Practical Messaging

A 101 guide to messaging

Ian Cooper

X/Hachyderm: ICooper

Who are you?

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

BRIGHTER

HOME INTRO TUTORIALS CONCEPTS DOCS

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

Day One Messaging

- Distribution
- Integration Styles
- Reactive Architectures
- Messaging Patterns

Day Two Conversations

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

When do you get to write code?

When we get to messaging patterns, the afternoon of Day One, and then interspersed over Day Two

Prerequisites

We will use Rabbit MQ and Kafka for examples. You should have Docker (or an equivalent) installed on your machine, as exercises provide a Docker Compose file to spin up RMQ and Kafka.

You will need to be able to write code with an editor/IDE of your choice.

You can choose from: C#; Java; Python; Go; JavaScript

Exercise Code

https://github.com/iancooper/Practical-Messaging-Sharp
https://github.com/iancooper/Practical-Messaging-Python
https://github.com/iancooper/Practical-Messaging-JavaScript
https://github.com/iancooper/Practical-Messaging-Go
https://github.com/iancooper/Practical-Messaging-Java

Day One

What is driving messaging

DISTRIBUTED SYSTEMS

Why Distribute?

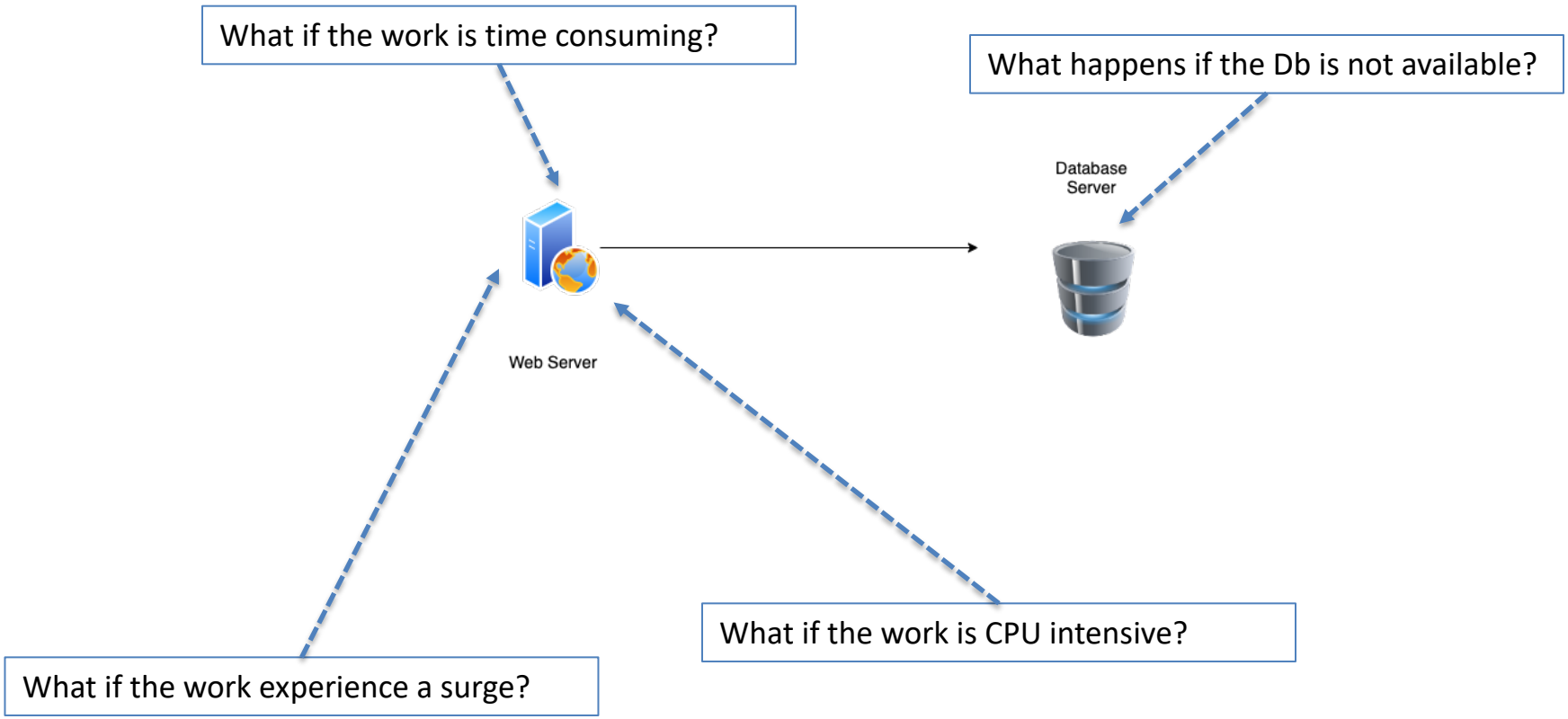
Performance and Scalability

Availability

Maintainability

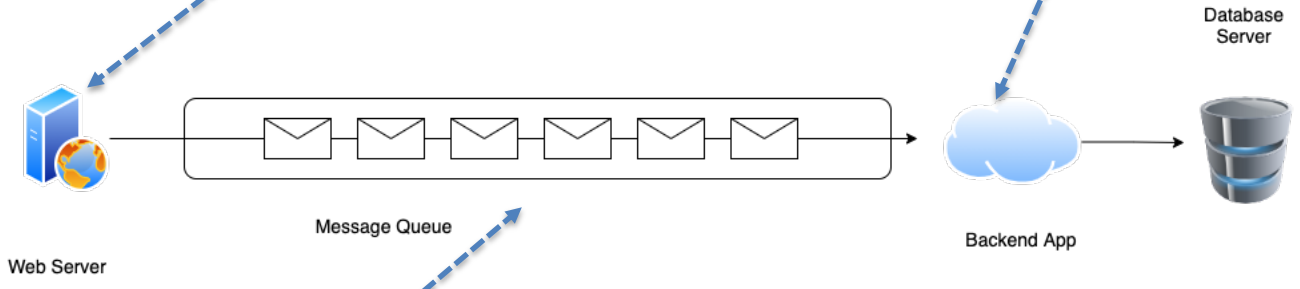
Inherent Distribution

Example: Task Queues


What if the work is time consuming?
What happens if the Db is not available?
Database Server
Web Server
What if the work is CPU intensive?
What if the work experience a surge?

A backend application can perform long-running or CPU intensive work, allowing the web server to service new requests.

The web server puts the work on a queue

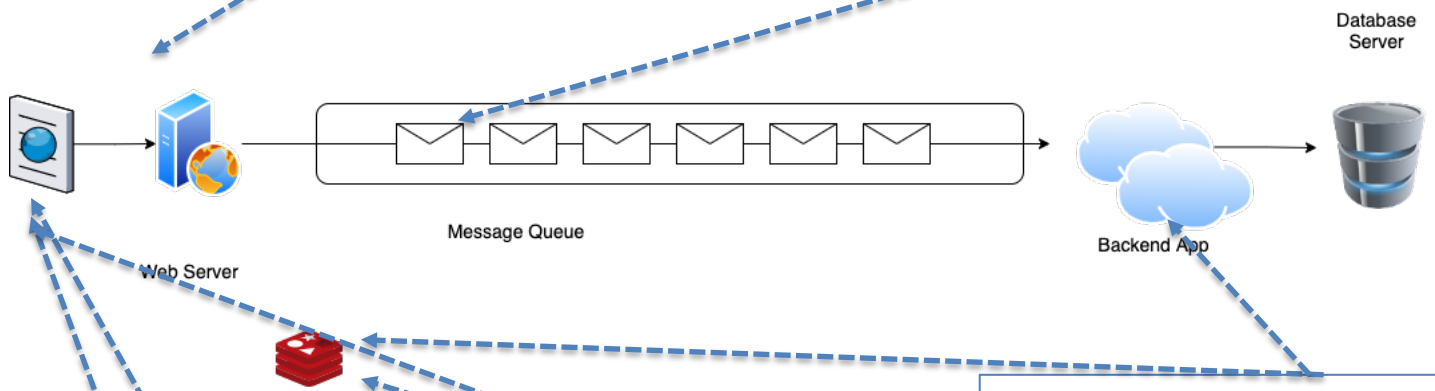


The queue stores work until we are ready to consume it. We can *throttle* to prevent surges.

We can scale out the backend services using a competing consumers approach, to ensure the queue does not backup.

We return 202 Accepted – we have your work request, and won't lost it.

We enqueue a work item for the request.



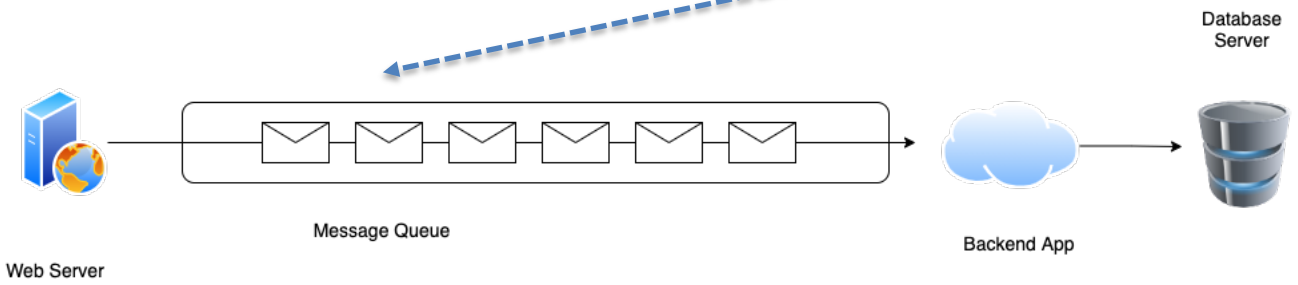
Backend app does work at sustainable pace, and updates KV store if required.

We return a Location header, so you can monitor progress.

Link to a progress page, backed by a KV store where we note progress

Link to Resource – 404 until created

This general technique is known as Decoupled Invocation – we separate building the command from executing it.



Decoupled Invocation Pattern

Use Decoupled Invocation. A producer puts a message onto a queue at the service endpoint. A consumer reads messages from the queue.

The queue stores messages for eventual processing.

If the rate of arrival at the endpoint is unpredictable, the queue acts as a buffer that makes it possible to predict the rate of consumption.

This makes it simpler to do capacity planning because peaks of requests are smoothed out by the queue.

The consumer must be able to control the rate of processing, otherwise a spike is simply passed down the wire.

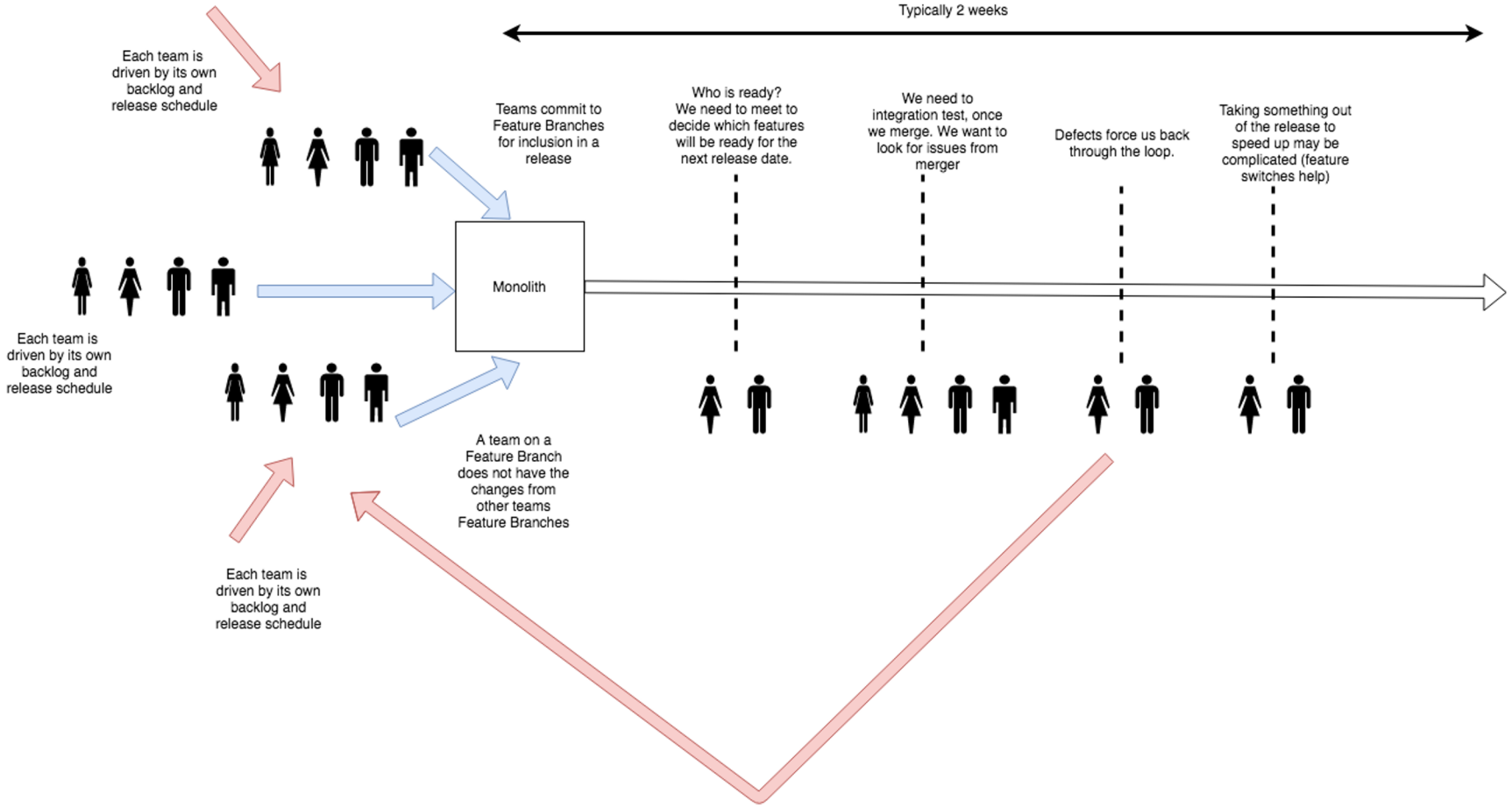
Example: Microservices

It’s all about velocity!!!

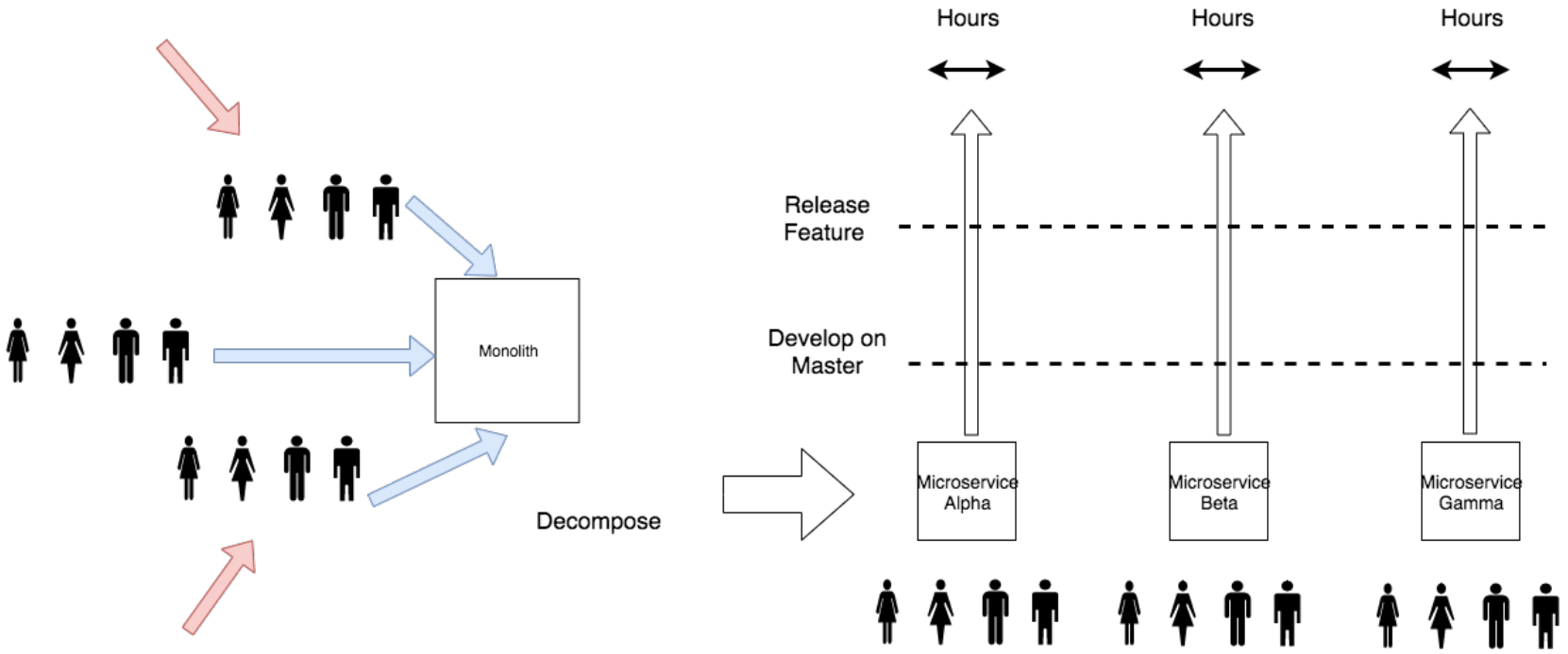
“Speed wins in the marketplace”

Adrian Cockcroft, former lead architect at Netflix

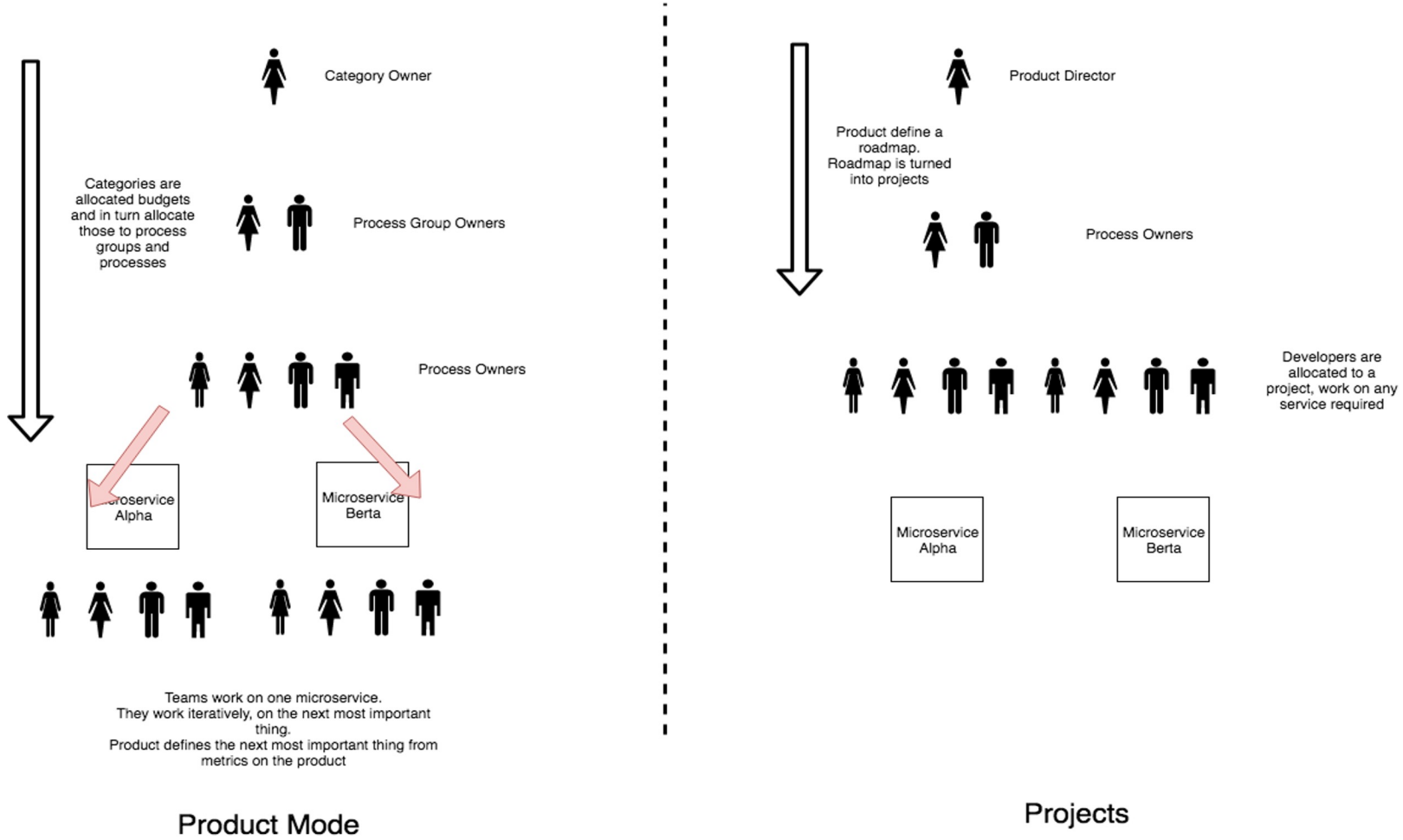
Monoliths Do Not Scale To Many Teams!



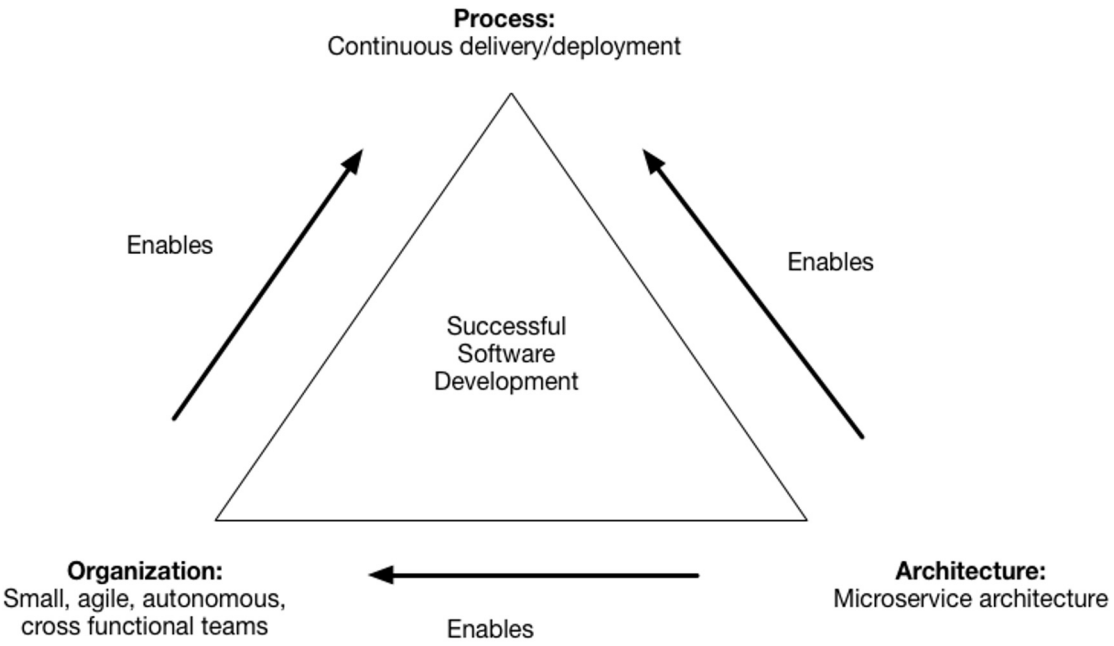
Microservices let us scale an organisation



Product Mode



Microservices enable agility



https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

The Price of Distribution

Fallacies of Distributed Computing

The network is reliable.

Latency is zero.

Bandwidth is infinite.

The network is secure.

Topology doesn't change.

There is one administrator.

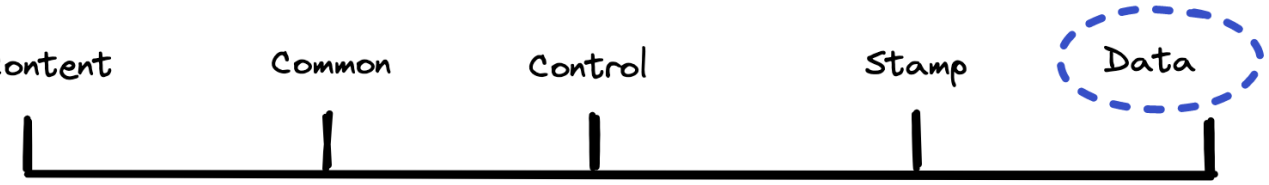
Transport cost is zero.

The network is homogeneous.

How do we communicate between microservices?

INTEGRATION STYLES

Coupling



Tight

More Interdependency

More Coordination

More Information Flow

Loose

Less Interdependency

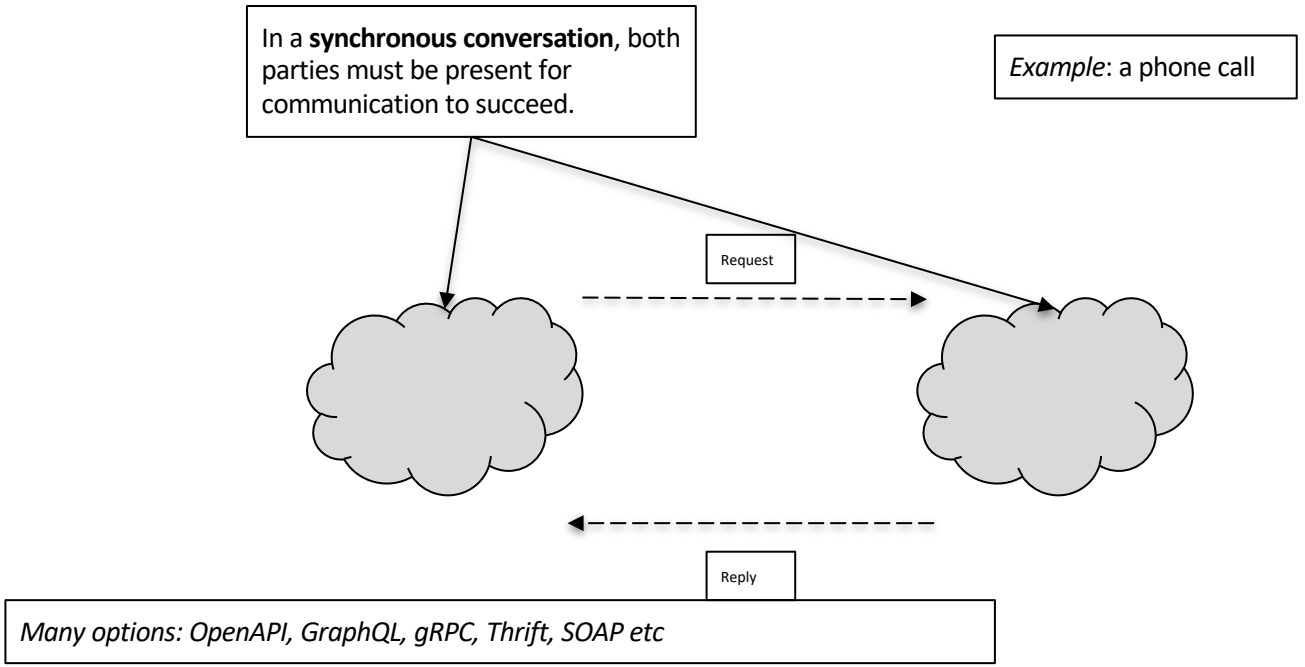
Less Coordination

Less Information Flow

Behavioral Coupling: a form of control coupling where we exchange a sequence of calls to complete work

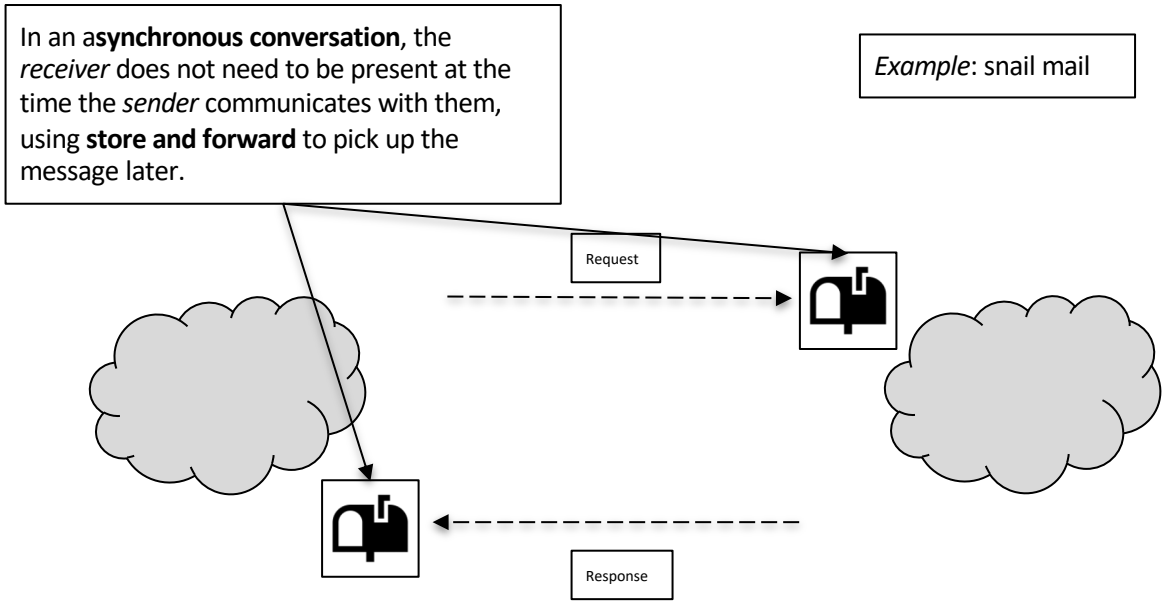
Temporal Coupling

Synchronous Conversation

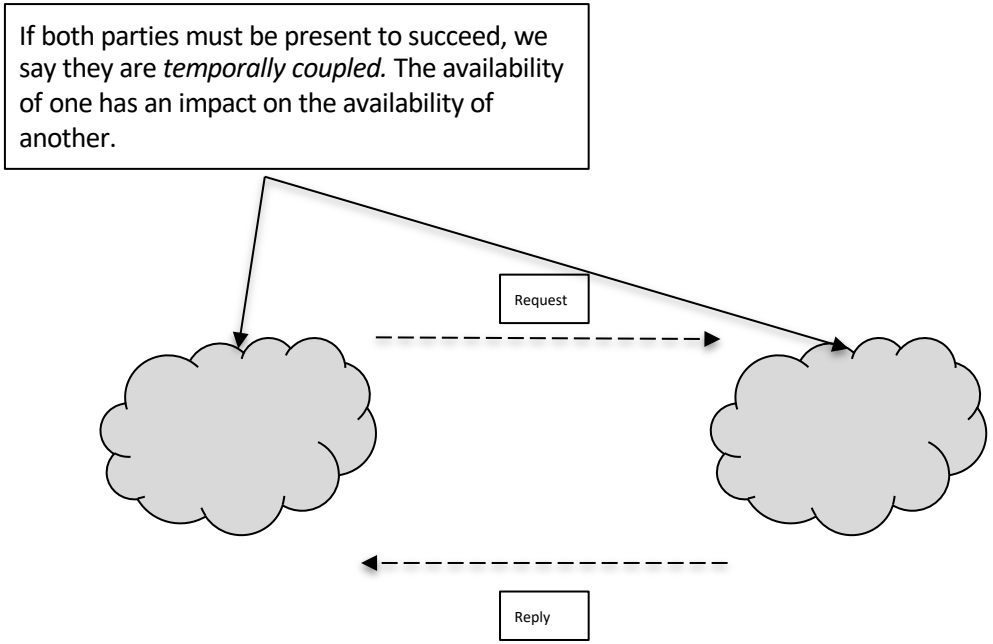


Temporal Coupling

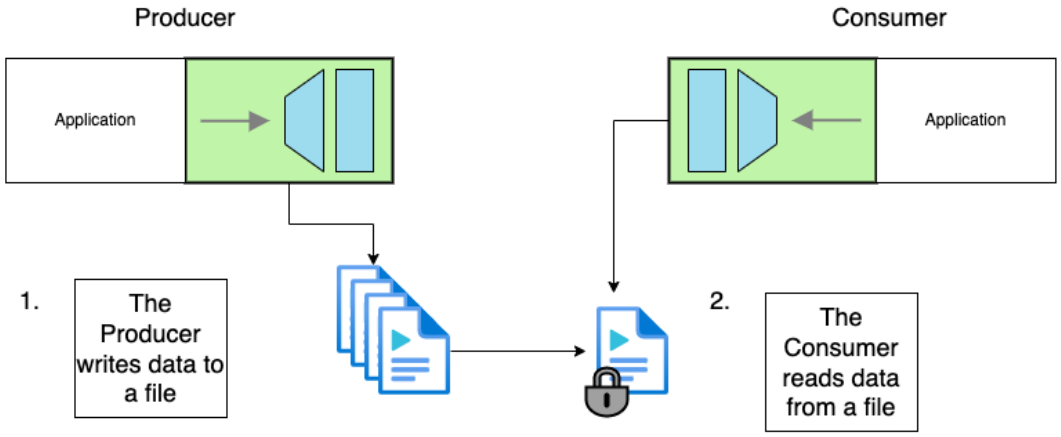
Asynchronous Conversation



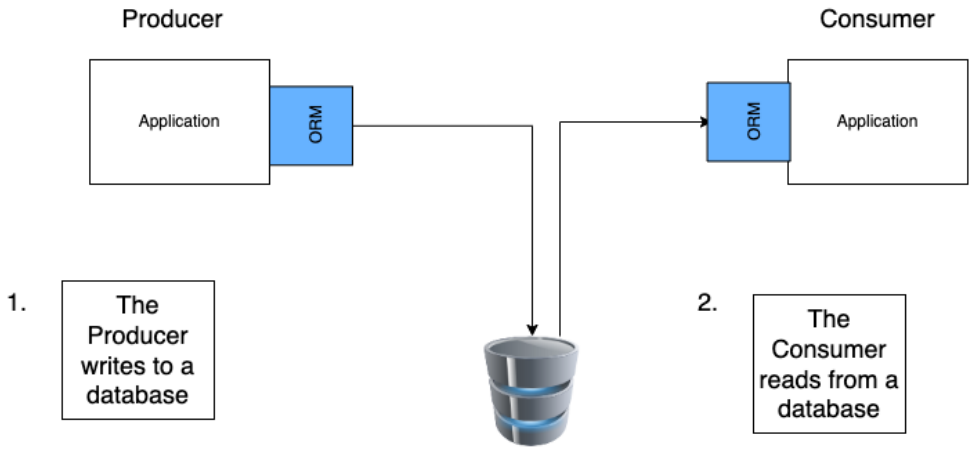
Temporal Coupling



File Transfer



Shared Database

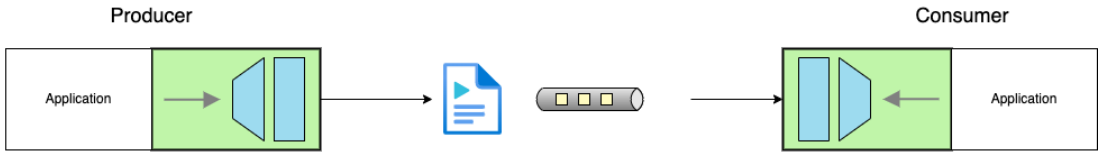


Remote Procedure Call

1. The Client calls a remote procedure on the Server

2. The Server listens for calls, actions them, and returns results

Messaging

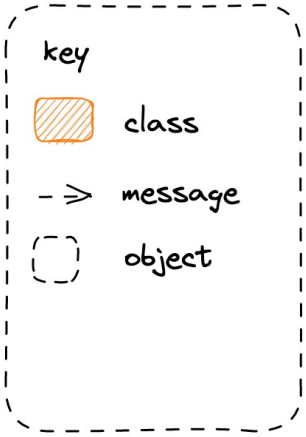


1. The Producer writes a message to a channel

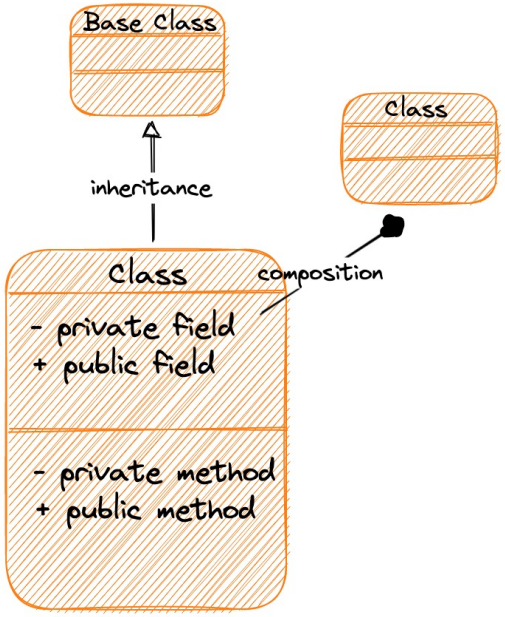
2. The Consumer reads a message from a channel

Integrating using events

REACTIVE ARCHITECTURES



Object Orientation

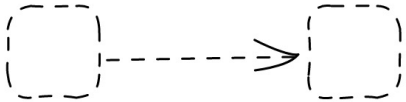


Data and Behaviour

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

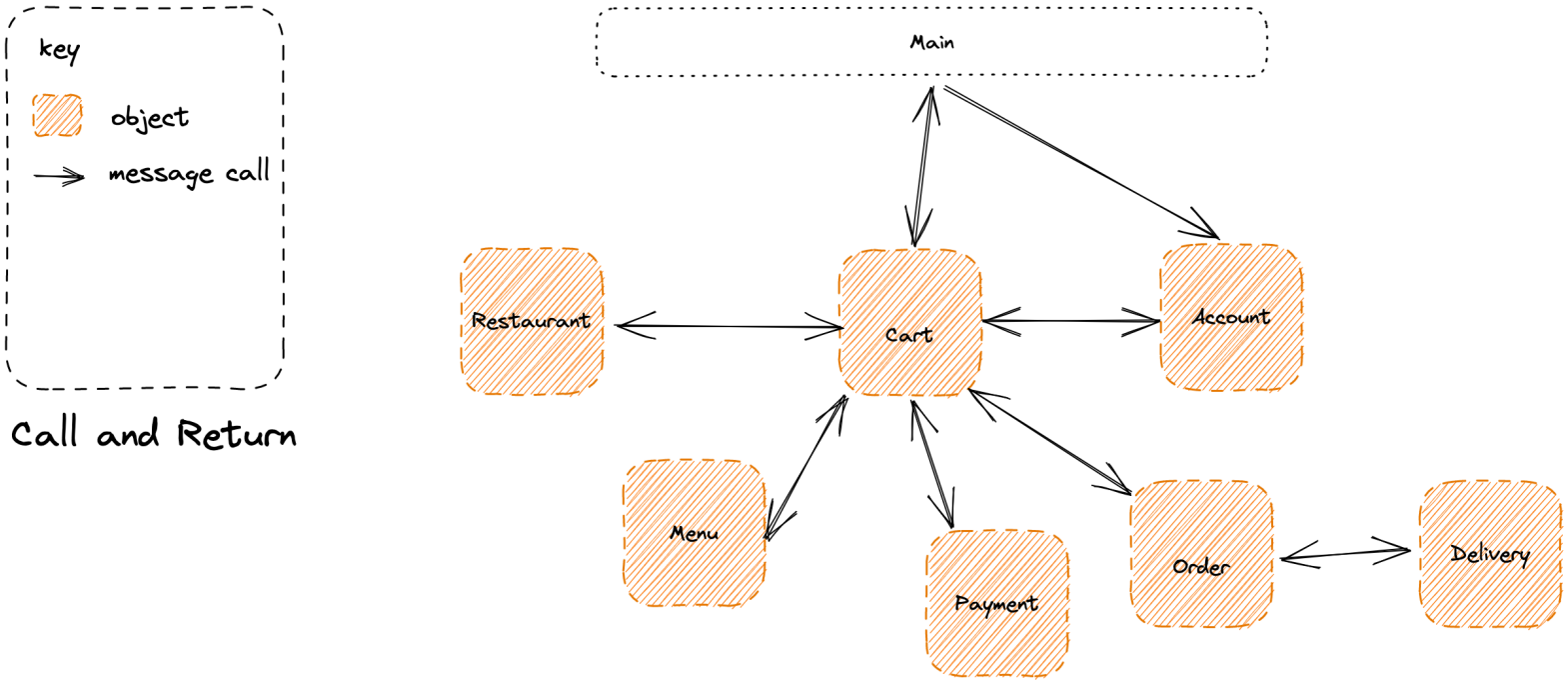
Due to inheritance the method chosen in response to a message may come from the base class of an object



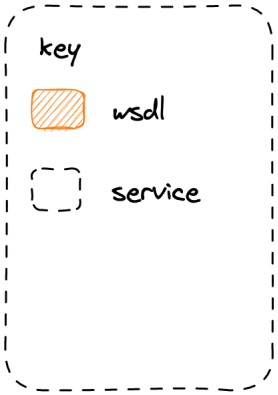
Message Passing

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

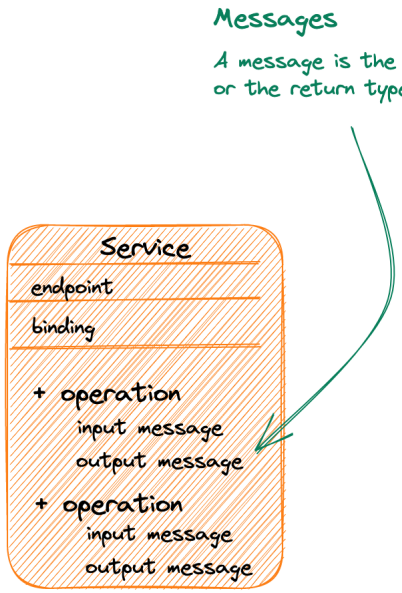
Call and Return: The main method represents our entry point and it uses message passing to invoke objects, which in turn invoke other objects, and return to their caller



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

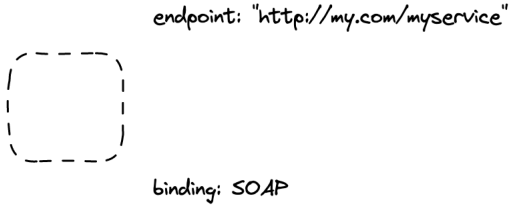


Service Orientation



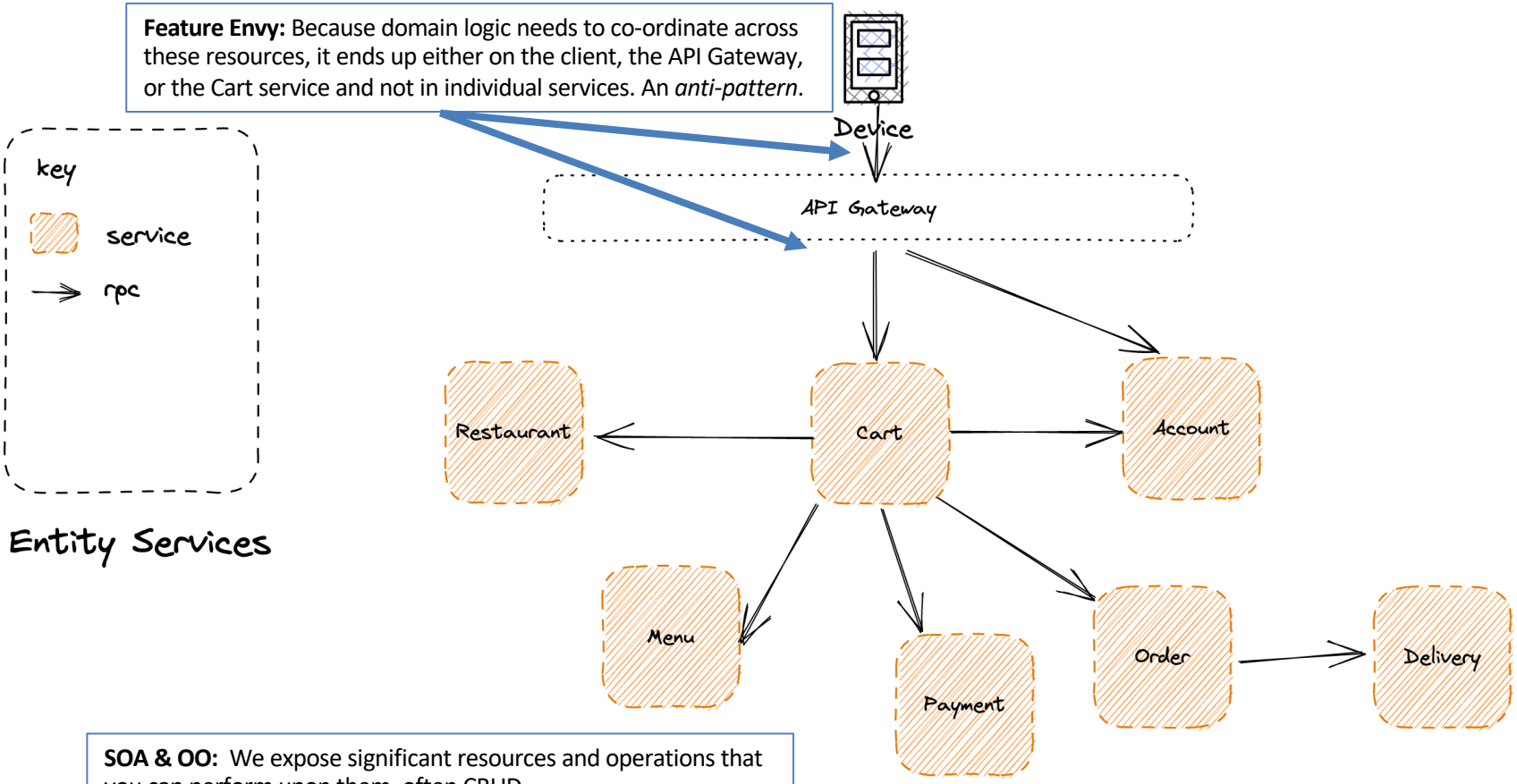
Data and Behaviour

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



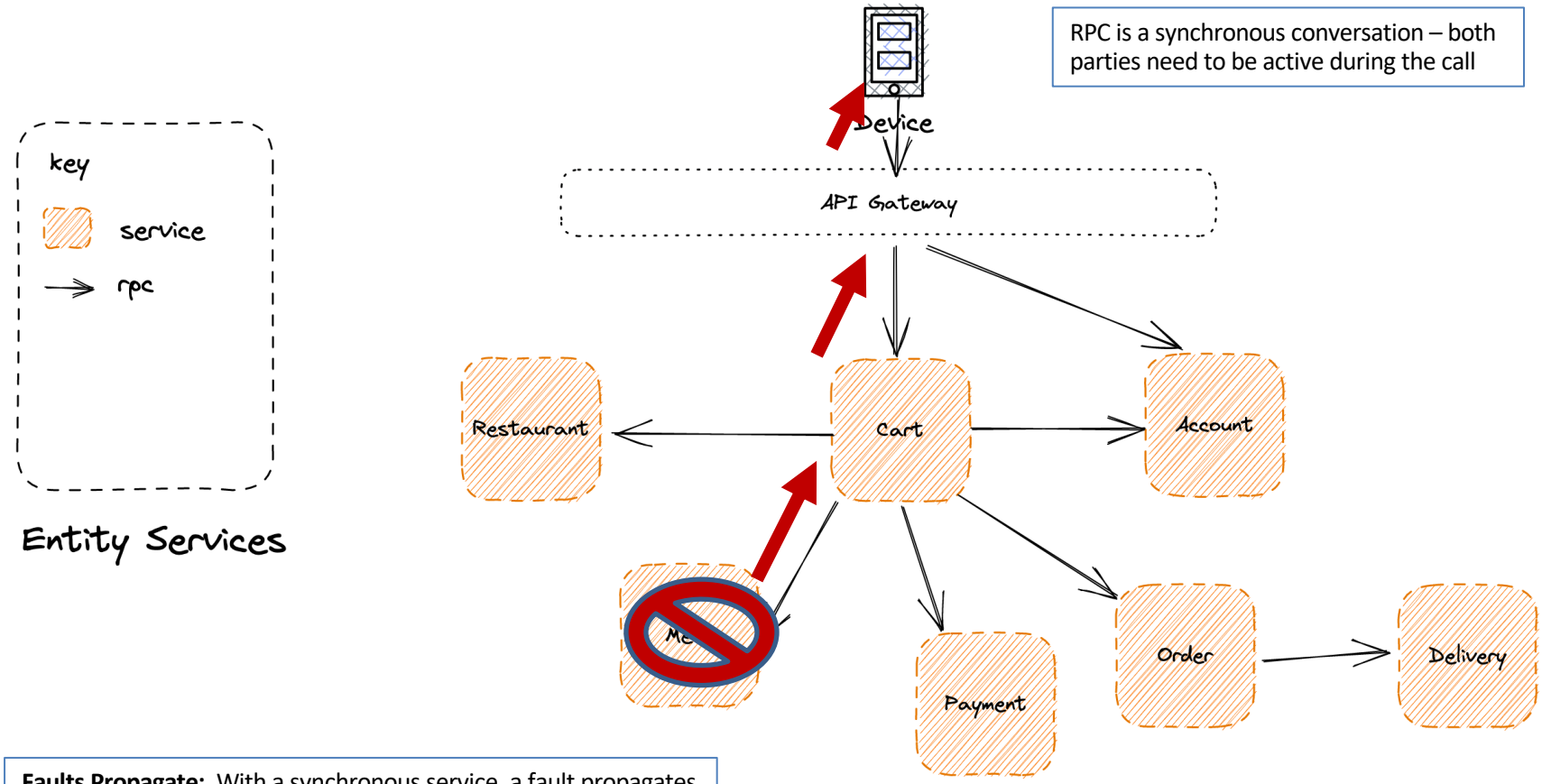
SOA & OO: SOA creates OO-like components, they encapsulate their data and expose behavior that is coupled to that data. This is the Web Services approach to services.

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*.



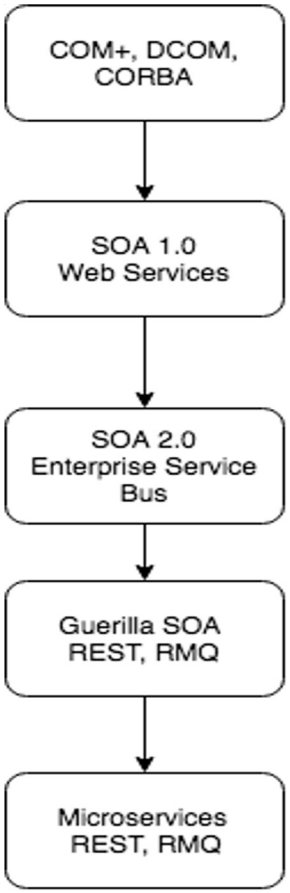
SOA & OO: We expose significant resources and operations that you can perform upon them, often CRUD.

RPC is a synchronous conversation – both parties need to be active during the call



Faults Propagate: With a synchronous service, a fault propagates

A Brief History of Autonomous Components



Distributed Object Models dominate, a merger of RPC with OO

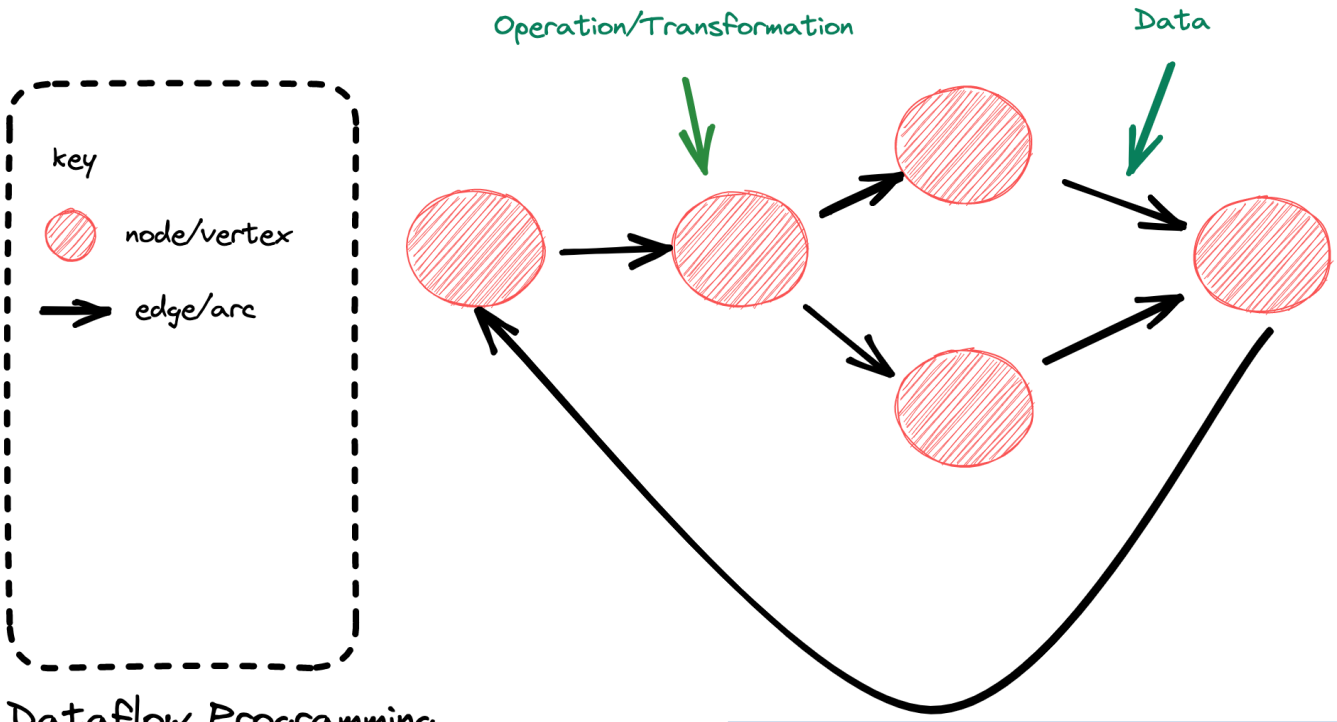
Don Box at Microsoft creates SOAP and the 4 tenets of SOA, launching the popularity of Web Service based SOA

Events emerge as the key to loosely coupled services and vendors quickly own the middleware space

Jim Webber tries to break SOA away from centralized governance and the ESB favouring a bottom up approach using lightweight middleware or REST

Netflix under Adrian Cockcroft pioneers the Fine-Grained services model; Martin Fowler and James Lewis document as microservices

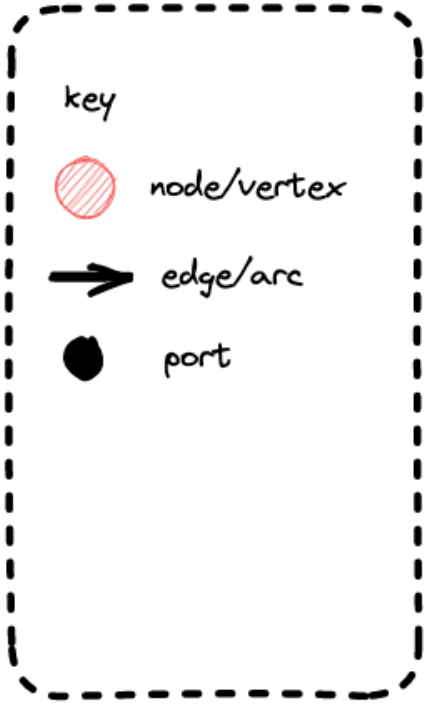
The definition of Microservices, we use comes from the paper by Martin Fowler and James Lewis



Dataflow Programming

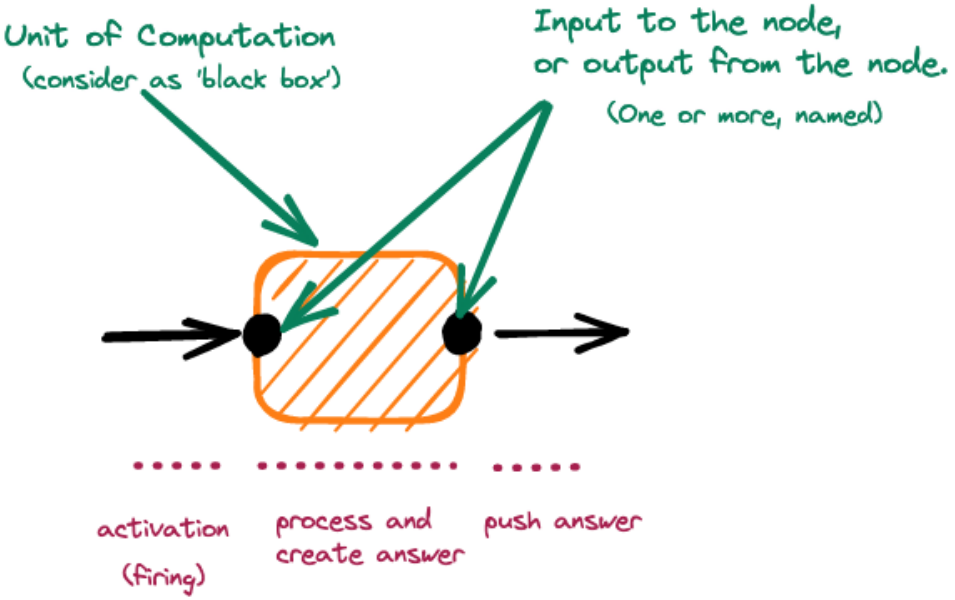
Jack Dennis et al.

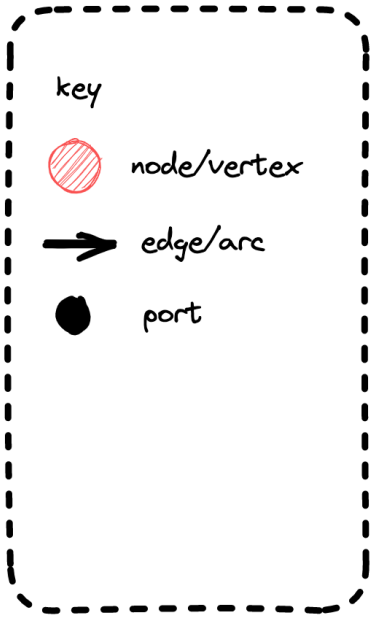
Data Flow Programming conceptualizes a program as a directed graph where operations are represented as nodes which are connected via arcs through which data flows. A node performs an operation when input data is available.

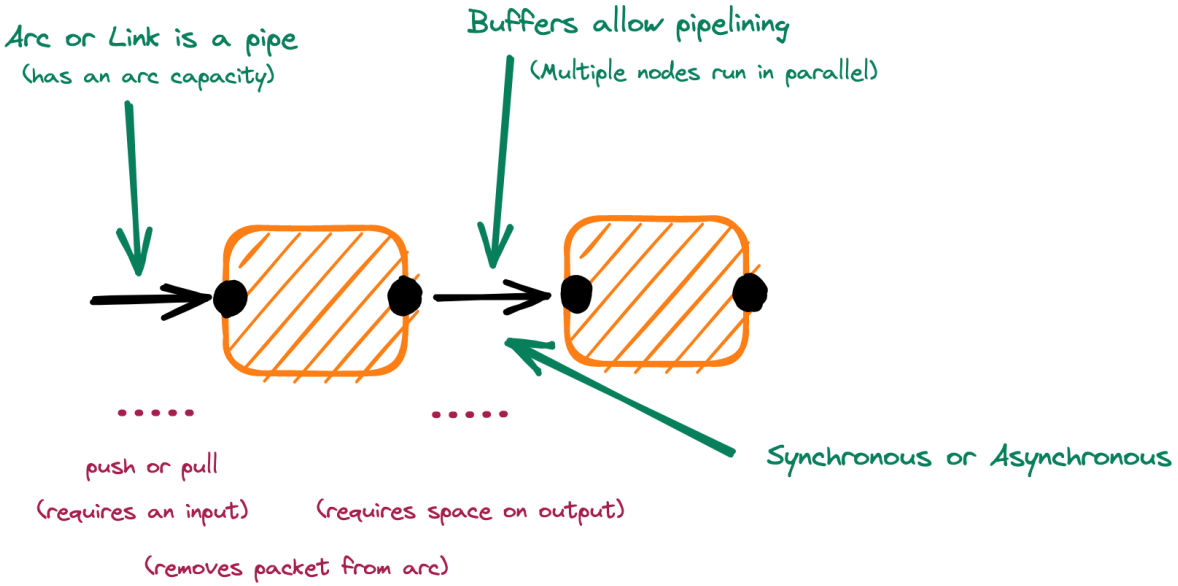


Dataflow Programming

Jack Dennis et al.

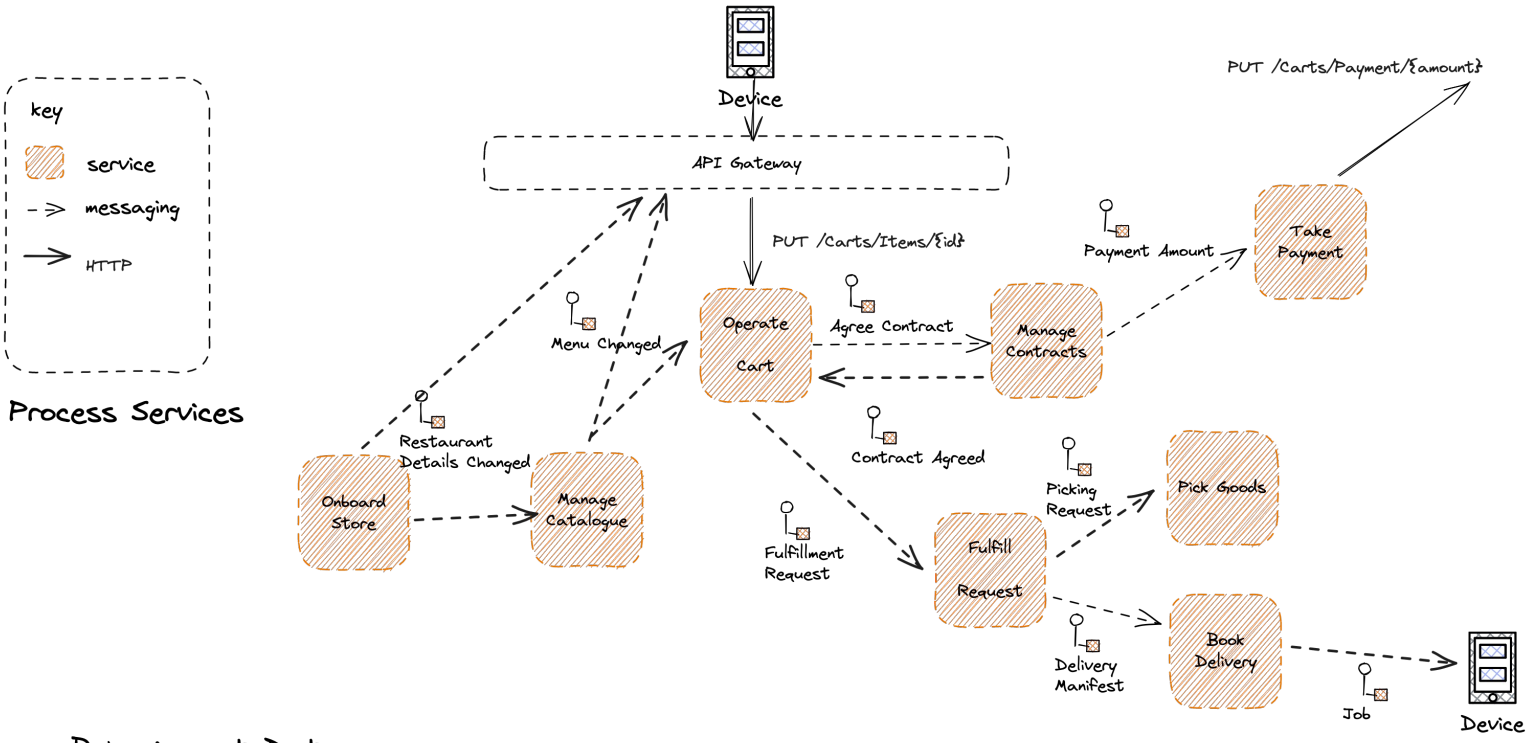






Dataflow Programming

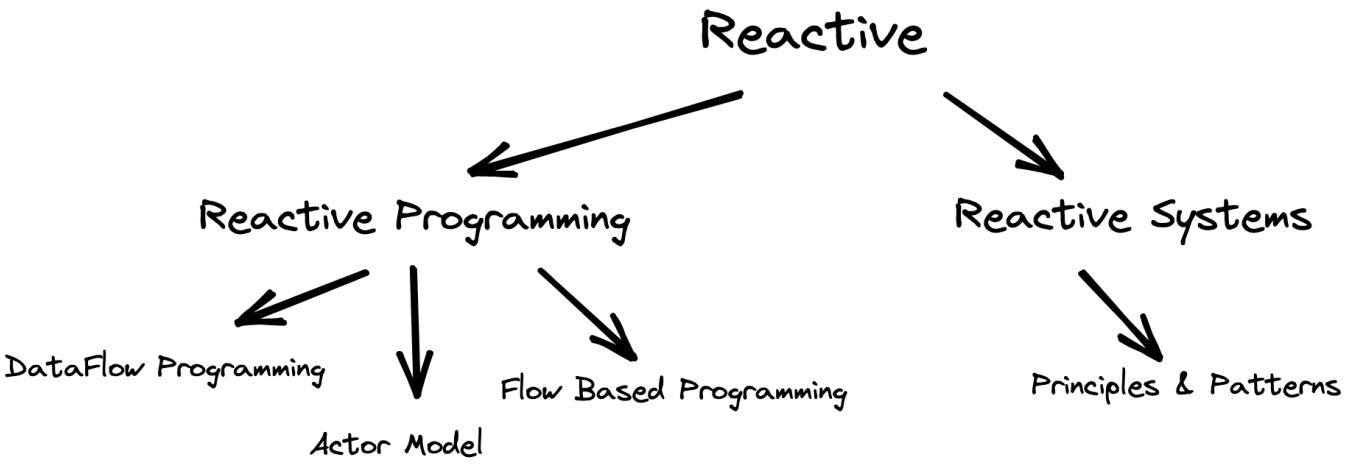
Jack Dennis et al.



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



Core Idea

τὰ πάντα ῥεῖ καὶ οὐδὲν μένει

trans: everything flows, and nothing stays

Heraclitus

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 architectures.

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 makes them easier 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. Responsiveness is the cornerstone of usability and utility, 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, mission-critical systems — any system that is not resilient 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 flow control by shaping and monitoring the message queues 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.

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)

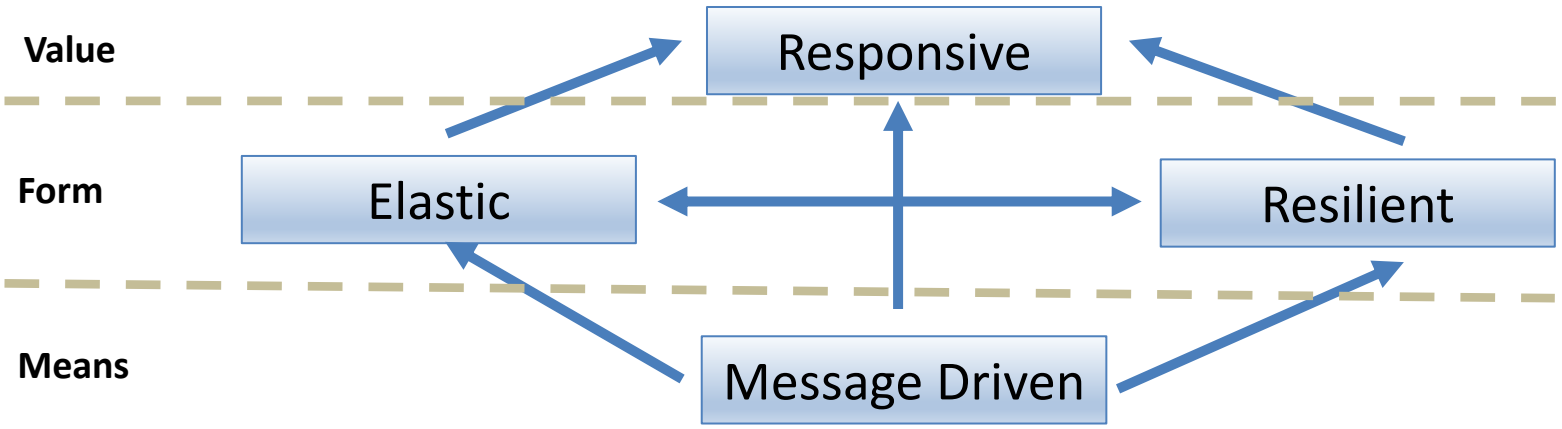
Defines an architectural style: **Reactive Applications**

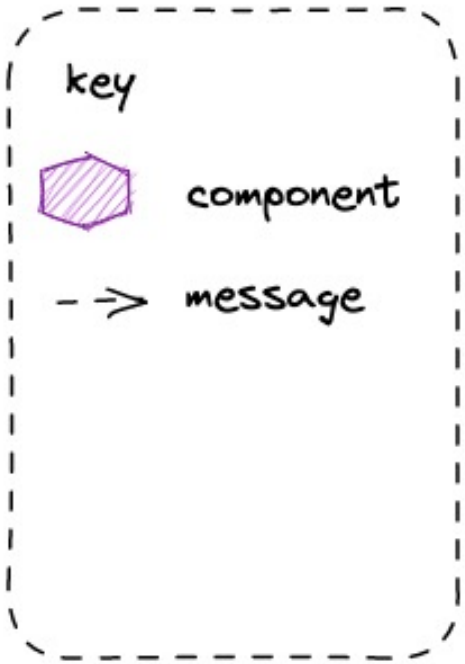
Write applications that:

- *react to events*: 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

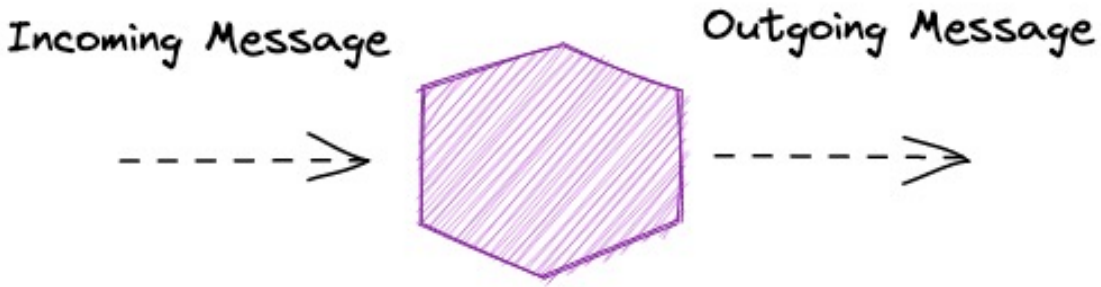
Responsive: The system responds in a timely manner.
Resilient: The system stays responsive in the presence of failure.
Elastic: The system stays responsive under varying workload.
Message Driven: Rely on asynchronous message passing

https://www.reactivemanifesto.org/

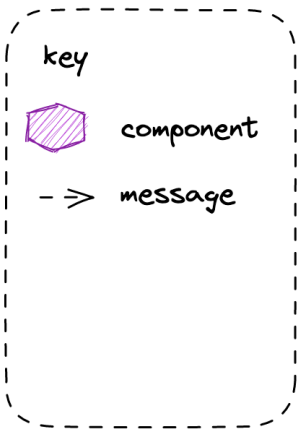

Value
Form
Means
Responsive
Elastic
Resilient
Message Driven



Message Passing

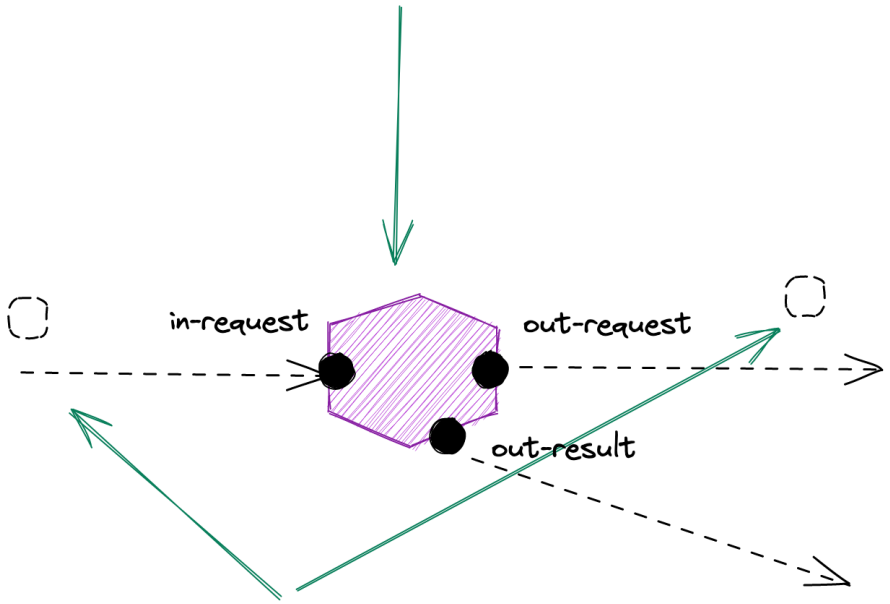


Message Passing is an asynchronous method of communication – both parties do not have to be simultaneously present for communication to occur, instead mail is delivered to 'mailbox' of some form for later retrieval.



Message Passing

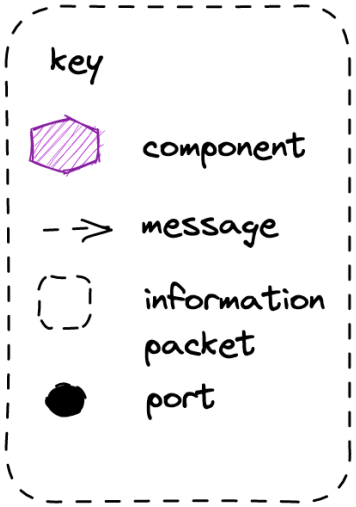
Service should focus on a transformation to the data
Usually a verb + noun combination
Place Order or Onboard Restaurant

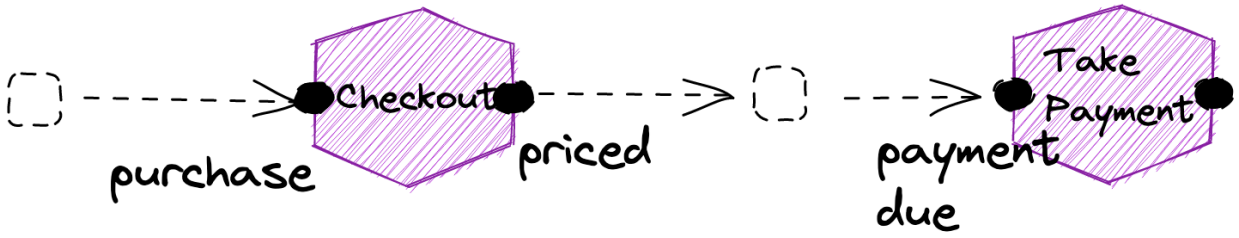


Things I am being told, a command

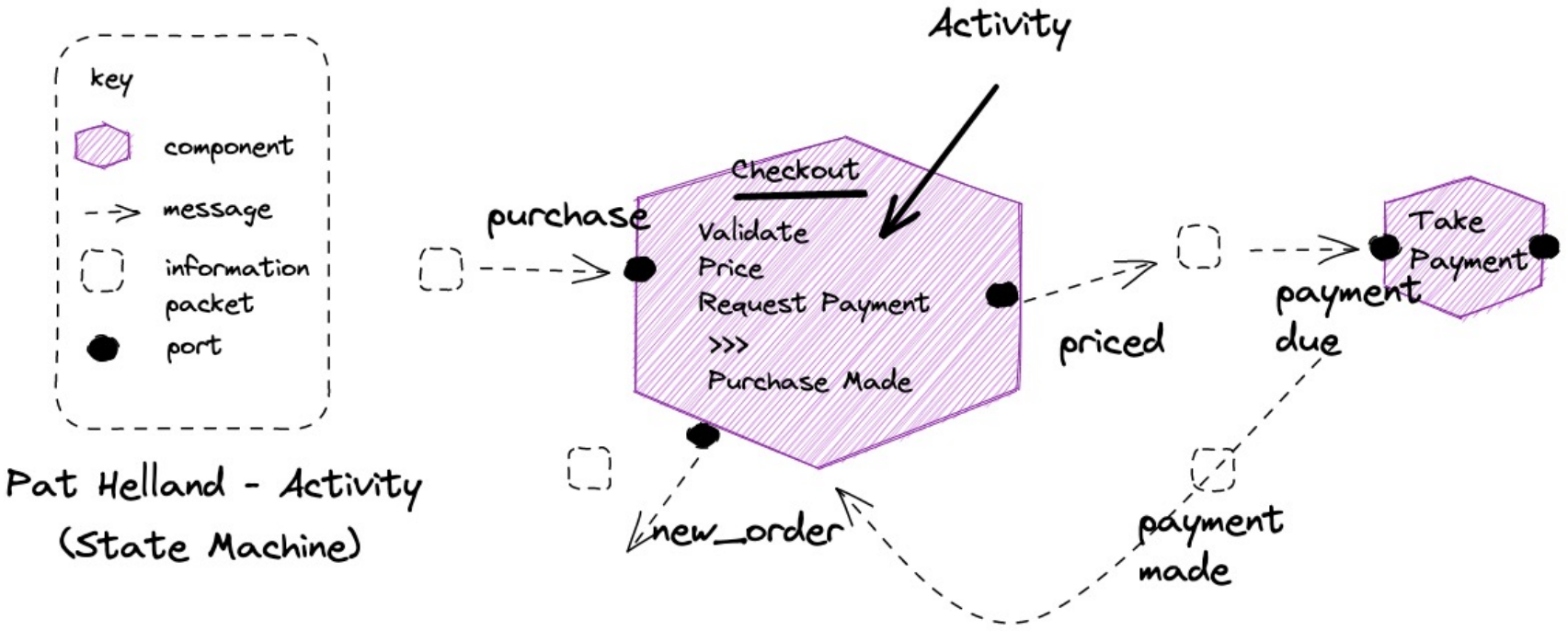
Things I know, facts, an event

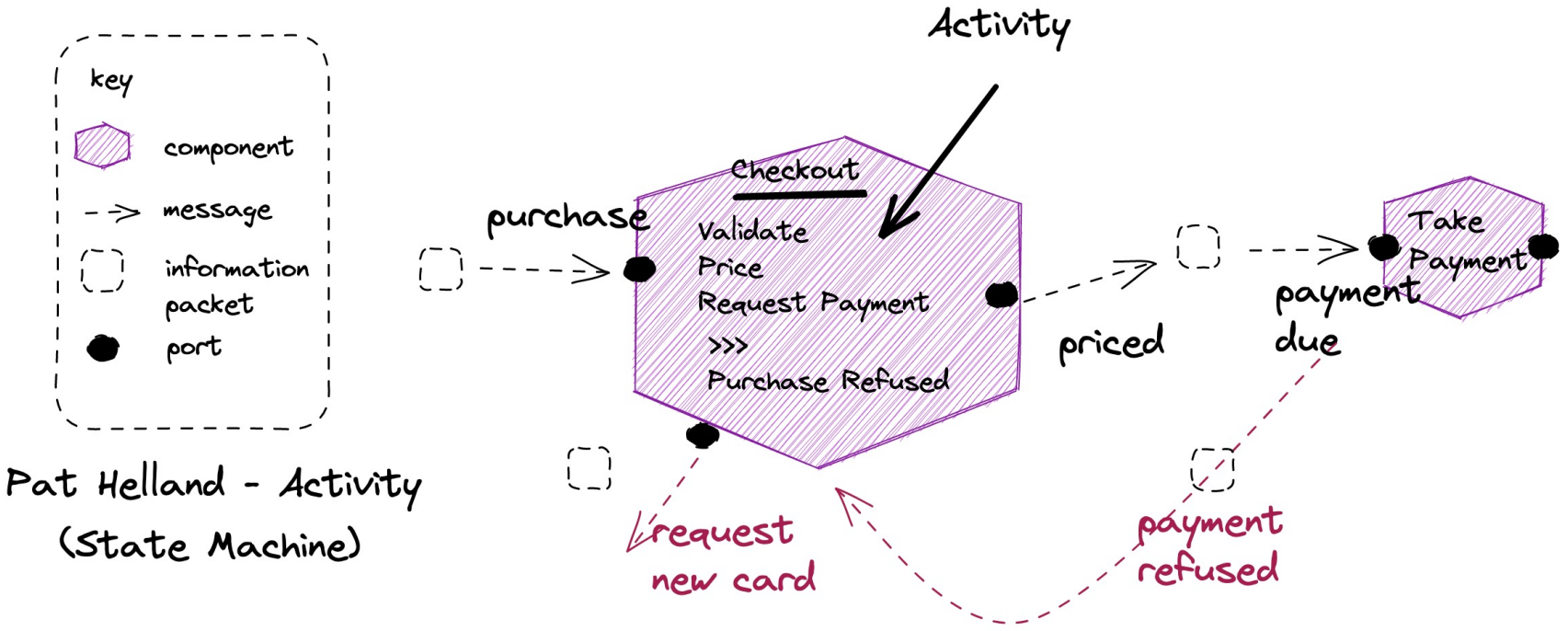
Coordinate Dataflow A reactive *principle* where we "orchestrate a continuous steady flow of information" focusing on division by behavior, not structure



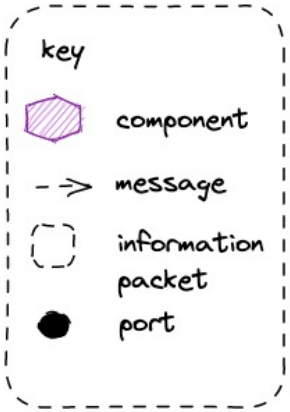


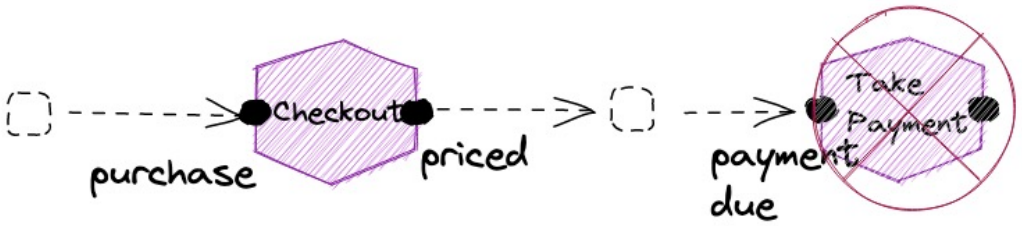
Partitioning Partition to take advantage of parallelism in the system, tasks that could be done concurrently with each other


key
component
message
information packet
port
Activity
Checkout
purchase
Validate
Price
Request Payment
>>>
Purchase Made
priced
Take Payment
payment due
new_order
payment made
Pat Helland - Activity (State Machine)



Pat Helland - Activity (State Machine)

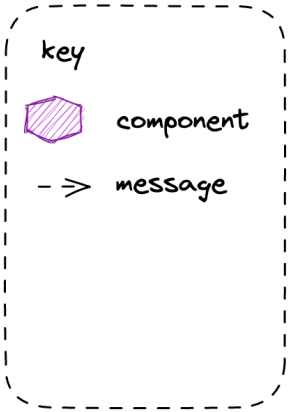




Bulkheads: In an asynchronous conversation a fault does not propagate back up the chain – we have a bulkhead that protects us against failure

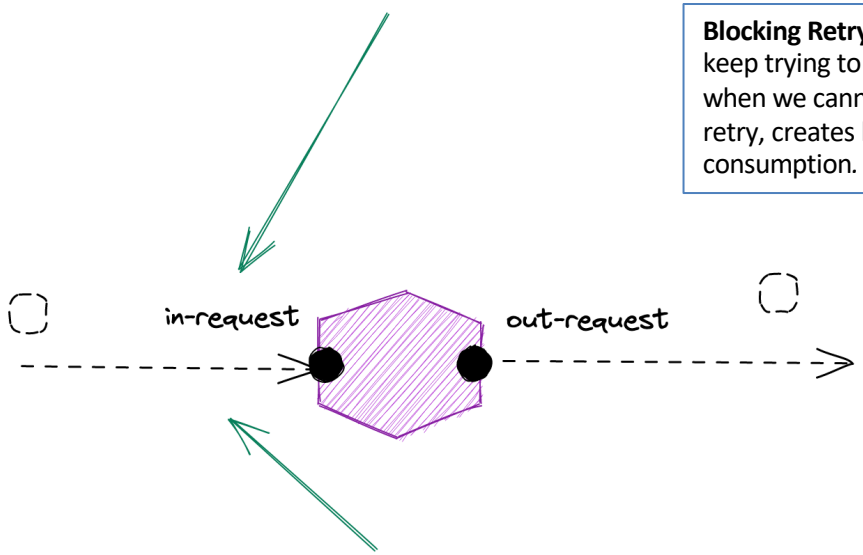
Work Queues on a Fault

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



Message Passing

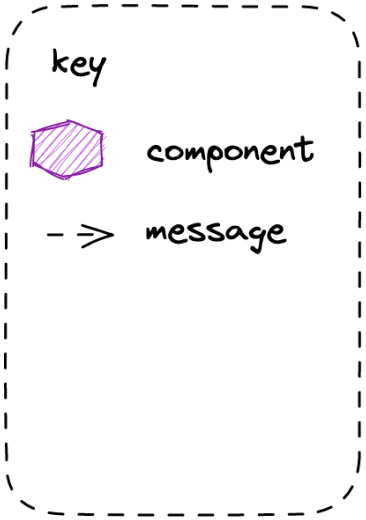
Push => open socket, middleware calls us as messages arrive
Pull => We poll for messages



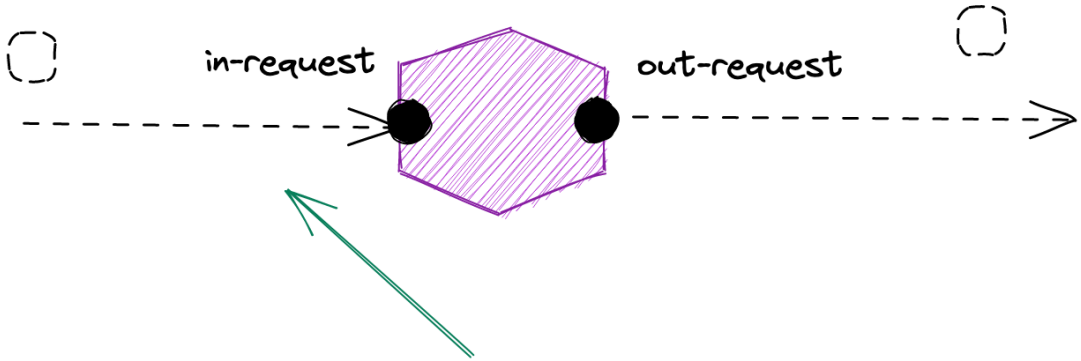
Blocking Retry: Blocking Retry – when we keep trying to process a message, such as when we cannot connect to a DB and retry, creates backpressure as we slow consumption.

Backpressure

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

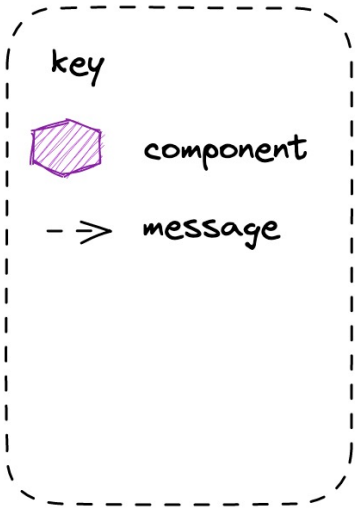


Message Passing

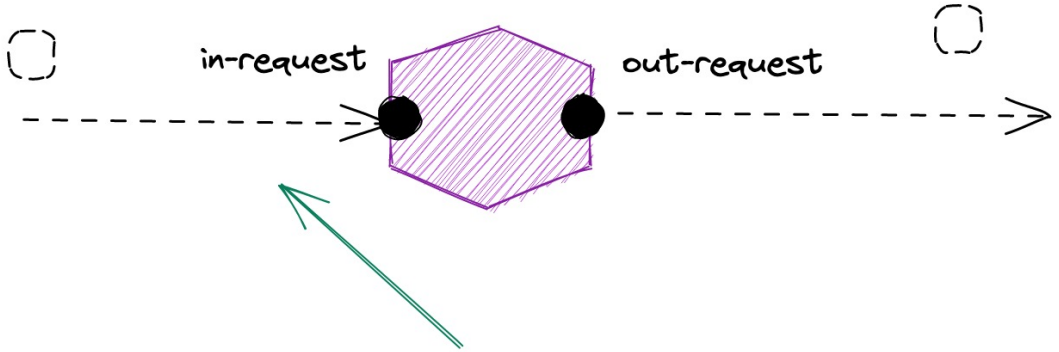


Load Shedding

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

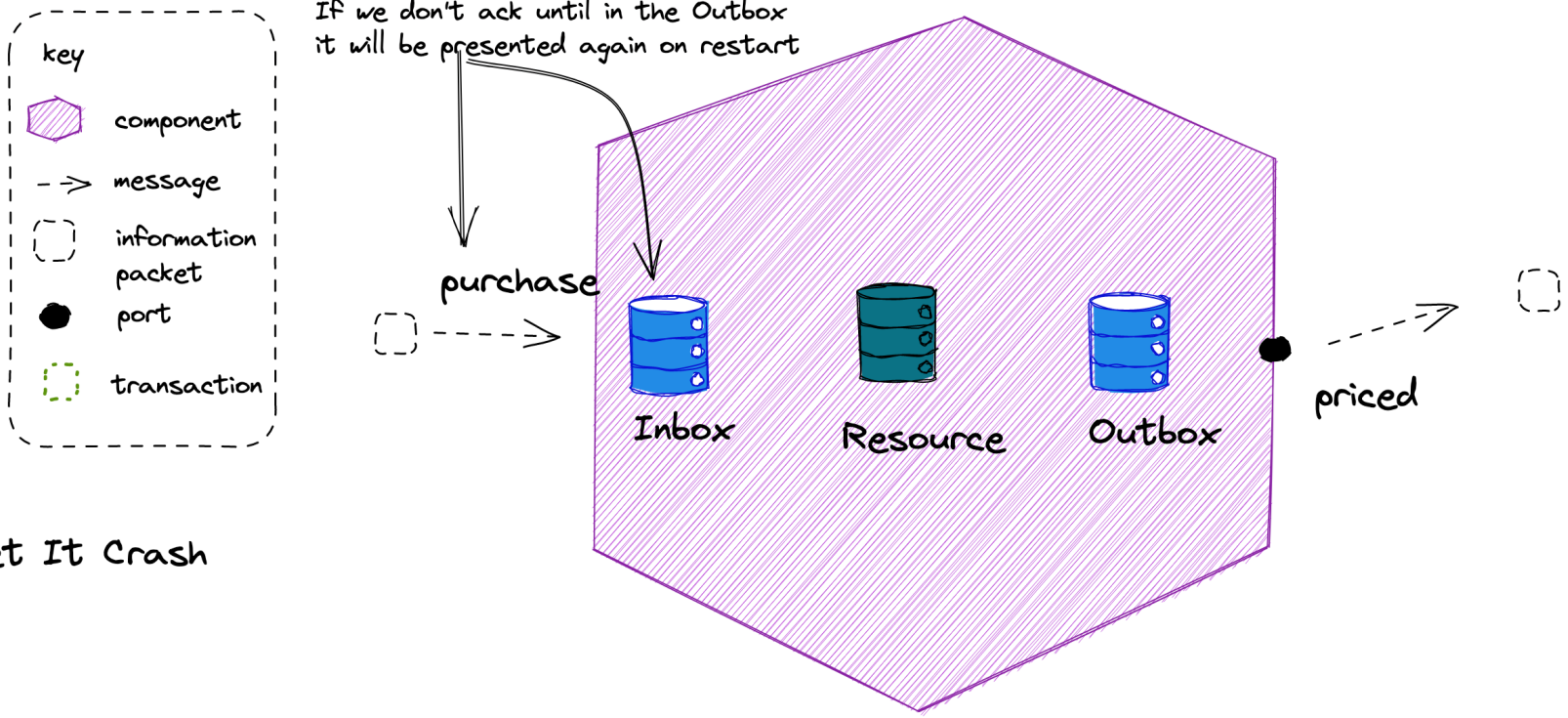
Crash and Restart =>

Start Up is Recovery

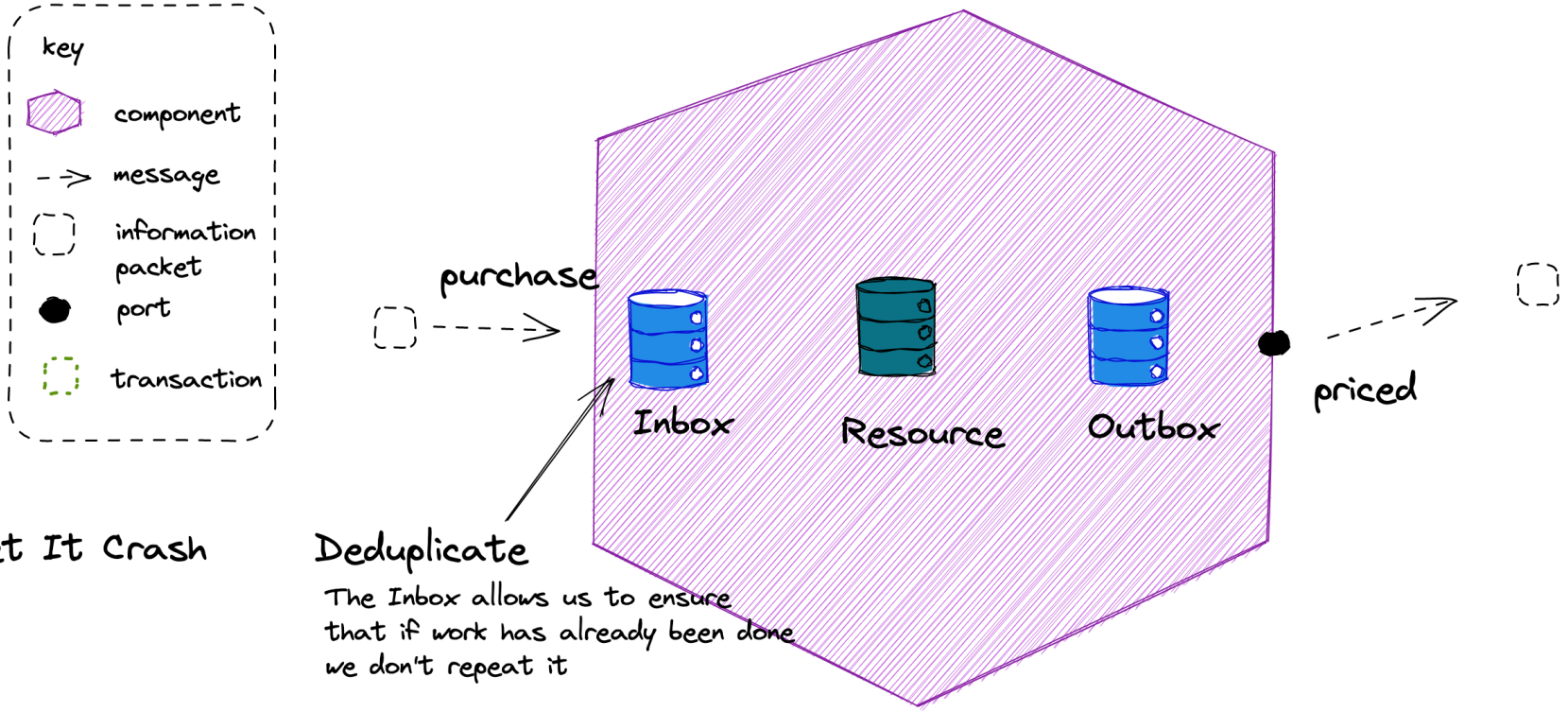
Shut Down is Clean

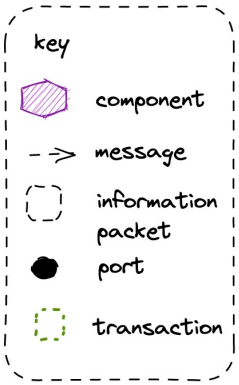
Don't Ack until Done

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

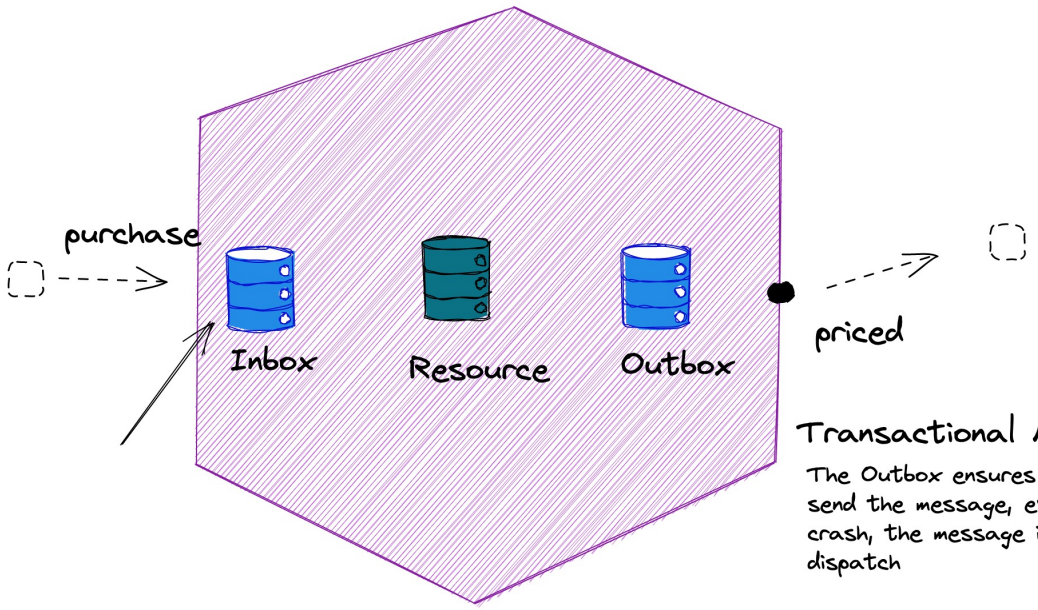


Let It Crash


key
component
message
information packet
port
transaction
purchase
Inbox
Resource
Outbox
priced
Let It Crash
Deduplicate
The Inbox allows us to ensure that if work has already been done we don't repeat it

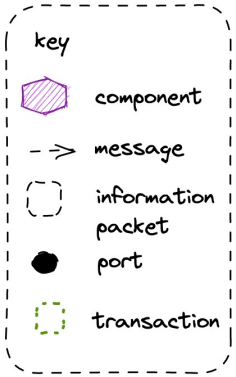


Let It Crash

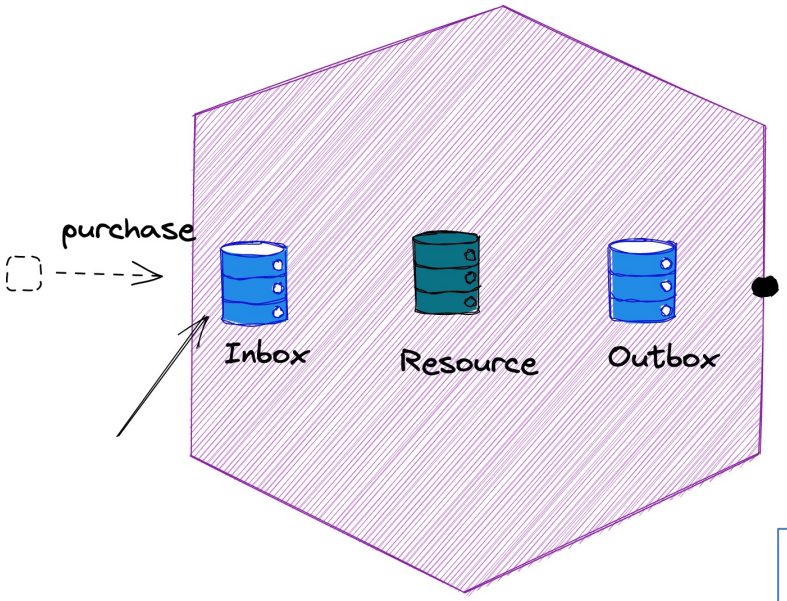


Transactional Messaging

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



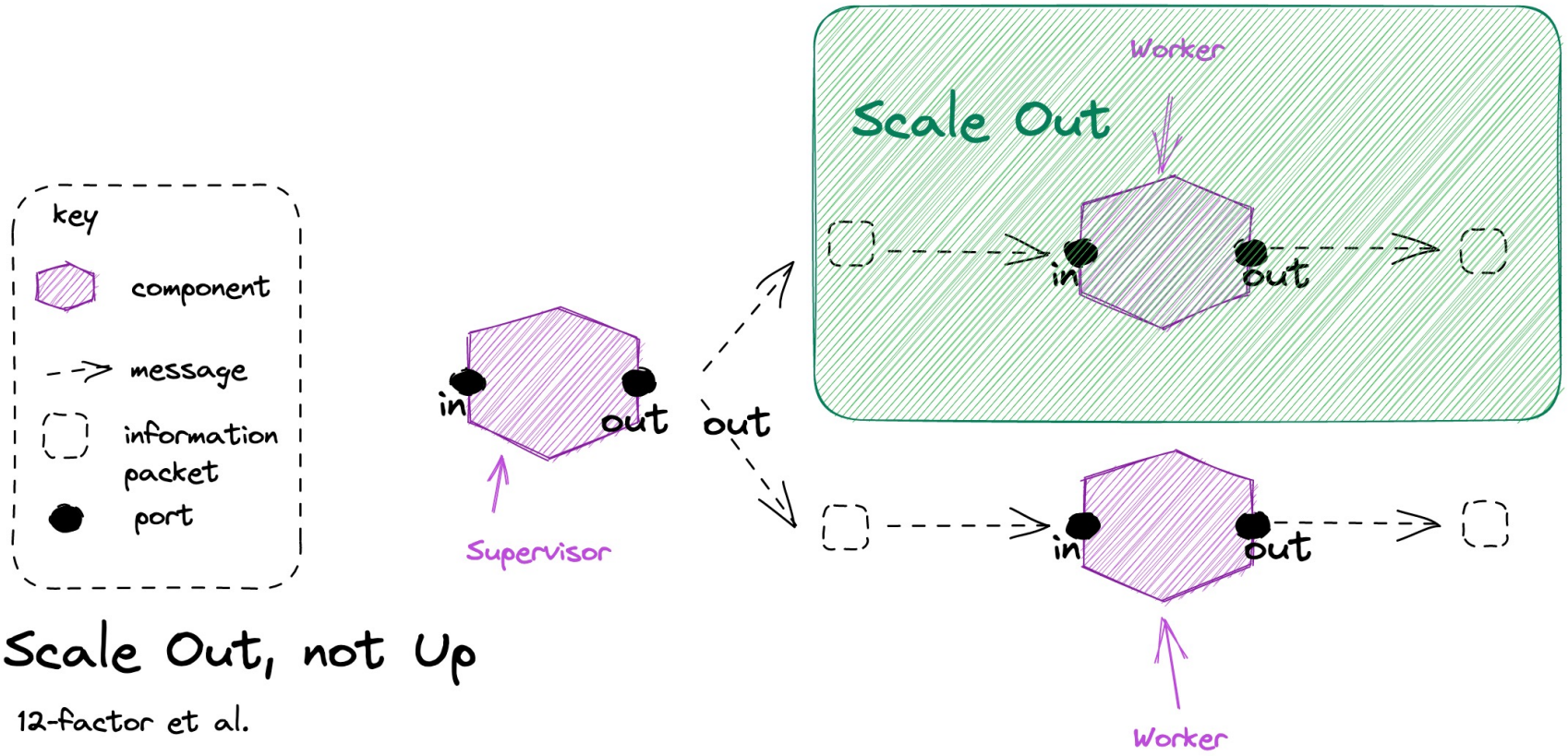
Let It Crash



Asynchronous Messaging

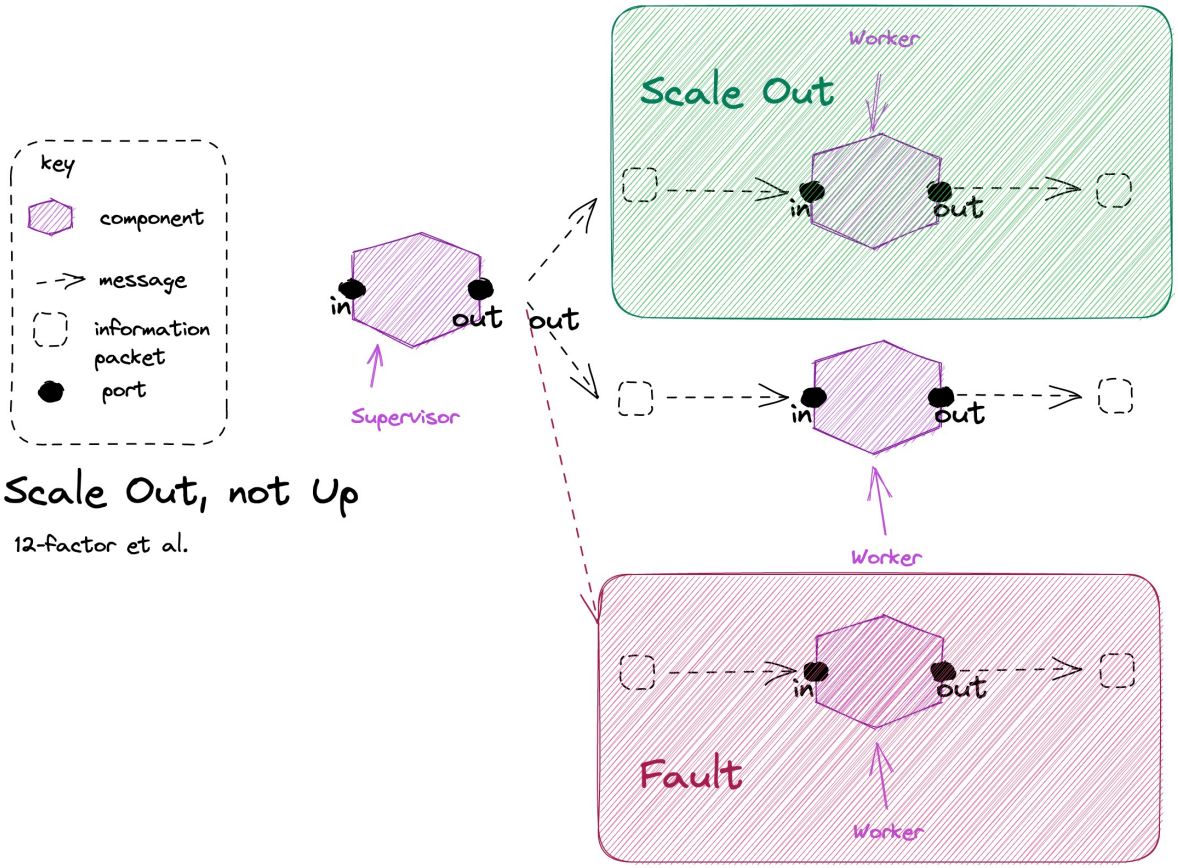
Because we communicate via messaging we are resilient to the use of "Let it Crash" by other components because we queue messages for their restart

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



Scale Out, not Up

12-factor et al.

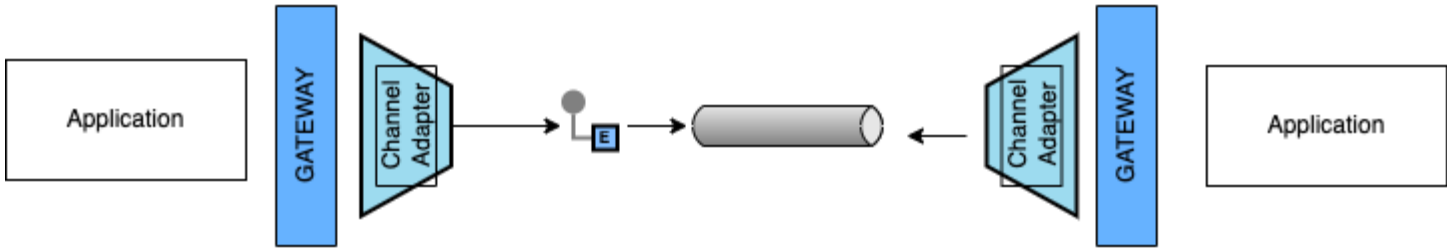


Scale Out, not Up

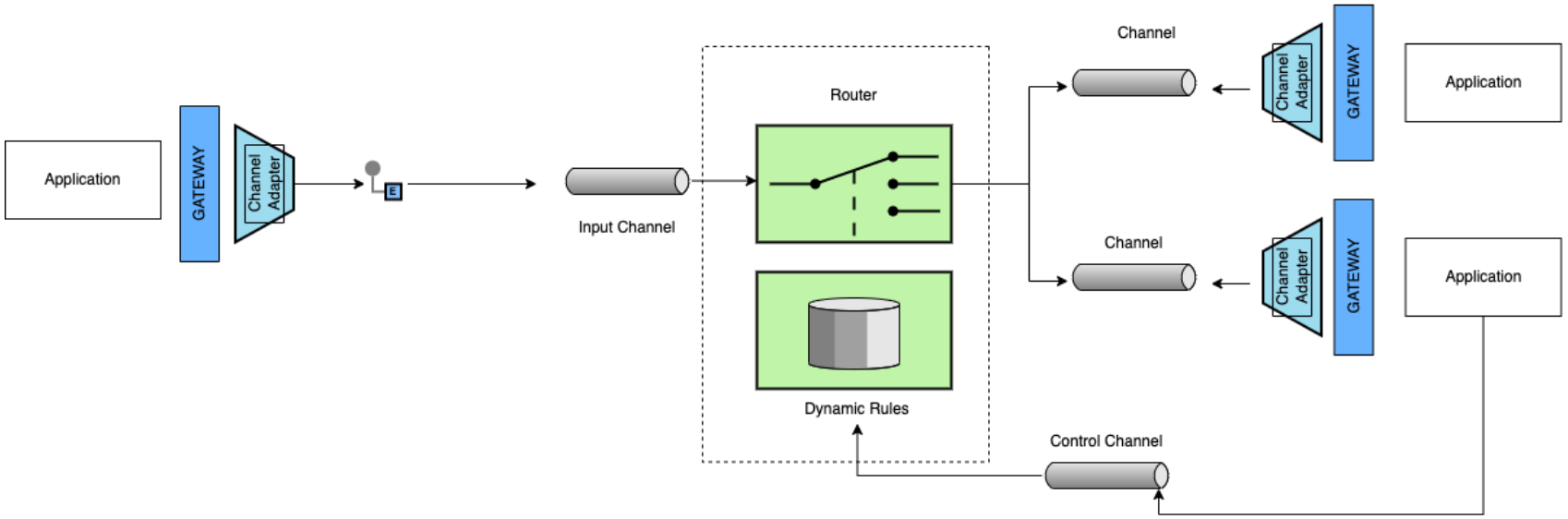
12-factor et al.

Integrating using events

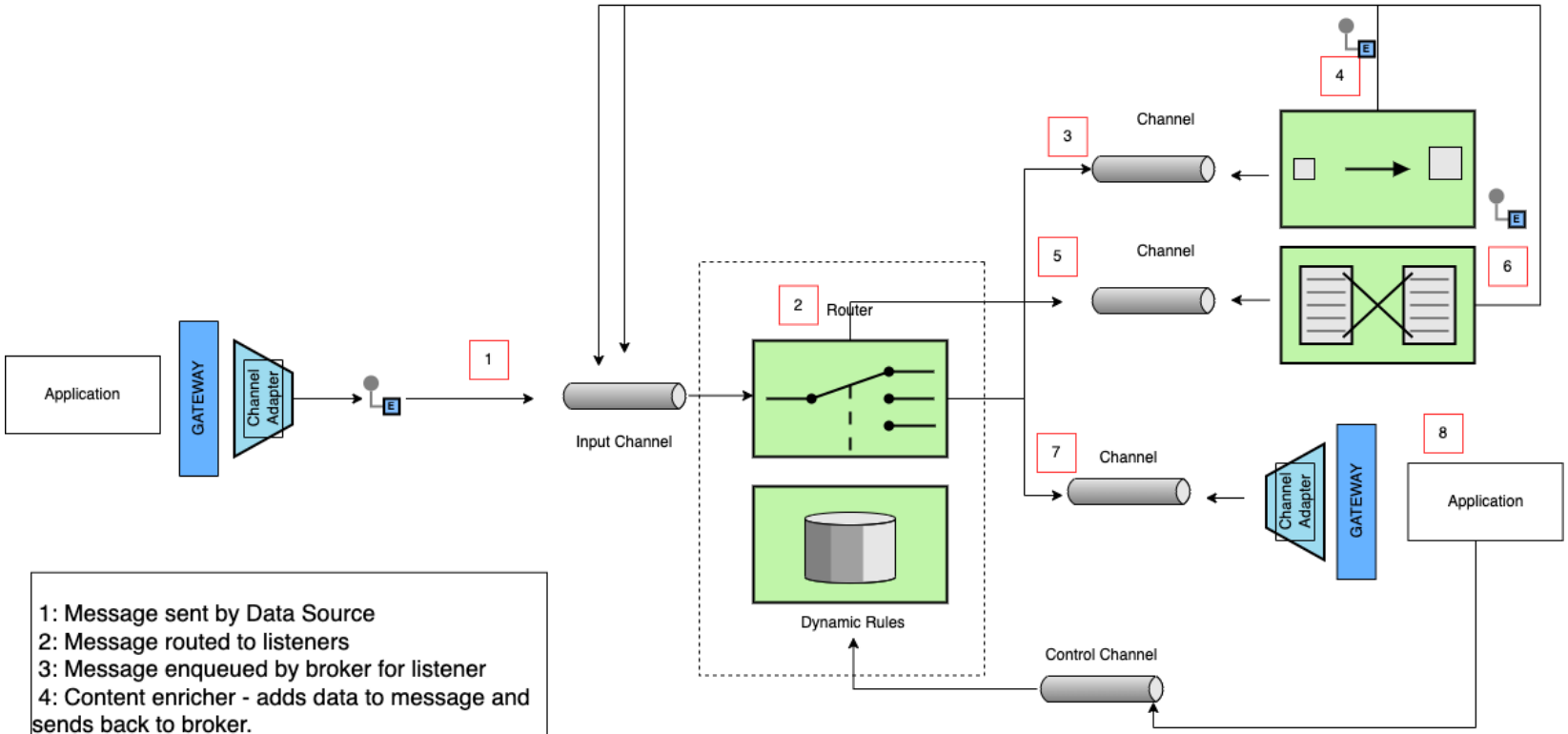
MESSAGING PATTERNS


Application
GATEWAY
Channel Adapter
Channel Adapter
GATEWAY
Application

Messaging with A Broker



Pipes and Filters



1: Message sent by Data Source
2: Message routed to listeners
3: Message enqueued by broker for listener
4: Content enricher - adds data to message and sends back to broker.
5: Message routed and enqueued by broker for listener
6: Message translated - changes shape and sends back to broker
7: Message routed and enqueued for data sink
8: Data sink processes the final message

What is a message?

A MESSAGE

Message Construction

A message has a header and body

The body contains data for the consumer

The header contains metadata for any *filter* in the pipeline.

The header should indicate the format of the body

Break a large message into pieces as a
Message Sequence or use a Claim Check

MESSAGING AND EVENTS

Message Types

Messaging

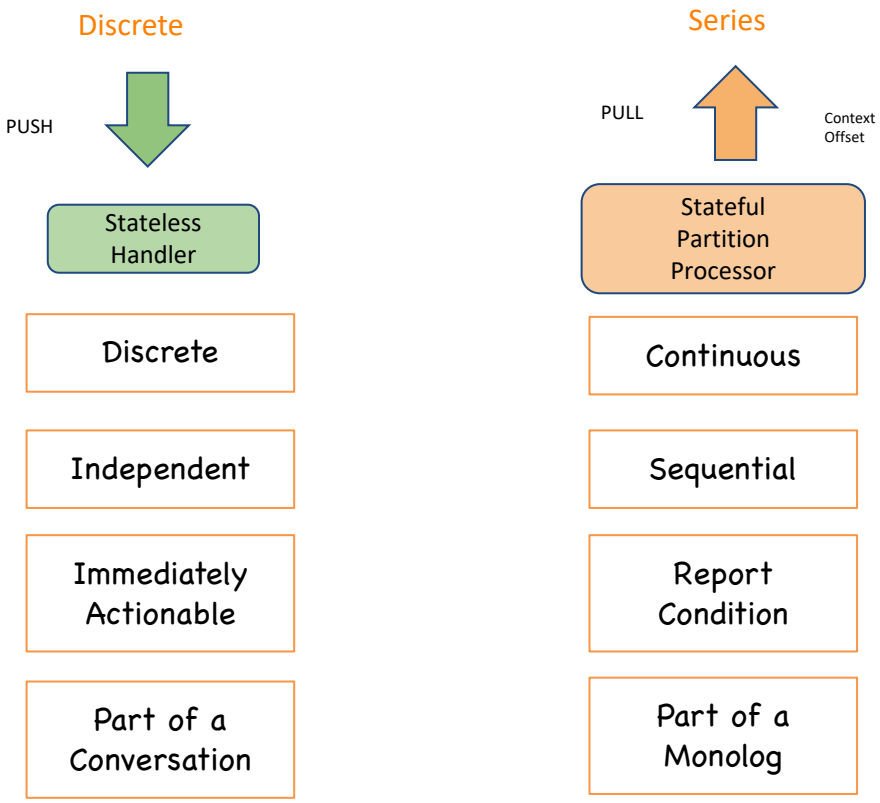
- Has Intent
- Request An Answer (Query)
  Transfer of Control (Command)
  Transfer of Value
- Part of a Workflow
  Part of a Conversation
- Concerned with the Future

Eventing

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

After Clemens Vasters https://youtu.be/ITrlLErsqzY

Eventing Types



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

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

Message Types

Messaging

- Command

Eventing

- Event (Notification)
- Document

See Gregor Hohpe: https://www.enterpriseintegrationpatterns.com/patterns/messaging/Message.html

Command Message

Use a Command Message to reliably invoke a procedure in another application

Uses the well-established pattern for encapsulating a request as an object. The Command pattern [GoF] turns a request into an object that can be stored and passed around.

Document Message

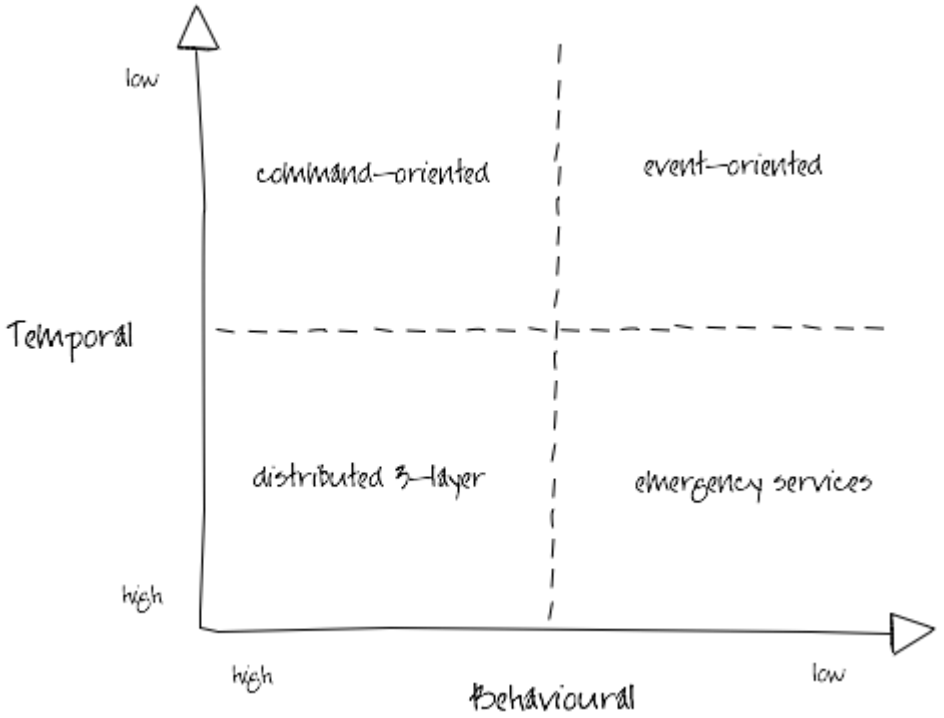
Use a Document Message to reliably transfer a data structure between applications.

The receiver decides what, if anything, to do with the data

Event Message

Use an Event Message for reliable, asynchronous event notification between applications.

The difference between an Event Message and a Document Message is a matter of timing and content. An event's contents are typically less important.



Ian Robinson: http://iansrobinson.com/2009/04/27/temporal-and-behavioural-coupling/

Self-paced material

EXERCISES

EXERCISE MATERIAL

Introduction to Exercises

- Readme
- Videos
- Scripts & Slides

Introduction to RMQ

CHANNELS

Channels

A virtual pipe that connects producer and consumer

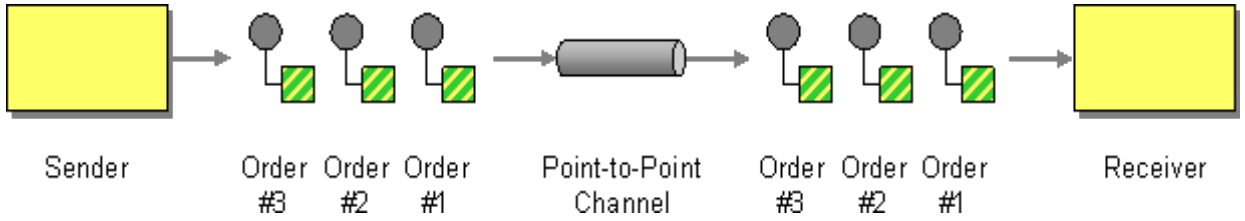
Logical Address (Topic or Routing Key)

Unidirectional

One-to-One or One-to-Many

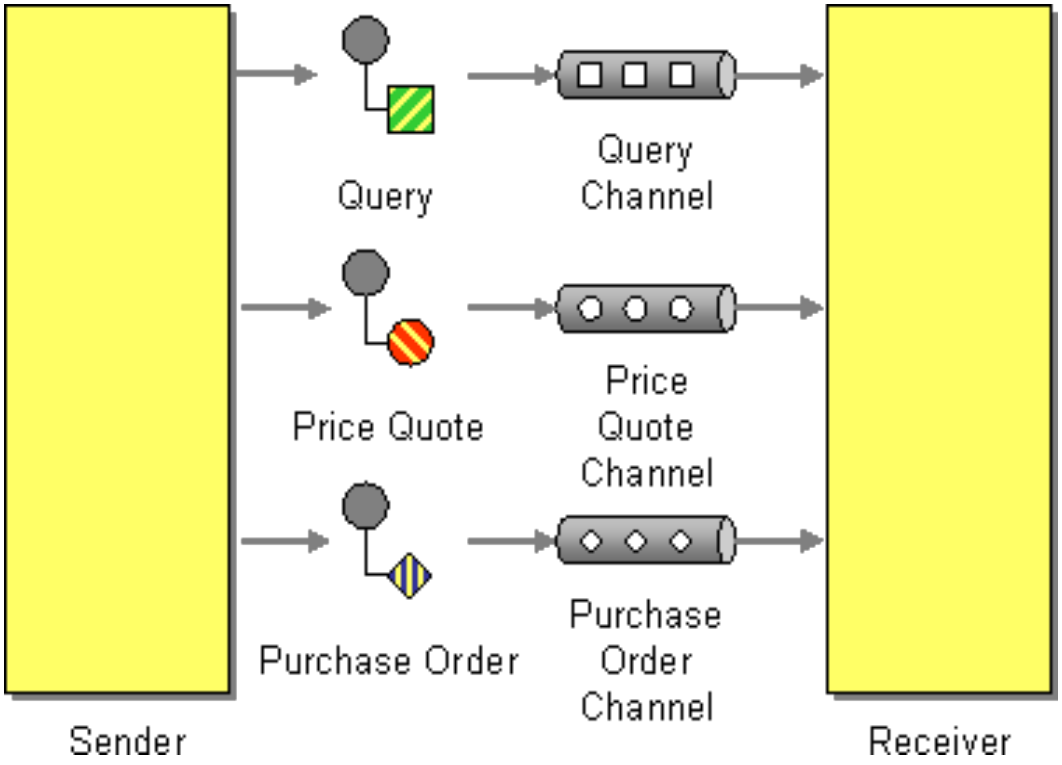
Messaging is a ‘pipe’ not a ‘bucket’.

Point-to-Point Channel



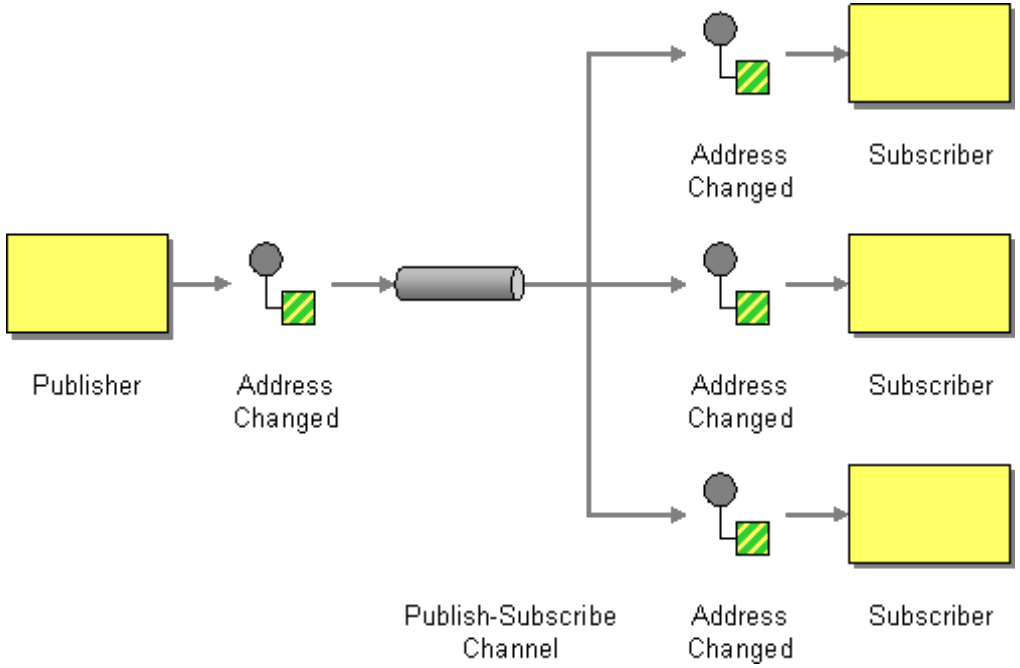
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html

Datatype Channel



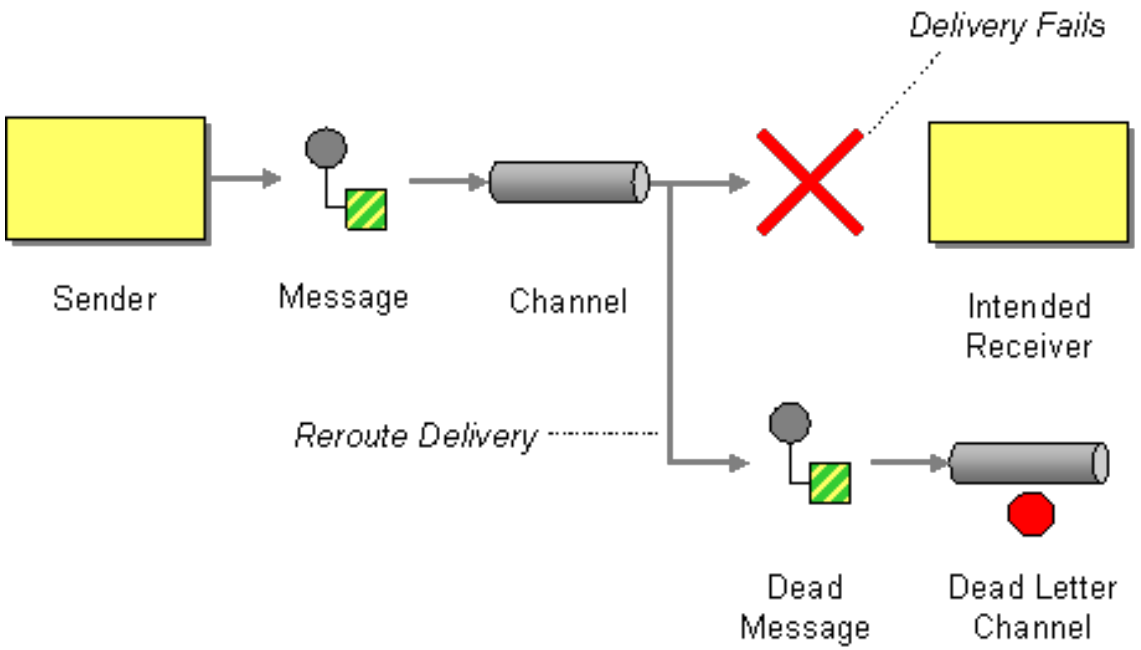
http://www.enterpriseintegrationpatterns.com/patterns/messaging/DatatypeChannel.html

Publish-Subscribe Channel



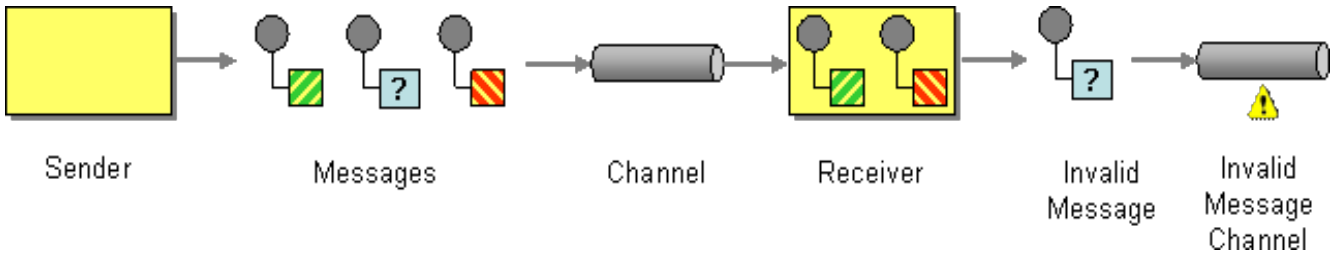
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html

Dead Letter Channel



http://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html

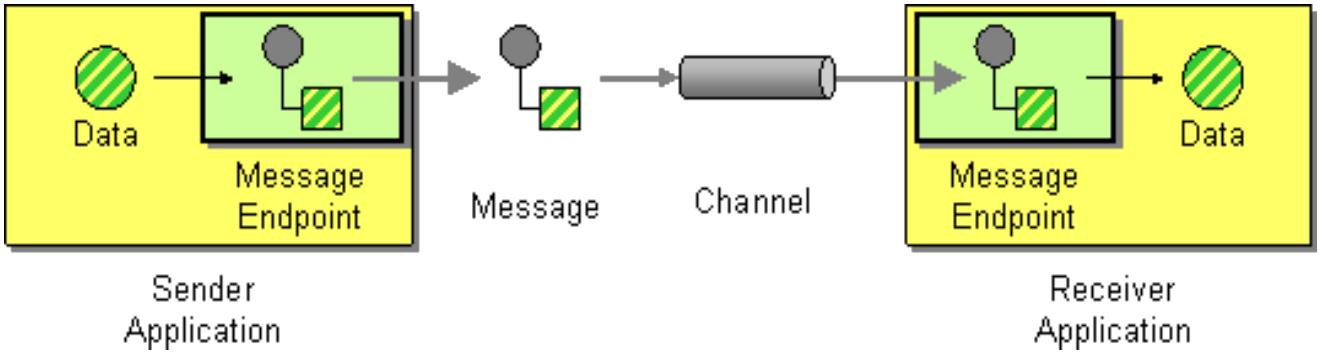
Invalid Message Channel



http://www.enterpriseintegrationpatterns.com/patterns/messaging/InvalidMessageChannel.html

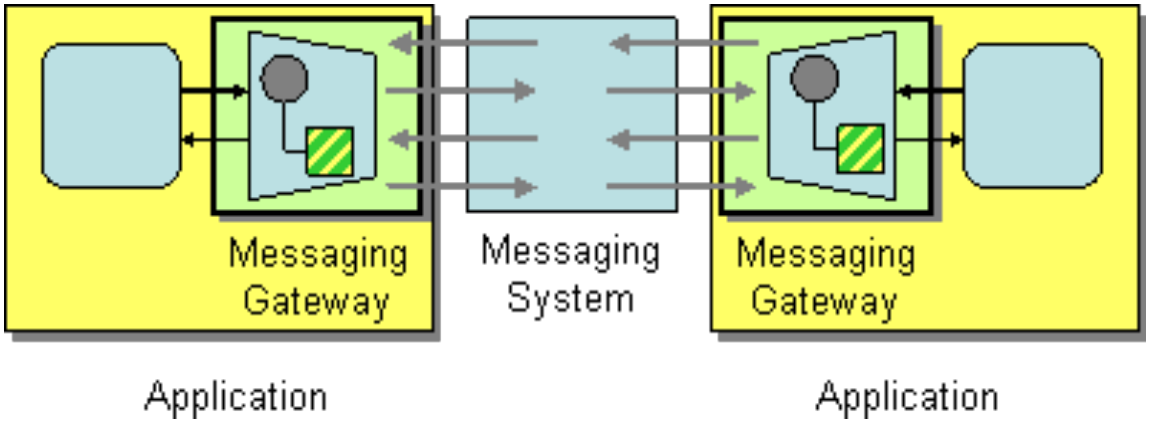
ENDPOINTS

Message Endpoint



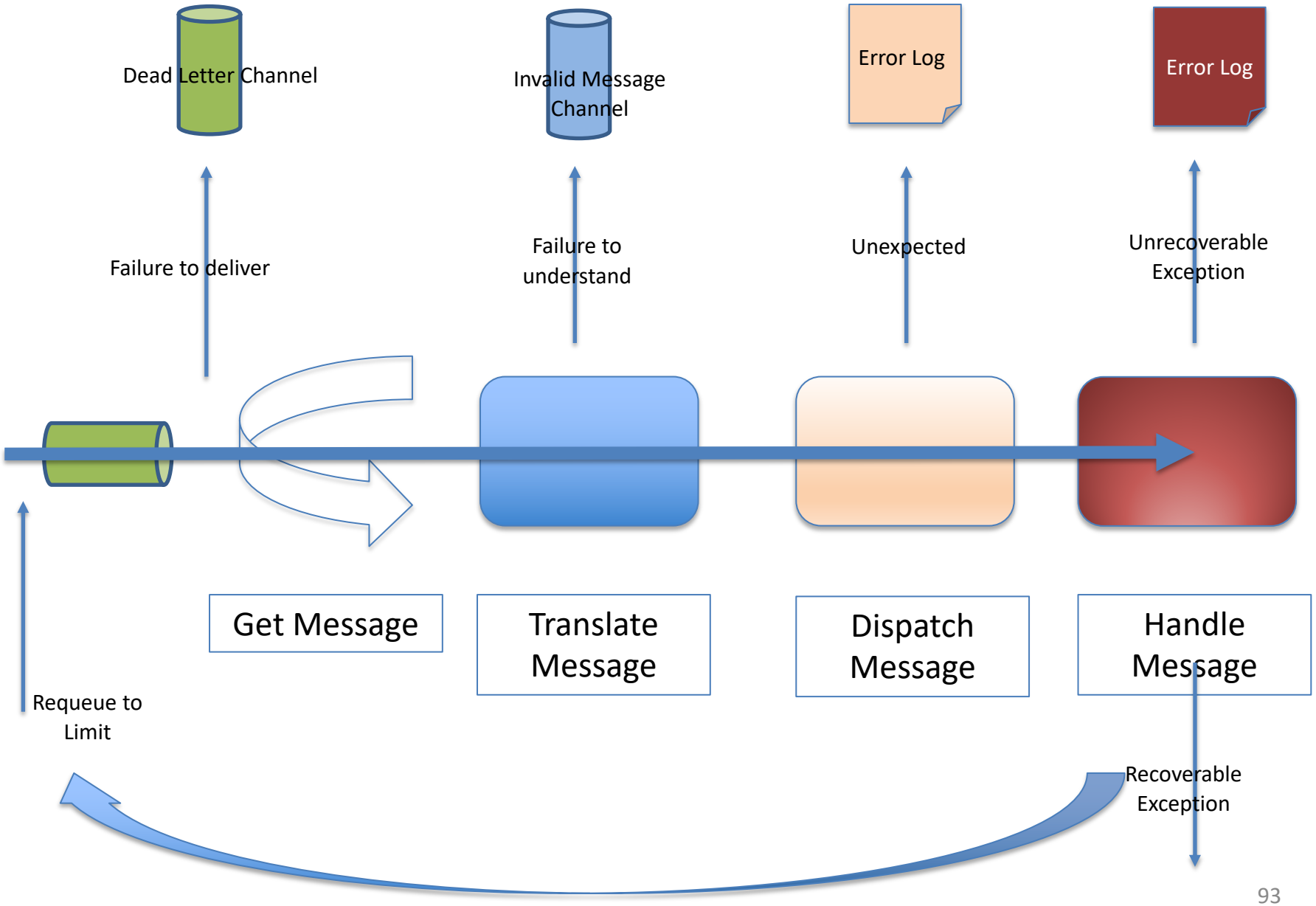
http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html

Messaging Gateway

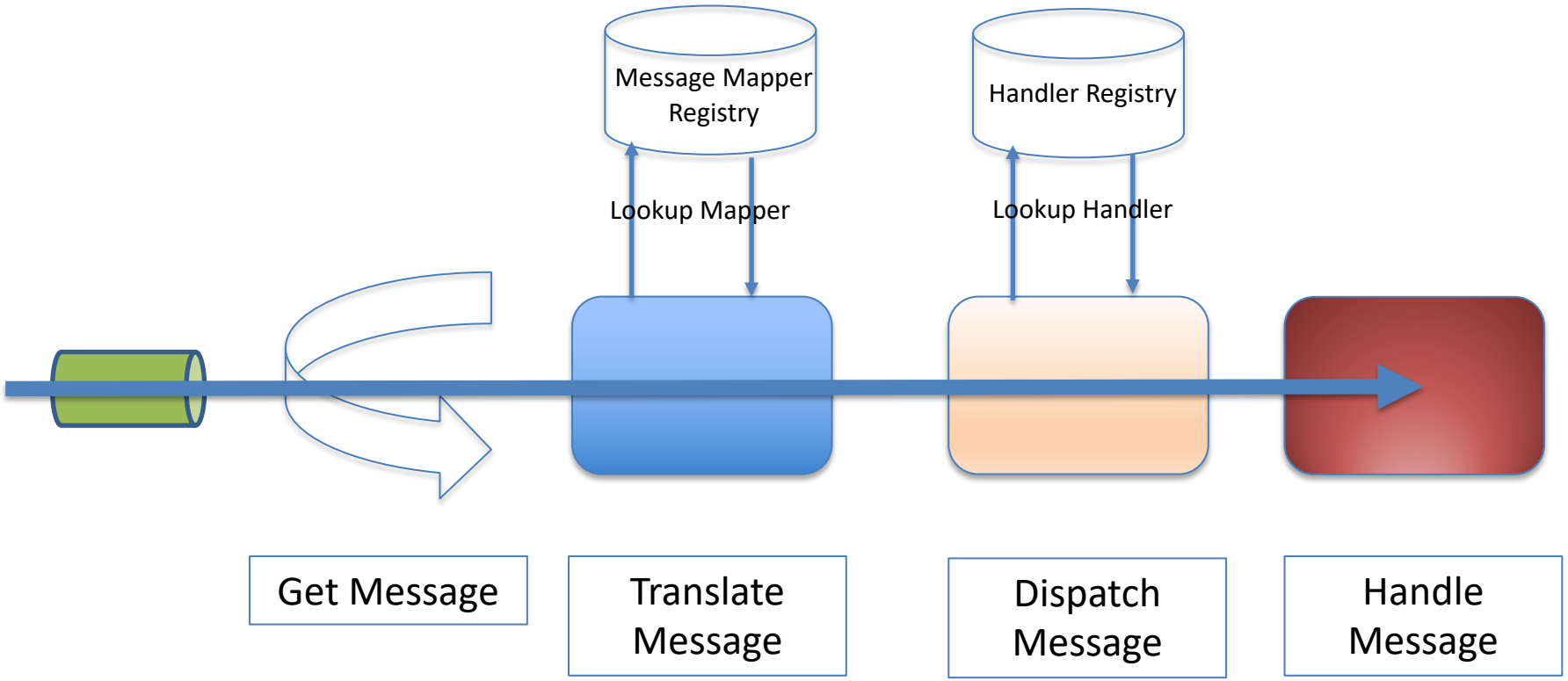


http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingGateway.html

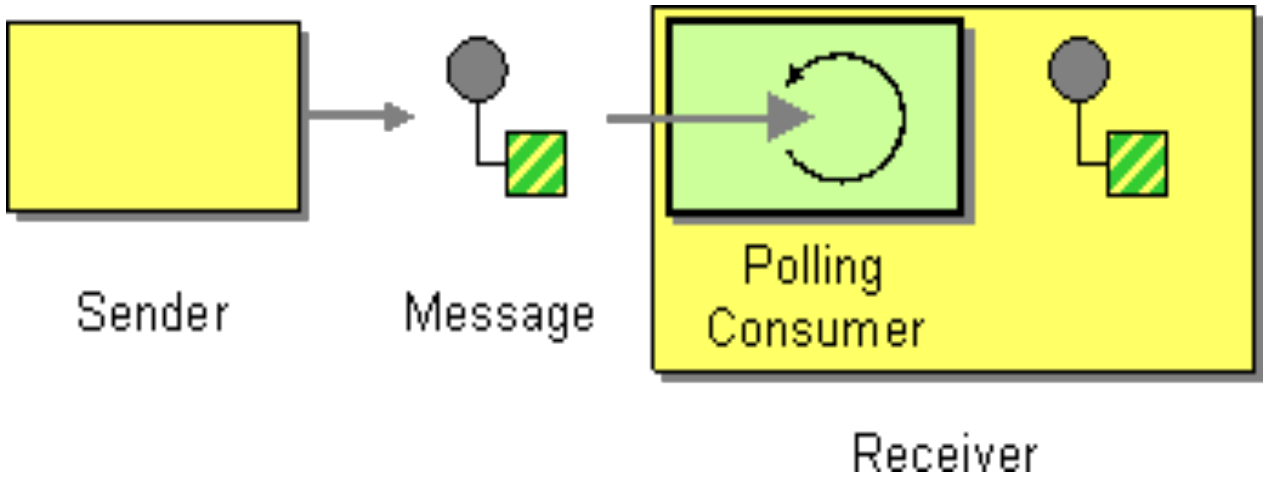
THE MESSAGE PUMP


Dead Letter Channel
Invalid Message Channel
Error Log
Error Log
Failure to deliver
Failure to understand
Unexpected
Unrecoverable Exception
Get Message
Translate Message
Dispatch Message
Handle Message
Requeue to Limit
Recoverable Exception

Translate and Dispatch

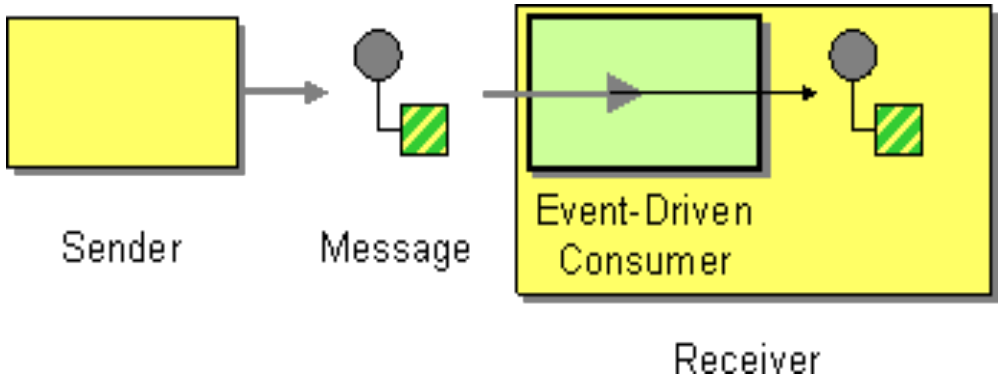


Polling Consumer



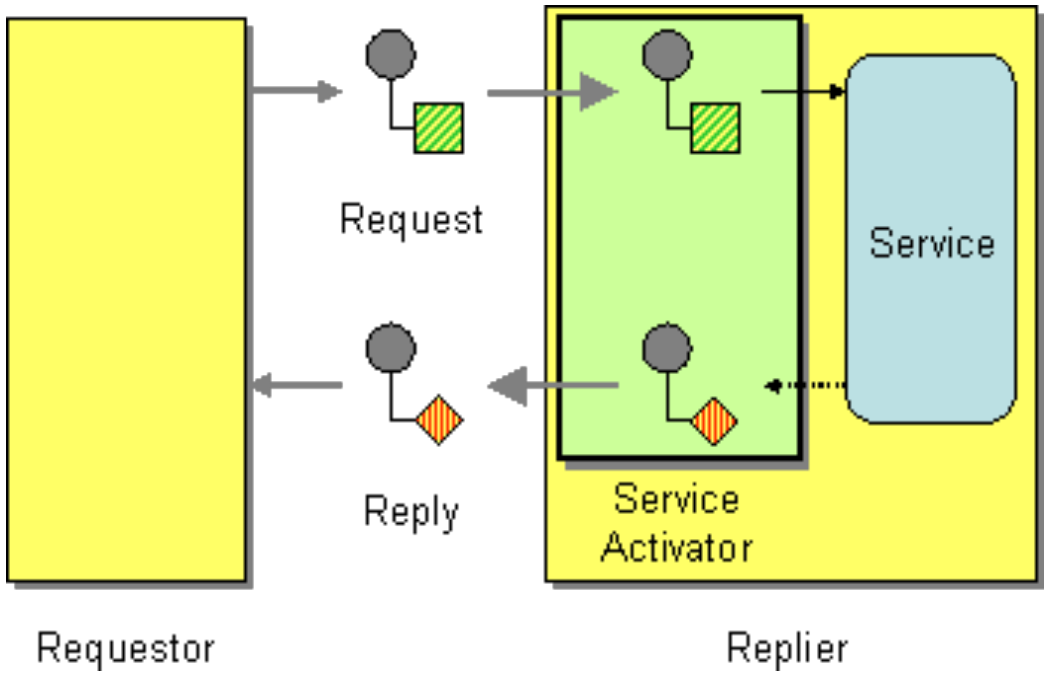
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PollingConsumer.html

Event Driven Consumer



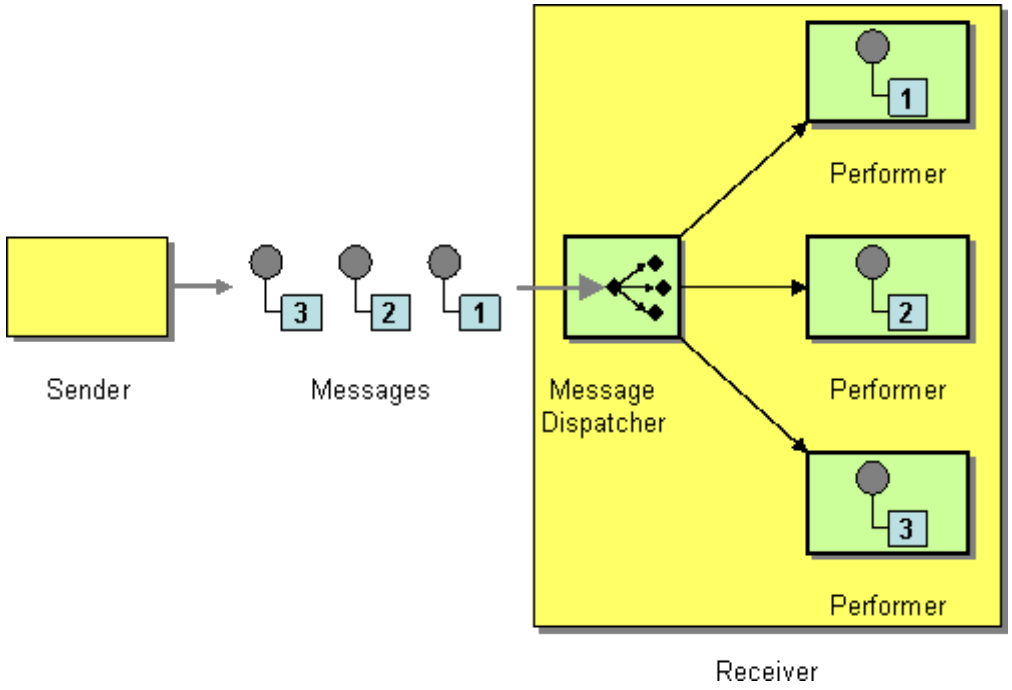
http://www.enterpriseintegrationpatterns.com/patterns/messaging/EventDrivenConsumer.html

Service Activator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingAdapter.html

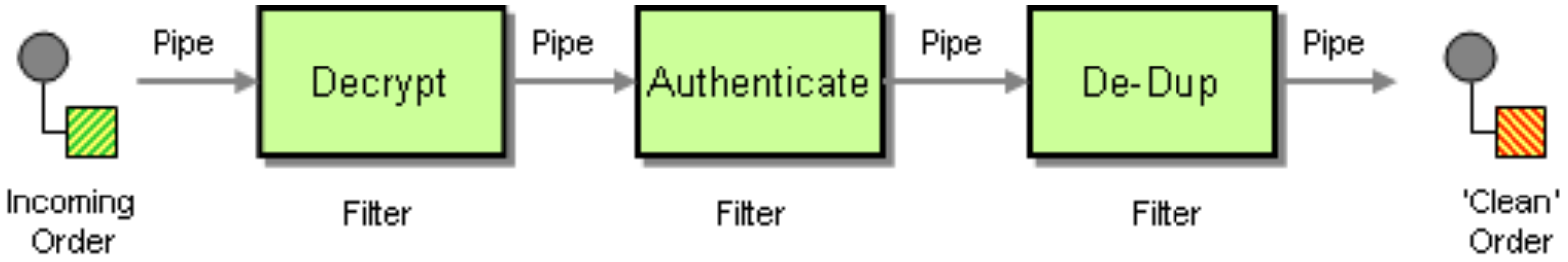
Competing Consumers



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageDispatcher.html

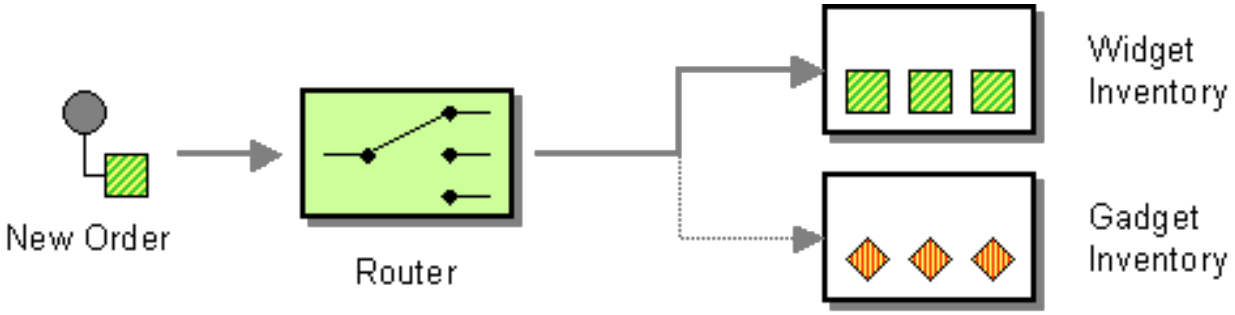
PIPELINES

Pipes and Filters



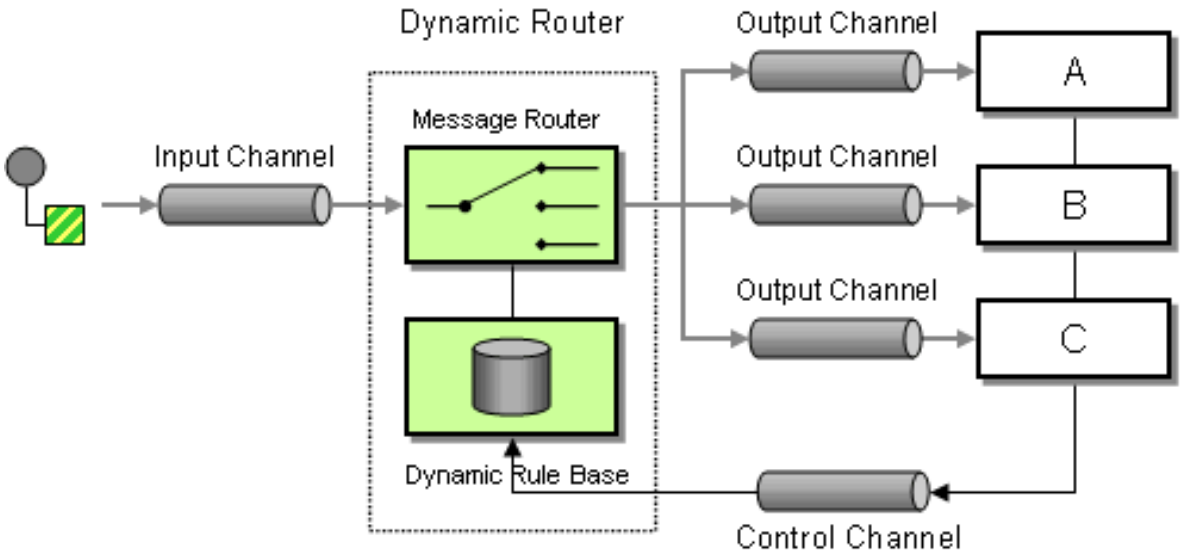
http://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html

Content Based Router



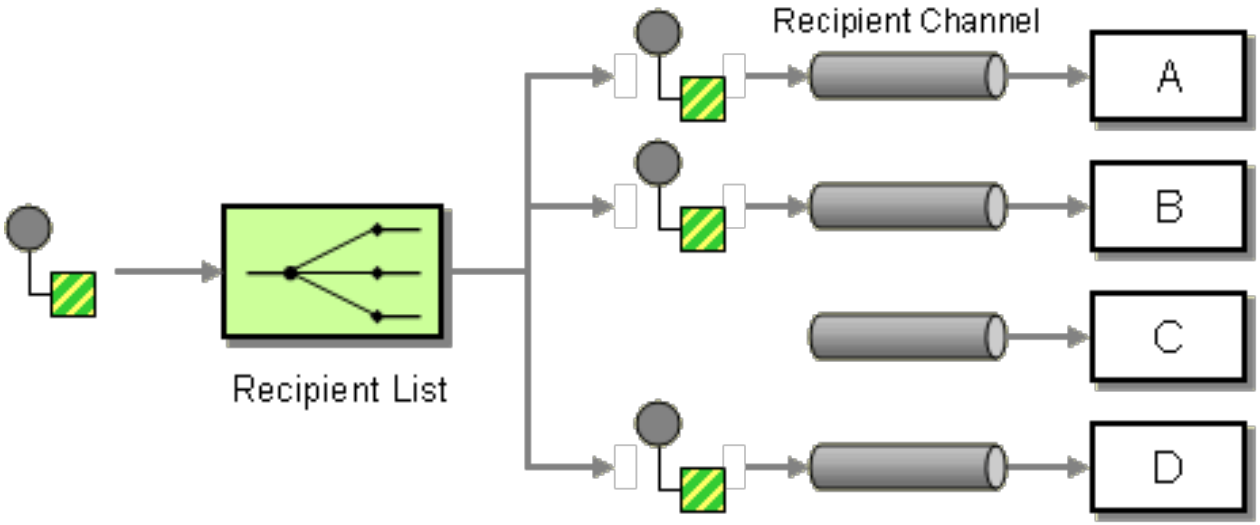
http://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html

Dynamic Router



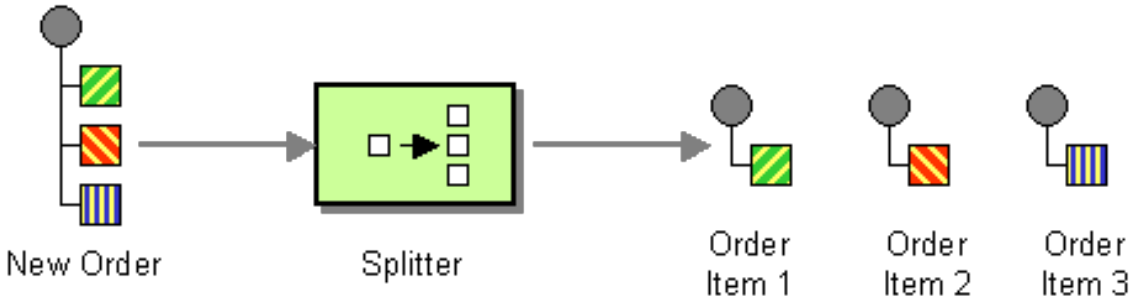
http://www.enterpriseintegrationpatterns.com/patterns/messaging/DynamicRouter.html

Recipient List



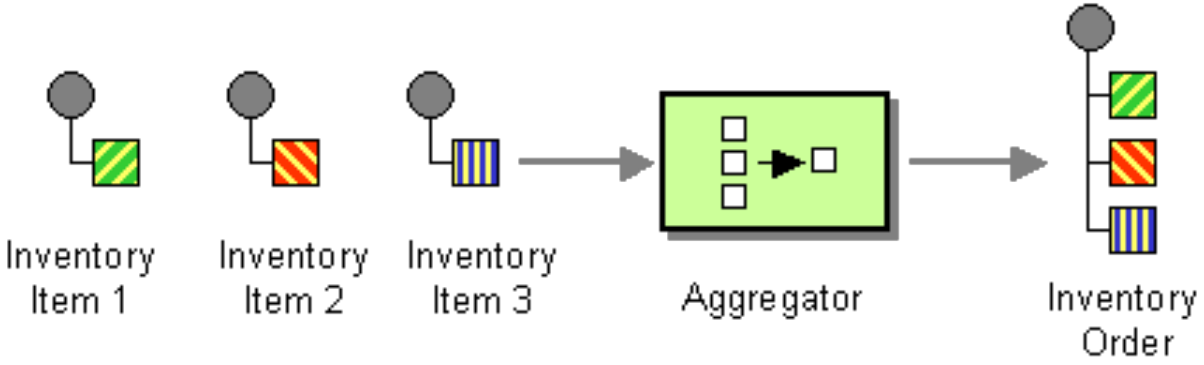
http://www.enterpriseintegrationpatterns.com/patterns/messaging/RecipientList.html

Splitter



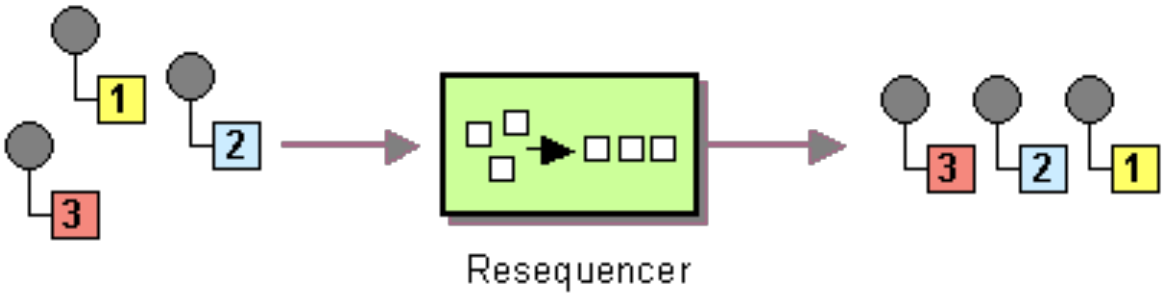
http://www.enterpriseintegrationpatterns.com/patterns/messaging/Sequencer.html

Aggregator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/Aggregator.html

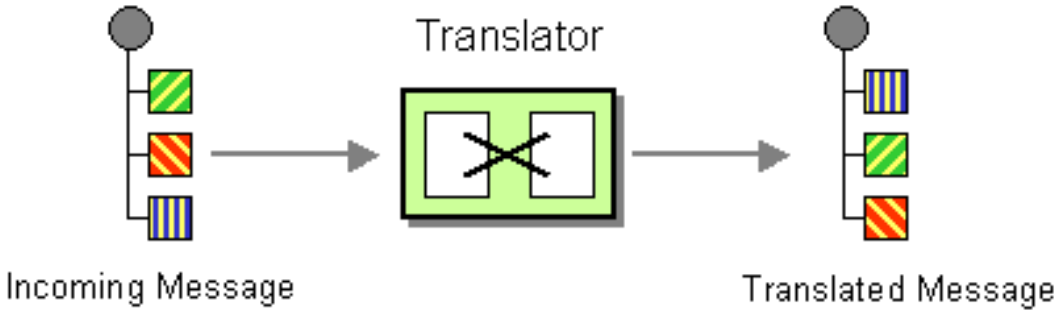
Resequencer



http://www.enterpriseintegrationpatterns.com/patterns/messaging/Resequencer.html

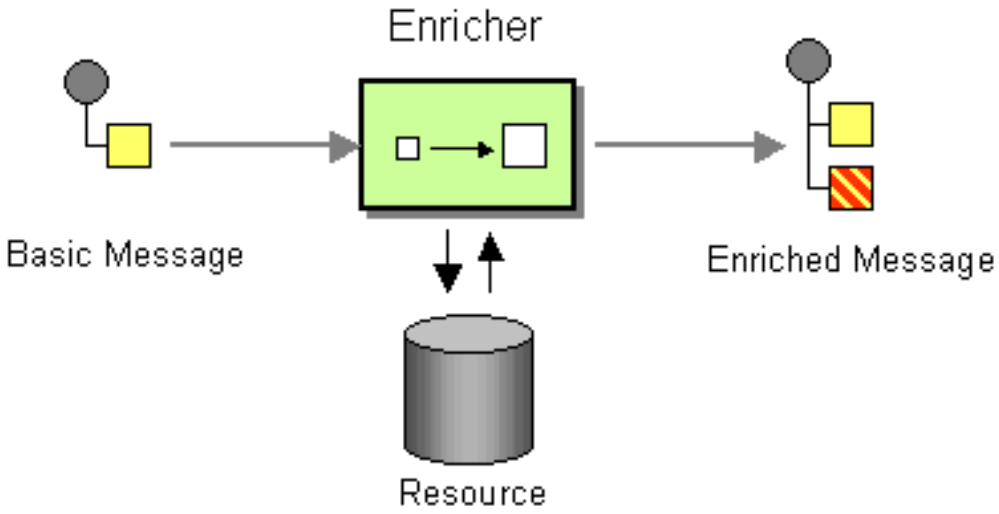
TRANSFORMATION

Message Translator



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html

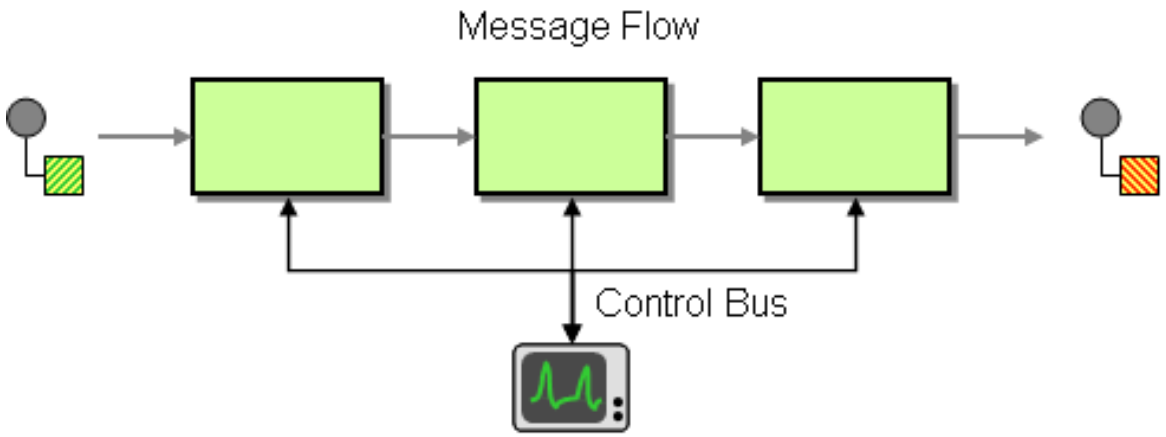
Content Enricher



http://www.enterpriseintegrationpatterns.com/patterns/messaging/DataEnricher.html

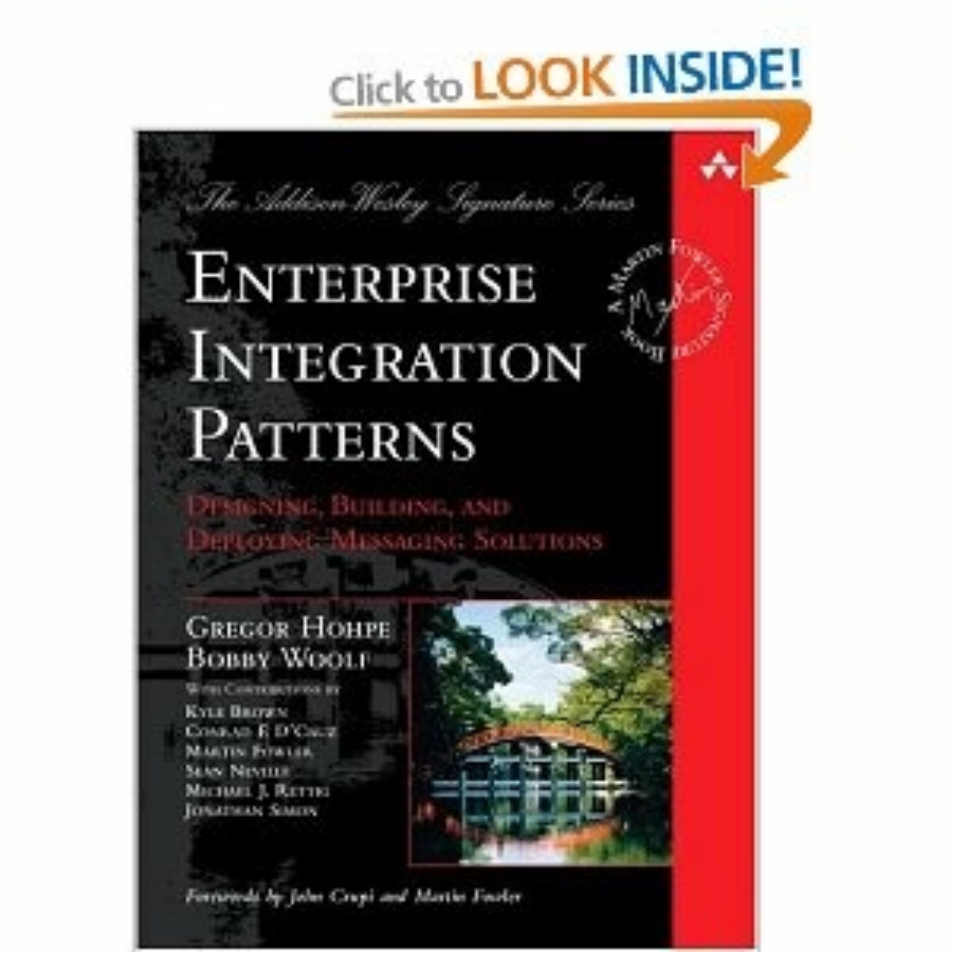
MANAGEMENT

Control Bus



http://www.enterpriseintegrationpatterns.com/patterns/messaging/ControlBus.html

Further Reading



Q&A