Prof. Dr. Jörg Haake, apl. Prof. Dr. Christian Icking, Dr. Britta Landgraf, Dr. Till Schümmer

Modul 63211

Verteilte Systeme

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhalt

Einleitende Bemerkungen

Liebe Fernstudentin, lieber Fernstudent,

herzlich willkommen beim Kurs *1678 Verteilte Systeme*! Im diesem Kurs sollen Sie kennenlernen, wie verteilte Systeme auf der Basis moderner Betriebssysteme und Rechnernetze entworfen und realisiert werden können.

Verteiltes System

Ein *Verteiltes System* besteht aus mehreren Komponenten, die auf vernetzten Rechnern angesiedelt sind und ihre Aktionen durch den Austausch von Nachrichten über Kommunikationskanäle koordinieren. Benutzer des Systems sollen aber nicht merken, dass das System nicht vollständig auf ihrem lokalen Rechner läuft. Im Vergleich zu einzelnen Rechensystemen treten bei verteilten Systemen ganz neue Probleme auf: Daten, welche auf unterschiedlichen Rechensystemen auch unterschiedlich dargestellt werden, sollen ausgetauscht werden, Prozesse müssen synchronisiert werden, verteilte persistente Datenbestände sollen konsistent gehalten werden. In diesem Kurs werden wir uns in erster Linie mit der Lösung solcher Probleme beschäftigen.

Buch besorgen!

Dieser Text ist als Begleittext zu dem Buch

> *Distributed Systems: Principles and Paradigms*
> Andrew S. Tanenbaum, Maarten van Steen
> Prentice Hall, second edition, 2006, als Paperback 2008 und 2013

verfasst worden, das auch in deutscher Übersetzung erhältlich ist:

> *Verteilte Systeme: Prinzipien und Paradigmen*
> Andrew S. Tanenbaum, Maarten van Steen
> Pearson Studium, zweite Ausgabe, 2007

zweite Auflage

Im Folgenden wird vorausgesetzt, dass Sie über dieses Buch verfügen und jeweils die hier angegebenen Abschnitte lesen. Bitte verwenden Sie nur die zweite Auflage (2006 – 2008, auch 2013) und keine ältere.

englisch(!) oder deutsch(?)

Wenn Sie englische Texte gut lesen können, macht Ihnen die Originalausgabe sicher mehr Spaß, im Prinzip sind aber beide Ausgaben benutzbar, in beiden wird dieselbe Nummerierung für Abschnitte, Abbildungen usw. verwendet, die Seitenzahlen sind natürlich verschieden.

Ergänzungen und Fehlerkorrekturen

Je länger wir uns als Kursbetreuer mit dem Buch beschäftigen, desto notwendiger empfanden wir *Ergänzungen und Fehlerkorrekturen*, deshalb ist dieser Begleittext im Laufe der Zeit von einer reinen Leseliste zu einem eigenen Lerntext geworden, der einige Buchinhalte hoffentlich verständlicher macht. Deshalb achten Sie bitte genau auf die Lesehinweise zu jeder Kurseinheit sowie auch auf die (leider) immer länger gewordene Fehlerliste am Ende.

Lesehinweise

Die sieben Kurseinheiten sind aus den folgenden Abschnitten in verschiedenen Kapiteln des Buchs zusammengestellt:

Kurseinheit	Abschnitte im Buch	Inhalt
1	1.1 bis 1.3.2	Grundlagen
	2.1 bis 2.2	Architekturen
2	3.1 bis 3.4.2	Prozesse
	4.1 bis 4.2.3, 4.3 bis 4.3.2	Kommunikation
3	5.1 bis 5.3	Namen
	6.1 bis 6.3	Synchronisation
4	7	Konsistenz und Replikation
5	8.1 bis 8.5	Fehlertoleranz
6	9	Sicherheit
7	11.1 bis 11.4.1	Verteilte Dateisysteme
	12.1 bis 12.4	Web-basierte Systeme

Dazu kommt in Kurseinheit 7 noch unser Abschnitt 7.2 über Versionsverwaltung in diesem Text.

Zusammenfassungen

Außerdem zum Lesen empfohlen sind alle *Zusammenfassungen* der angesprochenen Buchkapitel jeweils am Ende.

Fehlerlisten

Wie alle technischen Texte ist auch der Basistext nicht fehlerfrei, und es handelt sich nicht nur um Übersetzungsfehler in der deutschen Version, sondern auch um sachliche Fehler (auf englisch und auf deutsch). Deshalb haben wir, wie schon erwähnt, diesem Text noch den Abschnitt 8 hinzugefügt, in dem wir alle Fehler auflisten, die uns in den relevanten Abschnitten bisher aufgefallen sind.

Zitate

Zitate werden in unserem Text genauso angegeben wie im Buch, wenn die Referenz bereits im Literaturverzeichnis des Buches enthalten ist, z. B. Tanenbaum (2003), ansonsten als vollständige Quellenangabe in einer Fußnote.

Einsendeaufgaben

Zu allen sieben Kurseinheiten erhalten Sie *Einsendeaufgaben*, die zur intensiveren Beschäftigung mit dem Lernstoff anregen sollen. Wenn Sie diese Aufgaben bearbeiten und einschicken, bekommen Sie sie korrigiert zurück. Alle Kursbeleger erhalten dann auch die ausgearbeiteten Lösungshinweise. Weitere Aufgaben finden Sie im Buch.

Wir wünschen Ihnen bei der Bearbeitung des Kurses Spaß und viel Erfolg bei den Prüfungen!

Hagen, im September 2015
Ihre Kursautoren und -betreuer

1 Grundlagen und Architekturen

Verteilte Systeme
Transparenz
Offenheit
Skalierbarkeit
Sicherheit

Zu Beginn interessieren wir uns dafür, was *Verteilte Systeme* sind, welche Ziele damit angestrebt werden und welche Varianten es gibt. *Transparenz*, *Offenheit*, *Skalierbarkeit* und *Sicherheit* sind Eigenschaften, die von Verteilten Systemen erwartet werden, und gleichzeitig Herausforderungen, die nicht immer leicht zu erfüllen sind. Die *Middleware-Schicht* spielt eine zentrale Rolle für die Realisierung dieser Eigenschaften.

Architektur

Danach beschäftigen wir mit der *Architektur* von verteilten Systemen, d. h. wie ein solches System organisiert wird, aus welchen Komponenten es besteht und welche Beziehungen sie zueinander haben sowie wie sie miteinander interagieren. Vier Architekturmodelle werden in Abschnitt 2.1 im Buch vorgestellt, z. B. das zentrale *Client/Server-Modell* und das dezentrale *Peer-to-Peer-System*.

Client/Server-Modell
Peer-to-Peer-System

1.1 Lesehinweise

Bitte lesen Sie die Abschnitte 1.1 bis 1.3.2 und 2.1 bis 2.2 im Buch.

1.2 Lernziele

Nach der Bearbeitung der ersten Kurseinheit sollten Sie z. B.

- definieren können, was überhaupt ein verteiltes System ist,
- darstellen können, welche Rolle die Middleware-Schicht bei verteilten Systemen spielt,
- benennen können, welche Ziele ein verteiltes System verfolgen kann,
- erklären können, was Transparenz, Offenheit und Skalierbarkeit bedeuten,
- verstehen, welcher Unterschied zwischen verteilten Rechensystemen und verteilten Informationssystemen besteht,
- erklären können, was das ACID-Prinzip ist,
- verstehen können, wie Architekturen von verteilten Systemen charakterisiert werden und welche Architekturen es gibt,
- das Client/Server-Modell und den Unterschied zu Peer-to-Peer-Systeme beschreiben können,
- verstehen, warum man ein Overlay für ein Peer-to-Peer-System braucht,
- erklären können, wie die Peer-to-Peer-Systeme Chord, CAN und Superpeers organisiert sind.

2 Prozesse und Kommunikation

2.1 Lesehinweise

Bitte lesen Sie die Abschnitte 3.1 bis 3.4.2, 4.1 bis 4.2.3 und 4.3 bis 4.3.2 im Buch.

2.2 Prozesse, Threads, Clients und Server

Verteilte Systeme benötigen die Unterstützung von Betriebssystemen durch Verwendung von *Prozessen* und *Threads*, die wir in der zweiten Kurseinheit zuerst beschäftigen. Aus den Kursen *Betriebssysteme und Rechnernetze* oder *Betriebssysteme* kennen Sie möglicherweise schon Prozesse und Threads und wir werden lernen, wie nützlich Threads für verteilte Anwendungen im Client/Server-Modell sind. Wir betrachten noch, was der *Thin-Client-Ansatz* ist, wie Clients Transparenz schaffen und wie Server im *Cluster* oder *verteilt* erreicht werden können, dafür spielt das *Domain Name System* mit den *URLs* nicht nur für die Ortstransparenz sondern auch für die Migrationstransparenz eine wichtige Rolle.

Prozesse
Threads
DNS
URL

Eine wichtige Eigenschaft bei verteilten Systemen ist die *Migrationstransparenz*. Ein verteiltes System wird normalerweise auf einer Menge von heterogenen Rechnersystemen eingerichtet. Da jedes ein eigenes Betriebssystem hat und man nicht voraussetzen kann, dass jedes beliebige System die *Migration* von Code unterstützt, brauchen wir hierfür einen anderen Mechanismus. Ein Lösungsansatz dafür ist die *virtuelle Maschine*. Beispielsweise generiert der Java-Compiler einen ausführbaren Code für eine *Java-VM* statt für eine bestimmte Hardware. Die virtuelle Maschine ermöglicht nicht nur die Migration von Code und Prozessen, sondern auch die Migration einer ganzen Umgebung, in der Prozesse laufen können, wie z. B. eines Betriebssystems.

Migrationstransparenz
Migration
virtuelle Maschine
Java-VM

2.3 Kommunikation, Protokolle und Nachrichten

Ein weiteres wichtiges Thema im Kontext der verteilten Systeme ist die Kommunikation zwischen Prozessen. Prozesse auf unterschiedlichen Rechnern in verteilten Systemen kommunizieren miteinander durch das Senden von Nachrichten, dafür müssen sie Regeln einhalten, z. B. Formate, Bedeutungen der ausgetauschten Nachrichten und die erforderlichen Aktionen zur Übermittlung. Diese Regeln nennt man *Protokolle*. Die Kommunikation wird in mehrere Schichten aufgeteilt, und jede Schicht hat ihre Aufgaben und ihre eigenen Protokolle. Schließlich wollen wir noch wissen, welche *Kommunikationsmodelle* es gibt und wie sie funktionieren.

Protokolle

Mittels *RPC* soll es Anwendungen ermöglicht werden, für Zugriffe auf lokale und entfernte Ressourcen dieselben Operationen zu verwenden. Also ist RPC ein Mechanismus für die Realisierung der *Zugriffstransparenz*. Für *Marshaling* und *Unmarshaling* wird die *Interface Definition Language* benötigt, die wie ein Übersetzer zwischen Client- und Server-Systemen agiert.

RPC
Zugriffstransparenz
Marshaling
Unmarshaling
IDL

2.4 Lernziele

Nach der Bearbeitung dieser Kurseinheit sollten Sie z. B.

- definieren können, was Prozesse und Threads und ihre Unterschiede sind,
- beschreiben können, wie Threads implementiert werden können und welche Vor- und Nachteile die Implementierungen haben,
- Beispiele angeben können, wie Threads im Client/Server-Modell verwendet werden,
- erklären können, was der Thin-Client-Ansatz ist und wie ein Server erreicht wird,
- erzählen können, was ein Server-Cluster ist, welche Unterschiede zwischen *transport-layer switch* (deutsch: Schalter auf der Transportschicht) und *content-aware request distribution* (deutsch: inhaltsbewusste Anforderungsverteilung) bestehen,
- verstehen können, mit welchen Mechanismen *MIPv6* das transparente Routing von Paketen zu einem *mobilen Knoten* ermöglicht,
- darstellen können, wie verteilte Server arbeiten,
- Protokolle im Allgemeinen, das Schichtenprinzip, das OSI-Schichtenmodell, die Interfaces und die Aufgaben jeder Schicht erklären können,
- beschreiben können, wie ein RPC funktioniert und welche Probleme das RPC mit der Übergabe der Parameter hat,
- den Unterschied zwischen traditionellem und asynchronem RPC darstellen können,
- nachrichtenorientierte transiente von persistenter Kommunikation unterscheiden können.

3 Namen und Synchronisation

3.1 Lesehinweise

Bitte lesen Sie die Abschnitte 5.1 bis 5.3 und 6.1 bis 6.3 im Buch und jeweils unsere dazu passenden Ergänzungen hier. Bitte beachten Sie insbesondere unsere Präzisierungen in Abschnitt 3.3 zur Lamport-Uhr (6.2.1) inklusive der happens-before-Relation sowie unsere Richtigstellungen zur Vektoruhr (6.2.2).

3.2 Namen und Adressen

Name

Für einen Menschen ist der *Name* eines Hosts wie `www.fernuni-hagen.de` normalerweise erheblich leichter zu merken als eine IP-Adresse wie 132.176.114.181. Für einen Computer dagegen ist die Verwendung solch eine IP-Adresse, gespeichert in 4 Byte (IPv4) bzw. in 16 Byte (IPv6), aus verschiedenen Gründen deutlich sinnvoller als der Name. Um diese Lücke zwischen Mensch und Maschine für die Identifikation derselben Ressource zu schließen, benötigen wir ein *Namenssystem* wie z. B. das *DNS*, das dem Benutzer die Verwendung von sinnvollen Namen erlaubt und diese vor der Kommunikation über die festgelegten Protokolle in die entsprechenden IP-Adressen übersetzt, und umgekehrt. Darüber hinaus hat das DNS noch weitere nützliche Funktionen, z. B. für die effiziente Nutzung von Ressourcen oder die Lastverteilung.

DNS

Adresse
Beispiel
Dateisystem
inode
inode-Nummer

Das Namenssystem muss einer gemeinsam benutzten Ressource einen *Namen* und für den Zugriff eine maschinengeeignete *Adresse* geben. Ein gutes Beispiel für ein Namenssystem ist schon ein Dateisystem, wie wir es aus dem Bereich der Betriebssysteme sicherlich kennen, z. B. das inode-Dateisystem: jede Datei ist unter mindestens einem Namen erreichbar (hard links sind mehrere Namen für dieselbe Datei) und besitzt einen *inode* mit einer eindeutigen *Nummer*. Benutzer kennen und benutzen ausschließlich die Dateinamen. Nur das Betriebssystem kennt den inode einer Datei, darin werden Informationen wie z. B. die Blockadressen auf der Festplatte für die Dateiinhalte gespeichert. Die inode-Nummer ist die eigentliche Adresse für einen Zugriff auf eine Datei.

Namensauflösung

Ein Namenssystem benötigt zunächst eine syntaktische Definition, um gültige von nicht gültigen Namen zu unterscheiden, die Namen können z. B. hierarchisch aufgebaut werden. Bevor ein Zugriff auf eine Ressource stattfindet, muss der Name auf eine Adresse abgebildet werden. Diese Abbildung nennen wir die *Namensauflösung*. Diese Kurseinheit beschäftigt sich mit den Unterschieden zwischen *Namen*, *Bezeichnern* und *Adressen* und dem Aufbau eines Namenssystems, insbesondere dem *Domain Name System* (DNS). Das DNS im Internet definiert die Syntax der Namen von Entitäten in einer hierarchischen Struktur wie z. B. www.fernuni-hagen.de. Diese Struktur erlaubt eine hierarchische Auflösung eines Namen in eine IP-Adresse. Allerdings ist weder ein Name noch eine IP-Adresse im DNS der eindeutige Bezeichner für eine Entität. Beispielsweise können mehrere Server denselben Namen `www.fernuni-hagen.de` haben und mehrere Server eine einzige IP-Adresse wie z. B. in einem Server-

Cluster besitzen. Auch im Chord-System findet eine Namensauflösung statt, hier über die Verwendung von verteilten Hash-Tabellen, siehe Abschnitt 5.2.3 im Buch.

3.3 Synchronisation und Uhren

Synchronisation

In den Kursen *Betriebsysteme und Rechnernetze* und *Betriebssysteme* haben wir die *Synchronisation* zwischen Prozessen kennengelernt, hier sollen gemeinsame Speicherressourcen durch exklusiven Zugriff kontrolliert genutzt werden. In verteilten Systemen haben die Prozesse im Allgemeinen keinen gemeinsamen Speicher, für den sie ihre Aktionen synchronisieren müssen. Mehrere Algorithmen für die Koordinierung von Zugriffen auf gemeinsame Ressourcen sind in Kapitel 6 im Buch zu finden. Für die Koordinierung z. B. eines Zugriffs auf eine gemeinsame Ressource findet Nachrichtenaustausch statt und dafür muss die Reihenfolge der Anfragen auf eine Ressource festgelegt werden. D. h. die Ereignisse wie z. B. das Senden und das Empfangen einer Nachricht müssen in eine eindeutige Ordnung gebracht werden. Dies geschieht z. B. durch

Zeitstempel

einen physischen oder logischen *Zeitstempel* für jede Nachricht. Dafür benötigt ein verteiltes System natürlich Uhren, die selbst wieder synchronisiert werden müssen. Das Protokoll *NTP* ist ein Beispiel für die Synchronisierung von physischen Uhren (d. h. mit realer Zeit).

NTP

Kommunizierende Prozesse $P_1, \ldots, P_n$ in einem verteilten System brauchen nicht unbedingt Uhren mit realer Zeit, sondern eine von allen akzeptierte Ordnung von Ereignissen im System. Jeder Prozess besteht aus einer Reihenfolge

Ereignis

von Ereignissen. Ein *Ereignis* kann die Ausführung einer Anweisung oder eines Programmteils sein, insbesondere sind das Senden und das Empfangen einer Nachricht Ereignisse. Alle Ereignisse eines einzelnen Prozesses P_i sind unter sich total geordnet, d. h. für zwei beliebige Ereignisse a und b desselben Prozesses P_i gilt immer die *happens-before*-Relation $\rightarrow_i$:

totale Ordnung
happens-before

$$a \rightarrow_i b, \text{ falls } a \text{ vor } b \text{ passiert oder } b \rightarrow_i a, \text{ falls } b \text{ vor } a \text{ passiert.}$$

Da die Ereignisse von P_i mit $1 \leq i \leq n$ total geordnet sind, können wir eine *logische Uhr* C_i für jeden Prozess P_i für $i \in \{1, \ldots, n\}$ definieren: jedem Ereignis a wird eine natürliche Zahl $C_i(a)$ zugeordnet, so dass

logische Uhr

$$C_i(a) < C_i(b), \text{ falls } a \rightarrow_i b \text{ gilt.}$$

Die einfachste Methode zur Definition der Uhrzeit C_i ist, dass bei jedem neuen Ereignis a die Uhr $C_i(a)$ um eine Konstante inkrementiert wird. Beispielsweise werden die logischen Uhren der Prozesse P_1, P_2 und P_3 in Abbildung 6.9 im Buchabschnitt 6.2.1 für jedes neue Ereignis jeweils um 6, 8 und 10 inkrementiert.

Wollen wir aber nun eine logische Uhr für die Ereignisse E *aller* Prozesse $P_1, \ldots, P_n$ im System definieren, dann müssen auch manche (nicht alle) Paare von Ereignissen von verschiedenen Prozessen in Beziehung gebracht werden. Wir erweitern die nur auf jeweils einem Prozess definierten totalen Ordnungen $\rightarrow_i$ zu einer *partiellen Ordnung* $\rightarrow$ für gewisse Ereignispaare in $E \times E$ wie folgt:

partielle Ordnung

Die Relation $a \to b$ (*happens-before*) gilt dann und genau dann, falls eine der folgenden Bedingungen erfüllt ist:

$a \to b$ happens-before

1. Ereignisse a und b gehören zu demselben Prozess P_i für ein $i \in \{1, \ldots, n\}$ und es gilt $a \to_i b$. — lokale Reihenfolge
2. Ereignis a ist das *Senden* bei P_i und Ereignis b das *Empfangen* genau derselben Nachricht bei P_j mit $i, j \in \{1, \ldots, n\}, i \neq j$. — Senden vor Empfangen
3. Es gibt bereits ein Ereignis c, so dass $a \to c$ *und* $c \to b$ gilt. — transitiver Abschluss

Bedingung 3 bedeutet, dass $a \to b$ eben nicht nur direkt wegen Bedingung 1 oder 2 gilt, sondern auch, wenn es eine Kette von solchen direkten Beziehungen (nach 1 oder 2) gibt, an deren Anfang a und an deren Ende b stehen, z. B.: $a \to p \to q \to r \to s \to b$. Diese Transitivitätsbedingung müssen partielle Ordnungen immer erfüllen.

Weil in dieser Definition die Ursächlichkeit von Ereignis a für Ereignis b ausgedrückt wird, sagen wir auch: Das Ereignis a geht dem Ereignis b *kausal voraus*. Dies ist also nur eine andere Bezeichnung für $\to$ bzw. *happens-before*. Die Relation $\to$ ist, wie gesagt, nur eine partielle Ordnung (Halbordnung), denn sie kann nicht für jedes beliebige Paar von Ereignissen ausgewertet werden. — kausal vorausgehen

Nach der Idee von Lamport verwendet man nun solche Zahlen $C(a)$ als Zeitstempel für jedes Ereignis, so dass diese Halbordnung in der Ordnung solcher Zahlen enthalten ist.

Wir nennen also eine Abbildung C von Ereignissen auf Zahlen eine *Lamport-Uhr*, wenn gilt: — Lamport-Uhr

$$\text{Falls } a \to b, \text{ dann } C(a) < C(b). \quad (*)$$

(*)

Wenn nun also Prozesse $P_1, \ldots, P_n$ eine gemeinsame Lamport-Uhr C besitzen sollen, dann müssen sie jeweils eine eigene lokale logische Uhr C_i definieren, so dass $C(a) = C_i(a)$ für alle Ereignisse a aus P_i und, falls $a \to b$, dann $C(a) < C(b)$. Dies kann man erreichen durch eine Anpassung (Erhöhung) der lokalen Uhr bei Nachrichtenempfang. Wie die lokalen logischen Uhren nach den Regeln von Raynal und Singhal konkret gestellt werden, sieht man im angegebenen Algorithmus in der Mitte von Buchabschnitt 6.2.1 (Seite 277 in der deutschen, Seite 246–247 in der englischen Fassung).

Lamport-Uhren kann man zum Beispiel für vollständig geordnetes Multicasting benutzen, wie im darauf folgenden Abschnitt (*Beispiel: Vollständig geordnetes Multicasting* bzw. *Example: Totally Ordered Multicasting*) erläutert wird.

Allerdings gilt die Umkehrung von $(*)$ bei Lamport-Uhren nicht, d. h. aus $C(a) < C(b)$ kann man nicht schließen, dass Ereignis a kausal dem Ereignis b vorausgeht; diese Eigenschaft bekommen wir erst durch die *Vektoruhr* (Buchabschnitt 6.2.2), bei der jeder Zeitstempel ein Zahlentupel $VC(a)$ ist. Hier gilt dann $(*)$ auch mit der gewünschten Umkehrung: — Vektoruhr

$$a \to b \text{ genau dann, wenn } VC(a) < VC(b).$$

Auch die VC-Zeitstempel bilden keine totale Ordnung, denn $VC(a) < VC(b)$ gilt genau dann, wenn die Beziehung „$\leq$“ darin komponentenweise gilt und „$<$“ für mindestens eine Komponente des Tupels.

Ein Zeitstempel $VC(a)$ enthält an seiner i-ten Stelle die Anzahl der Ereignisse von Prozess i, die dem Ereignis a kausal vorausgegangen sind. Aus diesen Zahlen kann ein empfangender Prozess nun Konsequenzen ziehen wie z. B. das Warten auf ihm noch fehlende Ereignisse.

Im Buchabschnitt *Erzwingen kausaler Kommunikation* bzw. *Enforcing Causal Communication* soll nun eigentlich eine sinnvolle Anwendung von Vektoruhren demonstriert werden, was leider wenig verständlich ist, zum einen durch eine fehlende Definition (kausale Ordnung von Nachrichten) und zum anderen durch falsche Beschreibung („$\max\{VC_j[k], ts(m)[k]\}$“) und ein schlecht gewähltes Beispiel, deshalb hier unsere Ergänzungen.

kausal geordnetes Multicasting

Unser Ziel ist ein *kausal geordnetes Multicasting*, wobei die Kommunikation wie in Abbildung 6.10 durch eine Middleware-Schicht von kommunizierenden Prozessen organisiert wird. Insbesondere kann eine Nachricht empfangen (received) und erst verzögert ausgeliefert (delivered) werden. Multicasting soll hier, wie vorher auch, bedeuten, dass alle Nachrichten an alle Prozesse geschickt werden.

kausale Ordnung von Nachrichten

Während beim total geordneten Multicasting oben alle Nachrichten überall in derselben Reihenfolge ausgeliefert werden mussten, soll jetzt nur noch die *kausale Ordnung von Nachrichten* respektiert werden, die wir ähnlich wie die happens-before-Relation „$\rightarrow$“ von Ereignissen definieren: Nachricht m_1 geht einer anderen Nachricht m_2 kausal voraus, falls es einen Prozess gibt, in dem m_1 vor dem Senden von m_2 gesendet oder ausgeliefert wird.

Algorithmus zum Erzwingen kausaler Kommunikation

Der *Algorithmus zum Erzwingen kausaler Kommunikation*[1] soll also das Ausliefern von Nachrichten so lange verzögern, bis alle kausal vorausgehenden Nachrichten ausgeliefert sind. Dafür benutzt er VC-Zeitstempel nach folgenden Regeln:

1. Beim Absenden einer Nachricht erhöht P_i die Komponente $VC_i[i]$ seiner Uhr um 1 und schickt diesen Zeitstempel mit.

2. Eine Nachricht von P_i mit Zeitstempel S, die bei P_j empfangen wird, darf erst dann ausgeliefert werden, wenn zwei Bedingungen erfüllt sind:

$$S[i] = VC_j[i] + 1$$
$$\text{und } S[k] \leq VC_j[k] \text{ für } k \neq i$$

3. Direkt vor der Auslieferung einer Nachricht, die P_j von P_i bekommen hat, wird $VC_j[i]$ um 1 erhöht.

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto. Causal Memory: Definitions, Implementations, and Programming. Distributed Computing 9 (1995), pp. 37–49.

Mit der ersten Aktion zählt P_i seine abgesendeten Nachrichten. Durch die erste Auslieferungsbedingung (in 2.) stellt der Empfänger sicher, dass er alle vorherigen Nachrichten des Absenders schon vorher ausgeliefert hat, mit der zweiten Bedingung außerdem, dass auch mindestens die Nachrichten von anderen Prozessen bereits ausgeliefert sind, die der Absender vor dem Senden bereits ausgeliefert hatte. Mit der dritten Aktion zählt P_j die ausgelieferten Nachrichten, die von P_i stammen.

Das in Abbildung 6.13 angegebene Beispiel ist zwar im Prinzip korrekt, aber zu einfach, um das zu illustrierende, kausal geordnete Multicasting vom vorherigen, total geordneten zu unterscheiden. Zudem ist die Nummerierung der Prozesse mit P_0, P_1, P_2 recht ungeschickt, weil dann auch die Tupel (Vektoren) unüblich bei der 0-ten Komponente anfangen müssen.

Betrachten wir also unser Beispiel in Abbildung 1, hier werden insgesamt vier Nachrichten (per Multicast) versendet, zwei von P_1 und zwei von P_2. Zunächst senden P_1 und P_2 jeweils eine Nachricht, hier mit den Zeitstempeln

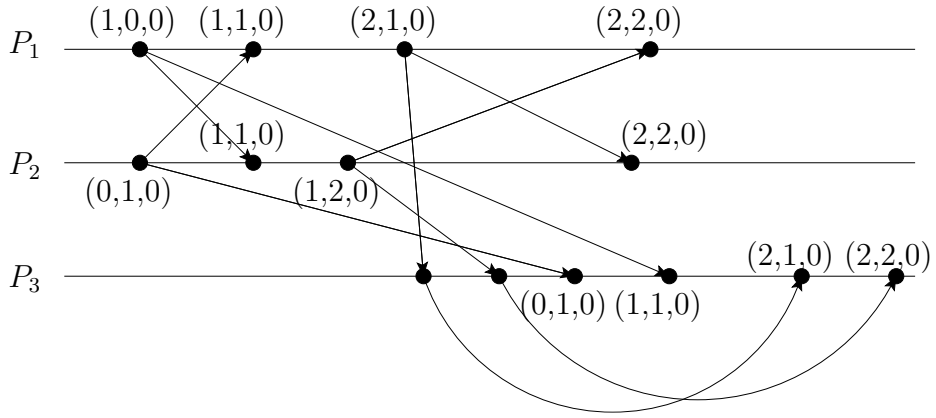

Abbildung 1: Reihenfolge der Lese- und Schreiboperationen mit angepassten Zeitstempeln.

$(1,0,0)$ und $(0,1,0)$, diese sind voneinander unabhängig und deshalb nebenläufig, sie können in unterschiedlicher Folge ausgeliefert werden. Die Nachricht $(0,1,0)$ kommt sogleich bei P_1 an, wird nach der zweiten Regel ausgeliefert, und P_1 aktualisiert seine Vektoruhr auf $(1,1,0)$. Jetzt senden P_1 seine zweite Nachricht $(2,1,0)$ und P_2 seine zweite Nachricht $(1,2,0)$ in die Welt, diese werden bei P_2 bzw. P_1 ausgeliefert, was beide Uhren auf $(2,2,0)$ stellt.

Aus irgendeinem Grund haben sich die Nachrichten an P_3 verspätet und kommen sogar in der anderen Reihenfolge $(2,1,0)$, $(1,2,0)$, $(0,1,0)$ und $(1,0,0)$ an. Deshalb kann $(2,1,0)$ nach der zweiten Regel nicht sofort ausgeliefert werden, und auch $(1,2,0)$ noch nicht. Wohl aber kann dann $(0,1,0)$ an die Reihe kommen, was zu eben diesem Zeitstempel in der Uhr von P_3 führt. Weiterhin muss $(2,2,0)$ noch warten und auch das $(1,2,0)$ kann noch nicht ausgeliefert werden. Erst das eintreffende $(1,0,0)$ lässt sich nach Regel 2 ausliefern, was zu Zeitstempel $(1,1,0)$ führt und dann zur Auslieferung von zuerst $(2,1,0)$ und ganz zum Schluss $(1,2,0)$. Am Ende stehen natürlich alle Uhren gleich auf $(2,2,0)$.

In unserem Beispiel sind also die beiden Nachrichten $m_1 = (1, 0, 0)$ und $m_2 = (0, 1, 0)$ nebenläufig, aber auch $m_3 = (2, 1, 0)$ und $m_4 = (1, 2, 0)$. Deshalb sind alle drei vorkommenden Auslieferungsreihenfolgen zulässig: Bei P_1 haben wir m_1, m_2, m_3, m_4, bei P_2 ist es m_2, m_1, m_4, m_3, und bei P_3 sogar m_2, m_1, m_3, m_4. Die kausale Ordnung ist bei allen diesen Reihenfolgen eingehalten worden.

wechselseitiger Ausschluss

Auch zur Synchronisation gehört der in Buchabschnitt 6.3 angesprochene *wechselseitige Ausschluss*.

3.4 Lernziele

Nach der Bearbeitung dieser Kurseinheit sollten Sie z. B.

- erklären können, was Namen, Bezeichner, Adressen sind und wo die Unterschiede liegen,
- Beispiele von Namenssystemen wie das UNIX-Dateisystem, das DNS und ihre Namenauflösungen angeben können,
- beschreiben können, wie die Auflösung eines Schlüssels in einen Nachfolger durch Finger-Tabellen im Chord-System funktioniert,
- Beispiele zur Notwendigkeit der Uhrzeitsychronisation nennen können,
- erläutern können, wie NTP und der Berkeley-Algorithmus funktionieren,
- definieren können, was *Lamport-Uhren* sind und wie diese synchronisiert werden,
- berichten können, wie man mit Hilfe der Lamport-Uhren ein *total geordnetes Multicasting* realisieren kann,
- erklären können, was *Vektoruhren* sind und wie deren Synchronisation funktioniert,
- ein Beispiel für die Anwendung von Vektoruhren beschreiben können,
- die verschiedenen Algorithmen zur Koordinierung der Zugriffe auf gemeinsame Ressourcen und die damit verbundenen Probleme darstellen können.

4 Konsistenz und Replikation

4.1 Lesehinweise

Bitte lesen Sie Kapitel 7 im Buch, es wird empfohlen, die Abschnitte 7.2.1 und 7.5.1 (jeweils „Stufenlose Konsistenz“) auszulassen.

4.2 Konsistente Speichersysteme

Ein verteiltes System soll sich durch eine große Zuverlässigkeit und eine gute Leistung auszeichnen. Mit *Zuverlässigkeit* meinen wir die Wahrscheinlichkeit, dass das System innerhalb eines Zeitintervalls ohne Unterbrechung funktioniert. Ein Ansatz dafür ist die *Replikation* von Daten, also die gleichzeitige Verwendung von mehreren Kopien über mehrere Standorte. Durch diese *Skalierungstechnik* bekommen wir aber Probleme mit der *Konsistenz*, d. h. nicht alle Replikate sind zu jeder Zeit identisch, obwohl sie es idealerweise sein sollten. Wenn eine Kopie verändert wird, müssen jedenfalls möglichst bald alle anderen auch auf denselben Stand gebracht werden.

Zuverlässigkeit

Replikation

Konsistenz

Durch die Übertragungsverzögerung von Nachrichten in Netzwerken ist die Realisierung von Konsistenz durchaus anspruchsvoll, deshalb ist es sehr sinnvoll, zunächst einmal zu definieren, welche Eigenschaften gewünscht werden. Wir benötigen ein Konsistenzmodell, dafür müssen wir zuerst ein Speichersystem definieren.

Ein *Speichersystem* besteht aus einer Menge von Prozessen, die jeweils ein Replikat auf einem lokalen Speicher besitzen und die Operationen zur Änderung durchführen, siehe Abbildung 2. Die Prozesse im System kommunizieren miteinander per Nachrichtenaustausch über ein Netzwerk, um z. B. die Reihenfolge der Operationen zu koordinieren. Das *Konsistenzmodell* eines Speichersystems verspricht verteilten Anwendungen die Korrektheit der ausgelieferten Daten. Beispielsweise verspricht es, dass eine Leseoperation auf einer Dateneinheit ein Ergebnis zurückgibt, das sich aus der letzten Schreiboperation auf dieser ergibt. Wir bezeichnen ein Speichersystem mit einem Konsistenzmodell als *konsistentes Speichersystem*. Es nimmt Anfragen entgegen und liefert Antworten, die die versprochenen Eigenschaften aufweisen.

Speichersystem

Konsistenzmodell

konsistentes Speichersystem

Der Begriff *Konsistenz* wird für Verteilte Systeme in einem anderen Sinne verstanden, als man von den bekannten *Datenbanken* her vielleicht gewohnt ist. Bei Datenbanken verbindet man die Konsistenz mit der Garantie der *ACID*-Eigenschaft durch Transaktionen, die mittels eines Locking-Mechanismus realisiert werden.

Datenbanken

ACID

Die Konsistenz in Verteilten Systemen ist jedenfalls schwächer. Dabei unterscheiden wir die Konsistenzmodelle aus der Sicht der *Server* von denen aus *Client*-Sicht. Die Server brauchen *datenzentrierte Konsistenzmodelle*, um die Operationen auf den Replikaten so zu organisieren, dass vorgegebenen Kriterien eingehalten werden, siehe dazu Buchabschnitt 7.2.2 und unsere Ergänzungen in Abschnitt 4.3 und 4.4. Die Clients benötigen *Client-zentrierte Konsistenzmodelle*, siehe Buchabschnitt 7.3, für sie ist das Speichersystem eine Black-

Server

datenzentriert

Client-zentriert

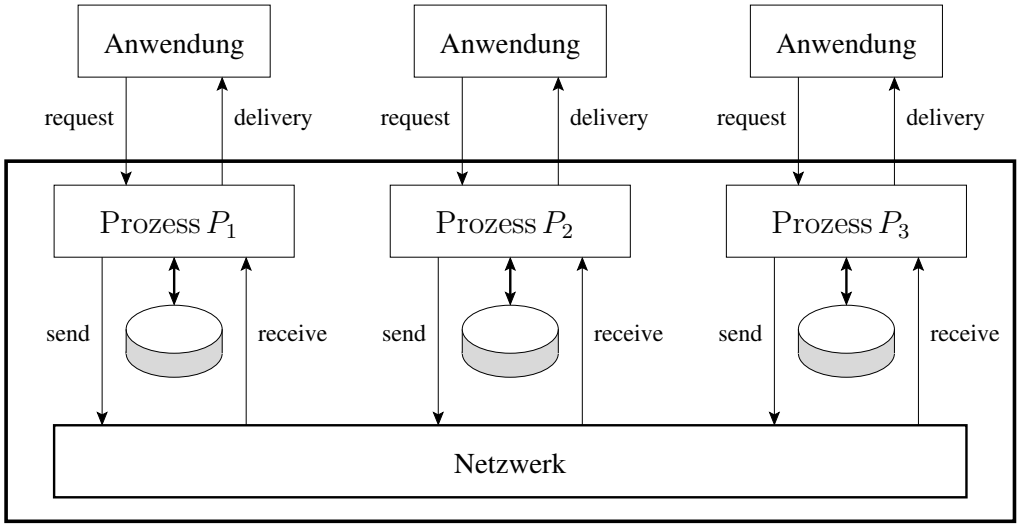

Abbildung 2: Ein Speichersystem aus kommunizierenden Prozessen mit lokalen Speichern, das Anforderungen von Anwendungen bedient.

box, die Ihnen die Daten mit bestimmten Qualitätskriterien zur Verfügung stellt.

4.3 Sequenzielle Konsistenz

$R(x)a$
$W(y)b$

Operationen zum Lesen und Schreiben von Datenelementen sind Aufträge von Anwendungen an einzelne Prozesse eines Speichersystems: $R(x)a$ bezeichnet das Lesen von Datenelement x mit dem Ergebnis (Wert) a, und $W(y)b$ ist unsere Bezeichnung für das Schreiben auf Datenelement y mit dem Wert b. Wenn zur Verdeutlichung der beauftragte Prozess i genau genannt werden soll, dann können wir auch $R_i(x)a$ und $W_i(y)b$ schreiben.

Serialisierung

Eine *Serialisierung* einer (ungeordneten) Menge von solchen Operationen ist eine Reihenfolge dieser Operationen, so dass jede Leseoperation auf einem bestimmten Datenelement x genau denselben Wert ergibt, den die in dieser Reihenfolge letzte Schreiboperation auf x vor diesem Lesen festgelegt hat. Mit anderen Worten: das letzte Schreiben von x vor $R(x)a$ muss ein $W(x)a$ sein.

Wir betrachten nun n Prozesse $P_1, \ldots, P_n$, wobei jeder dieser Prozesse eine eigene Ausführungsgeschichte hat, d. h. Prozess i hat seine Folge F_i von Schreib- und Leseoperationen auf Datenelementen. Alle Folgen zusammen bilden die Geschichte G, dazu definieren wir den Begriff *sequenzielle Konsistenz* folgendermaßen.

sequenzielle Konsistenz

> Die Geschichte $G = (\ F_1, \ldots, F_n\ )$ hat die Eigenschaft *sequenzielle Konsistenz*, wenn es eine Serialisierung aller Operationen aus G gibt, bei der jede Folge F_i in genau ihrer Reihenfolge darin vorkommt.

Natürlich brauchen die Operationen von F_i nicht unbedingt alle zusammenhängend („am Stück“) in der Serialisierung vorkommen, sondern nur so, dass die Ordnung innerhalb von F_i weiter eingehalten wird.

Betrachten wir nun die Beispiele in Abbildung 7.5 im Buch, hier gilt für das linke Teilbild (a):

$$\begin{aligned} F_1 &= [\, W(x)a \,] \\ F_2 &= [\, W(x)b \,] \\ F_3 &= [\, R(x)b, R(x)a \,] \\ F_4 &= [\, R(x)b, R(x)a \,] \\ G &= (\, F_1, F_2, F_3, F_4 \,) \end{aligned}$$

mit der Serialisierung

$$[\, W(x)b, R(x)b, W(x)a, R(x)a \,]$$

die offensichtlich alle F_i-Reihenfolgen respektiert. Für das rechte Teilbild (b) ist der einzige Unterschied

$$F_4 = [\, R(x)a, R(x)b \,]$$

mit der Konsequenz, dass es hier keine Serialisierung gibt, die die Reihenfolgen von F_3 und F_4 gleichzeitig respektiert.

Um nun für einen konkreten Datenspeicher die Eigenschaft *sequenzielle Konsistenz* zu garantieren, braucht man Algorithmen, die im laufenden Betrieb die Operationen so organisieren, dass die Ausführungsgeschichte, im Nachhinein betrachtet, die genannten Eigenschaften aufweist.

Primary Backup Protocol

Der einfachste solche Algorithmus ist im Buchabschnitt 7.5.2 unter dem Namen *Primary Backup Protocol* (Abbildung 7.20) beschrieben. Leseoperationen werden von jedem Prozess selbständig durchgeführt und beantwortet (R1 und R2), doch alle Schreiboperationen (W1) werden an einen einzigen, zentralen Server übergeben (W2), der alle lokalen Speicher zu einem Update auffordert (W3) und sich dies bestätigen lässt (W4). Erst dann wird dem Auftraggeber der Operation die Durchführung bestätigt (W5).[2]

Man kann sich leicht vorstellen, dass diese blockierende (sperrende) Methode höchstens dann gut funktioniert, wenn es relativ wenige Schreiboperationen im Vergleich zu vielen Leseoperationen gibt.

Gewisse Verbesserungen erreicht man durch nicht-blockierende Vorgehensweise mittels vorzeitiger Bestätigung, was aber im Fehlerfall (eine lokale Kopie fällt aus) nachteilig ist.

4.4 Kausale Konsistenz

Die sequenzielle Konsistenz erfordert in Verteilten Systemen einen hohen Kommunikationsaufwand und ist eigentlich nicht unbedingt nötig. Die kausale Konsistenz ist eine Abschwächung dieses Prinzips auf das wirklich Notwendige. Wie im vorherigen Abschnitt 4.3 betrachten wir eine Menge von Prozessen $P_1, \ldots, P_n$, und Schreib- und Leseoperationen wie $W_i(x)a$ und $R_j(y)b$ auf gemeinsam genutzten Datenelementen.

kausal vorausgehen

Ähnlich wie in Abschnitt 3.3 (kausal vorausgehen, Seite 13) wollen wir eine kausale Relation „⇒“ zwischen solchen Operationen definieren, die sich gegenseitig beeinflussen:

Für eine Ausführungsgeschichte $G = (F_1, \ldots, F_n)$ von Folgen von Operationen in Prozessen gilt für zwei ihrer Operationen o_1 und o_2 die Beziehung $o_1 \Rightarrow o_2$ (o_1 geht o_2 kausal voraus) dann und genau dann, wenn eine der drei folgenden Bedingungen erfüllt ist.

kausale Ordnung

lokale Reihenfolge

1. o_1 und o_2 sind Operationen desselben Prozesses, und o_1 wird in der Ausführungsgeschichte dieses Prozesses vor o_2 ausgeführt.

Schreiben vor Lesen

2. o_1 ist eine *write*-Operation auf einem Datenelement, sagen wir $W_i(x)a$, und o_2 ist eine *read*-Operation $R_j(x)a$ auf demselben x mit demselben Ergebnis a von einem anderen Prozess P_j, $j \neq i$.

transitiver Abschluss

3. Es gibt bereits eine Operation o, so dass $o_1 \Rightarrow o$ *und* $o \Rightarrow o_2$ gilt.

Die kausale Konsistenz wird nun aus der Sicht jedes einzelnen Prozesses P_i definiert: Wichtig ist für P_i nur seine lokale Geschichte und die Schreiboperationen anderer Prozesse auf gemeinsamen Datenelementen. Sei also A_i die Menge aller Operationen in F_i und aller Schreiboperationen in G, das ist sozusagen die Menge aller für P_i relevanten Operationen.

kausal konsistent

Eine Ausführungsgeschichte $G = (F_1, \ldots, F_n)$ von n Prozessen heißt *kausal konsistent*, wenn es für jeden Prozess P_i eine Serialisierung der Menge A_i gibt, die die kausale Ordnung repektiert.

Betrachten wir das Beispiel in Abbildung 7.8 im Buch:

$$\begin{aligned} F_1 &= [W_1(x)a, W_1(x)c] \\ F_2 &= [R_2(x)a, W_2(x)b] \\ F_3 &= [R_3(x)a, R_3(x)c, R_3(x)b] \\ F_4 &= [R_4(x)a, R_4(x)b, R_4(x)c] \end{aligned}$$

Diese Geschichte ist *nicht sequenziell konsistent*, da sich die Reihenfolgen von $R(x)c$ und $R(x)b$ in F_3 und F_4 widersprechen. Sie ist aber *kausal konsistent*, denn sie lässt sich z. B. folgendermaßen serialisieren:

$$\begin{aligned} A_1 &: [W_1(x)a, W_1(x)c, W_2(x)b] \\ A_2 &: [W_1(x)a, R_2(x)a, W_2(x)b, W_1(x)c] \\ A_3 &: [W_1(x)a, R_3(x)a, W_1(x)c, R_3(x)c, W_2(x)b, R_3(x)b] \\ A_4 &: [W_1(x)a, R_4(x)a, W_2(x)b, R_4(x)b, W_2(x)c, R_4(x)c] \end{aligned}$$

Zum Vergleich ist in Abbildung 7.9(a) ein Beispiel für eine nicht kausal konsistente Ausführungsgeschichte angegeben.

[2]Korrektur zu Abbildung 7.20: der Pfeil W5 sollte direkt vom Primärserver (primary server) zum Client gehen.

Wie kann jetzt die kausale Konsistenz in einem verteilten Speichersystem erreicht werden? Hier können wir die Vektoruhr-Zeitstempel benutzen, die wir in der letzten Kurseinheit kennengelernt haben. Wir brauchen den (korrigierten) Algorithmus zum Erzwingen kausaler Kommunikation (Seite 14) für kausal geordnetes Multicasting (Ahamad et al. 1995) mit seinen Regeln nur folgendermaßen anzupassen:

1. Jeder Prozess kann eine eigene Schreiboperation auf seiner lokalen Kopie selbst initiieren, anschließend sendet er darüber eine Nachricht mit seinem Zeitstempel an alle anderen, siehe Regel 1.
2. Das Nachvollziehen einer Schreiboperation auf einem lokalen Replikat erfolgt genau nach den Regeln der Auslieferungsoperation, siehe Regel 2.
3. Die Anpassung der lokalen Vektoruhr (Erhöhung einer Komponente um 1) geschieht direkt vor einer Schreiboperation, siehe Regel 1 und 3.

Vereinfacht gesagt entspricht das erstmalige Schreiben eines Datenelements dem Absenden einer Multicast-Nachricht, während das nachvollziehende Schreiben einer Auslieferung entspricht.

Auch das Finden der besten Standorte für replizierende Server und die Verteilung von Inhalten sind interessante Probleme, die in dieser Kurseinheit behandelt werden.

4.5 Lernziele

Nach der Bearbeitung der Kurseinheit sollten Sie z. B.

- erklären können, warum Replikate gebraucht werden und warum es schwierig ist, harte Konsistenz zu realisieren,
- Konsistenzmodelle erklären können,
- den unterschiedlichen Blickpunkt auf die Konsistenz von Datenbanken und verteilte Systemen beschreiben könnnen,
- Daten-zentrierte und Client-zentrierte Konsistenzmodelle und ihre Anwendungen beschreiben können,
- das Standortproblem der Platzierung von replizierten Servern und Inhalten erklären können,
- erläutern können, wie die Aktualisierung von Replikaten funktioniert,
- berichten können, welche Konsistenzprotokolle es gibt, welche Ziele sie haben und wie sie arbeiten.

5 Fehlertoleranz

5.1 Lesehinweise

Bitte lesen Sie in dieser Reihenfolge:

- unsere Anmerkungen in Abschnitt 5.2,
- Buchkapitel 8.1 und weiter bis einschließlich 8.2.2,
- unser Ersatzkapitel in Abschnitt 5.3,
  Buchabschnitt 8.2.3 können Sie auslassen,
- Buchkapitel 8.2.4,
- unsere Bemerkungen in Abschnitt 5.4,
- Buchkapitel 8.3 bis 8.5
  und gerne auch 8.7 (Summary bzw. Zusammenfassung, wie immer).

5.2 Anmerkungen zu 8.1 und 8.2

Die Übersetzung dieses Kapitels der englischen Buchausgabe ins Deutsche ist nicht ganz leicht, zugegeben, aber die deutsche Buchausgabe ist hier wirklich ganz schwach, immer wieder werden wechselnde deutsche Begriffe für denselben englischen gebraucht oder umgekehrt. Deswegen wollen wir die wichtigen Begriffe und ihre Definitionen im Folgenden auf Deutsch und gleich auch auf Englisch nennen.

Man beachte aber den Unterschied zwischen dem allgemeinsprachlichen Gebrauch von Wörtern (also was wir normalerweise meinen, wenn wir ein Wort „einfach so“ verwenden) und dem fachsprachlichen Gebrauch (wir geben eine Definition an und meinen das Wort nur genau in diesem Sinne). Es kann durchaus sein, dass zwei Wörter im allgemeinsprachlichen Sinne als Synonyme verwendet werden können (z. B. Verlässlichkeit und Zuverlässigkeit, dependability and reliablity), die als Fachwörter aber verschieden definiert sind.

Zuerst beschäftigen wir uns mit den Grundbegriffen der *Verlässlichkeit* (Dependability), nämlich der

Verlässlichkeit

- *Verfügbarkeit* (Availability),
- *Zuverlässigkeit* (Reliability),
- *Sicherheit* (Safety) und
- *Wartbarkeit* (Maintainability).

Die Verfügbarkeit gibt die Wahrscheinlichkeit an, dass das System zu einem beliebigen Zeitpunkt benutzbar ist, z. B. 95 % Verfügbarkeit. Die Zuverlässigkeit misst dagegen die Wahrscheinlichkeit, dass das System innerhalb eines Zeitintervalls ohne Ausfall funktioniert, diesen Begriff hatten wir schon in Kurseinheit 4 kennengelernt. Eine Zuverlässigkeit von z. B. 99 % pro Stunde entspricht einer Zuverlässigkeit von $(99\ \%)^{24} \approx 78{,}5\ \%$ pro Tag. Aus den Bei-

Verfügbarkeit ≠ Zuverlässigkeit!

spielen im Buch wird auch deutlich, dass sich Verfügbarkeit und Zuverlässigkeit nicht ineinander umrechnen lassen. Sicherheit (Safety, zu unterscheiden von Security) ist gegeben, wenn im Fehlerfall nichts wirklich Schlimmes passiert. Gute Wartbarkeit im Sinne des Buches schließlich ist vielleicht eher die gute Wiederherstellbarkeit: nach einem Fehler soll es leicht möglich sein, den Betriebszustand wieder zu erreichen.

Das Hauptthema dieser Kurseinheit ist Fehlertoleranz, zur Definition des Begriffs siehe weiter unten. Zunächst müssen wir die drei Begriffe

- *Ausfall* (Failure),
- *Fehler* (Error) und
- *Fehler* (Fault)

unterscheiden, was so auf Deutsch natürlich Probleme macht. Deshalb wollen wir hier lieber *Error* als deutschen Begriff benutzen, dann haben wir auch drei verschiedene Wörter.

Ausfall
Error
Fehler

Ein *Ausfall* liegt vor, wenn ein System einmal nicht das macht, was von ihm erwartet wird. Ein *Error* ist ein Systemzustand, der eigentlich nicht vorkommen soll und doch eingetreten ist. Die Ursache eines Errors ist ein *Fehler*.[3]

Beispiel: wie viele „Fehler“ passieren bei einer „Division durch Null“?

Beispiel *Division durch Null*:

Ein Programmierer übersieht, dass eine Variable unter gewissen Umständen den Wert Null annehmen kann, und er lässt ohne weitere Tests eine Division durch diesen Wert durchführen. Das ist ein Programmierfehler (engl. Mistake, noch so ein Wort!). Dieser Fehler (Fault) ist im Programm enthalten. Nun läuft das Programm und das Unerwartete passiert: *Error, division by Zero*. Der Error ist der Zustand unmittelbar nach dem Auftreten dieser unerlaubten Operation, der Grund dafür ist der Fehler im Programm. Der Error kann nun zum Ausfall (Failure) des Gesamtsystems führen.

Fehler über Fehler...

Dasselbe als abschreckendes Beispiel, kurz und schwer verständlich: Ein Programmierer macht einen Fehler, sein Programm enthält einen Fehler, der Fehler passiert und führt zum Fehler (auch eine mögliche Übersetzung von Failure).

fehlertolerant
k-fehlertolerant

Ein System ist *fehlertolerant*, wenn es in der Lage ist, trotz gewisser Fehler (Faults) zu funktionieren. Genauer gesagt, ein System ist *k-fehlertolerant*, wenn es trotz Fehlern in k Komponenten noch korrekt funktioniert.

Replikation
Ausfälle

Wie viel *Replikation* wird nun für Fehlertoleranz benötigt? Wenn nur *Ausfälle* (eine Komponente funktioniert entweder korrekt oder macht gar nichts) betrachtet werden sollen, dann ist klar, dass $(k+1)$-fache Replikation ausreicht, um k Fehler zu überstehen, denn mindestens ein korrekter Prozess bleibt immer übrig.

Schwieriger wird es, wenn sogenannte *byzantinische Fehler* berücksichtigt werden müssen, d. h. eine fehlerhafte Komponente kann nicht nur ausfallen, sondern zufällige oder sogar böswillige Antworten geben. Dann kann man versuchen, eine Entscheidung per *Mehrheitsbeschluss* zu treffen: Wir verwenden mindestens $2k+1$ Replikate, und selbst wenn k davon bösartig agieren, bleiben $k+1$ und damit die sichere Mehrheit korrekt.

byzantinische Fehler

Mehrheitsbeschluss

Es ist nun in der Praxis kaum möglich, Garantien abzugeben, dass maximal genau k Fehler passieren können, aber niemals $k+1$, schon deshalb sind die Aussagen über $k+1$ bzw. $2k+1$ Replikate zu simpel. Außerdem sind noch keine Details über die Prozesse und ihre Kommunikation geklärt worden, dafür benötigen wir genauere Modelle, wie wir im nächsten Abschnitt sehen werden.

5.3 Ersatzkapitel für 8.2.3: Agreement in Faulty Systems – Einigung in fehlerhaften Systemen

Dieser Buchabschnitt ist leider schon im englischen Original so fehlerhaft, dass wir ihn nicht zum Lesen empfehlen können, insbesondere falsch bzw. irreführend sind die Abbildungen und Beispiele. Die deutsche Übersetzung macht den Text nicht gerade besser, deshalb lesen Sie stattdessen bitte diesen Abschnitt.

Um Fehlertoleranz zu erreichen, replizieren wir Prozesse, d. h. in einer *Gruppe von mehreren Prozessen* sollen im Prinzip alle dieselben Aufgaben bekommen, aber einige arbeiten möglicherweise nicht korrekt, trotzdem soll das Gesamtsystem korrekt arbeiten.

Gruppe von mehreren Prozessen

5.3.1 Das Einigungsproblem

Das allgemeine Ziel beim *Einigungsproblem* (auch Konsensproblem) ist die gemeinsame Entscheidung: die fehlerfreien Prozesse arbeiten zusammen und finden in endlicher Zeit einen *Konsens*, die fehlerhaften Prozesse der Gruppe können durch Ausfälle oder ggfs. selbst durch byzantinische Fehler daran nichts ändern. Erfunden wurde das Einigungsproblem für die Flugsicherheit:[4] wenn in einem Flugzeug mehrere, unabhängige Höhenmesser vorhanden sind, die auch widersprüchliche Informationen liefern können, sollen sich die Prozessoren des Gesamtsystems trotzdem daraus auf eine Antwort (aktuelle Höhe) einigen.

Einigungsproblem

Konsens

Motivation

Wenn man nun etwas genauer hinschaut, dann muss man verschiedene Voraussetzungen unterscheiden bezüglich des Verhaltens von Prozessen und der Kommunikation innerhalb des Systems: Nicht immer ist eine Einigung über-

[3] „the cause of an error is called a fault“, im Buch auf Seite 323 bzw. auf Seite 356 völlig unverständlich übersetzt mit Ausfall, Störung, Defekt, Fehlerursache...

[4] N. Lynch (1996), p. 100.

haupt möglich. Turek und Shasha (1992) unterscheiden die folgenden, jeweils voneinander unabhängigen Aspekte, siehe auch Abbildung 3 auf Seite 27.

synchron oder asynchron

1. *Synchron oder asynchron*: Prozesse einer Gruppe arbeiten asynchron, wenn manche von ihnen beliebig viel mehr arbeiten können als andere.Synchron sind sie also im Gegensatz dazu, wenn es eine Konstante c gibt, so dass jeder Prozess mindestens einen Schritt voran kommt, während ein anderer Prozess $c + 1$ Schritte macht.

beschränkt oder unbeschränkt

2. *Beschränkte oder unbeschränkte Verzögerung* (bounded/unbounded): Beschränkt bedeutet, dass global eine Maximalzeit bekannt ist, innerhalb derer eine beliebige Nachricht zugestellt sein muss.

geordnet oder ungeordnet

3. *Geordnete Auslieferung*: Nachrichten desselben Senders an denselben Empfänger werden in genau derselben Reihenfolge ausgeliefert, wie sie abgesendet wurden. In der ungeordneten Auslieferung dagegen können Nachrichten sich gegenseitig überholen.

Unicast oder Multicast

4. *Unicast oder Multicast*: Nachrichten gehen jeweils nur an einen (Unicast) oder an alle (Multicast) Prozesse der Gruppe gleichzeitig.

Abbildung 3 zeigt nun als Tabelle jede mögliche Kombination dieser vier Aspekte (insgesamt $2^4 = 16$ Fälle). Eine Einigung kann demnach dann erzielt werden, wenn

- die Prozesse synchron arbeiten und die Verzögerung beschränkt ist,
- die Prozesse synchron arbeiten und die Auslieferung geordnet stattfindet
- oder die Nachrichten per Multicast an alle geschickt und geordnet ausgeliefert werden.

Bitte Vorsicht, Abbildung 8.4 im Buch (*Circumstances under which...*, *Bedingungen unter denen...*) wurde nicht korrekt aus der Originalarbeit von Turek und Shasha übernommen, die X-Zeichen sind komplett falsch gesetzt, unsere Abbildung 3 entspricht dem Original.

synchron und beschränkt

Im Folgenden gehen wir davon aus, dass die (praxisnahen) Eigenschaften *synchron und beschränkt* gegeben sind.

Einigungsalgorithmus

Formal können wir einen *Algorithmus zur Lösung eines Einigungsproblems* nun folgendermaßen beschreiben. Eine Gruppe von n Prozessen, P_1, P_2, ..., P_n, ist beteiligt an einer anstehenden Entscheidung, manche Prozesse arbeiten inkorrekt oder sogar böswillig. Jeder Prozess $P_i, 1 \leq i \leq n$, macht für die Entscheidung einen Vorschlag $v_i \in \mathbb{N}$ und teilt diesen jedem anderen Prozess $P_j, j \neq i$, mit. Nach einer gewissen Zeit weiterer Kommunikation unter-

1. Process	2. Delay	3. Ordering	4. Transmission	unicast		multicast	
				ordered	unordered	ordered	unordered
synchronous	bounded			X	X	X	X
	unbounded			X		X	
asynchronous	bounded					X	
	unbounded					X	

Abbildung 3: Tabelle der Bedingungen für verteilte Einigung nach Turek und Shasha (1992), ein X kennzeichnet die Kombinationen der Bedingungen, unter denen verteilte Einigung garantiert werden kann, Korrektur von Abbildung 8.4 im Buch.

einander berechnet jeder Prozess P_i seine Entscheidung $d_i \in \mathbb{N}$. Ein Einigungsalgorithmus muss dabei die folgenden *Bedingungen* erfüllen:

Bedingungen:

- *Terminierung:* Jeder korrekte Prozess P_i trifft eine Entscheidung d_i. Inkorrekte Prozesse können auch nichts entscheiden.

  Terminierung

- *Einigung:* Alle korrekten Prozesse treffen dieselbe Entscheidung, d. h. falls P_i und P_j korrekt arbeiten, dann muss auch $d_i = d_j$ gelten, selbst wenn ihre Vorschläge v_i und v_j verschieden waren.

  Einigung

- *Integrität:* Falls alle Prozesse, darunter P_i, denselben Wert $v_i = v$ vorgeschlagen haben, dann entscheiden sie sich alle auch genau dafür, d. h. $d_i = v$.

  Integrität

Wir unterscheiden nun, welche Fehler vorkommen können, entweder nur Ausfälle oder auch byzantinische Fehler. Der erste Fall ist der einfachere: Ausfälle (crash failures) können in diesem Modell jederzeit passieren, danach ist der betroffene Prozess aber *beendet* und macht also gar nichts mehr.

Ausfall beendet Prozess

5.3.2 Ein Einigungsalgorithmus in synchronen Systemen mit Ausfällen

Wenn *nur Ausfälle* vorkommen können, dann kann der Einigungsalgorithmus nach Attiya und Welch (2004), dort Abschnitt 5.1.3, folgendermaßen vorgehen:

nur Ausfälle

Runden

Zu Anfang wird ein $f < n$ festgelegt für die maximale Zahl der Ausfälle. Durch die Synchronitäts-Eigenschaft können die n Prozesse rundenweise zusammenarbeiten, eine Runde besteht aus zwei Schritten, die jeder Prozess durchführt:

Erzeugen und Senden

1. Basierend auf den ihm vorliegenden Informationen erzeugt der Prozess eine Nachricht und sendet sie per Multicast an die ganze Gruppe.

Empfangen und Verarbeiten

2. Alle ankommenden Nachrichten werden empfangen und zu neuen Informationen verarbeitet.

Eine neue Runde startet erst dann, wenn alle ihre letzte Runde beendet haben oder ausgefallen sind.

Einigungsalgorithmus

$f + 1$ Runden

Der *Einigungsalgorithmus* arbeitet in genau $f+1$ Runden. Jeder der n Prozesse hat anfangs (ohne Informationen der anderen) einen Vorschlag, d. h. eine Zahl. Diesen teilt er zunächst in Schritt 1 den anderen mit, in Schritt 2 der ersten Runde empfängt er die Vorschläge der anderen und sammelt sie in einer Menge. In jeder weiteren Runde (2 bis $f + 1$) sendet er in Schritt 1 alle Vorschläge, die er noch nicht gesendet hatte, und sammelt weiterhin in Schritt 2. Nach der $(f + 1)$-ten Runde entscheidet sich jeder Prozess für das Minimum aus der Menge der ihm bekannten Vorschläge.

Korrektheit

Warum arbeitet dieser Algorithmus korrekt? Die Schwierigkeit liegt ja darin, dass ein Ausfall zu jeder Zeit passieren kann, dass also ein Prozess eine Nachricht nur an einen Teil der Empfänger schickt, im schlimmsten Fall nur an einen, und dann ausfällt. Auch solche Nachrichten müssen allen funktionierenden Prozessen irgendwann bekannt sein oder komplett unter den Tisch fallen.

Bei $f+1$ Runden und maximal f Ausfällen gibt es mindestens eine Runde, die ohne Ausfall stattfindet. Die funktionierenden Prozesse in dieser „sauberen“ Runde haben auch in allen vorherigen Runden funktioniert, das heißt, dass alle Prozesse dieser Teilmenge nach der sauberen Runde die exakt gleichen Vorschläge kennen und gesammelt haben. Daran kann sich in möglicherweise folgenden Runden gar nichts mehr ändern. Am Ende werden sich also alle noch funktionierenden Prozesse für denselben Vorschlag entscheiden.

Sind nun die Bedingungen für einen Einigungsalgorithmus erfüllt? Die *Terminierung* ist sicher, denn die Anzahl der Runden ist fest. Die *Einigung* haben wir gerade eben bewiesen, und die *Integrität* ist auch garantiert, denn wenn alle Vorschläge von Anfang an gleich sind, ist auch das Minimum dasselbe.

Das Hauptergebnis für dieses Modell der Fehlertoleranz ist also nicht ganz unerwartet folgendes: Der *Einigungsalgorithmus in $f+1$ Runden* arbeitet korrekt, wenn maximal f Ausfälle vorkommen können.

5.3.3 Das Problem der byzantinischen Generäle

Die Bezeichnung für dieses Problem wurde geprägt von Lamport et al. (1982), sie dachten dabei an hochrangige Offiziere von Armeen, die sich durch Nach-

richten darüber einigen müssen, ob sie gemeinsam angreifen oder sich zurückziehen, kommuniziert wird durch zuverlässige Boten. Einige der Kommandeure können aber Verräter sein, die den Erfolg verhindern wollen, dies soll gerade im byzantinischen Reich (395 – 1453) ein großes Problem gewesen sein.

Befehlshaber
Generäle

Im Modell von Lamport gibt es einen *Befehlshaber* und $n-1$ weitere *Generäle*. Der Befehlshaber erteilt einen Befehl (Angriff oder Rückzug, 1 oder 0), und durch weitere Kommunikation sollen sich alle, der Befehlshaber und die anderen Generäle, auf eine Aktion einigen, wobei sie sich gegenseitig informieren, welchen Befehl sie bekommen haben. Wenn der Befehlshaber selbst ein Verräter ist, dann wird er den Generälen sich widersprechende Befehle erteilen. Wenn ein anderer General ein Verräter ist, dann wird er teilweise darüber lügen, welchen Befehl er empfangen hat. Keiner weiß, wer ein Verräter ist.

Das Ziel ist wieder nur die Einigung, also eine übereinstimmende Entscheidung von allen loyalen Generälen, unter der Voraussetzung, dass nicht zu viele Verräter dabei sind. Es ist nicht das Ziel, die Zahl der Verräter zu ermitteln oder sogar die Verräter zu entlarven. Wir können uns auch genauso gut wieder Prozesse vorstellen, die jetzt korrekt oder böswillig inkorrekt arbeiten.

Die Terminierungs- und die Einigungsbedingung für einen Algorithmus gelten wie oben, nur die Integritätsbedingung wird folgendermaßen angepasst.

Integrität (Byz.)

- *Integrität:* Falls der Befehlshaber kein Verräter ist, dann entscheiden alle Generäle sich für seinen Befehl.

$f < \frac{n}{3}$

Schon Lamport hat gezeigt, dass es für $n = 3$ kein korrektes Verfahren gibt, wenn einer ($f = 1$) der drei ein Verräter ist. Im Allgemeinen gilt, dass mehr als zwei Drittel der Generäle korrekt handeln müssen, also $f < \frac{n}{3}$, damit eine Einigung möglich ist. Umgekehrt bedeutet dies, dass für $f \geq \frac{n}{3}$ das Problem unlösbar ist.

Algorithmus von Lamport
Baumstruktur
rekursiv
Mehrheitsregeln

Es gibt einen *Algorithmus von Lamport*, den wir hier aber nicht im Detail vorstellen wollen. Er benötigt $f+1$ Runden, wobei in der ersten Runde der Befehlshaber den Befehl austeilt, in der zweiten Runde jeder General jedem anderen berichtet, welchen Befehl er erhalten hat, und in jeder weiteren Runde jeder General jedem anderen General alles mitteilt, was er in der vorherigen Runde gehört hat, alles wird von jedem General in einer Baumstruktur gespeichert. Es ist deshalb klar, dass die Anzahl bzw. Größe der Nachrichten und damit auch der erforderliche Speicherplatz mit der Anzahl der Runden exponentiell wächst. Die Entscheidung am Ende geschieht rekursiv durch Mehrheitsregeln in dem Baum der gespeicherten Nachrichten. Die Terminierung des Algorithmus ist klar, die Beweise für Einigung und Integrität sind allerdings recht aufwendig und können z. B. bei Lynch (1996) nachgelesen werden.

Beispiel

Als einfachstes mögliches Beispiel betrachten wir vier Generäle, von denen einer ($f = 1$) illoyal ist, die Bedingung $f < \frac{n}{3}$ ist also erfüllt. In Abbildung 4 repräsentieren die Knoten die Generäle, Nummer 0 ist der Befehlshaber und die Nummer 2 (grau) der Verräter. Der Befehlshaber sendet (Runde 1) allen Generälen den Wert 1, und die loyalen von ihnen teilen diesen Wert ihren Kollegen mit (Runde 2), der Verräter will Verwirrung stiften. Damit ist die

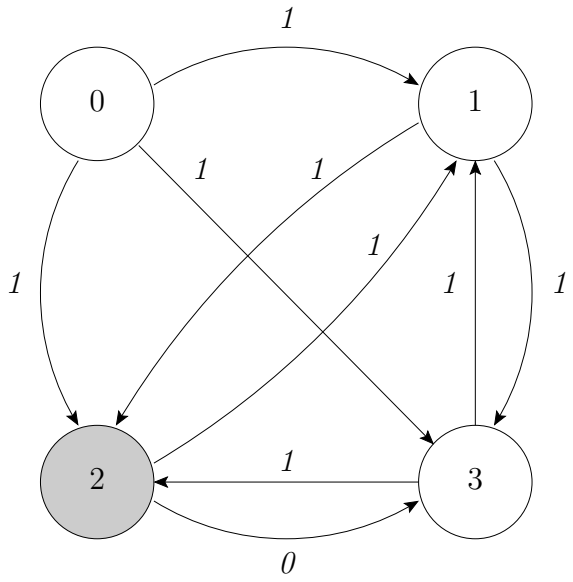

Abbildung 4: Der Befehlshaber 0 und die Generäle 1 und 3 sind loyal, General 2 leider nicht.

Kommunikation hier schon abgeschlossen und die Generäle müssen sich entscheiden. Jeder hat den Befehl des Befehlshabers (Wurzel des Baumes, Tiefe 0) und die weitergeleiteten Informationen (Stimmen) von den anderen (Tiefe 1).

Die loyalen Generäle 1 und 3 entscheiden sich nun für die jeweilige Mehrheit der Stimmen in Tiefe 1 (beide also für 1). In der folgenden Tabelle ist noch einmal die Lage zusammengefasst. Man beachte, dass der Befehlshaber selbst nicht mit abstimmt und sein in Runde 1 übermittelter Befehl nicht direkt einbezogen wird.

General	erhaltene Stimmen	Entscheidung
1	$(1, 1, 1)$	1
3	$(1, 1, 0)$	1

Jetzt verändern wir die Situation, der Befehlshaber selbst ist der Verräter, die drei Generäle sind loyal, siehe Abbildung 5. Der Befehlshaber will verwirren und gibt unterschiedliche Befehle aus (Runde 1), die Generäle tauschen wieder ihre Informationen aus; die folgende Tabelle gibt eine Übersicht nach Runde 2.

General	erhaltene Stimmen	Entscheidung
1	$(0, 1, 0)$	0
2	$(1, 0, 0)$	0
3	$(0, 0, 1)$	0

Die Entscheidung aller Generäle ist nun einheitlich 0, da der Befehlshaber zweimal 0 als Befehl ausgegeben hatte. Wichtig ist nur, dass alle Generäle dasselbe entscheiden, obwohl der verräterische Befehlshaber sie mit widersprüchlichen Befehlen in die Irre führen wollte; das Verfahren hat also in beiden Situationen seinen Zweck erreicht.

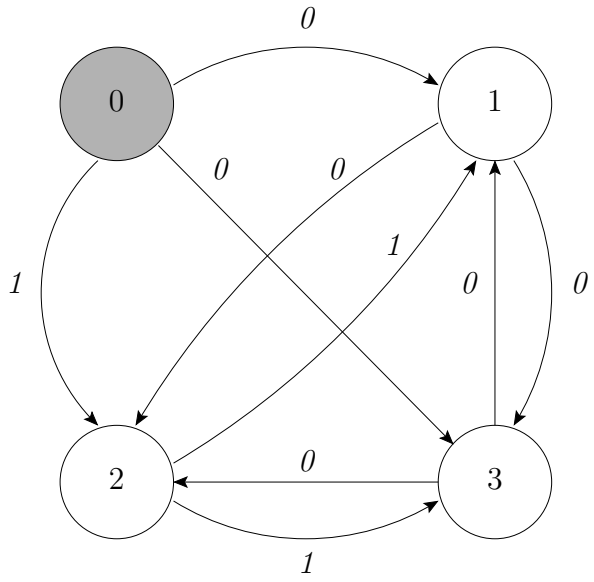

Abbildung 5: Der Befehlshaber 0 ist ein Verräter, alle Generäle sind loyal.

5.4 Anmerkungen zu 8.3 bis 8.5

Wir haben im Kurs *Betriebssysteme und Rechnernetze* gelernt, dass das Transportprotokoll *TCP* eine zuverlässige Kommunikation bietet, d. h. die gesendeten Daten kommen vollständig und in der richtigen Reihenfolge wie gesendet beim Empfänger an, auch wenn einzelne Pakete ausfallen oder zunächst in der falschen Reihenfolge ankommen. Einen Verbindungsabbruch kann TCP aber nicht kompensieren, dazu kann man gegebenenfalls automatisch eine Neuaufnahme der Verbindung anfordern lassen.

Zum Beispiel für die Replikation benötigen wir eine zuverlässige Gruppenkommunikation, dafür gibt es verschiedene Anforderungen und Regeln.

Bei einer verteilten Transaktion müssen Prozesse sich einigen, ob sie ausgeführt werden oder überhaupt nicht. Dafür gibt es die Zwei-Phasen und Drei-Phasen-Protokolle.

5.5 Lernziele

Nach der Bearbeitung der Kurseinheit sollten Sie z. B.

- die Begriffe Verlässlichkeit, Verfügbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit, Ausfall (Failure), Error, Fehler (Fault) und Fehlertoleranz erklären können,
- die verschiedene Ausfallmodelle und die Techniken für Fehlertoleranz nennen können,
- verstehen, was das Einigungsproblem ist und unter welchen Bedingungen eine Einigung möglich ist,
- erklären können, wann das Problem der byzantinischen Generäle lösbar ist,

- erzählen können, wie RPC mit Ausfällen umgeht und welche Semantiken es bei einem Server-Ausfall gibt,
- verschiedene Multicasting- und geordnete Auslieferungsreihenfolgen unterscheiden können,
- die Zwei-Phasen- und Drei-Phasen-Commit-Protokolle erklären können.

6 Sicherheit

Diese Kurseinheit behandelt die *Sicherheit* (security) in verteilten Systemen. Zuerst müssen wir uns mit *Bedrohungen*, *Sicherheitrichtlinien*, und *Sicherheitsmechanismen* beschäftigen. Danach schauen wir genauer in die *Kryptographie* und lernen z. B. symmetrische (*DES*) und asymmetrische (*RSA*) Verschlüsselung kennen. Bevor eine Kommunikation gestartet wird, muss ein sicherer Kanal aufgebaut werden, insbesondere müssen die beiden kommunizierenden Partner sich authentifizieren.

Ganz praktische Fragen werden angesprochen, z. B. wie für Nachrichten die *Vertraulichkeit* und *Integrität* durch Public-Key-Verfahren gesichert wird, wie eine Nachricht digital signiert wird und wie symmetrische und asymmetrische Verfahren zu einer insgesamt effizienteren Methode kombiniert werden können. Auch die Zugriffskontrolle ist hier ein Thema, eines, das allen Computerbenutzern nicht unbekannt sein dürfte.

6.1 Lesehinweise

Bitte lesen Sie Kapitel 9 im Buch.

6.2 Lernziele

Nach der Bearbeitung der Kurseinheit sollten Sie z. B.

- die Begriffe Bedrohung, Sicherheitsrichtlinie, Sicherheitsmechanismus, Kryptographie, Vertraulichkeit, Integrität und digitale Signatur erklären können,
- die beiden Kryptographie-Systeme DES und RSA vorstellen können,
- Authentifizierungsverfahren nennen können,
- erklären können, was ein Sitzungsschlüssel (session key) ist,
- das Erstellen eines Schlüssels mit Hilfe des Diffie-Hellman-Algorithmus durchführen können.

7 Verteilte Dateisysteme und verteilte Web-basierte Systeme

verteilte Dateisysteme
Versions-verwaltungs-systeme
verteilte Web-basierte Systeme
Network File System
World Wide Web

Bisher ging es eher um allgemeine Konzepte von verteilten Systemen. Diese Kurseinheit behandelt nun konkrete Systeme, die in der Praxis eine wichtige Rolle spielen, nämlich *verteilte Dateisysteme*, *Versionsverwaltungssysteme* und *verteilte Web-basierte Systeme*.

Das *Network File System* (NFS) ist ein sehr bekanntes verteiltes Dateisystem, das vor allem in UNIX-artigen Systemen eingesetzt wird. Wir werden insbesondere die Kommunikation durch *RPC*, das Namenssystem und das *Automounting* betrachten. Auch als verteiltes Dateisystem, aber mit ganz anderem Zweck, kann man die Versionsverwaltung ansehen, die sich insbesondere für Programmierer anbietet, die innerhalb einer Arbeitsgruppe die immer wieder veränderten Dateien verwalten und miteinander austauschen müssen.

Verteilte Web-basierte Systeme bauen auf dem *World Wide Web* (WWW) auf. Die Client/Server-Architektur, das Protokoll HTTP zur Kommunikation und die Adressierung via URL sind uns sicherlich bekannt.

7.1 Lesehinweise

Bitte lesen Sie die Abschnitte 11.1 bis 11.4.1 und 12.1 bis 12.4 im Buch sowie hier Abschnitt 7.2 über Versionsverwaltung.

7.2 Versionsverwaltung

Für den gemeinsamen Zugriff auf Dateien gibt es neben der Forderung nach Konsistenz (dass es also immer eine gültige Fassung der Datei gibt) noch weitere Forderungen, die sich aus dem Prozess der Zusammenarbeit auf gemeinsam genutzten Dateien ergeben:

Isolation Benutzer möchten isoliert voneinander an Dateien arbeiten können, so als wären sie alleinige Besitzer.

Integration Es muss möglich sein, die eigenen Änderungen wieder in die Gruppe zu integrieren.

Aktualität Es soll leicht sein, aktuelle Dateiinhalte zu erhalten oder veraltete zu aktualisieren.

Nachvollziehbarkeit Änderungen sollen nachvollziehbar sein, d. h. alle früheren Zustände sind verfügbar und vergleichbar, alle Veränderungen sind mit Namen und Datum gespeichert.

Gruppenwahrnehmung Benutzer möchten über Aktivitäten von anderen Benutzern informiert werden, um ihre eigenen Aktivitäten entsprechend zu koordinieren.

Repository

Hierfür hat sich die Versionsverwaltung bewährt. Mehrere Versionen von Dateien eines Projekts werden in einem gemeinsamen Datenspeicher, dem sogenannten *Repository* abgelegt. Das Versionsverwaltungssystem unterstützt den Benutzer bei der Auswahl, Erzeugung und Manipulation von Versionen.

Alte Versionen können nicht gelöscht werden, sondern jegliche Veränderung einer Datei resultiert in einer neuen Version der Datei im Repository.

Versionsliste

So entsteht eine Versionsliste, da für jede Version gesagt werden kann, wie die Vorversion verändert wurde, um die neue Version zu erstellen. Damit ist die Nachvollziehbarkeit gewährleistet. Bei einer linearen Ordnung der Versionen ist es außerdem sehr einfach, die aktuellste Version einer Datei zu erhalten: Man muss lediglich das letzte Element der Liste wählen.

Eine Version kann auch mehrere Nachfolgeversionen bekommen, so dass die Versionsliste aufgespalten wird (branch), oder eine neue Version kann auf mehreren Vorversionen basieren, die zu einer gemeinsamen Nachfolgeversion zusammengefasst werden (merge).

Die grundsätzliche Arbeitsweise mit einem solchen Versionsverwaltungssystem beinhaltet drei Phasen:

1. Checkout

   Der Benutzer wählt die zu bearbeitenden Dateien aus dem Repository, das ist normalerweise ein komplettes (Teil-)Projekt, und kopiert sie in seinen lokalen Arbeitsbereich. Diesen Schritt bezeichnet man als *Checkout* (Aktualität).

   Update

   Oder wenn es neue Versionen von anderer Seite gibt, bekannt z. B. durch automatische Benachrichtigungen (Gruppenwahrnehmung), kann der Benutzer seine lokale Kopie des Projekts auf den neusten Stand bringen, dies ist der *Update*-Vorgang (auch Aktualität).

2. Bearbeiten

   Die Dateien werden im lokalen Arbeitsbereich verändert. Der Besitzer des lokalen Arbeitsbereichs ist der einzige, der darauf zugreifen kann, er kann also ungestört lokal verändern und testen (Isolation).

3. Commit

   Nachdem der Bearbeiter seine Aufgabe erledigt hat, überträgt er die veränderten Dateien wieder in das Repository. Dabei werden für die Dateien neue Versionen erzeugt und mit Zeitstempel und weiteren Zusatzinformationen abgespeichert. Diesen Vorgang bezeichnet man auch als *Commit* (Integration). Die neuen Versionen werden mit den Vorgängern verknüpft (Nachvollziehbarkeit).

Concurrent Version System
CVS

Ein System, das nach diesem Muster arbeitet, ist das *Concurrent Version System* CVS. Es gibt frei verfügbare CVS-Client-Systeme für fast alle aktuellen Betriebssysteme, auch welche mit leicht zu bedienender graphischer Benutzeroberfläche, siehe Abbildung 6, oder integriert in eine Entwicklungsplattform wie Eclipse (www.eclipse.org). CVS-Server gibt es im wesentlichen nur für UNIX-artige Systeme.

[6] Um was für eine Art von Projekt mag es sich hier handeln?

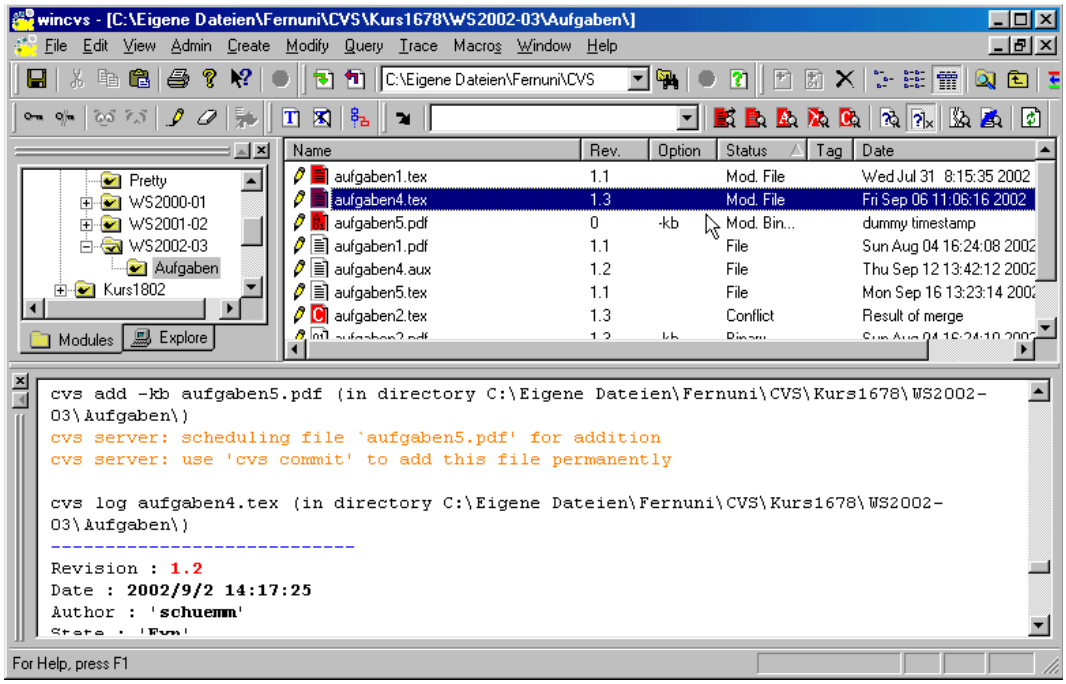

Abbildung 6: Eine Sicht auf ein Projekt,[6] das mit WinCVS verwaltet wird. Zu erkennen sind aktuelle, lokal veränderte und lokal hinzugefügte Dateien sowie ein Merge-Konflikt (s. u.).

Das beschriebene dreiphasige Modell (Checkout oder Update – Bearbeiten – Commit) gleicht im Ablauf einer Bücherei. Auch hier können Benutzer Bücher ausleihen, mit diesen Büchern arbeiten (und hoffentlich nicht zu viele Veränderungen wie Kugelschreibermarkierungen in den Büchern anbringen...), und die Bücher in ihrem aktuellen Zustand zurückgeben. Der Vergleich mit der Bücherei macht ein Problem deutlich: Was kann man tun, wenn zwei Benutzer gleichzeitig mit demselben Buch arbeiten wollen? Der einfachste Weg ist, das Buch nur einmal auszuleihen, also nur einem Benutzer zu erlauben, die Datei auszuchecken. Für andere Benutzer wird die Datei so lange gesperrt. Andere Benutzer müssen mit ihrer Aufgabe solange warten, bis der erste Benutzer die Datei wieder eincheckt. Allerdings kann die Arbeit eines Benutzers beliebig lange dauern; insofern wäre dieses Verhalten für den zweiten Benutzer ziemlich unbefriedigend.

Eine erste Näherung besteht darin, beim Auschecken anzugeben, ob man die Datei nur lesen, oder ob man sie auch verändern möchte. In diesem Modell kann genau ein Benutzer die Datei zum Schreiben auschecken. Beliebig viele andere Benutzer können die Datei zum Lesen auschecken. Auf diese Art und Weise kann z. B. eine Gruppe von Softwareentwicklern schon sehr gut an einem gemeinsamen Projekt arbeiten: Das Projekt wird aufgeteilt in verschiedene Teile, für deren Weiterentwicklung jeweils nur ein Beteiligter zuständig ist. Jeder hat zum Testen eine Kopie des gesamten Projekts auf seinem eigenen Arbeitsplatzrechner, die er auch immer wieder über das Versionsverwaltung aktualisiert, jeder ändert aber nur die Dateien, die in seine exklusive Zuständigkeit fallen. Aber solche Einschränkungen können die Benutzer leicht

frustrieren, deshalb ist in CVS üblicherweise die parallele Bearbeitung erlaubt, siehe weiter unten.

CVS-Kommandos

Schauen wir uns nun zunächst die wichtigsten *Kommandos* des CVS-Systems an:

Auschecken

checkout *projname* kopiert die aktuelle Version des gesamten Projektes namens *projname* in den lokalen Arbeitsbereich (Aktualität, Checkout).

Hinzufügen

add *filename* fügt die in *filename* angegebene Datei zum Repository hinzu. Hierdurch wird zunächst ein Dateieintrag erzeugt. Die eigentliche Datei kann danach mit commit übertragen werden.

Einchecken

commit *filename version comment* erstellt eine neue Version einer Datei auf einem CVS-Server (Commit). Wenn filename ein Verzeichnis ist, so betrifft diese Aktion alle Dateien darin und rekursiv alle Unterverzeichnisse. Die neue Version erhält die Versionsnummer *version* (bequemerweise wird aber meistens automatisch nummeriert) und wird mit dem Speicherungskommentar *comment* versehen. Der Kommentar ist für die spätere Nutzung der Versionen oft enorm wichtig (Nachvollziehbarkeit).

Zustand

status *filename* gibt zurück, ob und inwiefern die lokale Version der Datei von der neuesten Version im Repository abweicht. Die wichtigsten Antworten sind: *Up-to-Date* (die beiden Versionen stimmen überein), *Locally Modified* (die lokale Version wurde verändert), *Locally Added* (die lokale Version wurde hinzugefügt) und *Needs Update* (die Datei wurde von einem anderen Benutzer auf dem Server verändert).

Unterschiede

diff *filename* listet die genauen Unterschiede zwischen zwei Versionen auf, z. B. zwischen der aktuellen und der lokalen Kopie.

Aktualisieren

update *filename* lädt die aktuellste Version der durch *filename* bezeichneten Datei (oder des Verzeichnisses) in den lokalen Arbeitsbereich (Aktualität).

Liste

log *filename* listet für alle bisher erstellten Versionen der Datei filename die Versionsnummern, Änderungsdaten und Speicherungskommentare auf (Nachvollziehbarkeit).

Abzweigung Versionsbaum

branch *name* erzeugt eine Abzweigung aus der Versionsliste, die Liste wird zu einem *Versionsbaum*. Dies kann für große Projekte sinnvoll sein, in denen abgespaltene Versionen weiterentwickelt und gepflegt werden sollen. Die Erfahrung zeigt allerdings, dass solche Abzweigungen schnell als Sackgassen enden, und dass es extrem kompliziert sein kann, die hier vorgenommenen Änderungen später in den Hauptzweig zurück zu integrieren. Empfohlen wird, falls es irgendwie möglich ist, beim Hauptzweig zu bleiben.

Gruppenwahrnehmung

Mit der letzten Forderung an eine Versionsverwaltung, der Gruppenwahrnehmung, ist Wahrnehmung der Aktionen anderer Gruppenmitglieder in

einem gemeinsamen Arbeitsbereich gemeint. Im Kontext der Versionsverwaltung arbeiten Benutzer mit den Daten, die im Repository abgelegt sind. Die Benutzer können aus Sicht des Systems vor allem Dateien auschecken und Dateien einchecken. Gruppenwahrnehmung bedeutet jetzt, dass Benutzer, die am selben Projekt arbeiten, über die Aktivitäten ihrer Kollegen informiert werden. In CVS gibt es hierfür einen allgemeinen Mechanismus, der im Anschluss an commit-Aktionen ausgeführt werden kann. Zum Beispiel lässt er sich so konfigurieren, dass immer, wenn ein Benutzer eine neue Version erstellt, interessierte Benutzer hiervon über E-Mail in Kenntnis gesetzt werden. Die Benutzer abonnieren dazu den Projektarbeitsbereich, indem sie ihre E-Mail-Adresse in eine Notifikationsliste eintragen. Das System verschickt nach jeder Änderung im Projektarbeitsbereich dann Informationen über die Änderung (welche Datei eingecheckt wurde und welchen Kommentar der Benutzer dazu verfasst hat) an alle Abonnenten.

parallele Bearbeitung

Die parallele Bearbeitung von Dateien wird bei CVS üblicherweise erlaubt: wenn mehrere Benutzer dasselbe Projekt ausgecheckt haben, dann können auch alle im Prinzip Änderungen wieder einchecken. Damit dies nicht so leicht zu Inkonsistenzen führt, sind gewisse *Sicherheitsvorkehrungen* eingebaut.

Sicherheitsvorkehrungen

So kann man nur eine neue Version einchecken, die auf der gerade aktuellen Version basiert. CVS überprüft dies, indem beim `commit` die bisher lokal vorhandene Versionsnummer mit der aktellen des Repositories verglichen wird. Stimmen die Nummern überein, wird die Operation zugelassen, anderenfalls meldet sich CVS mit einem Fehler:

```
cvs commit: Up-to-date check failed for `...'
cvs [commit aborted]: correct above errors first!
```

Dies passiert also dann, wenn mehrere Benutzer eine Datei bearbeiten, einer seine Änderungen bereits eingecheckt hat und ein Zweiter nun ebenfalls einchecken möchte. Dieser Benutzer wird nun aufgefordert, seine lokale Version zu aktualisieren. CVS unterstützt ihn dabei, indem beim `update` nicht etwa die lokale Datei einfach durch die aktuelle ersetzt, sondern eine gemischte Version erstellt wird, diesen Vorgang nennen wir *Merge*. Es gibt also kein `merge`-Kommando beim CVS, sondern der Merge ist ein Nebeneffekt der `update`-Operation. Ein Merge geschieht auch nicht, wie man vielleicht meinen könnte, bei der `commit`-Operation und somit direkt im Repository, denn dieses wäre doch zu riskant, sondern nur über das `update`-Kommando in der lokalen Kopie.

Merge

Sollte beim lokalen Mischen ein Problem auftreten, was nicht automatisch gelöst werden kann, dann nennen wir dies einen *Merge-Konflikt*. Als Ergebnis des `update`-Kommandos bekommen wir eine lokale Datei, in der einige Stellen von CVS besonders gekennzeichnet und die unterschiedlichen Zeilen aus beiden Fassungen enthalten sind. In diesem Fall muss der Benutzer in eigener Verantwortung zuerst lokal eine sinnvolle Version der Datei erstellen und diese erst dann über das `commit`-Kommando in das Repository stellen.

Merge-Konflikt

Solch ein Merge, eine Integration von verschiedenen Versionen einer Datei, funktioniert überhaupt nur für sogenannte Text-Dateien: Der Typ einer Datei (Text oder binär) wird beim Erzeugen (automatisch) festgelegt, und nur Text-

dateien können inhaltlich verglichen und gegebenenfalls vermischt werden. Für Binärdateien kann man nur feststellen, ob zwei Versionen vollständig identisch sind oder nicht.

zeilenorientiert

Außerdem muss man wissen, dass CVS für Textdateien *zeilenorientiert* arbeitet: Bei Versionsvergleichen werden Abschnitte von identischen Zeilen gesucht, zwischen solchen identischen Abschnitten liegen dann veränderte Zeilen, wobei es keine Rolle spielt, wie viel an einer Zeile verändert wurde.

Dreiwegevergleich

Der automatische Merge von CVS im Verlauf der update-Operation benutzt nun den sogenannten Dreiwegevergleich: Aus den beiden in Konflikt stehenden Versionen (die aktuelle im Repository und die lokal veränderte) sowie der gemeinsamen Vorgängerversion wird ermittelt, welche Zeilen jeweils gelöscht, hinzugefügt und verändert wurden. Solche Modifikationen stehen dann nicht miteinander in Konflikt, wenn sie an verschiedenen Stellen der Datei vorgenommen wurden oder wenn sie identisch sind. Wenn dies aber nicht der Fall ist, ergibt das den oben erwähnten Merge-Konflikt.

Im Prinzip ist dieses Vergleichsverfahren durchaus noch verbesserungsfähig. Da nur Zeilen verglichen werden, führen zwei unterschiedliche Änderungen in derselben Zeile automatisch zu einem Merge-Konflikt, erkennbar im Ergebnis an den gekennzeichneten, nebeneinander stehenden unterschiedlich veränderten Zeilen. Ein Vergleich auf Wortebene könnte möglicherweise das eigentlich gewünschte Ergebnis erzeugen, aber garantieren lässt sich so etwas natürlich nicht. Ohne Kenntnis der Semantik wird man nie in der Lage sein, eine semantisch korrekte Integration herzustellen.

unverzichtbares Werkzeug

Zusammenfassend lässt sich sagen, dass Versionsverwaltung heute ein *unverzichtbares Werkzeug* insbesondere bei der Softwareentwicklung aber auch bei anderen über längere Zeit laufenden Projekten ist und jedem Informatiker geläufig sein sollte.

7.3 Lernziele

Nach der Bearbeitung der letzten Kurseinheit sollten Sie

- die Architektur von NFS im UNIX-System, Hard Links, Soft Links, Mounten, Mountpunkt, Automounter und die Transparenz von NFS erklären können,
- berichten können, welche Informationen man über das exportierte Verzeichnis auf dem NFS-Server haben muss und was der Administrator auf Client- wie auf Server-Seite tun muss, damit NFS auch funktioniert,
- das Google-Dateisystem vorstellen können,
- ein Webservercluster vorstellen und erklären können, wie eine inhaltsbewusste Anforderungsverteilung funktioniert und welche Unterschiede sie im Vergleich zum TCP-Handoff aufweist,
- erklären können, was SOAP ist, wie eine HTTP-Nachricht aussieht und was der Unterschied zwischen HTTP 1.0 und 1.1 ist,

- den Sinn und Zweck von Versionsverwaltungssytemen erläutern und die wichtigsten Befehle und Operationen von CVS angeben können.

8 Fehlerlisten

Leider sind in der deutschen Version des Buchs recht viele Übersetzungsfehler enthalten. Oft fehlt auch die Konsistenz, derselbe englische Begriff sollte natürlich einheitlich mit demselben passenden deutschen Begriff übersetzt werden, was nicht immer gelungen ist.

Hier also unsere Zusammenstellung, die keinen Anspruch auf Vollständigkeit erhebt. Ergänzungsvorschläge sind willkommen.

8.1 Buchkapitel 1 bis 4

Seite	Absatz	Zeile	deutsche Version	Korrektur oder anderer Übersetzungsvorschlag
63	3	3	Bezeichner $id\ k$	Bezeichner $id \geq k$
109	1	4	Dokumentenverband	Verbunddokument
114	2	9	Clients werden niemals	Cookies werden niemals
115	Abb. 3.12		Verteilte Daten/ Datenbanksystem	Verteiltes Datei-/ Datenbanksystem
118	2	2	In diesem Fall wird dem Server ursprünglich	In diesem Fall wird dem Server anfangs
145	5	5	früher als ein später gesendetes ankommen	früher als ein vorher gesendetes ankommen
151	2	2	anstatt auf den Wert der Variablen	anstatt der Wert der Variablen

8.2 Buchkapitel 5 bis 7

Seite	Absatz	Zeile	deutsche Version	Korrektur oder anderer Übersetzungsvorschlag
208	1	2	Funktionseinheit	Entität
208	4	1	Funktionseinheit	Entität
209	3	1	An Entitäten kann gearbeitet werden	Eine Entität kann angesprochen werden
209	4	1	Zur Bearbeitung einer Entität	Um eine Entität anzusprechen
210	3	7	immer auf die gleiche Entität	immer auf dieselbe Entität
212	2	4	Funktionseinheit	Entität
218	4	7	mit denen seines Vorgängers	mit denen seines Nachfolgers
227	5	2	für die gleiche Entität	für dieselbe Entität
228	1	10	der Stamm eines Namensraums	die Wurzel eines Namensraums
228	3	4	NR_2	NS_2
228	3	4	NR_1	NS_1
228	3	5	NR_2	NS_2
229	2	1	Der Name des Montpoint	Der Name des Mountpoint des fremden Namenraums
230	1	4	dabei abwechselnd	daraufhin
238	2	9	`flits.cs.vu.nl`	`flits.cs.vu.nl.`
275	4	4	gleichzeitig	nebenläufig
Kapitel 7 ab S. 305			Verlässlichkeit	Zuverlässigkeit
308	Überschrift		Stufenlose Konsistenz	Stetige Konsistenz
313	Kasten komplett			Definition Seite 18
315	5	3	parallel	nebenläufig
315	Kasten komplett			Definition Seite 20
337	4	1	Wenn $TW[i,j]$ die von Server S_i vorgenommenen Schreibvorgänge sind, die ... ... gilt Folgendes:	Sei $TW[i,j]$ die Summe der Gewichte der von S_i ausgeführten Schreiboperationen mit Ursprung S_j:
339	Titel		Urbildbasierte Protokolle	Primärbasierte Protokoll
341	Titel	1	Protokolle für lokale Schreibvorgänge	Protokolle für lokale Schreiben

8.3 Buchkapitel 8 bis 12

Seite	Absatz	Zeile	deutsche Version	Korrektur oder anderer Übersetzungsvorschlag
356	3	4	Ein Fehler (Error) ist ein	Ein Error ist ein
356	4	1	Die Ursache eines Ausfa–lls wird als Störung (. . .)	Die Ursache eines Error wird als Fehler (Fault)
356	5	1	Der Aufbau zuverlässiger	Der Aufbau verlässlicher
356	5	1	Kontrolle von Störungen	Kontrolle von Fehlern
356	5	5	bestimmten Störungen	bestimmten Fehler
356	6	1	Störungen werden	Fehler werden
356	6	3	tritt die Störung nicht	tritt der Fehler nicht
356	7	1	wiederkehrende Störung	wiederkehrender Fehler
356	7	3	wiederkehrende Störung	wiederkehrender Fehler
356	8	1	permanente Störung	permanenter Fehler
356	8	3	permanente Störung	permanenter Fehler
357	8.1.2		Fehlermodelle	Ausfallmodelle
357	1	7	Störungsgrund	Fehler
359	5	3	Störungen	Fehlern
361	4	1	einen gestörten Prozess	einen fehlerhaften Prozess
362	3		Lineare und hierarchische Gruppen	Gleichberechtigte und hierarchische Gruppen
362	Abb. 8.3		Flache Gruppe	gleichberechtigte Gruppe
362	Abb. 8.3		in einer linearen	in einer gleichberechtigten
363	5	4	Unveränderlich muss ein Prozess die Initiative ergreifen	Es muss immer ein Prozess die Initiative ergreifen
364	8.2.3 (oft)		Übereinstimmung	Einigung
365	3	5	Prozessor	Prozess
365	4	1, 2	gebunden	beschränkt
365	8	8	gebunden	beschränkt
381	Zwischentitel		Virtuelle Gleichzeitigkeit	Virtuelle Synchronität
382	4	2	m nicht an alle Prozesse	m an gar keinen Prozess
386	6	2	Ohne Verlust der Allgemeinheit	Ohne Beschränkung der Allgemeinheit (O.B.d.A.)
393	Code a	15	}	exit; }
393	Code b	2	alle ankommenden	irgendeine ankommende
413	3.	8.	Einbringen (Fabrikation)	Fälschen (Fabrikation)
430	1	8	In RSA werden die privaten und privaten Schlüssel	In RSA werden der private und der öffentliche Schlüssel
450	Titel	1	Zugriffssteuerung	Zugriffskontrolle
Ersetze Zugriffssteuerung durch Zugriffskontrolle				
Ersetze Zugriffssteuerungsliste durch Zugriffskontrollliste				
Ersetze Zugriffssteuerungsmatrix durch Zugriffskontrollmatrix				
Ersetze Abbildung 8.4 im Buch durch Abbildung 3 in diesem Text				
Ersetze Abschnitt 8.2.3 im Buch durch Abschnitt 5.3 in diesem Text				