

# Reactive Architectures

Succeeding with Reactive Architectures

Ian Cooper

ICooper@Twitter

@lcooper@hachyderm.io

@icooper.bsky.social

# 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



# Welcome to Brighter

This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.

It can be used for implementing [Ports and Adapters](#) and [CQRS \(PDF\)](#) architectural styles in .NET.

It can also be used in microservices architectures for decoupled communication between the services

[GET STARTED](#)

# The Reactive Manifesto

## The Reactive Manifesto

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architecture.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This means they are easy to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.

Reactive Systems are:

**Responsive:** The system responds in a timely manner if at all possible. This means that the system is both responsive and available, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

**Resilient:** The system stays responsive in the face of failure. This applies not only to highly available, fault-tolerant systems – any system that can fail will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

**Elastic:** The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

**Message Driven:** Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and the reuse of existing infrastructure. It also allows for consistency in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.



Large systems are composed of smaller ones and therefore depend on the Reactive properties of their constituents. This means that Reactive Systems apply design principles so these properties apply at all levels of scale, making them composable. The largest systems in the world rely upon architectures based on these properties and serve the needs of billions of people daily. It is time to apply these design principles consciously from the start instead of rediscovering them each time.

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:* the 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

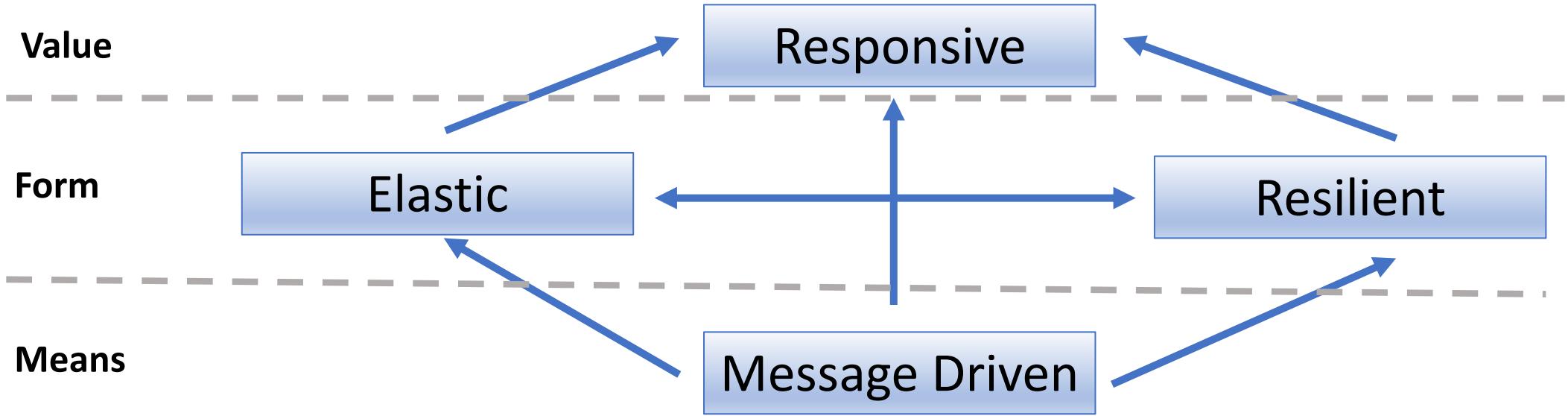
**Responsive:** The system responds in a timely manner.

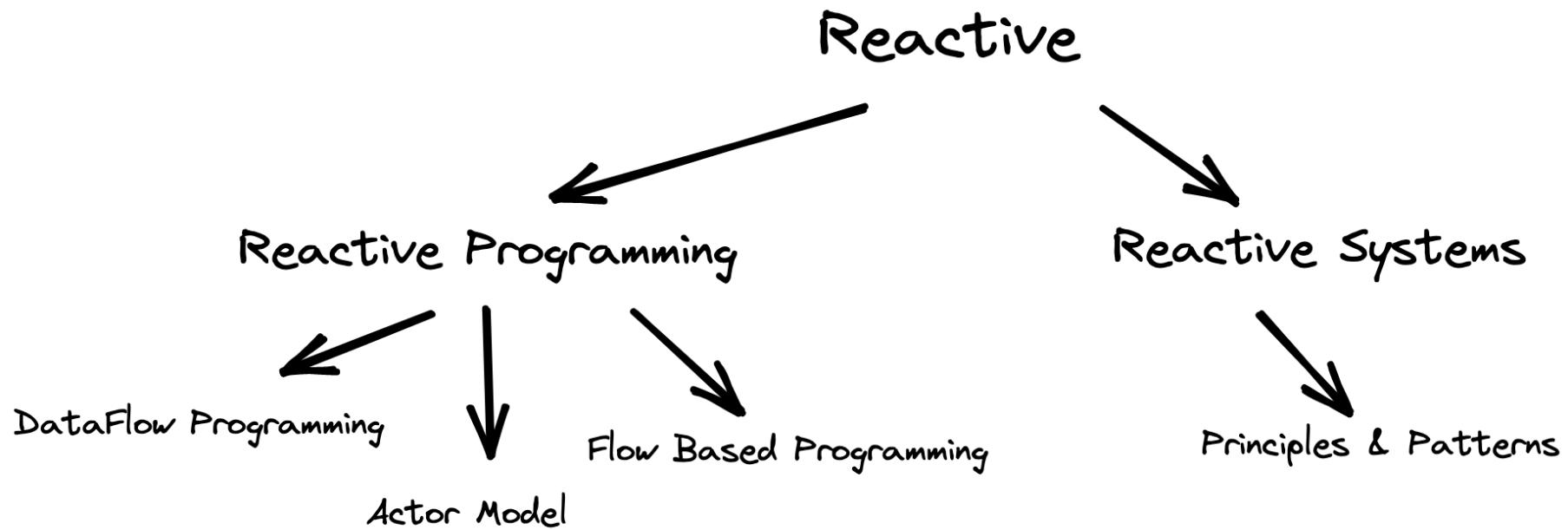
**Resilient:** The system stays responsive in the presence of failure.

**Elastic:** The system stays responsive under varying workload.

**Message Driven:** Rely on asynchronous message passing

<https://www.reactivemanifesto.org/>





## Core Idea

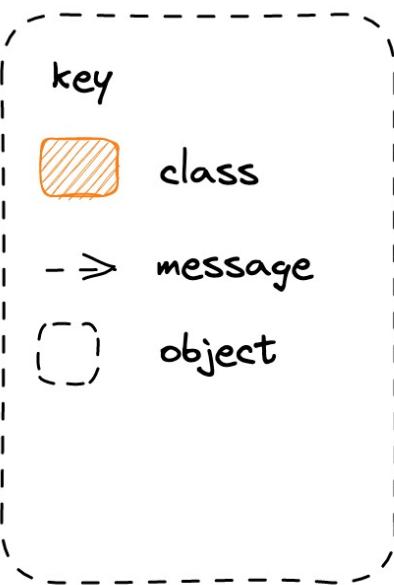
τὰ πάντα ὁρίζει καὶ οὐδὲν μένει

trans: everything flows, and nothing stays

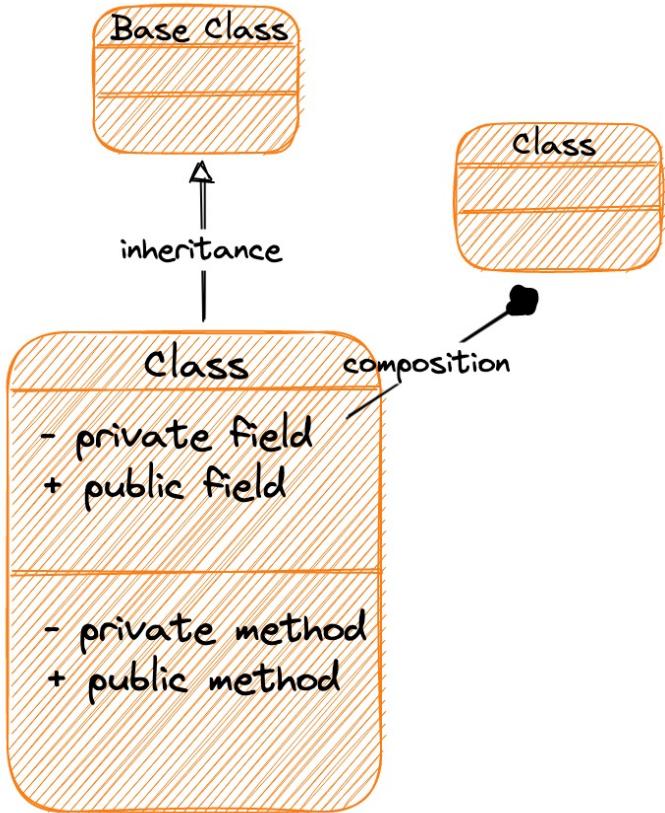
Heraclitus

# Agenda

- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness



## Object Orientation

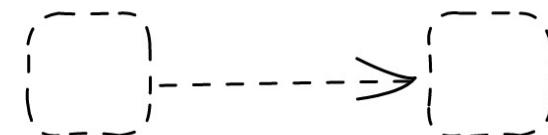


## Data and Behaviour

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

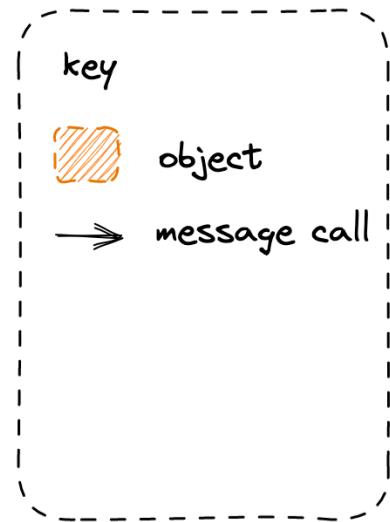
Due to inheritance the method chosen in response to a message may come from the base class of an object



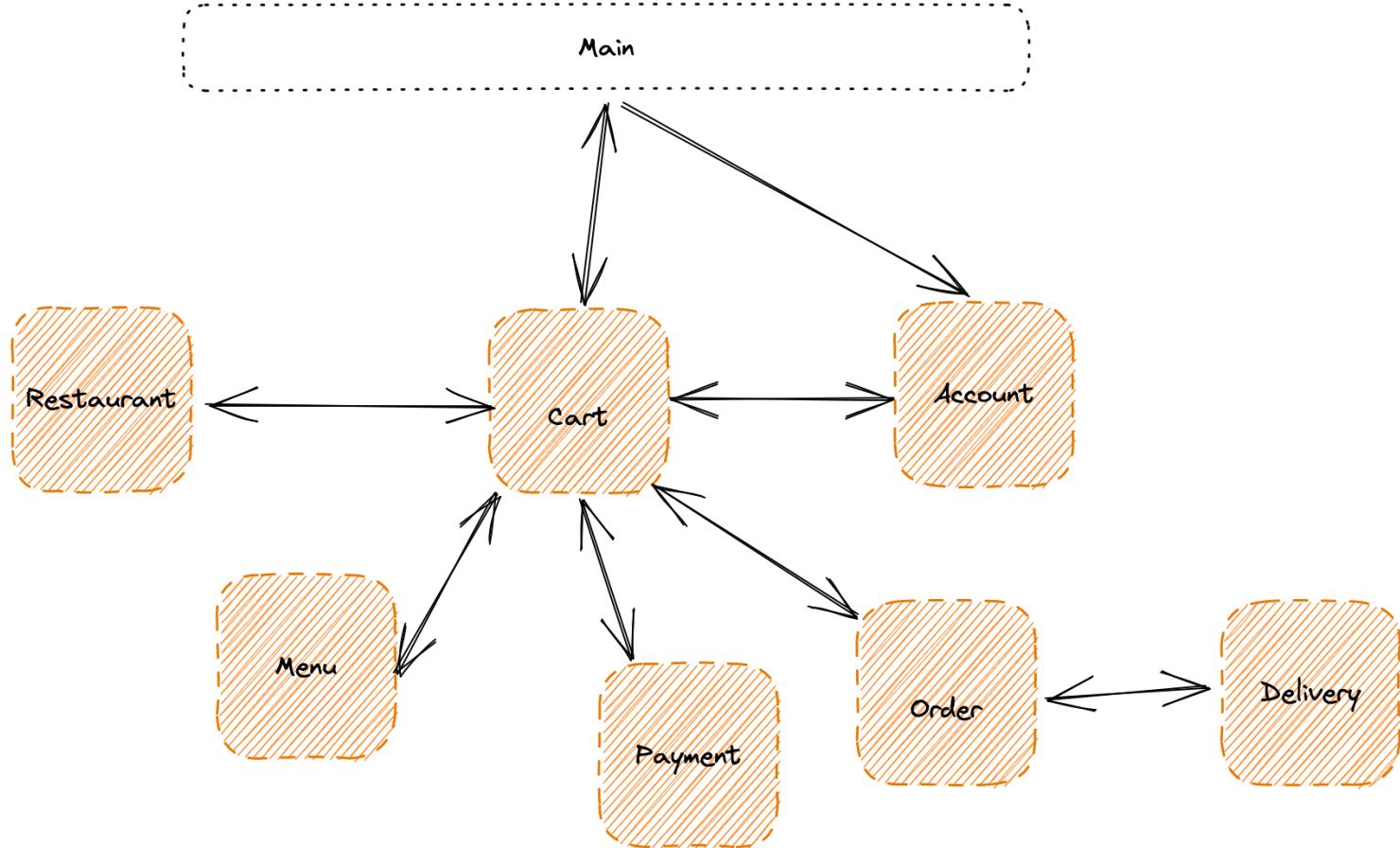
## Message Passing

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

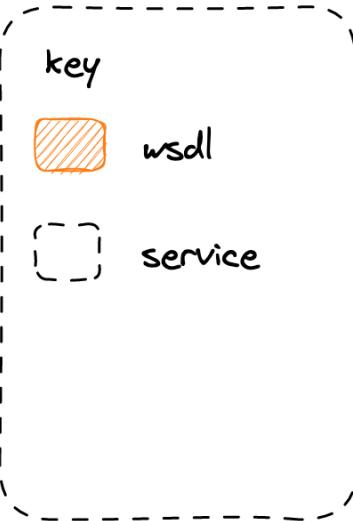
**Call and Return:** The main method represents our entry point and it uses message passing to invoke objects, which in turn invoke other objects, and return to their caller



Call and Return



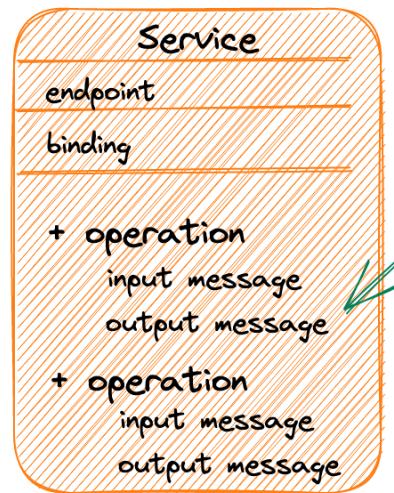
**God Object:** A danger here is that we end up with a "god" object, such as Cart here, that controls all the other objects. This is a high-degree of behavioral coupling



## Service Orientation

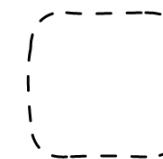
### Data and Behaviour

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



### Messages

A message is the parameter to an operation or the return type from it



endpoint: "http://my.com/myservice"

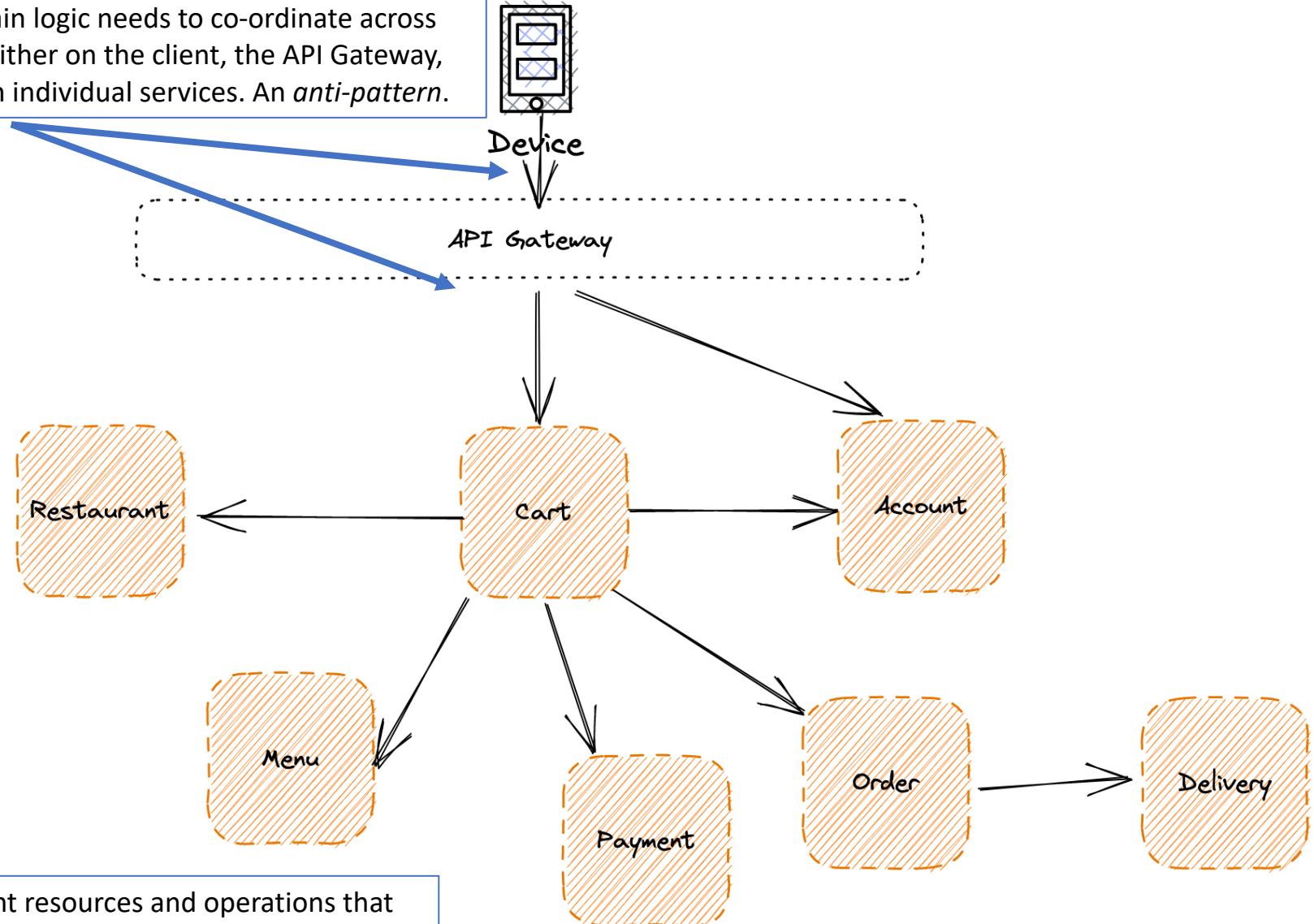
binding: SOAP

**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.

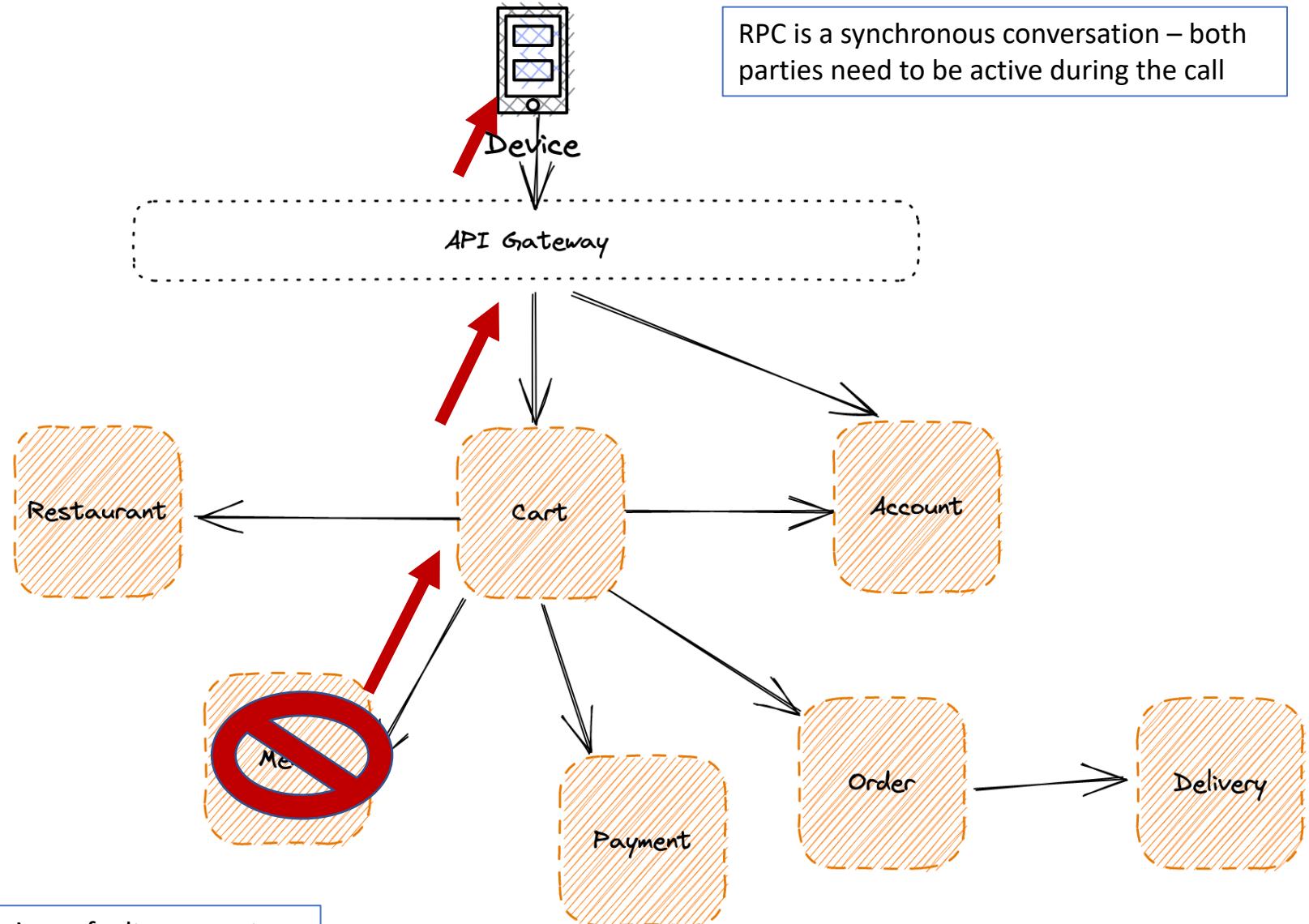
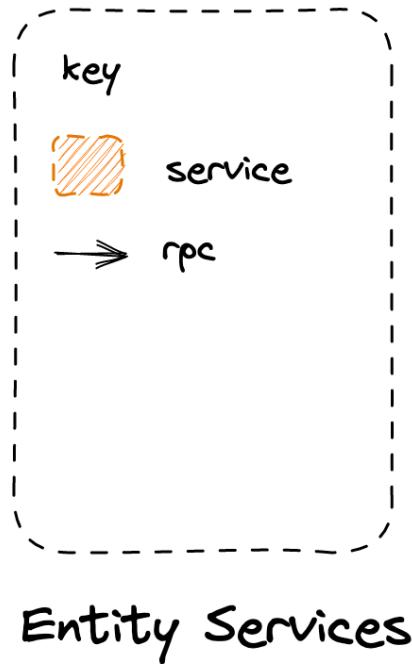
**Feature Envy:** Because domain logic needs to co-ordinate across these resources, it ends up either on the client, the API Gateway, or the Cart service and not in individual services. An *anti-pattern*.

key  
service → rpc

Entity Services



**SOA & OO:** We expose significant resources and operations that you can perform upon them, often CRUD.



# Agenda

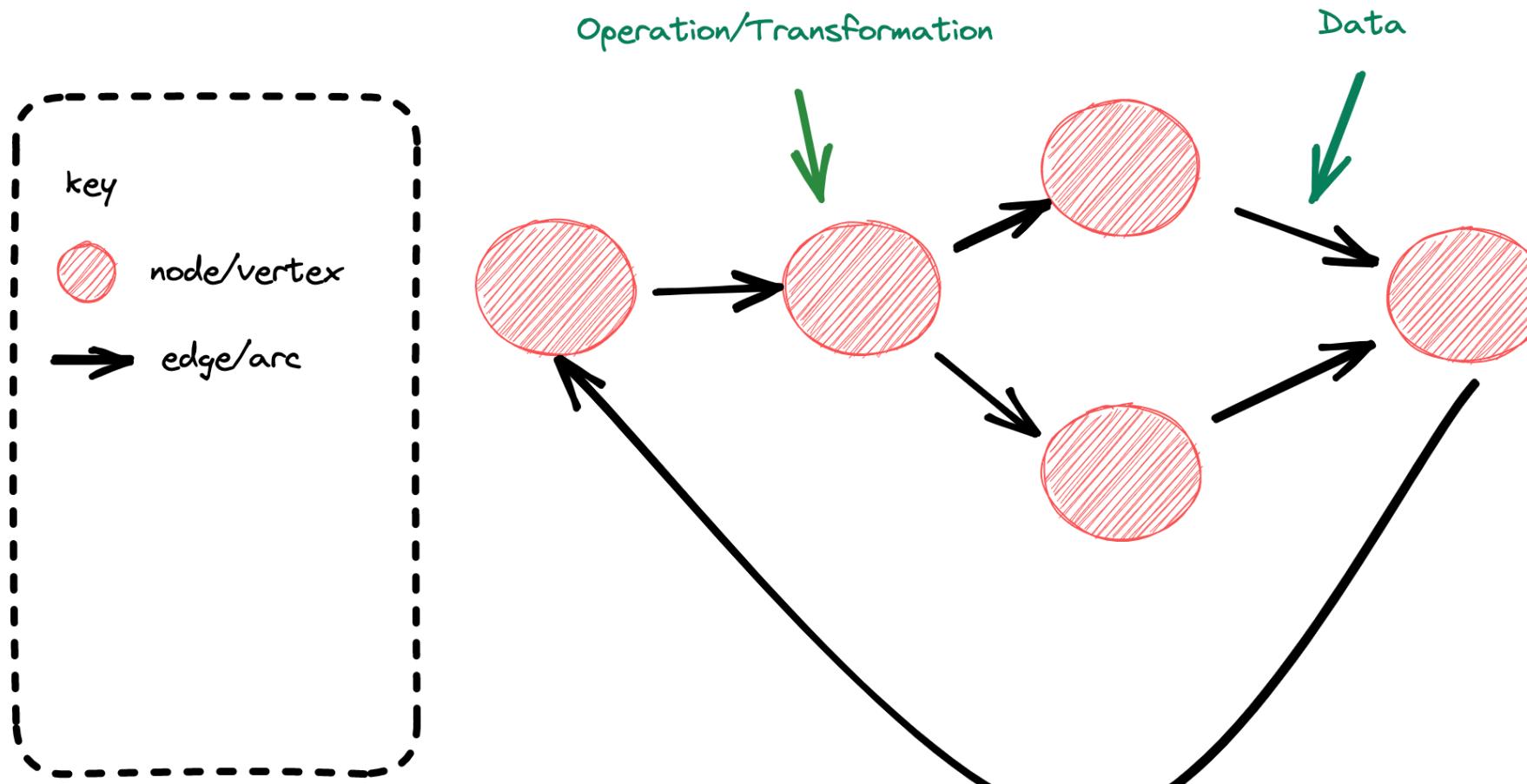
T: 10

- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Flow Based Programming
    - Actors
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness

# Agenda

T: 10

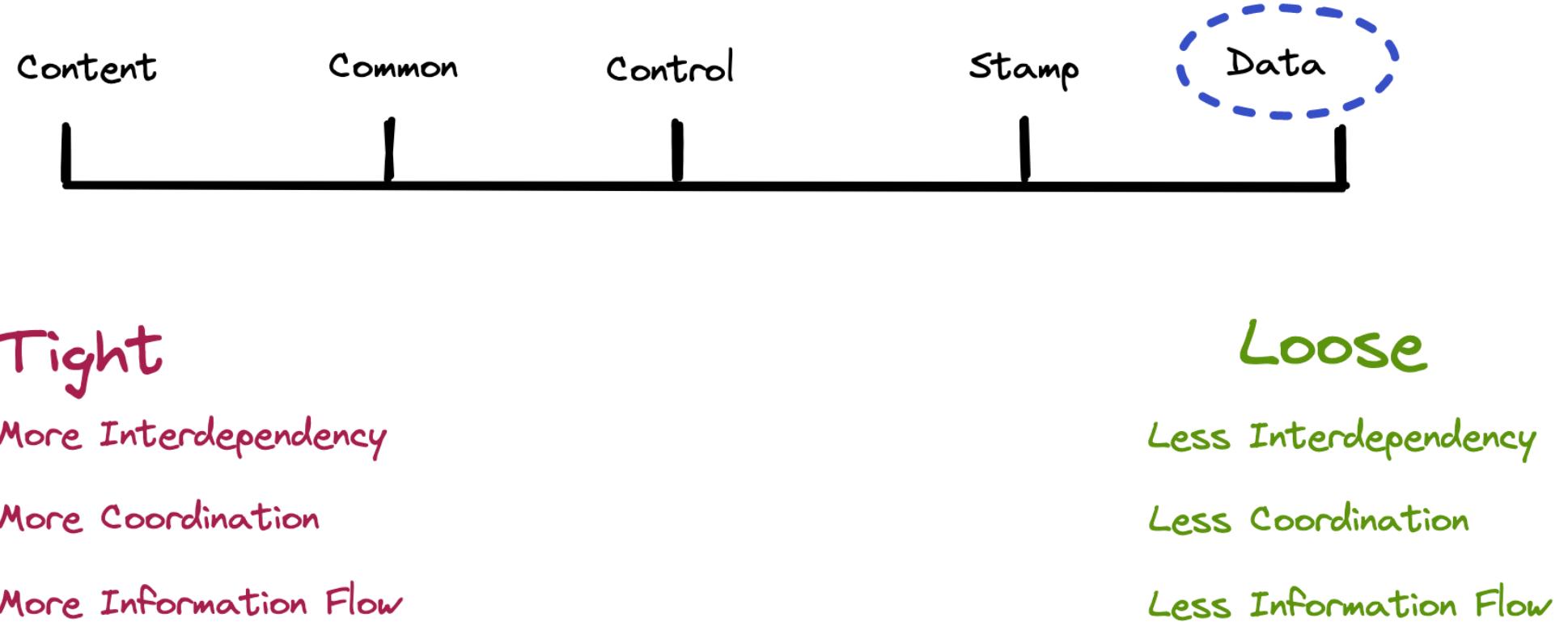
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Flow Based Programming
    - Actors
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness

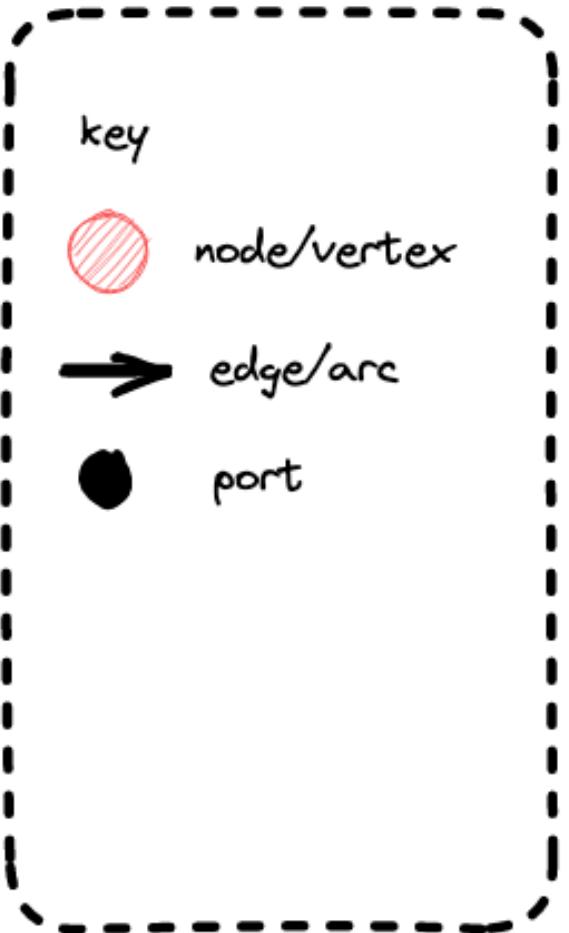


Jack Dennis et al.

**Data Flow Programming** conceptualizes a program as a directed graph where operations are represented as nodes which are connected via arcs through which data flows. A node performs an operation when input data is available.

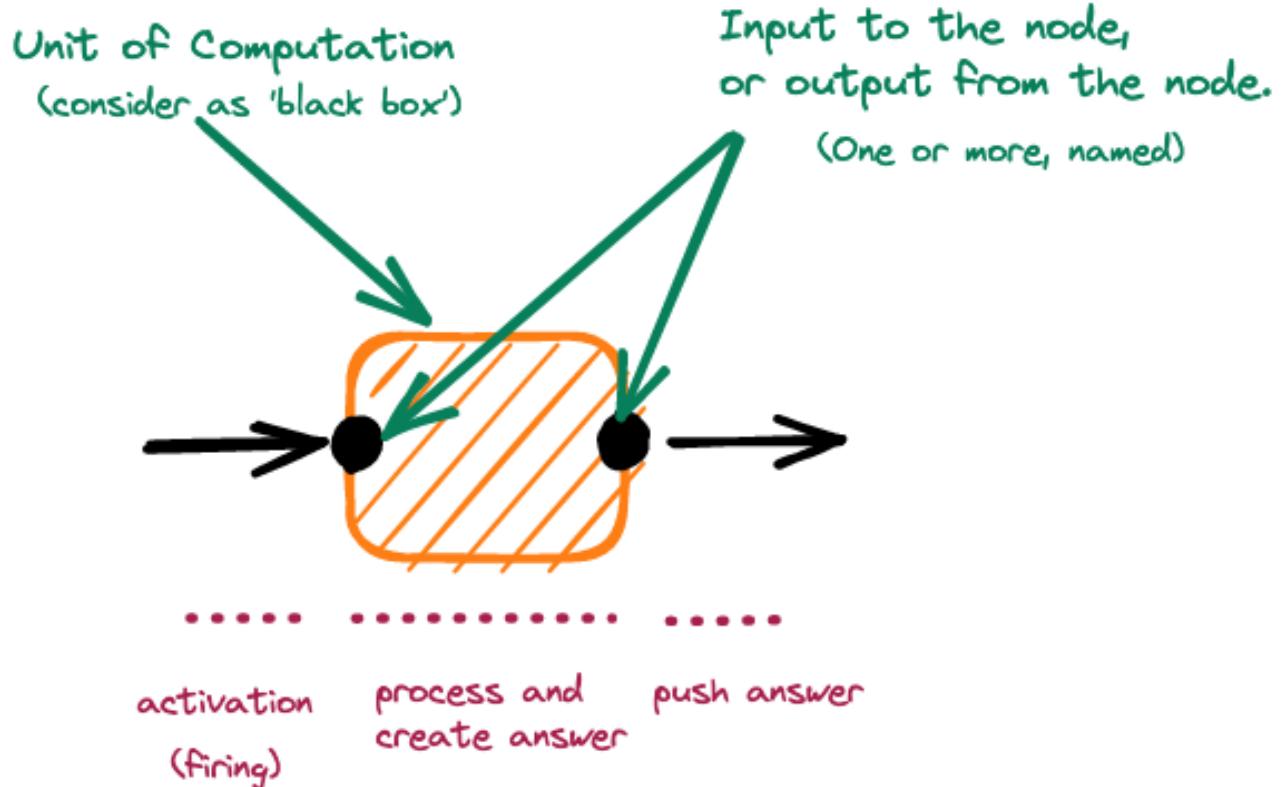
# Coupling





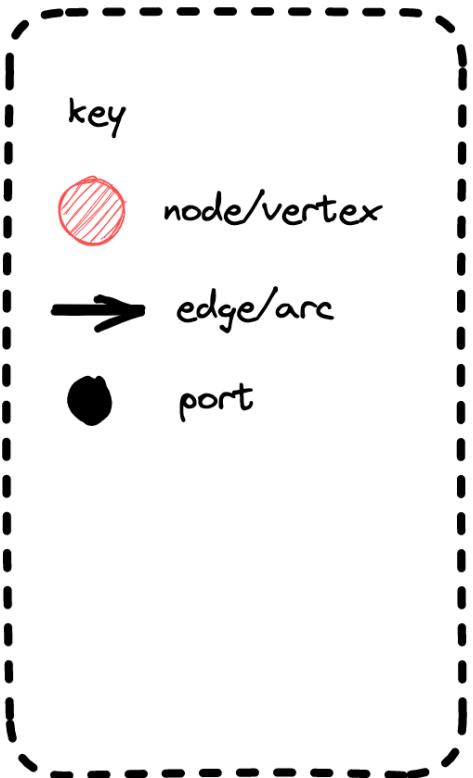
## Dataflow Programming

Jack Dennis et al.



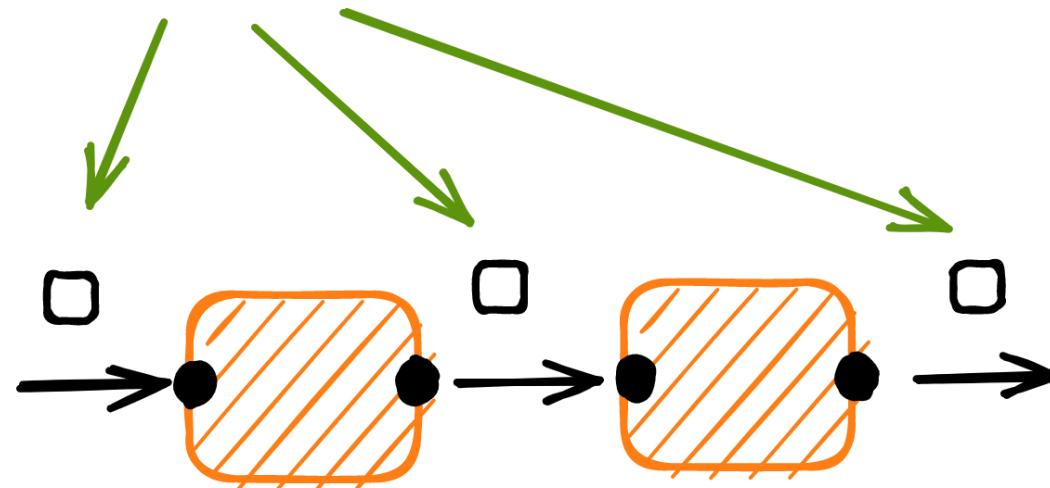
## Data Packet

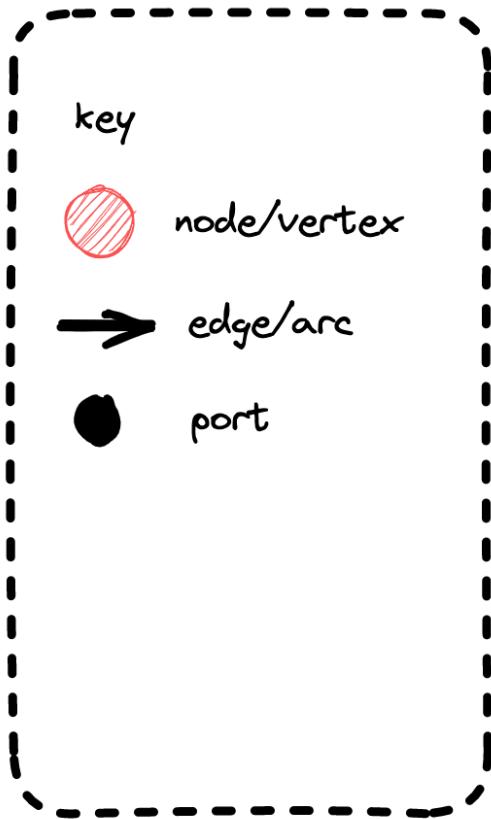
primitive/compound, structured/unstructured values, and even include other data packets



Dataflow Programming

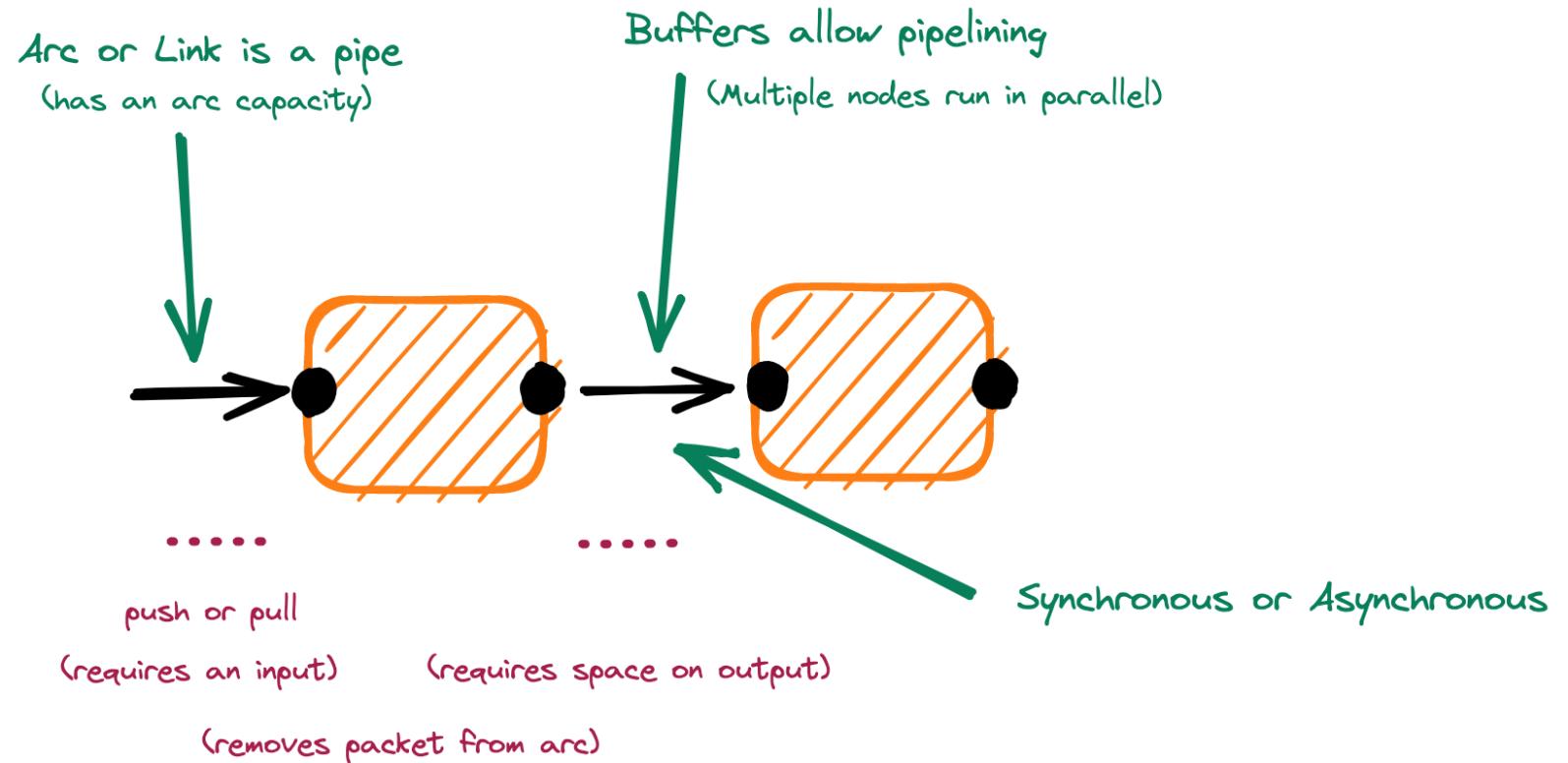
Jack Dennis et al.





## Dataflow Programming

Jack Dennis et al.



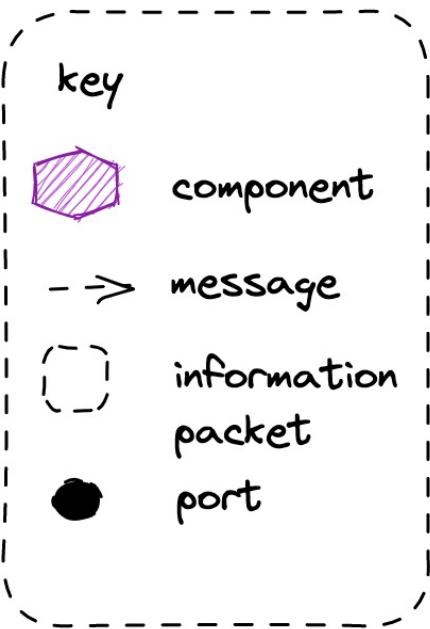
**Node Lifetime:** In 'classic' dataflow programming the lifetime of a node is from activation when there is work to do (push) or work is requested (pull) until it has pushed the answer.

**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**.

# Agenda

T: 25

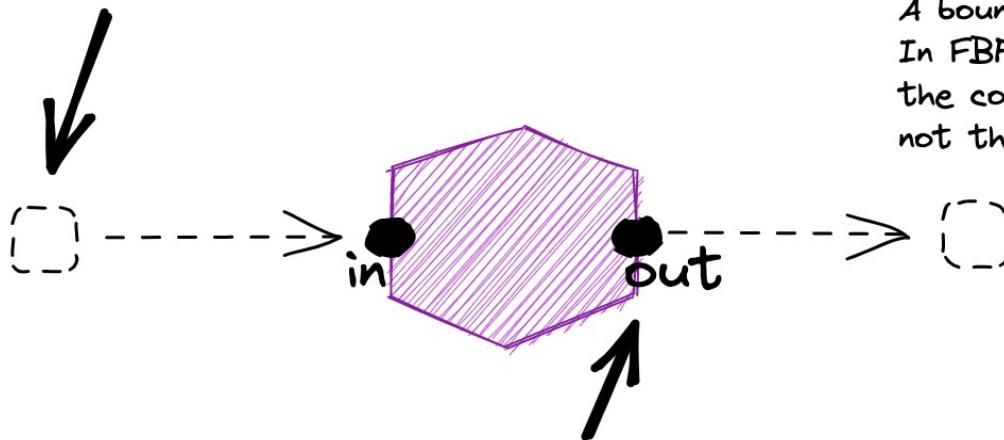
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Flow Based Programming
    - Actors
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness



## Flow Based Programming (J. Paul Morrison)

### Information Packet (IP)

An independent structured piece of information with a defined lifetime.

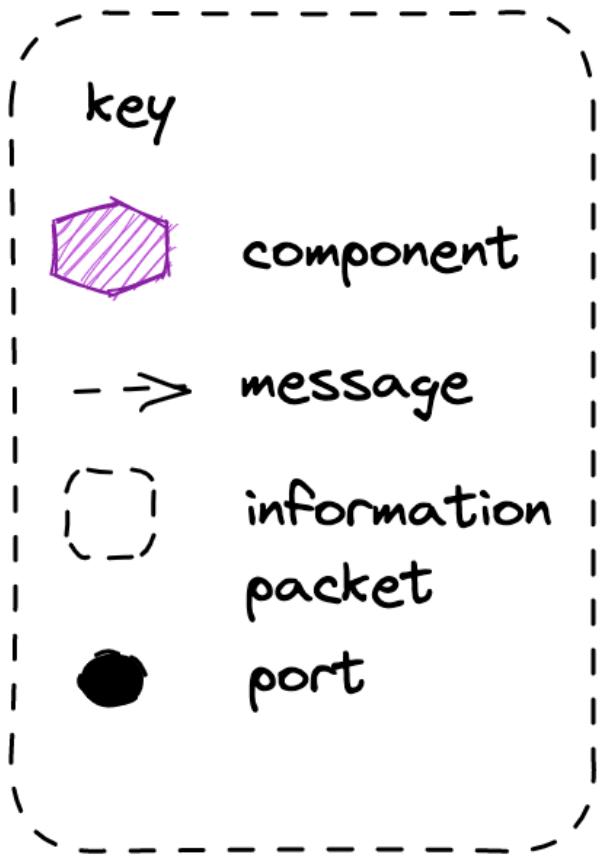


### Port

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

### 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



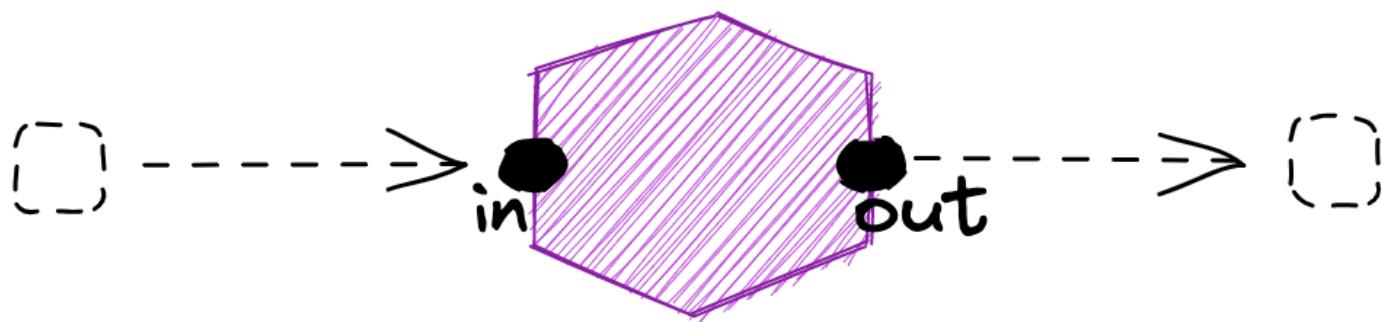
# Flow Based Programming

(J. Paul Morrison)

Packet Lifetime



Packet Lifetime



Process & Wait

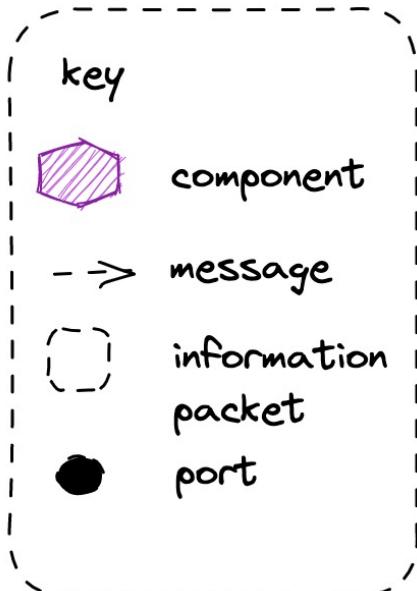


**Node Lifetime:** In flow base programming the a node can remain running whilst there is work on an input queue, can can suspend instead of terminate if there is no work on its connector. **We may receive an explicit signal to indicate we should stop.**

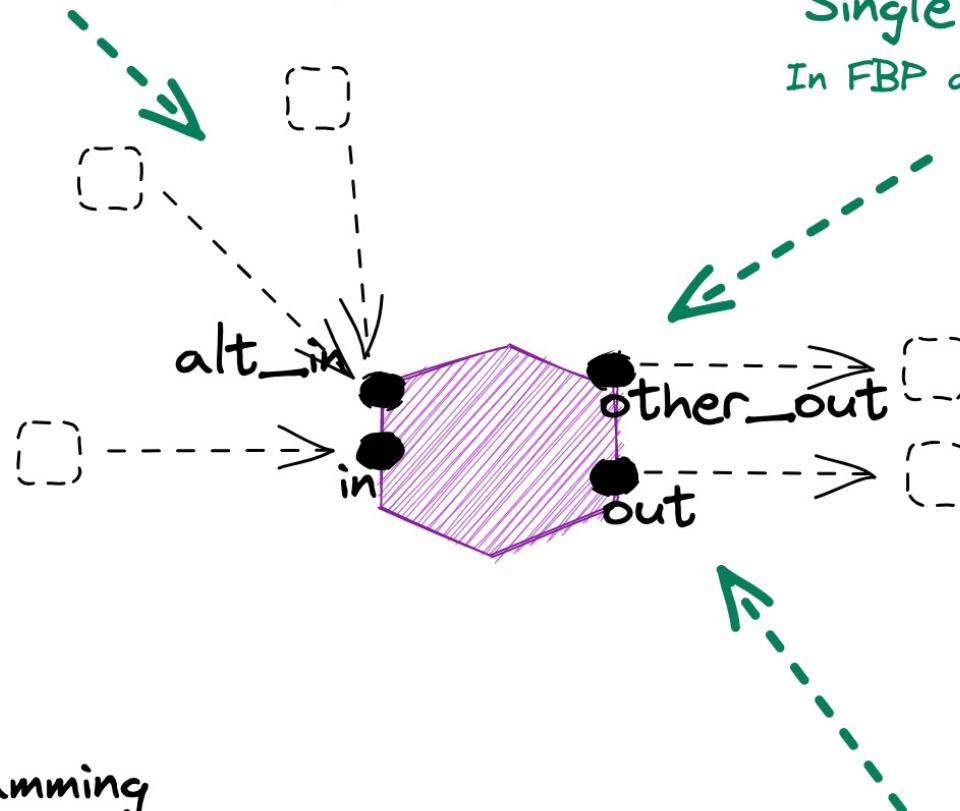
**Capacity:** In flow-based programming a component can test to see if it can send to out (**backpressure**), and if not decide whether to ignore (**load shedding**) or even halt.

## Multiple Writers

In FBP an in port can have multiple writers



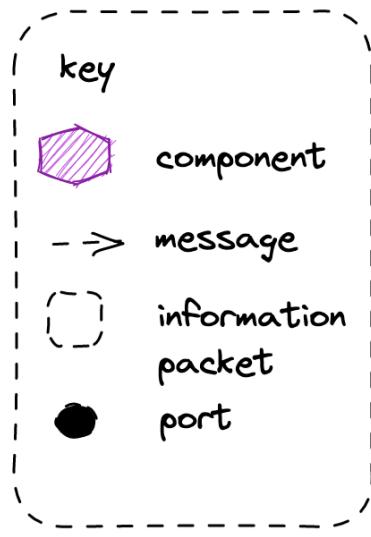
Flow Based Programming  
(J. Paul Morrison)



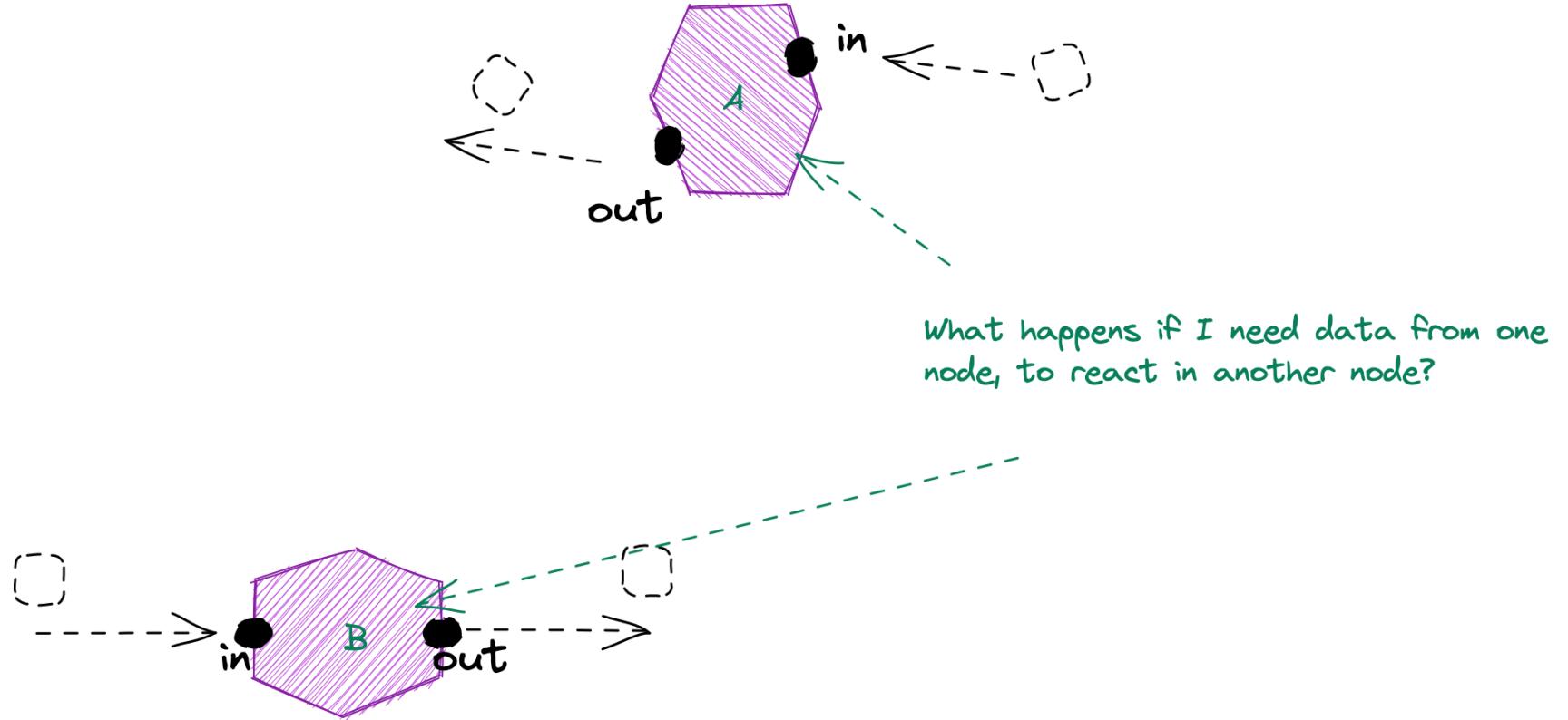
## Single Writer

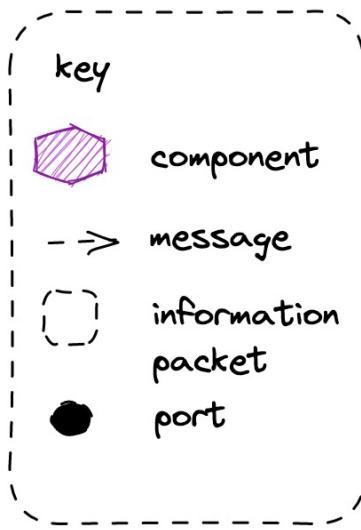
In FBP an out port is single writer

Multiple In/Out Ports  
In FBP we can have multiple  
in or out ports



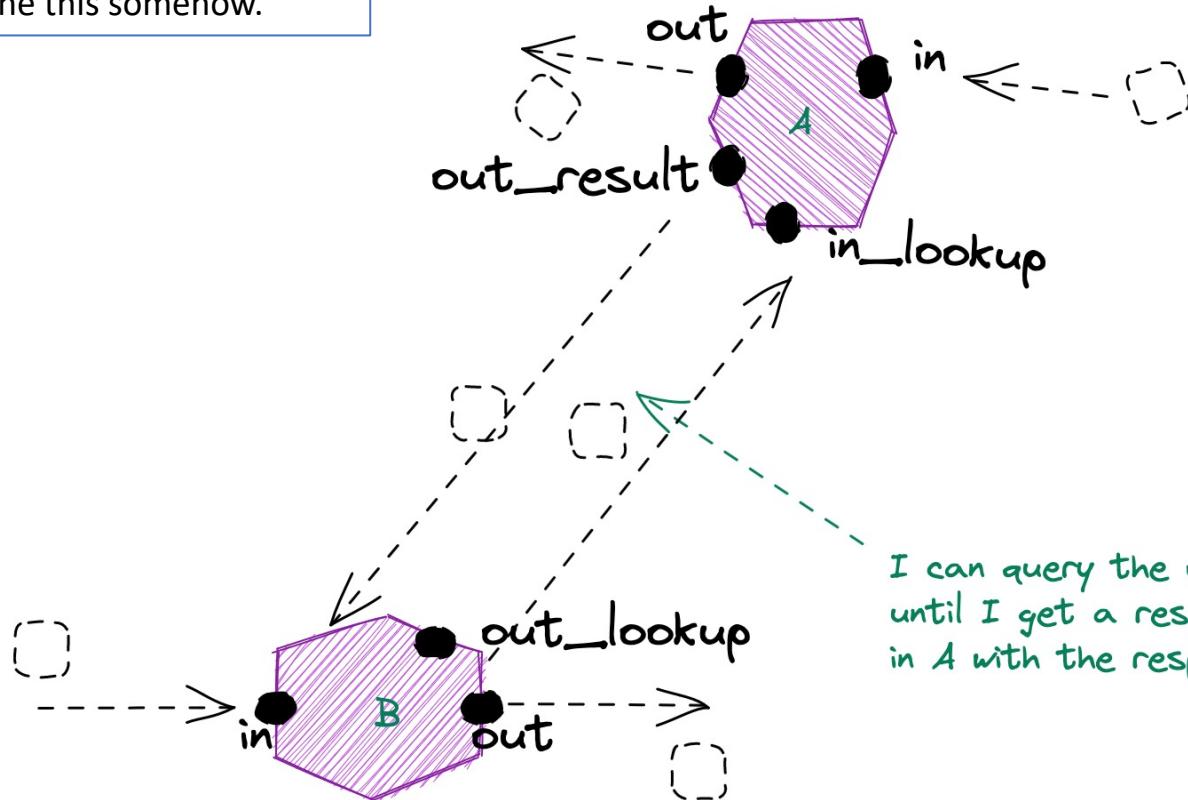
## Flow Based Programming (J. Paul Morrison)



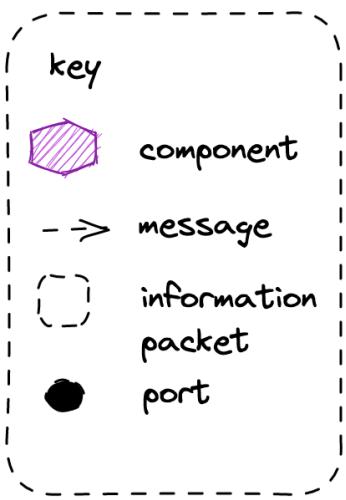


Flow Based Programming  
(J. Paul Morrison)

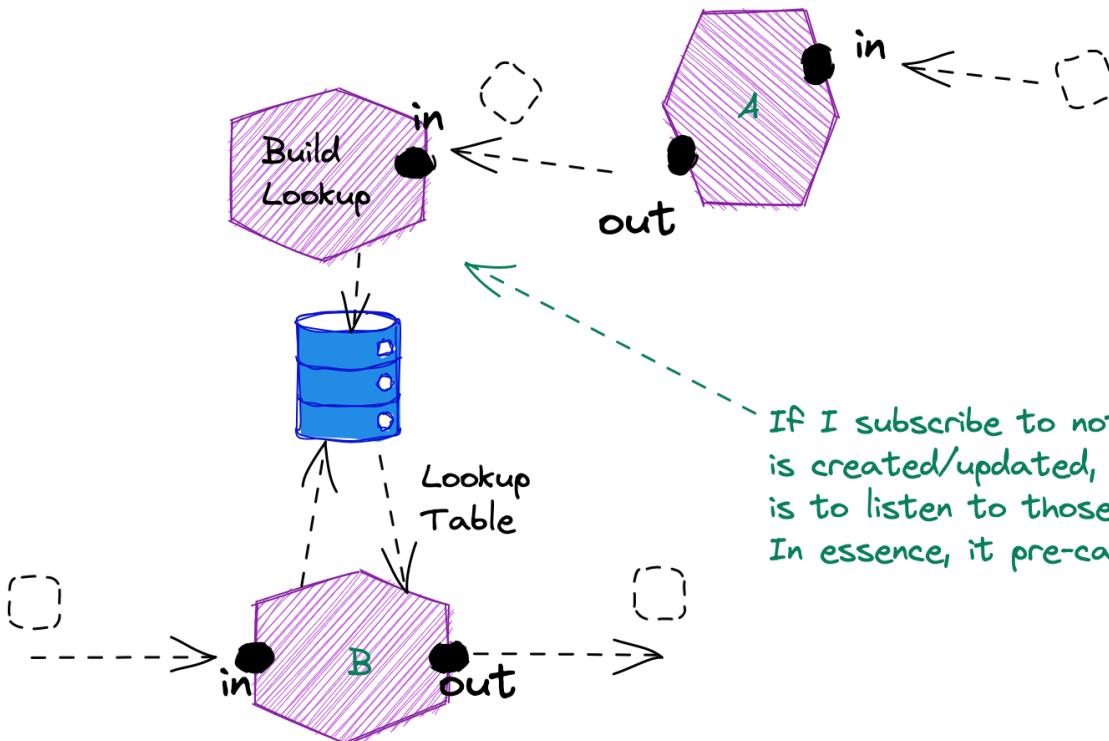
**Walk of Shame:** The trouble is that we may keep asking, so it makes sense to cache this somehow.



I can query the other component, pausing B until I get a response from A, then continuing in A with the response.



## Flow Based Programming (J. Paul Morrison)

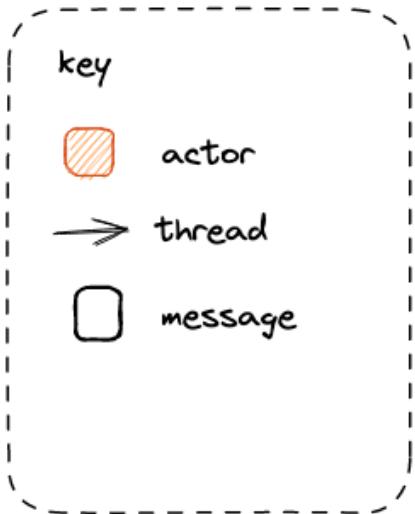


If I subscribe to notifications from A, as new data is created/updated, I can write a node whose responsibility is to listen to those messages and builds a lookup table for B. In essence, it pre-caches the results for B.

# Agenda

T: 20

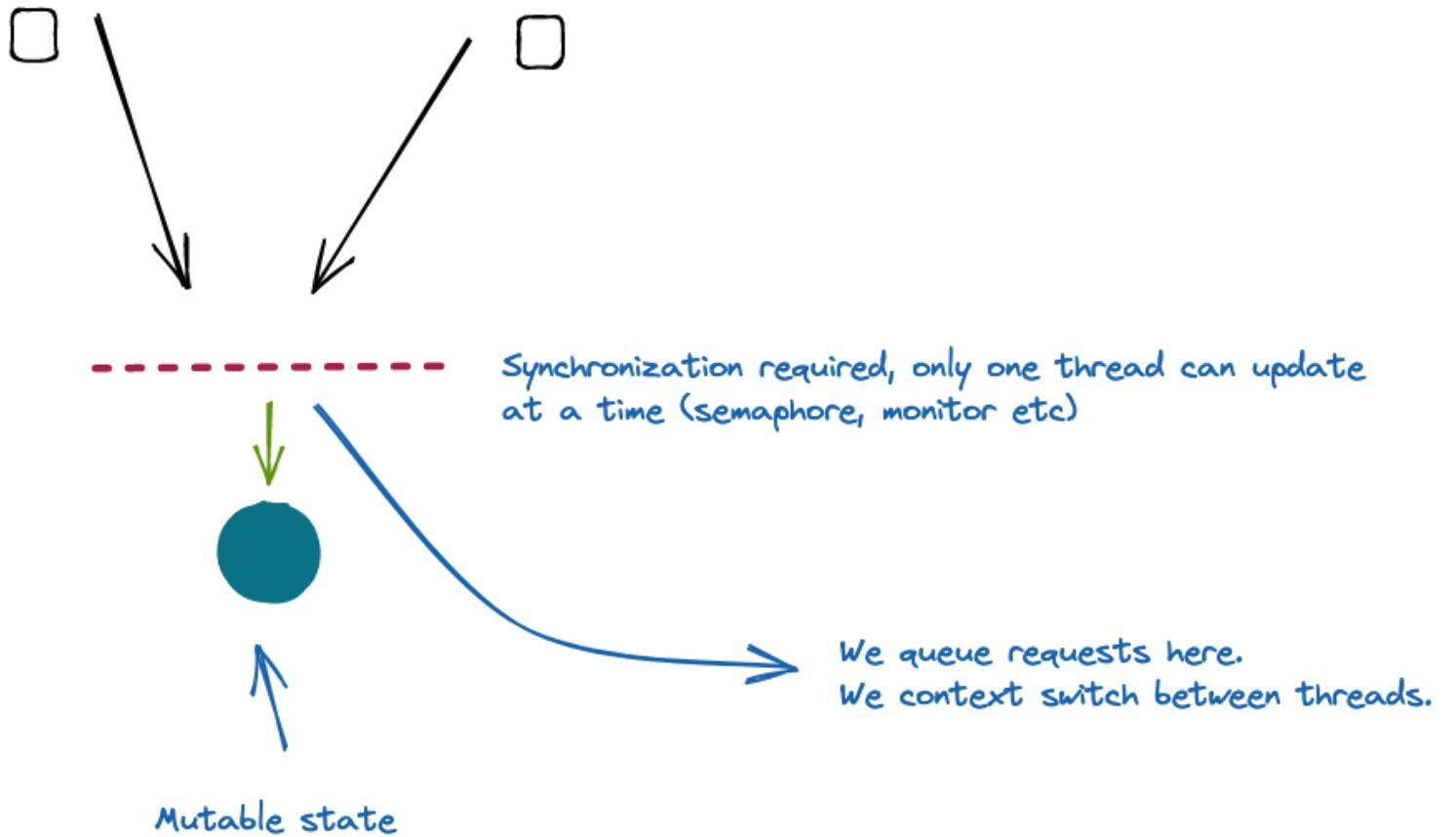
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness

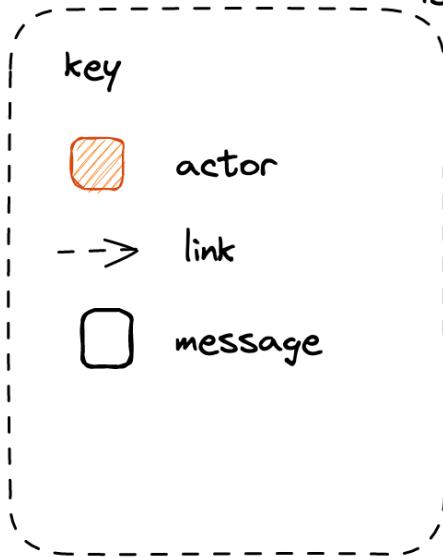


## No Actors Model

Edsger W. Dijkstra

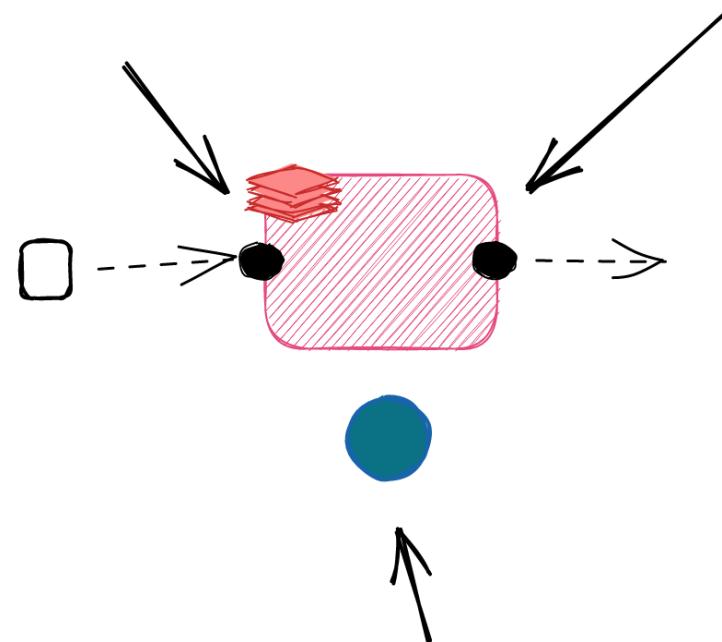
Requests to update state from different threads





## Actor Model

Carl Hewitt

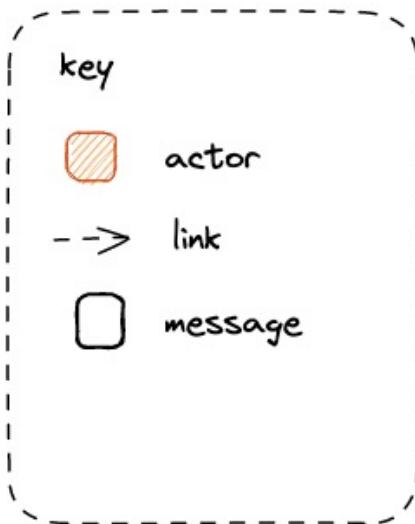


An actor owns mutable state and only the actor can update it

In an actor the port is a mailbox of work for that actor. We partition by the address of the actor who the mail is for.

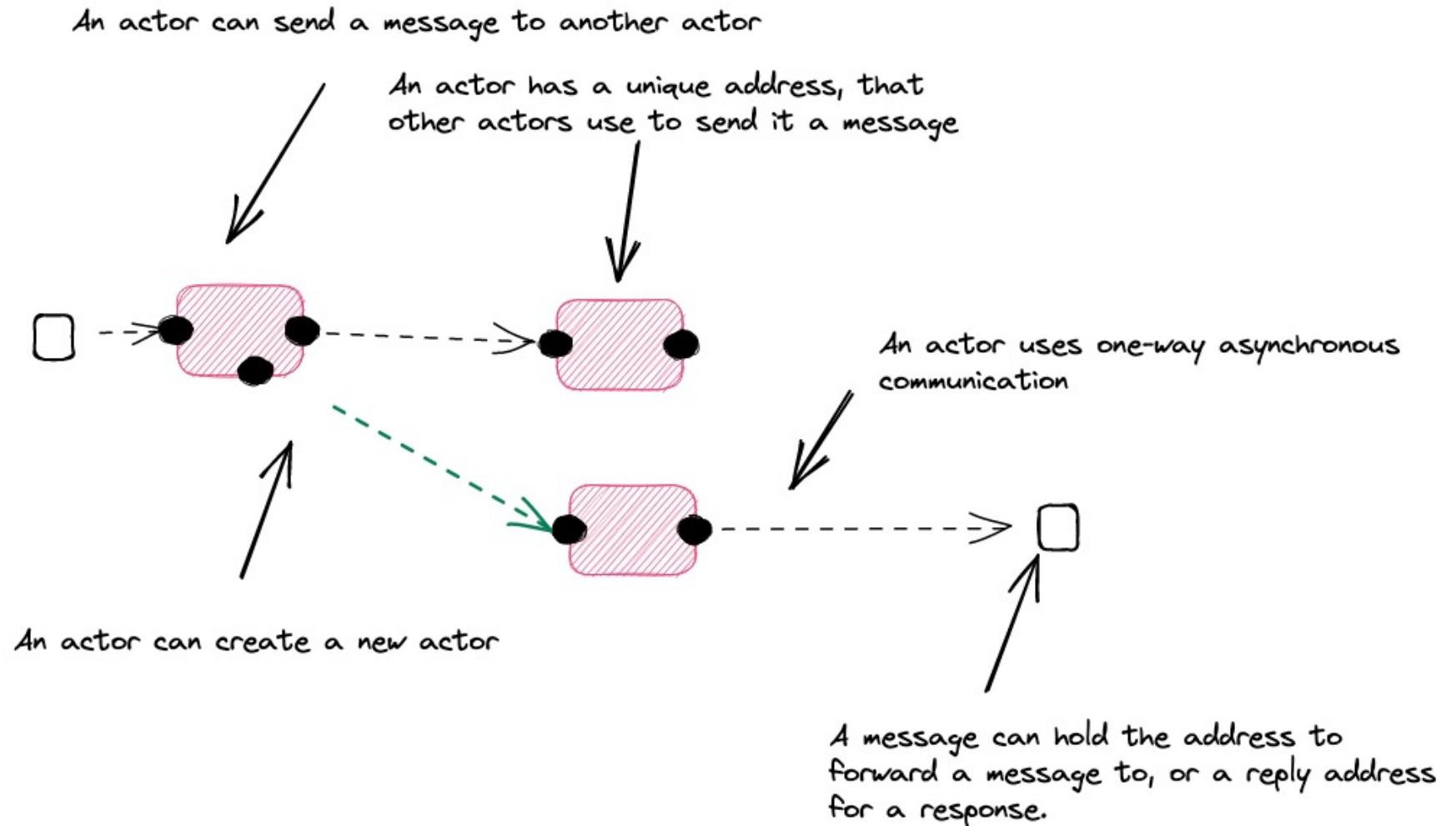
The actor is single-threaded, so it needs the mailbox to allow it to process one message at a time.

**Concurrency:** Hewitt's motivation is to provide an alternative to synchronization primitives.



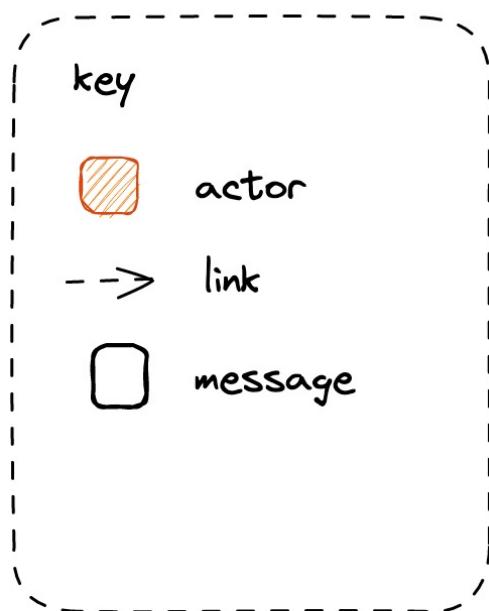
## Actor Model

Carl Hewitt

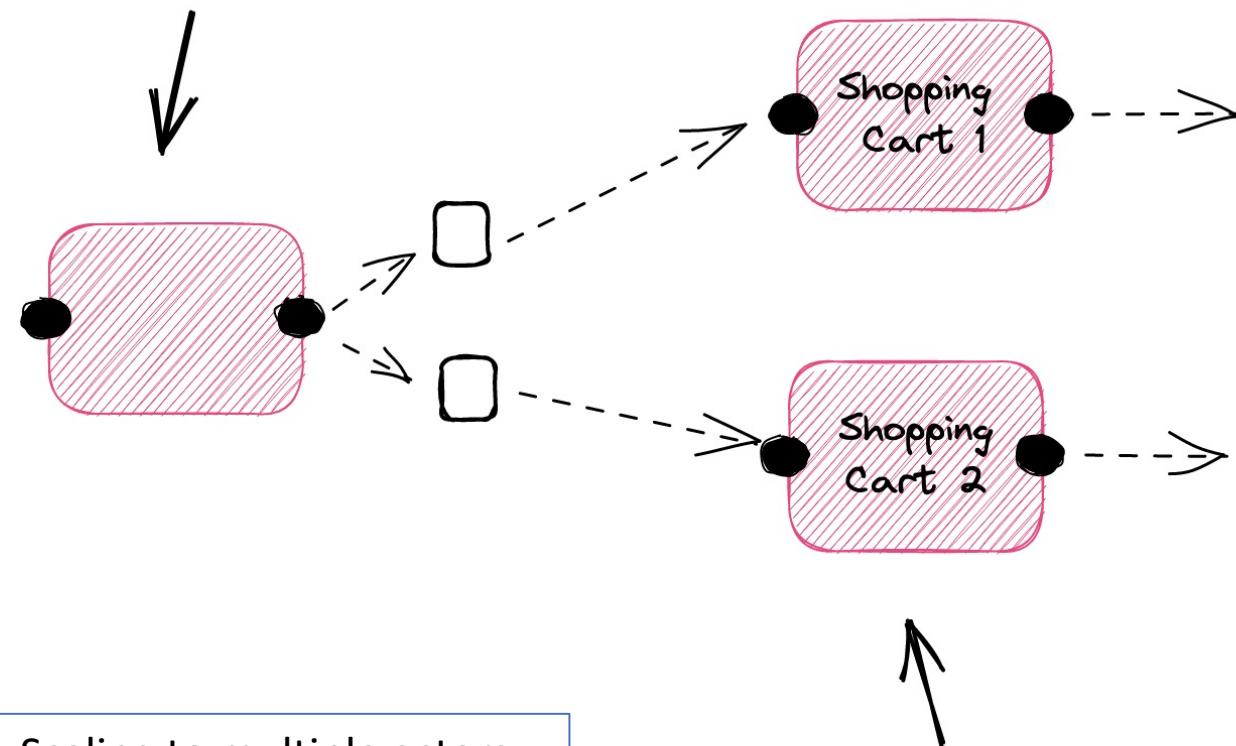


## Actor Model

Carl Hewitt



We create child actors, so we know the address that corresponds to the state we want to send a message to



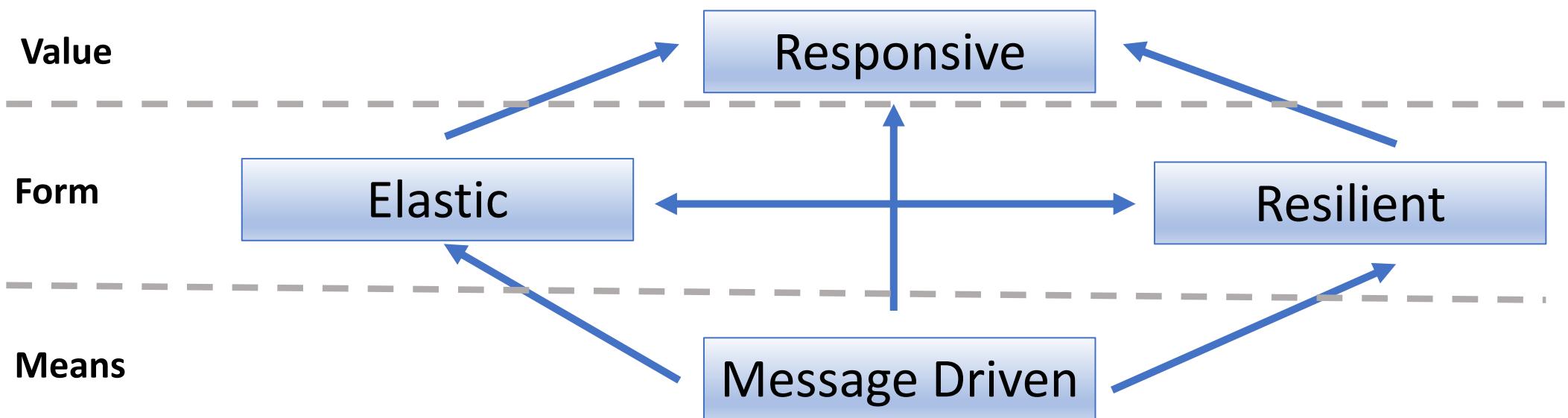
**Scaling:** Scaling to multiple actors requires partitioning messages for actors i.e. all messages for the same mutable to a single actor

We partition state amongst actors, and each actor receives messages for its state.

# Agenda

T: 30

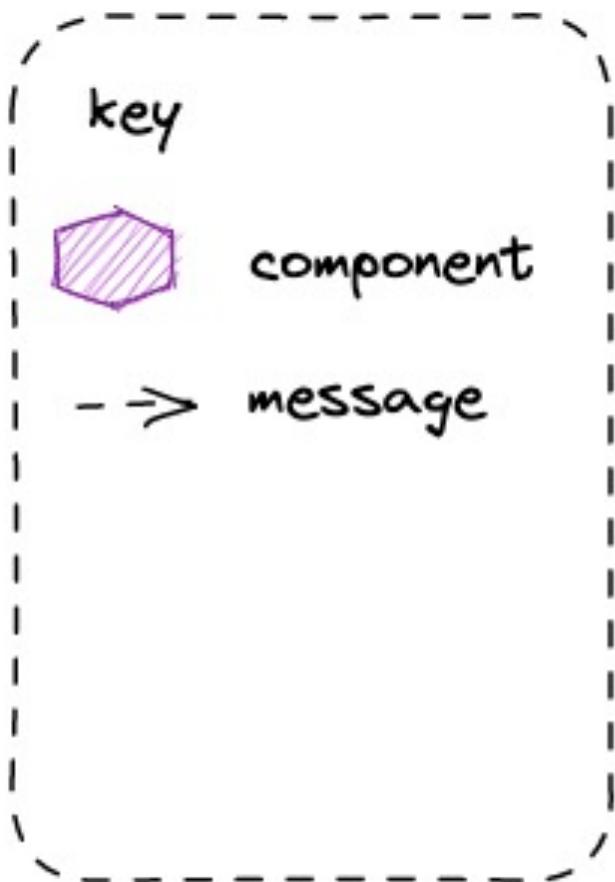
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - **Reactive Systems**
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness



# Agenda

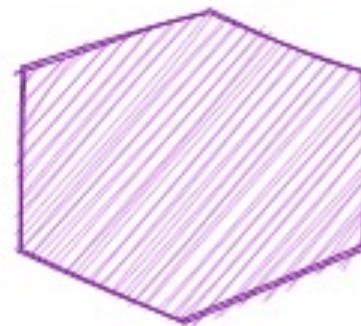
T: 35

- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - Reactive Systems
    - Message Passing
    - Resilience
    - Elasticity
    - Responsiveness



## Message Passing

Incoming Message

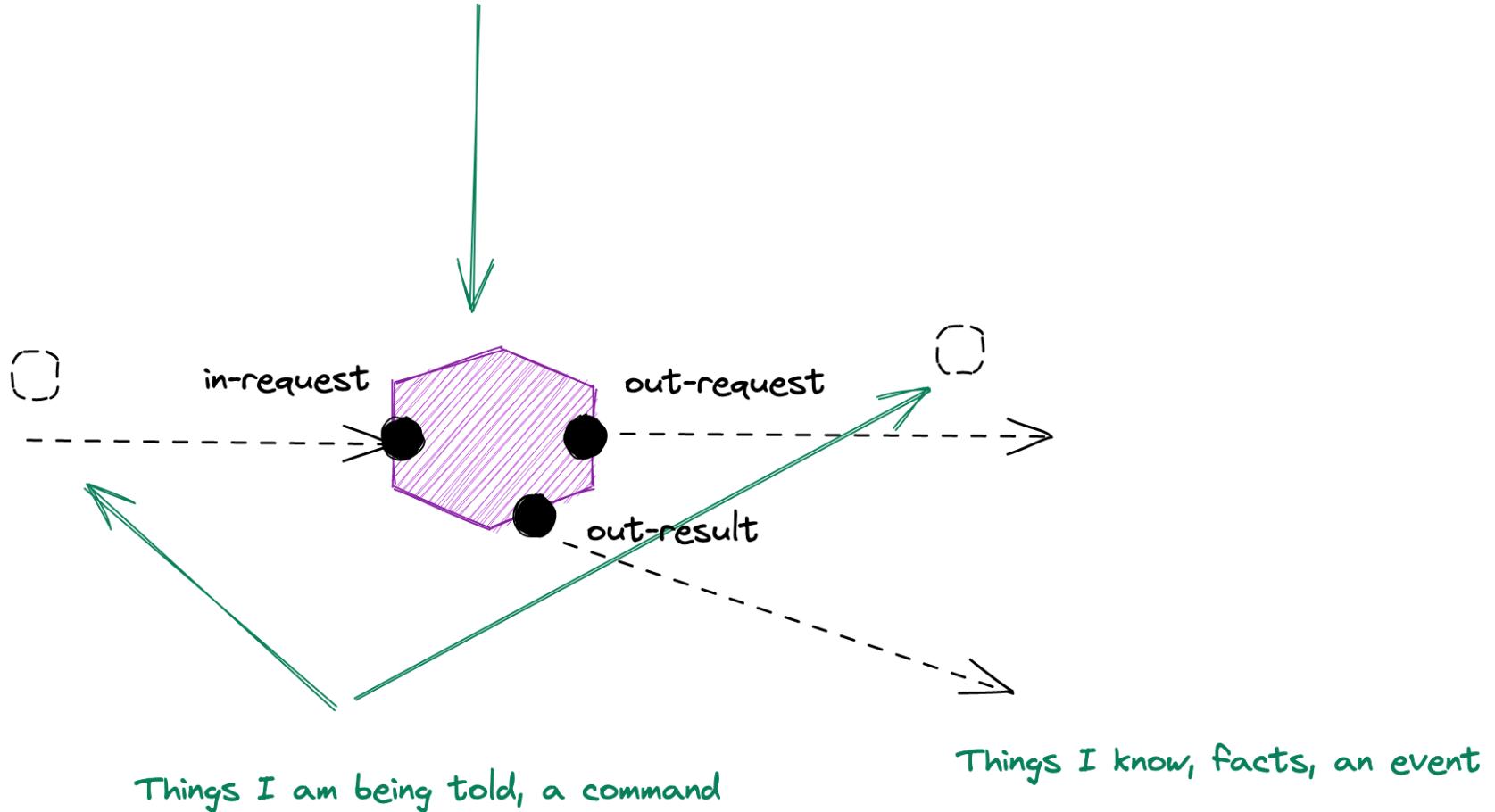
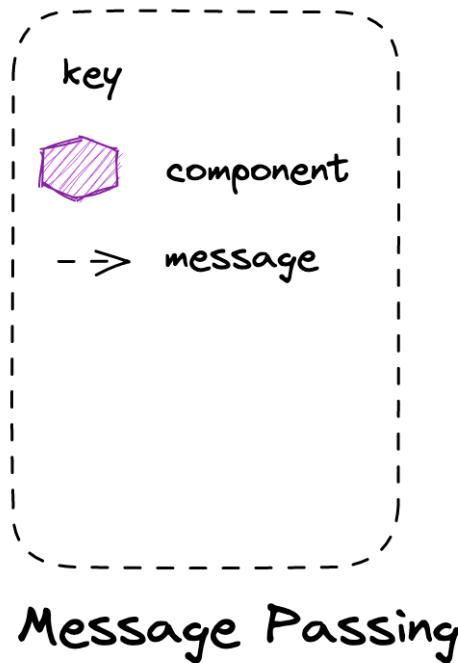


Outgoing Message

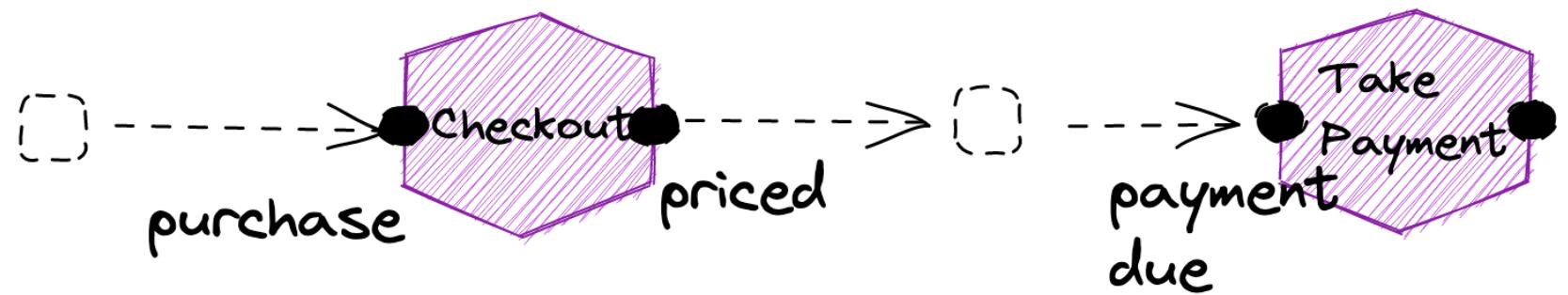
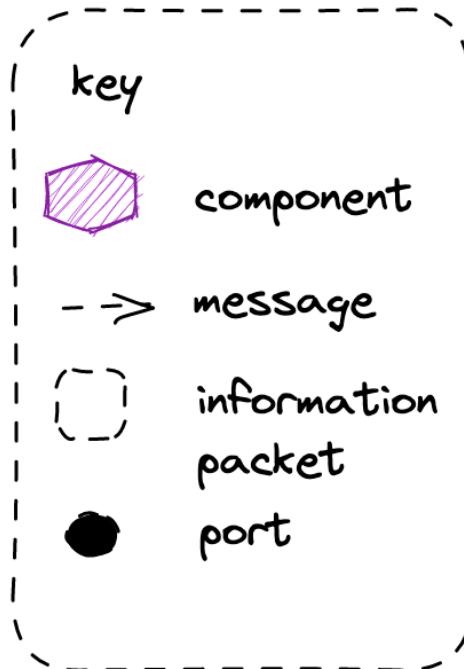


**Message Passing** is an asynchronous method of communication – both parties do not have to be simultaneously present for communication to occur, instead mail is delivered to ‘mailbox’ of some form for later retrieval.

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

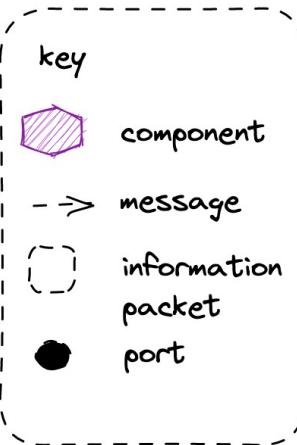


**Coordinate Dataflow** A reactive *principle* where we “orchestrate a continuous steady flow of information” focusing on division by behavior, not structure



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

## Autonomous Component Pat Helland

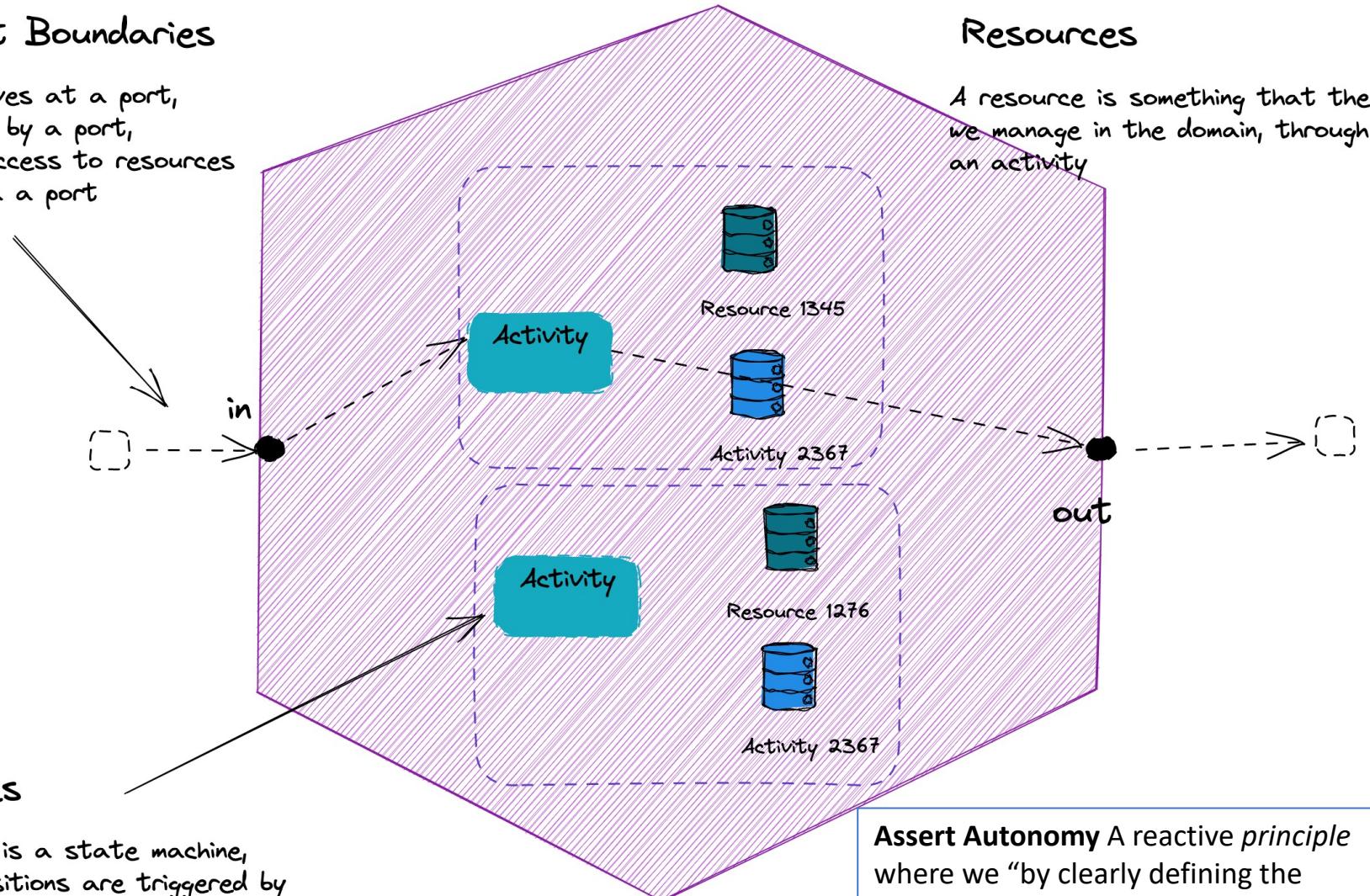


## Explicit Boundaries

Work arrives at a port,  
or leaves by a port,  
and all access to resources  
is only via a port

## Activities

An activity is a state machine,  
whose transitions are triggered by  
the arrival of messages. It tracks  
messages in and out, current state.

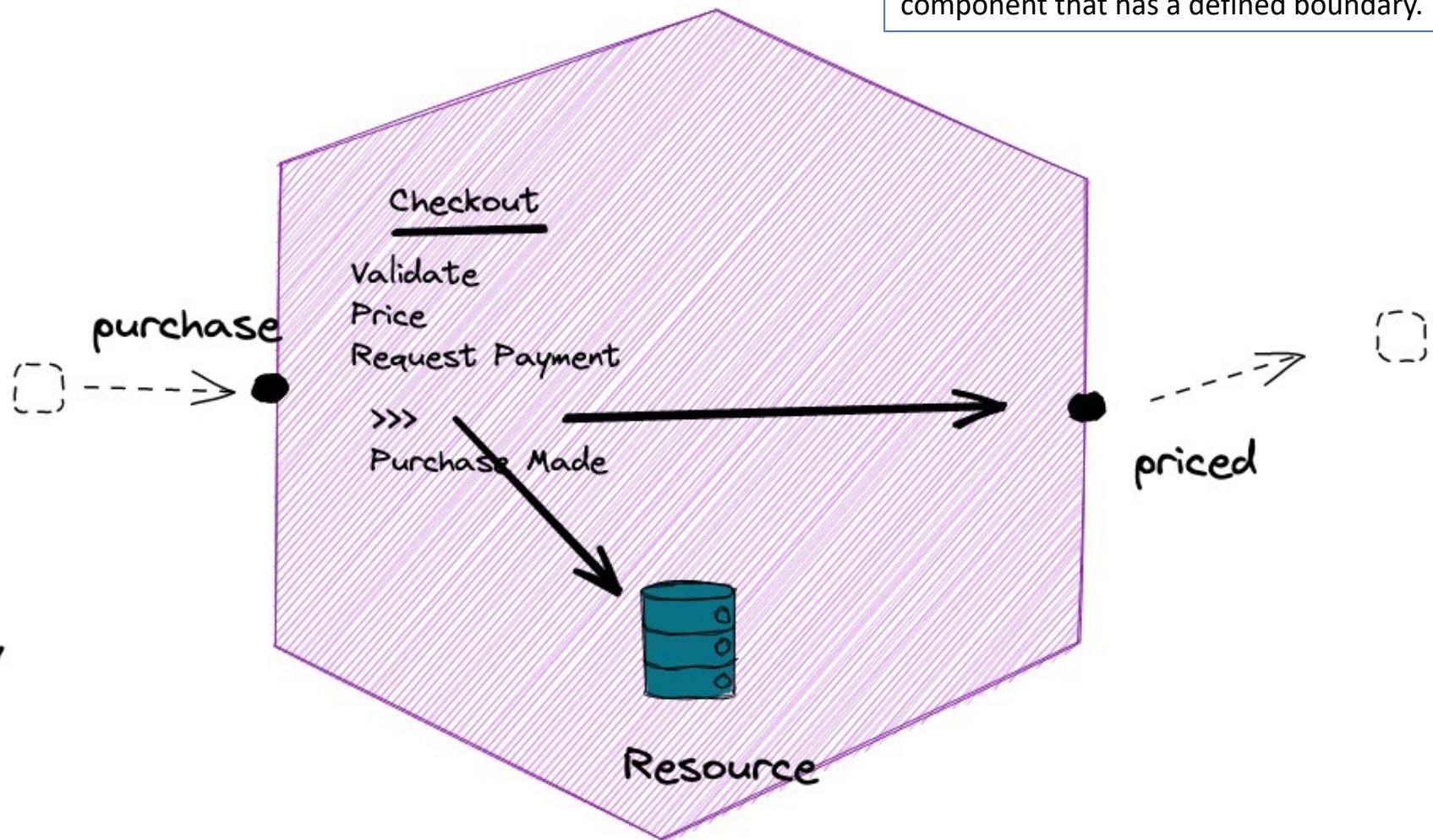
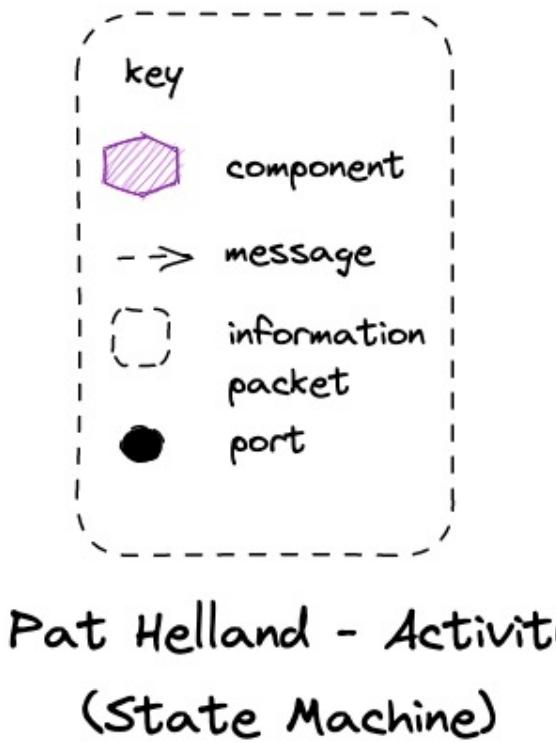


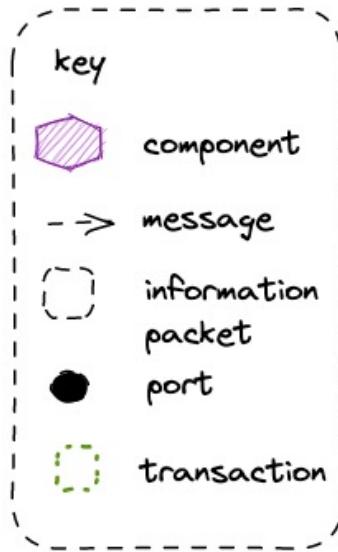
## Resources

A resource is something that we manage in the domain, through an activity

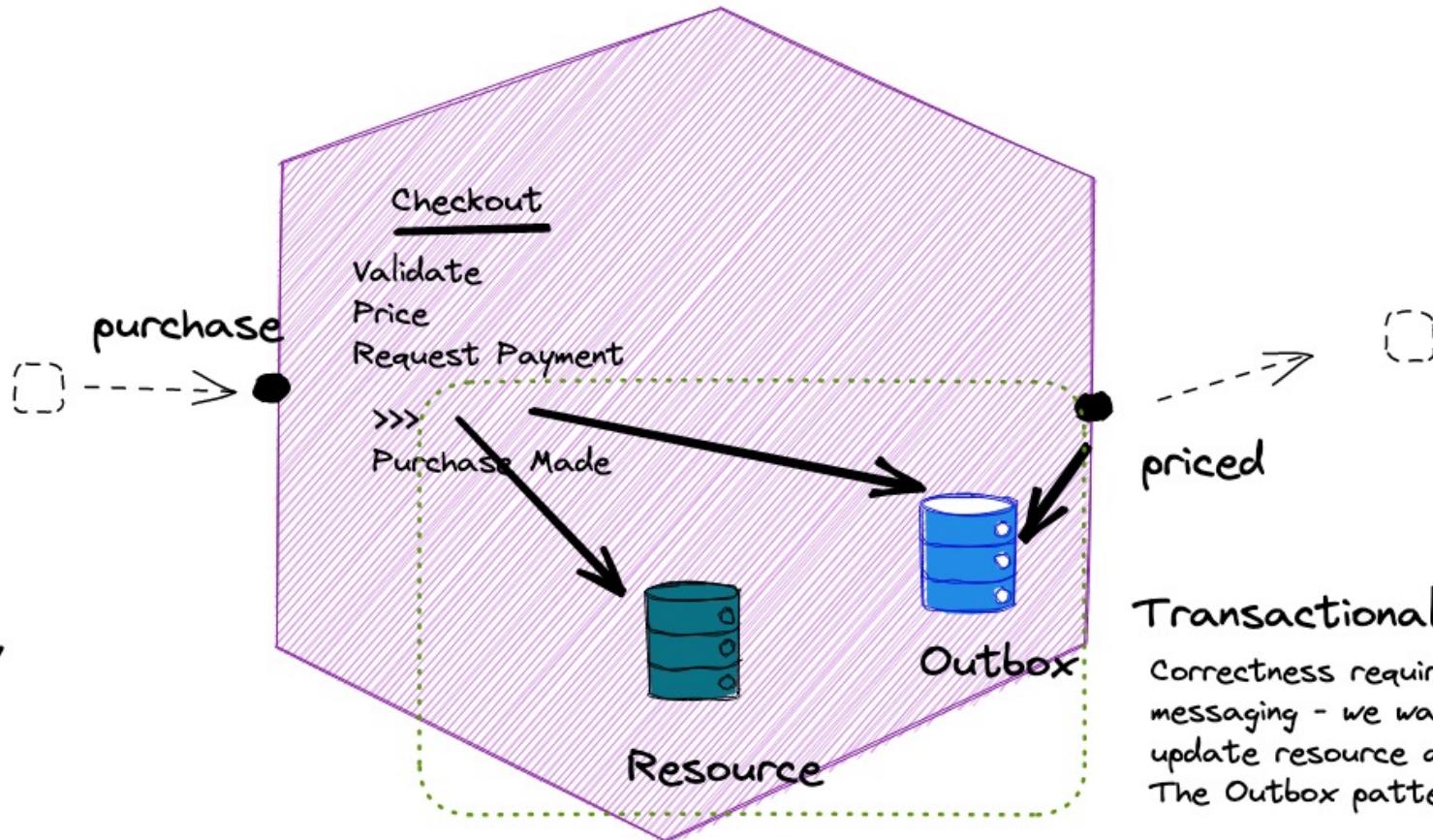
**Assert Autonomy** A reactive principle where we “by clearly defining the component boundaries, who owns what data and how the owners make it available”

**Isolate Mutations** A reactive pattern where we “contain and isolate mutable state”. The mutable state lives within a component that has a defined boundary.



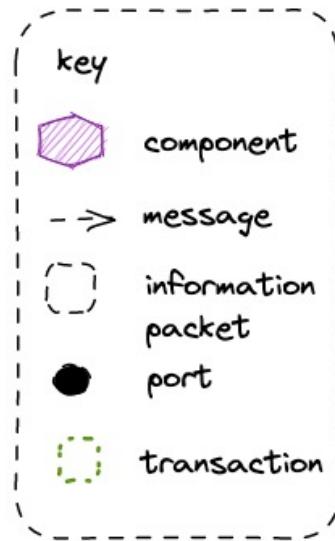


Pat Helland - Activity  
(State Machine)

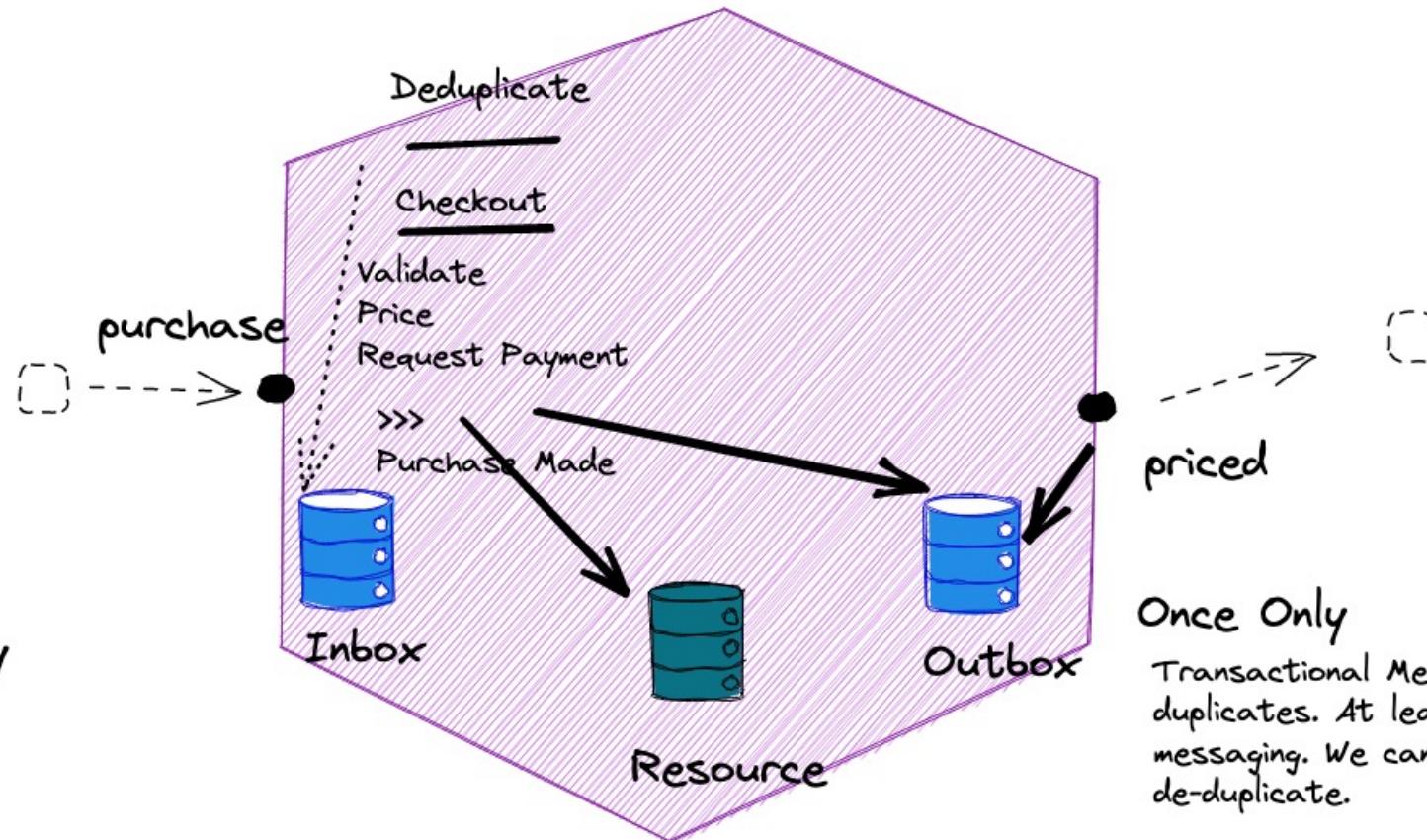


### Transactional Messaging

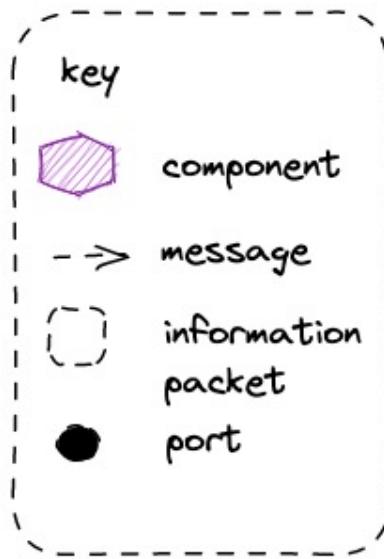
Correctness requires transactional messaging - we want to ensure we update resource and send message  
The Outbox pattern offers this



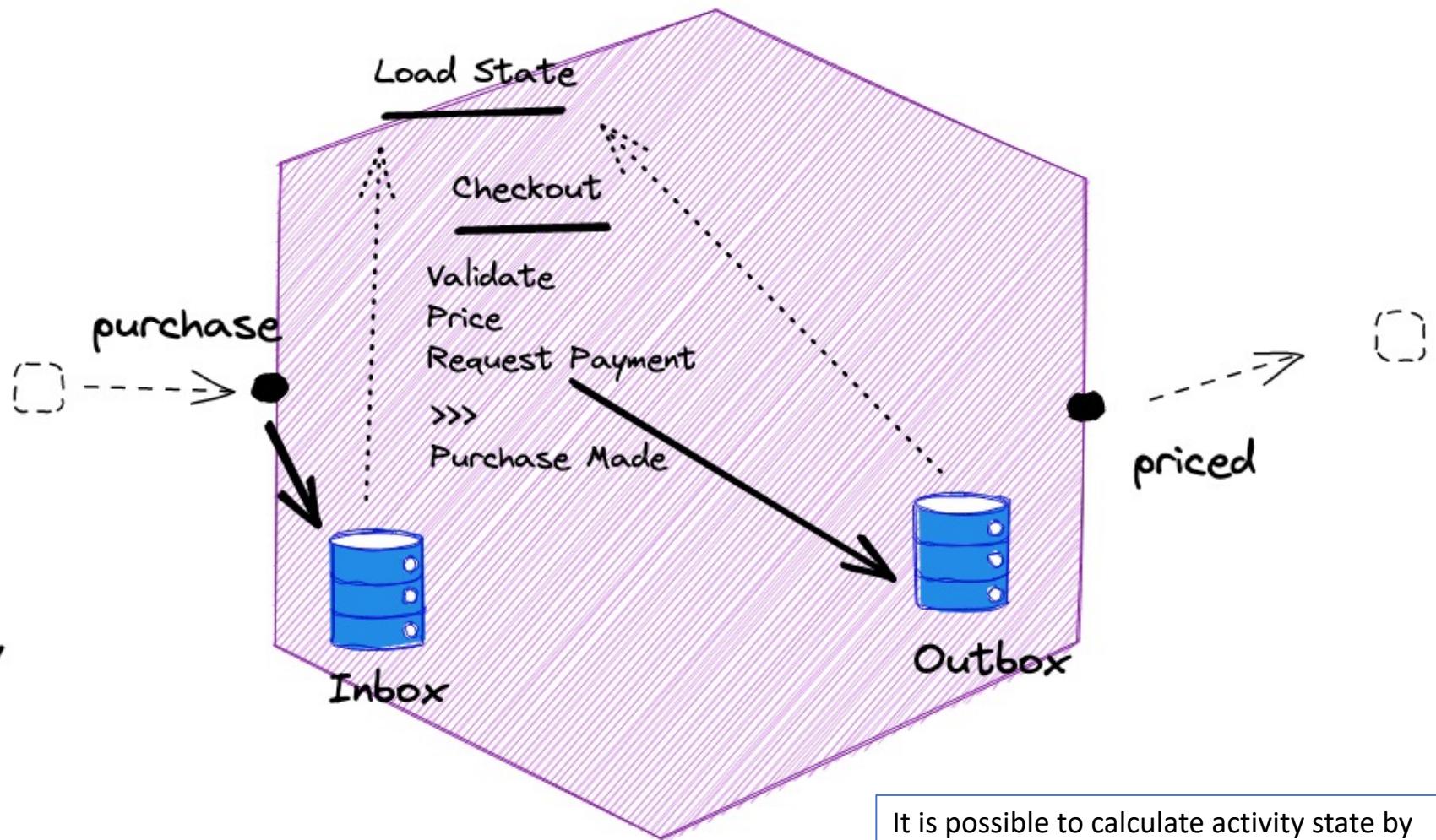
Pat Helland - Activity  
(State Machine)



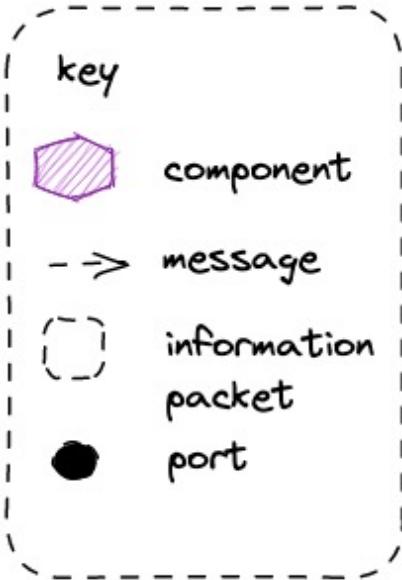
Transactional Messaging will send us duplicates. At least once, guaranteed messaging. We can use an Inbox to de-duplicate.



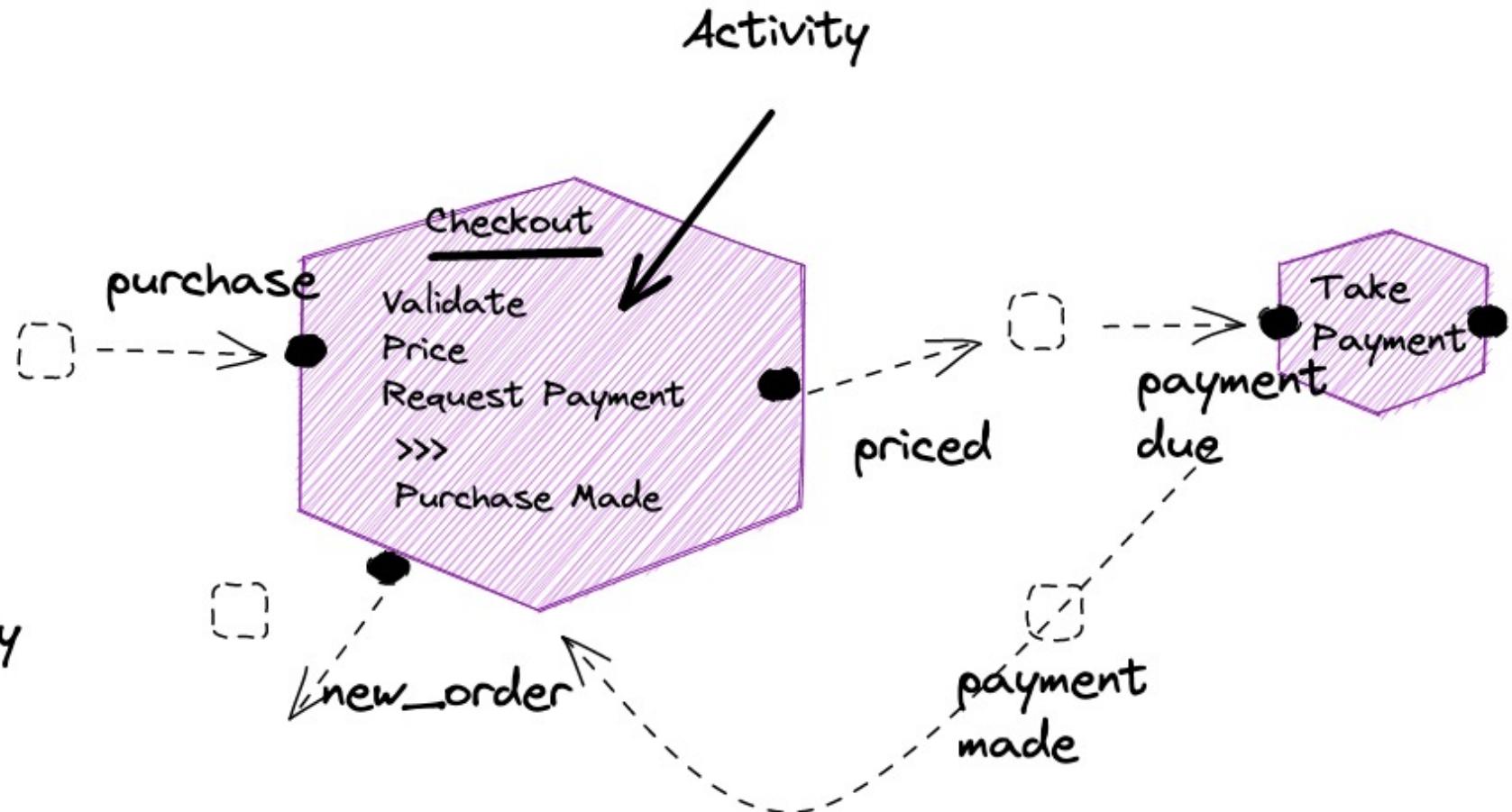
## Pat Helland - Activity (State Machine)

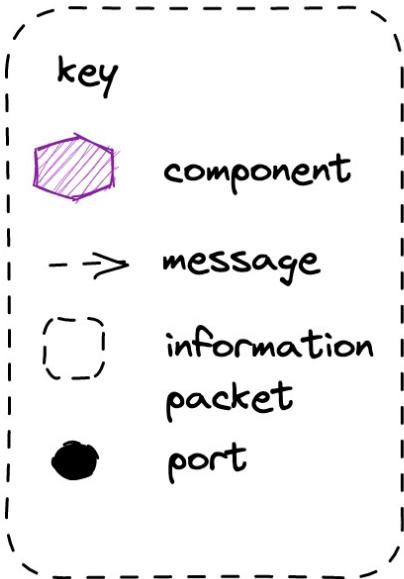


It is possible to calculate activity state by comparing the inbox and outbox to determine where a flow reached, in the event of failure, over storing current state.

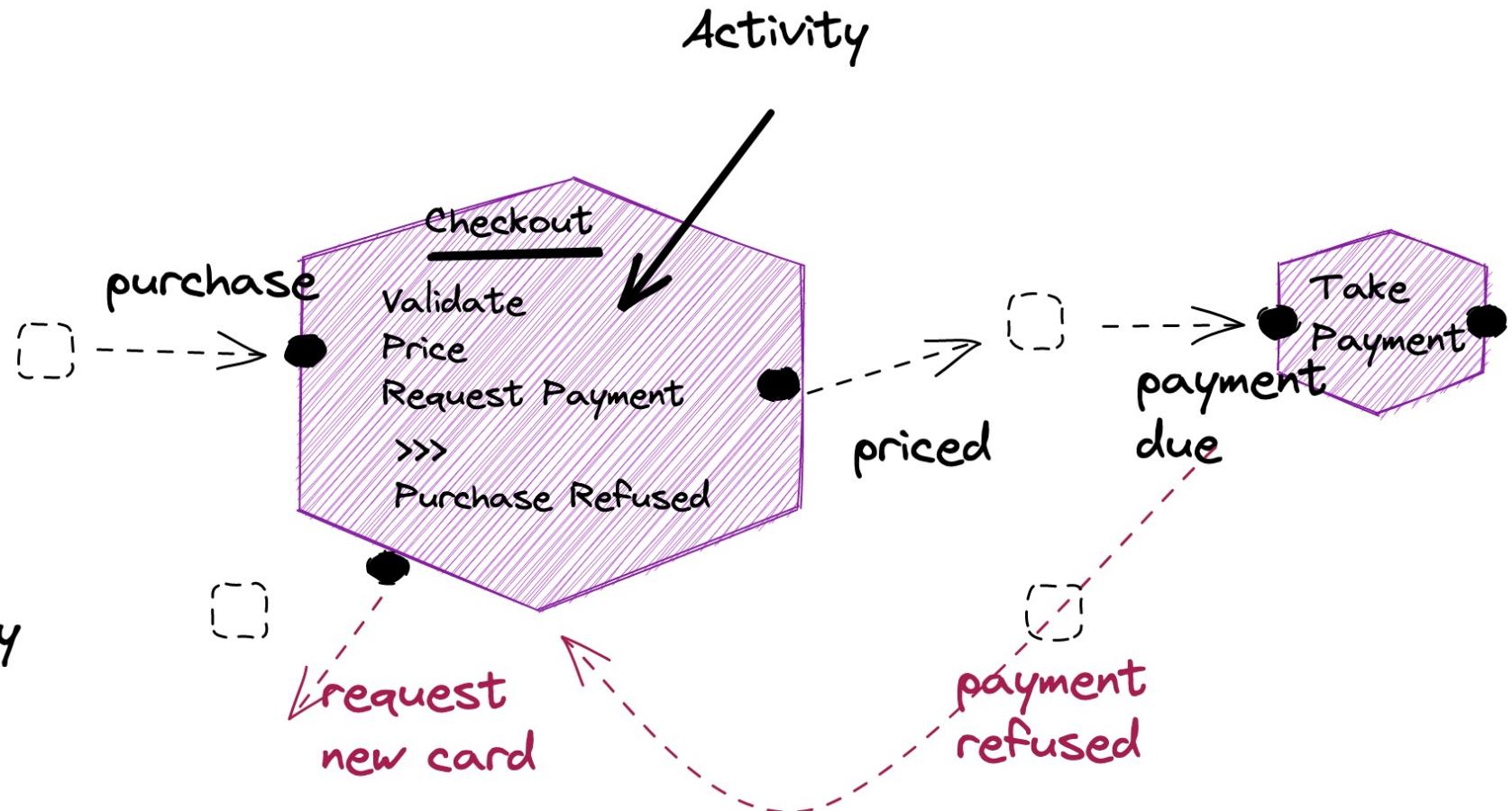


Pat Helland - Activity  
(State Machine)





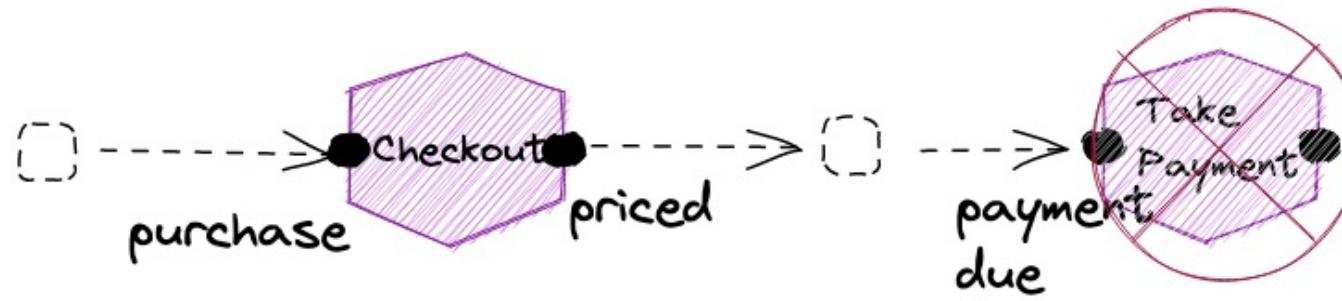
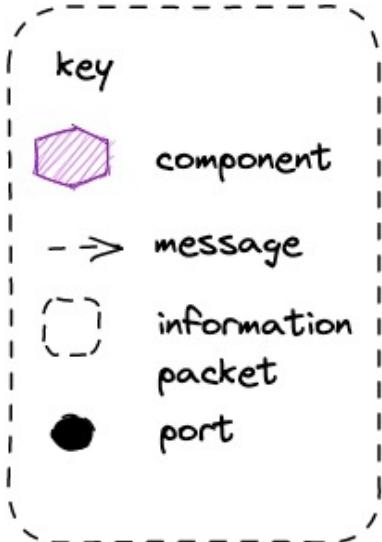
Pat Helland - Activity  
(State Machine)



# Agenda

T: 45

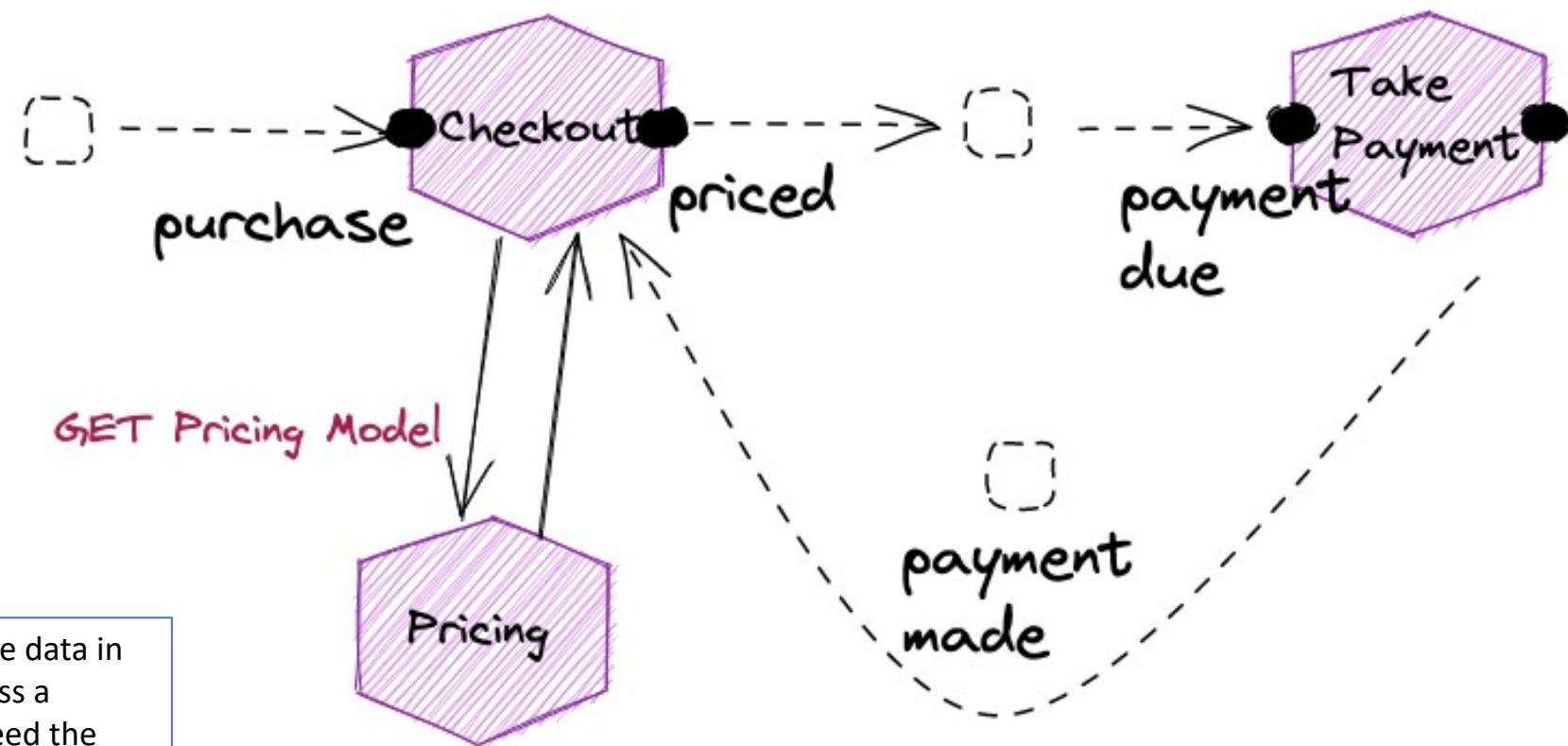
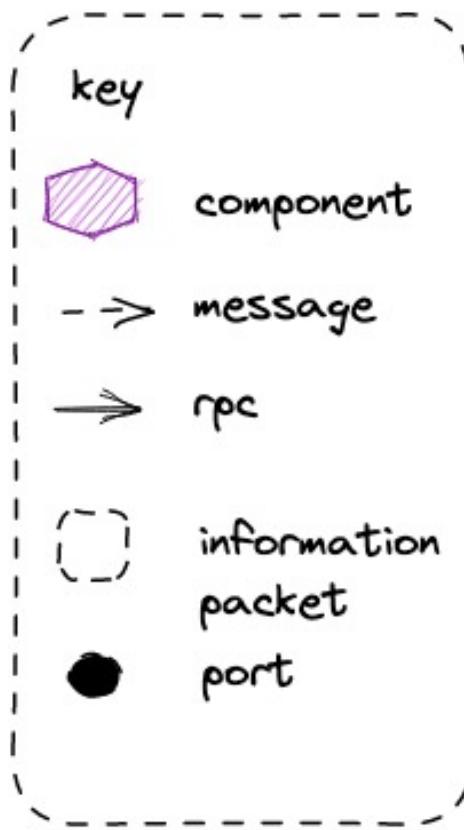
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - **Reactive Systems**
    - Message Passing
    - **Resilience**
    - **Elasticity**
    - **Responsiveness**



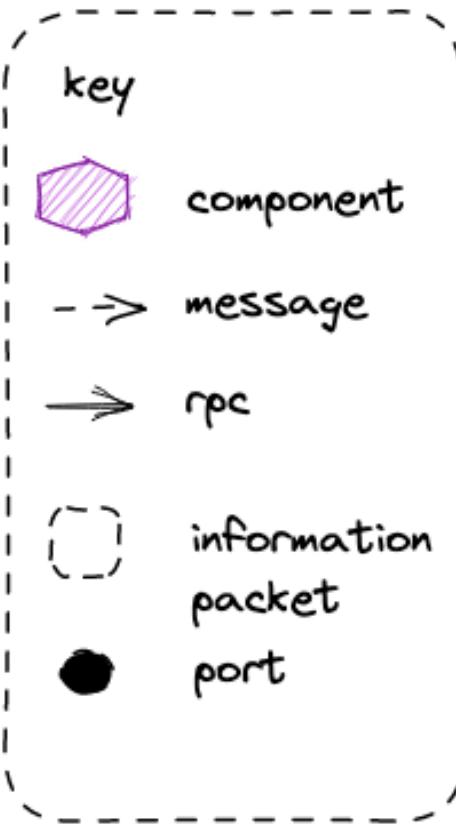
**Bulkheads:** In an asynchronous conversation a fault does not propagate back up the chain – we have a bulkhead that protects us against failure

## Work Queues on a Fault

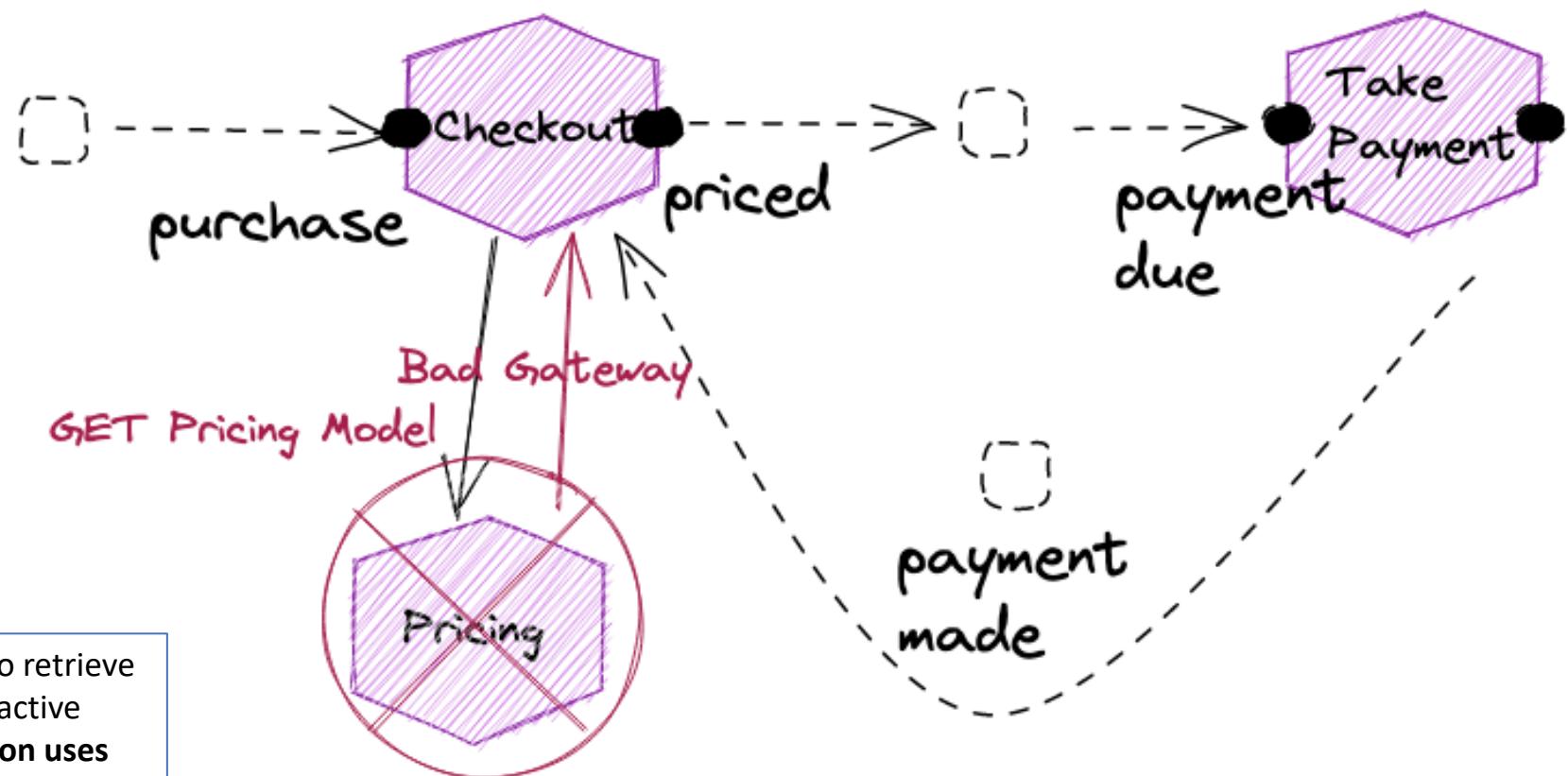
If Take Payment is not available, we can queue requests. When it starts up, Take Payment can process those requests and respond as normal.

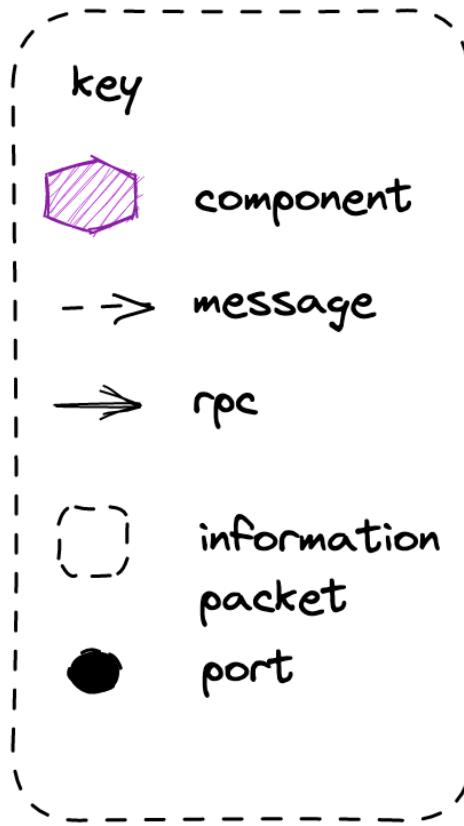


**Lookups:** We may not have all the data in our component needed to process a message we receive - here we need the prices for your purchase – which implies a lookup such as an HTTP GET to the owner

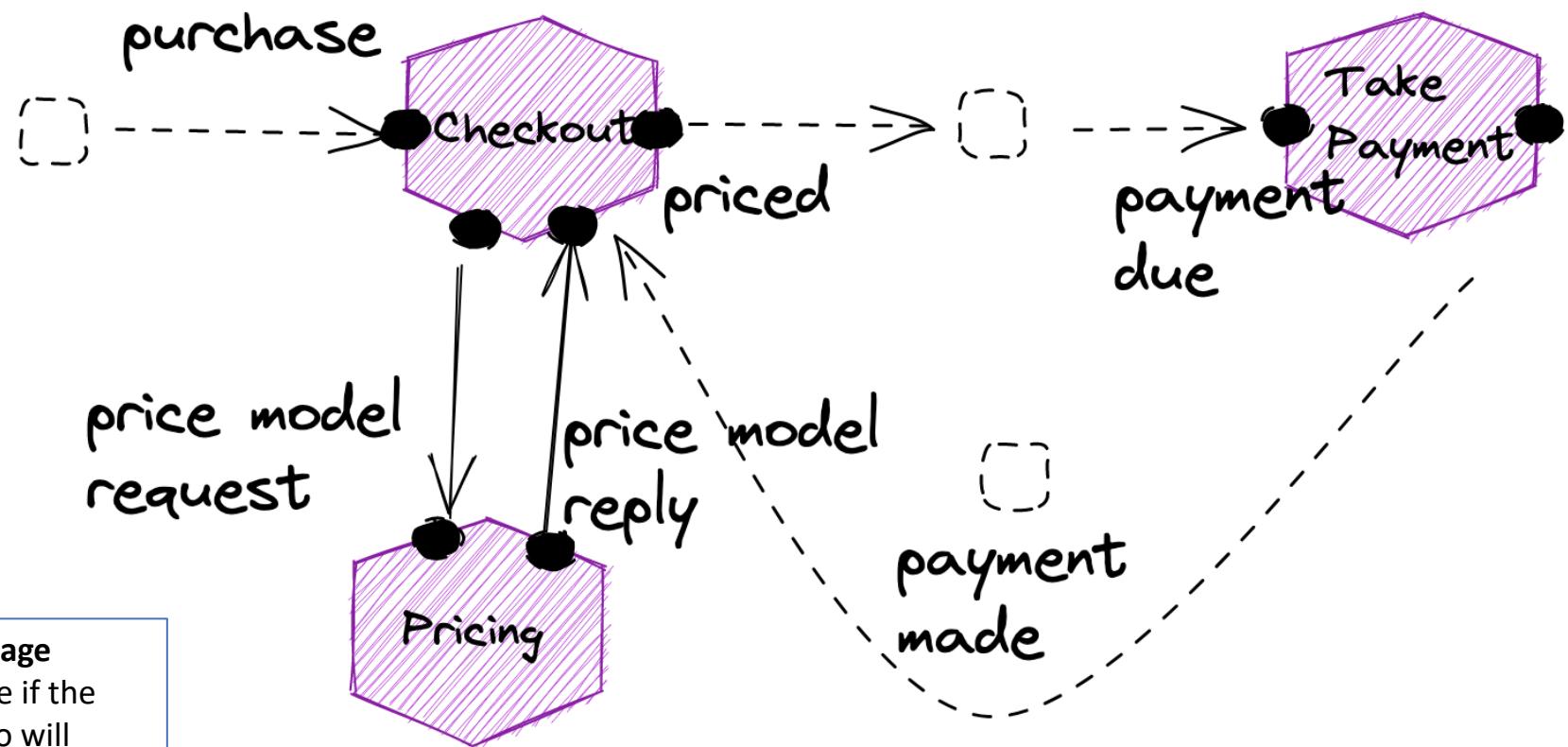


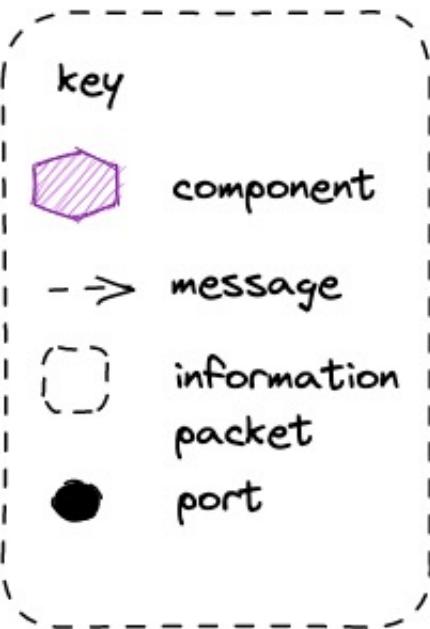
**Don't Do Gets:** Don't use a GET to retrieve data from another service in a reactive architecture. A reactive application uses message passing.



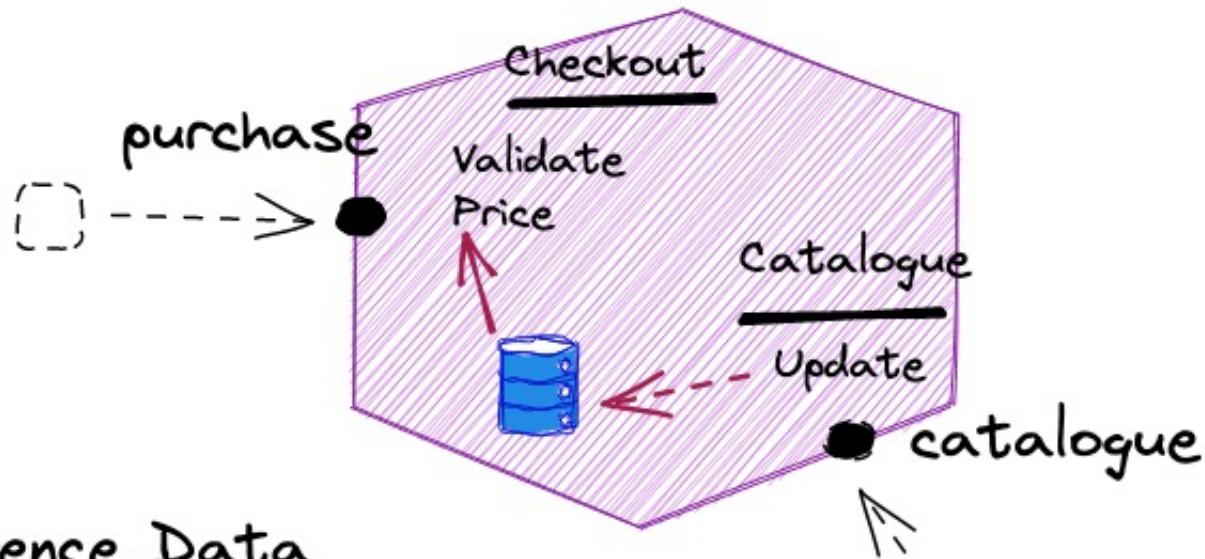


A reactive application uses **message passing**. Requests here will queue if the pricing module is not available, so will eventually be serviced.





A reactive application uses message passing. What if we want to reduce the latency of the lookup?

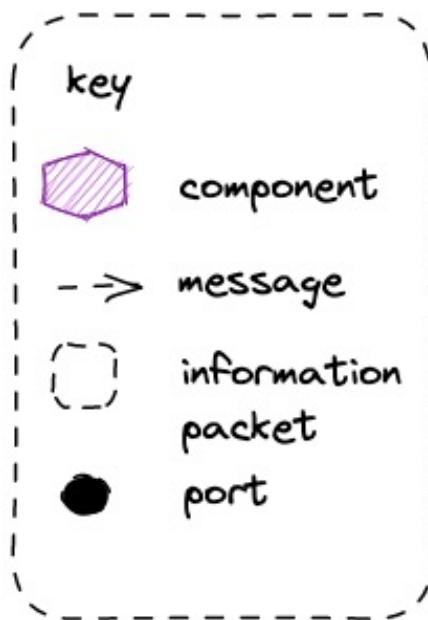


Pat Helland - Reference Data

(Event Carried State Transfer (ECST))

**ECST:** Subscribes to events from a component to build a forward cache (projection) of that component's published data, and uses it for lookups

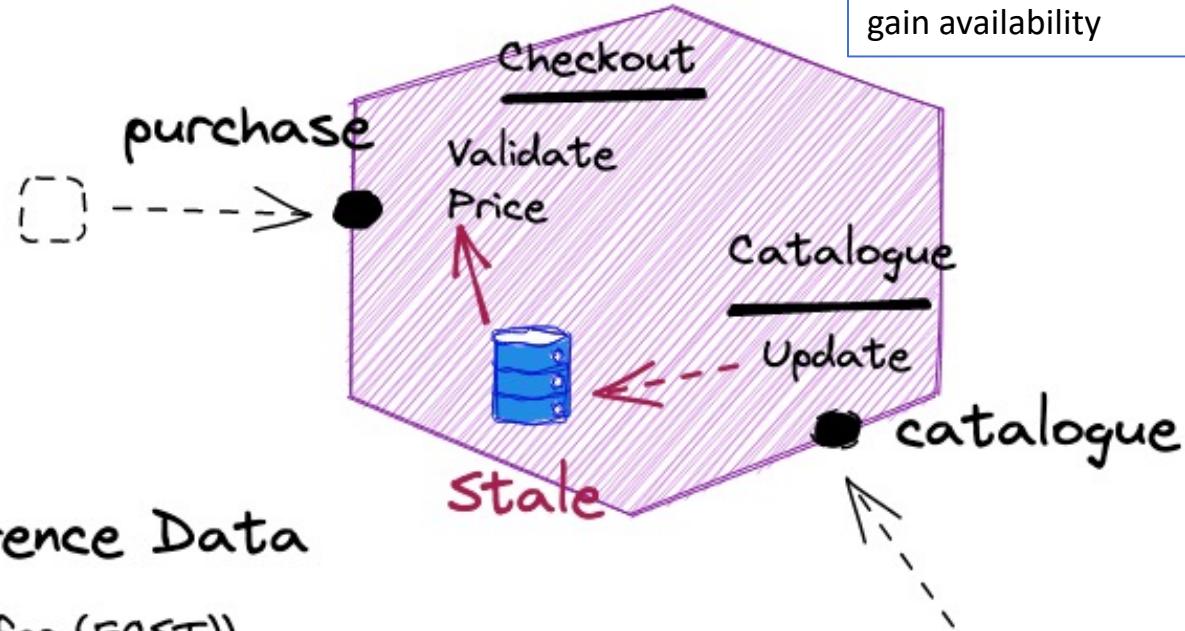




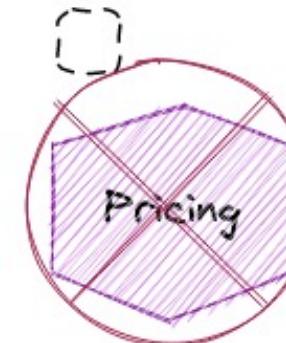
Pat Helland - Reference Data

(Event Carried State Transfer (ECST))

**Eventual Consistency:** We are eventual consistent, stale, but continue to be able to lookup. Accepting eventual consistency is a reactive principle.



**Tailor Consistency:** Reactive advises choosing the consistency you need for its trade-offs. Usually we can tolerate eventual consistency – nearly everything is to an extent its just degrees of latency – to gain availability



# Let it Crash Pattern

(Candea & Fox "Crash-Only Software")

Transient or rare failures are hard to detect

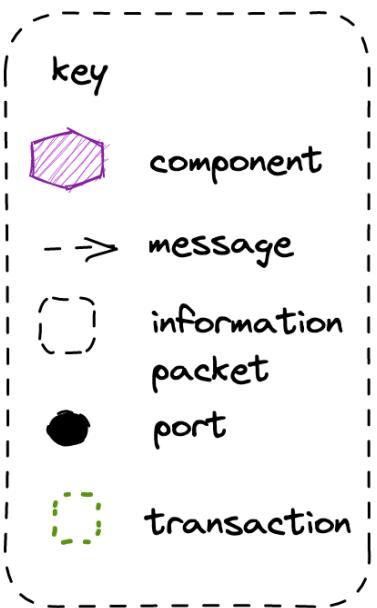
Recovery from a fault may be a complex problem

Crash and Restart =>

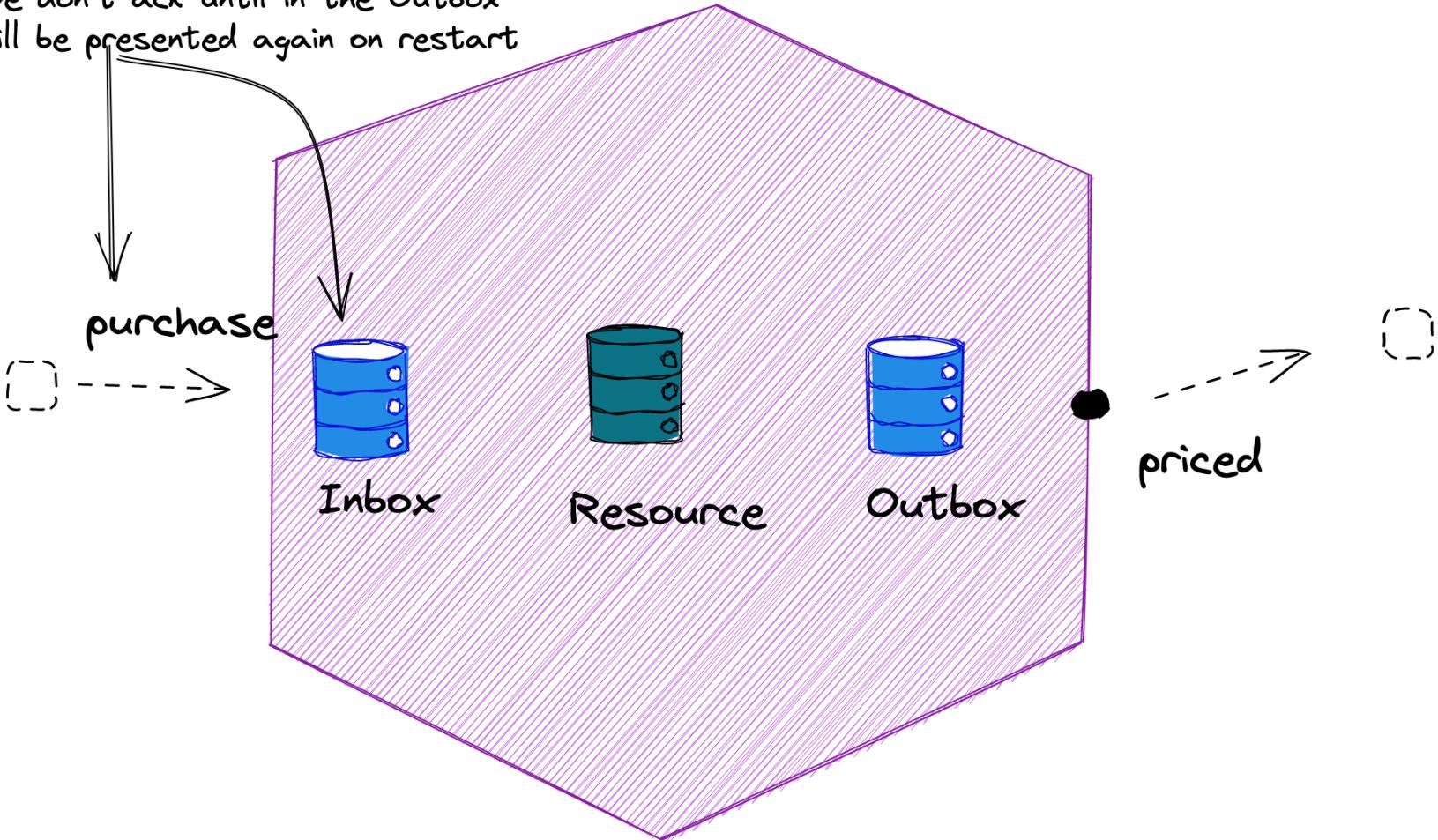
Start Up is Recovery

Shut Down is Clean

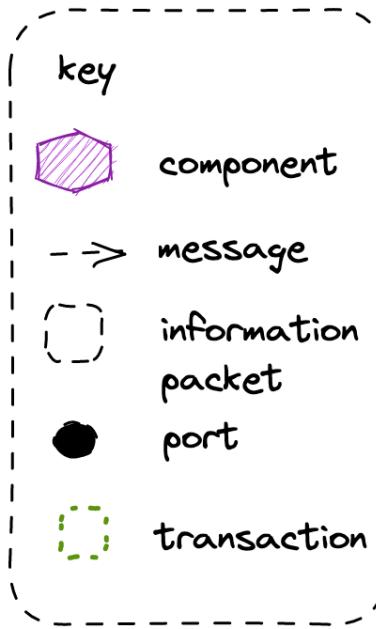
## Don't Ack until Done



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



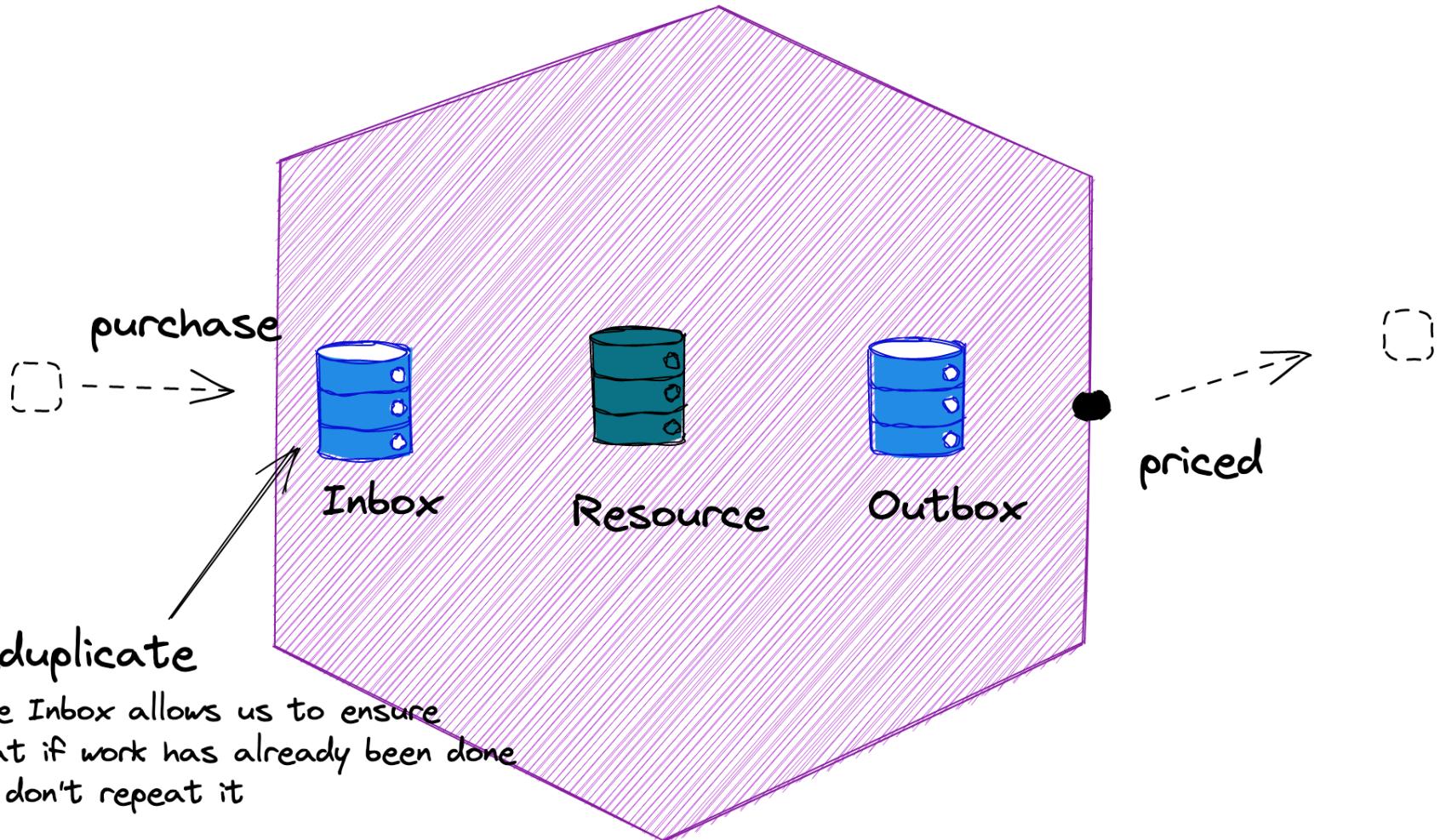
Let It Crash

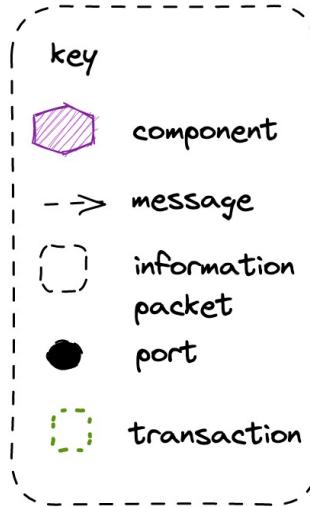


Let It Crash

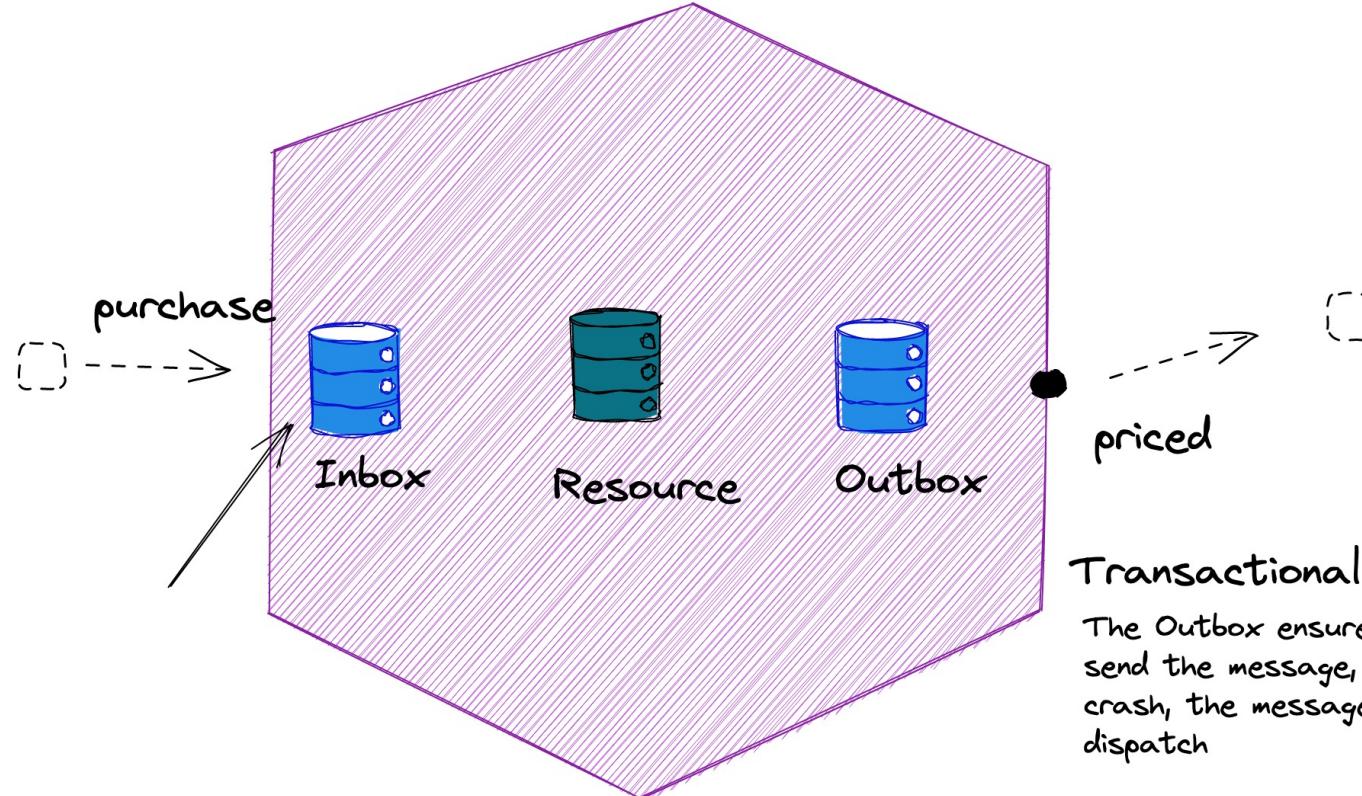
Deduplicate

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



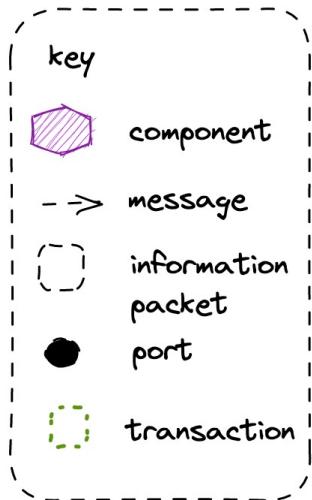


Let It Crash

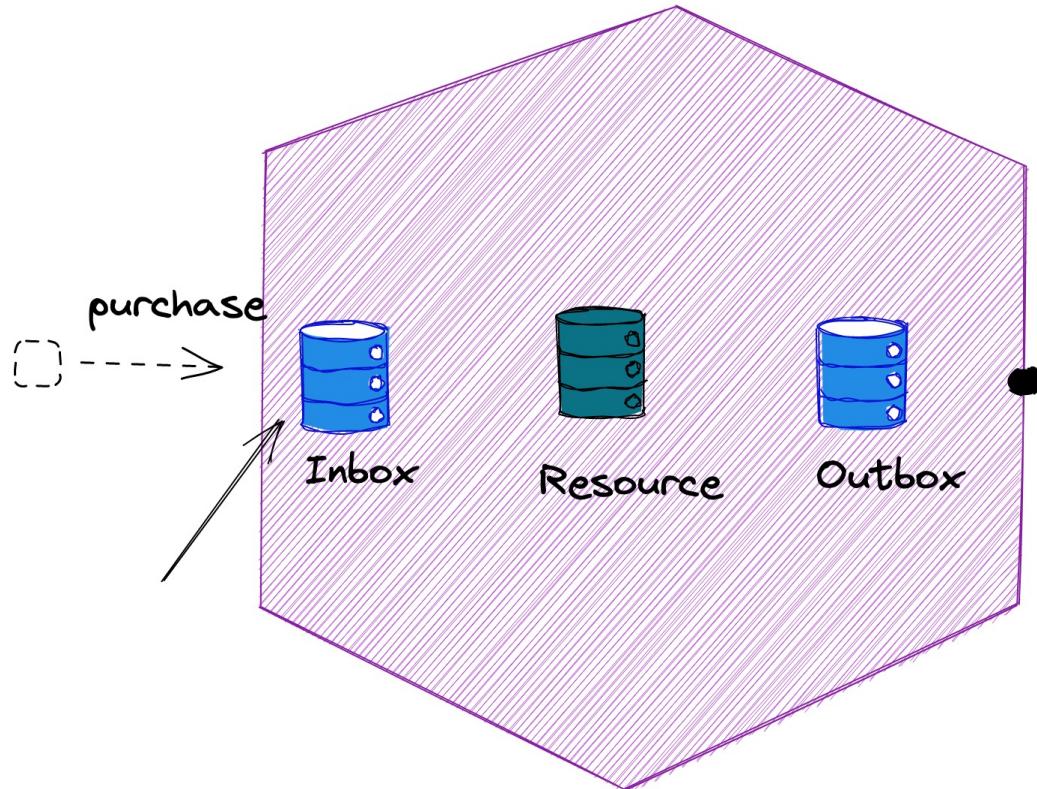


### Transactional Messaging

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



Let It Crash

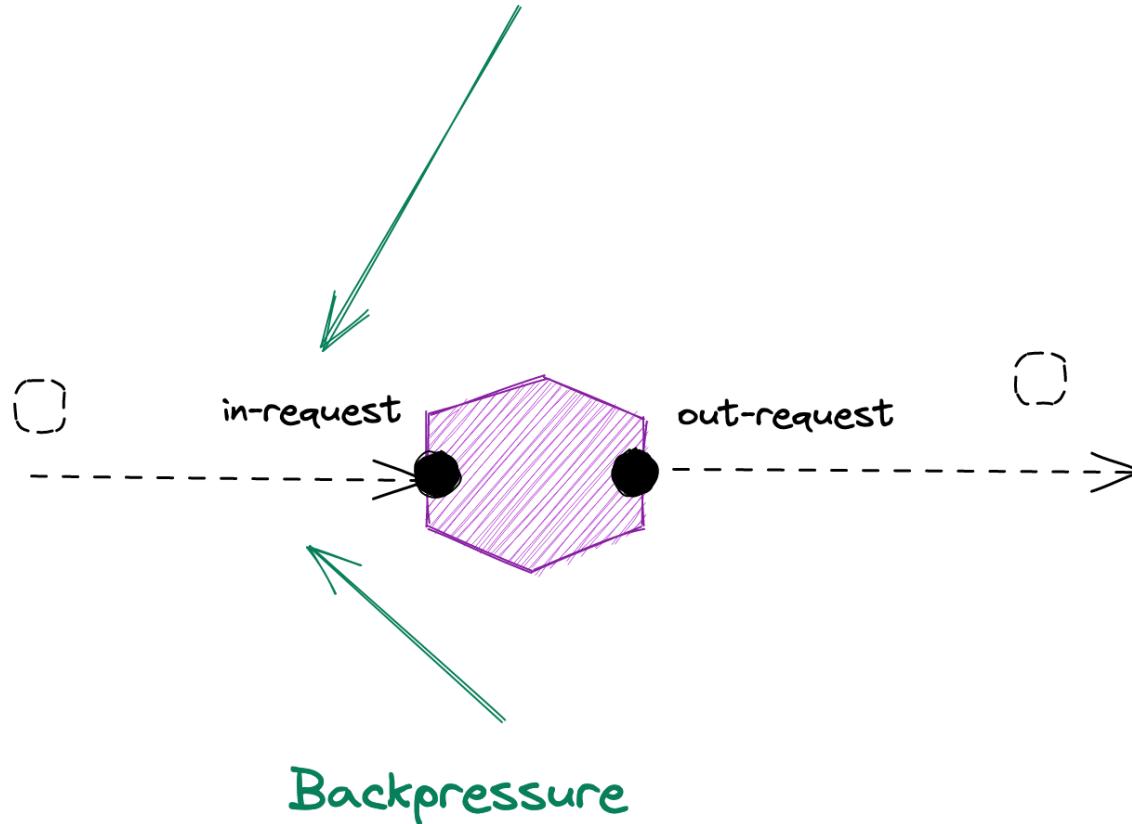
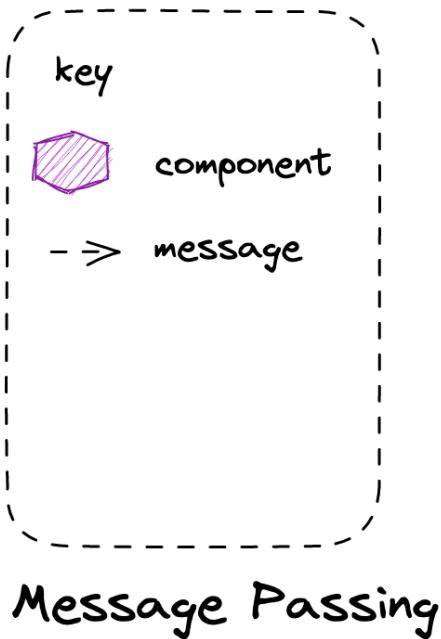


## Asynchronous Messaging

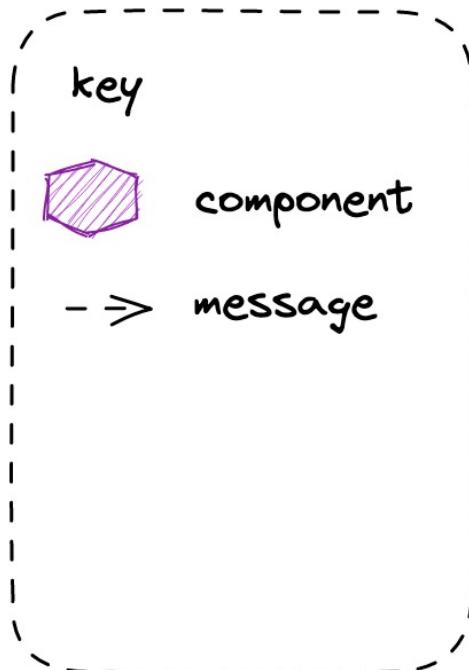
Because we communicate via messaging we are resilient to the use of "Let it Crash" by other components because we queue messages for their restart

**ECST:** Don't forget, because we use ECST and not a GET, we support "Let it Crash" by components where we need to lookup

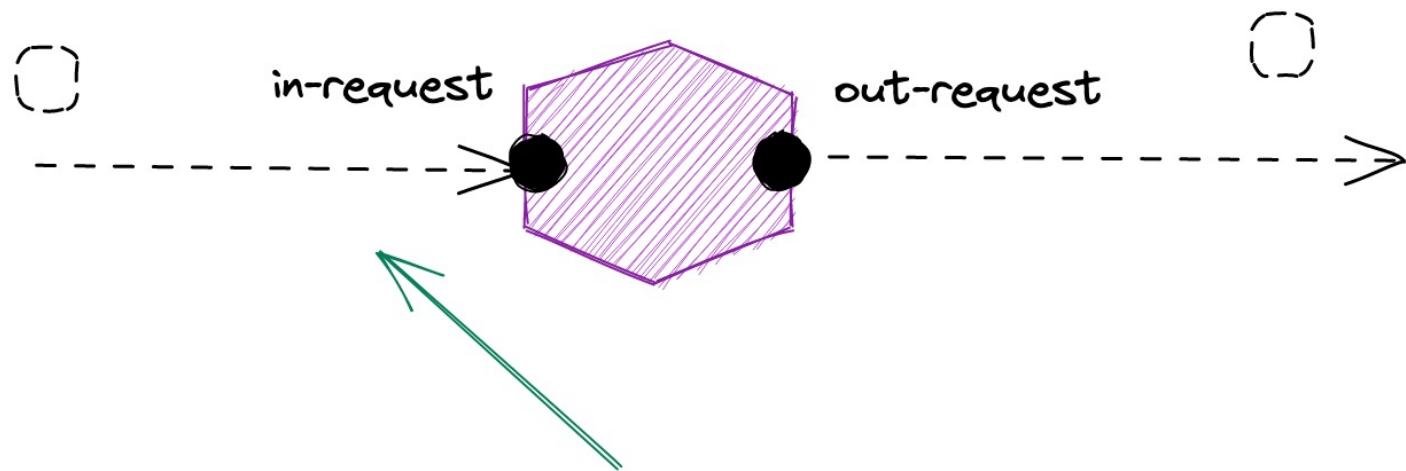
Push => open socket, middleware calls us as messages arrive  
Pull => We poll for messages



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

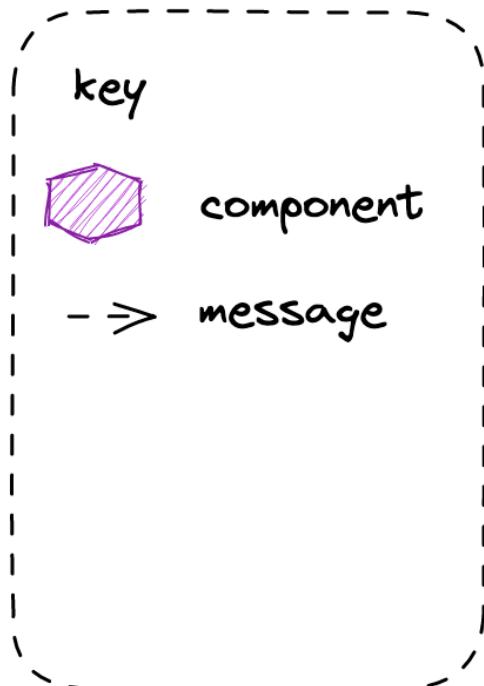


## Message Passing

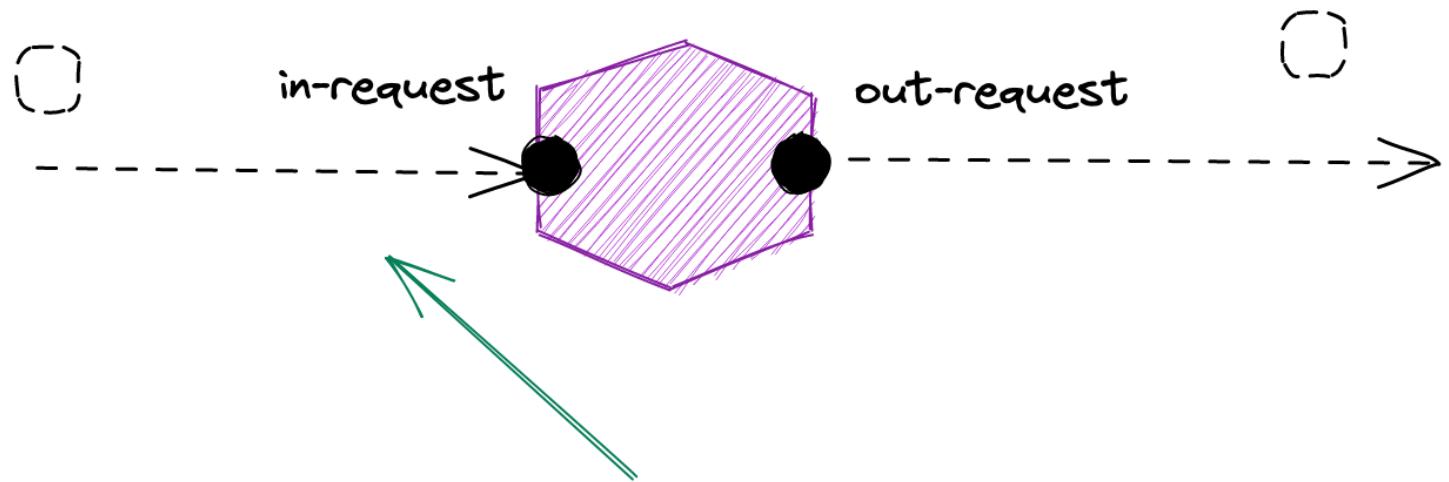


## Circuit Breaker

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



## Message Passing



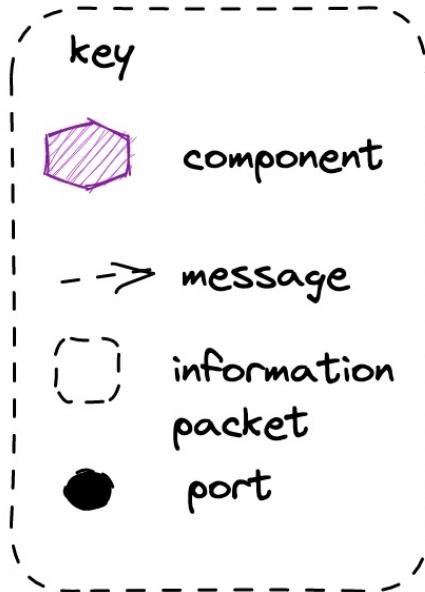
## Load Shedding

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

# Agenda

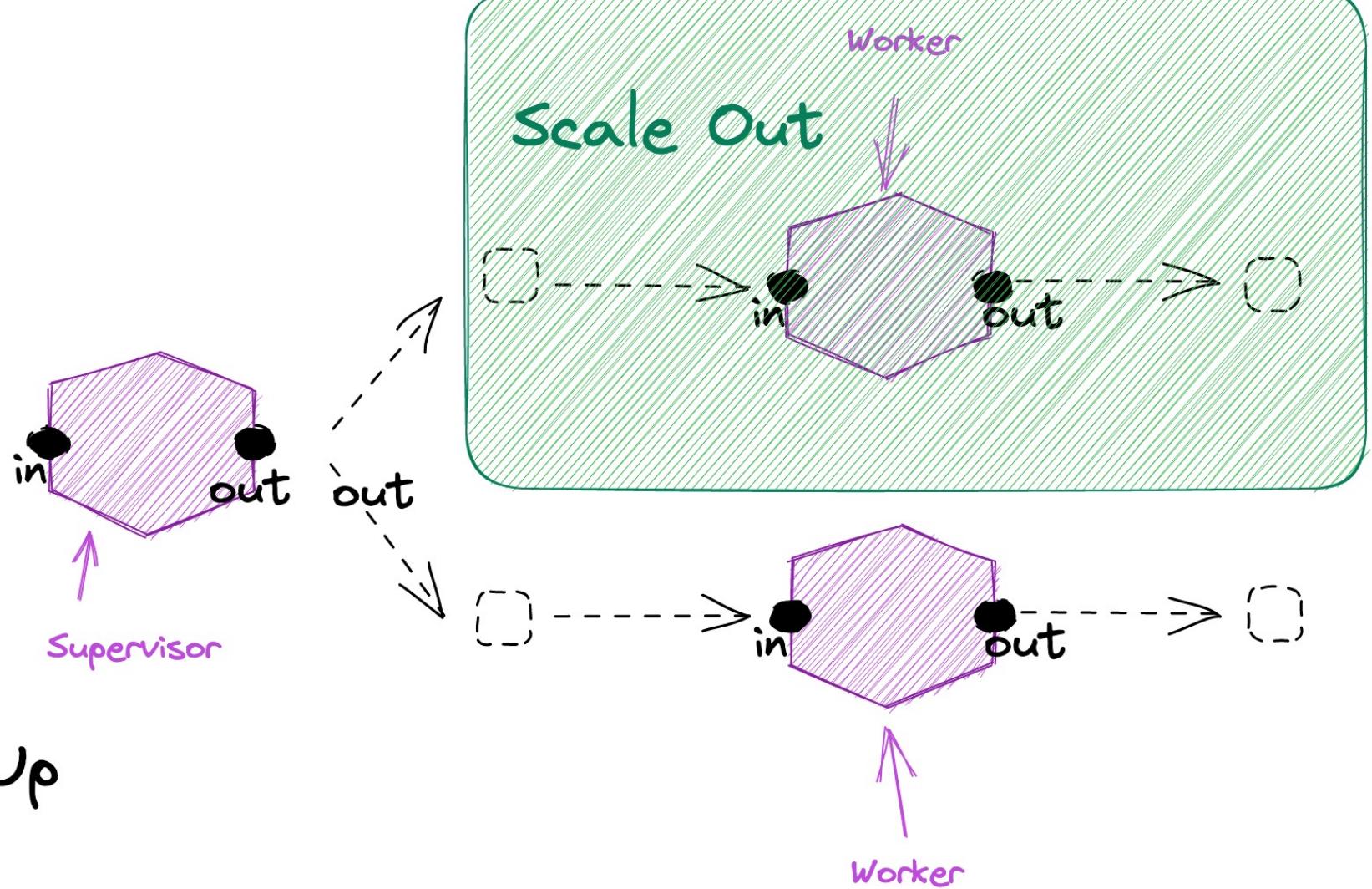
T: 50

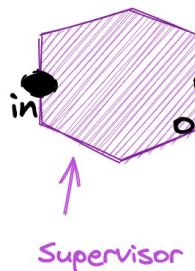
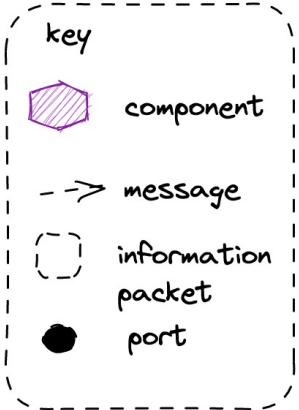
- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - **Reactive Systems**
    - Message Passing
    - Resilience
    - **Elasticity**
    - Responsiveness



Scale Out, not Up

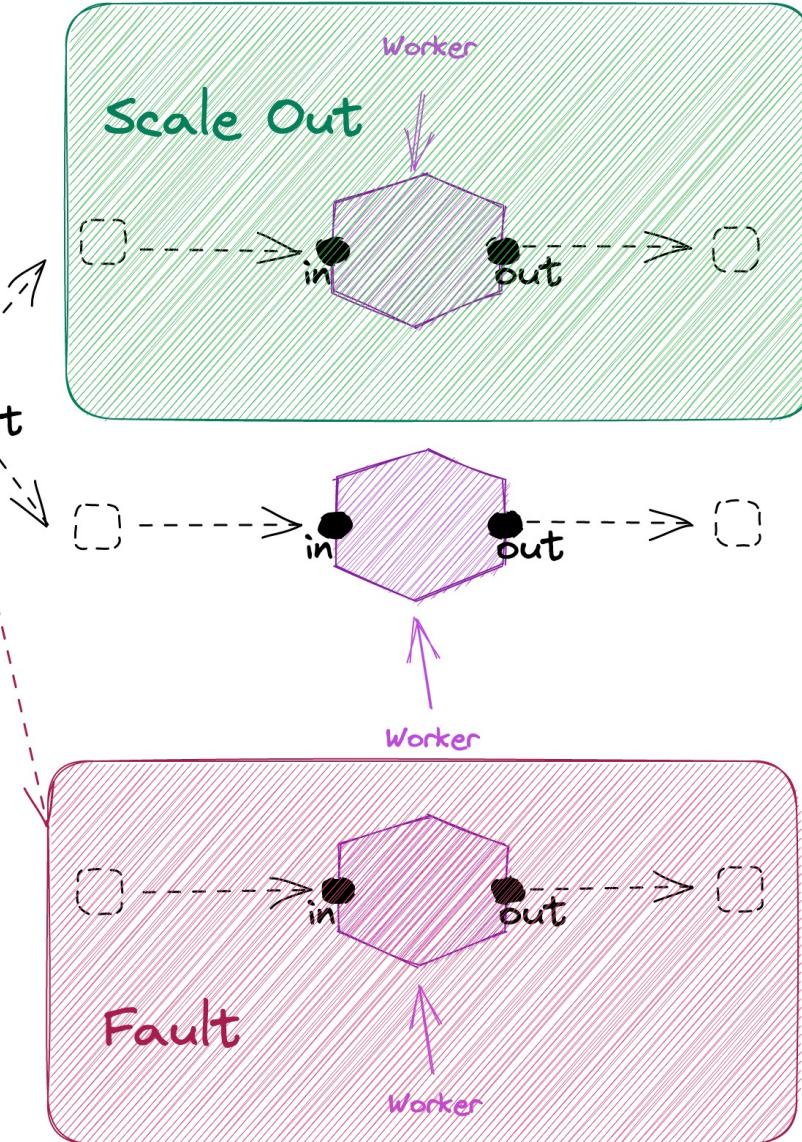
12-factor et al.



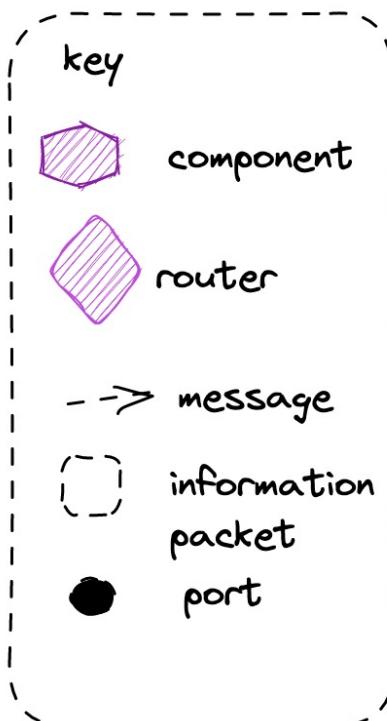


**Scale Out, not Up**

12-factor et al.



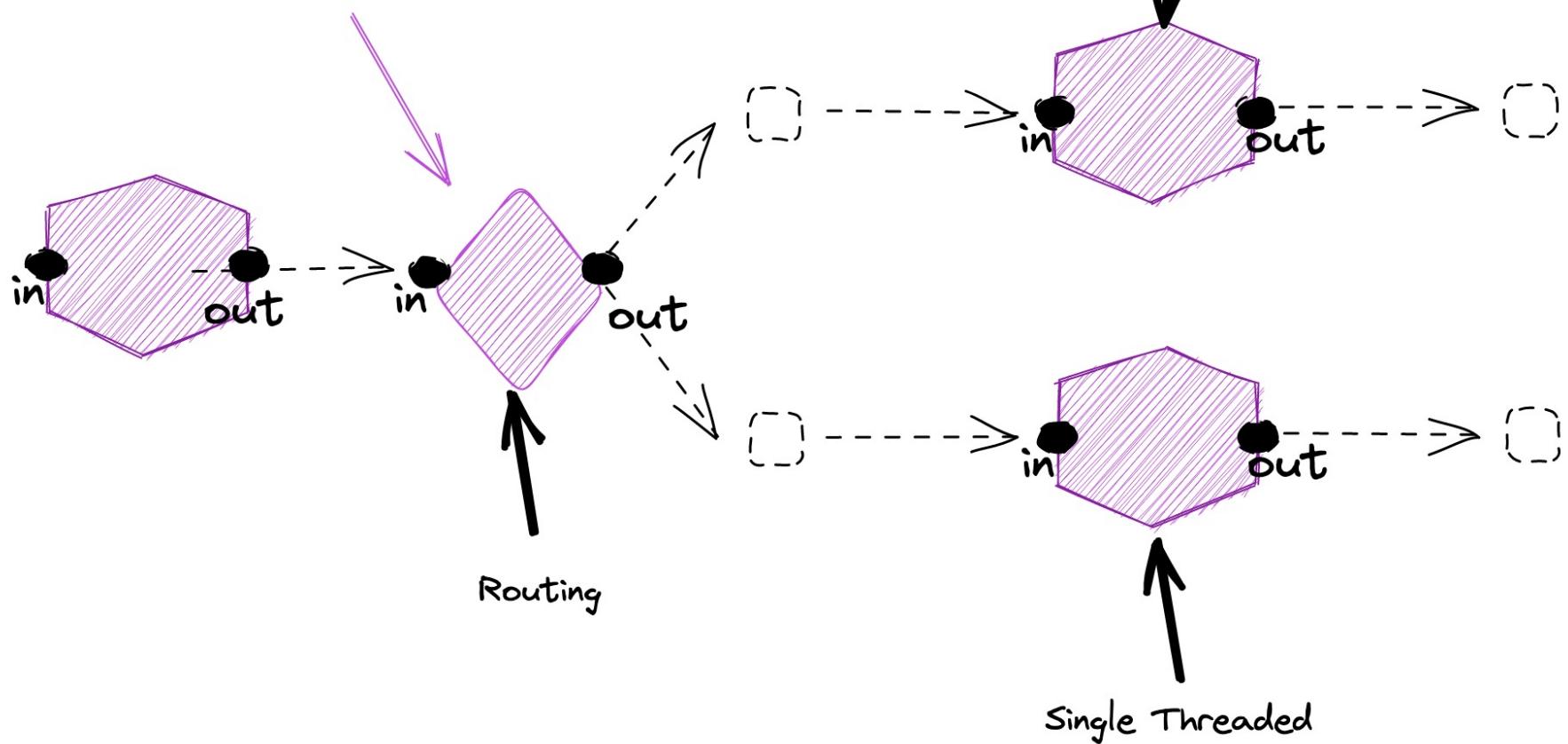
**Single Threaded** We can use the Reactor pattern for message processing and be single threaded, but if sequential updates are required we must be able to consistently hash messages to partitions when routing to scale



## Routing

Random - any consumer

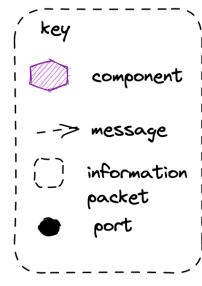
Consistent Hashing - same consumer



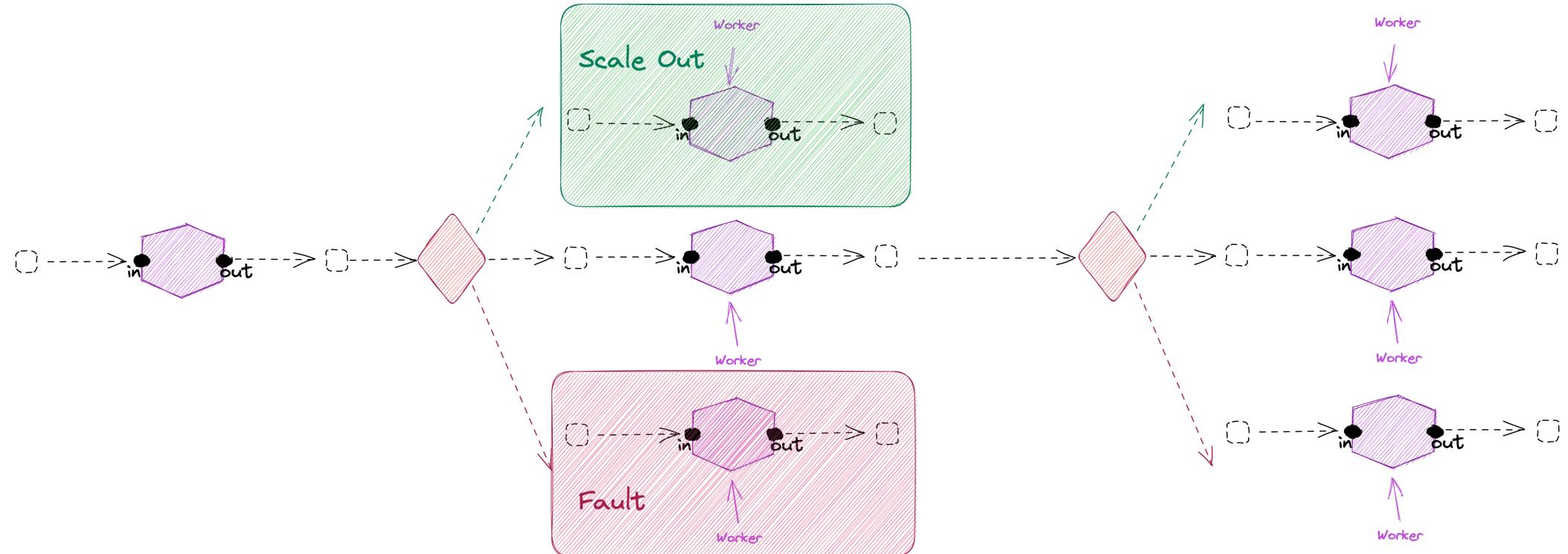
# Agenda

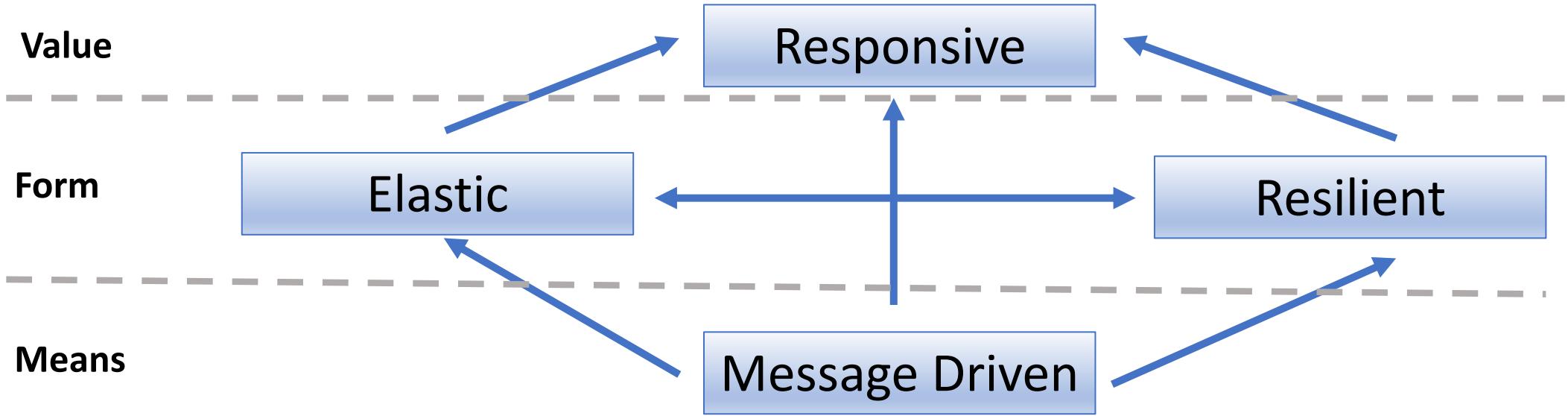
T: 55

- Objects
  - SOA
- Reactive
  - Reactive Programming
    - Dataflow
    - Actors
    - Flow Based Programming
  - **Reactive Systems**
    - Message Passing
    - Resilience
    - Elasticity
    - **Responsiveness**



**Responsive**  
12-factor et al.





And Done. Thank You.