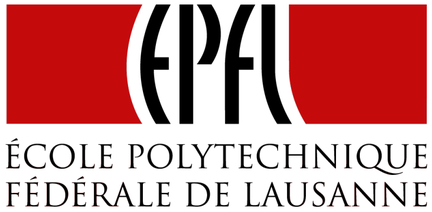

Integrating Streams with (Typed) Actors

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Actors as Sinks or Sources

While most things are simple to implement by combining various stream operators, or implementing custom GraphStages (which we do not cover in this course, but are not too hard once you get to know them), sometimes it is very useful to inter-op between Actors and Streams.

This can be done using simple `!` messages in-line, however then you'd lose the back-pressure aspects of streams (sometimes this is absolutely fine though).

In this section we introduce a number of provided operators that integrate streams and actors.

Typed ActorRef as Source

As such, we can trivially use an Actor as Sink:

```
val ref: ActorRef[String] =
  ActorSource.actorRef[String](
    completionMatcher = { case "complete" => },
    failureMatcher = PartialFunction.empty,
    bufferSize = 256,
    OverflowStrategy.dropNew
  ) // or Head, or Tail, or Buffer
  .to(Sink.ignore).run()

ref ! "one"
ref ! "two"
ref ! "complete"
```

(For an untyped actor this would be simply `Source.actorRef`).

Alternatives to sourcing an Akka Stream

Alternatively one can use a simple `Source.queue[T]` to materialize a `SourceQueueWithComplete[T]`, which can be used to `offer` elements into a stream. This is also nicely usable from within Actors other normal functions.

```
val queue: SourceQueueWithComplete[Int] =
  Source.queue[Int](bufferSize = 1024, OverflowStrategy.dropBuffer)
  .to(Sink.ignore)
  .run()

val r1: Future[QueueOfferResult] = queue.offer(1)
val r2: Future[QueueOfferResult] = queue.offer(2)
val r3: Future[QueueOfferResult] = queue.offer(3)
```

Typed Actor as Sink, no backpressure

```
val ref = spawn(Behaviors.receiveMessage[String] {
  case msg if msg.startsWith("FAILED: ") =>
    throw new Exception(s"Stream failed: $msg")
  case "DONE" =>
    Behaviors.stopped
  case msg =>
    /* handle and... */
    Behaviors.same
})

Source(1 to 10).map(_ + "!")
  .to(ActorSink.actorRef(
    ref = ref, onCompleteMessage = "DONE",
    onFailureMessage = ex   "FAILED: " + ex.getMessage)
  ).run()
```

Typed Actor as Sink, with back-pressure

Preserving back-pressure in communicating with Actors is also possible, however requires collaboration from the Actor.

Our ACK signal will be `Ack`. And we need a protocol to talk to the actor

```
sealed trait Ack
case object Ack extends Ack
```

Typed Actor as Sink, with back-pressure

Preserving back-pressure in communicating with Actors is also possible, however requires collaboration from the Actor.

```
sealed trait AckProtocol
case class Init(streamSender: ActorRef[Ack]) extends AckProtocol

case class Msg(streamSender: ActorRef[Ack], msg: String)
  extends AckProtocol

case object Complete extends AckProtocol
case object Failed(ex: Throwable) extends AckProtocol
```

Typed Actor as Sink, with back-pressure

First we implement a pilot behavior that will accept the stream's messages:

```
val pilot: Behaviors.Receive[AckProtocol] =
  Behaviors.receiveMessage[AckProtocol] {
    case m @ Init(sender)   =>
      // do something on init ...
      sender ! Ack
      Behaviors.same
    case m @ Msg(sender, _) =>
      // do something with each message ...
      sender ! Ack
      Behaviors.same
    case m                  =>
      // knowingly ignore others
      Behaviors.ignore
  }
```

Typed Actor as Sink, with back-pressure

Next we spawn and

```
val targetRef: ActorRef[AckProtocol] = spawn(pilot) // testkit's spawn

val source: Source[String, NotUsed] = Source.single("")

val in: NotUsed =
  source
    .runWith(ActorSink.actorRefWithAck(
      ref = targetRef,
      messageAdapter = Msg(_, _),
      onInitMessage = Init(_),
      ackMessage = Ack,
      onCompleteMessage = Complete,
      onFailureMessage = Failed(_))
    )
```

Asking Actors in a Flow

It sometimes is useful to delegate some operation to an Actor, since perhaps it is highly dynamic and performs other kinds of updating it's state upon which the function it performs on an element changes (e.g. by or being applied external configuration).

One can ask actors by using the inline `ask` operator:

```
val replier = spawn(Behaviors.receiveMessage[Asking] {
  case asking =>
    asking.replyTo ! Reply(asking.s + "!!!")
    Behaviors.same
})
```

Asking Actors in a Flow

For the `ask` message types to align, we need to "pack" the element into the `Asking` type, which the actor is able to handle, and we provide it a ref that it is supposed to answer to; That refs type determines the outgoing element type of the resulting `Source`.

```
val replier: ActorRef[Asking]

val in: Future[immutable.Seq[Reply]] =
  Source.repeat("hello")
    .via(
      ActorFlow.ask(replier)(
        (el, replyTo: ActorRef[Reply]) => Asking(el, replyTo)
      )
    )
    .take(3) // : Source[Reply, _]
```

Summary

In this video we learnt:

- how to inter-op between (typed) Actors and Streams
- how to create ActorRefs that serve as sources of streams
- how to use `ask` properly within streams to query Actors