

Ein Socket-Beispiel in C

- Verwendung von sockets in C erfordert u.a.
 - Header-Dateien mit C-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 connect 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 (Online-Dokumentation von UNIX oder Bücher)
- 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

- 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

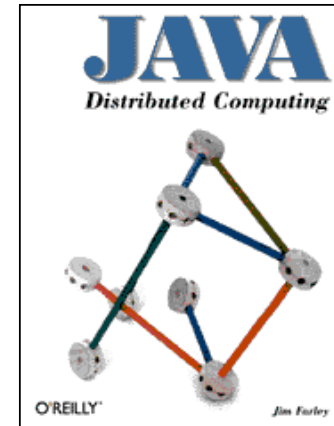
Port Nummer 7 sendet alles zurück

- 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, 1998, ISBN 1-56592-206-9)



- 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 Portnummer 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 Verbindungswünsche 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 DataInputStream 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 Methodenaufwurf 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)

Im Vergleich zu RPC: Nicht notw. Master-Slave, sondern peer-to-peer

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. (Vgl. "mobiler Code", "mobile Agenten", 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.getByAddress("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 vom Anwender verborgen benutzt

- Beispiel: zeilenweises Lesen einer Textdatei

- aber auch hier noch diverse Fehlerbedingungen abfangen!

```
// 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 gibt auch Möglichkeiten, Daten an eine URL zu senden (POST-Methode, z.B. an ein CGI-Skript)

- Ferner: Information über das Objekt ermitteln

- z.B. Grösse, letztes Änderungsdatum, HTTP-Header etc.

Übungsbeispiel: Ein Bookmark-Checker

```
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;
    }
}
```

Adressierung

```
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;
}
}
```

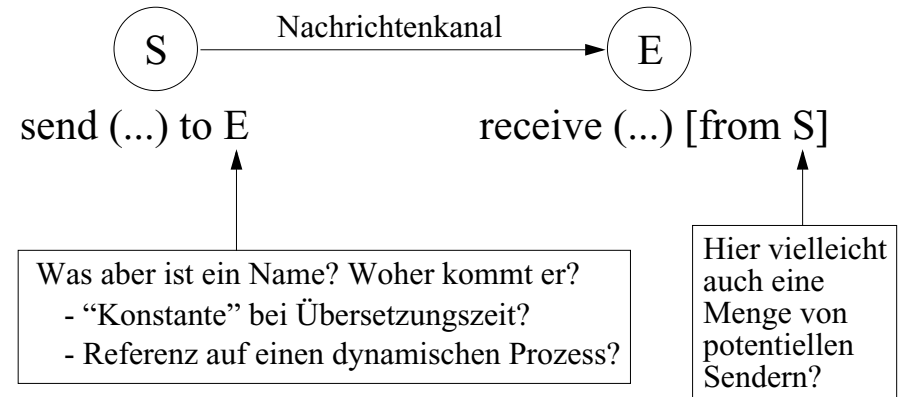
- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
 - ggf. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
 - ggf. mehrere Adressaten gleichzeitig (Broadcast, Multicast)
- *Empfänger* ist ggf. nicht bereit, jede beliebige Nachricht von jedem 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 auf Ethernet-Basis), 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

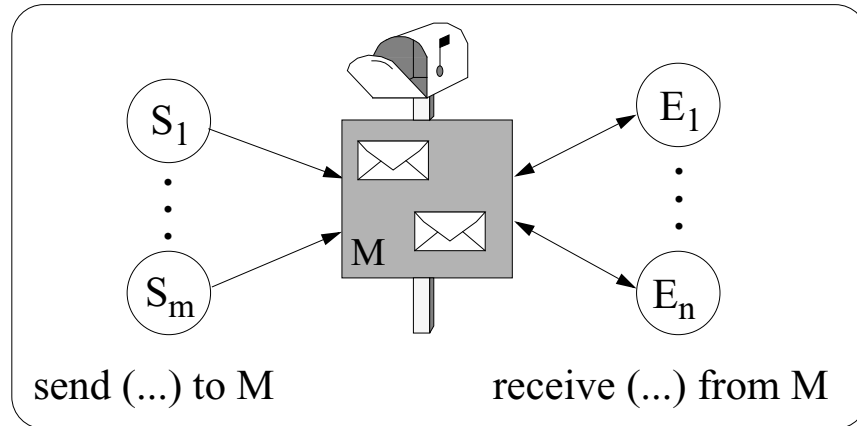
- *Direct Naming* (1:1-Kommunikation):



- 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

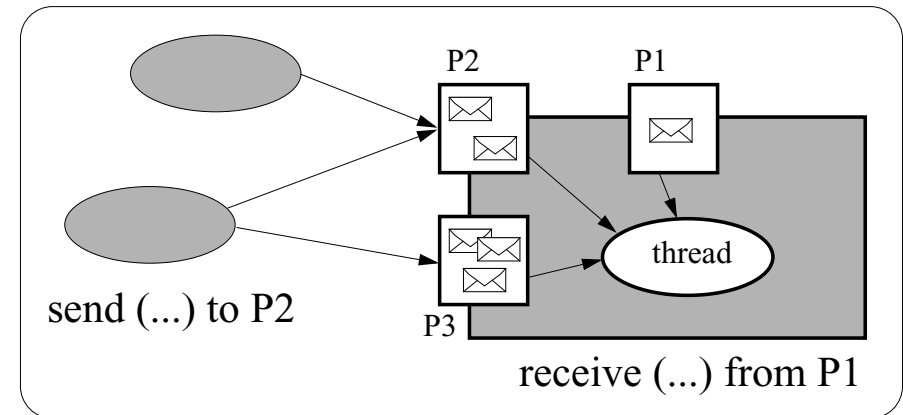
- m:n-Kommunikation möglich



- 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

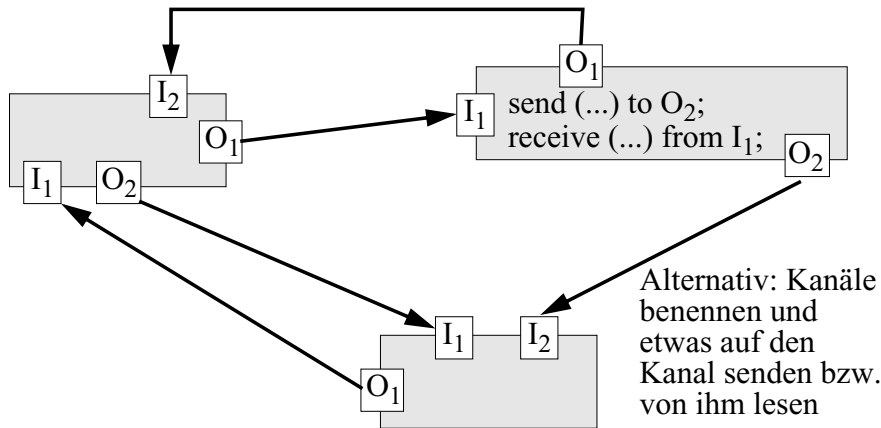


Pragmatische Aspekte (Sprachdesign etc.):

- 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?
 - unterstützt den selektiven Nachrichtempfang
- 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

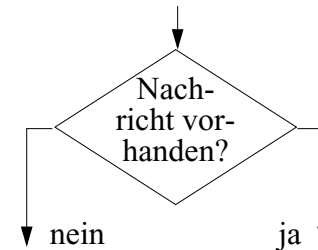


- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* (“Kanäle”) gewählt werden
- Dazu werden je ein in- und out-Port miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect p1 to p2**
 - denkbar sind auch broadcastfähige Kanäle
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen
- Größere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
- Kommunikationsbeziehung: wahlweise 1:1, n:1, 1:n, n:m

Konfigurationsphase

Varianten beim Empfangen von Nachrichten - Nichtblockierung

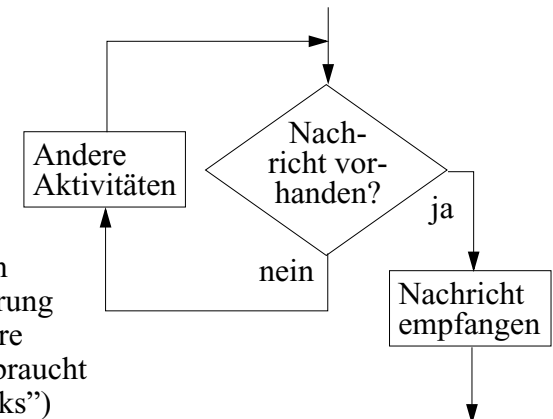
- Typischerweise ist ein “receive” blockierend
- Aber auch nichtblockierender Empfang denkbar:



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

- Aktives Warten: (“busy waiting”)

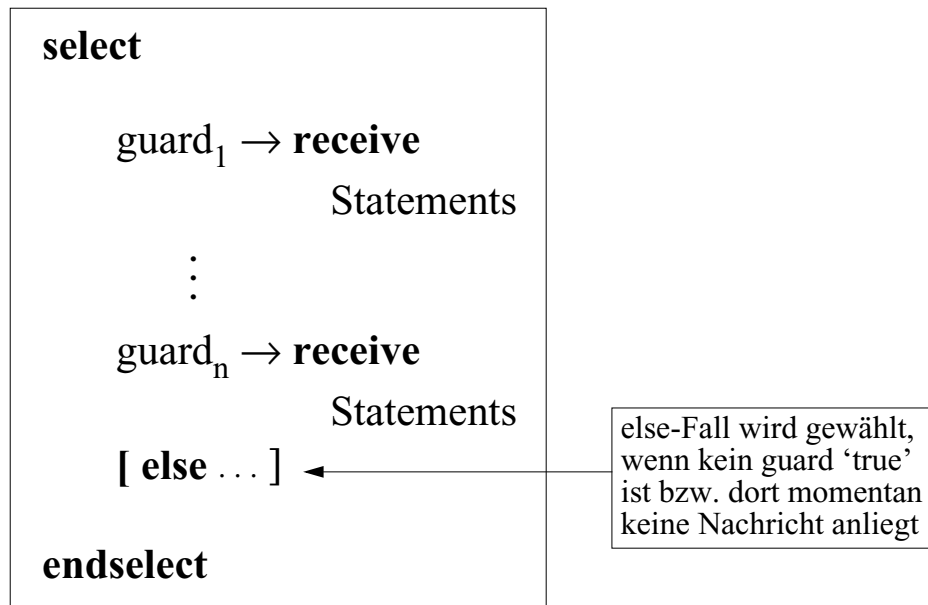
- Nachbildung des blockierenden Wartens wenn “andere Aktivitäten” leer
- Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)



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

Nichtblockierendes, alternatives Empfangen

- Sprachliche Realisierung z.B. so:



Zeitüberwacher Nachrichtenempfang

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

Quelle vielfältiger Probleme...

- Aktives Warten durch umschliessende while-Schleife

- im else-Fall könnte dann die while-Bedingung auf false gesetzt werden, falls das aktive Warten abgebrochen werden soll, oder es könnte mittels timer ("wait") eine kurze Zeit gewartet werden...
- else-Fall kann auch einfach das leere Statement enthalten

- Typischerweise blockierend, wenn else-Alternative ganz fehlt

- Analog ggf. auch für synchrones (!) *Senden* sinnvoll

- Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun "asynchron" an...

Zeitüberwacher Empfang

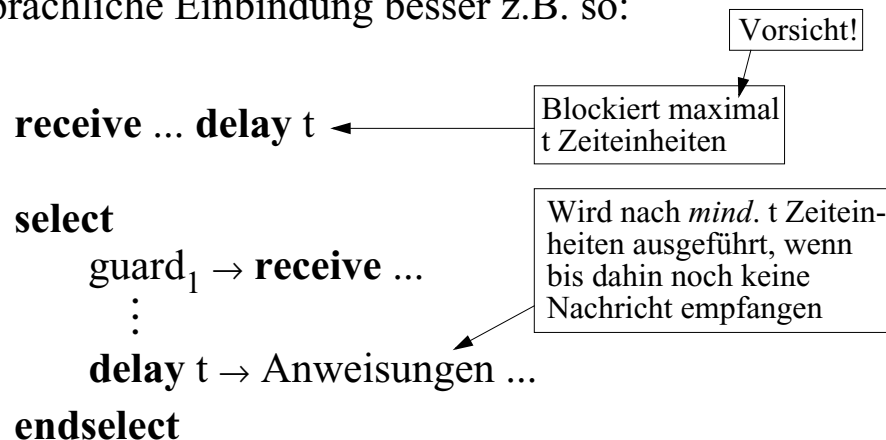
- Möglicher Realisierung:
 - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
 - Sprungziel könnte z.B. eine Unterbrechungs-routine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive
- “systemnahe”, unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

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

- Sprachliche Einbindung besser z.B. so:



- Vorteil bei der Anwendung:

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

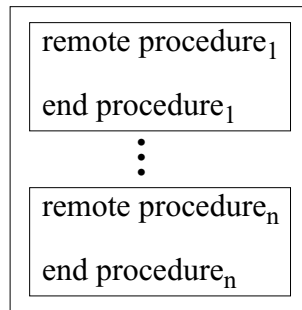
- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!
- Frage: ist “delay 0” äquivalent zu “else”?

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... im Empfänger?
- Vielleicht sogar mehrere Instanzen der gleichen Routine?

- Atomare Routinen?

- Wird eine aktive Routine ggf. unterbrochen, um eine andere aktivierte auszuführen?

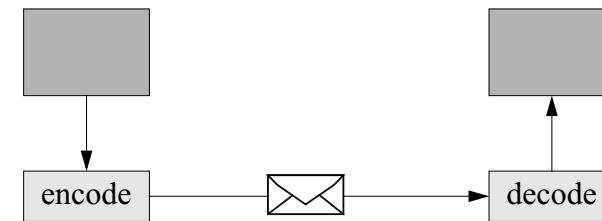
“Routine”

Kommunizierbare Datentypen?

- Werte von “klassischen” einfachen Datentypen
 - int, character, string, floating point,...

- Kompatibilität in heterogenen Systemen

- Grösse von int?
- Format von floating point?
- 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”