

Building Stateful Microservices With Akka

Yaroslav Tkachenko

Senior Software Engineer at Demonware (Activision)

About me

Yaroslav (Slava) Tkachenko, Vancouver, Canada



- Java, Scala, Python, Node
- Microservices
- Event-driven Systems
- Distributed Systems
- DevOps
- ... [and more](#)

Demonware (Activision), 2017

- Senior Software Engineer [*Data Pipeline*]

Mobify, 2016 - 2017

- Senior Software Engineer, Lead [*Platform*]

Bench Accounting, 2011 - 2016

- Director of Engineering [*Platform*]
- Engineering Lead
- Software Engineer

Freelance, 2007 - 2011

- Web Developer

<https://sap1ens.com/slides/stateful-services/>

Agenda

- Microservices: stateless vs stateful
- Actor systems
- Akka
- Akka Cluster and Persistence
- Real-world applications

Microservices: stateless vs stateful

Microservices: stateless vs stateful

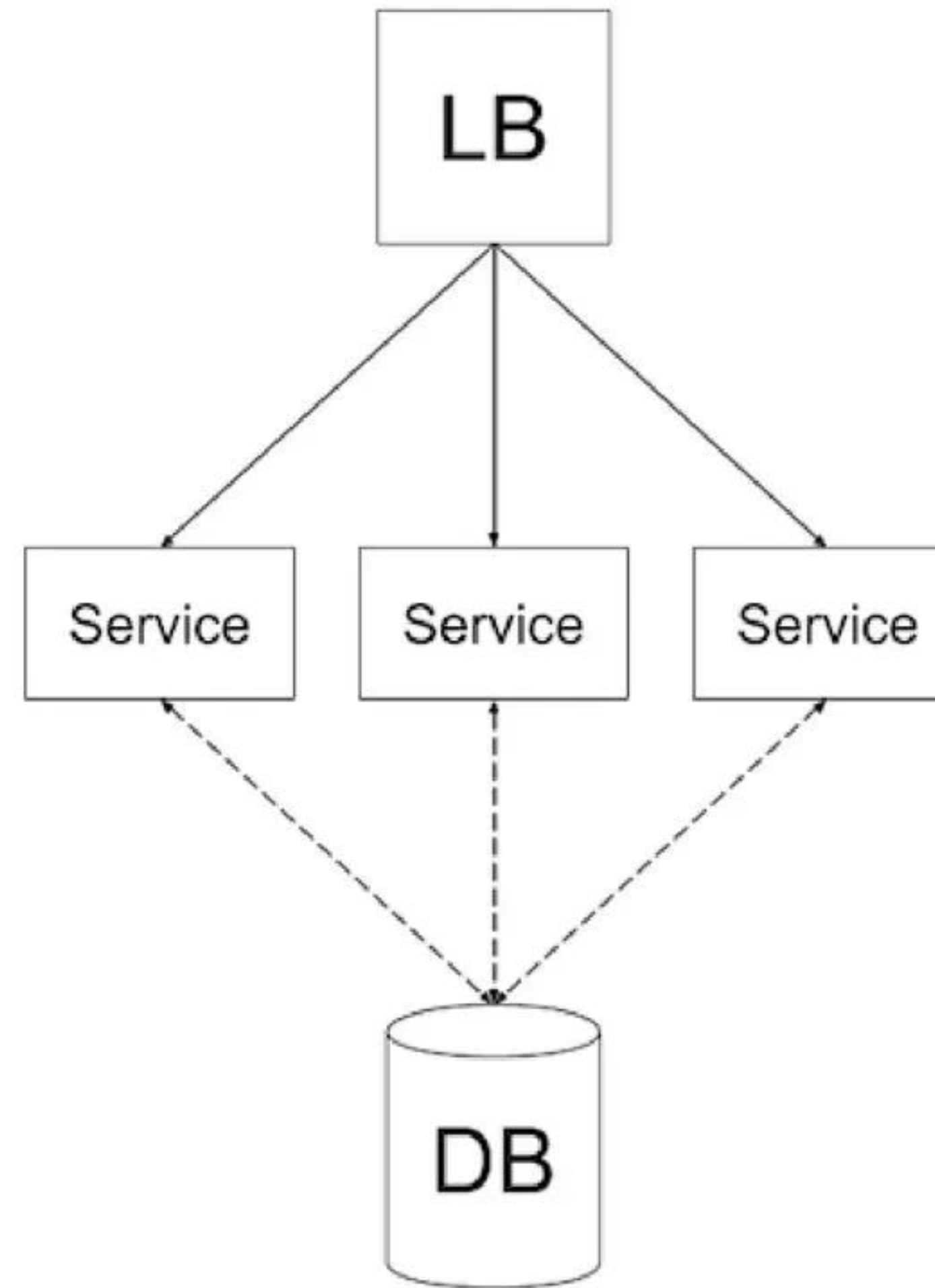
Stateless application: application that doesn't keep any state in *memory / runtime*, but uses external services instead.

- External service: database, cache, API, etc.
- Examples: most of the web apps are stateless or designed to be stateless (Spring, Django, Rails, Express, etc.).

Stateful application: application that keeps internal state in *memory / runtime*, instead of relying on external services.

- Examples: actors *can* be stateful, so Akka and other actor-based systems (Erlang/OTP, Orleans) can be stateful. But it's also possible to create stateful applications in Node.js or Python, for example.

Microservices: stateless



Microservices: stateless

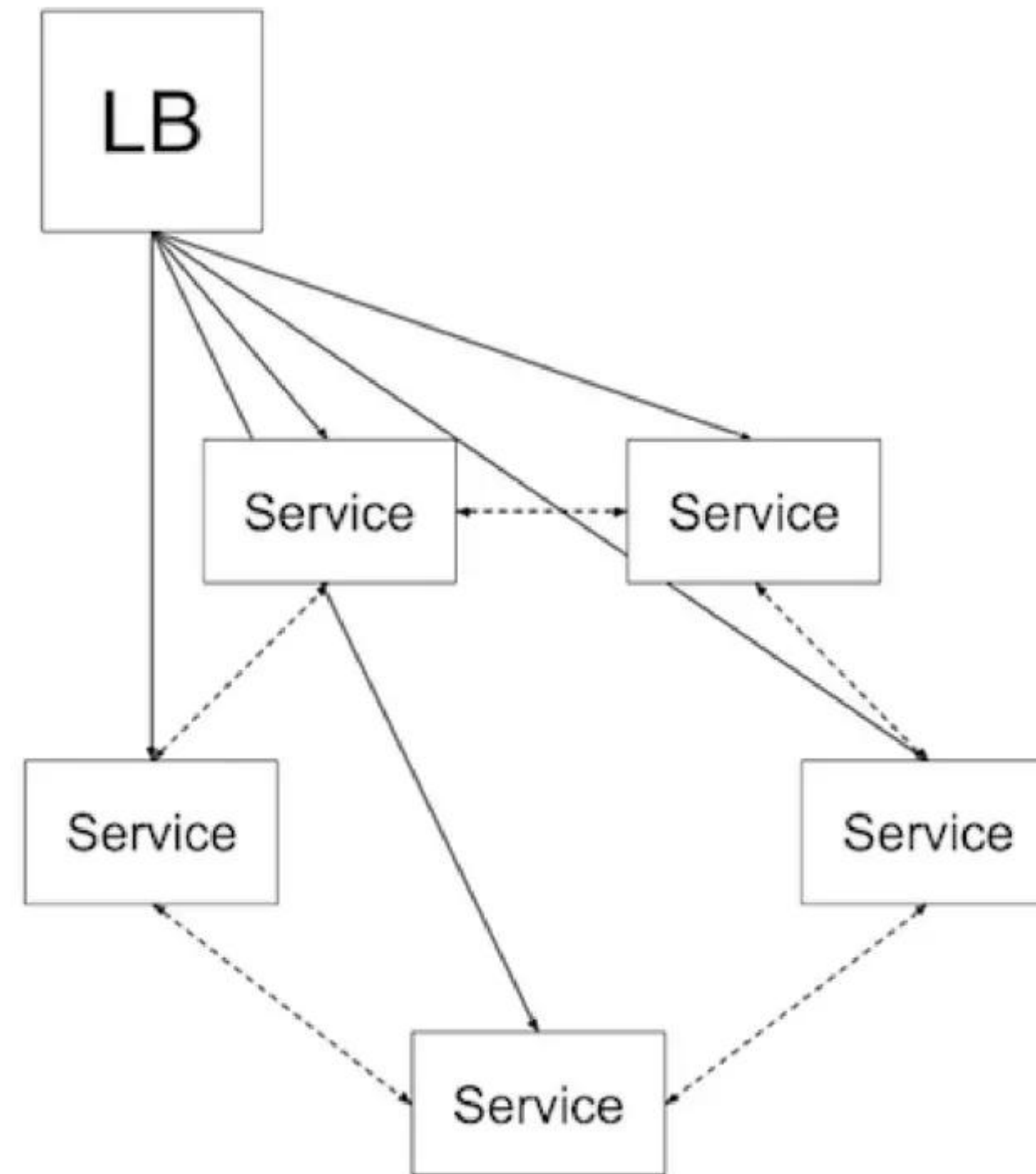
Benefits:

- Simple development & deployment
- Simple to scale out -> just add more nodes

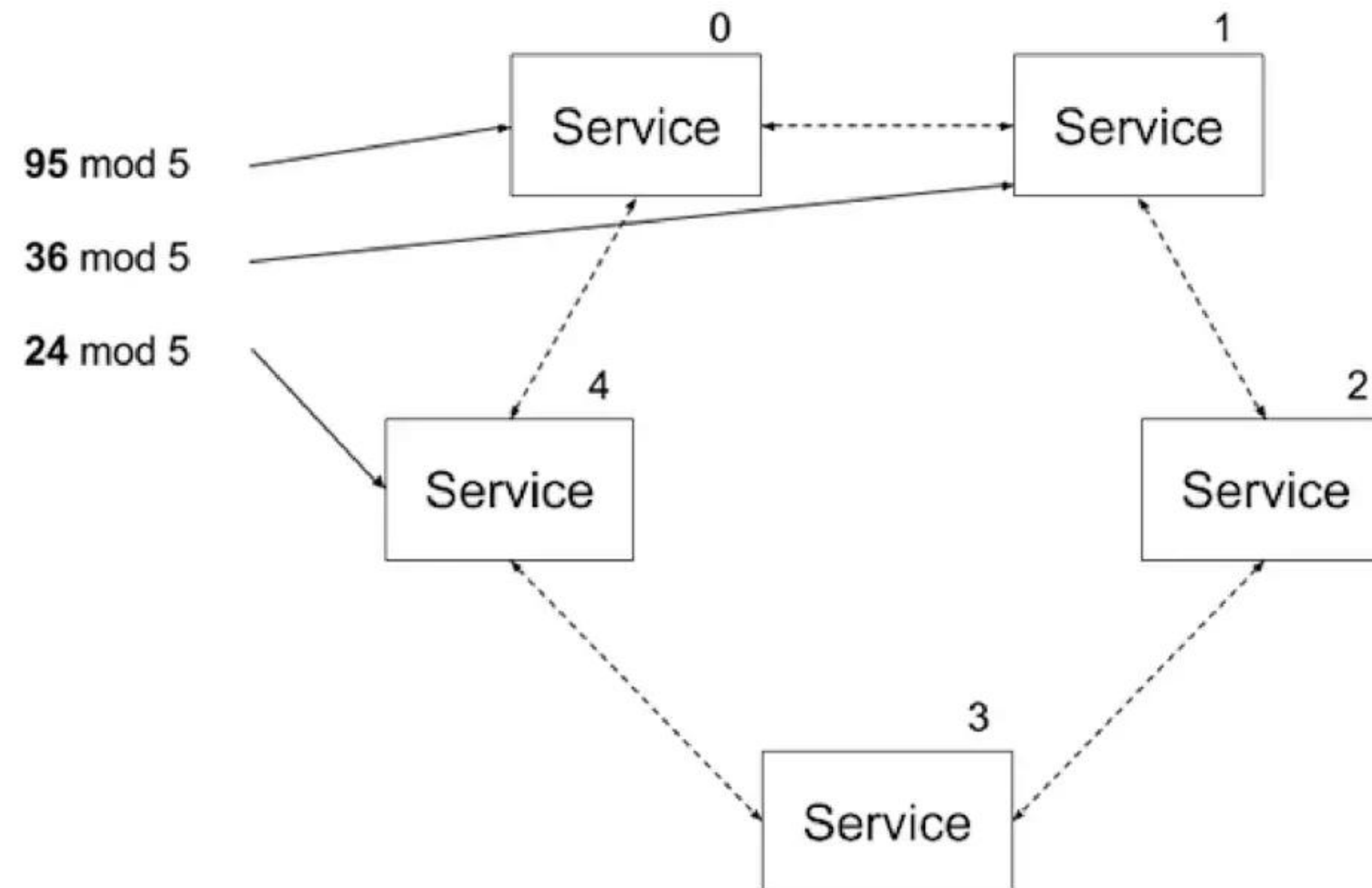
Biggest challenges:

- Low latency -> can use caching, but not when strong consistency is needed
- Concurrent modifications -> conflict resolution with optimistic / pessimistic locking

Microservices: stateful



Microservices: stateful



Microservices: stateful

Benefits:

- Data locality -> low latency, fast processing
- Sticky consistency -> "simple" and "cheap" consistency without using consensus protocols

Biggest challenges:

- High availability
- Scaling out

Actor systems

Actor systems

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 behavior to be used for the next message it receives.

There is no assumed sequence to the above actions and they could be carried out in parallel.

Every actor has:

- A mailbox
- A supervisor
- Some state [optionally]

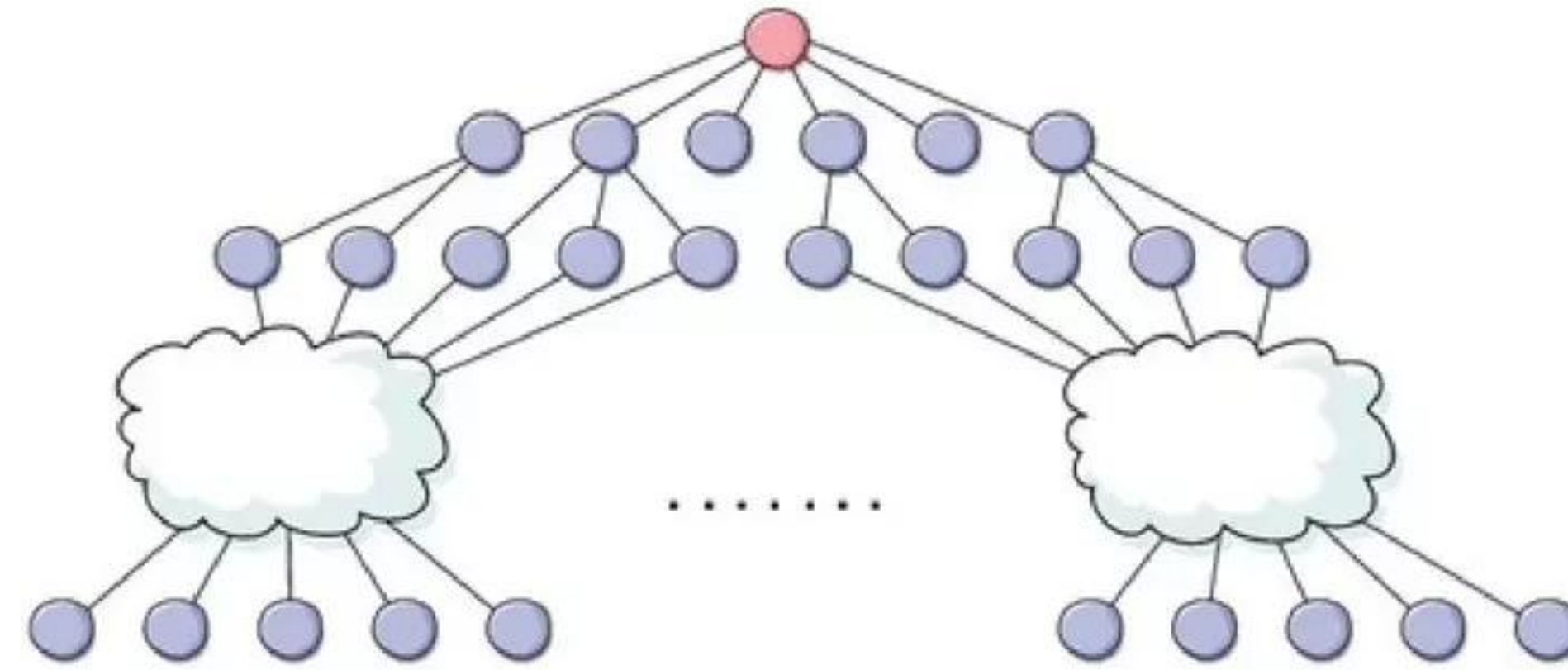
Actor systems - Examples

 **Rachel**

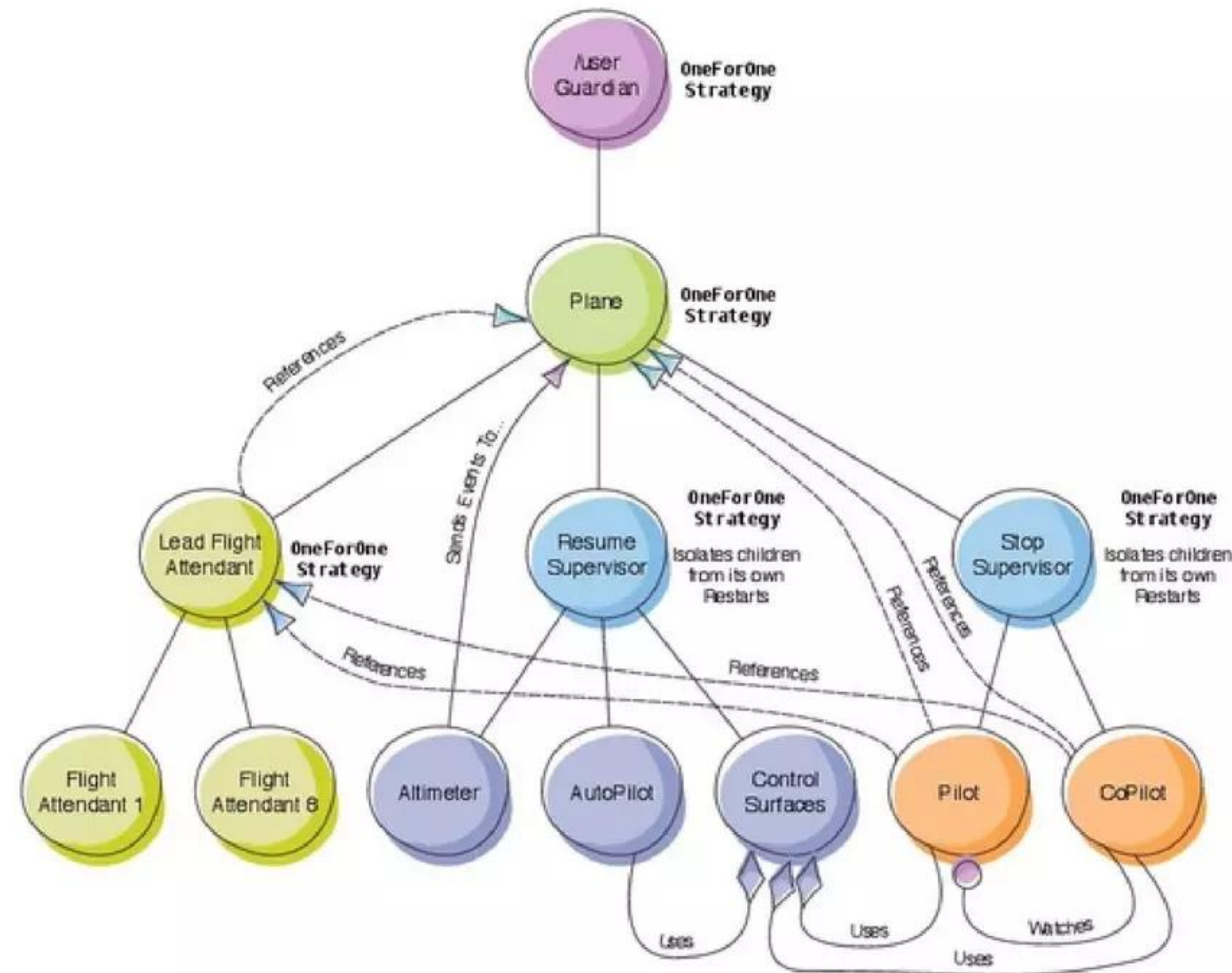
 **Alex**

 **Fred**

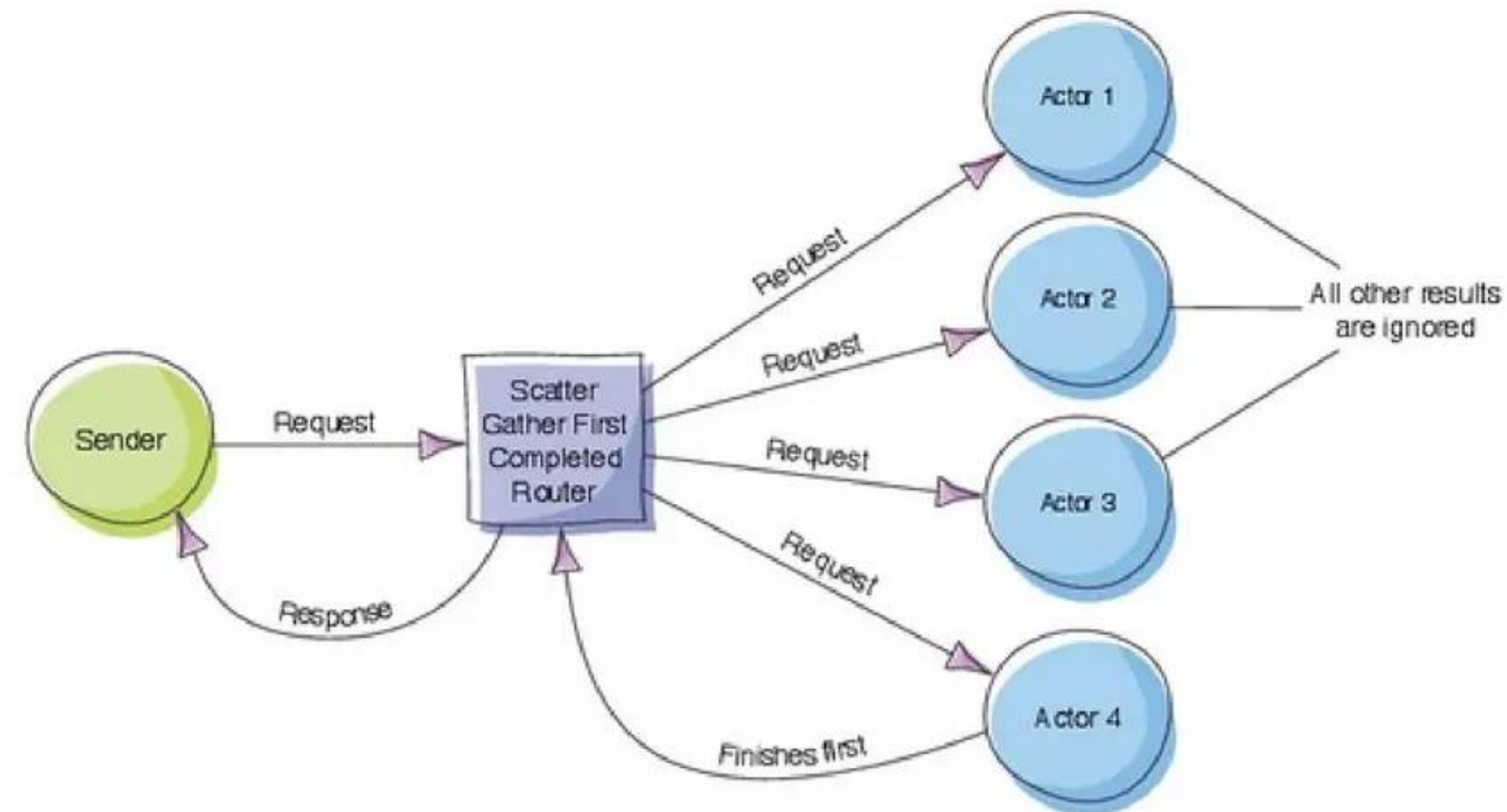
Actor systems - Examples



Actor systems - Examples



Actor systems - Examples



Akka

Akka

Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed applications on the JVM.

Akka supports multiple programming models for concurrency, but it emphasizes actor-based concurrency, with inspiration drawn from Erlang.

Akka - Actors

```
case class Greeting(who: String)

class GreetingActor extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[GreetingActor], name = "greeter")
greeter ! Greeting("Charlie Parker")
```

- Messages are handled one by one
- Immutability of messages

Akka - Communication

```
class HelloActor extends Actor with ActorLogging {  
  def receive = {  
    case who => sender() ! "Hello, " + who  
  }  
}  
  
object ConversationActor {  
  def props(fellowActor: ActorRef): Props = Props(classOf[ConversationActor], fellowActor)  
}  
  
class ConversationActor(fellowActor: ActorRef) extends Actor with ActorLogging {  
  def receive = {  
    case "start" => fellowActor ! "it's me!"  
    case message => log.info(message)  
  }  
}  
  
val system = ActorSystem("MySystem")  
  
val helloActor = system.actorOf(Props[HelloActor])  
val conversationActor = ConversationActor.props(helloActor)  
  
conversationActor ! "start"
```

Actor systems and Akka - Why?

So, why actors?

- Simple concurrency
- Clean asynchronous programming model
- Great fit for event-driven systems
- Resilience
- Scalability

Akka Persistence

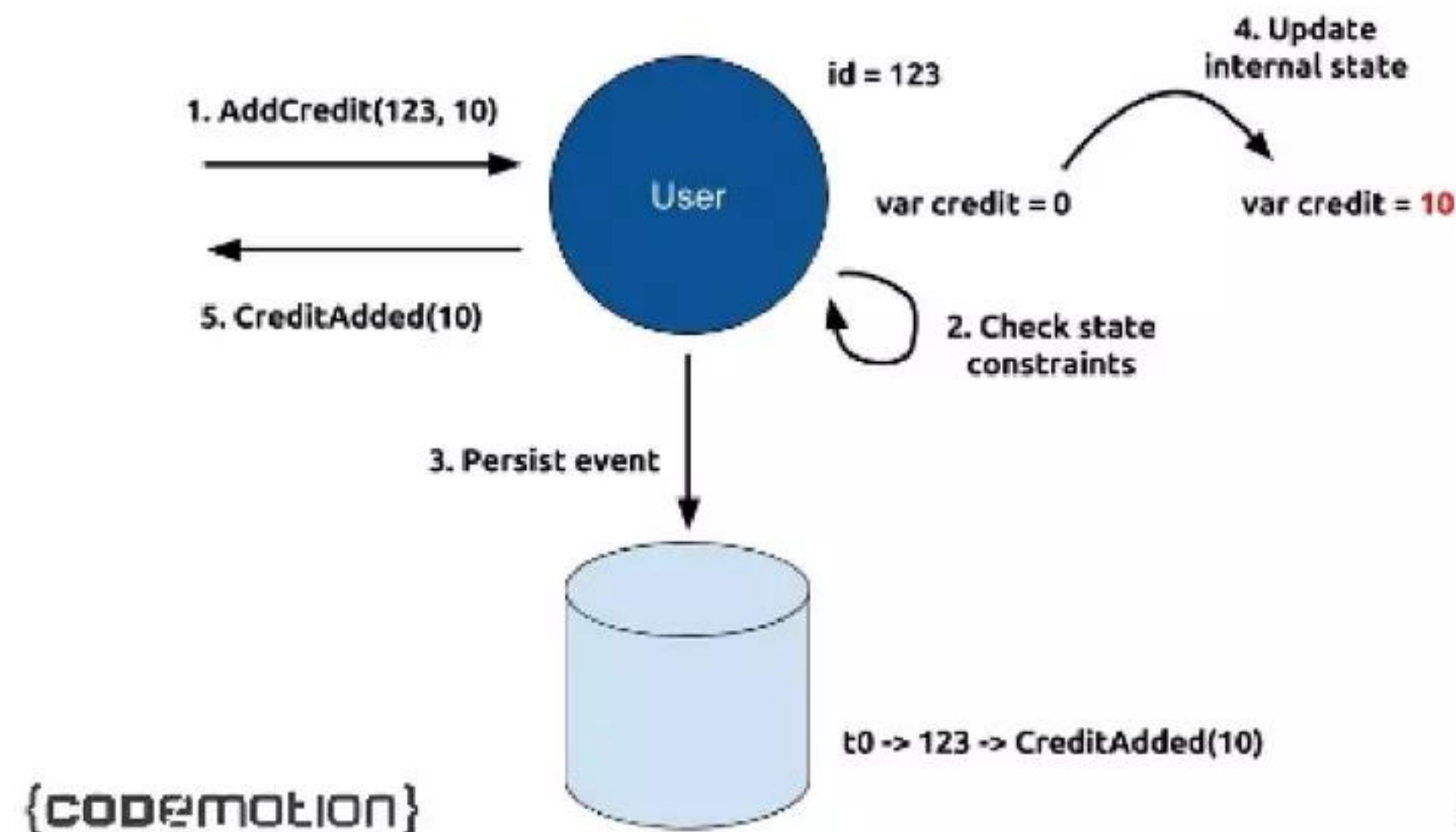
Akka Persistence - Overview



Akka persistence

```
case class AddCredit(user: Id[User], amount: Double) extends Command[User] { val to = user }
```

```
case class CreditAdded(amount: Double) extends Event
```



MADRID · NOV 18-19 · 2016

Akka Persistence - Overview

- Event Sourcing
- Persistent Actor
- Journal
- Snapshot

Has plugins for JDBC (MySQL, Postgres, ...), MongoDB, Cassandra, Kafka, Redis and more.

Akka Persistence - Example

```
case class Cmd(data: String)
case class Evt(data: String)

case class ExampleState(events: List[String] = Nil) {
  def updated(evt: Evt): ExampleState = copy(evt.data :: events)
  override def toString: String = events.reverse.toString
}

class ExamplePersistentActor extends PersistentActor {
  override def persistenceId = "sample-id-1"

  var state = ExampleState()

  def updateState(event: Evt): Unit =
    state = state.updated(event)

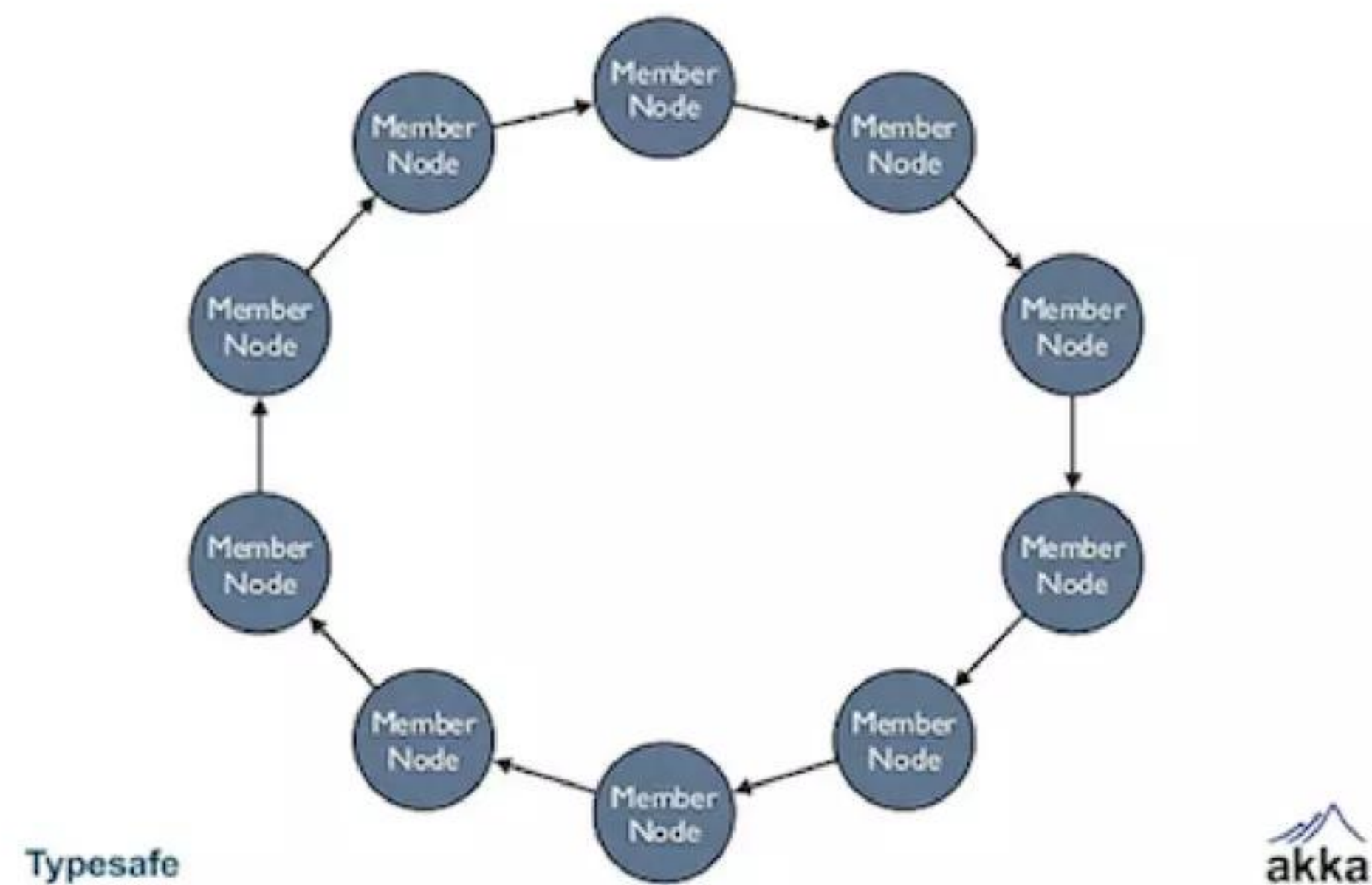
  val receiveRecover: Receive = {
    case evt: Evt => updateState(evt)
    case SnapshotOffer(_, snapshot: ExampleState) => state = snapshot
  }

  val receiveCommand: Receive = {
    case Cmd(data) => persist(Evt(data))(updateState)
    case "snap" => saveSnapshot(state)
    case "print" => println(state)
  }
}
```

Akka Cluster

Akka Cluster - Overview

Node ring with gossiping members



- Cluster
- Node

- Gossip protocol
- Failure Detector

Akka Cluster - Sharding

Features:

- One of the most powerful Akka features!
- Allows to route messages across nodes in a cluster using a sharding function (actually two)
- You don't need to know the physical location of an actor - cluster will forward message to a remote node if needed
- Uses Akka Persistence internally (or brand-new Distributed Data)

Concepts:

- Coordinator
- Shard Region
- Shard
- Entity

Entities (actors) are "activated" by receiving a first message and can be "passivated" using `context.setReceiveTimeout`.

Akka Cluster - Sharding

Counter interface:

```
case object Increment
case object Decrement
final case class Get(counterId: Long)
final case class EntityEnvelope(id: Long, payload: Any)

case object Stop
final case class CounterChanged(delta: Int)
```

Akka Cluster - Sharding

Counter implementation:

```
class Counter extends PersistentActor {  
  context.setReceiveTimeout(120.seconds)  
  
  override def persistenceId: String = "Counter-" + self.path.name  
  
  var count = 0  
  
  def updateState(event: CounterChanged): Unit =  
    count += event.delta  
  
  override def receiveRecover: Receive = {  
    case evt: CounterChanged ⇒ updateState(evt)  
  }  
  
  override def receiveCommand: Receive = {  
    case Increment      ⇒ persist(CounterChanged(+1))(updateState)  
    case Decrement      ⇒ persist(CounterChanged(-1))(updateState)  
    case Get(_)         ⇒ sender() ! count  
    case ReceiveTimeout ⇒ context.parent ! Passivate(stopMessage = Stop)  
    case Stop           ⇒ context.stop(self)  
  }  
}
```

Akka Cluster - Sharding

Create a region on every node:

```
val counterRegion: ActorRef = ClusterSharding(system).start(
  typeName = "Counter",
  entityProps = Props[Counter],
  settings = ClusterShardingSettings(system),
  extractEntityId = extractEntityId,
  extractShardId = extractShardId)
```

Sharding functions:

```
val extractEntityId: ShardRegion.ExtractEntityId = {
  case EntityEnvelope(id, payload) ⇒ (id.toString, payload)
  case msg @ Get(id)              ⇒ (id.toString, msg)
}

val numberOfShards = 100

val extractShardId: ShardRegion.ExtractShardId = {
  case EntityEnvelope(id, _) ⇒ (id % numberOfShards).toString
  case Get(id)               ⇒ (id % numberOfShards).toString
}
```

Akka Cluster Sharding + Persistence = ❤️

Akka Cluster Sharding:

- Consistent hashing for all requests based on user-defined function
- Automatic forwarding (from local to remote and vice versa)

Akka Persistence:

- Keeping internal state
- Easy and fast recovery (journal + snapshots)
- Event-sourcing built-in

Real-world applications

Real-world applications

- Complex event-driven state machine with low latency API (aka **The Tracker**)
- More (online gaming, data aggregation, trading, complex domains, ...)

Real-world applications - The Tracker

Complex event-driven state machine:

Consuming:

- Domain Events via messaging queue (Akka Camel)

Interface for querying:

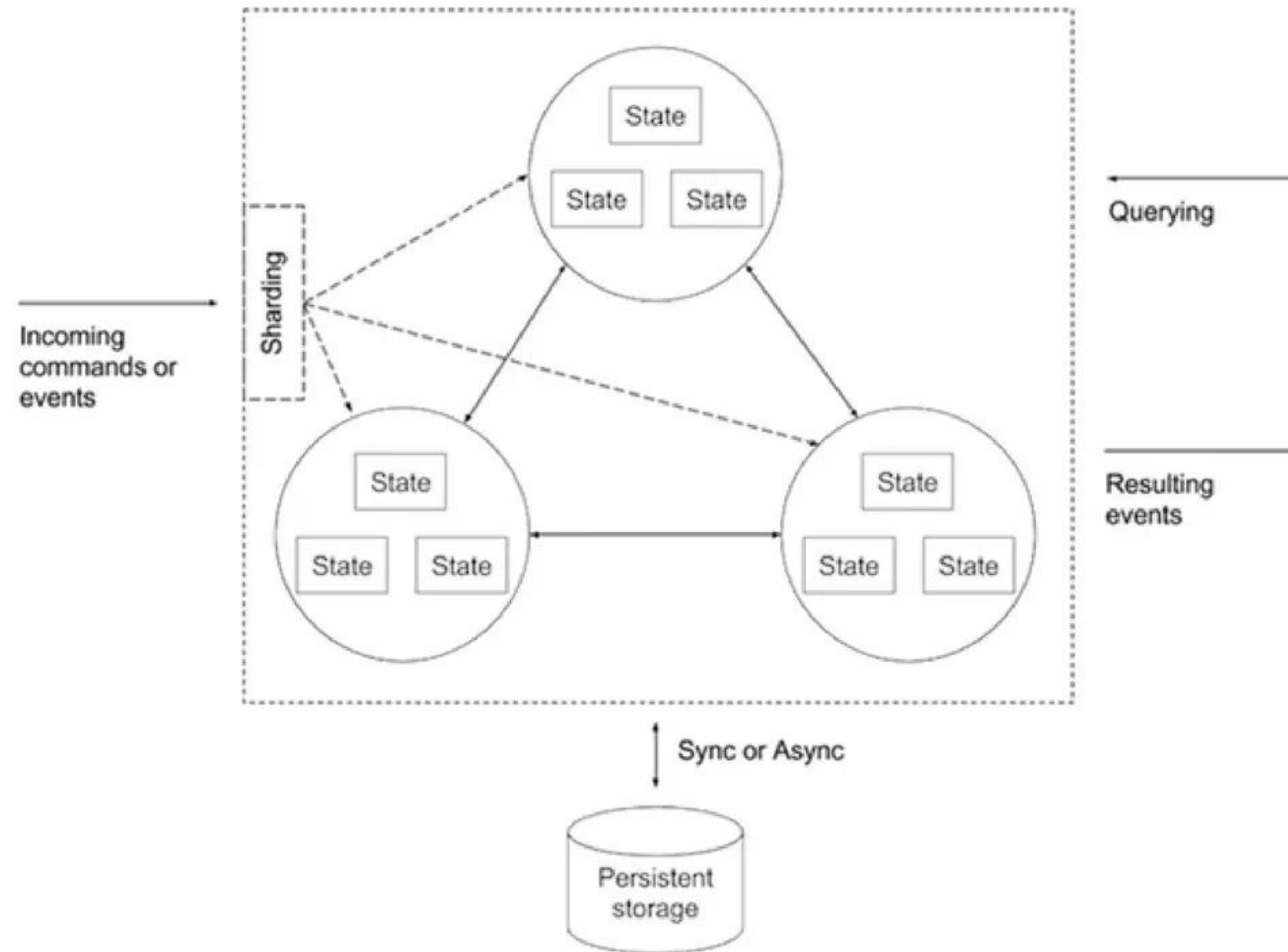
- HTTP API (Akka HTTP)
- Websockets (Akka HTTP)

Every entity has a **clientId** and they never intersect - it's a perfect use-case for sharding (clientId as a sharding key).

Real-world applications - The Tracker

```
object TrackerService {  
  case class TrackerData(  
    accounts: Map[String, BankAccount] = Map[String, BankAccount]()  
  )  
}  
  
class TrackerService extends PersistentActor {  
  private var state = TrackerData()  
  
  private def handleMessage(message: EventMessage) {  
    val maybeUpdatedState = message match {  
      case b: BankAccountMessage => handleBankMessage(b)  
      case c: ClientMessage => handleClientMessage(c)  
      case _ => None  
    }  
    maybeUpdatedState.foreach { updatedState =>  
      updateState(updatedState)  
    }  
  }  
  
  private def updateState(updatedState: TrackerData) = {  
    state = state.copy(  
      accounts = (state.accounts ++ updatedState.accounts).filterNot(_._2.deleted)  
    )  
  }  
}
```

Real-world applications



Summary

- Actor-based programming simplifies building highly scalable and reliable systems
- It's not easy to build & maintain a stateful application, but you never know when it's going to be needed
- Don't try to write abstractions for distributed programming from scratch (unless you're an expert)
- Akka has a few great abstractions already, use them!
- It's easier to build a stateful application as a microservice - smaller state size, more flexibility and great separation of concerns

Questions?

@sap1ens