Akka Introduction

LLAAMA - May 2022

Akka is developed and supported by **Lightbend**

Agenda

Akka Actors & Cluster

Multithreading, Akka Actors & Systems, Actor location, Actor hierarchy, Supervision, Exception-handling, Reactive Architecture, Cluster, Domain-Driven Design, Singleton, Sharding, Split-Brain Resolution, Protocol and serialization…

Akka Streams

Source, Flow, Sink, Backpressure, Materialization, Graph, Operators,...

Akka Background

Akka Background

- Actor Paradigm first described in a paper in 1973 by Carl Hewitt
- Akka is a Scala project that started in 2009, current version is 2.6.x
- Very much inspired from Erlang Actors
- Akka Actors were **not typed** until 2.6
- Akka can be used in a **Object Oriented** as well as in a **Functional Programming** approach
- Scala engineers tend to favor the FP approach
- Java engineers tend to use the OO approach
- Used by ING, PayPal, Tesla, Netflix,...

Akka, Bottom-Up

Java Concurrency

Java and the JVM aimed from the very beginning at becoming the platforms for **multi-processed, multithreaded and concurrent** applications development but :

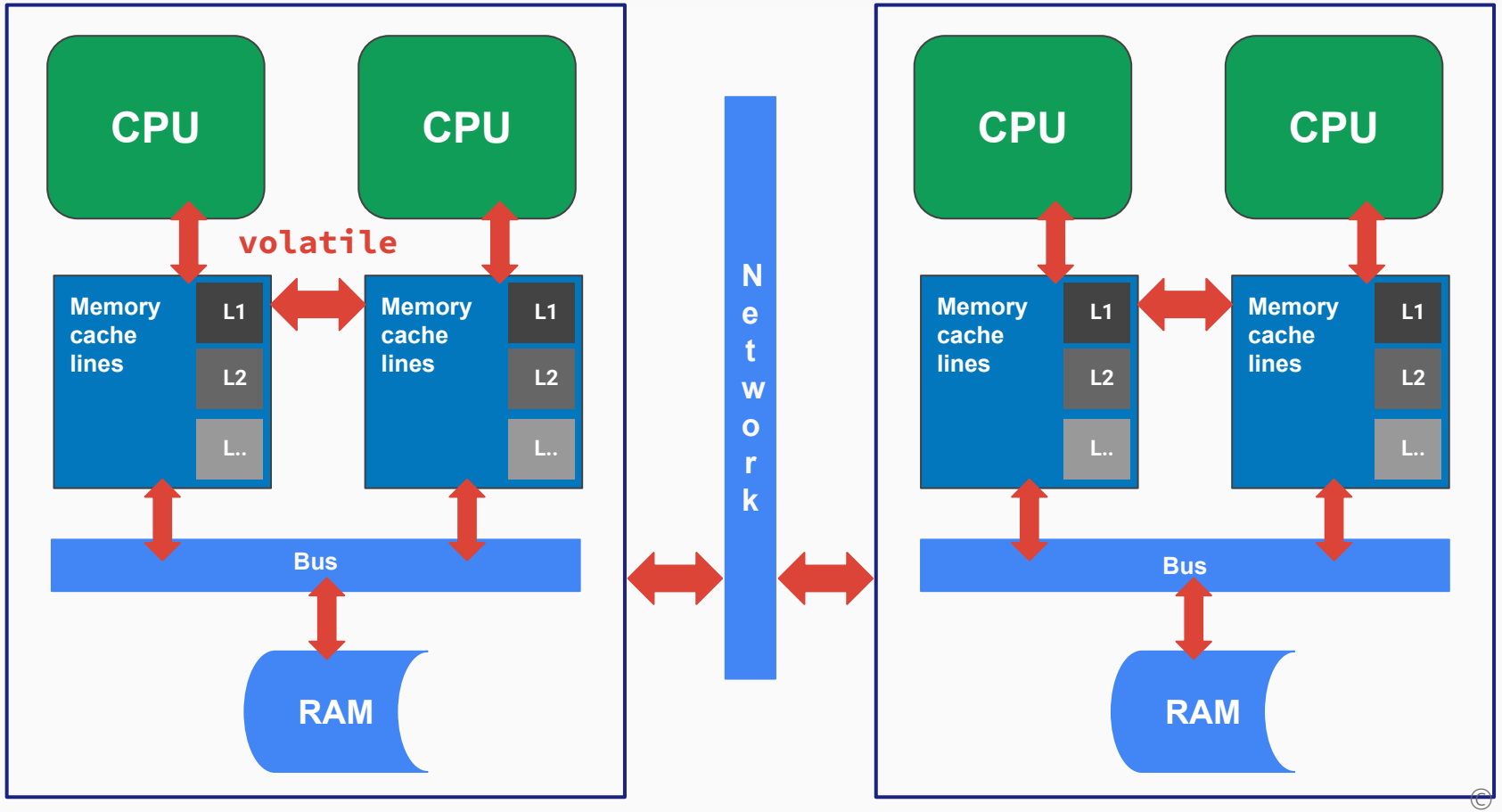
Concurrency (and reasoning about it) is difficult and has always been!

Low level constructs for concurrent programming are complex:

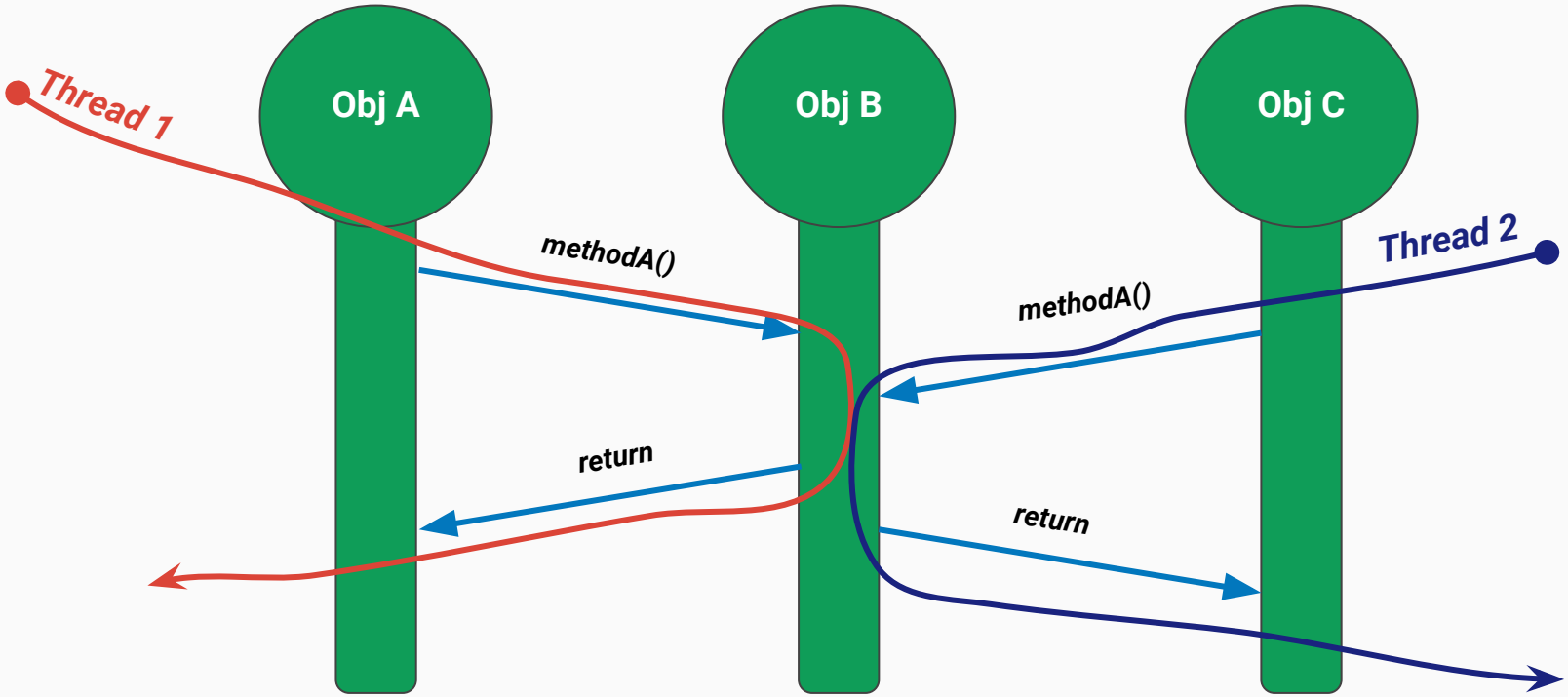
- `happens-before` relationship
- Monitor locks with `synchronized`
- `volatile` to synchronize writes/reads

Java Memory Model is improving but still difficult to understand

CPU Architecture and Concurrent Programming



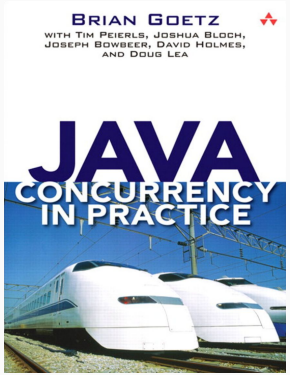
(Java) Concurrent Object-Oriented Programming is Broken



Solutions?

runnable() {...}

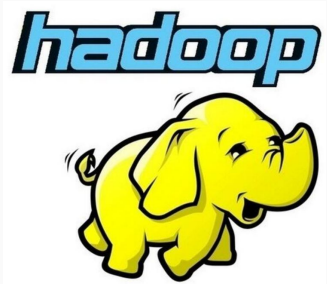
synchronized(...) {...}



Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

Interface Summary
Interface
BlockingDeque<E>
BlockingQueue<E>
Callable<V>
CompletableFuture.Asynchronou
CompletionService<V>
CompletionStage<T>
ConcurrentMap<K,V>
ConcurrentNavigableMap<K,V>



Actors: a better approach of Concurrent Programming

New Viewpoints

No shared memory anymore

~~**read-write**~~

Think about **inter-CPU** communication (almost) the same way as **network communication**

Think distributed as default
Optimize when local

Use **immutable messages** for concurrency

Actor Model Contract

An Actor is an abstraction for a computational entity that can:

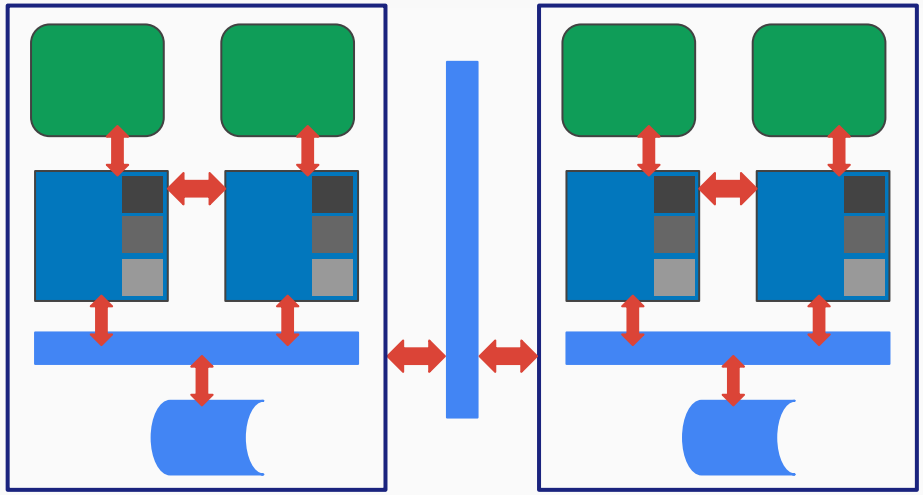
1. **send messages to other actors**
2. **create new actors**
3. **designate the behavior to be applied to the next message**
4. **Make local decisions, like modifying private state**

Akka Actors ***Happens-Before*** Rules

1. **Actor send rule,** send of message M happens before receive of message M by the same actor

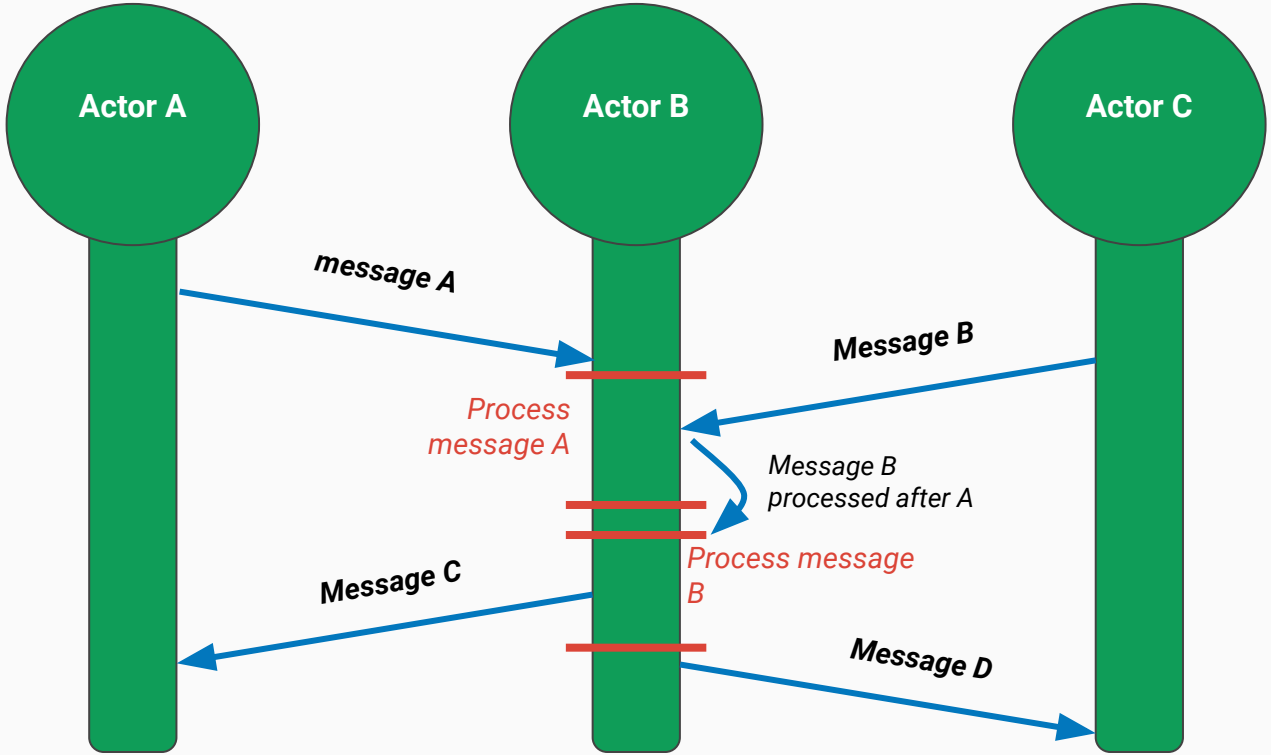
2. **Actor subsequent processing rule,** processing of a message happens before processing of next message by same actor

Actors, CPU architecture and concurrent programming



- Read-write paradigm is finished: there is **no shared memory** anymore
- Data exchange over the **network is not different from inter-CPU communication**
- Actor approach: keep state local and exchange immutable messages

Fixing Concurrent Object-Oriented Programming with Actors



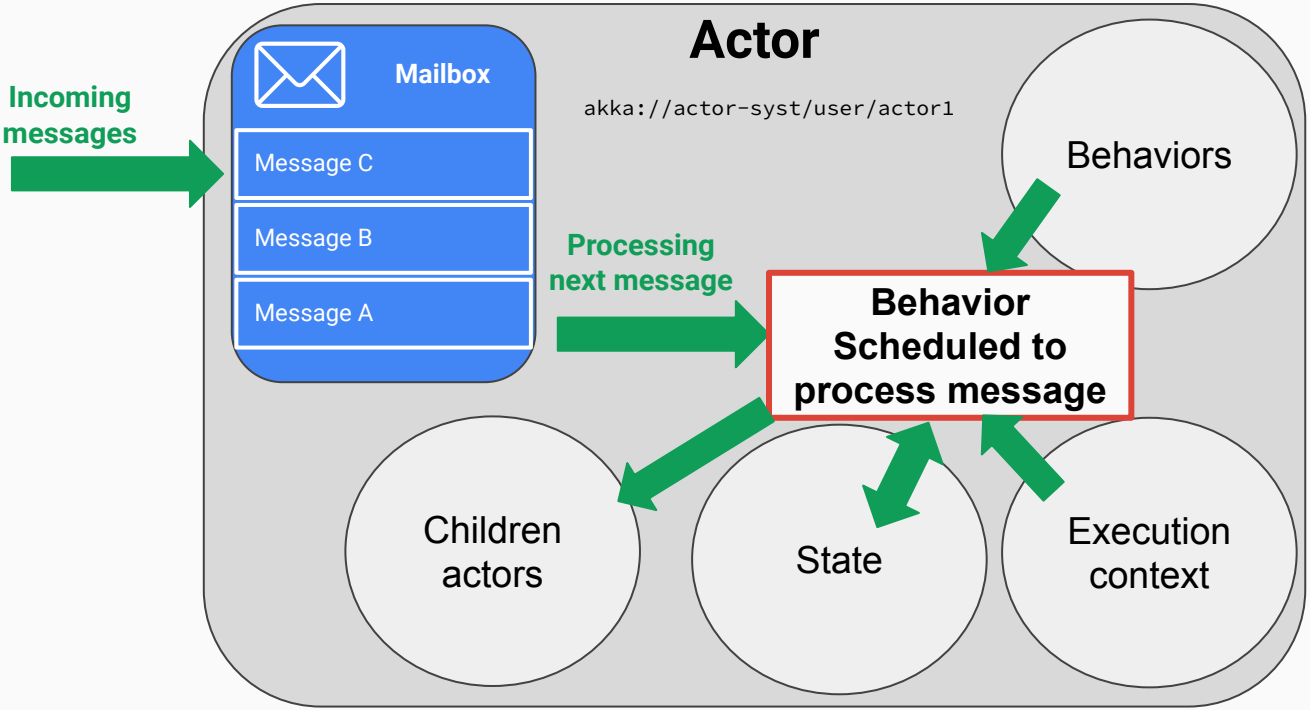
Akka Actors

- **Akka Actors are distributed by default,** all functions are built for clusters of hundreds of machines and JVMs
- Upside-down: **designed to be distributed,** optimized when local (instead of generalization from local to remote)
- Interactions between actors happen through **sending asynchronous immutable messages**
- Messages must be **immutable** and **serializable!** - when local, JVM object references

Are Actors what Object-Oriented Programming should be ?

Akka Actors: Lightweight Objects

Akka Actor, Lightweight Construct



What does an Akka Actor contain?

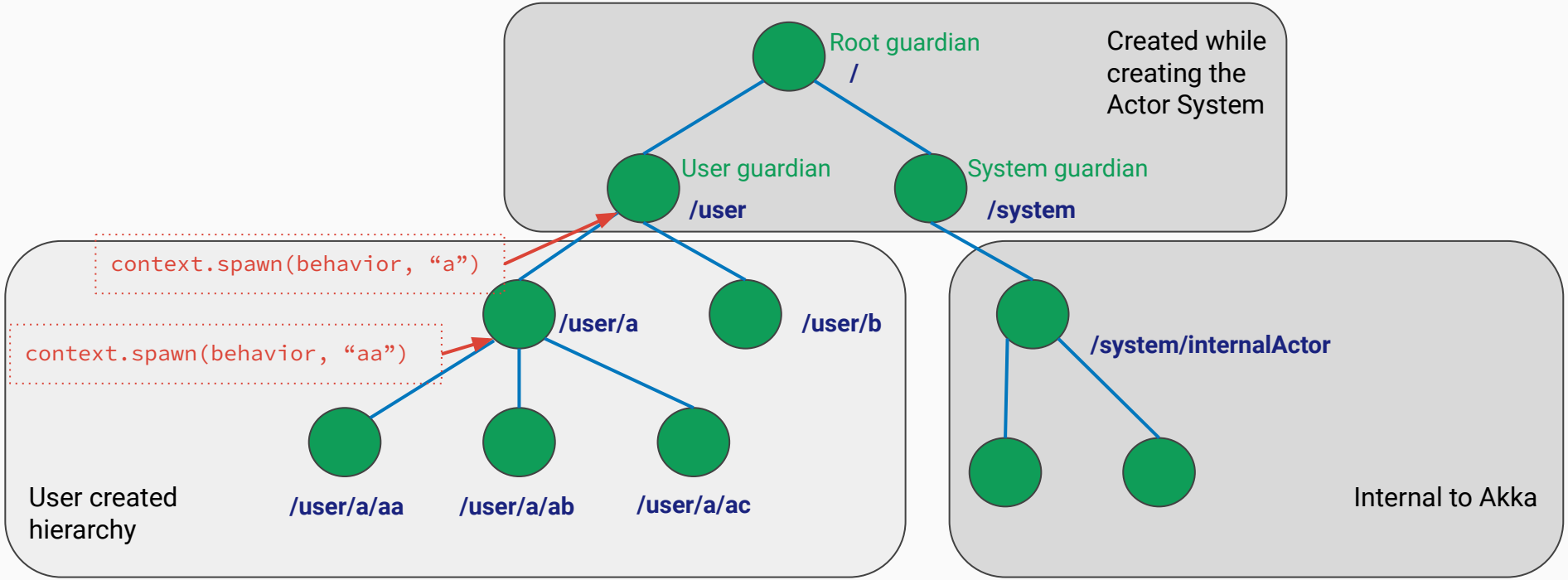
- **Actor Reference**
  - A kind of proxy that enables to talk to the actor from anywhere while shielding Actor internal state
- **State**
  - Any state related to the application logic, hidden from outside, modified like it would own its own thread
- **Behavior**
  - Behaviors define what action should be performed for which received message
  - Behaviors change over time and every message processing should provide next behavior
  - Messages are statically typed
- **Mailbox**
  - An Actor has exactly one mailbox, a FIFO queue.
- **Child Actors**
  - An Actor can (should) have child actors to delegate tasks
- **Supervision Strategy**
  - What to do in case of failure ? `restart` x times - `resume` - `stop`, ...

Akka ActorSystem: The plumbing

Akka Actor System

- An `ActorSystem` is the home of a hierarchy of **Actors**
- An `ActorSystem` is a heavyweight construct
  - It might allocate many threads
  - There should be one per logical application
- The `ActorSystem` hierarchy enables splitting-up tasks in small and logical pieces
- The top-level Actor (given at construction) should be a supervisor

Akka Actor Hierarchy



/deadLetters	Receives messages that cannot reach destination
/temp	Guardian for short-lived system actors (e.g. to manage `ask` implementation)
/remote	For Remote references

Akka Actor References and Addresses

- **Actor references** are available when actors **are created**

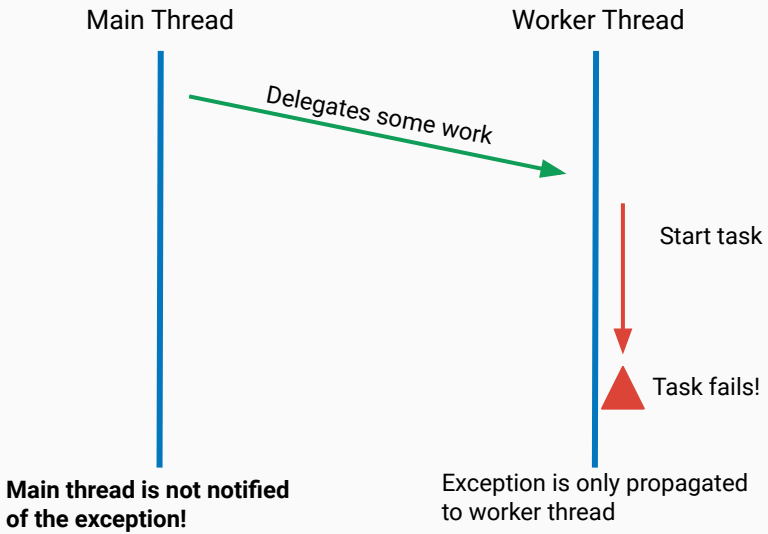
```
val greeter: ActorRef[HelloWorld.Greet] = context.spawn(HelloWorld(), “greeter”)
```

```
ActorRef<HelloWorld.Greet> greeter = context.spawn(HelloWorld.create(), “greeter”
```

- **Actor references** can be part of the message **protocols**
- **Actor references** correspond to real existing **actors** and are bound to their life-cycle
- **Actor references** can be obtained through the **Receptionist**
- **Actor Paths** are unique accessors, in form **of path structures**, from the actor system, through the actor hierarchy to the actor itself
  - `akka://actorsystem/user/a/b/c` (local)
  - `akka://actorsystem@host.example.com:5555/user/a/b/c` (remote)

Call stack

Traditional



Actor based

- Non-blocking, message-driven tasks delegation amongst Threads
- ***Failures*** are part of the domain model
- ***Let it crash*** approach
- Response deadlines are handle with ***timeouts***

Warnings

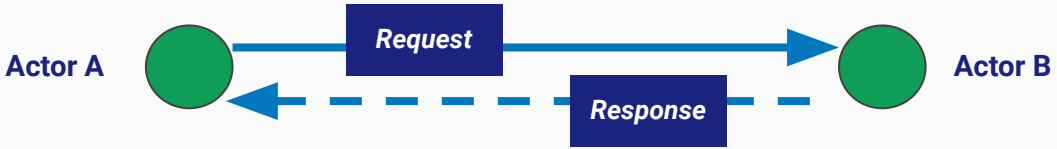
- In an actor, messages can be processed by **different** threads
- Don’t **close** over internal Actor state
- Avoid as much as possible **blocking** in actors, when unavoidable use a **dedicated dispatcher**
- Don’t try to **control the order** in which **messages are processed** in big systems !

Asynchronous Interactions

Akka Actor Interactions

`Tell` - *Fire and Forget*

`Tell` - *Request-Response*



`Ask` - *Request-Response, 1-1, with timeout*

- `Tell` is thread safe!
- `Tell` is asynchronous and returns directly
- Wrapper can be used to adapt protocol objects
- `Ask` needs timeout
- Many other constructs can be built up on top of `Tell`

Akka Actor Protocol

- The protocol of an actor are the **types** the actor understands
  - It includes **accepted messages** as well as **reply types**
- Protocols can be defined for multiple actors, contributing to the application logic
- A `Behavior` is defined with a protocol type
-  `sealed trait` and `case class`
-  `interface` and `static final class`

ActorContext Kind of “Decorator”

Akka Actor Context

`ActorContext` *provides the Actor with important features*:

- `spawn` to create child actors (**Actor contract**)
- Access to own identity (self)
- Access to children
- `messageAdapter` (wrap messages to translate protocols)
- Access to `ActorSystem` of the actor
- Provides `watch` to register for `Terminated` notifications
- Logging

Message Delivery Contract

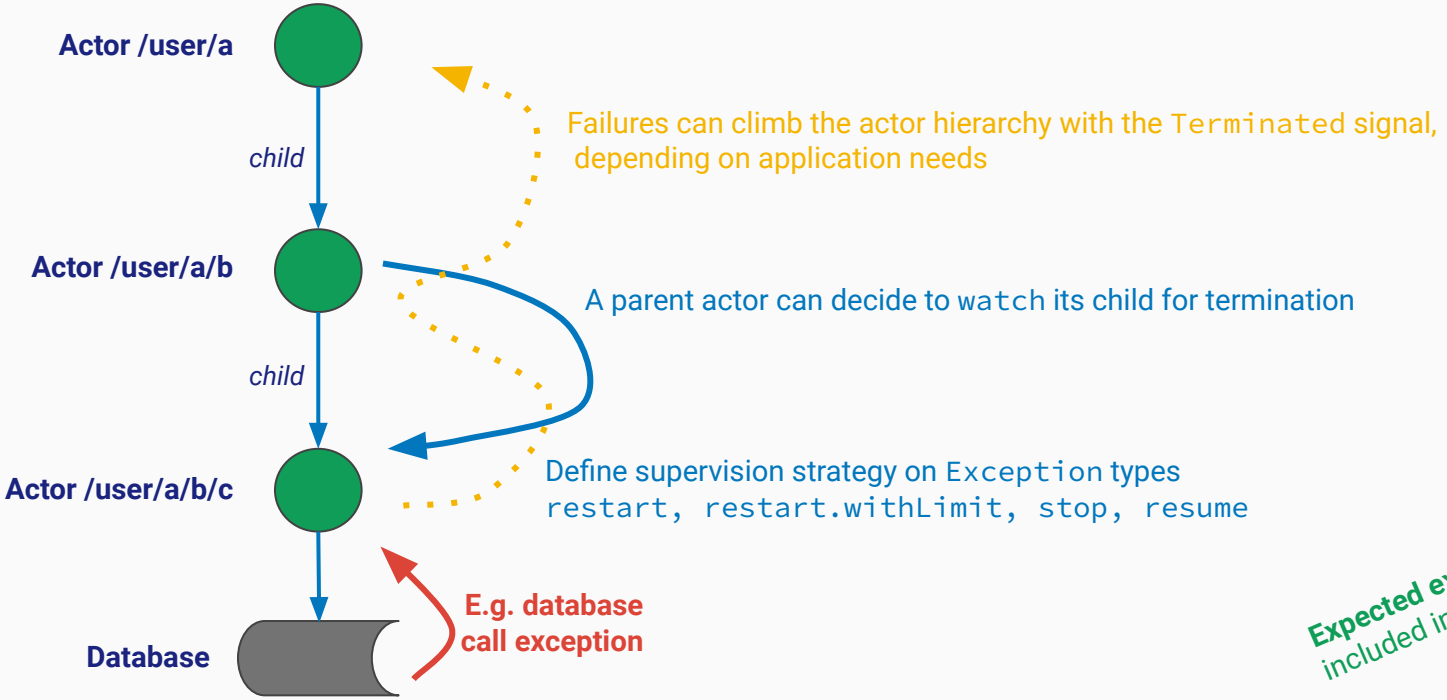
1. **At-most-once**

2. **Message ordering** per **sender-receiver pair**

“Let it crash!”

Akka Fault Tolerance and Supervision

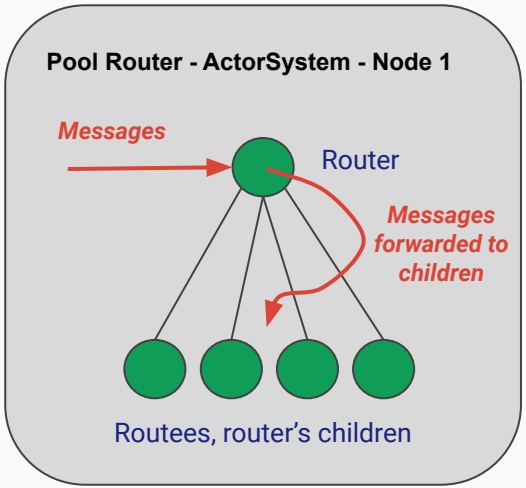
"Shit happens" therefore apply the ***Let it crash approach*** !

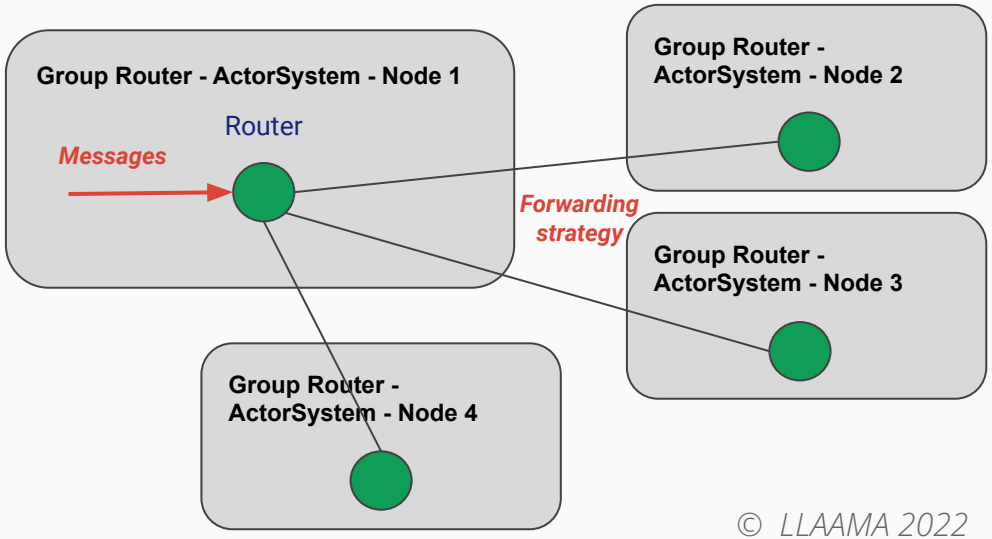


Expected exceptions should rather be included in the **actors protocols**!

Routers

- Distributing messages to set of *delegated* actors
- Enables *parallel* processing
- **Pool Router**: *routees* are *router's* children, local
- **Group Router**: uses `ServiceKey` and `Receptionist` to find *routee*, clustered
  - Available routing strategies: Round Robin, Random, Consistent Hashing





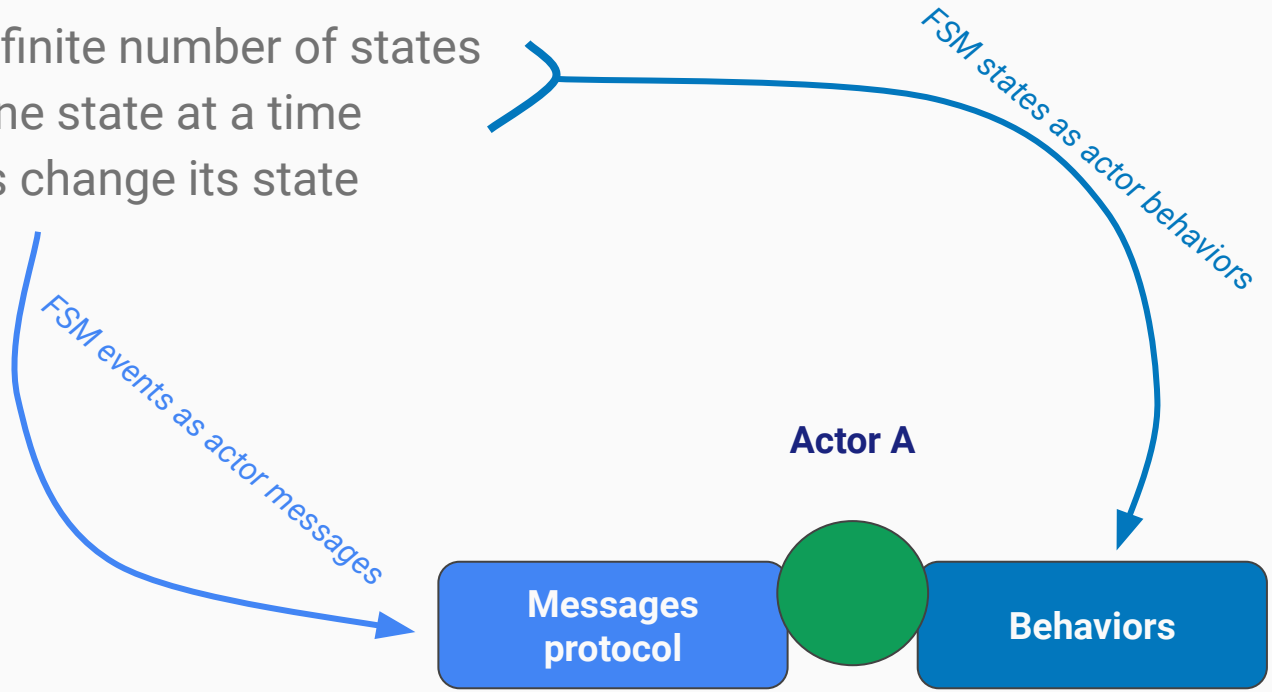
Akka Message Stashing

- Stashing enables buffering messages before they are passed on to an actor
- It is an important feature in **Domain Driven Design** and **Sharding**

Akka Actor model and Finite-State Machine

Mathematical model of computation, part of automata theory, a Finite-State Machine (FSM):

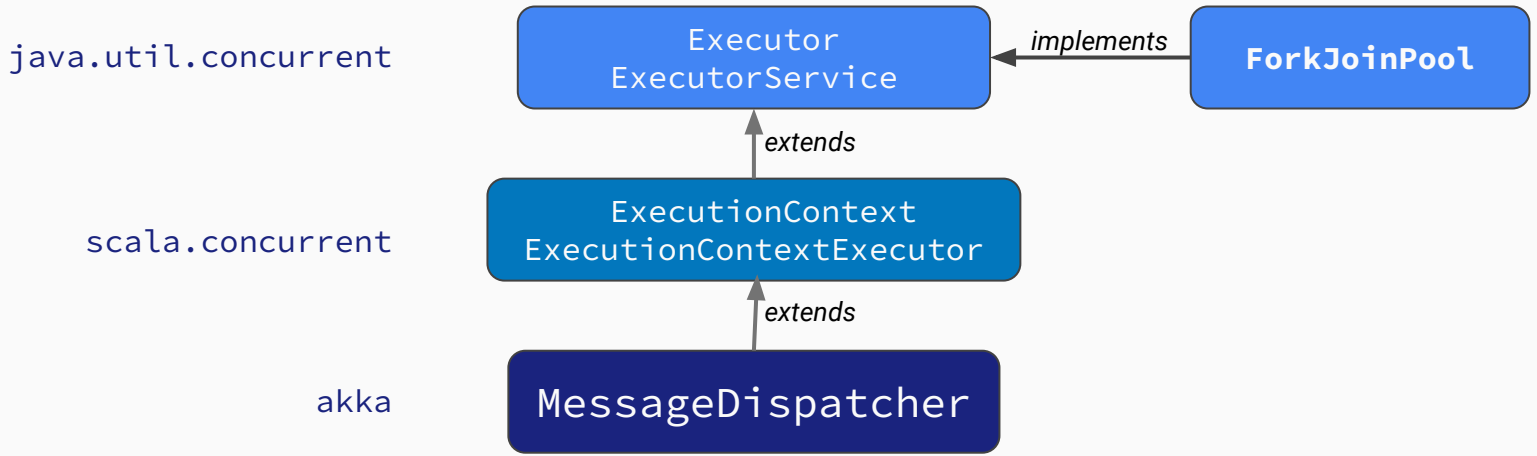
- Has a finite number of states
- Is in one state at a time
- Events change its state



Under the hood: **MessageDispatcher**

Akka Dispatchers

- `MessageDispatcher` is the **engine** at the heart of an `ActorSystem`
- Default `MessageDispatcher` implements a **fork-join-executor**
- Internal actors are protected by an internal dispatcher
- If needed, other dispatchers can be used (e.g. *blocking*, for I/O)



Miscellaneous

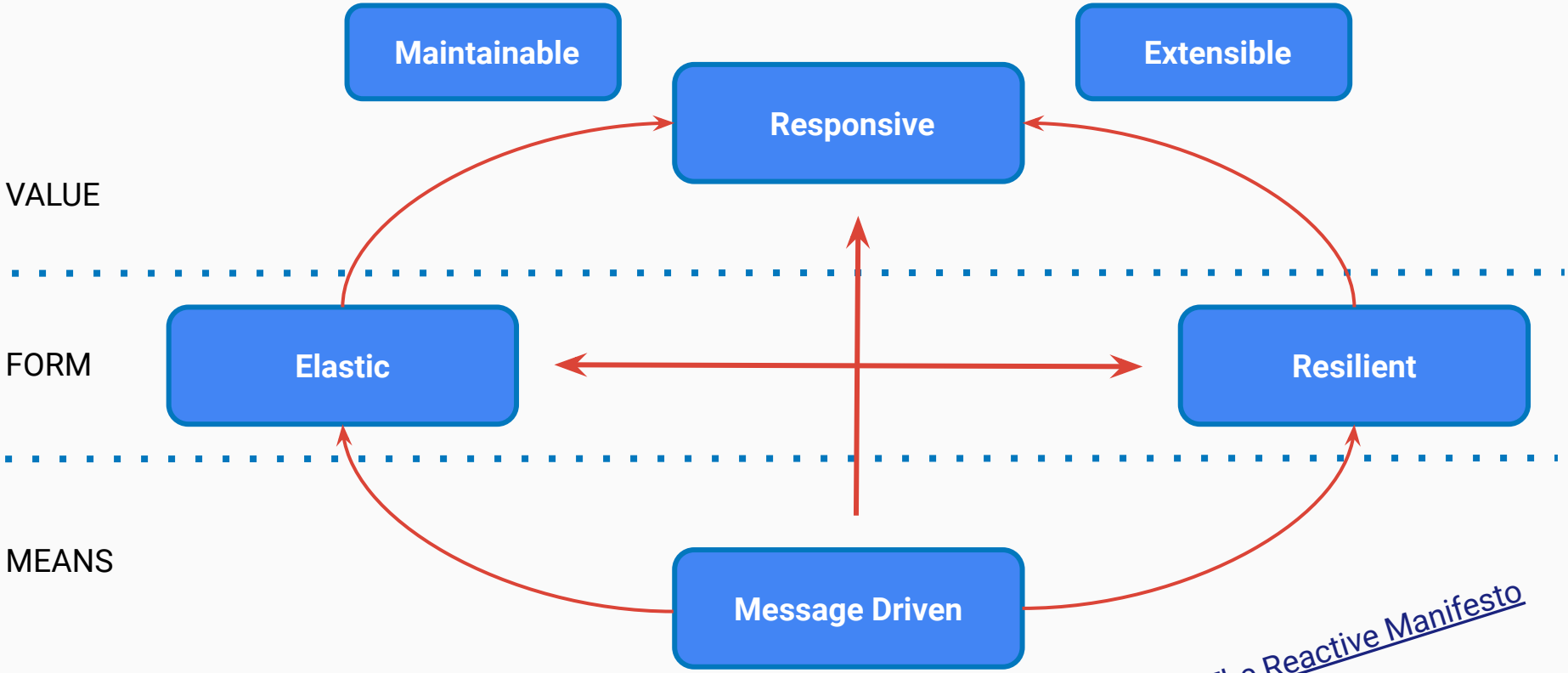
- By default, every Actor has an unbounded `Mailbox`
  - Mailboxes can be configured
  - Other mailbox types are available
  - A custom mailbox can be provided
  - Akka doc - mailboxes

- Configuration
  - All configuration for Akka happens in instances of `ActorSystem`
  - Defaults come from `reference.conf`
  - Merging is done with `application.conf`, `application.json`, `application.properties`
  - Akka doc - configuration

- Testing
  - Asynchronous testing and Synchronous behavior testing
  - Akka doc - testing

Akka: Top Down

Reactive Architecture



Reactive Systems Features

Responsive

Flexible

Loosely-coupled

Scalable

Resilient and tolerant to failure

Elastic

Message-drive (asynchronous)

Location transparency

Failures as messages

Back-pressure

Non-blocking

Distributed Computing with Akka Cluster

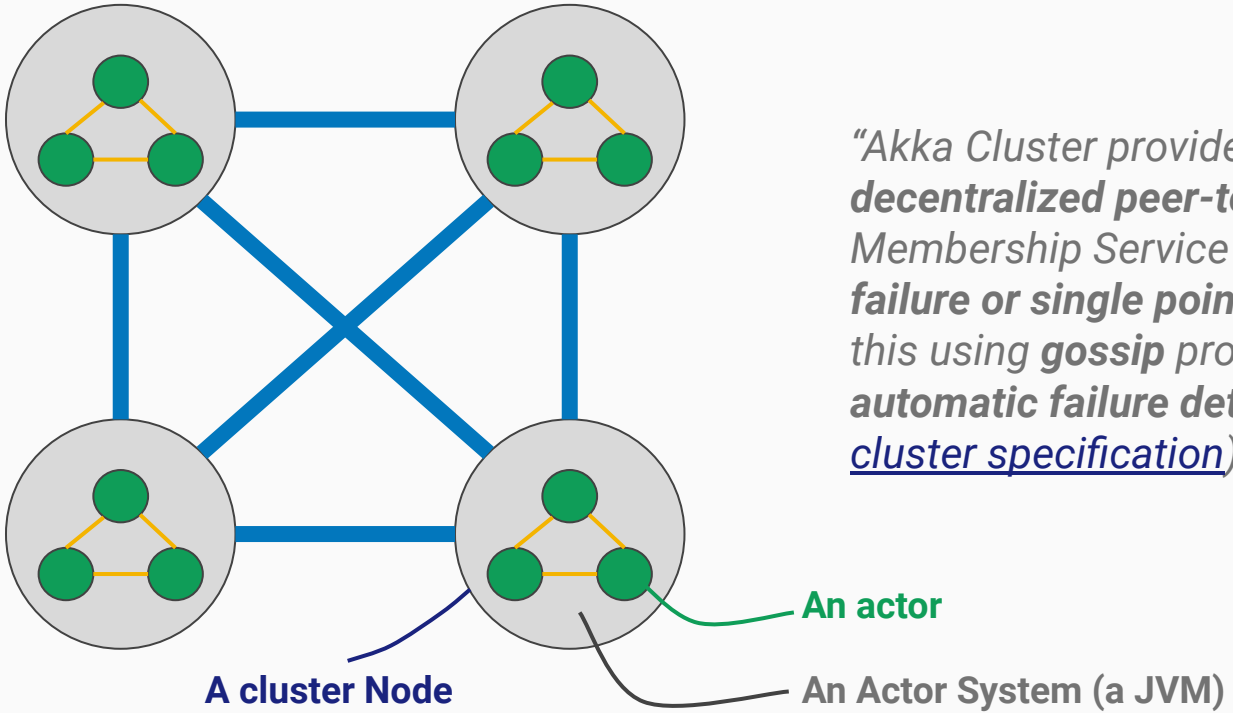
One Actor

is

No Actor

What is an Akka Cluster?

Multiple ActorSystems nodes joining to form one coordinated distributed application



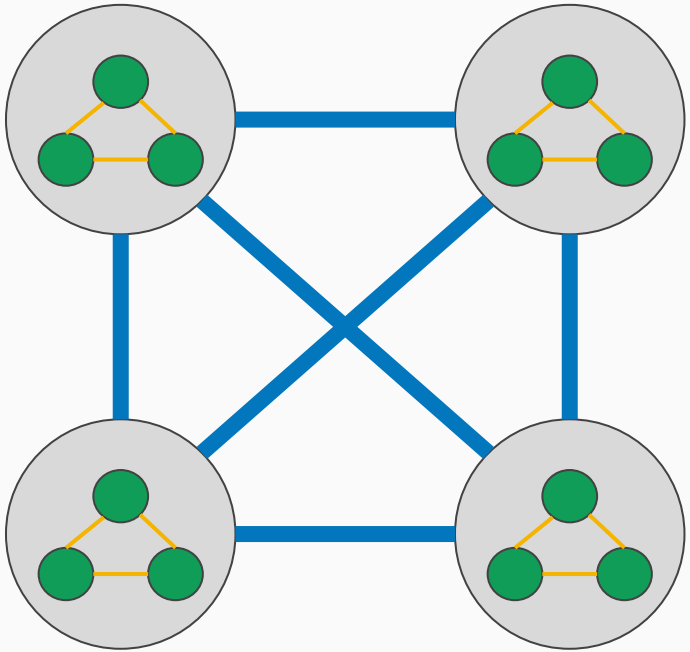
*"Akka Cluster provides a **fault-tolerant decentralized peer-to-peer** based Cluster Membership Service with **no single point of failure or single point of bottleneck**. It does this using **gossip** protocols and an **automatic failure detector.**" (Akka doc - cluster specification)*

Akka Cluster Benefits

- **Akka Cluster** is the best way to build **Akka Actor Systems** with several hundreds actors
- **Akka Cluster** provides everything that is needed to build complex **distributed systems**
- **Akka Cluster** fits perfectly with the goal of **Reactive systems** as well as **cloud Native** systems
- **Akka Cluster** fits very well with platforms like **Kubernetes**
  - E.g. it can easily use K8s APIs for node discovery
  - It can use K8s lease feature for SBR
- There is rarely the need to use **Akka remote** anymore

Akka Cluster is like a Beehive

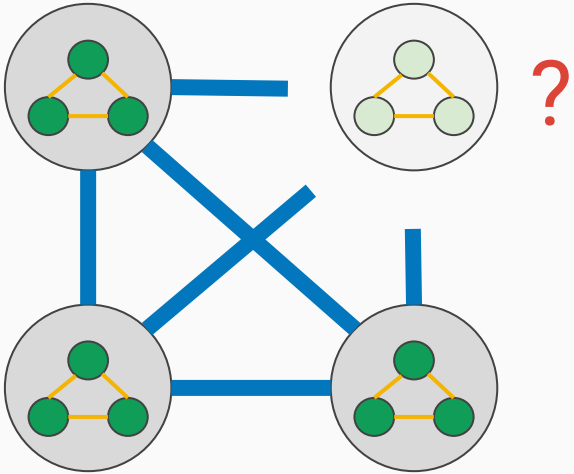
How to Build a Cluster?



Challenges:

- Starting Cluster
- Cluster Membership
- Leadership
- Failure Detection
- Network Partition
- Roles
- Message Serialization

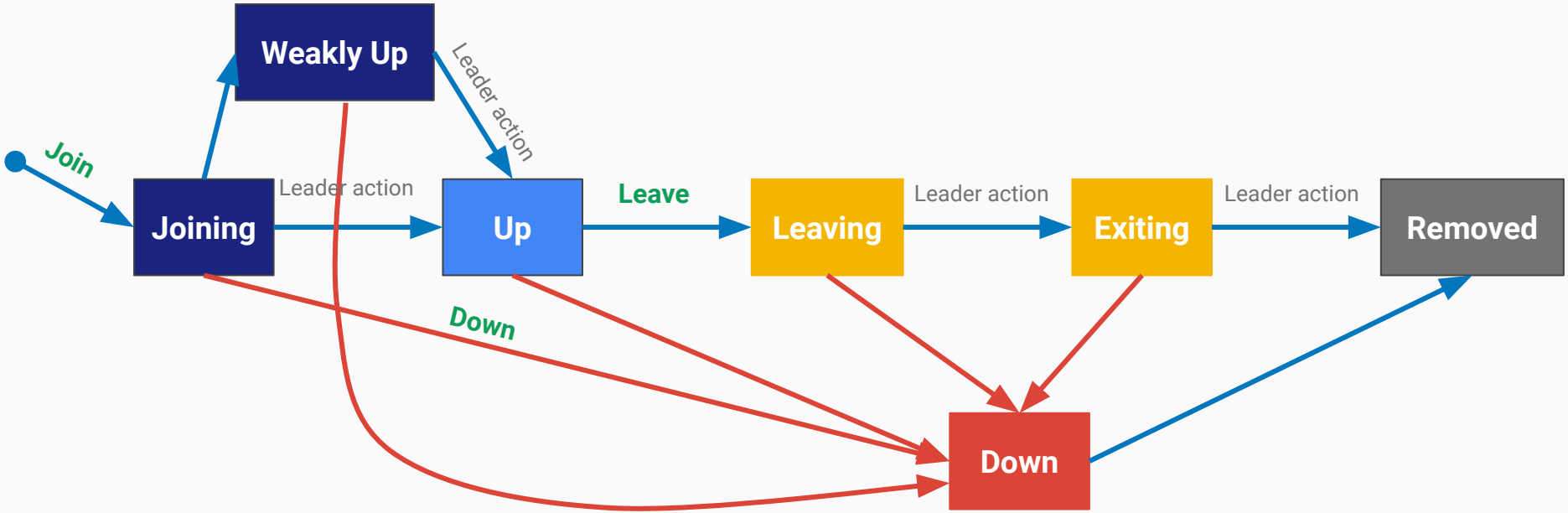
Membership



Node Member States

- **Joining**
- **Weakly Up**
- **Up**
- **Leaving**
- **Down**
- **Removed**

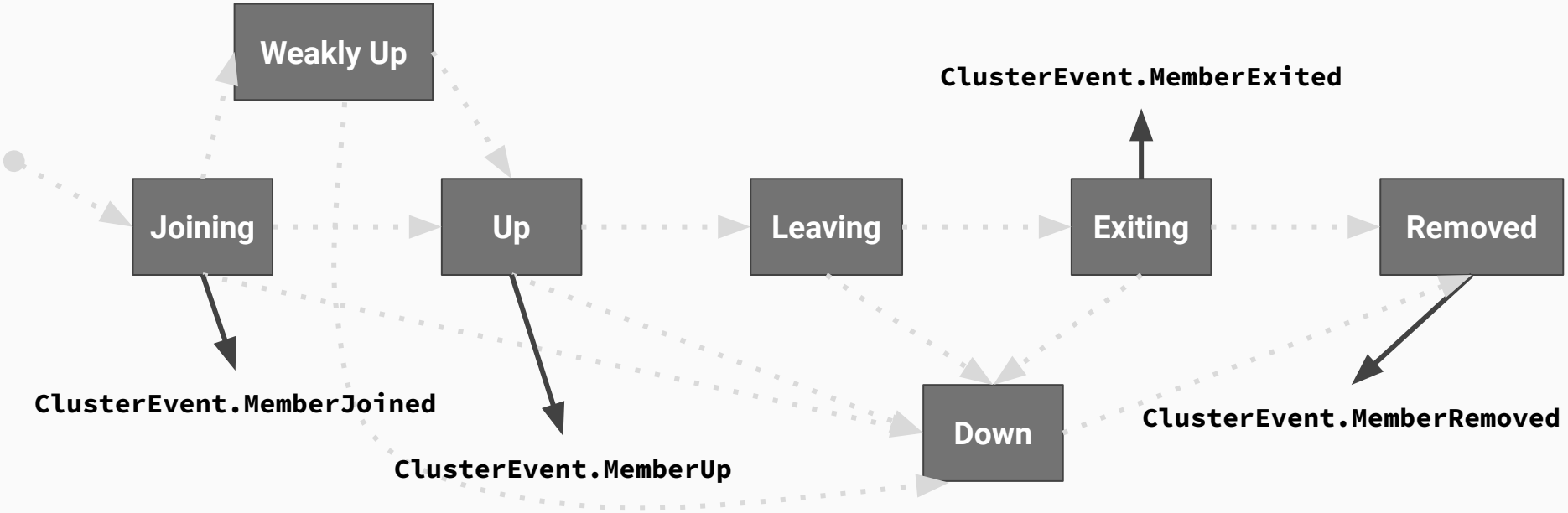
Nodes States Transitions



Join, Leave, Down: User actions

Weakly up: nodes waiting for gossip convergence to be reached

Members Events



`ClusterEvent.Unreachable:` member not reachable by failure detector

`ClusterEvent.Reachable:` member reachable again (after being unreachable)

`ClusterEvent.MemberPreparingForShutdown:` member preparing for full cluster shutdown

`ClusterEvent.MemberReadyForShutdown:` member is ready for full cluster shutdown

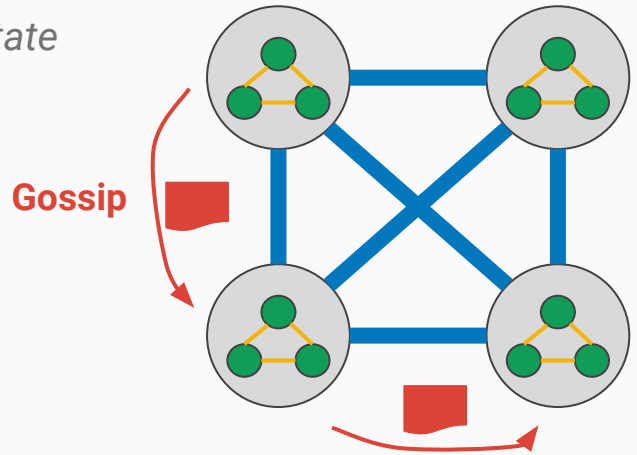
“Anyone can start a rumor, but nobody can stop one.”

Gossip Convergence

- **Gossip Protocol** (aka epidemic protocol) to spread out cluster state around nodes
- Random information dissemination, vector clock (node, counter, state) for partial ordering of events and causality violation detection
- Gossip message Serialization done with *protobuf*

- **Gossip Convergence**
  - when "*one node can prove that the cluster state it is observing has been observed by all other nodes from the cluster*"
  - Any **unreachable** node prevents Gossip Convergence

- Optimizations of the **Gossip Protocol**
  - Digests sent instead of full data
  - Changes to the algorithm depending on number of nodes



Special Roles in Akka cluster

- **Leader**
  - Defined when (gossip) convergence is reached
  - No special node, just a role
  - Deterministic way to recognize it
  - Confirms states changes (e.g. joining -> up)

- **Seed Nodes**
  - Fixed entry points for nodes to join the cluster
  - No special other role
  - Can be defined in conf or as `system.properties`
  - First seed node needs to be available when starting the cluster
  - *Cluster Bootstrap* module enable automatic discovery of nodes

application.conf:

```
akka.cluster.seed-nodes = [
  "akka://MyCluster@host1:2552",
  "akka://MyCluster@host2:2552"]
```

JVM system properties:

```
-Dakka.cluster.seed-nodes.0=akka://ClusterSystem@host1:2552
-Dakka.cluster.seed-nodes.1=akka://ClusterSystem@host2:2552
```

Failure Detector

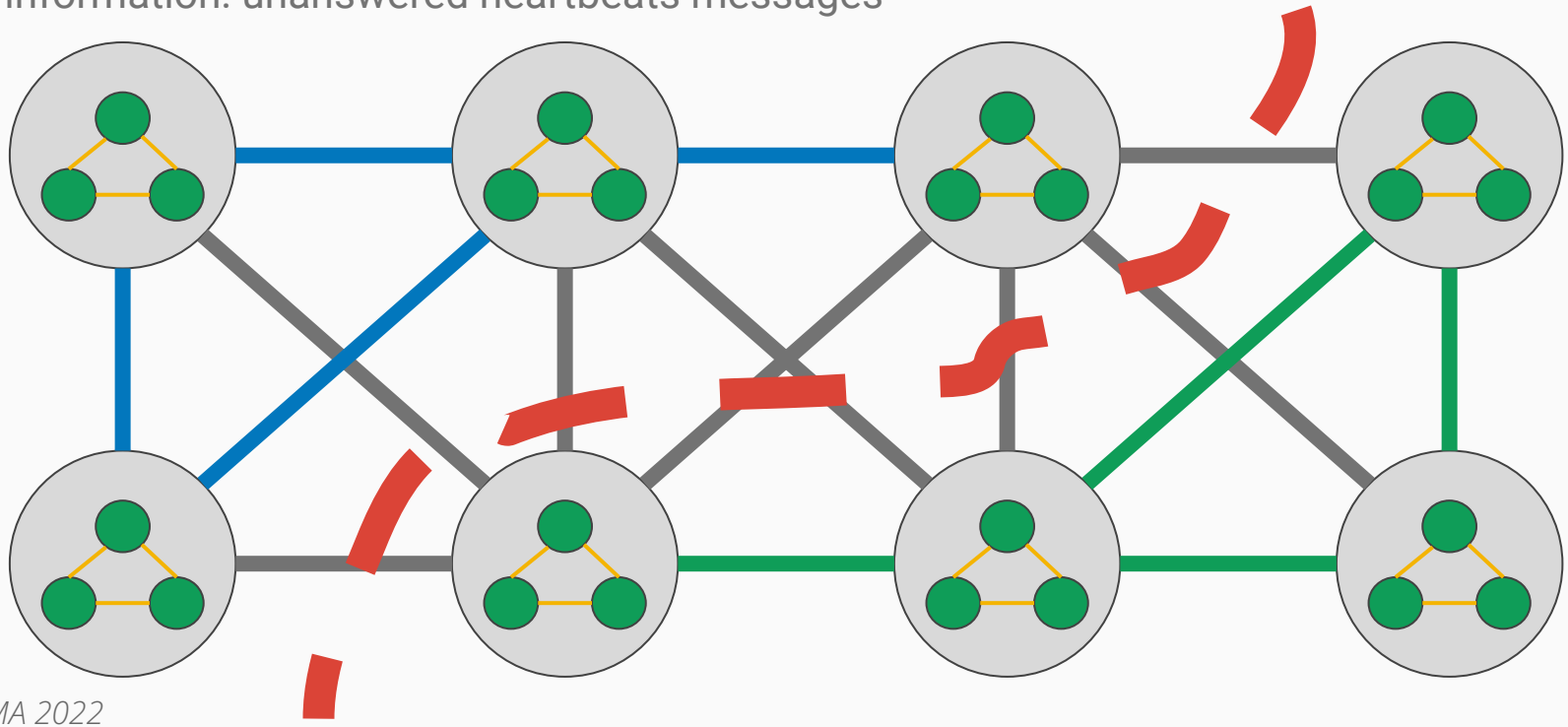
- Detects **unreachable** nodes
- Decides how to react
- Uses the gossip protocol
- One node is monitored by N (default 5) other nodes
- An **unreachable** node will be quarantined and eventually **downed** and **removed**

- **Failure detection** is based on ***Phi Accrual Failure Detector***
  - Distinguish expected latencies (network, garbage collections, etc.) from crashes
  - Based on regular *heartbeat* messages
  - Learns from previous failures
  - Calculates a likelihood (`phi`) of a node to be down
  - Threshold (`acceptable-heartbeat-pause`) can be defined by user

Split Brain

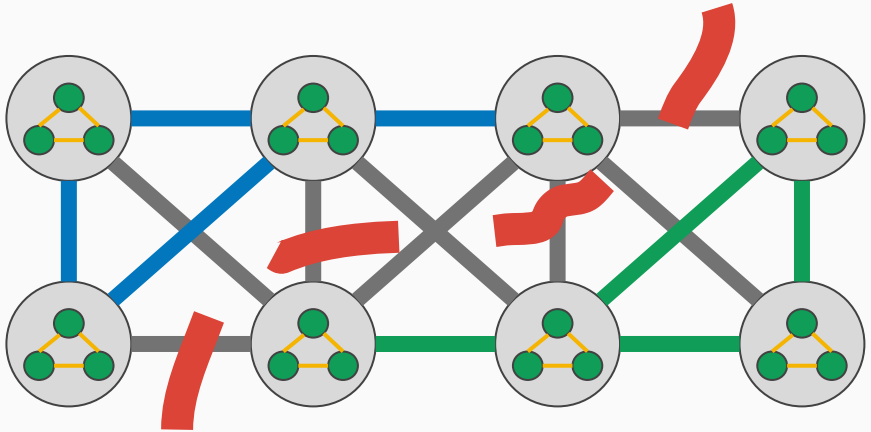
Machine crashes, unresponsiveness,
network partitions cannot be distinguished from each other !

Only information: unanswered heartbeats messages



Split Brain Resolution

- Decisions must be taken…
- There are multiple SBR strategies
- Most obvious strategy: **Keep Majority**
  - Downs the nodes if they are in the minority
- **Static Quorum**
- **Lease**
- **Keep Oldest**
- **Down all**

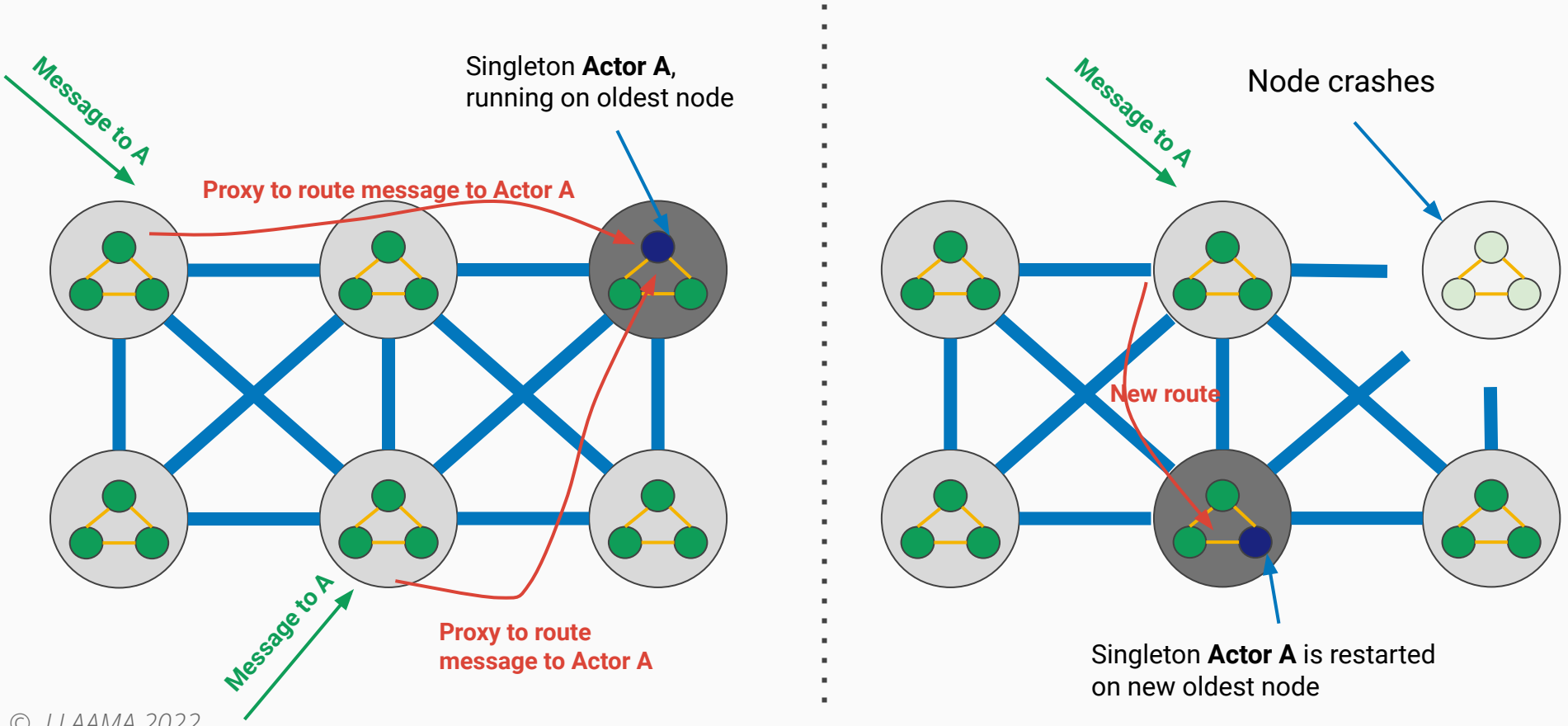


Singleton

Exactly one instance of an **actor** with a defined role in the **whole cluster**

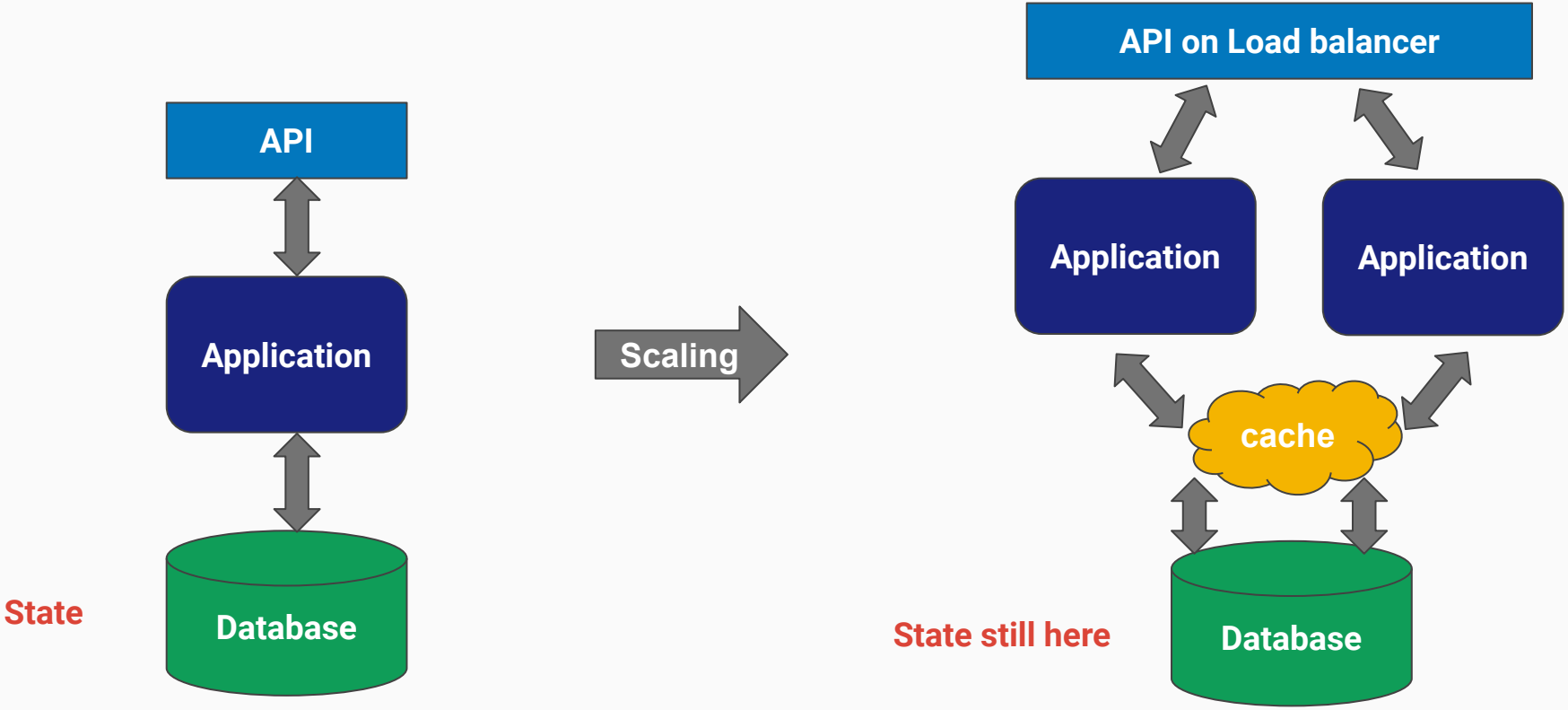
- Goal: **single point of responsibility**
- Start with `ClusterSingleton.init` on **all nodes** with given **role**
- **Singleton** will run on oldest node
- Cluster ensures there is one and only one instance available
- `ClusterSingleton.init` provides a **proxy ActorRef** to the Singleton wherever it's actually running
- Issues
    - Downing
    - Singletons can by definition become bottlenecks!

Singleton Reference and Proxy



Stateless versus Stateful Applications

Stateless (traditional) Applications

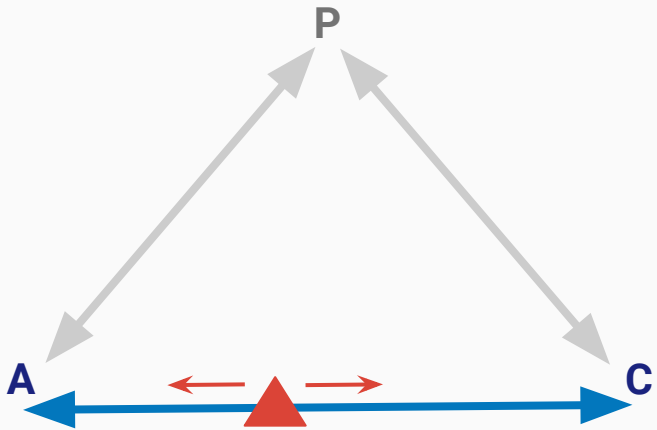


CAP (Theorem)

Any distributed system can only guarantee 2 out of 3 from **Consistency, Availability** and **Partition tolerance**

1. **Consistency:** read returns the most recent write
2. **Availability:** every request gets a response (maybe not latest write)
3. **Partition tolerance:** system keeps working even though some messages are lost

- Interpretation can be complicated as *partitions* will happen but rarely
- It sounds “binary”, but it’s continuous !
- It determines **design decisions** that can vary depending on data or use cases



Recommendation: *Forget about CAP, instead reason about trade-offs between A and C !*

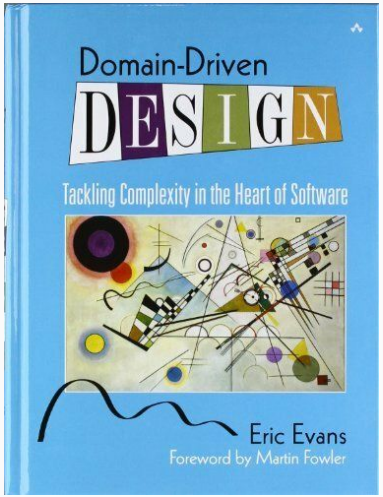
Domain-Driven Design

Domain-Driven Design, Reactive Architecture and Microservices

- Domain-Driven Design and Reactive Architecture are often used together because they are very compatible

- DDD main goal
  - Breaking down a large domain model into smaller pieces
  - Determine boundaries between different smaller domain
  - Define a good communication channel between domain experts and software engineers

- Reactive Microservices also try to define clear boundaries and roles



Different Approaches to Domain-Driven Design

Defining the Domain using the domain expert language

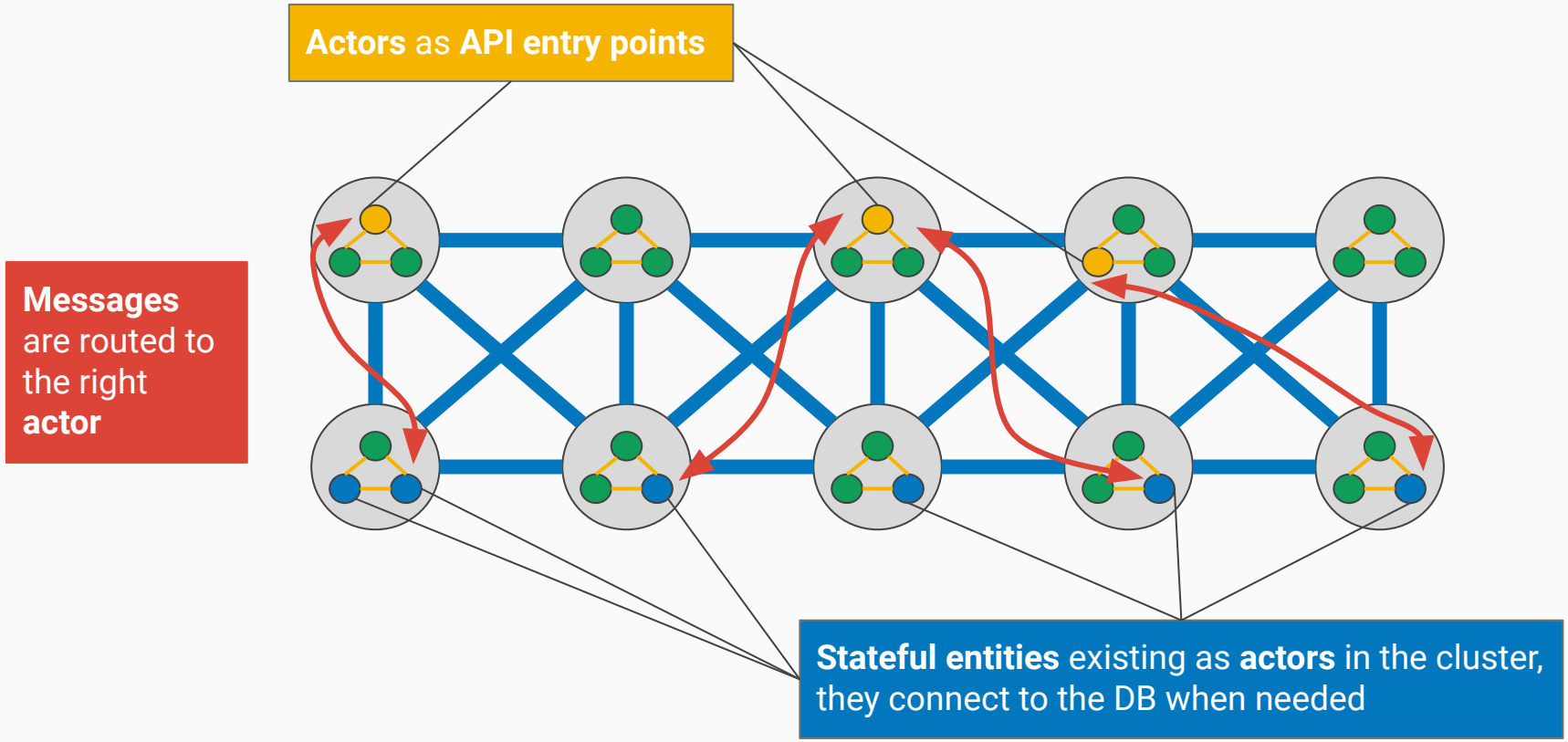
Finding subdomains

Boundaries around the domain and subdomains

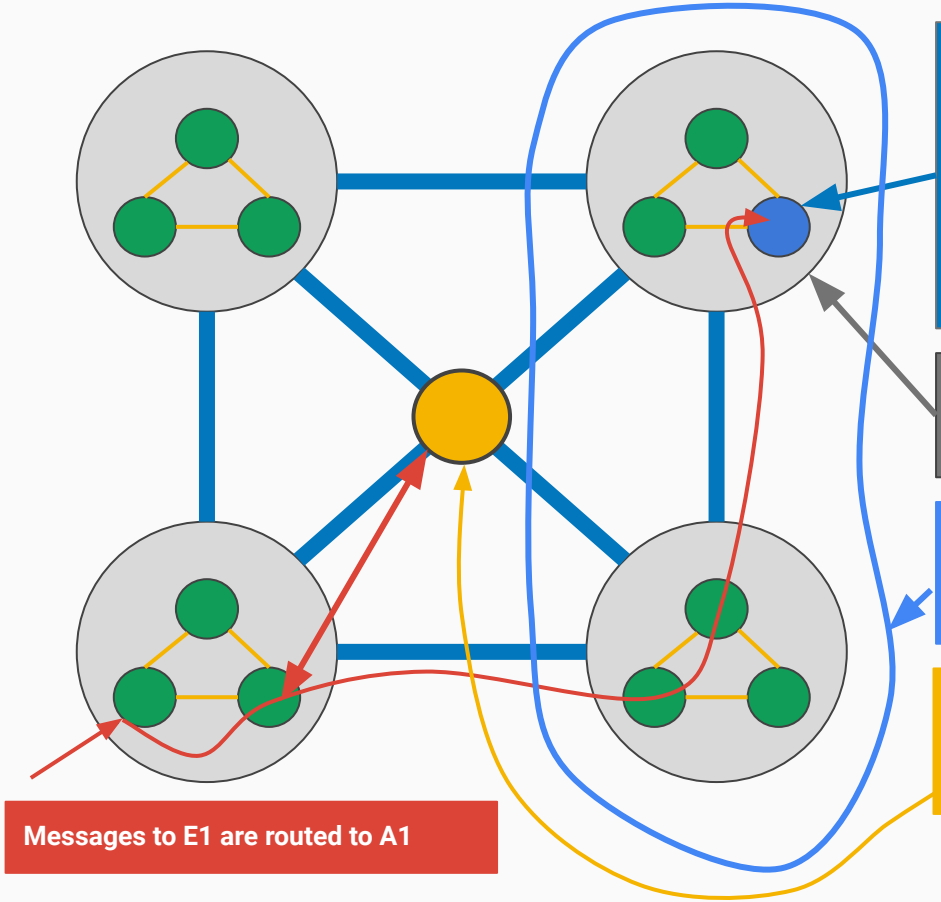
Analyze the Domain from an Event DDD perspective using the **Subject-Verb-Object Notation**

Stateful Applications through Sharding

Cluster for Stateful Application



Akka Sharding for Stateful Applications



One Actor = one DDD Entity
Actor becomes a *Consistency* boundary for its data, enabling *Strong Consistency*
Unique *Entity Id* through out the cluster
Messages are addressed to the Entity Id

Actor A1 managing Entity E1

Entity Actors are grouped in Shards
A *function* gives the *Shard Id* (usually based on the *Entity Id*)

Shards are grouped in Shard Regions
Typically, one Shard Region per JVM for an Entity type

Shard Coordinator, Singleton for message routing to the right Shards

Messages to E1 are routed to A1

Sharding: Routing, Entities, Shards

- **Entities Ids** must be unique throughout the cluster
- A message to an entity contains its **Id** to enable routing

- A common pattern is an **Envelope** for a message:

```
case class EntityEnvelope(entityId: String, message: Any)
```

- An *Extractor* function enables to extract the **Entity Id**

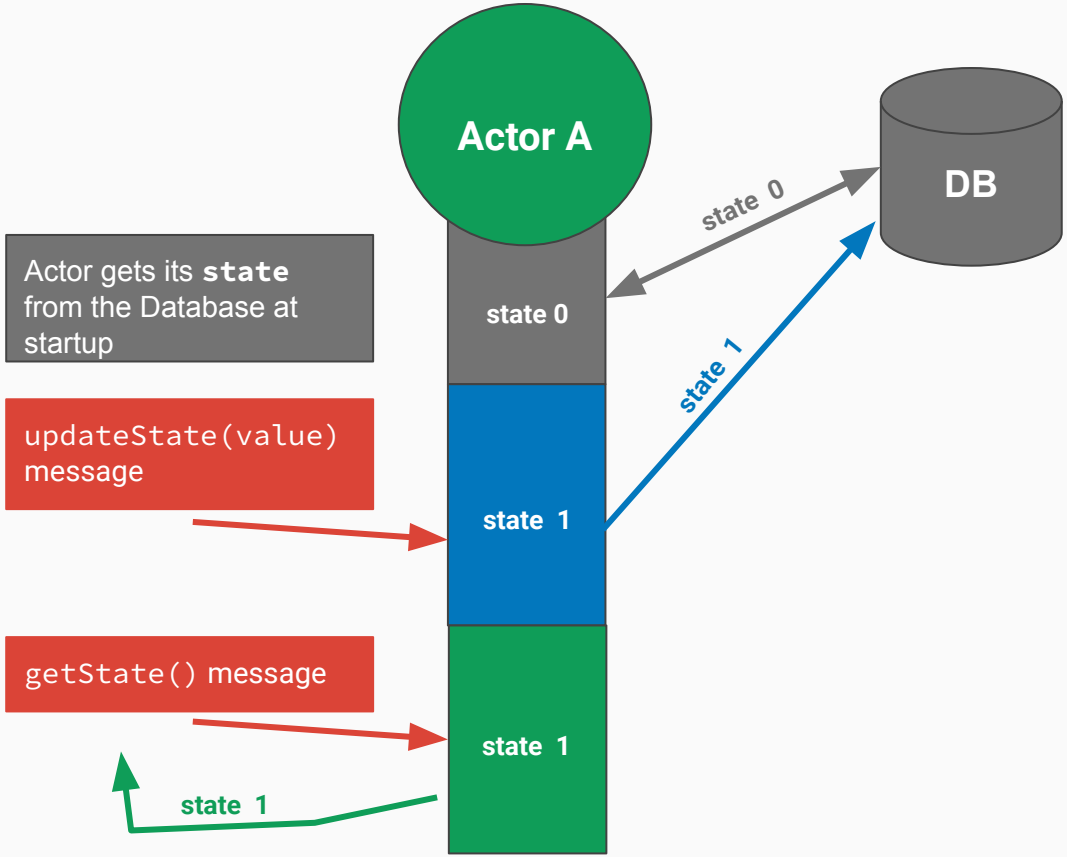
- To distribute **Entities** in **Shards**, a function needs to map **Entity Ids to Shard Ids**

- Example:

```
Math.abs(entityId.hashCode % nbrOfShards)).toString()
```

Akka Sharding

- Ensures **strong Consistency** with **Single source of Truth** (one Actor, one Entity)
- State is in the Actor, sync with the DB happens from the Actor
- Single threaded illusion (from Akka Actor)
- Contention problems can be mitigated in increasing the number of nodes or changing cluster topology
- Failures are isolated to actors or actor Systems



Akka Sharding Features

Blocking

- Blocking calls can be wrapped into Futures
- Messages can be stashed (e.g. during processing of non-blocking database calls)
- Stashed messages are kept while an actor is restarting

Passivation

- Automatically stop and removes Actor after a defined period without a message
- Messages are buffered during passivation
- Can be done manually (`Passivate` message)
- Passivation time can be optimized looking at memory footprint and/or actor being active

Rebalancing

- **Rebalancing** happens when the number of nodes in a cluster is changed (failure, scaling)
- **The Shard Coordinator** initiates the rebalancing, redistributing the shards among nodes
- The cluster must be in an healthy state (no **Unreachable** nodes)
- During Rebalancing, messages to **re-balanced** shards are buffered

Akka Cluster Miscellaneous

- **Coordinated shutdown**
  - Enables clean and stepwise shutdown of an ActorSystem

- **Akka Management**
  - **Akka Discovery**
    - Endpoint lookup delegation using some kind of discovery service (e.g. DNS)
  - **Akka Cluster Bootstrap**
  - **Cluster HTTP Management**
    - Extension library providing a REST API to query and manage an Akka cluster
    - Enables Health checks queries

- **Akka Serverless**
  - Fully managed Akka Stateful application environment

- **Lightbend Subscription**
  - Tools (e.g. Telemetry) and support directly from the Akka team

Akka Cluster Miscellaneous

- **Serialization**
  - **Akka** for its internal messages uses **protobuf**
  - **Jackson** is a good choice but any other library or own code can be used
  - Obvious: Java serialization **should be avoided** !!

- **Event-Sourcing**
  - Storing events, not state
  - State is reconstructed from stored events

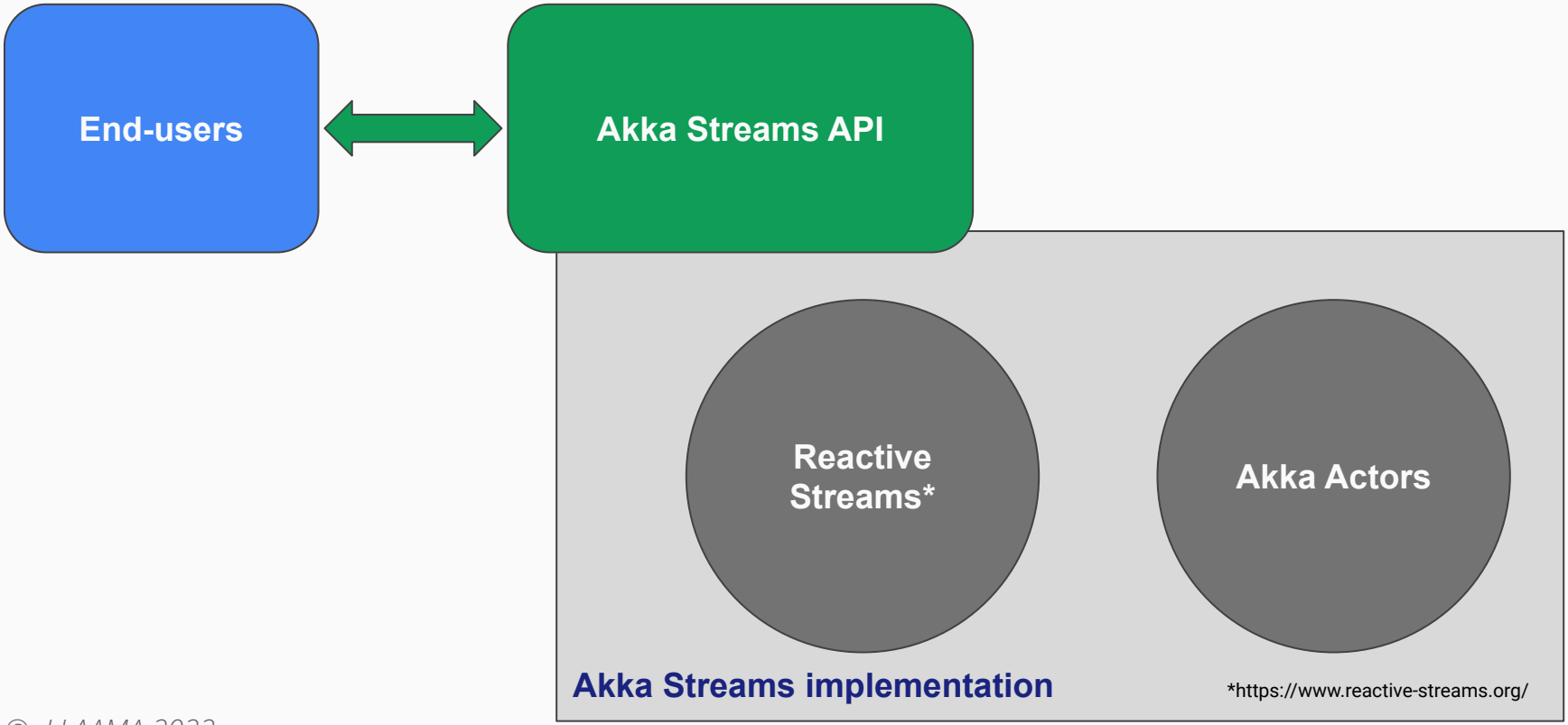
- **Command Query Responsibility Segregation (CQRS)**
  - Fits well with ES, but is also possible without it

Akka Streams

Introduction

- **Why?**
  - Some real big data can only be processed as streams
  - Always more data only exist as streams
- **How?**
  - Reactive Streams Specification (www.reactive-streams.org)
  - Asynchronous streams with back-pressure
  - Constructs: publisher | processor | subscriber
  - ***Reactive streams*** is a Service Provider Interface (SPI)
  - included in JDK9
  - **Akka Streams is a friendly user API** that can use Reactive Streams Interfaces

Akka Streams big picture

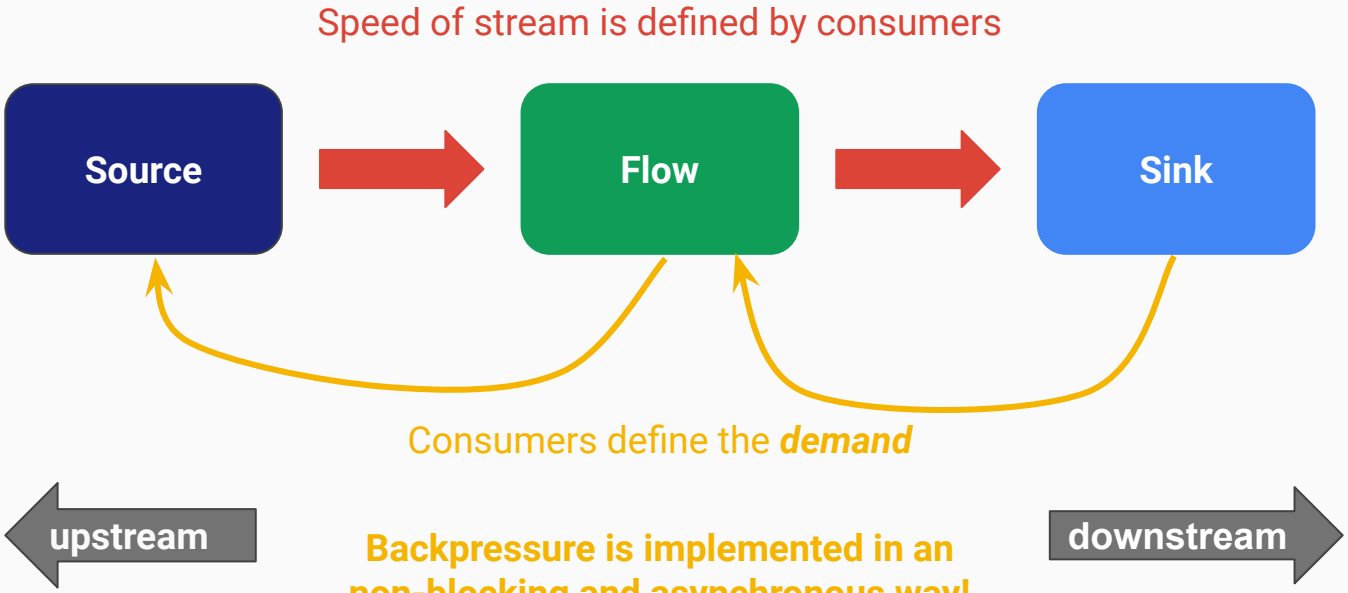


Akka Streams

- **`Source`:** *produces* elements asynchronously
- **`Sink`:** *receives* elements
- **Flow:** *processes* elements (transformer)
- **Back pressure** included

- From **`Source`** to **Flow** to **`Sink`**
- Blueprints at every level

Akka Streams, Backpressure



From **`Source`** to **`Sink`**

`source.via(flow)` is a source

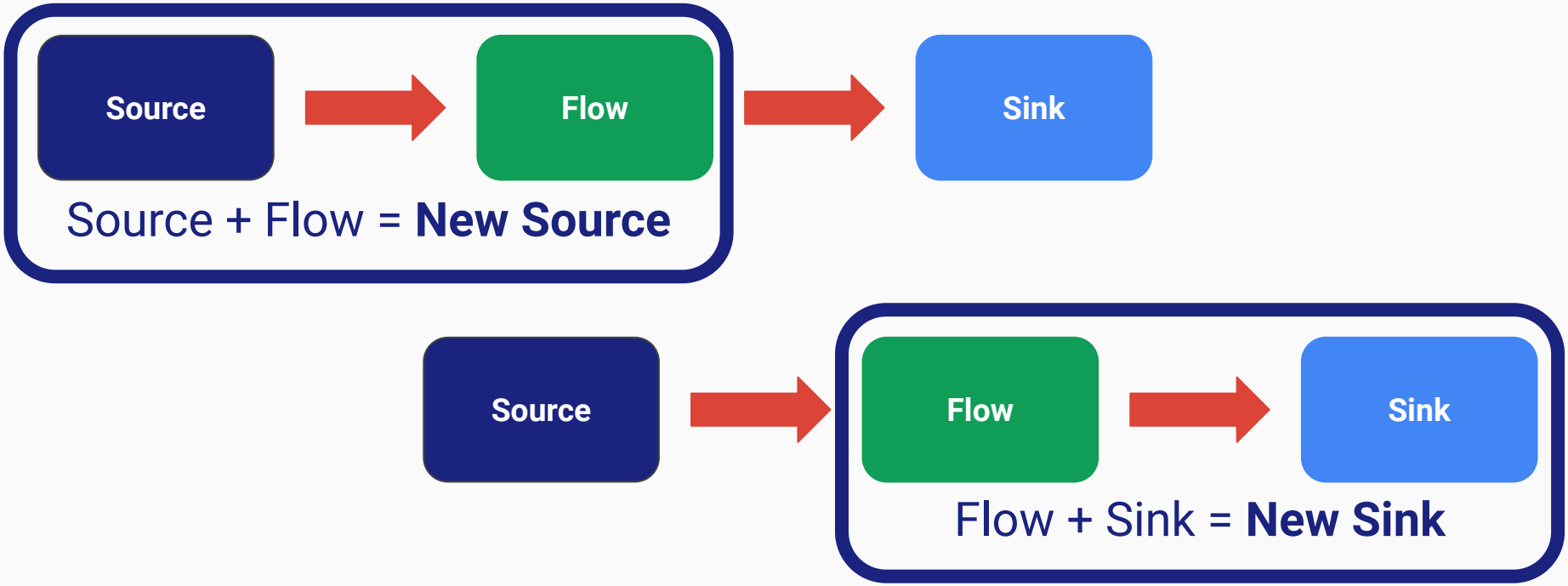
`flow.to(sink)` is a sink

Blue prints are executed with **`.run()`** which needs an ActorSystem for materialization

Nothing happens with the graph blueprint until materialization (**run**)

`Nulls` are not allowed!

`Source, Flow, Sink:` Lego blocks



Materializing

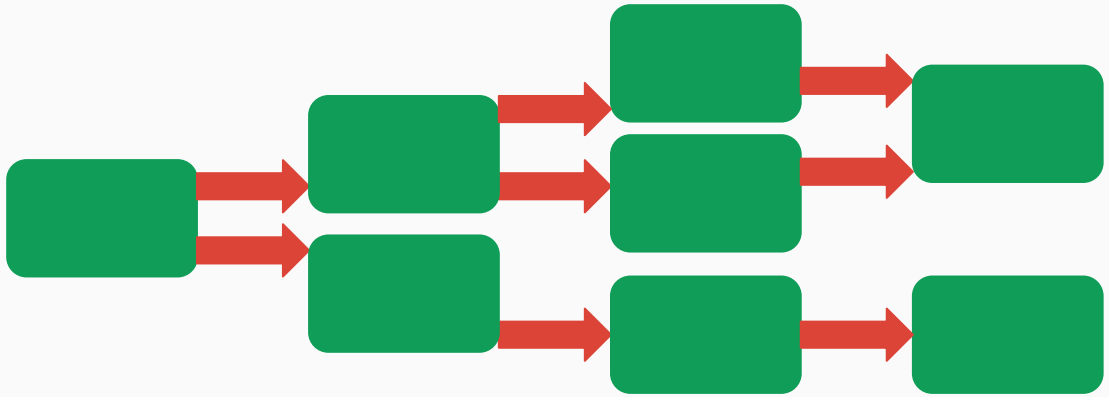
- A graph element can be used multiple times (think *blueprint*)
- Graph component are static until **run** is called
- Every **run** is a different **materialization**
- **Run** result is a **materialized value, running graph = materializing**
  - Resources allocations for a blueprint happens at ***materialization***
    - Actors, threads, connections, etc.
- For every component, running => producing a **materialized value**
- But the graph produces only one **materialized value**

Materialized values can be everything from **nothing** to any **object**!

Graphs

Graph DSL

- Fan-out
  - Broadcast
  - Balance
- Fan-in
  - Zip
  - ZipWith
  - Merge
  - Concat



The End