

Nachtrag zu Swapchannels

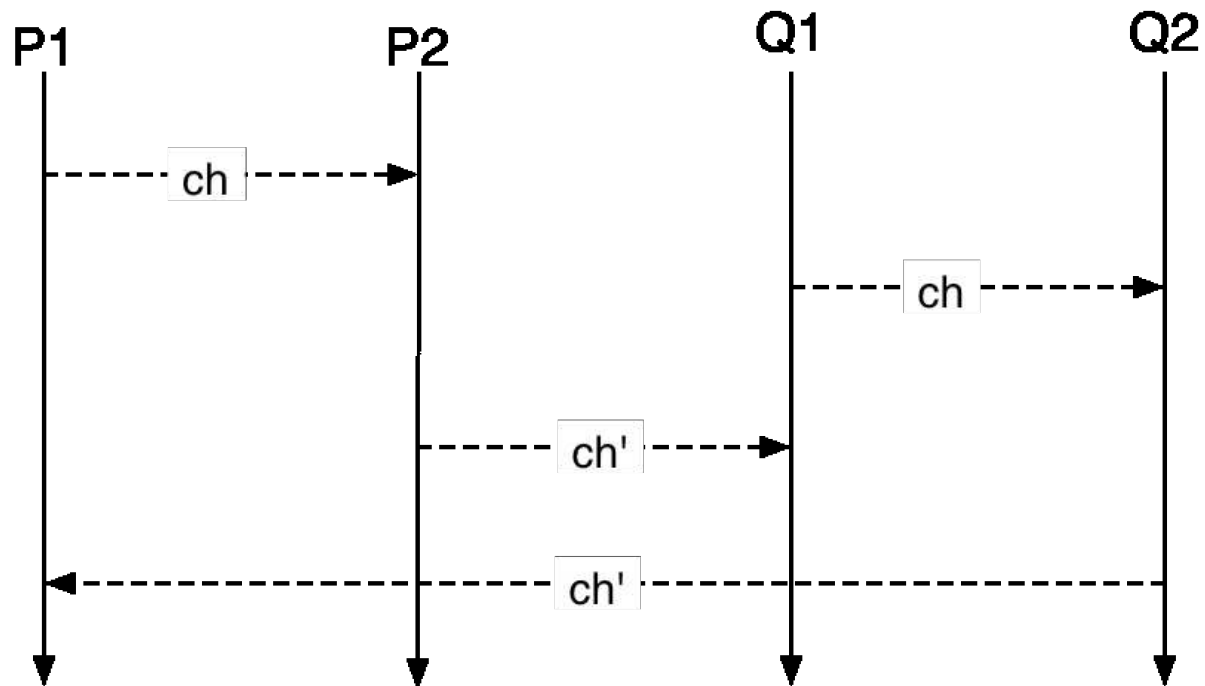
Problem war:

- Mehrere Thread-(Paare) kommunizieren über einen Swap-Channel
- Antwort könnte eventuell vom falschen Sender empfangen werden

Darum: Senden schickt Channel für Antwort mit, erzeugen ihn stets neu mit **guard**

Falsche Zuteilung bei Swap-Channel

Ein einzelner Antwortchannel reicht nicht aus:



Negative Bestätigung

choose wählt aus einer Liste von Rendezvous das erste aus, das synchronisiert werden kann

Problem: Die Kommunikationspartner der restlichen Rendezvous erfahren nicht, dass "ihre" Rendezvous nicht ausgewählt wurden

Tritt häufig bei Client-Server-Programmierung auf: Server muss wissen, ob Client noch mit ihm kommunizieren will

with-nack

`with-nack` erzeugt zu einem Rendezvous A ein Rendezvous B auf das synchronisiert werden kann, wenn A von `choose` nicht ausgewählt wird

`with-nack` nimmt als Argument eine Prozedur, die als Parameter Rendezvous B übergeben bekommt und Rendezvous A liefert

```
(with-nack (lambda (B)  
            ... A))
```

Beispiel

Gib für ein Rendezvous aus, ob `choose` es ausgewählt oder verworfen hat

```
(define (display-message-rv rv ack-message nack-message)
  (with-nack (lambda (nack)
    (spawn
      (lambda ()
        (sync nack)
        (display nack-message)
        (newline))))
    (wrap rv (lambda (ignore)
      (display ack-message)
      (newline))))))
```

Beispiel: select auf 2 Channel

```
(define ch1 (make-channel))  
  
(define ch2 (make-channel))  
  
(spawn  
  (lambda ()  
    (select (display-message-rv (receive-rv ch1) "ack 1" "nack 1")  
           (display-message-rv (receive-rv ch2) "ack 2" "nack 2")))))  
  
(send ch2 'a)
```

nack 1

ack 2

Größeres Beispiel

Erzeuge Liste von durchnummerierten Channels

```
(define (make-channel-list n)
  (map (lambda (n)
        (cons n (make-channel))))
      (numbers n)))
```

Versuche aus allen zu lesen

```
(define (wait-for-data channels)
  (spawn
   (lambda ()
     (let lp ()
       (apply select
        (map (lambda (n.ch)
              (display-message-rv
               (receive-rv (cdr n.ch))
               (ack-message (car n.ch))
               (nack-message (car n.ch))))))
        channels))
     (lp))))))
```

Beispiel starten

```
(define cs (make-channel-list 10))
```

```
(wait-for-data cs)
```

```
(send (cdr (car cs)) 25)
```

```
nack 2
```

```
nack 3
```

```
nack 4
```

```
nack 5
```

```
nack 6
```

```
nack 7
```

```
nack 8
```

```
nack 9
```

```
nack 10
```

```
ack 1
```

Beispiel: Lock-Server

Clients können beim Server Locks erhalten und freigeben

- Clients kommunizieren mit Server über einen Channel
- Clients warten ggf. bis sie Lock erhalten
- Server liefert dazu Rendezvous

Problem: Server muss mitbekommen, wenn ein Client anderes Rendezvous vorzieht

⇒ Server verwendet **with-nack** wenn er Client Lock zuteilt

Datentyp für Locks

Locks sind eigener Recordtyp mit eindeutiger ID

```
(define-record-type :lock
  (really-make-lock id)
  lock?
  (id lock-id))

(define make-lock
  (let ((id 0))
    (lambda ()
      (set! id (+ 1 id))
      (really-make-lock id))))
```

Nachricht zum Loslassen des Locks

Enthält nur die ID des freizugebenden Locks

```
(define-record-type :release-message  
  (make-release-message id)  
  release-message?  
  (id release-message-id))
```

Nachricht zum Holen des Locks

Enthält:

- ID des Locks
- Channel für die Antwort
- Abort-Rendezvous

```
(define-record-type :obtain-message
  (make-obtain-message id reply-channel abort-rv)
  obtain-message?
  (id obtain-message-id)
  (reply-channel obtain-message-reply-channel)
  (abort-rv obtain-message-abort-rv))
```

Request-Channel und Lock loslassen

Request-Channel ist globale Variable

- Sollte von außen nicht sichtbar sein

```
(define request-channel (make-channel))
```

Release-Nachricht erzeugen und synchron verschicken:

```
(define (release-lock lock)
  (send request-channel
        (make-release-message (lock-id lock))))
```

Lock holen

Nachricht asynchron an Server schicken

Aus Reply-Channel lesen

Mit `with-nack` Rendezvous erzeugen, das eintritt, wenn Lesen verworfen wird

```
(define (obtain-lock-rv lock)
  (with-nack
    (lambda (nack)
      (let ((reply-channel (make-channel)))
        (spawn
          (lambda ()
            (send request-channel
                  (make-obtain-message (lock-id lock)
                                       reply-channel
                                       nack))))))
      (receive-rv reply-channel))))))
```

Lock-Server

- Thread mit Endlosschleife
- `lock-alist` enthält für geholte Locks Informationen über wartende Threads
- `lock-alist` ist Liste aus Paaren von IDs und Informationen

```
(define (start-lock-server!)  
  (spawn  
    (lambda ()  
      (let serve ((lock-alist '()))  
        (let ((request (receive request-channel)))  
          (cond  
            ...))))))
```

obtain-message

- Locks in `lock-alist` sind bereits von anderem Thread geholt
- Für freies Lock versuchen, es an Thread zuzuweisen

```
((obtain-message? request)
 (let ((id (obtain-message-id request))
       (reply-channel (obtain-message-reply-channel request))
       (abort-rv (obtain-message-abort-rv request)))
  (cond
   ((assoc id lock-alist)
    => (lambda (pair)
         (set-cdr! pair
                   (append (cdr pair)
                           (list (cons reply-channel abort-rv)))))
      (serve lock-alist)))
   (else
    (if (reply-to-obtain reply-channel abort-rv)
        (serve (cons (cons id '()) lock-alist))
        (serve lock-alist))))))
```

Lock an Thread zuweisen

Gleichzeitig

- Lock über Antwortchannel zuweisen, `#t` liefern
- Bei Nichtinteresse `#f` liefern

`abort-rv` stammt von `with-nack`

```
(define (reply-to-obtain reply-channel abort-rv)
  (select
    (wrap (send-rv reply-channel #f)
          (lambda (ignore) #t))
    (wrap abort-rv
          (lambda (ignore) #f))))
```

release-message

- Wenn keiner mehr wartet, Lock aus lock-alist löschen
- Wenn vorderster synchronisiert ist Lock vergeben
- Ansonsten mit nächstem Thread weiter versuchen

```
((release-message? request)
 (let ((id (release-message-id request)))
  (cond
   ((assoc id lock-alist)
    => (lambda (pair)
         (let assign ((pending (cdr pair)))
          (cond
           ((null? pending)
            (serve (delq pair lock-alist)))
           ((reply-to-obtain (caar pending) (cdar pending))
            (set-cdr! pair (cdr pending))
            (serve lock-alist))
           (else
            (assign (cdr pending)))))))))))
```

Zusammenfassung

- **with-nack** liefert Rendezvous, auf das synchronisiert werden kann, wenn **choose** ein gegebenes Rendezvous nicht auswählt
- Häufig verwendet um Servern mitzuteilen, dass Client anderes Rendezvous gewählt hat
- Konsequenz aus der Anforderung, immer Rendezvous bereitzustellen