Persistent Actor State

Principles of Functional Programming

Roland Kuhn

Persistent Actor State

Actors representing a stateful resource

- shall not lose important state due to (system) failure
- must persist state as needed
- must recover state at (re)start

Persistent Actor State

Actors representing a stateful resource

- shall not lose important state due to (system) failure
- must persist state as needed
- must recover state at (re)start

Two possibilities for persisting state:

- in-place updates
- persist changes in append-only fashion

Changes vs. Current State

Benefits of persisting current state:

- Recovery of latest state in constant time.
- Data volume depends on number of records, not their change rate.

Changes vs. Current State

Benefits of persisting current state:

- Recovery of latest state in constant time.
- Data volume depends on number of records, not their change rate.

Benefits of persisting changes:

- History can be replayed, audited or restored.
- Some processing errors can be corrected retroactively.
- Additional insight can be gained on business processes.
- Writing an append-only stream optimizes IO bandwidth.
- Changes are immutable and can freely be replicated.

Snapshots

Immutable snapshots can be used to bound recovery time.

Persistence Primitive

- being persistent means “taking notes”

```
persist(MyEvent(...)) { event =>
  // now <event> is persisted
  doSomethingWith(event)
}
```

Event Example (1)

```scala
case class NewPost(text: String, id: Long)
case class BlogPosted(id: Long)
case class BlogNotPosted(id: Long, reason: String)

sealed trait Event
case class PostCreated(text: String) extends Event
case object QuotaReached extends Event

case class State(posts: Vector[String], disabled: Boolean) {
  def updated(e: Event): State = e match {
    case PostCreated(text) => copy(posts = posts :+ text)
    case QuotaReached      => copy(disabled = true)
  }
}
```

Event Example (2)

```
class UserProcessor extends PersistentActor {
  var state = State(Vector.empty, false)
  def receiveCommand = {
    case NewPost(text, id) =>
      if (state.disabled) sender() ! BlogNotPosted(id, ”quota reached”)
      else { persist(PostCreated(text)) { e =>
                 updateState(e)
                 sender() ! BlogPosted(id) }
             persist(QuotaReached)(updateState) }
  }
  def updateState(e: Event) { state = state.updated(e) }
  def receiveRecover = { case e: Event => updateState(e) }
}
```

When to Apply the Events?

- Applying after persisting leaves actor in stale state.

When to Apply the Events?

- Applying after persisting leaves actor in stale state.
- Applying before persisting means replay can lead to different state.

When to Apply the Events?

- Applying after persisting leaves actor in stale state.
- Applying before persisting means replay can lead to different state.

It is a trade-off between throughput and consistency.

Event Example (3)

```
case NewPost(text, id) =>
  if (!state.disabled) {
    val created = PostCreated(text)
    update(created)
    update(QuotaReached)
    persistAsync(created)(sender() ! BlogPosted(id))
    persistAsync(QuotaReached)(_ => ())
  } else sender() ! BlogNotPosted(id, ”quota reached”)
```

At-Least-Once Delivery (1)

- Guaranteeing delivery means retrying until successful
- Retries are the sender's responsibility
- The recipient needs to acknowledge receipt
- Lost receipts lead to duplicate deliveries $\Rightarrow$ *at-least-once*

At-Least-Once Delivery (1)

- Guaranteeing delivery means retrying until successful
- Retries are the sender's responsibility
- The recipient needs to acknowledge receipt
- Lost receipts lead to duplicate deliveries $\Rightarrow$ *at-least-once*

- Retrying means taking note that the message needs to be sent
- Acknowledgement means taking note of the receipt of the confirmation

At-Least-Once Delivery (2)

```scala
class UserProcessor(publisher: ActorPath)
    extends PersistentActor with AtLeastOnceDelivery {
  def receiveCommand = {
    case NewPost(text, id) =>
      persist(PostCreated(text)) { e =>
        deliver(publisher, PublishPost(text, _))
        sender() ! BlogPosted(id) }
  }
  def receiveRecover = {
    case PostCreated(text) => deliver(publisher, PublishPost(text, _))
  }
}
```

At-Least-Once Delivery (3)

```scala
class UserProcessor(publisher: ActorPath)
    extends PersistentActor with AtLeastOnceDelivery {
  def receiveCommand = {
    case NewPost(text, id) =>
      persist(PostCreated(text)) { e =>
        deliver(publisher, PublishPost(text, _))
        sender() ! BlogPosted(id) }
    case PostPublished(id) => confirmDelivery(id)
  def receiveRecover = {
    case PostCreated(text) => deliver(publisher, PublishPost(text, _))
    case PostPublished(id) => confirmDelivery(id)
  }
}
```

Exactly-Once Delivery (1)

- *At-least-once* delivery leads to duplicates at the receiver
- The receiver needs to remember what it has already done to avoid redoing it

Exactly-Once Delivery (2)

```
class Publisher extends PersistentActor {
  var expectedId = 0L
  def receiveCommand = {
    case PublishPost(text, id) =>
      if (id > expectedId) () // ignore, not yet ready for that
      else if (id < expectedId) sender() ! PostPublished(id)
      else persist(PostPublished(id)) { e =>
             sender() ! e
             // modify website
             expectedId += 1
           }
  }
  def receiveRecover = { case PostPublished(id) => expectedId = id + 1 }
}
```

When to Perform External Effects?

Performing the effect and persisting that it was done cannot be atomic.

- Perform it before persisting for at-least-once semantics.
- Perform it after persisting for at-most-once semantics.

This choice needs to be made based on the underlying business model.

When to Perform External Effects?

Performing the effect and persisting that it was done cannot be atomic.

- Perform it before persisting for at-least-once semantics.
- Perform it after persisting for at-most-once semantics.

This choice needs to be made based on the underlying business model.

If processing is *idempotent* then using at-least-once semantics achieves *effectively exactly-once processing*.

Summary

- Actors persist facts that represent changes to their state.
- Events can be replicated and used to inform other components.
- Recovery replays past events to reconstruct state; snapshots reduce this cost.