

Practical Messaging

A 101 guide to messaging

Ian Cooper

X and Hachyderm: ICooper

Who are you?

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

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

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

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

[www.linkedin.com/in/ian-cooper-
2b059b](https://www.linkedin.com/in/ian-cooper-2b059b)

Day One Messaging

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

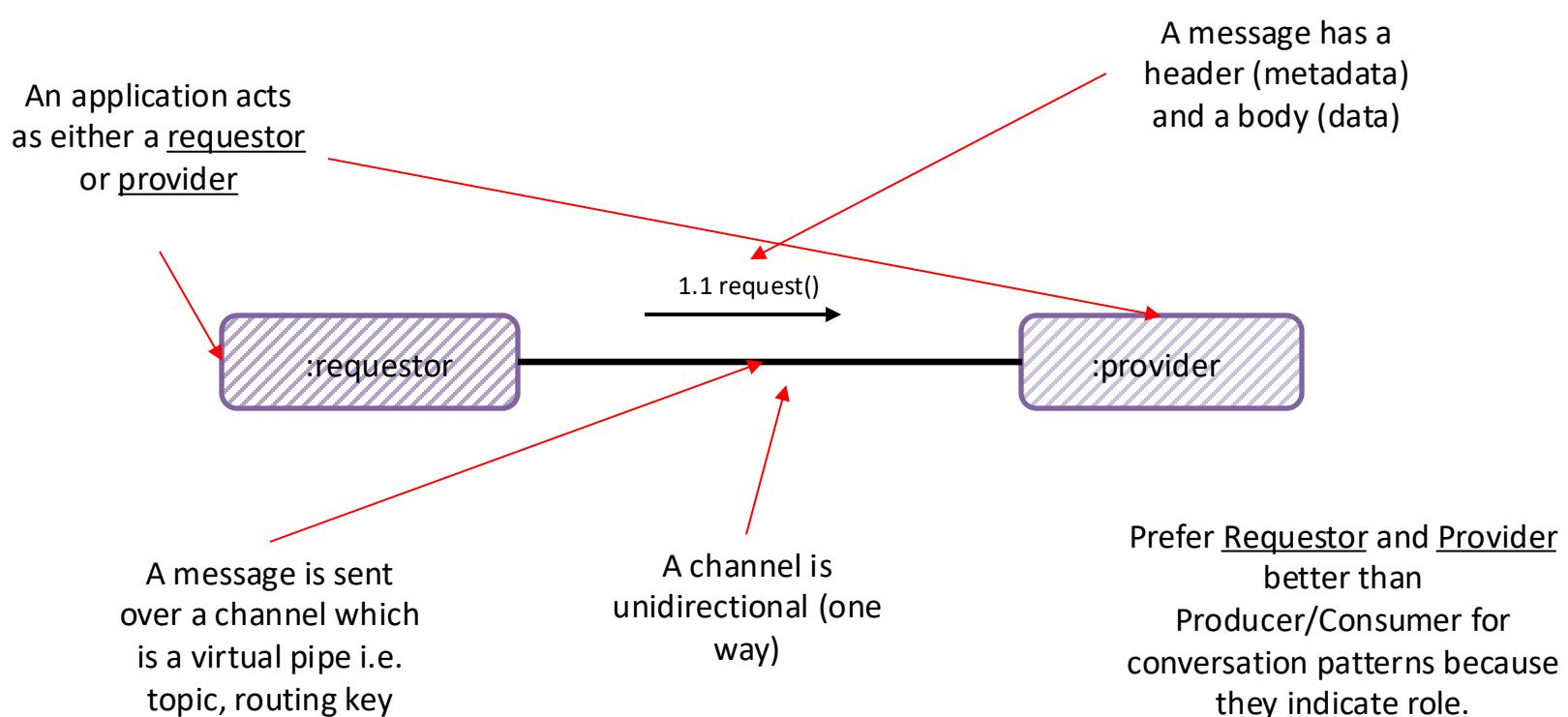
Day Two Conversations

- Conversation Patterns
 - Activity and Correlation
 - Repair and Clarification
 - Reliable Messaging
 - Fat and Skinny
 - Conversations
- Reactive Architectures
 - Message Passing
 - Paper Based Flows
 - Flow Based Programming
- Next Steps

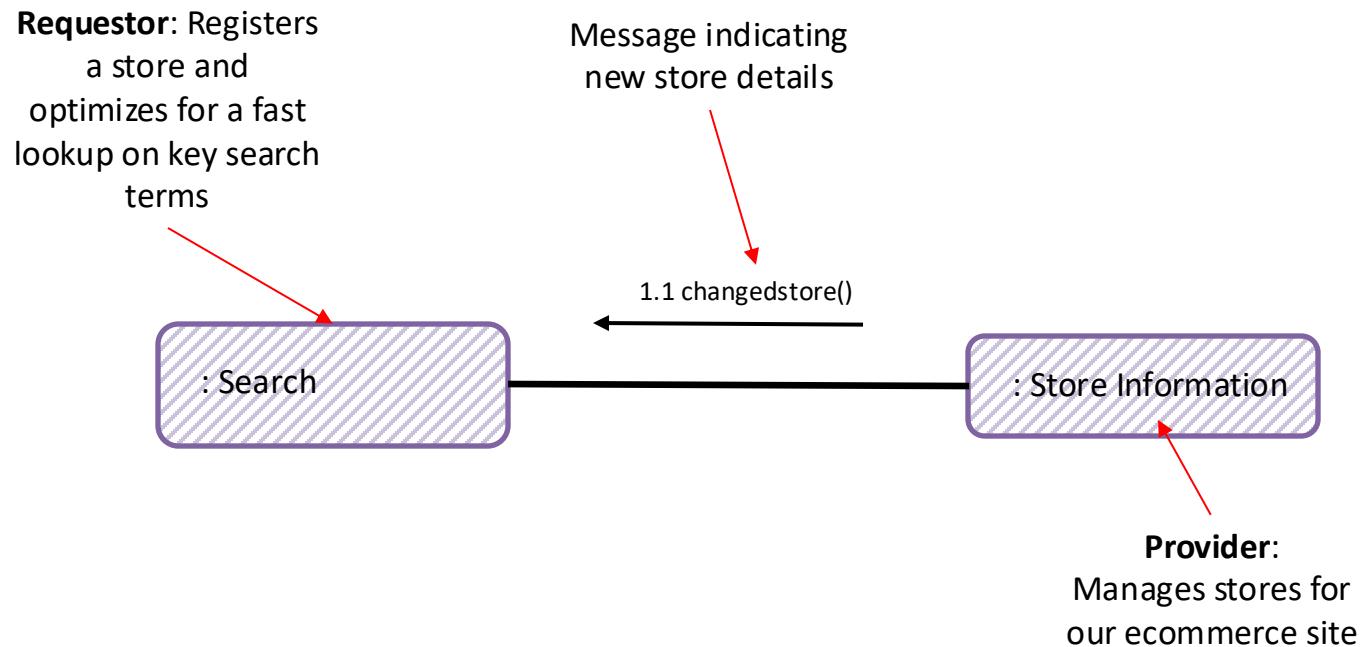
Day Two

CONVERSATION PATTERNS

Messaging Participants



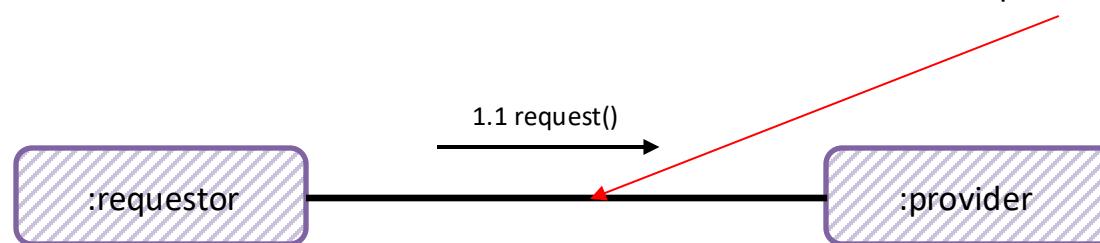
Messaging Participants



In-Only (Fire and Forget)

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

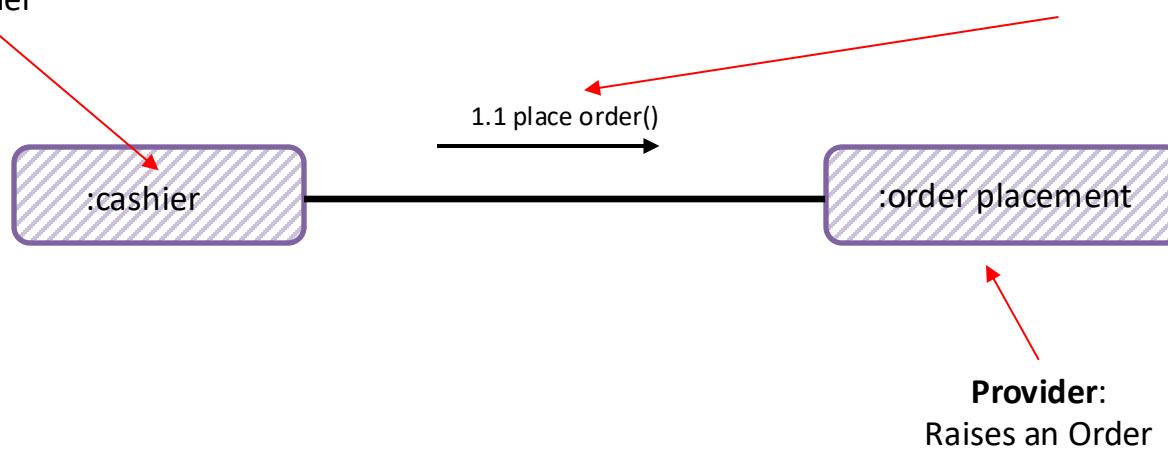
Under a In-Only pattern the requestor sends a request to the provider, but does not seek an acknowledgment of completion of the requested operation



In-Only (Fire and Forget)

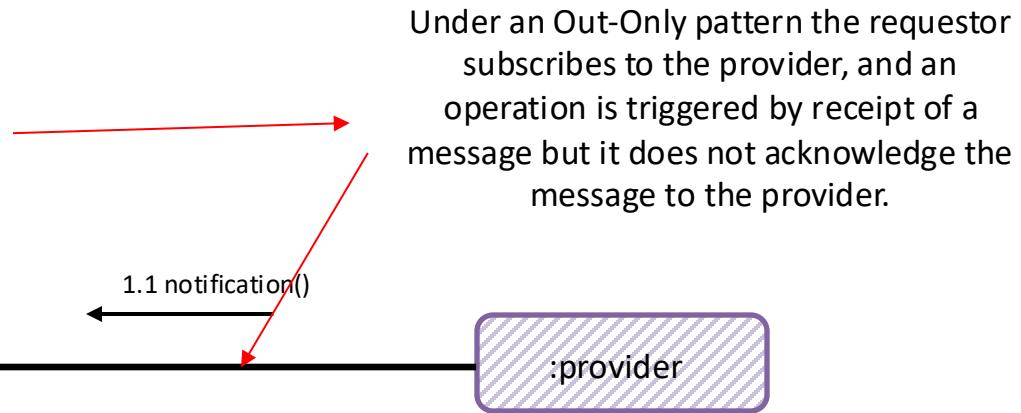
Requestor: Has a paid for basket it wants to turn into an order

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

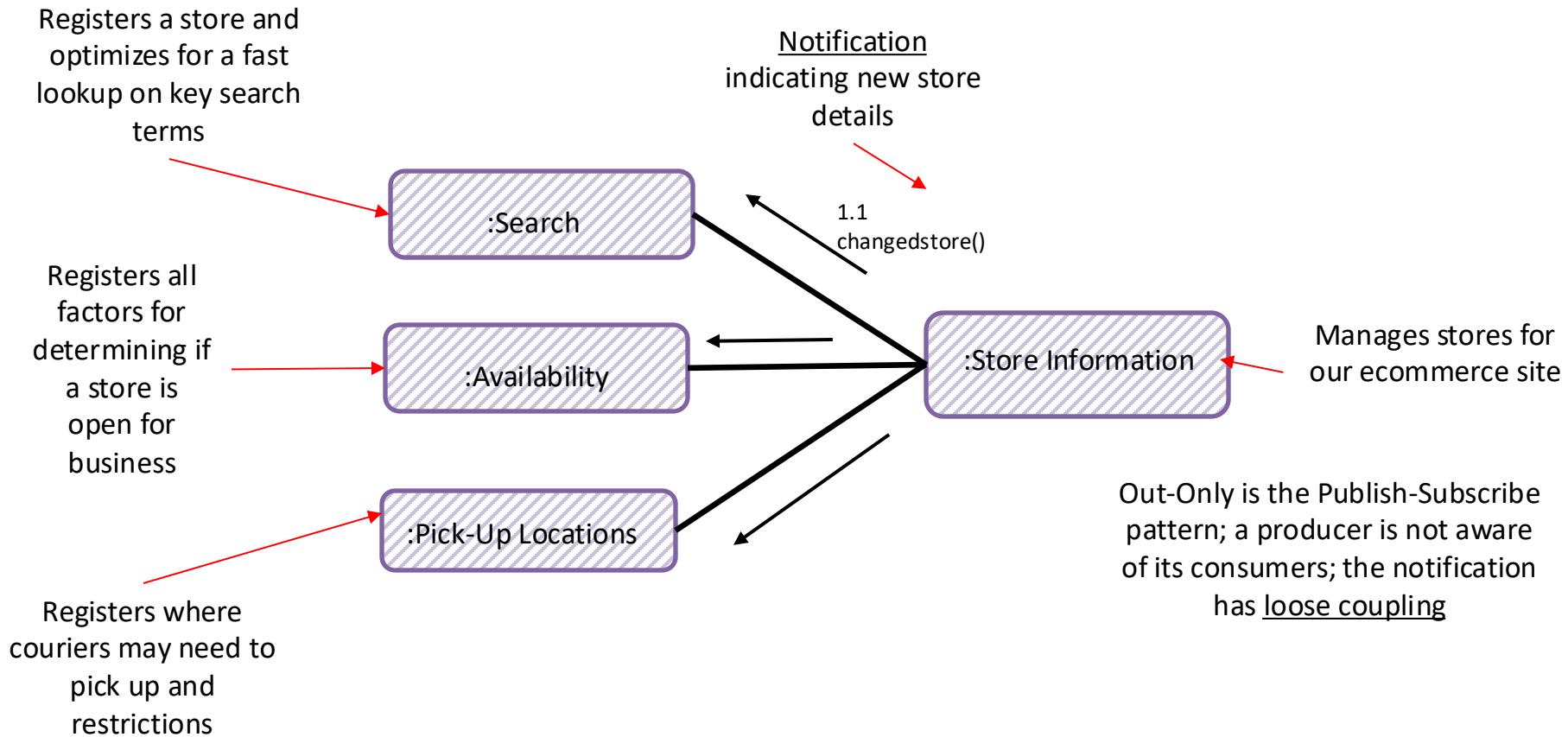


Out-Only (Notification)

Typically we call this a **notification**.

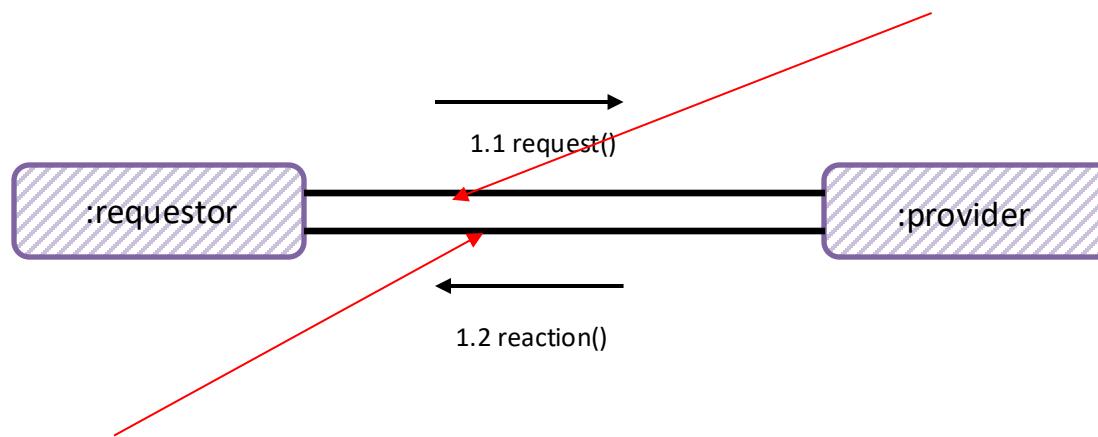


Out-Only (Publish-Subscribe)



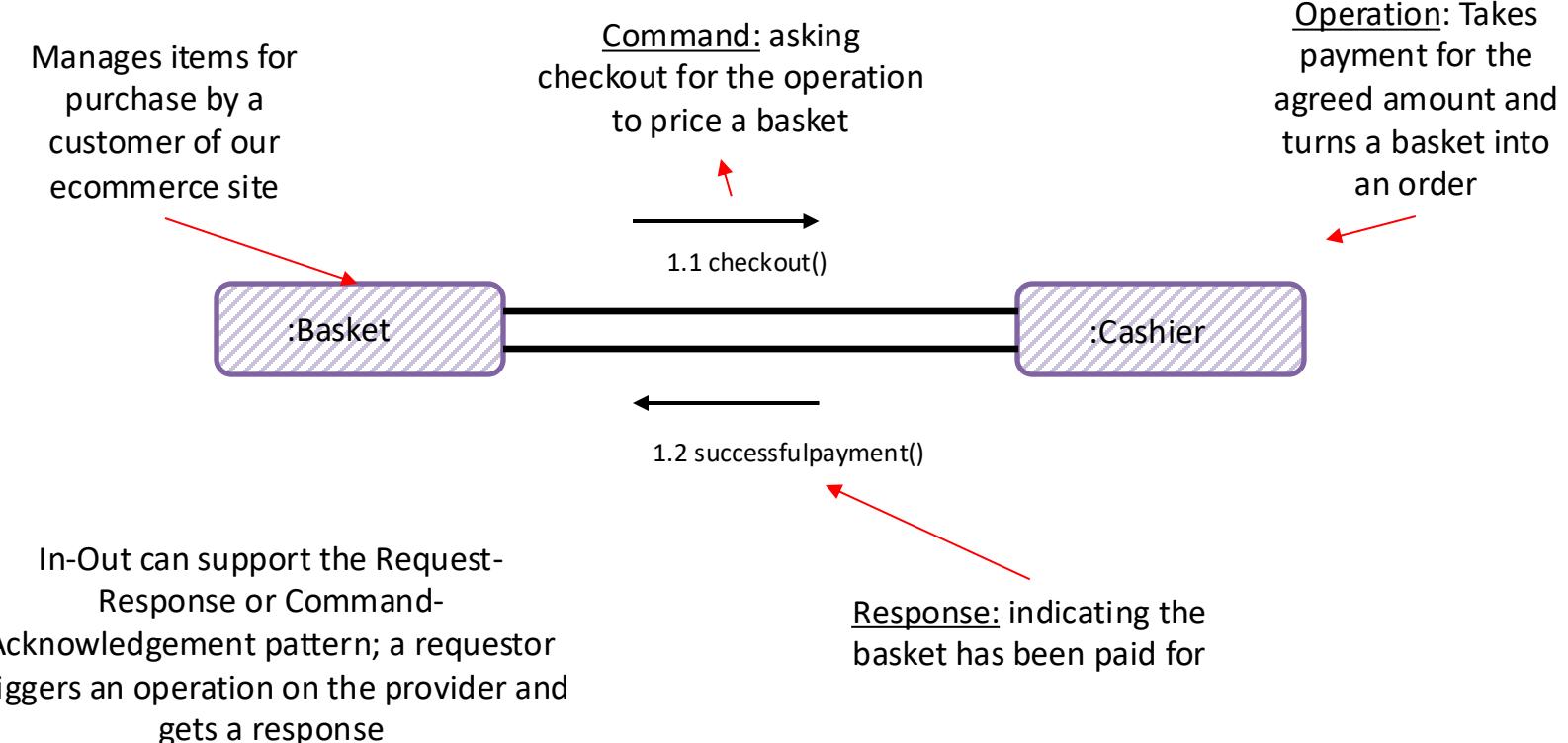
In-Out (Request-Reaction)

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



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

Request-Reaction (Command-Acknowledgement)



In-Out (Query-Result)

Manages items for purchase by a customer of our ecommerce site



Query asking for delivery fees for this basket

1.1 getdeliveryfees()



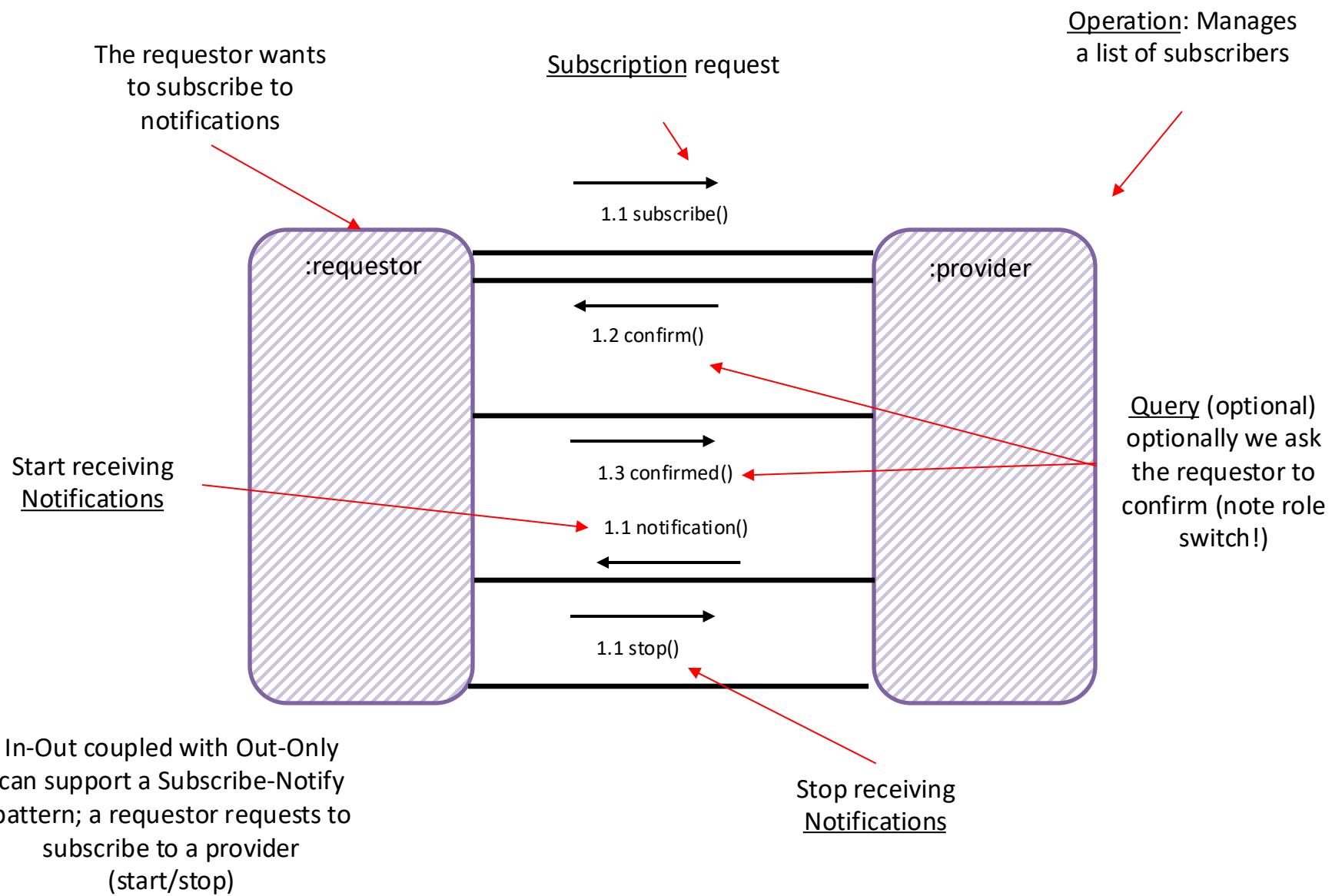
Operation:
Determines the delivery fee based on store and items in the basket

1.2 deliverfee()

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

Result indicating the fee for the current state of the basket

In-Out (Subscribe-Notify)

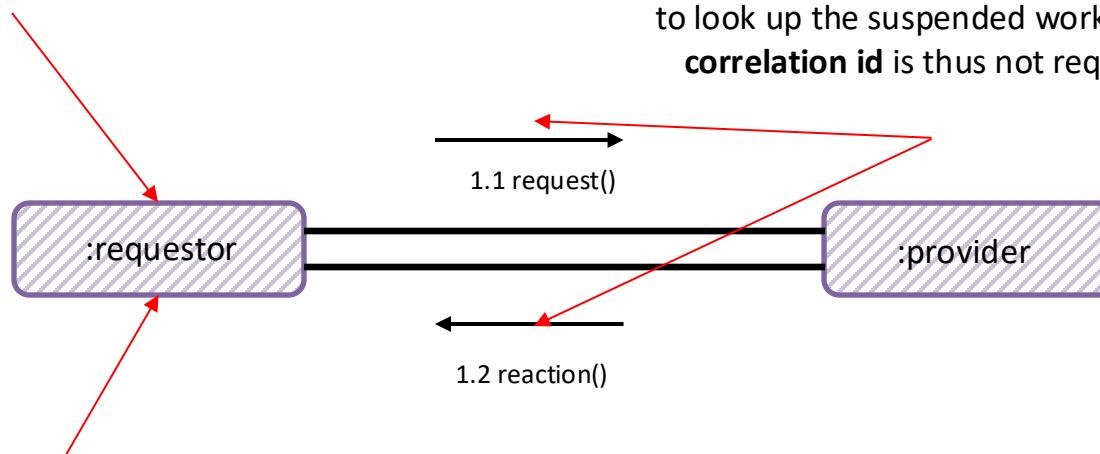


In-Out coupled with Out-Only can support a Subscribe-Notify pattern; a requestor requests to subscribe to a provider (start/stop)

Blocking In-Out (Request-Reply)

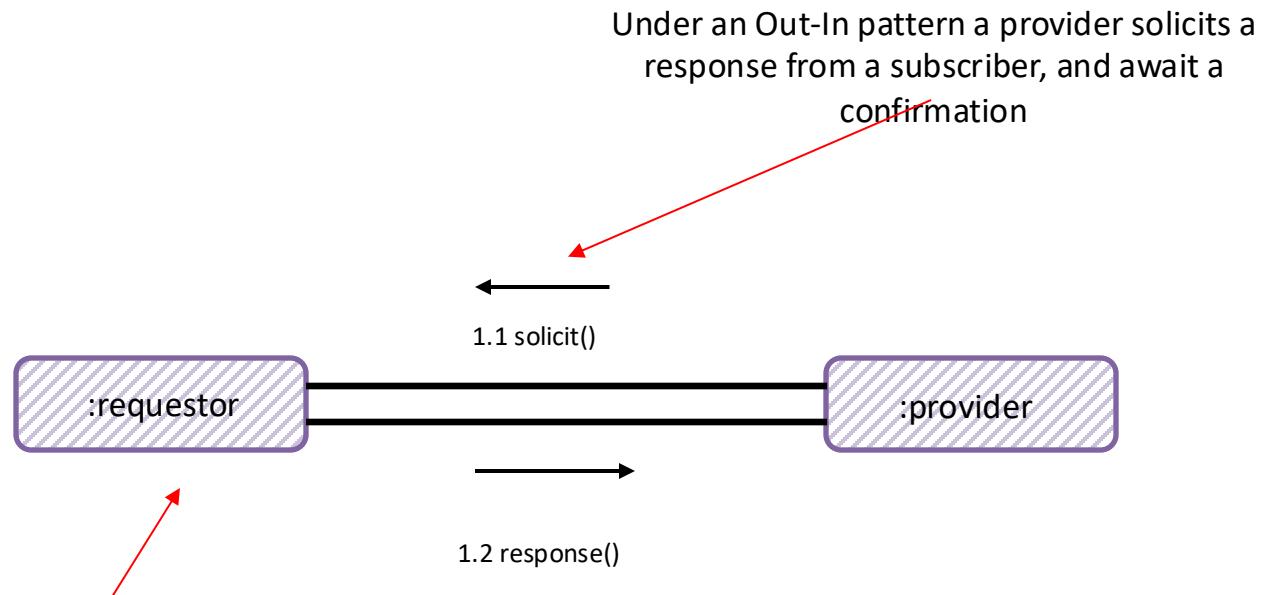
In order to resume, we need a way to identify the suspended workflow.

The requestor provides a unique **Reply To** channel (in the request message header/metadata) and the provider uses that channel for the response, allowing the sender to look up the suspended workflow. A **correlation id** is thus not required



The requestor may block on the Reply To channel whilst awaiting the reply.

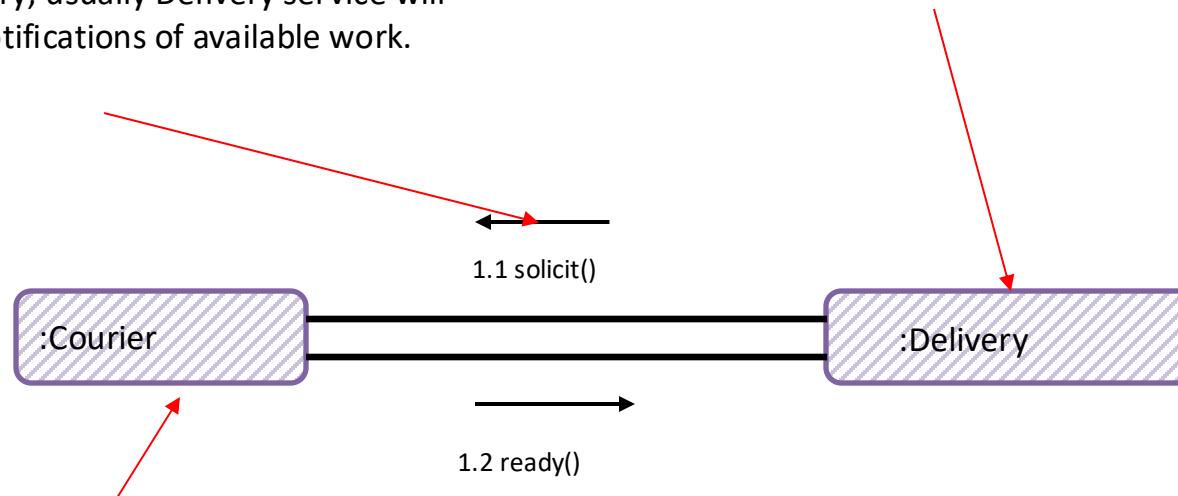
Out-In (Solicit-Response)



Out-In (Solicit-Response)

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

Delivery service assigns delivery request to drivers



In-Only (fire and forget)

Messaging

Has Intent

Request An Answer (Query)

Transfer of Control

(Command)

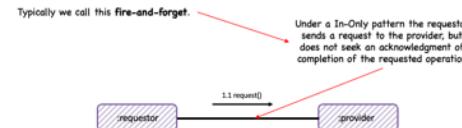
Transfer of Value

Part of a Workflow

Part of a Conversation

Concerned with the Future

In-Only (Fire and Forget)

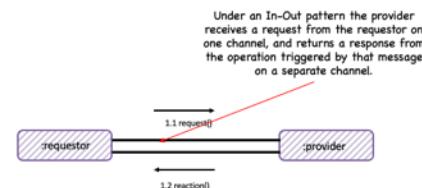


Ian Cooper

25

In-Out (request-reaction)

In-Out (Request-Reaction)



Ian Cooper

29

Eventing

Provides Facts

Things you Report On

No Expectations

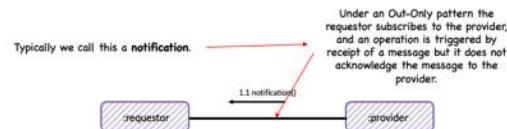
History

Context

Concerned with the Past

Out-Only (notification)

Out-Only (Notification)

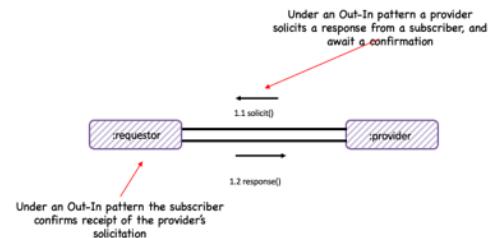


Ian Cooper

27

Out-In (solicit-response)

Out-In (Solicit-Response)



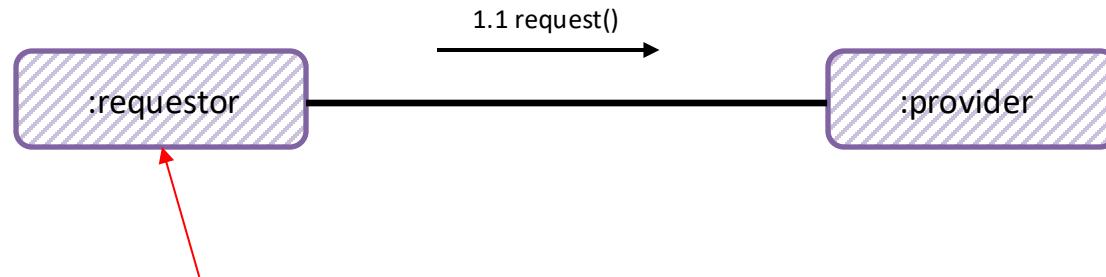
Ian Cooper

34

How do we manage workflows in requestors and providers

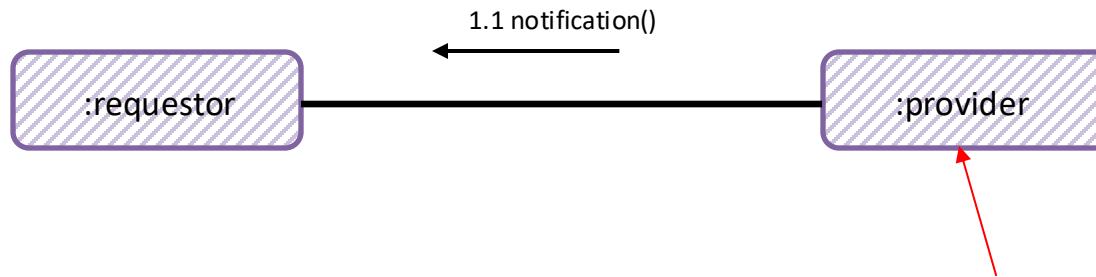
ACTIVITY AND RESOURCES

In-Only (Fire and Forget)



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

Out-Only (Notification)

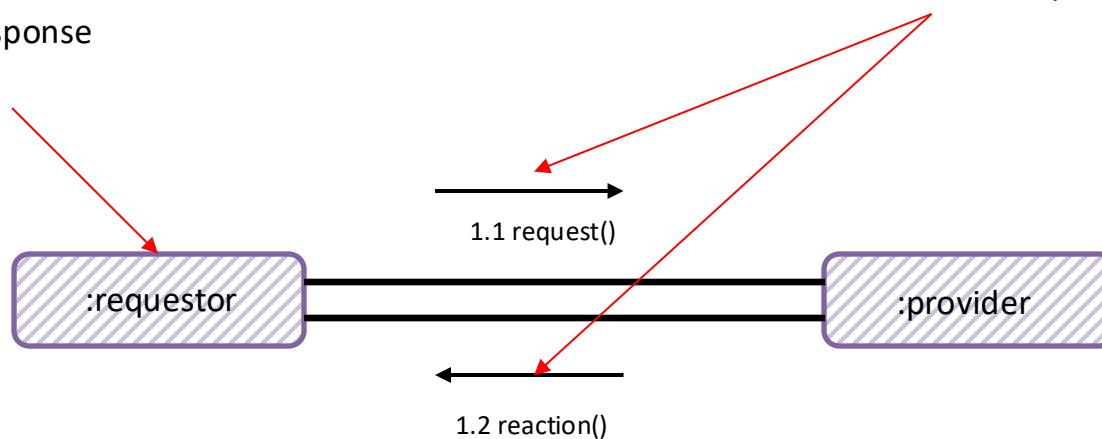


Because we only **notify** there is no need to maintain state to correlate with a response; we don't expect an acknowledgement

In-Out (Request-Reaction)

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

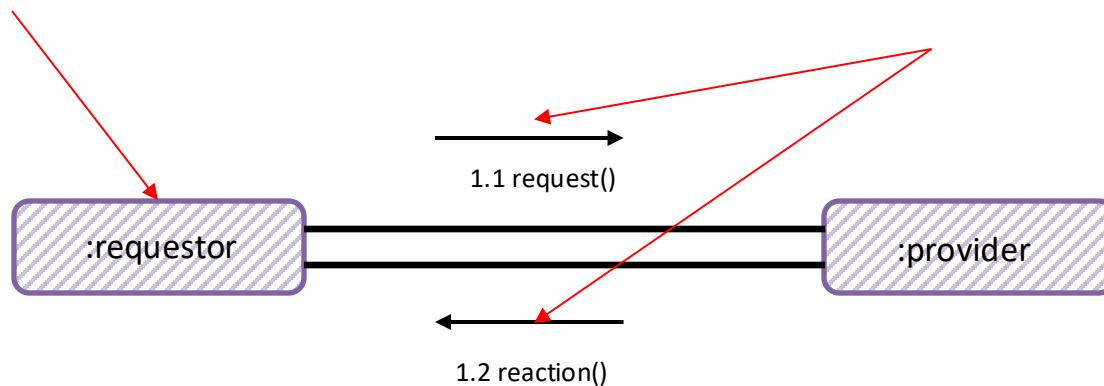
Because we expect a response to our request
we may need to save state which we then correlate with the request



In-Out (Request-Reaction) Correlation Id

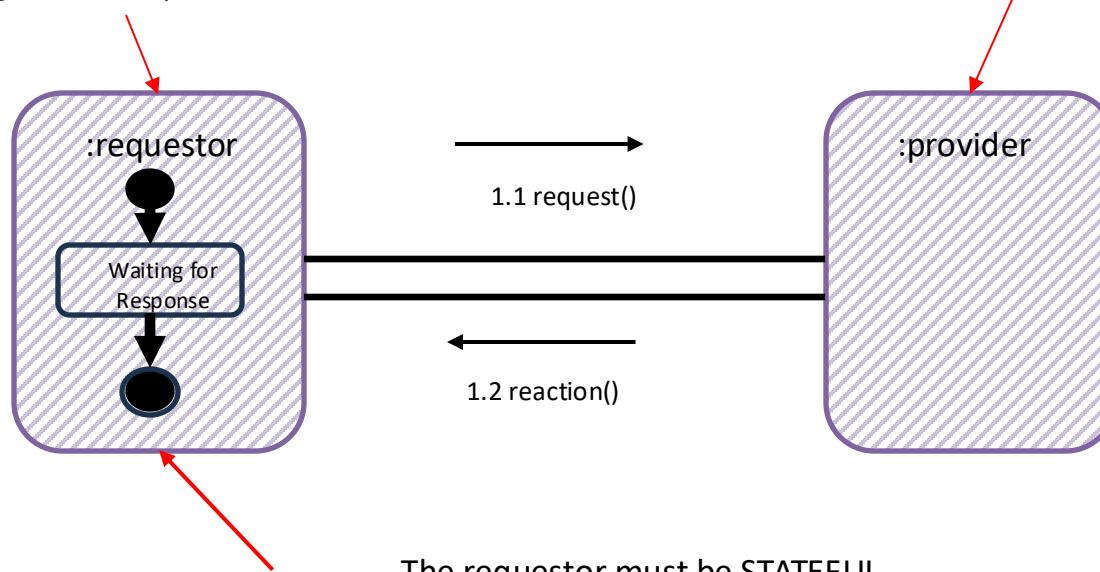
In order to resume, we need a way to identify the suspended workflow.

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



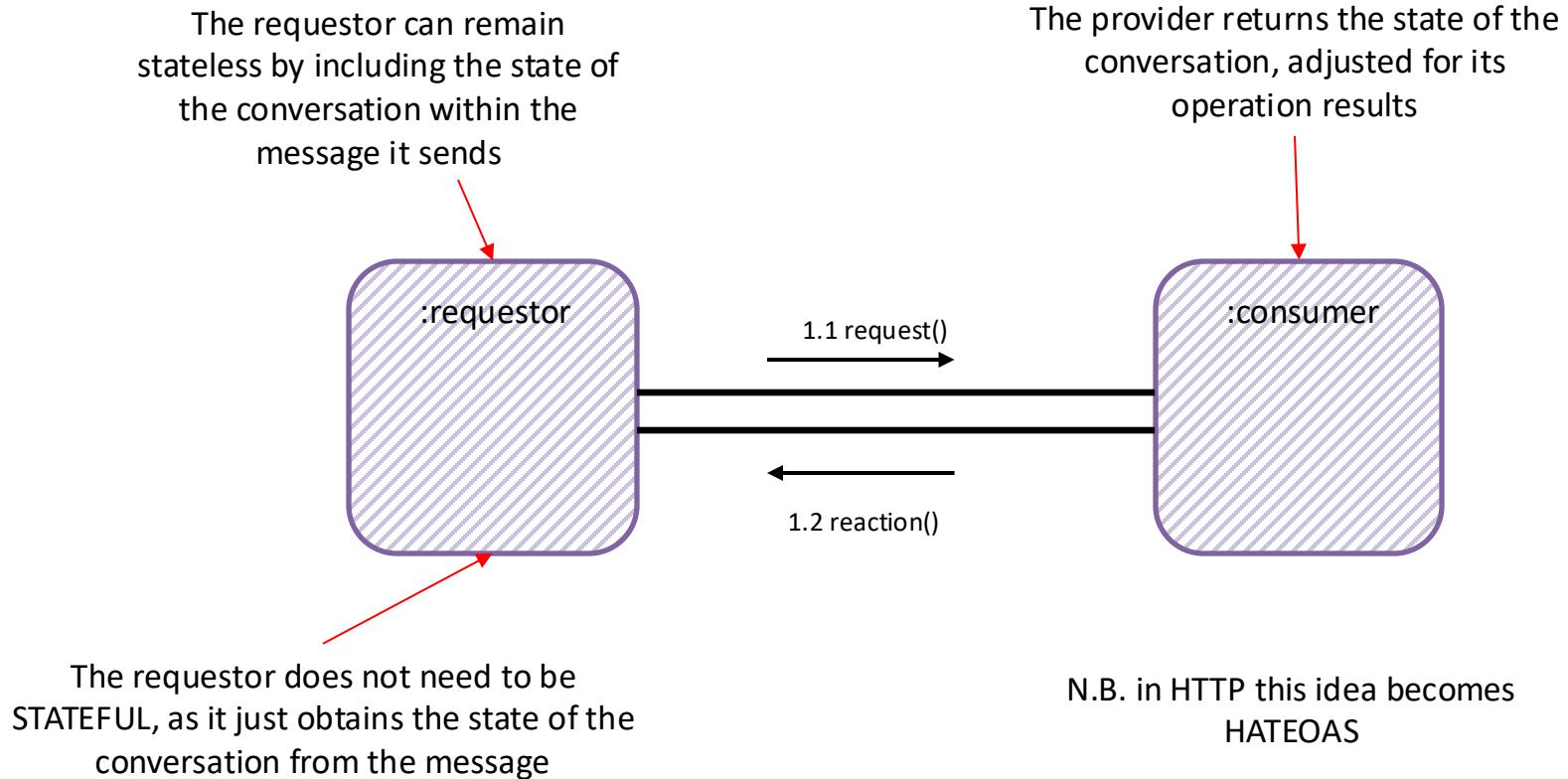
In-Out (Request-Reaction)

Because the flow is asynchronous, the requestor enters a waiting for response state – as it cannot complete its own operation until it gets the response.

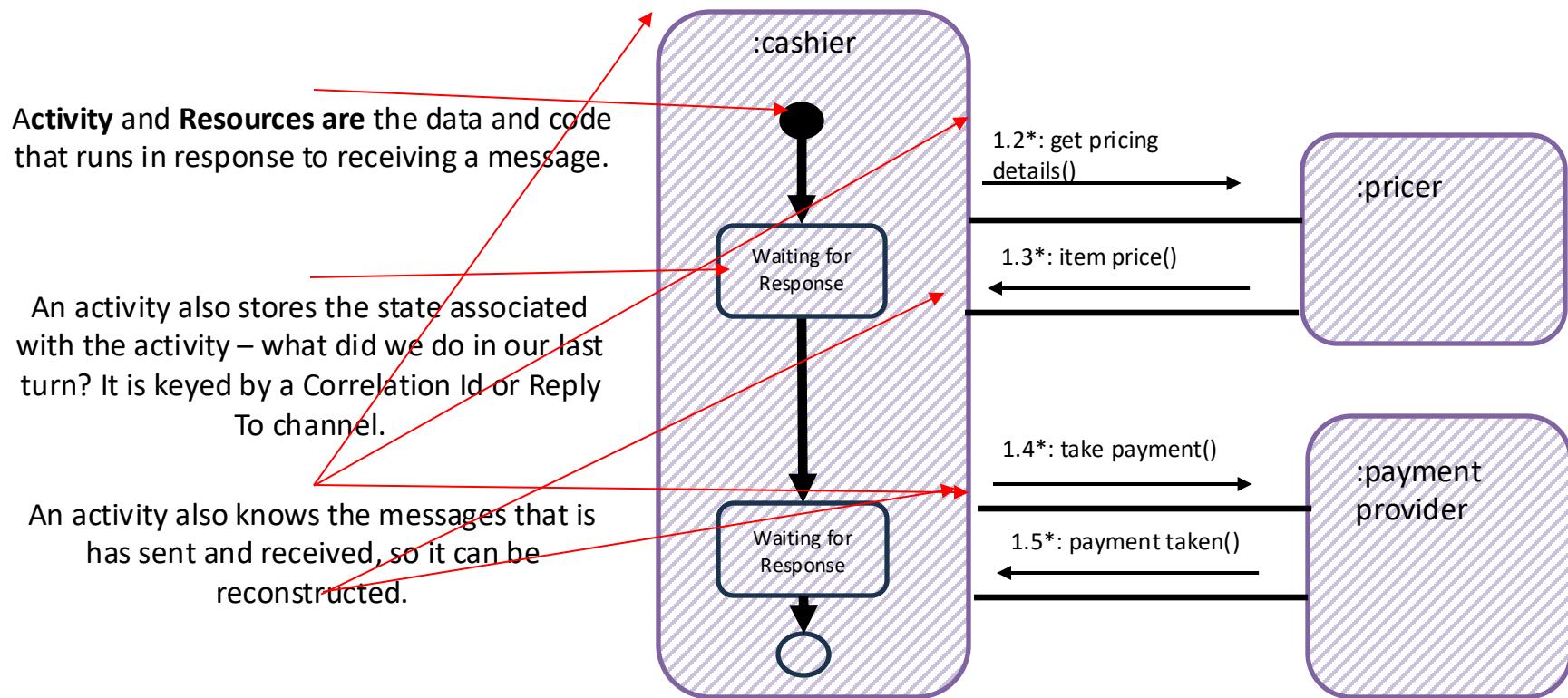


The provider can remain stateless as it only needs to return a response to the requestor

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



In-Out (Request-Reaction) Activity and Resources



Activity. Data and computation used to track the work, or part of the work, for a single collaboration.

Resources. Code and data used to manage shared items coordinated across multiple activities such as widgets in inventory or space on a truck needed for a shipment.

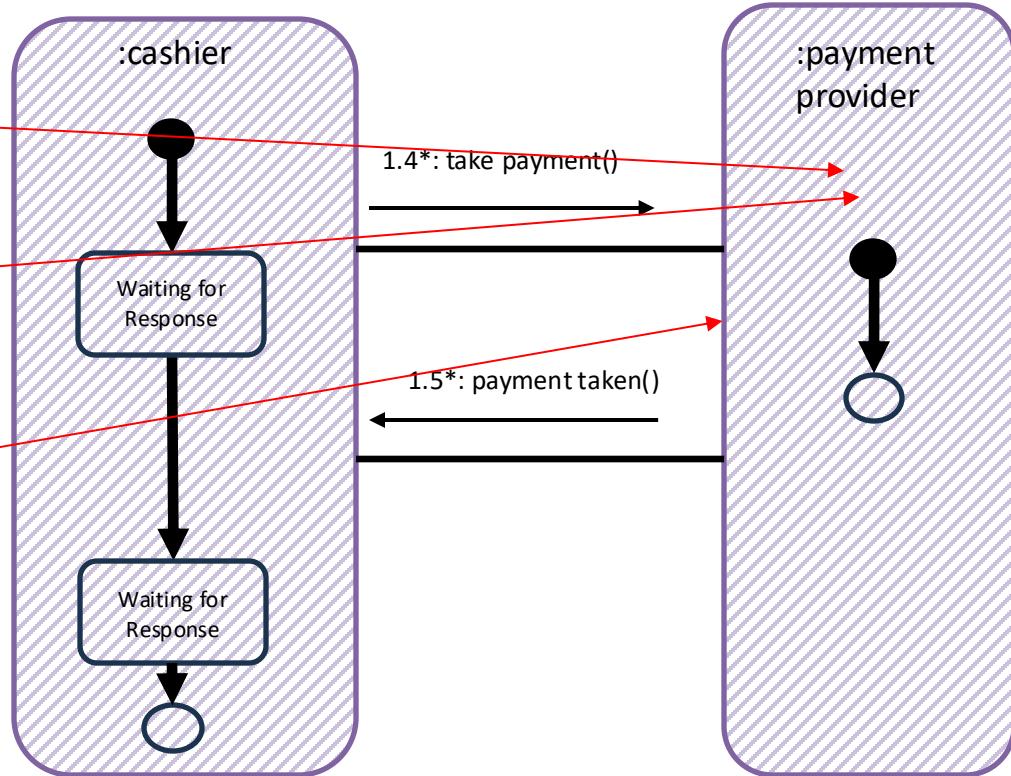
– Path Helland, Autonomous Computing

In-Out (Request-Reaction) Activity and Resources

Activity and **Resources** are the code that runs in response to receiving a message.

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

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



Activity. Data and computation used to track the work, or part of the work, for a single collaboration.

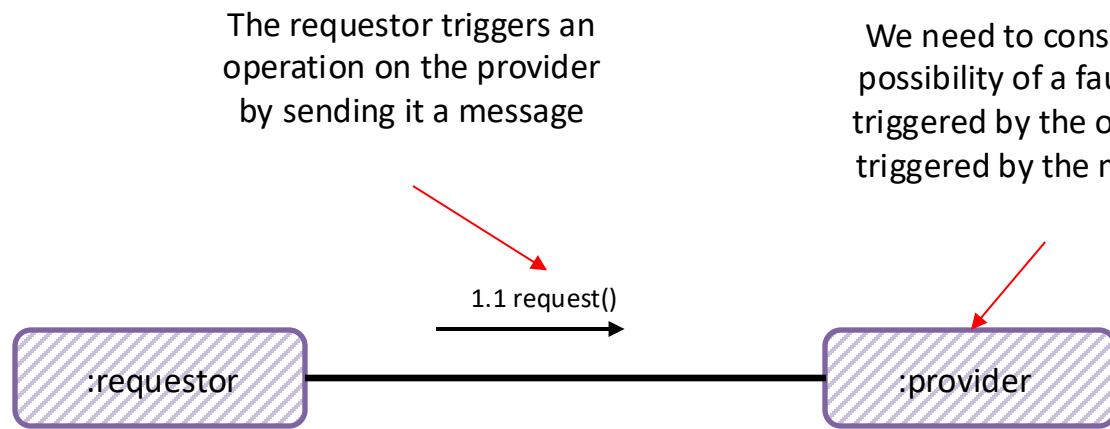
Resources. Code and data used to manage shared items coordinated across multiple activities such as widgets in inventory or space on a truck needed for a shipment.

– Path Helland, Autonomous Computing

What happens when it all goes wrong

REPAIR AND CLARIFICATION

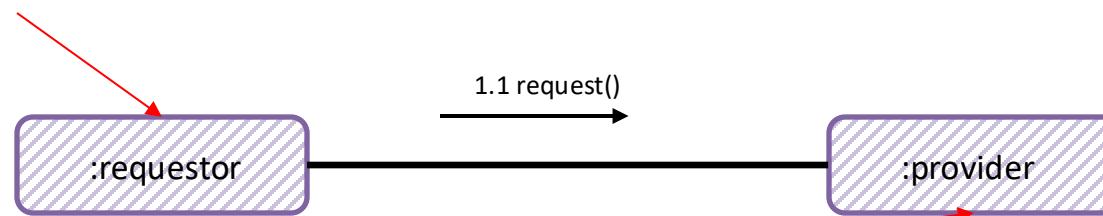
Repair and Clarification



What guarantees does the provider make to the requestor about communicating any faults that are triggered?

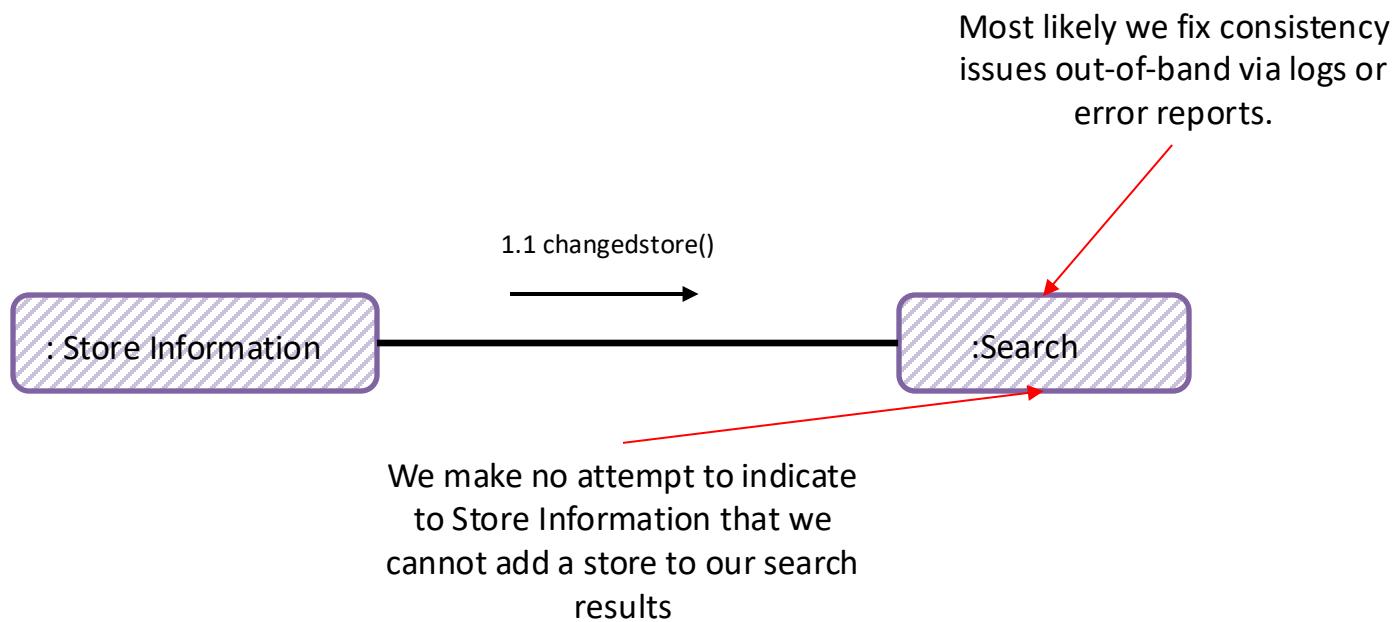
No Fault

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



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

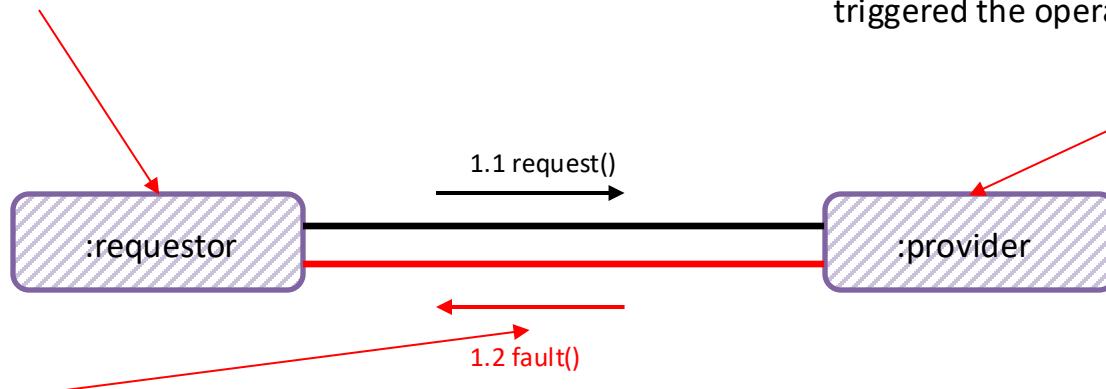
No Fault



Message Triggers Fault (Robust In-Only)

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

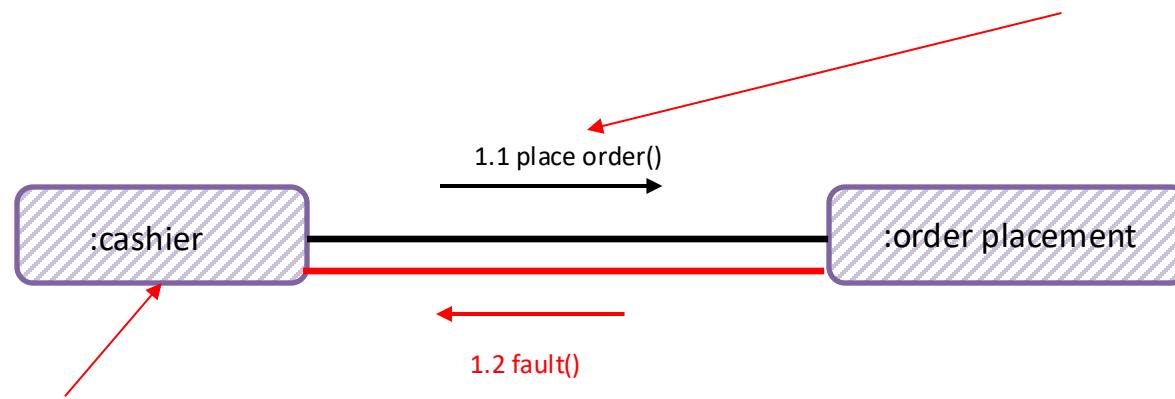
Under Message Triggers Fault pattern a **provider** propagates triggered faults from the operation back to the **requestor** that triggered the operation via a message



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

Message Triggers Fault

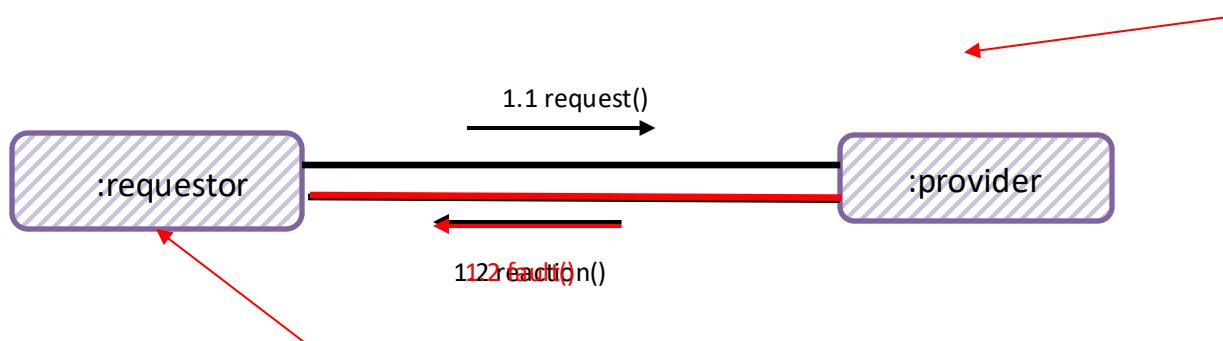
If Order Placement cannot place the order due to a fault we may raise an error



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

Fault Replaces Message

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

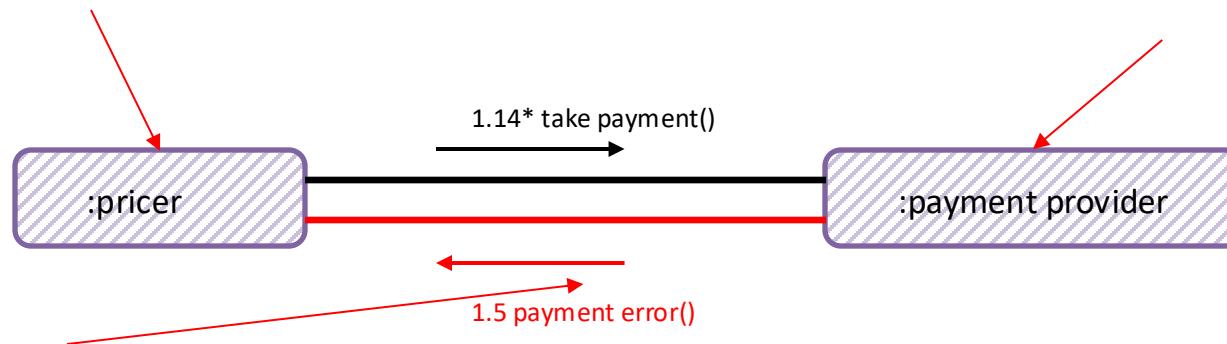


The requestor can handle the error; the error replaces the existing response

Fault Replaces Message (Robust In-Out)

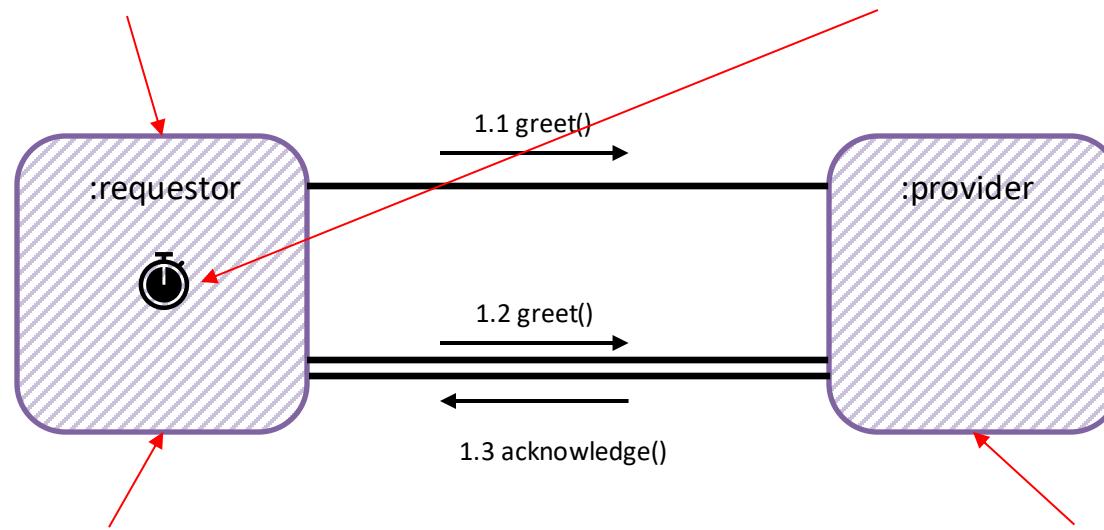
Assumption here is that the pricer can orchestrate a new flow, such as asking for an alternate payment method, or cancel the order.

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



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

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



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

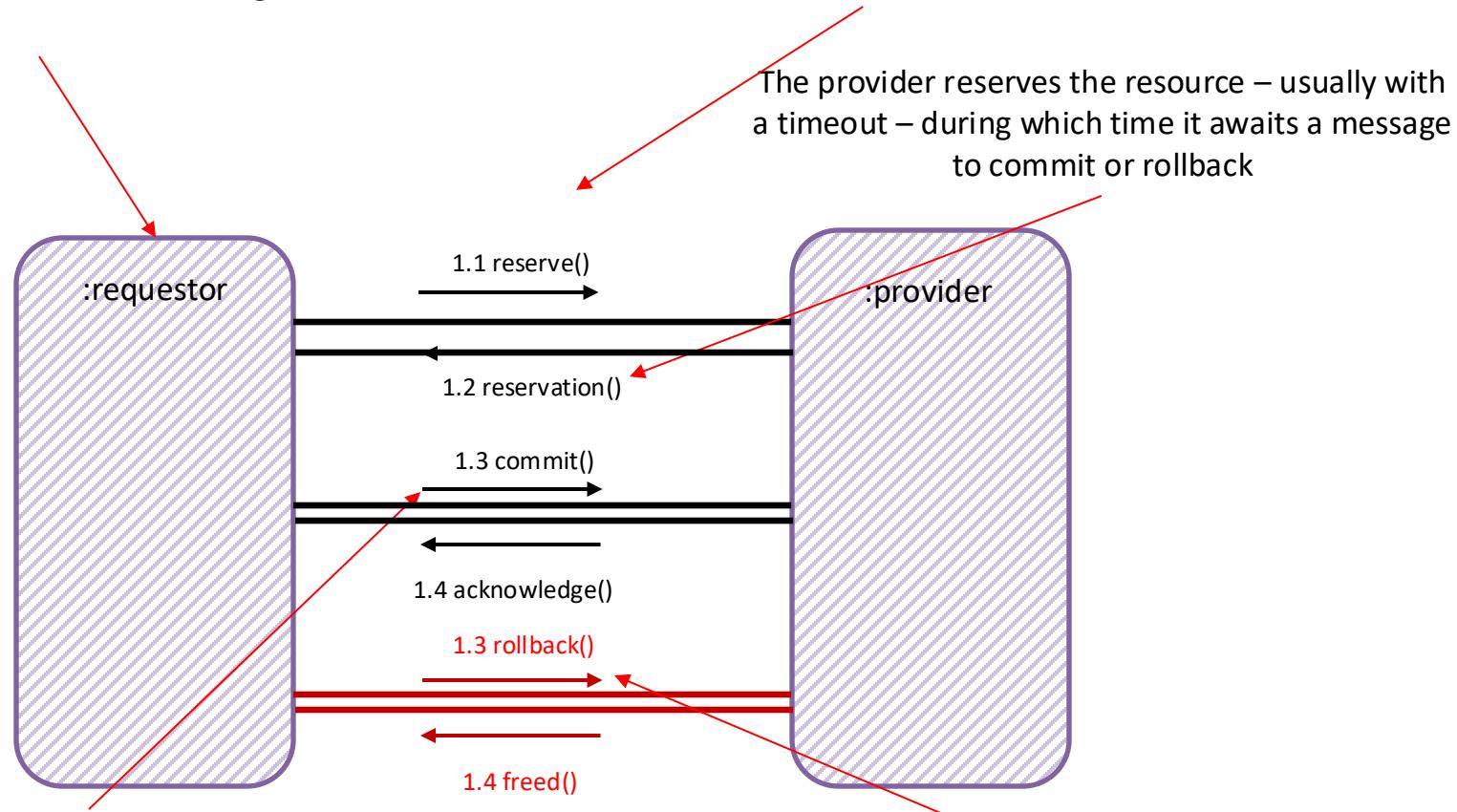
The requestor can set a timeout within which to receive a response.

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

Tentative Operations

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

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



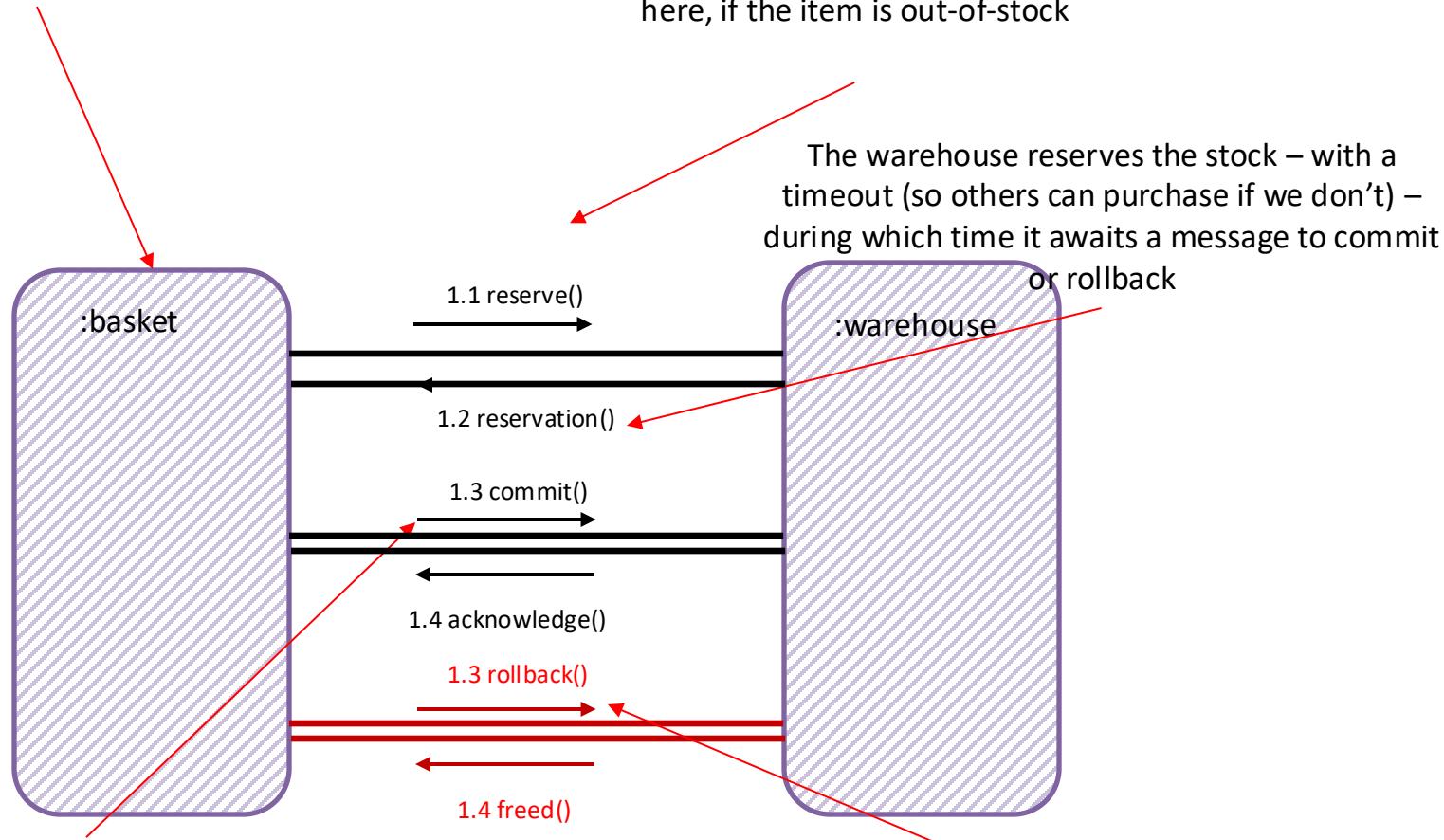
If the requestor completes its work, it then commits to ask the provider to allocate the reserved capacity.

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

Tentative Operations

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

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



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

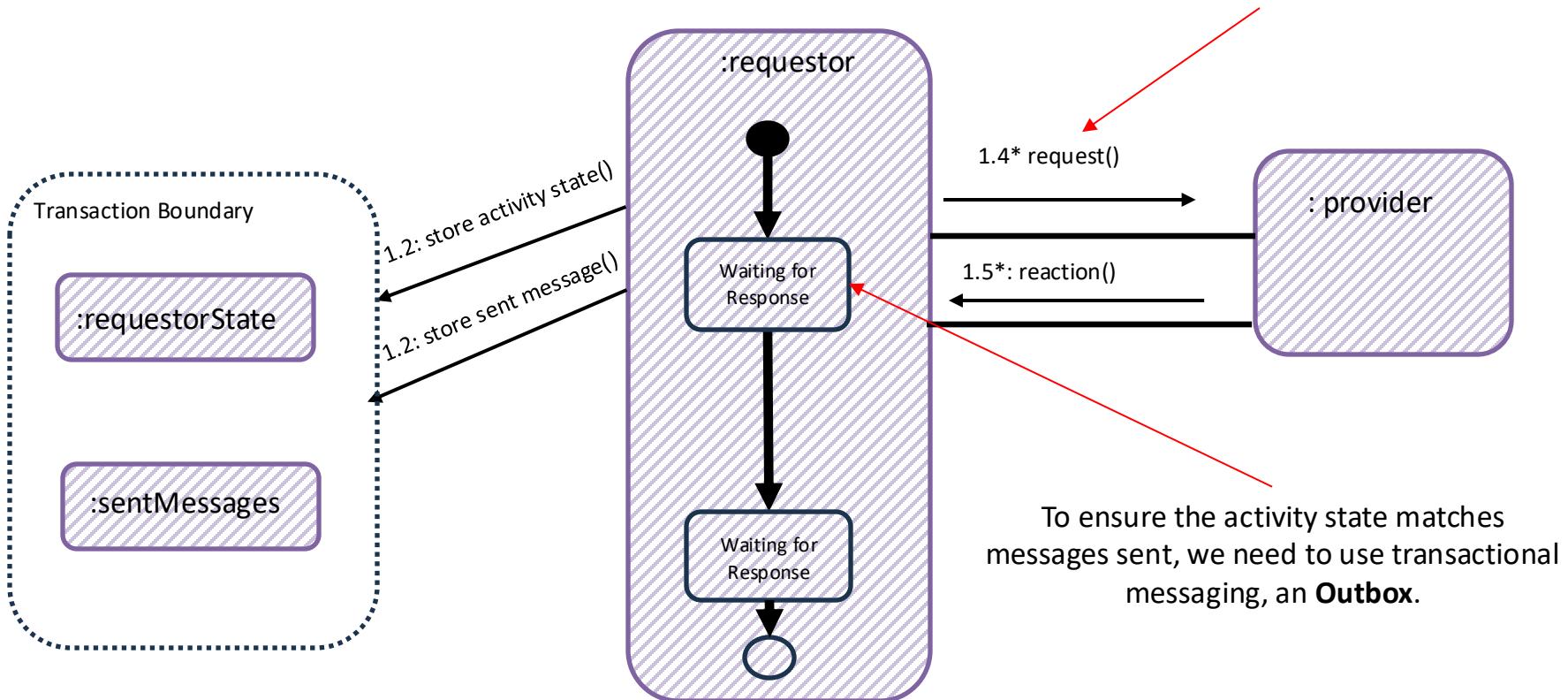
If the basket does not complete, then it rolls back to ask the warehouse to free the stock.

How do we ensure that our messaging is reliable?

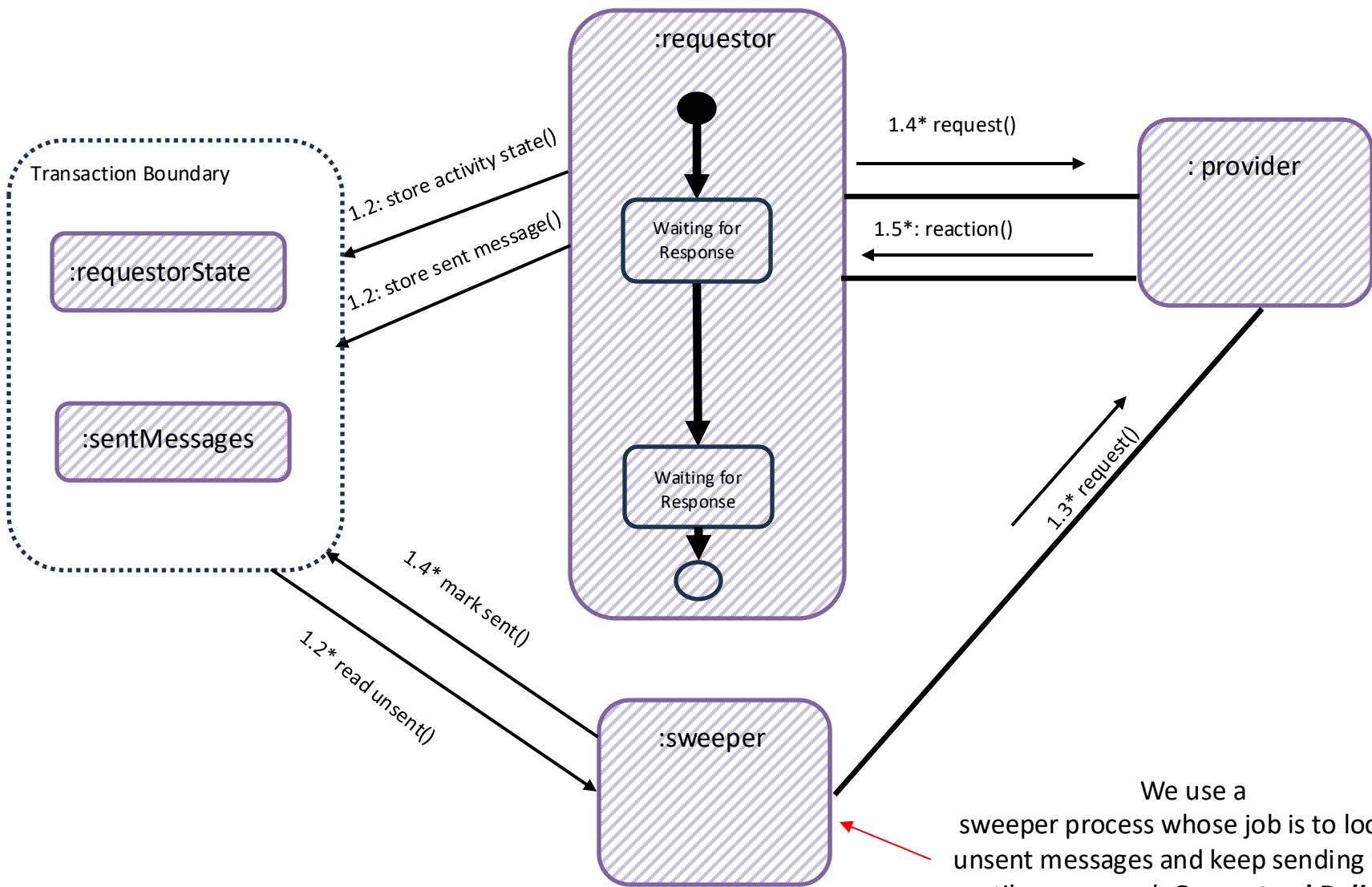
RELIABLE MESSAGING

Reliable Messaging – Outbox

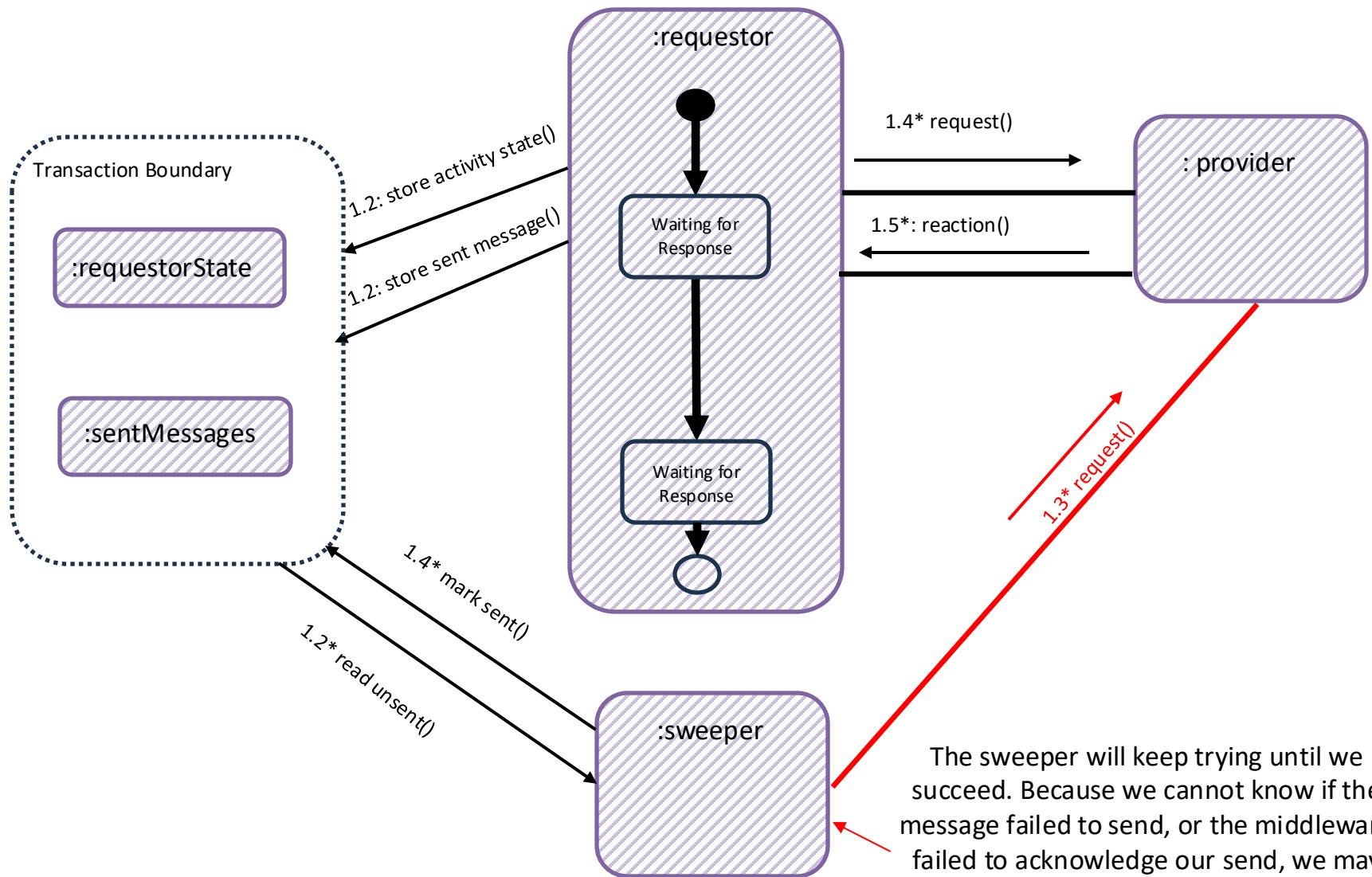
Sending the message might fail, this would result in our requestor's state being incorrect – it's waiting for a response it will never get!



Reliable Messaging – Outbox - Guaranteed Delivery

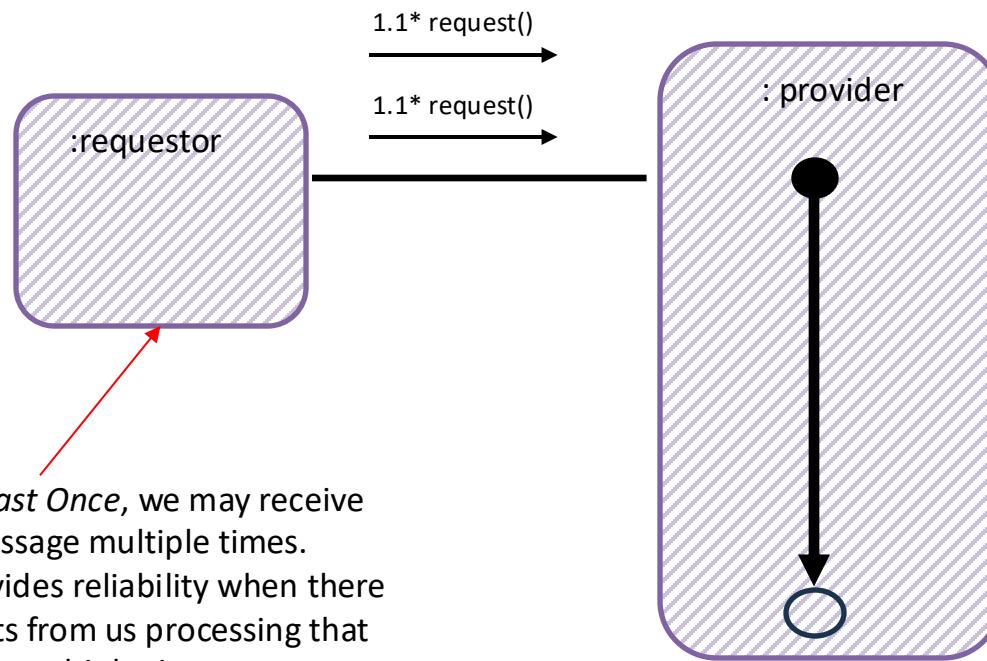


Reliable Messaging – At Least Once

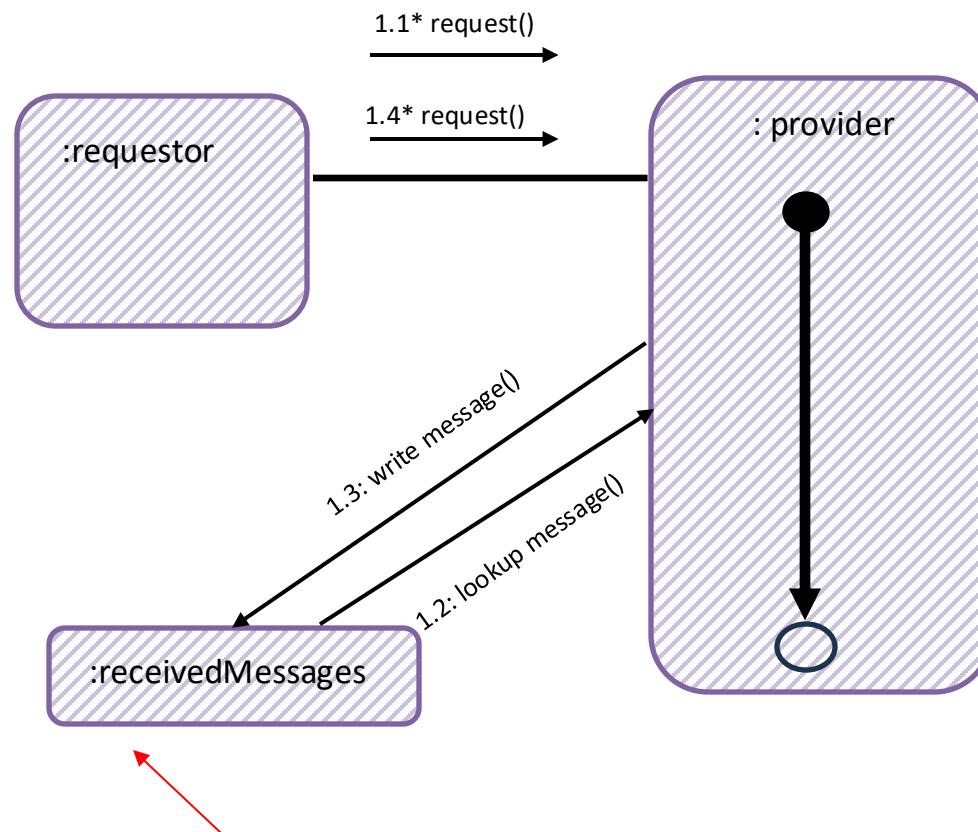


The sweeper will keep trying until we succeed. Because we cannot know if the message failed to send, or the middleware failed to acknowledge our send, we may resend a message that has been sent. **At Least Once.**

Reliable Messaging - Idempotency

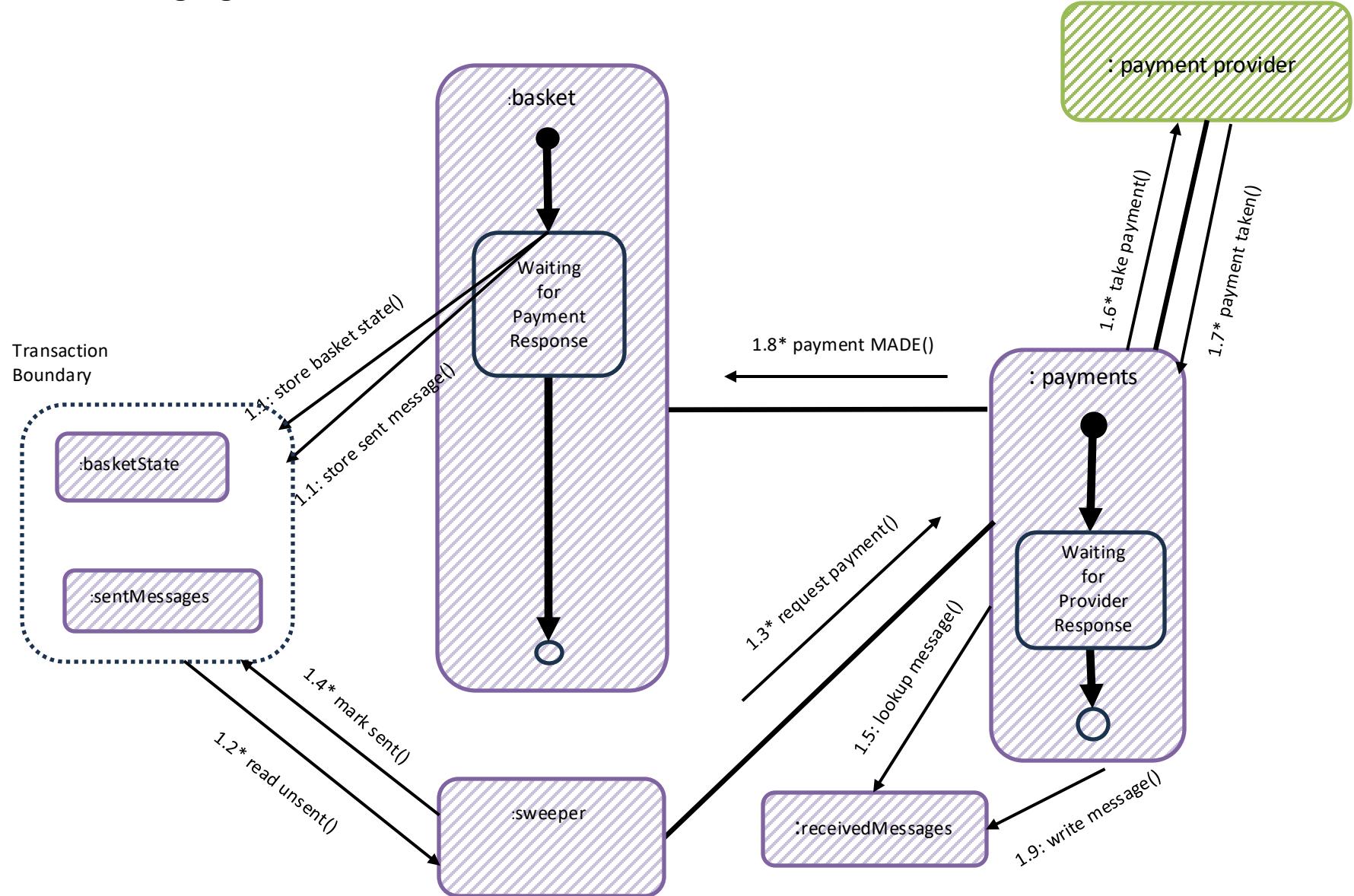


Reliable Messaging - Inbox



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

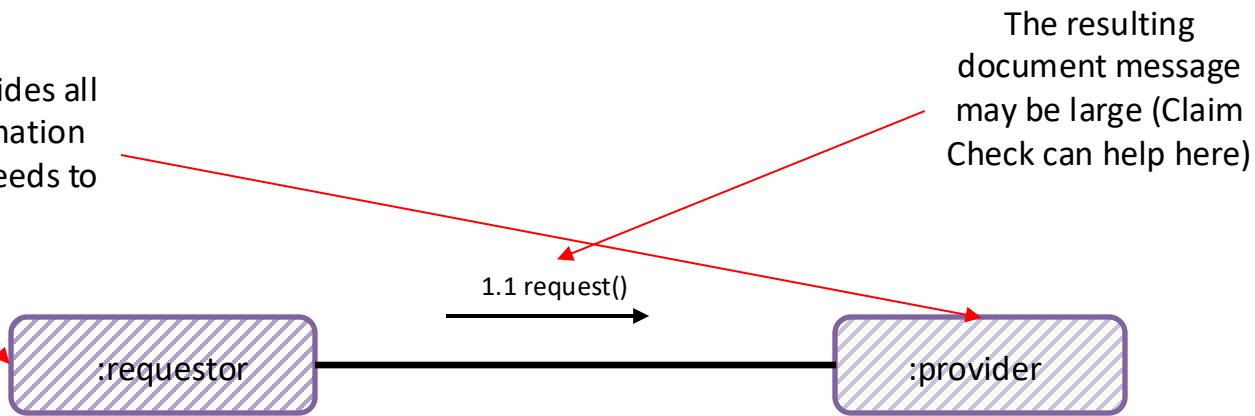
Reliable Messaging



FAT AND SKINNY

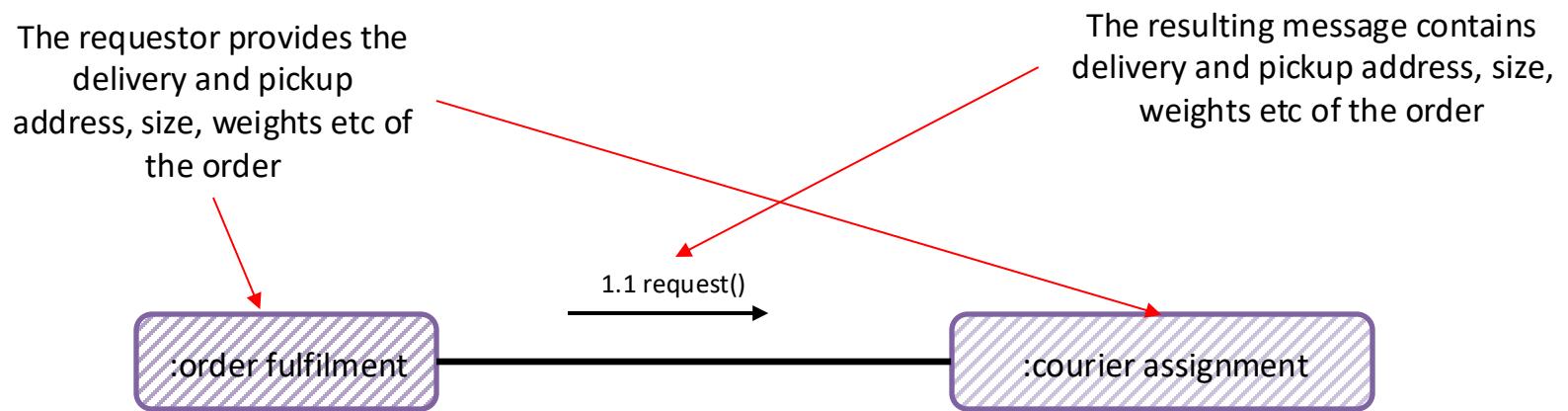
Fat Message

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



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

Fat Message Example



Fat Message, Transitive Dependencies

Purchase Order Message

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

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

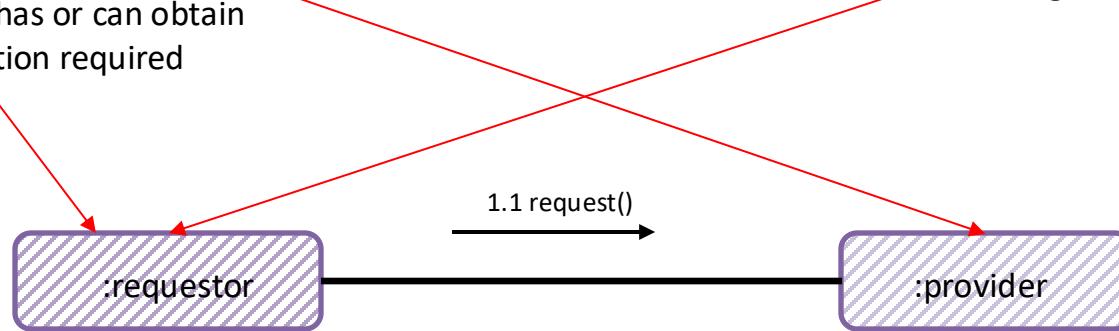
This data is a **transitive dependency**, it has
a lifetime of the Customer and their
schema changes if the Customer schema
changes – which may force us to change
the message for Customer changes

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

Skinny Message

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

The resulting notification message is normally skinny

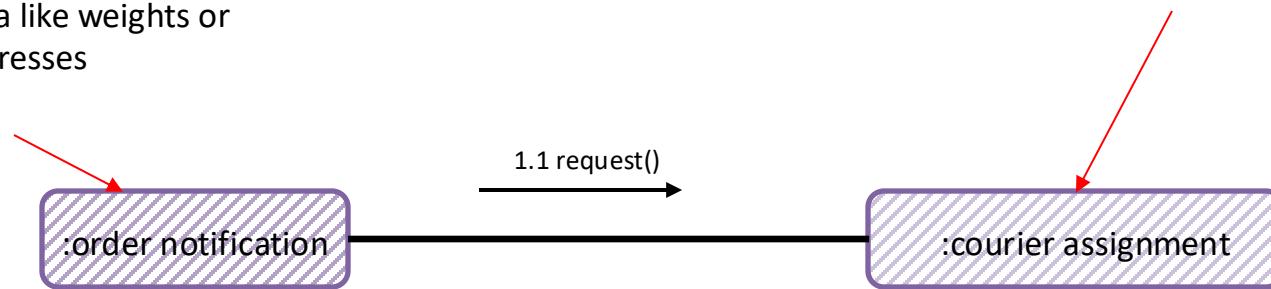


With a **Skinny Message** the requestor assumes the provider has necessary external information to perform the operation.

Skinny Request, Example

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

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



Skinny Message, Normalized

Purchase Order Message

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

Customer

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

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

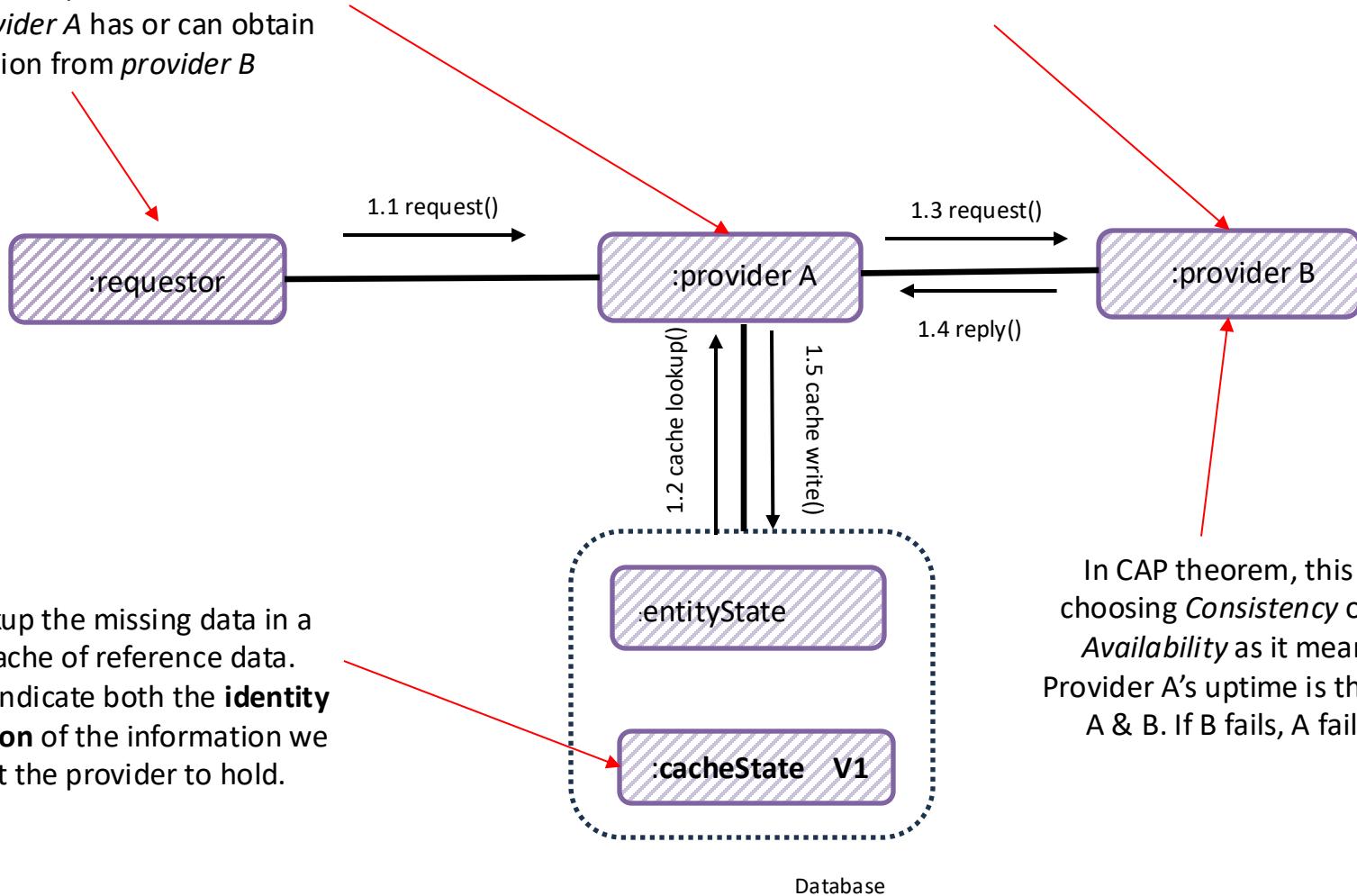
We need to lookup this id with other providers

Restaurant

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

Skinny Message via REST/RPC

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



Skinny Message using Reference Data

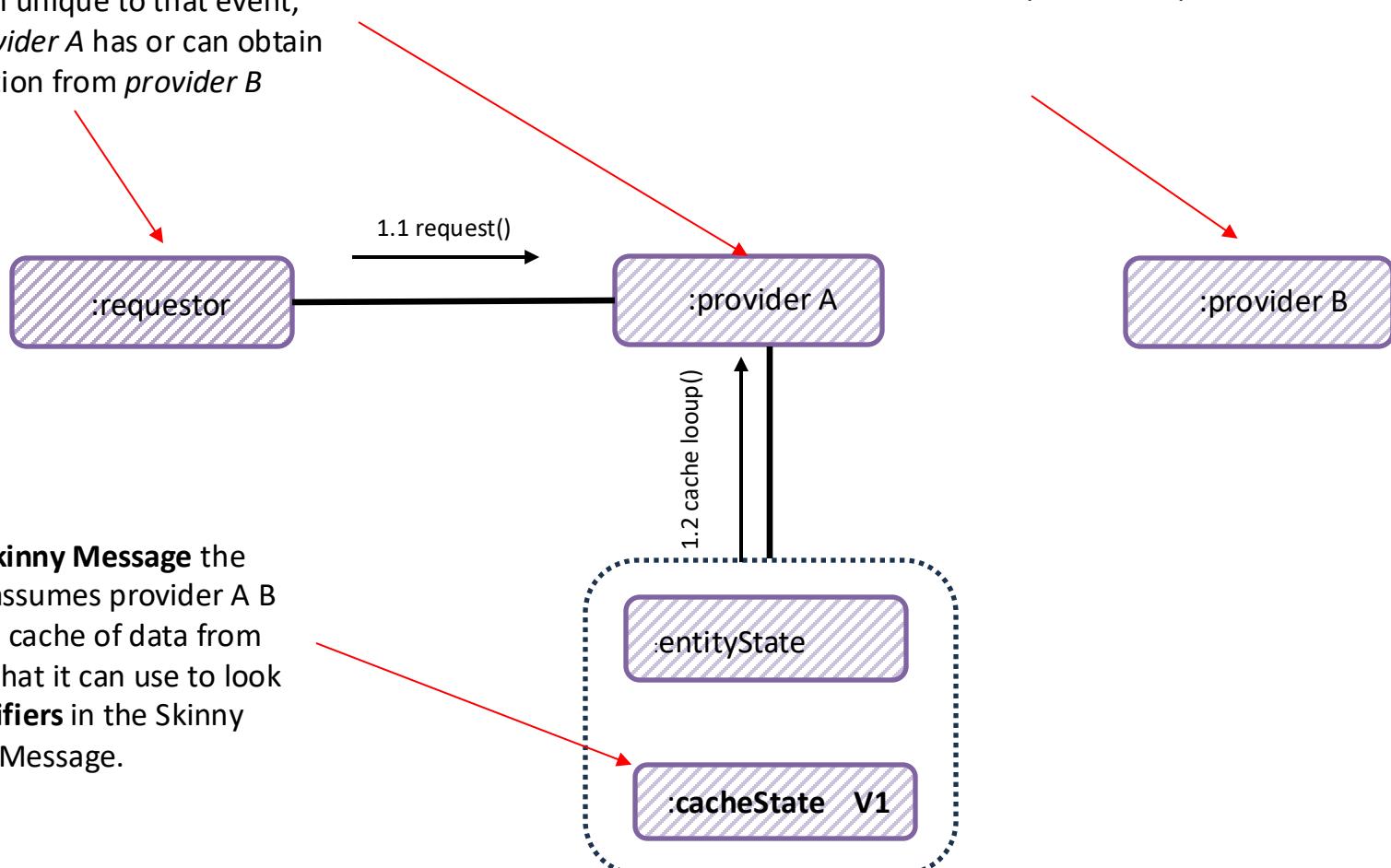
Data that leaves Provider B via an API is **Reference Data**.

It is:

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

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

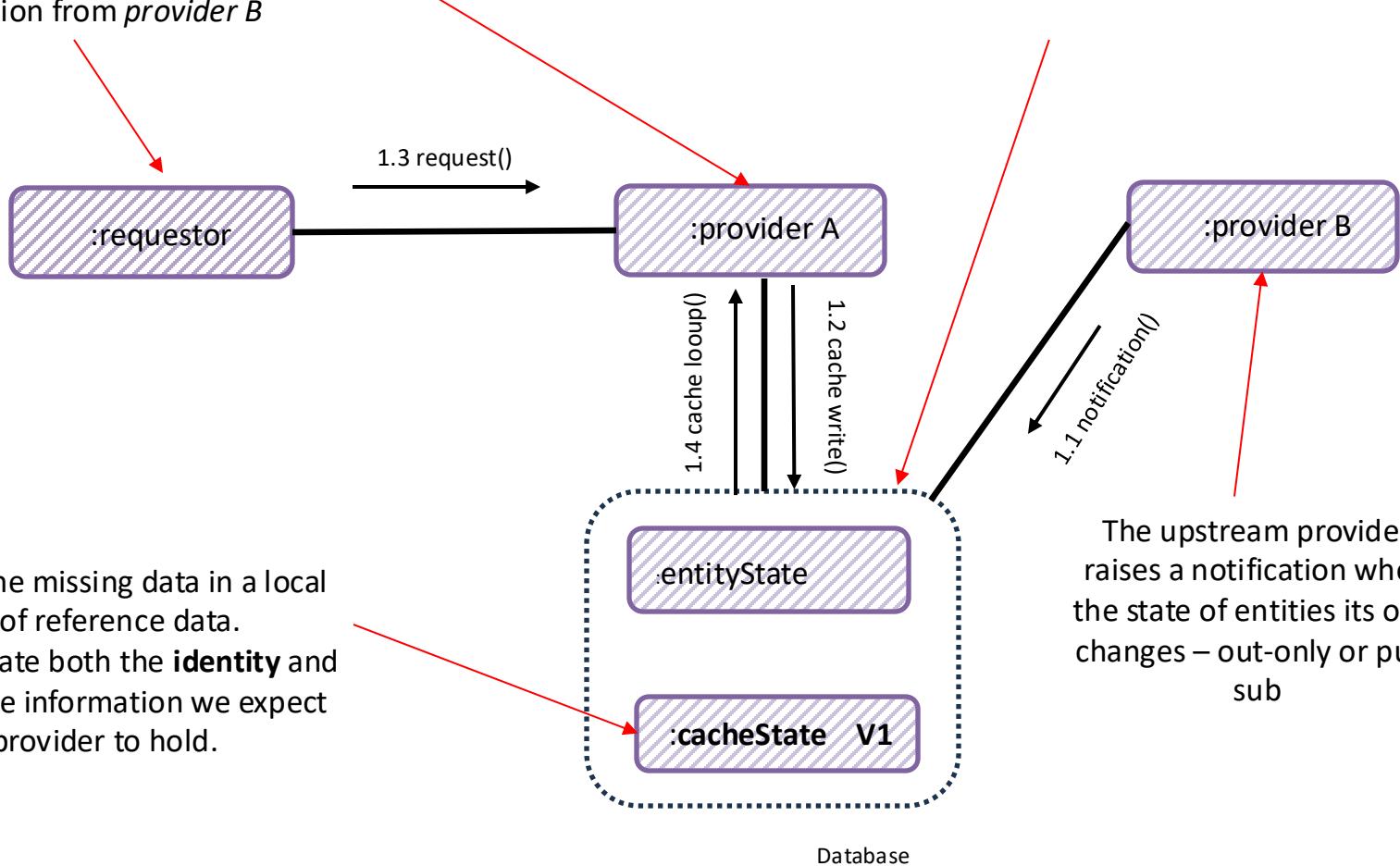
It can be cached locally to provider A to prevent the need for frequent lookups



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

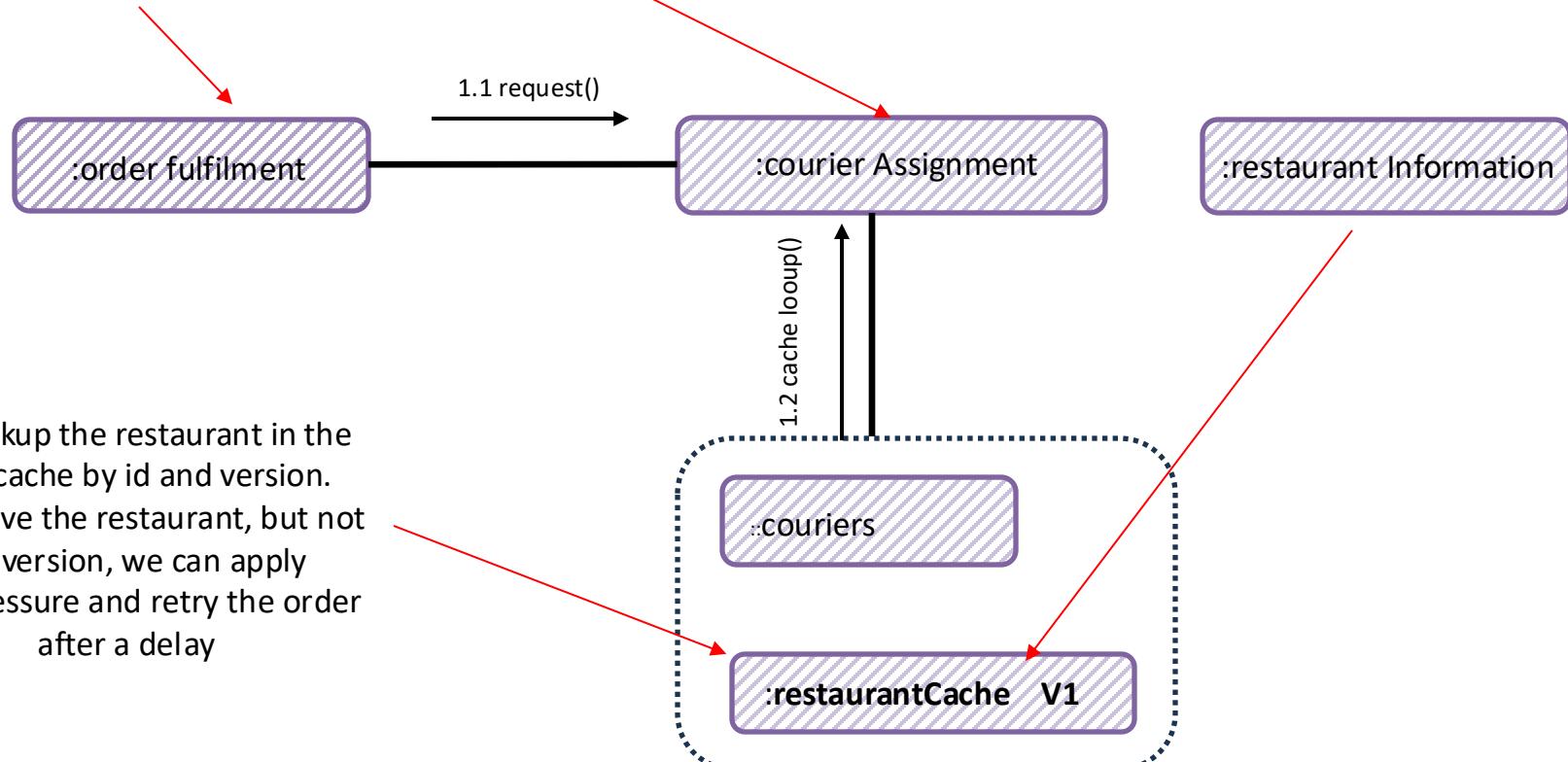
The requestor provides only information unique to that event, assumes *provider A* has or can obtain information from *provider B*

The downstream provider subscribes to these notifications and writes these to the local cache.
In CAP Theorem we choose *Availability* over *Consistency* – we choose to accept stale data over accepting the risk of failure due to a *Partition*.



Skinny Message using Reference Data

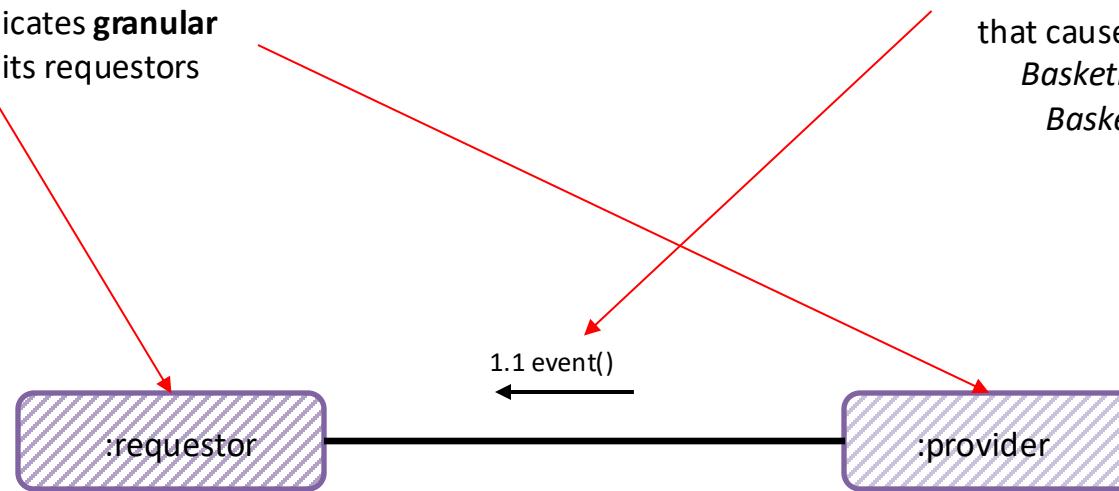
Order Fulfilment does not include address details for the restaurant for pickup, instead we assume that courier assignment has obtained it from restaurant information



DOMAIN, SUMMARY, ORDERING

Domain Event

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



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

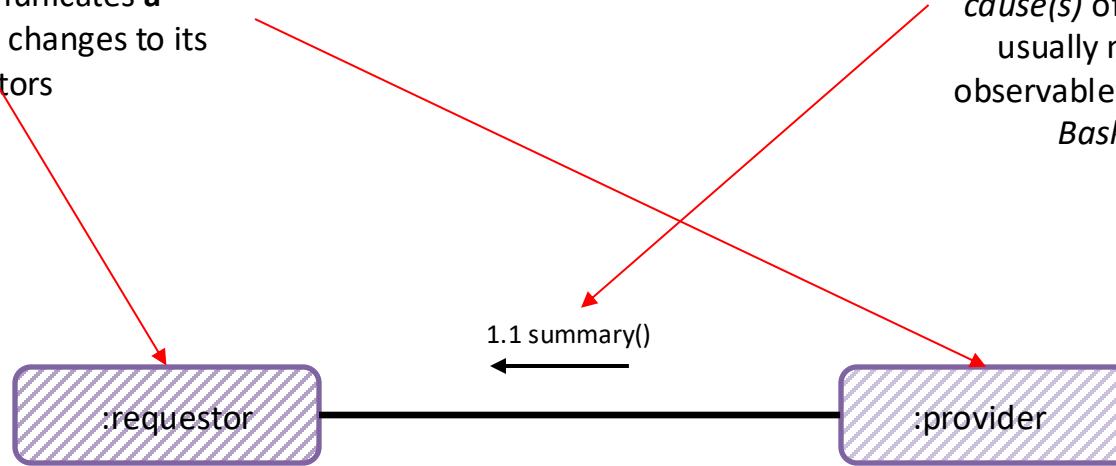
With a **Domain Event** the provider cannot use a *DataType Channel* as the domain events for the observable must be ordered on the same channel, and have a different schema.

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

Summary Event

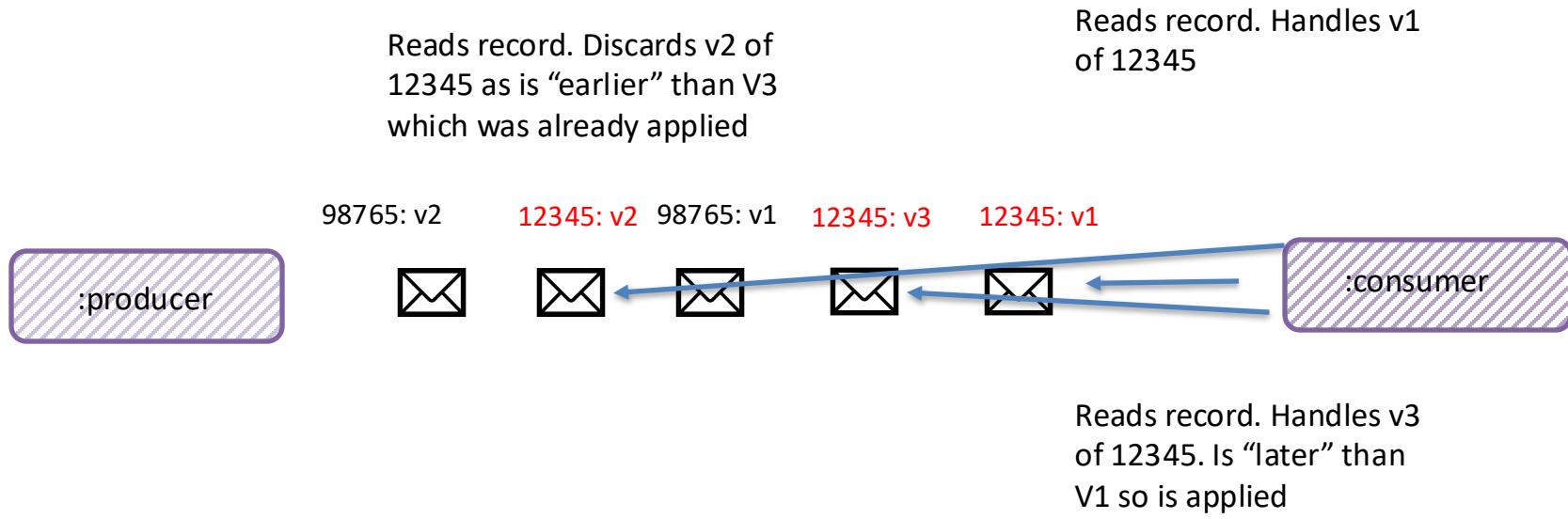
An approach to Pub-Sub where the provider communicates a **summary** of state changes to its requestors

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



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

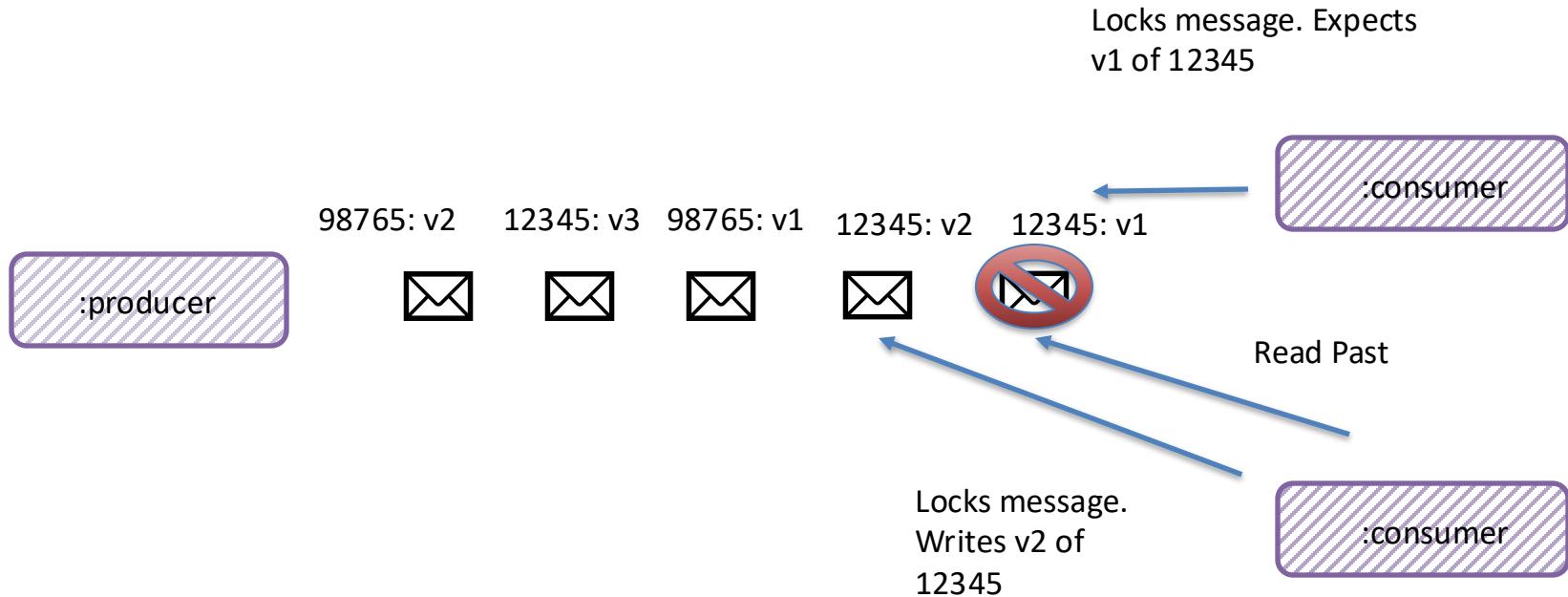
If Later, Stream



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

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

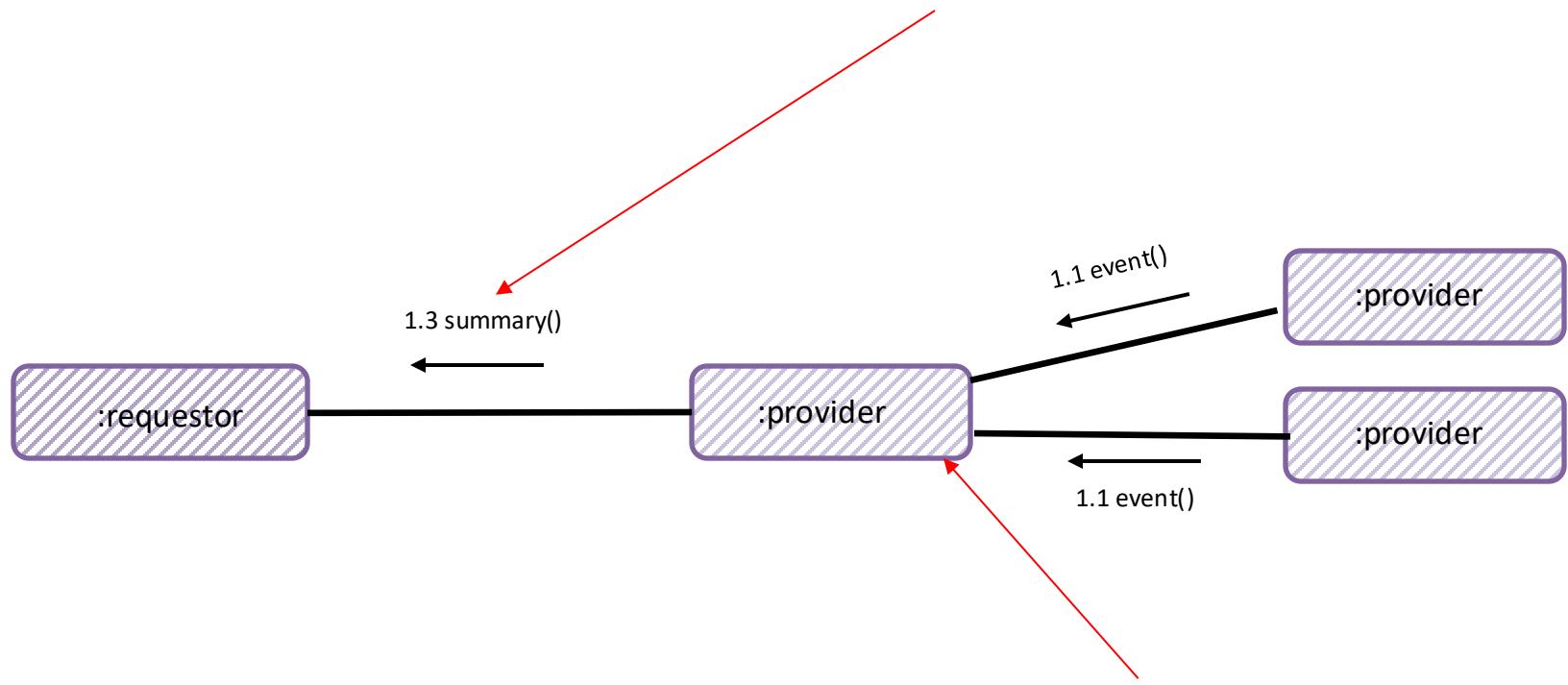
If Later, Queue



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

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

Public and Private Providers



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

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

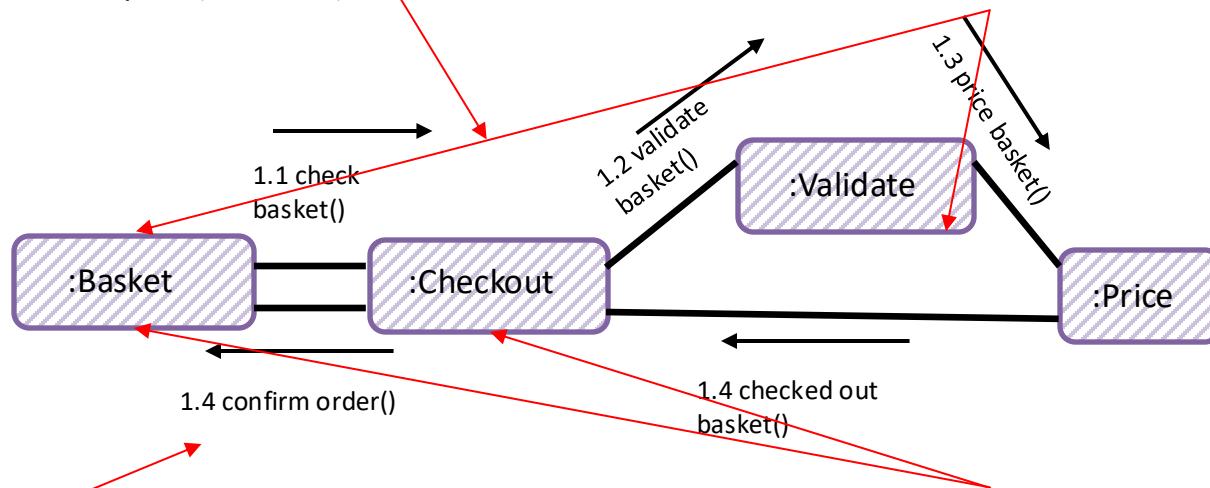
CONVERSATIONS

Services are Processes

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

Lack of Distributed Transactions

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



Command and Control

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

Coupling

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

Definitions

Conversation: when a component communicates with other components; may be a dialog (in-*, or out-*) or a monolog (in-only or out-only)

Workflow: literally the flow of work to achieve an outcome, partitioned into processes, which are in turn partitioned into activities, and again into tasks. A workflow may be implemented as one or more conversations between components.

Choreography: a workflow where conversations are Out-only, and components listen for work to do from the bus. Loosely coupled, emergent behaviour.

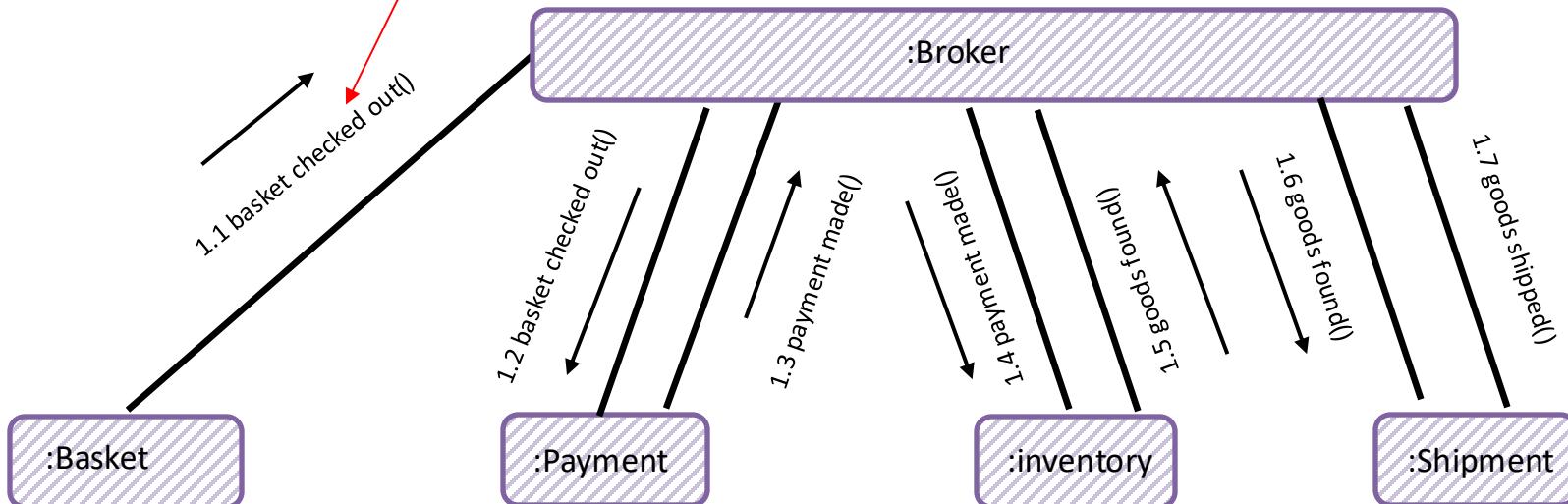
Orchestration: a workflow where the conversation is directed. Authority may be distributed: each component knows who it talks to, or the message tells one component which step is next; or hub-and-spoke, a central controller talks to each component.

Saga: a workflow that offers an eventually consistent transaction. Each component updates their local state for the change; if one component cannot make the change then that component signals to other components to reverse the change. May be choreography or orchestration.

Choreography – Composing with Events

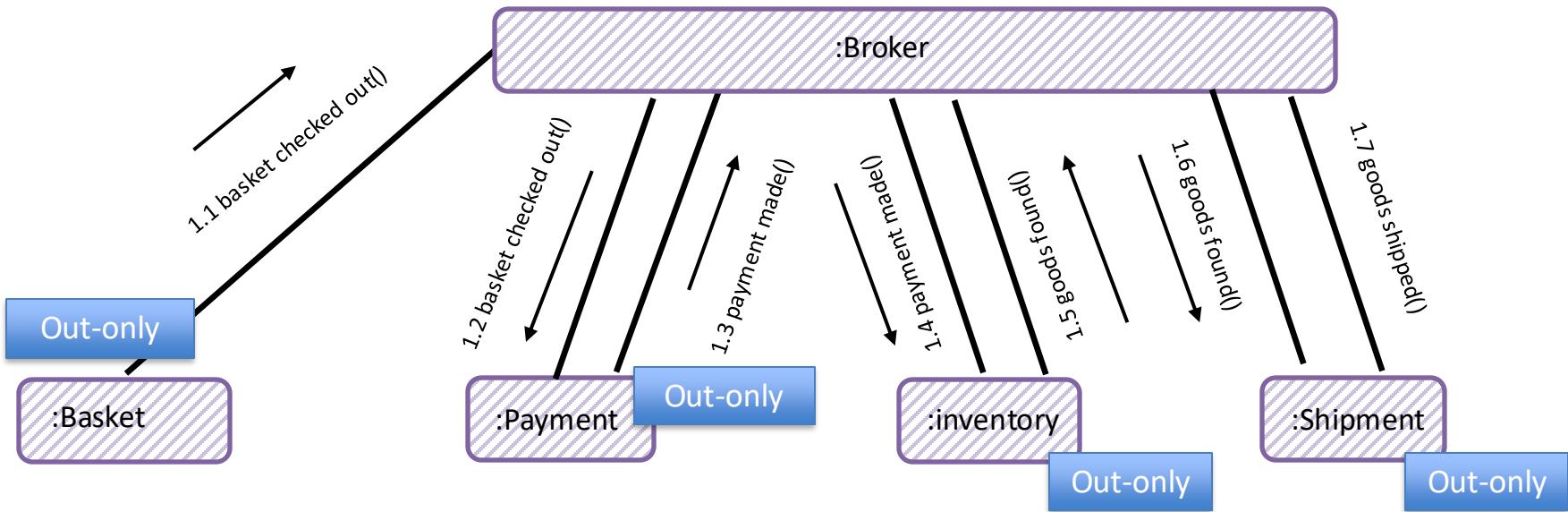
Events are Facts

Events tell us about the outcome of work
flowing through a component



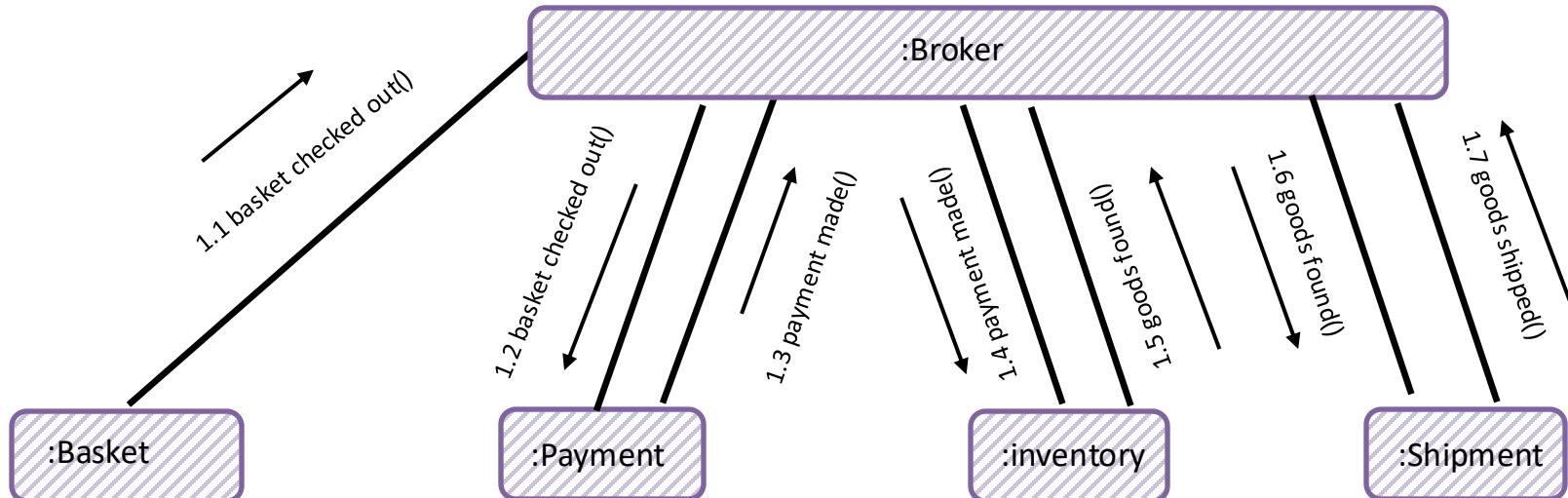
Loose Coupling: this is attractive because it is loosely coupled –components only know about the broker, and not about each other; this makes it easy to change who listens, add listeners etc. FAST to get started as teams have high independence!!!

Choreography – Composing with Events



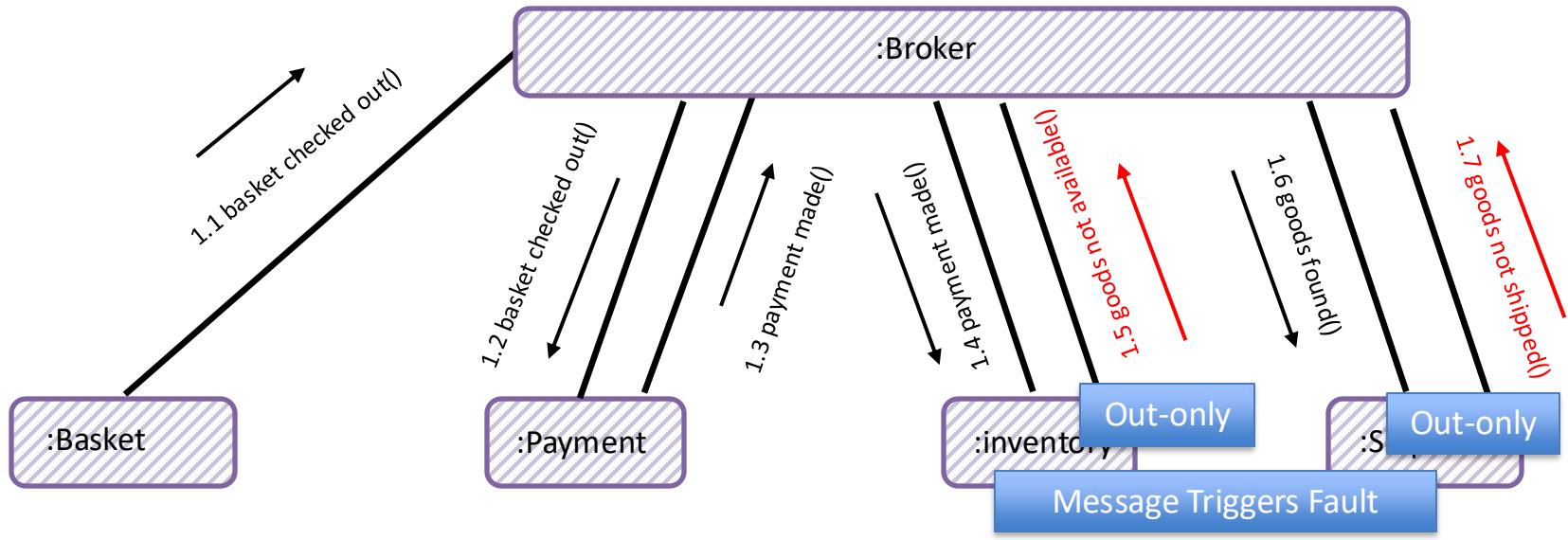
Event Chaining: to make an outcome happen, goods shipped, we rely on listeners existing who can trigger another action. This is known as event chaining – we chain events together for an outcome. The problem with event chaining is that we make it hard to see the flow – the **Pinball** anti-pattern: this is hard to reason about, observe, or diagnose.

Choreography – Saga (Eventing)



Saga: there is a business transaction, in this example: if picking or shipping fails, we need to refund the customer. The transaction requires each component in the event chain to succeed, or they must roll back. So we need to provide a saga over choreography.

Choreography – Saga (Eventing)



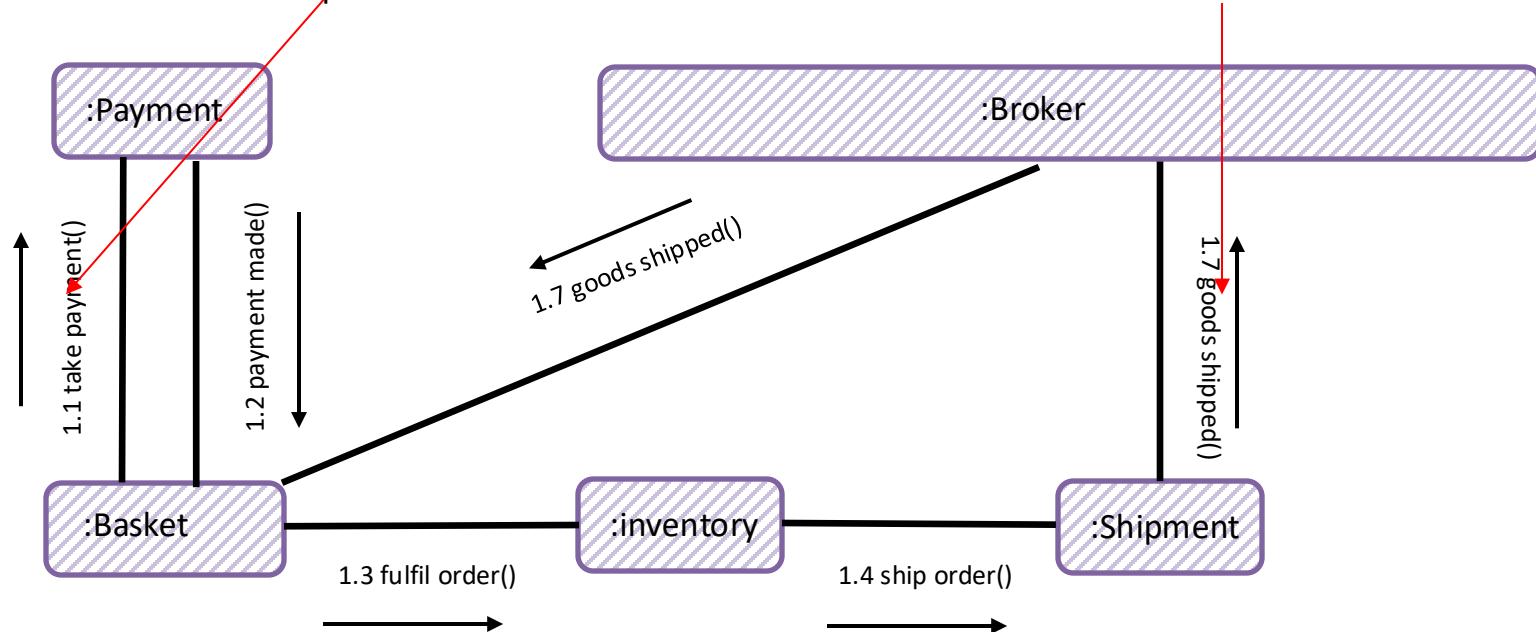
Saga over Choreography: we use a **Fault Triggers Message** approach and raise an event if we fail. Each component that could fail, must offer Fault Triggers message. This fault is Out-only to preserve our choreography, and assumes upstream components will reverse in response.

Complexity: with propagation back up the chain, what happens if a chain step fails to reverse the transaction, due to error or state? We need to listen to failure events from any part of the chain, implying handling code for each provider. We may get emergent behaviour over order of reversal – goods not shipped needs to reverse both picking from the inventory and payment.

Orchestration – Composing with Messaging (Turn Taking)

Messaging is Operations
A message invokes an operation exposed by another component

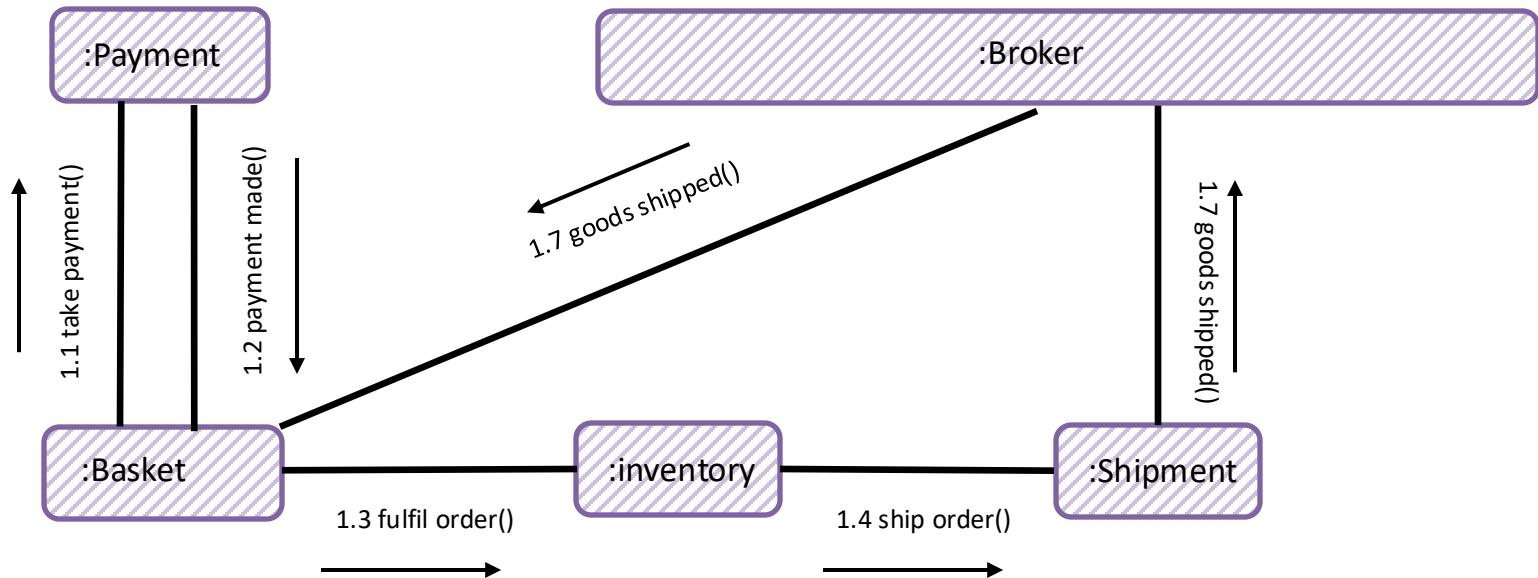
Can Mix with Events
If someone needs to be informed about outcomes, or progress, of a workflow, we may use events to broadcast that.



Turn Taking: a component has a dialog with other components; each component knows how to run its part of the workflow – happy path and failure path; knowledge of the workflow is distributed among the actors.

Responsibility and Causality: this is attractive because we have causality – things happen in a known sequence and components clearly have responsibilities for carrying out actions within the flow.

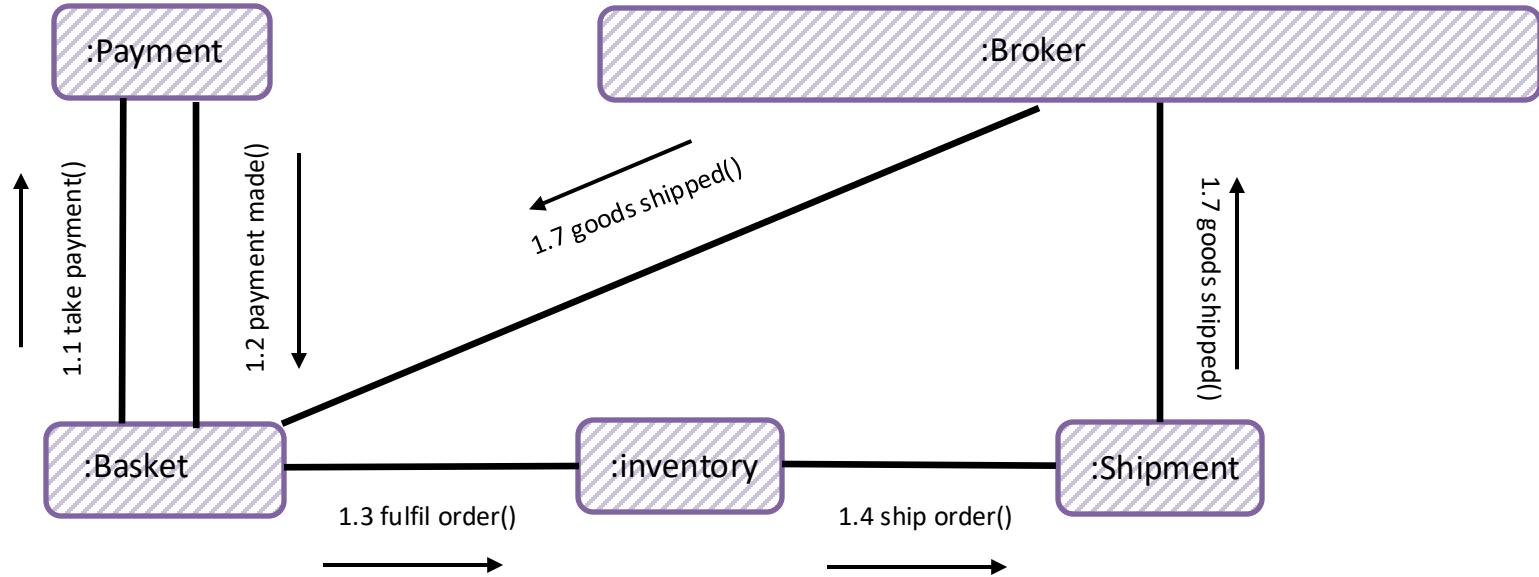
Orchestration – Composing with Messaging (Turn Taking)



Behavioural Coupling: There is coupling, we know about the provider we must call – and the workflow is distributed amongst the components; the trade-offs are simplicity (each component knows what it needs nothing else) and predictable as opposed to emergent behaviour (causality).

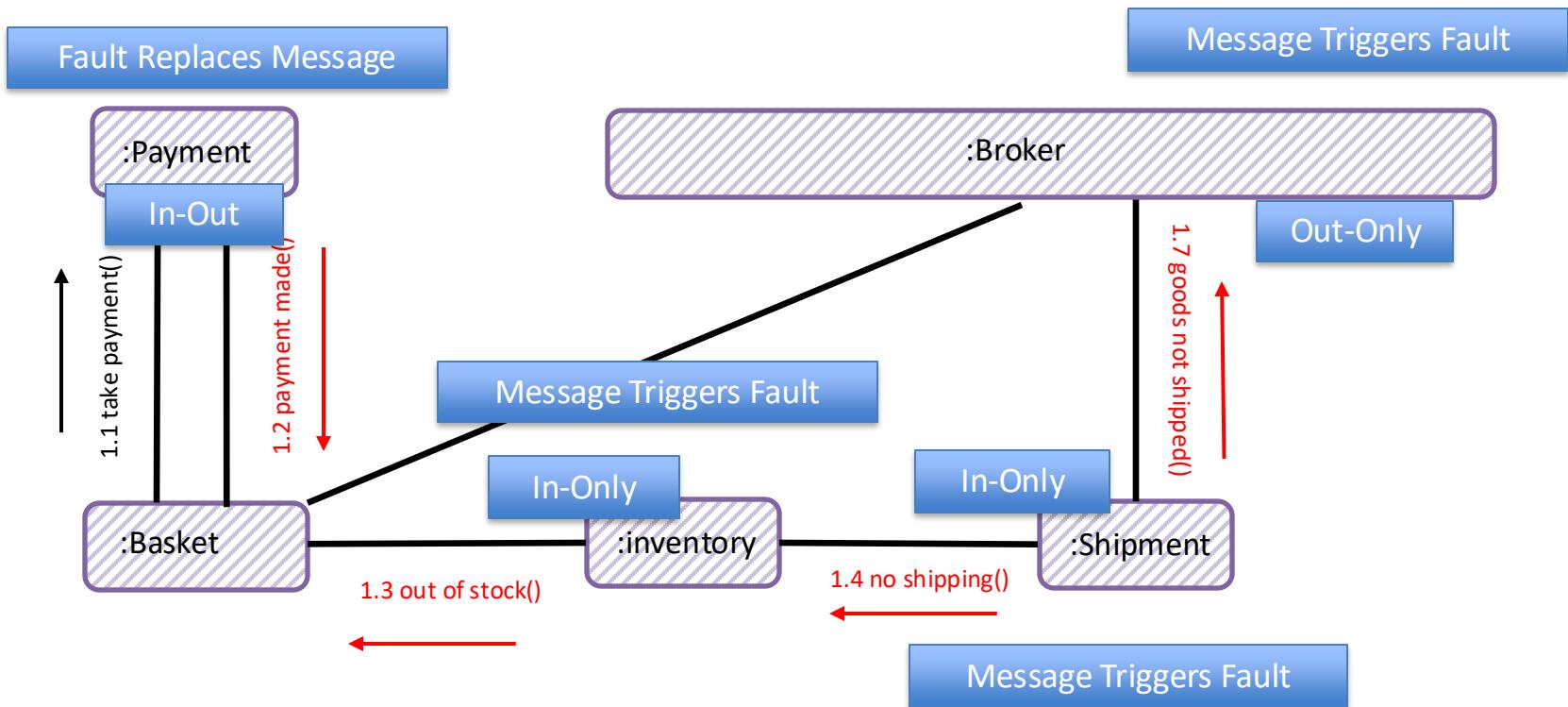
Coupling vs. Control: behavioural coupling is reduced to neighbours, but changing the workflow requires altering multiple components; no central control to understand flow

Orchestration – Saga (Turn Taking)



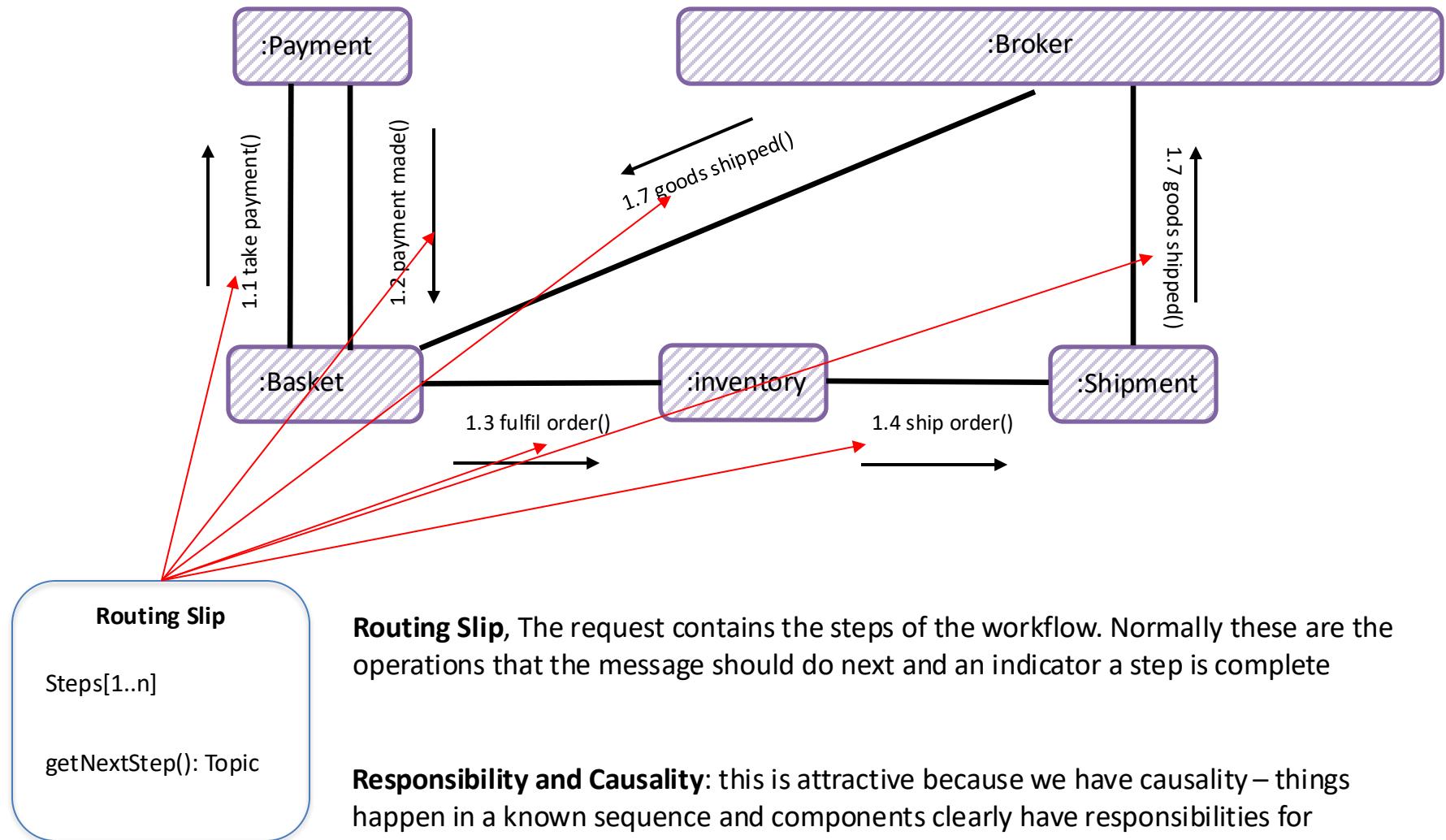
Saga: there is a business transaction, in this example: if picking or shipping fails, we need to refund the customer. The transaction requires that we roll back steps that have already been completed i.e. we reverse the flow.

Orchestration – Saga (Turn Taking)

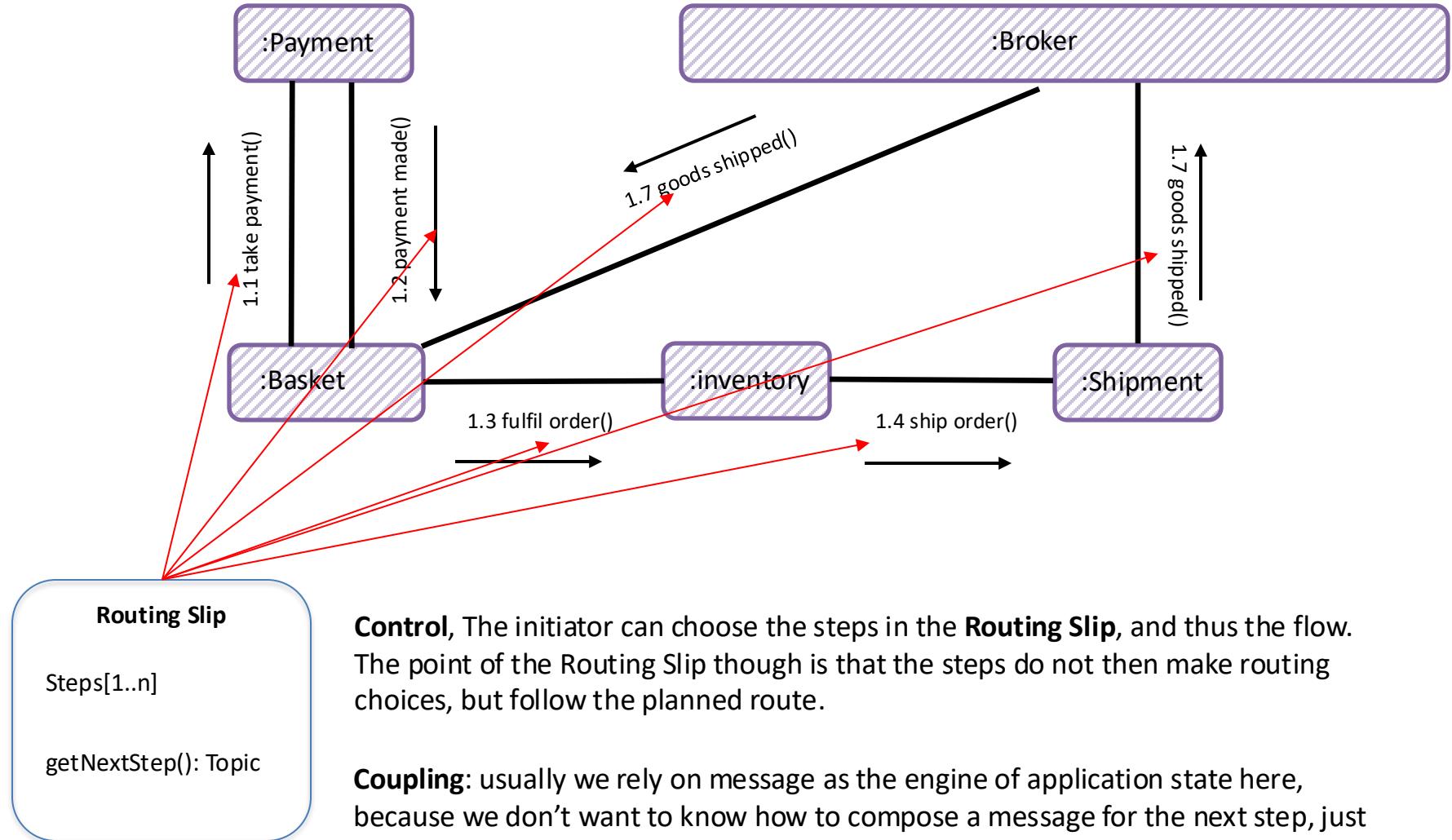


Saga over Orchestration: with either Fault Triggers Message, or Fault Replaces Message, we can propagate errors back up the chain. This provides a simple model for flowing an error – I know how to deal with errors in the operation I invoked, and the chain reverses the operation.

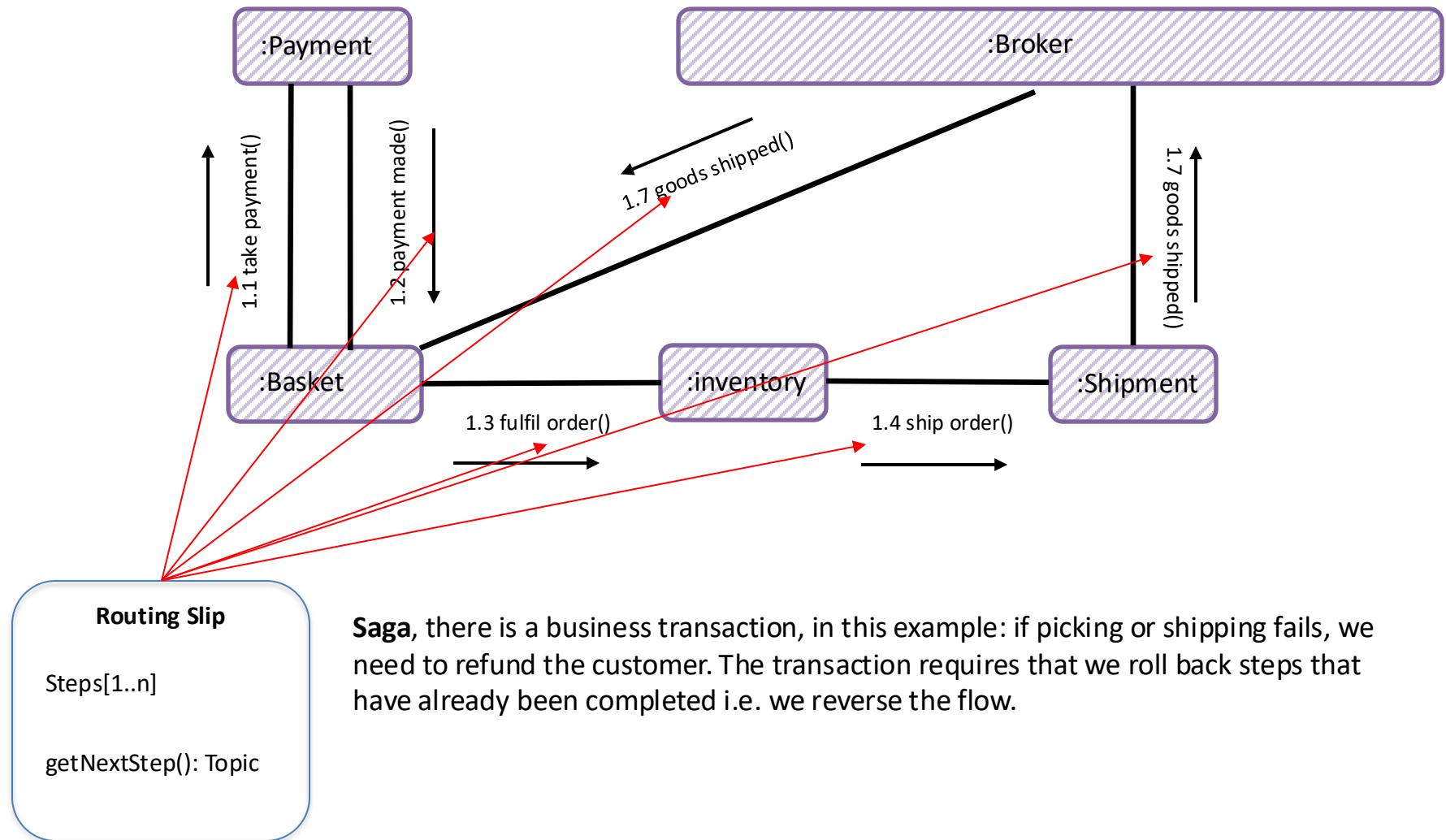
Orchestration – Composing with Messaging (Routing Slip)



Orchestration – Composing with Messaging (Routing Slip)

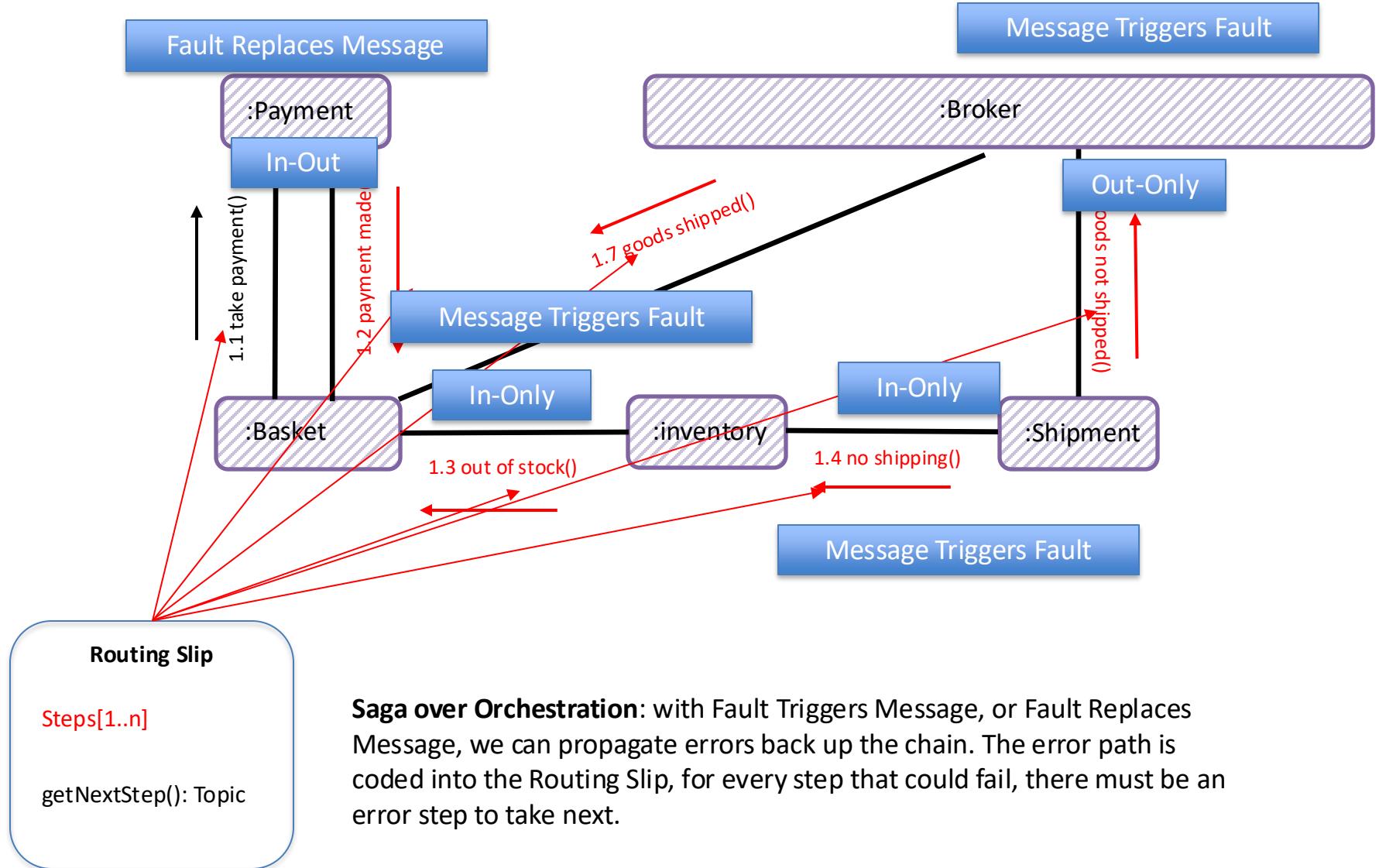


Orchestration – Saga (Routing Slip)

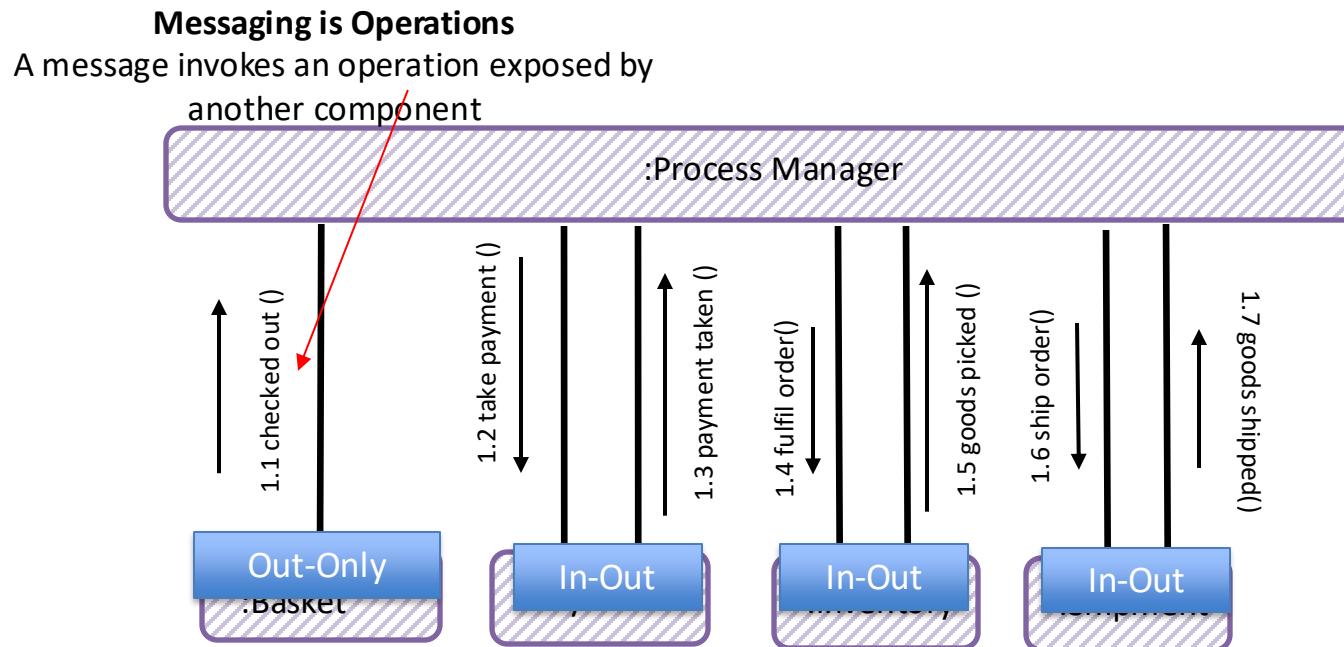


Saga, there is a business transaction, in this example: if picking or shipping fails, we need to refund the customer. The transaction requires that we roll back steps that have already been completed i.e. we reverse the flow.

Orchestration – Saga (Routing Slip)



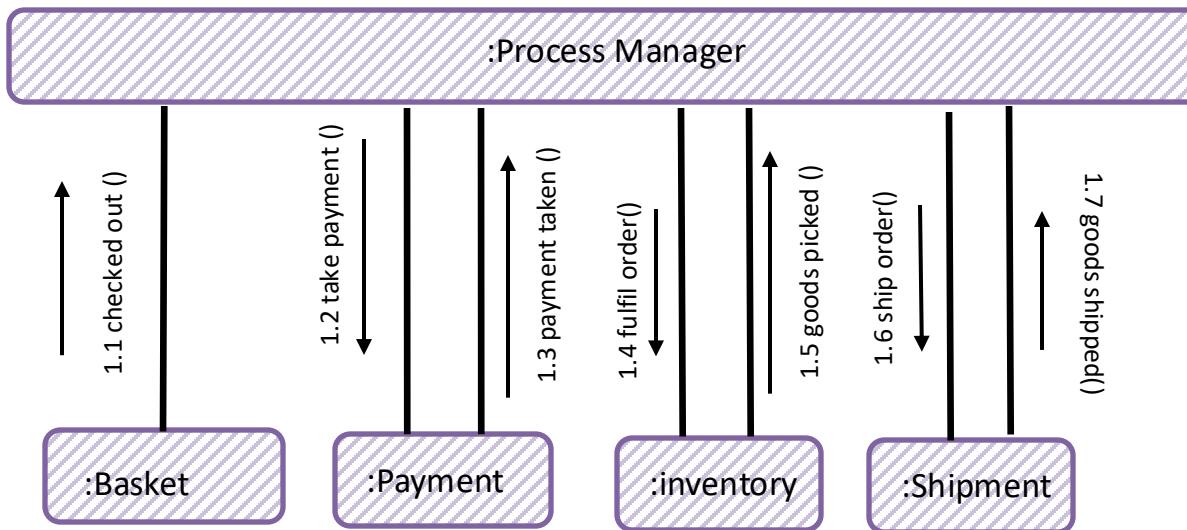
Orchestration – Composing with Messaging (Process Manager)



Process Manager: a component has a dialog with the process manager; knowledge of the workflow belongs to the process manager; each component responds to invocation of its operations

Responsibility and Causality: this is attractive because we have causality – things happen in a known sequence and components clearly have responsibilities for carrying out actions within the flow.

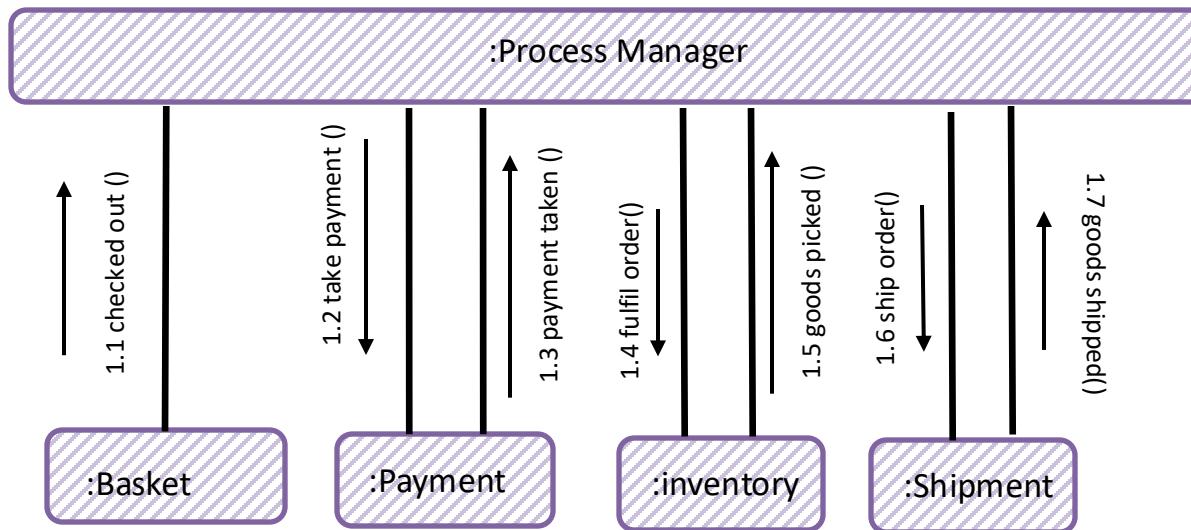
Orchestration – Composing with Messaging (Process Manager)



Coupling vs. Control: behavioural coupling is to the process manager, but changing the workflow requires just altering it in the process manager; central control provides understanding of flow.

But understanding a component's role in the flow, requires consulting the process manager, it is not local to the component.

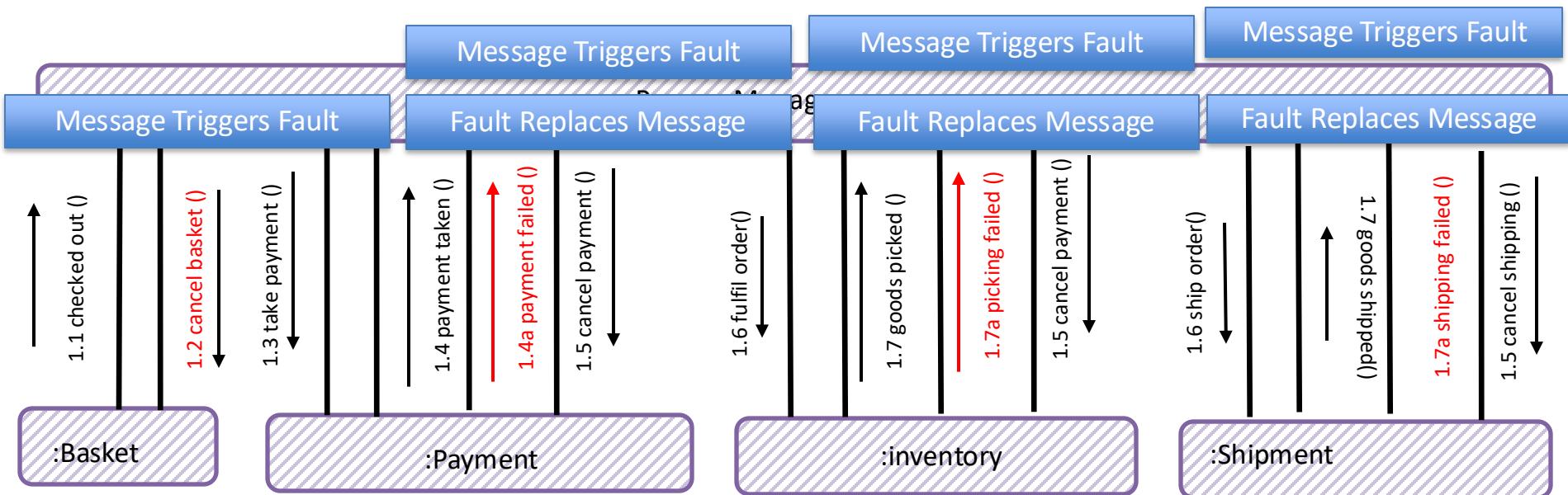
Orchestration – Saga (Process Manager)



Saga over Orchestration: there is a business transaction, in this example: if picking or shipping fails, we need to refund the customer; if shipping fails we need to return the goods to the warehouse. The transaction requires that we roll back steps that have already been completed i.e. we reverse the flow.

We notify the Process Manager, which now runs a workflow to unwind the workflow steps already taken (which it knows).

Orchestration – Saga (Process Manager)



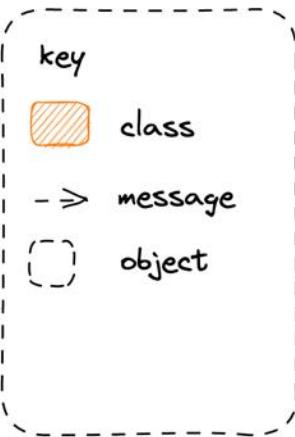
Transactions (Saga): uses Fault Replaces Message, to propagate errors back up to the bus, which triggers the error path on the flow, sending rollback messages. This provides a reliable model for flowing an error – I know that we will reverse the actions taken. It requires a set of operations on each component: raise error, respond to cancellation, but is less complex than choreography for multi-process flows.

Orchestration – Saga, Summary

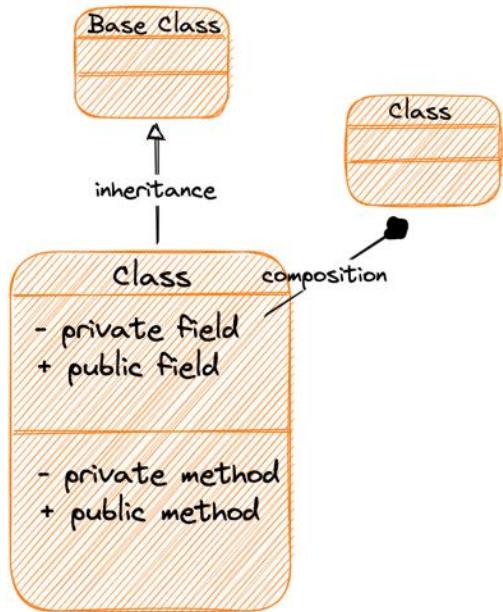
Turn Taking	Routing Slip	Process Manager
Complex Flow	Simple Flow	Complex Flow
Rigid Flow	Dynamic Flow	Dynamic Flow
No central point of failure	No central point of failure	Central point of failure
Distributed	Distributed	Hub-and-Spoke
No central administration or reporting	Central administration but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

Integrating using events

REACTIVE ARCHITECTURES



Object Orientation

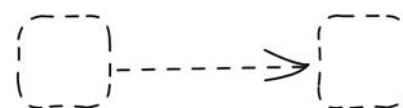


Data and Behaviour

A class couples data and the behaviour that depends upon that data. This allows encapsulation: the data can be hidden and just behaviour exposed.

Dynamic Dispatch

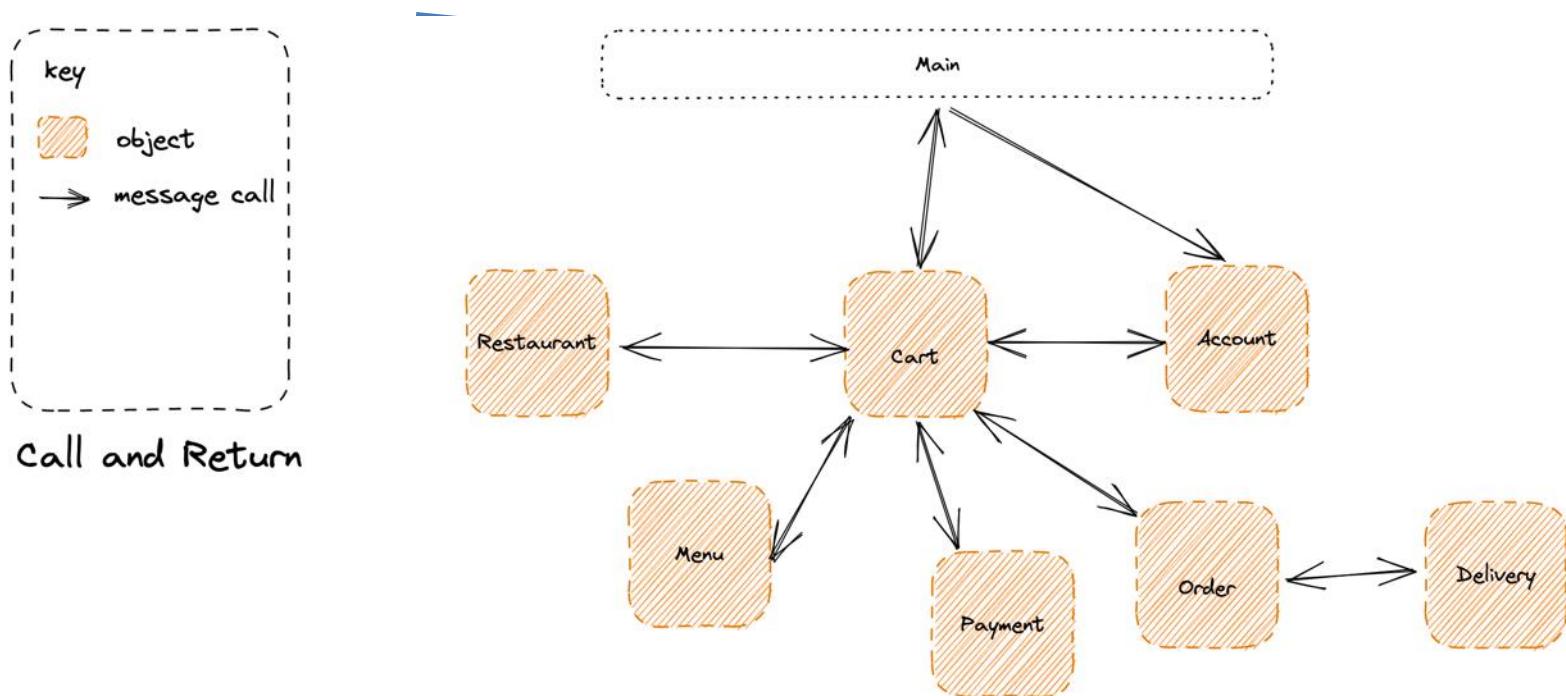
Due to inheritance the method chosen in response to a message may come from the base class of an object



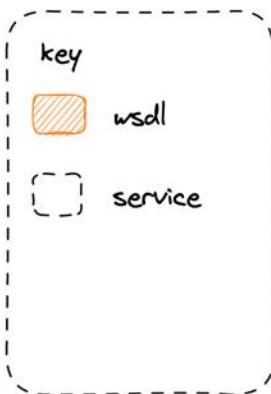
Message Passing

Objects communicate by message passing. A message is the method to call, and the parameters to that method.

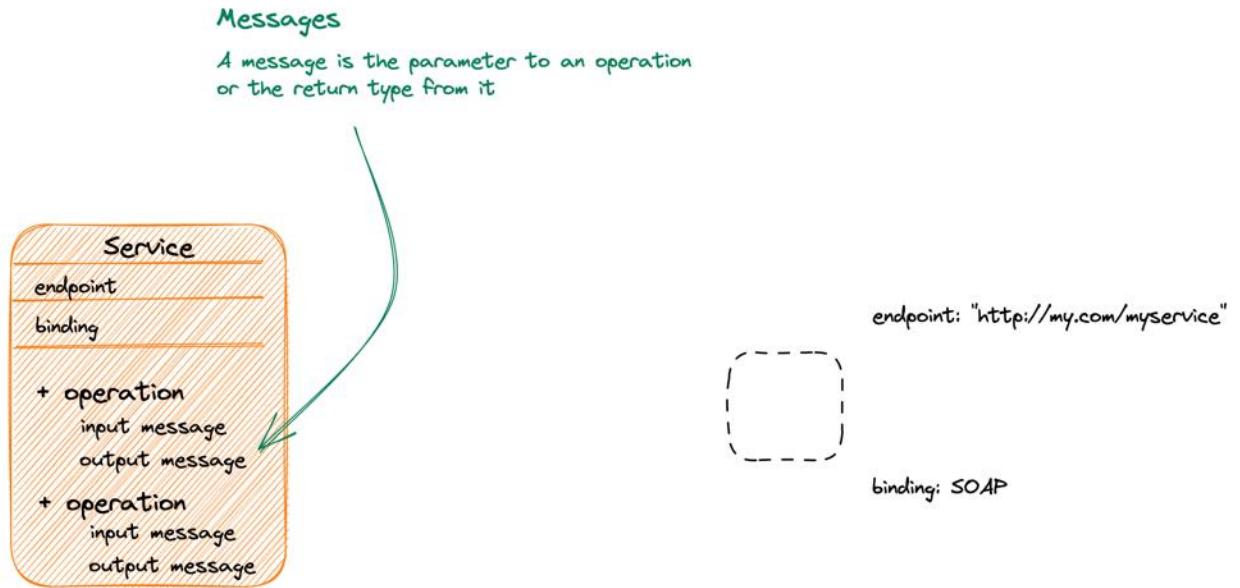
Call and Return: The main method represents our entry point and it uses message passing to invoke objects, which in turn invoke other objects, and return to their caller



God Object: A danger here is that we end up with a "god" object, such as Cart here, that controls all the other objects. This is a high-degree of behavioral coupling



Service Orientation

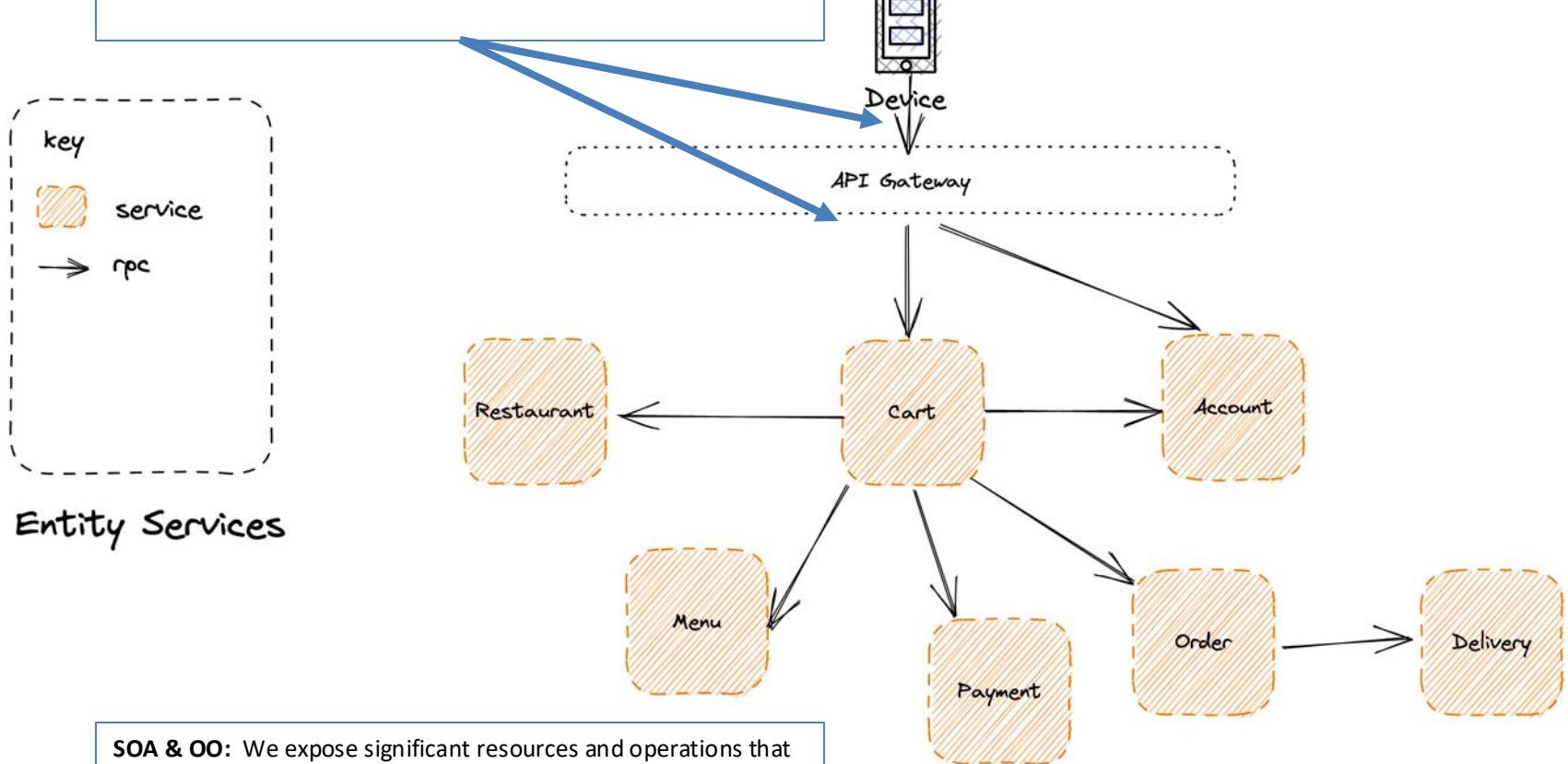


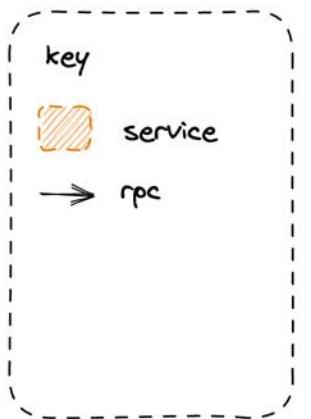
Data and Behaviour

A service couples data and the behaviour that depends upon that data. This allows encapsulation the data can be hidden and just behaviour exposed

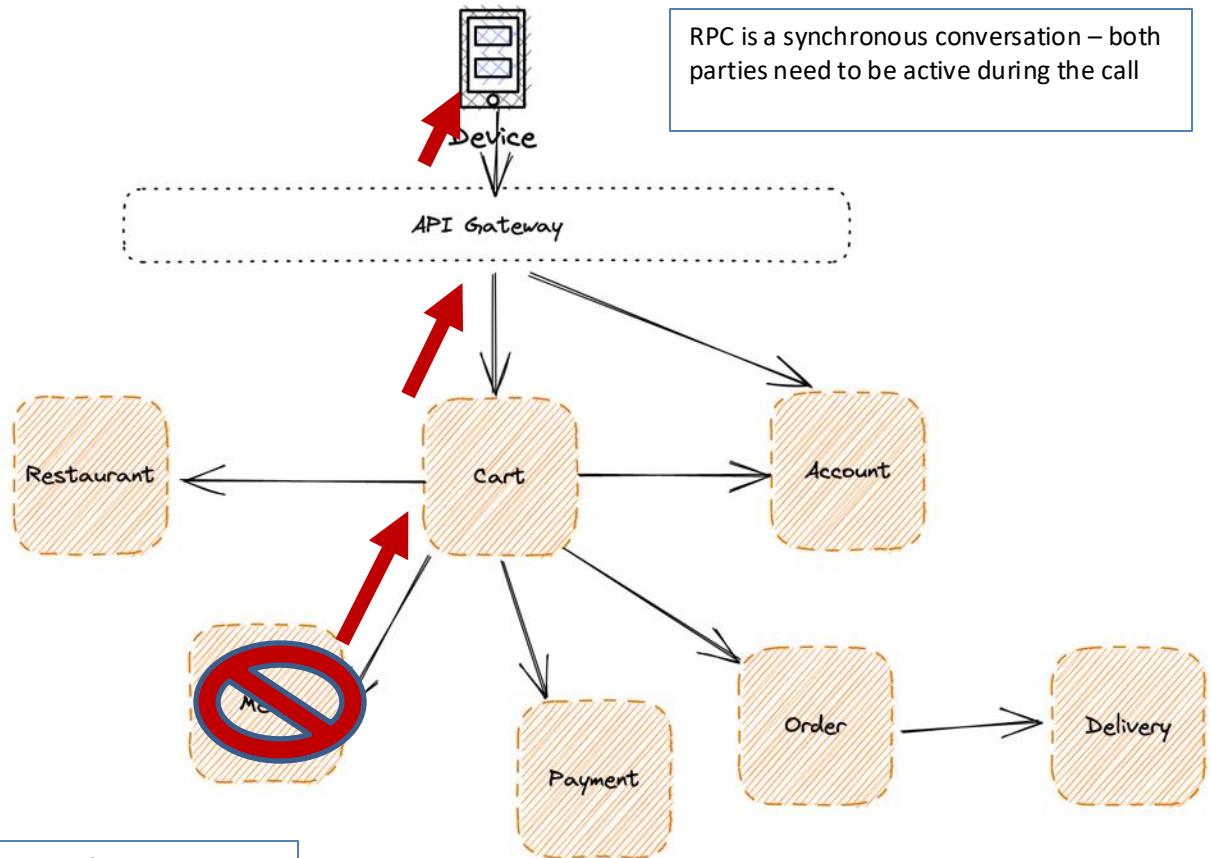
SOA & OO: SOA creates OO-like components, they encapsulate their data and expose behavior that is coupled to that data. This is the Web Services approach to services.

Feature Envy: Because domain logic needs to co-ordinate across these resources, it ends up either on the client, the API Gateway, or the Cart service and not in individual services. An *anti-pattern*.

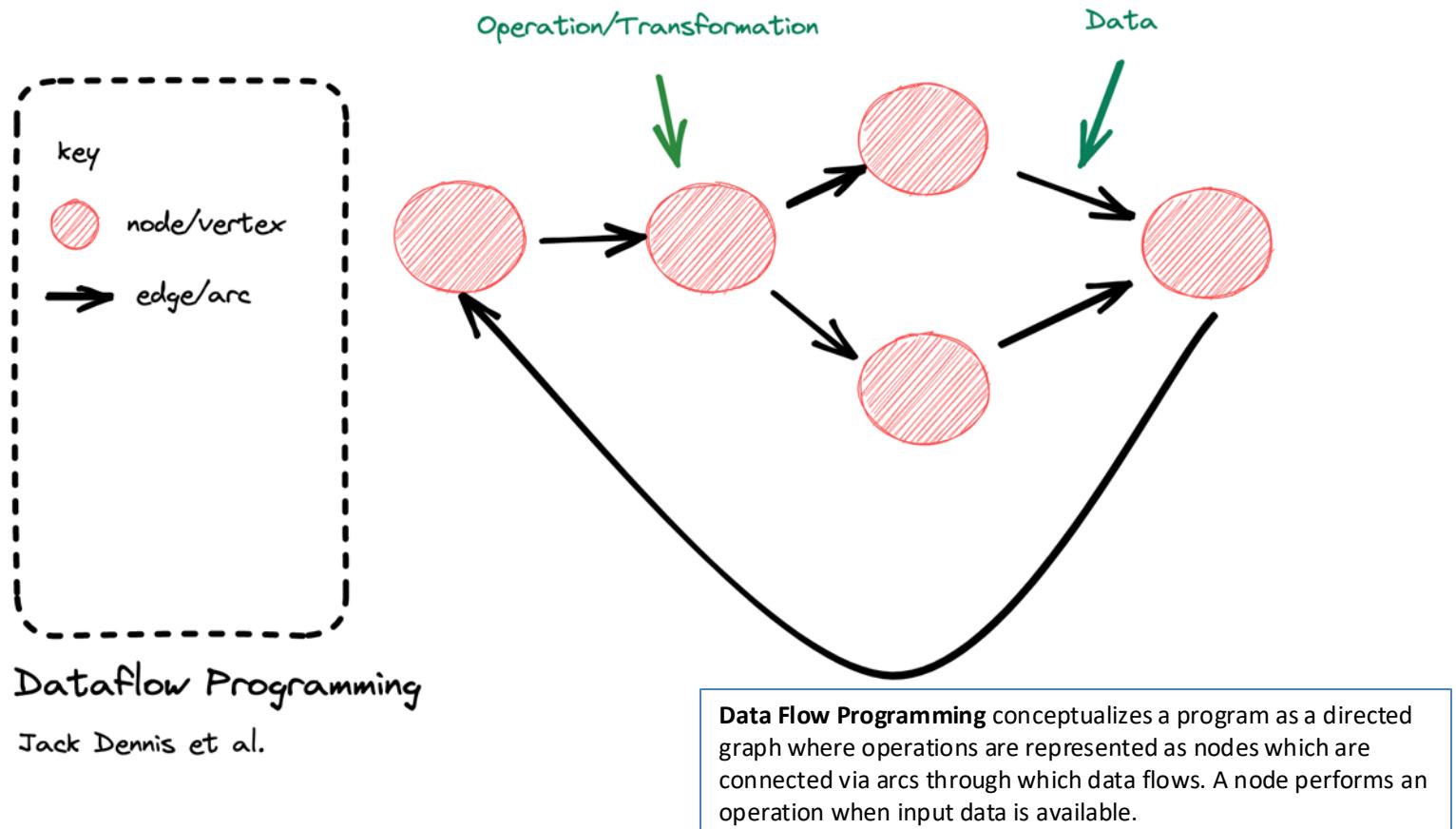


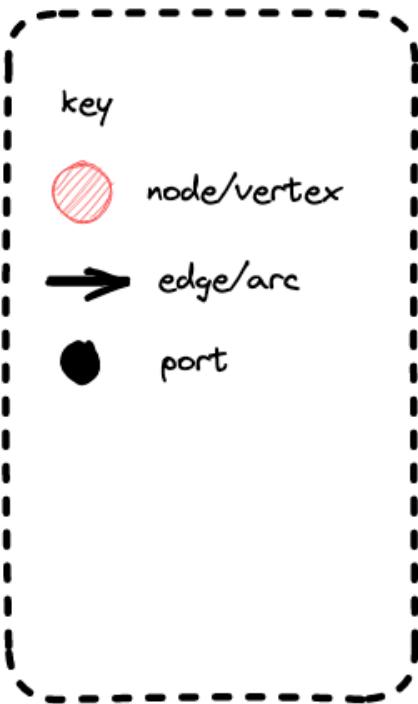


Entity Services



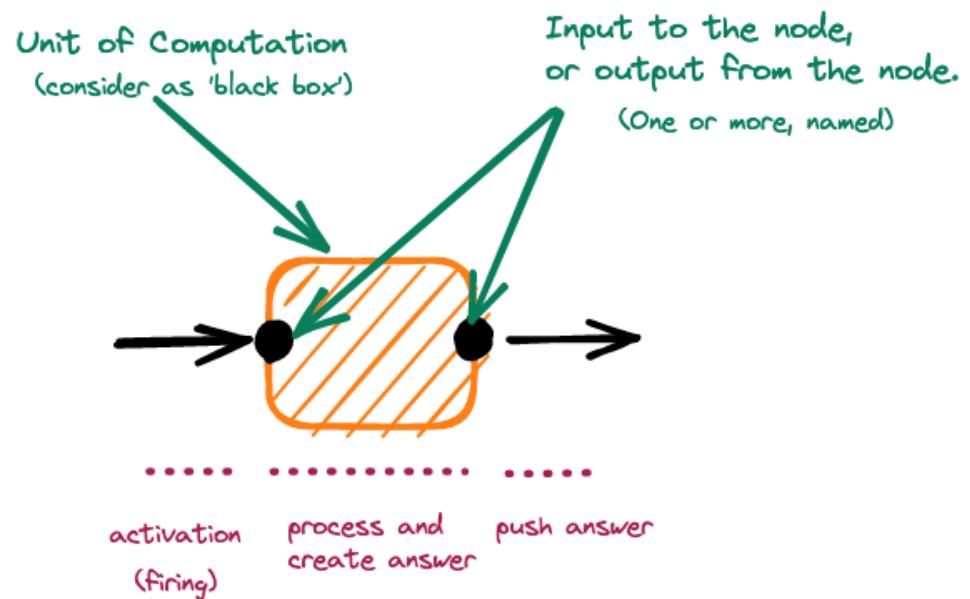
Faults Propagate: With a synchronous service, a fault propagates





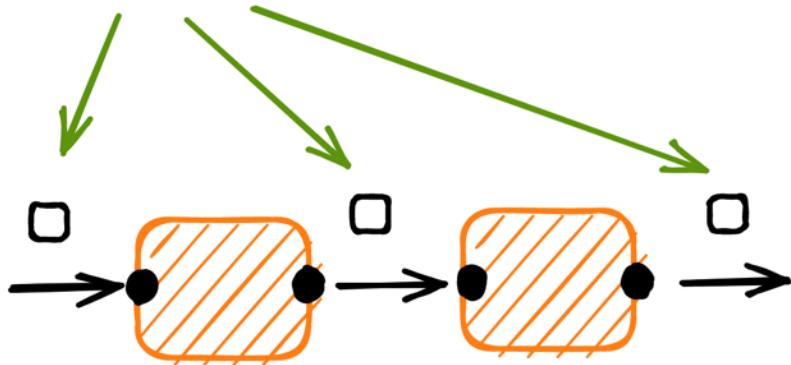
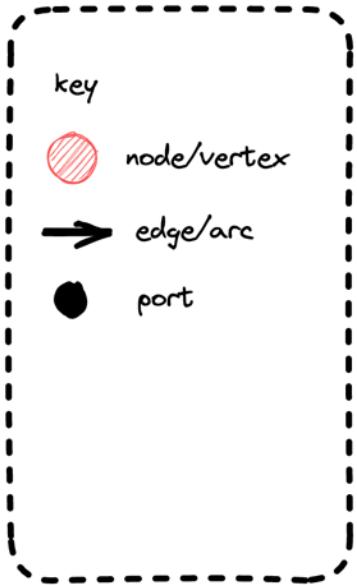
Dataflow Programming

Jack Dennis et al.



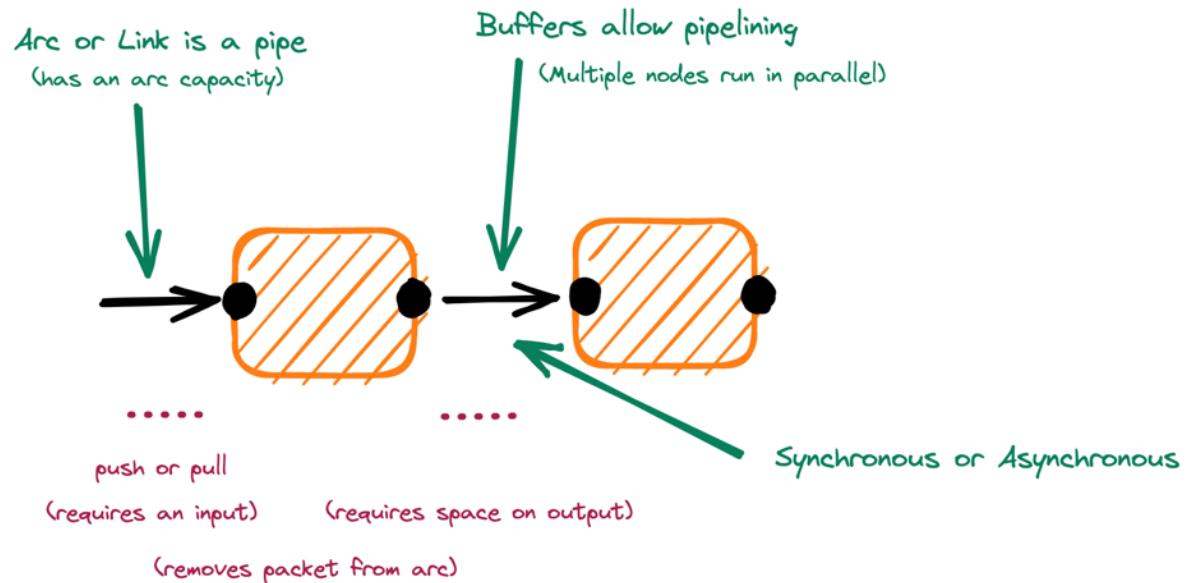
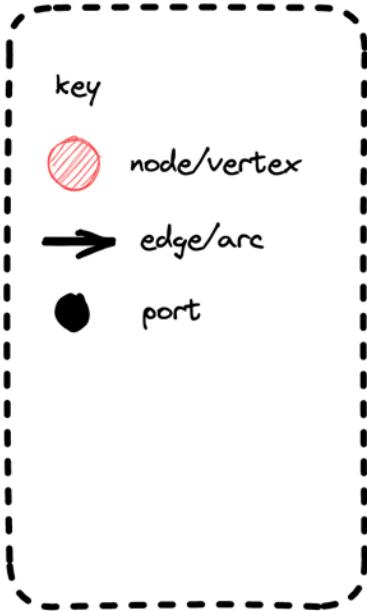
Data Packet

primitive/compound, structured/unstructured values, and even include other data packets



Dataflow Programming

Jack Dennis et al.



Dataflow Programming

Jack Dennis et al.

Node Lifetime: In 'classic' dataflow programming the lifetime of a node is from activation when there is work to do (push) or work is requested (pull) until it has pushed the answer.

Capacity: In dataflow programming we do not activate a node to read from the input, if there is no space on the output. (Infinite capacity links exist only in theory). This creates **backpressure**.

The Reactive Manifesto

Published in September 2014

Author(s): Jonas Bonér (Erik Meijer, Martin Odersky, Greg Young, Martin Thompson, Roland Kuhn, James Ward and Guillaume Bort)

Defines an architectural style: **Reactive Applications**

Write applications that:

react to events: their event-driven nature enables the following qualities

react to load: focus on scalability rather than single-user performance

react to failure: resilient systems with the ability to recover at all levels

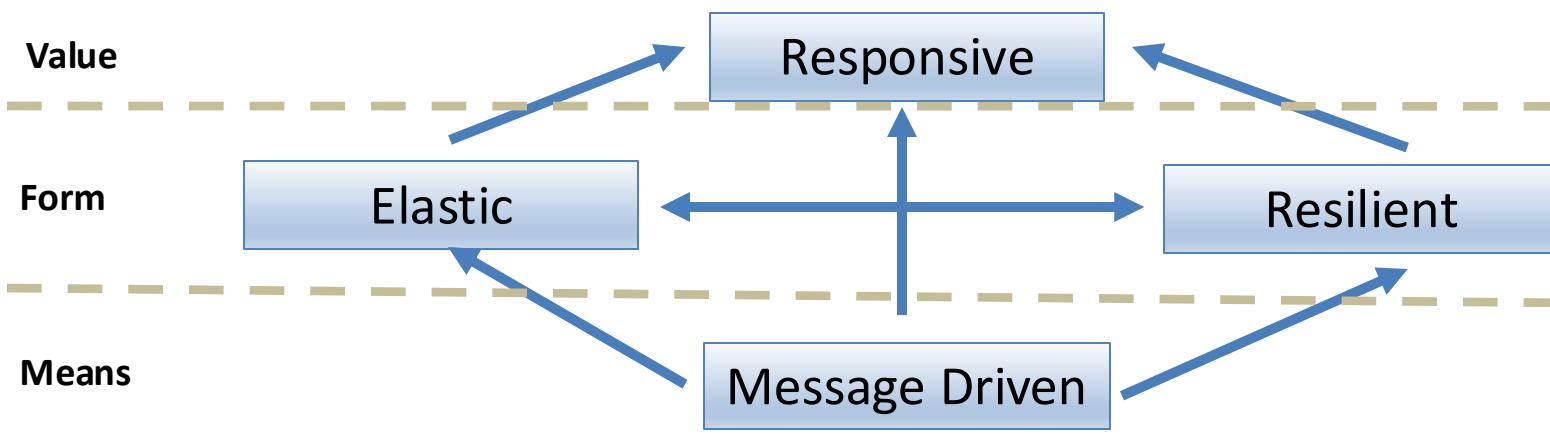
react to users: combine the above for an interactive user experience

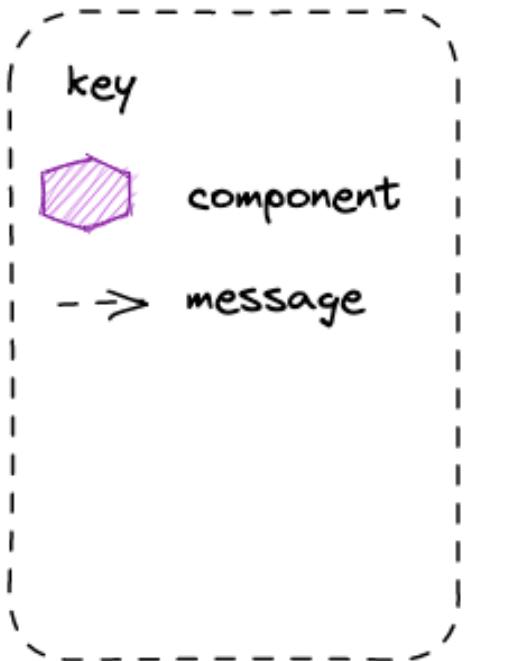
Responsive: The system responds in a timely manner.

Resilient: The system stays responsive in the presence of failure.

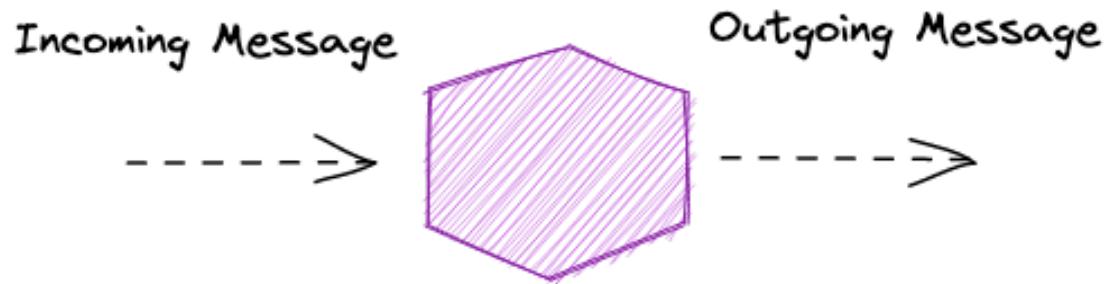
Elastic: The system stays responsive under varying workload.

Message Driven: Rely on asynchronous message passing

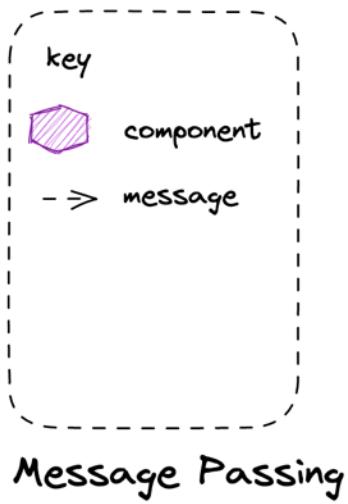




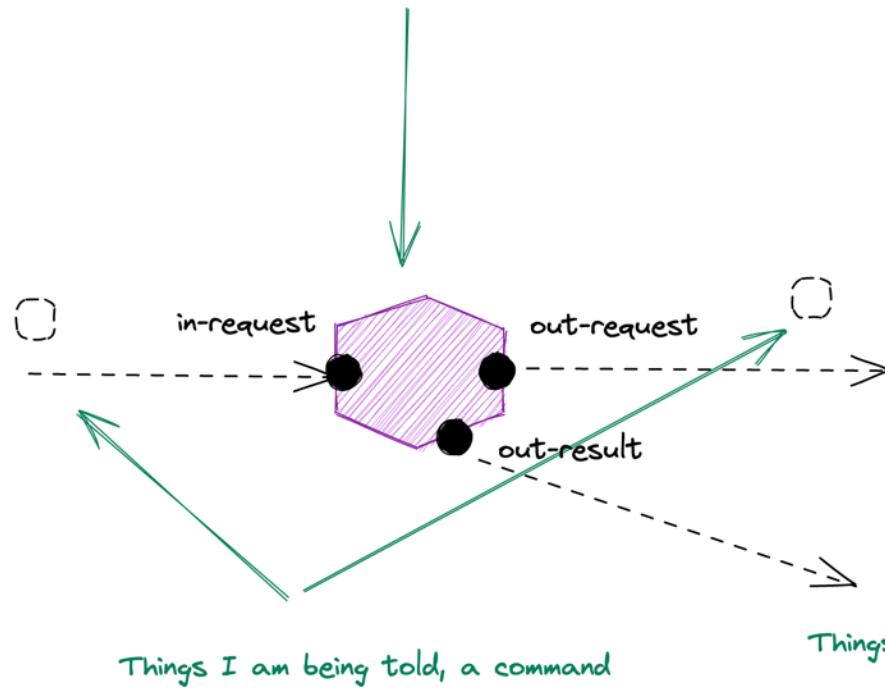
Message Passing

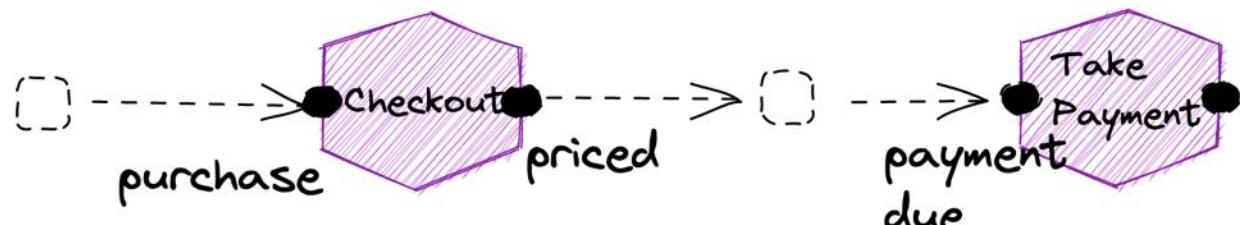
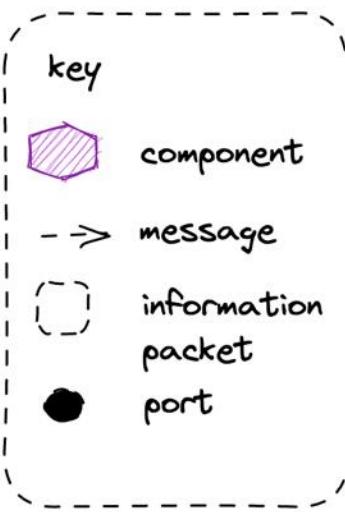


Message Passing is an asynchronous method of communication – both parties do not have to be simultaneously present for communication to occur, instead mail is delivered to ‘mailbox’ of some form for later retrieval.



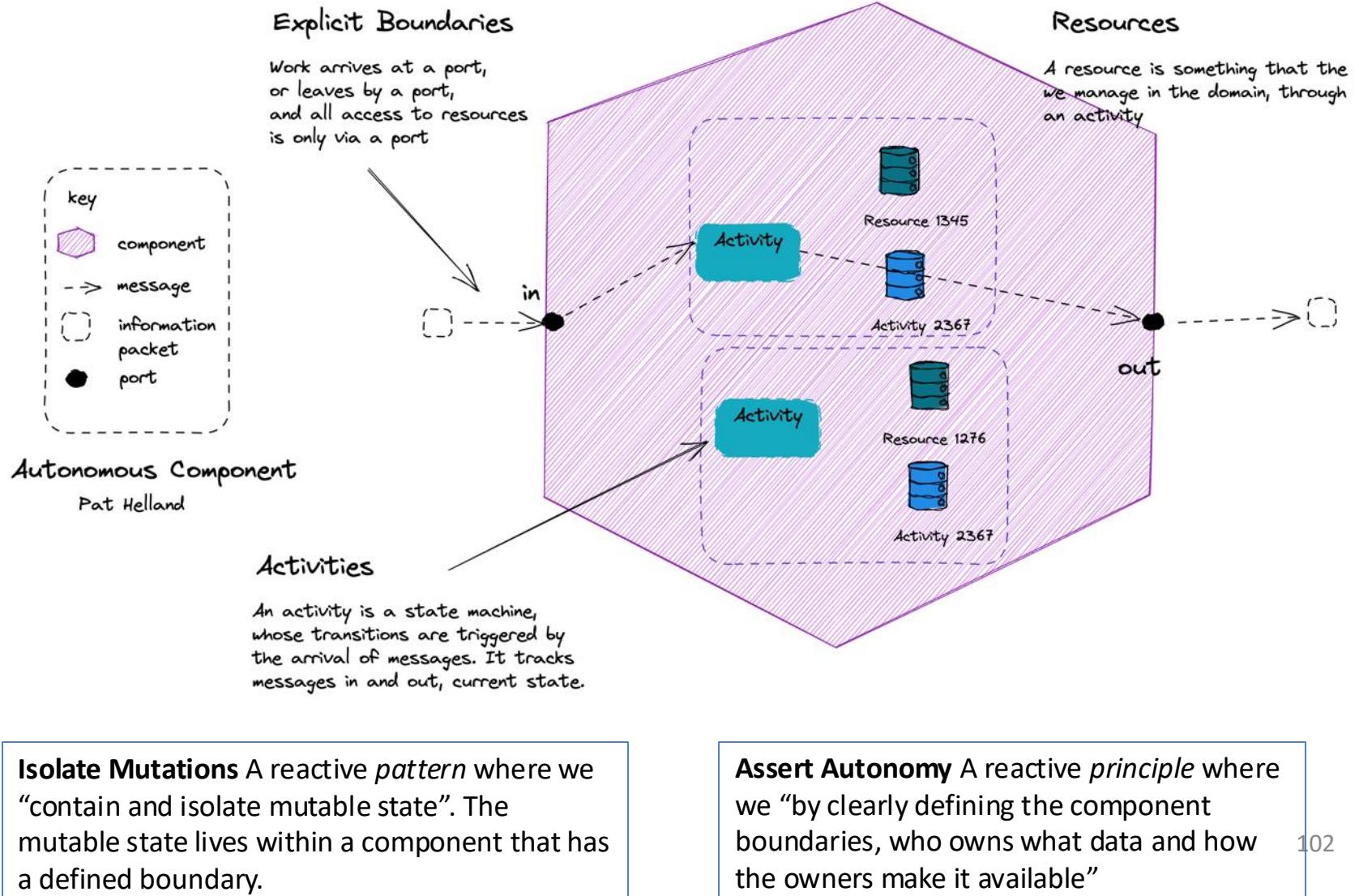
Service should focus on a transformation to the data
Usually a verb + noun combination
Place Order or Onboard Restaurant

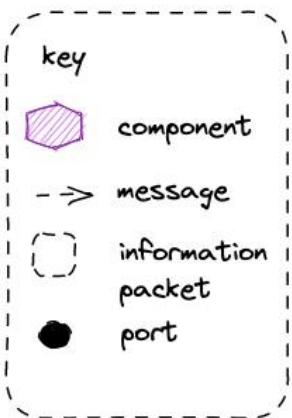




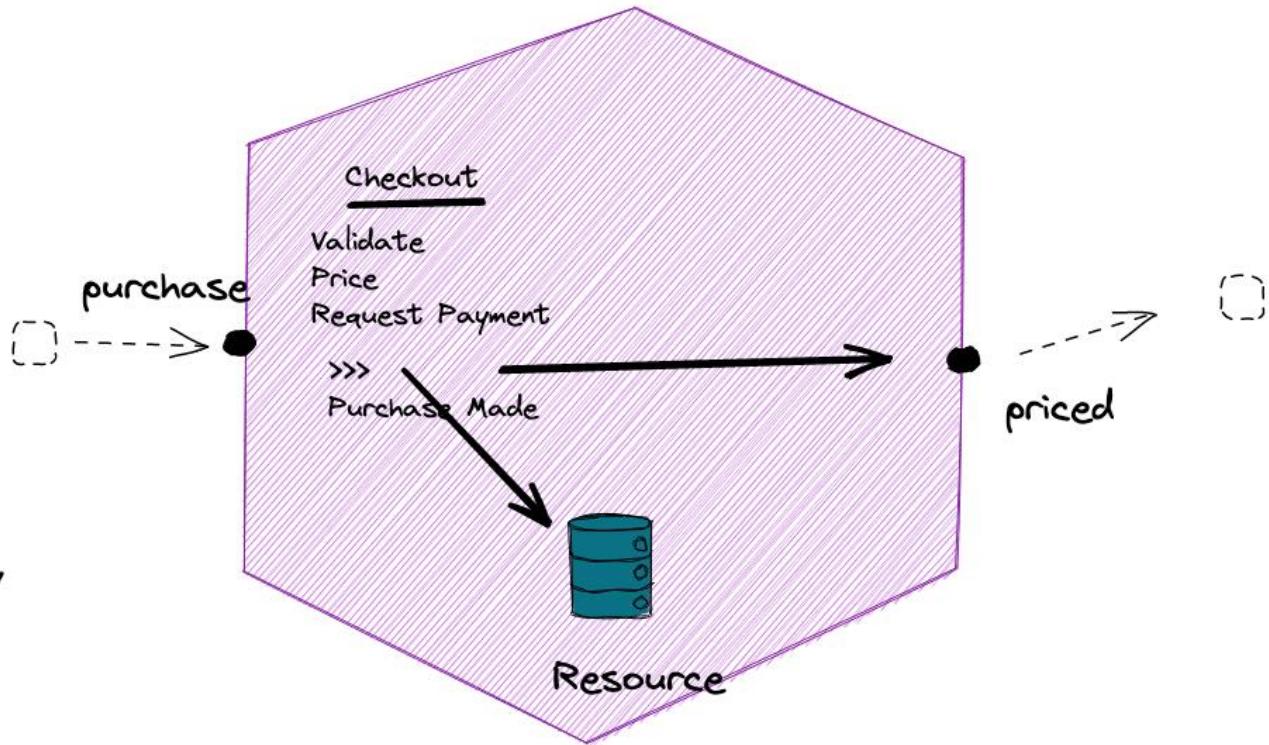
Coordinate Dataflow A reactive *principle* where we “orchestrate a continuous steady flow of information” focusing on division by behavior, not structure

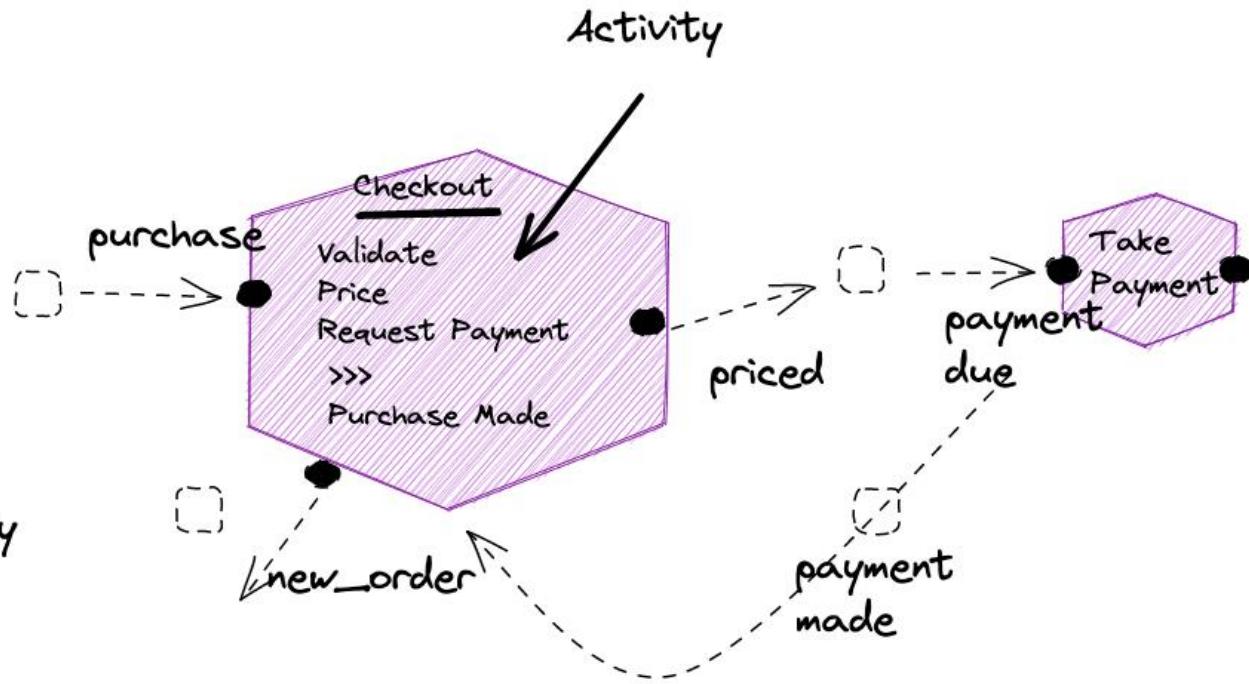
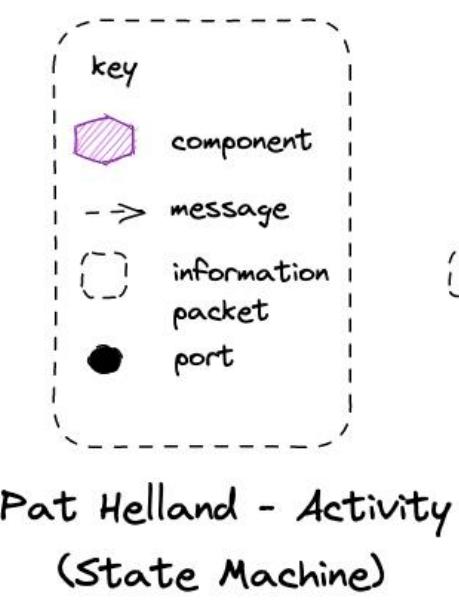
Partitioning Partition to take advantage of parallelism in the system, tasks that could be done concurrently with each other

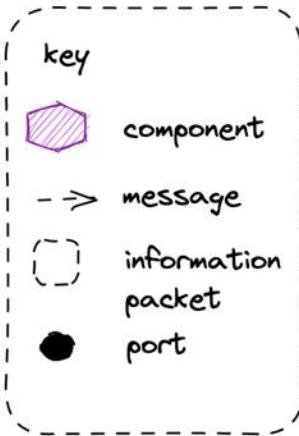




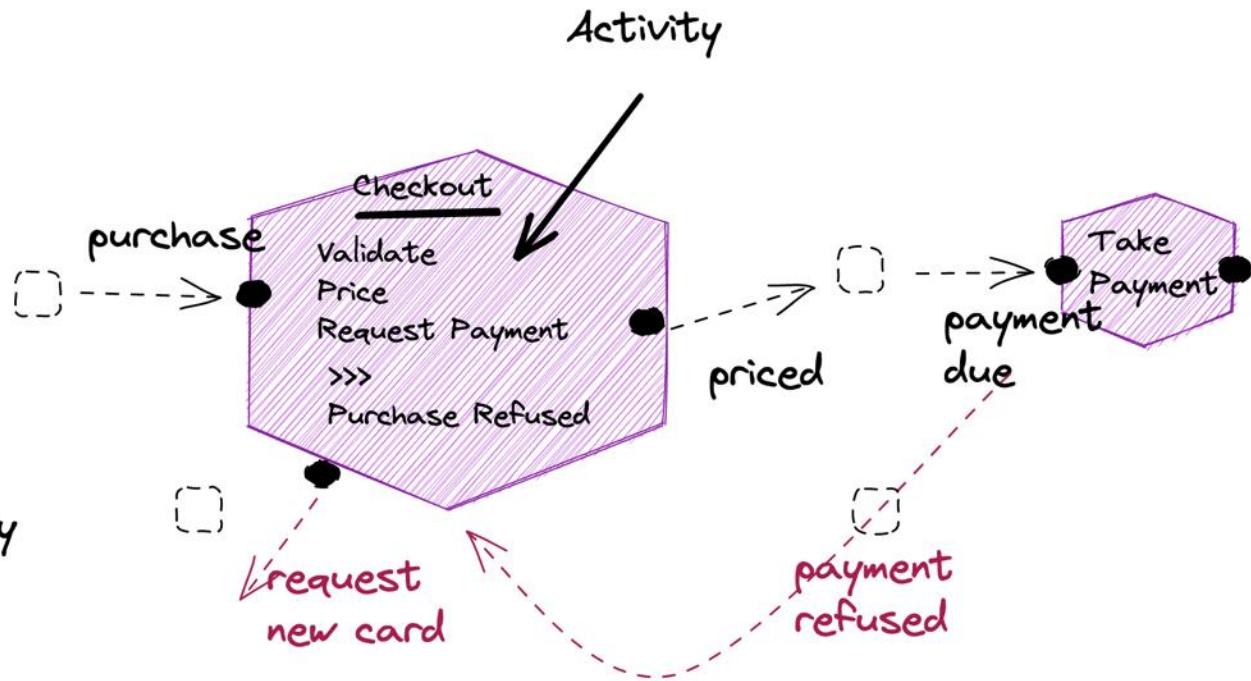
Pat Helland - Activity
(State Machine)

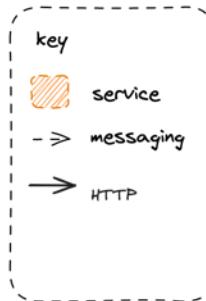




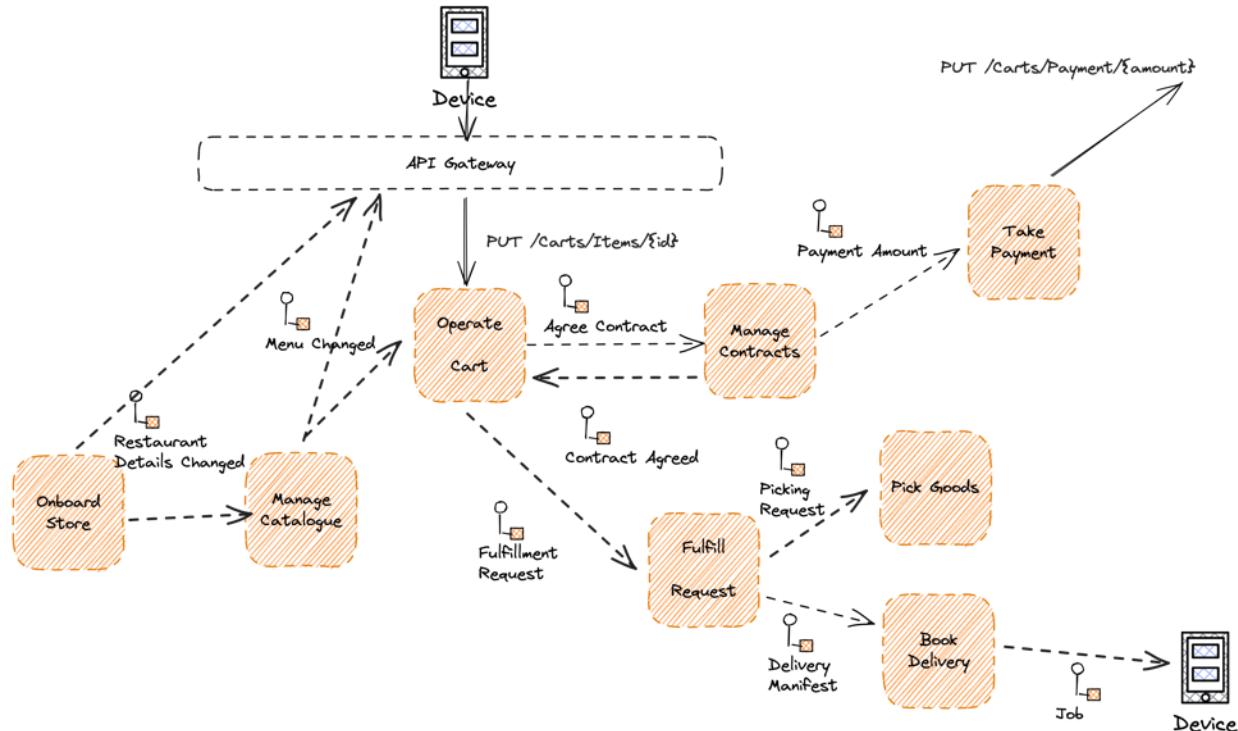


Pat Helland - Activity
(State Machine)





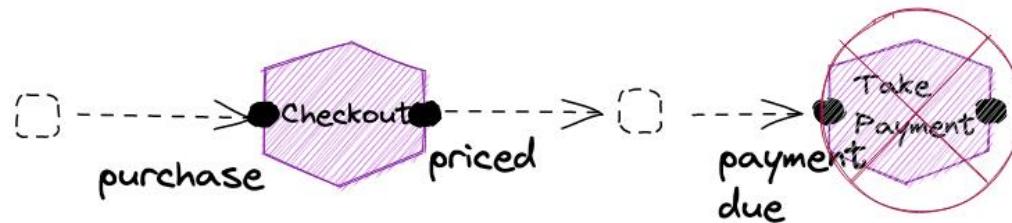
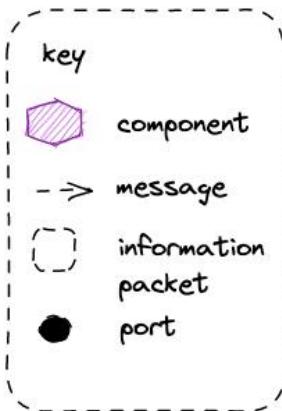
Process Services



Focus on Behavior not Data

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

- Michael Nygard

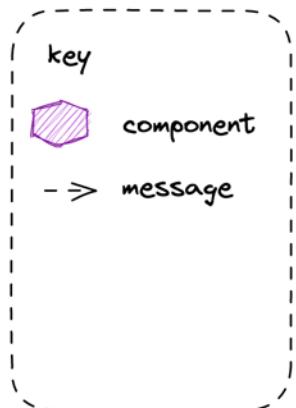


Work Queues on a Fault

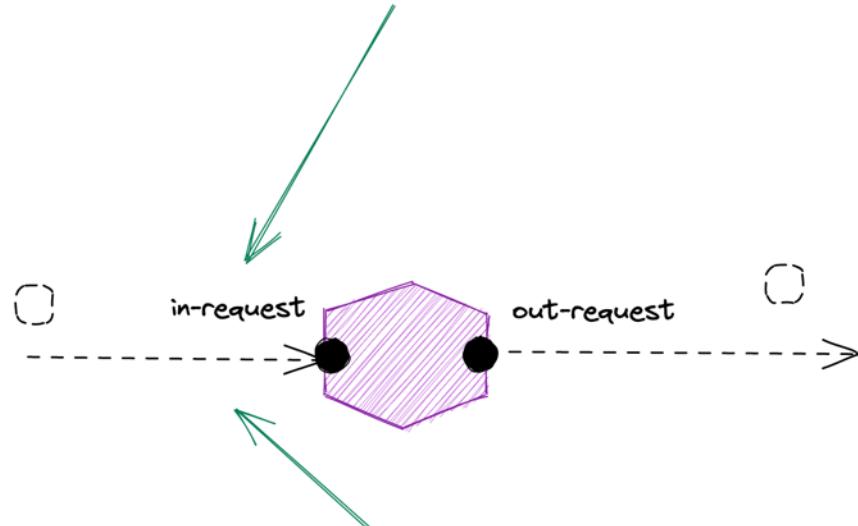
If Take Payment is not available, we can queue requests. When it starts up, Take Payment can process those requests and respond as normal.

Bulkheads: In an asynchronous conversation a fault does not propagate back up the chain – we have a bulkhead that protects us against failure

Push => open socket, middleware calls us as messages arrive
Pull => We poll for messages



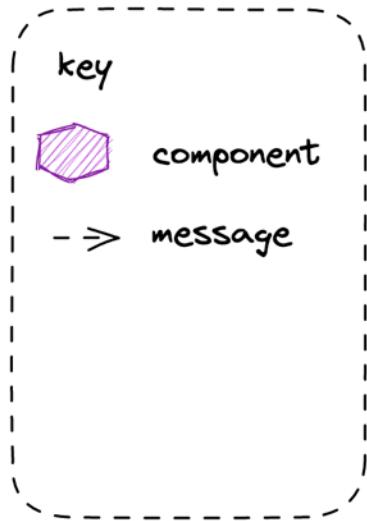
Message Passing



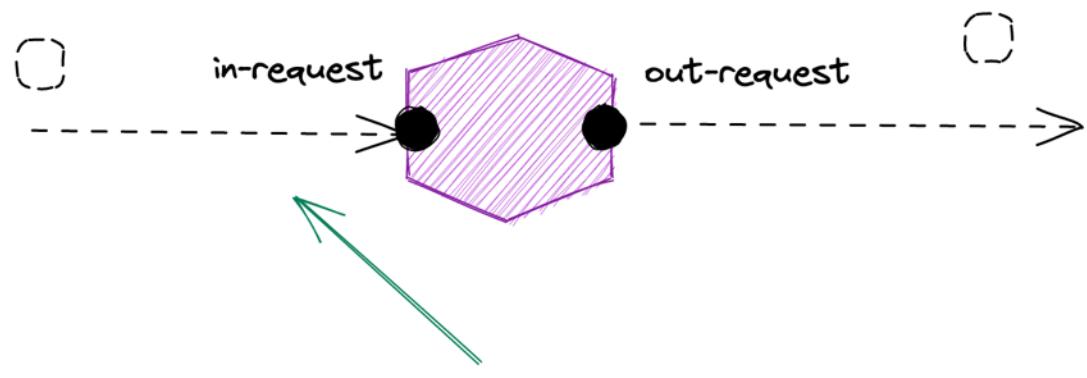
Backpressure

Push => Need to control how many messages are "in flight" with us
Pull => We control the rate at which we poll

Blocking Retry: Blocking Retry – when we keep trying to process a message, such as when we cannot connect to a DB and retry, creates backpressure as we slow consumption.

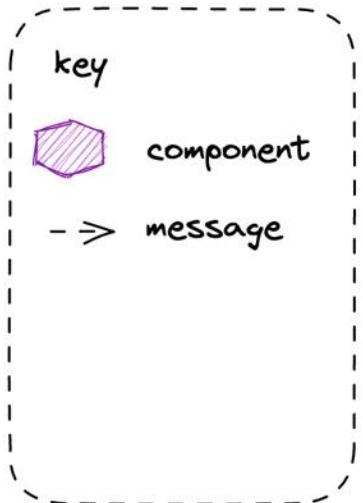


Message Passing

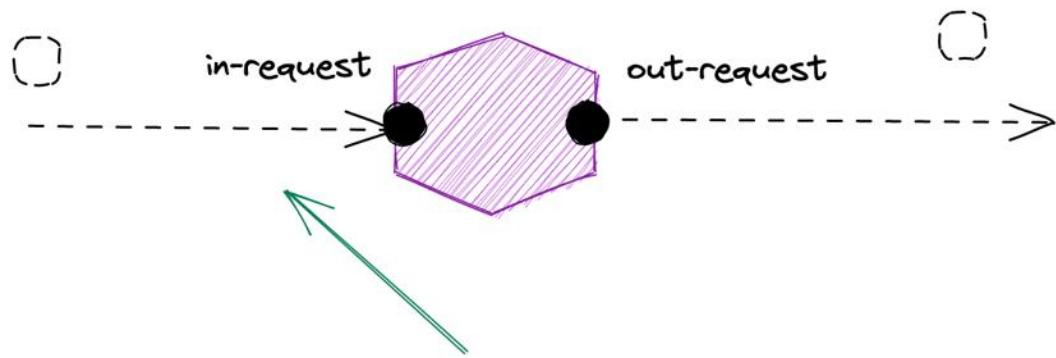


Load Shedding

Sometimes we may prefer to simply drop messages or shed load, when there is a fault.



Message Passing



Circuit Breaker

We may want to stop consuming, due to a fault.
 We can periodically let a message through to see if the fault has cleared.

Let it Crash Pattern

(Candea & Fox "Crash-Only Software")

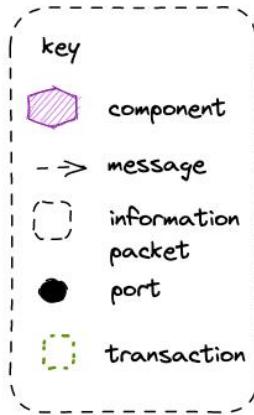
Transient or rare failures are hard to detect

Recovery from a fault may be a complex problem

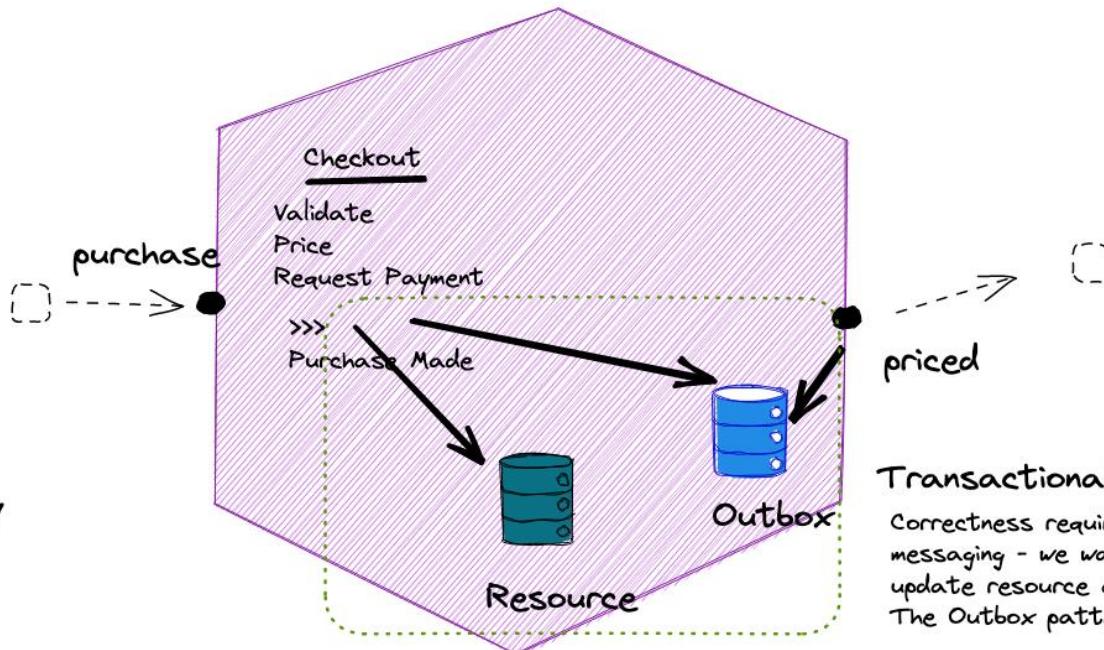
Crash and Restart =>

Start Up is Recovery

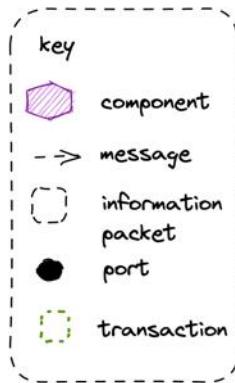
Shut Down is Clean



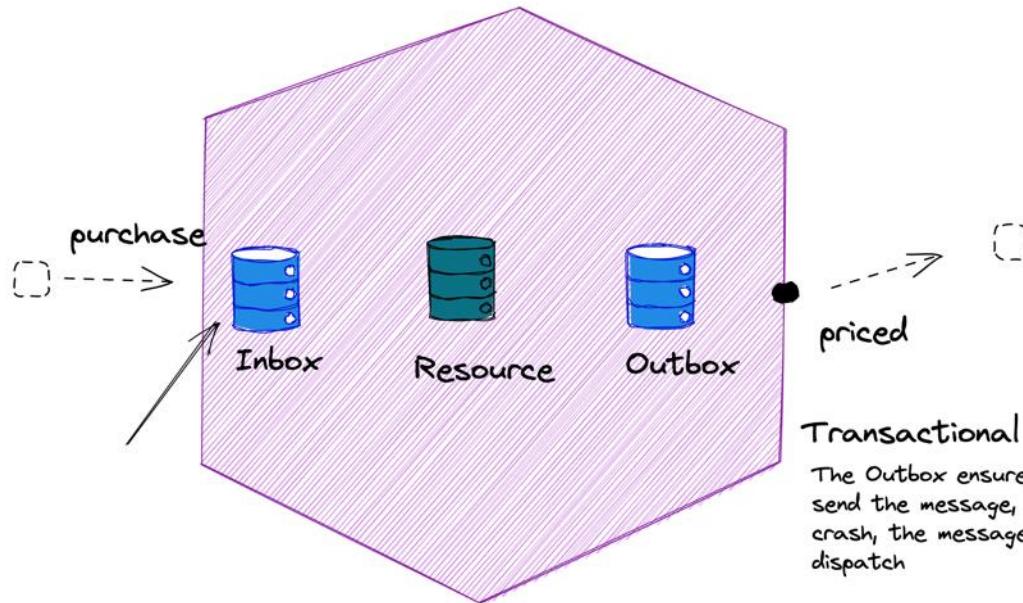
Pat Helland - Activity
(State Machine)



Transactional Messaging
Correctness requires transactional messaging - we want to ensure we update resource and send message
The Outbox pattern offers this

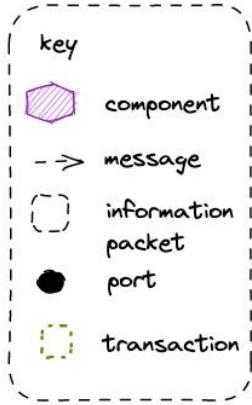


Let It Crash

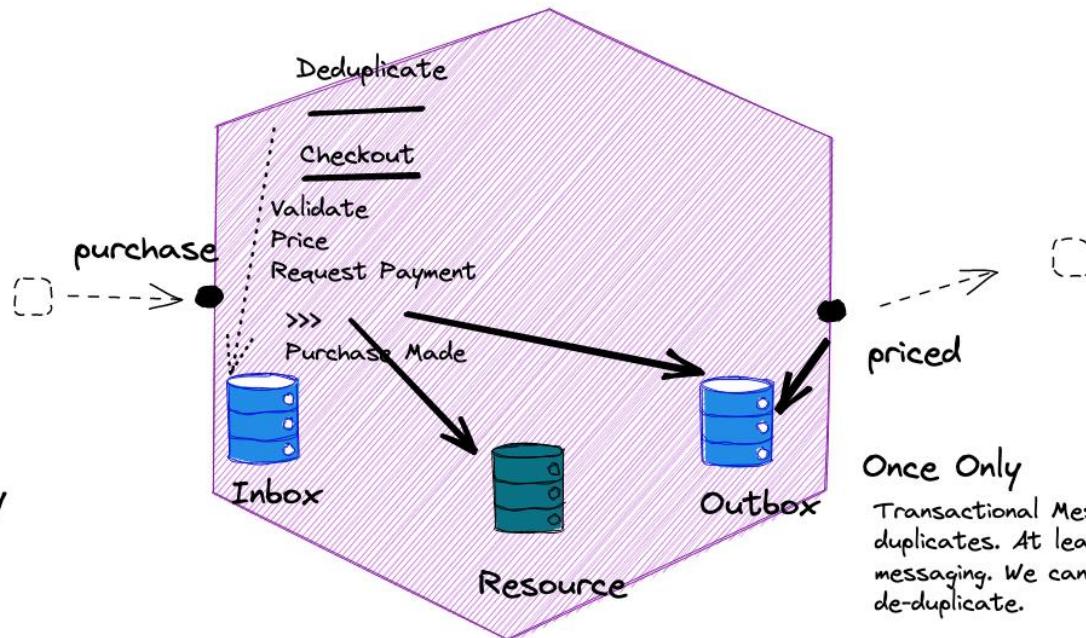


Transactional Messaging

The Outbox ensures that we will eventually send the message, even if our component should crash, the message is in the Outbox for later dispatch

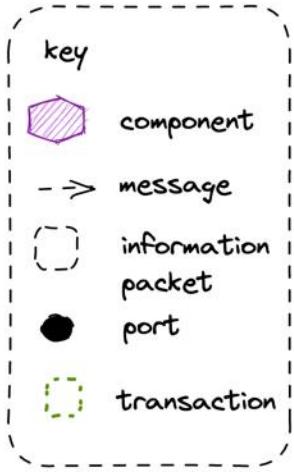


Pat Helland - Activity
(State Machine)



Once Only

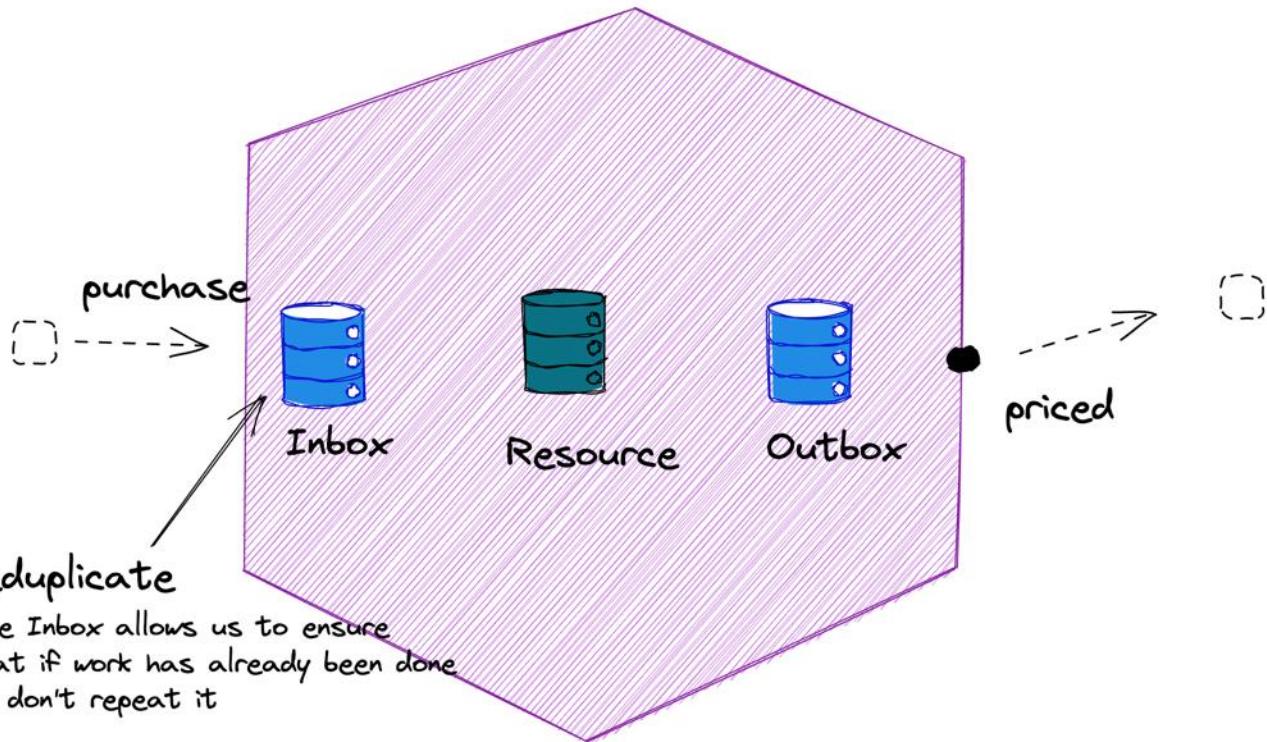
Transactional Messaging will send us duplicates. At least once, guaranteed messaging. We can use an Inbox to de-duplicate.



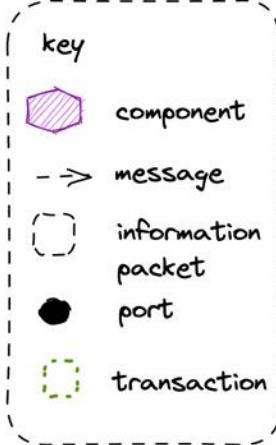
Let It Crash

Deduplicate

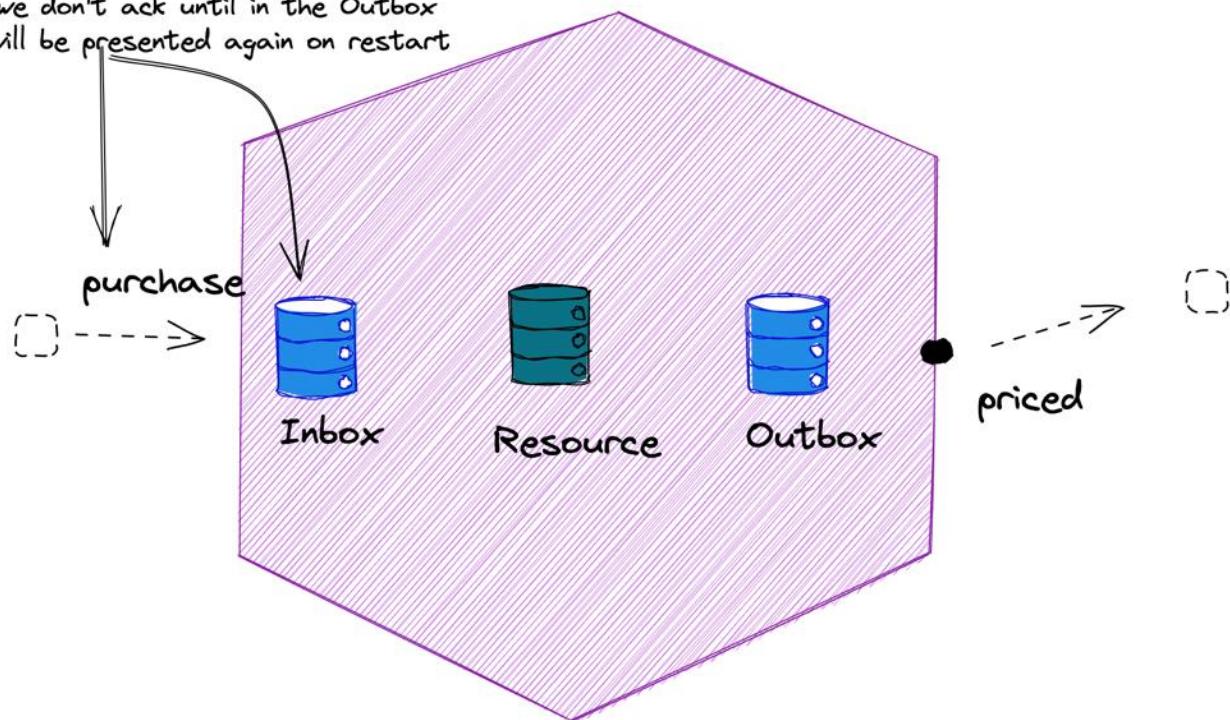
The Inbox allows us to ensure
that if work has already been done
we don't repeat it



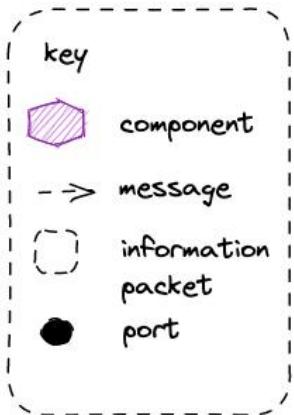
Don't Ack until Done



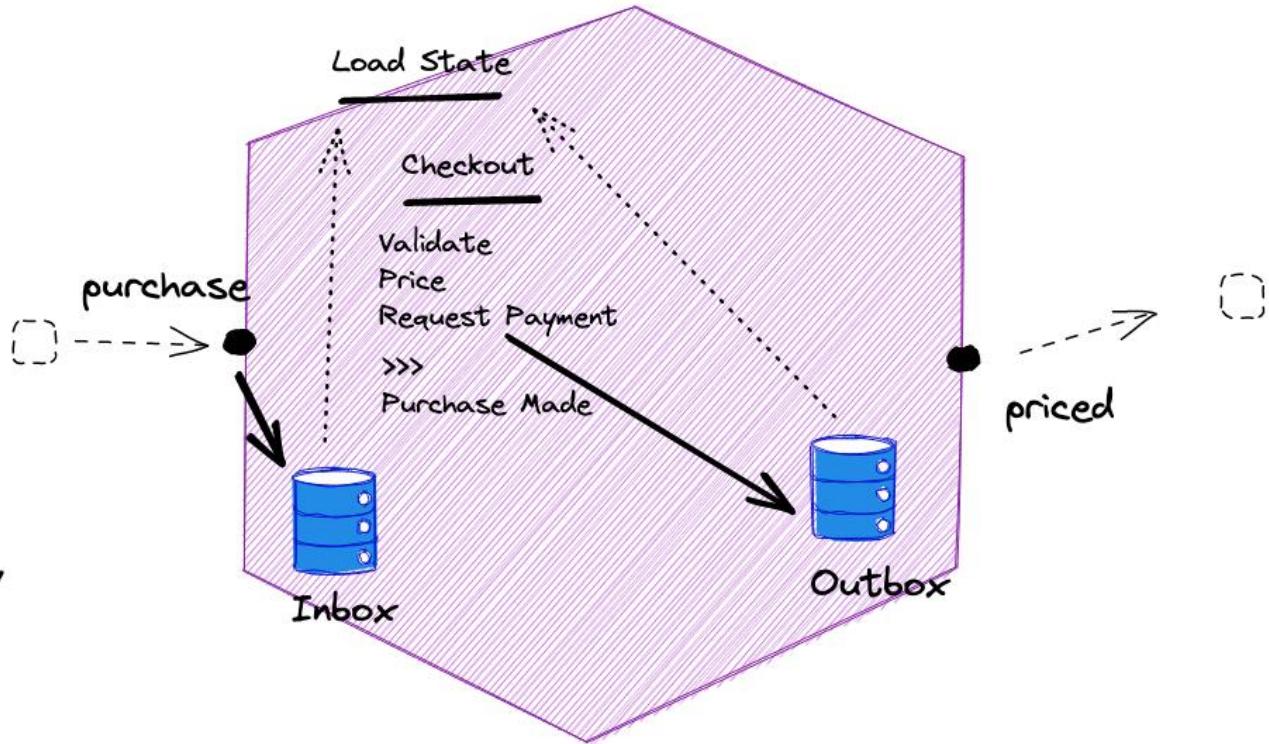
A message is a queue of work
If we don't ack until in the Outbox
it will be presented again on restart



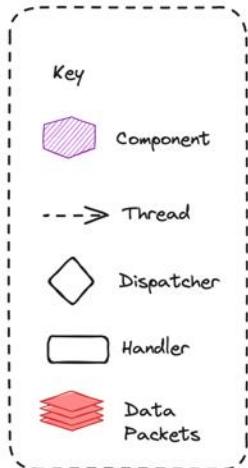
Let It Crash



Pat Helland - Activity (State Machine)

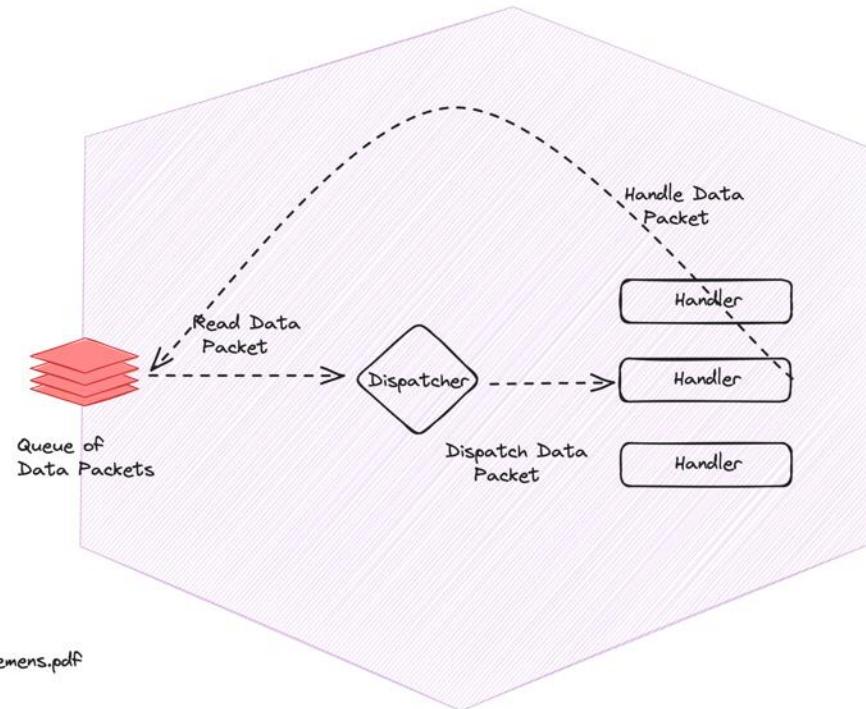


It is possible to calculate activity state by comparing the inbox and outbox to determine where a flow reached, in the event of failure, over storing current state.



Reactor Pattern

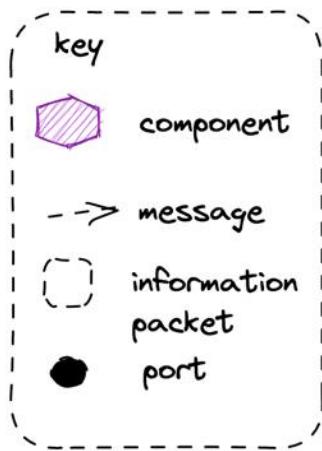
Douglas Schmidt
<http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf>



Single Threaded

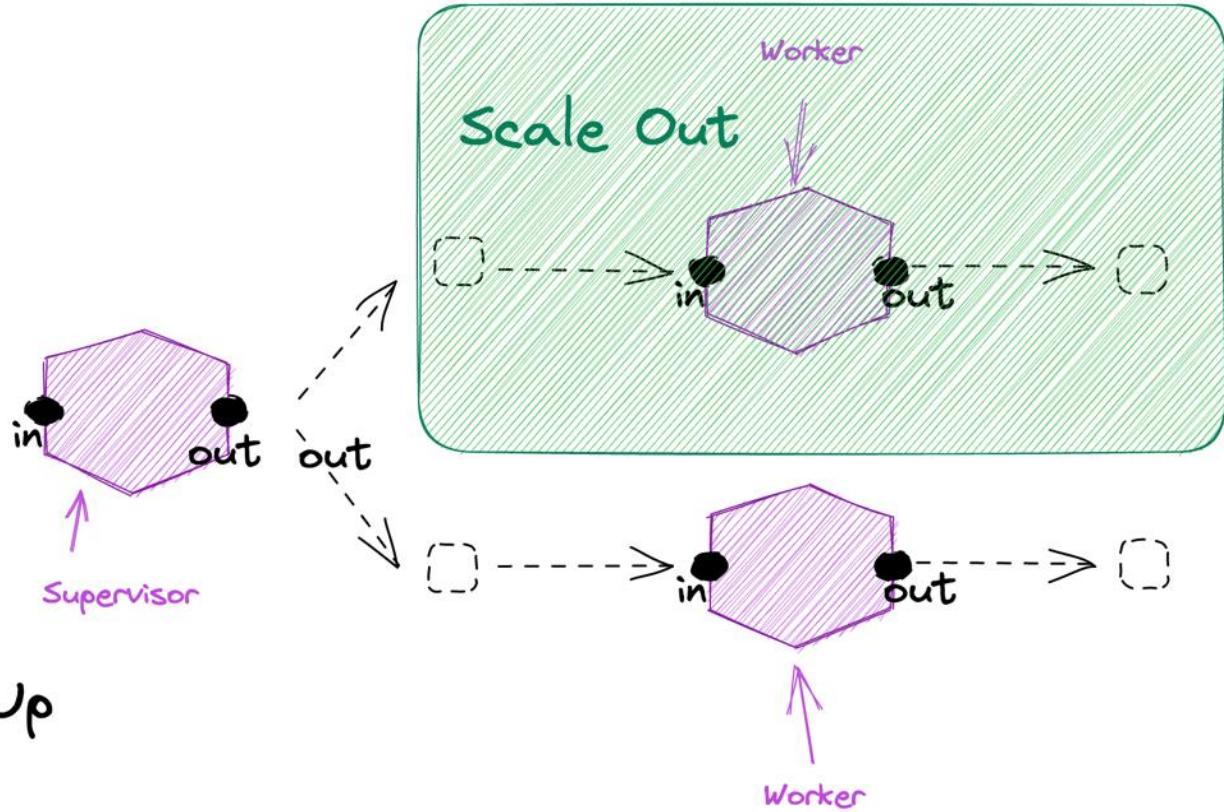
The Reactor uses a single thread to: read from the queue; dispatch to a handler; and execute the handler.

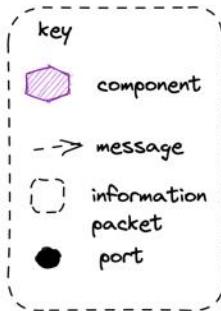
Note that this preserves ordering and does not require locking to protect mutable state.



Scale Out, not Up

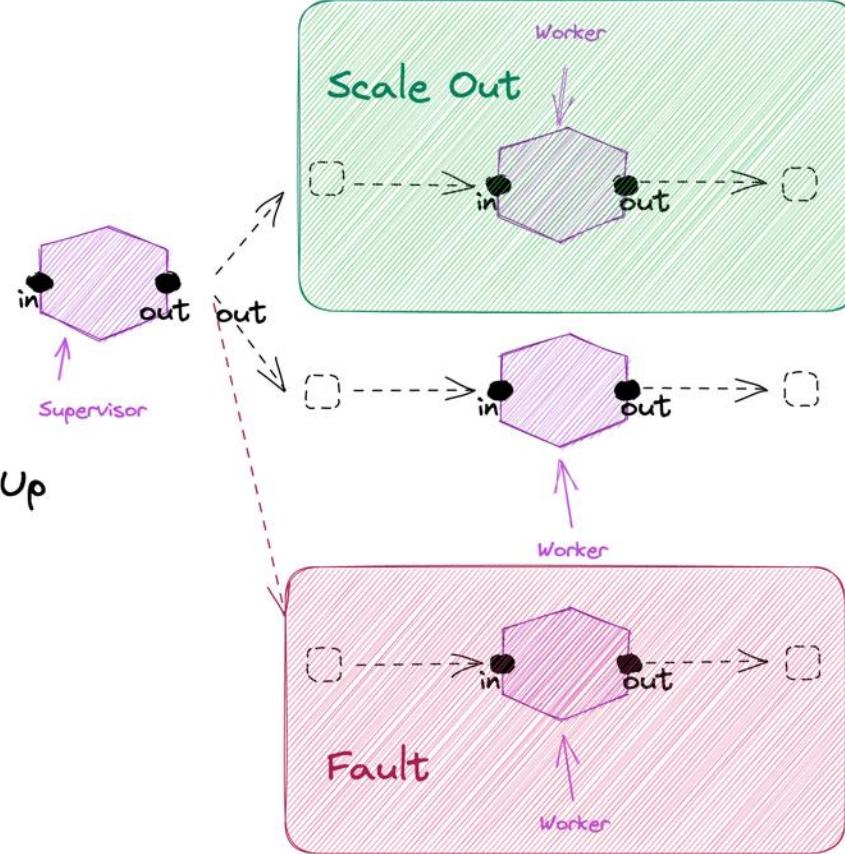
12-factor et al.





Scale Out, not Up

12-factor et al.



Paper Workflows

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



Messaging/Discrete Event

Discrete, Immediately Actioned

Skinny



Series Event (Document)

Series, Supports Action

Fat





OE DIVISION
OEPI ROUTING SHEET

TITLE: EID-E15 (23 July 84)

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

Upon Completion, Return to **ETI Barmag**

**Destroy
File**

DPS/OE Form No. XXXXXXXXXX



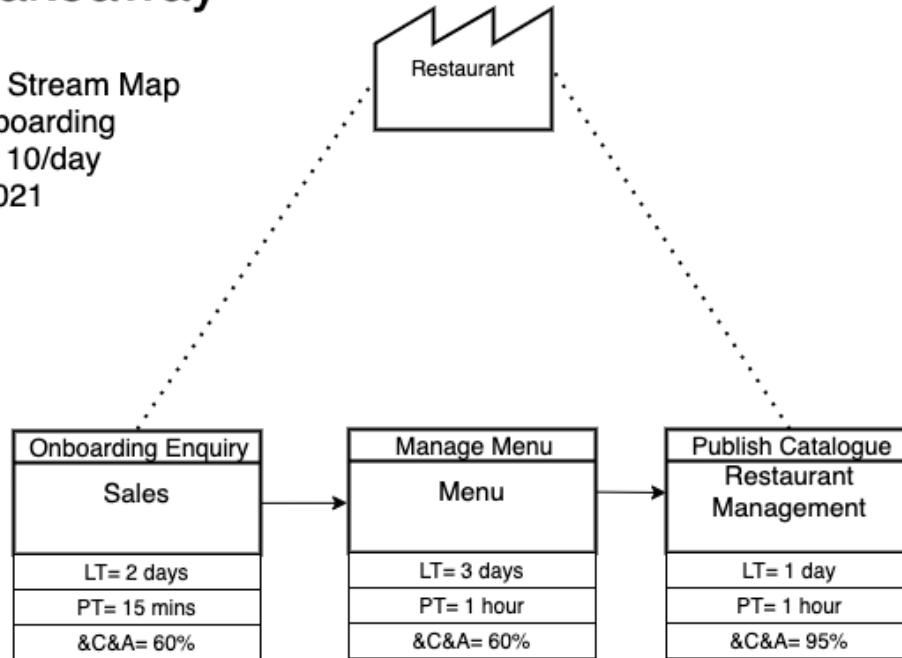
What is a microservice?

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

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

Just Paper Takeaway

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



1

Restaurant
Owner



Restaurant Onboarding



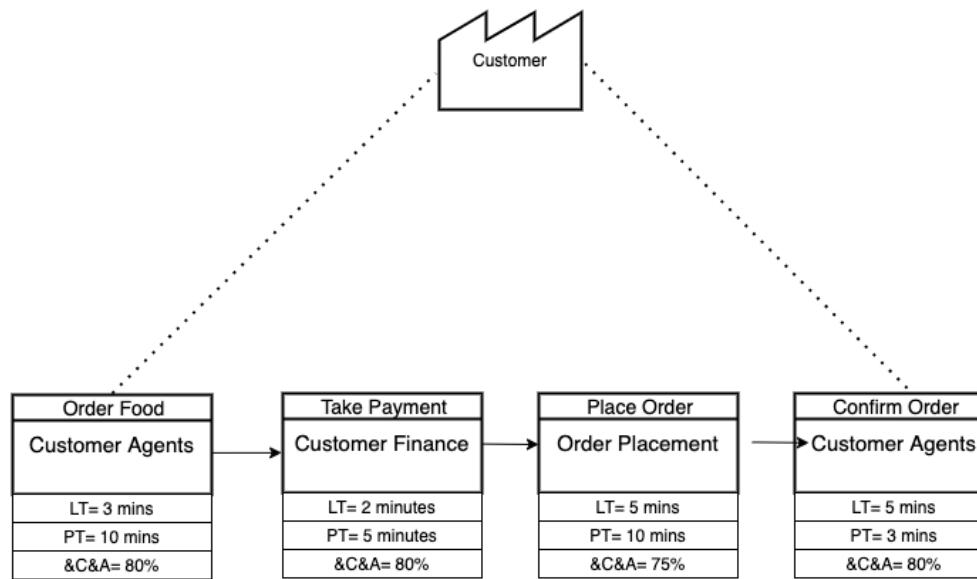
Just Paper Takeaway

Current State Value Stream Map

Order Flow

Demand Rate 10/day

29 SEP 2021



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

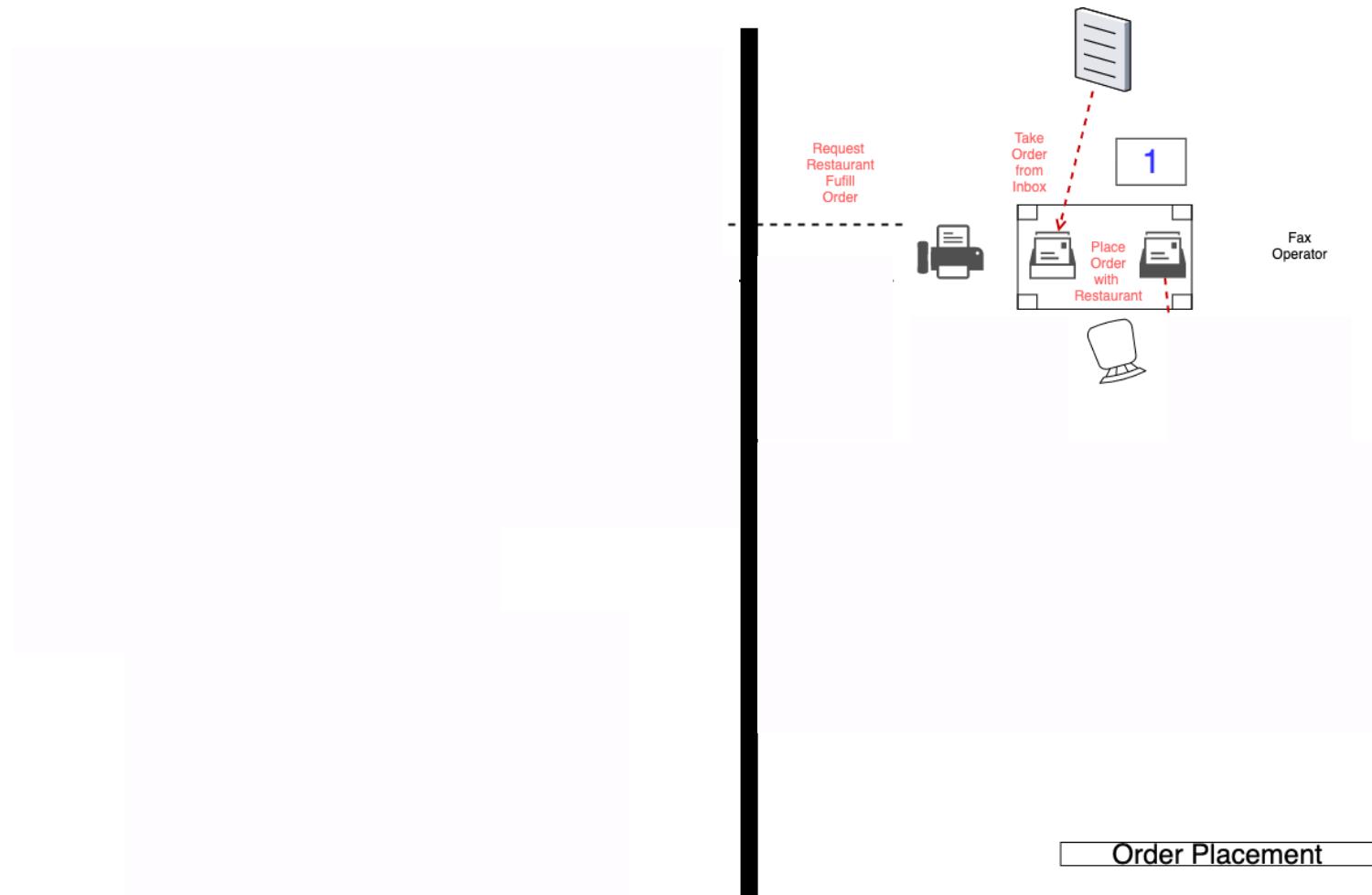
Hungry Customer

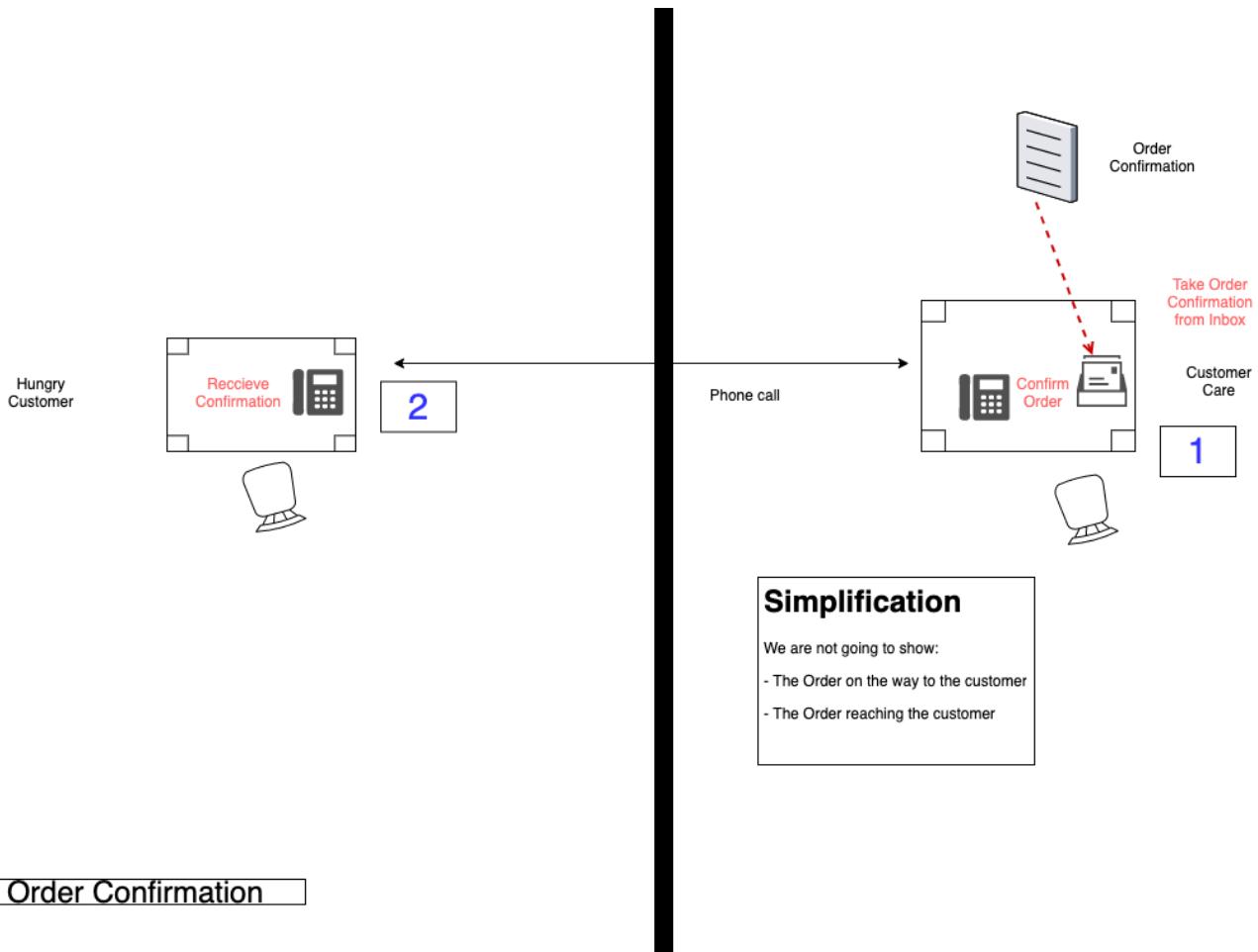


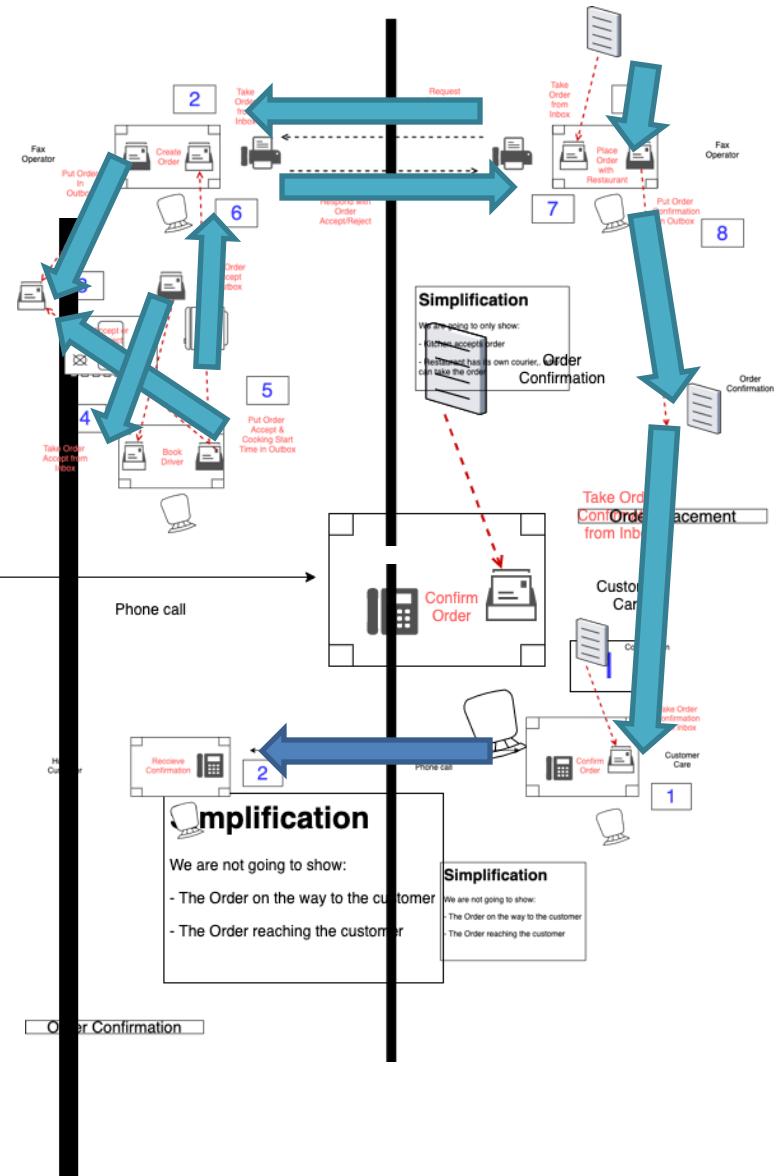
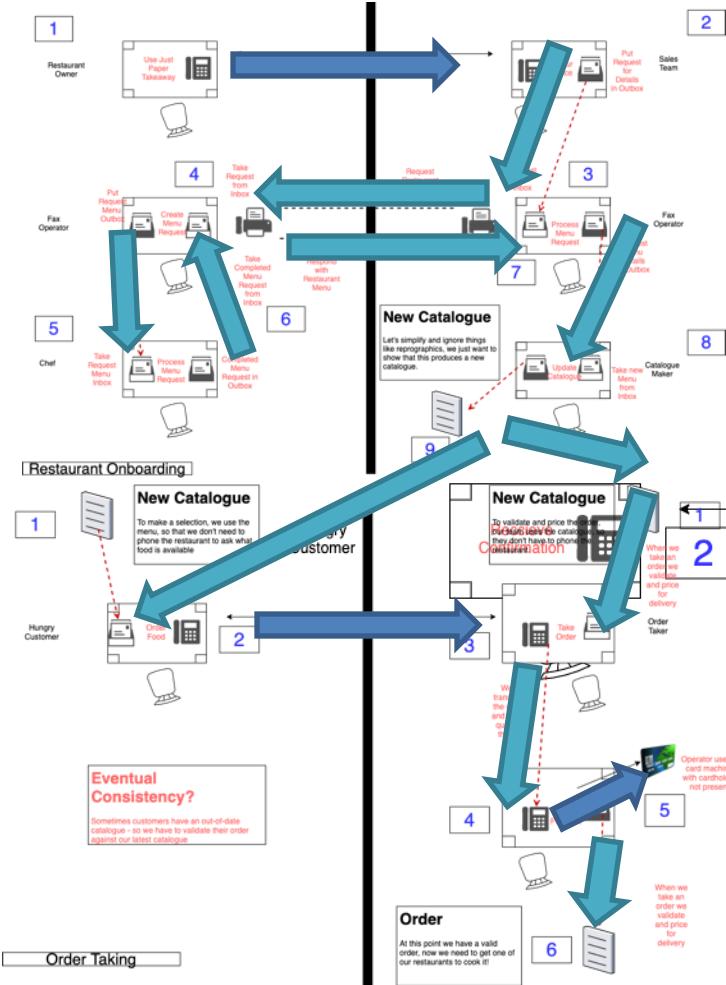
Order Taking

Order Wheel



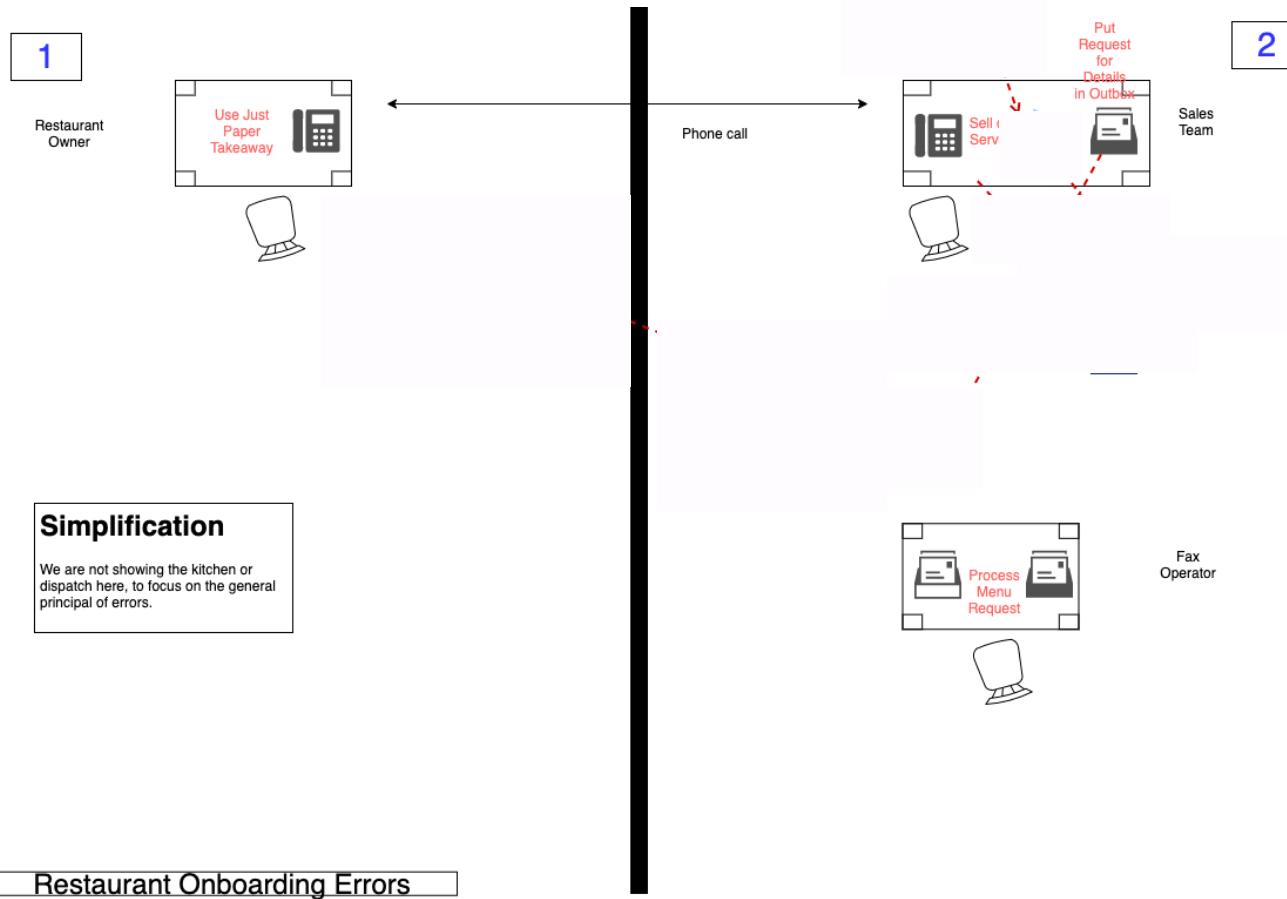






Compensation

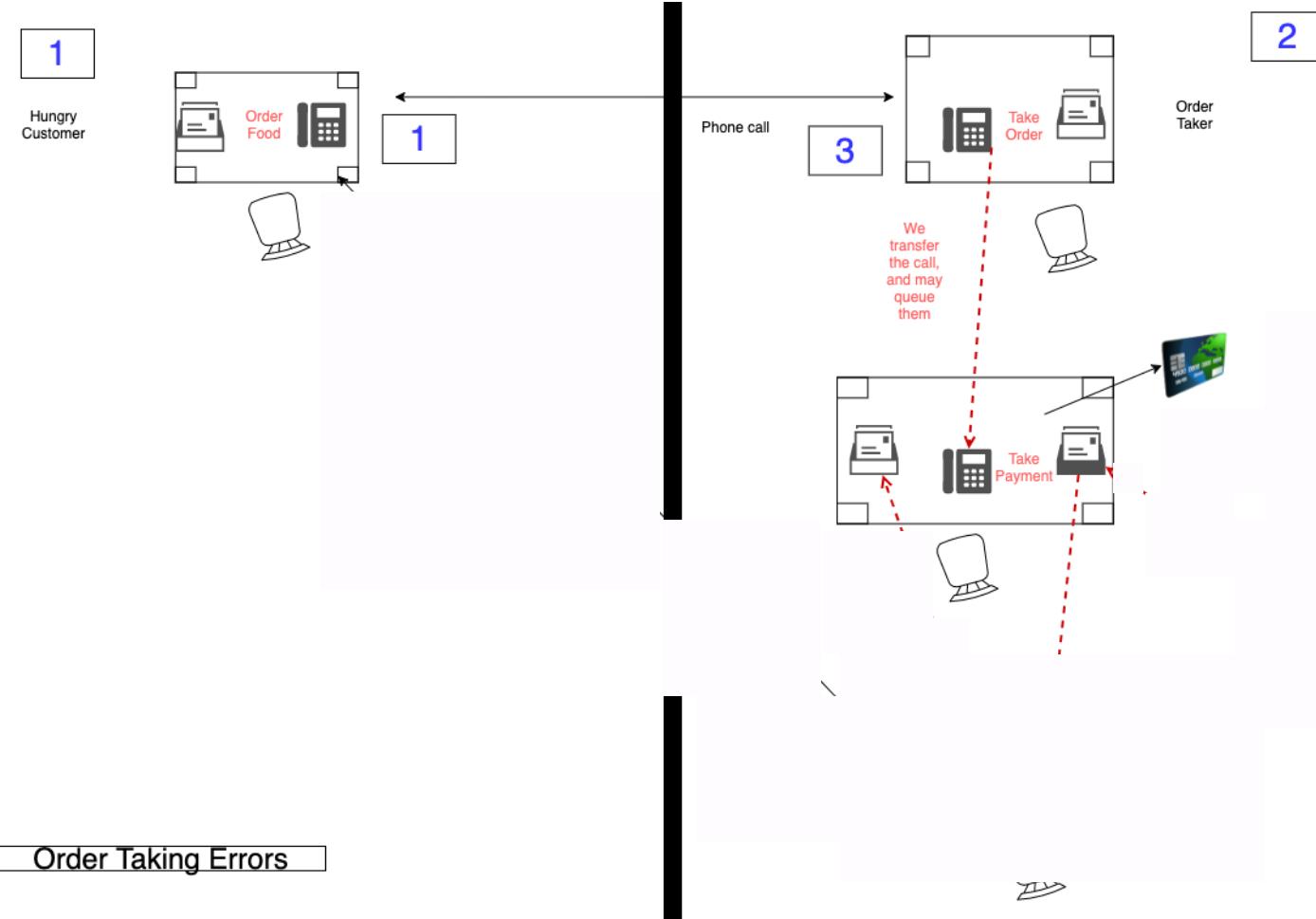


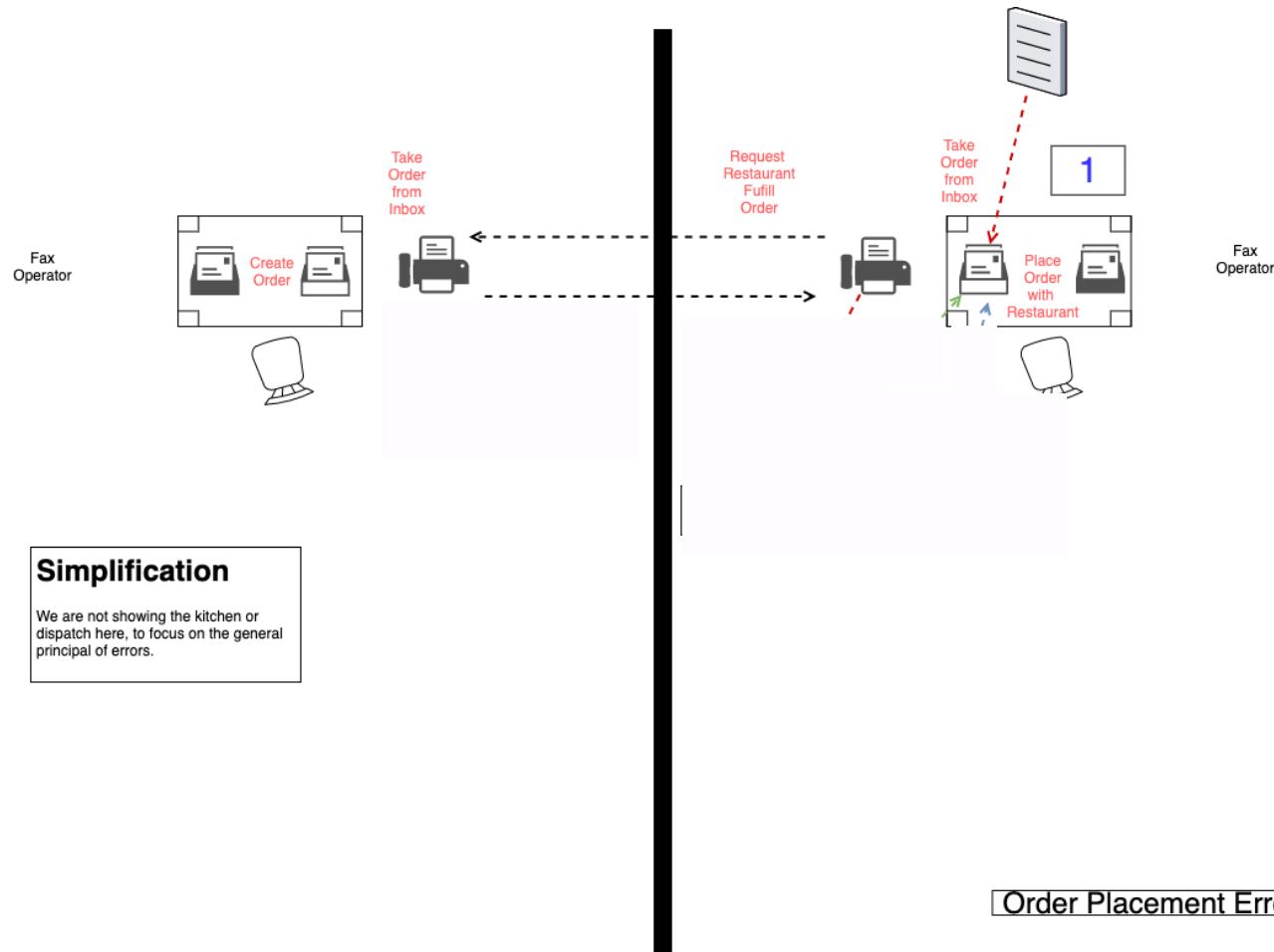


Fax Call Log

Monday, 2010-11-08 11:19

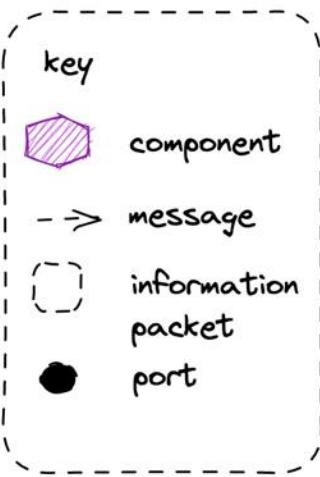
Date	Time	Type	Job #	Length	Speed	Station Name/Number	Pages	Status
2010-06-18	08:31	SCAN	92	0:25	28800	[REDACTED]	0	E-705 V.34 1M31
2010-03-22	11:39	RECV	59	0:20	26400	[REDACTED]	1	OK -- V.34 BM31
2010-03-22	11:45	RECV	60	0:20	26400	[REDACTED]	1	OK -- V.34 BM31
2010-03-22	12:29	RECV	61	0:21	26400	[REDACTED]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





Flow Based Programming

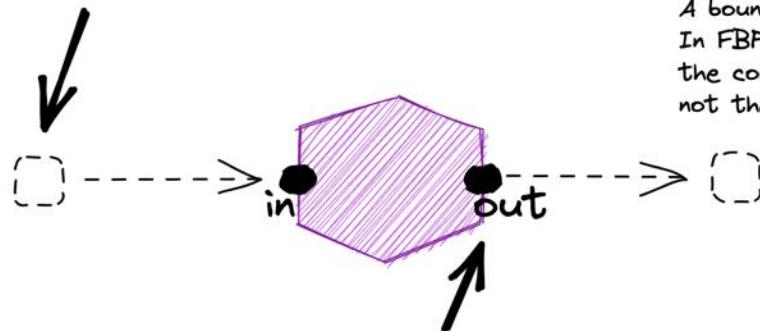
“Everything Flows and Nothing Stays.” - Heraclitus



Flow Based Programming (J. Paul Morrison)

Information Packet (IP)

An independent structured piece of information with a defined lifetime.



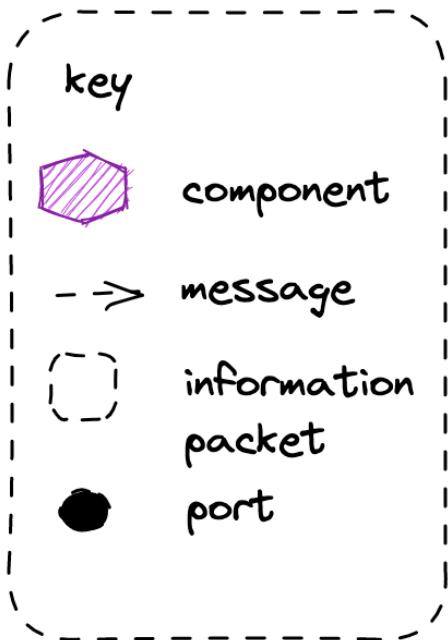
Port

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

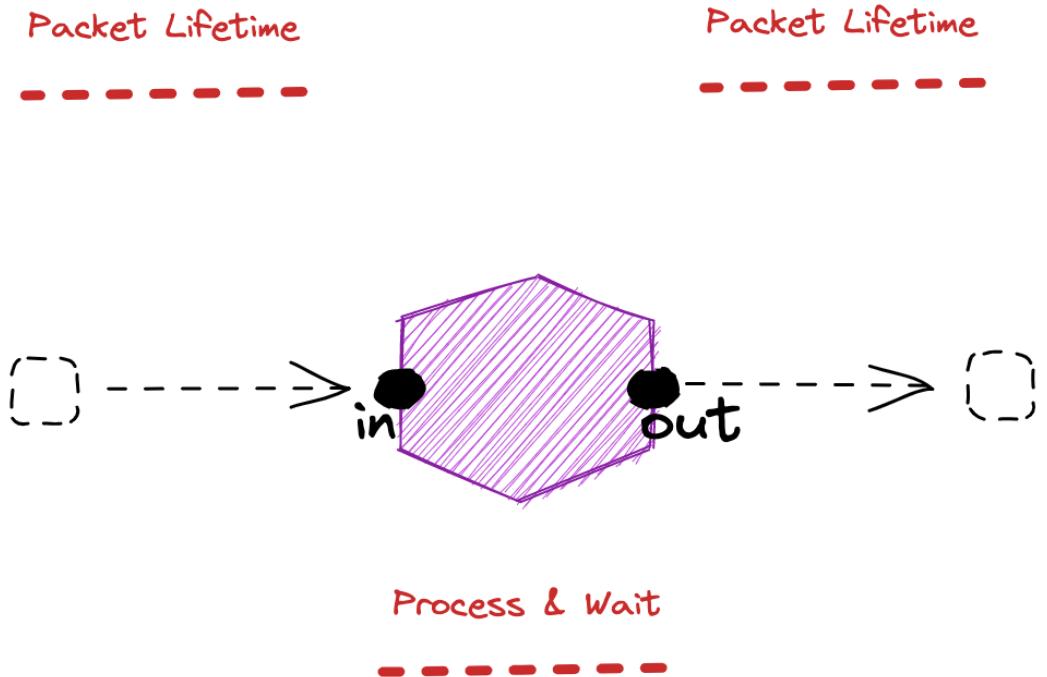
Flow-Based Programming (FBP) is a subclass of DFP. While DFP can be synchronous or asynchronous, FBP is always asynchronous. FBP allows multiple input ports, has bounded buffers and applies back pressure when buffers fill up.

Connector

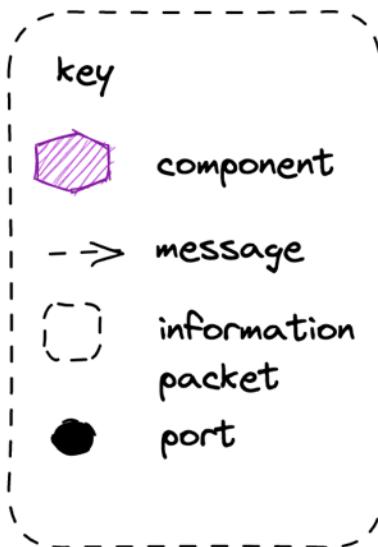
A bounded pipe of information packets.
In FBP a connector is external - that is the component is only aware of the named port not the connector, so it could be swapped out



Flow Based Programming (J. Paul Morrison)



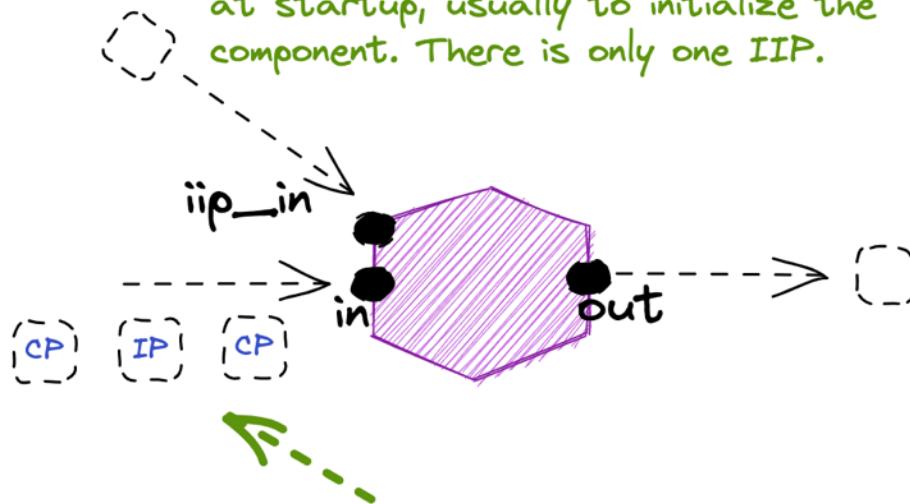
Node Lifetime: In flow base programming the a node can remain running whilst there is work on an input queue, can can suspend instead of terminate if there is no work on its connector.



Flow Based Programming (J. Paul Morrison)

Initial Information Packet

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

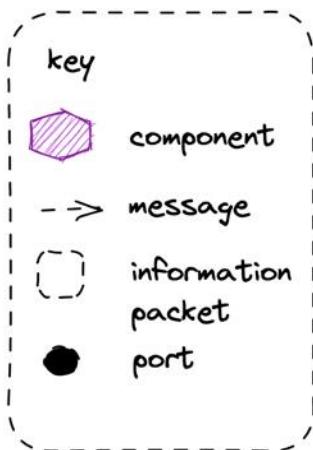


Control Packets

Control Packets can be used to 'bracket' groups

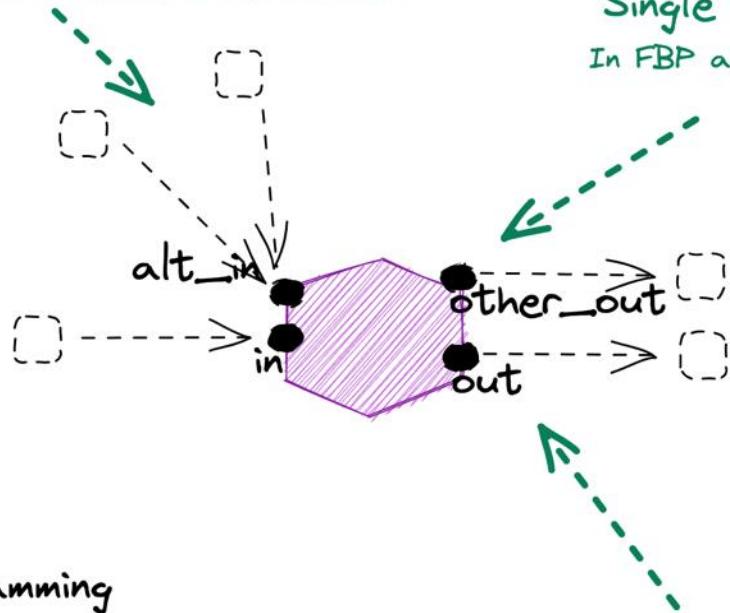
Multiple Writers

In FBP an in port can have multiple writers



Flow Based Programming (J. Paul Morrison)

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

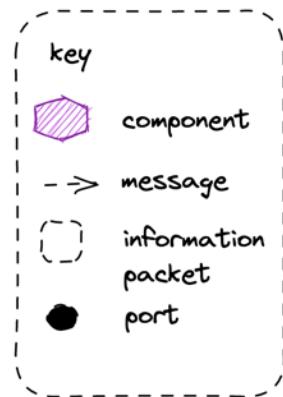


Single Writer

In FBP an out port is single writer

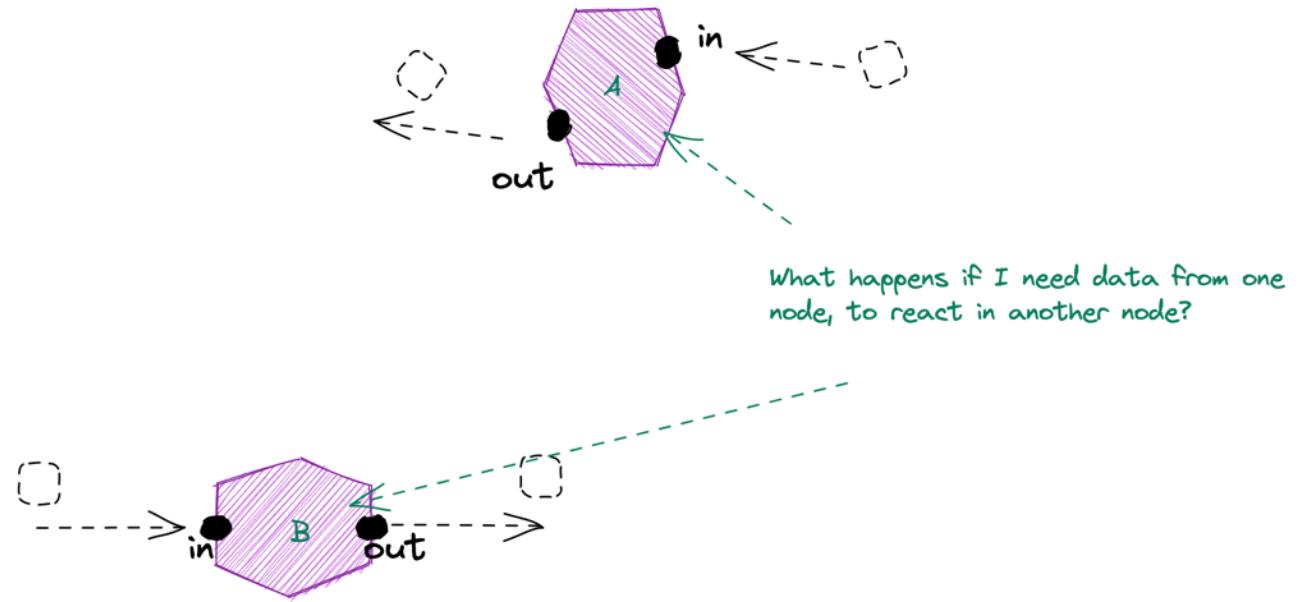
Multiple In/Out Ports

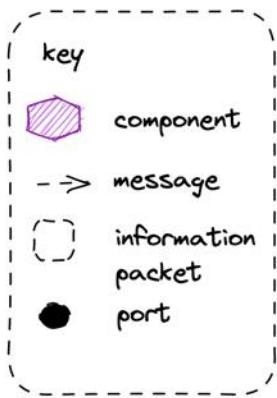
In FBP we can have multiple in or out ports



Flow Based Programming

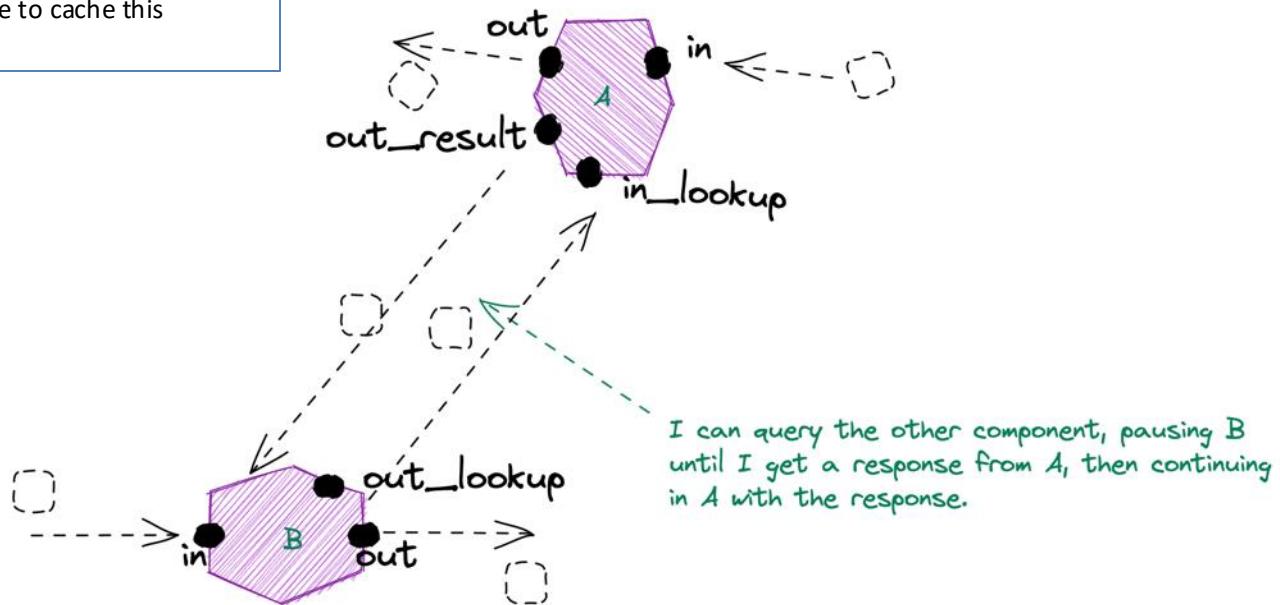
(J. Paul Morrison)

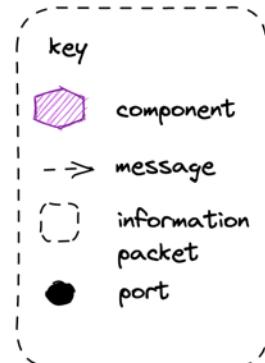




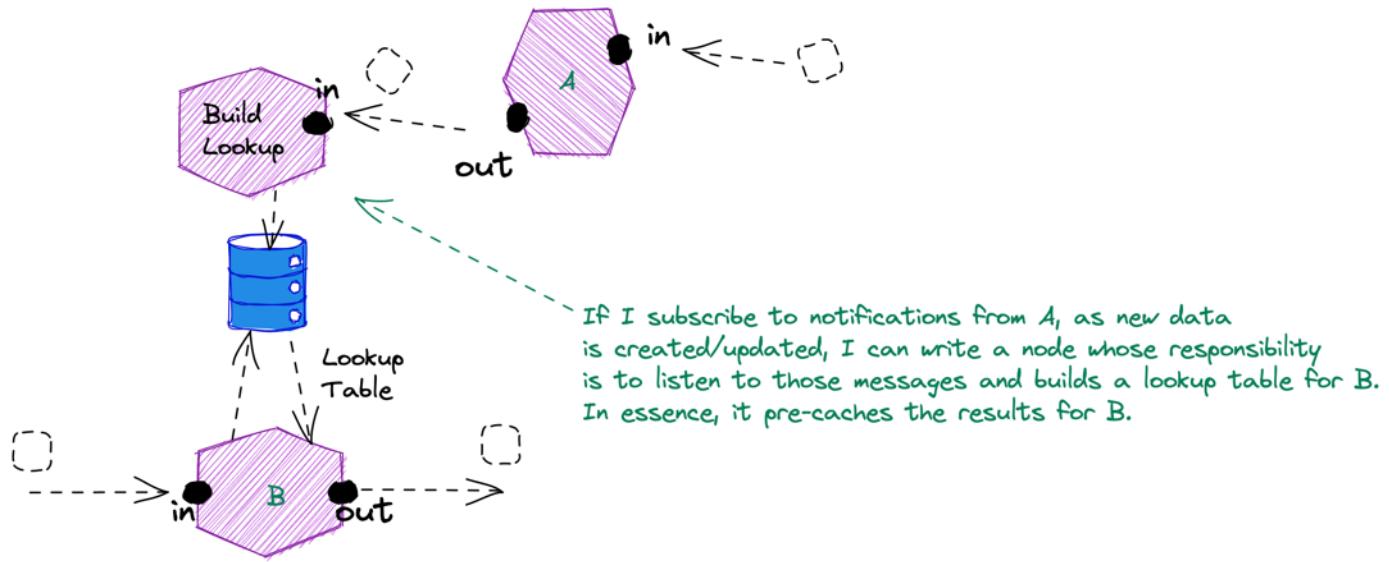
Flow Based Programming (J. Paul Morrison)

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



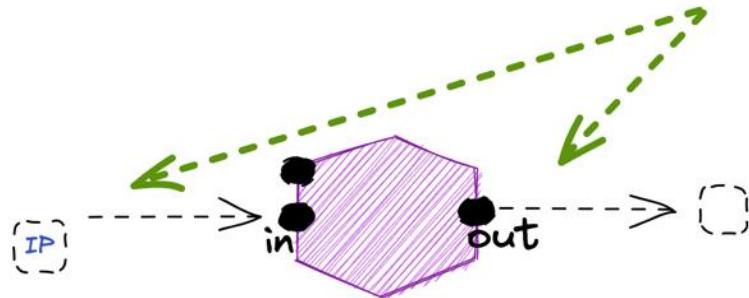
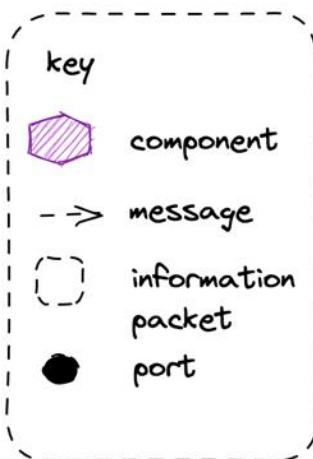


Flow Based Programming (J. Paul Morrison)

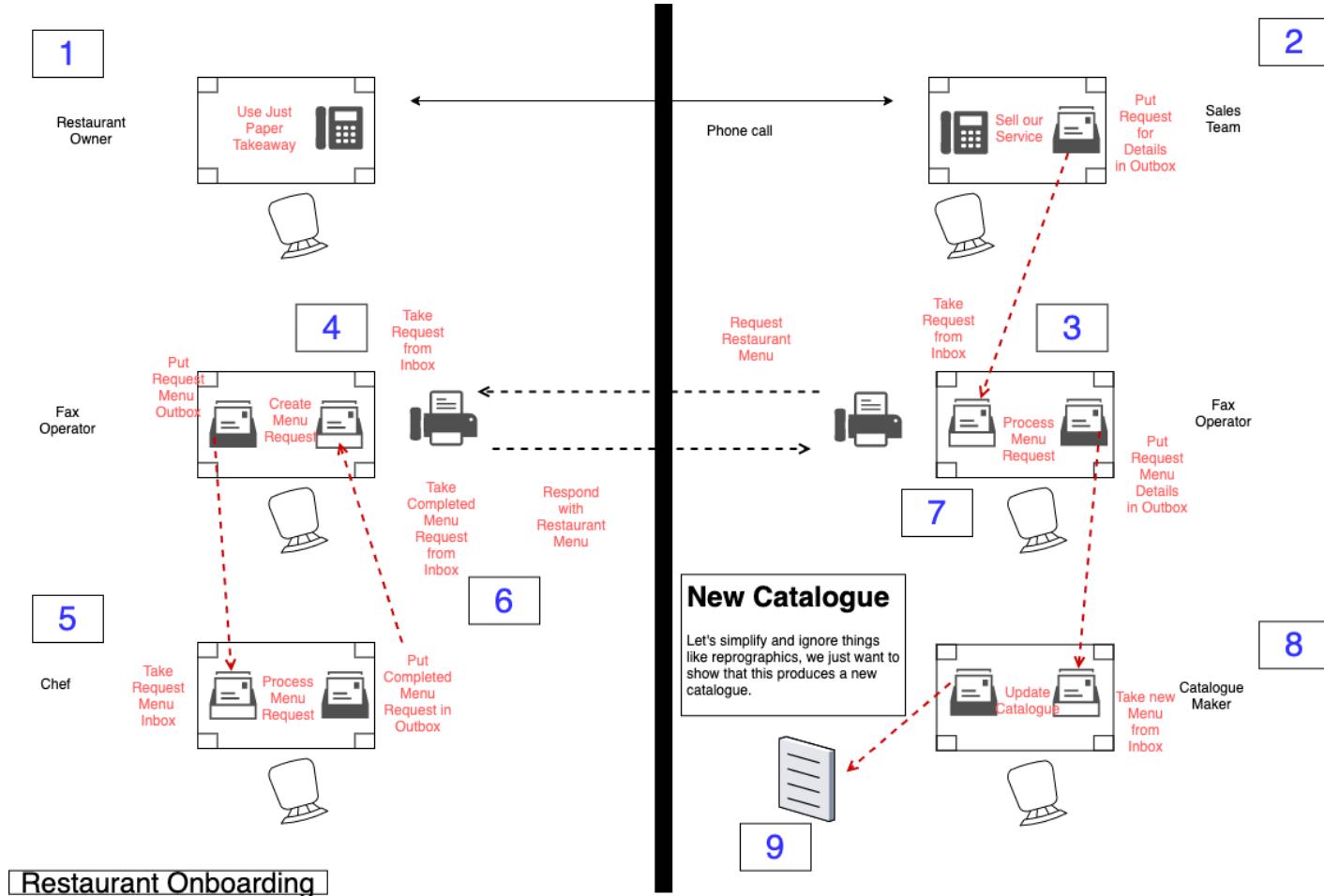


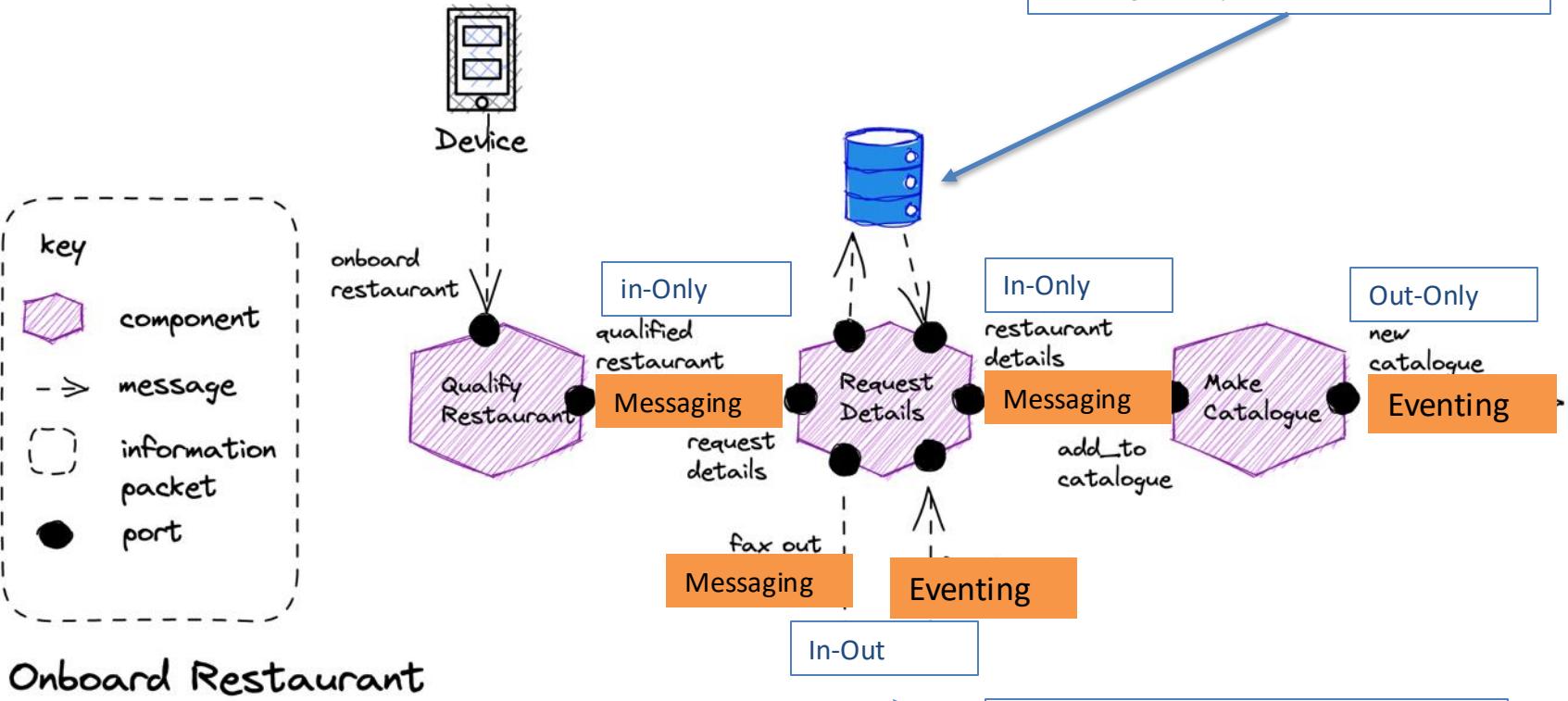
Message Oriented Middleware (MoM)

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



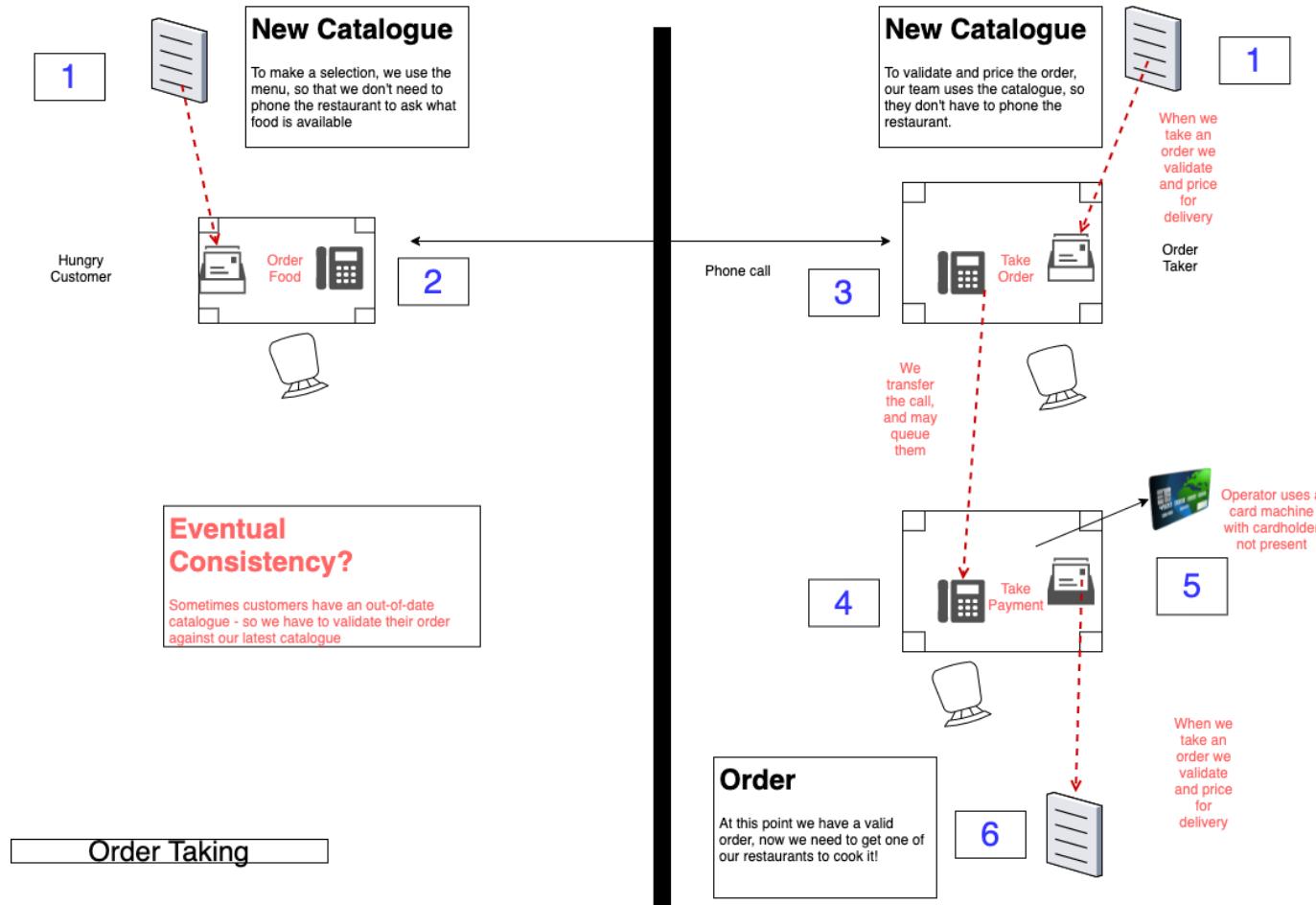
Flow Based Programming - 2010 Update
(J. Paul Morrison)

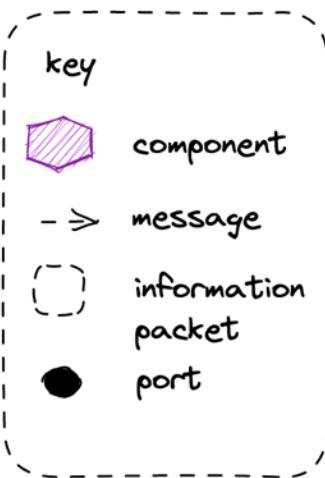




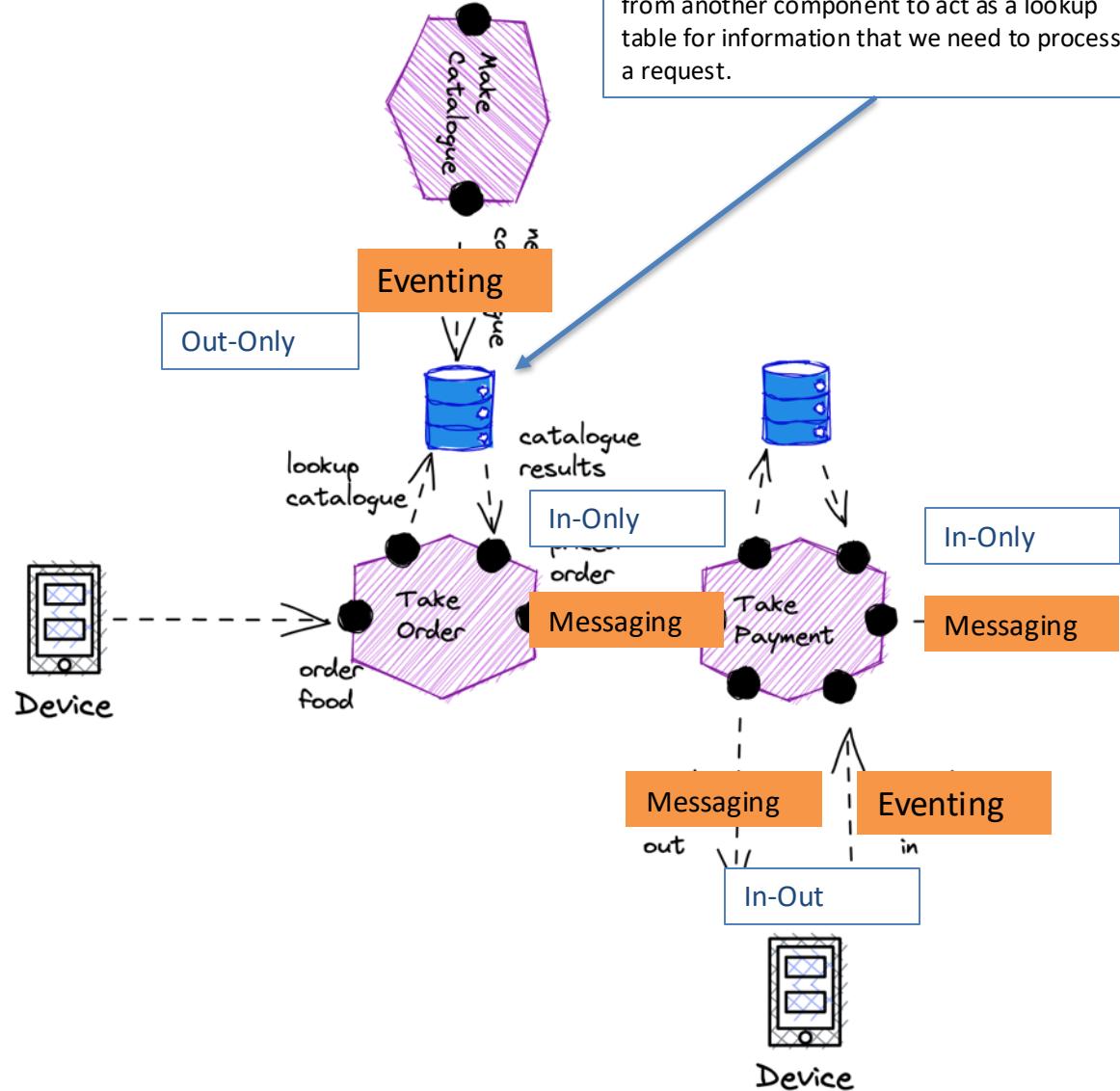
Storage: Request details may need storage to be reliable, it takes work from the **request_details** port, but cannot forward it to the **restaurant_details** port without waiting for an answer from **fax_in**. We store the state of the workflow in case we crash before receiving the response.

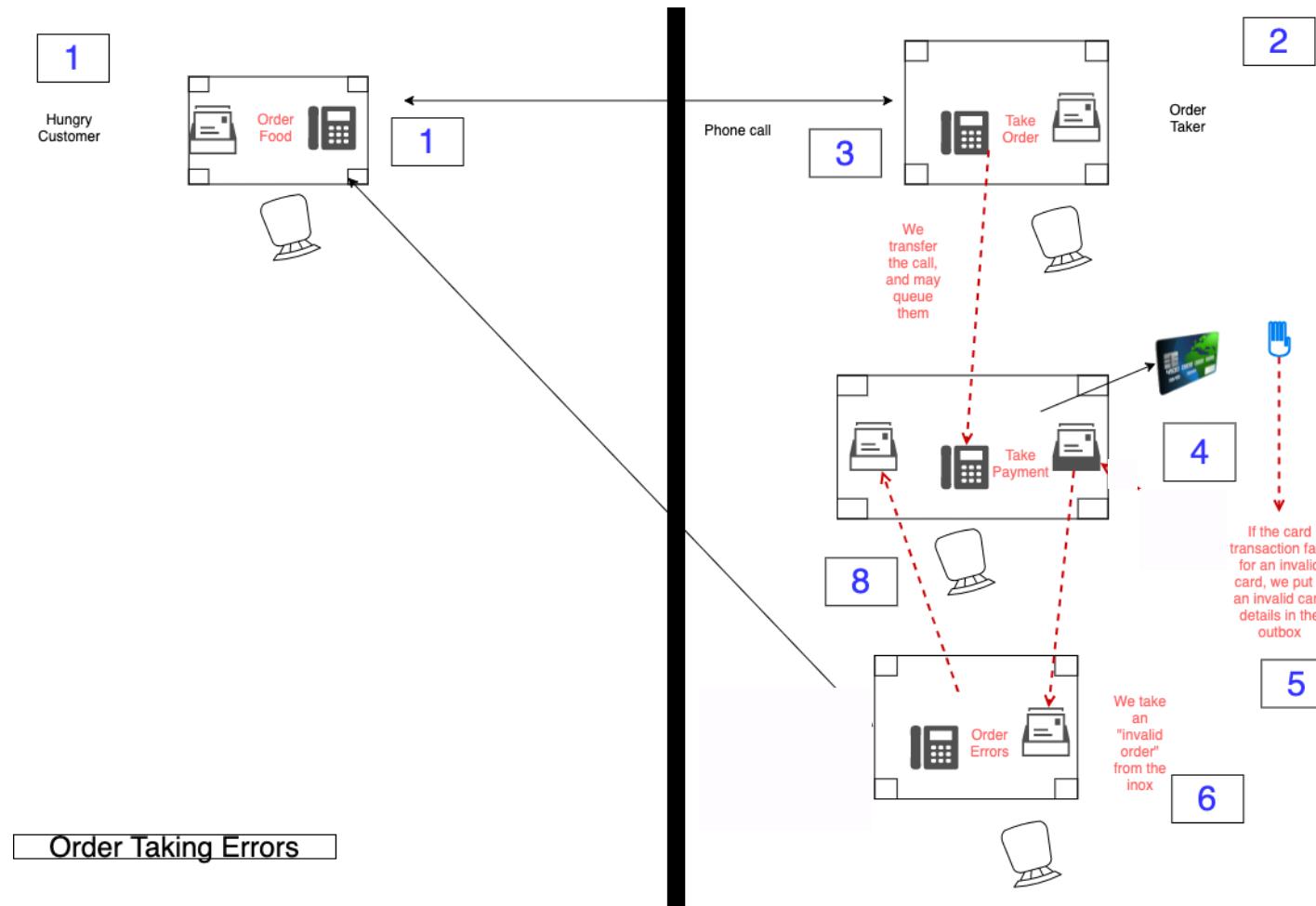
Correlation: To allow us to match the response to the saved state we use a *correlation id*. We send it on the ip to **fax_out**, and the restaurant includes it on the ip from **fax_in**. We use the ip to look up the workflow we stored.

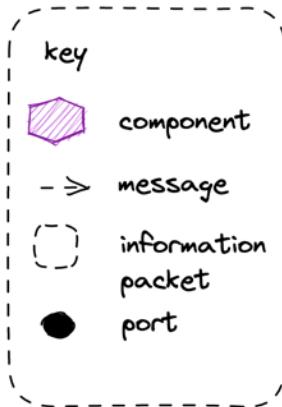




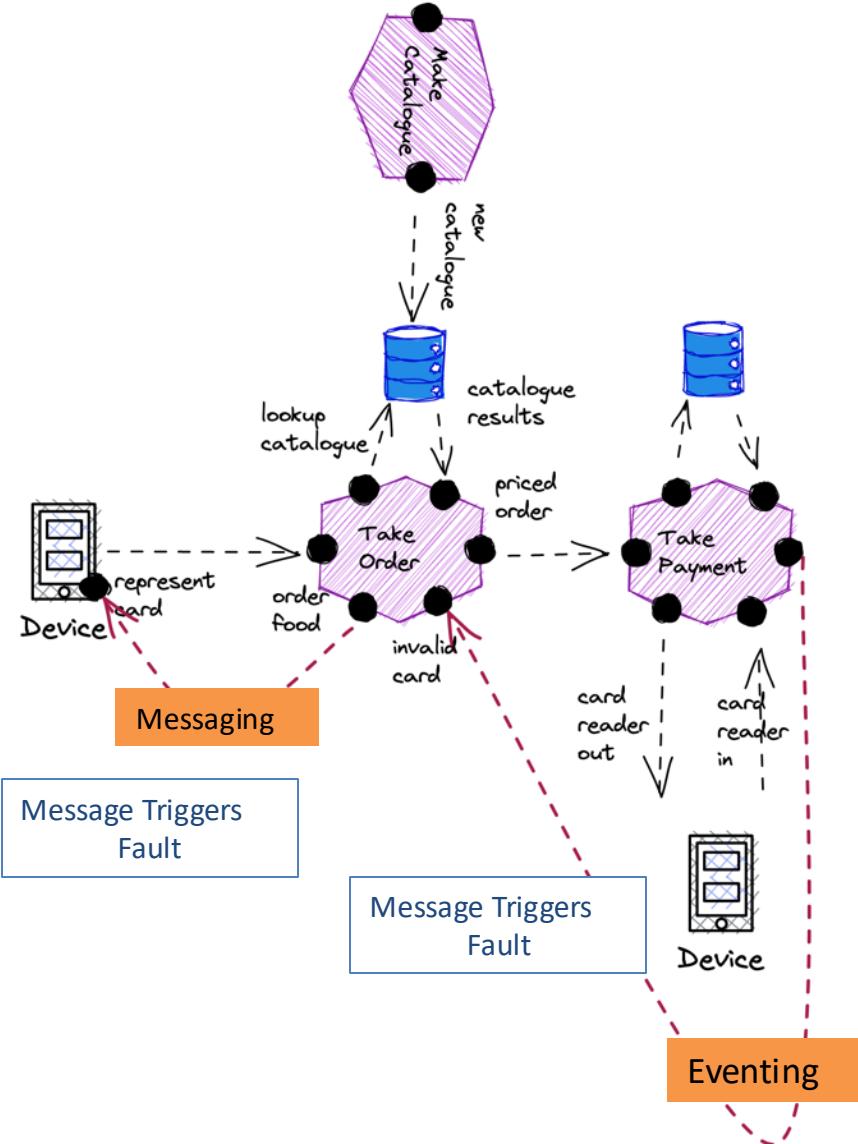
Order Food

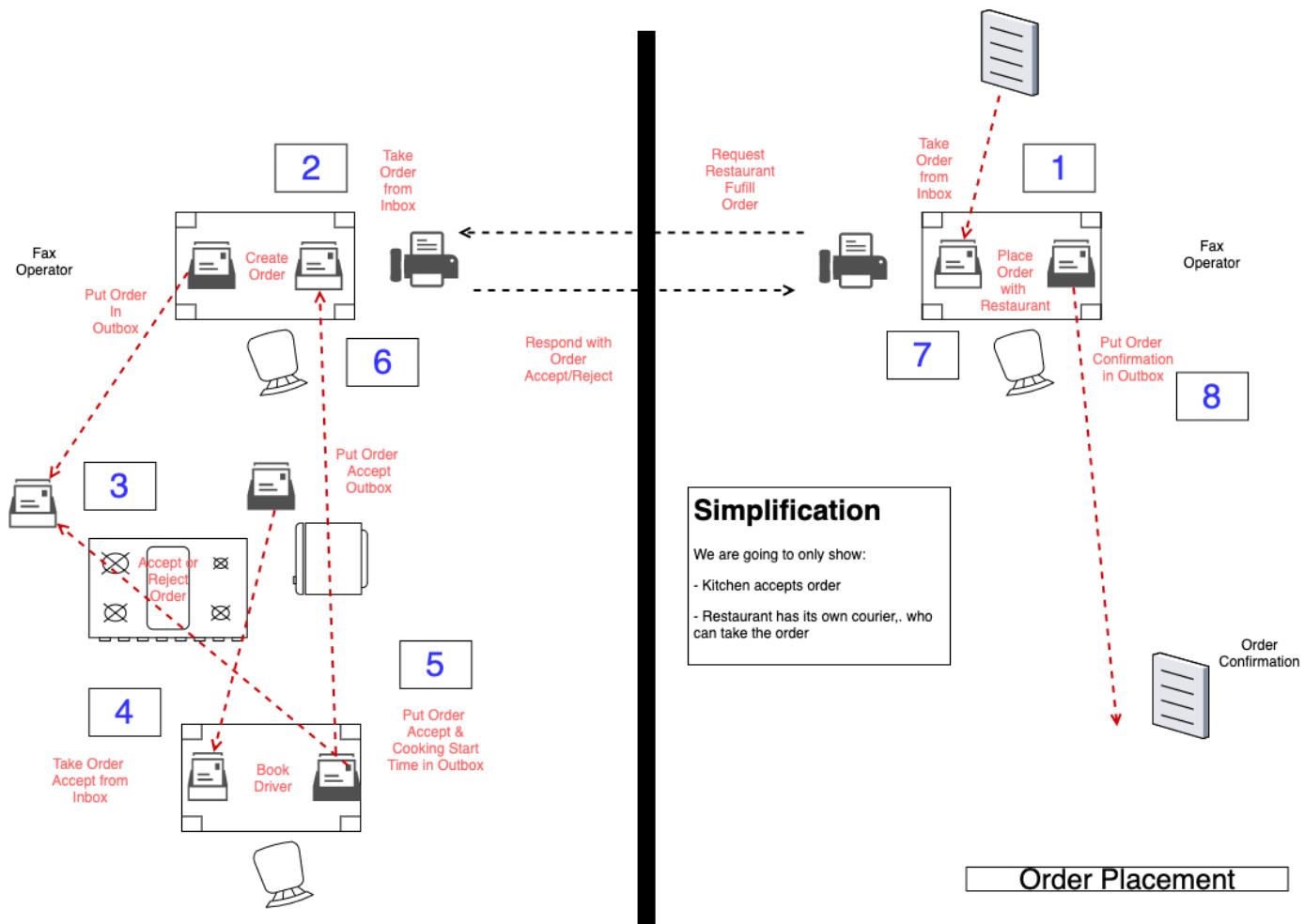


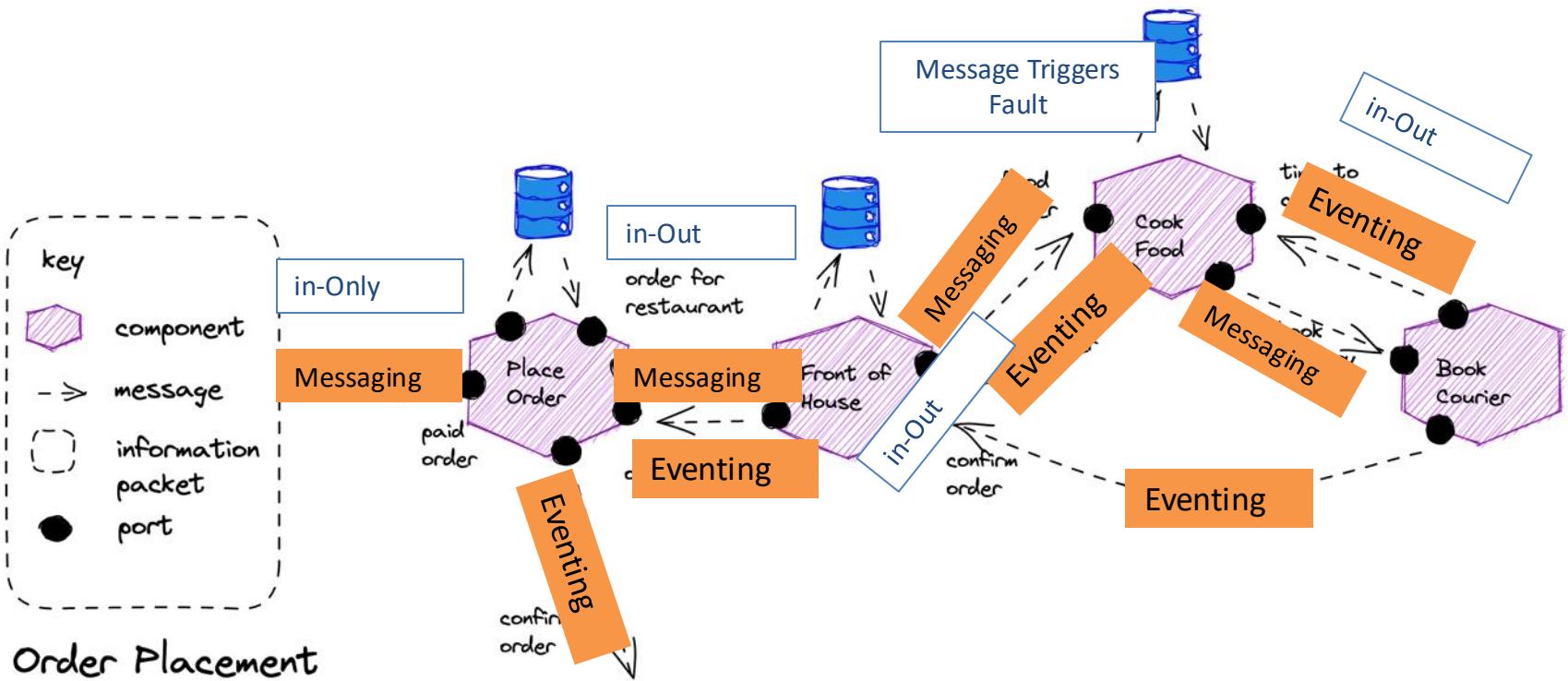


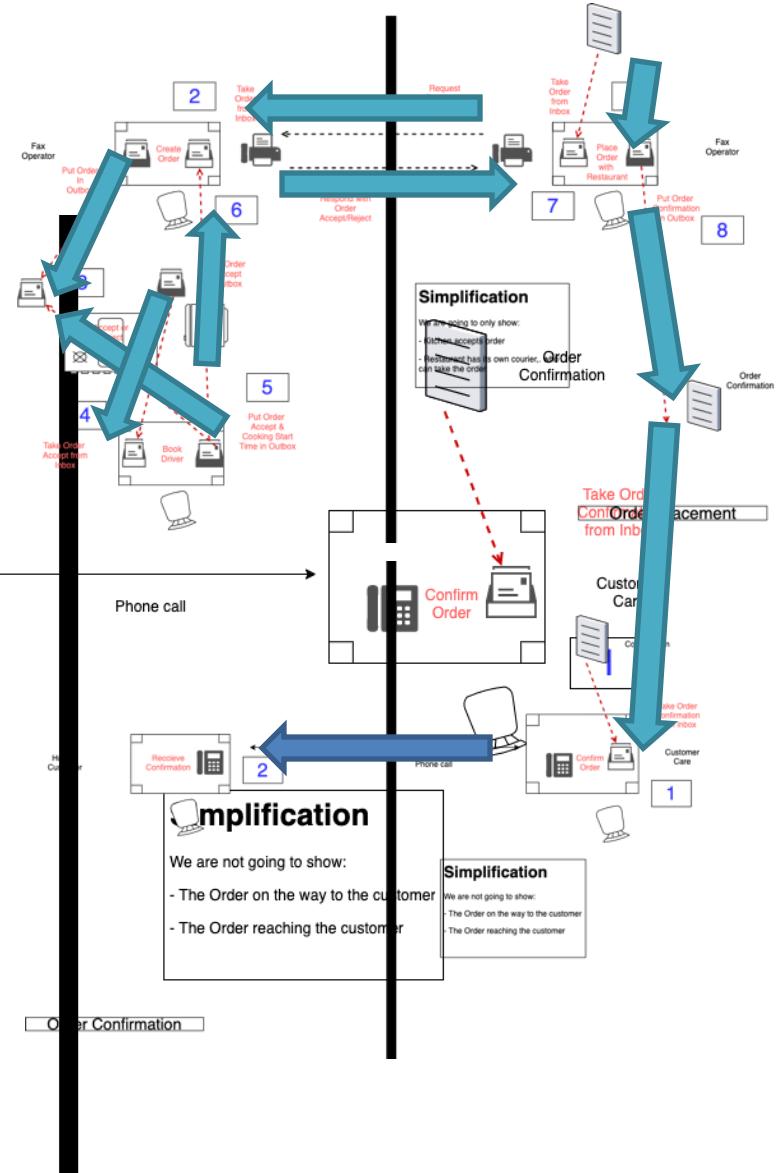
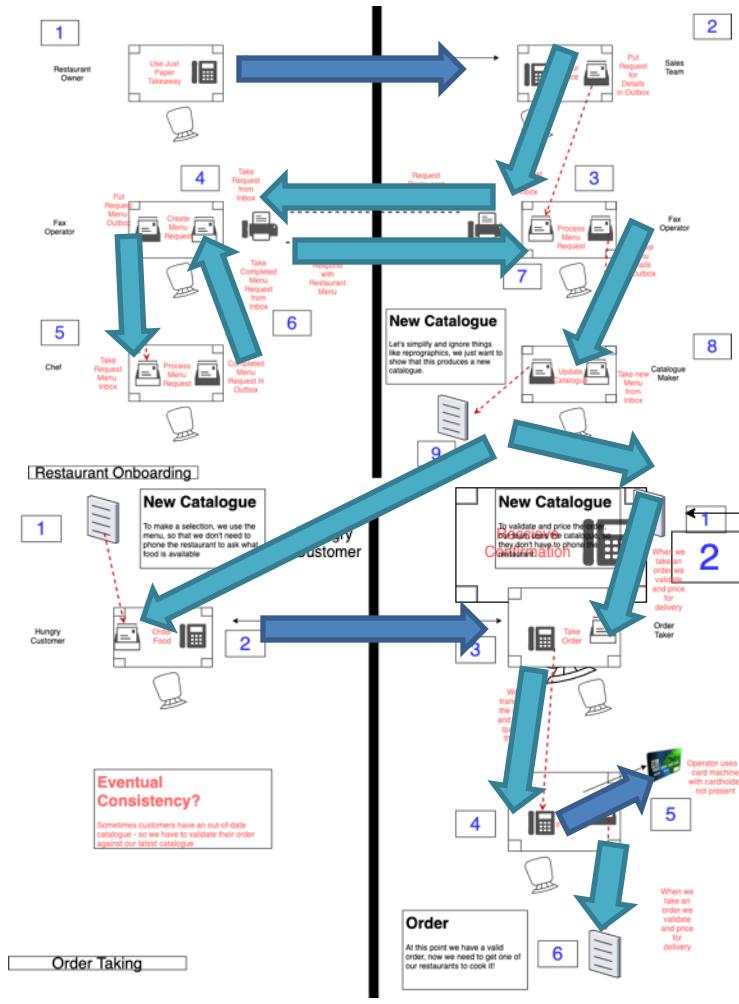


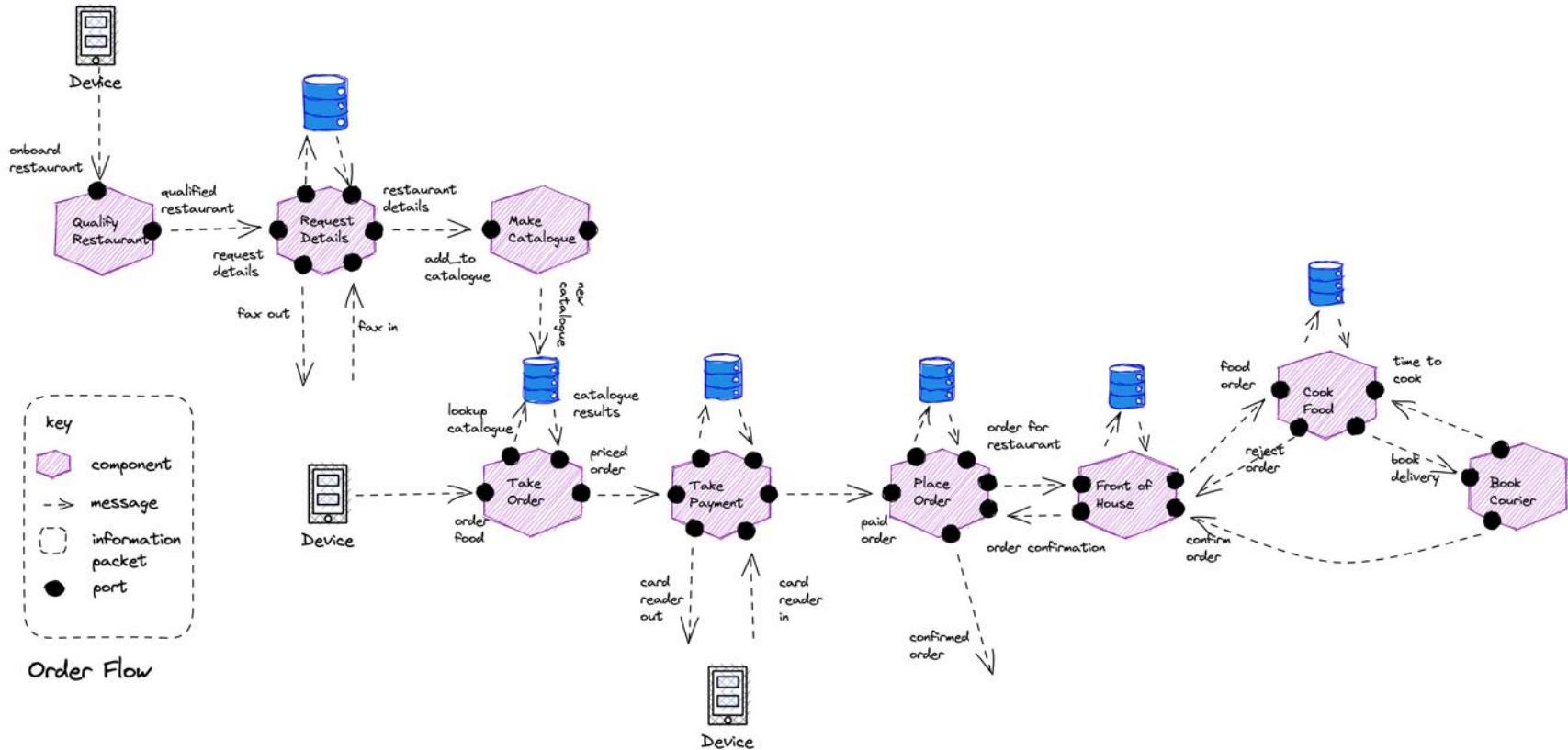
Order Errors











EXERCISE MATERIAL

Paper Flow

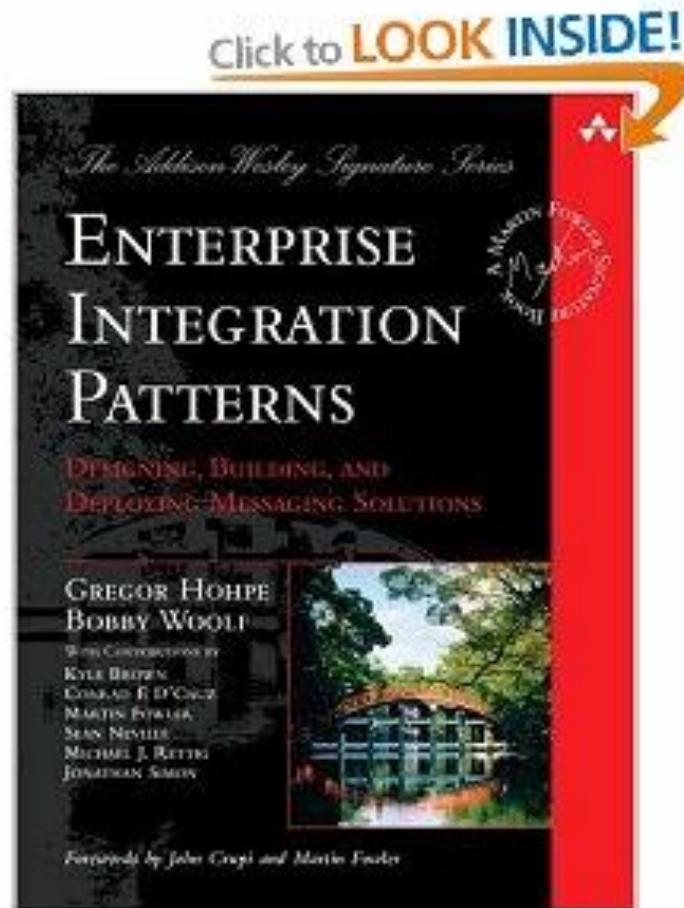
- Readme
- Slides



DON'T PANIC

NEXT STEPS

Further Reading



Further Reading

Serverless Land Content ▾ Learn EDA ▾ Search Search

EDA VISUALS

Small bite sized visuals about event-driven architectures

Designs and thoughts from [@bogmey123](#)

Avoiding the big ball of mud in event-driven architectures

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

Local cache copy vs requesting data

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

What are events in event-driven architectures?

Bite sized visual to help you understand events.

What is Event Sourcing?

Bite sized visual to help you understand event sourcing.

Queues vs Streams vs Pub/Sub

Bite sized visual to help understand the differences.

Reducing team cognitive load with event-driven architecture

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

What are Queues?

Bite sized visual to help understand queues.

What are Streams?

Bite sized visual to help understand streams.

What are Pub/Sub?

Bite sized visual to help understand pub/sub.

Reducing team cognitive load with event-driven architectures

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

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

Q&A