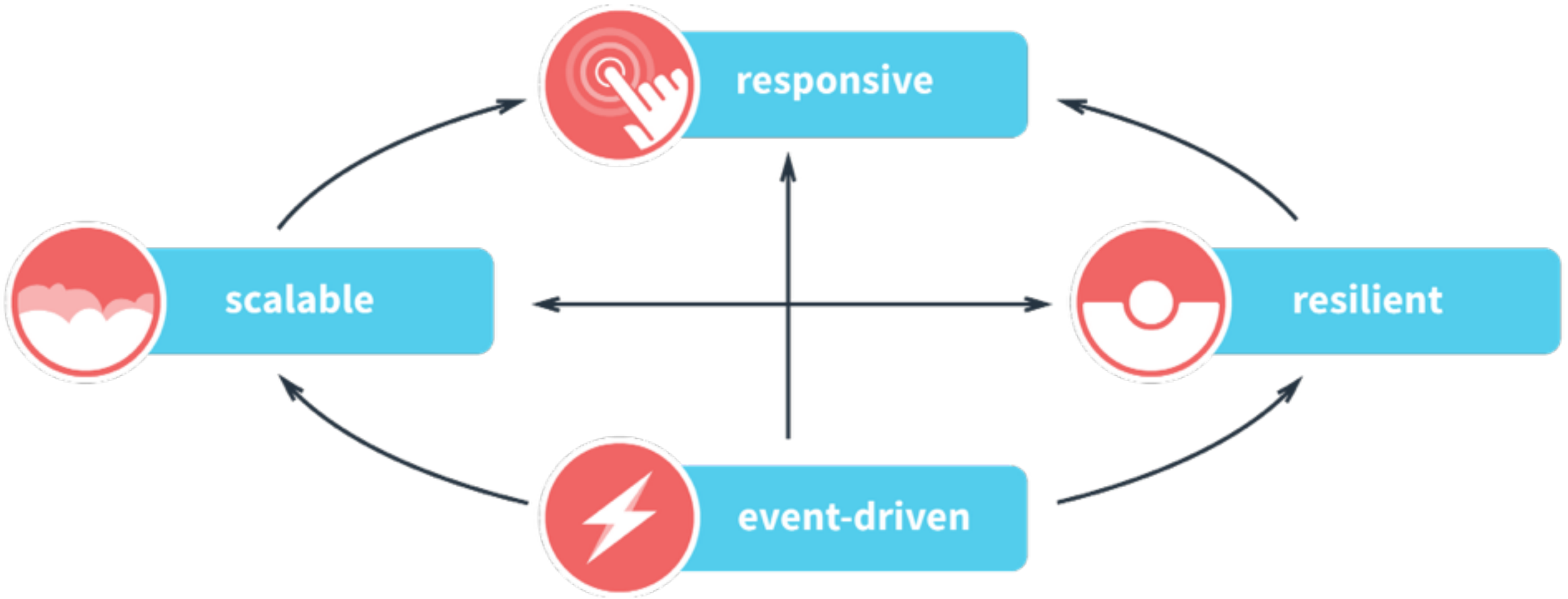
Resilient Applications with Akka Persistence

Björn Antonsson
@bantonsson

Konrad Malawski
@ktosopl

Patrik Nordwall
@patriknw

Reactive Applications



Resilient

- Embrace Failure
  - Failure is a normal part of the application lifecycle
- Self Heal
  - Failure is detected, isolated, and managed

The Naïve Way

- Write State to Database
- Transactions Everywhere
- Problem Solved?
- Not Scalable, Responsive, Event-Driven!

Command and Event Sourcing

Command and Event Sourcing

- State is the sum of Events
- Events are persisted to Store
- Append only
- Scales well

Command v.s. Event

- Command
  - What someone wants me to do
  - Can be rejected
- Event
  - Something that has already happened
  - An immutable fact

Commands can Generate Events

- If I accept a Command and change State
  - Persist Event to Store
- If I crash
- Replay Events to recover State

Persist All Commands?

- If I crash on a Command
  - I will likely crash during recovery
- Like the Army
  - Don't question orders
  - Repeat until success

Only Persist Events

- Only accepted Commands generate Events
  - No surprises during recovery
- Like a dieting method
  - You are what you eat

Achievement Unlocked?

- Resilient
  - State is recoverable
- Scalable
  - Append only writes
- Something Missing?
  - Queries

CQRS

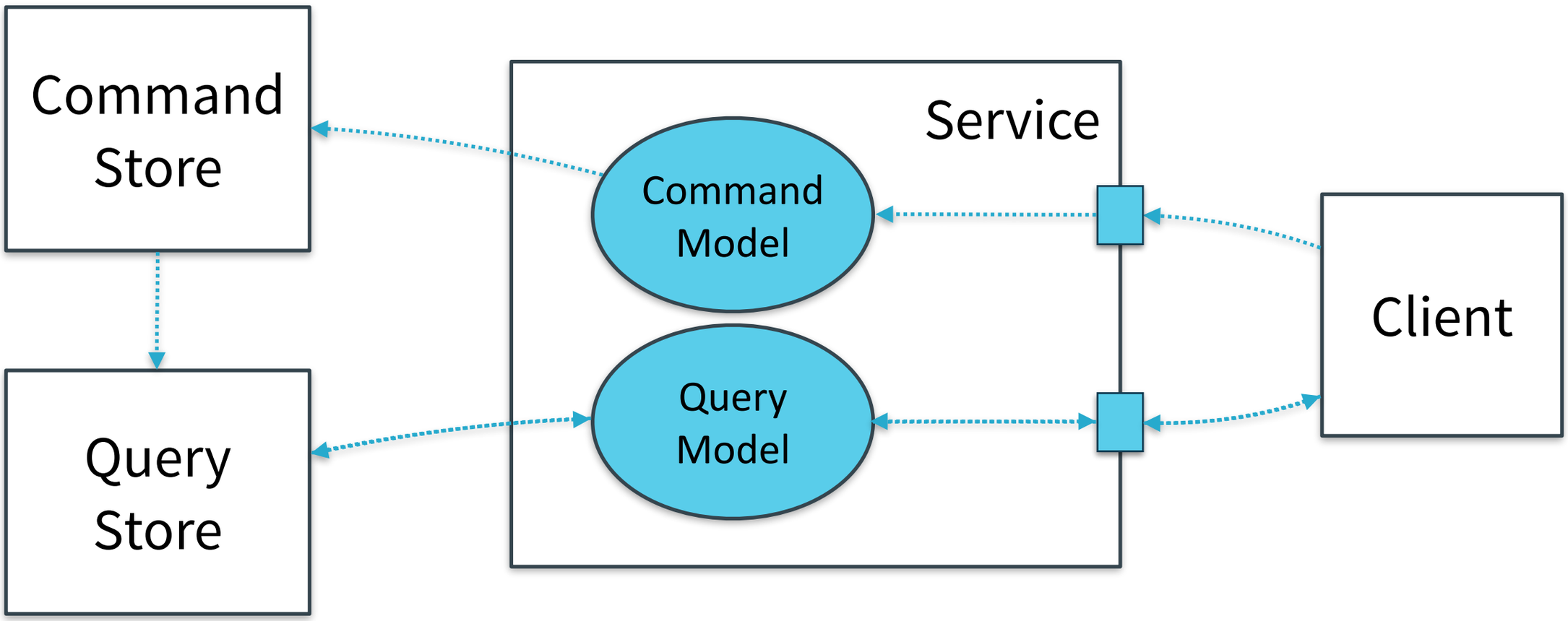
Command Query Responsibility Segregation

CQRS

- Separate Models
- Command Model
  - Optimized for command processing
- Query Model
  - Optimized data presentation

Query Model from Events

- Source the Events
- Pick what fits
  - In Memory
  - SQL Database
  - Graph Database
  - Key Value Store


Command Store
Service
Command Model
Query Model
Client
Query Store

PersistentActor

PersistentActor

Replaces:

Processor & Eventsourced Processor

*in **Akka 2.3.4+***

super quick domain modelling!

Commands - what others "tell" us; not persisted

```
sealed trait Command
case class GiveMe(coins: Int) extends Command
case class TakeMy(coins: Int) extends Command
```

Events - reflect effects, past tense; **persisted**

```
sealed trait Event
case class BalanceChangedBy(coins: Int) extends Event
```

State - reflection of a series of events

```
case class Wallet(coins: Int) {
  def updated(diff: Int) = State(coins + diff)
}
```

PersistentActor

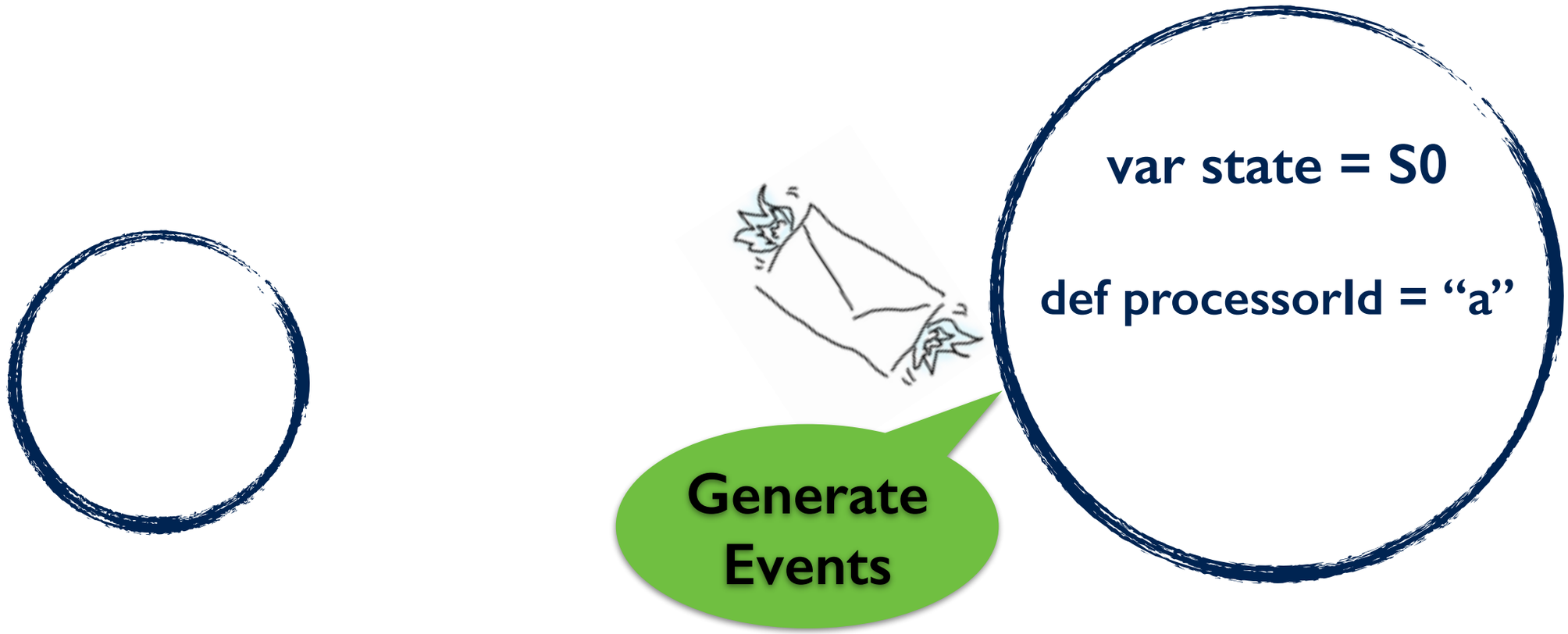
Command

var state = S0

def processorId = “a”

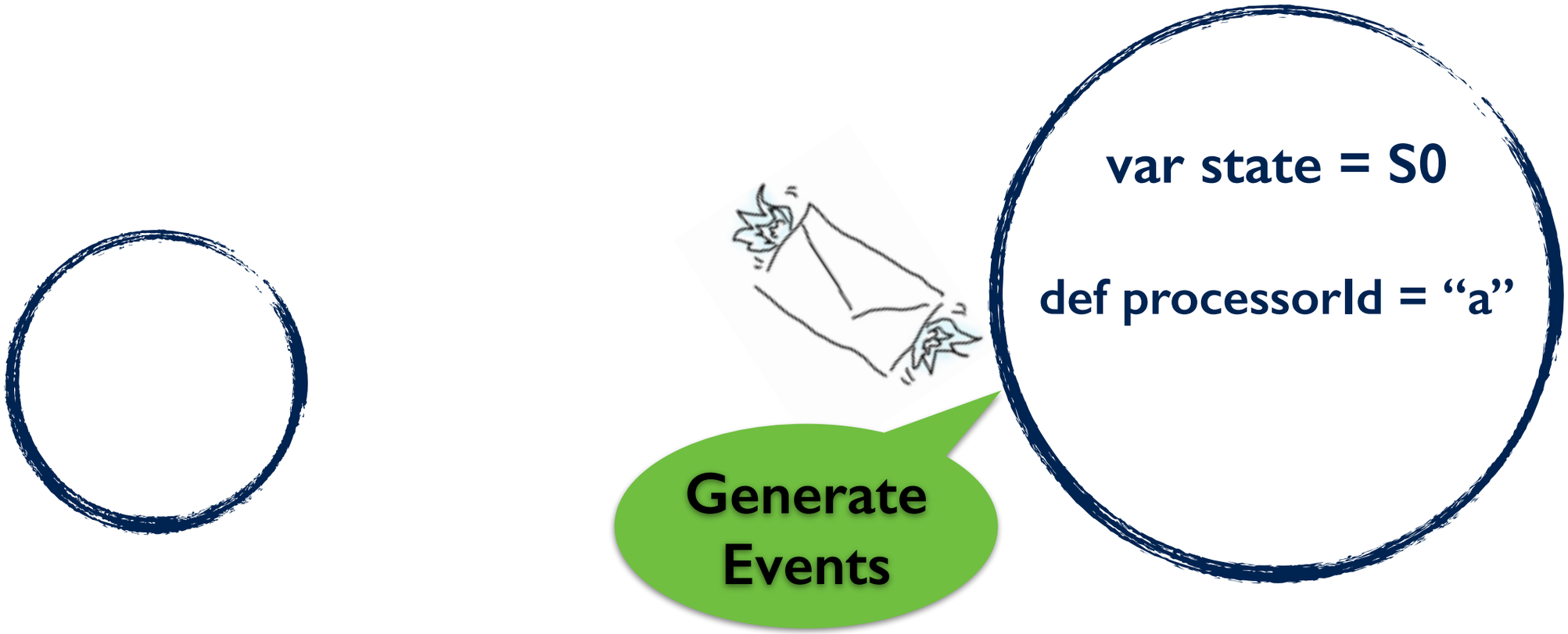
Journal

PersistentActor

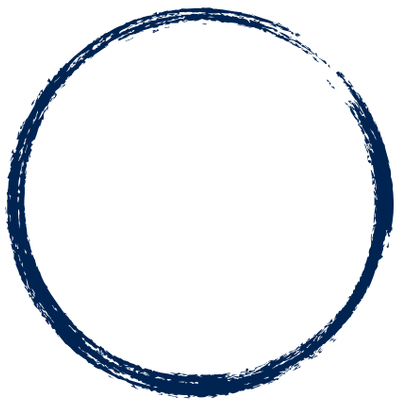


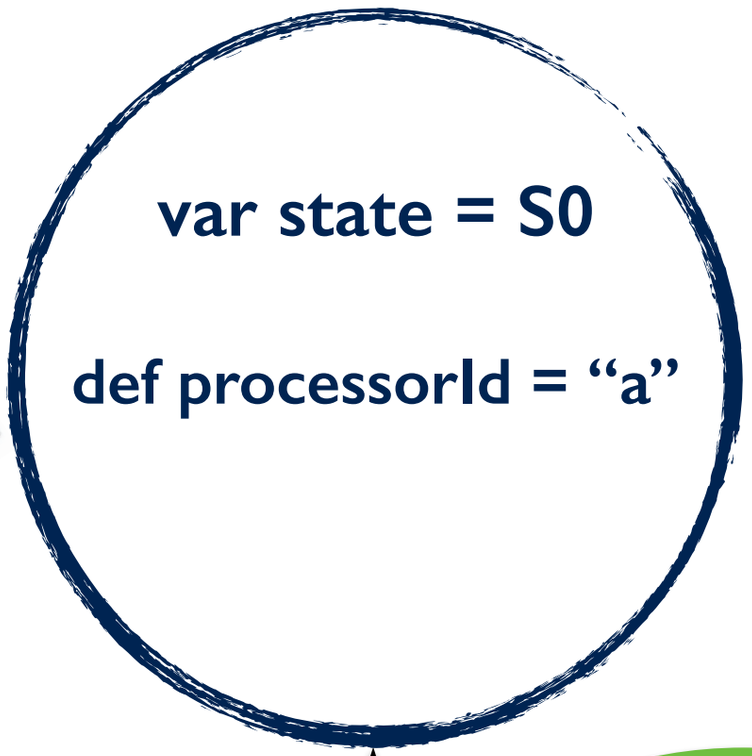
Journal

PersistentActor

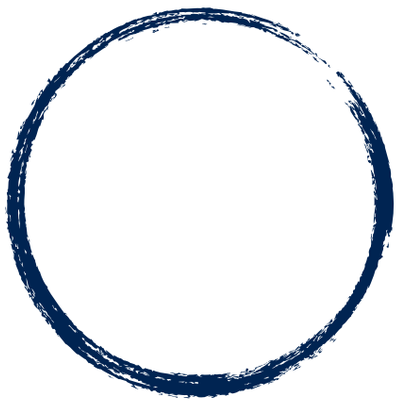


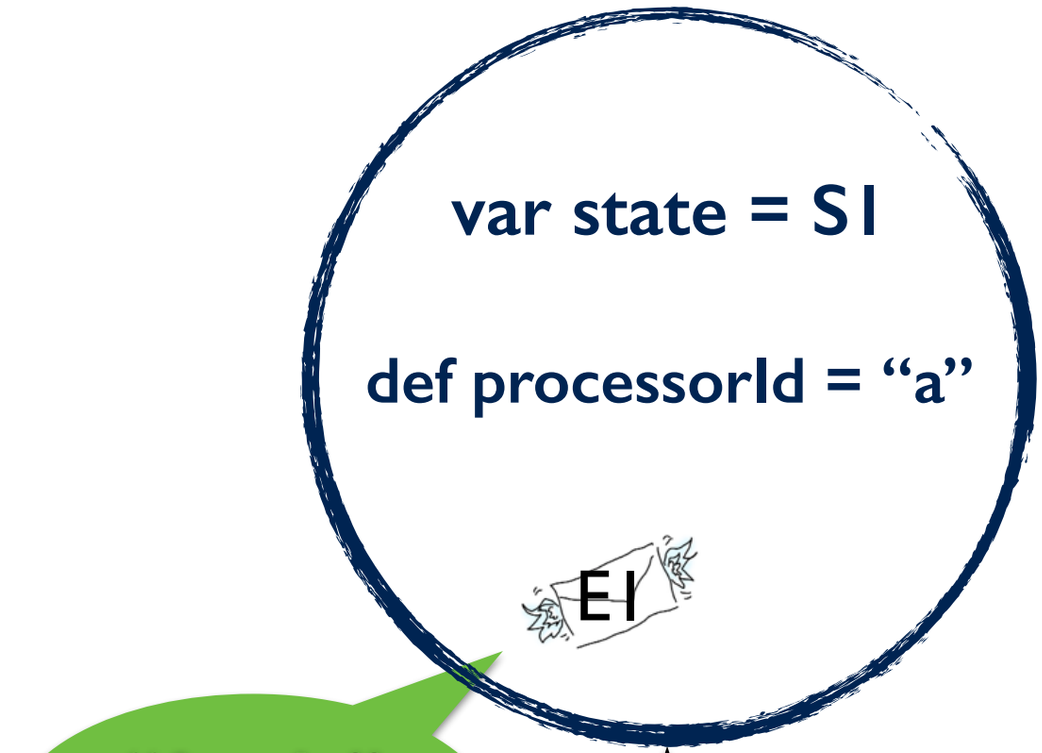
PersistentActor



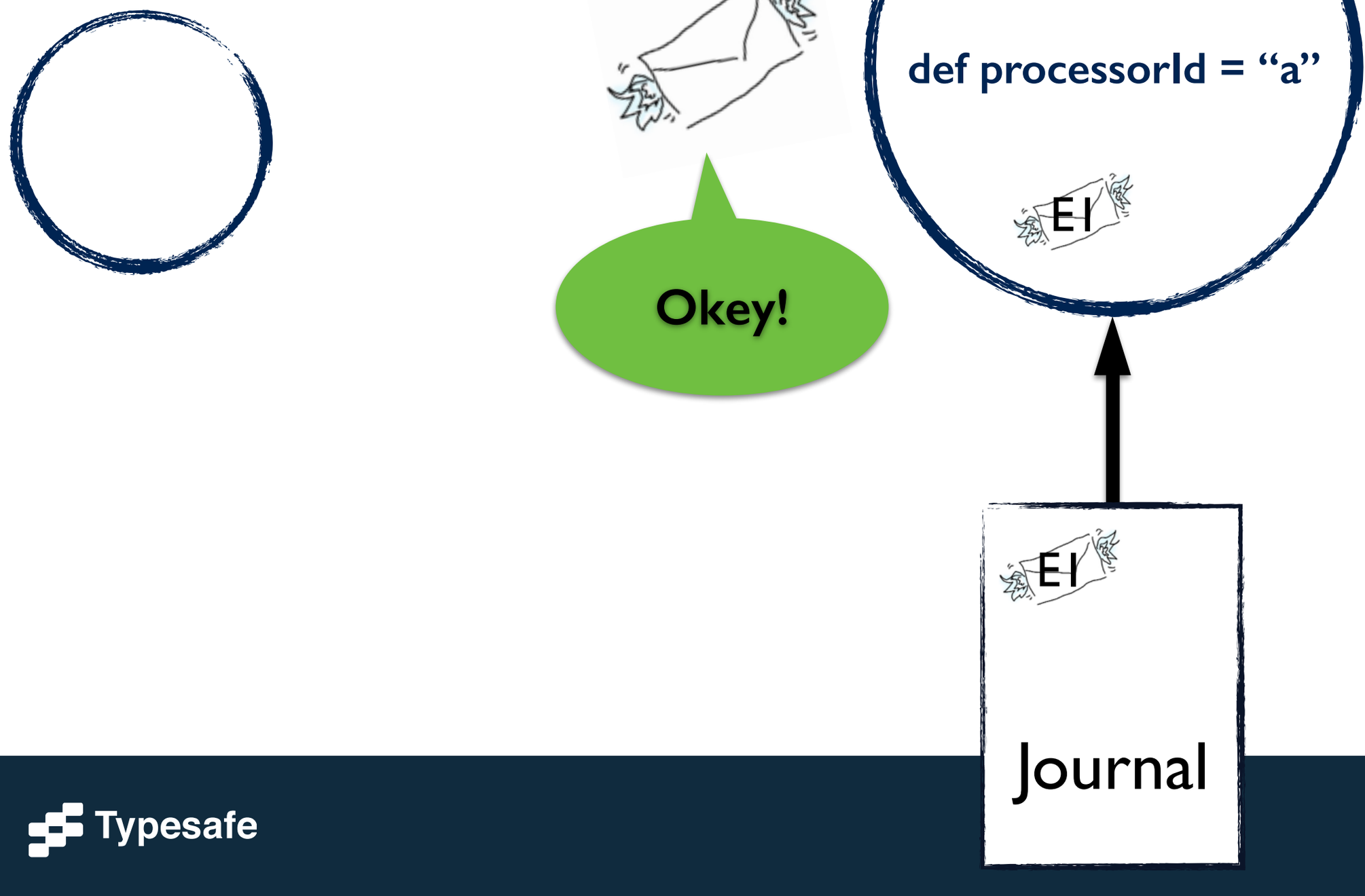


PersistentActor

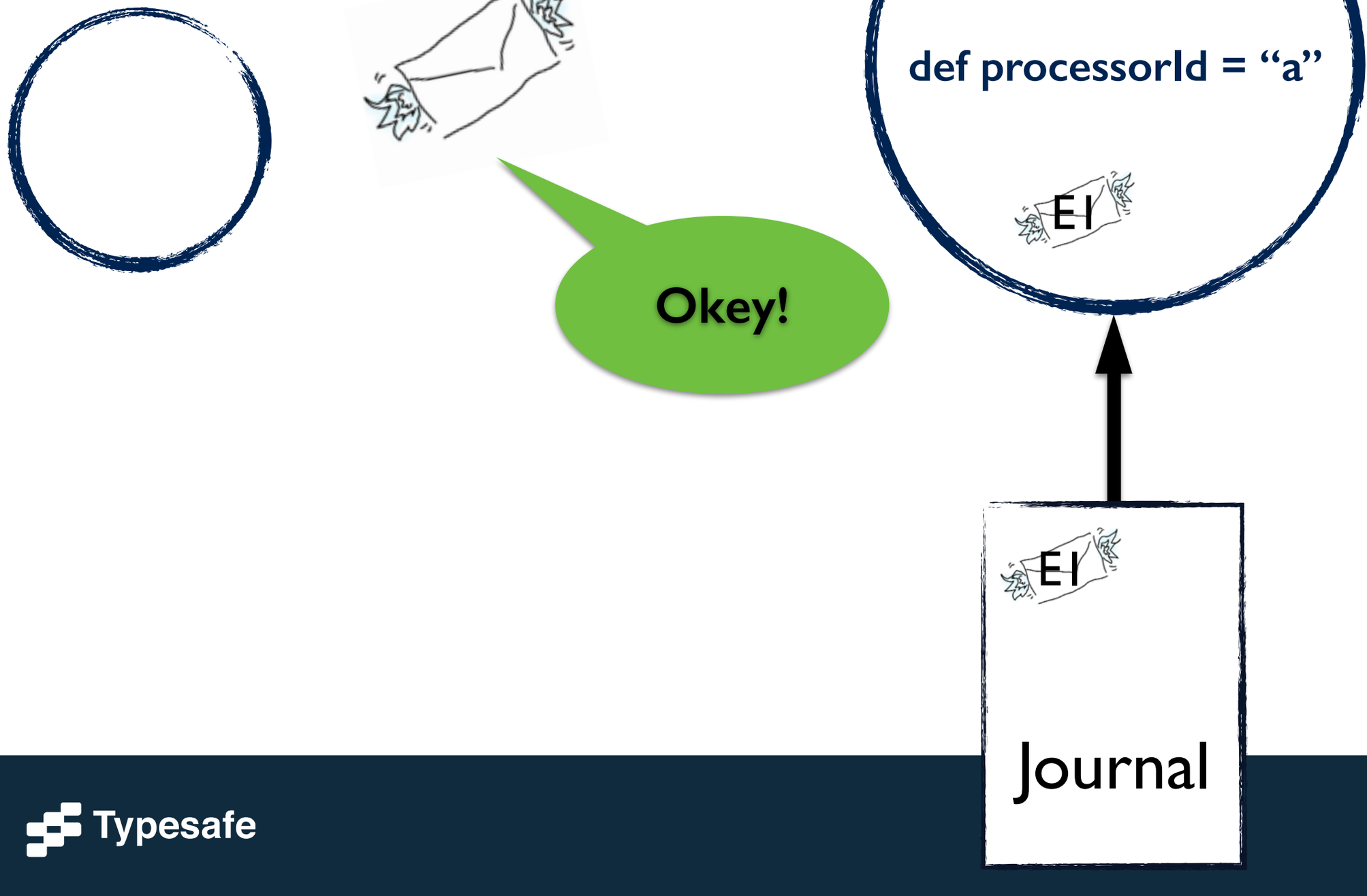




PersistentActor



PersistentActor



PersistentActor

PersistentActor

```
class BitCoinWallet extends PersistentActor {

  var state = Wallet(coins = 0)

  def updateState(e: Event): State = {
    case BalanceChangedBy(coins) => state.updatedWith(coins)
  }

  // API:

  def receiveCommand = ??? // TODO

  def receiveRecover = ??? // TODO

}
```

```
persist(e) { e => }
```

PersistentActor

```
def receiveCommand = {

  case TakeMy(coins) =>
    persist(BalanceChangedBy(coins)) { changed =>
      state = updateState(changed)
    }

}
```

async callback

PersistentActor: persist(){}

```
def receiveCommand = {
  case GiveMe(coins) if coins <= state.coins =>
    persist(BalanceChangedBy(-coins)) { changed =>
      state = updateState(changed)
      sender() ! TakeMy(coins)
    }
}
```

Safe to mutate the Actor's state

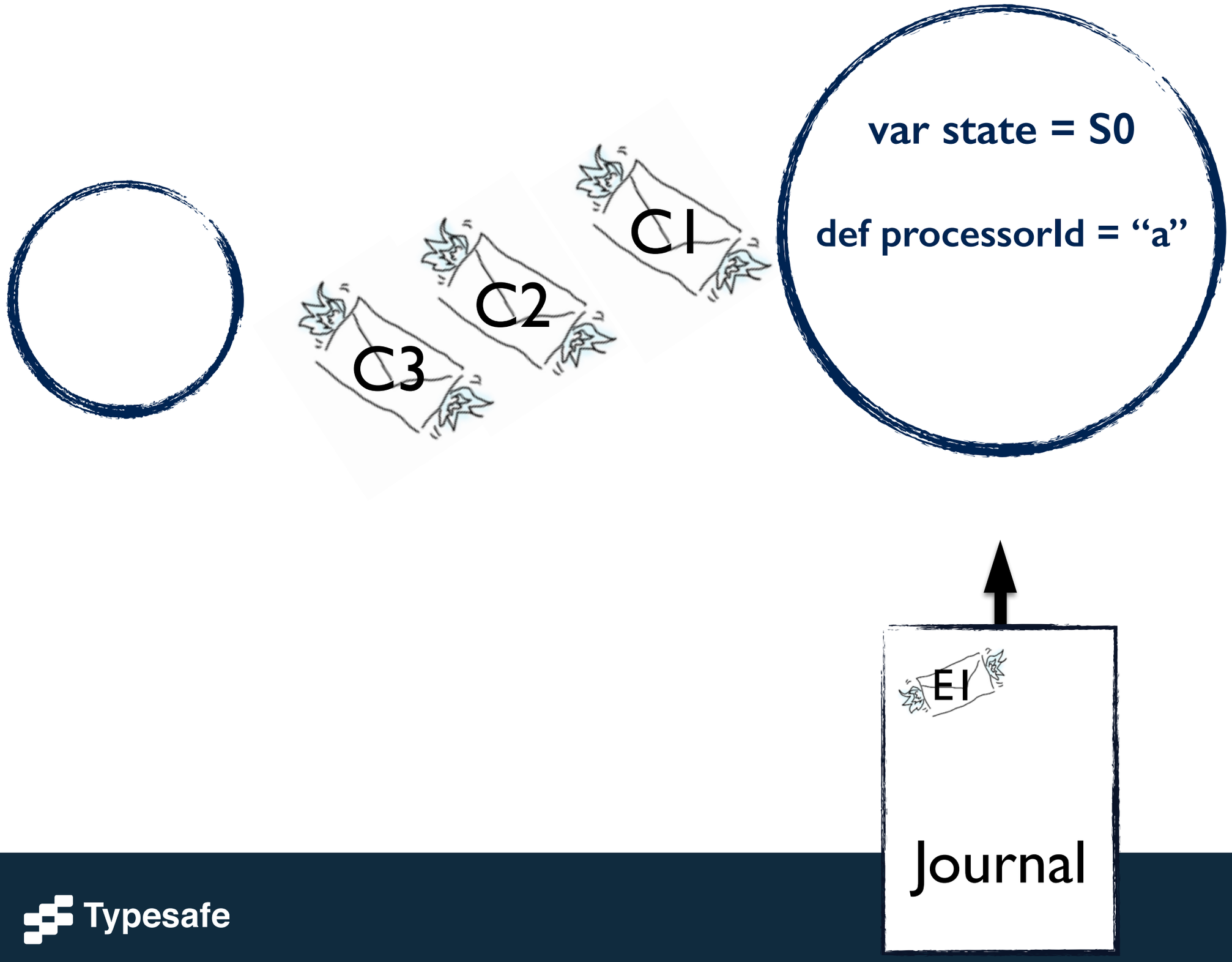
async callback

PersistentActor

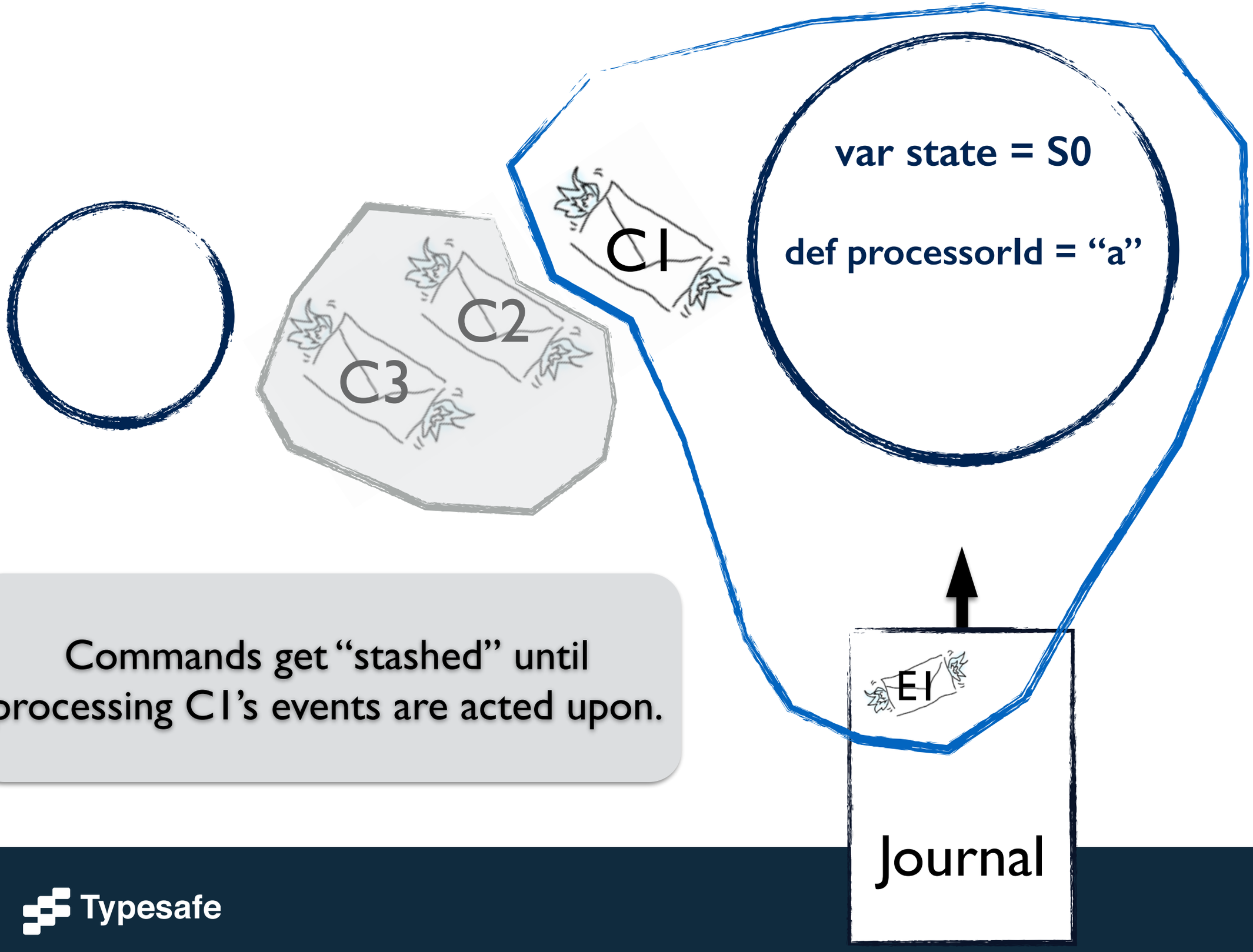
```
def receiveCommand = {
  case GiveMe(coins) if coins <= state.coins =>
    persist(BalanceChangedBy(-coins)) { changed =>
      state = updateState(changed)
      sender() ! TakeMy(coins)
    }
}
```

Safe to access sender here

persist(){} - Ordering guarantees

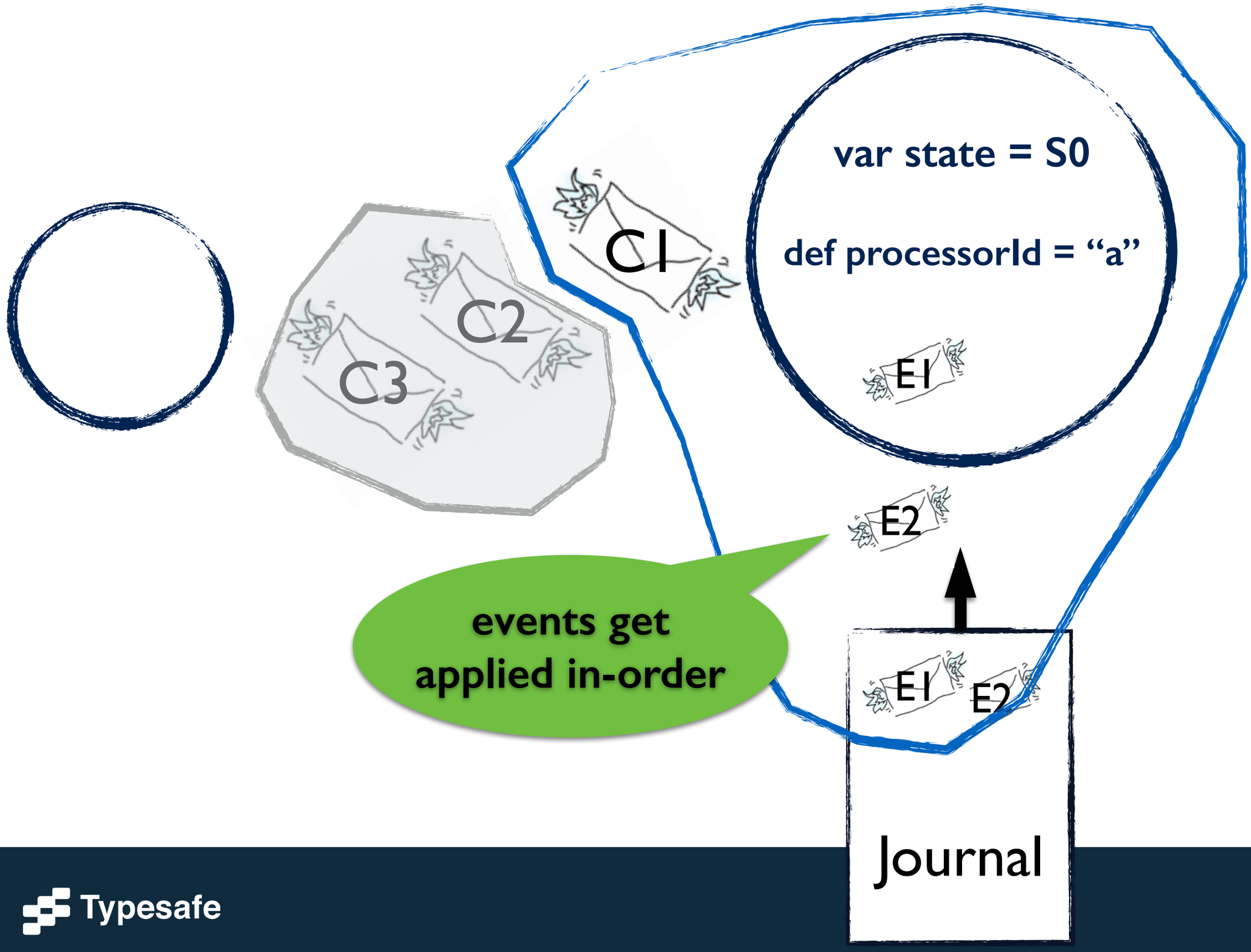


persist(){} - Ordering guarantees

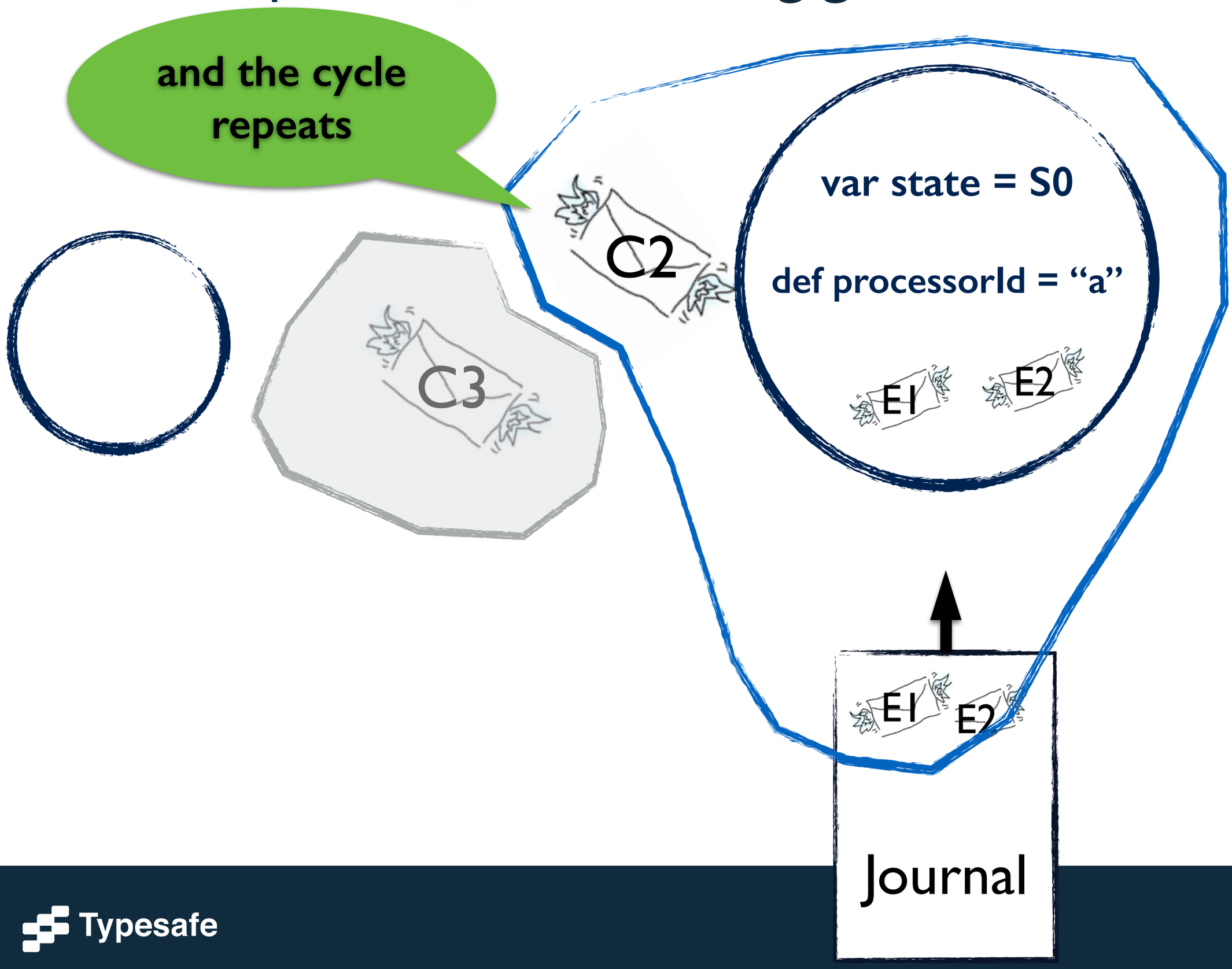


Commands get “stashed” until processing C1’s events are acted upon.

persist(){} - Ordering guarantees



persist(){} - Ordering guarantees



persistAsync(e) { e => }

persistAsync(e) { e => }

+

defer(e) { e => }

PersistentActor: persistAsync(){}

```
def receiveCommand = {


  case Mark(id) =>
   sender() ! InitMarking
   persistAsync(Marker) { m =>
     // update state...
   }


}
```

persistAsync

will *NOT* force stashing of commands

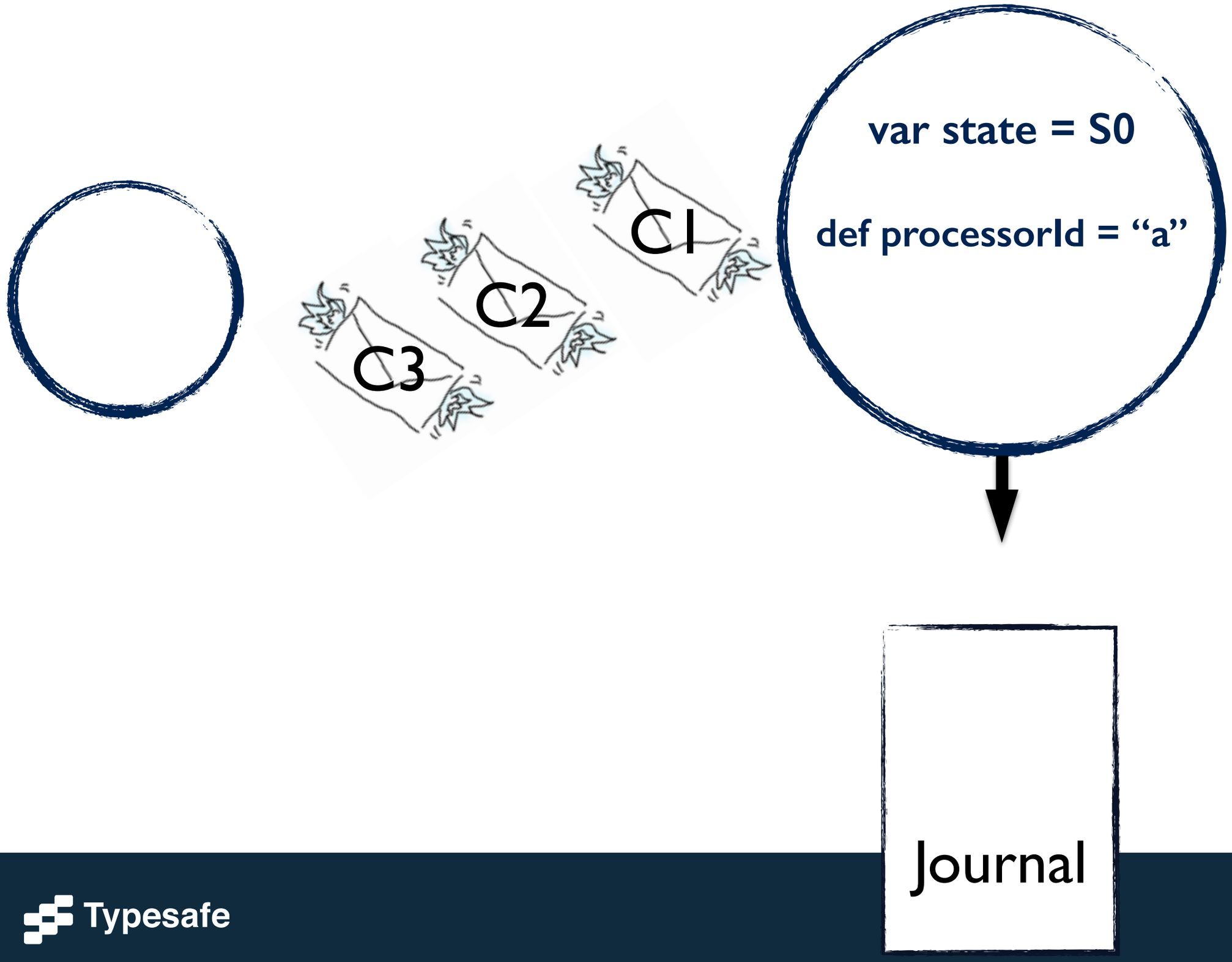
PersistentActor: persistAsync(){}

```
def receiveCommand = {


  case Mark(id) =>
   sender() ! InitMarking
   persistAsync(Marker) { m =>
     // update state...
   }

   defer(Marked(id)) { marked =>
     sender() ! marked
   }
}
```

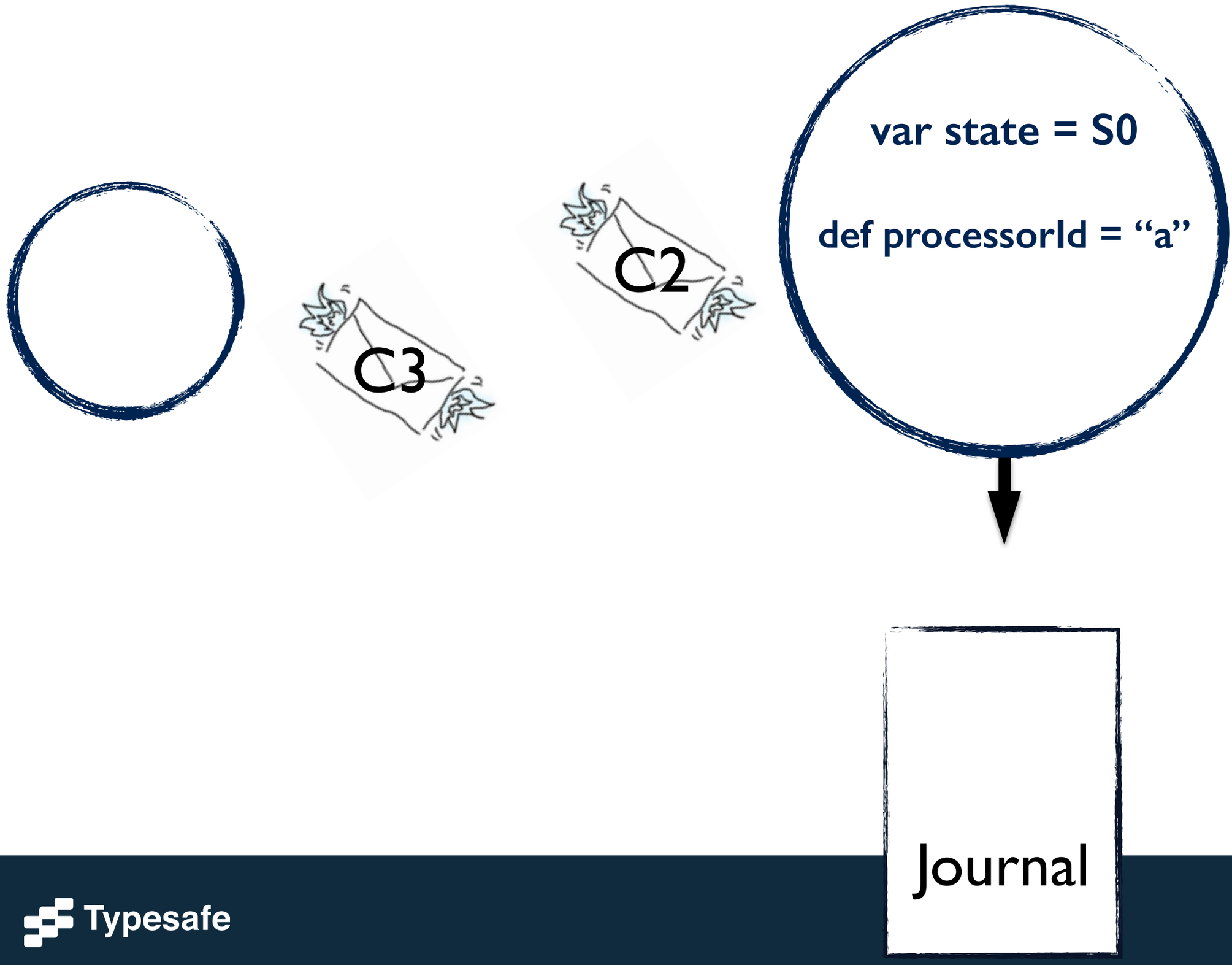
NOT persisted

execute once all
persistAsync handlers done

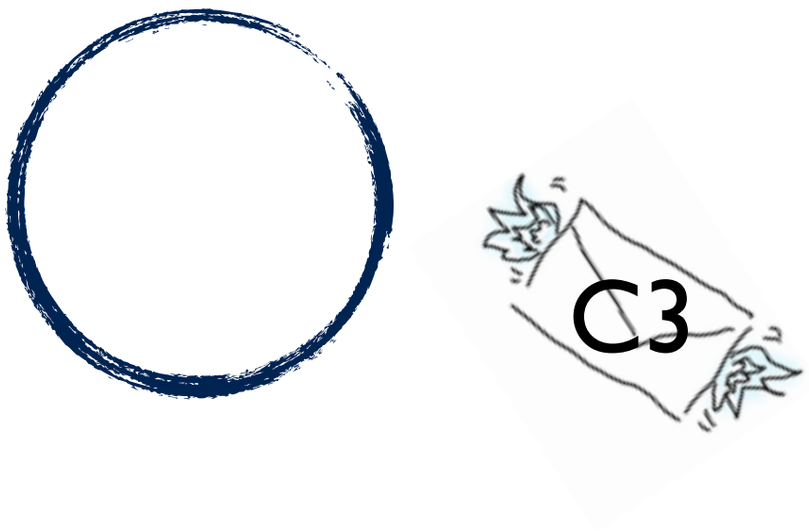
persistAsync(){} - Ordering guarantees

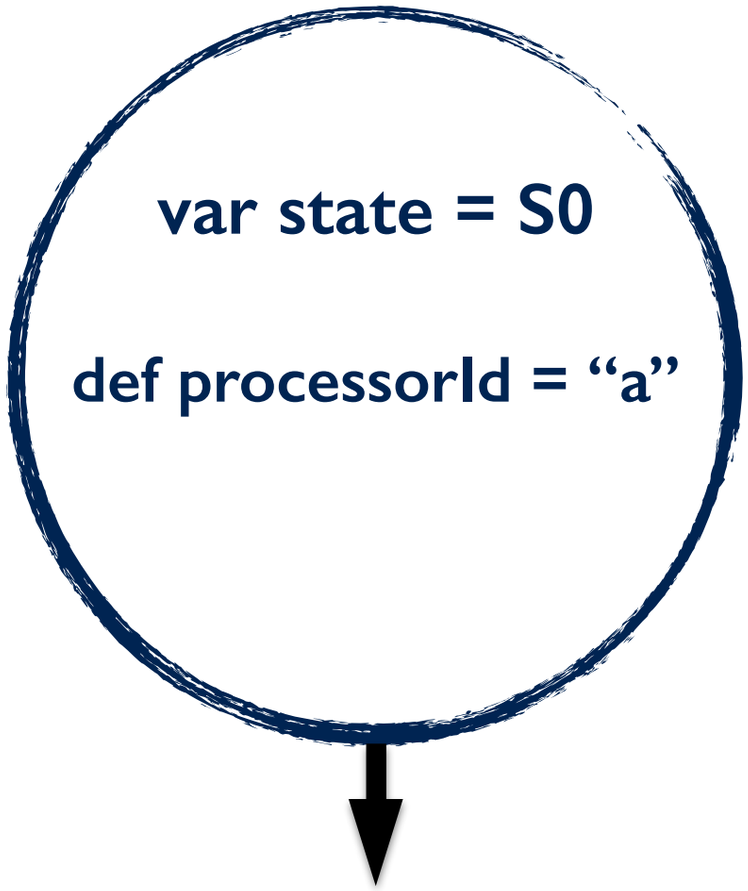


persistAsync(){} - Ordering guarantees

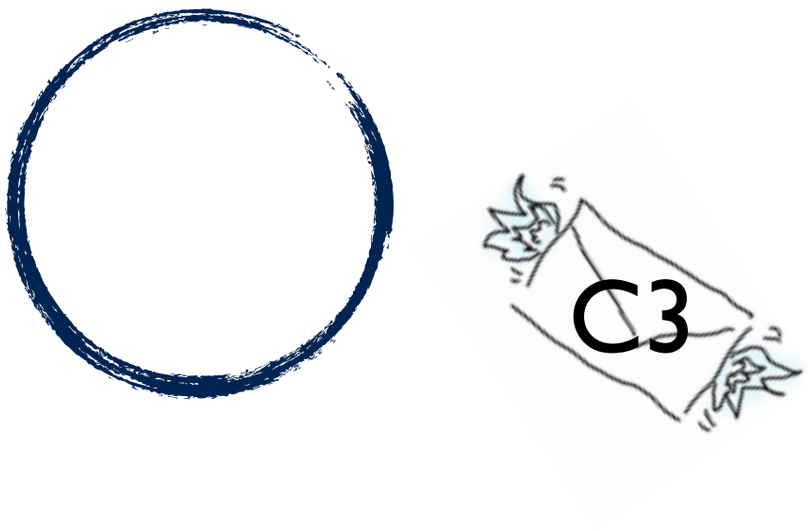


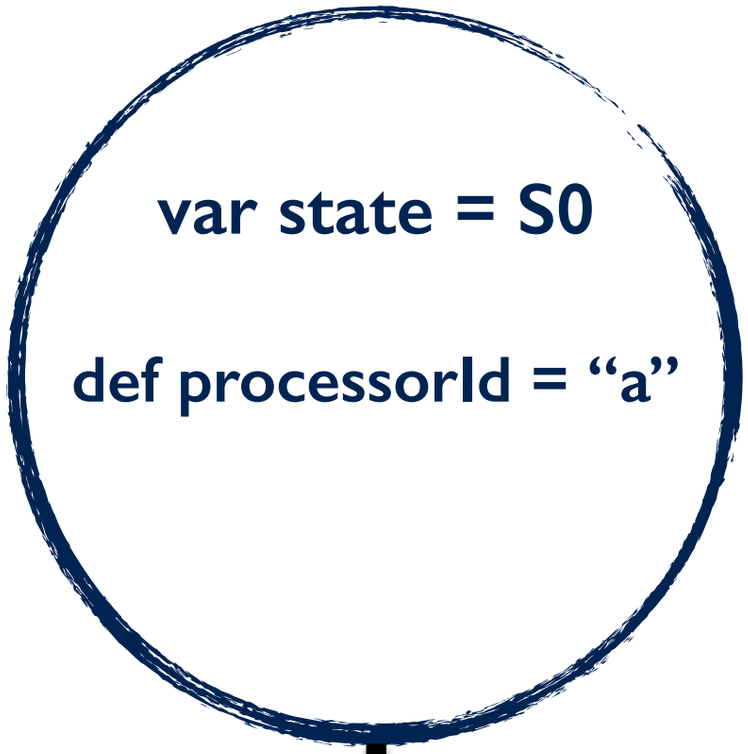
persistAsync(){} - Ordering guarantees



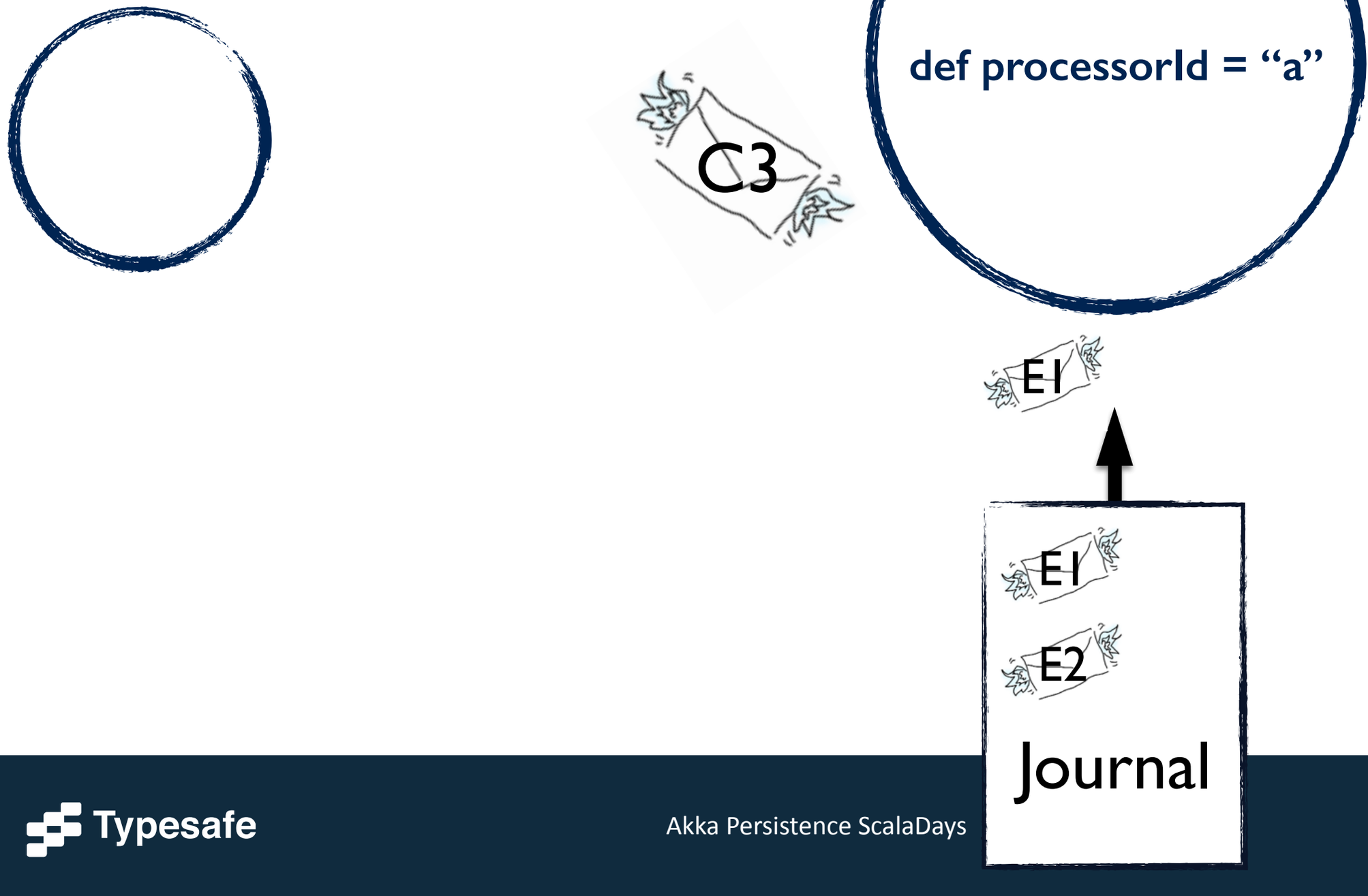


persistAsync(){} - Ordering guarantees

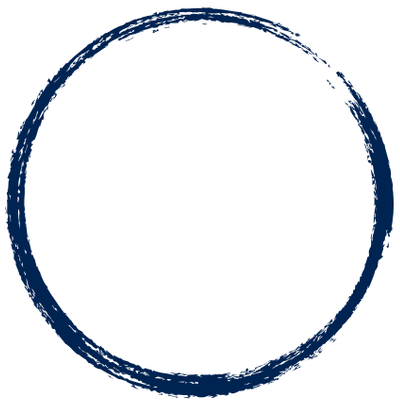


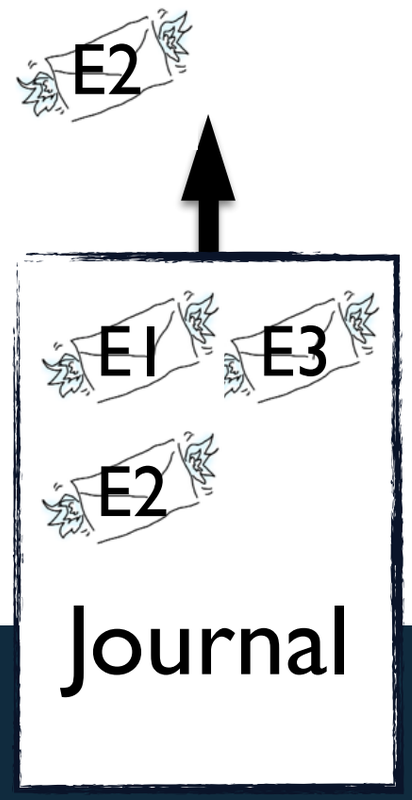


persistAsync(){} - Ordering guarantees

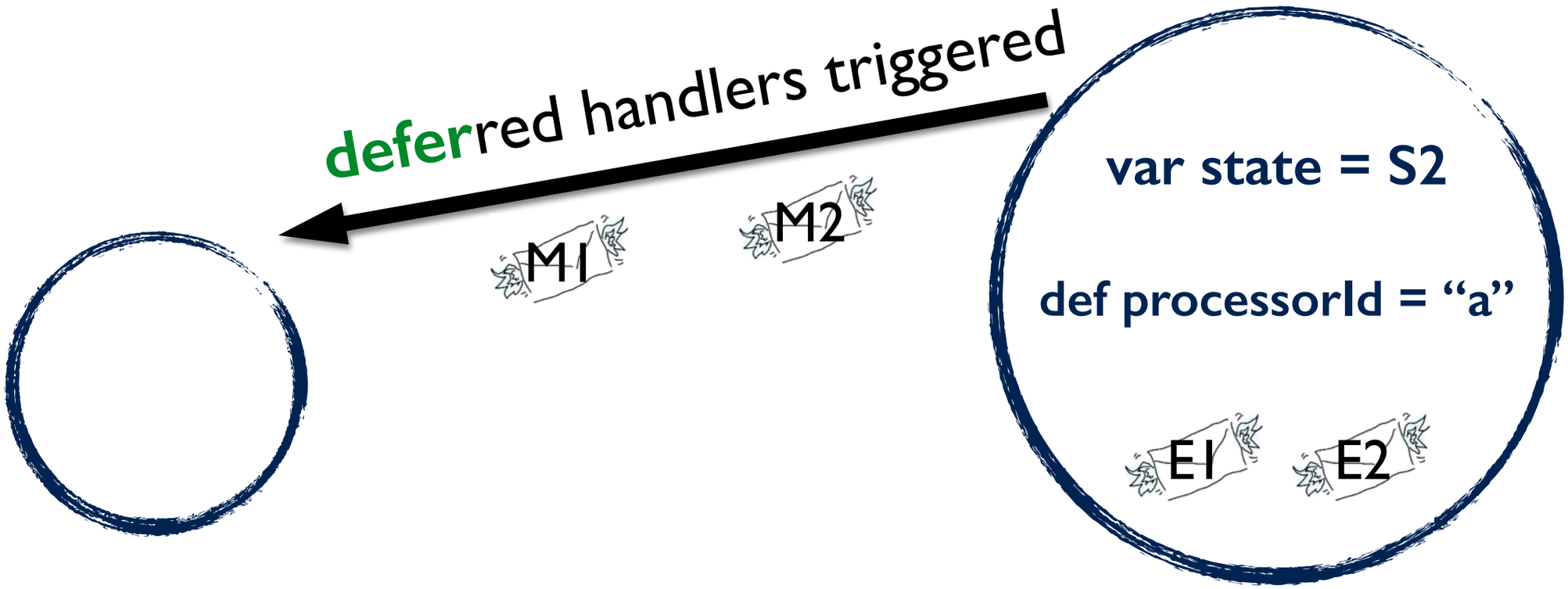


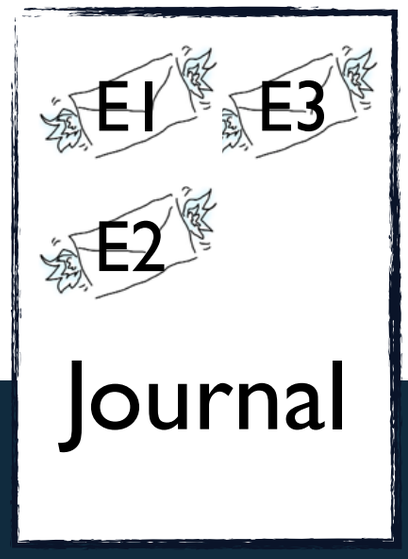
persistAsync(){} - Ordering guarantees





persistAsync(){} - Ordering guarantees





Recovery

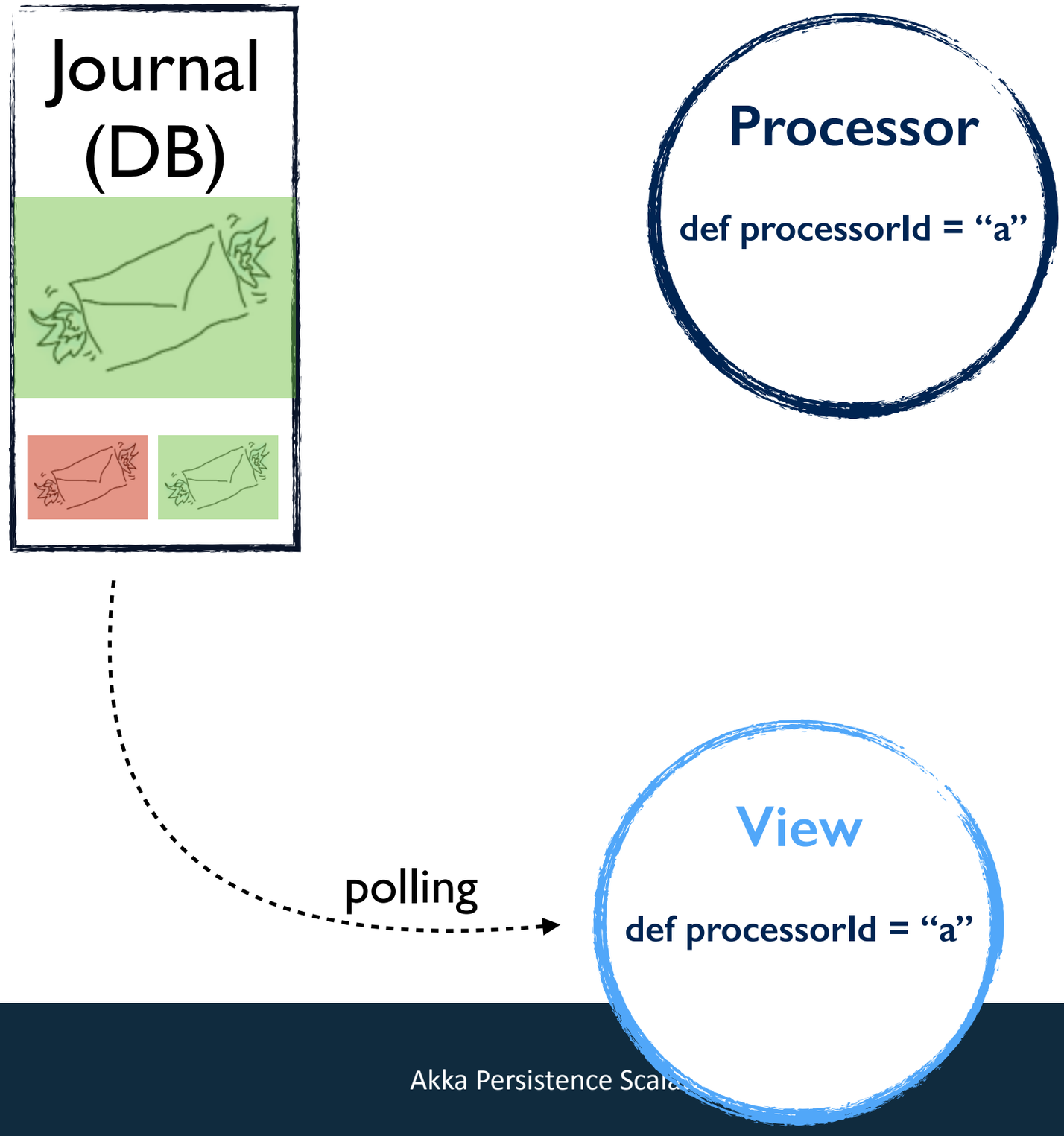
Eventsourced, recovery

```
/** MUST NOT SIDE-EFFECT! */
def receiveRecover = {
  case replayedEvent: Event  =>
    state = updateState(replayedEvent)
}
```

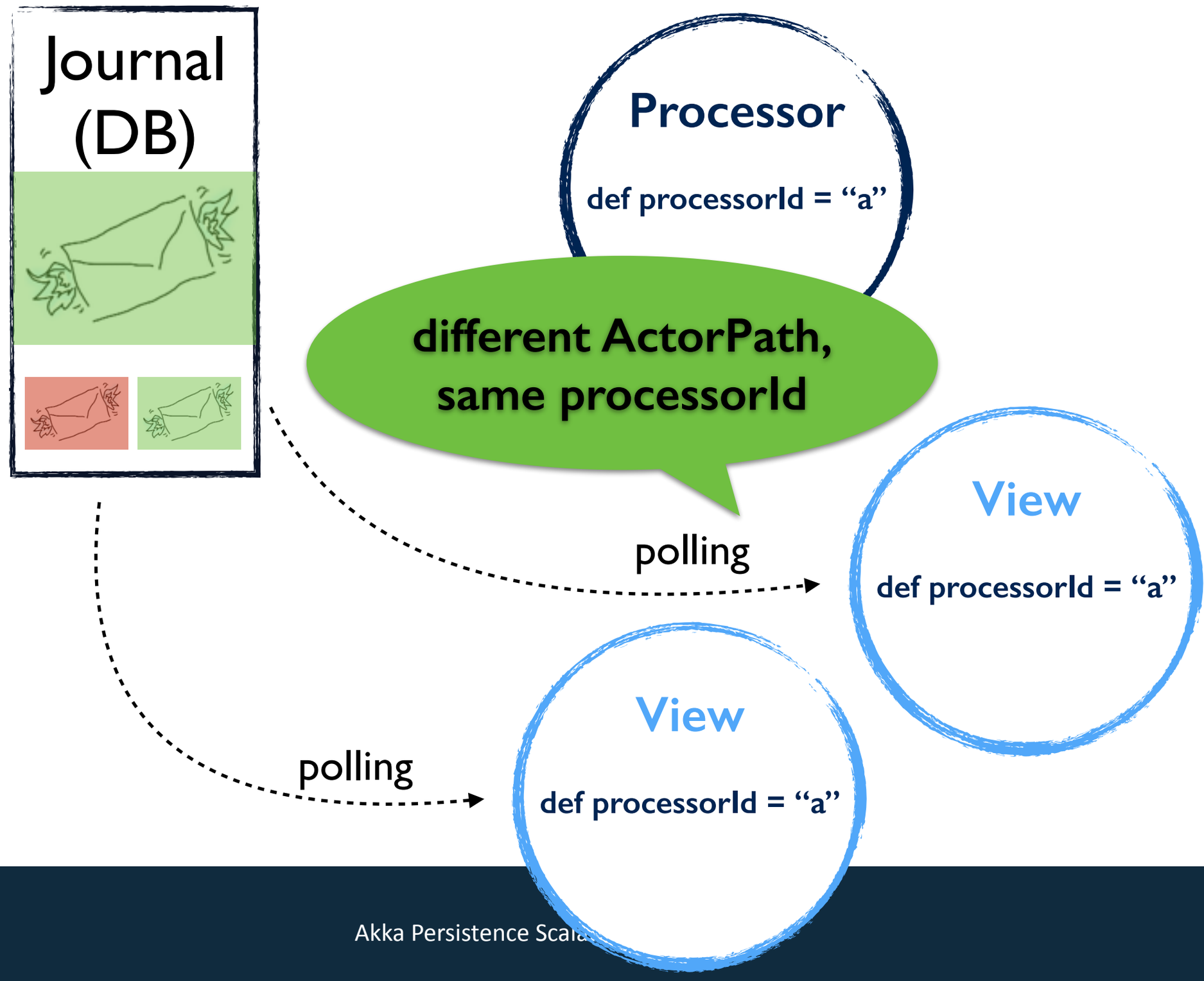
re-using *updateState*, as seen in *receiveCommand*

Views

Views


Journal
(DB)
Processor
def processorId = “a”
polling
View
def processorId = “a”

Views


Journal
(DB)
Processor
def processorId = “a”
different ActorPath,
same processorId
polling
View
def processorId = “a”
polling
View
def processorId = “a”

View

```
class DoublingCounterProcessor extends View {
  var state = 0

  override val processorId = "counter"

  def receive = {
    case Persistent(payload, seqNr) =>
      // “state += 2 * payload”

  }
}
```

subject to change!

Views, as Reactive Streams

View, as ReactiveStream

early preview

pull request
by *krasserm*

```
// Imports ...

import org.reactivestreams.api.Producer

import akka.stream._
import akka.stream.scaladsl.Flow
import akka.persistence._
import akka.persistence.stream._


val materializer = FlowMaterializer(MaterializerSettings())
```

View, as ReactiveStream

early preview

pull request
by *krasserm*

```
// 1 producer and 2 consumers:
val p1: Producer[Persistent] = PersistentFlow.
  fromProcessor(“processor-1").
  toProducer(materializer)

Flow(p1).
  foreach(p => println(s"consumer-1: ${p.payload}”)).
  consume(materializer)

Flow(p1).
  foreach(p => println(s"consumer-2: ${p.payload}”)).
  consume(materializer)
```

View, as ReactiveStream

early preview

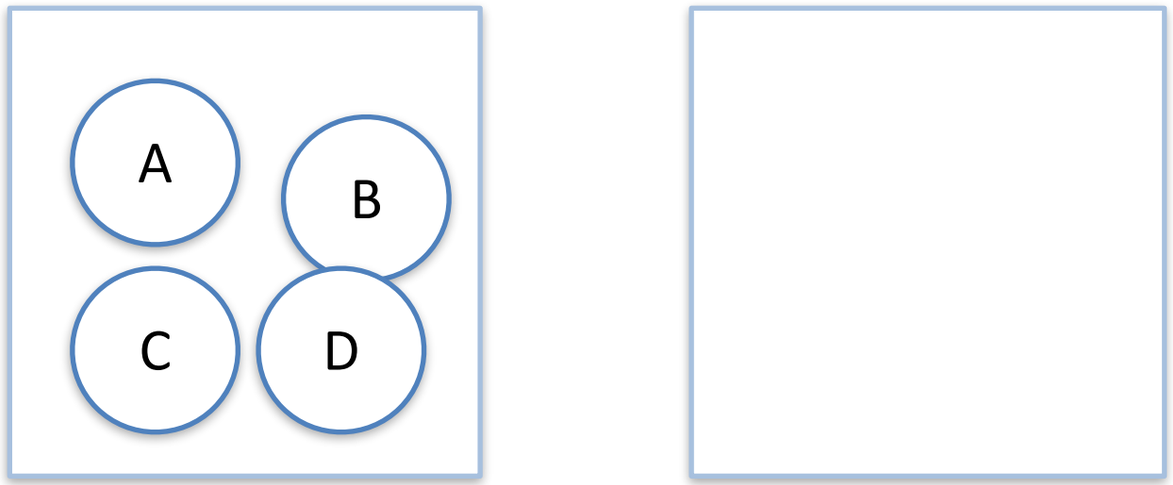
pull request
by *krasserm*

```
// 2 producers (merged) and 1 consumer:
val p2: Producer[Persistent] = PersistentFlow.
  fromProcessor("processor-2").
  toProducer(materializer)

val p3: Producer[Persistent] = PersistentFlow.
  fromProcessor("processor-3").
  toProducer(materializer)

Flow(p2).merge(p3). // triggers on "either"
  foreach { p => println(s"consumer-3: ${p.payload}") }.
  consume(materializer)
```

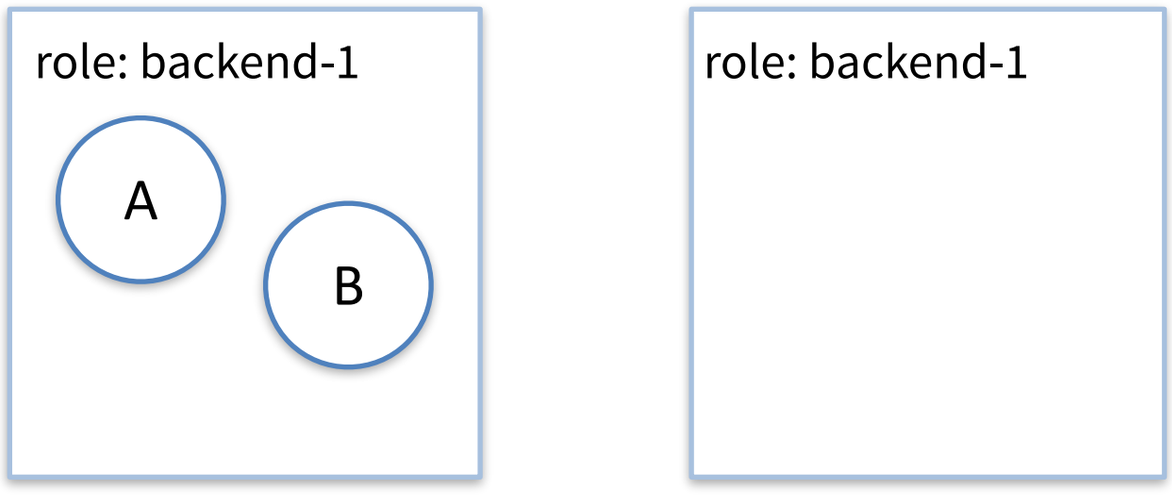
Usage in a Cluster

- distributed journal (http://akka.io/community/)
  - Cassandra
  - DynamoDB
  - HBase
  - MongoDB
  - shared LevelDB journal for testing
- single writer
  - cluster singleton
  - cluster sharding

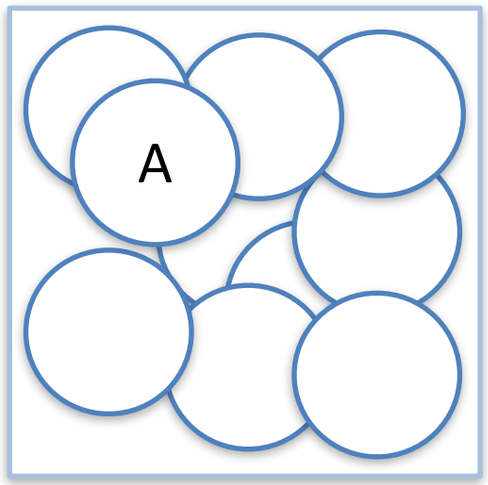
Cluster Singleton

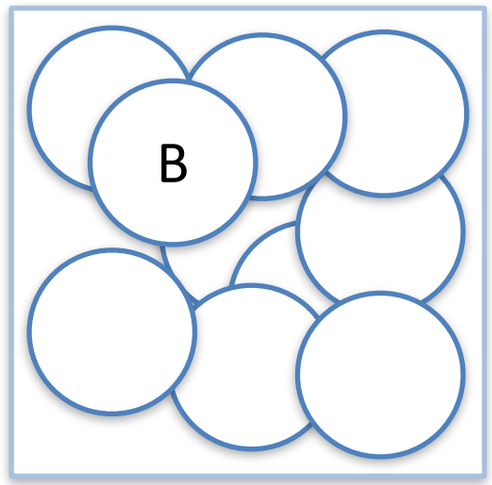


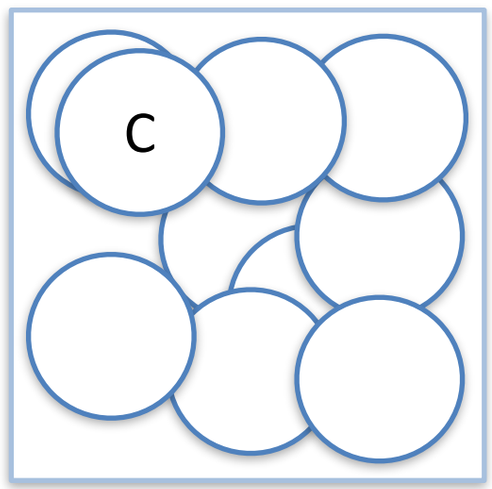
Cluster Singleton

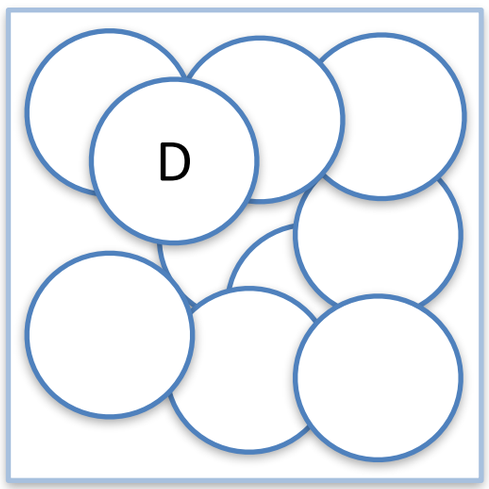


Cluster Sharding

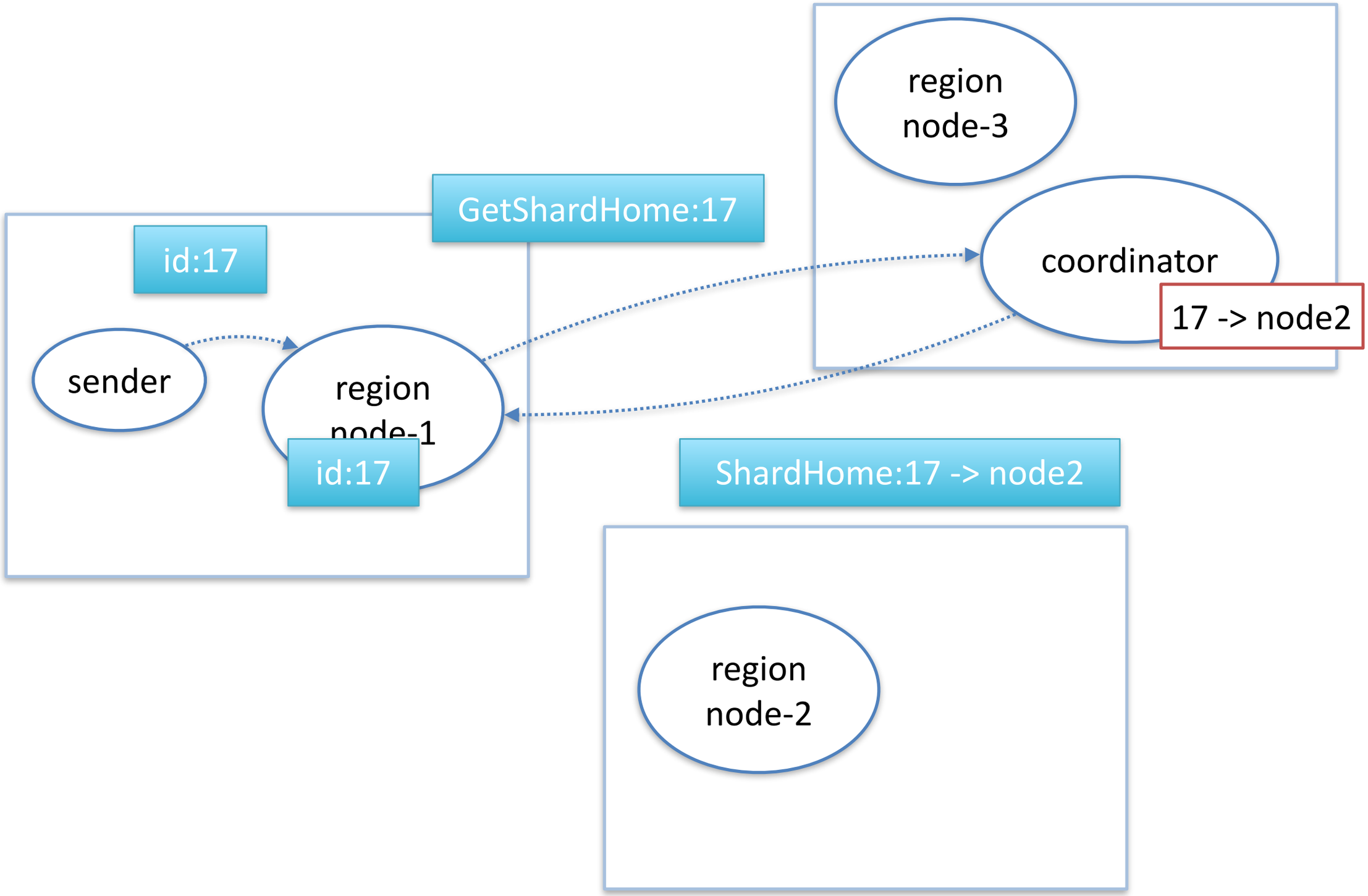




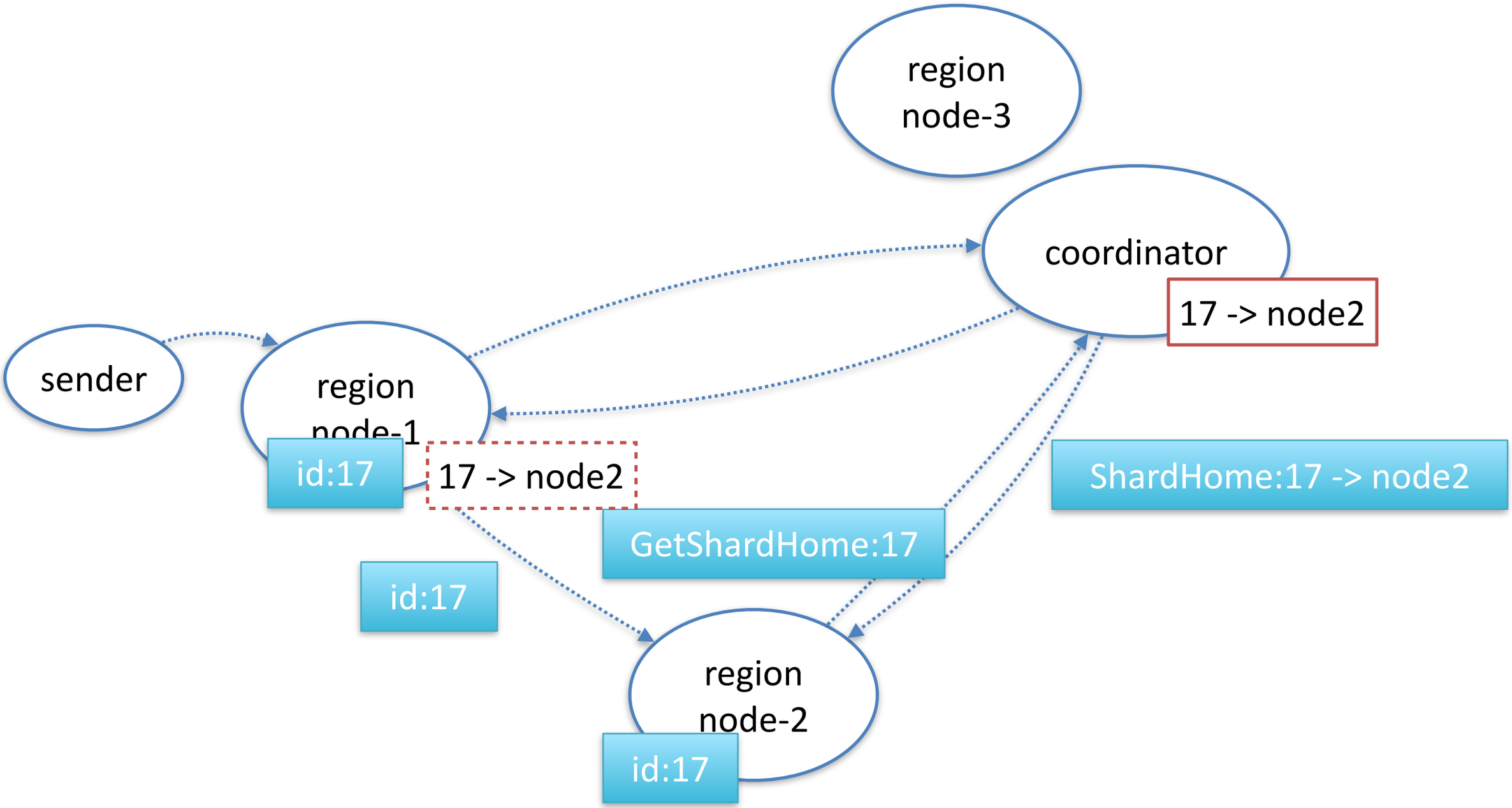




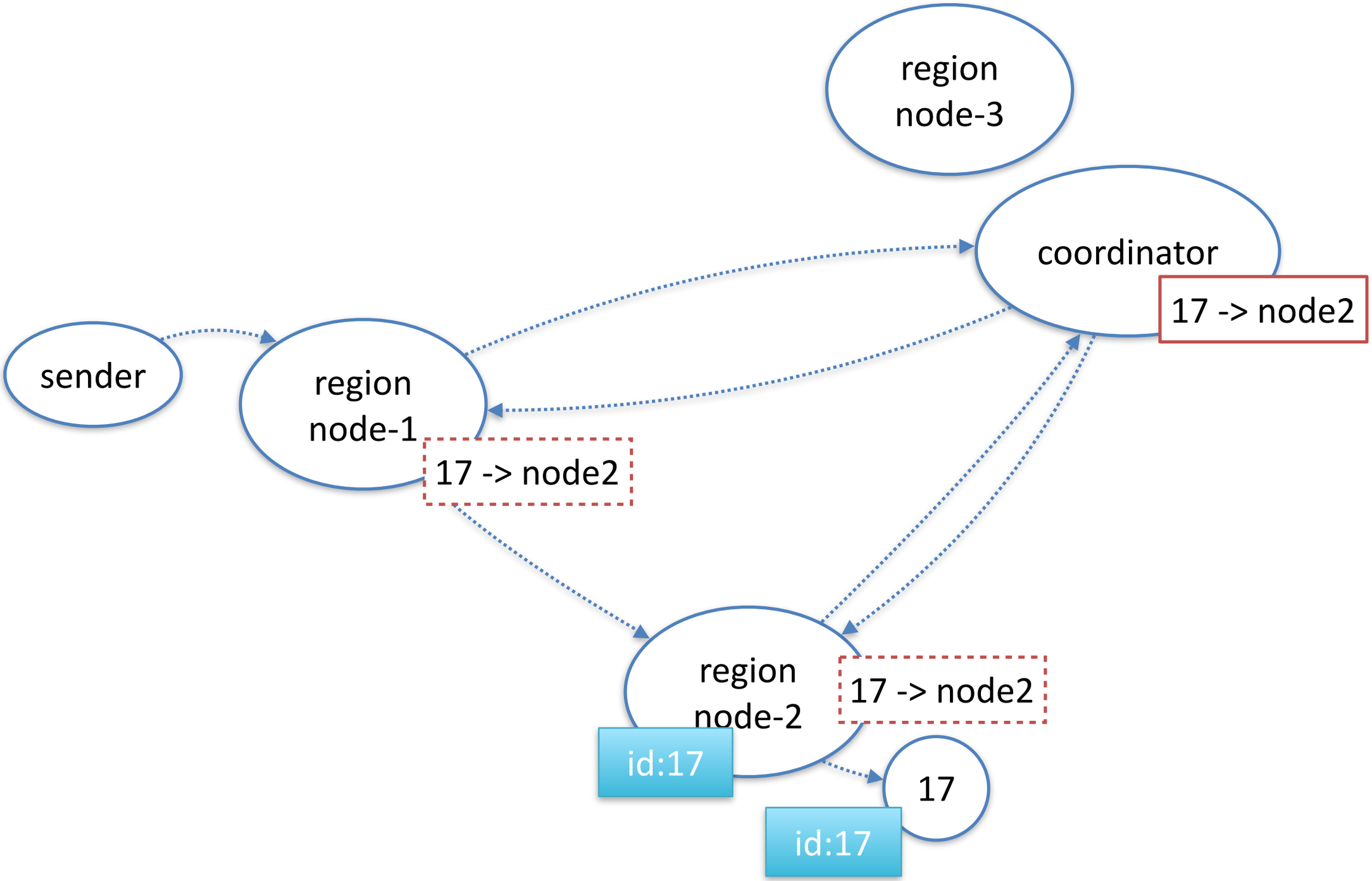
Cluster Sharding



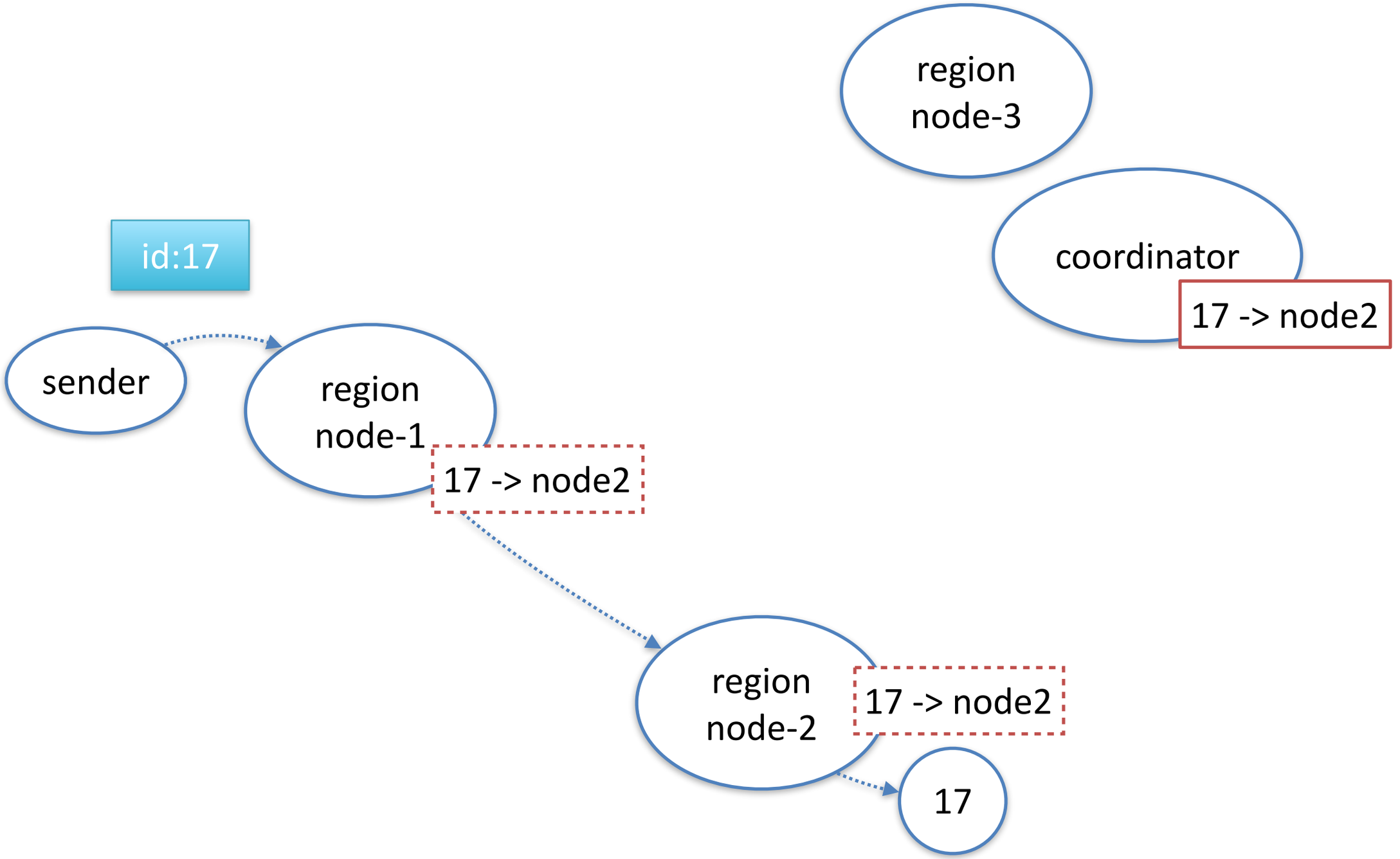
Cluster Sharding



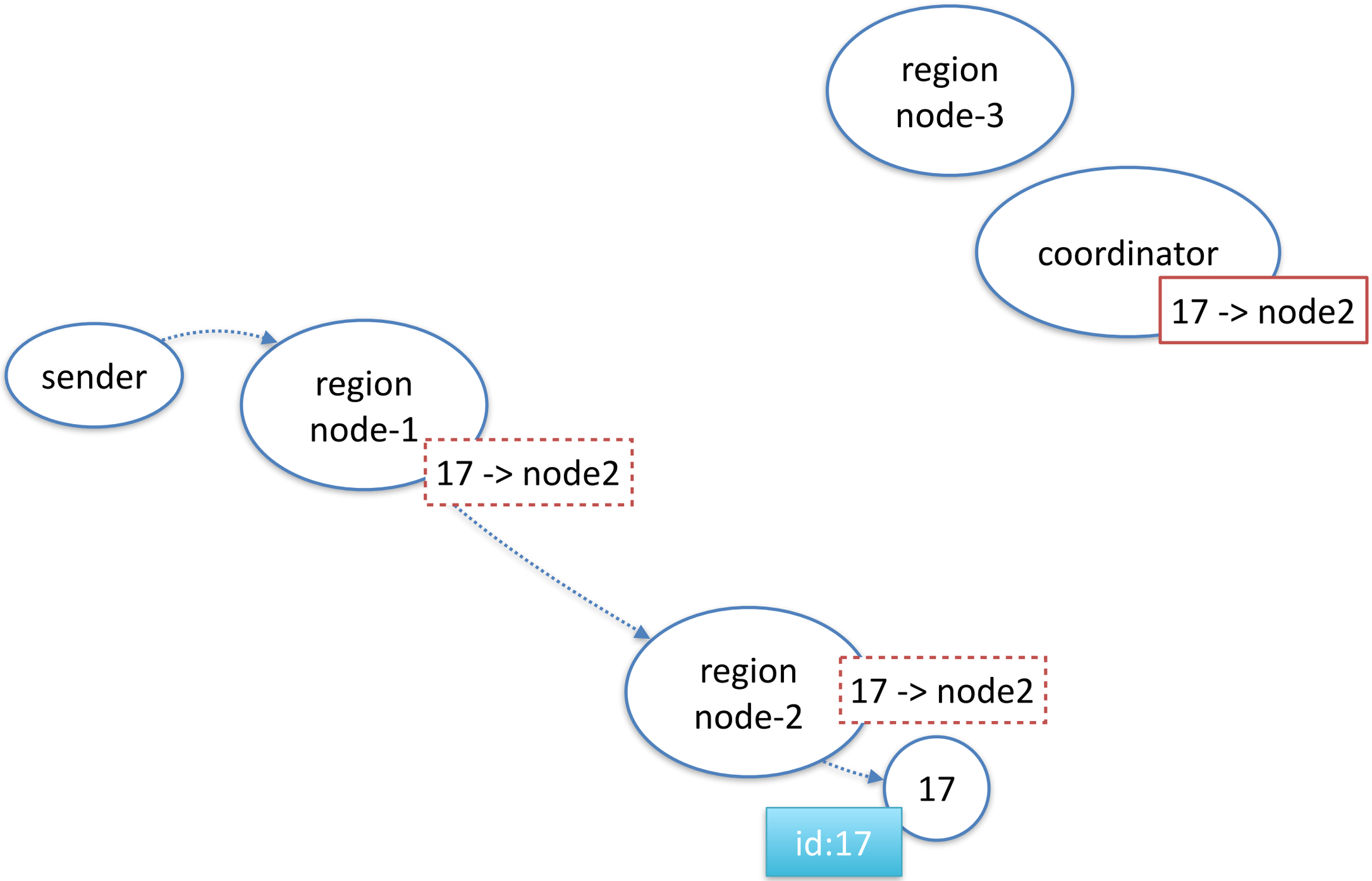
Cluster Sharding



Cluster Sharding



Cluster Sharding



Cluster Sharding

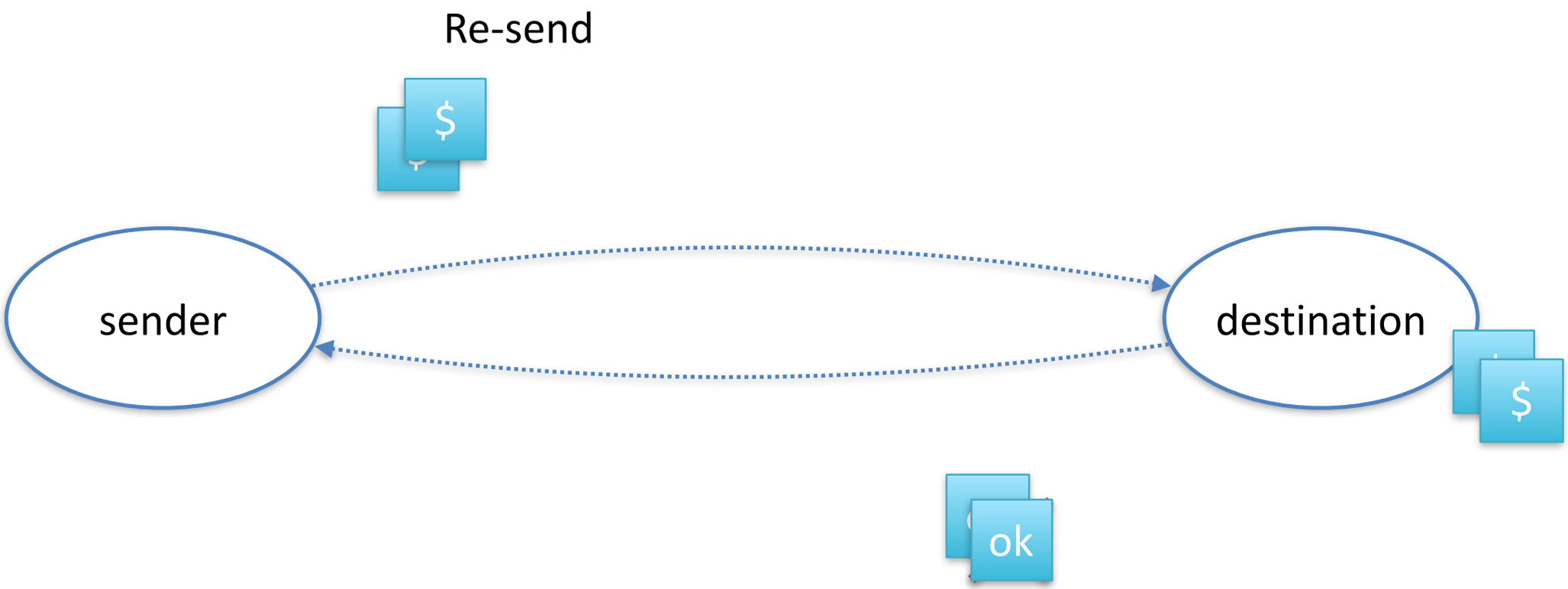
```
val idExtractor: ShardRegion.IdExtractor = {
  case cmd: Command => (cmd.postId, cmd)
}

val shardResolver: ShardRegion.ShardResolver = msg => msg match {
  case cmd: Command => (math.abs(cmd.postId.hashCode) % 100).toString
}
```

```
ClusterSharding(system).start(
  typeName = BlogPost.shardName,
  entryProps = Some(BlogPost.props()),
  idExtractor = BlogPost.idExtractor,
  shardResolver = BlogPost.shardResolver)
```

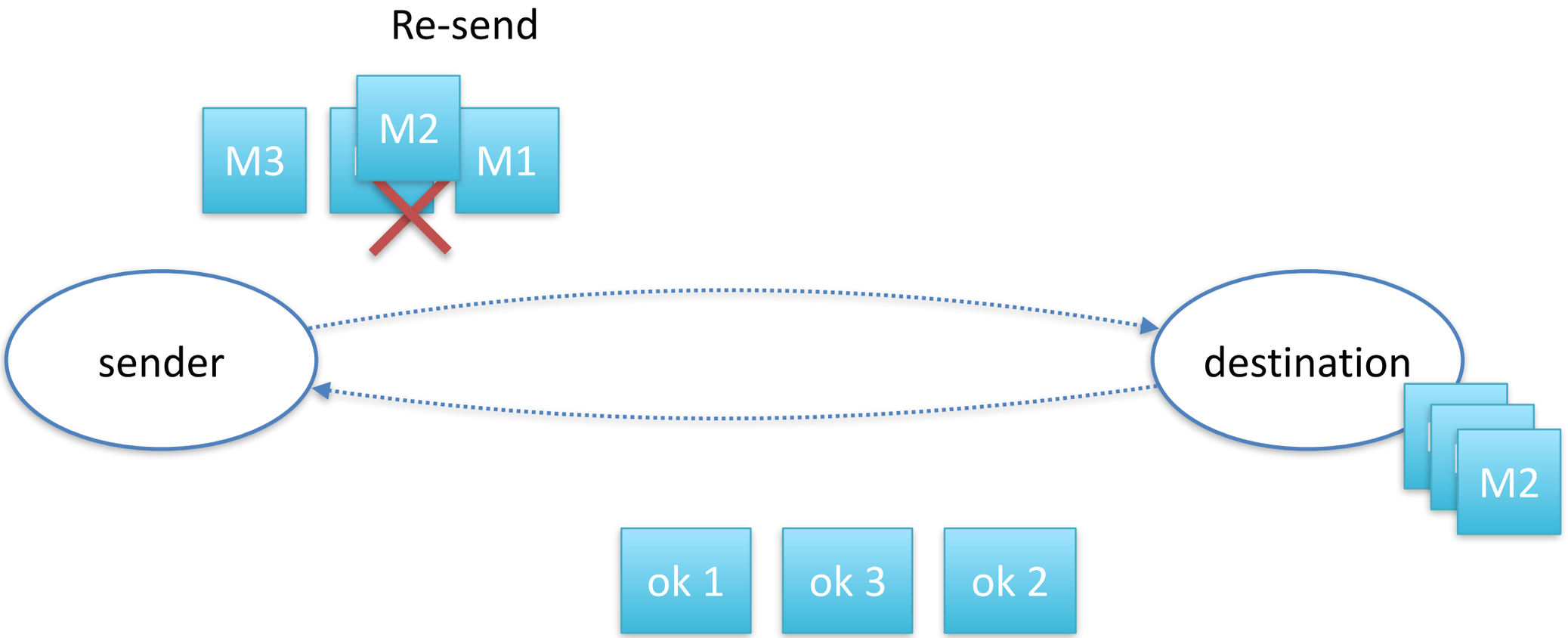
```
val blogPostRegion: ActorRef =
  ClusterSharding(context.system).shardRegion(BlogPost.shardName)

val postId = UUID.randomUUID().toString
blogPostRegion ! BlogPost.AddPost(postId, author, title)
```

Lost messages

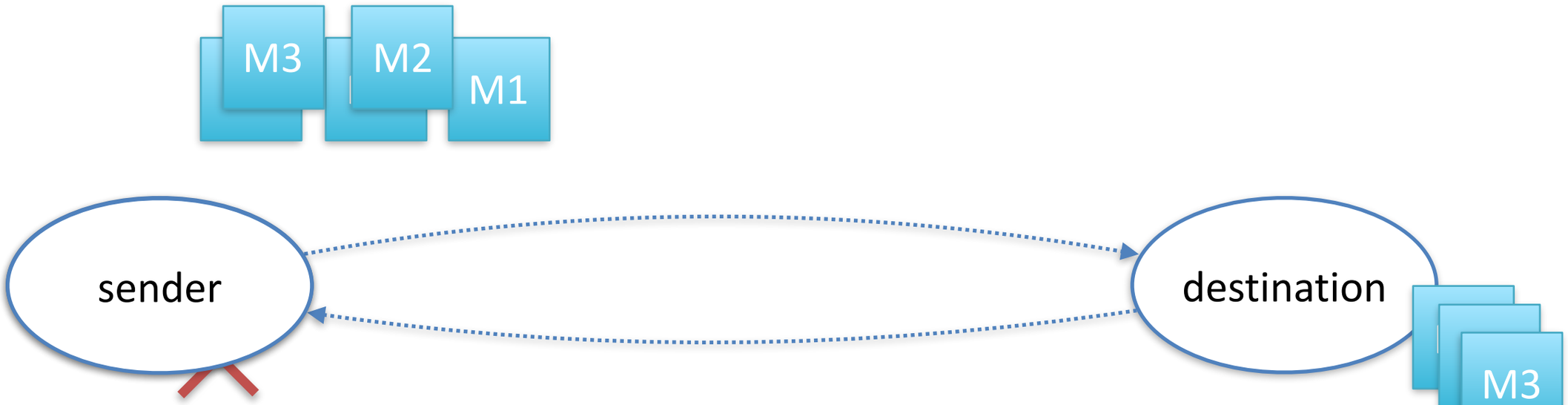
At-least-once delivery - duplicates



At-least-once delivery - unordered



At-least-once delivery - crash



1. Sent M1
2. Sent M2
3. Sent M3
4. M1 Confirmed
5. M2 Confirmed
6. M3 Confirmed

ok 1 ok 2 ok 3

PersistentActor with AtLeastOnceDelivery

```scala
case class Msg(deliveryId: Long, s: String)
case class Confirm(deliveryId: Long)
sealed trait Evt
case class MsgSent(s: String) extends Evt
case class MsgConfirmed(deliveryId: Long) extends Evt
```

```scala
class Sender(destination: ActorPath)
  extends PersistentActor with AtLeastOnceDelivery {

  def receiveCommand: Receive = {
    case s: String           => persist(MsgSent(s))(updateState)
    case Confirm(deliveryId) => persist(MsgConfirmed(deliveryId))(updateState)
  }

  def receiveRecover: Receive = { case evt: Evt => updateState(evt) }

  def updateState(evt: Evt): Unit = evt match {
    case MsgSent(s) =>
      deliver(destination, deliveryId => Msg(deliveryId, s))

    case MsgConfirmed(deliveryId) => confirmDelivery(deliveryId)
  }
}
```

Next step

- Documentation
  - http://doc.akka.io/docs/akka/2.3.3/scala/persistence.html
  - http://doc.akka.io/docs/akka/2.3.3/java/persistence.html
  - http://doc.akka.io/docs/akka/2.3.3/contrib/cluster-sharding.html
- Typesafe Activator
  - https://typesafe.com/activator/template/akka-sample-persistence-scala
  - https://typesafe.com/activator/template/akka-sample-persistence-java
  - http://typesafe.com/activator/template/akka-cluster-sharding-scala
- Mailing list
  - http://groups.google.com/group/akka-user
- Migration guide from Eventsourced
  - http://doc.akka.io/docs/akka/2.3.3/project/migration-guide-eventsourced-2.3.x.html