Handling failures and actor state

Weiming Huang and Tarik Blaoui
CSCI Intro to Reactive Programming

Akka

What is akka?

-Project begun by Jonas Boner in 2008

-Cpus are not getting much faster

-Multicore cpu

-Built on the actor model: conceptual concurrent computational model

Actors

-Has mailbox

-Can only communicate by exchanging messages

-Upon receiving a message an actor can

1. Send a number of messages to other actors it knows
2. Create new actors
3. Designate how to handle the next message

-Process messages

Actor state

-Actor objects contain variables that reflect state

-Actor responsible for its own state

-State changes can occur when messages are received

-Actors do not have to worry about concurrency

-Only processes one message at a time

Persistent Actor State

-Store actor states

-Read stored actor states

-Recover state at restart

- Two ways of persisting state

1) In place update
2) Persist change in append-only fashion

Typical failure handling

-Exception: abnormal event disrupts normal flow

-Exception handling using try/catch/finally

```
try {
  val writer = new FileWriter("test.out")
  writer.write(bigBlockOfData)
} catch {
  case e: FileOutputException =>
    println("Failed to write data.")
} finally {
  writer.close()
}
```

Try

-Allows for the execution of code where an exception is expected to be thrown

-Success and failure

```
def readTextFile(filename: String): Try[List[String]] = {
    Try(Source.fromFile(filename).getLines.toList)
}

val filename = "/etc/passwd"
readTextFile(filename) match {
    case Success(lines) => lines.foreach(println)
    case Failure(f) => println(f)
}
```

Failing fast/Let it crash

-If failure occurs, make sure the failure is caught and a response is deployed

-Single responsibility principle

-”Let it crash” - phrase and philosophy used by akka engineering team

-Let actor fail and restart or recreate and try again

Fault tolerance

-Allowing for some failure, isolate actors

-Actor crashes, fixed by parent actor

-Difference between failures and validation errors

-Validation error: data sent is not valid

-Failure: something unexpected or outside the control of actor

Supervisor

-Any actor in the system has a parent actor

-Guardian actors at the top

-Actors act as a supervisor for their children

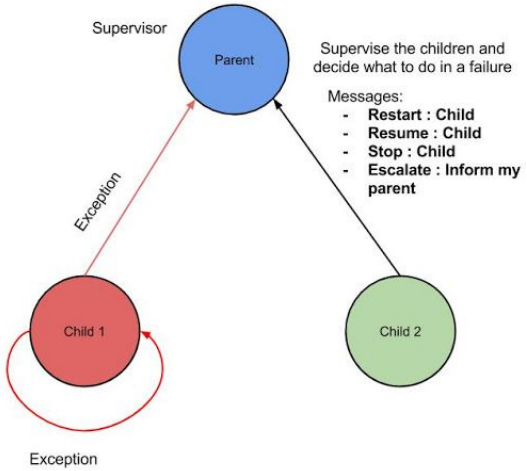
-Handled in structure called SupervisorStrategy

3 Types of failures

- Systematic (i.e. programming) error for the specific message received
- (Transient) failure of some external resource used during processing the message
- Corrupt internal state of the actor

Supervision actions

- The supervisor actor hands tasks to the child actor
- The supervisor deals with Throws when a child sends a signals
- Depending on the nature of the Throw or work the child is doing the supervisor has 4 options on what it can order the child:
- Option 1: Restart
- Option 2: Resume
- Option 3: Stop
- Option 4: Escalate



Signals

- The supervisor is itself a subordinate to another supervisor this affects how it handles signals. Subordinates can be supervisors to their own subordinates.
- **Resume** the subordinate and all its subordinates keep their accumulated internal state
- **Restart** clear the subordinate and all its subordinates internal state and restart them. (By default restart will terminate the subordinates of the subordinate)
- **Stop** Terminate the subordinate
- **Escalate** pass the exception to the supervisor’s supervisor and let it handle it. It fails itself.

Actor life cycle methods

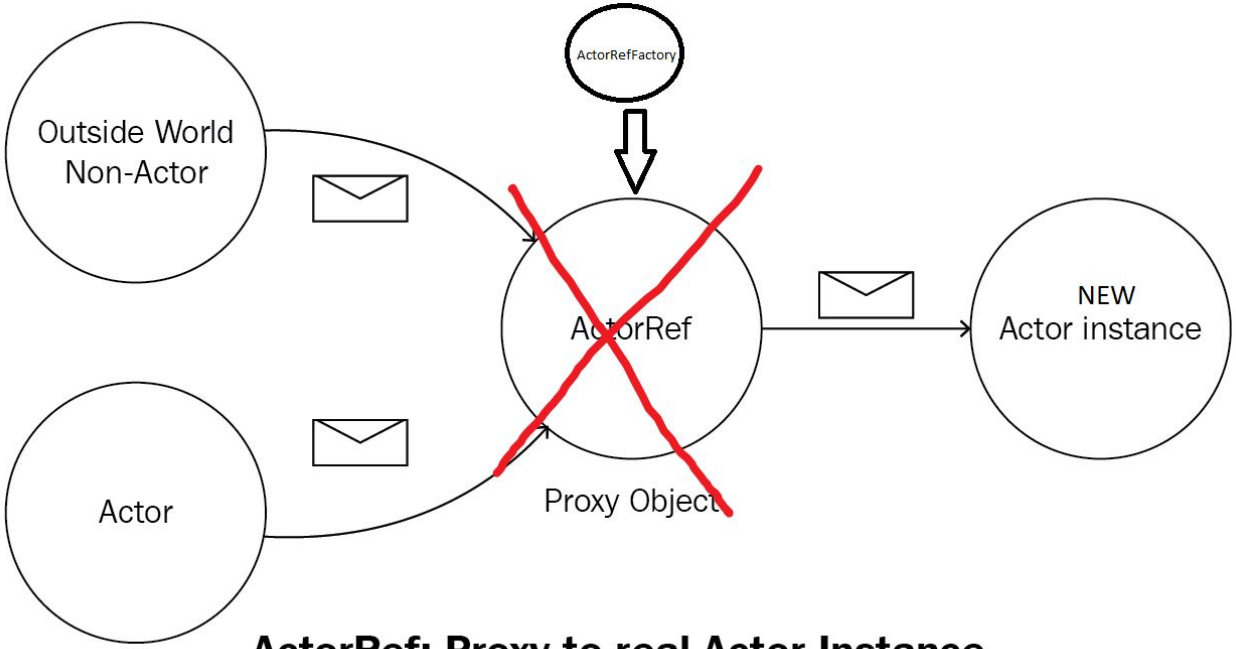
- Actor constructor (regular scala constructor) called when actor is created
- preStart called after the actor is started, after a restart it is called by default by postRestart
- postStop called after actor is stopped, can be used to log before exiting or do clean up.
- preRestart when an actor is restarted the old actor is informed of that when preRestart is called by the new actor, telling it what exception caused the restart.
- postRestart

Restart sequence of events from doc.akka.io

life-cycle methods: preStart,postStop,preRestart,posRestart

1. Suspend the actor and recursively suspends children
2. Call preRestart hook on the actor, forcing all its children to call postStop
3. preRestart will have started context.stop it is non blocking so it will continue without checking that all children have been terminated
4. Create a new actor instance and replace the failed instance by the new one inside the child ActorRef (this is what is used to communicate, it is shared via message passing.)
5. Invoke postRestart on the new instance which calls preRestart by default
6. If all children have not been terminated do step 2 again
7. Restart the children recursively
8. Resume the actor.

New Instance replacement



ActorRef: Proxy to real Actor Instance

Logging Overwrite

Logging can be handled in any life-cycle methods: preStart,postStop,preRestart,posRestart + receive

```
import akka.event.Logging
import akka.actor._

class Test extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    log.debug("Received")
  }

  override def preStart() = {
    log.debug("Starting")

  override def postStop() = {
    log.debug("Starting")
  }

    override def preRestart(reason: Throwable, message: Option[Any]) = {
      log.error(reason, "Restarting: [{}]", reason.getMessage)
    }

  }

}
```

Default Supervisor strategy

- ActorInitializationException: stop the failing child actor
- ActorKilledException: stop the failing child actor
- DeathPactException: stop the failing child actor
- Exception: restart the failing child actor
- Other Throwable: Escalate

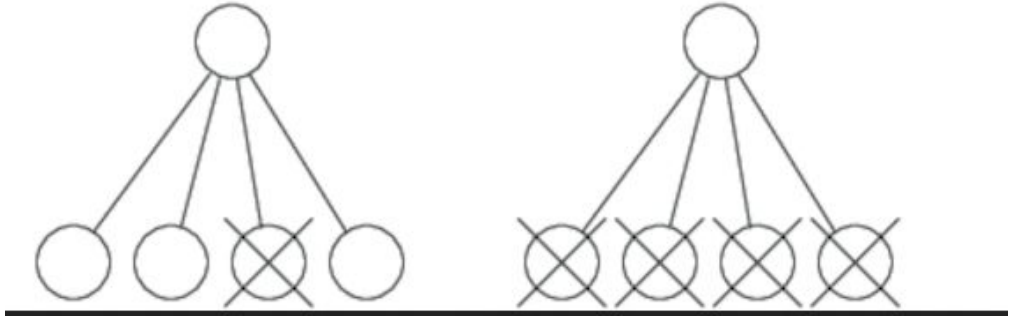
The object handling errors is called the decider.

This can be overwritten, but if it escalates to the root guardian it would be handled as default.
Error logging is done by supervisor handling the error. Logging is important to find where the issue happened.

By default log messages are printed to STDOUT you can use SLF4J logger or your own

One for one and one for all

- Two supervision strategies:



```
AllForOneStrategy(maxNrOfRetries = 5, withinTimeRange = 10 seconds) {
      case _: Exception => stop
    }
  }

  OneForOneStrategy(maxNrOfRetries = 5, withinTimeRange = 10 seconds) {
      case _: Exception => stop
    }
```

Use AFO/OFA strategies efficiently

- Trying to handle too general exceptions using one for one will make things counterproductive. Assume a file shared amongst many threads.

```
import akka.actor.OneForOneStrategy
import akka.actor.SupervisorStrategy._

override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 5, withinTimeRange = 10 seconds) {
    case _: ArithmeticException      => Resume
    case _: NullPointerException     => Restart
    case _: IllegalArgumentException => Stop
    case _: LockedFile               => Restart
    case _: RuntimeException         => Stop
    case _: Exception                => Escalate
  }
```

Supervisor Backoff

- Oftentimes you may get an exception thrown because a resource was unavailable e.g: Database being restarted, file in use by another process, not enough memory to perform an operation....
- In those case we may need to give to give our process some time, we use a backoff, we can customize the back off as we wish but conventional method is to retry the strategy after a certain time a certain number of times before either escalating or stopping the process, exponentially increasing the time to wait each time.
- This strategy is useful when a PersistentActor fails with a persistent failure.

BackOff example options

From package akka.pattern, public class BackoffOpts

Example from doc.akka, when using onStop we expect the subordinate to stop on failure, when using onFailure we expect it to restart.

```
val supervisor = BackoffSupervisor.props(
  BackoffOpts
    .onFailure(
      childProps,
      childName = "myEcho",
      minBackoff = 3.seconds,
      maxBackoff = 30.seconds,
      randomFactor = 0.2 // adds 20% "noise" to vary the intervals slightly
    )
    .withAutoReset(10.seconds) // reset if the child does not throw any errors within 10 seconds
    .withSupervisorStrategy(OneForOneStrategy() {
      case _: MyException => SupervisorStrategy.Restart
      case _              => SupervisorStrategy.Escalate
    }))
```

Importance of Supervision in Failure handling

- Supervision follows the same principle (Unix Supervisor)
- Provides security
- Increases availability
- Convenience
- Efficiency
- Simplicity
- Centralization or Decentralization, one place to start stop and monitor, can use groups, pools or monitor individually.

Questions