Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

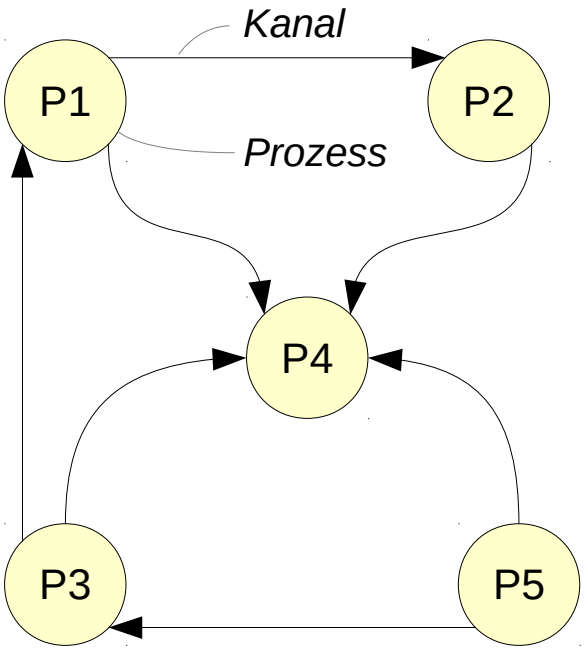
University of Applied Sciences

Prozessnetze, Pipes, Filter, aktive Monitore

- Prozessnetze: statisch / dynamisch
- Algorithmen in Prozessnetzen
  - Pipes und Filter
  - aktive Monitore

Prozesse

- sind **aktive Einheiten** die in blockierenden Anweisungen suspendiert werden können (**eigener Stack**!)
- haben ausschließlich Zugriff auf lokale Ressourcen (Speicher, ...)
- können diskrete Aktionen ausführen
  - Lokale Aktionen: interne Berechnungen
  - Nachrichten senden
  - Nachrichten empfangen

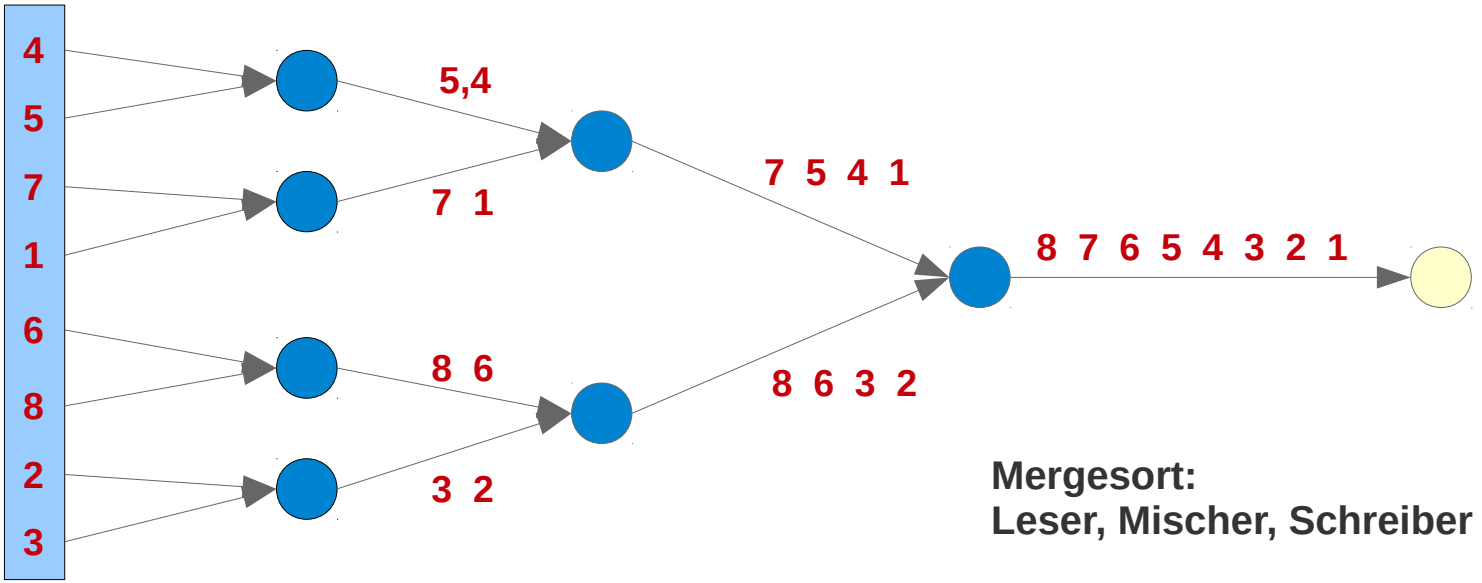


Prozessnetze

- sind verteilte Systeme
- aus aktiven Knoten: Prozesse / Threads
- mit einer bestimmten **Topologie** (Prozesse von bestimmten Typ / Anzahl / Verbindungen)
  - statische Topologie: fixe Mengen von Prozessen mit festen Beziehungen
  - dynamische: variable Mengen von Prozessen mit wechselnden Beziehungen
- Prozesse können sich Nachrichten über ein (irgendwie geartetes) Kommunikationssystem senden (oft als Kanal modelliert)
- Topologie: Welcher Prozess kann welchem anderen Nachrichten senden
- Beispiel: Kanal-basiertes Kommunikationssystem:
  Die Prozesse sind durch uni- oder bidirektionale Punkt-zu-Punkt Kanäle verbunden

Prozessnetze / Beispiele:

Schreiber-Leser



Mergesort:
Leser, Mischer, Schreiber

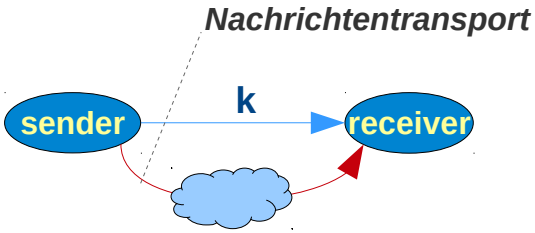
Kommunikation

Synchron / Asynchron

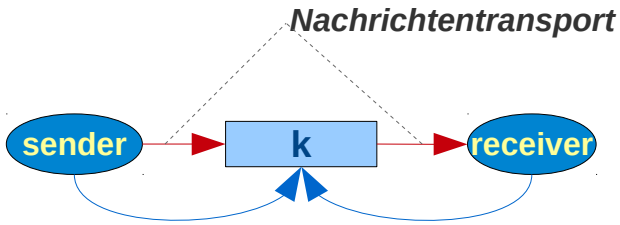
- **Synchron**: Sende- und Empfangs-Operation erfolgen gekoppelt / gleichzeitig
- **Asynchron**: Sende und und Empfangsoperation erfolgen unabhängig von einander (natürlich kann der Empfänger nur etwas Gesendetes empfangen)

Direkt / Indirekt

- **Direkt**: Die Prozesse senden sich Nachrichten direkt zu
- **Indirekt**: Die Prozesse kommunizieren indirekt über Kanäle / Puffer o.Ä.



Direkte Kommunikation



Indirekte Kommunikation

Statische und dynamische Prozessnetze

Statische Prozessnetze

Zur Laufzeit ist die Topologie des Netzes fix:

- Keine Erzeugung neuer Knoten (Prozesse)
- Keine Erzeugung neuer Kanäle / Prozessbekanntschaften

Statische Prozessnetze werden in drei Phasen erzeugt und ausgeführt:

- Definition von Prozessen (Knoten-Typen)
- Definition der Topologie: Knoten (Prozessinstanzen) und ihre Verbindung
- Ausführung der Prozesse

Dynamische Prozessnetze

Die Topologie des Netzes ist während der Laufzeit nicht fix:

- Neue Knoten können erzeugt werden
- Kanäle / Prozessbekanntschaften können erzeugt und modifiziert werden

Die Ausführung dynamischer Prozessnetze kann nicht in Phasen unterteilt werden

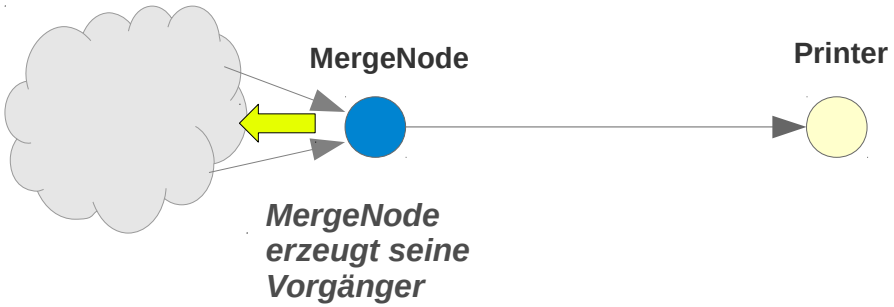
Dynamische Prozessnetze

Die Topologie des Netzes kann zur Laufzeit konstruiert / verändert werden

Beispiel:

Mergesort mit einem „selbst-entfaltenden“ Prozess-Netz – 1

```
object MergeSortDynamic_Main extends App {

  val lst = List(1, 3, 1, 9, 2, 8, 3, 7, 8, 7, 6, 5, 4, 3, 2, 1)

  val merger = new MergeNode(Printer.in, lst)

  Printer.start()

  new Thread(merger).start()

}
```



Dynamische Prozessnetze Mergesort mit einem „selbst-entfaltenden“ Prozess-Netz – 2

```
object Printer extends Thread {
  val in = new Channel[Int]

  override def run(): Unit = {
    var dataAvailable = true
    while (dataAvailable) {
       val v = try { in.receive } catch { case e: NoSuchElementException => dataAvailable= false; -1 }
       if (dataAvailable) {
         println("\t Printer received " +v)
       }
    }
    println("Printer finished")
  }
}
```

Dynamische Prozessnetze Mergesort mit einem „selbst-entfaltenden“ Prozess-Netz – 3

```
class MergeNode(out: Channel[Int], lst: List[Int]) extends Runnable {

  override def run(): Unit = {
    if (lst.length == 1) { out.send(lst(0)); out.close
    } else {
      val in_1 = new Channel[Int]
      val in_2 = new Channel[Int]
      val n = lst.length
      val pred_1 = new MergeNode(in_1, lst.take(n/2))
      val pred_2 = new MergeNode(in_2, lst.drop(n/2))

      new Thread(pred_1).start()
      new Thread(pred_2).start()

      var v1: Int = 0
      var v2: Int = 0
      var in1_EOF = false
      var in2_EOF = false

      v1 = try { in_1.receive } catch {
         case e: NoSuchElementException => in1_EOF = true; -1
        }
      v2 = try { in_2.receive } catch {
         case e: NoSuchElementException => in2_EOF = true; -1
        }
      while (!in1_EOF && !in2_EOF) { . . .}
      while (!in1_EOF) { . . . }
      while (!in2_EOF) { . . . }
      out.close
    }
  }
}
```

```
while (!in1_EOF && !in2_EOF) {
  if (v1 < v2) {
    out.send(v1);
    v1 = try { in_1.receive } catch {
      case e: NoSuchElementException =>
              in1_EOF = true; -1
    }
  } else {
    out.send(v2);
    v2 = try { in_2.receive } catch {
      case e: NoSuchElementException =>
              in2_EOF = true; -1
    }
  }
}
```

```
while (!in1_EOF) {
  out.send(v1);
  v1 = try { in_1.receive } catch {
     case e: NoSuchElementException =>
             in1_EOF = true; -1
  }
}
```

```
while (!in2_EOF) {
  out.send(v2);
  v2 = try { in_1.receive } catch {
     case e: NoSuchElementException =>
             in2_EOF = true; -1
  }
}
```

Dynamische Prozessnetze

Mergesort mit einem „selbst-entfaltenden“ Prozess-Netz – 4

```
import java.util.concurrent.ArrayBlockingQueue
import scala.collection.JavaConversions._

class Channel[T] {

  private val queueSize = 5
  private val q = new ArrayBlockingQueue[Option[T]](1)

  @volatile private var closed = false

  def send(x: T): Unit = q.synchronized {
    if (closed) throw new IllegalStateException
    q.put(Some(x))
  }

  def close: Unit = q.synchronized {
    closed = true
    q.put(None)
  }

  def isClosed: Boolean = closed

  def receive: T = {
    q.take match {
        case Some(x) => x
        case None => { closed = true; throw new NoSuchElementException }
    }
  }

}
```

Kommunikationskanäle verbinden Prozesse

Bei der Erzeugung der Topologie (des Netzes) werden sie mit Prozessinstanzen verknüpft.

```
class MergeNode(out: Channel[Int], lst: List[Int]) extends Runnable {
      ...
      val in_1 = new Channel[Int]
      val in_2 = new Channel[Int]
      ...
}
```

Statische Prozessnetze

Beispiel Mergesort

```
object MergeSortStatic_Main extends App {
  val c1 = new Channel[Int]
  val c2 = new Channel[Int]
  val c12 = new Channel[Int]
  val c12m = new Channel[Int]

  object producer1 extends Producer(3) {
    val out = c1
  }

  object producer2 extends Producer(1) {
    val out = c2
  }

  object merger extends Merger {
    val in_1 = c1
    val in_2 = c2
    val out = c12m
  }

  object printer extends Printer {
    val in = c12m
  }

  printer.start()
  merger.start()
  producer1.start()
  producer2.start()

}
```

Hier wird das Prozessnetz aufgebaut:

- *Knoten werden als Objekte definiert (Objekt ~ Modul)*
- *Die offenen Verbindungsstellen, die Kanäle (abstrakte Werte) werden mit konkreten Werten belegt.*

Prozesse als *Module*

Knoten (Prozesse) enthalten Kanäle als abstrakte Objekt-Variablen (*abstract fields*)

Im Kontext eines Netzes sind die Kanäle bekannt, mit denen sie belegt werden.

Werden Knoten zu Objekten instantiiert, dann werden die Kanäle konkrete Werte zugewiesen.

Statische Prozessnetze Beispiel Mergesort / Knoten mit „offenen Verbindungsstellen“

```
abstract class Printer extends Thread {
  val in: Channel[Int]

  override def run(): Unit = {
    var dataAvailable = true
    while (dataAvailable) {
       val v = try { in.receive } catch { case e: NoSuchElementException => dataAvailable = false; -1 }
       if (dataAvailable) {
         println("\t Printer received " +v)
       }
    }
    println("Printer finished")
  }
}

abstract class Producer(v: Int) extends Thread {
  val out: Channel[Int]

  override def run(): Unit = {
    println(s"producer sends $v")
    out.send(v)
    out.close
  }
}
```

Noch unverbundene Kanäle als abstrakte Objektvariablen

Statische Prozessnetze Beispiel Mergesort / Knoten mit „offenen Verbindungsstellen“

Noch unverbundene Kanäle als abstrakte Objektvariablen

```
abstract class Merger extends Thread {
  val out:  Channel[Int]
  val in_1: Channel[Int]
  val in_2: Channel[Int]

  override def run(): Unit = {
    var v1: Int = 0
    var v2: Int = 0
    var in1_EOF = false
    var in2_EOF = false

    v1 = try { in_1.receive } catch { case e: NoSuchElementException => in1_EOF = true; -1 }
    v2 = try { in_2.receive } catch { case e: NoSuchElementException => in2_EOF = true; -1 }

    while (!in1_EOF && !in2_EOF) {
      if (v1 < v2) {
        out.send(v1);
        v1 = try { in_1.receive } catch {
          case e: NoSuchElementException => in1_EOF = true; -1
        }
      } else {
        out.send(v2);
        v2 = try { in_2.receive } catch {
          case e: NoSuchElementException => in2_EOF = true; -1
        }
      }
    }
    while (!in1_EOF) {
      out.send(v1);
      v1 = try { in_1.receive } catch { case e: NoSuchElementException => in1_EOF = true; -1 }
    }
    while (!in2_EOF) {
      out.send(v2);
      v2 = try { in_2.receive } catch {
        case e: NoSuchElementException => in2_EOF = true; -1 }
     }
    out.close
  }
}
```

Datenfluss / *Dataflow Programming*

Datenfluss

- Architekturstil: Das System wird als eine Kombination von Datentransformatoren angesehen.
- Die Komponenten (Datentransformatoren) interagieren ausschließlich über den Austausch von Daten
- Varianten
  - **sequentielle Batch-Systeme**

    Eine Komponente beginnt ihre Arbeit, wenn die vorhergehenden ihre abgeschlossen haben
  - **Pipes und Filter**

    Die Komponenten arbeiten inkrementell: Daten werden nach Verfügbarkeit an die nachfolgenden Komponenten weiter geleitet
  - **Prozess-Kontrolle / Spreadsheet-Systeme**

    Die Komponenten repräsentieren Daten

    sie werden von werden von einem Prozess überwacht,

    der nach jeder Änderung einer Komponente alle abhängigen Komponenten neu berechnet

Datenfluss / *Dataflow Programming*

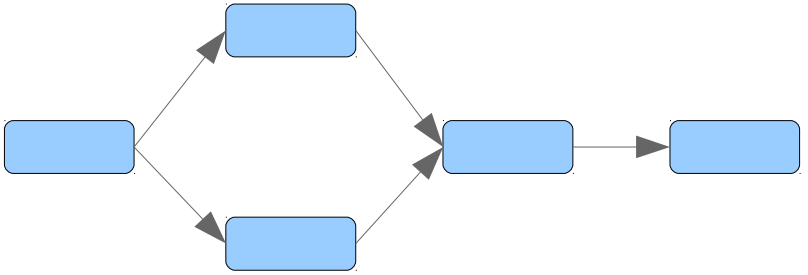
Strukturen von Datenfluss-Systemen

- Linear (Pipeline)
- in Stufen (keine Zykel)
- Beschränkte Zykel
- Beliebig

Pipeline

Visuelle Programmierung

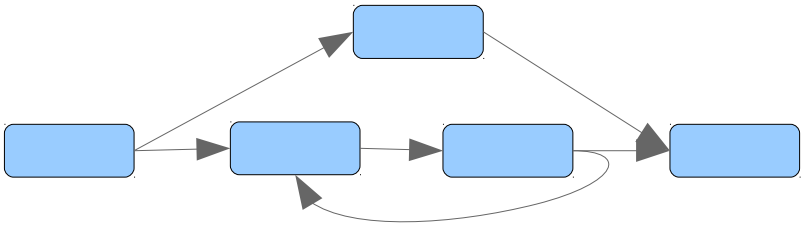
- Datenflusssysteme eignen sich gut zur visuellen Programmierung



gestuft

Funktionale Programmierung

- lineare, gestufte und beschränkt zyklischen Datenflusssysteme sind / entsprechen funktionalen Programmen
- Beschränkte Zykel können dabei mit Rekursion ausgedrückt werden



beschränkte Zykel

Pipes und Filter: Variante des Datenfluss-Stils / -Musters

Filter

Verarbeitungseinheiten

Transformieren die Daten ihrer Eingabeströme

Geben sie auf ihren Ausgabeströmen wieder aus

Varianten

- **aktive Filter**
  Filter ist aktiver Prozess / Thread:
  holt die Daten aus den Pipes und schreibt sie in Pipes
- **passive / reaktive Filter** (betrachten wir später etwas genauer)
  Push-Filter: wird von seinem Vorgänger getrieben
  pull-Filter: wird von seinem Nachfolger getrieben

Pipe

Transportiert Daten

ist (in der Regel) passiv

synchronisiert die aktiven Komponenten (Filter, Quellen, Senken)

Kann puffern:

- Puffer-Kapazität 0 : Synchrones Netz
- Puffer-Kapazität >0 oder unendlich: Asynchrones Netz

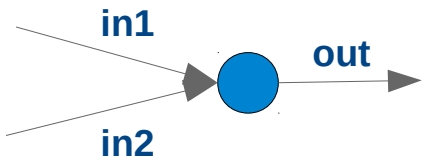
Quellen / Senken

Spezielle Verarbeitungseinheiten ohne Eingabe- oder Ausgabe-Ströme

Mergesort als Pipes und Filter

Mergesort

ist ein gestuftes Datenfluss-System in der Pipes-und-Filter-Variante



Monitor (Hoare, Brinch-Hansen, 1975)

Ein Monitor ist ein Modul / Objekt, in dem die von Prozessen gemeinsam genutzten Daten und ihre Zugriffsprozeduren (oder Methoden) zu einer Einheit zusammengeführt sind.

Monitor = Definition als Klasse mit Synchronisation

Kapselt Daten mit ihren synchronisierten Zugriffsmethoden

Synchronisation:

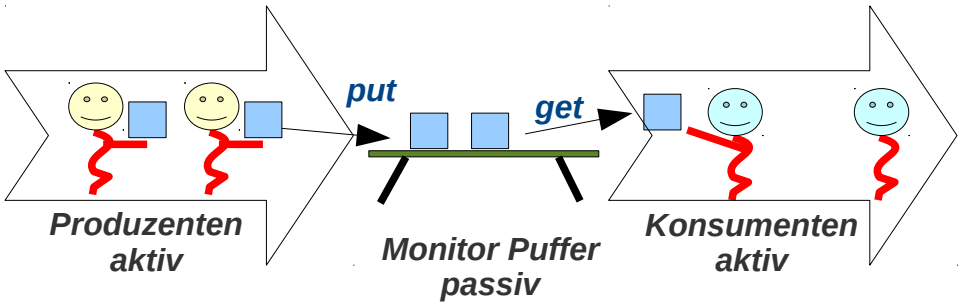
- **Gegenseitiger Ausschluss**
  Garantiert exklusiven Zugriff (Vermeide *race conditions)*
  Java: synchronized / Lock
- **Bedingungssynchronisation**
  Garantiert Zugriff unter den notwendigen Bedingungen (*conditional synchronisation*)
  Java: wait/notify, Condition

Monitore sind passive Konstrukte

Monitor basiert auf gemeinsamen Speicher
Aktive Einheiten (Prozesse) greifen auf passive Ressourcen zu,
die ihre Zugriffe synchronisieren

Monitor Beispiel Puffer

3 Zustände: leer / nicht voll, nicht leer / voll
Lesen: nur wenn nicht leer
Schreiben: nur wenn nicht voll



Zustand

count	~ Zahl der Elemente
leer	~ count > 0
nicht leer, nicht voll	~ 0 <count < SIZE
voll	~ count = SIZE

Vorbedingungen

put : nur wenn nicht voll
=> erwarte nicht voll

get: nur wenn nicht leer
=> erwarte nicht leer

put : *erwarte nicht voll*

```
while (!(count < SIZE))
  wait();
```

get: *erwarte nicht leer*

```
while (! (count > 0))
  wait();
```

+

put : *Benachrichtigen* bei Modifikation von *nicht leer*

```
count++;
notifyAll();
```

get: *Benachrichtigen* bei Modifikation von *nicht voll*

```
count--;
notifyAll();
```

Monitor Beispiel Puffer

Puffer mit *einem* Ablageplatz

Lesen: nur wenn nicht leer

Schreiben: nur wenn nicht voll

```
object ProducerConsumer_Main extends App {
  val buffer = new Buffer
   val producers = Array.tabulate(5){
    i => new Thread(new Producer(i, buffer))
   }
   val consumers = Array.fill(5){
    new Thread(new Consumer(buffer))
   }

   producers.foreach(_.start)
   consumers.foreach(_.start)

   producers.foreach(_.join)
   println("producers finished")

   Thread.sleep(100)
   System.exit(0)
}
```

In diesem Beispiel wird nur eine Bedingungsvariable verwendet.

```
class Buffer {
    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
    }
}
```

```
class Producer(id: Int, buffer : Buffer) extends Runnable {
  val items = List(1,2,3,4,5);
  override def run() {
      items.foreach( item => {
    buffer.put(10*id + item)
    Thread.sleep(500) } )
    }
}
```

```
class Consumer(buffer : Buffer) extends Runnable {
  override def run() {
      while (true) {
        val item = buffer.get
        println(s"consume $item")
      }
    }
  }
```

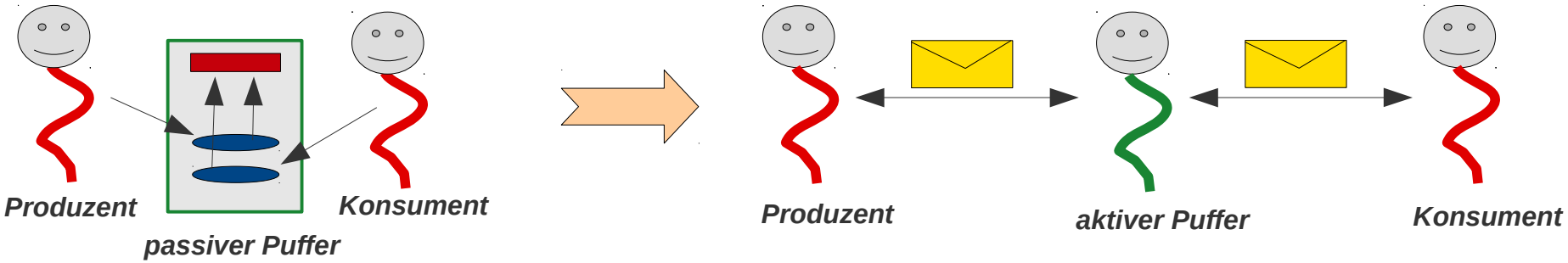
Vom Monitor zum Server: Puffer als aktiver Monitor

aktiver Puffer : Server

- als Nachrichten-verarbeitender aktiver Prozess: Server

Produzenten und Konsumenten: Clients

- Wollen / können keinen gemeinsamen Speicher nutzen, oder
- Nutzen als Clients den aktiven Puffer als Server



passive Ressource, aktive Nutzer:
Monitor-Programm

aktive Ressource, aktive Nutzer:
Client-Server-Programm

Vom Monitor zum Server: Puffer als aktiver Monitor

Gemeinsamer Zustand ist nicht möglich

Die Prozesse können nicht auf eine gemeinsame Ressource zugreifen.

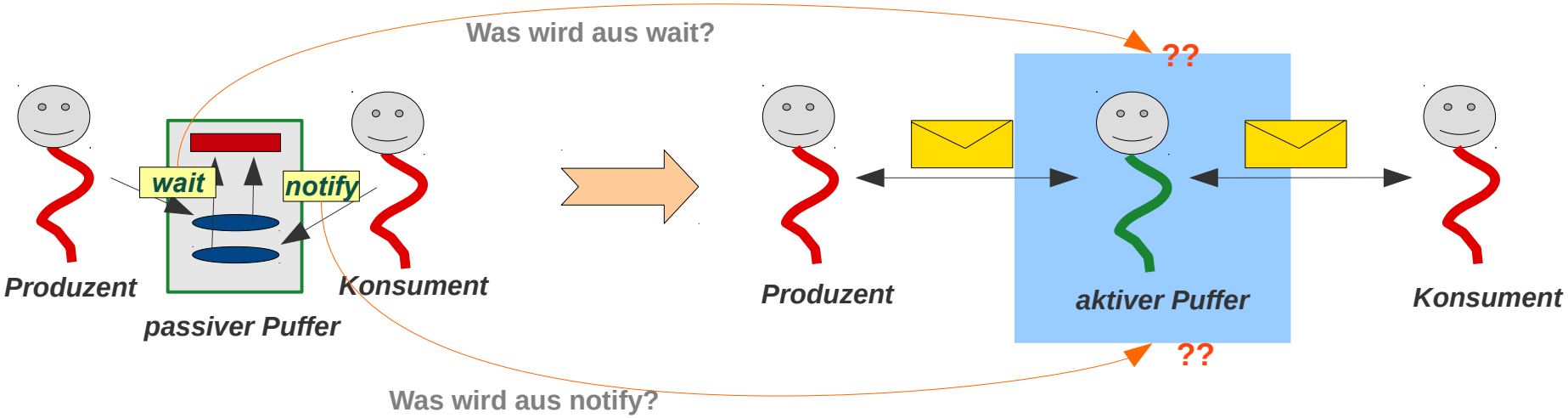
Die gemeinsame Ressource muss darum einen Server umgewandelt werden

- mit eigenem Kntrollfluss
- der die Aufträge als Nachrichten entgegen nimmt und bearbeitet.

Aktiver statt passiver Monitor: Kommunikation statt Synchronisation

Prozesse ohne gemeinsamen Zustand haben nur

- Synchronisationsanweisungen sind nicht mehr notwendig, da es keine gemeinsamen Ressourcen gibt: Gegenseitiger Ausschluss ist kein Thema mehr
- Die korrekte Reihenfolge der Verarbeitung muss aber weiterhin gewährleistet sein: Bedingungssynchronisation ist weiterhin ein Thema



Vom Monitor zum Server: Puffer als aktiver Monitor

Strategie der Umwandlung passiv => aktiv:

- *synchronisierter Aufruf* => Nachrichtenart, Empfang, Senden einer Antwort
- *wait* => Server Speichert Auftrag bis er ausführbar ist
- *notify* => Verarbeitung des gespeicherten Auftrags, Client-Benachrichtigung

```
class Buffer {
    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
    }
}
```

Verarbeite Nachrichten:
Put-Nachricht sende OK-Antwort
Get-Nachricht sende Daten-Antwort

Put-Nachricht mit ihrem Sender wird in WS gespeichert

verzögerte Get-Nachricht verarbeiten

Get-Nachricht mit Sender wird in WS gespeichert

verzögerte Put-Nachricht verarbeiten

Vom Monitor zum Server: Puffer als aktiver Monitor

```
process BufferServer {
  channel in[0..n-1];        channel out[0..n-1]
  Queue[Int,Int]pendingPuts;  Queue[Int] pendingGets;
  Token place;                boolean empty = true;

  do for i = 0 to n-1 do

    in[i].receive(msg) ->
    case msg {
     Put(item) =>
          if (!empty) {
               pendingPuts.enqueue(msg, sender)
          } else {
               place = msg.item;
               send OK to sender;
               empty = false;
               processPendingGet();
          }

     Get() =>
          if (empty) {
               pendingGets.enqueue(sender)
          } else {
               send Data(place) to sender;
               empty = true;
               processPendingPut();
          }
     }
  }
```

```
  proc processPendingGet() {
     if (! pendingGets.empty) {
       sender = pendingGets.remove()
       out[sender].send(place)
       empty = true;
       processPendingPut();
     }
  }

  proc processPendingPut() {
     if (! pendingPuts.empty) {
       (v, sender) = pendingGets.remove()
       out[sender].send(OK)
       place = v
       empty = false;
       processPendingGet();
     }
  }

}
```

aktiver Puffer, Pseudocode

Beispiel: Implementierung Puffer als aktiver Monitor

```
abstract class BufferServer extends Thread {

  val in: ArrayBlockingQueue[Msg]
  val out: Map[Int, ArrayBlockingQueue[Msg]]

  var place: Int = -99
  var empty: Boolean = true
  var pendingPuts: Queue[(Int,Int)] = Queue()
  var pendingGets: Queue[Int] = Queue()

  def processPendingGets(): Unit = {
    if (!pendingGets.isEmpty ) {
      val sender: Int = pendingGets.dequeue()
      out(sender).put(ValMsg(place))
      empty = true;
      processPendingPuts();
    }
  }

  def processPendingPuts(): Unit = {
    if (!pendingPuts.isEmpty) {
      val (sender, v) = pendingPuts.dequeue()
      out(sender).put(OkMsg)
      place = v
      empty = false;
      processPendingGets();
    }
  }


}
```

```
override def run(): Unit = {
  while (true) {
     val msg = in.take
     msg match {
        case (GetMsg(requester)) =>
          if(!empty) {
            out(requester).put(ValMsg(place))
            empty = true
            processPendingPuts()
          } else {
            pendingGets.enqueue(requester)
          }
        case PutMsg(requester, v) =>
          if(empty) {
            place = v
            out(requester).put(OkMsg)
            empty = false
            processPendingGets()
          } else {
            pendingPuts.enqueue((requester,v))
          }
     }
  }
}
```

Ressourcen-Allokation *(Resource Allocation)*

Allokations-Monitor

verwaltet eine Kollektion von Ressourcen beliebiger Art

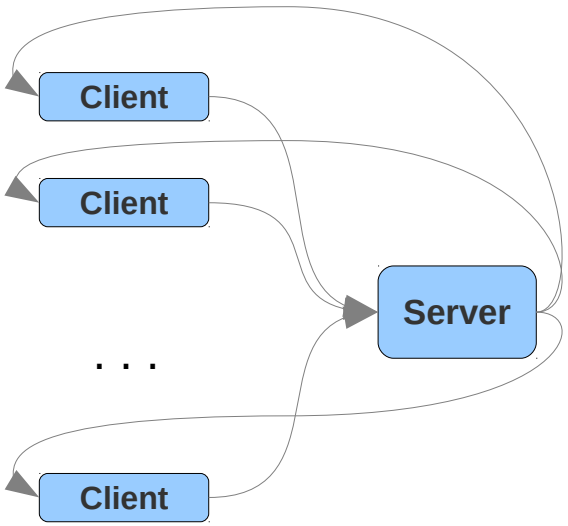
Nutzer fordern eine Ressource an, warten bis ihnen eine zugeteilt wird, nutzen sie, geben sie zurück.

```
monitor Allocator {
  var  avail = … // nr available units
  var  units = … // units
  cond free      // Condition variable

  def acquire(): Unit {
     if avail = 0 ~> free.wait
        avail > 0 ~> avail = avail-1
     fi
     return removeOne(units)
  }

  def release(u: Unit) {
     units.insert(u)
     avail = avail+1
     free.signal
  }

}
```

passiver Allocator, Pseudocode,

Ressourcen-Allokation *(Resource Allocation)*

Allokations-Server



aktiver Allocator, Pseudocode, kontrollorientierte Notation mit unidirektionalen Kanälen und Ports. Notation mit „Elipse“ (Auslassungspünktchen)

```
portIn request[0..n-1]   // attach n request-channels
potOut reply[0..n-1]          // attach n reply channels

process Allocator {
  var  avail = … // nr available units
  var  units = … // units
  var  pending : Queue<Msg>;

  do
     request[0].receive(AcquireMsg) ~>
          if avail > 0 ~>
               avail = avail-1
               reply[0].send(removeOne(units))
             avail = 0 ~>
               pending.add(0)

     request[0].receive(ReleaseMsg(u: Unit)) ~>
          if pending.size = 0 ~>
               avail = avail+1
               units.insert(u)
             pending.size > 0 ~>
               reply[pending.remove()].send(u)

     request[1].receive(AcquireMsg) ~>
          if avail > 0 ~>
               avail = avail-1
               reply[0].send(removeOne(units))
             avail = 0 ~>
               pending.add(1)
     . . .
  od
}
```

Ressourcen-Allokation *(Resource Allocation)*

Allokations-Server mit syntaktischer Schleife

aktiver Allocator, Pseudocode, kontrollorientierte Notation mit unidirektionalen Kanälen und Ports. Notation mit syntaktischer for-Schleife

```
channelIn request[0..n-1]  // attach n request-channels
channelOut reply[0..n-1]         // attach n reply channels

process Allocator {
  var  avail = … // nr available units
  var  units = … // units
  var  pending : Queue<Msg>;

  do for i = 0 to n-1 do
     request[i].receive(AcquireMsg) ~>
          if avail > 0 ~>
               avail = avail-1
               reply[i].send(removeOne(units))
            avail = 0 ~>
               pending.add(i)

     request[i].receive(ReleaseMsg(u: Unit)) ~>
          if pending.size = 0 ~>
               avail = avail+1
               units.insert(u)
            pending.size > 0 ~>
               reply[pending.remove()].send(u)

  od od
}
```

Leser-Schreiber

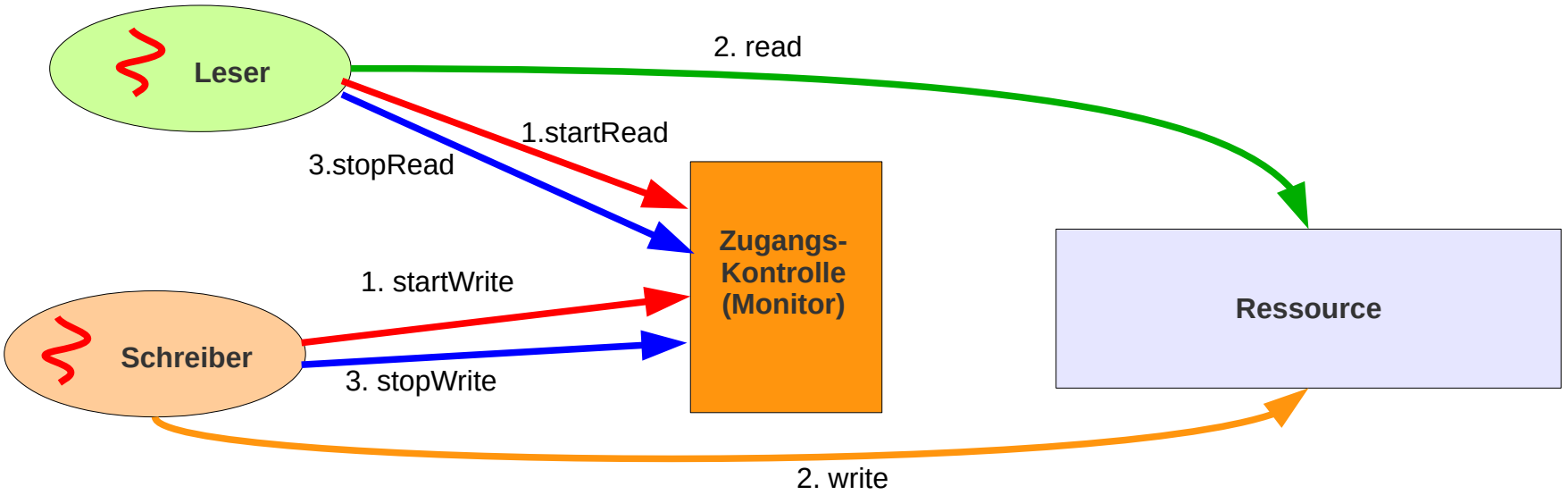
Leser-Schreiber Shared-Memory- / Monitor- Synchronisation

Eine Ressource wird mit teilweise nicht exklusiven gegenseitigem Ausschluss genutzt.

- **Leser dürfen gleichzeitig nutzen**
- **Schreiben müssen exklusiven Zutritt bekommen.**

Dies wird im *shared memory* Fall mit einer Zugangs-Kontrolle und einem Zugriffsprotokoll geregelt:

- **startRead / StartWrite : blockierende Monitor-Aufrufe**
- **read / write: unsynchronisierte eventuell überlappende Methoden der Ressource**
- **stopRead / StopWrite: Monitor-Aufrufe mit notify**



Leser-Schreiber

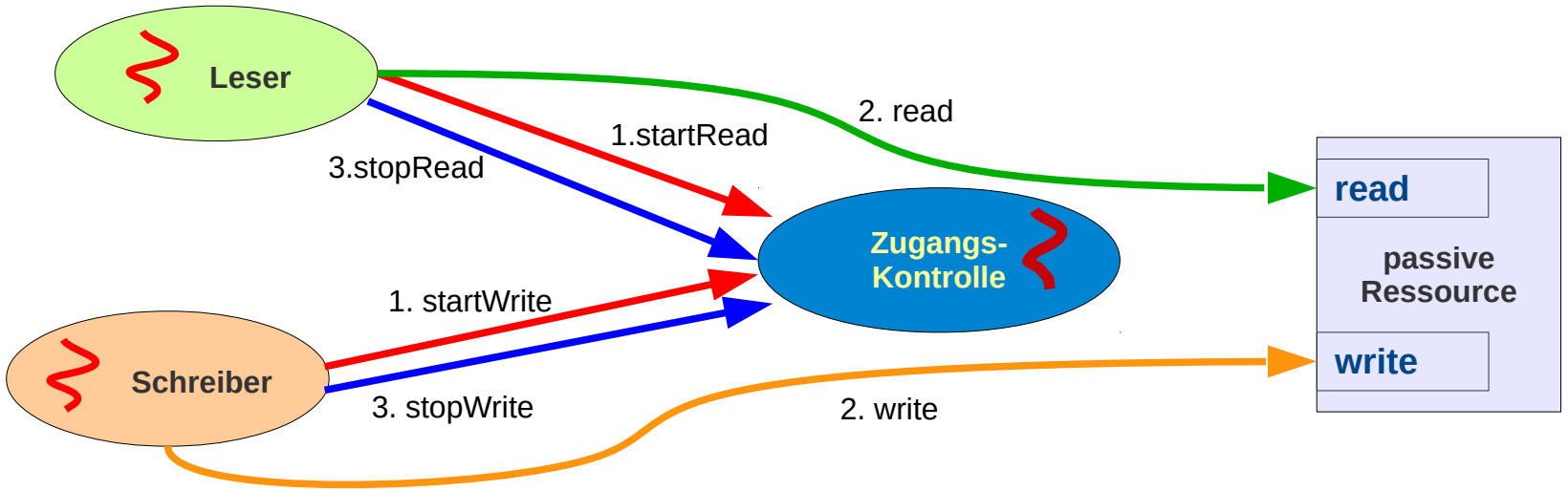
Leser-Schreiber mit aktiver Zugangskontrolle

Die kritischen Aktionen sind die zum Teil erlaubten gleichzeitigen Zugriffe auf die Ressource.

Deren Kontrolle soll von einer aktiven Komponente übernommen werden.

Die gleichzeitige Nutzung kann durch

- **durch die Threads der Kunden (passive Ressource) ausgeführt werden**



Szenario 1 : aktive Kunden, passive Ressource, aktive Zugangs-Kontrolle. Interaktionen nur durch Nachrichten und Methodenaufrufe.

Leser-Schreiber

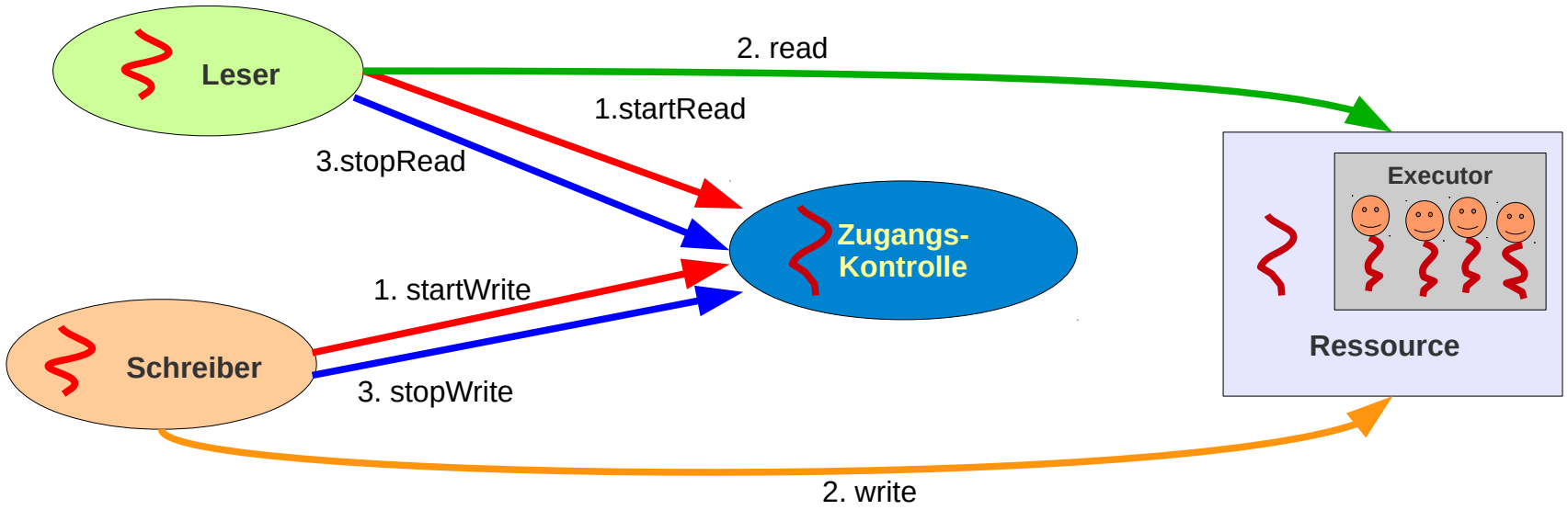
Leser-Schreiber mit aktiver Zugangskontrolle

Die kritischen Aktionen sind die zum Teil erlaubten gleichzeitigen Zugriffe auf die Ressource.

Deren Kontrolle soll von einer aktiven oder reaktiven Komponente übernommen werden.

Die gleichzeitige Nutzung kann durch

- **Threads der Ressource (aktive Ressource) ausgeführt werden**
  Beispielsweise mit Threads der Ressource in einem Executor



***Szenario 2:** aktive Kunden, aktive Zugangskontrolle, aktive Ressource (Threadpool + aktive Bearbeitung der Nachrichten), aktiver Monitor zur Zugangs-Kontrolle.*
Interaktionen nur durch Nachrichten.

Vom Monitor zum Server: Zusammenfassung

Monitor

aktiver Monitor ~> Server-Prozess

Monitor-Nutzer ~> Client-Prozesse

passiver => aktiver Monitor

Klasse => Prozess

Aufruf einer Monitor-Prozedur => Empfang einer Nachricht

Körper einer Monitor-Prozedur => Verarbeitung der Nachricht, Ergebnis an den Sender

Umsetzung Gegenseitiger Ausschluss

völlig unproblematisch:

Ein sequenzieller Serverprozess => streng sequenzieller Zugriff auf die Ressource

Umsetzung Bedingungssynchronisation

Der Serverprozess kann nicht (in wait) blockiert werden

wait ~> Speichern der Nachricht / ausbleibende Antwort blockiert Client

notify ~> Gelegenheit gespeicherte Nachrichten zu verarbeiten

Vom Monitor zum Server: Bemerkung

Natürlich kann man es sich auch etwas einfacher machen und im Server die Verwaltung der Warteschlangen Synchronisationsmechanismen überlassen.

Der Server enthält dann einfach den Monitor und startet für jede Anfrage einen Thread der auf dem Monitor operiert.

Beispiel PufferServer:

```
abstract class BufferServer extends Thread {

  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
      }
  }


}
```

```
val in: ArrayBlockingQueue[Msg]
val out: Map[Int, ArrayBlockingQueue[Msg]]

override def run(): Unit = {
  while (true) {
     val msg: Msg = in.take()
     msg match {
        case GetMsg(p) =>
          Future {
            val v = BufferMonitor.get()
            out(p).put(ValMsg(v))
          }

        case PutMsg(p, v) =>
          Future {
            BufferMonitor.put(v)
            out(p).put(OkMsg)
          }
     }
  }
}
```