ZIO ACTORS

GITHUB.COM/MTSOKOL

TWITTER: @MT_SOKOL

AGENDA

AGENDA

> ACTORS OUTLINE

AGENDA

> ACTORS OUTLINE

> AKKA RECALL

AGENDA

> ACTORS OUTLINE

> AKKA RECALL

> ZIO ACTORS BY EXAMPLE

AGENDA

- ACTORS OUTLINE
- AKKA RECALL
- ZIO ACTORS BY EXAMPLE
- PROBLEMS FACED

ACTORS

A Universal Modular ACTOR Formalism for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired in a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, or semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

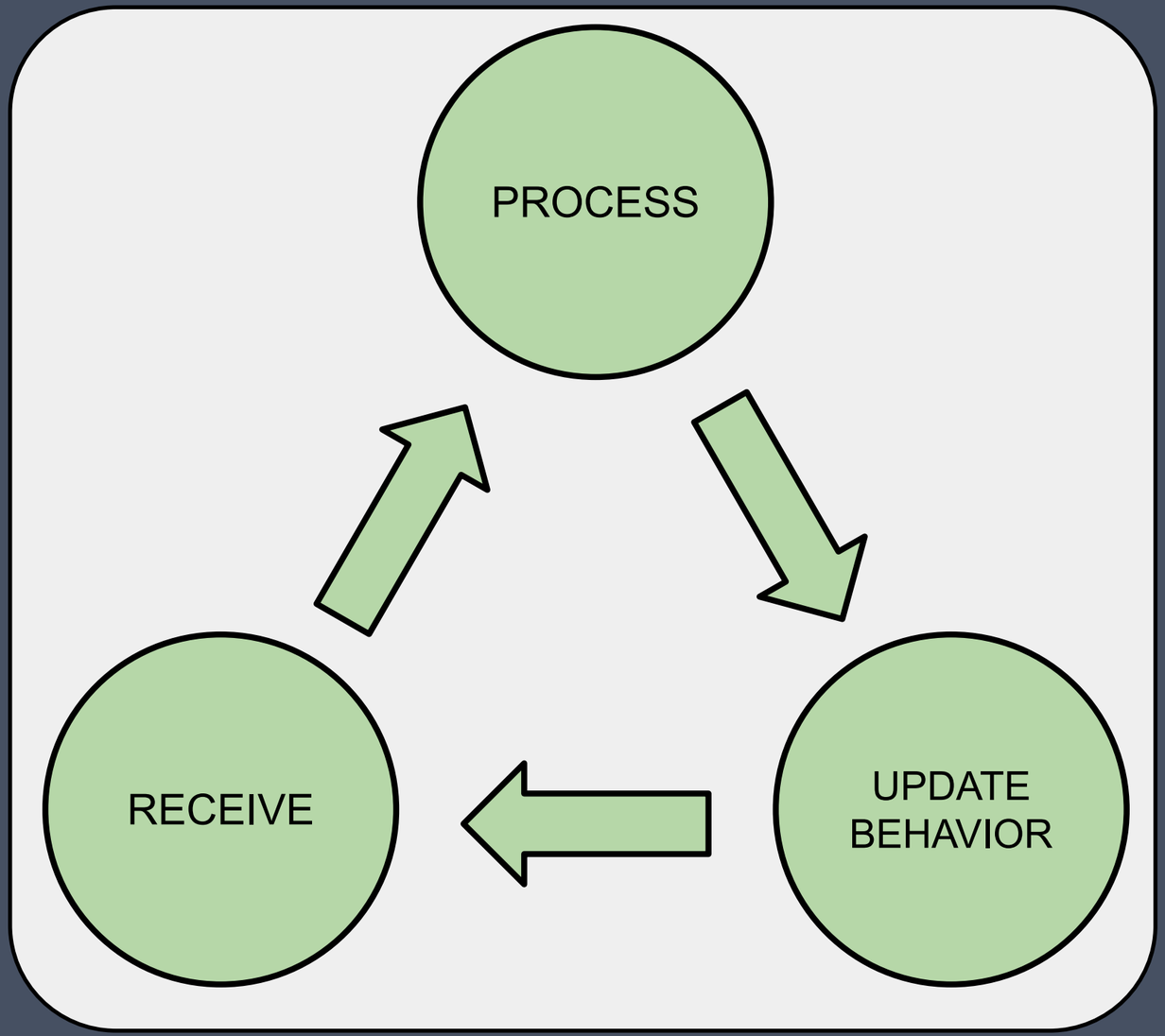
"Programs should not only work,
but they should appear to work as well."
PDP-1X Dogma

The PLANNER project is continuing research in natural and effective means for embedding knowledge in procedures. In the course of this work we have succeeded in unifying the formalism around one fundamental concept: the ACTOR. Intuitively, an ACTOR is an active agent which plays a role on cue according to a script. We use the ACTOR metaphor to emphasize the inseparability of control and data flow in our model. Data structures, functions, semaphores, monitors, ports, descriptions, Quillian nets, logical formulae, numbers, identifiers, demons, processes, contexts, and data bases can all be shown to be special cases of actors. All of the above are objects with certain useful modes of behavior. Our formalism shows how all of the modes of behavior can be defined in terms of one kind of behavior: sending messages to actors. An actor is always invoked uniformly in exactly the same way regardless of whether it behaves as a recursive function, data structure, or process.

"It is vain to multiply Entities beyond need."
William of Occam

"Monotheism is the Answer."

The unification and simplification of the formalisms for the procedural embedding of


PROCESS
UPDATE
BEHAVIOR
RECEIVE

ACTORS SHOULD

ACTORS SHOULD

- HAVE A DEFINED MESSAGES TYPE

ACTORS SHOULD

- HAVE A DEFINED MESSAGES TYPE
- FORM SUPERVISION TREES

ACTORS SHOULD

- HAVE A DEFINED MESSAGES TYPE
- FORM SUPERVISION TREES
- BE LOCATION-TRANSPARENT

AKKA ACTORS

```
class ClassicActor extends Actor {
  def receive = {
    case "Hello" => println("Hello!")
    case _       => println("Hello anyway.")
  }
}

object TypedActor {
  def act(): Behavior[Msg] = Behaviors.receive {
    (ctx, message) =>
      //computations...
      Behaviors.same
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[ClassicActor], name = "actor1")
  helloActor ! "Hello"
  helloActor ! "Hi"
}
```

```
class ClassicActor extends Actor {
  def receive = {
    case "Hello" => println("Hello!")
    case _       => println("Hello anyway.")
  }
}

object TypedActor {
  def act(): Behavior[Msg] = Behaviors.receive {
    (ctx, message) =>
      //computations...
      Behaviors.same
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[ClassicActor], name = "actor1")
  helloActor ! "Hello"
  helloActor ! "Hi"
}
```

```scala
class ClassicActor extends Actor {
  def receive = {
    case "Hello" => println("Hello!")
    case _       => println("Hello anyway.")
  }
}

object TypedActor {
  def act(): Behavior[Msg] = Behaviors.receive {
    (ctx, message) =>
      //computations...
      Behaviors.same
  }
}

object Main extends App {
  val system = ActorSystem("HelloSystem")
  val helloActor = system.actorOf(Props[ClassicActor], name = "actor1")
  helloActor ! "Hello"
  helloActor ! "Hi"
}
```

ZIO ACTORS

ZIO[R, E, A]

ALIASES

UIO[A]	=>	ZIO[ANY, NOTHING, A]
URIO[R, A]	=>	ZIO[R, NOTHING, A]
TASK[A]	=>	ZIO[ANY, THROWABLE, A]
IO[E, A]	=>	ZIO[ANY, E, A]

```scala
object ActorSystem {
  def apply(name: String, configFile: Option[File]): Task[ActorSystem]
}

final class ActorSystem() {
  def make[R, S, F[+_]](
    actorName: String,
    sup: Supervisor[R],
    init: S,
    stateful: Stateful[R, S, F]
  ): RIO[R, ActorRef[E, F]]

  def select[F[+_]](path: String): Task[ActorRef[F]]

  def shutdown: Task[List[_]]
}
```

```scala
object ActorSystem {
  def apply(name: String, configFile: Option[File]): Task[ActorSystem]
}

final class ActorSystem() {
  def make[R, S, F[+_]](
    actorName: String,
    sup: Supervisor[R],
    init: S,
    stateful: Stateful[R, S, F]
  ): RIO[R, ActorRef[E, F]]

  def select[F[+_]](path: String): Task[ActorRef[F]]

  def shutdown: Task[List[_]]
}
```

```scala
trait Stateful[-R, S, -F[+_]] {

  def receive[A](
    state: S,
    msg: F[A],
    context: Context
  ): RIO[R, (S, A)]

}
```

```
sealed trait ActorRef[-F[+_]] {

  def ?[A](fa: F[A]): Task[A]

  def !(fa: F[_]): Task[Unit]

  val path: UIO[String]

}
```

TYPED

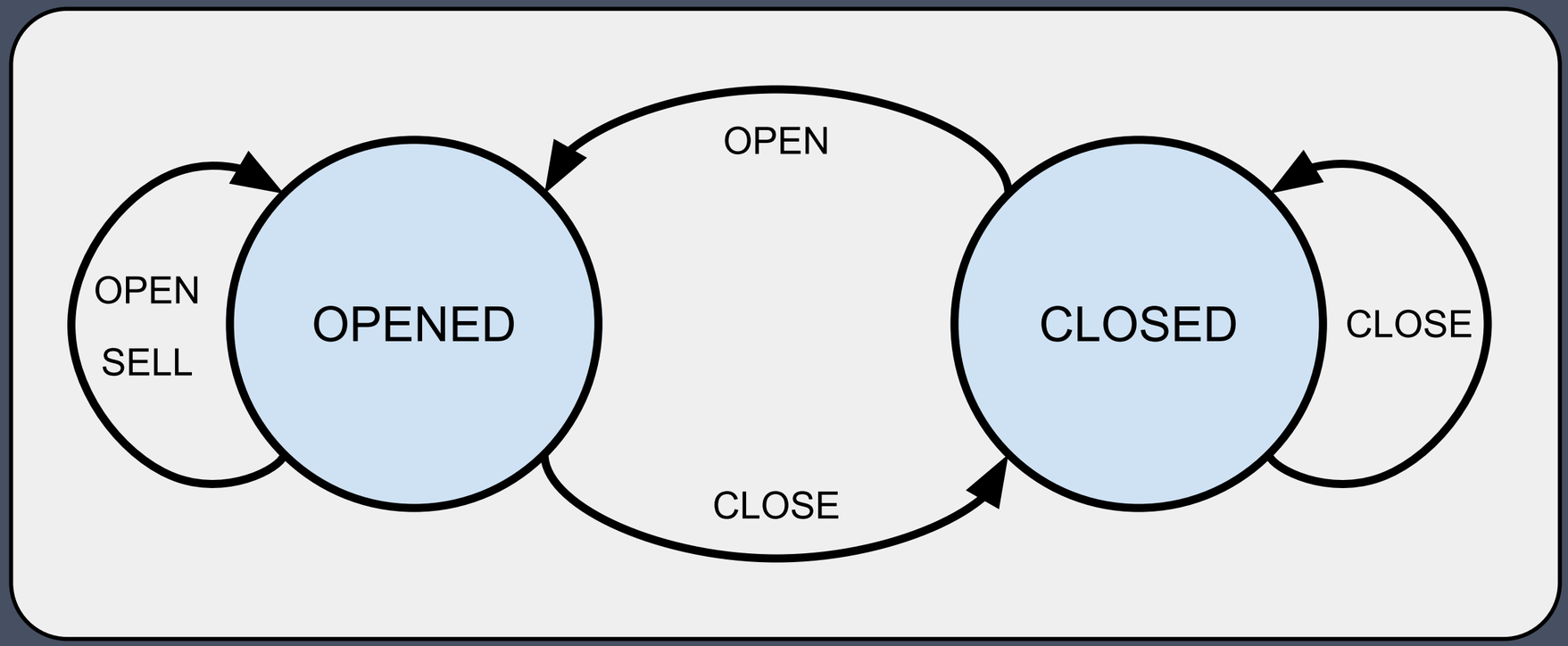
```
sealed trait Message[+_]
case object Reset    extends Message[Unit]
case object Increase extends Message[Unit]
case object Get      extends Message[Int]

val handler =
  new Stateful[Any, Int, Message] {
    override def receive[A](state: Int, msg: Message[A], context: Context) =
                msg match {
                  case Reset    => UIO((0, ()))
                  case Increase => UIO((state + 1, ()))
                  case Get      => UIO((state, state))
    }
  }
```

```
sealed trait Message[+_]
case object Reset    extends Message[Unit]
case object Increase extends Message[Unit]
case object Get      extends Message[Int]

val handler =
  new Stateful[Any, Int, Message] {
    override def receive[A](state: Int, msg: Message[A], context: Context) =
                msg match {
                  case Reset    => UIO((0, ()))
                  case Increase => UIO((state + 1, ()))
                  case Get      => UIO((state, state))
    }
  }
```

```
for {
  system <- ActorSystem("test1")
  actor  <- system.make("actor1", Supervisor.none, 0, handler)
  _      <- actor ! Increase
  _      <- actor ! Increase
  c1     <- actor ? Get
  _      <- actor ! Reset
  c2     <- actor ? Get
} yield assert(c1, equalTo(2)) && assert(c2, equalTo(0))
```

```
for {
  system <- ActorSystem("test1")
  actor  <- system.make("actor1", Supervisor.none, 0, handler)
  _      <- actor ! Increase
  _      <- actor ! Increase
  c1     <- actor ? Get
  _      <- actor ! Reset
  c2     <- actor ? Get
} yield assert(c1, equalTo(2)) && assert(c2, equalTo(0))
```

FSM


OPEN
OPEN
SELL
OPENED
CLOSED
CLOSE
CLOSE

```
sealed trait Command[+A]

case object Open  extends Command[Unit]
case object Close extends Command[Unit]
case object Sell  extends Command[String]


sealed trait State

case object Opened extends State
case object Closed extends State


case class SaleException(msg: String) extends Throwable
```

```
sealed trait Command[+A]

case object Open  extends Command[Unit]
case object Close extends Command[Unit]
case object Sell  extends Command[String]


sealed trait State

case object Opened extends State
case object Closed extends State


case class SaleException(msg: String) extends Throwable
```

```
new Stateful[Console, State, Command] {
  override def receive[A](
    state: State,
    msg: Command[A],
    context: Context
  ): ZIO[Console, SaleException, (State, A)] = /*...*/
}
```

```
state match {
  case Opened =>
    msg match {
      case Open  => UIO((Opened, ()))
      case Close => putStrLn("Closing...") *> UIO((Closed, ()))
      case Sell  => UIO((Opened, "Selling..."))
    }
  case Closed =>
    msg match {
      case Open  => putStrLn("Opening...") *> UIO((Opened, ()))
      case Close => UIO((Closed, ()))
      case Sell  => IO.fail(SaleException("Shop is closed"))
    }
}
```

SUPERVISION

```
for {
  system   <- ActorSystem("sys1")
  schedule =  Schedule.recurs(10)
  policy   =  Supervisor.retry(schedule)
  actor    <- system.make("actor1", policy, (), handler)
  _        <- actor ! MightFail
} yield ()

val s1 = Schedule.exponential(10.milliseconds)
val s2 = Schedule.exponential(1.second) && Schedule.recurs(10)
```

```
for {
  system   <- ActorSystem("sys1")
  schedule =  Schedule.recurs(10)
  policy   =  Supervisor.retry(schedule)
  actor    <- system.make("actor1", policy, (), handler)
  _        <- actor ! MightFail
} yield ()

val s1 = Schedule.exponential(10.milliseconds)
val s2 = Schedule.exponential(1.second) && Schedule.recurs(10)
```

REMOTING SUPPORT

CONFIGURATION

MADE WITH ZIO-CONFIG 🎉

```
# application.conf

testSystem1.zio.actors.remoting {
  hostname = ${HOST}
  port     = ${PORT}
}
```

```
for {
  sys1 <- ActorSystem("testSystem1")
  _    <- sys1.make("actorOne", Supervisor.none, 0, handler)
} yield ()

//============ OTHER NODE ==============

for {
  sys2   <- ActorSystem("testSystem2")
  actor  <- sys2.select[MyErrorDomain, Message](
    s"zio://testSystem1@127.0.0.1:$port1/actorOne"
  )
  result <- actor ? Str("ZIO-Actor response... ")
} yield assert(result, equalTo("ZIO-Actor... received 1"))
```

```
for {
  sys1 <- ActorSystem("testSystem1")
  _    <- sys1.make("actorOne", Supervisor.none, 0, handler)
} yield ()

//============= OTHER NODE ==============

for {
  sys2   <- ActorSystem("testSystem2")
  actor  <- sys2.select[MyErrorDomain, Message](
    s"zio://testSystem1@127.0.0.1:$port1/actorOne"
  )
  result <- actor ? Str("ZIO-Actor response... ")
} yield assert(result, equalTo("ZIO-Actor... received 1"))
```

PERSISTENCE

```
class EventSourcedStateful[R, S, -F[+_], Ev](id: PersId) {

  def receive[A](
    state: S,
    msg: F[A],
    context: Context
  ): RIO[R, (Command[Ev], S => A)]

  def sourceEvent(state: S, event: Ev): S

}
```

```
object CounterUtils {
  sealed trait Message[+_]
  case object Reset    extends Message[Unit]
  case object Increase extends Message[String]
  case object Get      extends Message[Int]

  sealed trait CounterEvent
  case object ResetEvent    extends CounterEvent
  case object IncreaseEvent extends CounterEvent
}
```

```
new EventSourcedStateful[Any, Int, Message, CounterEvent](id) {
  override def receive[A](
    state: Int,
    msg: Message[A],
    context: Context
  ): UIO[(Command[CounterEvent], Int => A)] =
    msg match {
      case Reset    => UIO((Comm.persist(ResetEvent), _ => ()))
      case Increase => UIO((Comm.persist(IncreaseEvent), s => s"updated to $s"))
      case Get      => UIO((Comm.ignore, _ => state))
    }

  override def sourceEvent(state: Int, event: CounterEvent): Int =
    event match {
      case ResetEvent    => 0
      case IncreaseEvent => state + 1
    }
}
```

```
new EventSourcedStateful[Any, Int, Message, CounterEvent](id) {
  override def receive[A](
    state: Int,
    msg: Message[A],
    context: Context
  ): UIO[(Command[CounterEvent], Int => A)] =
    msg match {
      case Reset    => UIO((Comm.persist(ResetEvent), _ => ()))
      case Increase => UIO((Comm.persist(IncreaseEvent), s => s"updated to $s"))
      case Get      => UIO((Comm.ignore, _ => state))
    }

  override def sourceEvent(state: Int, event: CounterEvent): Int =
    event match {
      case ResetEvent    => 0
      case IncreaseEvent => state + 1
    }
}
```

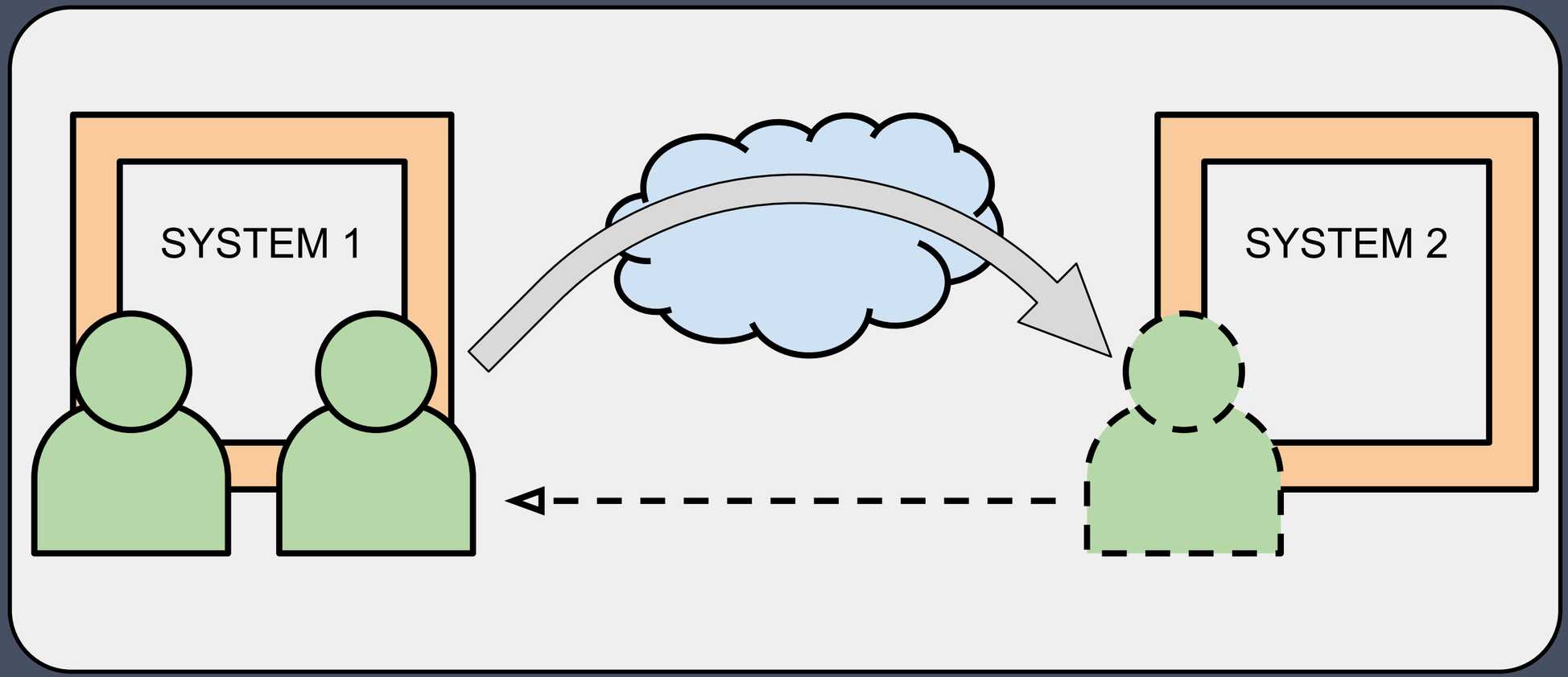
```
new EventSourcedStateful[Any, Int, Message, CounterEvent](id) {
  override def receive[A](
    state: Int,
    msg: Message[A],
    context: Context
  ): UIO[(Command[CounterEvent], Int => A)] =
    msg match {
      case Reset    => UIO((Comm.persist(ResetEvent), _ => ()))
      case Increase => UIO((Comm.persist(IncreaseEvent), s => s"updated to $s"))
      case Get      => UIO((Comm.ignore, _ => state))
    }

  override def sourceEvent(state: Int, event: CounterEvent): Int =
    event match {
      case ResetEvent    => 0
      case IncreaseEvent => state + 1
    }
}
```

PEEK INTO INTERNALS

PROBLEMS FACED

```
29  private[akka] class RepointableActorRef(
30      val system: ActorSystemImpl,
31      val props: Props,
32      val dispatcher: MessageDispatcher,
33      val mailboxType: MailboxType,
34      val supervisor: InternalActorRef,
35      val path: ActorPath)
36      extends ActorRefWithCell
37      with RepointableRef {
38
39    import AbstractActorRef.{ cellOffset, lookupOffset }
40
41    /*
42     * H E R E   B E   D R A G O N S !
43     *
44     * There are two main functions of a Cell: message queueing and child lookup.
45     * When switching out the UnstartedCell for its real replacement, the former
46     * must be switched after all messages have been drained from the temporary
47     * queue into the real mailbox, while the latter must be switched before
48     * processing the very first message (i.e. before Cell.start()). Hence there
49     * are two refs here, one for each function, and they are switched just so.
50     */
51    @silent @volatile private var _cellDoNotCallMeDirectly: Cell = _
52    @silent @volatile private var _lookupDoNotCallMeDirectly: Cell = _
53
54    def underlying: Cell = Unsafe.instance.getObjectVolatile(this, cellOffset).asInst
55    def lookup = Unsafe.instance.getObjectVolatile(this, lookupOffset).asInstanceOf[
```

SERIALIZATION


SYSTEM 1
SYSTEM 2

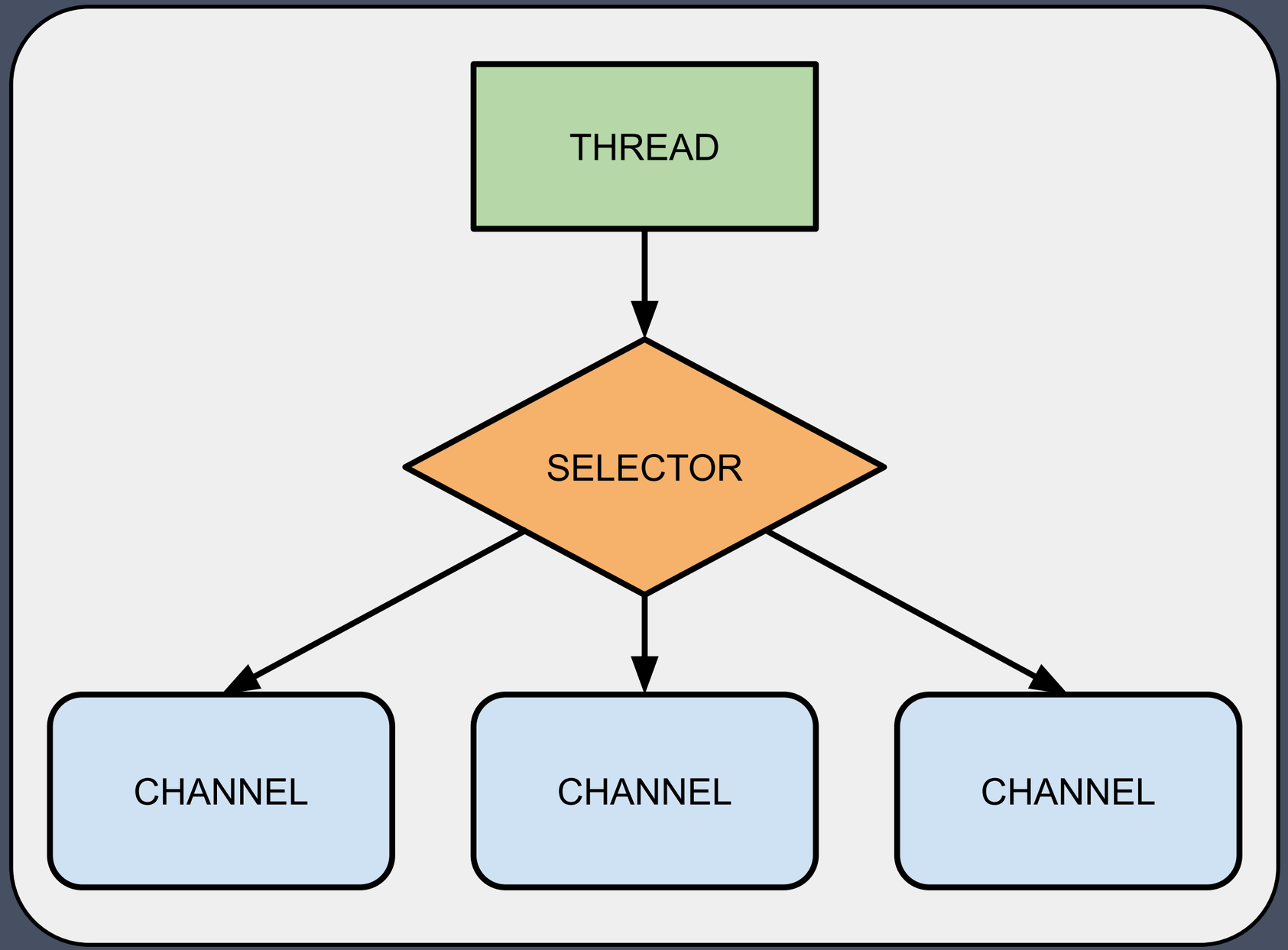
```
@throws[IOException]
protected def writeObject1(out: ObjectOutputStream): Unit =
  out.writeObject(actorPath)

@throws[ObjectStreamException]
protected def readResolve1(): Object = {
  val remoteRef = for {
    resolved           <- resolvePath(actorPath)
    (_, addr, port, _) = resolved
    address <- InetAddress.byName(addr)
                .flatMap(iAddr =>
                  SocketAddress.inetSocketAddress(iAddr, port)
                )
  } yield ActorRefRemote[F](actorPath, address)

  ActorRefSerial.runtimeForResolve.unsafeRun(remoteRef)
}
```

```
@throws[IOException]
protected def writeObject1(out: ObjectOutputStream): Unit =
  out.writeObject(actorPath)

@throws[ObjectStreamException]
protected def readResolve1(): Object = {
  val remoteRef = for {
    resolved           <- resolvePath(actorPath)
    (_, addr, port, _) = resolved
    address <- InetAddress.byName(addr)
                .flatMap(iAddr =>
                  SocketAddress.inetSocketAddress(iAddr, port)
                )
  } yield ActorRefRemote[F](actorPath, address)

  ActorRefSerial.runtimeForResolve.unsafeRun(remoteRef)
}
```

Java serialization

Java serialization is known to be slow and prone to attacks of various kinds - it ne
after all. One may think that network bandwidth and latency limit the performance
typical bottleneck.

SOCKET HANDLING


THREAD
SELECTOR
CHANNEL
CHANNEL
CHANNEL

ZIO NIO

GOALS FOR 2020

GOALS FOR 2020

- PERSISTENCE (WIP)

GOALS FOR 2020

- PERSISTENCE (WIP)
  - SERIALIZATION

GOALS FOR 2020

- PERSISTENCE (WIP)
- SERIALIZATION
- MONITORING

GOALS FOR 2020

> PERSISTENCE (WIP)

> SERIALIZATION

> MONITORING

> CLUSTERING (ZIO-KEEPER 🎉)

THANK YOU!

GITHUB.COM/MTSOKOL

TWITTER: @MT_SOKOL