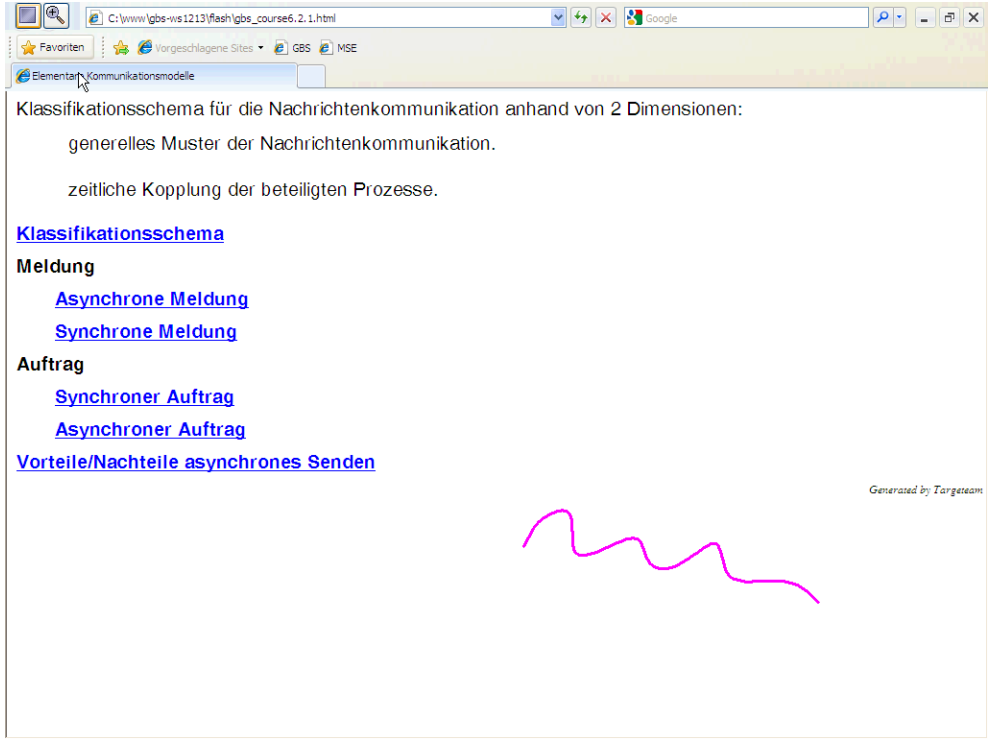
Script **generated by TTT**

Title: Grundlagen_Betriebssysteme (11.01.2013)

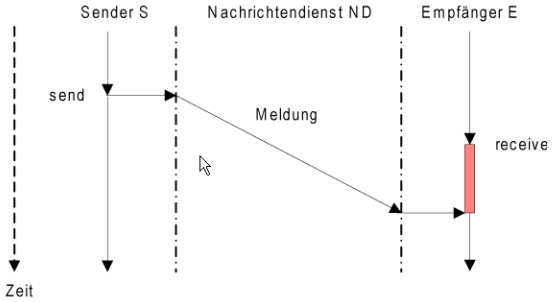
Date: Fri Jan 11 08:29:45 CET 2013

Duration: 89:34 min

Pages: 32



Sender wird lediglich bis zur Ablieferung der Meldung an das Nachrichtensystem (Kommunikationssystem) blockiert.



Nachrichtendienst des Betriebssystems puffert Nachricht;

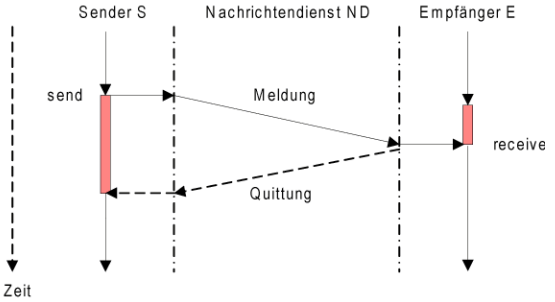
Sender S kann seine Ausführung fortsetzen, sobald Nachricht N in den Nachrichtenpuffer des ND eingetragen ist.

S wartet ***nicht*** , bis E die Nachricht empfangen hat.

Empfänger E zeigt durch receive an, dass er am Empfang der Nachricht N interessiert ist.

Empfänger wird blockiert, bis Sender Nachricht bereit stellt.

Sender und Empfänger von Meldungen sind zeitlich gekoppelt.



Rendezvous -Technik: Sender und Empfänger stellen vor Austausch der eigentlichen Meldung die Sende- und Empfangsbereitschaft her.

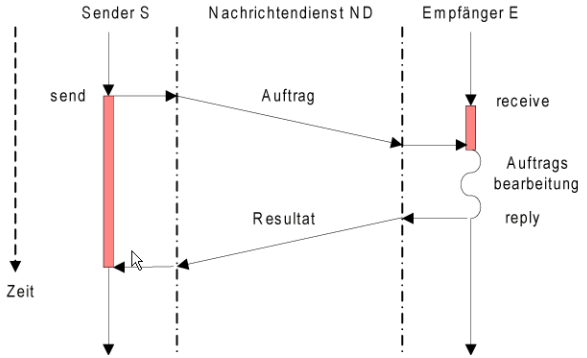
Elementare Kommunikationsmodelle

Klassifikationsschema für die Nachrichtenkommunikation anhand von 2 Dimensionen:

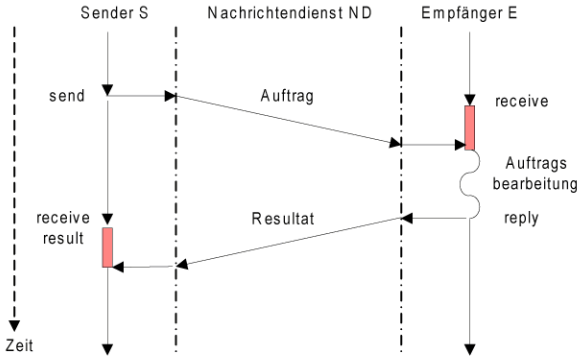
generelles Muster der Nachrichtenkommunikation.

zeitliche Kopplung der beteiligten Prozesse.

Bearbeitung der Nachricht durch Empfänger und Senden der Resultatnachricht sind Teil der Nachrichtentransaktion.



Auftrag und Resultat werden als Paar unabhängiger Meldungen verschickt.



Vorteile asynchrones Senden

nützlich für Realzeitanwendungen.

ermöglicht parallele Abarbeitung durch Sender und Empfänger.

anwendbar zum Signalisieren von Ereignissen.

Nachteile asynchrones Senden

Verwaltung des Nachrichtenpuffers durch BS erforderlich.

Benachrichtigung des Senders S im Fehlerfall und Behandlung von Fehlern ist problematisch.

Entwurf und Nachweis der Korrektheit des Systems ist schwierig.

Elementare Kommunikationsmodelle

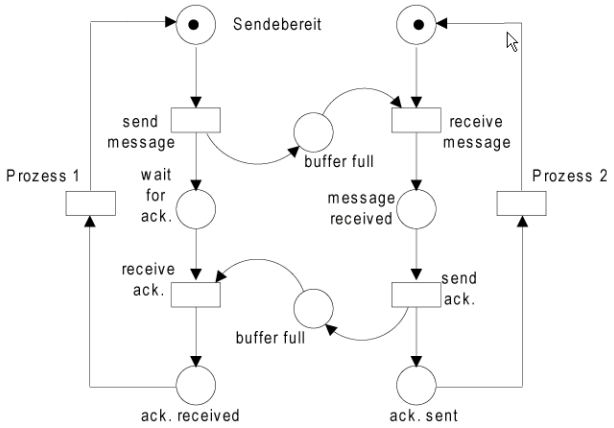
Klassifikationsschema für die Nachrichtenkommunikation anhand von 2 Dimensionen:

generelles Muster der Nachrichtenkommunikation.

zeitliche Kopplung der beteiligten Prozesse.

Modellierung durch ein Petrinetz

Petri-Netze dienen häufig zur Modellierung von Kommunikationsabläufen, sogenannten Kommunikationsprotokollen. Sie ermöglichen die Analyse der Protokolle, z.B. Erkennung von Verklemmungen.
Modellierung einer synchronen Kommunikation:



Problem: unendliches Warten

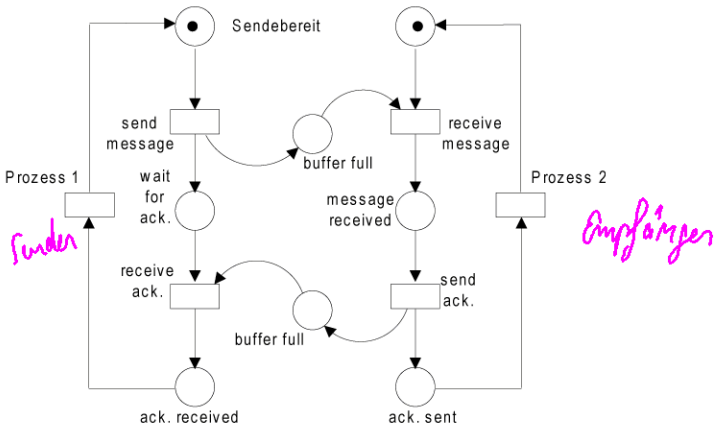
Pragmatische Lösung mit Hilfe von Timeouts

Sender bzw. Empfänger warten nur eine festgelegte Zeit, Sender: falls kein Acknowledgement (Quittung) eintrifft, z.B. erneutes Senden.

Probleme dabei? u.a. Duplikate müssen vom Empfänger erkannt werden; gesendete Nachrichten

Modellierung durch ein Petrinetz

Modellierung einer synchronen Kommunikation:



Problem: unendliches Warten

Pragmatische Lösung mit Hilfe von Timeouts

Sender bzw. Empfänger warten nur eine festgelegte Zeit, Sender: falls kein Acknowledgement (Quittung) eintrifft, z.B. erneutes Senden.

Probleme dabei? u.a. Duplikate müssen vom Empfänger erkannt werden; gesendete Nachrichten kommen zu spät an, sind veraltet etc.

Bei nachrichtenbasierter Prozessinteraktion tauschen Prozesse gezielt Informationen durch Verschicken und Empfangen von Nachrichten aus; ein Kommunikationssystem unterstützt an der Schnittstelle wenigstens die Funktionen `send` und `receive` .

Ports

Bisher bestand zwischen Sender und Empfänger eine feste Beziehung, die über Prozessidentifikatoren (z.B. Namen oder Nummer) hergestellt wurde. Nachteile

Prozessnummern ändern sich mit jedem Neustart.

Prozessnamen sind nicht eindeutig, z.B. falls Programm mehrmals gestartet wurde.

⇒ Deshalb Senden von Nachrichten an ***Ports*** . Sie stellen Endpunkte einer Kommunikation dar. Sie können bei Bedarf dynamisch eingerichtet und gelöscht werden. Dazu existieren folgende Funktionen:

```
portID = createPort();
deletePort(portID);
send(E.portID, message);
receive(portID, message);
```

ein Port ist mit dem Adressraum des zugehörigen Prozesses verbunden.

der Empfängerprozess kann sender-spezifische Ports einrichten.

ein Rechner mit einer IP-Adresse unterstützt mehrere tausend Ports.

der Name des Ports ist für einen Rechner eindeutig.

die Portnummern 1 - 1023 sind fest reserviert für bestimmte Protokolle (bzw. deren Applikationen).

Übersicht: fest zugeordnete Ports

Übersicht: fest zugeordnete Ports

Protokoll	*Port*	*Beschreibung*
FTP	21	Kommandos für Dateitransfer (get, put)
Telnet	23	interaktive Terminal-Sitzung mit entferntem Rechner
SMTP	25	Senden von Email zwischen Rechnern
time	37	Time-Server liefert aktuelle Zeit
finger	79	liefert Informationen über einen Benutzer
HTTP	80	Protokoll des World Wide Web
POP3	110	Zugang zu Email durch einen sporadisch verbundenen Client
RMI	1099	Zugang zum Registrieren von entfernten Java Objekten.

Bisher bestand zwischen Sender und Empfänger eine feste Beziehung, die über Prozessidentifikatoren (z.B. Namen oder Nummer) hergestellt wurde. Nachteile

Prozessnummern ändern sich mit jedem Neustart.

Prozessnamen sind nicht eindeutig, z.B. falls Programm mehrmals gestartet wurde.

⇒ Deshalb Senden von Nachrichten an ***Ports*** . Sie stellen Endpunkte einer Kommunikation dar. Sie können bei Bedarf dynamisch eingerichtet und gelöscht werden. Dazu existieren folgende Funktionen:

```
portID = createPort();
deletePort(portID);
send(E.portID, message);
receive(portID, message);
```

ein Port ist mit dem Adressraum des zugehörigen Prozesses verbunden.

der Empfängerprozess kann sender-spezifische Ports einrichten.

ein Rechner mit einer IP-Adresse unterstützt mehrere tausend Ports.

der Name des Ports ist für einen Rechner eindeutig.

die Portnummern 1 - 1023 sind fest reserviert für bestimmte Protokolle (bzw. deren Applikationen).

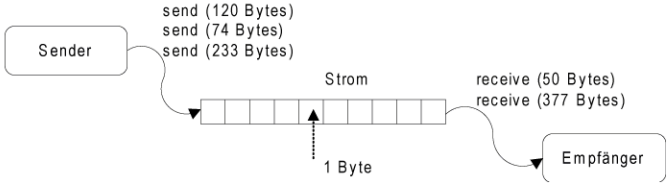
Übersicht: fest zugeordnete Ports

Generated by Targeteam

Nachrichtenbasierte Kommunikation

Bei nachrichtenbasierter Prozessinteraktion tauschen Prozesse gezielt Informationen durch Verschicken und Empfangen von Nachrichten aus; ein Kommunikationssystem unterstützt an der Schnittstelle wenigstens die Funktionen `send` und `receive` .

Ströme (engl. streams) sind eine Abstraktion von Kanälen. Sie verdecken die tatsächlichen Nachrichtengrenzen.



BS-Dienste: Verbindungsauf- und -abbau, schreiben in Strom, lesen aus Strom.

Dienste für Dateizugriffe oder Zugriffe auf Geräte: spezielle Ausprägung der stromorientierten Kommunikation.

I/O in Java basiert auf Ströme.

Klasse `java.io.OutputStream` zum Schreiben von Daten

Klasse `java.io.InputStream` zum Lesen von Daten

Spezialisierungen z.B. durch FileOutputStream, BufferedOutputStream oder FileInputStream.

Realisierung von Strömen über Pipe-Konzept (z.B. in Unix, Windows); Strom zwischen 2 Kommunikationspartnern

unidirektional

FIFO-artiger Datentransfer mit Operationen: open pipe, read, write.

gepuffert und zuverlässig.

Pipes können als Dateien ohne Plattenbeteiligung betrachtet werden.

Ordnung der Zeichen bleibt erhalten (Zeichenstrom)

Blockieren bei voller Pipe (write) und leerer Pipe (read)

Beispiele für Unix Pipes

ls -l | head

grep "name" datei.txt | sort | more

cat datei.txt | awk '{print $1}' | more

Prinzipieller Aufbau

Beispielnutzung einer Pipe

Named Pipes

Der pipe-Systemaufruf erzeugt ein i-node Objekt sowie 2 File-Objekte. Damit können Prozesse mit read() und write() Systemaufrufen darauf zugreifen.

```
struct pipe_inode_info {
    wait_queue_head_t wait; /* Warteschlange blockierter Prozesse, die auf
    Pipe zugreifen wollen */
    char *base; /* Adresse des Kernel Buffers */
    unsigned int len; /* Anzahl geschriebener, aber noch nicht gelesener
    Bytes */
    unsigned int start; /* Leseposition im Buffer */
    unsigned int readers; /* Anzahl lesender Prozesse */
    unsigned int writers; /* Anzahl schreibender Prozesse */
    unsigned int waiting_writers;
    unsigned int r_counter; /* Anzahl der Prozesse, die auf neue Zeichen in
    der Pipe warten */
    unsigned int w_counter; /* Anzahl der Prozesse, die Zeichen in die Pipe
    schreiben möchten */
}
```

Zugriffe auf Pipe müssen intern über Semaphore geregelt werden.

Die Pipe wird im Vaterprozess angelegt, der anschließend 2 Kindprozesse startet, die über eine Pipe kommunizieren.

Systemaufruf `dup()` ermöglicht Umlenkung von Standard-Ein/-Ausgabe auf Pipe

```
main (int argc, char *argv[]) {
    int fhandle[2]; int rc;
    pipe(fhandle); /* erzeugt Pipe*/
    rc = fork();
    if (rc ==0) {
       close(0); /* Schließen Standard Eingabe */
       dup(fhandle[0]); /* Umlenken von fhandle[0] auf Standard Eingabe */
       close(fhandle[0]); /* Schließen der nicht mehr benötigten fhandle[0]
       */
          /* der nun gestartete Prozess liest bei Zugriff auf Standard
          Eingabe aus der Pipe */
       rc = execl("kind1", "Kind 1", "\0");
    } else {
       rc = fork(); /* Vaterprozess */
       if (rc == 0) {
          close(1); /* Schließen Standard Ausgabe */
          dup(fhandle[1]); /* Umlenken von fhandle[1] auf Standard Ausgabe
          */
          close(fhandle[1]); /* Schließen der nicht mehr benötigten
```

Beispielnutzung einer Pipe

```
    if (rc ==0) {
       close(0); /* Schließen Standard Eingabe */
       dup(fhandle[0]); /* Umlenken von fhandle[0] auf Standard Eingabe */
       close(fhandle[0]); /* Schließen der nicht mehr benötigten fhandle[0]
       */
          /* der nun gestartete Prozess liest bei Zugriff auf Standard
          Eingabe aus der Pipe */
       rc = execl("kind1", "Kind 1", "\0");
    } else {
       rc = fork(); /* Vaterprozess */
       if (rc == 0) {
          close(1); /* Schließen Standard Ausgabe */
          dup(fhandle[1]); /* Umlenken von fhandle[1] auf Standard Ausgabe
          */
          close(fhandle[1]); /* Schließen der nicht mehr benötigten
          fhandle[1] */
             /* der nun gestartete Prozess schreibt bei Zugriff auf
             Standard Ausgabe in die Pipe; zwischen den Kindern ist
             dadurch eine unidirektionale Kommunikation mittels Pipes
             erzeugt worden */
          rc = execl("kind2", "Kind 2", "\0");
    }
    sleep(1);
    exit(0);
```

Kinderprozesse

Kind 1 liest aus der Pipe, während Kind 2 in die Pipe schreibt.

```
/* Kind 1 */
main (int argc, char *argv[]) {
     char buffer[100];
     read(0, buffer, 28); /* Prozess liest von Pipe */
     write(1, buffer, 28); /* schreibt auf Standard Aus */
     exit(0);
}

/* Kind 2 */
main (int argc, char *argv[]) {
     write(1, "Dies erscheint in der Queue\n", 28);
     exit(0);
}
```


Beispielnutzung einer Pipe

Die Pipe wird im Vaterprozess angelegt, der anschließend 2 Kindprozesse startet, die über eine Pipe kommunizieren.

Systemaufruf `dup()` ermöglicht Umlenkung von Standard-Ein/-Ausgabe auf Pipe

```
main (int argc, char *argv[]) {
    int fhandle[2]; int rc;
    pipe(fhandle); /* erzeugt Pipe*/
    rc = fork();
    if (rc ==0) {
       close(0); /* Schließen Standard Eingabe */
       dup(fhandle[0]); /* Umlenken von fhandle[0] auf Standard Eingabe */
       close(fhandle[0]); /* Schließen der nicht mehr benötigten fhandle[0]
       */
          /* der nun gestartete Prozess liest bei Zugriff auf Standard
          Eingabe aus der Pipe */
       rc = execl("kind1", "Kind 1", "\0");
    } else {
       rc = fork(); /* Vaterprozess */
       if (rc == 0) {
          close(1); /* Schließen Standard Ausgabe */
          dup(fhandle[1]); /* Umlenken von fhandle[1] auf Standard Ausgabe
          */
          close(fhandle[1]); /* Schließen der nicht mehr benötigten
```

Pipes

Realisierung von Strömen über Pipe-Konzept (z.B. in Unix, Windows); Strom zwischen 2 Kommunikationspartnern

- unidirektional
- FIFO-artiger Datentransfer mit Operationen: open pipe, read, write.
- gepuffert und zuverlässig.
- Pipes können als Dateien ohne Plattenbeteiligung betrachtet werden.

- Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
- Blockieren bei voller Pipe (write) und leerer Pipe (read)

Beispiele für Unix Pipes

- ls -l | head
- grep "name" datei.txt | sort | more
- cat datei.txt | awk '{print $1}' | more

Named Pipes

Problem: nur Prozesse, die über `fork()` eng miteinander verwandt sind, können im Beispielprogramm kommunizieren.

Einführung von Named Pipes, die mittels `mknod` erzeugt werden.

ermöglicht die Kommunikation zwischen Prozessen, die nicht miteinander verwandt sind.

```
main (int argc, char *argv[]) {
    int rc;
    mknod("myfifo", 0600, S_IFIFO);
    if (rc == 0) {
       int fd;
       fd = open("myfifo", O_WRONLY);
       write(fd, "Satz 1\n", 7);
       close(fd);
       exit(0)
    } else {
       int fd;
       char block[20];
       fd = open("myfifo", O_RDONLY);
       rc = read(fd, block, 7);
       write(1, ":", 1);
       if (rc > 0) { write(1, block, rc); 0
```

Named Pipes

ermöglicht die Kommunikation zwischen Prozessen, die nicht miteinander verwandt sind.

```
main (int argc, char *argv[]) {
    int rc;
    mknod("myfifo", 0600, S_IFIFO);
    if (rc == 0) {
       int fd;
       fd = open("myfifo", O_WRONLY);
       write(fd, "Satz 1\n", 7);
       close(fd);
       exit(0)
    } else {
       int fd;
       char block[20];
       fd = open("myfifo", O_RDONLY);
       rc = read(fd, block, 7);
       write(1, ":", 1);
       if (rc > 0) { write(1, block, rc); 0
       else printf("Fehler: %d\n", rc);
       close(fd);
       exit(0)
    }
```

Prozesskommunikation

Disjunkte Prozesse, d.h. Prozesse, die völlig isoliert voneinander ablaufen, stellen eher die Ausnahme dar. Häufig finden Wechselwirkungen zwischen den Prozessen statt ⇒ Prozesse interagieren. Die Unterstützung der Prozessinteraktion stellt einen unverzichtbaren Dienst dar.

Fragestellungen

Dieser Abschnitt beschäftigt sich mit den Mechanismen von Rechensystemen zum Austausch von Informationen zwischen Prozessen.

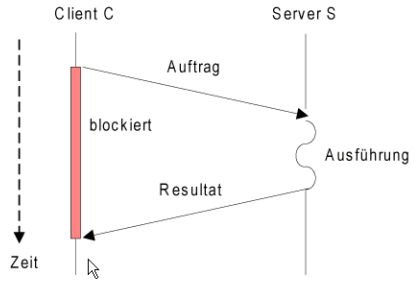
Kommunikationsarten.

nachrichtenbasierte Kommunikation, insbesondere Client-Server-Modell.

Netzwerkprogrammierung auf der Basis von Ports und Sockets.

Client-Server Modell basiert i.a. auf der Kommunikationsklasse der synchronen Aufträge. ***Server*** stellen Dienste zur Verfügung, die von vorher unbekannten ***Clients*** in Anspruch genommen werden können.

Client-Server Architektur



Beispiele für Dienste (Services), die mittels Server realisiert werden: Dateidienst, Zeitdienst, Namensdienst.

Definition: Client

Ein Client ist eine Anwendung, die auf einer Clientmaschine läuft und i.a. einen Auftrag initiiert, und die den geforderten Dienst von einem Server erhält.

Clients sind meist a-priori nicht bekannt.

Definition: Server

Ein Server ist ein Subsystem, das auf einer Servermaschine läuft und einen bestimmten Dienst für a-priori unbekannte Clients zur Verfügung stellt.

Server sind dedizierte Prozesse, die kontinuierlich folgende Schleife abarbeiten.

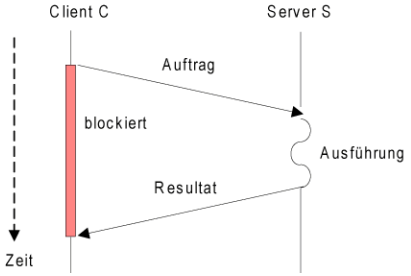
```
while (true) {
   receive (empfangsport, auftrag);
   führe Auftrag aus und erzeuge Antwortnachricht;
   bestimme sendeport für Antwortnachricht;
   send (sendeport, resultat);
}
```

Client und Server kommunizieren über Nachrichten, wobei beide Systeme auf unterschiedlichen Rechnern ablaufen können und über ein Netz miteinander verbunden sind.

Client-Server-Modell

Client-Server Modell basiert i.a. auf der Kommunikationsklasse der synchronen Aufträge. ***Server*** stellen Dienste zur Verfügung, die von vorher unbekannten ***Clients*** in Anspruch genommen werden können.

Client-Server Architektur



Definitionen

Beispiele für Dienste (Services), die mittels Server realisiert werden: Dateidienst, Zeitdienst, Namensdienst.

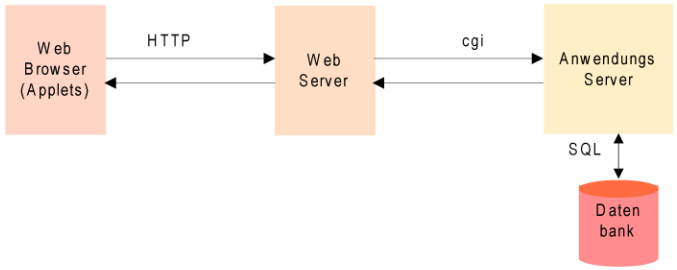
Multi-Tier

Peer-to-Peer Computing

Multi-Tier

ein System kann sowohl Client als auch Server sein.

Beispiel



Peer-to-Peer Computing

Es findet keine Unterscheidung zwischen Client und Server statt

⇒ Systeme übernehmen beide Rollen, d.h. sie sind sowohl Client als auch Server (Peers).

Damit soll der Server-Flaschenhals elimiert werden.

Teilnahme an einem P2P-System erfordert explizites Beitreten.

Feststellung, welche Dienste im P2P-Systeme verfügbar sind

ein Knoten registriert seine Dienste bei einem zentrale Directory-Service.

es existiert ein Discovery-Protokoll, mit Hilfe dessen ein Client-Knoten eine Liste möglicher Dienste im P2P-System erstellen kann.

Beispielsysteme sind Napster und Gnutella.

P2P-Systeme wurde in der Vergangenheit häufig im Zusammenhang mit Musik-Tauschbörsen eingesetzt ⇒ rechtliche Probleme.

Eine ***verteilte Anwendung*** ist eine Anwendung A, dessen Funktionalität in eine Menge von kooperierenden Teilkomponenten A_1 , .., A_n , $n \in IN$, $n > 1$ zerlegt ist;

> Jede Teilkomponente umfasst Daten (interner Zustand) und Operationen, die auf den internen Zustand angewendet werden.
>
> Teilkomponenten A_i sind autonome Prozesse, die auf verschiedenen Rechensystemen ausgeführt werden können. Mehrere Teilkomponenten können demselben Rechensystem zugeordnet werden.
>
> Teilkomponenten A_i tauschen über das Netz untereinander Informationen aus.

Die Teilkomponenten können z.B. auf der Basis des Client-Server Modells realisiert werden.