Client/Server-Strukturen

Client/Server

Gliederung in zwei logische Teile

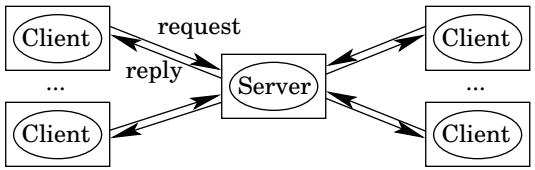
- Clients: nehmen Dienste oder Daten eines Servers in Anspruch (auslösendes Element)
- Server: stellt Dienste oder Daten zur Verfügung (reagierendes Element)
- Zuständigkeiten/Rollen sind fest zugeordnet: entweder ist ein Prozess ein Client oder ein Server
- Clients haben keinerlei Kenntnis voneinander und stehen in keinem Bezug zueinander.

→ Client und Server sind zwei Ausführungspfade mit Interaktion nach dem Erzeuger/Verbraucher-Muster

Client/Server (2)

Interaktionen zwischen Client und Server verlaufen nach einem fest vorgegebenen Protokoll:

- Client sendet Anforderung (request) an den Server
- Server erledigt Anforderung oder Anfrage und schickt Rückantwort (reply) an den Client



Wie koordinieren sich Client und Server?

· Wartet der Client auf die Antwort des Servers

· oder ruft der Server am Ende eine callback-Funktion auf?

Koordination der Interaktion

blockierend/synchron: Client wartet nach Absenden der Anforderung auf die Antwort des Servers.

- leicht zu implementieren
- ineffizient, da die Arbeit des Clients ruht, während der Server die Anfrage bearbeitet

nicht blockierend/asynchron: Client verschickt die Anforderung, arbeitet sofort weiter und nimmt irgendwann später die Antwort entgegen.

- Antwort kann in Warteschlange abgelegt werden
- oder Server ruft am Ende eine callback-Funktion auf
- oder Einweg-Kommunikation (Antwort nicht benötigt)

Interaktionssemantik

Anforderung: lokale und entfernte Interaktion soll gleiche Syntax und Semantik besitzen.

- Reagieren auf Übertragungsfehler und Ausfälle durch Ausnahmebehandlung (exception handling)

Interaktionsfehler:

- Anforderung geht verloren oder wird verzögert
- Rückantwort geht verloren oder wird verzögert
- Server oder Client sind zwischenzeitlich abgestürzt

unzuverlässige Kommunikation: Nachricht wird ans Netz übergeben, aber es gibt keine Garantie, dass die Nachricht beim Empfänger ankommt. (**may be Semantik**)

Interaktionssemantik (2)

zuverlässige Kommunikation:

- jede Nachrichtenübertragung wird durch Senden einer Rückantwort quittiert: Request mit anschließendem Reply benötigt vier Nachrichtenübertragungen
- Request und Reply werden zusammen durch eine Rückantwort quittiert: der Client blockiert, bis Rückantwort eintrifft, nur die Rückantwort wird quittiert

Interaktionssemantik (3)

at least once-Semantik:

- Sendeprozess wartet, bis Rückantwort innerhalb einer gewissen Zeit eintrifft. Überschreiten der Zeitschranke führt zu erneutem Verschicken der Nachricht und Setzen der Zeitschranke.
- schlagen Versuche mehrmals fehl, ist im Moment kein Senden möglich (Leitung gestört, Adressat nicht empfangsbereit, ...)
- erhält der Empfänger durch mehrfaches Senden dieselbe Nachricht mehrmals, kann durch erneute Bearbeitung sichergestellt werden, dass die Anforderung mindestens einmal bearbeitet wird

Interaktionssemantik (4)

Nachteil: evtl. inkonsistente Daten! Bsp: File-Server hängt gesendeten Datensatz an bestehende Datei an.

at most once-Semantik:

- Empfänger verwaltet Anforderungsliste, die die bisher gesendeten Anfragen enthält
- Anfragen werden mit Hilfe der Liste überprüft:
  * steht die Anfrage in der Liste, so ging die Rückantwort verloren und wird nochmal gesendet
  * sonst wird die Anfrage in der Liste vermerkt, die Anfrage bearbeitet und Rückantwort gesendet
- wurde die Rückantwort bestätigt, wird die zugehörige Anfrage aus der Liste gelöscht

Interaktionssemantik (5)

exactly once-Semantik: Systemfehler (z.B. Plattenausfall) ausschalten, indem Anforderungsliste im stabilen Speicher gehalten wird.

Server-Ausfall, nachdem die Anfrage den Server erreicht hat: Client kann nicht über den Fehler benachrichtigt werden, Client kann Fehler nicht feststellen → Client schickt Anfrage erneut, exactly once-Semantik nicht realisierbar.

⇒ **entfernte Interaktion kann nicht die Semantik von lokaler Interaktion erreichen**

Client-Ausfall, nachdem Anfrage gesendet wurde: für den Server ist kein Partner mehr vorhanden, Ergebnisse werden nicht abgenommen

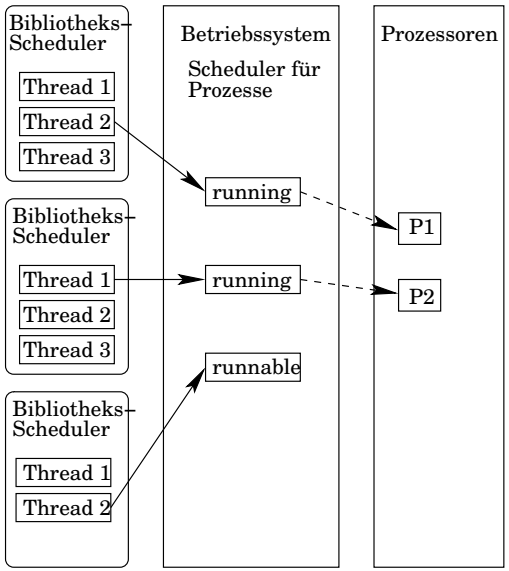
Parallele Server

Ein Server muss mehrere Clients bedienen können:

- ein **iterativer Server** bearbeitet die Anfrage, sendet die Rückantwort und wartet auf die nächste Anfrage.
- ein **paralleler Server** startet nach dem Eintreffen einer Anfrage einen neuen Prozess/Thread zur Bearbeitung der Anfrage. Der Thread sendet die Rückantwort, der Hauptprozess steht sofort zur Annahme weiterer Anfragen zur Verfügung → i.Allg. kürzere Antwortzeiten

Prozesserzeugung und -umschaltung muss mit minimalem Aufwand erfolgen, daher wird in der Regel mit Threads und nicht mit Prozessen gearbeitet.

Threads auf Benutzerebene



das Betriebssystem kennt keine Threads, sondern es verwaltet Prozesse (z.B. Linux bis Kernel-Version 2.4)

Thread-Bibliothek wählt einen der ablaufbereiten Threads aus, der dann in einem Prozess-Kontext läuft

→ all-to-one mapping

Threads auf Benutzerebene (2)

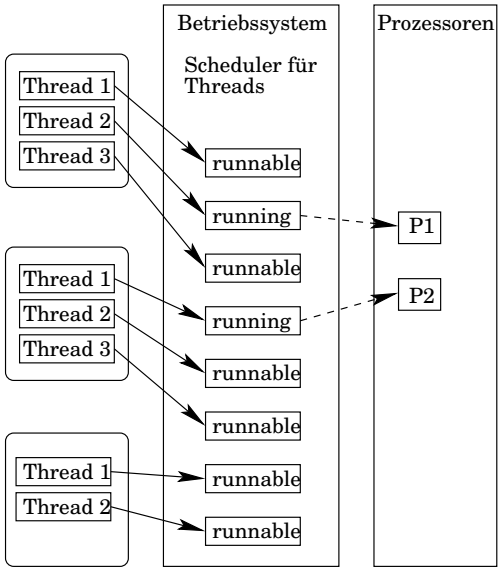
Vorteile:

- die Thread-Bibliothek kann einfach auf das bestehende Betriebssystem aufgesetzt werden
- Implementierung benötigt keine aufwendigen Systemaufrufe zum Anlegen von/Umschalten zwischen Threads
- es können beliebig viele Threads erzeugt werden, ohne das Betriebssystem zu belasten

Nachteile:

- unfaires Scheduling, da das BS nur Prozesse kennt
  Bsp: Prozess A mit 1 Thread, Prozess B mit 100 Threads
- Threads eines Prozesses laufen nie echt parallel, da das BS die CPUs nur den Prozessen zuordnet

Threads auf Betriebssystemebene



das BS kennt Threads (Linux ab Kernel-Version 2.6)

Threads sind Scheduling-Einheiten des Systems

→ one-to-one mapping

Scheduling ist fair

CPUs werden Threads zugeteilt

erzeugen und umschalten von Threads ist teuer

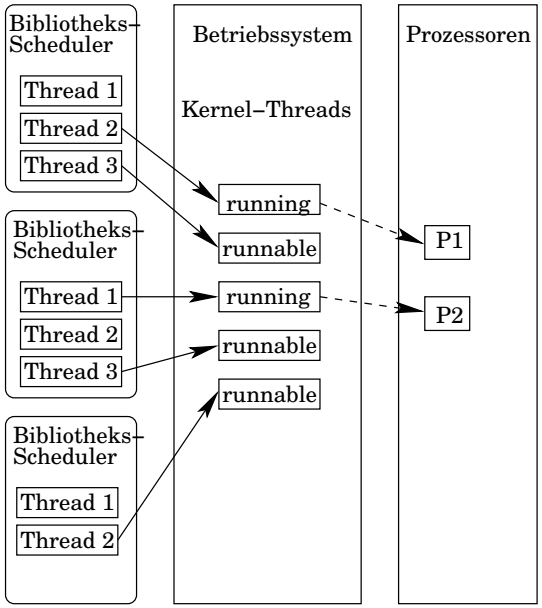
viele Threads belasten das System

two-level Scheduler

Im Solaris-Betriebssystem wurde hybrider Ansatz gewählt, um die Vorteile des einen Ansatzes nicht zu Nachteilen des anderen Ansatzes werden zu lassen:

- dynamische Zuordnung von User- zu Kernel-Threads:
  * jedem CPU-aktiven Thread wird ein Kernel-Thread zugeordnet
  * Threads, die oft schlafen oder auf Ereignisse warten, werden keinem Kernel-Thread zugeordnet
- die Thread-Bibliothek und der Kernel enthalten Datenstrukturen zur Repräsentation von Threads
- Thread-Bibliothek bildet Benutzer-Threads auf Kernel-Threads ab

two-level Scheduler (2)



Server-Klassifikation

Unterscheidung nach Anzahl der Dienste pro Server:

- Ein **shared Server** stellt für jeden Service/Dienst einen eigenen Prozess bereit. Bei Anfrage testen, ob Prozess aktiv ist und ggf. Prozess starten. Prozess kann zur Erledigung der Anforderung einen Thread starten.
- Ein **unshared Server** stellt nur einen Dienst bereit, verschiedene Dienste liegen auf unterschiedlichen Servern (nicht notwendigerweise auf verschiedenen Rechnern).

In beiden Fällen bleibt der Server nach der ersten Anfrage aktiv und kann weitere Anfragen entgegennehmen.

Server-Klassifikation (2)

Ein Server muss aktiv sein, um Client-Anfragen beantworten zu können. Aktivierung ist abhängig von der Art der Server:

- Ein **per request Server** startet jedesmal neu, wenn eine Anfrage eintrifft. Mehrere Prozesse für den gleichen Dienst können konkurrent aktiv sein.
- Ein **persistent Server** wird beim booten des Systems oder beim Start der Server-Applikation aktiv. Client-Anfragen werden direkt an den Server-Prozess weitergeleitet oder bei nicht gestartetem Server vom BS mit einer Fehlermeldung beantwortet.

Server-Klassifikation (3)

Client-Anfragen können Änderungen an den vom Server verwalteten Daten und Objekten bewirken. Dies kann sich wiederum auf nachfolgende Client-Anfragen auswirken:

- **zustandsinvariante** Server liefern Informationen, die sich zwar ändern können, die aber unabhängig von Client-Anfragen sind. Beispiele: Web-, FTP-, Name- und Time-Server. Reihenfolge der Anfragen spielt keine Rolle.
- **zustandsändernde** Server wechseln zwischen 2 Client-Anfragen ihren Zustand, wodurch evtl. Anforderungen von Clients nicht mehr erfüllt werden können (z.B. File-Server: löschen einer Datei). Reihenfolge der Anfragen ist zur Erledigung der Aufgaben entscheidend.

Server-Klassifikation (4)

Unterscheide zustandsändernde Server danach, ob neuer Zustand gespeichert wird oder nicht.

- **zustandslose** (stateless) Server müssen vom Client bei jeder Anfrage die komplette Zustandsinformation erhalten, um die Anforderung korrekt erfüllen zu können. Vorteile beim Absturz des Servers!?
- **zustandsspeichernde** (stateful) Server speichern Zustände in internen Zustandstabellen (Gedächtnis). Vorteile: Clients müssen den Zustand nicht in jeder Anfrage mitteilen (reduzierte Netzlast), Server kann auf zukünftige Anfragen schließen und entsprechende Operationen im Voraus durchführen.

Server-Klassifikation (5)

Beispiel: zustandsspeichernder File-Server

- öffnet ein Client eine Datei, liefert der Server einen Verbindungsidentifier zurück, der eindeutig ist für die Datei und den zugehörigen Clients: wer hat die Datei geöffnet, gegenwärtige Dateiposition, letzter geschriebener/gelesener Satz, ...
- nachfolgende Zugriffe nutzen den Identifier zum Zugriff auf die Datei (reduziert die Nachrichtenlänge, da der Zustand der Datei nicht übertragen werden muss)
- Zustand enthält Informationen darüber, ob die Datei für sequentiellen, direkten oder index-sequentiellen Zugriff geöffnet wurde $\rightarrow$ durch vorausschauendes Lesen evtl. den nächsten Block bereitstellen

Client-Caching

Zeiten für zukünftige Zugriffe auf dieselbe Datenmenge verkürzen, indem eine Kopie der Daten lokal beim Client gespeichert wird.

- Beispiel: bei File- und Web-Servern werden komplette Dateien im Cache abgelegt (Dokumente, Bilder, Audio-Dateien, ...)
- Caching im Hauptspeicher oder auf Platte? Frage des Platzes gegenüber der Performance. Stehen überhaupt Platten zur Verfügung (diskless clients)?
- größenbeschränkter Cache: lagere die am längsten nicht benutzen Daten aus (least recently used). Dirty-Bit zeigt an, ob Daten überschrieben werden können, oder modifizierte Daten zunächst abzuspeichern sind.

Client-Caching (2)

Enthält der Cache gültige Daten?

- zustandsinvariante Server: die vom Client angestoßenen Operationen ändern die Daten nicht, aber die Server-Daten können sich unabhängig vom Client ändern (z.B. ersetzen einer Web-Seite)

$\Rightarrow$ Verfallsdatum für Cache-Eintrag, oder vorm Verwenden der Daten (einmal pro Sitzung, bei jedem Zugriff auf die Seite, ...) deren Gültigkeit vom Server bestätigen lassen (Datum der letzten Modifikation, Versionsnummer der Datei, Checksumme, ...)

Client-Caching (3)

Enthält der Cache gültige Daten?

- zustandsändernde Server: Betrachte File-Server und 2 Clients, die je eine Kopie derselben Datei besitzen und modifiziert haben
  * ein dritter Client liest die Originaldatei, erhält aber nicht die Änderungen von Client 1 und 2
  * die Datei wird von beiden Clients zurückgeschrieben: das Ergebnis hängt davon ab, welcher Client die Datei zuletzt schließt

$\Rightarrow$ Algorithmen wie write through, delayed write oder write on close lösen nur (bedingt) das erste Problem

Client-Caching (4)

write through: bei Änderung eines Cache-Eintrags neuen Wert auch an Server schicken, Server vollzieht Änderung an Originaldatei $\rightarrow$ **erhöhte Netzlast, kein Vorteil durch Caching!** Client sieht Änderungen anderer Clients nicht, wenn alter Cache-Eintrag existiert (Gültigkeit prüfen)

Beispiel:

- P_1 auf A liest f ohne Modifikation
- P_2 auf B ändert f und schreibt es zurück
- P_3 auf A liest alten Cache-Eintrag

delayed write: zur Reduktion des Netzverkehrs mehrere Änderungen sammeln und nach Ablauf eines Zeitintervals (30s bei SUN NFS) an Server schicken

Client-Caching (5)

write on close: weitere Reduktion, wenn Änderungen erst beim Schließen der Datei an Server geschickt werden

Probleme:

- nicht fehlertolerant bei Client-Crash: alle noch nicht geschriebenen Daten gehen verloren
- bei *delayed write* und *write on close* bekommt ein dritter Client die Änderungen von Client 1 und 2 ggf. nicht mit
- die Konsistenz zwischen Client- und Server-Daten ist nicht sichergestellt. Beispiel: Client 1 und 2 löschen aus derselben Datei jeden zweiten Datensatz. Ergebnis?

Lösung: zentrale Kontollinstanz

Client-Caching (6)

Zwei Möglichkeiten der **Zentralen Kontrollinstanz:**

1. Server tabelliert, welcher Client welche Datei zum Lesen oder Schreiben geöffnet hat:
   - zum Lesen geöffnete Datei kann anderen Clients zum Lesen bereitgestellt werden
   - Zugriff auf zum Schreiben geöffnete Datei wird anderen Clients untersagt (Anfrage verweigern oder in Warteschlange stellen)
2. Sobald ein Client eine Datei zum Schreiben öffnet, allen lesenden Clients Nachricht zum Löschen der Cache-Einträge schicken. Schreib-/Lesezugriffe gehen ab dann zum Server. $\Rightarrow$ Leser und Schreiber können parallel weiterarbeiten, aber Server muss Clients benachrichtigen.

Verteilte Prozesse

Nachteile bei Client/Server-Systemen:

- Single Point of Failure: Ausfall des zentralen Servers bedeutet Ausfall aller Dienste
- keine Skalierung: Server ist in seiner Leistung begrenzt

Ausweg: den Server (für die Anwendungen transparent) replizieren $\Rightarrow$ die Server müssen Daten-Konsistenz sicherstellen und sich untereinander koordinieren, ein Server wird zum Client eines anderen Servers

Verteilte Prozesse: können Client und Server sein

Client/Server-Server

Zwischen den Clients und mehreren Servern kann ein weiterer Server plaziert sein.

Man unterscheidet folgende Aufgaben:

- **Proxy:** Stellvertreter für mehrere Server
- **Broker:** vermittelt zwischen Client und Server
- **Trader:** sucht den am besten geeigneten Server heraus
- **Balancer:** spezieller Trader, verteilt die Arbeitslast auf die Server
- **Agent:** koordiniert mehrere Server-Anfragen für Client

Proxy

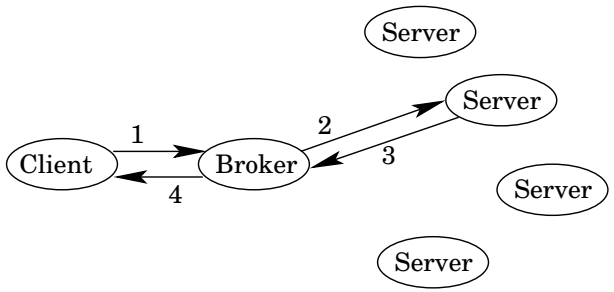
- geeignet für zustandsinvariante Server
- ordne nicht jedem Client einen eigenen Cache zu
- stattdessen: mehrere Clients verwenden einen gemeinsamen Cache, den des zwischen Client und Server liegenden Servers
- der dazwischenliegende Server ist Server für die Clients und ein Client für die Server
- Beispiel: HTTP-Proxy `squid`

Broker

- enthält Informationen über die Services von Servern
- alle Server des verteilten Systems registrieren ihre angebotenen Dienste und Eigenschaften beim Broker
- Ortstranzparenz: Clients wissen nichts über die Server
- Broker wird oft zur Adressverwaltung erweitert
- 3 Typen: intermediate oder forwarding broker, separate oder handle-driven broker, hybrider broker

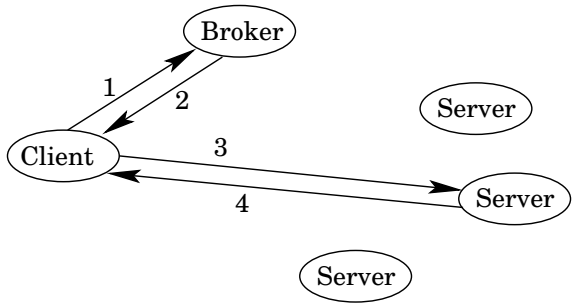
Broker (2)

intermediate oder **forwarding broker**: Client-Anfrage an den betroffenen Server übergeben, Rückantwort annehmen und an Client leiten



Broker (3)

separate oder **handle-driven broker**: Broker gibt Server-Informationen an Client zurück, Client kommuniziert direkt mit dem Server



Broker (4)

hybrider Broker: Unterstützt beide genannten Versionen. Der Client gibt an, welches Modell er wünscht.

Nachteil: Broker ist Single Point of Failure

⇒ Broker mehrfach auslegen!

Im Extremfall ist jedem Client ein Broker zugeordnet.
Vorteil: Ausfall eines Brokers betrifft nur einen Client.
Nachteil: Aufwendige Konsistenzhaltung der Datenbasis.

Trader, Balancer und Agenten

Trader: verteilt Client-Anfragen, falls mehrere Server für einen Dienst zur Verfügung stehen, die Dienste jedoch unterschiedliche Qualität haben. Unterscheide 3 Typen wie beim Broker.

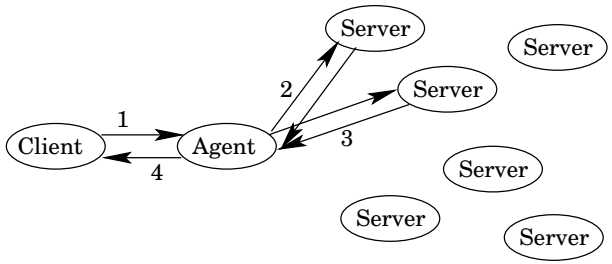
Balancer: spezieller Trader, dessen einziges Entscheidungskriterium die Last der Server ist

Agent: Clients, die mehrere Dienste abwickeln wollen oder komplexere Dienste in Anspruch nehmen, benötigen eine Ablaufplaner. Anfrage in mehrere verschiedene Teilanfragen zerlegen, Antworten sammeln und eine einzige Rückantwort an Client schicken

Agenten (2)

Ziele:

- mehrere Services hinter einem Service verstecken
- Performance: Teilanfragen parallel durch mehrere Server bearbeiten (Divide and Conquer, Master/Slave)



Agent kennt die Server nicht, deren Dienste er in Anspruch nehmen will: Broker zwischen Agent und Server schalten

Agenten (3)

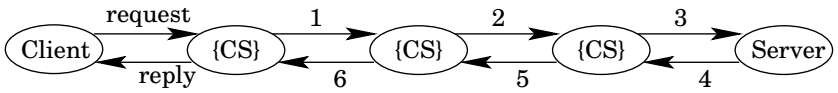
Agent kann Teilanfragen iterativ oder parallel ausführen:

- iterativ: die angeforderten Dienste bauen aufeinander auf und müssen in einer bestimmten Reihenfolge bearbeitet werden. Ein neu anzufordernder Dienst kann von dem Ergebnis eines vorhergehenden Dienstes abhängen.
- parallel: falls Dienste unabhängig voneinander sind oder alle Repliken eines replizierten Servers benachrichtigt werden müssen

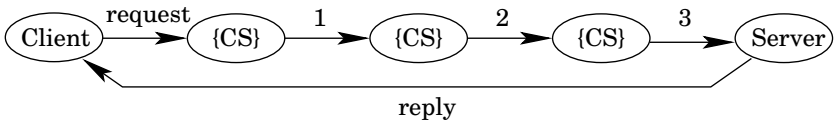
Nachteil: Agent ist Single Point of Failure.
⇒ verteile die Agenten-Funktionalität auf die Server: jeder Server bestimmt seinen Nachfolge-Server

Client/Server-Ketten

rekursiv: Anforderungen werden an den nächsten verteilten Prozess {CS} weitergereicht, die Rückantworten werden an den verteilten Prozess der vorhergehenden Rekursionsstufe zurückgereicht

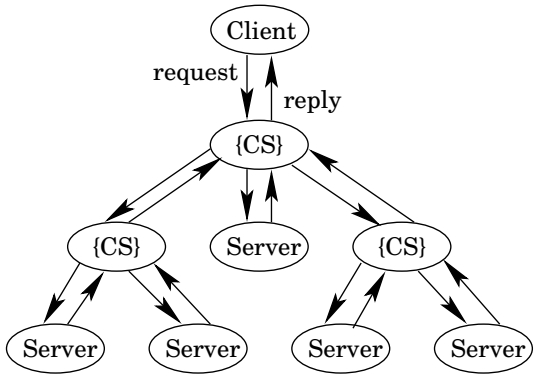


transitiv: Anforderungen werden an den nächsten Prozess weitergereicht, die Rückantwort des rekursionsbeendenden Prozesses wird an den Client zurückgeschickt



Client/Server-Bäume

ist die Anzahl der Nachfolgeprozesse größer als eins, so ergeben sich Baumstrukturen:



die einzelnen Anforderungsketten innerhalb des Baumes können wieder rekursiv oder transitiv sein

Client/Server-Ketten und -Bäume

bei mehreren Nachfolgeknoten enthält jeder Prozess {CS} einen internen Agenten, der die nachfolgenden (iterativen oder parallelen) Server-Anfragen koordiniert.

Anwendungen: verteilte Berechnungen

- rekursive Kette: parallele Prozesse, die durch einen Kommunikationskanal oder eine Pipe verbunden sind
- transitive Kette: ring oder token-basierte Algorithmen (z.B. Wahl-Algorithmus: alle Prozesse einigen sich darauf, wer der neue Koordinator sein soll)