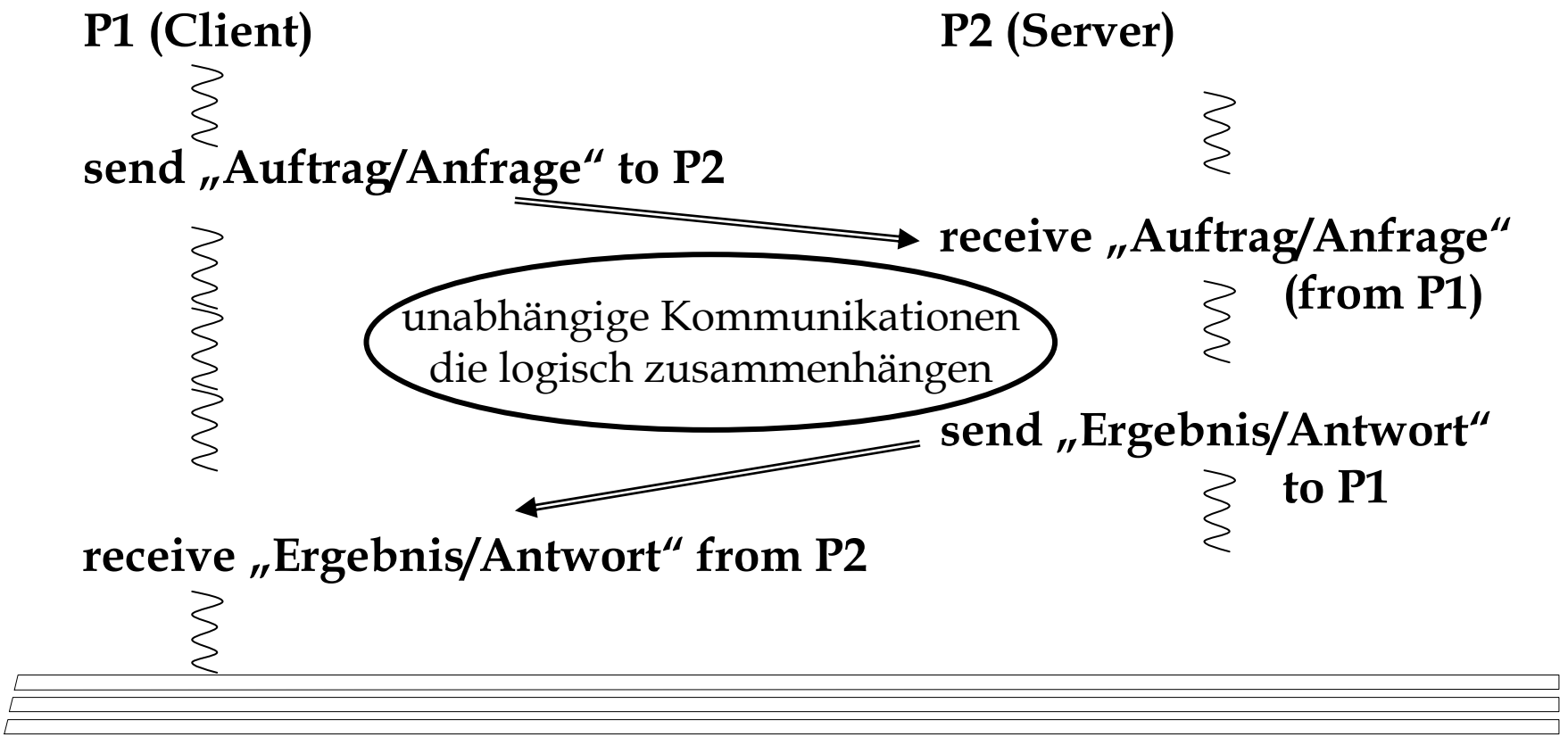
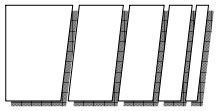
Abstraktere Kommunikationskonstrukte

- **send/receive gelten als „goto“ der parallelen Programmierung**
  => modelliere typische Kommunikationsmuster
  als abstraktere Konstrukte
- **Bsp:**



- **besser: Modellierung mit Remote Procedure Call oder Rendezvous**



Remote Procedure Call (RPC)

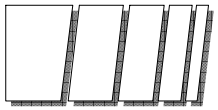
- **P1 (Client): call service (<input_args>, <result_args>)**

  |

  in MPD: proc

- **Aufruf einer Prozedur, die von einem anderen Prozess, der häufig eigens generiert wird, ausgeführt wird.**

- **impliziter Ablauf:**
  - **Senden der Eingabeargumente**
  - **Ausführen des Auftrags (durch eigenen Prozess)**
  - **Rücksenden der Ergebnisse**
  - **Zuweisen an Variablenparameter**



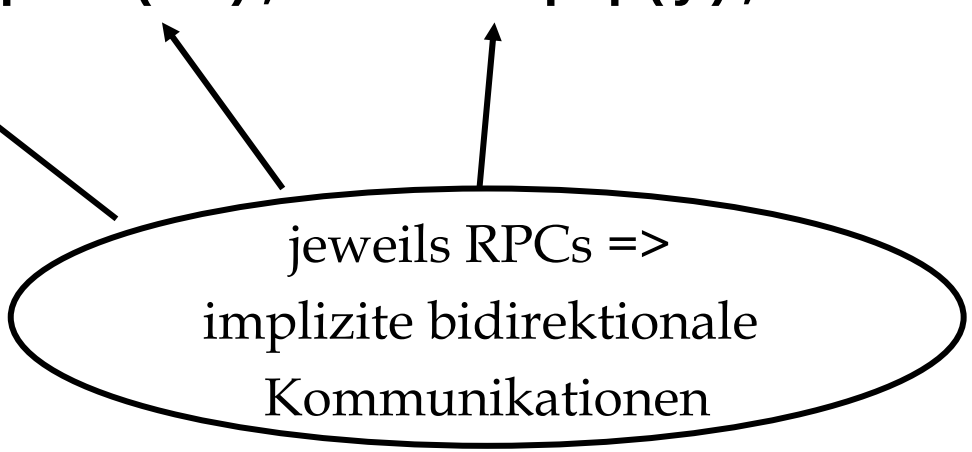
Beispiel: Stack Resource

(ggfs auf anderer virtueller/physikalischer Maschine)

```
resource Stack ()
   type result = enum(OK, OVERFLOW, UNDERFLOW)
   op push   (val int item)  returns result r
   op pop    (var int item)  returns result r
body Stack (int size)
   int store [1..size]
   int top = 0
   proc push (item) returns r {
     if (top < size) {store[++top] = item; r:= OK}
     else if (top == size) {r = OVERFLOW }
   }
   proc pop (item) returns r {
     if (top > 0 ) { .... }
end Stack
```

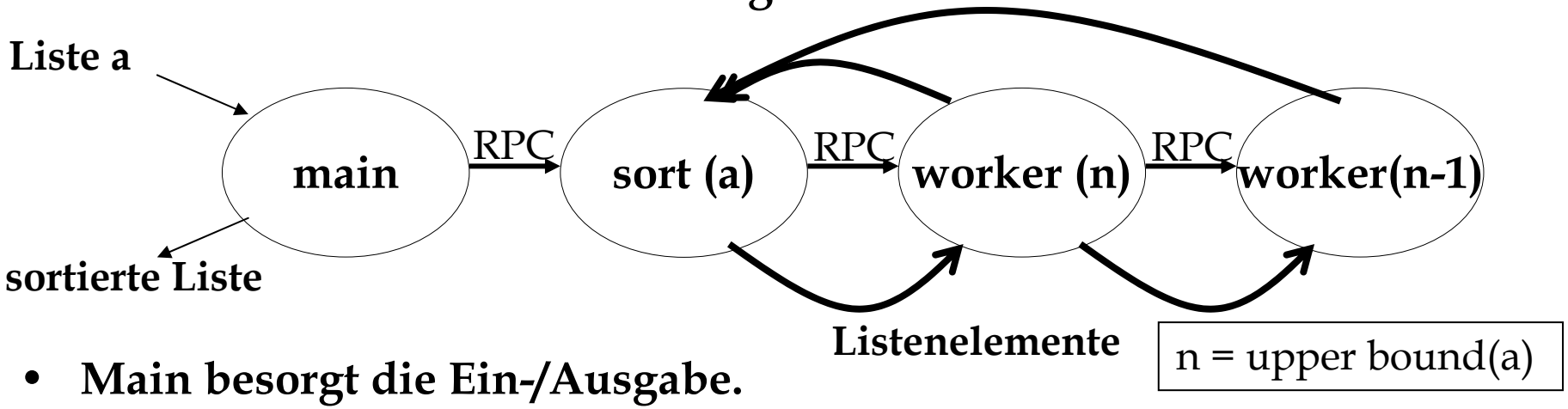
Beispiel: Stack User

```
resource Stack_User ()
   import Stack
   Stack.result x
   cap Stack s1, s2
   int y
   s1 = create Stack(10); s2 = create Stack(20)
   [call] s1.push(25); s2.push(15); r = s1.pop(y);
   if (r!= OK) { ... }
   ...
end Stack_User
```

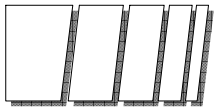


Beispiel: Dynamische Sortierpipeline

- **Methode: Sortieren durch Einfügen**



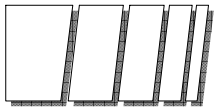
- **Main besorgt die Ein-/Ausgabe.**
- **sort initiiert den Aufbau der Pipeline mit worker-Prozessen und speist die Pipeline mit der zu sortierenden Liste.**
- **Jeder Worker-Prozess generiert nach Bedarf seinen Nachfolger, der als Ergebnis seinen Eingabekanal zurückliefert. Anschließend werden die Liste der zu sortierenden Werte gelesen, der kleinste erhaltene Wert gespeichert und größere Werte weitergeleitet.**
  Nach Erhalt von i Werten sendet worker(i) den gespeicherten Wert mit der Position i über einen globalen Antwortkanal an den sort-Prozess zurück.



MPD-Rückgabeanweisungen

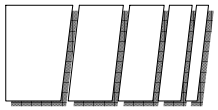
- **return:**
  - **Ende des Prozeduraufrufes**
  - **Rückgabe der Resultate**
  - **Variablenparameter**

- **reply**
  - **(vorzeitige) Kontroll- und Ergebnisrückgabe an aufrufenden Prozess**
  - **Fortsetzung der Prozedurbearbeitung**



Beispiel: Pipeline-Sortieren mit RPC

```
resource pipeline_sort()
  op print_array(int a[1:*]); op sort(var int a[1:*])
  op result(int pos, int value) {send}  # results
  optype pipe(int value) {send}         # values to sort
  op worker(int m) returns cap pipe pi {call}

  process main_routine {
    int n; writes("number of integers? "); read(n)
    int nums[1:n]; write("input integers, sep. by space")
    for [i = 1 to n] { read(nums[i]) }
    write("original numbers");  print_array(nums)
    sort(nums) # Aufbau der Sortierpipeline
    write("sorted numbers"); print_array(nums)
  }
```



Beispiel (Forts.): Sortieroperation

```
# Sort array a into non-decreasing order
proc sort(a) {
    if (ub(a) == 0) { return }
    cap pipe first_worker
    # Call worker; get back a capability for its pipe operation;
    # use the pipe to send all values in a to the worker.
  first_worker = worker(ub(a)-lb(a)+1)
  for [i = lb(a) to ub(a)] {   send first_worker(a[i])    }
  # Gather the results into the right places in a
  for [i = lb(a) to ub(a)] {
    int pos, value;   receive result(pos, value);  a[lb(a) + ub(a) - pos] = value
  }
 }
```

Beispiel (Forts.) Worker-Spezifikation

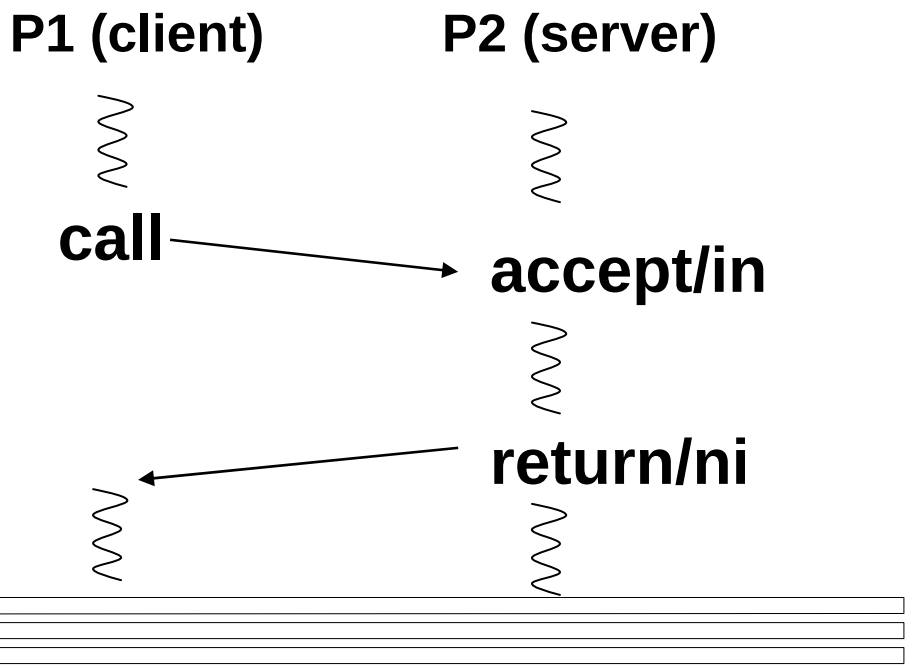
```
# Worker receives m integers on mypipe from its predecessor.
# It keeps smallest and sends others on to the next worker.
# After seeing all m integers, worker sends smallest to sort,
# together with the position (m) in array a where smallest is
# to be placed.
proc worker(m) returns pi {
  int smallest    # the smallest seen so far
  op pipe mypipe; pi = mypipe
  reply  # invoker now has a capability for mypipe
  receive mypipe(smallest)
  if (m > 1) {
      # create next instance of worker
      cap pipe next_worker # pipe to next worker
      next_worker = worker(m-1)
      for [i = m-1 downto 1] {
        int candidate; receive mypipe(candidate)
        # save new value if it is smallest so far; pass others
        if (candidate < smallest) { candidate :=: smallest }
        send next_worker(candidate)  }   }

  send result(m, smallest)  # send smallest to sort  }
```

Rendezvous

- **Bedienung des Clients durch bestehenden Prozess**
- **Vermeidung der Generierung eines separaten Prozesses**

  P2 (Server): accept service(input_pars,results_pars) -> body
  (in MPD in-Anweisung)
- **Verallgemeinerung synchroner Kommunikation:**

Ausführung von accept blockiert Server, bis entsprechende call Anweisung eines anderen Prozesses erfolgt.



MPD-Anweisung "in"

- **Syntax:**

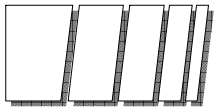
```
in  <op_command> [ ] <op_command> [ ]  ... ni
```

wobei `<op_command>` **die folgende Form hat:**

```
operation (<formal_id_list>) {returns <result_id>}
                                st <guard_expr> -> <block>
```

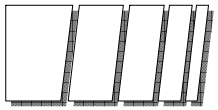
- **Ein Prozess, der eine in-Anweisung ausführt, wird suspendiert, bis eine der Operationen aufgerufen wird. Die Bedingungsausdrücke dürfen Operationsparameter referenzieren.**
- **Die receive-Anweisung ist ein Spezialfall der in-Anweisung:**

receive op(v1,v2) ist gleichbedeutend mit in op(p1,p2) ->
v1:= p1;v2:= p2 ni



Beispiel: Bounded Buffer in MPD

```
global buffer
    op enter(int); op remove() returns int
body buffer
const int n = 3; int daten [0:n-1] /* Puffer */
int anzahl = 0; int anfang = 0; int ende   = 0
process worker {
 while (true) {
    in enter(item) st (anzahl < n) ->
        daten[ende] = item; ende = (ende +1) mod n
        anzahl = anzahl +1
    [] remove() returns item st (anzahl > 0) ->
        item = daten[anfang]; anfang = (anfang +1) mod n
        anzahl = anzahl - 1
    ni
  }
}
end buffer
```



Beispiel: Dining Philosophers

Lösung mit zentralem Serverprozess (verklemmungsfrei, aber nicht fair)

```
resource Main()
import Philosopher, Servant
cap Servant s; s=create Servant(5)
for [i=1 to 5] {
   create Philosopher(s,i) }
end
```

```
resource Philosopher
import Servant
body Philosopher
   (s:cap Servant; id:int)
   process phil {
   while (true) {
     s.getforks(id) # eat
     s.relforks(id)   # think }
  }
end
```

```
resource Servant
  op getforks (id:int)
  op relforks (id:int)
body Servant (n:int)
  process server {
    bool eating[1:n] = ([n] false)
    while (true) {
      in getforks(id)
        st not eating[(id%n)+1]
        and not eating[((id-2)%n)+1]
        -> eating[id] := true
      [ ] relforks(id)
        -> eating[id] := false
      ni
    }
  }
end
```

Virtuelle Maschinen in MPD

- **Aufspaltung von Programmen in mehrere Adressräume, sog. virtuelle Maschinen**
- **dynamische Erzeugung:**

  cap vm c

  c := create vm( ) {on <expr>}

  erzeugt Verweis auf virtuelle Maschine

  optionale Plazierungs-information

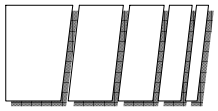
  physikalische Maschine als String oder Integer (installationsabhängig)

- **Platzierung auf spezifischen physikalischen Maschinen**
- **explizite Plazierung von Ressourcen auf virtuellen Maschinen:**

  create res_name(args) on c

  Verweis auf virtuelle Maschine

- **transparente Kommunikation zwischen Ressourcen auf verschiedenen virtuellen Maschinen**



Virtuelle Maschinen in MPD (Forts.)

- **Auf jeder virtuellen Maschine werden implizit eigene Instanzen importierter Globals erzeugt.**
- **Termination virtueller Maschinen mittels**

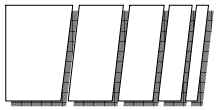
  destroy <expr>

  <expr> muss vom Typ cap vm sein.
  - **Termination aller Ressourcen mit Ausführung des `final code´**
  - **dann Termination aller Globals mit Ausführung des `final code´**
- **weitere vordefinierte Funktionen:**
  - **locate (n, hostname)**
    - **Assoziation von Nummer n mit Rechner hostname**
    - **Ändern der installationsabhängigen Default-Nummerierung**
    - **-> portable Programmierung**
  - **mymachine liefert Nummer der eigenen Maschine**
  - **myvm liefert Verweis auf eigene virtuelle Maschine**

Beispiel: test_vm.mpd

```
global glob
   int x= 0; sem mutex = 1
body glob
   final  {
      write(x)
   }
end
resource test(int N, int n, cap () signal)
   import glob
   process p [i=1 to N] {
      P(mutex); x+=n; V(mutex);
      send signal()
   }
end
```

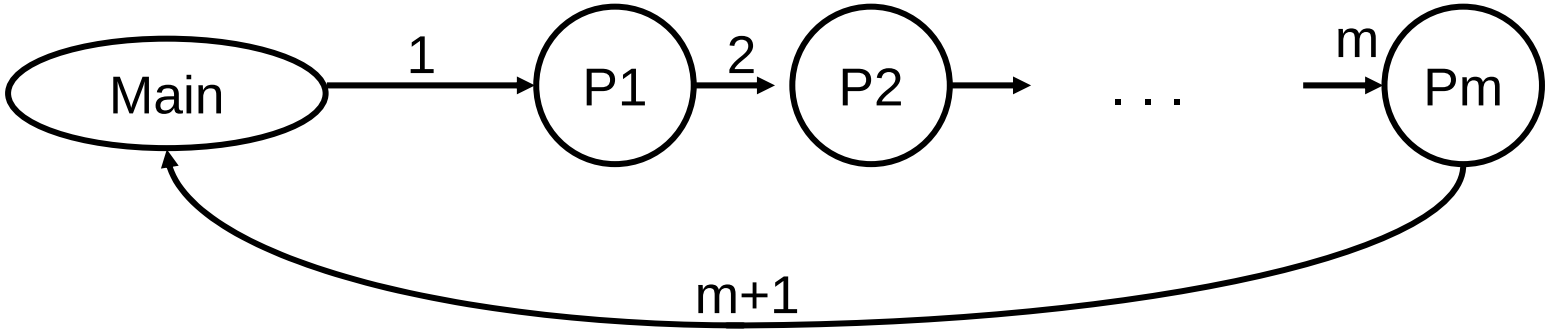
```
resource main()
     import test
     const int N = 5; op done()
     cap vm vmcap
     cap test t1, t2

     t1 = create test(N,1,done)
     vmcap = create vm()  on „tabora"
     t2 = create test(N,2,done)   on vmcap
     for [i=1 to 2*N] { receive done() }
     destroy t1; destroy t2; destroy vmcap
end
```



Beispiel: logischer Ring von VMs

```
const n = ... # Anzahl von Hosts
const m = ... # Anzahl von virtuellen Maschinen
const string[10]  hosts[n]  = ("tanga","annaba", ...)
cap vm vmcap[m]

for [i=1 to n] { locate(i,hosts[i]) }
for [i=1 to m] {
  vmcap[i] = create vm() on ((i-1) mod n)+1
  }
```

Beispiel: Sortieren durch Einfügen mit verteilter statischer Prozesspipeline



- **Main speist die Pipeline mit der zu sortierenden Liste.**
- **Pi speichert den kleinsten erhaltenen Wert und sendet m-i größere Werte weiter.**
- **Nach Erhalt von m+1-i Werten leitet Prozess Pi die sortierten nachfolgenden i-1 Werte weiter, hängt den eigenen Wert an und terminiert.**

MPD-Programm für Pipeline: tin.mpd

```
resource main()
import pipe
const int n = 4; const int m = 10
const string[10] hosts[n] =(„maseru",„tabora",„boende",„tabora")
cap vm vmcap[m]; int inp; op chan[1:m+1] (int); op ret (int)

for [i=1 to n] { locate(i,hosts[i]) }
for [i=1 to m] { vmcap[i] = create vm() on ((i-1) mod n)+1;
                 write(hosts[((i-1) mod n)+1]," bereit") }
write("Bitte Werte eingeben")
for [i=1 to m] { read(inp); send chan[1](inp) }
for [i=1 to m] {
       create pipe(i,m,chan[i],chan[i+1],ret) on vmcap[i] }
for [i=1 to m] { receive  chan[m+1](inp); writes(inp,"   ") }
# for [i=1 to m] { receive  ret(inp); writes(inp,"   ") }
write()

end main
```

Pipe-Resource: tin2.mpd

```
resource pipe(int i, int m, cap(int) inp, cap(int) out,
                                              cap(int) result)
int my_el; int value

process test {
   receive inp(my_el)
   for [j= 1 to m-i] {  receive inp(value);
      if (my_el <= value) { send out(value) }
      else                { send out(my_el); my_el=value }
      }
   write("Prozess ",i," ermittelte Element ",my_el)
   for [j = 1 to i-1] { receive inp(value); send out(value) }
send out(my_el)
# send result(my_el)
}
end
```