FUNDAMENTALS

DESENHO DE SOFTWARE DE LARGA ESCALA

(ULTRA-)LARGE SCALE SYSTEMS

“ … a term used to refer to software intensive systems with **unprecedented amounts** of hardware, lines of source code, numbers of users, and volumes of data.

Q. How much is unprecedented?

Wikipedia

(ULTRA-)LARGE SCALE SYSTEMS

The scale give rise to many problems:

- Developed and used by **many stakeholders** across **multiple organisations**, often with **conflicting** purposes and needs;
- Constructed from **heterogeneous parts** with **complex dependencies** and **emergent properties;**
- **Continuously evolving**; and software, hardware and human **failures are the norm**, not the exception.

(ULTRA-)LARGE SCALE SYSTEMS

- Have **decentralised** data, development, evolution and operational control;
- Address inherently **conflicting**, unknowable, and diverse requirements;
- **Evolve continuously while it is operating**, with different capabilities being deployed and removed;
- Contain **heterogeneous**, inconsistent and **changing** elements;
- Erode the **people-system boundary**: people are not just be users, but elements of the system and affecting its overall **emergent behaviour**;
- Encounter **failure as the norm**, rather than the exception, with it being extremely unlikely that all components are functioning at any one time;
- Require new paradigms for **acquisition** and **policy**, and new methods for **control**.

WHERE CAN WE STUDY THEM?

There are several examples of (U)LSS in today's (2019) software architectures, from **finance** to **advertising**, from **healthcare** to **defense**, but we believe there are two (enabling/pushing/use-your-own-adjective-here™) technologies that are bringing the characteristics of (U)LSS to the **mainstream**:

- **Cloud Computing** (simply referred to as *cloud*), because it actively **prescribes** technologies and techniques engineered to cope with (U)LSS;
- The **Internet-of-Things** (IoT), because it is an emergent (U)LSS due to heterogeneity, failure as the norm, and people-in-the-loop (amongst others).

WHAT IS CLOUD COMPUTING

Cloud computing is an information technology (IT) **paradigm**, a model for enabling ubiquitous access to **shared pools** of **configurable resources** (such as computer networks, servers, storage, applications and services), which can be **rapidly provisioned** with minimal management effort, often **over the Internet**.

Cloud computing relies on sharing of resources to achieve **coherence** and **economy of scale**, similar to a **utility.**

Wikipedia

AWS LIST OF SERVICES

Compute

- **EC2**
  Virtual Servers in the Cloud
- **Lambda**
  Run Code in Response to Events
- **EC2 Container Service**
  Run and Manage Docker Containers

Storage & Content Delivery

- **S3**
  Scalable Storage in the Cloud
- **Storage Gateway**
  Integrates On-Premises IT Environments with Cloud Storage
- **Glacier**
  Archive Storage in the Cloud
- **CloudFront**
  Global Content Delivery Network

Database

- **RDS**
  MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB**
  Predictable and Scalable NoSQL Data Store
- **ElastiCache**
  In-Memory Cache
- **Redshift**
  Managed Petabyte-Scale Data Warehouse Service

Networking

- **VPC**
  Isolated Cloud Resources
- **Direct Connect**
  Dedicated Network Connection to AWS
- **Route 53**
  Scalable DNS and Domain Name Registration

Administration & Security

- **Directory Service**
  Managed Directories in the Cloud
- **Identity & Access Management**
  Access Control and Key Management
- **Trusted Advisor**
  AWS Cloud Optimization Expert
- **CloudTrail**
  User Activity and Change Tracking
- **Config**
  Resource Configurations and Inventory
- **CloudWatch**
  Resource and Application Monitoring

Deployment & Management

- **Elastic Beanstalk**
  AWS Application Container
- **OpsWorks**
  DevOps Application Management Service
- **CloudFormation**
  Templated AWS Resource Creation
- **CodeDeploy**
  Automated Deployments

Analytics

- **EMR**
  Managed Hadoop Framework
- **Kinesis**
  Real-time Processing of Streaming Big Data
- **Data Pipeline**
  Orchestration for Data-Driven Workflows
- **Machine Learning**
  Build Smart Applications Quickly and Easily

Application Services

- **SQS**
  Message Queue Service
- **SWF**
  Workflow Service for Coordinating Application Components
- **AppStream**
  Low Latency Application Streaming
- **Elastic Transcoder**
  Easy-to-use Scalable Media Transcoding
- **SES**
  Email Sending Service
- **CloudSearch**
  Managed Search Service

Mobile Services

- **Cognito**
  User Identity and App Data Synchronization
- **Mobile Analytics**
  Understand App Usage Data at Scale
- **SNS**
  Push Notification Service

Enterprise Applications

- **WorkSpaces**
  Desktops in the Cloud
- **WorkDocs**
  Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW
  Secure Email and Calendaring Service

SOFTWARE AS A SERVICE (SAAS)

- Users simply gain **access to application software** and databases over the Internet;
- Cloud providers **manage the infrastructure** and platforms that run the applications;
- Sometimes referred to as "**on-demand software**";
- Usually priced on a **pay-per-use** basis or using a **subscription** fee.

PLATFORM AS A SERVICE (PAAS)

- Provides a platform **allowing customers to develop**, run, and manage applications without the complexity of building and **maintaining the infrastructure** typically associated with developing and launching an app.

- The consumer controls software deployment with **minimal configuration** options.

- The provider provides the networks, servers, storage, operating system (OS), middleware (e.g. Java runtime, .NET runtime), database and other services to **host the consumer's** application.

INFRASTRUCTURE AS A SERVICE (IAAS)

- Providers offer **computing infrastructure**, such as as virtual machines and other resources, as a service to subscribers;
- **High-level APIs** used to dereference various low-level details of underlying network infrastructure, such as physical computing resources, location, data partitioning, scaling, security, backup…
- **Hypervisor** is responsible for loading virtual machines as guests; **Linux containers** run in isolated partitions of a single kernel;
- Other services: virtual-machine disk-image library, raw block storage, file or object storage, firewalls, load balancers, IP addresses, virtual local area networks (VLANs), software bundles…

FUNCTION AS A SERVICE (FAAS)

- Provides a platform **allowing customers to develop**, run, and manage applications without the complexity of building and **maintaining the infrastructure** typically associated with developing and launching an app.

Wait, isn't this the same thing as PaaS ?!

SERVERLESS (EXECUTION MODEL)

- The provider **dynamically manages** the allocation of **machine resources**. Pricing is based on the actual **amount of resources consumed** by an application, rather than on pre-purchased units of capacity.

- Serverless computing still **requires servers**. The name is used because the server management and capacity planning decisions are completely **hidden** from the developer.

- Serverless code can be used in conjunction with code deployed in traditional styles, such as micro-services. Alternatively, applications can be written to be purely serverless and use no provisioned services at all.

CATEGORIES OF CLOUD COMPUTING

Cloud Clients

Web browser, Mobile App, Thin Client, Terminal Emulator, …

SAAS

CRM, Email, Virtual Desktop, Games, …

PAAS

Serverless

Execution runtime, Database, Web Server, Development Tools, …

IAAS

Virtual Machines, Servers, Storage, Load Balancers, Network, …

MONOLITHS

Traditional enterprise systems are designed as monoliths. Written in a monolithic way, they tend to have **strong coupling** between the components in the service and between services. A system with the **services tangled** and **interdependent** is harder to **write**, **understand**, **test**, **evolve**, **upgrade** and **operate independently**. Strong coupling can also lead to **cascading failures** — where one failing service can **take down the entire system**, instead of allowing you to deal with the failure in **isolation**. These can quickly turn into nightmares that stifle innovation, progress, and joy.

MICROSERVICES

Microservices-Based Architecture is a simple concept: it advocates creating a system from a **collection of small**, **isolated** services, each of which **owns their data**, and is independently isolated, **scalable** and **resilient to failure**.

Services **integrate** with other services in order to form a **cohesive system** that's far more flexible than the typical enterprise monolith systems.

These principles are based on 1998's SOAs. Technical constraints held us back: single machines/single core processors, slow networks, expensive disks, expensive RAM, and **organisations structured as monoliths** (Conway's law).

CONWAY'S LAW

" … organisations which design **systems** are constrained to produce **designs** which **are copies** of the **communication structures** of these organisations.

Wikipedia

CONWAY'S LAW

> … organisations which design **systems** are constrained to produce **designs** which **are copies** of the **communication structures** of these organisations.

Wikipedia

WHAT IS SOA?

… is a style of software design where services are provided to the **other components** by application **components**, through a **communication protocol** over a network. The basic principles of service-oriented architecture are **independent of vendors, products and technologies**. A service is a **discrete unit of functionality** that can be **accessed remotely** and acted upon and updated **independently**, such as retrieving a credit card statement online.

Wikipedia

PROPERTIES OF A SERVICE

1. You shall represent a **business activity**;
2. You shall be **self-contained**;
3. You shall be a **black-box** for your consumers;
4. You may be a **composite** of other services.

SOA MANIFESTO

Things on the left are given **more importance** than things on the right (they are still important, though):

Business Value	Technical Strategy
Strategic Goals	Project-specific Benefits
Intrinsic Inter-Operability	Custom Integration
Shared Services	Specific-Purpose Implementations
Flexibility	Optimization
Evolutionary Refinement	Initial Perfection

DID THE 70'S CALLED?

What exactly is the difference between SOA and Modular Programming from the seventies?

> It doesn't prescribe how to **modularize** an application, but instead focus on how to **compose** an application by integration of distributed, separately-maintained and deployed software components.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services adhere to a standard communications agreements, as defined collectively by one or more service-description documents within a given set of services

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

The relationship between services is minimised to the level that they are only aware of their existence, and can be called from anywhere within the network that it is located no matter where it is present.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Many services which were not initially planned under SOA, may get encapsulated or become a part of SOA.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services should be designed to be long lived. Where possible, avoid forcing consumers to change if they do not require new features, if you call a service today you should be able to call it tomorrow.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

The services act as black boxes, that is their inner logic is hidden from the consumers.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services are independent and control the functionality they encapsulate, both from a design-time and a run-time perspective.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Each request to the service must contain all the information necessary to understand the request, and cannot take advantage of any stored context, as it may come from anywhere, anytime.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

A principle to ensure services have an adequate size and scope. The functionality provided by the service to the user must be relevant.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services are decomposed or consolidated (normalised) to minimize redundancy. Sometimes, performance optimisation, access, and aggregation mandates otherwise.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services can be used to compose other services.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.

PRINCIPLES AND ATTRIBUTES

Standardised Contract

Encapsulation

Abstraction

Statelessness

Normalization

Discovery

Loosely Coupled

Longevity

Autonomy

Granularity

Composability

Reusability

Logic is divided into various services, to promote reuse of code.

EXAMPLE (BEFORE SOA)

Application Dependent Business Functions

Service Scheduling

Check Customer Status

Determine Product Availability

Order Processing

Check Customer Status

Determine Product Availability

Verify Customer Credit

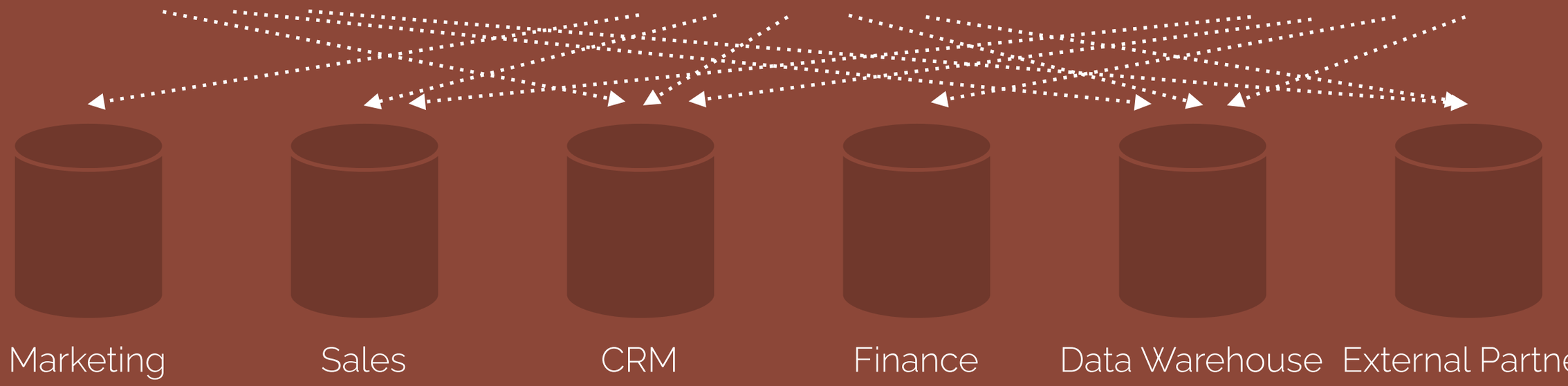
Order Status

Account Management

Order Status

Check Credit

Calculate Shipping Charges



EXAMPLE (AFTER SOA)

Composite Applications

Composite Application	Order Processing	Account Management	Service Scheduling
Composed Business Process			

Create Invoice | Check Customer Status | Check Order Status | Check Credit | Check Inventory

Marketing | Sales | CRM | Finance | Data Warehouse | External Partner

REACTIVE SYSTEMS

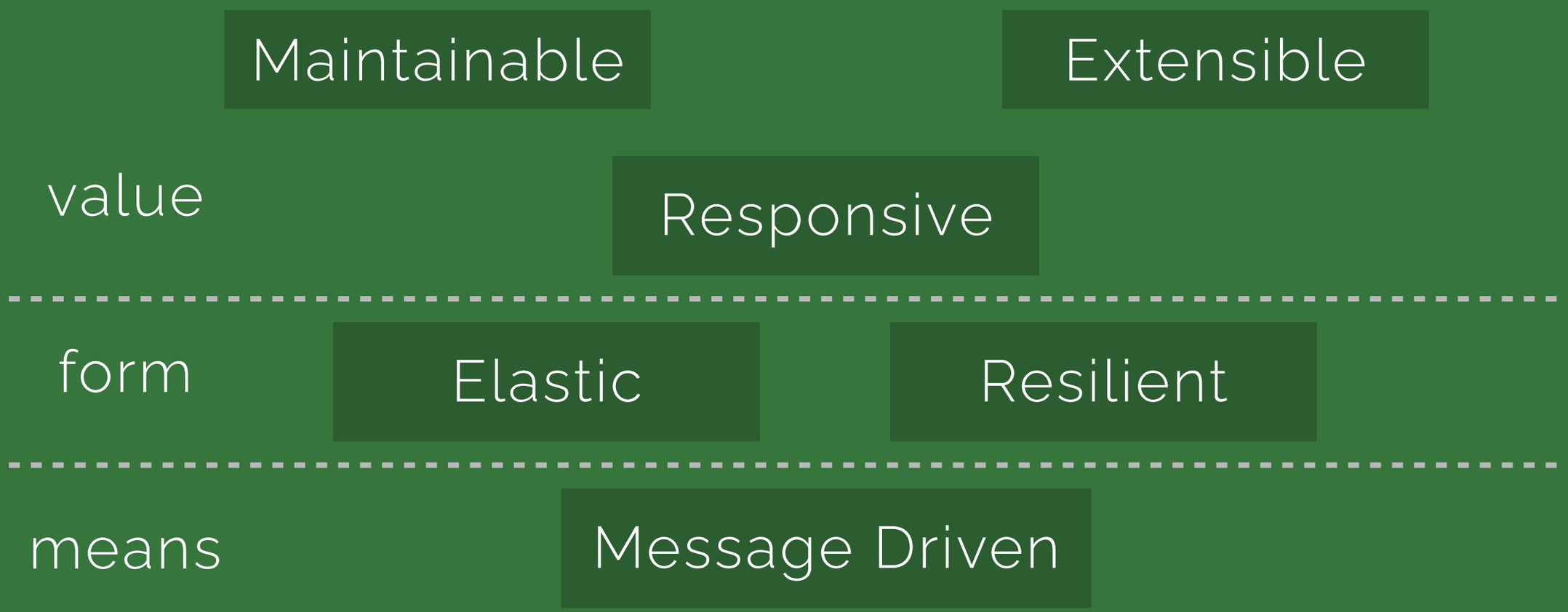
Application requirements have changed dramatically:

- 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.

REACTIVE SYSTEMS

… an answer to the challenge of having systems **easier to develop** and **amenable to change** while making them more **tolerant of failure** and when failure does occur they meet it with **elegance rather than disaster**.



REACTIVE SYSTEMS

- **Responsive.** The system responds in a **timely** manner if at all possible. Responsive systems focus on providing **rapid** and **consistent response times**, establishing **reliable upper bounds** so they deliver a consistent **quality of service** (QoS).

- **Resilient.** The system stays **responsive in the face of failure.** Resilience is achieved by **replication**, **containment**, **isolation** and **delegation**.

REACTIVE SYSTEMS

- **Elastic.** Stay responsive under **varying workload**. The system react to changes in the input rate by increasing or decreasing the **resources allocated**. There is no **contention points** or central **bottlenecks**, resulting in the ability to **shard** or **replicate** components (predictive/reactive scaling).

- **Message Driven.** Reactive Systems rely on **asynchronous message-passing** to establish a boundary between components that ensures **loose coupling**, **isolation** and **location transparency**. 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.

ROLES

Provider

Broker

Consumer

Creates a service and provides its information to the service registry. Each provider ponder on which service to expose, whom to give more importance: security or easy availability, what price to offer the service for, and many more. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.

ROLES

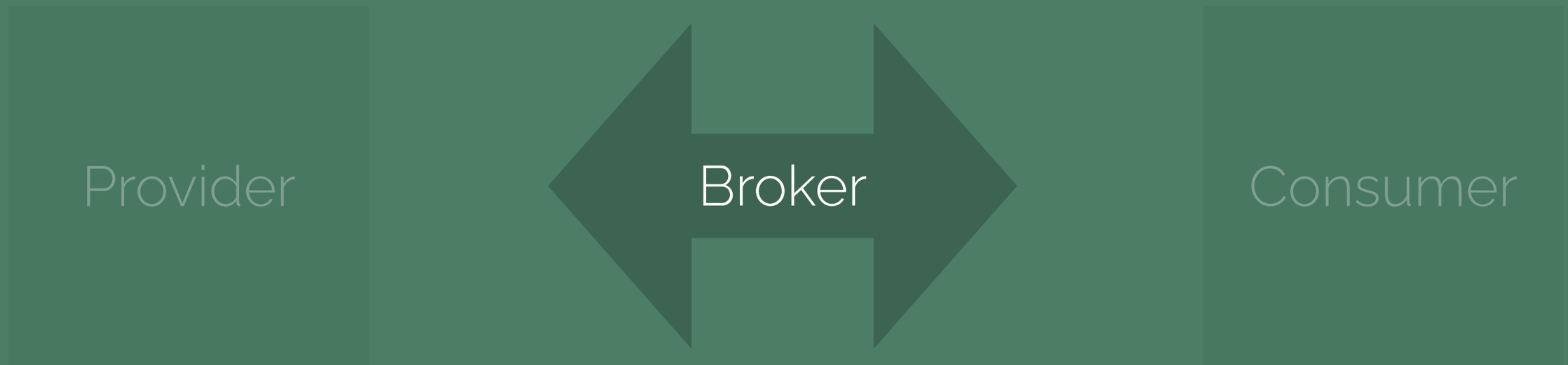
Provider

Broker

Consumer

It locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its web services. Whichever service the service-consumers need, they have to take it into the brokers, bind it with respective service and then use it. They can access multiple services if the service provides multiple services.

ROLES



Its main functionality is to make the information regarding the web service available to any potential requester. Whoever implements the broker decides the scope of the broker. There are public and private brokers. The broker also establishes the communication medium; for example, a Message Broker such as RabbitMQ imposes both a protocol (AMQP) and a style (PubSub).

WHAT IS INTERNET-OF-THINGS

- An infrastructure of **interconnected** objects, people, systems and information resources together with **intelligent services** to allow them to process information about the **physical and the virtual world** and react;

- By the International Organisation for Standardisation (ISO) and the International Electrotechnical Commission (IEC);

- Utopia of **Ubiquitous Computing** and **Ubiquitous Connectivity**.

IOT AS A (U)LSS

- Have a high degree of **heterogeneity** of devices and/or standards;
- Result from the combination of other, new or already existent, systems (**systems of systems**);
- The **number** of devices, applications, users and amounts of data that are part of the system allows it to be considered of large-scale (requiring **orchestration**);
- The **volatility** of the devices that are connected to the system at a given point allows the system to be considered **highly-dynamic** in terms of topology;
- The IoT utopia depends on the **interoperability** among IoT solutions.

INTERNET-OF-THINGS

Air Pollution
Control of CO_2 emissions of factories, pollution emitted by cars and toxic gases generated in farms.

Forest Fire Detection
Monitoring of combustion gases and preemptive fire conditions to define alert zones.

Wine Quality Enhancing
Monitoring soil moisture and trunk diameter in vineyards to control the amount of sugar in grapes and grapevine health.

Offspring Care
Control of growing conditions of the offspring in animal farms to ensure its survival and health.

Sportsmen Care
Vital signs monitoring in high performance centers and fields.

Structural Health
Monitoring of vibrations and material conditions in buildings, bridges and historical monuments.

Smartphones Detection
Detect iPhone and Android devices and in general any device which works with Wifi or Bluetooth interfaces.

Perimeter Access Control
Access control to restricted areas and detection of people in non-authorized areas.

Radiation Levels
Distributed measurement of radiation levels in nuclear power stations surroundings to generate leakage alerts.

Electromagnetic Levels
Measurement of the energy radiated by cell stations and WiFi routers.

Traffic Congestion
Monitoring of vehicles and pedestrian affluence to optimize driving and walking routes.

Smart Roads
Warning messages and diversions according to climate conditions and unexpected events like accidents or traffic jams.

Smart Lighting
Intelligent and weather adaptive lighting in street lights.

Intelligent Shopping
Getting advices in the point of sale according to customer habits, preferences, presence of allergic components for them or expiring dates.

Noise Urban Maps
Sound monitoring in bar areas and centric zones in real time.

Water Leakages
Detection of liquid presence outside tanks and pressure variations along pipes.

Vehicle Auto-diagnosis
Information collection from CanBus to send real time alarms to emergencies or provide advice to drivers.

Item Location
Search of individual items in big surfaces like warehouses or harbours.

Waste Management
Detection of rubbish levels in containers to optimize the trash collection routes.

Smart Parking
Monitoring of parking spaces availability in the city.

Water Quality
Study of water suitability in rivers and the sea for fauna and eligibility for drinkable use.

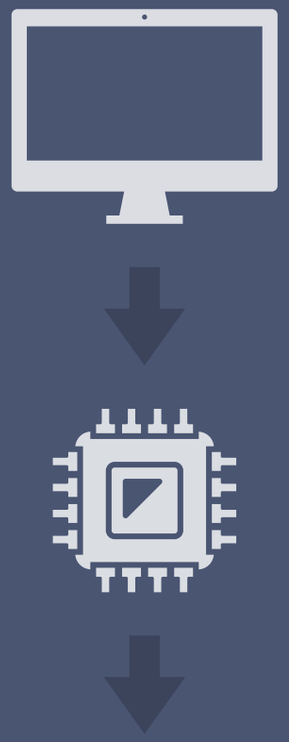
Golf Courses
Selective irrigation in dry zones to reduce the water resources required in the green.

Quality of Shipment Conditions
Monitoring of vibrations, strokes, container openings or cold chain maintenance for insurance purposes.

libelium
www.libelium.com

> *... purchasing 10-20 different services from 10-20 different vendors using 10-20 different apps with 10-20 different user interfaces. If that's the way IoT goes, it will be a long tough slog to Nirvana.*
>
> — Bob Harden

BACK TO BASICS



This is a **computer**! (yes, it's actually a monitor, Sherlock… bear with the metaphor)

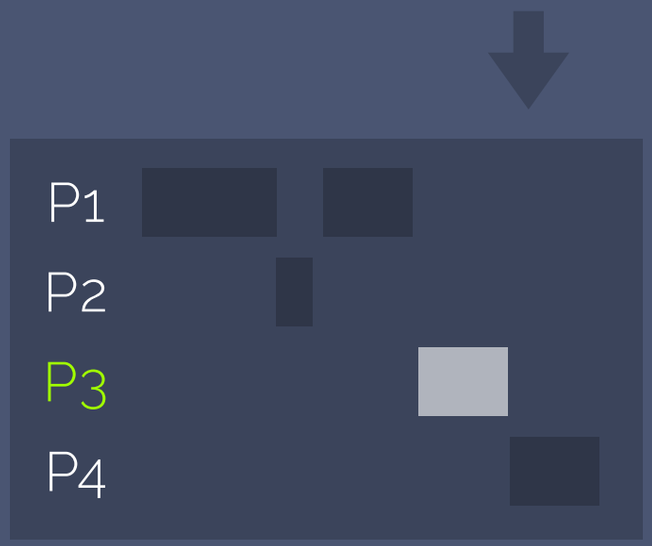
Inside you have a **Central Processing Unit** (CPU)

```
  0×4113C0  push ebp
  0×4114C1  mov  ebb, esp
> 0×4113C3  sub  esp, 0×C0
  0×4113C9  push ebx
  0×4113CA  push esi
  0×4113CB  push edi
```

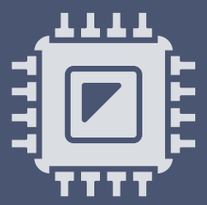
CPUs execute instructions in a **sequential manner** (one at a time). But when you use a modern Operating System (OS), several things happen **simultaneously**. How is this achieved?

BACK TO BASICS

It is the job of the OS to provide a way of **executing** (hopefully) simultaneous jobs (or processes) in a **timely manner** (which means respecting priorities).

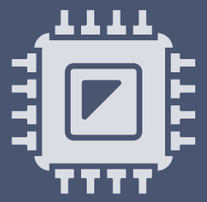


In the good-old-days™, when processes competed for a single resource (the CPU). the OS would allocate **small slices of time** for them to execute (**time-sharing**), **suspending all others**. Do this fast enough and they appear to be executing simultaneously, albeit they aren't.



But then, processors started hitting a **limit** on how fast they could (sequentially) execute. Not to mention the **overhead** of context switching (try to do 5 things at once, and you'll feel firsthand the meaning of *overhead*).

BACK TO BASICS



So if a single processor can't go any faster, how can we make a system (as a whole)… faster?

Trivial, uh? We just use **more machines**.

It makes sense, doesn't it? If we want to build an house faster, we use **more** builders! Plant a field faster? **More** workers! Clean an hospital faster? **More** cleaners! Fly faster? **More** pilots! Deliver a baby in 1 month? 9 pregnant women! Wait… something's not right…

TWO THINGS PEOPLE FORGET

(AMONG OTHER THINGS I AM FORGETTING)

(1) 

Things take time to get from point A to point B. Besides, nothing can go faster than the **speed of light** (300 000 kms/s). So the further two processing units are apart, the longer it takes for them to communicate;

(2) 

And even if you are able to **communicate efficiently**, you need a **strategy** to break down a single task into multiple smaller tasks that can be **distributed** across processing units.

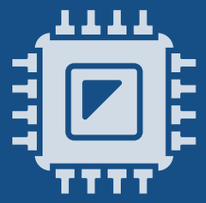
Cooperation needs both a strategy and communication channels!

LATENCY COMPARISON

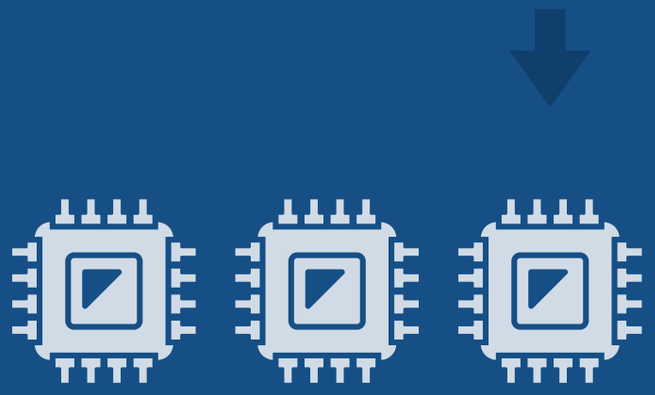
L1 cache reference	0,5	ns
Branch mispredict	5,0	ns
L2 cache reference	7,0	ns
Mutex lock/unlock	25,0	ns
Main memory reference	100,0	ns
Compress 1K bytes with Zippy	3 000,0	ns
Send 1K bytes over 1 Gbps network	10 000,0	ns
Read 4K randomly from SSD	150 000,0	ns
Read 1 MB sequentially from memory	250 000,0	ns
Round trip within same datacenter	500 000,0	ns
Read 1 MB sequentially from SSD	1 000 000,0	ns
Disk seek	10 000 000,0	ns
Read 1 MB sequentially from disk	20 000 000,0	ns
Send packet CA > Netherlands > CA	150 000 000,0	ns

https://gist.github.com/jboner/2841832

MULTIPLE CORES



To solve problem (1), we can create faster networks between computers. But its just more efficient to…



… have multiple CPUs in a single machine; or even better, multiple cores in a single CPU die. (it's also way more comfortable to **avoid** carrying multiple personal interconnected laptops)

Nowadays multiple cores are so common, that even mobile phones, such as the latest Apple A14 CPU, have: 2 high-performance cores, 4 energy-efficient cores, 16 Neural cores, 5 GPU cores, and some bells and whistles concerning dedicated silicon for DSP.

CONCURRENT & PARALLEL

But this approach only solves one side of the problem (the hardware side). What we actually want is something along the lines:

- I want my email application to run on the background, and receive notifications in the foreground while I am playing a game;

- I want to hear sound coming out of my speakers as I watch a video in Youtube, and I want that sound to be synchronised;

- I want to play World of Warcraft with 1M people in the same map, and I don't want to be killed by someone if I kill him first;

- I want Photoshop to be as fast as possible when applying that cool old-school filter to my selfie.

Some of these tasks are inherently parallel (easy breakdown and few dependencies), while others require high-levels of synchronisation and data transfer just to keep consensus.

CONCURRENCY

(TO BE CONTRASTED WITH PARALLELISM)

… is the **decomposability** property of a program, algorithm, or problem into order-independent or partially-ordered components or units. This means that even if the concurrent units of the program, algorithm, or problem are executed **out-of-order** or in partial order, the final outcome will remain the same. This allows for **parallel execution** of the concurrent units, which can significantly **improve overall speed**.

Wikipedia

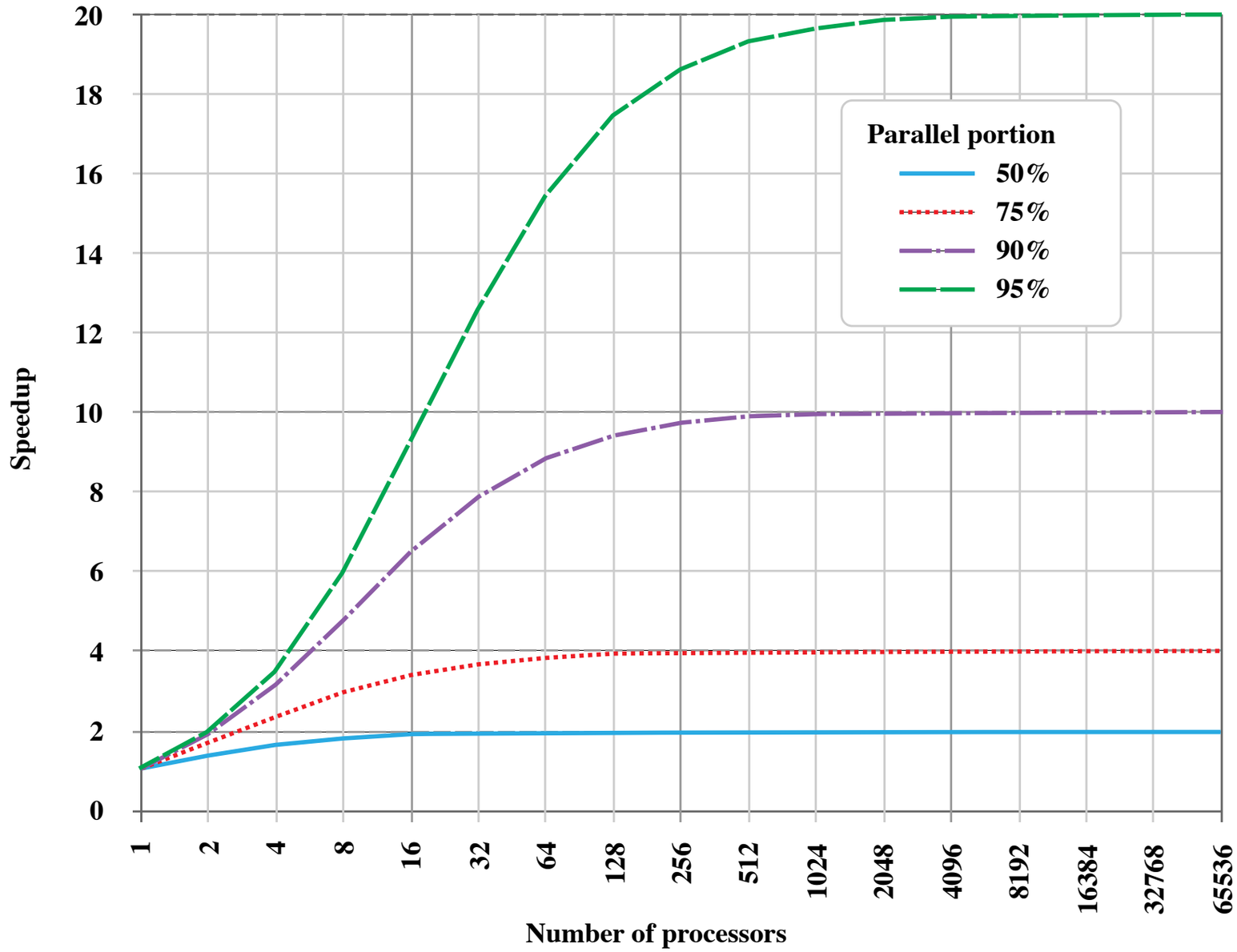
PARALLELISM

(TO BE CONTRASTED WITH CONCURRENCY)

… is a type of computation in which many calculations or the execution of processes are carried out **simultaneously**. Large problems can often be **divided into smaller ones**, which can then be solved at the same time.

Wikipedia

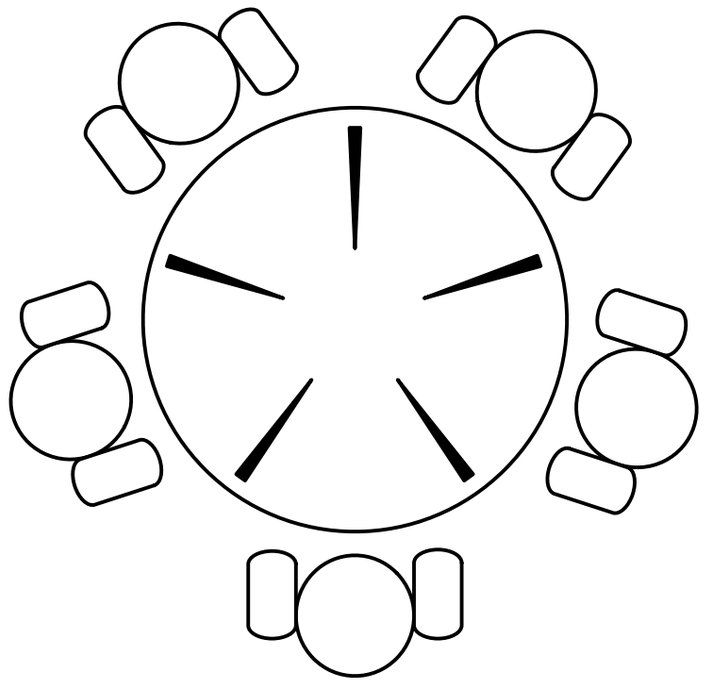
AMDAHL'S LAW



The speedup of a program is limited by its serial part.

DINING PHILOSOPHERS

Imagine that five philosophers are sitting around a table, with five (not ten) chopsticks arranged just like this:



A philosopher is either thinking or hungry. If he's hungry, he picks up the chopsticks on either side of him and eats for a while. When he's done, he puts them down.

DINING PHILOSOPHERS

(CAN'T EACH HAVE THEIR OWN CUTLERY?)

This example allows us to remember some concepts:

- Philosopher: a **process** or **thread**;
- Chopsticks: **shared resources**, where:
- Attempting to pick up: **acquiring** a resource;
- Using it: resource **locked**;
- Laying down the chopstick: **releasing** a resource.

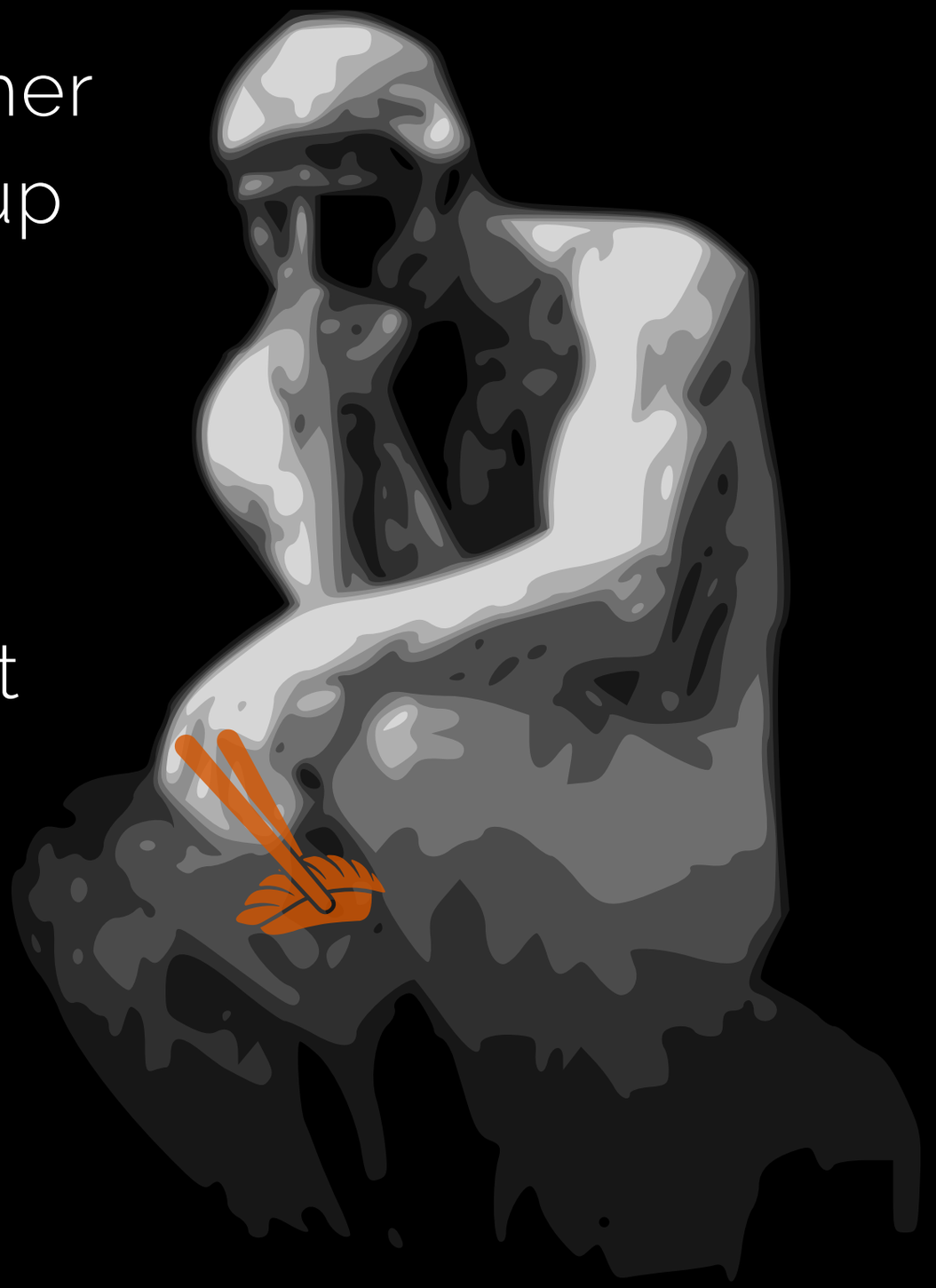
EXAMPLE (PSEUDO-)CODE

```
class Philosopher extends Thread {
  private Chopstick left, right;
  private Random random;

  public Philosopher(Chopstick left, Chopstick right) {
    this.left = left, this.right = right;
    random = new Random();
  }

  public void run() {
    while(true) {
      Thread.sleep(random.nextInt(1000));
      synchronized(left) {
        synchronized(right) {
          Thread.sleep(random.nextInt(1000));
        }
      }
```

RACING CONDITIONS

“ What happens if every philosopher decides to simultaneously pick up the left chopstick?

A **deadlock!** All threads will wait forever trying to acquire the right chopstick, where there’s none left.



CAN WE FIX IT?

Sort of...

We basically need to come up with a **strategy** to avoid the **deadlock**. A naïve approach would try to acquire the right chopstick within a **specified time**. If the time elapses without acquiring that resource, the philosopher would **release the left chopstick** and wait some time before **repeating the attempt**.

The timeout is a **very important** pattern in distributed systems!

IS IT GOOD?

Sort of…

Concurrency is managed **explicitly**, where the developer needs to "foresee" racing conditions and appropriately manage them: **not trivial!**

- If you lock too early, you hinder **parallelism**;
- If you lock too late, you hinder **correctness**.

SO WHERE'S THE PROBLEM?

Let's think about this. To achieve parallelism you use?

" Threads!

What do you need to use w/ threads?

" Locks!

What are they for?

" They prevent multiple threads from running in **parallel**!

WHAT'S THE REAL PROBLEM?

Why can't all threads run in parallel? Picture this: two person can't be **using simultaneously** the same screwdriver, but they can **watch simultaneously** the same movie. Using a screwdriver requires a lock. Watching a movie doesn't. Got the gist?

Mutability!

THE ACTOR MODEL

(ALSO KNOWN AS OBJECT-ORIENTED PROGRAMMING DONE RIGHT)

> “ I’m sorry that I coined the term **objects**, because it gets many people to focus on the lesser idea. The big idea is **messaging**.
> — Alan Kay

An actor is a computational entity that, in response to a message it receives, can concurrently:

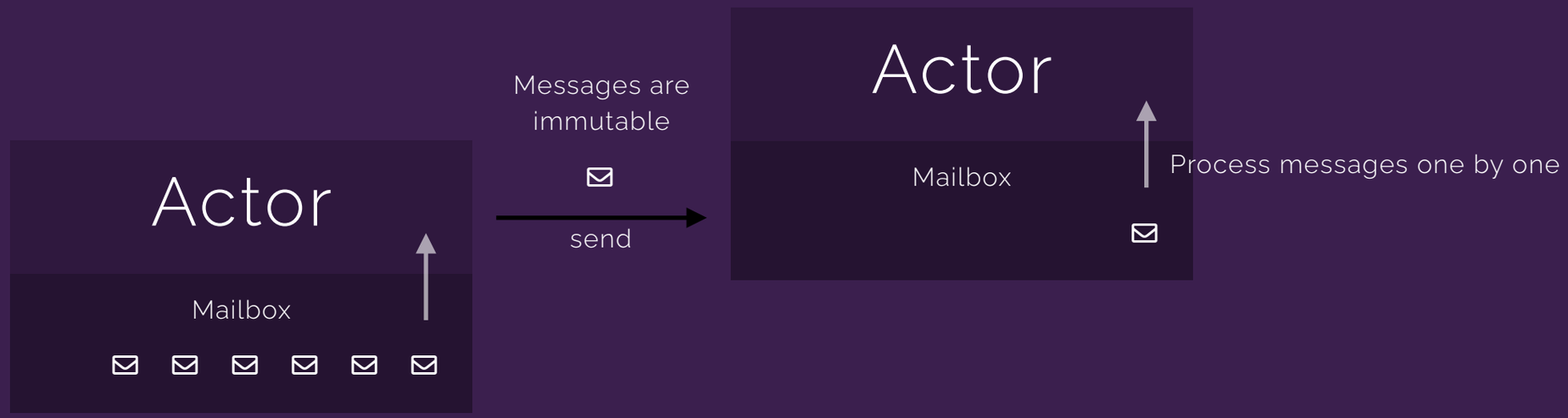
- **Send** a finite number of **messages** to other actors;
- **Create** a finite number of new actors;
- **Designate the behaviour** to be used for the next message it receives.

SHARE NO STATE!

(SO WHERE'S MY STATE?)

- An actor can **maintain a private state**;
- "Designating what to do with the next message" basically means defining how this **state will look like** for the **next message it receives**. Or, more clearly, it's how actors mutate state;
- An Actor that behaves like a calculator, has its **initial state as the number 0**. When it receives an add(1) message, instead of mutating its original state, it designates that **for the next message it receives, the state will be 1**.

THE ACTOR MODEL



Recipients of messages are identified by **address**.

LET IT FAIL!

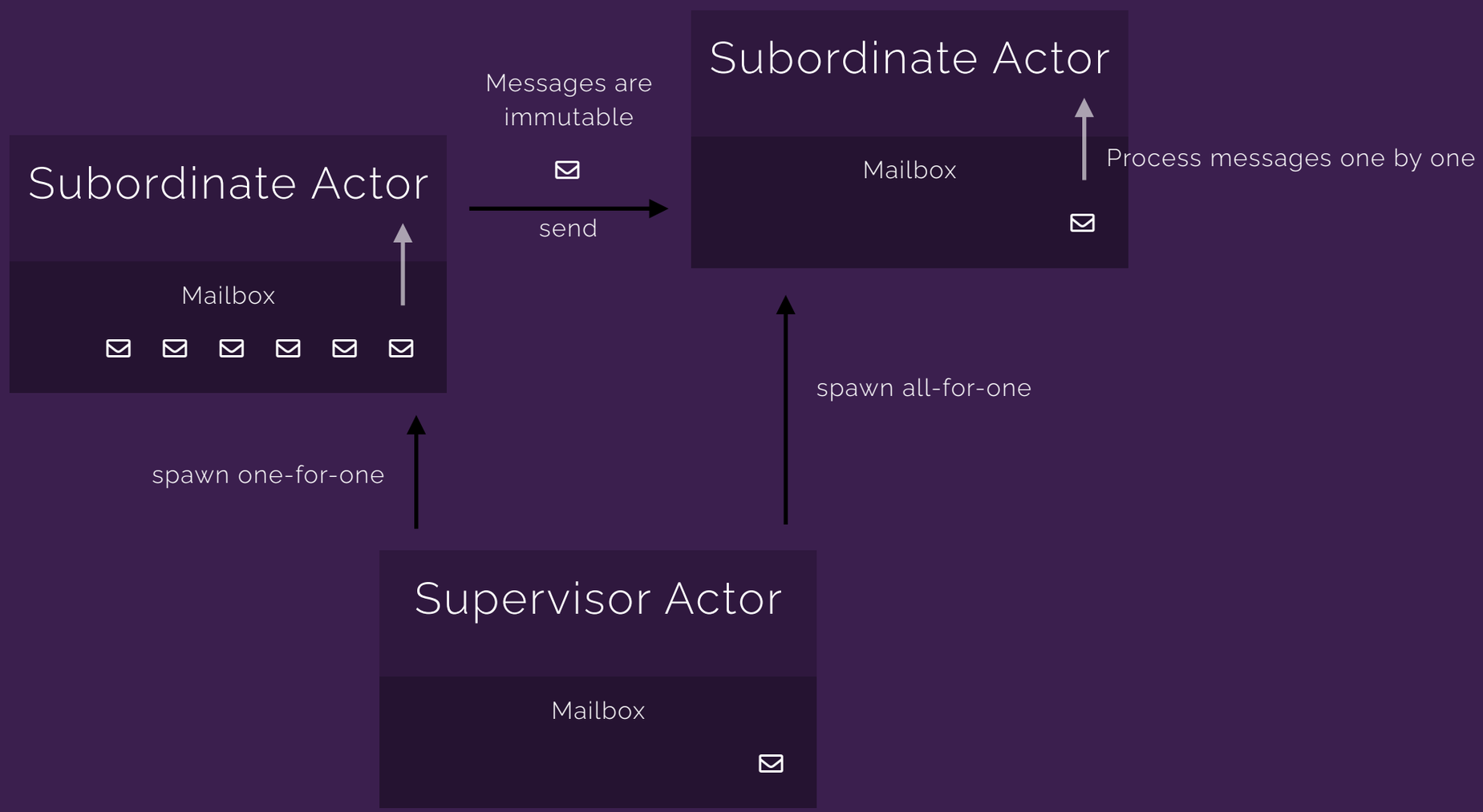
(WHAT'S WRONG IN TRYING TO PREVENT ERRORS?)

Throwing an exception in concurrent code just simply **blow up the thread** that currently executes the code that threw it:

- There is no way to **find out what went wrong**, apart from inspecting the stack trace;
- There is nothing you can do to **recover from the problem** and **bring back your system** to a normal functioning.

Supervised actors provide a clean error recovery strategy encouraging **non-defensive programming**.

ACTOR SUPERVISION



Actors spawn other actors, and are responsible for their failures.

EXAMPLE (PSEUDO-)CODE

```
class Blackboard extends UntypedActors {
  int sum = 0;
  public void onReceive(Object message) {
    if (message instance Integer)
      sum += (Integer) message;
  }
}

class Attendant {
  int age;
  Blackboard blackboard;

  public void sendAge() {
    blackboard.tell(age);
```

IMPLEMENTATIONS

Some languages were made with **actors in mind**:

- Erlang;
- Elixir;
- Pony.

More common nowadays is to use a **framework** for existing languages. Popular frameworks include:

- Akka (Java and Scala);
- Celluloid (Ruby);
- Protoactor (Go, C#, Python, Javascript and Java).

SECONDARY RANT

Actors remember a pattern we've seen before… What is it?

“ Message-Oriented Architectures!

THE ACTOR MODEL

Pros

- Immutability **prevents non-determinism**;
- Declarative programming style improves **readability**;
- Parallelising functional (side-effect free) code can be **embarrassingly easy**;
- Better **confidence** on your program **correctness**;
- Good support for **distributed computation** (location transparency).

Cons

- Functional thinking can be "**unfamiliar**" to OO developers;
- May be **less efficient** than its imperative counter-part;
- Immutability stresses the **Garbage Collector** to its limits;
- **Less control**: computational tasks are split and scheduled by the framework;
- Great abstraction for parallelism, but not for concurrency.

THE THREAD MODEL

Pros

- **Close to the metal**: very efficient if done right;
- **Low abstraction level**: widest range of applicability and control;
- Threads and locks can be implemented in **existing languages** with minimal effort.

Cons

- **Low abstraction level**: programming is hard and memory management is even harder;
- Inter-thread communication via shared mutable state leads to **non-determinism**;
- Threads are a **scarce resource**;
- It only works with shared memory architectures (doesn't work in **distributed systems**);
- Writing is difficult; **testing is nearly impossible**!

BUT WAIT, THERE'S MOAR!

The actor model is just but one technique to cope with parallelism and concurrency. You should also be aware of the following concepts:

- Functional Programming;
- Reactive (Functional) Programming;
- Lambda Architectures / MapReduce;
- Futures;
- Software Transactional Memory; and many more…

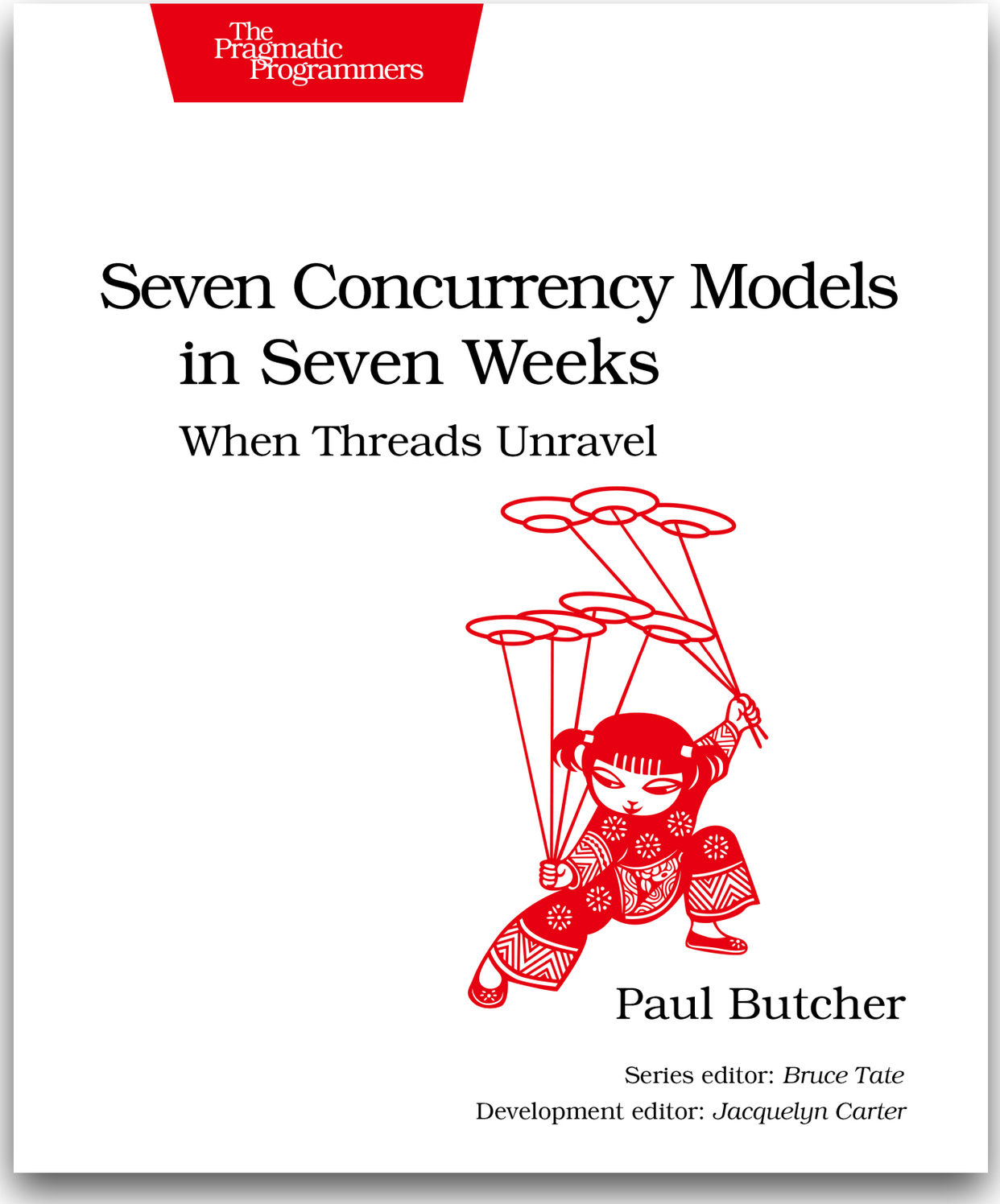
We will cover some of these concepts in later classes.

PRELIMINARY RANT

No matter how many concurrency models you learn, one thing will become evident with time, to wit:

“ **Shared Mutable State** is the Root of All Evil.

PRAGMATIC BIBLIOGRAPHY

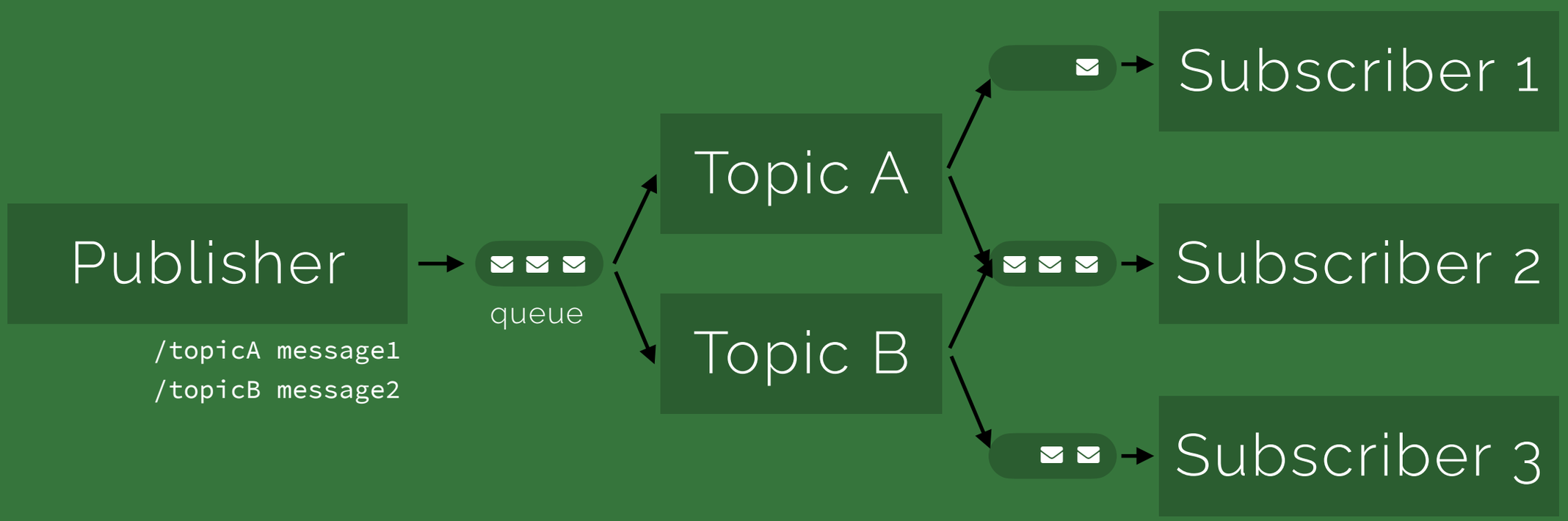


DSLE

{hugo.sereno}@fe.up.pt

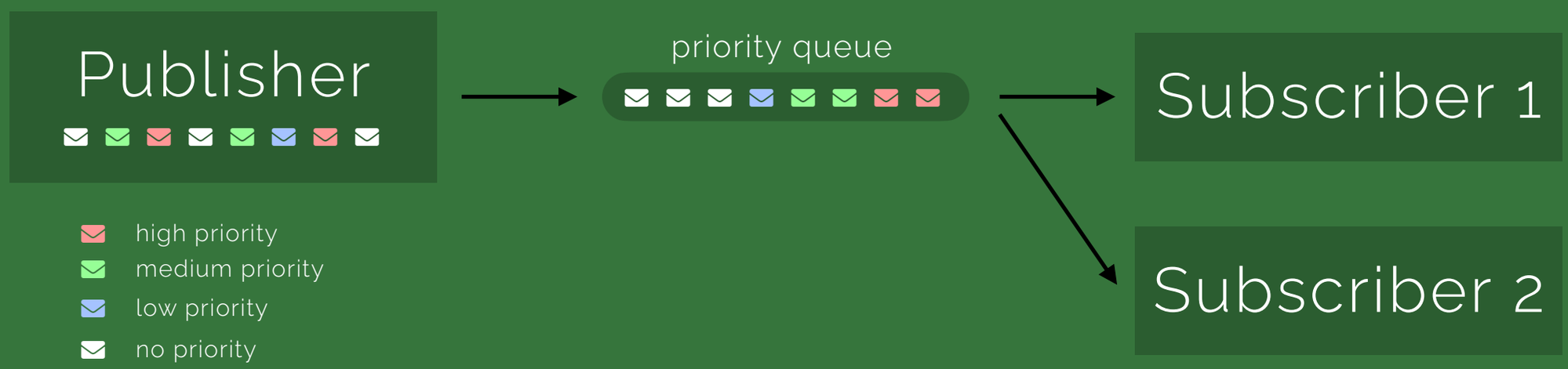
PUBLISH-SUBSCRIBE

A messaging pattern where senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers**, but instead characterise published messages into **classes** (or **topics**) without knowledge of which subscribers, if any, there may be.



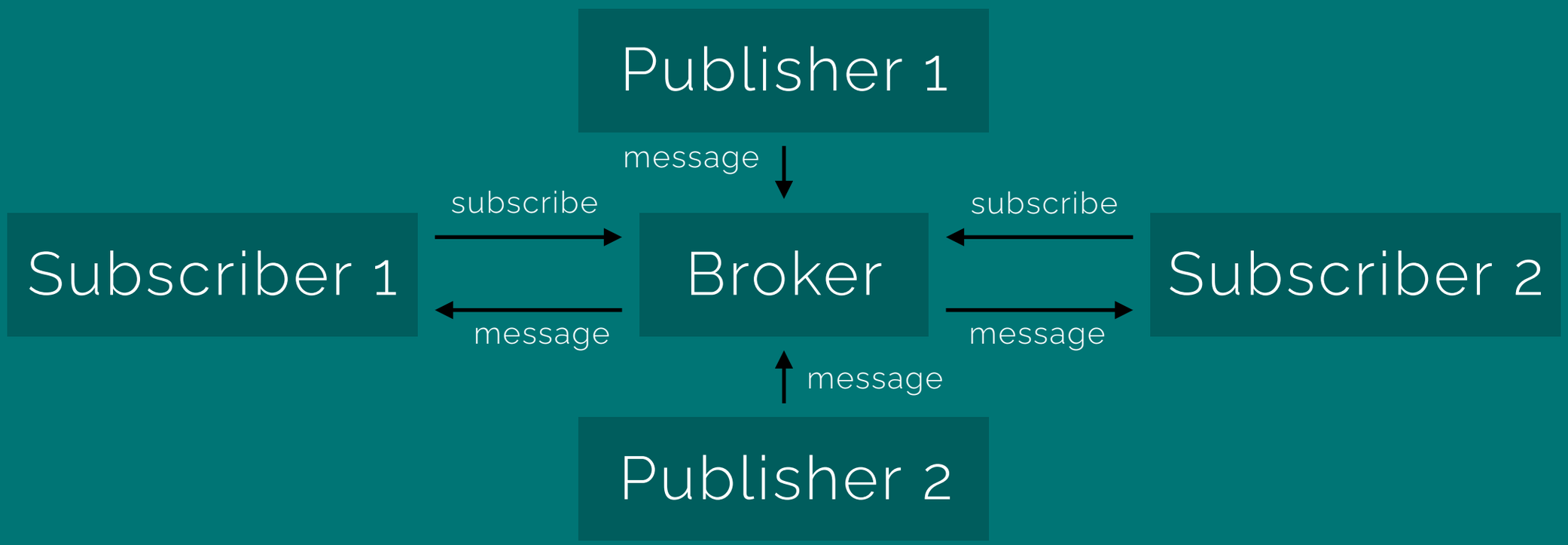
PRIORITY QUEUE

Like a regular queue, but where each element has a **priority associated** with it. In a priority queue, an element with high priority is **served before** (or with an **higher probability**) an element with low priority. Otherwise, they are served according to their order in the queue.



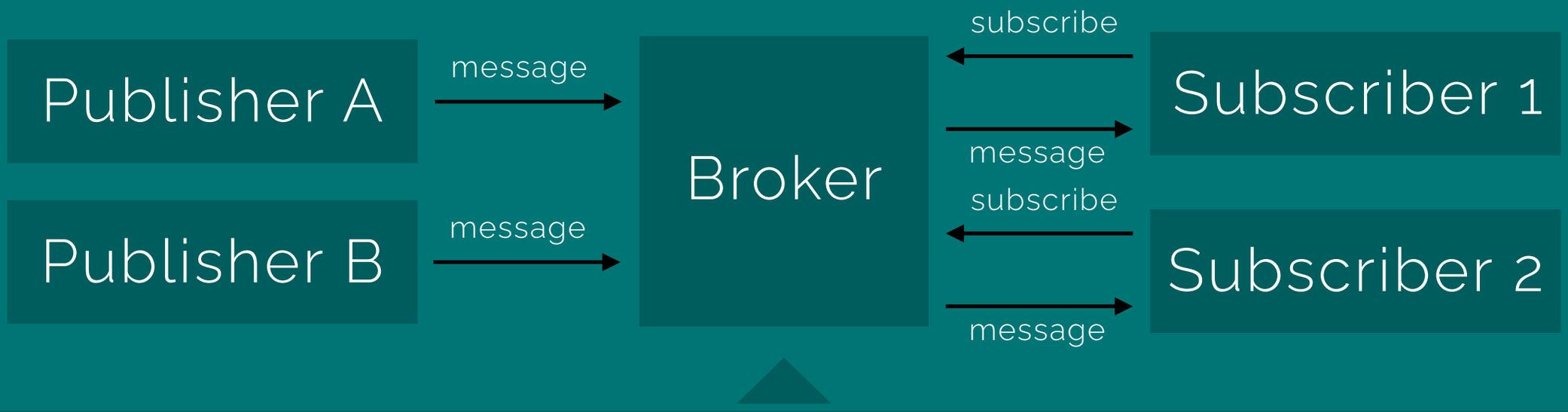
(MESSAGE) BROKER

Mediates communication amongst applications, minimising the **mutual awareness** that applications should have of each other in order to be able to exchange messages, effectively implementing **decoupling**.



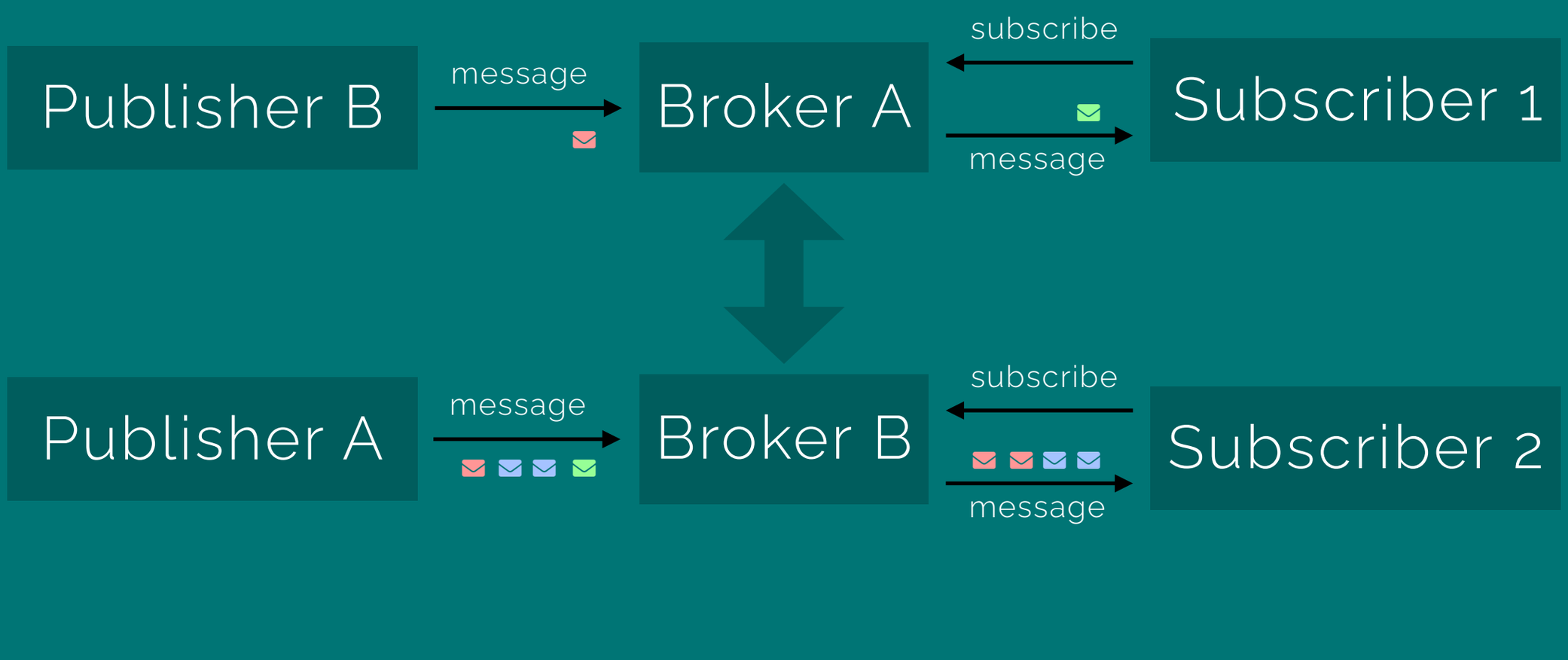
CONTENTION POINT

Using a single broker may lead to a contention point, that hinders both **performance** and **reliability**.



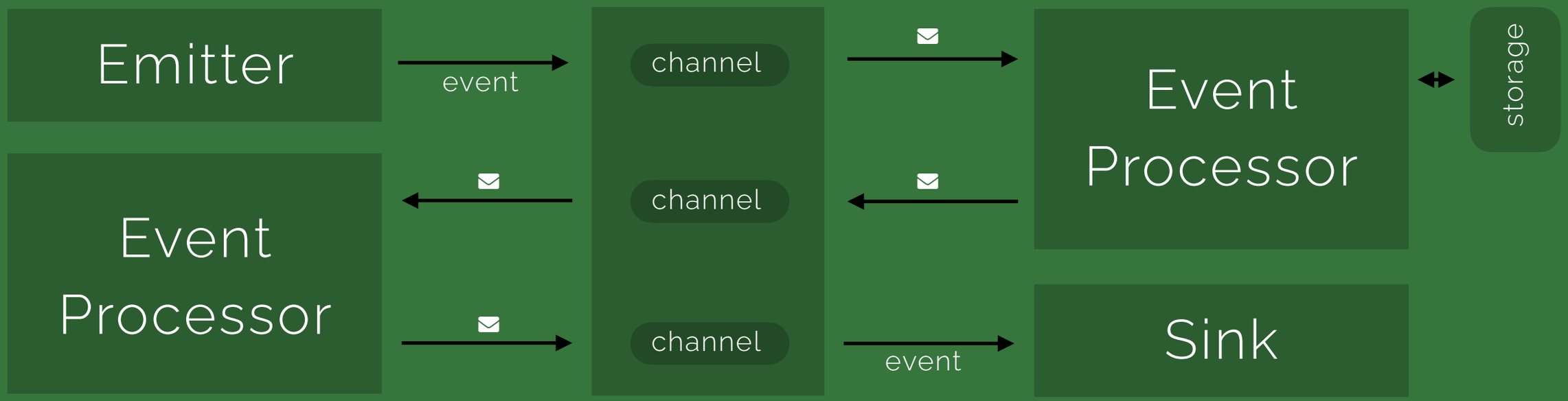
FEDERATION/CLUSTERING

The broker can be federated/clustered, making publisher connection independent of the node. Inner logic makes sure that the message is routed to the appropriate subscriber.



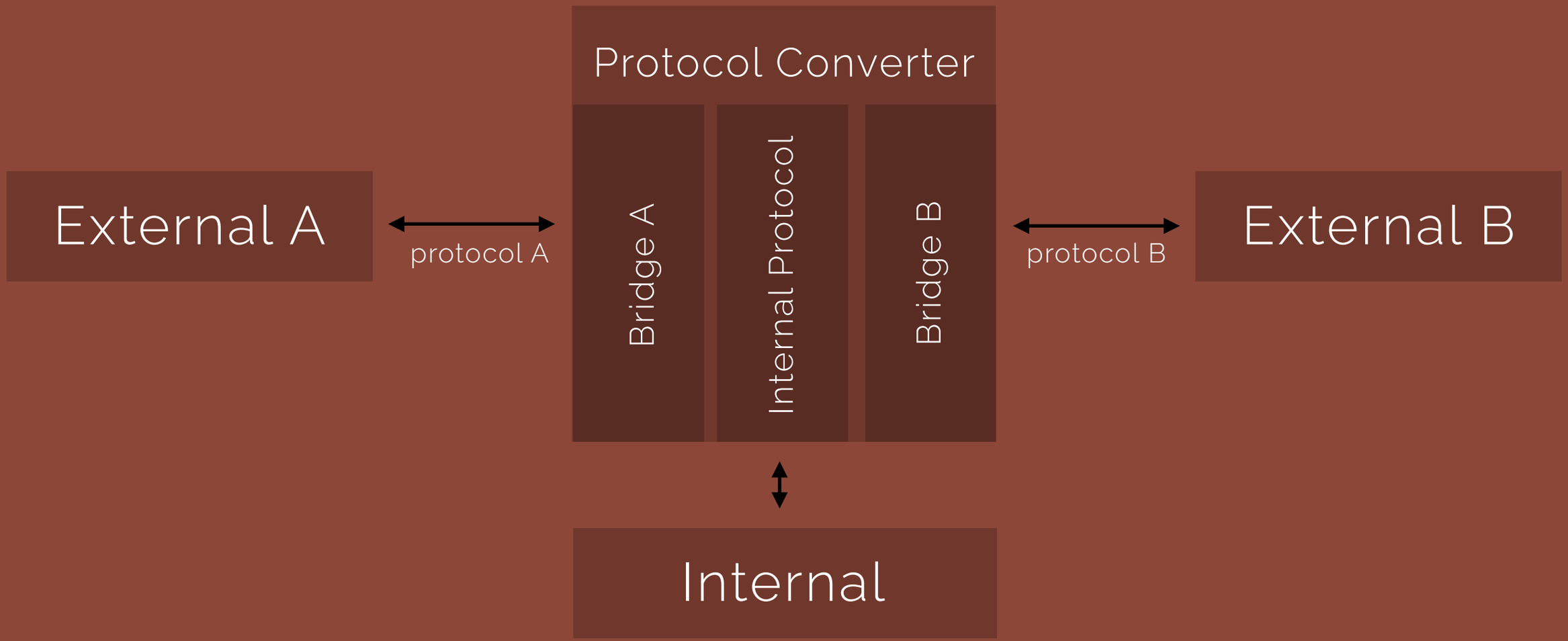
EVENT DRIVEN ARCHITECTURE

Propagates events among loosely coupled components. **Event emitters** detect, gather, and transfer events. **Event consumers** (or sinks) apply a reaction as soon as an event is presented. **Channels** are conduits in which events are transmitted from event emitters to event consumers.



PROTOCOL BRIDGE

A mechanism used to **convert** standard or proprietary protocol of one device to the protocol suitable for the other device or tools to achieve the **interoperability**.

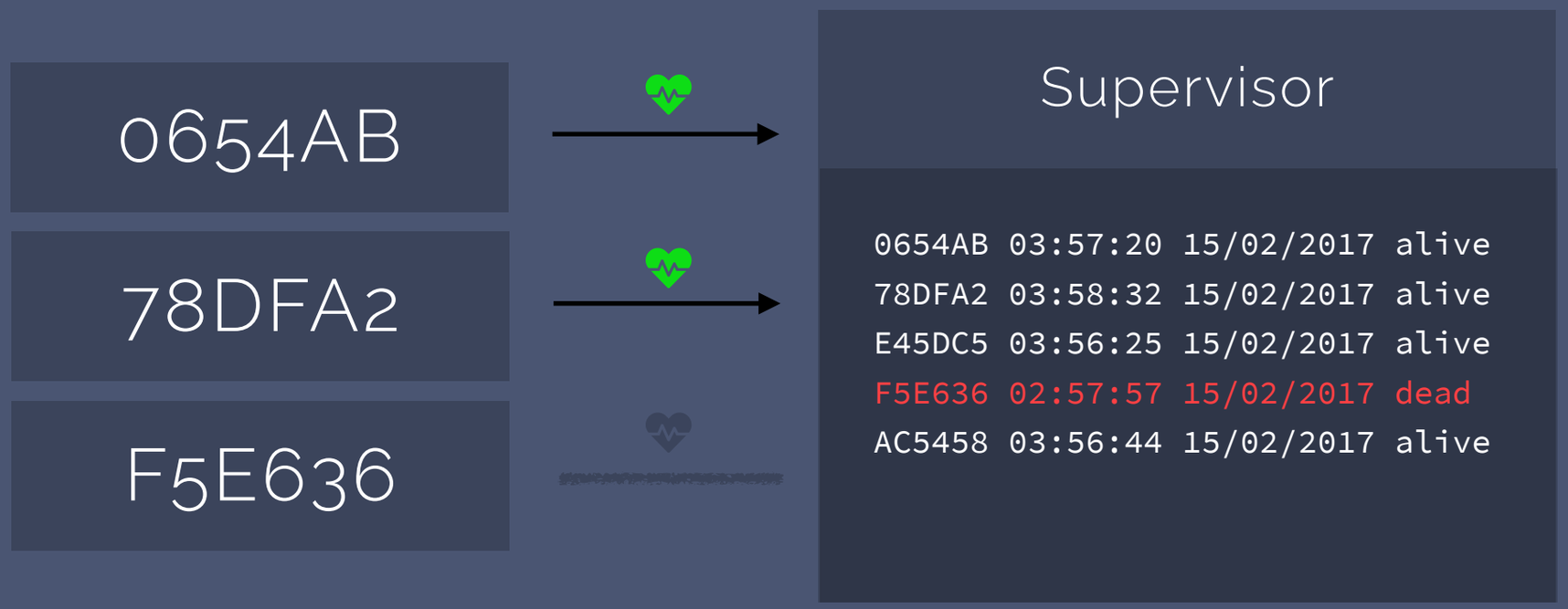


SERVICE LOCATOR

Encapsulates the processes involved in obtaining a service with a strong abstraction layer, by using a central registry known as the **service locator**, which on request returns the information necessary to perform a certain task.

HEARTBEAT

A **periodic signal** indicate normal operation or to synchronise parts of a system. Usually sent between at a regular interval in the order of seconds. If the endpoint does not receive a heartbeat for a time — usually a few heartbeat intervals — the component that should have sent it is assumed to have **failed**.

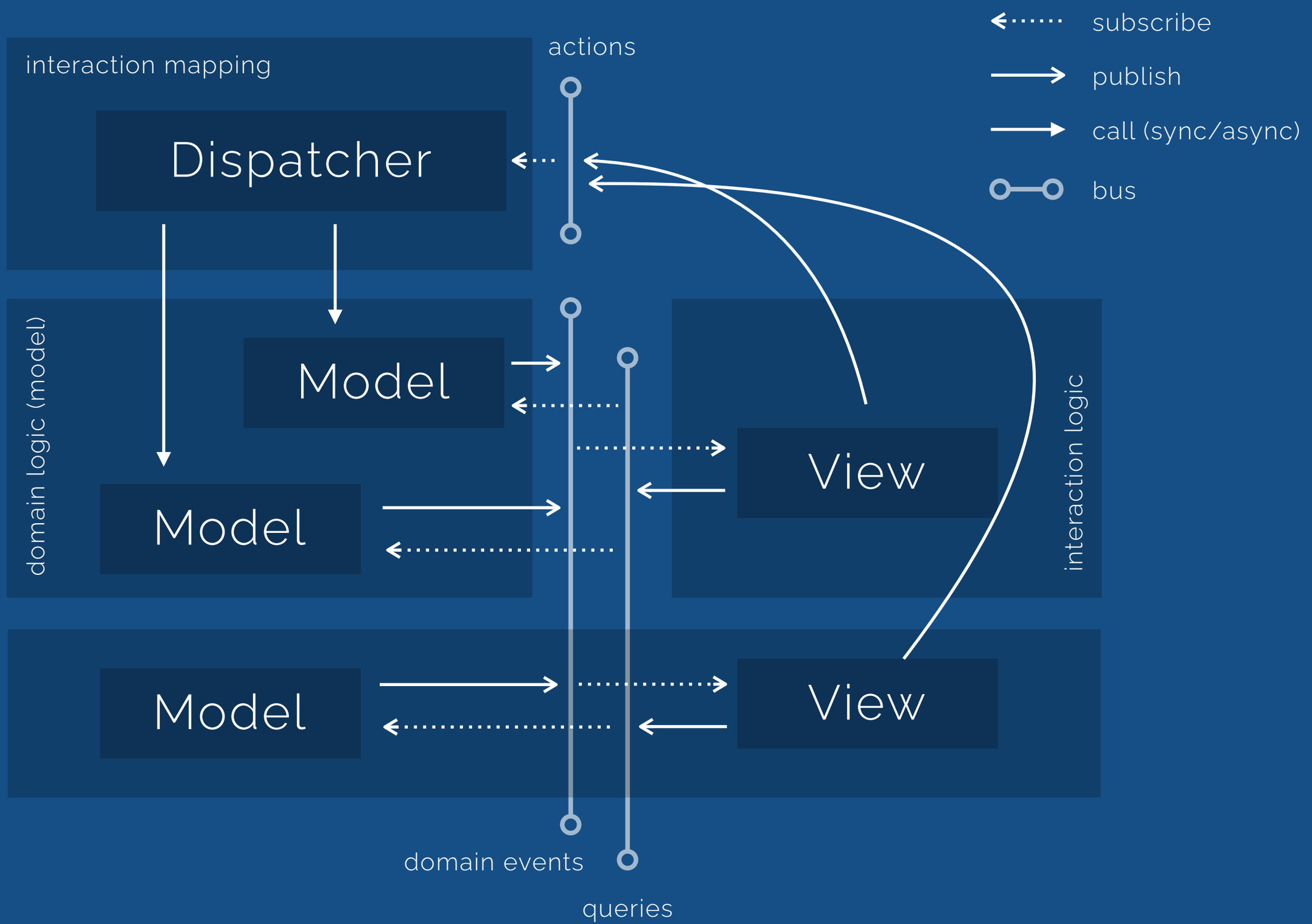


TIMEOUT

A specified **period of time** that will be allowed to elapse in a system before a specified event is to take place, unless another specified event occurs first; in either case, the period is **terminated** when **either event takes place**.

```
def getValue: Double = {
   device.getValue(timeout = 5s).orElse(0.0);
}
```

CQRS ARCHITECTURE



TECHNOLOGICAL ARCHITECTURE
(EXAMPLE)