Practical Messaging

A 101 guide to messaging

Ian Cooper

Twitter: 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 Two Agenda

- Series and Discrete
- Order
- Guaranteed Delivery
- Event Driven Collaboration
- Versioning
- Documentation
- Observability
- CAP Theorem
- Consumers
- Next Steps

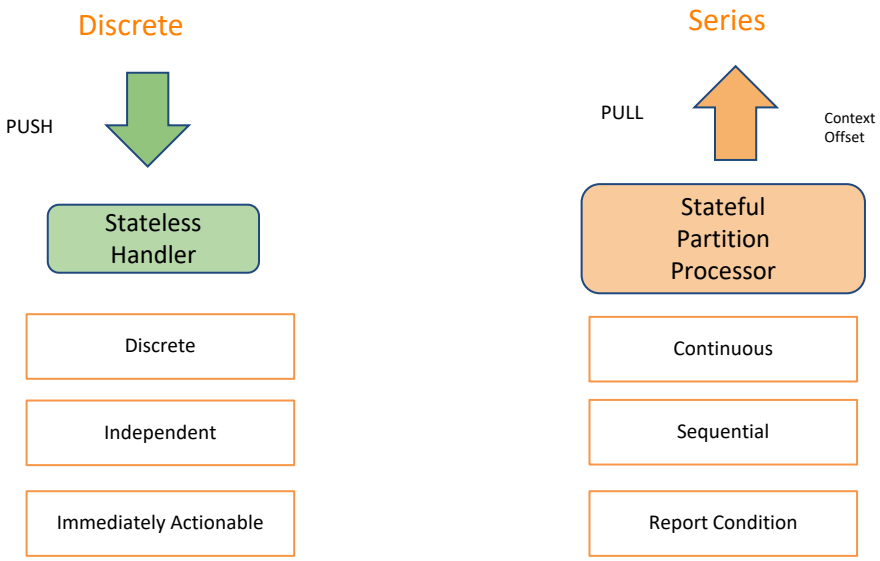
Housekeeping

- Some of these slides are dense
  - Part of the goal is to give you a takeaway that you can refer to when you need the detail after this course
  - But it can make some hard to follow
  - Worse in person than remote
- Download the PDF or Slides
  - You can refer to that on your laptop for the detail if it helps you
  - https://github.com/iancooper/Presentations
  - A PDF/Slides for each day

Day Two

1 SERIES AND DISCRETE

Event Types



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

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

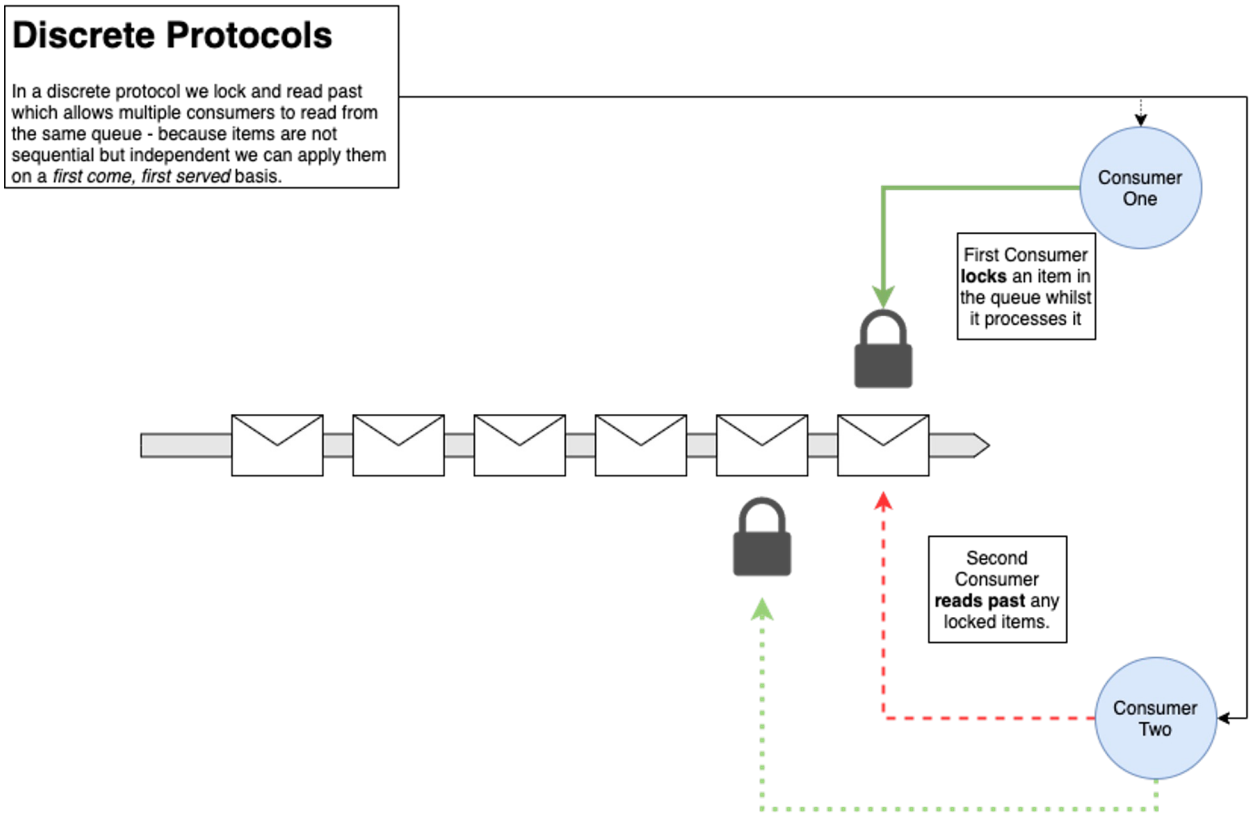
(Message)Event – Skinny, Notification

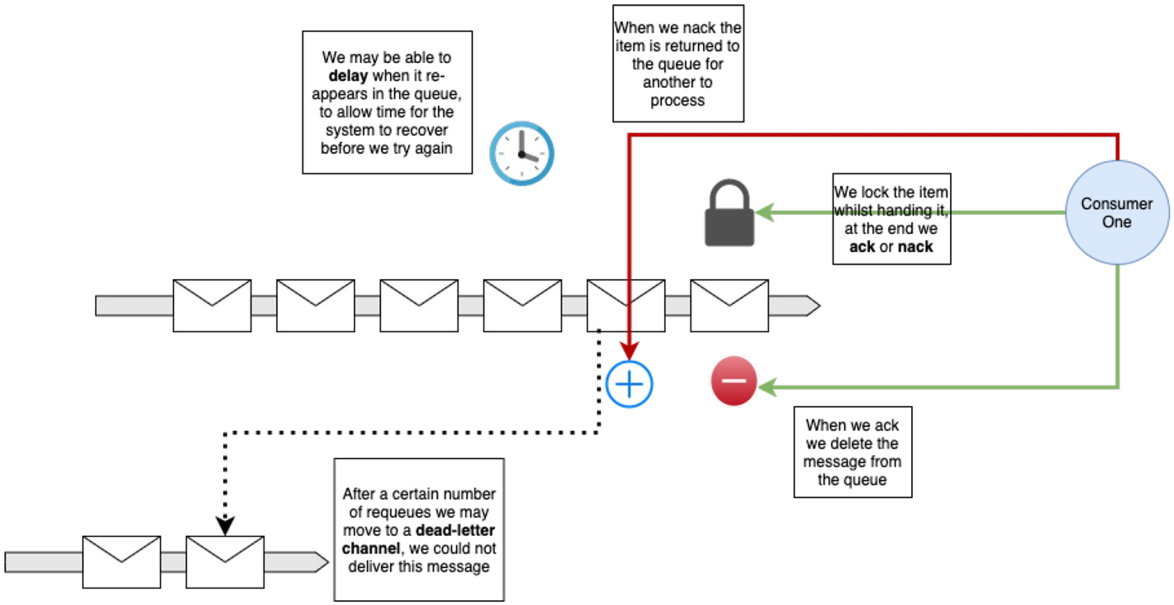
Discrete, Immediately Actioned

(Message)Document – Fat

Series, Supports Action

Discrete Events





No Archive and Replay

With discrete messages we delete a message once it is consumed - it is a task that we want to do once - but that means we have no built in archive and replay. Replay depends on the producer, and in a pub-sub model would require broadcast to all subscribers who need to discard duplicates unless we can send directly to one consumer.

Discrete Messaging as Tasks

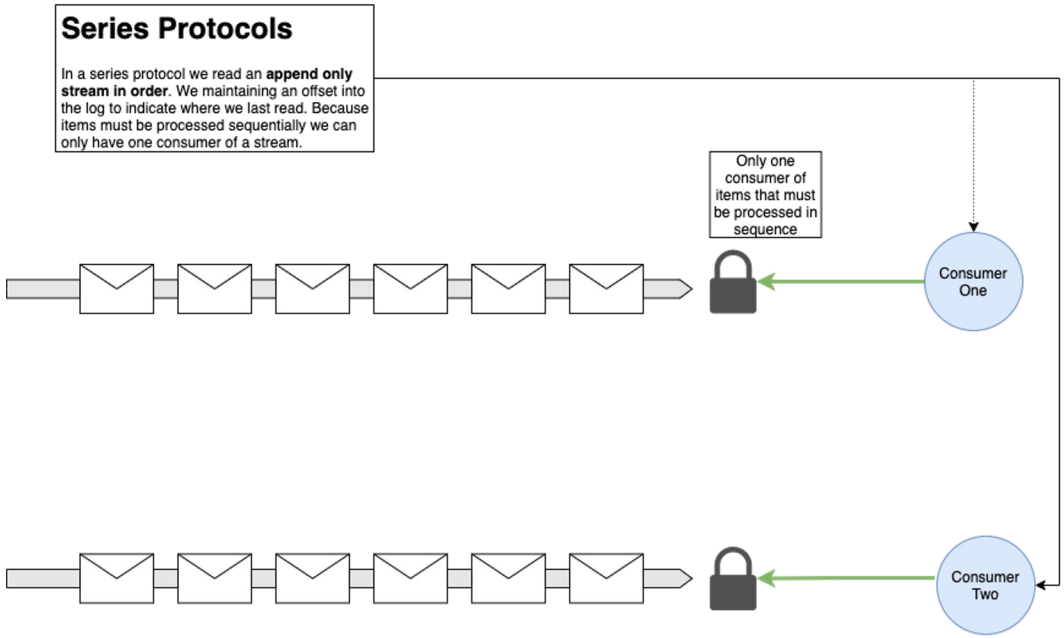
With discrete messages we can think of information packets that are tasks - we do something in response to receiving them. Once the task is done we delete the task,

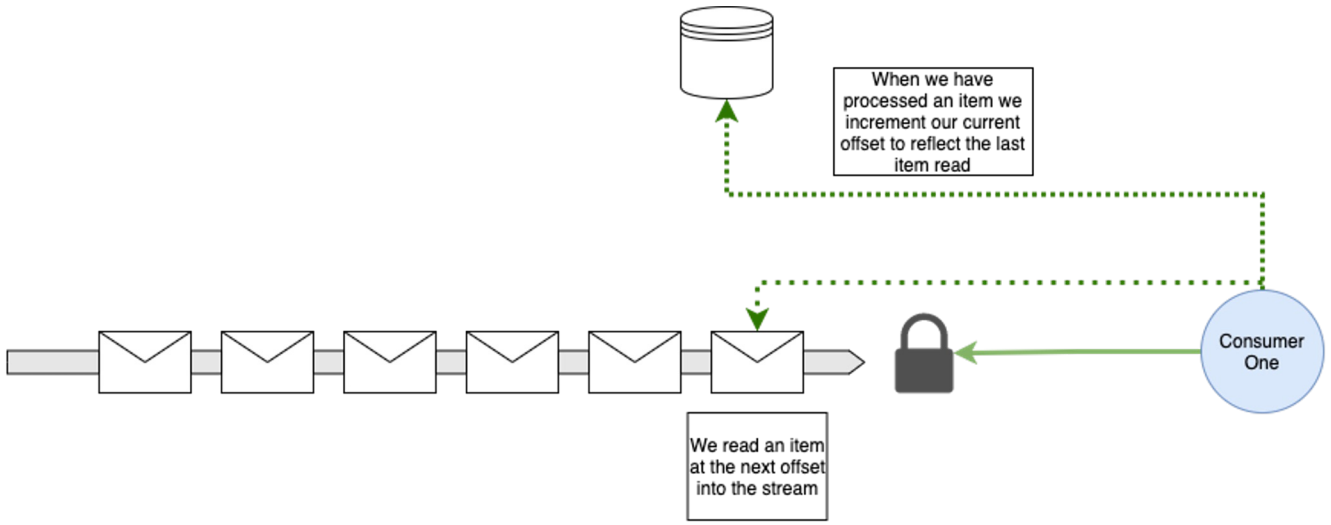
+ We don't want anyone else to attempt it, it's already done.

+ An originator should re-raise the task if it should be repeated.

+ If we can't process it but it is processable, we can nack, so that someone else can attempt it

Series Events





Series Messaging as Streams

With series messages, the stream is append-only. We do not ack or nack, instead we increment our offset into the stream.

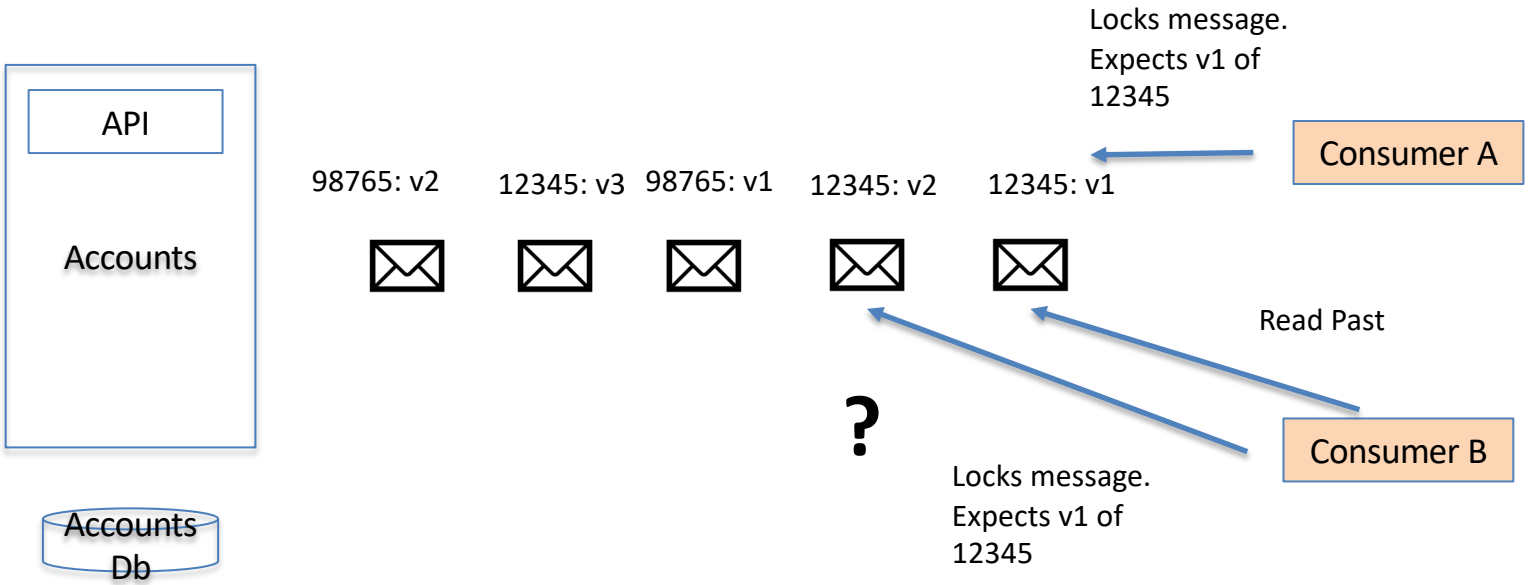
+ We can replay the stream if we want to.

+ The stream can be used for complex event processing i.e. create a table over the stream for a given window, often called inverting the database.

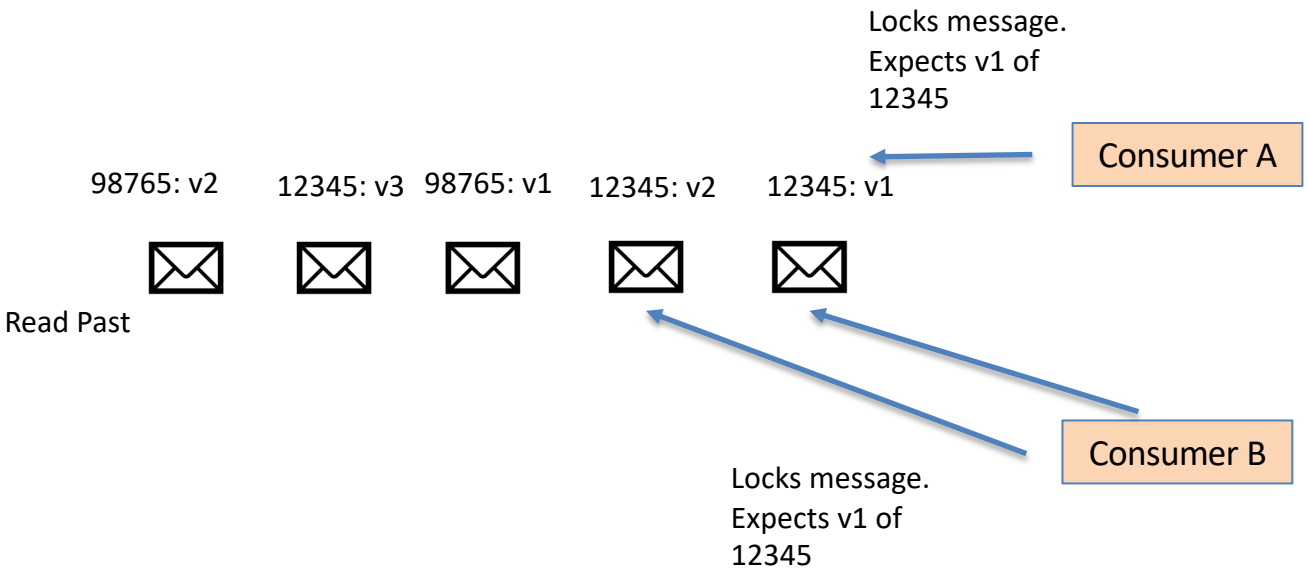
In Order Delivery

2 ORDER

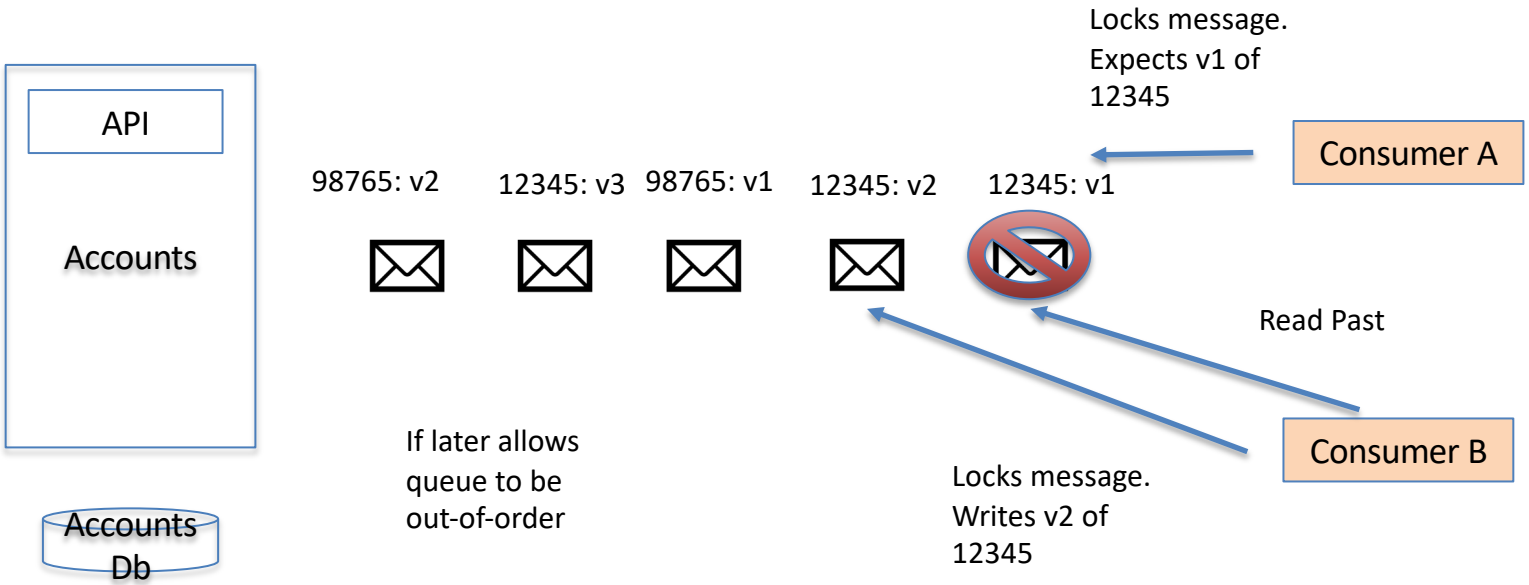
Messaging Order and Competing Consumers



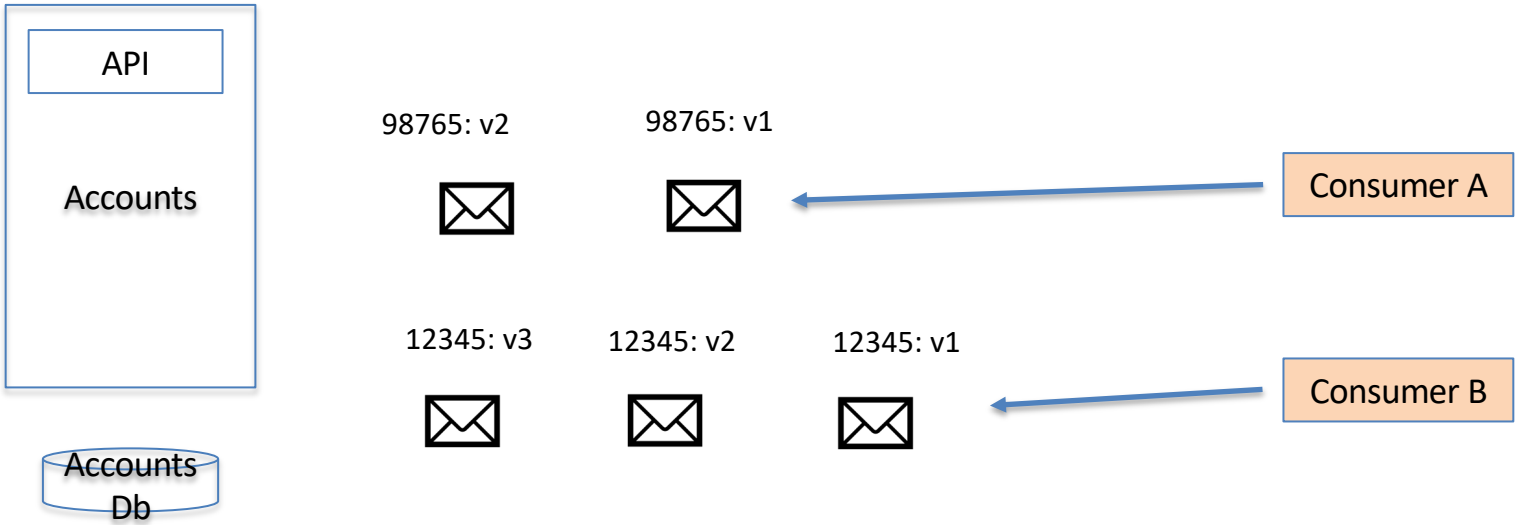
Requeue?


Locks message.
Expects v1 of
12345
Consumer A
98765: v2
12345: v3
98765: v1
12345: v2
12345: v1
Read Past
Consumer B
Locks message.
Expects v1 of
12345

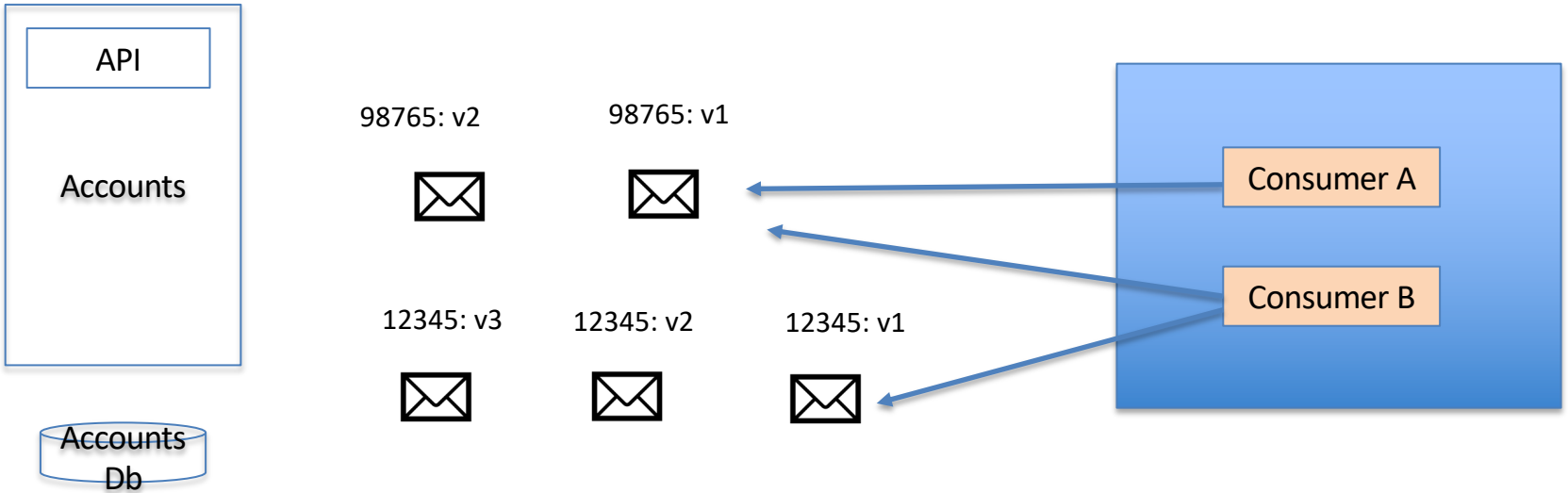
If Later


API
Accounts
Accounts Db
98765: v2
12345: v3
98765: v1
12345: v2
12345: v1
Locks message. Expects v1 of 12345
Consumer A
Read Past
Consumer B
If later allows queue to be out-of-order
Locks message. Writes v2 of 12345

Consistent Hashing



Consistent Hashing w. Consumer Groups



Transactional Messaging

3 GUARANTEED DELIVERY

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

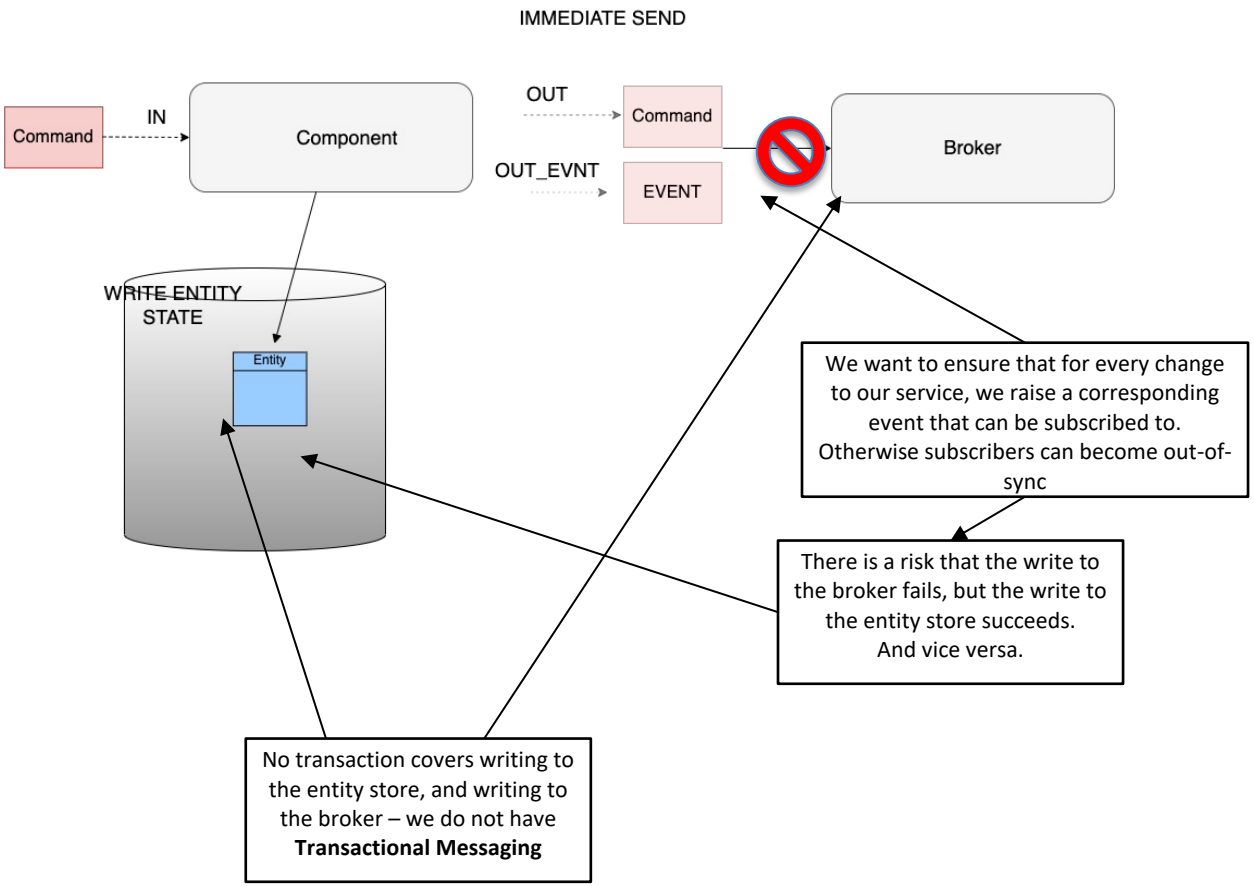
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Correctness



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

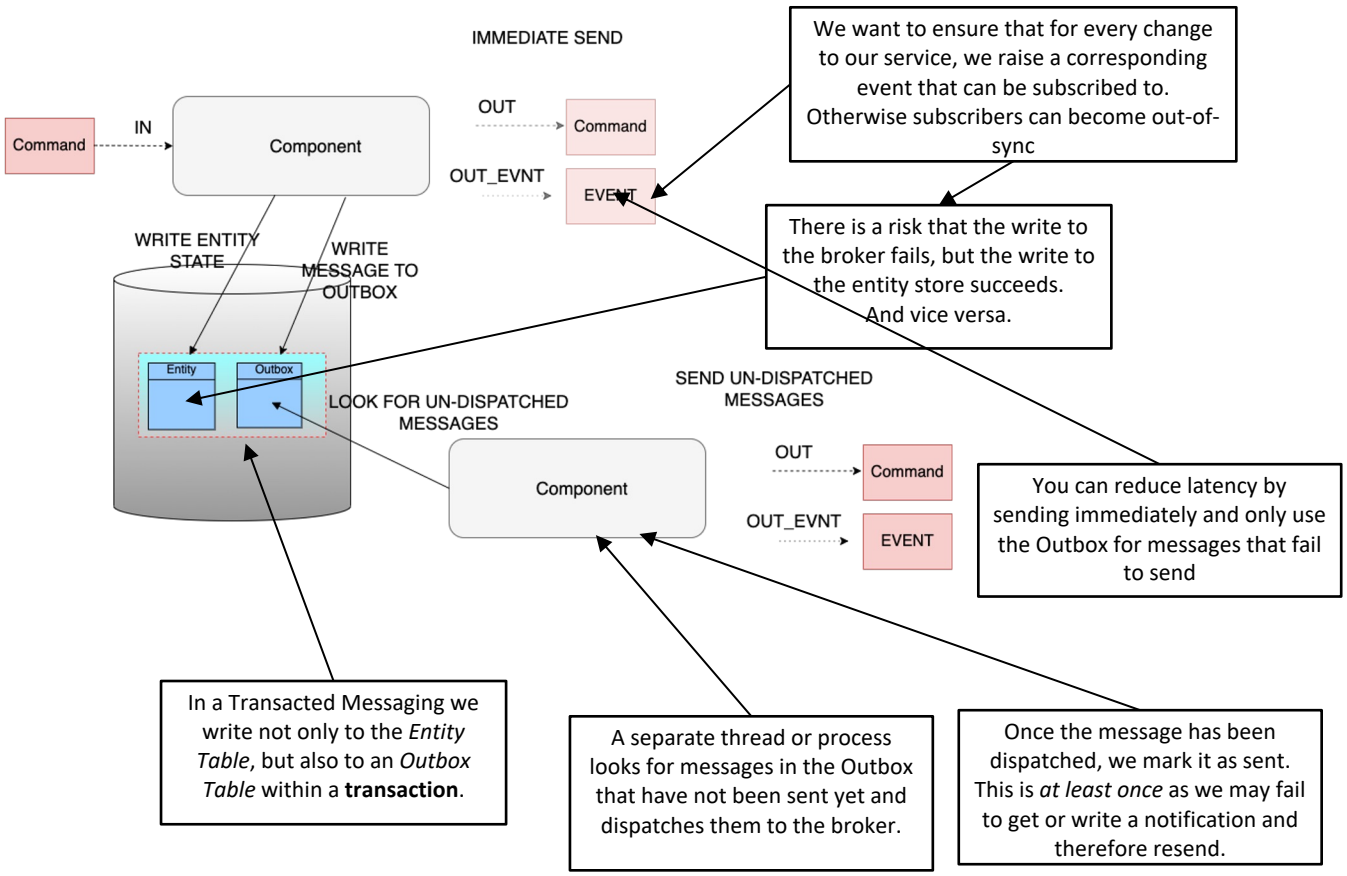
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Outbox



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

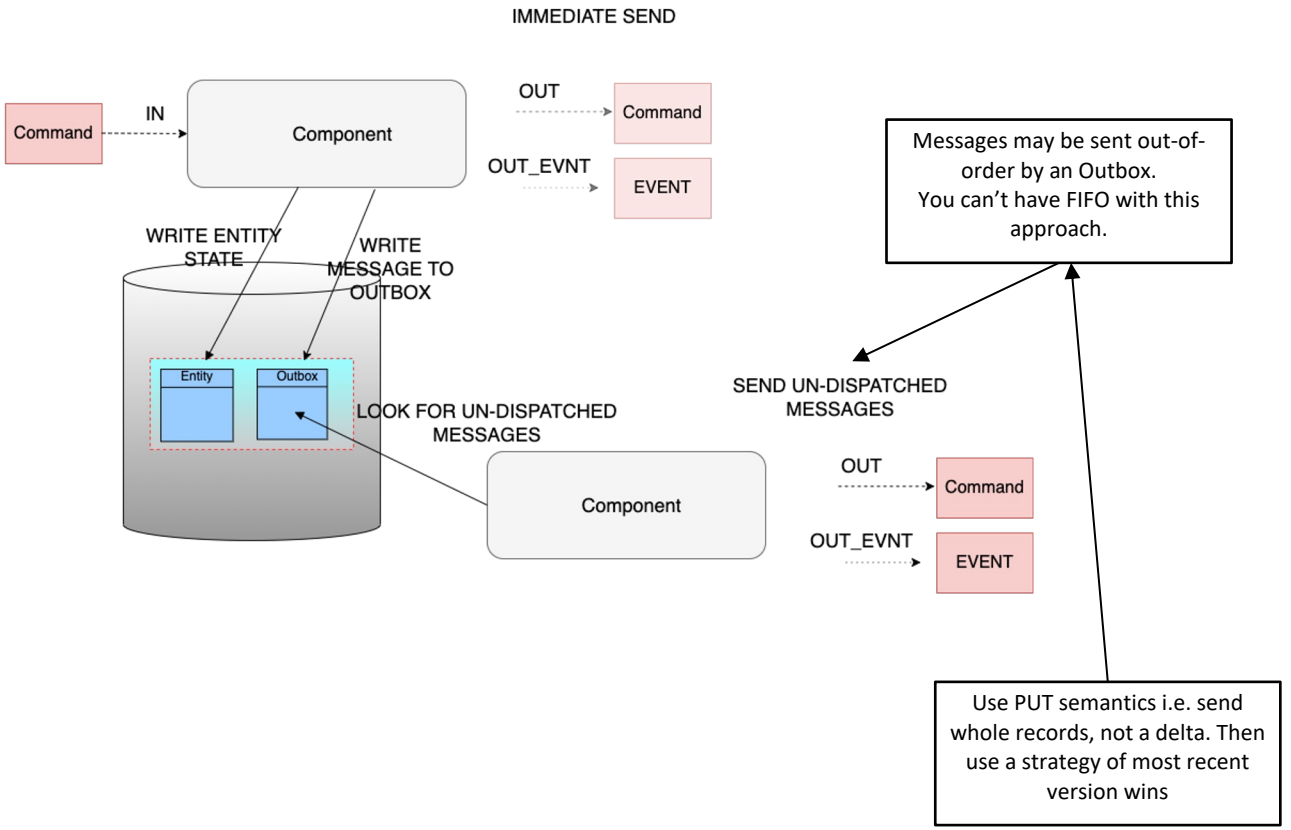
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Outbox



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

Log
- Store and Forward
- Immutable and Ordered
- Partitioned

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

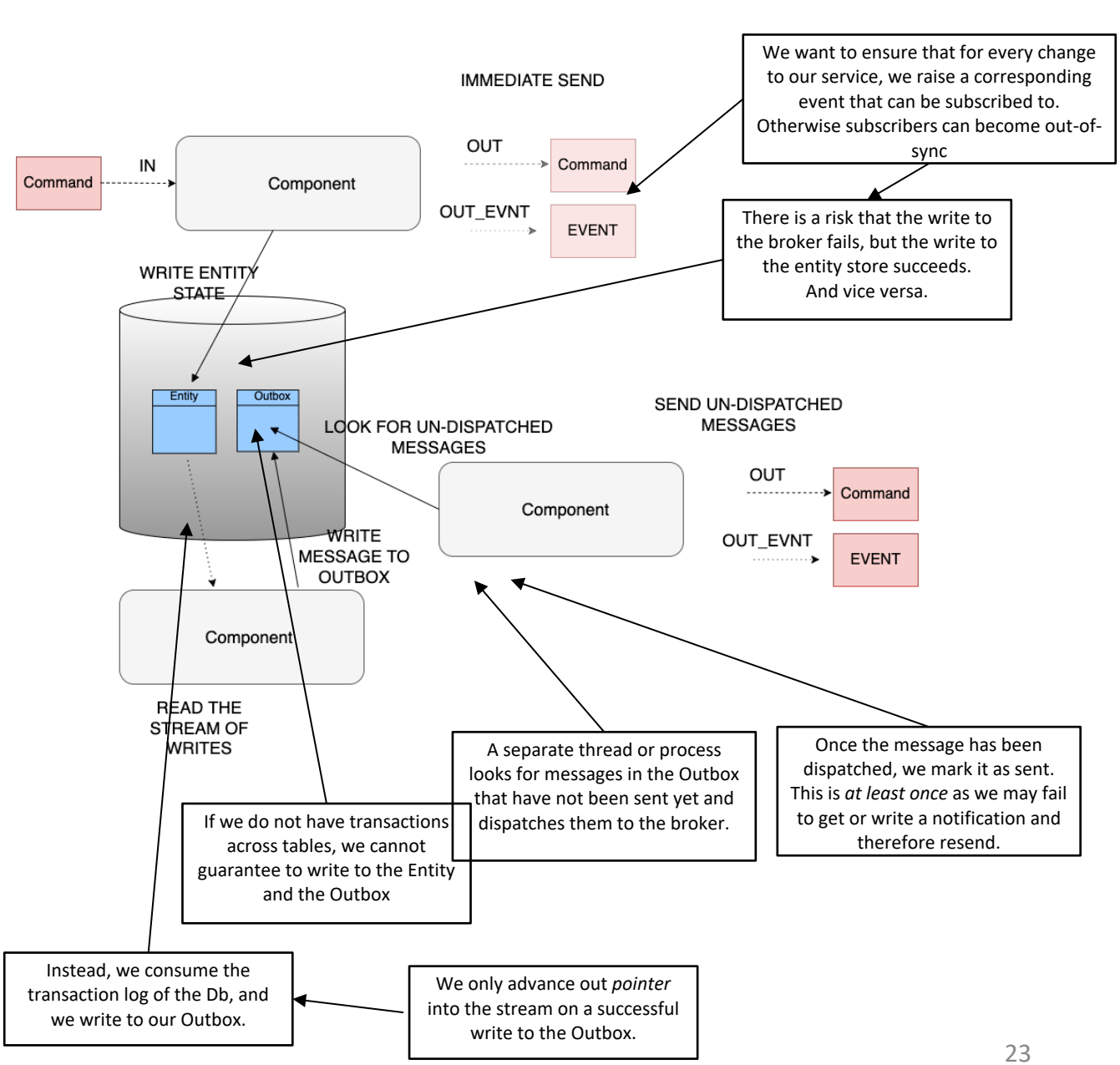
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Log Tailing



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

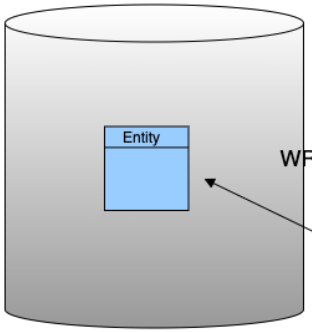
Local Data
- Mutable, Fresh
- Provides State
- Details, Private

State Change Capture

We want to ensure that for every change to our service, we raise a corresponding event that can be subscribed to.

We avoid the transactional message problem – we just raise the event describing the intended change of state, and do not write to the backing store

IMMEDIATE SEND



WRITE ENTITY STATE

LISTEN TO EVENT FOR ENTITY STATE CHANGE

Component

We update our entity state in response to the event that notifies of a change in the state of the entity. So we and our listeners build current state from the same stream.

This can lead to consistency issues where the downstream reacts to the new state, before the upstream does.

At Least Once

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete

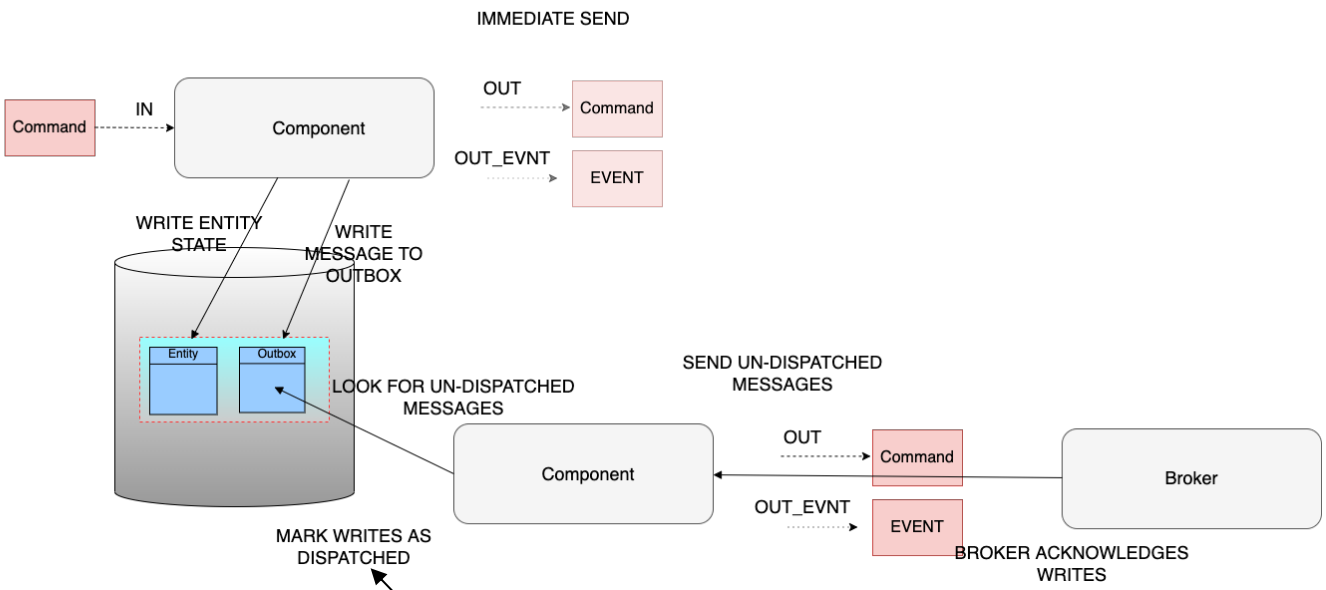
Event

Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Outbox



We mark a write as dispatched in response to a confirmation from the broker. But we cannot know from lack of confirmation:
-Did the broker fail to write & propagate
-Did we fail to get the response

A broker acknowledges writes. It may acknowledge prior to propagating to all nodes, but for safety it often acknowledges writes that propagate to all nodes **asynchronously**

Because we do not mark as dispatched, we will **send again** regardless of whether the write happened.
Thus we must expect duplicates.
At least once.

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

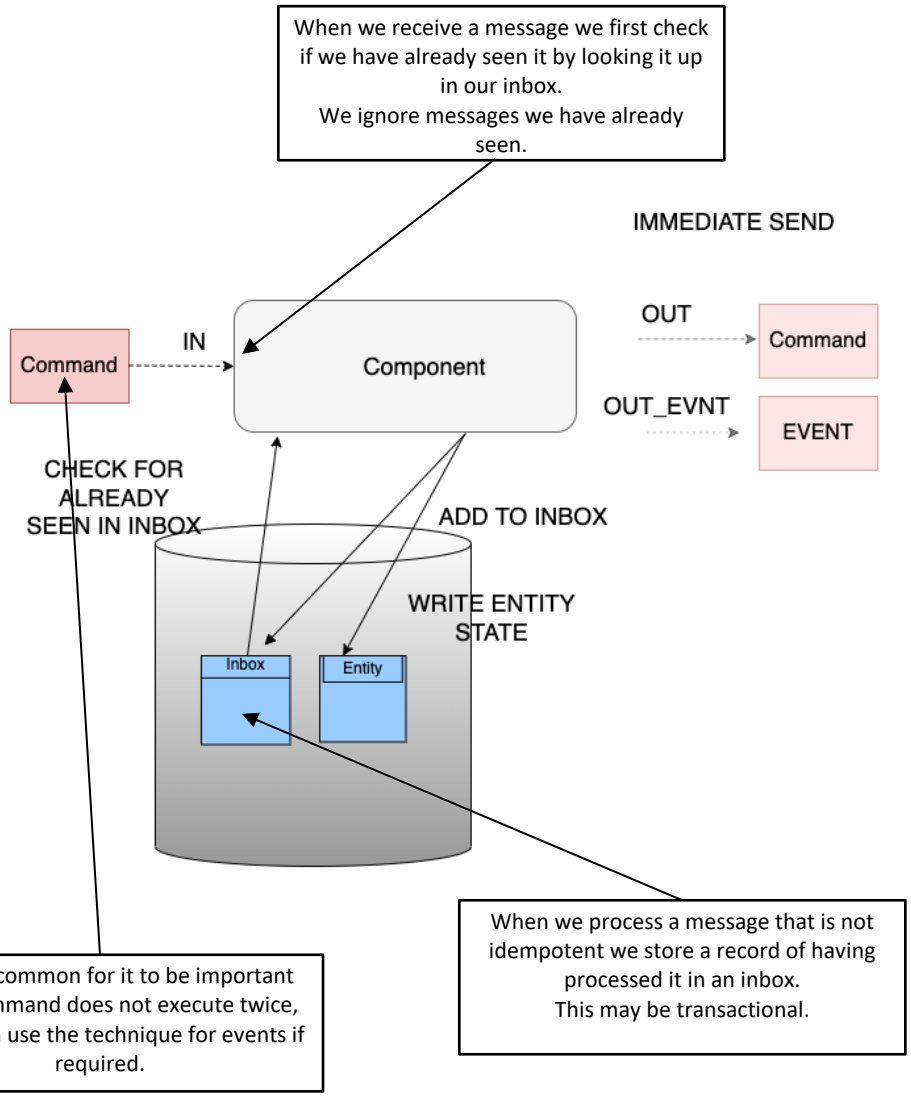
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Inbox



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber
- Actionable, Independent, Discrete

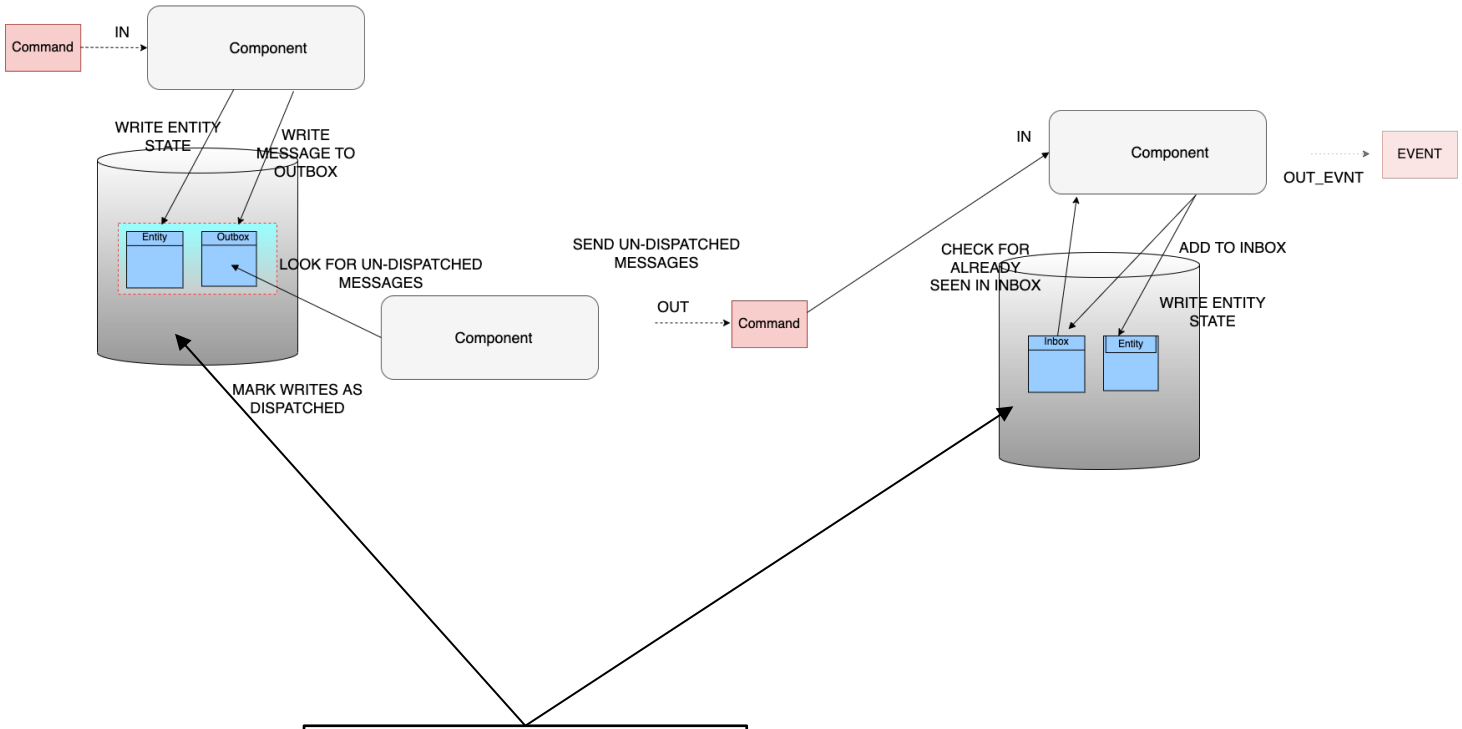
Event

Event
- An Information Packet
- Zero or More Subscribers
- Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

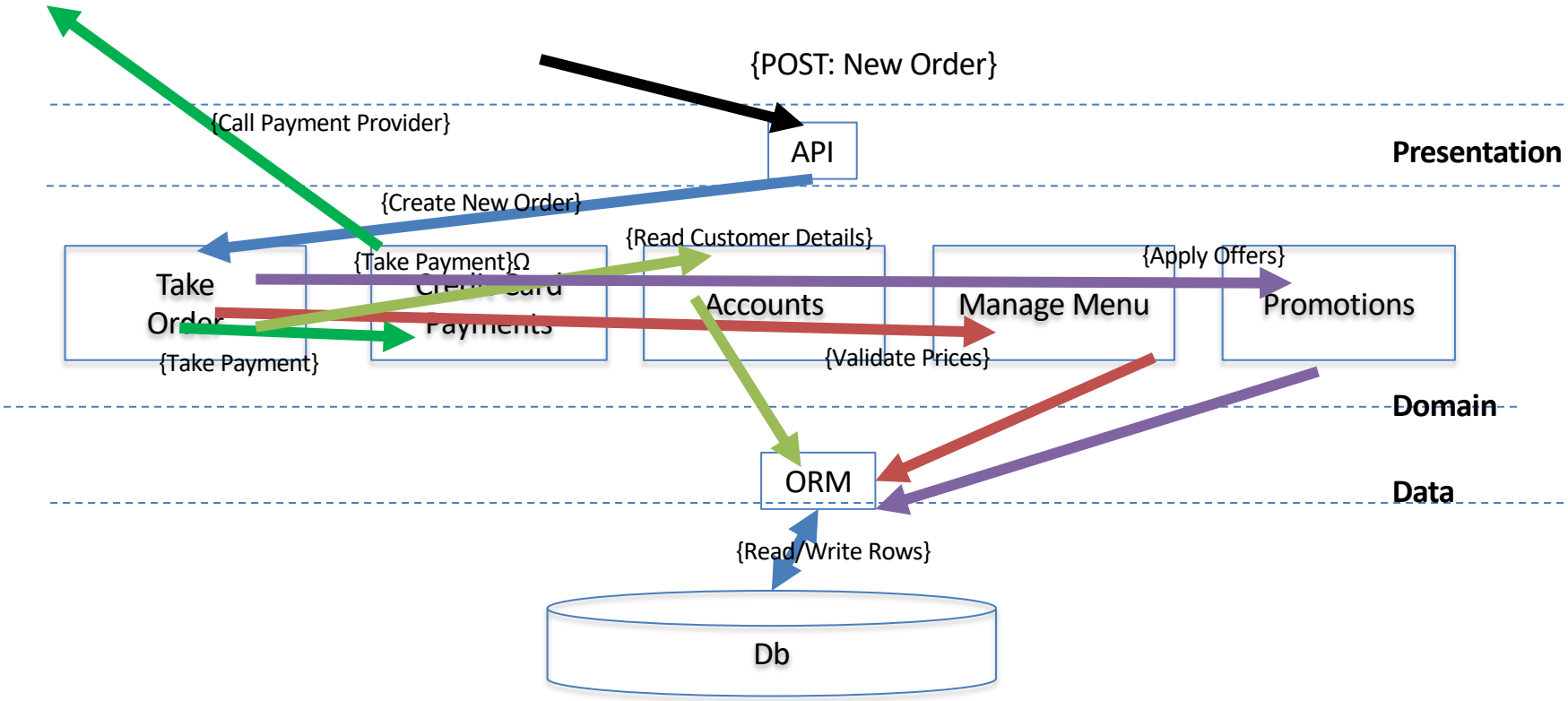
Outbox and Inbox



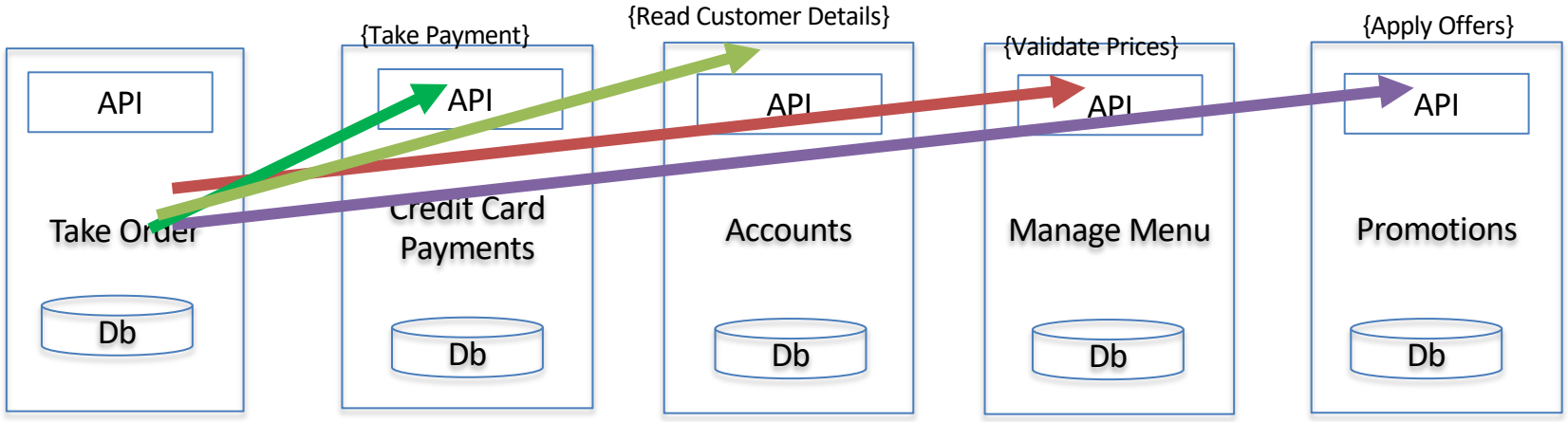
The Outbox and Inbox work together to create **guaranteed once-only delivery**.

4 EVENT DRIVEN COLLABORATION

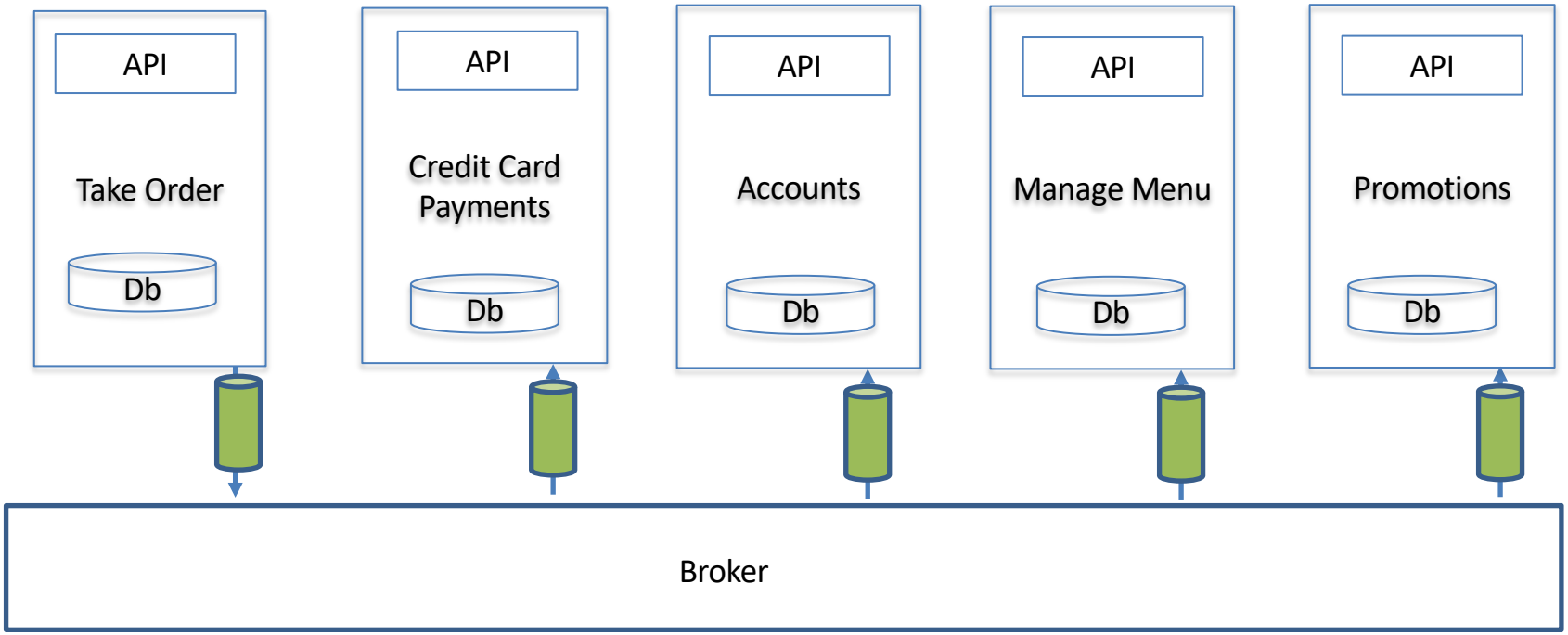
Getting work done in a Monolith



Microservices



Event Driven Architecture

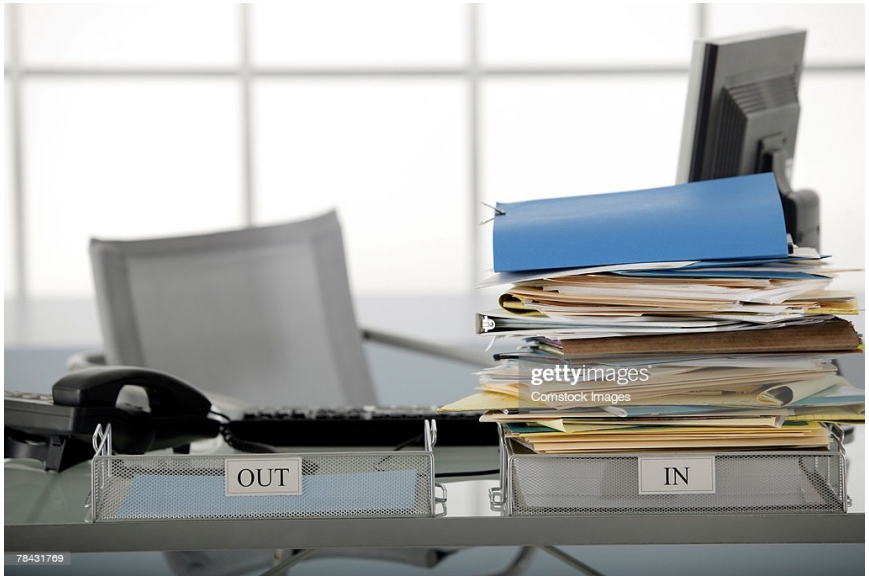


Paper Workflows

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

(Message)Event – Skinny, Notification

(Message)Document – Fat


OUT
IN
gettyimages
Comstock Images
78431769

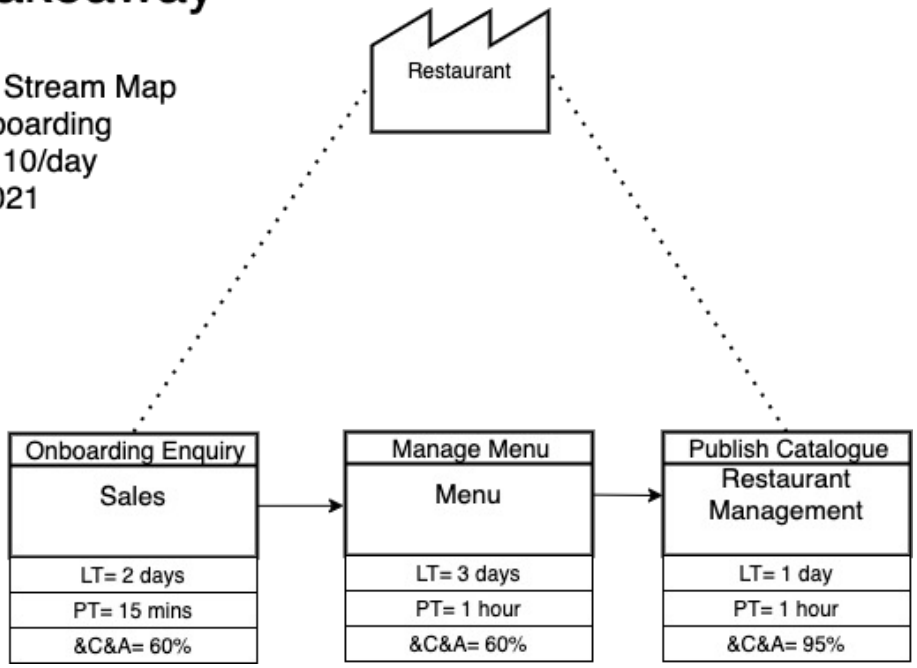
What is a microservice?

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*.

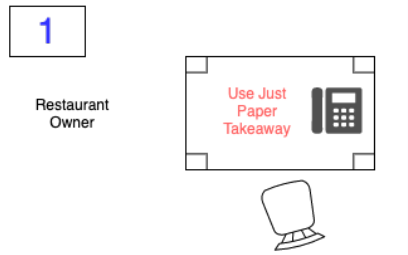
Josuttis, Nicolai M.. SOA in Practice: The Art of Distributed System Design . O'Reilly Media. Kindle Edition.

Just Paper Takeaway

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



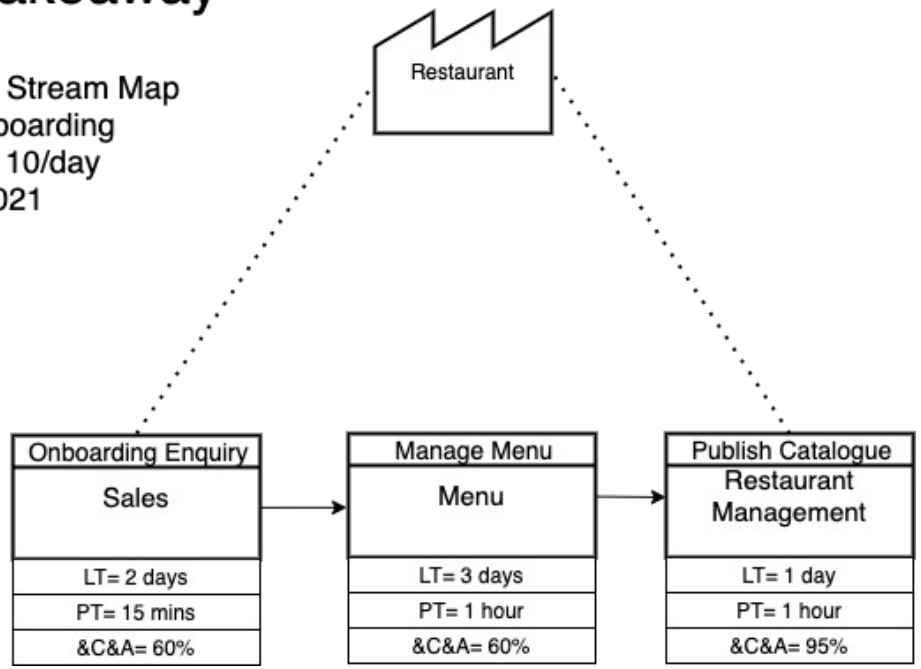
JUST EAT Takeaway.com

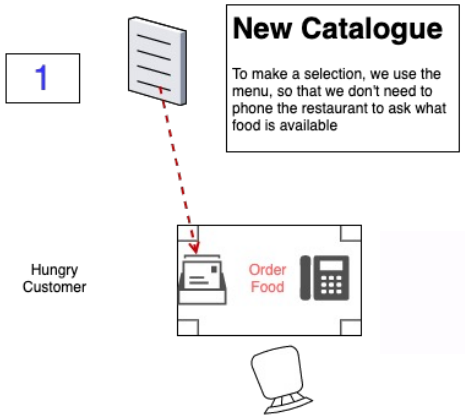


Restaurant Onboarding

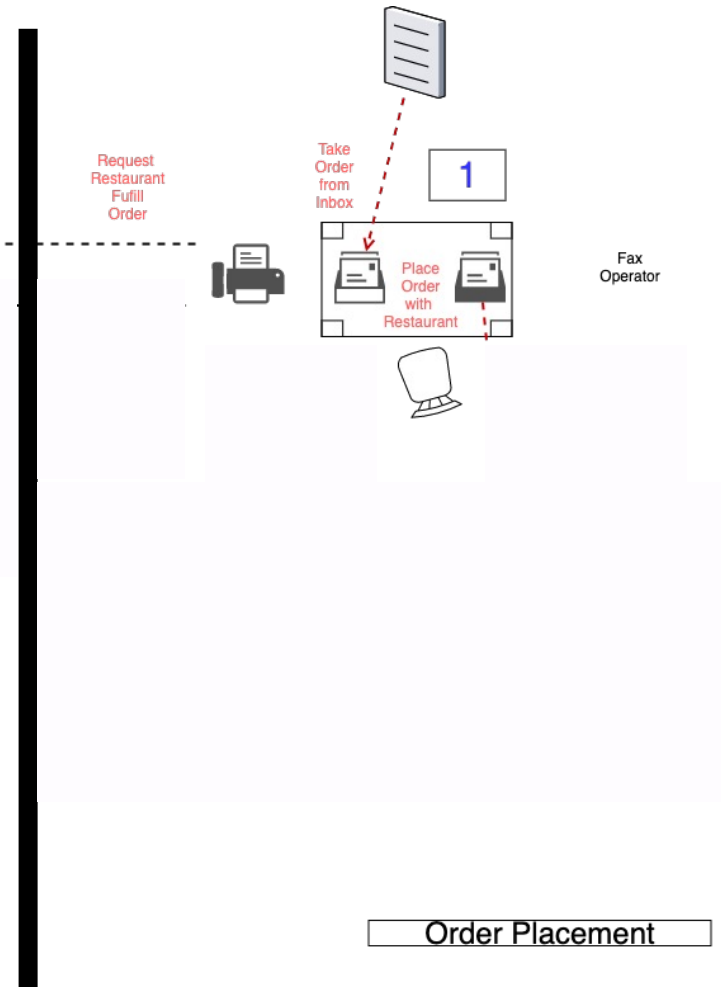
Just Paper Takeaway

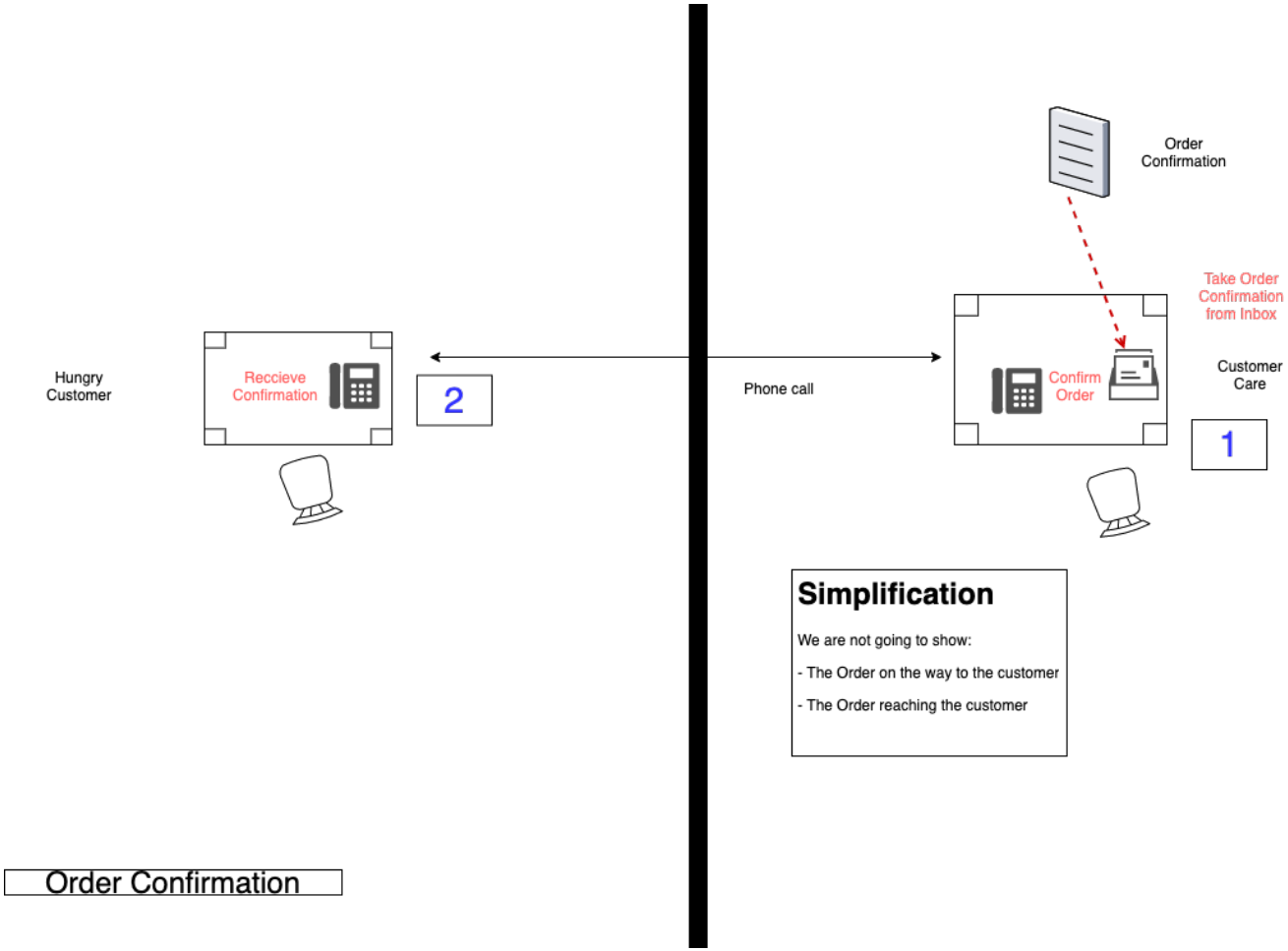
Current State Value Stream Map
Restaurant Onboarding
Demand Rate 10/day
29 SEP 2021

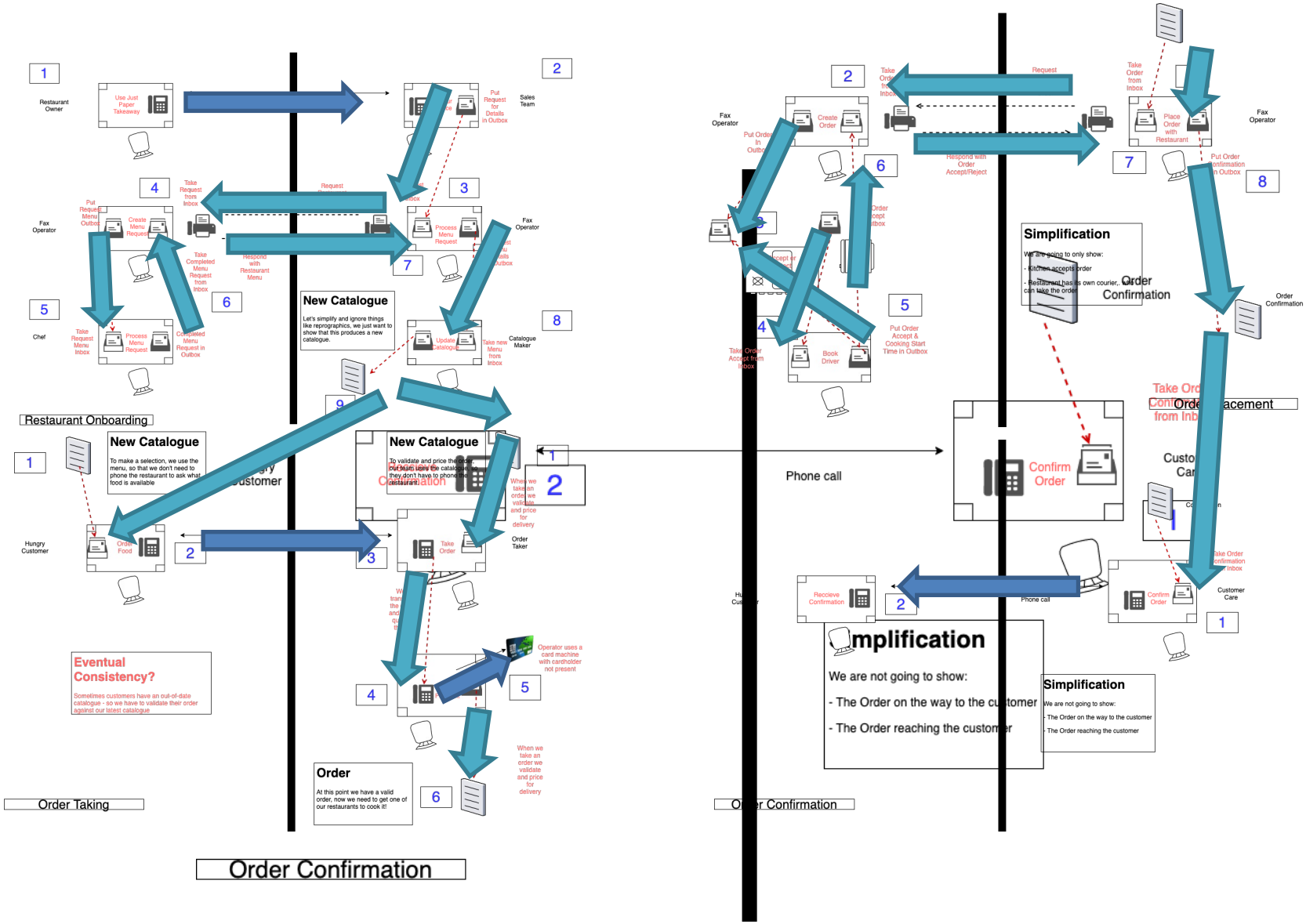




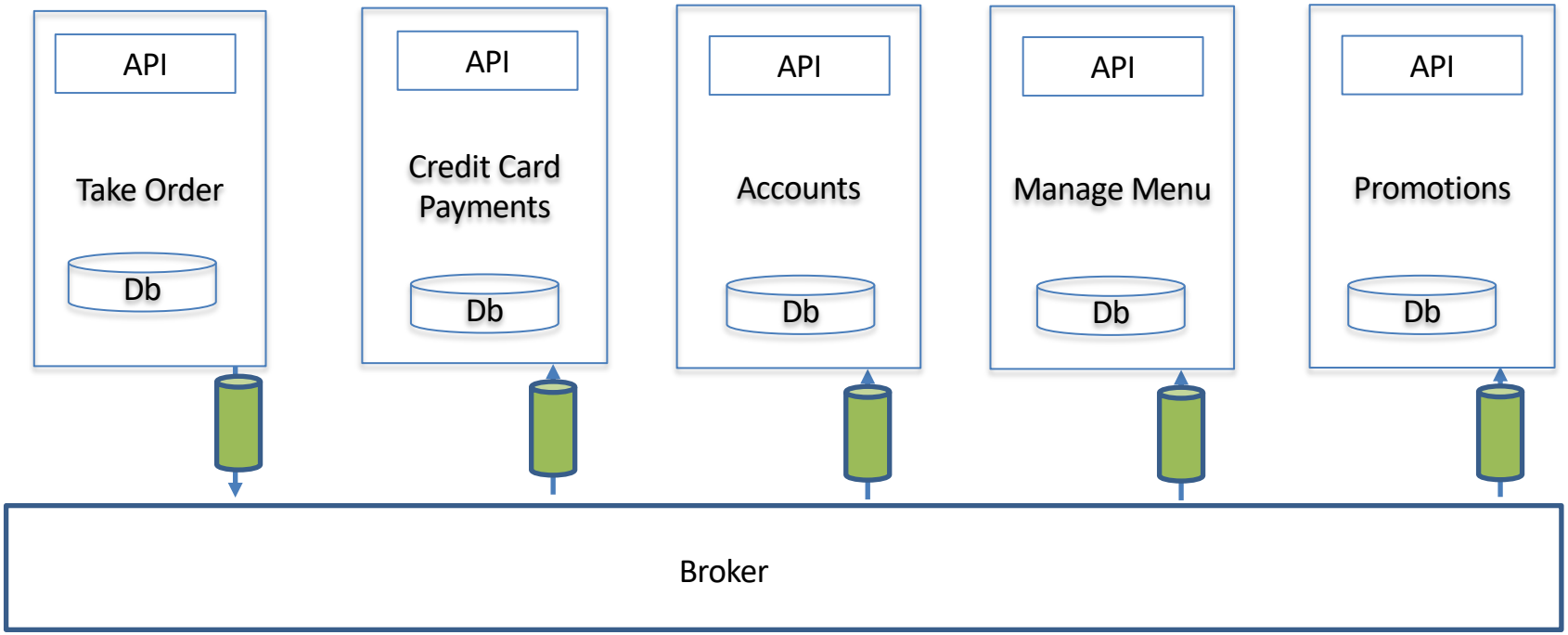
Order Taking


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

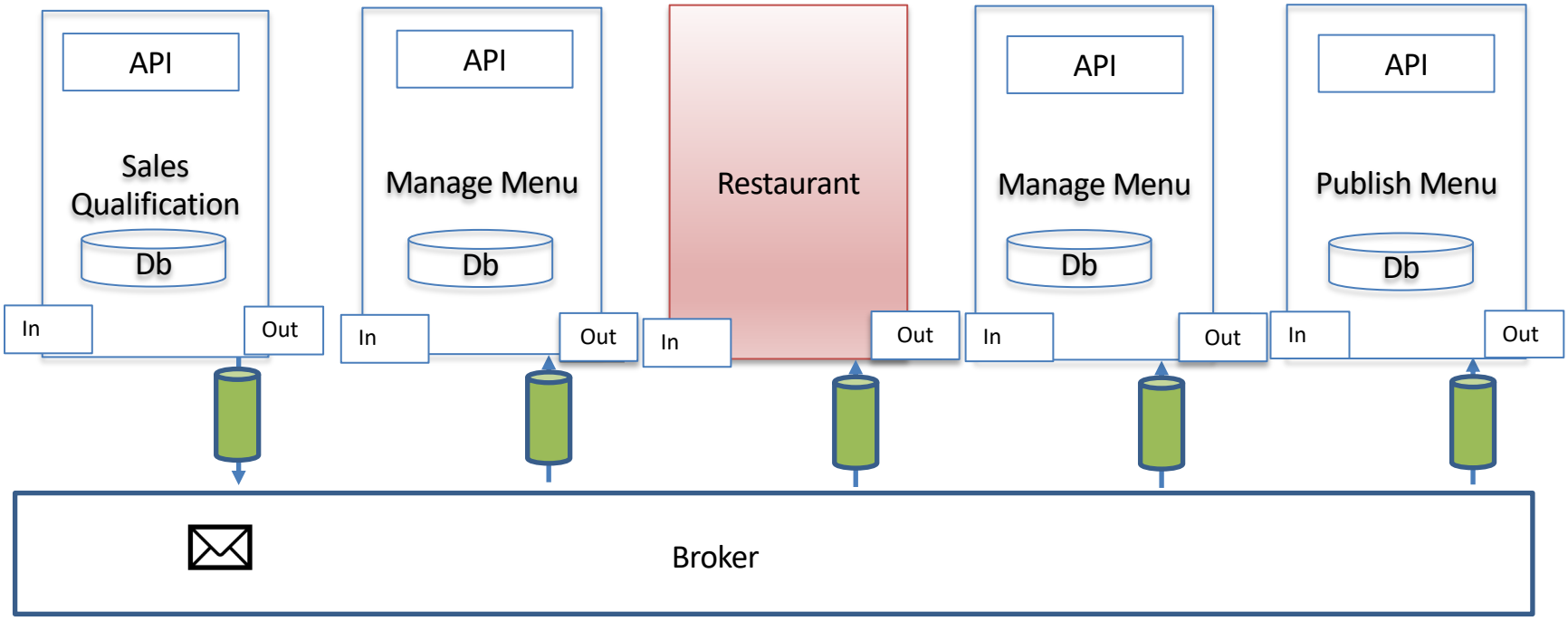



Restaurant Owner
Use Just Paper Takeaway
Sales Team
Put Request for Details in Outbox
Fax Operator
Put Request Menu Outbox
Create Menu Request
Take Request from Inbox
Request
Process Menu Request
Respond with Restaurant Menu
Take Completed Menu Request from Inbox
Chef
Take Request Menu Inbox
Process 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

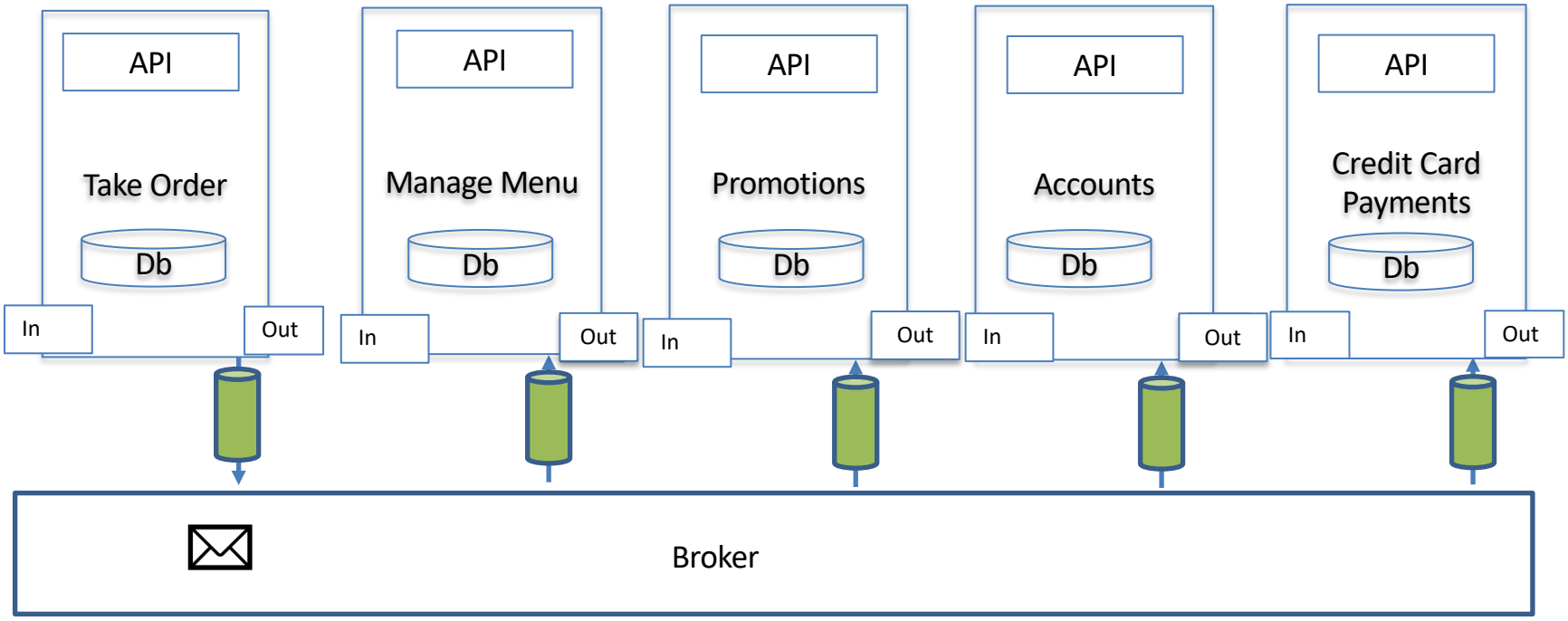
Event Driven Architecture



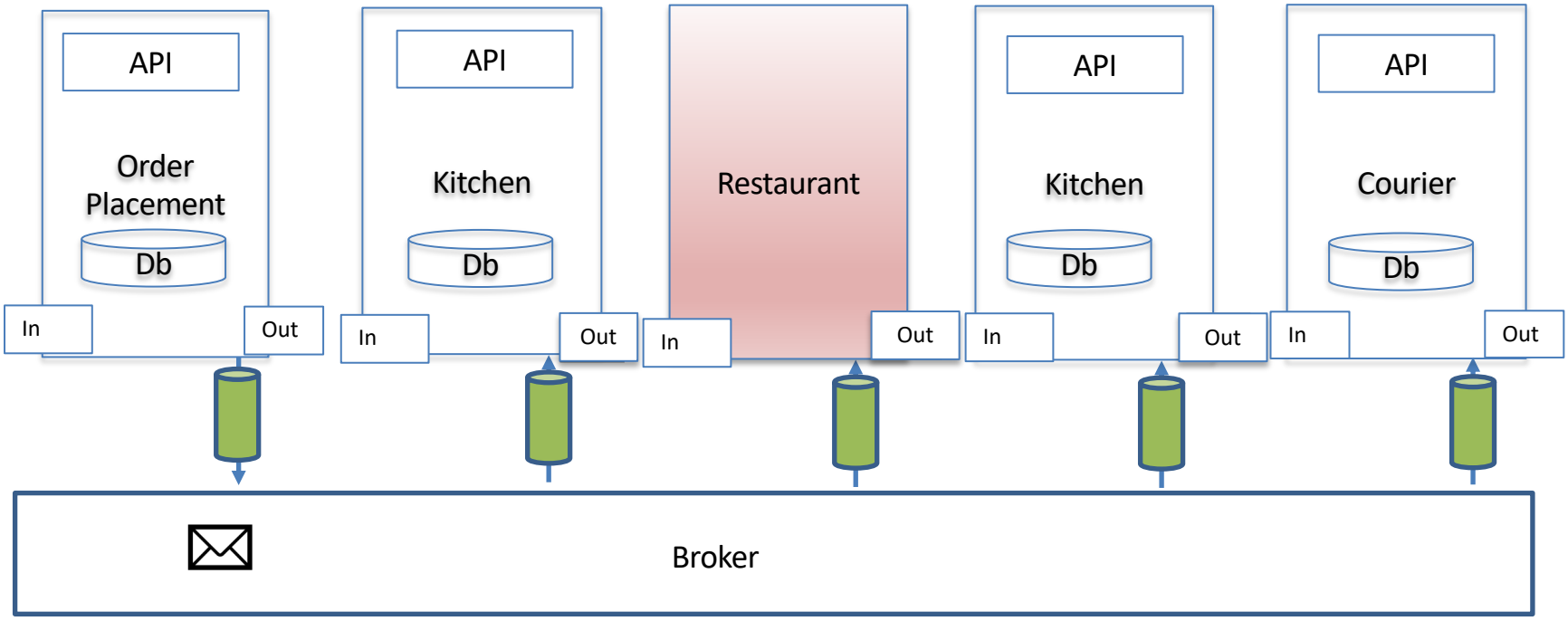
Onboard Restaurant



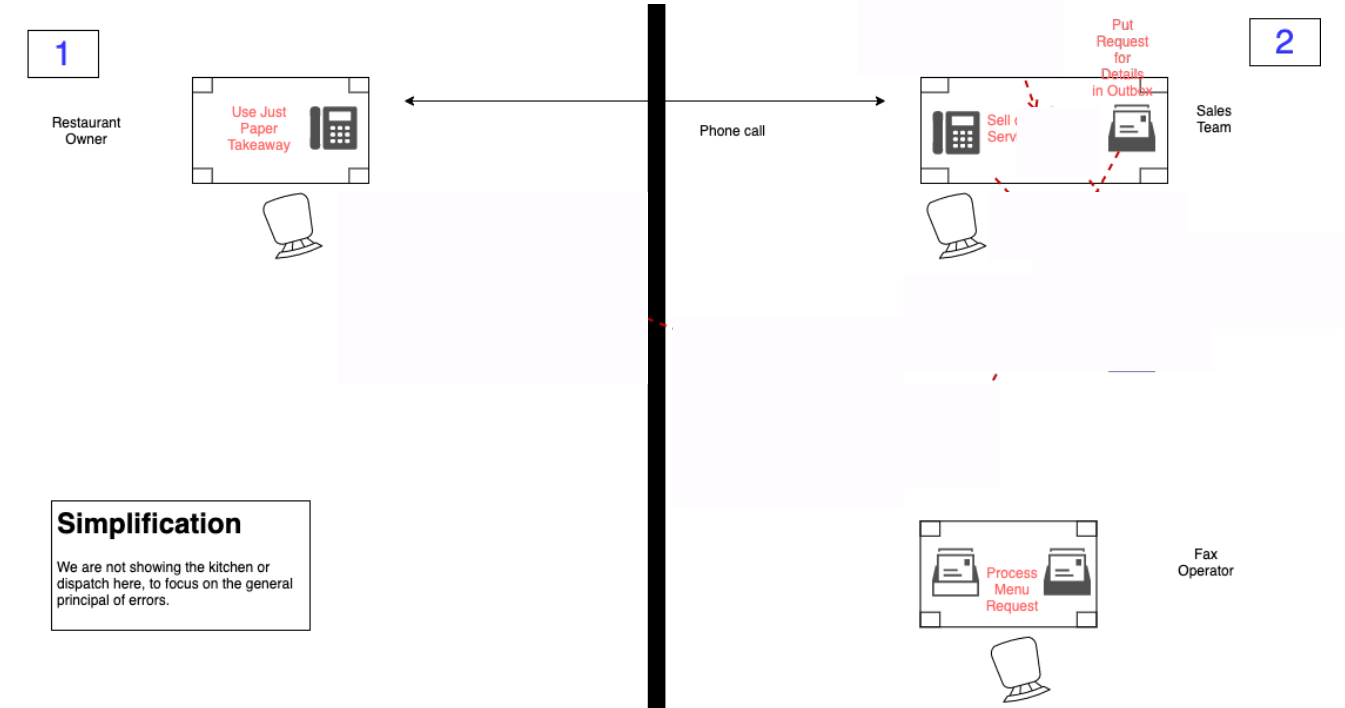
Take Order

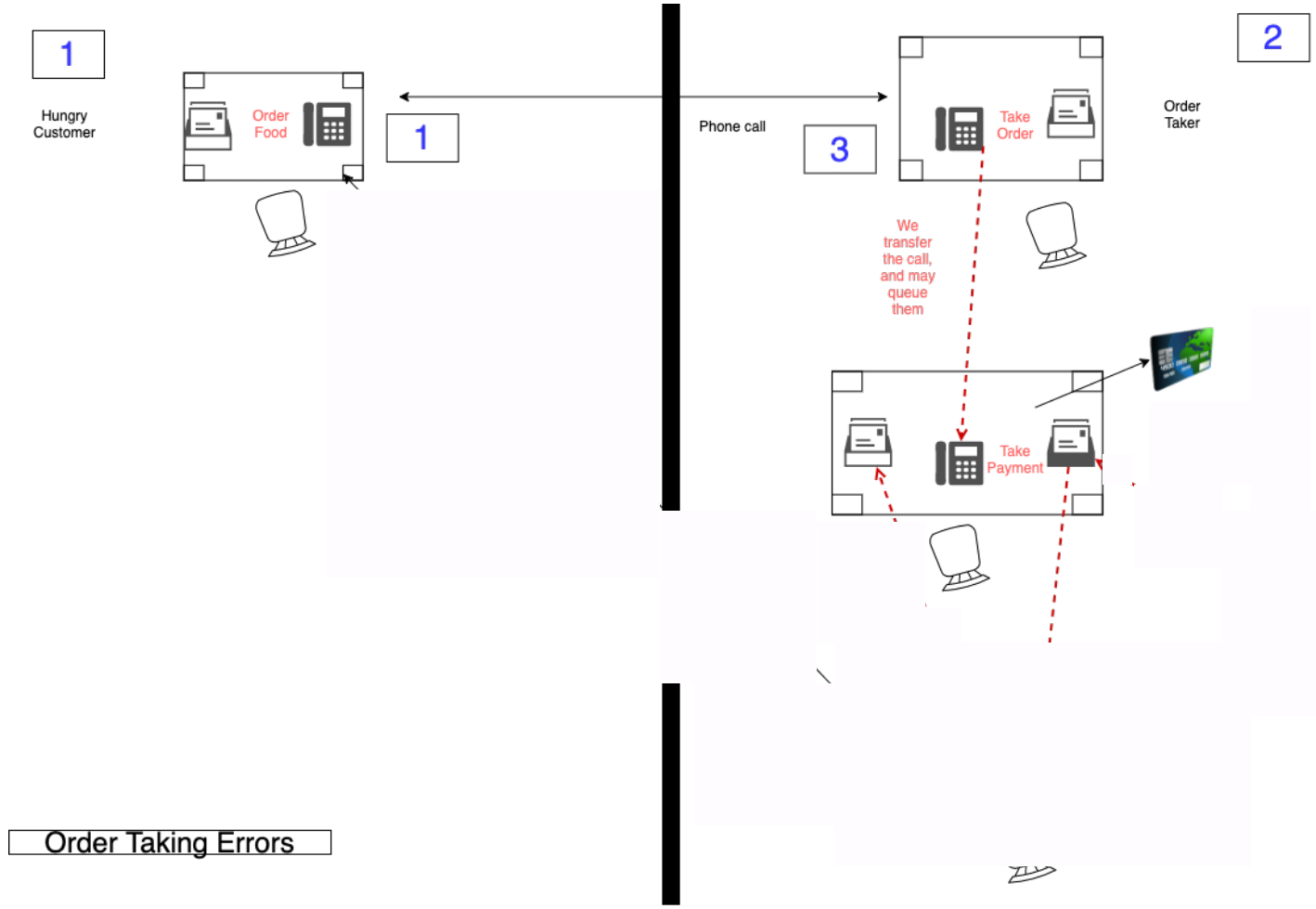

API
Take Order
Db
In
Out
API
Manage Menu
Db
In
Out
API
Promotions
Db
In
Out
API
Accounts
Db
In
Out
API
Credit Card Payments
Db
In
Out
Broker

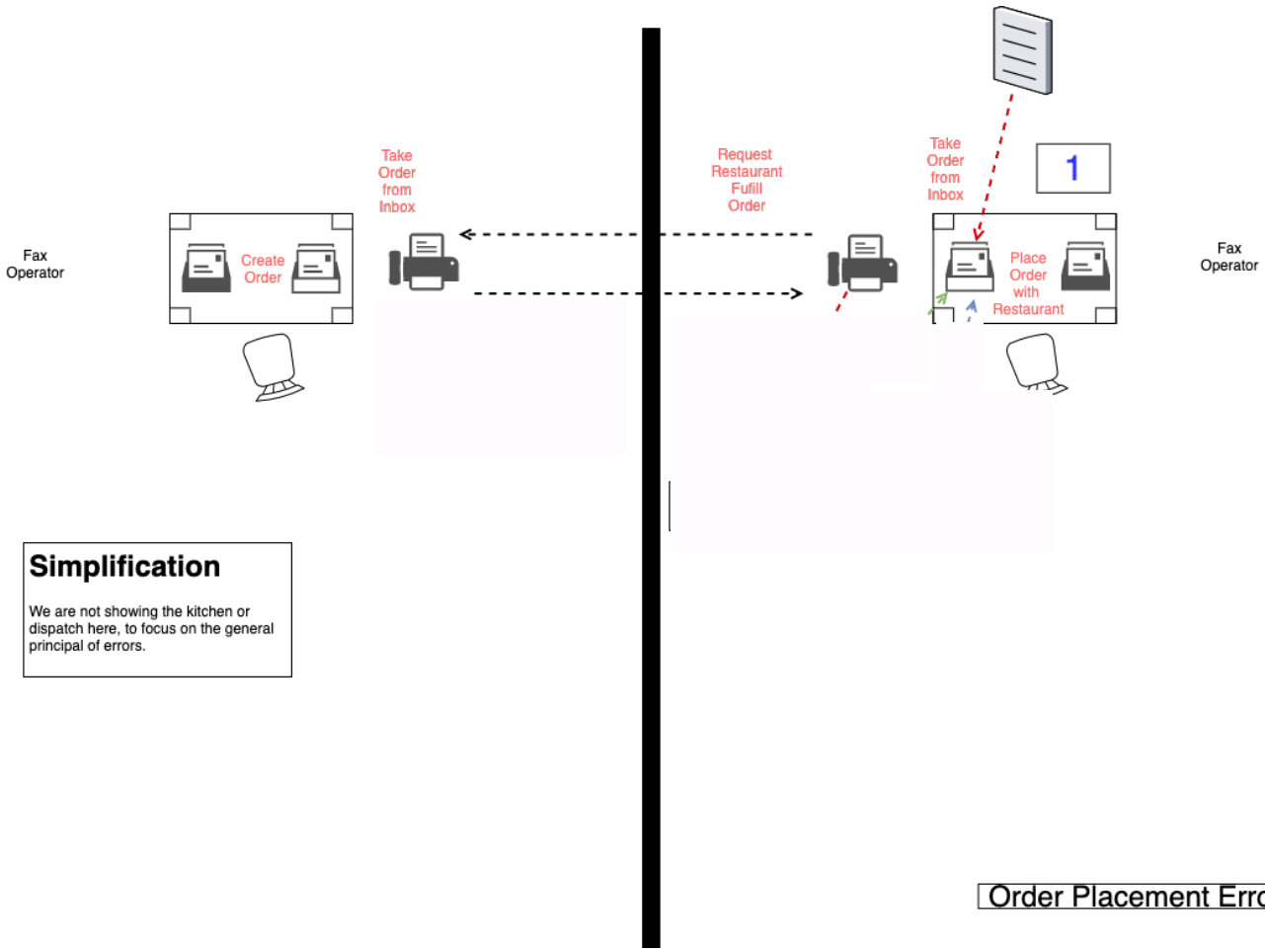
Order Placement



Compensation

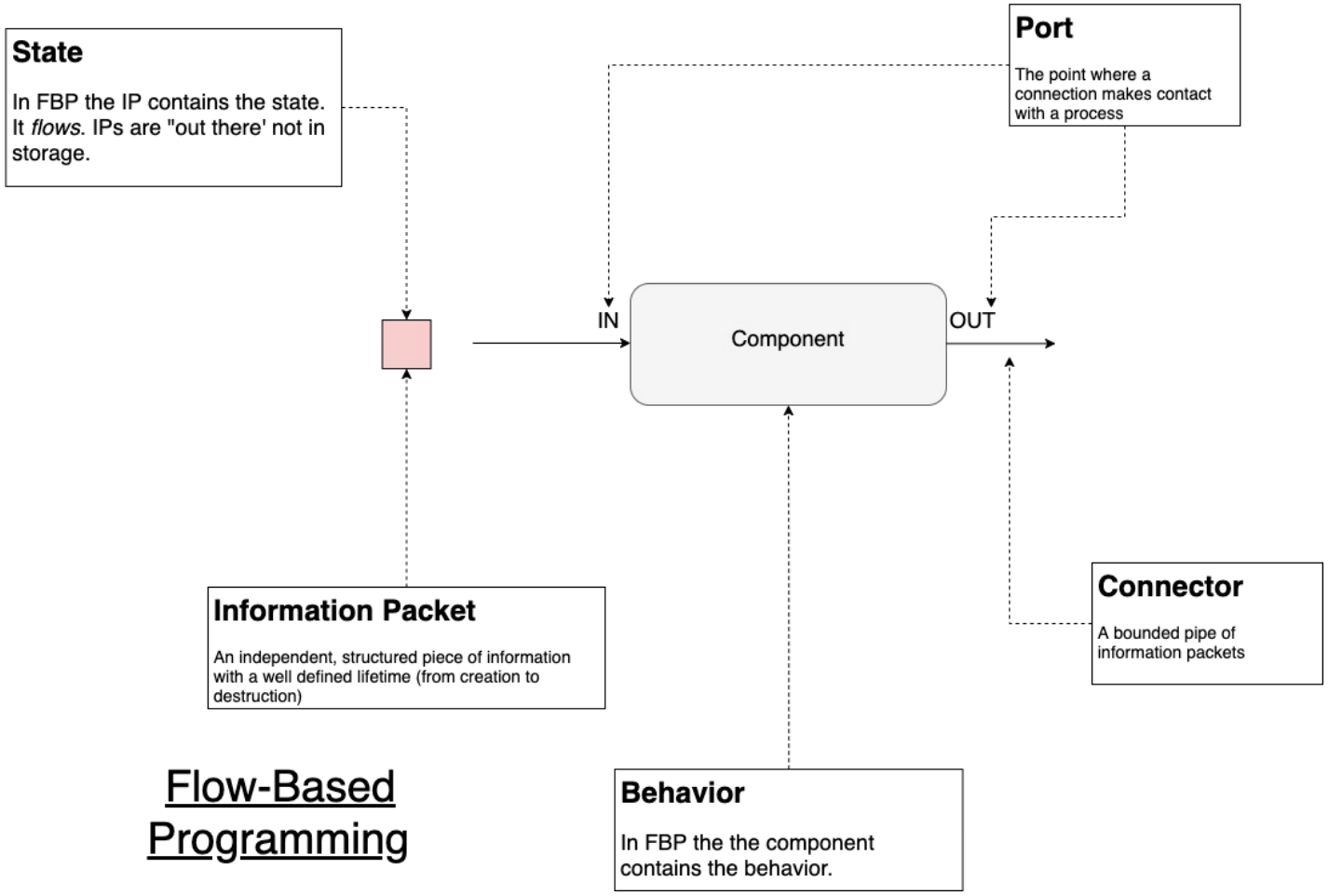

1
Restaurant Owner
Use Just Paper Takeaway
Phone call
Put Request for Details in Outbox
2
Sales Team
Simplification
We are not showing the kitchen or dispatch here, to focus on the general principal of errors.
Process Menu Request
Fax Operator
Restaurant Onboarding Errors

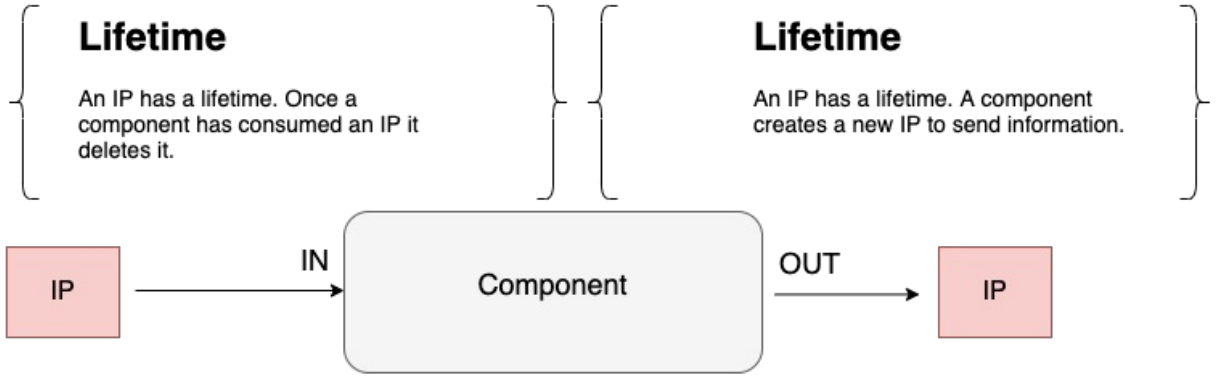

1
Hungry Customer
Order Food
1
Phone call
3
2
Take Order
Order Taker
We transfer the call, and may queue them
Take Payment
Order Taking Errors


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

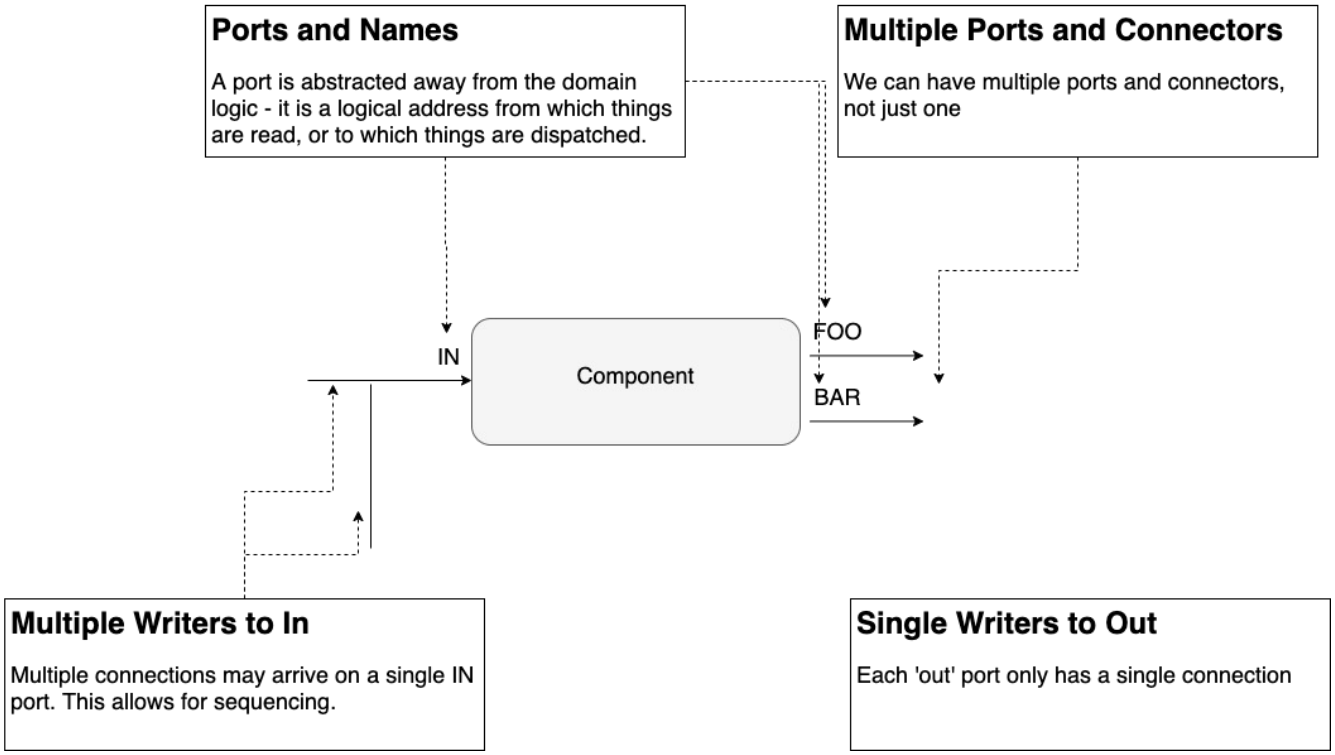
"Πάντα ῥεῖ – Everything flows" (Heraclitus of Ephesus, ca. 500 BCE)

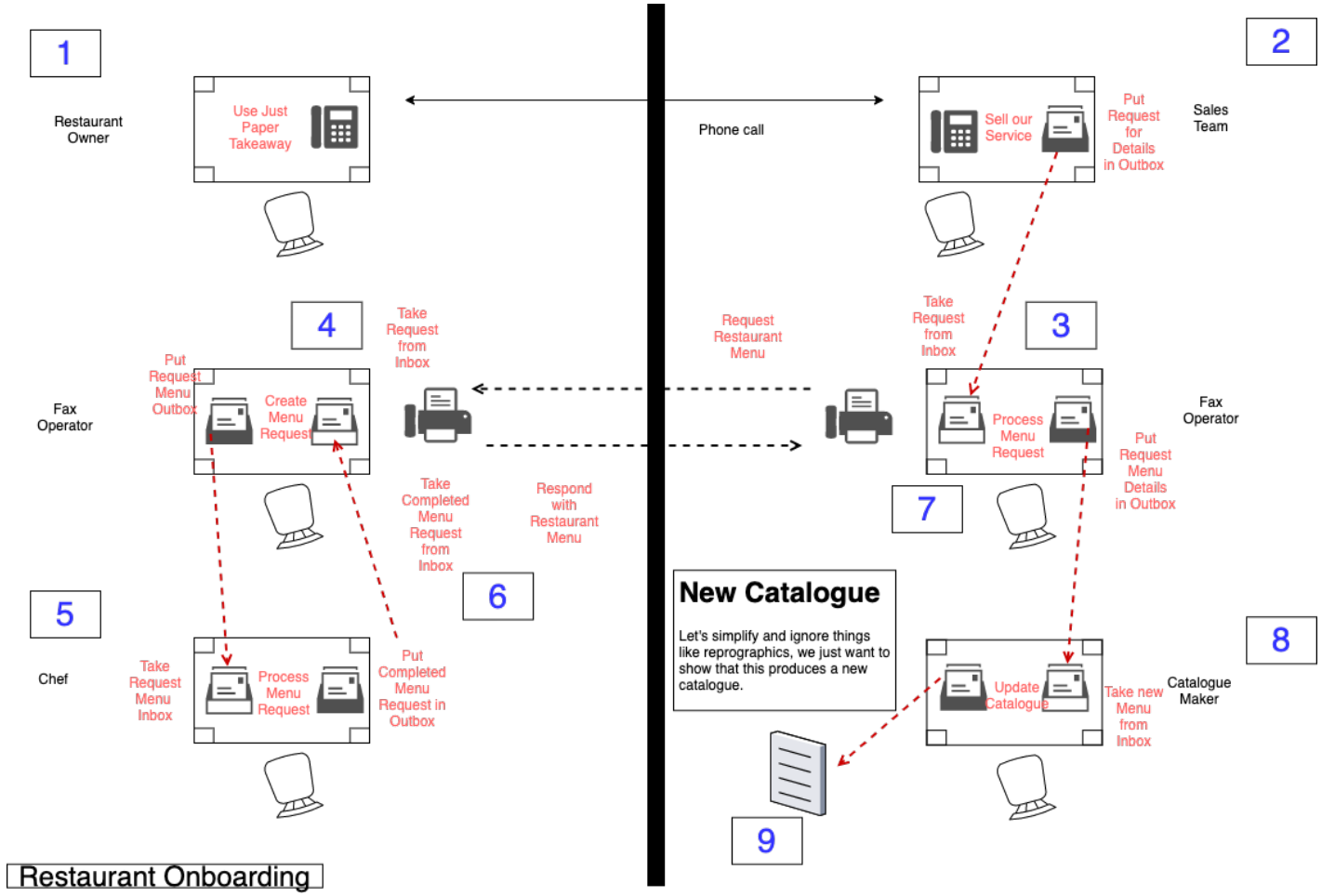

State
In FBP the IP contains the state. It *flows*. IPs are "out there' not in storage.
Port
The point where a connection makes contact with a process
IN
Component
OUT
Information Packet
An independent, structured piece of information with a well defined lifetime (from creation to destruction)
Connector
A bounded pipe of information packets
Flow-Based Programming
Behavior
In FBP the the component contains the behavior.

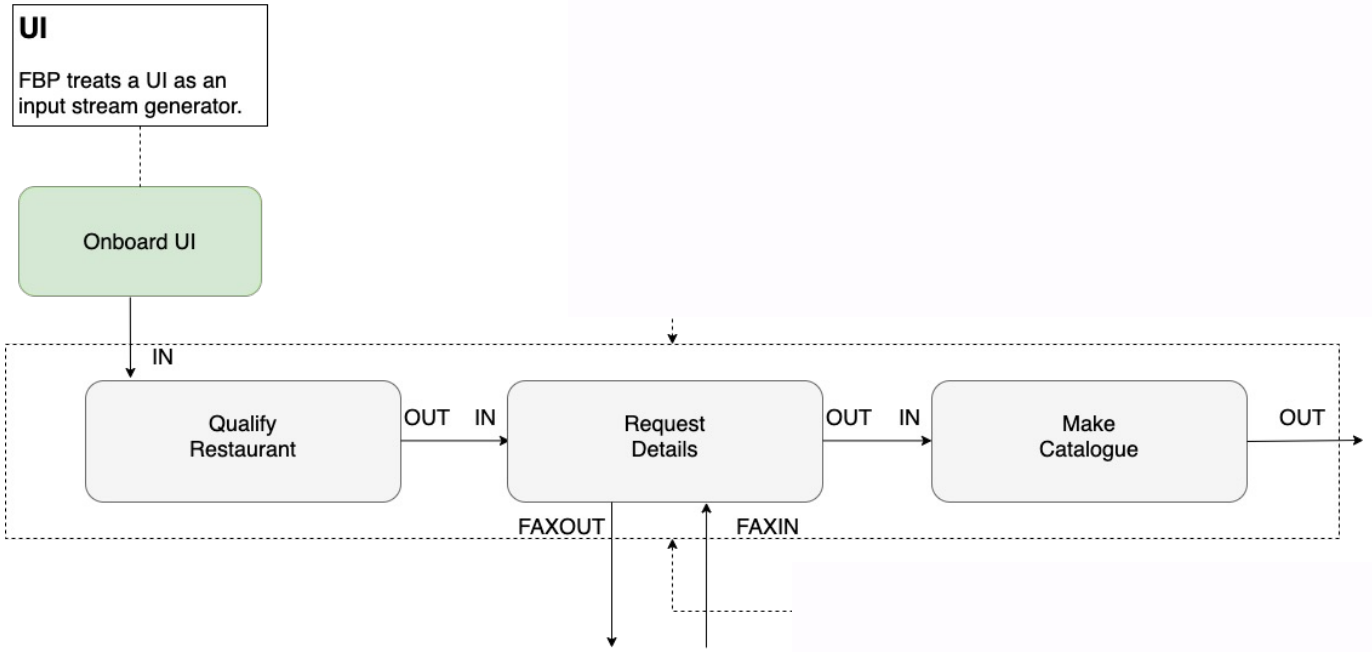


Flow-Based Programming

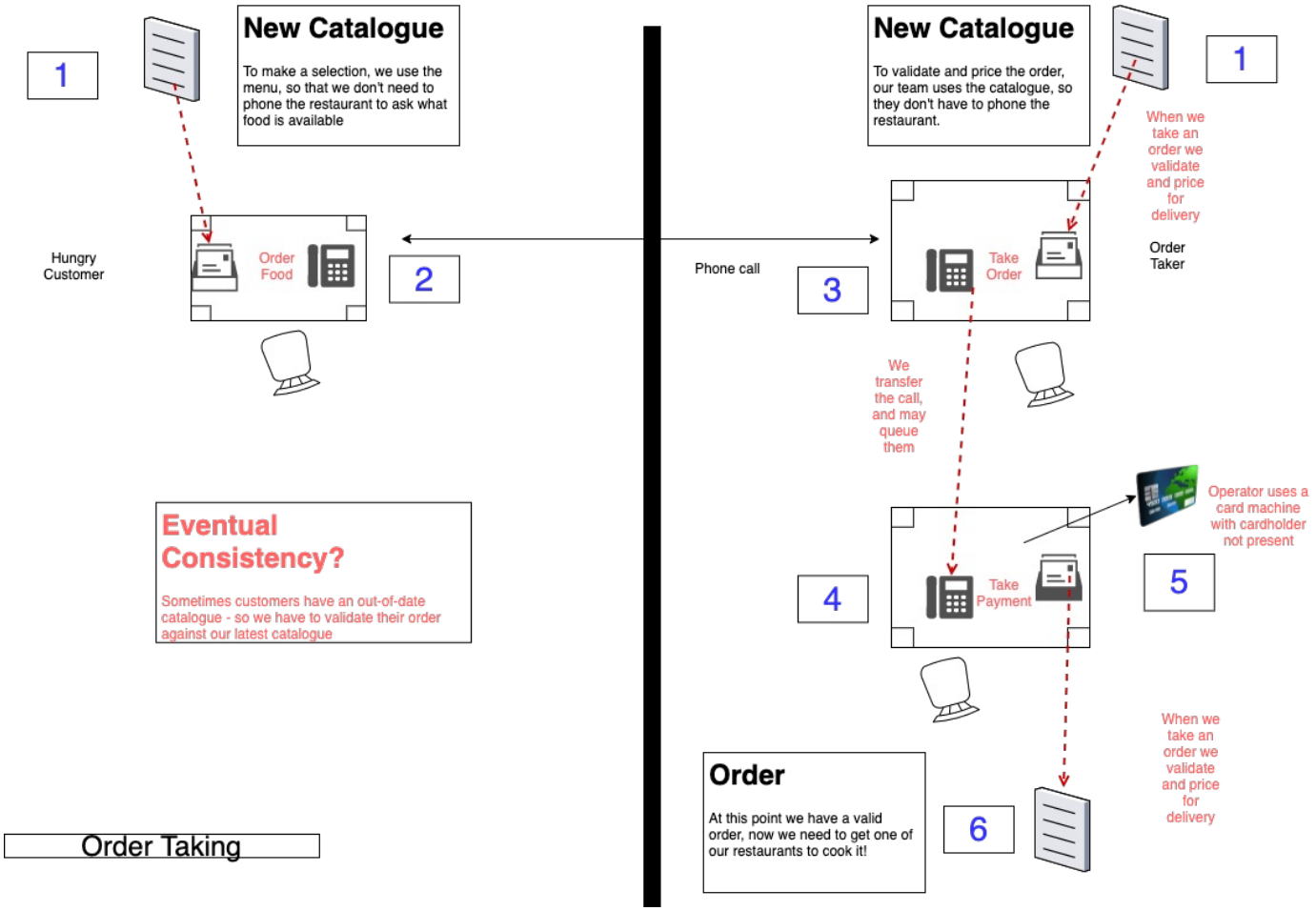
Ports and Connectors

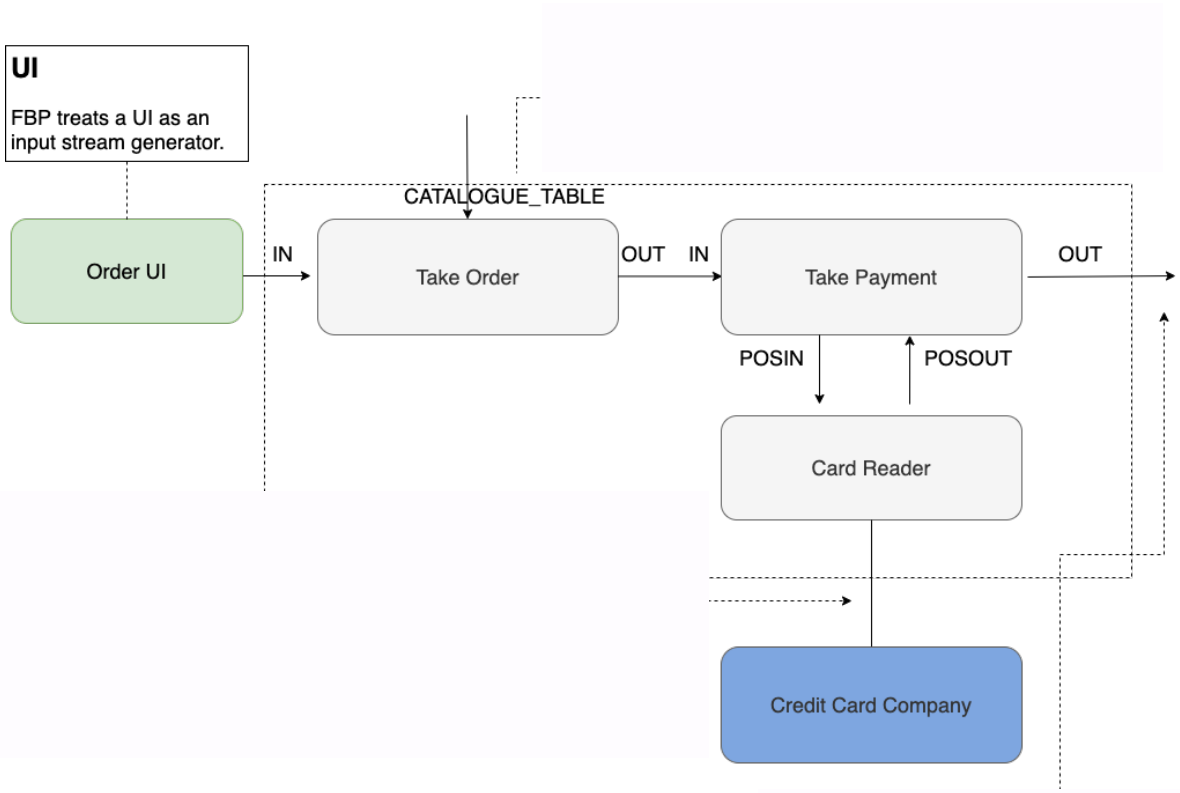



1
Restaurant Owner
Use Just Paper Takeaway
Phone call
2
Sell our Service
Put Request for Details in Outbox
Sales Team
4
Take Request from Inbox
Put Request Menu Outbox
Create Menu Request
Fax Operator
Request Restaurant Menu
Take Request from Inbox
3
Process Menu Request
Put Request Menu Details in Outbox
Fax Operator
Take Completed Menu Request from Inbox
Respond with Restaurant Menu
6
7
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

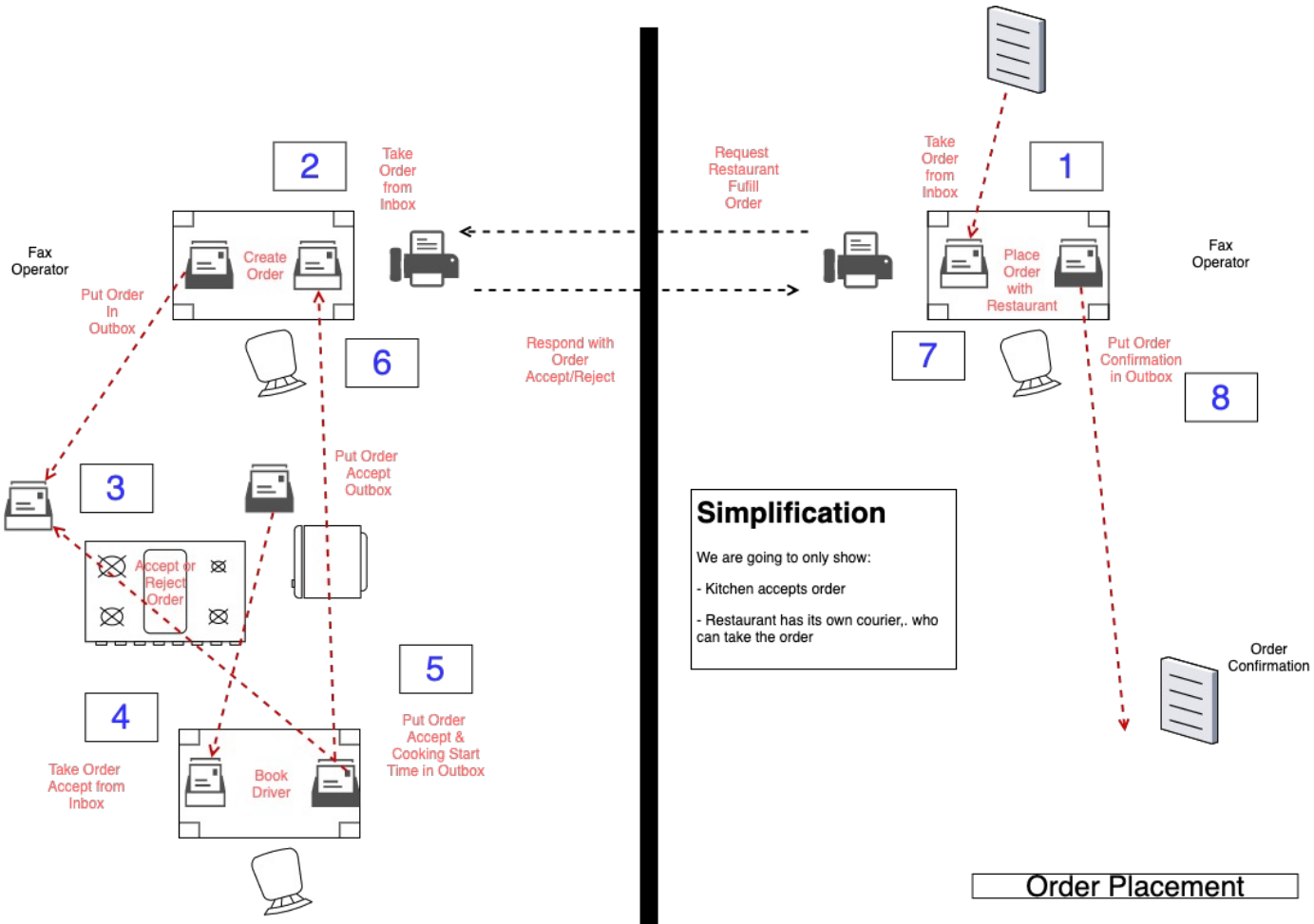


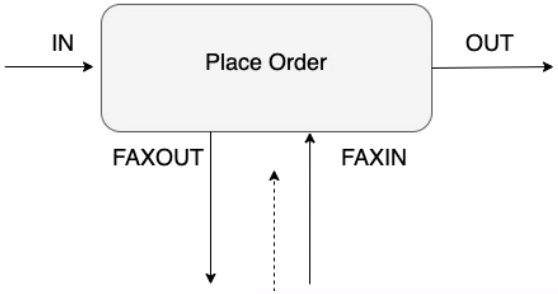
Onboard Restaurant


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

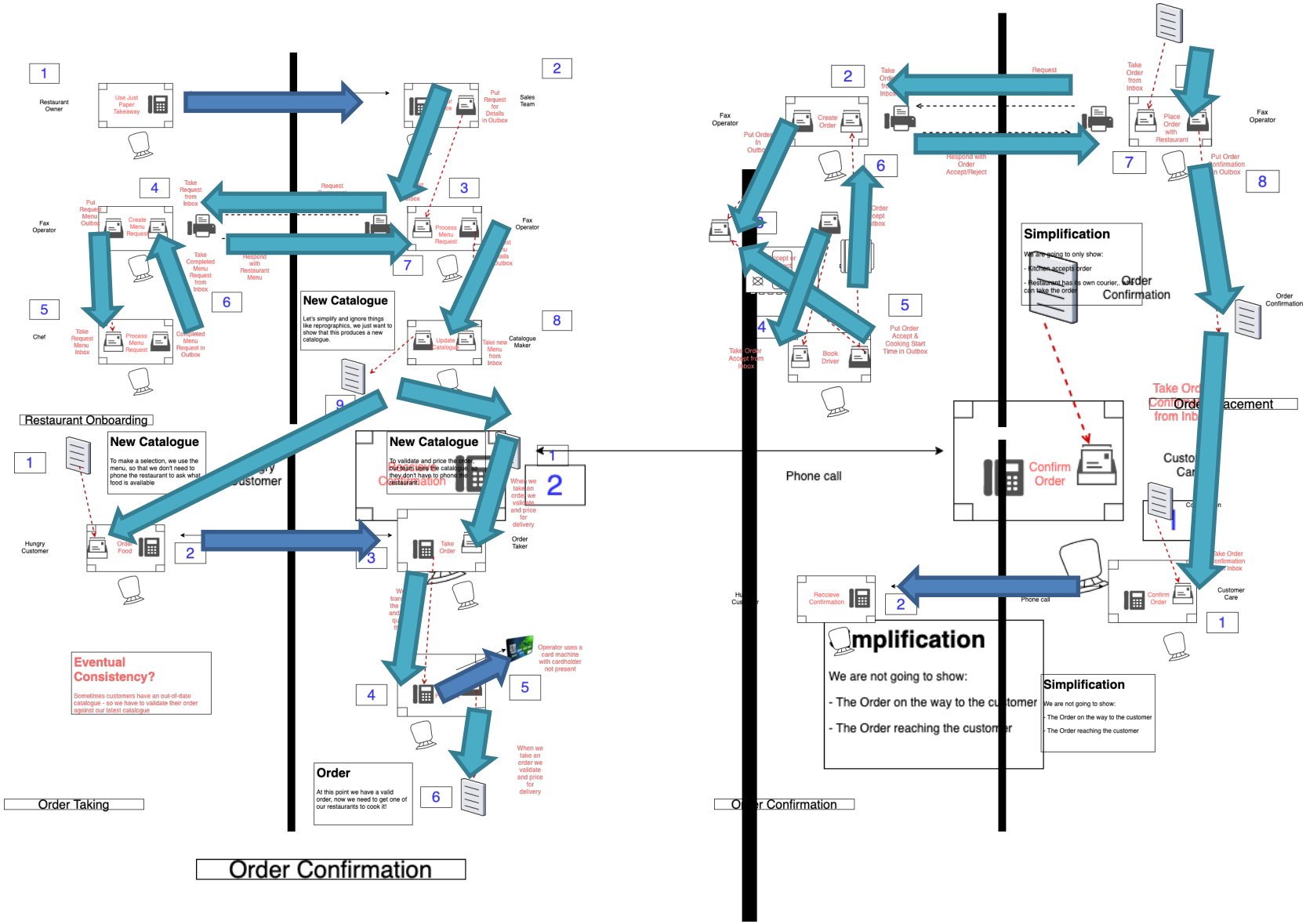

UI
FBP treats a UI as an input stream generator.
CATALOGUE_TABLE
Order UI
IN
Take Order
OUT
IN
Take Payment
OUT
POSIN
POSOUT
Card Reader
Credit Card Company

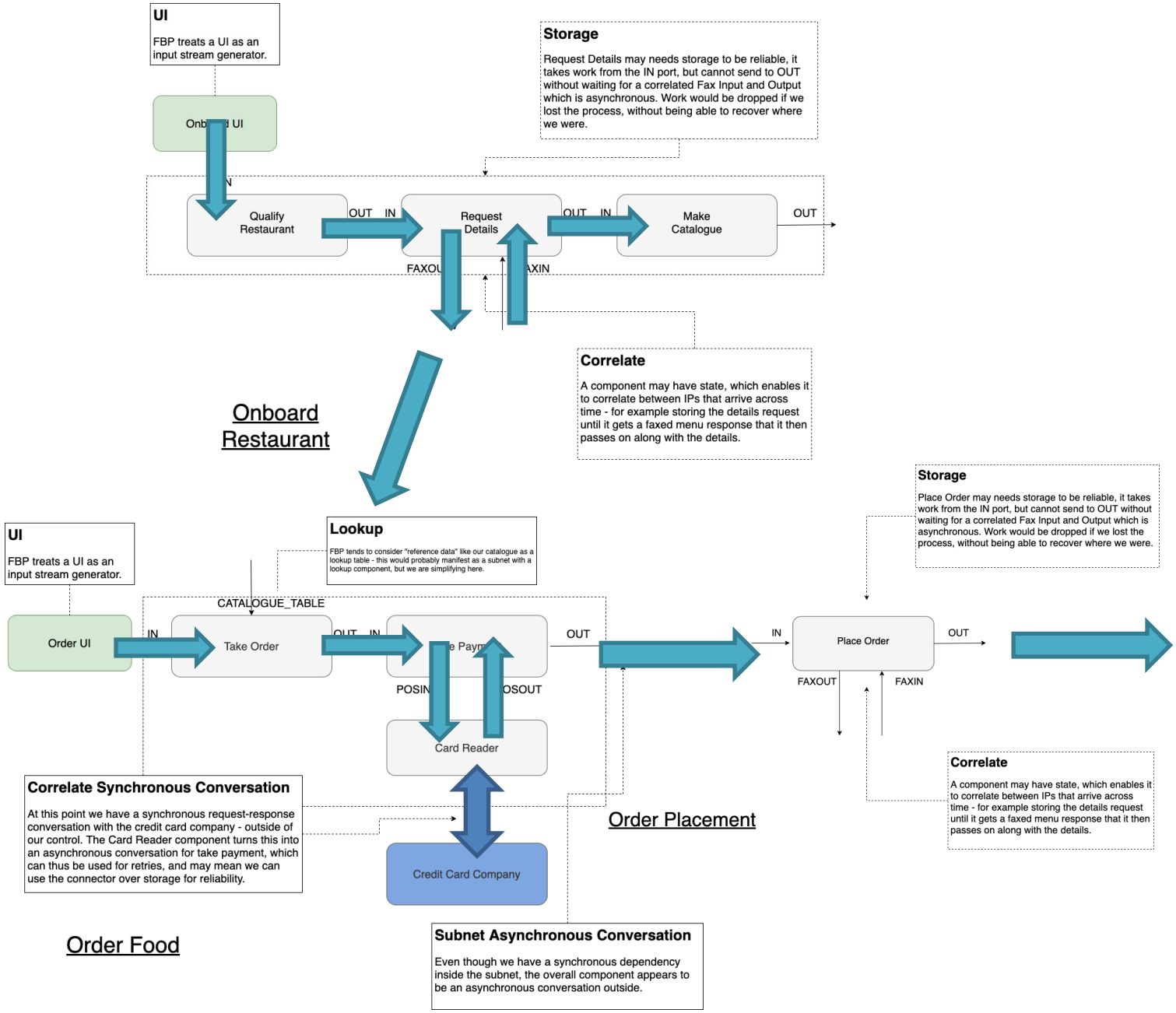
Order Food


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



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


UI
FBP treats a UI as an input stream generator.
Storage
Request Details may needs storage to be reliable, it takes work from the IN port, but cannot send to OUT without waiting for a correlated Fax Input and Output which is asynchronous. Work would be dropped if we lost the process, without being able to recover where we were.
Qualify Restaurant
OUT
IN
Request Details
OUT
IN
Make Catalogue
OUT
Correlate
A component may have state, which enables it to correlate between IPs that arrive across time - for example storing the details request until it gets a faxed menu response that it then passes on along with the details.
Onboard Restaurant
Storage
Place Order may needs storage to be reliable, it takes work from the IN port, but cannot send to OUT without waiting for a correlated Fax Input and Output which is asynchronous. Work would be dropped if we lost the process, without being able to recover where we were.
UI
FBP treats a UI as an input stream generator.
Lookup
FBP tends to consider "reference data" like our catalogue as a lookup table - this would probably manifest as a subnet with a lookup component, but we are simplifying here.
CATALOGUE_TABLE
Order UI
IN
Take Order
OUT
IN
OUT
IN
Place Order
OUT
FAXOUT
FAXIN
Card Reader
Correlate
A component may have state, which enables it to correlate between IPs that arrive across time - for example storing the details request until it gets a faxed menu response that it then passes on along with the details.
Correlate Synchronous Conversation
At this point we have a synchronous request-response conversation with the credit card company - outside of our control. The Card Reader component turns this into an asynchronous conversation for take payment, which can thus be used for retries, and may mean we can use the connector over storage for reliability.
Order Placement
Credit Card Company
Order Food
Subnet Asynchronous Conversation
Even though we have a synchronous dependency inside the subnet, the overall component appears to be an asynchronous conversation outside.

Choreography

Orchestration

Flow-Based Commands

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Process and Acknowledge Work
-Raise New Work

Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete

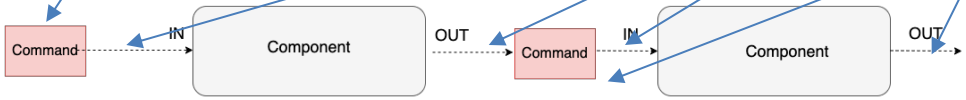
Command

A Command is an Imperative. An instruction to another party to do something.
It is usually issued as part of a workflow.

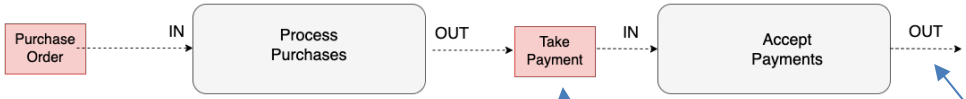
A useful question is: is there another service I *expect* to handle this request?
If so you have a Command.

A Command is Discrete.
We use a Queue for a Command

A Command may be *fire-and-forget* or *request-reply*



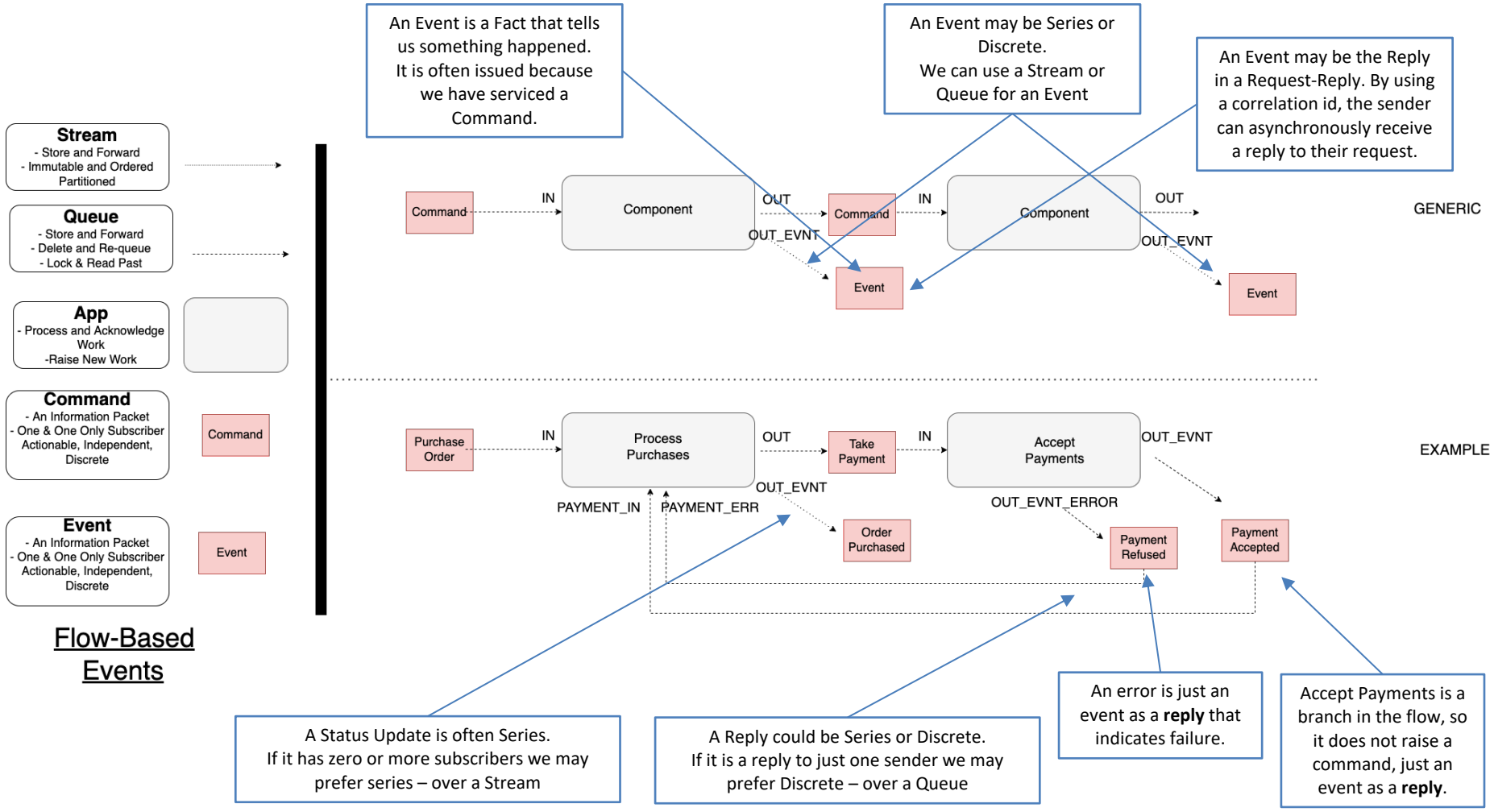
GENERIC

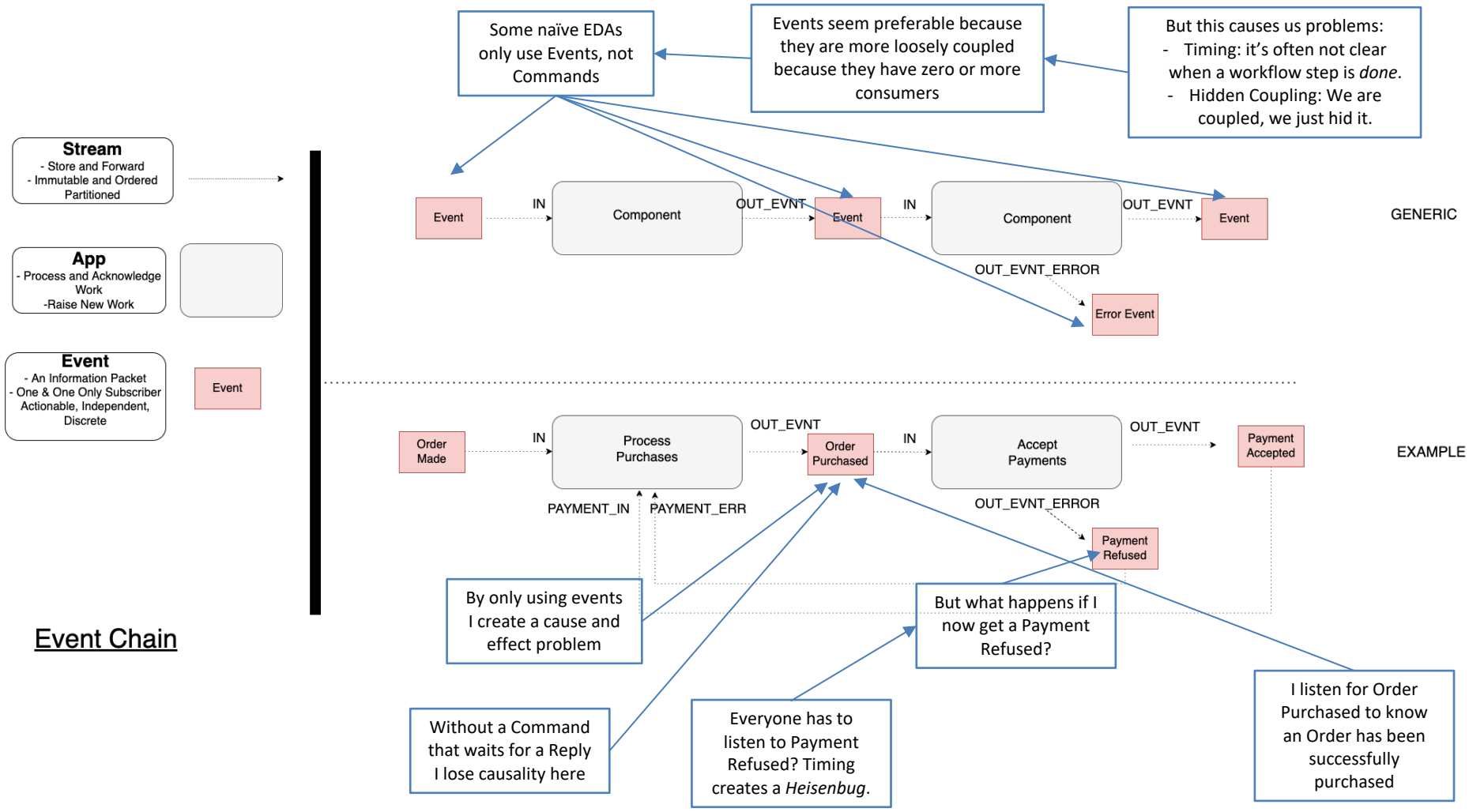


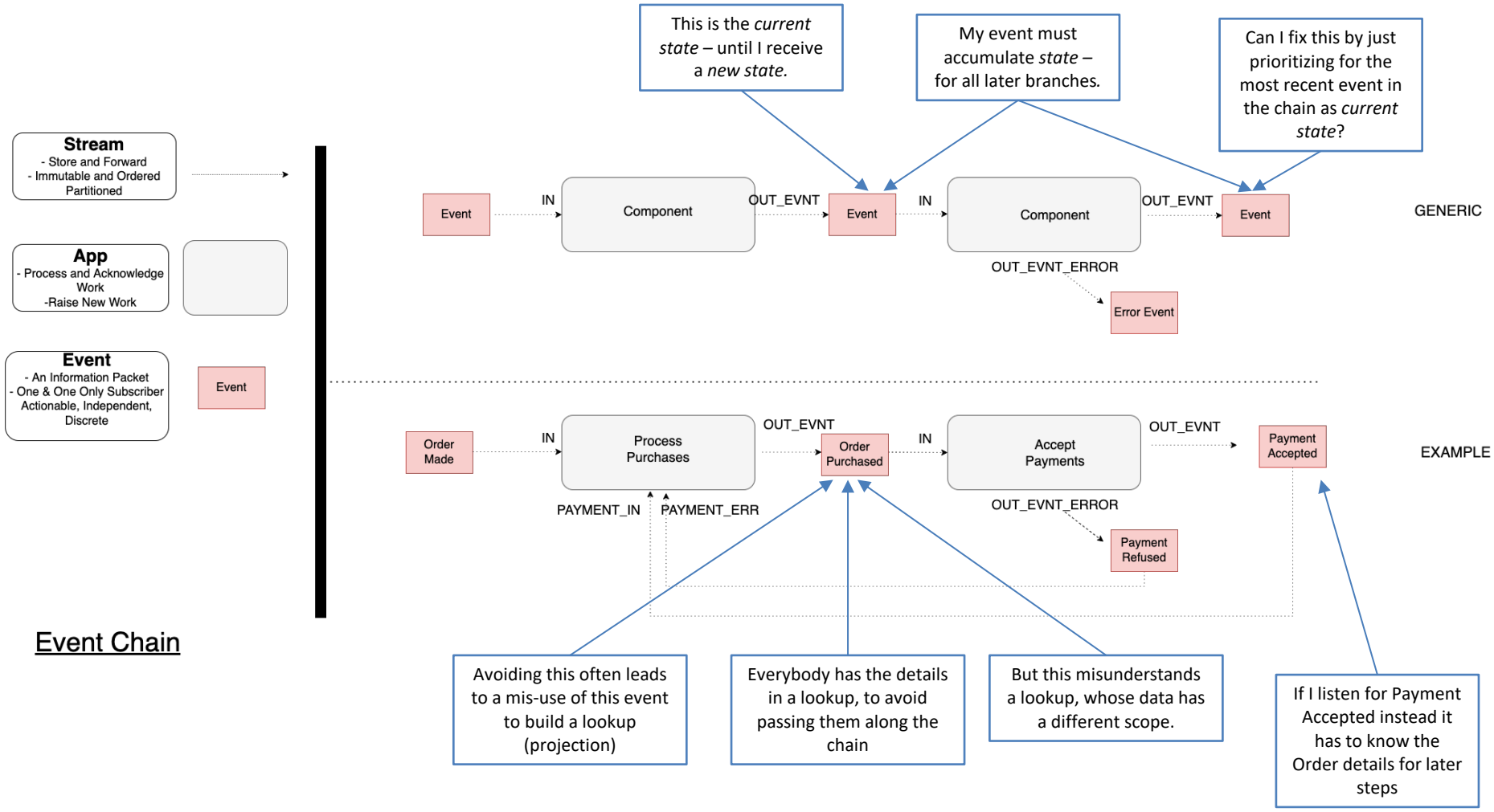
EXAMPLE

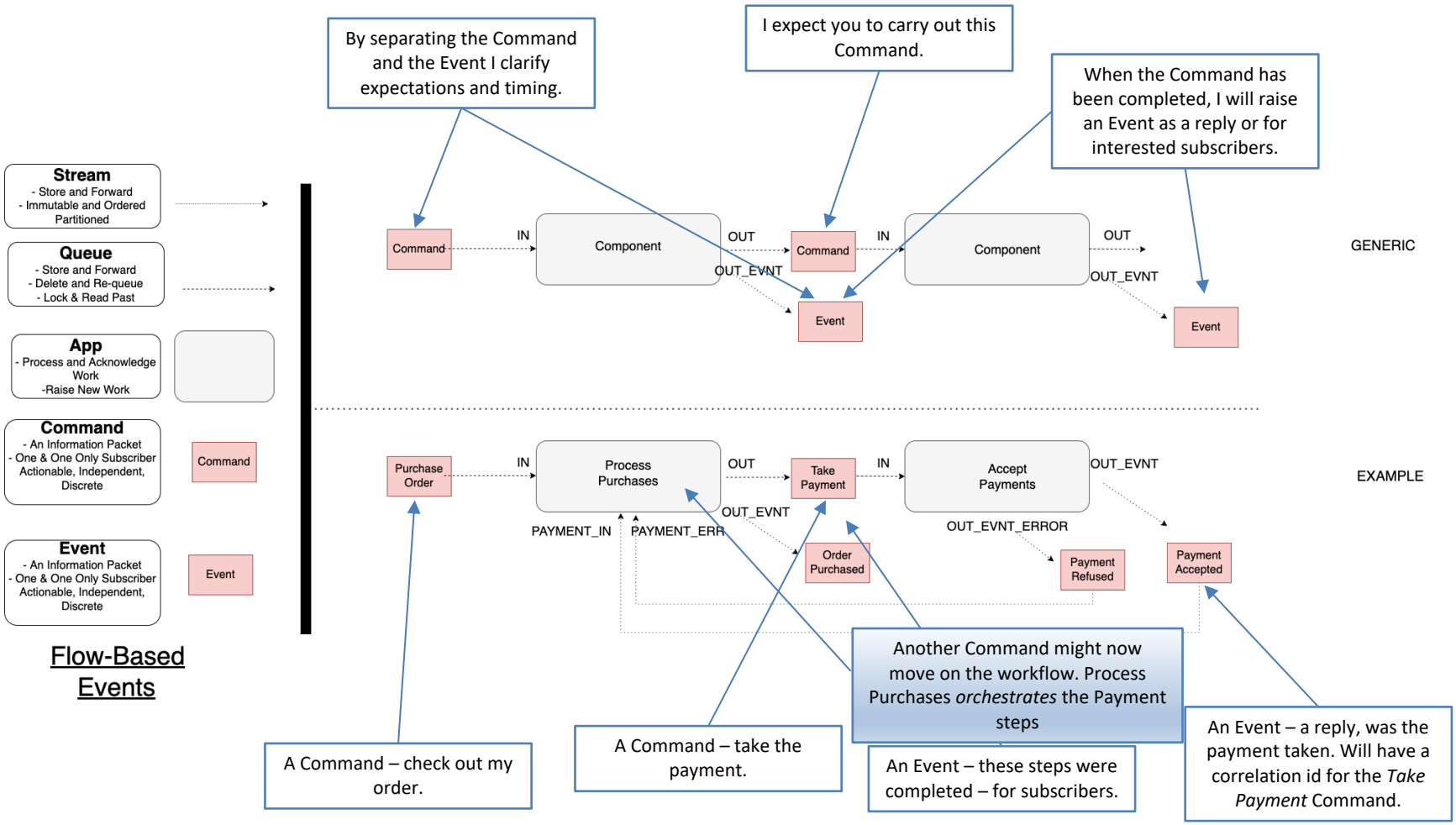
I *expect* that Accept Payments will take the customer's payment for this order.
This is a Command.

I *expect* that Accept Payments will reply to my request.
What does this?


Stream
- Store and Forward
- Immutable and Ordered
Partitioned
Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past
App
- Process and Acknowledge Work
-Raise New Work
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Command
Event
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Flow-Based Events
An Event is a Fact that tells us something happened. It is often issued because we have serviced a Command.
An Event may be Series or Discrete.
We can use a Stream or Queue for an Event
An Event may be the Reply in a Request-Reply. By using a correlation id, the sender can asynchronously receive a reply to their request.
GENERIC
Command
IN
Component
OUT
OUT_EVNT
Command
IN
Component
OUT
OUT_EVNT
Event
Event
EXAMPLE
Purchase Order
IN
Process Purchases
OUT
OUT_EVNT
Take Payment
IN
Accept Payments
OUT_EVNT
PAYMENT_IN
PAYMENT_ERR
Order Purchased
OUT_EVNT_ERROR
Payment Refused
Payment Accepted
A Status Update is often Series.
If it has zero or more subscribers we may prefer series – over a Stream
A Reply could be Series or Discrete.
If it is a reply to just one sender we may prefer Discrete – over a Queue
An error is just an event as a reply that indicates failure.
Accept Payments is a branch in the flow, so it does not raise a command, just an event as a reply.

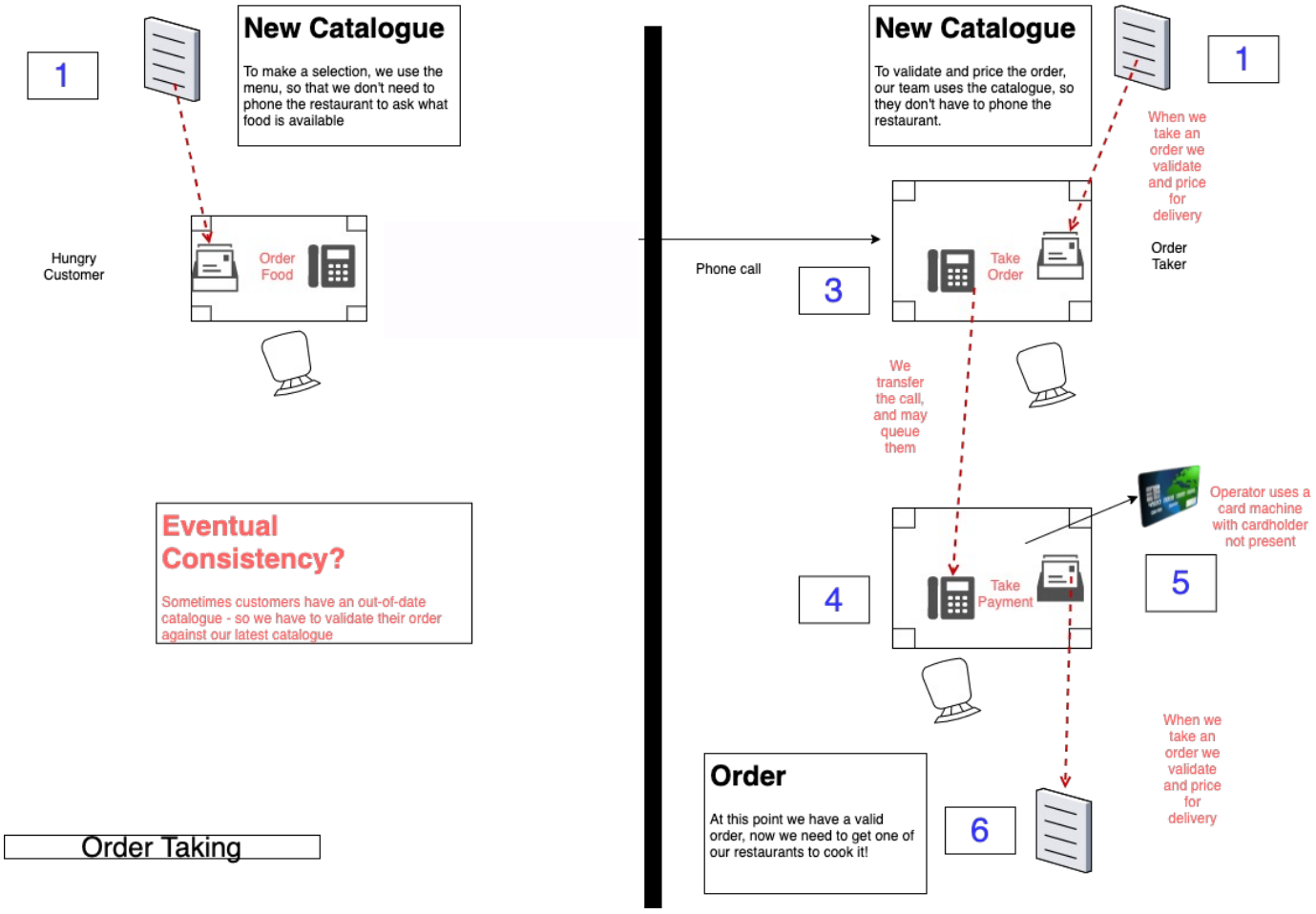

Some naïve EDAs only use Events, not Commands
Events seem preferable because they are more loosely coupled because they have zero or more consumers
But this causes us problems:
- Timing: it's often not clear when a workflow step is done.
- Hidden Coupling: We are coupled, we just hid it.
Stream
- Store and Forward
- Immutable and Ordered Partitioned
App
- Process and Acknowledge Work
-Raise New Work
Event
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Event
IN
Component
OUT_EVNT
Event
IN
Component
OUT_EVNT
Event
OUT_EVNT_ERROR
Error Event
GENERIC
Order Made
IN
Process Purchases
OUT_EVNT
Order Purchased
IN
Accept Payments
OUT_EVNT
Payment Accepted
PAYMENT_IN
PAYMENT_ERR
OUT_EVNT_ERROR
Payment Refused
EXAMPLE
Event Chain
By only using events I create a cause and effect problem
But what happens if I now get a Payment Refused?
I listen for Order Purchased to know an Order has been successfully purchased
Without a Command that waits for a Reply I lose causality here
Everyone has to listen to Payment Refused? Timing creates a Heisenbug.

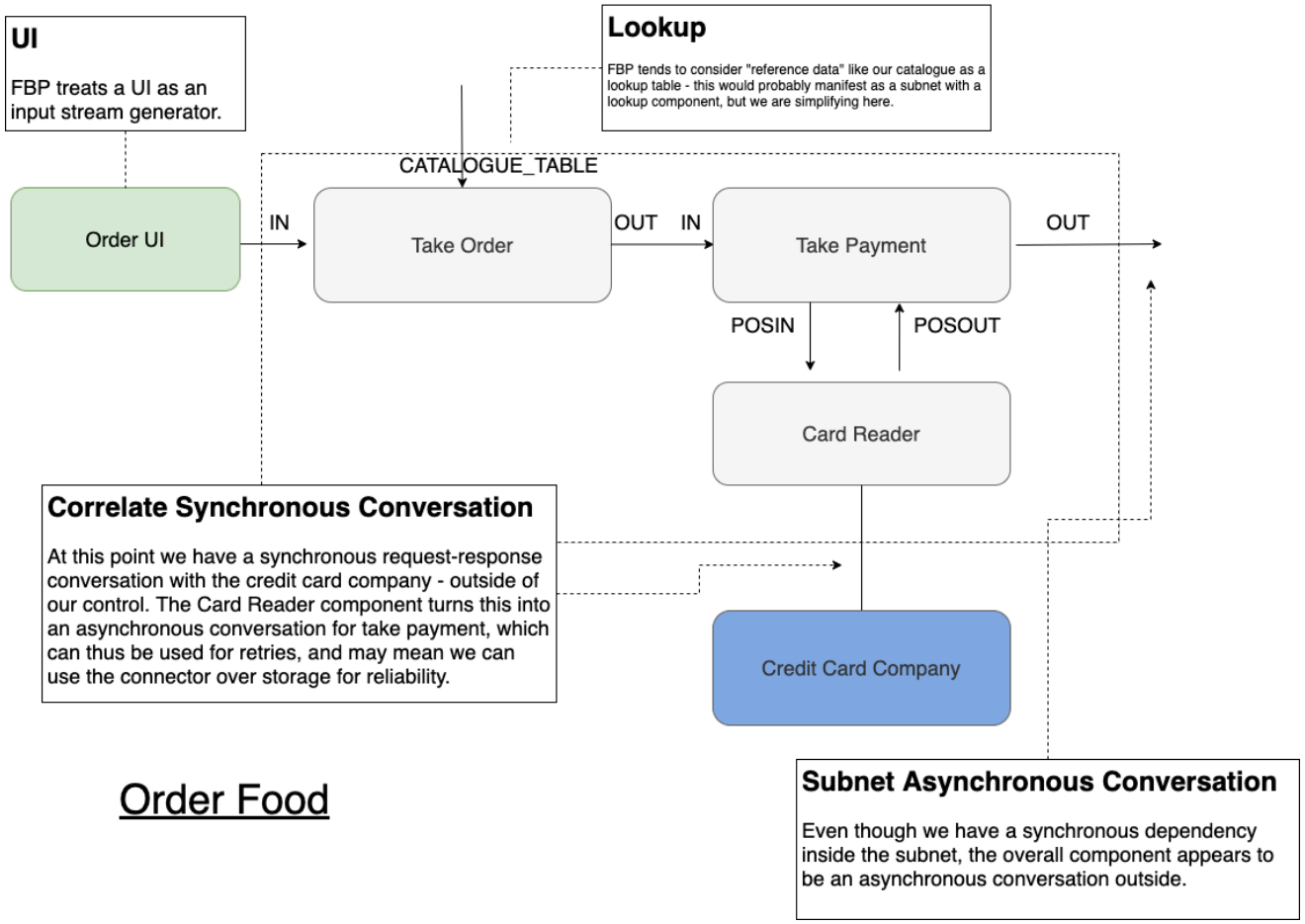

Stream
- Store and Forward
- Immutable and Ordered Partitioned
App
- Process and Acknowledge Work
-Raise New Work
Event
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Event Chain
This is the *current state* – until I receive a *new state.*
My event must accumulate *state* – for all later branches.
Can I fix this by just prioritizing for the most recent event in the chain as *current state*?
Event
IN
Component
OUT_EVNT
Event
IN
Component
OUT_EVNT
Event
GENERIC
OUT_EVNT_ERROR
Error Event
Order Made
IN
Process Purchases
OUT_EVNT
Order Purchased
IN
Accept Payments
OUT_EVNT
Payment Accepted
EXAMPLE
PAYMENT_IN
PAYMENT_ERR
OUT_EVNT_ERROR
Payment Refused
Avoiding this often leads to a mis-use of this event to build a lookup (projection)
Everybody has the details in a lookup, to avoid passing them along the chain
But this misunderstands a lookup, whose data has a different scope.
If I listen for Payment Accepted instead it has to know the Order details for later steps

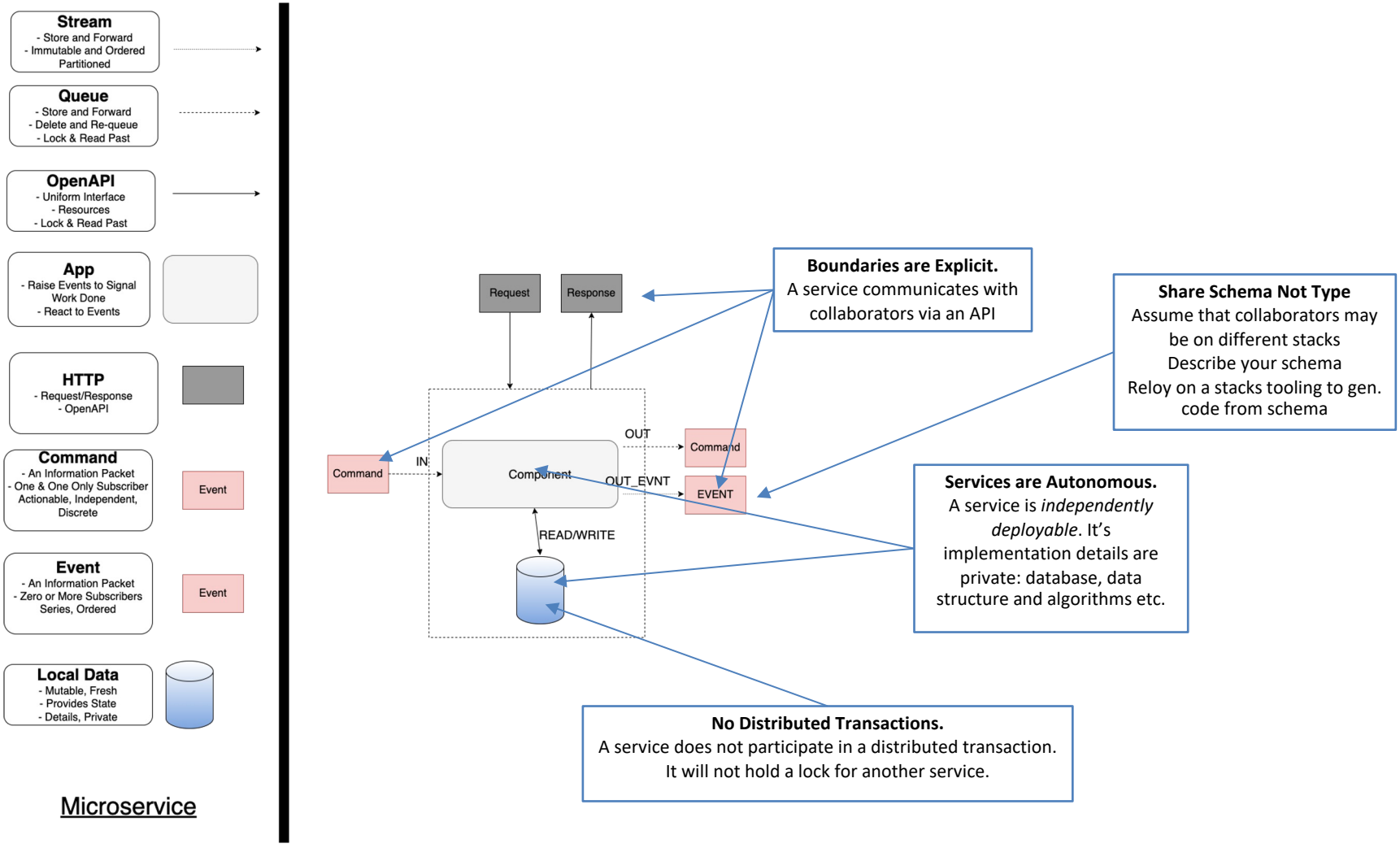

Stream
- Store and Forward
- Immutable and Ordered Partitioned
Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past
App
- Process and Acknowledge Work
-Raise New Work
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Command
Event
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Flow-Based Events
By separating the Command and the Event I clarify expectations and timing.
I expect you to carry out this Command.
When the Command has been completed, I will raise an Event as a reply or for interested subscribers.
Command
IN
Component
OUT
OUT_EVNT
Command
IN
Component
OUT
OUT_EVNT
Event
Event
GENERIC
Purchase Order
IN
Process Purchases
OUT
Take Payment
IN
Accept Payments
OUT_EVNT
OUT_EVNT
PAYMENT_IN
PAYMENT_ERR
OUT_EVNT_ERROR
Order Purchased
Payment Refused
Payment Accepted
EXAMPLE
A Command – check out my order.
A Command – take the payment.
Another Command might now move on the workflow. Process Purchases orchestrates the Payment steps
An Event – these steps were completed – for subscribers.
An Event – a reply, was the payment taken. Will have a correlation id for the Take Payment Command.

Event carried state transfer

Reference Data/Forward Cache/Data Pump/View/Projection


1
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
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
Order Taking
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
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


UI
FBP treats a UI as an input stream generator.
Lookup
FBP tends to consider "reference data" like our catalogue as a lookup table - this would probably manifest as a subnet with a lookup component, but we are simplifying here.
CATALOGUE_TABLE
Order UI
IN
Take Order
OUT
IN
Take Payment
OUT
POSIN
POSOUT
Card Reader
Correlate Synchronous Conversation
At this point we have a synchronous request-response conversation with the credit card company - outside of our control. The Card Reader component turns this into an asynchronous conversation for take payment, which can thus be used for retries, and may mean we can use the connector over storage for reliability.
Credit Card Company
Order Food
Subnet Asynchronous Conversation
Even though we have a synchronous dependency inside the subnet, the overall component appears to be an asynchronous conversation outside.


Stream
- Store and Forward
- Immutable and Ordered
Partitioned
Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past
OpenAPI
- Uniform Interface
- Resources
- Lock & Read Past
App
- Raise Events to Signal
Work Done
- React to Events
HTTP
- Request/Response
- OpenAPI
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent,
Discrete
Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered
Event
Local Data
- Mutable, Fresh
- Provides State
- Details, Private
Microservice
Request
Response
Command
IN
Component
OUT
Command
OUT_EVNT
EVENT
READ/WRITE
Boundaries are Explicit.
A service communicates with
collaborators via an API
Share Schema Not Type
Assume that collaborators may
be on different stacks
Describe your schema
Reloy on a stacks tooling to gen.
code from schema
Services are Autonomous.
A service is independently
deployable. It's
implementation details are
private: database, data
structure and algorithms etc.
No Distributed Transactions.
A service does not participate in a distributed transaction.
It will not hold a lock for another service.

Stream
- Store and Forward
- Immutable and Ordered
Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

OpenAPI
- Uniform Interface
- Resources
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

HTTP
- Request/Response
- OpenAPI

Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete

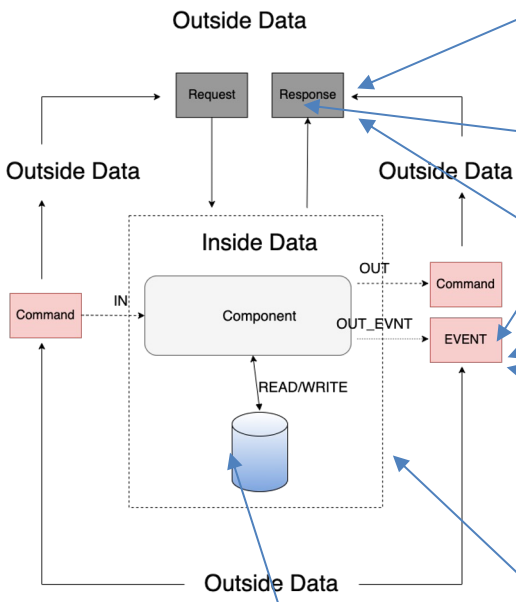
Event

Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Inside and Outside Data



Outside Data is what leaves the border of our service – it is the data our API produces.

Now only exists Inside
As transactions cannot leave a service boundary the state of entities within the service is only accurately known within the service.
Everyone else leans only the past

Outside Data is:
- Immutable
- Stale
- -Versioned

Outside Data is a contract where we accept *necessary coupling*.
Outsiders can *depend* upon our Outside Data.

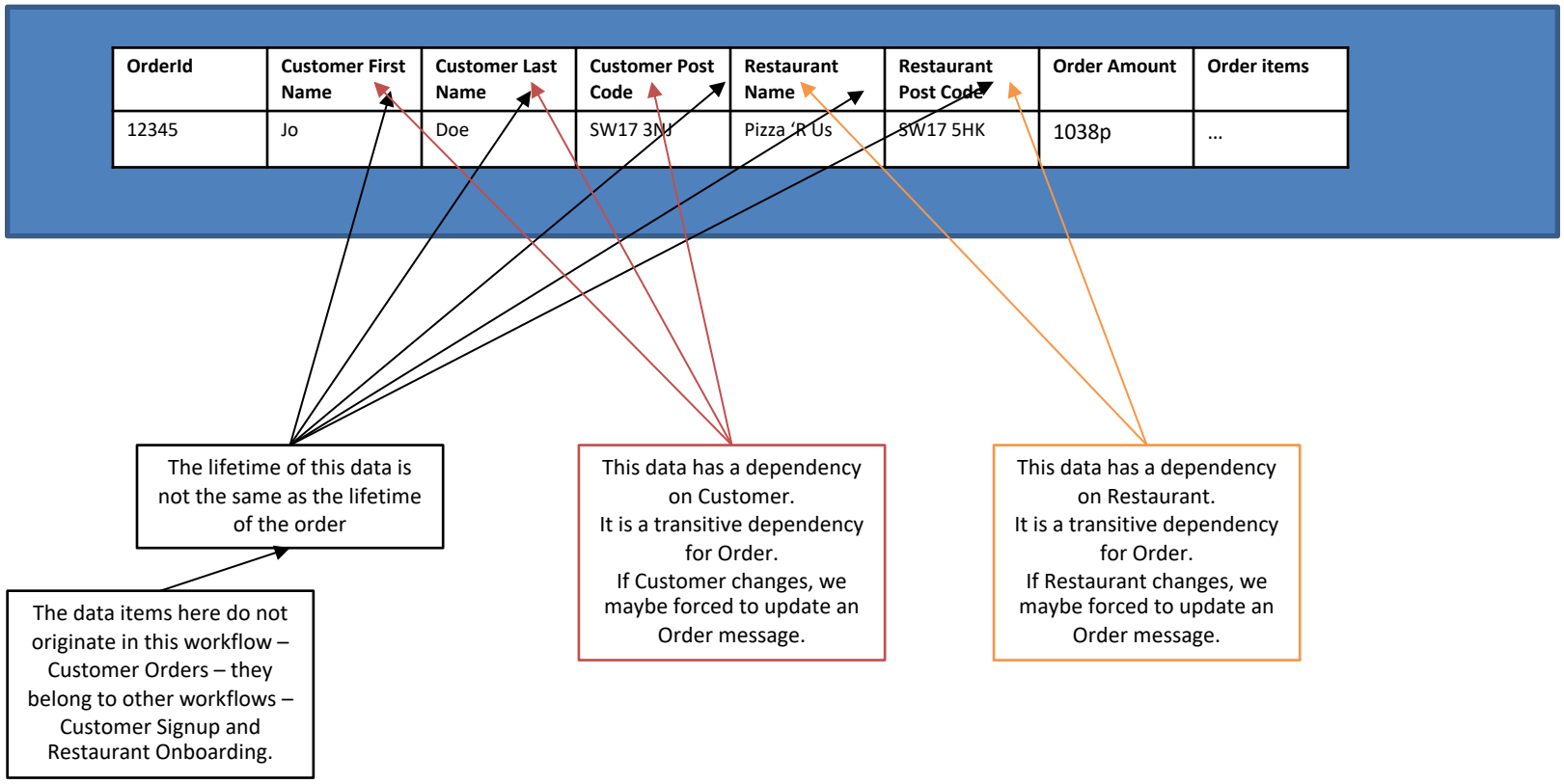
Inside Data is an implementation detail of the service.

Inside Data does not pass the border of our service – where the border is our API

Data on the Outside vs Data on the Inside - Pat Helland

Transitive Dependencies

Purchase Order Message



Normalized

Purchase Order Message

OrderId	Order Amount	Order items	CustomerId	RestaurantId
12345	1038p	…		

We normalize the message contents.
We have an Id for data from other workflows/services

Customer Upserted Message

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

What we need to do to use the Purchase Order Message is to "join" it with the Customer and Restaurant data.

The Customer workflow gives us Customer messages

Restaurant Onboarded Message

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

Just like a database we can sometimes inline – denormalize – for performance, but this should be an exception, not the rule

This Restaurant workflow gives us Restaurant messages

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

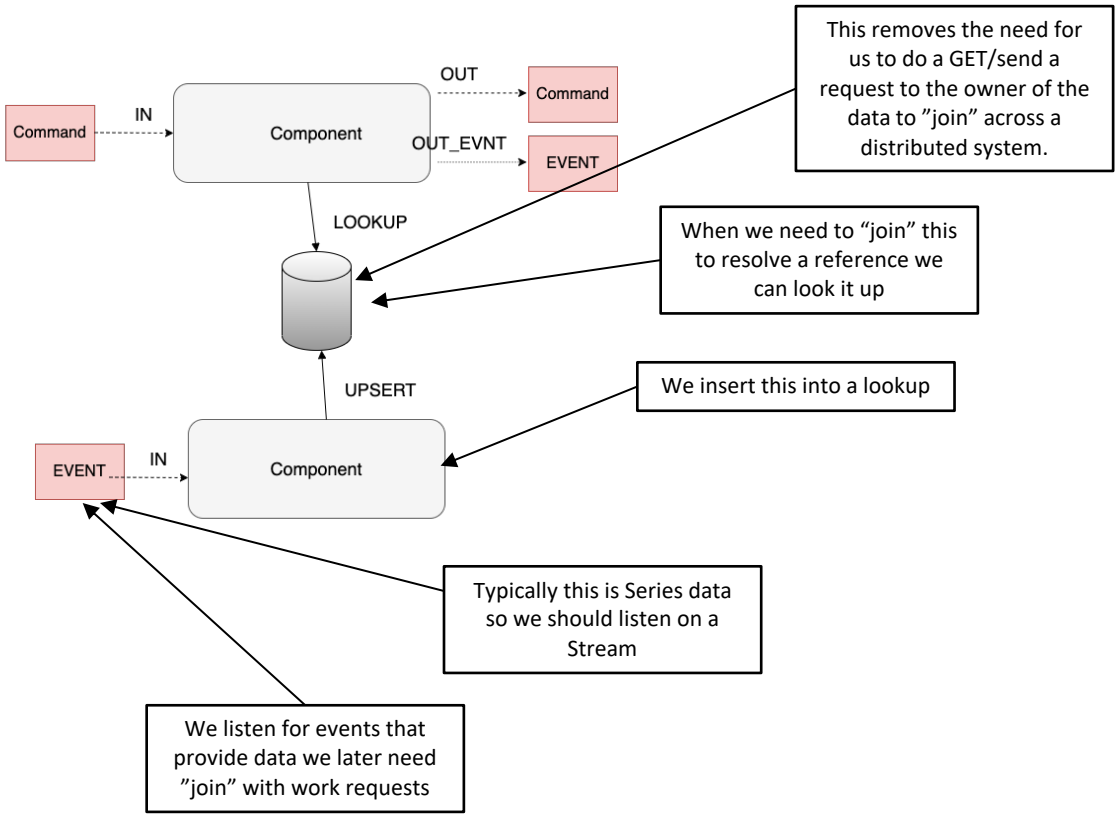
Event

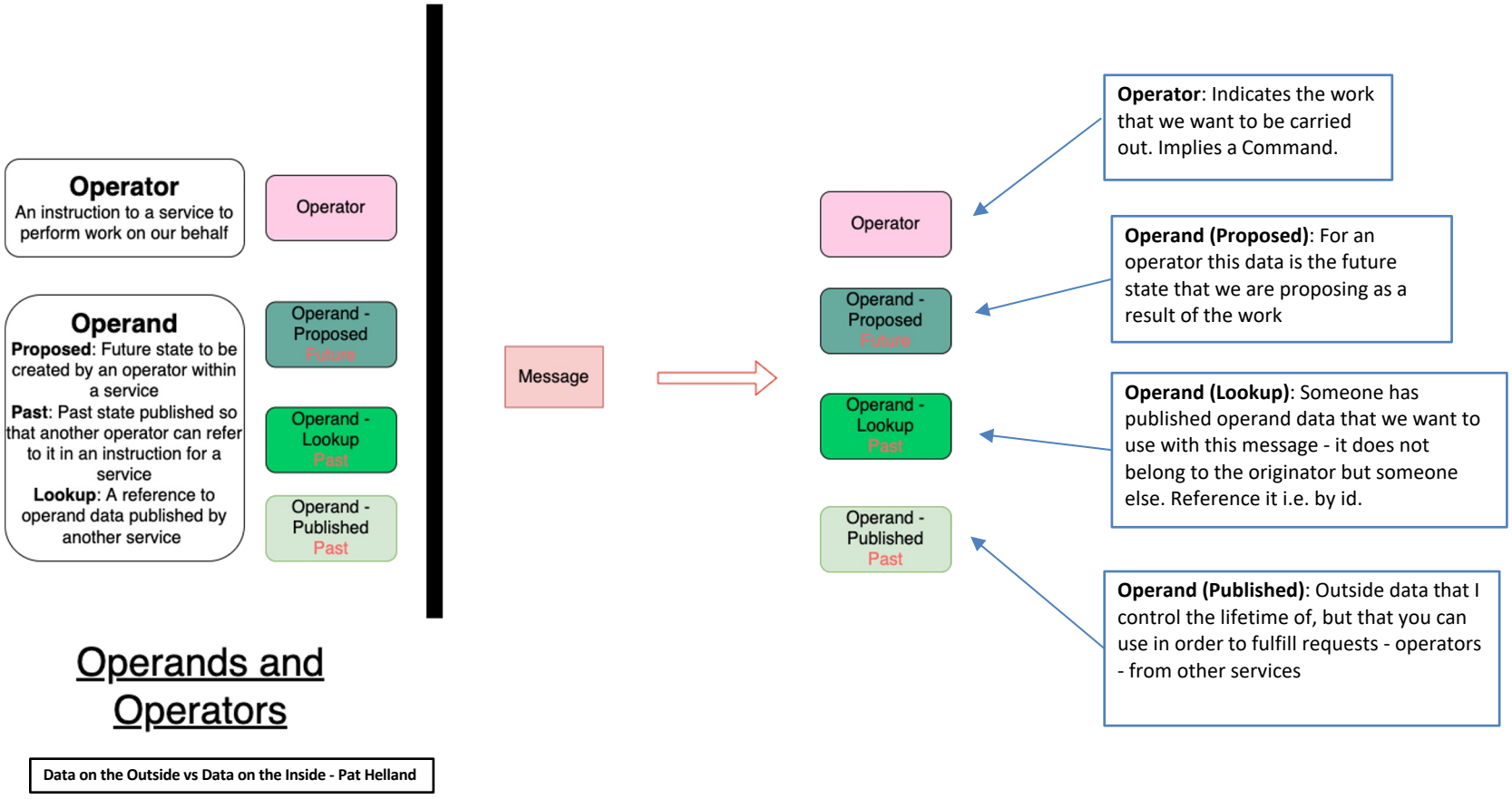
Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

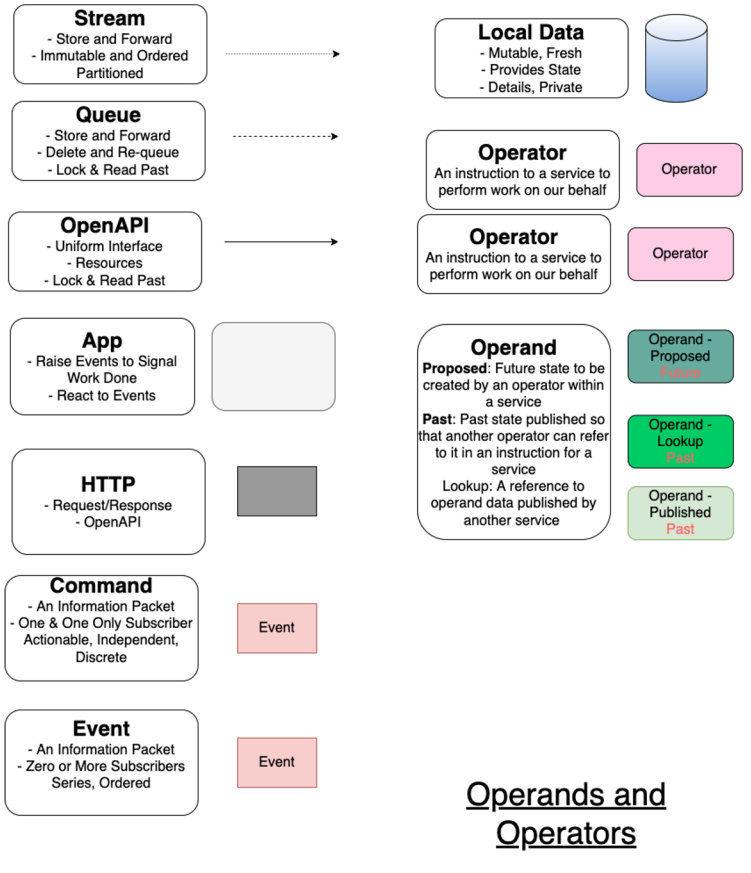
Flow-Based Lookup



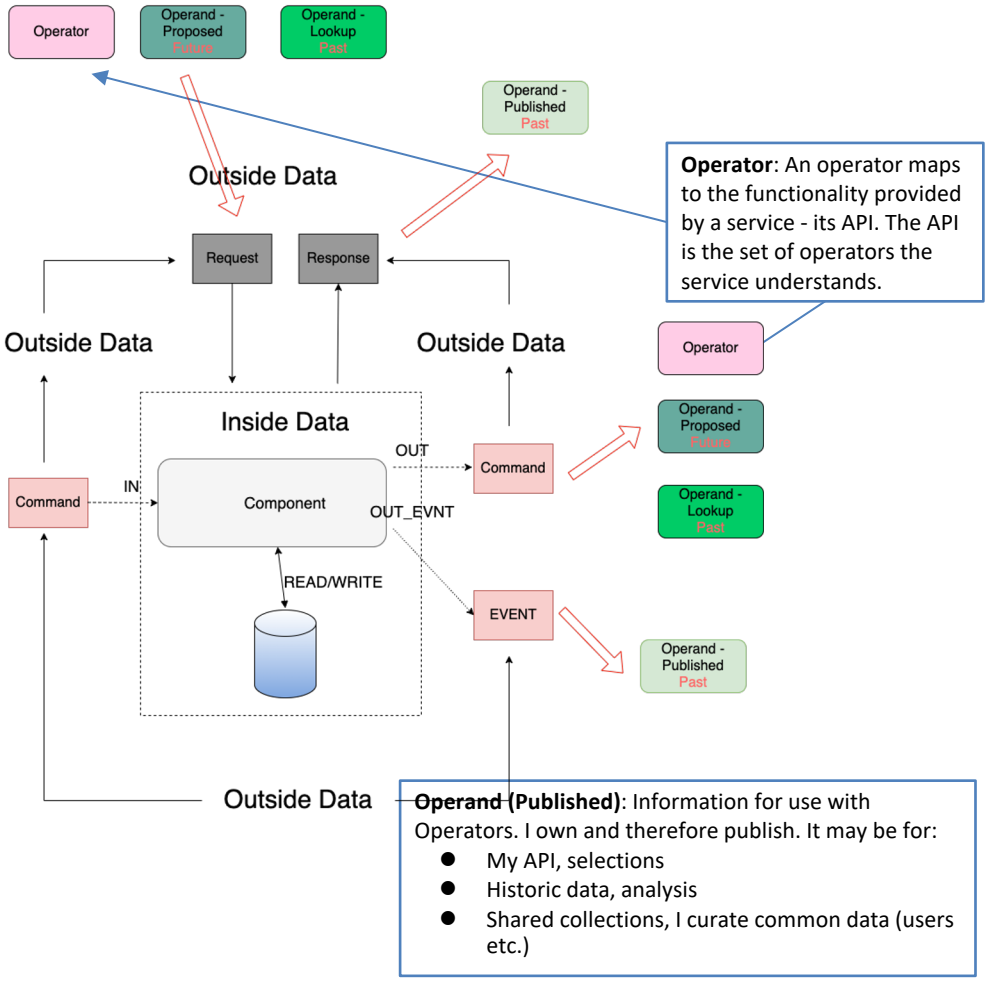


Operands and Operators

Data on the Outside vs Data on the Inside - Pat Helland



Data on the Outside vs Data on the Inside - Pat Helland



Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

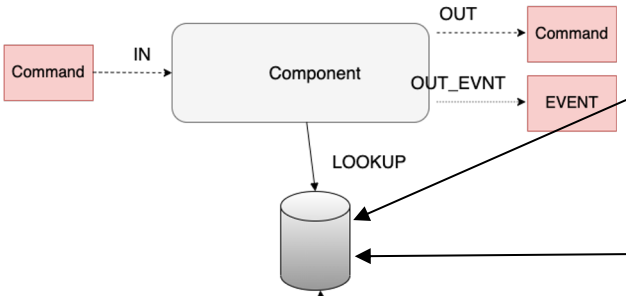
Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private



This removes the need for us to do a GET/send a request to the owner of the data and thus "*join*" across a distributed system.

When we need to to *resolve a reference* to carry out work we can now look it up

We insert this into a database so that later we can **lookup** the *operand* data required by an *operator*.

We listen for events that provide **operand data** we later need "join" with **operator** requests

Flow-Based Lookup

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

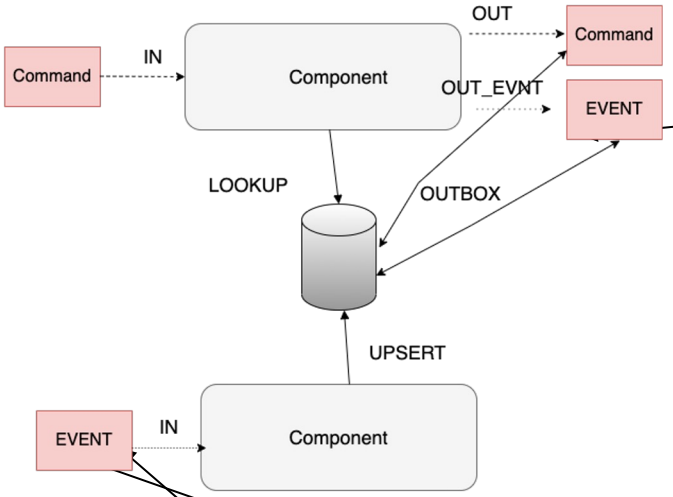
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Stream Based Reference Data



Unless it is a private **reply** we tend to default to a stream for an event because it gives us flexibility about its usage as operand data.

Typically operand data is Series data so we should use a Stream to publish it.

Archive and Replay means:
- We can rebuild a cache on demand
- We can add new consumers of existing operand data and "catch up" with the past.

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

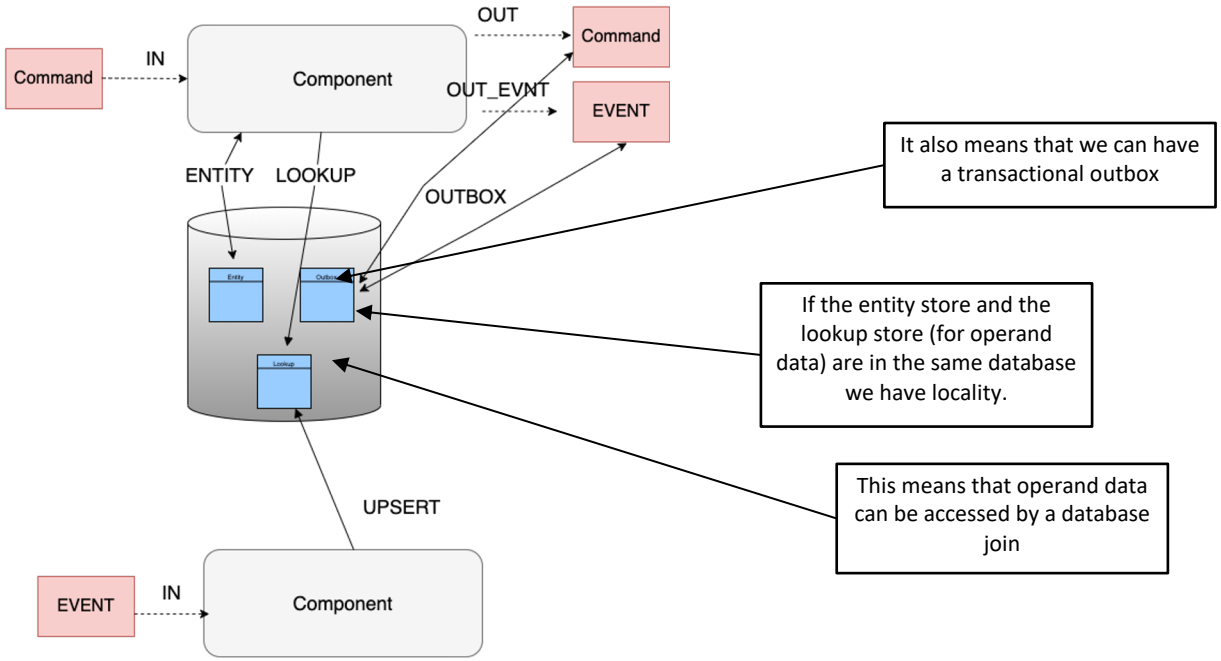
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Locality



Stream
- Store and Forward
- Immutable and Ordered
Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

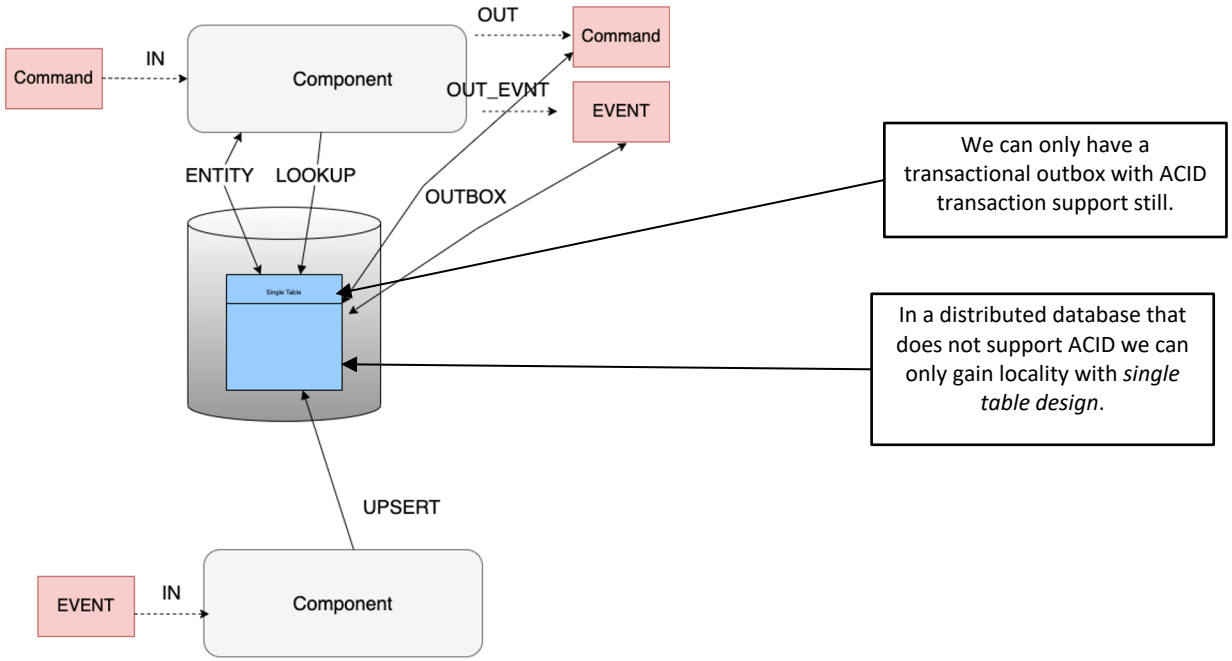
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete

Event

Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private



We can only have a transactional outbox with ACID transaction support still.

In a distributed database that does not support ACID we can only gain locality with *single table design*.

Locality No ACID

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

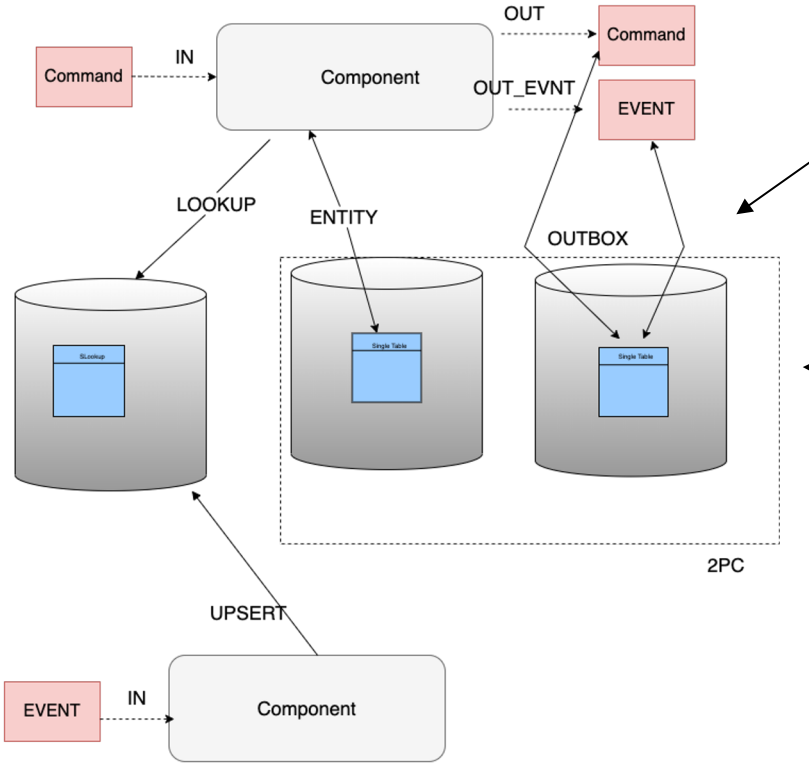
Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private



We can only get a Transactional Outbox with support for 2PC

Without *single table design* we cannot get locality, and there is no difference between a lookup to our operand data collection and another database.

<u>No Locality, No ACID</u>

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

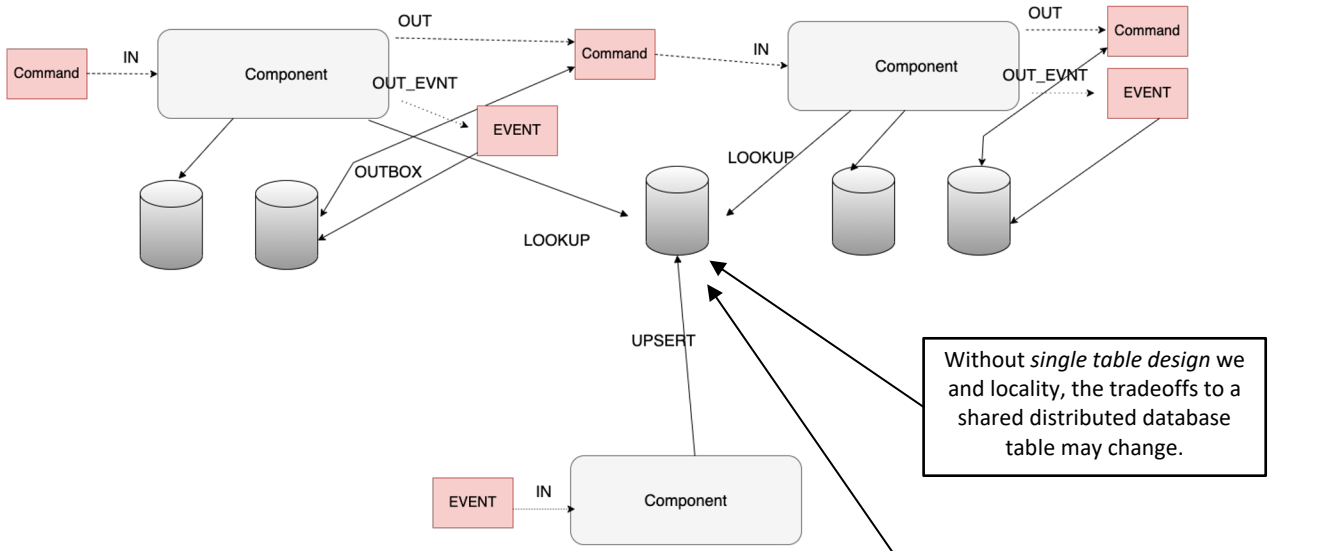
Event

Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

Shared Stream Based Reference Data



Without *single table design* we and locality, the tradeoffs to a shared distributed database table may change.

If we agree to fork for a different schema, we may be able to share with others to reduce costs of replication.

Stream
- Store and Forward
- Immutable and Ordered Partitioned

Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past

App
- Raise Events to Signal Work Done
- React to Events

Command
- An Information Packet
- One & One Only Subscriber Actionable, Independent, Discrete

Event

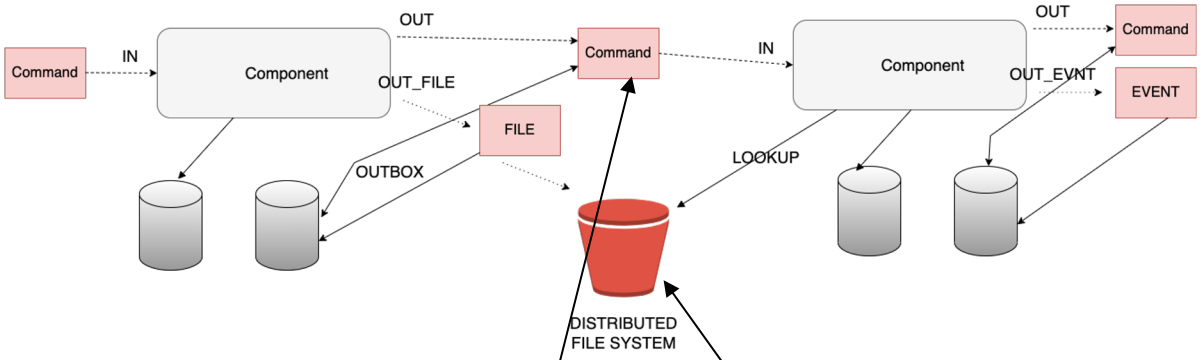
Event
- An Information Packet
- Zero or More Subscribers Series, Ordered

Event

File
- An Information Packet
- Stored on DFS
-Many Readers/One Writer

File

Local Data
- Mutable, Fresh
- Provides State
- Details, Private

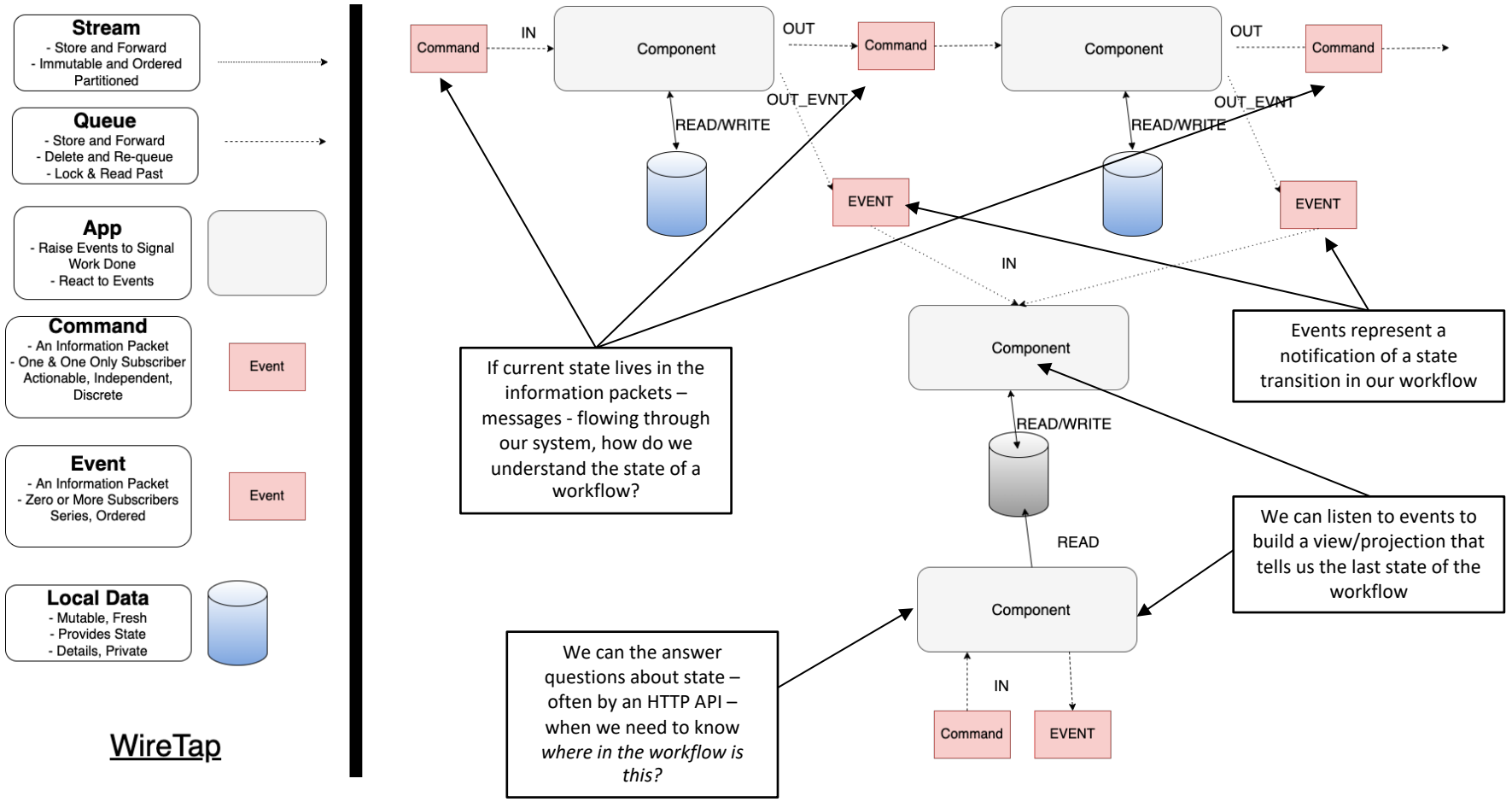


Either the command needs to provide a URI to the reference data it uses
Or an event needs to be provided to allow consumers to build a lookup table of URIs

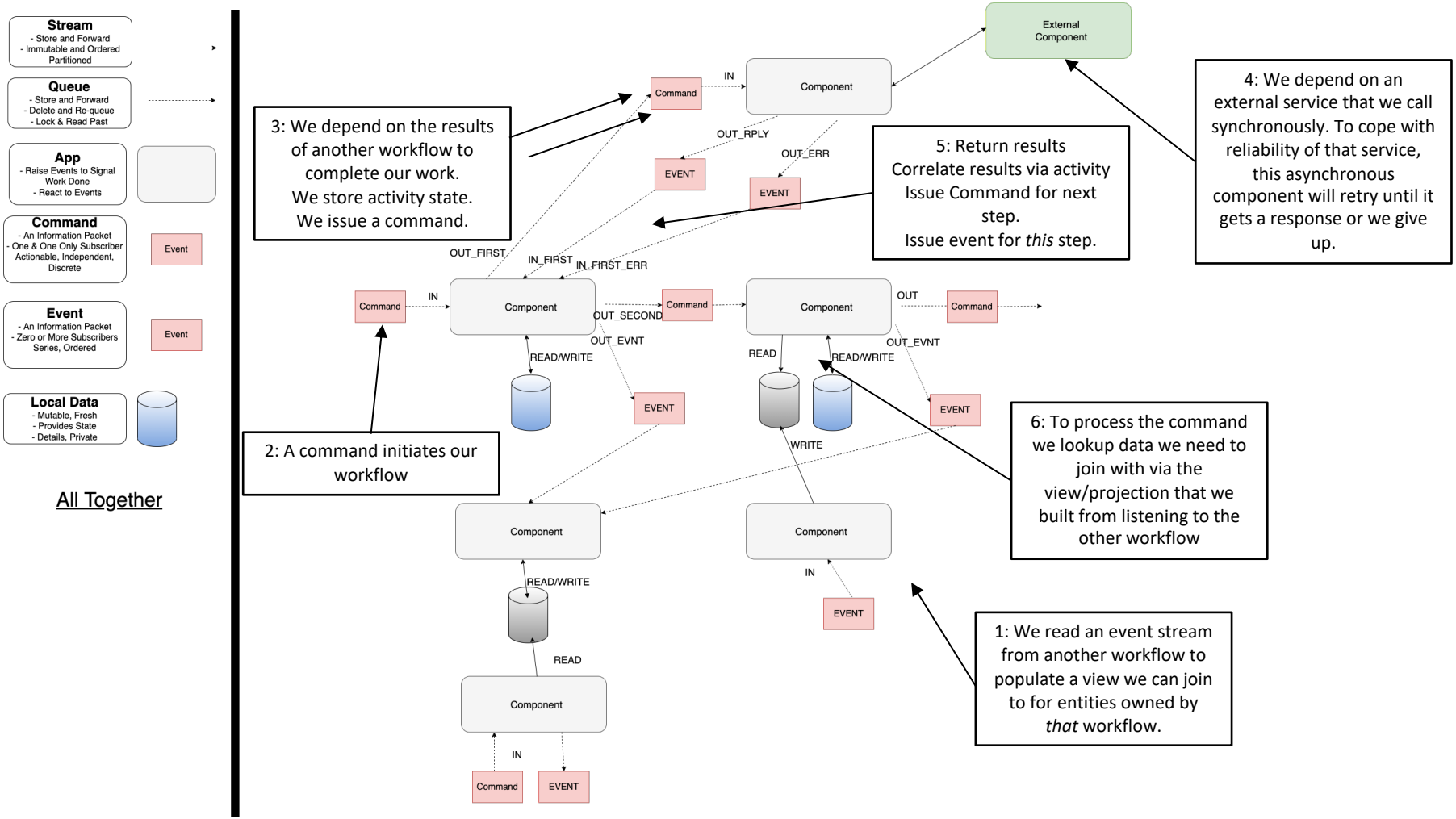
Where operand data is large in size, and/or very slow moving it may make sense to store it in file storage

Shared File Based Reference Data

Pipeline Observability


Stream
- Store and Forward
- Immutable and Ordered Partitioned
Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past
App
- Raise Events to Signal Work Done
- React to Events
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered
Event
Local Data
- Mutable, Fresh
- Provides State
- Details, Private
WireTap
Command
IN
Component
OUT
Command
Component
OUT
Command
READ/WRITE
OUT_EVNT
READ/WRITE
OUT_EVNT
EVENT
EVENT
IN
Component
READ/WRITE
READ
Component
IN
Command
EVENT
If current state lives in the information packets – messages - flowing through our system, how do we understand the state of a workflow?
Events represent a notification of a state transition in our workflow
We can listen to events to build a view/projection that tells us the last state of the workflow
We can the answer questions about state – often by an HTTP API – when we need to know *where in the workflow is this?*

Putting It Together

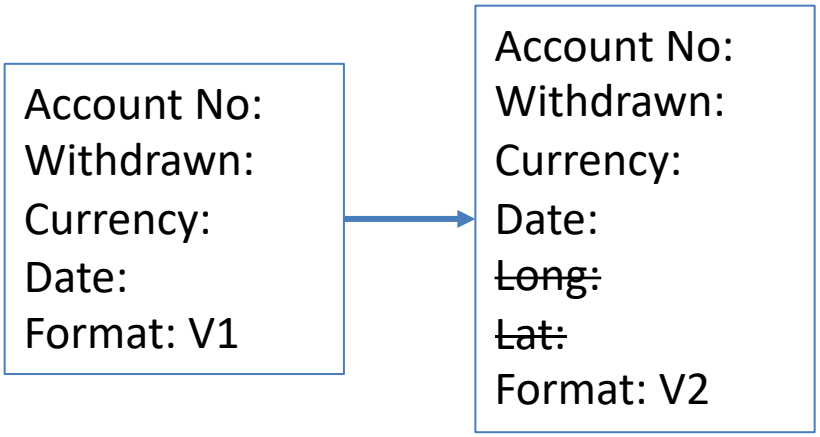

Stream
- Store and Forward
- Immutable and Ordered
Partitioned
Queue
- Store and Forward
- Delete and Re-queue
- Lock & Read Past
App
- Raise Events to Signal Work Done
- React to Events
Command
- An Information Packet
- One & One Only Subscriber
Actionable, Independent, Discrete
Event
Event
- An Information Packet
- Zero or More Subscribers
Series, Ordered
Event
Local Data
- Mutable, Fresh
- Provides State
- Details, Private
All Together
External Component
Command
IN
Component
OUT_RPLY
OUT_ERR
EVENT
EVENT
3: We depend on the results of another workflow to complete our work.
We store activity state.
We issue a command.
5: Return results
Correlate results via activity
Issue Command for next step.
Issue event for this step.
4: We depend on an external service that we call synchronously. To cope with reliability of that service, this asynchronous component will retry until it gets a response or we give up.
OUT_FIRST
IN_FIRST
IN_FIRST_ERR
Command
IN
Component
OUT_SECOND
Command
Component
OUT
Command
READ/WRITE
OUT_EVNT
READ
READ/WRITE
OUT_EVNT
EVENT
EVENT
2: A command initiates our workflow
6: To process the command we lookup data we need to join with via the view/projection that we built from listening to the other workflow
WRITE
Component
Component
READ/WRITE
IN
EVENT
READ
Component
IN
Command
EVENT
1: We read an event stream from another workflow to populate a view we can join to for entities owned by that workflow.

5 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.

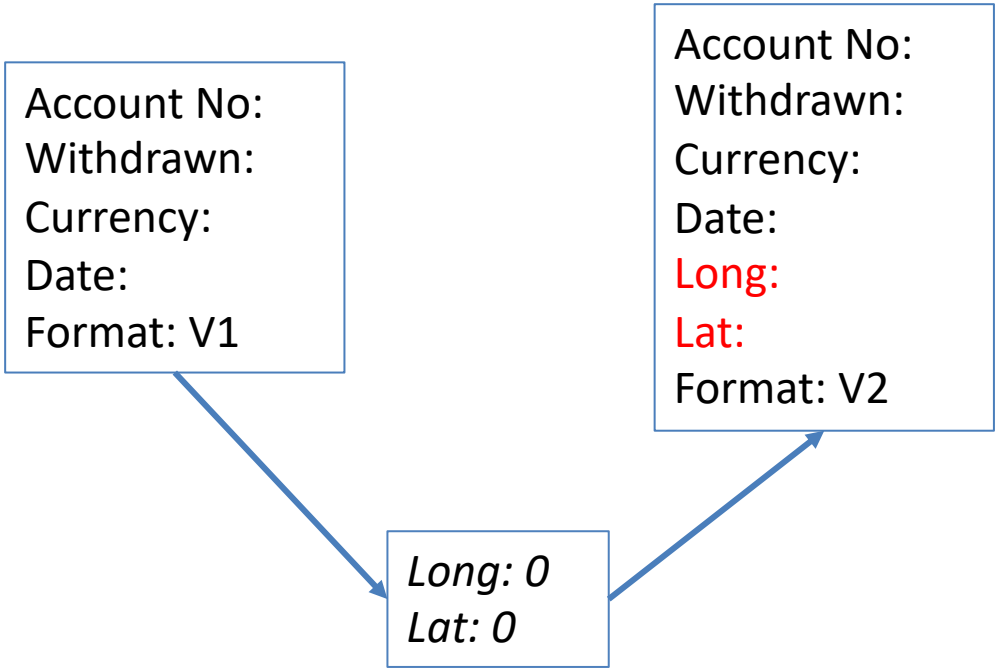
Robustness Principal or Postel's Law – Jon Postel RFC 1958

Tolerant Reader



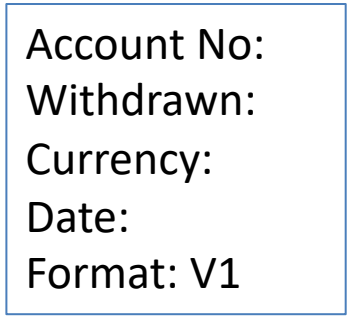
Ignore New Fields

Tolerant Reader

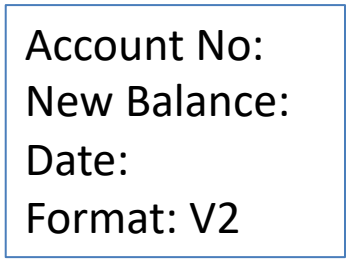


Default Missing Fields

Breaking Change



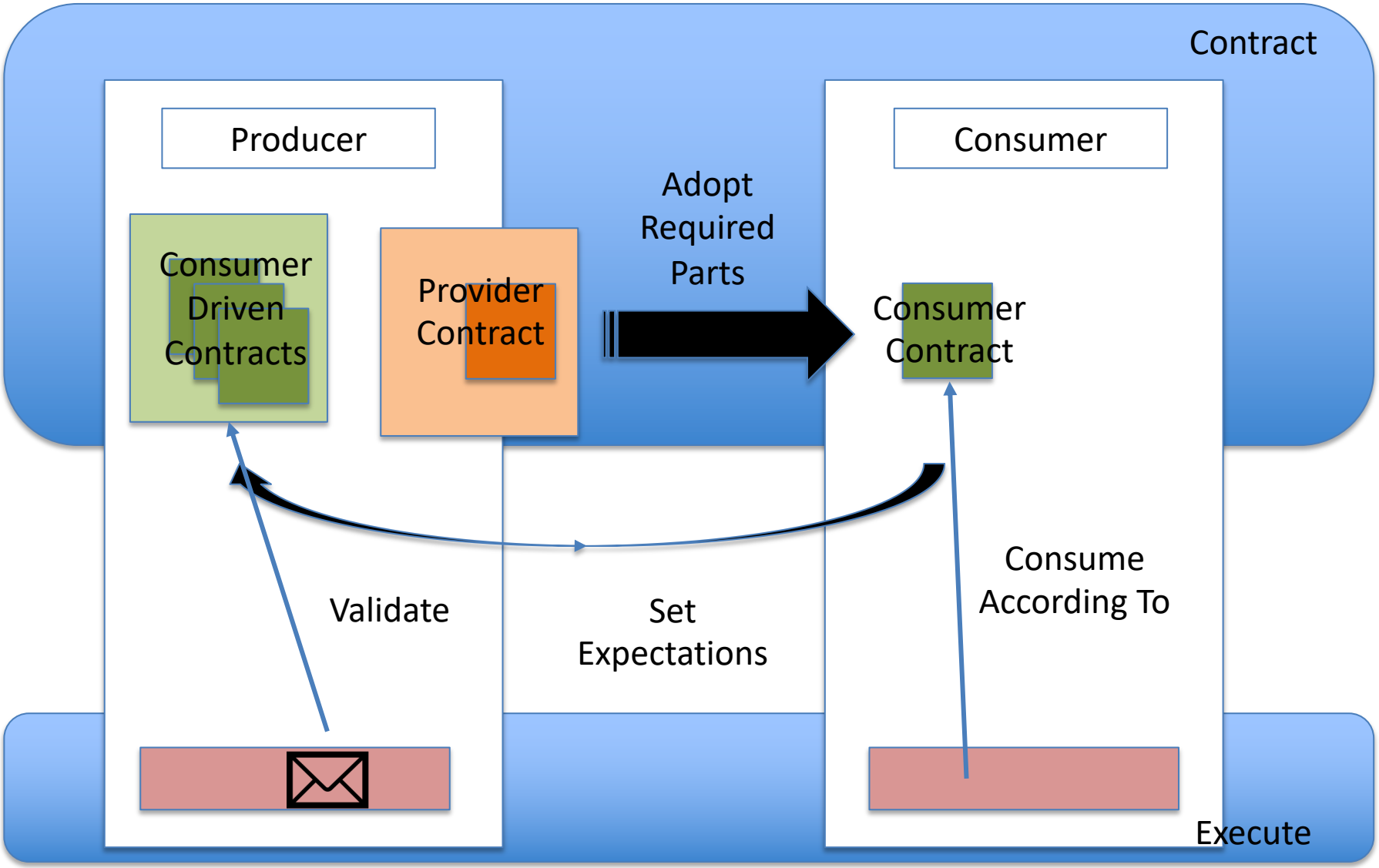
Account.Withdrawal.Event



Account.NewBalance.Event

New Message

Consumer Driven Contracts



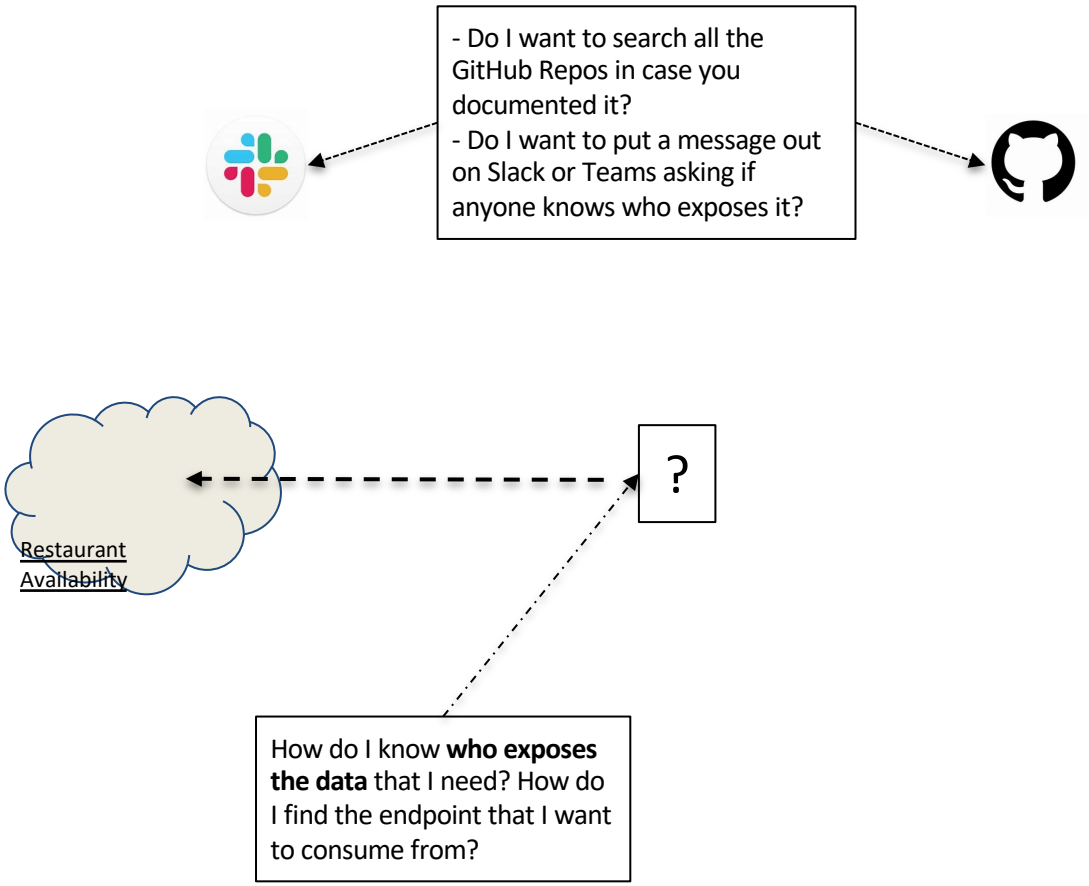
Standards & How to Describe Messaging

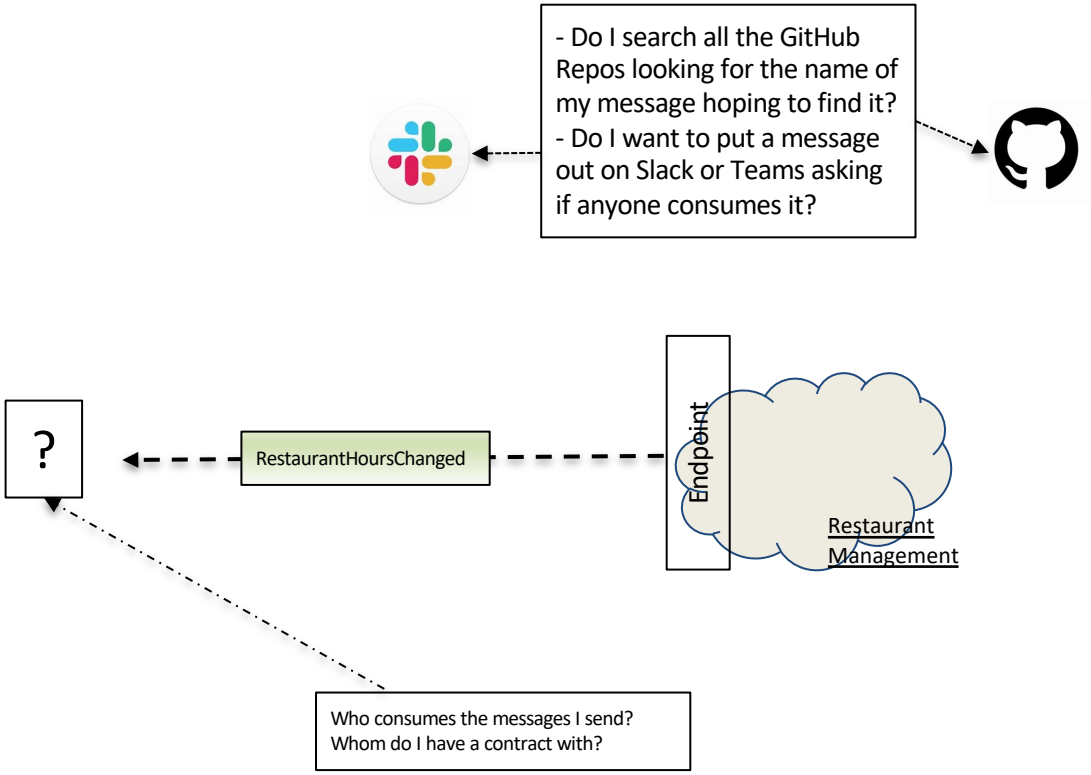
6 DOCUMENTATION

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

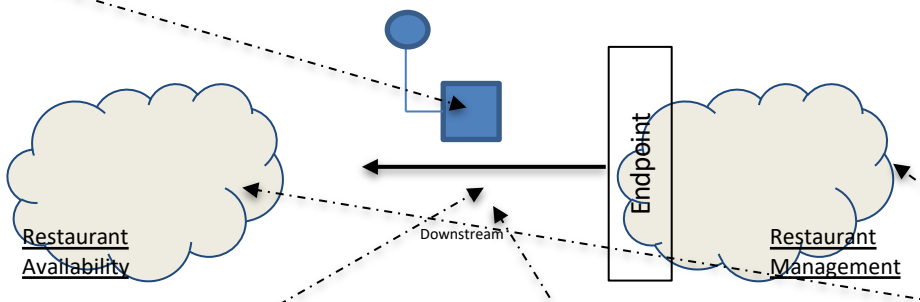

- Do I want to search all the GitHub Repos in case you documented it?
- Do I want to put a message out on Slack or Teams asking if anyone knows who exposes it?
Restaurant Availability
?
How do I know **who exposes the data** that I need? How do I find the endpoint that I want to consume from?


- 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

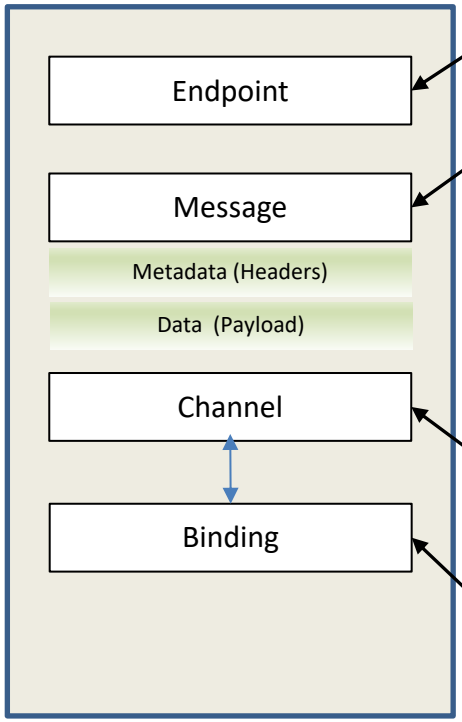


We need to document who sends and receives to understand flow

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

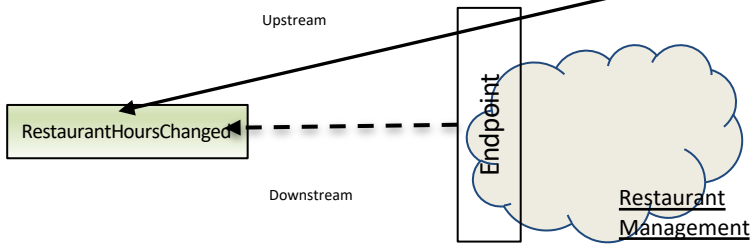
Endpoint – Documenting the Contract



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.

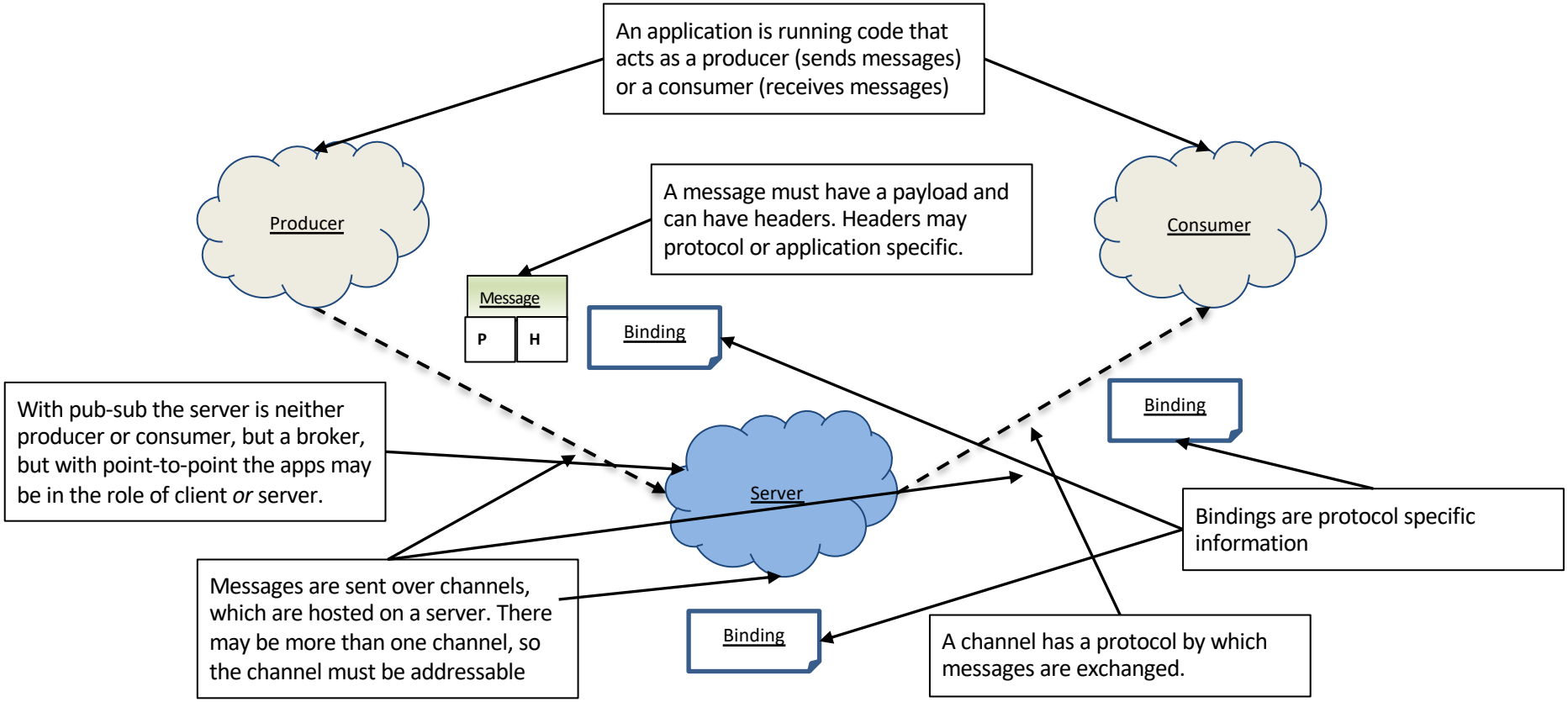
Read more about Open Governance TSC Members

And much more...

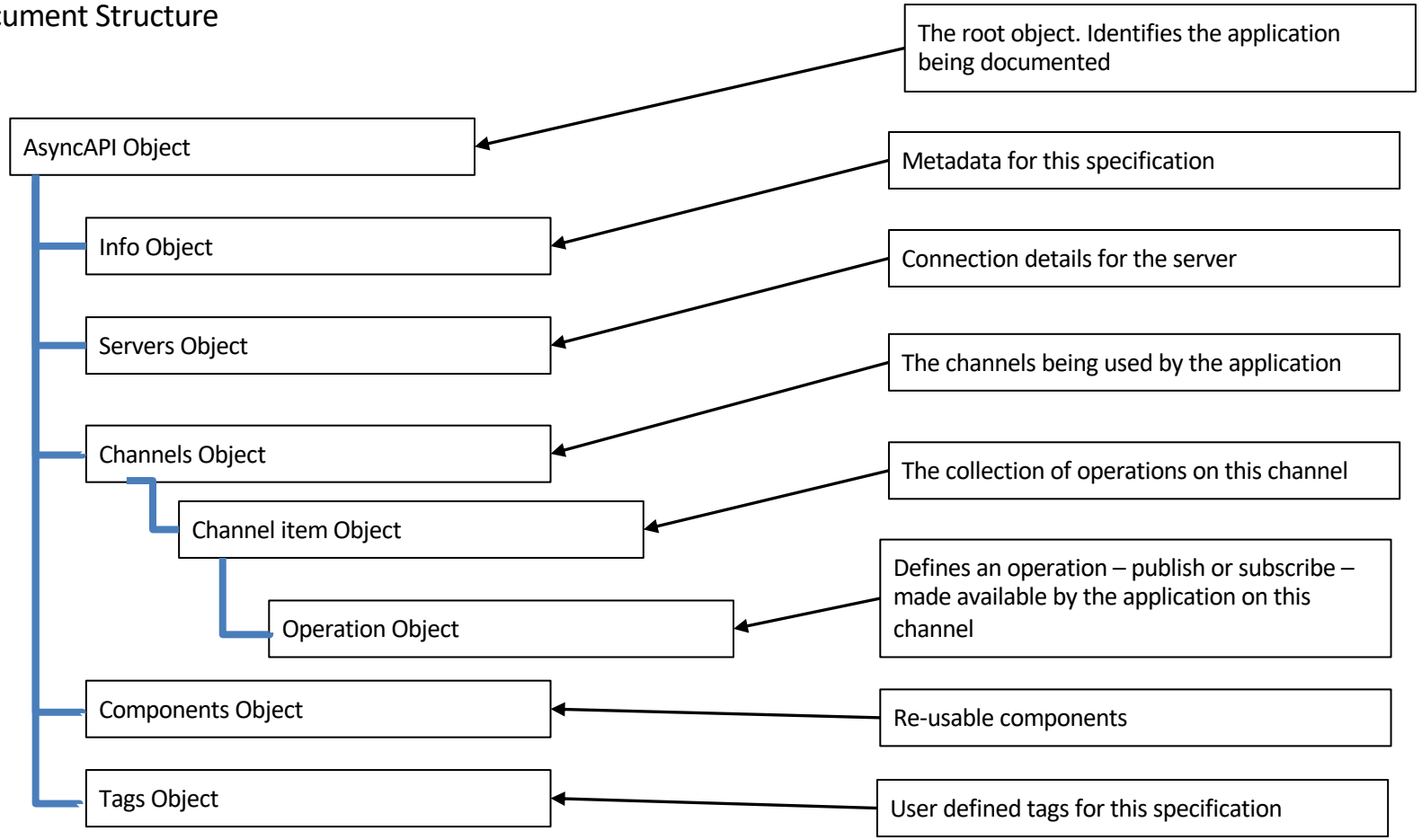
We have many different tools and welcome you to explore our ideas and propose new ideas to AsyncAPI.

View GitHub Discussions

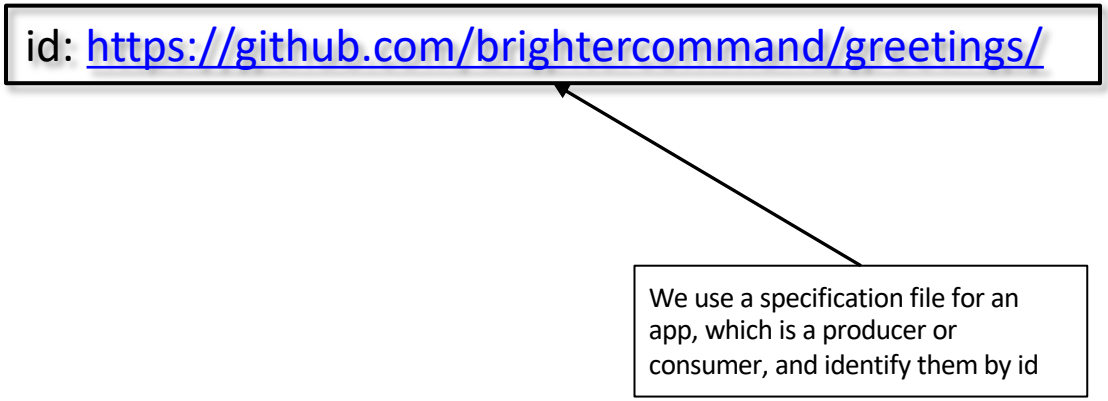
AsyncAPI Elements



Document Structure



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
```

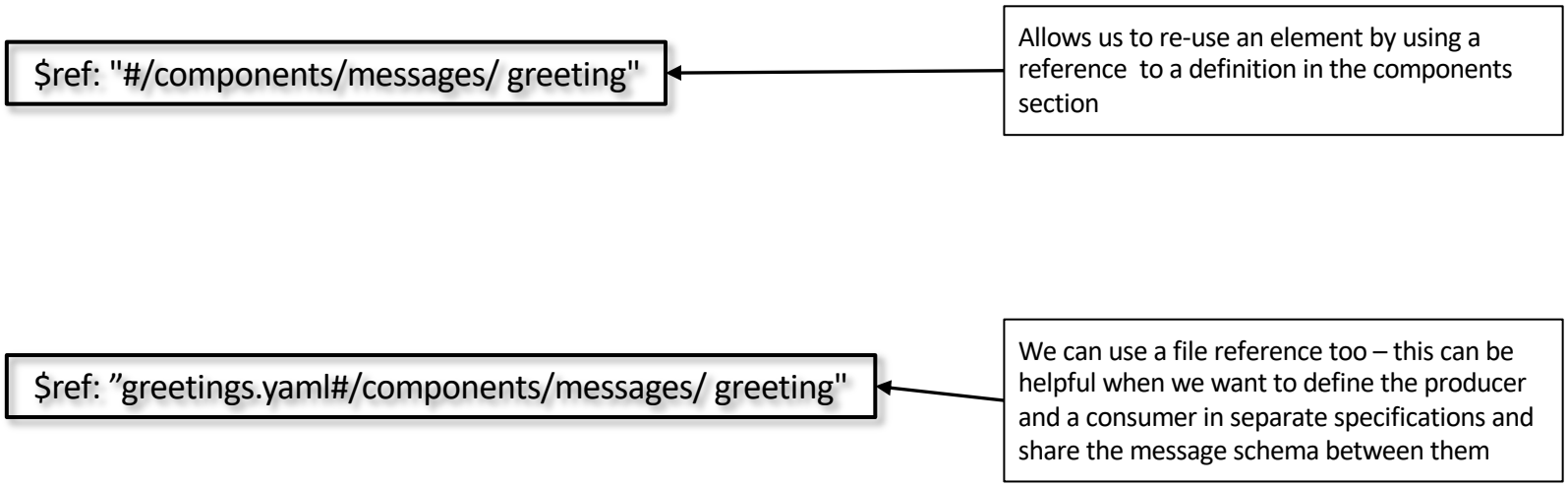
Servers Object

```
local:
  url: localhost:9092
  protocol: kafka
```

Channels

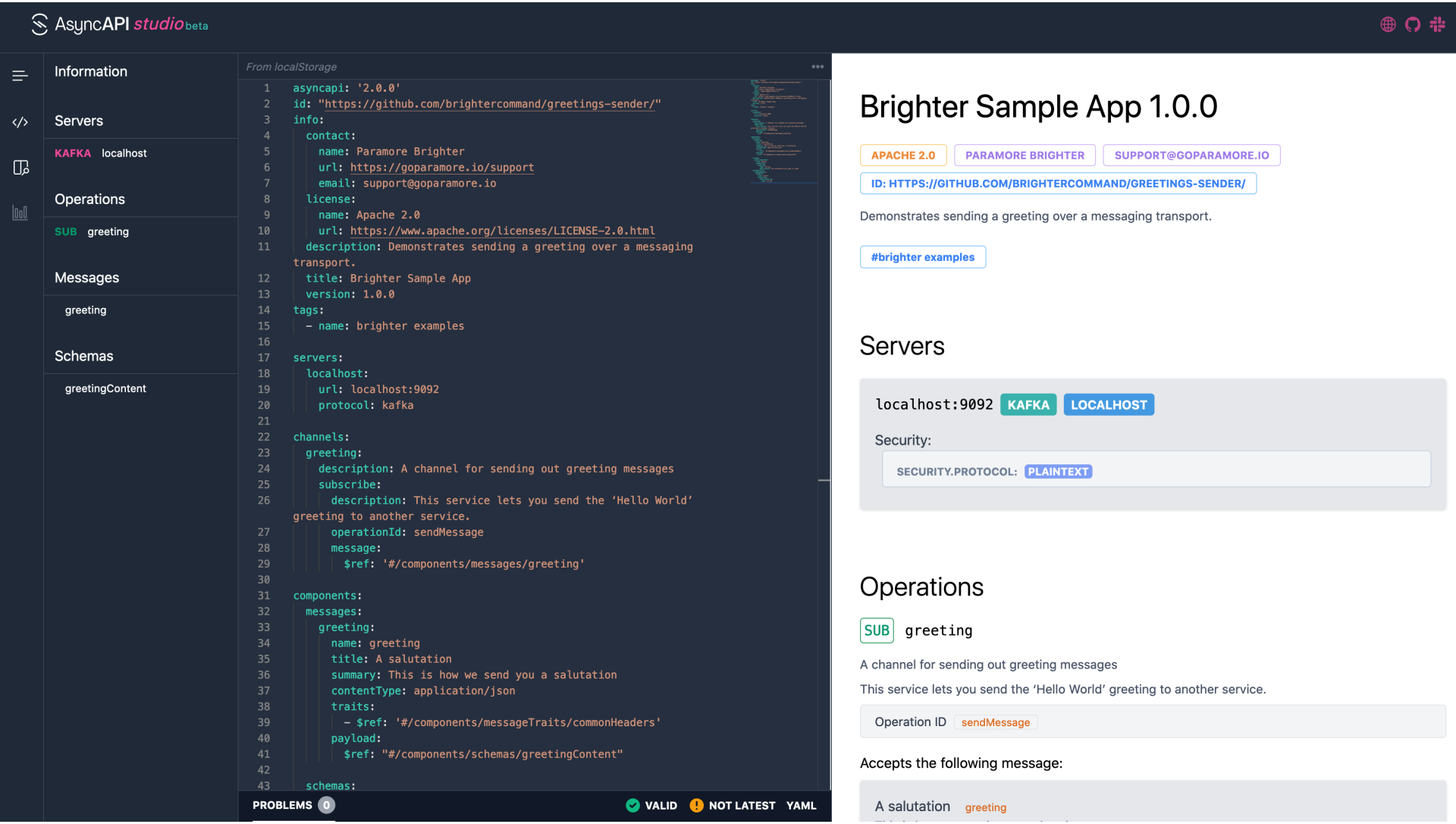
```
greeting:
  subscribe:
    operationid: sendGreeting
    summary: sends a greeting
    description: This service lets you send the ‘Hello World’ greeting to another service.
    message:
      $ref: "#/components/messages/greeting”
```

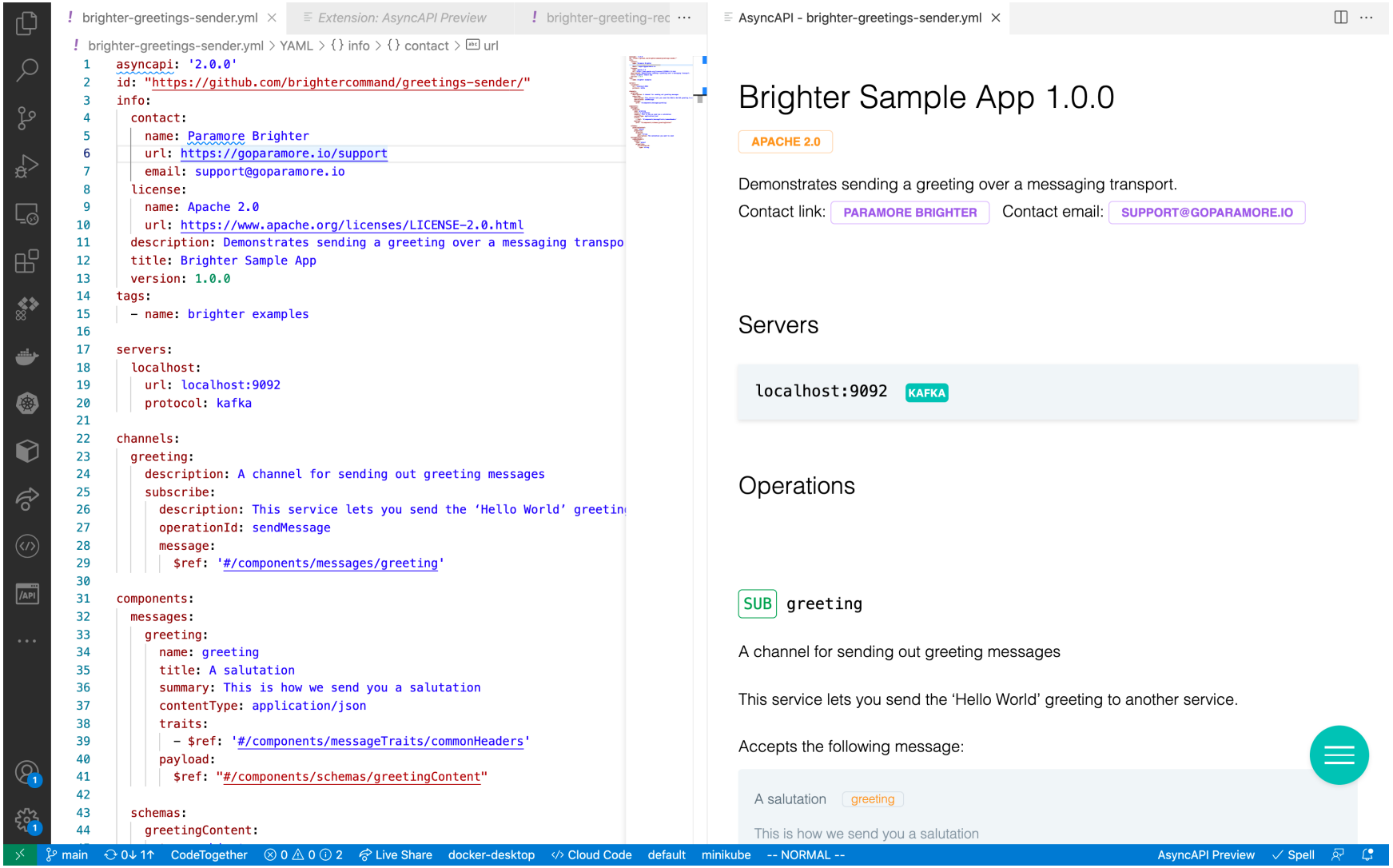
Ref

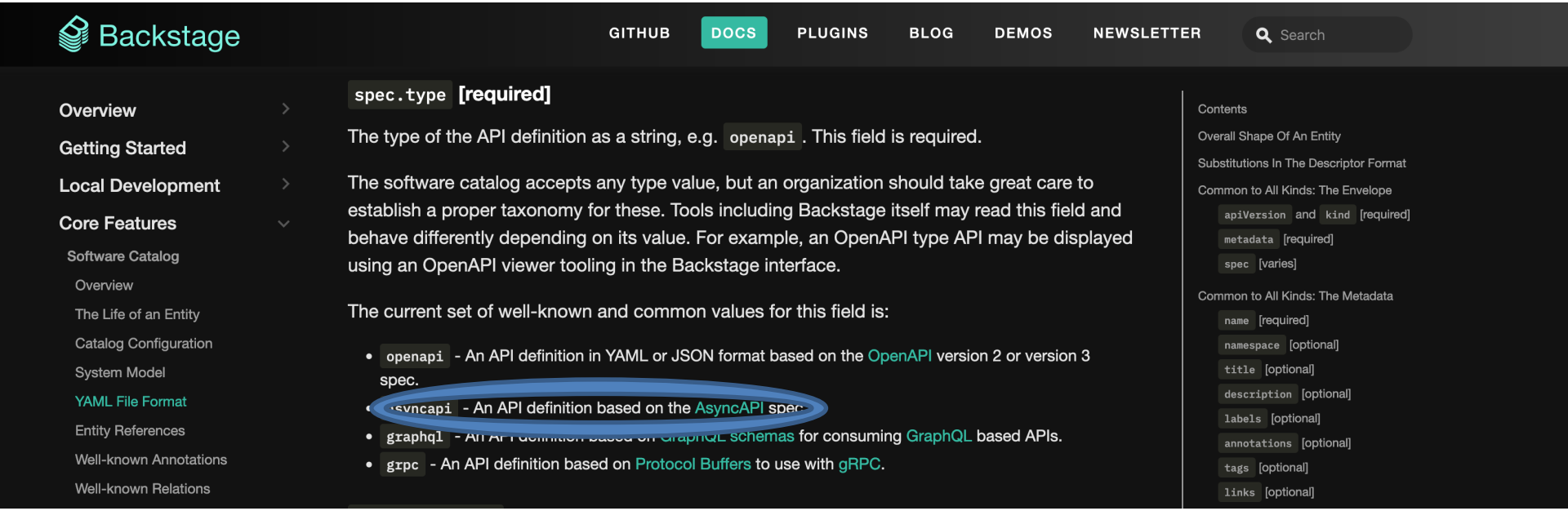

$ref: "#/components/messages/ greeting"
Allows us to re-use an element by using a reference to a definition in the components section
$ref: ”greetings.yaml#/components/messages/ greeting"
We can use a file reference too – this can be helpful when we want to define the producer and a consumer in separate specifications and share the message schema between them

Components

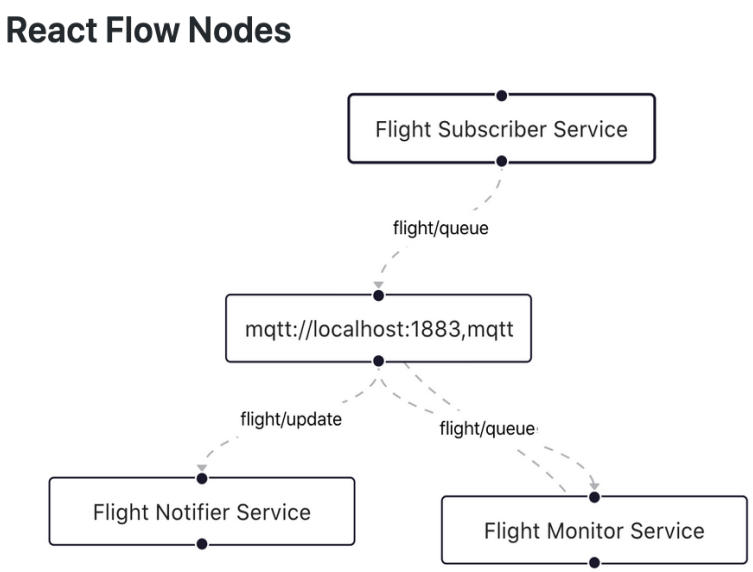
```
components:
  messages:
    greeting:
      name: greeting
      title: A salutation
      summary: This is how we send you a salutation
      contentType: application/json
      traits:
        - $ref: '#/components/messageTraits/commonHeaders'
      payload:
        $ref: "#/components/schemas/greetingContent"

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


AsyncAPI studio beta
Information
Servers
KAFKA localhost
Operations
SUB greeting
Messages
greeting
Schemas
greetingContent
From localStorage
asyncapi: '2.0.0'
id: "https://github.com/brightercommand/greetings-sender/"
info:
contact:
name: Paramore Brighter
url: https://goparamore.io/support
email: support@goparamore.io
license:
name: Apache 2.0
url: https://www.apache.org/licenses/LICENSE-2.0.html
description: Demonstrates sending a greeting over a messaging transport.
title: Brighter Sample App
version: 1.0.0
tags:
- name: brighter examples
servers:
localhost:
url: localhost:9092
protocol: kafka
channels:
greeting:
description: A channel for sending out greeting messages
subscribe:
description: This service lets you send the 'Hello World' greeting to another service.
operationId: sendMessage
message:
$ref: '#/components/messages/greeting'
components:
messages:
greeting:
name: greeting
title: A salutation
summary: This is how we send you a salutation
contentType: application/json
traits:
- $ref: '#/components/messageTraits/commonHeaders'
payload:
$ref: "#/components/schemas/greetingContent"
schemas:
PROBLEMS 0
VALID
NOT LATEST
YAML
Brighter Sample App 1.0.0
APACHE 2.0
PARAMORE BRIGHTER
SUPPORT@GOPARAMORE.IO
ID: HTTPS://GITHUB.COM/BRIGHTERCOMMAND/GREETINGS-SENDER/
Demonstrates sending a greeting over a messaging transport.
#brighter examples
Servers
localhost:9092 KAFKA LOCALHOST
Security:
SECURITY.PROTOCOL: PLAINTEXT
Operations
SUB greeting
A channel for sending out greeting messages
This service lets you send the 'Hello World' greeting to another service.
Operation ID sendMessage
Accepts the following message:
A salutation greeting


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
0↓ 1↑
CodeTogether
0 0 2
Live Share
docker-desktop
Cloud Code
default
minikube
-- NORMAL --
AsyncAPI Preview
Spell

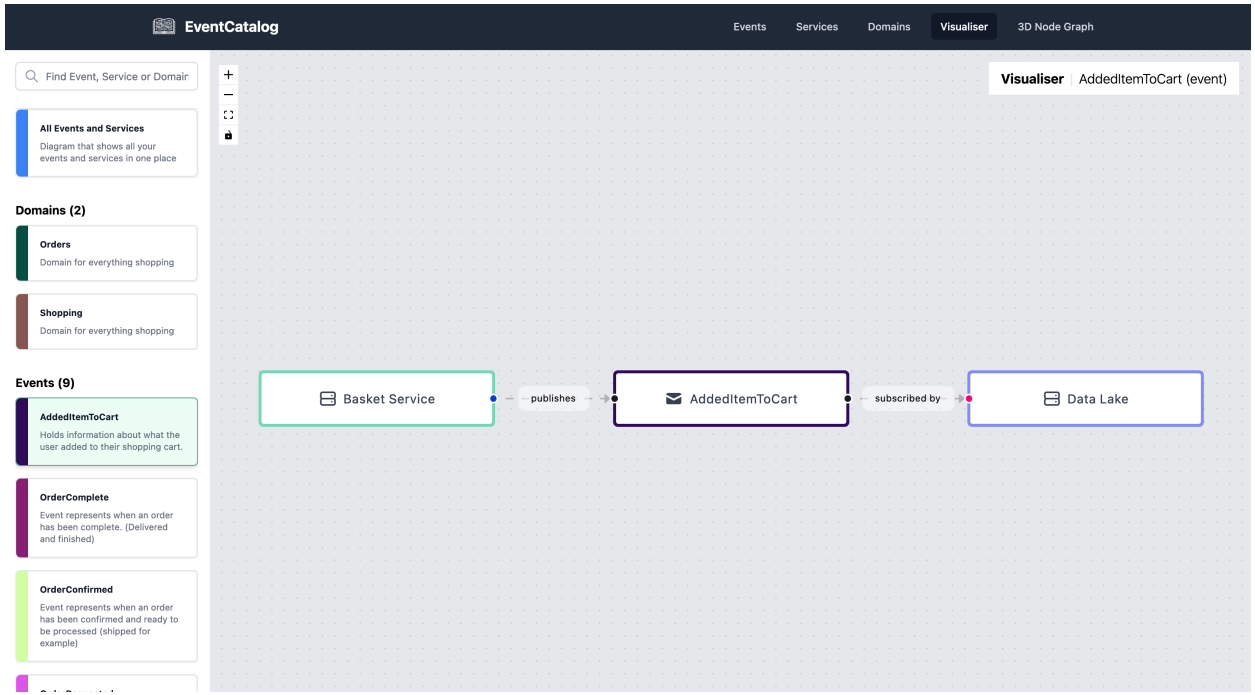

Backstage
GITHUB
DOCS
PLUGINS
BLOG
DEMOS
NEWSLETTER
Search
Overview
Getting Started
Local Development
Core Features
Software Catalog
Overview
The Life of an Entity
Catalog Configuration
System Model
YAML File Format
Entity References
Well-known Annotations
Well-known Relations
spec.type [required]
The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

Cupid

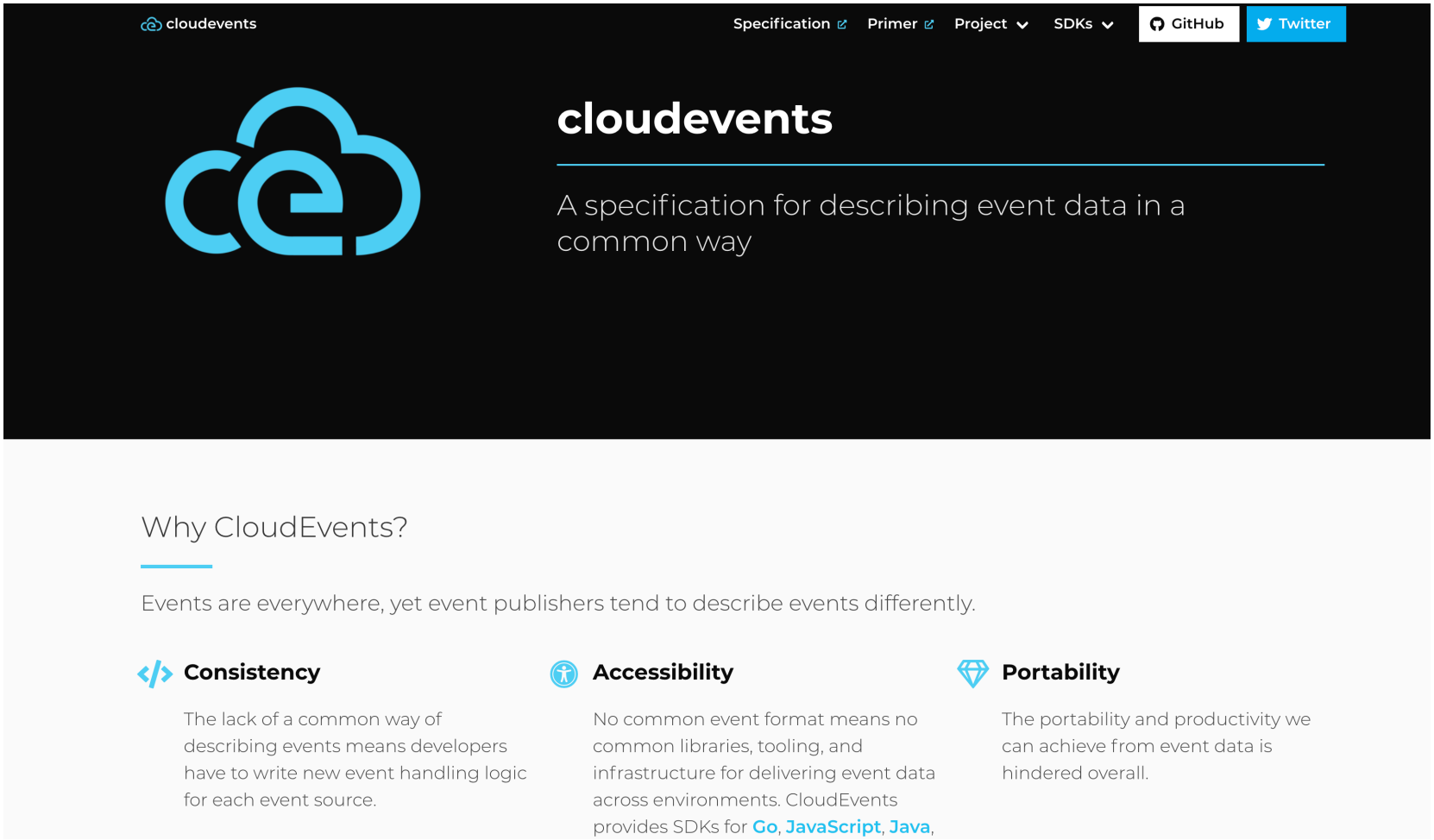


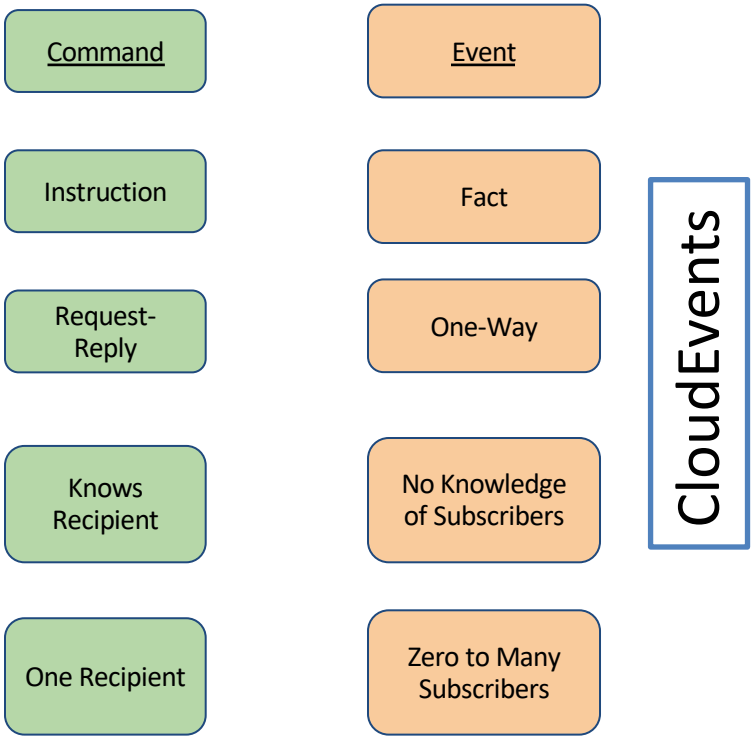
https://github.com/asyncapi/cupid

Event Catalog



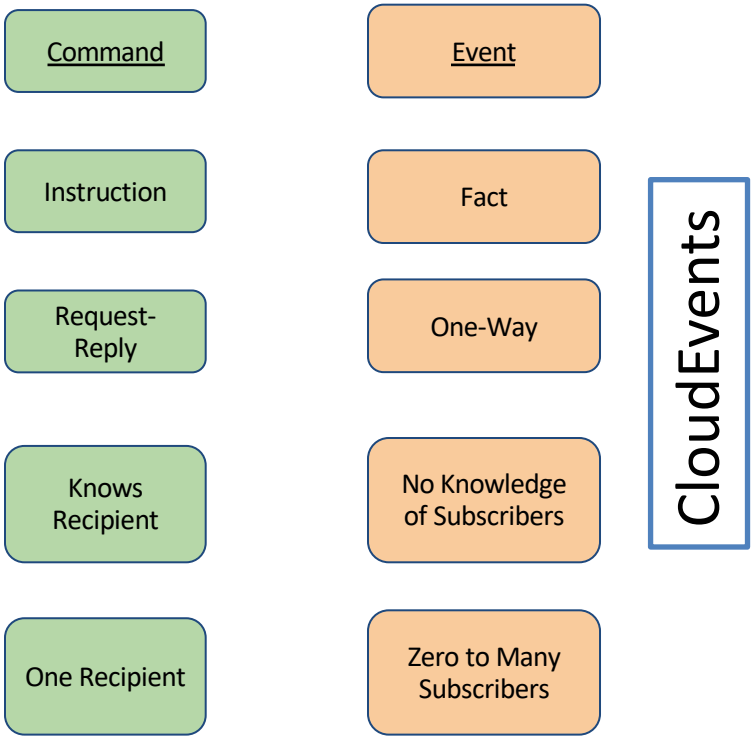
https://github.com/boyney123/eventcatalog


cloudevents
Specification
Primer
Project
SDKs
GitHub
Twitter
cloudevents
A specification for describing event data in a common way
Why CloudEvents?
Events are everywhere, yet event publishers tend to describe events differently.
Consistency
The lack of a common way of describing events means developers have to write new event handling logic for each event source.
Accessibility
No common event format means no common libraries, tooling, and infrastructure for delivering event data across environments. CloudEvents provides SDKs for Go, JavaScript, Java,
Portability
The portability and productivity we can achieve from event data is hindered overall.

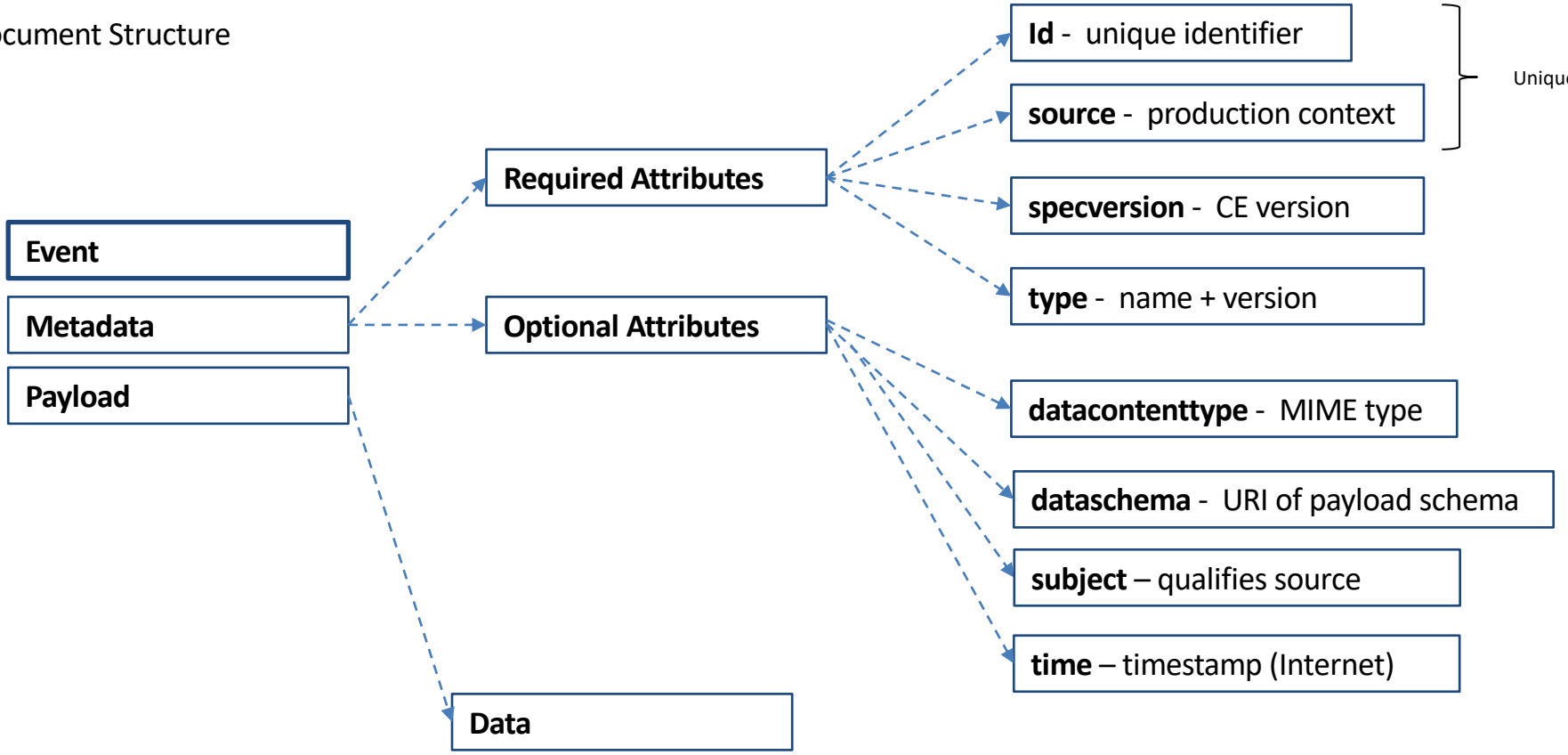

Command
Instruction
Request-Reply
Knows Recipient
One Recipient
Event
Fact
One-Way
No Knowledge of Subscribers
Zero to Many Subscribers
CloudEvents

No Routing Information

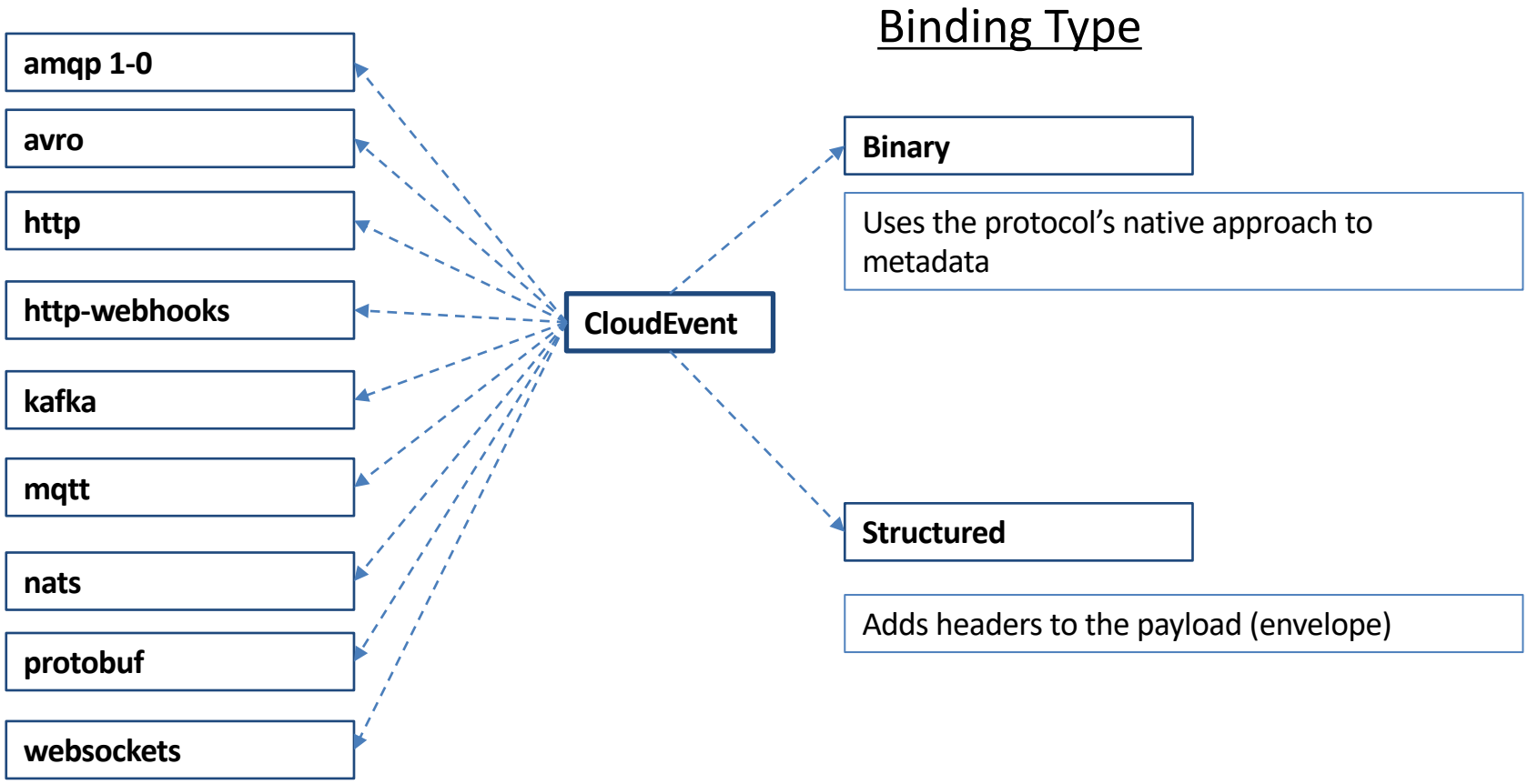
- Protocol Specific Concern
- Event May Flow



Document Structure



Protocol Binding



Binary

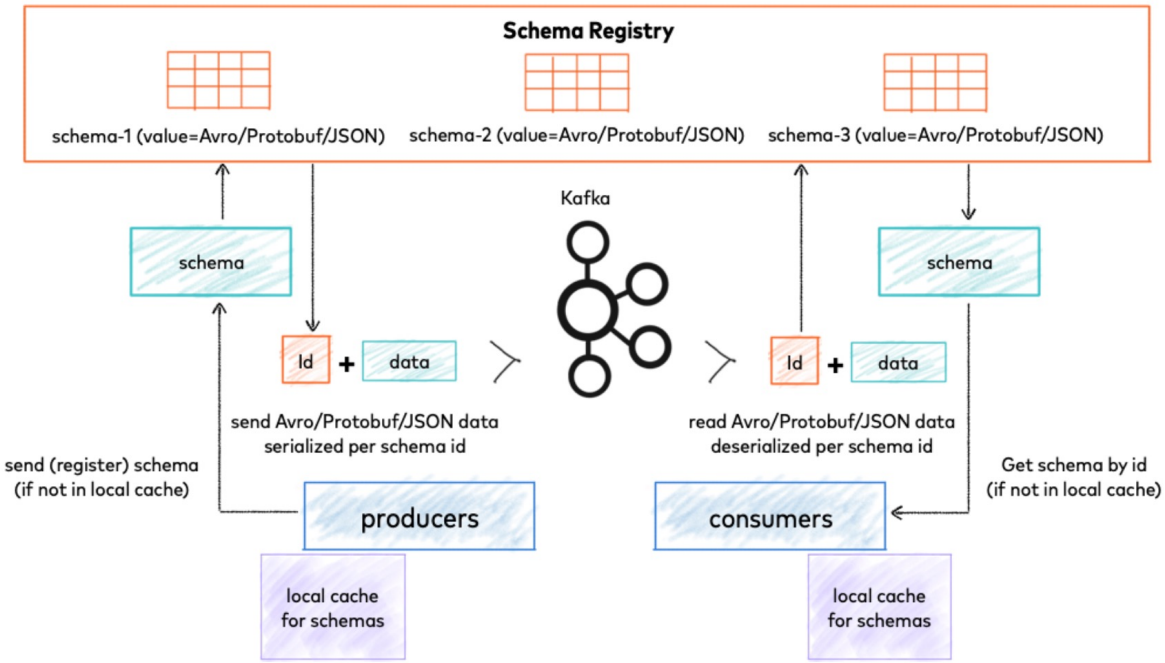
---------- Message ----------
Topic Name: mytopic
---------- key ----------
Key: mykey
---------- headers ----------
ce_specversion: "1.0"
ce_type: "com.example.someevent"
ce_source: "/mycontext/subcontext"
ce_id: "1234-1234-1234"
ce_time: "2018-04-05T03:56:24Z"
content-type: application/avro
----------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

Maintaining Event Schemas



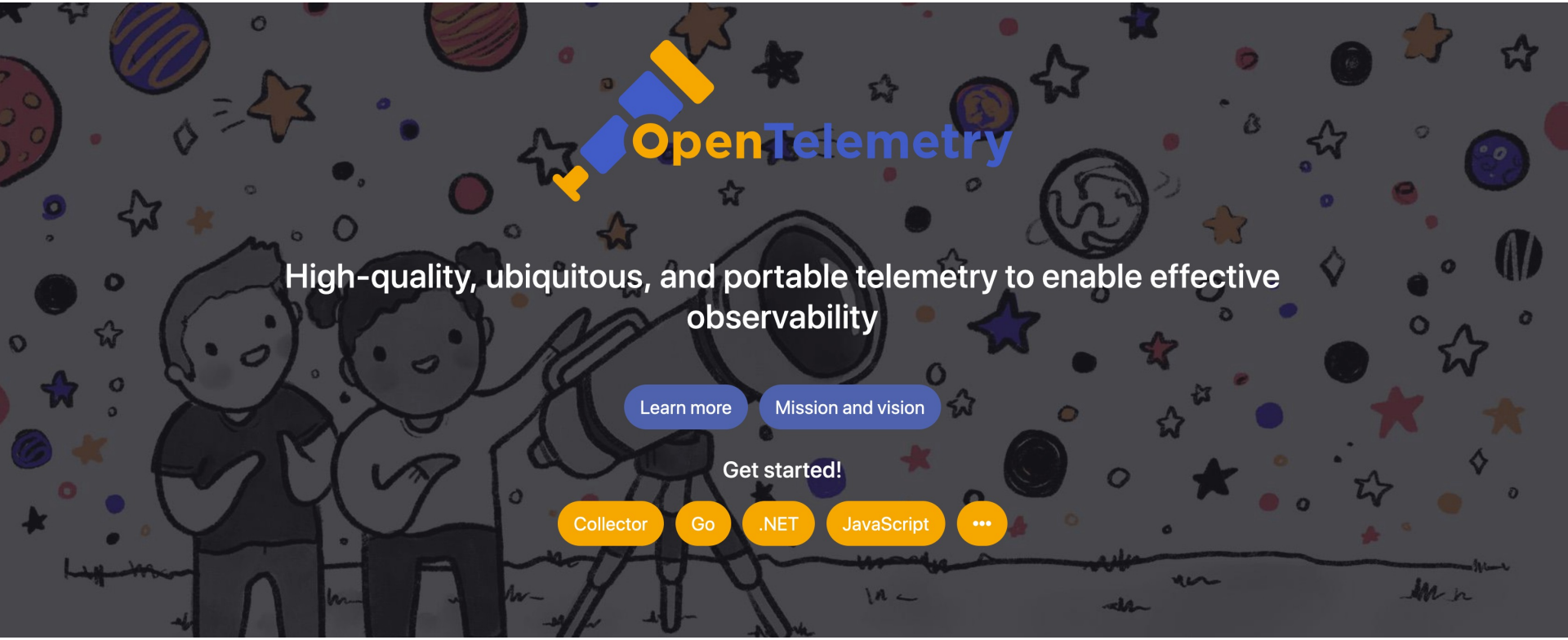
Confluent Schema Registry for storing and retrieving schemas

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

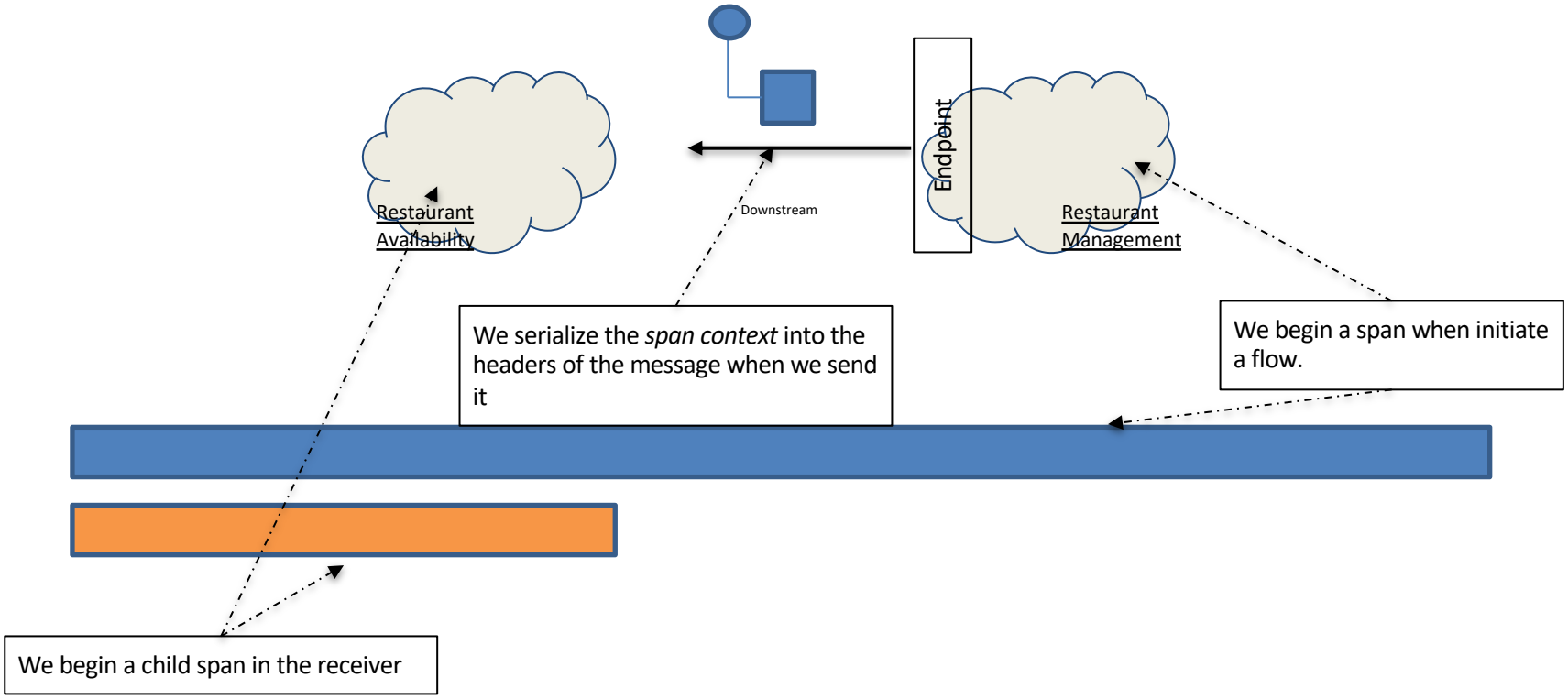
2
0

OpenTelemetry etc.

7 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



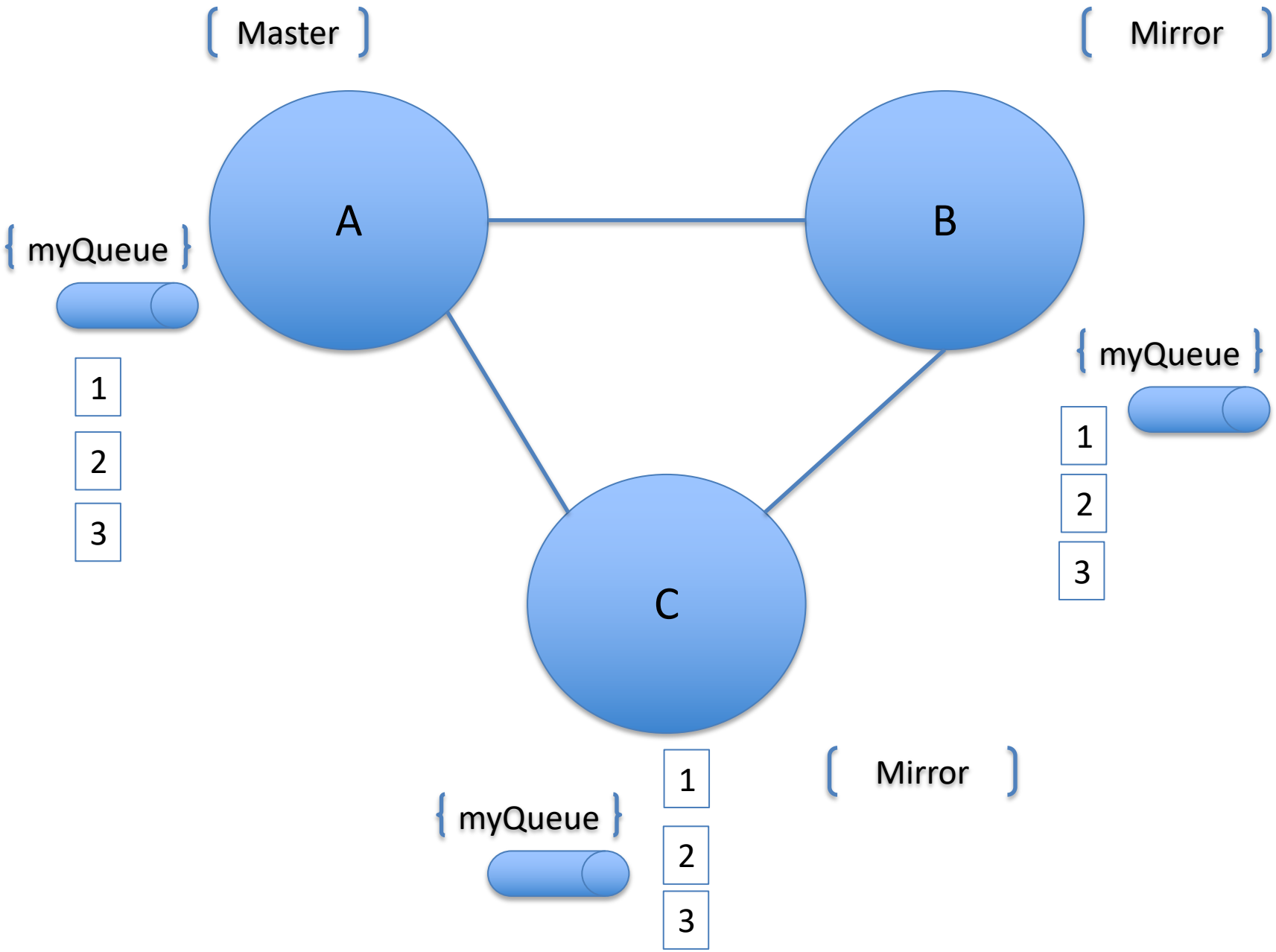
Spans

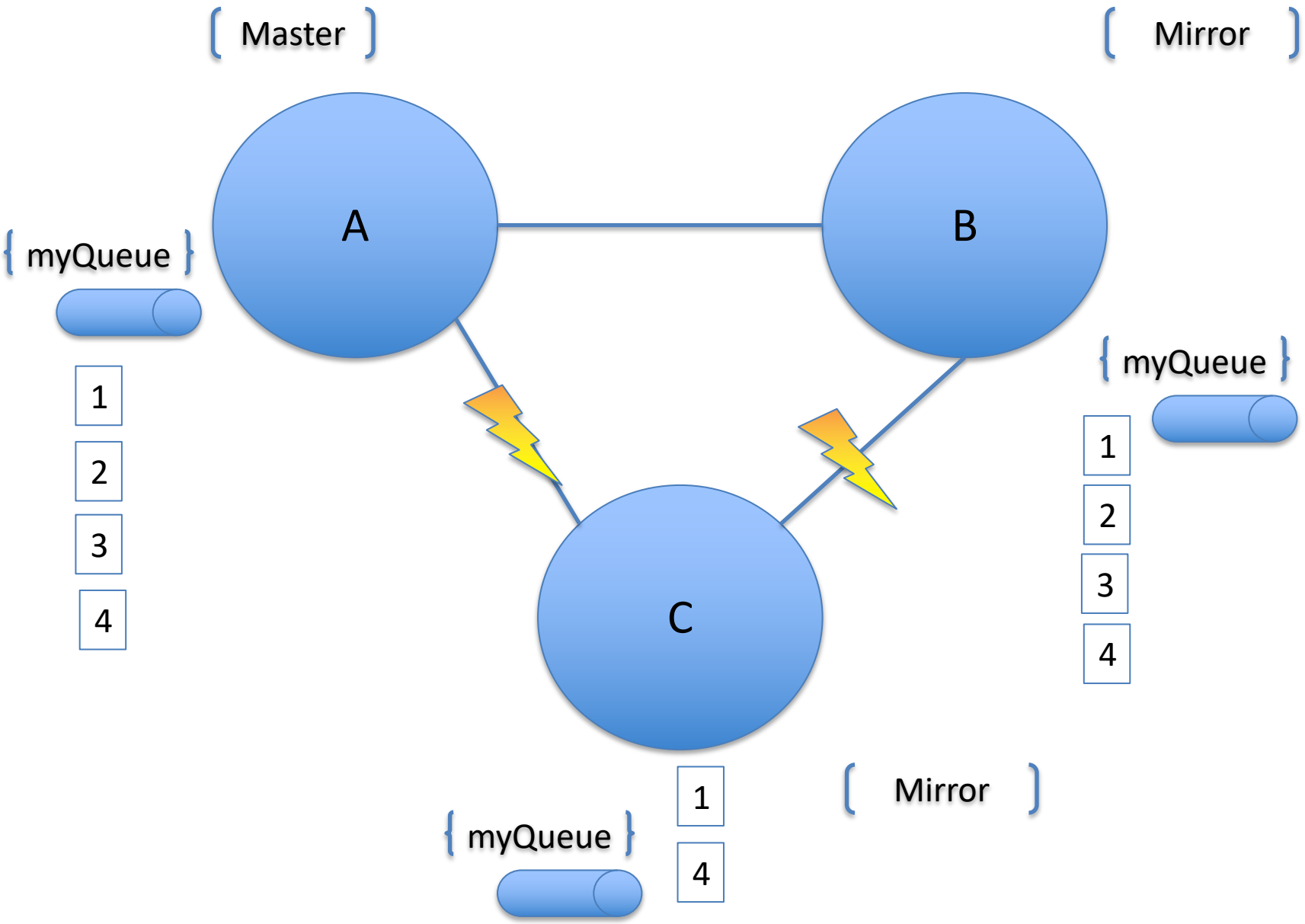
- Operation Name
- Start and Finish Timestamps
- Span Context (span_id, trace_id)
- Attributes (standard and defined)
- Events

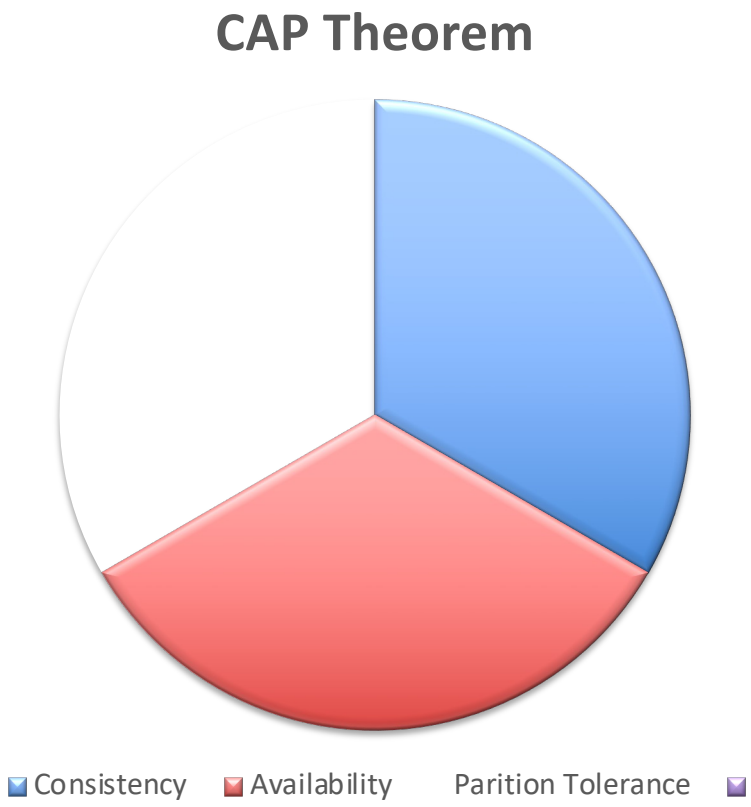
8 CAP THEOREM

CAP Theorem

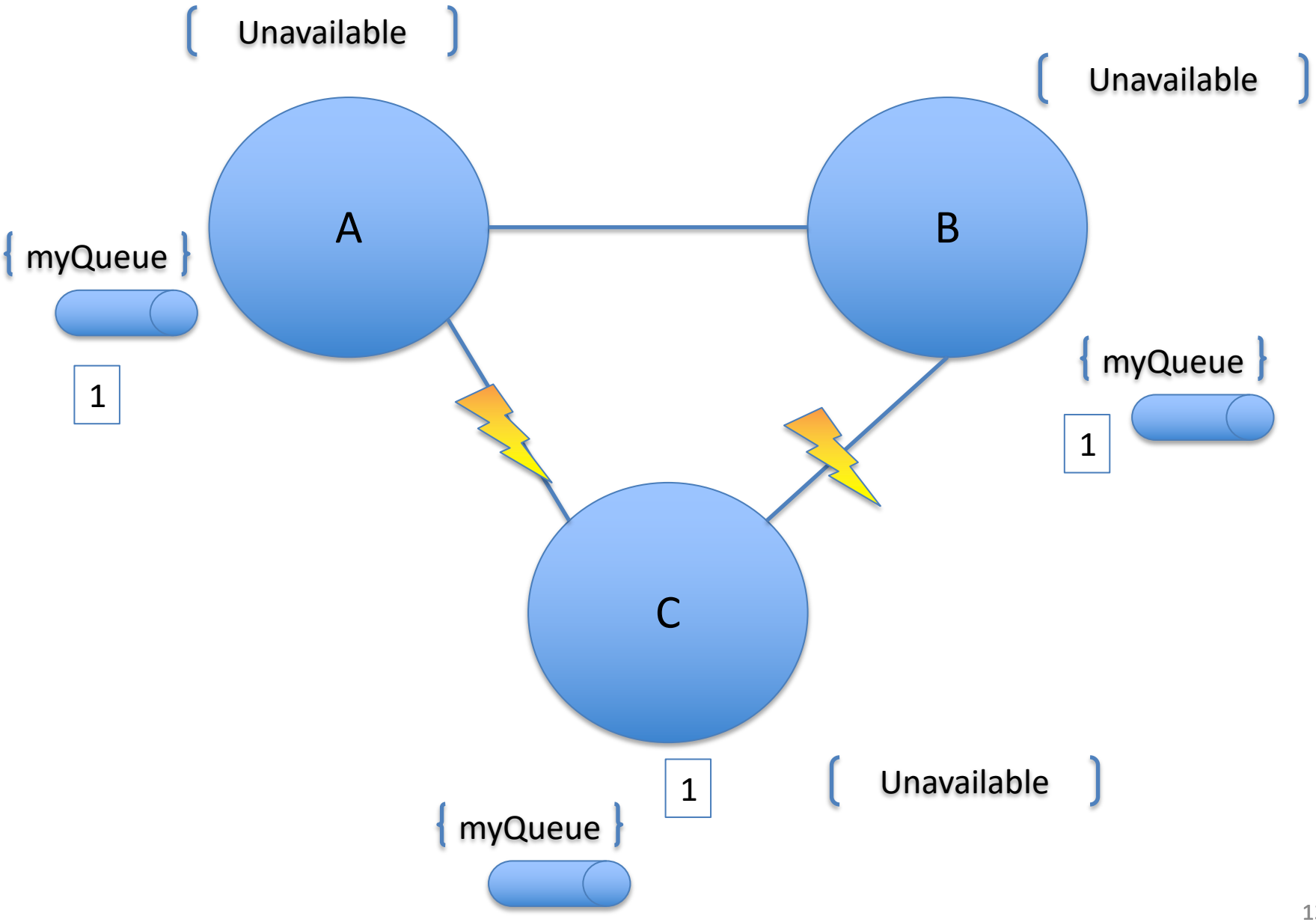
You can have at most two of these properties for any shared data system

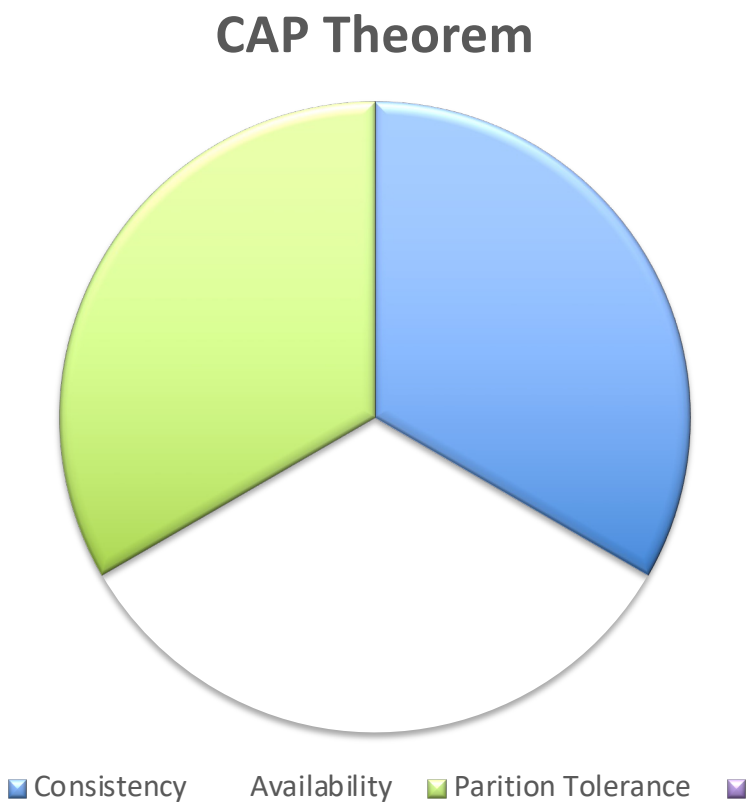

Master
Mirror
A
B
myQueue
1
2
3
myQueue
1
2
3
C
Mirror
myQueue
1
2
3


Master
Mirror
A
B
myQueue
1
2
3
4
myQueue
1
2
3
4
C
1
4
myQueue
Mirror

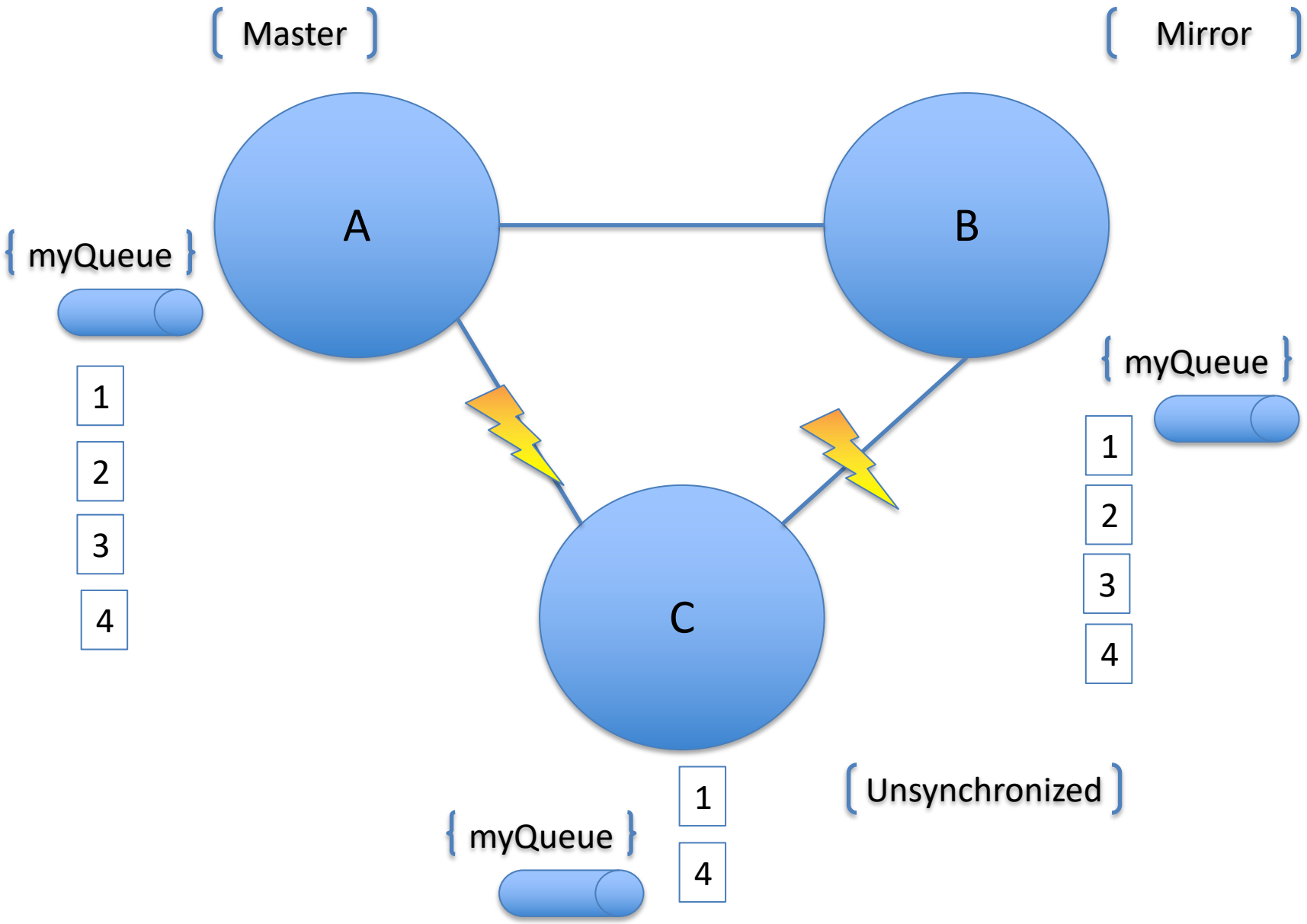


Forfeit Partitions

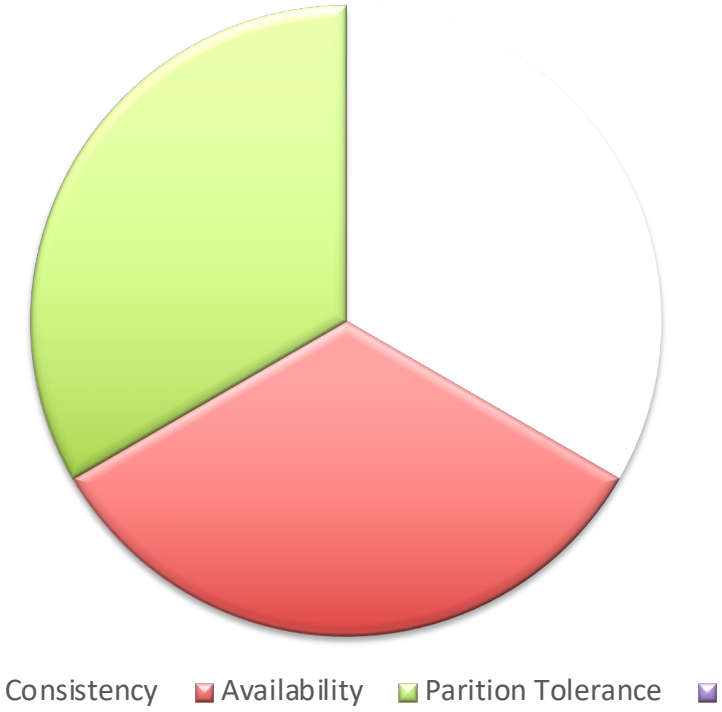

Unavailable
Unavailable
A
B
myQueue
1
myQueue
1
C
1
Unavailable
myQueue



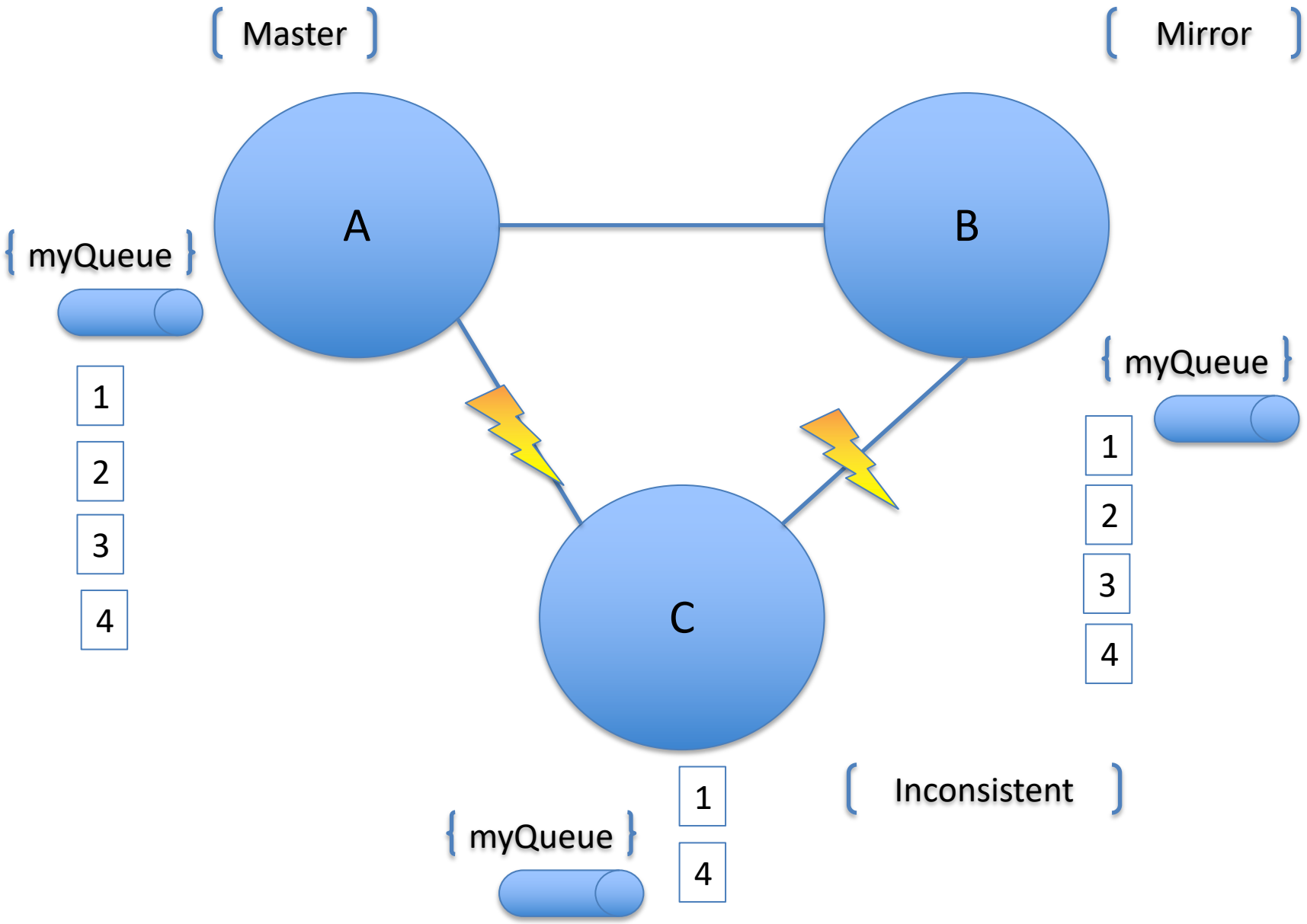
Forfeit Availability

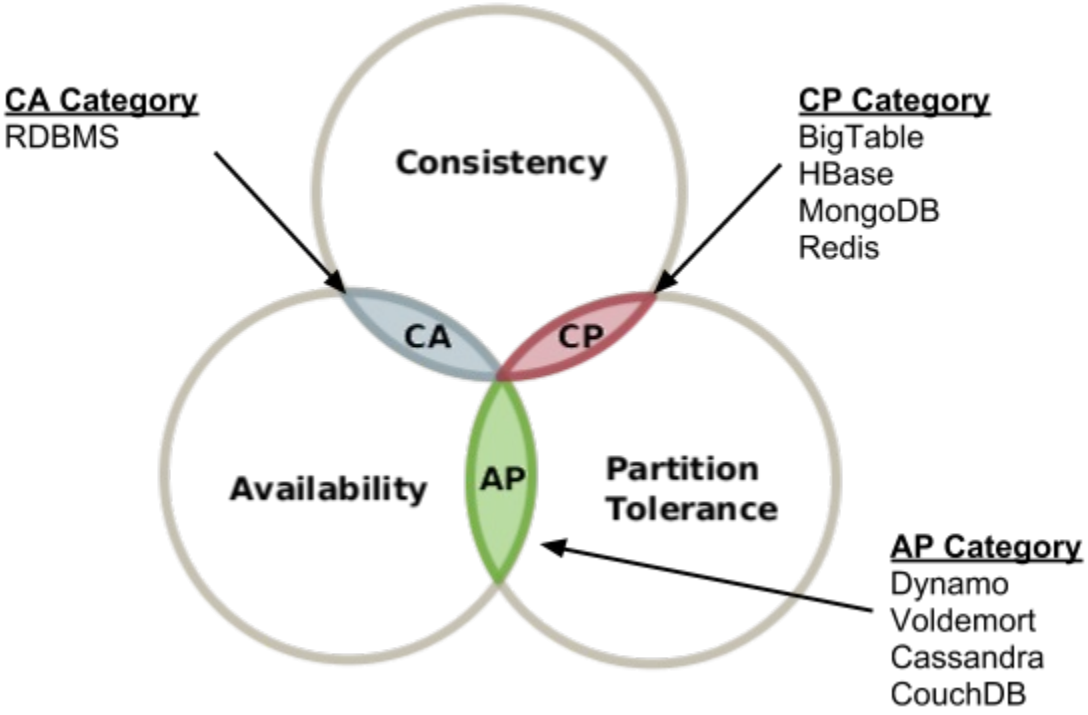

Master
Mirror
A
B
C
myQueue
1
2
3
4
myQueue
1
2
3
4
myQueue
1
4
Unsynchronized

CAP Theorem

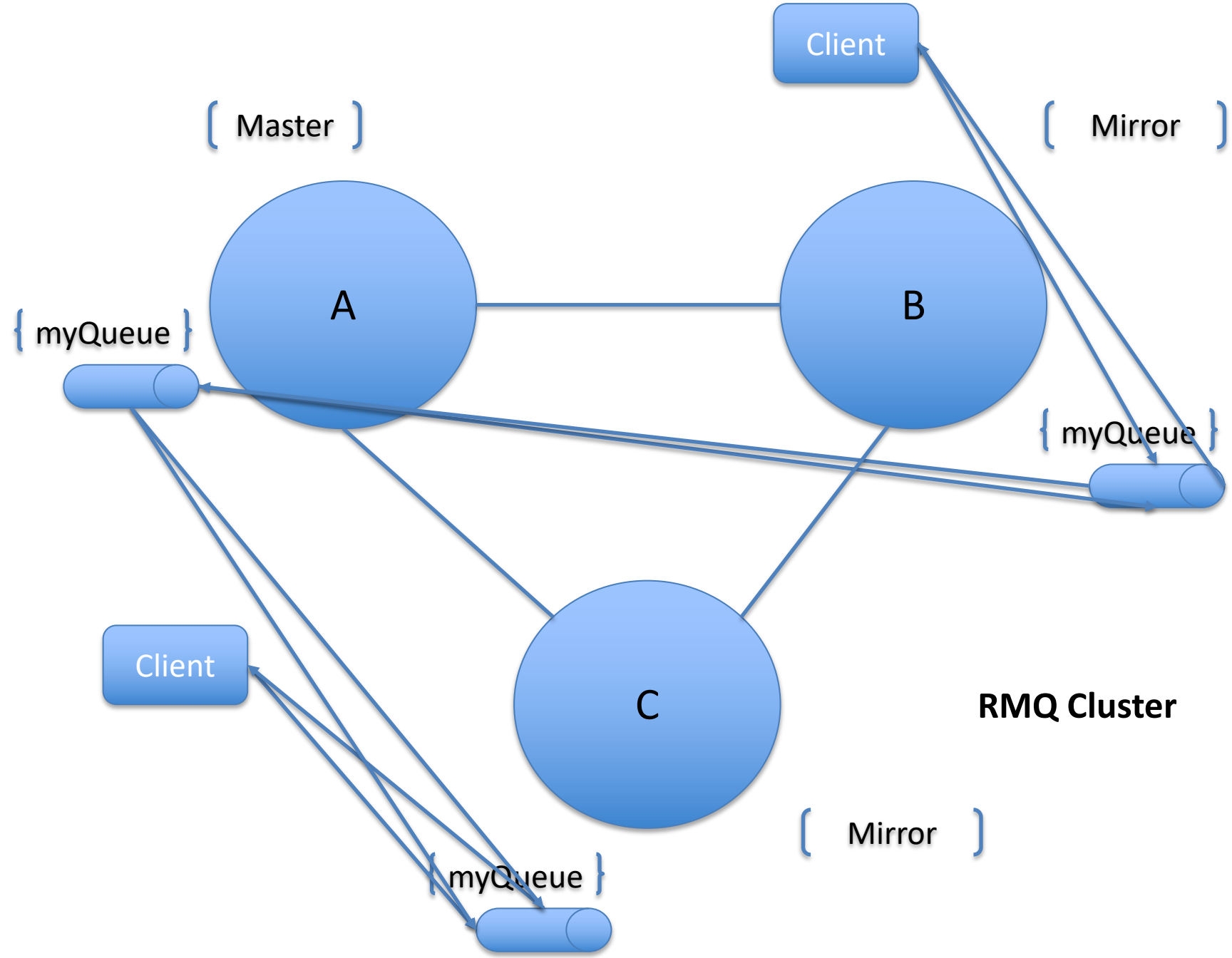


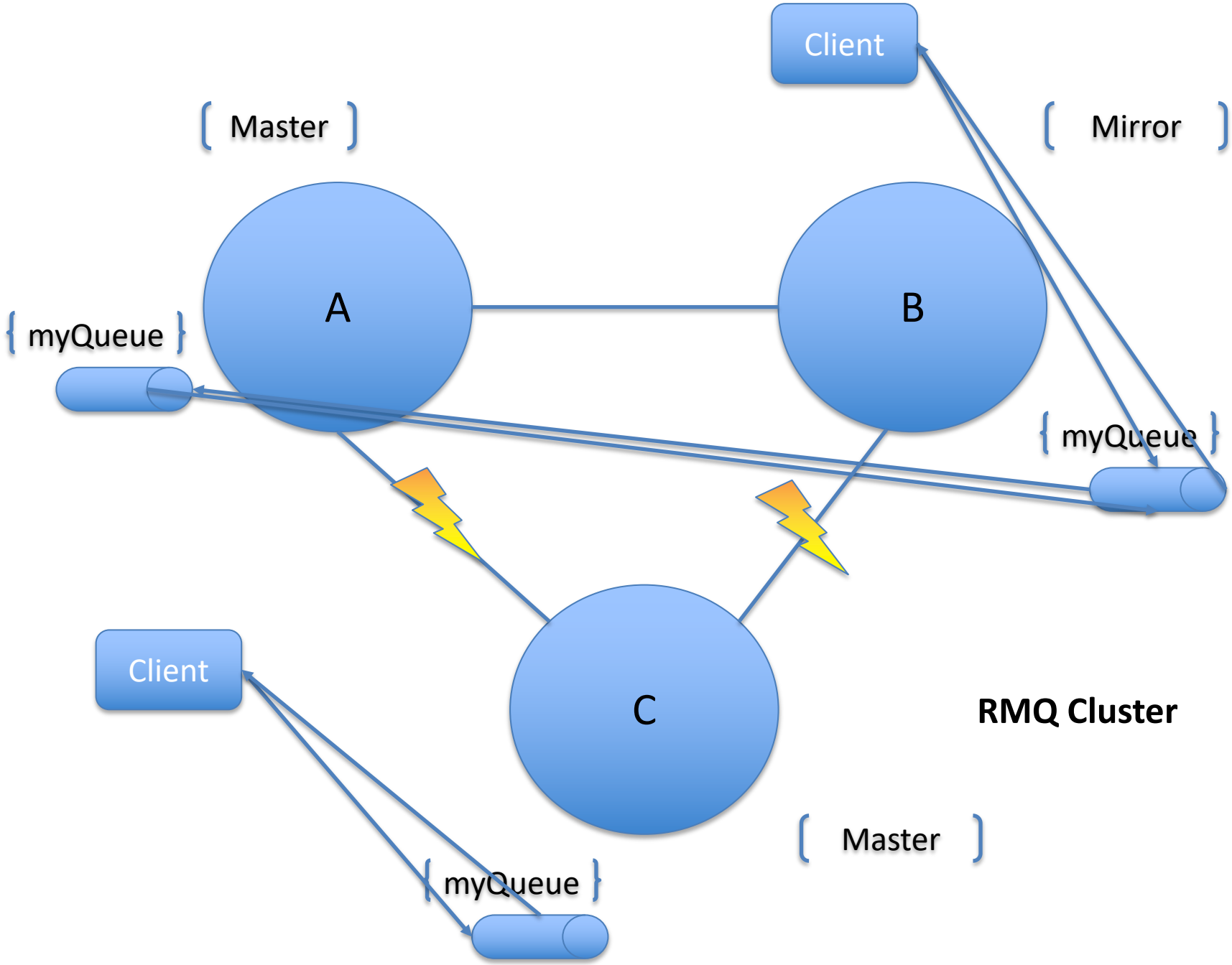
Forfeit Consistency

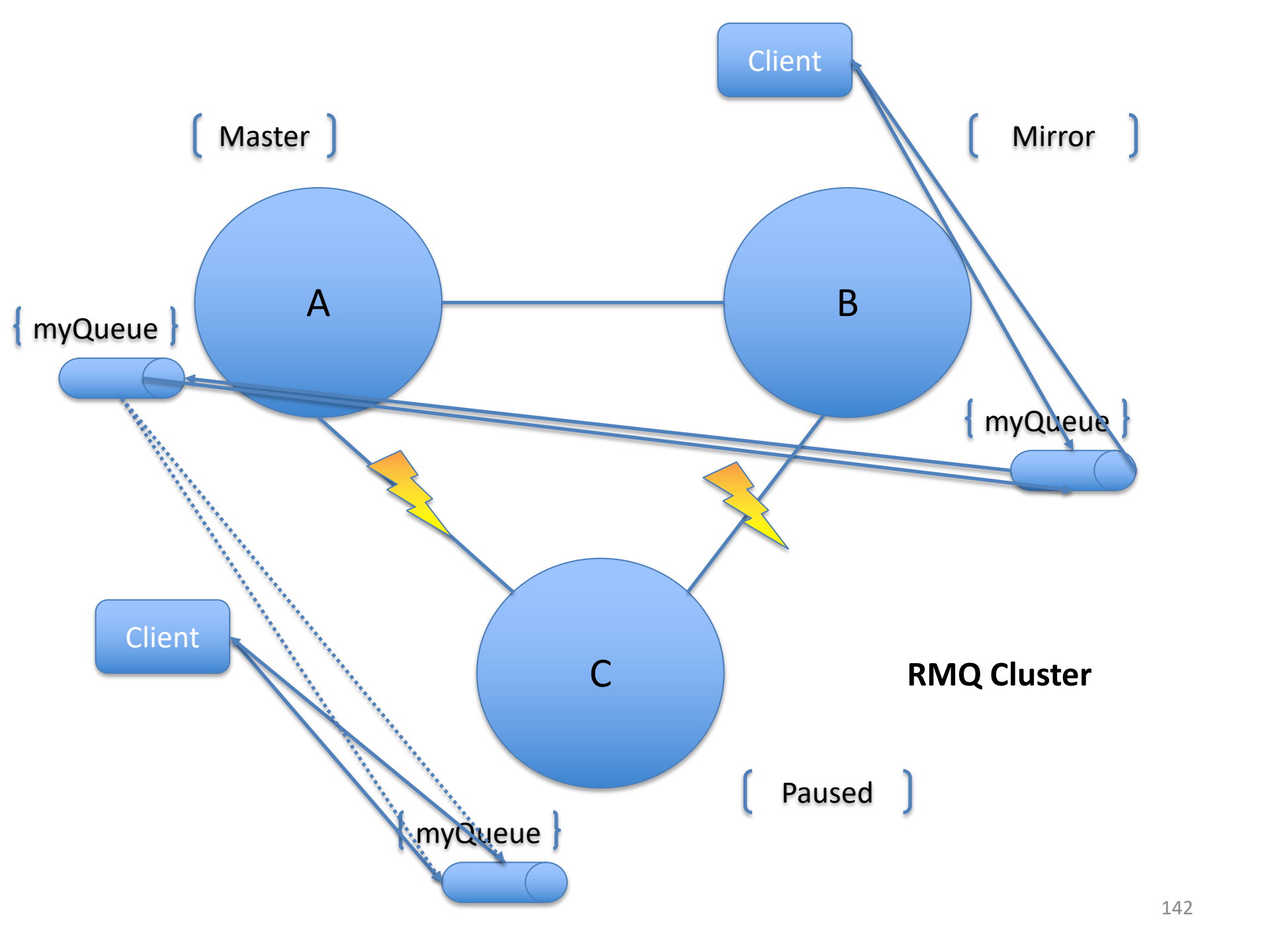

Master
Mirror
A
B
C
myQueue
1
2
3
4
myQueue
1
2
3
4
myQueue
1
4
Inconsistent

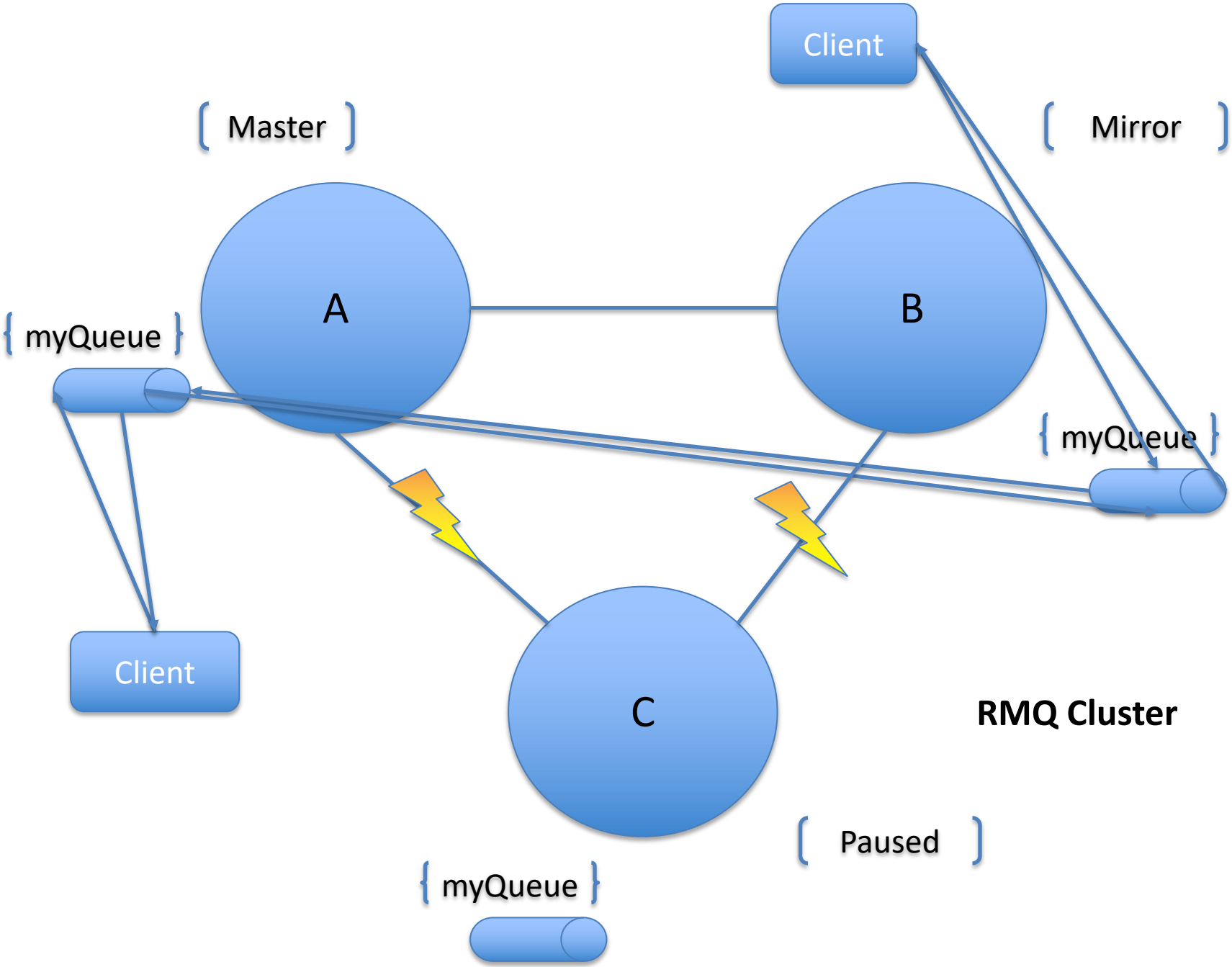

CA Category
RDBMS
Consistency
CP Category
BigTable
HBase
MongoDB
Redis
CA
CP
Availability
AP
Partition
Tolerance
AP Category
Dynamo
Voldemort
Cassandra
CouchDB

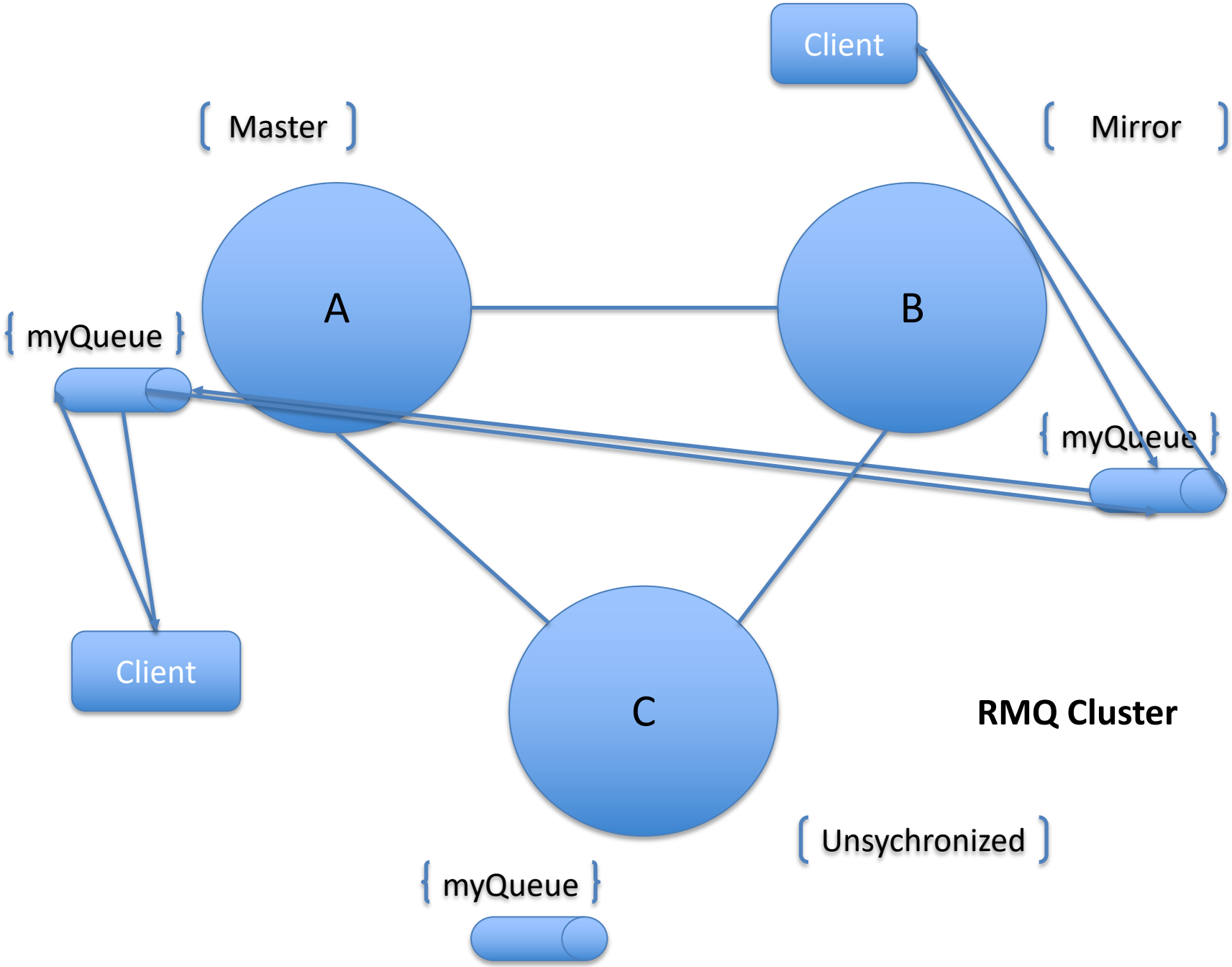
RabbitMQ and CAP Theorem

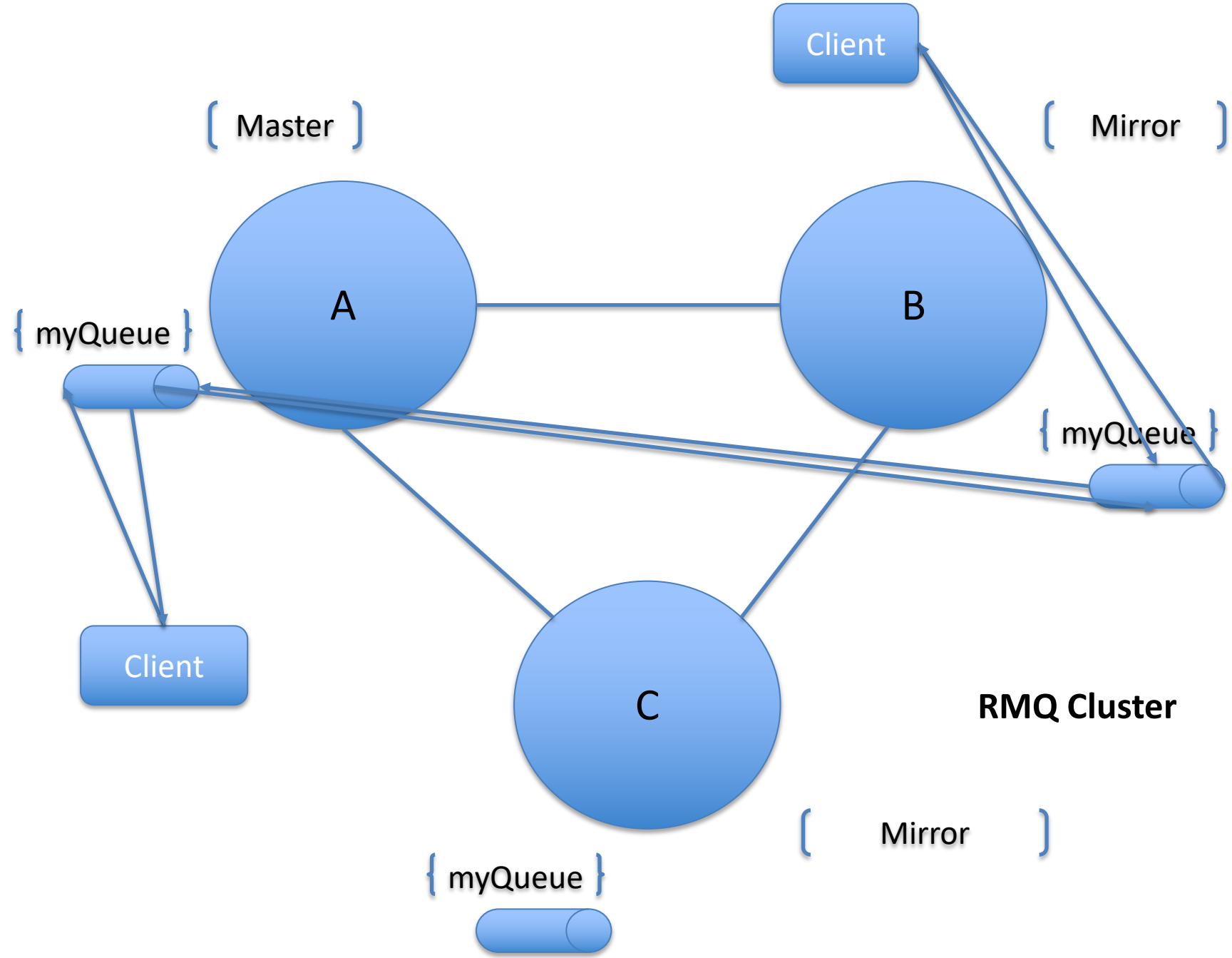

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

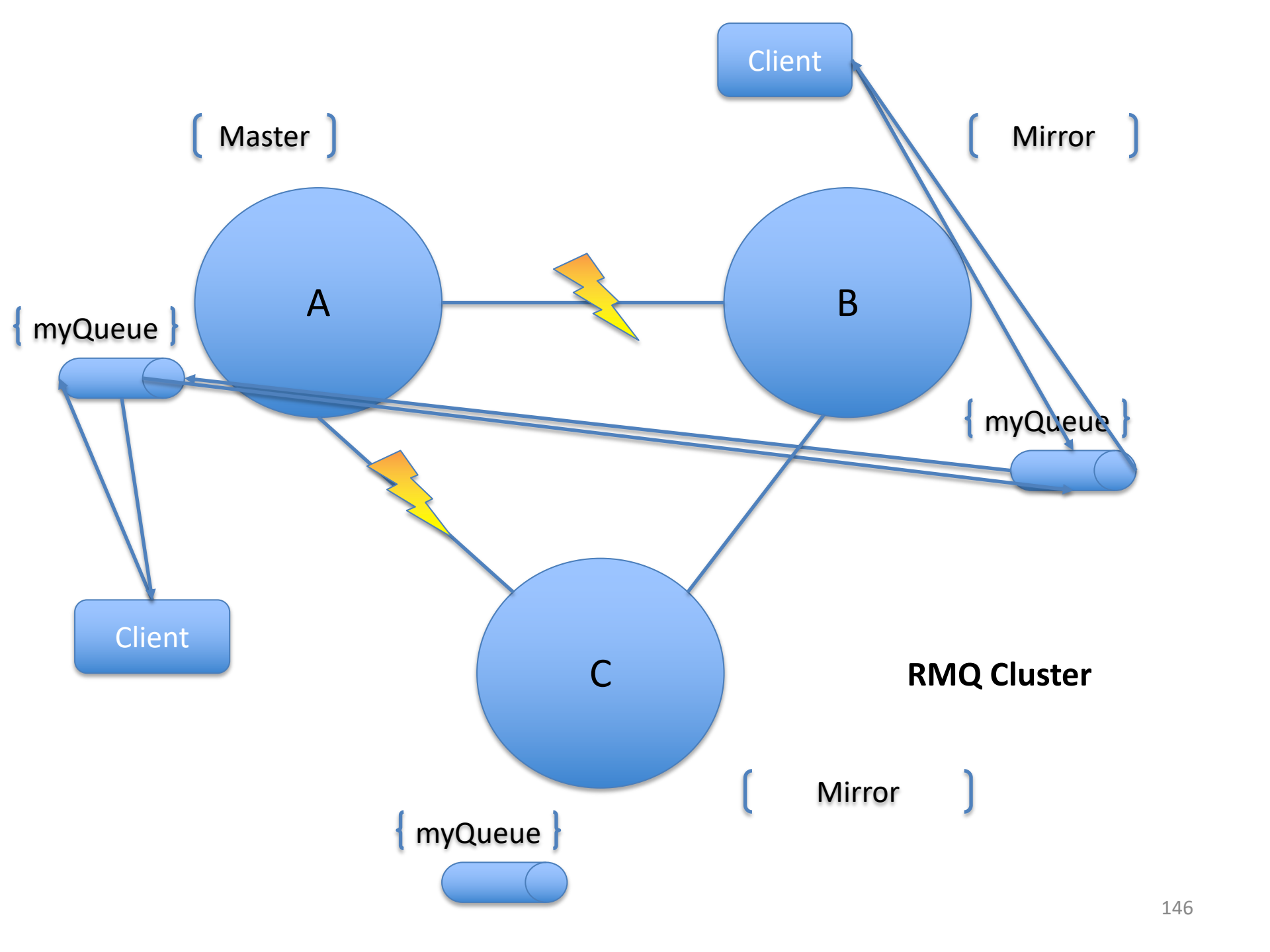

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Master
myQueue

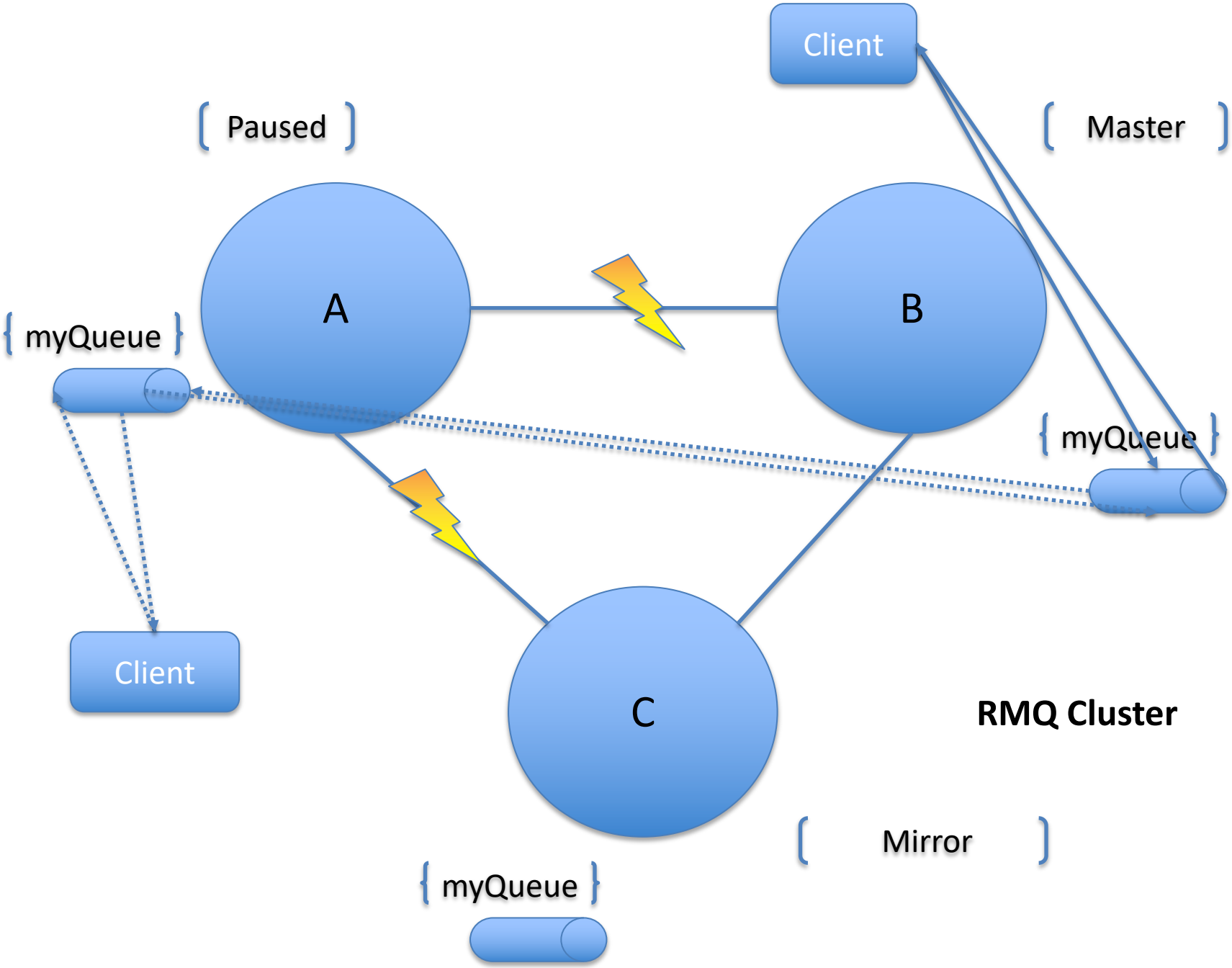

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Paused
myQueue

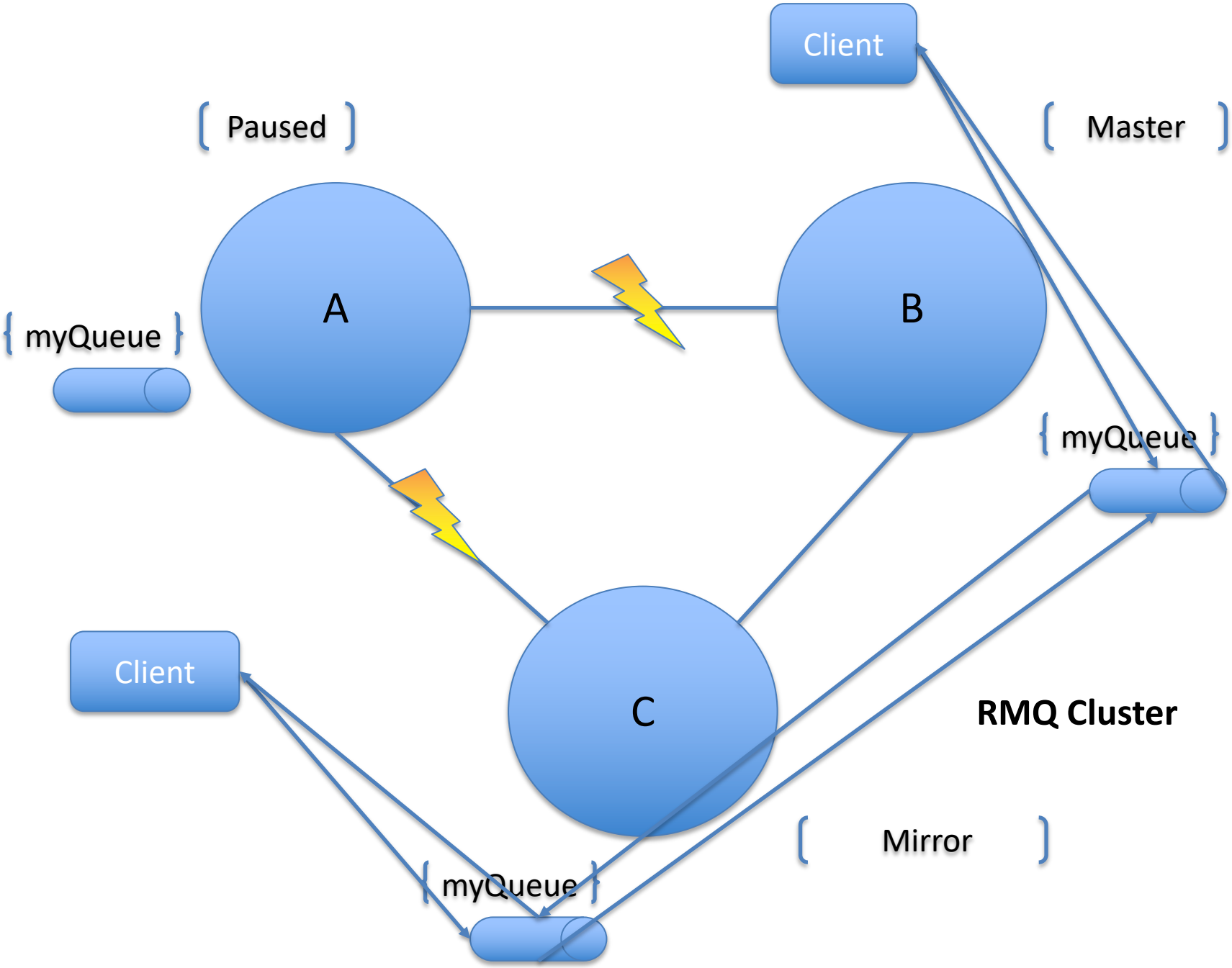

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Paused
myQueue

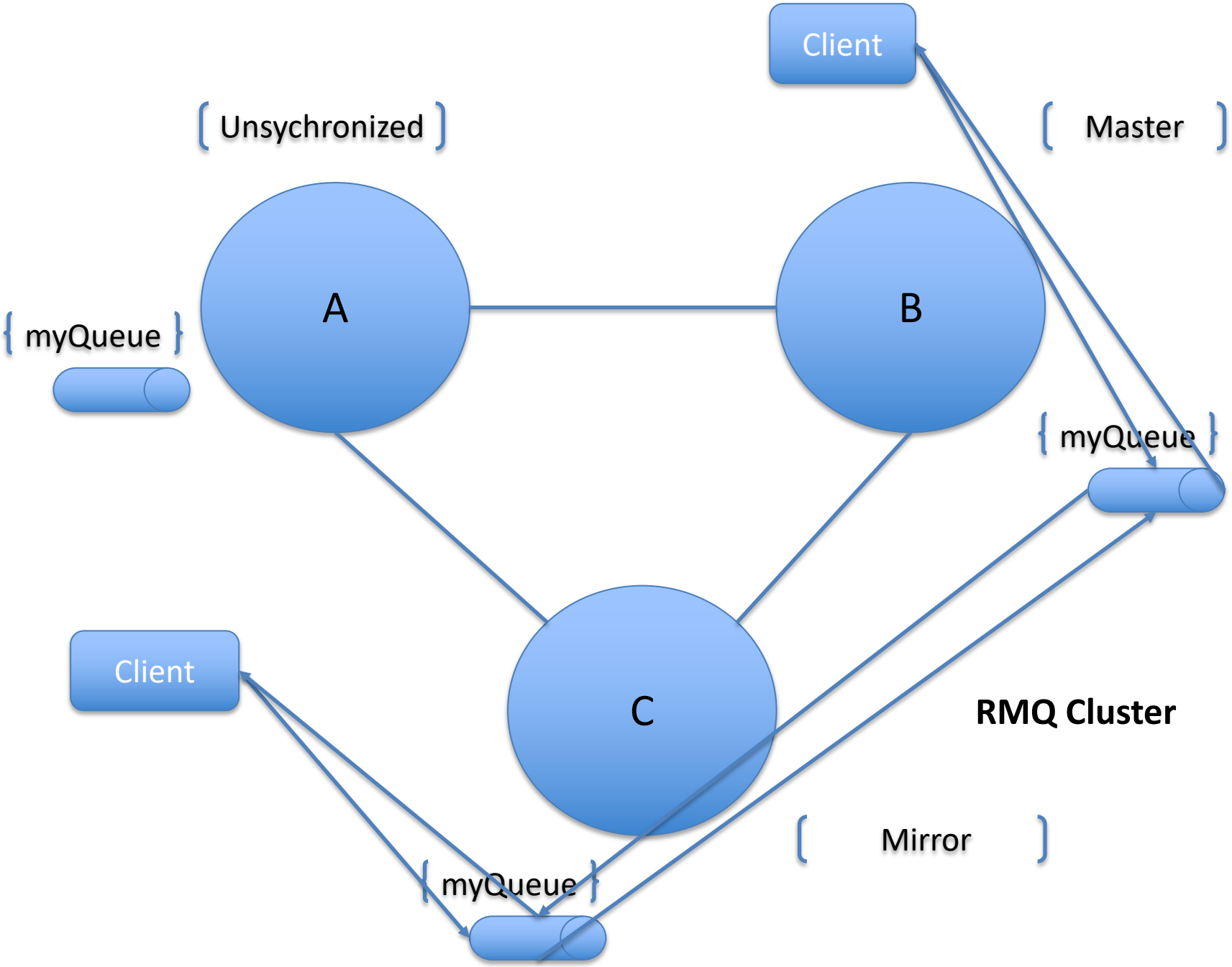

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Unsychronized
myQueue

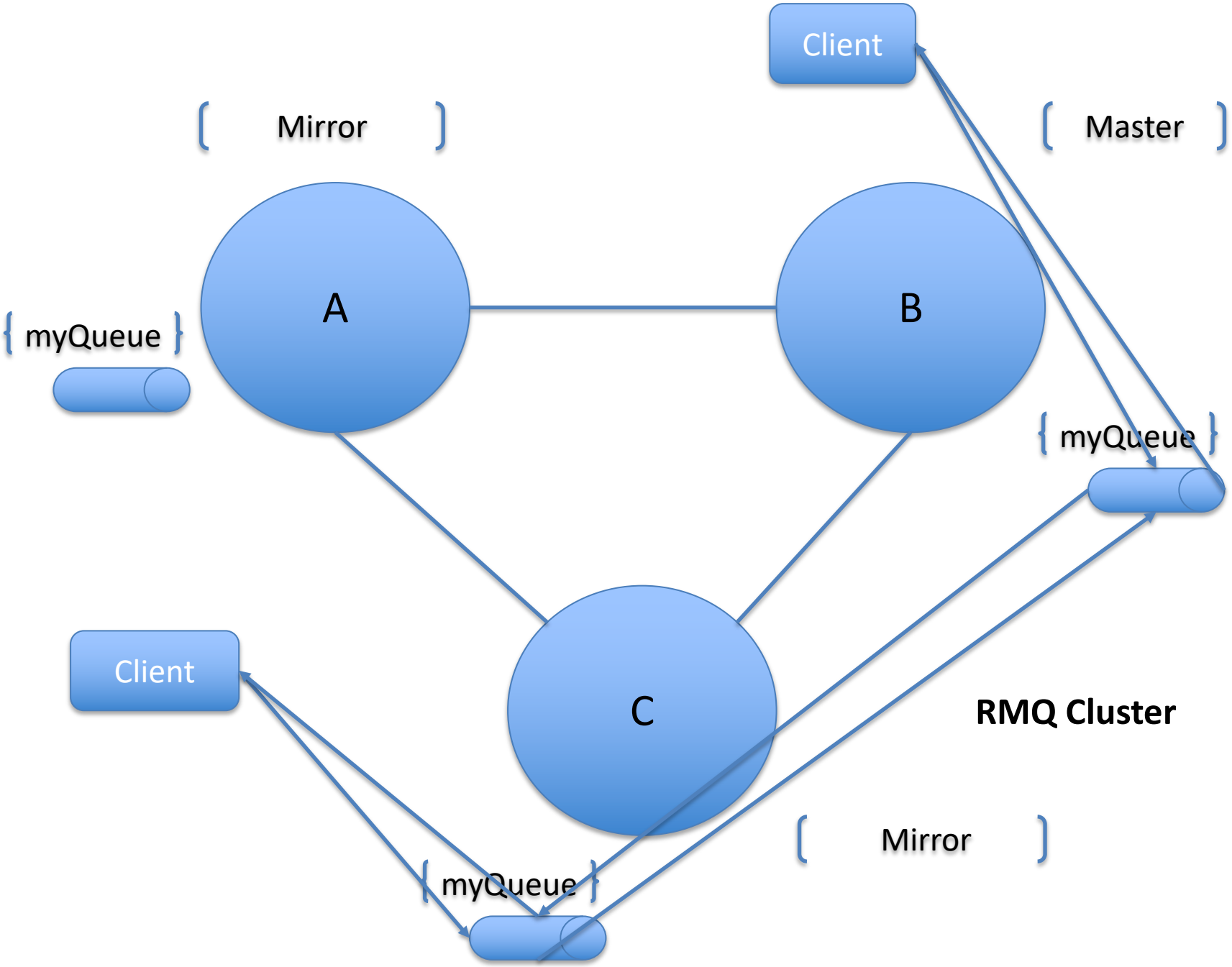

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

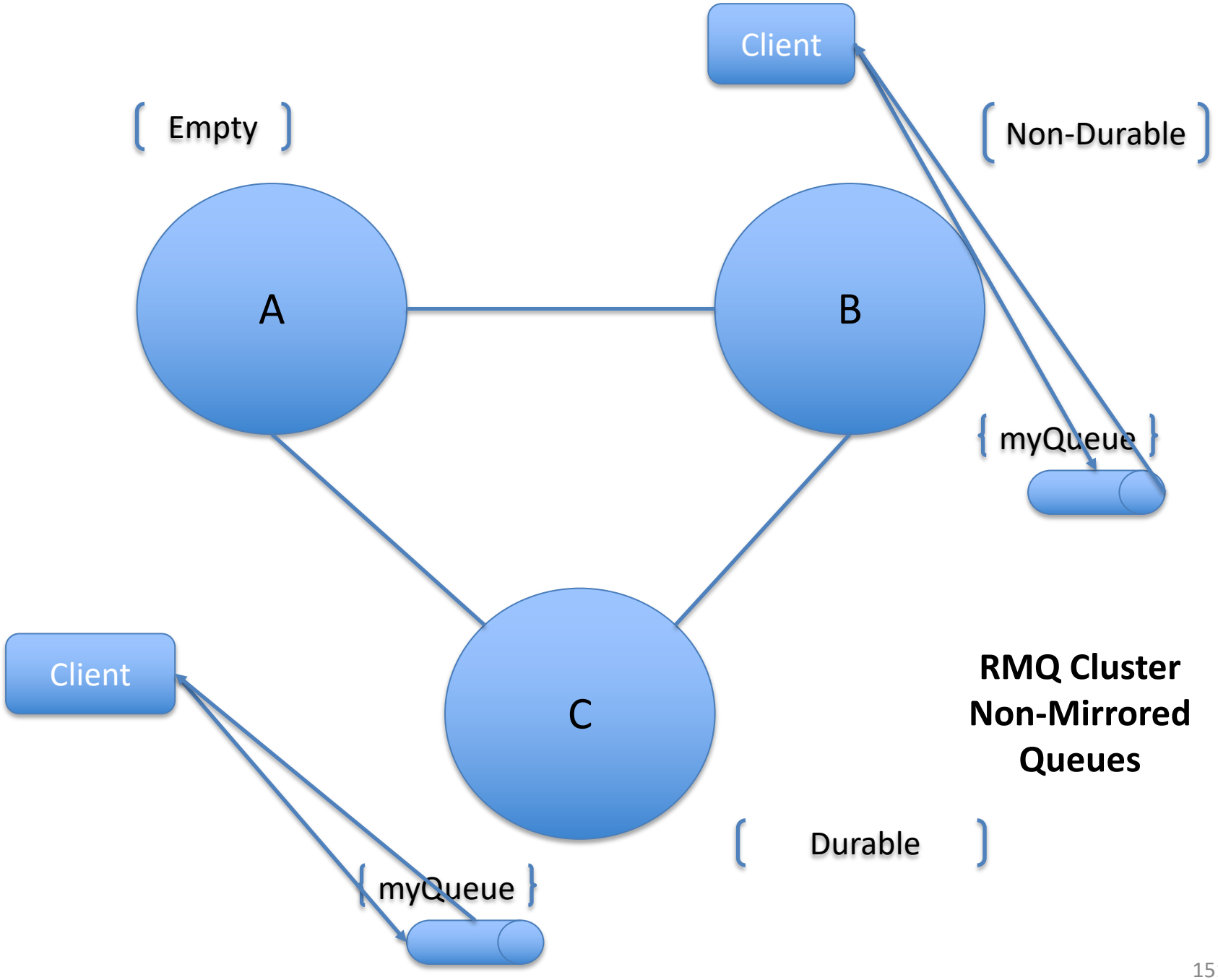

Client
Master
Mirror
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

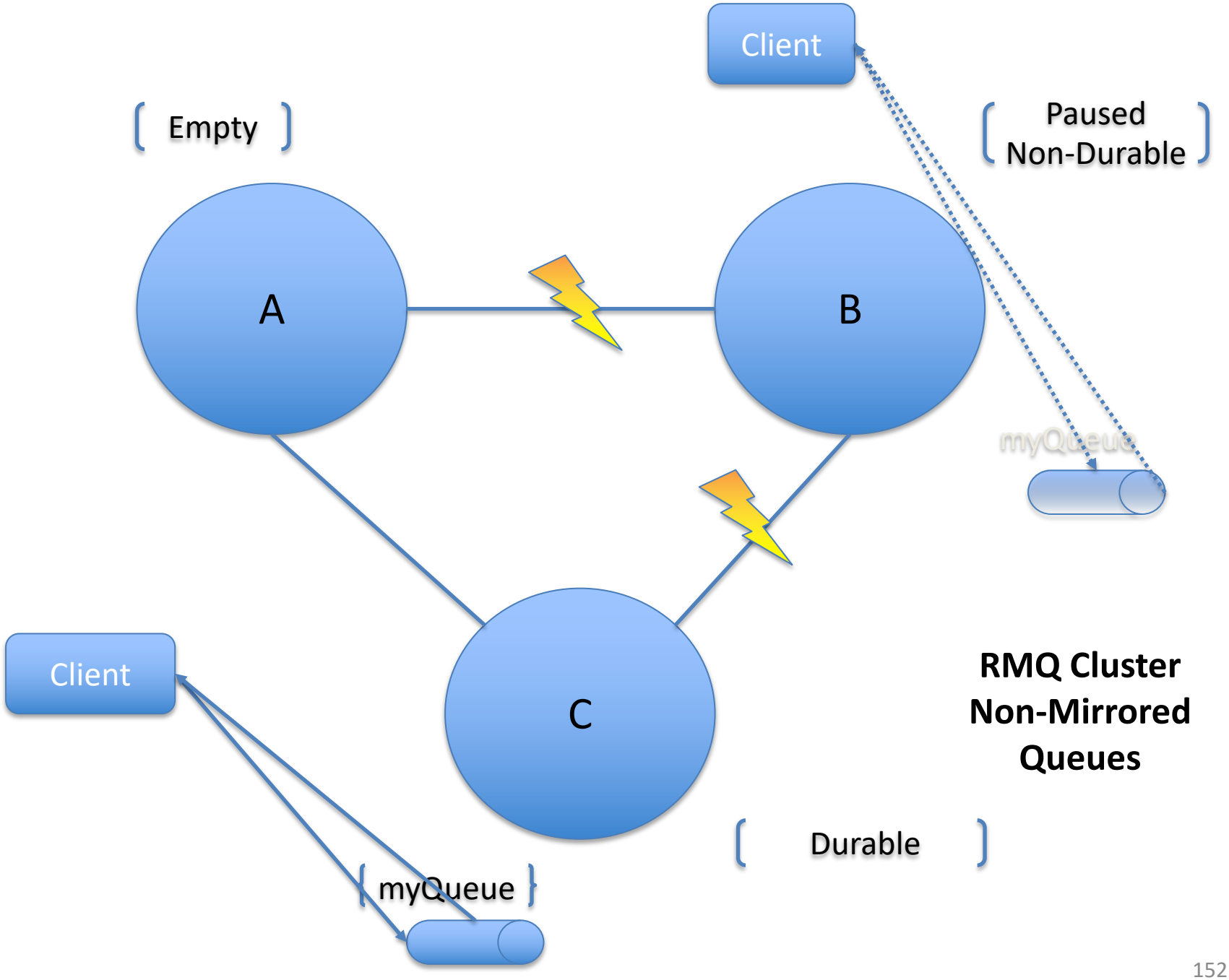

Client
Paused
Master
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

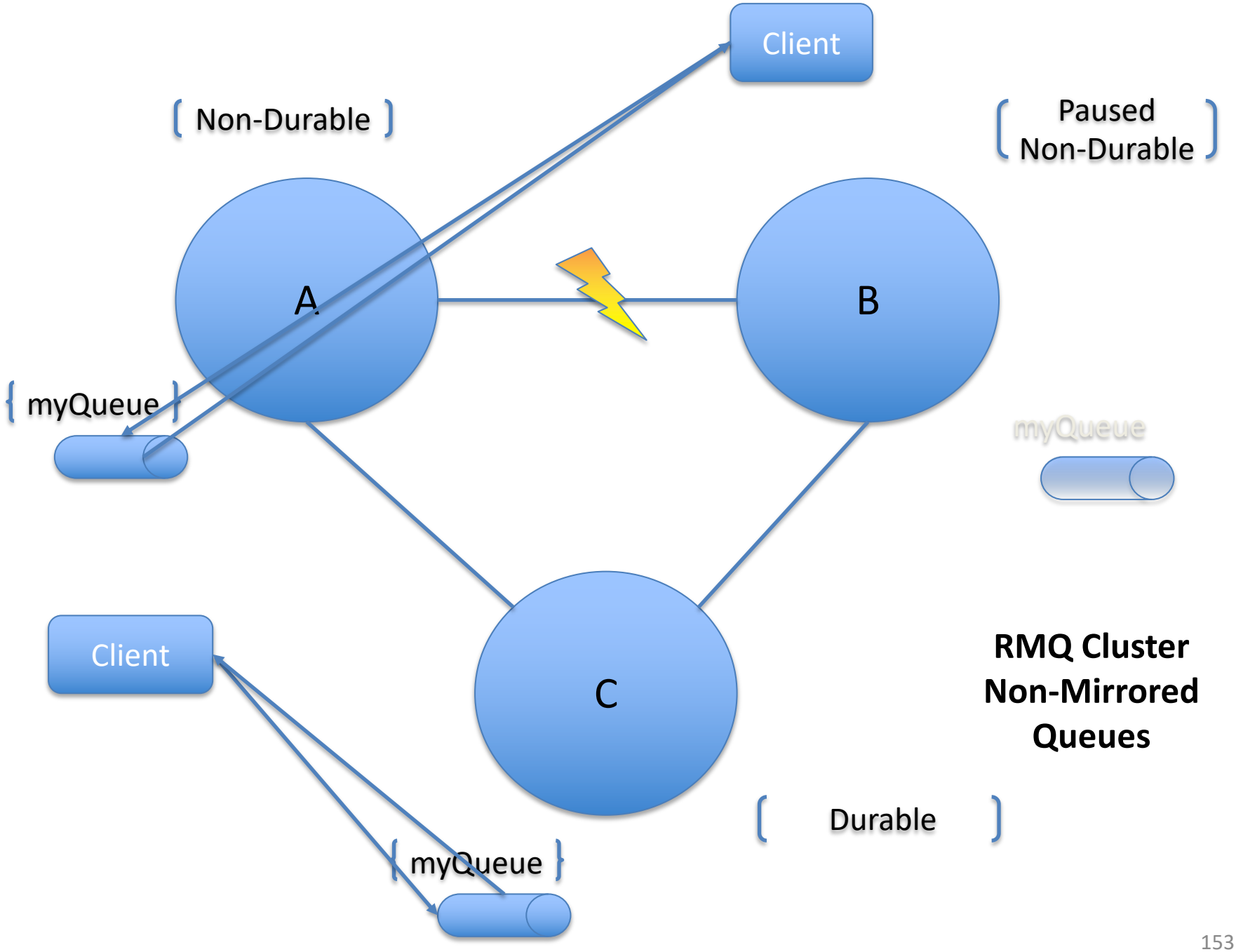

Client
Paused
Master
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

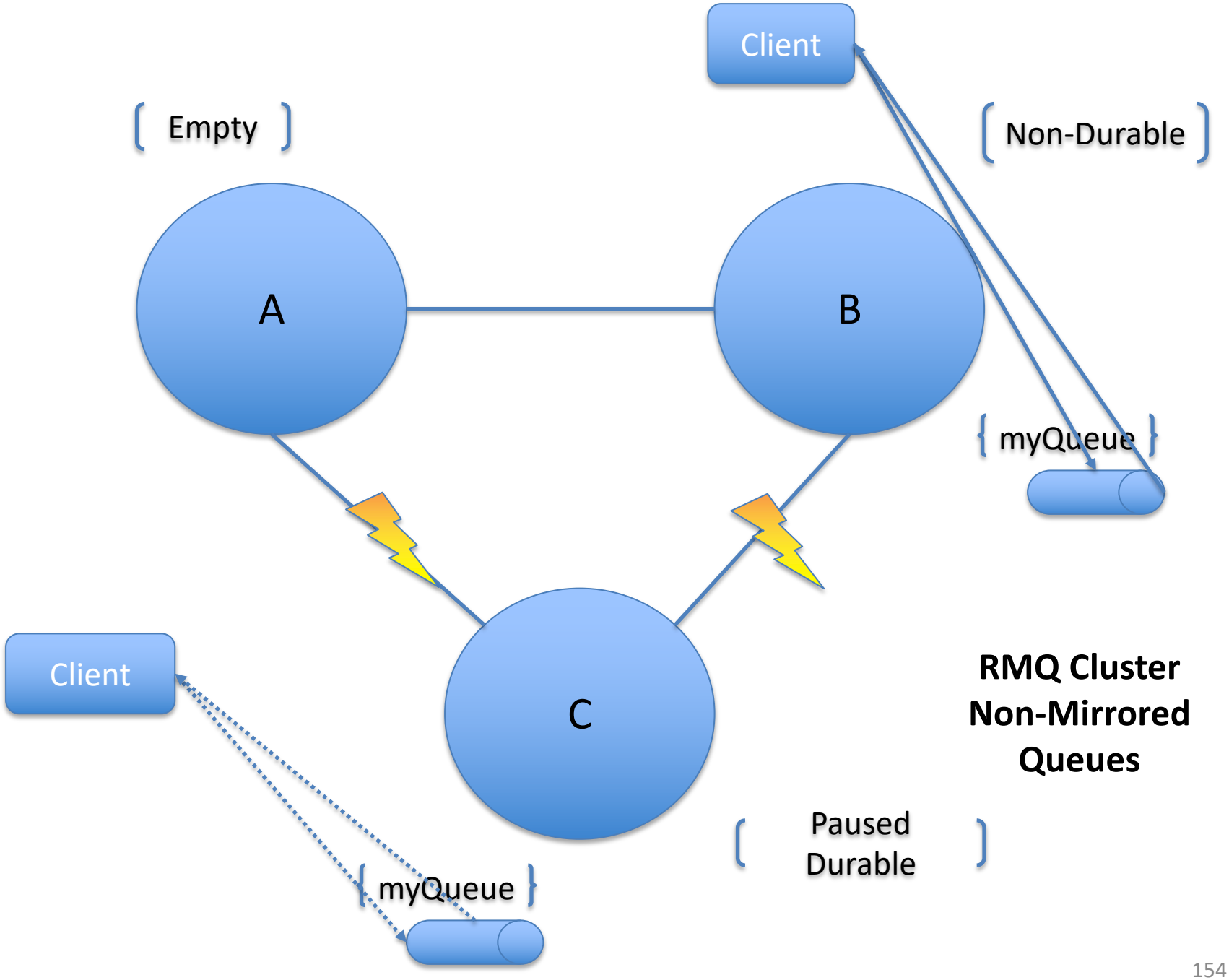

Client
Unsychronized
Master
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

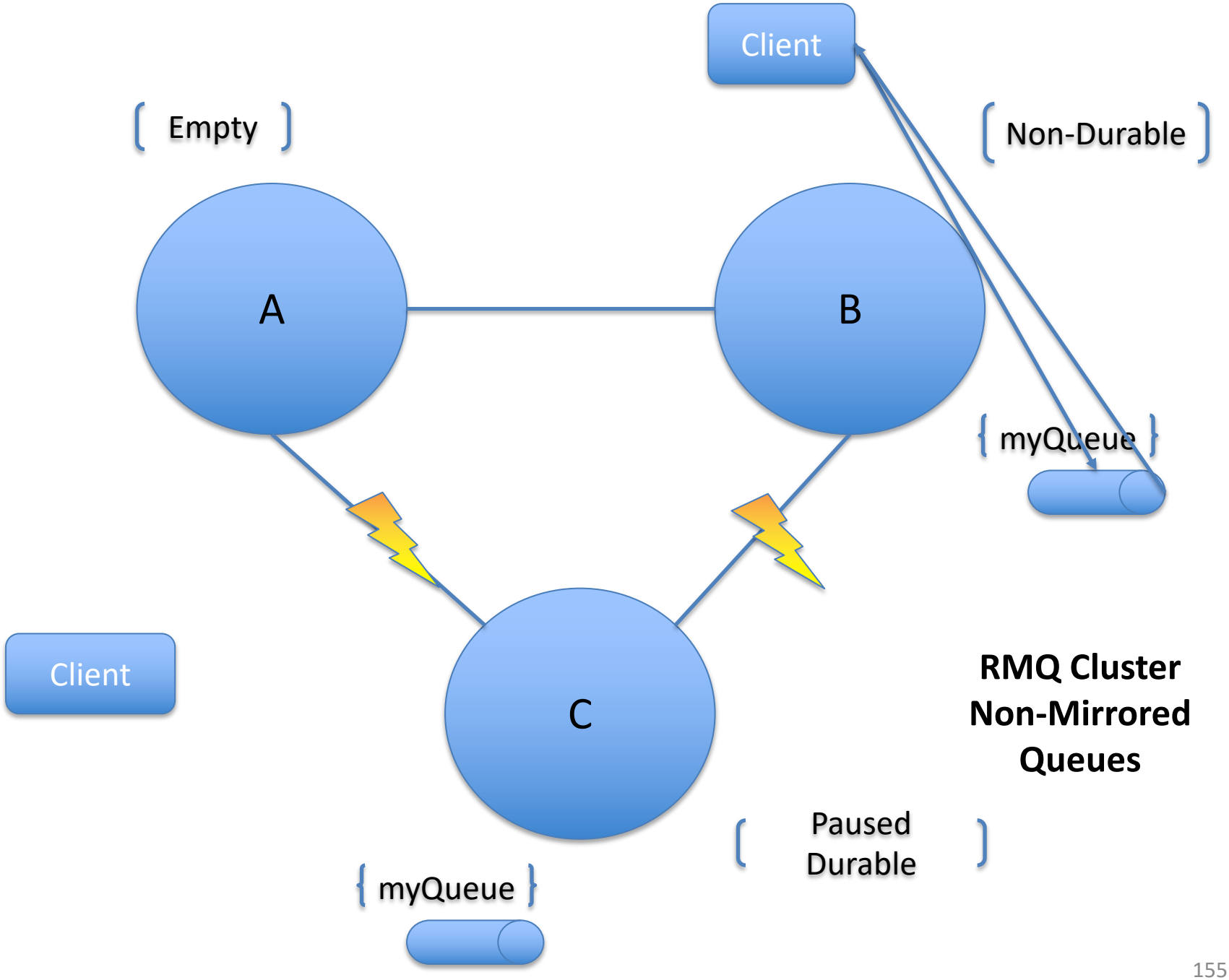

Client
Mirror
Master
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Mirror
myQueue

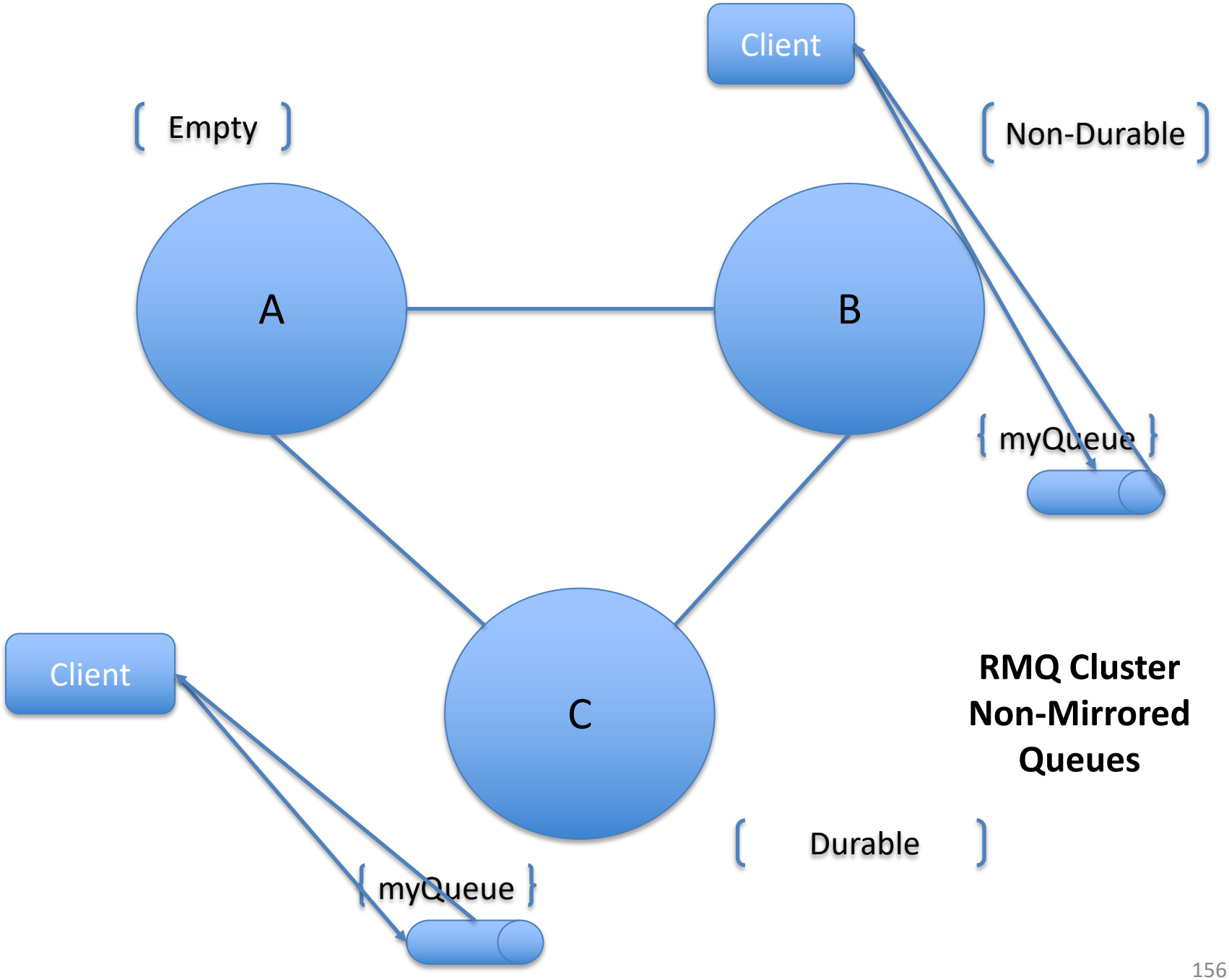

Client
Empty
Non-Durable
A
B
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Durable
myQueue


Client
Empty
Paused
Non-Durable
A
B
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Durable
myQueue


Client
Non-Durable
Paused
Non-Durable
A
B
myQueue
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Durable
myQueue


Client
Empty
Non-Durable
A
B
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Paused
Durable
myQueue


Client
Empty
Non-Durable
A
B
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Paused
Durable
myQueue


Client
Empty
Non-Durable
A
B
myQueue
Client
C
RMQ Cluster
Non-Mirrored
Queues
Durable
myQueue

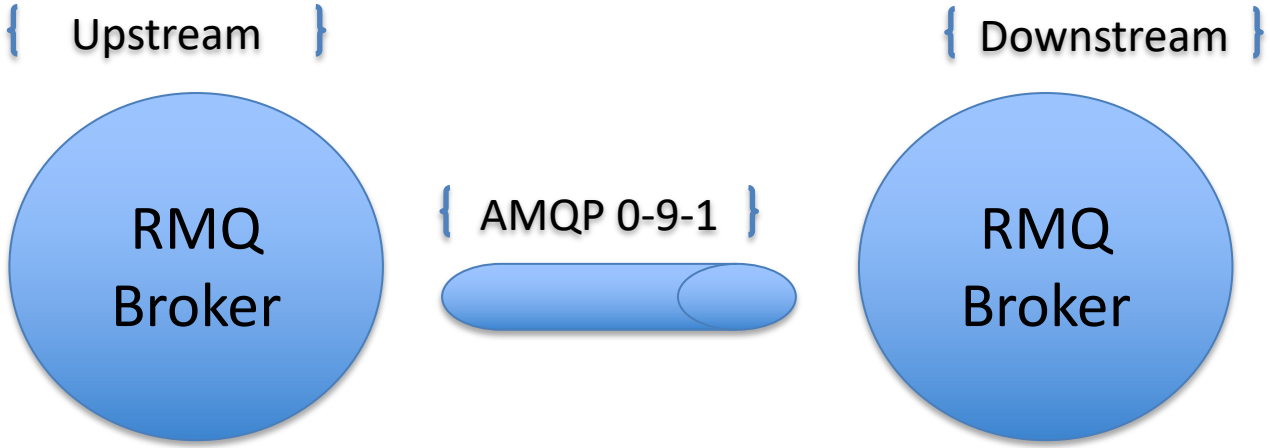
So which CAP options does an RMQ cluster support?

Pause Minority is Consistency (C) and Partition Tolerance (P)

In normal operation we sacrifice latency—time taken for a message to propagate to all nodes for consistency—all nodes will have a copy of the message in case of failure.

Non-durable non-mirrored queues are Availability (A) and Partition Tolerance (P)

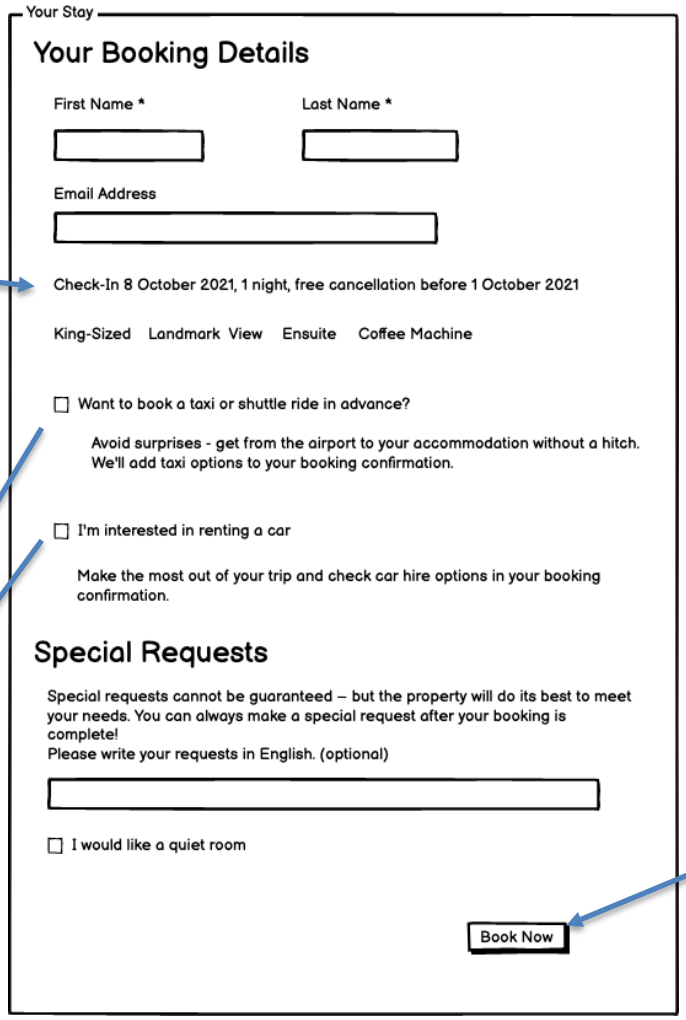
In normal operation we do sacrifice consistency—there are no copies—for improved latency as we do not have to copy the data to the same number of nodes.
But a message queue tends to always sacrifice L for A



RMQ Federation/Shovel

9. CONSUMERS

Task Based UI

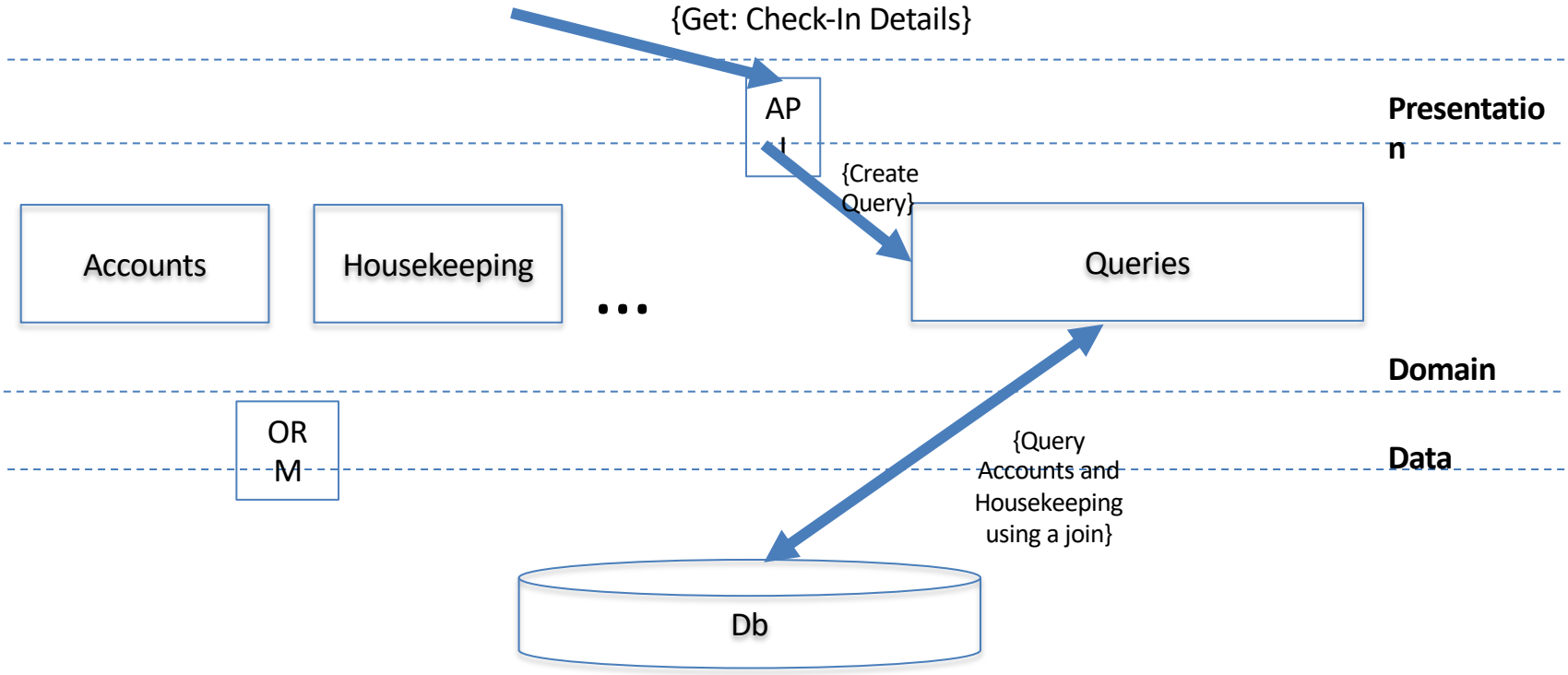


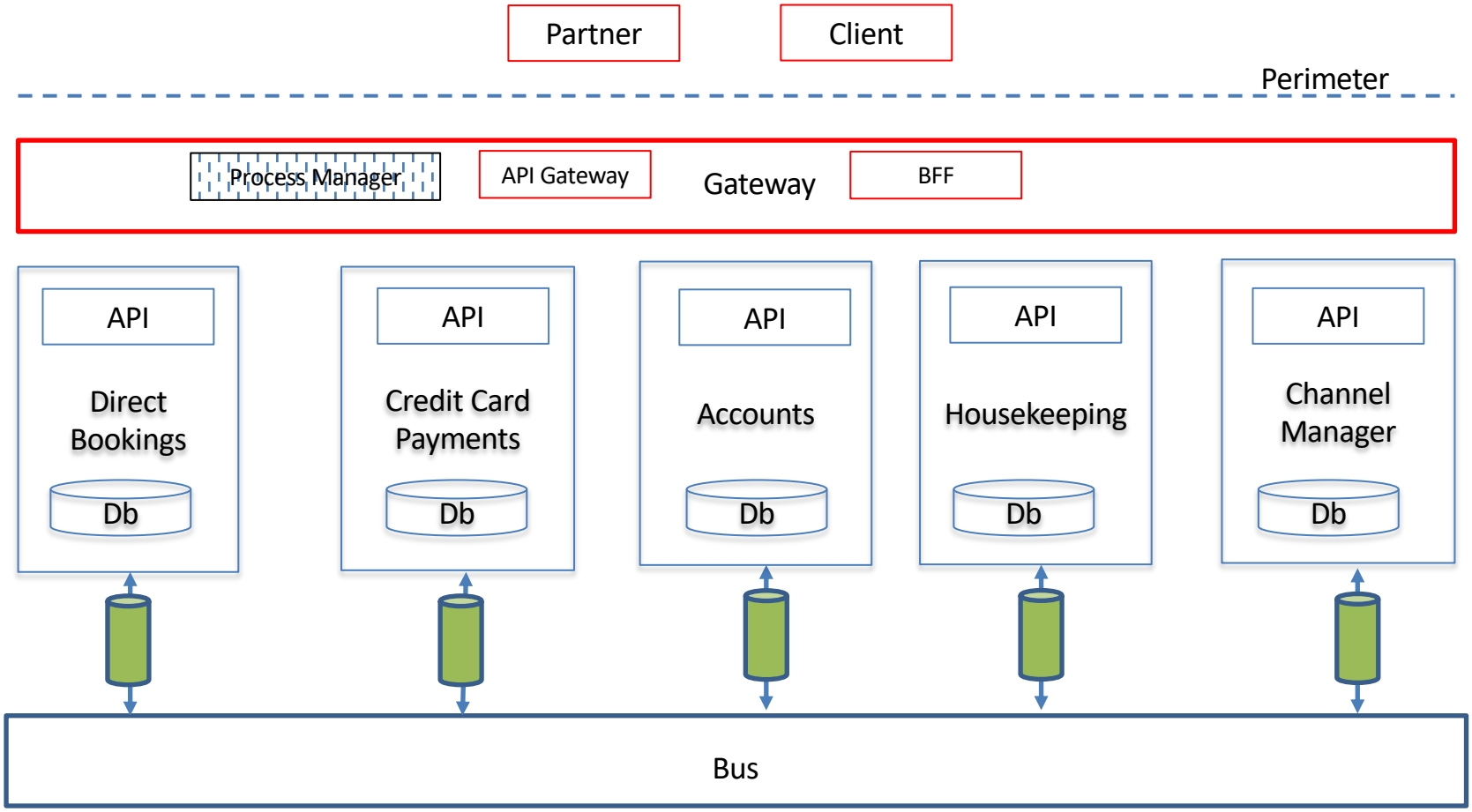
A sequence of screens can build up a request

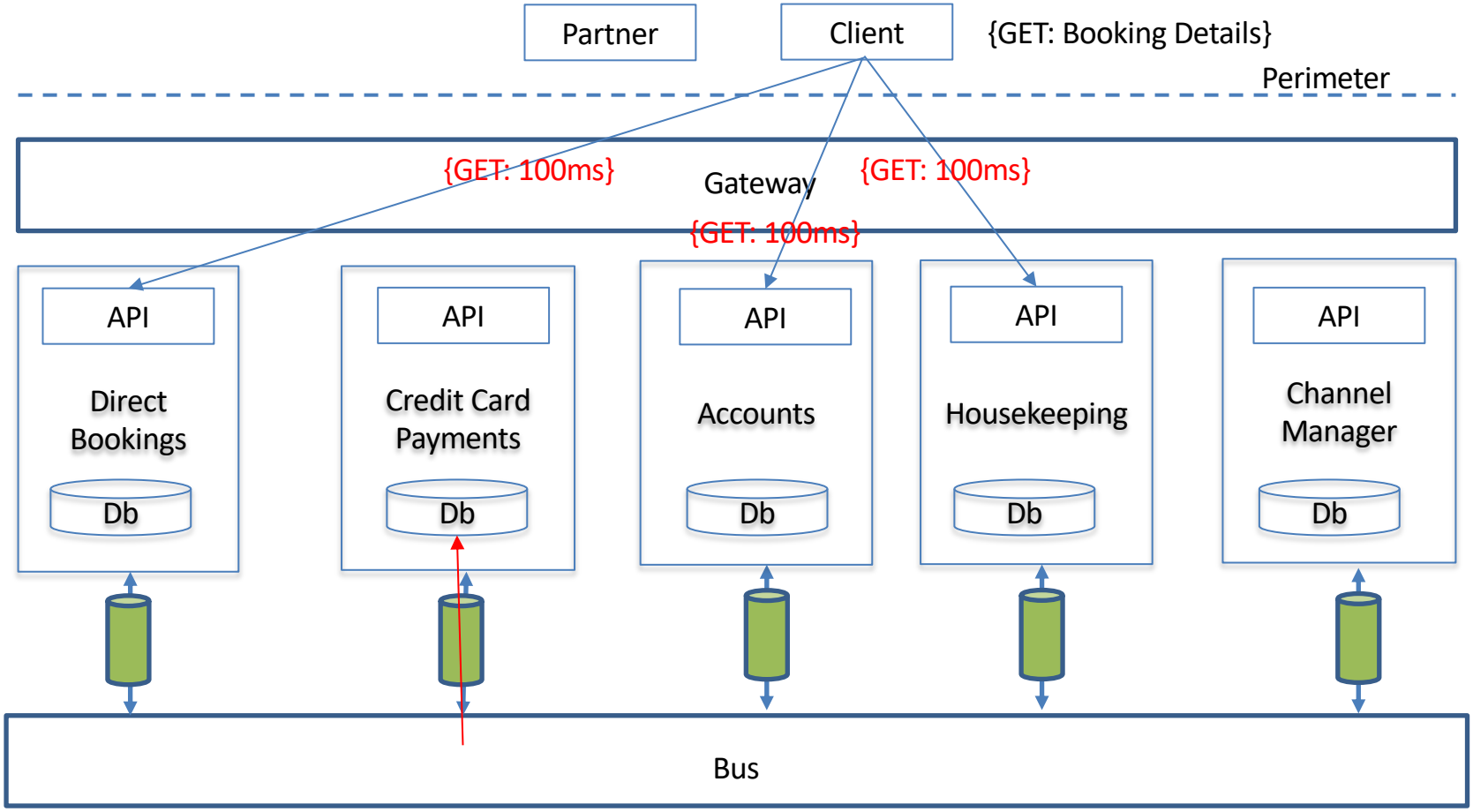
A sequence of screens can build up a request

At this point, the user expects async

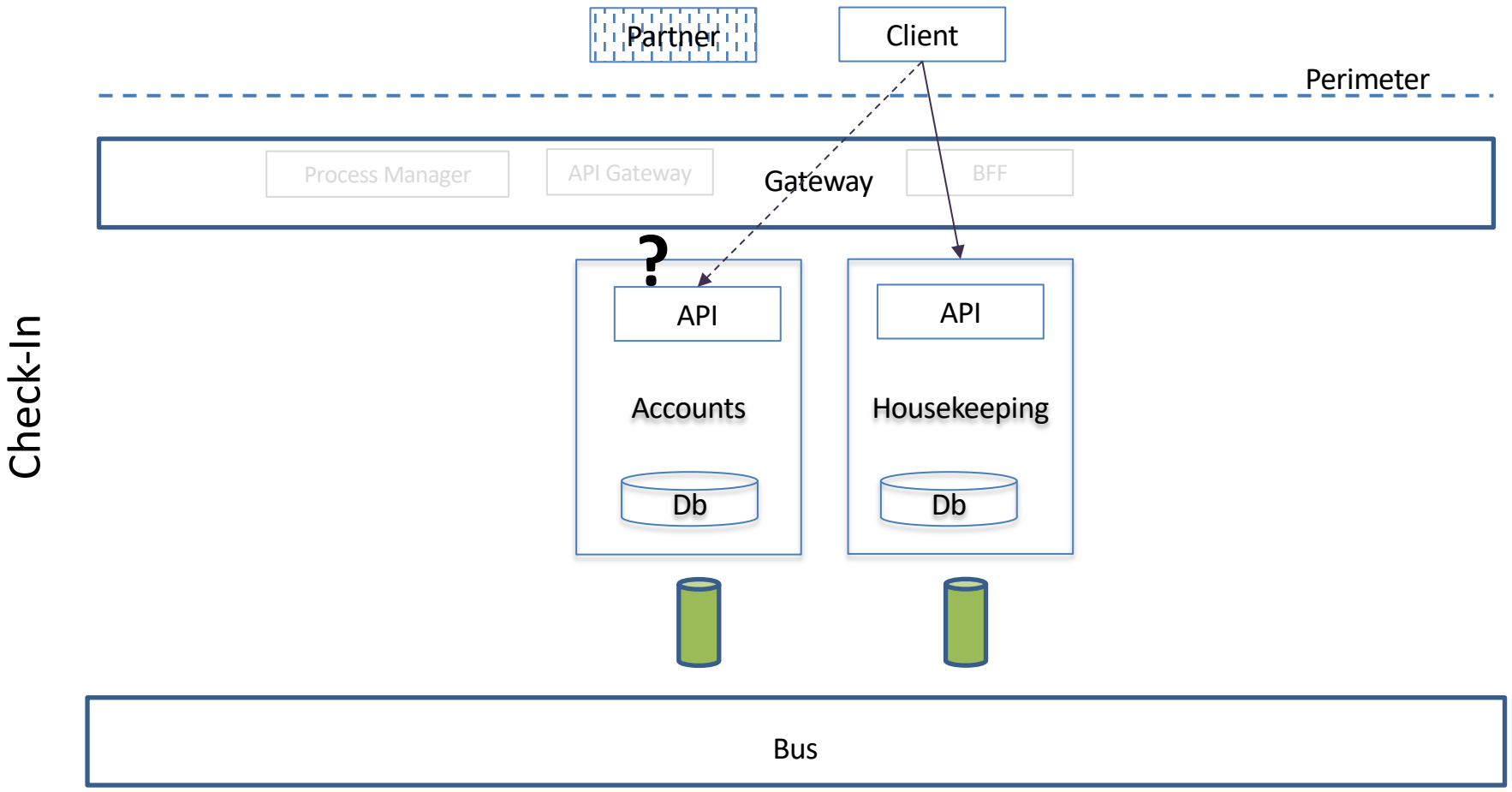
Getting work done in a Monolith




Partner
Client
Perimeter
Process Manager
API Gateway
Gateway
BFF
API
Direct Bookings
Db
API
Credit Card Payments
Db
API
Accounts
Db
API
Housekeeping
Db
API
Channel Manager
Db
Bus

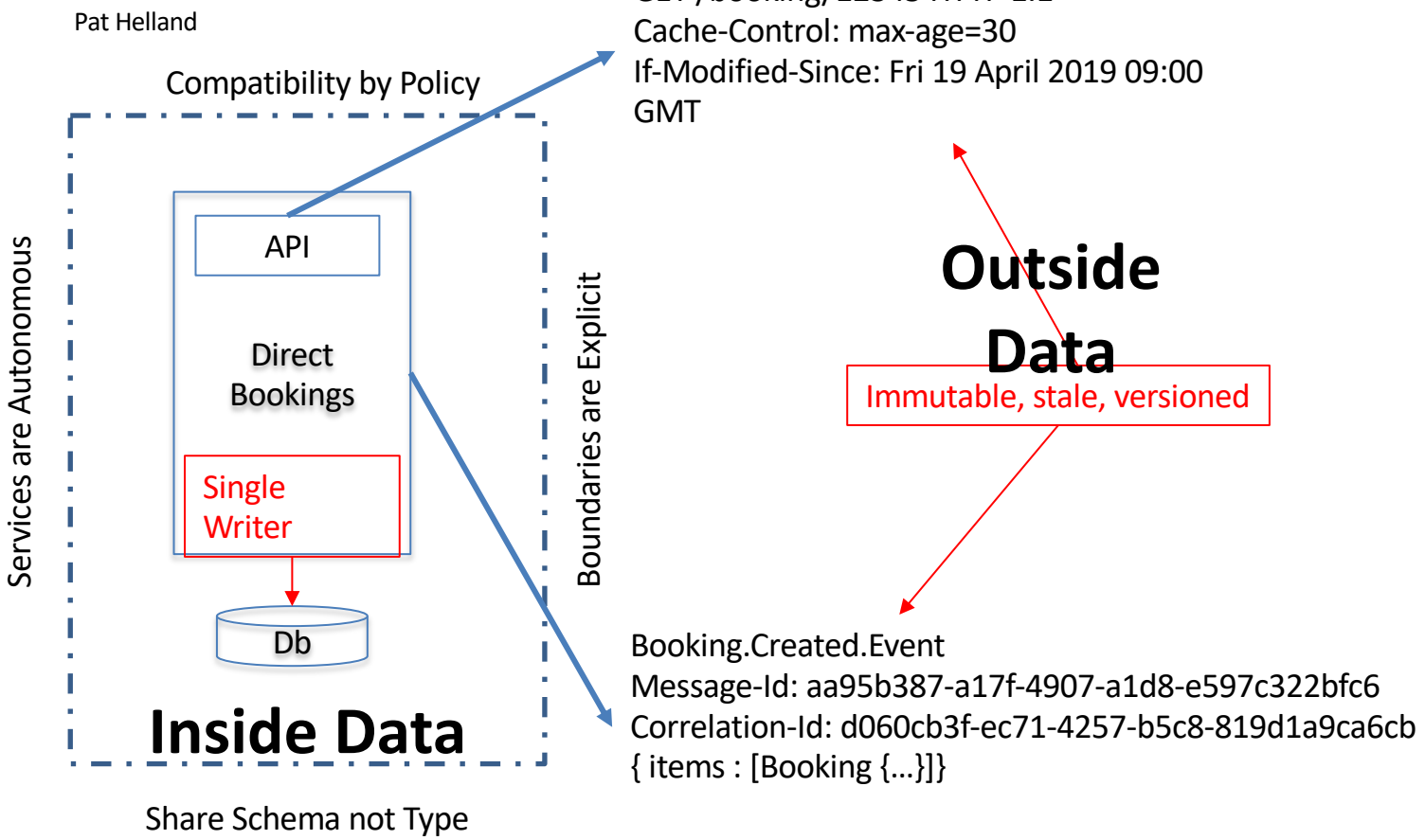

Partner
Client
{GET: Booking Details}
Perimeter
{GET: 100ms}
Gateway
{GET: 100ms}
{GET: 100ms}
API
Direct Bookings
Db
API
Credit Card Payments
Db
API
Accounts
Db
API
Housekeeping
Db
API
Channel Manager
Db
Bus

Check-In


Check-In
Partner
Client
Perimeter
Process Manager
API Gateway
Gateway
BFF
?
API
Accounts
Db
API
Housekeeping
Db
Bus

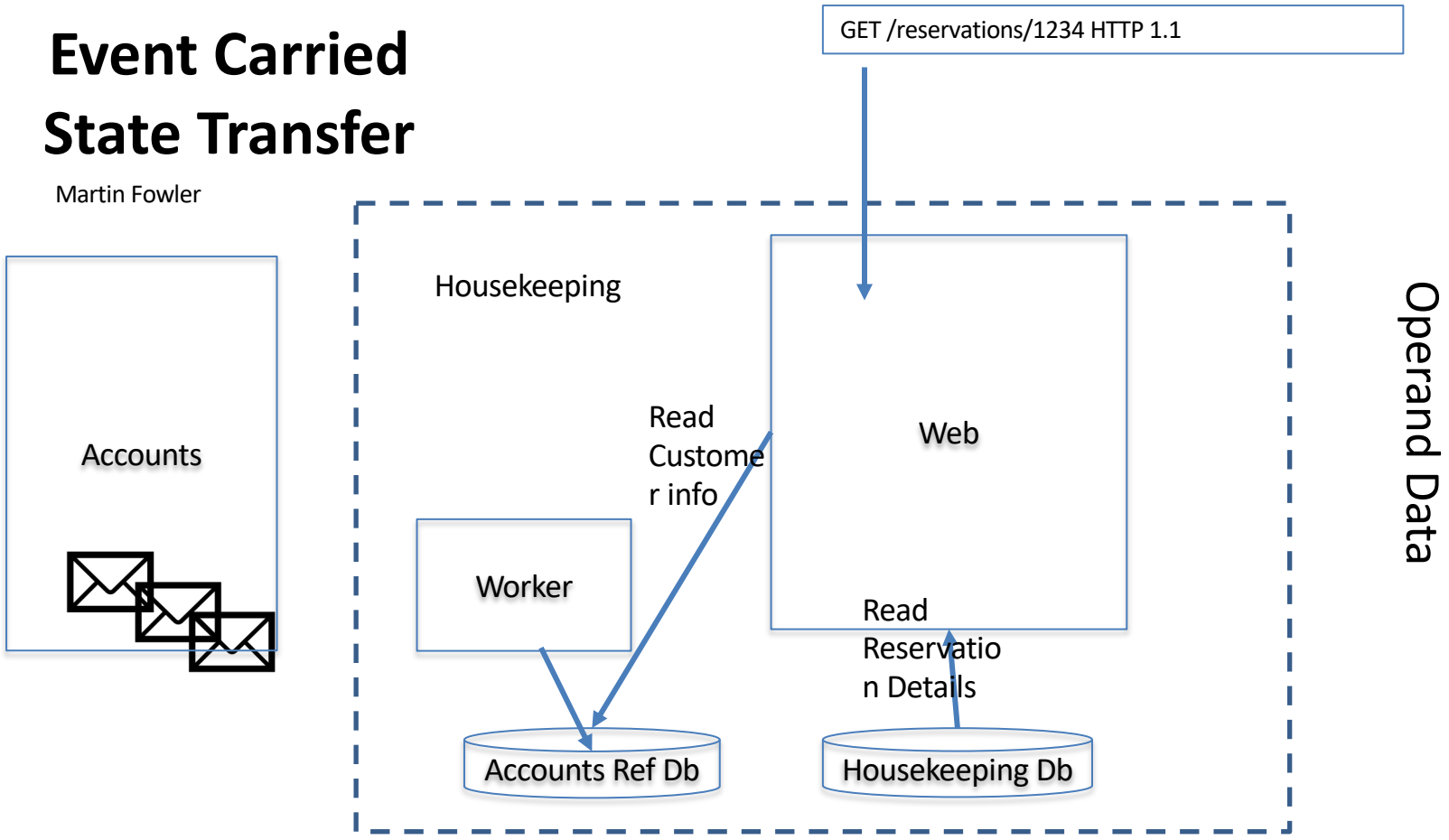
Reference Data

Pat Helland



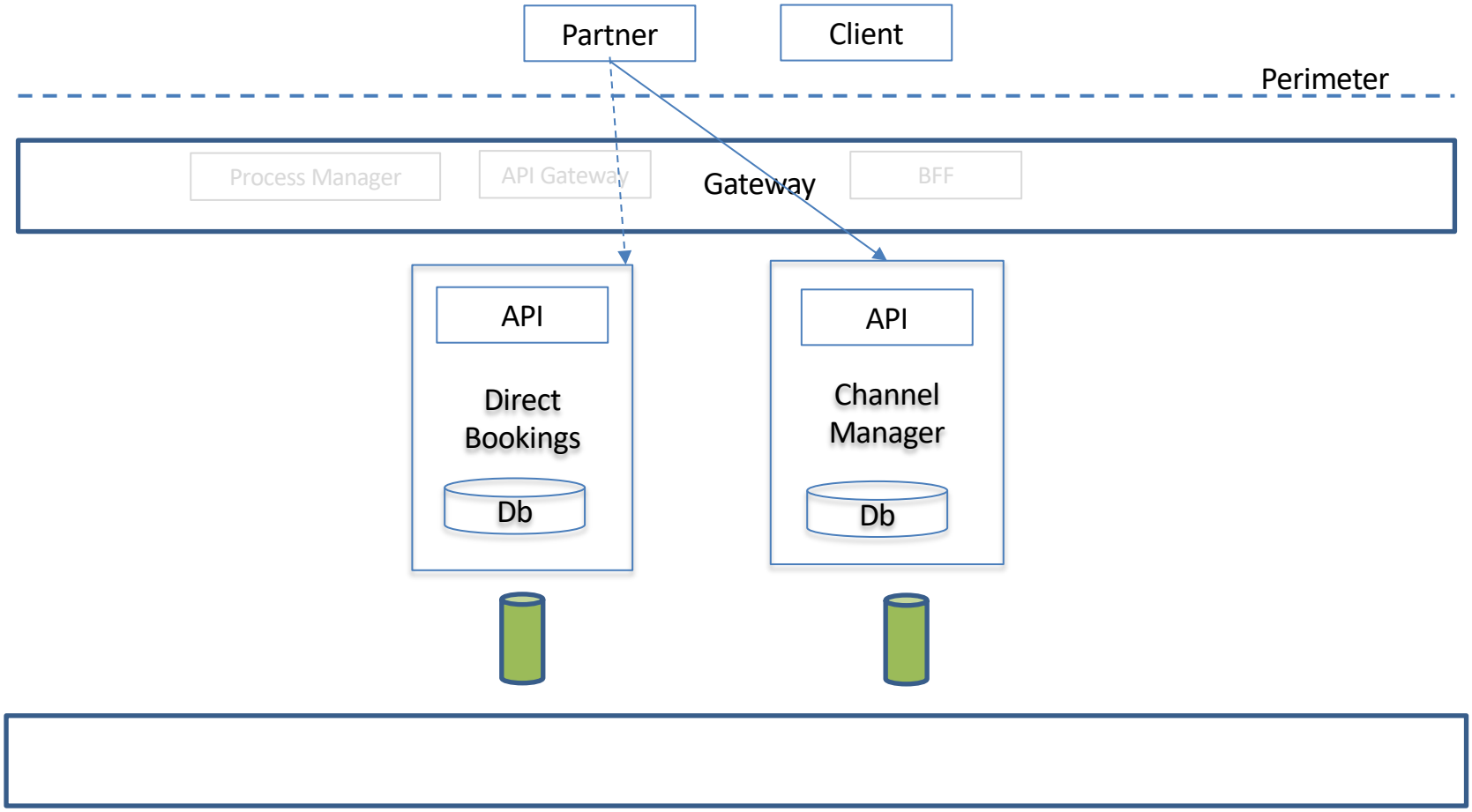
Event Carried State Transfer

Martin Fowler



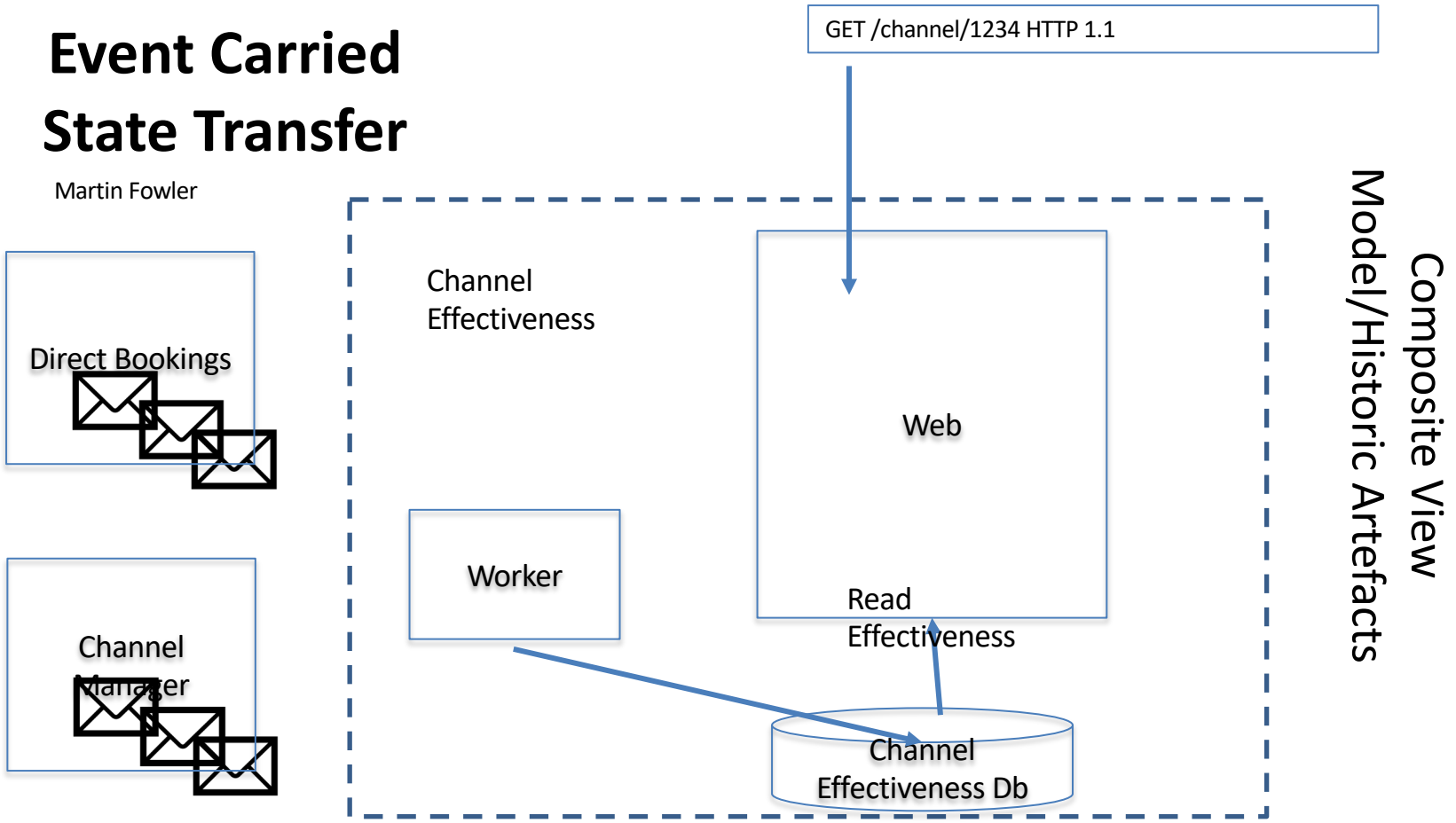
Channel Effectiveness

Channel Effectiveness



Event Carried State Transfer

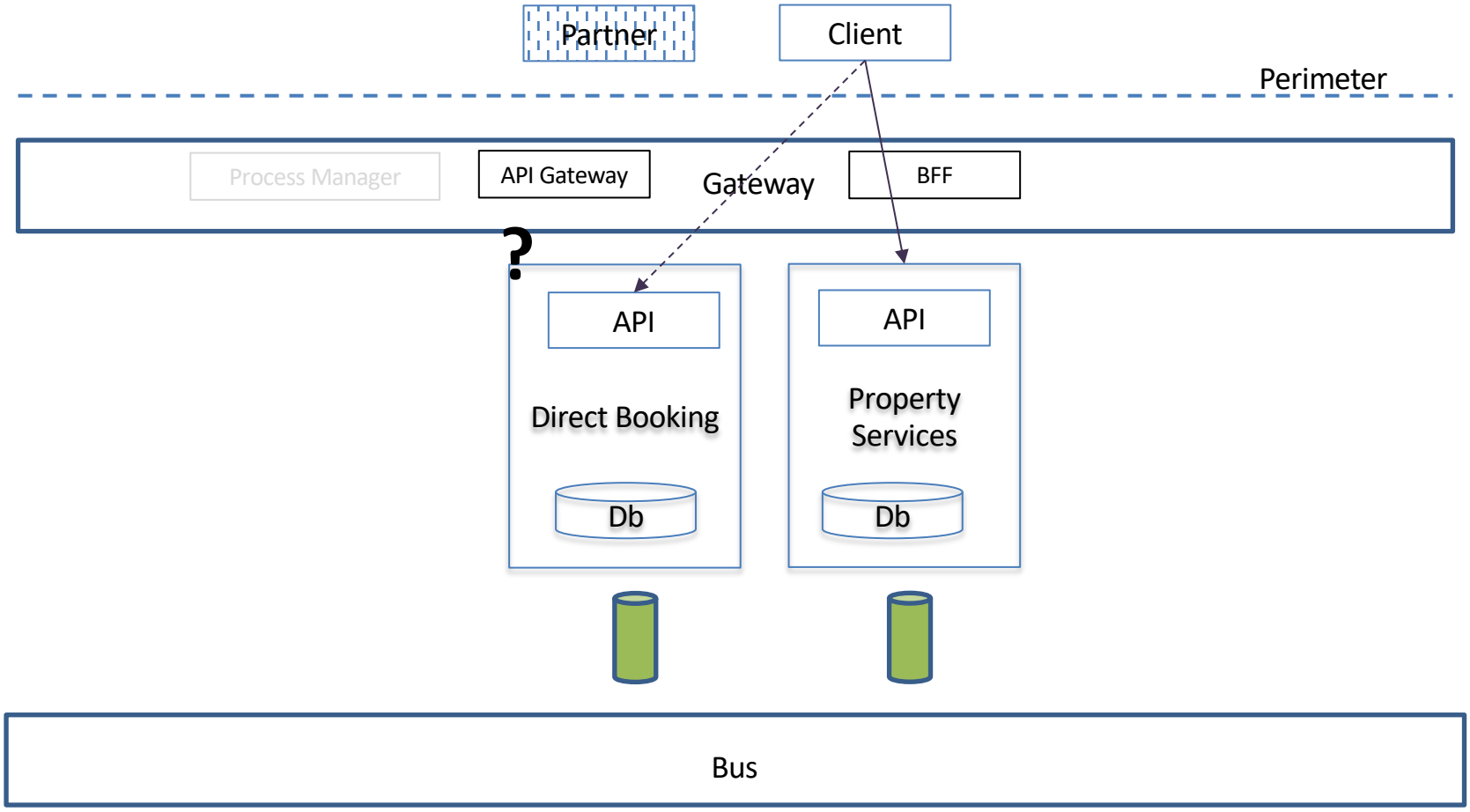
Martin Fowler



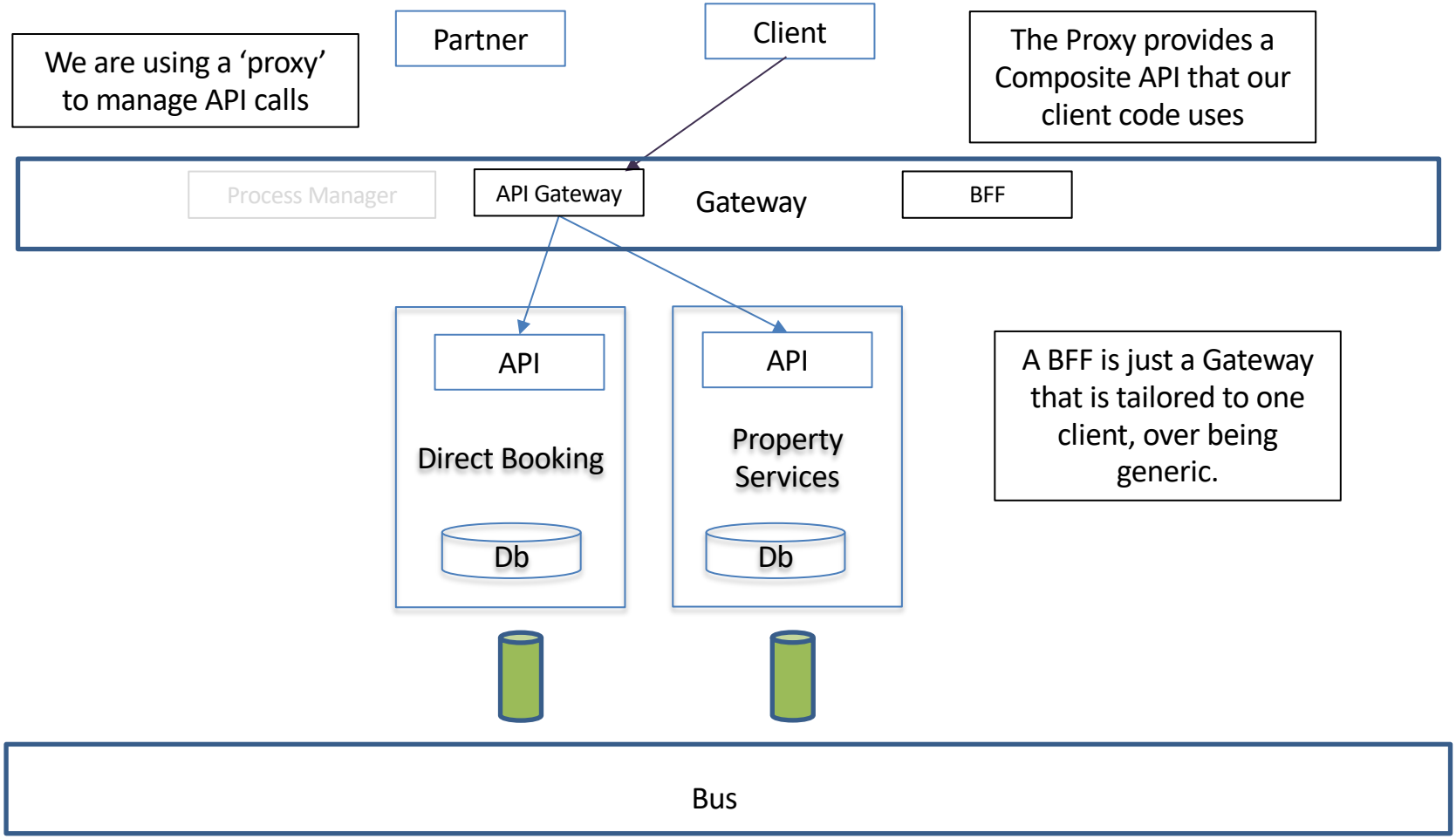
Composite View
Model/Historic Artefacts

Hotel Group Search

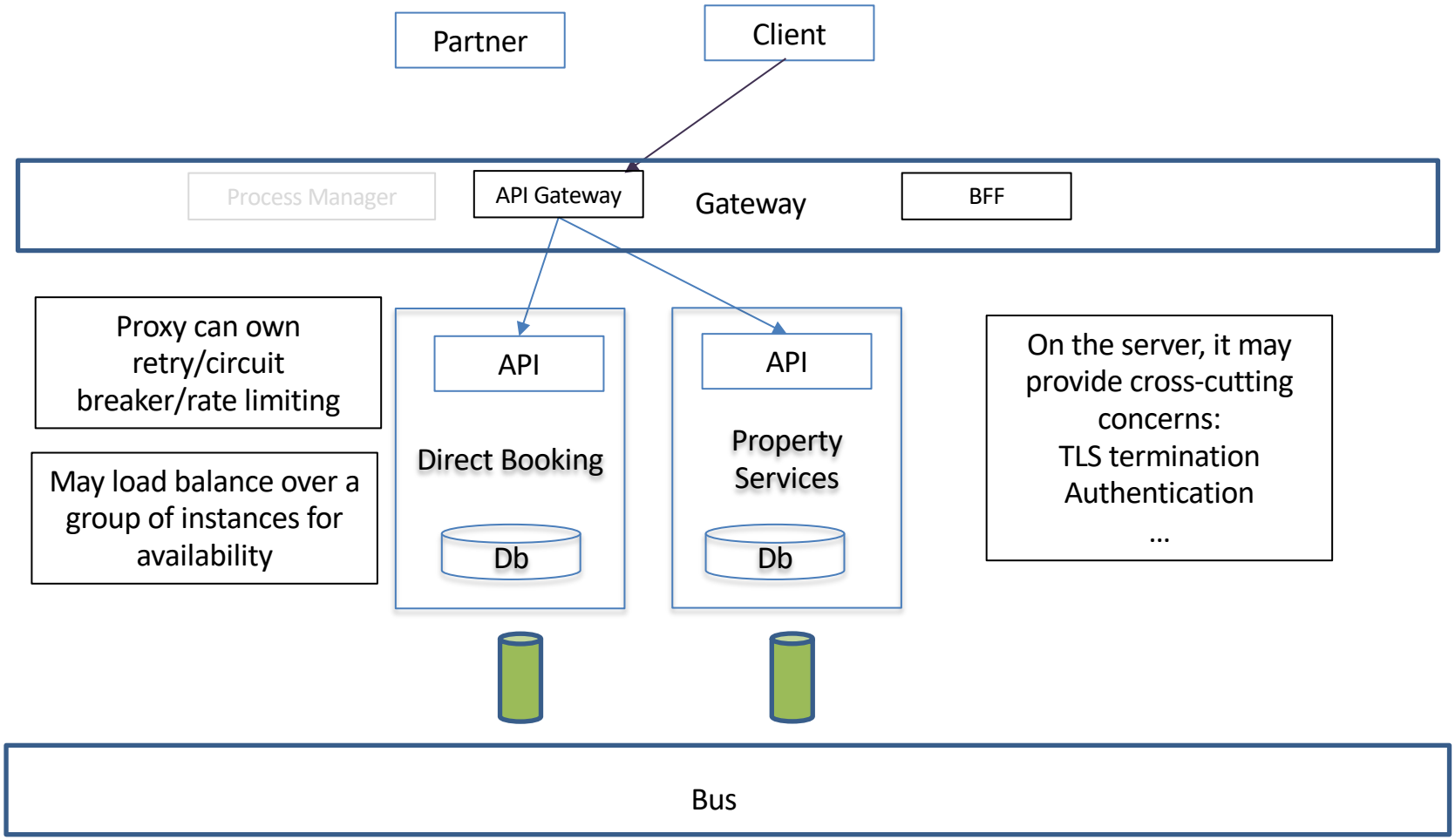
Group Room Search



Composition

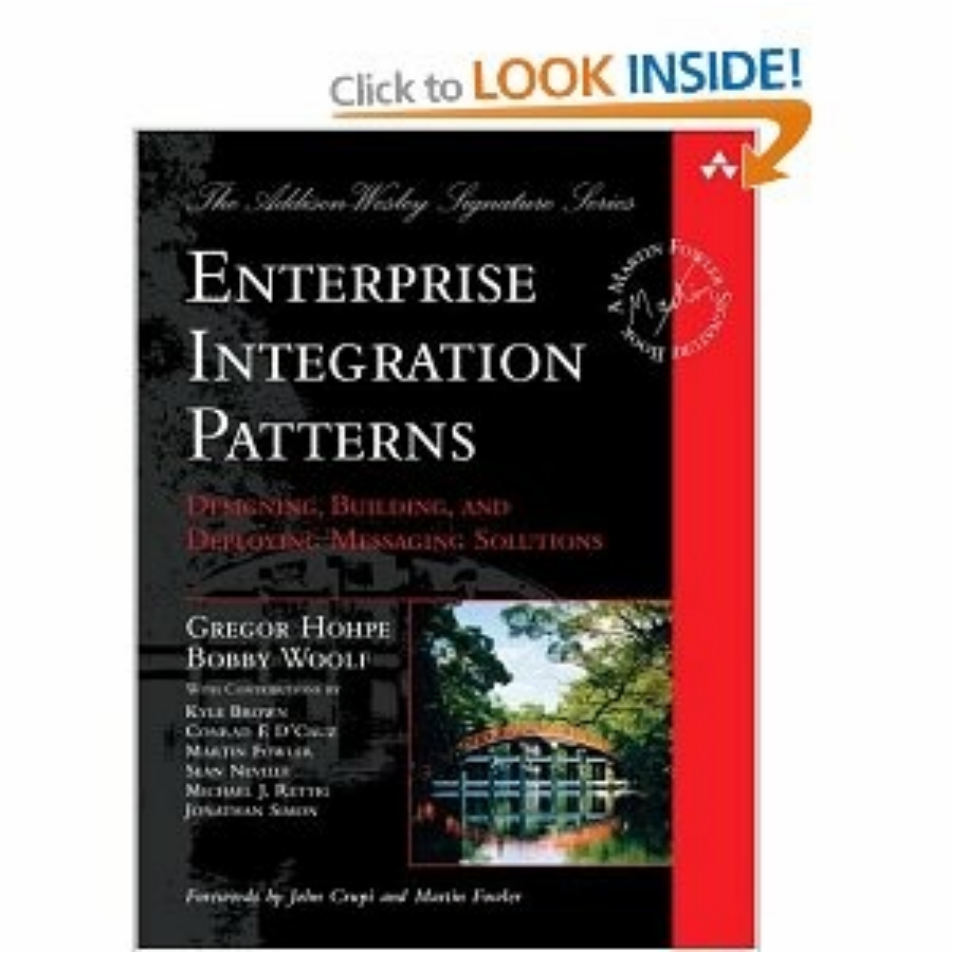


Composition



10. NEXT STEPS

Further Reading



Q&A