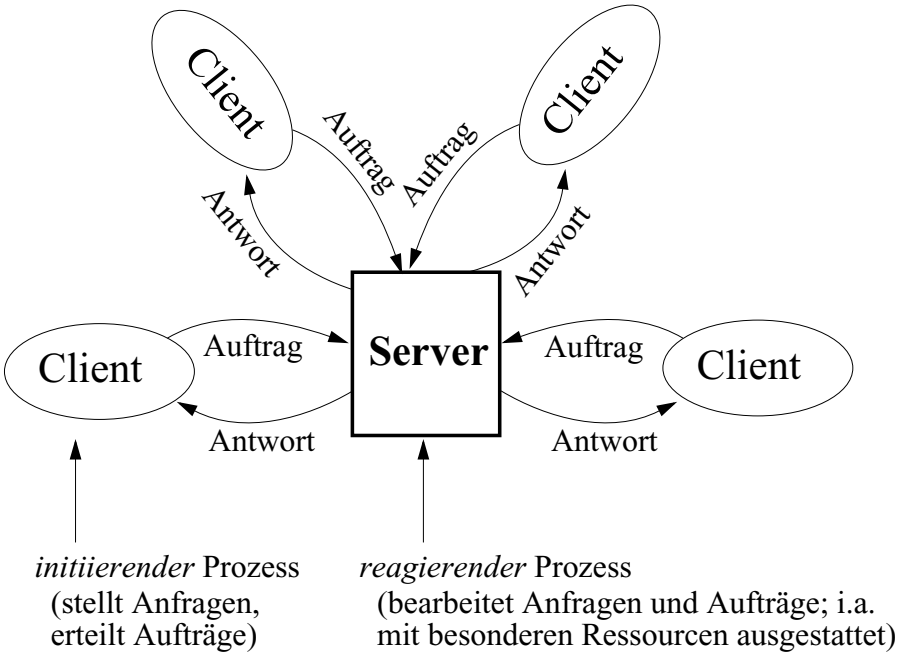
Client/Server-Modell

Das Client/Server-Modell



- Aufgabenteilung und asymmetrische Struktur
  - *Clients*: typischerweise Anwendungsprogramme und graphische Benutzungsschnittstelle (“front end”)
  - *Server*: zuständig für Dienstleistungen
- Typisches Kommunikationsparadigma: RPC

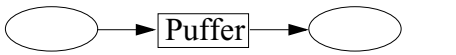
Eignung des Client/Server-Paradigmas

- Populär wegen des eingängigen Modells
  - entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
  - gewohntes Muster → intuitive Struktur, gute Überschaubarkeit
- Effizienz durch spezialisierte „Dienstleister“
  - grosszügige Ausstattung (CPU-Leistung, Speicherkapazität usw.)
  - bestückt mit spezieller Software (Datenbank etc.)
- Kosteneffektivität durch bessere Auslastung wertvoller Ressourcen bei “Compute Server”
  - Clients brauchen oft kurzfristig Spitzenleistung
  - einzelner Client kann Ressourcen aber nicht dauerhaft auslasten
- Passend für viele Kooperationsbeziehungen, z.B.
  - Client erbittet Auskunft von einem spezialisierten Service
  - gefährdete Clients geben wertvolle Daten in Obhut des (gegen Missbrauch, Verlust, Diebstahl usw.) hoch gesicherten Servers

---

- Modell ist für viele Zwecke geeignet, jedoch nicht für alle (z.B. Pipelines, “peer-to-peer”, asyn. Mitteilung)!

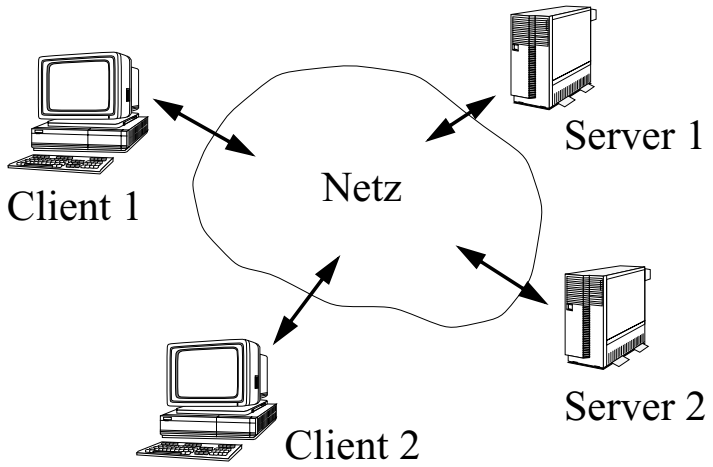
Beispiel Puffer als Pipeline:



- Puffer ist weder Client noch Server, sondern hat beide Rollen! (passiv gegenüber Produzent; aktiv gegenüber Konsument)
- Inversion der Kommunikationsbeziehung bei einem Puffer-Server!

Client- und Server-Maschinen

- Übertragung des Client/Server-Modells auf *Rechner*

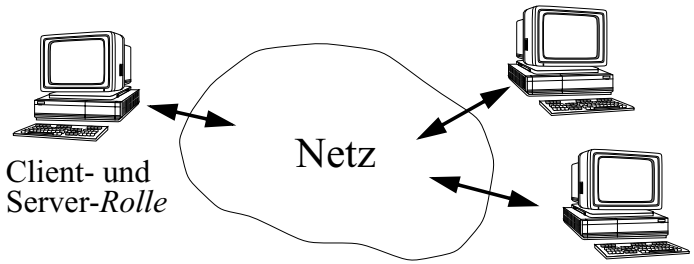


- Typischerweise PCs als Clients
  - u.a. mit graphischem Benutzungsinterface
- Andere, leistungsfähigere Rechner als Server
  - “zentrale” Dienste (z.B. Speicherserver)
  - gemeinsam benutzte Betriebsmittel
- Im allgemeinen müssen sich aber Server- und Client-*Prozesse* nicht auf dedizierten Rechnern befinden!

Peer-to-Peer-Strukturen

"Gleichrangiger"

- Im Gegensatz zum asymmetrischen Client/Server-Modell



- Jeder Client fungiert zugleich als Server für seine Partner
  - → keine (teuren) dedizierten Server notwendig
  - → oft als Billiglösung von "echtem" Client/Server-Computing angesehen

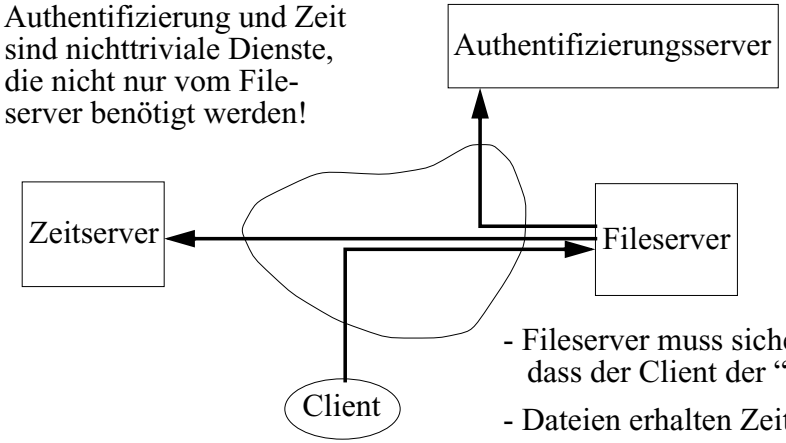
- In der Reinform keine zentralisierten Elemente
  - dies wird gelegentlich in "politischer" Weise artikuliert (vgl. Tauschbörsen)

- *Nachteile*:
  - "Anarchischer" als *maschinenbezogene* Client/Server-Architektur
  - Rechner müssen leistungsfähig genug sein (cpu-Leistung, Speicherausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Besitzer kann seine Maschine ausschalten...)
  - Datensicherung muss ggf. dezentral durchgeführt werden
  - Sicherheit und Schutz kritisch: Lizenzen, Viren, Integrität...

Client/Server-Rollen

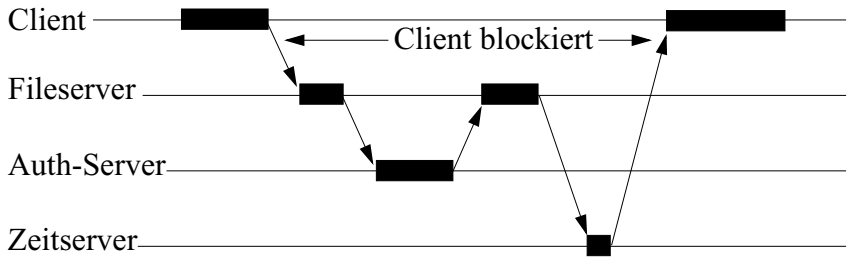
- Server müssen ggf. zur Durchführung eines Dienstes die Dienstleistungen anderer Server in Anspruch nehmen

- Authentifizierung und Zeit sind nichttriviale Dienste, die nicht nur vom Fileserver benötigt werden!



- Fileserver muss sicherstellen, dass der Client der "richige" ist
- Dateien erhalten Zeitstempel

- Fileserver hat prinzipiell die Rolle eines Servers, zwischenzeitlich jedoch die Rolle eines Clients



Zustandsändernde /-invariante Dienste

(Vgl. frühere Diskussion bzgl. RPC-Fehlersemantik!)

- Verändern Aufträge den Zustand des Servers?
  - nur kurzzeitig (→ Atomarität) oder langfristig / dauerhaft?
- Typische *zustandsinvariante* Dienste:
  - Auskunftsdienste (z.B. Name-Service)
  - Zeitservice
- Typische *zustandsändernde* Dienste:
  - Datei-Server

Idempotente Dienste / Aufträge

- Wiederholung eines Auftrags liefert gleiches Ergebnis

nicht zustandsinvariant!

  - Beispiel: “Schreibe in Position 317 von Datei XYZ den Wert W”
  - Gegenbeispiel: “Schreibe ans Ende der Datei XYZ den Wert W”
  - Gegenbeispiel: “Wie spät ist es?” ← aber zustandsinvariant!

Wiederholbarkeit von Aufträgen

- Bei Idempotenz oder Zustandsinvarianz kann bei Verlust des Auftrags (timeout beim Client) dieser erneut abgesetzt werden (→ einfache Fehlertoleranz)

Zustandslose / -behaftete Server

stateless — statefull

“session”

- Hält der Server Zustandsinformation über Aufträge hinweg?
  - z.B. (Protokoll)zustand des Clients
  - z.B. Information über frühere damit zusammenhängende (Teil)aufträge
- Aufträge an zustandslose Server müssen autonom sein

- *Beispiel*: Datei-Server

```
open(“XYZ”);
read;
read;
close;
```

In klassischen Systemen hält sich das Betriebssystem Zustandsinformation, z.B. über die Position des Dateizeigers geöffneter Dateien

  - bei zustandslosen Servern entfällt open/close; jeder Auftrag muss vollständig beschrieben sein (Position des Dateizeigers etc.)
  - zustandsbehaftete Server daher i.a. effizienter ← notw. Zustandsinformation ist beim Client
  - Dateisperren sind bei echten zustandslosen Servern nicht (einfach) möglich
  - zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. durch Speichern von Sequenznummern) → Idempotenz

- *Crash* eines Servers: Weniger Probleme im zustandslosen Fall (→ Fehlertoleranz)! ← entscheidender Vorteil!

- NFS (Network File System von Sun): zustandslos
- RFS (Remote File System von UNIX System V): zustandsbehaftet

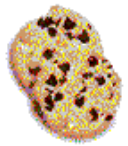
Sind WWW-Server zustandslos?

- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - jeder Hyperlink, den man anklickt, löst eine neue “Transaktion” aus
- Stellt ein Problem beim E-Commerce dar
  - gewünscht sind Transaktionen über mehrere clicks hinweg und
  - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
  - erforderlich z.B. für Realisierung von “Einkaufskörben” von Kunden
  - gewünscht vom Marketing (Verhaltensanalyse von Kunden)

Lösungsmöglichkeiten (für Einkaufskörbe im WWW)

- IP-Adresse des Kunden an Auftrag anheften
  - Problem: Proxy-Server → viele Kunden haben gleiche IP-Adresse
  - Problem: dynamische IP-Adressen → keine Langzeitwiedererkennung
- “tag propagation”
  - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig aufruft
  - diese Nummer jedem link anheften und mit zurückübertragen
- Cookies
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - nur der Sender des Cookies darf dieses später wieder lesen

Cookies



Auszug aus
http://home.netscape.com/newsref/std/cookie_spec.html:

Cookies are a general mechanism which server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection. The addition of a simple, persistent, client-side state significantly extends the capabilities of Web-based client/server applications.

A server, when returning an HTTP object to a client, may also send a piece of state information which the client will store... Any future HTTP requests made by the client... will include a transmittal of the current value of the state object from the client back to the server. The state object is called a cookie, for no compelling reason.

This simple mechanism provides a powerful new tool which enables a host of new types of applications to be written for web-based environments. Shopping applications can now store information about the currently selected items, for fee services can send back registration information and free the client from retyping a user-id on next connection, sites can store per-user preferences on the client, and have the client supply those preferences every time that site is connected to.

A cookie is introduced to the client by including a Set-Cookie header as part of an HTTP response... The expires attribute specifies a date string that defines the valid life time of that cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out... A client may also delete a cookie before it’s expiration date arrives if the number of cookies exceeds its internal limits.

- Denkübung: Müssen Proxy-Server geeignete Massnahmen vorsehen?
- Übung: Man finde heraus, was doubleclick.net macht (und wie)

Cookies (2)

- Anwendung von cookies war und ist umstritten (Ausspionieren des Verhaltens); dazu kam eine gewisse Paranoia:

  http://www.cookiecentral.com/creport.htm

  http://www.ciac.org/ciac/bulletins/i-034.shtml

  The Energy Department's Computer Incident Advisory Capability (CIAC) recently issued a report on cookie technology and its use on the web....

  The report stressed that there's a sense of paranoia involved with cookies, cookies cannot harm your computer or pass on private information such as an email address without the user's intervention in the first place. Paranoia has recently been sparked by one rumour involving AOL's new software, it claimed that AOL were planning to use cookies to obtain private information from users hard drives.

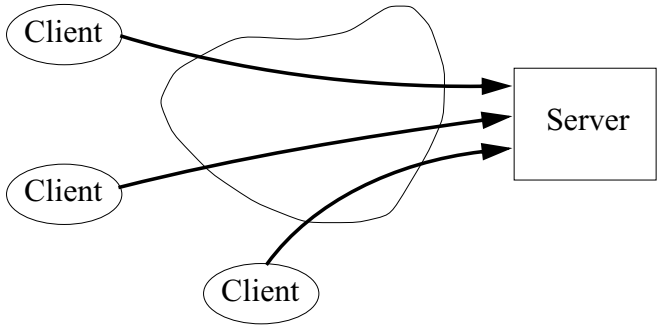
- Problemlos ist das allerding nicht:

  http://www.cookiecentral.com/cookie5.htm

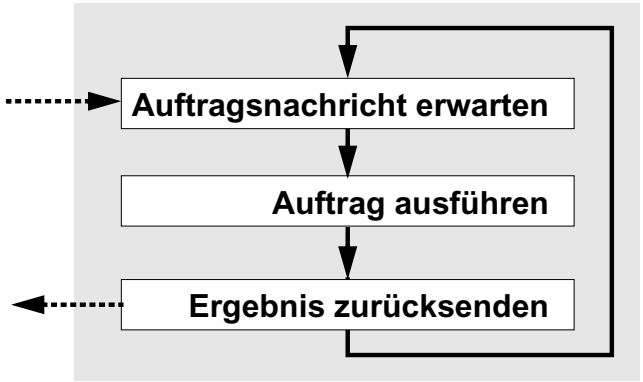
  Unfortunately, the original intent of the cookie has been subverted by some unscrupulous entities who have found a way to use this process to actually track your movements across the Web. They do this by surreptitiously planting their cookies and then retrieving them in such a way that allows them to build detailed profiles of your interests, spending habits, and lifestyle... it is rather scary to contemplate how such an intimate knowledge of our personal preferences and private activities might eventually be used to brand each of us as members of a particular group.

Iterative Server

- Problem: Viele "gleichzeitige" Aufträge



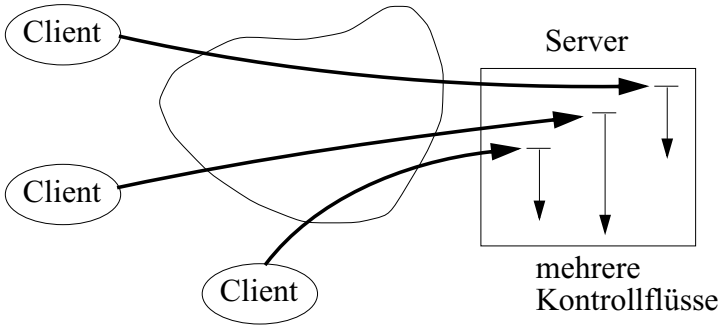
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- häufige Bezeichnung: "single threaded"
- eintreffende Anfragen während Auftragsbearbeitung: abweisen, puffern oder einfach ignorieren
- einfach zu realisieren
- bei "trivialen" Diensten sinnvoll (mit kurzer Bearbeitungszeit)

Konkurrente ("nebenläufige") Server

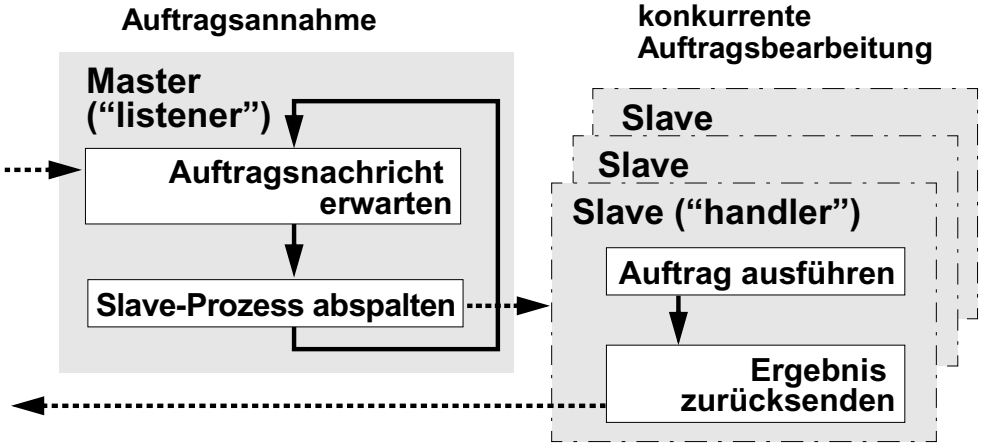
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei langen Aufträgen (z.B. in Verbindung mit E/A)
  - Beispiel: Web-Server oder Suchmaschinen



- Ideal bei Mehrprozessormaschinen (physische Parallelität)
  - aber auch bei Monoprozessor-Systemen (vgl. Argumente bei Timesharing-Systemen): Nutzung erzwungener Wartezeiten eines Auftrags für andere Jobs; kürzere mittlere Antwortzeiten bei Jobmix aus langen und kurzen Aufträgen
- Interne Synchronisation bei konkurrenten Aktivitäten sowie ggf. Lastbalancierung beachten
- Verschiedene denkbare Realisierungen, z.B.
  - mehrere Prozessoren
  - Verbund verschiedener Server-Maschinen (z.B. LAN-Cluster)
  - dynamische Prozesse (bei Monoprozessor-Systemen)
  - dynamische threads
  - feste Anzahl vorgegründeter Prozesse
  - internes Scheduling und Multiprogramming

Konkurrente Server mit dynamischen Handler-Prozessen

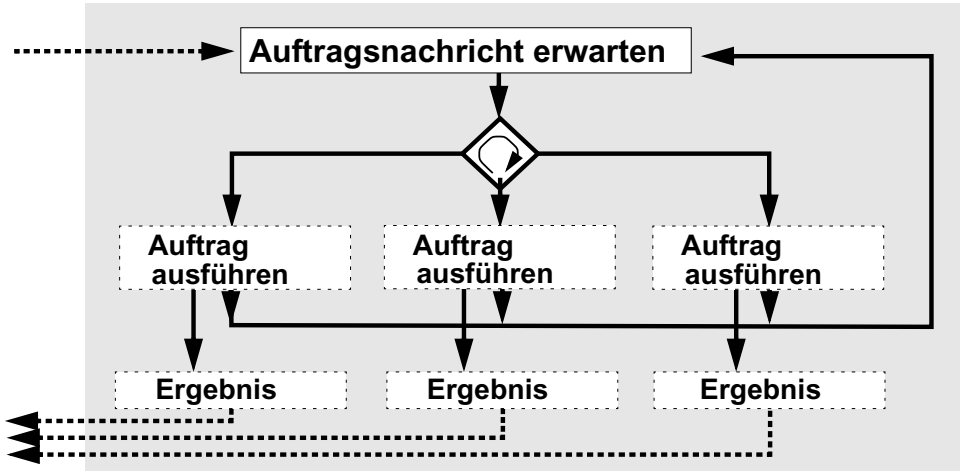
- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave ("handler") übernimmt den Auftrag
  - Client kommuniziert dann ggf. direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind ggf. Leichtgewichtsprozesse ("thread")
  - Slaves terminieren i.a. nach Beendigung des Auftrags
  - die Anzahl gleichzeitiger Slaves sollte begrenzt werden



- Alternative: "Process preallocation": Feste Anzahl statischer Slave-Prozesse
  - ggf. effizienter (u.a. Wegfall der Erzeugungskosten)
- Übungsaufgaben:
  - herausfinden, wie es bei Web-Servern gemacht wird (z.B. Apache)
  - wie sollte man bei grossen WWW-Suchmaschinen vorgehen?

Quasi-konkurrente Server

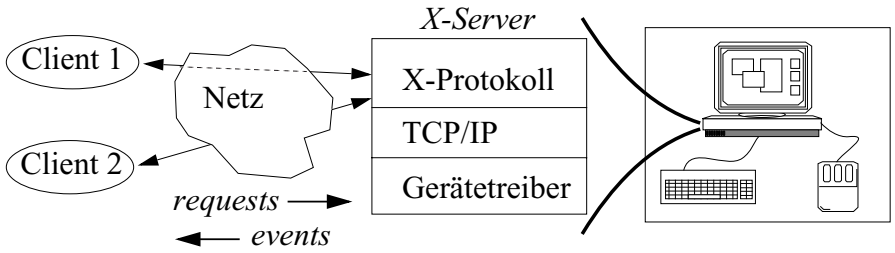
- Server besteht aus einem *einzigen Prozess*, der im Multiplexmodus mehrere Aufträge verschränkt abarbeitet
  - ggf. sinnvoll, wenn z.B. Clients grosse Datenmengen "stückweise" senden und die Wartezeiten dazwischen für die Bearbeitung der anderen Aufträge verwendet werden kann



- Keine Neugründung von Slave-Prozessen
- Keine Adressraumgrenzen zwischen Auftragsdaten
  - keine kostspieliger Kontextwechsel
  - auftragsübergreifende gemeinsame Datenhaltung effizienter (vgl. X-Server: Alle Clients schreiben Display-Daten in einen gemeinsamen "Display-Puffer")
- Potentielle Nachteile: kein Adressraumschutz zwischen verschiedenen Aufträgen; ggf. unnötige Wartezeiten z.B. bei blockierenden Betriebssystemaufrufen

"X-Window" als Client/Server-Modell

- Erstes netzwerkunabhängiges Graphik- und Fenstersystem für seinerzeit neue pixelorienterte Bildschirme
  - entwickelt Mitte der 80er Jahre am MIT, zusammen mit der Firma DEC



- i.a. bedient ein Server mehrere Client-Prozesse ("Applikationen"), die ihre Ausgabe auf dem gleichen Bildschirm erzeugen
- *Window-Manager*: Spezieller Client, der Grösse und Lage der Fenster und Icons steuert (Beispiele: twm, mwm, fvwm)

→ X windows system protocol (über TCP)

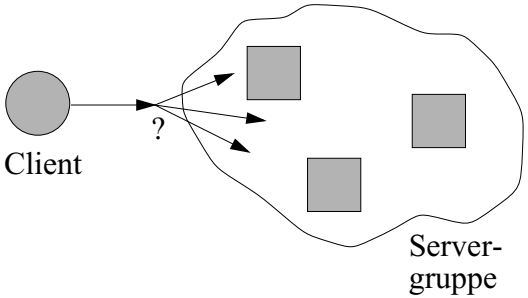
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete Ressourcen (Window, font...); höhere Funktionen (z.B. Dialogboxen) in einem (von mehreren) X-Toolkit
- *Events*: Tastatur- und Mauseingaben (bzw. -bewegungen) werden vom X-Server asynchron an den Client des "aktiven Fensters" gesendet (keine klassische Server-Rolle → schwierig mit RPCs zu realisieren!)
- X ist ein *verteiltes System*: Client-Prozesse können sich auf verschiedenen Rechnern befinden
- *X-Terminal* hatte Server-Software im ROM bzw. lädt sie beim Booten (heute gegenüber PC preislich kaum ein Vorteil, vgl. auch "Web-Terminal")
- vielfältige Standard-*Utilities* und *Tools* (xterm, xclock, xload...)

Servergruppen und verteilte Server

- Idee: Ein Dienst wird nicht von einem einzigen Server, sondern von einer Gruppe von Servern erbracht

a) *Multiple Server*

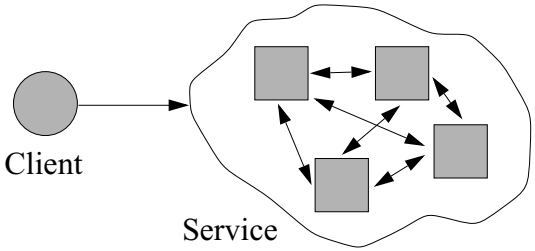
- Jeder einzelne Server kann den Dienst erbringen
- Zweck:
  - *Leistungssteigerung* (Verteilung der Arbeitslast auf mehrere Server) ← "Lastverbund"
  - *Fehlertoleranz* durch Replikation (Verfügbarkeit auch bei vereinzelten Server-Crashs) ← "Überlebensverbund"



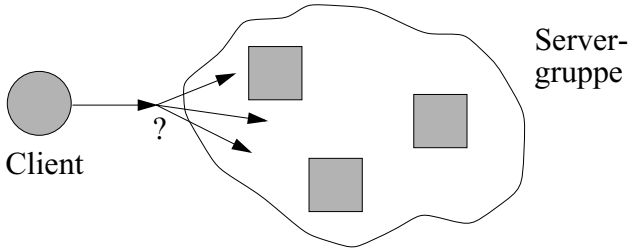
- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

b) *Kooperative Server* ← "Know-how-Verbund"

- ein Server allein kann den Dienst nicht erbringen



Serverwahl bei einem Lastverbund

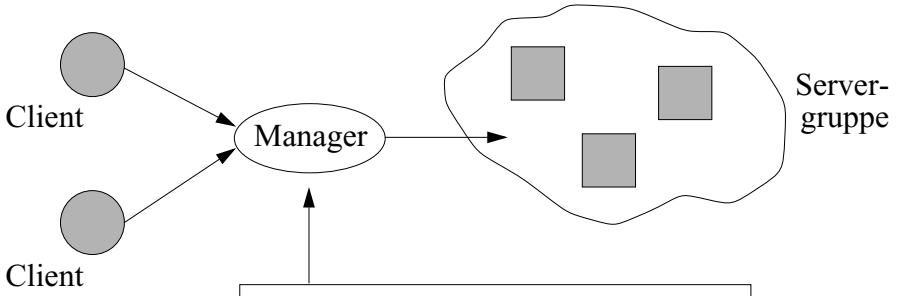


1) *Zufallsauswahl*

- Einfaches, effizientes Protokoll
- Nachteile:
  - Client muss mehrere Server kennen
  - ggf. ungleichmässige Auslastung

Stellen Verfahren mit "round robin"-Einträgen im DNS-System eine solche Zufallsauswahl dar?

2) *Zentraler Service-Manager*

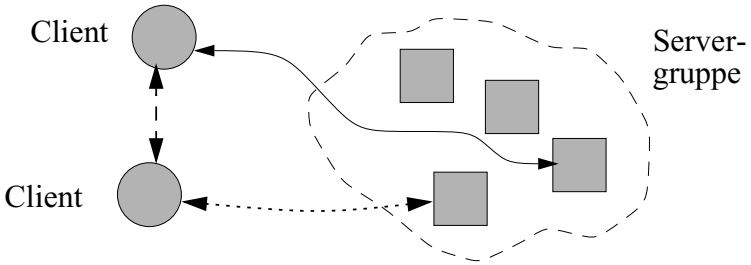


- sorgt für sinnvolle Verteilung (wie?)
- behält ggf. Überblick über Aufträge
- informiert sich ggf. von Zeit zu Zeit über die Server-Lastsituation

- Nachteile:
  - Overhead bei trivialen Diensten
  - ggf. Überlastung des Managers
  - Dienstblockade bei Ausfall des Managers

Serverwahl bei Lastverbund (2)

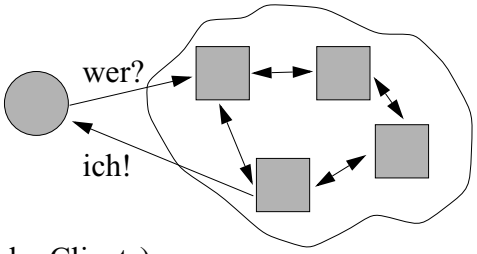
3) *Clients einigen sich untereinander*



- u.U. grosser Kommunikationsaufwand zwischen vielen Clients
- Clients kennen sich i.a. nicht (z.B. bei dynamisch gegründeten)

4) *Server einigen sich untereinander*, wer den Auftrag ausführt

- Abstimmung (aber fehlertolerant wegen möglichen Server-Ausfällen)
- i.a. nur bei wenigen Servern (relativ zur Zahl der Clients)



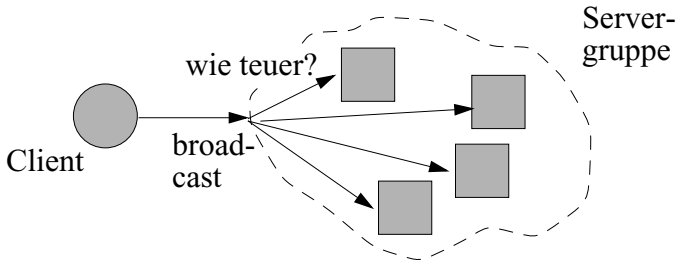
- Server führen Abstimmung diszipliniert durch (verlässlicher als Clients)

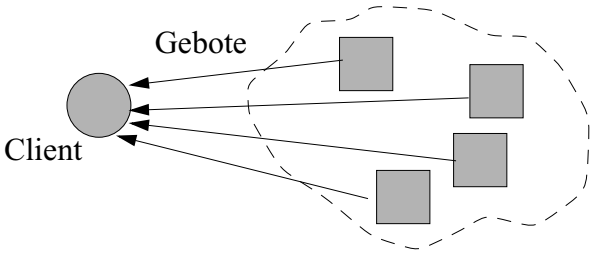
5) *"Anycast"*: der nächstliegende (=?) Server wird von der Netzinfrastruktur (Router etc.) bestimmt

Serverwahl bei Lastverbund (3)

6) *Bidding-Protokoll*

- Client fragt per Broadcast nach Geboten
- Server mit "billigstem" Angebot wird ausgewählt





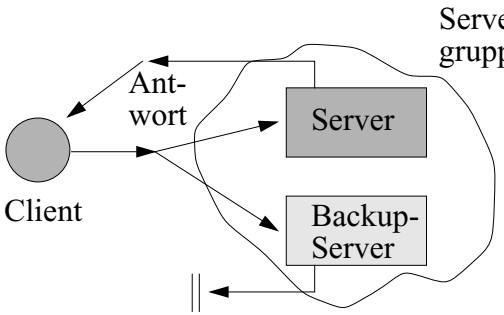
- Variante: nur *Stichprobe* befragen (multicast statt broadcast; sehr kleine Teilmenge von vielen Servern genügt i.a.!)

- Generelles Problem: Lastsituation kann veraltet sein!

Serverreplikation in Überlebensverbunden

1) *Zustandsinvariante Dienste*: im Prinzip einfach - nach Crash anderen Server nehmen...

2) *Zustandsändernde Dienste* (hier "hot stand by"):



aktiver Server

passiver Server im "hot stand-by"-Modus (ggf. mehr als einer)

- im Fehlerfall kann *unmittelbar* auf den Backup-Server umgeschaltet werden
- beachte: Replikation sollte transparent für die Clients sein!
- Auftrag wird per Multicast an alle Server verteilt
- nur die Antwort des aktiven Servers wird zurückgeliefert

Probleme:

(vom Server und Backup-Server)

- evtl. Subaufträge werden mehrfach erteilt → Probleme mit zustandsändernden bzw. gegenseitig ausgeschlossenen Subdiensten
- Reihenfolge der Aufträge muss bei allen Servern identisch sein (→ Semantik von multicast insbesondere bei mehreren Clients)
- Resynchronisation nach einem Crash: Nach Neustart muss ein (passiver) Server mit dem aktuellen Zustand des aktiven Servers initialisiert werden (Zustand kopieren bzw. replay)

Replikation

- Daten mehrfach halten; möglicher Zweck:
  - Effizienzsteigerung: Daten schneller verfügbar machen (z.B. Caches)
  - erhöhte Verfügbarkeit (auch bei Ausfall einzelner Server)
  - Fehlertoleranz durch Majoritätsvotum
- Forderungen: Transparenz und Erfüllung gewisser Konsistenzeigenschaften ("Kohärenz" der Replikate)
- Replikationsmanagement
  - *asynchron*: nur periodische Aktualisierung zwischen den Replikaten (inkohärente Replikate nach Änderung bis zur nächsten Synchronisation)
  - *synchron*: immer kohärente Replikate; logische Sicht eines einzelnen Servers (z.B. hot stand by)
  - Aufwand wächst, je näher man sich dem synchronen Modell annähert
- *Nicht-transparente* Replikation: Client führt Änderungen explizit auf allen Replikaten durch
- *Transparente* Replikation; unterschiedlich realisiert:
  - per Gruppenkommunikation (Semantik und Zuverlässigkeitsaspekte des Kommunikationssystems entscheidend)
  - Hauptserver ("primary"), der Sekundärserver aktualisiert
    - Schreibzugriffe nur beim Primärserver; Lesezugriffe beliebig
    - Hauptserver kann "sofort" oder schubweise ("gossip-Nachricht") die Sekundärserver aktualisieren (Konsistenzproblematik beachten!)
  - symmetrische Server, die sich jeweils untereinander abgleichen
- Voting-Verfahren: zum Schreiben bzw. Lesen auf jeweils mehr als 1+N/2 Server zugreifen
  - Abgleich (voting) bzgl. neuester Versionsnummer