3 Programmiermodelle für parallele und verteilte Systeme

Das vorherrschende Programmiermodell für parallele und verteilte Systeme ist das ***Client-Server-Modell***. Das Client-Server-Modell ist unabhängig von der zugrunde liegenden Hardwareplattform und läuft auf allen Architekturen:

Plattformen für das Client-Server-Modell

- Einprozessorsysteme,
- eng gekoppelten Multiprozessoren und Multicoreprozessoren und
- lose gekoppelten Multiprozessoren und Multicomputer und somit auch auf Cluster.

Das Client-Server-Modell ist in Abschnitt 3.1 beschrieben. Die Weiterentwicklung des Client-Server-Modells führt auf die ***Service-orientierten Architekturen (SOA)***, welche eine verteilte Architektur besitzen. Auf SOA geht Abschnitt 3.2 ein.

Bei der Implementierung des Client-Server-Modells, des SOA-Modells, den parallelen Servern und bei den Modellen für parallele und verteilte Verarbeitung ist zu unterscheiden,

Gemeinsamer Speicher oder Verteilter Speicher?

- ob die Programme auf einem ***System mit gemeinsamem Speicher***, also auf einem Einprozessorsystem oder eng gekoppelten Multiprozessorsystem oder Multicore-Prozessoren,
- oder auf einem ***System mit verteiltem Speicher***, also Prozessor-Speicherpärchen (lose gekoppelten Multiprozessor, Multicomputer oder Cluster) ausgeführt werden.

Bei einem gemeinsamen Speicher können die parallel abgewickelten Prozesse gleichzeitig auf gemeinsame Daten zugreifen (siehe Abschnitt 3.3). Das ist jedoch nicht möglich bei einem System mit verteiltem Speicher.

Das Gemeinsame verwaltet ein zentraler Server

Die Programmiermodelle für verteilten Speicher besitzen keine gemeinsamen Daten und erfordern eine verteilte Programmierung (siehe Abschnitt 3.4). Ein Ausweg, das fehlende Gemeinsame herzustellen, ist, eine zentrale Instanz oder einen ***zentralen Server*** einzuführen, auf den alle Prozesse zugreifen. Die gemeinsamen Daten können dann in den

zentralen Server gelegt werden, und alle Prozesse und somit Clients haben Zugriff darauf.

nebenläufig oder kooperativ

Die verschiedenen nachfolgend vorgestellten Programmiermodelle für gemeinsamen und für verteilten Speicher (Abschnitt 3.3 und Abschnitt 3.4) lassen sich noch horizontal untergliedern in nebenläufig und kooperativ. Bei ***nebenläufigen*** Prozessen ist ihr Einsatzgebiet hauptsächlich für parallele Systeme bestimmt. Ist das Programmiermodell ***kooperativ***, so ist das Einsatzgebiet die Client-Server-, Service-orientierte- oder Verteilte Programmierung.

Eine andere Übersicht und Klassifizierung und die Einbettung der objektorientierten Konzepte in konkurrente und verteilte Systeme sind in [BGL 98] beschrieben. Die Klassifizierung der mehr forschungsorientierten Ansätze unterscheidet

- den ***Bibliotheks-Ansatz*** und damit aus der Sichtweise des Systementwicklers,
- den ***integrativen Ansatz*** und damit mehr aus der Sichtweise des Anwendungsentwicklers und
- den ***reflektierenden Ansatz***, der eine Brücke bildet zwischen den beiden anderen Ansätzen.

Unsere hier gewählte Klassifikation unterscheidet nicht die bei objektorientierten Sprachen vorliegenden Bibliotheken und die Sprache integrierten Ansätze. Die nachfolgend vorgestellten Sprachen und Systeme sind mehr praxisorientiert und orientieren sich mehr an den Ansätzen, die in der Industrie und somit in der Praxis im Einsatz sind.

Eine weitere schöne Übersicht und Darstellung der verschiedenen Programmiermodelle für konkurrente und verteilte Programme ist die zweite Auflage des Buches von Ben-Ari [BA 06].

Die Java Programmiermodelle Sockets und Remote Method Invocation (RMI) sowie die Beschreibung vieler verteilter Algorithmen in Java und das Ausformulieren in Java von einer großen Anzahl von Algorithmen sind in Garg [G 04] enthalten.

3.1 Client-Server-Modell

Ein ***Client-Server-System,*** bezeichnet mit dem regulären Ausdruck ***C^+S,*** [B 04] besteht aus zwei logischen Teilen:

Client-Server-System C^+S

- Einem oder ***mehreren Clients***, welche die Services oder Daten des Servers in Anspruch nehmen und somit anfordern.

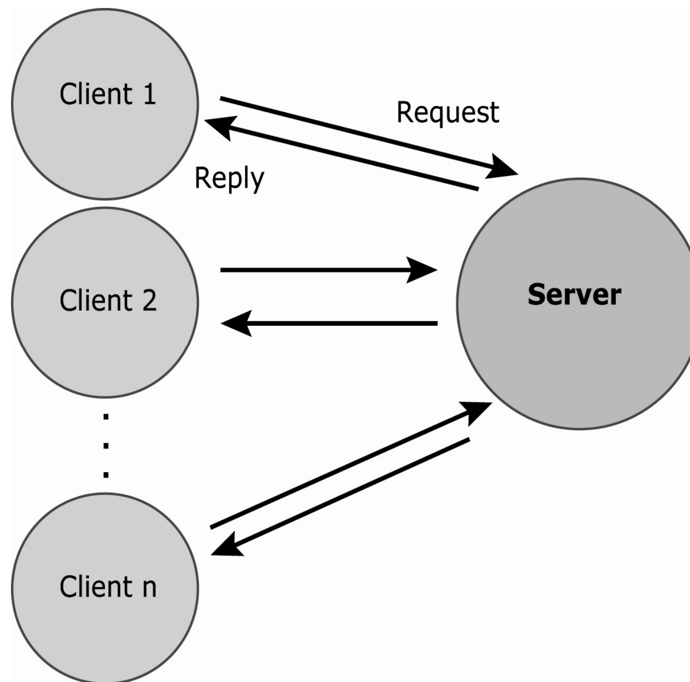
- ***Einem Server***, der Services oder Daten zur Verfügung stellt.

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. 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 über Services oder Information. Sie benutzen dann die Rückantwort zu ihrem eigenen Zweck und zur Erledigung ihrer Aufgabe. Server spielen die Rolle des Produzenten und erledigen 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 eine Reaktion des Servers auslöst. 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.

Abb. 3-1: Clients und Server C+S



3.1.1 Fehlersemantik

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

blockierend

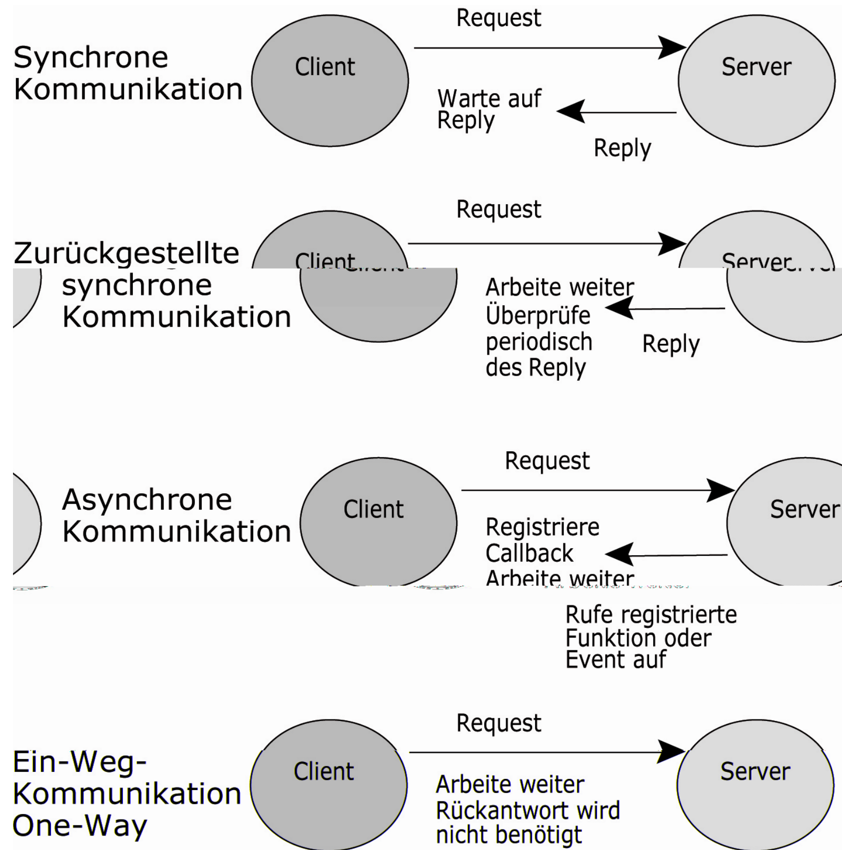
Wartet der Client nach Absenden der Anforderung an den Server auf eine Rückantwort, bevor er anderen Aktivitäten nachgeht, so liegt der ***blockierende*** oder ***synchrone*** Fall vor. Dieses Vorgehen ist leicht zu implementieren, jedoch ineffizient in der Ausnutzung der Prozessorfähigkeiten des Clients. Während der Server die Anfrage bearbeitet, ruht

die Arbeit des Clients, und erst wenn die Rückantwort kommt, setzt der Client seine Arbeit fort.

nicht blockierend

Sendet der Client nur seine Anforderung und arbeitet 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 Funktionseingangspunkte 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.

Abb. 3-2: Interaktions-koordinations-arten



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 In-

teraktion 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 zugrunde liegender Netzwerkkommunikation implementiert ist, vergrößert diese die Anzahl der Interaktionsfehler. Diese Fehler können sein:

Interaktionsfehler

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

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 beim Server an. In diesem Fall spricht man von may be-Semantik der Interaktion. Eine zuverlässige Interaktion muss dann selbst vom Benutzer implementiert werden.

may be-Semantik

Zur Erhaltung einer zuverlässigen Interaktion kann entweder

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

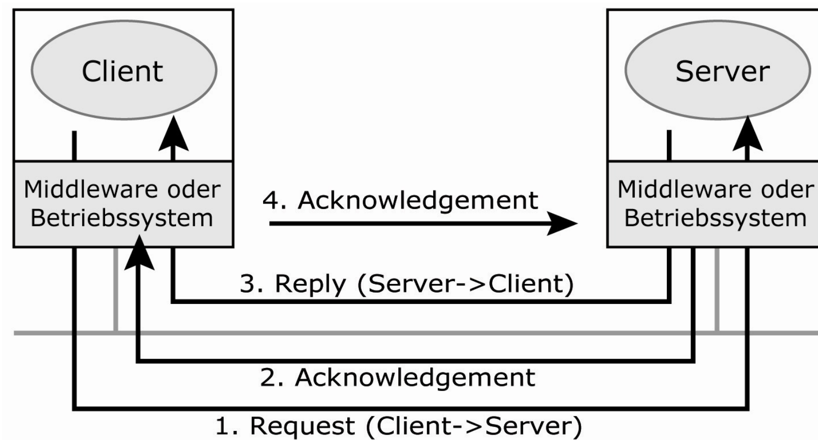
Im Fall Eins muss nach dem Senden der Anforderung der Server an den Client eine Quittierung zurückschicken. Eine Rückantwort vom Server an den Client wird dann vom Client an den Server quittiert. Damit braucht ein Request mit anschließendem Reply vier Nachrichtenübertragungen.

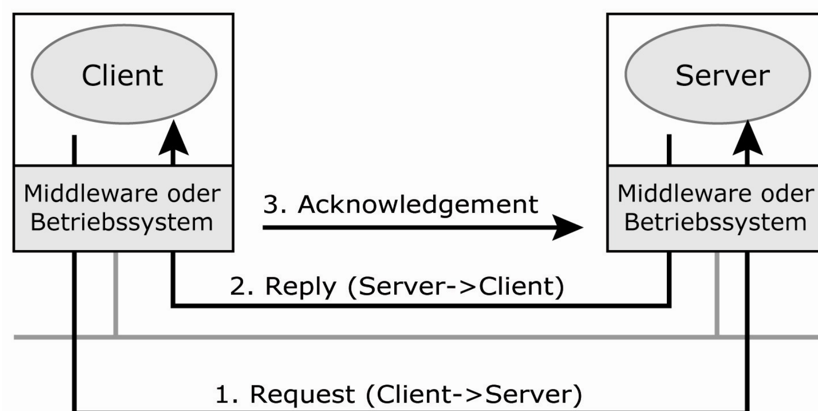
Im zweiten Fall betrachtet man eine Client-Server-Kommunikation als eine Einheit, die quittiert wird. Der Client blockiert dabei, bis die Rückantwort eintrifft, und diese Rückantwort wird quittiert.

Abb. 3-3: Zuverlässige Nachrichtenübertragung

a) durch individuell quittierte Nachrichten

b) durch Quittierung eines Request und Reply





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 Sen-

den 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***). Dabei wird jedoch für den Erhalt der Nachricht bei Systemausfällen keine Garantie gegeben.

at least once-Semantik

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 eine bestehende Datei anhängt. Die at least once-Methode hängt dann möglicherweise den Datensatz mehrfach an eine Datei hinten an. 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, 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 gesendet. Ist die Rückantwort bestätigt, kann die Anforderung aus der Liste gestrichen werden.

at most once-Semantik

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***.

exactly once-Semantik