— VS —

Eigenschaften

Verteilte Systeme, ©Wolfgang Schröder-Preikschat

Überblick

Physikalische Verteiltheit

. . . der Hardware bezieht sich auf die Verbindung von Rechnern zu einem Netz, über direkte *Leitungen* beliebiger Art oder über *Transportsysteme*.

- Transportsysteme bestehen ihrerseits aus Rechnern und Leitungen
  – sie bilden also ein Rechnernetz innerhalb eines Rechnernetzes
- die Transportsystemrechner dienen der Datenweitergabe (*Vermittlung*)

. . . der Software spiegelt sich in den Prozessen wider, die auf Grundlage des Rechnernetzes zur Ausführung kommen und eröffnet wichtige Vorteile durch:

- dezentrale Informationsverarbeitung
- gemeinsame Nutzung von Betriebsmitteln
- Erhöhung der Zuverlässigkeit

Logische Verteiltheit

> So naheliegend es ist, die physikalische Verteiltheit als Kennzeichen eines verteilten Systems anzusehen, so unklar ist es, wann man ein System als physikalisch verteilt betrachtet und wann nicht. Es drängt sich unwillkürlich die Frage auf, ab welcher Entfernung von Komponenten die Bezeichnung als verteiltes System gerechtfertigt ist. [2]

- Technologiefortschritt bei der Hardware lässt Distanzen schrumpfen
  – gestern noch Rechnernetz, heute/morgen ein „*system on chip*“ (SOC)
  – die Aufteilung von Funktionen auf eigenständige Komponenten bleibt

- physikalische Verteilung ganz außer acht zu lassen, wäre jedoch zu voreilig[1]

[1] Beispielsweise erfordert Zuverlässigkeit physikalisch voneinander getrennte bzw. entfernte Komponenten.

Gemeinsame Nutzung von Betriebsmitteln

- der Zugriff auf Betriebsmittel kann (zusätzlich) aus der Ferne erfolgen
  - Betriebsmittel allen Prozessen als *Dienstleistung* zugänglich machen
    * {Druck, Datenbank, Datei, Web, . . . , Namens}dienst
  - Betriebsmittelverwalter sind dabei selbst Prozesse → ***Server***

- die gesamte Betriebsmittelmenge könnte gemeinschaftlich verwaltet werden

  So wie es in einem konventionellen System für die Durchführung eines Prozesses z.B. unerheblich ist, welchen Speicherplatz er zugewiesen bekommt, kann es nun vollkommen egal sein, auf welchem Rechner er ausgeführt wird.

- die Betriebsmittelvergabe erweitert sich um eine geographische Komponente

Heterogenität

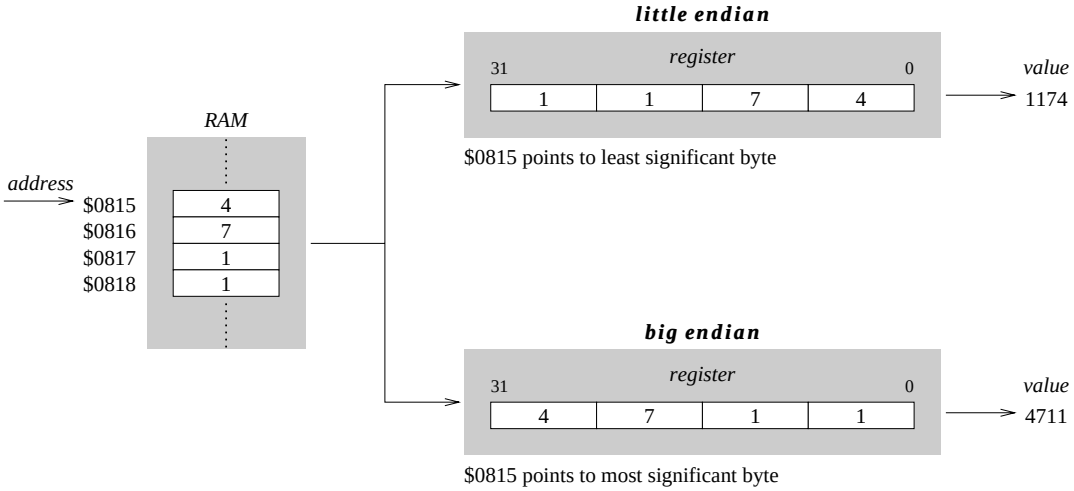
he·te·ro'gen <Adj.> *andersartig*, *ungleichartig*, *verschiedenartig*, *fremstoffig*; Ggs. *homogen* [2]

He·te·ro·ge·ni'tät <f.; -; unz.> ist in folgenden Bereichen vorzufinden:

- Netzwerke . . . . . Anschlusstyp, Medium, Technik, Topographie
- Prozessoren . . . . . Informationsdarstellung, „Byte Sex“
- Betriebssysteme . . . . . Ausführungsumgebung, API
- Programmiersprachen . . . . . Semantik, Pragmatik
- Implementierungen durch verchiedene Personen . . . . . Standards

[2] **ho·mo'gen** <Adj.> *gleichartig*, *gleichgeartet*, *gleichstoffig*, *gleichmäßig*, *zusammengesetzt*

Heterogenität „Byte Sex“



Heterogenität Ausführungsumgebung

```
wosch@faui42y 18> make 03vs.pdf
03vs.dvi -> 03vs.ps
This is dvips(k) 5.90a Copyright 2002 Radical Eye Software (www.radicaleye.com)
' TeX output 2003.03.28:2225' -> ps/03vs.ps
<texc.pro><texps.pro><special.pro><color.pro>. <cmssi10.pfb><cmss10.pfb>
<cmex10.pfb><cmsy10.pfb><cmr10.pfb><cmtt10.pfb><cmmi7.pfb><cmssbx10.pfb>
<cmmi10.pfb><cmbxsl10.pfb><cmsy7.pfb>[0] [1] [2] [3] [4] [5] [6
<fig/endian.eps>] [7] [8] [9] [10] [11<fig/server0.eps>] [12<fig/client0.eps>]
[13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25]
03vs.ps -> 03vs.pdf
wosch@faui42y 19>
```

Die Lösung des Problems:

```
alias make gmake
```

“You are not expected to understand this.”

$\Longleftarrow$

```
wosch@faui40u 20> make 03vs.pdf
This is TeX, Version 3.14159 (Web2C 7.3.1)
! I can't find file `.INIT'.
<*> .INIT

Please type another input file name:
```

Heterogenität $\Longrightarrow$ Middleware

- Softwareschicht zur Abstraktion von den jeweiligen Systemeigenheiten
  - Programmiersprachen $\begin{cases} \text{unabhängig} & \rightarrow & \text{CORBA} \\ \text{abhängig} & \rightarrow & \text{Java RMI} \end{cases}$
- einheitliches Programmiermodell zur Entwicklung verteilter Software
  - Prozedurfernaufruf (*remote procedure call*, RPC [3])
  - Objektfernaufruf (*remote object invocation*, ROI)
  - entfernte Ereignisbenachrichtigung oder SQL-Zugriffe
  - verteilte Transaktionsverarbeitung
- grundlegende Bausteine bilden **Prozesse** und **Botschaftenaustausch**

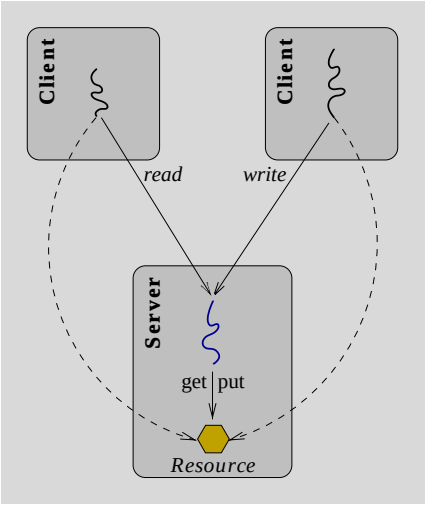
Heterogenität $\Longrightarrow$ Virtuelle Maschine

- der Begriff „*mobiler Kode*“ bezieht sich auf übertragbaren Maschinenkode
  - beispielsweise Java-Applets, aber auch (viele) PostScript-Programme :-)
  - nicht jedoch als Anhänge in e-Post an UNIX versandte `.exe`-Dateien :-(
- Anweisungsfolgen übertragbaren Maschinenkodes sind Hardware unabhängig
  - ein Übersetzer erzeugt Zwischenkode für eine virtuelle Maschine (VM)
  - die VM ist für jeden Typ Hardware nur einmal implementiert → JVM
  - ein Interpreter (d.h. die VM) führt den Zwischenkode aus, nicht die CPU
  - ggf. erfolgt auch eine „*just in time*“ Übersetzung einzelner Komponenten
- der Ansatz ist i.A. abhängig von der Programmiersprache Java, ¬ C++

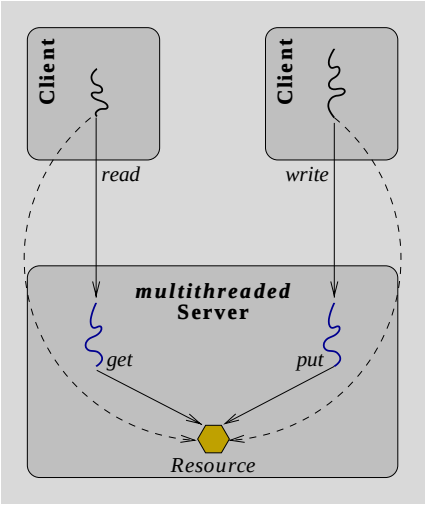
Nebenläufigkeit

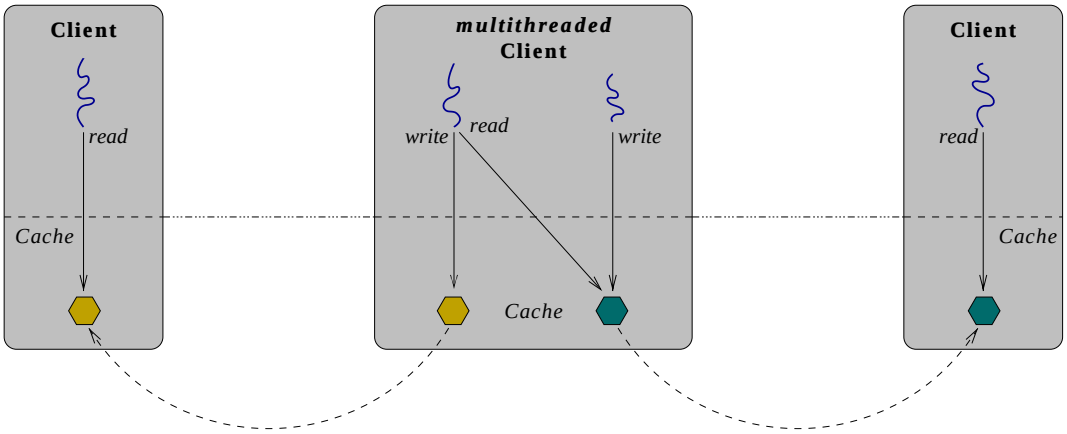
- gemeinsame Nutzung von Betriebsmitteln ist grundlegendes Charakteristikum
  - „gleichzeitige“, sich überlappende Betriebsmittelzugriffe sind höchst typisch
- die nebenläufigen Zugriffe finden auf verschiedenen Ebenen statt:
  1. mehrere Prozesse ($\rightarrow$ ***Clients***) benutzen einen Server zum selben Zeitpunkt,
  2. der Server ist mehrfädig ausgelegt, d.h. bedient mehrere Klienten gleichzeitig
  3. und/oder die Betriebsmittel liegen klientenseitig als Replikate vor (*Caching*)
- *Kooperation* der Zugriffe geht (weit) über klassische Semaphorverfahren hinaus
  - Konsistenzwahrung erfordert den Einsatz verteilt arbeitender Algorithmen

Nebenläufigkeit



Server-Varianten





Fehlerverarbeitung

- die Wahrscheinlichkeit von Fehlern (in technischen Systemen) ist niemals 0
  - fehlerbedingte Ausfälle in verteilten Systemen sind partiell
    - d.h., einige Komponenten fallen aus, während andere noch funktionieren
  - zur Verarbeitung von Fehlern kommen verschiedene Techniken zum Einsatz:
    - Fehler erkennen, maskieren, tolerieren
    - Wiederherstellung nach Fehlern
    - Redundanz
  - verteilte Systeme bieten einen (relativ) hohen Grad an *Verfügbarkeit*[3]
- „*5-nines*“ (99.999 %) Fehlertoleranz ist eine der großen Herausforderungen

[3] Ein Maß für den Zeitanteil, während dessen das System zur Verfügung steht.

Fehlerverarbeitung — Fehler . . .

∼ **erkennen** Einige Fehler sind erkennbar (z.B. durch Prüfsummen), andere sind unmöglich zu erkennen (z.B. einen Server-Ausfall). Die Herausforderung ist, mit nicht erkenn- aber vermutbaren Fehlern umzugehen.

∼ **maskieren** Einige Fehler, die erkannt wurden, können verborgen werden (z.B. verlorene Nachrichten wiederholen, Dateien auf mehrere Datenträger sichern) oder abgeschwächt werden (z.B. fehlerhafte Nachrichten verwerfen). Probleme bereitet der nicht ganz ausschließbare „schlimmste Fall“[4].

∼ **tolerieren** Einige Fehler, die nicht erkannt oder maskiert werden konnten, sind hinzunehmen und ggf. bis hinauf zur Anwendungsebene „hochzureichen“. Software verteilter Systeme soll „fehlergewahr“ sein. Redundanz hilft dabei.

[4] Beispielsweise ist der Wiederholungszähler abgelaufen oder die weiteren Datenträger sind ebenfalls defekt.

Fehlerverarbeitung — Wiederherstellung

- Rechnerabstürze bzw. Komponentenausfälle zeigen i.A. typische Fehlermuster:
  - Berechnungen von Programmen sind unvollständig
  - permanente Daten befinden sich möglichweise im inkonsistenten Zustand
- grundsätzlich wird dabei zwischen zwei Fehlerarten unterschieden:

  transiente Fehler werden durch ∼smaßnahmen behoben, die zum Ziel haben, einen konsistenten Systemzustand (wieder) zu erreichen
  - *checkpointing*, (*forward*/*backward*) *recovery*, Transaktionen

  permanente Fehler werden durch Reparatur behoben, indem die fehlerhafte Komponente ersetzt oder umgangen wird
- Maßnahmen zur Wiederherstellung sind im Softwareentwurf zu berücksichtigen

Fehlerverarbeitung — Redundanz

- Fehlertoleranz bedeutet „über das Notwendige hinausgehen“ zu müssen:
  - mehr als eine { Route zwischen zwei Punkten im Rechnernetz vorsehen / Serverinstanz (Prozess, Rechner) derselben Art einsetzen
- kritische „Funktionsgruppen“ liegen dazu oft *funktional repliziert* vor
  - redundante Implementierung ein und derselben Einheit
  - realisiert durch Einsatz unabhängiger Entwicklungsteams
- beträchtlicher Mehraufwand steht den erhofft seltenen Ausfällen gegenüber
  - Fernaufrufe an Servergruppen absetzen — die „richtige“ Antwort auswählen
  - Daten mehrfach speichern — Aktualisierungen überall nachvollziehen

Sicherheit

- den Eigenwert von Informationen für ihre Benutzer zu sichern bedeutet:

  Schutz vor {
  - Offenlegung gegenüber Unbefugten → **Vertraulichkeit**
  - Veränderung oder Beschädigung → **Integrität**
  - Störungen des Betriebsmittelzugriffs → **Verfügbarkeit**

- sensible Informationen sind sicher über ein Netzwerk zu übertragen
  - dabei reicht es nicht, nur den Inhalt der Nachrichten zu verbergen
  - die Identität des Absenders (Benutzer, Agent) ist sicherzustellen
- Verschlüsselungsverfahren helfen, die Authentizität von Klienten zu bestimmen

Offenheit

offene Systeme zeichen sich durch die Veröffentlichung (der Spezifikation und Dokumentation) der Schnittstellen ihrer „Schlüsselkomponenten“ aus.

offene verteilte Systeme basieren auf dem Vorhandensein eines einheitlichen Kommunikationsmechanismus und veröffentlichten Schnittstellen für den Zugriff auf gemeinsam genutze Betriebsmittel;

$\sim$ können aus heterogener Hard- und Software aufgebaut sein, die insbesondere auch von unterschiedlichen Herstellern stammen können.

Die Herausforderung ist, mit der Komplexität von Systemen zurechtzukommen, die aus vielen Komponenten (unterschiedlicher Herkunft) bestehen.

Skalierbarkeit

> Ein System, das als *skalierbar* bezeichnet wird, bleibt auch dann effektiv, wenn die Anzahl der Ressourcen und die Anzahl der Benutzer wesentlich steigt. [1]

- daraus leiten sich folgende Problemfelder für Entwurf und Implementierung ab:
  - Kontrolle
    - der Kosten für die physischen Betriebsmittel
    - des Leistungsverlusts
  - Vermeidung
    - von Betriebsmittelerschöpfung (32/128-Bit Internetadressen)
    - von Leistungsengpässen (Replikation, *Caching*)
- im Idealfall sollte der Zuwachs „transparent“ sein für System und Anwendungen

Skalierbarkeit — Komplexität

Kostenkontrolle Soll ein System mit n Benutzern skalierbar sein, so sollte die Anzahl der physischen Ressourcen für ihre Unterstützung mindestens proportional zu n sein, d.h. $O(n)$.

Verlustkontrolle Suchalgorithmen (um z.B. Einträge im *Domain Name System*, DNS, zu finden und aufzulösen), die hierarchische Strukturen verwenden, skalieren besser als solche mit linearen Strukturen. Gleichwohl resultiert der Größenanstieg in einen gewissen Leistungsverlust:

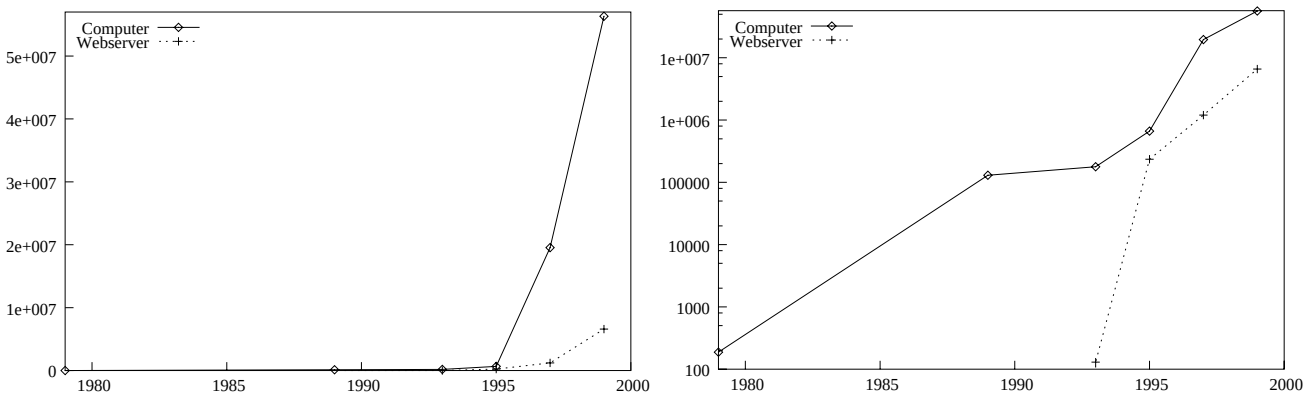
- die Zeit, die für den Zugriff auf eine hierarchische Struktur benötigt wird, ist $O(log\ n)$, wobei n die Größe der Datenmenge darstellt.

Damit ein System skalierbar ist, sollte der maximale Leistungsverlust nicht höher sein.

Skalierbarkeit — Internet-Rechner

Jahr	Computer	Webserver	Anteil
1979	188	0	0 %
1989	130 000	0	0 %
1993	1 776 000	130	0.008 %
1995	6 642 000	235 000	0.4 %
1997	19 540 000	1 203 096	6 %
1999	56 318 000	6 598 697	12 %

(Quelle: [1])



Transparenz (1)

Zugriffs∼ ermöglicht den Zugriff auf lokale und globale (d.h. entfernte) Betriebsmittel unter Verwendung identischer Operationen. Das betreffende API macht keinen Unterschied zwischen lokalen und entfernten Operationen.

Orts∼ (auch Positions∼) erlaubt den Zugriff auf Betriebsmittel, ohne ihre Position/ihren Ort kennen zu müssen. Beispielsweise erfolgt der Zugriff nicht direkt über Internetadressen, sondern indirekt über Domänennamen (DNS).

Beide werden unter dem Begriff **Netzwerktransparenz** zusammengefasst. So ist z.B. wosch@informatik.uni-erlangen.de Netzwerk transparent. [warum?]

Transparenz (2)

Nebenläufigkeits∼ erlaubt mehreren Prozessen gleichzeitiges und konfliktfreies Arbeiten mit denselben gemeinsam genutzten Betriebsmitteln.

Replikations∼ erlaubt die Verwendung mehrerer Betriebsmittelinstanzen (d.h. *Repliken*), um Zuverlässigkeit und Leistung zu verbessern.

Fehler∼ erlaubt den kontinuierlichen Rechnereinsatz trotz des möglichen Ausfalls von Hard- und Software-Komponenten (*non-stop computing*).

Transparenz (3)

Migrations$\sim$ (auch Mobilitäts$\sim$) erlaubt das Verschieben bzw. Wandern von Betriebsmitteln und Prozessen innerhalb eines Systems ohne Beeinträchtigung der laufenden Arbeit von Benutzern bzw. Programmen.

Leistungs$\sim$ erlaubt die Last abhängige Neukonfigurierung des Systems zum Zwecke der Leistungssteigerung.

Skalierungs$\sim$ erlaubt die Vergrößerung (ggf. auch Verkleinerung) des Systems ohne Auswirkungen auf die realisierte Struktur und die zum Einsatz gebrachten Algorithmen.

Unschärfeprinzip

Es ist die Eigenart verteilter Systeme, daß es auf Dauer keine zwei Prozesse gibt, die zur gleichen Zeit die gleiche, zutreffende Sicht des Systems haben. Ein Prozeß innerhalb des Systems verfügt entweder über **unvollständige**, **aktuelle** oder über **vollständige**, **überholte** Zustandsinformationen. [2]

Zusammenfassung

Everything should be made as simple as possible, but no simpler.
(Albert Einstein)

You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.
(Antoine de Saint Exupery)

Referenzen

[1] G. Coulouris, J. Dollimore, and T. Kimberg. *Verteilte Systeme: Konzepte und Design*. Pearson Education, 2002. ISBN 3-8273-7022-1.

[2] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*. Springer-Verlag, 1989. ISBN 3-540-51701-4.

[3] B. J. Nelson. Remote Procedure Call. Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[4] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability Issues in Conputing System Design. *ACM Computing Surveys*, 10(2):123–165, June 1978.

[5] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters Ltd., 3rd edition, 1998. ISBN 15-688-1092-X.