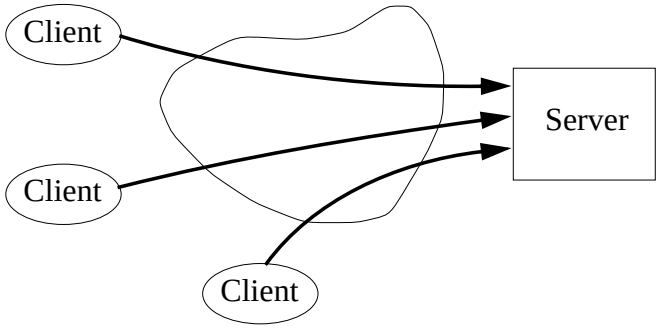
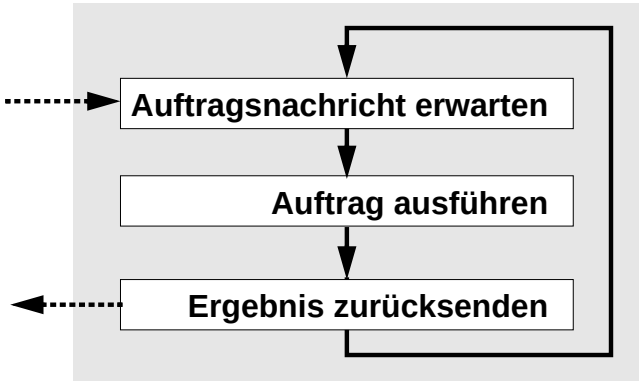
Iterative Server

- Problem: Viele “gleichzeitige” Aufträge



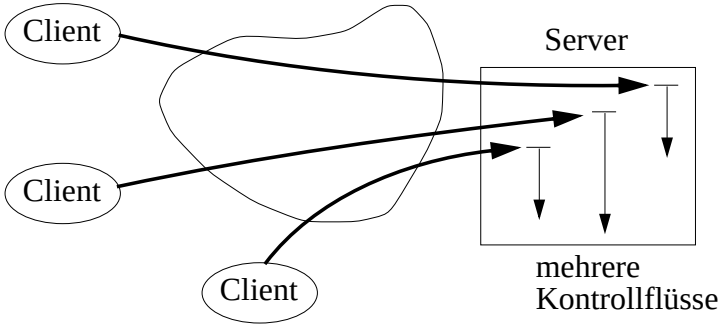
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- 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

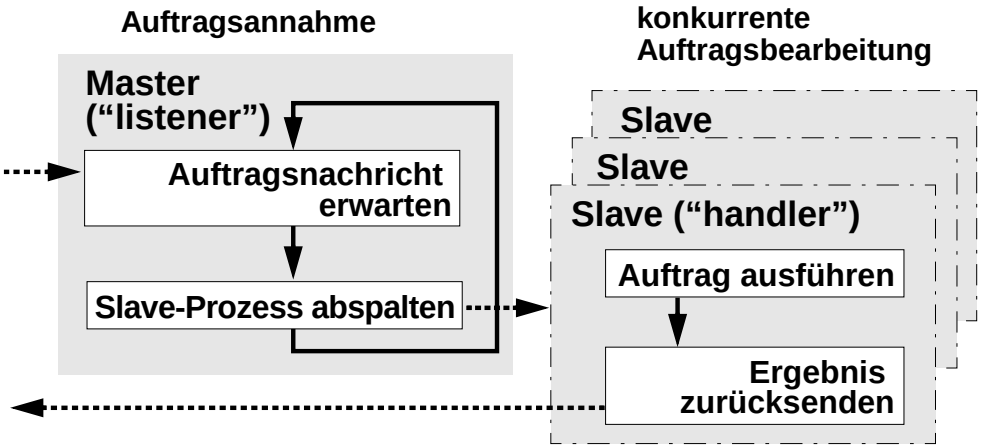
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei langen Aufträgen (z.B. in Verbindung mit E/A)
  - Beispiel: Web-Server oder Suchmaschinen



- 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

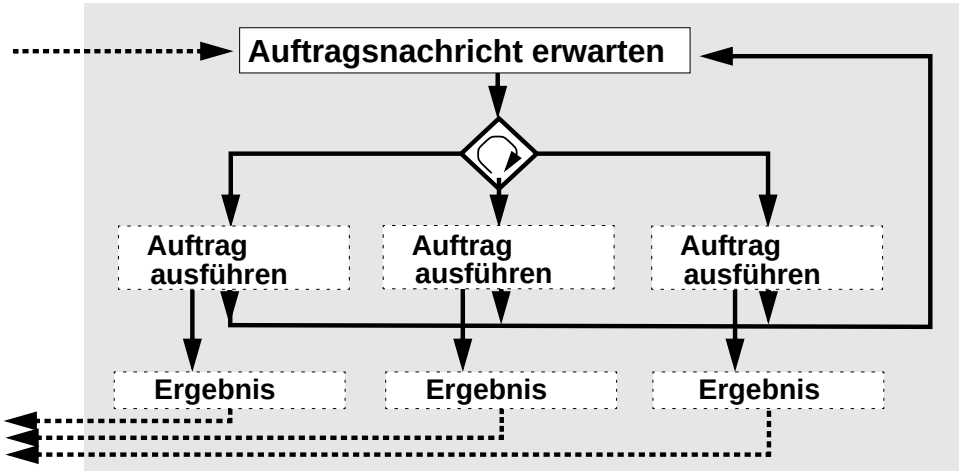
- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave ("handler") übernimmt den Auftrag
  - Client kommuniziert dann ggf. direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind ggf. Leichtgewichtsprozesse ("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

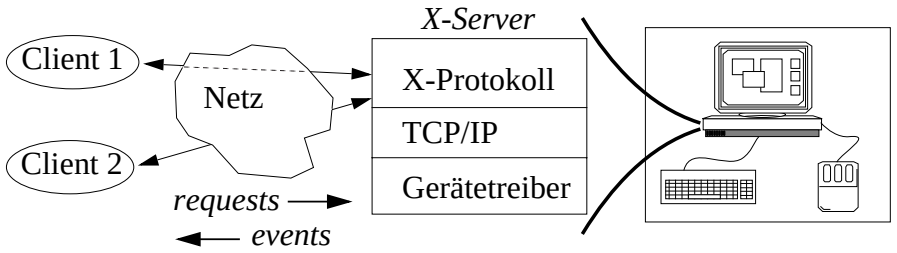
- Server besteht aus einem *einzigen Prozess*, der im Multiplexmodus mehrere Aufträge verschränkt abarbeitet
  - ggf. sinnvoll, wenn z.B. Clients grosse Datenmengen "stückweise" senden und die Wartezeiten dazwischen für die Bearbeitung der anderen Aufträge verwendet werden kann



- Keine Neugründung von Slave-Prozessen
- Keine Adressraumgrenzen zwischen Auftragsdaten
  - keine kostspieliger Kontextwechsel
  - auftragsübergreifende gemeinsame Datenhaltung effizienter (vgl. X-Server: Alle Clients (z.B. xclock) 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)

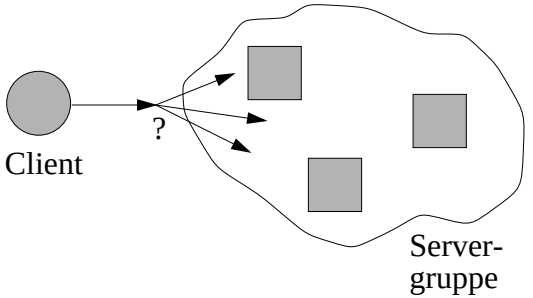
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete 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 (“Fenster verschiedener Rechner”)
- *X-Terminal* hat 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”

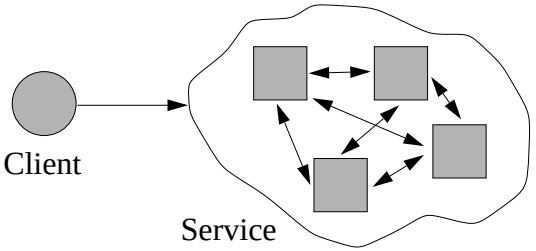


- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

b) *Kooperative Server*

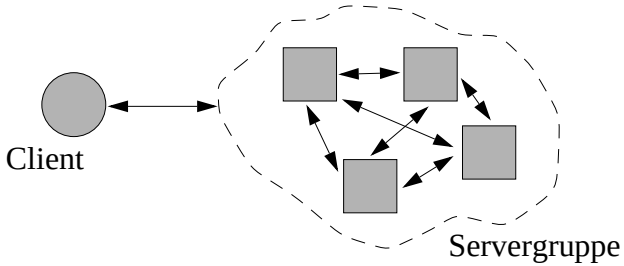
← “Know-how-Verbund”

- ein Server allein kann den Dienst nicht erbringen

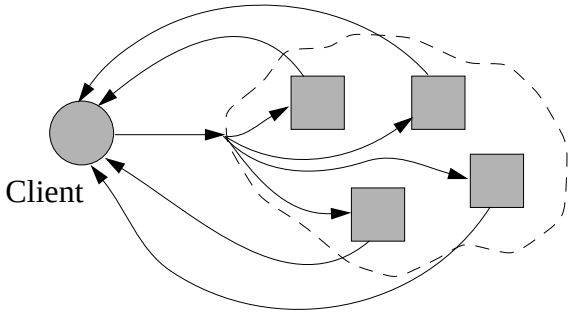


Strukturen kooperativer Server

1) Echte Kooperation: Server liefern gemeinschaftlich ein Gesamtergebnis

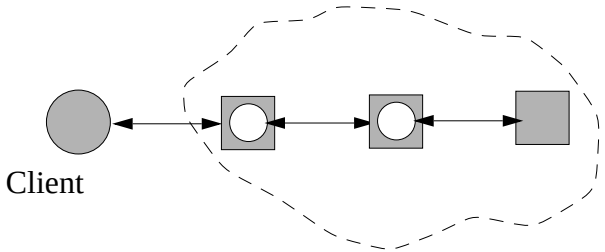


2) Paarweise Kooperation mit dem Client: Client akkumuliert Teilergebnisse



Z.B.: Broadcast stösst einzelne Rück-meldungen an und Client "akkumuliert" die Information

3) Kaskadierung: Dienst als Menge von Teildiensten realisiert, z.B.:



- Server während der Auftragsbearbeitung als Client bzgl. Teilaufträgen

Beispiel: ruptime

- "remote uptime" (vgl. UNIX-Kommando "uptime")
- UNIX-Kommando, das einen verteilten Dienst im LAN implementiert
- heute jedoch nicht mehr aktuell (durch andere Dienste ersetzt)

```
NAME
     ruptime - show host status of local machines

     ruptime gives a status line like uptime for each machine on
     the local network; these are formed from packets broadcast
     by each host on the network once a minute.

     Machines for which no status report has been received for 5
     minutes are shown as being down.

BUGS
     Broadcasting does not work through gateways.
```

Router etc.

```
sol[52] [~] ruptime
cadsun        up 34+12:39,     0 users,  load 1.28, 1.28, 1.06
hssun2      down     1:21
martine       up  5+10:55,     0 users,  load 0.10, 0.05, 0.04
nuriel        up  5+11:04,     0 users,  load 0.11, 0.11, 0.11
octopus       up  5+10:43,     0 users,  load 0.02, 0.04, 0.03
paloma        up  5+07:10,     0 users,  load 0.00, 0.08, 0.06
quantas       up  5+10:52,     0 users,  load 0.00, 0.02, 0.02
rtracer       up 39+13:18,     4 users,  load 2.05, 1.21, 0.52
salamander    up  2+05:18,     0 users,  load 0.00, 0.06, 0.06
sol           up  1+06:27,    11 users,  load 5.12, 5.12, 5.12
```

verschiedene Maschinen

Der rwhod-Dämon

- “Dämon”: Service, der auf das Auftreten von Ereignissen wartet, und dann darauf reagiert; wird i.a. bei Systemstart gegründet.

```
NAME
     rwhod - system status server

DESCRIPTION
     rwhod is the server which maintains the database used by the
     rwho(1C) and ruptime(1C) programs.

     rwhod operates as both a producer and consumer of status
     information.  As a producer of information it periodically
     queries the state of the system and constructs status mes-
     sages which are broadcast on a network.  As a consumer of
     information, it listens for other rwhod servers' status mes-
     sages,...

     Status messages are generated approximately once every 60
     seconds.

BUGS
     This service takes up progressively more network bandwidth
     as the number of hosts on the local net increases. For
     large networks, the cost becomes prohibitive.
```

- kein eigentliches Client/Server-Modell!
- aus Performance-Gründen oft deaktiviert
- Neuimplementierung (‘rup’ statt ‘ruptime’): kein default-Broadcast, sondern nur Broadcast bei Aufruf des Kommandos; rstatd-Dämonen anderer Rechner antworten dann

kernel statistics server

Das rup-Kommando

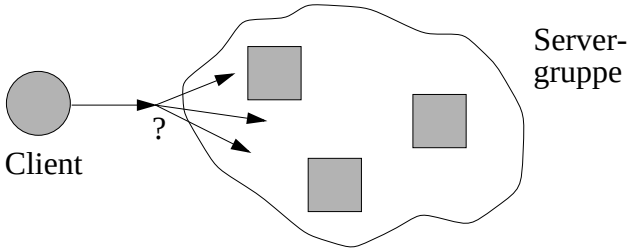
```
DESCRIPTION
     rup gives a status similar to uptime for remote machines.
     It broadcasts on the local network, and displays the
     responses it receives.

     Normally, the listing is in the order that responses are
     received.

BUGS  Broadcasting does not work through gateways.
```

```
sun10    up          11:56,    load average: 2.01, 2.01, 1.96
sun33    up 10 days,  2:51,    load average: 0.98, 1.00, 1.01
sun72    up           9:26,    load average: 0.21, 0.25, 0.30
sun13    up  1 day,  10:29,    load average: 0.02, 0.04, 0.04
sun14    up          15:24,    load average: 0.10, 0.05, 0.04
sun45    up  1 day,  11:07,    load average: 0.00, 0.02, 0.04
sun16    up 22 days,  9:36,    load average: 0.07, 0.02, 0.03
sun17    up          15:29,    load average: 0.02, 0.05, 0.05
sun18    up  2 days, 15:15,    load average: 0.01, 0.01, 0.01
sun19    up  2 days, 15:31,    load average: 0.84, 0.37, 0.21
sun20    up 10 days, 15:17,    load average: 0.00, 0.02, 0.05
sun27    up  9 days, 15:21,    load average: 1.00, 1.05, 1.07
sun18    up 14 days, 13:37,    load average: 0.09, 0.08, 0.07
sun31    up 65 days, 12:42,    load average: 0.04, 0.03, 0.05
sun34    up 23 days,  3:15,    load average: 0.02, 0.02, 0.02
sun56    up  2 days, 15:06,    load average: 0.00, 0.02, 0.04
sun57    up 22 days,  9:03,    load average: 0.02, 0.04, 0.04
sun58    up  3 days,  8:34,    load average: 0.00, 0.01, 0.03
sun59    up  3 days, 15:22,    load average: 0.05, 0.05, 0.04
sun60    up          15:23,    load average: 0.00, 0.02, 0.03
sun61    up 31 days,  7:10,    load average: 0.01, 0.03, 0.04
```

Serverwahl bei einem Lastverbund

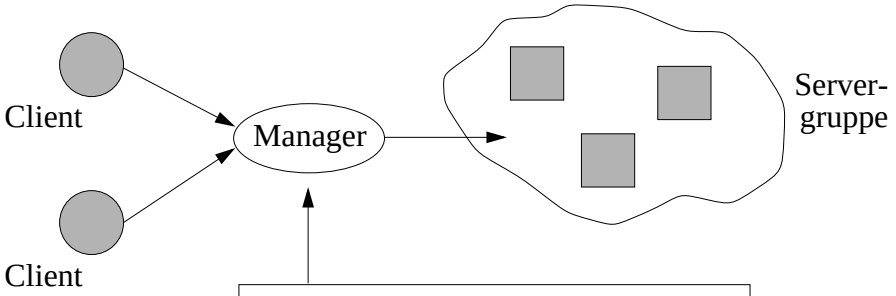


1) *Zufallsauswahl*

- 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*

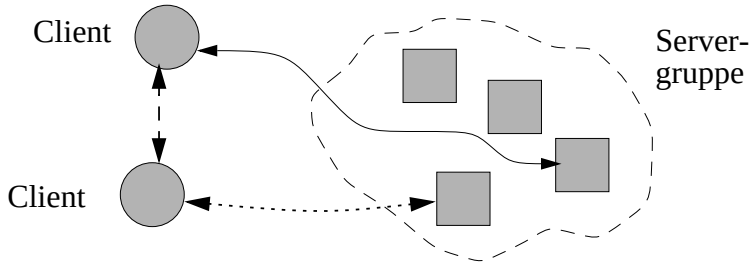


- sorgt für sinnvolle Verteilung (wie?)
- behält ggf. Überblick über Aufträge
- informiert sich ggf. von Zeit zu Zeit über die Server-Lastsituation

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

Serverwahl bei Lastverbund (2)

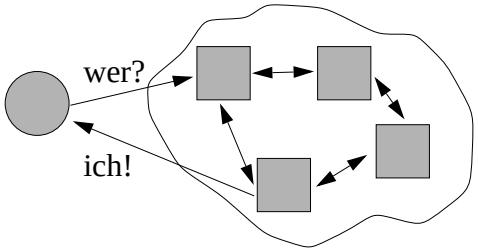
3) *Clients einigen sich untereinander*



- 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

- Election-Protokoll (aber fehlertolerant wegen möglichen Server-Ausfällen)
- ggf. Kooperations-topologie festlegen



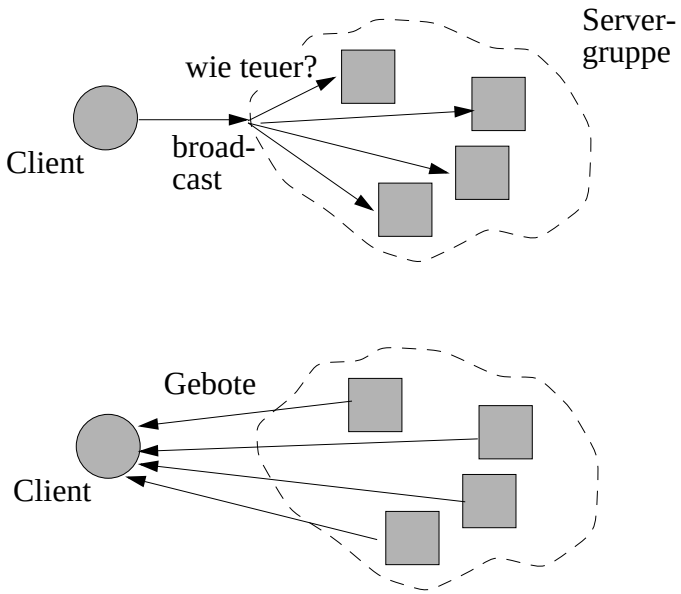
- i.a. nur wenige Server (relativ zur Zahl der Clients)
- Server führen Abstimmung diszipliniert durch (verlässlicher als Clients)

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



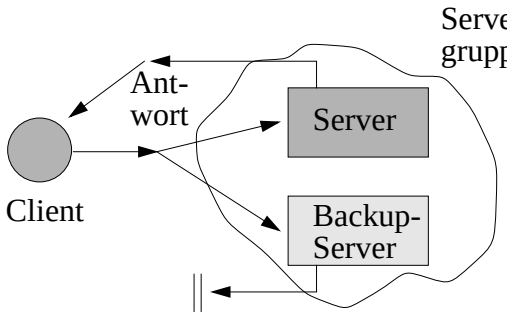
- Variante: nur *Stichprobe* befragen (multicast statt broadcast; sehr kleine Teilmenge von vielen Servern genügt i.a.!)

- 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:

- evtl. Subaufträge werden mehrfach erteilt (vom Server und Backup-Server) --> 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 Zuverlässigkeits-aspekte 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 und Lesen auf jeweils mehr als 1+N/2 Server zugreifen
  - Abgleich (voting) bzgl. neuester Versionsnummer