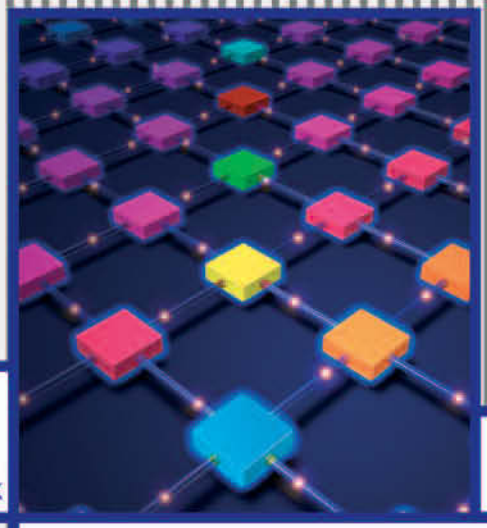
WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE



it
informatik

Andrew S. Tanenbaum
Maarten van Steen

Verteilte Systeme

Prinzipien und Paradigmen

2., aktualisierte Auflage

Prentice Hall

PEARSON
Studium

Prozesse

3

3.1 Threads 93
3.1.1 Einführung in Threads 93
3.1.2 Threads in verteilten Systemen 98

3.2 Virtualisierung 101
3.2.1 Die Rolle der Virtualisierung in verteilten Systemen . . 102
3.2.2 Architekturen virtueller Maschinen 103

3.3 Clients 104
3.3.1 Vernetzte Benutzerschnittstellen 105
3.3.2 Clientseitige Software für die Verteilungstransparenz . . 109

3.4 Server 110
3.4.1 Allgemeine Entwurfsfragen 110
3.4.2 Servercluster 114
3.4.3 Servercluster verwalten 119

3.5 Codemigration 126
3.5.1 Ansätze zur Codemigration 126
3.5.2 Migration und lokale Ressourcen 130
3.5.3 Migration in heterogenen Systemen 132

Zusammenfassung 135

Aufgaben 136

ÜBERBLICK

MOTIVATION

» In diesem Kapitel werfen wir einen genaueren Blick auf die verschiedenen Typen von Prozessen, die eine entscheidende Rolle in verteilten Systemen spielen. Das Konzept der Prozesse hat seinen Ursprung in Betriebssystemen, wo sie allgemein als Programme in der Ausführung definiert werden. Aus der Sicht eines Betriebssystems sind Handhabung und Zeitplanung von Prozessen die wahrscheinlich wichtigsten Aufgaben. In verteilten Systemen jedoch sind andere Aspekte mindestens genauso wichtig, wenn nicht gar wichtiger.

Um Client-Server-Systeme effizient aufzubauen, ist es z.B. häufig sehr bequem, Multithread-Techniken einzusetzen. Wie wir im ersten Abschnitt sehen werden, besteht ein Hauptbeitrag von Threads in verteilten Systemen darin, die Konstruktion von Clients und Servern auf eine Weise zu ermöglichen, dass Kommunikation und lokale Verarbeitung überlappen, was die Leistung deutlich steigert.

In den letzten Jahren ist das Prinzip der Virtualisierung immer beliebter geworden. Dadurch ist es möglich, dass eine Anwendung – und möglicherweise auch ihre gesamte Umgebung einschließlich des Betriebssystems – nebenläufig, also quasi zur gleichen Zeit, mit anderen Anwendungen arbeitet, und zwar unabhängig von der zugrunde liegenden Hardware und Plattform, was zu einem hohen Grad an Portabilität führt. Darüber hinaus hilft die Virtualisierung dabei, Ausfälle aufgrund von Fehlern oder Sicherheitsproblemen zu isolieren. Es handelt sich um ein wichtiges Konzept für verteilte Systeme, weshalb wir in einem eigenen Abschnitt wieder darauf zurückkommen.

Wie in *Kapitel 2* bereits erörtert, sind Client-Server-Organisationen in verteilten Systemen wichtig. In diesem Kapitel schauen wir uns den typischen Aufbau sowohl von Servern als auch Clients an. Außerdem kümmern wir uns um allgemeine Entwurfsfragen für Server.

Ein wichtiger Gesichtspunkt, vor allem in weiträumig verteilten Systemen, ist das Verschieben von Prozessen von einem Computer zum anderen. Die Prozessmigration, oder genauer gesagt die Codemigration, kann zur Skalierbarkeit beitragen, aber auch dabei helfen, Clients und Server dynamisch zu konfigurieren. Was Codemigration im Einzelnen bedeutet und welche Auswirkungen sie hat, besprechen wir ebenfalls in diesem Kapitel. «

3.1 Threads

Prozesse bilden zwar die Grundbausteine von verteilten Systemen, doch erweist es sich in der Praxis, dass der vom zugrunde liegenden Betriebssystem vorgesehene Detaillierungsgrad der Prozesse nicht ausreicht. Stattdessen zeigt es sich, dass eine feine Detaillierung in Form mehrerer Steuerthreads pro Prozess den Aufbau von verteilten Anwendungen erleichtert und zu einer höheren Leistung führt. In diesem Abschnitt sehen wir uns die Rolle von Threads in verteilten Systemen genauer an und erklären, warum sie so wichtig sind. Mehr über Threads und ihre Anwendung zum Aufbau von Anwendungen finden Sie bei Lewis und Berg (1998) sowie Stevens (1999).

3.1.1 Einführung in Threads

Um die Rolle von Threads in verteilten Systemen zu verstehen, müssen wir wissen, was ein Prozess ist und wie Prozesse und Threads miteinander in Verbindung stehen. Wenn ein Betriebssystem ein Programm ausführt, erstellt es eine Reihe von virtuellen Prozessoren – jeweils einen für jedes Programm. Um sich über diese virtuellen Prozessoren auf dem Laufenden zu halten, verwendet das Betriebssystem eine *Prozesstabelle* mit Einträgen zum Speichern von CPU-Registerwerten, Speicherabbildern, offenen Dateien, Informationen über Benutzerkonten, Zugriffsrechten usw. Ein Prozess wird häufig als ein Programm in der Ausführung definiert, also als ein Programm, das gerade in einem der virtuellen Prozessoren des Betriebssystem läuft. Das Betriebssystem achtet sehr sorgfältig darauf, dass unabhängige Prozesse das korrekte Verhalten von anderen nicht böswillig oder unbeabsichtigt beeinträchtigen. Mit anderen Worten, die Tatsache, dass mehrere Prozesse gleichzeitig dieselbe CPU und andere Hardwareressourcen nutzen, wird transparent gestaltet. Um diese Trennung zu erreichen, braucht das Betriebssystem gewöhnlich Unterstützung durch die Hardware.

Diese Form der Transparenz fordert einen vergleichsweise hohen Preis. So muss das Betriebssystem z.B. jedes Mal, wenn ein Prozess erstellt wird, einen vollständig unabhängigen Adressraum anlegen. Zuweisung kann bedeuten, Speichersegmente z.B. dadurch zu initialisieren, dass ein Datensegment auf null gesetzt, das zugehörige Programm in ein Textsegment kopiert und ein Stack oder Stapel für temporäre Daten angelegt wird. Auch das Umschalten der CPU von einem Prozess zu einem anderen kann relativ teuer werden. Das Betriebssystem muss nicht nur den CPU-Kontext sichern (der aus Registerwerten, Programmzähler, Stack-Zeiger (Stapelzeiger) usw. besteht), sondern auch die Register der Speicherverwaltungseinheit (Memory Management Unit, MMU) ändern und Adressübersetzungscaches wie den im TLB (Translation Lookaside Buffer) ungültig machen. Falls das Betriebssystem mehr Prozesse unterstützt, als es gleichzeitig im Hauptspeicher halten kann, muss es darüber hinaus Prozesse vom Arbeitsspeicher auf die Festplatte auslagern, bevor der eigentliche Wechsel stattfinden kann.

Wie ein Prozess führt auch ein Thread seinen eigenen Code unabhängig von anderen Threads aus. Im Gegensatz zu Prozessen wird jedoch kein Versuch unternommen, einen hohen Grad an Nebenläufigkeitstransparenz zu erreichen, falls dies zu einer Verschlechterung der Leistung führen würde. Ein Thread-System unterhält daher nur das Minimum an Informationen, damit eine CPU von mehreren Threads verwendet werden kann. Vor allem besteht der Thread-Kontext meistens aus nicht mehr als dem CPU-Kontext und einigen anderen Informationen zur Thread-Verwaltung. Ein Thread-

System kann z.B. die Information nachverfolgen, dass ein Thread zurzeit von einer Mutex-Variable blockiert wird, damit er nicht zur Ausführung ausgewählt wird. Informationen, die nicht unbedingt zur Handhabung von mehreren Threads benötigt werden, werden im Allgemeinen ignoriert. Aus diesem Grund bleibt der Schutz der Daten vor unangemessenem Zugriff durch Threads innerhalb eines Prozesses ganz den Entwicklern der Anwendung überlassen.

Dieser Ansatz hat zwei wichtige Konsequenzen. Erstens muss die Leistung einer Multithread-Anwendung nicht schlechter sein als die eines Gegenstücks mit nur einem Thread. Tatsächlich führt Multithreading in vielen Fällen zu einem Leistungsgewinn. Zweitens erfordert die Entwicklung von Multithread-Anwendungen zusätzliche geistige Anstrengungen, da Threads nicht wie Prozesse automatisch gegeneinander geschützt sind. Wie immer sind ein sauberer Entwurf und Einfachheit sehr hilfreich. Die gegenwärtigen Praktiken zeigen jedoch leider nicht, dass dieses Prinzip beachtet wird.

Verwendung von Threads in nicht verteilten Systemen

Bevor wir die Rolle von Threads in verteilten Systemen besprechen, schauen wir uns zuerst ihre Verwendung in herkömmlichen, nicht verteilten Systemen an. Verschiedene Vorteile von Multithread-Prozessen steigerten die Beliebtheit von Thread-Systemen.

Der wichtigste Vorteil ergibt sich daraus, dass ein Prozess mit nur einem Thread vollständig blockiert wird, sobald ein blockierender Systemaufruf ausgeführt wird. Stellen Sie sich zur Veranschaulichung eine Anwendung wie ein Tabellenkalkulationsprogramm vor und nehmen Sie an, dass ein Benutzer fortlaufend interaktiv Werte ändern möchte. Eine wichtige Eigenschaft eines solchen Programms besteht darin, dass es die funktionalen Abhängigkeiten zwischen verschiedenen Zellen und oft auch zwischen verschiedenen Arbeitsblättern beibehält. Sobald eine Zelle geändert wird, werden daher alle abhängigen Zellen automatisch aktualisiert. Wenn ein Benutzer den Wert in einer einzigen Zelle ändert, kann dies eine lange Reihe von Berechnungen auslösen. Falls es nur einen Steuerthread gibt, kann die Berechnung nicht weitergehen, während das Programm auf Eingaben wartet. Die einfache Lösung besteht darin, mindestens zwei Steuerthreads zu verwenden, einen für die Interaktion mit dem Benutzer und einen zum Aktualisieren des Arbeitsblattes. In der Zwischenzeit könnte ein dritter Thread verwendet werden, um das Arbeitsblatt auf Festplatte zu sichern, während die beiden anderen Threads jeweils ihre Aufgaben erfüllen.

Ein weiterer Vorteil von Multithreading besteht darin, dass es die Nutzung der Parallelität bei der Ausführung auf einem Mehrprozessorsystem ermöglicht. In diesem Fall wird jeder Thread einer anderen CPU zugewiesen, während gemeinsame Daten im gemeinsamen Hauptspeicher abgelegt werden. Richtig entworfen kann eine Parallelverarbeitung dieser Art transparent erscheinen: Der Prozess läuft genauso gut auf einem Einprozessorsystem, allerdings langsam. Multithreading für die Parallelverarbeitung wird aufgrund der Verfügbarkeit relativ günstiger Mehrprozessor-Arbeitsstationen immer wichtiger. Solche Computersysteme werden gewöhnlich für die Ausführung von Servern in Client-Server-Anwendungen eingesetzt.

Multithreading ist auch im Zusammenhang mit umfangreichen Anwendungen nützlich. Solche Anwendungen werden häufig als Sammlung von kooperierenden Programmen entwickelt, die jeweils von einem anderen Prozess ausgeführt werden. Dieser Ansatz ist typisch für UNIX-Umgebungen. Die Kooperation zwischen den Programmen wird mithilfe von IPC-Mechanismen (Interprocess Communication) implementiert. Für

UNIX-Systeme gehören gewöhnlich (benannte) Pipes, Nachrichtenwarteschlangen und gemeinsame Speichersegmente zu diesen Mechanismen (siehe auch Stevens und Rago [2005]). Der Hauptnachteil aller IPC-Mechanismen besteht darin, dass die Kommunikation häufig extensive Kontextwechsel erfordert. ▶Abbildung 3.1 zeigt solche Kontextwechsel an drei verschiedenen Stellen.

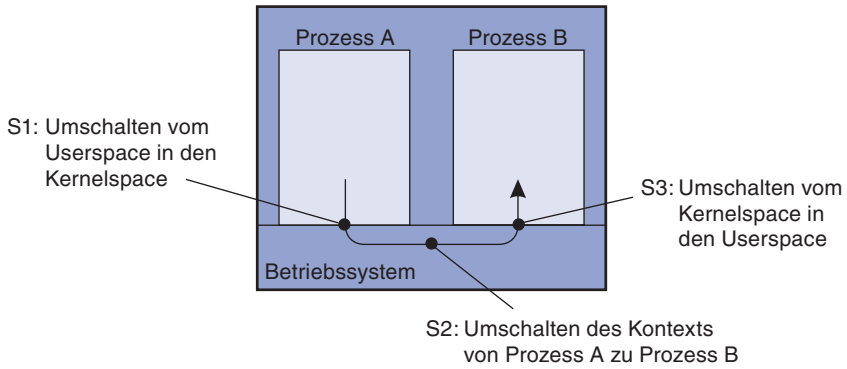


Abbildung 3.1: Kontextwechsel aufgrund von IPC

Da IPC ein Einschreiten des Kernels (oder Kerns) erfordert, muss ein Prozess im Allgemeinen zuerst vom Benutzer- oder User- in den Kernelmodus umschalten, was in ▶Abbildung 3.1 als *S1* gezeigt wird. Dies erfordert eine Änderung der Speicher-Map in der MMU und die Leerung des TLB. Innerhalb des Kernels findet ein Prozesskontextwechsel statt (*S2* in der Abbildung), nach dem die andere Partei durch Zurückschalten vom Kernel- in den Benutzermodus aktiviert werden kann (*S3*). Der letzte Wechsel erfordert wiederum die Änderung der MMU-Map und die Entleerung des TLB.

Eine Anwendung kann auch so aufgebaut sein, dass sie keine Prozesse verwendet, sondern die verschiedenen Teile in unterschiedlichen Threads ausführt. Die Kommunikation zwischen diesen Teilen wird vollständig mithilfe von gemeinsamen Daten erledigt. Thread-Wechsel können manchmal komplett im Userspace erfolgen, während in anderen Implementierungen der Kernel die Threads kennt und die Zeitplanung für sie vornimmt. Dadurch kann sich eine drastische Leistungssteigerung ergeben.

Schließlich gibt es auch noch einen rein entwicklungstechnischen Grund für die Verwendung von Threads: Viele Anwendungen lassen sich einfacher als Sammlung kooperierender Threads erstellen. Denken Sie dabei an Anwendungen, die mehrere (mehr oder weniger unabhängige) Aufgaben ausführen müssen. Im Fall einer Textverarbeitung können z.B. verschiedene Threads verwendet werden, um die Benutzereingaben, die Rechtschreib- und Grammatikprüfung, das Dokumentenlayout, die Indexerstellung usw. handzuhaben.

Implementierung von Threads

Threads werden häufig in Form eines Thread-Pakets bereitgestellt. Ein solches Paket enthält Operationen, um Threads zu erstellen und zu zerstören, sowie Operationen für Synchronisierungsvariablen wie Mutexe und Bedingungsvariablen. Es gibt zwei allgemeine Ansätze zur Implementierung eines Thread-Pakets. Der erste Ansatz besteht darin, eine Thread-Bibliothek aufzubauen, die dann komplett im Benutzermodus ausgeführt wird. Beim zweiten Ansatz hat der Kernel Kenntnisse von den Threads und plant sie.

Eine Thread-Bibliothek auf Benutzerebene hat verschiedene Vorteile. Zunächst einmal lassen sich Threads kostengünstig erstellen und zerstören[1]. Da die gesamte Thread-Verwaltung im Adressraum des Benutzers verbleibt, wird der Preis für das Erstellen eines Threads hauptsächlich durch die Kosten für die Zuweisung von Speicher für den Aufbau eines Threadstacks bestimmt. Ebenso erfordert das Zerstören eines Threads hauptsächlich, nicht mehr benötigten Speicher vom Stack freizugeben. Beide Operationen sind kostengünstig.

Ein zweiter Vorteil von Threads auf Benutzerebene besteht darin, dass der Wechsel des Thread-Kontextes meistens mit wenigen Anweisungen erledigt werden kann. Im Grunde müssen nur die Werte der CPU-Register gespeichert und anschließend mit den zuvor gespeicherten Werten des Threads geladen werden, zu dem der Wechsel stattfindet. Es ist nicht nötig, Speicherabbilder zu ändern, den TLB zu löschen, eine Berechnung der Prozessorzeit durchzuführen usw. Der Wechsel des Thread-Kontextes erfolgt, wenn zwei Threads synchronisiert werden müssen, z.B. wenn gemeinsame Daten eingegeben werden.

Ein großer Nachteil von Threads auf Benutzerebene besteht jedoch darin, dass der Aufruf eines blockierenden Systemaufrufes unmittelbar den gesamten Prozess lahm legt, zu dem der Thread gehört, und damit auch alle anderen Threads im Prozess. Wie wir bereits erklärt haben, sind Threads besonders nützlich, um große Anwendungen in Teile zu gliedern, die logisch zur selben Zeit ausgeführt werden können. Dabei sollte eine Blockierung der Ein-/Ausgabe andere Teile nicht an der Ausführung hindern. Für solche Anwendungen sind Threads auf Benutzerebene nicht geeignet.

Diese Probleme lassen sich meistens durch Threads im Betriebssystemkernel umgehen. Leider ist dafür ein hoher Preis zu zahlen: Jede Thread-Operation (Erstellen, Zerstören, Synchronisieren usw.) muss vom Kernel ausgeführt werden, was einen Systemaufruf erfordert. Der Wechsel des Thread-Kontextes kann dann genauso teuer werden wie ein Wechsel des Prozesskontextes. Dadurch werden die meisten Leistungsvorteile der Verwendung von Threads anstelle von Prozessen zunichte gemacht.

Eine Lösung besteht in einer Mischform von Benutzer- und Kernelthreads, die allgemein als Lightweight-Prozess *(LWP)* bezeichnet wird. Ein LWP läuft im Kontext eines einzelnen (normalen) Prozesses, wobei es mehrere LWPs pro Prozess geben kann. Neben LWPs kann ein System auch ein Paket von Benutzerthreads bereitstellen, das Anwendungen die üblichen Operationen zum Erstellen und Zerstören von Threads bietet. Daneben enthält dieses Paket auch Möglichkeiten zur Thread-Synchronisierung, z.B. Mutexe und Bedingungsvariablen. Wichtig dabei ist, dass das Thread-Paket komplett im Userspace implementiert ist. Mit anderen Worten, alle Thread-Operationen werden ohne Eingreifen des Kernels ausgeführt.

Das Thread-Paket kann dann von mehreren LWPs gemeinsam verwendet werden, wie ▶Abbildung 3.2 zeigt. Das bedeutet, dass jeder LWP in seinem eigenen Thread (auf Benutzerebene) läuft. Multithread-Anwendungen sind so aufgebaut, dass sie Threads erstellen und die einzelnen Threads anschließend einem LWP zuordnen. Diese Zuweisung erfolgt normalerweise implizit und wird vor dem Programmierer verborgen.

1 Die Kosten beziehen sich auf die Kosten für die Belegung von Ressourcen eines Computers (Speicherverbrauch, Rechenzeit). Mehr Informationen dazu in Tanenbaum, *Moderne Betriebssysteme.,* München: Pearson Studium, 2002

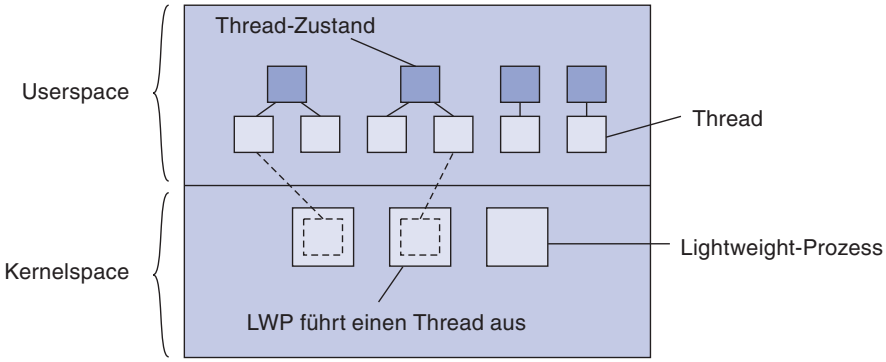


Abbildung 3.2: Kombination aus Lightweight-Kernelprozessen und Benutzerthreads

Die Kombination von (Benutzer-)Threads und LWPs funktioniert wie folgt. Das Thread-Paket verfügt über eine einzelne Routine, um den nächsten Thread einzuplanen. Wird ein LWP angelegt (was durch einen Systemaufruf geschieht), so erhält er seinen eigenen Stack und die Anweisung, die Scheduler-Routine (Planer-Routine) zu durchlaufen, um nach einem auszuführenden Thread zu suchen. Wenn es mehrere LWPs gibt, führen alle den Scheduler aus. Die Thread-Tabelle, in der Informationen über den aktuellen Thread-Satz gepflegt werden, wird auf diese Weise gemeinsam von den LWPs verwendet. Um gegenseitig ausschließenden Zugriff zu garantieren, wird diese Tabelle durch Mutexe geschützt, die komplett im Userspace implementiert sind. Mit anderen Worten, die Synchronisierung der LWPs erfordert keine Unterstützung durch den Kernel.

Wenn ein LWP einen lauffähigen Thread findet, schaltet er den Kontext zu diesem Thread um. In der Zwischenzeit können aber andere LWPs ebenfalls nach lauffähigen Threads Ausschau halten. Wird ein Thread aufgrund eines Mutex oder einer Bedingungsvariable blockiert, führt er die nötigen Verwaltungsaufgaben durch und ruft schließlich die Scheduler-Routine auf. Sobald ein weiterer lauffähiger Thread gefunden wird, erfolgt ein Kontextwechsel zu diesem Thread. Das Schöne daran ist, dass der LWP, der den Thread ausführt, nicht informiert werden muss: Der Kontextwechsel wird vollständig im Userspace implementiert und erscheint dem LWP gegenüber als normaler Programmcode.

Lassen Sie uns als Nächstes betrachten, was geschieht, wenn ein Thread einen blockierenden Systemausruf durchführt. In diesem Fall wechselt die Ausführung vom Benutzer- in den Kernelmodus, bleibt aber immer noch im Kontext des aktuellen LWP. An dem Punkt, an dem der LWP nicht mehr weiterarbeiten kann, kann das Betriebssystem entscheiden, einen Kontextwechsel zu einem anderen LWP durchzuführen, was wiederum einen Kontextwechsel zurück in den Benutzermodus mit sich bringt. Der ausgewählte LWP fährt einfach an der Stelle fort, an der er zuvor aufgehört hat.

Die Verwendung von LWPs zusammen mit einem Thread-Paket auf Benutzerebene weist verschiedene Vorteile auf.

1. Ist das Erstellen, Zerstören und Synchronisieren von Threads relativ billig und erfordert kein Eingreifen des Kernels.

2. Kann ein blockierender Systemaufruf nicht den gesamten Prozess lahm legen, sofern dieser über ausreichend LWPs verfügt.

3. Muss eine Anwendung nichts über die LWPs wissen, sondern sieht lediglich die Benutzerthreads.

4. Können LWPs in Mehrprozessorumgebungen sehr einfach eingesetzt werden, wobei die einzelnen LWPs jeweils auf verschiedenen CPUs ausgeführt werden.

Diese Verteilung kann vollständig vor der Anwendung verborgen werden. Der einzige Nachteil der Kombination von Lightweight-Prozessen mit Benutzerthreads besteht darin, dass die LWPs erstellt und zerstört werden müssen, was genauso teuer ist wie bei Kernelthreads. Dies ist jedoch nur gelegentlich erforderlich und wird häufig vollständig vom Betriebssystem gesteuert.

Ein alternativer, aber ähnlicher Ansatz besteht darin, Scheduler-Aktivierungen zu nutzen (Anderson *et al.*, 1991). Der wichtigste Unterschied zwischen ihnen und LWPs besteht darin, dass der Kernel bei der Blockierung eines Threads einen *Aufruf zur nächsten Schicht* für das Thread-Paket und damit letztendlich die Scheduler-Routine durchführt, um den nächsten lauffähigen Thread auszuwählen. Dieser Vorgang wird wiederholt, wenn die Blockierung aufgelöst wird. Der Vorteil dieses Ansatzes besteht darin, dass die Verwaltung von LWPs durch den Kernel eingespart wird. Die Verwendung solcher Aufrufe gilt jedoch als unelegant, da sie die Struktur mehrschichtiger Systeme verletzen, in der Aufrufe nur zur nächsttieferen Schicht zugelassen sind.

3.1.2 Threads in verteilten Systemen

Eine wichtige Eigenschaft von Threads ist, dass sie eine bequeme Maßnahme darstellen, um blockierende Systemaufrufe zuzulassen, ohne den gesamten Prozess lahm zu legen, in dem der Thread läuft. Dadurch werden Threads vor allem für die Verwendung in verteilten Systemen sehr attraktiv, denn damit wird es einfacher, Kommunikation durch die Pflege mehrerer logischer Verbindungen zur selben Zeit auszudrücken. Wir veranschaulichen dies, indem wir uns Multithread-Clients und -Server genauer anschauen.

Multithread-Clients

Für einen hohen Grad an Verteilungstransparenz müssen verteilte Systeme in Weitbereichsnetzwerken (Wide Area Networks) unter Umständen lange Weiterleitungszeiten von Nachrichten zwischen den Prozessen verbergen. Die Umlaufverzögerung in einem WAN kann leicht in die Größenordnungen von Hunderten von Millisekunden oder manchmal sogar Sekunden gelangen.

Der übliche Weg, um Kommunikationsverzögerungen zu verbergen, besteht darin, sofort nach dem Auslösen der Kommunikation mit etwas anderem weiterzumachen. Ein typisches Beispiel dafür sind Webbrowser. In vielen Fällen besteht ein Webdokument aus einer HTML-Datei mit reinem Text sowie einer Reihe von Bildern, Symbolen usw. Um die einzelnen Elemente eines Webdokuments abzurufen, muss der Browser eine TCP/IP-Verbindung aufbauen, die eingehenden Daten lesen und an die Anzeigekomponente übergeben. Der Aufbau der Verbindung und das Lesen eingehender Daten sind an sich blockierende Operationen. Bei langsamer Kommunikation besteht noch der zusätzliche Nachteil, dass die einzelnen Operationen lange Zeit brauchen, bis sie abgeschlossen sind.

Ein Webbrowser beginnt oft damit, dass er die HTML-Seite abruft und anschließend anzeigt. Um die Kommunikationsverzögerungen so weit wie möglich zu verbergen, beginnen einige Browser damit, Daten schon anzuzeigen, während sie noch empfan-

gen werden. Während der Text dem Benutzer zur Verfügung gestellt wird, einschließlich der Möglichkeiten für den Bildlauf u.Ä., fährt der Browser fort, andere Dateien abzurufen, die die Seite ausmachen, z.B. Bilder. Letztere werden angezeigt, sobald sie angekommen sind. Der Benutzer muss daher nicht darauf warten, bis alle Komponenten der gesamten Seite abgerufen sind, bevor ihm die Seite zur Verfügung steht.

Der Webbrowser führt letztlich eine Reihe von Aufgaben gleichzeitig aus. Es zeigt sich, dass alles einfacher wird, wenn man den Browser als Multithread-Client entwickelt. Sobald die HTML-Hauptseite abgerufen ist, können verschiedene Threads aktiviert werden, um sich um den Abruf der anderen Teile zu kümmern. Jeder Thread baut eine eigene Verbindung zum Server auf und ruft die Daten ab. Der Verbindungsaufbau und das Lesen der Daten vom Server lassen sich mithilfe der standardmäßigen (blockierenden) Systemaufrufe programmieren, wobei davon ausgegangen werden kann, dass ein blockierender Aufruf nicht den gesamten Prozess zum Erliegen bringt. Wie Stevens (1998) zeigt, ist der Code in den einzelnen Threads derselbe und vor allem einfach. In der Zwischenzeit bemerkt der Benutzer zwar Verzögerungen bei der Anzeige der Bilder und ähnlicher Elemente, er kann das Dokument aber abgesehen davon durchstöbern.

Es gibt einen weiteren wichtigen Vorteil bei der Verwendung eines Multithread-Webbrowsers, in dem verschiedene Verbindungen gleichzeitig geöffnet sind. Im vorstehenden Beispiel wurden diese Verbindungen alle zum selben Server hergestellt. Wenn dieser Server stark ausgelastet oder einfach sehr langsam ist, führt das gegenüber dem sukzessiven Abrufen der Dateien zu keinen echten Leistungsvorteilen.

In vielen Fällen sind Webserver jedoch auf mehrere Computer repliziert, wobei jeder Server genau dieselbe Menge von Webdokumenten bereitstellt. Die replizierten Server befinden sich auf derselben Site und sind unter demselben Namen bekannt. Wenn eine Anforderung nach einer Webseite eingeht, wird sie an einen der Server weitergeleitet, wobei häufig ein Umlaufverfahren (Round Robin) oder eine andere Lastenausgleichstechnik zum Einsatz kommt (Katz *et al.*, 1994). Bei der Verwendung eines Multithread-Clients können Verbindungen zu verschiedenen Replikaten aufgebaut werden, um Daten parallel zu übertragen. Dadurch lässt sich das gesamte Webdokument in sehr viel kürzerer Zeit anzeigen als bei einem nicht replizierten Server. Dieser Ansatz ist nur möglich, wenn der Client mit parallelen Streams von eingehenden Daten umgehen kann. Für diesen Zweck sind Threads ideal geeignet.

Multithread-Server

Auch wenn Multithread-Clients wichtige Vorteile bieten, wie wir gesehen haben, wird Multithreading in verteilten Systemen jedoch hauptsächlich auf der Server-Seite eingesetzt. Die Praxis zeigt, dass Multithreading nicht nur den Server-Code deutlich vereinfacht, sondern auch die Entwicklung von Servern, die die Parallelverarbeitung ausnutzen, um eine hohe Leistung zu erzielen, selbst auf Einzelprozessorsystemen. Nachdem Mehrprozessorcomputer inzwischen aber als Allzweck-Arbeitsstationen erhältlich sind, erweist sich Multithreading für die Parallelverarbeitung als umso nützlicher.

Um die Vorteile von Threads für das Schreiben von Server-Code zu verstehen, stellen Sie sich den Aufbau eines Dateiservers vor, der gelegentlich eine Blockierung durchführen muss, wenn er auf die Festplatte wartet. Dieser Server wartet normalerweise auf eine eingehende Anforderung für eine Dateioperation, führt die Anforderung aus und sendet die Antwort dann zurück. Ein möglicher und besonders beliebter Aufbau ist in ▶Abbildung 3.3 zu sehen. Hier liest ein Thread, der sogenannte Dispat-

cher, die eingehenden Anforderungen für eine Dateioperation. Die Anforderungen werden von Clients an einen Standardendpunkt für diesen Server gesendet. Nach der Untersuchung der Anforderung wählt der Server einen *Worker-Thread* im Leerlauf aus und übergibt ihm die Anforderung.

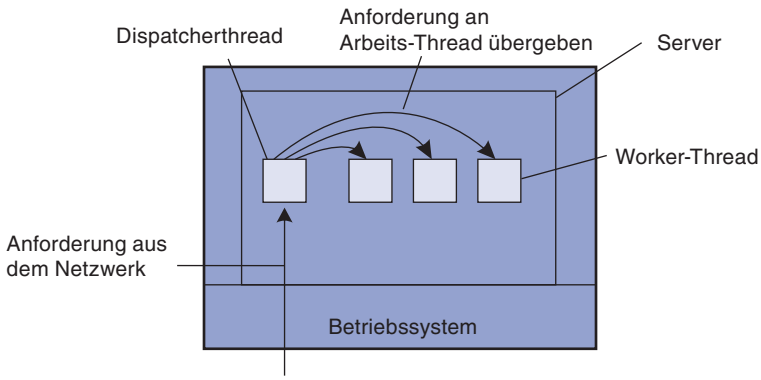


Abbildung 3.3: Ein Multithread-Server nach dem Dispatcher/Worker-Modell

Der Worker-Thread fährt fort, indem er einen blockierenden Lesevorgang im *lokalen* Dateisystem durchführt. Das kann dazu führen, dass der Thread ausgesetzt wird, bis die Daten von der Festplatte abgeholt sind. In diesem Fall wird ein anderer Thread zur Ausführung ausgewählt. So kann z.B. der Dispatcher ausgewählt werden, um weitere Aufgaben einzuholen. Alternativ lässt sich ein anderer Worker-Thread ausführen, der jetzt bereit ist.

Stellen Sie sich nun vor, wie der Dateiserver ohne Threads geschrieben worden wäre. Eine Möglichkeit besteht darin, ihn als einzelnen Thread auszuführen. Die Hauptschleife des Dateiservers erhält eine Anforderung. Diese wird untersucht und bis zum Abschluss durchgeführt, bevor die nächste angenommen wird. Infolgedessen können Anforderungen von anderen Clients nicht bearbeitet werden. Außerdem ist die CPU im Leerlauf, während der Dateiserver auf die Festplatte wartet, sofern der Dateiserver auf einem dedizierten Computer läuft, was meistens der Fall ist. Unter dem Strich können weniger Anforderungen pro Sekunde verarbeitet werden. Threads führen also zu einem beachtlichen Leistungsgewinn, wobei jedoch jeder Thread nacheinander in der üblichen Weise programmiert wird.

Bis jetzt sahen wir zwei mögliche Entwürfe: einen Multithread- und einen Singlethread-Dateiserver. Nehmen Sie aber an, dass Threads zwar nicht genutzt werden können, dass die Systemdesigner aber nicht mit dem Leistungsverlust durch Singlethreading leben können. Eine dritte Möglichkeit besteht darin, den Server als großen endlichen Zustandsautomaten auszuführen. Wenn eine Anforderung eingeht, wird sie von dem einzigen Thread untersucht. Falls sie aus dem Cache heraus beantwortet werden kann, ist das in Ordnung, wenn nicht, muss eine Nachricht an die Festplatte gesendet werden.

Anstatt zu blockieren, zeichnet der Server den Status der aktuellen Anforderung in einer Tabelle auf und geht dann los, um die nächste Nachricht einzuholen. Diese nächste Nachricht kann entweder eine Anforderung für weitere Arbeit oder eine Antwort von der Festplatte über die vorhergehende Operation sein. Falls es sich um eine neue Aufgabe handelt, wird sie sofort angegangen. Bei einer Antwort von der Festplatte werden die relevanten Informationen von der Tabelle abgerufen und die Ant-

wort wird verarbeitet und anschließend an den Client gesendet. Bei diesem Verfahren muss der Server nicht blockierende Aufrufe für `send` und `receive` verwenden.

Bei diesem Entwurf werfen wir das Modell des »sequenziellen Prozesses« aus den ersten beiden Fällen über Bord. Der Status der Berechnung muss für jede gesendete und empfangene Nachricht explizit in einer Tabelle gespeichert und wiederhergestellt werden. Letztlich simulieren wir dadurch Threads und ihre Stacks auf die harte Tour. Der Prozess wird als endlicher Zustandsautomat betrieben, dem ein Ereignis zugeführt wird und der dann je nachdem, was darin enthalten ist, darauf reagiert.

Modell	Merkmale
Threads	Parallelverarbeitung, blockierende Systemaufrufe
Prozess mit einem Thread	Keine Parallelverarbeitung, blockierende Systemaufrufe
Endlicher Zustandsautomat	Parallelverarbeitung, nicht blockierende Systemaufrufe

Abbildung 3.4: Drei Möglichkeiten zur Konstruktion eines Servers

Damit sollte klar geworden sein, welche Vorteile Threads bieten. Sie machen es möglich, das Prinzip von aufeinanderfolgenden Prozessen mit blockierenden Systemaufrufen (z.B. einem RPC zur Kommunikation mit der Festplatte) beizubehalten und trotzdem eine Parallelverarbeitung durchzuführen. Blockierende Systemaufrufe vereinfachen die Programmierung, während die Parallelverarbeitung die Leistung erhöht. Der Singlethread-Server behält zwar den Komfort und die Einfachheit von blockierenden Systemaufrufen bei, verzichtet aber auf einen Teil der Leistung. Beim Ansatz als endlicher Zustandsautomat wird eine hohe Leistung durch Parallelverarbeitung erzielt, aber es werden nicht blockierende Aufrufe verwendet, was sich schwer programmieren lässt. Eine Zusammenfassung dieser Modelle finden Sie in ▶Abbildung 3.4.

3.2 Virtualisierung

Threads und Prozesse können als eine Möglichkeit betrachtet werden, mehrere Dinge zur selben Zeit zu tun. Letztendlich ermöglichen sie es uns, Programme (oder Teile von Programmen) zu erstellen, die scheinbar gleichzeitig ausgeführt werden. Auf einem Einzelprozessorcomputer ist diese gleichzeitige Ausführung natürlich eine Illusion. Da es nur eine CPU gibt, kann jeweils nur eine Anweisung aus einem einzelnen Thread oder Prozess auf einmal ausgeführt werden. Durch die rasche Umschaltung zwischen Threads und Prozessen wird jedoch die Illusion einer parallelen Verarbeitung erreicht.

Das Prinzip, nur eine CPU zu haben, aber vorzugaukeln, es seien mehrere, lässt sich auch auf andere Ressourcen ausdehnen, was zur sogenannten Ressourcenvirtualisierung führt. Diese Technik wird schon seit Jahrzehnten angewendet, lenkt aber wieder mehr Aufmerksamkeit auf sich, seit (verteilte) Computersysteme immer häufiger anzutreffen sind und immer komplexer werden. Das führt dazu, dass Anwendungssoftware die zugrunde liegende Systemsoftware und Hardware meistens überlebt. In diesem Abschnitt kümmern wir uns um die Funktion der Virtualisierung und erörtern, wie sie realisiert werden kann.

3.2.1 Die Rolle der Virtualisierung in verteilten Systemen

In der Praxis bietet jedes (verteilte) Computersystem eine Programmierschnittstelle zu Software einer höheren Ebene, wie ▶Abbildung 3.5(a) zeigt. Es gibt viele verschiedene Arten von Schnittstellen, von den grundlegenden Befehlssätzen der CPUs bis zur großen Menge von APIs, die zusammen mit heutigen Middleware-Systemen ausgeliefert werden. Letztendlich geht es bei der Virtualisierung darum, eine bestehende Schnittstelle so zu erweitern oder zu ersetzen, dass sie das Verhalten eines anderen Systems nachahmt, wie Abbildung 3.5(b) zeigt. Die technischen Details der Virtualisierung werden wir in Kürze besprechen, aber zunächst konzentrieren wir uns auf die Frage, warum die Virtualisierung für verteilte Systeme so wichtig ist.

Einer der wichtigsten Gründe für die Einführung der Virtualisierung in den 1970er Jahren lag darin, veraltete Software auf teurer Mainframe-Hardware ausführen zu können. Zu dieser Software gehörten nicht nur verschiedene Anwendungen, sondern auch die Betriebssysteme, für die sie entwickelt worden waren. Dieser Ansatz zur Unterstützung veralteter Hardware wurde auf den IBM 370-Mainframes (und ihren Nachfolgemodellen) erfolgreich durchgeführt. Diese Computer boten eine virtuelle Maschine, auf die verschiedene Betriebssysteme portiert worden waren.

Als die Hardware billiger und Computer leistungsfähiger wurden und sich die Anzahl verschiedener Betriebssysteme verringerte, sank die Bedeutung der Virtualisierung. Seit den späten 1990er Jahren änderte sich dies jedoch wieder aus verschiedenen Gründen, die wir im Folgenden besprechen möchten.

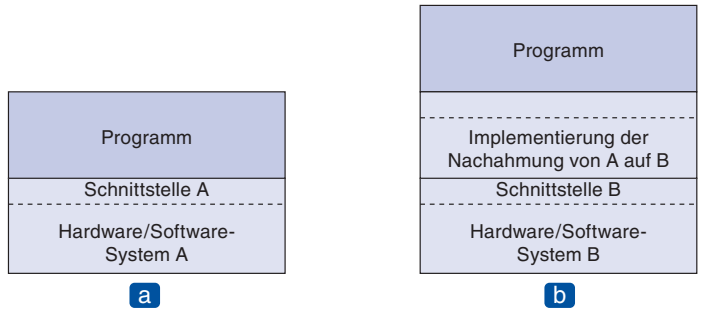


Abbildung 3.5: (a) Allgemeine Gliederung von Programm, Schnittstelle und System; (b) allgemeine Gliederung eines virtuellen Systems A oberhalb von B

Erstens ändern sich Hardware und Systemsoftware der niedrigen Ebene relativ schnell, während die Software auf höheren Abstraktionsebenen (z.B. Middleware und Anwendungen) sehr viel stabiler ist. Mit anderen Worten, wir befinden uns in einer Situation, in der veraltete Software nicht mit der Plattform Schritt halten kann, auf der sie basiert. Hier kann die Virtualisierung dadurch helfen, dass die veralteten Schnittstellen auf die neuen Plattformen portiert werden und Letztere damit sofort für große Menge der bereits bestehenden Programme geöffnet werden.

Ebenso wichtig ist die Tatsache, dass Netzwerke allgegenwärtig geworden sind. Ein moderner Computer, der nicht mit einem Netzwerk verbunden ist, lässt sich nur noch schwer vorstellen. In der Praxis bedeutet dies, dass Systemadministratoren eine große und heterogene Menge von Server-Computern mit unterschiedlichen Anwendungen verwalten müssen, auf die die Clients zugreifen. Gleichzeitig sollen aber die verschiedenen Ressourcen für die Anwendungen leicht zugänglich sein. Hier kann die Virtualisie-

rung viel helfen: Die Unterschiedlichkeit der Plattformen und Computer lässt sich reduzieren, indem letztlich jede Anwendung auf ihrer eigenen virtuellen Maschine läuft, möglicherweise zusammen mit den zugehörigen Bibliotheken und dem Betriebssystem, während diese virtuellen Maschinen wiederum auf einer gemeinsamen Plattform ausgeführt werden.

Die letzte Form der Virtualisierung bietet einen hohen Grad an Portabilität und Flexibilität. In Netzwerken, die die Replikation von dynamischen Inhalten auf einfache Weise unterstützen sollen, wird die Verwaltung nach Awadallah und Rosenblum (2002) deutlich vereinfacht, wenn die Edge-Server die Virtualisierung unterstützen, um eine komplette Site einschließlich ihrer Umgebung dynamisch zu kopieren. Wie wir später sehen werden, ist die Virtualisierung hauptsächlich aufgrund dieser Portabilitätsfragen ein wichtiger Mechanismus für verteilte Systeme.

3.2.2 Architekturen virtueller Maschinen

Es gibt viele verschiedene Möglichkeiten, um die Virtualisierung in die Praxis umzusetzen. Einen Überblick über die einzelnen Ansätze finden Sie bei Smith und Nair (2005). Um die Unterschiede bei der Virtualisierung zu verstehen, ist es wichtig zu erkennen, dass Computersysteme im Allgemeinen vier verschiedene Arten von Schnittstellen auf vier verschiedenen Ebenen aufweisen:

1. Eine Schnittstelle zwischen der Hardware und der Software mit *Maschinenbefehlen*, die von jedem Programm aufgerufen werden können
2. Eine Schnittstelle zwischen der Hardware und der Software mit Maschinenbefehlen, die nur privilegierte Programme wie Betriebssysteme aufrufen können
3. Eine Schnittstelle aus *Systemaufrufen*, wie sie von einem Betriebssystem angeboten wird
4. Eine Schnittstelle aus Bibliotheksaufrufen, allgemein gesagt ein *API* (*Application Programming Interface*). In vielen Fällen werden die zuvor genannten Systemaufrufe durch ein API verborgen.

Diese verschiedenen Typen sehen Sie in ▶Abbildung 3.6. Das Wesen der Virtualisierung besteht darin, das Verhalten dieser Schnittstellen nachzuahmen.

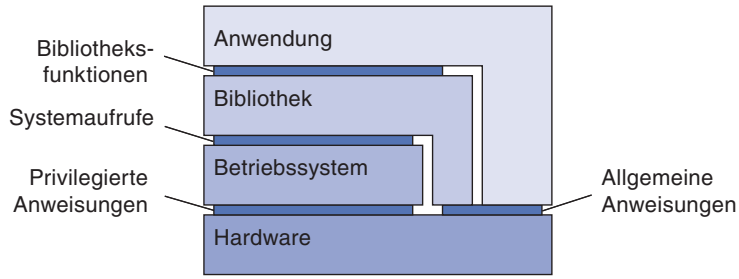


Abbildung 3.6: Verschiedene Schnittstellen von Computersystemen

Virtualisierung kann auf zwei verschiedene Weisen stattfinden. Einerseits können wir ein Laufzeitsystem erstellen, das im Grunde einen abstrakten Befehlssatz zum Ausführen von Anwendungen bereitstellt. Diese Befehle können interpretiert werden (wie es bei der Java-Laufzeitumgebung der Fall ist), aber auch emuliert werden, wie beim Ausführen von Windows-Anwendungen auf UNIX-Plattformen. Beachten Sie, dass der Emulator in letzterem Fall auch das Verhalten der Systemaufrufe nachahmen muss, was sich als alles andere als trivial erwiesen hat. Diese Form der Virtualisierung führt zu dem, was Smith und Nair (2005) als virtuelle Prozessmaschine bezeichnen, um zu betonen, dass die Virtualisierung im Wesentlichen nur für einen einzelnen Prozess stattfindet.

Ein alternativer Ansatz zur Virtualisierung besteht darin, ein System im Wesentlichen in Form einer Schicht bereitzustellen, die die ursprüngliche Hardware vollständig abschirmt, aber den vollständigen Befehlssatz dieser (oder einer anderen) Hardware als Schnittstelle bereitstellt. Entscheidend ist hierbei, dass diese Schnittstelle verschiedenen Programmen *gleichzeitig* angeboten werden kann. Damit ist es jetzt möglich, dass mehrere verschiedene Betriebssysteme auf derselben Plattform laufen. Diese Schicht wird allgemein als Virtual Machine Monitor (*VMM*) bezeichnet. Typische Beispiele für diesen Ansatz sind VMware (Sugerman *et al.*, 2001) und Xen (Barham *et al.*, 2003). ▶Abbildung 3.7 zeigt die beiden unterschiedlichen Ansätze.

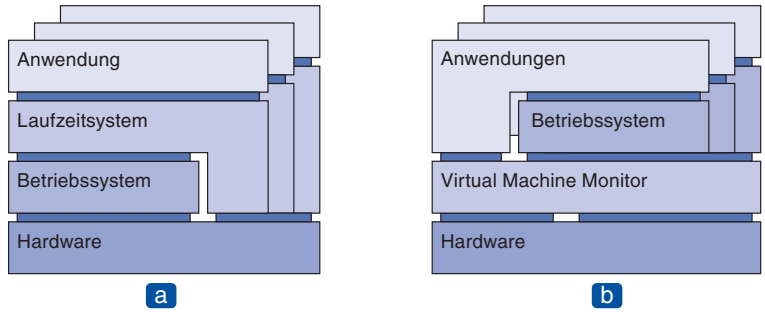


Abbildung 3.7: (a) Eine virtuelle Prozessmaschine mit mehreren Instanzen von Kombinationen aus Anwendung und Laufzeit; (b) ein Virtual Machine Monitor mit mehreren Instanzen von Kombinationen aus Anwendungen und Betriebssystem

Nach Rosenblum und Garfinkel (2005) werden VMMs in Zusammenhang mit Zuverlässigkeit und Sicherheit von (verteilten) Systemen immer bedeutender. Da sie die Isolierung einer vollständigen Anwendung und ihrer Umgebung ermöglichen, beeinträchtigt ein Ausfall aufgrund eines Fehlers oder eines Angriffs nicht mehr den gesamten Computer. Wie wir bereits erwähnt haben, erfährt auch die Portabilität deutliche Verbesserungen, da VMMs eine weitere Entkopplung von Hardware und Software erlauben, wodurch sich ganze Umgebungen von einem Computer auf den anderen verschieben lassen.

3.3 Clients

In den vorangegangenen Kapiteln beschrieben wir das Client-Server-Modell, die Rollen der Clients und Server und deren Interaktion. Jetzt wollen wir uns die Anatomie der Clients und Server genauer ansehen. In diesem Abschnitt beginnen wir mit der Erörterung der Clients. Server sind Thema des nächsten Abschnitts.

3.3.1 Vernetzte Benutzerschnittstellen

Eine der Hauptaufgaben von Client-Computern besteht darin, dem Benutzer eine Möglichkeit zur Interaktion mit einem entfernten Server zu bieten. Es gibt grob gesagt zwei Möglichkeiten, um diese Interaktion zu unterstützen. Bei der ersten verfügt der Client-Computer über ein eigenes Gegenstück zu jedem Remote-Dienst, mit dem er über das Netzwerk Kontakt aufnimmt. Ein typisches Beispiel dafür ist ein Terminkalender auf einem PDA, der mit einem entfernten und möglicherweise gemeinsam genutzten Terminkalender synchronisiert werden muss. In diesem Fall erledigt ein Protokoll auf Anwendungsebene die Synchronisierung, wie ▶Abbildung 3.8(a) zeigt.

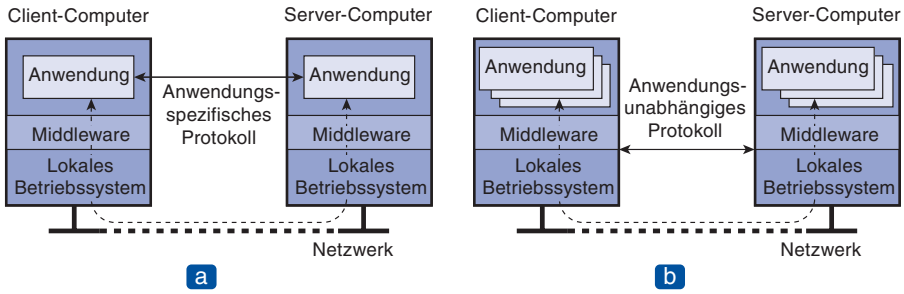


Abbildung 3.8: (a) Eine vernetzte Anwendung mit einem eigenen Protokoll; (b) eine allgemeine Lösung für den Zugriff auf entfernte Anwendungen

Eine zweite Lösung besteht darin, direkten Zugriff auf den entfernten Dienst zu bieten, indem nur eine bequeme Benutzerschnittstelle angeboten wird. Letztendlich bedeutet dies, dass der Client-Computer nur als Terminal ohne die Notwendigkeit eines lokalen Speichers verwendet wird, was zu der anwendungsneutralen Lösung aus Abbildung 3.8(b) führt. Bei vernetzten Benutzerschnittstellen werden alle Informationen auf dem Server verarbeitet und gespeichert. Dieser Thin-Client-Ansatz wird aufgrund der allgegenwärtigen Internetanbindung und der zunehmenden Komplexität von Handheld-Geräten immer bedeutender. Wie wir in einem früheren Kapitel gezeigt haben, sind Thin-Client-Lösungen auch deshalb so beliebt, da sie die Systemverwaltung erleichtern. Werfen wir nun einen Blick darauf, wie vernetzte Benutzerschnittstellen unterstützt werden können.

Beispiel: Das X-Window-System

Eine der möglicherweise ältesten und immer noch weiträumig genutzten vernetzten Benutzerschnittstellen ist das *X-Window-System*. Im Allgemeinen einfach als X bezeichnet, wird es zur Steuerung von Bitmap-Terminals verwendet, die aus einem Monitor, einer Tastatur und einem Zeigegerät wie einer Maus bestehen. In gewissem Sinne kann X als Teil eines Betriebssystems angesehen werden, das das Terminal steuert. Herzstück des Systems ist der sogenannte *X-Kernel*. Er enthält alle terminalspezifischen Gerätetreiber und ist damit im Allgemeinen hochgradig hardwareabhängig.

Der X-Kernel bietet eine Schnittstelle auf relativ niedriger Ebene, um den Bildschirm zu steuern, aber auch, um Ereignisse von der Tastatur und der Maus zu erfassen. Diese Schnittstelle wird Anwendungen in Form der Bibliothek *Xlib* bereitgestellt. Diesen allgemeinen Aufbau zeigt ▶Abbildung 3.9.

Der bemerkenswerte Aspekt von X besteht darin, dass der X-Kernel und die X-Anwendungen sich nicht auf demselben Computer befinden müssen. Insbesondere verfügt X über das *X-Protokoll*, ein Kommunikationsprotokoll auf Anwendungsebene, über das eine Instanz von Xlib Daten und Ereignisse mit dem X-Kernel austauschen kann. So kann Xlib z.B. Anforderungen an den X-Kernel senden, um u.a. ein Fenster zu erstellen oder zu löschen, Farben einzustellen und den Cursortyp festzulegen. Der X-Kernel wiederum reagiert auf lokale Ereignisse wie Tastatur- und Mauseingaben, indem er Ereignispakete zurück an Xlib sendet.

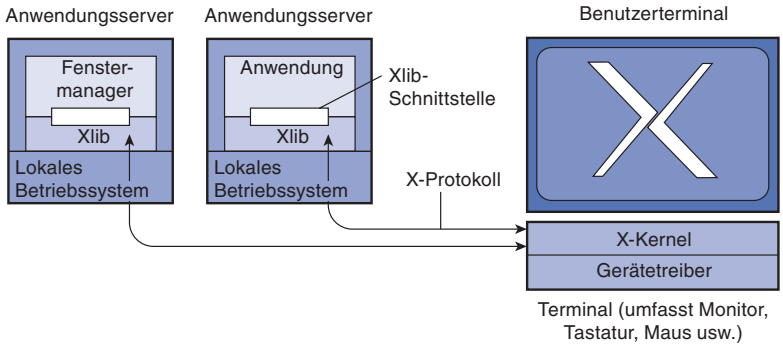


Abbildung 3.9: Grundlegender Aufbau des X-Window-Systems

Verschiedene Anwendungen können zur selben Zeit mit dem X-Kernel kommunizieren. Eine Anwendung hat besondere Rechte, der sogenannte Fenstermanager. Er kann das Erscheinungsbild der Anzeige bestimmen, indem er z.B. vorschreibt, wie die einzelnen Fenster mit zusätzlichen Schaltflächen ausgeschmückt und auf der Anzeige angeordnet werden usw. Andere Anwendungen müssen diese Regeln befolgen.

Es ist bemerkenswert, wie sich das X-Window-System in das Client-Server-Modell einfügt. Nach dem, was wir bis jetzt gesehen haben, ist klar, dass der X-Kernel Anforderungen empfängt, um die Anzeige zu ändern. Diese Anweisungen erhält er von (möglicherweise entfernten) Anwendungen. In diesem Sinne agiert der X-Kernel als Server, während die Anwendungen die Rolle der Clients übernehmen. Diese Terminologie wurde von X angenommen, was zwar streng genommen korrekt ist, aber doch sehr leicht zur Verwirrung führen kann.

Thin Clients im Netzwerk

Anwendungen bearbeiten die Anzeige offensichtlich mithilfe der besonderen Anzeigebefehle von X. Diese Befehle werden im Allgemeinen über das Netzwerk gesendet, um anschließend vom X-Kernel verarbeitet zu werden. Aufgrund ihrer Natur sollten für X geschriebene Anwendungen vorzugsweise die Anwendungslogik von den Befehlen für die Benutzerschnittstelle trennen. Leider ist dies oftmals nicht der Fall. Wie Lai und Nieh (2002) berichten, zeigt es sich, dass ein Großteil der Anwendungslogik und der Benutzerinteraktion eng gekoppelt sind, was bedeutet, dass eine Anwendung viele Anforderungen an den X-Kernel sendet, auf die sie eine Antwort erwartet, bevor sie mit dem nächsten Schritt fortfahren kann. Dieses synchrone Verhalten kann in einem Weitbereichsnetzwerk mit hohen Latenzzeiten die Leistung beeinträchtigen.

Es gibt verschiedene Lösungen für dieses Problem. Eine besteht darin, die Implementierung des X-Protokolls zu ändern, wie es bei NX getan worden ist (Pinzari, 2003). Ein wichtiger Teil dieser Arbeit bestand in der Bandbreitenreduzierung durch die Komprimierung von X-Nachrichten. Nachrichten werden dabei zunächst in einen festen Teil, der als Bezeichner behandelt wird, und einen variablen Teil zerlegt. In vielen Fällen haben mehrere Nachrichten denselben Bezeichner, weshalb sie häufig ähnliche Daten enthalten. Diese Eigenschaft lässt sich nutzen, indem nur die Unterschiede zwischen den Nachrichten mit demselben Bezeichner gesendet werden.

Sowohl der Sender als auch der Empfänger unterhalten einen lokalen Cache, dessen Einträge über den Bezeichner einer Nachricht eingesehen werden können. Wenn eine Nachricht gesendet wird, erfolgt zunächst eine Suche im lokalen Cache. Ein Treffer bedeutet, dass zuvor eine Nachricht mit demselben Bezeichner, aber möglicherweise anderen Daten gesendet worden ist. In diesem Fall wird differenzielle Codierung verwendet, um nur die Unterschiede zwischen den beiden Nachrichten zu senden. Beim Empfänger wird die Nachricht ebenfalls im lokalen Cache nachgeschlagen, woraufhin die Decodierung aufgrund der Unterschiede stattfindet. Ist keine ähnliche Nachricht im Cache, werden standardmäßige Komprimierungstechniken verwendet, die im Allgemeinen schon eine Bandbreitenverbesserung um den Faktor 4 hervorrufen. Insgesamt soll diese Technik zu einer Bandbreitenreduzierung bis zum Faktor 1000 führen, was es ermöglicht, X auch über Verbindungen geringer Bandbreite mit nur 9600 Kbit/s auszuführen.

Eine wichtige Nebenwirkung der Zwischenspeicherung von Nachrichten ist, dass Sender und Empfänger die Informationen über den aktuellen Status der Anzeige teilen. So kann die Anwendung z.B. geometrische Informationen über verschiedene Objekte anfordern, indem sie einfach eine Suche im lokalen Cache anfordert. Allein diese gemeinsamen Informationen verringern die Anzahl der erforderlichen Nachrichten, um Anwendung und Anzeige synchron zu halten.

Trotz dieser Verbesserungen erfordert X nach wie vor, dass ein Anzeigeserver läuft. Das ist schon viel verlangt, vor allem, wenn es sich bei dem Anzeigegerät um etwas so einfaches wie ein Handy handelt. Eine Lösung dafür, die Software bei der Anzeige zu belassen, besteht ganz einfach darin, die gesamte Verarbeitung auf der Seite der Anwendung durchzuführen. Letztlich bedeutet das, dass die gesamte Anzeige bis zur Ebene einzelner Pixel vollständig von der Anwendungsseite gesteuert wird. Änderungen in der Bitmap werden über das Netzwerk zur Anzeige gesendet, wo sie sofort in den lokalen Frame-Buffer übertragen werden.

Dieser Ansatz erfordert ausgeklügelte Komprimierungstechniken, damit die Bandbreite nicht zu einem Problem wird. Stellen Sie sich z.B. einen Video-Stream mit einer Rate von 30 Frames pro Sekunde auf einem Bildschirm von 320 x 240 vor, eine übliche Größe für PDAs. Wenn jeder Pixel mit 24 Bit codiert ist, dann brauchen Sie ohne Komprimierung eine Bandbreite von ungefähr 53 Mbit/s. In solchen Fällen ist eine Komprimierung eindeutig erforderlich und es werden zurzeit auch viele Techniken dafür zur Verfügung gestellt. Beachten Sie aber, dass eine Komprimierung eine Dekomprimierung beim Empfänger erfordert, was ohne Hardwareunterstützung die Berechnung aufwändig macht. Zwar ist eine Hardwareunterstützung möglich, doch schraubt diese wiederum die Gerätekosten in die Höhe.

Der Nachteil des Sendens roher Pixeldaten im Vergleich zu Protokollen höherer Ebene wie X besteht darin, dass Sie die Anwendungssemantik nicht nutzen können, da sie auf dieser Ebene verloren geht. Baratto *et al.* (2005) schlagen eine andere Technik

vor. In ihrer Lösung, die sie THINC nennen, stellen sie einige wenige Anzeigebefehle hoher Ebene bereit, die auf der Ebene von Videogerätetreibern arbeiten. Damit sind diese Befehle geräteabhängig, leistungsfähiger als direkte Pixeloperationen, aber weniger leistungsfähig als Protokolle wie X. Als Folge können Anzeigeserver viel einfacher ausfallen, was für die CPU-Nutzung gut ist, während gleichzeitig anwendungsabhängige Optimierungen verwendet werden können, um Bandbreite und Synchronisierung zu verringern.

In THINC werden Anzeigeanforderungen von der Anwendung abgefangen und in Befehle niedrigerer Ebene übersetzt. Dadurch kann THINC die Anwendungssemantik nutzen, um zu entscheiden, welche Kombination von Befehlen der niedrigen Ebene am besten geeignet ist. Die übersetzten Befehle werden nicht sofort zur Anzeige gesendet, sondern in eine Warteschlange eingereiht. Durch die Stapelung mehrerer Befehle ist es möglich, sie zu einem einzigen zusammenzufassen, was die Anzahl der Nachrichten verringert. Wenn z.B. ein neuer Zeichenbefehl für einen Bereich des Bildschirms das überschreibt, was ein vorheriger Befehl (der sich noch in der Warteschlange befindet) ausgeführt hätte, muss Letzterer nicht mehr an die Anzeige gesendet werden. Darüber hinaus muss die Anzeige nicht um Aktualisierungen nachsuchen, da diese von THINC Aktualisierungen gesendet werden, sobald sie verfügbar sind. Dieser Push-Ansatz verhindert Verzögerungen, da keine Notwendigkeit mehr besteht, dass die Anzeige eine Aktualisierungsanforderung sendet.

Es hat sich gezeigt, dass der Ansatz von THINC eine bessere Gesamtleistung bietet, auch wenn sie in der gleichen Größenordnung liegt wie die von NX. Einzelheiten über den Leistungsvergleich finden sich bei Baratto *et al.* (2005).

Verbunddokument

Moderne Benutzerschnittstellen leisten sehr viel mehr als Systeme wie X oder einfache Anwendungen. Vor allem ermöglichen viele Benutzerschnittstellen es den Anwendungen, ein grafisches Fenster gemeinsam zu nutzen und damit Daten durch Benutzeraktionen auszutauschen. Außerdem lassen sich zusätzliche Benutzeraktionen wie *Drag&Drop* und *direkte Bearbeitung* durchführen.

Ein typisches Beispiel der Drag&Drop-Funktionalität ist das Verschieben eines Symbols für Datei *A* zu einem Symbol, das einen Papierkorb darstellt, um damit die Datei zu löschen. In diesem Fall muss die Benutzerschnittstelle mehr tun, als nur Symbole auf dem Bildschirm anzuordnen: Sie muss den Namen der Datei *A* an die Anwendung übergeben, die mit dem Papierkorbsymbol verknüpft ist, sobald das Symbol der Datei auf das Symbol des Papierkorbes gezogen wird. Andere Beispiele dieser Art lassen sich leicht finden.

Die direkte Bearbeitung lässt sich am besten anhand eines Dokumentes zeigen, das Text und Grafiken enthält. Stellen Sie sich vor, dass dieses Dokument in einer standardmäßigen Textverarbeitung angezeigt wird. Sobald der Benutzer den Mauszeiger über einem Bild platziert, übergibt die Benutzerschnittstelle diese Information an ein Zeichenprogramm, damit der Benutzer das Bild bearbeiten kann. Dabei kann der Benutzer das Bild z.B. drehen, was die Platzierung der Grafik im Dokument ändert. Daher muss die Benutzerschnittstelle die neue Höhe und Breite des Bildes herausfinden und diese Informationen an die Textverarbeitung übergeben. Letztere kann dann wiederum das Seitenlayout des Dokumentes automatisch aktualisieren.

Der Schlüssel zu Benutzerschnittstellen dieser Art ist das Verbunddokument. Er lässt sich als Sammlung von Dokumenten möglicherweise unterschiedlicher Art (z.B. Text, Bilder, Arbeitsblätter usw.) auffassen, die auf der Ebene der Benutzerschnittstelle alle nahtlos integriert sind. Eine Benutzerschnittstelle, die einen solchen Dokumentenverband verarbeitet, kann die Tatsache verbergen, dass verschiedene Anwendungen an den einzelnen Teilen arbeiten. Für den Benutzer erscheinen alle Teile nahtlos zusammengesetzt. Wenn sich die Änderung eines Teils auf die anderen Teile auswirkt, kann die Benutzerschnittstelle die erforderlichen Maßnahmen ergreifen, indem sie z.B. die entsprechenden Anwendungen benachrichtigt.

Wie beim X-Window-System müssen die Anwendungen, die mit dem Verbunddokument verknüpft sind, nicht auf dem Client-Computer ausgeführt werden. Es sollte allerdings deutlich sein, dass Benutzerschnittstellen, die ein Verbunddokument unterstützen, sehr viel mehr Verarbeitung zu leisten haben als Benutzerschnittstellen ohne Verarbeitung von Verbunddokumenten.

3.3.2 Clientseitige Software für die Verteilungstransparenz

Zu Client-Software zählt mehr als nur Benutzerschnittstellen. In vielen Fällen werden einige Teile der Verarbeitungs- und Datenschicht einer Client-Server-Anwendung auch auf der Client-Seite ausgeführt. Eine besondere Klasse bildet die eingebettete Client-Software, z.B. für Geldautomaten, Registrierkassen, Balkencode-Lesegeräte, Fernseh-Boxen usw. Hierbei ist die Benutzerschnittstelle im Gegensatz zu den lokalen Verarbeitungs- und Kommunikationseinrichtungen nur ein relativ kleiner Teil der Client-Software.

Neben Benutzerschnittstellen und anderer anwendungsbezogener Software umfasst die Client-Software auch Komponenten für die Verteilungstransparenz. Im Idealfall sollte der Client nicht wissen, dass er mit einem entfernten Prozess kommuniziert. Im Gegensatz dazu ist die Verteilung für Server aus Gründen der Leistung und der Korrektheit oft weniger transparent. In *Kapitel 6* zeigen wir z.B., dass replizierte Server manchmal miteinander kommunizieren müssen, damit die Operationen auf den einzelnen Replikaten in einer bestimmten Reihenfolge ausgeführt werden.

Die Zugriffstransparenz wird im Allgemeinen erreicht, indem ein Client-Stub aus einer Schnittstellendefinition dessen erstellt wird, was der Server anzubieten hat. Der Stub bietet dieselbe Schnittstelle wie diejenige, die auf dem Server verfügbar ist, verbirgt aber mögliche Unterschiede in der Computerarchitektur sowie die Kommunikation.

Um Orts-, Migrations- und Relokationstransparenz zu erzielen, gibt es verschiedene Möglichkeiten. Der Einsatz eines geeigneten Namenssystems ist entscheidend, wie wir im nächsten Kapitel sehen werden. In vielen Fällen ist auch die Kooperation mit der clientseitigen Software wichtig. Wenn ein Client z.B. bereits an einen Server gebunden ist, kann er direkt darüber informiert werden, dass der Server seinen Standort ändert. In diesem Fall verbirgt die Middleware des Clients die geografische Position des Servers vor dem Benutzer und führt gegebenenfalls eine transparente Neubindung durch. Schlimmstenfalls erfährt die Client-Anwendung einen vorübergehenden Leistungsabfall.

Auf ähnliche Weise implementieren verteilte Systeme die Replikationstransparenz durch clientseitige Lösungen. Stellen Sie sich z.B. ein verteiltes System mit replizierten Servern vor. Eine solche Replikation lässt sich durch Weiterleiten einer Anforderung an jedes einzelne Replikat erreichen, wie ▶Abbildung 3.10 zeigt. Die clientseitige Software kann dann alle Antworten transparent sammeln und eine einzige Antwort an die Client-Anwendung übergeben.

Bedenken Sie zum Schluss noch die Ausfalltransparenz. Der Ausfall der Kommunikation mit einem Server wird gewöhnlich durch Client-Middleware verborgen. Sie kann z.B. so eingerichtet sein, dass sie wiederholt versucht, Verbindung mit einem Server aufzunehmen, oder es nach mehreren Fehlschlägen bei einem anderen Server versucht. Es gibt sogar Situationen, in denen die Client-Middleware Daten zurückgibt, die sie in einer vorherigen Sitzung zwischengespeichert hat, wie es zuweilen Webbrowser tun, die keine Verbindung mit einem Server aufnehmen können.

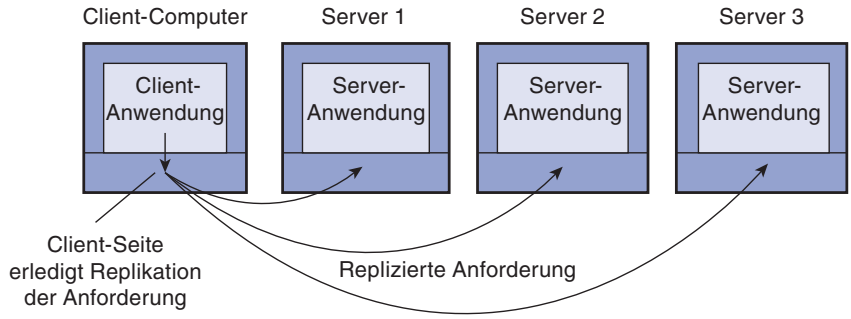


Abbildung 3.10: Transparente Replikation eines Servers mithilfe einer clientseitigen Lösung

Nebenläufigkeitstransparenz lässt sich durch besondere Zwischenserver erreichen, vor allem Transaktionsmonitore, und erfordert weniger Unterstützung durch die Client-Software. Auch die Persistenztransparenz liegt häufig vollständig in den Händen des Servers.

3.4 Server

Wir wollen uns nun den Aufbau von Servern genauer anschauen. Auf den folgenden Seiten konzentrieren wir uns zunächst auf einige allgemeine Entwurfsfragen, bevor wir anschließend Servercluster besprechen.

3.4.1 Allgemeine Entwurfsfragen

Ein Server ist ein Prozess, der einen bestimmten Dienst für verschiedene Clients implementiert. Letzten Endes sind alle Server gleich aufgebaut: Sie warten auf eine eingehende Anforderung von einem Client und sorgen dafür, dass er eine Antwort erhält. Anschließend warten sie wieder auf die nächste Anforderung.

Es gibt verschiedene Wege, um Server zu gestalten. Ein iterativer Server handhabt die Anforderung selbst und gibt dem Client gegebenenfalls eine Antwort zurück. Ein nebenläufiger Server *(concurrent server)* dagegen bearbeitet die Anforderung nicht selbst, sondern übergibt sie an einen separaten Thread oder einen anderen Prozess, woraufhin er dann sofort wieder auf den Eingang der nächsten Anforderung wartet. Ein Multithread-Server ist ein Beispiel für einen solchen parallelen Server. Eine alternative Implementierung eines parallelen Servers besteht darin, für jede eingehende Anforderung einen neuen Prozess zu erstellen. Diesem Ansatz folgen viele UNIX-Systeme. Der Thread oder Prozess, der die Anforderung bearbeitet, ist dafür verantwortlich, eine Antwort an den Client zurückzugeben.

Ein anderes Problem besteht darin, wie die Clients Kontakt mit dem Server aufnehmen. In jedem Fall sendet der Client seine Anforderungen an einen *Endpunkt* oder *Port* auf dem Computer, auf dem der Server ausgeführt wird. Ein Server hört jeweils einen bestimmten Endpunkt ab. Woher aber kennen die Clients den Endpunkt eines Dienstes? Ein Ansatz besteht darin, die Endpunkte für Standarddienste global zuzuweisen. Server, die FTP-Anforderungen bearbeiten, hören z.B. immer TCP-Port 21 ab, während HTTP-Server für das World Wide Web stets auf TCP-Port 80 lauschen. Diese Endpunkte wurden von der Internet Assigned Numbers Authority (IANA) zugewiesen und sind bei Reynolds und Postel (1994) dokumentiert. Bei einem zugewiesenen Endpunkt muss der Client nur noch die Netzwerkadresse des Computers herausfinden, auf dem der Server läuft. Wie wir im nächsten Kapitel erklären werden, können für diesen Zweck Namensdienste zum Einsatz kommen.

Es gibt viele Dienste, die keinen zuvor zugewiesenen Endpunkt benötigen. So kann z.B. ein Uhrzeitserver einen Endpunkt nutzen, der ihm von seinem lokalen Betriebssystem zugewiesen wird. In diesem Fall muss ein Client zunächst den Endpunkt herausfinden. Eine Lösung dafür besteht darin, dass auf allen Computern mit Servern ein besonderer Daemon läuft, der die jeweils aktuellen Endpunkte der einzelnen Dienste aller Server auf demselben Rechner aufzeichnet. Der Daemon selbst hört einen Standardendpunkt ab. Ein Client nimmt zuerst Kontakt mit dem Daemon auf, fordert den Endpunkt an und greift dann auf den gewünschten Server zu, wie Sie in ▶Abbildung 3.11(a) sehen.

Es ist üblich, einen Endpunkt mit einem bestimmten Dienst zu verbinden. Jeden Dienst durch einen eigenen Server zu implementieren, kann jedoch Ressourcenverschwendung bedeuten. In einem typischen UNIX-System laufen z.B. gewöhnlich viele Server gleichzeitig, wobei die meisten passiv warten, bis eine Client-Anforderung hereinkommt. Anstatt so viele passive Prozesse nachzuverfolgen, ist es oftmals effizienter, wenn ein einzelner Superserver alle mit einem bestimmten Dienst verbundenen Endpunkte abhört, wie Abbildung 3.11(b) zeigt. Dieser Ansatz wird z.B. beim `inetd`-Daemon in UNIX verfolgt. `Inetd` hört eine Reihe von Standardports für Internetdienste ab. Wenn eine Anforderung eingeht, erstellt der Daemon einen neuen Prozess, der sich um diese Anforderung kümmert. Wenn dieser Prozess abgeschlossen ist, wird er beendet.

Ein weiterer Aspekt, der beim Entwurf eines Servers bedacht werden muss, besteht darin, ob und wie ein Server unterbrochen werden kann. Stellen Sie sich z.B. einen Benutzer vor, der gerade damit begonnen hat, eine umfangreiche Datei auf einen FTP-Server hochzuladen. Plötzlich jedoch bemerkt er, dass dies die falsche Datei ist, und möchte den Server anhalten, um die Datenübertragung abzubrechen. Dazu gibt es mehrere Möglichkeiten. Ein Ansatz, der im Internet zurzeit nur zu gut funktioniert (und manchmal die einzige Alternative darstellt), besteht darin, dass der Benutzer abrupt die Client-Anwendung beendet (was automatisch die Verbindung zum Server trennt), sie sofort neu startet und so tut, als sei nichts gewesen. Der Server wird schließlich die alte Verbindung aufgeben, da er der Meinung ist, der Client sei abgestürzt.

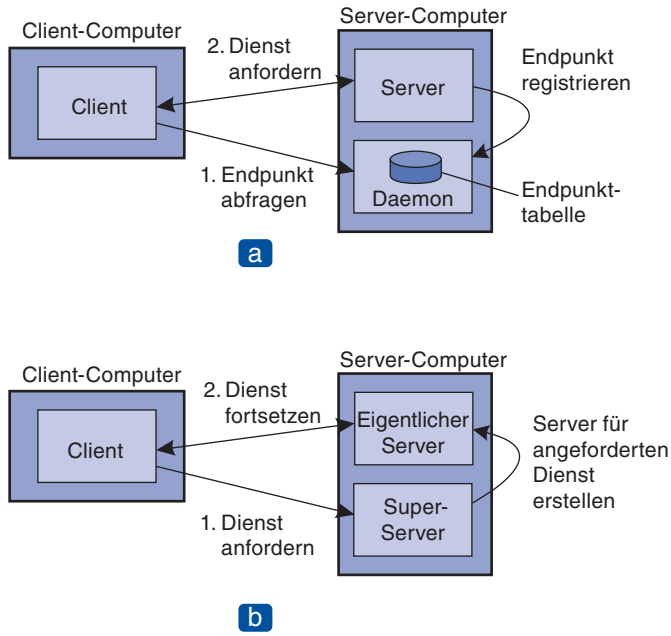


Abbildung 3.11: (a) Client-Server-Bindung mithilfe eines Daemon; (b) Client-Server-Bindung mithilfe eines Superservers

Ein weit besserer Ansatz zur Handhabung von Kommunikationsunterbrechungen besteht darin, den Client und den Server so zu entwickeln, dass sie in der Lage sind, Out-of-band-Daten zu senden. Dabei handelt es sich um Daten, die vom Server vor allen anderen Daten von diesem Client verarbeitet werden müssen. Eine Lösung dafür besteht darin, den Server einen Steuerendpunkt abhören zu lassen, an den der Client Out-of-band-Daten sendet, und gleichzeitig (mit geringerer Priorität) den Endpunkt für die normalen Daten. Es ist auch möglich, die Out-of-band-Daten über dieselbe Verbindung zu übertragen wie die eigentliche Anforderung. Bei TCP können Sie z.B. dringende Daten übertragen. Wenn sie am Server ankommen, wird er unterbrochen (in UNIX-Systemen z.B. durch ein Signal), sodass er die Daten untersuchen und entsprechend handhaben kann.

Eine letzte wichtige Entwurfsentscheidung legt fest, ob der Server zustandslos ist oder nicht. Ein zustandsloser Server unterhält keine Informationen über den Status seiner Clients und kann seinen eigenen Zustand ändern, ohne irgendeinen Client darüber informieren zu müssen (Birman, 2005). Webserver sind z.B. zustandslos. Sie antworten lediglich auf eingehende HTTP-Anforderungen, also auf Anfragen zum Hochladen einer Datei auf den Server oder (was am häufigsten der Fall ist) zum Abrufen einer Datei. Nachdem die Anforderung verarbeitet worden ist, vergisst der Webserver den Client vollständig. Auch die Zusammenstellung der Dateien, die der Webserver verwaltet (möglicherweise zusammen mit einem Dateiserver) kann verändert werden, ohne dass die Clients darüber informiert werden müssen.

In vielen zustandslosen Entwürfen unterhält der Server zwar in Wirklichkeit Informationen über die Clients, doch ist es hierbei entscheidend, dass es nicht zu Dienstunterbrechungen kommt, wenn diese Informationen verloren gehen. Ein Webserver protokolliert z.B. im Allgemeinen alle Client-Anforderungen. Diese Informationen

sind nützlich, z.B. um zu entscheiden, ob und wohin einzelne Dokumente repliziert werden sollen. Wenn diese Aufzeichnungen verloren gehen, führt das jedoch zu keinen weiteren Nachteilen außer vielleicht einer nicht mehr optimalen Leistung.

Eine Sonderform des zustandslosen Entwurfs ist ein Server mit einem »weichen« Status (*Soft State*). In diesem Fall verspricht der Server, Zustandsinformationen für den Client aufzuzeichnen, aber nur für begrenzte Zeit. Wenn diese Zeit abgelaufen ist, kehrt der Server zu seinem Standardverhalten zurück und verwirft dabei jegliche Informationen, die er für den zugehörigen Client aufbewahrt hat. Ein Beispiel für diesen Typ ist ein Server, der einen Client über Aktualisierungen auf dem Laufenden hält, aber nur eine bestimmte Zeit lang, nach deren Ablauf der Client sich wegen Aktualisierungen selbst an den Server wenden muss. Der Soft-State-Ansatz entspringt dem Protokollentwurf in Computernetzwerken, kann aber genauso gut auch auf das Server-Design angewendet werden (Clark, 1989; Lui *et al.*, 2004).

Dagegen pflegt ein zustandsbehafteter Server beständige Informationen über seine Clients. Das bedeutet, dass diese Informationen ausdrücklich vom Server gelöscht werden müssen. Ein typisches Beispiel ist ein Dateiserver, der es einem Client erlaubt, eine lokale Kopie einer Datei anzulegen, auch für Aktualisierungsvorgänge. Ein solcher Server unterhält eine Tabelle mit (*Client, Datei*)-Einträgen. Mithilfe einer solchen Tabelle kann der Server nachverfolgen, welcher Client zurzeit die Aktualisierungsberechtigungen für die Datei und damit wahrscheinlich auch die aktuellste Version hat.

Dieser Ansatz kann die Geschwindigkeit von Lese- und Schreiboperationen aus Client-Sicht verbessern. Die Leistungssteigerung gegenüber zustandslosen Servern ist häufig ein wichtiger Vorteil des zustandsbehafteten Entwurfs. Das Beispiel zeigt aber auch einen großen Nachteil von zustandsbehafteten Servern. Wenn der Server abstürzt, muss er die Tabelle mit den (*Client, Datei*)-Einträgen wiederherstellen, da er ansonsten nicht dafür garantieren kann, dass er die jüngsten Aktualisierungen einer Datei verarbeitet. Im Allgemeinen muss ein zustandsbehafteter Server seinen gesamten Status so wiederherstellen, wie er vor dem Ausfall war. Wie wir in *Kapitel 8* zeigen werden, können die Möglichkeiten zur Wiederherstellung zu einer bemerkenswerten Komplexität führen. Beim zustandlosen Entwurf müssen keine besonderen Maßnahmen für die Wiederherstellung eines abgestürzten Servers getroffen werden. Ein solcher Server startet einfach neu und wartet darauf, dass Client-Anforderungen eingehen.

Ling *et al.* (2004) haben gezeigt, dass zwischen dem (temporären) Sitzungsstatus und dem permanenten Status unterschieden werden sollte. Das vorstehende Beispiel ist typisch für einen Sitzungsstatus: Er ist mit einer Serie von Operationen eines einzelnen Benutzers verbunden und sollte eine bestimmte Zeit lang aufgezeichnet werden, aber nicht unbegrenzt. Der Sitzungsstatus wird häufig in dreistufigen Client-Server-Architekturen gepflegt, wo der Anwendungsserver über eine Reihe von Abfragen auf einen Datenbankserver zugreifen muss, bevor er in der Lage ist, dem anfordernden Client zu antworten. Hierbei wird kein großer Schaden angerichtet, wenn der Sitzungsstatus verloren geht, vorausgesetzt, dass der Client die ursprüngliche Anforderung einfach erneut stellen kann. Aufgrund dieser Beobachtung ist eine einfachere und weniger zuverlässige Speicherung des Status möglich.

Was für den permanenten Status übrig bleibt, sind gewöhnlich Informationen, die in Datenbanken aufbewahrt werden, z.B. Kundendaten, Schlüssel für erworbene Softwareprodukte usw. Die Pflege des Sitzungsstatus bedeutet bei den meisten verteilten Systemen aber bereits ein zustandsbehaftetes Design, bei dem besondere Maßnahmen für Ausfälle und explizite Annahmen über die Dauerhaftigkeit des auf dem Server

gespeicherten Status getroffen werden. Bei der Besprechung der Fehlertoleranz werden wir uns noch ausdrücklich um diese Themen kümmern.

Beim Entwurf eines Servers sollte die Entscheidung für ein zustandsloses oder -behaftetes Design die angebotenen Dienste nicht beeinträchtigen. Wenn z.B. Dateien geöffnet werden müssen, bevor sie gelesen oder in sie geschrieben werden kann, dann sollte auch ein zustandsloser Server dieses Verhalten auf irgendeine Weise nachahmen. Eine übliche Lösung, die wir ausführlicher in *Kapitel 11* besprechen, besteht darin, dass der Server auf eine Lese- oder Schreibanforderung reagiert, indem er zuerst die betreffende Datei öffnet, anschließend die eigentliche Lese- oder Schreiboperation durchführt und die Datei unmittelbar darauf wieder schließt.

In anderen Fällen soll ein Server vielleicht das Verhalten eines Clients aufzeichnen, sodass er effektiver auf dessen Anforderungen reagieren kann. Webserver bieten z.B. manchmal die Möglichkeit, einen Client direkt zu seinen Lieblingsseiten weiterzuleiten. Dies ist nur möglich, wenn der Server Verlaufsinformationen über diesen Client hat. Wenn der Server keine Statusinformationen aufzeichnen kann, besteht eine übliche Lösung darin, den Client zusätzliche Informationen über frühere Zugriffsvorgänge senden zu lassen. Im Web werden solche Informationen häufig im Browser des Clients im Hintergrund gespeichert, und zwar in einem so genannten *Cookie*, einer kleinen Datenstruktur, die clientspezifische Informationen für den Server enthält. Clients werden niemals von einem Browser ausgeführt, sondern lediglich gespeichert.

Wenn ein Client zum ersten Mal auf einen Server zugreift, sendet dieser zusammen mit den angeforderten Webseiten ein Cookie zurück an den Browser, woraufhin der Browser das Cookie sicher ablegt. Bei jedem zukünftigen Zugriff des Clients auf diesen Server wird das entsprechende Cookie zusammen mit der Anforderung gesendet. Dies funktioniert im Prinzip zwar sehr gut, doch wird die Tatsache, dass Cookies für die Aufbewahrung im Browser zurückgesendet werden, oft vollständig vor den Benutzern verborgen. So viel zum Thema Datenschutz.

3.4.2 Servercluster

In *Kapitel 1* haben wir Cluster kurz als eine der vielen Erscheinungsformen verteilter Systeme erwähnt. Jetzt schauen wir uns den Aufbau von Serverclustern zusammen mit den wichtigsten Entwurfsfragen genauer an.

Allgemeiner Aufbau

Einfach ausgedrückt, ist ein Servercluster nichts anderes als eine Ansammlung von Computern, die über ein Netzwerk verbunden sind, wobei jeder Rechner einen oder mehrere Server ausführt. Die Servercluster, die wir hier betrachten, sind über ein LAN (lokales Netzwerk) verbunden, das häufig eine hohe Bandbreite und geringe Latenz bietet.

In den meisten Fällen ist ein Servercluster logisch in drei Tiers, also logische Stufen, gegliedert, wie ▶Abbildung 3.12 zeigt. Die erste Schicht besteht aus einem (logischen) Schalter, durch den die Client-Anforderungen weitergeleitet werden. Die Natur eines solchen Schalters kann sehr stark schwanken. Schalter auf der Transportebene nehmen z.B. eingehende TCP-Verbindungsanforderungen entgegen und übergeben sie an einen der Server im Cluster, wie wir im Folgenden noch besprechen werden. Ein völlig anderes Beispiel stellt ein Webserver da, der eingehende HTTP-Anforderungen annimmt und sie in Teilen zur Weiterverarbeitung an Anwendungsserver übergibt, um später die Ergebnisse wieder zusammenzufassen und eine HTTP-Antwort zurückzugeben.

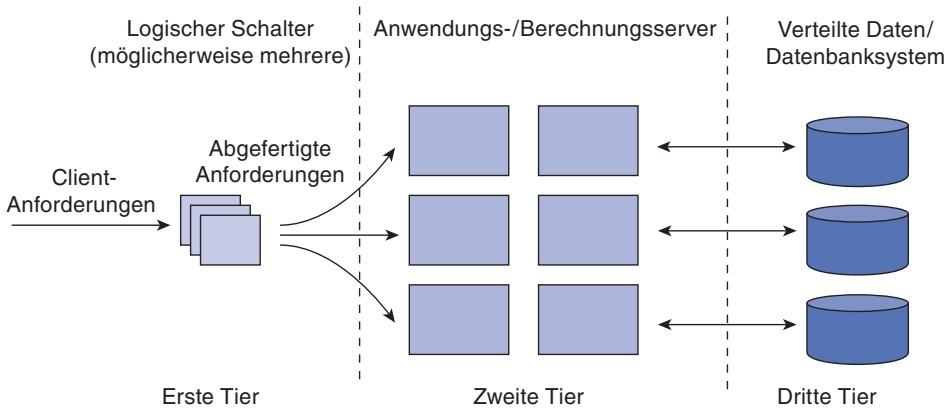


Abbildung 3.12: Allgemeiner Aufbau eines Drei-Tier-Serverclusters

Wie andere mehrschichtige Client-Server-Architekturen enthalten auch viele Servercluster dedizierte Server für die Anwendungsverarbeitung. In Clustern handelt es sich dabei gewöhnlich um Server auf moderner Hardware, die vor allem hohe Rechenleistung bieten soll. In Serverclustern von Unternehmen können jedoch auch Anwendungen betrieben werden, die auf relativ einfachen Rechnern laufen, da nicht die Rechenleistung einen möglichen Engpass darstellt, sondern der Zugriff auf den Speicher.

Damit kommen wir zur dritten Schicht, die aus Servern zur Datenverarbeitung besteht, vor allem aus Datei- und Datenbankservern. Je nach der Verwendung des Serverclusters können auch diese Server auf spezialisierten Computern für den schnellen Zugriff auf die Festplatten und mit großen serverseitigen Datencaches laufen.

Natürlich folgen nicht alle Servercluster dieser strikten Aufteilung. Es ist häufig der Fall, dass jeder Rechner mit seinem eigenen lokalen Speicher ausgestattet ist. Oft sind Anwendungs- und Datenverarbeitung auf einem einzelnen Server vereint, was zu einer zweischichtigen Architektur führt. Bei der Handhabung von Streaming Media durch Servercluster ist die Bereitstellung einer zweischichtigen Systemarchitektur üblich, bei der jeder Rechner als dedizierter Medienserver fungiert (Steinmetz und Nahrstedt, 2004).

Wenn ein Servercluster mehrere Dienste anbietet, kann es vorkommen, dass auf verschiedenen Computern unterschiedliche Anwendungsserver ausgeführt werden. Daher muss der Umschalter in der Lage sein, zwischen den Diensten zu unterscheiden, da er die Anforderungen sonst nicht an den jeweils geeigneten Computer weiterleiten kann. Es hat sich jedoch herausgestellt, dass viele Rechner auf der zweiten Tier nur eine einzige Anwendung ausführen.

Folglich kann es vorkommen, dass einige Rechner zeitweilig im Leerlauf sind, während andere von Anforderungen überlastet werden. In einem solchen Fall wäre es nützlich, Dienste vorübergehend auf die unbeschäftigten Computer zu verlagern. Eine von Awadallah und Rosenblum (2004) vorgeschlagene Lösung sieht die Verwendung virtueller Rechner vor, was eine relativ einfache Verlagerung des Codes auf physische Computer ermöglicht. Auf diese Codemigration gehen wir weiter hinten in diesem Kapitel noch ein.

Sehen wir uns die erste Tier, die aus dem Schalter besteht, etwas genauer an. Ein wichtiges Entwurfsziel für Servercluster ist, die Tatsache zu verbergen, dass mehrere Server vorhanden sind. Mit anderen Worten, die Client-Anwendungen auf Remote-Computern müssen nichts über den inneren Aufbau des Clusters wissen. Dieses

Verbergen des Zugriffs wird ausnahmslos mithilfe eines einzelnen Zugriffspunkts erreicht, der wiederum durch eine Art von Hardwareswitch wie einen dedizierten Computer implementiert wird.

Der Schalter bildet den Eintrittspunkt zum Servercluster, wobei er nur eine einzige Netzwerkadresse zeigt. Aus Gründen der Skalierbarkeit und Verfügbarkeit kann ein Servercluster mehrere Zugriffspunkte haben, die jeweils von einem anderen dedizierten Rechner realisiert werden. Wir betrachten hier nur den Fall des einzelnen Zugriffspunkts.

Die Standardmöglichkeit für den Zugriff auf einen Servercluster besteht in der Einrichtung einer TCP-Verbindung, über die Anforderungen auf Anwendungsebene im Rahmen einer Sitzung gesendet werden. Eine Sitzung endet, wenn die Verbindung aufgelöst wird. Schalter auf der Transportschicht nehmen eingehende TCP-Verbindungsanforderungen entgegen und geben eine solche Verbindung an einen der Server aus (Hunt *et al.*, 1997; Pai *et al.*, 1998). Die grundlegende Funktionsweise dieses sogenannten *TCP-Handoff* wird in ▶Abbildung 3.13 gezeigt.

Wenn der Schalter eine Anforderung für eine TCP-Verbindung empfängt, ermittelt er anschließend den Server, der am besten zur Bearbeitung dieser Anfrage geeignet ist, und leitet das Anforderungspaket an diesen Server weiter. Der Server wiederum sendet eine Bestätigung an den anfordernden Client zurück, setzt aber die IP-Adresse des Schalters als Quellfeld in den Header des IP-Pakets mit dem TCP-Segment ein. Beachten Sie, dass diese vorgegaukelte Adresse notwendig ist, damit der Client TCP ausführen kann: Das Protokoll erwartet eine Antwort vom Schalter zurück, nicht von einem anderen Server, von dem es noch nie zuvor gehört hat. Die Implementierung des TCP-Handoff erfordert daher offensichtlich Änderungen auf Betriebssystemebene.

Es wird jetzt bereits deutlich, dass der Schalter eine wichtige Rolle bei der Verteilung der Last über die einzelnen Server spielen kann. Durch die Entscheidung, wohin eine Anforderung weitergeleitet wird, bestimmt der Schalter auch, welcher Server die weitere Verarbeitung der Anfrage handhaben soll. Die einfachste Richtlinie zum Lastenausgleich, der der Schalter folgen kann, ist das Umlaufverfahren (»Round Robin«): Jedes Mal, wenn er eine Anforderung weiterleitet, wählt er dafür den nächsten Server von seiner Liste aus.

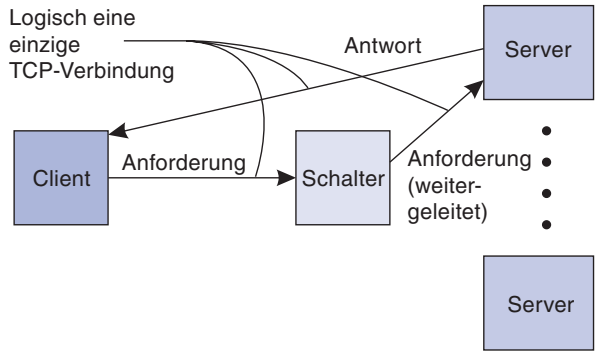


Abbildung 3.13: Das Prinzip des TCP-Handoff

Es lassen sich auch fortgeschrittene Auswahlkriterien für die Server-Auswahl angeben. Nehmen Sie an, dass ein Servercluster verschiedene Dienste anbietet. Falls der Switch die Dienste unterscheiden kann, wenn eine Anforderung eingeht, kann er eine

begründete Entscheidung darüber treffen, an wen er die Anfrage weiterleitet. Die Server-Auswahl kann nach wie vor auf der Transportschicht ablaufen, vorausgesetzt, dass die Dienste nach der Portnummer unterschieden werden. Wenn Sie noch einen Schritt weitergehen, können Sie den Schalter die Nutzlast der eingehenden Anforderungen untersuchen lassen. Diese Methode lässt sich nur anwenden, wenn bekannt ist, wie die Nutzlast aussehen kann. Bei Webservern kann der Schalter eine HTTP-Anforderung erwarten und auf deren Grundlage entscheiden, wer sie verarbeiten soll. Auf diese inhaltsbewusste Anforderungsverteilung kehren wir in *Kapitel 12* bei der Besprechung webbasierter Systeme zurück.

Verteilte Server

Die bis hierher besprochenen Servercluster sind im Allgemeinen eher statisch konfiguriert. Oft enthalten sie einen eigenen Verwaltungscomputer, der die verfügbaren Server nachverfolgt und diese Informationen wie erforderlich an die anderen Geräte, z.B. den Schalter, übergibt.

Wie bereits erwähnt bieten die meisten Servercluster einen einzigen Zugriffspunkt. Wenn dieser Punkt ausfällt, ist der Cluster nicht mehr erreichbar. Um dieses potenzielle Problem zu umgehen, können mehrere Zugriffspunkte bereitgestellt werden, deren Adressen öffentlich verfügbar gemacht werden. So kann der DNS (*Domain Name System*) z.B. mehrere Adressen zurückgeben, die alle zum selben Hostnamen gehören. Dieser Ansatz erfordert immer noch, dass die Clients mehrere Versuche unternehmen, wenn eine der Adressen unerreichbar ist. Außerdem wird dadurch nicht das Problem gelöst, dass ein statischer Zugriffspunkt erforderlich ist.

Stabilität, z.B. in Form eines langlebigen Zugriffspunkts, ist sowohl aus Client- als auch aus Serversicht eine erstrebenswerte Eigenschaft. Andererseits ist es auch gut, bei der Konfiguration eines Serverclusters einschließlich des Schalters einen hohen Grad an Flexibilität zu haben. Dies führte zum Entwurf eines verteilten Servers, der letztlich nichts anderes ist, als ein möglicherweise dynamisch veränderlicher Satz von Computern, bei dem sich zwar auch die Zugriffspunkte ändern können, der aber nach außen hin dennoch als ein einziger, leistungsfähiger Rechner erscheint. Der Entwurf eines solchen verteilten Servers ist bei Szymanik *et al.* (2005) dargestellt. Wir beschreiben ihn hier nur kurz.

Die Grundidee hinter einem verteilten Server besteht darin, dass der Client Nutzen aus einem soliden, hochleistungsfähigen und stabilen Server zieht. Diese Eigenschaften lassen sich oft von modernsten Mainframes erreichen, von denen einige eine mittlere Zeitspanne zwischen Ausfällen von 40 Jahren haben. Durch die transparente Gruppierung von einfacheren Computern zu einem Cluster kann es möglich sein, einen besseren Grad an Stabilität zu erreichen als mit jeder Komponente einzeln. Ein solcher Cluster kann z.B. dynamisch aus Endbenutzercomputern zusammengestellt werden, wie es bei kollaborativen verteilten Systemen der Fall ist.

Konzentrieren wir uns nun darauf, wie in einem solchen System ein stabiler Zugriffspunkt eingerichtet werden kann. Die Grundidee besteht darin, verfügbare Netzwerkdienste zu nutzen, vor allem die Mobilitätsunterstützung für IPv6 (MIPv6). In MIPv6 wird vorausgesetzt, dass ein Mobilitätsknoten ein *Heimatnetzwerk* hat, in dem er sich normalerweise befindet und für das er über eine stabile Adresse verfügt, die sogenannte *Heimatadresse*. An diesem Heimatnetzwerk ist ein besonderer Router angeschlossen, der *Heimatagent*, der sich um den Verkehr zum mobilen Knoten kümmert, wenn dieser nicht anwesend ist. Wenn ein mobiler Knoten an ein fremdes Netz-

werk angeschlossen wird, erhält er zu diesem Zweck eine vorübergehende *Care-of-Adress*, über die er erreicht werden kann. Diese Adresse wird an den Heimatagenten übermittelt, der dann versucht, den gesamten Verkehr an den mobilen Knoten weiterzuleiten. Beachten Sie, dass Anwendungen, die mit dem mobilen Knoten kommunizieren, nur die Adresse im Heimatnetzwerk sehen, aber niemals die Care-of-Adress.

Dieses Prinzip kann für die stabile Adresse eines verteilten Servers verwendet werden. In diesem Fall wird dem Server ursprünglich eine einzige, eindeutige *Kontaktadresse* zugewiesen und diese bleibt während der gesamten Lebensdauer des Servers dessen Adresse für die Kommunikation mit der Außenwelt. Zu jedem Zeitpunkt fungiert irgendein Knoten des verteilten Servers als Zugriffspunkt, der diese Kontaktadresse verwendet, aber diese Rolle kann auf einfache Weise von jedem anderen Knoten übernommen werden. Dabei meldet der Zugriffspunkt seine eigene Adresse als Care-of-Adress an den Heimatagenten des verteilten Servers. Jetzt wird der gesamte Verkehr an den Zugriffspunkt geleitet, der sich dann darum kümmert, die Anforderungen an die zurzeit teilnehmenden Knoten zu verteilen. Wenn der Zugriffspunkt ausfällt, kommt ein einfacher Failover-Mechanismus ins Spiel, durch den ein anderer Zugriffspunkt eine neue Care-of-Adress meldet.

Bei dieser einfachen Konfiguration wären sowohl der Heimatagent als auch der Zugriffspunkt mögliche Engpässe, da der gesamte Verkehr durch diese beiden Computer fließen muss. Dies lässt sich aber durch eine MIPv6-Funktion namens Routenoptimierung verhindern, die wie folgt funktioniert: Wenn ein mobiler Knoten mit der Heimatadresse *HA* seine aktuelle Care-of-Adress *CA* meldet, kann der Heimatagent *CA* an einen Client weiterleiten. Letzterer speichert dann das Paar (*HA*, *CA*) lokal. Von diesem Zeitpunkt an wird die Kommunikation direkt an *CA* weitergeleitet. Zwar kann die Anwendung auf der Client-Seite immer noch die Heimatadresse verwenden, doch übersetzt die zugrunde liegende Unterstützungssoftware für MIPv6 diese Adresse in *CA* und verwendet jene.

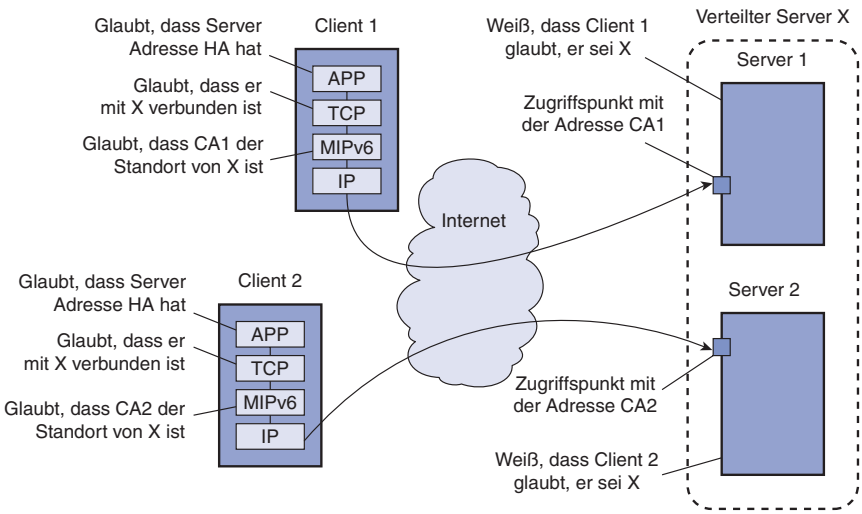


Abbildung 3.14: Routenoptimierung in einem verteilten Server

Mit der Routenoptimierung kann mehreren Clients glaubhaft gemacht werden, dass sie mit einem einzelnen Server kommunizieren, während sie in Wirklichkeit jeweils mit einem anderen Mitgliedknoten des verteilten Servers sprechen, wie ▶Abbildung 3.14 zeigt. Wenn ein Zugriffspunkt eines verteilten Servers eine Anforderung von Client C_1 z.B. an Knoten S_1 (mit der Adresse CA_1) sendet, übergibt er daher genügend Informationen an S_1, damit dieser das Routenoptimierungsverfahren auslösen kann, das dem Client letztendlich glaubhaft macht, die Care-of-Adress sei CA_1. Dadurch kann C_1 das Paar (HA, CA_1) speichern. Während dieses Vorgangs leiten der Zugriffspunkt (sowie der Heimatagent) den meisten Verkehr durch einen Tunnel zwischen C_1 und S_1. Dadurch wird der Heimatagent daran gehindert zu glauben, dass sich die Care-of-Adress geändert hat, weshalb er weiterhin mit dem Zugriffspunkt kommuniziert.

Während die Routenoptimierung stattfindet, können natürlich noch Anforderungen von anderen Clients eingehen. Sie verbleiben in einem ausstehenden Zustand auf dem Zugriffspunkt, bis sie weitergeleitet werden können. Die Anforderung von einem Client C_2 lässt sich dann an den Mitgliedsknoten S_2 (mit der Adresse CA_2) weiterleiten, damit Client C_2 das Paar (HA, CA_2) speichern kann. Als Ergebnis kommunizieren die verschiedenen Clients direkt mit unterschiedlichen Mitgliedern des verteilten Servers, wobei sich jede Client-Anwendung der Illusion hingibt, dass der Server die Adresse HA hat. Der Heimatagent kommuniziert weiterhin mit dem Zugriffspunkt, der mit der Kontaktadresse spricht.

3.4.3 Servercluster verwalten

Ein Servercluster sollte nach außen hin als ein einzelner Computer erscheinen, wie es in der Tat häufig der Fall ist. Bei der Verwaltung des Clusters ändert sich die Situation jedoch dramatisch. Um die Verwaltung von Serverclustern zu vereinfachen, gibt es verschiedene Ansätze, wie wir im Folgenden sehen werden.

Verbreitete Ansätze

Der bei weitem gebräuchlichste Ansatz zur Verwaltung eines Serverclusters besteht darin, die herkömmlichen Verwaltungsfunktionen eines Einzelcomputers auf einen Cluster auszudehnen. In der einfachsten Form bedeutet dies, dass sich ein Administrator von einem entfernten Client aus an einem Knoten anmelden und lokale Verwaltungsbefehle ausführen kann, um Komponenten zu überwachen, zu installieren und zu ändern.

Etwas kniffliger ist es, die Tatsache zu verbergen, dass Sie sich an einem Knoten anmelden müssen, und stattdessen eine Schnittstelle auf einem Verwaltungscomputer bereitstellen, die es erlaubt, Informationen von einem oder mehreren Servern einzuholen, Komponenten zu aktualisieren, Knoten hinzuzufügen und zu entfernen usw. Der Hauptvorteil dieses Ansatzes besteht darin, dass kollektive Operationen für eine ganze Gruppe von Servern einfacher durchgeführt werden können. Diese Form der Verwaltung von Serverclustern wird in der Praxis häufig angewendet, beispielhaft durch Verwaltungssoftware wie Cluster Systems Management von IBM (Hochstetler und Beringer, 2004).

Sobald Cluster die Größe von einigen wenigen Dutzend Knoten überschreiten, ist diese Art der Verwaltung nicht mehr geeignet. Viele Rechenzentren müssen Tausende von Servern verwalten, die in viele Cluster aufgeteilt sind, aber alle zusammenarbeiten. Hierbei mit zentralen Verwaltungsservern zu arbeiten, ist nicht mehr möglich. Außerdem lässt sich sehr einfach zeigen, dass sehr große Cluster eine fortlaufende Reparaturverwaltung (einschließlich Aktualisierungen) benötigen. Einfach gesagt: Wenn p die Wahrscheinlichkeit dafür ist, dass ein Server zurzeit fehlerhaft ist, und wir voraussetzen, dass die Fehler voneinander unabhängig auftreten, dann beträgt die Wahrscheinlichkeit dafür, dass in einem Cluster aus N Servern kein einziger fehlerhaft ist, $(1 - p)^N$. Bei $p = 0{,}001$ und $N = 1000$ besteht nur eine Chance von 36%, dass alle Server korrekt funktionieren.

Es zeigt sich, dass die Unterstützung für sehr große Cluster fast immer von Fall zu Fall erfolgen muss. Zwar gibt es verschiedene Faustregeln (Brewer, 2001), aber keinen systematischen Ansatz zum Umgang mit der Massenverwaltung von Systemen. Die Clusterverwaltung steckt immer noch in den Kinderschuhen, wobei jedoch zu erwarten ist, dass Selbstverwaltungslösungen, wie wir sie im vorhergehenden Kapitel beschrieben haben, letztlich ihren Weg machen werden, nachdem erst genügend Erfahrungen damit gesammelt worden sind.

Fallbeispiel – PlanetLab

PlanetLab

Wir wollen uns nun einen eher ungewöhnlichen Clusterserver genauer anschauen. PlanetLab ist ein kollaboratives verteiltes System, zu dem verschiedene Organisationen jeweils einen oder mehrere Computer spenden, was insgesamt Hunderte von Knoten ergibt. Zusammen formen diese Rechner einen 1-Tier-Servercluster, bei dem Zugriff, Verarbeitung und Speicherung einzeln auf den Knoten stattfinden. Die Verwaltung von PlanetLab ist notwendigerweise nahezu vollkommen verteilt. Bevor wir die Grundprinzipien erklären, möchten wir zunächst die wichtigsten Merkmale der Architektur beschreiben (Peterson *et al.*, 2005).

Für PlanetLab stellt eine Organisation einen oder mehrere Knoten bereit, wobei sich jeder Knoten am einfachsten als einzelner Computer vorstellen lässt, obwohl es sich dabei auch wiederum um einen Cluster handelt kann. Jeder Knoten ist wie in ▶Abbildung 3.15 aufgebaut. Es gibt zwei wichtige Komponenten (Bavier *et al.*, 2004). Die erste ist der Virtual Machine Monitor (VMM), bei dem es sich um ein erweitertes Linux-Betriebssystem handelt. Die Erweiterung besteht hauptsächlich aus Anpassungen, um die zweite Komponente, *vserver*, zu unterstützen. Stellen Sie sich einen (Linux-) vserver am besten als getrennte Umgebung vor, in der eine Gruppe von Prozessen ausgeführt wird. Prozesse in unterschiedlichen vservern sind *vollständig* unabhängig voneinander. Sie können Ressourcen wie Dateien, den Hauptspeicher oder Netzwerkverbindungen nicht direkt gemeinsam nutzen, wie es normalerweise bei Prozessen der Fall ist, die oberhalb eines Betriebssystems ausgeführt werden. Stattdessen bietet ein vserver eine Umgebung, die aus einer eigenen Sammlung von Softwarepaketen, Programmen und Netzwerkeinrichtungen besteht. So kann ein vserver z.B. eine Umgebung bereitstellen, in der ein Prozess Python 1.5.2 zusammen mit einem älteren Apache-Webserver verwenden kann, sagen wir httpd 1.3.1. Im Gegensatz dazu kann ein anderer vserver die jüngste Version von Python und httpd unterstützen. Daher ist die Bezeichnung „Server“ für einen vserver etwas irreführend, da vserver nur Gruppen von Prozessen voneinander isolieren. Wir werden in Kürze wieder auf vserver zurückkommen.

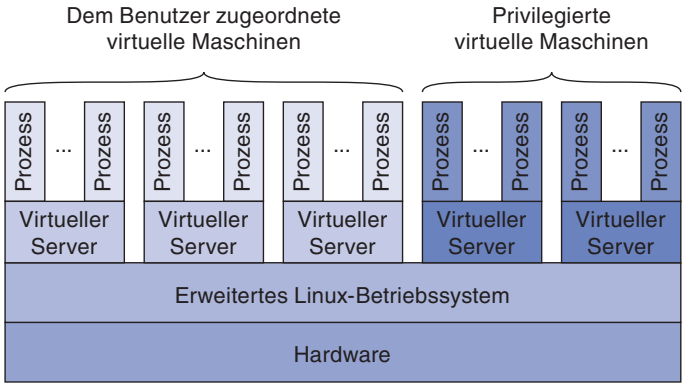


Abbildung 3.15: Der grundlegende Aufbau eines PlanetLab-Knotens

Fallbeispiel – PlanetLab

Der Linux-VMM sorgt dafür, dass die vserver voneinander getrennt werden: Prozesse in unterschiedlichen vservern werden nebenläufig und unabhängig voneinander ausgeführt, wobei jeder nur die Softwarepakete und Programme nutzt, die in seiner eigenen Umgebung zur Verfügung stehen. Die Isolierung der Prozesse in unterschiedlichen vservern ist streng. Zwei Prozesse in unterschiedlichen vservern können z.B. dieselbe Benutzer-ID haben, was aber nicht bedeutet, dass sie vom selben Benutzer stammen. Die Trennung erleichtert deutlich die Unterstützung von Benutzern aus unterschiedlichen Organisationen, die PlanetLab verwenden möchten, z.B. als Testlabor für Experimente mit ganz anderen verteilten Systemen und Anwendungen.

Um solche Experimente zu unterstützen, werden in PlanetLab sogenannte Slices eingeführt. Dabei handelte es sich um einen Satz von vservern, die jeweils auf einem anderen Knoten laufen. Ein Slice kann daher als virtueller Servercluster aus einer Ansammlung von virtuellen Maschinen angesehen werden. Die virtuellen Maschinen in PlanetLab werden oberhalb des Linux-Betriebssystems ausgeführt, das mit einer Reihe von Kernelmodulen erweitert wurde.

Verschiedene Aspekte machen die Verwaltung von PlanetLab zu einem besonderen Problem. Die wichtigsten lauten wie folgt:

1. Die einzelnen Knoten gehören zu unterschiedlichen Organisationen. Jede Organisation sollte festlegen dürfen, wer Anwendungen auf ihren Knoten ausführen darf, und die Ressourcennutzung entsprechend einschränken.

2. Es gibt verschiedene Überwachungswerkzeuge, aber alle setzen eine bestimmte Kombination von Hardware und Software voraus. Außerdem sind sie auf die Verwendung in einer einzigen Organisation zugeschnitten.

3. Programme aus verschiedenen Slices, die auf demselben Knoten laufen, sollten einander nicht stören. Dieses Problem ähnelt dem der Prozessunabhängigkeit in Betriebssystemen.

Schauen wir uns diese Probleme nun genauer an.

Kernstück der Verwaltung von PlanetLab-Ressourcen ist der Knotenmanager. Jeder Knoten verfügt über einen solchen Manager, der durch einen eigenen vserver implementiert wird. Seine einzige Aufgabe besteht darin, andere vserver auf dem von ihm verwalteten Knoten zu erstellen, um die Ressourcenzuweisung zu steuern. Der Knotenmanager trifft keine Richtlinienentscheidungen, sondern dient lediglich als Mechanismus, um die erforderlichen Zutaten bereitzustellen, damit ein Programm auf einem gegebenen Knoten laufen kann.

Die Nachverfolgung von Ressourcen wird durch eine Ressourcenspezifikation (»rspec«) erledigt. Sie legt das Zeitintervall fest, für das bestimmte Ressourcen zugewiesen sind. Zu den Ressourcen gehören der Festplattenplatz, Dateideskriptoren, ein- und ausgehende Netzwerkbandbreite, Endpunkte auf Transportebene, der Hauptspeicher und die CPU-Nutzung. Bezeichnet wird eine Ressourcenspezifikation mit einer global eindeutigen 128-Bit-ID, die als Ressourcenfähigkeit (*Resource Capability, rcap*) bezeichnet wird. Mit einer gegebenen Ressourcenfähigkeit kann der Knotenmanager die zugehörige Ressourcenspezifikation in einer lokalen Tabelle nachschlagen.

Fallbeispiel – PlanetLab

Ressourcen sind an Slices gebunden. Mit anderen Worten, um Ressourcen nutzen zu können, müssen Sie einen Slice erstellen. Jeder Slice ist mit einem *Dienstprovider* verbunden, den Sie sich am besten als eine Entität mit einem Konto für PlanetLab vorstellen können. Jeder Slice kann dann durch ein (*Haupt-ID*, *Slice-Tag*)-Paar bezeichnet werden, wobei die Haupt-ID den Provider angibt und das Slice-Tag ein vom Provider gewählter Bezeichner ist.

Um einen neuen Slice zu erstellen, muss jeder Knoten einen *Slice-Erstellungsdienst* (*Slice Creation Service*, *SCS*) ausführen, der wiederum Verbindung mit dem Knotenmanager aufnimmt, um ihn aufzufordern, einen vserver anzulegen und Ressourcen zuzuweisen. Der Knotenmanager selbst kann über das Netzwerk nicht erreicht werden, damit er sich nur auf die lokale Ressourcenverwaltung konzentriert. Im Gegenzug akzeptiert der SCS keine Anforderungen zur Slice-Erstellung von jedermann. Nur bestimmte Slice-Autoritäten dürfen das Erstellen eines Slice anfordern. Jede Slice-Autorität hat Zugriffsrechte für einen Satz von Knoten. Im einfachsten Fall gibt es nur eine einzige Slice-Autorität, die Anforderungen für die Slice-Erstellung für alle Knoten stellen darf.

Als letzter Mosaikstein nimmt ein Dienstprovider Kontakt mit einer Slice-Autorität auf und fordert die Erstellung eines Slice über mehrere Knoten hinweg an. Der Provider ist der Autorität bekannt, z.B. da er schon zuvor authentifiziert worden ist und anschließend als PlanetLab-Benutzer registriert wurde. In der Praxis nehmen PlanetLab-Benutzer über einen webbasierten Dienst Kontakt mit einer Slice-Autorität auf. Weitere Einzelheiten finden Sie bei Chun und Spalink (2003).

Dieses Verfahren zeigt, dass die Verwaltung von PlanetLab über Zwischenträger erfolgt. Eine wichtige Klasse solcher Zwischenträger bilden die Slice-Autoritäten, die auf den Knoten die Berechtigung erworben haben, Slices zu erstellen. Der Erwerb dieser Rechte erfolgt außerhalb des technischen Systems, hauptsächlich durch Kontaktaufnahme mit den Systemadministratoren an den verschiedenen Standorten. Offensichtlich handelt es sich hierbei um einen zeitraubenden Vorgang, der nicht von Endbenutzern durchgeführt werden kann (bzw. Dienstprovider in der Terminologie von PlanetLab).

Neben Slice- gibt es auch Verwaltungsautoritäten. Während sich eine Slice-Autorität nur auf die Verwaltung von Slices konzentriert, ist eine Verwaltungsautorität dafür zuständig, ein Auge auf alle Knoten zu halten. Vor allem soll sie sicherstellen, dass die von ihr betreuten Knoten die grundlegende PlanetLab-Software ausführen und sich an die PlanetLab-Regeln halten. Die Dienstprovider vertrauen darauf, dass die Verwaltungsautorität ihnen Knoten bereitstellt, die sich ordnungsgemäß verhalten.

Fallbeispiel – PlanetLab

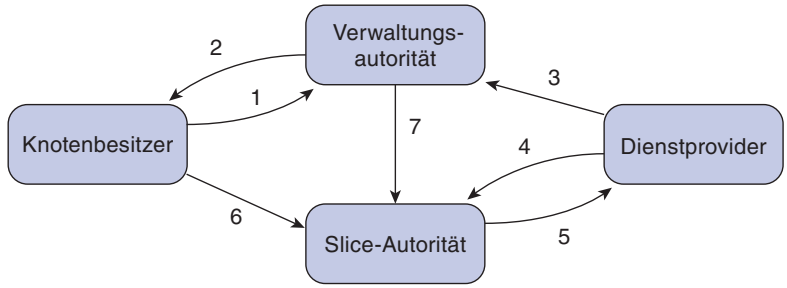


Abbildung 3.16: Verwaltungsbeziehungen zwischen den verschiedenen PlanetLab-Entitäten

Dieser Aufbau führt zu der Verwaltungsstruktur aus ▶Abbildung 3.16, die von Peterson *et al.* (2005) in Form von Vertrauensbeziehungen beschrieben wird. Es gibt dabei folgende Beziehungen:

1. Der Besitzer unterstellt seinen Knoten einer Verwaltungsautorität, wobei er die Nutzung gegebenenfalls einschränkt.
2. Eine Verwaltungsautorität stellt die notwendige Software zur Verfügung, um einen Knoten zu PlanetLab hinzuzufügen.
3. Ein Dienstprovider registriert sich bei einer Verwaltungsautorität und vertraut darauf, dass sie ordnungsgemäß funktionierende Knoten bereitstellt.
4. Ein Dienstprovider nimmt Verbindung mit einer Slice-Autorität auf, um einen Slice über mehrere Knoten hinweg zu erstellen.
5. Die Slice-Autorität muss den Dienstprovider authentifizieren.
6. Ein Knotenbesitzer bietet einer Slice-Autorität einen Slice-Erstellungsdienst an. Im Wesentlichen delegiert er die Ressourcenverwaltung an die Slice-Autorität.
7. Eine Verwaltungsautorität delegiert die Erstellung von Slices an eine Slice-Autorität.

Diese Beziehungen decken die gesteuerte Delegierung von Knoten so ab, dass sich jeder Knotenbesitzer auf eine annehmbare und sichere Verwaltung verlassen kann. Das zweite zu lösende Problem betrifft die Überwachung. Gebraucht wird ein einheitlicher Ansatz, mit dem Benutzer sehen können, wie sich ihre Programme innerhalb eines bestimmten Slice verhalten.

Fallbeispiel – PlanetLab

PlanetLab folgt einem einfachen Ansatz. Jeder Knoten wird mit einer Reihe von Sensoren ausgestattet, die jeweils Angaben wie die CPU-Nutzung, die Festplattenaktivität usw. weitermelden können. Diese Sensoren können beliebig komplex sein, wichtig ist aber, dass sie ihre Informationen stets knotenweise weitergeben. Diese Informationen werden dann über einen Webserver zugänglich gemacht: Jeder Sensor ist über einfache HTTP-Anforderungen zugänglich (Bavier *et al.*, 2004).

Zugegebenermaßen ist dieser Ansatz zur Überwachung immer noch sehr einfach, er kann aber als Grundlage für verbesserte Verfahren angesehen werden. So gibt es z.B. im Prinzip keinen Grund dafür, Astrolabe (besprochen in *Kapitel 2*) nicht für die aggregierte Sensorablesung über mehrere Knoten hinweg einzusetzen.

Was unser drittes Verwaltungsproblem betrifft, den Schutz von Programmen voreinander, so verwendet PlanetLab virtuelle Linux-Server (vserver) zur Isolierung der Slices. Wie bereits erwähnt, besteht die Grundidee eines vserver darin, Anwendungen in ihrer eigenen Umgebung auszuführen, zu der alle Dateien gehören, die normalerweise auf einem einzelnen Computer gemeinsam verwendet werden. Eine solche Trennung lässt sich relativ einfach mithilfe des UNIX-Befehls `chroot` erreichen, der den Stamm des Dateisystems ändert, sodass er sich nicht mehr dort befindet, wo Anwendungen nach Dateien suchen. Nur der Superuser kann `chroot` ausführen.

Natürlich sind noch weitere Mosaiksteine erforderlich. Virtuelle Linux-Server trennen nicht nur Dateisysteme, sondern auch normalerweise gemeinsam genutzte Informationen über Prozesse, Netzwerkadressen, die Speichernutzung usw. Daraus folgt, dass ein physischer Rechner in mehrere Einheiten aufgeteilt wird, die jeweils einer voll ausgestatteten Linux-Umgebung entsprechen, aber von den anderen Teilen getrennt sind. Einen Überblick über virtuelle Linux-Server finden Sie bei Potzl *et al.* (2005).

3.5 Codemigration

Bis jetzt haben wir uns hauptsächlich mit verteilten Systemen beschäftigt, in denen sich die Kommunikation auf die Übergabe von Daten beschränkte. Es gibt jedoch auch Situationen, in denen Programme übergeben werden müssen, manchmal sogar während ihrer Ausführung, um den Entwurf eines verteilten Systems zu vereinfachen. In diesem Abschnitt werfen wir einen genaueren Blick darauf, was Codemigration eigentlich ist. Wir beginnen damit, die verschiedenen Ansätze zur Codemigration zu untersuchen, und besprechen anschließend den Umgang mit den lokalen Ressourcen, die von den migrierten Programmen verwendet werden. Eine besonders harte Nuss ist die Migration von Code in heterogenen Systemen, mit der wir uns ebenfalls beschäftigen.

3.5.1 Ansätze zur Codemigration

Bevor wir uns die verschiedenen Formen der Codemigration ansehen, wollen wir zunächst untersuchen, warum die Migration von Code sinnvoll ist.

Gründe für die Codemigration

Traditionell erfolgte die Codemigration in verteilten Systemen als Prozessmigration, bei der ein ganzer Prozess von einem Computer auf einen anderen verschoben wurde (Milojicic *et al.*, 2000). Die Übertragung eines laufenden Prozesses von einem Rechner zu einem anderen ist eine teure und knifflige Angelegenheit, die man lieber nur dann ausführen sollte, wenn man einen guten Grund dafür hat. Dieser Grund war immer die Leistungssteigerung. Die Grundidee besteht darin, dass die Gesamtleistung des Systems gesteigert werden kann, wenn Prozesse von stark auf schwach ausgelastete Computer verschoben werden. Die Last wird häufig in Form der Länge der CPU-Warteschlange oder der CPU-Nutzung ausgedrückt, wobei jedoch auch andere Leistungsmaße Verwendung finden.

Algorithmen zur Lastverteilung, die Entscheidungen über die Zuordnung und Neuverteilung von Aufgaben in einem Satz von Prozessoren treffen, spielen in rechenintensiven Systemen eine wichtige Rolle. In vielen modernen verteilten Systemen ist die Optimierung der Rechenkapazität jedoch weniger von Bedeutung als z.B. die Minimierung der Kommunikation. Darüber hinaus beruht die Leistungsverbesserung durch Codemigration aufgrund der heterogenen Natur der zugrunde liegenden Plattformen und Computernetzwerke häufiger auf qualitativen Überlegungen als auf mathematischen Modellen.

Stellen Sie sich z.B. ein Client-Server-System vor, in dem der Server eine umfangreiche Datenbank verwaltet. Wenn eine Client-Anwendung viele Datenbankoperationen mit großen Datenmengen durchführen muss, kann es besser sein, einen Teil der Client-Anwendung auf den Server zu verlagern und nur die Ergebnisse über das Netzwerk zu übertragen, da das Netzwerk anderenfalls mit der Datenübertragung vom Server zum Client überflutet würde. In diesem Fall beruht die Codemigration auf der Annahme, dass es allgemein sinnvoll ist, Daten nahe an dem Ort zu verarbeiten, wo sie sich befinden.

Derselbe Grund lässt sich auch für die Migration von Teilen des Servers zum Client vorbringen. In vielen interaktiven Datenbankanwendungen muss der Client z.B. Formulare ausfüllen, die anschließend in eine Folge von Datenbankoperationen übersetzt werden. Die Formulare auf der Client-Seite zu verarbeiten und nur das Ergebnis an den Server zu senden, kann manchmal eine relativ große Anzahl an kleinen Nachrichten

verhindern, die über das Netzwerk gesendet werden. Dadurch erzielt der Client eine bessere Leistung, während der Server gleichzeitig weniger Zeit auf Formularverarbeitung und Kommunikation verwenden muss.

Die Unterstützung für die Codemigration kann die Leistung auch durch die Ausnutzung der Parallelverarbeitung verbessern, aber ohne die gewöhnlich mit der Parallelprogrammierung verbundenen Schwierigkeiten. Ein typisches Beispiel dafür ist die Suche nach Informationen im Web. Es ist relativ einfach, eine Suchabfrage in der Form eines kleinen mobilen Programms zu implementieren, eines sogenannten mobilen Agenten, der sich von Site zu Site bewegt. Wenn Sie verschiedene Kopien eines solchen Programms machen und zu verschiedenen Sites senden, lässt sich eine lineare Beschleunigung im Vergleich mit der Verwendung einer einzigen Programminstanz erreichen.

Neben der Leistungssteigerung gibt es noch andere Gründe für die Unterstützung der Codemigration. Der wichtigste ist dabei die Flexibilität. Der herkömmliche Ansatz zum Aufbau verteilter Anwendungen besteht darin, die Anwendung in verschiedene Teile aufzugliedern und im voraus zu entscheiden, wo die einzelnen Bestandteile ausgeführt werden sollen. Dies hat z.B. zu den verschiedenen Multi-Tier-Client-Server-Anwendungen aus *Kapitel 2* geführt.

Wenn Code von einem Computer auf einen anderen verschoben werden kann, ist es jedoch auch möglich, verteilte Systeme dynamisch zu konfigurieren. Stellen Sie sich z.B. einen Server vor, der eine standardisierte Schnittstelle zu einem Dateisystem implementiert. Damit entfernte Clients auf das Dateisystem zugreifen können, verwendet der Server ein proprietäres Protokoll. Normalerweise müsste die clientseitige Implementierung der Dateisystemschnittstelle, die auf dem Protokoll basiert, mit der Client-Anwendung verknüpft sein. Dieser Ansatz erfordert es, dass die Software schon bei der Entwicklung der Client-Anwendung für den Client zur Verfügung steht.

Eine Alternative besteht darin, dass der Server die Client-Implementierung nicht früher bereitstellt, als unbedingt nötig, also erst dann, wenn der Client an den Server gebunden wird. Zu diesem Zeitpunkt lädt der Client die Implementierung dynamisch herunter, führt die notwendigen Initialisierungsschritte durch und ruft anschließend den Server auf. Dieses Prinzip wird in ▶Abbildung 3.17 dargestellt. Dieses Modell der dynamischen Verschiebung von Code von einem entfernten Standort erfordert ein standardisiertes Protokoll für das Herunterladen und Initialisieren des Codes. Außerdem ist es notwendig, dass der heruntergeladene Code auf dem Client-Computer ausgeführt werden kann. Andere Lösungen werden weiter hinten und in späteren Kapiteln vorgeführt.

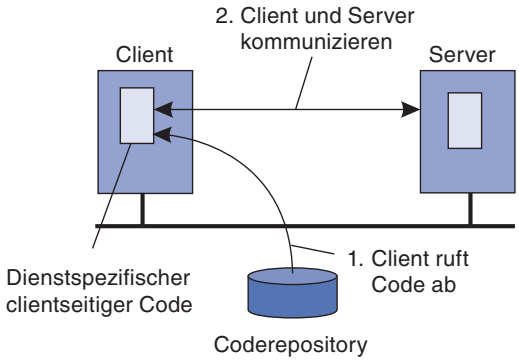


Abbildung 3.17: Das Prinzip der dynamischen Konfiguration eines Clients zur Kommunikation mit dem Server. Der Client holt sich die notwendige Software und ruft dann den Server auf.

Der wichtige Vorteil dieses Modells, bei dem die clientseitige Software dynamisch heruntergeladen wird, besteht darin, dass die Clients nicht bereits die gesamte Software installiert haben müssen, um mit dem Server zu reden. Stattdessen kann die Software bei Bedarf hinzugefügt und ebenso wieder entfernt werden, wenn sie nicht mehr gebraucht wird. Ein weiterer Vorteil ist die Tatsache, dass wir das Client-Server-Protokoll und seine Implementierung so oft austauschen können, wie wir wollen, sofern die Schnittstellen genormt sind. Die Änderungen wirken sich nicht auf bestehende Client-Anwendungen auf, die sich auf den Server stützen. Es gibt natürlich auch Nachteile. Der wichtigste von ihnen, den wir in *Kapitel 9* besprechen, betrifft die Sicherheit. Blind darauf zu vertrauen, dass der heruntergeladene Code nur die angekündigte Schnittstelle implementiert, während er auf Ihre ungeschützte Festplatte zugreift, und die pikantesten Daten wer weiß wohin sendet, ist keine sonderlich gute Idee.

Modelle für die Codemigration

Obwohl »Codemigration« suggeriert, dass wir Code nur zwischen Computern übertragen, beschreibt dieser Begriff doch einen viel weiteren Bereich. Traditionell geht es bei der Kommunikation in verteilten Systemen um den Austausch von Daten zwischen Prozessen. Codemigration im weitesten Sinne ist also die Verschiebung von Programmen von einem Computer zum anderen, damit diese Programme auf dem Ziel ausgeführt werden. In manchen Fällen, etwa bei der Prozessmigration, müssen auch der Status des Programms, ausstehende Signale und andere Teile der Umgebung verschoben werden.

Um die verschiedenen Modelle für die Codemigration besser zu verstehen, verwenden wir ein von Fuggetta *et al.* (1998) beschriebenes Framework. Darin besteht ein Prozess aus drei Segmenten. Das Codesegment ist der Teil, der den Befehlssatz enthält, aus dem sich das auszuführende Programm zusammensetzt. Das Ressourcensegment dagegen enthält Referenzen auf externe Ressourcen, die von dem Prozess gebraucht werden, z.B. Dateien, Drucker, Geräte, andere Prozesse usw. Schließlich wird noch ein Ausführungssegment verwendet, um den aktuellen Ausführungszustand eines Prozesses zu speichern. Es besteht aus privaten Daten, dem Stapel (Stack) und natürlich dem Programmzähler.

Für die Codemigration ist es mindestens erforderlich, die *schwache Mobilität* bereitzustellen. In diesem Modell ist es möglich, nur das Codesegment und vielleicht einige Initialisierungsdaten zu übertragen. Ein charakteristisches Merkmal der schwachen Mobilität besteht darin, dass ein übertragenes Programm aus in einer von mehreren vordefinierten Positionen heraus gestartet wird. Das ist genau das, was z.B. bei Java-Applets geschieht, die immer von Anfang an starten. Der Vorteil dieses Ansatzes liegt in seiner Einfachheit. Schwache Mobilität erfordert lediglich, dass der Zielcomputer den Code ausführen kann, was letztlich nur bedeutet, den Code portierbar zu machen. Bei der Erörterung der Migration in heterogenen Systemen werden wir auf diese Punkte zurückkommen.

Im Gegensatz dazu wird bei Systemen mit *starker Mobilität* auch das Ausführungssegment übertragen. Das charakteristische Merkmal besteht darin, dass ein laufender Prozess angehalten, auf einen anderen Computer verschoben und an der gleichen Stelle wieder fortgesetzt werden kann. Die starke Mobilität ist eindeutig universeller einsetzbar, aber auch schwieriger zu implementieren.

Unabhängig von der Stärke der Mobilität lässt sich auch eine Unterscheidung zwischen sender- und empfängerinitiierter Migration durchführen. Die *senderinitiierte Migration* wird auf dem Computer ausgelöst, auf dem sich der Code zurzeit befindet oder ausgeführt wird. Sie wird gewöhnlich beim Hochladen von Programmen zu einem Server ausgeführt. Ein weiteres Beispiel ist das Senden von Suchprogrammen über das Internet, um einen Datenbankserver im Web abzufragen. Bei der *empfängerinitiierten Migration* geht die Initiative für die Codemigration dagegen vom Zielcomputer aus. Ein Beispiel dafür stellen Java-Applets dar.

Die empfängerinitiierte Migration ist einfacher als die senderinitiierte. In vielen Fällen tritt die Codemigration zwischen einem Client und einem Server auf, wobei der Client die Initiative übernimmt. Das sichere Hochladen von Code zu einem Server wie bei der senderinitiierten Migration erfordert oft, dass der Client zuvor beim Server registriert und authentifiziert worden ist. Mit anderen Worten, der Server muss all seine Clients kennen, da der Client möglicherweise Zugriff auf Server-Ressourcen wie dessen Festplatte anfordert. Der Schutz solcher Ressourcen ist von entscheidender Bedeutung. Im Gegensatz dazu kann der empfängerinitiierte Download von Code häufig anonym erledigt werden. Darüber hinaus ist ein Server im Allgemeinen nicht an den Ressourcen des Clients interessiert. Die Codemigration zum Client wird nur durchgeführt, um die Leistung auf der Client-Seite zu verbessern. Dazu muss nur eine begrenzte Anzahl von Ressourcen geschützt werden, wie Speicher und Netzwerkverbindungen. Die sichere Codemigration besprechen wir ausführlich in *Kapitel 9*.

Bei der schwachen Mobilität macht es auch einen Unterschied aus, ob der migrierte Code durch einen Zielprozess ausgeführt oder ob ein eigener Prozess gestartet wird. Java-Applets werden z.B. einfach auf einen Webbrowser heruntergeladen und dann im Adressraum des Browsers ausgeführt. Der Vorteil dieses Ansatzes liegt darin, dass es nicht notwendig ist, einen eigenen Prozess zu starten, wodurch eine Kommunikation mit dem Zielcomputer vermieden wird. Als Hauptnachteil muss der Zielprozess gegen böswilliges oder unbeabsichtigtes Ausführen von Code geschützt werden. Eine einfache Lösung besteht darin, dass sich das Betriebssystem darum kümmert, indem es einen eigenen Prozess zum Ausführen des migrierten Codes erstellt. Beachten Sie, dass diese Lösung das zuvor erwähnte Problem des Ressourcenzugriffs nicht angeht. Hierzu sind andere Maßnahmen erforderlich.

Anstatt einen laufenden Prozess zu migrieren, was als Prozessmigration bezeichnet wird, kann die starke Mobilität auch durch entferntes Klonen unterstützt werden. Im Gegensatz zur Prozessmigration ergibt das Klonen eine genaue Kopie des ursprünglichen Prozesses, die aber auf einem anderen Computer läuft. Der geklonte Prozess wird parallel zum ursprünglichen ausgeführt. In UNIX-Systemen erfolgt entferntes Klonen durch Abzweigen eines Kindprozesses, der dann auf einem entfernten Computer fortgesetzt wird. Der Vorteil beim Klonen liegt darin, dass das Modell sehr stark demjenigen ähnelt, das bereits in vielen Anwendungen verwendet wird. Als einziger Unterschied wird der geklonte Prozess auf einem anderen Computer ausgeführt. In diesem Sinne ist die Migration durch Klonen eine einfache Möglichkeit, um die Verteilungstransparenz zu erhöhen.

Die verschiedenen Alternativen für die Codemigration sind in ▶Abbildung 3.18 zusammengefasst.

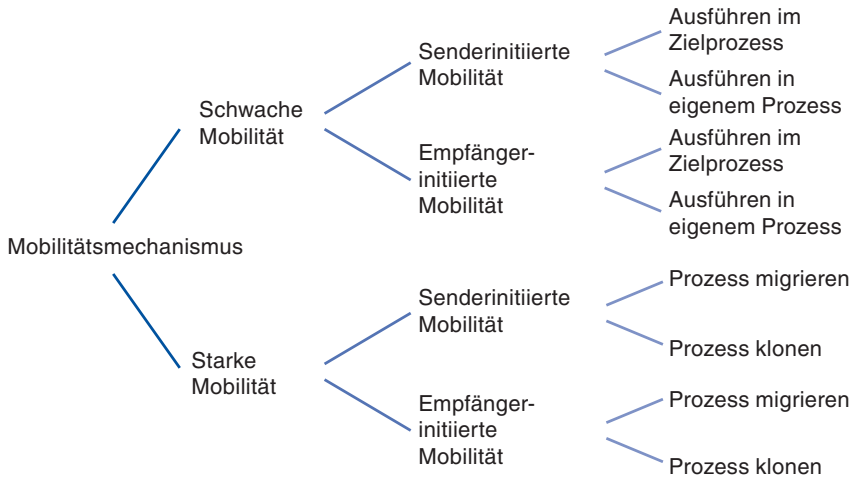


Abbildung 3.18: Möglichkeiten der Codemigration

3.5.2 Migration und lokale Ressourcen

Bis jetzt haben wir uns nur um die Migration des Code- und des Ausführungssegments gekümmert. Das Ressourcensegment bedarf besonderer Beachtung. Was die Codemigration oft so schwierig macht, ist die Tatsache, dass das Ressourcensegment nicht immer einfach ohne Änderungen zusammen mit den anderen Segmenten übertragen werden kann. Stellen Sie sich z.B. eine Prozess mit einer Referenz auf einen TCP-Port vor, durch den er mit anderen (entfernten) Prozessen kommuniziert. Eine solche Referenz wird im Ressourcensegment festgehalten. Wenn der Prozess an einen anderen Ort verschoben wird, muss er den Port aufgeben und auf dem Ziel einen neuen beantragen. In anderen Fällen jedoch führt die Übertragung von Ressourcen zu keinerlei Problemen. Eine Referenz auf eine Datei in Form eines absoluten URL bleibt unabhängig von dem Computer gültig, auf dem sich der Prozess mit diesem Verweis befindet.

Um die Auswirkungen der Codemigration auf das Ressourcensegment verständlich zu machen, unterscheiden Fuggetta *et al.* (1998) drei Typen von Prozess-Ressourcen-Bindungen. Die stärkste Bindung tritt auf, wenn ein Prozess auf eine Ressource über deren Bezeichner verweist. In diesem Fall braucht der Prozess genau diese Ressource und keine andere. Ein Beispiel einer solchen Bindung durch Bezeichner ist ein Prozess, der mit einem URL auf eine bestimmte Website oder einen FTP-Server verweist. Mit der gleichen Begründung stellen auch Referenzen auf lokale Kommunikationsendpunkte eine Bindung durch Bezeichner dar.

Eine schwächere Form der Prozess-Ressourcen-Bindung besteht, wenn nur der Wert einer Ressource benötigt wird. In diesem Fall wird die Ausführung des Prozesses nicht beeinträchtigt, wenn eine andere Ressource denselben Wert liefert. Ein typisches Beispiel dieser Bindung durch Wert ist ein Programm, das sich auf Standardbibliotheken stützt, wie sie bei der Programmierung in C und Java verwendet werden. Solche Bibliotheken sollten immer lokal verfügbar sein, aber ihr genauer Speicherort im lokalen Dateisystem kann sich von Site zu Site unterscheiden. Für die korrekte Ausführung der Prozesse sind aber nicht die Dateien, sondern ihr Inhalt wichtig.

Die schwächste Form der Bindung liegt schließlich vor, wenn ein Prozess nur eine Ressource eines bestimmten Typs benötigt. Diese Bindung durch Typ zeigt sich z.B. bei Referenzen auf lokale Geräte, wie Monitore, Drucker usw.

Bei der Codemigration müssen wir oft die Referenzen auf Ressourcen austauschen, ohne aber die Art der Bindung ändern zu können. Falls und wie eine Referenz geändert werden muss, hängt davon ab, ob die Ressource zusammen mit dem Code auf den Zielcomputer verschoben werden kann. Genauer gesagt, müssen wir die Ressourcen-Computer-Bindungen bedenken und dabei die folgenden Fälle unterscheiden. *Ungebundene Ressourcen* können einfach zwischen verschiedenen Computern verschoben werden. Dabei handelt es sich gewöhnlich um (Daten-)Dateien, die nur mit dem Programm verknüpft sind, das migriert werden soll. Dagegen ist das Verschieben oder Kopieren von *Gebundene Ressourcen* zwar möglich, aber nur zu relativ hohen Kosten. Typische Beispiele dafür sind lokale Datenbanken und ganze Websites. Solche Ressourcen sind zwar theoretisch nicht vom jeweiligen Computer abhängig, aber dennoch ist es meistens nicht möglich, sie in eine andere Umgebung zu verschieben. *Fixe Ressourcen* sind schließlich eng an einen bestimmten Computer oder eine Umgebung gebunden und können nicht verschoben werden. Oft handelt es sich dabei um lokale Geräte. Ein weiteres Beispiel ist ein lokaler Kommunikationsendpunkt.

Die drei Typen der Prozess-Ressourcen-Bindung und die drei Typen der Ressourcen-Computer-Bindung ergeben zusammen neun Kombinationen, die wir bei der Codemigration bedenken müssen. Eine Übersicht finden Sie in ▶Abbildung 3.19.

Betrachten wir als Erstes die Möglichkeiten, die sich bei der Prozess-Ressourcen-Bindung über einen Bezeichner ergeben. Ist die Ressource lose, wird sie am besten zusammen mit dem zu migrierenden Code verschoben. Falls die Ressource aber auch von anderen Prozessen verwendet wird, kann alternativ eine globale Referenz erstellt werden, die die Grenzen des Computers überschreitet. Ein Beispiel dafür ist ein URL. Bei befestigten oder fixen Ressourcen besteht die beste Lösung darin, eine globale Referenz anzulegen.

		Ressourcen-Computer-Bindung		
		Lose	Befestigt	Fix
Prozess-Ressourcen-Bindung	Durch Bezeichner	V (oder GR)	GR (oder V)	GR
	Durch Wert	WK (oder V, GR)	GR (oder WK)	GR
	Durch Typ	NB (oder V, WK)	NB (oder GR, WK)	NB (oder GR)

GR	Einrichten einer globalen, systemweiten Referenz
V	Verschieben der Ressource
WK	Kopieren des Wertes der Ressource
NB	Neubinden des Prozesses an eine lokal verfügbare Ressource

Abbildung 3.19: Maßnahmen für Referenzen auf lokale Ressourcen bei der Codemigration

Machen Sie sich klar, dass das Einrichten einer globalen Referenz mehr erfordert, als lediglich URLs zu verwenden, und dass dies zuweilen zu unvertretbaren Kosten führt. Stellen Sie sich z.B. ein Programm vor, dass qualitativ hochwertige Bilder für eine dedizierte Multimedia-Arbeitsstation erstellt. Das Erstellen hochwertiger Bilder in Echtzeit ist eine rechenintensive Aufgabe, weshalb das Programm auf einen hochleistungsfähigen Server verschoben werden kann. Das Einrichten einer globalen Referenz auf die Multimedia-Arbeitsstation bedeutet, einen Kommunikationspfad zwischen

Server und Arbeitsstation einzurichten. Außerdem ist sowohl auf dem Server als auch auf der Arbeitsstation eine signifikante Verarbeitungsleistung erforderlich, um die Bandbreitenanforderungen für die Übertragung der Bilder zu erfüllen. Unter dem Strich mag sich zeigen, dass das Verschieben des Programms auf den Server allein aufgrund der hohen Kosten für die globale Referenz keine so gute Idee ist.

Ein weiteres Beispiel dafür, dass das Einrichten einer globalen Referenz nicht immer einfach ist, stellt die Migration eines Prozesses dar, der lokale Kommunikationsendpunkte verwendet. In diesem Fall haben wir es mit einer fixen Ressource zu tun, an die der Prozess durch einen Bezeichner gebunden ist. Hierfür gibt es zwei grundlegende Lösungen. Eine besteht darin, dass der Prozess nach der Migration eine Verbindung zum Quellcomputer aufbaut und ein weiterer Prozess dort alle eingehenden Nachrichten einfach weiterleitet. Der Hauptnachteil dabei ist, dass bei einer Fehlfunktion des Quellcomputers die Kommunikation mit dem migrierten Prozess ausfallen kann. Bei der alternativen Lösung müssen alle Prozesse, die mit dem migrierten Prozess kommuniziert haben, ihre *eigenen* globalen Referenzen ändern und Nachrichten an den neuen Kommunikationsendpunkt auf dem Zielcomputer senden.

Bei der Bindung durch Wert liegt eine andere Situation vor. Betrachten wir zunächst eine fixe Ressource. Eine solche Kombination tritt z.B. auf, wenn ein Prozess davon ausgeht, dass sich mehrere Prozesse den Speicher teilen können. Das Einrichten einer globalen Referenz bedeutet in diesem Fall, dass wir eine verteilte Form des gemeinsam genutzten Speichers implementieren. In vielen Fällen ist das keine stabile oder effiziente Lösung.

Bei *befestigten Ressourcen*, auf die über ihren Wert verwiesen wird, handelt es sich gewöhnlich um Laufzeitbibliotheken. Normalerweise sind Kopien solcher Ressourcen auf dem Zielcomputer verfügbar; anderenfalls müssen sie vor der Codemigration kopiert werden. Wenn dabei große Mengen von Daten kopiert werden müssen, z.B. Wörterbücher und Thesauri in Textverarbeitungen, stellt die Einrichtung einer globalen Referenz die bessere Alternative dar.

Der einfachste Fall liegt bei *losen Ressourcen* vor. Die beste Lösung besteht darin, sie an den neuen Bestimmungsort zu kopieren (oder zu verschieben), sofern sie nicht von anderen Prozessen genutzt werden. In letzterem Fall bleibt nur die Möglichkeit, eine globale Referenz einzurichten.

Der letzte Fall ist die *Bindung nach Typ*. Unabhängig von der Ressourcen-Computer-Bindung liegt die offensichtliche Lösung darin, den Prozess neu an lokal verfügbare Ressourcen desselben Typs zu binden. Nur wenn solche Ressourcen nicht vorhanden sind, ist es nötig, die ursprüngliche Ressource an den Zielort zu kopieren oder zu verschieben oder eine globale Referenz zu verwenden.

3.5.3 Migration in heterogenen Systemen

Bis jetzt setzten wir stillschweigend voraus, dass der migrierte Code auf dem Zielcomputer problemlos ausgeführt werden kann. Diese Annahme ist in homogenen Systemen gerechtfertigt. Im Allgemeinen werden verteilte Systeme jedoch auf einer Zusammenstellung verschiedener Plattformen aufgebaut, die jeweils ihr eigenes Betriebssystem und ihre eigene Rechnerarchitektur aufweisen. In solchen Systemen erfordert die Migration, dass jede dieser Plattformen unterstützt wird, das also das Codesegment auf allen Plattformen ausgeführt werden kann. Außerdem müssen wir sicherstellen, dass das Ausführungssegment auf allen Plattformen korrekt dargestellt werden kann.

Die Probleme mit heterogenen Systemen sind in vieler Hinsicht identisch mit denen der Portabilität, weshalb es nicht überrascht, dass sich auch die Lösungen gleichen. Ende der 1970er Jahre bestand z.B. eine einfache Lösung zur Behebung vieler Probleme mit der Portierung von Pascal auf verschiedene Computer darin, maschinenunabhängigen Zwischencode für eine abstrakte virtuelle Maschine zu erstellen (Barron, 1981). Diese Maschine musste natürlich auf verschiedenen Plattformen implementiert werden, ermöglichte es dann aber, Pascal-Programme überall auszuführen. Diese einfache Idee wurde zwar mehrere Jahre lang häufig genutzt, entwickelte sich aber nie zu einer allgemeinen Lösung für Portabilitätsprobleme anderer Sprachen, vor allem C.

Ungefähr 25 Jahre später wird der Stellenwert der Codemigration in heterogenen Systemen durch Skriptsprachen und hochgradig portierbare Sprachen wie Java aufgegriffen. Im Wesentlichen folgen diese Lösungen dem für die Portierung von Pascal verwendeten Ansatz. Alle stützen sich auf eine virtuelle (Prozess-)Maschine, die entweder unmittelbar den Quellcode interpretiert (wie es bei den Skriptsprachen der Fall ist) oder einen vom Compiler generierten Zwischencode (Java). Für Sprachentwickler ist es wichtig, sich zum richtigen Zeitpunkt am richtigen Ort zu befinden.
Die jüngsten Entwicklungen weichten die Abhängigkeit von Programmiersprachen auf. Vor allem wurden Lösungen entwickelt, die nicht nur Prozesse migrieren, sondern ganze Computerumgebungen. Die Grundidee besteht darin, die gesamte Umgebung aufzuteilen und den Prozessen in einem Teil jeweils eine eigene Sicht der Umgebung anzubieten.

Wenn die Aufteilung korrekt erfolgt, ist es möglich, einen Teil vom zugrunde liegenden System zu entkoppeln und tatsächlich zu einem anderen Computer zu migrieren. Auf diese Weise bietet die Migration eine Form von starker Mobilität für Prozesse, da sie an einem beliebigen Punkt der Ausführung verschoben werden und anschließend am selben Punkt fortfahren können. Darüber hinaus können die Probleme gelöst werden, die durch die Bindung von Prozessen an lokale Ressourcen auftreten, da diese Bindungen in vielen Fällen einfach erhalten bleiben. Namentlich die lokalen Ressourcen sind oft Teil der Umgebung, die migriert wird.

Betrachten wir ein Beispiel der Migration virtueller Maschinen, das bei Clark *et al.* (2005) erörtert wird. In diesem Fall konzentrierten sich die Autoren auf die Echtzeitmigration eines virtualisierten Betriebssystems. Ein solches Vorgehen ist typisch für Servercluster, in denen eine enge Kopplung über ein einzelnes, gemeinsam genutztes LAN errichtet wird. Unter diesen Umständen treten bei der Migration zwei Hauptprobleme auf: die Migration des gesamten Speicherabbilds und der Bindungen an lokale Ressourcen.

Für das erste Problem gibt es im Prinzip drei Lösungsmöglichkeiten (die auch kombiniert werden können):

1. Sie verschieben die Speicherseiten auf den neuen Computer und senden diejenigen zurück, die während des Migrationsprozesses verändert werden.
2. Sie halten die aktuelle virtuelle Maschine an, migrieren den Speicher und starten die neue virtuelle Maschine.
3. Sie lassen die neue virtuelle Maschine alle neuen Seiten jeweils bei Bedarf abrufen, d.h., die Prozesse starten sofort auf der neuen virtuellen Maschine und kopieren die Speicherseiten immer dann, wenn sie sie benötigen.

Die zweite Option kann zu unannehmbaren Ausfallzeiten führen, falls die zu migrierende virtuelle Maschine einen kontinuierlichen Dienst ausführt. Auf der anderen Seite kann der reine bedarfsgesteuerte Ansatz der dritten Option die Migrationszeit außerordentlich verlängern und darüber hinaus zu einer schlechten Leistung führen, da es lange dauert, bis der Arbeitssatz der migrierten Prozesse übertragen ist.

Als Alternative schlugen Clark *et al.* (2005) einen Vorabkopieransatz vor, der die erste Option mit einer kurzen Anhalten-und-Kopieren-Phase aus der zweiten Option kombiniert. Es hat sich gezeigt, dass dieses Arrangement zu Ausfallzeiten von 200 ms oder weniger führt.

Was die lokalen Ressourcen angeht, vereinfacht sich die Sache, wenn wir es nur mit einem Clusterserver zu tun haben. Da es nur ein einziges Netzwerk gibt, müssen wir nur die neue Bindung zwischen Netzwerk und MAC-Adresse bekannt machen, damit die Clients über die richtige Netzwerkschnittstelle Kontakt mit dem migrierten Prozess aufnehmen können. Wenn wir voraussetzen können, dass die Speicherung auf einer eigenen Schicht stattfindet (wie in ▶Abbildung 3.12), dann ist die Migration der Bindung zu Dateien ähnlich einfach.

Insgesamt ergibt sich, dass wir nicht mehr nur Prozesse migrieren, sondern ein in Betrieb befindliches Systeme inklusive Betriebssystem von einem Computer zu einem anderen übertragen können.

ZUSAMMENFASSUNG

Prozesse spielen eine grundlegende Rolle in verteilten Systemen, da sie die Grundlage der Kommunikation zwischen verschiedenen Computern bilden. Ein wichtiger Aspekt ist dabei der interne Aufbau der Prozesse sowie vor allem die Frage, ob sie mehrere Steuerthreads unterstützen oder nicht. Threads sind in verteilten Systemen vor allem nützlich, um die CPU auch während der Durchführung einer blockierenden E/A-Operation weiter nutzen zu können. Auf diese Weise ist es möglich, hochgradig effiziente Server zu erstellen, die mehrere Threads parallel ausführen, wobei einige von ihnen blockiert sind, um auf den Abschluss von Festplatten-E/A-Vorgängen oder Netzwerkkommunikation zu warten.

Die Gliederung verteilter Anwendungen in **Clients** und **Server** hat sich als sehr nützlich erwiesen. Clientprozesse implementieren im Allgemeinen Benutzerschnittstellen, die von sehr einfachen Anzeigen bis zu erweiterten Oberflächen zur Handhabung eines Dokumentverbunds reichen. Außerdem sorgt die Client-Software für Verteilungstransparenz, indem sie die Einzelheiten der Kommunikation mit den Servern verbirgt, unter anderem deren Aufenthaltsort und die Frage, ob sie repliziert sind oder nicht. Darüber hinaus ist die Client-Software auch teilweise dafür verantwortlich, Ausfälle und deren Behebung zu verbergen.

Server sind häufig komplizierter als Clients, weisen aber dennoch nur relativ wenige neue Aspekte für den Entwurf auf. So können Server z.B. entweder iterativ oder nebenläufig sein, einen oder mehrere Dienste implementieren, zustandslos oder zustandsbehaftet sein. Andere Gesichtspunkte des Entwurfs sind die Adressierung von Diensten und Mechanismen, die einen Server unterbrechen, nachdem eine Dienstanforderung ausgegeben worden ist und möglicherweise bereits verarbeitet wird.

Besondere Aufmerksamkeit empfiehlt sich bei der Gliederung von Servern in einem **Cluster**. Ein übliches Ziel besteht darin, die Interna des Clusters vor der Außenwelt zu verbergen. Das bedeutet, dass der Aufbau des Clusters vor den Anwendungen verborgen werden muss. Dazu verwenden die meisten Cluster einen einzelnen Eintrittspunkt, der Nachrichten an die Server im Cluster verteilt. Eine große Herausforderung besteht darin, in einer vollständig verteilten Installation diesen einzelnen Eintrittspunkt transparent zu ersetzen.

Ein wichtiges Thema bei verteilten Systemen ist die **Migration** von Code zwischen verschiedenen Computern. Zwei wichtige Gründe für die Codemigration sind Leistungssteigerung und Flexibilität. Wenn die Kommunikation teuer ist, können wir sie manchmal reduzieren, indem wir Berechnungen vom Server auf den Client verlagern und den Client so viel an lokaler Verarbeitung durchführen lassen wie möglich. Die Flexibilität wird erhöht, wenn ein Client die für die Kommunikation mit einem bestimmten Server erforderliche Software dynamisch herunterladen kann. Diese Software kann eigens auf den Server abzielen, ohne dass der Client sie vorinstalliert haben muss.

Bei der **Codemigration** treten Probleme aufgrund der Verwendung lokaler Ressourcen auf. Dabei müssen entweder die Ressourcen ebenfalls migriert, neue Bindungen an lokale Ressourcen auf dem Zielcomputer eingerichtet oder systemweite Netzwerkreferenzen verwendet werden. Ein weiteres Problem ist die Heterogenität von Plattformen bei der Codemigration. Die beste Lösung dafür scheinen zurzeit virtuelle Maschinen zu sein. Sie können entweder die Form von virtuellen Prozessmaschinen annehmen, wie es z.B. bei Java der Fall ist, oder als Virtual Machine Monitors auftreten, die die Migration eines Satzes von Prozessen zusammen mit dem zugrunde liegenden Betriebssystem erlauben.

Aufgaben

1. Vergleichen Sie das Lesen einer Datei mithilfe eines Einzel- und eines Multithread-Dateiservers. Sofern sich die Daten in einem Cache des Arbeitsspeichers befinden, dauert es 15 ms, um einen Anforderung zu erhalten und zu verteilen sowie die restliche Verarbeitung durchzuführen. Wir nehmen an, dass in ein Drittel der Fälle eine Festplattenoperation nötig ist. Dann kommen weitere 75 ms hinzu, in denen der Thread ruht. Wie viele Anforderungen pro Sekunden kann ein Singlethread-Server handhaben? Wie viele sind es bei einem Multithread-Server?

2. Ist es sinnvoll, die Anzahl der Threads in einem Serverprozess zu begrenzen?

3. Im Text haben wir einen Multithread-Server beschrieben und gezeigt, warum er besser ist als ein Singlethread-Server und ein Server nach dem Prinzip eines endlichen Zustandsautomaten. Gibt es Umstände, unter denen ein Singlethread-Server besser ist? Nennen Sie ein Beispiel.

4. Einen einzigen Thread statisch mit einem Lightweight-Prozess zu verbinden, ist im Allgemeinen keine gute Idee. Warum nicht?

5. Es ist auch nicht günstig, in einem Prozess nur einen einzigen Lightweight-Prozess zu haben. Warum nicht?

6. Beschreiben Sie ein einfaches Verfahren, bei dem es so viele Lightweight-Prozesse gibt wie lauffähige Threads.

7. In X-Windows wird das Terminal eines Benutzers als Server bezeichnet, während die Anwendung als Client bezeichnet wird. Ist das sinnvoll?

8. Das X-Protokoll leidet unter Skalierbarkeitsproblemen. Wie können Sie sie angehen?

9. Proxys können die Replikationstransparenz unterstützen, indem sie wie im Text erklärt jedes einzelne Replikat aufrufen. Kann eine Anwendung (die Serverseite der Anwendung) das Ziel mehrerer replizierter Aufrufe sein?

10. Die Konstruktion eines nebenläufig ablaufenden Servers durch Einrichten eines neuen Prozesses weist gegenüber Multithread-Servern Vor- und Nachteile auf. Nennen Sie einige davon.

11. Skizzieren Sie den Entwurf eines Multithread-Servers, der mehrere Protokolle unterstützt und Sockets als Schnittstelle der Transportschicht zum zugrunde liegenden Betriebssystem verwendet.

12. Wie können wir eine Anwendung daran hindern, einen Fenstermanager zu umgehen und die Anzeige auf dem Bildschirm durcheinanderzubringen?

13. Ist ein Server, der eine TCP/IP-Verbindung unterhält, zustandslos oder zustandsbehaftet?

Aufgaben

14. Stellen Sie sich einen Webserver mit einer Tabelle vor, in der Client-IP-Adressen den Webseiten zugeordnet sind, auf die sie zuletzt zugegriffen haben. Wenn ein Client Verbindung mit diesem Server aufnimmt, schlägt dieser den Client in der Tabelle nach und gibt gegebenenfalls die registrierte Seite zurück. Ist dies ein zustandsloser oder ein zustandsbehafteter Server?

15. Die starke Mobilität kann in UNIX-Systemen dadurch unterstützt werden, dass sich ein Prozess in einen Kindprozess auf einem entfernten Computer aufteilt. Erklären Sie, wie das funktioniert.

16. ▶Abbildung 3.18 scheint anzudeuten, dass die starke Mobilität nicht mit der Ausführung von migriertem Code in einem Zielprozess kombiniert werden kann. Geben Sie ein Gegenbeispiel.

17. Stellen Sie sich einen Prozess *P* vor, der auf die Datei *F* zugreifen muss, wobei sich *F* lokal auf dem Computer befindet, auf dem *P* läuft. Wenn *P* auf einen anderen Rechner verschoben wird, muss er nach wie vor auf *F* zu greifen. Wie kann die systemweite Referenz auf *F* implementiert werden, wenn die Datei-Computer-Bindung fix ist?

18. Beschreiben Sie im Einzelnen, wie TCP-Pakete bei einem TCP-Handoff fließen. Gehen Sie dabei auch auf die Quell- und Zieladressen in den verschiedenen Headern ein.