ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Streaming Protocols & Stream Composition

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Streaming protocols

The concept of streaming is nothing new.

In fact one of the most popular protocols in use nowadays also is a streaming protocol.

Protocol classification reminder

Protocols can be classified as either *stream* or *message* -based.

Stream based protocols generally speaking will expose "(potentially) infinite stream of data", and writes and reads operate by adding/reading from this infinite stream.

Examples:

- TCP

Message based protocols allow sending/reading a specific "single" message, framing and other implementation details behind obtaining an *entire* message, are hidden from end users. Examples of message protocols:

Examples:

- UDP (datagrams)
- Actor messaging

Using TCP in Akka Streams

Reactive Streams, including Akka Streams, are message based – you can receive a single element and the request semantics also are defined in terms of *elements*.

It is however possible to combine the two, and in combination with a framing mechanism, obtain an useful, back-pressured way to utilise raw TCP connections in your applications.

TCP Echo Server in Akka Streams

A simple TCP echo server may be implemented like this:

```
import akka.actor._
import akka.stream.scaladsl._

implicit val system = ActorSystem(”tcp-echo”)
implicit val mat = ActorMaterializer()

val echoLogic = Flow[ByteString]
Tcp(system).bindAndHandle(echoLogic, ”127.0.0.1”, 1337)
```

TCP Client using Akka Streams

The client side of our echo example, can send messages and print responses:

```scala
val clientFlow: Flow[ByteString, ByteString, Future[Tcp.OutgoingConnection]] =
  Tcp().outgoingConnection("127.0.0.1", 1337)

val localDataSource =
  Source.repeat(ByteString("Hello!"))
    .throttle(1, per = 1.second, maximumBurst = 10, mode = ThrottleMode.shaping)

val localDataSink =
  Sink.foreach[ByteString](data => println(s"from server: $data"))

localDataSource.via(clientFlow).to(localDataSink)
  .run()
```

Side note: Materialized value of the clientFlow

```
val clientFlow: Flow[
    ByteString, // incoming element type
    ByteString, // outgoing element type
    Future[Tcp.OutgoingConnection] // materialized value type
  ] = Tcp().outgoingConnection(”127.0.0.1”, 1337)
```

Materialized values are tremendously useful to carry additional information or control structures.

```
clientFlow.mapMaterializedValue(_.map { connection =>
  println(s”Connection established; ” +
    s”  local address ${connection.localAddress}, ” +
    s”  remote: ${connection.remoteAddress}”)})
```

Information obtained only once the stream has materialized is exposed using the *materialized value*.

Stream composability

A key feature of Akka (and Reactive) Streams is that they are nicely *composable*.

A type is composable when it is possible to combine multiple instances.

> *In Scala this is often also associated with the composition being type-safe; I.e if a composition of objects compiles, it is expected to be correct.*

Using Stream composability for testing

In the previous example, we had defined the server's logic as a `Flow[ByteString, ByteString, _]`.

We can make use of this in order to test the logic *without having to invoke the real* TCP infrastructure.

```
"echo a single value" in {
  val source = Source.single("hello").map(ByteString(_))
  val sink = Sink.seq[ByteString]

  val futureSeq = source.via(echoLogic).runWith(sink)

  futureSeq.futureValue shouldEqual Seq(ByteString("hello"))
}
```

Simulating TCP, with stub streams

We can do the same for the client code, where we want to stub-out the server side.

In other words, we can provide a `Flow` that will *act as-if* it was the real TCP connection.

```
def run(connection: Flow[ByteString, ByteString, _]) =
 // our client logic



run(Tcp().outgoingConnection(”127.0.0.1”, 1337)) // real TCP

run(Flow[ByteString]) // ”mock” TCP
```

Summary

In this video we learnt:

- the difference between streaming and message based network protocols
- how Akka Streams and reactive streams can make use of TCP
- how to use the compositional building blocks of Akka Streams to enable nice “pluggable” testing