One year of Akka 2.6

Sean Glover, Lightbend
@seg1o

Lightbend

I am Sean Glover

Principal Engineer at Lightbend

Member of the Akka engineering team

Herder of the Alpakka project

Organizer of Scala Toronto (scalator)

OSS maintainer/contributor in the Akka and Kafka ecosystems

/ seg1o

Some major features since Akka 2.6

Akka

- Akka Typed APIs marked stable
- Akka Persistence & Sharding APIs
- Artery default remoting layer
- Serialization with Jackson
- Separation of internals and default user dispatchers
- Drop need to provide Materializer to streams
- External shard allocation strategy
- Reliable delivery
- Sharded daemon process
- Distributed publish subscribe
- New persistence testkit
- Open sourced split brain resolver
- Replicated event sourcing AKA Multi-DC persistence
- Faster cluster sharding rebalance algorithms

Akka Projects

- Alpakka Kafka 2.0
- Akka Persistence Cassandra 1.0
- Akka Persistence JDBC 4.0
- Akka Http 10.2.0 using Akka 2.6
- Akka gRPC 1.0 (built on top of Akka-http)
- Akka Projections 1.0
- Akka Platform Guide

We'll only talk about these today 😉

Akka Actors [Typed]

Akka Actors Typed

- The Typed Actors API was introduced with Akka 2.4.0 in 2015
- Chiefly the creation of Dr. Roland Kuhn
- Since the release of Akka 2.6.0 the “old” Actors API is known as Classic Akka
- The Typed Actors API became the default for docs, reference projects, and new features

Akka Actors Typed

Define a message protocol

```
sealed trait Command
final case class Withdraw(amount: Long, ack: ActorRef[Response]) extends Command
final case class Deposit(amount: Long, ack: ActorRef[Done]) extends Command
final case class GetBalance(replyTo: ActorRef[Long]) extends Command

sealed trait Response
case object Ack extends Response
case object InsufficientFunds extends Response
```

Akka Actors Typed

Implement an Actor Behavior

```
def openAccount(balance: Long = 0L): Behavior[Command] = {
 Behaviors.receiveMessage {
   case Deposit(amount, ack) =>
     ack ! Done
     openAccount(balance + amount)
   case GetBalance(replyTo) =>
     replyTo ! balance
     Behaviors.same
   case Withdraw(amount, ack) =>
     ...
 }
}
```

Akka Actors Typed

State transition: Open Account => Frozen Account!

Append message protocol

```
sealed trait Command
...
final case class FreezeAccount(replyTo: ActorRef[Done]) extends Command

sealed trait Response
...
case object AccountFrozen extends Response
```

Akka Actors Typed

State transition: Transition to a Frozen Account Behavior

```
def openAccount(balance: Long = 0L): Behavior[Command] = {
 Behaviors.receiveMessage {
   ...
   case FreezeAccount(replyTo) =>
     replyTo ! Done
     frozenAccount(balance)
 }
}

def frozenAccount(balance: Long = 0L): Behavior[Command] = ...
```

Akka Actors Typed

State transition: Implement a new Behavior for frozen state

```
def frozenAccount(balance: Long = 0L): Behavior[Command] = {
 Behaviors.receiveMessage {
   case GetBalance(replyTo) =>
     replyTo ! balance
     Behaviors.same
   case Deposit(_, ack) =>
     ack ! AccountFrozen
     Behaviors.same
   case Withdraw(_, ack) =>
     ...
 }
}
```

Akka Persistence Typed

Akka Persistence & Event Sourcing

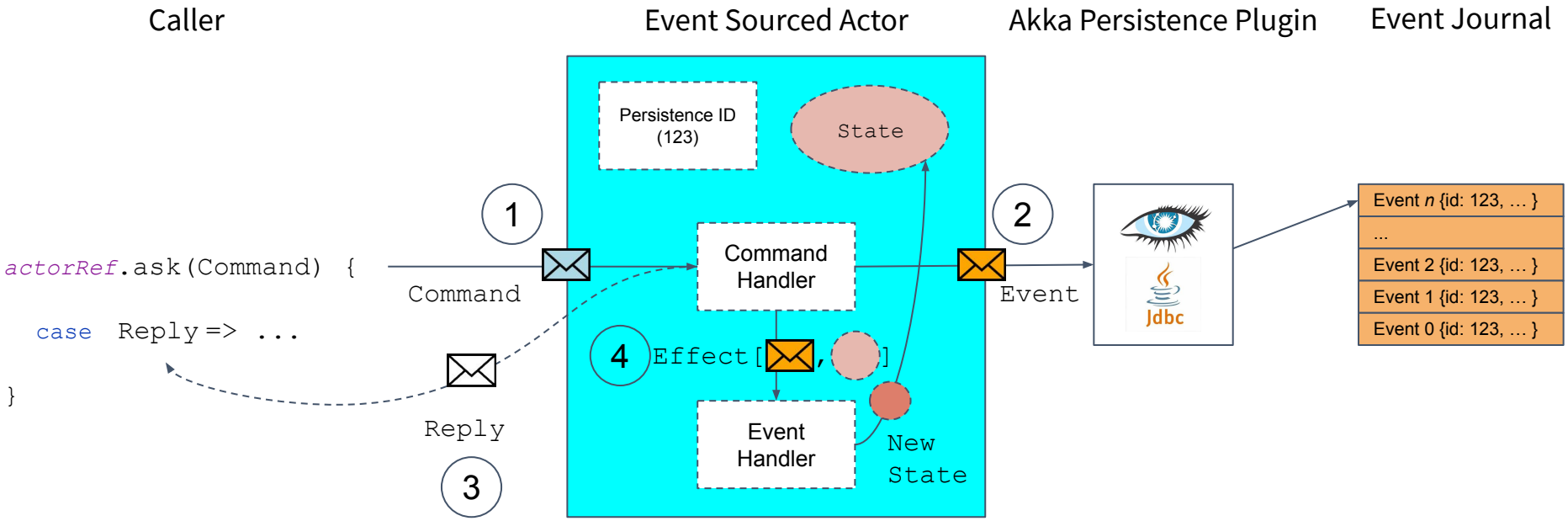
Expand message protocol to include

- `Command`
- `Event`
- `State` (optional)
- `Reply` (optional)

Ex)

```
final case class Deposit(amount: Long, replyTo: ActorRef[Reply]) extends Command
final case class Deposited(amount: Long) extends Event
final case class AccountState(balance: Long) extends State
case object Ack extends Reply
```

Akka Persistence & Event Sourcing



Akka Persistence & Event Sourcing

Usage

```
val actorRef = context.spawn[Command](
  persistedAccount("123")
)

actorRef.ask(Deposit(10, context.self)) {
 case Reply => ...
}
```

Implementation

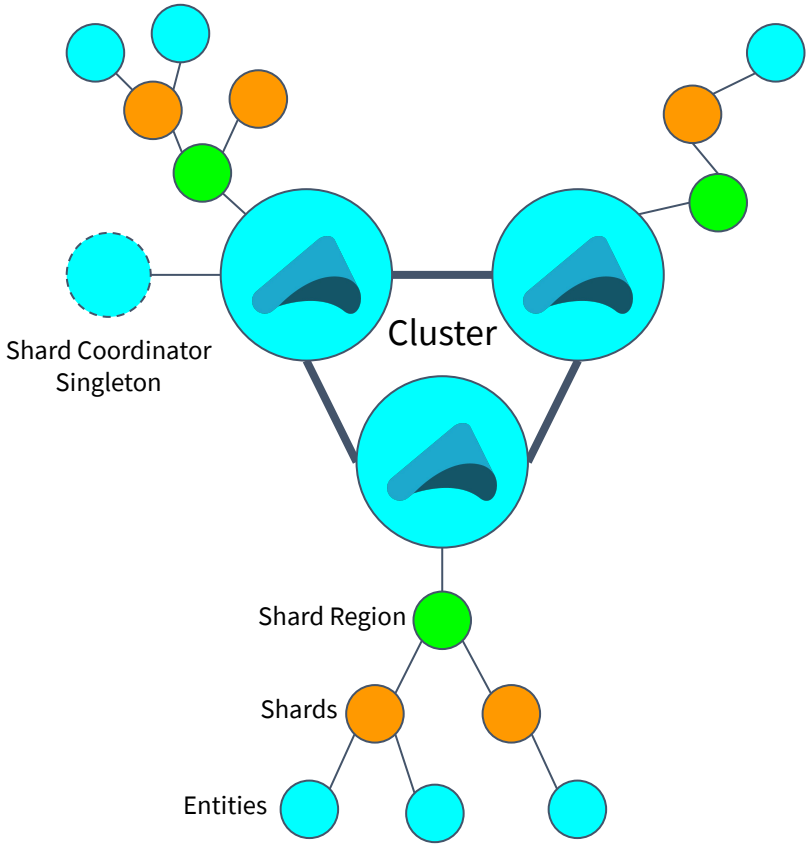
```
def persistedAccount(accountId: String): Behavior[Command] = {
 EventSourcedBehavior[Command, Event, AccountState](
   persistenceId = PersistenceId.ofUniqueId(accountId),
   emptyState = AccountState(balance = 0L),
   commandHandler = (state, command) => command match {
     case Deposit(amount, ackTo) =>
       Effect.persist(Deposited(amount)).thenRun { _ =>
         ackTo ! Ack
       }
   },
   eventHandler = (state, event) => state.applyEvent(event)
 )
}
```

Akka Cluster Sharding Typed

Akka Cluster Sharding

Scale, Consistency, and Failover

- Balance resources (memory, disk, network) across multiple nodes for scalability
- Distribute entities and data across cluster
- Location transparency: Interact with entities by logical id instead of physical address
- Automatic relocation on failure or rebalance

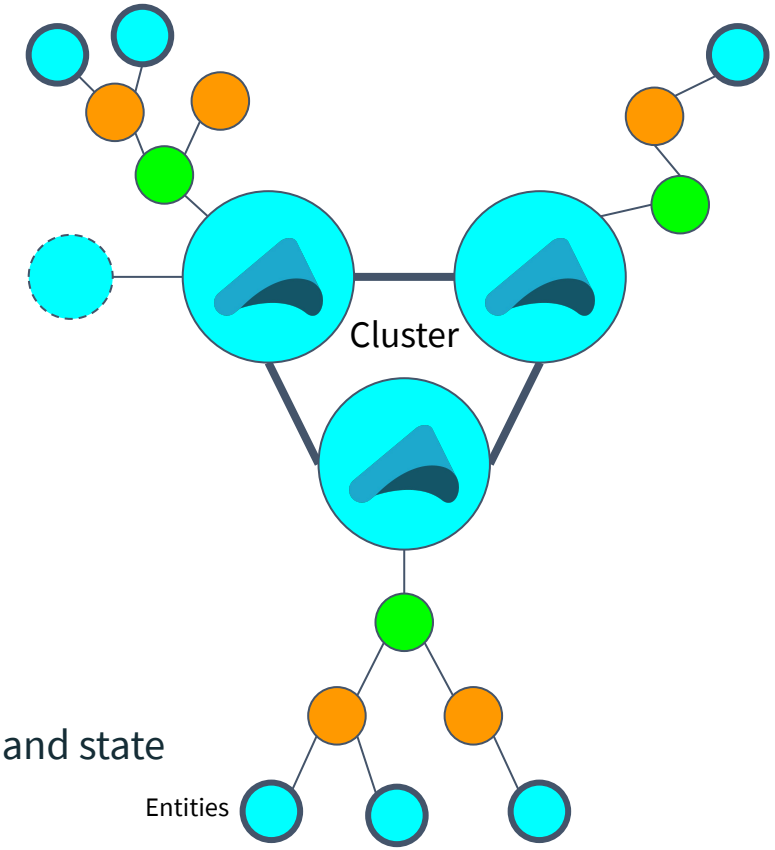


akka cluster flux capacitor 😂

Akka Cluster Sharding

Entities

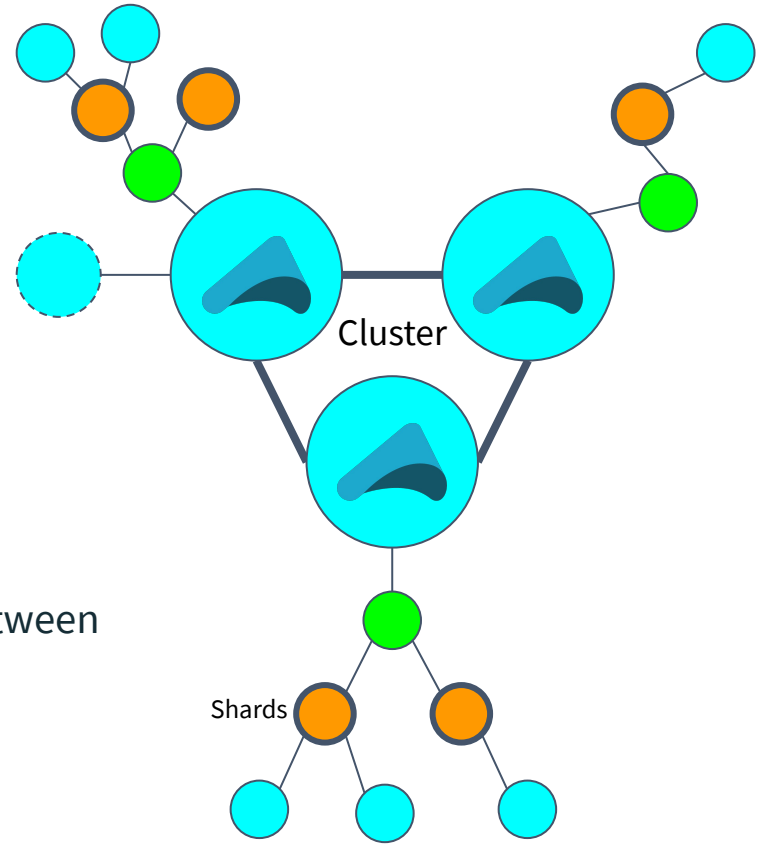
- Represented by an Actor
- Sharding identifies an entity by a unique id
- Only 1 instance of an entity is running
- Entities are a consistency boundary for messages and state



Akka Cluster Sharding

Shards

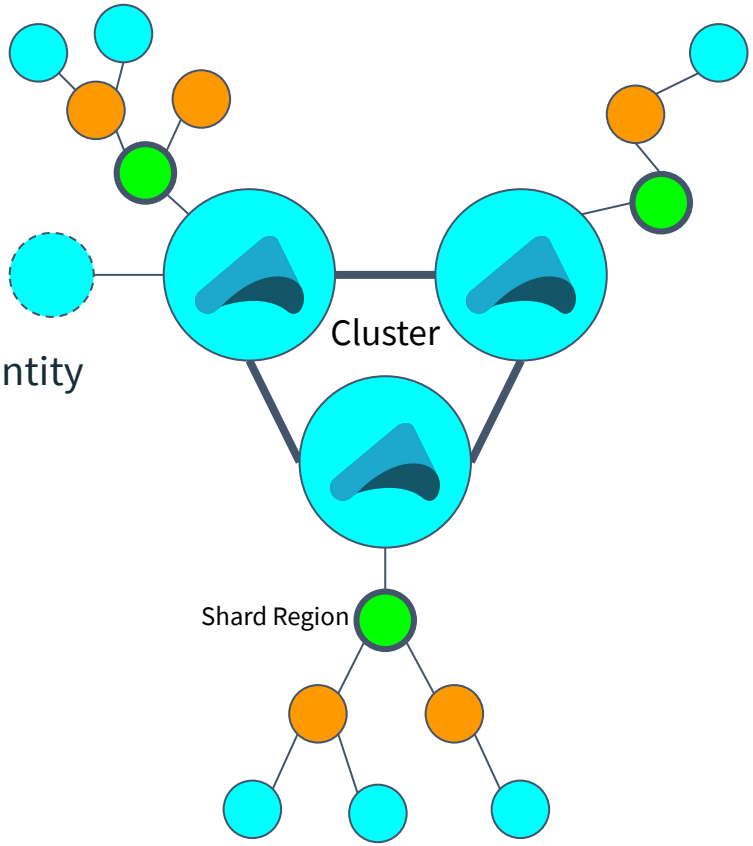
- Encapsulate a set of entities
- Create entities on demand
- Distribution of entities is represented as a map between shards and entities



Akka Cluster Sharding

Shard regions

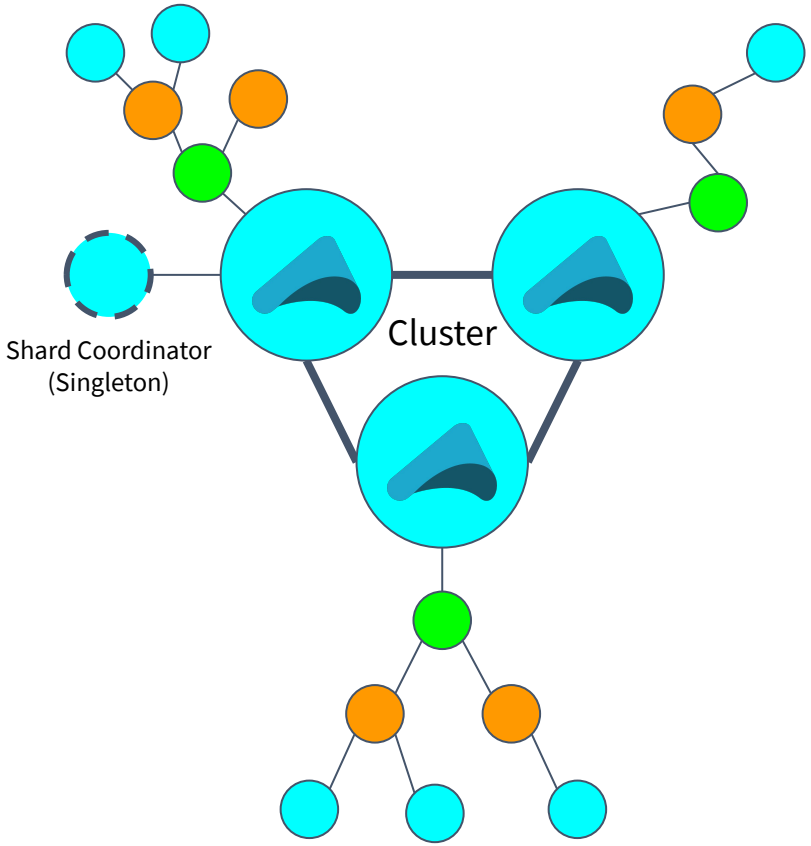
- Forwards messages to the appropriate shard and entity
- May be represented as a local or proxy
  - Local manages entities on the same JVM
  - Proxies forward messages to other regions



Akka Cluster Sharding

Shard coordinator

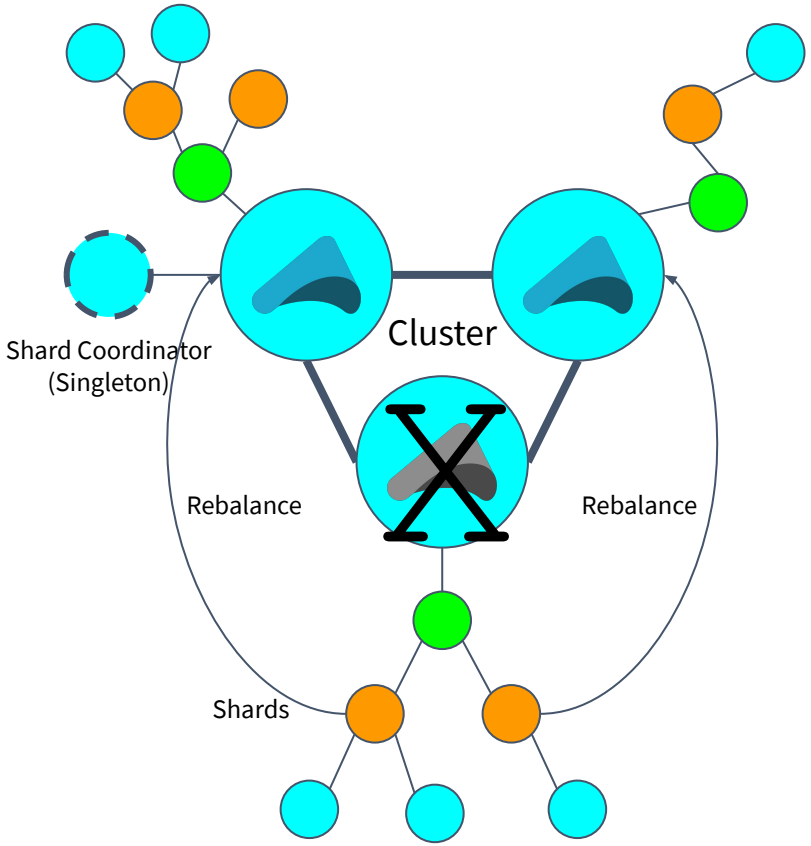
- Brains of cluster sharding
- Executes shard allocation strategy to control distribution of shards across cluster nodes
- Runs as a Cluster Singleton
- Retains state of shard locations



Akka Cluster Sharding

Shard Rebalancing

- Shard coordinator rebalances shards when a cluster node goes down



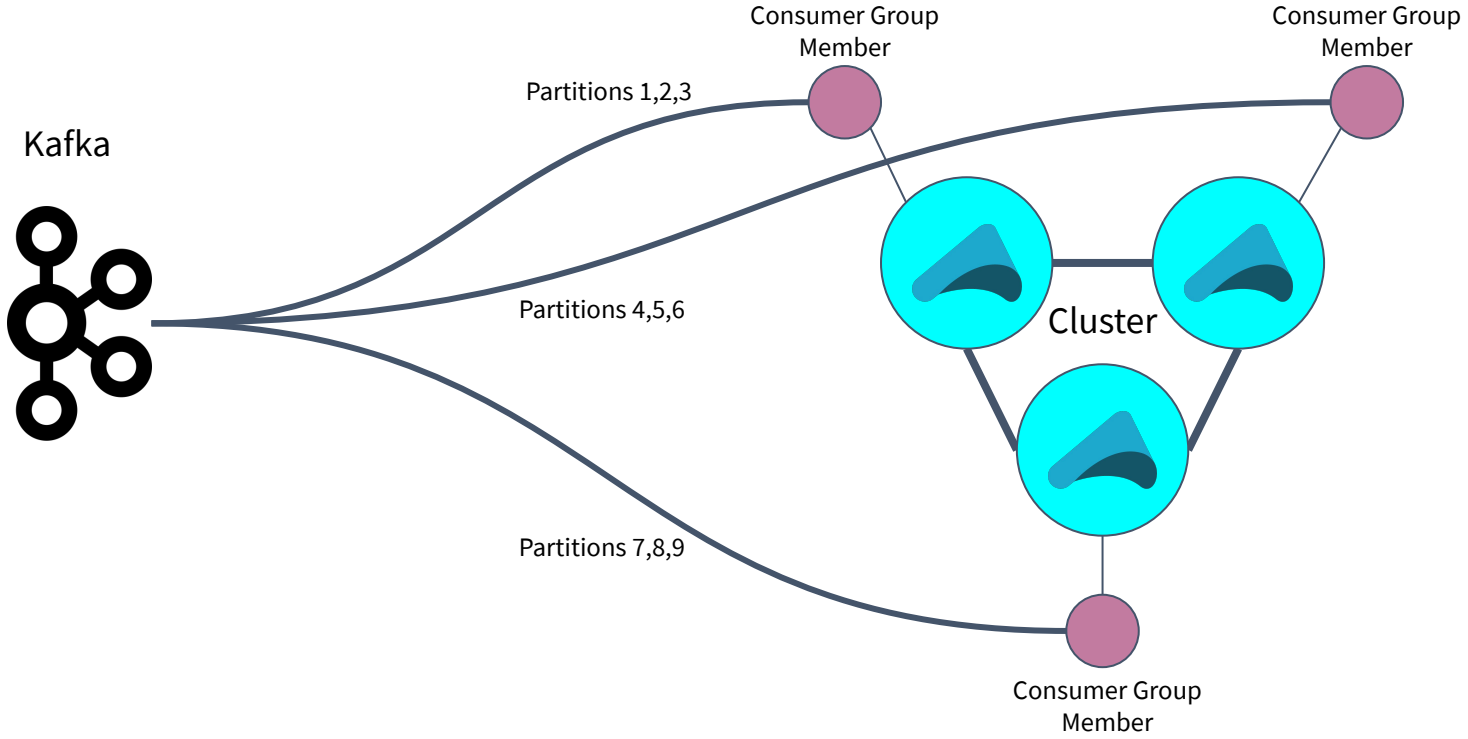
External Shard Allocation Strategies

External Shard Allocation Strategy

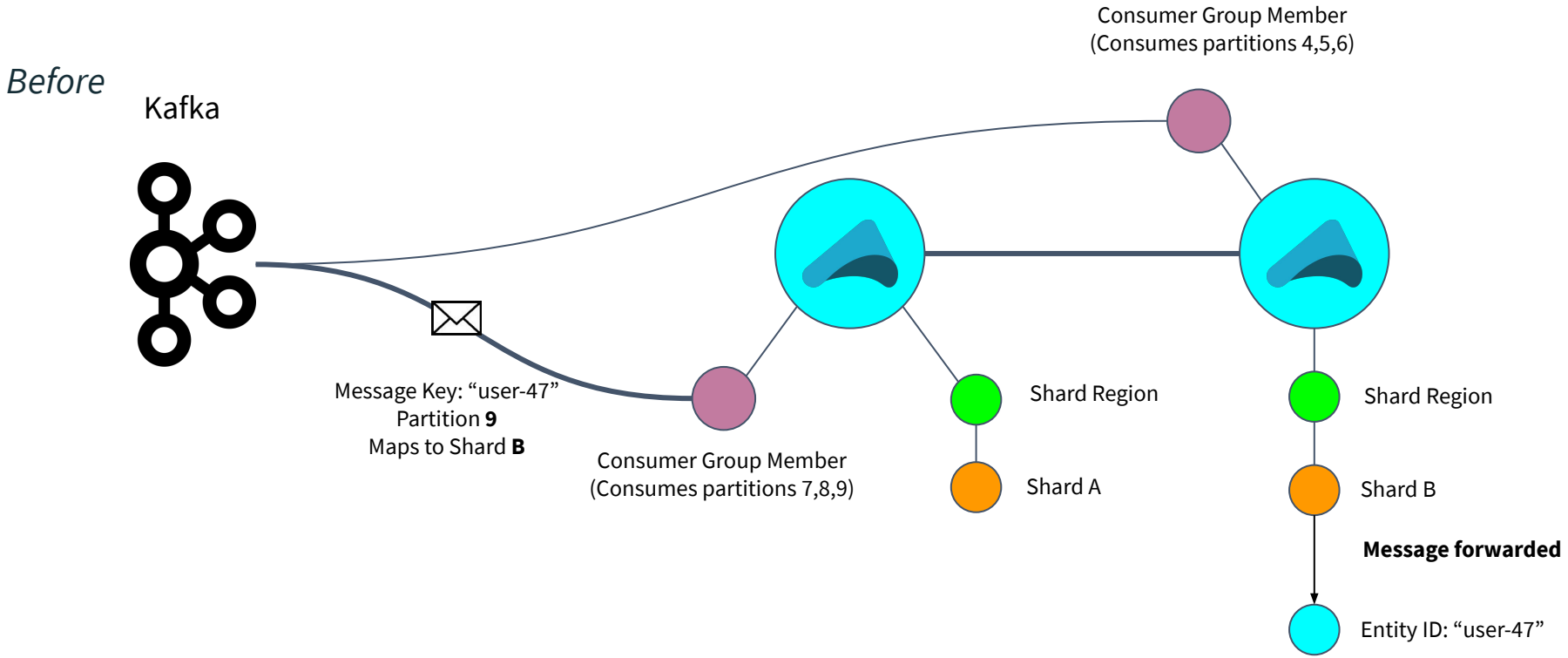
Control shard to Akka cluster node map

- Provides a way to “follow” the partition/sharding strategy of another distributed system
- Canonical example: Kafka Consumer group partitions to Akka cluster shards
- Reduces inter-cluster message traffic by eliminating unnecessary network hops

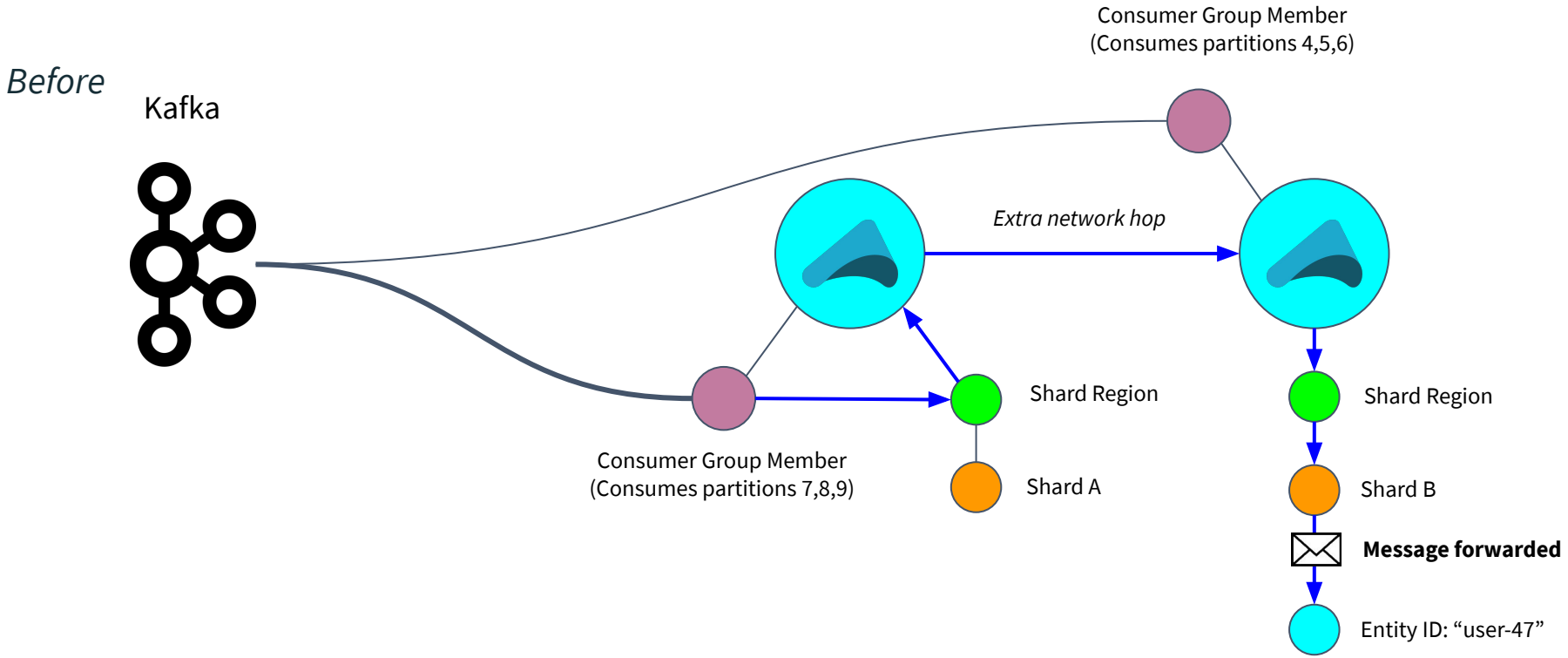
External Shard Allocation Strategy



External Shard Allocation Strategy



External Shard Allocation Strategy



External Shard Allocation Strategy

Congruency between Kafka partitioning and Akka sharding models

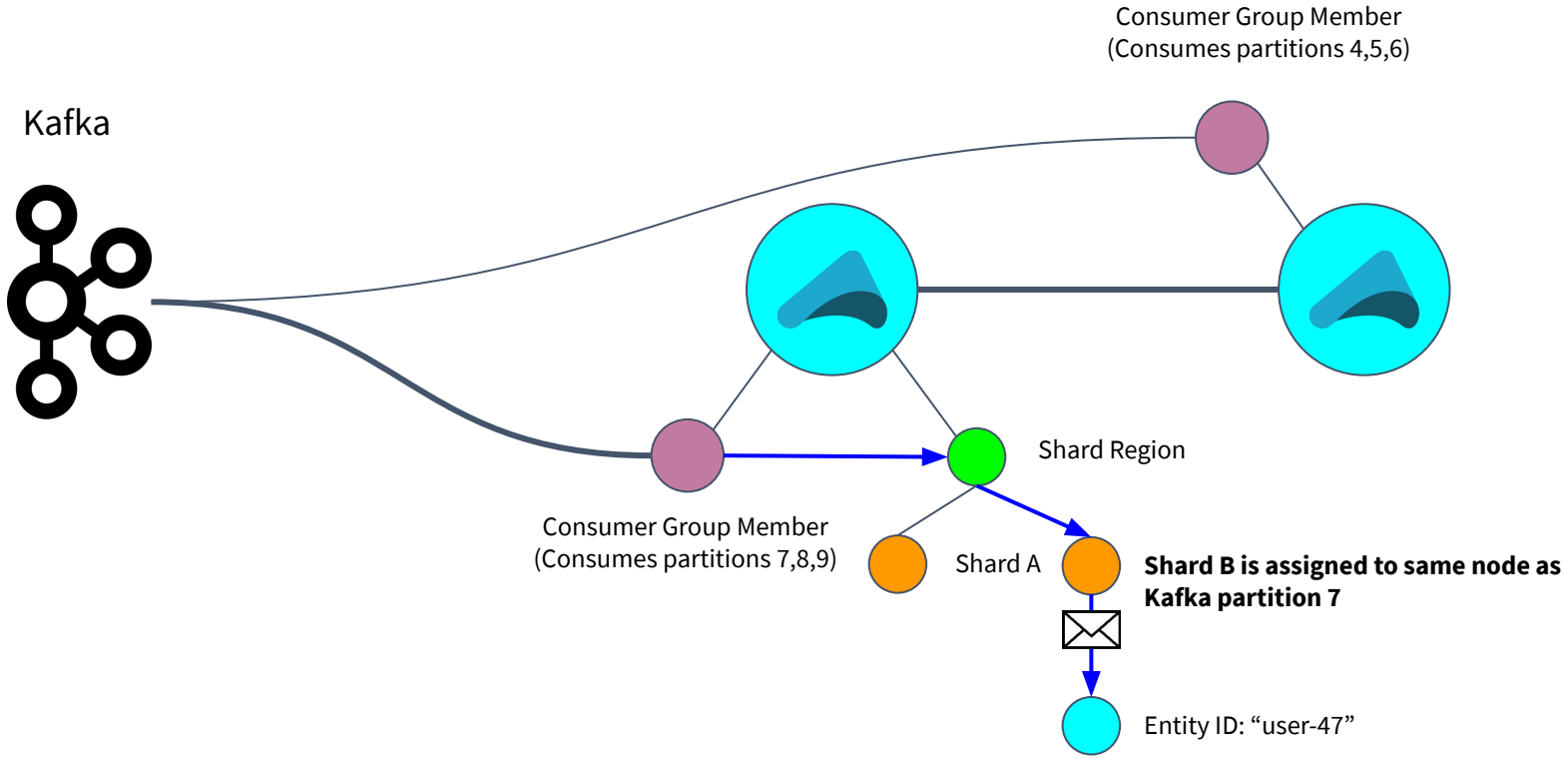
- Alpakka Kafka Cluster Sharding module
- Akka shards and Kafka partitions on same node
- Same sharding & partitioning function with the same number of shards & partitions
- Kafka rebalances trigger Akka shard rebalances
- Sharded Actors (entities) belonging to rebalanced shards are shut down and relocated

Kafka Partition to Akka Shard Map

Kafka Partition	Consumer Group Member	Akka Shard	Akka Cluster Node
1	192.168.0.1	"shard-1"	192.168.0.1
2	192.168.0.1	"shard-2"	192.168.0.1
3	192.168.0.1	"shard-3"	192.168.0.1
4	192.168.0.2	"shard-4"	192.168.0.2
5	192.168.0.2	"shard-5"	192.168.0.2
6	192.168.0.2	"shard-6"	192.168.0.2

External Shard Allocation Strategy

After



Akka Projections

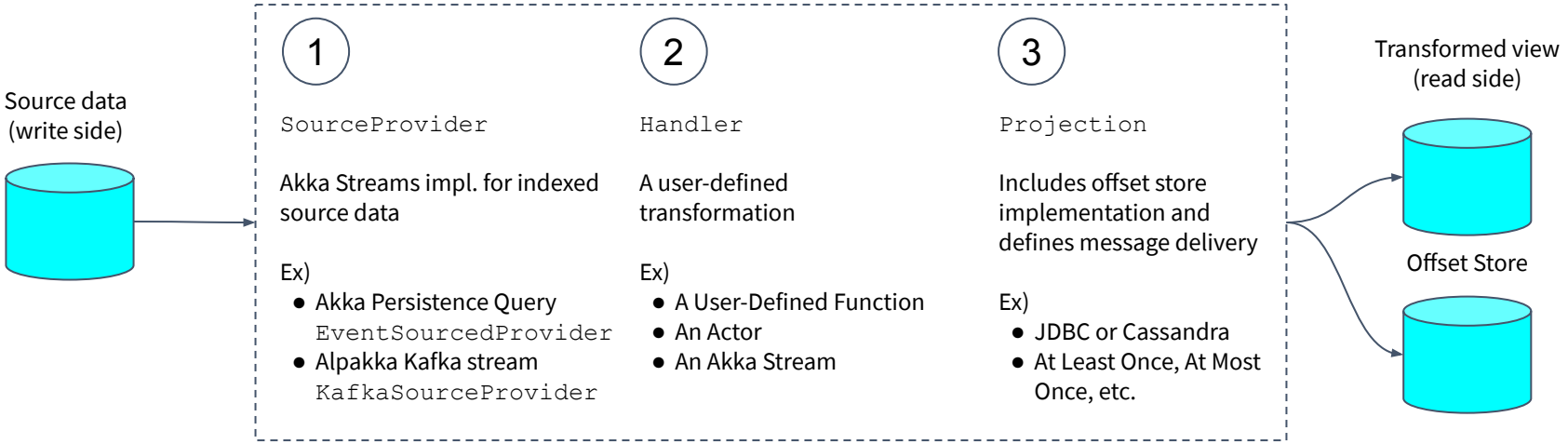
Akka Projections

Akka Projections is a library to generate a read side view

- Consume events from an Akka Stream Source, transform, and write the results to the read side view (i.e. the read-side model in CQRS, a Kafka topic, a Database table, an Actor, etc.)
- Track the offset in an Offset Store
- Manage its lifecycle, restart in the case of failures
- Distribute the consumption in an Akka Cluster

Akka Projections

A Projection is defined with 3 components



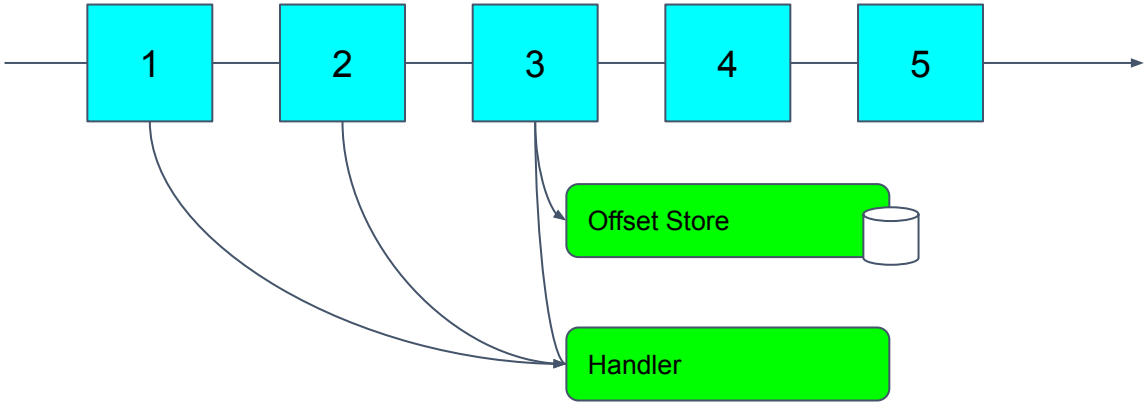
Akka Projections

Offset Stores and Message Delivery Strategies

Message Delivery Strategy		
`atLeastOnce`	✓	✓
`atLeastOnceFlow`	✓	✓
`atMostOnce`	✓	
`exactlyOnce`		✓
`groupedWithin`	✓	✓

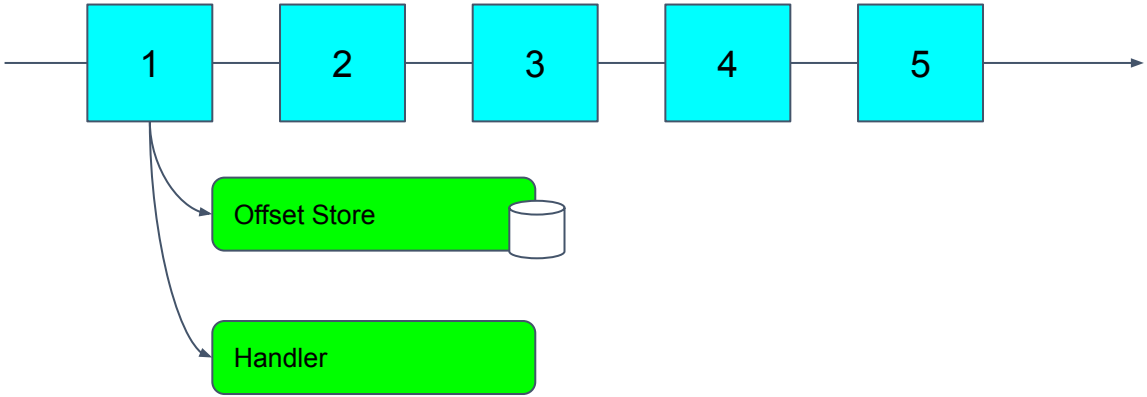
Akka Projections

`atLeastOnce`



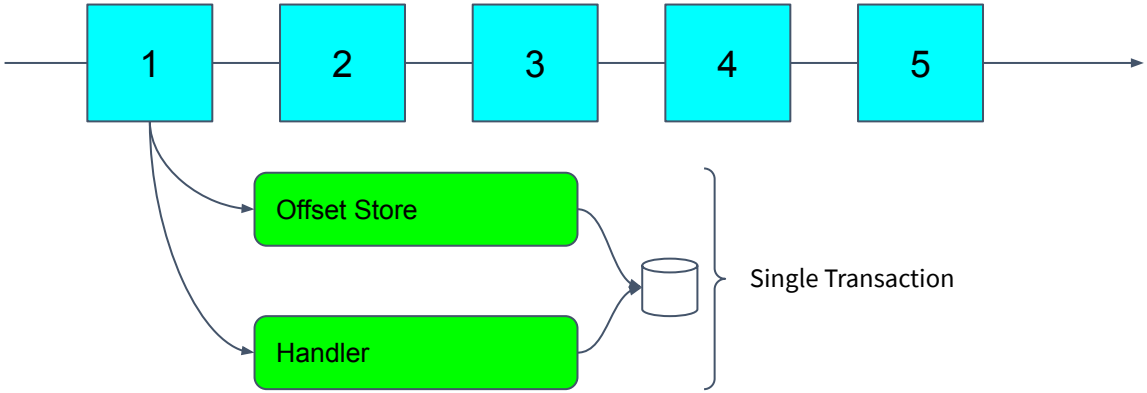
Akka Projections

`atMostOnce`



Akka Projections

`exactlyOnce`



Akka Projections

```
val tag = "shopping-cart-checkout" // this becomes important when we distribute the projection

CassandraProjection.atLeastOnce[Offset, EventEnvelope[ShoppingCartEvents.Event]](
 projectionId = ProjectionId("shopping-carts", tag),
 sourceProvider = EventSourcedProvider.eventsByTag(system,
   readJournalPluginId = CassandraReadJournal.Identifier,
   tag = tag),
 handler = () => Handler { envelope =>
   // do something with `envelope.event`
   Future.successful(Done)
 })
```

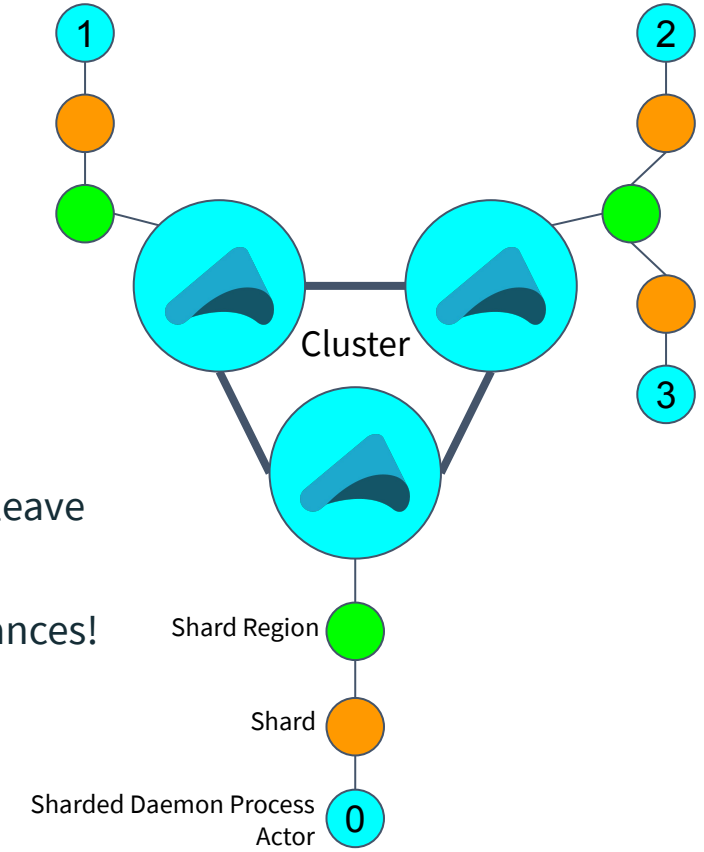
Sharded Daemon Process

To distribute an Akka Projection

Sharded Daemon Process

Distribute a fixed number of workers across Akka Cluster

- Each actor given a unique index: 0, 1, 2 … *n*
- Automatic rebalancing when cluster nodes join and leave
- Common use case: distributing Akka Projection instances!



Sharded Daemon Process

write side

To distribute Akka Projections instances to read from an Akka Persistence event journal we must “slice” or “partition” the data when it’s written.

```
val tags = Vector("shopping-cart-0", "shopping-cart-1", "shopping-cart-2", "shopping-cart-3")
ClusterSharding(system).init(Entity(EntityKey) { entityContext =>
 val i = scala.math.abs(entityContext.entityId.hashCode %  tags.size)
 val selectedTag = tags(i)

 EventSourcedBehavior[Command, Event, State](
   PersistenceId(EntityKey.name, entityContext.entityId),
   State.empty,
   (state, command) => handleCommand(state, command),
   (state, event) => handleEvent(state, event)
 ).withTagger(_ => Set(selectedTag))
})
```

Sharded Daemon Process

read side

With a partitioned event journal we can run as many Akka Projection instances as there are partitions using a Sharded Daemon Process.

```
val tags = Vector("shopping-cart-0", "shopping-cart-1", "shopping-cart-2", "shopping-cart-3")

ShardedDaemonProcess(system).init[ProjectionBehavior.Command](
 name = "shopping-cart-projection",
 numberOfInstances = tags.size,
 behaviorFactory = (i: Int) => ProjectionBehavior(shoppingCartProjection(tags(i)))
 stopMessage = ProjectionBehavior.Stop
)
```

Split Brain Resolver

Split Brain Resolver

Split Brain Resolver (SBR) was open sourced in Akka 2.6.6!

- Formerly a commercial Lightbend component
- Battletested by Lightbend customers
- Now open source and free to use by the community
- Missing piece of the puzzle to run Akka Cluster

Split Brain Resolver

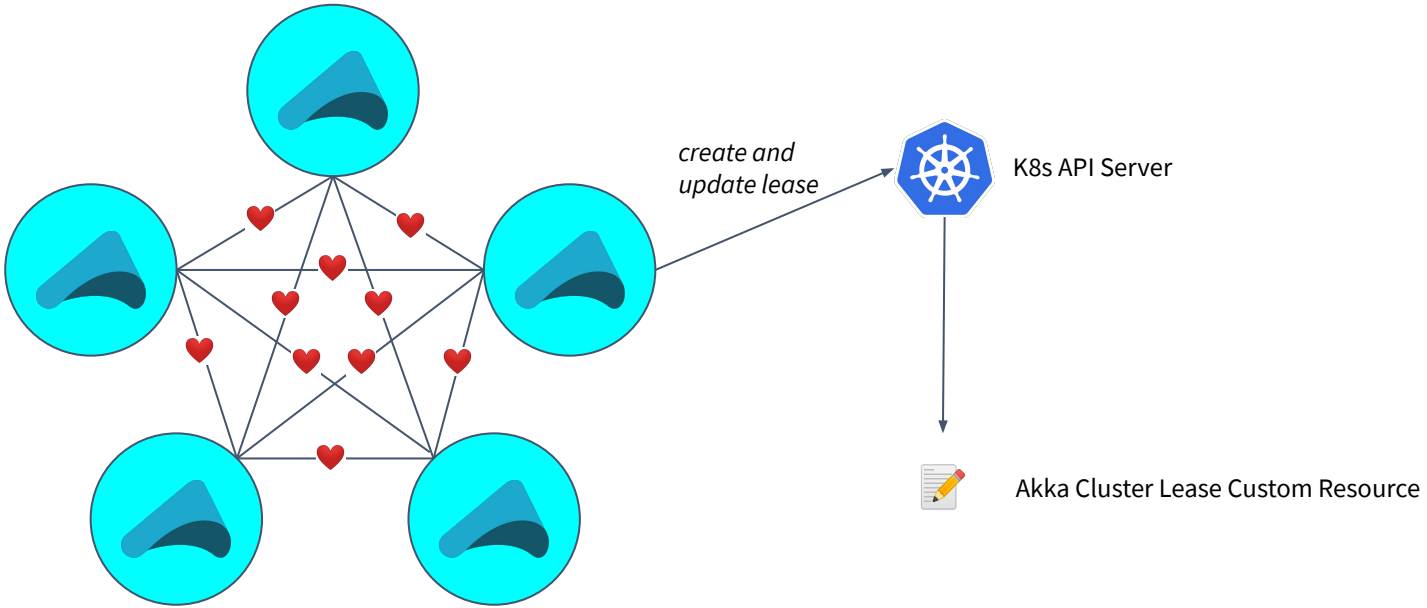
Split Brain Resolver (SBR) in a nutshell

- Strategies to handle network partitions in Akka Cluster
- Network partition: one node cannot communicate with another
- **There's no way to know** if a node is down or unreachable
- Akka Cluster Singleton and Cluster Sharding require a complete view of the cluster

Split Brain Resolver

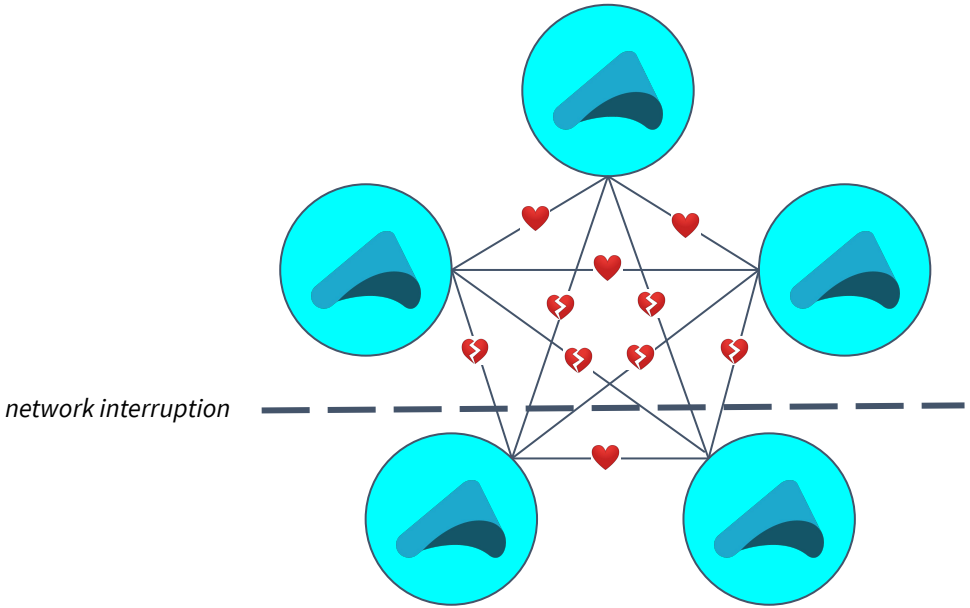
❤️ *cupid + rock & roll* 🤘 *?*

Ex) Kubernetes Lease Strategy “lease-majority”



Split Brain Resolver

Network partition

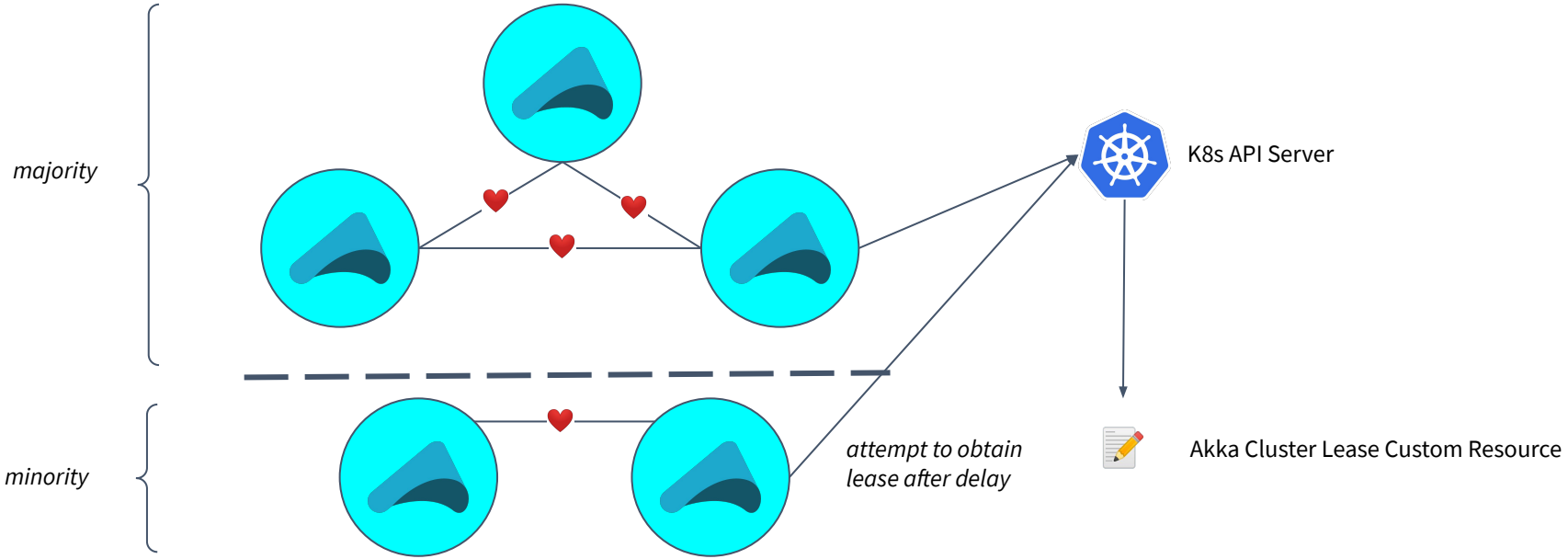


K8s API Server

Akka Cluster Lease Custom Resource

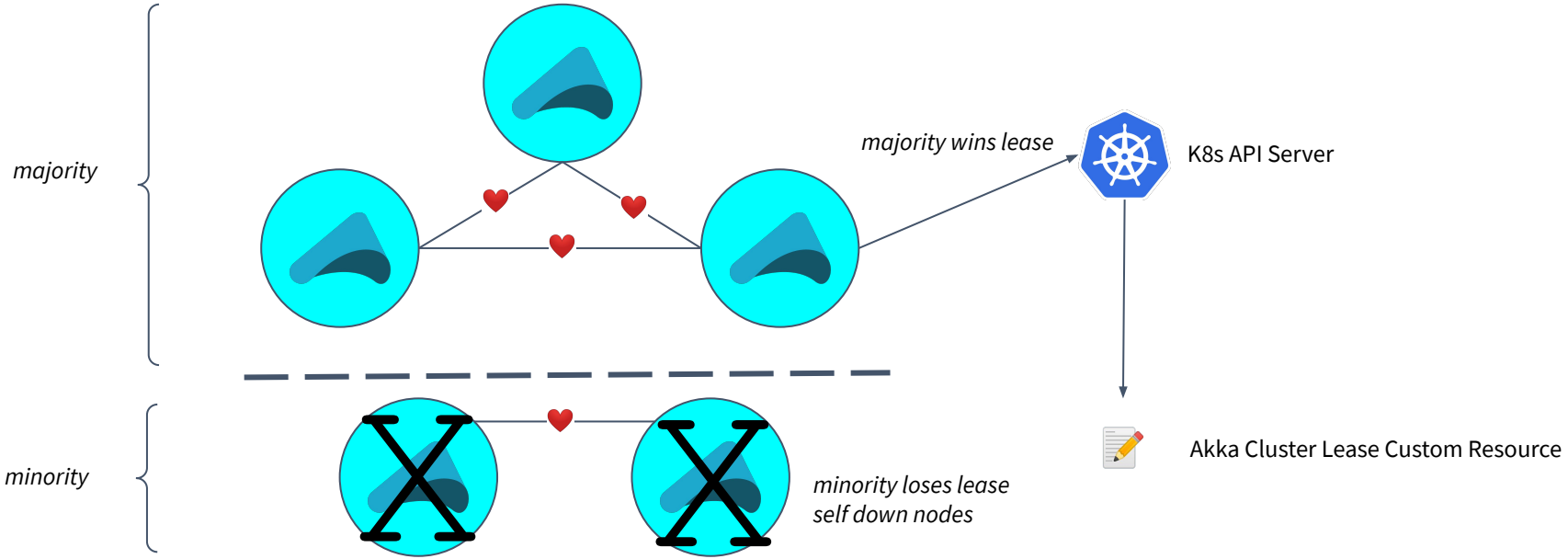
Split Brain Resolver

Heal Split Brain



Split Brain Resolver

Heal Split Brain

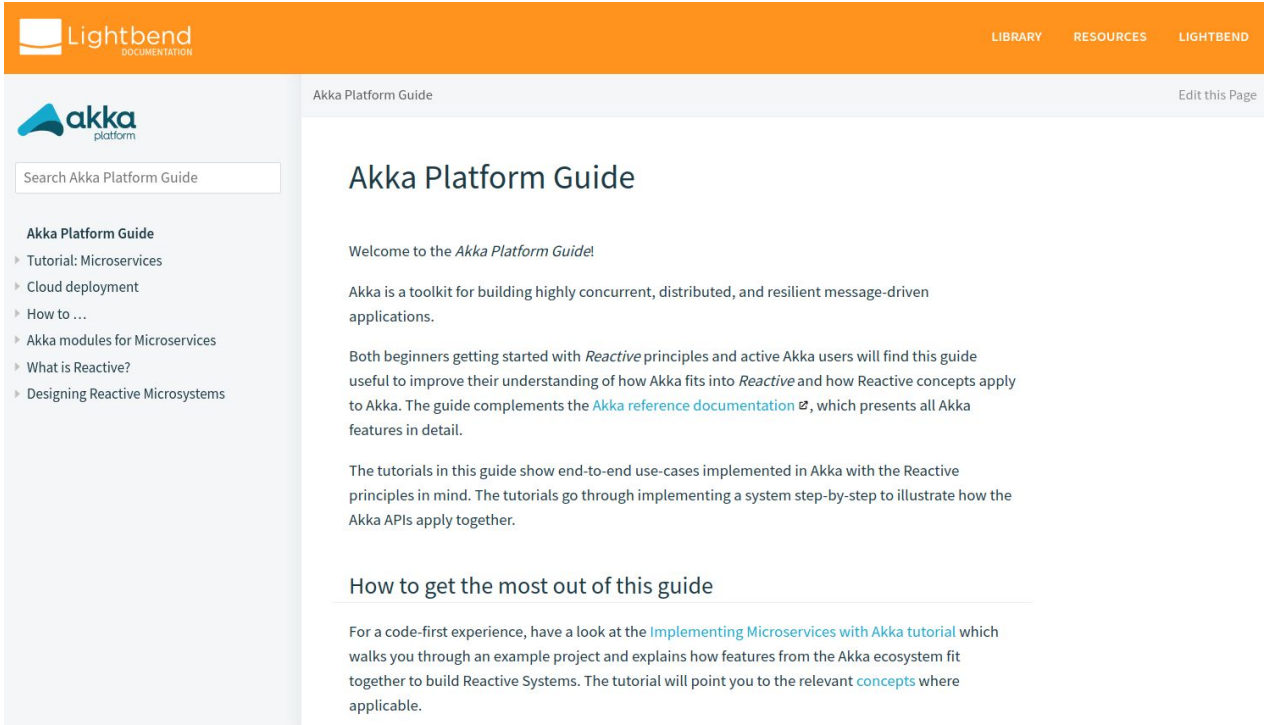


Akka Platform Guide

Lightbend

Akka Platform Guide

https://developer.lightbend.com/docs/akka-platform-guide/index.html



Resources

Learning Refs and Resources

- One year of Akka 2.6 (blog post)
- Akka 2.6.0 Release Announcement
- Akka Platform Guide
- Documentation
  - Akka Actors
  - Akka Persistence Typed
  - Akka Cluster Sharding Typed
  - External Shard Allocation Strategy
  - Alpakka Kafka Cluster Sharding Support
  - Aligning Kafka Partitions with Akka Cluster Sharding Sample Project
  - Akka Projections
  - Sharded Daemon Process
  - Split Brain Resolver
  - Akka Management Kubernetes API
  - Akka Management Kubernetes Lease
- Videos
  - Six things you can do in Akka 2.6
  - Distributed processing with Akka Cluster & Kafka (External Shard Allocation Strategy)
  - Intro to Akka Cluster Sharding
  - Akka Projections 1.0
  - Replicated event sourcing for active-active event sourcing
  - Data modeling for Replicated event sourcing
  - Split Brain Resolver in Akka Cluster
  - How Akka Cluster works, Split Brain Resolver

Thank You!

Sean Glover

@seg1o

in/seanaglover

sean.glover@lightbend.com