S-BPM Groupware

Internet-Praktikum TK WS14/15

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vorstellung Akka

André Röder

Telecooperation Lab

Gliederung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundprinzipien Akka
- Actors und dazugehörige Konzepte
- Messaging
- Typed Actors
- Logging
- Futures

Grundprinzipien Akka

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Actors**
  - Einfache, high-level Abstraktion für Nebenläufigkeit (concurrency) und Parallelität (parallelism)
  - Asynchron, non-blocking und event-driven
- **Fehlertoleranz**
  - „let-it-crash“ Semantik: Systeme sollen sich durch eine Supervisor Hierarchie selbstständig heilen (auch über mehrere JVM hinweg)
- **Location Transparency**
  - Alle Interaktionen erfolgen über Messaging und sind vollkommen von der darunterliegenden Umgebung unabhängig
- Verwendung als library oder microkernel

Actors

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundgedanke: Divide & Conquer – Tasks werden so lange ge- und verteilt bis ein Actor sie alleine lösen kann
- Actors sind Container (Objekte) für Zustand, Verhalten, Kinder, Mailbox und eine Supervisor Strategie
- Actors folgen, wie eine reale Organisation, einer Hierarchie – jeder Actor hat genau einen Supervisor

Best Practice

- Fehlerbehandlung sollte immer im Supervisor statt finden, da er die Aufgabenbereiche seiner Kinder kennt
- Error Kernel Pattern: Ein Actor sollte, sofern sein Zustand wichtig ist, gefährliche Aufgaben an Kinder weiter geben und mögliche Fehlerzustände der Kinder managen

Actors # 2

TECHNISCHE UNIVERSITÄT DARMSTADT

- **State & Behavior**
  - Der Zustand eines Actors ist nach außen abgeschirmt (wie ein Objekt)
  - Ein Neustart des Actors erzeugt ihn in seinem Ursprungsverhalten, in der Zwischenzeit veränderte Verhaltensweisen gehen verloren (become/unbecome)

- **Mailbox**
  - Jeder Actor hat eine Mailbox deren Nachrichten nach ihrer Sendezeit gequeued werden
    - Wenn eine Actor mehrere Nachrichten an einen anderen sendet, sind diese genau in der Reihenfolge in der Warteschlange
    - Senden mehrere Actoren Nachrichten an einen Actor, kann durch Threading nicht sichergestellt werden, welche Nachricht zuerst ankommt
    - Jeder Actor MUSS die nächste Nachricht verarbeiten, es gibt kein scannen der Queue nach passenden Nachrichten
    - Tritt ein Fehler bei der Verarbeitung auf und wird Neustart als Supervisor Strategie gewählt, so geht die aktuelle Nachricht verloren. Die Queue bleibt jedoch ansonsten bestehen.

Actors # 3

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Children**
  - Jeder Actor ist ein potentieller Supervisor und hat eine Liste seiner Children, welche dem ausgeführten Kontext zur Verfügung steht
  - Die möglichen Operationen sind Erzeugung oder Beendung von Children

- **Supervisor Strategy**
  - Der Supervisor regelt die Fehlerbehandlung seiner Children
  - Es gibt genau eine Strategie zur Fehlerbehandlung, sollten unterschiedliche Fehlerbehandlungen nötig sein, müssen die Children gruppiert werden und eine tiefere Hierarchie erzeugt werden

Supervision

TECHNISCHE UNIVERSITÄT DARMSTADT

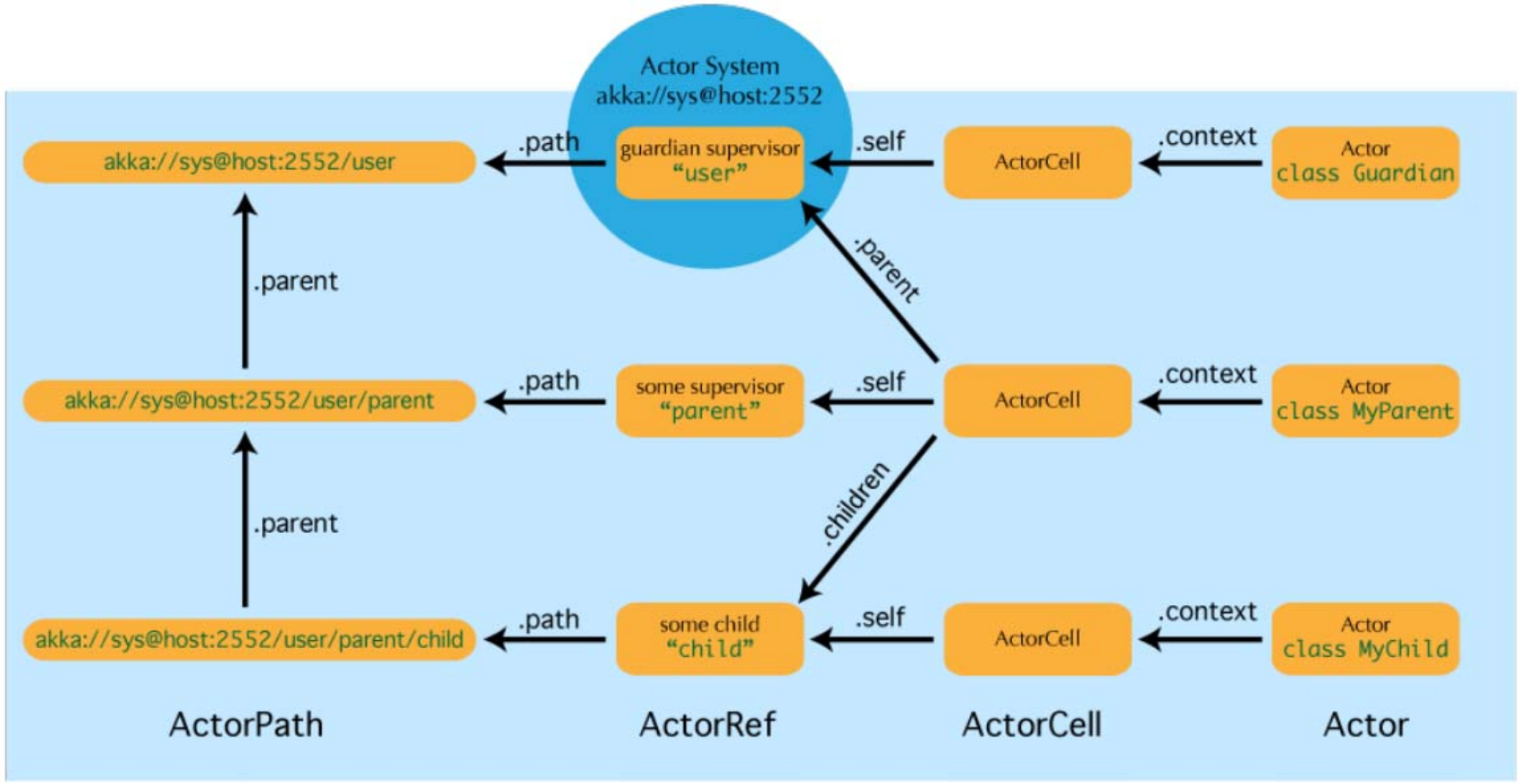
- Wenn ein Actor einen Fehler bemerkt (z.B. eine Exception) stoppt er und meldet er den Fehler an seinen Supervisor

- Dieser hat nun folgende Möglichkeiten:
  - Actor weiter ausführen und den internen Zustand bewahren
  - Actor neustarten (und internen Zustand verlieren)
  - Actor komplett beenden
  - Fehler eskalieren (ebenfalls ein Fehler melden und jetzt seinem Supervisor diese Entscheidung überlassen)

Hinweis
Die Befehle gelten immer für die komplette Hierarchie (Neustart eines Childs erzeugt ebenfalls den Neustart seiner Children etc.)

Akka Hierarchie

TECHNISCHE UNIVERSITÄT DARMSTADT



Best Practice
Immer die ActorRef verarbeiten (self), nicht den Actor selbst (über this)

Code: Creating an Actor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" ⇒ log.info("received test")
    case _      ⇒ log.info("received unknown message")
  }
}
```

Ein eigener Actor erweitert immer die Actor Klasse und muss die Funktion receive definieren

```
object Main extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name = "myactor")
```

actorOf erzeugt eine Actor Instanz für das actorSystem. Rückgabe Typ ist actorRef

Hinweis
Actors werden automatisch gestartet. Dabei wird der Hook preStart() automatisch ausgeführt und kann überschrieben werden.

Code: Messaging

TECHNISCHE UNIVERSITÄT DARMSTADT

```
case object Tick
case object Get

class Counter extends Actor {
  var count = 0

  def receive = {
    case Tick => count += 1
    case Get  => sender ! count
  }
}

object AkkaProjectInScala extends App {
  val system = ActorSystem("AkkaProjectInScala")

  val counter = system.actorOf(Props[Counter])

  counter ! Tick
  counter ! Tick
  counter ! Tick

  implicit val timeout = Timeout(5 seconds)

  (counter ? Get) onSuccess {
    case count => println("Count is " + count)
  }

  system.shutdown()
}
```

Es gibt zwei Möglichkeiten zum senden:

- **!** für tell (fire & forget)
- **?** für ask (auf Antwort warten und reagieren)

Eine weitere Möglichkeit ist das weiterleiten mit **forward**

Future Callback

Best Practice
Immer tell benutzen wenn möglich, da ask aufwändiger (Performance)

Typed Actors

TECHNISCHE UNIVERSITÄT DARMSTADT

- TypedActors ist die Implementation des Active Object Pattern
  - Methodenaufruf und –ausführen werden von einander getrennt
- Er ist ein Actor bei welcher statt Messages zusätzlich Methodenaufrufe empfängt und in der Regel die Schnittstelle zu nicht-Actor Code bereitstellt

Beispiel hier: http://letitcrash.com/post/19074284309/when-to-use-typedactors

Best Practice
Nur Methoden deklarieren die Unit (void) oder einen Future liefern, um Blockierungen zu verhindern.

Logging

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  override def preStart() = {
    log.debug("Starting")
  }
  override def preRestart(reason: Throwable, message: Option[Any]) {
    log.error(reason, "Restarting due to [{}] when processing [{}]",
      reason.getMessage, message.getOrElse(""))
  }
  def receive = {
    case "test" ⇒ log.info("Received test")
    case x      ⇒ log.warning("Received unknown message: {}", x)
  }
}
```

Log Level: debug

Log Level: error

Log Level: warning

- Logging wird asynchron über ein Event-Bus ausgeführt
- Darüber können Event Handler definiert werden, die weitere Aktionen auslösen

Futures

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Future ist eine Struktur die einem den Zugriff auf ein zukünftiges (nebenläufige), möglicherweise eintretendes Ergebnis ermöglicht
- Wird häufig als Rückgabewert eines Asks (?) genutzt, kann jedoch auch außerhalb eines Actors eingesetzt werden

```
implicit val timeout = Timeout(5 seconds)
val future = actor ? msg // enabled by the "ask" import
val result = Await.result(future, timeout.duration).asInstanceOf[String]
```

Await ist blockierend und sollte daher nur in Ausnahmefällen genutzt werden

Future liefert immer Typ Any zurück, daher cast

- Listener auf Future Events sind: onComplete, onSuccess, onFailure
- Non-blocking Futures:

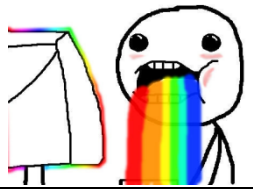
```
val f1 = ask(actor1, msg1)
val f2 = ask(actor2, msg2)

val f3 = for {
  a ← f1.mapTo[Int]
  b ← f2.mapTo[Int]
  c ← ask(actor3, (a + b)).mapTo[Int]
} yield c

val result = Await.result(f3, 1 second).asInstanceOf[Int]
```

Warnung
Futures werden ab Version 2.1 Scala Bestandteil und gehören wohl nicht mehr zu Akka.

Wie installieren?



TECHNISCHE UNIVERSITÄT DARMSTADT

- Akka ist in Typesafe enthalten und Beispielprojekte können mit g8 ausgecheckt werden
  - z.B. g8 typesafehub/akka-first-tutorial-scala
- Vorsicht: Die Projekte können nicht in Eclipse importiert werden, sie müssen vorher umgewandelt werden
  - Es muss das Eclipse SBT Plugin ausgeführt werden um die Ordner in Eclipse Projekte umzuwandeln
  - Hierzu im Beispielprojektordner unter \project (z.B. akka-project-in-scala\project) die Datei **plugins.sbt** anlegen oder erweitern

```
// Comment to get more information during initialization
logLevel := Level.Warn

// The Typesafe repository
resolvers += "Typesafe repository" at "http://repo.typesafe.com/typesafe/releases/"

// Use the eclipse sbt plugin
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.0.0")
```

  - Dann in den Hauptordner navigieren, **SBT** starten in der Konsole und einfach „**eclipse**“ ausführen
  - Schließlich über Import als Projekt in Eclipse importieren