Nebenläufige und verteilte Programme CS2301

Th. Letschert

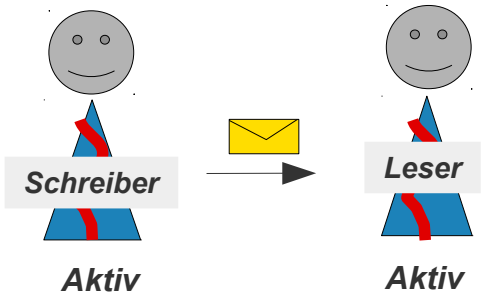
TH Mittelhessen Gießen

University of Applied Sciences

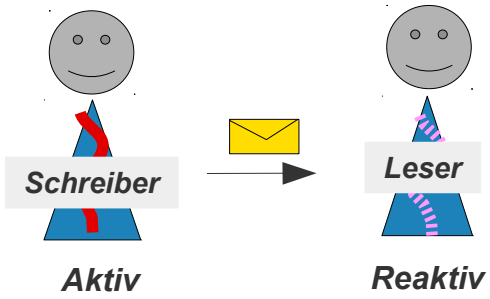
Verteilte Systeme

- Verteilte Systeme (Netze) mit aktiven, passiven und reaktiven Komponenten
- Produzent <~> Konsument: aktiv / passiv / reaktiv
- Leser-Schreiber: Push- / Pull- Netze
- Pipline mit reaktiven Filtern
- Monitore: passiv, aktiv, reaktiv
- Reaktive vs Kontrollorientierte Komponenten-Definition

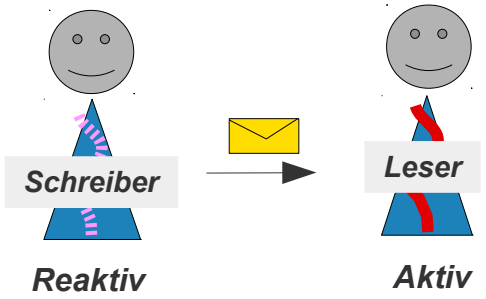
Schreiber / Produzent – Leser / Konsument



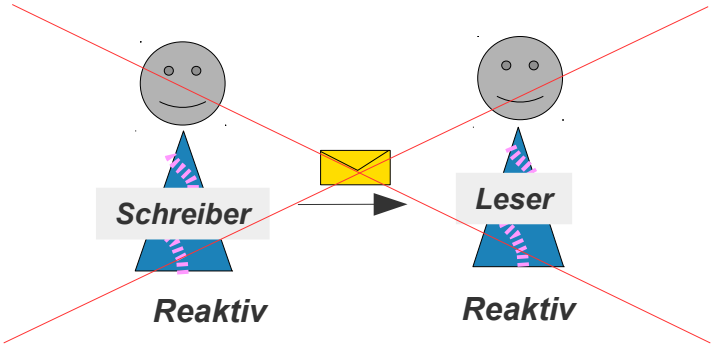
„Klassisch“: Produzent: Thread, Konsument: Thread



Push: Produzent: Thread, Konsument: Aktor



Pull: Produzent: Aktor, Konsument: Thread



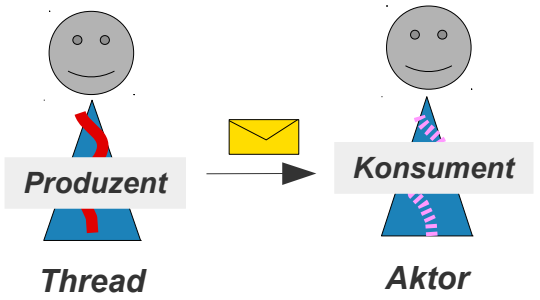
Unsinn: Produzent: Aktor, Konsument: Aktor

Push: Produzenten-Thread – Konsumenten-Aktor

ohne Flusskontrolle

Produzent sendet Daten ohne Rücksicht auf Konsument

Konsument arbeitet seine Mailbox ab, diese wird eventuell von Nachrichten überflutet



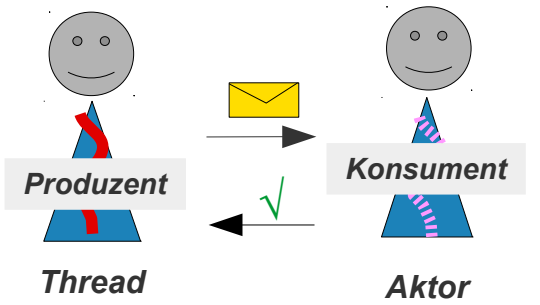
aktiver Produzent, reaktiver Konsument
Interaktion ohne Flusskontrolle

mit Flusskontrolle

Produzent und Konsument beachten Protokoll

z.B.: *Stop-and-Wait*-Protokoll:

- Produzent bestätigt jede Nachricht
- Konsument wartet nach jeder Nachricht eine Bestätigung ab.

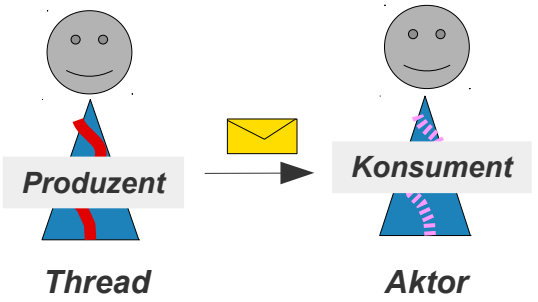


aktiver Produzent, reaktiver Konsument
Interaktion mit Flusskontrolle

Push: Produzenten-Thread – Konsumenten-Aktor

ohne Flusskontrolle

```
import akka.actor.{ ActorRef, Actor, ActorSystem, Props }

class Reader extends Actor {
  def receive = {
    case x: Int => {
      consume(x)
    }
  }
  def consume(v: Int) {
    println(s"Reader consumes $v")
  }
}

class Writer(start: Int, reader : ActorRef) extends Runnable {
  override def run {
    var i = start
    while (true) {
      reader ! i
      i = i+1
    }
  }
}

object ReaderWriter_Main extends App {
  val system = ActorSystem("RWSystem")
  val reader = system.actorOf(Props[Reader], name = "reader")
  val writer = new Writer(50, reader)
  new Thread(writer) . start
}
```

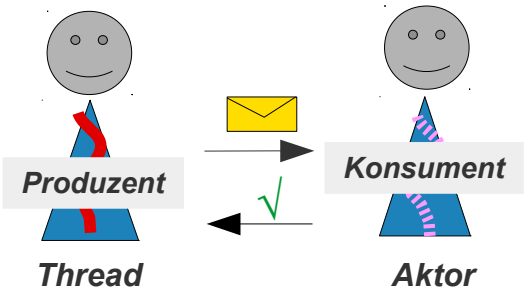


Produzent / Konsument

Push: Produzenten-Thread – Konsumenten-Aktor

mit Flusskontrolle

```
import akka.actor.{ActorRef, Actor, ActorSystem, Props}
import scala.concurrent.{ExecutionContext, Future, Await}
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.duration._

case object Ack

class Reader extends Actor {
  def receive = {
    case x: Int => {
      consume(x)
      sender ! Ack
    }
  }
  def consume(v: Int) {
    println(s"Reader consumes $v")
  }
}

class Writer(start: Int,
 reader : ActorRef,
 implicit val execContext: ExecutionContext)  extends Runnable {
  implicit val timeout = Timeout(3 seconds)
  override def run {
    var i = start
    while (true) {
      val f: Future[_] = (reader ? i)
      Await.result(f, 1.seconds)
      i = i+1
    }
  }
}
```

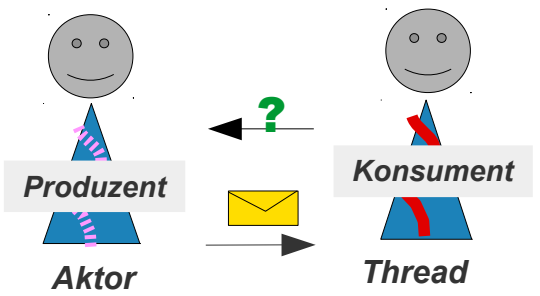



```
object ReaderWriter_Main extends App {
  val system = ActorSystem("RWSystem")
  val execContext = system.dispatcher

  val reader = system.actorOf(Props[Reader], name = "reader")
  val writer = new Writer(50, reader, execContext)

  new Thread(writer) . start
}
```

Pull: Produzenten-Aktor – Konsumenten-Thread

ohne Flusskontrolle – 1

```
import akka.actor.{ActorRef, Actor, ActorSystem, Props}
import scala.concurrent.{ExecutionContext, Future, Await}
import scala.concurrent.duration._
import scala.util.{Success, Failure}
import akka.pattern.ask
import akka.util.Timeout

case object Ask

class Reader(writer : ActorRef,
  implicit val execContext: ExecutionContext) extends Runnable {

  implicit val timeout = Timeout(3 seconds)

  override def run() {
    while (true) {
      val f: Future[_] = (writer ? Ask)
      consume(f.asInstanceOf[Future[Int]])
    }
  }
  def consume(f: Future[Int]) {
    f andThen {
      case Success(v) => println(s"reader received $v")
      case Failure(e) => println(s"reader failed with $e")
    }
  }
}
```



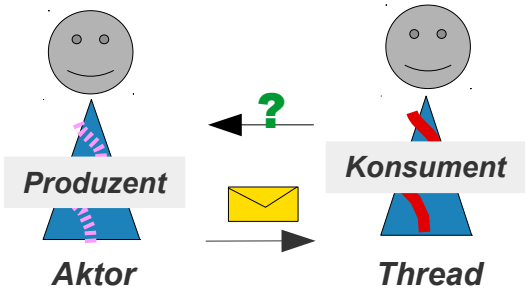
reaktiver Produzent, aktiver Konsument

***Asynchrone** Verarbeitung der Antwort des Schreibers => **Keine** Flusskontrolle !*

Pull: Produzenten-Aktor – Konsumenten-Thread

ohne Flusskontrolle – 2

```
class Writer(start: Int)  extends Actor {
  var i = start
  def receive = {
    case Ask => {
      sender ! i
      i = i+1
    }
  }
}
```



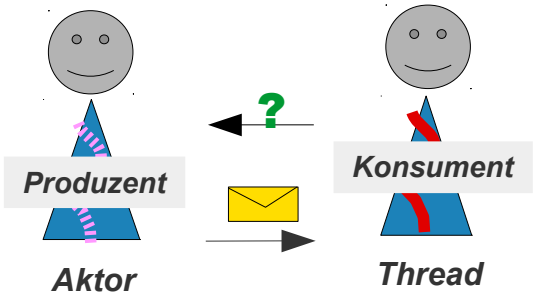
reaktiver Produzent, aktiver Konsument

```
object ReaderWriter_Main extends App {
  val system = ActorSystem("RWSystem")
  val execContext = system.dispatcher

  val writer = system.actorOf(Props(classOf[Writer],50), name = "writer")
  val reader = new Reader(writer, execContext)

  new Thread(reader) . start
}
```

Pull: Produzenten-Aktor – Konsumenten-Thread

mit Flusskontrolle – 1

```
import akka.actor.{ActorRef, Actor, ActorSystem, Props}
import scala.concurrent.{ExecutionContext, Future, Await}
import scala.concurrent.duration._
import akka.pattern.ask
import akka.util.Timeout

case object Ask

class Reader(writer : ActorRef,
  implicit val execContext: ExecutionContext) extends Runnable {

  implicit val timeout = Timeout(3 seconds)

  override def run() {
    while (true) {
      val f: Future[_] = (writer ? Ask)
      consume(Await.result(f.asInstanceOf[Future[Int]], 1.seconds))
    }
  }
  def consume(v: Int) {
    println(s"Reader consumes $v")
  }
}
```



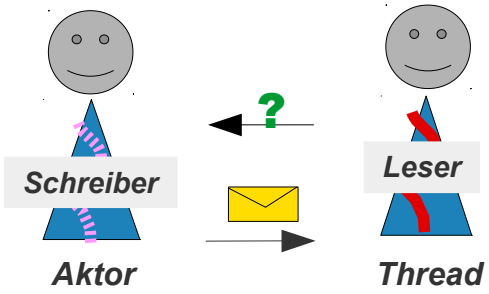
reaktiver Produzent, aktiver Konsument

***Synchrone** Verarbeitung der Antwort des Schreibers => Flusskontrolle !*

Pull: Produzenten-Aktor – Konsumenten-Thread

mit Flusskontrolle – 2

```
class Writer(start: Int)  extends Actor {
  var i = start
  def receive = {
    case Ask => {
      sender ! i
      i = i+1
    }
  }
}
```



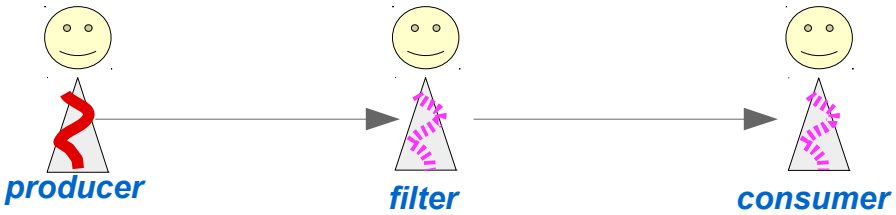
reaktiver Produzent, aktiver Konsument

```
object ReaderWriter_Main extends App {
    val system = ActorSystem("RWSystem")
  val execContext = system.dispatcher

  val writer = system.actorOf(Props(classOf[Writer],50), name = "writer")
  val reader = new Reader(writer, execContext)

  new Thread(reader) . start

}
```

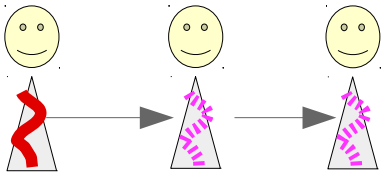
Pipeline (1)

```
object Pipeline_Main extends App {

  val system = ActorSystem("ProducerConsumer")
  val consumer = system.actorOf(Props[Consumer], name = "Consumer")
  val filter = system.actorOf(Props(classOf[Filter], consumer), name = "Filter")
  val producer = new Thread(new Producer(filter))

  producer.start()
  producer.join()

  val filter_stopped: Future[Boolean] = gracefulStop(filter, 5 seconds)
  val consumer_stopped: Future[Boolean] = gracefulStop(consumer, 5 seconds)
  Await.result(filter_stopped, 5 seconds)
  Await.result(consumer_stopped, 5 seconds)

  system.shutdown()
}
```

Filter-Pipeline mit freundlichem Herunterfahren



Pipeline (2)

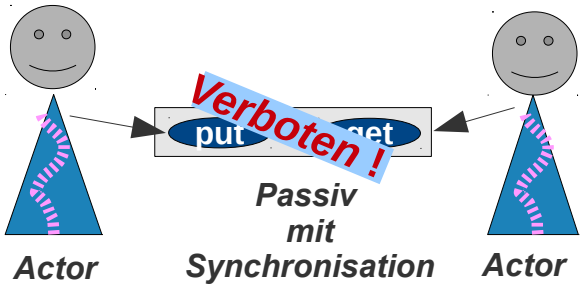


```
import akka.actor.{ActorRef, Actor, ActorSystem, Props}
import scala.util.Random
import scala.concurrent.{Await, Future}
import akka.util.Timeout
import scala.concurrent.duration._
import akka.pattern.gracefulStop

class Consumer extends Actor {
  def receive = {
    case x: Int => println("Consumer received " + x)
    case _ => println("Consumer received unkown msg ")
  }
}

class Filter(dest: ActorRef) extends Actor {
  def receive = {
    case -1 => {
    }
    case x: Int => {
      if (x%2 == 0) dest ! x
    }
    case _ => println("Filter received unkown msg ")
  }
}

class Producer(dest: ActorRef) extends Runnable {
  override def run() {
      for (i <- List(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21)) {
        Thread.sleep(Random.nextInt(100))
        dest ! i
      }
  }
}
```

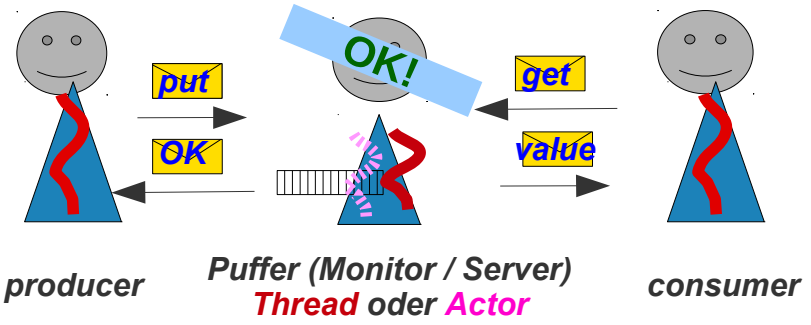
Aktor-Netze mit Monitoren

Passive Monitore mit synchronisierten blockierenden Methoden in Kombination mit reaktiven Komponenten sind verboten. Kein Passiver Puffer (synchronisierender Kanal):

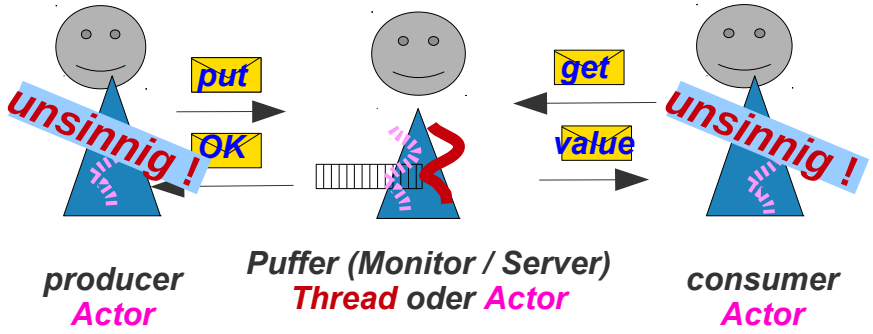
- der Produzent **darf** kein blockierendes Put ausführen
- der Konsument **darf** kein wartendes Take ausführen



Ein aktiver oder reaktiver Monitor, z.B. ein Puffer, kann zwei aktive Partner koppeln

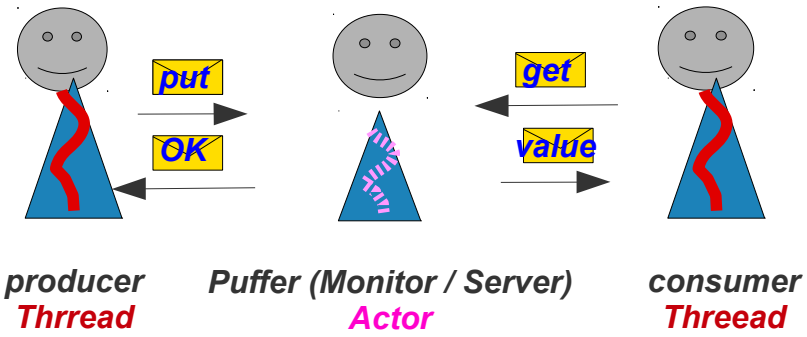


Reaktive Clients eines aktiven Monitors (Server) sind unsinnig! Ein (re)aktiver Server benötigt aktive Partner!



Aktor-Netze mit Monitoren

Beispiel reaktiver Puffer (1)



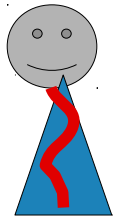

```
import akka.actor.{Actor, ActorSystem, ActorRef, Props}
import scala.util.Random
import scala.concurrent.{Await, Future}
import akka.util.Timeout
import scala.concurrent.duration._
import akka.pattern.{ask, gracefulStop, AskTimeoutException}

case class PutMsg(x: Int)
case class DataMsg(x: Int)
case object GetMsg
case object OKMsg
```

```
object ReactiveBuffer_Main extends App {
  val system = ActorSystem("ProducerConsumer")
  val buffer = system.actorOf(Props[ReactiveBuffer], name = "Buffer")

  object producer1 extends ActiveProducer(buffer) {
    val data = List(1,3,5,7,9,11,13,15,17,19,21)
  }
  object producer2 extends ActiveProducer(buffer) {
    val data = List(2,4,6,8,10,12,14,16,18,20,22)
  }

  val p1: Thread = new Thread(producer1)
  val p2: Thread = new Thread(producer2)
  val consumer: Thread = new Thread(new ActiveConsumer(buffer));

  consumer.start()
  p1.start();   p2.start()
  p1.join();    p2.join()

  consumer.interrupt()

  val buffer_stopped: Future[Boolean] = gracefulStop(buffer, 5 seconds)
  Await.result(buffer_stopped, 5 seconds)

  system.shutdown()
}
```

Aktor-Netze mit Monitoren

Beispiel reaktiver Puffer (2)

```
abstract class ActiveProducer(buffer: ActorRef) extends Runnable {

  val data : List[Int]

  implicit val timeout = Timeout(5 seconds)

  override def run() {
      for (i <- data) {
        Thread.sleep(Random.nextInt(100))
        val futureOK = buffer ? PutMsg(i)
        Await.result(futureOK, timeout.duration)
      }
  }
}
```

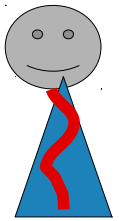


producer
Thread

Aktor-Netze mit Monitoren

Beispiel reaktiver Puffer (3)

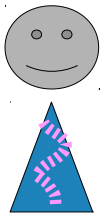
```
class ActiveConsumer(buffer: ActorRef) extends Runnable {

  implicit val timeout = Timeout(5 seconds)

  override def run() {
      while(! Thread.currentThread().isInterrupted()) {
        val futureDataMsg = buffer ? GetMsg
        try {
          val v = Await.result(futureDataMsg, timeout.duration)
          println("Consumer received " + v)
        } catch {
          case e: InterruptedException => println("consumer interrupted")
          case e: AskTimeoutException  => println("consumer timeOut")
        }
      }
  }
}
```



consumer
Thread

Aktor-Netze mit Monitoren

Beispiel reaktiver Puffer (4)



Puffer (Monitor-Server)
Actor

```
class ReactiveBuffer extends Actor {

  implicit val execContext = context.dispatcher

  object BufferMonitor {
    var item : Int = -999
    var empty : Boolean = true

    def put(x: Int) = synchronized {
      while (!empty) wait
      empty = false
      item = x
      notifyAll()
    }

    def get() : Int = synchronized {
      while (empty) wait
      empty=true
      notifyAll()
      item
    }
  }


}
```

```
def receive = {

    case PutMsg(x) =>
      val lSender = sender
      Future {
        BufferMonitor.put(x)
        lSender ! OKMsg
      }

    case GetMsg =>
      val lSender = sender
      Future {
        lSender ! DataMsg(BufferMonitor.get())
      }
    case x => println("Buffer received unkown msg " + x)
  }
```

Reaktiv: Nicht-blockierender Code da das Warten asynchron erfolgt!

Aktor-Netze mit Monitoren

Zusammenfassung

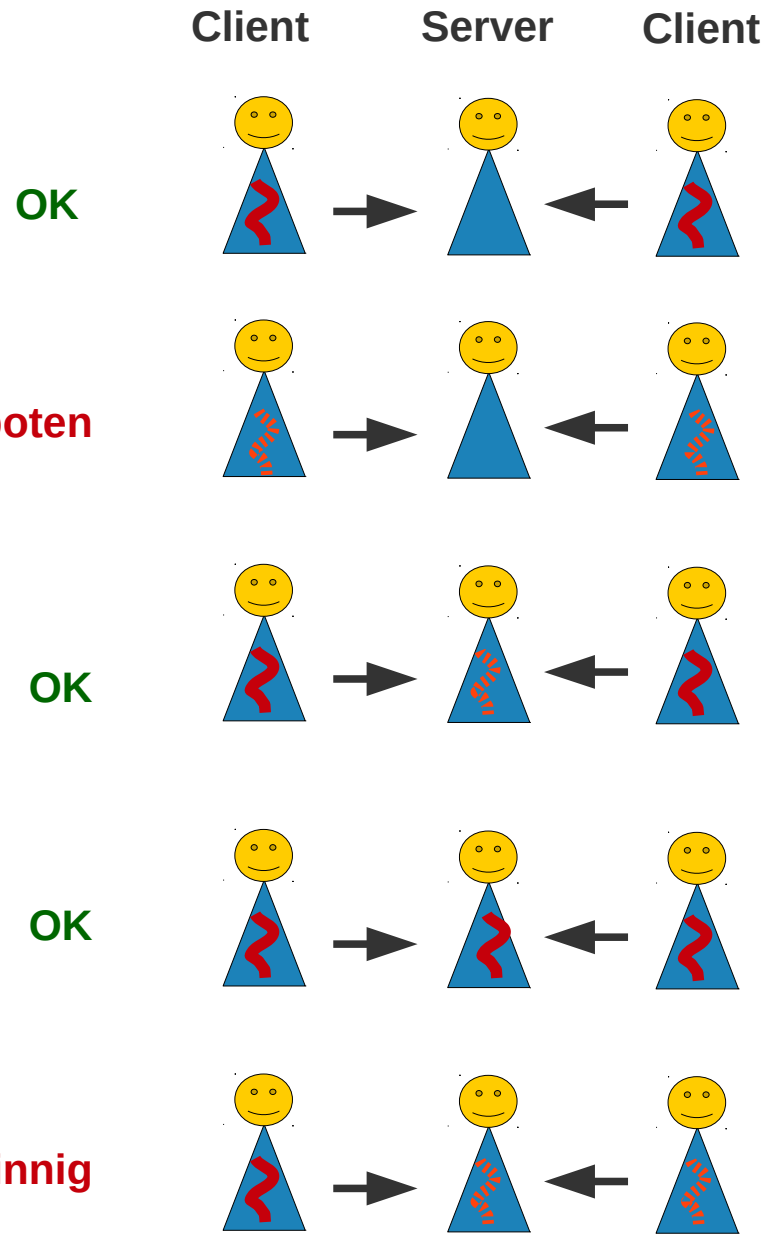
Client / Server Kombinationen

Ein Monitor synchronisiert die Kontrollflüsse aktiver Komponenten die in einer dritten Komponente zusammenstoßen

In einer passiven Komponente durch Synchronisation (Anhalten / reaktivieren des Kontrollflusses)

In einem Actor durch Verwaltung der eintreffenden Nachrichten und Senden eigener Nachrichten.

In einem Thread durch Verwaltung der eintreffenden Nachrichten und Senden eigener Nachrichten.
Ein eigener Kontrollfluss prinzipiell unnötig wenn auch gelegentlich vorteilhaft.



Verteilte Algorithmen

Verteilte Algorithmen werden mit folgenden Komponenten spezifiziert:

Agenten / Akteure

Ein verteilter Algorithmus besteht aus einer Menge von Agenten / Akteueren die über keine gemeinsamen Ressourcen verfügen, ausser dem Zugang zu einem Kommunikationsmedium.

Nachrichten

Die Akteuere interagieren ausschließlich über Nachrichten.

Topologie

Gegenseitige Bekanntschaften / Kommunikationskanäle o.Ä. erlauben es u.U. dass ein Akteur nur an eine Teilmenge anderer Akteure Nachrichten senden kann.

Akteuere in verteilten Algorithmen

Akteure in verteilten Systemen

Die Akteure in verteilten Systemen sind konzeptionell Prozesse, sie können und werden üblicherweise in einer von zwei möglichen Formen spezifiziert:

Kontroll-orientiert

Ein Akteur wird als sequentieller Prozess mit eingestreuten Sende- und Empfangs-Operationen angegeben.

Ereignis-orientiert

Ein Akteur wird als Zustandsautomat spezifiziert. Die eintreffenden Nachrichten sind die Ereignisse.

Kontroll- oder Ereignisorientierte Akteuere

Kontroll-orientierte vs. Ereignis-orientierte Spezifikationen

Die Kontroll-orientierte Spezifikation

- **ist gewohnter / einfacher**
- **Empfangsoperationen können beliebig verwendet werden.**

In einer Ereignis-orientierter Spezifikation muss

- **der Prozess um die Empfangs-Operation herum konstruiert werden und**
- **die Zustände müssen explizit verwaltet werden (Keine Warten / kein Stack)**

Es ist stets (mit einigem Aufwand) möglich eine Spezifikation in einer Form in eine äquivalente der anderen Form zu transformieren.

Kontroll- oder Ereignisorientierte Akteuere

Technische Umsetzung der Notationen

– Ein Prozess in kontroll-orientierter Notation kann in Empfangs-Operationen an jedem beliebigen Punkt blockieren

  z.B. innerhalb einer Aufruf-Hierarchie von Funktionen,

  er muss darum als Thread/Prozess (mit Stack!) teuer realisiert werden

– Ein Prozess in Ereignis-orientierter Notation kann nur an einem Punkt in Empfangs-Operationen blockieren.

  In diesem Punkt ist er garantiert nicht innerhalb innerhalb einer Aufruf-Hierarchie von Funktionen,

  er muss darum nicht als Thread/Prozess (mit Stack!) teuer realisiert werden

– Die ereignisorientierte Notation führt darum
  - zu einem „leichtgewichtigen Thread“, d.h.
  - zu einer reaktiven Komponente

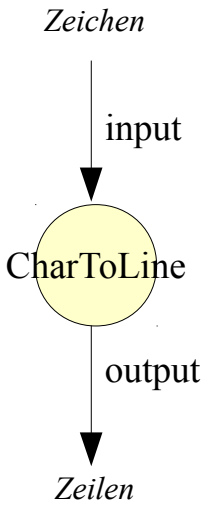
  die als Aktor realisiert werden kann.

Beispiel: Kontroll-orientiert

Char-to-Line (Pseudocode)

Wandle Zeichenfolgen mit CR-Zeichen um in Zeilen

```
channel input( char ), output( char[ 0..MAXLINE ] );

process CharToLine:
    char[] line; int i=0;
    do
    true : receive input( line[i] ) -> {
            do
            line[i] != CR && i < MAXLINE-1 -> {
                    i++;
                    receive input( line[i] );
                }
            od
            send output(line);
            i = 0;
        }
    od
```



Kontroll- vs Ereignis-orientiert

Char-to-Line Kontroll-orientierte Implementierung

ArrayBlockingQueue als Kommunikationskanal.

Nicht reaktiv: Der Code enthält eingestreute blockierende (Empfangs-) Operationen.

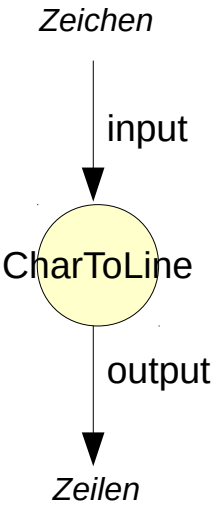
```
abstract class CharToLine extends Thread {
  val input:  ArrayBlockingQueue[Option[Char]]
  val output: ArrayBlockingQueue[Option[List[Char]]]

  val MAXLINE = 10
  val line = new Array[Char](10)
  var i = 0

  override def run(): Unit = {
    var continue = true
    while (continue) {
      input.take match {
        case None => continue = false; output.put(Some(line.slice(0,i).toList)); output.put(None)
        case Some(c) =>
          line(i) = c
          while (continue && line(i) != '|' && i < MAXLINE-1) {
           i = i+1
           input.take match {
             case None => continue = false; output.put(None)
             case Some(c) => line(i) = c
           }
       }
       output.put(Some(line.slice(0,i).toList))
       i = 0;
      }
    }
  }
}
```

Beispiel: Ereignis-orientiert

Char-to-Line (Pseudocode)

Wandle Zeichenfolgen mit CR-Zeichen um in Zeilen

```
channel input( char ), output( char[ 0..MAXLINE ] );

process CharToLine:

  char[] line; int i=0;

  // Ereignis: Empfang von CR
  receive input( CR ) -> {  send output(line);  }

  // Ereignis: Empfang eines beliebigen anderen Zeichens
  receive input( c ) -> {
      line[i] = c ;
      if  i < MAXLINE -> i++;
      else
            send output(line);
            i = 0;
  }
```



Kontroll- vs Ereignis-orientiert

Char-to-Line Ereignis-orientierte Implementierung als Thread

```
abstract class CharToLine extends Thread {
  val input:  ArrayBlockingQueue[Option[Char]]
  val output: ArrayBlockingQueue[Option[List[Char]]]

  val MAXLINE = 10
  val line = new Array[Char](10)
  var i = 0

  override def run(): Unit = {
    var continue = true
    while (continue) {
      input.take match {
        case None =>
                continue = false;
                output.put(Some(line.slice(0,i).toList));
                output.put(None)
        case Some('|') =>
                i= 0;
                output.put(Some(line.slice(0,i).toList))
        case Some(c) =>
          if(i < MAXLINE-1) {
            line(i) = c
            i = i+1
          } else {
            output.put(Some(line.slice(0,i).toList))
            i = i+1
          }
      }
    }
  }
}
```

ArrayBlockingQueue als Kommunikationskanal.

Reaktiv: Keine in den Code eingestreute blockierende (Empfangs-) Operation.

Implementiert als Thread, der um eine Empfangsschleife organisiert ist.

Kontroll- vs Ereignis-orientiert

Char-to-Line Ereignis-orientierte Implementierung als Actor

```
class CharToLine(output: ActorRef) extends Actor {

  val MAXLINE = 10
  val line = new Array[Char](10)
  var i = 0

  def receive = {

    case None =>
      output ! Some(line.slice(0,i).toList)
      output ! None

    case Some('|') =>
      i= 0;
      output ! Some(line.slice(0,i).toList)

    case Some(c) =>
      if (i < MAXLINE-1) {
         line(i) = c.asInstanceOf[Char]
         i = i+1
      } else {
         output !Some(line.slice(0,i).toList)
         i = i+1
      }
  }
}
```

Kontroll- vs Ereignis-orientiert

Char-to-Line Ereignis-orientierte Implementierung als Actor – Kontext

```
class CharToLine(output: ActorRef) extends Actor {
  ...
}
```

```
abstract class Producer extends Thread {
  val out: ActorRef
  val data: List[Char]

  override def run(): Unit = {
    for (v <- data) {
      out ! Some(v)
    }
    out ! None
  }
}
```

```
class Consumer extends Actor {
  def receive = {
     case Some(v) => println("\t Consumer received " +v)
     case _ => println("Consumer finished")
  }
}
```

```
object CharToLIne_Main extends App {
  val system = ActorSystem("ChrToLine")
  val consumer = system.actorOf(Props[Consumer], name = "consumer")
  val charToLine = system.actorOf(Props(classOf[CharToLine], consumer), name = "charToLine")

  object producer extends Producer {
   val out = charToLine
   val data: List[Char] = "abcdefghijkl|m|nop||qrstuvwxyz".split("").toList.map(_.charAt(0))
  }

  producer.start()
}
```