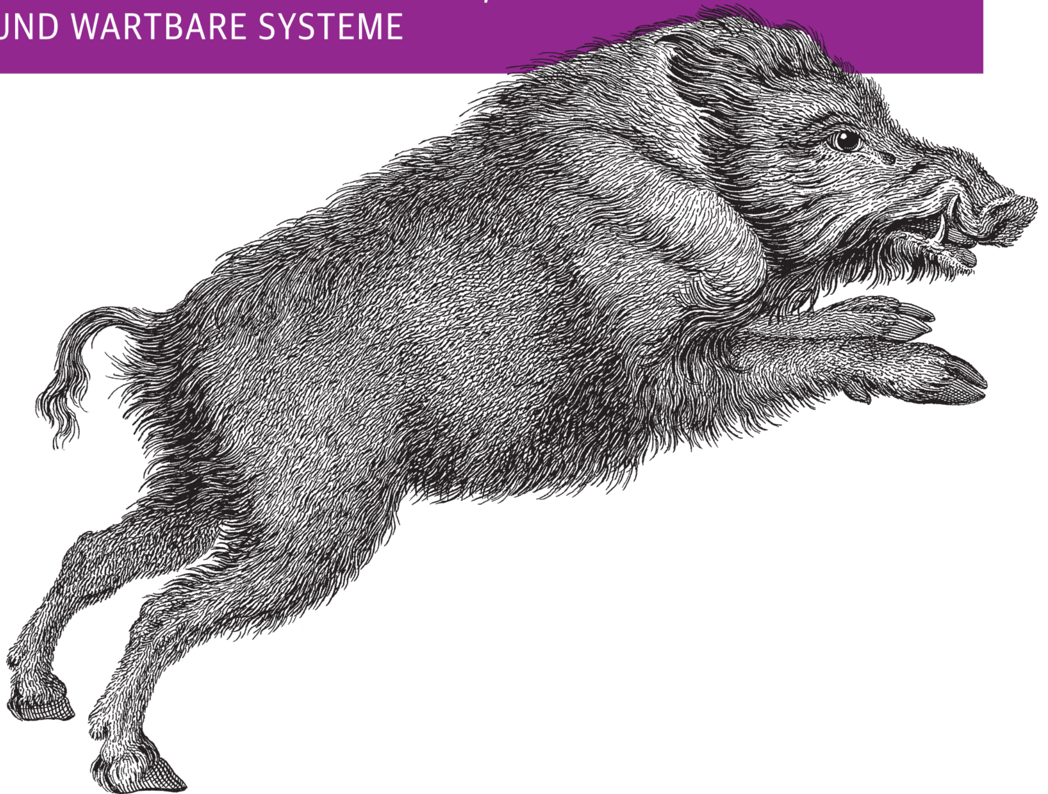
O'REILLY®

Datenintensive Anwendungen designen

Konzepte für zuverlässige, skalierbare und wartbare Systeme



Martin Kleppmann
Übersetzung von Frank Langenau

Papier
plus+
PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern –
können Sie auch das entsprechende E-Book im PDF-Format
herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus+:

www.oreilly.plus

Datenintensive Anwendungen designen

Konzepte für zuverlässige,
skalierbare und wartbare Systeme

Martin Kleppmann

Übersetzung aus dem Englischen
von Frank Langenau

O'REILLY®

Martin Kleppmann

Lektorat: Alexandra Follenius
Übersetzung: Frank Langenau
Korrektorat: Claudia Lötschert, www.richtiger-text.de
Satz: III-satz, www.drei-satz.de
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, www.oreal.de
Druck und Bindung: C.H.Beck, Nördlingen

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-075-5
PDF 978-3-96010-183-3
ePub 978-3-96010-184-0
mobi 978-3-96010-185-7

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

1. Auflage 2019
Copyright © 2019 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable,and Maintainable Systems*, ISBN 978-1-449-37332-0 © 2017 Martin Kleppmann. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

5 4 3 2 1 0

Technologie übt in unserer Gesellschaft eine große Macht aus. Daten, Software und Kommunikation können missbraucht werden: um ungerechte Machtstrukturen tiefer zu verankern, Menschenrechte auszuhöhlen und Eigeninteressen zu schützen. Aber sie können auch für gute Zwecke genutzt werden: um Minderheiten Gehör zu verschaffen, Chancen für alle zu eröffnen und Katastrophen abzuwenden. Dieses Buch ist jedem gewidmet, der sich für das Gute einsetzt.

Inhalt

Einleitung . . . . **XIII**

Teil I: Grundlagen von Datensystemen

1 Zuverlässige, skalierbare und wartbare Anwendungen **3**
Gedanken zu Datensystemen 4
Zuverlässigkeit 6
Hardwarefehler 7
Softwarefehler 9
Menschliche Fehler 10
Wie wichtig ist Zuverlässigkeit? 11
Skalierbarkeit 11
Lasten beschreiben 11
Performance beschreiben. 14
Konzepte zur Bewältigung von Belastungen 18
Wartbarkeit 20
Betriebsfähigkeit: Den Betrieb erleichtern. 21
Einfachheit: Komplexität im Griff. 22
Evolvierbarkeit: Änderungen erleichtern. 23
Zusammenfassung 24

2 Datenmodelle und Abfragesprachen **29**
Relationales Modell vs. Dokumentmodell 30
Die Geburt von NoSQL. 31
Die objektrelationale Unverträglichkeit. 32
n:1- und n:n-Beziehungen 35
Wiederholen Dokumentdatenbanken die Geschichte?. 38
Heutige relationale Datenbanken vs. Dokumentdatenbanken 41

Abfragesprachen für Daten 45
Deklarative Abfragen im Web 47
MapReduce-Abfragen 49
Graphen-ähnliche Datenmodelle 52
Property-Graphen 53
Die Abfragesprache Cypher 55
Graph-Abfragen in SQL 57
Triple-Stores und SPARQL 59
Das Fundament: Datalog 63
Zusammenfassung 66

3 Speichern und Abrufen 73
Datenstrukturen, auf denen Ihre Datenbank beruht 74
Hash-Indizes 76
SSTables und LSM-Bäume 80
B-Bäume 85
B-Bäume und LSM-Bäume im Vergleich 89
Andere Indizierungsstrukturen 91
Transaktionsverarbeitung oder Datenanalyse? 97
Data-Warehousing 98
Sterne und Schneeflocken: Schemas für die Analytik 101
Spaltenorientierte Speicherung 103
Spaltenkomprimierung 105
Sortierreihenfolge in spaltenorientierten Datenbanken 107
In spaltenorientierte Datenbanken schreiben 108
Aggregation: Datenwürfel und materialisierte Sichten 109
Zusammenfassung 111

4 Codierung und Evolution 119
Formate für das Codieren von Daten 120
Sprachspezifische Formate 121
JSON, XML und binäre Varianten 122
Thrift und Protocol Buffers 125
Avro 130
Die Vorzüge von Schemas 136
Datenflussmodi 137
Datenfluss über Datenbanken 137
Datenfluss über Dienste: REST und RPC 140
Datenfluss beim Nachrichtenaustausch 146
Zusammenfassung 148

Teil II: Verteilte Daten

5 Replikation **159**
Leader und Follower 160
Synchrone und asynchrone Replikation 161
Neue Follower einrichten 163
Knotenausfälle behandeln 164
Implementierung von Replikationsprotokollen 167
Probleme mit der Replikationsverzögerung 170
Die eigenen Schreiboperationen lesen 171
Monotones Lesen 173
Präfixkonsistenz 175
Lösungen für Replikationsverzögerung 176
Multi-Leader-Replikation 177
Einsatzfälle für Multi-Leader-Replikation 177
Schreibkonflikte behandeln 180
Topologien für Multi-Leader-Replikation 185
Replikation ohne Leader 187
In die Datenbank schreiben, wenn ein Knoten ausgefallen ist 188
Grenzen der Quorumkonsistenz 191
Sloppy Quoren und Hinted Handoff 194
Parallele Schreibvorgänge erkennen 196
Zusammenfassung 204

6 Partitionierung **211**
Partitionierung und Replikation 212
Partitionierung von Schlüssel-Wert-Daten 213
Partitionierung nach Schlüsselbereich 214
Nach dem Hashwert des Schlüssels partitionieren 215
Schiefe Arbeitslasten und Entlastung von Hotspots 217
Partitionierung und Sekundärindizes 218
Sekundärindizes nach Dokument partitionieren 219
Sekundärindizes nach Begriff partitionieren 220
Rebalancing – Partitionen gleichmäßig belasten 222
Strategien für Rebalancing 222
Operationen: Automatisches oder manuelles Rebalancing 226
Anfragen weiterleiten 227
Parallele Abfrageausführung 229
Zusammenfassung 230

7 Transaktionen **235**
Das schwammige Konzept einer Transaktion 236
Die Bedeutung von ACID 237
Einzelobjekt- und Multiobjektoperationen 242
Schwache Isolationsstufen 248
Read Committed 249
Snapshot-Isolation und Repeatable Read 252
Verlorene Updates verhindern 258
Schreibversatz und Phantome 262
Serialisierbarkeit 268
Tatsächliche serielle Ausführung 269
Zwei-Phasen-Sperrverfahren (2PL) 274
Serialisierbare Snapshot-Isolation (SSI) 278
Zusammenfassung 284

8 Die Probleme mit verteilten Systemen **291**
Fehler und Teilausfälle 292
Cloud-Computing und Supercomputing 293
Unzuverlässige Netzwerke 296
Netzwerkfehler in der Praxis 298
Fehler erkennen 299
Timeouts und unbeschränkte Verzögerungen 300
Synchrone und asynchrone Netzwerke 304
Unzuverlässige Uhren 307
Monotone Uhren und Echtzeituhren 308
Uhrensynchronisierung und Genauigkeit 309
Sich auf synchronisierte Uhren verlassen 311
Prozesspausen 316
Wissen, Wahrheit und Lügen 321
Die Wahrheit wird von der Mehrheit definiert 322
Byzantinische Fehler 326
Systemmodell und Realität 329
Zusammenfassung 333

9 Konsistenz und Konsens **343**
Konsistenzgarantien 344
Linearisierbarkeit 346
Was macht ein System linearisierbar? 347
Auf Linearisierbarkeit setzen 352
Linearisierbare Systeme implementieren 355
Die Kosten der Linearisierbarkeit 358

Ordnungsgarantien . . . 362
Ordnung und Kausalität . . . 363
Ordnung nach Sequenznummern . . . 368
Total geordneter Broadcast . . . 372
Verteilte Transaktionen und Konsens . . . 377
Atomarer Commit und Zwei-Phasen-Commit (2PC) . . . 379
Verteilte Transaktionen in der Praxis . . . 386
Fehlertoleranter Konsens . . . 391
Mitgliedschafts- und Koordinationsdienste . . . 397
Zusammenfassung . . . 401

Teil III: Abgeleitete Daten

10 Stapelverarbeitung . . . 417
Stapelverarbeitung mit Unix-Tools . . . 419
Einfache Protokollanalyse . . . 419
Die Unix-Philosophie . . . 422
MapReduce und verteilte Dateisysteme . . . 426
MapReduce-Jobausführung . . . 428
Reduce-seitige Verknüpfungen und Gruppierungen . . . 432
Map-seitige Verknüpfungen . . . 438
Die Ausgabe von Stapel-Workflows . . . 441
Hadoop im Vergleich mit verteilten Datenbanken . . . 446
Jenseits von MapReduce . . . 450
Zwischenzustände materialisieren . . . 451
Graphen und iterative Verarbeitung . . . 456
Höhere APIs und Sprachen . . . 460
Zusammenfassung . . . 462

11 Stream-Verarbeitung . . . 471
Ereignisströme übertragen . . . 472
Nachrichtensysteme . . . 473
Partitionierte Protokolle . . . 479
Datenbanken und Streams . . . 485
Systeme synchron halten . . . 486
Erfassen von Datenänderungen . . . 487
Event Sourcing . . . 491
Zustand, Streams und Unveränderlichkeit . . . 494

Streams verarbeiten 499
Anwendungen der Stream-Verarbeitung 501
Überlegungen zur Zeit 505
Stream-Joins 509
Fehlertoleranz 513
Zusammenfassung 517

12 Die Zukunft von Datensystemen 527
Datenintegration 528
Spezialisierte Tools durch Ableiten von Daten kombinieren 528
Batch- und Stream-Verarbeitung 533
Die Entflechtung von Datenbanken 538
Zusammenstellung verschiedener Datenspeichertechniken 539
Anwendungen datenflussorientiert entwickeln 544
Abgeleitete Zustände beobachten 550
Auf der Suche nach Korrektheit 557
Das Ende-zu-Ende-Argument für Datenbanken 558
Durchsetzung von Einschränkungen 564
Zeitnähe und Integrität 567
Vertrauen ist gut, Kontrolle ist besser 572
Das Richtige tun 578
Prädiktive Analytik 579
Datenschutz und Nachverfolgung 582
Zusammenfassung 590

13 Glossar 601

Index 609

Einleitung

Computing ist Popkultur. [...] Die Popkultur verachtet die Geschichte. In der Popkultur dreht sich alles um Identität und das Gefühl mitzumachen. Sie hat nichts mit Zusammenarbeit, Vergangenheit oder Zukunft zu tun – es geht um das Leben in der Gegenwart. Ich denke, das Gleiche gilt für die meisten Leute, die Code für Geld schreiben. Sie haben keine Ahnung, woher [ihre Kultur kommt].

– **Alan Kay**, im Interview mit *Dr Dobb's Journal* (2012)

Wenn Sie in den letzten Jahren in der Softwareentwicklung gearbeitet haben und hier vor allem mit Server- und Backendsystemen, dann sind Sie wahrscheinlich mit einer Fülle von Schlagwörtern rund um die Speicherung und Verarbeitung von Daten bombardiert worden. NoSQL! Big Data! Skalierbarkeit! Sharding! Konsistenz! ACID! CAP-Theorem! Cloud-Dienste! MapReduce! Echtzeit!

Das letzte Jahrzehnt hat viele interessante Entwicklungen hervorgebracht, und zwar bei Datenbanken, in verteilten Systemen und in der Art und Weise, wie wir Anwendungen darauf aufbauen. Für diese Entwicklungen gibt es vielfältige Triebkräfte:

- Internetfirmen wie Google, Microsoft, Amazon, Facebook, LinkedIn, Netflix und Twitter verarbeiten riesige Mengen an Daten und Datenverkehr, was sie dazu zwingt, neue Tools zu entwickeln, die in dieser Größenordnung effizient arbeiten.
- Unternehmen müssen agil sein, Hypothesen kostengünstig testen und schnell auf neue Markterkenntnisse reagieren, indem sie die Entwicklungszyklen kurz und die Datenmodelle flexibel halten.
- Freie und Open-Source-Software ist sehr erfolgreich geworden und wird heute in vielen Umgebungen kommerzieller oder intern entwickelter Software vorgezogen.
- Während sich die Taktfrequenz der Prozessoren kaum mehr erhöht, sind Multi-Core-Prozessoren zum Standard avanciert, und Netzwerke werden immer schneller. Die Entwicklung strebt also in Richtung Parallelverarbeitung.

- Dank Infrastructure as a Service (IaaS) wie zum Beispiel Amazon Web Services können Sie selbst in einem kleinen Team Systeme aufbauen, die über mehrere Computer und sogar über mehrere geografische Regionen hinweg verteilt sind.
- Bei vielen Diensten erwartet man heutzutage, dass sie hochverfügbar sind. Längere Ausfallzeiten aufgrund von Störungen oder Wartungsarbeiten sind kaum noch tragbar.

Datenintensive Anwendungen erweitern die Grenzen des Machbaren, indem sie diese technologischen Entwicklungen nutzen. Eine Anwendung bezeichnen wir als *datenintensiv*, wenn sie vorrangig Probleme in Bezug auf Daten – die Menge der Daten, ihre Komplexität oder die Geschwindigkeit, mit der sie sich verändern – zu lösen hat, im Unterschied zu *rechenintensiv*, wenn die CPU-Takte einen Engpass darstellen.

Die Werkzeuge und Techniken, mit denen datenintensive Anwendungen Daten speichern und verarbeiten, haben sich diesen Änderungen schnell angepasst. Neue Arten von Datenbanksystemen (»NoSQL«) haben viel Aufmerksamkeit erregt, doch auch Nachrichtenwarteschlangen, Caches, Suchindizes, Frameworks für Batch- und Stream-Verarbeitung sowie verwandte Techniken sind auch sehr wichtig. Viele Anwendungen kombinieren diese Techniken.

Die diesbezüglichen Schlagwörter kennzeichnen die Begeisterung für die neuen Möglichkeiten, was prinzipiell zu begrüßen ist. Allerdings brauchen wir als Softwareentwickler und -architekten ein technisch genaues und präzises Verständnis für die verschiedenen Techniken und deren Abwägungen, wenn wir gute Anwendungen erstellen wollen. Für dieses Verständnis dürfen wir uns nicht mit Schlagwörtern begnügen, sondern müssen tiefer eintauchen.

Glücklicherweise stehen hinter diesen rasanten technischen Veränderungen dauerhafte Prinzipien, die ihre Gültigkeit behalten, unabhängig davon, welche Version eines bestimmten Tools Sie verwenden. Wenn Sie diese Prinzipien verstehen, können Sie sehen, wozu jedes Werkzeug passt, wie Sie es zweckmäßig einsetzen und wie Sie seine Fallstricke vermeiden. Hier kommt dieses Buch ins Spiel.

Es soll Ihnen helfen, sich in der vielfältigen und sich schnell verändernden Landschaft der Techniken zum Verarbeiten und Speichern von Daten zurechtzufinden. Dabei ist das Buch weder ein Tutorial für ein bestimmtes Tool noch ein Lehrbuch voller trockener Theorie. Vielmehr sehen wir uns Beispiele erfolgreicher Datensysteme an: Techniken, die die Grundlage vieler bekannter Anwendungen bilden und die täglich den Anforderungen an Skalierbarkeit, Leistung und Zuverlässigkeit in der Produktion gerecht werden müssen.

Wir dringen in die Interna dieser Systeme ein, nehmen ihre Schlüsselalgorithmen auseinander und diskutieren ihre Prinzipien und Abwägungen, die sie treffen müssen. Dabei streben wir nach zweckmäßigen Methoden, um über Datensysteme

nachzudenken – nicht nur darüber, *wie* sie funktionieren, sondern auch, *warum* sie auf diese Weise funktionieren und welche Fragen wir stellen müssen.

Nachdem Sie dieses Buch gelesen haben, befinden Sie sich in einer komfortablen Position, um zu entscheiden, welche Technik für welchen Zweck geeignet ist, und um zu verstehen, wie sich die Tools kombinieren lassen, um die Grundlage für eine gute Anwendungsarchitektur zu bilden. Zwar werden Sie Ihre eigene Datenbank noch nicht von Grund auf neu erstellen können, doch zum Glück ist das auch kaum notwendig. Allerdings werden Sie ein gutes Gespür dafür entwickeln, was Ihre Systeme im Verborgenen tun, sodass Sie über ihr Verhalten nachdenken, gute Entwurfsentscheidungen treffen und eventuelle Probleme aufspüren können.

Wer sollte dieses Buch lesen?

Wenn Sie Anwendungen entwickeln, die Daten auf einer Art Server oder Backend speichern oder verarbeiten, und Ihre Anwendungen das Internet nutzen (zum Beispiel Webanwendungen, mobile Apps oder Sensoren mit Internetanschluss), dann ist dieses Buch genau richtig für Sie.

Die Zielgruppe dieses Buchs sind Softwareentwickler, Softwarearchitekten und technische Führungskräfte, die gerne programmieren. Es ist vor allem relevant, wenn Sie Entscheidungen über die Architektur der Systeme treffen müssen, an denen Sie arbeiten – wenn Sie zum Beispiel Tools für ein bestimmtes Problem auswählen und herausfinden müssen, wie Sie sie am besten anwenden. Doch selbst wenn Sie keinen Einfluss auf die Auswahl Ihrer Tools haben, hilft Ihnen das Buch, deren Stärken und Schwächen besser zu verstehen.

Sie sollten einige Erfahrungen im Erstellen von webbasierten Anwendungen oder Netzwerkdiensten mitbringen und mit relationalen Datenbanken und SQL vertraut sein. Wenn Sie nichtrelationale Datenbanken und andere datenbezogene Tools kennen, ist das prinzipiell von Vorteil, aber nicht unbedingt erforderlich. Hilfreich ist ein allgemeines Verständnis der gängigen Netzwerkprotokolle wie zum Beispiel TCP und HTTP. Welche Programmiersprache oder welches Framework Sie wählen, ist für dieses Buch nicht entscheidend.

Wenn einer der folgenden Punkte auf Sie zutrifft, wird Ihnen dieses Buch nützlich sein:

- Sie möchten lernen, wie sich Datensysteme skalierbar machen lassen, um beispielsweise Webanwendungen oder mobile Apps mit Millionen von Benutzern zu unterstützen.
- Sie müssen Anwendungen hochverfügbar machen (Ausfallzeiten minimieren) und betriebssicher gestalten.
- Sie suchen nach Möglichkeiten, Systeme langfristig wartungsfreundlicher zu machen, selbst wenn sie wachsen und sich die Anforderungen und Techniken verändern.

- Sie sind von Haus aus neugierig, wie die Dinge funktionieren, und wollen wissen, was bei den großen Websites und Onlinediensten hinter den Kulissen vor sich geht. Dieses Buch nimmt die Interna der verschiedenen Datenbanken und Datenverarbeitungssysteme auseinander, und es macht großen Spaß, die cleveren Überlegungen zu erkunden, die in ihr Design eingeflossen sind.

Wenn es um skalierbare Datensysteme geht, hört man manchmal Meinungen wie »Du bist nicht Google oder Amazon. Mach dir keine Gedanken um die Skalierbarkeit und nimm einfach eine relationale Datenbank.« In dieser Aussage liegt etwas Wahres: Ein System zu erstellen für eine Skalierung, die Sie noch nicht brauchen, ist unnützer Aufwand und kann Sie in ein unflexibles Design zwängen. Letztlich handelt es sich dabei um eine Form von vorschneller Optimierung. Allerdings ist es auch wichtig, das richtige Tool für den Job auszuwählen, und verschiedene Techniken zeichnen sich jeweils durch ihre eigenen Stärken und Schwächen aus. Wie wir sehen werden, sind relationale Datenbanken zwar wichtig, aber kein Allheilmittel für den Umgang mit Daten.

Das Themenspektrum dieses Buchs

Dieses Buch liefert keine detaillierten Anleitungen, wie Sie bestimmte Softwarepakete oder APIs installieren oder verwenden. Hierfür gibt es genügend Dokumentationen. Stattdessen erörtern wir die verschiedenen Prinzipien und Abwägungen, die für Datensysteme maßgeblich sind, und wir analysieren die verschiedenen Entwurfsentscheidungen, die für verschiedene Produkte typisch sind.

In den E-Book-Ausgaben sind Links zum vollständigen Text von Onlinequellen enthalten. Zwar wurden alle Links zeitnah zur Veröffentlichung überprüft, doch leider liegt es in der Natur des Webs, dass Links häufig ungültig werden. Falls Sie einen defekten Link finden oder die gedruckte Ausgabe dieses Buchs lesen, können Sie mit einer Suchmaschine nach den Quellen suchen. Bei akademischen Arbeiten können Sie nach dem Titel in Google Scholar nach frei zugänglichen PDF-Dateien suchen. Alternativ finden Sie alle Quellen unter *https://github.com/ept/ddia-references*, wo wir aktuelle Links pflegen.

Wir betrachten vor allem die *Architektur* von Datensystemen und ihre Integration in datenintensive Anwendungen. Der Platz im Buch reicht nicht aus, um Bereitstellung, Betrieb, Sicherheit, Verwaltung und andere Bereiche zu behandeln – solchen komplexen und wichtigen Themen könnten wir nicht gerecht werden, wenn wir sie zu oberflächlichen Randbemerkungen in diesem Buch machen. Sie verdienen eigene Bücher.

Viele der im Buch beschriebenen Techniken lassen sich dem Schlagwort *Big Data* zuordnen. Allerdings ist der Begriff »Big Data« so überstrapaziert und ziemlich schwammig definiert, dass er für eine ernsthafte technische Diskussion nicht viel taugt. Dieses Buch verwendet weniger zweideutige Begriffe, wie zum Beispiel Ein-

zelknotensysteme vs. verteilte Systeme oder online/interaktiv vs. offline/Batch-Verarbeitungssysteme.

Dieses Buch betont *Freie Software* und *Open-Source-Software*, denn wenn man Quellcode selbst lesen, ändern und ausführen kann, versteht man viel besser, wie etwas im Detail funktioniert. Bei offenen Plattformen ist auch die Gefahr einer Anbieterabhängigkeit geringer. Wo es passend erscheint, diskutieren wir aber auch proprietäre Software (Closed-Source-Software, Software as a Service oder firmeninterne Software, die lediglich in der Literatur beschrieben, aber nicht als Code veröffentlicht wird).

Gliederung dieses Buchs

Dieses Buch ist in drei Teile gegliedert:

1. In *Teil I* beschäftigen wir uns mit den prinzipiellen Konzepten, die dem Design von datenintensiven Anwendungen zugrunde liegen. Zuerst erläutert Kapitel 1, was wir eigentlich anstreben: Zuverlässigkeit, Skalierbarkeit und Wartbarkeit, welche Überlegungen wir anstellen müssen und wie wir die Ziele erreichen können. In Kapitel 2 vergleichen wir verschiedene Datenmodelle und Abfragesprachen und zeigen, wie sie sich für unterschiedliche Situationen eignen. In Kapitel 3 geht es um Speichermodule: wie Datenbanken die Daten auf der Festplatte anordnen, damit wir sie effizient wiederfinden können. Kapitel 4 erläutert die Formate für die Datencodierung (Serialisierung) und die Entwicklung von Schemas im Laufe der Zeit.
2. In *Teil II* gehen wir von den Daten, die auf einem Computer gespeichert sind, zu Daten über, die über mehrere Computer verteilt sind. Dies ist oftmals für Skalierbarkeit erforderlich, bringt aber auch eine Vielzahl von speziellen Problemen mit sich. Zuerst befassen wir uns mit Replikation (Kapitel 5), Partitionierung und Sharding (Kapitel 6) und Transaktionen (Kapitel 7). Dann gehen wir näher auf die Probleme mit verteilten Systemen ein (Kapitel 8) und zeigen, was es bedeutet, Konsistenz und Konsens in einem verteilten System zu erreichen (Kapitel 9).
3. In *Teil III* geht es um Systeme, die bestimmte Datenmengen aus anderen Datenmengen ableiten. Abgeleitete Daten sind häufig in heterogenen Systemen zu finden: Wenn eine Datenbank nicht alle Aufgaben gleichermaßen gut übernehmen kann, müssen Anwendungen mehrere verschiedene Datenbanken, Caches, Indizes usw. integrieren. Zu Beginn zeigt Kapitel 10 das Konzept einer Stapelverarbeitung für abgeleitete Daten, und darauf aufbauend erläutert Kapitel 11 eine Stream-Verarbeitung. Schließlich bringen wir in Kapitel 12 alles zusammen und diskutieren Ansätze, um in Zukunft zuverlässige, skalierbare und wartbare Anwendungen zu erstellen.

Quellen und weiterführende Literatur

Das meiste von dem, was wir in diesem Buch besprechen, ist bereits an anderer Stelle in der einen oder anderen Form gesagt worden – auf Konferenzen, in Forschungsarbeiten, Blog-Posts, Code, Bugtracker-Berichten, Mailinglisten und dem mündlich überlieferten technischen Mythos. Dieses Buch fasst die wichtigsten Ideen aus vielen verschiedenen Quellen zusammen und gibt im gesamten Text Verweise auf die Originalliteratur an. Die Verweise am Ende jedes Kapitels bieten sich an, wenn Sie in einem Themenbereich weiter recherchieren wollen. Zudem sind die meisten davon online frei verfügbar.

Vorwort des Autors zur deutschen Übersetzung

Nachdem die englische Originalfassung dieses Buchs viele begeisterte Leser gefunden hat, war ich sehr erfreut von den Plänen zu erfahren, *Designing Data-Intensive Applications* in mehrere andere Sprachen zu übersetzen. Zu den meisten Übersetzungen, zum Beispiel auf Koreanisch, Russisch und Polnisch, habe ich allerdings keinen Beitrag leisten können, weil ich in Sprachen leider nicht so bewandert bin.

Die deutsche Übersetzung ist jedoch ein Ausnahmefall: Zwar wohne ich inzwischen seit 15 Jahren in Großbritannien, aber da ich in Deutschland aufgewachsen bin, ist mein Deutsch immerhin gut genug, um beim Korrekturlesen der deutschen Übersetzung zu helfen. Ich war von Frank Langenaus Übersetzung von Anfang an beeindruckt: Er hat den beabsichtigten Sinn präzise erfasst und zugleich gute, idiomatische Ausdrucksweisen gefunden – viel besser, als ich es selbst geschafft hätte.

Das vorliegende Buch habe ich komplett überprüft und bei der Übersetzung begleitet; dabei haben wir auch den Inhalt auf den neuesten Stand gebracht. Wir sind bei der Übersetzung auf einige Details gestoßen, die im englischen Original unklar formuliert waren: Diese haben wir nicht nur in der deutschen Fassung korrigiert, sondern auch als Verbesserungen in die nächste Aktualisierung der englischen Ausgabe übernommen. Somit ist die Übersetzung dem Buch doppelt zugutegekommen!

Mein großer Dank gilt also Frank Langenau für seine hervorragende Übersetzung des Buchs. Ebenfalls danken möchte ich Alexandra Follenius und dem Team bei O'Reilly Deutschland.

Danksagungen der englischen Originalfassung

Dieses Buch verschmilzt und systematisiert viele Ideen und Erkenntnisse anderer Menschen, wobei es Erfahrungen sowohl aus der akademischen Forschung als auch der industriellen Praxis zusammenbringt. In der Datenverarbeitung fühlen wir

uns angezogen von neuen und modischen Dingen, doch meiner Ansicht nach können wir noch jede Menge lernen aus den Dingen, die in der Vergangenheit getan wurden. Im Buch finden Sie über 800 Verweise auf Artikel, Blogposts, Vorträge, Dokumentationen und mehr, die auch für mich wertvolle Lernunterlagen waren. Ich bin den Autoren dieses Materials sehr dankbar, dass sie ihr Wissen teilen.

Auch aus persönlichen Gesprächen habe ich viel mitgenommen, dank einer großen Anzahl von Menschen, die sich die Zeit genommen haben, Ideen zu diskutieren oder mir Sachverhalte geduldig zu erklären. Insbesondere möchte ich mich bedanken bei Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Helland, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O'Neil, Onora O'Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden und Brett Wooldridge.

Weitere Personen haben beim Schreiben dieses Buchs wertvolle Hilfe geleistet, indem sie Entwürfe begutachtet und Feedback gegeben haben. Für diese Beiträge danke ich Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, and Ben Summers. Selbstverständlich übernehme ich die gesamte Verantwortung für alle verbliebenen Fehler oder nicht akzeptable Meinungen in diesem Buch.

Für die Unterstützung, dass dieses Buch Wirklichkeit werden konnte, und für ihre Geduld mit meinem langsamen Schreiben und den ausgefallenen Anfragen danke ich meinen Redakteuren Marie Beaugureau, Mike Loukides, Ann Spencer und dem ganzen Team bei O'Reilly. Bedanken möchte ich mich auch bei Rachel Head für Ihre Hilfe, die richtigen Worte zu finden. Dafür, dass sie mir die Zeit und Freiheit gegeben haben, trotz anderweitiger beruflicher Verpflichtungen zu schreiben, danke ich Alastair Beresford, Susan Goodhue, Neha Narkhede und Kevin Scott. Ein ganz besonderer Dank geht an Shabbir Diwan und Edie Freedman, die mit großer Sorgfalt die Karten zu den Kapiteln illustriert haben. Es ist wunderbar, dass sie die unkonventionelle Idee aufgegriffen haben, Landkarten zu erstellen, und dass sie sie so schön und ansprechend gestaltet haben.

Schließlich gilt meine Liebe meiner Familie und meinen Freunden, ohne die ich das Schreiben, das fast vier Jahre gedauert hat, nicht hätte durchstehen können. Ihr seid die Besten.

TEIL I

Grundlagen von Datensystemen

Die ersten vier Kapitel beschäftigen sich mit den fundamentalen Ideen, die auf alle Datensysteme anwendbar sind, egal, ob sie auf einem einzelnen Computer laufen oder über einen Cluster von Computern verteilt sind:

1. Kapitel 1 führt die Terminologie und die Konzepte ein, die wir das gesamte Buch hindurch verwenden werden. Es untersucht, was wir mit Begriffen wir *Zuverlässigkeit*, *Skalierbarkeit* und *Wartbarkeit* tatsächlich meinen und wie sich diese Ziele erreichen lassen.
2. Kapitel 2 vergleicht verschiedene Datenmodelle und Abfragesprachen – der offensichtlichste Unterscheidungsfaktor zwischen Datenbanken aus dem Blickwinkel des Entwicklers. Hier erfahren Sie, wie verschiedene Modelle für verschiedene Situationen geeignet sind.
3. Kapitel 3 wendet sich den Interna der Speichermodule zu und zeigt, wie Datenbanken die Daten auf dem Datenträger anordnen. Verschiedene Speichermodule sind für unterschiedliche Arbeitsbelastungen optimiert, und die Auswahl des richtigen Moduls kann sich drastisch auf die Performance auswirken.
4. Kapitel 4 vergleicht Formate für die Codierung (Serialisierung) von Daten und untersucht insbesondere, wie sie sich in einer Umgebung verhalten, in der sich die Anforderungen an die Anwendungen ändern und Schemas im Laufe der Zeit angepasst werden müssen.

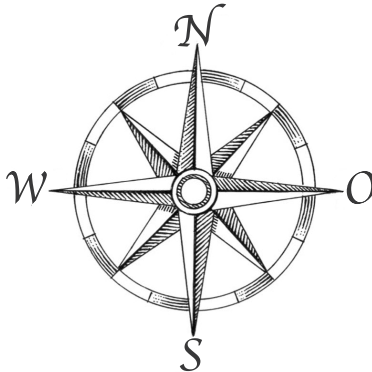
Später wendet sich Teil II den speziellen Fragen verteilter Datensysteme zu.

Datenintensive
Anwendungen
designen
Konzepte für zuverlässige, skalierbare
und wartbare Systeme
ZUVERLÄSSIGKEIT
SKALIERBARKEIT
WARTBARKEIT

ZUVERLÄSSIGKEIT
Hardware- und
Softwarefehler
tolerieren
Menschlicher
Fehler

SKALIERBARKEIT
Messen von
Belastung &
Leistung
Latenz-Perzentile,
Durchsatz

WARTBARKEIT
Betriebsfähigkeit,
Einfachheit &
Erweiterbarkeit


N
W
O
S

KAPITEL 1

Zuverlässige, skalierbare und wartbare Anwendungen

> *Das Internet wurde so gut gemacht, dass die meisten Menschen es als eine natürliche Ressource wie den Pazifischen Ozean betrachten, und nicht als etwas, das vom Menschen geschaffen wurde. Wann war das letzte Mal eine Technologie in einer solchen Größenordnung so fehlerfrei?*
>
> – **Alan Kay** im Interview mit *Dr Dobb's Journal* (2012)

Viele Anwendungen sind heutzutage *datenintensiv* im Gegensatz zu *rechenintensiv*. Die CPU-Leistung an sich ist für diese Anwendungen kaum ein begrenzender Faktor – größere Probleme ergeben sich üblicherweise aus dem Umfang der Daten, ihrer Komplexität und der Geschwindigkeit, mit der sie sich verändern.

Eine datenintensive Anwendung besteht normalerweise aus Standardbausteinen, die häufig benötigte Funktionalität bereitstellen. Zum Beispiel müssen viele Anwendungen

- Daten speichern, damit sie oder andere Anwendungen die Daten später wiederfinden können (*Datenbanken*),
- das Ergebnis einer aufwendigen Operation zwischenspeichern, um Lesevorgänge zu beschleunigen (*Caches*),
- Benutzern ermöglichen, Daten nach Schlüsselwörtern zu durchsuchen oder nach verschiedenen anderen Methoden zu filtern (*Suchindizes*),
- eine Nachricht an einen anderen Prozess senden, um eine asynchrone Verarbeitung zu veranlassen (*Streamverarbeitung*) und
- regelmäßig eine große Menge akkumulierter Daten verarbeiten (*Stapelverarbeitung*).

Sollte das zu offensichtlich klingen, dann nur, weil diese *Datensysteme* eine so erfolgreiche Abstraktion sind: Wir verwenden sie die ganze Zeit, ohne groß darüber nachzudenken. Wenn ein Entwickler eine Anwendung erstellt, wird er kaum davon träumen, ein neues Speichermodul von Grund auf neu zu schreiben, denn für diese Aufgabe sind Datenbanken prädestiniert.

Die Realität sieht aber nicht so einfach aus. Es existieren viele Datenbanksysteme mit unterschiedlichen Eigenschaften, weil verschiedene Anwendungen unterschiedliche Anforderungen stellen. Für das Zwischenspeichern gibt es verschiedene Methoden, das Gleiche gilt für das Erstellen von Indizes usw. Wenn wir eine Anwendung erstellen, müssen wir immer noch herausfinden, welche Werkzeuge und welche Ansätze für die konkrete Aufgabe am besten geeignet sind. Und falls ein einzelnes Werkzeug diese Aufgabe nicht allein bewältigen kann, ist es mitunter schwierig, passende Tools zu kombinieren.

Dieses Buch führt Sie sowohl durch die Prinzipien als auch die praktischen Aspekte von Datensystemen und zeigt, wie Sie damit datenintensive Anwendungen erstellen können. Wir untersuchen, was verschiedene Werkzeuge gemeinsam haben, was sie unterscheidet und wie sie zu ihren Eigenschaften kommen.

In diesem Kapitel untersuchen wir zunächst die Grundlagen für das, was wir erreichen wollen: zuverlässige, skalierbare und wartbare Datensysteme. Wir machen deutlich, was diese Dinge bedeuten, umreißen Methoden, sie zu analysieren, und wenden uns den Basics zu, die für die späteren Kapitel erforderlich sind. In den folgenden Kapiteln fahren wir mit den einzelnen Ebenen nacheinander fort und sehen uns dabei die verschiedenen Entwurfsentscheidungen an, die beim Arbeiten an einer datenintensiven Anwendung betrachtet werden müssen.

Gedanken zu Datensystemen

In der Regel stellen wir uns Datenbanken, Warteschlangen, Caches usw. als vollkommen verschiedene Kategorien von Werkzeugen vor. Obwohl eine Datenbank und eine Nachrichtenwarteschlange einige oberflächliche Berührungspunkte aufweisen – beide speichern Daten für einen gewissen Zeitraum –, unterscheiden sie sich in ihren Zugriffsmustern. Das bedeutet verschiedene Leistungscharakteristika und somit sehr verschiedene Implementierungen.

Warum sollten wir sie alle unter einem Sammelbegriff wie *Datensysteme* zusammenfassen?

1In den letzten Jahren sind zahlreiche neue Tools für das Speichern und Verarbeiten von Daten entstanden. Optimiert für eine breite Vielfalt von Einsatzfällen lassen sie sich nicht mehr streng den herkömmlichen Kategorien zuordnen [1]. So gibt es zum Beispiel Datenspeicher, die auch als Nachrichtenwarteschlangen dienen (Redis), und Nachrichtenwarteschlangen mit datenbankähnlichen Beständigkeitsgarantien (Apache Kafka). Die Grenzen zwischen den Kategorien verschwimmen immer weiter.

Zweitens stellen heute immer mehr Anwendungen so anspruchsvolle oder breit gefächerte Anforderungen, dass ein einzelnes Tool nicht mehr sämtliche Ansprüche an die Verarbeitung und Speicherung der Daten realisieren kann. Stattdessen teilt man die Arbeit in Aufgaben auf, die ein einzelnes Tool effizient durchführen *kann*, und der Anwendungscode verknüpft diese verschiedenen Tools.

Haben Sie zum Beispiel eine von der Anwendung verwaltete Caching-Ebene (etwa mit einem Cache-Server wie *Memcached*) oder einen Server für die Volltextsuche (wie zum Beispiel Elasticsearch oder Solr) eingerichtet, die von Ihrer Hauptdatenbank getrennt sind, ist normalerweise der Code der Anwendung dafür zuständig, diese Caches und Indizes mit der Hauptdatenbank synchron zu halten. Abbildung 1-1 veranschaulicht dieses Prinzip (mehr Einzelheiten folgen in späteren Kapiteln).

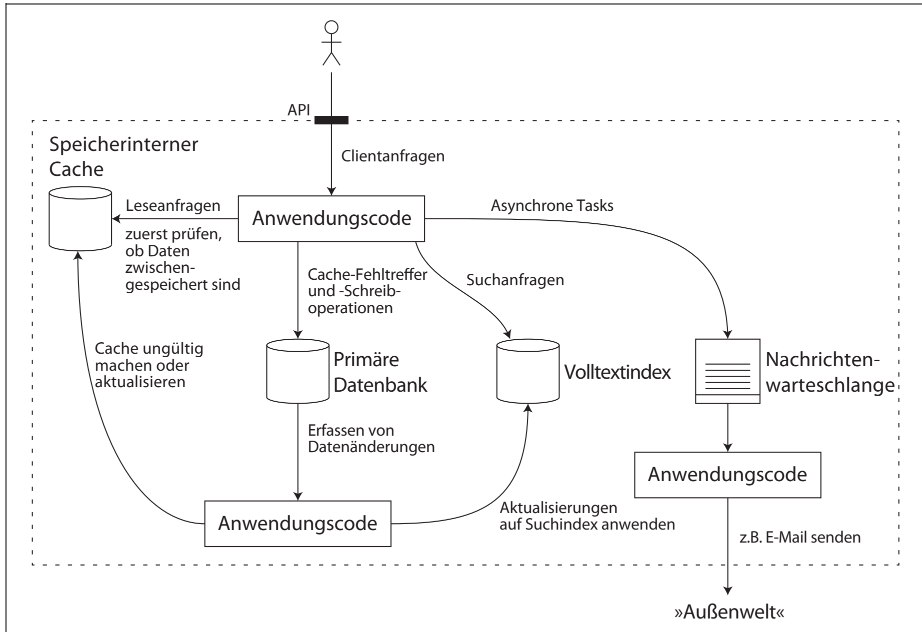


Abbildung 1-1: Eine mögliche Architektur für ein Datensystem, das mehrere Komponenten kombiniert

Wenn Sie mehrere Tools kombinieren, um einen Dienst zu realisieren, verbirgt die Oberfläche des Diensts oder die API[1] normalerweise diese Implementierungsdetails vor den Clients. Praktisch haben Sie nun ein neues spezialisiertes Datensystem aus kleineren, universellen Komponenten erzeugt. Das zusammengesetzte Datensystem kann bestimmte Garantien bieten: beispielsweise, dass der Cache korrekt ungültig gemacht oder bei Schreibvorgängen aktualisiert wird, sodass externe Clients konsistente Ergebnisse sehen. Jetzt sind Sie nicht nur Anwendungsentwickler, sondern auch Datensystemdesigner.

Beim Entwurf eines Datensystems oder eines Diensts tauchen viele knifflige Fragen auf. Wie stellen Sie sicher, dass die Daten korrekt und vollständig bleiben, selbst wenn intern etwas schiefläuft? Wie bieten Sie den Clients eine konstant gute Performance, selbst wenn Teile Ihres Systems ausfallen? Wie skalieren Sie, um einer wachsenden Belastung gerecht zu werden? Wie sieht eine gute API für den Dienst aus?

1 API – Application Programming Interface, Schnittstelle zur Anwendungsprogrammierung

Viele Faktoren können das Design eines Datensystems beeinflussen, unter anderem die Fertigkeiten und Erfahrungen der beteiligten Entwickler, Abhängigkeiten von einem Legacysystem, die Lieferzeit, die Toleranz Ihres Unternehmens gegenüber verschiedenen Arten von Risiken, regulatorische Beschränkungen usw. Derartige Faktoren hängen stark von der jeweiligen Situation ab.

In diesem Buch konzentrieren wir uns auf drei Aspekte, die in den meisten Softwaresystemen wichtig sind:

Zuverlässigkeit
: Das System sollte auch bei Widrigkeiten (Hardware- oder Softwarefehlern und sogar menschlichem Versagen) weiterhin *korrekt* arbeiten (die richtige Funktion auf dem gewünschten Leistungsniveau ausführen). Siehe Abschnitt »Zuverlässigkeit« unten.

Skalierbarkeit
: Wenn das System wächst (in Bezug auf Datenvolumen, Verkehrsaufkommen oder Komplexität), sollte es vernünftige Maßnahmen geben, um mit diesem Wachstum umzugehen. Siehe Abschnitt »Skalierbarkeit« auf Seite 11.

Wartbarkeit
: Im Laufe der Zeit arbeiten verschiedene Leute am System (Techniker und Betreiber, die sowohl das aktuelle Verhalten sicherstellen als auch das System an neue Einsatzfälle anpassen), und sie alle sollten daran *produktiv* arbeiten können. Siehe Abschnitt »Wartbarkeit« auf Seite 20.

Diese Begriffe werden oftmals in den Raum geworfen, ohne überhaupt ihre Bedeutung genau verstanden zu haben. Im Sinne einer nachvollziehbaren Herangehensweise werden wir im Rest dieses Kapitels Möglichkeiten untersuchen, um Überlegungen zu Zuverlässigkeit, Skalierbarkeit und Wartbarkeit anzustellen. In den darauffolgenden Kapiteln sehen wir uns dann die verwendeten Techniken, Architekturen und Algorithmen an, mit denen sich diese Ziele erreichen lassen.

Zuverlässigkeit

Wohl jeder hat eine intuitive Vorstellung davon, was zuverlässig oder unzuverlässig bedeutet. Zu den typischen Erwartungen an Software gehören:

- Die Anwendung führt die Funktion aus, die der Benutzer erwartet.
- Sie kann tolerieren, dass der Benutzer Fehler macht bzw. die Software auf unerwartete Art und Weise benutzt.
- Ihre Leistung ist gut genug für den vorgesehenen Einsatzfall, unter der erwarteten Arbeitslast und für das anfallende Datenvolumen.
- Das System verhindert jeden nicht autorisierten Zugriff und jeden Missbrauch.

Wenn all dies zusammengenommen »korrekt arbeiten« bedeutet, dann können wir *Zuverlässigkeit* ganz grob verstehen als »weiterhin korrekt arbeiten, auch wenn etwas schiefläuft«.

Die Dinge, die schiefgehen, bezeichnet man als *Fehler*, und Systeme, die Fehler einkalkulieren und bewältigen können, heißen *fehlertolerant* oder *robust*. Der erste Begriff ist etwas irreführend, suggeriert er doch, dass wir ein System gegenüber jeder Art von Fehlern tolerant machen könnten, was in der Praxis aber nicht realisierbar ist. Wenn man damit rechnet, dass ein schwarzes Loch den gesamten Planeten Erde (und alle Server auf ihm) verschluckt, müsste Webhosting im Weltraum stattfinden, um Fehlertoleranz für dieses Ereignis zu bieten – viel Erfolg dabei, diesen Haushaltsposten genehmigt zu bekommen. Es ist demnach nur sinnvoll, von der Toleranz gegenüber *bestimmten Fehlerarten* zu sprechen.

Beachten Sie, dass ein Fehler nicht dasselbe ist wie ein Ausfall [2]. Entsprechend der üblichen Definition ist ein *Fehler* eine Komponente des Systems, die von ihrer Spezifikation abweicht, während bei einem *Ausfall* das System als Ganzes aufhört, dem Benutzer den gewünschten Dienst bereitzustellen. Da es nicht möglich ist, die Wahrscheinlichkeit eines Fehlers auf null zu verringern, ist es normalerweise am besten, Fehlertoleranzmechanismen zu entwickeln, die verhindern, dass Fehler zu Ausfällen führen. In diesem Buch behandeln wir verschiedene Techniken, um zuverlässige Systeme aus unzuverlässigen Bestandteilen aufzubauen.

Auch wenn es der Intuition widerspricht, kann es in derartigen fehlertoleranten Systemen sinnvoll sein, die Fehlerrate zu *erhöhen*, indem man Fehler bewusst auslöst – zum Beispiel durch zufälliges Beenden einzelner Prozesse ohne Warnung. Viele kritische Bugs gehen auf eine schlechte Fehlerbehandlung zurück [3]. Indem Sie Fehler bewusst herbeiführen, stellen Sie sicher, dass der Fehlertoleranzmechanismus laufend beansprucht und getestet wird. Das kann Ihr Vertrauen stärken, dass Fehler ordnungsgemäß behandelt werden, wenn sie im regulären Betrieb auftreten. Ein Beispiel für dieses Konzept ist das Tool *Chaos Monkey* [4] von Netflix.

Obwohl wir im Allgemeinen Fehler lieber tolerieren als verhindern, gibt es Fälle, in denen Vorbeugen besser ist als Heilen (zum Beispiel, weil es keine Heilung gibt). Das ist unter anderem bei Sicherheitsfragen der Fall: Wenn ein Angreifer ein System gehackt und Zugriff auf vertrauliche Daten erlangt hat, lässt sich ein solches Ereignis nicht mehr rückgängig machen. Allerdings beschäftigt sich dieses Buch vorrangig mit solchen Fehlern, die sich beheben lassen, wie die folgenden Abschnitte beschreiben.

Hardwarefehler

Denkt man an die Ursachen für Systemausfälle, kommen einem schnell Hardwarefehler in den Sinn: ein Festplattencrash tritt auf, RAM-Zellen verlieren Speicherinhalte, das Stromnetz bricht kurzzeitig zusammen, ein falsches Netzwerkkabel

wurde eingesteckt. Jeder, der schon einmal mit großen Datencentern gearbeitet hat, wird bestätigen, dass diese Dinge *ständig* passieren, wenn nur genügend Computer vorhanden sind.

Bei Festplatten beträgt die mittlere Betriebszeit bis zum Ausfall (Mean Time To Failure, MTTF) etwa 10 bis 50 Jahre [5, 6]. Folglich ist in einem Speichercluster mit 10.000 Festplatten im Durchschnitt ein Festplattenausfall pro Tag zu erwarten.

Unsere erste Reaktion darauf ist üblicherweise, die einzelnen Hardwarekomponenten mit mehr Redundanz auszustatten, um die Ausfallrate des Systems zu verringern. Festplatten lassen sich in einer RAID-Konfiguration betreiben, Server sind mit doppelten Stromversorgungen und Hot-Swap-fähigen CPUs ausgerüstet, und in Rechenzentren sichern Batterien und Dieselgeneratoren die Notstromversorgung ab.

Wenn eine Komponente kaputtgeht, kann die redundante Komponente ihren Platz einnehmen, während die defekte Komponente ersetzt wird. Zwar lässt sich mit diesem Konzept nicht komplett verhindern, dass Hardwareprobleme zu Ausfällen führen, doch es ist praktikabel und sorgt oftmals dafür, dass ein Computer jahrelang ununterbrochen läuft.

Bis vor Kurzem genügten redundante Hardwarekomponenten für die meisten Anwendungen, da dank dieser Maßnahme einzelne Computer nur ziemlich selten komplett ausfallen. Sofern Sie recht schnell eine Sicherung auf einem neuen Computer wiederherstellen können, ist die Ausfallzeit bei den meisten Anwendungen nicht dramatisch. Somit ist eine Redundanz mit mehreren Computern nur für eine kleine Anzahl von Anwendungen erforderlich, bei denen Hochverfügbarkeit an vorderster Stelle steht.

Wegen größerer Datenmengen und gestiegener rechentechnischer Anforderungen geht man jedoch bei immer mehr Anwendungen dazu über, eine größere Anzahl von Computern zu nutzen, wodurch die Rate der Hardwarefehler proportional zunimmt. Darüber hinaus kommt es bei manchen Cloudplattformen wie zum Beispiel Amazon Web Services (AWS) durchaus vor, dass Instanzen virtueller Computer ohne Vorwarnung unverfügbar werden [7], weil die Plattformen dafür konzipiert sind, Flexibilität und Elastizität[2] über die Zuverlässigkeit einzelner Computer zu priorisieren.

Folglich gibt es eine Verschiebung hin zu Systemen, die den Verlust ganzer Computer tolerieren können, indem sie softwareseitige Fehlertoleranztechniken bevorzugen oder ergänzend zur Hardwareredundanz einsetzen. Solche Systeme bieten auch operative Vorteile: Bei einem Einzelserversystem müssen Sie die Stillstandszeiten planen, falls Sie den Computer neu starten müssen (um zum Beispiel Sicherheitspatches des Betriebssystems zu installieren), während sich ein System, das

2 Wird definiert in Abschnitt »Konzepte zur Bewältigung von Belastungen« auf Seite 18.

den Ausfall eines Computers tolerieren kann, knotenweise mit Patches versorgen lässt, ohne dass das gesamte System stillsteht (*rollendes Upgrade*; siehe Kapitel 4).

Softwarefehler

Normalerweise geht man davon aus, dass Hardwarefehler zufällig und unabhängig voneinander auftreten: Fällt bei einem Computer die Festplatte aus, heißt das nicht, dass die Festplatte in einem anderen Computer ebenfalls kaputtgeht. Es kann zwar schwache Korrelationen geben (etwa bei zu hohen Temperaturen im Servergestell), doch ansonsten ist es unwahrscheinlich, dass eine große Anzahl von Hardwarekomponenten gleichzeitig ausfällt.

Zu einer anderen Fehlerklasse gehören systematische Fehler innerhalb des Systems [8]. Derartige Fehler sind schwerer vorherzusehen, und weil sie über Knoten korreliert sind, verursachen sie mehr Systemausfälle als nicht korrelierte Hardwarefehler [5]. Beispiele dafür sind:

- Ein Softwarebug bewirkt, dass jede Instanz eines Anwendungsservers abstürzt, wenn eine bestimmte Fehleingabe erfolgt. Denken Sie nur an die Schaltsekunde am 30. Juni 2012, die aufgrund eines Fehlers im Linux-Kernel [9] dazu führte, dass viele Anwendungen gleichzeitig hängenblieben.
- Ein unkontrollierbarer Prozess erschöpft eine gemeinsam genutzte Ressource – CPU-Zeit, Arbeitsspeicher, Festplattenplatz oder Netzwerkbandbreite.
- Ein Dienst, von dem das System abhängig ist, wird langsamer, reagiert nicht mehr oder gibt beschädigte Antworten zurück.
- Kaskadenartiges Ausbreiten von Fehlern, wobei ein kleiner Fehler in der einen Komponente einen Fehler in einer anderen Komponente auslöst, die ihrerseits weitere Fehler auslöst [10].

Die Bugs, die für derartige Softwarefehler verantwortlich sind, schlummern oftmals eine ganze Zeit lang, bis sie durch das Zusammentreffen ungewöhnlicher Umstände zutage treten. In diesen Fällen zeigt sich, dass die Software bestimmte Annahmen über ihre Umgebung trifft – und normalerweise sind diese Annahmen auch richtig, doch schließlich treffen sie aus irgendeinem Grund nicht mehr zu [11].

Für das Problem systematischer Softwarefehler gibt es keine schnelle Lösung. Viele kleine Dinge können helfen: gründliches Nachdenken über Annahmen und Wechselwirkungen im System, umfangreiche Tests, Prozessisolierung; Zulassen, dass Prozesse abstürzen und neu starten; Messen, Überwachen und Analysieren des Systemverhaltens in der Produktion. Wenn man von einem System eine gewisse Garantie erwartet (dass zum Beispiel in einer Nachrichtenwarteschlange die Anzahl der eintreffenden Nachrichten gleich der Anzahl der ausgehenden Nachrichten ist), kann es sich im Betrieb ständig selbst überprüfen und einen Alarm auslösen, wenn es eine Abweichung feststellt [12].

Menschliche Fehler

Menschen entwerfen und erstellen Softwaresysteme, und die Betreiber, die die Systeme am Laufen halten, sind ebenfalls Menschen. Selbst wenn sie die besten Absichten haben, sind Menschen bekanntlich unzuverlässig. So geht aus einer Studie über große Internetdienste hervor, dass Konfigurationsfehler von Betreibern die Hauptursache für Ausfälle waren, während Hardwarefehler (Server oder Netzwerk) nur in 10 bis 25 % der Ausfälle eine Rolle gespielt haben [13].

Wie machen wir nun unsere Systeme trotz unzuverlässiger Menschen zuverlässig? Die besten Systeme kombinieren mehrere Ansätze:

- Systeme so entwerfen, dass Fehlermöglichkeiten minimiert werden. Beispielsweise erleichtern es gut konzipierte Abstraktionen, APIs und Administrationsoberflächen, »das Richtige« zu tun und »das Falsche« zu unterbinden. Wenn jedoch die Schnittstellen zu restriktiv sind, umgeht der Programmierer sie und negiert damit ihren Nutzen. Dadurch ist es schwierig, das richtige Gleichgewicht zu finden.
- Die Stellen, an denen Programmierer die meisten Fehler machen, von den Stellen entkoppeln, wo sie Ausfälle hervorrufen können. Stellen Sie insbesondere voll ausgestattete *Sandbox*-Testumgebungen bereit, die Programmierer erkunden und in ihnen mit echten Daten gefahrlos experimentieren können, ohne dass wirkliche Benutzer davon betroffen sind.
- Gründlich auf allen Ebenen testen, angefangen bei Komponententests (Unit Tests) bis hin zu Integrationstests mit dem gesamten System und manuellen Tests [3]. Automatisiertes Testen ist weit verbreitet, hinreichend bekannt und vor allem wertvoll, um die Grenzfälle abzudecken, die im normalen Betrieb selten auftreten.
- Schnelle und einfache Wiederherstellung bei Fehlern, die auf den Menschen zurückgehen, ermöglichen, die Auswirkungen im Fehlerfall zu minimieren. Zum Beispiel: Konfigurationsänderungen schnell zurücksetzen, neuen Code schrittweise einführen (sodass unerwartete Bugs nur eine kleine Untergruppe von Benutzern betreffen) und Tools bereitstellen, mit denen sich Daten neu berechnen lassen (falls sich herausstellt, dass die alten Berechnungen nicht korrekt waren).
- Detaillierte und klare Überwachung einrichten, wie zum Beispiel Leistungskennziffern und Fehlerquoten. Andere technische Bereiche sprechen hier von *Telemetrie*. (Nachdem eine Rakete den Boden verlassen hat, sind die Telemetriedaten unerlässlich, um das Geschehen verfolgen und Fehlerereignisse deuten zu können [14].) Die Überwachung liefert uns frühzeitige Warnsignale und erlaubt uns zu überprüfen, ob Annahmen oder Einschränkungen verletzt werden. Tritt ein Problem auf, sind Messwerte unabdingbar für die Diagnose der Ursache.
- Gute Verwaltungspraktiken und Schulungen umsetzen – ein komplexer und wichtiger Aspekt, der aber über den Rahmen dieses Buchs hinausginge.