1 Einführung in verteilte Systeme

Was ist denn eigentlich ein verteiltes System? Diese Frage wird heute in der Literatur nicht eindeutig beantwortet. Es gibt so viele verschiedene Facetten verteilter Systeme bzw. verteilter Anwendungen, dass es schwer fällt, eine allgemeingültige Definition zu finden. Einige Definitionsversuche sollen in diesem Kapitel vorgestellt werden. Da verteilte Systeme sehr viel mit den Technologien, die verwendet werden können, zu tun haben, wird im Anschluss daran eine kurze Historie der Technologieentwicklung speziell für Technologien, die in verteilten Umgebungen eingesetzt werden, gegeben.

Damit Rechner bzw. Programme miteinander über ein Netzwerk kommunizieren können, ist ein gemeinsames Verständnis über die Regeln der Kommunikation erforderlich. Alle Partner müssen einheitliche Kommunikationsprotokolle einhalten. In Referenzmodellen wie dem ISO/OSI-Referenzmodell und dem TCP/IP-Referenzmodell werden die Grundlagen festgelegt. Die Kenntnis der grundlegenden Mechanismen und der entsprechenden Referenzmodelle, insbesondere des TCP/IP-Referenzmodells wird für die weiteren Betrachtungen weitgehend vorausgesetzt (siehe hierzu Tanenbaum 2003 und Mandl 2008b).

Nach der Einführung werden wichtige Designziele verteilter Systeme, wie Lastverteilung, Skalierbarkeit, Ausfallsicherheit und Flexibilität diskutiert. Es wird auch erläutert, dass Transparenz in verschiedenen Varianten für die Entwicklung und Nutzung verteilter Systeme sehr wichtig ist. Die Probleme der Heterogenität und weitere Aspekte, die in verteilten Umgebungen gelöst werden müssen, werden ebenfalls erörtert.

Zielsetzung des Kapitels

Ziel dieses Kapitels ist es, eine Einführung in die Problematik der verteilten Systeme zu geben. Der Studierende soll grundlegende Begriffe verteilter Systeme kennenlernen und Technologien für die Entwicklung verteilter Systeme einordnen können.

Wichtige Begriffe

Verteiltes System, verteilte Anwendung, Transparenz und Heterogenität, Persistenz, Transaktionssicherheit, Fehlersemantiken, Maybe, At-Most-Once, At-Least-Once, Exactly-Once

1.1 Definitionen und Festlegungen

Heute sind fast alle betrieblichen Informationssysteme in der einen oder anderen Form verteilt, wobei unterschiedlichste Technologien eingesetzt werden. Dieser Abschnitt soll nach einer kurzen Einführung einiger Begriffe und einer Klärung, was man unter einem verteilten System heute versteht, eine kurze Geschichte zu den Technologien für die Entwicklung verteilter Systeme geben.

1.1.1 Grundlegende Begriffe

Für unsere weiteren Betrachtungen sollen zunächst einige Begriffe erläutert werden, die im Umfeld der betrieblichen Informationssysteme häufig verwendet werden. Hierzu gehören die Begriffe *System*, *Informationssystem*, *Technik* und *Technologie*.

System und Informationssystem: In der Informatik wird oft von Systemen gesprochen. Ganz allgemein wird der Begriff *System* in unterschiedlicher Bedeutung verwendet. Im Prinzip ist allerdings immer die "Zusammenstellung" mehrerer Elemente, die untereinander in Wechselwirkung stehen, gemeint.

Ein *Informationssystem* dient dagegen prinzipiell der rechnergestützten Erfassung, Speicherung, Verarbeitung, Verwaltung, Pflege usw. von Information. Es besteht aus Hardware (Rechner oder ein ganzer Rechnerverbund), Datenbank(en), Software (System- und Anwendungssoftware), Daten und deren Anwendungen. Informationssysteme sind soziotechnische Systeme, die auch aus einzelnen Teilsystemen bestehen können, und für die optimale Bereitstellung von Information und (technischer) Kommunikation dienen. Diese Beschreibung lässt viel Spielraum für Interpretationen.

Ein *betriebliches Informationssystem* (Business Information System) ist ein Informationssystem, das sich in erster Linie mit betrieblichen Funktionen und Daten befasst, um diese effizient zu bearbeiten und bereitzustellen.

In der praktischen Informatik bezeichnet man auch ein Rechnersystem mit Hard- und Software einfach kurz als System und spricht zum Beispiel von einer Systemarchitektur, wenn Hardware und Software gleichermaßen in die Architekturbetrachtung eines Informationssystems mit einbezogen werden.

Technik und Technologie: Unter *Technik* versteht man nach (WWW-038) Verfahren und Fähigkeiten zur praktischen Anwendung der Naturwissenschaften und zur Produktion industrieller, handwerklicher oder auch künstlerischer Erzeugnisse. Technik kann auch als die Fähigkeit des Menschen verstanden werden, Naturgesetze, Kräfte oder Rohstoffe zur Sicherung seiner Existenzgrundlage sinnvoll einzusetzen oder umzuwandeln.

Technologie ist nach (WWW-038) dagegen die Wissenschaft vom Einsatz der Technik im engeren Sinne, in der es um die Umwandlung von Roh- und Werkstoffen in fertige Produkte und Gebrauchsartikel geht, im weiteren Sinne geht es aber auch

um Handfertigkeiten und Können. Eine andere Definition bezeichnet mit Technologie die Gesamtheit der Verfahren zur Produktion von Gütern und Dienstleistungen, die einer Gesellschaft zur Verfügung steht. Technologie beinhaltet die Komponenten der Technik (Werkzeuge, Geräte, Apparate), die materiellen und organisatorischen Voraussetzungen und deren Anwendung. Häufig wird Technologie in der Praxis auch einfach anstelle von Technik verwendet. Spricht man im Zusammenhang z.B. bei Fahrzeugen von neuester eingesetzter Technologie, ist nur die Technik gemeint.

Unter *High-Tech* versteht man hochentwickelte Technologien, die neueste wissenschaftliche Erkenntnisse umsetzen, so beispielsweise die Produktion von CPUs oder bakteriell hergestelltes Insulin. Im Gegensatz dazu bezeichnet Low-Tech absichtlich möglichst einfache, ausfallsichere Technologien (bei deren Entwicklung natürlich auch neueste wissenschaftliche Erkenntnisse zum Einsatz kommen können), die dadurch einfach in Herstellung, Anwendung oder Wartung sind.

1.1.2 Was sind verteilte Systeme?

Was genau sind „verteilte Systeme“? Auch dieser Begriff ist in der Informatik nicht einheitlich definiert, obwohl er - insbesondere in der Praxis - sehr häufig verwendet wird. Mehrere Definitionen bzw. Beschreibungen für verteilte Systeme sind in der Literatur zu finden. Einige davon sollen hier erwähnt werden:

Tanenbaum definiert ein verteiltes Systen wie folgt (Tanenbaum 2003): „Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen“.

Bengel’s Definition eines verteilten Systems lautet folgendermaßen (Bengel 2004): „Ein verteiltes System ist definiert durch eine Menge von Funktionen oder Komponenten, die in Beziehung zueinander stehen (Client-Server-Beziehung) und eine Funktion erbringen, die nicht erbracht werden kann durch die Komponenten alleine.“[1]

(Kurmann 2004) beschreibt ein verteiltes System in einer „Arbeitsdefinition“ wie folgt: „Ein verteiltes System ist ein Softwaresystem, welches die Verteilung einzelner Bausteine eines Anwendungssystems auf verschiedene, räumlich getrennte Knoten eines Netzwerks, unterstützt. Einzelne Teile des Systems können dezentral administriert werden und kommunizieren, um gemeinsam Aufgaben zu bewältigen. Ein verteiltes System unterstützt nebenläufige Verarbeitung von Geschäftsprozessen und ist sowohl für den Anwender als auch für den Entwickler transparent“.

---

[1] Eine Client-Server-Beziehung ist nach Meinung des Autors nicht zwingend erforderlich. Wie wir noch sehen werden, gibt es auch genügend Anwendungssysteme, in denen die einzelnen Komponenten auf unterschiedliche Weise miteinander kommunizieren.

Über die Definitionen lässt sich wie immer diskutieren. Der Begriff der Transparenz wird z.B. in diesem Zusammenhang recht unterschiedlich benutzt. Einem Entwickler muss z.B. durchaus bewusst sein, auf welchen Systemen die einzelnen Komponenten ablaufen. Auch ist die Unterstützung von Nebenläufigkeit keine zwingende Notwendigkeit, obwohl sie in heutigen verteilten Systemen meist üblich ist.

Uns soll im Weiteren die Definition von Tanenbaum für *verteilte Systeme* genügen, da sie am flexibelsten ist. Die Begriffe *verteiltes System* und *verteiltes Anwendungssystem* verwenden wir in diesem Dokument als Synonyme, da es aus unserer Sicht in erster Linie um Anwendungssysteme geht. Wir erweitern daher die Definition von Tanenbaum und nehmen diese als Basis für unsere Betrachtungen:

Verteiltes System: Ein *verteiltes System* basiert auf einer Menge voneinander unabhängiger Rechnersysteme und Softwarebausteine, die dem Benutzer wie ein einzelnes, kohärentes System bzw. Anwendungssystem erscheinen. Jeder Softwarebaustein einer verteilten Anwendung kann auf einem eigenen Rechner liegen. Es können aber auch mehrere Softwarebausteine auf dem gleichen Rechner installiert sein.

Typische verteilte Systeme sind z.B. Webanwendungen, das E-Mailsystem im Internet, das Domain Name System (DNS) für die Verwaltung der IP-Adressen, verteilte Datenbankanwendungen, verteilte Dateisysteme wie das Network File System (NFS) von Sun Microsystems und das WWW. Dies sind verteilte Anwendungssysteme, die im Wesentlichen auf einem intelligenten Protokoll der Anwendungsschicht basieren und von der Komplexität her klar umrissen sind.

Verteiltes Informationssystem: Wir beschäftigen uns vorwiegend mit *verteilten Anwendungssystemen* im betrieblichen Umfeld bzw. verteilten Informationssystemen, bei denen einzelne Funktionen auf mehrere Rechner (mindestens zwei) verteilt sind, also verteilte Systeme, die in betrieblichen Informationssystemen relevant sind. Verteilte Informationssysteme sind spezielle, meist komplizierte verteilte Systeme, die sich nach (Hammerschall 2005) durch folgende Eigenschaften auszeichnen:

- Verteilte Informationssysteme sind meist sehr groß, was den Codeumfang anbelangt (mehrere 100.000, teilweise mehrere Millionen Lines of Code).
- Verteilte Informationssysteme sind sehr datenorientiert, d.h. die Datenhaltung steht im Zentrum der Anwendung. Üblicherweise werden die Daten in einer Datenbank verwaltet. Die zugrundeliegenden Datenmodelle sind meist sehr umfangreich.
- Verteilte Informationssysteme sind extrem interaktiv und verfügen (neben Hintergrund- und Batch-Funktionalität) überwiegend über graphische Benutzerschnittstellen.
- Verteilte Informationssysteme sind meistens auch sehr nebenläufig, was sich durch eine große Anzahl an parallel arbeitenden Benutzern äußert.

Hinzu kommt, dass derartige Anwendungen oft auch sehr unternehmenskritisch sind. Fällt beispielsweise ein Warenwirtschaftssystem eines Versandhändlers oder ein Core-Banking-System über längere Zeit aus, ist dies durchaus existenzbedrohend für das Unternehmen. Verteilte betriebliche Informationssysteme stehen auch im Mittelpunkt unserer weiteren Betrachtung. Derartig komplexe Systeme benötigen eine entsprechend komfortable Infrastruktur mit Transaktions-, Sicherheits- und sonstigen Diensten.

In anderen, mehr technischeren Bereichen (z.B. in Automobilen) entstehen derzeit ebenfalls immer komplexere verteilte Anwendungssysteme. In manchen Fahrzeugen kommunizieren mehr als 50 Microcontroller bzw. die darauf laufenden Softwarebausteine über verschiedene Bussysteme miteinander. Hier handelt es sich auch um hochkomplexe verteilte Systeme, die wir aber in unseren Ausführungen weniger betrachten.

In Abgrenzung hierzu sind Netzwerkbetriebssysteme und verteilte Betriebssysteme zu nennen:

Verteilte Betriebssysteme verteilen Betriebssystemfunktionalität so, dass das gesamte Netz wie ein Rechnersystem erscheint. Für den Anwender ist es nicht mehr ersichtlich, wo welche Ressourcen verwaltet werden (siehe Abbildung 1-1).

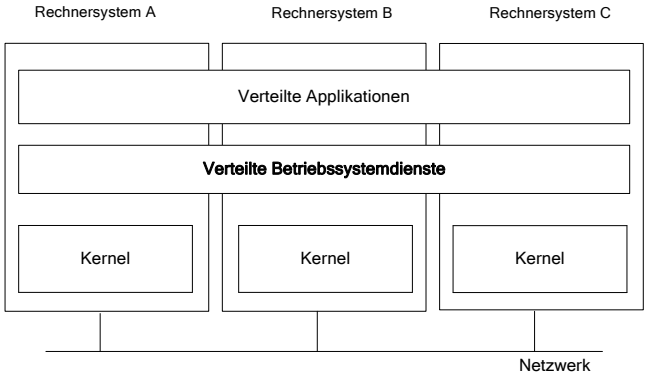

Abbildung 1-1: Verteiltes Betriebssystem nach Tanenbaum 2003

Netzwerkbetriebssysteme liegen über den lokalen Betriebssystemen und stellen Dienste wie den Zugriff auf entfernte Ressourcen (Dateien, Drucker) im Netzwerk zur Verfügung. Sie sind heute nicht mehr so verbreitet, als das Novell-System die PC-Netze eroberte. Diese Funktionalität ist heute schon in den Standardbetriebssystemen enthalten. In Abbildung 1-2 ist der Einsatz eines Netzwerkbetriebssystems skizziert.

Verteilte Betriebssysteme haben sich in der Praxis nie richtig durchgesetzt. Verteilte Systeme allerdings - und hier sind vor allem Anwendungssysteme gemeint -

sind heute fast überall im Einsatz. Sie sind im Internet genauso zu finden wie in Intranets und es spielt im Prinzip keine Rolle, welche Definition nun am besten passt.

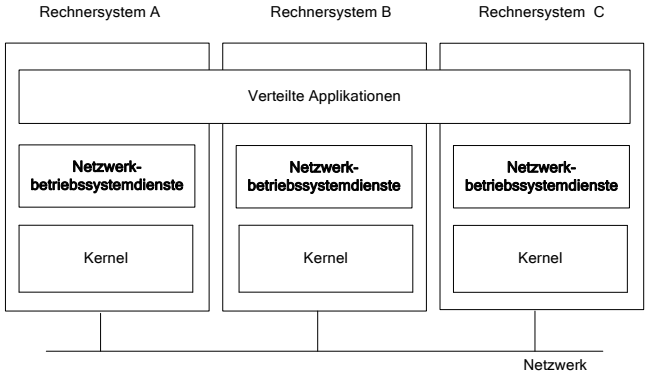

Abbildung 1-2: Netzwerkbetriebssystem

1.1.3 Klassifizierung verteilter Systeme

Eine Typisierung bzw. Klassifizierung verteilter Systeme wurde schon mehrfach versucht. Tanenbaum unterscheidet beispielsweise grob nach verteilten Computersystemen (Distributed Computing Systems), verteilten Informationssystemen (Distributed Information Systems) und verteilten pervasiven Systemen (Distributed pervasive Systems, pervasive = überall vorhanden, durchdringend) (Tanenbaum 2007):

– Unter *verteilten Computersystemen* versteht Tanenbaum verteilte Systeme für Hochleistungsaufgaben und ordnet hier *Cluster*[2]-Computersysteme und *Grid*-Computersysteme ein. Cluster-Computersysteme bestehen aus einer Menge ähnlich aufgebauter Rechnersysteme, die üblicherweise in einem LAN miteinander vernetzt sind und Komponenten einer verteilten Anwendung enthalten. Unter Grid-Computersystemen kann man auch eine Menge (evtl. sogar eine sehr große Menge) von Rechnersystemen verstehen, die weiter verteilt und zum Teil in sehr unterschiedlichen Adminstrationsdomänen stehen und gemeinsam eine verteilte Anwendung unterstützen. Typische Grid-basierte Anwendungen finden sich heute nur noch selten und dann vor allem für rechenintensive Aufgaben. Man versucht auch, die vielen Rechner im Internet, die ja die meiste Zeit ihre CPUs fast gar nicht nutzen, in

[2] Cluster = number of things of the same kind growing closely together (Bauke 2006).

Grids[3] zusammenzuschalten. Ein Beispiel für eine Grid-Anwendung ist das *Earth System Grid Project*, in dem Simulationen für die Klimaforschung durchgeführt werden. Cluster- und Grid-Computersysteme zeichnen sich durch eine sehr gute Skalierbarkeit, durch gute Möglichkeiten der Lastverteilung und durch hohe Fehlertoleranz aus.

- Unter *verteilten Informationssystemen* versteht Tanenbaum die bereits oben angesprochenen klassischen betrieblichen Anwendungen, die Transaktionen ausführen. Diese Anwendungssysteme stehen im Fokus unserer Betrachtung. Insbesondere Transaktionsanwendungen werden wir noch ausführlich besprechen.
- *Verteilte pervasive Systeme* sind kleine oder auch sehr kleine, oft batteriegetriebene Systeme und Systeme, die auch mobil sein können. Hierunter fallen Sensoren-Systeme für das Gesundheitswesen etwa zur Überwachung von Körperfunktionen (Herz-Kreislauf usw.). Diese Systeme erfreuen sich heute bereits zunehmender Verbreitung. Man spricht in diesem Zusammenhang auch von *Ubiquitous Computing* (kurz: Ubicomp). Ubicomp beschäftigt sich mit Computer-Systemen, die unauffällig und unsichtbar agieren, in Alltagsgegenständen vorhanden sind und intelligente Funktionen übernehmen. Für die Entwicklung derartiger Systeme gibt es heute noch keine Standards, da sie sich noch weitgehend im Forschungsstadium befinden.

Andere Klassifizierungsversuche ordnen verteilte Systeme anhand ihrer Architekturmerkmale. In (Dustdar 2003) wird beispielsweise eine Einordnung nach Architekturstilen vorgenommen. Es wird zwischen datenzentrierten Architekturen, datenflussorientierten Architekturen, Call-And-Return-Architekturen, unabhängigen Komponenten-Architekturen und auch virtuellen Maschinen-Architekturen unterschieden. Call-And-Return-Architekturen, unabhängige Komponenten-Architekturen sowie hierfür vorhandene Technologien werden wir im Rahmen unserer Betrachtung noch genauer untersuchen. Eine allgemein anerkannte Klassifizierung verteilter Systeme liegt aber nicht vor. Wie wir im Weiteren noch sehen werden, nutzen heutige verteilte Anwendungen meist mehrere Architekturstile und auch die Typen nach Tanenbaum sind in der Praxis nicht eindeutig abzugrenzen. Beispielsweise werden in heutigen verteilten Informationssystemen auch Cluster-Ansätze verwendet. Cluster findet man heute beispielsweise häufig in internetbasierten Verkaufssystemen vor, die sehr viele Anwender bedienen müssen.

1.1.4 Beispiel einer verteilten Systemlandschaft

Betrachten wir ein Beispiel aus dem Umfeld der betrieblichen Informationssysteme und zwar speziell eine (völlig unvollständige) Anwendungslandschaft eines Großhändlers, der seine Unternehmensdaten in einem ERP-System (Enterprise Resour-

[3] In Analogie zum Stromnetz kann man sich ein Computer-Grid auch so vorstellen, dass man Rechenleistung aus der Steckdose oder über den Internetanschluss bezieht.

ce Planning) verwaltet und der ein großes Lagersystem unterhält, das über IT-Systeme gesteuert wird. In Abbildung 1-3 ist die logische Sicht auf die Anwendungssysteme und deren Zusammenspiel dargestellt.

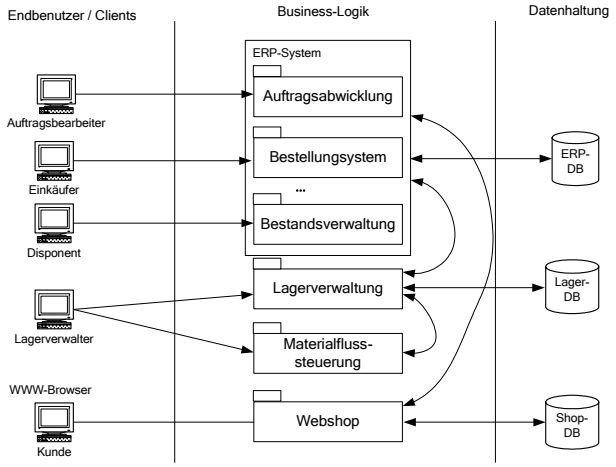

Abbildung 1-3: Beispiel einer verteilten Anwendung (logische Sicht)

In einem ERP-System werden u.a. die Teilsysteme zur Auftragsabwicklung, das Bestellwesen und die Bestandsverwaltung abgewickelt. Die Lagerverwaltung wird über ein eigenes Lagerverwaltungssystem ausgeführt, das wiederum über mehrere Subsysteme verfügt (im Bild nicht dargestellt). Die Materialflusssteuerung für die unterlagerten logistischen Systeme wie Fördertechnik und automatische Lager werden über ein Materialflusssystem bedient. Zudem wird den Kunden über einen Webshop eine direkte Bestellmöglichkeit über das Internet angeboten. Insgesamt werden drei disjunkte Datenbanken verwaltet. Die einzelnen Teilsysteme kommunizieren bei Bedarf miteinander. Die Clients bedienen verschiedene Benutzerrollen und verfügen entweder über eine klassische grafische Oberfläche oder über einen Web-Zugang.

In der logischen Sicht ist noch keine Aussage über die Verteilung der Teilsysteme genannt. In einer exemplarischen physikalischen Architektur wird diese Verteilung skizziert (siehe hierzu Abbildung 1-4). Die ERP-Teilsysteme und auch die Lagerverwaltung liegen auf einem doppelt ausgelegten und ausfallgesicherten ERP- bzw. einem LVS-Serverrechner. Die Datenhaltung für die beiden Datenbanken ERP-DB und Lager-DB wird in einem ebenfalls ausfallgesicherten Datenbankserver abgewickelt. Der Webshop wird von einem eigenen Shop-Server bedient, der auch die Datenhaltung übernimmt. Die Web-Zugriffe werden über einen Webserver an das Shopsystem weitergeleitet.

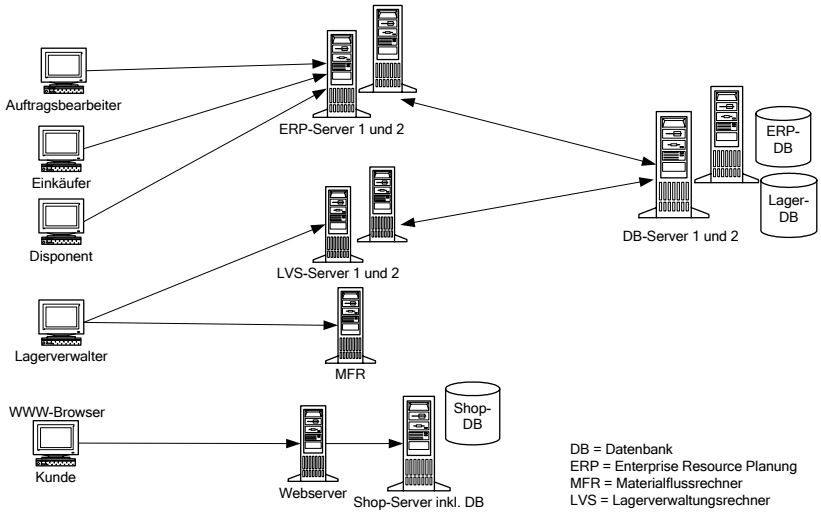

Abbildung 1-4: Beispiel einer verteilten Anwendung (physikalische Sicht)

Die Kommunikation der Rechner erfolgt über das interne Netzwerk des Unternehmens bzw. die WWW-Clients greifen über das Internet an den unternehmenseigenen Webserver. Wie die einzelnen Rechnersysteme miteinander kommunizieren und auf welche Basismechanismen und Technologien sie zurückgreifen, ist im Bild nicht dargestellt. Mehrere Varianten sind möglich und sollen auch noch diskutiert werden. Eines ist aber deutlich sichtbar: Die Systeme stehen teilweise in einer Client-Server-Beziehung. Dieses Grundmodell der Kommunikation wird noch ausführlich diskutiert.

In der Praxis wird die anwendungsübergreifende Kommunikation heute noch vielfach über den Austausch von Dateien in eigenen Formaten oder Standardformaten ausgeführt (EDIFACT, SWIFT, XML-Schemata, CSV-Dateien,...). Dieses Vorgehen hat den großen Vorteil, dass die kommunizierenden Anwendungssysteme keine technischen Abhängigkeiten haben und weitgehend unabhängig voneinander arbeiten können. Insbesondere bei der unternehmensübergreifenden Kommunikation ist eine Prozess-Prozess-Beziehung selten anzutreffen. Hier handelt es sich aber auch um eine verteilte Verarbeitung. Beispiele hierfür sind:

- Bestellaufträge werden über eine EDIFACT- oder XML-basierende Schnittstelle vom Besteller zum Auftraggeber gesendet.
- Banken tauschen für das Clearing ihre Umsatzdaten über sog. Datenträger aus.

1.2 Historische Technologie-Betrachtung

Aus unserer Sicht stehen die Softwarekonzepte für verteilte Systeme (Anwendungssysteme) insbesondere für die oben erwähnten Call-And-Return-Architekturen und für die unabhängigen Komponenten-Architekturen im Vordergrund. In den letzten Jahrzehnten ist hier sehr viel Pionierarbeit geleistet worden. Es wurde und wird auch in der Praxis viel Zeit und Geld in die Evaluation neuer Technologien gesteckt. Viele Projekte werden aber auch durch nicht ausgereifte Produkte für Basissysteme erschwert.

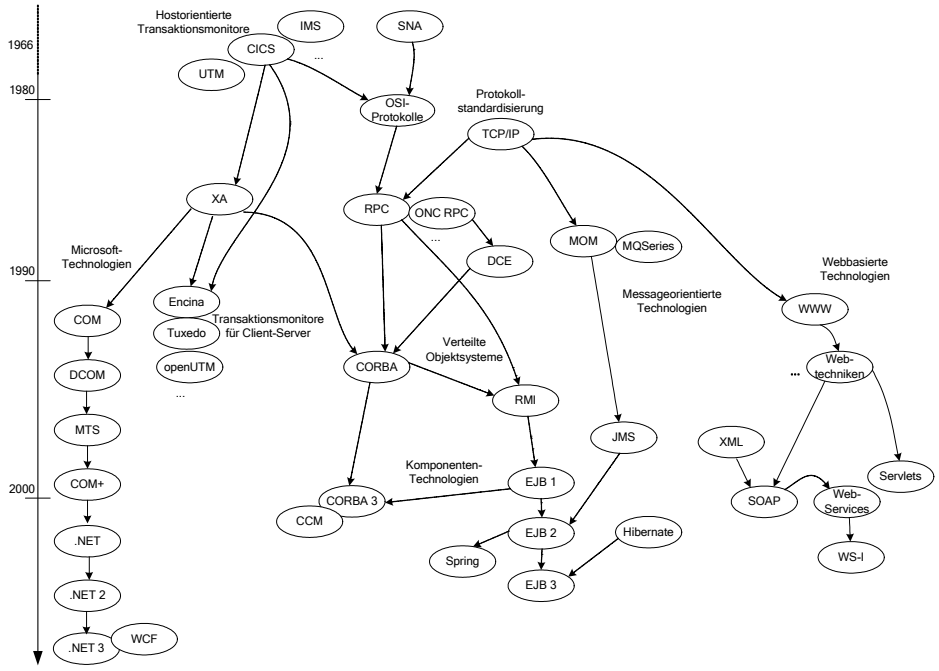

Abbildung 1-5: Entwicklung von Modellen/Technologien für verteilte Systeme

In Abbildung 1-5 ist der zeitliche Ablauf skizziert, wobei natürlich nicht alle, sondern nur die aus unserer Sicht wesentlichen „Erfindungen" eingezeichnet sind. Die Pfeile zwischen den Knoten deuten Einflüsse bzw. Weiterentwicklungen bestehender Technologien und Modelle an. Teilweise beeinflussen sich die Technologien gegenseitig, da die einzelnen Hersteller bzw. Organisationen voneinander profitieren. Wir beginnen mit den 80er Jahren.

Entwicklung in den 80er Jahren:

Ein wesentlicher Grund für die Dezentralisierungsbemühungen waren in den 80er Jahren des letzten Jahrtausends die hohen Kosten für Mainframes, ein anderer natürlich die hohe Erwartungshaltung bzgl. der Flexibilität neuer verteilter Systeme. Die Verbreitung von PCs brachte in diesem Jahrzehnt schließlich auch die ersten Netzwerke und Netzwerkbetriebssysteme (siehe Novell Netware) mit sich. Zunächst wurden Dienste wie File- und Druckersharing realisiert und nach und nach wollte man auch komplexere Anwendungssysteme auf Serverrechnern außerhalb der Hostsysteme platzieren. Datenbanksysteme auf eigenen Datenbankservern verbreiteten sich.

Die Standardisierung der Kommunikation zwischen Rechnersystemen wurde im Rahmen der ISO/OSI-Spezifikationen (OSI = Open Systems Interconnection) und der TCP/IP-Entwicklung in dieser Zeit stark vorangetrieben. Der Begriff des *offenen Systems* wurde durch die ISO/OSI-Spezifikationen geprägt. Ein offenes System ist nach OSI kein reales Rechnersystem, sondern allgemein ein System, dessen Komponenten sich dem OSI-Modell entsprechend verhalten. Ein System ist dann offen, wenn es nach außen ein nach OSI genormtes Verhalten zeigt. Diese Definition zeigt, dass hier der Grundstein für das Zusammenspiel von verteilten Systemen mehrerer Hersteller gelegt werden sollte.

Die Transportzugriffsschnittstelle *Sockets* und zeitweise *XTI* (Extendent Transport Interface) sowie *TLI* (Transport Layer Interface) haben sich als Standards für Transportzugriffssysteme entwickelt, wobei letztendlich Sockets für den Einsatz in der Praxis übrig blieb. Das Client-Server-Modell wurde Mitte der 80er Jahre herausgearbeitet. Remote-Procedure-Call-Implementierungen (RPC) entstanden. Schließlich wurde von der damaligen Open Software Foundation (OSF), einer Herstellervereinigung, die verschiedene Standards festzulegen versuchte, das Distributed Computing Environment (DCE) als Basis für die Entwicklung verteilter Anwendungen standardisiert. Das grundlegende Kommunikationsparadigma war auch hier der Remote Procedure Call (DCE RPC).

Entwicklung in den 90er Jahren:

Die 90er Jahre waren vor allem durch die Weiterentwicklung des Client-Server-Modells und der Remote-Procedure-Call-Mechanismen (RPC) geprägt (siehe vor allem Sun RPC). OSF DCE wurde um eine ganze Reihe von Standarddiensten (Time Service, Thread Service, Directory Service, Security Service,...) erweitert, die die Entwicklung verteilter Anwendungen erleichtern sollten.

Mit der Object Management Architecture (OMA) und der Common Object Request Broker Architecture (CORBA) wurden schließlich von der Object Management Group (OMG), einem Herstellerkonsortium, das in dieser Zeit die Entwicklung stark beeinflusste, Standards für objektbasierte verteilte Systeme entwickelt, die das „prozedurale“, C-basierte OSF DCE sehr schnell in Vergessenheit geraten lie-

sen. OMA (Object Management Architecture) und CORBA (Common Object Request Broker Architecture) spezifizierten eine Fülle von Diensten (Eventing Service, Transaction Service, Concurrrency Control Service, Persistency Service,...), die für verteilte Anwendungen nützlich sein sollten, und das auf Basis eines verteilten Objektmodells.

Sowohl DCE als auch CORBA trugen wesentlich dazu bei, dass sich die Softwarekonzepte für verteilte Anwendungen und deren Basistechnologien immer mehr durchsetzten. Aber auch CORBA wurde schnell wieder überboten, als man erkannte, dass die Entwicklung verteilter Anwendungen immer noch zu komplex und fehleranfällig war. Sun Microsystems brachte einen leichtgewichtigen Mechanismus zur objektbasierten Kommunikation in Java heraus, der als Java RMI (Remote Method Invocation) bezeichnet wurde. Parallel dazu entwickelte Microsoft verteilte Plattformdienste und komponentenbasierte Systeme wie COM (Component Object Model), DCOM (Distributed Component Object Model), COM+ und MTS (Microsoft Transaction Server).

Nebenbei verbreitete sich das World Wide Web mit seinen mittlerweile unzähligen Techniken zur Entwicklung von Webanwendungen von einer zunächst einfachen Informationsplattform immer weiter und ermöglichte komplexe Anwendungen. Webtechniken wie CGI, PHP, Java Servlets, Microsoft ASP usw. kamen auf den Markt und wurden vielfach eingesetzt.

Entwicklung seit dem Jahr 2000:

Ende der 90er und mit Beginn des neuen Jahrtausends wurden komponentenorientierte verteilte Systeme moderner, die die Objektorientierung nicht mehr so ganz in den Vordergrund rückten. Verteilte Komponenten, die man sich etwas größer als verteilte Objekte vorstellen kann, sollten möglichst einfach mit einer Schnittstelle versehen werden können und in einer komfortablen Umgebung zum Ablauf gebracht werden.

Aus der Java-Welt heraus wurde insbesondere von Sun Microsystems J2EE (Java 2 Extended Edition) mit ihren vielen Programmierschnittstellen und insbesondere Anfang 2000 die EJB-Spezifikation (Enterprise Java Beans) in ihrer ersten Version entwickelt, die heute als Basis für die Realisierung sog. EJB-Application-Server, eine Infrastruktur für serverseitige Komponenten, dient. EJB-Application-Server sind im Wesentlichen moderne Transaktionsmonitore für verteilte Systeme.

Parallel dazu wurden auch gesicherte Message-Queuing-Systeme weiterentwickelt. Ein typisches Produkt dieses Typs ist heute Websphere MQ von IBM. Derartige Produkte werden auch unter dem Begriff *MOM* (Message-orientierte Middleware) zusammengefasst.

Das WWW und die Webtechniken entwickelten sich ständig weiter. Komplexeste dynamische Webanwendungen wurden und werden realisiert und Webservices in Verbindung mit SOAP (Ursprünglich Kurzbezeichnung für *Simple Object Access*

Protocol) und XML sowie der Begriff der Service-orientierten Architektur (SOA) wurden erfunden. Teilweise entstand der Eindruck, dass man mit XML und Webservices alle Probleme lösen könnte. Aber das war natürlich nicht ganz korrekt.

Die Konzepte komponentenorientierter verteilter Systeme wurden weiterentwickelt. Eine „Standardisierung" von CORBA CCM (CORBA Component Model) mit CORBA V3.0 wurde durch die OMG durchgeführt. Microsoft brachte mit der .NET-Initiative ein entsprechendes Gegenstück heraus, in dem alle ihre gewachsenen Bausteine verteilter Systeme (COM, DCOM, COM+, …) vereint werden sollten. Im Mittelpunkt von .NET wurde vor allem auch die neue Sprache C# als Konkurrent zu Java platziert. Neue Ansätze der verteilten Kommunikation unter Windows sind im Windows Communication Framework (WCF) auf den Markt gekommen.

CORBA und vor allem J2EE/EJB sowie .NET mit WCF werden heute ständig weiterentwickelt. CORBA scheint allerdings nur noch für heterogene Lösungen und als Plattform-interne Technologie in Middlewareprodukten eingesetzt zu werden. Das Komponentenmodell CCM wurde bis heute kommerziell noch nicht implementiert. Die beiden praktisch relevanten, komponentenbasierten Basissysteme sind daher J2EE/EJB und .NET Enterprise Services. Aber auch Webservices scheinen sich für bestimmte Aufgabenstellungen durchzusetzen. MOM ist für bestimmte Anwendungen ebenfalls von großer Bedeutung und im Zuge der Vereinfachung wurde das Spring-Framework als leichtgewichtige Konkurrenz zu EJB entwickelt und erfreut sich zunehmender Beliebtheit (WWW-035).

Gerade in den letzten Jahren ist allerdings nach einer starken Euphorie eine leichte Ernüchterung eingetreten, da man erkannte, dass verteilte Systeme wesentlich aufwändiger in der Administration sind als zentralisierte Systeme. Die Software muss verteilt, die Serversysteme müssen überwacht und die vielen Clientsysteme müssen ständig aktuell gehalten werden. In der Praxis hat man viel über die Vor- und Nachteile der Verteilung gelernt. Man ist dazu übergegangen, auf eine gezielte und kontrollierbare („administrierbare") Verteilung zu setzen. Die im letzten Jahrzehnt entstandenen IT-Infrastrukturen brachten eine Inflation an Serversystemen mit sich, die heute wieder auf wenige, leistungsfähige (und damit Mainframeähnliche) Systeme konsolidiert werden. Dieser Trend der Serverkonsolidierung ist aktuell zu erkennen. Allerdings sind die Vorteile einer vernünftigen Verteilung so gravierend, dass eine Rückkehr zu reinen Mainframe-basierten Systemen sehr unwahrscheinlich ist.

1.3 Gründe für die Verteilung betrieblicher Informationssysteme

Dieser Abschnitt stellt einige Gründe für eine Verteilung von Softwarebausteinen vor. Nicht immer sind alle Gründe relevant, aber vor einer Entscheidung, ob ein System verteilt sein soll oder nicht, ist es nützlich, über die genannten Aspekte nachzudenken. Es werden heute mehrere Gründe für den Einsatz von verteilten

Systemen genannt. Manche gelten uneingeschränkt, andere nur unter bestimmten Randbedingungen. Die wesentlichen Gründe sollen zunächst im Überblick aufgezeigt werden:

- Lastverteilung
- Ausfallsicherheit
- Skalierbarkeit
- Flexibilität
- Transparenz

Lastverteilung: Lastverteilung bedeutet, dass man gewisse Systembausteine auf mehrere Rechnersysteme verteilen kann. Dadurch kann ein Leistungsgewinn erreicht werden. Beispielsweise ist es möglich, einen Serverbaustein mehr als einmal laufen zu lassen und die Anfragen der Clients (siehe auch Client-Server-Modell) auf die verschiedenen Instanzen zu verteilen. Damit kann auch die Netzbelastung beeinflusst werden. Man spricht in diesem Zusammenhang oft auch von Cluster-Lösungen, wobei ein Cluster aus mehreren Rechnersystemen besteht, die der Lastverteilung dienen.

Skalierbarkeit: Skalierbarkeit ist ein wichtiger Aspekt, der für verteilte Systeme spricht. Man kann - bei entsprechender Anwendungsarchitektur – ein System so konzipieren, dass es auf höhere Belastungen (z.B. durch eine zunehmende Anzahl an Benutzern) durch Hinzunahme weiterer Rechnersysteme reagieren kann.

Ausfallsicherheit: Ausfallsicherheit ist ebenfalls ein wichtiger Aspekt. Insbesondere durch die Verteilung bestimmter Services, die auf mehreren Serversystemen platziert werden, ist es möglich ein System redundant auszulegen und damit gegen Ausfälle zu schützen.

Flexibilität: Die Flexibilität ist einer der wichtigsten Vorteile verteilter Systeme. Durch eine geschickte Architektur kann man heute Systeme so gestalten, dass sie mit den Anforderungen wachsen können. Die Verteilung der verschiedenen Komponenten eines Anwendungssystems auf mehrere Rechner und bei entsprechender Architektur auf beliebige Rechnersysteme ermöglicht es, auf Änderungen zu reagieren. Verteilte Systeme können auch durch kooperierende Entwicklungsteams realisiert werden. Eine geeignete Architektur ist hier natürlich vorausgesetzt. Man kann auch fertige Bausteine hinzukaufen, die in die eigene Anwendung integriert werden. Dies setzt natürlich eine gemeinsame Ablaufumgebung voraus. Ein richtiger Markt für „Fertigbausteine“ wie bei Hardwarebausteinen ist heute noch nicht vorzufinden, ist jedoch seit langem ein Ziel. Es ist aber auch möglich, dass dieses nie oder zumindest in absehbarer Zeit nicht erreicht wird, weil Hersteller meist dagegen arbeiten.

Transparenz: Grundsätzlich wird der Begriff der Transparenz verwendet, um zu erläutern, wie die tatsächliche Verteilung der Bausteine eines verteilten Systems vor dem Anwender oder Entwickler verborgen werden kann. Es gibt verschiedene Facetten der Transparenz:

- Mit *Ortstransparenz* ist gemeint, dass man dem Anwender und wenn möglich sogar dem Entwickler verbergen möchte, auf welchen Rechnersystemen bestimmte Softwarebausteine ablaufen, wie man diese findet und wie sie dahin kommen.
- Der Begriff der *Migrationstransparenz* wird verwendet, um darzustellen, wie gut sich Softwarebausteine von einem Rechnersystem zum anderen verschieben lassen, ohne dass der Benutzer oder Entwickler es merkt.
- Von *Skalierungstransparenz* spricht man, wenn man ein System ausbauen kann, so dass es mehr leistet, ohne dass der Benutzer es merkt.
- Unter *Zugriffstransparenz* ist zu verstehen, dass man über ein System auf Ressourcen zugreifen kann (z.B. auf Objekte in einer Datenbank), ohne dass der Anwender wissen muss, wo diese liegen.
- Man spricht von *Fehlertransparenz*, wenn vor dem Anwender und/oder Programmierer gewisse Fehlersituationen des Systems verborgen werden.

Der Begriff der Transparenz wird darüber hinaus verwendet, um die Art der Datendarstellung in den verschiedenen Rechnersystemen zu verbergen (siehe weiter unten).

All diese Arten von Transparenz und auch noch weitere (*Nebenläufigkeitstransparenz*, *Persistenztransparenz*, *Replikationstransparenz*,...) können von verteilten Systemen ermöglicht werden. Einschränkend muss ergänzt werden, dass heutige Plattformen für verteilte Systeme weitestgehend für Transparenz aus Sicht des Benutzers sorgen, jedoch nur eingeschränkt aus der Sicht eines Anwendungsprogrammierers oder zumindest des Systemarchitekten, der meist genau darüber informiert sein sollte, wo die einzelnen Bausteine eines verteilten Systems platziert werden. Er trifft ja sogar die Entscheidungen darüber.

Die meisten der aufgeführten Gründe für eine Verteilung eines Systems sind sinnvoll. Es gibt aber auch gute Gründe gegen eine Verteilung, wie z.B. der erhöhte Aufwand an Administration und ggf. eine Beeinträchtigung der Leistungsfähigkeit eines Systems. Bei der Konzeption des Systems und bei der Gestaltung der Architektur sind jeweils die Vor- und Nachteile abzuwägen. Meist gibt es kein Patentrezept, wohl aber mittlerweile gute Erfahrungswerte.

Für die Unterstützung der genannten Aspekte ist es wichtig, dass das verteilte Anwendungssystem über eine geeignete Architektur verfügt, da ansonsten Skalierbarkeit, Lastverteilung usw. nicht oder nur erschwert möglich ist. Wir müssen uns also mit geeigneten Architekturen für verteilte Systeme befassen.

1.4 Das Problem der Heterogenität

Transparenz wird vor allem dadurch erschwert, dass kommunizierende Systeme heute oft nicht homogen konstruiert sind. In verteilten Systemen trifft man gewöhnlich auf verschiedene Rechnersysteme, auf denen Anwendungen ablaufen,

die möglicherweise auch noch in unterschiedlichen Programmiersprachen entwickelt wurden, aber miteinander kommunizieren.

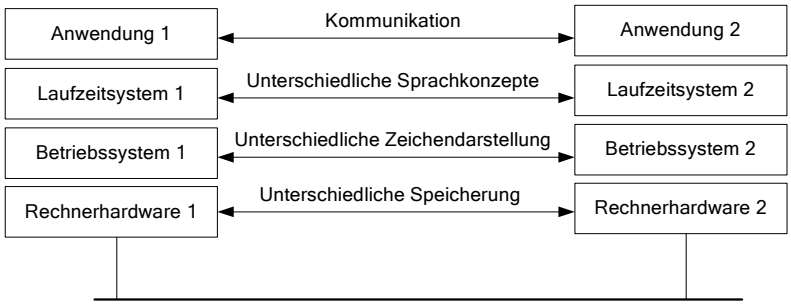

Abbildung 1-6: Heterogenität auf verschiedenen Ebenen

Die Heterogenität der beteiligten Systeme kann sich also über mehrere Ebenen erstrecken. Man muss in einem verteilten System möglicherweise verschiedene Rechnerarchitekturen, Betriebssysteme und Programmiersprachen (bzw. deren Laufzeitsysteme) so aufeinander abstimmen, dass sie auch kommunikationsfähig sind (siehe Abbildung 1-6).

Zur Verbesserung der Transparenz ist ein netzweites, oder noch besser, ein anwendungsweites Standardformat für Nachrichteninhalte anzustreben. Dies wird üblicherweise in höheren Kommunikationsschichten realisiert. Um die Problematik besser zu verstehen, sollen die verschiedenen Ebenen der Heterogenität in verteilten Systemen noch weiter erörtert werden:

Heterogenität in den Betriebssystemen: Betriebssysteme nutzen zum Teil recht unterschiedliche Zeichendarstellungen. Manche Betriebssysteme verwenden als internen Zeichensatz den EBCDIC-Code, andere den ASCII-Code und wieder andere nutzen eine Unicode-Variante.

Heterogenität in den Rechnerarchitekturen: Rechnerarchitekturen weisen Unterschiede bei der Speicherung der Daten auf. Ein klassisches Beispiel sind die Formate Big- und Little-Endian[4], die die interne Anordnung der Bytes eines Integerwerts festlegen. Beim Little-Endian-Format wird das höchstwertige Byte eines Integerwerts (short, long integer mit und ohne Vorzeichen) an der höheren Speicheradresse abgelegt. Beim Big-Endian-Format ist es gerade umgekehrt (siehe hierzu Abbildung 1-7). Das höchstwertige Byte wird beim Big-Endian-Format an der niedrigsten Speicheradresse abgelegt.

[4] Little Endian bedeutet wörtlich „Kleinender“, Big Endian bedeutet wörtlich „Großender“. Die Bezeichnung stammt von den Namen der Politiker in der Geschichte *Gulliver's Reisen*, die darüber Krieg führten, an welchem Ende ein Ei aufzumachen sei.

In Intel-Prozessoren nutzt man gewöhnlich das Little-Endian-Format. Motorola-Prozessoren vom Typ 68000, PowerPC- und SPARC-Prozessoren sowie z/Series-Prozessoren von IBM nutzen das Big-Endian-Format. Manche Prozessoren wie z.B. die Prozessoren der IA64-Architektur von Intel beherrschen beide Formate.

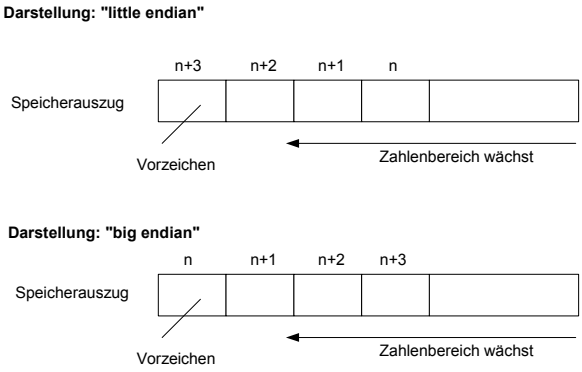

Abbildung 1-7: Little- und Big-Endian

Heterogenität in den Programmiersprachen und Laufzeitsystemen: Verschiedene Programmiersprachen haben zum Teil ganz unterschiedliche Daten- oder Objektmodelle. Wenn also z.B. eine Java-Anwendung ein Objekt vom Typ „Kunde“ an eine COBOL-Anwendung sendet, kann die COBOL-Anwendung nichts damit anfangen. Die interne Darstellung von Datentypen weicht zum Teil erheblich voneinander ab. Zeichenfolgen (Strings) können z.B. mit einer führenden Längenangabe ausgestattet sein oder nicht. Sie können mit einem speziellen Zeichen (z.B. „\0“ bei C/C++) abgeschlossen werden usw. Arrays können ebenfalls ganz unterschiedlich verwaltet werden. Schließlich können Referenzen bzw. Zeiger nicht sinnvoll an eine Partneranwendung übertragen werden, da es sich hier um Adressen im lokalen Adressraum handelt.

Es kommt sogar vor, dass Programmiersprachen auf verschiedenen Zielplattformen unterschiedlich implementiert sind und die Laufzeitsysteme damit nicht kompatibel sind. Dies war bei C-Implementierungen durchaus nichts Ungewöhnliches. Heute setzen sich für moderne Sprachen wie Java und C# zwar immer mehr sog. virtuelle Maschinen durch, die nach einer einheitlichen Spezifikation implementiert werden und unabhängig von der Plattform (Hardware+Betriebssystem) einheitliche Daten- und Objektformate unterstützen. Unterschiedliche virtuelle Maschinen können aber ebenfalls nicht ohne weiteres miteinander kommunizieren.

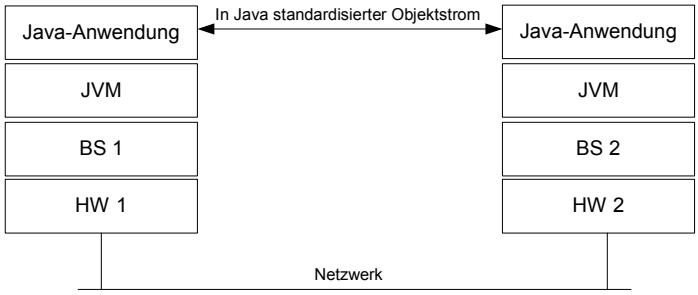

Abbildung 1-8: JVM-JVM-Kommunikation

Eine Kommunikationsverbindung könnte man auf der gemeinsamen Transportschicht TCP z.B. problemlos über eine Socket-Kommunikation herstellen, allerdings funktioniert damit die Kommunikation noch nicht. In den einzelnen Sprachen werden nämlich meist eigene Serialisierungsmechanismen, die in Java und C# als Objektströme bezeichnet werden, unterstützt. Diese sorgen für eine einheitliche Datenübertragung. Dies funktioniert innerhalb der gleichen Sprachumgebung sogar dann, wenn die Plattformen[5] unterschiedlich sind (siehe hierzu Abbildung 1-8 und Abbildung 1-9). Zwei Java-Anwendungen oder auch zwei C#-Anwendungen können also problemlos miteinander kommunizieren, wenn sie die vorhandenen Sprachmechanismen nutzen.

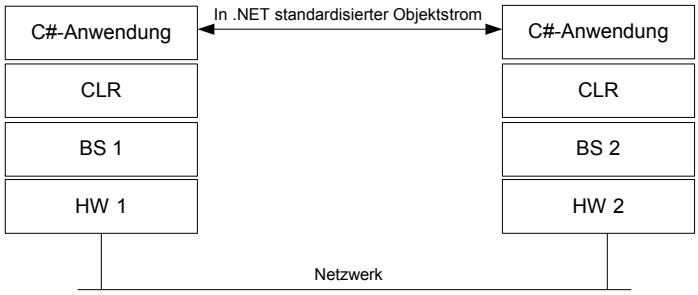

Abbildung 1-9: CLR-CLR-Kommunikation

Kommuniziert aber z.B. eine Java-Anwendung, die in einer Java Virtual Machine (JVM) abläuft mit einer C#-Anwendung, die in der .NET Common Language Runtime (CLR) abläuft, so muss man sich bewusst sein, dass diese beiden Laufzeitsysteme unterschiedliche Typsysteme aufweisen.

[5] Als Plattform bezeichnet man gewöhnlich die Hardware und das Betriebssystem.

Für alle .NET-Sprachen (C#, J#, C++,...) implementiert die CLR ein einheitliches Typsystem, so dass ein Datenaustausch sogar zwischen Anwendungen, die in unterschiedlichen .NET-Sprachen realisiert werden, möglich ist.

Die Übertragung von serialisierten Daten und Objekten sorgt aber noch nicht für allgemeine Kommunikationsfähigkeit. Beide Kommunikationspartner müssen die gleichen Serialisierungs- und Deserialisierungsregeln anwenden. Eine Java-Anwendung kann beispielsweise einen C#-Objektstrom nicht interpretieren und umgekehrt (siehe Abbildung 1-10). Wenn also eine Java-Anwendung mit einer C#-Anwendung kommunizieren muss, braucht man ein gemeinsames Verständnis der auszutauschenden Daten/Objekte. Dies könnte z.B. eine Einigung auf einen kleinsten gemeinsamen Nenner, wie z.B. dem ASCII-Zeichensatz sein.

Alternativ könnte man sich eine Zwischenschicht in Form eines Gateways vorstellen, das eine Transformation der Nachrichten vornimmt, ohne dass die Kommunikationspartner dies bemerken (Abbildung 1-11).

Die Transparenz kann durch eine Vereinheitlichung der Syntax erreicht werden, die sowohl der Sender als auch der Empfänger versteht. Der Vorgang des Umwandelns, also der Transformation in eine einheitliche Syntax nennt man *Kodierung*, *Serialisierung* oder auch *Marshalling*. Der umgekehrte Vorgang wird als *Dekodierung*, *Deserialisierung* oder *Unmarshalling* bezeichnet.

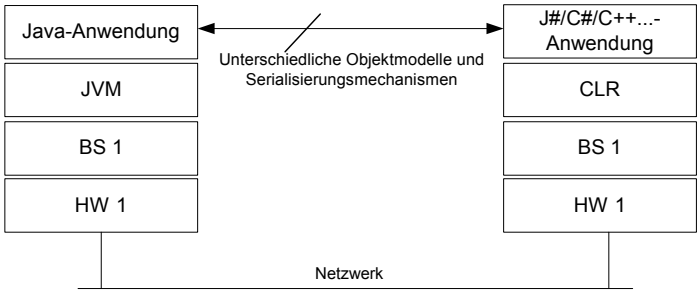

Abbildung 1-10: Problematische JVM-CLR-Kommunikation

Die ISO/OSI hat das Problem bereits in den 80er Jahren mit dem Standard für eine einheitliche Transportsyntax namens ASN.1 (Abstract Syntax Notation 1) und den dazugehörigen Kodierungsregeln BER (Basic Encoding Rules) adressiert. Dieser Standard ist funktional der Schicht 6 des ISO/OSI-Referenzmodells zugeordnet. ASN.1/BER sieht für jeden Datentypen und auch für komplexe Strukturen sog. Tags vor, die in der Nachricht jeweils vor den eigentlichen Werten übertragen werden. Aus ASN.1-Datenbeschreibungen kann man über von verschiedenen Herstellern angebotene ASN.1-Compiler Kodierungs- und Dekodierungsroutinen erzeugen, die in den Kommunikationsprogrammen eingesetzt werden, um einheitli-

che Datentypen zu übertragen. Heute nutzt z.B. SNMP (Simple Network Management Protokoll) noch ASN.1 zur Beschreibung der sog. Management Information Bases (MIBs). Dies sind die Daten, die im Rahmen des Netzwerkmanagements ausgetauscht werden.

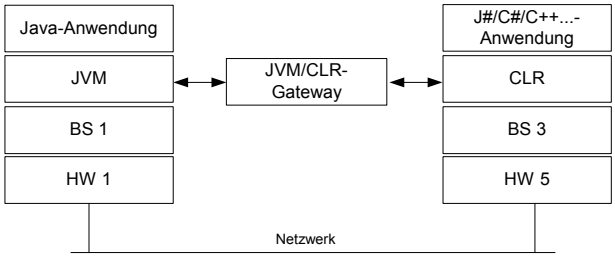

Abbildung 1-11: JVM-CLR-Kommunikation über Gateway

Wie wir in den folgenden Kapiteln noch sehen werden, bietet heute jede Middleware-Lösung ebenfalls eigene Mechanismen an. Da die Problematik recht einfach formalisierbar ist, setzt man auch das Mittel der Codegenerierung ein. In Implementierungen höherer Kommunikationsparadigmen wird die Erzeugung einer einheitlichen Transportsyntax meist über die automatisierte Erzeugung von sog. *Stubs* für die aktive Seite und *Skeletons* für die passive Seite unterstützt (Abbildung 1-12). Dies sind Programmteile, die meist automatisch aus einer Schnittstellenspezifikation ableitbar sind. Beispiele für diese Art der Codegenerierung werden noch gezeigt und sind u.a. in Sun ONC RPC, in CORBA, RMI, .NET Remoting und in den verschiedenen Webservice-Implementierungen auf recht ähnliche Weise integriert.

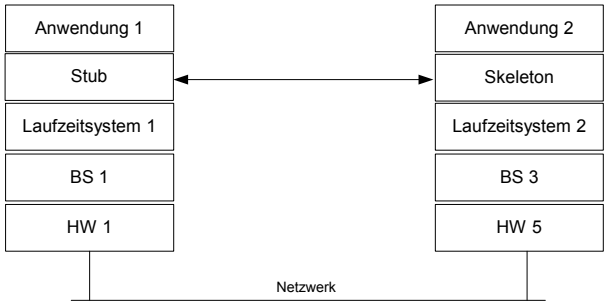

Abbildung 1-12: JVM-CLR-Kommunikation über Stub und Skeleton

Stub und Skeleton vereinheitlichen die transportierten Nachrichten und realisieren ein einheitliches Kommunikationsmodell. Die Transparenz wird dadurch unterstützt. Ob nun über diesen Mechanismus Anwendungen, die in verschiedenen

Sprachen entwickelt wurden, miteinander kommunizieren können, hängt von der Unabhängigkeit des zugrunde liegenden Konzepts ab. Sprachabhängige Mechanismen sind z.B. Java RMI, .NET Remoting, Sun ONC RPC, sprachunabhängige Mechanismen sind CORBA oder WSDL-basierte Webservices.

Für die Kommunikation ganz unterschiedlicher Anwendungen etwa über Webservices stellt sich immer mehr heraus, dass sich XML-basierte Sprachen sehr gut eignen. Wie wir noch sehen werden, gibt es für Webservices bereits einen anerkannten Standard zur Beschreibung von Kommunikationsdiensten, der mit WSDL (Web Service Description Language, siehe WWW-010) bezeichnet wird.

1.5 Spezielle Probleme verteilter Informationssysteme

In diesem Abschnitt gehen wir auf einige wichtige Problemstellungen speziell von verteilten betrieblichen Informationssystemen ein. Hierzu gehören die persistente Datenhaltung in verteilter Umgebung, die Fehleranfälligkeit, die sich aufgrund der Verteilung von Softwarekomponenten ergibt sowie die Gewährleistung der für betriebliche Systeme so wichtigen Transaktionsicherheit bei einer Verteilung der Systeme.

1.5.1 Persistente Datenhaltung in verteilter Umgebung

Unter dem Begriff „persistente Datenhaltung“ versteht man die Speicherung von Daten, die in einem Anwendungssystem über eine Sitzung hinweg benötigt werden. Persistente Daten sind also typischerweise alle Daten, die man heute in Datenbanken oder in sonstigen externen Medien ablegt. Klassisch verwendet man für größere Datenbestände relationale Datenbanken. Man kann sich heute keine größere Anwendung im betrieblichen Umfeld ohne Datenbanken vorstellen. Dies trifft auch auf verteilte Anwendungssysteme im betrieblichen Umfeld zu.

Wie wir noch sehen werden, gibt es für verteilte Systeme wie auch für lokal ablaufende Anwendungssysteme verschiedene Möglichkeiten, wie man auf persistente Daten zugreift. Größere Anwendungen kapseln üblicherweise den Zugriff auf die Datenbank(en) in einer Zugriffsschicht.

Seit es objektorientierte Sprachen gibt, besteht das Problem des *Impedance-Mismatch*. Objekte müssen auf Tabellen in relationalen Datenbanken abgebildet werden und umgekehrt. Für diese Aufgabe wurden sog. Object-Relational-Mapper (O/R-Mapper, ORM) entwickelt, die in eigenen Werkzeugen bzw. Produkten oder auch gemeinsam mit Basisprodukten für die verteilte Anwendungsentwicklung verfügbar sind. Diese O/R-Mapper ermöglichen es z.B. aus Objektklassen und deren Beziehungen automatisiert Tabellenstrukturen und Relationen zu erzeugen. Auch die Formulierung von Zugriffsoperationen wie Queries aus dem OO-Coding heraus ist möglich. Hier muss allerdings erwähnt werden, dass die Funktionalität von SQL

im Prinzip nur in eigenen Sprachen nachgebaut wird. Beispiele für derartige Ansätze sind:

- *JDO* (Java Data Objects) ist ein Ansatz aus dem Java Community Process (JSP) und gilt als ein Java-Standard für diese Aufgabe. JDO ist ein sehr breiter Ansatz, der nicht unbedingt relationale Datenbanken voraussetzt, sondern auch objektorientierte Datenbanken und XML-Speicherung unterstützt.
- Einen neueren Standardisierungsversuch der Java-Gemeinde stellt die *Java Persistence API* (*JPA*) dar, die als eigene API vorliegt und auch in neueren EJB-Versionen verwendet wird (WWW-003).
- *Hibernate* ist ebenfalls eine Lösung aus dem Java-Open-Source-Umfeld, die sich mehr und mehr zum Defakto-Standard entwickelt (Beeger 2006). Hibernate wird auch im Application-Server JBoss eingesetzt.
- *TopLin*k ist ein kommerzielles Produkt der Firma Oracle.

Die Wichtigkeit des Datenbankzugriffs wurde von den Standardisierungsgremien und Herstellern erkannt. Beispielsweise bietet auch CORBA eine Persistency-Spezifikation für diese Aufgabe. Allerdings wurde sie nie praktisch relevant.

Wir werden im Rahmen der Architekturdiskussion im Weiteren immer wieder auf den Datenbankzugriff zu sprechen kommen.

1.5.2 Fehleranfälligkeit verteilter Kommunikation

In lokalen Systemen wird ein Aufruf einer Methode oder Prozedur im lokalen Adressraum eines Prozesses durchgeführt. Sieht man von System- oder Programmabstürzen während der Ausführung ab, erfolgt die Ausführung genau einmal. Man spricht hier von einer Aufrufsemantik, die als Exactly-Once bezeichnet wird. Diese Eigenschaft möchte man auch gerne in einer verteilten Umgebung erreichen. Bei der Kommunikation in höheren Schichten gibt es aber, trotz gesicherter Transportprotokolle, viel mehr Fehlerursachen als in lokalen Umgebungen.

In manchen Situationen ist es schwierig, festzustellen, ob eine Anfrage schon ausgeführt wurde oder nicht. Dies geht weit über die Sicherstellung der Kommunikation hinaus. Es geht darum, ob aufgrund einer Nachricht (Auftrag, Request) schon eine Abarbeitungslogik durchlaufen wurde. Neben den klassischen Fehlern im Netzwerk kann z.B. ein Sender vor dem Empfang des Ergebnisses ausfallen, was zu *verwaisten Aufträgen* (sog. *Orphans*) führt. Der Empfänger kann während der Bearbeitung eines Requests ausfallen, wenn z.B. der Anwendungsprozess aufgrund eines Programmfehlers abstürzt.

Ausfälle sind zu jeder Zeit möglich und das Kommunikationssystem kann sich hier je nach Realisierung unterschiedlich verhalten. Wir gehen bei dieser Betrachtung davon aus, dass ein Dienstnehmer (oder auch Client) eine entfernte Operation aufruft, die ein Diensterbringer (auch Server genannt) bereitstellt. Je nachdem, wie viel Aufwand man in die Behandlung von Fehlern steckt, unterscheidet man bei

einem verteilt ausgeführten Request verschiedene Möglichkeiten der Fehlerbehandlung. Es gibt verschiedene Fehlerbearbeitungsmöglichkeiten (Coulouris 2002):

- Ein verteiltes System könnte z.B. überprüfen, ob beim Client eine Antwort ankommt und falls dies nicht der Fall ist, den Request erneut senden. Hier spricht man von Übertragungswiederholung. Dies ist ein bekannter Protokollmechanismus, der auch in niedrigeren Kommunikationsschichten eingesetzt wird.
- Wenn man vermeiden möchte, Requests erneut zu übertragen, kann der Server so realisiert werden, dass er Duplikate erkennt.
- Schließlich kann ein Server es ermöglichen, dass eine Antwort für einen Request gespeichert wird, damit sie noch einmal an den Client übertragen werden kann, wenn dieser denselben Request erneut sendet.
- Ein verteiltes System könnte auch Mechanismen bereitstellen, die bei Ausfall einer beliebigen Komponente zu einer beliebigen Zeit dafür sorgt, dass ein Request genau einmal ausgeführt wird und Duplikate ausgeschlossen sind. Dies ist sicherlich die komplizierteste Form der Fehlerbearbeitung.

Je nachdem, wie viel Aufwand man in die Behandlung von Fehlern steckt, spricht man bei der Fehlerbehandlung in verteilter Umgebung von den Fehlersemantiken *Maybe*, *At-Least-Once*, *At-Most-Once* und *Exactly-Once*:

Maybe ist die schwächste Form der Fehlersemantik, in der keine Fehlerbehandlung stattfindet. Im Fehlerfall kann man nicht feststellen, ob der Auftrag ausgeführt wurde oder nicht.

At-Least-Once stellt sicher, dass der Auftrag wenigstens einmal ausgeführt wird, garantiert aber nicht, dass er genau einmal abgearbeitet wird. Eine Mehrfachausführung ist möglich. Dies bedeutet aber, dass eine Anwendung, die *At-Least-Once* nutzt, es auch tolerieren muss, dass ein Request ggf. mehrmals ausgeführt wird. Die Ausführung einer erneuten Überweisungsoperation in einem Online-Banking-System ist sicherlich nicht tolerierbar.

At-Most-Once ist ähnlich wie *At-Least-Once*, filtert aber auch noch zusätzlich Duplikate aus. *At-Most-Once* wird heute von den meisten RPC-Mechanismen und deren Nachfolgern erfüllt. *At-Most-Once* hat gegenüber *At-Least-Once* Vorteile, ist aber auch aufwändiger in der Implementierung. Der Server muss eine Requestliste verwalten und bei jedem ankommenden Request prüfen, ob für ihn schon ein Eintrag in der Liste steht. Je nachdem sind dann die entsprechenden Aktionen auszuführen. Ist der Request z.B. schon in der Liste und schon ausgeführt, ist nur noch die Antwort zu übertragen. Weiterhin benötigt man eine zusätzliche ACK-Nachricht des Clients, die bestätigt, dass eine Antwort angekommen ist. Bei Systemausfällen gibt es auch bei *At-Most-Once* keine Garantie, also *At-Most-Once* kann dann ggf. nicht eingehalten werden.

Exactly-Once bedeutet, dass ein Request, komme was wolle (auch bei einem Systemausfall), genau einmal ausgeführt wird. Wenn man die Anforderung genau

betrachtet, ist sie sogar noch strenger als die Anforderung, die an lokale Aufrufe gestellt wird, da im lokalen Umfeld auch nicht ohne weiteres garantiert werden kann, dass die Ausführung einer Methode nicht durch einen Systemausfall unterbrochen werden kann. Will man *Exactly-Once* vollständig erfüllen, erfordert dies für die Behandlung von Systemausfällen ausgefeilte Recovery-Mechanismen, ein konsistentes Zurücksetzen und damit die Verwaltung von Wiederaufsetzinformationen über Logging-Mechanismen. Diese Mechanismen sind in verteilten Transaktionssystemen und Datenbankmanagementsystemen zu finden. *Exactly-Once* kann also nur durch komplexe Transaktionsmechanismen (Alles-oder-nichts-Regel) gewährleistet werden. Für die herkömmliche Kommunikation ist eine Umsetzung dieser Fehlersemantik zu aufwändig.

Verschiedene Beispielsituationen sollen die Problemstellung nochmals verdeutlichen. Es kann u.a. passieren, dass

- ein Request verloren geht
- das Ergebnis des Servers verloren geht
- der Server während der Ausführung des Requests abstürzt
- der Server für die Bearbeitung des Requests zu lange braucht.

In Abbildung 1-13 ist an dem Fehlerszenario „Ergebnis des Servers geht verloren“ dargestellt, was diese Fehlersituation nach sich zieht. Der Server hat die Auftragsbearbeitung schon ausgeführt und muss sich das merken, bei *Exactly-Once* sogar über einen Systemausfall hinweg. Der Client muss mit einer erneuten Anfrage reagieren, der Server darf aber die Anfrage bei *At-Most-Once* und *Exactly-Once* nicht noch einmal bearbeiten, sondern muss das temporär gespeicherte Ergebnis an den Client senden.

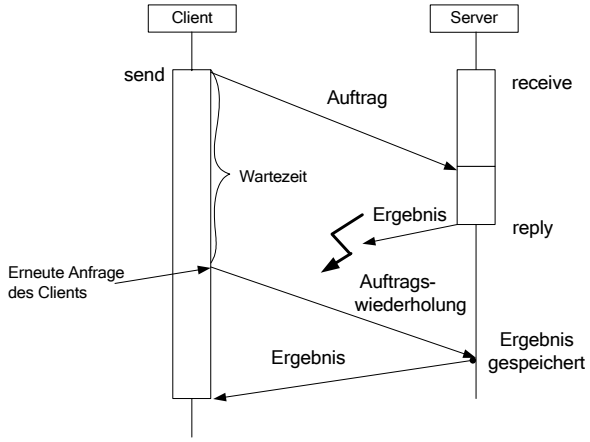

Abbildung 1-13: Beispielsituation: Ergebnis geht verloren nach Weber (1998)

Wie wir in den folgenden Kapiteln noch sehen werden, ist in Technologien wie CORBA und Java RMI maximal die Fehlersemantik *At-Most-Once* realisiert. Exact-

ly-Once erfordert spezielle Mechanismen, die einen Ausfall überleben. Nutzbar sind hier z.B. Transaktionsmechanismen für verteilte Systeme. Transaktionsprotokolle, die *Exactly-Once* garantieren sollen, verwenden für die Ergebniskoordination sog. Commit- oder Koordinationsprotokolle (1-Phase-Commit, 2-Phase-Commit, 3-Phase-Commit). Für einen Entwickler wird die Implementierung der Fehlerbehandlung in höheren Kommunikationsmechanismen wie RPC, Java RMI, CORBA usw. aber verborgen. Er merkt z.B. nichts, wenn eine Antwort nicht rechtzeitig eintrifft. Das System sorgt z.B. für eine erneute Anfrage bei einem Timerablauf. Hier geht es, wie oben bereits beschrieben, um Transparenz. Ein Request soll wie der Aufruf einer lokalen Prozedur behandelt werden.

1.5.3 Transaktionssicherheit in verteilten Informationssystemen

Aus der lokalen Verarbeitung in Großrechnern entwickelte sich das Konzept der Transaktionen als Fehlertoleranzkonzept zur Sicherstellung bzw. Verbesserung der semantischen Integrität der Daten eines Anwendungssystems. Die Grundüberlegung ist, dass es möglich sein muss, mehrere Operationen auf Datenbestände ganz oder gar nicht auszuführen. Man spricht hier von Atomarität. Eine Unterbrechung einer Operationsfolge, die zusammengehört, könnte zu einer Inkonsistenz in den Daten führen. Beispielhaft werden hier häufig eine Soll- und eine Habenbuchung in einem Buchhaltungssystem aufgeführt. Beide Ümsätze müssen entweder ganz oder gar nicht ausgeführt werden, sonst stimmen die Buchhaltungsdaten nicht mehr. In lokalen Umgebungen ist dieses Problem auch als Synchronisationsproblem bekannt und wir nutzen zur Lösung Semaphore, Monitore, Sperren und dergleichen.

Will man die Alles-oder-Nichts-Semantik vollständig implementieren, benötigt man ein System, das den Status der Bearbeitung auch im Fehlerfall rekonstruieren kann, um ein Wiederaufsetzen bei einem Neustart zu ermöglichen. Dieses System bezeichnet man auch als Transaktionssystem. Es muss Mechanismen bereitstellen, die es ermöglichen, jede Zustandsänderung auf einem ausfallsicheren Speicher zu protokollieren. Der Entwickler einer Transaktionsanwendung muss Möglichkeiten haben, in seinem Programm die Grenzen von Transaktionen anzugeben (Begin-Transaction, Commit) oder ggf. auch selbst ein Zurücksetzen einer Transaktion (Rollback) zu initiieren. Dies alles selbst zu programmieren wäre zu aufwändig. Transaktionssysteme sorgen für die notwendigen Basismechanismen.

Das Transaktionskonzept stammt aus der Welt der Datenbanken. Datenbanksysteme unterstützen dieses Konzept schon lange, dort ist es heute eine Selbstverständlichkeit. Keine etwas komplexere Anwendung, die eine Datenbank benutzt, kommt ohne Transaktionen aus.

Seit langem versucht man auch, das Transaktionskonzept in die Welt der verteilten Systeme zu transferieren. Wie bereits diskutiert, kann man z.B. für die Implementierung einer *Exactly-Once*-Fehlersemantik für einen Remote-Procedure-Call das Transaktionskonzept ebenfalls sehr gut nutzen, obwohl es hier nur um die Aus-

führung einer Einzeloperation geht. Wenn man aber ausfallsicher sein möchte, benötigt man Wiederanlaufmechanismen und diese sind in Transaktionssystemen implementiert.

Es stellt sich die Frage, ob das Transaktionskonzept für die Kommunikation in verteilten Anwendungen genau so sinnvoll ist wie in lokaler Umgebung und welche zusätzlichen Mechanismen man benötigt, um eine Transaktion über Rechnergrenzen hinweg zu realisieren. Eine andere Frage beschäftigt sich grundsätzlich mit dem Sinn von Transaktionen in verteilten Umgebungen und versucht, seine Grenzen für praktisch nutzbare Anwendungen zu ermitteln. Wir werden in diesem einführenden Kapitel nicht weiter auf diese Fragen eingehen, sondern sie in den späteren Kapiteln immer wieder aufgreifen.

1.6 Übungsaufgaben

1. Versuchen Sie eine Definition für ein verteiltes Informationssystem mit eigenen Worten!
2. Nennen Sie drei Gründe für die Verteilung eines betrieblichen Informationssystems und erläutern Sie diese!
3. Welche Arten der Heterogentät wurden identifiziert?
4. Welche Fehlersituationen muss eine At-Most-Once-Implementierung für einen entfernten Aufruf zusätzlich zu At-Least-Once implementieren und wie kann die Implementierung erfolgen?
5. Wird in lokalen Methodenaufrufen innerhalb eines Prozesses eine *Exactly-Once*-Fehlersemantik garantiert?
6. Ist es sinnvoll, dass eine Java-Anwendung und eine C#-Anwendung ohne weitere Maßnahmen über eine Kommunikationsschnittstelle Objekte austauschen? Begründen Sie Ihre Entscheidung!
7. Wie löst Java das Problem der Heterogenität in der reinen Java-Welt?
8. Was ist ein Big-Endian- im Unterschied zu einem Little-Endian-Format?
9. Wozu braucht man in verteilten Systemen eine symbolische Adressierung der verteilten Bausteine und welche Lösung gibt es hierfür?
10. Wozu braucht man Object-Relational-Mapper?
11. Erläutern Sie anhand eines Beispiels, wie eine Verteilung von Anwendungsbausteinen dazu beitragen kann, dass ein Anwendungssystem ausfallsicherer wird!