Programming Paradigms for Concurrency
Lecture 12 – Failure Handling with Actors

Based on a course on
Principles of Reactive Programming
by Martin Odersky, Erik Meijer, Roland Kuhn

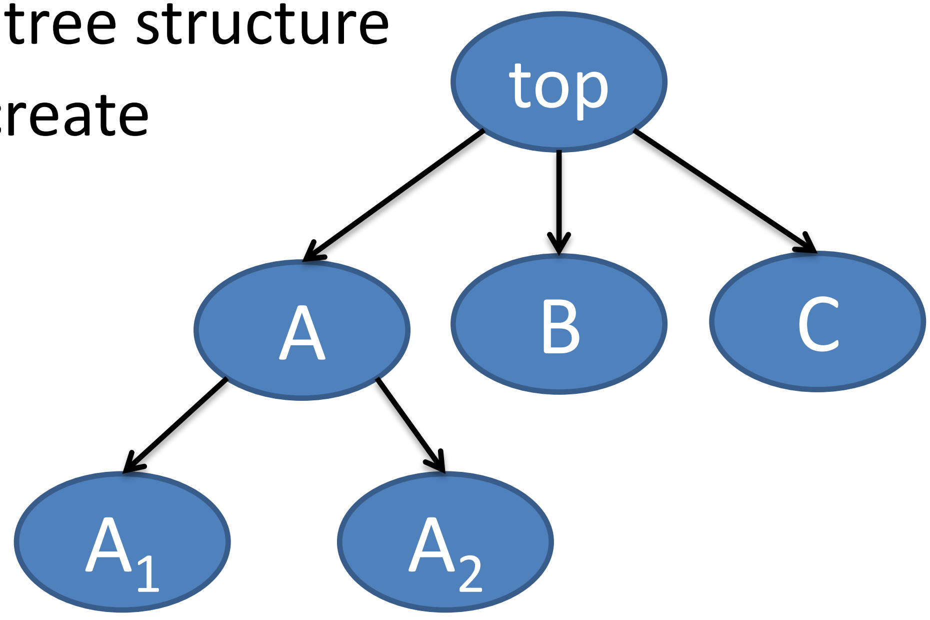
Modified by
Thomas Wies
New York University

Failure Handling in Asynchronous Systems

- Where should failures go?
  - reify as message
  - send to a known address

Actor Supervision

- Actor systems enable containment and automatic response to failures
  - failed actor is terminated or restarted
  - decision must be taken by one other actor (the supervisor)
  - supervised actors form a tree structure
  - the supervisor needs to create its subordinates



Supervisor Strategy

In Akka the parent declares how its child `Actors` are supervised:

```
class Manager extends Actor {
  override val supervisorStrategy = OneForOneStrategy() {
    case _: DBException => Restart // reconnect to DB
    case _: ActorKilledException => Stop
    case _: ServiceDownException => Escalate
  }
  ...
  context.actorOf(Props[DBActor], ”db”)
  context.actorOf(Props[ImportantServiceActor], ”service”)
  ...
}
```

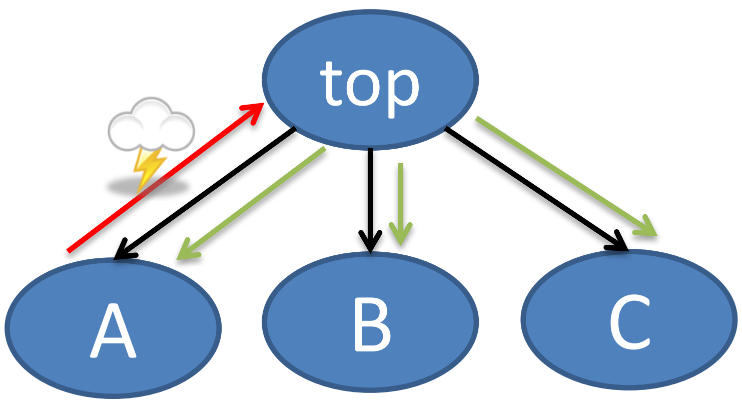
Supervisor Strategy (cont'd)

Failure is sent and processed like a message:

```
class Manager extends Actor {
  var restarts = Map.empty[ActorRef, Int].withDefaultValue(0)
  override val supervisorStrategy = OneForOneStrategy() {
    case _: DBException =>
      restarts(sender) match {
        case toomany if toomany > 10 =>
          restarts -= sender; Stop
        case n =>
          restarts = restarts.updated(sender, n + 1); Restart
      }
  }
}
```

Supervisor Strategy (cont'd)

If decision applies to all children: `AllForOneStrategy`



Supervisor Strategy (cont’d)

If decision applies to all children: `AllForOneStrategy`

Simple rate trigger included:

- allow a finite number of restarts
- allow a finite number of restarts in a time window
- if restriction violated then `Stop` instead of `Restart`

```
OneForOneStrategy(maxNrOfRetries = 10,
                  withinTimeRange = 1.minute) {
  case _: DBException => Restart // will turn into Stop
}
```

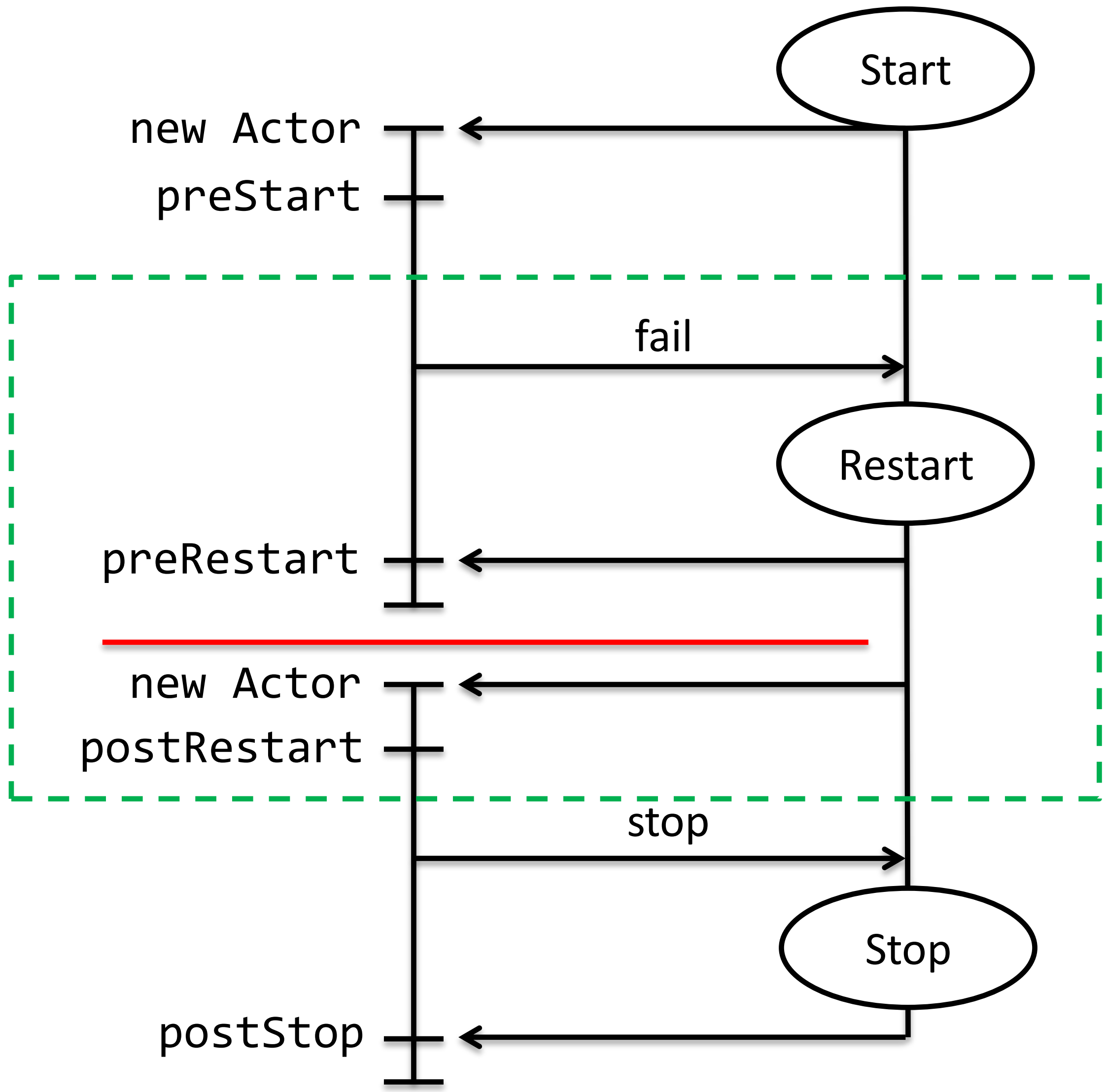
Actor Identity

Recovery by restart requires stable identifier to refer to the service:

- in Akka the ActorRef stays valid after a restart

Actor Lifecycle

- start
- restart*
- stop



Actor Lifecycle Hooks

```
trait Actor {
  def preStart(): Unit = {}
  def preRestart(reason: Throwable,
                 message: Option[Any]): Unit = {
    context.children foreach (context.stop(_))
    postStop()
  }
  def postRestart(reason: Throwable): Unit = {
    preStart()
  }
  def postStop(): Unit = {}
  ...
}
```

The Default Lifecycle

```scala
class DBActor extends Actor {
  val db = DB.openConnection(...)
  ...
  override def postStop(): Unit = {
    db.close()
  }
}
```

In this model the actor is fully reinitialized during restart.

Lifecycle-Spanning Restart

```scala
class Listener(source: ActorRef) extends Actor {
  override def preStart() { source ! RegisterListener(self) }
  override def preRestart(reason: Throwable,
                          message: Option[Any]) {}
  override def postRestart(reason: Throwable) {}
  override def postStop() { source ! UnregisterListener(self) }
}
```

Actor-local state cannot be kept across restarts, only external state can be managed like this.

Child actors not stopped during restart will be restarted recursively.

Lifecyle Monitoring

The only observable transition occurs when stopping an actor:

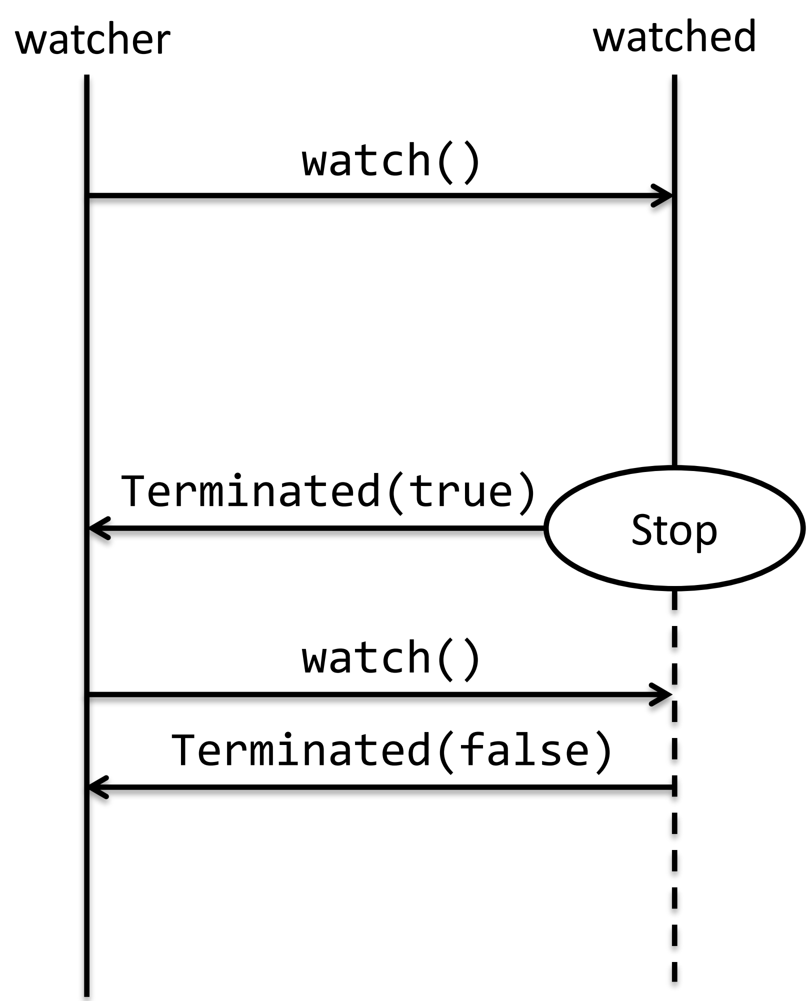
- having an `ActorRef` implies actor has been live
  (at some earlier point)
- restarts are not externally visible
- after stop there will be no more responses

The DeathWatch API

```
trait ActorContext {
  def watch(target: ActorRef): ActorRef
  def unwatch(target: ActorRef): ActorRef
  ...
}

case class Terminated private[akka] (actor: ActorRef)
  (val existenceConfirmed: Boolean,
   val addressTerminated: Boolean)
  extends AutoReceiveMessage with PossiblyHarmful
```

The `DeathWatch` API (cont'd)



The DeathWatch API (cont'd)

```
trait ActorContext {
  def watch(target: ActorRef): ActorRef
  def unwatch(target: ActorRef): ActorRef
  ...
}

case class Terminated private[akka] (actor: ActorRef)
  (val existenceConfirmed: Boolean,
   val addressTerminated: Boolean)
  extends AutoReceiveMessage with PossiblyHarmful
```

Applying DeathWatch to Controller/Getter (1)

```
class Getter(url: String, depth: Int) extends Actor {
  ...
  def receive = {
    case body: String =>
      for (link <- findLinks(body))
        context.parent ! Controller.Check(link, depth)
      context.stop(self)
    case _: Status.Failure => context.stop(self)
  }
}
```

Simply terminating the `Getter` when it is done uses `DeathWatch` to signal end of conversation.

The Children List

Each actor maintains a list of the actors it created:

- the child has been entered when `context.actorOf` returns
- the child has been removed when Terminated is received
- an actor name is available iff there is no such child

```
trait ActorContext {
  def children: Iterable[ActorRef]
  def child(name: String): Option[ActorRef]
  ...
}
```

Applying DeathWatch to Controller/Getter (2)

```
class Controller extends Actor with ActorLogging {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 5) {
      case _: Exception => SupervisorStrategy.Restart
    }
  def receive = {
    case Check(url, depth) =>
      if (!cache(url) && depth > 0)
        context.watch(context.actorOf(getterProps(url, depth - 1)))
      cache += url
    case Terminated(_) =>
      if (context.children.isEmpty) context.parent ! Result(cache)
    case ReceiveTimeout => context.children foreach context.stop
  }
  ...
}
```

Lifecycle Monitoring for Fail-Over

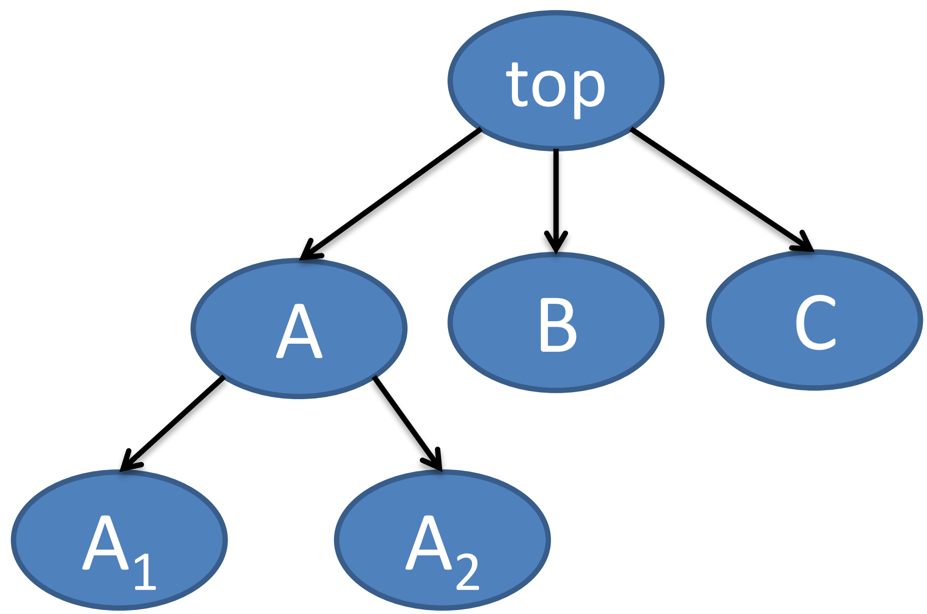
```
class Manager extends Actor {
  def prime(): Receive = {
    val db = context.actorOf(Props[DBActor], ”db”)
    context.watch(db)

    {
      case Terminated(‘db‘) => context.become(backup())
    }
  }
  def backup(): Receive = { ... }
  def receive = prime()
}
```

The Error Kernel

Keep important data near the root, delegate risk to the leaves.

- restarts are recursive (supervised actors are part of the state)
- restarts are more frequent near the leaves
- avoid restarting `Actors` with important state



Application to Receptionist (1)

- Always stop `Controller` if it has a problem.
- React to `Terminated` to catch cases where no `Result` was sent.
- Discard `Terminated` after `Result` was sent.

```
class Receptionist extends Actor {
  override def supervisorStrategy =
    SupervisorStrategy.stoppingStrategy
  ...
}
```

Application to Receptionist (2)

```
class Receptionist extends Actor {
  ...
  def runNext(queue: Vector[Job]): Receive = {
    reqNo += 1
    if (queue.isEmpty) waiting
    else {
      val controller = context.actorOf(controllerProps,
                                         s”c$reqNo”)
      context.watch(controller)
      controller ! Controller.Check(queue.head.url, 2)
      running(queue)
    }
  }
}
```

Application to Receptionist (3)

```
def running(queue: Vector[Job]): Receive = {
  case Controller.Result(links) =>
    val job = queue.head
    job.client ! Result(job.url, links)
    context.stop(context.unwatch(sender))
    context.become(runNext(queue.tail))
  case Terminated(_) =>
    val job = queue.head
    job.client ! Failed(job.url)
    context.become(runNext(queue.tail))
  case Get(url) =>
    context.become(enqueueJob(queue, Job(sender, url)))
}
```

Digression: the `EventStream` (1)

Actors can direct messages only at known addresses.

The `EventStream` allows publication of messages to an unknown audience.

Every actor can optionally subscribe to (parts of) the `EventStream`.

```
trait EventStream {
  def subscribe(subscriber: ActorRef, topic: Class[_]):
    Boolean
  def unsubscribe(subscriber: ActorRef, topic: Class[_]):
    Boolean
  def unsubscribe(subscriber: ActorRef): Unit
  def publish(event: AnyRef): Unit
}
```

Digression: the EventStream (2)

```
class Listener extends Actor {
  context.system.eventStream.subscribe(self, classOf[LogEvent])
  def receive = {
    case e: LogEvent => ...
  }
  override def postStop(): Unit = {
    context.system.eventStream.unsubscribe(self)
  }
}
```

Where do Unhandled Messages Go?

`Actor.Receive` is a partial function, the behavior may not apply.

Unhandled messages are passed into the unhandled method:

```
trait Actor {
  ...
  def unhandled(message: Any): Unit = message match {
    case Terminated(target) =>
      throw new DeathPactException(target)
    case msg =>
      context.system.eventStream.publish
        (UnhandledMessage(msg, sender, self))
  }
}
```

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

Changes vs. Current State

Benefits of persisting current state:

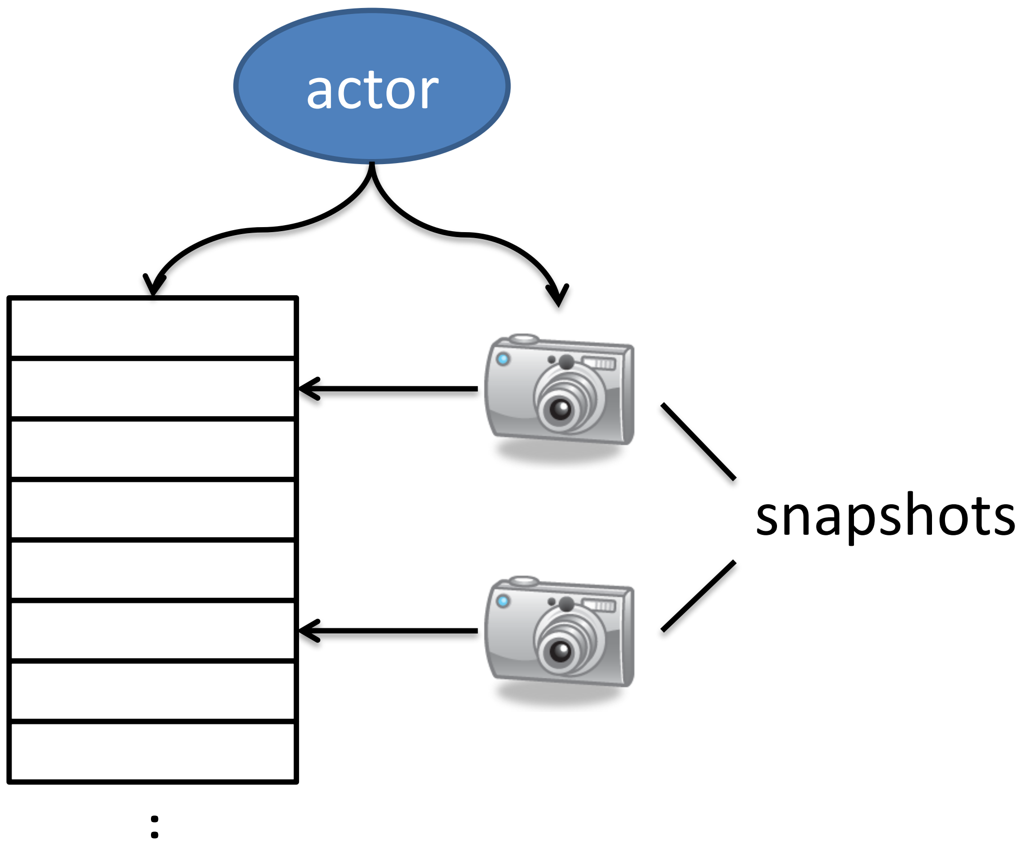
- Recovery of latest state in constant time.
- Data volume depends on number of records, not their change rate.

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 be freely replicated.

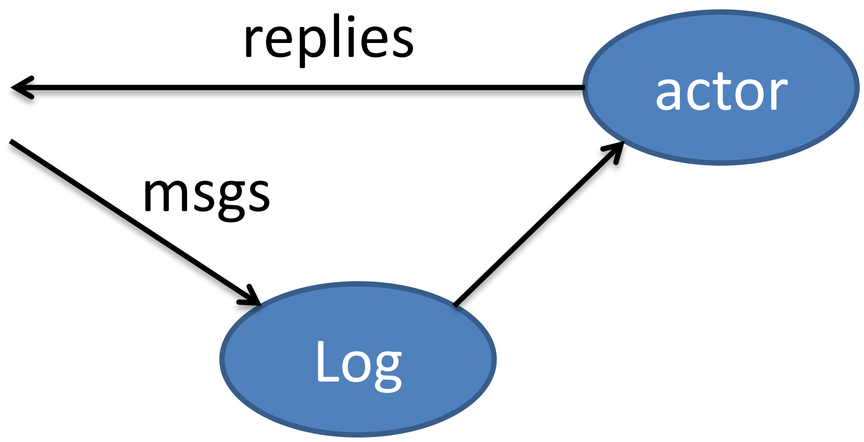
Snapshots

Immutable snapshots can be used to bound recovery time.



Command Sourcing

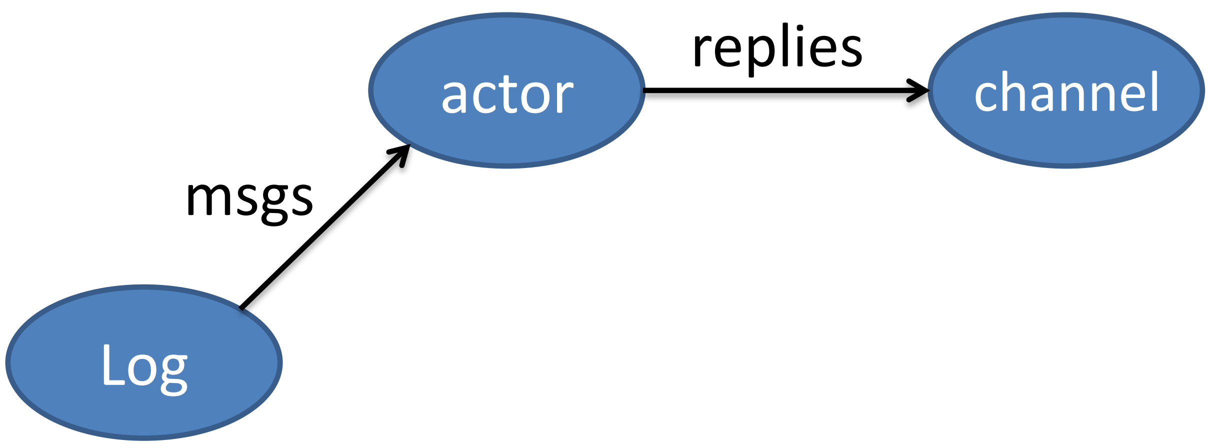
Command Sourcing: Persist the command before processing it, persist acknowledgement when processed.



Recovery of State

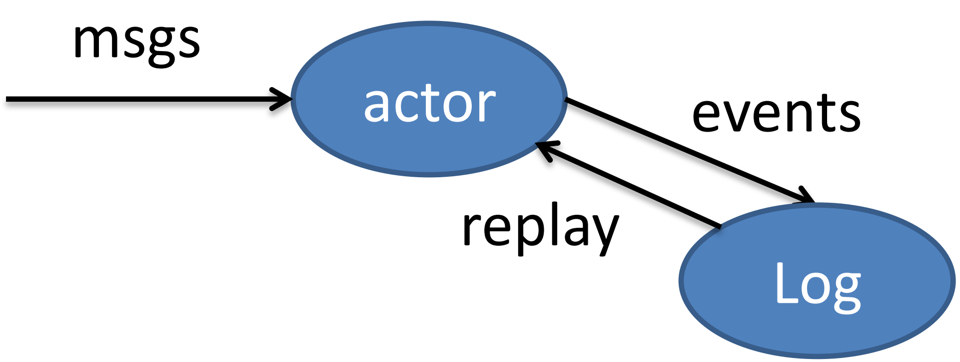
During recovery

- all commands are replayed in order to recover state.
- a persistent `Channel` discards messages already sent to other actors.



Event Sourcing

Event Sourcing: Generate change requests (“events”) instead of modifying local state; persist and apply them.



Event Example (1)

```scala
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 Actor {
  var state = State(Vector.empty, false)
  def receive = {
    case NewPost(text) =>
      if (!state.disabled)
        emit(PostCreated(text), QuotaReached)
    case e: Event =>
      state = state.updated(e)
  }
  def emit(events: Event*) = ... // send to log
}
```

When to Apply the Events?

- Applying after persisting leaves actor in stale state.
- Applying before persisting relies on regenerating during replay.

When to Apply the Events?

- Applying after persisting leaves actor in stale state.
- Applying before persisting relies on regenerating during replay.

When to Apply the Events?

- Applying after persisting leaves actor in stale state.
- Applying before persisting relies on regenerating during replay.

Trading performance for consistency:

- Do not process new messages while waiting for persistence.

The Stash Trait

```
class UserProcessor extends Actor with Stash {
  var state: State = ...
  def receive = {
    case NewPost(text) if !state.disabled =>
      emit(PostCreated(text), QuotaReached)
      context.become(waiting(2), discardOld = false)
    }
  def waiting(n: Int): Receive = {
    case e: Event =>
      state = state.updated(e)
      if (n == 1) { context.unbecome(); unstashAll() }
      else context.become(waiting(n - 1))
    case _ => stash()
  }
}
```

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 concrete application.

Summary

- Actors can persist incoming messages or generated events.
- Events can be replicated and used to inform other components.
- Recovery replays past commands or events; snapshots reduce this cost.
- Actors can defer handling certain messages by using the `Stash` trait.