

Reaktive Programmierung
Vorlesung 12 vom 16.06.15: Reactive Streams III

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- ▶ Teil I: Grundlegende Konzepte
- ▶ Teil II: Nebenläufigkeit
 - ▶ Futures and Promises
 - ▶ Das Aktorenmodell
 - ▶ Aktoren und Akka
 - ▶ Reaktive Datenströme - Observables
 - ▶ Reaktive Datenströme - Back Pressure und Spezifikation
 - ▶ Reaktive Datenströme - Akka Streams
- ▶ Teil III: Fortgeschrittene Konzepte

Rückblick: Akka Streams

- ▶ Vollständige Implementierung der **Reactive Streams** Spezifikation
- ▶ Basiert auf **Datenflussgraphen** und **Materialisierern**
- ▶ Datenflussgraphen werden als **Aktornetzwerk** materialisiert
- ▶ Fast final (aktuelle Version 1.0-RC3)

Heute

- ▶ Datenflussgraphen
 - ▶ geschlossen
 - ▶ partiell
 - ▶ zyklisch
- ▶ Puffer und Back-Pressure
- ▶ Fehlerbehandlung
- ▶ Integration mit Aktoren
- ▶ Anwendungsbeispiel: akka-http
 - ▶ Routen
 - ▶ HTTP
 - ▶ WebSockets

Akka Streams - Grundkonzepte

Datenstrom (Stream) – Ein Prozess, der Daten überträgt und transformiert

Element – Recheneinheit eines Datenstroms

Back-Pressure – Konsument signalisiert (asynchron) Nachfrage an Produzenten

Verarbeitungsschritt (Processing Stage) – Bezeichnet alle Bausteine, aus denen sich ein Datenfluss oder Datenflussgraph zusammensetzt.

Quelle (Source) – Verarbeitungsschritt mit genau einem Ausgang

Senke (Sink) – Verarbeitungsschritt mit genau einem Eingang

Datenfluss (Flow) – Verarbeitungsschritt mit jeweils genau einem Ein- und Ausgang

Ausführbarer Datenfluss (RunnableFlow) – Datenfluss, der an eine Quelle und einen Senke angeschlossen ist

Akka Streams - Beispiel

```
implicit val system = ActorSystem("example")
implicit val materializer = ActorFlowMaterializer()

val source = Source(1 to 10)
val sink = Sink.fold[Int,Int](0)(_ + _)
val sum: Future[Int] = source runWith sink
```

Datenflussgraphen

- ▶ Operatoren sind Abzweigungen im Graphen
- ▶ z.B. Broadcast (1 Eingang, n Ausgänge) und Merge (n Eingänge, 1 Ausgang)
- ▶ Scala DSL um Graphen darzustellen

```
val g = FlowGraph.closed() { implicit builder =>
  val in = source
  val out = sink
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))
  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
      bcast ~> f4 ~> merge
}
```

Operatoren in Datenflussgraphen

- ▶ Auffächern
 - ▶ Broadcast [T] – Verteilt eine Eingabe an n Ausgänge
 - ▶ Balance [T] – Teilt Eingabe gleichmäßig unter n Ausgängen auf
 - ▶ UnZip [A,B] – Macht aus [(A,B)]-Strom zwei Ströme [A] und [B]
 - ▶ FlexiRoute [In] – DSL für eigene Fan-Out Operatoren
- ▶ Zusammenführen
 - ▶ Merge [In] – Vereinigt n Ströme in einem
 - ▶ MergePreferred [In] – Wie Merge, hat aber einen präferierten Eingang
 - ▶ ZipWith [A,B,...,Out] – Fasst n Eingänge mit einer Funktion f zusammen
 - ▶ Zip [A,B] – *ZipWith* mit zwei Eingängen und $f = (_, _)$
 - ▶ Concat [A] – Sequentialisiert zwei Ströme
 - ▶ FlexiMerge [Out] – DSL für eigene Fan-In Operatoren

Partielle Datenflussgraphen

- ▶ Datenflussgraphen können partiell sein:

```
val pickMaxOfThree = FlowGraph.partial() {  
  implicit builder =>  
  
  val zip1 = builder.add(ZipWith[Int,Int,Int](math.max))  
  val zip2 = builder.add(ZipWith[Int,Int,Int](math.max))  
  
  zip1.out ~> zip2.in0  
  
  UniformFanInShape(zip2.out, zip1.in0, zip1.in1,  
    zip2.in1)  
}
```

- ▶ Offene Anschlüsse werden später belegt

Sources, Sinks und Flows als Datenflussgraphen

- ▶ Source — Graph mit genau einem offenen Ausgang

```
Source(){ implicit builder =>
  outlet
}
```

- ▶ Sink — Graph mit genau einem offenen Eingang

```
Sink() { implicit builder =>
  inlet
}
```

- ▶ Flow — Graph mit jeweils genau einem offenen Ein- und Ausgang

```
Flow() { implicit builder =>
  (inlet,outlet)
}
```

Zyklische Datenflussgraphen

- ▶ Zyklen in Datenflussgraphen sind erlaubt:

```
val input = Source(Stream.continually(readLine()))
```

```
val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}

  input ~> merge ~> print ~> bcast ~> Sink.ignore
          merge      <~      bcast
}

```

- ▶ Hört nach kurzer Zeit auf etwas zu tun — Wieso?

Zyklische Datenflussgraphen

- ▶ Besser:

```
val input = Source(Stream.continually(readLine()))

val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}
  val buffer = Flow.buffer(10, OverflowStrategy.dropHead)

  input ~> merge ~> print ~> bcast ~> Sink.ignore
      merge <~ buffer <~ bcast
}
```

Pufferung

- ▶ Standardmäßig werden bis zu **16 Elemente** gepuffert, um parallele Ausführung von Streams zu erreichen.
- ▶ Danach: Backpressure

```
Source(1 to 3)
```

```
.map( i => println(s"A: $i"); i)  
.map( i => println(s"B: $i"); i)  
.map( i => println(s"C: $i"); i)  
.map( i => println(s"D: $i"); i)  
.runWith(Sink.ignore)
```

- ▶ Ausgabe nicht deterministisch, wegen paralleler Ausführung
- ▶ Puffergrößen können angepasst werden (Systemweit, Materialisierer, Verarbeitungsschritt)

Fehlerbehandlung

- ▶ Standardmäßig führen Fehler zum Abbruch:

```
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
```

- ▶ `result = Future(Failure(ArithmeticException))`
- ▶ Materialisierer kann mit Supervisor konfiguriert werden:

```
val decider: Supervisor.Decider = {
  case _ : ArithmeticException => Resume
  case _ => Stop
}
implicit val materializer = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system)
    .withSupervisionStrategy(decider))
```

- ▶ `result = Future(Success(228))`

Integration mit Aktoren - ActorPublisher

- ▶ ActorPublisher ist ein Akteur, der als Source verwendet werden kann.

```
class MyActorPublisher extends ActorPublisher[String] {  
  def receive = {  
    case Request(n) =>  
      for (i ← 1 to n) onNext("Hallo")  
    case Cancel =>  
      context.stop(self)  
  }  
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```

Integration mit Aktoren - ActorSubscriber

- ▶ ActorSubscriber ist ein Aktor, der als Sink verwendet werden kann.

```
class MyActorSubscriber extends ActorSubscriber {  
  def receive = {  
    case OnNext(elem) =>  
      log.info("received {}", elem)  
    case OnError(e) =>  
      throw e  
    case OnComplete =>  
      context.stop(self)  
  }  
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```


Integration für einfache Fälle

- ▶ Für einfache Fälle gibt es `Source.actorRef` und `Sink.actorRef`

```
val source: Source[Foo,ActorRef] = Source.actorRef[Foo](  
  bufferSize = 10,  
  overflowStrategy = OverflowStrategy.backpressure)
```

```
val sink: Sink[Foo,Unit] = Sink.actorRef[Foo](  
  ref = myActorRef,  
  onCompleteMessage = Bar)
```

- ▶ Problem: Sink hat kein Backpressure. Wenn der Aktor nicht schnell genug ist, explodiert alles.

Anwendung: akka-http

- ▶ Minimale HTTP-Bibliothek (Client und Server)
- ▶ Basierend auf *akka-streams* — reaktiv
- ▶ From scratch — keine Altlasten
- ▶ Kein Blocking — Schnell
- ▶ Scala DSL für Routen-Definition
- ▶ Scala DSL für Webaufrufe
- ▶ Umfangreiche Konfigurationsmöglichkeiten

Low-Level Server API

- ▶ HTTP-Server wartet auf Anfragen:

```
Source[IncomingConnection, Future[ServerBinding]]
```

```
val server = Http.bind(interface = "localhost", port =  
    8080)
```

- ▶ Zu jeder Anfrage gibt es eine Antwort:

```
val requestHandler: HttpRequest => HttpResponse = {  
    case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>  
        HttpResponse(entity = "PONG!")  
}
```

```
val serverSink =  
    Sink.foreach(_.handleWithSyncHandler(requestHandler))
```

```
serverSource.to(serverSink)
```

High-Level Server API

- ▶ Minimalbeispiel:

```
implicit val system = ActorSystem("example")
implicit val materializer = ActorFlowMaterializer()

val routes = path("ping") {
  get {
    complete { <h1>PONG!</h1> }
  }
}

val binding =
  Http().bindAndHandle(routes, "localhost", 8080)
```

HTTP

- ▶ HTTP ist ein Protokoll aus den frühen 90er Jahren.
- ▶ Grundidee: Client sendet **Anfragen** an Server, Server **antwortet**
- ▶ Verschiedene Arten von Anfragen
 - ▶ GET — Inhalt abrufen
 - ▶ POST — Inhalt zum Server übertragen
 - ▶ PUT — Resource unter bestimmter URI erstellen
 - ▶ DELETE — Resource löschen
 - ▶ ...
- ▶ Antworten mit Statuscode. z.B.:
 - ▶ 200 — Ok
 - ▶ 404 — Not found
 - ▶ 501 — Internal Server Error
 - ▶ ...

Das Server-Push Problem

- ▶ HTTP basiert auf der Annahme, dass der Webclient den (statischen) Inhalt **bei Bedarf** anfragt.
- ▶ Moderne Webanwendungen sind alles andere als statisch.
- ▶ Workarounds des letzten Jahrzehnts:
 - ▶ **AJAX** — Eigentlich *Asynchronous JavaScript and XML*, heute eher **AJAJ**
— Teile der Seite werden dynamisch ersetzt.
 - ▶ **Polling** — "Gibt's etwas Neues?", "Gibt's etwas Neues?", ...
 - ▶ **Comet** — Anfrage mit langem Timeout wird erst beantwortet, wenn es etwas Neues gibt.
 - ▶ **Chunked Response** — Server antwortet stückchenweise

WebSockets

- ▶ TCP-Basiertes **bidirektionales** Protokoll für Webanwendungen
- ▶ Client öffnet nur **einmal** die Verbindung
- ▶ Server und Client können **jederzeit** Daten senden
- ▶ Nachrichten ohne Header (1 Byte)
- ▶ **Ähnlich** wie Aktoren:
 - ▶ JavaScript Client sequentiell mit lokalem Zustand (\approx Actor)
 - ▶ `WebSocket.onmessage` \approx `Actor.receive`
 - ▶ `WebSocket.send(msg)` \approx `sender ! msg`
 - ▶ `WebSocket.onclose` \approx `Actor.postStop`
 - ▶ Außerdem `onerror` für Fehlerbehandlung.

WebSockets in akka-http

- ▶ WebSockets ist ein `Flow[Message,Message,Unit]`
- ▶ Können über bidirektional Flows gehandhabt werden
 - ▶ `BidiFlow[-I1,+O1,-I2,+O2,+Mat]` – zwei Eingänge, zwei Ausgänge: Serialisieren und deserialisieren.
- ▶ Beispiel:

```
def routes = get {  
  path("ping") (handleWebsocketMessages(wsFlow))  
}
```

```
def wsFlow: Flow[Message,Message,Unit] =  
  BidiFlow.fromFunctions(serialize,deserialize)  
    .join(Flow.collect {  
      case Ping => Pong  
    })
```


Zusammenfassung

- ▶ **Datenflussgraphen** repräsentieren reaktive Berechnungen
 - ▶ Geschlossene Datenflussgraphen sind ausführbar
 - ▶ Partielle Datenflussgraphen haben **unbelegte** ein oder ausgänge
 - ▶ **Zyklische** Datenflussgraphen sind erlaubt
- ▶ Puffer sorgen für **parallele Ausführung**
- ▶ Supervisor können bestimmte Fehler ignorieren
- ▶ *akka-stream* kann einfach mit *akka-actor* integriert werden
- ▶ Anwendungsbeispiel: *akka-http*
 - ▶ Low-Level API: Request \Rightarrow Response
 - ▶ HTTP ist **pull basiert**
 - ▶ **WebSockets** sind **bidirektional** \rightarrow Flow

Bonusfolie: WebWorkers

- ▶ JavaScript ist singlethreaded.
- ▶ Bibliotheken machen sich keinerlei Gedanken über Race-Conditions.
- ▶ Workaround: Aufwändige Berechnungen werden gestückelt, damit die Seite responsiv bleibt.
- ▶ Lösung: HTML5-WebWorkers (Alle modernen Browser)
 - ▶ `new WebWorker(file)` startet neuen Worker
 - ▶ Kommunikation über `postMessage`, `onmessage`, `onerror`, `onclose`
 - ▶ Einschränkung: Kein Zugriff auf das DOM — lokaler Zustand
 - ▶ WebWorker können weitere WebWorker erzeugen
 - ▶ *"Poor-Man's Actors"*