Kapitel 2

Begriffsklärung und verwandte Arbeiten

Im folgenden Kapitel werden ausgewählte Grundkonzepte verteilter Systeme und deren Beziehung zu dienstorientierten Architekturen diskutiert, um für den weiteren Verlauf dieser Arbeit auf einem konsistenten Begriffsmodell aufbauen zu können. Zudem werden verwandte Arbeiten zu Dienstinfrastrukturen und dynamischer Ressourcenverwaltung bzgl. ihrer Eigenschaften untersucht.

2.1 Verteilte Systeme

Im Zuge der zunehmenden Anzahl von Konferenzen und Publikationen zu dienstorientierten Umgebungen sieht man sich mit einer Vielzahl von unscharfen und inkonsistent verwendeten Begriffen konfrontiert, welche eine genaue Einordnung von Konzepten erschweren. In einigen Fällen entsteht dadurch der subjektive Eindruck einer reinen Neuauflage bekannter Ideen aus dem Forschungsgebiet der verteilten Systeme [van der Aalst u. a. 2003a; Elfwing u. a. 2002; Staab u. a. 2003]. Um für diese Arbeit ein konsistentes Begriffsmodell zugrunde legen zu können, werden daher einige Begriffe aus dem Gebiet der verteilten Systeme betrachtet, um darauf aufbauend relevante Begriffe von SOA klarer definieren zu können.

Ein **verteiltes System** definiert sich als Menge von unabhängigen **Verarbeitungsknoten**, welche über ein **Kommunikationsnetzwerk** miteinander interagieren [Bal u. a. 1989]. Ein Verarbeitungsknoten besteht aus einem oder mehreren Prozessoren, einer oder mehreren Ebenen von Speicher, und einer beliebigen Menge externer Geräte. Im Unterschied zu parallelen Systemen sind die beteiligten Knoten typischerweise nicht homogen, sondern basieren auf *commercial-off-the-shelf (COTS)* Hardware mit den üblichen Betriebssystemen. Die Kommunikation über das Verbindungsnetzwerk eines verteilten Systems ist um Größenordnungen langsamer als ein lokaler Speicherzugriff.

Eine **verteilte Anwendung** definiert sich als Menge von zusammengehörigen Modulen oder Programmen, welche konkurrierend auf einem oder parallel auf multiplen Knoten im verteilten System ausgeführt werden. Eine **nebenläufige Anwendung** enthält eine Menge von Aktionen, die simultan ausgeführt werden können. Eine **parallele Anwendung** ist somit eine nebenläufige Anwendung, die für die Ausführung auf einem Parallelrechner geschrieben wurde. Eine *verteilte Anwendung* ist eine nebenläufige Implementierung, welche für die Ausführung auf autonomen durch ein Netzwerk verbundenen Prozessoren geschrieben wurde.

Die Geschichte der verteilten Systeme zeigt verschiedene Ansätze für die Granularität der Module einer verteilten Anwendung, wobei die verteilte Ausführung entweder direkt von der Sprache oder von der Middleware unterstützt wird. Typische Beispiele sind Ada (Tasks), Hermes

(Prozess) oder Java und C++ (Objekte), neuere Forschungsarbeiten setzen dabei zumeist auf das Konzept der verteilten Objekte [Briot u. a. 1998; Jul u. a. 1988; Polze 1995]. Trotzdem werden die auf den Knoten ausgeführten Teile der verteilten Anwendung in der Literatur zumeist als **Prozess** bezeichnet.

Gängige Definitionen [Andrews 1991; Soares 1992] führen den Austausch von Nachrichten über ein Netzwerk als einzige Kommunikationsform auf, während die Interaktion über gemeinsamen Speicher (*shared memory*) eher den parallelen Systemen zugeordnet wird. In der Praxis wird der Nachrichtenaustausch als Programmierparadigma aber auch für Shared-Memory-Systeme eingesetzt [Forum 1997].

Wenn man die Grundidee verteilter Systeme auf dienstorientierte Umgebungen abbildet, so erfüllt ein einzelner Dienst die gleiche Aufgabe wie ein Modul in einem verteilten System. Der Dienst stellt über seine z.B. in der *Web Service Description Language* (WSDL) beschriebene Schnittstelle bestimmte Operationen bereit, welche dann mit Hilfe von nachrichtenbasierter Kommunikation genutzt werden können. Die Programmierung von Anwendungen erfolgt typischerweise mit Fernaufrufsemantik (RPC). Einzelne Module nutzen oder bieten also aus Sichtweise der Programmiersprache Operationen mit Eingabe- und Ausgabeparametern an. Die Übertragung eines solchen entfernten Rufs realisiert dann ein nachrichtenorientiertes Protokoll, wie z.B. das *Internet Inter-ORB Protocol (IIOP)* in der *Common Object Request Broker Architecture (CORBA)* oder SOAP in Web-Service-Umgebungen.

Ähnlich wie bei Komponentensystemen findet die Entwicklung der Module (also der einzelnen Dienste) in SOA-Umgebungen nicht zum gleichen Zeitpunkt oder durch den gleichen Entwickler statt. Ein direkter Zugriff auf interne Zustandsvariablen ist in einer Dienstumgebung (wie auch bei den verteilten Objektsystemen) nicht möglich. Der Empfang einer Nachricht bewirkt die Ausführung der Dienstfunktionalität sowie das eventuelle Zurücksenden einer Antwortnachricht. Verschiedene derartige Dienste werden dann letztendlich zu einem gemeinsamen Kontrollfluss kombiniert, der die für den Nutzer sichtbare verteilte Anwendung realisiert. Somit können verteilte objektorientierte Systeme, verteilte Komponentensysteme und Dienstkompositionen (zusammen mit ihren Diensten) alle einheitlich als **verteilte Anwendung** bezeichnet werden.

2.1.1 Interprozesskommunikation

Die Unterstützung der Kommunikation zwischen verteilten Prozessen durch Middleware bezieht sich sowohl auf die Informationsübertragung selbst (Interprozesskommunikation) als auch auf den Zeitpunkt der Übertragung (Synchronisierung). Die meisten Umgebungen für verteilte Anwendungen realisieren diese Kommunikation durch den Versand und Empfang von Nachrichten. Bei der Interaktion fungiert dabei ein **Client** bzw. **Sender** als auslösender Prozess, um durch eine Nachricht im reaktiven **Server**- bzw. **Empfänger**-Prozess eine Aktivität auszulösen. **Synchrone Kommunikation** zwischen Client und Server bedingt dabei eine Blockierung des Client-Prozesses, bis der Server-Prozess eine Antwortnachricht gesendet und somit den Kontrollfluss zurückgegeben hat [Mahmoud 2004]. **Asynchrone Kommunikation** erlaubt es dem Client, in seinem Kontrollfluss nach Versand einer Nachricht fortzusetzen.

In dienstorientierten Systemen werden vor allem Web-Service-Protokolle eingesetzt. Die Verwendung von SOAP für die Kommunikation wird dabei leider zu häufig auf Interaktion im Stile eines entfernten Prozedurrufs (RPC) reduziert. Ein Endpunkt wird dabei einer entfernt aufrufbaren Menge von Prozeduren mit Eingabe- und Ausgabewerten gleichgesetzt, wie sie in objektorientierten verteilten Systemen vorkommt.

Zwar wird in WSDL eine Menge von *operation*-Definitionen zu einem Endpunkt kombiniert, allerdings können diese Operationen verschiedene Kombinationen von Ein- und Ausgabenachricht erwarten. Ein Beispiel ist der Einsatz von Web Services als nachrichtenorientierte Middleware, bei der aus der WSDL-Definition abgeleitete Nachrichten im *oneway* Modus versendet werden. Zusätzlich werden in Spezifikationen wie *WS-Notification* auch asynchrone Benachrichtigungsmodelle auf der Basis von SOAP definiert [Niblett und Graham 2005]. Daher wird SOAP in dieser Arbeit primär als Protokoll zum Versand von Nachrichten, und weniger als Kommunikationsmechanismus für ein RPC-Programmiermodell betrachtet. Entsprechend den Konzepten aus der WSDL definieren sich Operationen damit als eine Kombination aus Anfrage- und etwaiger Antwortnachricht.

Neben dem entfernten Prozedurruf und dem Nachrichtenversand sind auch speziellere oder auf anderen Abstraktionsebenen angesiedelte Interaktionsmuster in Middleware bekannt, die allerdings bisher keine weite Verbreitung in Dienstumgebungen gefunden haben. Andrews liefert eine umfassende Darstellung der verschiedenen Ansätze [Andrews 1991].

Die Kommunikation in einem verteilten System bedingt eine Adressierung des entfernten Nachrichtenempfängers. Tanenbaum et al. trennen begrifflich zwischen Adressen, Namen und Identifikatoren [Tanenbaum und van Steen 2002]. Jeder Prozess ist über seine **Adresse** nutzbar. Diese ist abhängig von Art und Ort der Ressource, auf der sich der Prozess befindet (Bsp. CORBA). Um ortsunabhängige Referenzen zu ermöglichen, führt man eine Indirektion mit von der Addresse unabhängigen **Namen** ein. Diese sind typischerweise in einem **Namenssystem** hierarisch geordnet, und werden durch einen **Namensdienst** zu Adressen aufgelöst. In Dienstumgebungen werden die Adressen von Prozessen auf Knoten als **Endpunkt** bezeichnet. Als Namensdienst kommt das *Domain Name System (DNS)* zum Einsatz.

Zusammengefasst werden für dienstorientierte verteilte Anwendungen also die gleichen Kommunikationsparadigmen wie für klassische verteilte Anwendungen verwendet. Die Betrachtung von Web Services als nachrichtenorientierte Middleware gewinnt dabei zunehmend an Bedeutung – Indikatoren dafür sind der *document-style* als bevorzugte Kodierungsmethode für SOAP [Ballinger u. a. 2007] und die Abbildung von SOAP auf zusätzliche Transportprotokolle [Werner u. a. 2005]. Hier wird das Programmiermodell des Fernaufrufs zugunsten einer nachrichtenorientierten Behandlung in der jeweiligen Implementierung zunehmend aufgegeben.

2.1.2 Kopplungsgrad

Im Zusammenhang mit dem Synchronisationsverhalten einer verteilten Anwendung werden häufig auch die Begriffe **lose Koppelung** und **enge Koppelung** verwendet. Existierende Definitionen dieser Begriffe sind zumeist vage formuliert [Coulouris u. a. 2005; Haas und Brown 2004; Kaye 2003] oder bestätigen nur die Abwesenheit einer konkreten Metrik [MacKenzie u. a. 2006]. Vinoski schlägt daher vor, die etablierten und umfassend erforschten Metriken der Softwareentwicklung als Basis einer Definition zu verwenden [Vinoski 2005]. Folgende Definition greift diesen Ansatz auf, und dient als Grundlage für spätere Argumentationen:

Definition 1. *Der **Grad der Koppelung** von Kommunikationspartnern in einer verteilten Anwendung bestimmt sich durch ihren Grad an Abhängigkeit. Eine Abhängigkeit entsteht durch auf beiden Seiten notwendiges Wissen über den jeweiligen Partner.*

Eine geringere Koppelung in einem verteilten System soll es ermöglichen, einzelne Kommunikationspartner zu modifizieren, ohne dass der jeweils andere Partner in seiner Funktionalität davon betroffen ist. Dies funktioniert nur dann, wenn der modifizierte Teil des Moduls (z.B. die verwendete Programmiersprache) nicht Teil der Abhängigkeiten ist.

Im Umkehrschluss erleichtert lose Koppelung somit den Austausch und die Modifikation einzelner Prozesse der verteilten Anwendung. Dieses Verständnis steht im Kontrast zu vielen informalen Definitionen, bei dem der Kopplungsgrad direkt an der gewählten Interaktionstechnik (synchron, asynchron, entfernter Prozedurruf, verteilter gemeinsamer Speicher) festgemacht wird. Die Auswahl einer spezifischen Implementierungstechnologie kann lediglich eine bestimmte Art der Koppelung begünstigen oder verhindern. Beispielsweise werden nachrichtenorientierte Systeme als lose gekoppelt eingestuft, eine Nutzung anwendungsspezifischer Nachrichtenformate kann aber leicht das Gegenteil bewirken.

Zur genaueren Eingrenzung des Begriffs werden die Klassen von Abhängigkeiten in Anlehnung an [Offutt u. a. 1993; Vinoski 2005] definiert:

Definition 2. *Eine Abhängigkeit zwischen Client A und Server B kann in eine oder mehrere der folgenden Klassen eingeordnet werden:*

Keine Abhängigkeit *A kommuniziert nicht mit B, und es existieren keine gemeinsamen externen Kommunikationspartner oder Datensenken.*

Aufrufabhängigkeit *A und B kommunizieren, tauschen dabei aber keine zusätzlichen Daten (Eingabe-/Ausgabeparameter) neben der eigentlichen Nachricht aus. Dies kann z.B. bei einer rein ereignisorientierten Kommunikation der Fall sein.*

Datenabhängigkeit *A und B kommunizieren, wobei Eingabeparameter und / oder Ausgabeparameter übertragen werden. Art und Menge dieser Daten haben Einfluss auf den Grad der Abhängigkeit.*

Schnittstellenabhängigkeit *A kommuniziert mit B anhand einer Schnittstellendefinition, welche Nachrichten zu Operationen kombiniert. Eine größere Anzahl an genutzten Operationen erhöht den Grad der Schnittstellenabhängigkeit zwischen den Kommunikationspartnern.*

Kontrollabhängigkeit *Aufgrund von Wissen über die Implementierung von B kann A den Kontrollfluss in B beeinflussen. Dies betrifft beispielsweise den Aufruf von Operationen in einer bestimmten Reihenfolge.*

Externe Abhängigkeit *A und B teilen sich eine gemeinsame Datensenke, wie z.B. globale Variablen, Datenbanken oder Verzeichnisse.*

Implementierungsabhängigkeit *A und B basieren in ihrer Funktionalität auf einem Verständnis der internen Funktionsweise des Kommunikationspartners.*

Die Definition eindeutiger und stabiler Schnittstellen kann den Grad der Koppelung verringern, da neben der Schnittstellenabhängigkeit nicht noch zusätzlich eine Kontrollabhängigkeit oder Implementierungsabhängigkeit vorliegt. Schlecht gestaltete Schnittstellen in Softwaresystemen verlangen dagegen vom Rufer mehr Wissen als durch die Schnittstellendefinition gegeben ist. Der reine Bezug auf Nachrichten und ihren Inhalt kann sogar eine Schnittstellenabhängigkeit vermeiden, und somit den Kopplungsgrad nochmalig verringern.

Mit der o.g. Definition für den Koppelungsbegriff ist eine qualitative Bewertung von Architekturen möglich. Diese kann im Bedarfsfall noch durch Betrachtungen der Laufzeitaspekte ergänzt werden.

2.1.3 Zustandsbehaftung

Die Zustandsbehaftung der Kommunikationspartner ist in klassischen verteilten Systemen ein selbstverständlicher Fakt. Viele Arbeiten beschäftigen sich mit der periodischen oder durch Ereignisse ausgelösten Speicherung von Zustandsdaten, um beispielsweise eine Migration von Prozessen der verteilten Anwendung zu ermöglichen [Fuggetta u. a. 1998]. Lösungen für ein solches *Checkpointing* gibt es für Betriebssysteme [Attardi u. a. 1988; Ferrari 1998; Smith und Hutchinson 1998], Middleware-Systeme [Philippsen und Zenger 1997; Tröger und Polze 2003] und als Erweiterung von Programmiersprachen [Shapiro u. a. 1989].

Im Gegensatz dazu findet bei der Verwendung von Web Services häufig eine Diskussion darüber statt, ob eine Dienstimplementierung zustandsloses oder zustandsbehaftetes Verhalten aufweisen soll. Da die Terminologie häufig uneinheitlich verwendet wird, sollen die Begriffe im Folgenden anhand dieser Definition verstanden werden:

Definition 3. *Ein **zustandsloser Nachrichtenaustausch** zwischen Sender-Prozess und Empfänger-Prozess in einer verteilten Anwendung löst eine empfängerseitige Aktivität aus, die nur durch die empfangende Nachricht bestimmt wird. Ein **zustandsbehafteter Nachrichtenaustausch** löst eine empfängerseitige Aktivität aus, welche zusätzlich von vorher empfangenden Nachrichten dieses Senders bestimmt wird. Die Menge aller zusammengehörigen Nachrichten in einem zustandsbehafteten Nachrichtenaustausch wird als **Sitzung** bezeichnet.*

Die Begriffe **zustandslose / zustandsbehaftete Kommunikation** oder **zustandslose / zustandsbehaftete Interaktion** sind analog zur obigen Definition zu sehen. Für einen zustandsbehafteten Nachrichtenaustausch muss die Bindung von Client und Server an die gleiche Sitzung für beiden Seiten jeweils erkennbar sein. Die geschieht typischerweise anhand einer beiden Seiten bekannten **Sitzungsreferenz**. Die Informationen welche den Zustand der Sitzung charakterisieren werden als **Sitzungsdaten** bezeichnet. Werden diese Daten bei der Interaktion mit mehreren Servern verwendet, spricht man von **Sitzungskontext** [Little u. a. 2006].

Sitzungsdaten werden beim Server anhand der Identität des Client oder anhand einer übermittelten Referenz identifiziert, oder direkt als Bestandteil der Nachricht mit übertragen. Die ausgelöste Aktivität ist das Berechnen und Versenden einer Antwortnachricht. Entsprechend der Definition 3 hängt die Antwortnachricht im zustandslosen Fall nur von der empfangenden Nachricht ab. Bei zustandbehafteter Interaktion nimmt der Empfänger mit jeder eingehenden Nachricht einen (potentiell neuen) Zustand ein, und verschickt somit potentiell auch eine andere Antwortnachricht.

Bei dieser Definition muss beachtet werden, dass der Begriff der Sitzung in Web-Service-Umgebungen eigentlich durch bestimmte technische Lösungen wie *HTTP Cookies* vorbelegt ist. Theoretisch könnte daher argumentiert werden, dass auch ohne eine aktive Sitzung der Empfänger auf Folgen von Nachrichten jeweils anders reagieren kann. Die Definition versucht aber, die verschiedenen praktischen Umsetzungen ausschließlich anhand der Art der ausgetauschten Nachrichten zusammenzufassen. Somit liegt im Sinne der Definition tatsächlich immer eine Sitzung vor, wenn das beschriebene Verhalten erkennbar ist, auch wenn die konkrete technische Realisierung nicht als Sitzungskonzept bezeichnet wird. Ein Beispiel wäre eine Web-Service-Implementierung, die ohne *HTTP Cookies*, *URL Rewriting* oder sonstige Mechanismen zustandsbehaftet agiert. In diesem Fall liegt also ein zustandsbehafteter Nachrichtenaustausch vor, obwohl keine Sitzungsreferenz verwendet wird.

Die Definition erklärt einen zustandsbehafteten Dienst nur anhand eingehender Nachrichten und auf Basis des aktuellen Zustandes der Sitzung. Einflüsse externer Ressourcen oder anderer nebenläufiger Aktivitäten werden also nicht als Einflussfaktoren berücksichtigt. Dies ähnelt der

Abstraktion im *Communicating Sequential Processes (CSP)*-Formalismus, und kann daher wie dort durch die Modellierung aller Einflussfaktoren als eigene Prozesse ergänzt werden [Hoare 1985].

Für den Fall einer Punkt-zu-Multipunkt Kommunikation wird in Anlehnung an [Little u. a. 2006] festgelegt:

Definition 4. *Eine **Aktivität** bezeichnet die Menge aller Nachrichten, die im Rahmen zusammengehöriger zustandsbehafteter Kommunikationsvorgänge zwischen einem Sender-Prozess und mehreren Empfänger-Prozessen übermittelt werden.*

Der Begriff der Aktivität beschreibt somit eine Sitzung mit mehr als zwei Teilnehmern, und wird vor allem im Umfeld transaktionaler Systeme verwendet.

Um die bisher besprochenen Begriffe besser greifen zu können, werden nun zwei Beispiele aus realen Kommunikationsprotokollen betrachtet:

Das HTTP-Protokoll [Khare und Lawrence 2000] regelt den Austausch von Dokumenten zwischen einem *Web Browser* und einem *Web Server*, welche gemeinsam als verteilte Anwendung mit synchroner Punkt-zu-Punkt-Kommunikation agieren. Der *Web Server* funktioniert im ursprünglichen Sinne als zustandsloser Empfänger – der Empfang einer HTTP-Anfrage führt zum Auslesen und zur Übertragung von Dateiinhalt, ohne dass eine Beeinflussung durch vorherige oder darauf folgende HTTP-Anfragen berücksichtigt werden muss.

Erweiterungen des HTTP-Protokolls wie der *Cookie*-Mechanismus realisieren eine **zustandsbehaftete Sitzung** für multiple Nachrichten [Kristol und Montulli 2000]. Die Sitzung wird vom Server initiiert, der dafür in seiner Antwortnachricht in den Nachrichtenkopf ein *Cookie*-Element einfügt, wahlweise mit den Sitzungsdaten oder einer Sitzungsreferenz. Der Client übermittelt bei jeder darauf folgenden Anfrage an den gleichen Server die *Cookie*-Daten erneut. Da die Daten immer als implizite Parameter im Nachrichtenkopf an den Server übertragen werden, sind sie kein direkter Bestandteil einer einzelnen Nutzeranfrage, sondern müssen unabhängig von dieser bei allen Anfragen berücksichtigt werden. Der *Web Server* realisiert in diesem Fall einen zustandsbehafteten Nachrichtenaustausch.

Das NFS-Protokoll [Sun Microsystems 1989] ist ein Beispiel, bei dem zwischen Client und Server ein zustandsloser Nachrichtenaustausch stattfindet. Ein NFS-Client greift auf Dateisysteme zu, indem er entfernte Prozeduraufrufe gemäß der Spezifikation von *Sun RPC* am Server durchführt. Hier findet eine synchrone Punkt-zu-Punkt Kommunikation zwischen Client und Server statt. Der Server agiert im Normalfall komplett zustandslos, was eine Realisierung von Dateisperren (*flock, fcntl*) erschwert. Als Ausgleich existiert eine Kombination von zusätzlichen Systemdiensten (*statd, lockd*), die den Dateiserver um zustandsbehaftetes Verhalten ergänzen. Der Client arbeitet also mit dem NFS-Server in einem zustandsbehafteten Nachrichtenaustausch, während der RPC-Dienst selbst einen zustandslosen Nachrichtenaustausch umsetzt.

Anhand der zwei Beispiele lässt sich leicht erkennen, dass die Einordnung konkreter verteilter Systeme selten statisch und absolut erfolgen kann. Die meisten Systeme sind in Fragen des Kopplungsgrades und der Zustandsbehaftung frei verwendbar und anpassbar. Somit muss eine Bewertung verteilter Infrastrukturen bzgl. ihrer Koppelungseigenschaften immer mit Bezug zur Anwendung erfolgen. Eine allgemeine Aussage, welche dienstorientierte Umgebungen und Architekturen per se' als lose gekoppelt bezeichnet, ist somit unrealistisch und kaum nachweisbar.

2.1.4 Konsistenzmodelle

Der konkurrierende nachrichtenbasierte Zugriff auf einen gemeinsamen Speicher ist ein klassisches Thema von verteilten Systemen, und spielt im Modell einer Dienstinfrastruktur eine wich-

tige Rolle. Die gewünschten Einschränkungen an den konkurrierenden Zugriff, welche durch die Ausführungsumgebung bzw. die Hardware sichergestellt werden sollen, werden im Allgemeinen als **Konsistenzmodell** bezeichnet. Dieses formuliert Einschränkungen an die Reihenfolge der gemeinsamen Speicherzugriffe der beteiligten Prozesse im verteilten System [Gharachorloo u. a. 1991]. Es stellt somit eine Menge von Regeln dar, die zwischen Software und dem gemeinsamen Speicher vereinbart sind und somit von beiden Seiten beachtet werden müssen.

Konsistenzmodelle unterscheiden sich bzgl. der Schärfe ihrer Anforderungen, und lassen dementsprechend mehr oder weniger Raum für Performanzoptimierungen, wie z.B. verzögerte Schreiboperationen (*pipelining*), die Zwischenspeicherung oder Vorabermittlung von gelesenen Variablen (*prefetching, caching*), oder die Veränderung der Operationsreihenfolge auf dem gemeinsamen Speicher (*out of order execution*). Typische Konsistenzmodelle in verteilten Systemen sind [Mosberger 1993]:

Strict consistency Dieses Modell [Tanenbaum 1995] stellt die stärksten Anforderungen an die Konsistenz des gemeinsamen Speichers. Jede Leseoperation einer Speichervariablen soll dabei garantiert den zuletzt geschriebenen Wert liefern. Das Modell bildet das intuive Verständnis von Uniprozessorarchitekturen ab [Adve und Gharachorloo 1995], und bedingt die Existenz synchronisierter Uhren, sowie möglichst keine Verzögerung bei der Bereitstellung geschriebener Daten an alle Prozesse. Besonders aufgrund der letzteren Anforderung ist dieses Modell in realen verteilten Systemen nicht sinnvoll implementierbar.

Sequential consistency (SC) Das SC-Modell [Lamport 1997] legt fest, dass die Lese- und Schreiboperationen eines Prozesses im Gesamtsystem in ihrer originalen Reihenfolge beobachtbar sind, und sich lediglich miteinander verschachteln. Dies bedingt dann nicht das Lesen des aktuellsten Wertes bzgl. einer globalen Zeit, sondern lediglich bzgl. einer global einheitlichen Reihenfolge, welche die Ordnung auf den einzelnen Prozessoren berücksichtigt. Das SC-Modell bietet daher kaum Möglichkeiten zur Performanzoptimierung durch *Caching* oder *Pipelining* auf den einzelnen Prozessoren.

Processor consistency (PC) Das PC-Modell ermöglicht gegenüber dem SC-Modell die Verzögerung von Schreiboperationen [Goodman 1989]. Schreibzugriffe eines einzelnen Prozessors auf einer Speicherstelle müssen aus Sichtweise des Gesamtsystems geordnet bleiben, während alle Schreibzugriffe im System konkurrierend und ohne globale Reihenfolge auftreten können. Zudem erlaubt das Modell, das verzögerte Schreibzugriffe von Lesezugriffen auf andere Speicherbereiche 'überholt' werden können, da nur noch die sequentielle Abarbeitung der Schreiboperationen pro Prozessor gefordert wird.

Weak consistency (WC) Das WC-Modell lockert die Konsistenzanforderungen dadurch, dass ein geordneter Schreibzugriff nur an spezifischen Synchronisationspunkten sichergestellt seien muss [Dubois u. a. 1986]. Dies trägt dem Umstand Rechnung, dass nicht alle Zwischenergebnisse (und deren Reihenfolge beim Schreiben) im Gesamtsystem sichtbar seien müssen. Der geschützte Bereich im Kontrollfluss des Prozesses wird dabei als **kritischer Abschnitt** (*critical section*) bezeichnet. Beim Zugriff auf eine **Synchronisationsvariable** für einen kritischen Abschnitt müssen daher alle vorangegangenen Schreiboperationen abgeschlossen sein, und alle nachfolgenden Schreib- und Lesezugriffe noch nicht gestartet sein. Der Zugriff auf die Synchronisationsvariable muss mit sequentieller Konsistenz erfolgen. Basierend auf diesem Konsistenzmodell können die Datenzugriffe innerhalb des kritischen Abschnittes wieder einem *Pipelining* unterliegen, da die Speicherung der Änderung erst mit dem nächsten Synchronisationspunkt zugesichert wird. Im Gegensatz

zu den strikteren Modellen wird hier die Konsistenz für eine Gruppe von Operationen, und nicht mehr für einzelne Operationen zugesichert.

Release consistency (RC) Das RC-Modell verfeinert das WC-Modell dahingehend, dass der Zugriff auf Synchronisationsvariablen in *acquire* und *release* aufgeteilt wird. Somit kann unterschieden werden, ob eine Anwendung ihre Schreiboperationen abschließen, oder Leseoperationen starten möchte. Ein *acquire* - Vorgang wird ausgeführt, um den exklusiven Zugriff auf einen gemeinsam genutzten Speicherbereich zu erhalten. Dafür müssen alle Schreiboperationen anderer Prozesse beendet sein. Eine *release* - Operation ist erst dann fertiggestellt, wenn alle noch ausstehenden Lese- und Schreibzugriffe beendet wurden. Offensichtlich hilft eine derartige Unterscheidung bei der Minimierung des Synchronisationsaufwandes zwischen den Prozessen. Der Zugriff auf die Synchronisationsvariablen muss nur noch mit *processor consistency* erfolgen. Im Vergleich zum WC-Modell gibt es zwischen *release* und *acquire* eine Phase von **nicht-synchronisierten Zugriff** auf den verteilten gemeinsamen Speicher.

Entry consistency (EC) Das EC-Modell [Bershad und Zekauskas 1991] stellt eine weitere Lockerung dar, hat aber dafür stärkeren Einfluss auf das Programmiermodell. In diesem Modell wird jede Variable im gemeinsamen verteilten Speicher mit einer Synchronisationsvariablen ausgestattet, die entsprechend dem RC-Modell eingesetzt wird. Mehrfache Lesezugriffe (nicht-exklusive Sperren) sind erlaubt, für einen Schreibzugriff muss eine exklusive Sperre angefordert werden. Somit können verschiedene kritische Abschnitte überlappend ausgeführt werden, solange diese nicht auf die gleichen Variablen zugreifen.

Das notwendige Konsistenzmodell für die gemeinsame Nutzung von Zustandsdaten in unserer Dienstinfrastruktur wird auf Basis dieser Kategorisierung im Abschnitt 4.1.3 ermittelt.

Nachdem nun die für diese Arbeiten relevanten Basisbegriffe verteilter Systeme diskutiert wurden, geht der folgende Abschnitt auf grundlegende Merkmale dienstorientierter Architekturen ein.

2.2 Dienstorientierte Systeme

Dienstorientierte Architekturen haben sich in den letzten Jahren zu einem großen Trend der IT-Industrie entwickelt. Das Referenzmodell der *Organization for the Advancement of Structured Information Standards* (**OASIS**) bezeichnet dienstorientierte Architektur (SOA) als Paradigma zur Organisation und Verwendung verteilt kontrollierter Fähigkeiten, welche jeweils eine spezifische Geschäftsfunktionalität realisieren [MacKenzie u. a. 2006]. [Melzer 2007] definiert eine SOA als Systemarchitektur, welche heterogene Methoden und Anwendungen als Dienst repräsentiert und damit eine technologieneutrale Nutzung und Wiederverwendung ermöglicht. Er benennt die lose Koppelung als primäres Merkmal, wobei besonders auf das Auffinden und dynamische Binden von Dienstimplementierungen zur Laufzeit verwiesen wird. Chappell definiert den Unterschied zwischen objektorientierten und dienstorientierten verteilten Systemen vor allen anhand der geänderten Programmierkonzepte [Chappell 2005].

Seit 2005 zeichnet sich in den Veröffentlichungen das einheitliche Verständnis ab, dienstorientierte Architekturen zunächst als organisatorisches Modell für Geschäftsprozesse und Firmenabläufe zu verstehen. Technische Dienste sollen in Aufbau und Granularität eher an den organisatorischen Strukturen, und weniger an technischen Gegebenheiten orientiert werden

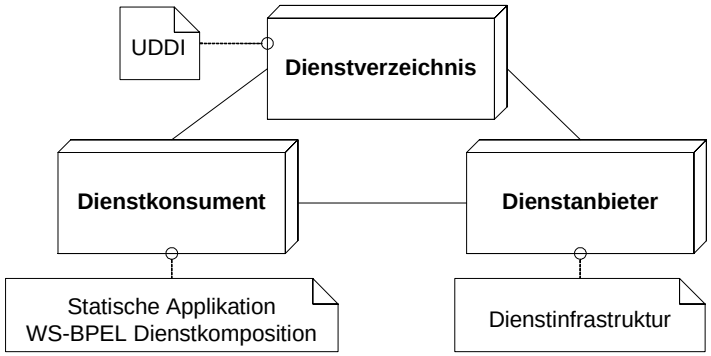

Abbildung 2.1: SOA Rollenmodell

[Krähenbühl 2006]. Im Idealfall können angebotene Geschäftsfunktionen dadurch mehrfach verwendet werden [Kuropka und Weske 2006].

Somit steht SOA in der Tradition früherer Ansätze wie *Enterprise Application Integration (EAI)* oder *Hub-and-Spoke* [Conrad u. a. 2005]. Natürlich werden für dienstorientierte Architekturen noch immer existierende Middleware-Technologien eingesetzt, wenn diese die gewünschten Diensteigenschaften wie grob granulare Schnittstellen, Trennung von Daten und Funktion oder technologieunabhängige Schnittstellenbeschreibungen unterstützen.

Das Rollenmodell in SOA wird häufig als Dreiecksbeziehung wie in Abbildung 2.1 dargestellt [Booth u. a. 2004]. Dienstanbieter veröffentlichen die syntaktische und semantische Beschreibung ihrer Dienste in einem standardisierten Format im Dienstverzeichnis. Dienstkonsumenten suchen die von ihnen benötigten Dienste über standardisierte Verzeichnisschnittstellen und erhalten als Ergebnis eine Referenz auf den Dienst beim jeweiligen Anbieter. Dieser Verweis kann die notwendigen Schnittstellenbeschreibungen bereits enthalten oder ermöglicht die Abfrage dieser Metadaten beim Dienstanbieter. Mit Hilfe der empfangenden Informationen kann der Dienstkonsument nun die Dienstfunktionen nutzen.

Auf der Grundlage dieses Konzepts definiert die SOA-Arbeitsgruppe im OASIS Standardisierungsgremium ein Referenzmodell für dienstorientierte Architekturen [MacKenzie u. a. 2006]. Dieses Modell bemüht sich um eine technologieneutrale, abstrakte Beschreibung der Eigenschaften dienstorientierter Softwarearchitekturen. Auf Grundlage einer minimalen Menge von gemeinsamen Konzepten, Axiomen und Relationen sollen dann konkrete Referenzarchitekturen, wie z.B. die ASG-Architektur, für SOA-Systeme abgeleitet werden können.

Ein Dienst im SOA-Referenzmodell soll verschiedene Informationen bieten, um Dienstkonsumenten die Nutzung seiner Fähigkeiten zu ermöglichen. Analog zu den Komponentensystemen gehört dazu eine Dienstbeschreibung, welches das Verhalten der Dienstimplementierung für den Konsumenten repräsentiert. Weiterhin sollten die realen Effekte anhand der Dienstbeschreibung ermittelbar sein, was über rein funktional orientierte Schnittstellenbeschreibungen hinausgeht.

Die Dienstbeschreibung ermöglicht dann dem Konsumenten nicht nur die technische Nutzung eines spezifischen Dienstes, sondern bietet auch Informationen zur Auswahl aus einer Menge von angebotenen Diensten. Dieser **Ausführungskontext** definiert die Menge der Infrastrukturelemente, Nutzungsbedingungen und Vereinbarungen, welche eine Dienstinteraktion später ausmachen. Somit ist schon bei der Dienstauswahl eine Berücksichtigung von Laufzeitaspekten möglich. Dieses Merkmal ist für diese Arbeit von Bedeutung, da die Überwachung der Dienstausführung offensichtlich im Referenzmodell als wesentlicher Faktor identifiziert wird.

Das Referenzmodell trifft keine direkten Aussagen zur Zustandsbehaftung von Diensten. Einige Teile der Spezifikation legen aber eine Exponierung von Zustandsänderungen an der