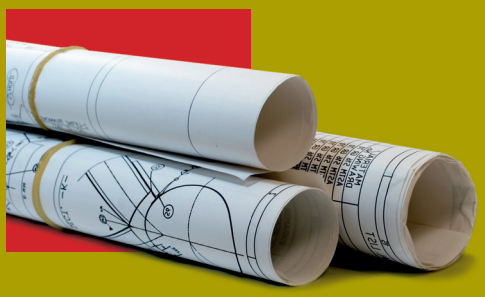

jürgen **DUNKEL**
andreas **EBERHART**
stefan **FISCHER**
carsten **KLEINER**
arne **KOSCHEL**

# SYSTEM-ARCHITEKTUREN FÜR VERTEILTE ANWENDUNGEN

**CLIENT-SERVER // MULTI-TIER // SOA // EVENT-DRIVEN ARCHITECTURE // P2P // GRID // WEB 2.0**

HANSER

eBOOK

Dunkel/Eberhart/Fischer/Kleiner/Koschel

# Systemarchitekturen für Verteilte Anwendungen

Jürgen Dunkel
Andreas Eberhart
Stefan Fischer
Carsten Kleiner
Arne Koschel

# Systemarchitekturen für Verteilte Anwendungen

## Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0

HANSER

*Prof. Dr. Jürgen Dunkel*, Fachhochschule Hannove, Fakultät IV, Abteilung Informatik
*Dr.-Ing. Andreas Eberhart*, Fluid Operations GmbH, Walldorf
*Prof. Dr. Stefan Fischer*, Universität zu Lübeck, Institut für Telematik
*Prof. Dr. Carsten Kleiner*, Fachhochschule Hannover, Fakultät IV, Abteilung Informatik
*Prof. Dr. Arne Koschel*, Fachhochschule Hannover, Fakultät IV, Abteilung Informatik



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.


Lektorat: Margarete Metzger
Herstellung: Irene Weilhart
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisation: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-41321-4

# Inhaltsverzeichnis

# Vorwort

Verteilte DV-Lösungen werden mehr und mehr zum Kern der IT-Infrastruktur sowohl bei Weltkonzernen als auch bei kleinen und mittleren Unternehmen. Sie ermöglichen einen effizienten Informationsfluss innerhalb eines Unternehmens und in Kunden- Lieferanten-Beziehungen. Über die Jahre haben sich Client-Server, Multi-Tier und SOA als mögliche Architekturen etabliert, aber auch neue Ansätze wie Event-Driven Architecture, P2P, Grid und Web 2.0 liefern wichtige Impulse.

Entscheider, Projektleiter, Software-Architekten und -Ingenieure stehen vor der Herausforderung, aus der Vielzahl der Möglichkeiten die „richtige“ Architektur für ihr Einsatzszenario auszuwählen.

In diesem Buch möchten wir dazu schrittweise Entscheidungshilfen geben. Zunächst stellen wir die grundlegenden Konzepte für die wichtigsten Systemarchitekturen vor. Um Ihnen ein Gefühl für die technische Umsetzung der verschiedenen Architekturen zu geben, werden jeweils die wichtigsten Realisierungsplattformen sowie einfache Code-Beispiele vorgestellt. Dabei werden wir uns nicht in technische Details verlieren, sondern auf die entscheidenden Merkmale und Eigenschaften beschränken.

Im nächsten Schritt entwickeln wir Vergleichskriterien, mit deren Hilfe sich die verschiedenen Ansätze abgrenzen und bewerten lassen. Die aufgezeigten architektur-spezifischen Vor- und Nachteile sollen Ihnen dabei helfen, eine Ihrem konkreten Projektumfeld angepasste Architekturentscheidung treffen zu können. Abschließend zeigen wir anhand einer Reihe von Fallbeispielen den Einsatz der Architekturen in der Praxis und geben einen Ausblick auf zukünftige Entwicklungen.

Wir hoffen, dass Ihnen das vorliegende Buch Spaß beim Lesen bereitet und dabei hilft, die aktuellen Trends im Bereich Systemarchitekturen verstehen und bewerten zu können. Über Feedback von unseren Lesern freuen wir uns unter `autoren_sva@gmx.de`.

*Hannover, Bruchsal, Lübeck im Juli 2008*

*Jürgen Dunkel, Andreas Eberhart, Stefan Fischer, Carsten Kleiner und Arne Koschel*

# Teil I

# Einführung

# Kapitel 1

# Motivation und Überblick

Wir befinden uns in der Informationstechnologie zurzeit in einer Umbruchphase, die mindestens mit dem durch die Einführung des Personal Computers ausgelösten Paradigmenwechsel gleichzusetzen ist. Die Anfang der 90er-Jahre des vergangenen Jahrhunderts zum ersten Mal konkret durch Marc Weiser beschriebenen Ideen des Pervasive bzw. *Ubiquitous Computing* werden mehr und mehr zur Wirklichkeit und sorgen so dafür, dass die reale Welt, in der wir leben, immer mehr mit der virtuellen Welt, dem Cyberspace, verschmilzt. Wir beobachten bzw. erleben selbst mit, wie jeder Mensch – zumindest in den Industrienationen – Zugriff auf immer mehr Computer hat. Mussten sich zu Beginn des Informationstechnologiezeitalters noch viele Menschen einen Rechner teilen, so bekam mit dem PC jeder seinen eigenen. Bereits heute verwenden die meisten Menschen mehrere Geräte, die als Computer zu bezeichnen sind: Handys, Personal Digital Assistants (PDAs), Notebooks, PCs etc. In jedem modernen Auto sind heute mehrere Dutzend CPUs verbaut (mit der Folge, dass die meisten Pannen heute auf Softwarefehler zurückzuführen sind). Immer mehr eingebettete Geräte (*embedded devices*) sind mit Rechenintelligenz ausgestattet. In den Visionen von Marc Weiser werden Computer in wenigen Jahren zum Wegwerfartikel: man nimmt sich, wann immer nötig, eine Handvoll Computer aus der Hosentasche, benutzt sie und wirft sie weg, wenn sie ihre Aufgabe erfüllt haben.

Das ist aber noch nicht alles. Als zweite große Entwicklung ist die zunehmende Kommunikationsfähigkeit aller Geräte und deren Vernetzung untereinander festzustellen. War noch vor wenigen Jahren ein Datenaustausch zwischen verschiedenen Computern drahtgebunden und typischerweise eher langsam, so stehen den Anwendern heute zahllose Möglichkeiten zur Verfügung, Daten mit hoher Geschwindigkeit über unterschiedlichste Kanäle auszutauschen. Ausgehend vom Handynetz GSM, das für die Kommunikation über weite Strecken geeignet ist, entstanden Hochgeschwindigkeitslösungen für den Fernbereich wie UMTS, aber auch Nahverkehrsnetze wie WLAN oder Bluetooth und neuerdings mit Zigbee

auch Systeme, die sehr sparsame Kommunikation über kurze Strecken gestatten. Damit besteht bereits heute die Möglichkeit, von jedem Ort der Welt auf das Internet zuzugreifen, aber auch Daten an jedem Ort der Welt zu erfassen und ins Netz einzuspeisen. In vielen Wirtschaftsbranchen steht damit ein erheblicher Wandel kurz bevor bzw. ist bereits in vollem Gange. So lässt sich etwa für die Logistikbranche absehen, dass der Weg einer Ware oder eines Transportmittels lückenlos online verfolgt werden kann und so Transportströme sehr viel besser kontrolliert und gesteuert werden können.

Der Kern solcher Anwendungen liegt in einer zunehmenden Dezentralisierung. Immer mehr informationsverarbeitende Komponenten arbeiten zusammen, um die Gesamtleistung eines Systems zu erbringen. Jede dieser Komponenten läuft autonom ab, produziert möglicherweise Informationen, ruft Informationen aus anderen Teilen des Netzes ab und kombiniert diese zu neuen Informationen, die wiederum für andere Komponenten in ganz anderen Systembereichen von Interesse sind. Offenbar wird dies genau durch die oben geschilderten Entwicklungen bzgl. der Zunahme der Zahl der Computer und ihrer Kommunikationsfähigkeit begünstigt. Die Fähigkeit, dezentrale verteilte Systeme und Anwendungen in Netzwerken aufzubauen und zu organisieren, wird zu einem der wichtigsten Wettbewerbsfaktoren unserer Zeit und der unmittelbaren Zukunft.

Eine entscheidende Bedeutung kommt dabei der Softwarearchitektur des verteilten Systems zu. Sie beschreibt im Wesentlichen, wie das Zusammenspiel der einzelnen Komponenten organisiert ist und wie daraus die Gesamtleistung des Systems entsteht. Die Entscheidung über eine konkrete Architektur ist eine der wesentlichen Entscheidungen für die Realisierung des Gesamtsystems – ein hier einmal begangener Fehler ist nur unter äußerst hohen Kosten wieder zu korrigieren. Dementsprechend müssen Systemdesigner genau ihre Optionen kennen und auf der Basis des gegebenen Anwendungsproblems, der Systemumgebung und verschiedener anderer Parameter entscheiden, welche Architektur die richtige ist.

An dieser Stelle möchte dieses Buch ansetzen. Es will einerseits die heute verfügbaren Systemarchitekturen für verteilte Systeme darstellen und dabei auch vergleichen und andererseits einem Entscheider Mittel an die Hand geben, um seine Entscheidung wohl begründet treffen zu können.

Zu diesem Zweck ist das Buch in vier große Abschnitt aufgeteilt. Dieses Überblickskapitel sowie das anschließende Kapitel, in dem die Begriffe „Softwarearchitektur“ und „Systemarchitektur“ erläutert werden, bilden den einführenden ersten Teil. Teil II geht in insgesamt sieben Kapiteln auf die heute verfügbaren und relevanten Softwarearchitekturen für verteilte Systeme ein. Ausgehend von den „Mainstream“-Ansätzen der N-Schichten-Architekturen (*Client-Server*, *3-* und *4-Tier*) werden darauf aufbauend die *Service-Oriented* und *Event-Driven Architectures* (SOA, EDA) eingeführt. Diese eher Business-orientierten Ansätze haben in der letzten Zeit Gegenpole in anderen Bereichen gefunden, von denen vor allem *Peer-to-Peer-Architekturen* (P2P) und *Grid-Computing* (heute vielfach im wissenschaftlichen Bereich vertreten) eine Rolle spielen. Teil II schließt mit den zur Zeit der

Abfassung dieses Buches aktuellsten Ansätzen rund um Web 2.0. Dabei ist jedes Teilkapitel in der gleichen Art und Weise aufgebaut: einer Beschreibung des generellen Konzeptes, meist im Vergleich mit den vorher beschriebenen „Konkurrenzarchitekturen“ folgt eine Darstellung heute verfügbarer Realisierungsplattformen, mit denen sich eine Lösung basierend auf der jeweiligen Architektur dann konkret realisieren lässt. Das Kapitel schließt jeweils mit einigen Code-Beispielen, um dem Leser einen Eindruck von den Möglichkeiten des jeweiligen Ansatzes zu vermitteln.

Teil III als zweiter Hauptteil des Buches soll Ihnen Entscheidungshilfen an die Hand geben. Dazu ist er in zwei Kapitel aufgeteilt: Kapitel 10 entwickelt zunächst eine Reihe von Kriterien, anhand derer die verschiedenen Architekturen verglichen werden können. Letztere gehen sowohl von der Architektur selbst als auch von den Anforderungen der Anwendung aus. Mit diesen Hilfsmitteln kann in vielen Fällen bereits eine Entscheidung oder mindestens Vorentscheidung getroffen werden, welche Architektur(en) im Prinzip geeignet ist (sind). In Kapitel 11 wird dann anhand einiger Beispiele dargestellt, welche Entscheidung in der Praxis in ganz konkreten Fällen getroffen wurde und warum. Anhand dieser Beispiele kann der Leser selbst weitere Parallelen zu den eigenen Anwendungsfällen ziehen und die Entscheidung weiter verfeinern.

Teil IV schließt das Buch mit einem Ausblick auf künftige Entwicklungen und mit einer kurzen Zusammenfassung ab.

# Kapitel 2

# Softwarearchitekturen

In diesem Kapitel werden wir uns allgemein mit dem Begriff der „Softwarearchitektur für verteilte Systeme“ beschäftigen. Dazu klären wir im ersten Abschnitt zunächst den generellen Begriff der Softwarearchitektur. Bei genauerem Hinsehen kann man feststellen, dass man zunächst einige einfache, aber wichtige Leitideen verinnerlichen sollte, bevor man an die Details der Softwareerstellung geht. Diese Grundgedanken sind Thema des zweiten Abschnitts. Sie münden in Überlegungen zu einigen überschaubaren Kriterien für eine „gute“ Softwarearchitektur, die wir in einem späteren Teil des Buches noch einmal intensiv benötigen werden. Die beiden restlichen Abschnitte des Buches beschäftigen sich dann mit den Anforderungen, die die Verteiltheit einer Anwendung für die Architektur mit sich bringt. Zunächst betrachten wir die wichtigsten Aspekte, die ein verteiltes von einem nicht verteilten System unterscheidet. Schließlich geben wir einen Überblick über die bekanntesten Architekturen verteilter Systeme, die im späteren Verlauf des Buches viel ausführlicher behandelt werden.

## 2.1 Der Begriff „Softwarearchitektur“

Den Begriff „Architektur“ kennt man zunächst einmal allgemein vom Bauwesen her. Er meint im Wesentlichen den planvollen Entwurf und die Gestaltung von Bauwerken. Dies lässt sich nun leicht auf den Begriff der Softwarearchitektur übertragen: es geht um einen systematischen Ansatz zur Entwicklung von Software. Im engeren Sinne beschreibt die Architektur einer Software den Aufbau ihrer Komponenten und deren Zusammenspiel, oder, um es mit Helmut Balzert zu formulieren [6]:

**Definition 2.1** *Unter dem Begriff Softwarearchitektur versteht man eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie die Beschreibung ihrer Beziehungen.*

Software wird demnach nicht als monolithisches Paket angesehen, sondern als ein System, das aus Teilkomponenten bzw. Teilsystemen besteht. Diese Teilsysteme erbringen gemeinsam die Funktionalität, die das Problem löst, für das die Software entwickelt wurde. Dazu müssen die Teilsysteme zusammenarbeiten. Wir werden gleich sehen, dass diese Zusammenarbeit im Falle eines verteilten Systems anders aussieht als bei einem zentralisierten System. Jedenfalls muss eine konkrete Softwarearchitektur die Fragen beantworten, wie ein System in Teilkomponenten aufgeteilt wird und wie diese Komponenten interagieren.

## 2.2 Leitgedanken zur Strukturierung von Software

Unabhängig von der konkret verwendeten Softwarearchitektur gibt es einige wenige Prinzipien, die man immer anwenden sollte, die also praktisch zu den Grundfesten des Software Engineering gehören. Diese beiden Prinzipien sind die *starke Kohärenz* und die *lose Kopplung*.

Die Idee der starken Kohärenz verdeutlicht Abbildung 2.1.

Ein gegebenes Problem wird zunächst genau analysiert, um einer späteren Implementierung zugänglich gemacht zu werden. Die Analyse ergibt normalerweise eine Aufteilung des gesamten Problemraums in klar abgrenzbare Teilprobleme. Das nun zu entwickelnde Lösungssystem ist ebenfalls in Teilsysteme aufgeteilt. Die starke Kohärenz verlangt, dass jedes Teilsystem sich mit einem klar abgegrenzten Teilproblem beschäftigt. Das in der Grafik durchgestrichene Teilsystem sollte in einer Lösung nicht auftauchen, da es sich mit unterschiedlichen Teilproblemen beschäftigt.

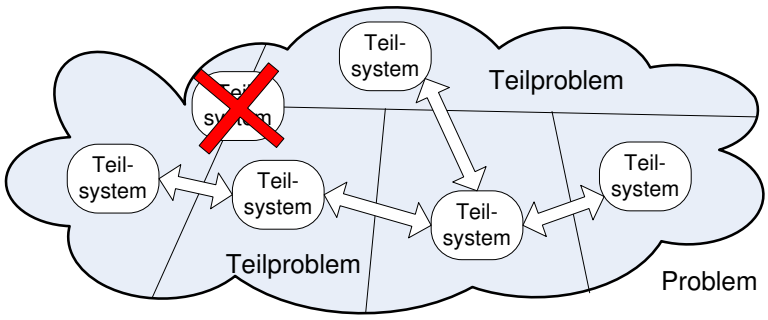


**Abbildung 2.1:** Leitgedanke der starken Kohärenz

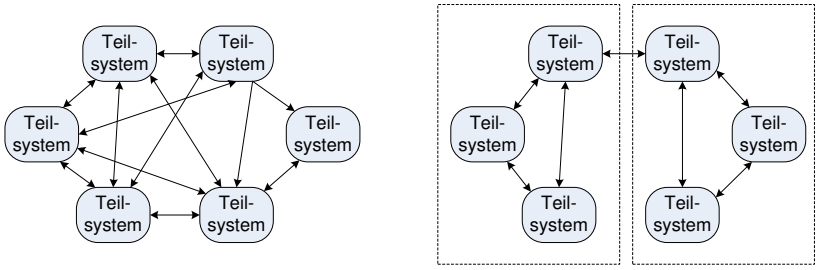


**Abbildung 2.2:** Leitgedanke der losen Kopplung

Starke Kohärenz ist kein Selbstzweck. Zunächst soll sie die Einfachheit und Verständlichkeit des Systems, im Prinzip also die Wirklichkeitsnähe fördern. Es wird aber auch in hohem Maße eine Redundanzfreiheit erreicht, da sich nicht mehrere Systeme mit demselben Teilproblem beschäftigen und jeweils eigene Lösungen erarbeiten müssen. Schließlich wird auch die Teambildung und damit die Effizienz der Erstellungsprozesse für eine bestimmte Lösung deutlich verbessert.

Abbildung 2.2 visualisiert den Gedanken der losen Kopplung. In Abbildungsteil (a) ist ein System dargestellt, das stark gekoppelt ist: jedes Teilsystem ist mit fast jedem anderen Teilsystem verbunden, nutzt also Funktionalität von dort. Die Änderung einer Schnittstelle eines Teilsystems hat dann zur Folge, dass alle anderen Teilsysteme ebenfalls angepasst werden müssen.

Die in Abbildungsteil (b) dargestellte lose Kopplung versucht hingegen, die Zahl der Beziehungen zwischen Teilsystemen zu minimieren, indem stattdessen Verbünde von Teilsystemen gebildet werden, die dann jeweils nur über eine einzige Schnittstelle, die noch dazu aus möglichst wenigen Schnittstellenfunktionen besteht, verbunden sind. Auf diese Weise wird der Änderungsaufwand minimiert. Insgesamt ist die lose Kopplung in verteilten Systemen zu bevorzugen. Diese Prinzipien sollten also beim Softwareentwurf immer eingehalten werden; sie sind ein Qualitätsmerkmal. Es gibt jedoch weitere Faktoren, die die Güte einer Softwarearchitektur beeinflussen. Darauf geht der folgende Abschnitt ein.

## 2.3 Kriterien für gute Softwarearchitekturen

Eine Softwarearchitektur ist kein Selbstzweck – sie muss vielmehr das einzusetzende System bei dieser Lösung unterstützen. Dies kann erreicht werden, indem die Architektur die Anforderungen an Funktionaliät, Dienstgüte und Lebenszy-

kluseigenschaften zu erfüllen hilft. Anders formuliert, behindert eine schlechte Softwarearchitektur das System bei der Erfüllung dieser Anforderungen.

Dies heißt aber auch, dass man eine gegebene Architekturform nicht per se als gut oder schlecht bezeichnen kann, sondern nur als gut oder schlecht geeignet für ein bestimmtes gegebenes Anwendungsproblem mit seinen Nebenbedingungen – wir besprechen dies noch ausführlich in Kapitel 10. Das Problem selbst hat demnach massiven Einfluss auf die gewählte Architektur. So kann es etwa bei einer Datenbankanwendung sinnvoll sein, das System auf geringen Speicherbedarf zu optimieren, da es sich bei den eingesetzten Endgeräten um Handys handelt. Genauso kann es aber sein, dass man stattdessen schnelle Zugriffszeiten braucht, weil man auf bestimmte Ereignisse sehr schnell reagieren können muss. Beide Anforderungen bilden in vielen Fällen sich widersprechende Ziele, die sich oftmals auch in unterschiedlichen Architekturen widerspiegeln.

Neben diesen noch nicht sehr operativen Zielen und den beiden Leitgedanken der starken Kohärenz und losen Kopplung kann man weitere Kriterien für die Güte einer gegebenen Softwarearchitektur und für die Entscheidung für oder gegen sie angeben. Ein Aspekt wurde schon bei der losen Kopplung kurz angesprochen: es ist wichtig, die Schnittstellen zu anderen Systemteilen möglichst „schmal“ zu halten und gleichzeitig möglichst abstrakt. Man sollte also nicht zu viele Funktionen an einer Schnittstelle bereit halten, da dies für den Programmierer erstens schwer überschaubar ist und zweitens meist zu einem hohen Änderungsaufwand führt. Hohe Abstraktion bedeutet, dass die Schnittstelle nicht zu stark die Implementierung widerspiegeln darf. So ist es z.B. immer gut, wenn die Implementierung einer bestimmten Anwendungslogik keine Voraussetzungen bzgl. der Darstellung mit sich bringt. Vielmehr sollte sie rein diejenigen Datenstrukturen verwenden, die allein zur Lösung des Anwendungsproblems nötig sind, aber nicht für das Layout.

Durch eine hohe Abstraktion wird dann ein weiteres wichtiges Kriterium gefördert, nämlich das der Wiederverwendbarkeit. Je besser eine Architektur die Möglichkeit unterstützt, bestimmte Komponenten in anderen Anwendungen einzubauen, desto besser. Wir werden später sehen, dass dieses Kriterium in den modernen Architekturen stark an Bedeutung gewonnen hat.

Aber auch die Wartbarkeit und die Erweiterbarkeit sind wichtige Ziele, die durch lose Kopplung und Abstraktion unterstützt werden. Es ist ja nicht so, dass ein einmal erstelltes System bis ans Ende seiner Tage unverändert weiterläuft, sondern es wird immer wieder modifziert werden; Komponenten werden ausgetauscht, neue Funktionalitäten eingebaut, etc. Hat man dann auf eine starke Kopplung mit zu konkreten (im Sinne des Gegensatzes zu abstrakten) Schnittstellen gesetzt, dann hat man sich ein erhebliches Problem eingehandelt.

Unabhängig von der Wahl der Architektur ist es jedenfalls von großer Bedeutung, diese Wahl begründet zu treffen, indem man die genannten Kriterien gegeneinander abwägt. Ein häufig angewandtes Kriterium, das aber sehr selten hilfreich ist, lautet: „Das ist eine neue Technik, die jetzt alle verwenden, die müssen wir auch

mal nehmen.“ Durch eine klare Dokumentation der Entscheidung kann man hier leicht Fehlentscheidungen vermeiden, da sich dann oft herausstellt, dass etwas ältere Lösungen besser sind, weil sie einfach besser zum System passen.

In diesem Abschnitt sollte klar geworden sein, dass die Auswahl einer konkreten Softwarearchitektur für ein System eine komplexe und sehr stark anwendungsabhängige Angelegenheit ist. Trotzdem wollen wir das Problem der Architekturwahl in diesem Buch angehen, indem wir generell die verschiedenen Architekturen vorstellen und später versuchen, Anwendungstypen zu identifizieren, die aufgrund ihrer Eigenschaften besonders gut für bestimmte Architekturen geeignet sind.

Uns interessieren in diesem Buch allerdings nicht beliebige Softwarearchitekturen, sondern diejenigen für verteilte Systeme. Neben den oben bereits genannten Anforderungen allgemeiner Softwaresysteme treten nun diejenigen, die sich aus der verteilten Ausführung eines Programmes ergeben. Was macht nun aber ein verteiltes System gegenüber einem nicht-verteilten aus? Dies soll im folgenden Abschnitt geklärt werden.

## 2.4 Die Dimensionen verteilter Systeme

Dazu muss zunächst definiert werden, was ein verteiltes System ist. Praktisch in jedem Lehrbuch zu verteilten Systemen findet sich eine eigene Definition. Eine der bekanntesten Beschreibungen stammt von Tanenbaum und van Steen [88], die ein verteiltes System vor allem aus Nutzersicht definieren:

**Definition 2.2** *Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen.*

Diese Definition ist sehr allgemein gehalten, macht aber schon eine wichtige Aussage über die Rechner, die das System bilden (unabhängig, aber sonst beliebig), und über das Verhalten gegenüber dem Benutzer – der Benutzer nimmt ein verteiltes System gerade nicht als aus Komponenten bestehend wahr, sondern als ein einziges kohärentes System. Die Kernaufgabe der Software in einem verteilten System besteht darin, diese Eigenschaft zu erfüllen. Die Softwarearchitektur muss sie darin unterstützen.

Damit die Software dieser Aufgabe gewachsen ist, muss sie verschiedene Teilaufgaben erfüllen. Sicherlich lässt sich hier darüber streiten, welche Bedeutung jede einzelne dieser Aufgaben hat und ob beispielsweise die Güte, mit der sie bewältigt wird, eine wichtige Rolle spielt. Unstrittig ist jedoch, dass es drei Dimensionen eines verteilten Systems gibt, die jede Software abdecken muss:

- Verteilung und Kommunikation
- Nebenläufigkeit
- Persistenz

Diese Punkte behandeln wir in den drei folgenden Abschnitten.

### 2.4.1 Verteilung und Kommunikation

Eine weitere bekannte Definition eines verteilten Systems stammt von Coulouris et al. [16]:

**Definition 2.3** *A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.*

Diese Definition geht schon sehr viel konkreter auf die Art und Weise ein, wie Computer miteinander kooperieren. In jedem beteiligten Rechner gibt es einen oder mehrere *Prozesse*, die miteinander kommunizieren bzw. kooperieren, indem sie *Nachrichten austauschen*. Ohne dieses Mittel können die Einzelkomponenten eines verteilten Systems also ihre Aktionen nicht abstimmen, so dass ein kooperatives Verhalten zur Zielerreichung nicht möglich ist. Es ist somit ein sehr wichtiges Merkmal, das verteilte Systeme z.B. von Multiprozessorrechnern unterscheidet – diese kommunizieren meist über andere Mittel, wie etwa gemeinsamen Speicher. Nachrichtenkommunikation ist also ein Aspekt, der in jeder Softwararchitektur für verteilte Systeme abgedeckt werden muss.

Die Verfahren zur Nachrichtenkommunikation unterscheiden sich in verschiedenen Aspekten, die wir bei der Betrachtung der einzelnen Architekturen noch genauer kennen lernen werden. Ein wichtiger Punkt besteht im „Komfort“ der Programmierung der Kommunikationsaspekte einer Anwendung. Die einfachste Schnittstelle stellt ein Programmier-Interface zur Verfügung, über das Datenströme bzw. , noch einfacher, Datenpakete übertragen werden können. Wie der Programmierer diese Datenströme mit Inhalt füllt, so dass die Daten auf der anderen Seite bei einer möglicherweise komplett anders strukturierten/programmierten Komponente auch verstanden werden, bleibt ihm überlassen. Diese Verfahren bringen also viel Programmieraufwand mit sich, sind aber meist auch sehr effizient.

Viel einfacher in der Programmierung sind die so genannten „Middleware“-Ansätze, die die Heterogenität des Gesamtsystems vor dem Programmierer verbergen. Ein wichtiges Stichwort in diesem Zusammenhang ist der so genannte „Remote Procedure Call“ (RPC): der Anwendungsprogrammierer kann entfernte Komponenten auf genau die gleiche Art und Weise aufrufen, wie er es bei lokalen Komponenten tun würde. Dieses Verfahren gibt es heute in einer Reihe von Ausprägungen; am weitesten vorangeschritten und damit am komfortabelsten zu benutzen ist es in den objektorientierten Prorgrammiersprachen.

In den letzten Jahren wurde das RPC-Verfahren weiter ausgebaut, indem die reine Kommunikation von Prozessen um weitere Möglichkeiten ergänzt wurde. So gibt es heute beispielsweise Verfahren, die eine formale Beschreibung ganzer Prozessabläufe in einer grafisch orientierten Darstellung erlauben und damit etwa eine automatische Hintereinanderausführung verschiedener externer Komponenten

gestatten. Allgemein kann man diese Verfahren unter dem Begriff der „Service-Orientierung“ zusammenfassen, der uns im weiteren Verlauf des Buches noch intensiv beschäftigen wird.

### 2.4.2 Nebenläufigkeit

Verteilte Systeme werden vor allem dadurch zu einer interessanten Form von Anwendungen, dass sie das gleichzeitige Stattfinden vieler verschiedener Aktivitäten gestatten. Wenn wie in einem traditionellen zentralisierten System alle Dinge nur nacheinander passieren könnten, wäre ein verteiltes System durch die zusätzliche Kommunikation einfach immer nur langsamer.

Die Tatsache, dass Ereignisse „gleichzeitig“ und unabhängig voneinander – man spricht dann von *nebenläufig* – stattfinden können, sorgt jedoch auch für einen höheren Koordinationsbedarf. Dies ist insbesondere dann der Fall, wenn mehrere Komponenten auf ein- und dieselbe Ressource im System zugreifen möchten. Trifft man hier keine Vorkehrungen, kann das schnell zu inkonsistenten Systemzuständen führen. Hier hat man in den verschiedenen Systemarchitekturen unterschiedliche Verfahren zur *Synchronisation* entwickelt, die wir jeweils genauer betrachten werden.

Daneben soll aber die Ressource natürlich auch möglichst effizient genutzt werden, d.h., wenn möglich im Parallelbetrieb und nicht in der sequentiellen Ausführunsgvariante. Hier gibt es programmiertechnisch schon lange Lösungen, die eine Abarbeitung von solchen gleichzeitig auftretenden Anfragen in unterschiedlichen Prozessen oder Threads (leichtgewichtige Prozesse) gestatten. Auch hier haben die verschiedenen Systemarchitekturen natürlich unterschiedliche Lösungen gefunden.

Schließlich stellt sich die Frage, was eine Komponente, die eine externe Ressource angefragt hat, so lange macht, bis die Antwort eintrifft – wenn viel Betrieb herrscht, kann das unter Umständen sehr lange dauern. Das typische und einfach zu implementierende Vorgehen besteht in einem so genannten *synchronen Aufruf*. Nachdem die anfragende Komponente die Nachricht an die Partnerkomponente abgesetzt hat, wartet sie auf die Antwort und tut derweil auch nichts anderes. Wenn man die Wartezeit jedoch auch auf der anfragenden Seite für weitere Aktivitäten nutzen möchte, dann muss ein *asynchroner Aufruf* erfolgen, der programmiertechnisch meist schwieriger umzusetzen ist. Im Wesentlichen gibt es heute zwei eingesetzte Verfahren, nämlich die Verwendung von Callback-Funktionen und das Polling. Beim Callback-Verfahren (Abbildung 2.3 (a)) ruft die ausführende Komponente den Aufrufer über eine zuvor registrierte Funktion/Prozedur (im Prozessraum des Aufrufers) zurück und übermittelt das Ergebnis, sobald es vorliegt. Beim Polling (Abbildung 2.3 (b)) fragt der Aufrufer in vorher festgelegten Zeitabständen beim Aufgerufenen immer wieder nach, ob das Ergebnis inzwischen vorliegt. Sobald dies der Fall ist, schickt der Aufgerufene das Ergebnis als Antwort auf den Poll zurück.

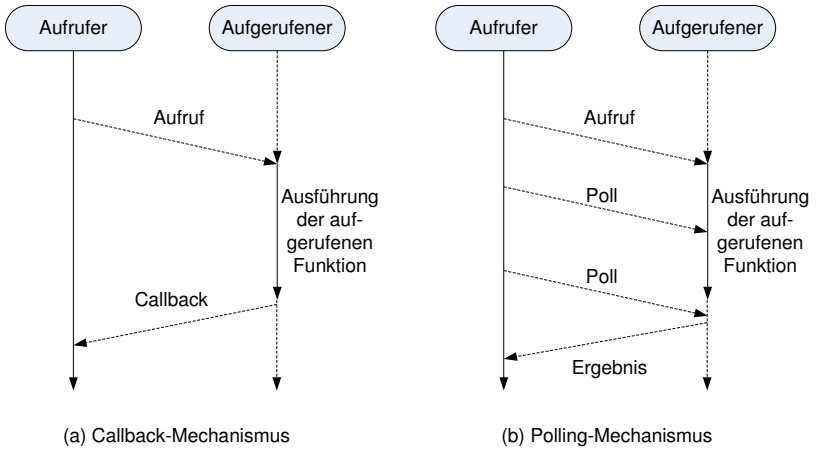


**Abbildung 2.3:** Verfahren zur Realisierung asynchroner Kommunikation

### 2.4.3 Persistenz

Bei diesem dritten Aspekt geht es um die Frage der *dauerhaften Speicherung* von Daten einer verteilten Anwendung auf nichtflüchtigem Speicher. Ziel ist eine spätere Wiederverwendung entweder durch die speichernde Komponente selbst oder durch eine andere. Als nicht-flüchtige Medien werden vor allem Magnetfestplatten eingesetzt, im Prinzip können jedoch auch Bänder, optische Medien oder andere Systeme verwendet werden.

Wie bei den beiden ersten Punkten kann man die bis heute entwickelten Verfahren wieder nach dem Konfort ihrer Benutzung einteilen. In den einfacheren Varianten muss der Programmierer typischerweise den Speicherort selbst angeben und sich auch um den korrekten Zugriff kümmern. Am anderen Ende der Skala finden sich Ansätze, die die komplette Komplexität vor dem Programmierer verbergen und nur noch Methoden zum Speichern und Laden eines Datenobjekts anbieten; alle Details der Speicherung wie Medium, Art des Zugriffs, Ort der Speicherung etc. werden von einer entsprechenden Middleware verwaltet. Die Idee ist in Abbildung 2.4 dargestellt. In diesem Buch wird auf den Bereich Persistenz im Kontext verteilter Systeme nur am Rande eingegangen. Weiterführend wäre hier beispielsweise [14].

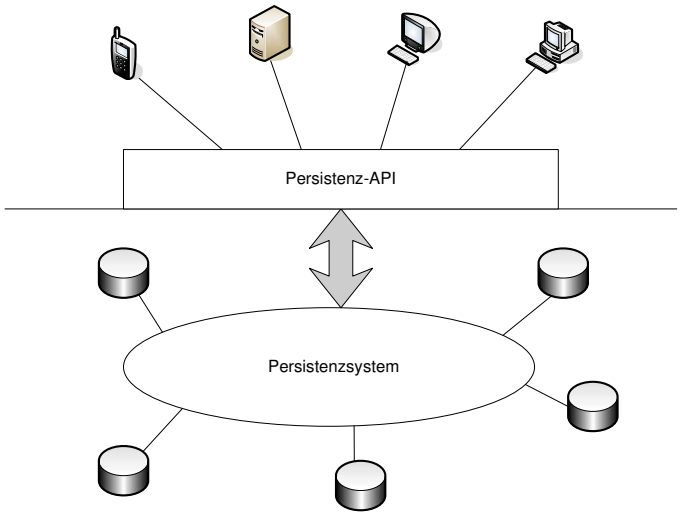


**Abbildung 2.4:** Schema eines Persistenzsystems

## 2.5 Existierende Softwarearchitekturen für verteilte Systeme

Historisch gesehen begann der Boom der verteilten Anwendungen mit der Einführung des Personal Computers und der lokalen Netze wie Ethernet und Token Ring Anfang der 80er-Jahre des 20. Jahrhunderts.[1] Plötzlich hatte jeder Anwender nicht nur ein „dummes Terminal“ für den Zugriff auf einen Großrechner auf dem Schreibtisch stehen, sondern einen Rechner, der selbst mit Ressourcen ausgestattet war. Weitere Ressourcen wie große Festplatten oder Drucker waren über Server verfügbar. So wurde es sinnvoll, neue Anwendungen durch die kombinierte Nutzung lokaler und entfernter Ressourcen zu entwickeln.

Die erste Architektur zur Nutzung dieser neuen Möglichkeiten war die *Client-Server-Architektur*. Sie basiert auf einer klaren Rollenverteilung: es gibt diejenigen Komponenten, die eine Ressource verwalten (die Server), und andere, die diese Ressource nutzen wollen (die Clients). Typischerweise ist die Kommunikation in Client-Server-Systemen sehr einfach strukturiert: ein Client sendet eine Anfrage an einen Server, der diese Anfrage bearbeitet und dann die Antwort an den Client zurücksendet.

[1] Auch wenn die Mittel dazu schon vorher vorhanden waren, wurde deren Verwendung erst zu jener Zeit populär.

Mit dem Aufkommen des World Wide Web wuchsen jedoch die Bedürfnisse, und die simple Client-Server-Struktur wurde vielfach als zu einschränkend empfunden. Als Ergebnis wurden mehrschichtige Beziehungen entworfen, die im Prinzip einer Komponente sowohl die Client- als auch die Server-Rolle zuwiesen. Während für diese Art von Architektur verschiedene Namen existieren, verwenden wir im Folgenden den Begriff *N-Tier-Architektur*. Gemeint ist im Wesentlichen, dass die Funktionalität in einem verteilten System nicht mehr nur auf zwei Schichten, nämlich üblicherweise einen Client, der die Anwendung darstellt, und einen Server, der die Daten verwaltet, verteilt ist, sondern auf mehrere. Eine erste Erweiterung führte das Drei-Schichten-System (3-Tier) ein, in dem auf dem Benutzerrechner nicht mehr der gesamte Anwendungscode, sondern nur noch eine dünne Darstellungsschicht ausgeführt wurde. Die Rolle der Server wurde verfeinert: Einerseits gab es immer noch die Daten-Provider aus dem alten System (also das Persistenzsystem), neu hinzu kam jedoch eine Server-Klasse, auf der nun der gesamte Anwendungscode lief und die an den eigentlichen Client nur noch Informationen zur Darstellung einer Seite schickte. Diese Idee wurde wenig später zum *4-Tier-System* ausgebaut, in dem die mittlere Server-Klasse noch weiter ausdifferenziert wurde: man vollzog die Trennung zwischen dem wirklich reinen Anwendungscode und der Berechnung des Layouts für die Darstellung der Seite. Die Aufgabenteilung ermöglichte eine weitere Steigerung bei der Effizienz der Anwendungserstellung. Heutige Web-Anwendungen, wie sie jeder kennt, sind überwiegend nach diesen Mustern konzipiert.

Ein wesentliches Problem dieser Architekturen besteht in ihrer sehr eingeschränkten Ausbaubarkeit. Zwar sind sie typischerweise öffentlich nutzbar, aber nur über eine ganz bestimmte Schnittstelle, nämlich das WWW. Eine massive Steigerung der Nutzbarkeit vieler Anwendungen oder auch nur Anwendungskomponenten wäre erreichbar, wenn diese Komponenten über reine Daten- oder Service-Schnittstellen eingebunden werden könnten und eben nicht über eine layoutorientierte grafische Schnittstelle, wie sie HTML-Dateien im Prinzip darstellen. Diese Überlegungen führten zur Einführung eines neuen Architekturkonzepts, den Service-orientierten Architekturen (*service-oriented architectures*, SOA). In diesem Konzept können beliebige Komponenten eines verteilten Systems mit einer Programmierschnittstelle ausgestattet und von beliebigen anderen Komponenten im System aufgerufen werden. Als Ergebnis ergibt sich eine ungeheure Steigerung der Möglichkeiten zur Kombination der Komponenten und damit zur Schaffung neuer Anwendungen. SOAs sind zum Zeitpunkt der Abfassung dieses Buches einer der Hype-Begriffe in der IT-Landschaft.

Neben diesen Mainstream-Architekturen hat sich jedoch auch eine Reihe von Nischenlösungen entwickelt. Die heute bekannteste – von einer Nische zu sprechen ist eigentlich auch nicht mehr korrekt – stellt die Idee des Web 2.0 bzw. schon bald Web 3.0 dar. Die Idee ist ganz ähnlich wie bei SOAs: Verwende existierende Softwarekomponenten, und baue sie zu neuen Anwendungen zusammen. Allerdings findet Web 2.0 nicht auf der Service-, sondern auf der grafischen Ebene statt: Komponenten von unterschiedlichsten Web-Seiten werden miteinander zu neuen

Web-Seiten und damit zu neuen Anwendungen kombiniert. Während sich SOA vor allem im Business-Umfeld und damit in einem meist abgeschlossenen Raum etabliert hat, spielt Web 2.0 heute eine wichtige Rolle im öffentlichen Internet, in dem es Anwendungen zur Verfügung stellt, die von jedermann genutzt werden können.

Weitere Nischenlösungen, die in diesem Buch dargestellt werden sollen, sind die *Ereignis-gesteuerten Architekturen* (*event-driven architectures*, EDA) sowie die Grid- und die Peer-to-Peer-Architekturen. Insbesondere Letztere sind von großem Interesse, da sie das Client-Server-Prinzip, das ja auch den N-Tier- und SOA-Architekturen zugrunde liegt, am radikalsten über Bord werfen: in einem Peer-to-Peer-System sind alle Partner gleichberechtigt; es gibt also keine Client- und Server-Rollen mehr. Am ehesten könnte man noch sagen, dass jede Komponente beide Rollen spielt, wie ein Client zwar Ressourcen nutzt, gleichzeitig aber auch wie ein Server Ressourcen anbietet. Ziel solcher Architekturen ist eine bessere Nutzung der verfügbaren Ressourcen, da auch die „Clients“ einbezogen werden.

Der folgende zweite Teil unseres Buches behandelt nun die hier nur kurz angerissenen Softwarearchitekturen im Detail.