Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Akka und Physisch verteilte Systeme

- Aktoren und entfernte Kommunikation
- Akka.io
- Entfernte Aktoren

Aktoren und (entfernte) Kommunikation

Aktoren und Netzwerk-Kommunikation können auf verschiedene Arten kombiniert werden:

- **Lokal: Aktor; Entfernt: Thread-basierte Kommunikation**
  - für die entfernte Kommunikation werden klassische Thread-basierte Kommunikations-Mechanismen verwendet
  - Lokal kommen bei Bedarf Aktoren zum Einsatz
  - Interaktion Aktor <~> Thread muss realisiert werden
- **Lokale und Entfernte Aktoren**
  - Einheitliches Modell mit Kommunikation von lokalen Aktoren und Aktoren auf anderen Knoten
  - Geeignet für Peer-to-Peer-Anwendungen mit Ortstransparenz der Aktoren
  - Basis des Akka-Clusterings
  - Wenig geeignet für (den Server von) Client-Server-Anwendungen
- **akka.io**
  - Kommunikation im Client-Server-Stil
  - mit Unterstützung der Interaktion von Aktoren und dem Kommunikationssystem

akk.io Übersicht

Package akka.io

- **Scala-Framework**
- **Basiert auf / entspricht dem Modul *spray.io* des Spray-Frameworks [http://spray.io/] (gemeinsame Entwicklung von Spray und Typesafe)**
- **nutzt speziell speziell den Netzwerkcode von Spray der auf NIO- / NIO.2-Features basiert**
- **ist vergleichbar mit einem (minimalen) NIO-Framework**

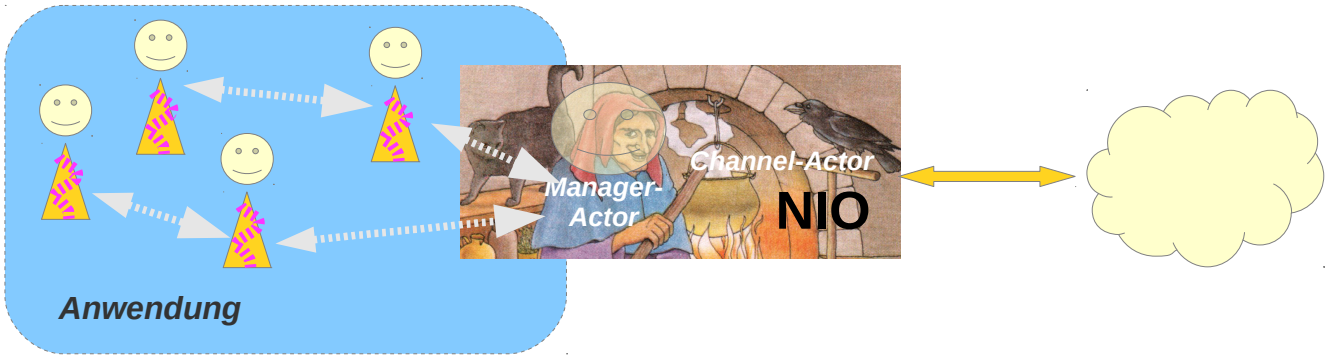
> The guiding design goal for this I/O implementation was to reach extreme scalability, make no compromises in providing an API correctly matching the underlying transport mechanism and to be fully event-driven, non-blocking and asynchronous. The API is meant to be a solid foundation for the implementation of network protocols and building higher abstractions; it is not meant to be a full-service high-level NIO wrapper for end users.

Quelle: http://doc.akka.io/docs/akka/current/scala/io.html

akk.io Übersicht

Prinzip: NIO mit menschlichem Angesicht

- ein Manager-Actor kontrolliert die IO
- die Anwendung kommuniziert mit dem Manager über Kommando-Nachrichten
  Verbindungs-Management (connect, … close) und Nachrichten-Transfer (read, write)



The central element of the implementation is the transport-specific “selector” actor; in the case of TCP this would wrap a `java.nio.channels.Selector`. The channel actors register their interest in readability or writability of their channel by sending corresponding messages to their assigned selector actor. However, the actual channel reading and writing is performed by the channel actors themselves, which frees the selector actors from time-consuming tasks and thereby ensures low latency. The selector actor’s only responsibility is the management of the underlying selector’s key set and the actual select operation, which is the only operation to typically block.

http://doc.akka.io/docs/akka/current/scala/common/io-layer.html

TCP-Beispiel ein einfacher Echo-Server – 1 Übersicht

```
import akka.actor.{ Actor, ActorRef, Props, ActorSystem}
import akka.io.{ IO, Tcp }
import java.net.InetSocketAddress

class ConnectionHandler extends Actor { … }


class Server(managerActor: ActorRef) extends Actor {...}


object EchoServer_Main {

  implicit val system = ActorSystem("TCPEchoSystem")

  //obtain a reference to the manager actor for the given IO
  //extension (implicit parameter system)
  val managerActor = IO(Tcp)

  // create server actor which will accept connections
  val server = system.actorOf(
                Props(classOf[Server], managerActor),
                "server")

  def main(args: Array[String]): Unit = {
    println("starting server as actor " + server)
  }

}
```

Der ConnectionHandler bedient Verbindungen

Der Server nimmt Verbindungen an

Ein Manager-Actor für TCP wird erzeugt. Er dient als Schnittstelle zu Kommunikations-Diensten.

Ein Server wird erzeugt, er erhält eine Referenz auf den Manager-Actor um über diesen seine Kommunikation abwickeln zu können.

TCP-Beispiel ein einfacher Echo-Server – 2 Server

```
class Server(managerActor: ActorRef) extends Actor {

  val acceptor = self  // the actor which will receive incoming connection requests

  // socket bind
  managerActor ! Tcp.Bind(acceptor, new InetSocketAddress("127.0.0.1", 4711))

  def receive = { // msgs from the connection actor

    case Tcp.CommandFailed(_: Tcp.Bind) => context stop self

    case Tcp.Bound(localAddress) => // bind succeeded
      println("Server ready")

    case Tcp.Connected(remoteAdr, localAdr) => // client connected to server
      println("Server accepted connection from " + remoteAdr + ", at " + localAdr)

      val connectionHandler = context.actorOf(Props[ConnectionHandler])

      val connection = sender()

      connection ! Tcp.Register(connectionHandler)
  }
}
```

Ich bin der Handler für für den ServerSocket, also für accept

Bind ist fehlgeschlagen

Bind ist gelungen

Connect eingetroffen

Connection-Handler erzeugen

Der Sender dieser Nachricht repräsentiert die Verbindung

Der connectionHandler wird für die Verbindung installiert

TCP-Beispiel ein einfacher Echo-Server – 3 Connection-Handler

```
class ConnectionHandler extends Actor {

  def receive = {

    case Tcp.Received(data) =>
      println(s"Server received byteString $data from  $sender")
      println(s"   bytestring decoded: ${data.decodeString("UTF-8")}")
      sender() ! Tcp.Write(data) // send back raw data

    case Tcp.PeerClosed =>
      println("Connection closed")
      context stop self

    case x: Any => println("Server received unkown " + x)
  }
}
```

Ein ConnectionHandler nimmt Daten an und sendet sie zurück. …

Solange bis die Verbindung geschlossen wird.

TCP-Beispiel ein einfacher Echo-Client – 1 Main

```
import akka.actor.{ Actor, ActorRef, Props, ActorSystem }
import akka.io.{ IO, Tcp }
import akka.util.ByteString
import java.net.InetSocketAddress

class ClientActor(managerActor: ActorRef) extends Actor {…}

object EchoClient_Main {
  implicit val system = ActorSystem("TCPEchoSystem")

  val managerActor = IO(Tcp)

  val clientActor = system.actorOf(
          Props(classOf[ClientActor],
                managerActor),
          "client")

  def main(args: Array[String]): Unit = {
    Thread.sleep(500)

    for (i <- 1 to 10) {
      val v = scala.io.StdIn.readLine()
      clientActor ! Some(v)
    }
    Thread.sleep(500)
    clientActor ! "close"

    Thread.sleep(1000)
    system.terminate()
  }

}
```

Client sendet Daten via Manager-Aktor an den Server.

Ein Manager-Actor für TCP wird erzeugt. Er dient als Schnittstelle zu Kommunikations-Diensten.

Ein Client-Actor wird erzeugt, er erhält eine Referenz auf den Manager-Actor um über diesen seine Kommunikation abwickeln zu können.

10 Strings einlesen und an ClientActor senden, damit dieser sie an den Server sendet.

TCP-Beispiel ein einfacher Echo-Client – 2 Client

```
class ClientActor(managerActor: ActorRef) extends Actor {

  managerActor ! Tcp.Connect(new InetSocketAddress("127.0.0.1", 4711)) // please connect to server

  def receive = { // start with establishing connection to server

    case Tcp.CommandFailed(_: Tcp.Connect) =>
      println("connect failed")
      context stop self

    case Tcp.Connected(remote, local) =>
      println("connected to " + remote + " at " + local)

      val connection = sender() // sender: internal actor representing the new connection

      connection ! Tcp.Register(self)  // register ClientActor as handler at connection-actor

      context become { // connection is established change behaviour to that of a connection handler
        case Tcp.CommandFailed(w: Tcp.Write) =>
          println("write failed")

        case Tcp.Received(data) =>
          println("ClientActor received  bytes from network: " + data)
          println(" bytes as UTF-8 decoded string: " + data.decodeString("UTF-8"))

        case "close" =>
          connection ! Tcp.Close

        case _: Tcp.ConnectionClosed =>
          println("connection closed")
          context stop self

        case msg @ Some(x) => // local msg
          println("client actor received local msg " + msg)
          println("... and sends to " + ByteString(x.toString + "\n") + " to " + connection)
          connection ! Tcp.Write(ByteString(x.toString + "\n")) // send to server
      }
  }
}
```

UDP-Beispiel ein einfacher Echo-Server

```
import akka.actor.{ Actor, ActorRef, Props, ActorSystem}
import akka.io.{ IO, Udp }
import java.net.InetSocketAddress

class ServerActor(localAdr: InetSocketAddress) extends Actor {
  import context.system

  // the server will listen for incoming pkts, so bind
  val managerActor = IO(Udp) ! Udp.Bind(self, localAdr)

  def receive = {
    case Udp.Bound(local) =>
      context become { // binding completed now listen for pkts
        case Udp.Received(data, remoteAdr) =>
          val client = sender()
          println(s"Server received $data from $remoteAdr")
          println(s"   data as string: ${data.decodeString("UTF-8")}")
          client ! Udp.Send(data, remoteAdr)

        case Udp.Unbind =>
          val client = sender()
          client ! Udp.Unbind

        case Udp.Unbound => context.stop(self)
      }
  }
}

object EchoServer_Main {
  val SERVER_PORT = 4712
  val system = ActorSystem("UDPEchoSystem")
  val serverActor = system.actorOf(
      Props(classOf[ServerActor], new InetSocketAddress("127.0.0.1", SERVER_PORT)),
      "server")

  def main(args: Array[String]): Unit = {
    println("starting serverActor " + serverActor)
  }
}
```

UDP-Beispiel ein einfacher Echo-Client – 1

```
import akka.actor.{Actor, ActorRef, Props, ActorSystem, PoisonPill}
import akka.io.{IO, Udp}
import akka.util.ByteString
import java.net.InetSocketAddress

class ClientActor(localAdr: InetSocketAddress, remoteAdr: InetSocketAddress) extends Actor {
     …
}

object EchoClient_Main {

  val MyPORT = 4711
  val SERVER_PORT = 4712

  val system = ActorSystem("UDPEchoSystem")
  val clientActor = system.actorOf(
    Props(classOf[ClientActor],
      new InetSocketAddress("127.0.0.1", MyPORT),
      new InetSocketAddress("127.0.0.1", SERVER_PORT)), "client")

  def main(args: Array[String]): Unit = {
    Thread.sleep(500)

    for (i <- 1 to 10) {
      val v = scala.io.StdIn.readLine()
      clientActor ! s"Msg Nr. $i : $v"
    }
    Thread.sleep(500)
    clientActor ! PoisonPill

    Thread.sleep(500)
    system.terminate()
  }
}
```

UDP-Beispiel ein einfacher Echo-Client – 2

```
class ClientActor(localAdr: InetSocketAddress, remoteAdr: InetSocketAddress) extends Actor {

  import context.system

  // uses unconnected UDP: only local address is bound
  IO(Udp) ! Udp.Bind(self, localAdr)

  def receive = {
    case Udp.Bound(local) =>

      val server = sender()

      context become {
        case msg: String => // got local msg to be send to server
          println(s"client sends $msg")
          server ! Udp.Send(ByteString(msg), remoteAdr)

        case Udp.Received(data, remote) => // got msg from remote server
          println(s"Client received $data  from $remote")
          println(s"   decoded to ${data.decodeString("UTF-8")}")
      }
  }

}
```

Bytestring

akka.io basiert auf NIO

NIO nutzt java.nio.ByteBuffer

- sehr *low-level* und schwierig zu nutzen

akka.util.ByteString

- Schnittstelle zwischen Anwendung und ByteBuffer

Serialisierung

- Serialisierung ist ein zentrales (oft unterschätztes) Thema aller verteilten Anwendungen
- Java Serialisierung

  (Serializable, DataInput- DataOutputStream, ObjectInput-, ObjectOutputStream...)
- Scala Serialisierung auf der Basis von Java-Features
- Diverse Serialisierungs-Frameworks
  (z.B. Protocol Buffers https://code.google.com/p/protobuf/)
- Akka: erweiterbare konfigurierbare Serialisierung
- Für die Kommunikation von Aktoren über Prozessgrenzen stehen Akka leistungsfähige Serialisierungsmechnanismen zur Verfügung

  siehe http://doc.akka.io/docs/akka/current/scala/serialization.html
- Für die Kommunikation mit komplett externen Prozessen können eigenen Decoder/Decoder geschrieben werden. Dazu werden einige Hilfsmittel zur Verfügung gestellt.

  Diese werden im folgenden Beispiel kurz angesprochen – Das ist nicht das, was z.B. Netty bietet und es ist nicht beabsichtigt, dass Anwendungsentwickler auf dieser Ebene arbeiten

Serialisierung / ByteString

***akka.util.*ByteString**

Helferklasse zum Umgang mit Bytearrays Bytebuffers

Beispiel

Serialisiere / Deserialisiere Nachrichten vom folgendem Typ:

```
abstract class Msg
case class StringMsg(str: String) extends Msg
case class NumberPairMsg(d1: Double, d2: Double) extends Msg
```

Serialisierung / ByteString Beispiel: Serialisiere / Deserialisiere Nachrichten

```
object Msg {

  implicit val byteOrder = java.nio.ByteOrder.BIG_ENDIAN

  val STRING_MSG = 0
  val NUMBER_PAIR_MSG = 1

  def decode(msg: ByteString): Msg = {
    val byteIter: ByteIterator = msg.iterator
    val msgType: Int = byteIter.getInt
    msgType match {
      case STRING_MSG => StringMsg(getString(byteIter))
      case NUMBER_PAIR_MSG => NumberPairMsg(byteIter.getDouble, byteIter.getDouble);
    }
  }

  private def getString(iter: ByteIterator): String = {
    val length = iter.getInt
    val bytes = new Array[Byte](length)
    iter getBytes bytes
    ByteString(bytes).utf8String
  }

  def encode(msg: Msg) : ByteString = msg match {
    case StringMsg(str) =>
      val strBytes = str.getBytes
      val strBytesLen = strBytes.length
      new ByteStringBuilder().putInt(STRING_MSG).putInt(strBytesLen).putBytes(str.getBytes).result
    case NumberPairMsg(d1, d2) =>
      new ByteStringBuilder().putInt(NUMBER_PAIR_MSG).putDoubles(Array[Double](d1, d2)).result
  }

}
```

Serialisierung / ByteString Beispiel: Serialisiere / Deserialisiere Nachrichten

```
import akka.actor.{ Actor, Props, ActorSystem}
import akka.io.{ IO, Udp }
import java.net.InetSocketAddress

class ServerActor(localAdr: InetSocketAddress) extends Actor {
  import context.system

  val managerActor = IO(Udp) ! Udp.Bind(self, localAdr)

  def receive = {
    case Udp.Bound(local) =>
      context become { // binding completed now listen for pkts
        case Udp.Received(data, remoteAdr) =>
          val client = sender()
          println(s"Server received raw $data from $remoteAdr")
          println(s"   raw data as msg: ${Msg.decode(data)}")

        case Udp.Unbind =>
          val client = sender()
          client ! Udp.Unbind

        case Udp.Unbound => context.stop(self)
      }
  }
}

object SerialServer_Main {
  val SERVER_PORT = 4712
  val system = ActorSystem("UDPSerialSystem")
  val serverActor = system.actorOf(
      Props(classOf[ServerActor], new InetSocketAddress("127.0.0.1", SERVER_PORT)),
      "server")

  def main(args: Array[String]): Unit = {
    println("starting serverActor " + serverActor)
  }
}
```

Ein deserialisierender UDP-Server

Serialisierung / ByteString Beispiel: Serialisiere / Deserialisiere Nachrichten

```
import akka.actor.{Actor, Props, ActorSystem, PoisonPill}
import akka.io.{IO, Udp}
import java.net.InetSocketAddress

class ClientActor(localAdr: InetSocketAddress, remoteAdr: InetSocketAddress) extends Actor {
  import context.system

  IO(Udp) ! Udp.Bind(self, localAdr)

  def receive = {
    case Udp.Bound(local) =>
      val server = sender()
      context become {
        case msg: Msg =>
          println(s"client sends String msg $msg")
          server ! Udp.Send(Msg.encode(msg), remoteAdr)
      }
  }
}

object SerialClient_Main {

  val MyPORT = 4711
  val SERVER_PORT = 4712

  val system = ActorSystem("UDPSerialSystem")
  val clientActor = system.actorOf(
    Props(classOf[ClientActor],
      new InetSocketAddress("127.0.0.1", MyPORT),
      new InetSocketAddress("127.0.0.1", SERVER_PORT)), "client")

  def main(args: Array[String]): Unit = {
    Thread.sleep(500)

    (1 to 10).foreach( i =>
      if (i % 2 == 0) { clientActor ! StringMsg(s"string nr $i") } else { clientActor ! NumberPairMsg(i, i+1) })
    Thread.sleep(500)
    clientActor ! PoisonPill

    Thread.sleep(500)
    system.terminate()
  }

}
```

Ein serialisierender UDP-Client

Aktoren und entfernte Kommunikation

Entfernte Kommunikation

- Kommunikation über Prozess- (und eventuell Rechner-) Grenzen hinweg

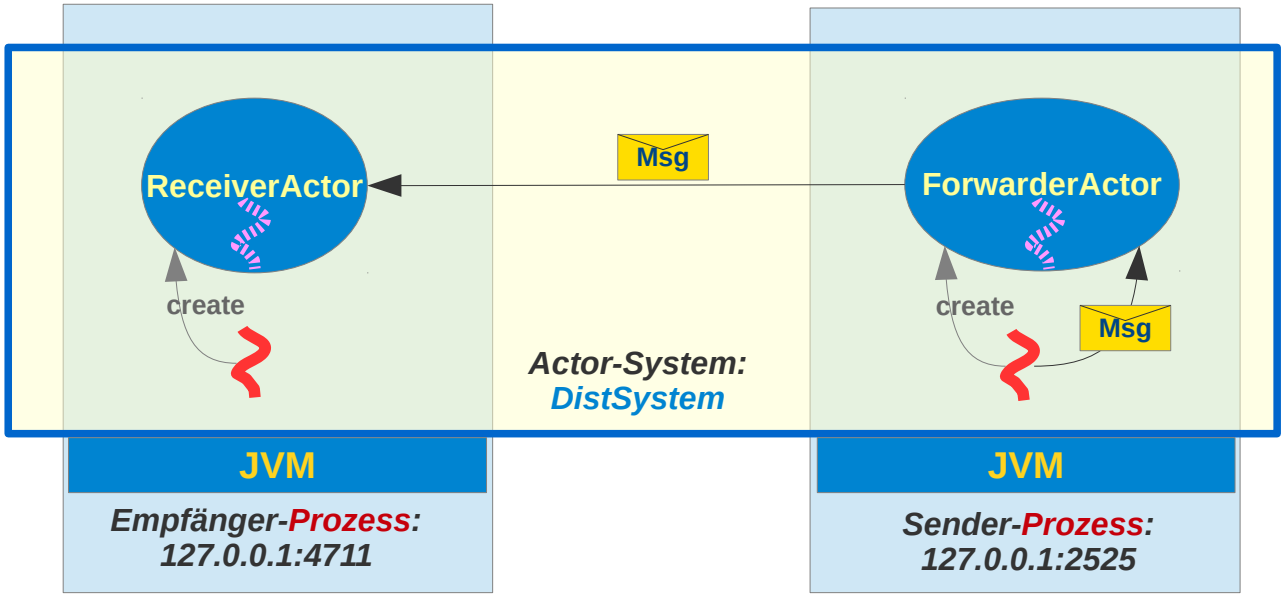
Entfernte Aktoren

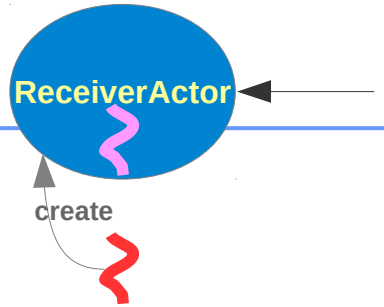
- Aktoren in einem anderen Prozess (eventuell auch auf einem anderen Rechner)
- Konzept / Mechanismus **entfernter Aktor / Remote Actor** ermöglicht

  Aktor – zu – Aktor – Kommunikation über Prozessgrenzen
- Entfernte Aktoren benötigen eine weitere Jar:

```
libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.5.2"
libraryDependencies += "com.typesafe.akka" %% "akka-remote" % "2.5.2"
```

Beispiel: Sender und Empfänger

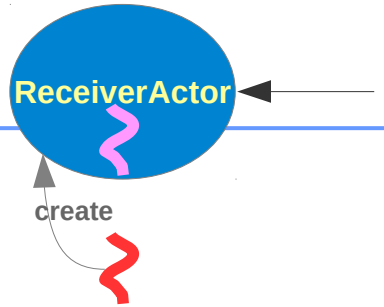




Beispiel: Empfänger

```
import akka.actor.{Actor, ActorSystem, Props}
import com.typesafe.config.{ConfigFactory, ConfigValueFactory}



class ReceiverActor extends Actor {
  def receive = {
    case msg: String =>
      println(s"Actor ${context.self} received $msg from $sender")
    case _ => println("Received non-string msg ")
  }
}
```

Dieser Aktor wird von einem entfernten Aktor angesprochen werden. Diese Tatsache ist für den Aktor selbst völlig transparent.

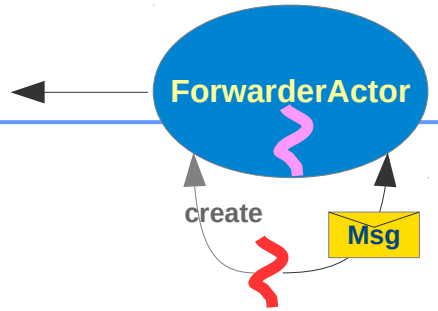


Beispiel: Empfänger

```
object Receiver_Main {

  val config = ConfigFactory.
    parseString(
      """akka {
    //loglevel = "DEBUG"
    actor {
       provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
      enabled-transports = ["akka.remote.netty.tcp"]
      netty.tcp {
            hostname = "127.0.0.1"
            port = 4711
      }
      log-sent-messages = on
      log-received-messages = on
    }
    }""")

  val system = ActorSystem("DistSystem",config)
  val receiverActor = system.actorOf(Props[ReceiverActor], name = "receiverActor")

  def main(args: Array[String]): Unit = {
    println(s" ${receiverActor.path} is ready")
  }

}
```

Der Empfänger ist ein eigenständiger Prozess

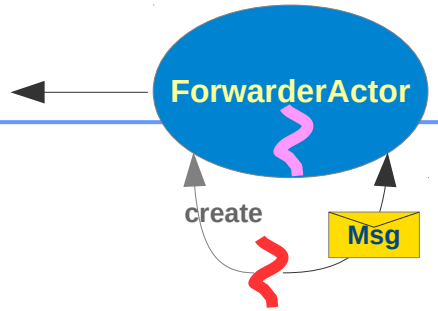
Das Aktorsystem muss konfiguriert werden, um entfernte Aktoren nutzen zu können. Die Konfiguration wird in der Regel in einer Konfigurationsdatei erfolgen. Hier wird sie der Übersicht halber programmatisch realisiert.



Beispiel: Sender – 1

```
import akka.actor.{Actor, ActorSystem, ActorSelection, ActorRef, Props}
import akka.util.Timeout
import scala.concurrent.duration._
import scala.concurrent.{Await, Future}
import com.typesafe.config.{ConfigFactory, ConfigValueFactory}

case object SendCmd // local msg to forwarder


// forwarder forwards msg to remote receiver
class ForwarderActor(remoteReceiverActor: ActorRef) extends Actor {
  def receive = {
    case SendCmd => remoteReceiverActor ! "Hello from remote sender actor!"
  }
}
```

Dieser Aktor leitet die Nachrichten des Threads an den entfernten Aktor weiter. Die Tatsache, dass die Nachricht an einen entfernten Aktor geht, ist auch hier völlig irrelevant.



Beispiel: Sender – 2

```
object Sender_Main {
  val config = ...

  implicit val timeout = Timeout(3 seconds)

  val system = ActorSystem("DistSystem", config)


  def main(args: Array[String]): Unit = {

    // create actor selection: a path to a remote actor
    // (actorFor is deprecated)
    val receiverActorS : ActorSelection =
                          system.actorSelection("akka.tcp://DistSystem@127.0.0.1:4711/user/receiverActor")
    println("receiverActor selection = " + receiverActorS)

    // resolve actor selection to actor path
    val futureActorR : Future[ActorRef] =  receiverActorS.resolveOne
    val serverActorR : ActorRef = Await.result(futureActorR , 2.seconds)
    println("Remote receiverActorR resolved to: " + serverActorR)

    // send to actor selection without resolving it to a actor ref
    // resolving will be done automatically
    receiverActorS ! "Greeting from sender"

    // create loacl actor via constructor with remote ActorRef parameter
    val forwarderActor = system.actorOf(
           Props(classOf[ForwarderActor], serverActorR),
           name = "remoteReceiverActor")

    // send to lacal actor
    forwarderActor ! SendCmd

  }

}
```

Der Sender ist ein eigenständiger Prozess

ActorSelection*: Referenz auf einen Unterbaum der Aktor-Hierarchie.*

resolveOne*: Suche die* ActorReference *die zur* ActorSelection *passt.*

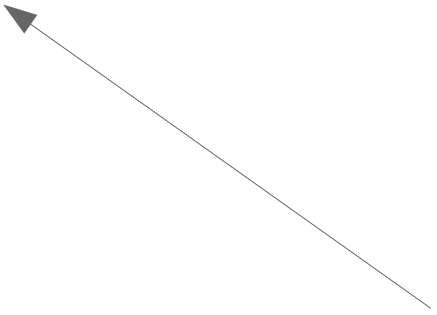
Sende direkt an eine ActorSelection

Erzeuge Forwarder übergib dabei Referenz auf entfernten Aktor

Beispiel: Sender – 3

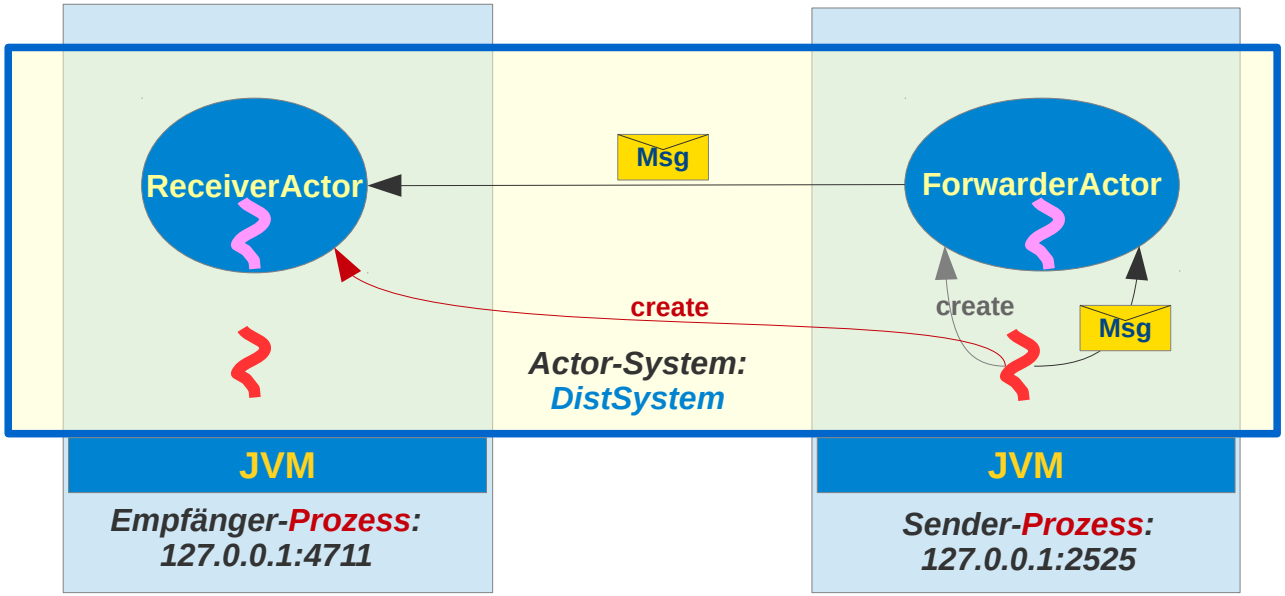
```
...

object Sender_Main {
  val config = ConfigFactory.
    parseString(
      """akka {
                //loglevel = "DEBUG"
                actor {
                   provider = "akka.remote.RemoteActorRefProvider"
                }
                remote {
                   enabled-transports = ["akka.remote.netty.tcp"]
                   log-sent-messages = on
                   log-received-messages = on
                   netty.tcp {
                       hostname = "127.0.0.1"
                       port = 2526
                   }
                }
             }""")

  implicit val timeout = Timeout(3 seconds)
...

}
```

Auch auf der Sendeseite muss die Verwendung entfernter Aktoren konfiguriert werden.



Beispiel: Entfernten Aktor erzeugen

Veranlasse die Erzeugung eines Aktors auf einem entfernten System (in einem anderen Prozess)



Beispiel: Entfernten Aktor erzeugen / Empfänger – 1

```
import akka.actor.{Actor, ActorSystem, Props}
import com.typesafe.config.{ConfigFactory, ConfigValueFactory}

class ReceiverActor extends Actor {
  def receive = {
    case msg: String =>
      println(s"Actor ${context.self} received $msg from $sender")
    case _ => println("Received non-string msg ")
  }
}
```

Empfänger-Aktor: Ein Aktor dieser Klasse wird von entfernt (von einem anderen Prozess) erzeugt werden.

Beispiel: Entfernten Aktor erzeugen / Empfänger – 2

```
object Receiver_Main {
  val config = ConfigFactory.
    parseString(
      """akka {
    //loglevel = "DEBUG"
    actor {
       provider = "akka.remote.RemoteActorRefProvider"
    }
    remote {
      enabled-transports = ["akka.remote.netty.tcp"]
      netty.tcp {
            hostname = "127.0.0.1"
            port = 4711
      }
      log-sent-messages = on
      log-received-messages = on
    }
    }""")
  val system = ActorSystem("DistSystem",config)

  //removed because actor will be created by remote process
  //val receiverActor = system.actorOf(Props[ReceiverActor], name = "receiverActor")

  def main(args: Array[String]): Unit = {
    println("receiver process is ready")
  }

}
```

Hier wird nur das Aktorsystem erzeugt und die Klasse der Aktoren bereit gestellt.

Beispiel: Entfernten Aktor erzeugen / Sender – 1

```
import scala.language.postfixOps
import akka.actor.{Actor, ActorSystem, ActorRef, Props, AddressFromURIString, Deploy}
import com.typesafe.config.ConfigFactory
import akka.remote.RemoteScope

case object SendCmd

class ForwarderActor(remoteReceiverActor: ActorRef) extends Actor {
  def receive = {
    case SendCmd => remoteReceiverActor ! "Hello from remote sender actor!"
  }
}
```

Beispiel: Entfernten Aktor erzeugen / Sender – 2

```
object Sender_Main {
  val config = ConfigFactory.
    parseString(
      """akka {
                //loglevel = "DEBUG"
                actor {
                   provider = "akka.remote.RemoteActorRefProvider"
                }
                remote {
                   enabled-transports = ["akka.remote.netty.tcp"]
                   log-sent-messages = on
                   log-received-messages = on
                   netty.tcp {
                       hostname = "127.0.0.1"
                       port = 2526
                   }
                }
             }""")

  val system = ActorSystem("DistSystem", config)

  val address = AddressFromURIString("akka.tcp://DistSystem@127.0.0.1:4711")
  val receiverActorR : ActorRef = system.actorOf(
     Props[ReceiverActor].withDeploy(Deploy(scope = RemoteScope(address))))

  // create local actor via constructor with remote ActorRef parameter
  val forwarderActor = system.actorOf(Props(classOf[ForwarderActor], receiverActorR),
                                      name = "remoteReceiverActor")

  def main(args: Array[String]): Unit = {
    // send to local actor
    forwarderActor ! SendCmd
  }

}
```

*Die Klasse **ReceiverActor** muss hier in diesem Prozess auf dem Klassenpfad zur Verfügung stehen und ebenfalls auf dem Klassenpfad der entfernten Anwendung auf der der Aktor erzeugt werden wird.*

Akka überträgt (im Gegensatz zu RMI) keinen Klassencode! - Zumindest nicht automatisch!

Erzeugung eines Aktors auf einem entfernten Prozess.

Aktoren und entfernte Kommunikation

Empfehlung:

Verteilte Anwendung komplett unter eigener Kontrolle

Anwendung ist aus technologischen Gründen (Effizienz- / Robustheits- / Lastverteilung) verteilt:

=> Aktorsystem mit entfernten Aktoren für die Anwendungs-interne Kommunikation

(Client-) Server-Anwendung

Eine Anwendung ist aus inhaltlichen Gründen verteilt:

=> NIO via akka.io für externe Kommunikation

Details und weitere Möglichkeiten können der Akka-Doku entnommen werden.