Überblick

- charakteristische Eigenschaften:
  - Heterogenität
  - Nebenläufigkeit
  - Fehlerverarbeitung
- wünschenswerte Eigenschaften:
  - Sicherheit
  - Offenheit
  - Skalierbarkeit
  - Transparenz

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
- 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 ein „*system on chip*“
  - die Aufteilung auf eigenständige Komponenten bleibt
- physikalische Verteilung ganz außer acht zu lassen, wäre jedoch zu voreilig

  (Beispielsweise erfordert Zuverlässigkeit physikalisch voneinander getrennte bzw. entfernte Komponenten.)

Gemeinsame Nutzung von Betriebsmitteln

- der Zugriff auf Betriebsmittel kann aus der Ferne erfolgen
  - Betriebsmittel allen Prozessen als *Dienstleistung* zugänglich
    - {Druck, Datenbank, Datei, Web, . . . , Namens}dienst
  - Betriebsmittelverwalter sind dabei selbst Prozesse → **Server**
- die gesamte Betriebsmittelmenge könnte gemeinschaftlich verwaltet werden

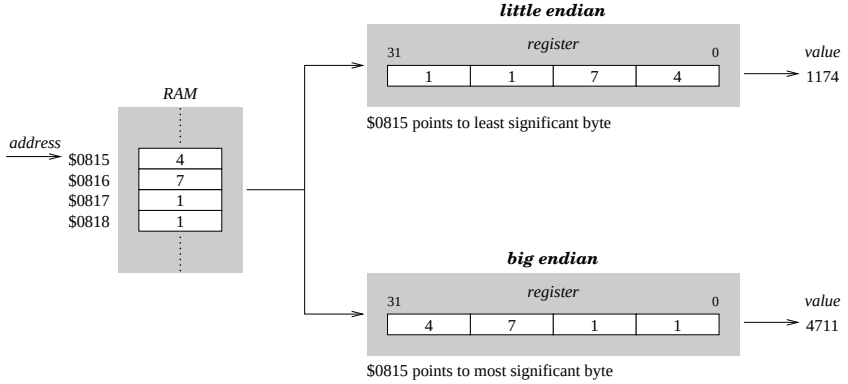
  So wie es in einem konventionellen System für die Durchführung eines Prozesses z.B. unerheblich ist, welchen Speicherplatz er zugewiesen bekommt, kann es nun vollkommen egal sein, auf welchem Rechner er ausgeführt wird.
- die Betriebsmittelvergabe erweitert sich um eine geographische Komponente

Heterogenität

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

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 verschiedene Personen . . . . . Standards



Heterogenität „Byte Sex“

Heterogenität Middleware

- Softwareschicht zur Abstraktion von den jeweiligen Systemeigenheiten
  - Programmiersprachen { unabhängig → CORBA / Web Service; abhängig → Java RMI
- einheitliches Programmiermodell zur Entwicklung verteilter Software
  - Prozedurfernaufruf (*remote procedure call*, RPC [3])
  - Objektfernaufruf (*remote method invocation*, RMI)
  - entfernte Ereignisbenachrichtigung oder SQL-Zugriffe
  - verteilte Transaktionsverarbeitung
- grundlegende Bausteine bilden **Prozesse** und **Botschaftenaustausch**

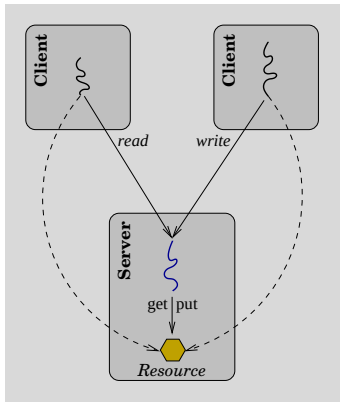
Heterogenität Virtuelle Maschine

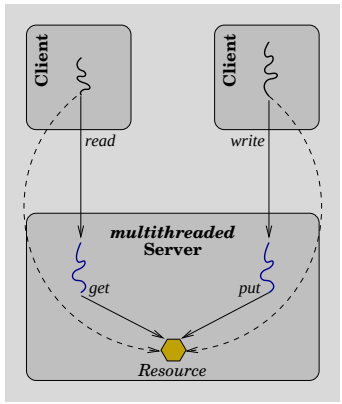
- der Begriff „*mobiler Code*“ bezieht sich auf übertragbaren Maschinencode
  - beispielsweise Java Bytekode
  - oder auch in Form von Schadsoftware (z.B. per Email)
- Anweisungsfolgen übertragbaren Maschinencodes sind Hardware unabhängig
  - ein Übersetzer erzeugt Zwischencode für eine virtuelle Maschine (VM)
  - die VM ist für jeden Typ Hardware nur einmal implementiert
  - 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 Beispiel: Java, C#, ¬ C++

Nebenläufigkeit

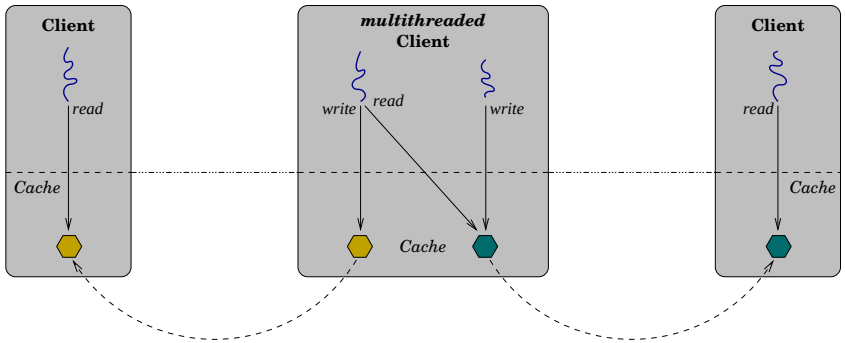
- gemeinsame Nutzung von Betriebsmitteln ist grundlegendes Charakteristikum
  - „gleichzeitige“, sich überlappende Betriebsmittelzugriffe sind höchst typisch
- die nebenläufigen Zugriffe finden auf verschiedenen Ebenen statt:
  1. mehrere Prozesse ($\rightarrow$ **Clients**) benutzen einen Server zum 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





Nebenläufigkeit — Replikation



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*[1]
- „*5-nines*“ (99.999 %) Fehlertoleranz ist eine der großen Herausforderungen

Fehlerverarbeitung — Was ist ein Fehler?

- **Fault (Fehlerusache)** – unerwünschter Zustand, der zu einem Fehler führen kann
- **Error (Fehler)** – Systemzustand, der nicht den Spezifikationen entspricht
- **Failure (Funktionsausfall)** – Diensterbrinung ist nicht mehr möglich

Singal/Shivaratri

An error is a manifestation of a fault in a system, which could lead to system failure.

Fehlerverarbeitung — Klassifikation

- Gutmütige Fehler (benign faults)
  - **Crash Stop:** Ein Knoten fällt komplett aus
    - **Fail Stop**: Jeder korrekte Knoten erfährt innerhalb endlicher Zeit vom Ausfall eines Knoten
    - **Fail Silent**: Keine perfekte Ausfallerkennung möglich (asynchrones oder partiell synchrones Modell)
  - **Crash Recovery**: Ein korrekter Knoten kann endlich oft ausfallen und wiederanlaufen
- Bösartige Fehler (malicious faults)
  - Häufig als **byzantinische** Fehler bezeichnet
  - Fehlerhafte Prozesse können beliebige Aktionen ausführen, und dabei auch untereinander kooperieren
  - Modell, das alle beliebigen Arten von Fehler umfasst, z.B. auch gezielte Angriffe von außen auf das System

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 erkennaber 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“[2].

~ 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.

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 Netz 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 genutzte Betriebsmittel;
  - 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.

Transparenz (1)

- Zugriffstransparenz
  - 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.
- Ortstransparenz (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∼
  - (auch Mobilitäts∼) erlaubt das Verschieben bzw. Wandern von Betriebsmitteln und Prozessen innerhalb eines Systems ohne Beeinträchtigung der laufenden Arbeit von Benutzern bzw. Programmen.
- Leistungs∼
  - erlaubt die Last abhängige Neukonfigurierung des Systems zum Zwecke der Leistungssteigerung.
- Skalierungs∼
  - 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]*

Verteilte Systeme: Definition (die 2.) (V. Garg)

Aus Sicht der Fehlertoleranz: [6]

- Keine gemeinsame Uhr
  - Uhrensynchronisation schwierig aufgrund nicht bekannter Verzögerungszeiten bei der Kommunikation, physikalische Uhren werden daher nur selten zur Koordinierung verwendet
- Kein gemeinsamer Speicher
  - Kein einzelner Knoten kennt den vollständigen globalen Zustand. Es ist daher schwierig, globale Eigenschaften des Systems zu beobachten
- Keine akkurate Ausfallerkennung
  - Es ist in einem asynchronen verteilten System (d.h. keine obere Schranke für Kommunikationszeiten bekannt) unmöglich, langsame von ausgefallenen Prozessen zu unterscheiden

Ausklang. . .

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

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

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

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

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

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.

P. Vijay K. Garg.
Elements of distributed computing.
John Wiley & Sons, Inc., New York, NY, USA, 2002.