AKKA & MONIX

CONTROLLING POWER PLANTS

Alexandru Nedelcu

Software Developer @ eloquentix.com
@alexelcu / alexn.org

MY WORK AT E.ON

- System for monitoring & controlling power plants
- Architecture based on **micro-services**
- Processing of signals in **soft real-time**
- State machines responding to command or state signals
- SLAs, **resilience** built-in
- Much like an orchestra playing a symphony

REAL-TIME

- *"controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time"*
- Input is received continuously
- Implies **Asynchrony** (on the JVM)

ASYNCHRONY

- *"the occurrence of events independently of the main program flow and ways to deal with such events"*
- Implies **Nondeterminism**
- Implies **Concurrency**

ASYNCHRONY

Not something you can fix later!

AKKA ACTORS

THE GOOD PARTS

- Standard solution
- Encapsulation
- Concurrency guarantees
- Message-passing over address spaces
- Supervision

THE BAD PARTS

- No best practices
- Too flexible (e.g. Any => Unit)
- Bidirectional comms => cycles
- Concurrency problems in communication
- Stateful & Async
  (worse than the worst of OOP)

THE BAD PARTS

All problems of Actors are problems of micro-services!

ANTI-PATTERN: INTERNAL MESSAGES

```
class SomeActor extends Actor {

  private val scheduler = context.system.scheduler
    .schedule(3.seconds, 3.seconds, self, Tick)

  def receive = {
    case Tick => doSomething()
  }
}
```

ANTI-PATTERN: CAPTURING INTERNALS

```
class SomeActor extends Actor {

  private val readings =
    mutable.ListBuffer.empty[Double]

  def receive = {
    case Tick =>
      for (r <- fetchReading()) {
        // Oops, multi-threading issues!
        readings += r
      }
  }
}
```

ASYNCHRONOUS BLOCKING

```
class SomeActor extends Actor {
  def fetchReading(): Future[Double] = ???
  private val readings = ListBuffer.empty[Double]

  def receive = {
    case Tick =>
      fetchReading pipeTo self
      context.become(waitForReading)
  }

  def waitForReading: Receive = {
    case Tick => () // ignore
    case reading: Double =>
      readings += reading
      context.become(receive)
  }
}
```

EVOLUTIONS & CONTEXT.BECOME

```
class SomeActor extends Actor {
  def fetchReading(): Future[Double] = ???
  def receive = active(Queue.empty[Double])

  def active(readings: Queue[Double]): Receive = {
    case Tick =>
      fetchReading() pipeTo self
      context become waitForReading(readings)
  }

  def waitForReading(readings: Queue[Double]): Receive = {
    case Tick => () // ignore
    case r: Double =>
      context become active(readings.enqueue(r))
  }
}
```

NO I/O LOGIC IN ACTORS

```scala
case class Update(r: Double)

class SomeActor extends Actor {
  def receive = active(Queue.empty)

  def active(readings: Queue[Double]): Receive = {
    case Update(r) =>
      context become active(readings.enqueue(r))
  }
}
```

EXPLICIT TIME 1/2

- All input must be explicit
- Including input provided by *The World*
- Or else you introduce **Nondeterminism**
- No **DateTime.now**

EXPLICIT TIME 2/2

```scala
case class Update(r: Double, now: DateTime)

class SomeActor extends Actor {
  def receive = {
    case Update(r,now) =>
      context become active(empty, now)
  }

  def active(state: Queue[Double], ts: DateTime): Receive = {
    case Update(r, now) =>
      val next = active(state enqueue r, now)
      context become next
  }
}
```

PURELY FUNCTIONAL STATE (1/5)

```scala
case class Update(reading: Double)

case class StateMachine(readings: Queue[Double]) {
  def evolve(r: Double): StateMachine =
    copy(readings.enqueue(r))
}

class StateMachineActor extends Actor {
  def receive = active(StateMachine(Queue.empty))

  def active(state: StateMachine): Receive = {
    case Update(r) =>
      context become active(state update r)
  }
}
```

PURELY FUNCTIONAL STATE (2/5)

```
type Evolve[S,U] =
  (S,U) => S

type Produce[S,A] =
  S => (A,S)
```

PURELY FUNCTIONAL STATE (3/5)

```scala
sealed trait Output

case class StatusUpdate
  (assetId: Long, powerOutput: Double)
  extends Output

case class Transition
  (assetId: Long, oldState: State, newState: State)
  extends Output

case class Dispatch(assetId: Long, value: Double)
  extends Output

case class Alert(assetId: Long, error: String)
  extends Output
```

PURELY FUNCTIONAL STATE (4/5)

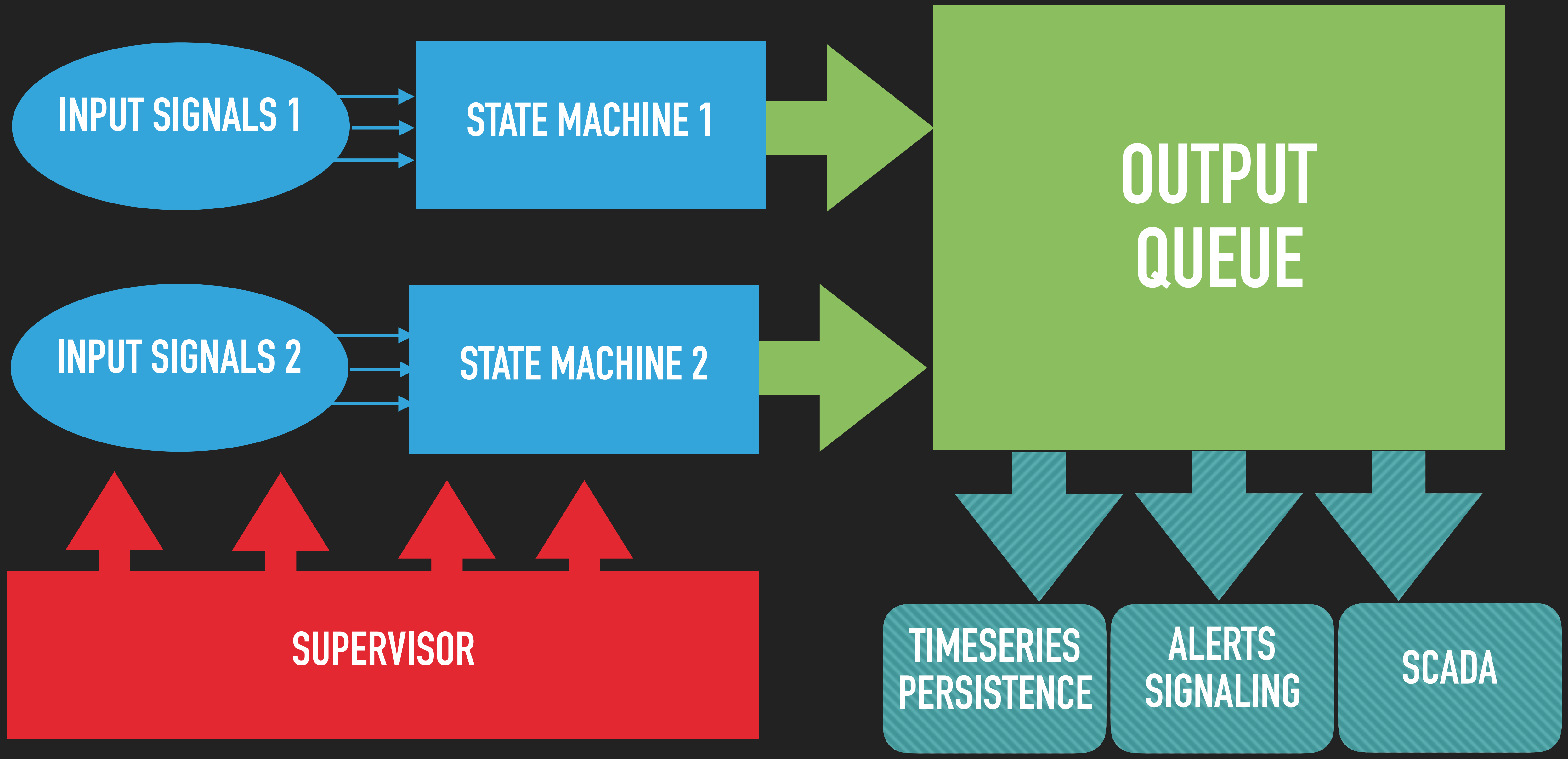
```scala
case class StateMachine(
  state: State,
  output: Queue[Output]) {

  def evolve(input: Input): StateMachine = ???

  def produce: (Seq[Output], StateMachine) =
    (output, copy(output=Queue.empty))
}
```

PURELY FUNCTIONAL STATE (5/5)

```
class StateMachineActor(channel: SyncObserver[Output])
  extends Actor {

  def receive = active(StateMachine.initial)

  def active(fsm: StateMachine): Receive = {
    case input: Input =>
      context become active(fsm.evolve(input))

    case Produce =>
      val (output, next) = fsm.produce
      for (event <- output) channel.onNext(event)
      context become active(next)
  }
}
```

ARCHITECTURE

HIGH LEVEL OVERVIEW



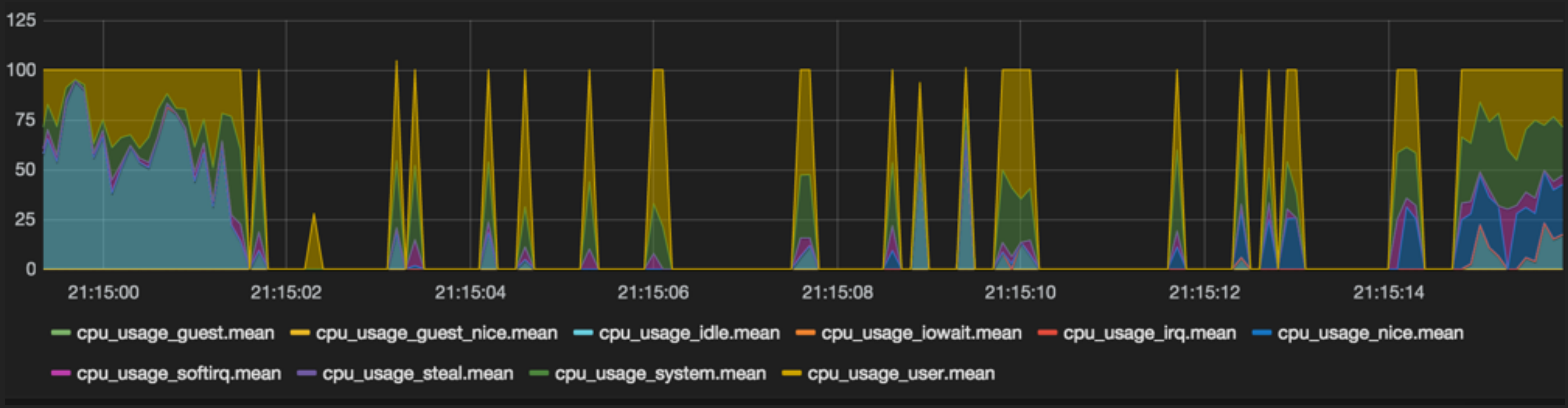
DECOUPLING PHILOSOPHY

- Mocks & Stubs => tight coupling
- DI techniques are for hiding junk (Guice, Cake Pattern, etc.)
- Pain Driven Development: Don't hide the junk, pain is good :-)

BACK-PRESSURE (1/3)

- Q: What if the Producer is too fast?
- Q: What if Networking goes down?
- Q: What if you’re left without CPU?
- Problem: Any unlimited queue can blow up!

BEST PRACTICE: BACK-PRESSURE (2/3)



In a distributed system, shit happens ;-)

BEST PRACTICE: BACK-PRESSURE (3/3)

- A: Pause the producer
- A: Or have an overflow strategy
- 1. E.g. drop messages on the floor
- 2. Be redundant

MONIX

WHAT IS MONIX?

- Scala / Scala.js library
- For composing asynchronous programs
- Exposes Observable & Task
- Modular design
- Typelevel Incubator
- 2.0-RC2
- See: monix.io

MONIX SUB-PROJECTS

- Minitest: Scala/Scala.js testing
- Sincron: Atomic references
- **monix-execution:** Scheduler, Cancelable
- **monix-eval:** Task, Coeval
- **monix-reactive:** Observable
- **monix-cats, monix-scalaz:** *Work in progress!*

MONIX OBSERVABLE

A CONSTRAINT AT ONE LEVEL GIVES US FREEDOM AND POWER AT A HIGHER LEVEL.

Rúnar Bjarnason

OBSERVABLE

- Unidirectional streaming of events
- Asynchronous
- One producer => one/many consumers
- Handles back-pressure
- Composable

OBSERVABLE

	Single	Multiple
Synchronous	() => A	Iterable[A]
Asynchronous	Future[A] / Task[A]	Observable[A]

OBSERVABLE: SAMPLE

```
Observable.fromIterable(0 until 1000)
  .filter(_ % 2 == 0)
  .map(_ * 2)
  .flatMap(x => Observable.fromIterable(Seq(x,x)))
```

OBSERVABLE: SUBSCRIBE (1/2)

```scala
import monix.execution.Scheduler
import Scheduler.Implicits.global

val cancelable = observable.subscribe
```

OBSERVABLE: SUBSCRIBE (2/2)

```scala
val cancelable = observable.subscribe(
  new Observer[Int] {
    def onNext(elem: Int): Future[Ack] = {
      println(elem)
      Continue
    }

    def onComplete(): Unit = ()
    def onError(ex: Throwable): Unit =
      global.reportFailure(ex)
  })
```

OBSERVABLE: BUILDERS (1/4)

```scala
Observable.now("Hello!")

Observable.evalAlways { println("effect"); "Hello!" }

Observable.evalOnce { println("effect"); "Hello!" }

Observable.defer(Observable.now("Hello!"))

Observable.fork(Observable.evalAlways { "Hello!" })

Observable.fromFuture(future)
```

OBSERVABLE: BUILDERS (2/4)

```
Observable.fromIterable(0 to 1000)

Observable.fromIterator((0 to 1000).iterator)

Observable.fromReactivePublisher(publisher)

Observable.fromStateAction(pseudoRandom)(1023)

Observable.repeatEval(Random.nextInt())
```

OBSERVABLE: BUILDERS (3/4)

```
Observable.repeat(1,2,3)

Observable.interval(1.second)

Observable.intervalAtFixedRate(1.second)

Observable.intervalWithFixedDelay(1.second)
```

OBSERVABLE: BUILDERS (4/4)

```
// Safe builder for cold observable
Observable.create[Int](Unbounded) { subscriber =>
  subscriber.onNext(1)
  subscriber.onNext(2)
  subscriber.onNext(3)
  subscriber.onComplete()

  Cancelable.empty
}
```

HOT OBSERVABLES

HOT OBSERVABLE 1/2

```scala
val subject = ConcurrentSubject
  .publish[Int](Unbounded)

subject.dump("O").subscribe()
subject.onNext(1)
subject.onNext(2)
```

HOT OBSERVABLE 2/2

```
val coldObservable =  Observable.interval(1.second)
val connectable = coldObservable.publish

val s1 = connectable.dump("S1").subscribe()
val s2 = connectable.dump("S2").subscribe()
val s3 = connectable.dump("S3").subscribe()

val cancelable = connectable.connect()
```

POLLING

```
Observable.interval(1.second).concatMap { _ =>
  Observable.fromFuture(WS.url("http://some-url.com").get)
}
```

POLLING

```
val request = Task.defer {
 Task.fromFuture(WS.url(“..."").get)
   .delayExecution(1.second)
}

Observable.repeat(0).flatMap { _ =>
  Observable.fromTask(request)
}
```

SCAN: FILTERING DATA 1/2

```scala
case class SimpleMovingAverage(
  points: Queue[Double],
  windowSize: Int) {

  lazy val value: Double =
    if (points.isEmpty) 0.0
    else points.sum / points.length

  def evolve(point: Double) = {
    val newQueue = points.enqueue(points)
    copy(newQueue.takeRight(windowSize))
  }
}
```

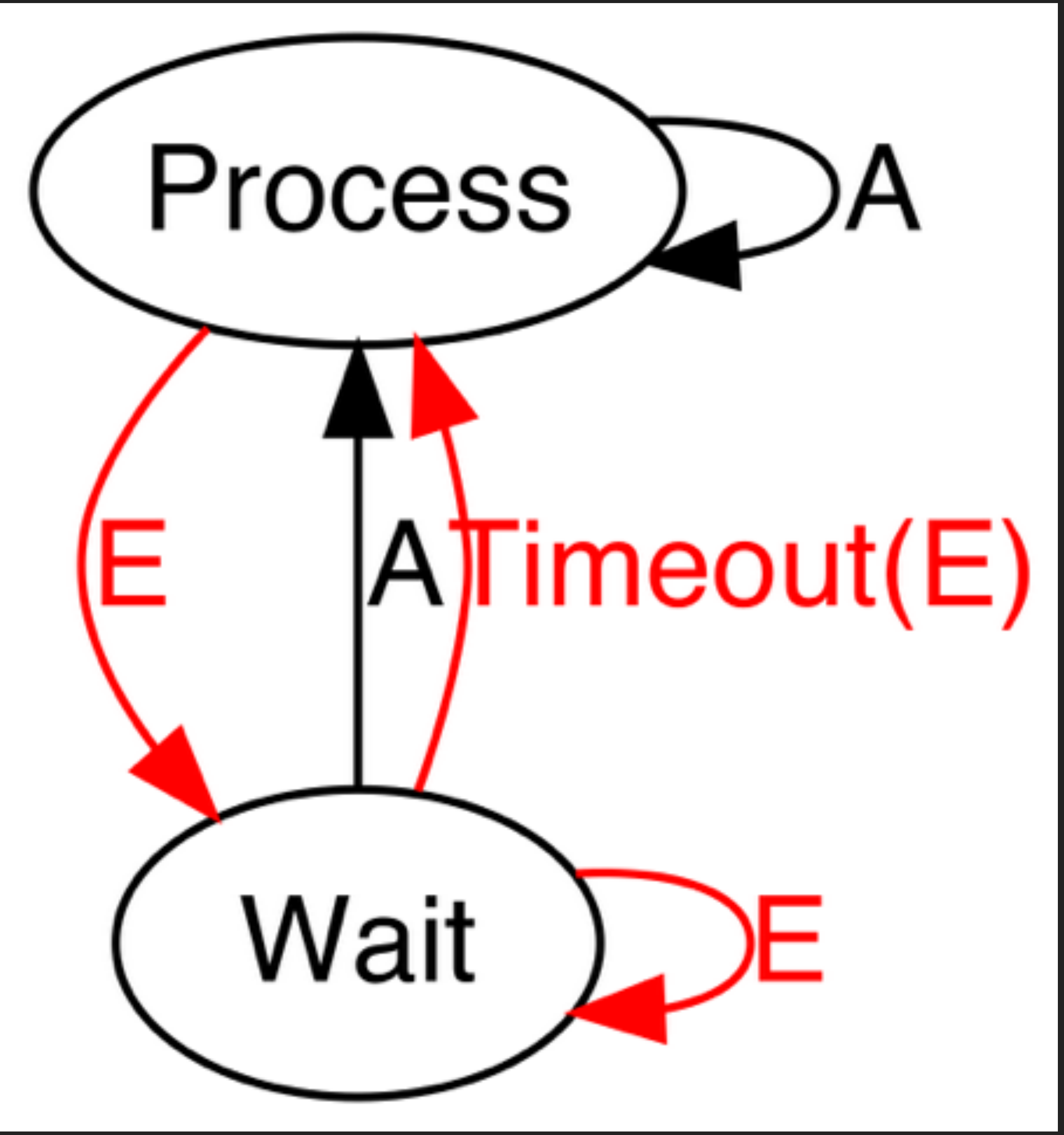
SCAN: FILTERING DATA 2/2

```scala
val source: Observable[Double] = ???

val init = SimpleMovingAverage(
  Queue.empty, windowLength = 10)

val scanned: Observable[SimpleMovingAverage] =
  source.scan(init)((state, e) => state.evolve(e))

val mapped: Observable[Double] =
  scanned.map(_.value)
```

STATE MACHINES WOOT!

TIMEOUT GATE (1/4)

- Purpose is to filter out sporadic errors
- On error, transitions to **Wait** but with a timeout



TIMEOUT GATE (2/4)

```scala
sealed trait State[+T]

case object Init extends State[Nothing]

case class Wait[+T](value: T, expiresAtTS: Long)
  extends State[T]

case class Process[+T](value: T)
  extends State[T]
```

TIMEOUT GATE (3/4)

```scala
case class TimeoutGate[E,A]
  (timeout: FiniteDuration, timestamp: Either[E,A] => Long) {

  type S = State[Either[E,A]]
  def init: S = Init

  def evolve(acc: S, elem: Either[E,A]): S = ???
}
```

TIMEOUT GATE (4/4)

```scala
val gate = TimeoutGate[E,A](1.minute, ???)

observable.scan(gate.init)(gate.evolve)
  .collect { case Process(signal) => signal }
```

THROTTLING

THROTTLING

```scala
source
  .distinctUntilChanged
  .throttleLast(1.second)
  .echoRepeated(5.seconds)
```

THROTTLING

```
source.groupBy(_.assetId).mergeMap { gr =>
  gr.distinctUntilChanged
    .throttleLast(1.second)
    .echoRepeated(5.seconds)
}
```

THROTTLING

```scala
source.debounce(4.seconds)

source.switchMap { x =>
  Observable.now(x).delaySubscription(4.seconds)
}

source.debounceRepeated(4.seconds)

source.switchMap { x =>
  Observable.intervalAtFixedRate(4.seconds)
    .map(_ => x)
}
```

BACK-PRESSURE

BACK-PRESSURE

```
source.whileBusyDropEvents
```

BACK-PRESSURE

```scala
source.whileBusyBuffer(
  OverflowStrategy.DropOld(bufferSize = 1024))
```

BACK-PRESSURE

```scala
val source: Observable[A] = ???

val buffered: Observable[List[A]] =
  source.bufferIntrospective(maxSize = 1024)
```

WHAT'S THE POINT?

MONIX OBSERVABLE

- Decoupling (Observer pattern, ftw)
- Handles the non-determinism
- Back-pressure provided for free
- Can be used in combination with Actors, Task, Future, whatever…

CATS INTEGRATION CHALLENGES

- Monad, MonadError, MonadFilter, MonadCombine, CoflatMap, Applicative
- Foldable, Traverse: not implementable (need async versions, foldRight not possible)
- Missing, potentially useful type-classes, e.g. Zippable, Scannable, Evaluable

MONIX.IO

QUESTIONS?