

Bevor wir auf eine Klassifikation von Client-Server-Architekturen eingehen, müssen wir zunächst die Bedeutung der Begriffe

- *konkurrent*,
- *parallel* und
- *verteilt*

klären.

Ein *konkurrentes Programm* besteht aus mehreren Prozessen (Tasks) oder Threads, in denen die Prozesse in irgendeiner Ordnung ausgeführt werden. Ein Prozess kann dabei vor einem anderen Prozess oder einige oder alle Prozesse können alle zur gleichen Zeit ausgeführt werden. Konkurrenz macht keine Aussagen zur Ausführungsrelation von Prozessen; es werden dabei alle möglichen Ausführungsrelationen in Betracht gezogen. Die Ausführungsplattform für ein konkurrentes Programm kann ein Einprozessorsystem, Multiprozessorsystem oder ein Parallelrechner sein. Konkurrenz ist eine semantische Eigenschaft eines Programms, während *Parallelität* die Implementierung eines Programms betrifft, wie sie durch einen Compiler oder die Systemsoftware bestimmt ist. Konkurrenz kann in der Programmiersprache integriert sein (ein bekanntes Beispiel ist die Sprache Ada, welche eine konkurrente objekt-basierte Programmiersprache ist; ein weiteres Beispiel ist die Programmiersprache Java, die ein Thread Paket enthält) oder kann durch die Systemplattform vorgegeben sein (ein Beispiel ist der fork-Systemaufruf in Unix).

Ein *verteiltes Programm* ist ein Programm, bei welchem die Prozesse miteinander über Rechnergrenzen hinaus in Interaktion treten. Die einfachste Interaktion ist ein Nachrichtenaustausch und somit ein Senden und Empfangen von Nachrichten. Der Name verteiltes Programm kommt von der Tatsache, dass diese Programme auf einer verteilten Architektur ausgeführt werden, wie Multicomputer oder vernetzte Computer, bei denen die Prozesse keinen gemeinsamen Speicher besitzen. Dies schließt jedoch nicht aus, dass ein verteiltes

Programm auf einem eng gekoppelten Multiprozessor oder sogar auf einem Einprozessor-system ausführbar ist.

Das am weitesten verbreitete Modell für verteilte Programme ist das *Client-Server-Modell*: Ein Prozess, der Client, fordert eine Operation oder einen Service von einem anderen Prozess, dem Server, an. Nach Erhalt einer Anforderungsnachricht führt der Server den angeforderten Service aus und gibt dem Client ein Resultat oder das Ergebnis des Service zurück. Dieses einfache Client-Server-Modell führt zu einer Reduktion auf mehrere Clients und einem Server, und es legt fest, wie eine Anwendung einen Service eines Servers in Anspruch nehmen kann. Die begrenzten Möglichkeiten des Client-Server-Modells liegen in der Beschränkung, dass ein Client nur einen individuellen Service in Anspruch nehmen kann und ein Prozess entweder nur als Client oder nur als Server agieren kann. In realen Anwendungen muss jedoch ein Client eine Vielzahl von Services koordiniert in Anspruch nehmen und ein Server wird zum Client, wenn er weitere Services eines anderen Servers anfordert.

Parallelität wird normalerweise gesehen als die Implementierung der Konkurrenz und der Verteiltheitsbegriff steht über dem Konkurrenzbegriff: Ein Server kann konkurrent sein oder nicht, während ein Client selten konkurrent ist; betrachten wir ein Client-Server-System als ein System, so sehen wir ein konkurrentes System (mit einem konkurrenten Server), das jedoch verteilt arbeitet. Weiterhin impliziert die Verteiltheit unabhängige Ausfälle, das bedeutet, ein Teil eines Programms kann ausfallen, während der Rest des Programms weiterläuft. Im Gegensatz dazu geht man in einem konkurrenten, jedoch nicht verteilten Kontext davon aus, dass das komplette Programm ausfällt (totale Ausfallsemantik) und kein Teil des Programms weiterlaufen kann.

Das einfache und einschränkende Client-Server-Modell dient im Folgenden als Ausgangspunkt zur schrittweisen Erweiterung. Die Modellerweiterungen zerlegen dabei die Serverfunktion, und sie liefern Koordinationsmodelle für unabhängige Prozesse, die dann komplexere Services für Clients zur Verfügung stellen können. Die Koordinationsmodelle bilden die Klassen, in welche die Client-Server-Strukturen eingeordnet sind. Zur Kennzeichnung und Beschreibung der Klassen benutzen wir die gleiche Notation, wie sie für reguläre Ausdrücke üblich ist. Die Analyse und Klassifikation der Client-Server-Strukturen orientiert sich dabei an den von R.M. Adler [A 95] vorgestellten Koordinationsmodellen für Client-Server-Berechnungen. Die vollständige Erfassung aller Klassen orientiert sich an den von G.R. Andrews [A 91] vorgestellten Musterlösungen für Prozessinteraktion in verteilten Programmen.

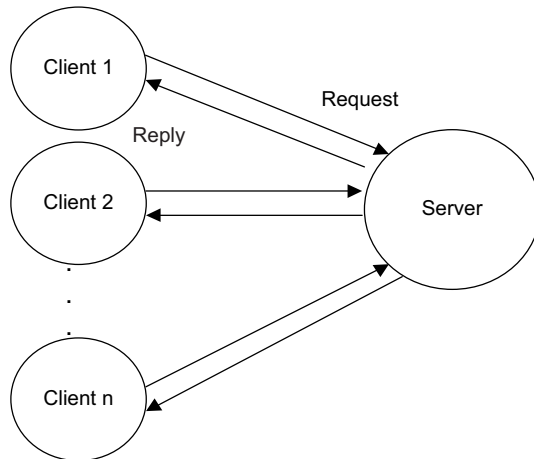
---

## 2.1 Client-Server

Ein *Client-Server-System*, bezeichnet mit  $C^+S$ , besteht aus zwei logischen Teilen:

- Einem oder mehreren Clients, der die Services oder Daten des Servers in Anspruch nimmt und somit anfordert.
- Einem Server, der Services oder Daten zur Verfügung stellt.

**Abb. 2.1** Clients und Server  
C+S



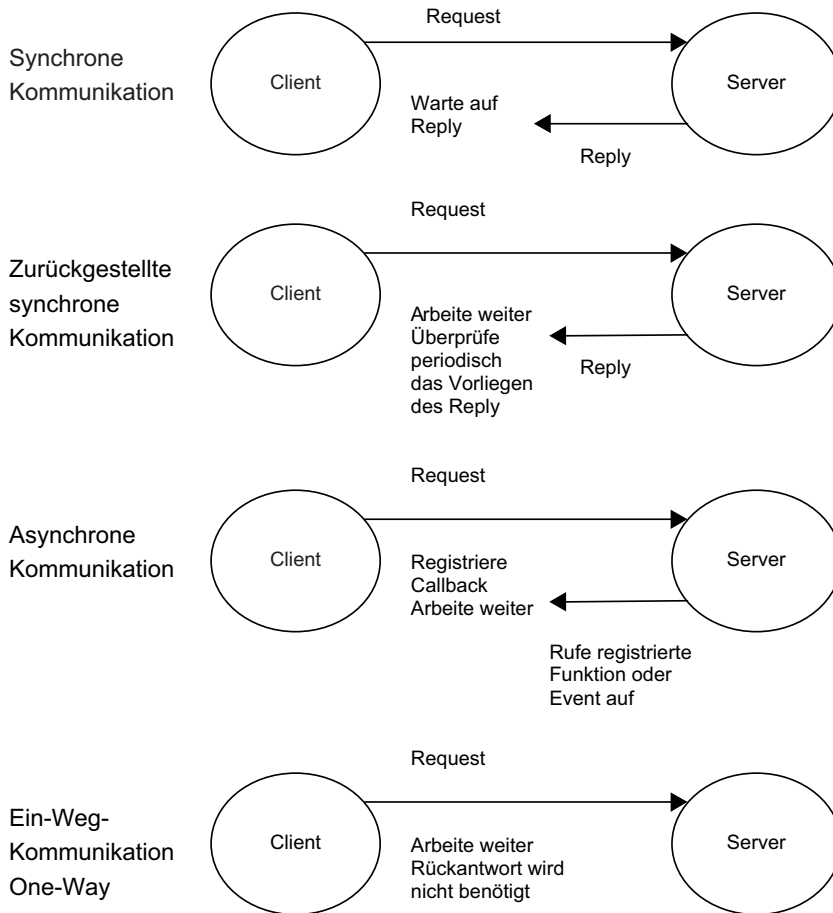
Zusammen bilden beide ein komplettes System mit unterschiedlichen Bereichen der Zuständigkeit, wobei diese Zuständigkeiten oder Rollen fest zugeordnet sind, entweder ist ein Prozess ein Client oder ein Server. Die Aufteilung in Client und Server und deren Beziehung zeigt Abb. 2.1. Ein Server kann mehrere Kunden oder Clients bedienen. Die Kunden eines Servers haben keinerlei Kenntnis voneinander und stehen demgemäß auch in keinem Bezug zueinander, außer der Tatsache, dass sie den gleichen Server verwenden. Clients und Server können auf dem gleichen oder auf unterschiedlichen Rechnern ablaufen.

Client und Server sind zwei Ausführungspfade oder -einheiten mit einer Konsumenten-Produzentenbeziehung. Clients dienen als Konsumenten und tätigen Anfragen an Server für Services oder Information und benutzen dann die Rückantwort zu ihrem eigenen Zweck und ihrer Aufgabe. Server spielen die Rolle des Produzenten und füllen die Daten- oder Serviceanfragen, die von den Clients gestellt wurden. Die Interaktion zwischen den Clients und dem Server verlaufen somit nach einem fest vorgegebenen Protokoll: Der Client sendet eine *Anforderung (request)* an den Server, dieser erledigt die Anforderung oder Anfrage und schickt eine *Rückantwort (reply)* zurück an den Client.

Ein Client ist ein auslösender Prozess und ein Server ist ein reagierender Prozess. Clients tätigen eine Anforderung, die Reaktionen des Servers auslösen. Clients initiieren Aktivitäten zu beliebigen Zeitpunkten, und andererseits warten Server auf Anfragen von Clients und reagieren dann darauf. Der Server stellt somit einen zentralen Punkt dar, an den Anforderungen geschickt werden können, und nach Erledigung der Anfrage sendet der Server das Ergebnis an den Client zurück.

### 2.1.1 Interaktionssemantik

Wenn zwischen einem Client und dem Server eine Interaktion stattfindet, so muss festgelegt werden, wie der Client und Server sich koordinieren beim Ablauf der Interaktion.



**Abb. 2.2** Interaktions-Koordinations-Arten

Da eine lokale Interaktion (Interaktion auf einem Rechner) sich nicht von einer entfernten Interaktion (Interaktion auf unterschiedlichen, voneinander entfernten Rechnern) unterscheiden soll, muss überprüft werden, inwieweit sich die lokalen Gegebenheiten auf den entfernten Fall übertragen lassen.

**Interaktionskoordination** Die verschiedenen Interaktions-Koordinations-Arten zeigt Abb. 2.2. Wartet der Client nach Absenden der Anforderung an den Server auf eine Rückantwort, bevor er anderen Aktivitäten nachgeht, so liegt der *blockierende oder synchrone* Fall vor. Dieses Vorgehen ist leicht zu implementieren, jedoch ineffizient in der Ausnutzung der Prozessorfähigkeiten des Clients. Während der Server die Anfrage bearbeitet, ruht die Arbeit des Clients und erst wenn die Rückantwort kommt, setzt der Client seine Arbeit fort.

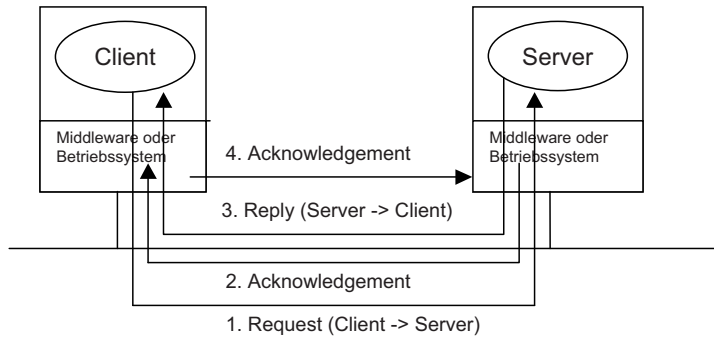
Sendet der Client nur seine Anforderung und arbeitet er sofort weiter, so liegt der *nicht blockierende oder asynchrone* Fall vor. Irgendwann später nimmt er dann die Rückantwort entgegen. Der Vorteil dieses Verfahrens ist, dass der Client parallel zur Nachrichtenübertragung weiterarbeiten kann und den Client-Prozess nicht durch aktives Warten belastet, wie beim blockierenden Fall. Jedoch muss bei dieser Methode der erhöhte Effizienzgewinn mit erhöhter Kontrollkomplexität bei Erhalt der Rückantwort erkauft werden. Die Rückantwort muss dabei in einer lokalen Warteschlange abgelegt werden, welche der Client dann so lange abfragen muss, bis die Rückantwort eingetroffen ist und somit in der Warteschlange vorliegt. In diesem Fall spricht man auch von verschobener oder *zurückgestellter synchroner (deferred synchronous)* Kommunikation. Ein alternatives Vorgehen sieht beim Client eine Registrierung von *Rückrufen (callbacks)* vor. Die Rückrufe können dann Funktionszugangspunkte oder Ereignisse sein. Beim Eintreffen der Rückantwort werden dann die registrierten Funktionen bzw. Ereignisbehandlungsroutinen aktiviert. Dieser Ansatz eliminiert das ständige Abfragen der lokalen Warteschlange, generiert jedoch möglicherweise Rückrufe zu ungelegenen Zeiten und benötigt damit zusätzlichen Kontrolloverhead, um solche unerwünschten Unterbrechungen auszuschließen. Eine weitere Möglichkeit ist, dass der Client nur eine Anforderung abschickt und sich dann nicht mehr um die Rückantwort kümmert. In diesem Fall liegt eine *Ein-Weg-Kommunikation (one-way)* vor.

**Ablaufsemantik der Interaktion** Der Ablauf der Interaktion, die zwischen zwei Rechnern stattfindet, soll die gleiche Semantik besitzen, wie wenn die Interaktion lokal, also auf einem Rechner abläuft; d. h. lokale und entfernte Interaktion sollen die gleiche Syntax und Semantik besitzen. Selbst wenn die Anforderungen oder Aufrufe der Clients keinerlei syntaktischen Unterschied zwischen lokaler und entfernter Interaktion aufweisen, so muss doch der semantische Unterschied mit in eine die Interaktion benutzende Anwendung einfließen.

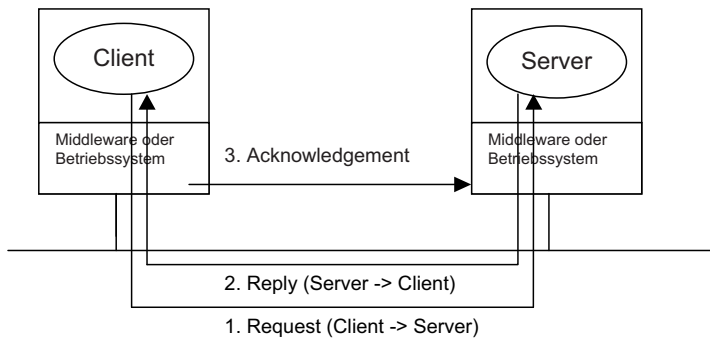
Um auf Übertragungsfehler und Ausfälle zu reagieren, kann eine Ausnahmebehandlung (exception handling) eingeführt sein, was dann jedoch zu syntaktischen Unterschieden bei lokaler und entfernter Interaktion führt. Weiterhin führt das zu semantischen Unterschieden zwischen lokaler und entfernter Transaktion, da diese Fehlerfälle gar nicht bei einer lokalen Interaktion auftreten können. In vielen Programmiersprachen wie z. B. Ada, C++, Java kann eine Ausnahmebehandlungsroutine angegeben werden, die dann beim Auftreten eines speziellen Fehlerfalles angesprungen wird. In C unter Unix lassen sich für solche Zwecke auch Signal-Handler einsetzen.

Da die Interaktion mit Hilfe zugrundeliegende Netzwerkkommunikation implementiert ist, vergrößert diese die Anzahl der Interaktionsfehler. Diese Fehler können sein:

1. Die *Anforderung geht verloren* oder erfährt eine Verzögerung, oder
2. die *Rückantwort geht verloren* oder erfährt eine Verzögerung, oder
3. der *Server* oder der *Client* können zwischenzeitlich *abgestürzt* und dadurch nicht erreichbar sein.



a) Individuell quitierte Nachrichten



b) Quittierung eines Request und Reply

**Abb. 2.3** Zuverlässige Nachrichtenübertragung: **a** durch individuell quitierte Nachrichten, **b** durch Quittierung eines Request und Reply

Eine *unzuverlässige Interaktion* übergibt die Nachricht nur dem Netz und es gibt keine Garantie, dass die Nachricht beim Empfänger ankommt. Die Anforderungsnachricht kommt *nicht oder höchstens einmal* dabei beim Server an. In diesem Fall spricht man von *may be Semantik* der Interaktion. Eine zuverlässige Interaktion muss dann selbst vom Benutzer implementiert werden.

Zur Erhaltung einer zuverlässigen Interaktion (siehe für beide Fälle Abb. 2.3) kann entweder

1. jede Nachrichtenübertragung durch Senden einer Rückantwort quitiert werden, oder
2. ein Request und ein Reply werden zusammen durch eine Rückantwort quitiert.

Im Fall Eins muss nach dem Senden der Anforderung der Server an den Client eine Quittierung zurückschicken. Eine Rückantwort vom Server an den Client wird dann vom Client an den Server quitiert. Damit braucht ein Request mit anschließendem Reply vier Nachrichtenübertragungen.

Im zweiten Fall betrachtet man eine Client-Server-Kommunikation als eine Einheit, die quittiert wird. Der Client blockiert dabei, bis die Rückantwort eintrifft, und diese Rückantwort wird quittiert.

Bei einer zuverlässigen Nachrichtenübertragung muss der Sendeprozess blockiert werden und er muss warten, bis die Rückantwort innerhalb einer vorgegebenen Zeit eintrifft. Trifft die Rückantwort nicht innerhalb der vorgegebenen Zeitschranke ein, so wird die Nachricht erneut gesendet und die Zeitschranke neu gesetzt. Führt das nach mehrmaligen Versuchen nicht zum Erfolg, so ist im Moment kein Senden möglich (die Leitung ist entweder gestört und die Pakete gehen verloren, oder der Empfänger ist nicht empfangsbereit). Erhält ein Empfänger durch mehrfaches Senden die gleiche Nachricht mehrmals, so kann er die erneut eingehende gleiche Nachricht bearbeiten, und er stellt so sicher, dass die eingehende Anforderung *mindestens einmal* bearbeitet wird (*at least once*). Siehe dazu Abb. 2.4a. Dabei wird jedoch für den Erhalt der Nachricht bei Systemausfällen keine Garantie gegeben.

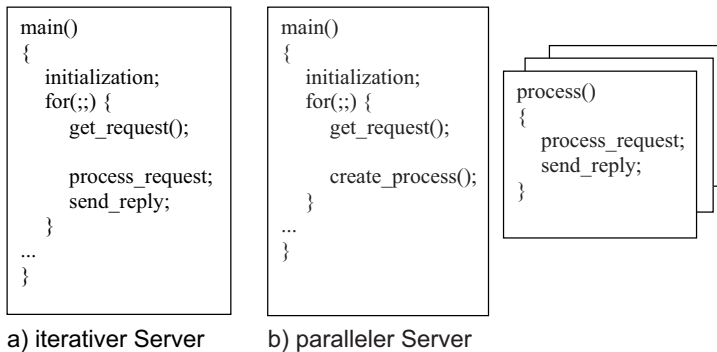
Die *at least once* Semantik hat den Nachteil, dass durch die mehrfache Bearbeitung der Anforderung die Daten inkonsistent werden können. Betrachten Sie dazu beispielsweise einen File-Server, der einen gesendeten Datensatz an einen bestehenden File anhängt. Die *at least once* Methode hängt dann möglicherweise den Datensatz mehrfach an einen File hintenan. Diese Methode arbeitet jedoch korrekt, wenn ein Client einen bestimmten Datensatz eines Files vom File-Server zurückhaben möchte. Hier tritt nur der Umstand auf, dass der Client diesen Datensatz möglicherweise mehrfach erhält.

Besser, aber mit erhöhtem Implementierungsaufwand, lässt sich auch bewerkstelligen, dass die Nachricht *höchstens einmal* (*at most once*) erhalten wird (siehe dazu Abb. 2.4b), jedoch ohne Garantie bei Systemfehlern, d. h. möglicherweise auch gar nicht. Bei dieser Methode benötigt der Empfänger eine Anforderungsliste, welche die bisher gesendeten Anforderungen enthält. Jedes Mal, wenn dann eine neue Anforderung eintrifft, stellt der Empfänger mit Hilfe der Nachrichtenidentifikation fest, ob schon die gleiche Anforderung in der Liste steht. Trifft dies zu, so ging die Rückantwort verloren und es muss erneut eine Rückantwort gesendet werden. Ist die Anforderung noch nicht in der Liste vermerkt, so wird sie in die Liste eingetragen. Anschließend wird die Anforderung bearbeitet und eine entsprechende Rückantwort wird gesendet. Ist die Rückantwort bestätigt, kann die Anforderung aus der Liste gestrichen werden.

Soll auch noch der Systemfehler des Plattenausfalls sich nicht auswirken, so muss die Anforderungsliste im stabilen Speicher (*stable storage*) gehalten werden. Bei *genau einmal mit Garantie bei Systemfehlern* spricht man von der *exactly once Semantik*.

Fällt der Server aus, nachdem die Anforderung den Server erreicht hat, so gibt es keine Möglichkeit, dies dem Client mitzuteilen, und für den Client gibt es keine Möglichkeit, dies herauszufinden. Der Client kann beim Ausbleiben der Rückantwort erneut die Anforderung senden und hoffen, dass der Server wieder läuft. Dadurch gleitet man auf die *at most once* Semantikebene ab und die *exactly once* Semantik ist nicht realisierbar. Die Möglichkeit eines Serverausfalles ist der Grund, dass entfernte Interaktion nicht die Semantik





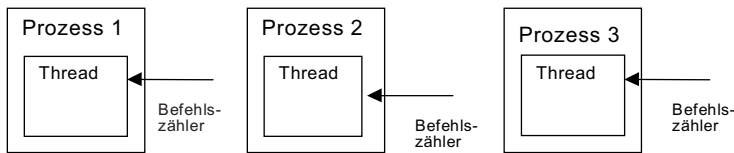
**Abb. 2.5** Iterativer versus paralleler Server

## 2.1.2 Parallele Server

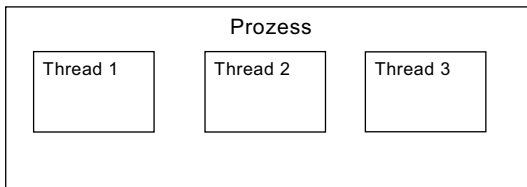
Ein Server muss mehrere Clients bedienen können. Dazu wartet zunächst der Server auf eingehende Anforderungen (Nachrichten) von Clients. Die Analyse und Bearbeitung der Anforderung, also die Bedienung des Clients, kann dann entweder sequentiell oder parallel geschehen:

1. Im *sequentiellen Fall* (siehe dazu Abb. 2.5a) bearbeitet der Server nach dem Eintreffen der Anforderung diese Anforderung. Nach der Bearbeitung der Anforderung sendet er die Rückantwort an den Client zurück. Er wartet nun anschließend wieder auf die nächste Anforderung eines Clients oder falls dessen Anforderung bereits vorliegt, beginnt er mit deren Bearbeitung. Der Server bearbeitet somit bei jeder Iteration eine Anforderung.
2. Im *parallelen Fall* (siehe dazu Abb. 2.5b) startet der Server nach Eintreffen der Anforderung einen neuen Prozess zur Bearbeitung der Anforderung. Nach Bearbeitung der Anforderung sendet der Prozess die Rückantwort an den Client zurück. Der Hauptprozess kehrt nach dem Start des Prozesses sofort zur Annahme der nächsten Anforderung des Clients zurück und startet, falls eine weitere Anforderung bereits vorliegt, einen weiteren Prozess zu dessen Bearbeitung. Durch dieses Vorgehen ist im Allgemeinen die Antwortzeit für Clients im parallelen Fall günstiger als im sequentiellen Fall.

Bei einem parallelen Server müssen die Prozesserschöpfung und die Prozessumschaltung mit minimalem zeitlichem Aufwand erfolgen, um einen effizienten Server zu erhalten. Aus diesem Grund benutzen parallele Server meistens nicht den Prozess- oder Taskmechanismus zur Implementierung der Konkurrenz, sondern die im nachfolgenden Abschnitt beschriebenen Threads.



a) Drei Prozesse mit jeweils einem Thread



b) ein Prozess mit drei Threads

**Abb. 2.6** Prozesse und Threads

### 2.1.2.1 Threads

Ein Prozess ist definiert durch die Betriebsmittel, die er benötigt, und den Adressbereich, in dem er abläuft. Ein Prozess wird im Betriebssystem beschrieben durch einen *Prozesskontrollblock* (*Process Control Block – PCB*). Der PCB besteht aus dem Hardwarekontext und einem Softwarekontext. Ein Prozesswechsel bewirkt den kompletten Austausch des PCB. Zur Erreichung eines schnelleren Prozesswechsels muss die Information, die bei einem Prozesswechsel auszutauschen ist, reduziert werden. Dies erreicht man durch Aufteilung eines Prozesses in mehrere „Miniprozesse“ oder bildlich gesprochen, durch Auffädeln mehrerer dieser „Miniprozesse“ unter einem Prozess. Zum Unterschied Prozess und Thread siehe Abb. 2.6. Demgemäß bezeichnet man solche „Miniprozesse“ als Threads (Thread – Faden). Mehrere oder eine Gruppe von Threads haben den gleichen Adressraum und besitzen die gleichen Betriebsmittel (gleiche Menge von offenen Files, Kindprozesse, Timer, Signale usw.). Nachteilig ist natürlich bei einem Adressraum für die Threads, dass alle Threads den gleichen Adressraum benutzen und damit die Schutzmechanismen zwischen verschiedenen Threads versagen. Die Umgebung, in welcher ein Thread abläuft, ist ein Prozess (Task). Ein traditioneller Prozess entspricht einem Prozess mit einem Thread.

Ein Prozess bewirkt nichts, wenn er keinen Thread enthält und ein Thread muss genau in einem Prozess liegen. Ein Thread hat wenigstens seinen eigenen Programmzähler, seine eigenen Register und gewöhnlich auch seinen eigenen Keller. Damit ist ein Thread die Basiseinheit, zwischen denen die CPU einer Ein- oder Multiprozessor-Maschine umgeschaltet werden kann. Der Prozess ist die Ausführungsumgebung, und die dazugehörigen Aktivitätsträger sind die Threads. Ein Prozess besitzt einen virtuellen Adressraum und eine Liste mit Zugriffsrechten auf die Betriebsmittel nebst notwendiger Verwaltungsinformation. Ein Thread ist ein elementares ausführbares Objekt für einen realen Prozessor und läuft im Kontext eines Prozesses.



<http://www.springer.com/978-3-8348-1670-2>

Grundkurs Verteilte Systeme  
Grundlagen und Praxis des Client-Server und  
Distributed Computing  
Bengel, G.  
2014, XV, 355 S. 103 Abb., Softcover  
ISBN: 978-3-8348-1670-2