ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Actors are Distributed

Programming Reactive Systems

Roland Kuhn

The Impact of Network Communication

Compared to in-process communication:

- data sharing only by value
- lower bandwidth
- higher latency
- partial failure
- data corruption

Multiple processes on the same machine are quantitatively less impacted, but qualitatively the issues are the same.

The Impact of Network Communication

Compared to in-process communication:

- data sharing only by value
- lower bandwidth
- higher latency
- partial failure
- data corruption

Multiple processes on the same machine are quantitatively less impacted, but qualitatively the issues are the same.

Distributed computing breaks assumptions made by the synchronous programming model.

Actors are Distributed

Actor communication is asynchronous, one-way and not guaranteed.

Actor encapsulation makes them look the same, regardless where they live.

Actors are “Location Transparent”, hidden behind `ActorRef`.

Actor Paths

Every actor system has an Address, forming scheme and authority of a hierarchical URI.

Actor names form the URI's path elements:

```
val system = ActorSystem(”HelloWorld”)
val ref = system.actorOf(Props[Greeter], ”greeter”)
println(ref.path)
// prints: akka://HelloWorld/user/greeter
```

Actor Paths

Every actor system has an Address, forming scheme and authority of a hierarchical URI.

Actor names form the URI's path elements:

```scala
val system = ActorSystem(”HelloWorld”)
val ref = system.actorOf(Props[Greeter], ”greeter”)
println(ref.path)
// prints: akka://HelloWorld/user/greeter
```

Remote address example: `akka.tcp://HelloWorld@10.2.4.6:6565`

Every actor is identified by at least one URI.

The Difference between ActorRef and ActorPath

Actor names are unique within a parent, but can be reused.

- ActorPath is the full name, whether the actor exists or not.
- ActorRef points to an actor which was started; an *incarnation*.

ActorPath can only optimistically send a message.

ActorRef can be watched.

ActorRef example: `akka://HelloWorld/user/greeter#43428347`

Resolving an ActorPath

```scala
import akka.actor.{ Identify, ActorIdentity }
case class Resolve(path: ActorPath)
case class Resolved(path: ActorPath, ref: ActorRef)
case class NotResolved(path: ActorPath)

class Resolver extends Actor {
  def receive = {
    case Resolve(path) =>
      context.actorSelection(path) ! Identify((path, sender))
    case ActorIdentity((path, client), Some(ref)) =>
      client ! Resolved(path, ref)
    case ActorIdentity((path, client), None) =>
      client ! NotResolved(path)
  }
}
```

Relative Actor Paths

Looking up a grand-child:

- `context.actorSelection(”child/grandchild”)`

Looking up a sibling:

- `context.actorSelection(”../sibling”)`

Looking up from the local root:

- `context.actorSelection(”/user/app”)`

Broadcasting using wildcards:

- `context.actorSelection(”/user/controllers/*”)`

What is a Cluster?

A set of nodes (actor systems) about which all members are in agreement.

These nodes can then collaborate on a common task.

The Formation of a Cluster

A single node can declare itself a cluster ("join itself").

A single node can join a cluster:

- a request is sent to any member
- once all current members know about the new node it is declared part of the cluster

Information is spread using a gossip protocol.

Starting Up a Cluster (1)

Prerequisites:

- `"com.typesafe.akka" %% "akka-cluster" % "2.2.1"`
- configuration enabling cluster module:

  ```
  akka {
    actor {
      provider = akka.cluster.ClusterActorRefProvider
    }
  }
  ```

  in `application.conf` or as `-Dakka.actor.provider=...`

Starting Up a Cluster (2)

```
class ClusterMain extends Actor {
  val cluster = Cluster(context.system)
  cluster.subscribe(self, classOf[ClusterEvent.MemberUp])
  cluster.join(cluster.selfAddress)

  def receive = {
    case ClusterEvent.MemberUp(member) =>
      if (member.address != cluster.selfAddress) {
        // someone joined
      }
  }
}
```

This will start a single-node cluster on port 2552.

Starting Up a Cluster (3)

This needs configuration `akka.remote.netty.tcp.port = 0`.

```
class ClusterWorker extends Actor {
  val cluster = Cluster(context.system)
  cluster.subscribe(self, classOf[ClusterEvent.MemberRemoved])
  val main = cluster.selfAddress.copy(port = Some(2552))
  cluster.join(main)

  def receive = {
    case ClusterEvent.MemberRemoved(m, _) =>
      if (m.address == main) context.stop(self)
  }
}
```

Cluster-Aware Routing (1)

```scala
class ClusterReceptionist extends Actor {
  val cluster = Cluster(context.system)
  cluster.subscribe(self, classOf[MemberUp])
  cluster.subscribe(self, classOf[MemberRemoved])

  override def postStop(): Unit = {
    cluster.unsubscribe(self)
  }

  def receive = ...
}
```

Cluster-Aware Routing (2)

```
def receive = awaitingMembers

val awaitingMembers: Receive = {
  case current: ClusterEvent.CurrentClusterState =>
    val addresses = current.members.toVector map (_.address)
    val notMe = addresses filter (_ != cluster.selfAddress)
    if (notMe.nonEmpty) context.become(active(notMe))
  case MemberUp(member) if member.address != cluster.selfAddress =>
    context.become(active(Vector(member.address)))
  case Get(url) => sender ! Failed(url, ”no nodes available”)
}

def active(addresses: Vector[Address]): Receive = ...
```

Cluster-Aware Routing (3)

```
def active(addresses: Vector[Address]): Receive = {
  case MemberUp(member) if member.address != cluster.selfAddress =>
    context.become(active(addresses :+ member.address))
  case MemberRemoved(member, _) =>
    val next = addresses filterNot (_ == member.address)
    if (next.isEmpty) context.become(awaitingMembers)
    else context.become(active(next))
  ...
}
```

Cluster-Aware Routing (4)

```
def active(addresses: Vector[Address]): Receive = {
  ...
  case Get(url) if context.children.size < addresses.size =>
    val client = sender
    val address = pick(addresses)
    context.actorOf(Props(new Customer(client, url, address)))
  case Get(url) =>
    sender ! Failed(url, ”too many parallel queries”)
}
```

Remote Deployment (1)

```
class Customer(client: ActorRef, url: String, node: Address) extends Actor {
  implicit val s = context.parent

  override val supervisorStrategy = SupervisorStrategy.stoppingStrategy
  val props = Props[Controller].withDeploy(Deploy(scope = RemoteScope(node)))
  val controller = context.actorOf(props, ”controller”)
  context.watch(controller)

  context.setReceiveTimeout(5.seconds)
  controller ! Controller.Check(url, 2)

  def receive = ...
}
```

Remote Deployment (2)

```
implicit val s = context.parent

def receive = ({
  case ReceiveTimeout =>
    context.unwatch(controller)
    client ! Receptionist.Failed(url, ”controller timed out”)
  case Terminated(_) =>
    client ! Receptionist.Failed(url, ”controller died”)
  case Controller.Result(links) =>
    context.unwatch(controller)
    client ! Receptionist.Result(url, links)
}: Receive) andThen (_ => context.stop(self))
```