Ein Glossar wichtiger Begriffe zu verteilten und parallelen Systemen
Technical Report IFI-TR 93.44
Institut für Informatik der Universität Zürich

Clemens H. Cap, Silvano Maffeis, Lutz Richter, Volker Strumpen
{ cap, maffeis, richter, strumpen }@ ifi.unizh.ch

30. November 1993

64 Bit Adress Bereich:

Siehe (→ *Wide Address Spaces*).

Adaptive Parallelism:

Bezeichnet parallele Berechnungen, während derer die Anzahl der eingesetzten Prozessoren dynamisch variiert. Die Nutzung ungenutzter CPU-Leistung in Workstation-Netzwerken ist das derzeit wichtigste Einsatzgebiet dieser Technologie [LLM88, TL88].
LITERATUR: [CGKW93]

Agent Process:

Prozess, der als Schnittstelle zwischen einem lokalen Betriebssystem und dem Verbindungs-Netzwerk wirkt.

ARPA, ARPANET:

Advanced Research Projects Agency, U.S. Department of Defense, heute DARPA genannt; das ARPANET ist das erste und bekannteste globale Netzwerk Nordamerikas mit Verbindungen zu anderen Kontinenten, das in den frühen 80-er Jahren von DARPA initiiert wurde.

Asynchrones System:

Ein asynchrones verteiltes System kann wie folgt charakterisiert werden:

- Es gibt keine bekannte, obere Schranke für die Zeit die benötigt wird eine Nachricht zu übertragen.
- Die lokalen Uhren der einzelnen Komponenten des verteilten Systemes weichen um einen nicht bekannten Faktor voneinander ab.
- Es gibt keine bekannte obere und untere Schranke für die Zeit welche ein Prozess benötigt, um eine Anweisung auszuführen.

(→ *Synchrones System*)
LITERATUR: [HT93]

Atomic Multicast:

Atomic Multicast (auch *reliable multicast* genannt) garantiert, dass eine Nachricht bei sämtlichen Empfängern (→ *Prozessgruppe*) ankommt, trotz unzuverlässiger Netzwerkverbindungen und Ausfällen. LANs wie das Ethernet oder wie IBM Token-Ring sehen an sich Broadcast und Muticast vor, jedoch auf eine unzuverlässige Weise (Nachrichten können verloren gehen oder dupliziert werden). Atomic Multicast wird deshalb als Softwareschicht realisiert, um das multicasting zuverlässig zu gestalten. (→ *Multicast, Broadcast*).
LITERATUR: [HT93] [KTHB89] [BSS91]

Atomic Object:

Atomic Objects sind Objekte welche so implementiert sind, dass parallel ablaufende Prozesse (oder Threads (→ *Thread*)) auf das gleiche Atomic Object gleichzeitig zugreifen können. Zugriffs-Konflikte werden dabei vom Atomic Object vermieden, indem Synchronisations-Mechanismen wie read-write-locking (→ *Locking*) direkt im Objekt implementiert sind. Die Entwicklung paralleler Anwendungen wird damit erleichtert, da sich der Programmierer nicht mehr um die Synchronisation der Zugriffe auf gemeinsame Objekte kümmern müssen. Der Begriff des Atomic Object wird auch weitergehender verstanden. Beispielsweise ist in CAMELOT [EMS91] ein Atomic Object im weiteren dadurch charakterisiert, dass dessen Operationen atomar ablaufen: Eine Operation läuft entweder vollständig ab oder das Objekt wird automatisch auf den Zustand, den es vor Beginn der missglückten Operation hatte, zurückgesetzt (→ *Transaktion*).
LITERATUR: [Wei89, Seite 215 ff.] [Her86] [EMS91] [SDP]

Ausfälle:

Prozesse eines verteilten Systemes fallen gelegentlich, infolge von Software- oder Hardwarefehlern, aus. Um Ausfälle in verteilten Anwendungen charakterisieren zu können, sind in der Literatur diverse Modelle vorgeschlagen worden [Sch93, S. 22]:

Failstop: Ein Prozess fällt vollständig aus und bleibt dann in diesem Zustand stehen. Andere Prozesse können Ausfälle erkennen. Ausfallende Prozesse versenden keine verfälschten Nachrichten. Diverse Autoren gehen bei der Entwicklung von Modellen verteilter Systeme von der Failstop Annahme aus.

Crash: Ein Prozess fällt vollständig aus und bleibt dann in diesem Zustand stehen. Andere Prozesse können Ausfälle aber *nicht* erkennen.

Crash and Link: Ein Prozess fällt vollständig aus und bleibt dann in diesem Zustand stehen. Eine ausgefallene Kommunikations-Verbindung verliert Nachrichten, verzögert, dupliziert oder verfälscht diese aber nicht.

Receive Omission: Ein Prozess fällt aus indem er nur noch einen Teil der an ihn gerichteten Nachrichten empfängt oder er fällt vollständig aus und bleibt dann in diesem Zustand stehen.

Send Omission: Ein Prozess fällt aus indem er nur noch einen Teil der Nachrichten die er versenden sollte wirklich versenden kann, oder er fällt vollständig aus und bleibt dann in diesem Zustand stehen.

General Omission: Dieses Modell vereinigt das Receive Omission und das Send Omission Modell.

Byzantine Failures: Ein ausfallender Prozess kann auch Nachrichten verfälschen. Byzantine Failures implizieren zudem auch die anderen aufgeführten Modelle.

Batch Job System:

Verteiltes *batch processing* bezeichnet die dynamische Prozess-Allokation von Jobs auf einen ausgewählten Prozessor zur Minimierung der Laufzeit eines Jobs oder der mittleren Laufzeit aller Jobs. Einige Systeme unterstützen auch die Relokation von Prozessen auf andere Prozessoren, wenn der Prozessor überbelastet wird oder abstürzt.
LITERATUR: [BCH$^+$93, LLM88, The86, Tur93]

Berkeley Socket Interface:

Schnittstelle zwischen *BSD UNIX* (→ *BSD*) und dem Systemprogrammierer für den Zugriff auf die Kommunikationsprotokolle.
LITERATUR: [Ste90]

Binden (Binding):

Vorgang, durch den ein Dienst-Benutzer (→ *Client*) in Verbindung mit einem Anbieter (→ *Server*) eines benötigten Dienstes gebracht wird. Auf den unteren Protokoll-Schichten beinhaltet dies die Auswahl eines bestimmten Dienst-Anbieters und die Information über die physikalische Komponente, auf welche dieser verteilt wurde (→ *Trader*).

Broadcast:

(→ *Message Passing, Multicast*) Variante des selektiven Nachrichtenaustausches, bei der ein Rechner eine Nachricht an alle anderen Rechner sendet.
LITERATUR: [HH89]

BSD:

(University of California's *Berkeley Software Distribution*) Die UNIX Betriebssystemsversion der Universität von Berkeley heisst *BSD UNIX*.
LITERATUR: [Com91, Ste90]

Byzantine Failures:

Spricht man von der (→ *Fehlertoleranz*) von Systemen, so ist meist das Verhalten bei vollständiger Funktionsunfähigkeit einzelner Komponenten gemeint. Unter dem Schlagwort *byzantine failure* wird das Verhalten eines Systems untersucht, bei dem einzelne Komponenten sich zwar fehlerhaft verhalten, aber nicht gänzlich ausfallen. Bei Flugleitsystemen etwa untersucht man unter dem Schlagwort der Fehlertoleranz das Verhalten des Systems bei Totalausfall zB. des Bodenradars, unter dem Schlagwort der byzantine failures wird das Verhalten des Systems untersucht, wenn das Radar zwar funktioniert, in einem gewissen Sektor die Flugzeuge jedoch am falschen Ort darstellt. Der seltsame Name der byzantine failures stammt vom angeblich historischen Problem der byzantinischen Generäle: Vier Generäle belagern Byzanz. Aufgrund der Situation ist klar, dass sie grosse Vrluste erleiden werden, ausser sie entschliessen sich gemeinsam zum Angriff oder aber gemeinsam zum Rückzug. Die vier Generäle können durch zuverlässige Boten miteinander in Kontakt treten. Es herrscht jedoch der Verdacht, dass einer der Generäle ein Verräter ist. Dieser zeichnet sich dadurch aus, dass er etwa einem General sagt, er wäre für Angriff, einem anderen jedoch, er wäre für Rückzug. Der Verräter kann auch die Meinung eines Generals den anderen Generälen gegenüber falsch darstellen. Wie es bei Verrätern so üblich ist, zur Tarnung sagen sie auch manchmal die Wahrheit. Gibt es nun ein Verfahren, mit welchem die restlichen drei getreuen Generäle zu einem gemeinsamen Beschluss kommen können? Solche und ähnliche Fragestellungen spielen bei Systemen der Verkehrstechnik und der Raumfahrt eine grosse Rolle. Auch hier sind viele Komponenten redundant ausgeführt und einigen sich jeweils – auch in Gegenwart fehlerhafter Komponenten – auf ein gemeinsames Vorgehen. Beachten Sie bitte, dass die naheliegende Lösung eines Mehrheitsbeschlusses nicht funktioniert, denn gerade der Rechner, der die Stimmen zählt, kann der teilweise fehlerhafte sein.

Caching:

Das hauptsächliche Ziel des Caching ist die Zugriffszeit auf Objekte zu minimieren. Beispielsweise setzen immer mehr Prozessor-Architekturen zwischen Hauptspeicher und CPU einen schnellen, in der Regel kleinen und teuren Zwischenspeicher, das sogenannte Cache, ein. Ins Cache werden dann beispielsweise die Hauptspeicherblöcke kopiert, welche am häufigsten gebraucht wurden oder auf die zuletzt zugegriffen wurde. Speicherzugriffe müssen dann mit grosser Wahrscheinlichkeit nicht im langsameren Hauptspeicher erfolgen, sondern können aufs schnell arbeitende Cache umgeleitet werden.

Grundvoraussetzung für ein effektives Caching ist, dass auf einen relativ kleinen Anteil der Objekte relativ oft zugegriffen wird (*locality of reference* [Maf93d]).

Wo Caching ebenfalls eine wichtige Rolle spielt, ist in verteilten Filesystemen (→ *verteiltes Filesystem*). Will eine Client-Workstation auf ein File zugreifen, so erhält es vom Fileserver eine Kopie dieses Files, welches dann im Cache der Client-Workstation abgelegt wird. In diesem Falle ist beim Client in der Regel eine Harddisk als Cache Speicher vorgesehen. Ist der Platz auf der Harddisk aufgebraucht, so wird z.Bsp. das in der Vergangenheit am wenigsten benötigte File ausgelagert, wenn Platz für ein neu referenziertes File geschaffen werden muss. In einer ähnlichen Art und Weise arbeiten das ANDREW (Transarc Inc., Teil von OSF/DCE) und das CODA (Carnegie Mellon Univ.) Filesystem [Sat93].

Wird ein File nun aber verändert, so sind die sich in den Caches der Client-Workstations befindlichen Kopien des Files zu invalidieren oder ebenfalls zu ändern. Dazu gibt es vor allem zwei Techniken:

- Die Client-Workstations müssen vor jedem Öffnen einer sich in ihrem Cache befindlichen Filekopie den Server kontaktieren um zu ermitteln, ob die Kopie noch aktuell ist. Diese Methode ist zwar einfach zu realisieren aber ineffizient.

- Eine andere Methode besteht darin, mittels sogenannter *callbacks* zu arbeiten. Wird ein File von einem Client verändert, so informiert er den Server von der Änderung. Der Server handkehrum informiert sämtliche Clients, welche eine Kopie des Files im Cache haben, mittels eines Callback RPCs (→ *Remote Procedure Calling*). Dieses Schema wird im ANDREW und im CODA Filesystem verwendet und soll das Problem der Invalidierung effizient lösen. Callbacks können auch in Form von Multicasts (→ *Multicast*) versandt werden, um die Leistung des Filesystemes weiter zu erhöhen.

(Siehe auch (→ *Stashing*)).
LITERATUR: [Tai89] [Hay88] [Tan92] [Maf93a]

Callback:

Siehe (→ *Caching*).

Checkpointing:

Ausfalltoleranz in einer verteilten, parallelen Anwendung kann z.Bsp. dadurch erzielt werden, dass der Zustand der Prozesse in der Anwendung periodisch in einem nicht-volatilen Speicher abgelegt wird (checkpointing). Fällt nun ein Prozess aus, so kann dieser neu gestartet werden und mittels des letzten Checkpoints initialisiert werden. Das Hauptproblem des Checkpointing ist, dass diverse Prozesse in derselben Anwendung Checkpointing betreiben und deshalb die Konsistenz der Checkpoints gesichert werden muss. Beispielsweise kann das Checkpointing bei einem Sender und Empfänger währenddem sich eine Nachricht noch zwischen den beiden in einem Kommunikations-Kanal befindet, zu Inkonsistenzen führen. Die Prozesse müssen den Zeitpunkt an dem ein Checkpoint vollzogen werden soll vereinbaren. Diesbezügliche Protokolle werden in [EJZ92] vorgestellt. (siehe auch (→ *Prozessgruppe*)).

Circuit switching:

(*Leitungsvermittlung*) Vermittlungstechnik, die eine durchgehende Verbindung zwischen dem sendenden (rufenden) und dem empfangenden (gerufenen) Kommunikationspartner aufbaut. Diese Verbindung bleibt für die Dauer des gesamten Kommunikationsvorgangs erhalten und kann während dieser Zeit von weiteren Kommunikationsteilnehmern nicht benutzt werden.

Client:

Logische Komponente einer Anwendung, die einen Dienst anfordert. (→ *Client-Server Konzept*).

Client-Server Konzept:

Architektur, die eine Anwendung in Dienstanbieter (Client) und Dienstbenutzer (Server) strukturiert.
LITERATUR: [CD88]

Clustered Workstation Computing:

Die zunehmende Verbreitung von Workstations hat zu diesem neuen parallelen Programmier-Paradigma geführt. Vernetzte Workstations werden mit Software-Plattformen zu virtuellen Parallelrechnern zusammengeschaltet. Einen Überblick über derzeitig verfügbare Systeme gibt Turcotte in [Tur93].

Configuration:

(→ *Topography*) Die Abbildung der logischen Prozess-Topographie auf die physikalische Prozessor-Topographie legt die *Configuration* eines verteilten Systems fest. Ziel einer *guten* Konfiguration ist die Steigerung der Effizienz und Zuverlässigkeit eines Programms, ohne seine Semantik zu ändern.
LITERATUR: [HH89]

CORBA:

CORBA steht für *Common Object Request Broker Architecture*. CORBA ist ein händlerunabhängiger Standard welcher Mechanismen definiert, mit Hilfe deren Objekte (→ *Objektorientierte Programmierung*) in einem verteilten System Nachrichten austauschen können. Die OMG (*Object Management Group*) ist für die Entwicklung von CORBA zuständig. Die Firmen DEC, HP, HyperDesk, NCR, Object Desing und Sun unterstützen CORBA aktiv. Zur Zeit sind es bereits über 300 Firmen welche Mitglied der OMG sind. Ziel von CORBA ist, auf einfache Art und Weise Anwendungen in heterogenen verteilten Systemen entwickeln zu können. Hersteller sind frei eigene CORBA Implementierungen zu entwickeln und zu vermarkten.

CORBA sieht die Möglichkeit vor, Anwendungen welche auf CORBA Implementierungen unterschiedlicher Hersteller beruhen, zu vernetzen. Die OMG bietet einen CORBA-IDL (→ *Interface Definition Language*) Compiler kostenlos an, um die Verbreitung des Standards zu beschleunigen. Erste CORBA kompatible Produkte werden von DEC, Iona, SuiteSoftware, Sun, HP, HyperDesk und IBM vertrieben.

Kritiken an der jetzigen Version (1.1) des Standards beziehen sich vor allem auf das Fehlen von Objektgruppen-Mechanismen (→ *Objektgruppe*) und darauf, dass die konsistente Behandlung von Ausfällen (→ *Ausfälle*) im Standard noch nicht enthalten ist.
LITERATUR: [Dig91] [Bey93] [Isi93]

CSCW:

Computer Supported Cooperative Work (CSCW), wie der Begriff schon sagt, beschreibt die durch Computer unterstützte menschliche Arbeitsweise. Es geht dabei hauptsächlich darum, Gruppenarbeit mittels verteilter Systeme zu unterstützen. Beispiele von CSCW Anwendungen sind e-mail Systeme, Telekonferenz Systeme, Cooperative Software Engineering Systeme, Bulletin Boards usw.
LITERATUR: [EGG91] [DR93] [S+88] [RB91]

CSMA/CD:

(*Carrier Sense Multiple Access / Collision Detection*) CSMA-Netze sind lokale Netze, bei denen als Übertragungsmedium zumeist Koaxialkabel oder Lichtleiter benutzt werden. Typische CSMA-Netze sind sog. Ether-Netze. Bevor eine Station eines CSMA-Netzes zu senden beginnt, hört sie das Übertragungsmedium ab, ob bereits eine Übertragung stattfindet. Ist dies der Fall, so wartet sie, bis diese Übertragung zu Ende ist; andernfalls kann sie mit dem Senden beginnen. Wenn die sendende Station durch Mithören erkennt, dass gleichzeitig auch eine andere Station zu senden begonnen hat (Kollision), so wiederholt sie ihren Sendevorgang zu einem späteren, zufällig gewählten Zeitpunkt.

Data Representation:

Die Speicherung von Datentypen auf Bit-Ebene, d.h. die Festlegung der Bedeutung jedes einzelnen Bits im Speicher. Bekannte Byte-Ordnungen sind *Little Endian* und *Big Endian*.

Datagram:

In sich geschlossene Informationseinheit, welche über verbindungslose Protokolle wie (→ *IP*) oder (→ *UDP*) versandt wird. Im Unterschied zu Netzwerk-Paketen sind Datagramme von unterschiedlicher Grösse.
LITERATUR: [Com91, CS91, CS93b, Ste90]

Deadlock:

Von *Deadlock* spricht man, wenn ein Prozess auf ein Betriebsmittel wartet, das für seine weitere Arbeit unbedingt von Nöten ist. Er kann dieses Betriebsmittel aber nicht erhalten, weil es gerade von einem anderen Prozess belegt ist. Da sich dieser andere Prozess in einer ähnlich hoffnungslosen Situation befindet und wartet, ist mit einer Freigabe dieses Betriebsmittels auch nicht zu rechnen. Charakteristisch für ein Deadlock ist also ein Warten auf unteilbare Betriebsmittel, die von den Prozessen exklusiv genutzt werden (kein sharing), nicht vorzeitig zurückgegeben werden können und letztlich auf ein zyklisches aufeinander Warten führen. Bei zentralisierten Systemen können Deadlocks auftreten, wenn mehrere Prozesse sich gleichzeitig um Betriebsmittel bemühen. Bei verteilten Systemen gibt es dieses Phänomen ebenso, nur kann sich das Warten auf Betriebsmittel über ein physikalisch weit verteiltes Netzwerk erstrecken. Es ist deshalb ein komplexeres und schwieriger zu vermeidendes Problem als bei zentralisierten Systemen.

Dedicated System:

Im herkömmlichen Sinne steht ein dediziertes Rechnersystem einem einzigen Benutzer exklusiv zur Verfügung. Unter *dediziertem Workstation Cluster* [Tur93] verteht man ein Netzwerk von Workstations, die nicht interaktiv als Arbeitsplatzrechner, sondern zur Verringerung der Programmlaufzeiten für Parallelrechnungen oder den Batch Betrieb (→ *Batch Job System*) genutzt werden.

Design Patterns:

Design- und Programmier-Erfahrungen sind sicherlich wichtig um robuste, verteilte Anwendungen innert nützlicher Frist zu realisieren. Es hat sich gezeigt, dass es in der objektorientierten Programmierung (→ *Objektorientierte Programmierung*) Ideome und Klassenstrukturen gibt, welche allgemein eingesetzt werden können und dazu verhelfen, ein Systemdesign flexibler zu gestalten. Ein Beispiel eines wohlbekannten solchen Ideomes ist das *Model-View-Controller* Konzept in Smalltalk [PW88]. Erich Gamma [GHJV93] schlägt *design patterns* vor, um Design-Erfahrungen zu fixieren, zu dokumentieren und weiterzugeben. Wir gehen davon aus, dass eine Bibliothek von probaten *design patterns* für Design und Implementierung komplexer verteilter Systeme nützlich wäre. Diesbezügliche Bestrebungen sind unseres Wissens aber noch nicht im Gange.

Distributed Data Structure:

Eine verteilte Datenstruktur ist eine Datenstruktur, die von mehr als einem Prozess gleichzeitig manipuliert werden kann.

Distributed Operating System:

(*Verteiltes Betriebssystem*) Ein homogenes Betriebssystem, das identisch als eine Menge kooperierender Betriebssysteme auf den verschiedenen Knoten eines verteilten Systems implementiert ist.

Efficiency:

Die Effizienz gibt den mittleren Nutzungsgrad der Prozessoren während einer parallelen Berechnung an. Sie ist definiert als das Verhältnis aus Speedup (→ *Speedup*) zu Anzahl der beteiligten Prozessoren:

$$E(p) = \frac{S(p)}{p}.$$

LITERATUR: [BT89]

Fairness:

Bei einem (→ *Server*) treffen von mehreren (→ *Client*) Anforderungen zur Ausführung von Diensten ein. Der Server muss sich nun in einer bestimmten Reihenfolge diesen einzelnen Aufgaben widmen. Man nennt den Server *fair*, wenn er letzlich alle Anforderungen, die an ihn herangetragen werden, im Laufe der Zeit bearbeitet, also keine Anforderung unendlich lange warten muss. Wenn ein Server jedoch immer dann, wenn Client A eine Aufgabe für ihn hat, diesen Client allen anderen gegenüber bevorzugt, so kann dieser Client A den Server monopolisieren, wenn er nur rasch genug Anforderungen an den Server formuliert. Einen solchen Server nennt man *unfair*. Allgemein nennt man einen Agenden in einem System, der zu bestimmen Zeiten die (nichtdeterministische) Auswahl zwischen mehreren möglichen Aktionen hat *fair*, wenn jede Aktion, die möglich ist, irgendwann einmal auch ausgeführt wird. Im Rahmen der sogenannten Prozess-Algebren wird eine mathematisch präzise Definition dieses Begriffes gegeben.

Fehlertoleranz:

Ein System heisst fehlertolerant, wenn es auch bei Ausfall einzelner seiner Komponenten dem Benutzer gegenüber voll funktionsfähig bleibt. Spezielle und abgeschwächte Formen der Fehlertoleranz sind bekannt als (→ *Graceful degradation*) und als (→ *Byzantine failures*).

File Server:

(*Datei-Server*) Eine Bedienungseinheit in einem verteilten System, die Speicherkapazität und Datei-Verwaltungs-Funktionen zur Verfügung stellt.

Gateway:

Ein Kommunikationsprozessor, der eine Brücke zwischen zwei Netzwerken herstellt. Durch Gateways lässt sich die begrenzte Reichweite lokaler Netze durch kaskadierte Zusammenschaltung praktisch beliebig vergrössern.

Graceful Degradation:

Viele verteilte Systeme sind nicht voll fehlertoleran (→ *Fehlertoleranz*), können bei Ausfall einzelner Komponenten aber zumindest noch einen Teil der ursprünglichen Funktionalität gewährleisten. Man spricht in diesem Fall nicht von (→ *Fehlertoleranz*) sondern von graceful degradation.

Grand Challenges:

Probleme von weitreichender Bedeutung, nicht nur für die Wissenschaft, sondern auch für die Ökonomie und Soziologie, welche die Rechenleistung konventioneller Supercomputer um Grössenordnungen überschreiten. Typische *Grand Challange* Probleme stammen aus der Molekularbiologie oder der Klimamodellierung.
LITERATUR: [CSG+93]

Granularity:

Zum Begriff Granularität existieren zahlreiche Definitionen. Zumeist wird er intuitiv verwendet, um das Verhältnis der Ausführungszeit von Prozessen und den durch die Parallelisierung entstandenen *Overhead* zu charakterisieren. Die Parallelisierbarkeit eines Algorithmus wird sehr vage durch Begriffe wie *grobe* und *feine Granularität* angegeben. Nachwievor gibt es keine allgemein anerkannte präzise Definition dieses Begriffes.
LITERATUR: [KS88]

Happened-before:

Leslie Lamport [Lam78] definiert die *happened-before* Beziehung (dargestellt als $a \rightarrow b$, "a happens before b") für asynchrone Ereignisse in einem verteilten System wie folgt: (1) Falls Ereignis a und Ereignis b im selben Prozess stattfinden und a vor b abgelaufen ist, dann gilt $a \rightarrow b$. (2) Falls a das Senden einer Nachricht von einem Prozess und b das Empfangen dieser Nachricht von einem anderen Prozess ist, dann gilt wiederum $a \rightarrow b$. (3) falls $a \rightarrow b$ und $b \rightarrow c$ dann $a \rightarrow c$, das heisst, dass auch c von a *kausal* abhängig ist ($\rightarrow$ *Kausalität*).
LITERATUR: [Lam78]

Heterogeneity:

($\rightarrow$ *Homogeneity*) Im Bereich verteilter Rechnersysteme bezeichnet man ein Rechnernetz als *heterogen*, wenn die Rechner im Netz verschieden sind. Die Unterschiede reichen von relativ geringfügigen, wie unterschiedliche Leistung ($\rightarrow$ *Load Balancing*) bei Binärkompatibilität, bis hin zu unterschiedlichen Architekturen mit verschiedenen Datenrepräsentationen ($\rightarrow$ *Data Representation*), die bei der Kommunikation eine Konvertierung erfordern, unterschiedlichen Betriebssytemen, Netzwerk-Interfaces, und so weiter.

Heterogeneous Computing:

Dieser Ansatz zu effizientem Rechnen basiert auf der Idee, **eine** Applikation mit unterschiedlichen Rechenanforderungen in einem Netzwerk von Höchstleistungsrechnern mit unterschiedlichen architekturellen Charakteristiken auszuführen. Heterogenität bezieht sich auf die unterschiedliche Eignung einzelner Operationen bis hin zu Programm-Modulen in Hinblick auf Parallelisierung auf unterschiedlichen Rechnerarchitekturen, von *shared memory* Rechnern bis hin zu Workstation Netzwerken ($\rightarrow$ *Clustered Workstation Computing*).
LITERATUR: [FS93, KPS93]

Homogeneity:

($\rightarrow$ *Heterogeneity*) Im Bereich verteilter Rechnersysteme bezeichnet man ein Rechnernetz als *homogen*, wenn alle Rechner im Netz gleich sind.

Horizontale Separation:

Bei ($\rightarrow$ *Strukturierten Daten*) spricht man von horizontaler Separation, wenn die Daten nach Entitäten aufgeteilt werden: In jeder Partition der Daten befinden sich Entitäten mit jeweils allen Attributen. Wenn Sie sich die Daten in einer tabellarischen Form gespeichert vorstellen, bei der jede Zeile eine Entität darstellt und jede Spalte ein Attribut, so bedeutet horizontale Separation gerade das horizontale Durchschneiden dieser Tabelle.

HPF:

(*High Performance Fortran*) Ziel der HPF-Programmiersprache ist die Definition von Standarderweiterungen für Fortran, um daten-parallele Programmierung und höchste Rechenleistung auf Rechnern mit hierarchischen Speicherarchitekturen zu bieten.
LITERATUR: [HPF93]

Hypercomputer:

Die Rechner eines LANs werden durch Vernetzung und Nutzung ansonsten vergeudeter Betriebsmittel zu einem virtuellen Supercomputer umfunktioniert. Synonym für (→ *Metacomputer*).
LITERATUR: [CS93a]

Idempotent:

Eine Funktion, zumeist im Kontext eines Dienstes, der von einem Server ausgeführt wird, heisst idempotent, wenn die mehrfache Ausführung dieser Funktion dieselbe Semantik im Rahmen der Anwendung besitzt, wie eine einfache Ausführung dieser Funktion. Idempotente Funktionen gestatten eine sehr einfache Form der Fehlertoleranz im Kontext der (→ *Client-Server Architektur*): Falls der Client auf die Dienst-Anforderung keine Antwort erhält, so setzt er diese Anforderung ein weiteres Mal ab. Für den Fall, dass der Server bereits die erste Dienst-Anforderung korrekt durchgeführt hat und ein Fehler erst während des Übersendens der Bestätigung an den Client auftrat, wird eine wiederholte Ausführung der idempotenten Funktion nichts am korrekten Verhalten der Anwendung ändern. Bei Funktionen, die nicht idempotent sind, müssen bei der Fehlerbehandlung jedoch die folgenden Fälle unterschieden werden:

- Fehler des Servers noch vor dem Eintreffen der Dienst-Anforderung: Die Funktion wurde nicht ausgeführt.
- Fehler des Servers während der Ausführung der Funktion: Die Funktion wurde nur teilweise ausgeführt.
- Fehler des Servers nach Ausführng der Funktion aber vor Versenden der Bestätigung: Die Funktion wurde ausgeführt.
- Korrektes Verhalten des Servers, jedoch Verlust der Bestätigung: Die Funktion wurde ausgeführt.

Zudem ist bei einem idempotenten Server ein paket-orientiertes Transportprotokoll (wie etwa UDP), das manchmal eine mehrfache Verlieferung von Datenpaketen bewirken kann, unproblematisch.

IMP:

(*Interface Message Processor*) Kommunikationsknoten im (→ *ARPANET*).

Indeterminateness:

(*Unbestimmtheits*) Prinzip, das keine verbindliche Vorgabe einer bestimmten Reihenfolge von Ereignissen zulässt.

Interface Definition Language:

Eine Interface Definition Language (IDL) dient dazu, die Schnittstelle eines Dienstes eines verteilten Systemes, z.B. eines Nameservers (→ *Nameserver*), zu deklarieren. Bestandteile dieser Deklaration sind die RPCs (→ *Remote Procedure Calling*) welche der Dienst exportiert, die Parameter zu den einzelnen RPCs, Angaben für die Fehlerbehandlung (exceptions), Vererbungs-Beziehungen usw.

Mittels einer IDL wird ein deklarativer Programmierstil ermöglicht (→ *Komponenten-basierte Entwicklung*), bei dem die Details einer verteilten Anwendung vorerst ausser Acht gelassen werden und lediglich die Komponenten der Anwendungen im Zentrum stehen. Ein Stub-Generator kann aus einer vorgegebenen IDL Spezifikation den gesamten (→ *Marshaling*) Programmcode automatisch generieren. Ein Beispiel einer bekannten IDL ist CORBA-IDL (→ *CORBA*).

IP:

(*Internet Protocol*) Dieses Protokoll definiert die Kommunikation auf der Internet Netzwerkebene zum Austausch von *datagrams* (→ *Datagram*), der grundlegenden Informationseinheit, welche über das Internet

verschickt werden.
LITERATUR: [Com91, CS91, CS93b, Ste90]

IPC:

(*Inter-Process Communication*) Sende- und Empfangsoperationen zur Interprozess-Kommunikation sind entweder auf gemeinsamen Variablen definiert oder basieren auf dem Austausch von Daten mit Hilfe von Nachrichten. Die erste Klasse von Operationen wird vorzugsweise bei fest gekoppelten (→ *tightly coupled system*), die zweite bei lose gekoppelten (→ *loosely coupled system*) Multicomputer-System verwendet. Zugriffe auf gemeinsame Variable müssen synchronisiert werden, um deren Datenkonsistenz zu gewährleisten.

ISO:

(*International Organization for Standardization*) Das *ISO Reference Model of Open System Interconnection (ISO/OSI)* (→ *OSI*) beschreibt sieben konzeptuelle Ebenen für die Protokoll-Software.
LITERATUR: [Com91, Ste90]

Kausalität:

Kausalität, auch *Virtuelle Synchronie* [BJ89] ganannt, stützt sich auf der von Leslie Lamport [Lam78] eingeführten *happened-before* Beziehung ab (→ *Happened-before*). Kausalität ist ein fundamentales Konzept verteilter, paralleler Systeme. Es ist die Erweiterung der FIFO (First-In-First-Out) Ordnung von Nachrichten in Punkt-zu-Punkt Verbindungen, für Systeme in denen mehrere Sender mit mehreren Empfängern Nachrichten austauschen (→ *Prozessgruppe*). Kausale Ordnung von Ereignissen bedeutet, dass Nachrichten welche kausal voneinander abhängig sind, bei allen Empfängern in derselben Reihenfolge eingehen. Nachrichten welche nicht kausal voneinander abhängig sind, müssen nicht geordnet werden. Dies ist unproblematisch, weil die Empfänger keine Möglichkeit haben, diese Unordnung zu bemerken. (Hätten sie diese Möglichkeit, wären die Nachrichten per Definition kausal voneinander abhängig und das zugrundeliegende Synchronisations-Protokoll hätte bei allen Empfängern eine Gleichordnung bereits realisiert).

In einem verteilten System ist Kausalität ohne Einführung von Zusatznachrichten realisierbar. Ein relativ einfaches Vektor-Zeit Protokoll (siehe [Tan92, Seite 458 ff.]), welches lediglich Synchronisations-Information an die versandten Nutzdaten anfügt, reicht in der Regel aus, um Kausalität zu erreichen. Wahrung der Kausalität ist notwendig, damit Operationen in einem verteilten System logisch korrekt ablaufen. Eine kausale Ordung verhält sich zu einer strikten Ordung (→ *Strikte Ordnung*) ungefähr gleich, wie serialisierbare Transaktionen zu seriell ablaufenden Transaktionen.
LITERATUR: [Lam78] [BJ87] [BJ89] [BSS91] [vR93]

Komponenten-basierte Entwicklung:

Die Komponenten-basierte Entwicklung verteilter, paralleler Systeme sieht das System aus der Perspektive eines externen Betrachters (*outside-in approach*) und weniger von den inneren Mechanismen her, wie das häufig noch gemacht wird (*inside-out approach*). Bei der Komponenten-basierten Entwicklung steht vor allem das *was* im Vordergrung (deklarativer Ansatz) und weniger das *wie* (imperativer Ansatz). Verteilte Systeme werden nicht mehr als Sequenz von parallel ablaufenden Instruktionen gesehen, sondern als Menge von kooperierenden Komponenten (Objekten). Die inneren Mechanismen sind zwar immer noch wichtig, jedoch werden diese nun abstrahiert und erst wenn nötig wieder offengelegt. Ziel der Komponenten-basierten Entwicklung verteilter Systeme ist es "standard" Komponenten zu entwickeln, welche Schritt um Schritt zu komplexen verteilten Systemen zusammengefügt werden können. Dadurch sollen die Wiederverwendbarkeit, die Modularität und der Abstraktionsgrad gesteigert werden. (Siehe auch (→ *Design Patterns*)).
LITERATUR: [Weg93]

Konsistenz:

Bei Datenbanken bedeutet Konsistenz das Vorliegen gewisser, meist aus der Anwendung stammender Bedingungen für den Datenbestand. So muss etwa das Lebensalter einer Person stets positiv sein und bei einer doppelten Buchhaltung müssen jeweils die Buchungseinträge auf der Soll-Seite und der Haben-Seite vorhanden sein. Konsistenz tritt oft im Kontext von (→ *Transaktionen*) auf. Bei verteilten Systemen mit (→ *Replikation*) bedeutet Konsistenz, dass geeignete Algorithmen verhindern, dass die Anwendung eine unaktuelle, veraltete Kopie eines Datums liest.

Leichtgewichtiger Prozess:

(→ *Thread*)

Linearisierbarkeit:

Linearisierbarkeit (Linearizability) [HW90] ist eine Korrektheits Bedingung für Ereignisse, welche in einem objektorientierten (→ *Objektorientierte Programmierung*) verteilten System asynchron ablaufen. Das Modell der Linearisierbarkeit impliziert einen objektorientierten Programmierstil, in welchem Nachrichten an rechnerfremde Objekte gesandt werden (Operationen) und Resultate zurückkommen. Gemäss Herlihy und Wing [HW90] ist eine parallele Ausführung linearisierbar, wenn eine sequentielle Ausführung mit gleichen Abhängigkeiten zwischen den Ereignissen existiert. Dabei ist eine Operation b von einer Operation a abhängig, wenn b von einem Prozess ausgeführt wurde, welcher die Resultate von a empfangen hat. Zudem muss bei der parallelen Ausführung jede Operation das gleiche Resultat ergeben wie bei der sequentiellen Ausführung.

Das Modell der Linearisierbarkeit ist bedeutend weniger einschränkend als das der Serialisierbarkeit (→ *Serialisierbarkeit*), wodurch oftmals ein wesentlich höherer Parallelitätsgrad in einer Anwendung erzielt werden kann. Linearisierbarkeit baut auf dem Modell der Kausalität auf (→ *Kausalität*). Die oben erwähnte Abhängigkeit von Operationen ist im Grunde genommen gleich der *happened-before* Bedingung von Leslie Lamport (→ *Happened-before*).
LITERATUR: [HW90] [Bir93]

Livelock:

Prozesse, die zur Lösung einer Aufgabe miteinander kooperieren, müssen neben der Arbeit am eigentlichen Problem auch verschiedene Koordinationsaufgaben lösen. So sind etwa zu bestimmten Zeiten Daten auszutauschen und für empfangene Daten Bestätigungen zu versenden. Falls diese Daten oder solche Bestätigungen verloren gehen, so müssen Anforderungen zum erneuten Senden verschickt werden. Es kann nun sein, dass sich zwei Prozesse im Bemühen um eine derartige Koordination beliebig lange mit dem Versenden derartiger Status-Informationen aufhalten und keinen eigentlichen Fortschritt an der zu lösenden Aufgabe machen. Im Gegensatz zum (→ *Deadlock*) sind zwar alle Prozesse mit Arbeit beschäftigt, sie kommen aber mit der Lösung der eigentlichen Aufgabe nicht voran. Bei Programmen, die das Auftreten eines Livelocks gestatten, tritt dieses meist nur dann auf, wenn ganz bestimmte äussere Randbedingungen zufälligerweise erfüllt sind. Dies hat drei wichtige Konsequenzen: Zum einen tritt ein Livelock sehr selten tatsächlich auf. Tritt es auf, so ist es meist instabil: Nach einigen Sekunden oder Minuten sind die Bedingungen, die zum Auftreten des Livelock geführt haben, nicht mehr erfüllt, und die Prozesse können wieder mit der Bearbeitung der eigentlichen Aufgabe fortfahren. Insbesondere ist ein Livelock durch Testen prakisch nie zu finden. Bei Realzeitsystemen, welche die Bearbeitung bestimmter Funktionen innerhalb vorgegebener maximaler Ausführungszeiten gewährleisten müssen, muss man jedoch nachweisen können, dass das System keine Livelocks enthält. Hierzu werden tiefliegende mathematische Theorien benutzt.

Load Balancing:

Technik zur Minimierung der Antwortzeiten sowohl von seqentiellen als auch parallelen Programmen in Mehrprozessorsystemen. Spezielle Anforderungen an Lastverteilungsstrategien für parallele Programme in verteilten Systemen sind in [CS93a] beschrieben.

Locking:

Wenn parallele Prozesse auf dasselbe Objekt zugreifen, und mindestens einer dieser Prozesse das Objekt auch ändert, so bedarf es von Mechanismen welche die Zugriffe auf das gemeinsame Objekt synchronisieren. Eine bekannte Möglichkeit ist mittels Sperren zu arbeiten (*locking*).

Will ein Prozess (eine Datenbank-Transaktion z.Bsp.) auf ein gemeinsames Objekt zugreifen, kann er dieses exklusiv sperren. Nach erfolgter Sperroperation ist gewiss, dass der Prozess das Objekt solange exklusiv benutzen kann, bis er die Sperre wieder auflöst. Ist das Objekt bereits von einem anderen Prozess gesperrt, so wird in der Regel der nächste Prozess der das Objekt zu sperren versucht solange blockiert, bis das Objekt wieder freigegeben wurde.

Um Prozesse nicht übermässig zu blockieren, können auch Teilsperren (*shared locks*) auf Objekte ausgeübt werden. Mittels Teilsperren kann erreicht werden, dass mehrere Prozesse ein Objekt gleichzeitig lesen können, aber immer nur ein Prozess aufs Mal das Objekt ändern kann.

Um Deadlocks zu vermeiden, sollte ein Prozess sämtliche von ihm benötigten Sperren gleich zu Beginn seiner Ausführung verlangen. Sperren dürfen dann erst wieder freigegeben werden, nachdem alle notwendigen Sperren gesetzt worden sind. (Zwei-Phasen Sperrprotokoll).
LITERATUR: [BHG87]

Loosely coupled system:

Im Gegensatz zum (→ *tightly coupled system*) ein Multicomputer-System, bei dem alle Prozessoren über ihren privaten Hauptspeicher verfügen und die in der Regel auch ihr eigenes Betriebssystem haben.

Mail server:

Eine Bedienungseinheit in einem verteilten System, die den Benutzern eines verteilten Systems den gezielten Austausch von einem Brief ähnlichen Nachrichten erlaubt.

Marshaling:

Unter Marshaling versteht man das Übertragen der Parameter eines RPCs (→ *Remote Procedure Calling*) in einen Buffer, welcher dann anschliessend zum Empfänger des RPCs übermittelt werden kann. Die Übermittlung des Buffers geschieht mittels low-level Kommunikationsprimitiven wie `send()/receive()` und ist für den RPC Programmierer nicht ersichtlich.

Beim Marshaling sind auch Probleme der Datenkonversion (→ *Data Representation*) zu lösen, wenn Sender und Empfänger underschiedliche Datenformate vorsehen (z.Bsp. ASCII vs. EBCDIC, little endian vs. big endian usw.).

Beim Empfänger findet ein komplementärer Vorgang statt, in dem die Daten im erhaltenen Buffer für den Aufruf der effektiven Prozedur aufbereitet werden (Unmarshaling).

In der Regel wird der für einen RPC benötigte Marshaling-Programmcode automatisch aus der Spezifikation (→ *Interface Definition Language*) des RPCs von einem *Stub-Generator* erzeugt.
LITERATUR: [Nel81]

Massively Parallel:

Multicomputer- (→ *Multicomputer*) oder Multiprozessorsysteme (→ *Multiprocessor*) mit mehr als 1.000 Prozessoren.
LITERATUR: [Bel92]

Megaprogramming:

Megaprogramming erlaubt das Programmieren mittels Megamodulen, das heisst Komponenten, welche die Dienste, die von grossen Organisationen wie Fluggesellschaften oder Banken zur Verfügung gestellt werden, einkapseln. Ein Megamodul beinhaltet Daten, Funktionen und Wissen. Die Aktivitäten im Inneren laufen in der Regel parallel ab. Megamodule werden autonom verwaltet und sind über eine wohldefinierte Schnittstelle zugänglich, wobei die Schnittstellen-Deklaration mittels einer *Megamodule Definition Language (MDL)* vollzogen wird. Im Innern von Megamodulen können die diversesten Technologien angewendet werden, wie Datenbanken, Echtzeit Systeme, Parallelrechner, verteilte Systeme usw. Ziel des Megaprogrammings ist, die Entwicklung grosser Systeme (mit mehreren Millionen Programmzeilen), welche die unterschiedlichsten Technologien und Standards einbinden, zu unterstützen (→ *Komponenten-basierte Entwicklung*).
LITERATUR: [WWC92]

Metacomputing:

Nutzung von Netzwerken heterogener (→ *Heterogeneity*) Betriebsmittel auf der Basis einer Software-Ebene, die die einzelnen Netzwerkkomponenten transparent (→ *Transparency*) miteinander verbinden. Ziel dieses

Konzeptes ist die Bereitstellung von Betriebsmitteln zur Bearbeitung eines einzelnen Problems, das ohne einen Metacomputer praktisch unlösbar wïe, weil es einem einzelnen *general purpose* Computer beispielsweise an Rechenleistung oder an spezieller Hardware, wie zum Zugriff zu Massenspeichern oder zur Graphikverarbeitung, mangelt.
LITERATUR: [SC92]

Message Passing:

Bezeichnet den expliziten Austausch von Daten zwischen Prozessen mittels Nachrichten. Dieses Programmier-Paradigma umfasst eine Vielzahl von Formen des Punkt-zu-Punkt Nachrichtenaustausches sowie selektiver Kommunikationstechniken (→ *Broadcast, Multicast*). Die Eleganz dieses Konzeptes liegt darin, sowohl den Transfer von Daten als auch die Synchronisation von Prozessen gleichzeitig zu bewirken. Ein Nachrichtenaustausch heisst *synchron*, wenn Sender und Empfänger blockieren, bis die Sende- und Empfangsoperationen abgeschlossen sind. Ein Nachrichtenaustausch heisst *asynchron*, wenn mindestens einer der beiden Kommunikationspartner nicht blockiert. Die am häufigsten eingesetzte asynchrone Kommunikationsart verwendet eine nichtblockierende Sende- und eine blockierende Empfangsoperation, d.h. der Sender fährt nach Absenden der Nachricht mit der Programmabarbeitung fort, während der Empfänger wartet, bis die Nachricht zum Empfang vorliegt.
LITERATUR: [HH89, Sei90]

Message switching:

(*Nachrichtenvermittlung*) Eine Nachrichtenvermittlung, die den Kommunikationsweg in Segmenten alloziiert. Es existiert daher zu keiner Zeit ein vollständiger, dedizierter Weg vom Sender zum Empfänger.

MPI:

(*Message Passing Interface*) Das Ziel des MPI ist die Entwicklung eines Standards für Programme, die mit dem *Message Passing*-Paradigma (→ *Message Passing*) implementiert werden. Die wesentlichen Aspekte der Standardisierung sind Praktikabilität, Portabilität, Effizienz und Flexibilität.
LITERATUR: [MPI93]

Microkernel:

Traditionelle Betriebssysteme sehen in ihren Betriebssystem-Kernen umfangreiche Dienste vor. Beispielsweise sind im UNIX Kernel Kommunikations-Protokolle (TCP, UDP, IP), Filesystem-Dienste usw. implementiert. Der Vorteil eines solchen Ansatzes ist, dass zahlreiche Dienste ohne zeitaufwendige Kontextwechsel zwischen dem Kernel- und dem Benutzer-Adressraum angesprochen werden können.

Moderne Betriebssysteme sehen immer öfter sogenannte *Microkernel* vor. Die Idee dahinter ist, nur einen minimalen, effizienten Betriebssystem-Kern zur Verfügung zu stellen. Kommunikations-Protokolle, Filesystem Funktionen und andere Dienste werden dann in Form von eigenständigen *Services* dem System zur Laufzeit hinzugefügt. Vorteile des Microkernel Ansatzes sind:

- Die Flexibilität wird erhöht. Die Services können vom Programmierer umgestaltet und einfach zur Laufzeit neu installiert werden.
- Das Betriebssystem wird übersichtlicher und besser wartbar.
- Das Entwickeln neuer Dienste erfolgt gänzlich im Benutzer-Adressbereich und wird dadurch vereinfacht. Vor Allem das Debugging wird dabei wesentlich einfacher.
- Das Ausfallen eines Dienstes führt nicht mehr unbedingt zum Ausfall des gesamten Betriebssystemes.

Der Hauptnachteil von Microkernel Architekturen ist, dass der Zugriff auf einen Systemdienst zeitaufwendiger wird (Kontext Wechsel). Bei modernen Microkernel Architekturen wie Amoeba soll dies aber kaum

spürbar sein. Beispiele von Microkernel Betriebssystemen sind Amoeba, Chorus und Mach.
LITERATUR: [Mul93] [Tan92]

Migration:

Unter Prozess-Migration versteht man das Verlagern eines Prozesses auf eine andere Komponente eines verteilten Systemes. Der Prozess muss dafür zuerst "eingefroren" werden, das heisst jegliche Aktivität im Prozess muss angehalten werden. Nach erfolgter Migration läuft der Prozess bei der Instruktion weiter, wo er zuvor angehalten wurde. Sämtliche mit dem Prozess assoziierte Kontext Information (offene Files, Environment Variablen etc.) müssen mitmigriert werden.

Prozess-Migration wird vor allem zwecks Lastverteilung und zwecks Erhöhung der Verfügbarkeit eingesetzt. Beispielsweise können die Dienste eines Rechners, welcher zu Wartungszwecken abgeschaltet werden muss, zur Laufzeit migriert werden. Die Client-Applikationen sollten davon nichts merken und die Verfügbarkeit des Dienstes wird nicht unterbrochen.

Mirrowing:

Mirrowing ist eine spezielle Form der (→ *Replikation*), bei welcher darauf geachtet wird, die einzelnen physikalischen Kopien möglichst rasch konsistent zu machen. Ziel des Mirrowing ist es, jederzeit ein voll funktionsfähiges, aktuelles Ersatz-System zu haben, wenn das Primärsystem ausfällt. Mirrowing wird deshalb vorwiegend bei sicherheitskritischen Anwendungen wie der Raumfahrt, der Verkehrstechnik oder im Bankenbereich eingesetzt. Eine spezielle Form des Mirrowing findet sich bei Festplatten-Speichern: Hier werden alle Transaktionen auf der Primärplatte identisch auf einer Spiegelplatte nachvollzogen.

Multicast:

(→ *Message Passing, Broadcast*) Ein Rechner sendet eine Nachricht an eine definierte Menge von anderen Rechnern im Netzwerk. Umfasst diese Menge alle Rechner des Netzes, handelt es sich um ein *Broadcast*.
LITERATUR: [HH89]

Multicomputer:

(→ *Multiprocessor*) *Distributed Memory* Rechnerarchitektur, bei der das explizite Versenden von Nachrichten zwischen zwei Prozessoren nötig ist, um auf den Speicher des anderen Prozessors zuzugreifen. Damit ergibt sich eine hierarchische Speicherstruktur. Jeder Computer verfügt über sein eigenes Betriebssystem. Die Jobgrösse ist durch die Speicherkapazität des einzelnen Computers begrenzt.
LITERATUR: [Bel92, Sei90]

Multiprocessor:

(→ *Multicomputer*) *Distributed Memory* Rechnerarchitektur, bei der ein gemeinsamer Addressraum und konsistente Datenhaltung von der Hardware unterstützt werden, und ein gleichmässiger Zugriff auf den Speicher gewährleistet ist. Es wird ein einziges Betriebssystem mit einer allen Prozessoren gemeinsamen Warteschlange verwendet. Der Speicherbedarf eines Job's kann durch den virtuellen gemeinsamen Speicher auf alle Prozessoren verteilt werden.
LITERATUR: [Bel92, Sei90]

Multi-tasking/Multi-user Betriebssystem:

Ein Multi-tasking Betriebssystem erlaubt, mehrere Prozesse quasi-parallel auszuführen. Dabei erhält jeder Prozess vom Betriebssystem eine Zeitscheibe, Grössenordnung eine Zehntelsekunde [L^+89], zugeordnet. Währen dieser Zeit steht dem Prozess die CPU zur Verfügung. Ist die Zeitscheibe abgelaufen, oder musste der Prozess in Folge einer Vorrangunterbrechung (Interrupt) die CPU vorzeitig freigeben, ordnet das Betriebssystem die CPU einem anderen Prozess, welcher für die Ausführung bereit ist, zu. Es entsteht dabei die Illusion, dass jedem Prozess eine eigene CPU zur Verfügung steht. Dank Multi-tasking können CPU und I/O Geräte wesentlich besser genutzt werden und die Wartezeit bis ein Prozess bedient wird verringert sich.

Beispiele von Multi-tasking Betriebssystemen sind UNIX, Mach oder OS/2. UNIX und Mach lassen zudem auch zu, dass mehrere Benutzer gleichzeitig am selben Computersystem arbeiten (Multi-user Betriebssysteme).
LITERATUR: [SP91] [Ric85]

Multi-threading:

(→ *Thread*)

Nameserver:

Ein Dienst in einem verteilten System, welcher symbolische Namen von Objekten des Systemes in deren physikalische Adressen umwandelt und umgekehrt. Ziele des Nameserving in verteilten Systemen sind einen globalen Namensraum zu realisieren, Transparenz zu gewährleisten, den Zugriff auf Objekte zu vereinfachen und das gemeinsame Benutzen von Objekten zu ermöglichen.

Ein verteiltes System sieht normalerweise mehrere Nameserver vor. Jeder dieser Server verwaltet seinen bestimmten Namensbereich (*administrative domain*). Anfragen, die sich auf Objekte ausserhalb seines Bereiches beziehen, leitet der Nameserver dann an den Nameserver des entsprechenden Bereiches weiter.

Beispielsweise erhält ein Server als Auftrag den Namen `swissair.ch:/pub/flugplan7` aufzulösen. Der Auftrag wird an den Nameserver des Bereiches `swissair.ch` weitergeleitet, welcher als Antwort beispielsweise eine Adresse der Form `130.77.48.65,9047,17` zurückgeben wird.
LITERATUR: [Nee93] [Moc87]

Network Operating System:

Eine Menge von heterogenen Betriebssystemen, die vermittels (→ *agent processes*) in einem Verbindungs-Netzwerk kooperieren.

Objektgruppe:

Die Abstraktion von Prozessgruppen (→ *Prozessgruppe*) oder Thread-Gruppen (→ *Thread-Gruppe*) mittels objektorientierten (→ *Objektorientierte Programmierung*) Mechanismen wird zur Zeit rege diskutiert und vorangetrieben. Eine Objektgruppe umfasst eine Menge von Objekten, welche auf vorbestimmte Art und Weise kooperieren. Im Extremfall befindet sich jedes Objekt auf einer anderen Komponente eines verteilten Systemes. Objektgruppen können mittels Atomic Multicast (→ *Atomic Multicast*) angesprochen werden, das heisst dieselbe Nachricht kann mittels nur *einer* Operation gleich an eine Gruppe von Objekten übermittelt werden. Wichtige Vorteile von Objektgruppen sind, dass verteilte Systeme objektorientiert konzipiert und realisiert werden können, dass mittels Objektgruppen Ausfalltoleranz erreicht wird (Replikation), dass überlastete Objekte durch Objektgruppen ersetzt werden können ohne Client-Programme modifizieren zu müssen (Schrittweise Erhöhung der Parallelität, Loadsharing) und dass die Wiederverwendung und Wartbarkeit verbessert wird.
LITERATUR: [Maf93b] [Isi93]

Objektorientierte Programmierung:

Objektorientierte Programmierung (OOP) unterstützt Abstraktion, Kapselung, Polymorphismus und Vererbung.

Abstraktion bedeutet, dass ein System modelliert werden kann ohne dessen Details von Anfang an berücksichtigen zu müssen. OOP erlaubt eine deklarative Programmierweise, in der vor allem das “was”, also die Eigenschaften des Systemes im Vordergrund stehen. Bei anderen Programmiermethoden, z.Bsp. bei der imperativen oder prozeduralen Programmierung, steht meist das “wie”, also die innere Funktionsweise des

Systemes im Vordergrund.

Kapselung bedeutet, dass Daten und die zugehörigen Funktionen zu sauber abgegrenzten Objekten zusammengefügt werden können. Dank Kapselung kann die Implementierung eines Objektes von externen Aspekten abgegrenzt werden. Dadurch kann Software übersichtlicher gestaltet werden und die Wartbarkeit des Systemes wird verbessert.

Polymorphismus (dynamic binding) bedeutet, dass wir dieselbe Nachricht an unterschiedliche Objekte senden können und diese typspezifisch darauf reagieren. Beispielsweise kann dieselbe Nachricht `draw` an ein Kreis-Objekt und an ein Viereck-Objekt gesandt werden. Das Objekt "weiss" dabei selber wie es sich auf dem Bildschirm darstellen muss. Polymorphismus erlaubt oft Systeme übersichtlicher zu strukturieren und Kontrollstrukturen zu vereinfachen.

Vererbung bedeutet, dass neue Objekte die Eigenschaften bestehender Objekte übernehmen können. Dadurch kann vor allem die Wiederverwendung bestehender Software unterstützt und die Erweiterung komplexer Systeme erleichtert werden.

Beispiele von bekannten, objektorientierten Programmiersprachen sind C++, Eiffel, Modula-3, Oberon-2, Object-Pascal, Objective-C, Self und Smalltalk.
LITERATUR: [Mey88] [PW88]

OSI:

(*Open Systems Interconnection*) Beschreibt eine Familie von Protokollen zur Kommunikation zwischen Rechnern, die auf den ISO Standards (→ *ISO*) basieren. TCP/IP (→ *TCP, IP*) ist kein Protokoll eines Standardisierungs Kommittees, wie das ISO/OSI-Protokoll, sondern eine eigenständige Entwicklung aus der Forschung.
LITERATUR: [Com91, Ste90]

Packet Switching:

(*Paketvermittlung*) Nachrichten werden in Pakete gleicher Länge aufgeteilt und unabhängig voneinander übertragen. Wie beim (→ *message switching*) gibt es keinen vollständigen, dedizierten Weg vom Sender zum Empfänger.

Persistenz:

Ein persistentes Objekt ist dadurch gekennzeichnet, dass es über das Ende des Prozesses von welchem es generiert wurde erhalten bleibt. Ein persistentes Objekt existiert solange, bis es explizit gelöscht wird. Persistente Objekte werden in objektorientierten Datenbanken oder für das Checkpointing (→ *Checkpointing*) in ausfalltoleranten Anwendungen eingesetzt.

Polling:

(*Sendeabruf*) Systematisches Verfahren zur Ermittlung, ob Datenstationen Daten zur Übertragung bereithalten.

Port:

Ein Port ist eine Abstraktion, die eine einfache Form lokaler Orts-Transparenz (→ *Transparenz*) gewährleistet: Um einen Server zu kontaktieren braucht man nicht seine Prozess-Nummer zu kennen, man wendet sich nur an den Port, unter dem der betreffende Dienst angeboten wird. Im Gegensatz zur Prozess-Nummer, die von Maschine zu Maschine und von Tag zu Tag variieren kann, haben Dienste meist feste Port-Nummern. Im UNIX Betriebssystem stehen diese in der Datei `/etc/services` verzeichnet. Die Umsetzung von Ports auf Prozesse wird durch den Mechanismus des Portmappers bewerkstelligt. (→ *Trader*).

Primary Node:

Derjenige Knoten in einem verteilten System, der bei Veränderungen immer sofort aktualisiert wird, wohingegen die anderen Knoten erst zu einem späteren Zeitpunkt aktualisiert werden können.

Printer Server:

Eine Bedienungseinheit in einem verteilten System, die mehreren verteilten Benutzern den Zugriff auf gemeinsame Drucker ermöglicht.

Proxy:

Ein Proxy ist ein lokales Objekt, welches ein rechnerfremdes Objekt repräsentiert. Zugriffe auf das Proxy werden dabei zum rechnerfremden Objekt, welches mit dem Proxy gekoppelt ist, transparent weitergeleitet. Proxies werden vor allem dazu eingesetzt, um einen logisch lokalen Zugangspunkt für ein Objekt zu schaffen, welches rechnerfremd ist. Im weiteren können Proxies auch dazu eingesetzt werden, einen logisch nicht-replizierten Zugangspunkt für ein repliziertes Objekt (→ *Objektgruppe*) zu schaffen, lokales Caching durchzuführen, transparente Objekt-Migration (→ *Migration*) zu ermöglichen usw.
LITERATUR: [Maf93b] [S+89]

Prozess:

Grundsätzlich ist ein Prozess ein Programm, das sich in Ausführung befindet. Es besteht aus Code Segmenten, Daten Segmenten, Stack Segmenten, einem Prozess Kontrollblock usw. In einem Multi-tasking System (→ *Multi-tasking Betriebssystem*) kann ein Prozess grundsätzlich die Zustände blockiert, bereit und aktiv annehmen.
LITERATUR: [Ric85]

Prozessgruppe:

Für zahlreiche verteilte, parallele Anwendungen ist es nützlich, Prozessgruppen als Strukturierungs- und Programmier-Abstraktion zur Verfügung zu haben. Eine Prozessgruppe ist eine Menge von Prozessen, welche auf eine bestimmte Art und Weise kooperieren. Prozessgruppen sollten dynamisch verwaltet werden, das heisst Prozesse sollten zu Laufzeit einer Gruppe beitreten oder von einer Gruppe austreten können. Prozessgruppen werden mittels *atomic group multicast* [KTHB89] (→ *Atomic Multicast*) angesprochen. Das heisst, dass ein Sender mittels von nur *einer* Operation dieselbe Nachricht an alle Mitglieder der Prozessgruppe senden kann. Die Systemsoftware garantiert dabei, dass die Nachricht bei allen Teilnehmern oder bei keinem eingeht. Letzteres vor allem dann, wenn während des Multicasts ein Gruppenmitglied ausfällt.

Wichtig ist auch, in welcher Reihenfolge die Gruppenmitglieder Nachrichten Empfangen. Je nach Anwendung kann eine strikte Gleichordnung (→ *Strikte Ordnung*) aller Nachrichten notwendig sein, die Ordnung gar keine Rolle spielen oder eine kausale Ordnung (→ *Kausalität*) ausreichend sein.

Ein weiteres Problem ist das Erhalten einer konsistenten Gruppensicht. Prozesse in einem verteilten System treten den vorhandenen Gruppen dynamisch bei oder davon aus. Zu einem bestimmten Zeitpunkt können also die Sichten, welche die einzelnen Prozesse bezüglich den momentanen Gruppenzugehörigkeiten haben variieren. Es bedarf sogenannter *Group View Management Protokolle* [Ric92] und *Group Membership Servers*, um zu gewährleisten, dass in der Zeit in welcher eine Gruppensicht instabil ist, keine Multicasts ausgeführt werden. Eine Gruppensicht ist z.B. dann instabil, wenn ein Prozess einer Gruppe hinzugetreten ist und die daraus resultierende neue Gruppensicht gerade zu den Gruppenmitgliedern propagiert wird. Ein bekanntes System, welches die Programmierung mit Prozessgruppen erlaubt, ist ISIS [B+90].
LITERATUR: [Tan92, Seite 445 ff.] [Bir91]

Reliable Multicast:

Siehe (→ *Atomic Multicast*).

Remote Method Calling (RMC):

Darunter verstehen wir eine objektorientierte (→ *Objektorientierte Programmierung*) Variante des wohlbekannten RPC Paradigmas [Nel81] (→ *Remote Procedure Calling*). RMC unterscheidet sich davon, dass ein objektorientierter Programmierstil erlaubt wird. Über lokale *Proxy* Objekte (→ *Proxy*), kann der Programmierer die Methoden eines rechnerfremden Objektes aufrufen. Als Parameter können komplexe Objekte dienen. Name-overloading, Vererbung und Polymorphismus gelten nun auch für die objektorientierte Programmierung verteilter Systeme. Mittels desselben RMC Mechanismus können auch Objektgruppen (→ *Objektgruppe*) angesprochen werden, wodurch erreicht wird, dass ein Einzelobjekt auf einfachste Art und Weise durch eine Objektgruppe ersetzt werden kann, um Ausfalltoleranz oder Leistung einer verteilten Anwendung schrittweise zu erhöhen.
LITERATUR: [Isi93] [Maf93e] [Maf93c]

Remote Procedure Calling (RPC):

Der Grundmechanismus, welcher der Kommunikation in verteilten Systemen zugrundeliegt, ist input/output. Mittels Operationen wie `send()` und `receive()` können dabei Nachrichten zwischen Prozessen versandt und empfangen werden. Input/output ist aber meist kein geeignetes Programmiermodell, um komplexe, verteilte Anwendungen zu realisieren: Der Abstraktionsgrad ist zu tief und das Konzept führt in Programmen, welche mittels Prozeduren, Modulen oder Klassen strukturiert sind, oft zu einem Stilbruch.

Nelson [Nel81] erkannte dieses Problem anfangs der 80'er Jahre und erarbeitete das nun weitverbreitete Modell des *Remote Procedure Calling (RPC)*. RPC erlaubt Prozeduren eines rechnerfremden Programmes aufzurufen. Die Syntax eines Aufrufes ist dabei derjenigen eines herkömmlichen Prozeduraufrufes ähnlich oder gar identisch dazu. Vorteile des RPC sind, dass das Kommunikations-Modell für Programmierer, welche mit einer prozeduralen Sprache vertraut sind, einfacher verständlich ist, dass Programme übersichtlicher und besser lesbar gestaltet werden können, dass das Marshaling (→ *Marshaling*) von RPC Parametern meist automatisch erfolgt und dass Typenprüfung der RPC Parameter zu compilierzeit vorgenommen werden kann.

Probleme des RPC sind die Semantik eines RPCs angesichts von Ausfällen und Pointer (Zeiger) als RPC Parameter (→ *Remote Method Calling*).
LITERATUR: [Nel81] [Tan92] [ATK92] [TvR88]

Replikation:

Unter Objekt-Replikation versteht man das verfielfältigen einer Software-Komponente eines verteilten Systemes. Ein Ziel der Replikation ist es, die Verfügbarkeit zu erhöhen. Beispielsweise kann auf Daten eines replizierten Files solange zugegriffen werden als ein Replikat zugänglich ist. Ein anderes Ziel der Replikation ist die Minimierung der Zugriffszeit auf ein Objekt. Wird auf ein rechnerfremdes File oft zugegriffen, kann eine lokale Kopie davon erstellt werden (→ *Caching*) und fortan auf diese lokale Kopie zugegriffen werden. Das Hauptproblem der Replikation ist die Konsistenzerhaltung der Replikate, sobald Replikate auch verändert werden können. Während der letzten zwanzig Jahren wurden zahlreiche Algorithmen zur Konsistenzerhaltung vorgestellt [Cap90, Dav89a, Dav89b, DGMS85, Her86, HW90].

Replikationsgrad:

Die Anzahl Replikate (→ *Replikation*) eines Objektes, welche zum Betrachtungszeitpunkt im System vorhanden sind.

RPC:

Siehe (→ *Remote Procedure Calling*).

Scalability:

Dieser Begriff fasst eine Reihe von Teilaspekten zusammen, über die die derzeitige Forschung Aussagen dahingehend zu gewinnen versucht, ob ein sehr grosser Computer realisierbar und sinnvoll ist. *Size Scalability* bezeichnet die Eigenschaft linearer Leistungssteigerung mit wachsender Anzahl von Grundkomponenten wie Prozessoren, Speicher und Netzwerkschalter. *Generation Scalability* ist die proportionale Skalierbarkeit aller Rechnerkomponenten über technologische Generationen hinweg. *Problem Scalability* bezieht sich darauf, ob eine bestimmte Anwendung eine Maschine bestimmter Grösse effizient ausnutzen kann.
LITERATUR: [Bel92, Sei90]

Separation:

Mit *Separation (separation)* wird die funktionale und/oder operationelle Trennung von Programmen in einzelne Komponenten bezeichnet. Diese können dann auf unterschiedlichen physikalischen Komponenten eines verteilten Systems bearbeitet oder gespeichert werden. Diese Sicht auf eine Anwendung verlangt vom Programmierer die explizite Verwendung von Kommunikationsanweisungen sowie von System-Management- und Integrationstechniken (→ *Transparenz*).
LITERATUR: [CD88]

Serialisierbarkeit:

Wenn zwei oder mehr Transaktionen (→ *Transaktion*) parallel ablaufen, so können Zugriffe auf gemeinsame Datenobjekte zu Konflikten und Inkonsistenzen führen. Um Konflikte zu vermeiden könnten die Transaktionen *seriell* abgewickelt werden, das heisst eine Folge-Transaktion wird erst dann gestartet, wenn ihre Vorgänger-Transaktion komplett beendet ist. In diesem Fall würden die Ressourcen des Datenbanksystems aber sehr schlecht genutzt und die Leistung des Systems liesse zu wünschen übrig.

Aus diesem Grund verwendet man Algorithmen, die zwar eine parallele Bearbeitung der Aufträge gestatten, der Anwendung aber stets eine Sicht auf den Datenbestand vorspiegeln, der sich ergäbe, wenn die Aufträge sequentiell bearbeitet würden. In diesem Sinne heissen parallele Auftragsbearbeitungen *serialisierbar*, wenn sie nach aussen gleich wie eine geeignete sequentielle Bearbeitungsreihenfolge aussehen.
LITERATUR: [BHG87]

Server:

Logische Komponente einer Anwendung, die einen Dienst (service) anbietet. (→ *Client-Server Konzept*).

Service:

Unter einem Service verstehen wir einen Dienst in einem verteilten System. Ein verteiltes System besteht aus mehreren solchen kooperierenden Services. Es ist die abstrakte Definition der Aufgabe welche ein Server für Client-Programme wahrnehmen kann. Ein Service kann in Form eines Objektes oder einer Objektgruppe (→ *Objektgruppe*) (eines Prozesses oder einer Prozessgruppe) implementiert werden. Ein Service wird in der Regel mittels einer Spezifikations-Sprache (→ *Interface Definition Language*) definiert. Ein Stub-Generator kann aus dieser Spezifikation automatisch den Code erzeugen, welcher es den Client-Programmen ermöglicht, den Service zu kontaktieren (→ *Marshaling*), einen Service zur Laufzeit zu replizieren usw.

Singleton Object:

Ein Singleton Object versteht sich als Gegenteil einer Objektgruppe (→ *Objektgruppe*). Es stellt lediglich ein Einzelobjekt dar. Ein Ziel der Objektgruppen-orientierten Programmierung verteilter Systeme ist, dass verteilte Anwendungen in Form von kooperierenden Singleton Objects konzipiert und realisiert werden

können, und dass ein Singleton Object dann durch eine Objektgruppe ersetzt werden kann ohne dass Client Programme verändert werden müssen (→ *Remote Method Calling*).
LITERATUR: [Isi93]

Speedup:

(→ *Efficiency*) Ein Mass, das angibt, um wiviel schneller ein paralleler Algorithmus unter Verwendung von p Prozessoren im Vergleich zu seiner sequentiellen Variante mit nur einem Prozessor ist. Ist $T(p)$ die Laufzeit eines Algorithmus bei Verwendung von p Prozessoren, so ist der *Speedup* definiert als

$$S(p) = \frac{T(1)}{T(p)}.$$

LITERATUR: [BT89]

SPMD:

(*Single Program, Multiple Data*) Programmier-Modell zur Ausnutzung von Parallelität auf der Ebene von parallelen Datenstrukturen, die über einem einzigen Adressraum verteilt sind. SPMD lehnt sich an die bekannten Parallelrechner Taxonomie von M. Flynn an.
LITERATUR: [Bel92][Fly66]

Stashing:

Stashing ist eine zum Caching (→ *Caching*) analoge Technik. Das Stashing versucht aber zusätzlich Objekt-Zugriffe im Voraus zu ermitteln und die im nächsten Zugriff am wahrscheinlichsten benötigten Objekte vorher ins Cache zu laden. Erfolgt nun der antizipierte Zugriff, so liegen mit hoher Wahrscheinlichkeit die benötigten Objekte bereits im Cache vor. Ziel des Stashing ist Leistung und Verfügbarkeit gegenüber herkömmlichem Caching zu erhöhen.
LITERATUR: [Tai89]

Status-behaftet:

Ein (→ *Server*) heisst Status-behaftet, wenn er für sein korrektes Funktionieren Status-Information seiner (→ *Client*) speichern muss. Bei einem Status-behafteten Server geht bei Versagen des Servers diese Information über die Clients verloren, was ein Wiederanlaufen erschweren kann. Ein Status-behafteter Server ist etwa ein File-Server, der sich für jeden seiner Clients die aktuelle Position des Datei-Zeigers merkt. Die Schnittstelle eines solchen File-Servers kann deshalb eine Funktion der Form `read (datei, buffer, zahl-zu-lesender-bytes)` beinhalten, da sich der Server um die jeweilige Position in der Datei selber kümmert.

Status-frei:

Ein (→ *Server*) heisst Status-frei, wenn er für sein korrektes Funktionieren keinerlei Status-Information seiner (→ *Client*) speichern muss. Bei einem Status-freien Server ist das Wiederanlaufen deshalb wesentlich einfacher als bei einem (→ *Status-behafteten*) Server. Ein Status-freier File-Server etwa überlässt es den Clients, sich ihre jeweilige Position in einer sequentiellen Datei zu merken. Die Schnittstelle eines solchen File-Servers muss deshalb im Funktions-Aufruf `read (datei, dateizeiger, buffer, zahl-zu-lesender-bytes` diese Information jeweils mitsenden.

Store-and-forward:

(*Teilstrecken-Vermittlung*) Rechnernetze auf der Basis von Teilstrecken-Vermittlung speichern die zu übertragende Information in den Knoten (→ *IMP*) und geben sie von Knoten zu Knoten weiter.

Strikte Ordnung:

Eine strikte Ordnung der Ereignisse in einem verteilten System bedeutet, dass sämtliche Nachrichten in einem verteilten System bei allen Empfänger-Prozessen in derselben Reihenfolge auftreten. Dies unabhängig davon, ob Nachrichten kausal voneinander abhängig sind oder nicht (→ *Kausalität*). Protokolle welche eine strikte Ordung realisieren, fügen dem verteilten System oftmals zusätzliche Nachrichten bei und sind dann bezüglich der Anzahl Nachrichtenempfänger von überlinearer Komplexität. Verteilte Systeme welche eine strikte Ordung realisieren, skalieren in der Regel bezüglich der Anzahl Nachrichtenempfänger schlecht. Verteilte Systeme welche eine kausale Ordung realisieren skalieren in der Regel bezüglich der Anzahl Nachrichtenempfänger besser.
LITERATUR: [BJ89] [BSS91]

Strukturierte Daten:

Im Kontext (→ *vertikaler*) oder (→ *horizontaler Separation*) von Daten bedeutet Strukturierung hier das Vorliegen der Daten in der Form mehrerer Entitäten, zu denen jeweils eine Anzahl von Attributen definiert ist. Stellen Sie sich zudem die Daten in einer tabellarischen Form gespeichert vor, bei der jeder Zeile eine Entität und jeder Spalte ein Attribut entspricht.

Stub:

Bei einem (→ *Remote Procedure Call*) erscheint es dem Programmierer so, als ob auf einer entfernten Maschine eine Prozedur aufgerufen wird. Tatsächlich wird die Kontrolle lokal an eine sogenannte Stub-Prozedur übertragen, welche die Kommunikation mit der entfernten Maschine durchführt. Solche Stubs gibt es auf der Server- und der Client-Seite. Bei moderneren RPC-Systemen können sie auch aus einer formalen Beschreibung der Schnittstelle generiert werden.

Synchrones System:

Ein verteiltes System ist synchron wenn es folgende Anforderungen erfüllt:

- Es gibt eine bekannte obere Schranke für die Zeit die benötigt wird eine Nachricht zu übertragen.
- Die lokalen Uhren der am verteilten System beteiligten Komponenten weichen maximal um einen vorgegebenen Wert voneinander ab.
- Es gibt eine bekannte obere und untere Schranke für die Zeit welche ein Prozess benötigt um eine Anweisung auszuführen.

Es ist möglich in synchronen Systemen Ausfälle zuverlässig mittels timeouts festzustellen. Diverse Probleme lassen sich in Systemen mit approximativ synchronisierten Uhren mittels einfacherer Algorithmen lösen (→ *Asynchrones System*).
LITERATUR: [HT93]

TCP:

(*Transmission Control Protocol*) Ein Kommunikationsprotokoll das zuverlässige, bidirektionale, verbindungsorientierte Kommunikation zwischen zwei Prozessen auf verschiedenen Rechnern gestattet. Die TCP ProtokollSchicht basiert auf dem IP Protokoll (→ *IP*) zur Kommunikation auf der Internet Netzwerkebene. Seit seinem Einsatz durch das amerikanische Verteidigungsministerium ist TCP/IP ein wichtiger Standard für die Verbindung verteilter Systeme geworden.
LITERATUR: [Com91, CS91, CS93b, Ste90]

Thread:

In den meisten traditionellen Betriebssystemen beherbergt jeder Prozess (→ *Prozess*) einen einzigen Ausführungspfad (*thread of execution*). In vielen Situationen ist es aber vorteilhaft, mehrere Ausführungspfade pro Prozess zu haben. Dabei spricht man von *Multi-threading.* Ein solcher Ausführungspfad wird als *thread* oder als leichtgewichtiger Prozess bezeichnet. Die Threads eines Prozesses laufen in der Regel *quasi*-parallel ab. Normalerweise läuft ein Thread solange bis dieser, z.Bsp. während einer I/O Operation, blockiert wird. Wird ein Thread blockiert, so kommt sofort der nächste Thread innerhalb des Prozesses zum Zuge. Welcher der "nächste" Thread ist entscheidet ein Scheduling-Algorithmus, der vom Programmierer beeinflusst werden kann.

Der Kontext-Wechsel zwischen Threads kommt dabei wesentlich rascher Zustande als der Kontext-Wechsel zwischen Prozessen. Dies weil die mit einem Thread assoziierte Kontext-Information gering ist und in der Regel lediglich den Programm-Zähler, den Stack-Pointer und Register-Inhalte umfasst. Multi-threading erlaubt die einem Prozess zugeordnete Prozessor-Zeit wesentlich besser zu nutzen. Der Prozess kann nun solange aktiv bleiben wie ein Thread in ihm arbeiten kann und die dem ganzen Prozess zugeordnete Zeitscheibe noch nicht abgelaufen ist. Ohne Multi-threading wird ein in einer I/O Operation blockierter Prozess gleich vollständig suspendiert. Moderne Betriebssystem-Kerne wie Mach oder Amoeba unterstützen Multi-threading. Bei anderen Betriebssystemen muss Multi-threading vollständig im *user space* implementiert werden. Der Nachteil dabei ist, dass ein vom Prozess empfangenes Signal sämtliche Threads blockieren kann, falls keine besonderen Vorkehrungen getroffen werden (Abfangen und weiterleiten sämtlicher Signale, spezielle non-blocking I/O Bibliotheken etc.).

Multi-threading erlaubt die Effizienz zahlreicher verteilter Anwendungen signifikant zu steigern, Ergebnisse von asynchronen Operationen (z.B. von asynchronem RPC oder asynchronem `send()`) zu empfangen und einen reaktiven Programmierstil zu praktizieren.
LITERATUR: [Tan92, Seite 507 ff.]

Thread-Gruppen:

Wenn sehr viele Prozessgruppen (→ *Prozessgruppe*) von erheblicher Grösse notwendig sind, können Prozessgruppen zu ineffizienten, schlecht skalierbaren Lösungen führen. Dies vor allem deshalb, weil Prozesse (→ *Prozess*) in der Art und Weise wie BSD 4.3 UNIX und Derivate sie vorsehen, "schwergewichtige" Konstrukte sind. Das heisst, mit einem Prozess ist viel Kontext-Information assoziiert und ein Prozesswechsel kommt daher teuer zu stehen. Neuerdings wird der Ansatz verfolgt, Gruppen von "leichtgewichtigen" Prozessen (→ *Thread*), sprich Thread-Gruppen, zu realisieren. Das Hinzutreten zu einer Gruppe, die Verwaltung der Gruppe und die Kontext-Wechsel werden dabei erheblich effizienter. Thread-Gruppen werden beispielsweise im HORUS Toolkit realisiert [vRBC+92].

Tightly Coupled System:

Im Gegensatz zum (→ *loosely coupled system*) ein Multicomputer-System, das für mehr als einen Prozessor Zugriff auf einen gemeinsamen Speicher (shared memory) erlaubt und über ein einziges gemeinsames Betriebssystem gesteuert wird.

Timestamp:

Eine Zeit-Marke, die einer Transaktion zur zeitlichen Einordnung relativ zu anderen vergeben wird.

Token:

Als Token bezeichnet man im Compilerbau ein in einem Text auftretendes Wort, das durch eine lexikalische Spezifikation näher definiert wird. Im Kontext verteilter Systeme und Protokolle ist ein Token ein abstraktes Recht, das einem Programm oder Protokoll zur Verfügung steht, wenn es diesen Token "besitzt". Man kann sich darunter ein Märkchen vorstellen, auf dem bestimmte Daten stehen und das von rechner zu Rechner weitergereicht werden kann.

Token Access:

Eine Methode, bei welcher der Zugriff auf Daten oder Geräte durch Rechner vom Besitz eines Tokens gesteuert wird.

Token passing:

Bei diesem Verfahren lässt man auf dem Ring eines Ringnetzes einen Token zirkulieren. Will eine Station Daten senden, so wartet sie auf den Token, entfernt diesen vom Ring, sendet die Daten und schickt nach Abschluss der Sendung den Token auf dem Ring weiter.

Topologie:

Ein verteiltes System kann physikalisch durch Prozessoren und Kommunikationsverbindungen, logisch durch Prozesse und Kommunikationskanäle als ein Graph beschrieben werden. Hierbei stellen die Knoten des Graphen die Prozessoren oder Prozesse, die Kanten die Kommunikationsverbindungen oder -kanäle dar. Die durch diesen Graphen festgelegte Struktur wird als (Kommunikations)-Topologie bezeichnet.

Trader:

Der Trader ist ein Prozess oder Mechanismus, der Orts-Transparenz (→ *Transparenz*) gewährleistet, in dem er Dienstnamen auf physikalische Adressen abbildet.

Transaktion (Transaction):

Veränderung eines Datenbestandes, welche die *ACID*–Eigenschaften besitzt: Sie ist *a*tomar, wird also entweder als Ganzes ausgeführt oder, etwa im Fall von Fehlern oder beim Auftreten von Ausnahmebedingungen, überhaupt nicht. Sie erhält die (→ *Konsistenz*) des Datenbestandes. Sie wirkt auf den Datenbestand *i*soliert, als ob nur die betreffende Transaktion selber auf den Datenbestand zugreifen würde. Änderungen des Datenbestandes werden für gleichzeitig ablaufende Transaktionen erst sichtbar, wenn die betreffende Transaktion beendet ist. Sie bewirkt *d*auerhafte Änderungen des Datenbestandes, die nach einem erfolgreichen Abschluß der Transaktion durch Systemfehler nicht mehr gefährdet werden können.
LITERATUR: [BHG87]

Transaktion vs. Kausale Kommunikation:

Als allgemeines Programmierparadigma für verteilte, parallele Systeme werden oft (verschachtelte) Transaktionen vorgeschlagen. Wir sind der Auffassung, dass Transaktionen als generelles Programmierparadigma für verteilte, parallele Anwendungen schlecht geeignet ist. Dafür sprechen folgende Gründe: Bekanntlich ist mittels herkömmlichen Transaktionsmechanismen in vielen Fällen kein hoher Grad an Parallelität erreichbar. Um die Parallelität zu steigern wird das *nested transactions* (verschachtelte Transaktionen) Modell vorgeschlagen [Wei89]. Das Realisieren von Systemen für verschachtelte Transaktionen ist aber mit hohem Aufwand verbunden (das heisst teuer), und eine feinere Synchronisation zwischen verteilten Threads, wie dies beispielsweise bei diversen CIM Anwendungen notwendig ist, ist nachwievor nicht möglich. Das Modell der verschachtelten Transaktionen geht zudem von einer hierarchischen Prozesstruktur aus, in der eine Vater-Transaktion diverse parallele Sub-Transaktionen generiert. Gewisse Anwendungen, wie z.B. ausfalltolerante Filesysteme welche mit Objekt-Replikation arbeiten, benötigen andere Prozessstrukturen, wie beispielsweise überlappende Gruppen von gleichwertigen Prozessen. Anwendungen wie verteilte Filesysteme (→ *Verteiltes Filesystem*) könnte meist auch mit einem permissiveren Synchronisationsmodell als Serialisierbarkeit (→ *Serialisierbarkeit*) auskommen und dadurch eine wesentlich höhere Leistung erzielen. Beispielsweise arbeitet das ANDREW Filesystem [Sat93] indem gewisse Inkonsistenzen zwischen File-Replikaten über eine kurze Zeit zugelassen werden und Serialisierbarkeit oft verletzt wird. Serialisierbarkeit der Filesystem-Operationen würde die Skalierbarkeit von ANDREW stark und unnötig einschränken.

Andererseits sind Transaktionen aber ein mächtiges und wertvolles Instrument um datenorientierte Anwendungen zu realisieren, wo die oben geforderte feingranulare Synchronisation meist nicht erforderlich ist. Neuerdings wurden die Modelle der Linearisierbarkeit (→ *Linearisierbarkeit*) und der Kausalität (→ *Kausalität*) vorgeschlagen, um einen allgemeiner einsetzbaren Synchronisationsmechanismus zu schaffen. Serialisierbarkeit und Transaktionen könnten dann darauf aufgebaut werden, womit die Unterstützung für rein datenorientierte Anwendungen ebenfalls gegeben wäre. Eine Möglichkeit wie man Linearisierbarkeit, Kausalität und Serialisierbarkeit integrieren kann wird in [Bir93] vorgestellt.

Transparenz:

Mit *Transparenz (transparency)* als Sammelbegriff werden Techniken bezeichnet, welche die (→ *Separation*) einer Anwendung in einzelne Komponenten und deren Verteilung auf unterschiedliche Rechner vor dem Benutzer und dem Anwendungsprogrammierer weitgehend unsichtbar gemacht werden (→ *Separation*).

- *Konfigurations-Transparenz* gestattet das Umkonfigurieren der physikalischen Komponenten eines verteilten Systems, ohne dass dies Auswirkungen auf die Funktionsweise der Anwendung hat.
- *Migrations-Transparenz* gestattet das Umverteilen logischer Komponenten einer Anwendung auf andere physikalische Komponenten eines verteilten Systems, ohne dass dies Auswirkungen auf die Funktionsweise der Anwendung hat. (→ *Separation*).
- *Orts-Transparenz* gestattet ein Ansprechen von Daten, Programmen und anderen logischen Komponenten einer Anwendung, ohne die physikalische Komponente kennen zu müssen, auf welche diese logische Komponente verteilt wurde. (→ *Trader*), (→ *Port*).
- *Parallelitäts-Transparenz* gestattet mehreren Benutzern und Programmen das gleichzeitige Ansprechen gemeinsam genutzter Datenbestände, ohne dass sich der Benutzer oder der Programmierer um mögliche Interferenzen aus dieser gemeinsamen Nutzung kümmern muss.
- *Referentielle Transparenz* Im Gegensatz zu vielen Formen der Transparenz, die im Bereich verteilter Systeme wichtig sind, stammt die referentielle Transparenz aus dem Gebiet der funktionalen Programmierung. Sie besagt das in jeder eigentlichen funktionalen Sprache gültige Prinzip, dass syntaktisch gleiche Ausdrücke auch semantisch gleich sind. So gilt in einer funktionalen Sprache für eine Funktion $f(x)$ stets die auch in der Mathematik gültige Gleichung $f(4) + f(4) = 2 * f(4)$. In einer imperativen Programmiersprache kann die Funktion f etwa auf globale Variable zugreifen. In diesem Fall braucht der zweite Aufruf von f keineswegs dasselbe Resultat liefern wie der erste, auch bei gleichen Argumenten.
- *Replikations-Transparenz* Technik, bei der Daten in mehreren physikalischen Kopien abgespeichtert werden, ohne dass sich die zugreifenden Programme um die gegenseitige Konsistenz der einzelnen Kopien kümmern müssen. Replikations-Transparenz lässt mehrere physikalische Kopien eines Datums nach aussen wie eine einzige Kopie aussehen.
- *Zugriffs-Transparenz* gestattet auf lokale und entfernte Dateien und Objekte mit denselben Mechanismen zuzugreifen.

LITERATUR: [CD88]

UDP:

(*User Datagram Protocol*) Dieses Kommunikationsprotokoll des TCP/IP Standards erlaubt die Kommunikation von *datagrams* (→ *datagrams*) zwischen Prozessen auf verschiedenen Rechnern mittels des IP Protokolls (→ *IP*). UDP erlaubt im Gegensatz zu TCP (→ *TCP*) selektive Kommunikation (→ *Broadcast, Multicast*), gewährleistet aber nicht den Empfang eines gesendeten Packets.
LITERATUR: [Com91, Ste90]

Ultracomputer:

Supercomputer mit einer Spitzengeschwindigkeit von mehr als einem *Teraflop* (10^{12} Gleitkommaoperationen pro Sekunde). Ultracomputer der ersten Generation bestehen aus (evtl. tausenden von) vernetzten Computern (→ *Multicomputer*), die allerdings nur von grobgranularen (→ *Granularity*) Anwendungen effizient genutzt werden können.
LITERATUR: [Bel92]

Verfügbarkeit:

Der Benutzer eines Systems erwartet sich von diesem während eines gewissen Zeitraumes ein fehlerfreies Arbeiten. Dieser Zeitraum kann die jeweilige Arbeitszeit des Benutzers, aber auch das ganze Kalenderjah sein. Jener Anteil der Zeit, zu dem das System tatsächlich voll funktionsfähig ist, heisst die Verfügbarkeit des Systems.

Verteiltes Objekt:

Was ein verteiltes Objekt genau ist, darüber ist man sich heute noch nicht ganz einig. Grundsätzlich geht es dabei um objektorientierte Programmierung (→ *Objektorientierte Programmierung*) verteilter Systeme. Uneinig ist man sich, ob ein verteiltes Objekt vollständig in *einem* Adressraum instanziert ist oder ob sich ein verteiltes Objekt über *mehrere* Adressräume erstreckt.
LITERATUR: [Bey93] [Maf93e] [NWM93]

Verteiltes Filesystem:

Ein verteiltes Filesystem verwaltet Files welche auf diversen Rechnern abgelegt sind einheitlich. Ein Benutzer eines verteilten Filesystemes kann transparent auf die Daten eines anderen Benutzers zugreifen, welche z.Bsp. auf einer weit entfernten Workstation abgelegt sein können. Anforderungen an ein verteiltes Filesystem sind Transparenz, Verfügbarkeit, Sicherheit, Leistung und Skalierbarkeit. Beispiele moderner verteilter Filesysteme sind das ANDREW Filesystem, CODA und DECEIT.
LITERATUR: [Sat93] [Sie92] [Bor92] [S+87] [KS92]

Verteiltes System:

Leslie Lamport soll ein verteiltes System einst wie folgt definiert haben: "A distributed system is one that stops you from getting any work done when a machine you've never heard of crashes". Dies mag darauf hindeuten, dass heute allerhand unter dem Etikett "Verteiltes System" angepriesen wird. Tanenbaum [Tan92, Seite 382] definiert ein verteiltes System wie folgt: "A distributed system is one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer". Damit wird vor allem die Transparenz unterstrichen. Mullender [Mul89, Seite 5] betont bei seiner Definition vor allem die Ausfalltoleranz: "A distributed operating system must not have any single point of failure — no single part failing should bring the whole system down". Wichtige Anforderungen an ein verteiltes System sind sicherlich Transparenz, Flexibilität, Erweiterbarkeit, Zuverlässigkeit (Ausfalltoleranz), Leistung und Skalierbarkeit.

Vertikale Separation:

Bei (→ *strukturierten Daten*) spricht man von vertikaler Separation, wenn die Daten nach Attributen aufgeteilt werden: In jeder Partition der Daten befinden sich alle Entitäten, jedoch nur jeweils bestimmte Attribute von ihnen. Wenn Sie sich die Daten in einer tabellarischen Form gespeichert vorstellen, bei der jede Zeile eine Entität darstellt und jede Spalte ein Attribut, so bedeutet vertikale Separation gerade das vertikale Durchschneiden dieser Tabelle.
LITERATUR:

Virtual Circuit:

Verbindungsart in einem Netz mit Datenpaketvermittlung. Für die Übertragung aller Pakete einer Nachricht wird der gleiche Kommunikationsweg fest gewählt und die Reihenfolge der Pakete bleibt erhalten.

Virtual Shared Memory:

Software-Ebene, die zur einfacheren Programmierung von Multicomputern (→ *Multicomputer*) mit verteiltem Adressraum einen einzigen Adressraum zur Verfügung stellt.

Virtuelle Synchronie:

(→ *Kausalität*).

Wide Address Space:

Die meisten heutigen Architekturen bieten einen 32 Bit Addressbereich an, mit dem 4 Gigabytes Hauptspeicher addressiert werden können. Prozessoren wie DEC Alpha, Intel 80486 oder MIPS R-4000 sehen bereits 64 Bit Address Bereich vor, mit dem über 16 Millionen Terabytes addressiert werden können!

Anders ausgedrückt, falls eine Anwendung 500 Megabytes Hauptspeicher pro Sekunde alloziert ohne diesen wieder freizugeben, kann diese Applikation über tausend Jahre lang laufen bevor die zugeteilten virtuellen Adressen ausgehen.

Mittels 64 Bit Address-Räumen könnten Konzepte wie "global virtual shared memory" (→ *Virtual Shared Memory*) realisiert werden. Dabei erstreckt sich ein Adressraum über sämtliche Prozesse eines (homogenen) verteilten Systemes (→ *Homogeneity*), und die gesamte Interprozess-Kommunikation liesse sich über dieses "global virtual shared memory" abwickeln.

Die Speicheradresse eines Objektes könnte dann als "unique identifier" für das Objekt herangezogen werden. Konzepte wie globales Naming (→ *Naming*) und Persistenz (→ *Persistenz*) sind damit einfacher zu realisieren.
LITERATUR: [BMvdV93]

Workstation:

Unter einer Workstation versteht man i.d.R. einen Arbeitsplatzcomputer, welcher mit einem hochauflösenden Bildschirm, einem Netzwerk-Anschluss, einem Multi-tasking Betriebssystem (→ *Multi-tasking Betriebssystem*), einer Fenster-Benutzeroberfläche sowie mit Anschlussmöglichkeiten für diverse Peripheriegeräte (Harddisks, Lautsprecher, Mikrophon etc.) ausgestattet ist. Beispiele moderner Workstations sind die SparcStation 10 von Sun Microsystems, HP 9000 von Hewlett Packard, Silicon Graphics Workstations etc.

Zuverlässigkeit:

(*Reliability*). Die Wahrscheinlichkeit $R(t)$, dass ein System seine Dienste nach Ablauf des Zeitintervalls t immer noch korrekt vollbringt, wenn das System zu $t = 0$ korrekt funktioniert hat.

References

[ATK92] A.L. Ananda, B.H. Tay, and E.K. Koh. A Survey of Asynchronous Remote Procedure Calls. *ACM Operating Systems Review*, 26(2), April 1992.

[B+90] K. P. Birman et al. *The ISIS System Manual, Version 2.1.* The ISIS Project, September 1990.

[BCH+93] J.-P. Baud, F. Cane, F. Hemmer, E. Jagel, A. Kumar, G. Lee, L. Robertson, B. Segal, and A. Trannoy. Shift User Guide and Reference Manual, Version 1.3. Technical report, CERN, 1993.

[Bel92] Gordon Bell. Ultracomputers — A Teraflop Before Its Time. *Communications of the ACM*, 35(8), August 1992.

[Bey93] Torsten Beyer. Objektbörse. *IX Multiuser Multitasking Magazin*, Februar 1993.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[Bir91] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report 91-1216, Cornell University, Dept. of Computer Science, July 1991. To appear in Communications of the ACM, Dec. 1993.

[Bir93] Kenneth P. Birman. Integrating Runtime Consistency Models for Distributed Computing. Technical Report 91-1240, Cornell University, Dept. of Computer Science, July 1993. To appear in Journal of Parallel and Distributed Computing.

[BJ87] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *ACM Operating Systems Review*, 21(5), November 1987.

[BJ89] K. P. Birman and T. A. Joseph. Exploiting Replication in Distributed Systems. In Sape Mullender, editor, *Distributed Systems.* ACM Press, 1989.

[BMvdV93] Alberto Bartoli, Sape J. Mullender, and Martijan van der Valk. Wide-Address Spaces – Exploiting the Design Space. *ACM Operating Systems Review*, 27(1), January 1993.

[Bor92] Uwe M. Borghoff. *Catalogue of distributed file/operating systems.* Springer Verlag, 1992.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, 1989.

[Cap90] Clemens H. Cap. Distributed Systems with Data Replication – a Non-Technival Survey. Technical Report IFI-90.11, CS Dept. University of Zurich, Switzerland, December 1990.

[CD88] G. F. Coulouris and J. Dollimore. *Distributed Systems.* Addison Wesley, 1988. Pages 301–312.

[CGKW93] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive Parallelism with Piranha. Technical report, Yale University, 1993.

[Com91] D. E. Comer. *Internetworking with TCP/IP — Vol. 1: Principles, Protocols, and Architecture.* Prentice-Hall, 1991.

[CS91] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP — Vol. 2: Design, Implementation, and Internals.* Prentice-Hall, 1991.

[CS93a] C. H. Cap and V. Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, 19(11), 1993.

[CS93b] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP — Vol. 3: Client-Server Programming and Applications.* Prentice-Hall, 1993.

[CSG+93] M. Ciment, W. Scherlis, S. Griffin, C. Brownstein, et al. *Grand Chellanges 1993: High Performance Computing and Communications.* National Science Foundation, 1993.

[Dav89a] S. B. Davidson. Replicated Data and Partition Failures. In Sape Mullender, editor, *Distributed Systems*, chapter 13.1-13.3, pages 265–281. ACM Press, 1989.

[Dav89b] S. B. Davidson. Replicated Data and Partition Failures. In Sape Mullender, editor, *Distributed Systems*, chapter 13., pages 265–292. ACM Press, 1989.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3), September 1985.

[Dig91] Digital Equipment Corp., Hewlett-Packard Co., HyperDesk Corp., NCR Corp., Object Design Inc., SunSoft Inc. *The Common Object Request Broker: Architecture and Specification*, December 1991. Revision 1.1, OMG Document Number 91.12.1.

[DR93] Prasun Dewan and John Riedl. Toward Computer-Supported Concurrent Software Engineering. *IEEE Computer*, January 1993.

[EGG91] C. Ellis, S. Gibbs, and Rein G. Groupware – Some Issues and Experiences. *Communications of the ACM*, 34(10), 1991.

[EJZ92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[EMS91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector. *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., 1991.

[Fly66] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12), December 1966.

[FS93] R. F. Freund and H. J. Siegel. Heterogeneous Processing. *IEEE Computer*, 26(6), June 1993.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *OOPSLA Conference Proceedings*, Kaiserslautern, Germany, July 1993.

[Hay88] John P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, second edition, 1988.

[Her86] Maurice. P. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[HH89] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz.* Springer-Verlag, 1989.

[HPF93] High Performance Fortran Language Specification. Technical report, High Performance Fortran Forum, May 1993.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison Wesley, second edition, 1993.

[HW90] Maurice. P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[Isi93] Isis Distributed Systems Inc., Ithaca, NY. A Response to the ORB 2.0 RFI, April 93.

[KPS93] A. A. Khokhar, V. K. Prasanna, and M. E. Shaaban. Heterogeneous Computing: Challenges and Opportunities. *IEEE Computer*, 26(6), June 1993.

[KS88] C. P. Kruskal and C. H. Smith. On the Notation of Granularity. *The Journal of Supercomputing*, 1, 1988.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the CODA File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An Efficient Reliable Broadcast Protocol. *ACM Operating Systems Review*, 23(4), October 1989.

[L+89] Samuel J. Leffler et al. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison Wesley, 1989.

[Lam78] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — A Hunter of Idle Workstations. In *8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[Maf93a] Silvano Maffeis. Cache Management Algorithms for Flexible Filesystems. Technical Report 93.03, CS Dept. University of Zurich, 1993. To appear in ACM SIGMETRICS Performance Evaluation Review, Vol. 21,3.

[Maf93b] Silvano Maffeis. Distributed Programming using Object-Groups. Technical Report 93.38, CS Dept. University of Zurich, September 1993. To appear in Lecture Notes in Computer Science, Springer-Verlag.

[Maf93c] Silvano Maffeis. Electra – Making Distributed Programs Object-Oriented. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV (SEDMS IV)*, San Diego, CA, 1993. USENIX.

[Maf93d] Silvano Maffeis. File Access Patterns in Public FTP Archives and an Index for Locality of Reference. *ACM SIGMETRICS Performance Evaluation Review*, 20(3), March 1993.

[Maf93e] Silvano Maffeis. Remote Method Calling and Object Group Communication. ECOOP Workshop on Object-based Distributed Programming, Kaiserslautern, Germany, July 1993.

[Mey88] Bertrand Meyer. *Object Oriented Software Constrution.* Prentice-Hall, 1988.

[Moc87] P. V. Mockapetris. Domain Names – Concepts and Facilities. RFC 1034, Network Working Group, November 1987.

[MPI93] Document for a Standard Message-Passing Interface. Technical report, Message Passinge Interface Forum, August 1993.

[Mul89] Sape Mullender. *Distributed Systems.* ACM Press, 1989.

[Mul93] Sape J. Mullender. Kernel Support for Distributed Systems. In Sape Mullender, editor, *Distributed Systems*, chapter 15. Addison Wesley, second edition, 1993.

[Nee93] Roger M. Needham. Names. In Sape Mullender, editor, *Distributed Systems*, chapter 12. Addison Wesley, second edition, 1993.

[Nel81] B.J. Nelson. *Remote Procedure Call.* PhD thesis, Carnegie Mellon University, 1981. CMU-CS-81-119.

[NWM93] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, 26(6), June 1993.

[PW88] Lewis J. Pinson and Richard S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk.* Addison Wesley, 1988.

[RB91] Tom Rodden and Gordon Blair. CSCW and Distributed Systems: Problem and Control. In *Proceedings of the Second European Conference on Computer Supported Cooperative Work*, Amsterdam, The Netherlands, September 1991.

[Ric85] Lutz Richter. *Betriebssysteme*. Teubner Verlag, 1985.

[Ric92] Aleta Marie Ricciardi. *The Group Membership Problem in Asynchronous Systems*. PhD thesis, Cornell University, Dept. of Computer Science, Ithaca, New York, November 1992. No. 92-1313.

[S+87] M. Satyanarayanan et al. Scale and Performance in a Distributed File System. *ACM Operating Systems Review*, 21(5), September 1987.

[S+88] M. Stefik et al. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. In I. Greif, editor, *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers, 1988.

[S+89] M. Shapiro et al. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems*, 2(4), December 1989.

[Sat93] M. Satyanarayanan. Distributed Filesystems. In Sape Mullender, editor, *Distributed Systems*, chapter 14. Addison Wesley, second edition, 1993.

[SC92] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6), June 1992.

[Sch93] Fred B. Schneider. What Good are Models and What Models are Good? In Sape Mullender, editor, *Distributed Systems*, chapter 2. Addison Wesley, second edition, 1993.

[SDP] Santosh K. Shrivastava, Graeme N. Dixon, and Graham D. Parrington. An Overview of the Arjuna Distributed Programming System. Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK.

[Sei90] C. L. Seitz. Concurrent Architectures. In R. Suaya and G. Birtwistle, editors, *VLSI and Parallel Computation*. Morgan Kaufmann Publishers, 1990.

[Sie92] Alexander Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Faculty of the Graduate School, Cornell University, Dept. of Computer Science, May 1992.

[SP91] Abraham Silberschatz and James Peterson. *Operating System Concepts*. Addison Wesley, third edition, 1991.

[Ste90] W. R. Stevens. *UNIX Network Programming*. Prentice-Hall, 1990.

[Tai89] Carl D. Tait. Techniques for Building Highly Available Distributed File Systems. Technical Report CUCS-497-89, Columbia University, New York, March 1989.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[The86] M. M. Theimer. Preemptable Remote Execution Facilities for Loosly-Coupled Distributed Systems. Technical report, Computer Systems Lab., Stanford University, June 1986.

[TL88] M. M. Theimer and K. A. Lantz. Finding Idle Machines in a Workstation-based Distributed System. In *8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[Tur93] L. H. Turcotte. A Survey of Software Environments for Exploiting Networked Computing Resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State, June 1993.

[TvR88] Andrew S. Tanenbaum and Robbert van Renesse. A Critique of the Remote Procedure Call Paradigm. In R. Speth, editor, *Research into Networks and Distributed Applications*. Elsevier Science Publisher B.V., 1988.

[vR93] Robbert van Renesse. Causal Controversy at Le Mont St.-Michel. *ACM Operating Systems Review*, 27(2), April 1993.

[vRBC+92] Robbert van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures*, Seattle, Washington, April 1992.

[Weg93] Peter Wegner. Towards Component-Based Software Technology. Technical Report CS-93-11, Brown University, Dept. of Computer Science, July 1993.

[Wei89] W. E. Weihl. Theory of Nested Transactions. In Sape Mullender, editor, *Distributed Systems*. ACM Press, 1989.

[WWC92] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward Megaprogramming. *Communications of the ACM*, 35(11), November 1992.