Verteilte Systeme

...für C++ Programmierer

by

Dr. Günter Kolousek

© Dr. Günter Kolousek

Creative Commons Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International Lizenz

Verteilte Systeme – Definition

> *Ein verteiltes System ist eine Menge von einander unabhängiger Computer, die dem Benutzer wie ein einzelnes System erscheinen.* Andrew S. Tanenbaum

- Anwendung steht im Vordergrund
  - über mehrere Rechner verteilt, aber von außen nicht erkennbar
- Software nötig, die diese Abstraktionsebene zur Verfügung stellt
  - z.B. $\rightarrow$ MOM, ESB,…

Verteilte Systeme – Definition

Ein verteiltes System ist eine Menge von einander unabhängiger Computer, die dem Benutzer wie ein einzelnes System erscheinen. Andrew S. Tanenbaum

- Anwendung steht im Vordergrund
  - über mehrere Rechner verteilt, aber von außen nicht erkennbar
- Software nötig, die diese Abstraktionsebene zur Verfügung stellt
  - z.B. $\rightarrow$ MOM, ESB,…

A distributed system is one in which I cannot get something done because a machine I've never heard of is down.

Leslie Lamport

Verteilte Systeme – Gedankenfehler

The eight fallacies of distributed computing

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

1. *The network is reliable*
2. *Latency is zero*
3. *Bandwidth is infinite*
4. *The network is secure*
5. *Topology doesn't change*
6. *There is one administrator*
7. *Transport cost is zero*
8. *The network is homogeneous*

Peter Deutsch

Verteilte Systeme – Ziele

- Benutzer und Ressourcen verbinden
- Transparenz
- Offenheit
- Skalierbarkeit

Benutzer und Ressourcen verbinden

- Ziele
  - Zugriff auf entfernte Ressourcen
  - mit anderen Benutzern kontrolliert benutzen
- Beispiele
  - Drucker, Speichergeräte
  - Prozessorleistung, Netzwerk
  - Dateien, Webseiten
  - Dienste
- Gründe
  - Kosten der HW, SW (Lizenzen)
  - Informationsaustausch (z.B. Dateien, E-Mails)
  - Vereinfachung der Zusammenarbeit (z.B. Workflow-Lösungen)
  - Sicherheit (Zugriffskontrolle, Protokollierung)

Transparenz

- Ziel
  - Tatsache verbergen, dass Prozesse und Ressourcen physisch über mehrere Computer verteilt sind
- Verteiltes System, das sich Benutzern und Applikationen so präsentiert als wäre es ein System, wird als transparent bezeichnet.
- nicht immer sinnvoll, hohen Transparenzgrad erreichen zu wollen. Gründe:
  - Systemleistung: jede Art von Transparenz benötigt Ressourcen.
  - Physische Gegebenheiten können teilweise nicht verborgen werden
    - Prozess in Wr. Neustadt soll Verbindung zu Prozess in New York aufnehmen!

Transparenz – Begriff

- Deutsch
  - transparent … durchsichtig
- englische Begriffe
  - *opaque* … undurchsichtig
  - *transparent* … durchsichtig
- in der Informatik
  - *transparent* … nicht sichtbar, nicht merklich

Transparenz – Arten

- Zugriffstransparenz
  - Unterschiede in der Datendarstellung und wie der Zugriff des Benutzers erfolgt sind transparent.
  - Bsp.: Wie werden Zahlen repräsentiert? Welcher Zeichensatz wird verwendet? Mit welchem Protokoll erfolgt der Zugriff?
- Positionstransparenz
  - Der physische Ort einer Ressource bleibt vor dem Benutzer verborgen.
  - Bsp.: Wo befindet sich die Ressource physisch im System? Wie werden die Namen der Ressourcen vergeben?
- Migrationstransparenz
  - Ressourcen können verschoben werden.
  - Bsp.: Eine Datei wird von einem Server auf einen anderen Server verschoben.

Transparenz – Arten – 2

- Relokationstransparenz
  - Ressource kann verschoben werden während der Zugriff erfolgt ($\rightarrow$ Positionstransparenz).
  - Bsp.: Ein mobiler Benutzer mit Funk-Laptop bewegt sich bei bestehender Datenverbindung von einer GSM Zelle in eine andere.
- Replikationstransparenz
  - Es existieren mehrere Kopien einer Ressource. Benutzer muss nicht wissen auf welche Replik er zugreift ($\rightarrow$ Positionstransparenz).
  - Bsp.: Der Zugriff erfolgt auf eine lokal verfügbare Datenbank, die mittels der Mechanismen des verwendeten DBMS repliziert wird.

Transparenz – Arten – 3

- Persistenztransparenz
  - Ressource befindet sich entweder im flüchtigen Speicher oder z.B. auf einer Festplatte.
  - Bsp.: Bei der Verwendung von objektorientierten Datenbanken wird auf ein Objekt in der Datenbank genauso zugegriffen wie auf ein Objekt im Hauptspeicher.
- Nebenläufigkeitstransparenz
  - Benutzer erkennt nicht, dass ein anderer Benutzer dieselbe Ressource simultan benutzt.
  - Bsp.: Benutzer greift auf Daten in einer Tabelle zu, ohne Rücksicht darauf nehmen zu müssen, dass eventuell andere Benutzer ebenfalls auf diese Daten zugreifen.

Transparenz – Arten – 4

- Fehlertransparenz
  - Ein Fehler in einer Ressource ist für den Benutzer nicht sichtbar, d.h. wird vom System aufgelöst.
  - Bsp.: Welche Festplatte einen Fehler in einer RAID-5 Konfiguration aufweist bleibt dem Benutzer verborgen und außerdem wird der Fehler korrigiert. Dazu muss natürlich folgendes passieren:
    1. Fehler erkennen
    2. Fehler maskieren oder Fehler tolerieren
    3. Wiederherstellen nach Fehlern

Offenheit

- Ziele
  - Interoperabilität
    - Ausmaß in dem zwei Implementierungen von Systemen nebeneinander existieren und zusammenarbeiten können, indem sie sich auf die Dienste des anderen verlassen, die gemäß einer Spezifikation implementiert sind (siehe Folien ”Daten und Interoperabilität”)
  - Portabilität
    - Ausmaß, das angibt inwieweit eine Anwendung, die für ein System A entwickelt wurde, ohne Veränderung auf einem System B ausgeführt werden kann, das dieselbe Schnittstelle wie A implementiert.

Offenheit – 2

- Ein *offenes* verteiltes System ist ein System, das Dienste Standardregeln entsprechend anbietet (diese beschreiben die Syntax und die Semantik dieser Dienste).
  - Standardregeln: Format, Inhalt und die Bedeutung gesendeter und empfangener Nachrichten. Solche Regeln werden in Protokollen (siehe Folien ”Kommunikation”) formalisiert.
    - Grund warum Komponenten austauschbar sind

Skalierbarkeit

- Ziel
  - Durch Hinzufügen von Ressourcen die Leistung des Systems zu steigern.
- Arten
  - Skalierbarkeit bezüglich der Größe des Systems: Hinzufügen weiterer Benutzer und Ressourcen.
  - Geographische Skalierbarkeit: Benutzer und Ressourcen können sehr weit aus- einanderliegen.
  - Administrative Skalierbarkeit: Verwaltbarkeit, wenn sich das System über viele unabhängige administrative Organisationen erstreckt.

Skalier… – Vertikal vs. horizontal

- Vertikale Skalierung
  - Hinzufügen von Ressourcen zu einzelnem Rechner
    - z.B. RAM, schnellere CPU,…
  - SW muss *nicht* geändert werden
  - man stößt rasch an Grenze
- Horizontale Skalierung
  - Hinzufügen weiterer Rechner
  - SW muss meist geändert werden
    - z.B. parallele Algorithmen, Verteilung,…
  - praktische Grenze durch Amdahlsches Gesetz

Skalierungsfaktor

- Skalierungsfaktor (engl. speedup): beschreibt tatsächlichen Leistungszuwachs durch Hinzufügen einer zusätzlichen Ressource
- Lineare Skalierbarkeit: Skalierungsfaktor bleibt je hinzugefügter Ressource gleich $\rightarrow$ wünschenswert
- Sublineare Skalierbarkeit: Skalierungsfaktor nimmt mit hinzugefügter Ressource ab

Skalierbarkeit – Grenzen

- Verwendung eines zentralen Dienstes
  - ein Server stellt alle Dienste zur Verfügung
- zentrale Speicherung der Daten
  - alle Daten werden an einem Ort gespeichert
- Verwendung von sequenziellen Algorithmen (im weitesten Sinne)
  - anstatt von parallelen Algorithmen
- Sicherheitsmaßnahmen
  - zentralisierte Struktur notwendig

Skalierungstechniken

- geographische und organisatorische Verteilung
  - Beispiel: der DNS-Namensraum ist hierarchisch als Domänenbaum aufgebaut und die DNS-Server sind ebenfalls hierarchisch vernetzt
- Load balancing (Lastverteilung)
  - Leistung
  - Verteilung der Daten auf mehrere Hosts (1 Master, mehrere Slaves)
- Replikation
  - Beispiel: Zugriff auf lokale, replizierte Daten
  - $\rightarrow$ Folie ”Replikation”
- Caching: Spezialform der Replikation
  - Änderungen immer an einer Master-Datenquelle!
  - Beispiel: Cachen von DNS-Anfragen
- Partitionierung der Daten auf mehrere Hosts

Verteilte Systeme – weitere Themen

- Adressen und Namen
- Replikation
- Verlässliche Systeme und Fehlertoleranz

Adresse

- Zugangspunkt: notwendig, um auf Ressource zugreifen zu können
  - eine Ressource kann über mehrere Zugangspunkte verfügen
  - Ressource kann über Zeit ihre Zugangspunkte ändern
- Adresse: *Name* eines Zugangspunktes
  - muss vom Menschen nicht interpretierbar sein
- Beispiel
  - Telefon als Zugangspunkt zu Person
    - Telefonnummer entspricht Adresse
    - Name der *Person* $\rightarrow$ Telefonbuch $\rightarrow$ Telefonnummern (Adressen der Zugangspunkte)

Name

- Name: Zeichenkette, um auf Ressource zu verweisen
  - vom Menschen interpretierbar
- sinnvoll
  - da Zugangspunkte sich ändern können
    - Beispiel: Telefonnummern ändern sich…
  - vom Menschen interpretierbar
- ID
  - identifizieren Ressource eindeutig
    - Spezialfall eines Namens
  - Eigenschaften
    - eine ID verweist auf höchstens eine Ressource
    - Jede Ressource wird durch höchstens eine ID angesprochen
    - ID verweist immer auf selbe Ressource (d.h. wird nicht weiterverwendet)

Nachschlagsdienste

- meist: Adressen und ID nur maschinenlesbar
- Namensdienste (name service)
  - Namen in Namensräumen organisiert
    - flach oder hierarchisch
  - Namensdomäne: Namensraum für den es eine einzige administrative Autorität gibt
    - z.B. DNS
  - Namen $\rightarrow$ Adressen
    - solche Dienste: *white pages*

Nachschlagsdienste – 2

- Verzeichnisdienste (directory service)
  - Zuordnung von Attributen zu Ressourcen (oder allg. Objekten)
  - Auswahl über Attributspezifikationen
  - Beispiele: LDAP, Active Directory
  - solche Dienste: *yellow pages*
- Erkennungsdienste (service discovery)
  - spontane Netzwerke: Ressourcen zum Netzwerk werden hinzugefügt und wieder entfernt
  - automatische Registrierung bzw. Deregistrierung von Ressourcen (wie z.B. Diensten)
  - Beispiele: Bonjour (Apple), Jini ($\rightarrow$ Java Objekte), UDDI (universal description, discovery, and integration), DHCP, UPnP (universal plug and play)

Replikation

- mehrfache Speicherung derselben Daten an verschiedenen Standorten
  - $\rightarrow$ Synchronisation!
- Vorteile
  - Erhöhung der Zuverlässigkeit
  - Verbesserung der Leistung (zumindest bei Leseoperationen)
  - $\rightarrow$ Lastverteilung
- Nachteile
  - Replikationsprotokolle $\rightarrow$ komplex $\rightarrow$ erhöhter Aufwand
  - erhöhter Speicherbedarf
  - Probleme der Konsistenz: Replikation $\rightarrow$ Zeit für Abgleich $\rightarrow$ Unterschiede in Datenbeständen

Verlässliche Systeme

- Ziel: verlässliche Systeme entwickeln
- Anforderungen
  - Verfügbarkeit: Wahrscheinlichkeit, dass ein System zu einem gewissen (d.h. jeden beliebigen) Zeitpunkt korrekt funktioniert:

    $$V = \frac{t_{gesamt} - t_{gesamtausfallzeit}}{t_{gesamt}}$$

  - Zuverlässlichkeit: Eigenschaft, dass ein System fehlerfrei über ein gewisses Zeitintervall funktioniert.
    Ein Maß für die Zuverlässlichkeit ist die MTBF (Mean Time Between Failures):

    $$\text{MTBF} = \frac{\sum(downtime - uptime)}{n}$$

Verlässliche Systeme – 2

Beispiel:

- Wenn System jede Stunde für eine Millisekunde ausfällt hat es eine Verfügbarkeit von 0.999999, aber es ist jedoch höchst unzuverlässig!
- System stürzt niemals ab, aber in jedem August zwei Wochen heruntergefahren wird, und damit zwar zuverlässig ist, aber nur 96 Prozent Verfügbarkeit aufweist.

  Achtung: verschiedene Definitionen von Verfügbarkeit, z.B. ob geplante Abschaltzeiten hinzugezählt werden oder nicht.

Verlässliche Systeme – 3

- weitere Anforderungen
  - Fehlertoleranz: Eigenschaft, dass das System seine Dienste trotz Vorliegen eines Fehlers einer Komponente bereitstellen kann (Fehlermaskierung).
  - Sicherheit (im Kontext der Fehlertoleranz): Es darf nichts Katastrophales passieren, wenn ein System vorübergehend nicht korrekt funktioniert.
  - Wartbarkeit: wie einfach oder schwierig ist es, ein ausgefallenes System zu reparieren
    - oder generell Änderungen durchzuführen

Verlässliche Systeme – 4

- Fehlerarten
  - vorübergehender Fehler: tritt nur einmal auf und verschwindet wieder
  - periodischer Fehler: tritt auf, verschwindet wieder, tritt auf…
  - permanenter Fehler: existiert bis der Fehler behoben ist
- Techniken, um verlässliche Systeme zu erhalten:
  - Redundanz als Mittel zur Fehlermaskierung
  - Gestaltung und Implementierung von geeigneten Benutzerschnittstellen
  - Diversität im Entwicklungsprozess

Redundanz

- physische Redundanz
  - mehrfache Ausführung der Komponenten (HW) oder der Prozesse (SW)
  - Beispiele
    - Notstromversorgung
    - mehrfache Ausführung der Netzgeräte, Lüfter, Netzwerksschnittstellen
    - gespiegelte Festplatten
    - 2 Server, wobei einer im Standby (hot standby vs. cold standby) läuft
    - 2 Server, um Fehler zu erkennen
    - n Servern, wobei eine Einigung erzielt werden muss
  - Wichtig: Fehlererkennung (Benutzer oder automatisch), damit die redundant ausgelegte Systemkomponente den Betrieb übernimmt.
    I.d.R. zusätzliche **zuverlässige** HW, die die Fehler erkennt und die Umschaltung durchführt.

Redundanz – 2

- Informationsredundanz
  - es wird zusätzliche Information hinzugefügt, um die Wiederherstellung der Daten zu ermöglichen.
  - Beispiele: Fehlerkorrekturcodes (z.B. CRC-Prüfsummen), Fehlertoleranz bei Eingaben (z.B. ISBN-13),3 Festplatten als RAID-5
- zeitliche Redundanz: eventuelle Wiederholung einer Aktion.
  - mehrmaliges Durchführen von idempotenten Aktionen $\rightarrow$ bei kurzzeitigen HW Fehlern
  - mehrmaliges Durchführen von Berechnungen
    - mit gleichen Daten $\rightarrow$ bei kurzzeitigen HW Fehlern
    - mit leicht veränderten Daten $\rightarrow$ Rundungsfehler
  - Verwendung von Transaktionen: Abbruch, dann problemlos eine Wiederholung möglich

Fehlerkorrektur

- Vorwärtsfehlerkorrektur: weiter als ob kein Fehler
  - bei falschen Eingabewerten
    - Verwendung von alten Erfahrungswerten
    - Verwendung von Werten aus funktionierenden Schnittstellen
  - Verwendung eines funktionierenden Ersatzsystem (z.B. hot standby)
- Rückwärtsfehlerkorrektur
  - Versuch in Zustand zurückzugelangen, der vor Fehler
    - Aktion nochmals durchzuführen (z.B. falsche Berechnung)
    - u.U. transparent für den Benutzer
  - Extremfall $\rightarrow$ Notbetrieb (eingeschränkte Fkt.), Neustart
- Externe Fehlerkorrektur
  - Benutzer
  - externe Systeme (z.B. übergeordnet)

Benutzerschnittstelle

- Validierung der eingegebenen Daten:
  - Plausibilitätsüberprüfungen, Fehlererkennung, Fehlerkorrektur
- Entwurf/Layout
  - wichtige Informationen im Zentrum des Blickfeldes
  - Farben (rot, grün, orange), Blinken,…
- zusätzliche Erläuterungen
- geführte Interaktion (Umsetzung der Richtlinien)
- Mehrfacheingaben, Rückfragen/Bestätigung vor Ausführung
- Visualisierung, Simulationen

Entwicklungsprozess

U.U. jeweils durch **verschiedene** Teams

- Entwicklung mehrerer Spezifikationen
  - u.U. formale Spezifikation und Verifikation
- Mehrere HW Plattformen
- Entwicklung mehrerer SW Entwürfe
- Einsatz verschiedener Tools
  - OS, Programmiersprachen, Compiler, Debugger
- Verschiedene Testverfahren

Fazit

- Viele Gründe sprechen für *verteilte Systeme*!
- aber:

Fazit

- Viele Gründe sprechen für *verteilte Systeme*!
- aber:
  - es können viele Fehler auftreten

Fazit

- Viele Gründe sprechen für *verteilte Systeme*!
- aber:
  - es können viele Fehler auftreten
  - viel Know-How nötig ($\rightarrow$ Personal!)

Fazit

- Viele Gründe sprechen für *verteilte Systeme*!
- aber:
  - es können viele Fehler auftreten
  - viel Know-How nötig ($\rightarrow$ Personal!)
  - Entwicklung ist aufwändig, komplex und (daher) teuer

Fazit

- Viele Gründe sprechen für *verteilte Systeme*!
- aber:
  - es können viele Fehler auftreten
  - viel Know-How nötig ($\rightarrow$ Personal!)
  - Entwicklung ist aufwändig, komplex und (daher) teuer
- Entwicklung eines (single-threaded) Programms auf *einem* Computer ist (relativ) einfach!