9 Verteilte Systeme

M. Mühlhäuser

9.1 Grundbegriffe und Besonderheiten

Definition. Ein verteiltes (Datenverarbeitungs-)System besteht aus mehreren autonomen Prozessor-Speicher-Systemen, die mittels Nachrichtenaustausch kooperieren.

Häufig sind die Prozessor-Speicher-Systeme in verteilten Systemen vollständige Rechner. Mehrprozessorsysteme mit gemeinsamem Speicher sind für sich allein *keine* verteilten Systeme, ebensowenig wie eng gekoppelte Mehrrechnersysteme, die durch zentrale oder verteilte Koordination die Autonomie der Teilsysteme einschränken. Häufig wird in verteilten Systemen unterschieden zwischen den beteiligten *Rechnernetzknoten* (Prozessor-Speicher-Systemen) und dem sie verbindenden *Kommunikationssubsystem* – oft und im

folgenden auch kurz *Knoten* und *Netz* genannt. Rechner, die als Zwischenstationen in einem nicht voll vermaschten Netz Nachrichten weiterleiten, sind dabei gleichzeitig Knoten *und* Teil des Netzes.

Eine erweiterte Definition wird sinnvoll in der anbrechenden sog. Post-PC-Ära des allgegenwärtigen Rechnens (*ubiquitous computing*): mobile Endgeräte, sogar beliebige Alltagsgegenstände und Geräte (*internet appliances*) werden Elemente verteilter Systeme. Knoten müssen hier zumindest *Identität* und gegebenenfalls (passive) *Kommunikationsfähigkeit* aufweisen, Prozessor und Speicher werden optional. Beispielsweise kann eine Jacke durch Anbringen eines passiven, drahtlos auslesbaren digitalen Etiketts (*smart label*) in Form eines *radio frequency identifiers* (*RfID*) zum Bestandteil des Internets werden. Der vom digitalen Etikett gelieferten eindeutigen Bezeichnung des Gegenstandes kann eine Internet-Adresse als Verweis auf Details über den Alltagsgegenstand zugeordnet werden (der sog. *Datenschatten*), bei der Jacke z. B. der Besitzer oder die Waschanleitung zur Nutzung durch ans Internet angeschlossene Waschmaschinen. Diese erweiterte Definition gewinnt für die Softwaretechnik an Bedeutung, nicht aber für das Wesen verteilter Berechnungen.

Charakteristika. Zentrales Charakteristikum und Problem verteilter Systeme ist das *Fehlen einer aktuellen konsistenten Sicht.* Autonomie, Parallelität und nicht vernachlässigbare räumliche Entfernung der Knoten führen dazu, daß von vornherein weder ein Knoten die volle Kontrolle über Zustand und Verlauf eines kooperativen Ablaufes hat, noch sich vollständige Zustandsinformation verzögerungsfrei in einem Knoten sammeln läßt. Als Folge hiervon ergeben sich weitere Probleme wie beispielsweise *Nichtdeterminismus* (infolge variierender Übertragungs- und Bearbeitungszeiten) und *Inkonsistenzen* (durch Ausfälle von Knoten oder Teilnetzen sowie Netzpartitionierung). Mangels exakter gemeinsamer Zeit lassen sich auch *Kausalitäten* nicht immer nachvollziehen. Viele praktische Probleme rühren zudem von der *Heterogenität* von Rechnerarchitekturen, Leistungskenngrößen, Kommunikationsprotokollen, Betriebssystemen und Datendarstellungen her.

Kennzeichnend gegenüber Parallelrechnern sind die höhere Bedeutung der *Sicherheit* (Vertraulichkeit, Authentizität, Integrität und Verfügbarkeit) sowie *dynamischer Softwarekonfigurationen* (Anzahl und Topologie kooperierender Prozesse variieren zur Laufzeit).

Probleme der Synchronisation, des Programmierens und Testens verteilter Programme führen mit den vorgenannten Eigenschaften häufig zu einer deutlich *höheren Komplexität* als bei sequentiellen oder nebenläufigen nicht-verteilten Programmen.

Klassifikationen. Die verschiedenen existierenden Klassifikationen erlauben kaum die ausschließliche Zuordnung eines verteilten Systems zu einer bestimmten Klasse, sondern sind eher Verständnishilfen. Hier sollen nur zwei Klassifikationen erwähnt werden:

- Nach *Betriebszielen* werden vier Klassen unterschieden: *Funktionsverbunde* zur netzweiten Nutzung spezifischer Fähigkeiten von Knoten; *Datenverbunde* für die lokale Speicherung, aber netzweite Verfügbarkeit von Daten; *Lastverbunde* zur Lastverteilung; *Verfügbarkeitsverbunde* mit redundanten Komponenten oder Daten.
- Nach der *räumlichen Ausdehnung* werden folgende Netzarten unterschieden: *lokale Netze* (*local area network, LAN*) unter der Betriebshoheit einer juristischen Person, ursprünglich bis rund 5 km; *Weitverkehrsnetze* (*wide area network, WAN*) unter der Betriebshoheit eines sog. Netzbetreibers, meist einer Telekommunikationsgesellschaft; bisweilen sog. *Metropolitan Area Networks* (*MAN*) als Zwischenstufe sowie weltumspannende Netze (unter dem in sich widersprüchlichen Begriff *Global Area Network, GAN*). Die räumliche Ausdehnung hat ihre Bedeutung in dieser Einteilung

zugunsten der Betriebshoheit und verwendeten Kommunikationsprotokolle weitgehend verloren. Eine wahre Begriffsinflation bescherte der Literatur beispielsweise die neuen Netzarten *Body Area Network* und *Desktop Area Network* sowie die anwendungsspezifischen Klassen *Car Area Network* und *Home Area Network*.

Prinzipien. Für die Anwendung und Entwicklung verteilter Systeme haben sich zwei Grundprinzipien herauskristallisiert:

- *Transparenz*: Dem Benutzer und Software-Entwickler soll sich das verteilte System als eine einzige abstrakte (nebenläufige) Maschine präsentieren. Dieses Ziel der Verteilungstransparenz wird üblicherweise nur in Teilbereichen erreicht, für die in der Literatur viele, teils uneinheitliche Begriffe eingeführt wurden. Beispielsweise versucht man, den Ort eines Datums oder einer Funktionsausführung zu verbergen (*Orts- oder Lokationstransparenz*), verteilungstypische Fehler durch Systemunterstützung zu maskieren (*Fehlertransparenz*) oder den Unterschied zwischen lokalen und entfernten Ressourcen oder Funktionen zu verdecken (*Zugriffstransparenz*) und sogar die Unterschiede zwischen lokalen und entfernten Ausführungs- oder Zugriffszeiten vernachlässigbar zu machen (*Leistungstransparenz*). Dabei ist es wichtig, die mit der Abstraktion verbundenen Mechanismen (wie die Plazierung von Daten und Funktionen) beeinflussen und dabei die Transparenz durchbrechen zu können.
- *Schichtung*: Wie bei sequentiellen Rechnern und Programmen werden häufig verschiedene Schichten abstrakter Maschinen aufeinandergebaut, wobei die Maschinen einer Schicht die der jeweils darunterliegenden Schicht verbergen. Das Grundverständnis über verwendete Systemmodelle und Mechanismen unterscheidet sich von dem für sequentielle Systeme, denn die abstrakten Maschinen sind verteilt und arbeiten nebenläufig. Zusammengefaßte Leistungen einer Schicht werden als „Dienst“ bezeichnet, sog. „Dienstprimitive“ dienen der Kommunikation zwischen Schichten. Kommunikationsprotokolle bestimmen die Kooperation zwischen den nebenläufigen Instanzen, insbesondere den Austausch sog. Protokolldateneinheiten und die Interpretation darin enthaltener sog. Protokollkontrollinformation (nach DIN, korrekter wäre -steuerungsinformation, oft auch als Header und Trailer von Dateneinheit oder Paketkopf genannt). Der Begriff *Dateneinheit* wird anstelle von *Nachrichten* verwendet im Zusammenhang mit Verarbeitungseinheiten, welche nur diesen Paketkopf interpretieren. Näheres zu den Schichten wird in Kapitel C6 erläutert. Auf die dort aufgeführten Protokollmechanismen wird nachfolgend Bezug genommen.

9.2 Verteilte Algorithmen

Eindeutige Berechnungsvorschriften, welche auf die Ausführung in verteilten Systemen zugeschnitten sind, werden *verteilte Algorithmen* genannt. Entsprechend der Definition verteilter Systeme in Abschnitt 9.1 sind sie gekennzeichnet durch die Beteiligung mehrerer Prozessor-Speicher-Paare sowie durch Kommunikation mittels Nachrichtenaustausch. An die Stelle von Prozessor-Speicher-Paaren treten im verteilten Algorithmus Prozesse mit lokalem Speicher. Bei der Realisierung verteilter Programme werden im allgemeinen Betriebssystem-Prozesse verwendet, wobei es (bis auf Effizienz) unschädlich ist, wenn sich statt der 1:1-Abbildung von Prozessen auf Prozessoren mehrere Betriebssystem-Prozesse je einen Prozessor teilen.

Zu den herkömmlichen Zielen des Algorithmenentwurfs kommen hoher Parallelitätsgrad, Fehlertoleranz und mit steigender Anzahl von Prozessen und Kommunikationsverbindungen möglichst geringe Zunahme von Anzahl und Länge der Nachrichten – hierzu existie-

ren verschiedene Erweiterungen des Komplexitätsbegriffes aus der theoretischen Informatik. Eindeutigkeit der Berechnungsvorschrift kann auf das Verhalten der beteiligten Komponenten beschränkt sein; deterministisches Verhalten in dem Sinn, daß aus den selben (zulässigen) Eingabedaten immer die selben Ausgabedaten verteilt berechnet werden, ist in manchen Zusammenhängen unmöglich oder unerwünscht.

Abschnitt 9.2 ist nur im Licht des erwünschten hohen Parallelitätsgrades und der Fehlertoleranz verständlich. Sonst könnte man beispielsweise eine (einzige) Berechtigungsmarke einführen und nur demjenigen Prozeß, der im Besitz der Marke ist, Berechnungen gestatten. Die Marke könnte reihum von Prozeß zu Prozeß weitergereicht werden, und die erlaubte Besitzdauer könnte beschränkt werden. Das in der Einleitung genannte zentrale Charakteristikum *Fehlen einer aktuellen konsistenten Sicht* könnte dann leicht umgangen werden und verteilte Algorithmen würden sich nicht wesentlich von herkömmlichen unterscheiden. Offensichtlich ist ein solches Vorgehen aber im allgemeinen unerwünscht. Hoher Parallelitätsgrad bedeutet sogar, daß die Ausführung mancher verteilter Algorithmen parallel zum „normalen“ Ablauf stattfinden soll, den man dann nicht unterbrechen möchte. Beispielsweise möchte man häufig prüfen, ob ein verteiltes Programm terminiert ist in dem Sinne, daß alle Prozesse auf ankommende Nachrichten warten, aber keine Nachrichten mehr unterwegs sind; die zyklische verteilte Prüfung dieser Bedingung soll nicht erfordern, daß das eigentliche verteilte Programm jedes Mal angehalten werden muß – es ließe sich ja erst nach der Prüfung sagen, ob eine laufende oder eine ohnehin abgeschlossene Berechnung angehalten wurde. Ein anderes Beispiel: wollte man beim elektronischen bargeldlosen Zahlungsverkehr – verteilt über alle Banken – ermitteln, welche Geldmenge insgesamt im Umlauf ist (auf Konten gelagert oder in Überweisung begriffen), so würde man dafür nicht den gesamten Zahlungsverkehr anhalten wollen. Zur Unterscheidung werden nachfolgend die Begriffe *systemorientiert* (im Beispiel: Geldmengen-Ermittlung, Prüfung auf Terminierung, also die hier interessierenden Probleme) und *anwendungsorientiert* (im Beispiel: Zahlungsverkehr) für verteilte Algorithmen und Berechnungen verwendet.

Auch streng nach dem Master-Slave-Prinzip arbeitende Algorithmen ließen sich vergleichsweise leicht entwerfen, sind aber im allgemeinen unerwünscht: dezentral arbeitende Algorithmen sind fehlertoleranter, bei zentralistischen Varianten ist Wiederaufsetzen nach Ausfall des koordinierenden Prozesses oft problematisch; zudem sind ein Master und seine direkten Kommunikationsverbindungen potentielle Leistungsengpässe, man spricht daher von schlechter *Skalierbarkeit*.

9.2.1 Problematik

Das *Fehlen einer aktuellen konsistenten Sicht* bedeutet für verteilte Algorithmen, daß ein Prozeß im verteilten System zu einem gegebenen Zeitpunkt den *gegenwärtigen* globalen Systemzustand nicht kennen kann. Daher möchte man wenigstens feststellen, welche globalen Zustände ein verteiltes Programm in der Vergangenheit durchlaufen hat. Leider ist auch dieses Problem in gängigen Rechnernetzen nicht vollständig lösbar. Die Taktzeit moderner Prozessoren liegt unter 1 ns. Bei der Datenübertragung dauert die Signalausbreitung rund 100 ns pro 20 m Entfernung (in Kupfer bei 2/3 der Lichtgeschwindigkeit). Hinzu kommt die Zeit, die der Länge der Nachricht entspricht, beispielsweise 1 ms für 100 Bit über FastEthernet. Verzögerungen in Warteschlangen, durch Interrupts, Scheduling usw. führen zu erheblichen Schwankungen, so daß Uhrensynchronisation durch Austausch von Nachrichten (sog. *interne Synchronisation*) trotz Teilkompensation der Verzögerungen bei gängigen lokalen Netzen nur mit einer Genauigkeit im Millisekunden-Bereich möglich ist. Häufige Synchronisation reduziert die für normale Kommunikation

verfügbare Übertragungsrate, weshalb man nicht sehr viel häufiger als einmal pro Sekunde synchronisiert. Die quarzgesteuerten rechnerinternen Uhren können aber innerhalb einer Sekunde um bis zu 1 µs vor- oder nachgehen; dieser vermeintlich kleine Wert entspricht bei modernen Prozessoren und Netzen Tausenden Prozessorzyklen und mehreren Datenübertragungen. Beispielsweise können Synchronisationsfehler und Gangungenauigkeiten dazu führen, daß der Empfangszeitstempel einer Nachricht im Empfängerrechner kleiner sein kann als der mitgeschickte Sendezeitstempel; das müßte so interpretiert werden, daß die Nachricht empfangen wurde, bevor sie überhaupt gesendet wurde. Zusammenfassend können reale Uhren weder hinreichend genau noch hinreichend oft synchronisiert werden, als daß durch mitprotokollierte Aufzeichnungen und Zeitstempel nachträglich für zwei verteilt aufgetretene Ereignisse in jedem Fall bestimmt werden könnte, in welcher Reihenfolge sie auftraten.

Das Problem läßt sich auch nicht vollständig beseitigen, wenn man die sog. externe Synchronisation mittels in alle Rechner eingebauter Funkuhren wählt. Das DCF77-Signal, das in Deutschland zur Steuerung von Funkuhren verwendet wird, weist zwar nur Gangungenauigkeiten von 10^{-13} s pro Sekunde auf, doch bleiben die Gangunterschiede der Empfängeruhren wie oben beschrieben erhalten. Auch das DCF77-Signal erlaubt nur einmal pro Sekunde den Uhrenabgleich, so daß die für interne Synchronisation beschriebenen Ungenauigkeiten und Folgen für verteilte Berechnungen auch hier auftreten. Nur die Synchronisation selbst ist wesentlich exakter; insbesondere wirkt sich der räumliche Abstand nur auf Laufzeitunterschiede des Signals aus.

Von den bekannten Algorithmen und Verfahren zur internen und externen Uhrensynchronisation ist insbesondere das im Internet verbreitete NTP (network time protocol) zu nennen. NTP unterstützt Hierarchien von „Time-Servern“, wodurch z. B. in Organisationen ein Server für die (ungenaue) Synchronisation im Weitverkehrsnetz bereitgestellt werden kann und pro lokalem Netz einer für vergleichsweise genaue Synchronisation direkt benachbarter Rechner. Zwei Server synchronisieren sich, indem einer eine Anfrage-Nachricht schickt, die der andere umgehend beantwortet. Sendezeit der Anfrage und Empfangszeit der Antwort sind lokal bekannt, Empfangszeit der Anfrage und Sendezeit der Antwort werden mit der Antwort geschickt. Aus diesen vier Werten läßt sich die Summe der Übertragungsdauern beider Nachrichten d (delay) bestimmen – und zwar exakt, weil die Uhrenunterschiede durch die Summenbildung herausfallen – sowie der Versatz o (offset) mit einer Genauigkeit von $\pm d/2$. Da im Internet Übertragungsverzögerungen stark schwanken können, wird das Verfahren im allgemeinen mehrfach direkt hintereinander angewendet, die Messung mit dem kleinsten Fehler wird dann verwendet. Die exakte Bestimmung des maximalen Meßfehlers gilt allgemein als großer Vorteil von NTP, neben der besonders Internet-tauglichen hierarchischen Server-Struktur. Auch externe Synchronisation über Funkuhren wird unterstützt.

Synchronisationsverfahren wie NTP berücksichtigen auch, daß Uhren nicht zurückgestellt sondern höchstens angehalten werden sollten (um lokal zu garantieren, daß Ereignisse mit aufsteigenden Zeitstempeln tatsächlich nacheinander auftraten). Zusammenfassend läßt sich festhalten, daß im verteilten System aufsteigende Zeitstempel nicht zeitliches Nacheinander garantieren. Versteht man den globalen Zustand eines verteilten Programmes als Vektor (geordnete Liste) der Zustände seiner Komponenten, dann läßt sich nach den bisherigen Überlegungen nicht immer feststellen, welche von zwei oder mehr nebenläufigen Komponenten zuerst einen lokalen Zustandswechsel durchgeführt hat (mangels Aussagekraft der Zeitstempel), ob also beispielsweise bei drei Komponenten der Übergang vom globalen Zustand (1,2,2) nach (2,2,3) über den Zustand (1,2,3) oder über (2,2,2) erfolgte.

9.2.2 Logische Uhren als Ersatz für synchrone Uhren

Bei genauerer Analyse des Synchronisationsproblems fragt sich, ob und wann die Ereignis- und Zustandsreihenfolge in nebenläufigen Komponenten überhaupt wichtig ist. Bei der Ausführung verteilter Berechnungen und bei der Fehlersuche interessieren vor allem kausale Zusammenhänge (kann Ereignis e_1 Ereignis e_2 hervorgerufen haben?). Dazu hat Lamport die „happened-before"-Relation „$\rightarrow$" eingeführt. Er unterscheidet drei Klassen von Ereignissen: lokale, Sende- und Empfangsereignisse. Lokale Ereignisse e_1 und e_2 eines Prozesses p_i stehen in „happened-before"-Relation $e_1 \rightarrow e_2$, wenn e_2 nach e_1 auftrat (sonst stehen sie in umgekehrter Relation, wobei davon ausgegangen wird, daß lokale Ereignisse leicht total zu ordnen sind). Ein Empfangsereignis steht in „happened-before"-Relation zu einem Sendeereignis, wenn beide dieselbe Nachrichtenübertragung betreffen (der Empfang einer Nachricht erfolgt zwingend nach ihrem Senden). Für sonstige Ereignisse e_1 und e_2 gilt $e_1 \rightarrow e_2$ genau dann, wenn es eine Kausalkette mit Ereignissen e_{i1}, e_{i2}, … e_{in} gibt, so daß gilt: $e_1 \rightarrow e_{i1} \wedge e_{i1} \rightarrow e_{i2} \wedge \ldots \wedge e_{in} \rightarrow e_2$.

Lamport entwarf auch einen einfachen Algorithmus zum Umgang mit dieser Relation. Er führte dazu den Begriff der *logischen Uhren* ein: Jeder Prozeß eines verteilten Programms besitzt eine logische Uhr in Form einer ganzzahligen Variablen. Diese wird vor jedem lokalen Ereignis um eins erhöht, das Ereignis wird mit dem Wert der logischen Uhr als Zeitstempel versehen (das erste Ereignis erhält den Zeitstempel 1, das zweite 2 usw.). Auch vor einem Sendeereignis wird entsprechend verfahren. Die gesendete Nachricht erhält dann den Zeitstempel des Sendeereignisses. Die einzige Besonderheit des Algorithmus besteht darin, beim Empfang einer Nachricht die logische Uhr des Empfängers mit dem Maximum aus deren aktuellem Wert und dem Zeitstempel der Nachricht zu aktualisieren, diesen Wert unmittelbar um eins zu inkrementieren und das Empfangsereignis mit dieser neuen logischen Zeit zu stempeln.

Es läßt sich leicht zeigen, daß der Algorithmus eine partielle Ordnung auf den Zeitstempeln $t(e)$ von Ereignissen herbeiführt, so daß Ereignisse, die in „happened-before"-Beziehung zueinander stehen, hinsichtlich ihrer Zeitstempel geordnet sind. Es gilt also:

$$e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$$

Allerdings gilt nicht die Umkehrung: Hat ein Ereignis e_1 einen größeren Zeitstempel als ein Ereignis e_2, so muß e_2 nicht in „happened-before"-Beziehung zu e_1 stehen.

Dieses Manko läßt sich durch die Einführung von *Vektoruhren* beseitigen. Dazu hält jeder Prozeß einen sog. Vektor (im Sinne einer Liste von logischen Uhren-Werten), dessen Länge der Prozeßanzahl im verteilten Programm entspricht. Die i. Stelle des Vektors T_j von Prozeß p_j repräsentiert die Kenntnis, welche p_j über die logische Uhr von p_i hat. Solange nur lokale Ereignisse auftreten, zählt p_j diese in $T_j[j]$ wie oben beschrieben. Jedem Ereignis und jeder Nachricht wird nicht nur die lokale Uhr, sondern der ganze Vektor als Zeitstempel aufgeprägt. Bei Empfang einer Nachricht maximiert ein Empfänger p_j jedes Element seines Vektors mit dem entsprechenden aus der Nachricht, erhöht dann $T_j[j]$ um eins und markiert das Empfangsereignis mit dem so erhaltenen Vektor. Der Vergleichsoperator „<" wird nun so definiert, daß für zwei Vektoren die Ungleichung $T_i < T_j$ genau dann gilt, wenn alle Komponenten von T_i kleiner oder gleich der entsprechenden Komponente von T_j sind und außerdem mindestens eine Komponente von T_i echt kleiner ist als die entsprechende Komponente von T_j . Es gilt also:

$$T_i < T_j \Leftrightarrow (T_i[k] \leq T_j[k]\ \forall\ k \wedge \exists\ l\colon T_i[l] < T_j[l])$$

Der Vorteil des Vektoruhren-Algorithmus besteht nun darin, daß nur noch Ereignisse, welche in „happened-before"-Relation stehen, auch geordnete Zeitstempel besitzen:

$$e_1 \rightarrow e_2 \Leftrightarrow T_i(e_1) < T_j(e_2)$$

wobei $T_n(e_m)$ den Zeitstempel eines Ereignisses e_m kennzeichnet und *i* bzw. *j* die Prozesse seien, an denen e_1 bzw. e_2 auftraten.

Bild 1 stellt den Ablauf dreier mit Vektoruhren vermessener Prozesse von links nach rechts dar. Beispielsweise besitzen das erste Ereignis in p_1 und das zweite in p_3 die ungeordneten Zeitstempel (1,0,0) und (0,0,2). Ihre zeitliche Reihenfolge ist also nicht bestimmbar. Dagegen trägt das Sendeereignis der Nachricht m_1 mit (0,1,0) einen nach Definition kleineren Zeitstempel als das Empfangsereignis von m_2, (3,1,3). Diese stehen also in „happened-before“-Relation.

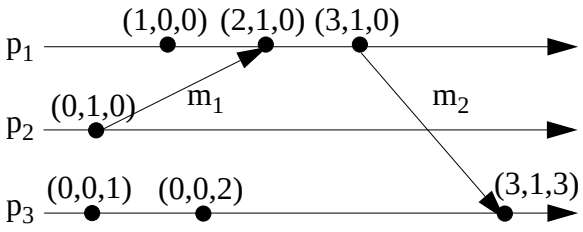

Bild 1 Beispiel zu Vektoruhren

Offensichtlich steigt der Aufwand pro Knoten für lokale Speicherung von Ereignishistorie und aktueller Vektorzeit und für den Uhrenabgleich bei Empfang sowie die Länge des Zeitstempels pro Nachricht linear mit der Zahl der an einer Berechnung beteiligten Knoten. Dies ist im Einzelfall gegen den geschilderten Nachteil der einfachen logischen Uhren abzuwägen. Weitergehende Literatur zeigt, wie man die Vorteile der Vektoruhren beibehalten und die Nachrichtenlänge (bezüglich Zeitstempel) konstant halten kann; dabei kommen allerdings die lokalen Datenstrukturen (sog. „Matrix-Uhren“) in die Nähe quadratischen Wachstums.

9.2.3 Schnitte als Ersatz für globale Zustände

Am Schluß von Abschnitt 9.2.1 wurde darauf hingewiesen, daß bei verteilten Berechnungen normalerweise nicht alle durchlaufenen Zwischenzustände eindeutig bestimmt werden können. Zwei Prozesse mit je zwei Zuständen 0 und 1 können beispielsweise von (0,0) nach (1,1) entweder über (0,1) oder über (1,0) gekommen sein, je nachdem, welcher Prozeß zuerst den Zustandswechsel durchführte, was auf Grund der diskutierten Uhrenproblematik bisweilen nicht entschieden werden kann.

Es ist immer möglich, eine Abbildungsfunktion von bei einem Prozeß auftretenden Ereignissen auf die Zustände dieses Prozesses zu definieren. Deshalb sei hier vereinfachend angenommen, daß jede logische Uhr den Zustand des zugehörigen Prozesses widerspiegelt. Dann kann der Zustand einer verteilten Berechnung als Vektor der logischen Uhren der beteiligten Prozesse definiert werden. Die in Abschnitt 9.2.2 eingeführten Vektoruhren stellen aber nur die im allgemeinen veraltete lokale Sicht der Prozesse auf den Gesamtzustand dar (die Uhren werden bei Nachrichtenempfang häufig in mehreren Komponenten „vorgestellt“, der Fortschritt kann in verschiedenen Prozessen abschnittsweise stark unterschiedlich verlaufen).

Auf Grund dieser Erkenntnisse ersetzt man den Begriff des globalen Zustandes durch den des Schnittes (*cut*) und sucht nach verteilten Algorithmen zur Ermittlung von Schnitten. Wie verteilte Zustände und Vektoruhren stellen Schnitte geordnete Listen (sog. Vektoren)

dar und werden häufig als Tupel von Zählern dargestellt. Im Unterschied zum Zustandsbegriff muß die im Verlauf einer (anwendungsorientierten) verteilten Berechnung ermittelte Folge von Schnitten nicht tatsächlich durchlaufen worden sein sondern nur potenziell. Das bedeutet für eine terminierte Berechnung: gegeben ein globaler Startzustand (als Vektor der wohldefinierten Startzustände aller beteiligten Prozesse) und ein Endzustand (ebenfalls eindeutig zu ermitteln bei terminierten verteilten Berechnungen). Dann ist unter Berücksichtigung der aufgezeichneten Ereignisse und deren „happened-before"-Beziehungen eine aufgezeichnete Schnittfolge *einer der möglichen* Wege vom Start- zum Endzustand, nicht notwendig der tatsächlich durchlaufene Weg. Eine vollständige Schnittfolge wird als *Spur* (*trace*) bezeichnet. Bei einer Spur unterscheiden sich zwei aufeinanderfolgende Schnitte nur in einer Komponente und dort nur um eins. Die tatsächlich durchlaufenen Zustände können sich von der ermittelten Spur nur im Ausmaß einer begrenzten Umsortierung unterscheiden, welche keine „happened-before"-Beziehungen verletzt.

Konkrete Algorithmen zur Schnittermittlung sollten nicht auf dem Vektoruhren-Algorithmus aufbauen, da dieser zuviel Speicher und zu lange Nachrichten benötigt und die anwendungsorientierten Nachrichten (um umfangreiche Zeitstempel) erweitert. Eine erste Überlegung könnte dahin gehen, alle Prozesse reihum nach ihrem aktuellen Zustand zu fragen und die dabei aufgezeichneten Zustände zu einem Schnitt zusammenzufassen. Das Ergebnis ist aber möglicherweise kein Schnitt nach obiger Definition, sondern ein sog. *inkonsistenter Schnitt*, der der happened-before-Regel widerspricht; z.B. kann er einen Prozeßzustand nach Erhalt einer Nachricht *N* enthalten, während er den Sender von *N* in einem Prozeßzustand vor Versenden von *N* enthält. Schnitte nach Definition werden zur Unterscheidung *konsistent* genannt. Die Ermittlung konsistenter Schnitte muß offensichtlich Nachrichten berücksichtigen, die zwischen Beginn und Ende der Schnittermittlung gesendet und/oder empfangen werden. Dies kann auf verschiedene Weise geschehen.

Ein verbreiteter Algorithmus stammt von Chandy und Lamport. Er frägt nicht nur alle Prozesse nach deren Zustand (auch nicht reihum, wie noch erläutert wird), sondern berücksichtigt Nachrichten wie folgt. Dabei sei angenommen, daß Prozeß p_i seinen Zustand zum Zeitpunkt $t(p_i)$ aufzeichnet und p_j zum Zeitpunkt $t(p_j) > t(p_i)$. Dann gilt für alle zwischen $t(p_i)$ und $t(p_j)$ von p_i an p_j gesendeten Nachrichten: wurden sie auch vor $t(p_j)$ von p_j empfangen, dann sind sie im dort aufgezeichneten Zustand berücksichtigt; wurden sie zum Zeitpunkt $t(p_j)$ noch nicht empfangen, dann werden sie (von p_j) als „zum Zeitpunkt des Schnittes unterwegs" aufgezeichnet. Alle zwischen $t(p_i)$ und $t(p_j)$ von p_j an p_i gesendeten Nachrichten werden als „zum Zeitpunkt des Schnittes unterwegs" aufgezeichnet. Der Chandy-Lamport-Algorithmus beruht auf sog. Markern. Dabei wird von einer beliebigen aber festen Zahl von unidirektionalen, reihenfolge-erhaltenden Kommunikationsverbindungen (sog. Kanälen) zwischen den Prozessen ausgegangen. Ein Prozeß kann eine Schnittermittlung anstoßen, indem er einen Marker an sich selbst sendet (für den nachfolgenden Pseudocode kann hierfür ein eingehender Kanal Nr. 0 reserviert werden). Für jeden Prozeß beginnt die Schnittermittlung mit dem Empfang eines Markers über einen Kanal (oder von sich selbst) und endet, nachdem auf jedem eingehenden Kanal ein Marker empfangen wurde. Auf den Empfang des ersten Markers wird reagiert, indem auf jedem ausgehenden Kanal ein Marker gesendet wird. Das Netz wird „geflutet": über jeden Kanal geht (genau) ein Marker, dieser spült gewissermaßen die normalen Nachrichten aus dem Kanal (sie werden empfangen, bevor der Marker empfangen wird).

Nachfolgend wird der Algorithmus in Pseudocode beschrieben, und zwar anhand einer Prozedur, die mit dem Empfang von Markern verknüpft sei.

```
MarkerEmpfang(↓kanal-ein):
   param  kanal-ein: integer;-                -- Nr. des Kanals des Marker-Empfangs

begin
   if not schnittErmittlungLaeuft then        -- dies ist der erste Marker, den wir sehen
      schnittErmittlungLaeuft := true;
      for kanalnr := 1 to maxKanalAus do      -- flute alle Kanäle mit Markern
         MarkerSenden (kanalnr)
      end,
      for kanalnr := 1 to maxKanalEin do      -- ab jetzt empfangene Nachrichten speichern
         if not (kanalnr = kanal-ein)
            then StarteAufzeichnung (kanalnr)
            else
               StarteAufzeichnung (kanalnr)  -- für Marker-Empfangskanal: leere
               StoppeAufzeichnung (kanalnr)  -- Nachrichtenmenge speichern
         end
      end
   else                                       -- Berechnung läuft, ein Kanal ist „fertig“
      StoppeAufzeichnung (kanal-ein);
      if alleKanäleGestoppt then              -- Schnitterkennung für diesen Prozeß fertig
         schnittErmittlungLaeuft := false;
   end
   return
end MarkerEmpfang
```

9.3 Entwurfskriterien und weitere Algorithmen

Erwünschte Eigenschaften. Am Chandy-Lamport-Algorithmus läßt sich eine Reihe allgemeiner Aspekte des Entwurfs und der Bewertung verteilter Algorithmen diskutieren. Beispielsweise wird deutlich, daß einige interessierende Eigenschaften (möglichst formal, anhand eines Kalküls) nachgewiesen werden sollten wie z. B. die Terminierung – die hier leicht nachzuprüfen ist, da alle ausgehenden Kanäle mit je einem Marker beschickt werden und jeder Prozeß nach Empfang je eines Markers auf jedem eingehenden Kanal die Berechnung beendet. Schwieriger ist die Frage, ob der Algorithmus in jedem Fall tut, was er soll (konsistente Schnitte berechnen). Außerdem wäre noch zu klären, wie die aufgezeichneten Informationen zu Schnitten zusammengefaßt werden und wer diese Zusammenfassung wann vornimmt; hierzu muß je nach Anforderungen der bisher vorgestellte Algorithmus unterschiedlich erweitert werden. Häufig interessiert die Frage, ob ein Algorithmus toleriert, von mehreren Prozessen verschränkt angestoßen zu werden. Im vorliegenden Fall hieße das, daß ein Prozeß eine Schnittermittlung startet, während in anderen Teilen des Prozeßnetzes schon Marker unterwegs sind; man kann zeigen, daß dies bei dem besprochenen Detaillierungsgrad unschädlich ist, bei der detaillierten Programmierung sowie Sammlung und Auswertung der Schnitte aber berücksichtigt werden muß. Die allgemein erwünschten Eigenschaften verteilter Algorithmen werden ähnlich kategorisiert wie bei herkömmlichen Algorithmen (Sicherheit, Lebendigkeit, Fairneß). Auf zusätzliche Effizienzkriterien (Nachrichtenkomplexität) wurde bereits hingewiesen.

Topologien. Der Chandy-Lamport-Algorithmus benutzt offensichtlich die (im allgemeinen teilvermaschte) Topologie der Kanäle des Verbindungsnetzes, welches der eigentlichen Anwendung zu Grunde liegt. Häufig werden die Prozesse aber auch in einem Ring angeordnet, der mit der anwendungsorientierten Verbindungstopologie nichts zu tun hat. Vorteil ist das nur lineare Wachstum der Anzahl von Nachrichten mit der Anzahl von Knoten, Nachteile sind die möglicherweise lange (weil sequentielle) Bearbeitungszeit und die Schwierigkeit, alle anwendungsorientierten Nachrichten zu berücksichtigen. Auch span-

nende Bäume werden häufig verwendet, weil die Nachrichtenanzahl nur logarithmisch wächst und das anwendungsorientierte Verbindungsnetz mitverwendet werden kann; manche Algorithmen bauen den spannenden Baum während der verteilten Berechnung auf und „vergessen“ ihn nach jedem Durchlauf.

Weitere wichtige Algorithmenklassen. Die meisten Algorithmen für konsistente Schnitte können so modifiziert werden, daß Information über beliebige Charakteristika einer verteilten Berechnung gewonnen werden kann. So interessiert bei verteilten Simulationen das Minimum aller (pro Prozeß vorhandenen) „lokalen Simulationszeiten“ und aller Simulationszeiten, die in unterwegs befindlichen anwendungsorientierten Nachrichten festgehalten werden; in anderen Fällen interessiert die Anzahl noch nicht quittierter Nachrichten usw. Solche Berechnungen werden mit sog. *Schnappschuß-Algorithmen* durchgeführt. Auf Grund der für Schnitte besprochenen Beschränkungen können die interessierenden Werte im allgemeinen nicht exakt berechnet, sondern nur konservativ abgeschätzt werden, was für viele Zwecke ausreicht. Auch die Frage der Terminierung im eingangs erwähnten Sinn kann als Schnappschuß interpretiert werden.

Auch in dezentral arbeitenden verteilten Algorithmen wird häufig ein temporärer Koordinator gesucht oder ein „Sieger“ bei konkurrierenden Anforderungen. Derlei Probleme werden mit Wahl- oder *Voting-Algorithmen* gelöst. Typische Beispiele sind gegenseitiger Ausschluß für den Zugriff auf kritische Abschnitte (mutex) und Überwachungsfunktionen in lokalen Netzen mit beliebig an- und abschaltbaren Teilnehmern.

Mit der steigenden Bedeutung von Groupware (Software für Gruppenarbeit, z.B. zum verteilten „Malen“ auf einer Zeichenfläche) und von Verteil-Medien (wie „streaming video“) werden *Multicast-Algorithmen* immer wichtiger. Ziel von Multicast ist es, Daten an eine Gruppe von Empfängern zu leiten und dabei über jede zum Erreichen aller Empfänger erforderliche Teilstrecke nur einmal (korrekt) zu senden; Daten werden also erst unterwegs nach Bedarf dupliziert anstatt vom Sender weg gleich n mal gesendet zu werden. Die am intensivsten erforschte Frage bei Multicast-Algorithmen ist die, wie allen nicht zwischenzeitlich ausfallenden Empfängern trotz Übertragungsfehlern und Ausfällen zuverlässig alle Daten reihenfolgetreu ausgeliefert werden können. Die von unidirektionaler Übertragung bekannten Quittungen sollen dabei vermieden werden, weil der Vorteil, beim Sender ein statt n Datenpakete abzuschicken, nicht durch die Notwendigkeit eingeschränkt werden soll, n Quittungen zu empfangen. SRM (*scalable reliable multicast*) ist dabei die Bezeichnung für eine Reihe von Algorithmen für Anwendungen, bei denen sich alle Gruppenmitglieder Multicasts schicken (Gruppenvideokonferenz, Groupware). Jeder Multicast enthält Aufzeichnungen über die vom Sender empfangenen jüngsten Multicasts der verschiedenen Sender. Durch Mithören dieser Aufzeichnungen erfahren Gruppenmitglieder von anderen „in der Nachbarschaft“, wenn diese Multicasts erhalten haben, welche bei ihnen nicht ankamen; sie können dann diese „Nachbarn“ oder den Sender bitten, eine Kopie des verlorengegangenen Datums nachträglich zu senden.

Häufig verwendete Prinzipien. Das bei Chandy-Lamport verwendete *Fluten* ist ein wichtiges Entwurfsprinzip für verteilte Algorithmen. Von *Berechnung in Wellen* spricht man dagegen, wenn innerhalb einer verteilten Berechnung mehrfach alle Knoten „besucht“ werden. Will man beispielsweise in Schnappschuß-Algorithmen nicht fluten, sondern mit spannenden Bäumen arbeiten, kann man offensichtlich nicht alle unterwegs befindlichen anwendungsorientierten Nachrichten „aus dem Netz spülen“. Man kann in diesem Fall in der ersten Welle in jedem Prozeß die verteilte Berechnung anstoßen und gleichzeitig in den Nachrichten dieser ersten Welle die Anzahl der bis dahin von allen Prozessen versendeten anwendungsorientierten Nachrichten summieren. Angestoßene Prozesse markieren ab dem Anstoß ihre anwendungsorientierten Nachrichten, so daß der Empfängerprozeß

feststellen kann, ob eine Nachricht vor oder nach dem Anstoßen des Senders abgeschickt wurde. Nun kann man so lange weitere Wellen ausführen, welche die Zahl der empfangenen und vor Anstoßen der Sender versandten Nachrichten summieren, bis diese Summe gleich der genannten Summe aus der ersten Welle ist. So ist sichergestellt, daß alle beim Anstoßen noch unterwegs befindlichen Nachrichten ankamen und vom Empfänger in die verteilte Berechnung einbezogen wurden.

Als drittes sei noch das Prinzip des *Färbens* erläutert. Dazu sei zunächst das erwähnte Schnappschuß-Beispiel verteilter Simulation aufgegriffen, bei dem das Minimum der Simulationszeiten aller Prozesse und der Simulationszeiten aller unterwegs befindlicher Nachrichten interessiert. Dieses Minimum steigt stetig, weil die Simulationszeit voranschreitet, außer wenn sie durch eintreffende Nachrichten zurückgestellt wird, und weil verarbeitete Nachrichten gelöscht werden; darum wird die verteilte Minimum-Berechnung laufend wiederholt. Flutet man die Kanäle mit Markern ähnlich wie bei Chandy-Lamport, dann kann jeder Prozeß mit der ersten Marker-Nachricht seine lokale Simulationszeit in einen Zwischenspeicher kopieren. Bis zum Eintreffen der Marker auf allen anderen Kanälen minimiert ein angestoßener Prozeß die Simulationszeit-Einträge eintreffender Nachrichten mit dem Wert im Zwischenspeicher. Am Ende des Flutens ist sichergestellt, daß alle zu Beginn der Berechnung unterwegs befindlichen Nachrichten beim Empfänger berücksichtigt wurden; dann kann in einem weiteren Durchgang (z.B. auf einem Ring) eine Nachricht von Prozeß zu Prozeß weitergereicht werden mit einem Wert „Gesamtminimum“, der von jedem Prozeß mit dem Wert im Zwischenspeicher minimiert wird. Das so entstehende Gesamtminimum stellt wie bei Schnappschüssen üblich eine konservative Abschätzung dar – wie besprochen ist dies der „Preis“ für die Ermittlung ohne Anhalten der anwendungsorientierten Berechnung. Um den Algorithmus zu optimieren, kann man den zweiten Durchlauf mit der Flut zusammenlegen, die den nächsten Durchlauf der Minimumberechnung startet. Damit werden Nachrichten gespart. Um die verschränkten Berechnungen unterscheiden zu können, kann man ein Prinzip einsetzen, das „Färbung“ (von Prozessen) genannt wird. Im Beispiel könnte jede geradzahlige Flut mit „roten Markern“ durchgeführt werden, die empfangende Prozesse „rot“ färben und zur laufenden Berechnung des lokalen Minimums den „roten Zwischenspeicher“ verwenden. Jede ungeradzahlige Flut könnte „dasselbe in Grün“ durchführen, gleichzeitig aber das „rote Gesamtminimum“ ermitteln (während bei jeder roten Flut das grüne Gesamtminimum ermittelt würde).

Verteilte Algorithmen werden ausführlicher in Kapitel A7 behandelt. Als weiterführende Literatur seien insbesondere [Tel 00] und das Buch von Colouris et al. empfohlen.

9.4 Anwendungsorientierte Protokolle

Die transportorientierten Kommunikationsprotokolle gemäß Kapitel C6 ermöglichen fehlerfreien Nachrichtenaustausch unmittelbar zwischen Anwendungsprozessen über vermittelnde Rechner hinweg, ohne Kenntnis der Netztopologie und unter Beachtung von Optimierungszielen wie Vermeidung von Stau im Netz oder beim Empfänger. Weitergehender Komfort für Entwickler verteilter Anwendungen ist wünschenswert. Als Architekturansätze hierfür sind zu unterscheiden: (1) schichtenstrukturierte Kommunikationsdienste, siehe Abschnitt 9.4.1, (2) Anwendungsdienste, siehe Abschnitt 9.4.2 und (3) Plattformen, siehe Abschnitt 9.6. Ansatz (1) wurde im ISO-OSI favorisiert; wie in Kapitel C6.2.4 dargestellt, hat sich der ISO-OSI-Ansatz nicht durchgesetzt, er wird aber wegen seiner vorbildlichen Konzepte und wegen verbreiteter Teilstandards (insbesondere ASN.1) behandelt.

9.4.1 Kommunikationssteuerung und Datendarstellung

Unter *Kommunikationssteuerung* versteht man die Strukturierung einer Kommunikationsbeziehung, typischerweise aufbauend auf einer – oder mehreren aufeinander folgenden – virtuellen Transportschicht-Verbindung. Gängige Begriffe sind *Aktivität* für eine Menge zusammengehöriger Datenaustauschoperationen und *Dialogeinheit* für semantisch gekoppelte Teilabschnitte, z.B. von der Hotelsuche bis zur Zimmerbuchung innerhalb einer Reiseplanung.

Die ISO standardisierte einen umfangreichen Kommunikationssteuerungsdienst und sah eine separate Schicht (Schicht 5) dafür vor. Spezielle Dienstfunktionen zur Signalisierung von Aktivitäten und Dialogeinheiten an Kommunikationspartner wurden vorgesehen, andere dienten dazu, die Erstellung von Sicherungspunkten anzufordern und Rücksetzoperationen bei Partnern auszulösen als Grundlage für Konsistenzerhaltung. Die Mechanismen selbst (sichern, rücksetzen usw.) waren nicht Gegenstand des Standards. Weiter wurde der Austausch von Berechtigungen und die Steuerung der Übertragungsrichtung bei Richtungsverkehr unterstützt. Die entsprechenden Konzepte dienen häufig als Vorbild für maßgeschneiderte Anwendungen, der eigentliche OSI-Standard wird aber nur noch wenig verwendet, z.B. in kaum bedeutenden Telematik-Diensten wie Teletext.

Der Abgleich der *Datendarstellungen* in heterogenen Systemen erfordert zweierlei:

- Die lokalen Darstellungen der Daten müssen aufeinander abgebildet werden. Um eine quadratisch wachsende Anzahl von Konvertierungsvorschriften zu vermeiden, überträgt man in einem genormten Format, der sog. *Transfersyntax*.
- Die Kommunikationspartner müssen sich über den Typ ihrer Anwendungsdaten verständigen können, möglichst auch für zusammengesetzte Strukturen wie Mengen, Vektoren usw. Dazu werden Typdefinitionen ausgetauscht, sog. *abstrakte Syntaxen.*

Die ISO hat sowohl einen Darstellungsdienst (also Schicht-6-Dienst) genormt als auch die Transfersyntax BER sowie die Notation ASN.1 zur Spezifikation abstrakter Syntaxen.

ASN.1. Die *Abstract Syntax Notation No.1* teilt Typen in vier Gruppen ein: vordefinierte Typen (*universal*); Typen, die im Kontext eines Anwendungsbereiches eingeführt werden (*application*); anwenderspezifische (*private*) Typen und Typen, die nur im lokalen Kontext einer Definition gültig sind (*context-specific*). Die Schlüsselwörter *context-specific* und *universal* werden in Spezifikationen weggelassen. Pro Gruppe werden Typen mit eindeutigen Nummern versehen, sog. Tags. Der Programmierer muß alle Tags außer den vordefinierten spezifizieren. Basis- und zusammengesetzte Typen werden unterschieden. Da auch ungeordnete Mengen unterstützt werden, dienen die Tags auch zur eindeutigen Identifikation der Elemente beim Empfänger. Beispielsweise kann in der ASN.1-Spezifikation

```
Buch    ::=  [PRIVATE 0]   SET {
             Titel ::=  [0]   IMPLICIT  IA5STRING
             Autor::=   [1]   IMPLICIT  IA5STRING}
```

der Empfänger einer entsprechenden Nachricht am Tag *private-0* den Typ *Buch* erkennen und durch *context-specific-0* bzw. *context-specific-1* die Elemente trotz unbekannter Reihenfolge zuordnen. Da *Set* und *IA5String* als vordefinierte Typen auch die Tags *universal-17* und *universal-22* besitzen, würden im Beispiel für Menge und Elemente je zwei Tags vergeben und übertragen. Als Abhilfe verhindert das Kennwort *implicit* in einer Spezifikation, daß der Universal-Typ in Transfersyntax codiert wird.

BER. Die sog. *basic encoding rules* sind auf ASN.1 abgestimmt. BER codiert Typ- und Tag-Information meist in einem Byte: zwei Bits kennzeichnen die Gruppe (z.B. 00 für *universal*), ein Bit unterscheidet zusammengesetzte (c = complex) und elementare (p = primitive) Typen, fünf Bits nehmen die Tag-Nummer auf (11111 kennzeichnet Überlauf

ins Folge-Byte). Im nächsten Byte folgt die Längenangabe, dann der Inhalt, der entweder direkt die codierten Daten enthält oder geschachtelt die nächste Typinformation. Im obigen Beispiel wäre im ersten Byte *private-c-0* codiert; im zweiten die Anzahl von Bytes für alles, was noch folgt; dann *universal-c-22* (*set*); dann wieder die Restlänge, dann *context-specific-p-0*, dann (infolge *implicit* unmittelbar) die Länge der ersten Zeichenkette, dann die Zeichenkette selbst, gefolgt von Längenangabe und zweiter Zeichenkette.

OSI-Kritik, Internet- und Web-Ansatz. Schichten in offenen verteilten Systemen realisieren verteilte virtuelle Maschinen; sie stellen erheblichen Mehraufwand dar und sind nur zu rechtfertigen, wenn fast alle Anwendungen von der virtuellen Maschine profitieren können, erhebliche Funktionalität erbracht wird und die Softwareentwicklung oberhalb der virtuellen Maschine (durch Abstraktion von Details) deutlich einfacher wird. Kommunikationssteuerung wird jedoch von den wenigsten Anwendungsentwicklern geschätzt, Datendarstellung ist häufig erforderliche Funktion, erhöht aber nicht allgemein das Abstraktionsniveau der Anwendungsentwicklung. Deshalb war es aus heutiger Sicht falsch, in OSI die Schichten 5 und 6 einzuführen.

Im Internet wird Kommunikationssteuerung nicht nennenswert unterstützt. Für die Datendarstellung ist mit *XDR* (*external data representation*) ein neutrales Datenformat verbreitet ohne den Ballast einer separaten Schicht; es wird häufig beim Prozedur-Fernaufruf verwendet (siehe Abschnitt 9.5.3). Anstelle von ASN.1 dient eine Schnittstellendefinitionssprache der Spezifikation abstrakter Syntaxen. Anstelle selbstbeschreibender Datenstrukturen mit Tags genügt der Prozedurname, um empfängerseitig die Daten in lokale Darstellung zurückzuwandeln; die Nachrichten werden kompakter, der vorübersetzte Transformationscode effizienter.

XML setzt sich als Web-Standard für viele Zwecke durch (siehe Kapitel E3.5.3), auch als Alternative zu ASN.1. Als Schnittstellendefinitionssprache wird der auf XML basierende Standard *SOAP* (*simple object access protocol*) propagiert, XML löst damit auch XDR ab. XML folgt der Internet-Philosophie menschen-lesbarerer Formate und ist daher nicht sehr kompakt, binäre XML-Repräsentationen vergleichbar BER werden sich aber kaum durchsetzen auf Grund ständig steigender Internet-Übertragungsgeschwindigkeiten.

Zum Austausch abstrakter Beschreibungen von kommerziellen Dokument-Typen wie Rechnungen, Überweisungen usw. wurde der EDIFACT-Standard eingeführt; er enthält keine konkrete Transfersyntax. Auch für EDIFACT sind XML-basierte Alternativen wie XML/EDI auf dem Vormarsch.

9.4.2 Anwendungsprotokolle

Anwendungsprotokolle werden in OSI treffender als (*Anwendungs-*)*Dienstelemente* (*application service elements*) bezeichnet. Sie stellen verteilt erbrachte funktionelle Einheiten dar, die vom Anwendungsprogrammierer nach Bedarf kombiniert werden können. Mit dem Internet selbst hat auch dessen unsaubere Begrifflichkeit den OSI-Ansatz weitgehend überlebt, der Begriff „Protokoll“ steht oft fälschlich für „Dienst“ oder „Dienstelement“.

Etliche konkrete OSI-Standards setzten sich vor allem wegen übertriebener Komplexität und Funktionsvielfalt nicht durch. So realisieren OSI-Dienstelemente oft verteilte virtuelle Geräte (virtuelle Dateispeicher, Mailsysteme, Endgeräte und andere). Dazu werden komplexe Datenstrukturen auf den beteiligten Knoten konsistent gehalten. Anwendungsprozesse werden kompliziert durch umfangreiche Normen zur Benutzung der Dienstelemente und zum Funktionsabgleich zwischen virtuellem und realem Gerät. Will man einfach benutzbare Funktionen den Anwendungen oder interaktiven Benutzern zur Verfü-

gung stellen, dann müssen zusätzlich Benutzeragenten (user agents) als eigenständige Prozesse oder Bibliotheken spezifiziert werden.

Im Internet dagegen sind virtuelle Geräte nur ansatzweise modelliert. Pro Anwendungsklasse existiert oft nur ein Dienstelement (als Anwendungsprotokoll bezeichnet). Auf Grund weit geringerer Gesamtkomplexität ist dieses vergleichsweise leicht in Anwendungen zu realisieren, üblicherweise in einem Client- und einem Server-Prozeß, die direkt auf dem Internet-Transportdienst TCP aufsetzen. Diese Prozesse können direkt interaktiv benutzt werden, meist wird auch eine Programmierschnittstelle angeboten. Internet-Anwendungsprotokolle sind komplexer als ein einzelnes OSI-Dienstelement, aber wesentlich kompakter und meist effizienter als bei OSI das Zusammenspiel von Benutzeragenten, mehreren Dienstelementen sowie den Schichten 5 und 6. Andererseits realisieren Internet-Anwendungen oft weniger Funktionen und lassen viele Details offen.

Vier grundlegende Anwendungsdienstelemente von OSI seien noch erwähnt, um das Konzept zu veranschaulichen. Das *association control service element ACSE* realisierte ein schichteninternes Äquivalent virtueller Verbindungen, das *reliable transfer service element RTSE* bot sichere Übertragung auf unsicheren Transportdiensten, das *remote operations service element ROSE* ermöglichte Prozedurfernaufruf und das *commitment concurrency and recovery service element CCR* realisierte das Zweiphasen-Commit-Protokoll als Grundlage für Datenbank- und Transaktions-Anwendungen. Nachfolgend werden die wichtigsten Anwendungsprotokoll-Klassen behandelt. Der OSI-Ansatz wird jeweils kurz als Vorbild vorgestellt und mit verbreiteten Internet-Lösungen verglichen.

Virtuelle Endgeräte. Das Anwendungsprotokoll *Network Virtual Terminal NVT* des Internets wird in der verteilten Anwendung *Telnet* realisiert. NVT regelt die zeilenweise Übertragung von 7-Bit-ASCII-Zeichen zwischen Benutzer und Anwendung sowie wenige spezielle Steuersignale wie *EC* (*erase character*) zum Löschen des vorangegangenen Zeichens. Telnet implementiert das eigentliche Kommunikationsprotokoll; Steuerfunktionen werden mit dem reservierten Zeichen *IAC* (*interpret as command*) in die Zeichenübertragung eingebettet. Da die Norm nur ein Minimum vorschreibt, müssen bei der Wahl von Funktionen immer beide Seiten zustimmen. Je nach initiierender Seite erfolgt die Funktions-Anforderung mit dem *IAC will* oder *do*, die Annahme/Ablehnung mit *do/dont* oder *will/wont*.

OSI spezifiziert gemäß dem oben beschriebenen Konzept in seinem Dienstelement *Virtual Terminal (VT)* ein virtuelles Endgerät. Es wird auf den beiden beteiligten Seiten durch je eine Datenstruktur modelliert, die konsistent gehalten wurden: eine dort, wo sich Bildschirm und Benutzer physisch befinden, die andere auf dem entfernten Knoten, an dem der Benutzer arbeiten möchte.

ITU-Standards für Videokonferenzen unterstützten ebenfalls virtuelle Endgeräte (T.120, siehe Kapitel E3), aber auch die Verteilung der Bildschirm-Ausgabe (z.B. eines Textverarbeitungsprogramms) auf mehrere Teilnehmer-Rechner sowie die entfernte Eingabe (meist via Berechtigungsmarke je einem Teilnehmer im Wechsel zugeteilt).

Neben diesen Standards ist das sogenannte *Teleporting* zu erwähnen. Darunter versteht man die Möglichkeit, eine laufende Bildschirm-Sitzung am lokalen Rechner unterbrechen und an einem anderen Rechner fortsetzen zu können (z.B. unerledigte Arbeit von zu Hause aus). Dazu wird die gesamte Bildschirmausgabe lokal auf einen „virtuellen“ Bildschirm umgelenkt, hinter dem sich ein Programm zum effizienten Verpacken und Verschicken dieser Bildschirmausgabe verbirgt; auf dem entfernten Rechner wird der ausgepackte Bildschirminhalt in einem Fenster dargestellt, so daß er konkurrierend zu den dort lokalen Ausgaben betrachtet und interaktiv benutzt werden kann. In Betriebssystemen wie

Windows wird diese Funktionalität mitgeliefert, Programme wie *VNC* (*virtual network computing*) ermöglichen Teleporting auch zwischen verschiedenen Betriebssystemen.

Einen ganz anderen Ansatz repräsentiert *Xwindow*, das ein Kommunikationsprotokoll zwischen Anwendung und Interaktionskomponente (Display-Server) definiert.

Dateitransfer. Hier lohnt erneut ein Vergleich des gescheiterten OSI-Ansatzes mit dem älteren, aber überlebenden Internet-Ansatz. In OSI wurde mit *File Transfer, Access, and Management (FTAM)* ein kompliziertes Anwendungsdienstelement spezifiziert, das Dateispeicher-Verwaltungsoperationen (Änderungen von Attributen), Dateitransfer (Austausch und Löschen ganzer Dateien) sowie selektiven Dateizugriff erlaubte. Gemäß dem eingangs von Abschnitt 9.4.2 beschriebenen Prinzip verteilter virtueller Geräte modellierte FTAM einen virtuellen Dateispeicher, der zwischen entfernt zugreifendem Prozeß (Client) und physisch zugreifendem Prozeß (Server) konsistent gehalten wird. Die Funktionsvielfalt führt erneut zu hoher Komplexität, viele der eigentlichen Funktionen (lokaler Dateizugriff, Ein-/Ausgabe, …) müssen aber immer noch im Rahmen der Anwendungsentwicklung programmiert werden.

Im Gegensatz dazu umfaßt Internet-FTP das Anwendungsprotokoll und die Standardanwendung. Weder virtuelle Dateien noch Abbildungen zwischen Dateitypen werden unterstützt.

FTP kennt wenige Dateitypen (Text, Bitstrom, Bitgruppen), Authentifizierung, einfache (z.B. Auskunfts-)Operationen auf dem Dateisystem, Dateiübertragung in beide Richtungen (put, get), Wiederaufsetzpunkte bei Verbindungsabbruch und Kompression.

Mitteilungstransfer (E-Mail). Für den Austausch elektronischer Mitteilungen hat OSI die X.400-Norm der ITU als *MOTIS* (*message oriented text interchange system*) übernommen; MOTIS wird von vielen Telekommunikationsorganisationen eingesetzt. Benutzeragenten-Prozesse bedienen sich dabei eines Netzes von Vermittlungsprozessen für Mitteilungen, den Mitteilungsagenten (*message transfer agent*, *MTA*, im allgemeinen einer pro Knoten des Systems). Falls Benutzeragenten auf nicht ständig am Netz angeschlossenen Systemen ablaufen, können sie von einem Mitteilungsspeicher (*message store*) vertreten werden. Weitere Komponenten leisten den Übergang zu anderen Mitteilungssystemen und zur Briefpost.

Im Rahmen dieser Architektur werden nicht weniger als fünf spezielle Dienstelemente spezifiziert. Mit Hilfe von *Message Administration* kann sich ein Benutzeragent oder Mitteilungsspeicher bei einem Anbieter (Mitteilungsspeicher oder Mitteilungsagent) authentifizieren und registrieren. Über *Message Delivery* werden Mitteilungen vom Ziel-Mitteilungsagenten abgeliefert sowie Auslieferung und Auslieferungsbericht gesteuert. Mit *Message Retrieval* kann der Benutzeragent beim Mitteilungsspeicher Mitteilungen auflisten, abholen, löschen; in Gegenrichtung kann der Eingang einer neuen Nachricht signalisiert werden. *Message Submission* dient der Ablieferung von Mitteilungen beim Quell-Mitteilungsagenten zwecks Weiterleitung. Zur Kommunikation zwischen Mitteilungsagenten wird *Message Transfer* eingesetzt.

Im weitverbreiteten Internet-*SMTP* (*simple mail transfer protocol* [Postel 82]) werden die Mitteilungsagenten nach Sender und Empfänger unterschieden. Sie werden mit den lokalen Mailsystemen in je einem Anwendungsprozeß verquickt, z.B. *sendmail* bei UNIX. Empfängerseitig werden Mitteilungen einfach nacheinander in eine benutzerspezifische Datei geschrieben. Benutzeragenten und Mitteilungsspeicher sind nicht genormt, mit Ausnahme des Protokolles zum Aufgeben von Mitteilungen und der Regeln zur Behandlung der Mitteilungsdatei. Empfängerprozesse können Mitteilungen auch weitervermitteln. Fortgeschrittene Funktionen von MOTIS wie Sende-/Ablieferungsberichte und ver-

zögerte Auslieferung finden sich in SMTP kaum. *MIME* (*multipurpose internet mail extensions* [Borenstein 92]) ermöglicht nationale Zeichensätze und Multimedia. Neben einfachen ASCII-Texten sind Unterstrukturen und Formatinformation erlaubt. Medien wie Audio und Video wurden als sog. *content-type* definiert. Im Rahmen des *content-transfer-encoding* können MIME-Nachrichten über herkömmliche Zwischensysteme „getunnelt“ und Binärdaten direkt übermittelt werden.

Namensdienste. Logische Namen sind frei wählbare Bezeichner, die vom System auf physische Adressen abgebildet werden. Sie können in Anwendungen Platzhalter sein für Objekte, die erst zur Laufzeit konkretisiert werden oder deren Adresse infolge Migration wechselt. Zum Beispiel bildet bei E-Mail-Adressen wie

„klaus_meier@mathe.uni-bonn.de“

der hintere Teil einen hierarchischen Namen, dessen Gliederung (Land = de, Organisation = uni-bonn, Abteilung = mathe) mit den Hierarchiestufen des physischen Netzes nicht übereinstimmen muß. In verteilten Systemen werden logische Namen von Namensdiensten abgebildet, im Beispiel liefern sie die physische Knotenadresse des zuständigen Mail-Servers. Diese Funktion ist mit den „weißen Seiten“ des Telefonbuches zu vergleichen. Bisweilen werden auch „gelbe Seiten“ unterstützt: physische Adressen können auf Grund von charakterisierenden Beschreibungen (Attributen) ermittelt werden. In Abschnitt 9.6.1 wird auf Namensdienste im Rahmen von Entwicklungsplattformen eingegangen.

Die ITU- und OSI-Norm X.500 beruht auf einem Netz aus *Namensagenten* (*directory system agents*, *DSA*). Für hierarchische Namen wird der Namensraum mehrstufig in Kontexte (Teilbäume) eingeteilt, die von einzelnen oder replizierten Agenten verwaltet werden. Da X.500 nicht nur Knotennamen verwalten kann, muß vorab im *User Information Model* festgelegt werden, welche Objekttypen (für E-Mail vielleicht: Land, Organisation, Abteilung) mit welchen Attributtypen vorkommen und wie sie hierarchisch angeordnet werden. Auf Grund dieser größeren Flexibilität müssen Objekt- und teilweise Attributtypen bei Anfragen mitangegeben werden; eine E-Mail-Adresse könnte beispielsweise lauten: (Name = klaus_meier/Land = de/Organisation = uni-bonn/Abteilung = mathe).

Namensagenten verständigen sich über das *Directory System Protocol* und bei teilweiser Replikation der Verzeichnisse zudem über das *Directory Information Shadowing Protocol* und das *Directory Operational Binding Protocol*. Wie bei MOTIS gibt es Benutzeragenten (*Directory User Agents*). Sie kommunizieren mit den Namensagenten über das *Directory Access Protocol*. Dieses Protokoll benutzt gleich drei spezielle Dienstelemente sowie alle grundlegenden Dienstelemente außer CCR. Für die anderen genannten Protokolle werden weitere spezielle Dienstelemente eingeführt. Die Komplexität von X.500 ist also beträchtlich; sie hängt mit der Funktionsvielfalt zusammen, wie das folgende Beispiel zeigt. Anfragen eines Benutzeragenten nach Namen, die der angesprochene Namensagent nicht selbst abbilden kann, leitet er entweder an einen geeigneten Partner-Namensagenten weiter (*chaining*) oder, falls der geeignete Partner nicht bekannt ist, an eine Menge von geeignet erscheinenden Partnern (*multicasting*). Alternativ kann er auch nur die Adressen dieser geeigneten Partner dem Benutzeragenten zurückgeben. Häufig können Anfragen nur fragmentarisch vom lokalen Namensagenten beantwortet werden, während Teilanfragen weitergeleitet werden müssen.

Das Internet-DNS (*domain name service*) unterstützt ausschließlich Knoten- und (Sub-)Netz-Namen. Die Welt ist in Domänen aufgeteilt: einige wenige Nutzergruppen-Domänen wie „.edu“ für Ausbildungsstätten sowie eine Domäne pro Staat. Trotz beliebig wählbarer Tiefe der Namenshierarchie entsteht so ein einziger Baum von Namensagenten (hier: Domain Name Server, Verwalter einer „Domain Information Base“). Pro Anwen-

dungsklasse werden den Namen einfache Attribute zugeordnet: der E-Mail-Server aus dem Beispiel findet sich im *mx-record* des Namens „mathe.uni-bonn.de“. DNS ist weniger komplex als MOTIS. Ungewöhnlich für das Internet ist der Einsatz von Benutzeragenten, den sog. „resolvern“ (meist als Bibliotheksroutinen realisiert).

Netz-Management. Für das Netz-Management hat OSI ausführliche Modelle und Standards entwickelt. Basierend auf dem Begriff des *Managed Object* wurde eine komplette Architektur entwickelt, die durch zwei spezielle Dienstelemente unterstützt wird, *system management application* und *common management information*. Zusammen mit ACSE und ROSE werden sie als *system management application entity* bezeichnet, die kommunizierenden (nicht genormten) Prozesse als *agent application process* und *manager application process*. Der OSI-Ansatz fand viel Beachtung, doch konnte sich das *simple network management protocol* (*SNMP*) des Internet durch Firmware-Implementierungen auf Spezialhardware wie Router, Switches usw. bis jetzt besser behaupten. Zu Details dieser umfangreichen Normen siehe [Subramanian 00].

Sicherheit in verteilten Systemen. Im OSI-Kontext wurden bislang wenig konkrete Standards zum Aspekt Sicherheit verabschiedet. ITU-Standard X.800 ergänzt das OSI-Referenzmodell um eine sog. *Sicherheitsarchitektur*, die nur allgemein gehaltene Definitionen enthält und Hinweise darauf gibt, wo welche Sicherheitsaspekte verankert werden können. Als konkreter Standard spezifiziert X.509 Authentifikation und Schlüsselvergabe über den Namensdienst nach X.500. Außerhalb dieser Normen sind noch Internet-Aktivitäten zu nennen, insbesondere für E-Mail *privacy-enhanced mail* (*PEM*) und *pretty good privacy* (PGP) sowie eigens zur Abgrenzung privater und öffentlicher Netzteile eingesetzte Rechner (*firewalls*). Aus dem Athena-Projekt am MIT ist mit *Kerberos* eines der ersten verteilten Authentifikationssysteme hervorgegangen; weitere werden heute im Rahmen von Plattformen für verteilte Anwendungen angeboten (siehe Abschnitt 9.5). Näheres zum Thema Sicherheit in Netzen findet sich in [Canavan 01] und Kapitel E9.

Weitere Internet-Dienste und -Protokolle. Einerseits hat das Internet zu einigen OSI-Anwendungsdienstelementen keine Entsprechung, andererseits existieren viele weitere Internetdienste auf Anwendungsebene: Werkzeuge zur Online-Kommunikation von Benutzern wie *Talk*, *IRC* (für Gruppen) und audio-basierte Varianten, Informationsdienste (*Gopher*, *WAIS* usw.), Hypermedia-Informationsdienste (*World-Wide-Web*), elektronische schwarze Bretter (*News*), ferner Multimedia-fähige Verteildienste wie die *mbone*-Protokolle und -Werkzeuge. Anwendungsprotokolle für Internet-Fernsehen und Internet-Telefonie wie RSVP und RTCP werden in Kapitel E3 beschrieben.

9.5 Grundlagen der Entwicklung verteilter Anwendungen

9.5.1 Grundlegende Ansätze und Paradigmen

Der primitive, immer noch weitverbreitete Ansatz besteht darin, mit sequentiellen Programmiersprachen einzelne Programme zu entwickeln, die auf einen Kommunikationsdienst zugreifen. Das Modell der verteilten Anwendung existiert dabei während der gesamten Entwicklung nur im Kopf der Entwickler. Von UNIX und dem Internet-Transportprotokoll TCP her ist der Begriff *Socket-Programmierung* verbreitet. Während dieser Ansatz für die in Abschnitt 9.5.5 beschriebenen Punkt-zu-Punkt-Anwendungen noch annehmbar erscheint, kann die Entwicklung von aus Prozeß*netzen* bestehenden Anwendungen kaum mehr bewältigt werden. In diesem Abschnitt werden Ansätze besprochen,

die gezielt die Entwicklung kommunizierender Prozesse unterstützen. Bevor der wichtigste Ansatz detailliert behandelt wird, seien einige grundsätzliche Alternativen erwähnt:

- Beim *Betriebssystem-Ansatz* wird ein verteiltes Betriebssystem oder ein Netz-Betriebssystem zugrunde gelegt. (Zum Unterschied: im Netz-Betriebssystem bleibt die Möglichkeit erhalten, einzelne Rechner in herkömmlicher Weise zu nutzen.) In beiden Fällen erscheint das verteilte System dem Anwender und Programmierer wie ein einziger (nebenläufiger, also quasi Mehrprozessor-) Rechner, er kann daher Mechanismen und Programmiersprachen zur parallelen Programmierung verwenden. Das Netz- oder verteilte Betriebssystem übernimmt mit generellen Mechanismen viele Aufgaben, die sonst dem Entwickler verteilter Anwendungen zufallen; das ist zwar komfortabel, aber oft ineffizient. Die verringerte Autonomie und nötige Homogenität der einzelnen Knoten beschränkt diesen Ansatz auf Spezialfälle und lokale Netze.
- Der *Datenbank-Ansatz* bildet sämtliche Kooperationen zwischen Prozessen auf Datenbankzugriffe ab. Die Prozesse müssen sich daher nicht direkt kennen, sie können mit herkömmlichen sequentiellen Programmiersprachen entwickelt werden. Auch hier werden generelle und somit häufig ineffiziente Mechanismen eingesetzt. Schwerer wiegt das Problem, daß die Abbildung beliebiger Interprozeßkommunikation auf sternförmigen Datenbankzugriff oft schwierig und selten problemkonform ist.
- Der *Protokoll-Ansatz* stellt für häufig wiederkehrende Funktionen eines Anwendungsfeldes vorgefertigte Server bereit. Die Interprozeßkommunikation wird in einer Programmierschnittstelle versteckt, häufig nach dem Prinzip des Prozedur-Fernaufrufes (siehe weiter unten). Der Entwickler programmiert wie bei der Benutzung von Systembibliotheken. Beispiele sind das Xwindow-Protokoll und die Programmierschnittstellen von Mail- und Namensdiensten (siehe Abschnitt 9.5.5). Dieser Ansatz ist auf wenige, breit eingeführte Anwendungsfelder beschränkt.

Beim wichtigsten Ansatz, der *verteilten Programmierung*, wird der verteilte und nebenläufige Charakter der Anwendung nicht versteckt, sondern seine Beherrschung unterstützt. Einerseits benötigt die bestmögliche Unterstützung Wissen über alle beteiligten Prozesse, andererseits sind auch *offene* Anwendungen interessant, die mit getrennt entwickelten Prozessen kooperieren können. Für die verteilte Programmierung haben sich drei grundlegende Paradigmen herauskristallisiert, die nachfolgend in der Chronologie aufgelistet werden, in der sie Bedeutung gewannen: Interprozeß-Kommunikation, Prozedur-Fernaufruf und verteilter objektorientierter Ansatz. Realisiert wurden sie als Bibliotheken, Spracherweiterungen oder neue Programmiersprachen. Abschließend wird auf neuere Entwicklungen eingegangen, insbesondere mobile Agenten und Nachrichtendienste nach dem publish-subscribe-Modell.

9.5.2 Interprozeßkommunikation

Interprozeßkommunikation wurde zuerst vorgeschlagen. Dabei werden durch Konstrukte wie *send* und *receive* oder „!“ und „?“ Nachrichten zwischen zwei Prozessen ausgetauscht. Beim Senden wird neben dem Zielprozeß meist der Name einer Variablen, deren Inhalt versandt wird, oder eine Konstante spezifiziert. Das gleiche gilt auch für den Empfang. Varianten beziehen sich auf folgende Aspekte:

- Bei *synchroner* Kommunikation blockiert die Sende-Anweisung den Ablauf, bis der Empfänger die Nachricht entgegengenommen hat. Das vereinfacht die formale Handhabung und das Verständnis für das Programmverhalten, da im nebenläufigen Programm definierte Synchronisationspunkte entstehen. *Asynchrone* Kommunikation bezeichnet nicht-blockierendes Senden. Nicht-blockierendes Empfangen ist selten; es

liefert nur eine Nachricht zurück, falls diese bereits eingetroffen ist, und muß auf jeden Fall durch eine blockierende Variante ergänzt werden. Mit *Wächtern* (*guards*) können Empfänger ähnlich einer CASE-Anwendung auf unterschiedliche Sender oder Nachrichtentypen unterschiedlich reagieren; bei formalen Methoden bezeichnen Wächter eine genau definierte Semantik.

- *Direktadressierung* steht für den Ansatz, den Sende- oder Empfangsprozeß selbst in der Anweisung zu nennen. Dazu muß bei der Programmierung der Prozesse das gesamte Prozeßnetz bekannt sein. Diese starke Einschränkung vermeidet *indirekte Adressierung*, wobei Partnerprozeßnamen durch Stellvertreter ersetzt werden.
- *Verbindungsorientierte* Kommunikation verlangt zuerst den Aufbau von Verbindungen, idealerweise durch eine separate *Konfigurationsverwaltung* für die gesamte verteilte Anwendung. Werden sende- und empfangsseitig Stellvertreter für die Kommunikationspartner verwendet (oft *Sende-Port* und *Empfangs-Port* genannt), dann beziehen sich Sende- und Empfangsoperationen immer nur auf lokale Bezeichner; die Partner-Identität kann durchweg unbekannt bleiben. Typisierte Ports und Mehrpunkt-Verbindungen sowie hierarchische Prozeßstrukturen (unter Delegation von Ports an Unterprozesse) können den Programmierkomfort weiter erhöhen. *Verbindungslose* Kommunikation ist einfacher, aber weniger komfortabel. Indirekte Adressierung ist dabei auf die Verwendung eines gemeinsamen Stellvertreter-Bezeichners bei Sender und Empfänger beschränkt. Dieser modelliert meist eine Mailbox des Empfängers. Vom verteilten Laufzeitsystem können alle genannten Alternativen sowohl über verbindungsorientierte als auch über verbindungslose Transportdienste geführt werden.

Die Interprozeßkommunikation wurde bald vom Prozedur-Fernaufruf verdrängt (siehe Abschnitt 9.5.3). Derzeit zeichnet es sich ab, daß die verteilte objektorientierte Programmierung (siehe Abschnitt 9.5.4) wiederum den Prozedur-Fernaufruf ablösen könnte.

9.5.3 Prozedur-Fernaufruf

Der Prozedur-Fernaufruf (oder: entfernter Prozeduraufruf, *remote procedure call*, *RPC*) legt im Gegensatz zur Interprozeßkommunikation eine asymmetrische Kommunikationsstruktur zugrunde, bestehend aus Aufrufern und aufgerufenen Prozessen, meist *Client* und *Server* genannt. Eine Anfrage vom Client an den Server und eine Antwort in der Gegenrichtung werden stets gepaart und als Prozeduraufruf mit Übergabe der Aufruf- oder Ergebnisparameter „verkleidet“. Tatsächlich ist die weite Verbreitung des Prozedur-Fernaufrufs auf seine Ähnlichkeit mit dem herkömmlichen Prozeduraufruf zurückzuführen, die den Übergang von der herkömmlichen zur verteilten Programmierung scheinbar einfach gestaltet.

Schnittstellenbeschreibung. Systeme mit Prozedur-Fernaufruf umfassen fast immer Schnittstellenbeschreibungssprachen (*interface definition language*, *IDL*). Ihre Syntax ist oft an C oder C++ angelehnt und dient dazu, entfernte Prozeduren bzw. angebotene Prozeduren mit ihren Signaturen im Kontext des jeweiligen Programmes zu definieren. Dazu werden für gängige Programmiersprachen Präprozessoren angeboten, die aus Prozedur-Schnittstellenbeschreibungen in IDL automatisch wirtssprachenspezifische Codestücke erzeugen, sog. *stubs* (Stümpfe). Stubs sind lokale Stellvertreter des Servers beim Client und umgekehrt. Ein lokaler Prozeduraufruf beim Client führt zum Client-Stub. In ihm werden die Aufrufparameter in eine Nachricht gepackt (*marshalling*) und zum Server gesandt. Beim Server wartet ein prozedurspezifischer Server-Stub auf die Nachricht, packt sie aus (*unmarshalling*) und ruft lokal zum Server die eigentliche Prozedur auf. Die Ergebnisparameter nehmen den umgekehrten Weg.

IDL-Präprozessoren übernehmen neben der Stub-Generierung auch die Funktion der OSI-Darstellungsschicht, indem sie lokale Darstellungen in eine Transfersyntax übersetzen, häufig nach dem Internet-Standard XDR. Die abstrakte Syntax von Aufruf- und Rückkehrparametern wird dem IDL-Code entnommen.

Binden, Namens- und Maklerdienste. Die Gesamtheit der Aktivitäten zur Bekanntgabe, zum Auffinden und zur exakten Identifikation eines Servers und seiner exportierten Prozedur, passend zu einem Aufrufwunsch auf der Client-Seite, wird als Bindevorgang bezeichnet. („Binden“ übersetzt hier das englische „binding“, nicht „linking“; leider wird im Deutschen für beides derselbe Begriff verwendet.) Zu diesem Zweck kann ein spezieller Namensdienst (siehe Abschnitt 9.5.5) zwischengeschaltet werden, der einem suchenden Client auf Anfrage einen oder mehrere geeignete Server nennt. Wird vom Client eine exakte Angabe von Prozedur und Signatur verlangt, dann wird meist der eigentliche Begriff *Namensdienst* (*name service* oder auch *directory service*) gebraucht. Kann dagegen auch mit unvollständigen Namen, Attributen oder Optimierungskriterien angefragt werden (vergleichbar den „gelben Seiten“ des Telefonbuches), dann wird im RPC-Kontext meist der Begriff *Maklerdienst* (*trader*) verwendet.

Unterschiede zum lokalen Prozeduraufruf. Zunächst scheint der entfernte dem lokalen Prozeduraufruf sehr ähnlich zu sein. Im Detail zeigen sich aber vier Unterschiede, die etliche Tücken verteilter Anwendungen auf den Entwickler durchschlagen lassen:

1 *Unterschiedliche Adreßräume*: Da kommunizierende Prozesse verschiedene Adreßräume benutzen, sind nur Wertübergaben (call-by-value) direkt möglich. Namens- und Referenzübergabe sowie Zugriff auf globale Variablen können höchstens mit RPC-spezifischen Mechanismen nachgeahmt werden.

2 *Komplexere Fehlersemantik*: Während sequentielle Programme als Ganzes erfolgreich terminieren oder aber mit Fehler abbrechen, können in verteilten Programmen Client und Server alleine abbrechen und zahlreiche Netz- und Übertragungsfehler auftreten. Bei Fehlern muß insbesondere verhindert werden, daß Clients endlos in der Aufrufanweisung warten oder Server unnötig oder fehlerhaft die Ausführung verwaister Prozeduren (*Orphans*) weiterführen. Die Fehlerbehandlung nimmt oft einen beträchtlichen Anteil einer RPC-Implementierung ein und kann von Fall zu Fall stark variieren. Es hat sich eingebürgert, vier verschiedene Fehlersemantiken zu unterscheiden, die von konkreten RPC-Implementierungen unterschiedlich unterstützt werden:

   - Maybe-Semantik: Bei dieser schwächsten Form der Garantie werden endloses Warten und verwaiste Aufrufe verhindert, und die Mehrfach-Ausführung eines Aufrufs wird vereitelt. Im Fehlerfall ist aber nicht bekannt, wie weit die Abarbeitung gekommen ist; diese Semantik kann für Auskunftsdienste ausreichen.
   - At-Least-Once-Semantik: Von Komplett-Ausfällen der Serverseite abgesehen, wird hier Ausführung der entfernten Prozedur garantiert, allerdings ist Mehrfach-Ausführung nicht ausgeschlossen. Diese Form ist nur für Operationen geeignet, die problemlos mehrfach ausgeführt werden können.
   - At-Most-Once-Semantik: Hier wird die Maybe-Semantik zu atomarer Prozedurausführung erweitert; sie erfolgt vollständig oder wird nachwirkungsfrei abgebrochen. Die meisten Anwendungen können mit dieser Semantik korrekt entwickelt werden.
   - Exactly-Once-Semantik: Wenn bei At-Most-Once-Semantik ein komfortables Laufzeitsystem die Reaktion auf fehlerhafte Ausführung übernimmt, kann genau einmalige Ausführung garantiert werden. Das erforderliche verteilte Transaktionskonzept steht aber für viele Anwendungen *nicht* in einem sinnvollen Kosten-Nutzen-Verhältnis.

3 *Wunsch nach Nebenläufigkeit*: Um die Ähnlichkeit zum lokalen Prozeduraufruf groß zu halten, wird beim traditionellen RPC der Aufrufer bis zum Ende der entfernten Prozedurausführung blockiert. Ein Vorteil verteilter Systeme ist aber gerade die Möglichkeit der Parallelverarbeitung. Daher wurde *asynchroner RPC* in verschiedenen Varianten vorgeschlagen; die Ähnlichkeit zu lokalen Prozeduren sinkt dabei allerdings stark.
   - Bei reinem asynchronen RPC dürfen die Ergebnisparameter nach dem Prozeduraufruf beim Clienten nicht verwendet werden, bis die Ausführung abgeschlossen ist. Daher werden nach dem Aufruf möglichst viele Anweisungen eingeschoben, die die Ergebnisparameter nicht verwenden, und danach wird mit einer speziellen Anweisung auf das Ende des RPC gewartet.
   - Die Namen *futures* und *promises* bezeichnen Varianten der Automatisierung der soeben beschriebenen Vorgehensweise. Ergebnisparameter müssen dazu besonders deklariert werden, und der Übersetzer fügt Warteanweisungen automatisch an der spätestmöglichen Stelle ein.
   - Nebenläufigkeit trotz *synchronem* RPC kann mit nebenläufigen Programmiersprachen erreicht werden. Dabei teilt sich der Client vor dem Prozeduraufruf in mehrere *nebenläufige Zweige*, von denen einer am RPC blockiert wird; bevor Zweige auf Ergebnisparameter zugreifen, werden sie wieder zusammengeführt. Alternativ kann der Server in der aufgerufenen Prozedur einen Zweig starten und schnell wieder zurückkehren. Danach müssen Ergebnisparameter über einen zweiten RPC abgeholt werden, wodurch RPC allerdings zu einer Art umständlicher Interprozeßkommunikation wird.

4 *Optimierung für spezifische Kommunikationsmuster*: RPC ist für ein spezifisches Kommunikationsmuster ausgelegt: Erstens sind Anfrage und Antwort immer gepaart, zweitens wird eine asymmetrische Dienstnehmer-Diensterbringer-Beziehung zu Grunde gelegt, drittens wird auf Grund der Anlehnung an lokale Prozeduraufrufe die Übertragung für kurze Nachrichten und minimale Verzögerung optimiert. Zwar lassen sich im Prinzip alle Kommunikationsmuster mittels RPC realisieren, aber nicht immer effizient und problemnah. Muster, die asynchronen Nachrichtenaustausch an viele Partner ohne sofortige Antwort nahelegen, leiden an der überflüssigen Blockierung und Antwort; wechselnde Auftragsbeziehungen lassen sich schwer in das einfache Client-Server-Muster zwängen; usw. Multimedia- und Massendatentransfer erfordern teilweise so starke Erweiterungen, daß vom ursprünglichen RPC-Paradigma wenig übrig bleibt: Unter anderem wurde vorgeschlagen, asynchrone Kommunikation gemäß Abschnitt 9.5.2 für Massendatentransfer zu verwenden, nur geklammert von vor- und nachbereitenden RPC-Aufrufen.

9.5.4 Verteilter objektorientierter Ansatz

Die erste nennenswerte Realisierung des verteilten objektorientierten (*distributed object-oriented*, DOO) Paradigmas geht auf die Programmiersprache Emerald (etwa 1987) zurück. Anstatt Prozesse als Einheiten der Verteilung zu betrachten, erlauben DOO-Sprachen die flexible Konfiguration von Objekten, also wesentlich kleineren Einheiten. Die strenge Datenkapselung mittels Methoden (Operationen) verhindert Abhängigkeiten zwischen Objekten, die eine Plazierung auf verschiedenen Rechnern behindern könnten. Meist erfolgt die Plazierung zweistufig mit einer speziellen Konfigurationsverwaltung: Die Menge der Anwendungsobjekte wird auf sog. logische Knoten verteilt, diese werden vor dem Start der Anwendung auf physischen Rechnernetz-Knoten plaziert.

Lokationstransparenter Methodenaufruf. Die Aufteilung und Plazierung *nach* der Programmierung hat einen erheblichen Vorteil: Verteilungsrelevante, oft von Details des Zielsystems abhängige Entscheidungen müssen nicht zur Programmentwicklungszeit vorgenommen werden. Die späte Verteilung bringt aber auch die Notwendigkeit mit sich, lokale und entfernte Methodenaufrufe syntaktisch gleich zu halten. Aufrufe an entfernte Objekte gehen dabei meist zunächst an lokale *Proxy-Objekte*, und werden von dort aus an das entsprechende entfernte Objekt weitergeleitet.

Objektmigration. Schon das Laufzeitsystem von Emerald unterstützte die Verlagerung von Objekten zur Laufzeit. Voraussetzung hierfür sind global eindeutige Objektreferenzen sowie eine Erweiterung der Proxy-Objekte derart, daß Referenzketten über ehemalige Aufenthaltsorte von Objekten hinweg verfolgt (und dabei geschickt aktualisiert) werden. Da größere objektorientierte Anwendungen fast immer als System nebenläufiger Leichtgewicht-Prozesse realisiert werden, ist kaum vorhersehbar, wann zur Laufzeit welche Objekte aktiv sind. Es ist daher wichtig, die Objektmigration auch für Objekte zuzulassen, die gerade aufgerufen sind (und möglicherweise gerade Methoden anderer Objekte aufrufen). Dazu müssen verteilte Aufrufstapel unterstützt werden. Emerald faßt den zu einem einzelnen Methodenaufruf gehörenden Stapel-Abschnitt als „Aktivierungssatz“ zusammen, der gemeinsam mit dem aufgerufenen Objekt migriert. Da die Stapel bei den Leichtgewicht-Prozessen gehalten werden und nicht bei den Objekten, müssen bei Migration eines Objektes die Stapel gefunden werden, in denen Aktivierungssätze mit der entsprechenden Objektreferenz liegen. Die dazu notwendigen Rückwärts-Verweise von Objekten zu ihren Aktivierungssätzen werden häufig erst aktualisiert, bevor der aktive Thread unterbrochen wird, weil viele Aktivierungssätze diesen Zeitpunkt nicht erleben.

Objektmigration wird vom Anwendungsprogramm beim Laufzeitsystem angefordert, entweder explizit für ein bestimmtes Objekt per Programmanweisung oder implizit im Rahmen eines Methodenaufrufes. Im letzteren Fall migrieren alle Parameterobjekte zum aufgerufenen Objekt, so daß im Rahmen der Methodenausführung nur lokale Zugriffe auf Parameterobjekte stattfinden. Man unterscheidet hier zwei Unterarten: Bei *call by visit* migrieren die Parameterobjekte nach der Methodenausführung zum früheren Aufenthaltsort zurück. Bei *call by move* verbleiben sie am Zielort. Das ist bei nachfolgenden Methodenaufrufen mit teilweise gleichen Parametern sinnvoll. Während also infolge lokationstransparentem Methodenaufruf die Syntax verteilter objektorientierter Programmiersprachen der von herkömmlichen Sprachen gleicht, kommen für die Migrations*steuerung* Konstrukte hinzu: zur expliziten Migration, zur impliziten Migration wie soeben beschrieben, um Objekte von der Migration auszuschließen (etwa um sie bei ortsfesten Ressourcen zu halten) und um dies wieder rückgängig zu machen.

Optimierungsziel. Das Ziel Lastausgleich erscheint mit Objektmigrationen eher erreichbar als mit der traditionell untersuchten Prozeßmigration: das Anhalten, Migrieren und Fortsetzen kompletter Betriebssystemprozesse dauert oft länger, als die der Migrationsentscheidung zu Grunde liegenden Last-Meßdaten gültig sind. Selbst für die schnellere Objektmigration sind alternde Meßdaten problematisch. Alternativ wurden Migrationsentscheidungen beispielsweise auf Grund von statischem und dynamischem Wissen über die Kommunikationsintensität von Objekten getroffen. Ziel war es dabei, Kommunikation zwischen Objekten weitgehend lokal, also selten über Rechnergrenzen hinweg, durchzuführen; den genannten Methoden call-by-move und call-by-visit kommt dabei besondere Bedeutung zu. Die einfachste, optimale Lösung, alle Objekte auf einem Rechner zusammenzuführen, wird dabei durch Objekte verhindert, die bei ortsfesten Benutzern oder Ressourcen liegen müssen. Zunehmend wird versucht, durch *Objektreplikation* Fehlertole-

ranz *und* die anderen genannten Optimierungsziele zu erreichen; an die Stelle des Migrationsaufwandes tritt dabei der Aufwand zur Konsistenzhaltung der Kopien.

Offenheit und Heterogenität. Die meisten bisher bekannten verteilten objektorientierten Programmiersprachen sind *geschlossen*, d.h. alle Objektklassen sind bei Programmstart auf allen beteiligten Rechnern bekannt. Dadurch reduziert sich die Objektmigration auf die Verschiebung von Daten, der Methodencode wird vorab überall repliziert. In heterogenen Umgebungen muß allerdings das Problem der Datendarstellung gelöst werden, z.B. mit einer Transfersyntax wie in Abschnitt 9.4.1 beschrieben. Sollen offene Systeme in dem Sinn unterstützt werden, daß unabhängig entwickelte Anwendungsteile in einem verteilten System kooperieren können, dann müssen auch Klassenbeschreibungen, also Code, migrieren können. In heterogenen verteilten Systemen müssen diese Klassenbeschreibungen in einem architekturunabhängigen Format migrieren.

Java und Corba. Die in Kapitel E8.4 - Verweis worauf? beschriebene Programmiersprache Java wurde im Hinblick auf die Heterogenität des Internets entwickelt. Die Datendarstellung ist schon in der Programmiersprache selbst über alle Plattformen hinweg gleich; die Übersetzung in Transfersyntax entfällt (zulasten der Laufzeiteffizienz von Java). Klassenbeschreibungen werden in architekturneutralen Code der virtuellen Java-Maschine JVM, sog. Byte-Code, übersetzt; damit eignet sich Java grundsätzlich für Objektmigration und Offenheit. Das Standard-Laufzeitsystem für Java unterstützt allerdings nur die einmalige Migration des Methodencodes (von Java-Klassenbeschreibungen in Byte-Code): das Herunterladen von Applets von einem Server auf den aufrufenden Rechner. Die weitverbreitete Erweiterung Java-RMI fügt im wesentlichen die Funktionalität des Prozedurfernaufrufes hinzu. Die *Voyager*-Erweiterung der Firma Objectspace, frei erhältlich und zunehmend verbreitet, unterstützt dagegen in der neuesten Version ortstransparenten Methodenaufruf und Objektmigration. Letztere wird allerdings blockiert, solange das zu migrierende Objekt noch von einem Thread aufgerufen ist (sonst müßte JVM wie oben beschrieben erweitert werden). Zu Java in verteilten Systemen (siehe auch Abschnitt 9.5.5) siehe [Boger 99].

Der in Abschnitt 9.6.1 beschriebene Industriestandard *Corba* ist stärker mit dem Prozedurfernaufruf verwandt als mit den oben beschriebenen Ansätzen. Er unterstützt weder lokationstransparenten Methodenaufruf noch Objektmigration. Bei der Beschreibung weiter unten wird zudem darauf eingegangen, warum Corba so stark mit dem Prozeßbegriff (als Hülle einer Menge von Objekten) verknüpft ist, daß der Vorteil feingranularer Objekte sinkt.

Vergleich und Ausblick. Mit der beschriebenen vollen Funktionalität ist das DOO-Paradigma den anderen besprochenen Paradigmen in mehreren Punkten überlegen. Die strenge Kapselung der Objekte unterstützt nebenwirkungsfreies Programmieren, so daß Abhängigkeiten zwischen Einheiten der Verteilung minimal bleiben. Dienstnehmer-Diensterbringer-ähnliche Kommunikationsmuster werden nicht erzwungen wie beim Prozedurfernaufruf. Feingranulare Objekte können bei der initialen Konfiguration besser an Zielsysteme angepaßt werden als grobgranulare Prozesse. Migration und Replikation bieten weitere Möglichkeiten, verteilte Systeme effizient zu nutzen. Klassenkonzept und spätes Binden sind ideale Voraussetzungen für die Erstellung heterogener und evolutionärer Software (offener verteilter Anwendungen). Allerdings steckt deren Unterstützung noch in einem frühen Stadium, da funktional reiche akademische Sprachen offene Anwendungen häufig nicht unterstützen und Standards wie Corba das DOO-Paradigma noch recht unvollständig umsetzen.

9.5.5 Weitere Ansätze

Vor einigen Jahren wurde erkannt, daß die bisher besprochenen Prinzipien im explodierenden Internet Grenzen haben, weil die Initiative im verteilten Programm stark auf der Seite der Nachrichten-Konsumenten bzw. Klienten liegt („Pull-Prinzip“). Bei der Interprozeßkommunikation wird nur empfangen, wenn der Empfänger „sich entschließt“, eine Empfangsoperation anzustoßen; selbst die Vorabübermittlung von Nachrichten auf den Rechner des Empfängers setzt erstens eine existierende Kommunikationsverbindung voraus und ist zweitens nach dem Prinzip der Flußkontrolle auf wenige Nachrichten beschränkt; sonst bleiben Nachrichten bis zur Abholung beim Produzenten. Ist also der Kommunikationsablauf nicht vorab eindeutig bekannt, so muß der Konsument häufig nachprüfen, ob der Produzent eine Nachricht bereit hält oder nicht. Wenn die Produzenten nicht alle bekannt sind oder ihre Zahl größer ist als die zulässige Anzahl gleichzeitiger Kommunikationsverbindungen, stößt das Pull-Prinzip vollends an seine Grenzen. Auch Prozedurfernaufruf und entfernter Methodenaufruf gehen davon aus, daß der Klient die Initiative für den Aufruf einer Dienstleistung hat.

Die Möglichkeit, daß Produzenten bei Auftreten von Ereignissen bzw. Anfallen neuer Nachrichten selbst aktiv werden, wird als „Push-Prinzip“ bezeichnet. Nachfolgend werden Punkt-zu-Punkt-Nachrichtendienste als Vorläufer, Ereignisdienste als wichtigste Vertreter, sowie (kurz) verteilter gemeinsamer Speicher und Tupelräume als spezielle Varianten besprochen.

Punkt-zu-Punkt-Nachrichtendienste. Dieses Prinzip kann als erster Schritt in Richtung Pull-Prinzip gesehen werden; es wird seit Jahren neben den drei genannten Ansätzen Interprozeßkommunikation, Prozedurfernaufruf und verteilte objektorientierte Programmierung verfolgt, in der Forschung aber kaum beachtet. Eine gegebenenfalls replizierte Nachrichtenzentrale nimmt alle im Prozeßnetz versandten Nachrichten entgegen und speichert sie persistent bis zur Abholung durch einen Empfänger oder dem Ablauf der Gültigkeitsdauer. Da die Nachrichtenzentrale normalerweise alle Nachrichten (einer bestimmten Klasse) nach dem Fifo-Prinzip ordnet, hat sich der Begriff „message queue“ für das Prinzip eingebürgert. In der industriellen Praxis werden häufig Systeme eingesetzt, die von großen Systemsoftwareanbietern (IBM MQ-Series) oder auf die Unterstützung verteilter Anwendungen spezialisierten Firmen wie Tibco und BEA angeboten werden. Das Prinzip eignet sich sehr gut für offene verteilte Anwendungen, weil Produzenten und Konsumenten von Nachrichten stark entkoppelt sind. Aus dem gleichen Grund kann die Anzahl der Kunden und die Anzahl der Konsumenten (Server) leicht erweitert werden – falls der Nachrichtendienst selbst genügend Leistungssteigerung zuläßt. Für die Integration von bestehenden Anwendungen (auf beiden Seiten) werden häufig leistungsfähige Werkzeuge zur Format-Anpassung angeboten - so können bei neuen Kunden beispielsweise deren existierende e-Mail-Anwendungen eingebunden werden.

Ereignisdienste mit Verteilcharakter. Das Prinzip der Nachrichtendienste, angepaßt an Internet-Verhältnisse, entspricht dem Zugriff beliebiger Internet-Nutzer auf Informationsquellen mit Verteilcharakter. Im Gegensatz zur Nutzung von Suchmaschinen soll der Benutzer „automatisch“ mit neuen Informationen versorgt werden, von klassischen Nachrichten im populären Sinne bis zu neuen Softwareversionen. Während es für den ersteren Fall noch denkbar ist, den Klienten täglich beim Informationsanbieter die Nachrichten abholen zu lassen, steht der letztere Fall beispielhaft für viele Anbieter mit stark schwankender Ereignishäufigkeit. Der Begriff *Ereignisdienst* (*event service* oder *event routing*) hat sich bei Diensten mit Verteilcharakter durchgesetzt, also dort, wo eine Nachricht (ein „Ereignis“) potentiell an viele interessierte Konsumenten übermittelt wird. Zunehmend werden solche Dienste für sinnvollen Betrieb sehr großer Netze als unerläßlich angese-

hen. Soll die notwendige Registrierung der Informations-Konsumenten beim Anbieter betont werden, so verwendet man häufig den Begriff *publish-subscribe*: der Produzent „veröffentlicht“ in einem Medium, der Konsument „abonniert“.

Folgende Klassen und Merkmale von Ereignisdiensten schälen sich heraus:

- Medien-Typen: Kanal-bezogenes Abonnement (channel based subscription) bedeutet, daß alles auf einem Kanal Veröffentlichte (im allgemeinen entsprechend einem Produzenten, gegebenenfalls einem Thema) empfangen wird; zwei Kanäle haben keinen Bezug. Themenbezogenes (topic based) Abonnement ordnet Themen hierarchisch ähnlich Internet-News. Wer z.B. *kurse.geld* abonniert, erhält Nachrichten aus *kurse* und aus *kurse.geld.dollar*, nicht aber aus *kurse.aktien*.
  Der vielversprechendste Kompromiß aus Ausdrucksstärke und Skalierbarkeit ist vermutlich Betreff-bezogenes (subject-based) Abonnement. Es bezieht sich auf ausgezeichnete Felder jeder einzelnen Nachricht, vergleichbar dem Betreff einer eMail, aber strukturierter (z.B. als Attribut-Wert-Paare). Abonnements dürfen üblicherweise als reguläre Ausdrücke über Betreffs-Attribute formuliert werden, was sehr viel Flexibilität auf der Konsumentenseite ermöglicht. Die Schwierigkeit bei weltweiten Ereignisdiensten besteht darin, Ereignisse vom Produzenten möglichst per Multicast ohne unnötige Duplikate an genau diejenigen Zwischenkonten (Router) weiterzuleiten, über die die abonnierten Konsumenten erreichbar sind. Es ist aber undenkbar, in jedem Router Millionen von regulären Ausdrücken aller Konsumenten zu speichern. Daher wird versucht, die regulären Ausdrücke zu einem kompakteren sog. Ereignisfilter zusammenzufassen, der die Menge aller für einen Rechnernetzknoten relevanten Abonnements beschreibt. Jeder Router wiederum faßt die Ereignisfilter aller über ihn erreichbaren Knoten zu einem neuen Ereignisfilter zusammen und gibt diesen an die „vor ihm“ liegenden Router weiter und so fort. Die vorstehende Erörterung macht deutlich, warum der vierte Medien-Typ keine gute Skalierbarkeit verspricht: sogenannte Inhaltsbezogene (content based) Abonnements beziehen sich auf den gesamten Nachrichten-Inhalt und vergrößern damit üblicherweise den Aufwand zur Filterung mittels Ereignisfiltern beträchtlich; meist liegt der Schwerpunkt solcher Systeme in der Ausdrucksstärke; daher sind auch die Vorschriften für die Formulierung der regulären Ausdrücke üblicherweise sehr großzügig, mit dem Nachteil schlechter Chancen für die Vereinfachung in Ereignisfiltern.
- Abonnements und Anzeigen: rein Abonnement-bezogene Ereignisdienste können nur zum Vermittlungszeitpunkt die Nachrichtenflut eindämmen. Gerade bei Betreff-bezogenen Abonnements kommen aber alle Produzenten potentiell als Lieferanten für alle Konsumenten in Frage. Verlangt man dagegen auch von den Produzenten vorab Angaben darüber, welche Art von Ereignissen sie zu publizieren gedenken, dann können Ereignisfilter auch von den Produzenten in Richtung zu den Konsumenten durch die Router gereicht werden. Bei der Verschmelzung mit konsumentenbezogenen Filtern können häufig viele Kommunikationspfade von vorne herein ausgeschlossen werden. Die Vorab-Ankündigung eines Produzenten wird Anzeige (advertisement) genannt.
- Topologien: die den Ereignisdienst realisierenden Rechnernetzknoten (Server oder Router genannt) können im einfachsten Fall zentralisiert sein, bei klarer Trennung zwischen Produzenten und (vielen) Konsumenten kommt auch eine hierarchische Struktur in Frage (beim Hersteller Tibco anzutreffen); für überregionale Ereignisdienste im Internet sind nur azyklische oder allgemeine nichtzentralistische (peer-to-peer) Strukturen vorstellbar, wie sie SIENA im Forschungsstadium untersucht.

Die Übergänge zwischen den hier eingeführten Unterscheidungen sind in der Praxis oft fließend. Einerseits werden in Punkt-zu-Punkt-Nachrichtendiensten zunehmend verschie-

dene Gruppenkommunikationsmuster unterstützt, vor allem die Auslieferung unter den an einem Nachrichtentyp interessierten Prozessean an „garantiert alle“ und an „genau einen“. Andererseits enthalten Plattformen für verteilte Anwendungen wie in Abschnitt 9.6 besprochen, die vornehmlich Prozedurfernaufruf oder verteilte objektorientierte Programmierung unterstützen, zunehmend Ereignisdienste.

Tupelräume und virtueller gemeinsamer Speicher. Die maximale Übertragungsrate in drahtgebundenen Netzen steigt exponentiell und dabei noch schneller als Prozessor-Geschwindigkeiten, die sich laut „Moores Gesetz“ rund alle 18 Monate verdoppeln. Dadurch reduziert sich der „Preis“ verteilter Berechnungen im Vergleich zu lokalen Berechnungen, der bislang noch gekennzeichnet ist durch mehrere Größenordnung Laufzeitunterschied zwischen lokalem und entferntem Prozeduraufruf. Schon heute kommen in lokalen Netzen zunehmend Mechanismen zum Einsatz, die in der Vergangenheit nur für Parallelrechner bzw. Mehrprozessorsysteme als effizient galten. Einer davon ist *virtueller gemeinsamer Speicher*, bei dem zur verteilten Programmierung Konstrukte angeboten werden, die das Vorhandensein von gemeinsamem Speicher simulieren. Da in Wahrheit Fernzugriff auf Speicher bzw. Konsistenterhaltung von repliziertem Speicher realisiert werden muß, also Kommunikation, während der Programmierer in Konzepten gemeinsamen Speichers „denkt“, ist das Leistungsverhalten bislang nur für bestimmte Anwendungsdomänen zufriedenstellend. *Tupelräume* kann man sich als sehr einfache Variante von semistrukturiertem gemeinsamem Speicher vorstellen. Alle Prozesse eines verteilten Programmes sind nur mit dem Tupelraum verbunden, in den sie (im allgemeinen beliebige) Tupel, also Listen von Werten, ablegen können. Neben dieser sog. out-Operation gibt es nur noch zwei weitere, in und read, für erhaltendes und löschendes Lesen aus dem Tupelraum. Den ersten Realisierungen, bekannt geworden vor allem mit der Programmiersprache Linda, fehlte es noch an Effizienz, Persistenz und Konsistenz (z.B. bei gleichzeitigem read- und in-Zugriff). Inzwischen haben die JSpaces von Java und die persistenten, transaktionsorientierten TSpaces von IBM zu vielen industriellen Anwendungen geführt.

Mobile Objekte und mobile Agenten. Wie in Abschnitt 9.5.4 erwähnt, hat sich der verteilte objektorientierte Ansatz lange Zeit nur wenig vom Prozedurfernaufruf unterschieden. Die in der akademischen Forschung lange verfolgte Idee der Objektmigration, also *mobiler Objekte*, bekam erst mit der Einführung von Java-Applets (in einer einfachen Variante) industrielle Bedeutung. Dadurch wurde das jahrzehntelang dominierende Konzept der Datenübertragung durch das der Code-Übertragung ergänzt. Einen weiteren Schritt in dieser Richtung stellen *mobile Agenten* dar, die den in letzter Zeit vielbeachteten Begriff des Software-Agenten mit verteilten Anwendungen in Beziehung bringen. Im Vordergrund stehen dabei nicht Fähigkeiten wie das Anstellen von Vermutungen, die die intelligenten Agenten der künstlichen Intelligenz auszeichnen. Mobile Agenten sind vielmehr autonome Softwareprozesse, welche (1) von einem Benutzer einen für diesen leicht verständlichen Auftrag erhalten (z.B. „kaufe preisgünstige Hauptspeichermodule für meinen PC“) (2) den Auftrag in endlicher Zeit erfüllen sollen; (3) dabei auf mehreren Knoten eines verteilten DV-Systems zur Ausführung kommen (üblicherweise durch Migration) und (4) während der Auftragsausführung mit der Umwelt, insbesondere anderen Agenten, kommunizieren. Für die Realisierung liegt es nahe, verteilte objektorientierte Ansätze mit Migrationsunterstützung zu verwenden, insbesondere Java-Erweiterungen wie das oben angeführte Voyager. Tatsächlich werden Java-basierte Ansätze (z.B. Voyager, IBM-Aglets, Concordia) für die Agentenentwicklung in letzter Zeit häufiger eingesetzt als die ursprünglich vieldiskutierten Sprachen wie Agent-Tcl. In Voyager kommt zu der in Abschnitt 9.5.4 beschriebenen Funktionalität vor allem die Unterstützung aktiver Objekte

und von Persistenz hinzu. Für die Kommunikation zwischen Agenten in offenen Systemen werden spezielle Protokolle und Beschreibungssprachen verwendet.

Multimedia-Ströme. Wie in Kapitel E3.2.3 ausgeführt, bezeichnet man die Kommunikationsbeziehung bei Übertragung zeitabhängiger Medien als „Strom“. Das der Interprozeßkommunikation zugrundeliegende Modell entspricht dieser Beziehung besser als die anderen in diesem Kapitel beschriebenen Programmiermodelle. Allerdings treten neben Punkt-zu-Punkt-Beziehungen häufig Verteilbeziehungen (multicast), alle-an-alle-Beziehungen (anycast, z. B. bei Videokonferenzen) und andere vermaschte Strukturen. Neuere Standards für Multimedia-Ströme wie MPEG-4 mischen Datenkommunikation und mobilen Code, einige Internet-Werkzeuge und -Dienste für Video unterstützen zudem das Push-Prinzip. So könnte man einerseits behaupten, Multimedia-Ströme seien die Vereinigung und Weiterentwicklung fast aller bisher besprochener Prinzipien. Andererseits werden Ströme und Gruppenkommunikation heute meist auf dem Abstraktionsniveau der anfänglich erwähnten Socketkommunikation angeboten, haben also noch nicht wesentlich in Programmiersprachen Eingang gefunden. In dieser Hinsicht liegen sie in der Entwicklung also noch zurück.

In Bild 2 werden die besprochenen Ansätze zusammengefaßt dargestellt.

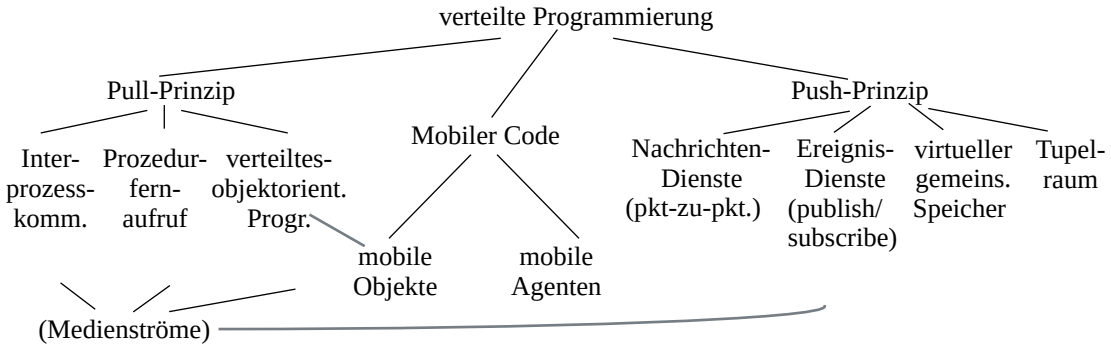

Bild 2 Zusammenfassung der Varianten verteilter Programmierung

9.6 Plattformen für verteilte Anwendungen

Der scheinbar geringe Lernaufwand für RPC infolge Ähnlichkeit zu lokalen Prozeduren ist nur einer der Gründe für seine Verbreitung. Ein anderer sind standardisierte Entwicklungs- und Laufzeitumgebungen (nachfolgend kurz: *Plattformen*). Eine solche Plattform enthält Dienste und Werkzeuge, die die Entwicklung und Ausführung verteilter Anwendungen erleichtern. Sie ist zudem bei zunehmender Verbreitung attraktiv als Basis des wachsenden Marktes für Komponentensoftware, also wiederverwendbare Softwaremodule.

In diesem Abschnitt soll zunächst die Plattform DCE für Prozedur-Fernaufruf besprochen werden, zusammen und im Vergleich mit der in den Zielen sehr ähnlichen Plattform Corba für verteilte objektorientierte Anwendungen. Dann werden kurz OSI-konforme Ansätze vorgestellt, vor allem das Anwendungsdienstelement ROSE sowie der ISO-Standard ODP, der als anwendungsorientiertes Pendant zum OSI-Referenzmodell aufgefaßt werden kann. Zu den Plattformen im Vergleich siehe [Schill 96].

9.6.1 DCE und Corba

DCE. Das *distributed computing environment* (*DCE*) wurde vom industriellen Gremium *open software foundation* (*OSF*) standardisiert. Wesentlich bei DCE ist die Aufteilung verteilter Anwendungen in disjunkte *Zellen*, die jeweils mehrere Knoten und Prozesse (Clients, Server) enthalten. Zellen hat man sich als zusammengehörige Verwaltungsbereiche, z.B. Abteilungen, vorzustellen. Zu DCE gehören eine an C angelehnte Schnittstellenbeschreibungssprache und sog. Stub-Compiler (siehe Abschnitt 9.5.3).

Bild 3 zeigt den Aufbau der DCE-Laufzeitkomponenten, von denen die eigentliche RPC-Unterstützung nur einen kleinen Teil ausmacht. Hinzu kommt das Threads-Package, mit dem nebenläufige Leichtgewicht-Prozesse in Clients und Servern realisiert werden können, unabhängig von der Programmiersprache.

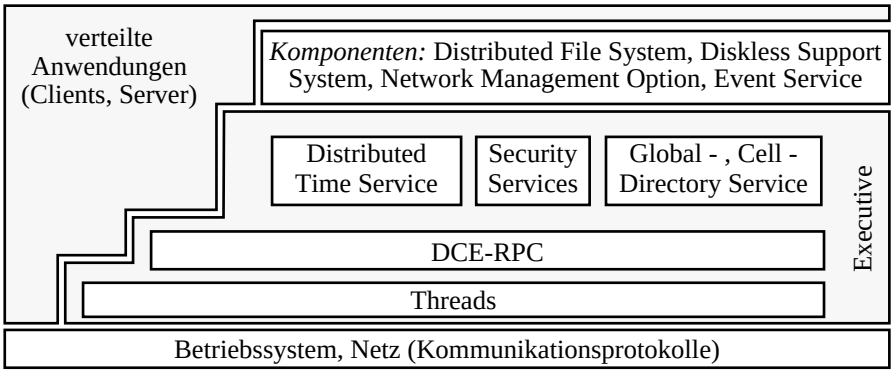

Bild 3 DCE-Laufzeitsystem

Drei Dienste(-Gruppen) erweitern die Funktionalität der RPC-Unterstützung. Sie werden mit Threads und RPC-Unterstützung zur *DCE-Exekutive* zusammengefaßt:

- *Directory Services*, unterteilt gemäß Zellenkonzept in einen zellenübergreifenden, globalen Namensdienst (global directory service) nach X.500 und in einen zelleninternen Dienst (cell directory service);
- *Distributed Time Service* zur Uhrensynchronisation, der ebenfalls zelleninterne und zellenübergreifende Funktionen unterscheidet; Ungenauigkeiten infolge Übertragungsverzögerung werden darin durch iterative Schätzung der Verzögerung reduziert;
- *Security Services* zur Authentisierung von Clients, zur Authorisierung von deren Serverzugriff und zur Zusicherung von Datenintegrität und -schutz durch Verschlüsselung; üblicherweise ist die Vertrauenswürdigkeit innerhalb einer Zelle höher als zwischen Zellen.

Hinzu kommt eine wachsende Anzahl von Diensten, die die DCE-Exekutive nutzen, um für verteilte Systeme allgemein sinnvolle Funktionalität anzubieten. Zu nennen sind insbesondere ein verteiltes Dateisystem (distributed file system), die Unterstützung für plattenlose Arbeitsstationen (Diskless Support System), ein Aufsatz für Netz-Management (Network Management Option) und ein Dienst zur Weiterleitung von Ereignissen an alle am betreffenden Ereignistyp interessierten Prozesse (Event Service).

Corba. Corba steht für *common object request broker architecture* und ist ein Industriestandard der Object Management Group. Corba wird meist und nachfolgend als umfassender Begriff benutzt, der mehrere ineinandergreifende Aktivitäten und Architekturen

einschließt, insbesondere die *object management architecture*, die mit der DCE-Architektur von Bild 3 direkt vergleichbar ist. Genaugenommen betrifft Corba nur den Kern dieser Architektur, den *object request broker* (*ORB*). Diese einleitende Fülle von Begriffen ist symptomatisch für Corba; der vorgesehene Endausbau enthält eine fast erdrückende Anzahl von Komponenten, Begriffen und Diensten, daher sind in Bild 4 auch nur Grobklassen dargestellt.

Auch in Corba wird eine Schnittstellenbeschreibungssprache (die Corba-IDL) verwendet; sie ist an C++ angelehnt. Prozesse, die Objekte und Methoden zum Aufruf anbieten, auch in Corba „Server“ genannt, müssen sich über den sog. Objekt-Adapter beim Object Request Broker anmelden, der auf Basis der mitgelieferten IDL-Beschreibung Stubs für den Aufruf generiert (siehe Bild 4). Will ein Prozeß quasi als Client entfernte Methoden aufrufen, beschreibt er diese ebenfalls in IDL; zur Übersetzungszeit wird daraus ein *statischer Stub* generiert. Als Alternative kann der Methodenaufruf dynamisch zusammengestellt werden, womit Polymorphie mit spätem Binden im objektorientierten Sinne erreicht wird; die Laufzeit-Effizienz dieser Variante ist allerdings deutlich geringer. Der Object Request Broker übernimmt die Vermittlung zwischen Client- und Server-Schnittstelle *und* die Funktion des Namens- und Maklerdienstes. Diese und weitere Funktionen sind über eine allgemeine Schnittstelle zugänglich.

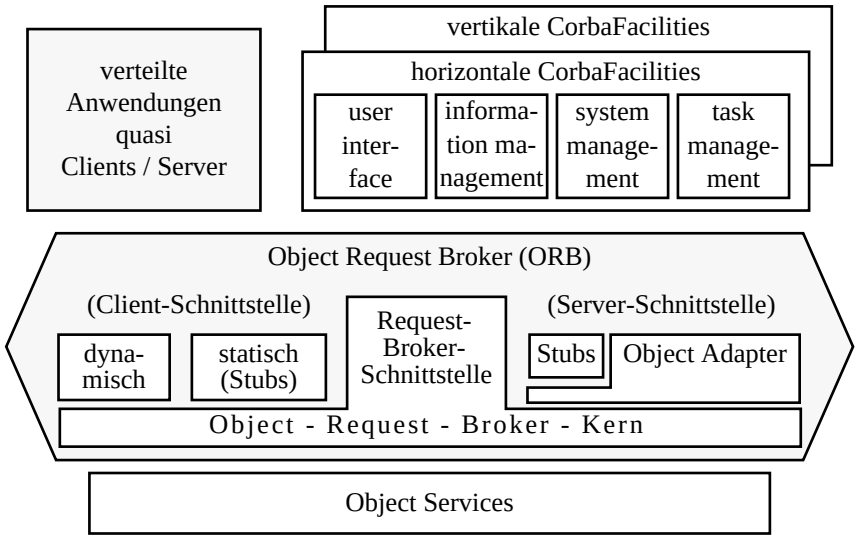

Bild 4 Corba: Object Management Architecture mit Object Request Broker

Vergleichbar der DCE-Exekutive bietet Corba Objekt-Dienste an, die als integraler Bestandteil einer verteilten Anwendung sinnvoll sein können. Zum Zeitpunkt der Verabschiedung von Corba 2.0 waren mehr als zwei Drittel von rund zwanzig Diensten fertig spezifiziert, etwa für Persistenz, Transaktionen und Nebenläufigkeit, andere in Entwicklung oder erst geplant. Corba-Dienste sind vergleichbar mit DCE-Komponenten, aber in weit größerer Anzahl spezifiziert oder in Diskussion. *Vertikale* Dienste sind jeweils auf eine Anwendungsdomäne zugeschnitten (etwa Finanzmärkte, Öl-/Gas-Industrie, Bildverarbeitung). *Horizontale* Dienste werden in die vier in Bild 4 genannten Untergruppen gegliedert; Beispiele umfassen bei Benutzerschnittstellen und Informationssystemen die Unterstützung offener verteilter Dokumente (compound documents), Agentenunterstützung bei der Systemverwaltung und Dienstgüteunterstützung bei der Prozeßverwaltung.

Vergleich. Obwohl DCE dem RPC-Paradigma zuzuordnen ist, können damit auch verteilte objektorientierte Anwendungen und sogar Programmiersprachen entwickelt werden. Selbst zur Identifikation von Objekten und -typen innerhalb eines Servers bietet DCE ein Konzept, die *universal unique identifiers*. Die Liste der Punkte, die Corba wirklich geeigneter für die Objektorientierung machen, ist kurz:

- Corba bietet die dynamische Schnittstelle zusätzlich zur üblichen Stub-Generierung.
- Die Corba-IDL ermöglicht Vererbung, sogar Mehrfach-Vererbung.
- Neben RPC-ähnlichen synchronen Methodenaufrufen gibt es zwei asynchrone Varianten; eine muß ohne Ergebnisparameter auskommen und stößt die entfernte Methode nur an, bei der anderen kann mit separatem Aufruf auf Ergebnisse gewartet werden.

Prozeßlokaler Methodenaufruf und Fernaufruf unterscheiden sich in Corba syntaktisch und im Zeitverhalten erheblich. Typische Corba-Anwendungen machen daher meist nur grobgranulare Objekte im Netz bekannt, die eigentlich Prozesse darstellen. Es entstehen die für RPC und Interprozeßkommunikation typischen Prozeßnetze statt der für DOO typischen feingranularen, dynamisch migrierenden Objekte. Entsprechend können von vier Corba-Servertypen drei nur ein *einziges* von außen referenzierbares Objekt anbieten. Zusammenfassend läßt sich sagen, daß Corba bislang eher zwischen den beiden in Abschnitt 9.5 beschriebenen Paradigmen DOO und RPC liegt. Das Corba-kompatible SOM2 von IBM kommt dem DOO-Paradigma wesentlich näher. Hervorzuheben an Corba ist die Unterstützung offener heterogener Softwaresysteme, vor allem seit Interoperabilität zwischen verschiedenen Laufzeitsystemen unterstützt wird.

9.6.2 ROSE und ODP / ANSAware

ROSE. Fast alle RPC-Ansätze ziehen kompakte Schnittstellenbeschreibungssprachen und einfache Transfersyntaxen den OSI-Standards für Schicht 6 vor. Dies gilt natürlich nicht für den entsprechenden OSI-Ansatz, nämlich das Anwendungsdienstelement *ROSE* (*remote operations service element*). Da sich das OSI-Referenzmodell auf die reine *Kommunikation* in offenen Systemen beschränkt, werden Bindevorgang, Fehlersemantik und Makler-Dienste nicht behandelt. Zudem geht ROSE von gleichberechtigten Kommunikationspartnern aus, verlangt also keine eindeutige Client-Server-Beziehung. Für die unbestätigten Dienstfunktionen *RO-Invoke* und *RO-Result* heißt das, daß dem RO-Invoke.Ind, welches offensichtlich den Empfang eines RPC-Aufrufs beim ausführenden Prozeß modelliert, nicht zwingend ein RO-Result.Req folgen muß. Weitere Dienstfunktionen regeln die Verständigung über fehlerhafte Operationsausführung sowie Zurückweisung durch Diensterbringer oder Beantworter. Zur Assoziationsverwaltung wird ACSE vorausgesetzt, für Massendatentransfer ist eine Kombination mit RTSE vorgesehen.

ODP. Das Referenzmodell *open distributed processing* (*ODP*) der ISO ist das Pendant zum OSI-Referenzmodell für verteilte Anwendungen. Es hat aber insofern einen weitergehenden Anspruch, als es den Software-Lebenszyklus mit umfaßt. Kerngedanke ist die Dekomposition der Anwendung in fünf verschiedene Blickwinkel, sog. *Viewpoints*; diese greifen Teilaspekte heraus, die für unterschiedliche, in die Softwareentwicklung einbezogene Gruppen wichtig sind:

- Der *Enterprise Viewpoint* modelliert die Unternehmenssichtweise, also die Einbettung der entstehenden Anwendung in ein Unternehmen, vor allem die Benutzung durch Mitarbeiter. Dazu stehen Modellelemente wie Rollen und Aktivitäten, Organisationsstrukturen sowie Sicherheit und Verwaltung aus Sicht der Organisation im Vordergrund.

- Im *Information Viewpoint* werden wesentliche Informationen eines Unternehmens modelliert sowie deren Fluß durch die verteilte Anwendung und die Gesamtorganisation. Datenbankaspekte und elektronische Dokumente werden betrachtet sowie automatisierte *und* manuelle Verarbeitungsschritte (Editieren, Versenden usw.).
- Der *Computation Viewpoint* betrifft die Zerlegung der Anwendung in Objekte, die zusammen die Funktionalität und Informationsstrukturen als Softwaresystem verteilungsunabhängig beschreiben. Der verwendete Objektbegriff bezeichnet kommunizierende Prozesse und entspricht nicht demjenigen objektorientierter Programmiersprachen.
- Im Kontext des *Engineering Viewpoint* werden die noch relativ abstrakten Objekte des Computation Viewpoint realisierungsnah verfeinert und erweitert durch Prozesse, Speicher, Netze usw. Beispielsweise wird im vorangegangenen Blickwinkel Synchronisation nur hinsichtlich der Synchronisationspunkte spezifiziert und Kommunikation hinsichtlich der Partner und Daten; hier wird dazu das „Wie“ beschrieben.
- Noch eine Stufe „technischer“ ist der *Technology Viewpoint*, in dem auf die Hardware und Ressourcen Bezug genommen wird. Dieser Blickwinkel kann nur im Rahmen einer konkreten Installation vollständig ausgeführt werden.

ANSAware, J2EE und .NET. Verschiedene Forschungs- und Entwicklungsprojekte führten zu Implementierungen von ODP-konformen Plattformen. Das von der Europäischen Union geförderte ANSA- und spätere ISA-Projekt brachte mit ANSAware eine der ersten erhältlichen Plattformen und beeinflußte anfangs den Standardisierungsprozeß stark. Heute geht die ODP-Norm teilweise über den Stand von ANSAware hinaus, beispielsweise bei den Sicherheitskonzepten, dennoch gilt ANSAware als wichtige Referenzimplementierung mit kommerzieller Bedeutung. Das Laufzeitsystem ähnelt stark dem von DCE. ANSAware war vor allem hinsichtlich der Maklerfunktionen wegweisend und unterstützte zuerst die Prüfung kompletter Signaturen bei der Anfrage eines Clienten nach einem Server, die Suche an Hand von Attributen, verschiedene Auswahlstrategien (zyklisch, zufällig usw.) und fortgeschrittene Konzepte der Abstimmung zwischen verschiedenen Maklerdiensten für netzweite Server-Vermittlung (Trader-Federation).

Alle bisher beschriebenen Middleware-Ansätze sinken in ihrer Bedeutung. DCE krankte an der fehlenden Unterstützung objekt-orientierter Ansätze, und ODP war wie alle ISO-Ansätze zu kompliziert und am grünen Tisch entworfen. Beide sind nur noch durch überlebende Anwendungen von Bedeutung. Mit Corba wird noch entwickelt, doch gilt es als langsam und überladen. Die strenge Objektorientierung kombiniert mit „fetten“ Objekten von Prozeßgröße (im Widerspruch zum Geist der Objektorientierung) behindert Modularisierung. Die Programmiersprachen-Unabhängigkeit und –Interoperabilität erfordert leistungsmindernde Abbildungen. Die Fülle horizontaler und vertikaler Dienste sowie nachträglich aufgepfropfte Paradigmen komplizieren Corba weiter.

Vor diesem Hintergrund ist der Erfolg von J2EE verständlich, der „Java 2 Enterprise Edition“. J2EE kombiniert Objektorientierung mit einem orthogonalen Komponentenmodell (von Microsoft mit COM seit jeher favorisiert) und vermeidet durch die Beschränkung auf Java viele Effizienzprobleme von Corba – die durch die Verwendung von virtueller Java-Maschine und Bytecode allerdings teilweise an anderer Stelle auftreten. Zentrales Konzept ist der Container, genannt „Bean“, als im verteilten System adressierbare Einheit und Maßstab für die Granularität. Der Standardmechanismus für Prozedurfernaufruf in Java heißt RMI, entsprechende Beans werden als EJB-Container bezeichnet. Auf Webservern ablaufende Beans, sogenannte Web-Container, enthalten aufrufbare Java-Objekte, sogenannte Servlets. In Anlehnung an das HTML-Konzept von Web-Seiten hat sich der Begriff *Java server pages* (*JSP*) durchgesetzt Die aufrufenden Objekte (Applets, in „Applet-

Containern“) können bei Bedarf ebenfalls über das Web geladen werden, wodurch Klienten verteilter Java-Anwendungen ohne vorherige Installation im Rahmen des Programmstarts (via URL) geladen werden können („Java WebStart“). Message-Driven Beans zur Realisierung ereignisbasierter Kommunikation und persistente Beans sind nur zwei Beispiele weiterer Ausprägungen. Mehr darüber in Kapitel E10.

Microsoft bietet seit vielen Jahren Middleware für die (verteilte) Anwendungsentwicklung; DCOM, die verteilte Variante der Klassenbibliothek COM, wurde mit anderen Bestandteilen in .NET (sprich: „dot-net“) vereint und erweitert. Nach großen Erwartungen und ersten kritischen Reaktionen nimmt .NET inzwischen eine respektierte Position unter den Middleware-Ansätzen ein. Wie bei Java-Bytecode werden damit Programme Rechnerarchitektur- und Betriebssystem-übergreifend vereinheitlicht, darüber hinaus aber auch Programmiersprachen-übergreifend.

Das verteilte Laufzeitsystem CLR ermöglicht anders als Java verschiedene Programmiersprachen. Anders als bei Corba erfolgt deren Vereinheitlichung aber im Rahmen der Compiler und Laufzeitsysteme der einzelnen Programmiersprachen: alle Sprachen verwenden denselben Bytecode, die *Microsoft Intermediate Language* (*MSIL*). Sie sind also zur Laufzeit im wesentlichen so vereinheitlicht wie zwei Java-Bytecode-Programme. Um das zu ermöglichen, muß die Sprachdefinition einer existierenden Sprache einem einheitlichen Typsystem (*CTS*, *common type system*) angepaßt werden; müssen Typen, die nicht Bestandteil von CTS sind, entfernt werden. Dementsprechend verwendet .NET spezielle Varianten von C++, Java und anderen Sprachen; C# wurde von Beginn an auf CTS abgestimmt.

Dieses Konzept macht es möglich, die Klassenbibliothek von .NET in jeder Programmiersprache zu verwenden. Wie Java organisiert auch .NET alle Klassen in einem baumförmigen Namensraum, der von Anwendungen erweitert werden kann; der Namensraum wird weltweit eindeutig. Wie erwähnt, ist COM weiterer wichtiger Bestandteil von .NET. Seine Wirkungsweise läßt sich an der Einbettung einer editierbaren Excel-Tabelle in ein Word-Dokument erläutern: die Tabellen-Komponente wird von Word nicht „verstanden“, sie ist für Präsentation (Tabellendarstellung in einem von Word zur Verfügung gestellten Darstellungsbereich), Funktionalität (Tabellenkalkulation) und Speicherung (bedingt) selbst zuständig. Über die *QueryInterface*-Schnittstelle kann Word in seinen Menüleisten die für Excel gültigen Funktionen anbieten und ungültige Funktionen ausblenden. DCOM erweitert diese Funktionalität um Prozedurfernaufruf, COM+ ermöglicht transaktionales Verhalten. .NET kennt wie J2EE die Möglichkeit, Komponenten via Webserver anzubieten; analog zu JSP wird der Begriff ASP bzw. ASP.Net (active server pages) verwendet.

ASP.Net und JSP bilden auch wichtige Ansätze zur Erstellung von sog. Web-Services, also von Diensten, die via Web-Server als Programme statt HTML-Seiten angeboten werden. Beide basieren auf der Möglichkeit des Aufrufs per Web-Browser, d.h. per http, ohne das für Prozeduraufruf sonst typischen Binding von beliebigen Klienten aus. Somit stellen Web-Services eine weitere Vereinheitlichung, also einen quasi plattformunabhängigen, offenen Ansatz für verteilte Anwendungen dar – soweit man das Web (http) nicht als Plattform auffaßt. Präziser formuliert, führt der Web-Service-Ansatz zu offen nutzbaren und ergänzbaren Mengen von Diensten im Internet; diese können auch mittels anderer Programmiersprachen/Middleware-Ansätze bzw. Skriptsprachen erstellt werden (PHP, CGI, Python usw.). Während http und Klienten-seitige Verwendung von Browsern für interaktiven Aufruf von Web-Services geeignet sind, erlauben XML und SOAP als Schnittstellen auch die Kommunikation zwischen Web-Service und Web-Service. JSP, ASP.Net und weitere unterstützen dies inzwischen. SOAP wird ergänzt von *WSDL* (*web service description language*), der Schnittstellendefinitionssprache für Web-Services,

sowie *UDDI* (*universal description, discovery, and integration*) als Schnittstelle zur Veröffentlichung von Dienstschnittstellen sowie Suche danach; alle drei basieren auf XML. Mittels *BPEL* (*business process execution language*) können Webservices zu Geschäftsprozessen im Internet zusammengeschlossen werden; weitere Standards dienen der IT-Sicherheit, Transaktionsunterstützung u. a. Näheres wird in Kapitel E10 beschrieben.

9.7 Zusammenfassung und Weiterführendes

Das OSI-Referenzmodell wurde bereits kritisch gewürdigt, vor allem die zentrale Bedeutung des grundlegenden Modells und Probleme mit der konkreten Schichtenarchitektur. Insgesamt versteht man das Gebiet Kommunikationsprotokolle heute weit besser als das der verteilten Anwendungen, vor allem hinsichtlich der Softwaretechnik, trotz des Referenzmodells ODP.

Die Dynamik des Internets und die neuen Aufgaben durch Hochgeschwindigkeitsnetze und neue Anwendungsformen werden in den nächsten Jahren die „Protokoll-Landschaft“ und Paradigmen der verteilten Programmierung deutlich beeinflussen. Wichtige neue Anwendungsformen umfassen Multimedia (siehe Kapitel E3), mobiles Rechnen und Gruppenkommunikation. Auch die zunehmende kommerzielle Nutzung des Internets hat starke Rückwirkungen, beispielsweise auf die Bedeutung von Sicherheitsaspekten.

Allgemeine Literatur

Bengel, G.: Verteilte Systeme. 2. erw. Aufl. Vieweg Braunschweig, 2002

Borghoff, U.; Schlichter, J.: Rechnergestützte Gruppenarbeit. Berlin: Springer 1999

Comer, C., Droms, R.: Computer networks and internets. Upper Saddle River: Prentice Hall 2001

Coulouris, G.; Dollimore, J.; Kindberg, T.: Distributed systems – Concepts & design. 3rd ed., Addison Wesley 2001

Hammerschall, U.: Verteilte Systeme und Anwendungen. Pearson, München 2005

Orfali, R.; Harkey, D.; Edwards, J.: The essential distributed objects survival guide. New York: Wiley 1996

Tanenbaum, A.; v. Steen, M.: Verteilte Systeme, Grundlagen und Paradigmen. Pearson, München, 2003

Weber, M.: Verteilte Systeme. Berlin: Spektrum 1998

Spezielle Literatur

[Boger 99] Boger, M.: Java in verteilten Systemen, Heidelberg: dpunkt 1999

[Borenstein 92] Borenstein, N.; Freed, N.: RFC 1341 – MIME (Multipurpose Internet Mail Extensions): Mechanisms for specifying and describing the format of internet message bodies. Erhältlich über FTP ftp:// ftp.ira.uka.de/pub/rfc1341.gz

[Canavan 01] Canavan, J.: The fundamentals of network security. London: Artech House 2001

[Postel 82] Postel, J.: RFC 821: Simple mail transfer protocol. Erhältlich über FTP ftp://ftp.ira.uka.de/pub/rfc/rfc821.gz

[Subramanian 00] Subramanian, M.: Network management: Principles and practice, Reading: Addison Wesley 2000

[Tel 00] Tel, G.: Introduction to distributed algorithms, 2nd ed. Cambridge University Press 2000