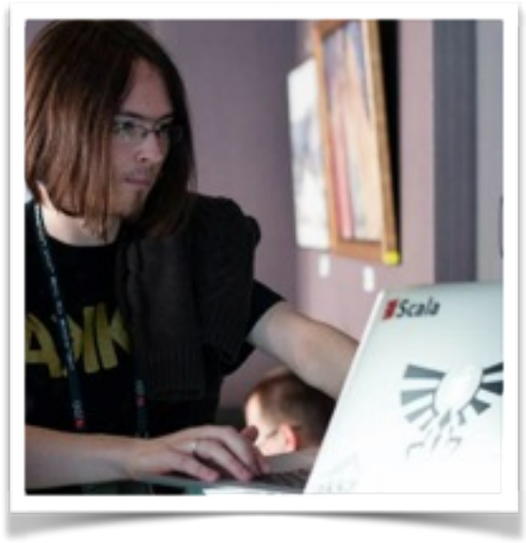
DDDing with Akka Persistence

Konrad `@ktosopl` Malawski

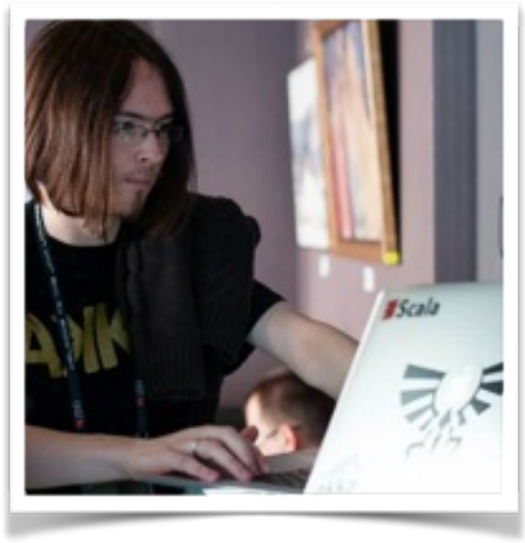
Typesafe

hAkker @ Typesafe



Konrad `@ktosopl` Malawski

hAkker @ Typesafe



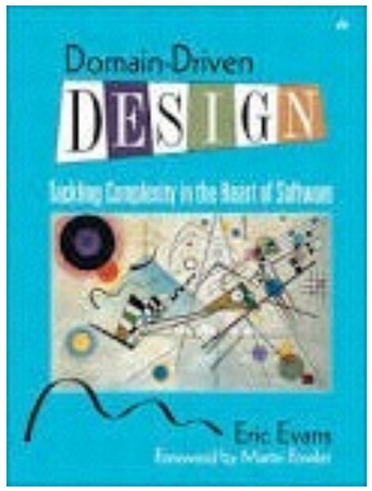
Konrad `@ktosopl` Malawski

typesafe.com
geecon.org
Java.pl / KrakowScala.pl
sckrk.com / meetup.com/Paper-Cup @ London
GDGKrakow.pl
meetup.com/Lambda-Lounge-Krakow

Show of hands!

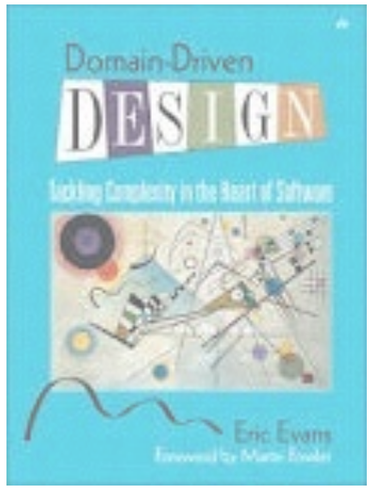
Show of hands!

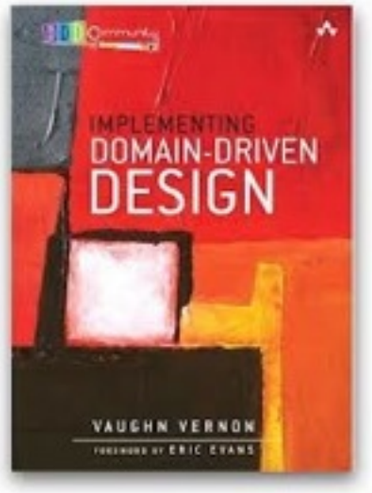
Show of hands!



Show of hands!

akka





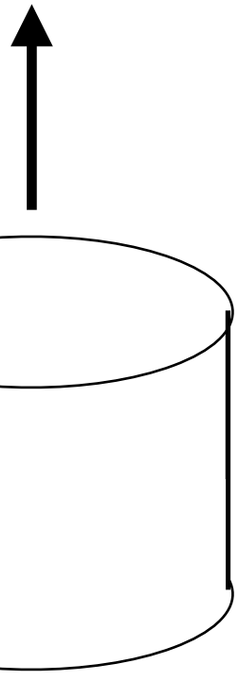
sourcing styles

Command Sourcing	**Event** Sourcing
msg: **DoThing**	
msg persisted before receive	
imperative, *“do the thing”*	
business logic change, can be reflected in *reaction*	
no validation before persisting	

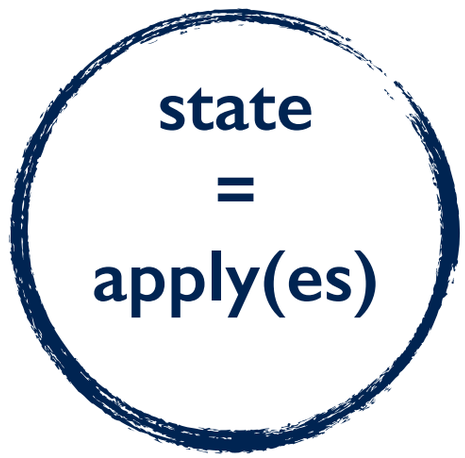
sourcing styles

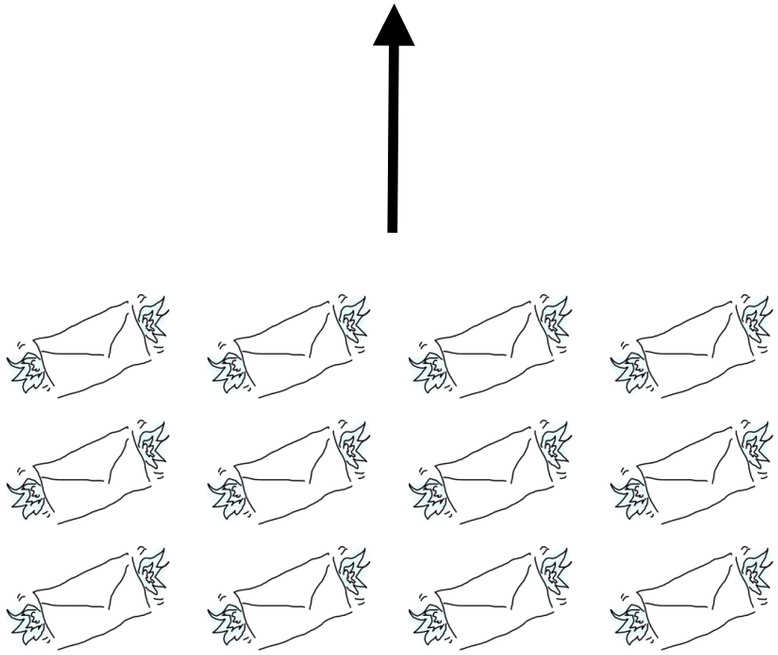
Command Sourcing	**Event** Sourcing
msg: **DoThing**	msg: **ThingDone**
msg persisted before receive	commands **converted** to events, must be **manually persisted**
imperative, *“do the thing”*	past tense, *“happened”*
business logic change, can be reflected in *reaction*	business logic change, won’t change previous events
Processor	validate, then maybe persist

Compared to “Good ol’ CRUD Model”



“Mutable Record”



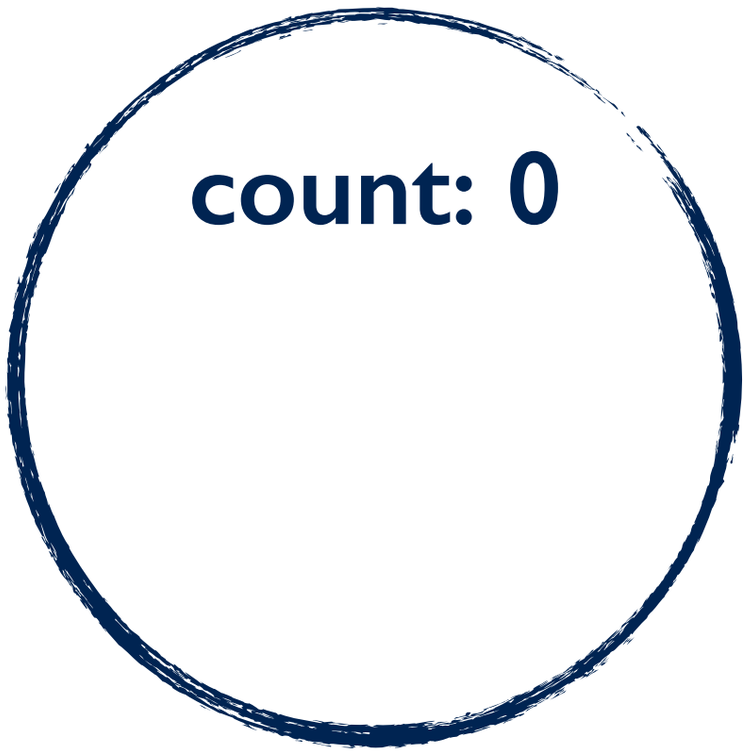


“Series of Events”

Actors

Actor

An **Actor** that keeps **count** of **messages** it processed

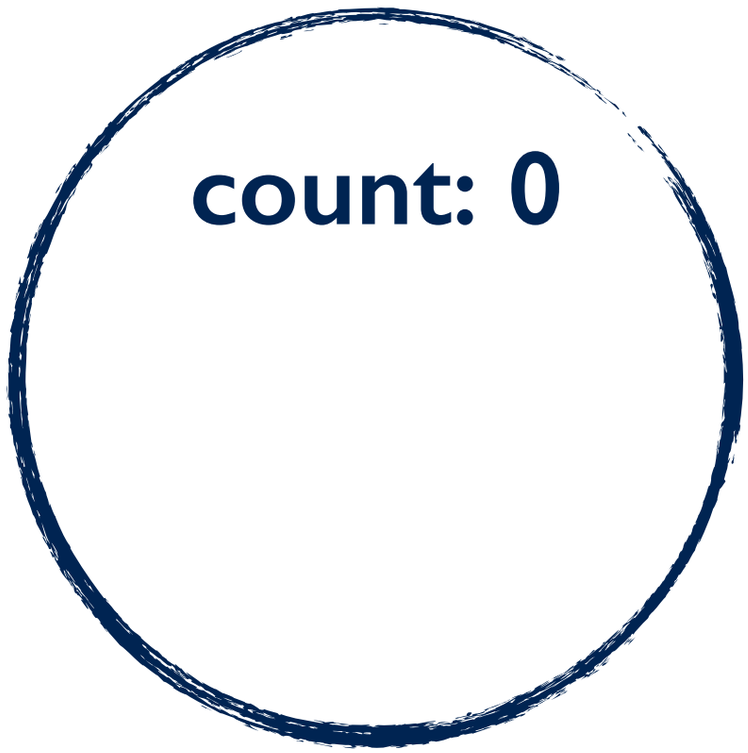


Let's send **2 messages** to it
(it's "commands")

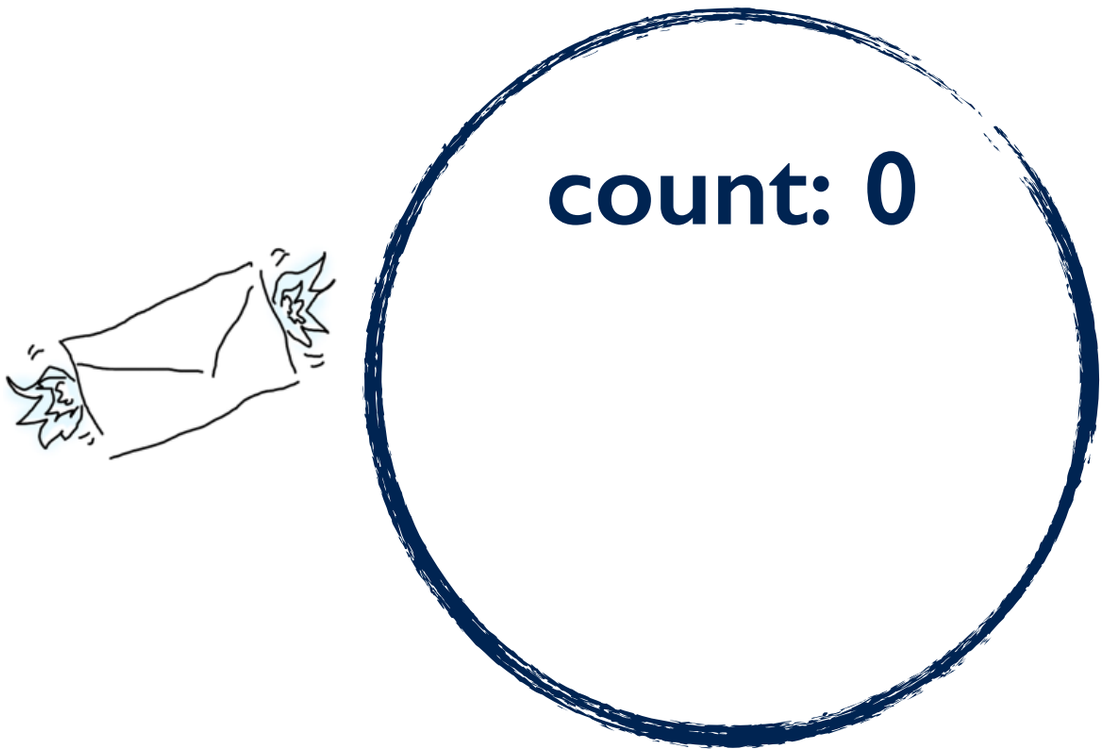
Actor

```
class Counter extends Actor {
  var count = 0

  def receive = {
    case _ => count += 1
  }
}
```

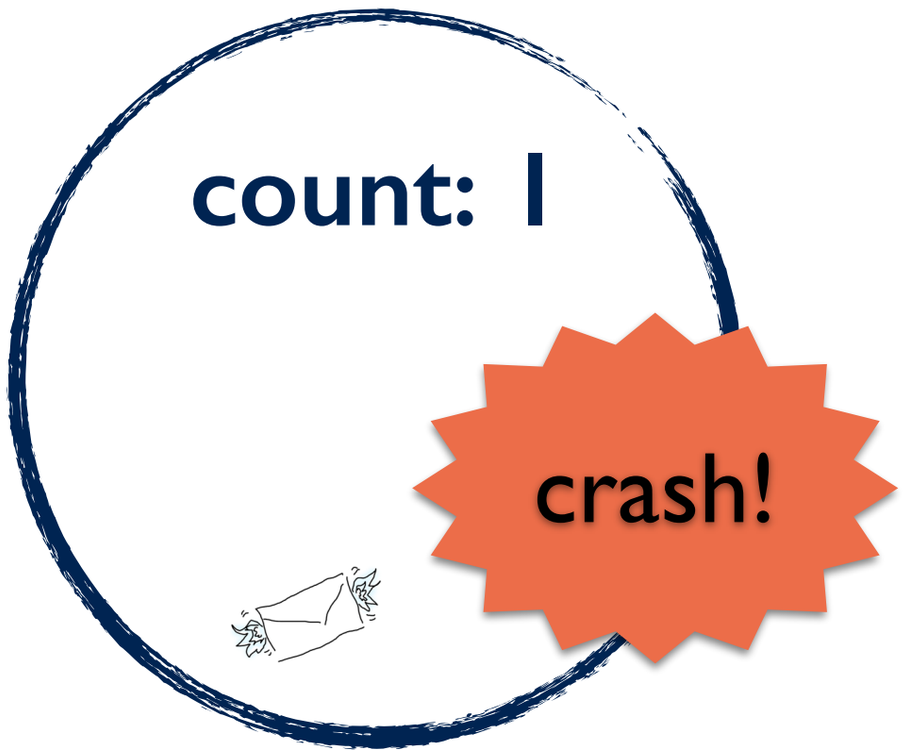
Actor



Actor



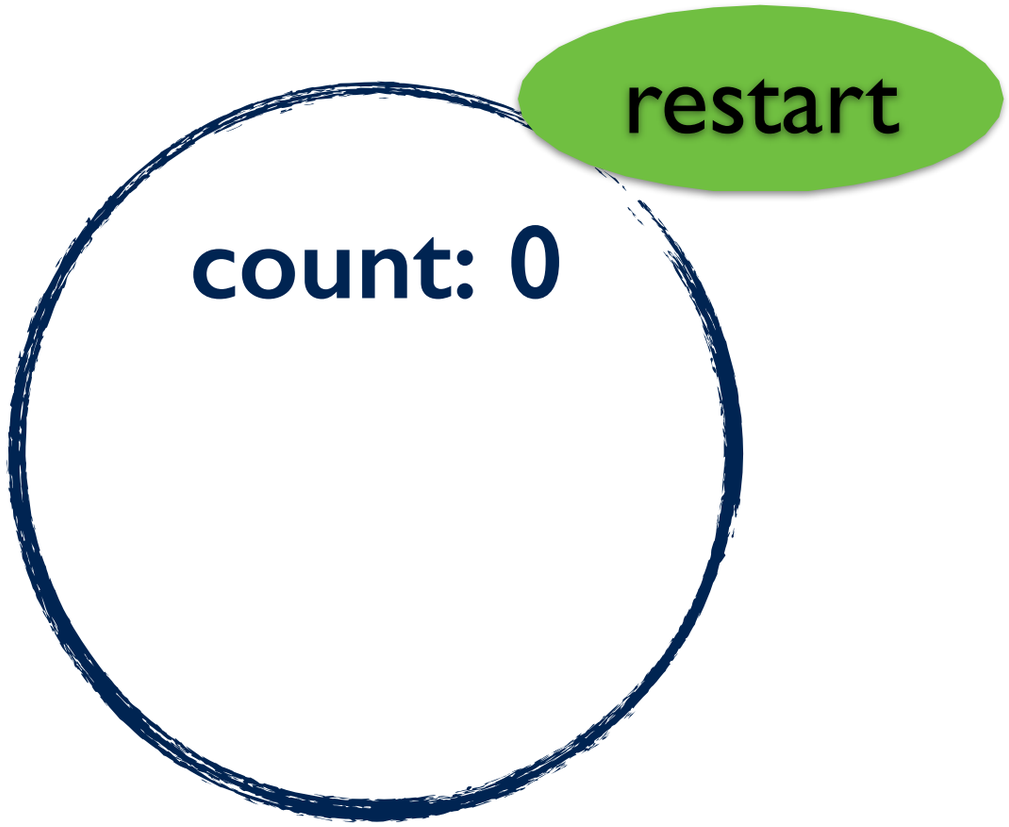
Actor



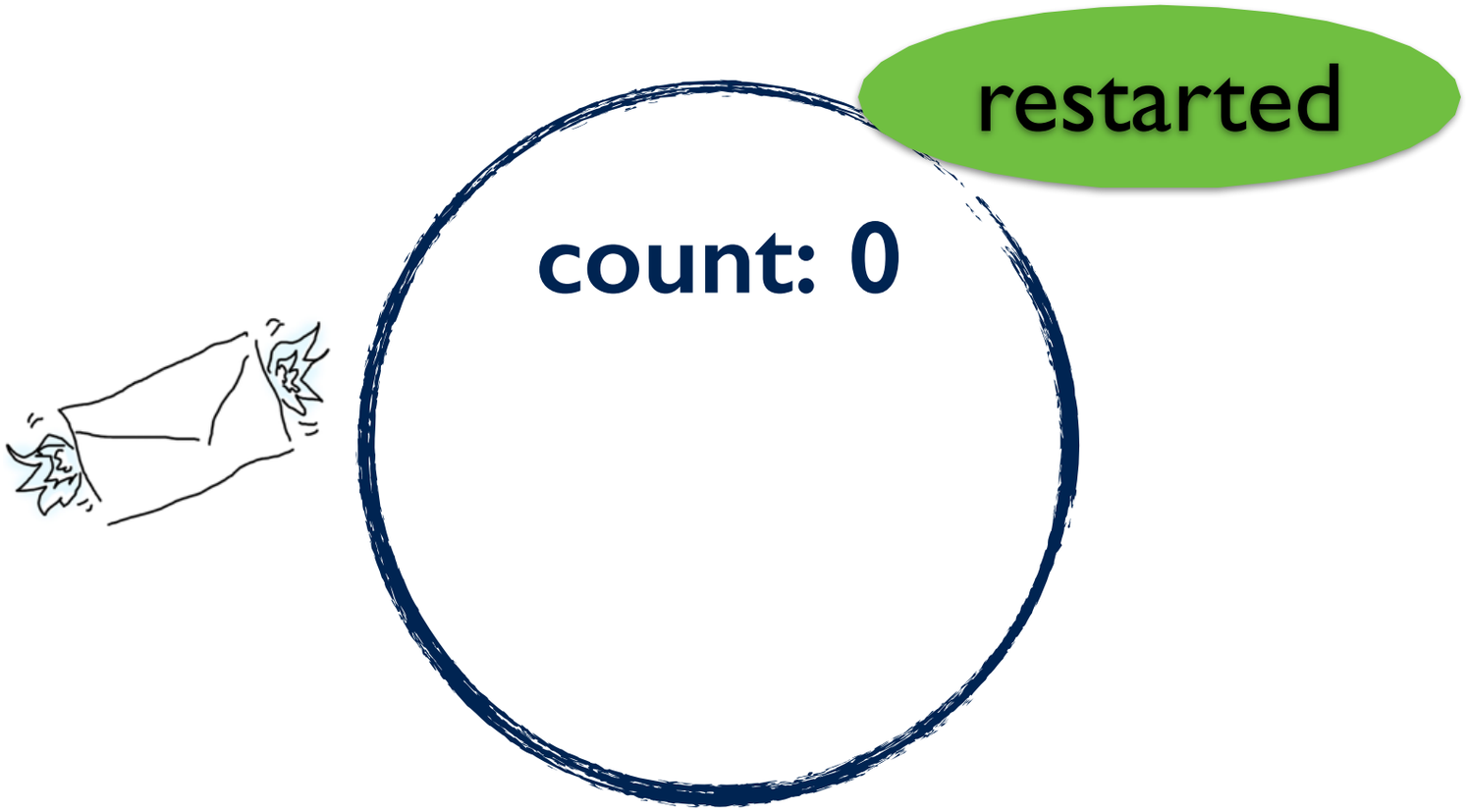
Actor

Actor

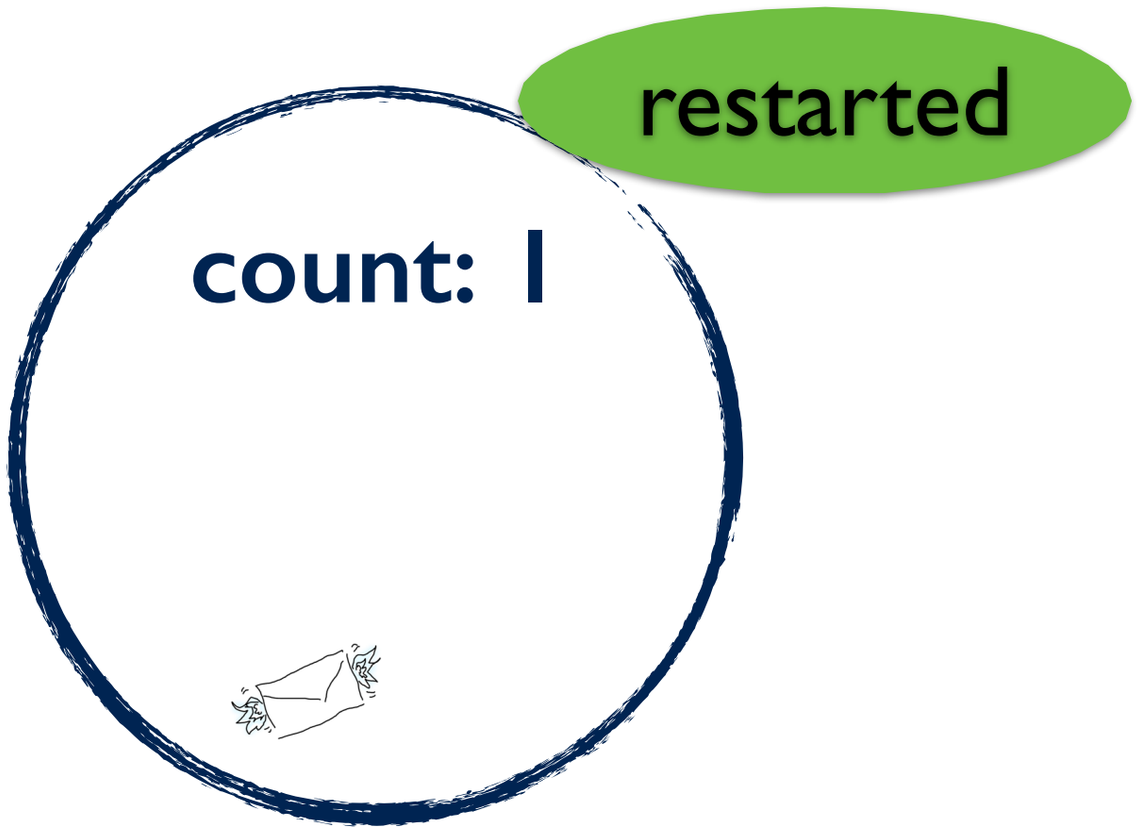
Actor



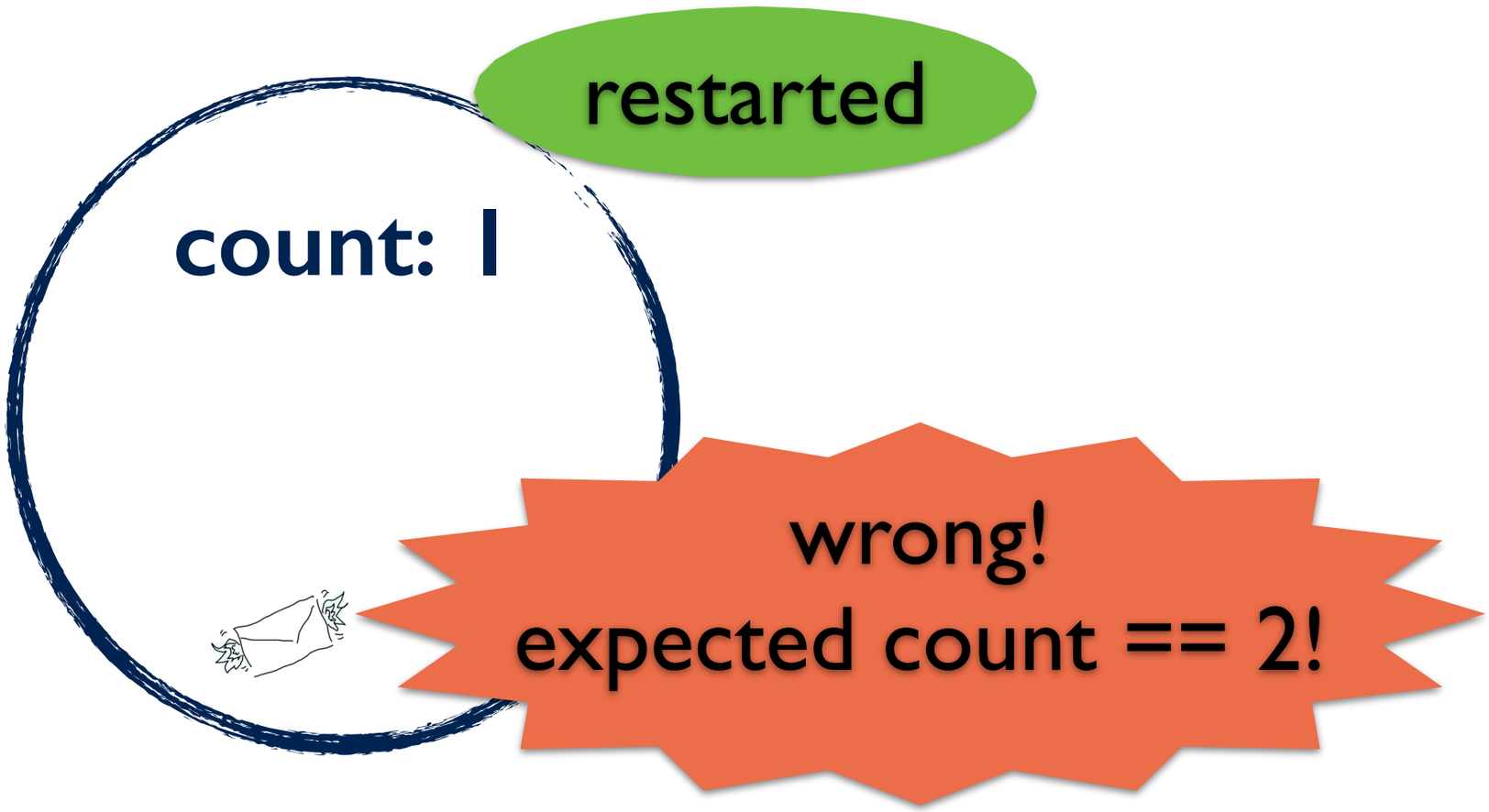
Actor



Actor



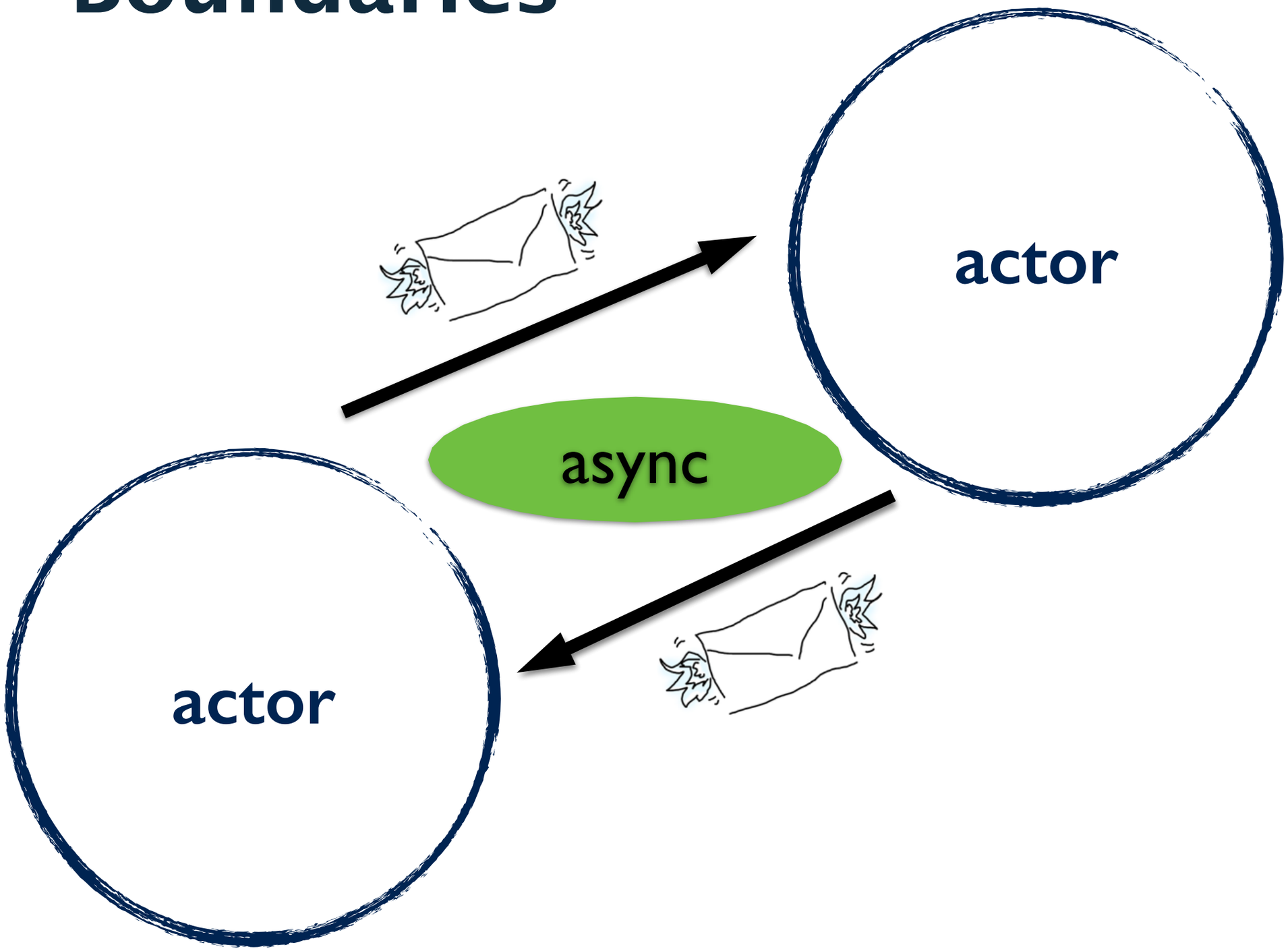
Actor



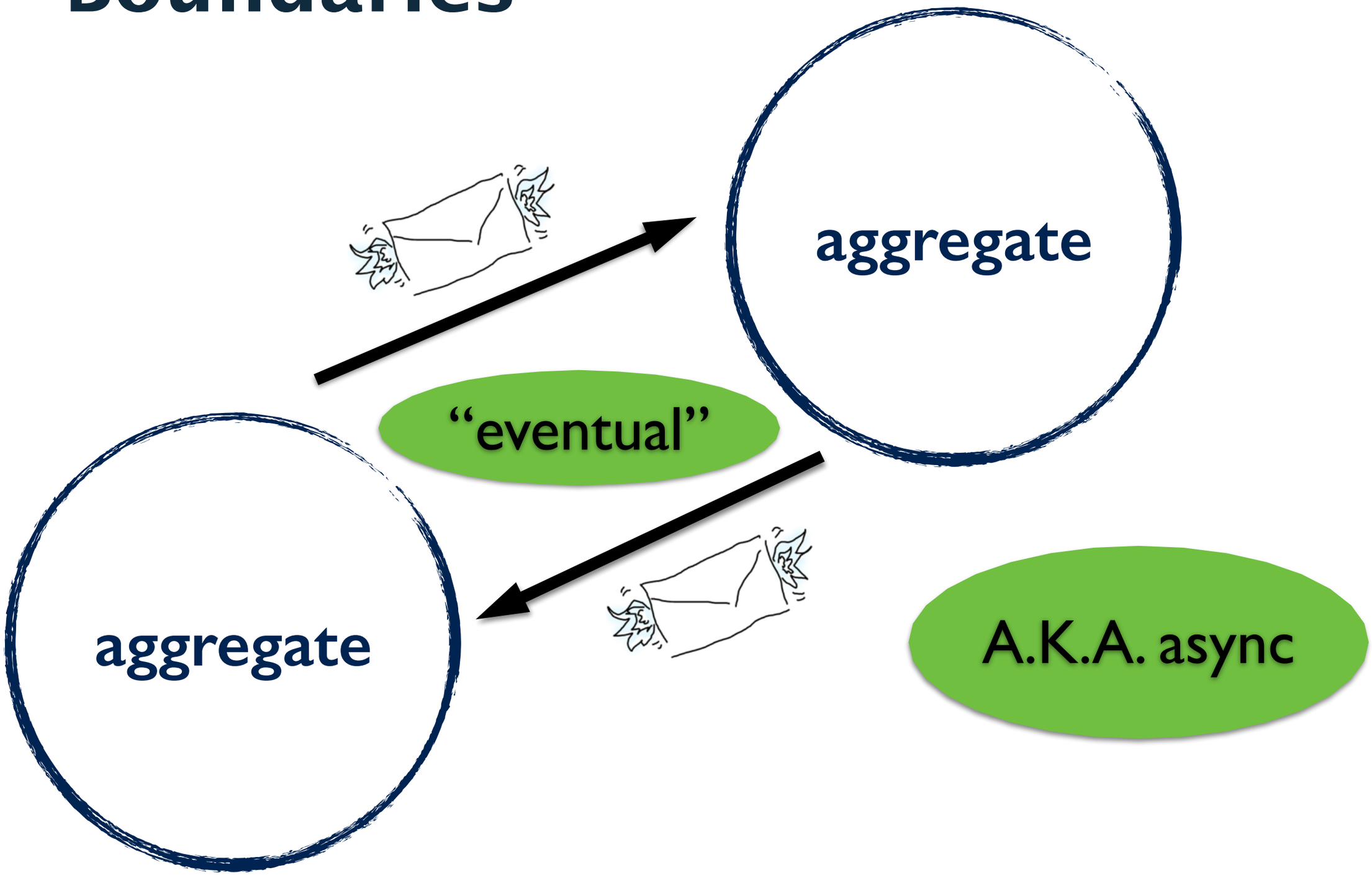
Consistency Boundary

Consistency Boundary equals Async Boundary

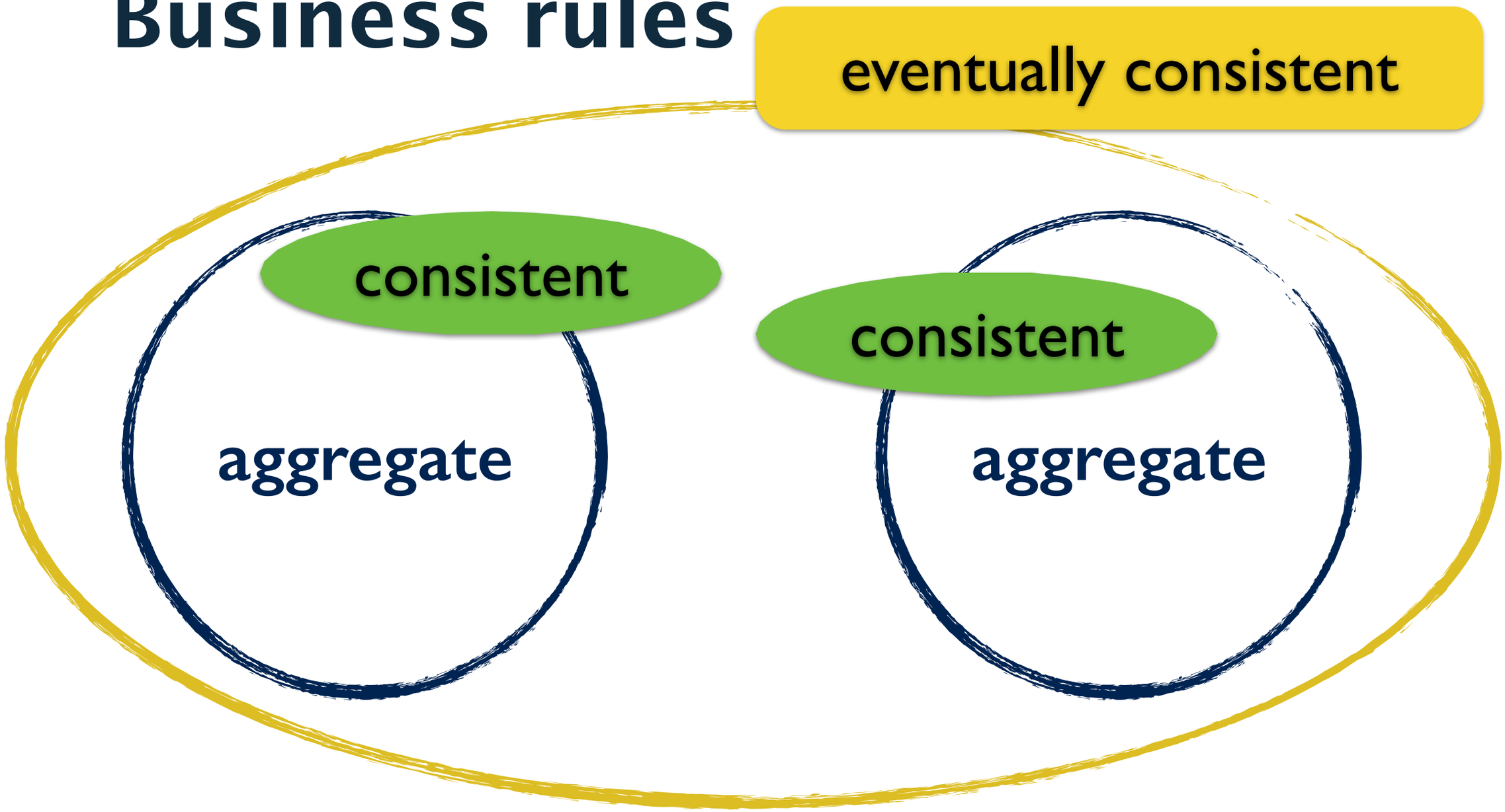
Boundaries



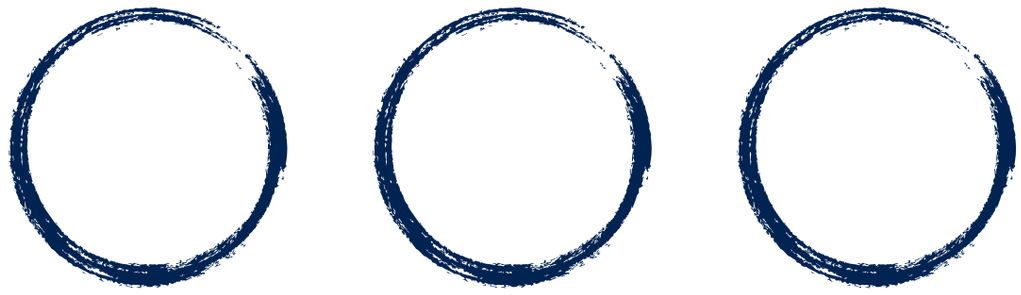
Boundaries



Business rules



Any rule that spans AGGREGATES will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time. [Evans, p. 128]



Let’s open the toolbox

PersistentActor

PersistentActor

Replaces:

Processor & EventsourcedProcessor

in Akka 2.3.4+

Typesafe

super quick domain modelling!

Commands - what others "tell" us; not persisted

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

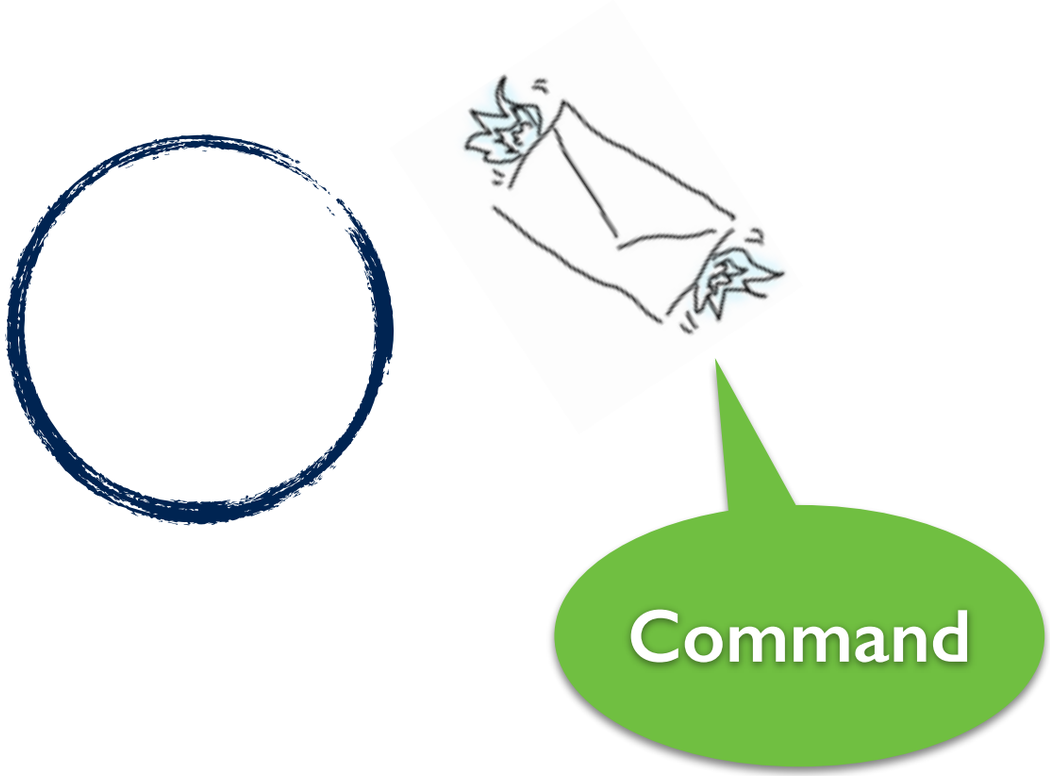
Events - reflect effects, past tense; **persisted**

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

State - reflection of a series of events

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

PersistentActor

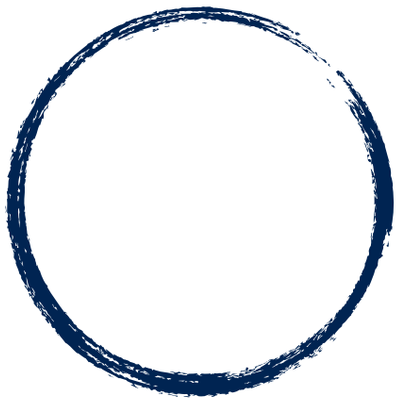


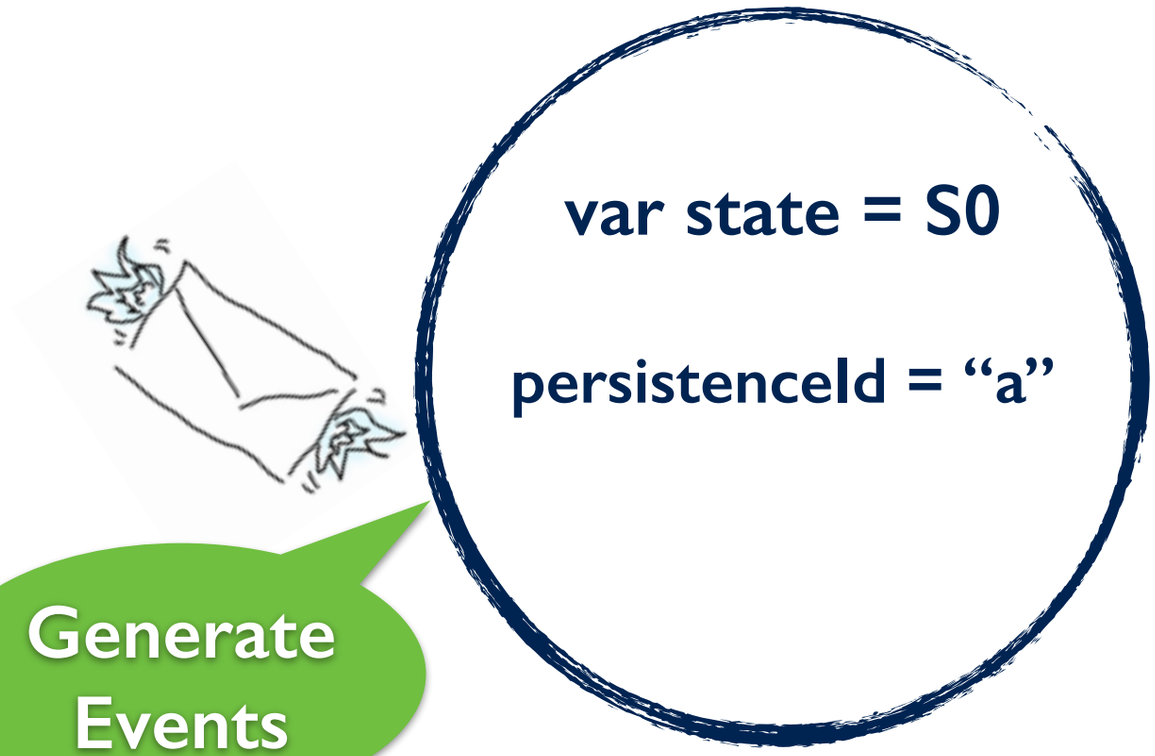
var state = S0

persistenceId = “a”

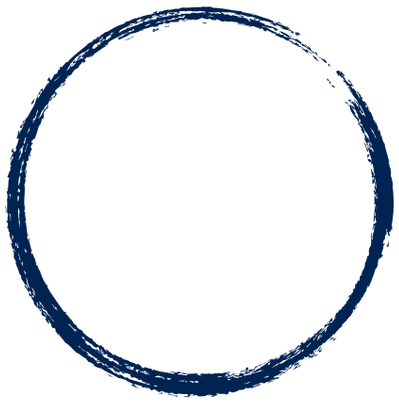
Journal

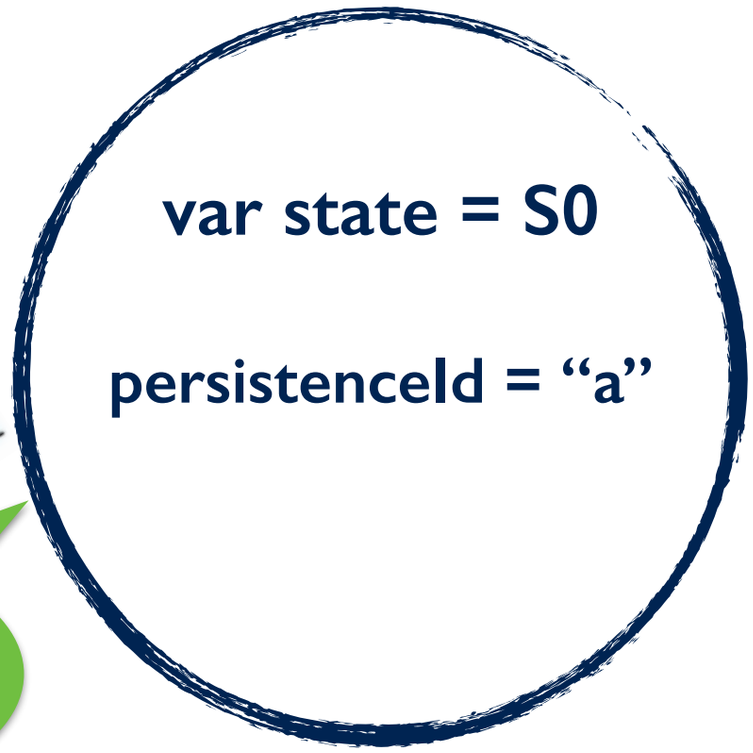
PersistentActor





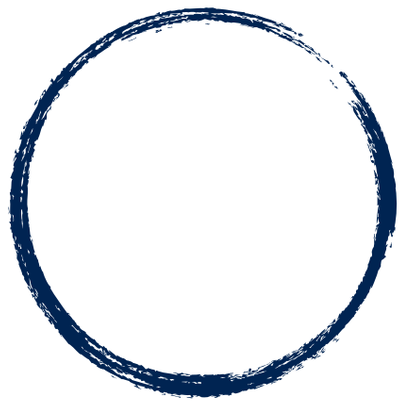
PersistentActor

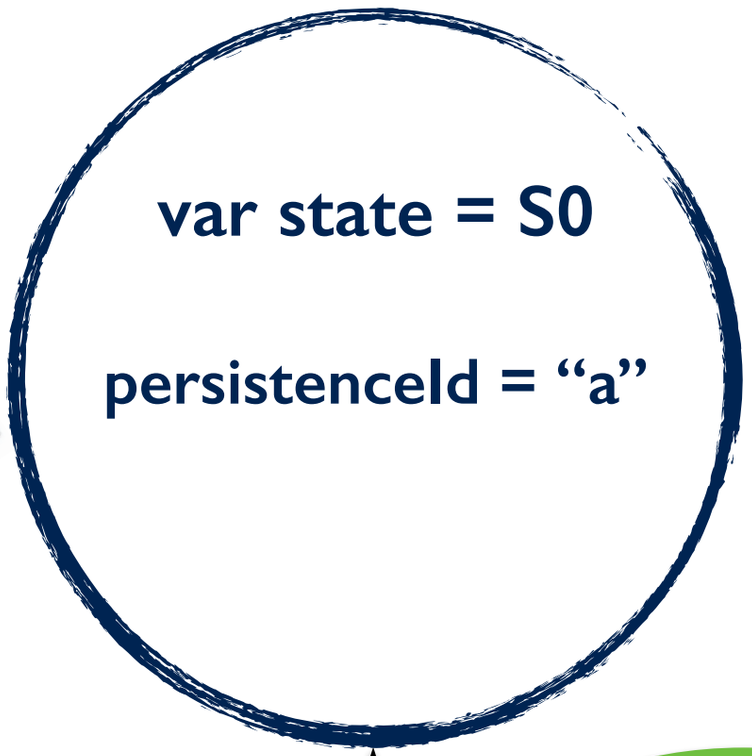




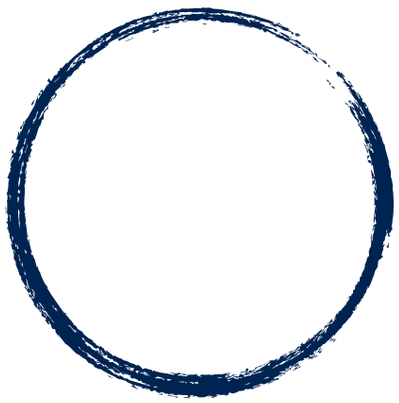
Generate Events

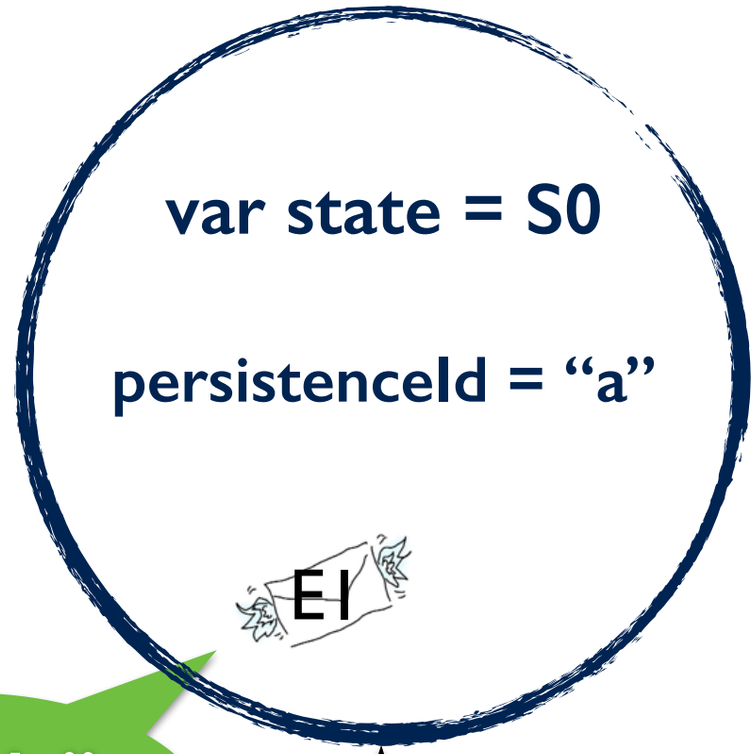
PersistentActor



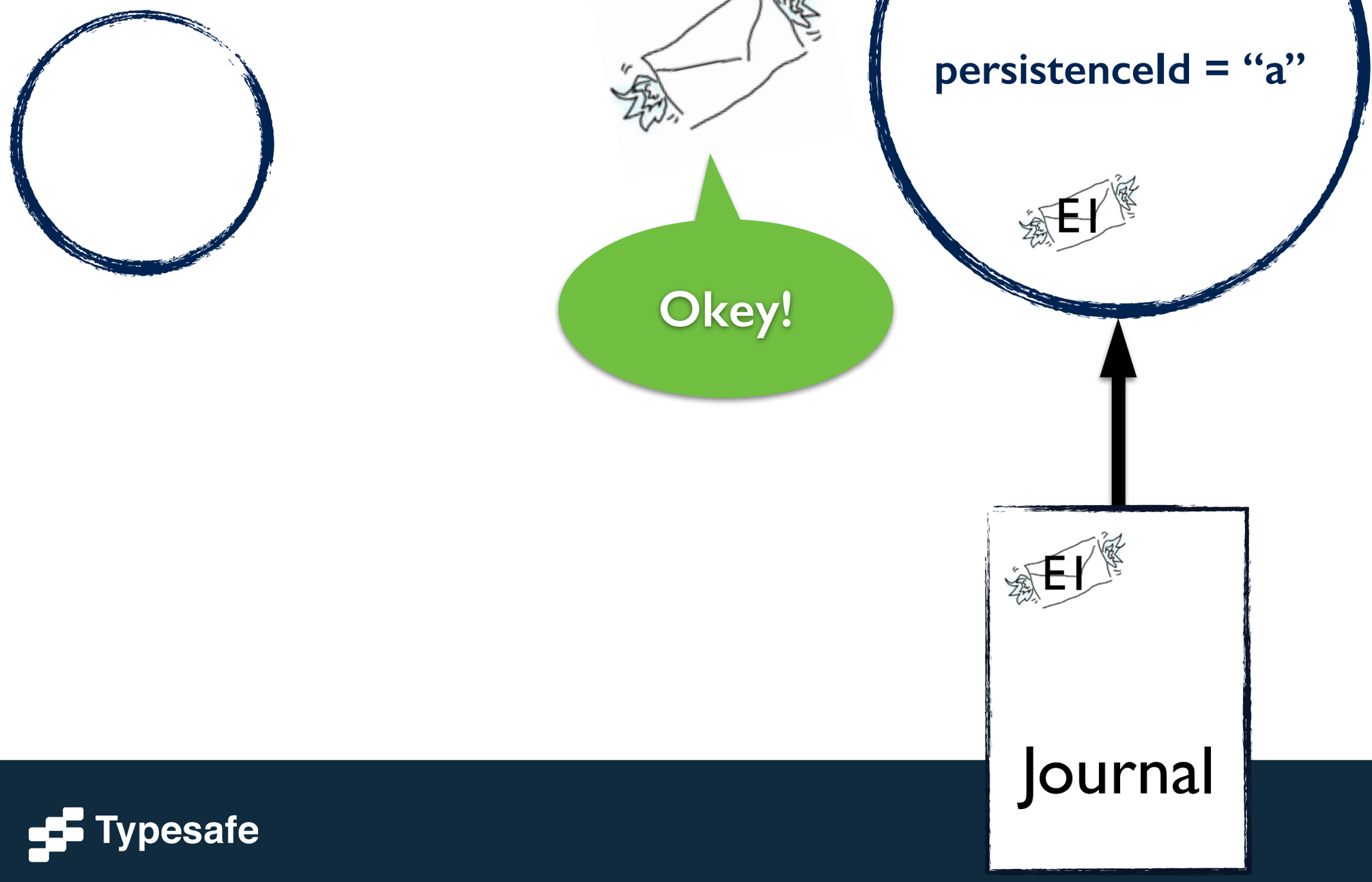


PersistentActor

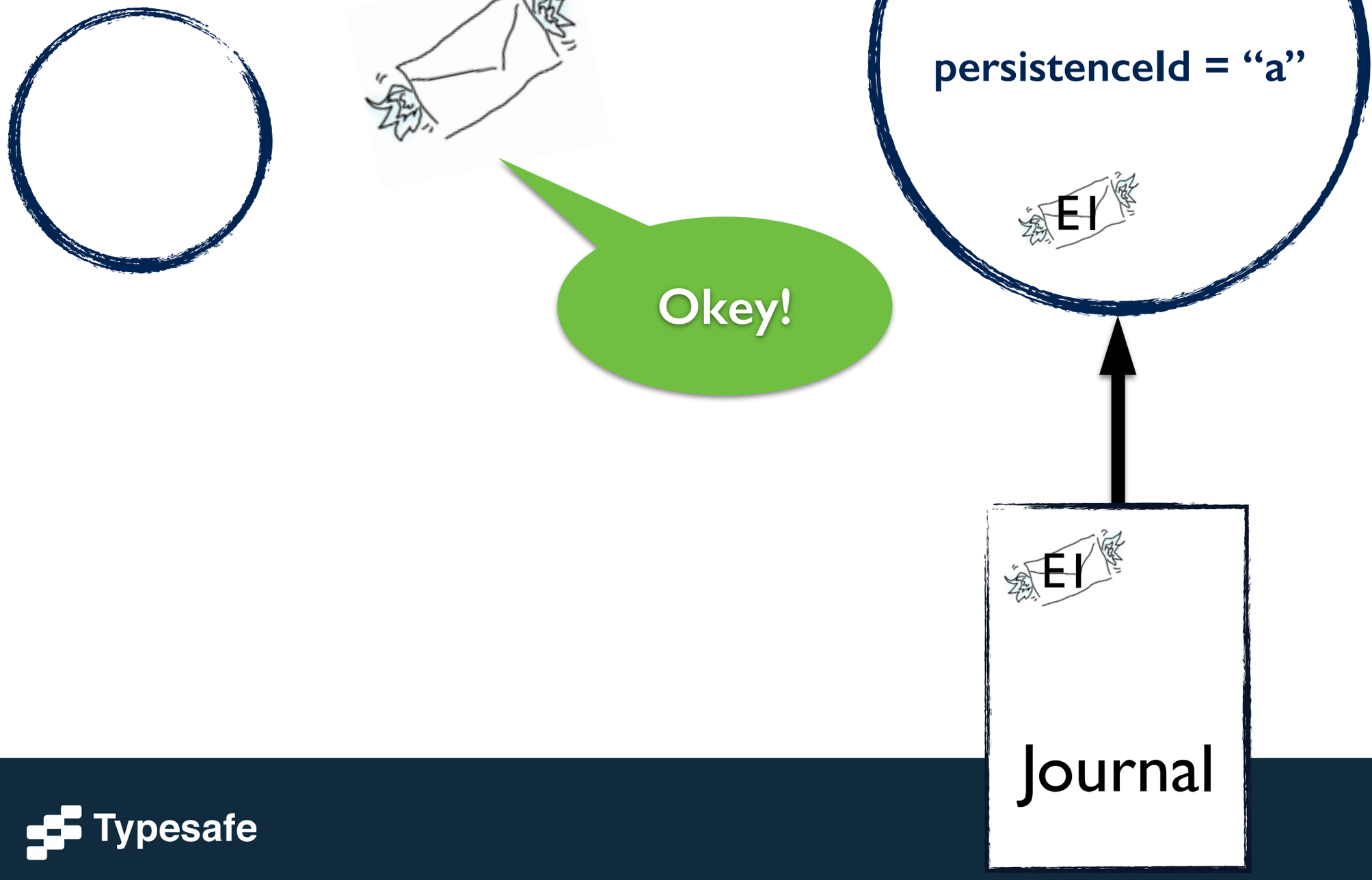




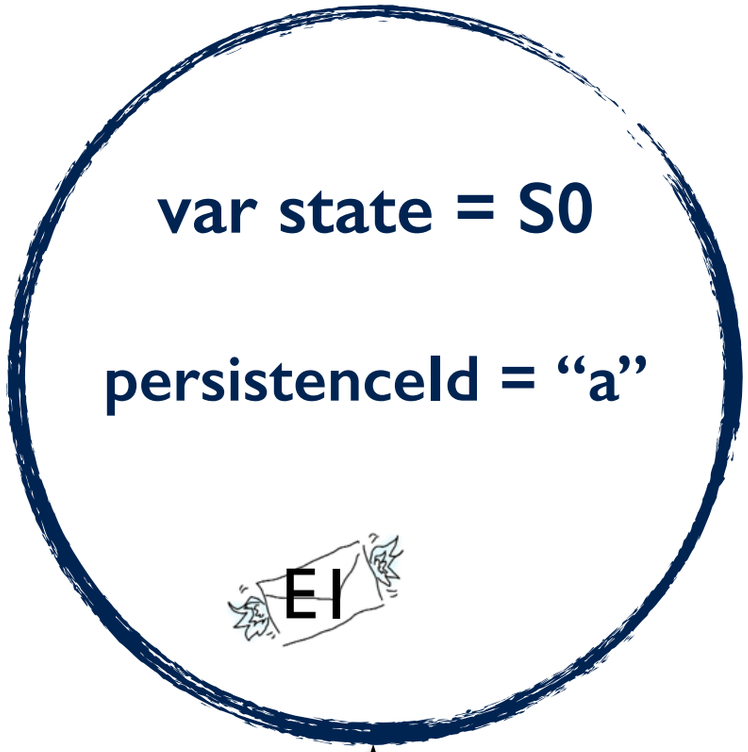
PersistentActor



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

}
```

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

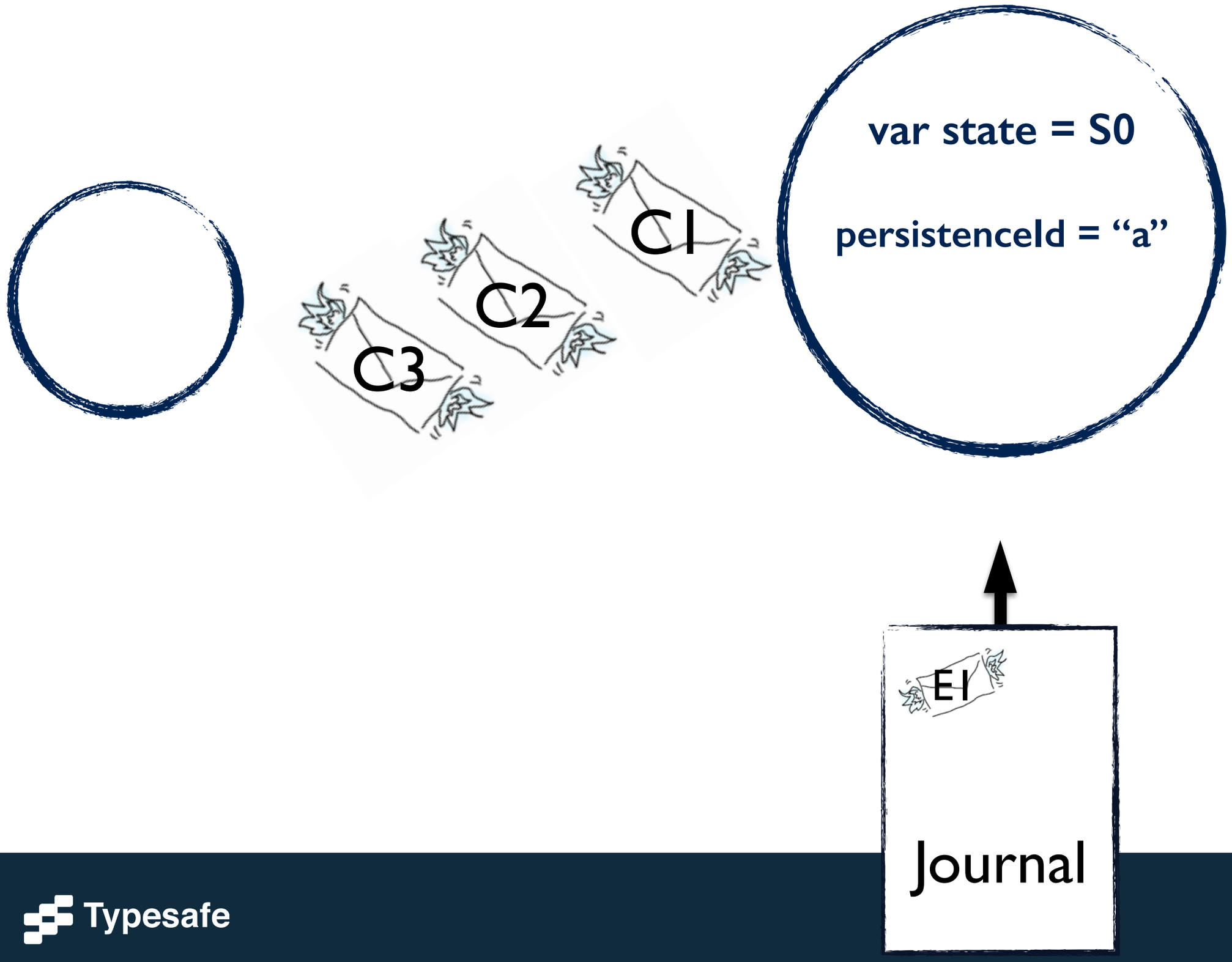
async callback

PersistentActor

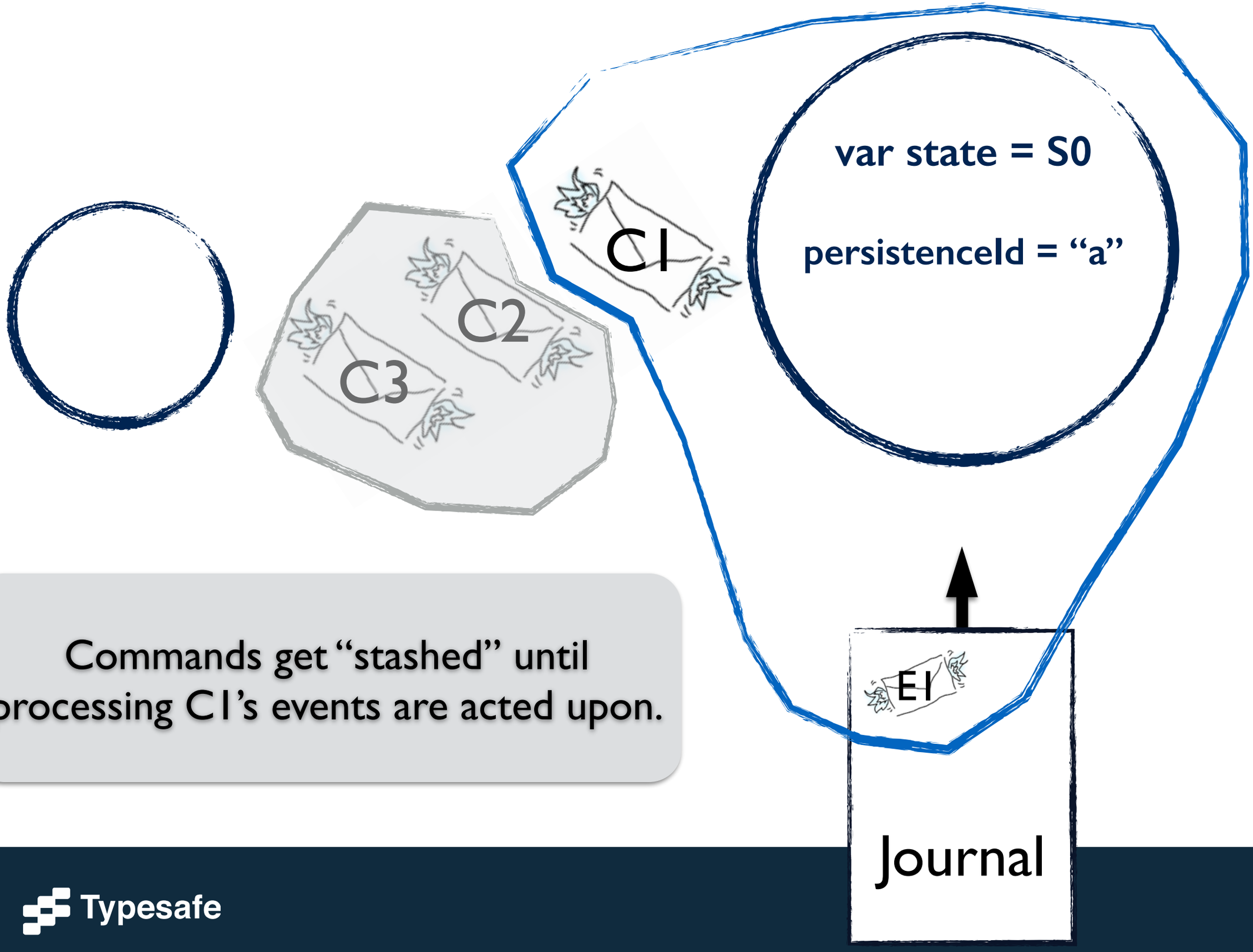
```
def receiveCommand = {
  case GiveMe(coins) if coins <= state.coins =>
    persist(BalanceChangedBy(-coins)) { changed =>
      state = updateState(changed)
      sender() ! TakeMy(coins)
    }
}
```

Safe to access sender here

persist(){} – Ordering guarantees

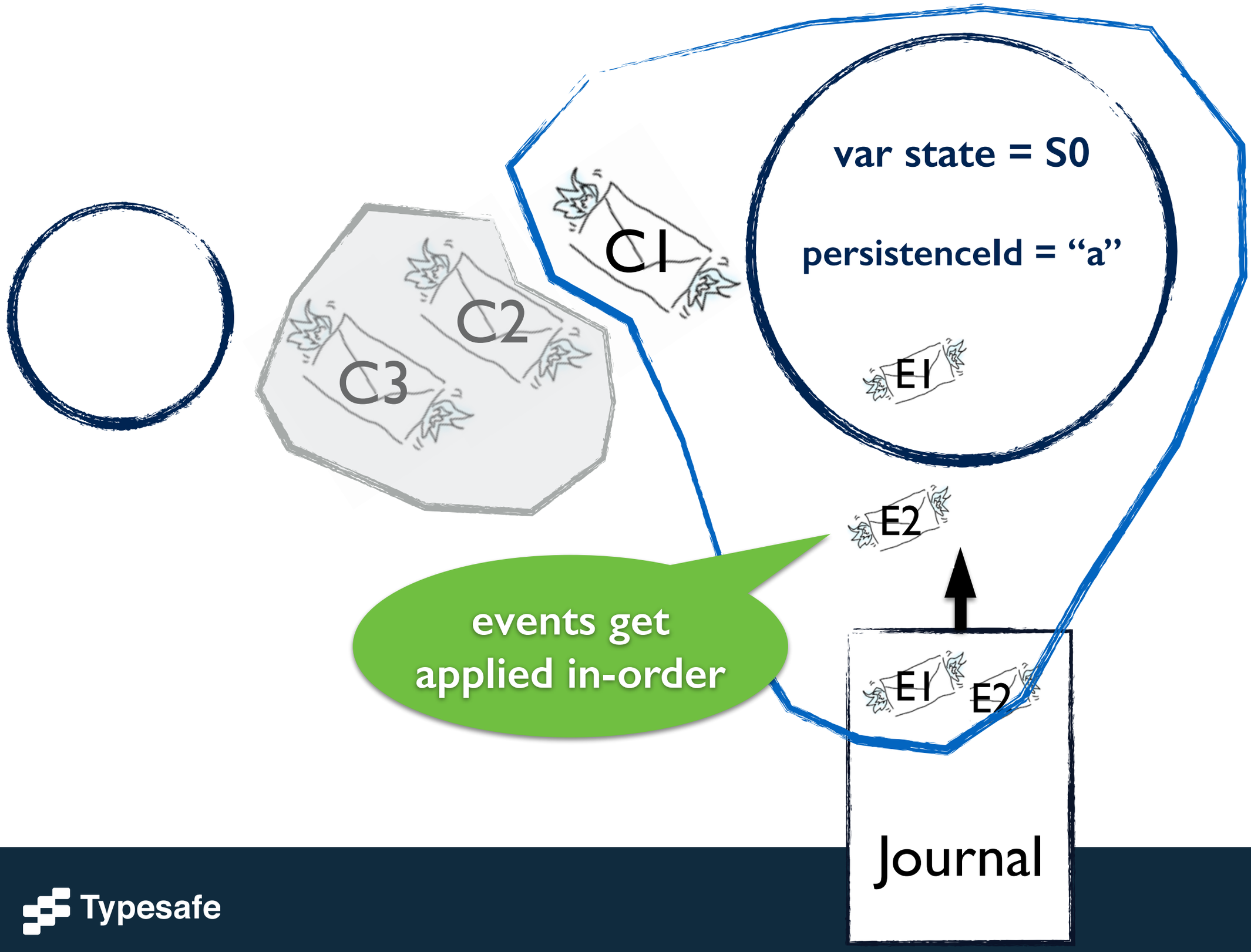


persist(){} – Ordering guarantees

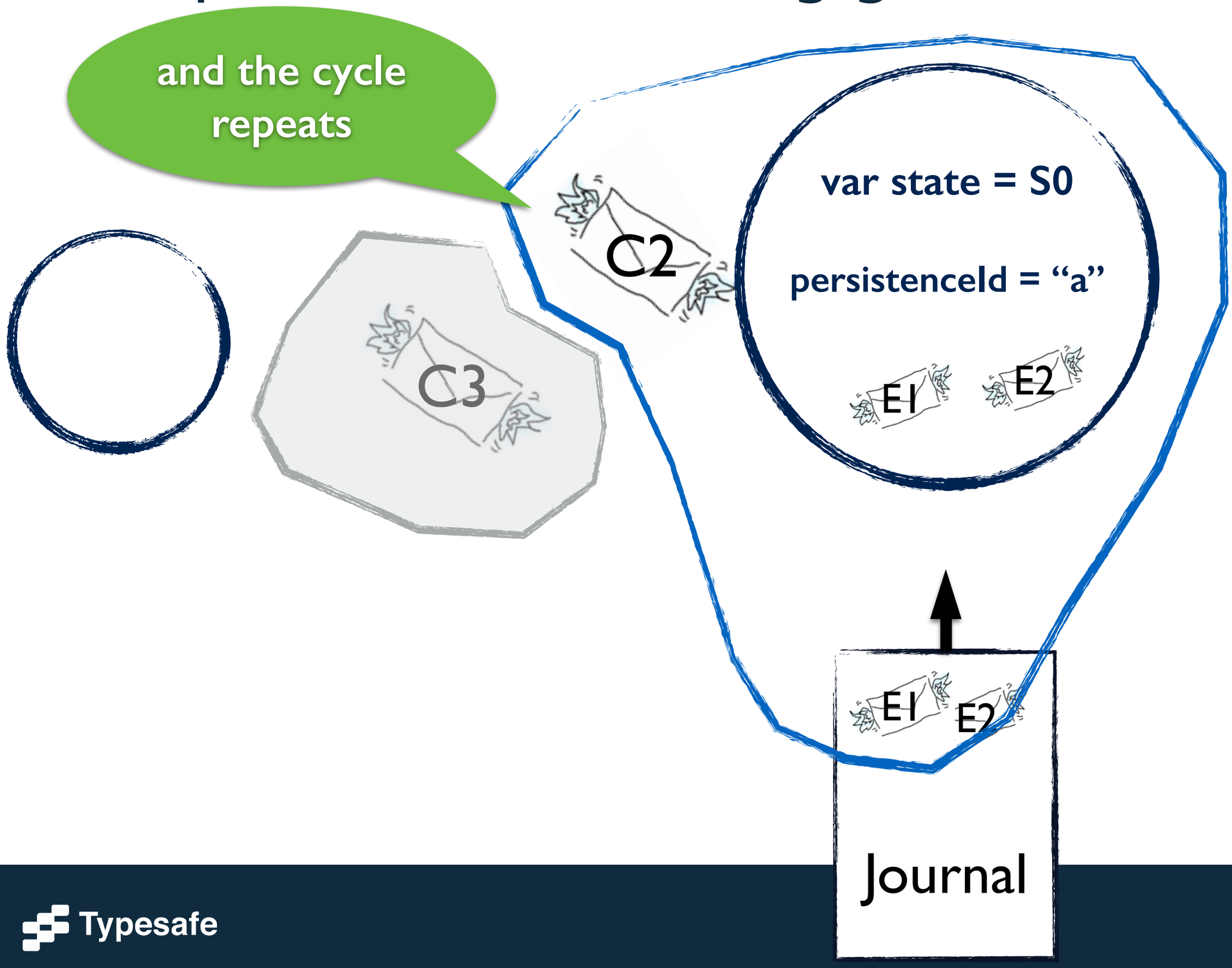


Commands get “stashed” until processing C1’s events are acted upon.

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

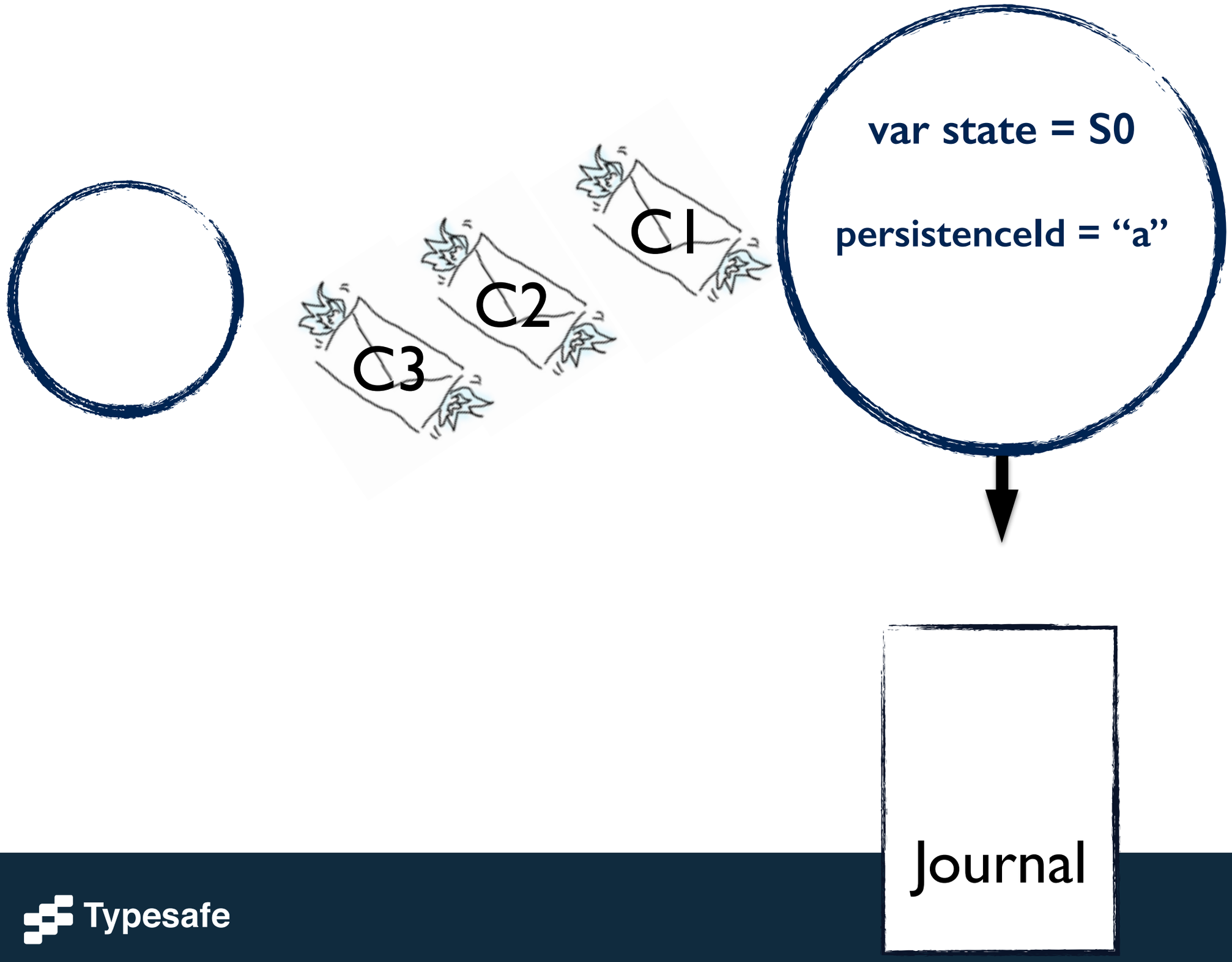
PersistentActor: persistAsync(){}

```
def receiveCommand = {


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

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

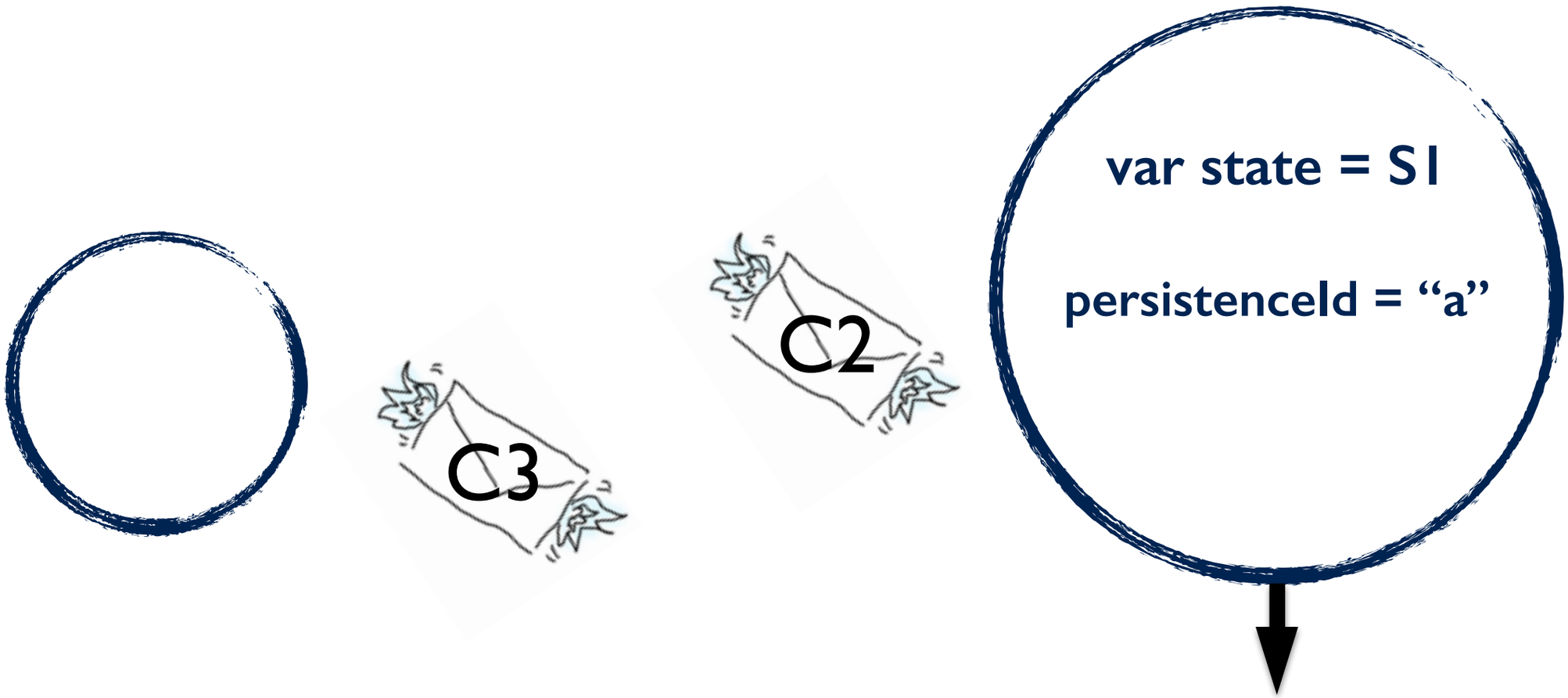
NOT persisted

execute once all
persistAsync handlers done

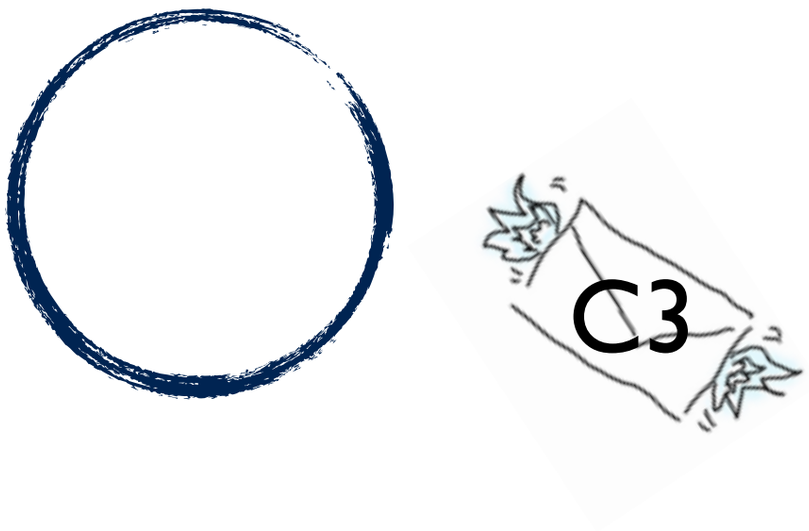
persistAsync(){} – Ordering guarantees

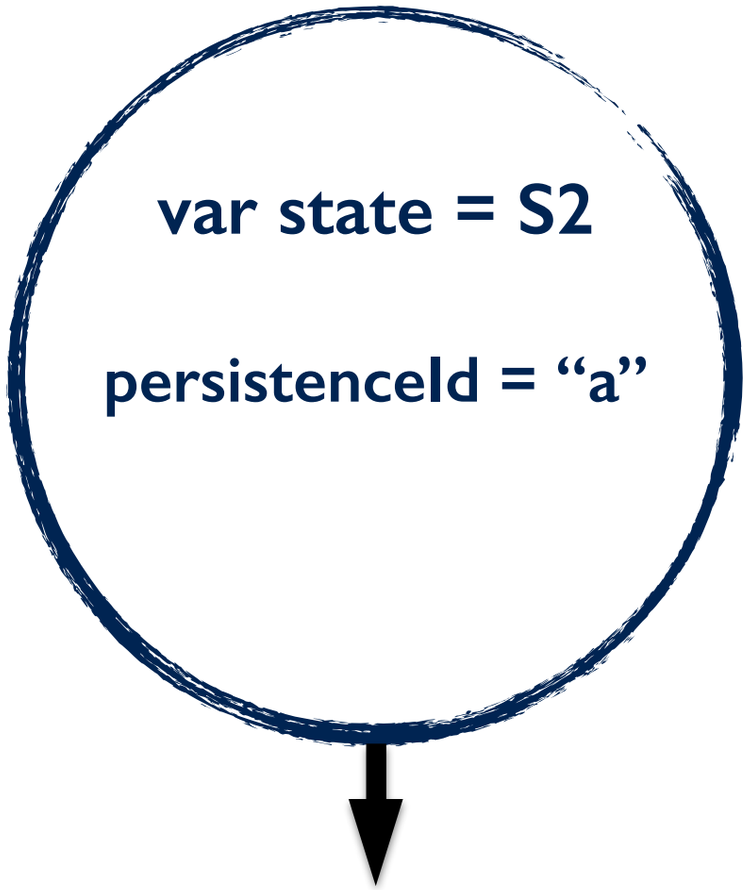


persistAsync(){} – Ordering guarantees

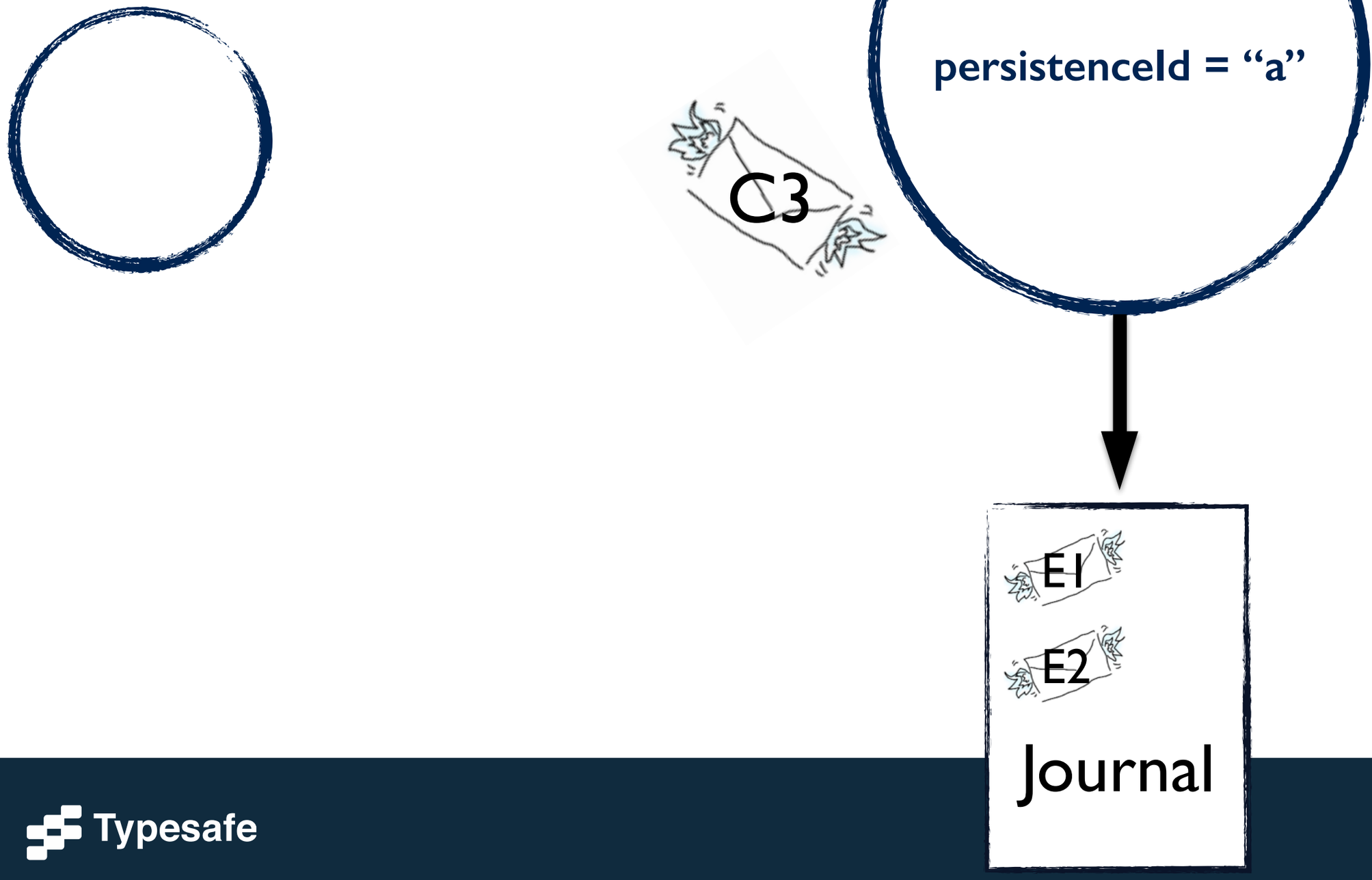


persistAsync(){} – Ordering guarantees

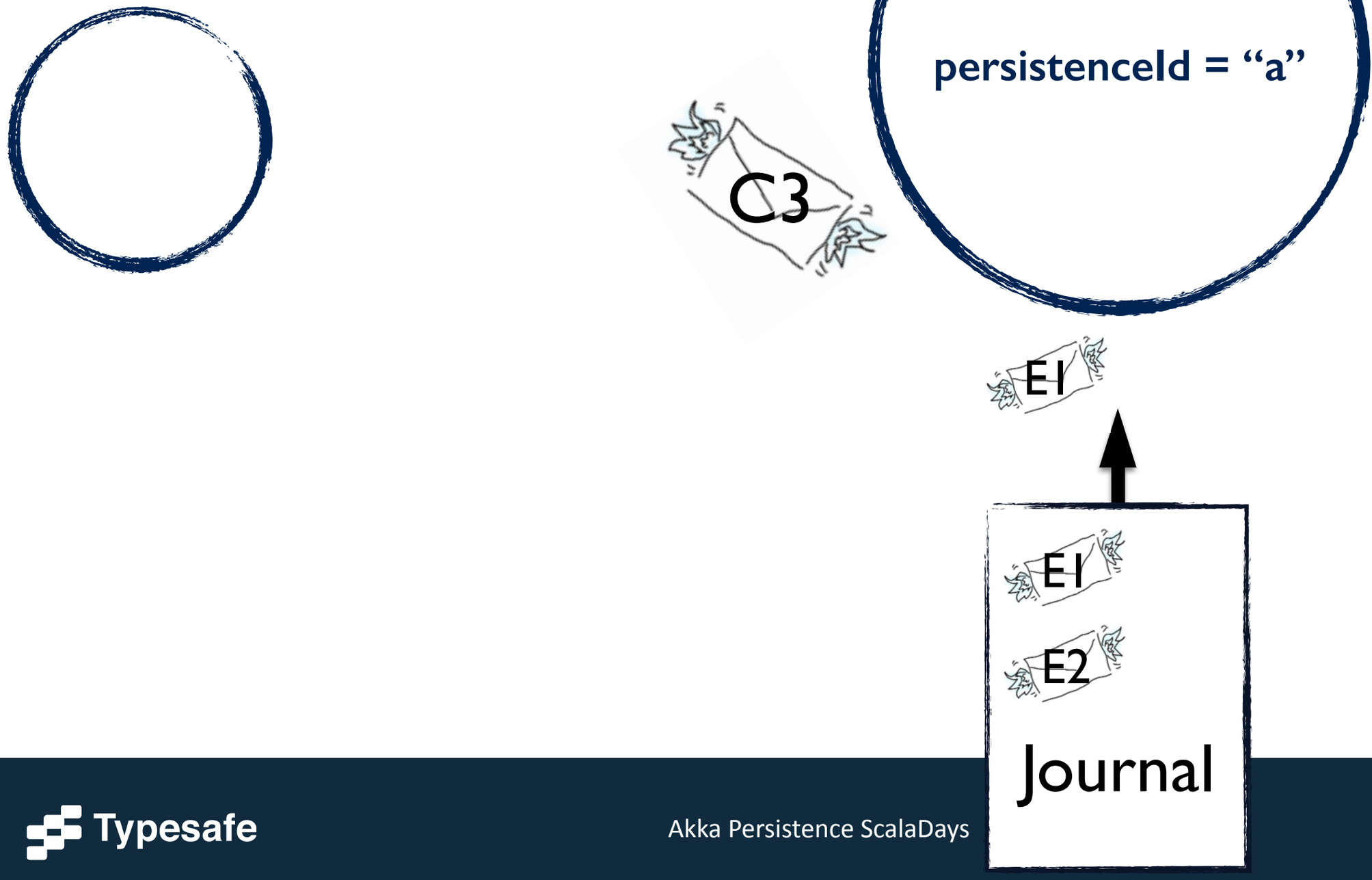




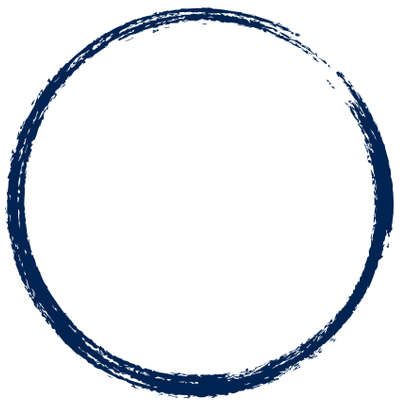
persistAsync(){} – Ordering guarantees

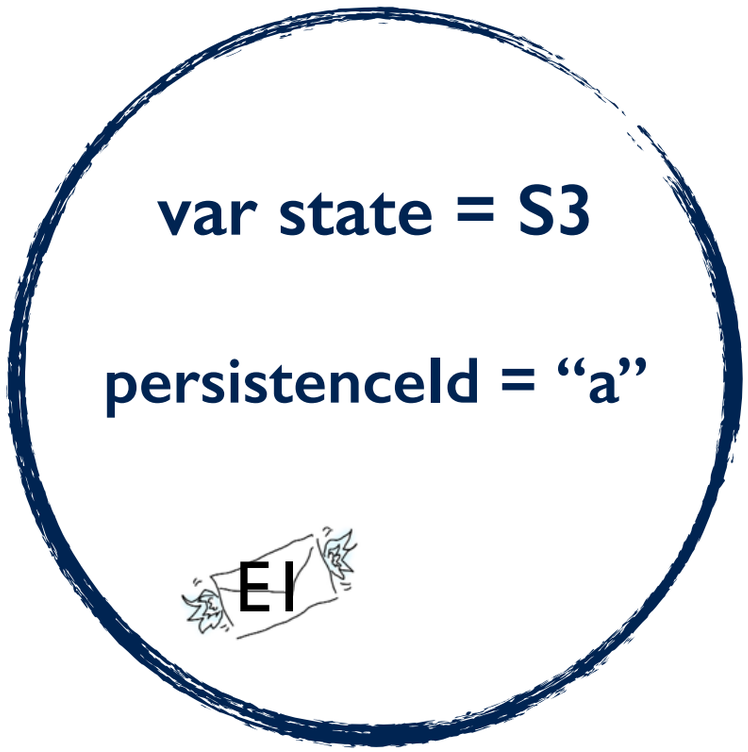


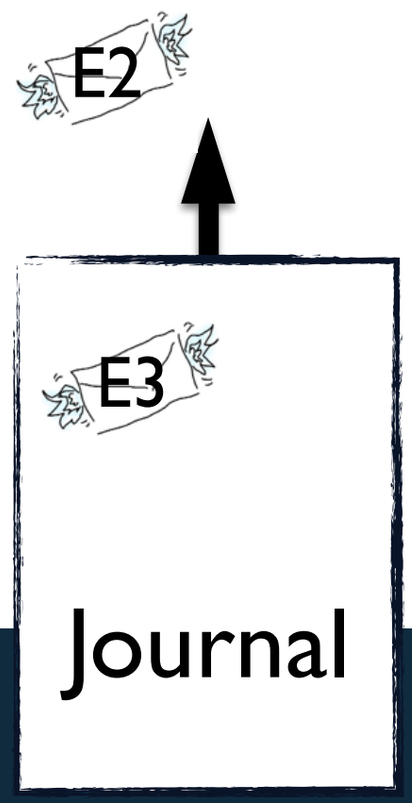
persistAsync(){} – Ordering guarantees



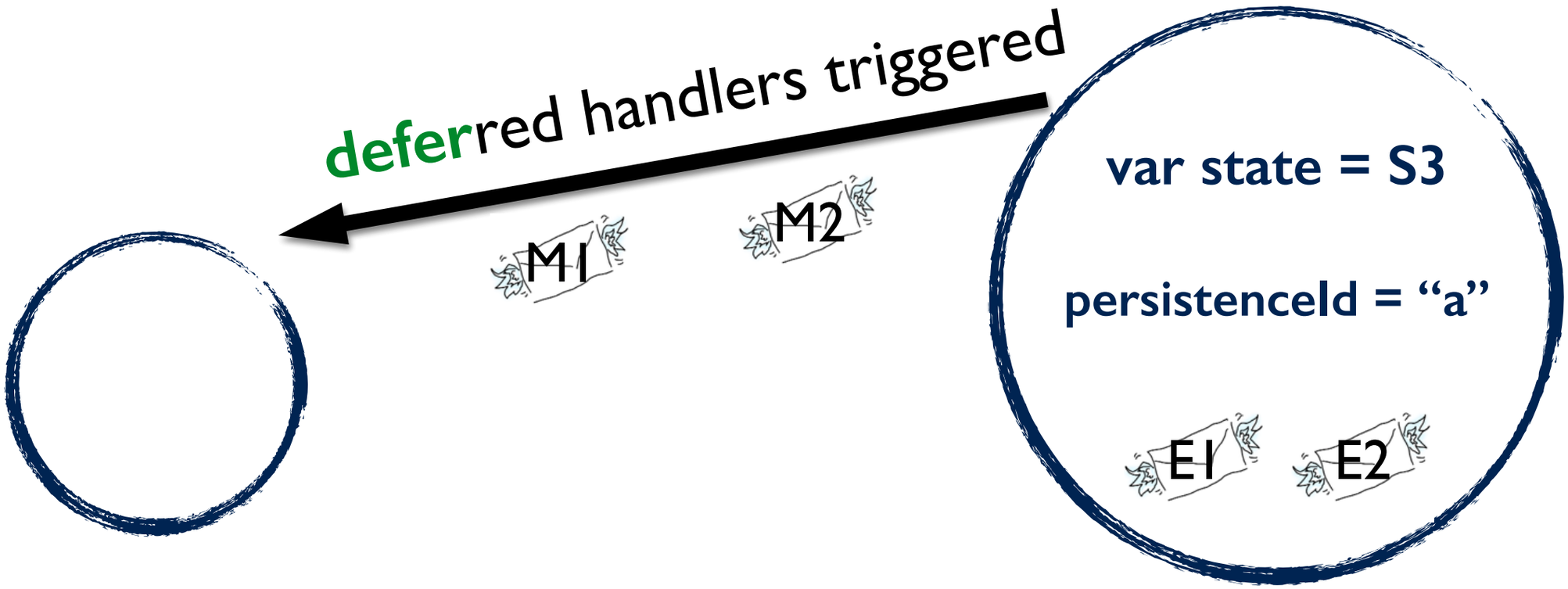
persistAsync(){} – Ordering guarantees







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*

Snapshots

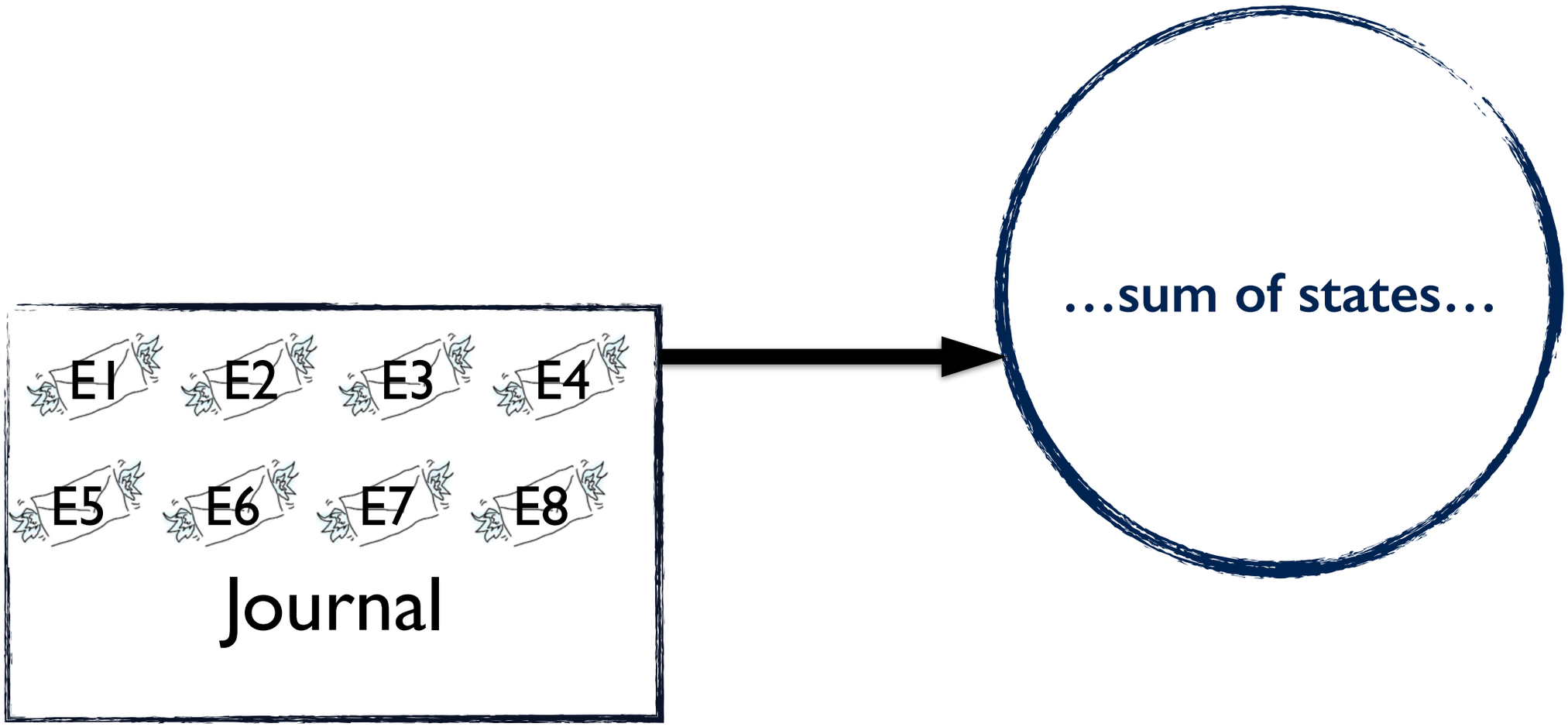
Snapshots
(in SnapshotStore)

Eventsourced, snapshots

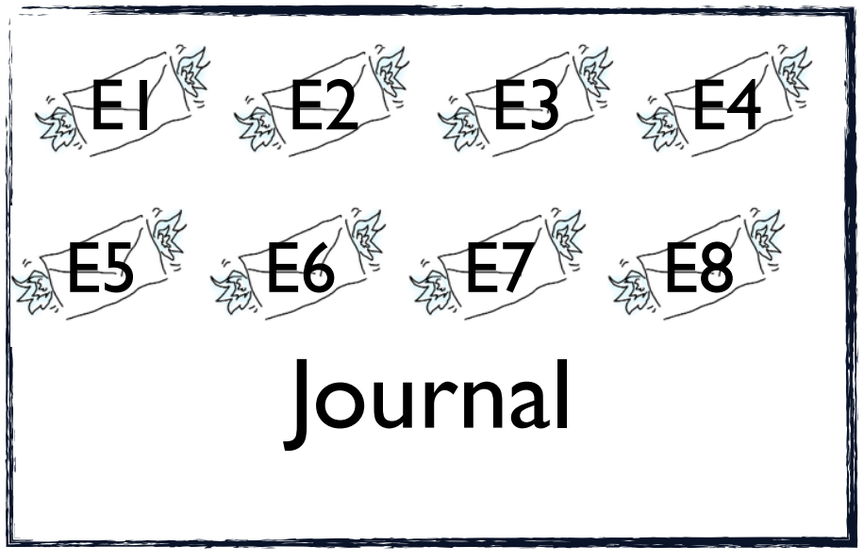
```
/** MUST NOT SIDE-EFFECT! */
def receiveRecover = {
  case SnapshotOffer(meta, snapshot: State) =>
    this.state = state

  case replayedEvent: Event  =>
    updateState(replayedEvent)
}
```

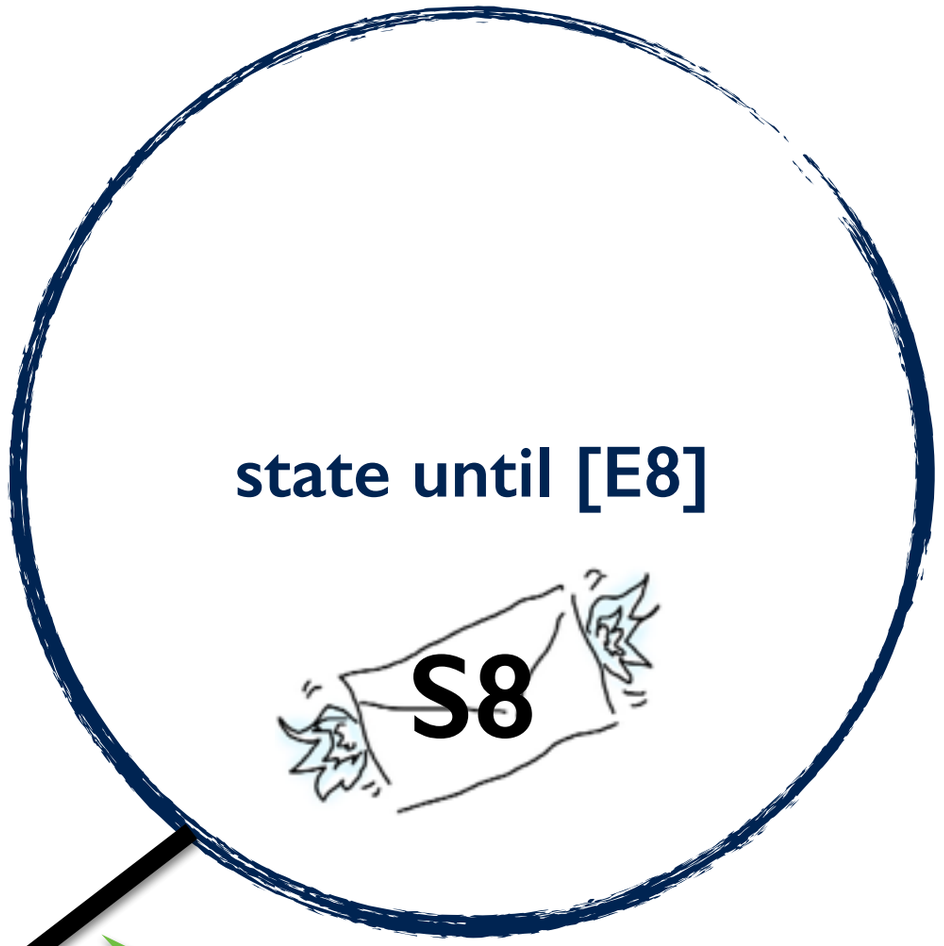

```
def receiveCommand = {
  case command: Command =>
    saveSnapshot(state) // async!
}
```

Snapshots



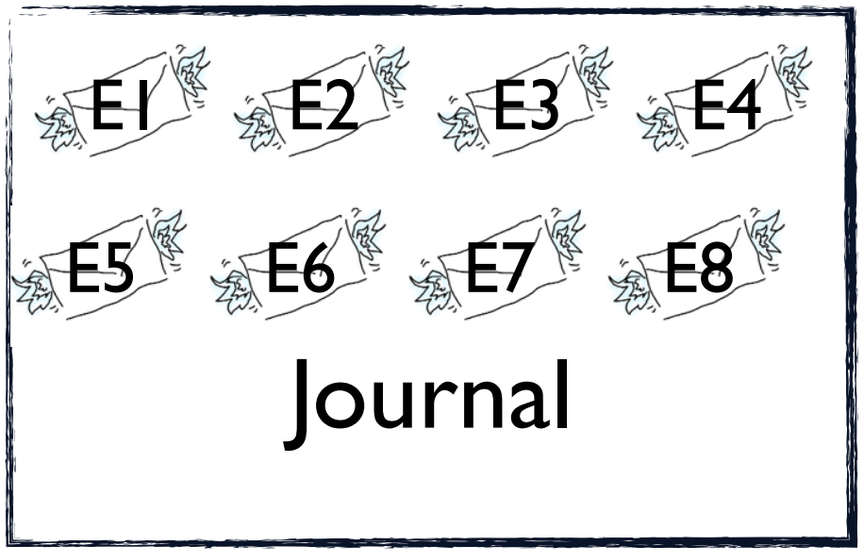
Snapshots

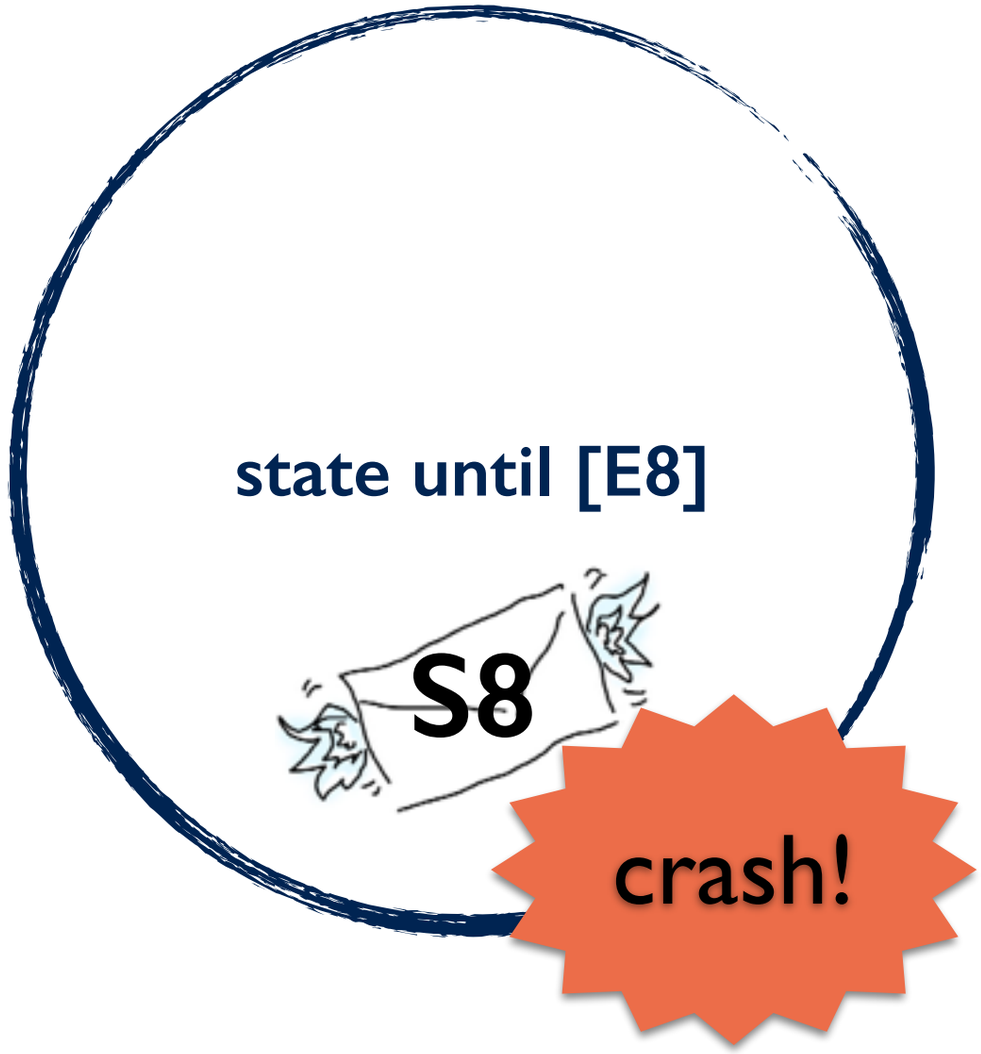




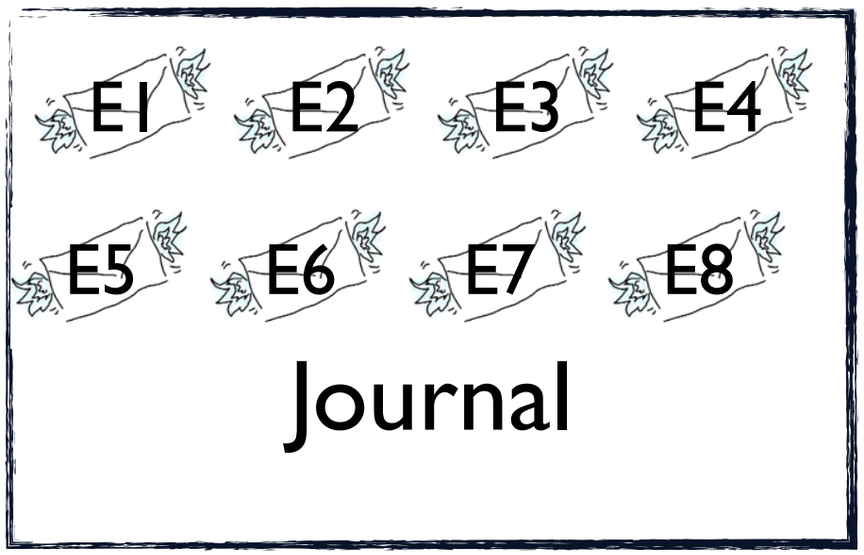
snapshot!

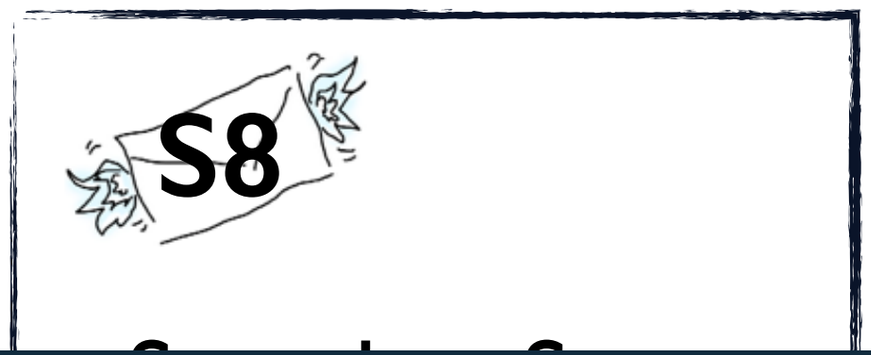
Snapshots



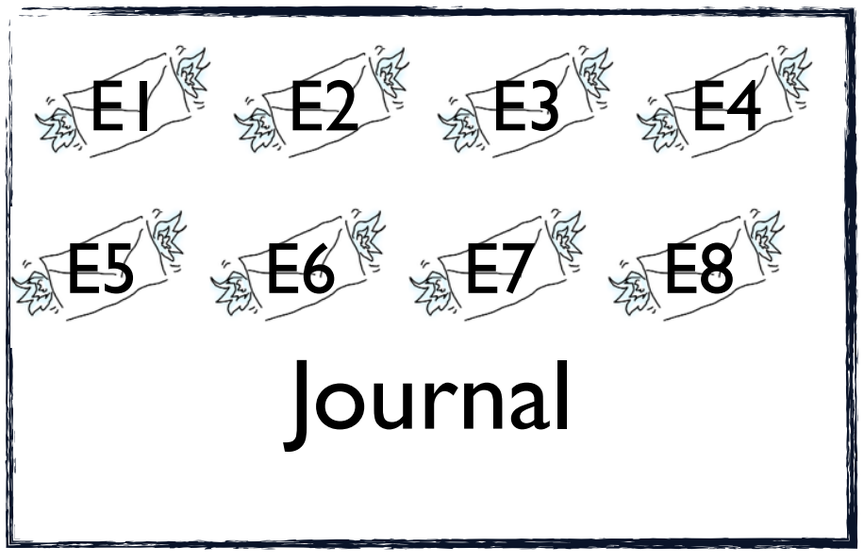


Snapshots





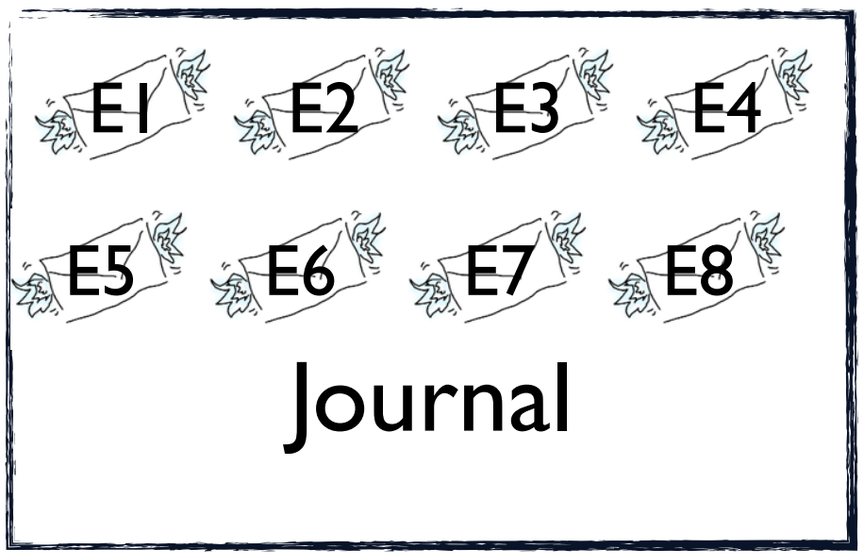
Snapshots

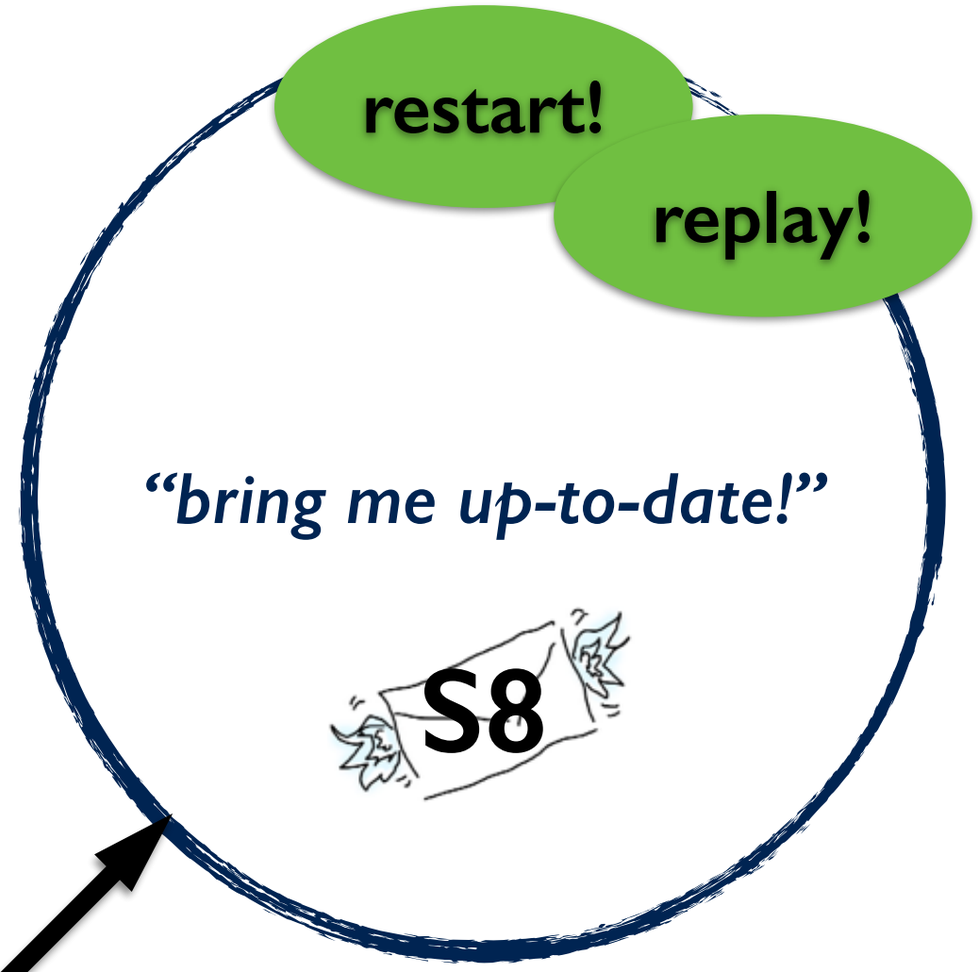

E1
E2
E3
E4
E5
E6
E7
E8
Journal


restart!
replay!
"bring me up-to-date!"


S8

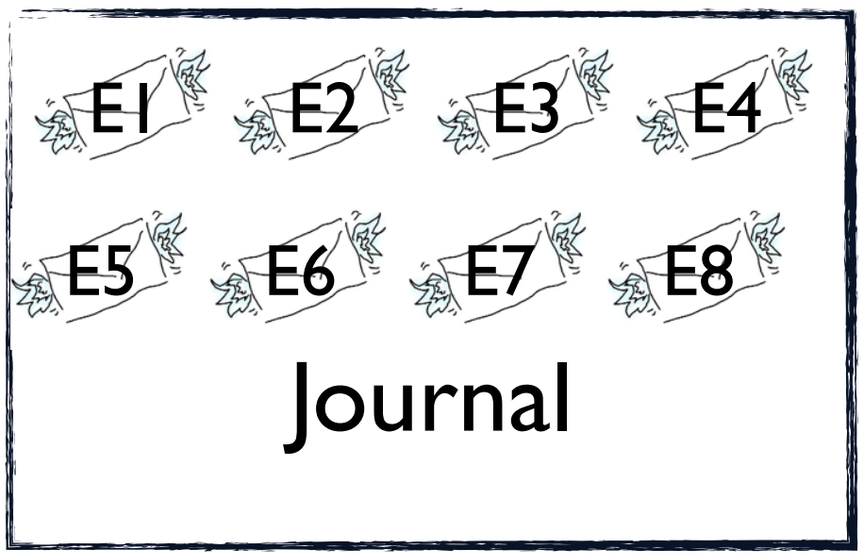
Snapshots

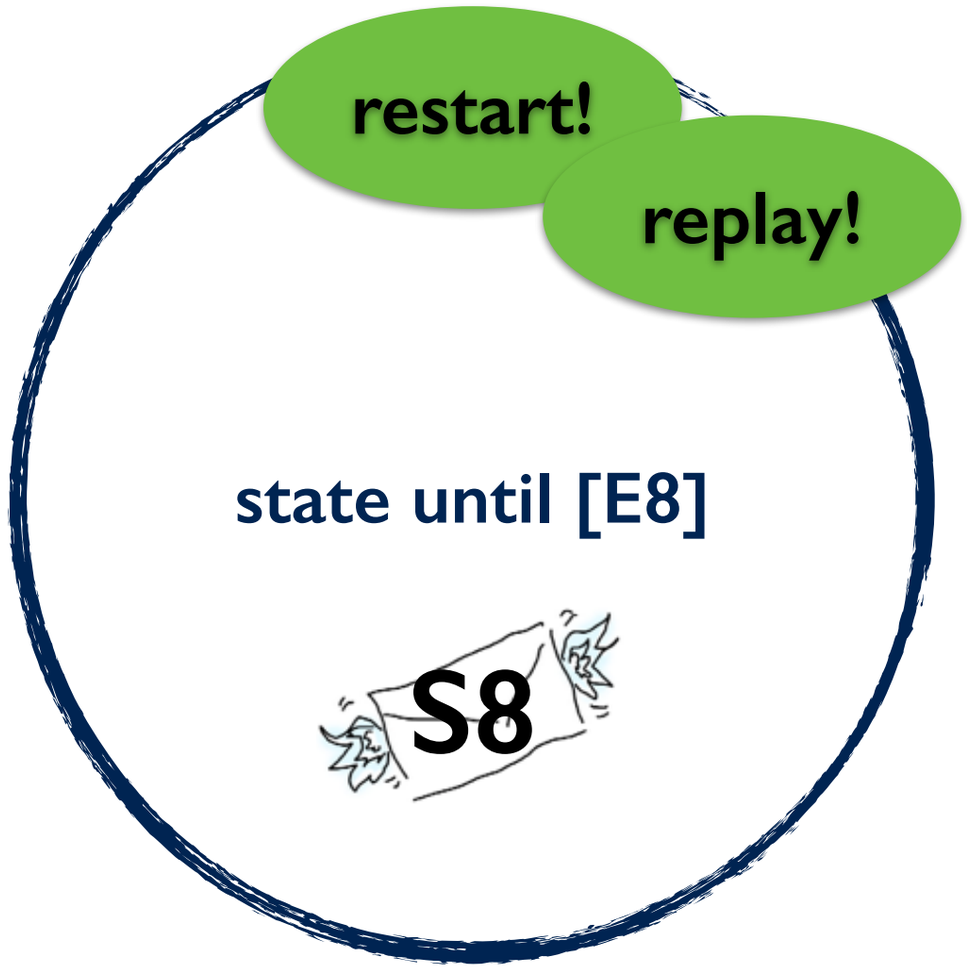

E1
E2
E3
E4
E5
E6
E7
E8
Journal


restart!
replay!
"bring me up-to-date!"
S8


S8

Snapshots

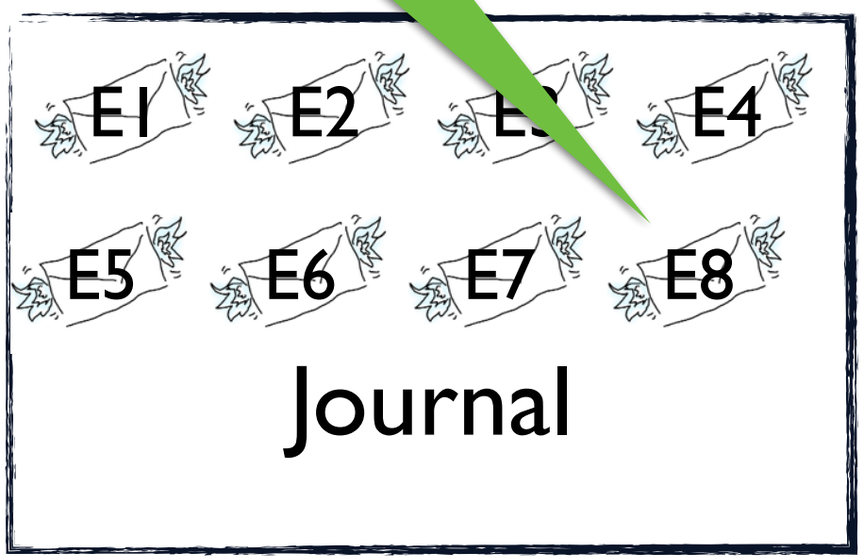

E1
E2
E3
E4
E5
E6
E7
E8
Journal

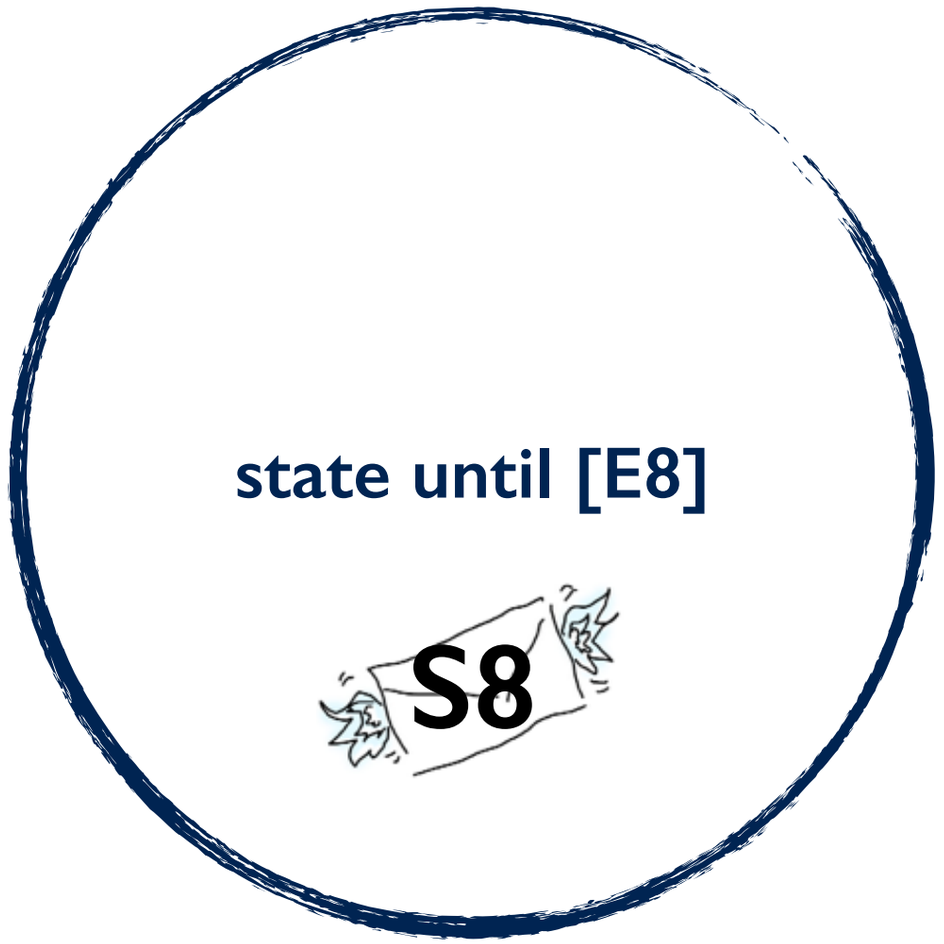

restart!
replay!
state until [E8]
S8


S8

Snapshots

We *could* delete these!





Snapshots, save

```
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap"                                    => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata)             => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}
```

Snapshots, success

```
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap"                                  => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata)           => // ...
    case SaveSnapshotFailure(metadata, reason)   => // ...
  }
}
```

```
final case class SnapshotMetadata(
  processorId: String, sequenceNr: Long,
  timestamp: Long)
```

Snapshots, success

```
trait MySummer extends Processor {
  var nums: List[Int]
  var total: Int

  def receive = {
    case "snap"                                    => saveSnapshot(total)
    case SaveSnapshotSuccess(metadata)             => // ...
    case SaveSnapshotFailure(metadata, reason) => // ...
  }
}
```

Snapshot Recovery

```
class Counter extends Processor {
  var total = 0

  def receive = {
    case SnapshotOffer(metadata, snap: Int) =>
      total = snap

    case Persistent(payload, sequenceNr)          => // ...
  }
}
```

Snapshots

Upsides

- **Simple!**
- **Faster recovery (!)**
- Allows to delete “older” events
- “known state at point in time”

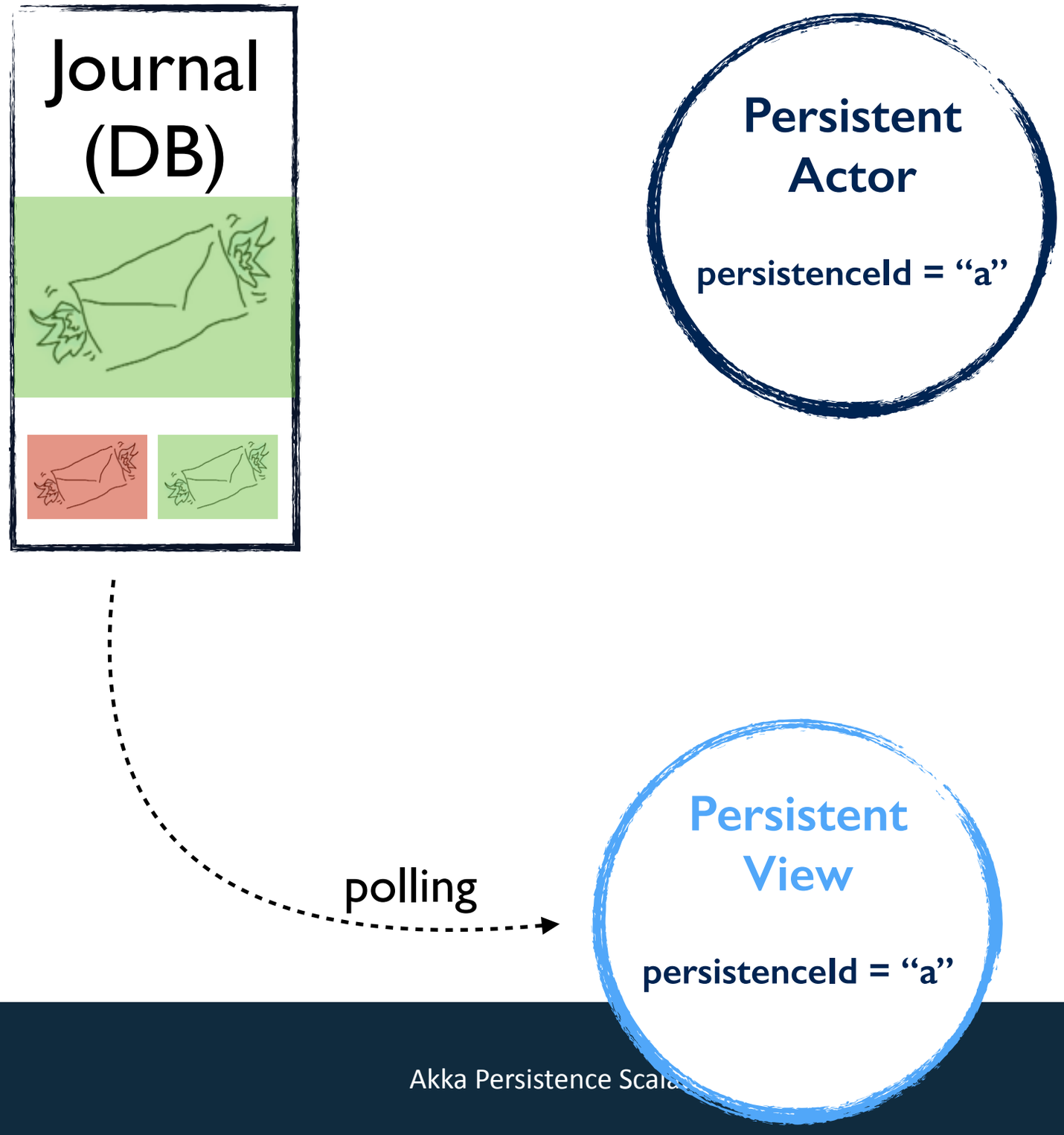
Snapshots

Downsides

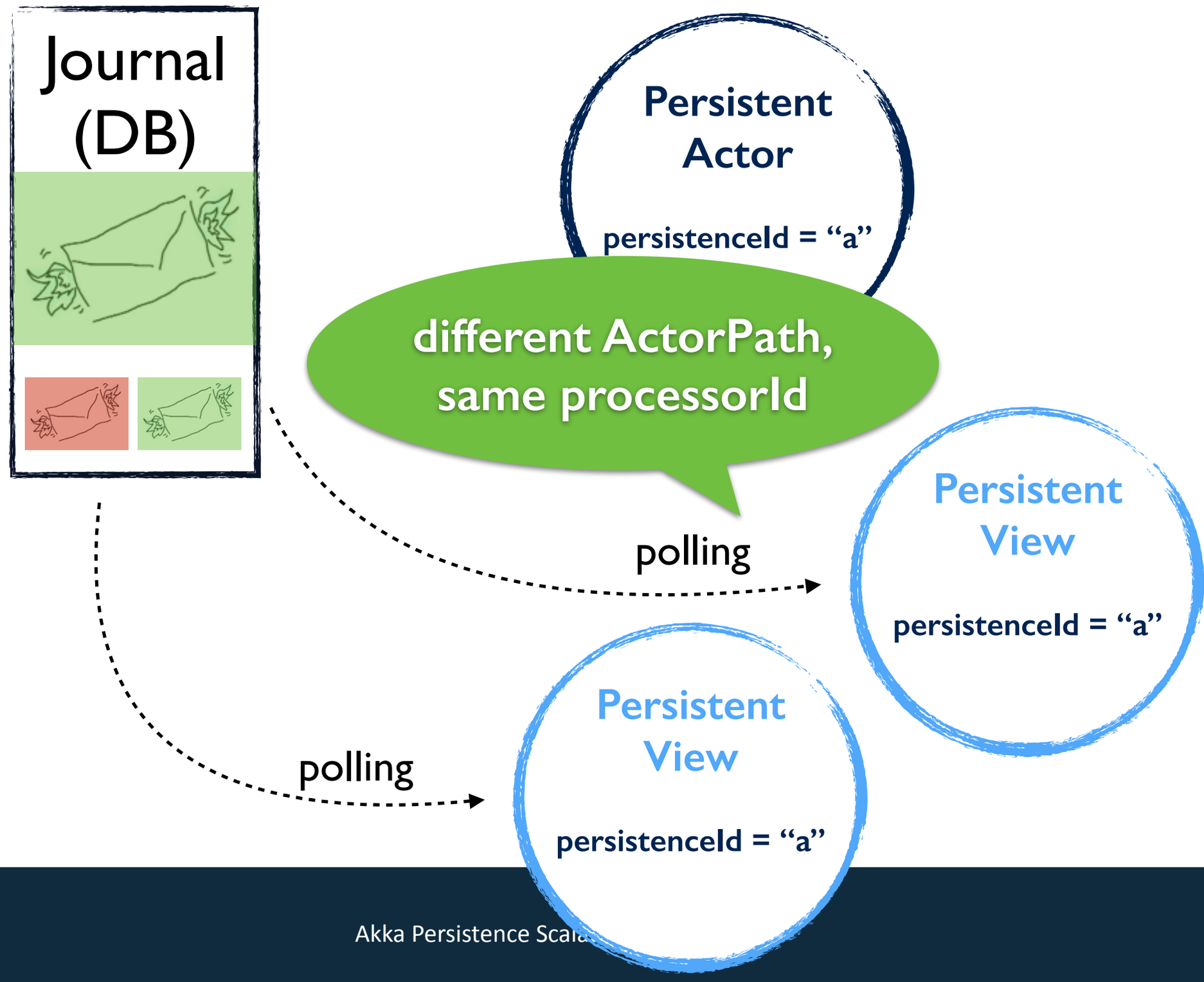
- More logic to write
- Maybe not needed if events replay "fast enough"
- Possibly "yet another database" to pick
  - snapshots are different than events, may be big!

Views

Views


Journal
(DB)
Persistent
Actor
persistenceId = “a”
polling
Persistent
View
persistenceId = “a”

Views


Journal
(DB)
Persistent
Actor
persistenceId = “a”
different ActorPath,
same processorId
polling
Persistent
View
persistenceId = “a”
polling
Persistent
View
persistenceId = “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.Publisher
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: Publisher[Persistent] = PersistentFlow.
  fromPersistenceId(“p-1").
  toPublisher(materializer)

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

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

View, as ReactiveStream

early preview

pull request
by *krasserm*

```
// 2 producers (merged) and 1 consumer:
val p2: Publisher[Persistent] = PersistentFlow.
  fromPersistenceId(“p-2").
  toPublisher(materializer)

val p3: Publisher[Persistent] = PersistentFlow.
  fromPersistenceId(“p-3").
  toPublisher(materializer)

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

Persistence + Cluster

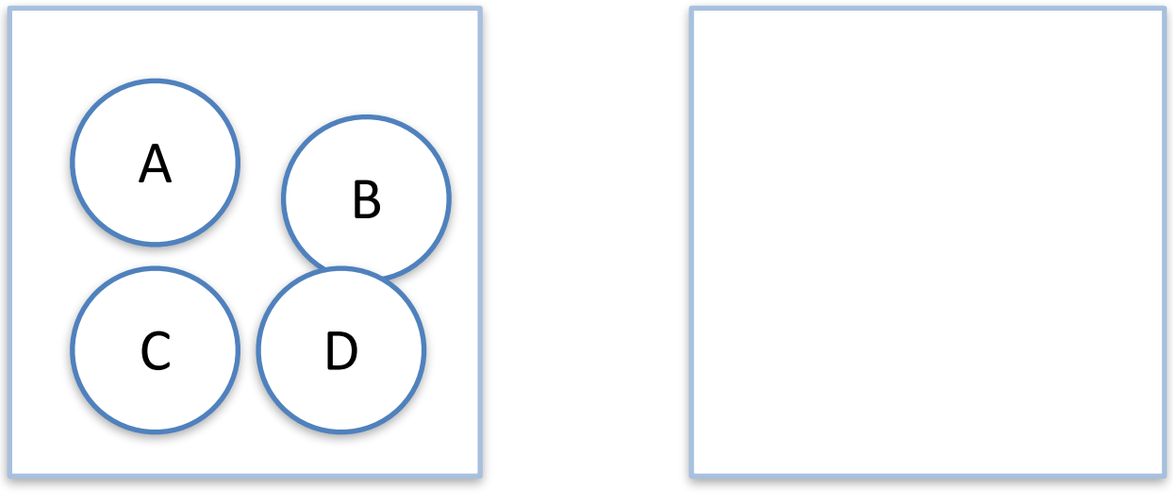
Usage in a Cluster

- **distributed journal** (http://akka.io/community/)
  - Cassandra
  - DynamoDB
  - HBase
  - Kafka
  - MongoDB
  - shared LevelDB journal for testing

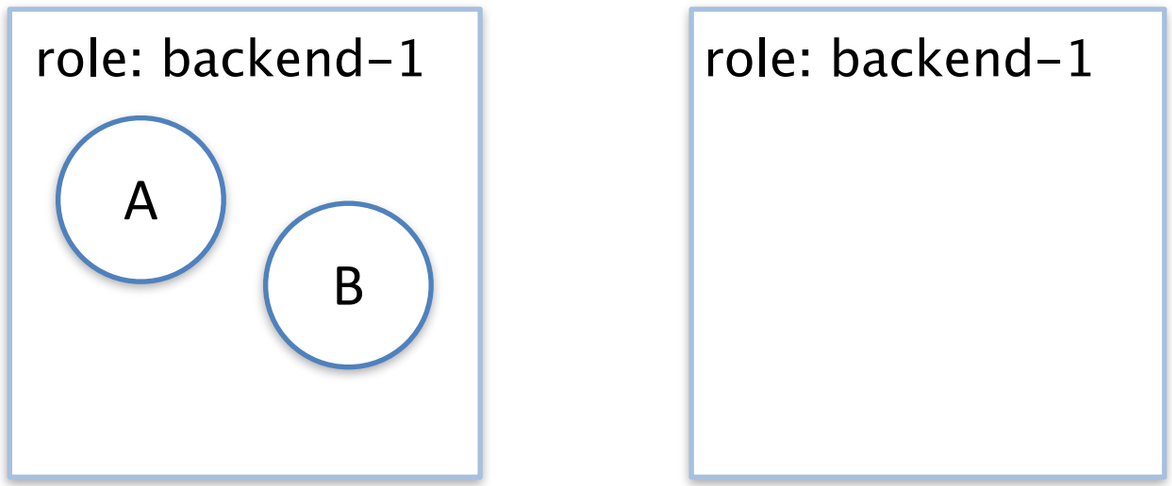
Usage in a Cluster

- **single writer**
  - cluster singleton
  - cluster sharding

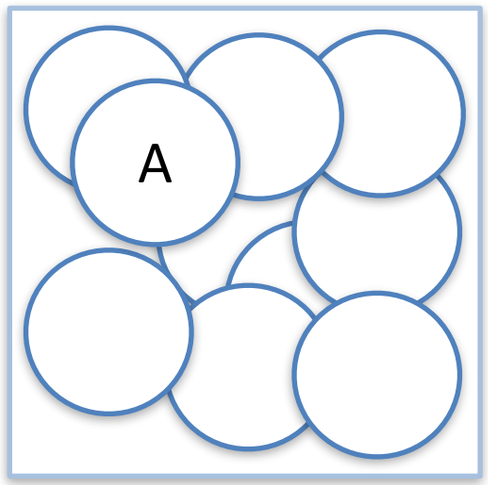
Cluster Singleton

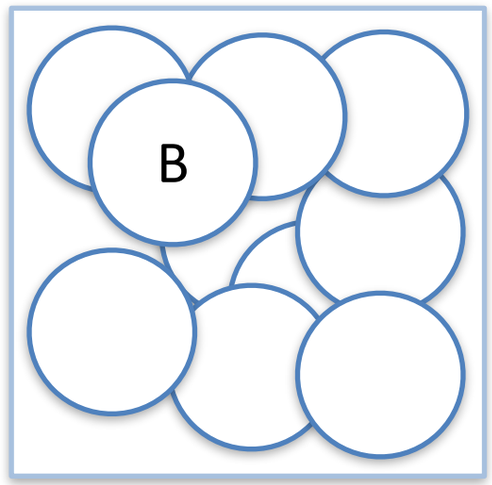


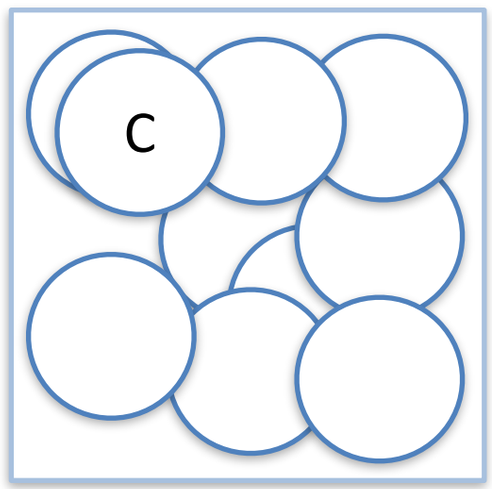
Cluster Singleton

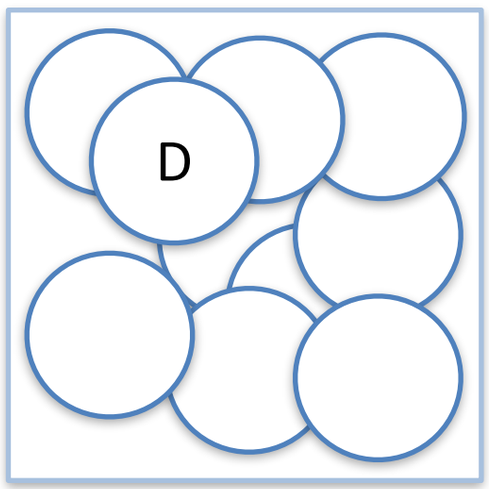


Cluster Sharding

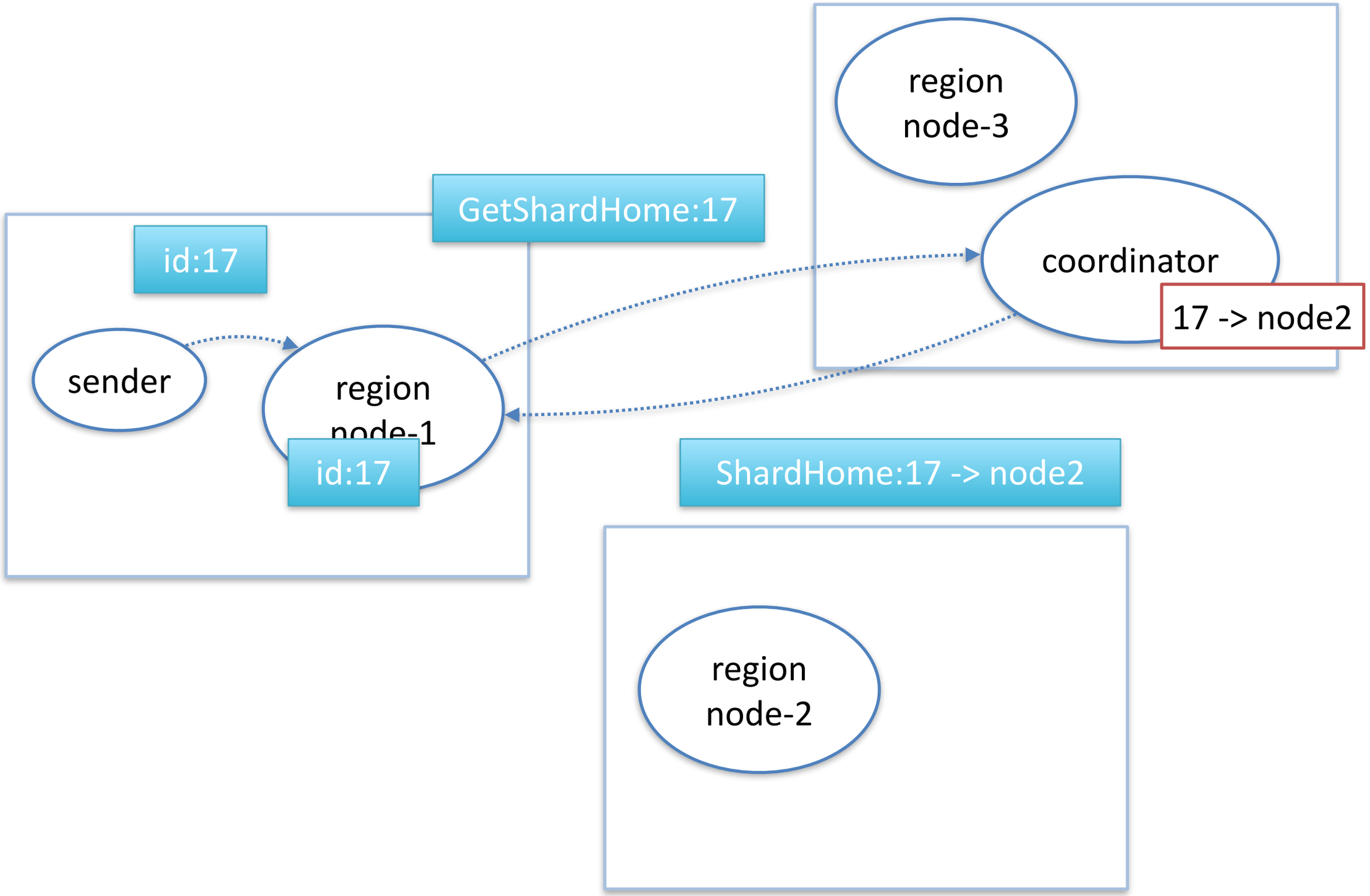




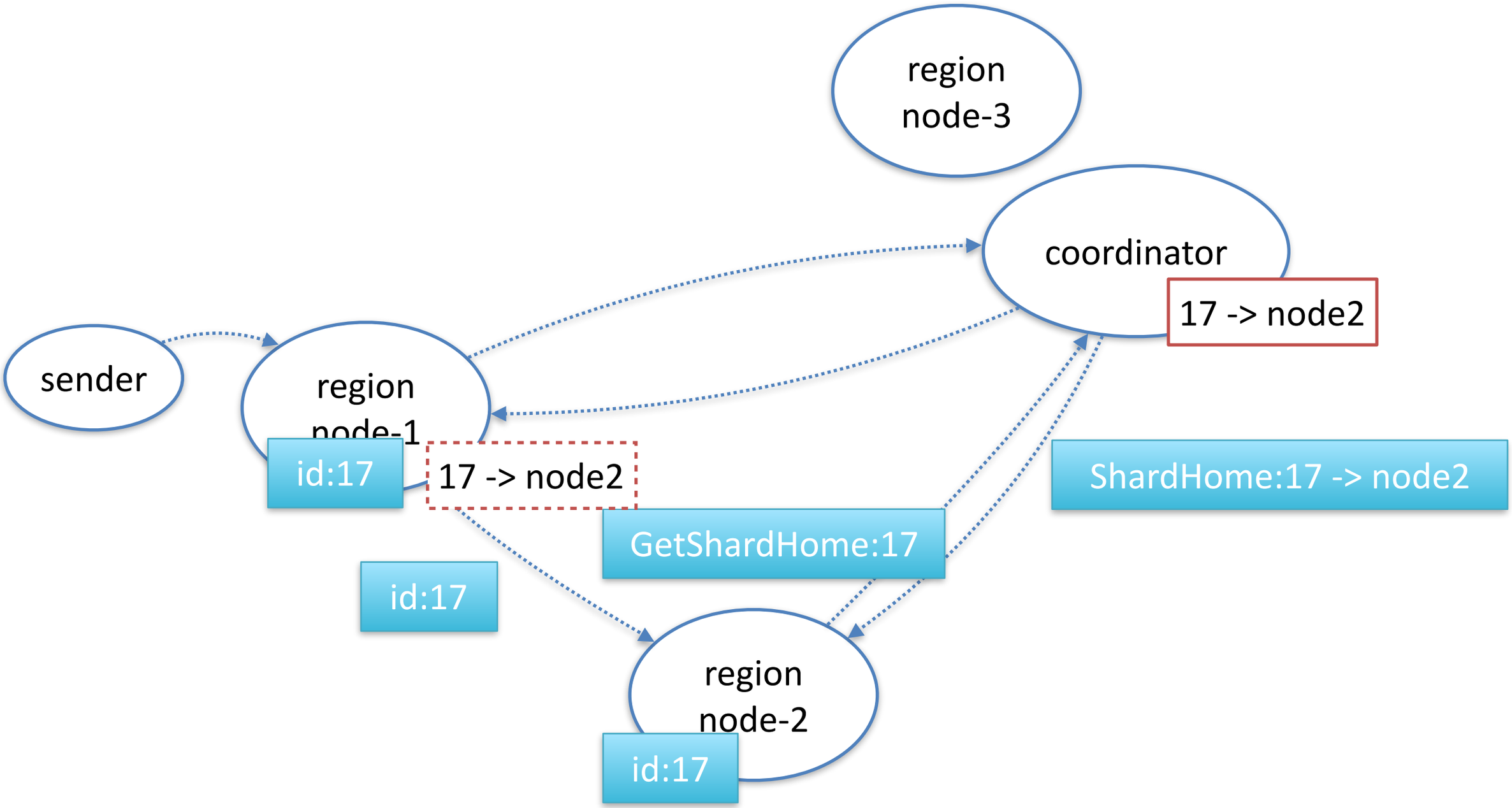




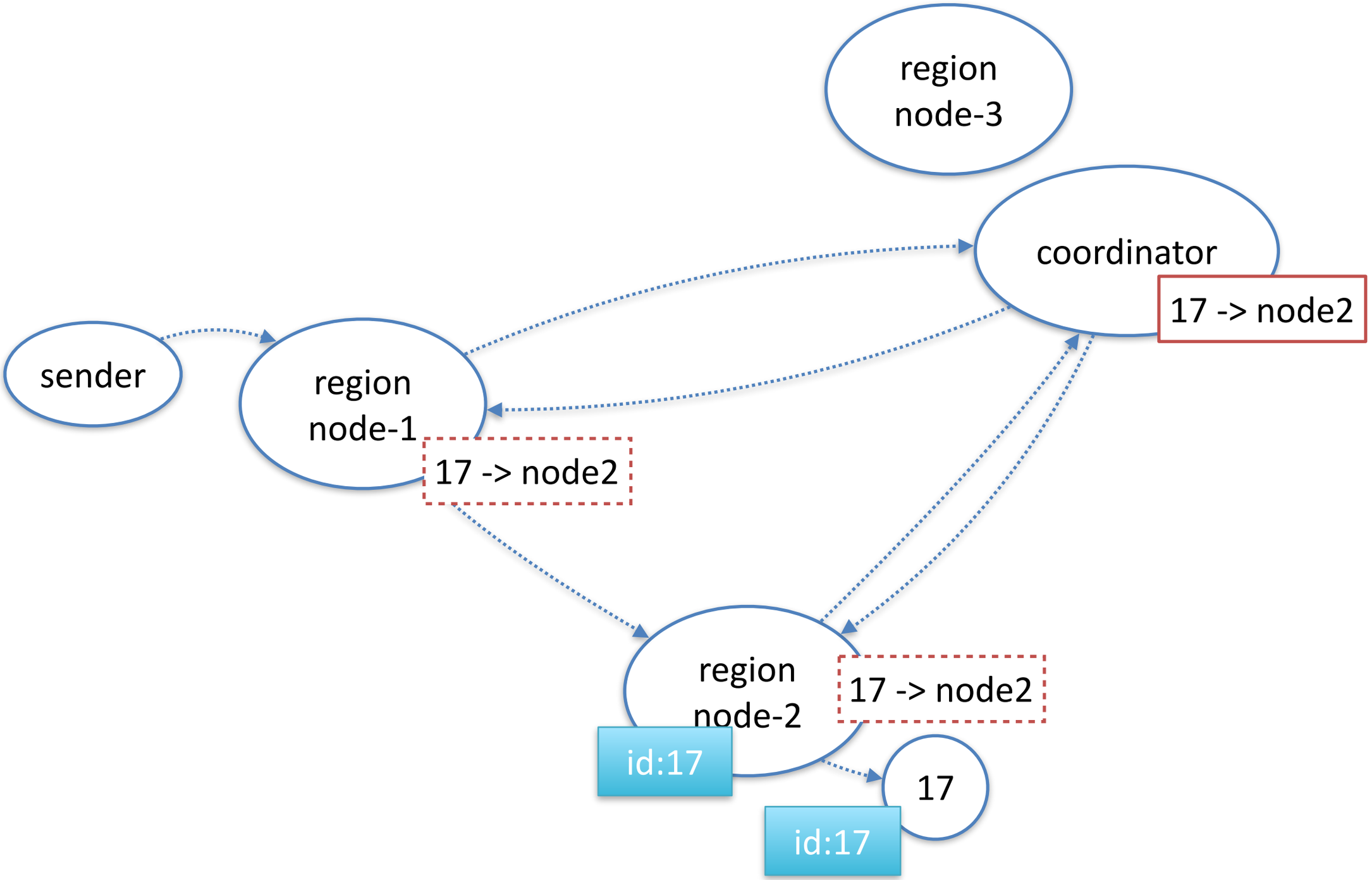
Cluster Sharding



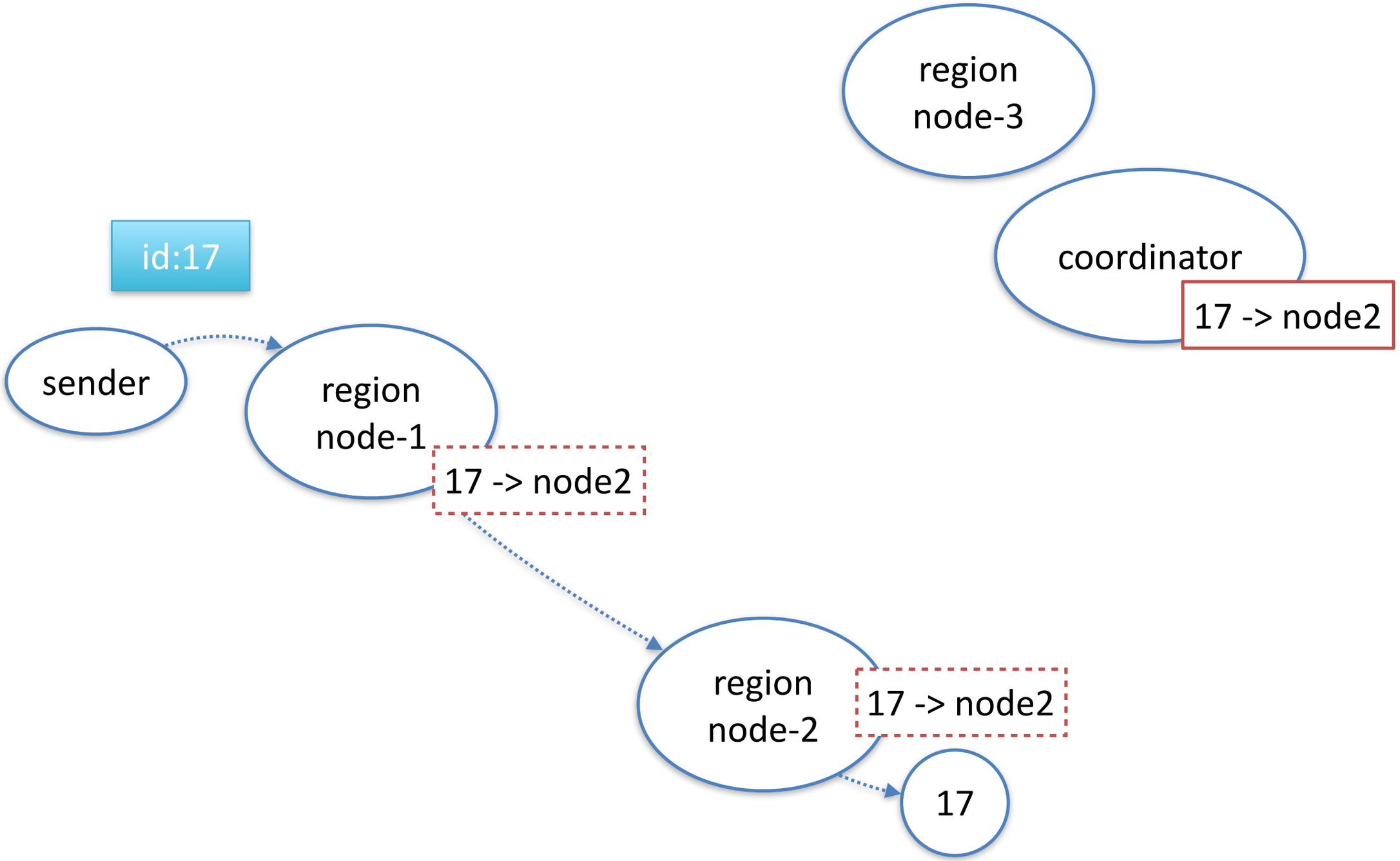
Cluster Sharding



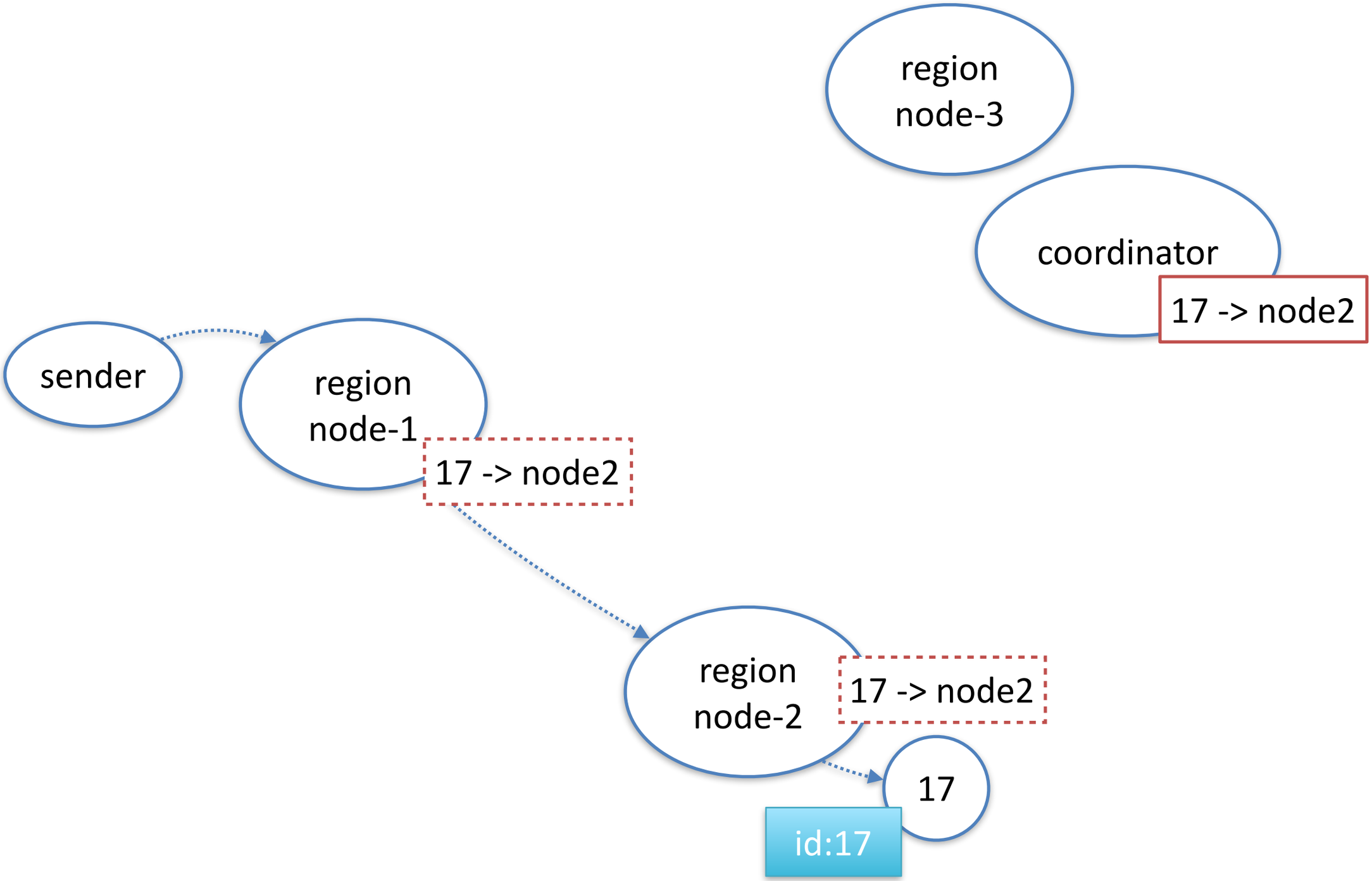
Cluster Sharding



Cluster Sharding



Cluster Sharding



Cluster Sharding

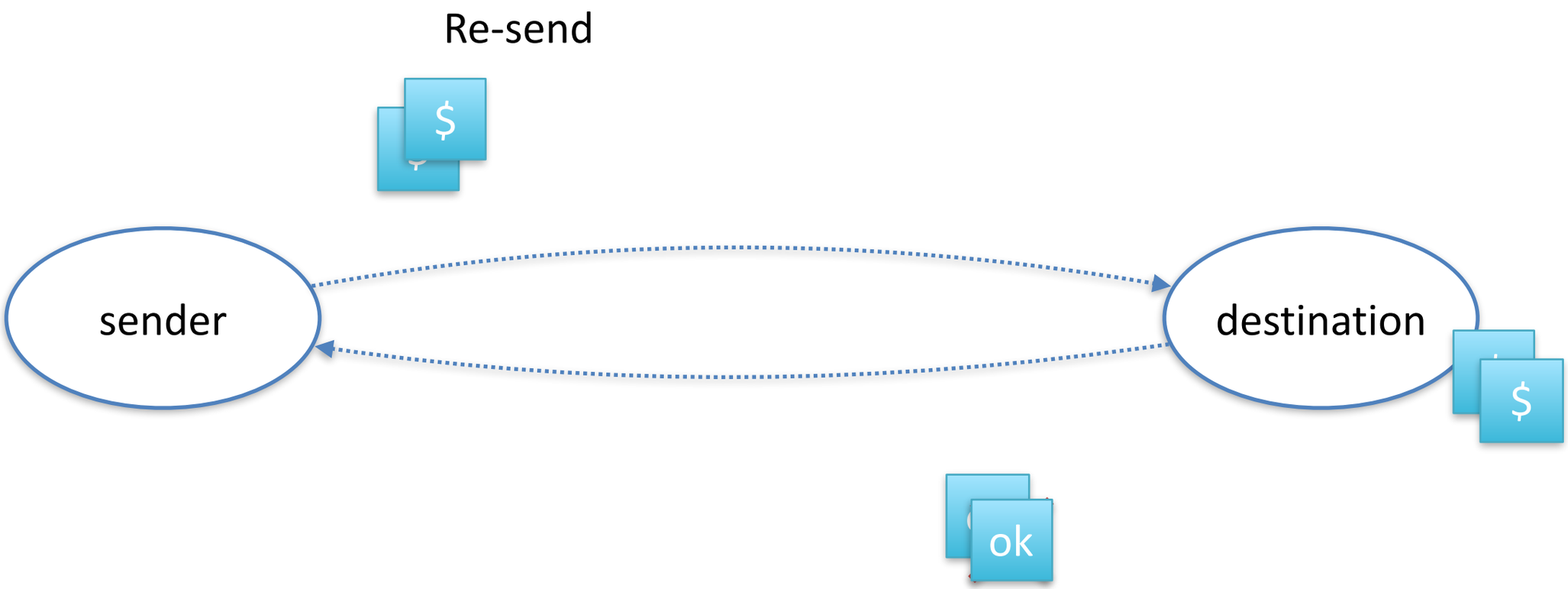
```
val idExtractor: ShardRegion.IdExtractor = {
  case cmd: Command => (cmd.postId, cmd)
}

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)
```

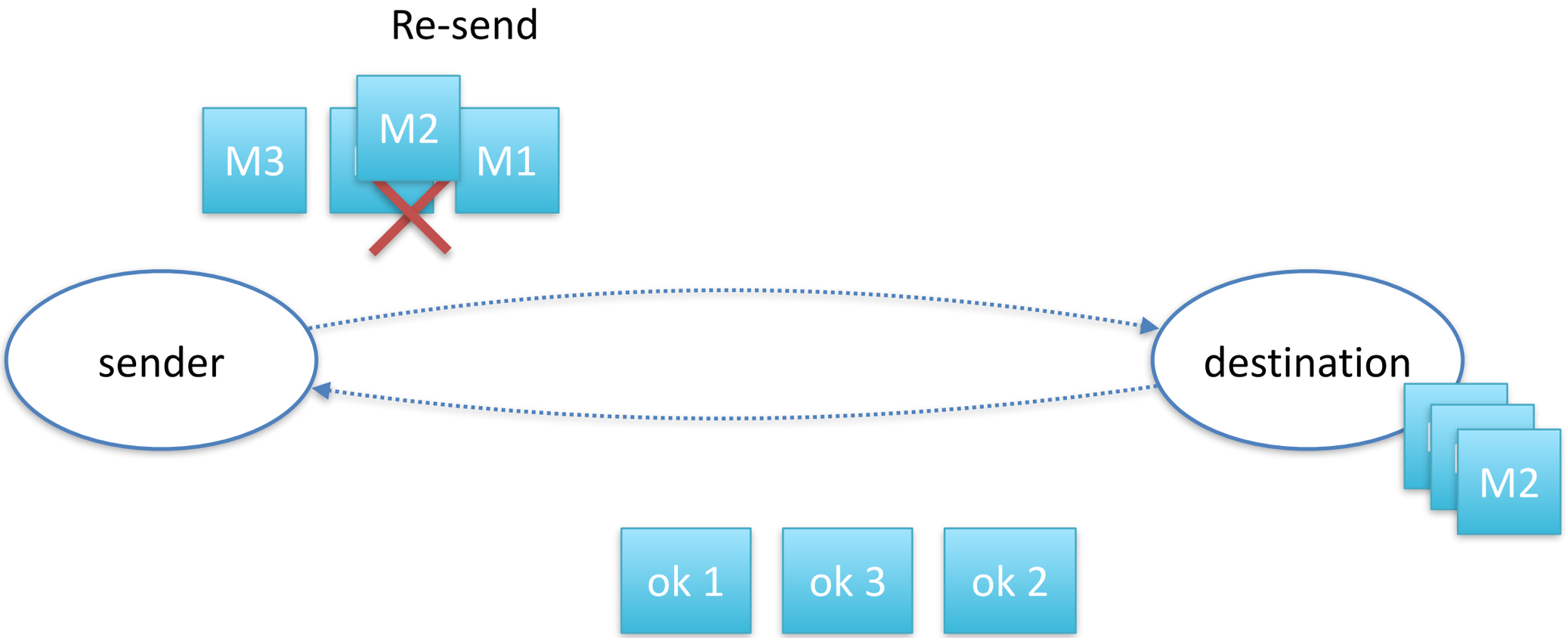
```
val blogPostRegion: ActorRef =
  ClusterSharding(context.system).shardRegion(BlogPost.shardName)

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

Lost messages

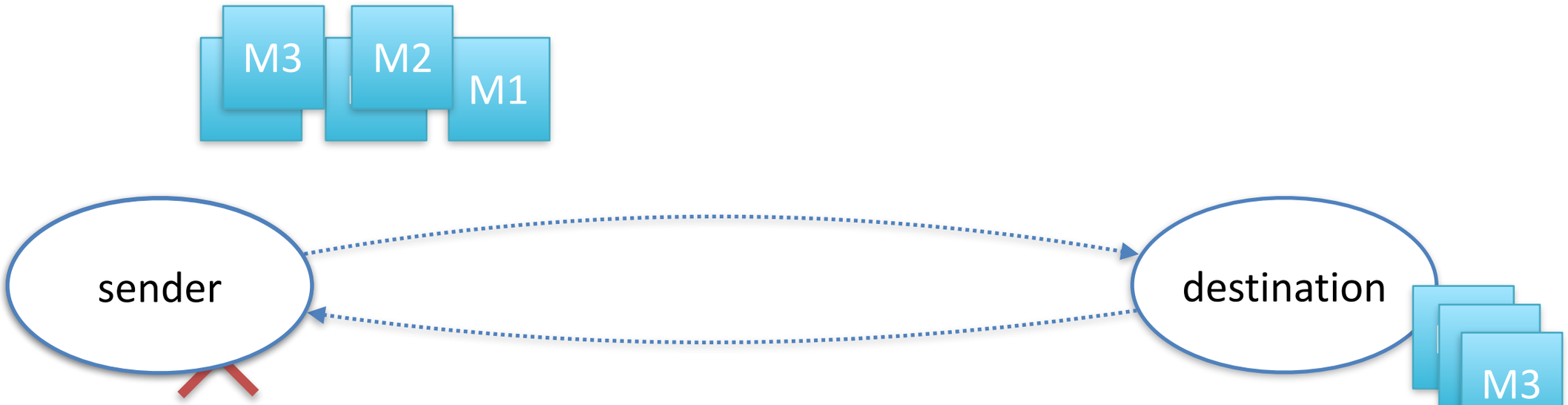
At-least-once delivery - duplicates



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

PersistentActor with AtLeastOnceDelivery

```
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
```

```
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)
  }
}
```

Try it now

Try it now() // !

typesafe.com/activator

akka-sample-persistence-scala

https://github.com/hseeberger/akkamazing

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

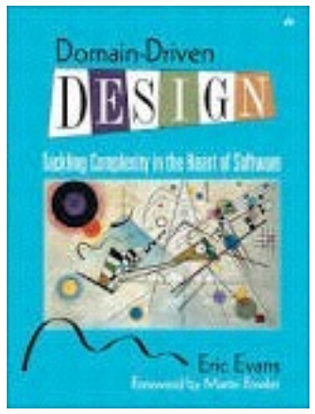
Links

- Official docs: http://doc.akka.io/docs/akka/2.3.0/scala/persistence.html
- Patrik's Slides & Webinar: http://www.slideshare.net/patriknw/akka-persistence-webinar
- Papers:
  - Sagas http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf
  - Life beyond Distributed Transactions: http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf
- Pics:
- http://misaspuppy.deviantart.com/art/Messenger-s-Cutie-Mark-A-Flying-Envelope-291040459

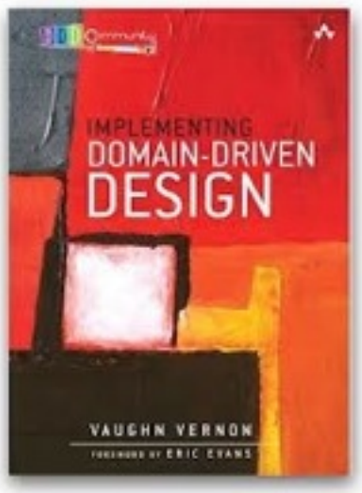
Mailing List

groups.google.com/forum/#!forum/akka-user

Links



Eric Evans:
"The DDD book"
Talk: "Acknowledging CAP at the Root"



Vaughn Vernon's Book and Talk

akka-user @ google groups

Dzięki!
Thanks!
ありがとう！

ktoso @ typesafe.com
twitter: ktosopl
github: ktoso
blog: project13.pl
team blog: letitcrash.com

ScalaCamp #6 2014
@ Kraków, PL