Resümee (1b)

- Cloud-Computing
  - motiviert durch schnelle, ubiquitäre, verlässliche Netze
  - Trend: „alles" irgendwo im Netz
  - Beispiele für Cloud-Datenzentren
  - wirtschaftliche Effekte: Skaleneffekte, Spot-Markt

Charakteristika und Problemaspekte verteilter Systeme

- Räumliche Separation und Autonomie der Komponenten führen (relativ zu zentralen Systemen) zu neuen Problemen:
  - partielles Fehlverhalten möglich (statt totaler "Absturz")
  - fehlender globaler Zustand / exakt synchronisierte Zeit
  - evtl. Inkonsistenzen (z.B. zwischen Datei und Verzeichnis / Index)
- Typischerweise Heterogenität in Hard- und Software
- Hohe Komplexität
- Sicherheit (Vertraulichkeit, Authentizität, Integrität, Verfügbarkeit,...)
  - notwendiger als in isolierten Einzelsystemen
  - aber schwieriger zu gewährleisten (mehr Angriffspunkte)

Gegenmittel?

- Gute Werkzeuge ("Tools") und Methoden
  - z.B. Frameworks und Middleware als Software-Infrastruktur
- Abstraktion als Mittel zur Beherrschung von Komplexität
  - z.B. Schichten (Kapselung, virtuelle Maschinen) oder
  - Modularisierung (z.B. Services)
- Adäquate Modelle, Algorithmen, Konzepte
  - zur Beherrschung der Phänomene rund um die Verteiltheit

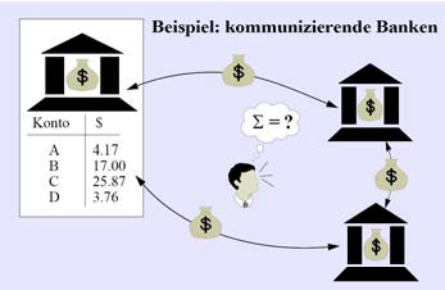
- Ziel der Vorlesung
  - Verständnis der grundlegenden Phänomene
  - Kenntnis von geeigneten Konzepte und Methoden

Einige konzeptionelle Probleme und Phänomene verteilter Systeme

1) Schnappschussproblem
2) Phantom-Deadlocks
3) Uhrensynchronisation
4) Kausaltreue Beobachtungen
5) Geheimnisvereinbarung über unsichere Kanäle

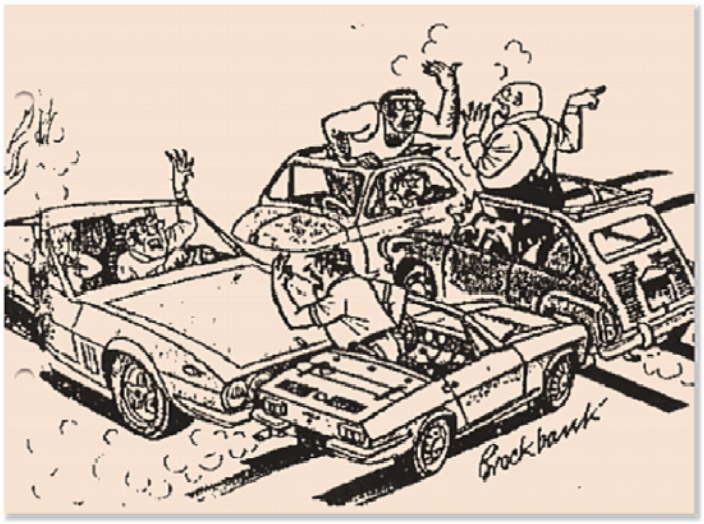
- Dies sind einige einfach zu erläuternde Probleme und Phänomene – natürlich gibt es noch viel mehr und viel komplexere Probleme konzeptioneller wie praktischer Art
- Achtung: Manches davon wird nicht hier, sondern in anderen Vorlesungen (z.B. "Verteilte Algorithmen") eingehender behandelt!

Ein erstes Beispiel: Wieviel Geld ist in Umlauf?



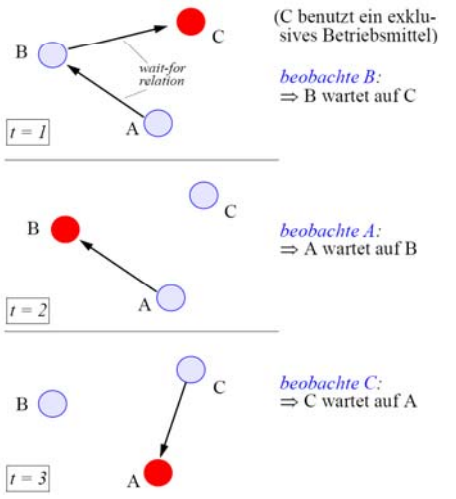
- Hier: konstante Geldmenge
- Ständige Transfers zwischen den Banken
- Niemand hat eine globale Sicht
- Es gibt keine gemeinsame Zeit ("Stichtag")
- Anwendung: z.B. verteilte Datenbank-Sicherungspunkte

Ein zweites Beispiel: Das Deadlock-Problem

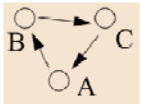


Ein zweites Beispiel: Das Deadlock-Problem

Phantom-Deadlocks

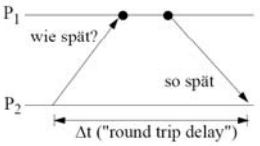


- Aus den Einzelbeobachtungen darf man nicht schliessen:
- A wartet auf B und B wartet auf C und C wartet auf A

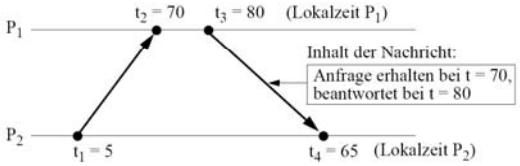


- Diese zyklische Wartebedingung wäre tatsächlich ein Deadlock
- Die Einzelbeobachtungen fanden hier aber zu unterschiedlichen Zeiten statt
- Lösung (nur echte Deadlocks erkennen) ohne Uhren, globale Zeit, Zeitstempel etc.?

Ein drittes Problem: Uhrensynchronisation



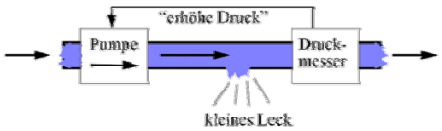
Laufzeit der Nachrichten variabel (und oft verschieden für Hin- und Rückweg)

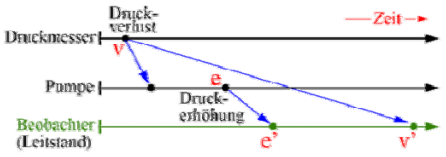


- Uhren gehen nicht unbedingt gleich schnell!
  - Gilt wenigstens "Beschleunigung ≈ 0", d.h. ist konstanter Drift gerechtfertigt?
- Wie kann man den Offset der Uhren ermitteln oder zumindest approximieren?

Ein viertes Problem: (nicht) kausaltreue Beobachtungen

- Gewünscht: Eine Ursache stets vor ihrer (u.U. indirekten) Wirkung beobachten



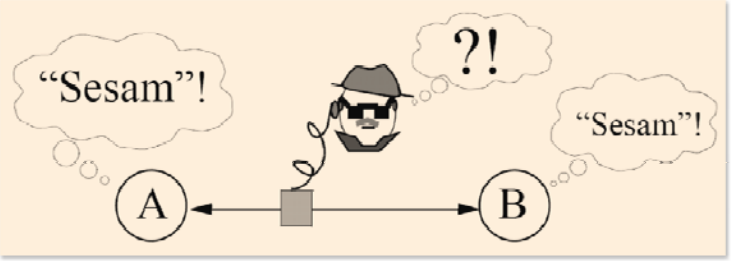


Falsche Schlussfolgerung des Beobachters:

Es erhöhte sich der Druck (aufgrund einer unbegründeten Aktivität der Pumpe), es kam zu einem Leck, was durch den abfallenden Druck angezeigt wird.

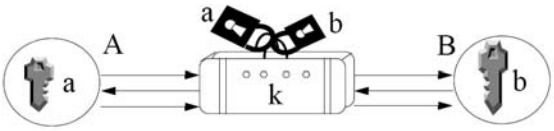
Und noch ein Problem: Verteilte Geheimnisvereinbarung

- Problem: A und B wollen sich über einen unsicheren Kanal auf ein gemeinsames geheimes Passwort einigen



Verteilte Geheimnisvereinbarung (2)

- Idee: Vorhängeschlösser um eine sichere Truhe:



1. A denkt sich Passwort k aus und tut es in die Truhe.
2. A verschliesst die Truhe mit einem Schloss a.
3. A sendet die so verschlossene Truhe an B.
4. B umschliesst das ganze mit seinem Schloss b.
5. B sendet alles doppelt verschlossen an A zurück.
6. A entfernt Schloss a.
7. A sendet die mit b verschlossene Truhe wieder an B.
8. B entfernt sein Schloss b.

- Problem: Lässt sich das so softwaretechnisch realisieren?

Kommunikation

Kooperation durch Informationsaustausch

- Prozesse sollen kooperieren, daher untereinander Information austauschen können
  - falls vorhanden, evtl. über einen gemeinsamen globalen Speicher (dieser kann physisch oder evtl. nur logisch existieren als „virtual shared memory“)
  - oder mittels Nachrichten:
    Daten an eine entfernte Stelle kopieren

Kommunikation

Notwendig, damit Kommunikation klappt, ist jedenfalls:

1. ein dazwischenliegendes physikalisches Medium
   - z.B. elektrische Signale in Kupferkabeln
2. einheitliche Verhaltensregeln
   - Kommunikationsprotokolle
3. gemeinsame Sprache und gemeinsame Semantik
   - gleiches Verständnis der Bedeutung von Kommunikationskonstrukten und -regeln

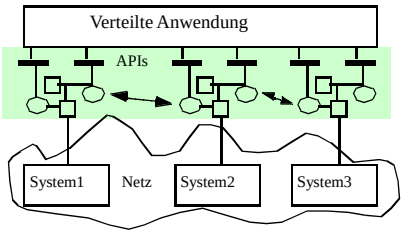
Also trotz Verteiltheit gewisse gemeinsame Aspekte!

Nachrichtenbasierte Kommunikation

- send → receive
- Implizite Synchronisation: Senden vor Empfangen
  - Empfänger erfährt, wie weit der Sender mindestens ist
- Nachrichten sind dynamische Betriebsmittel
  - verursachen Aufwand und müssen verwaltet werden

Message Passing System (1)

- Organisiert den Nachrichtentransport
- Bietet Kommunikationsprimitive (als APIs) an
  - z.B. send (...) bzw. receive (...)
  - evtl. auch ganze Bibliothek verschiedener Kommunikationsdienste
  - verwendbar mit gängigen Programmiersprachen (oft zumindest C)



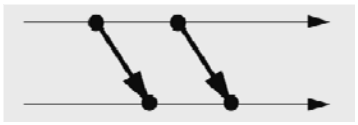
- Besteht aus Hilfsprozessen, Pufferobjekten, …
- Verbirgt Details des zugrundeliegenden Netzes bzw. Kommunikationssubsystems

Message Passing System (2)

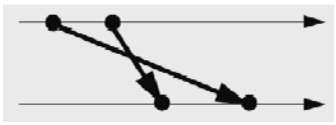
- Verwendet vorhandene Netzprotokolle und implementiert damit eigene, „höhere" Protokolle
- Garantiert (je nach „Semantik") gewisse Eigenschaften
  - z.B. Reihenfolgeerhalt oder Prioritäten von Nachrichten
- Abstrahiert von Implementierungsaspekten
  - z.B. Geräteadressen oder Längenrestriktionen von Nachrichten etc.
- Maskiert gewisse Fehler
  - mit typischen Techniken zur Erhöhung des Zuverlässigkeitsgrades: Timeouts, Quittungen, Sequenznummern, Wiederholungen, Prüfsummen, fehlerkorrigierende Codes,…
- Verbirgt Heterogenität unterschiedlicher Systemplattformen
  - erleichtert damit Portabilität von Anwendungen

Ordnungserhalt von Nachrichten: FIFO

- Manchmal werden vom Kommunikationssystem Garantien bzgl. Nachrichtenreihenfolgen gegeben
- Eine mögliche Garantie stellt FIFO (First-In-First-Out) dar: Nachrichten zwischen zwei Prozessen überholen sich nicht: Empfangsreihenfolge = Sendereihenfolge



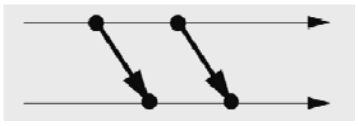
FIFO



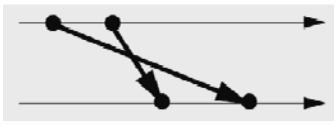
kein FIFO

Ordnungserhalt von Nachrichten: FIFO

- Manchmal werden vom Kommunikationssystem Garantien bzgl. Nachrichtenreihenfolgen gegeben
- Eine mögliche Garantie stellt FIFO (First-In-First-Out) dar: Nachrichten zwischen zwei Prozessen (also auf dem Kommunikationskanal zwischen Sender und Empfänger) überholen sich nicht: Empfangsreihenfolge = Sendereihenfolge



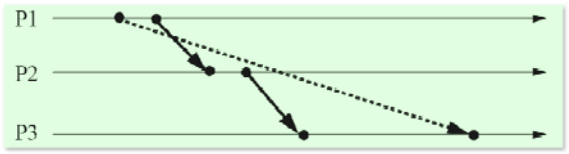
FIFO



kein FIFO

Ordnungserhalt von Nachrichten: kausale Ordnung

- FIFO verbietet allerdings nicht, dass Nachrichten evtl. indirekt (über eine Kette anderer Nachrichten) überholt werden



Zwar FIFO, aber nicht kausal geordnet

- Möchte man auch dies haben, so muss die Kommunikation kausal geordnet sein (Anwendungszweck?)
  - keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege („Dreiecksungleichung")
  - entspricht einer „Globalisierung" von FIFO auf mehrere Prozesse
  - Denkübung: Wie garantiert (d.h. implementiert) man kausale Ordnung auf einem System ohne Ordnungsgarantie?

Prioritäten von Nachrichten? (1)

- Achtung: Semantik ist a priori nicht ganz klar:
  - Soll (kann?) das Transportsystem Nachrichten höherer Priorität bevorzugt (=?) befördern?
  - Können (z.B. bei fehlender Pufferkapazität) Nachrichten niedrigerer Priorität überschrieben werden?
  - Wie viele Prioritätsstufen gibt es?
  - Sollen aus einer Mailbox (= Nachrichtenspeicher) immer zuerst Nachrichten mit höherer Priorität geholt werden?

Prioritäten von Nachrichten? (2)

- Mögliche Anwendungen:
  - Unterbrechen / abbrechen laufender Aktionen (→ Interrupt)
  - Aufbrechen von Blockaden
  - Out-of-Band-Signalisierung

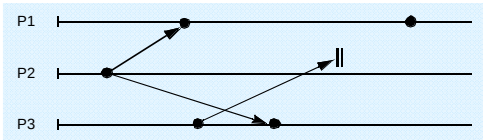
  } Durchbrechung der FIFO-Reihenfolge!

Vgl. auch Service-Klassen in Computernetzen: bei Rückstaus bei den Routern soll z.B. interaktiver Verkehr bevorzugt werden vor FTP etc.

- Vorsicht bei der Anwendung: Nur bei klarer Semantik verwenden; löst oft ein Problem nicht grundsätzlich!
  - Inwiefern ist denn eine (faule) Implementierung, bei der „eilige" Nachrichten (insgeheim) wie normale Nachrichten realisiert werden, tatsächlich nicht korrekt?

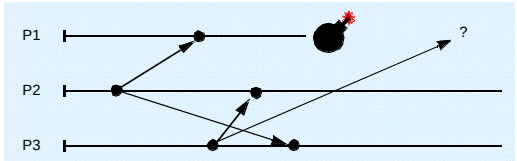
Fehlermodelle (1)

- Zweck: Klassifikation von Fehlermöglichkeiten; Abstraktion von den tieferliegenden spezifischen Ursachen
- Nachrichtenfehler beim Senden / Übertragen / Empfangen:



→ verlorene Nachricht

- Crash / Fail-Stop: Ausfall eines Prozessors:



→ Nicht mehr erreichbarer / mitspielender Prozess

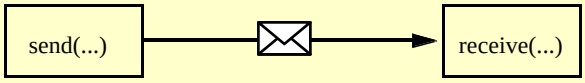
Fehlermodelle (2)

- Zeitfehler: Ereignis geschieht zu spät (oder zu früh)
- „Byzantinische" Fehler: Beliebiges Fehlverhalten, z.B.:
  - verfälschte Nachrichteninhalte
  - Prozess, der unsinnige Nachrichten sendet

(solche Fehler lassen sich nur teilweise, z.B. durch Redundanz, erkennen)

- Fehlertolerante Algorithmen bzw. Systeme müssen das „richtige" Fehlermodell berücksichtigen!
  - adäquate Modellierung der realen Situation / des Einsatzgebietes
  - Algorithmus verhält sich korrekt nur relativ zum Fehlermodell

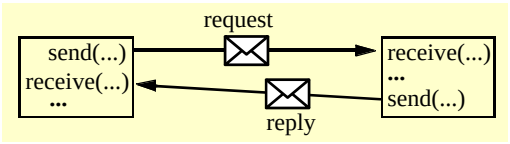
Mitteilungsorientierte Kommunikation



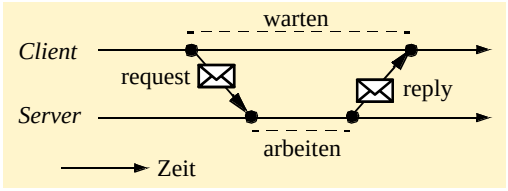
- Einfachste Form der Kommunikation
- Unidirektional
- Übermittelte Werte werden der Nachricht typw. als „Ausgabeparameter" beim send (-API) übergeben

Auftragsorientierte Kommunikation

Send und receive evtl. zu einem *einzigen* API zusammenfassen



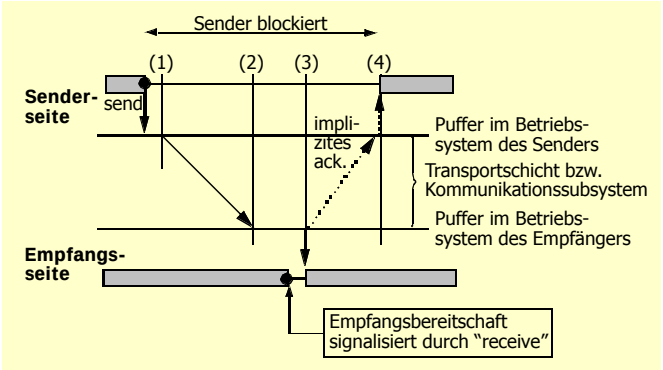
- Bidirektional
- Ergebnis des Auftrags wird als „Antwortnachricht" zurückgeschickt
- Auftraggeber („Client") wartet, bis Antwort eintrifft



Blockierendes Senden

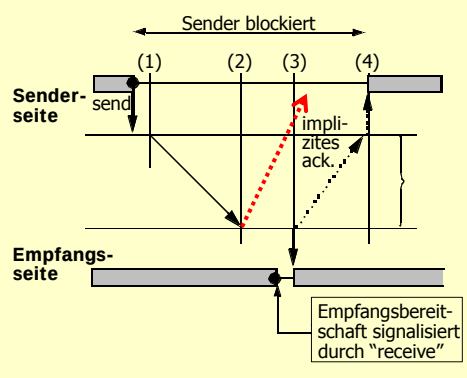
- Blocking send: Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert
  - Sender hat eine Garantie (Nachricht wurde zugestellt / empfangen)

was genau ist das?



Blockierendes Senden (2)

- Verschiedene Ansichten einer adäquaten Definition von „Abschluss der Transaktion“ aus Sendersicht:



Zeitpunkt 4 (automatische Bestätigung, dass der Empfänger receive ausgeführt hat) ist die höhere, anwendungsorientierte Sicht.

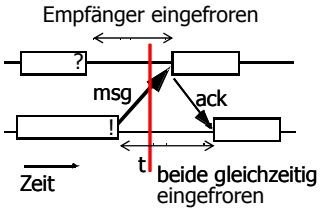
Falls eine Bestätigung bereits zum Zeitpunkt 2 geschickt wird, weiss der Sender nur, dass die Nachricht am Zielort zur Verfügung steht und der Sendepuffer wieder frei ist. (Vorher sollte der Sendepuffer nicht überschrieben werden, weil die Nachricht bei fehlerhafter Übertragung evtl. wiederholt werden muss.)

„gleich“ „zeitig“ Synchrone Kommunikation

- Idealisierung:
  Send und receive geschehen gleichzeitig
- Wodurch ist diese Idealisierung gerechtfertigt?
  (kann man auch mit einer Marssonde synchron kommunizieren?)

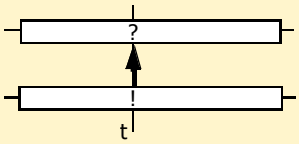
Synchrone Kommunikation mit „blocking send“ implementiert

a) „Receiver first“-Fall:



Bemerkung: „receive“ ist i.Allg. immer blockierend (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

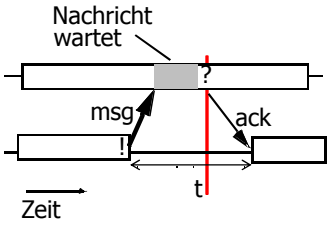
Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet („!“) und empfangen („?“) worden!

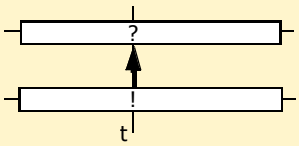
Synchrone Kommunikation mit „blocking send“ implementiert (2)

b) „Sender first“-Fall:



Zeit des Senders steht still → es gibt einen gemeinsamen Zeitpunkt t, wo die beiden Kommunikationspartner sich treffen → „Rendezvous“

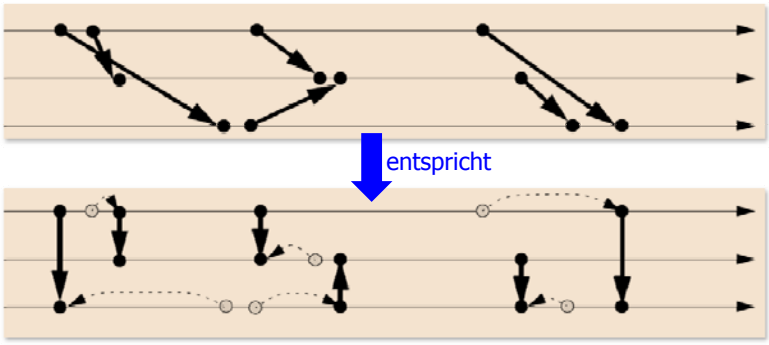
Idealisierung: senkrechte Pfeile in den Zeitdiagrammen



Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet („!“) und empfangen („?“) worden!

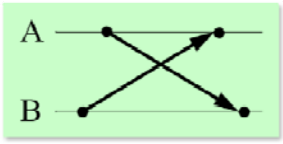
Virtuelle Gleichzeitigkeit

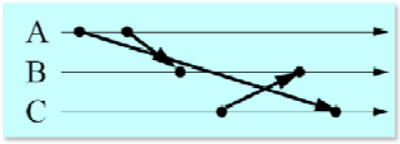
- Ein Ablauf, der synchrone Kommunikation benutzt, ist (bei Abstraktion von der Realzeit) durch ein äquivalentes Zeitdiagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen
  - nur stetige Deformation erlaubt („Gummiband-Transformation")



Virtuelle Gleichzeitigkeit?

- Folgendes geht nicht virtuell gleichzeitig (wieso?)





- Aber was geschieht eigentlich, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

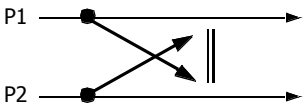
Mehr dazu (nur) für besonders Interessierte: B. Charron-Bost, F. Mattern, G. Tel: S*ynchronous, Asynchronous and Causally Ordered Communication.* Distributed Computing 9(4), pp. 173-191, www.vs.inf.ethz.ch/publ/

Deadlocks bei synchroner Kommunikation

```
P1:
   send (...) to P2;
   receive...
   ...
```

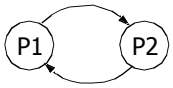
```
P2:
   send (...) to P1;
   receive...
   ...
```

In beiden Prozessen muss zunächst das send ganz ausgeführt werden, bevor es zu einem receive kommt



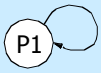
⇒ Kommunikationsdeadlock!

Zyklische Abhängigkeit der Prozesse voneinander: P1 wartet auf P2, und P2 wartet auf P1



„Wait-for-Graph"

Genauso tödlich:

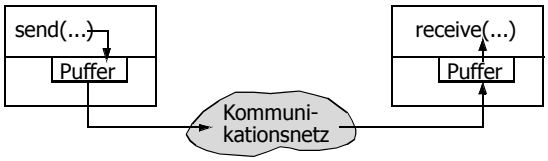



```
P1:
   send (...) to P1;
   receive...
   ...
```

Gleichnishaft entspricht der *synchronen* Kommunikation das Telefonieren, der *asynchronen* Kommunikation der Briefwechsel

Asynchrone Kommunikation

- No-wait send: Sender ist nur (kurz) bis zur lokalen Ablieferung der Nachricht an das Transportsystem blockiert
  - diese kurzzeitige Blockade sollten für die Anwendung transparent sein
- Jedoch i.Allg. länger blockiert, falls das System momentan keinen Pufferplatz für die Nachricht frei hat
  - Alternative: Sendenden Prozess nicht blockieren, sondern mittels „return value" über Misserfolg des send informieren



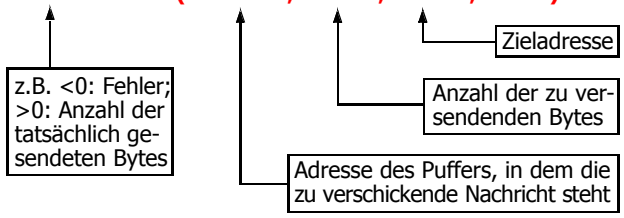
Asynchrone ↔ synchrone Kommunikation

- Vorteile asynchroner Kommunikation (im Vgl. zur syn. Komm.):
  - sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
  - stärkere Entkoppelung von Sender / Empfänger
  - höherer Grad an Parallelität möglich
  - geringere Gefahr von Kommunikationsdeadlocks
- Nachteile
  - Sender weiss nicht, ob / wann Nachricht angekommen ist
  - Debugging der Anwendung oft schwierig (wieso?)
  - System muss Puffer verwalten

Sendeoperationen in der Praxis

- Es gibt Kommunikationsbibliotheken, deren Dienste von verschiedenen Programmiersprachen (z.B. C) aus aufgerufen werden können
  - z.B. MPI (Message Passing Interface) { Quasi-Standard; verfügbar auf diversen Plattformen
- Typischer Aufruf einer solchen Send-Operation:

```
status = send(buffer, size, dest, ...)
```



Sendeoperationen in der Praxis (2)

- Derartige Systeme bieten im Allgemeinen mehrere verschiedene Typen von Send-Operation an
  - Zweck: Hohe Effizienz durch möglichst spezifische Operationen
  - Achtung: Spezifische Operation kann in anderen Situationen u.U. eine falsche oder unbeabsichtigte Wirkung haben (z.B. wenn vorausgesetzt wird, dass der Empfänger schon im receive wartet)
  - Vorsicht: Semantik und Kontext der Anwendbarkeit ist oft nur informell beschrieben