Wirkung der RPC-Fehlersemantik

wird in den Übungen aktualisiert

	Fehlerfreier Ablauf	Nachrichten-verluste	Ausfall des Servers
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥1 Ergebnis: ≥1	Ausführung: ≥0 Ergebnis: ≥0 (*)
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0/1 (+)	Ausführung: 0/1 Ergebnis: 0
Exactly-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

(*) Nach einem evtl. Server-Restart können Retries Ergebnisse verursachen

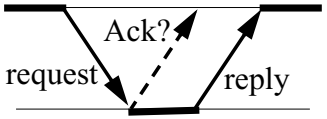
(+) Bei Verlust der *Antwort*nachricht wird diese erneut gesendet

May-be → At-least-once → At-most-once → ...
ist zunehmend aufwendiger zu realisieren!

- man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig
  (Aber dieses Motto gilt natürlich nicht in allen sonstigen Lebenssituationen! Ein Sicherheitsabstand durch “besser als notwendig” ist oft angebracht!)

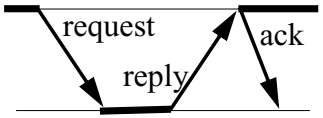
RPC-Protokolle

- *RR-Protokoll* (“Request-Reply”):



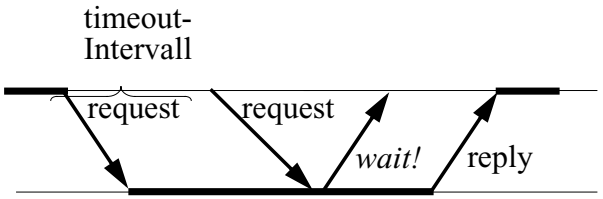
- Reply ist implizite Quittung für Request
- lohnt sich ggf. eine unmittelbare Bestätigung des Request?

- *RRA-Protokoll* (“Request-Reply-Acknowledge”):



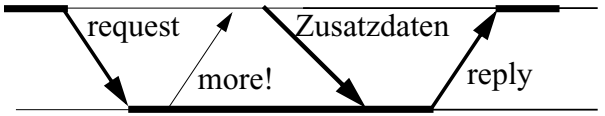
- “pessimistischer” als das RR-Protokoll
- Vorteil: Server kann evtl. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



“wait” = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand“*



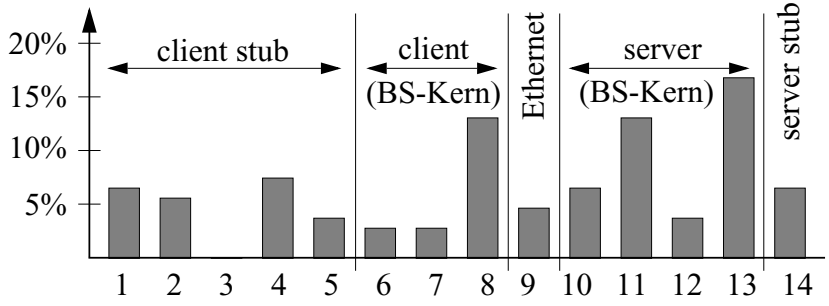
- spart Pufferkapazität
- bessere Flusssteuerung
- Zusatzdaten abhängig vom konkreten Ablauf

- *Weitere RPC-Protokollaspekte:*
  - effiziente Implementierung einer geeigneten (=?) Fehlersemantik
  - geeignete Nutzung des zugrundeliegenden Protokolls (ggf. aus Effizienzgründen eigene Paketisierung der Daten, Flusssteuerung, selektive Wiederholung einzelner Nachrichtenpakete bei Fehlern, eigene Fehlererkennung / Prüfsummen, kryptogr. Verschlüsselung...)

RPC: Effizienz

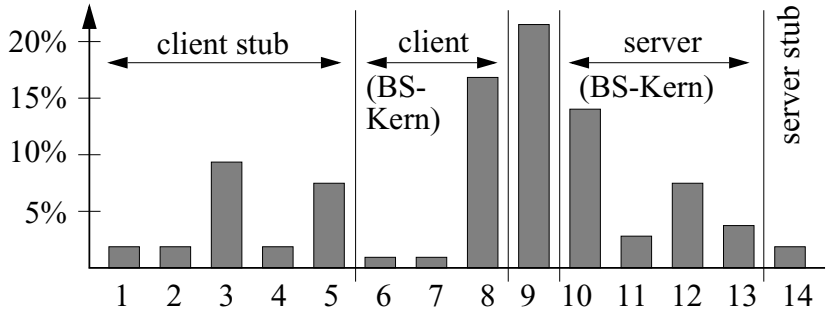
Analyse eines RPC-Protokolls
(zitiert nach A. Tanenbaum)

a) Null-RPC (Nutznachricht der Länge 0, kein Auftragsbearbeitung):



1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the bus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

b) 1440 Byte Nutznachricht (ebenfalls kein Auftragsbearbeitung):

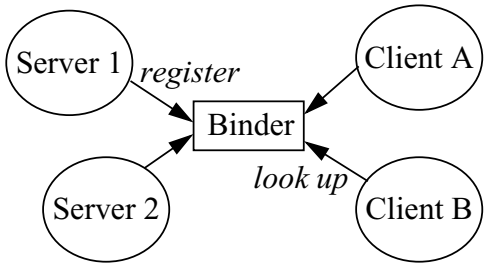


- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

RPC: Binding

- Problem: Wie werden Client und Server "gematcht"?
  - Verschiedene Rechner und i.a. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)

Dynamisches Binden mittels eines "Binders"



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem Binder bekannt
  - "*register*"; "exportieren" der RPC-Schnittstelle (Typen der Parameter...)
  - ggf. auch wieder abmelden
- Client erfragt beim Binder die Adresse eines geeigneten Servers
  - "*look up*"; "importieren" der RPC-Schnittstelle

oft auch "registry" oder "look-up service" genannt

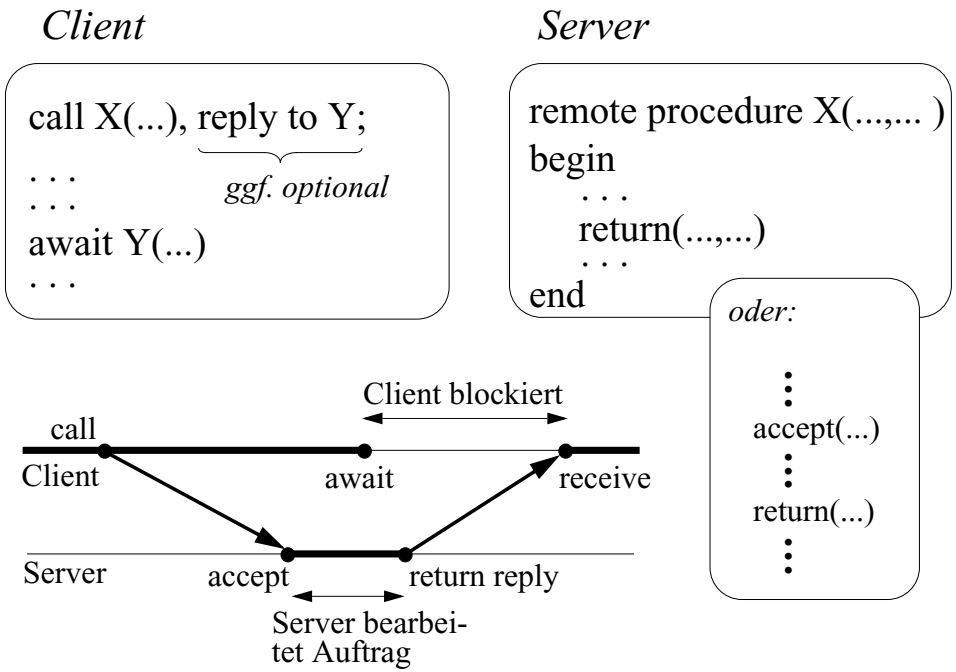
- *Vorteile*: im Prinzip kann Binder
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten

dann eher "Trader" oder "Broker"

- *Probleme*:
  - zentraler Binder ist ein potentieller Engpass (Binding-Service geeignet verteilen? Konsistenz!)
  - dynamisches Binden kostet Ausführungszeit

Asynchroner RPC

- andere Bezeichnung: “Remote Service Invocation”
- auftragsorientiert → Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Future-Variablen

- Zuordnung Auftrag / Ergebnisempfang bei der asynchron-auftragsorientierten Kommunikation?
  - unterschiedliche Ausprägung auf Sprachebene möglich
  - “await” könnte z.B. einen bei “call” zurückgelieferten “handle” als Parameter erhalten (also z.B. Y = call X(...); ... await (Y);)
  - ggf. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt-oder Exception-Routine)

- Spracheinbettung evtl. auch durch “Future-Variablen”
  - Future-Variable = handle, der wie ein Funktionsergebnis in Ausdrücke eingesetzt werden kann
  - Auswertung der Future-Variable erst, wenn unbedingt nötig
  - Blockade nur dann, falls Inhalt bei Auswertung noch nicht feststeht
  - Beispiel:

```
FUTURE future: integer;
some_value: integer;
…
future = call(…);
…
some_value = 4711;
print(some_value + future);
```

Die Socket-Programmierschnittstelle

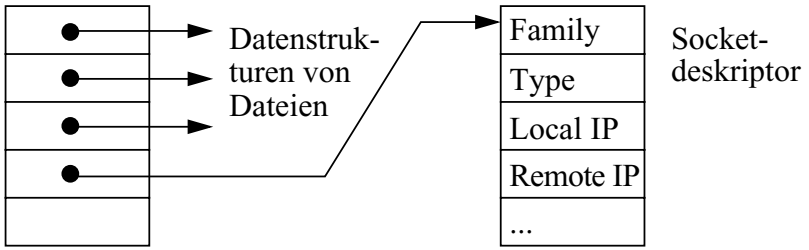
- Zu TCP (bzw. UDP) gibt es keine festgelegten “APIs”
- Bei UNIX sind dafür “sockets” als Zugangspunkte zum Transportsystem entstanden
  - -diese definieren in einer Sprache (z.B. Java) dann eine Art “API”
- Semantik eines sockets: analog zu Datei-Ein/Ausgabe
  - ein socket kann aber auch mit mehreren Prozessen verbunden sein
- Programmiersprachliche Einbindung (C, Java etc.)
  - sockets werden wie Variablen behandelt (können Namen bekommen)
  - Beispiel in C (Erzeugen eines sockets):

```
int s;
s = socket(int PF_INET, int SOCK_STREAM, 0);
```

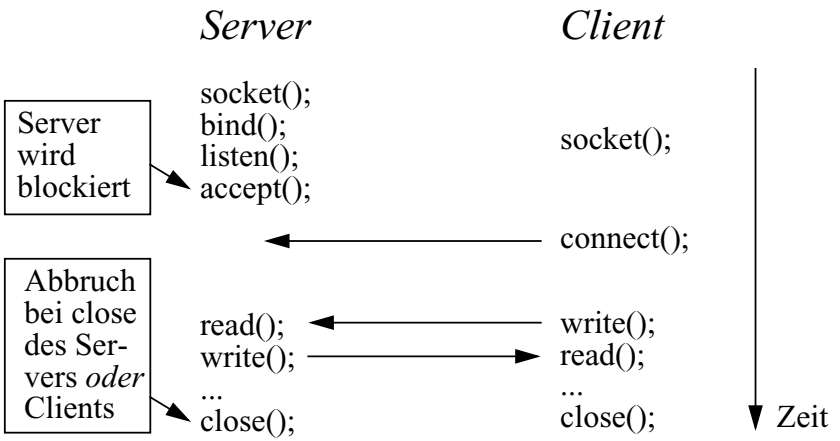
“Family”: Internet oder nur lokale Domäne

“Type”: Angabe, ob TCP verwendet (“stream”); oder UDP (“datagram”)

- Bibliotheksfunktion “socket” erzeugt einen Deskriptor
  - wird innerhalb der Filedeskriptor-Tabelle des Prozesses angelegt
  - Datenstruktur wird allerdings erst mit einem nachfolgenden “bind”-Aufruf mit Werten gefüllt (binden der Adressinformation aus Host-Adresse und einer “bekannten” lokalen Portnummer an den socket)



Client-Server mit Sockets (Prinzip)



- Voraussetzung: Client kennt die IP-Adresse des Servers sowie die Portnummer (des Dienstes)
  - muss beim connect angegeben werden
- Mit “listen” richtet der Server eine Warteschlange für Client-connect-Anforderungen ein
  - Auszug aus der Beschreibung: *“If a connection request arrives with the queue full, tcp will retry the connection. If the backlog is not cleared by the time the tcp times out, the connect will fail”*
- Accept / connect implementieren ein “Rendezvous”
  - mittels des 3-fach-Handshake von TCP
  - bei “connect” muss der Server bereits listen / accept ausgeführt haber
- Rückgabewerte von write bzw. read: Anzahl der tatsächlich gesendeten / empfangenen Bytes
- Varianten: Es gibt ein select, ein nicht-blockierendes accept etc., vgl. dazu die (Online-)Literatur

Ein Socket-Beispiel in C

- Verwendung von sockets in C erfordert u.a.
  - Header-Dateien mit C-Daten-Strukturen, Konstanten etc.
  - Programmcode zum Anlegen, Füllen etc. von Strukturen
  - Fehlerabfrage und Behandlung

- Socket-Programmierung ist ziemlich “low level”
  - etwas umständlich, fehleranfällig bei der Programmierung
  - aber “dicht am Netz” und dadurch ggf. manchmal von Vorteil

- Zunächst der Quellcode für den Client:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711
#define BUF_SIZE 1024

main(argc,argv)
int   argc;
char *argv[];
{
    int                 sock, run;
    char                buf[BUF_SIZE];
    struct sockaddr_in  server;
    struct hostent     *hp;
    if(argc != 2)
    {
        fprintf(stderr,”usage: client
                       <hostname>\n”);
        exit(2);
    }
```


Socket-Beispiel: Client

```
    /* create socket */
    sock = socket(AF_INET,SOCK_STREAM,0);
    if(sock < 0)
    {
        perror(“open stream socket”);
        exit(1);
    }
    server.sin_family = AF_INET;
    /* get internet address of host specified by command line */
    hp = gethostbyname(argv[1]);
    if(hp == NULL)
    {
       fprintf(stderr,”%s unknown host.\n”,argv[1]);
       exit(2);
    }
    /* copies the internet address to server address */
    bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
    /* set port */
    server.sin_port = PORT;
    /* open connection */
    if(connect(sock,&server,sizeof(struct sockaddr_in)) < 0)
    {
       perror(“connecting stream socket”);
       exit(1);
    }
    /* read input from stdin */
    while(run=read(0,buf,BUF_SIZE))
    {
       if(run<0)
       {
          perror(“error reading from stdin”);
          exit(1);
       }
       /* write buffer to stream socket */
       if(write(sock,buf,run) < 0)
       {
          perror(“writing on stream socket”);
          exit(1);
       }
    }
    close(sock);
}
```

Socket-Beispiel: Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711                    /* random port number */
#define MAX_QUEUE 1
#define BUF_SIZE 1024

main()
{
    int sock_1,sock_2;     /* file descriptors for sockets */
    int rec_value, length;
    char buf[BUF_SIZE];
    struct sockaddr_in server;

    /* create stream socket in internet domain*/
    sock_1 = socket(AF_INET,SOCK_STREAM,0);
    if (sock_1 < 0)
    {
       perror("open stream socket");
       exit(1);
    }
    /* build address in internet domain */
    server.sin_family = AF_INET;
    /* everyone is allowed to connet to server */
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = PORT;
    /* bind socket */
    if(bind(sock_1,&server,sizeof(struct sockaddr_in)))
    {
       perror("bind socket to server_addr");
       exit(1);
    }
```


Socket-Beispiel: Server (2)

```
    listen(sock_1,MAX_QUEUE);
    /* start accepting connection */
    sock_2 = accept(sock_1,0,0);
    if(sock_2 < 0)
    {
       perror("accept");
       exit(1);
    }
    /* read from sock_2 */
    while (rec_value=read(sock_2,buf,BUF_SIZE))
    {
       if(rec_value<0)
       {
          perror("reading stream message");
          exit(1);
       }
       else
          write(1,buf,rec_value);
    }
    printf("Ending connection.\n");
    close(sock_1); close(sock_2);
}
```

- Sinnvolle praktische Übungen (evtl. auch in Java):

1) Beispiel genau studieren; Semantik der socket-Operationen etc. nachlesen: Bücher oder Online-Dokumentation (z.B. von UNIX)

2) Varianten und andere Beispiele implementieren, z.B.:

- Server, der zwei Zahlen addiert und Ergebnis zurücksendet
- Produzent / Konsument mit dazwischenliegendem Pufferprozess (unter Vermeidung von Blockaden bei vollem Puffer)
- Server, der mehrere Clients gleichzeitig bedienen kann
- Trader, der geeignete Clients und Server zusammenbringt
- Messung des Durchsatzes im LAN; Nachrichtenlängen in mehreren Experimenten jeweils verdoppeln

Übungsbeispiel: Sockets unter Java

(auf den nächsten 5 Seiten)

- Auch unter Java lassen sich Sockets verwenden
  - sogar bequemer als unter C
  - Paket java.net.* enthält u.a. die Klasse “Socket”
  - Streamsockets (verbindungsorientiert) bzw. Datagrammsockets

- Beispiel:

```
DataInputStream in;
PrintStream out;
Socket server;
...
server = new Socket(getCodeBase().getHost(),7);
  // Klasse Socket besitzt Methoden
  // getInputStream bzw. getOutputStream, hier
  // Konversion zu DataInputStream / PrintStream:
in = new DataInputStream(server.getInputStream());
out = new PrintStream(server.getOutputStream());
...
  // Etwas an den Echo-Server senden:
out.println(...)
...
  // Vom Echo-Server empfangen; vielleicht
  // am besten in einem anderen Thread:
String line;
while((line = in.readLine()) != null)
  // line ausgeben
...
server.close;
```

Herstellen einer Verbindung

Hostname

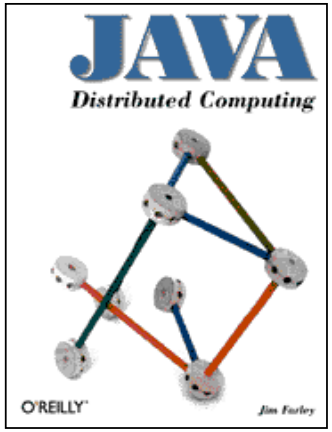
Echo-Port

Port Nummer 7 sendet alles zurück

- Zusätzlich: Fehlerbedingungen mit Exceptions behandeln (“try”; “catch”)
  - z.B. “UnknownHostException” beim Gründen eines Socket

Client-Server mit Sockets in Java

- Beispiel aus dem Buch *Java Distributed Computing* von Jim Farley (O’Reilly)



- Hier der Client:

```
import java.lang.*;
import java.net.*;
import java.io.*;

public class SimpleClient
{
  // Our socket connection to the server
  protected Socket serverConn;

    public SimpleClient(String host, int port)
      throws IllegalArgumentException {
    try {
      System.out.println(“Trying to connect to “
      + host + “ “ + port);
      serverConn = new Socket(host, port);
    }
    catch (UnknownHostException e) {
      throw new IllegalArgumentException
        (“Bad host name given.“);
    }
    catch (IOException e) {
      System.out.println(“SimpleClient: “ + e);
      System.exit(1);
    }

    System.out.println(“Made server connection.“);
  }
```

Konstruktor

```
  public static void main(String argv[]) {
    if (argv.length < 2) {
      System.out.println ("Usage: java \
         SimpleClient <host> <port>");
      System.exit(1);
    }

    int port = 3000;
    String host = argv[0];
    try { port = Integer.parseInt(argv[1]); }
    catch (NumberFormatException e) {}

    SimpleClient client  =  new SimpleClient(host, port);
    client.sendCommands();
  }

  public void sendCommands() {
    try {
      DataOutputStream dout =
        new DataOutputStream(serverConn.getOutputStream());
      DataInputStream din =
        new DataInputStream(serverConn.getInputStream());

      // Send a GET command...
      dout.writeChars("GET goodies ");
      // ...and receive the results
      String result = din.readLine();
      System.out.println("Server says: \"" + result + "\"");
    }
    catch (IOException e) {
      System.out.println("Communication SimpleClient: " + e);
      System.exit(1);
    }
  }

  public synchronized void finalize() {
    System.out.println("Closing down SimpleClient...");
    try { serverConn.close(); }
    catch (IOException e) {
      System.out.println("Close SimpleClient: " + e);
      System.exit(1);
    }
  }
}
```

Host- und Port-nummer von der Kommandozeile

Wird vom Garbage-Collector aufgerufen, wenn keine Referenzen auf den Client mehr existieren ('close' ggf. am Ende von 'sendCommands')

Der Server

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class SimpleServer {
  protected int portNo = 3000;
  protected ServerSocket clientConnect;

  public SimpleServer(int port) throws
      IllegalArgumentException {
    if (port <= 0)
      throw new IllegalArgumentException(
       "Bad port number given to SimpleServer constructor.");

    // Try making a ServerSocket to the given port
    System.out.println("Connecting server socket to port");
    try { clientConnect = new ServerSocket(port); }
    catch (IOException e) {
     System.out.println("Failed to connect to port " + port);
      System.exit(1);
    }

    // Made the connection, so set the local port number
    this.portNo = port;
  }

  public static void main(String argv[]) {
    int port = 3000;
    if (argv.length > 0) {
      int tmp = port;
      try {
        tmp = Integer.parseInt(argv[0]);
      }
      catch (NumberFormatException e) {}
      port = tmp;
    }

    SimpleServer server = new SimpleServer(port);
    System.out.println("SimpleServer running on port " +
       port + "...");
    server.listen();
  }
```

Default-Port, an dem der Server auf eine Client-Verbindung wartet

Socket, der Verbindungs-wünsche entgegennimmt

Konstruktor

Portnummer von Kommandozeile

Aufruf der Methode "listen" (siehe unten)

```
public void listen() {
    try {
      System.out.println("Waiting for clients...");
      while (true) {
        Socket clientReq = clientConnect.accept();
        System.out.println("Got a client...");
        serviceClient(clientReq);
      }
    }
    catch (IOException e) {
      System.out.println("IO exception while listening.");
      System.exit(1);
    }
  }

  public void serviceClient(Socket clientConn) {
    SimpleCmdInputStream inStream = null;
    DataOutputStream outStream = null;
    try {
      inStream = new SimpleCmdInputStream
        (clientConn.getInputStream());
      outStream = new DataOutputStream
        (clientConn.getOutputStream());
    }
    catch (IOException e) {
      System.out.println("SimpleServer: I/O error.");
    }
    SimpleCmd cmd = null;
    System.out.println("Attempting to read commands...");
    while (cmd == null || !(cmd instanceOf DoneCmd)) {
      try { cmd = inStream.readCommand(); }
      catch (IOException e) {
        System.out.println("SimpleServer (read): " + e);
        System.exit(1);
      }

      if (cmd != null) {
        String result = cmd.Do();
        try { outStream.writeBytes(result); }
        catch (IOException e) {
          System.out.println("SimpleServer (write): " + e);
          System.exit(1);
        }
      }
    }
  }
}
```

Warten auf connect eines Client, dann Gründen eines Sockets

Von DataInput-Stream abgeleitete Klasse

Klasse SimpleCmd hier nicht gezeigt

Schleife zur Entgegennahme und Ausführung von Kommandos

finalize-Methode hier nicht gezeigt

Java als "Internet-Programmiersprache"

- Java hat eine Reihe von Konzepten, die die Realisierung verteilter Anwendungen erleichtern, z.B.:
  - Socket-Bibliothek zusammen mit Input- / Output-Streams
  - Remote Method Invocation (RMI): Entfernter Methodenaufruf mit Transport (und dabei Serialisierung) auch komplexer Objekte
  - CORBA-APIs
  - eingebautes Thread-Konzept
  - java.security-Paket
  - plattformunabhängiger Bytecode mit Klassenlader (Java-Klassen können über das Netz transportiert und geladen werden; Bsp.: Applets)

Damit z.B. Realisierung eines "Meta-Protokolls": Über einen Socket vom Server eine Klasse laden (und Objekt-Instanz gründen), was dann (auf Client-Seite) ein spezifisches Protokoll realisiert. (Stichworte: "mobiler Code" bzw. Jini)

- Das UDP-Protokoll kann mit "Datagram-Sockets" verwendet werden, z.B. so:

```
try {
  DatagramSocket s = new DatagramSocket();
  byte[] data = {'H','e','l','l','o'};
  InetAddress addr = InetAddress.getByName("my.host.com");
  DatagramPacket p = new DatagramPacket(data,
    data.length, addr, 5000);
  s.send(p);
}
catch (Exception e) {
  System.out.println("Exception using datagrams:");
  e.printStackTrace();
}
```

Port-Nummer

  - entsprechend zu "send" gibt es ein "receive"
  - InetAddress-Klasse repräsentiert IP-Adressen
  - diese hat u.a. Methoden "getByName" (klassenbezogene Methode) und "getAddress" (instanzbezogene Methode)
  - UDP ist verbindungslos und nicht zuverlässig (aber effizient)

URL-Verbindungen in Java

- Java bietet einfache Möglichkeiten, auf "Ressourcen" (i.w. Dateien) im Internet mit dem HTTP-Protokoll lesend und schreibend zuzugreifen
  - falls auf diese mittels einer URL verwiesen wird
- Klasse "URL" in java.net.*
  - auf höherem Niveau als die Socket-Programmierung
  - Sockets (mit TCP) werden vor dem Anwender verborgen benutzt
- Beispiel: zeilenweises Lesen einer Textdatei
  - hier nicht gezeigt: Abfangen diverser Fehlerbedingungen!

```
// Objekt vom Typ URL anlegen:
URL myURL;
myURL = new URL("http", ..., "/Demo.txt");
...
DataInputStream instream;
instream = new DataInputStream(myURL.openStream());
String line = "";
while((line = instream.readLine()) != null)
  // line verarbeiten
...
```

hier Hostname angeben

Name der Datei

- Es ist auch möglich, Daten an eine URL zu senden
  - POST-Methode, z.B. an Skript auf dem Server
- Ferner: Information über das Objekt ermitteln
  - z.B. Grösse, letztes Änderungsdatum, HTTP-Header etc.

Übungsbeispiel: Ein Bookmark-Checker

(auf den nächsten 2 Seiten)

```
import java.io.*; import java.net.*;
import java.util.Date; import java.text.DateFormat;

public class CheckBookmark {

  public static void main (String args[]) throws
       java.io.IOException, java.text.ParseException {

    if (args.length != 2) System.exit(1);

    // Create a bookmark for checking...
    CheckBookmark bm = new CheckBookmark(args[0], args[1]);

    bm.checkit(); // ...and check

    switch (bm.state) {
      case CheckBookmark.OK:
        System.out.println("Local copy of " +
           bm.url_string + " is up to date"); break;
      case CheckBookmark.AGED:
        System.out.println("Local copy of " +
           bm.url_string + " is aged"); break;
      case CheckBookmark.NOT_SUPPORTED:
        System.out.println("Webserver does not support \
           modification dates"); break;
      default: break;
    }
  }

  String url_string, chk_date;
  int state;

  public final static int OK = 0;
  public final static int AGED = 1;
  public final static int NOT_SUPPORTED = 2;

  CheckBookmark(String bm, String dtm) // Constructor
  { url_string = new String(bm);
    chk_date = new String(dtm);
    state = CheckBookmark.OK;
  }
```

```
  public void checkit() throws java.io.IOException,
       java.text.ParseException {

    URL checkURL = null;
    URLConnection checkURLC = null;

    try { checkURL = new URL(this.url_string); }
    catch (MalformedURLException e) {
      System.err.println(e.getMessage() + ": Cannot \
          create URL from " + this.url_string);
      return;
    }

    try {
      checkURLC = checkURL.openConnection();
      checkURLC.setIfModifiedSince(60);
      checkURLC.connect();
    }

    catch (java.io.IOException e) {
      System.err.println(e.getMessage() + ": Cannot \
          open connection to " + checkURL.toString());
      return;
    }

    // Check whether modification date is supported
    if (checkURLC.getLastModified() == 0) {
      this.state = CheckBookmark.NOT_SUPPORTED;
      return;
    }

    // Cast last modification date to a "Date"
    Date rem = new Date(checkURLC.getLastModified());

    // Cast stored date of bookmark to Date
    DateFormat df = DateFormat.getDateInstance();
    Date cur = df.parse(this.chk_date);

    // Compare and set flag for outdated bookmark
    if (cur.before(rem)) this.state = CheckBookmark.AGED;
  }
}
```


Adressierung

- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
  - ggf. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
  - ggf. mehrere Adressaten gleichzeitig (Broadcast, Multicast)

- *Empfänger* ist evtl. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang (Spezialisierung)
  - Sicherheitsaspekte, Überlastabwehr

- Probleme
  - *Ortstransparenz*: Sender weiss *wer*, aber nicht *wo* (sollte er i.a. auch nicht!)
  - *Anonymität*: Sender und Empfänger kennen einander zunächst nicht (sollen sie oft auch nicht)

Kenntnis von Adressen?

- Adressen sind u.a. Rechneradressen (z.B. IP-Adresse oder Netzadresse in einem LAN), Portnamen, Socketnummern, Referenzen auf Mailboxes...

- Woher kennt ein Sender die Adresse des Empfängers?

  1) Fest in den Programmcode integriert → unflexibel

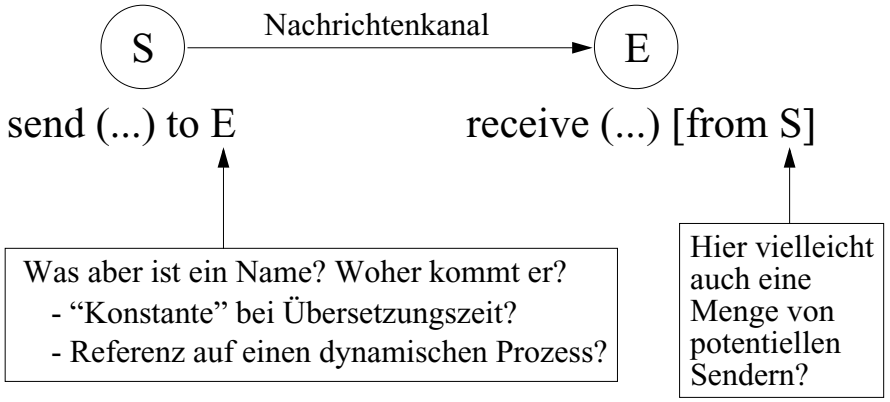
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt

  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.

  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)
     - wie realisiert man dies effizient und fehlertolerant?

Direkte Adressierung

- *Direct Naming* (1:1-Kommunikation):

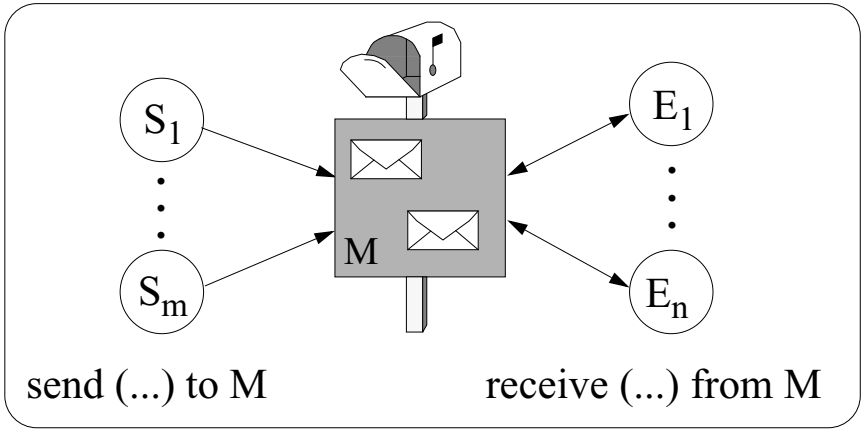


- Direct naming ist insgesamt relativ unflexibel

- Empfänger (= Server) sollten nicht gezwungen sein, potentielle Sender (= Client) explizit zu nennen
  - Symmetrie ist also i.a. gar nicht erwünscht

Indirekte Adressierung - Mailbox

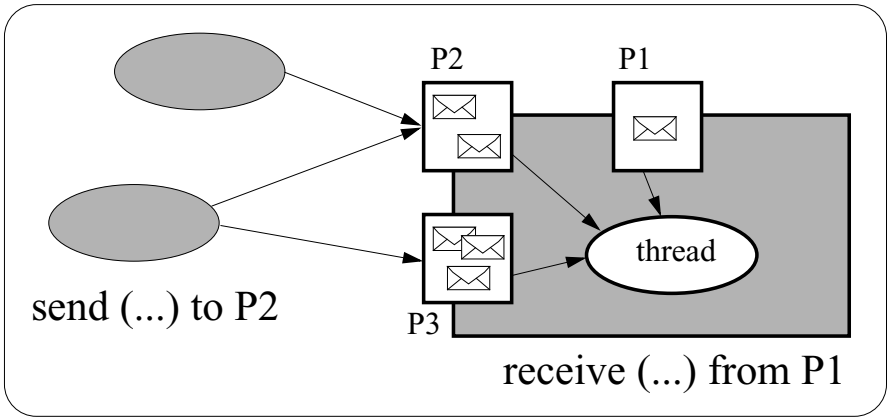
- Ermöglicht m:n-Kommunikation



- Eine Nachricht besitzt i.a. mehrere potentielle Empfänger
  - Mailbox spezifiziert damit eine *Gruppe* von Empfängern
- Kann jeder Empfänger die Nachricht bearbeiten?
  - Mailbox i.a. typisiert: nimmt nur bestimmte Nachrichten auf
  - Empfänger kann sich u.U. Nachrichten der Mailbox ansehen / aussuchen...
  - aber wer garantiert, dass jede Nachricht irgendwann ausgewählt wird?
- Wo wird die Mailbox angesiedelt? (→ Implementierung)
  - als ein einziges Objekt auf irgendeinem (geeigneten) Rechner?
  - repliziert bei den Empfängern? Abstimmung unter den Empfängern notwendig (→ verteiltes Cache-Kohärenz-Problem)
  - Nachricht verbleibt in einem Ausgangspuffer des Senders: Empfänger müssen sich bei allen (welche sind das?) potentiellen Sendern erkundigen
- Mailbox muss gegründet werden: Wer? Wann? Wo?

Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports sind Mailboxes mit genau einem Empfänger
  - Port gehört diesem Empfänger
  - Kommunikationsendpunkt, der die interne Empfängerstruktur abkapselt
- Ein Objekt kann i.a. mehrere Ports besitzen

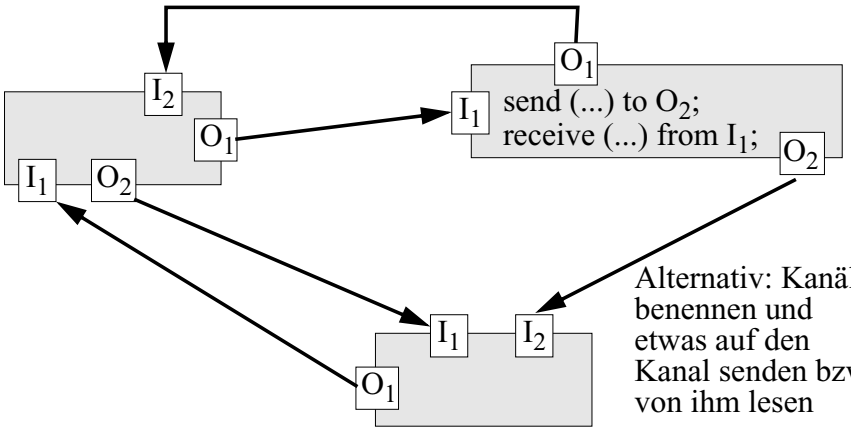


Pragmatische Aspekte (Sprachdesign etc.):

- Sind Ports statische oder dynamische Objekte?
- Wie erfährt ein Objekt den Portnamen eines anderen (dynamischen) Objektes?
  - können Namen von Ports verschickt werden?
- Sind Ports typisiert?
  - würde den selektiven Nachrichtenempfang unterstützen
- Grösse des Nachrichtenpuffers?
- Können Ports geöffnet und geschlossen werden?
  - genaue Semantik?

Kanäle und Verbindungen

- Neben *Eingangsports* (“in-port”) lassen sich auch *Ausgangsports* (“out-port”) betrachten



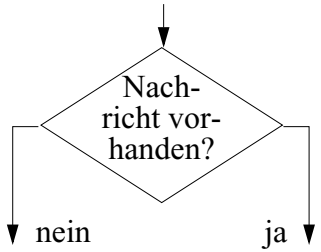
- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* (“Kanäle”) gewählt werden
- Dazu werden je ein in- und out-Port miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect** p1 **to** p2
  - denkbar sind auch broadcastfähige Kanäle
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen

Konfigurationsphase

- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
- Kommunikationsbeziehung: wahlweise 1:1, n:1, 1:n, n:m

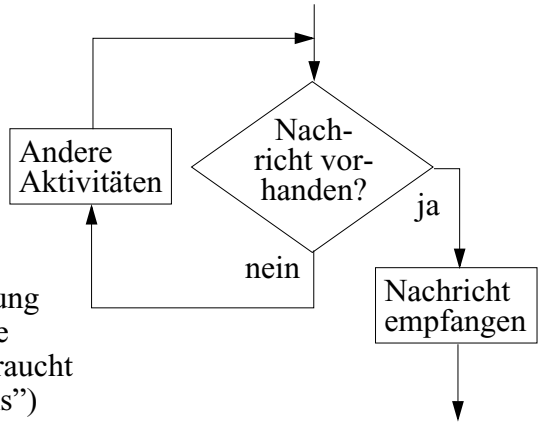
Varianten beim Empfangen von Nachrichten - Nichtblockierung

- Typischerweise ist ein “receive” blockierend
- Aber auch *nichtblockierender* Empfang ist denkbar:



- “Non-blocking receive”
- Sprachliche Realisierung z.B. durch “Returncode” eines als Funktionsaufruf benutzten “receive”

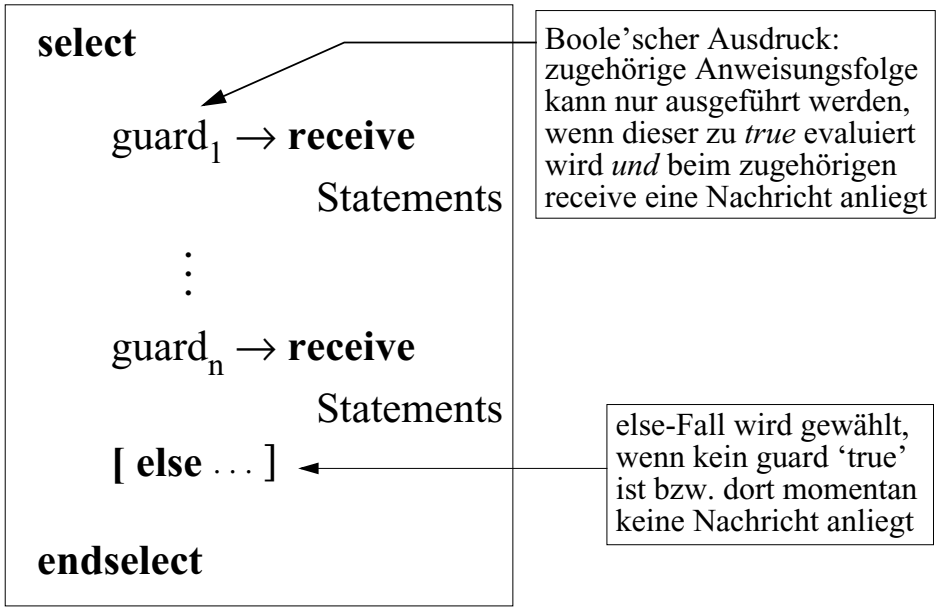
- Aktives Warten:
  (“busy waiting”)
  - Nachbildung des blockierenden Wartens wenn “andere Aktivitäten” leer
  - Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)



- Weitere Möglichkeit: unterbrechungsgesteuertes (“asynchrones”) Empfangen der Nachricht (→ nicht unproblematisch!)

Alternatives Empfangen

- Sprachliche Realisierung z.B. so:



- Was geschieht, wenn mehrere guards "true" sind?
  → nichtdeterministische Auswahl, Wahl des ersten Falles...

- Typischerweise *nichtblockierend*; aber blockierend, wenn else-Alternative fehlt

- Aktives Warten durch umschliessende while-Schleife
  - bei else könnte im Fall von aktivem Warten die while-Bedingung auf false gesetzt werden, falls das Warten abgebrochen werden soll, oder es könnte mittels timer ("wait") eine kurze Zeit gewartet werden
  - else-Fall kann auch einfach das leere Statement enthalten

Zeitüberwachte Kommunilation

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren ("timeout")
  - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat

- Sinnvoll bei:
  - Echtzeitprogrammierung
  - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)
    → dann sinnvolle Recovery-Massnahmen treffen ("exception")
    → timeout-Wert "sinnvoll" setzen!
    Quelle vielfältiger Probleme...

- Timeout-Wert = 0 kann ggf. genutzt werden, um zu testen, ob eine Nachricht "jetzt" da ist

---

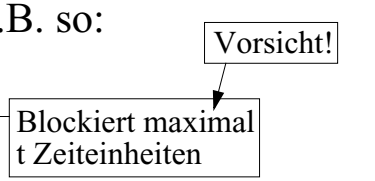
- Analog ggf. auch für synchrones (!) *Senden* sinnvoll
  → Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun "asynchron" an...

Zeitüberwachter Nachrichtenempfang

- Möglicher Realisierung:
  - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
  - Sprungziel könnte z.B. eine Unterbrechungsroutine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive

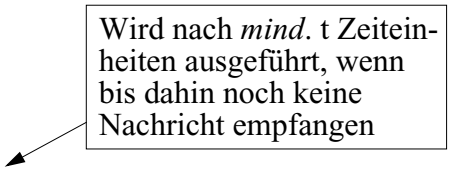
  → "systemnahe", unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:



receive ... **delay** t

- bzw. so:



select
  guard$_1$ → **receive** ...
  ⋮
  delay t → Anweisungen ...
endselect

- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!

- Frage: sollte "delay 0" äquivalent zu "else" sein?

Selektives Empfangen

≠ alternatives!

- Bedingung an den *Inhalt* (bzw. Typ, Format,...) der zu empfangenden Nachricht

- Dadurch werden gewisse ("unpassende") Nachrichten einfach ausgeblendet

- Bedingung wird oft vom aktuellen Zustand des Empfängers abhängen

---

- Vorteil bei der Anwendung:

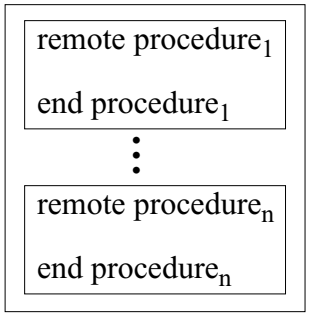
  Empfänger muss nicht mehr alles akzeptieren und zwischenspeichern, sondern bekommt nur das, was ihn momentan interessiert

Implizites Empfangen

- Keine receive, select...-Anweisung, sondern Spezifikation von Routinen, die bei Vorliegen einer Nachricht ausgeführt ("angesprungen") werden

- z.B. RPC:

bzw. asynchrone Variante oder "Remote Method Invocation" bei objektorientierten Systemen



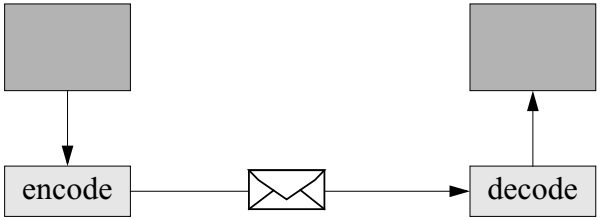
- Analog auch für den "Empfang" einer Nachricht *ohne Antwortverpflichtung* denkbar

---

- *Semantik*:
  - Interne Parallelität?
    - Mehr als eine gleichzeitig aktive Prozedur, Methode, thread... ("Routine") im Empfänger?
    - Vielleicht sogar mehrere Instanzen der gleichen Routine?
  - Atomare Routinen?
    - Wird eine aktive Routine ggf. unterbrochen, um eine andere aktivierte auszuführen?

Kommunizierbare Datentypen?

- Werte von "klassischen" einfachen Datentypen
  - int, character, string, floating point,...
- Kompatibilität in heterogenen Systemen
  - Grösse von int?
  - Codierung von Text?
  - höherwertiges Bit links oder rechts?
  - ...



- Vereinbarung einer *Standardrepräsentation* (z.B. XDR)
- marshalling (encode / decode) kostet Zeit

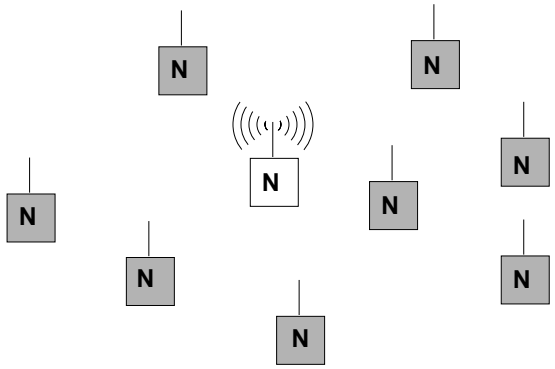
- Was ist mit *komplexen Datentypen* wie
  - Records, Strukturen
  - Objekte
  - Referenzen, Zeiger
  - Zeigergeflechte

  } sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
  } sollen Referenzen symbolisch, relativ... interpretiert werden? Ist das stets möglich?
  } wie wird Typkompatibilität sichergestellt?
- Ggf. "Linearisieren" und ggf. Strukturbeschreibung mitschicken (u.U. "sprachunabhängig")
- Sind (Namen von) Ports, Prozessen... eigene Datentypen, deren Werte versendet werden können?

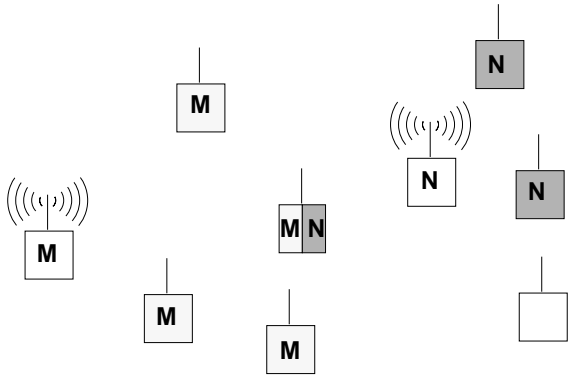
"first class objects"

Gruppen-kommunikation

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer

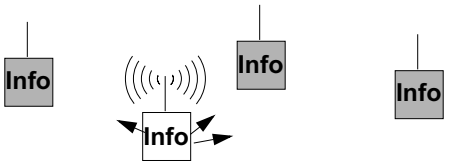


Multicast: Senden an eine *Untergruppe* aller Teilnehmer

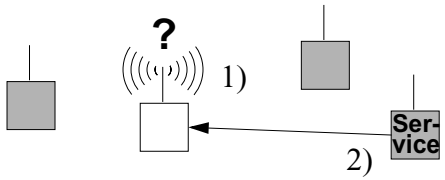
- entspricht Broadcast bezogen auf die Gruppe
- verschiedene Gruppen können sich ggf. überlappen
- jede Gruppen hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

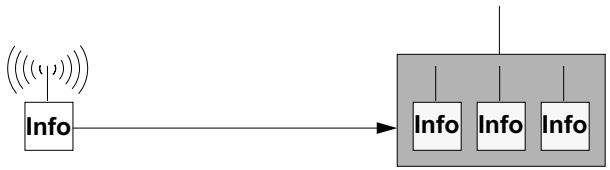
- *Informieren*
  - z.B. Newsdienste



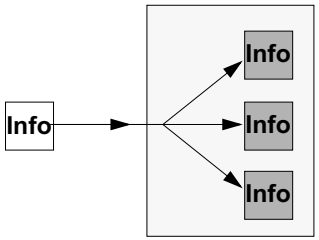
- *Suchen*
  - z.B. Finden von Objekten und Diensten



- *"Logischer Unicast"* an replizierte Komponenten



logisch

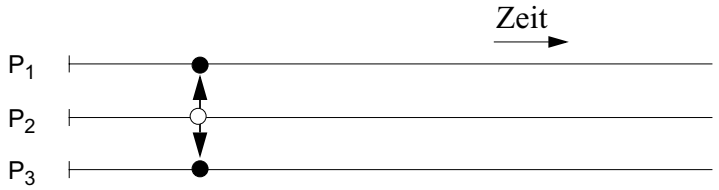


tatsächlich

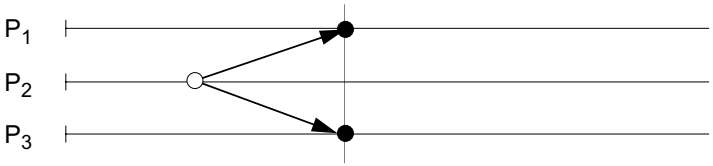
Typische Anwendungsklasse von Replikation: Fehlertoleranz

Gruppenkommunikation - idealisierte Semantik

- Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen
  - augenblicklicher Empfang
  - vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



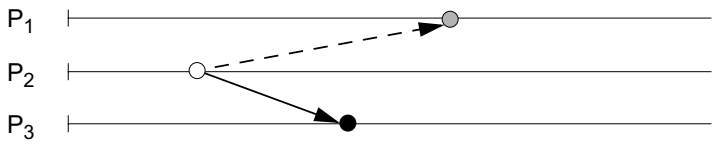
- Nachrichtenbasierte Kommunikation: Idealisierte Sicht
  - (verzögerter) gleichzeitiger Empfang
  - vollständige Zuverlässigkeit



Gruppenkommunikation - tatsächliche Situation

- Medium (Netz) ist oft nicht multicastfähig
  - LANs teilweise (z.B. klassisches Ethernet), jedoch i.a. nur innerhalb von Teilstrukturen; WLAN als Funknetz
  - multicastfähiges Netz ist effizient (Hardwareunterstützung), typischerweise jedoch nicht verlässlich (keine Empfangsgarantie)
  - bei Punkt-zu-Punkt-Netzen: "Simulation" von Multicast durch ein Protokoll (z.B. Multicast-Server, der an alle einzeln weiterverteilt)

- Nachrichtenkommunikation ist nicht "ideal"
  - indeterministische Zeitverzögerung → Empfang zu unterschiedlichen Zeiten
  - nur bedingte Zuverlässigkeit der Übermittlung



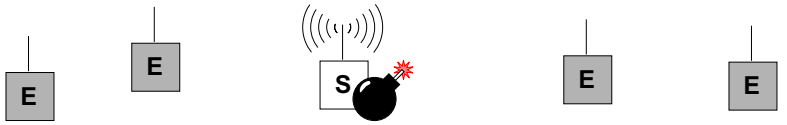
- Ziel von Broadcast / Multicast-Protokollen:
  - möglichst gute Approximation einer speicherbasierten Kommunikation
  - möglichst hohe Verlässlichkeit und Effizienz

  - Beachte: Verlust von Nachrichten und sonstige Fehler sind bei Broadcast ein ernsteres Problem als beim "Unicast"! (Wieso?)
  - Hauptproblem bei der Realsierung von Broadcast: (1) Zuverlässigkeit und (2) garantierte Empfangsreihenfolge

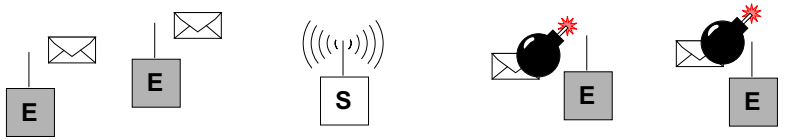
Senderausfall beim Broadcast

Wenn Broadcast implementiert ist durch Senden vieler Einzelnachrichten

a) Sender fällt ganz aus: *kein* Empfänger erhält Nachricht
  - „günstiger" Fall: *Einigkeit* unter den Überlebenden!



b) Sender fällt *während* des Sendevorgangs aus: nur *einige* Empfänger erhalten u.U. die Nachricht
  - "ungünstiger" Fall: *Uneinigkeit*



  - mögliche Abhilfe: Empfänger leiten die Nachricht untereinander weiter
  - Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein! (Da wäre es manchmal besser, *gar kein* Prozess hätte die Nachricht empfangen!)

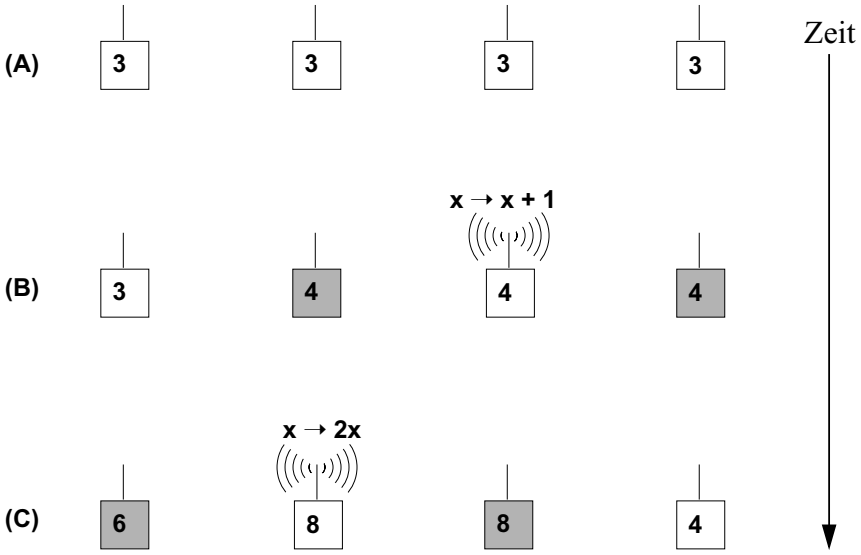
Zuverlässigkeitsgrad “best effort” bei Broadcasts

- Fehlermodell: Verlust von Nachrichten (und ggf. temporärer Crash von Prozessen)
  - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu...)
- Euphemistische Bezeichnung, da keine extra Anstrengung
  - typischerweise einfache Realisierung ohne Acknowledgements etc.
- Keinerlei Garantien
  - unbestimmt, wieviele / welche Empfänger eine Broadcastnachricht im Fehlerfall tatsächlich empfangen
  - unbestimmte Empfangsreihenfolge

  Kann z.B. beim Software-update über Satellit zu einem ziemlichen Chaos führen
- Allerdings effizient (im Erfolgsfall)
- Geeignet für die Verbreitung unkritischer Informationen
  - z.B. Informationen, die ggf. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
- Ggf. als Grundlage zur Realisierung höherer Protokolle
  - oft basierend auf der A-priori-Broadcastfähigkeit von Netzen
  - günstig bei zuverlässigen physischen Kommunikationsmedien (wenn Fehlerfall sehr selten → aufwendiges Recovery auf höherer Ebene tolerierbar)

k-Zuverlässigkeit

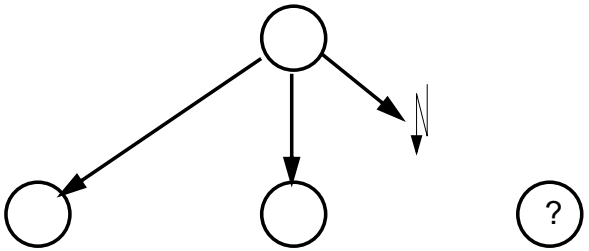
- Mindestens k Teilnehmer haben die Nachricht empfangen
  - grössere Werte von k sind “teurer”
  - Denkübungen: Wie realisiert man einen k-zuverlässigen Multicast? Ist ein “100%ig zuverlässiger” Broadcast überhaupt möglich? Wo lässt sich dies (für welches k?) sinnvoll verwenden?
- Problem der Fehlerakkumulation:
  - der Zustand (repräsentiert durch eine Variable x) sei repliziert
  - Zustandssynchronisation werde durch “function shipping” mittels 2-zuverlässigem Broadcast realisiert



- Ergebnis nach einiger Zeit: Alle Replikate sind verschieden!
- in einem solchen Fall hilft also k-Zuverlässigkeit nicht viel

“Reliable Broadcast”

- Ziel: Unter gewissen Fehlermodellen einen “möglichst zuverlässigen” Broadcast-Dienst realisieren



- Quittung (“positives Acknowledgement”: ACK) für jede Einzelnachricht ist teuer
  - im Verlustfall einzeln nachliefern oder (falls broadcastfähiges Medium vorhanden) einen zweiten Broadcast-Versuch? (→ Duplikaterkennung!)

- Skizze einer anderen Idee (“negatives Ack.”: NACK):
  - alle broadcasts werden vom Sender aufsteigend numeriert
  - Empfänger stellt beim *nächsten* Empfang u.U. eine Lücke fest
  - für fehlende Nachrichten wird ein “negatives ack” (NACK) gesendet
  - Sender muss daher Kopien von Nachrichten (wie lange?) aufbewahren und fehlende nachliefern
  - “Nullnachrichten” sind u.U. sinnvoll (→ schnelles Erkennen von Lücken)
  - Kombination von ACK / NACK mag sinnvoll sein

- Dies hilft aber nicht, wenn der Sender mittendrin crasht!

Reliable-Broadcast-Algorithmus

- Zweck: Jeder nicht gecrashte und zumindest indirekt erreichbare Prozess soll die Broadcast-Nachricht erhalten
  - Voraussetzung: zusammenhängendes “gut vermaschtes” Punkt-zu-Punkt-Netz
  - Fehlermodell: Knoten und Verbindungen mit Fail-Stop-Charakteristik

Sender *s*: Realisierung von **broadcast(*N*)**

- **send(*N*, *s*, *sequ_num*)** an alle Nachbarn (inklusive an *s* selber);
- sequ_num ++

Empfänger *r*: Realisierung des Nachrichtenempfangs

- **receive(*N*, *s*, *sequ_num*);**
  wenn *r* noch kein **deliver(*N*)** für *sequ_num* ausgeführt hat, dann:
  wenn $r \neq s$ dann **send(*N*, *s*, *sequ_num*)** an alle Nachbarn von *r* ;
  Nachricht an die Anwendungsebene ausliefern (“deliver(N)”) ;

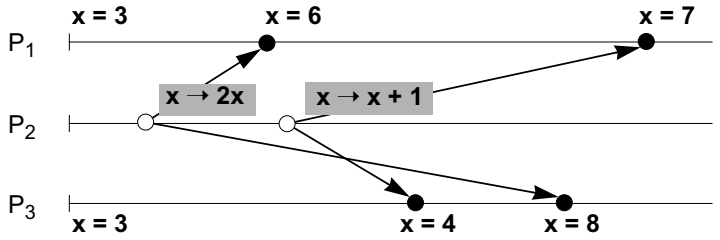
- Prinzip: “Fluten” des Netzes
  - vgl. dazu “Echo-Algorithmus” und Vorlesung “Verteilte Algorithmen”

- Beachte: receive ≠ deliver
  - unterscheide Anwendungsebene und Transportebene

- Fragen:
  - müssen die Kommunikationskanäle bidirektional sein?
  - wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
  - wie fehlertolerant? (wieviel darf kaputt sein / verloren gehen...?)
  - Optimierungen? Varianten?
  - kann man das gleiche auch ganz anders erreichen?

Broadcast: Empfangsreihenfolge

- Wie ist die Empfangsreihenfolge von Nachrichten?
  - problematisch wegen der i.a. ungleichen Übermittlungszeiten
  - Bsp.: Update einer replizierten Variablen mittels “function shipping”:

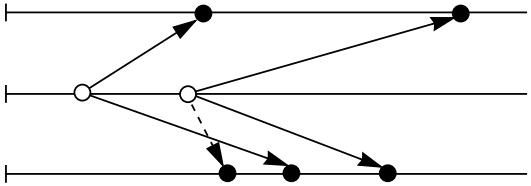


- Es sind verschiedene “Ordnungsgrade” denkbar
  - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet

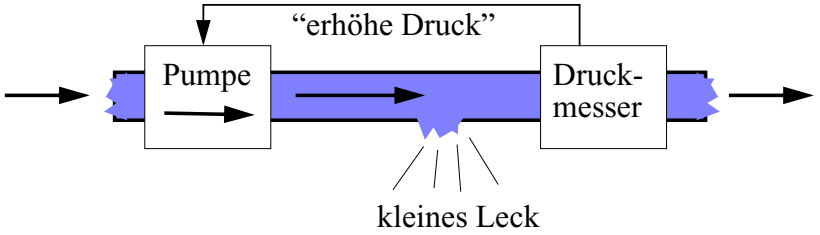
- *FIFO-Ordnung:*

  Alle Multicast-Nachrichten eines (d.h.: *ein und des selben*) Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an

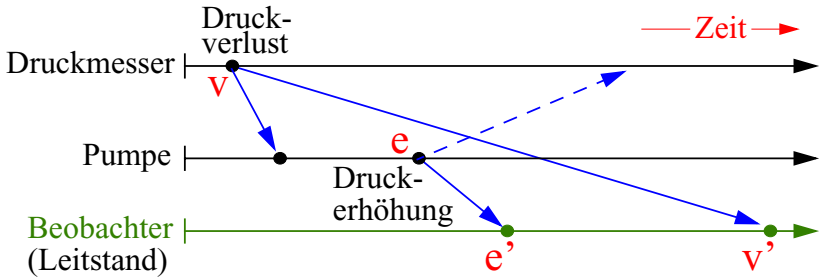
  - Denkübung: wie dies in einem Multicast-Protokoll garantieren?



Probleme mit FIFO-Broadcasts



- *Annahme*: Steuerelemente kommunizieren über FIFO-Broadcasts:



- “Irgendwie” kommt beim Beobachter die Reihenfolge durcheinander!

==> Falsche Schlussfolgerung des Beobachters:

“Aufgrund einer unbegreiflichen Pumpenaktivität wurde ein Leck erzeugt, wodurch schliesslich der Druck absank.”

Man sieht also:

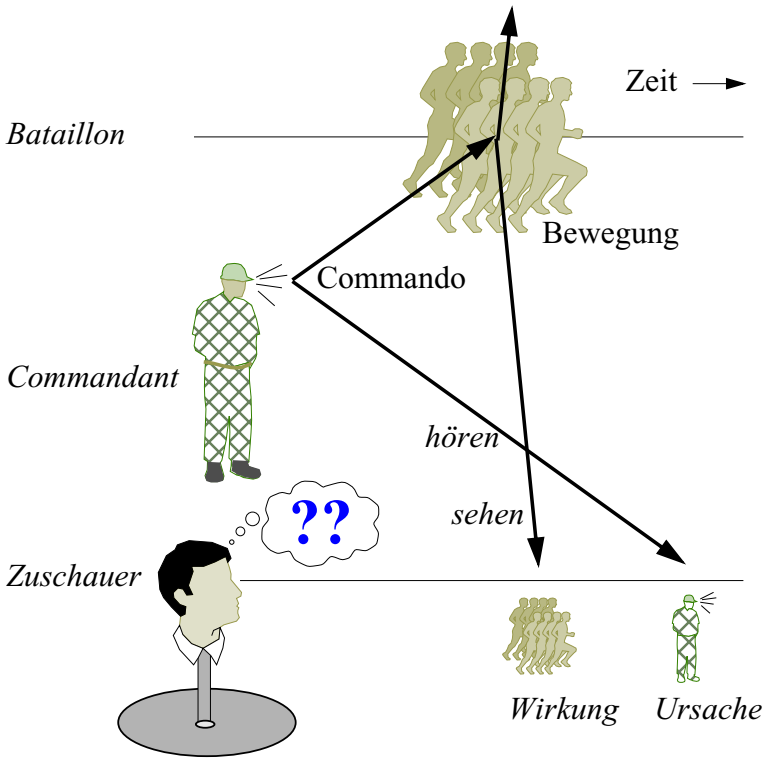
- FIFO-Reihenfolge reicht oft nicht aus, um Semantik zu wahren
- eine Nachricht *verursacht* oft das Senden einer anderen

Kausalität

Das “Broadcastproblem” ist nicht neu

- Licht- und Schallwellen werden in natürlicher Weise per Broadcast verteilt
- Wann handelt es sich dabei um FIFO-Broadcasts?
- Wie ist es mit dem Kausalitätserhalt?

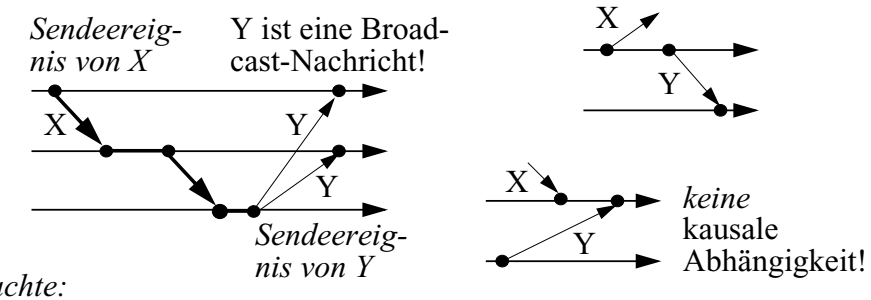
> Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, *ehe* er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der *Causalzusammenhänge* weiß er, daß die Bewegungen die *Wirkung* des gehörten Commandos sind, dieses also jenen *objectiv* vorangehen muß, und er wird sich sofort der Täuschung bewußt, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.
>
> *Christoph von Sigwart* (1830-1904) *Logik* (1889)



Kausale Broadcasts

Kausale Abhängigkeit zwischen Nachrichten (Def.):

- *Nachricht Y hängt kausal von Nachricht X ab,* wenn es im Raum-Zeit-Diagramm einen von links nach rechts verlaufenden Pfad gibt, der vom Sendeereignis von X zum Sendeereignis von Y führt



Beachte:

- Dies lässt sich bei geeigneter Modellierung auch abstrakter fassen (→ vgl. logische Zeit später in der Vorlesung und auch Vorlesung “Verteilte Algorithmen”)

Wahrung von Kausalität bei der Kommunikation:

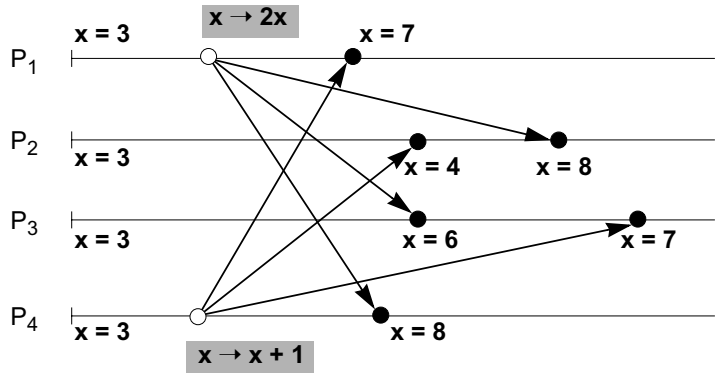
- *Kausale Reihenfolge (Def.):* Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben

Beachte: “causal order”

- “Kausale Reihenfolge” und “kausale Abhängigkeit” lassen sich insbesondere auch auf *Broadcasts* anwenden
- Kausale Reihenfolge *impliziert FIFO-Reihenfolge:* kausale Reihenfolge ist eine Art “globales FIFO”
- Das *Erzwingen* der kausalen Reihenfolge ist mittels geeigneter Algorithmen möglich (→ Vorlesung “Verteilte Algorithmen”, z.B. Verallgemeinerung der Sequenzzählermethode für FIFO)

Probleme mit kausalen Broadcasts ?

Beispiel: Aktualisierung einer replizierten Variablen x:



Problem: Statt *überall* 7 *oder* 8 als Ergebnis: Hier “beides”!

Konkrete Ursache des Problems:

- Broadcasts werden nicht überall “gleichzeitig” empfangen
- dies führt lokal zu verschiedenen Empfangsreihenfolgen

Abstrakte Ursache:

- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

- auch kausale Broadcasts haben keine “perfekte” Semantik
  (d.h. Illusion einer speicherbasierten Kommunikation)