Betriebssysteme II

Sommersemester 1994

Inhaltsverzeichnis

1 Verteilte Systeme **1**
1.1 Kommunikation in verteilten Systemen . . . 1
1.2 Das OSI Schichtenmodell . . . 2
1.3 Kurzbeschreibung der Schichten . . . 2
1.3.1 Physical Layer . . . 2
1.3.2 Data Link . . . 3
1.3.3 Network . . . 3
1.3.4 Transport . . . 3
1.3.5 Session (Dialogsteuerung) . . . 3
1.3.6 Presentation Layer . . . 3
1.3.7 Application Layer . . . 4
1.4 File Transfer . . . 4
1.5 Remote Login . . . 4
1.6 Remote Job Entry . . . 4
1.7 Verteiltes Rechnen . . . 4

2 Das Ethernet **5**
2.1 Aufbau eines Ethernet Pakets . . . 6

3 Die TCP/IP Protokolle **7**
3.1 TCP/IP Dienste . . . 7
3.2 Internet Adressen . . . 7
3.3 Ports . . . 7
3.4 IP (Internet Protokoll) . . . 8
3.5 ICMP (Internet Control Message Protocol) . . . 8
3.6 UDP (User Datagram Protocol) . . . 9
3.7 TCP (Transmission Control Protocol) . . . 9
3.8 Verwaltungsdateien . . . 9
3.9 Internetworking . . . 10
3.10 *UNIX*-Sockets . . . 10
3.11 Socketpairs . . . 10
3.12 Prozeßverbindungen durch sockets . . . 11
3.13 Datagrammkommunikation . . . 13
3.14 Datagramme im *UNIX* Adreßraum . . . 13
3.15 Datagramme im Internetadreßraum . . . 14
3.16 Stream Socket Verbindungen . . . 16

4 Das Client/Server Modell **19**
4.1 Blockierende und nichtblockierende Aufrufe . . . 19
4.2 Gepufferte und ungepufferte Aufrufe . . . 20
4.3 Zuverlässige und unzuverlässige Aufrufe . . . 21
4.4 Implementierung des Client/Server Modells . . . 21
4.5 Remote Procedure Call (RPC) . . . 21
4.6 Dynamisches Binden . . . 22
4.7 Fehlerbehandlung in RPC Systemen . . . 23

5 Synchronisation in verteilten Systemen **25**
5.1 Uhrsynchronisation . . . 25
5.1.1 Algorithmen . . . 25
5.1.2 Berkeley *UNIX* Verfahren . . . 25
5.1.3 Logische Zeit (logical clock) . . . 26
5.1.4 Lamport's Algorithmus . . . 26

5.1.5 Wechselseitiger Ausschluß . . . 26
5.1.6 Verteilter wechselseitiger Ausschluß . . . 27
5.1.7 Ein Token Ring Algorithmus . . . 27
5.1.8 Vektoruhren (vector clocks) . . . 27
5.1.9 Implementierung . . . 28
5.2 Kausale Nachrichtenordnung . . . 29
5.3 Das Birman-Shiper-Stephenson Protokoll . . . 29

6 Globaler Systemzustand **30**
6.1 Der Chandy-Lamport Algorithmus . . . 31
6.2 Schnitte einer verteilten Rechnung . . . 32
6.3 Verteilte Terminierung . . . 33
6.3.1 Systemmodell . . . 33
6.3.2 1. Algorithmus . . . 33
6.3.3 Huang's Algorithmus . . . 33
6.3.4 Wahlalgorithmen . . . 33
6.3.5 Der Rüpelalgorithmus . . . 34
6.3.6 Ein Ringalgorithmus . . . 34

7 Atomare Transaktionen **35**
7.1 Transaktionsbefehle . . . 35
7.2 Transaktionseigenschaften . . . 35
7.3 Implementierung . . . 36
7.3.1 Privater Arbeitsraum . . . 36
7.3.2 Vorausgeführtes Logbuch . . . 36
7.3.3 Zwei Phasen Festschreibung . . . 36

8 Verteilte Verklemmungen **37**
8.1 Erkennung verteilter Verklemmungen . . . 37
8.1.1 Zentrale Erkennung . . . 37
8.1.2 Verteilte Erkennung . . . 37
8.2 Verklemmungsvermeidung . . . 38
8.3 Prozessor-Allokation . . . 38

Abbildungsverzeichnis

1 Nachricht, die durch alle sieben Schichten weitergereicht wurde . . . 2
2 Transceiver . . . 5
3 Verbundenes Paar aus sockets . . . 11
4 Blockierendes Senden . . . 20
5 Nichtblockierendes Senden . . . 20
6 Parameter marshalling . . . 22
7 Raum-Zeit Diagramm . . . 28
8 Banktransfer . . . 30

1 Verteilte Systeme

Ein **verteiltes System** (VS) besteht aus mehreren Prozessoren mit getrenntem Hauptspeicher (HSP). Der Übergang zu parallelen Systemen (PS) ist z.T. fließend.

Hauptcharakteristika sind:

- Ein System ist ein VS, wenn es in mehrere Komponenten getrennt werden kann, die jeweils wieder ein vollständiges VS (oder Einzelsystem (ES) oder PS sind).
- Ein ES oder PS hat einen separaten, homogenen und kohärenten Adreßraum. (Ein Pointer ist im Adreßraum eindeutig; ein *write*() wirkt sich ohne Verzögerung auf dem ganzen System aus.) Ein VS hat mehrere getrennte Adreßräume. Datenobjekte können nicht per Referenz übergeben werden, sondern müssen kopiert werden. Ein *write*() wird i.a. im System erst mit Verzögerung wirksam.

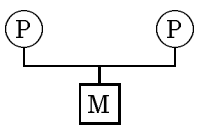
Verteilte Systeme sind (heute noch) typischerweise gekennzeichnet durch eine langsame Verbindung im Vergleich zur Prozessorleistung. Auf jeder Komponente läuft ein separates BS. Prozesse können zunächst nicht zwischen den Komponenten verschoben werden. Falls ein System zum Lastausgleich installiert ist, ist dies ein Zusatz, der nachträglich über die Einzelbetriebssysteme gelegt wurde.

Hauptvorteile von verteilten Systemen:

- **Leistung**: Nur ein Netz kann leistungsfähiger sein, als das leistungsfähigste ES oder PS. Entsprechendes gilt für HSP, Platten, I/O, ...
- **Preis**: Ein Netz aus Standardsystemen ist billiger, als ein entsprechend leistungsfähiges ES.
- **Flexibilität**: Ein VS kann langsam auf- und abgebaut werden.
- **Natürliche Parallelität**: Oft sind dezentrale, unabhängige Arbeiten zu erledigen, z.B. editieren, Übungsbetrieb etc., die in natürlicher Weise auf ein VS passen. Zusätzlich können große Aufgaben auf das VS verteilt werden, falls es passende Algorithmen gibt. (Typischerweise steht 90% der Kapazität eines Workstationnetzes leer.)

1.1 Kommunikation in verteilten Systemen

Kommunikation findet in traditionellen Systemen über einen gemeinsamen HSP statt. Z.B.:



In verteilten Systemen findet die Kommunikation über Nachrichtenaustausch mittels Netzen statt. Knoten des VS sind über Leitungen miteinander verbunden, über die elektrische Signale geschickt werden können. Typischerweise sind Verzögerungen (latency) relativ groß und die Bandbreite (bandwidth) realtiv klein (im Vergleich zur Prozessorleistung).

Kommunikation findet aber nur statt, wenn bei den Partnern Übereinkunft besteht, was die Signale, Bits, Bytes etc. bedeuten sollen. Zusätzlich müssen Formate für Anfragen und Antworten festgelegt werden. Diese Übereinkünfte heißen **Protokolle** (protocols).

Definition: Ein *Protokoll* ist ein Satz von Regeln, Datenformaten und Konventionen, der den Datentransfer zwischen den Kommunikationsteilnehmern regelt.

Protokolle werden von der ISO standardisiert, damit **offene Systeme** (open systems) gebaut werden können. Verschiedene Einzelsysteme verschiedener Hersteller können dann vernetzt werden. Die ISO hat ein Referenzmodell entwickelt, dessen Inhalt und Aufbau modellhaft ist:

Open System Interconnct(ion) Reference Model (ISO - OSI)

Es legt eine Hierarchie von Schichten (layers) fest, die sich um alle Aspekte der Kommunikation kümmern. Selbst wenn ein bestimmtes Protokoll im einzelnen von OSI abweicht (z.B. DECNET), so ist es einfacher zu verstehen und leichter mit anderen zu verknüpfen, wenn es im Aufbau dem OSI Schichtenmodell folgt.

1.2 Das OSI Schichtenmodell

Nr	Schicht	Name	Stichwort
7.	Application	Anwendungsdienste	file transfer, e-mail, R-Dienste ...
6.	Presentation	Darstellung	Heterogener Datenaustausch
5.	Session	Sitzung	Prozeßkommunikation
4.	Transport	Transport	Prozeß-Prozeß Verbindung
3.	Network	Vermittlung	Paket übers Netz
2.	Data link	Verbindung	zuverlässiger Transport auf Draht
1.	Physical	physikal. Übertragung	Hardware Bit Transport

Jede Schicht präsentiert nach oben eine Schnittstelle, die die festgelegten Dienste anbietet. Die tieferen Dienste werden benutzt, um die höheren zu implementieren. Wird eine Nachricht von einer Schicht verschickt, so wird ein Kopf (header) vorangestellt, z.B. eine Adresse. Dann wird das entsprechende Paket an die tiefere Schicht weitergereicht, die ihrerseits ihren Kopf voranstellt oder anfügt (z.B. Rechneradresse oder Prüfsumme).

Abbildung 1: Nachricht, die durch alle sieben Schichten weitergereicht wurde

1.3 Kurzbeschreibung der Schichten

1.3.1 Physical Layer

Elektrotechnische und mechanische Spezifikation der Kabel, z.B. RS-232-C. Übertragung einzelner Bits.

1.3.2 Data Link

Zuverlässige Übertragung einer Bitsequenz. Gruppiert den Bitstrom in **frames** und fügt Prüfsummen an. Wiederholt die Übertragung beschädigter frames und setzt die frames in richtiger Reihenfolge korrekt zusammen.

1.3.3 Network

Adressiert eine Nachricht und schleust adressierte Nachrichten über Zwischenstationen zum Empfänger (-rechner) (**routing**). Adressierbare Nachrichten heißen Pakete; sie können unabhängig voneinander verschickt werden (connectionless) oder es werden Verbindungen aufgebaut (connections). Optionale Pfadfindung (routing) wird auch hier behandelt. Beispiele sind X.25 (connection) und IP (Internet Protocol) (connectionless).

1.3.4 Transport

Die Schichten 1–3 verbinden Rechner. Schicht 4 stellt eine Punkt-zu-Punkt Verbindung zwischen Prozessen dar. Auch Transportprotokolle können connectionless sein, dann wird die Verbindung nur für eine Einzelnachricht hergestellt (Datagramm - UDP: User Datagram Protocol). Meist bauen Prozesse über den transport layer aber eine Verbindung auf, die die Teilnachrichten sicher und in der Reihenfolge des Senders transportiert (ohne daß jeder Teil adressiert werden muß). Die ISO Transportprotokolle sind **TP0** bis **TP4**. Das DoD [1] Transportprotokoll heißt **TCP** (Transmission Control Protocol) und ist ähnlich wie das TP4. Das UDP vom DoD greift im wesentlichen auf **IP** auf Schicht 3 zurück. Die Verbindungsprotokolle können sowohl auf *conectionful protocols* als auch auf *connectionless protocols* aus Schicht 3 aufgebaut werden (X.25, IP). Im letzten Fall stellt Schicht 4 auf Schicht 3 einen **virtual circuit** her, und ordnet die Pakete entsprechend. (Eine neuerliche Ordnung ist trotz der frame-Ordnung auf Schicht 2 nötig, da Teile der Nachricht über verschiedene Wege geleitet werden können.
Unter *UNIX* erlaubt das Konzept der **sockets** einen Zugriff auf Verbindungsprotokolle wie UDP und TCP (s.u.)

1.3.5 Session (Dialogsteuerung)

Die Sitzungsschicht ist eine Erweiterung der Transportschicht. Benutzer auf verschiedenen Rechnern eröffnen eine gemeinsame Sitzung. Die Dialogsteuerung stellt z.B. sicher, daß nur jeweils **eine** Partei spricht. Verwandt damit ist das **token management** (Zugangskontrolle) . Der Besitz eines tokens stellt eine Zugangsberechtigung dar, also eine virtuelle Art von **mutex**.
Ein weiterer Service ist die Synchronisation von Datentransfers. Hier können Wiederaufsetzpunkte (checkpoints) eingebaut werden, so daß bei einem Verbindungszusammenbruch nur ab diesem Punkt übertragen werden muß. Schicht 5 ist in vielen Realisierungen nicht vorhanden, so etwa weder in ARPANET noch im USENET. Man kann den wichtigen Dienst des **remote procedure call** (RPC) logisch in dieser Schicht ansiedeln.

1.3.6 Presentation Layer

Die Darstellungsschicht sorgt dafür, daß Daten in verschiedenen rechnerspezifischen Darstellungen ausgetauscht werden können. Die Abstraktionsebene der Kommunikation wird von Bits und Bytes auf höhere Datentypen wie *characters, integers,*

[1] Department of Defence

floats und *records* angehoben. Diese Daten können in verschiedenen Formaten dargestellt sein (EBCDIC, ASCII, little-endian (VAX 11, Intel: „little Byte # at the end"), big-endian (IBM 370, Motorola)), zwischen denen konvertiert werden muß. Weitere Themen sind Datenkompression und Verschlüsselung (RSA, DES).

1.3.7 Application Layer

Die Schicht 7 enthält Protokolle z.B. für e-mail (X.400), für Katalogzugriff (directory look up) (X.500), für file-transfer u.ä. Die Anwendungsdienste können eigenständig im Dialog betrieben werden (e-mail) oder auch in Programme eingebettet werden (verteiltes Dateisystem NFS)

1.4 File Transfer

Übertragung von Dateien zwischen unabhängigen Rechnern, z.B. zwischen den USA und Deutschland. Der OSI Dienst heißt FTAM (File Transfer Access Management). Innerhalb der TCP/IP Protokolle unter *UNIX* gibt es **FTP**.

1.5 Remote Login

Auf dem lokalen Rechner wird ein **virtuelles Terminal** simuliert (emuliert, nachgebildet), das an den fernen Rechner angeschlossen ist. OSI stellt VT (virtuel terminal) zur Verfügung, unter TCP/IP gibt es **TELNET** und **RLOGIN**

1.6 Remote Job Entry

Ziel ist es, eine run queue im verteilten System zu organisieren, so daß batch jobs auf einen fernen Rechner ausgelagert werden können. OSI: Job Transfer & Management (JTM).

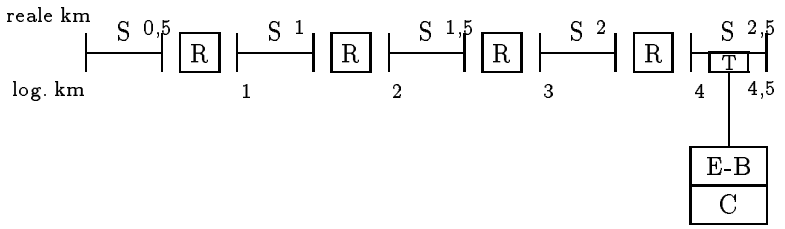
1.7 Verteiltes Rechnen

Auf einem Netz sollen parallele Programme verteilt werden. Unterstützt wird process creation, Verschicken von Nachrichten, graphische Anzeige der Kommunikation, Umlenken der I/O auf zentralen Rechner etc. PVM (parallel virtual machine)

Nicht OSI, aber ähnlich: SNA von IBM und NetBIOS.
UUCP: *UNIX-UNIX*-COPY. Mit dem Program UUCP kann ein *UNIX* Rechner von und zu einem anderen, mittels Wahlleitung und Modems, Files übertragen. Die Aufträge werden zunächst in `/usr/spool/uucp` abgelegt. UUCP wird (z.B. nachts) vom **cron daemon** aktiviert, baut eine Verbindung auf und sucht nach Kopieraufträgen, die er dann abwickelt.
Auf dieser Basis gibt es auch *UNIX* für remote execution und den verwandten USENET Bulletin Board Service. UUCP ist ein verteilt, über die Systemhacker der Unirechner verwaltetes System.

2 Das Ethernet

Das Ethernet entspricht den OSI Schichten 1 und 2. Ethernet wurde von XEROX PARC entwickelt und von XEROX, DEC und Intel (single chip controller) standardisiert.
Ethernet basiert auf einem durch KoaxKabel realisierten Bus mit 10 MBit/s Übertragungsrate, auf den alle Stationen zugreifen. Kommunikationsbasis ist eine Schnittstelle nach ISO 8802/3 bzw. IEEE 802.3. Die Übertragung erfolgt in Paketen zu mindestens 512 Bit. Jedes Paket beginnt mit einem Synchronisationsmuster von 64 Bits (1010...1011).
Das Zugriffsverfahren ist CSMA/CD (Carrier Sense Multiple Access / Collision Detection). Eine sendewillige Station hört zunächst ab, ob der Bus (die Leitung) frei ist (carrier sense). Ist dies der Fall, wird gesendet. Wegen der Kabellänge dauert es eine Zeit, bis das Signal den Bus geflutet hat (1 Fuß in 1 ns). So kann es vorkommen, daß eine weitere Station sendet, bevor das Signal bei ihr angekommen ist. Bei CSMA wird fertig gesendet und ein ACK abgewartet. Bei Ausbleiben (Störung) wird die Sendung wiederholt. Bei der Verbesserung /CD wird die Sendung aller beteiligten durch ein **jam** Signal abgebrochen. Danach wartet jeder Sender (selbst bei leerem Bus) eine **zufällig** gewählte Zeitspanne (back-off), bevor er erneut sendet.
Aus der minimalen Paketlänge, der Signalausbreitung und den Reaktionszeiten bei Kollision errechnet sich eine theoretische maximale Kabellänge von 5,2 km. (Ein Sender muß **während** der Sendung noch von einer Kollision erfahren, da ohne ACK gearbeitet wird. Wegen Eintreten der Dämpfung ist im Kanal nach 500m ein Verstärker erforderlich. Wir sprechen von **Segment** und **Repeater**. Der Repeater verzögert wie ein Segment. Zunächst ergibt sich eine maximale Ausdehnung von etwa



Mit Sicherheitsmarge oft weniger.
Rechner werden an ein Segment mittels eines Ethernetboards im Rechner, eines **Drop**-Kabels (max. 50m) und eines **Transceivers** angeschlossen. Der Transceiver erzeugt bei Kollisionsentdeckung durch kurzfristiges Weitersenden das **jam** Signal. Transceiver müssen 2,5m Abstand auf dem Kabel haben, da der Anschlußdorn das Kabelsignal stört.

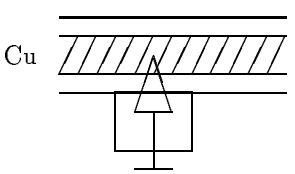


Abbildung 2: Transceiver

Verschiedene Ethernets können verbunden werden (z.B. CISCO Bridge). Sie leiten diejenigen Pakete weiter, deren Adresse auf einem anderen Netzteil liegt.
Original Ethernet beruht auf 0,4" KoaxKabel, das **Thick Ethernet Yellow Cable** oder **10Base5** (10 MBit/s, Basisband, 500m) genannt wird. Billigere Varianten sind **Thin-Wire** 10Base2 mit 0,25" und Systemen bis 185m sowie **10BaseT** oder **Twisted Pair** bis 30m. Jedes Ethernetboard trägt eine 48Bit Adresse, wobei ursprünglich die ersten 3 Byte für den Hersteller reserviert waren (von IEEE), und die anderen 3 Byte für die Gerätenummer.

2.1 Aufbau eines Ethernet Pakets

Präambel	8 Bytes	Synchronisation
Empfänger	6 Bytes	Adresse (11...1 = Broadcast)
Sender	6 Bytes	Adresse
Protokoll	2 Bytes	realisiertes Protokoll, z.B. IP
Daten	$\leq$ 1,5kB	Nach Abzug der header höherer OSI-Schichten bleiben ca. 1kB Nutzdaten
Prüfsumme	4 Bytes	CRC

Die Paketlänge ist ein Kompromiß zwischen Overhead der Verpackung und Kollisionsgefahr. Das Paketende wird durch Codierung automatisch erkannt.

Ethernet wird von vielen höheren Protokollen benutzt (XNS von XEROX, Novell (PC Netze mit PC als Netzwerk-Server unter NetWare) und TCP/IP des Internet).

3 Die TCP/IP Protokolle

Diese Protokolle sind Teil des Internet, das auf ARPA-Net zurückgeht. Wichtigste Protokolle sind TCP/IP (Transmission Control Protocol / Internet Protocol). Seit '82 wurde TCP/IP in Berkeley *UNIX* integriert (andere unter IBM/MVS, DEC/VMS, MS-DOS). TCP/IP beginnen auf OSI Schicht 3. Internetkommunikationspartner heißen **hosts**. Jeder hat eine Internetadresse und sie werden vom Netz verbunden. Internetprotokolle werden vom *Internet Activities Board* am SRI gepflegt. Standardisierungsvorschläge werden als request-for-comments publiziert (z.B. RFC 791 für IP).

3.1 TCP/IP Dienste

Frühe Dienste umfassen **remote login** (TELNET), **file transfer** (FTP) und **electronic mail** (SMTP). Später kamen Dienste wie NFS und X-Windows hinzu. TCP/IP ist zehn Jahre älter als OSI und benutzt vier Schichten:

<table>
<tr><td>OSI</td><td>TCP/IP
Schichtenaufbau</td><td colspan="3">TCP/IP
Dienste u. Protokolle</td></tr>
<tr><td>7 Anwendung
6 Darstellung
5 Komm./Sitzung</td><td>Anwendung
(application level)</td><td>TELNET
FTP, SMTP</td><td colspan="2">NFS
XDR
RPC</td></tr>
<tr><td>4 Transport</td><td>Transport
(transmission level)</td><td>TCP, UDP</td><td colspan="2">UDP</td></tr>
<tr><td>3 Vermittlung</td><td>Vermittlung
(internet level)</td><td>IP (ARP,
RARP)</td><td colspan="2">{ICMP}</td></tr>
<tr><td>2 Sicherung
1 Bitübertragung</td><td>Übertragung
(network level)</td><td>(CSMA/CD)
Ethernet(Bus)</td><td>Token
Bus</td><td>Token
Ring</td></tr>
</table>

3.2 Internet Adressen

Hostadressen sind eindeutig und unabhängig vom Übetragungsnetz. Adressen sind 4 Byte Integers.

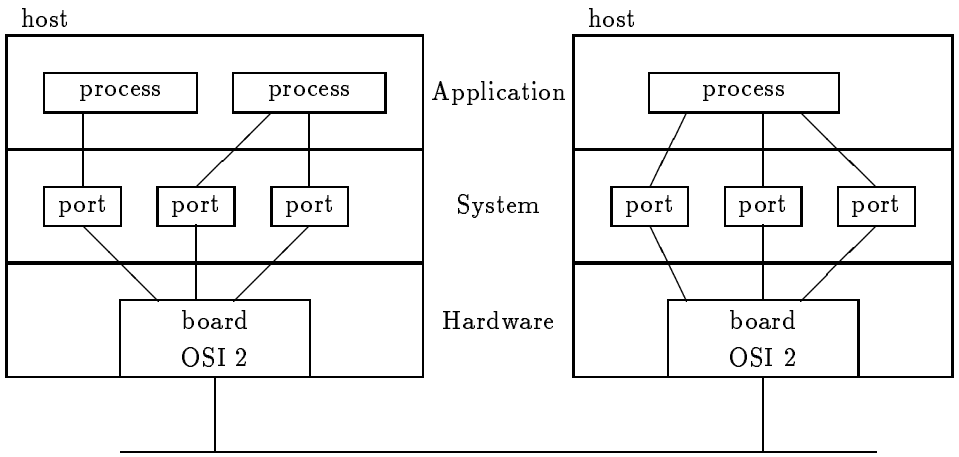
Bsp.: utu = 134.2.16.7 (siehe `/etc/hosts`), morlaix = 134.2.12.6

Internetadressen sind Teil der Hosts**oftware**. Sie werden auf HW Adressen (z.B. eines Ethernet boards) abgebildet. Bei Ethernet definiert dazu ARP (Adress Resolution Protocol) eine Zuordnungstabelle. ARP kann ein Ethernet Broadcast Paket senden mit einer Internetadresse und der entsprechende host antwortet mit seiner Ethernetadresse.
Ein diskless client (z.B. SunELC) kennt seine Internetadresse beim Booten nicht. Er sendet einen RARP (Reverse ARP) broadcast mit seiner Ethernetadresse (RARP in Ethernet Paket) und erhält vom Adressenserver seine Internetadresse zurück.

3.3 Ports

Auf der OSI Transportschicht werden Prozesse verbunden; die Hostadressen müssen ergänzt werden. TCP/IP stellt **ports** zur Verfügung (OSI kennt den TSAP - Transaction Service Access Point). Wohlbekannte Standardprozessen, wie etwa FTP, sind auf allen hosts die gleichen **well known ports** zugeordnet (FTP=21). Portnummern 1–1023 sind für den Systemverwalter reserviert, 1024–5000 werden auf Anfrage von der Portverwaltung vergeben, 5001 und höher können ohne Kontrolle vom Anwender zugeordnet werden.
Sowohl TCP als auch UDP Pakete speichern die Portnummern in einem Feld. Somit ist ein Verbindungspunkt adressiert als Paar `<Internetadresse,Port#>`.

Eine Verbindung ist charakterisiert durch:
`<Protokoll, lokale IA, lokale Port#, ferne IA, ferne Port#>`.
Unter *UNIX* wird ein Verbindungsende durch eine **socket** (Steckdose) dargestellt: `<Protokoll, IA, Port#>`. Auf eine lokale socket werden Daten wie auf eine Datei geschrieben unter Angabe der fernen socket. Dort können sie wie von einer Datei gelesen werden.



3.4 IP (Internet Protokoll)

RFC 791, zentrales Protokoll von TCP/IP.

Ein unsicheres, verbindungsloses Protokoll auf OSI Schicht 3. 20 Byte header, maximal 65535 Byte Paketlänge ($\hat{=}$ 2 Byte Längenangabe). Je nach Schicht 2 werden IP Pakete zum Transport weiter fragmentiert und wieder zusammengesetzt. IP vergibt dazu Fragmentnummern. IP zerstört Pakete, die sich zu lange im Netz aufhalten, bevor sie die Puffer zum Überlauf bringen. (Eine Lebensdauer wird herabgezählt.)

Der Protokollkopf enthält:

- Die Länge des headers
- Die Länge des Pakets
- Laufende Nummer für Fragmentierung
- Angabe des höheren Protokolls (TCP, UDP, ICMP)
- Kopfprüfsumme
- Internetadresse des Senders
- Internetadresse des Empfängers
- Lebensdauer

3.5 ICMP (Internet Control Message Protocol)

ICMP hilft, auf Basis des unsicheren IP, einen sicheren höheren Dienst schaffen zu können. ICMP benutzt IP Pakete und verschickt im Datenteil Kontrollnachrichten, etwa „Reduziere Datenrate“, „Neusenden eines Pakets“, „Ich lebe noch“.

Bsp.:
`ping utu 20 3`
sendet 3 ICMP Pakete zu je 20 Bytes an utu. Falls TCP/IP auf utu lebt, sendet er sie zurück, und ping kann eine Statistik über die Qualität der Verbindung angeben.

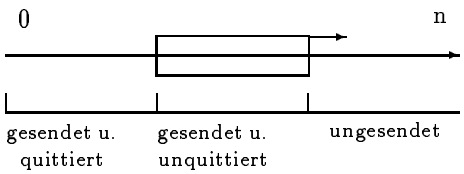
3.6 UDP (User Datagram Protocol)

UDP ist das Datagramm Protokoll der Transportschicht; unsicher aber sehr effizient. Für höhere Dienste empfehlenswert, falls man ein relativ sicheres physikalisches Netz hat. UDP wird im wesentlichen direkt auf IP abgebildet, allerdings adressiert UDP einen Prozeß-Port und IP einen host. Der UDP header enthält acht Bytes:

- Senderport (2 Byte)
- Empfängerport (2 Byte)
- Paketlänge (2 Byte)
- Kopfprüfsumme (2 Byte)

3.7 TCP (Transmission Control Protocol)

TCP realisiert eine sichere, bidirektionale Punkt-zu-Punkt Verbindung zwischen zwei ports. Der Benutzer sieht logisch einen Bytestrom; TCP zerlegt den Strom in IP-Pakete und verwaltet im TCP-header eine Sequenz- und eine Quittierunsnummer. Quittiert werden Bytes, d.h. eine Quittung sagt an, bis zum wievielten Byte korrekt empfangen wurde. Der Sender sendet fortlaufend („auf Vorrat"); die maximale Anzahl noch unquitierter Bytes ist durch die Größe eines **sliding windows** begrenzt.



Der Protokollkopf von 20 Bytes enthält u.a.:

- Senderport (2 Byte)
- Empfängerport (2 Byte)
- Sequenznummer (4 Byte)
- Quittungsnummer (4 Byte)
- Fenstergröße (2 Byte)
- Kopfprüfsumme (2 Byte)

3.8 Verwaltungsdateien

Textdateien zur Verwaltung von TCP/IP-Netzen unter *UNIX*.

	Name	**Kodierung**	**Alias**	**Kommentar**
`/etc/protocols:`	udp	17	UDP	#user datagr. prot.
`/etc/networks:`	loopback-net	127		software-loop

Name der installierten Internetsubnetze und ihrer Nummern. loopback-net ist ein Pseudonetz zu Testzwecken. Es gibt Pakete sofort zurück.

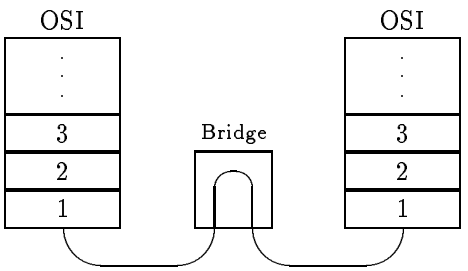
/etc/hosts: Zuordnung host-Namen und Internetadresse mit aliasen

/etc/services: Zuordnung der wohlbekannten Dienste zu ihren ports mit den verwendeten Protokollen.

3.9 Internetworking

Verbindung zwischen verschiedenen Netzen oder Netzteilen. Man unterscheidet nach der Höhe der Protokollschicht, auf der die Umsetzung erfolgen muß:

Schicht 1:	**Repeater**	Schleust Bits vom einen Teil zum anderen
Schicht 2:	**Bridge**	Schleust frames in die Sicherungsschicht, z.B. zw. Ethernet und Token Ring
Schicht 3:	**Router**	Schleust z.B. zw. verbindungslosen und verbindungsorientierten Netzen. Mit Pfadplanung
Höher:	**Gateway**	



3.10 *UNIX*-Sockets

Sockets sind eine TCP/IP Realisierung der OSI TSAP's. Sie realisieren eine Kommunikationsschnittstelle zwischen Anwendungsprogrammen und den TCP/IP Protokollen. Sie stellen jeweils das Ende einer Prozeß-Prozeß-Verbindung dar.
Grundidee ist es, in *UNIX* eine Nahverbindung bzgl. Lesen und Schreiben wie eine Datei zu behandeln. Statt *file descriptors* haben wir jetzt **socket descriptors** .
Aus Gründen der Uniformität gibt es socket-Verbindungen über das Internet und auch zwischen *UNIX*-Prozessen auf demselben Rechner. Man spricht von **Internet domain**, bzw. *UNIX* **domain**. Eine **domain** ist ein Adreßraum, wo eine bestimmte Art von Adressen gültig ist. (Innerhalb der *UNIX* domain sind socket-Verbindungen über das Dateisystem implementiert.)
In 4BSD werden auch pipes über sockets implementiert. Pipekommunikation ist aber unidirektional, Socketkommunikation ist bidirektional.

```
int endpoint[2];
pipe(endpoint);
```

erzeugt eine pipe mit zwei Deskriptoren, so daß von `endpoint[0]` gelesen und auf `endpoint[1]` geschrieben werden kann. Pipes sind stream-Verbindungen (stream = Fluß), die einen ununterbrochenen Fluß von Bytes realisieren. Als Seiteneffekt werden die Deskriptoren in den ersten freien Plätzen der Deskriptorentabelle eingetragen. Mit

```
close(stdin);
close(stdout);
pipe(endpoint);
```

hat man eine pipe von *stdin* nach *stdout*. Danach kann z.B. ein *fork()* stehen. Nun programmiert man Vater und Kind so, daß einer schreibt (auf *stdout*) und der andere liest (von *stdin*). (Die nicht benötigten Deskriptoren kann man schließen.)

3.11 Socketpairs

Ein verbundenes Paar aus sockets stellt eine Verallgemeinerung einer pipe dar, da der Datenfluß bidirektional ist.

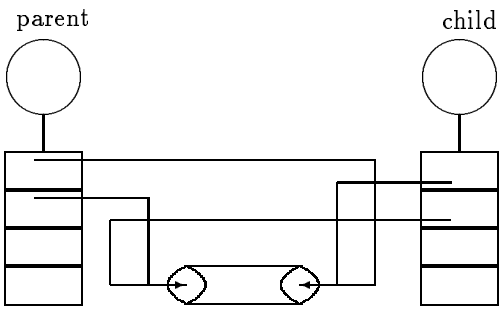


Abbildung 3: Verbundenes Paar aus sockets

```
#include <sys/types.h>
#include <sys/sockets.h>

int sockets[2], errorcode;
errorcode = socketpair(domain, type, protocol, sockets);
ASSERT(errorcode == 0)
```

erzeugt ein Paar socket-Deskriptoren im Array `socket`. Z.Zt. ist der Aufruf nur implemtiert für:

```
domain = AF_UNIX
type = SOCK_STREAM
protocol = 0 /* ein passendes TCP wird gewaehlt */
```

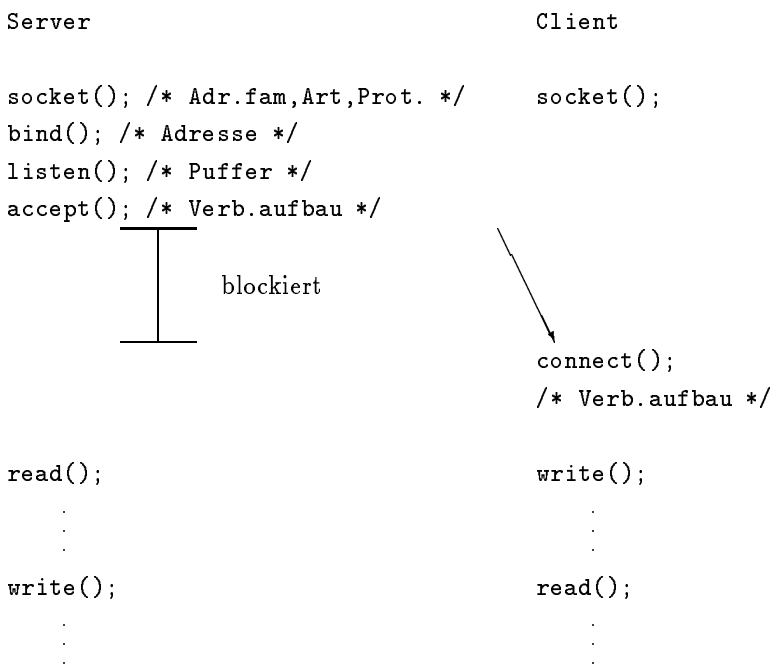
Anschließend kann mit dem üblichen *read()* und *write()* Systemaufrufen auf die sockets zugegriffen werden, z.B.:

```
char buf[1024];
#define DATA "Once upon a time ..."
read(sockets[1], buf[1024]);
write(sockets[1], DATA, sizeof(DATA));
```

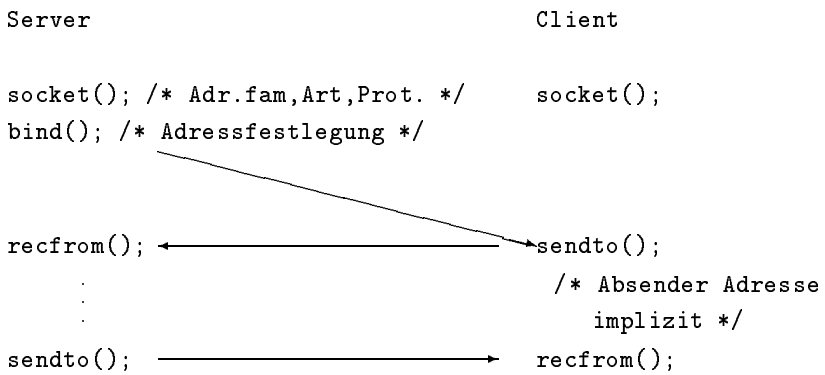
3.12 Prozeßverbindungen durch sockets

Die fünf Komponenten einer Prozeßverbindung (protocol, local host, local port, remote host, remote port) werden durch verschiedene Systemaufrufe der socket-Familie gesetzt. Grundsätzlich: Bei einem verbindungsorientierten Protokoll (TCP) sind die Adressen Teil der Verbindung. Durch ein *connect()* des Klienten und ein *accept()* des Servers findet ein rendez-vous statt, bei dem die Verbindung aufgebaut wird. Nachfolgendes Lesen und Schreiben geschieht unter Angabe der Verbindung, aber ohne weitere Angabe der Adresse.
Bei einem datagrammorientierten Protokoll (UDP) werden zunächst die Endpunkte eines Kommunikationsweges aufgebaut (adressierbare Briefkästen).
Jede Nachrichtenübermittlung geschieht als unabhängiges Paket unter Angabe der Zieladresse über einen lokalen Endpunkt.

Struktur der TCP Aufrufsequenz



Struktur einer typischen UDP Aufrufsequenz



Mit *socket()* wird zunächst nur die Art der Kommunikation und das Protokoll festgelegt:

```
int sd; /* socket descriptor */
int domain, type, protocol;
sd = socket(domain, type, protocol);
```

Resultat ist ein socket-Deskriptor, der wie die file-Deskriptoren einen Index in die Deskriptortabelle des Prozesses darstellt. Für jeden Adreßraum kann es verschiedene Kommunikationsarten geben und eine bestimmte **(domain, type)** Kombination kann durch verschiedene Protokolle realisiert sein, die sich in tieferen Schichten unterscheiden. In heutigen *UNIX* Systemen ist normalerweise nur ein Protokoll pro (domain, type) Paar realisiert.

Wir benutzen die Standardimplementierung durch Angabe von
`protocol = PF_UNSPEC /* protocol family */`
im socket-Aufruf (wobei `#define PF_UNSPEC 0` in `sys/socket.h`).
In der *UNIX* Welt sind besonders wichtig:

```
domain = AF_UNIX /* UNIX domain */
       = AF_INET /* Internet domain */

type   = SOCK_DGRAM  /* datagram connection */
       = SOCK_STREAM /* stream connection */
```

Jeder socket muß eine Adresse zugeordnet werden, bevor kommuniziert werden kann. Der allgemeine Adreßrahmen ist

```
struct sockaddr
{
 u_short sa_family   /* address family AF_xxx */
 char    sa_data[14] /* up to 14 bytes of direct address */
}
```

Je nach Adreßrahmen werden die 14 Byte unterschiedlich genutzt und interpretiert. Die Adreßfamilien haben eigene Adreßdefinitionen, die in diesen Rahmen meist passen (z.B. `sockaddr_un, sockaddr_in`).

3.13 Datagrammkommunikation

Der *bind()* Befehl ordnet einer socket eine Adresse zu. Danach kann an die socket unter der Adresse eine Nachricht geschickt werden. Die Adresse muß für jede Nachricht separat angegeben werden. Deshalb kann über eine (einzige) lokale socket eine beliebige Anzahl Kommunikationspartner angesprochen werden. (Es wird also nicht wie beim Verbindungsaufbau pro Partner ein Deskriptor gebraucht).
Der Adreßraum *UNIX* erlaubt Kommunikation nur innerhalb eines geschlossenen Systems, beim Internet Adreßraum fällt diese Einschränkung weg. Datagrammverbindungen sind flüchtig (transient) und existieren nur während der Nachrichtenübermittlung.

3.14 Datagramme im *UNIX* Adreßraum

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

Adressen in der *UNIX* domain sind Pfadnamen. Die Adreßstruktur ist:

```
struct sockaddr_un
{
 short sun_family;
 char  sun_path[108];
}
```

Die Varibale `sun_family` ist auf `AF_UNIX` zu setzen, damit das Protokoll weiß, wie die nachfolgende Adresse zu lesen ist. Mit *bind()* wird eine Adresse an die socket gebunden, damit ihr Nachrichten gesendet werden können. Als Seiteneffekt von *bind()* wird der Pfadname in den Dateibaum eingetragen. Er dient zum Herstellen des rendez-vous, die Daten werden aber nicht über diese Datei übertragen. Die rendez-vous Pfade bleiben bestehen, bis sie mit *unlink()* entfernt werden. Pfadnamen können relativ oder absolut sein. Bei relativen Pfadnamen muß der Partner wissen, wo sie eingetragen sind.

Beispiel:
Server: „lesen von Datagrammen in `AF_UNIX`“

```
struct sockaddr_un address;
char buf[bufsize];
sock = socket(AF_UNIX, SOCK_DGRAM, PF_UNSPEC);
     .
     .
     .
/* Adresse */
address.sun_family = AF_UNIX;
strcpy(address.sun_path,"server");
bind(sock, (struct sockaddr *)&address, sizeof(struct sockaddr_un));

/* Lesen */
read(sock, buf, bufsize);
     .
     .
     .
unlink("server");
```

3.15 Datagramme im Internetadreßraum

Diese Nachrichten können zwischen verschiedenen Maschinen (hosts) im Internet verschickt werden.

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

Das Adreßformat ist:

```
struct sockaddr_in
{
  short sin_family;
  u_short sin_port;
  struct in_addr sin_addr           /* struct in_addr
  char sin_zero[8]; /* empty */        { u_long s_addr; } */
}
```

Spezifiziert wird eine 32Bit Hostadresse und ein port (TSAP). Wir haben hier `sin_family = AF_INET`. Die Adresse eines namentlich bekannten fernen hosts kann man mit *gethostbyname()*, die eigene mit `INADDR_ANY` bekommen. Den port muß man wissen. Ein port wird einer socket via *bind()* zugeordnet, oder aber implizit als Seiteneffekt eines *sendto()* (oder *connect()*). Den erhaltenen port kann man mit *getsockname()* erfahren. Die Socketadresse ist als Teil des Internetstandards definiert, der auch die Bytezuordnung festlegt. Diese kann anders sein, als die des lokalen host. Deshalb gibt es die folgenden Konversionsroutinen Network $\leftrightarrow$ Host.

```
ulong  htonl(x); ulong  x; /* host to net long  */
ulong  ntohl(x); ulong  x; /* net to host long  */
ushort htons(x); ushort x; /* host to net short */
ushort ntohs(x); ushort x; /* net to host short */
```

gethostbyname() liefert einen Zeiger auf eine gefüllte Struktur hostent (host entity).

```
#include <netdb.h>

struct hostent* gethostbyname(host);
char *host;
struct hostent
{
 char *h_name;
 char *h_aliases[]; /* also known as */
 int   h_addrtype;
 int   h_length;
 char *h_addr_list;
}

#define h_addr h_addr_list[0]
```

Ein host kann unter mehreren Adressen erreichbar sein. `h_addr` liefert die erste (und oft einzige) Adresse. Die Adresse ist ein Byte-String, in C ein array of char. Es sind allerdings wirklich Bytes, unabhängig von einem character code; es kann z.B. das Byte 0x0 vorkommen. Deshalb kann man die Adresse nicht mit *strcpy()* umladen. Dazu gibt es die Funktion `void bcopy(s1,s2,n); char *s1,*s2;`, die `n` Bytes von `s1` nach `s2` kopiert. `bzero(s,n)` füllt `s` mit `n` Null-Bytes.

Beispiel:
Verschicken eines Internetdatagramms:

```
#define DATA "there were four little rabbits ..."
struct sockaddr_in address;
struct hostent *hp, *gethostbyname();
/* locale socket */
sock=socket(AF_INET, SOCK_DGRAM, PF_UNSPEC /* PF_INET */);

/*
  Ferne Adresse: wir kennen den Namen "utu",
  des fernen host und die Portnummer port des
  Empfaengers
*/
hp=gethostbyname("utu");
bcopy(hp->h_addr,(char *)&address.sin_addr,hp->h_length);
address.sin_family=AF_INET;
address.sin_port=htons(port); /* port ist bekannt */

/* Schicke Nachricht */
sendto(sock,DATA,sizeof(DATA),0,&address,sizeof(address));
```

Beispiel:
Empfangen eines Internetdatagramms:

```
struct sockaddr_in address;
char bug[bufsize];

/* socket */
sock=socket(AF_INET,SOCK_DGRAM,PF_UNSPEC);
address.sin_family=AF_INET;
address.sin_addr.s_addr=INADDR_ANY;
address.sin_port=0; /* Das System soll einen port zuteilen */
bind(sock,(struct sockaddr *)&address,sizeof(address));
```

```
/* Lesen */
read(sock(sock,buf,bufsize));

/* Drucke lokale Portnummer */
length=sizeof(address);
getsockname(sock,&address,&length);
printf("%d\n",ntohs(address.sin_port));
```

3.16 Stream Socket Verbindungen

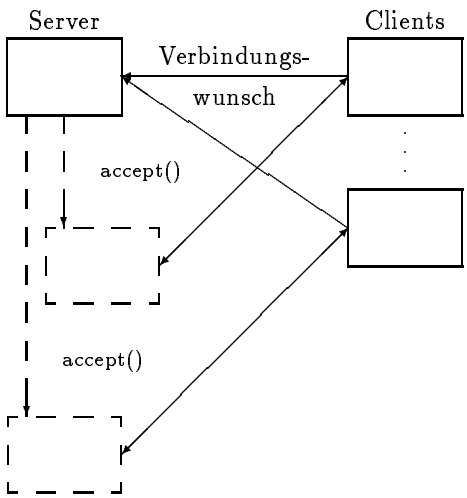
Der Verbindungsaufbau ist asymmetrisch. Ein Prozeß übernimmt die Rolle des Servers, der andere die des Clients. Der Server richtet eine socket ein (*socket(),bind()*) und bietet über diese seine Diesnte an. Mit dem connect-Befehl kann dann der Client (via seine eigene socket) eine Verbindung initiieren, um eine Anfrage zu starten.

UNIX **domain**	**Internet domain**

#include <sys/socket.h>
#include <sys/un.h>
#include <sys/types.h>
 	
int sd, nlen;
struct sockaddr_un sock;
sd=socket(AF_UNIX,
SOCK_STREAM,0);
sock.sun_family=AF_UNIX;
strcpy(sock,sun_path,socketname);
 	
nlen=sizeof(struct sockaddr_un);
	#include <sys/socket.h>
	#include <netinet/in.h>
	#include <netdb.h>
	
	int sd, nlen;
	struct sockaddr_in sock;
	sd=socket(AF_INET,
	SOCK_STREAM,0);
	sock.sin_family=AF_INET;
	sock.sin_port=htons(PortNo);
	bcopy(hostname,sock.sin_addr,
	sizeof(hostname));
	nlen=sizeof(struct sockaddr_in);

connect(sd,(struct sockaddr *)&sock,nlen);	

Bei stream sockets ist die Verbindung fest, bis sie durch schließen (*close()*) einer der beteiligten sockets aufgelöst wird, d.h. man kann nur genau einmal *connect()* aufrufen. Der Server richtet mit *listen()* eine queue ein, in der Verbindungsanfragen aufgelistet werden, bis sie zur Abarbeitung akzeptiert (*acccept()*)worden sind.



Systembedingt können zur Zeit maximal fünf Wünsche gespeichert werden. Die queue wird mit `listen(sd,MAX)` eingerichtet. Danach können Verbindungen akzeptiert werden.

UNIX	Internet
int newsock;	int newsock;
struct sockaddr_un from;	struct sockaddr_in from;
nlen=sizeof(struct sockaddr_un);	nlen=sizeof(struct sockaddr_in);
newsock=accept(sd,(struct sockaddr*)&from,&nlen);	

accept() erzeugt eine Kopie der eingerichteten socket (newsock) über die dann kommuniziert wird. Nach dem accept()-Aufruf steht in der **from**-Struktur die Absenderadresse des Klienten. Falls diese nicht interessiert übergibt man NULL an *accept()*. Ein *accept()* blockiert den Server, bis eine Verbindung zustande kommt. Dies ist besonders dann nicht erwünscht, wenn Verbindungen über mehrere sockets akzeptiert werden können. Der *select()* Befehl dient der Auswahl einer socket mit anstehendem Verbindungswunsch.

Beispiel:

```
#include <sys/time.h>
#include <sys/types.h>

fd_set rt, wt; /* read/write template   */
struct timeval wait;
int nb; /* number of ready descriptores */
for (;;)
{
 wait.tv_sec=5; /* wait 5 seconds */
 wait.tv_usec=0;
 FD_ZERO(&rt);    /* initialize read_template  */
 FD_ZERO(&wt);    /* initialize write_template */
 FD_SET(sd1,&rt); /* include sd1 with rt       */
 FD_SET(sd2,&rt); /* include sd2 with rt       */
 nb=select(FD_SETSIZE,&rt,&wt,(fd_set*)0,&wait);
 if(FD_ISSET(sd1,&rt)) { ... /* sd1 is ready */ }
 if(FD_ISSET(sd2,&rt)) { ... /* sd2 is ready */ }
}
```

Kommunikation über ready sockets kann dann mittels *read(), readv(), write()* und *writev()* erfolgen:

```
int read(sd,buf,bufsize);
int write(sd,DATA,DATA_size);
```

oder auch mittels *send()* und *recv()*:

```
int recv(sd,buf,bufsize,flag);
int send(sd,DATA,DATA_size,flag);
```

Als *flags* stehen zur Verfügung:

```
#include <sys/socket.h>

MSG_OOB  /* out of band data */
MSG_PEEK /* treat read data as unread */
```

Eilige Meldungen werden mit `send(...,MSG_OOP)` verschickt. `MSG_PEEK` löscht gelesene Daten nicht im Puffer. Wie beim Lesen von files kann man auch nichtblockierendes Lesen von sockets benutzen:

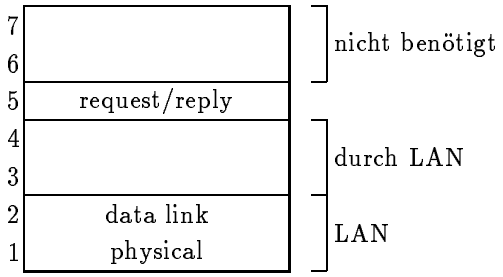
```
#include <fcntl.h>

fcntl(sd,F_SETFL,FNDELAY);
```

4 Das Client/Server Modell

Dienste der höheren OSI Schichten (5–7) und insbesondere verteilte Anwendungen sind oft nach dem **Client/Server Modell** aufgebaut. Ein Server (Dienstleistungsprogramm) stellt einen Service (Dienst, Funktionen) zur Verfügung. Ein Client fordert bei einem Server einen Dienst an. Dazu wird ein **request/reply** Protokoll benötigt, das Anforderungen Client→Server und Antworten Server→Client weiterleitet. Hierzu genügt meist ein verbindungsloser Datagrammdienst.

Selbst das (verteilte) BS kann nach diesem Prinzip strukturiert sein: eine Menge von Servern läuft auf dem selben Mikrokern (Dateiserver, I/O Server, ...) und bietet Dienste an. Ein Server kann selbst wieder Client eines anderen Servers sein

Falls dieses BS auf gleichen Maschinen eines LAN läuft, werden nur drei OSI Schichten zur Kommunikation benötigt:



Zur Implementierung des request/reply Protokolls muß das BS nur zwei Befehle zur Verfügung stellen: `send(addr,&mess)` und `receive(addr,&buff)`.
send() schickt eine Nachricht `mess` an die Adresse `addr`.
receive() empfängt eine Nachricht von der Adresse `addr` und kopiert sie in `buff`.
Beide Befehle blockieren den Aufrufer, bis sie abgeschlossen sind.

Beispiel: File-Server

Adressierung

Prozesse werden über einen TSAP angesprochen. Üblich ist eine BSinterne Abbildung TSAP↔ProzeßID, die durch einen BSaufruf erzeugt wird. Damit ist die Kommunikation unabhängig von der ID, die der Prozeß (z.B. Server) von Fall zu Fall bekommt. Ein TSAP besteht aus einem Paar <machine,port>.

Der Ansatz ist noch nicht so flexibel wie möglich, da Nachrichten an fest kodierte TSAPs geschickt werden. Ein Server kann nicht einfach auf eine andere Maschine verlagert werden, da ihn Nachrichten nicht mehr erreichen.
Ein zweites Schema besteht darin, daß eindeutige Servernamen vergeben werden, z.B. durch eine zentrale Instanz (potentielles Skalierproblem) oder als Zufallszahlen aus einem großen (z.B. 64Bit) Adreßraum. Diese Namen werden dann auf den konkreten TSAP abgebildet, über den der Server jeweils erreichbar ist. Dies kann durch ein **broadcast** geschehen, wie es z.B. auf LANs unterstützt wird. Wenn der Kern einen Sendewunsch bearbeitet, fragt er durch broadcast nach dem TSAP der zum Namen gehört. Der Kern, bei dem der Name registriert ist, antwortet mit dem zugehörigen TSAP.
Alternativ kann man an einem bekannten TSAP einen **name server** einrichten, bei dem alle Server registriert sind und der die Umsetzung vornimmt (Skalierbarkeit? Zuverlässigkeit?)

4.1 Blockierende und nichtblockierende Aufrufe

Die Standardversionen von *send()* und *receive()* sind blockierend (oder **synchron**. Der Aufrufer ist blockiert, bis die Nachricht verschickt (evtl. angekommen) bzw.

empfangen ist. Bei nichtblockierenden Aufrufen wird die Aktion lediglich initiiert, und der Aufrufer erhält danach sofort die Kontrolle zurück.

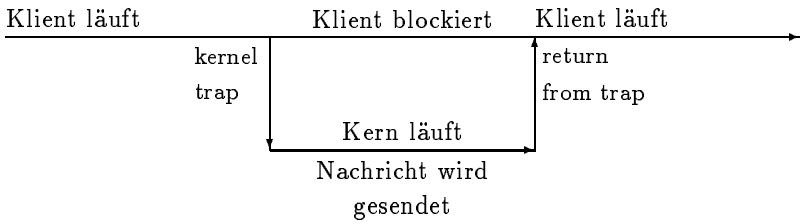


Abbildung 4: Blockierendes Senden

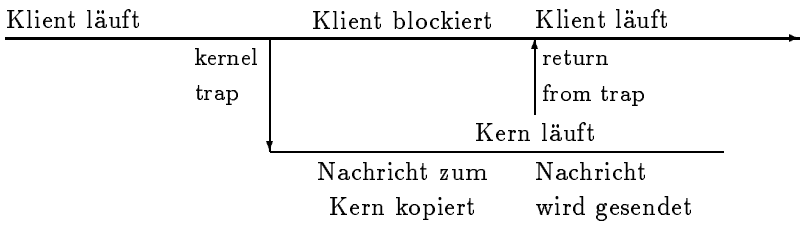


Abbildung 5: Nichtblockierendes Senden

Vorteil der blockierenden Lösung ist konventionelle Semantik: nach dem send()/-receive() Befehl ist die Befehlsausführung beendet. Außerdem braucht nicht kopiert zu werden. Bei Multithreading bekommt man die asynchronen Versionen durch forken der synchronen Befehle.
Vorteil der asynchronen Lösung ist der erhöhte Durchsatz durch Parallelität. Außerdem wird die Ausführungszeit berechenbar. Nachteil ist der Kopieraufwand und die geänderte Semantik der Befehlsausführung.
Wichtiger als asynchrones send() ist ein asynchrones receive(), da man nicht weiß, wann der Partner antwortet. Oft will man auch mehrere ports simultan bedienen. Unter *UNIX* gibt es den Befehl **select()**, um den port zu identifizieren, der kommunikationsbereit ist.
Bei Multithreading kann für jeden receive port ein thread eingesetzt werden. Der wartende receive thread hat als Kontext einen privaten Puffer und einen Zeiger auf die Empfangsroutine (message handler). Statt einen teuren BSthread mit dem Warten zu blockieren, kann das BS beim Eintreffen der Nachricht den thread spontan erzeugen (**pop-up thread**) und die Handlerroutine auf ihm starten. Der Aufruf wäre z.B. `receive(port,&buffer,handler);`.

4.2 Gepufferte und ungepufferte Aufrufe

Eine Nachrichtenübergabe stellt ein rendez-vous zwischen Sender und Empfänger dar. Dies gilt im engen Sinne dann, wenn die send()/receive() Aufrufe synchron und ungepuffert sind. Die Eigenschaft **asynchron** und **gepuffert** dienen dazu, das rendez-vous zeitlich zu entkoppeln, damit sich Sender und Empfänger nicht zu einem Zeitpunkt treffen müssen. Beim asynchronen Aufruf trifft sich nur ein Teilthread der Partner (evtl. synchron), läuft aber jeweils asynchron zum Hauptthread. Beim

gepufferten Aufruf wird die eingetroffene Nachricht solange vom BS zwischengengespeichert, bis sie vom Empfänger aufgerufen werden kann.
Beim ungepufferten `receive(addr,&buf);` wartet der Empfänger auf eine Nachricht von Adresse `addr`. Kommt sie an, so wird sie in `buf` kopiert und *receive()* deblockiert. Hatte der Empfänger noch nicht *receive()* aufgerufen, geht die Nachricht verloren, denn das BS weiß nicht wohin damit. Der Empfänger müßte gleich zu Beginn *receive()* aufrufen und dann blockieren.
Beim gepufferten Aufruf richtet der Empfänger zu Beginn beim BS unter einer Adresse einen Briefkasten (mailbox) ein. Von dort kann er die Nachricht Stück für Stück abrufen.

4.3 Zuverlässige und unzuverlässige Aufrufe

Nachrichten können auch verlorengehen. Soll dies ausgeschlossen sein, müssen sie quittiert und evtl. neu gesendet werden. Also etwa:

1. request (client→server)
2. ack (server→client)
3. reply (server→client)
4. ack (client→server)

send() bleibt hier blockiert, bis die Quittung da ist. Man kann auch auf das erste *ack* verzichten, falls *reply* früh genug gesendet werden kann. Man kann auf das zweite *ack* verzichten, wenn man die Anfrage nach *time out* wiederholen kann.

4.4 Implementierung des Client/Server Modells

Wie üblich müssen Nachrichten u.U. fragmentiert werden, damit sie in die Pakete der zugrundeliegenden Netzwerkschicht passen. Ein Client/Server Protokoll kann z.B. folgende Nachrichten (Pakete) umfassen:

CODE	Pakettyp	Richtung	Funktion
REQ	request	C→S	Enthält Anforderung (Auftrag)
REP	reply	S→C	Enthält Bearbeitung des Auftrags
ACK	acknowledge	S→C	Quittiere empfangenes Paket
AYA	are you alive?	C→S	Lebst Du noch ?
IAA	I am alive	S→C	Server funktioniert
TA	try again	S→C	Server z.Z. überlastet
AU	address unknown	S→C	Kein Prozeß benutzt diese Adresse

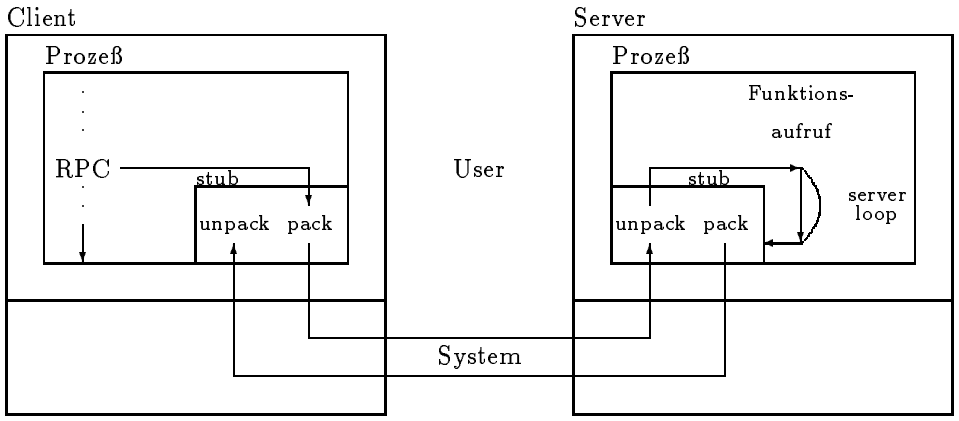
4.5 Remote Procedure Call (RPC)

Das RPC Kommando stellt den „fernen Funktionsaufruf“ als Prinzip des verteilten Rechnens zur Verfügung. Es handelt sich um ein spezielles Client/Server Modell. Ein Client kann eine Funktion des Servers aufrufen und es wird weitgehend transparent gemacht, daß Client und Server auf getrennten Maschinen laufen. Der Programmierer soll also z.B. nicht explizit Nachrichten verschicken müssen.

Implizit geschieht folgendes:
Prozedurname und Parameter werden in eine Nachricht verpackt, an den Server geschickt, dort ausgepackt, der Aufruf getätigt und das Ergebnis zurückgeschickt.

Kennt man die beteiligten Datentypen, so kann der entsprechende Code automatisch erzeugt werden (von einem Generator). Üblicherweise werden zwei Stummelprozeduren (stubs) erzeugt, die das Ein- und Auspacken, Senden und Empfangen

erledigen. Der Klient ruft den Clientstub auf, der den selben Namen trägt, wie die ferne Prozedur. Der benachrichtigt den Serverstub und blockiert, schickt das Ergebnis zurück; damit wird der Clietstub deblockiert, das Ergebnis wird ausgepackt und der Clientstub terminiert mit dem fernen Ergebnis als Ergebnis des Aufrufs. In heterogenen Netzen müssen die Daten natürlich in einer Standardrepräsentation verschickt werden (z.B. XDR).



Das Verpacken von Parametern nebst Konversion heißt **parameter marshalling** (aufreihen, zusammenstellen). Der Serverloop kann ebenfalls automatisch generiert werden. Der Server ruft in einer Endlosschleife den Serverstub immer wieder auf und bearbeitet so einen Auftrag nach dem anderen.

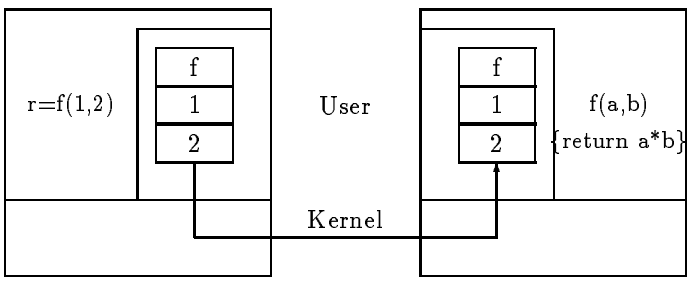


Abbildung 6: Parameter marshalling

4.6 Dynamisches Binden

Durch die RPC-Methode sind Aufrufer und Prozedur entkoppelt. Die beiden Programme können getrennt installiert werden. Außerdem macht der zugebundene Code das Programm nicht größer, und nicht benötigte Aufrufe treten nie in Erscheinung.
Zu Programmbeginn brauchen die Clientstubs ihre Partner noch nicht zu kennen. Wird einer aufgerufen, so wendet er sich zunächst an einen zentralen **Binder** (port mapper) und bittet um Nennung eines Servers (host & port), der die Prozedur in der benötigten Version zur Verfügung stellt.
Die Server melden sich beim Binder betriebsbereit, unter der Angabe von Name, Versionsnummer und Adresse. Der Binder reicht dem Client diese Information zur Laufzeit weiter und der Clientstub wendet sich direkt an den Port des Servers.

4.7 Fehlerbehandlung in RPC Systemen

Durch die Entkoppelung von Client und Server kann es zu folgenden Fehlern kommen:

1. Client findet Server nicht
2. Auftragsnachricht Client→Server geht verloren
3. Antwortnachricht Server→Client geht verloren
4. Server stürzt nach Auftragserhalt ab
5. Client stürzt nach Auftragsvergabe ab

Behandlungsmöglichkeiten:

Zu 1: Stub gibt Fehlermeldung. Wie?
Ein Fehlerwert, z.B. -1, genügt nicht, da es ein legales Resultat sein könnte. In manchen Sprachen gibt es **exceptions**, die Ausnahmen signalisieren. In C kann dies durch *signal handlers* und einem neuen Signal `SIG_NOSERVER` simuliert werden.

Zu 2: *lost request:* Der Sender startet einen **timer** und verschickt den Auftrag nach einem *timeout* neu. Evtl. mit Kennzeichnung als Duplikat (bei Zustandsänderung: Buchung).

Zu 3: *lost reply:* Manche Aufträge können problemlos wiederholt werden, andere nicht. Buchungsaufträge müssen als Orginal und Kopie gekennzeichnet werden.

Zu 4: *server crashes:* Vor Auftragserhalt siehe 1.
Spätere Abstürze vor oder nach Auftragsbearbeitung möglich. Im ersten Fall ist Wiederholung nach einem *reboot* möglich, im zweiten Fall nicht, falls sich ein Zustand geändert hat.
Der Client kennt aber die Absturzzeit nicht. Es gibt drei Konzepte der RPC-Abwicklung:

1. **at least once semantics:** Das RPC-System wiederholt den Auftrag so lange, bis ein Resultat zurück kommt.
2. **at most once semantics:** Das RPC-System bricht nach *timeout* mit einer Fehlermeldung ab.
3. **Keine Garantie:** Das RPC-System wiederholt Aufträge, aber gibt irgendwann auf. Der Auftrag kann nicht oder mehrmals bearbeitet werden.
4. **exactly once semantics:** läßt sich in Verbindung mit einem transaktionsbasierten Server realisieren.

Zu 5: *client crashes:* Ein Absturz während der Auftragsbearbeitung führt zu einer Waise (orphan, Prozeß ohne Eltern). Waisen brauchen Ressourcen (z.B. Zeit, Speicher, gemeinsame locks).
Methoden, um Waisen zu entfernen:

1. **extermination:** Notieren der Auftragsvergabe beim Client auf sicherem Medium. Nach reboot werden die Aufträge storniert. Problem:großer Aufwand, auch wenn nichts passiert.
2. **reincarnation:** Die Zeit wird in Epochen aufgeteilt. Jeder Clientreboot startet eine neue **Epoche**. Prozesse, die in der alten Epoche angestoßen wurden, werden storniert. Überlebt doch einer (z.B. verlorene Epochenmeldung), so tragen seine Resultate einen veralteten Epochenstempel.

3. **gentle reincarnation:** Server fragen bei Start einer neuen Epoche nach, ob die Eltern der Aufträge noch leben. Nur wenn sich die Eltern nicht melden, werden die Aufträge beendet.
4. **expiration:** Auftragsbearbeitung wird mit *timeouts* versehen. Nach *timeout* müssen sich die Eltern melden und dadurch den *timer* neu starten, ansonsten wird der Prozeß beendet.

5 Synchronisation in verteilten Systemen

Die Verteilung findet in Raum und Zeit statt. Spezielle Algorithmen müssen die Probleme lösen, die sich ergeben, weil z.B. keine gemeinsame Systemzeit vorhanden ist oder gemeinsame Mutex, Semaphor oder Condition-Variable fehlen.

Problemklassen sind:

1. Uhrsynchronisation
2. Gegenseitiger Ausschluß
3. Wahlverfahren
4. Atomare Transaktionen
5. Verklemmungen

Randbedingungen für verteiltes Rechnen:

1. Information ist über verschiedene Maschinen verteilt.
2. Es existiert keine synchrone Systemzeit
3. Prozesse basieren ihre Entscheidung auf lokaler Information.
4. Zentrale Abhängigkeiten mit zentralem Knackpunkt soll vermieden werden.

Punte 1. und 2. stellen das zentrale Problem verteilten Rechnens dar. Punkte 3. und 4. charakteriseren gute verteilte Algorithmen. Werden 3. und 4. verletzt, so hat man im Algorithmus eine inhärent sequentielle Komponente.

5.1 Uhrsynchronisation

Es ist prinzipiell unmöglich, mehrere (verteilt) unabhängig laufende Uhren völlig synchron zu halten, wenn die betrachteten Zeitquanten klein genug sind. Probleme sind, daß Uhren immer driften (clock skew) und daß (wegen endlicher Lichtgeschwindigkeit) die Laufzeit der Synchronisationspulse > 0 sind.

Das Beispiel **make** verdeutlicht die Wichtigkeit global synchroner Zeit. (Der neu veränderte Quellcode kann sonst ein früheres Datum tragen als der ältere Objektcode.)

5.1.1 Algorithmen

Es gibt einen internationalen Zeitstandard **UTC** (universal coordinated time), der in Paris geführt und über Funk verteilt wird. Allerdings können verschiedene Meschinen mit UTC Empfänger den Puls i.a. nicht zur exakt selben ns empfangen (da $c < \infty$).

5.1.2 Berkeley *UNIX* Verfahren

(aktiver timer-server ohne UTC)

Es gibt einen **time daemon**, der jede Maschine periodisch nach der Zeit frägt. Die Antworten werden gemittelt, und die Klienten werden instruiert, die Durchschnittszeit anzunehmen.
Das Verfahren kann dezentralisiert werden: jeder schickt seine Zeit an den anderen und bildet seinen Durchschnitt selbst.

5.1.3 Logische Zeit (logical clock)

Zentrale Überlegung ist, daß es sehr oft nicht auf absolut synchrone UTC Zeit ankommt, sondern nur auf **relativ** synchrone Zeit: ein Ereignis, das ein anderes beeinflussen kann, muß auch einen kleineren logischen Zeitstempel haben. Zwei verteilte Ereignisse können nur eine Abhängigkeit haben, wenn eine Nachricht übermittelt wurde.

5.1.4 Lamport's Algorithmus

Es wird zwischen Ereignissen die Abhängigkeitsrelation $a \rightarrow b$ (b ist potentiell abhängig von a):

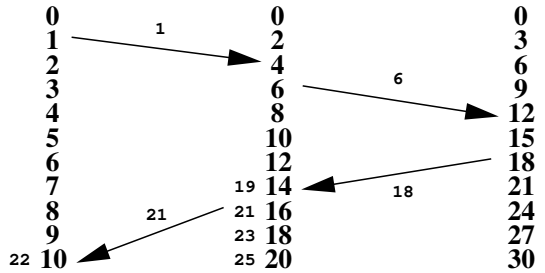
1. Falls a und b Ereignisse im gleichen Prozess sind und a vor b stattfindet, so gilt $a \rightarrow b$.

2. Ist a das Ereignis des Sendens einer Meldung m durch einen Prozeß und b das Ereignis des Empfangens von m durch einen anderen Prozeß, so gilt $a \rightarrow b$.

$\rightarrow$ ist eine Partialordnung: sind zwei Ereignisse unvergleichbar (weder gilt $a \rightarrow b$ noch $b \rightarrow a$), so heißen sie nebenläufig (concurrent).

Es wird nun eine logische Uhr C eingeführt, so daß $C(a) < C(b)$, falls $a \rightarrow b$. (D.h. insbesondere, daß Meldungen in der logischen Zeit nie rückwärts laufen.) Dies geschieht wie folgt: Jeder Prozeß P_i benutzt zunächst seine eigene Uhr C_i. Jede Nachricht m von P_i nach P_j trägt die logische Absendezeit $C_i(m)$ in P_i mit sich. Falls P_j beim Empfang feststellt, daß seine logische Zeit $C_j \leq C_i(m)$ ist, so schaltet er seine logische Uhr auf mindestens $C_j = C_i(m) + 1$ weiter. Jder Prozeß schaltet seine Uhr zwischen dem Senden zweier Nachrichten mindestens einen Tick weiter.

Falls in einem Zusammenhang nie zwei Ereignisse absolut gleichzeitig geschehen sollen, so kann man alle Prozesse durchnummerieren und die logische Zeit lexikographisch durch die Prozeßnummer erweitern: C_i wird zu $C_i.i$. Für $i \neq j, C_i = C_j$ gilt dann $C_i.i \neq C_i.j$.



5.1.5 Wechselseitiger Ausschluß

Zentralistische Lösung: Ein zentraler Serverprozeß vergibt die Zugangsberechtigung zu einem kritischen Abschnitt. Der Server nimmt Anforderungen entgegen. Er beantwortet die erste mit „OK“ und reiht die anderen in eine Warteschlange ein. Diese sind blockiert, solange sie auf „OK“ warten. Klienten melden Austritt aus kritischem Abschnitt an den Server.

RPC orientierte Sicht: `mutex_lock`, `mutex_unlock` laufen per RPC auf einem Server. Die Lösung ist einfach, hat aber den Server als zentralen Knackpunkt.

5.1.6 Verteilter wechselseitiger Ausschluß

Vorausgesetzt wird *Lamport's logische Uhr* mit $\rightarrow$ als Totalordnung und zuverlässige Kommunikation. Der Algorithmus geht auf *Ricart, Agrawala* und *Lamport* zurück. Um in einem kritischen Abschnitt (kA) zu kommen, schickt ein Prozeß P eine Nachricht mit dem Namen des kA, seinen eigenen Namen und der logischen Uhrzeit an alle beteiligten Prozesse im System. Danach wartet er auf das „OK" aller Prozesse (Bem.: Mehrheit genügt auch) und betritt den kA, sobald er es hat.
Empfängt ein Prozeß einen kA request, so gleicht er seine logische Zeit nach Lamport an und macht außerdem folgendes:

1. Wenn er selbst nicht am kA interessiert ist, schickt er „OK" zurück.
2. Wenn er im kA ist, stellt er die Beantwortung zurück.
3. Wenn er selbst auf den kA wartet, so vergleicht er die Zeit in der Nachricht mit der Zeit seiner eigenen Anforderung: Die frühere Zeit gewinnt: Ist seine eigene Zeit kleiner, stellt er die Beantwortung zurück, andernfalls schickt er „OK" zurück.

Verläßt der Prozeß den kA, so beantwortet er alle zurückgestellten Anforderungen mit „OK".

Ohne Beweis: Dieser Algorithmus ist frei von deadlock u. starvation.
Intuition: kein deadlock, da $\rightarrow$ als Totalordnung vorausgesetzt wurde. Keine starvation, da logisch synchrone Uhren. Eine Uhr kann nicht permanent hinter allen anderen nachgehen, da jede Requestnachricht sie wieder vorstellt und ihr nächster kA request dadurch später als die anderen zuvor eingereiht wird.

Kritik am Algorithmus: Braucht $2 \cdot (n-1)$ Nachrichten pro Eintritt in kA. Jeder Prozeß muß jede Anforderung genehmigen, also gibt es jetzt n Knackpunkte. Gleichermaßen muß jeder Prozeß die ganze Verwaltungslast tragen.

5.1.7 Ein Token Ring Algorithmus

Statt ein token immer wieder an die Zentrale zurückzugeben, können die Prozesse es unter sich im Ring weiterreichen. Das ist besser, falls viele in den kA wollen und schlechter, falls wenige in den kA wollen.

Vergleich

Alg.	Nachrichten pro Monitordurchgang	Verzögerung	Probleme
Zentral	3	2	Manager crash
Verteilt	$2 \cdot (n-1)$	$2 \cdot (n-1)$	Crash irgendeines Prozesses
Token Ring	unbegrenzt	max (n-1), min 0	Crash irgendeines Prozesses verlorenes token

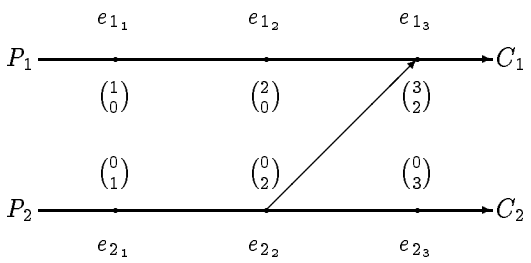
5.1.8 Vektoruhren (vector clocks)

Motivation: *Lamport's Uhren* haben den Nachteil, daß sie die Ereignisabfolge nur recht grob widerspiegeln. Es gilt

$$C(a) < C(b) \Rightarrow b \not\rightarrow a$$

aber man erkennt nicht, ob $a \rightarrow b$ oder nicht.

Beispiel:



Es gilt:

$$C(e_{2_2}) < C(e_{1_3}) \quad \text{aber} \quad e_{2_3} \not\rightarrow e_{1_3}$$
$$C(e_{2_2}) < C(e_{2_3}) \quad \text{und} \quad e_{2_2} \rightarrow e_{2_3}$$

Die Verfeinerung zu Vektoruhren erlaubt es, aus dem Zeitstempel direkt auf die Abhängigkeit zu schließen. Vektoruhren gehen unabhängig auf *Mattern* und *Fidge* zurück.

5.1.9 Implementierung

Die Uhrzeit ist nun nicht mehr eine einzelne Zahl, sondern ein Vektor, dessen Komponenten die logische Lokalzeit darstellen. Genauer enthält $C_i[i]$ die lokale Zeit in Prozeß i und $C_i[j]$ enthält die beste Näherung an die lokale Zeit in P_j, die Prozeß P_i erfahren hat. Noch genauer enthält $C_i(b)[j]$ die Zeit des letzten Ereignisses a in P_j, das kausal vor b geschah, d.h. wo $a \rightarrow \ldots \rightarrow b$ ex. Die Regeln für Vektoren sind:

V1: Falls $a \rightarrow b$ in P_i, dann ist $C_i(b)[i] = C_i(a)[i] + d, d > 0$

V2: Falls a in P_i eine Meldung m verschickt, so enthält m den Zeitstempel $t_m = C_i(a)$. Falls b in P_j eine Meldung m mit Zeitstempel t_m erhält, so wird $C_j(b)$ nachgeführt, gemäß $\forall k : C_j(b)[k] = max(C_j(b)[k], t_m[k])$

Durch Regel 2 erfährt P_j auch über transitive Abhängigkeiten, d.h. Zeiten, zu denen zuletzt Meldungen an P_i geschickt wurden usw.

Beispiel:

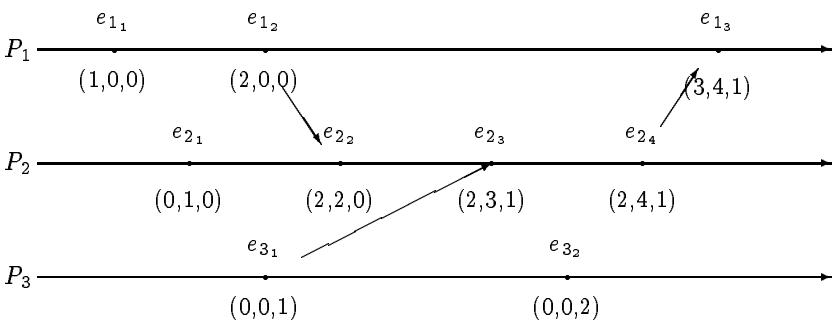


Abbildung 7: Raum-Zeit Diagramm

Vektorzeitstempel können wie folgt verglichen werden:

$$C(a) = C(b) \quad \Leftrightarrow \quad \forall i \, C(a)[i] = C(b)[i]$$
$$C(a) \leq C(b) \quad \Leftrightarrow \quad \forall i \, C(a)[i] \leq C(b)[i]$$
$$C(a) < C(b) \quad \Leftrightarrow \quad C(a) \leq C(b) \wedge \neg C(a) = C(b)$$

Es gilt: $C(a)\|C(b) \Leftrightarrow \neg(C(a) < C(b)) \wedge \neg(C(b) < C(a))$.
a und b sind *nebenläufig* (concurrent), falls $C(a)\|C(b)$.
a und b sind *kausal abhängig*, falls $C(a) < C(b) \vee C(b) < C(a)$.

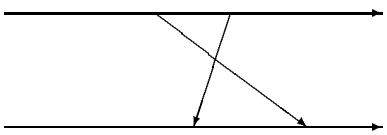
Für Vektoruhren gilt $a \rightarrow b \Leftrightarrow C(a) < C(b)$. Werden also z.B. Vektorzeitstempel protokolliert, so läßt sich die Ereignisfolge in einem verteilten Programm präzise rekonstruieren.

5.2 Kausale Nachrichtenordnung

Nachrichten sind in einem verteilten System (bei der Auslieferung) **kausal geordnet**, falls die „Empfangs"ereignisse die selbe kausale Ordnung haben wie die „Sende"ereignisse.

$$S(m_1) \rightarrow S(m_2) \Leftrightarrow R(m_1) \rightarrow R(m_2)$$

Diese Beziehung kann in nicht FIFOkanälen verletzt sein, z.B. packet switched networks. Im Raum-Zeit Diagramm erhält man sich überkreuzende Abhängigkeitspfeile in die gleiche Richtung.



Kausale Nachrichtenordnung ist wichtig, um die Datenbank zwischen updates konsistent zu halten. Das folgende Protokoll liefert eine Meldung erst dann aus, wenn auch die vorausgehende Meldung ausgeliefert wurde. Vektorzeitstempel werden benutzt, um das Senden und Empfangen von Nachrichten zu zählen. So kann man entscheiden, ob es eine vorausgehende Meldung gibt auf die gewartet werden muß.

5.3 Das Birman-Shiper-Stephenson Protokoll

Das Protokoll basiert auf **broadcasts**. Es wurde im ISIS System implementiert.
$C_{P_i}[i]$ zählt die Meldungen, die P_i verbreitet, $C_{P_j}[i], i \neq j$ zählt die von P_j empfangenen Meldungen mit Absender P_i.
Regeln:

1. Bevor Prozeß P_i die Meldung m verbreitet, inkrementiert er $C_{P_i}[i]$ und stempelt m.

2. Erhält ein Prozeß P_j von P_i eine Meldung m mit Zeitstempel C_m, so wird sie erst ausgeliefert, wenn die folgenden beiden Bedingungen erfüllt sind:

   (a) $C_{P_j}[i] = C_m[i] - 1$

   (b) $C_{P_j}[k] \geq C_m[k]\ \forall k \in \{1, 2, \ldots, n\}\backslash\{i\}$, n Anzahl der Prozesse

3. Wird eine Meldung an P_j ausgeliefert, so wird C_{P_j} nach Regel 2 für Vektoruhren nachgeführt.

Schritt 2 ist der Schlüssel zum Protokoll:
2(a) stellt sicher, daß P_j alle Meldungen von P_i gesehen hat, die m vorangehen.
2(b) stellt sicher, daß P_j alle Meldungen gesehen hat, die P_i vor dem Verbreiten von m selbst gesehen hat (broadcast).

6 Globaler Systemzustand

Da wir keine globale Zeit haben, können nicht alle Maschinen garantiert gleichzeitig ihren Zustand notieren (Inventur machen). Dies ist dann ein Problem, falls gleichzeitig eine verteilte Rechnung mit Kommunikation läuft. N.B.: Im verteilten System gehört der Zustand der Kommunikationskanäle zum Systemzustand.

Beispiel:

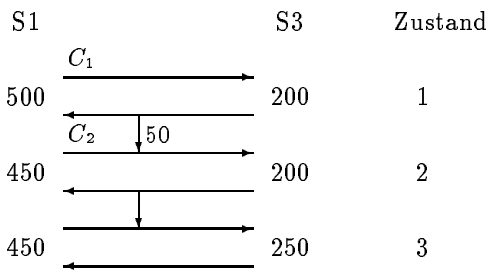


Abbildung 8: Banktransfer

Es muß bei S_1, S_2, C_1, C_2 Inventur gemacht werden, um den Systemzustand zu fixieren. Wird z.B. in S_1 im Zustand 1 Inventur gemacht, bei C_1, C_2 und S_2 aber in Zustand 2, so stimmt der Gesamtzustand nicht.

Ziel ist es nun, einen Globalzustand des verteilten Systems aufzunehmen, der aus **konsistenten** lokalen Zuständen einzelner Rechner und Kanäle besteht. Wir formalisieren:

Definition: *Lokalzustand*
Der Lokalzustand LZ_i jedes Rechnerorts S_i (site; computer, process) ist der relevante lokale Kontext der verteilten Anwendung.

Seien $send(m_{ij})$ und $rec(m_{ij})$ die Ereignisse des Sendens und Empfangens einer Meldung m_{ij} von S_i nach S_j. Wir definieren:

1. $send(m_{ij}) \in LZ_i \Leftrightarrow time(send(m_{ij})) < time(LZ_i)$
2. $rec(m_{ij}) \in LZ_j \Leftrightarrow time(rec(m_{ij})) < time(LZ_j)$

In-Transit:

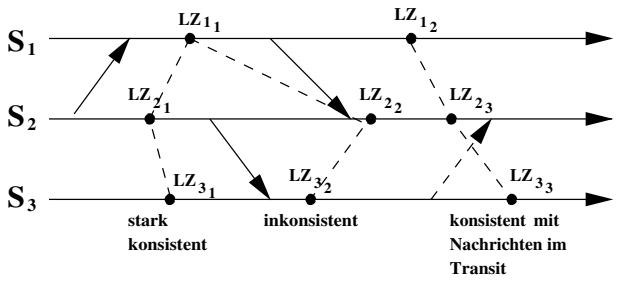
$$transit(LZ_i, LZ_j) = \{m_{ij} | send(m_{ij}) \in LZ_i \wedge rec(m_{ij}) \notin LZ_j\}$$

Inkonsistent:

$$inconsistent(LZ_i, LZ_j) = \{m_{ij} | send(m_{ij}) \notin LZ_i \wedge rec(m_{ij}) \in LZ_j\}$$

Eine Sammlung von Lokalzuständen ist inkonsistent, wenn sie inkonsistente Nachrichtenmengen enthält. (Es gibt Nachrichten, deren Empfang vermerkt wurde, deren Ausgang aber noch nicht vermerkt wurde.)

Transitfrei:
Ein Globalzustand heißt *transitfrei* $\Leftrightarrow transit(LZ_i, LZ_j) = \emptyset \ \forall i, j$. Ein Globalzustand ist *stark konsistent* (strongly consistent) gdw. er konsistent und transitfrei ist.



6.1 Der Chandy-Lamport Algorithmus

Der Algorithmus nimmt einen konsistenten Globalzustand auf; dieser wird auch als Schnappschuß (snapshot) bezeichnet. Es werden Marken benutzt, die keinen Einfluß auf die zugrundeliegende Berechnung haben. Die Kommunikationskanäle werden als FIFO angenommen.

Senderegel für Marken (für Prozeß P):

1. P nimmt seinen Zustand auf
2. P schickt eine Marke auf jeden Kanal, auf den er noch keine geschickt hat

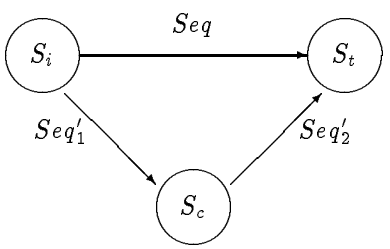
Empfangsregel für Marken (für Prozeß Q):
Wenn eine Marke in Kanal C empfangen wurde, verhält sich Q wie folgt:

1. Falls Q seinen Zustand noch nicht aufgenommen hatte
   (a) Nimm den Zustand von C als leer auf (leere Sequenz von Nachrichten)
   (b) Wende die Senderegel für Marken an
2. Falls Q seinen Zustand schon aufgenommen hatte
   (a) Nimm den Zustand von C auf als Sequenz der Nachrichten, die seither eingegangen sind

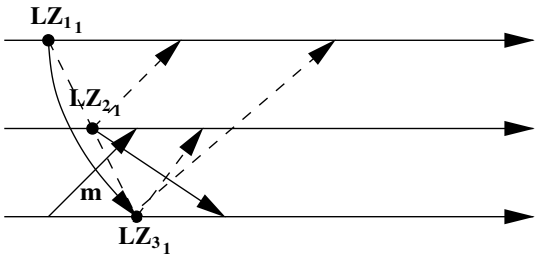
Der Algorithmus beginnt damit, daß ein Originator die Senderegel anwendet. Er endet, wenn jeder Prozeß über alle Kanäle eine Marke empfangen hat.
Die Marken dienen dazu, schon aufgenommene Meldungen von den noch aufzunehmenden zu trennen, und das Aufnehmen der Lokalzustände zu propagieren.

Der aufgenommene Globalzustand kann Meldungen im Transit enthalten (diese kann man dem Zustand entnehmen). Er kann aber keine inkonsistenten Nachrichten enthalten, da Marken und Berechnungsnachrichten sich wegen FIFO nicht überholen können.
Der aufgenommene Globalzustand muß nicht einem realen Systemzustand entsprechen, da die Berechnungen während der verteilten Zustandsaufnahme weiterlaufen.

Es gilt lediglich:



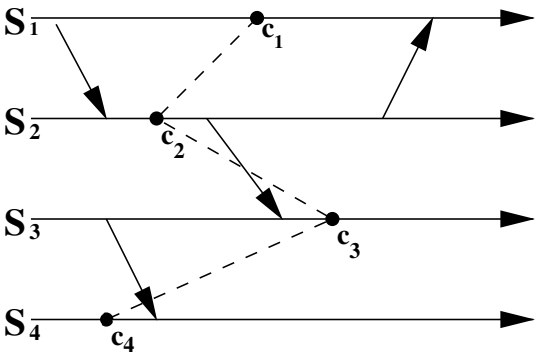
Ist S_c der aufgenommene Globalzustand zwischen den Zuständen S_i und S_t, und ist Seq die Sequenz der Aktionen, die S_i in S_t überführt, so gibt es eine Permutation $Seq' = Seq'_2 \circ Seq'_1$ von Seq so, daß Seq'_1 S_i nach S_t überführt.
Die Nützlichkeit des Globalzustands besteht hauptsächlich darin, **stabile** Systemeigenschaften zu erkennen, die unter jeder Permutation von Aktionen gelten, wie etwa Terminierung oder deadlock. Gilt die Eigenschaft in S_i, so gilt sie auch in S_c und wird bei der Aufnahme des Globalzustands erfaßt.



Globalzustand ist $\{LZ_{1_1}, LZ_{2_1}, LZ_{3_1}\} \cup \{C_{3_2} = m\}$

6.2 Schnitte einer verteilten Rechnung

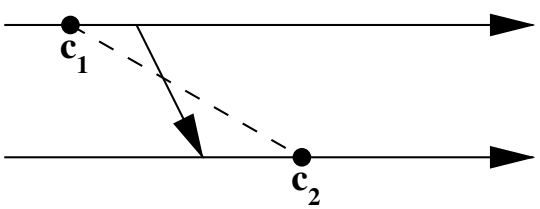
Ein **Schnitt** (cut) ist eine graphische Repräsentation eines Globalzustands im Raum-Zeit-Diagramm. Er besteht aus einer Linie, die die Schnittereignisse $C = \{c_1, c_2, \ldots, c_3\}$ verbindet.



Ein Schnitt ist konsistent $\Leftrightarrow$ es gibt zwischen zwei Schnittereignissen keine inkonsistenten Nachrichten.

Theorem: Ein Schnitt $C = \{c_1, c_2, \ldots, c_3\}$ ist konsistent gdw. es in C keine kausal abhängigen Ereignisse gibt.

Beweis(skizze):



Beispiel abhängiger Ereignisse
Sind die Ereignisse abhängig, so gibt es eine inkonsistente Nachricht. Gibt es keine inkonsistente Nachricht, so können die Ereignisse nicht abhängig sein.

6.3 Verteilte Terminierung

6.3.1 Systemmodell

Prozesse werden in *aktiv* und *inaktiv* (idle) eingeteilt. Nachrichten in Berechnungsnachrichten und Kontrollnachrichten des Algorithmus zum Feststellen der Terminierung. Die verteilte Berechnung hat terminiert, falls alle Prozesse inaktiv sind und sich keine Berechnungsnachrichten in Transit befinden.

6.3.2 1. Algorithmus

Wir verwenden den Chady-Lamport Algorithmus.
Ist ein Prozeß inaktiv, so initiiert er die Aufnahme des Globalzustands. Hat ein Prozeß über jeden Kommunikationskanal eine Marke empfangen, so teilt er dem Originator mit, ob er inaktiv ist. (Er ist selbst inaktiv und hat keine unerledigte Berechnungsnachrichten vorliegen.) Der Originator stellt Terminierung fest, falls alle Prozesse melden, daß sie inaktiv sind.
Dieser Algorithmus ist aufwendig, da i.a. mehrmals der Globalzustand aufgenommen wird.

6.3.3 Huang's Algorithmus

Die Idee ist, das Gewicht der noch zu erledigenden Arbeit zu protokollieren. Das Versenden einer Nachricht führt dazu, daß ein Teil des Arbeitsgewichts auf den Empfänger übergeht. Wird eine Arbeit erledigt, so kann der Gewichtsanteil zurückgegeben werden. Sind alle Gewichtsteile zurückgegeben, terminiert die Berechnung.

Der Algorithmus:
Es gibt einen Koordinator; sein Gewicht ist anfangs 1, und die anderen Prozesse haben Gewicht 0. $B(DW)$ ist eine Berechnungsnachricht mit Gewicht DW. $C(DW)$ ist eine Kontrollnachricht mit Gewicht DW.

Regel 1: Ein aktiver Prozeß mit Gewicht W schickt eine Berechnungsnachricht an P wie folgt: Finde $W_1 > 0, W_2 > 0$, so daß $W = W_1 + W_2$. $W := W_1$; Schicke $B(W_2)$ an P.

Regel 2: Empfangen von $B(DW)$ durch P mit W:
$W := W + DW$; Ist P inaktiv, so wird er jetzt aktiv.

Regel 3: Ein aktiver Prozeß P mit W wird inaktiv:
Schicke $C(W)$ an den Koordinator; P wird inaktiv.

Regel 4: Empfangen von $C(DW)$ durch Koordinator mit W:
Setze $W := W + DW$; ist $W = 1$, so herrscht Terminierung (falls Koordinator selbst inaktiv ist), da alle Arbeit erledigt ist.

Bem.: In der Praxis benutzt man $W = \frac{W}{2} + \frac{W}{2}$. Das kann effizient geschehen. *fork()* kann ähnlich durch Aufteilen der Gewichte realisiert werden.

6.3.4 Wahlalgorithmen

Viele verteilte Algorithmen benötigen irgendeinen speziellen Prozeß als Koordinator, Netzmanager o.ä. Im allgemeinen kann jeder der beteiligten Prozesse diese Rolle übernehmen, aber dieser muß irgendwie bestimmt werden. Bei Ausfällen muß dies dynamisch geschehen.

Wir nehmen an, daß alle Prozesse durchnummeriert sind (z.B. $< Netzadresse_process_id >$), daß die Nummern der beteiligten Prozesse bekannt sind und daß der größte Koordinator werden soll. Es ist aber nicht bekannt, welche Prozesse zu einem

Zeitpunkt laufen und welche abgestürzt sind. Die folgenden Alogorithmen stellen verschiedene Verfahren dar, um den momentan Größten zu bekommen.

6.3.5 Der Rüpelalgorithmus

Wenn ein Prozeß P bemerkt, daß der Koordinator nicht antwortet, so initiiert P eine Neuwahl wie folgt:

1. P schickt eine Wahlnachricht an jeden Prozeß mit einer höheren Nummer.
2. Falls niemand antwortet, gewinnt P und wird Koordinator.
3. Falls einer der Höheren antwortet, gibt P auf (der Höhere setzt die Wahl fort).

Falls P eine Wahlnachricht erhält, beantwortet er sie mit „OK“ und hält sofort seinerseits eine Wahl ab. Der letzte, der nicht aufgibt, wird Koordinator. Er informiert alle anderen mit einer Spezialnachricht.
Startet ein abgestürzter Prozeß neu, so hält er sofort eine Wahl ab, um zu sehen, ob er derzeit der Größte ist.

6.3.6 Ein Ringalgorithmus

Die lineare Ordnung des Rüpelalgorithmus kann auch zu einem Ring geschlossen werden. Prozeß P zirkuliert dann die Wahlnachricht auf dem Ring; nicht ansprechbare Prozesse werden übergangen. Jeder lebende Prozeß trägt seine Nummer in die Nachricht ein. Sobald die Nachricht zu P zurückkommt, bestimmt er den größten lebenden Prozeß und informiert alle vom Ergebnis.

Bem.: Es können zwei Prozesse ohne Konflikt gleichzeitig wählen lassen.

7 Atomare Transaktionen

Transaktionen sind ein höherer Synchronisationsmechanismus und im Datenbankbereich zu Hause. Sie sind **atomar**, d.h. eine aus mehreren Vorgängen bestehende Transaktion ist (logisch) nicht unterbrechbar. Transaktionen können aber scheitern (**fail**) (z.B. wenn keine Plätze mehr buchbar sind). In diesem Fall werden alle Teilvorgänge so zurückgenommen, daß die Transaktion keine permanente Seiteneffekte hinterläßt. Im Erfolgsfall (Festschreibung, **commit**) werden die Ergebnisse der Teilvorgänge in einem atomaren Vorgang festgeklopft, womit sie permanent wirksam werden. Z.B. geschieht dies auf permanentem Speicher wie Platten, Bändern, WORM etc.

7.1 Transaktionsbefehle

1. `BEGIN_TRANS`: Die folgenden Befehle bilden zusammen eine Transaktion
2. `END_TRANS`: Beende Transaktion und versuche Festschreibung
3. `ABORT_TRANS`: Abbruch der Transaktion. Rückkehr zum status quo ante
4. `READ` Datei
5. `WRITE` Datei

Beispiel:

```
BEGIN_TRANS
 if (reserve(ZRH,JFK) < 0) ABORT_TRANS;
 if (reserve(JFK,CMH) < 0) ABORT_TRANS;
END_TRANS
```

7.2 Transaktionseigenschaften

1. Serialisierbarkeit: Nebenläufige Transaktionen beeinflussen sich höchstens in dem Maße, wie es auch durch Nacheinanderausführung hätte geschehen können
2. Unteilbarkeit: Transaktion logisch atomar
3. Permanenz: Nach Festschreibung sind die Effekte der Transaktion permanent

Zu 1: Sind n nebenläufige Transaktionen gegeben, so darf das System sie nach einem beliebigen **Ablaufplan** (schedule) ausführen, aber nur so, daß der Gesamteffekt auch durch einen seriellen Plan mit atomaren Transaktionen hätte erzielt werden können.

Zu 2: Der Effekt der Transaktion nach außen ist entweder keiner (fail) oder der der gesamten Transaktion als Ganzes.

Zu 3: Nach einem *commit* sind die Ergebnisse der Transaktion nach außen festgeschrieben (auf permanentem Medium) und kein crash kann sie wieder rückgängig machen.

7.3 Implementierung

7.3.1 Privater Arbeitsraum

Die einfachste Lösung besteht darin, daß während der Transaktion zunächst ein privater Arbeitsraum (Konzeptpapier) benutzt wird. Am Ende werden dann in der commit-Phase die Ergebnisse auf das permanente Medium geschrieben.
Man kann z.B. im privaten Arbeitsraum die zu schreibenden Blöcke einer Datei als sog. Schattenblöcke aufbauen. Man benutzt einen neuen I-node (Kopie), arbeitet mit der Datei normal weiter und schreibt bei *commit* den Index des neuen I-node zurück. Bei *fail* passiert der Originaldatei überhaupt nichts.
Neben der größeren Effizienz minimiert man die Zeit des exklusiven Zugriffs auf den Index und somit auch die Zeit, innerhalb derer ein crash fatal wäre.

7.3.2 Vorausgeführtes Logbuch

Man führt ein Logbuch auf sicherem Speicher, wo man die Auswirkungen des Update bei nachfolgendem *commit* notiert (z.B. schreibt man die Ausgabe von **diff alt neu**). Ist das Logbuch korrekt geschrieben, macht man den eigentlichen Fileupdate und notiert das commit im Log. Gibt es beim Log-schreiben einen crash, so erkennt man den letzten unvollendeten Eintrag und wiederholt die Festschreibung aus den Log-Einträgen. Bei geeigneten Log-Einträgen kann man Änderungen auch zurücknehmen (roll back), z.B. bei Abbruch einer Transaktion.
Konzeptuell ist hier das Logbuch der Permanentspeicher und der normale HSP ist der Arbeitsraum.

7.3.3 Zwei Phasen Festschreibung

In einem verteilten System ist die atomare Festschreibung nicht trivial, da i.a. die erarbeiteten Daten verteilt gehalten werden (z.B. Auftragsdatenbank, Lagerhaltungsdatenbank, Produktionsplanungsdatenbank).
Im Zwei-Phasen-Protokoll gibt es einen bei der Transaktion federführenden Prozeß als Koordinator. Er notiert PREPARECOMMIT im Logbuch und benachrichtigt alle beteiligten Prozesse. Jeder Prozeß entscheidet, ob er zum commit tatsächlich bereit ist oder nicht. Er schreibt die Entscheidung ins Logbuch und sendet sie auch an den Koordinator. Dies beendet Phase 1.
Sind nicht alle bereit, notiert der Koordinator FAIL im Logbuch, ansonsten notiert er COMMIT und informiert die Prozesse über die Entscheidung. Im commit-Fall notiert jeder Prozeß selbst COMMIT im Logbuch und führt die Festschreibung durch. Dann informiert er den Koordinator über das Ende der Transaktion.
Durch die 1. Phase gibt es keine unkoordinierten commits einzelner Prozesse. Durch die Verwendung des Logfiles ist das Protokoll resistent gegen Abstürze.

8 Verteilte Verklemmungen

Prinzipiell ist das Problem das gleiche wie in Einzelprozessorsystemen. Es ist aber schwerer zu lösen, da die relevante Information über Prozeßabhängigkeiten jetzt verteilt ist. Es gibt prinzipiell vier Lösungsansätze:

1. Problem ignorieren (und im Zweifel Maschine neu booten)
2. deadlocks erkennen und dann (ohne reboot) beseitigen
3. deadlocks statisch vermeiden (durch geschickte statische Allokationsstrategien)
4. deadlocks dynamisch vermeiden (durch geschickte dynamische Allokation)

Wir diskutieren 2. und 3.

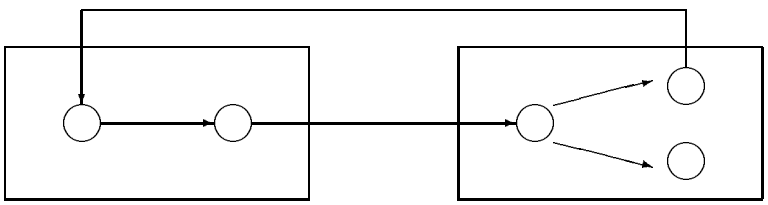
8.1 Erkennung verteilter Verklemmungen

8.1.1 Zentrale Erkennung

Ein Koordinator führt einen zentralen Abhängigkeitsgraphen. Die Systemteilnehmer informieren ihn durch Nachrichten über die Anforderung und Freigabe von Ressourcen. Hierbei kann es zu Pseudoverklemmungen kommen, falls eine Nachrichtensequenz (Freigabe, Anforderung) in der falschen Reihenfolge beim Koordinator ausgeliefert wird. (Freigabe wird aus versehen verspätet gemeldet.) Dies kann durch Ausputzen (flush) der Puffer vermieden werden, doch dies ist u.U. teuer.

8.1.2 Verteilte Erkennung

Wir verfolgen die Abhängigkeiten über Maschinengrenzen hinweg mit Nachrichten.



Der Chandy-Misra-Haas-Algorithmus verschickt immer dann eine Nachricht, wenn ein Prozeß wegen eines anderen blockiert.
Die Nachricht besteht aus $< Originator, Sender, Empfänger >$. Die erste Nachricht ist $< Originator, Originator, Empfänger >$. Ist der Empfänger blockiert, so muß er trotzdem die Nachricht an den oder die Prozesse verschicken, auf die er seinerseits wartet. Kommt einmal eine Nachricht zum Originator zurück, also $< Originator, Empfänger_n, Originator >$, so haben wir einen Zyklus und daher eine Verklemmung. Verklemmungen werden beseitigt, indem einer der beteiligten Prozesse seine Anforderung zurücknimmt. Es kann der Prozeß selbst terminiert werden, oder aber die Transaktion, die die Anforderung auslöste. Kennt man alle am Zyklus beteiligten Prozesse (z.B. durch ankleben der IDs an die Nachricht), so kann der zurückzutretende Prozeß durch Wahl bestimmt werden.
Ein Problem bei diesem Algorithmus ist, daß blockierte Prozesse die Nachricht behandeln müssen. Das kann durch einen separaten handler-thread geschehen. Dieser kann z.B. vom BS durch einen "upcall" erzeugt werden (pop-up threads), d.h. das BS ruft eine handler-procedure der Anwendung auf einem neuen thread auf.

8.2 Verklemmungsvermeidung

Eine statische Methode ordnet alle Ressourcen linear aufsteigend und erlaubt eine Aquisition nur in aufsteigender Ordnung, was Zyklen unmöglich macht. In der Praxis bedeutet dies aber, daß ein Prozeß wissen muß, welche Ressourcen er insgesamt braucht, denn sonst muß er häufig abbrechen.

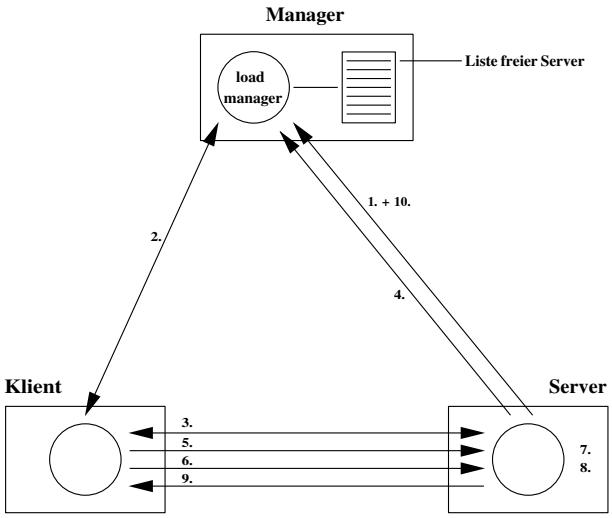
Eine andere Methode benutzt Lamport's verteilte Uhr, um zu einer strikt aufsteigenden linearen Ordnung der Anordnungen zu kommen. Dann vereinbart man, daß entweder nur ältere auf jüngere warten dürfen oder umgekehrt.

In der **wait-die** Variation wartet ein älterer Prozeß auf ein Betriebsmittel (BM), das ein jüngerer hält, aber ein jüngerer wird terminiert, wenn ein älterer das gewünschte BM hat.

Das kann mehrere **restart-kill** Sequenzen zur Folge haben, wenn der jüngere immer wieder gestartet wird. Werden Ressourcen immer aus Transaktionen heraus angefordert, so kann man einem Prozeß (Transaktion) auch ein BM entziehen, da lediglich die Transaktion abbricht. In der **wound-wait** Variation entzieht der ältere dem jüngeren das BM, und der jüngere wartet auf das BM des älteren. (Das verhindert die **restart-kill Sequenzen**.)

8.3 Prozessor-Allokation

Es gibt wieder die Alternative zwischen zentraler und verteilter Allokation.
Ein zentraler Algorithmus benutzt einen zentralen Manager, der eine Liste leerer Workstations hält. Auf Anfrage teilt er den Namen eines leeren Servers mit, und der Klient schickt den Auftrag direkt an den Server. Der Server meldet sich beim Manager an und ab.



1. Anmeldung
2. Anforderung mit Antwort
3. Auftrag mit Bestätigung
4. Abmeldung
5. Prozeßkontext
6. Prozeßstart
7. Prozeß läuft

8. Prozeß terminiert
9. Ergebnis zurück
10. Abmeldung

Das zentrale Verfahren ist (wie immer) bis zu einer gewissen Zahl von Workstations gut. Dann wird der zentrale Loadmanager zum Flaschenhals. Man kann es aber in einer Hierarchie fortsetzen, mit Verwaltern von Verwaltern etc.

Man kann Sender-initiiert oder Empfänger-initiiert vorgehen, insbesondere bei verteilten Lösungen, bei denen die Arbeit direkt zwischen den Wokstations verteilt wird. (Arbeit sucht Kapazität oder Kapazität sucht Arbeit.)
Verfahren 1 (verteilt) schickt den Prozeß zu einer zufällig gewählten Maschine. Verfahren 2 (verteilt) fragt bei zufällig gewählten Maschinen an, ob sie noch frei sind und schickt der ersten freien die Arbeit. Verfahren 3 (verteilt) ermittelt die am wenigsten belastete von n zufällig gewählten Maschinen und schickt den Prozeß dort hin.
Die Nutzung eines Pools als Einheit ist interessant, da der Durchsatz an Arbeit durch den Lastausgleich erhöht werden kann und da einzelne größere Anwendungen schneller gerechnet werden können, sofern sie verteilbar sind.

Die Motivation kann durch Warteschlangentheorie exakt angegeben werden. Sei λ die Häufigkeit $[\frac{1}{s}]$, mit der neue Prozesse generiert werden, und sei μ die Häufigkeit $[\frac{1}{s}]$, mit der sie bearbeitet werden. Das System ist stabil, wenn **im Mittel** $\mu > \lambda$.
Die mittlere Wartezeit auf fertige Prozesse beträgt

$$T = \frac{1}{\mu - \lambda}$$

Bei einem Pool aus n Workstations werden $n \cdot \lambda$ Prozesse generiert aber auch $n \cdot \mu$ bearbeitet. Dann ist

$$T_n = \frac{1}{n\mu - n\lambda} = \frac{1}{n(\mu - \lambda)} = \frac{1}{n} \cdot T$$

Wir haben also im Mittel einen n-fach besseren Durchsatz im System.

Index

accept(), 17
atomar, 35

big-endian, 4
Binder, 22
broadcast, 19
broadcasts, 29

checkpoints, 3
Client, 19
Client/Server Modell, 19
connectionless, 3
connections, 3
cron daemon, 4

domain, 10
    Internet, 10
    UNIX, 10

Einzelsystem, 1

file transfer, 4
frames, 3
ftp, 4

host, 14

Internet Protocol, 3
IP, 3
ISO, 2

little-endian, 4

name server, 19

offene Systeme, 2

parallel virtual machine, 4
paralleles System, 1
parameter marshalling, 22
pop-up thread, 20
ports, 7
Protokoll, 2
Protokolle, 2
Protokollkopf, 8

remote job entry, 4
remote login, 4
remote procedure call, 3
rlogin, 4
routing, 3

Schichten, 2
    application layer, 4
    data link, 3
    network, 3
    physical layer, 2
    presentation layer, 3
    session, 3
    transport, 3
select(), 20
Server, 19
sliding windows, 9
snapshot, 31
socket, 8
socket descriptors, 10
sockets, 3, 10
stream sockets, 16
stub, 21
    client, 22
    server, 22

TCP, 3
telnet, 4
timer, 23
token management, 3
Transceivers, 5

UUCP, 4

Vektoruhren, 28
verteiltes Rechnen, 4
verteiltes System, 1
virtual circuit, 3
virtuelles Terminal, 4

Wiederaufsetzpunkte, 3

Zugangskontrolle, 3