catalogue
In-Only
order
Take Order
Messaging
Take Payment
In-Only
Messaging
Device
order food
Messaging
Eventing
out
in
In-Out
Device
key
component
message
information packet
port
Order Food

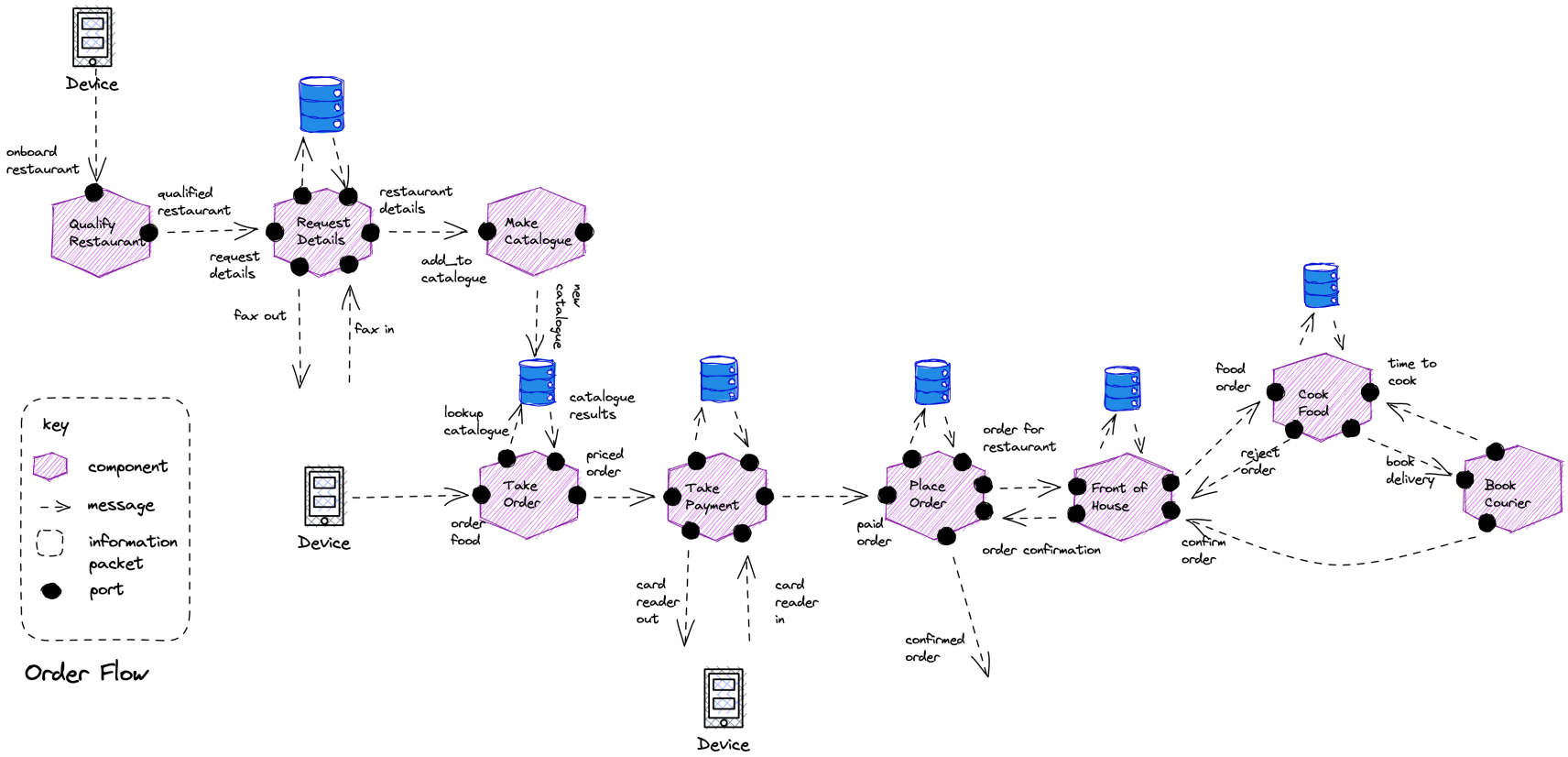

1
Hungry Customer
Order Food
1
Phone call
3
2
Take Order
Order Taker
We transfer the call, and may queue them
Take Payment
4
If the card transaction fails for an invalid card, we put a an invalid card details in the outbox
5
8
Order Errors
We take an "invalid order" from the inox
6
Order Taking Errors


key
component
message
information packet
port
Order Errors
Make Catalogue
new catalogue
lookup catalogue
catalogue results
priced order
Take Order
Take Payment
represent card
Device
order food
invalid card
card reader out
card reader in
Device
Messaging
Message Triggers Fault
Message Triggers Fault
Eventing


2
Take Order from Inbox
Request Restaurant Fufill Order
Take Order from Inbox
1
Fax Operator
Create Order
Place Order with Restaurant
Fax Operator
Put Order In Outbox
6
Respond with Order Accept/Reject
7
Put Order Confirmation in Outbox
8
Put Order Accept Outbox
3
Accept or Reject Order
Simplification
We are going to only show:
- Kitchen accepts order
- Restaurant has its own courier,. who can take the order
Order Confirmation
5
4
Put Order Accept & Cooking Start Time in Outbox
Take Order Accept from Inbox
Book Driver
Order Placement


key
component
message
information packet
port
in-Only
Messaging
Place Order
paid order
Eventing
confir order
in-Out
order for restaurant
Messaging
Front of House
Eventing
Message Triggers Fault
Messaging
in-Out
confirm order
Eventing
Cook Food
Eventing
in-Out
Messaging
Book Courier
Order Placement


Restaurant Owner
Use Just Paper Takeaway
Sales Team
Put Request for Details in Outbox
Fax Operator
Put Request Menu Outbox
Create Menu Request
Take Request from Inbox
Request
Process Menu Request
Respond with Restaurant Menu
Take Completed Menu Request from Inbox
Chef
Take Request Menu Inbox
Process Menu Request
Completed Menu Request in Outbox
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
Catalogue Maker
Update Catalogue
Take new Menu from Inbox
Restaurant Onboarding
New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
Hungry Customer
Order Food
New Catalogue
To validate and price the order, we take use the catalogue so they don't have to phone the restaurant
Take Order
Order Taker
When we take an order we validate and price for delivery
Operator uses a card machines with cardholder not present
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
When we take an order we validate and price for delivery
Order Taking
Order Confirmation
Phone call
Fax Operator
Create Order
Put Order In Outbox
Request
Take Order from Inbox
Respond with Order Accept/Reject
Place Order with Restaurant
Put Order Confirmation in Outbox
Order Accept Outbox
Accept or Reject
Book Driver
Take Order Accept from Inbox
Put Order Accept & Cooking Start Time in Outbox
Simplification
We are going to only show:
- Kitchen accepts order
- Restaurant has its own courier, who can take the order
Order Confirmation
Take Order Confirmation from Inbox
Confirm Order
Customer Care
Receive Confirmation
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation


Device
onboard restaurant
Qualify Restaurant
qualified restaurant
Request Details
request details
restaurant details
Make Catalogue
add_to catalogue
new catalogue
fax out
fax in
lookup catalogue
catalogue results
Take Order
order food
priced order
Device
Take Payment
card reader out
card reader in
Device
Place Order
paid order
confirmed order
order for restaurant
order confirmation
Front of House
food order
reject order
Cook Food
time to cook
book delivery
Book Courier
confirm order
key
component
message
information packet
port
Order Flow

# EXERCISE MATERIAL

Paper Flow

- Readme
- Slides

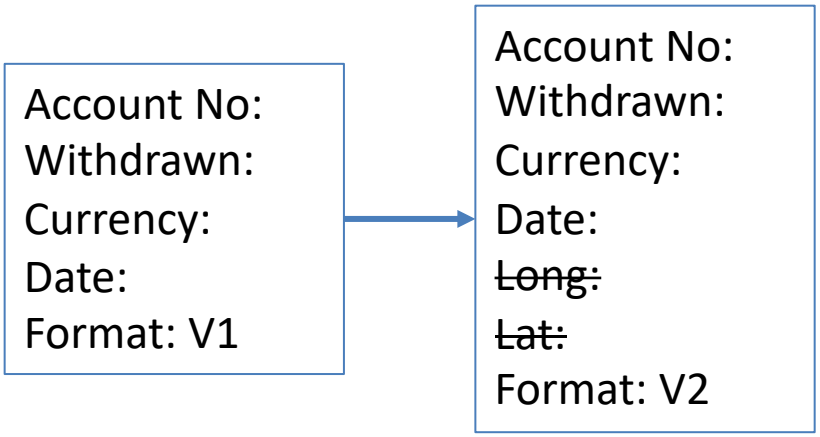
# MANAGING ASYNCHRONOUS APIS

# 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.

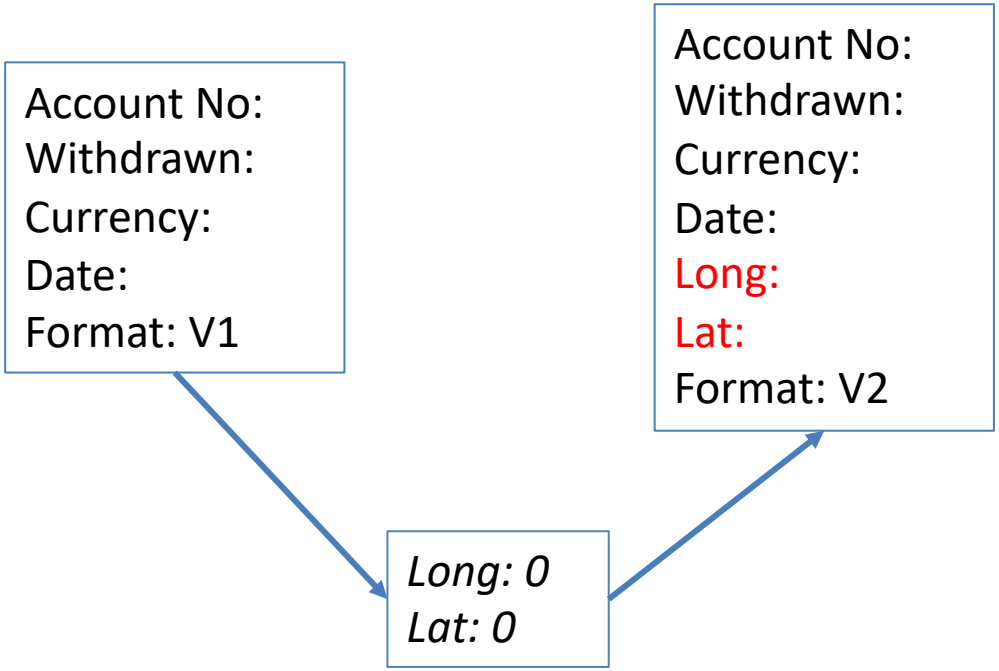
Robustness Principal or Postel's Law – Jon Postel RFC 1958

# Tolerant Reader



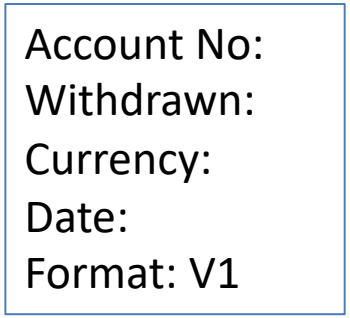

# Ignore New Fields

# Tolerant Reader

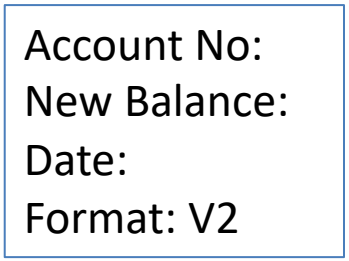



# Default Missing Fields

# Breaking Change




Account.Withdrawal.Event




Account.NewBalance.Event

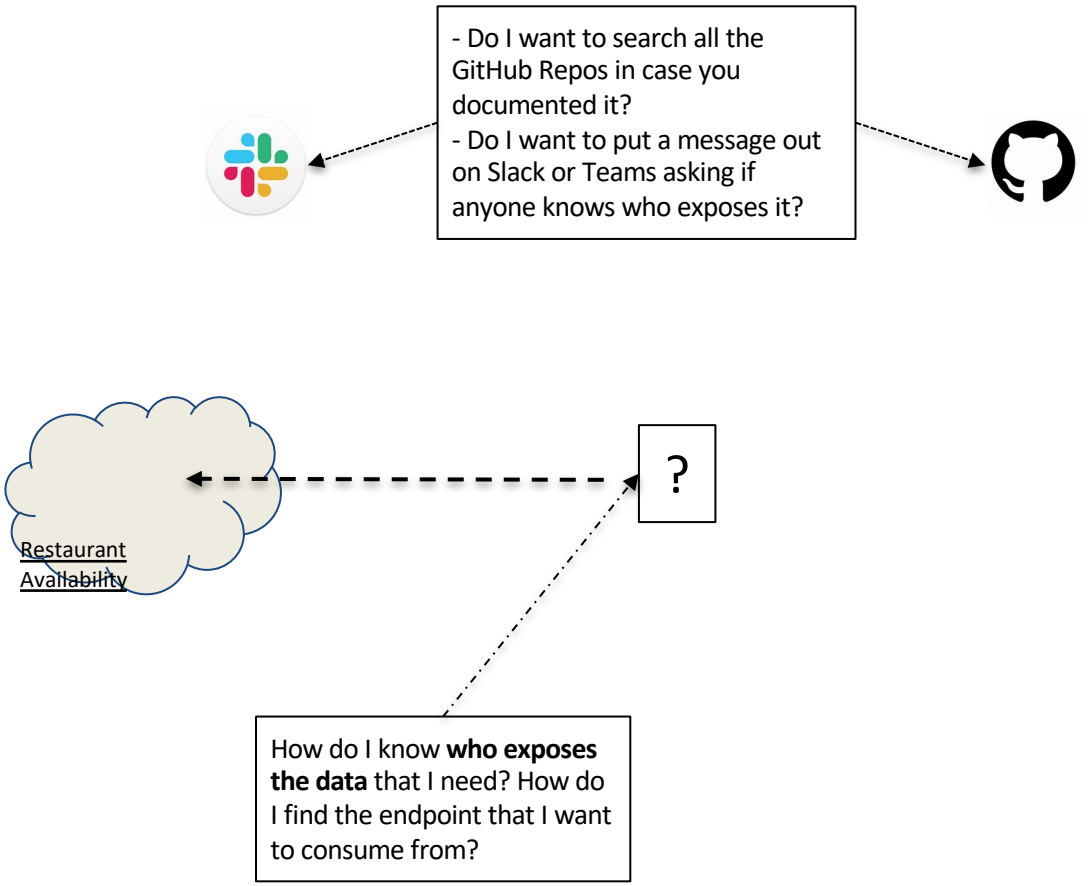
# New Message

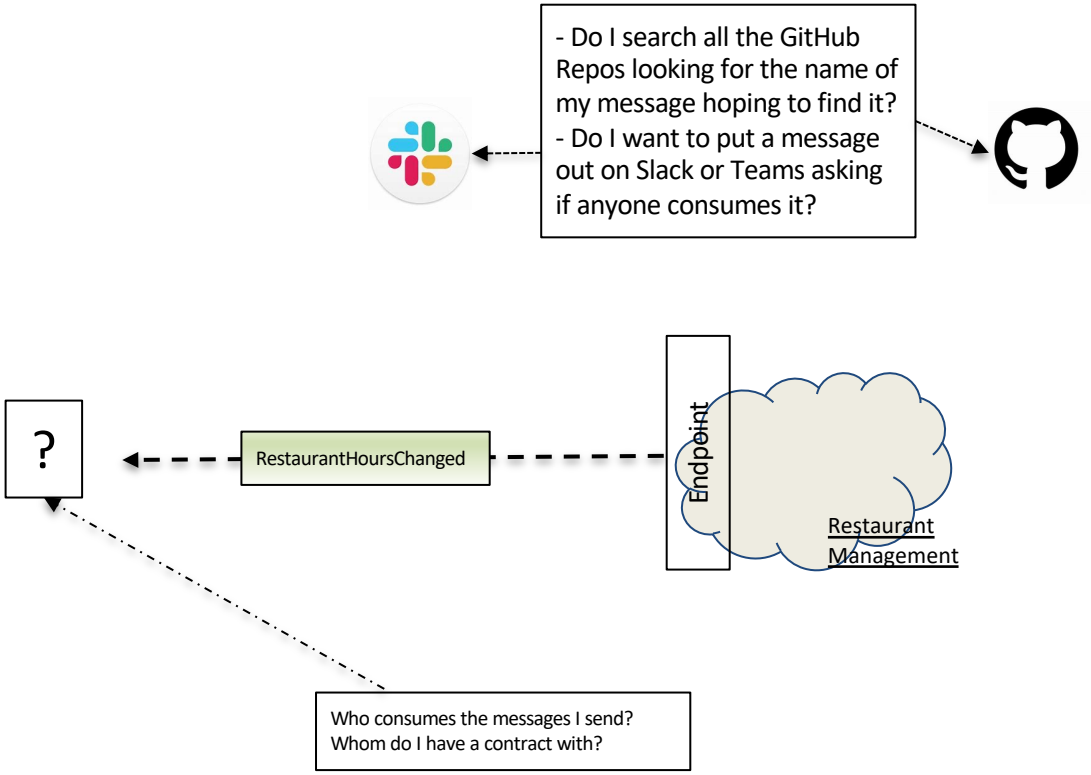
# Documentation

*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

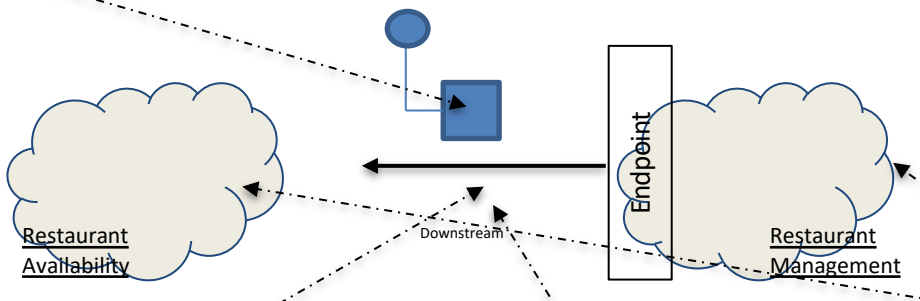

- Do I want to search all the GitHub Repos in case you documented it?
- Do I want to put a message out on Slack or Teams asking if anyone knows who exposes it?
Restaurant Availability
?
How do I know **who exposes the data** that I need? How do I find the endpoint that I want to consume from?


- 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

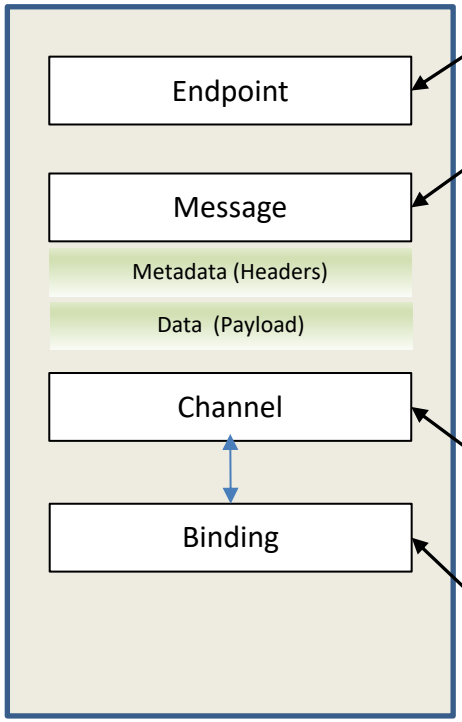



We need to document who sends and receives to understand flow

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

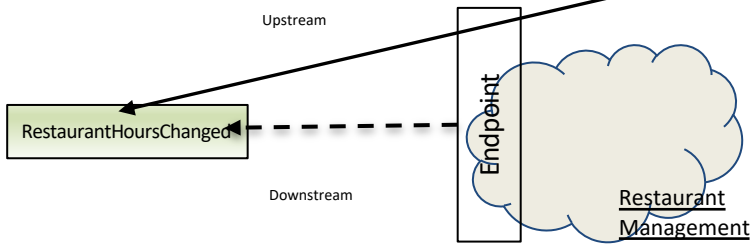
## Endpoint – Documenting the Contract




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.

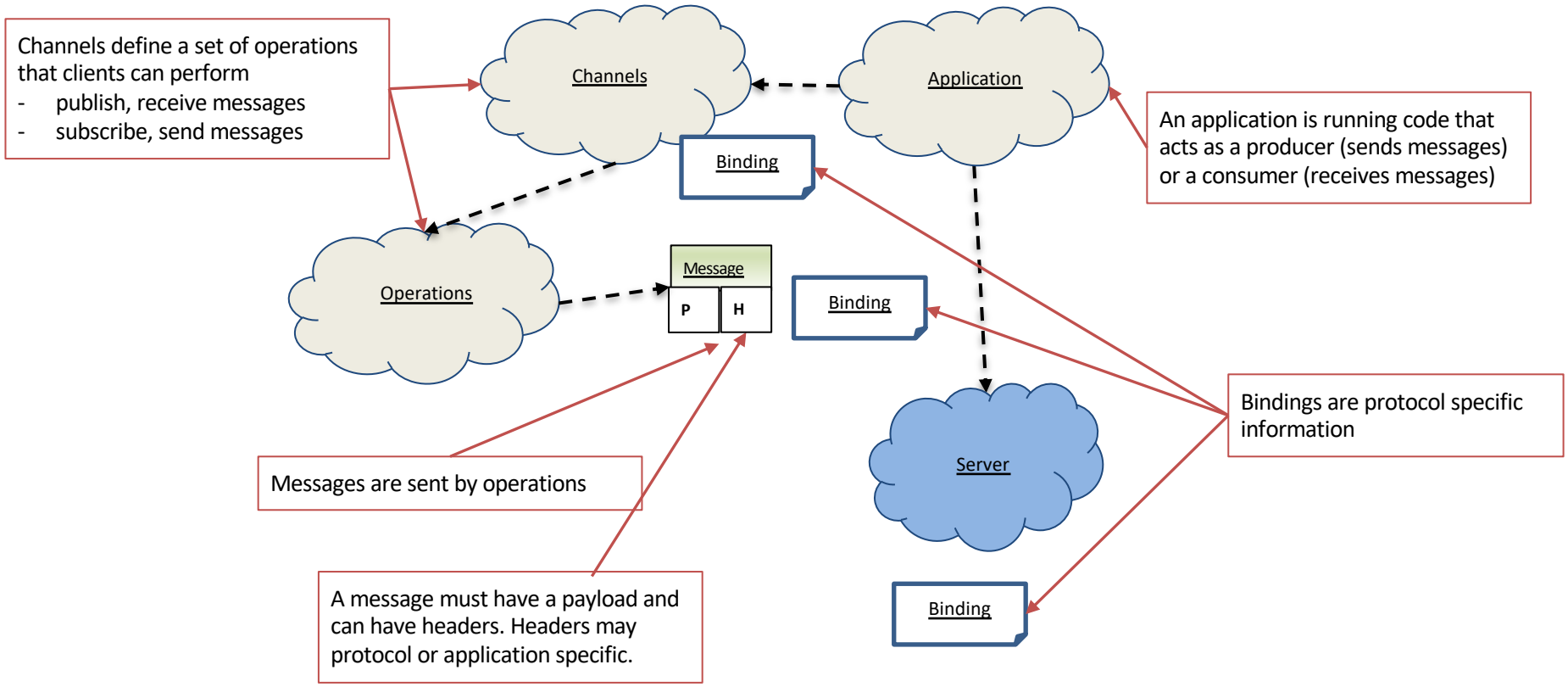
Read more about Open Governance TSC Members

### And much more...

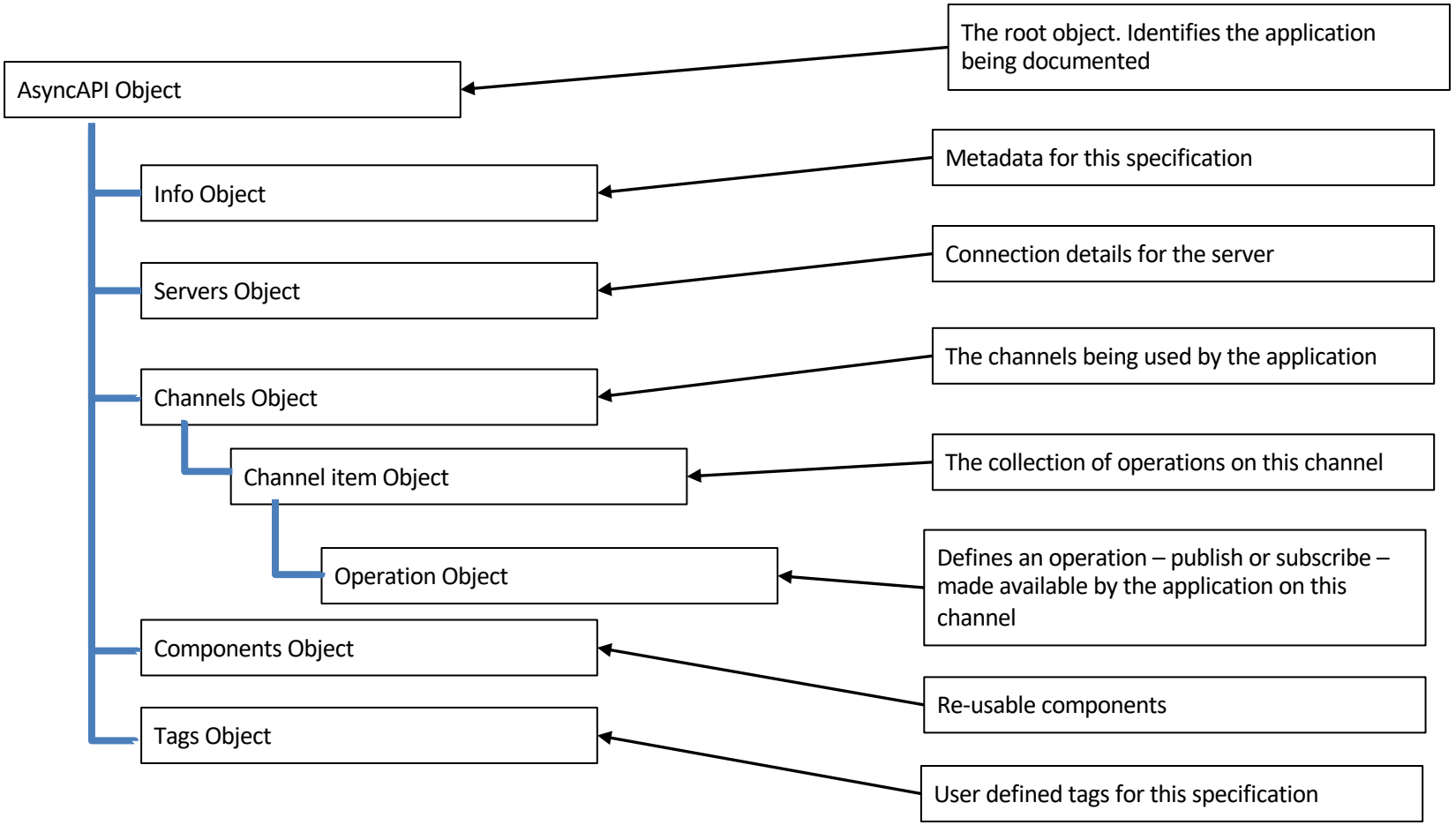
We have many different tools and welcome you to explore our ideas and propose new ideas to AsyncAPI.

View GitHub Discussions

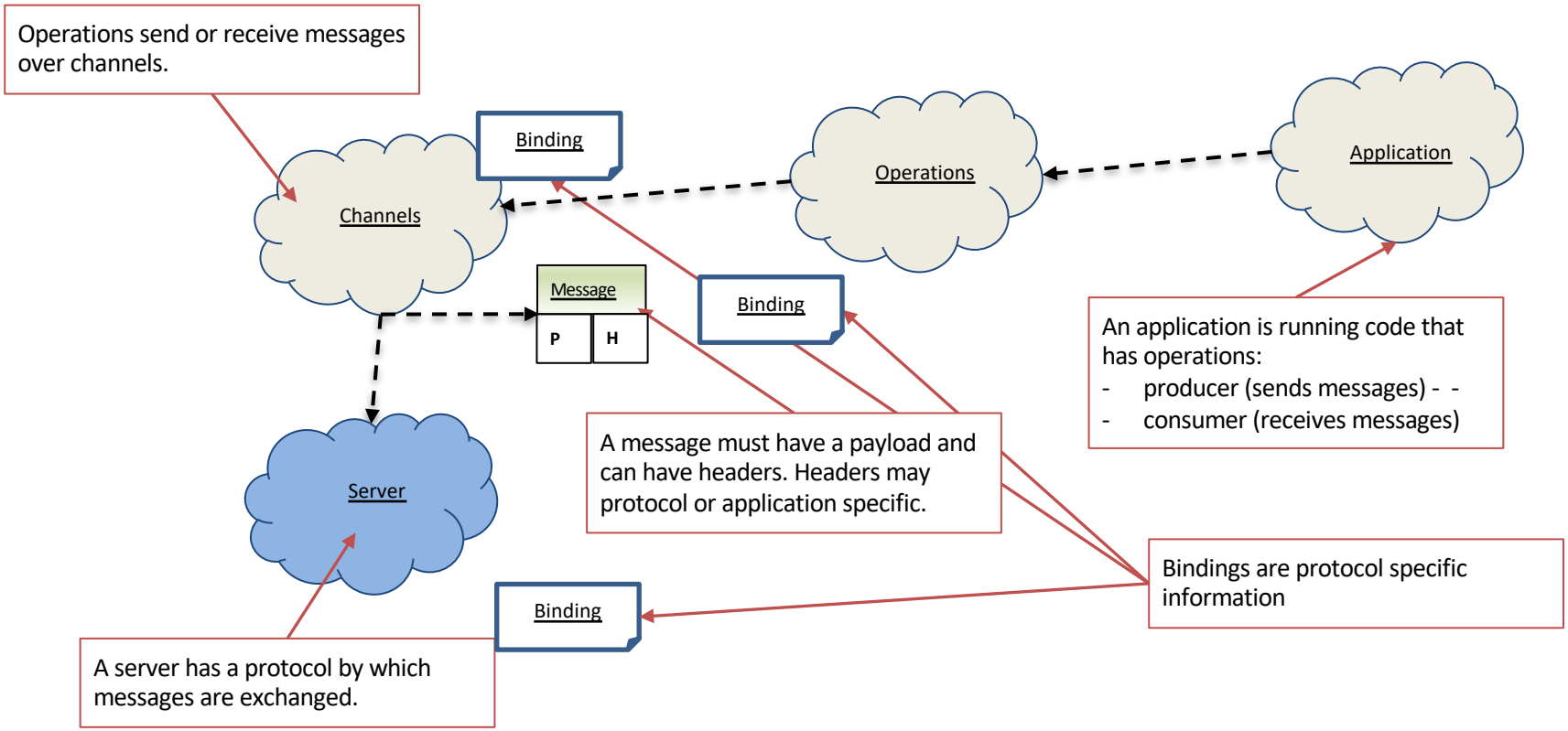
# AsyncAPI Elements (V2)



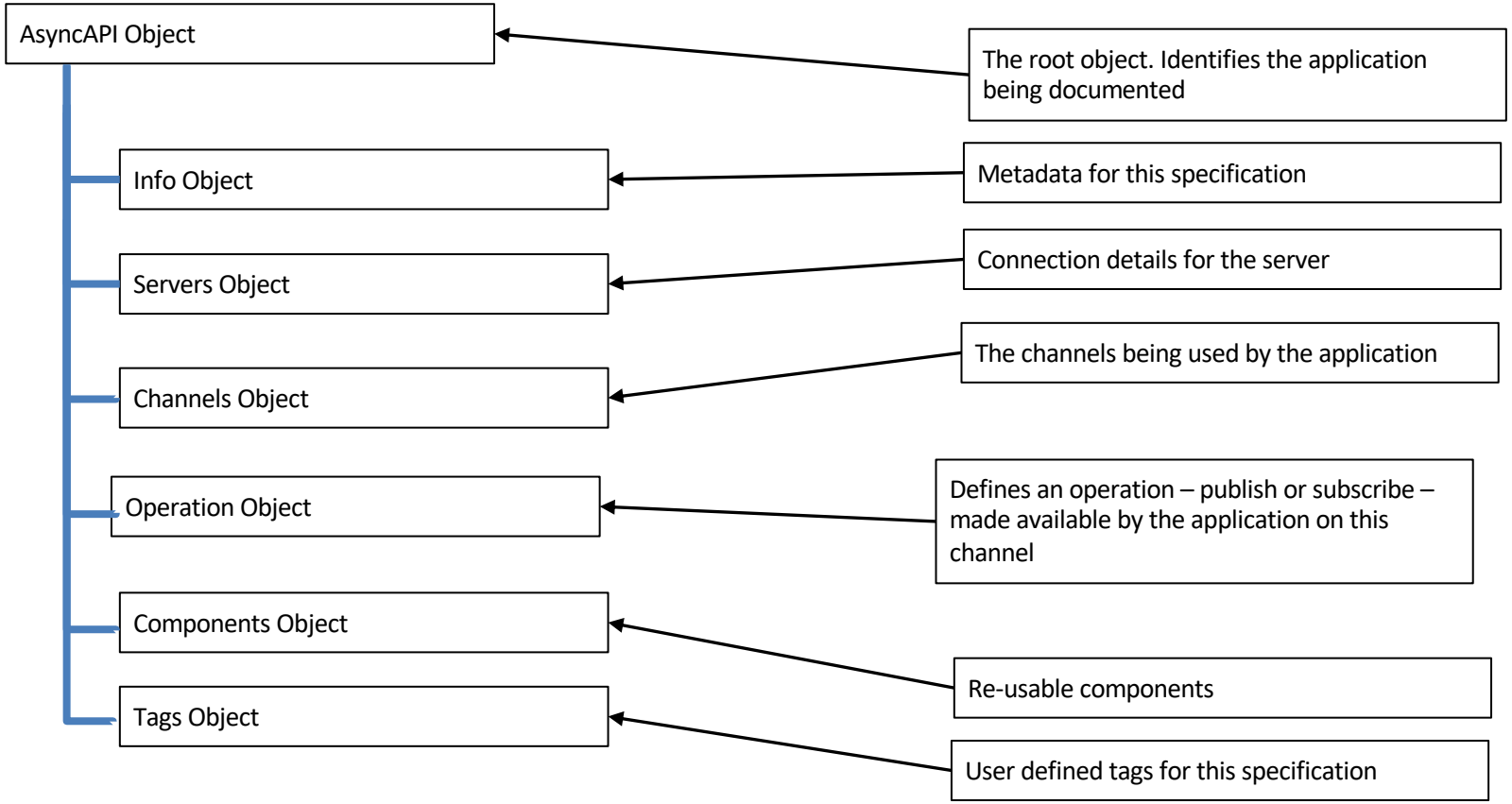
# Document Structure (V2)



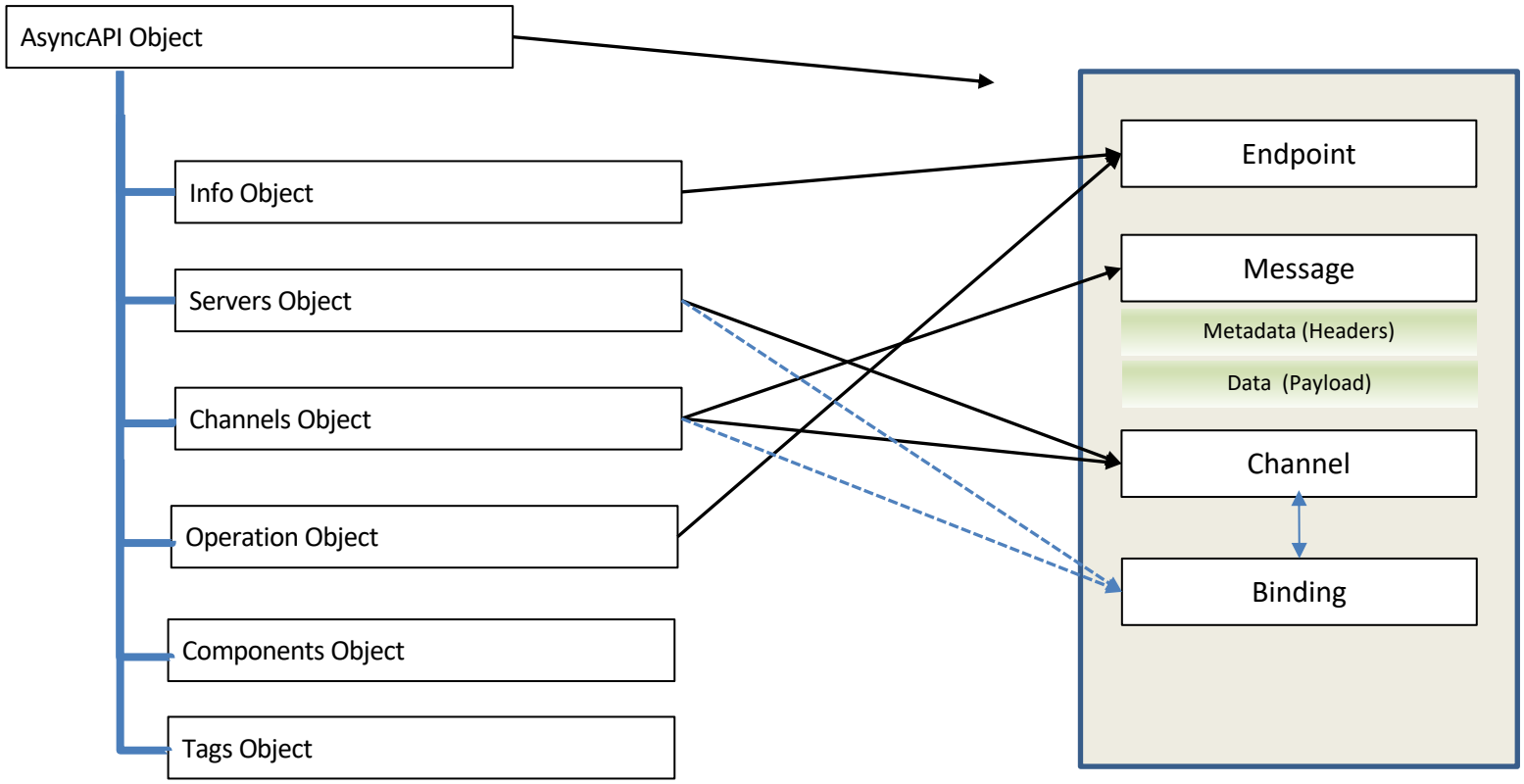
# AsyncAPI Elements (V3)



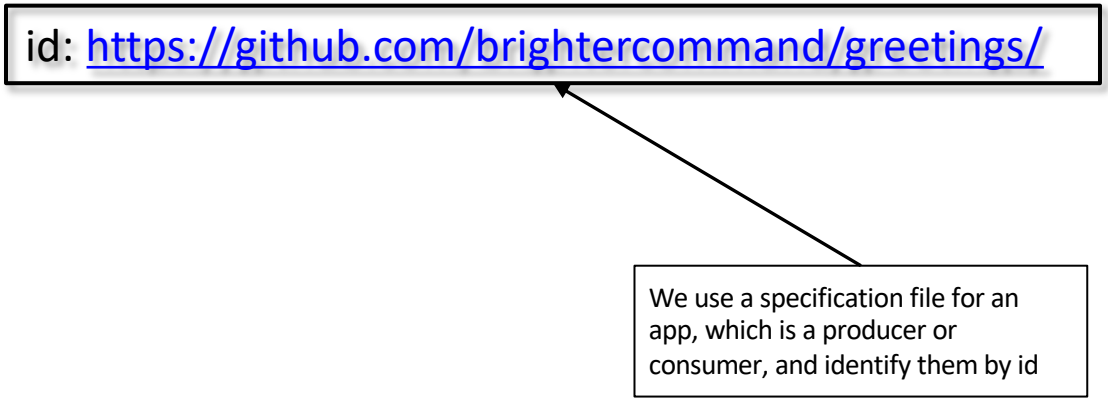
# Document Structure (V3)



# AsyncAPI (V3) and Endpoint ABCs



# AsyncAPI Object



## 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**

# 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

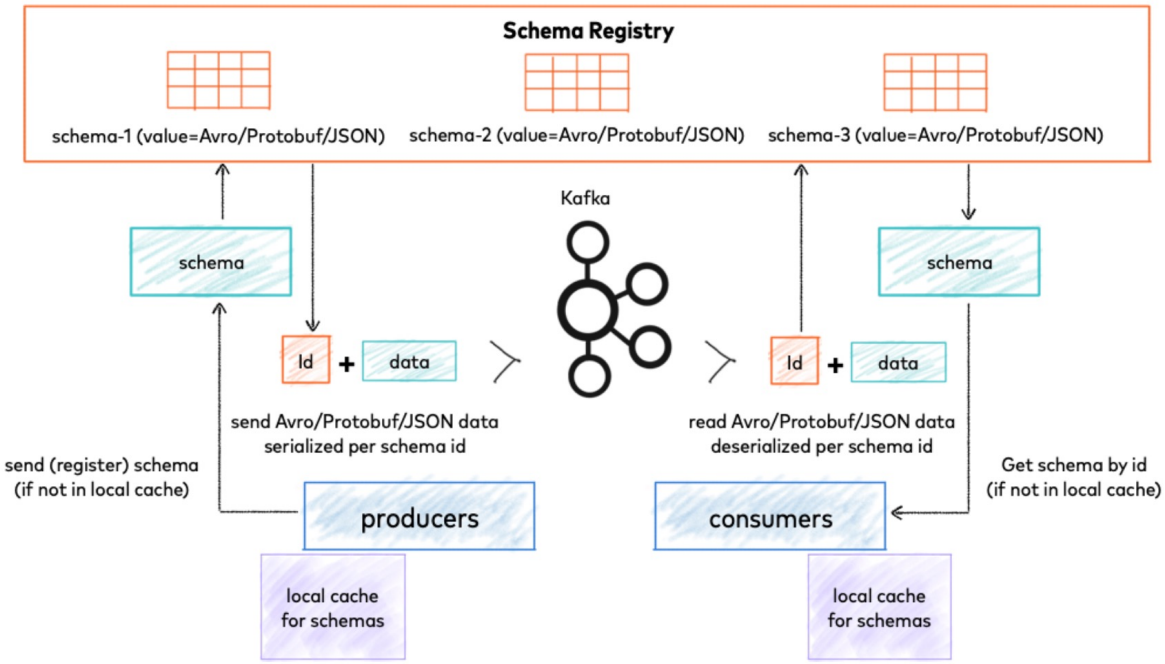
```
syntax = "proto3";

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

# Protobuf

Encodings: **Binary**

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

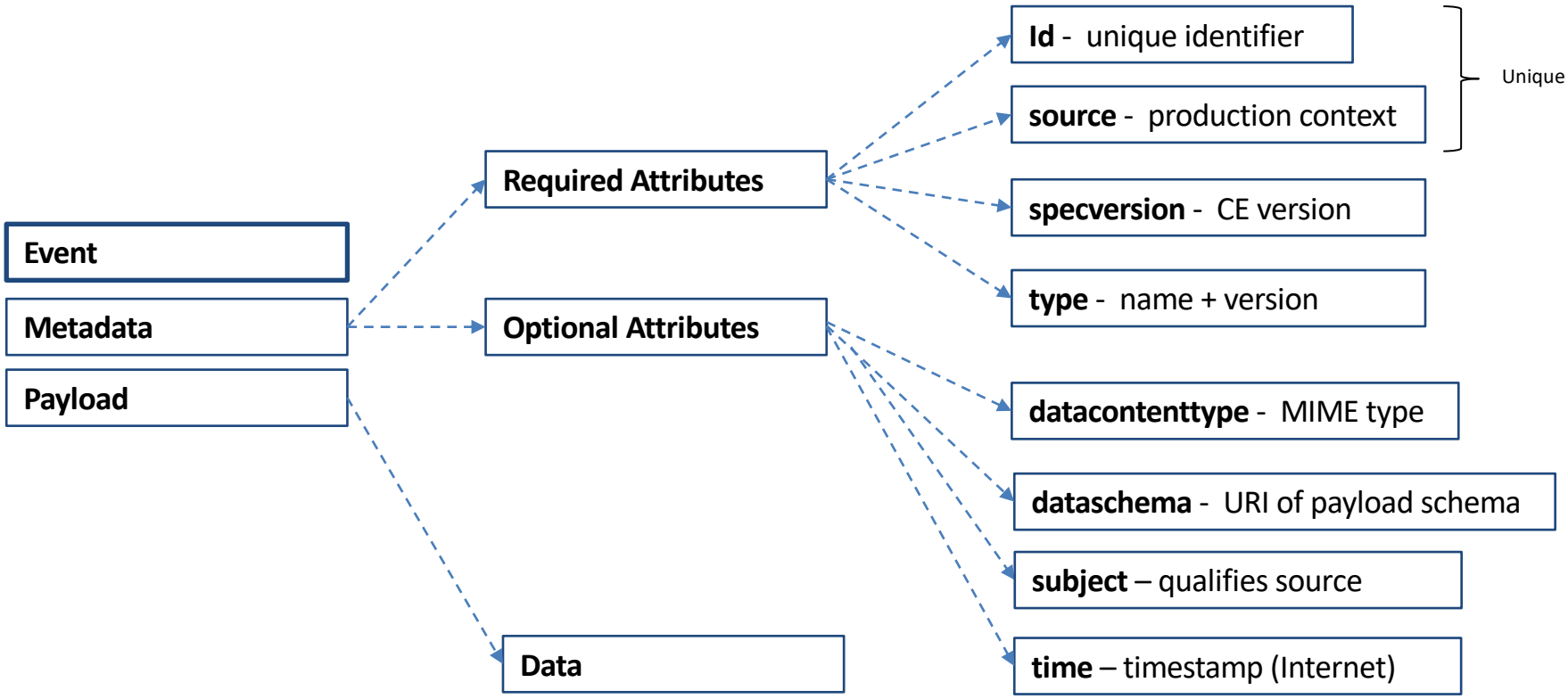
# Schema Registry



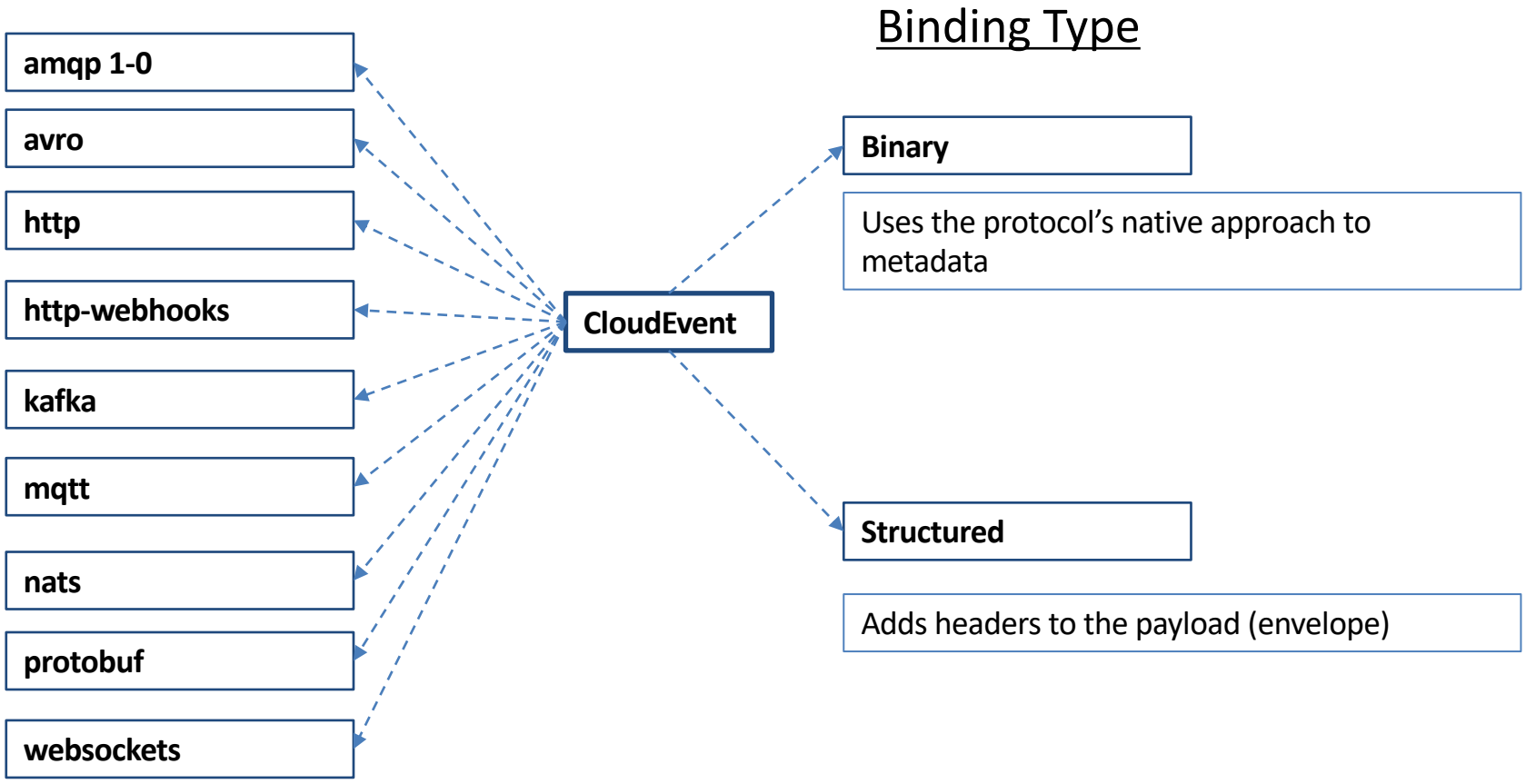

*Confluent Schema Registry for storing and retrieving schemas*

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

# Cloud Events



# Protocol Binding



# Protocol Binding

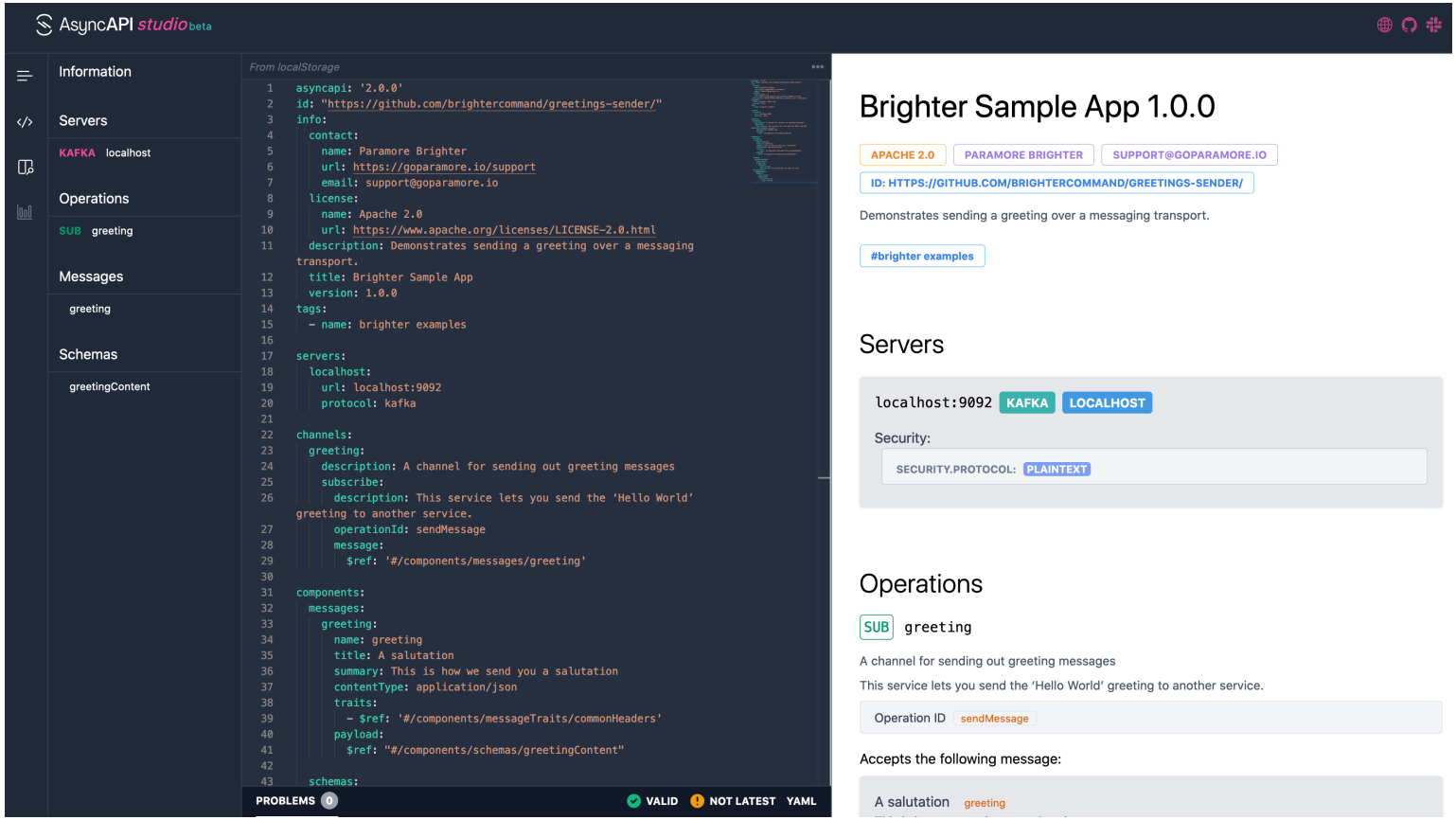
**Binary**

```
------------------ 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 ...
-----------------------------------------------
```

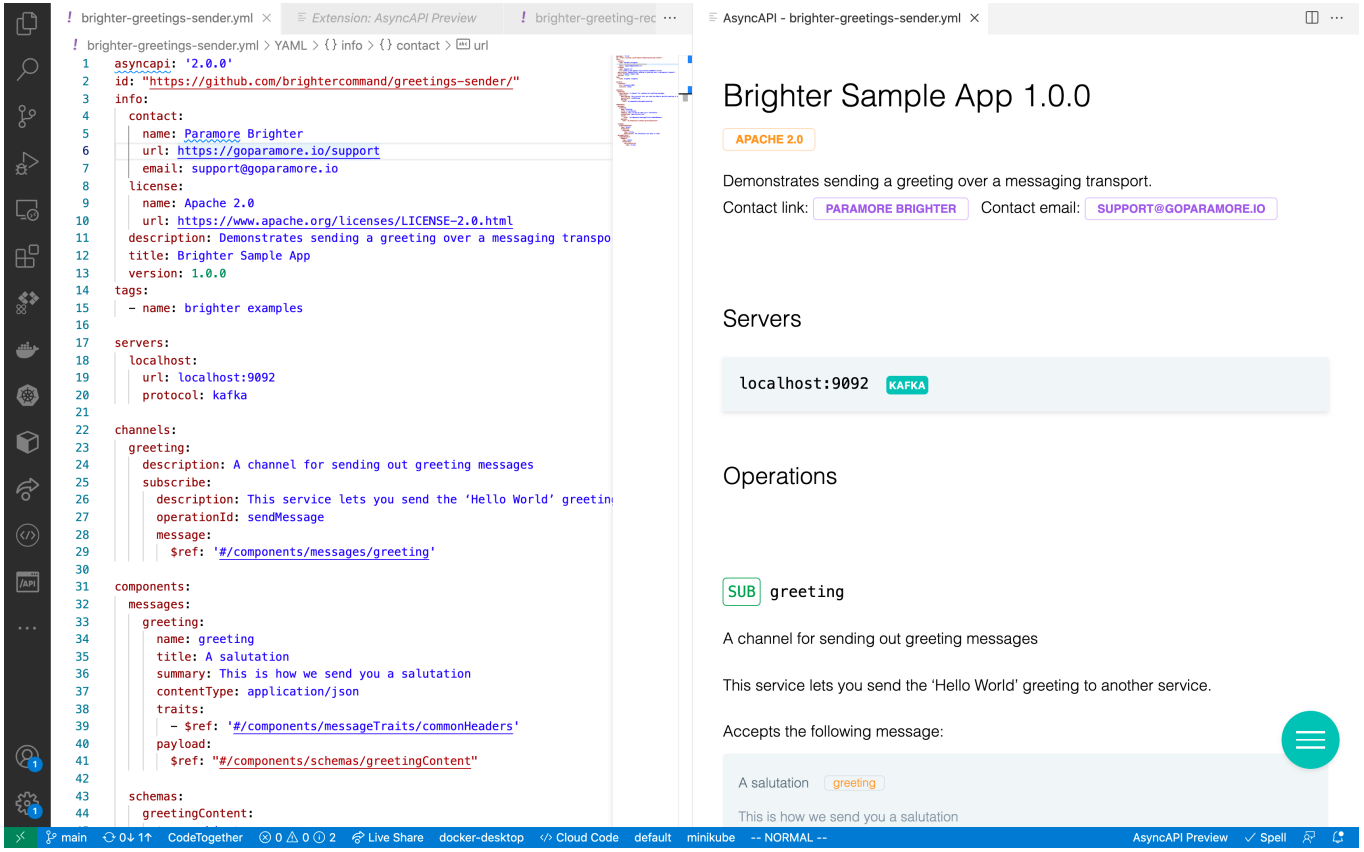
**Structured**

```
------------------ 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 ...
        }
}
-----------------------------------------------
```

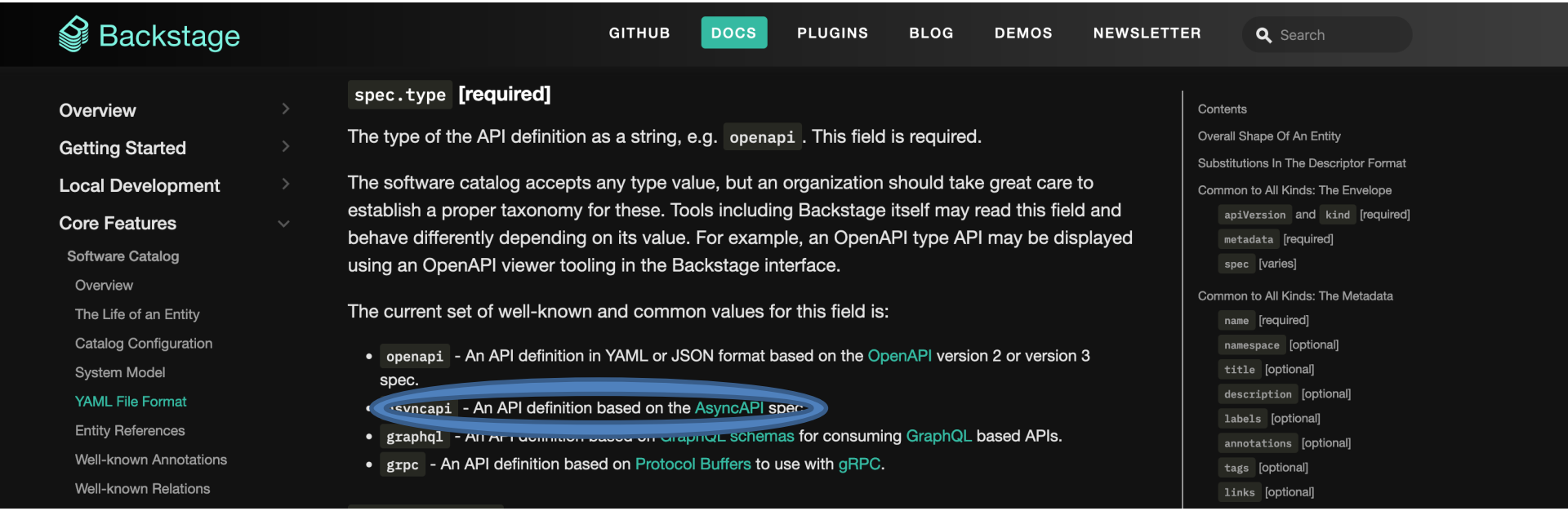
# AsyncAPI Studio



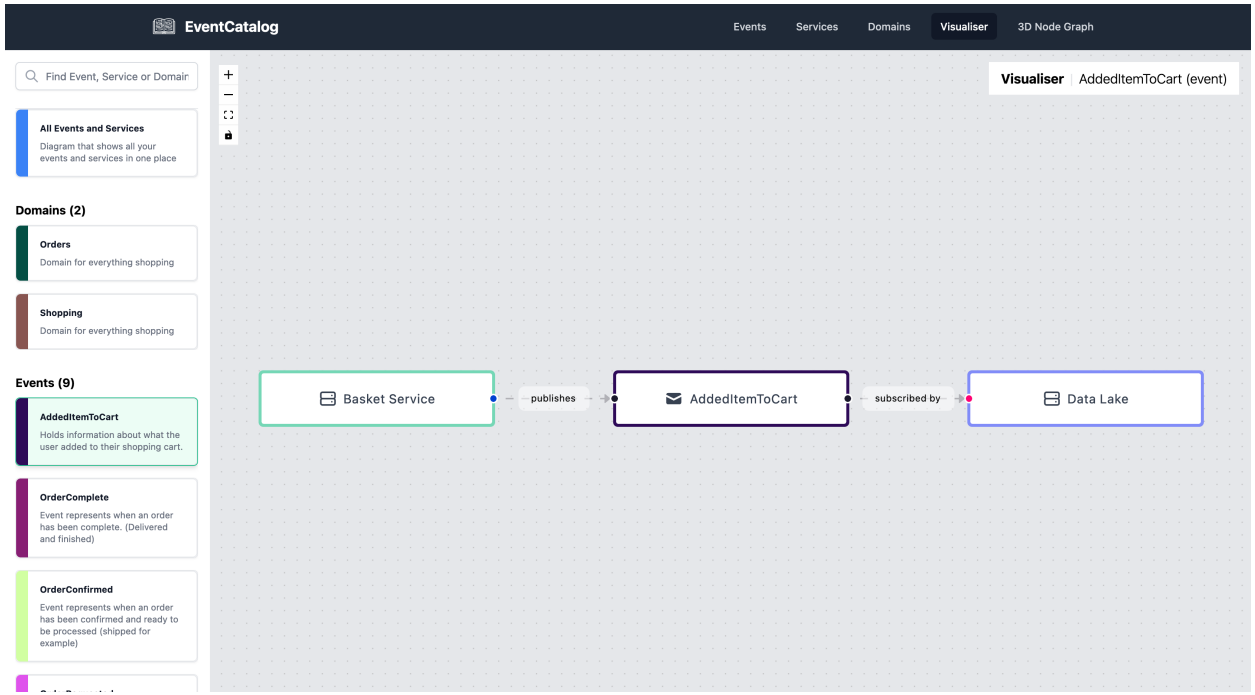
# VS Code


brighter-greetings-sender.yml
Extension: AsyncAPI Preview
brighter-greeting-rec
brighter-greetings-sender.yml > YAML > { } info > { } contact > url
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
main
CodeTogether
Live Share
docker-desktop
Cloud Code
default
minikube
-- NORMAL --
AsyncAPI Preview
Spell

# Backstage

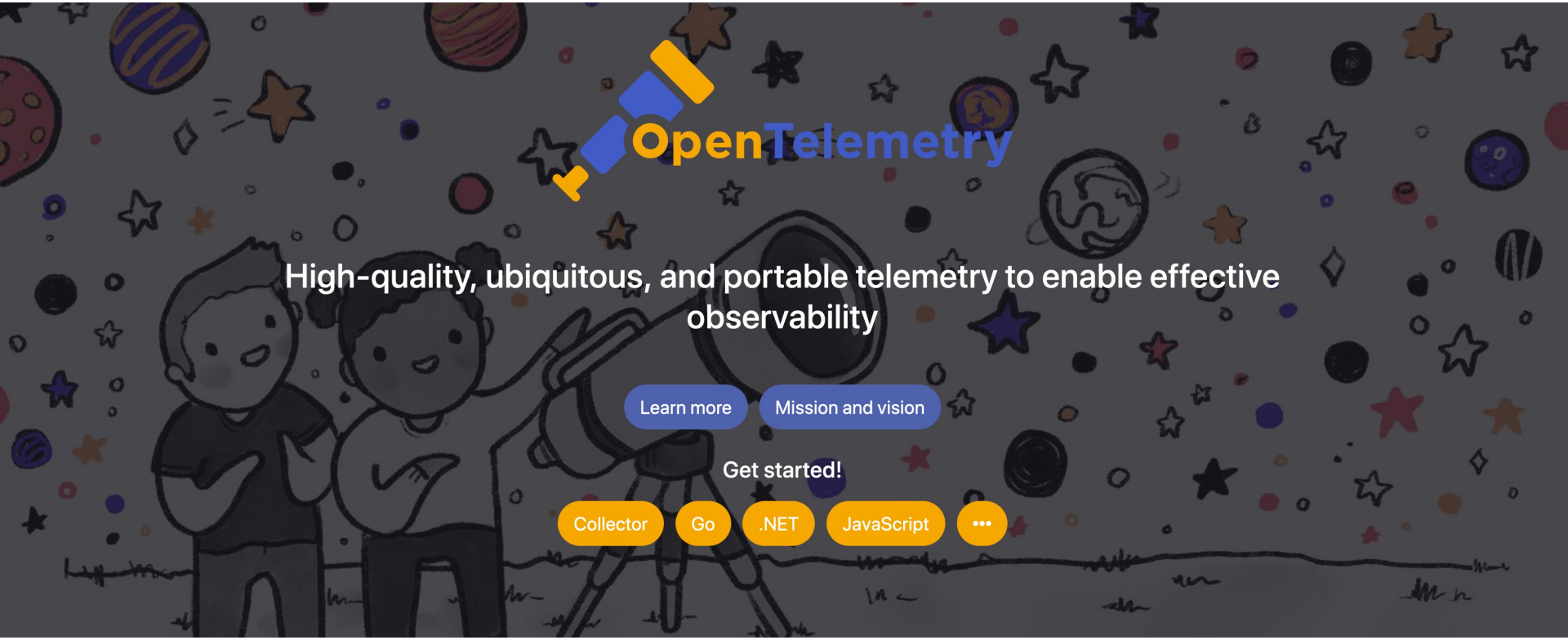

Backstage
GITHUB
DOCS
PLUGINS
BLOG
DEMOS
NEWSLETTER
Search
Overview
Getting Started
Local Development
Core Features
Software Catalog
Overview
The Life of an Entity
Catalog Configuration
System Model
YAML File Format
Entity References
Well-known Annotations
Well-known Relations
spec.type [required]
The 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
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

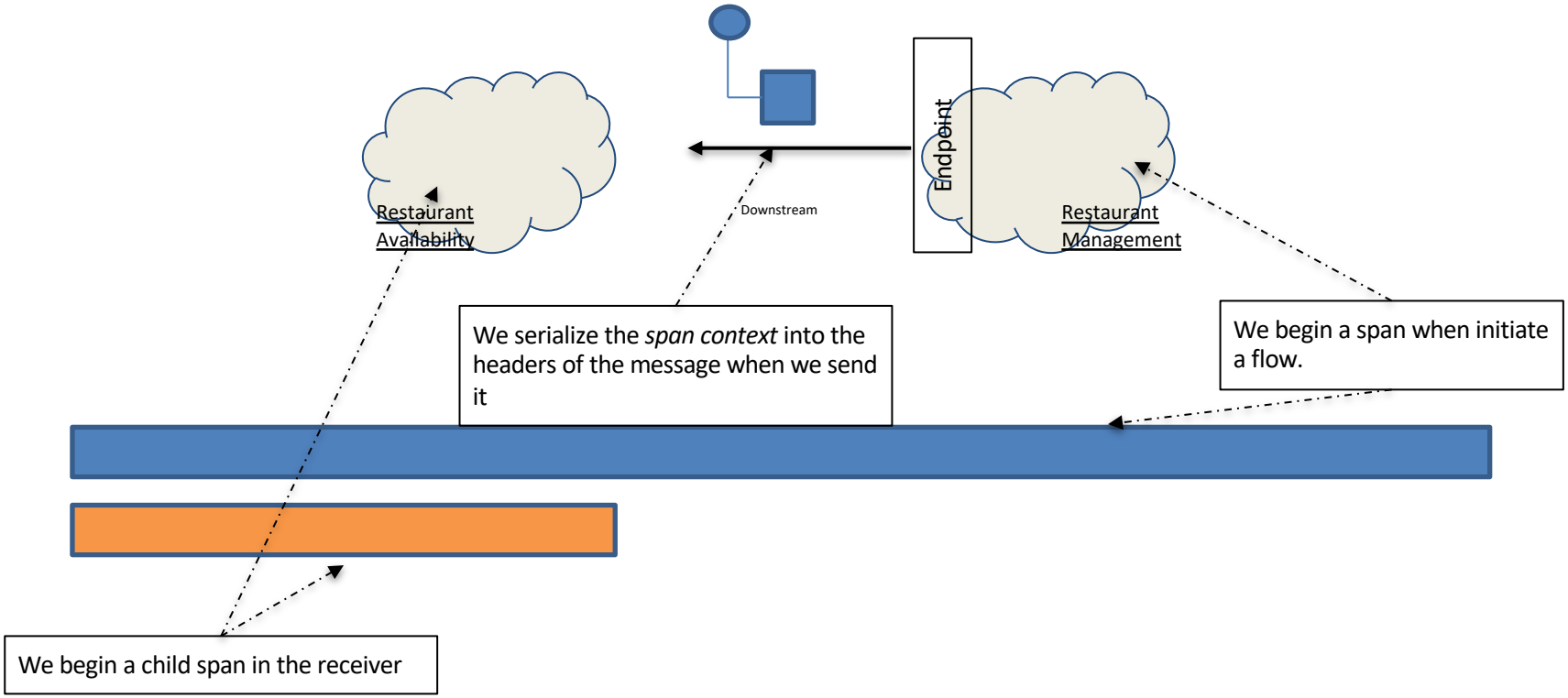


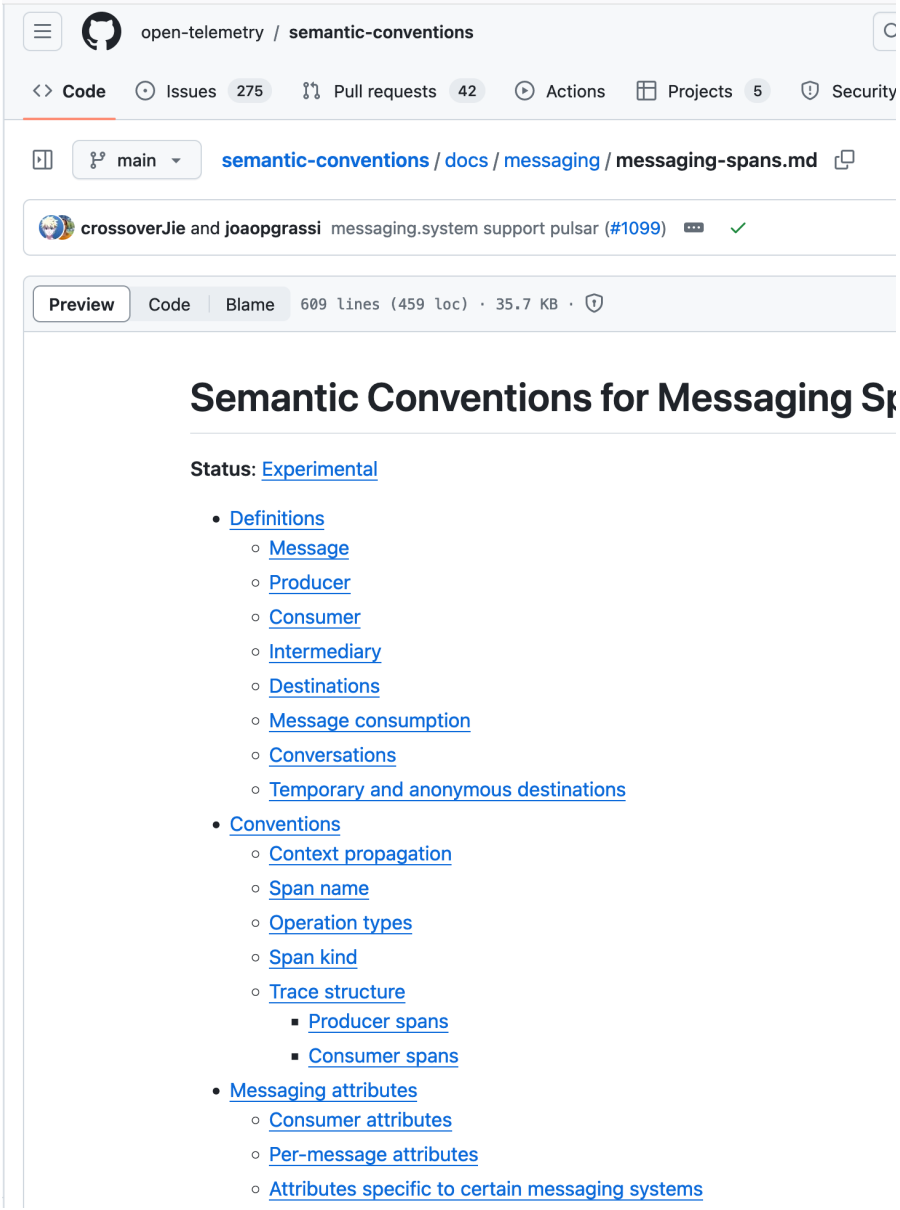

https://github.com/boyney123/eventcatalog

# Observability


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

OpenTelemetry Tracing






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

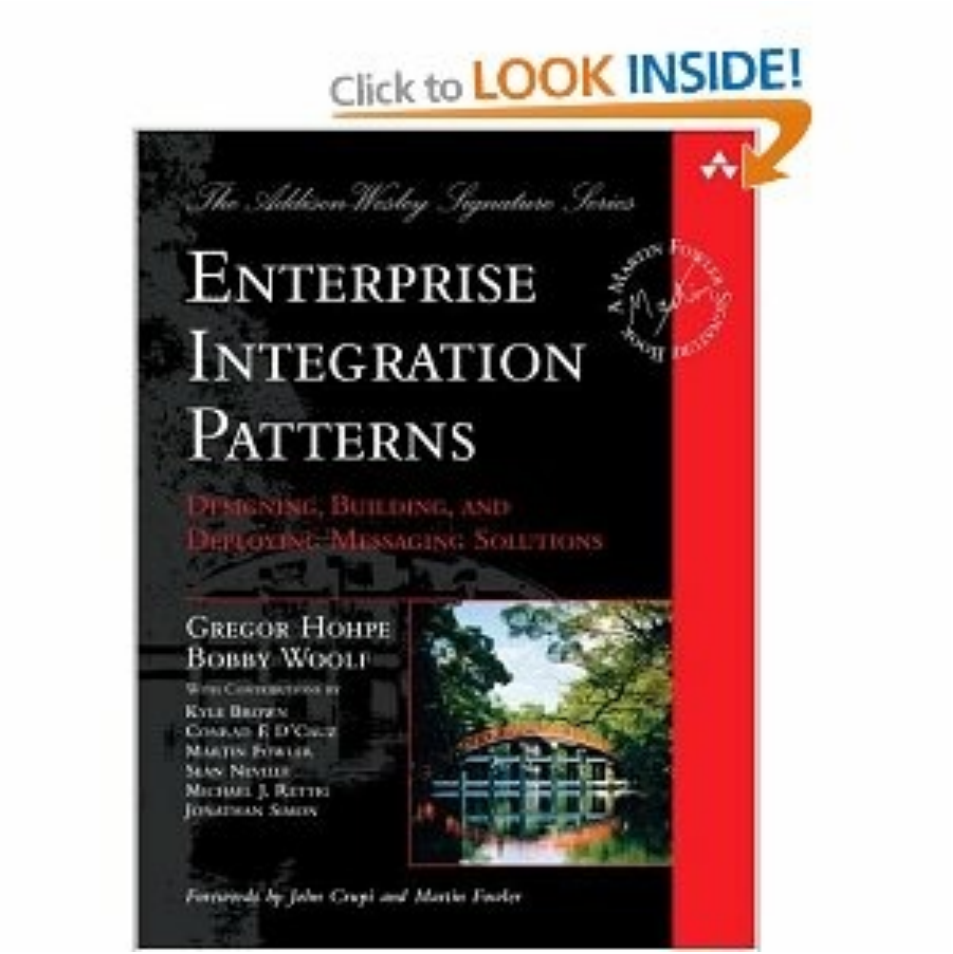
### ɔes

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

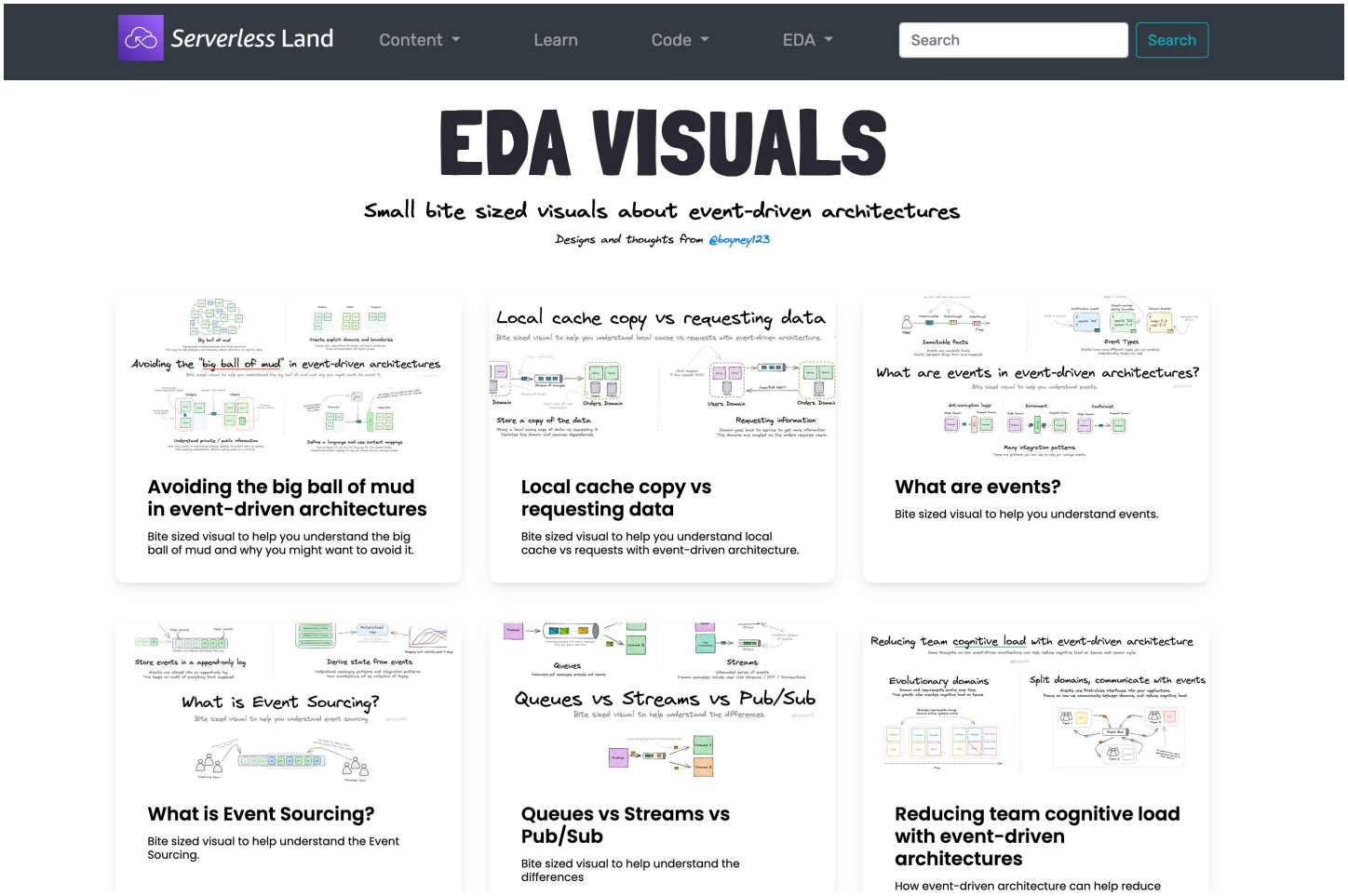
| | Description |
|---|---|
| | A message is created or passed to a client library for publishing. "Create" spans always r are used to provide a unique creation context for messages in batch publishing scenario |
| | One or more messages are provided for publishing to an intermediary. If a single messag 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. |

# NEXT STEPS

# Further Reading



# Further Reading




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

# Q&A