Verteilte Systeme

Organisatorisches

Tutorium

- Do, 8:30 - 10:00 Philipp Schmidt (ab 28.4.)
- Fr, 14:15 - 16:00 Andreas Nüßlein (ab 29.4.)

Verteilte Systeme

Klassifizierung von Kommunikationsdiensten II
Fehlertypen in Verteilten Systemen

Pseudocode für Senden und Empfangen

```
send MessageExpr to DestExpr

recv MessageVar [from SrcExp | SrcVar]
```

```
MessageExpr: zu sendende Nachricht
MessageVar:  Speicherplatz der die Nachricht aufnimmt

DestExpr:    Empfänger
SrcExp:      Quelle, von der Nachrichten empfangen werden
SrcVar:      Speicherplatz, der die Quelle der
             Nachricht aufnimmt
```

Kommunikationsdienste variieren im Hinblick auf

- Übertragungssemantik
- Adressierung der Kommunikationspartner
- Konfiguration von Prozessen und Kanälen
- Disjunktives Warten

Übertragungssemantik

- Pufferung
- Empfangsfolge
- Zuverlässigkeit
- Flußsteuerung

Pufferung

P:

send “Hello World” to Q

Q:

recv var from P

var == “Hello World”

ungepuffert mit Rendezvous:
(z.B. Ports in CSP oder Eerlang)

ungepuffert ohne Rendezvous:
(z.B. Ethernet)

var == null

gepuffert:
(z.B. TCP oder UDP)

var == “Hello World”

Empfangsfolge

```
send “Hello”
send “ ”
send “World”
```

permutiertes Prefix der Sendefolge	[“Hello”, “ “, “World”] [“ “, “Hello”, “World”] [“World”, “Hello”, “ “] …
Präfix der Sendefolge (reihenfolgetreu, FCFS)	[“Hello”, “ “, “World”]
Strom (stream)	“Hello World”

Zuverlässigkeit

Empfangsfolge der Nachrichten evtl. gefährdet durch

- Duplizierung
- Verlust
- Verstümmelung: wird mittels Prüfcode erkannt und als Verlust betrachtet

Flußsteuerung & Syncronisation

Empfang mit recv:

- blockierend bis eine Nachricht vorliegt
- nichtblockierend – falls keine neue Nachricht vorliegt:
  Leeroperation oder nochmaliges Lesen einer alten Nachticht

Senden mit send:

- blockierend bis Nachricht absendbar
- nichtblockierend – falls Nachricht nicht absendbar:
  Verwerfen dieser oder einer älteren Nachricht

Flußsteuerung & Syncronisation

Synchrone vs. asynchrone Übertragung:

- asynchrone Übertragung:
  send kehrt zurück, sobald die Nachricht an das Nachrichtensystem übergeben ist

- synchrone Übertragung:
  send ist beendet, sobald die Nachricht durch recv empfangen wurde

Adressierung

Kommunikation beschränkt sich nicht zwangsläufig auf zwei Kommunikationspartner

Vier Varianten:

- ohne Adressierung
- prozessbezogene Adressierung
- prozessgruppenbezogene Adressierung
- kanalbezogene Adressierung

ohne Adressierung

`send` MsgExpr

produziert Nachricht
(evtl. mit Absender)

`recv` MsgVar [`from` ProcVar]
übernimmt Nachricht

(und gegebenenfalls Absender)

prozessbezogene Adressierung

```
send MsgExpr to ProcExp        recv MsgVar [ from ProcVar ]
```

Idee: Jeder Prozess besitzt eine eigene Nachrichtenwarteschlange

```
send MsgExpr [ to ProcExp ]    recv MsgVar from ProcVar
```

Idee: Prozesse können Nachrichten aus einer globalen Nachrichten-warteschlange empfangen.

prozessgruppenbezogene Adressierung (multicast)

Es gibt Prozessgruppen, denen Prozesse mit Hilfe spezieller Operationen beitreten oder diese verlassen können:

```
group.enter(); … group.leave();
```

Nachrichten können dann an diese Gruppen gesendet werden:

```
send msg to group

                        recv var [ from group | proc ]
```

kanalbezogene / portbasierte Adressierung

Kanal als eingenständiges Pufferobjekt:

```
send msg to channel
```

```
recv var from channel
```

oder

```
channel.send(msg)
```

```
channel.recv(msg)
```

Begriffsklärung: Port

- Physischer Anschluß an Hardwarekomponente
- Kanal, der durch einen Prozess bereitgestellt wird
- Formaler Parameter, der innerhalb eines Prozesses einen Kanal beschreibt
  z.B. stdin/stdout über Filedescriptor 0/1

- Häufig sind mit Ports nur unidirektionale Ein- oder Ausgänge eines Kanals gemeint.

Konfiguration von Kanälen

bestimmt wie Prozesse über Kanäle verbunden sind, kann aber auch die Erzeugung von Prozessen beinhalten.

Statisches Binden:

- Hardware: durch verkabeln der Ports der Komponenten
- Software: mittels einer Konfigurationssprache

  Beispiel: Pipelines in Unix-Shells

  ```
  ls | awk '{print $5}' | sort
  ```

  Adressierung Kanalbezogen: Kanal = |

  Port = stdin/stdout

Dynamisches Binden:

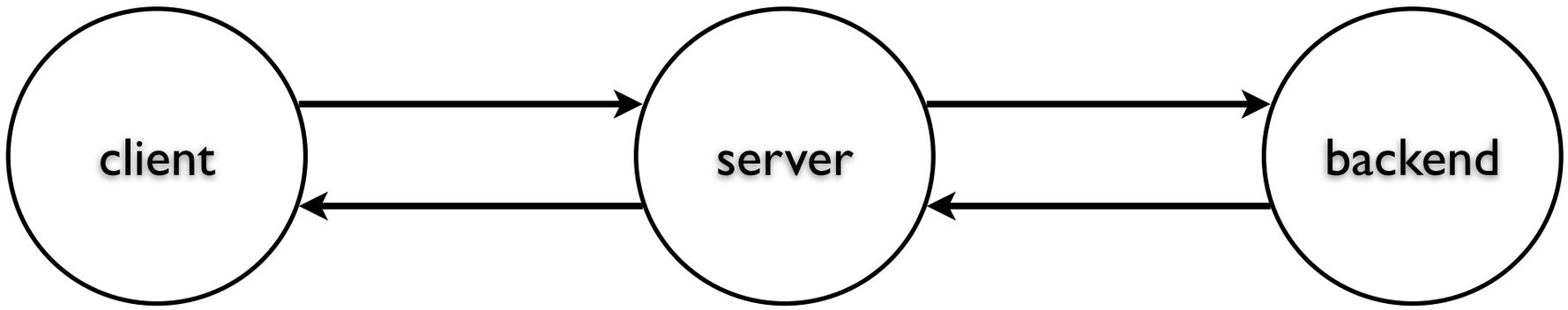
Prozesse Konfigurieren ihre Kanäle und Ports selbst

+ jederzeit dynamische Umkonfigurierung möglich
- Wiederverwendbarkeit in anderen Kontexten begrenzt

Beispiel: Verbinden an einen TCP-Socket:

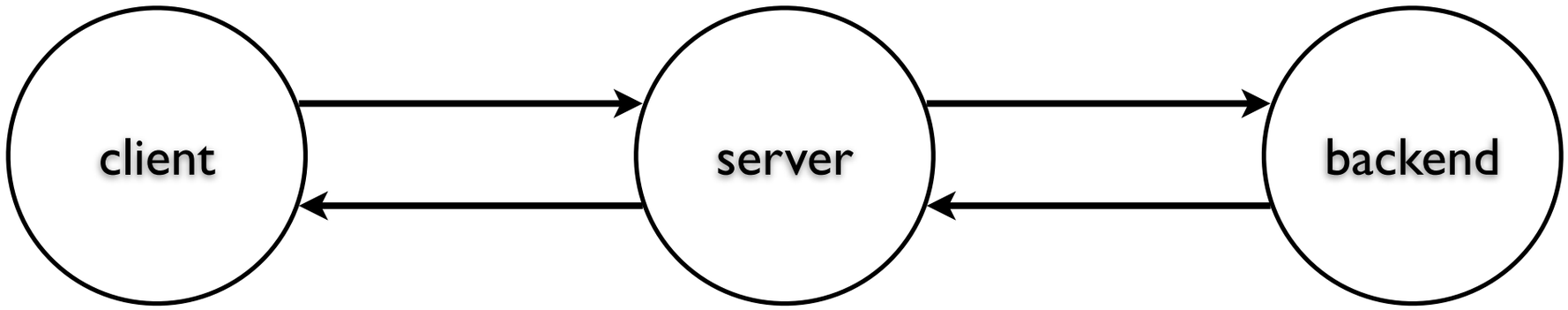
```
SocketChannel sc = SocketChannel.open();
sc.connect( new InetSocketAddress (
                    "sample.service.invalid", 2343));
```

Disjunktives Warten



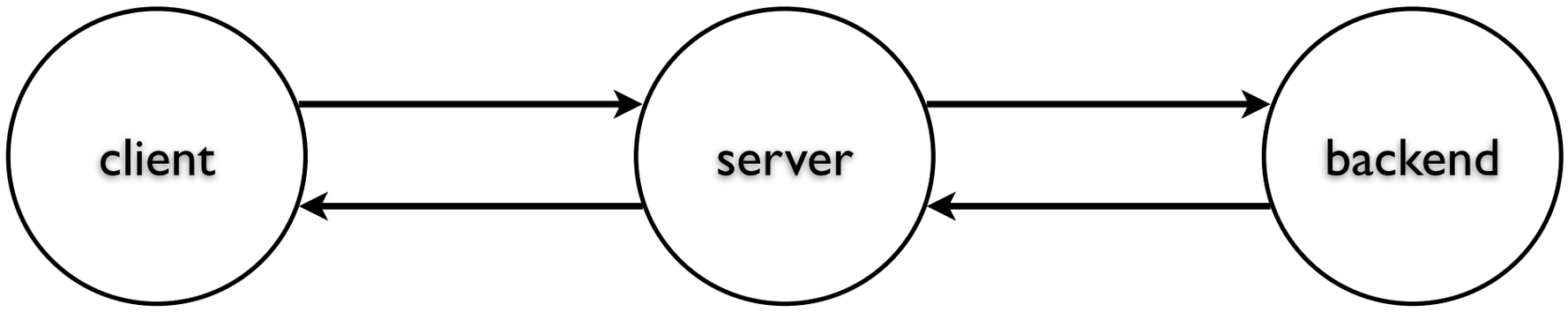

```
server:
while(true) {
    …
    recv request from client; …
    send subrequest to backend; …
    recv subresult from backend; …
    send result to client; …
}
```

nichtdeterministisches Disjunktives Warten




```
server:
while(true) {
    select:
        | recv request from client { …
                send subrequest to backend; … }
        | recv subresult from backend { …
                send result to client; … }
        | timeout t { … }
    ;;
}
```

Disjunktives Warten mit Threads




```
server:
    co:
        | { recv request from client; …
              send subrequest to backend; … }
        | { recv subresult from backend; …
              send result to client; … }
    ;;
}
```

Verteilte Systeme

Lokale Kommunikationsdienste

IPC in Unix

- Shared Memory (System V & BSD)
- Message Queues (System V)
- Pipes
- Sockets (BSD)

Shared Memory (Sys V)

```
#define SIZE 512

/* contents of file will not bet affected */
key_t shmkey = ftok(“/path/to/some/file”, 23);
int shmid = shmget(shmkey, SIZE, IPC_CREAT | 0600);
if (shmid >= 0) {
   char *p = shmat(shmid, 0, 0);
   if (p==(char *)-1) {
      perror(“failed”);
   } else {
      strncpy(p, “Hello World”, SIZE);
}
```

Shared Memory (BSD)

```
#define SIZE 512

/* file will reflect actual contents of shm */
int fd = fopen(“/path/to/some/file”,O_RDWR|O_CREAT);
if (fd >= 0) {
   char *p = mmap(0, SIZE, PROT_READ|PROT_WRITE,
                  MAP_SHARED, fd, 0 );
   if (p==(char *)-1) {
      perror(“failed”);
   } else {
      strncpy(p, “Hello World”, SIZE);
}
```

Message Queues (Sys V)

```
#define SIZE 512
struct mymsg { long mtype; char mtext[SIZE]; } datamsg;

/* contents of file will not bet affected */
key_t msqkey = ftok("/path/to/some/file", 23);
int msqid = msgget(msqkey, IPC_CREAT | 0600);
if (msqid >= 0) {
   strncpy(datamsg->mtext, "Hello World", SIZE);
   if ( msgsnd(msqid, &datamsg, SIZE, 0) ==-1) {
      perror("failed");
   }
}
```

Pipes

Es gibt verschiedene Konzepte die Pipe genannt werden, und sich ähnlich wie normale Filedeskriptoren verhalten:

Verkettung von stdout und stdin durch die Shell:

```
> ls | more
```

```
int piped[2];
pipe(piped);
proc1->stdout = piped[1]; proc2->stdin = piped[0];
fork_and_exec(proc1); fork_and_exec(proc2);
```

Fifos:

```
mkfifo(“/path/to/create/fifo”, 0600);
int fd_read = fopen(“/path/to/some/file”,O_READ);
int fd_write = fopen(“/path/to/some/file”,O_WRITE);
```

Sockets (BSD)

Universelle API für verschiedene IPC-Varianten.

Synopsis: socket(int domain, int type, int protocol);

Beispiele für Domains:

- PF_UNIX – lokale IPC über Sockets im Dateisystem
- PF_INET6 - Kommunikation IPv6
- PF_INET – Kommunikation per legacy IP

Beispiele für Typen:

- SOCK_STREAM – zuverlässiger Datenstrom
- SOCK_DGRAM - unzuverlässige Paketkommunikation
- SOCK_SEQPACKET - zuverlässige Paketkommunikation (unüblich)

Sockets (BSD)

```
int sockfd, servlen,n;
struct sockaddr_un  serv_addr;

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, “/path/to/some/file);
servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);
if ((sockfd = socket(AF_UNIX, SOCK_STREAM,0)) < 0)
    perror("Creating socket");
if (connect(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
    perror("Connecting");
write(sockfd,”Hello World”,12);
```

Sockets (BSD)

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, “/path/to/some/file);
servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

if ((sockfd = socket(AF_UNIX, SOCK_STREAM,0)) < 0)
    perror("Creating socket");
if(bind(sockfd,(struct sockaddr *)&serv_addr,servlen)<0)
    error("binding socket");
listen(sockfd,5);

clilen = sizeof(cli_addr);
clifd = accept(sockfd,(struct sockaddr *)&cli_addr,&clilen);
```

Sockets (BSD)

Sockets unterstützen disjunktives warten, auch gemischt mit anderen Filedeskriptoren:

```
select()
```

Es ist möglich read und write auf nichtblockieren umzustellen:

```
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```