Modul: 40755 - Verteilte Systeme

Technische Universität Berlin *SoSe 2021, 8. August 2022*

Studentische-Zusammenfassung

Version 1.0

Mitwirkende:

Michel Schnitker
Martin Hübner
Emily Seebeck
Matteo Gätzner
Nicolas Look

Vorwort

Liebe Lesende,

diese Zusammenfassung haben wir in Vorbereitung auf unsere Prüfung im Sommersemester 2021 verfasst. Wir haben sie nach bestem Wissen und Gewissen geschrieben und sie hat uns bei der Vorbereitung auf die Prüfung gute Dienste geleistet.

Bitte beachtet dennoch, dass diese Zusammenfassung Fehler und / oder Ungenauigkeiten enthalten kann. Auch erheben wir keinen Anspruch darauf, dass der komplette Stoff der Vorlesung abgedeckt sei.

Besonders hervorheben möchten wir die Fragen am Ende jedes Kapitels. Da es bei der Freitagsrunde leider nur wenige Altklausuren gab, haben wir uns kurzerhand eigene Fragen zum Stoff eines jeden Kapitels überlegt. Hierbei haben wir uns teilweise von Fragenkatalogen aus der Fachliteratur inspirieren lassen.
(→ *Tannenbaum. Distributed Systems: Principles and Paradigms*)

Zu den entsprechenden Fragen geben wir auf der Folgeseite exemplarische Antworten mit. Auch hier gilt wieder: Alle Angaben nach bestem Vermögen, aber ohne Gewähr. Wir empfehlen, die Fragen zuerst ohne die Lösungen zu beantworten. Denn nur so werden einem selbst auch die Stellen bewusst, an denen man noch nachbessern sollte.

Zuletzt haben wir noch ein Anliegen an Euch! Gerne möchten wir dieses Dokument aktuell halten, verbessern und mögliche Fehler korrigieren. Besucht dafür gern unser Repo oder einen aktuellen Fork.

https://git.tu-berlin.de/the-vs-writing-collective/zusammenfassung-verteilte-systeme

Viel Erfolg!

*Die Autor*innen*

Inhaltsverzeichnis

1 Einführung **5**
Kapitel 1.1. Charakteristika und Motivation . . . 6
(a) Definitionen Verteilter Systeme . . . 8
(b) Begriffsdefinitionen . . . 8
Kapitel 1.2. Challenges and Examples . . . 9
(a) Challenges . . . 9
(b) Vor- und Nachteile . . . 12
Fragen Kapitel 1 . . . 13

2 Algorithmen in Verteilten Systemen **19**
Kapitel 2.1. System Models . . . 20
(a) Physisches Basismodell (Physical Model) . . . 20
(b) Architekturmodell (Architectural Model) . . . 20
(c) Fundamentale Modelle (Fundamental Models): . . . 20
Fragen Kapitel 2.1 . . . 23
Kapitel 2.2. Cooperation . . . 26
(a) Message Passing . . . 26
(b) Externe Datendarstellung (External Data Representation) . . . 27
(c) Message Queues . . . 27
Fragen Kapitel 2.2 . . . 28
Kapitel 2.3. Time . . . 31
(a) Definitionen . . . 31
(b) Methoden um Uhren zu synchronisieren: . . . 31
Fragen Kapitel 2.3 . . . 35
Kapitel 2.4. Replication . . . 38
(a) Motivation . . . 38
(b) Definitionen . . . 38
(c) Consistency-Scalability Tradeoff . . . 39
(d) Single Leader Replication . . . 39
(e) Leaderless Replication . . . 39
(f) Single Leader Replication vs. Leaderless Replication . . . 40
(g) Netzwerk Partitionierung . . . 40
(h) CAP-Theorem . . . 40
(i) CAP-Theorem Beispiele . . . 42
(j) Stickyness-Problem . . . 42
(k) Heuristiken für Clouds . . . 42
Fragen Kapitel 2.4 . . . 43
Kapitel 2.5. Consensus . . . 46
(a) Coordination and Agreement . . . 46
(b) Distributed Mutual Exclusion . . . 46

(c) Election (Dynamische Wahl des Masters) . . . 47
(d) Bully Algorithmus zur Wahl des Masters (erfüllt E1 und E2) . . . 47
(e) Consensus . . . 48
(f) End-To-End Arguments . . . 49
(g) Avoiding Coordination . . . 49
Fragen Kapitel 2.5 . . . 51

3 Architekturmodelle **54**
Kapitel 3.1. . . . 55
(a) Client/Server System . . . 55
(b) Two and Three Tier Models . . . 55
(c) Extended Client Server Systems . . . 56
(d) Interaction (Failure) Semantics . . . 57
(e) Klassifikation von Stateless Servern . . . 57
(f) Mögliche Priorisierungen bei der Fehlervermeidung . . . 57
Fragen Kapitel 3.1 . . . 59
Kapitel 3.2. Peer-to-Peer Systeme . . . 62
(a) Merkmale von P2P-Systemen . . . 62
(b) Kernkonzepte . . . 62
(c) Vorteile/Nachteile . . . 62
(d) P2P-Beispiel Gnutella . . . 62
Kapitel 3.3. Publish/Subscribe Systeme . . . 63
(a) Wichtige Merkmale . . . 63
(b) Weitere Fragen bezüglich Pub/Sub-Systemen . . . 63
(c) Notification Selection . . . 64
(d) Broker Network . . . 64
(e) Kafka - Beispiel für Service zur Datenstromverarbeitung . . . 65
(f) Hauptaufgaben . . . 65
(g) Kafka vs. traditionelle Messaging Systeme (JMS, AMQ) . . . 65
(h) Eigenschaften eines Streams . . . 65
Kapitel 3.4. CAP-Theorem . . . 66
(a) Microservices . . . 66
(b) Kubernetes . . . 66
(c) Availibility . . . 66
(d) Brewer's CAP-Theorem . . . 66
Fragen Kapitel 3.4 . . . 67

4 Programmiermodelle **70**
Kapitel 4.1. Intro . . . 71
(a) Remote Method Invocation . . . 71
Fragen Kapitel 4.1 . . . 73
Kapitel 4.2. Service Oriented Architecture . . . 75
(a) Workflows . . . 75
(b) Microservices . . . 75
Fragen Kapitel 4.2 . . . 76
Kapitel 4.3. Event Driven Architecture . . . 78
(a) Komponenten . . . 78
(b) Event Processing . . . 78
Fragen Kapitel 4.3 . . . 80
Kapitel 4.4. Enterprise Application Integration . . . 82

(a) Patterns 82
Fragen Kapitel 4.4 84

5 Sicherheit **87**
(a) Angriffsmethoden 88
(b) Communication- und Computer Security 88
(c) Kryptographie 89
(d) Authentifizierung auf einem lokalen System 92
(e) Kerberos 93
Fragen Kapitel 5 95

Kapitel 1

Einführung

Kapitel 1.1. Charakteristika und Motivation

Der Begriff der *Verteilten-Systeme* ist relativ alt. Er stammt noch aus einer Zeit, als nicht-Verteilte-Systeme, also Systeme die auf einem einzigen Rechner laufen, die Norm waren. Heutzutage gibt es kaum noch Software-Systeme, die nicht verteilt sind.

Alleine am Versenden einer E-Mail oder dem Aufruf einer Website sind viele unterschiedliche Computer beteiligt.

Wesentliche Eigenschaften von Verteilten Systemen sind unter anderem:

- **Verteiltheit**
  Eine große Anzahl stark spezialisierter Teilsysteme (Agenten), die zeitgleich arbeiten, z.B. DNS, Webserver, Firewall, etc.
- **Kooperation**
  Kooperation zwischen findet zwischen diesen Komponenten statt, um eine komplexe Aufgabe zu erfüllen.
- **Konkurrenz/Nebenläufigkeit**
  Zeitgleicher Wettbewerb der Komponenten um begrenzte Ressourcen:
  - Wer darf gerade senden?
  - Wer darf auf X zugreifen?

  Eine Koordination und Synchronisation der Aktivitäten ist erforderlich.
- **Keine globale Zeit**
  Es ist aufgrund räumlicher Distanz und mangelnder Präzision von Hardware-Uhren oft nicht möglich, Uhren hinreichend genau zu synchronisieren.
- **Fehlertoleranz**
  Da das Netzwerk generell unzuverlässig ist und die Komponenten im System verteilt sind können Fehler entstehen, mit denen umgegangen werden muss.
- **Heterogenität**
  Sowohl Software als auch Hardware können von ganz unterschiedlicher Art sein: Unterschiedliche Plattformen und Programmiersprachen sind die Norm. Entsprechend ist die Standardisierung von Schnittstellen notwendig.

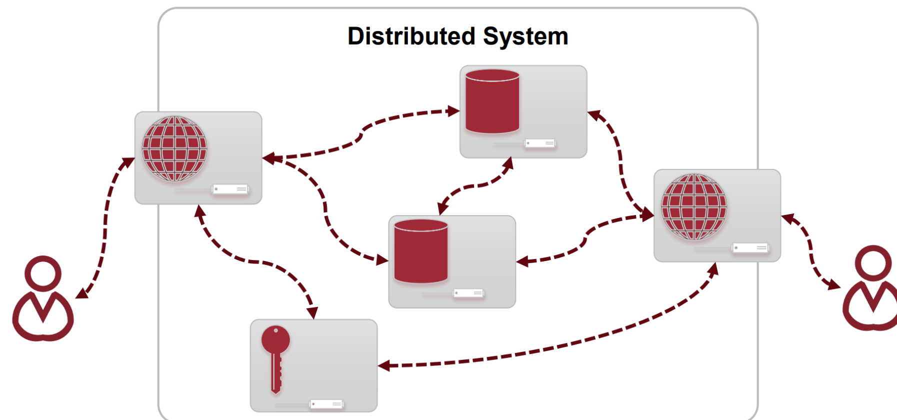


Abbildung 1.1: Ein Verteiltes System
Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 1.1 Slide 4

Verteilte Systeme lassen sich als eine Gruppen *autonomer*, intelligenter Agenten charakterisieren, die untereinander und mit der Außenwelt agieren.

Autonom heißt in diesem Kontext, dass die Einzelkomponenten selbständige Entscheidungen treffen. Sie sind auch ohne das Verteilte System „überlebensfähig“.

Diese Autonomie stellt eine wesentliche Herausforderung von Verteilten Systemen dar: Algorithmen und Protokolle für VS müssen diese Besonderheit stets beachten.

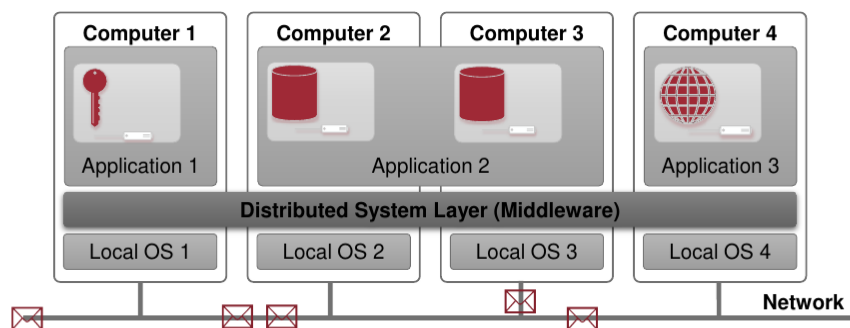


Abbildung 1.2: Ein Verteiltes System (physische Struktur)
Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 1.1 Slide 17

Zentral ist hierbei der Begriff Message-Passing: Jegliche Koordinierung und jeglicher Informationsaustausch findet *ausschließlich* durch das Senden und Empfangen von Nachrichten statt.

Obwohl Verteilte Systeme einige Herausforderungen und gesteigerte Komplexität mit sich bringen (siehe Eigenschaften), kann es sich dennoch lohnen sie zu erstellen:

- **Ressourcen teilen**
  Geteilte Ressourcen können deren Auslastung erhöhen und so die Effizienz steigern, z.B. Netzwerk-Drucker, gemeinsame Datenbanken, E-Mail-Server, etc.
- **Verarbeitung beschleunigen**
  Eine Aufgabe kann auf verschiedene Systeme aufgeteilt, dort parallel verarbeitet und

das Ergebnis anschließend wieder zusammengefügt werden. So ergeben sich oft große Zeitersparnisse.

- **Skalierbarkeit**

  Anpassung an erhöhte Nutzlast ist eventuell schwierig. Einbau stärkerer Hardware benötigt Offline-Zeit (vertical-scaling → *scale-up*).

  Verteilte Systeme ermöglichen horizontale Skalierung, *scale-out*: Lastverteilung durch hinzufügen mehrerer gleichartiger Systeme auf gleicher Stufe. Das ist sehr flexibel und im laufenden Betrieb möglich.

- **Stabilität und Redundanz**

  Verteilung von Berechnungen und Kopien der Daten schafft Sicherheit gegen Fehler und Ausfall einzelner Komponenten.

- **Modellierung der Realität**

  Verteilung auf Einzelkomponenten erlaubt eine bessere Anpassung an die Realität, z.B. bei Client-Server-Systemen, einzelnen Komponenten eines Autos oder einer Fabrik, etc.

(a) Definitionen Verteilter Systeme

Zeitgemäße Definition:

> *A distributed system is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate to perform a single or small set of related tasks.*

(b) Begriffsdefinitionen

- **Programm**

  Der Code den wir schreiben.

- **Prozess**

  Instantiierung (→ ausführen) des Codes.

- **Nachrichten**

  Kommunikation zwischen Prozessen.

- **Paket**

  Das Fragment einer Nachricht, das über das Kabel reist.

- **Protokoll**

  Formale Beschreibung von Format und Ablauf, damit zwei Knoten Nachrichten austauschen können.

- **Netzwerk**

  Infrastruktur, die Systeme miteinander verbindet. Besteht aus Routern, die durch Kommunikations-Links verbunden sind.

- **Komponente**

  Ein Prozess oder Hardware, die einen Prozess ausführt. Unterstützt Kommunikation zwischen Prozessen, speichert Daten, etc.

Kapitel 1.2. Challenges and Examples

Die folgenden Annahmen (nach Peter Deutsch / James Gosling) sind im Kontext verteilter Systeme *falsch*! Sie dürfen nicht getroffen werden und führen sonst zu schwer lösbaren Problemen:

- Das Netzwerk ist zuverlässig.
- Das Netzwerk ist sicher.
- Das Netzwerk ist homogen.
- Die Systemtopologie ändert sich nicht.
- Die Latenz ist quasi null.
- Bandbreite ist unbegrenzt da.
- Transport kostest nichts.
- Es gibt einen Administrator.

(a) Challenges

Die Eigenschaften Verteilter Systeme stellen einige besondere Anforderungen dar.

- **Heterogenität**

  Die Komponenten eines VS sind oft unterschiedlich: Betriebssysteme, Plattform, Programmiersprache und Implementierung unterscheiden sich.

  Heterogene Verteilte Systeme benötigen standardisierte Protokolle und Datenaustauschformate, die hardwareunabhängig sind. Weitere Lösungen sind z.B. Middleware (→ CORBA) oder Virtuelle Maschinen (Java-VM), welche die Hardware abstrahieren.

- **Offenheit**

  Offene Verteilte Systeme bieten einen hohen Grad an Erweiterbarkeit und Flexibilität. Dazu müssen die zugrunde liegende Protokolle und Mechanismen veröffentlicht werden (z.B. RFC-Dokumente für das Internet).

  Offene VS sind herstellerunabhängig und ermöglichen so die Implementierung auf beliebigen Plattformen bzw. in beliebigen Programmiersprachen.

- **Sicherheit**

  Nachrichten werden häufig über unsichere Netzwerke (z.B. das Internet) übertragen. Es muss

  - Privacy: Nachrichten werden nicht von unberechtigten Personen gelesen
  - Integrity: Nachrichten kommen so an wie der Sender sie abgeschickt hat
  - Availability: Das System ist immer bereit Anfragen zu akzeptieren

  sichergestellt werden.

  Wir erreichen das durch Verschlüsselung & Authentifikation, Prüfsummen und Redundanzen & Absicherung des Servers gegen DDoS.

- **Skalierbarkeit**

  Verteilte Systeme müssen auch bei steigenden Nutzer- / Komponentenzahl zuverlässig arbeiten. Wir erreichen das durch Caching, Virtualisierung und Automatisierung.

Es sollten immer hierarchische Strukturen bevorzugt werden, da diese besser skalieren. Des Weiteren sind dezentrale Algorithmen zu bevorzugen (→ keine Flaschenhälse).

- **Fehlerbehandlung**

  Bei steigender Anzahl beteiligter Komponenten steigt auch die Wahrscheinlichkeit von Fehlern. Manchmal kann man diese über z.B. Prüfsummen entdecken, oft bleibt **Fehlererkennung** jedoch ein Ratespiel (Server kaputt oder Netzwerk offline?).

  Entdeckte Fehler kann man versuchen zu verstecken (*maskieren*) oder zu entschärfen, indem z.B. eine Nachricht erneut gesendet wird oder Redundanzen genutzt werden (2 Festplatten statt einer, mehrere Server, etc.).

  Können Fehler nicht automatisch behoben werden, müssen wir sie *tolerieren*, also z.B. dem Benutzer die Entscheidung überlassen (→ Website nicht erreichbar, was nun?)

  Nach einem Fehler (z.B. Systemabsturz) muss sichergestellt werden, dass persistente Daten wieder in einen konsistenten Zustand überführt werden.

  Redundanz kann helfen, Fehlerfälle zu verringern und die Fehler-Toleranz zu erhöhen, z.B. durch mehrere mögliche Routen zum Internet, Replikation von DNS-Tabellen auf mehrere Hosts, Datenbankreplikation, etc.

- **Nebenläufigkeit**

  Um hohen Durchsatz zu erreichen ist Nebenläufigkeit zwischen Prozessen unvermeidbar. Nebenläufige Operationen, um z.B. auf die gleiche Ressource in einem Verteilten System zuzugreifen, müssen synchronisiert werden, um die Konsistenz der Daten zu wahren.

  Server und Komponenten müssen speziell designt werden, um in VS sicher arbeiten zu können. Es muss Möglichkeiten für Transaktionen und verteilte Mutual Exclusion geben.

- **Transparenz**

  Obwohl das Gesamtsystem aus sehr vielen unabhängigen Einzeldiensten bestehen kann, wird es als ein Ganzen wahrgenommen. Transparenz lässt sich in unterschiedliche Arten gliedern:

  - *Access transparency*

    Gleicher Zugriff auf lokale und remote-Ressourcen.

    Beispiel: Nextcloud-Integration für den Dateimanager.

  - *Location transparency*

    Der Speicherort ist nicht bekannt.

    Beispiel: Eine Fotosammlung ist über mehrere Datenbanken verteilt, es wird aber als eine, zusammenhängende, angezeigt.

  - *Concurrency transparency*

    Nebenläufigkeit von mehreren Prozessen wird vor den Clients versteckt.

    Beispiel: Zwei Clients greifen gleichzeitig auf Fotosammlung zu. Der Service hält die Daten konsistent und löst Konflikte auf.

  - *Replication transparency*

    Replikation der Daten ist für den User nicht erkennbar.

Beispiel: Eine Cloud-Service-Datenbank ist auf mehrere Server kopiert um Fehlertoleranz und Ausfallsicherheit zu steigern.

- *Failure transparency*

  Fehler werden maskiert und Service läuft weiter

  Beispiel: Bei Datenbankausfall werden Anfragen zu einer funktionierender Datenbank weitergeleitet.

- *Mobility (migration) transparency*

  Das Verschieben von Clients und / oder Ressourcen hat keinen Einfluss auf den Dienst.

  Beispiel: Roaming im Mobilfunk (Verbindung unterbricht nicht), Verschieben eines Fotos in andere Datenbank.

- *Performance transparency*

  Rekonfiguration des Dienstes bei Varianz in der Auslastung.

  Beispiel: Aktivierung von replizierten Servern bei Lastspitzen durch viele Nutzerzugriffe ($\rightarrow$ scaling-out).

- *Scaling transparency*

  Das System kann mit scale-out erweitert werden, ohne dass die Struktur oder die Algorithmen angepasst werden muss.

  Beispiel: Neue Datenbank-Server starten, statt bestehende zu vergrößern.

- *Network transparency* ist access + location transparency.

• **Quality of Service**

  Dienste sollen ihre Leistungen in bestimmter Qualität erbringen: Videos sollen hochauflösend gestreamt werden, Zahlungen nur wenige Sekunden dauern.

  Das Verteilte System muss also die Möglichkeiten bieten, um Ressourcen für bestimmte Aufgaben zu reservieren.

(b) Vor- und Nachteile

Vorteile	Nachteile
Reduktion der Kosten (→ scale-out ist günstiger als scale-up)	Komplexer Design- und Implementationsprozess
Verfügbarkeit	Probleme mit Heterogenität
Fehler tolerant	Schwer die Korrektheit zu verifizieren
Skalierbar	Komplexe Kommunikationssysteme
Sehr performant, da parallel gearbeitet wird	Die Umstellung von einem zentralisierten System zu einem dezentralisiertem ist oft schwierig
Modulare Software	Sicherheits-Probleme
Konsistenz mit organisatorischen Strukturen	Hohe Operations- und Servicekosten
Unabhängigkeit von Produzenten	

Tabelle 1.1: Vor- und Nachteile eines Verteilten Systems

Fragen Kapitel 1

Frage 1

Was zeichnet ein verteiltes System aus?

Frage 2

Was ist ein Protokoll?

Frage 3

Welche Probleme treten bei VS auf?

Frage 4

Wozu braucht man verteilte Systeme?

Frage 5

Welche Vor- und Nachteile bieten VS?

Frage 6

Was sind Basisanforderungen an ein verteiltes System?

Frage 7

Erkläre (distribution) transparency und gib drei Arten von Transparenz in einem verteilten System an.

Frage 8

Eine Definition von VS beschreibt sie als eine Gruppe von Computern, die als ein einzelnes System erscheinen, der User also nicht erkennen kann, dass es sich um mehrere Computer handelt.

Gebe ein Beispiel an, bei dem dieser Umstand sehr nützlich ist.

Frage 9

Erkläre die Begriffe „program“, „process“ und „component“ im Kontext von verteilten Systemen.

Frage 10

Nenne 3 Probleme, die bei dem Design von verteilten Systemen noch stärker als beim Design monolithischer Systeme auftreten und erläutere deren Lösungen.

Frage 11

Erläutere die Begriffe „scale-up“ sowie „scale-out“ und erkläre, welches der Vorgehen sich bei verteilten Systemen anbietet.

Frage 12

Erkläre wie ungewollte Redundanz durch die Verwendung verteilter Systeme vermieden kann und an welchen Stellen Redundanz in einem verteilten Systemen vorteilhaft ist.

Frage 13

Welche falschen Annahmen werden oft über verteilte Systeme getroffen?

Frage 14

Welche Aspekte der Sicherheit müssen beim Design von verteilten Systemen beachtet

werden?

Frage 15

Was bedeutet der Begriff „Transparency“ im Kontext von verteilten Systemen?

Frage 16

Beschreibe genau, was mit einem skalierbaren Verteilten System gemeint ist.

Frage 17

Skalierbarkeit kann durch verschiedene Techniken erreicht werden. Welche?

Frage 18

Wie verhindert man ständige Anpassungen an neue Implementierungen in verteilten Systemen?

Frage 19

Was der Unterschied zwischen Speedup und Scaling?

Antworten Kapitel 1

Antwort 1

Was zeichnet ein verteiltes System aus?

Ein verteiltes System ist ein System, welches parallel auf mehreren Rechnern läuft. Einem Nutzer bzw. einer Anwendung erscheint es jedoch wie ein monolithisches System.

Antwort 2

Was ist ein Protokoll?

Ein Protokoll ist eine Festlegung von Format und Ablauf der Kommunikation zwischen Prozessen.

Antwort 3

Welche Probleme treten bei VS auf?

- Nachrichten können verloren gehen
- Nachrichten können von unautorisierten Personen gelesen und gesendet werden
- Es gibt keine globale Uhrzeit, um die Reihenfolge von Events festzulegen.
- Es ist oft nicht möglich festzustellen, ob eine Nachricht verloren gegangen ist oder ob ein Kanal oder ein Knoten kaputt ist
- Bei der Replikation von Daten können Inkonsistenzen auftreten

Antwort 4

Wozu braucht man verteilte Systeme?

VS sind oft die einzige sinnvolle Möglichkeit Skalierungsprobleme zu lösen: Nutzer können z.B. auf andere Komponenten umgeleitet werden, um Last abzufangen.
VS sind oft leistungsfähiger als monolithische Systeme und können Ressourcen besser auslasten (Netzwerkdrucker, statt Drucker für jeden Mitarbeiter).

Antwort 5

Welche Vor- und Nachteile bieten VS?

Siehe Table 1.1

Antwort 6

Was sind Basisanforderungen an ein verteiltes System?

- Skalierbarkeit
- 24/7 Verfügbarkeit
- Portierbarkeit (egal welches System, egal welche Hardware)
- Persistenz – unveränderlich gültig

Antwort 7

Erkläre (distribution) transparency und gib drei Arten von Transparenz in einem verteilten System an.

Distribution Transparency beschreibt die Abstraktion der beteiligten Systeme: Für Nutzer/Anwendung sieht es so aus, als kommunizierte man mit einzelnem, nicht-verteilten,

System.

Für Beispiele siehe den Unterpunkt Transparenz

Antwort 8

Eine Definition von VS beschreibt sie als eine Gruppe von Computern, die als ein einzelnes System erscheinen, der User also nicht erkennen kann, dass es sich um mehrere Computer handelt.

Gebe ein Beispiel an, bei dem dieser Umstand sehr nützlich ist.

Bei Engpässen oder Systemabstürzen können wir Überlast bzw. Dysfunktion für Nutzer verbergen und stillschweigend skalieren oder umleiten.

Antwort 9

Erkläre die Begriffe „program", „process" und „component" im Kontext von verteilten Systemen.

Siehe Begriffsdefinitionen

Antwort 10

Nenne 3 Probleme, die bei dem Design von verteilten Systemen noch stärker als beim Design monolithischer Systeme auftreten und erläutere deren Lösungen.

- **Latenz**

  Nachrichtenübertragung dauert viel länger.

  Lösung: Asynchrone Kommunikation.

- **Konsistenz**

  Nachrichten könnten durch Netzwerk korrumpiert / verändert werden.

  Lösung: Checksummen einbauen.

- **Verlust**

  Nachrichten können im Netzwerk verloren gehen.

  Lösung: Nochmal senden, falls kein ACK erfolgt.

- **Sicherheit**

  Offenheit von VS biete Manipulationsmöglichkeiten durch bösartige Akteure.

  Lösung: Verschlüsselung einsetzen.

- **Verteiltheit**

  Verteiltheit bietet mehr Angriffsfläche.

  Lösung: Security-Modelle entwerfen und einsetzen.

Antwort 11

Erläutere die Begriffe „scale-up" sowie „scale-out" und erkläre, welches der Vorgehen sich bei verteilten Systemen anbietet.

- **scale-up**

  Performancesteigerung durch Tausch zu stärkerer Hardware.

- **scale-out**

  Performancesteigerung durch Hinzufügen weiterer Komponenten-Instanzen.

- Scale-out bedeutet im Gegensatz zu scale-up außerdem keine Ausfallzeit für das System.

Antwort 12

Erkläre wie ungewollte Redundanz durch die Verwendung verteilter Systeme vermieden kann und an welchen Stellen Redundanz in einem verteilten Systemen vorteilhaft ist.

Verringerung der Redundanz durch mehrfache Verwendung der gleichen Komponente: Netzwerkdrucker statt Drucker für Einzelpersonenen → bessere Ausnutzung der Ressourcen.

Dabei kann Redundanz positiv für Performance und Ausfallsicherheit sein (→ Replikation, Cache).

Antwort 13

Welche falschen Annahmen werden oft über verteilte Systeme getroffen?

- Netzwerk ist sicher, zuverlässig und homogen
- Latenz ist 0
- Bandbreite ist unbegrenzt
- Topologie ändert sich nicht
- Transport kostet nichts
- Es gibt einen Admin

Antwort 14

Welche Aspekte der Sicherheit müssen beim Design von verteilten Systemen beachtet werden?

- **Privacy**

  Außenstehende können nicht auf Inhalte zugreifen
- **Integrity**

  Schutz gegen Korruption der Inhalte
- **Availibility**

  Schutz gegen Angriffe (z.B. DDOS-Attacken)

Antwort 15

Was bedeutet der Begriff „Transparency“ im Kontext von verteilten Systemen?

Wenn Dinge transparent sind, dann sind sie vor dem Nutzer bzw. der Anwendung versteckt.

Im Webbrowser ist es z.B. für den Nutzer nicht ersichtlich, ob Elemente aus dem Cache oder vom Webserver geladen werden. Der Cache arbeitet also transparent.

Antwort 16

Beschreibe genau, was mit einem skalierbaren Verteilten System gemeint ist.

Ein skalierbares Verteilten System ist ein System, das auch bei ansteigenden Nutzerzahlen effizient bleibt ohne, dass „Quality of Service“ zu stark eingeschränkt wird.

Zusätzlich ist es möglich die physikalischen Ressourcen sowie Software Ressourcen zu erhöhen ohne, dass die Performance darunter leidet bzw. Bottlenecks entstehen.

Die Ressourcen-Erhöhung muss implementierbar sein bevor die Ressourcen ausgehen.

Antwort 17

Skalierbarkeit kann durch verschiedene Techniken erreicht werden. Welche?

Dies kann durch Modularität sowie Loose Coupling erreicht werden.

Antwort 18

Wie verhindert man ständige Anpassungen an neue Implementierungen in verteilten Systemen?

Man definiert für die Kommunikation zwischen den Komponenten einheitliche Schnittstellen und Standards. Diese sollten möglichst langlebig sein, sodass der Anpassungsaufwand nur bei fundamentalen Änderungen gemacht werden muss.

Antwort 19

Was der Unterschied zwischen Speedup und Scaling?

- **Speedup**
  Datenmenge und Nutzerzahl bleiben gleich, aber die Antwortzeit verkürzt sich. Für Nutzer erscheint das System performanter.
- **Scaling**
  Datenmenge und Nutzerzahl steigen, Antwortzeit bleibt jedoch konstant.

Kapitel 2

Algorithmen in Verteilten Systemen

Kapitel 2.1. System Models

System-Modelle machen es einfacher zu verstehen wie ein System funktioniert und helfen die korrekte Funktionsweise nachzuweisen.

(a) Physisches Basismodell (Physical Model)

Bezeichnet eine Darstellung der zugrunde liegenden Hardwareelemente. Dies ist meist eine Darstellung der wichtigsten Services als Knoten und deren mögliche Kommunikation zwischen einander als Kanten.

(b) Architekturmodell (Architectural Model)

Beschreibt die einzelnen Einheiten eines Verteilten Systems und wie sie mit anderen Einheiten kommunizieren. Ebenfalls enthalten sein können Rollen, Verantwortlichkeiten, Kommunikationsparadigmen wie Socket / Channel und auf welcher physischen Infrastruktur diese Einheiten platziert sind.

So lassen sich Strategien und Probleme genauer darstellen.

- **Communicating entities**
  - Ausschnitt eines Systems, wobei verschiedene Detailgrade erläutert werden. Beispiel: Objekte, Komponenten, Web Services.
- **Communication paradigms**
  - Interprozesskommunikation (IPC) z.B. low-Level Channel / Socket
  - Indirekte Kommunikation über z.B. einen Broker oder Proxy
  - Fernaufrufe (Remote Invocation)
- **Roles and responsibilities**
  - Client-Server
  - Peer-to-Peer

(c) Fundamentale Modelle (Fundamental Models):

Gibt eine abstrakte vereinfachte Beschreibung über Aspekte zur Gestaltung wieder.

- **Interaktionsmodell (Interaction Model)**
  Generelle Kommunikation bei Zeit basierten Events.
  - **Synchron**
    Faktoren wie Taktdrift, Prozessausführungsgeschwindigkeit sowie Übertragungszeiten sind bekannt. Timeouts können eingeschätzt werden.

    Meistens sind viele Faktoren jedoch nicht bekannt oder zu ungenau weshalb synchrones Interaktionsmodell oft nicht anwendbar ist.
  - **Asynchron**
    Geschwindigkeiten und Zeiten unbekannt und werden auch nicht geschätzt.
- **Fehlermodell (Failure Model)**
  Definiert die Möglichkeiten, bei denen Fehler auftreten können.

- Maskierung (Masking)
- Verlässlichkeit (Reliability)

Art des Fehlers	beeinflusst	Beschreibung
Fail-stop	Prozess	Der Prozess hält und bleibt angehalten. Andere Prozesse können diesen Zustand entdecken.
Crash	Prozess	Prozess hält und bleibt angehalten. Andere Prozesse können diesen Zustand möglicherweise nicht entdecken.
Omission	Channel	Eine Nachricht wird in den ausgehenden Nachrichtenpuffer gelegt, allerdings kommt diese Nachricht nie an.
Send-omission	Prozess	Ein Prozess schließt die sende-Operation ab, allerdings wird nie etwas in den ausgehenden Nachrichtenpuffer gelegt.
Receive-omission	Prozess	Eine Nachricht wird in den eingehenden Nachrichtenpuffer gelegt, allerdings bekommt der Prozess die Nachricht nicht.
Willkürlich (Byzantinisch)	Prozess / Channel	Der Prozess / Channel weist willkürliches Verhalten auf. Das bedeutet, dass beliebige Nachrichten gesendet werden können, ein Protokoll nicht beachtet, etc.

Abbildung 2.1: Art der Fehler

- **Sicherheitsmodell (Security Model)**

  Beschreibt Sicherheitsvorkehrungen um Prozesse und Daten abhängig von der Gefahrenklasse zu sichern.

  Die niedrigste Gefahrenklasse sind automatisierte Angriffe wie z.B. von Skript Kiddies, die verhindert werden müssen.

  Die höchste Gefahrenklasse enthält versierte, geplante Angriffe von Konkurrenten und Geheimdiensten, die sich praktisch nicht verhindern lassen.

  - Bedrohungen für Prozesse
  - Bedrohungen für Kommunikationskanäle: Feind kann Nachrichten lesen, kopieren, ändern oder injizieren.

  Abwehr von Sicherheitsbedrohungen

  - **Cryptography and shared secrets**

    Oft basierend auf gemeinsamen Geheimnissen → geheime Schlüssel (symmetrisch vs. asymmetrisch)

  - **Authentication**

    Authentifizierung von Nachrichten basierend auf gemeinsamen Geheimnissen und Verschlüsselung.

  - **Secure channels**

    Basierend auf Verschlüsselung und Authentifizierung werden zwei Prozesse verbunden. Die Identitäten sind gegenseitig bekannt.

Replay-Attacks werden mit eindeutigen Zeitstempeln (physisch oder logisch) in den Nachrichten verhindert.

Fragen Kapitel 2.1

Frage 1

Erläutere die System-Modelle „architectural model“ und „interaction model“.

Frage 2

Welche Systemmodelle gibt es?

Frage 3

Was bedeutet Remote Invocation?

Frage 4

Was sind die signifikanten Faktoren die die Leistung eines interaktiven Prozess ausmachen?

Frage 5

Welche Klassifizierung von Fehlern zwischen Prozessen kennen sie?

Frage 6

Was können wir bei Fehlern tun?

Frage 7

Welche Aspekte der Sicherheit müssen beim Design von verteilten Systemen beachtet werden?

Frage 8

Welche Arten von Fehlern gibt es? Worin unterscheiden sie sich?

Frage 9

Welcher Angreifer hat das höchste Skill-Level? Welche Maßnahmen werden für einen solchen Fall getroffen?

Frage 10

Welche Rolle spielt Middleware in VS?

Antworten Kapitel 2.1

Antwort 1

Erläutere die System-Modelle „architectural model“ und „interaction model“.

Siehe Architekturmodell und Fundamentale Modelle

Antwort 2

Welche Systemmodelle gibt es?

- Physikalisches modell
- Architekturmodell
- Interaktionsmodell
- Sicherheitsmodell

Antwort 3

Was bedeutet Remote Invocation?

Remote Invocation beschreibt einen Aufruf den ein lokaler Prozess macht, der dann auf einem anderen Prozess ausgeführt wird. Der andere Prozess sendet dann das Ergebnis mit einer Nachricht zurück, z.B. Request-Reply (Client / Server), Remote Procedure Calls, Remote Method Invocation (Objekt-Orientierung).

Antwort 4

Was sind die signifikanten Faktoren die die Leistung eines interaktiven Prozess ausmachen?

- Latenz
- Bandbreite
- Jitter (Variation in der Übertragungsgeschwindigkeit)
- Zeitliche Ordnung
- synchron (eher selten) bzw. asynchron (das ist eher die Regel)

Antwort 5

Welche Klassifizierung von Fehlern zwischen Prozessen kennen sie?

- **Omission**
  Fehler durch Unterlassen
- **Byzantinisch**
  Willkürliche Fehler - falsche Daten erzeugen falsche Aktionen - schwer erkennbar
- **Timing**
  Zeitliche Fehler die insbesondere bei synchronen System auftreten, siehe Fundamentale Modelle

Antwort 6

Was können wir bei Fehlern tun?

Wir können die Fehler beheben, weniger schlimm machen oder maskieren.

Beispiel Webbrowser: Wenn eine Seite nicht erreichbar ist, versuchen wir mehrmals erneut sie zu erreichen. Kann die Seite dann immer noch nicht erreicht werden, benachrichtigen

wir den Nutzer.

Antwort 7

Welche Aspekte der Sicherheit müssen beim Design von verteilten Systemen beachtet werden?

Aufgrund der Kommunikation zwischen den Rechnern innerhalb des VS, ist es besonders wichtig die Integrität von Message und Prozess zu schützen sowie den Zugriff zu sichern.

Antwort 8

Welche Arten von Fehlern gibt es? Worin unterscheiden sie sich?

Siehe Fundamentale Modelle

Antwort 9

Welcher Angreifer hat das höchste Skill-Level? Welche Maßnahmen werden für einen solchen Fall getroffen?

Die höchste Gefahrenklasse und damit das höchste Skill-Level haben Regierungen mit Hilfe von Geheimdiensten oder auch Konkurrierende Firmen die sich professionelle Hilfe leisten können. In diesem Fall ist eine direkte Abwehr nicht möglich.

Kapselung von wichtigen Systemen und ein Wiederanlaufkonzept ist gegen jede Gefahrenklasse zu implementieren.

Antwort 10

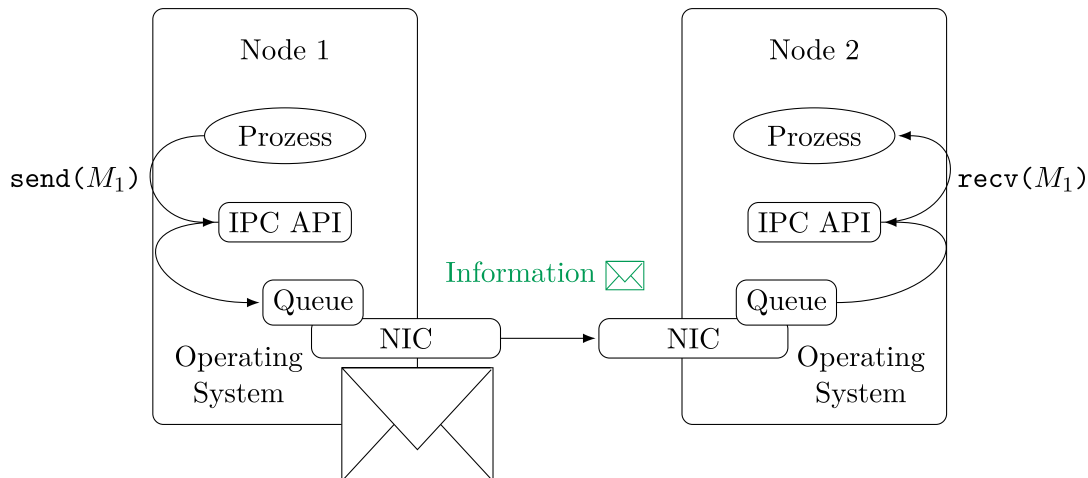
Welche Rolle spielt Middleware in VS?

Die Middleware ist eine anwendungsneutrales Programm, welches so zwischen Anwendungen vermitteln, dass die Komplexität dieser Anwendungen und ihre Infrastruktur verborgen werden.

Kapitel 2.2. Cooperation

(a) Message Passing

Für das Übermitteln von Messages stellt das Betriebsystem über die IPC-API `send()` und `receive()`, hier abgekürzt mit `recv()`, Funktionen bereit.



Synchrone Kommunikation

`send()` und `recv()` sind blockierende Funktionen. Nach dem Senden einer Message wird die Kommunikation pausiert bis die entsprechende Antwort empfangen wird.

Vorteile sind gesteigerte Transparenz, es muss nur eine Nachricht gepuffert werden und unterschiedliche Verarbeitungsleistung werden durch Warten ausgeglichen.

Nachteile sind vor allem die enge Kopplung da Sender und Empfänger gleichzeitig laufen müssen. Zusätzlich entsteht eine große Gefahr durch Deadlocks.

Asynchrone Kommunikation

Beim asynchronen Kommunizieren ist das Senden nicht blockierend. Das Empfangen kann blockierend oder nicht blockieren sein.

Vorteil ist, dass kein gleichzeitiger Betrieb von Sender und Empfänger erforderlich ist. Hohe Parallelität, reduzierte Gefahr von Deadlocks.

Die Synchronisierung gestaltet sich hingegen als schwieriger und durch den benötigten Puffer werden Buffer-Overflows möglich.

Charakteristiken

Je nach Protokoll und Anwendungen müssen bestimmte Nachrichten in einer vorbestimmten Reihenfolgen ankommen, da es sonst zu Fehlern kommen könnte. Bei der Nachrichtenweiterleitung redet man von der Verlässlichkeit (Reliability).
Verlässlichkeit bedeutet, dass Nachrichten genau einmal unbeschädigt ankommen sowie, dass die Zustellung von Nachrichten garantiert ist auch wenn Pakete verworfen werden oder verloren gehen.

Dabei gilt: Paket $\neq$ Nachricht!

(b) Externe Datendarstellung (External Data Representation)

Sockets unterstützen nur die Übertragung von Byte-Streams (TCP) oder Byte-Arrays (UDP). Somit müssen Daten in beliebiger Form zu einem übertragbaren Format umgewandelt werden.

Das Marshalling bezeichnet den Prozess, die Programm spezifische Datenstruktur in eine Externe Repräsentation zu übersetzen. Das Unmarshalling bezeichnet somit die Rückübersetzung.

(c) Message Queues

Empfänger sind oftmals nicht durchgehend verfügbar. In solchen Fällen schaffen Message Queues Abhilfe. Sie speichern Nachrichten zwischen, bis der Empfänger wieder verfügbar oder empfangsbereit ist. Lastspitzen lassen sich so entschärfen. Man unterscheidet hier in zwei Arten der Entkopplung.

- Räumliche Entkopplung (Space decoupling): Der Absender muss den tatsächlichen Empfänger nicht kennen.
- Zeitliche Entkopplung (Time decoupling): Sender und Empfänger müssen nicht gleichzeitig verfügbar sein. (offline oder beschäftigt)

Das Weiterleiten der Messages erfolgt dann One-to-One, One-to-Many, Many-to-One oder Many-to-Many.

Fragen Kapitel 2.2

Frage 1

Was ist Message Passing?

Frage 2

Was sind die wesentlichen Merkmale von Message Passing?

Frage 3

Beschreibe die charakteristischen Merkmale von synchroner vs. asynchroner Kommunikation!

Frage 4

Nennen Sie je 2 Vorteile und 2 Nachteile der synchronen (bzw. asynchronen) Kommunikation!

Frage 5

Auf welcher Basis ist Message Passing in der Praxis bei verteilten Systemen implementiert?

Frage 6

Was bedeutet marshalling / unmarshalling?

Frage 7

Wozu benötigt man indirekte Kommunikation über Message Queues?

Frage 8

Wie finden Sie den Status einer Session von Queue Transaktionen heraus und welche Aktionen können Sie bei Fehlern unternehmen?

Antworten Kapitel 2.2

Antwort 1

Was ist Message Passing?

Message Passing ist der Datenaustausch zwischen einem empfangenden und einem sendenden Prozess über Nachrichten.

Antwort 2

Was sind die wesentlichen Merkmale von Message Passing?

Beim Message Passing gibt es eine Zieladresse, meist die IP-Adresse und Port. Dabei unterscheidet man zwischen synchroner und asynchroner Kommunikation.

Bei der synchronen blockiert der Sender, bei der asynchronen läuft er nach dem senden weiter. Die `send(msg)` und `recv(msg)` Operationen werden dabei vom Betriebssystem über eine IPC API bereitgestellt.

Antwort 3

Beschreibe die charakteristischen Merkmale von synchroner vs. asynchroner Kommunikation!

Synchrone Kommunikation: Der sendende Prozess blockiert bis er eine Antwort erhält. Der Sender weiß dann, dass der Empfänger seine Nachricht erhalten hat. Außerdem muss nur eine Nachricht muss gebuffert werden, was es einfach zu implementieren macht.

Das führt aber zu Tight Coupling: Der Sender und Empfänger müssen gleichzeitig laufen. Entsprechend blockiert der Sender, was zu geringerem Parallelismus führt.

Antwort 4

Nennen Sie je 2 Vorteile und 2 Nachteile der synchronen (bzw. asynchronen) Kommunikation!

Synchrone Kommunikation: Siehe oben.

Asynchrone Kommunikation:

Loose Kopplung, Sender und Empfänger müssen nicht gleichzeitig online sein, erhöhter Parallelismus, geringeres Risiko für Deadlocks.

Buffer overflow möglich, Sender weiß nicht wann eine Nachricht ankommt $\Rightarrow$ Synchronisation schwieriger.

Antwort 5

Auf welcher Basis ist Message Passing in der Praxis bei verteilten Systemen implementiert?

Internetprotokolle TCP/IP und UDP.

Oftmals wird das Message Passing über eine Queue realisiert, welche JSON-Nachrichten bekommt. Diese Nachricht kann aber auch jedes andere Format, welches serialisiert ist, haben.

Dabei werden die einzelnen Nachrichten über TCP an die Queue gesendet. Eine solche Queue kann z.B. durch JMS realisiert werden.

Antwort 6

Was bedeutet marshalling / unmarshalling?

Marshalling bezeichnet die Übersetzung der lokaler Darstellung der Daten in eine (serialisierte) extern verschickbare Form. Unmarshalling bezeichnet die entgegengesetzte Operation.

Antwort 7

Wozu benötigt man indirekte Kommunikation über Message Queues?

Zur zeitlichen Entkopplung, da nicht jeder Empfänger permanent erreichbar ist. Zur räumlichen Entkopplung, sodass Sender und Empfänger einander nicht kennen direkt müssen.

Antwort 8

Wie finden Sie den Status einer Session von Queue Transaktionen heraus und welche Aktionen können Sie bei Fehlern unternehmen?

Mittels Correlation-ID's: R, R_e, R_d: Diese ID's werden hochgezählt und dann miteinander verglichen. Wenn alle gleich sind dann ist kein Fehler aufgetreten.

Falls aber einer aufgetreten ist, muss eine Fehlerbehandlung durchgeführt werden. Beispiel: Wenn $R \neq R_e$ gilt, dann wird die Nachricht nochmal gesendet.

Kapitel 2.3. Time

(a) Definitionen

- **Clock Skew**
  Differenz zwischen zwei Uhrzeiten
- **Clock Drift**
  Tatsache, dass verschiedene Uhren verschieden schnell laufen wegen physikalischen Unterschieden
- **Drift Rate**
  Rate zu der sich eine lokale Uhrzeit von der Referenz-Uhrzeit entfernt
- **Approximation der physikalischen Zeit**
  $C_i(t) = \alpha H_i(t) + \beta$ wobei H_i die Hardware-Uhr Uhrzeit ausgibt
- **Event**
  Einzelne Aktion die ein Prozess ausführt, z.B. Kommunikation (senden/empfangen), State-Änderung (Variablen, Dateien)
- **Monotonie**
  Eine Uhr geht niemals rückwärts, sonst könnte die lokale Ordnung von Events zerstört werden
- **Korrektheit einer Hardware-Uhr**
  Drift Rate von der betrachteten Uhr ist nach oben und unten begrenzt

(b) Methoden um Uhren zu synchronisieren:

Externe Synchronisation

Definition: Synchronisation mit einer externen, autoritären Uhr S, sodass $|S(t) - C(t)| < D$ für jede lokale Uhr C zu jeder Zeit gilt.

Cristians Algorithmus (Intranet)

1. Client C fragt Zeit-Server S nach der Uhrzeit
2. S sendet C seine Uhrzeit t_S
3. C berechnet die round-trip-time T_{round}
4. C setzt seine lokale Uhrzeit auf $t_S + \frac{1}{2}T_{\text{round}}$

Probleme: Single-Point-Of-Failure (Lösung: N Zeit-Server benutzen), inkorrekte Zeit-Server, Präzision $= \pm(T_{\text{round}}/2) - m$ wobei $m =$ minimale Übertragungszeit

NTP (Internet)

- Ziele: Skalierbare, verlässliche Synchronisation von Computer-Uhren mit UTC über das Internet
- Architektur: Hierarchische Struktur von wenigen Referenz-Uhren zu Root-Uhren zu Stratum-1-Uhren, Stratum-2-Uhren usw. bis zum Heim-Computer
- Modi:

  - Multi-Cast-Mode: Langsam, gut in high-speed LAN's
  - Precedure-Call-Mode: Höhere Präzision, ähnlich zu Cristians Algorithmus
  - Symmetric-Mode: Höchste Präzision, Server frägt mehrmals nach Zeit, schmeist unübliche Werte weg und setzt seine Uhrzeit auf den Durchschnitt der übrigen Werte

Interne Synchronisation

Definition: Synchronisation nur mit lokalen Uhren

Berkley-Algorithmus (Intranet)

1. Master sammelt Uhrzeiten von seinen Slaves
2. Master berechnet durchschnittliche Uhrzeit (inklusive seiner Eigenen)
3. Master sendet (durchschnittliche Uhrzeit - von Slave i gesammelte Uhrzeit) an Slave i
4. Slaves setzen ihre Uhrzeit auf ihre aktuelle Uhrzeit + den erhaltenen Wert

Logische Uhren

- Warum logische Uhren benutzen? Oft ist die richtige Uhrzeit unwichtig, nur die Ordnung von Events ist wichtig
- Definitionen:
  - Prozess $i = p_i$
  - Totale Ordnung von Events für Prozess i: $\rightarrow_i = \text{history}(p_i) = h_i = \left\langle e_i^0, \ldots, e_i^N \right\rangle$
  - Happened-before-Relation: $\rightarrow$ = „HB"
- Regeln:
  - **HB1**
    Wenn auf einem lokalen Prozess i ein Event vor einem anderen passiert ist, dann ist das erste Event auch global vor dem zweiten Event passiert.
    $$\exists p_i : e \rightarrow_i e' \Rightarrow e \rightarrow e'$$
  - **HB2**
    Wenn ein Prozess eine Nachricht an einen anderen Prozess geschickt hat, dann ist das `send`-Event vor dem `recv`-Event passiert.
    $$\texttt{send}(m) \rightarrow \texttt{recv}(m)$$
  - **HB3 (Transitivität)**
    Wenn Event e vor Event e' passiert ist, und Event e' vor Event e'' passiert ist, dann auch ist Event e vor Event e'' passiert.
    $$e \rightarrow e' \wedge e' \rightarrow e'' \Rightarrow e \rightarrow e''$$
  - **Nebenläufigkeit**
    Wenn weder festgestellt werden kann, dass Event e vor Event e' passiert ist noch,

dass Event e' vor Event e passiert ist, dann gelten e und e' als nebenläufige Events.

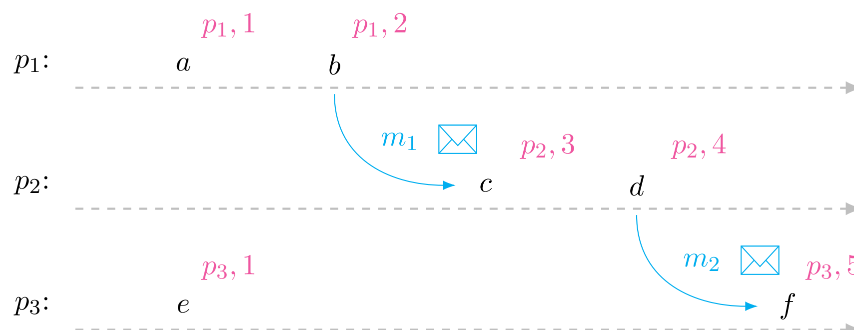
$$e \not\to e' \land e' \not\to e \Rightarrow e \parallel e'$$

- Lamport Uhren
  - Definitionen
    * Logische Uhrzeit von Prozess $i = L_i$
    * $L_i(e) =$ Zeitstempel von Event e auf von p_i
    * $L(e) =$ Zeitstempel von Event e
  - Algorithmus / Regeln
    * **LC1**
      Erhöhe L_i bevor jedem Event das auf Prozess p_i passiert
    * **LC2(a)**
      p_i versendet Nachrichten m mit Zeitstempel $t = L_i$
    * **LC2(b)**
      Wenn p_i eine Nachrichten m mit Zeitstempel t empfängt, dann setzt p_i L_i auf $\max(L_i, t)$
  - Eigenschaften
    * Wenn Event e vor Event e' passiert ist, dann spiegelt sich dies in der Lamport Zeit wieder.

      $$e \to e' \Rightarrow L(e) < L(e')$$

    * Dies gilt aber **nicht** in die andere Richtung!

      $$L(e) < L(e') \not\Rightarrow e \to e'$$



- Vektor Uhren
  - Eigenschaften Für alle $j \in \{1, \ldots, N\}$ gilt:
    * Zwei Vektoren gelten genau dann als gleich, wenn alle Einträge gleich sind.

      $$V = V' \Leftrightarrow V[j] = V'[j]$$

    * Ein Vektor V gilt genau dann als $\leq$, wenn alle Einträge $\leq$ sind.

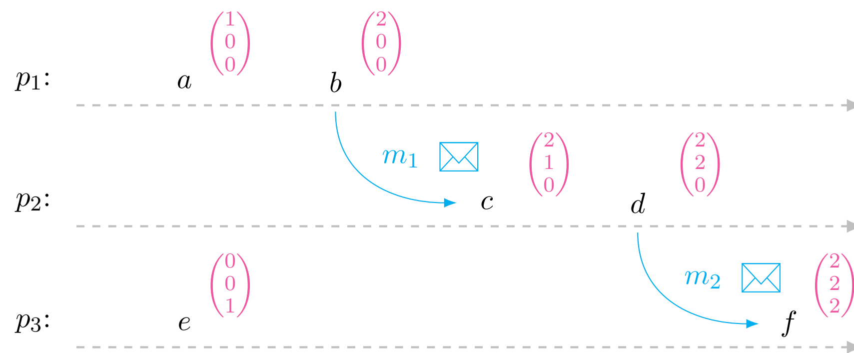
      $$V \leq V' \Leftrightarrow V[j] \leq V'[j]$$

* Ein Vektor V gilt genau dann als kleiner, wenn alle Einträge kleiner sind.

$$V < V' \Leftrightarrow V[j] < V'[j]$$

Das heißt: Falls $V(a) < V(f)$ gilt, dann ist a vor f passiert $(a \rightarrow f)$.

Falls $V(a) \nleq V(b)$ und $V(a) \ngeq V(b)$ gilt, dann gilt $a \parallel b$, a und b werden also als nebenläufige Events angesehen.



Fragen Kapitel 2.3

Frage 1

Wann ist eine Hardware-Uhr korrekt?

Frage 2

Was ist der Unterschied zwischen externer und interner Synchronisation?

Frage 3

Client C will seine Uhr mittels Cristians Algorithmus mit Zeit-Server S synchronisieren. Wie macht er das?

Frage 4

Client C will seine Uhr mittels Cristians Algorithmus mit Zeit-Server S synchronisieren. C fragt S nach seiner Uhr zum Zeitpunkt $t_1 = 2$. C erhält die Antwort nach seiner Uhr zum Zeitpunkt $t_2 = 5$. In der Antwort ist $t_3 = 7$ enthalten. Auf welche Uhrzeit setzt C seine Uhr nachdem er die Antwort bekommen hat?

Frage 5

Was ist eine Root-Uhr und eine Stratum-1-Uhr im Bezug auf NTP?

Frage 6

Auf vier Prozessen passieren die folgenden Events:

p_1 :	a	b	$\texttt{recv}(m_1)$	c	$\texttt{send}(m_3)$
p_2 :	$\texttt{recv}(m_2)$	$\texttt{send}(m_1)$	d	$\texttt{send}(m_4)$	
p_3 :	$\texttt{send}(m_2)$	e	f	$\texttt{recv}(m_3)$	
p_4 :	g	h	$\texttt{recv}(m_4)$	i	

Skizziere den Ablauf und trage die Zeitstempel in die Skizze(n) ein die der Lamport Algorithmus bzw. der Vector Uhren Algorithmus den Events zuweisen würde.

Antworten Kapitel 2.3

Antwort 1

Wann ist eine Hardware-Uhr korrekt?

Eine Hardware-Uhr ist korrekt gdw. ihre Drift Rate nach oben und unten begrenzbar ist.

Antwort 2

Was ist der Unterschied zwischen externer und interner Synchronisation?

Bei der externen Synchronisation synchronisieren sich lokale Uhren mit ≥ 1 authoritären Uhr, bei der lokalen Synchronisation hingegen synchronisieren sich mehrere lokale Uhren untereinander, sodass sie idealerweise die gleiche Uhrzeit anzeigen.

Antwort 3

Client C will seine Uhr mittels Cristians Algorithmus mit Zeit-Server S synchronisieren. Wie macht er das?

1. Client C fragt Zeit-Server S nach der Uhrzeit
2. S sendet C seine Uhrzeit t_S
3. C berechnet die round-trip-time T_{round}
4. C setzt seine lokale Uhrzeit auf $t_S + \frac{1}{2}T_{\text{round}}$

Antwort 4

Client C will seine Uhr mittels Cristians Algorithmus mit Zeit-Server S synchronisieren. C fragt S nach seiner Uhr zum Zeitpunkt $t_1 = 2$. C erhält die Antwort nach seiner Uhr zum Zeitpunkt $t_2 = 5$. In der Antwort ist $t_3 = 7$ enthalten. Auf welche Uhrzeit setzt C seine Uhr nachdem er die Antwort bekommen hat?

C setzt seine Uhrzeit auf

$$\begin{aligned} T &= t_3 + \frac{1}{2}T_{\text{round}} \\ &= t_3 + \frac{1}{2} \cdot (t_2 - t_1) \\ &= 7 - \frac{1}{2} \cdot (5 - 2) \\ &= 8.5 \end{aligned}$$

Entsprechend stellt er seine Uhrzeit auf 8.5.

Antwort 5

Was ist eine Root-Uhr und eine Stratum-1-Uhr im Bezug auf NTP?

Eine Root-Uhr ist direkt verbunden mit einer Radio Clock die ihr UTC gibt.

Eine Stratum-1-Uhr synchronisiert sich mit einer Root-Uhr und bietet z.B. Stratum-2-Uhren einen Time-Service an.

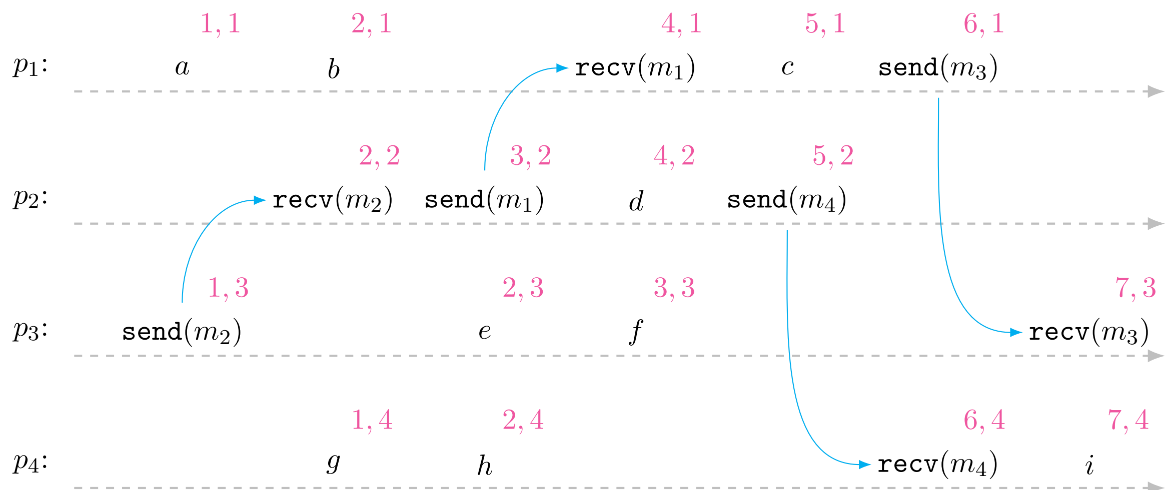
Antwort 6

Auf vier Prozessen passieren die folgenden Events:

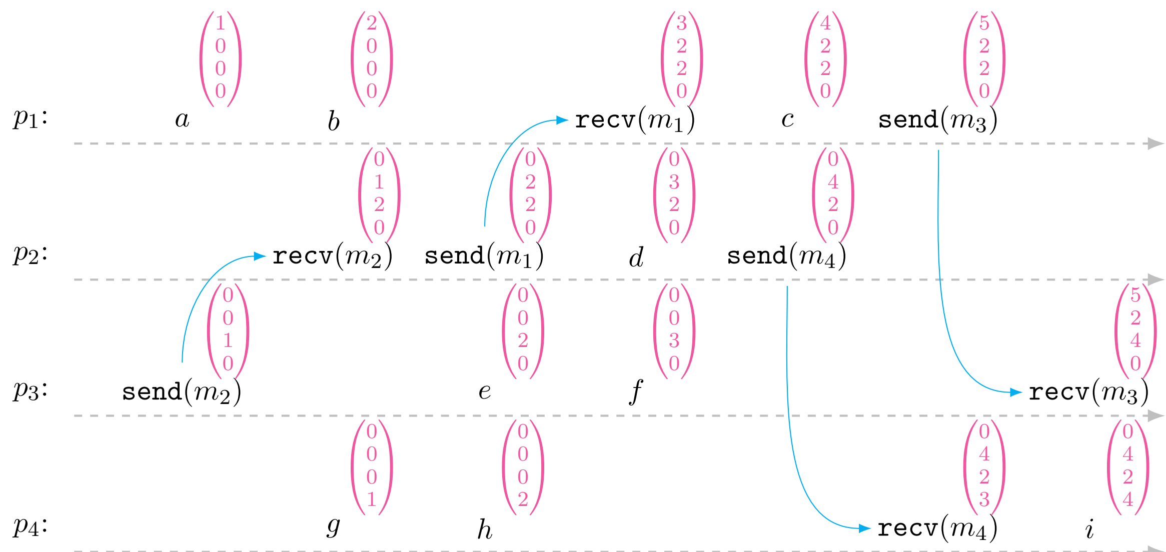
p_1 :	a	b	*recv*(m_1)	c	*send*(m_3)
p_2 :	*recv*(m_2)	*send*(m_1)	d	*send*(m_4)	
p_3 :	*send*(m_2)	e	f	*recv*(m_3)	
p_4 :	g	h	*recv*(m_4)	i	

Skizziere den Ablauf und trage die Zeitstempel in die Skizze(n) ein die der Lamport Algorithmus bzw. der Vector Uhren Algorithmus den Events zuweisen würde.

Lamport



Vector



Kapitel 2.4. Replication

(a) Motivation

Mithilfe von Replikation können wir erhöhte Performance (z.B. Caches), Verfügbarkeit und Fehler-Toleranz (z.B. mehrere Server mit gleichem Interface) erreichen

(b) Definitionen

- **Logisches Object (logical object)**
  Menge an physischen Kopien (Replicas)
- **Replica Manager (RM)**
  Verwaltet ein Replica, wendet rücksetzbare Operationen auf sein Replica an, sodass der Zustand zurückgesetzt werden kann, falls eine Operation fehlschlägt
- **Service**
  Menge an RM's die Zugang zum selben logischen Object zur Verfügung stellen. Mögliche Operationen:
  - Read-only-request ⇒ kein Update, nur lesen
  - Update-request ⇒ Update und lesen
- **Front End (FE)**
  Kümmert sich um Anfragen vom Client, ermöglicht Replication Transparency und leitet Anfragen zu den RM's weiter

(c) Consistency-Scalability Tradeoff

Definition

Ein Consistency Model ist ein Vertrag zwischen den Prozessen / Clients und dem verteilen Datenspeicher

Data-centric Consistency Model

Fokus liegt auf der Konsistenz von Lese- und Schreibzugriffen

Das Model schränkt ein welche Werte Lesezugriffe ausgeben können.

- **Strong model**
  Viele Einschränkungen, einfach zu implementieren, schlechte Performance
- **Weak model**
  Wenige Einschränkungen, schwer zu implementieren, gute Performance

Client-centric Consistency Model

Fokus liegt auf dem Datenspeicher, auf dem keine gleichzeitigen Updates geschehen dürfen, oder Inkonsistenzen die dadurch entstehen einfach aufgelöst werden können z.B. DNS (Konsequenz $\rightarrow$ schwache Konsistenz).

- Eventual Consistency: Keine updates passieren $\Rightarrow$ Replicas werden mit der Zeit untereinander konsistent
- Falls eine Client den RM wechselt, dann sollte er die Konsequenz des letzten Schreibzugriffs sehen

(d) Single Leader Replication

Beschreibung: Master-Slave Verfahren um Fehler-Toleranz zu erreichen

- Es gibt einen Primary/Leader-RM und mehrere Backup/Follower-RMs
- FE spricht mit der Leader-RM; Leader-RM propagiert Updates zu den Follower-RMs
- Leader-RM crasht $\Rightarrow$ Eine Follower-RM wird zur Leader-RM aufgestuft

(e) Leaderless Replication

Beschreibung: Gleichgestellte Replicas organisieren sich als eine Gruppe, in der jede Replica die gleiche Rolle einnimmt

(f) Single Leader Replication vs. Leaderless Replication

	Abstrakt	Single Leader Replication	Leaderless Replication
1. Anfrage	FE leitet Anfrage weiter	FE $\xrightarrow{\text{ID, Anfrage}}$ primary RM	FE $\xrightarrow{\text{ID, Anfrage}}$ alle RM's
2. Koordination	RM's koordinieren sich (Ordnung, Konsistenz)	ID bekannt ⇒ Sende Ergebnis nochmal, sonst goto 3	Group Management System sendet die Anfrage zu jedem korrekten RM in der selben Reihenfolge
3. Ausführung	RM's führen die Anfrage aus	Primary RM führt Anfrage aus und speichert das Ergebnis	Alle RM's führen die Anfrage aus, korrekte RM's haben gleiches Ergebnis
4. Einigung	RM's finden eine Einigung über das finale Ergebnis	Update-request ⇒ primary RM sendet finalen State an andere RM's, diese bestätigen die Nachricht	
5. Antwort	≥ 1 RM's $\xrightarrow{\text{Ergebnis}}$ FE	Primary RM $\xrightarrow{\text{Ergebnis}}$ FE	Alle RM's $\xrightarrow{\text{Ergebnis}}$ FE, FE aggregiert, selektiert, ...

Korrektheit von Fehler-toleranten Systemen mit Replikation

- Reagiert auch wenn ein Fehler auftritt
- Kein Unterschied zu einem Service mit nur einer korrekten RM feststellbar (→ Konsistenz zwischen den Replicas)

(g) Netzwerk Partitionierung

Definition: RM's werden in verschiedene abgeschottete Gruppen unterteilt.

Wie damit umgehen?

- **Optimistisch**
  Nachdem die Partitionierung aufgelöst wurde werden Updates validiert und Inkonsistenzen repariert
- **Pessimistisch**
  Verfügbarkeit begrenzen z.B. kleinere Gruppe ausschalten

(h) CAP-Theorem

Beschreibung: Verteilte Systeme können gleichzeitig nur zwei der folgenden drei Eigenschaften vollständig erfüllen.

- **Availability**
  Fähigkeit Anfragen zu beantworten
- **Partition Tolerance**
  System läuft weiter auch wenn Knoten abstürzen oder Kommunikations-Kanäle fehlschlagen

- **Consistency**
  Update von einem Datenelement $\Rightarrow$ Alle Replicas werden auch geupdated, sodass Lesezugriffe immer die aktuellste Version des Datenelements zurückgeben

  Consistency-Typen

  - **Strong Consistency**
    A geupdated $\Rightarrow$ Lesezugriff auf A gibt für jeden Leser den neuen Wert zurück
  - **Weak Consistency**
    Inconsistency-Window existiert, es gibt keine Konsistenz-Garantie innerhalb des Inconsistency-Windows
  - **Eventual Consistency (Spezialfall von Weak Consistency)**
    A geupdated $\Rightarrow$ Falls keine Fehler passieren wird irgendwann jeder das geupdatete A sehen. Das Inconsistency-Window ist also bestimmbar und begrenzt

    Spezialfälle von Eventual Consistency:

    * **Causal Consistency**
      $A \xrightarrow{\text{update(A)}} B$, und **nicht** $A \xrightarrow{\text{update(A)}} C \Rightarrow$ Wenn B auf A zugreift erhält B den neuen Wert, wenn allerdings C auf A zugreift kann C auch den alten Wert bekommen
    * **Read-Your-Writes Consistency**
      Die Werte der Variablen, die man selbst verändert hat, werden nie älter sein, als zu dem Zeitpunkt zu dem man sie zuletzt verändert hat. Man sieht also mindestens die Konsequenzen der letzten eigenen Veränderung.
    * **Session Consistency**
      Konsistenz-Garantie für eine gegebene Session
    * **Monotonic-Read Consistency**
      Man liest nie einen älteren Wert einer Variable, als den neusten Wert den man bisher gesehen hat.
    * **Monotonic Write Consistency**
      Lesezugriffe werden in der selben Ordnung durchgeführt wie der Client sie angefordert hat.
  - **Server-Side Consistency**
    * **Definitionen**
      - $N :=$ Anzahl Knoten
      - $W :=$ Anzahl Replicas die ein Update bestätigen müssen
      - $R :=$ Anzahl Replicas die einen Lesezugriff bestätigen müssen
    * **Eigenschaften**
      - $W > N/2$ ($w \parallel w$ Inkonsistenzen), $W + R > N$ ($r \parallel w$ Inkonsistenzen) $\Rightarrow$ Keine $w \parallel w$ oder $r \parallel w$ Inkonsistenzen möglich
      - $R = 1,\ W = N \Rightarrow$ Schnelles Lesen, langsames Schreiben

- $R = N,\ W = 1 \Rightarrow$ Langsames Lesen, schnelles Schreiben
- $W + R \leq N$ (oft auch $R = 1$) $\Rightarrow$ Weak/Eventual Consistency
- $W \gg N/2,\ R \gg N/2 \Rightarrow$ Fehler-Toleranz

(i) CAP-Theorem Beispiele

- **CA**

  Relationale Datenbank Management Systeme (RDBMS)

  Partition Toleranz kann nicht erreicht werden, da nicht zwischen langsamen Nachrichten und einem kaputten Kommunikations-Kanal unterschieden werden kann.

  - RDBMS wartet (wegen Partition Tolerance) $\Rightarrow$ Falls Kanal kaputt, dann wird Availability verletzt
  - RDBMS beurteilt Kanal als kaputt $\Rightarrow$ Falls Nachricht langsam, dann wird Consistency verletzt, Nachricht könnte ja ein Update sein

- **CP**

  Banking

- **AP**

  DNS

(j) Stickyness-Problem

Umso mehr wir den Client an den Server binden, desto mehr Konsistenz können wir garanieren. Die Konsistenz kommt also auf Kosten von der Performance, da wir weniger Load-Balancing betreiben können.

(k) Heuristiken für Clouds

- Availability ist wichtiger als Consistency
- Benutze *Quorum Consensus* (das R,W,N Ding unter *Server-Side Consistency*) um Parition Tolerance zu gewährleisten
- Eventual Consistency ist gut genug

Fragen Kapitel 2.4

Frage 1

Wie kann ich die Leistung einer Datenbank in verteilten Systemen verbessern?

Frage 2

Was bedeutet ROWA?

Frage 3

Was macht ein Replica Manager (RM)?

Frage 4

Was ist ein Consistency Model? Was ist der Unterschied zwischen einem *Data Centric Consistency Model* und einem *Client-centric Consistency Model*?

Frage 5

Was ist der Unterschied zwischen *Single Leader Replication* und *Leaderless Replication*?

Frage 6

In einem Verteilten System das *Leaderless Replication* benutzt, welche Komponente bestimmt das finale Ergebnis, also hat „das letzte Wort“?

Frage 7

In einem Verteilten System gibt es 10 Replicas, 5 Replicas müssen Schreibzugriffe bestätigen, 6 Replicas müssen Lesezugriffe bestätigen. Kann damit paralleles Lesen und Schreiben bzw. paralleles Schreiben verhindert werden?

Frage 8

Welche Eigenschaften des CAP-Theorem erfüllen Banking-Anwendungen? Warum kann so eine Anwendung nicht alle Eigenschaften gleichzeitig erfüllen?

Frage 9

Warum binden wir einen Client nicht einfach an genau einen Server um Inkonsistenzen zu minimieren?

Antworten Kapitel 2.4

Antwort 1

Wie kann ich die Leistung einer Datenbank in verteilten Systemen verbessern?

Durch Partitionierung (Aufteilung der Datenbank in mehrere Teile) und Replizierung (mehrere Kopien der Datenbank).

Antwort 2

Was bedeutet ROWA?

Read Once Write All – Lese irgendwo aber schreibe überall – siehe auch Consistency.

Antwort 3

Was macht ein Replica Manager (RM)?

Ein Replica Manager verwaltet ein Replica und wendet rücksetzbare Operationen auf sein Replica an, sodass der Zustand zurückgesetzt werden kann, falls eine Operation fehlschlägt.

Antwort 4

Was ist ein Consistency Model? Was ist der Unterschied zwischen einem Data Centric Consistency Model und einem Client-centric Consistency Model?

Ein Consistency Model ist ein Vertrag zwischen den Prozessen / Clients und dem verteilen Datenspeicher. Bei einem Data-centric Consistency Model liegt der Fokus auf der Konsistenz von Lese- und Schreibzugriffen.
Der Fokus bei einem Client-centric Consistency Model liegt hingegen auf dem Datenspeicher auf dem keine gleichzeitigen Updates geschehen dürfen bzw. die Inkonsistenzen die dadurch entstehen einfach aufgelöst werden können.

Antwort 5

Was ist der Unterschied zwischen Single Leader Replication und Leaderless Replication?

Bei der Single Leader Replication gibt es einen primary RM der Anfragen vom FE erhält und diese Anfragen dann ausführt und den veränderten State (bei einem Schreib-Zugriff) an die anderen RM's weitergibt.
Die Leaderless Replication ist dezentraler. Bei ihr gibt es keinen primary RM, stattdessen leitet das FE alle Anfragen an alle RM's weiter, die dann das Ergebnis autonom an das FE weitergeben.

Antwort 6

In einem Verteilten System das Leaderless Replication benutzt, welche Komponente bestimmt das finale Ergebnis, also hat „das letzte Wort“?

Das Front-End (FE), denn es aggregiert, selektiert, ... die Ergebnisse die die RM's ihr am Ende liefern.

Antwort 7

In einem Verteilten System gibt es 10 Replicas, 5 Replicas müssen Schreibzugriffe bestätigen, 6 Replicas müssen Lesezugriffe bestätigen. Kann damit paralleles Lesen und Schreiben bzw. paralleles Schreiben verhindert werden?

Paralleles Lesen und Schreiben kann verhindert werden, da um parallel Lesen und Schreiben zu können 11 Stimmen gebraucht werden, es aber nur 10 Replicas gibt.

Paralleles Schreiben kann nicht verhindert werden, da nur 5 Replicas einen Schreibzugriff bestätigen müssen, es kann also eine Hälfte den Zugriff von Person A bestätigen, und die andere Hälfte den Zugriff von Person B bestätigen, sodass beide gleichzeitig eine Variable updaten.

Antwort 8

Welche Eigenschaften des CAP-Theorem erfüllen Banking-Anwendungen? Warum kann so eine Anwendung nicht alle Eigenschaften gleichzeitig erfüllen?

Banking Anwendungen erfüllen Consistency und Partition Tolerance (CP). Damit eine Banking Anwendung auch Availability erfüllt müsste sie alle Anfragen direkt beantworten können.

Wenn eine Netzwerkstörung auftritt, dann sollte ein Geldautomat der darunter leider allerdings lieber garnicht reagieren als eine Überweisung anzunehmen die entweder falsch oder gar nicht ausgeführt wird. Er kann Anfragen also nicht immer direkt beantworten.

Antwort 9

Warum binden wir einen Client nicht einfach an genau einen Server um Inkonsistenzen zu minimieren?

Die Konsistenz kommt auf Kosten der Performance des Verteilten Systems. Wenn wir einen Client vollständig an einen einzigen Server binden, dann können wir kein / weniger Load-Balancing betreiben, was Performance-Einbußen bedeutet.

Kapitel 2.5. Consensus

(a) Coordination and Agreement

- Verschiedene Prozesse müssen sich einig darüber sein, welche Aktionen wann ausgeführt werden sollen, um Inkonsistenzen zu vermeiden
- Übereinstimmung sollte ohne festen Master erreicht werden, da sonst ein weiterer Single Point of Failure eingeführt wird und die Fault Tolerance verringert wird.

(b) Distributed Mutual Exclusion

Ansprüche:

- **ME1 (safety)**
  Ein kritischer Bereich kann nicht von mehreren Prozessen gleichzeitig betreten werden
- **ME2 (liveliness)**
  Deadlock / Starvation ist nicht möglich
- **ME3 (ordering)**
  Der Prozess, der den Zugang zum kritischen Bereich zuerst beantragt hat, erhält ihn zuerst

Algorithmus von Ricart und Agrawala (erfüllt ME1 bis ME3)

- Jeder Prozess hat eine Lamport Clock für eine totale Ordnung (via process ID's)
- Falls ein Prozess einen CS (critical section) betreten möchte, multicastet er eine Anfrage an alle anderen Prozesse
- Der Prozess darf den CS betreten, wenn er Antworten von allen Prozessen erhält
- Jeder Prozess p_i mit ID i befindet sich in einem von 3 Zuständen:
  - **Released**
    p_i hat kein Interesse an dem CS und schickt im Falle einer Anfrage direkt eine Antwort
  - **Held**
    p_i befindet sich im CS und antwortet im Falle einer Anfrage, nachdem er den CS verlässt
  - **Wanted**
    p_i möchte in den CS und antwortet im Falle einer Anfrage von einem p_j, wenn der Zeitstempel von p_i höher ist als der von p_j

Beispiel zum Algorithmus:

p_3 ist nicht am CS interessiert, p_1 und p_2 schicken simultan eine Anfrage

1. p_3 antwortet sofort
2. p_2 vergleicht Zeitstempel und antwortet nicht ($34 \not> 41$)
3. p_1 vergleicht Zeitstempel und antwortet ($41 > 34$)

4. p_2 betritt den Bereich (antwortet p_1, sobald er den CS verlässt)

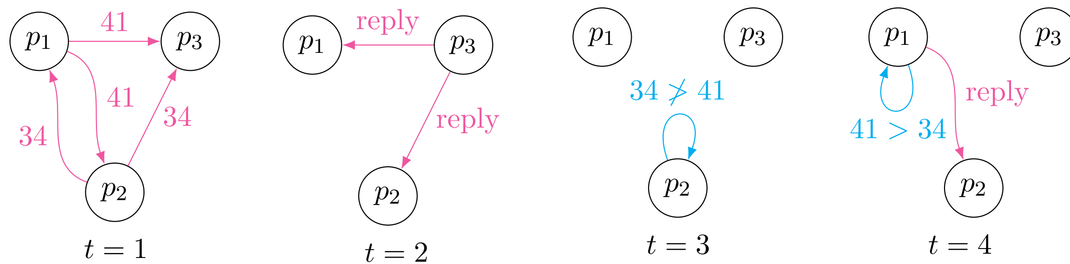


Abbildung 2.2: Beispiel Ricart und Agrawala Algorithmus

(c) Election (Dynamische Wahl des Masters)

- Jeder Prozess kann die Wahl starten
- Ein Prozess ist entweder ein Participant oder nicht
- Der Master wird anhand einzigartiger ID's mit totaler Ordnung gewählt
- Jeder Prozess p_i hält eine Variable e_i, welche die ID des momentanen Masters speichert
- Wenn ein Prozess p_i zum ersten Mal an einer Wahl teilnimmt, wird $e_i = \perp$ gesetzt
- Während einer Wahl müssen zwei Bedingungen gelten:
  - **E1 (safety)**
    Ein Prozess p_i hat $e_i = \perp$ oder $e_i = p_j$ gesetzt, wobei p_j der Prozess mit der höchsten ID ist, der nicht gecrasht ist
  - **E2 (liveliness)**
    Alle Prozesse partizipieren und haben am Ende einer Wahl $e_i \neq \perp$ gesetzt oder sind gecrasht

(d) Bully Algorithmus zur Wahl des Masters (erfüllt E1 und E2)

- Jeder Prozess weiß, welche Prozesse höhere ID's haben und kann mit diesen zuverlässig kommunizieren
- Es gibt 3 Message-Typen:
  - **Election Message**
    Starte eine Wahl
  - **Answer Message**
    Anwort auf eine Election Message
  - **Coordinator Message**
    Gib ID des gewählten Prozesses bekannt

Ablauf, wenn p_i eine Wahl startet:

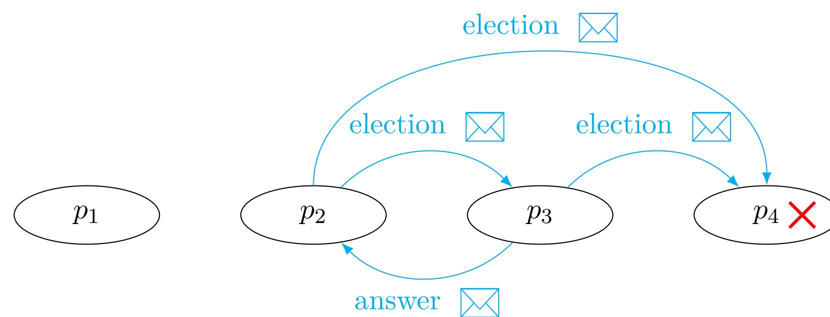
- Wenn p_i die höchste ID hat, schicke eine Coordinator Message mit eigener ID
- Sonst schicke eine Election Message an alle Prozesse mit höherer ID und warte eine Zeitdauer T auf eine Answer Message

- Wenn innerhalb von T keine Answer Message kommt, schicke Coordinator Message mit eigener ID
- Sonst warte weiteres T auf Coordinator Message und starte eine neue Wahl falls keine eintrifft

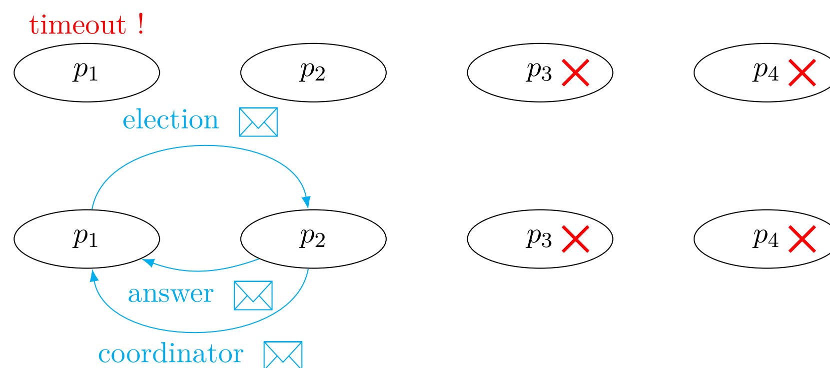
- Wenn ein Prozess eine Coordinator Message erhält, setzt er e_i auf die entsprechende ID
- Wenn ein Prozess eine Election Message erhält, schickt er eine Answer Message zurück und startet eine neue Wahl (es sei denn, er hat bereits eine gestartet)

Bully Algorithmus Beispiel:

- Beispiel mit Prozess p_1, p_2, p_3 und p_4. p_4 ist der Koordinator und ist nicht mehr erreichbar. p_1 erkennt den Fehler.
  - p_2 und p_3 starten eine Wahl
  - p_3 meldet p_2 dass er selbst noch aktiv ist.
  - p_3 bekommt keine Rückmeldung von p_4 und entscheidet selbst der Koordinator zu werden.



  - P_3 ist noch vor der Koordinator-Antworten ebenfalls nicht mehr erreichbar.
  - da keine Koordinator-Nachricht von p_3 bei p_1 eintrifft erkennt p_1 den Timeout.
  - p_1 startet eine neue Wahl bei der p_2 gewählt wird



Grundlegende Annahmen: Synchronisiertes System + Fehlererkennung

(e) Consensus

Wie kann erreicht werden, dass sich Prozesse über bestimmte Werte einig sind?

- Jeder Prozess beginnt im „nicht entschieden“ Zustand und schlägt einen Wert v_i aus einer Wertemenge D vor.
- Prozesse kommunizieren, setzen Entscheidungsvariable d_i und wechseln in den „entschieden“ Zustand
- Folgende Bedingungen sollten gelten:
  - **Termination**
    Irgendwann setzt jeder Prozess ein d_i
  - **Agreement**
    Wenn p_i und p_j korrekt entscheiden, gilt $d_i = d_j$
  - **Integrity (validity)**
    Wenn alle korrekten Prozesse dasselbe d_i wählen, wählt jeder korrekte Prozess im „entschieden“-Zustand dieses d_i
- Consensus kann niemals sicher in asynchronen Systemen erreicht werden. (FLP impossibility)
- Deshalb in der Realität:
  - Failure masking
  - Failure Detection
  - Relaxiere System Model zu Partial Synchronous Systems
- Ein t-resilientes Consensus Protocol ist ein Protokoll, dass t fehlerhafte Knoten toleriert

Fehler Art	Synchron	Asynchron	Partiell Synchron
Fail-Stop	t	∞	$2t+1$
Omission	t	∞	$2t+1$
Byzantinisch (Authentifiziert)	t	∞	$2t+1$
Byzantinisch	$3t+1$	∞	$3t+1$

Tabelle 2.1: Kleinste Anzahl an Prozessen N, für welche ein t-robuster Algorithmus existiert

(f) End-To-End Arguments

- Sicherheitsmaßnahmen auf low-level-Ebenen werden ab einem gewissen Grad oft redundant, unzureichend und ineffizient, da die Fehler schnell byzantinisch werden
- Beispiel: Sicherer File Transfer; Korrektheit des Transfers sollte besser auf höherer Ebene geschehen (z.B. durch Checksums) $\rightsquigarrow$ **End-To-End-Principle**

(g) Avoiding Coordination

- Koordination ist teuer und beeinträchtigt die Unabhängigkeit der Knoten

- Koordination kann vermieden werden, indem die Korrektheit nicht anhand von low-level-Operationen (read, write etc.) gemessen wird, sondern anhand von Invarianten auf höheren Ebenen
- Koordination ist nicht nötig, wenn die spezifizierten Invarianten den Invariance Confluence Test (ICT) bestehen (z.B.: die ID ist positiv, die Checksumme stimmt etc.)

Fragen Kapitel 2.5

Frage 1

Mit welchen Problemen bei einem Konsens muss gerechnet werden?

Frage 2

Welche Aspekte müssen bei Koordination und Abstimmung in einem verteilten System beachtet werden?

Frage 3

Welches ist die einfachste Methode ein VS zu Koordinieren? Welche Nachteile entstehen aus dieser Methode?

Frage 4

Welche Alternativen zur Koordinierung mit festem Master gibt es innerhalb von Verteilten Systemen?

Frage 5

Zu welchem Problem kann es bei einer automatisierten Wahl kommen wenn das Netz partitioniert wird?

Frage 6

Erläutere den Bully-Algorithmus.

Frage 7

Warum kann beim Byzantine-Generals-Problem keine Lösung erzielt werden, wenn $N = 3$ und $f = 1$ gilt, wobei N die Anzahl der Prozesse und f die Anzahl der fehlerhaften Prozesse ist?

Frage 8

Welches Problem gibt es in asynchronen Systemen in Bezug auf Timeouts?

Antworten Kapitel 2.5

Antwort 1

Mit welchen Problemen bei einem Konsens muss gerechnet werden?

- Es kann Unstimmigkeiten bzgl. eines Werts einer Variable geben.
- Prozesse könnten während der Konsensfindungs-Phase crashen.
- Falls ein Master Prozess benutzt wird, kann er crashen und wenn wir keinen guten Mechanismus haben um einen neuen Master Prozess zu wählen gibt es keinen Coordinator mehr.
- Bei byzantinischen Fehlern kann bei $N \leq 3f$, wobei f die Anzahl an fehlerhaften Prozessen und N die Anzahl aller Prozesse, im Allgemeinen kein Konsens erreicht werden.
- FLP Impossibility: Kein Algorithmus kann in asynchronen Systemen eine Konsens-Findung garantieren weil eine Unterscheidung zwischen einem gecrashten Prozess und einem langsamen Prozess generell nicht möglich ist.

Antwort 2

Welche Aspekte müssen bei Koordination und Abstimmung in einem verteilten System beachtet werden?

Siehe Ansprüche

Antwort 3

Welches ist die einfachste Methode ein VS zu Koordinieren? Welche Nachteile entstehen aus dieser Methode?

Koordinierung durch einen zentralen Master. Dies kann dann aber zu einem Single-Point-Of-Failure (SPOF) oder einem Bottleneck führen

Antwort 4

Welche Alternativen zur Koordinierung mit festem Master gibt es innerhalb von Verteilten Systemen?

Es gibt sowohl den Algorithmus von Ricart und Agrawala sowie den Bully Algorithmus.

Antwort 5

Zu welchem Problem kann es bei einer automatisierten Wahl kommen wenn das Netz partitioniert wird?

Die Partitionen wählen jeweils einen eigenen Master. Bei Einigung führt das dann zu Inkonsistenzen.

Antwort 6

Erläutere den Bully-Algorithmus.

Siehe Bully Algorithmus

Antwort 7

Warum kann beim Byzantine-Generals-Problem keine Lösung erzielt werden, wenn $N = 3$ und $f = 1$ gilt, wobei N die Anzahl der Prozesse und f die Anzahl der fehlerhaften Prozesse ist?

Byzantinische Fehlertoleranz ist erfüllt falls sich alle korrekten Generäle auf eine Strategie einigen. Ein Prozess muss wegen der *Integrity / Safety* Eigenschaft auf die Befehle des Kommandanten (Coordinator) hören und seine Variable entsprechend setzen.

Falls der Kommandant allerdings byzantinische Fehler aufweist und den Generälen verschiede Befehle gibt, so werden die Generäle verschiedene Dinge tun (ihre Variablen verschieden belegen).

Dadurch wird aber aber die *Agreement/Safety* verletzt, denn die Generäle haben sich nicht auf eine Aktion geeinigt (die Variable gleich belegt).

Das grundlegende Problem ist, dass es für die Generäle keine Möglichkeit gibt zu entscheiden ob der Kommandant fehlerhaft ist oder nicht.

Antwort 8

Welches Problem gibt es in asynchronen Systemen in Bezug auf Timeouts?

Wir können nie mit vollständiger Sicherheit sagen ob ein Prozess mit dem wir kommunizieren wollen tatsächlich gecrasht ist falls ein Timeout abläuft oder ob die Antworten von dem Prozess lediglich verspätet ankommt.

Kapitel 3

Architekturmodelle

Kapitel 3.1.

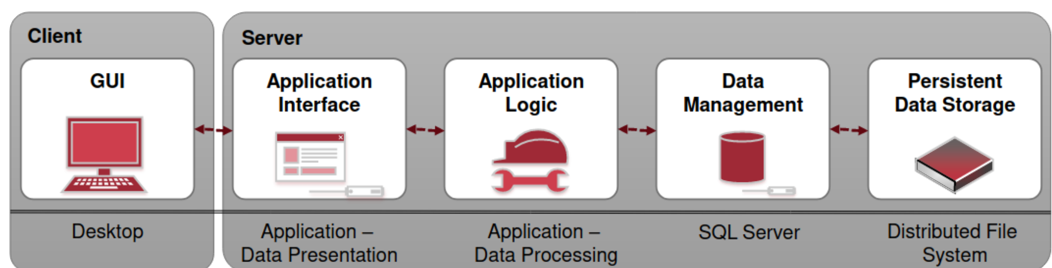
(a) Client/Server System

- Server ist das Bottleneck und Single Point of Failure
- **Verbesserungen**
  - Parallelisierung innerhalb des Servers
  - Erweiterte Client-Server Systeme (Partitionierung, Replikation, Kooperation zwischen Servern)
- **Basiskomponenten eines Client-Server-Systems**
  - GUI
  - Application Interface
  - Application Logic
  - Data Management
  - Persistent Data Storage

(b) Two and Three Tier Models

- Clients lassen sich in folgende Klassen unterteilen:
  - **Zero Client**

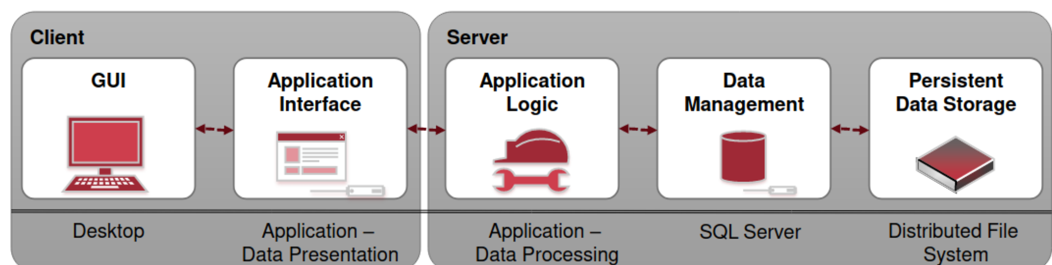
    Der Client steuert nur die Input Devices



Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 3.1 Slide 11

  - **Thin Client**

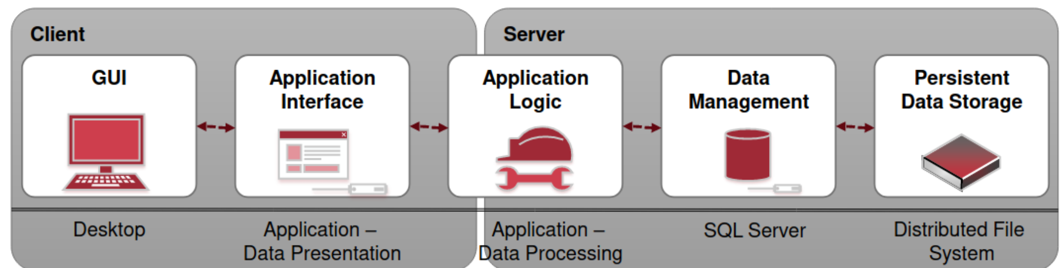
    Der Client ist zusätzlich für Präsentation und Ergebnis zuständig



Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 3.1 Slide 12

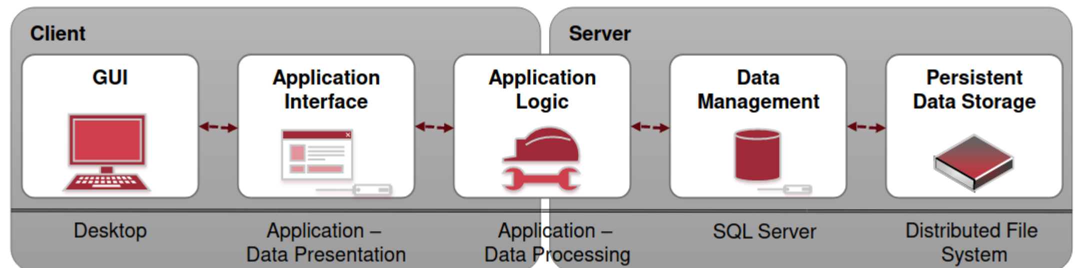
  - **Applet Client**

    Einige Prozesse des Servers werden lokal ausgeführt



Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 3.1 Slide 13

- **Fat Client**
  Client und Server operieren auf der Application-Logic Ebene



Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 3.1 Slide 14

- Three Tier Model besteht aus
  - Clients (presentation layer)
  - Application und Processing Server (application/processing layer)
  - Database Server (storage of information)

(c) Extended Client Server Systems

Ein Client kann bei Systemen mit mehreren Servern auf verschiedene Weisen an die Daten kommen:

- **rekursiv**
  Client fragt einen Server, dieser fragt rekursiv andere Server...bis der Zielserver erreicht ist, der Client erhält die Antwort vom befragten Server
- **transitiv**
  Client bekommt die Antwort direkt vom Zielserver

Server können klassifiziert werden als:

- **Proxy**
  Schnittstelle zwischen Client und anderen Servern, die die Anfragen cached, geeignet für sich wiederholende Anfragen
- **Broker**
  Teilt Clients einem Server entsprechend der geforderten Dienstleistung zu
  - Vorteil: Location/Replication transparency
  - Nachteil: Single-Point-Of-Failure

- **Trader**
  Wählt den passenden Server aus, falls mehrere Server denselben Service in unterschiedlicher Qualität anbieten
- **Balancer**
  Verteilt Workload gleichmäßig auf alle Server
- **Agent**
  Splittet komplexe Anfragen auf, sendet die simpleren Anfragen an andere Server und kombiniert die Antworten zu einer Antwort

(d) Interaction (Failure) Semantics

Mögliche Fehler bei Client-Server-Interaktion

- Crashes auf einer der beiden Seiten
- Nachrichtenverlust/Verzögerung
- Nachrichten werden nicht gesendet
- Verschiedene Nachrichten werden geschickt (byzantinisch)

Wann muss ein Server reagieren?

- Stateful Server: Reaktion notwendig
- Stateless Server: Reaktion nicht notwendig

(e) Klassifikation von Stateless Servern

- **Stateful Communication**
  Server speichert für jeden Client den Status
  - Session-basierte Kommunikation
  - Server kann Client-Verhalten oft voraussehen, was zur Steigerung der Performance führt
  - Bei einem System Crash muss der Server mehr Daten wiederherstellen
- **Stateless Communication**
  Client speichert den Status
  - Communication Overhead, da mehr Infos geschickt werden müssen
  - Höhere Fault Tolerance: Clients können Fehler durch Timeouts erkennen und die Anfrage erneut schicken

(f) Mögliche Priorisierungen bei der Fehlervermeidung

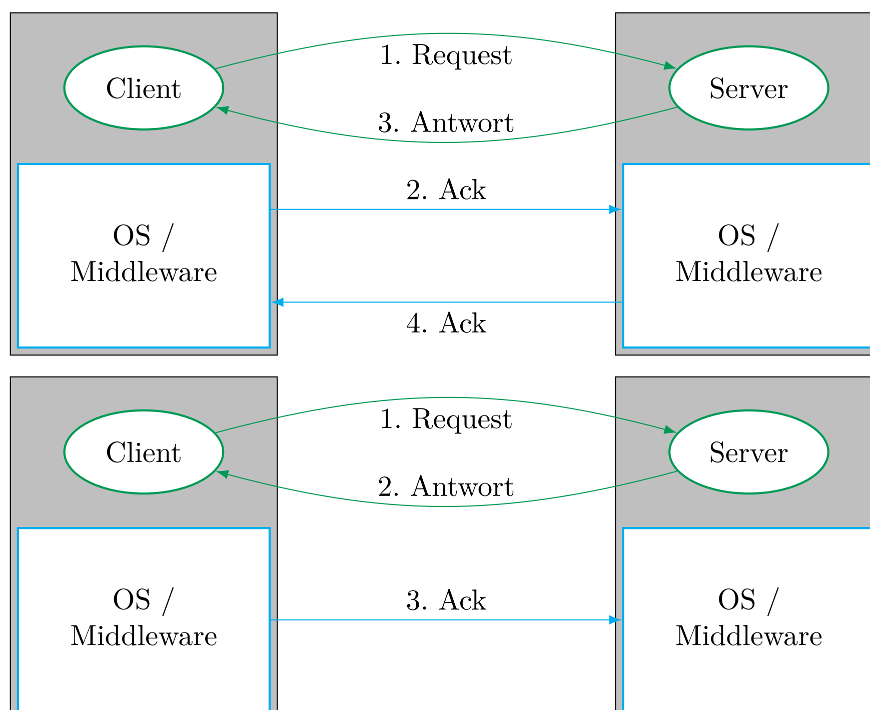
- At-Most-Once: Anfrage wird nie mehrmals gesendet
- At-Least-Once: Anfrage wird stets mindestens einmal gesendet
- Exactly-Once: Anfrage wird stets genau einmal gesendet (In der Praxis nicht zuverlässig umsetzbar)

Beispielszenario für gewünschtes Exactly-Once-Verhalten:

- Client schickt eine Anfrage an den Server, um ein Ticket zu drucken
- Mögliche Ereignisse auf Serverseite:
  - M: Das Acknowledgement wird gesendet
  - P: Das Ticket wird gedruckt
  - C: Der Crash passiert
- Die 3 Ereignisse können in 6 möglichen Reihenfolgen auftreten

Retransmission Strategie	**Strategie** $M \rightarrow P$			**Strategie** $P \rightarrow M$		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Immer	DUP	OK	OK	DUP	DUP	OK
Nie	OK	ZERO	ZERO	OK	OK	ZERO
Nur wenn ack	DUP	OK	ZERO	DUP	OK	ZERO
Nur wenn kein ack	OK	ZERO	OK	OK	DUP	OK

- Keine der Strategien führt stets zu einem Exactly-Once-Verhalten
- Deshalb muss zuverlässige Kommunikation durch den Client realisiert werden (z.B. durch das Request-Reply-Acknoledgement-Protocol (RRA)



Fragen Kapitel 3.1

Frage 1

Nennen Sie ein Beispiel für einen Zero Client.

Frage 2

Welche Abstufungen von Client-Systemen gibt es?

Frage 3

Was bedeutet Scale-Up, was Scale-Out?

Frage 4

Welche Varianten von „In-Between“-Servern gibt es?

Frage 5

Wie verhindern Sie einen unbefugten Zugriff auf eine Datenbank innerhalb eines „Three-Tier-Setup“?

Frage 6

Was unterscheidet einen Proxy von einem Broker?

Frage 7

Was unterscheidet einen Broker von einem Trader?

Frage 8

Welche Vor- und Nachteile gib es bei Einsatz eines Brokers?

Frage 9

Nennen Sie zwei Beispiele eines Handle-Driven Brokers.

Frage 10

Erklären Sie „Balancer“ in Bezug zu einem VS.

Frage 11

Erklären Sie das Wort „Agent“ in Bezug zu einem VS.

Frage 12

Welche Vor- und Nachteile haben Stateful-Server? Werden sie im Vergleich zu Stateless-Servern öfter eingesetzt?

Antworten Kapitel 3.1

Antwort 1

Nennen Sie ein Beispiel für einen Zero Client.

Zero-Client: Client, der ausschließlich Eingaben weiterleitet und keinerlei Berechnungen vornimmt. z.B: Remote-Desktop-Session, Terminal (im Kontext eines Mainframe.)

Antwort 2

Welche Abstufungen von Client-Systemen gibt es?

- Zero Client
- Thin Client
- Applet Client
- Fat Client

Antwort 3

Was bedeutet Scale-Up, was Scale-Out?

Scale-up: Bessere Hardware, mehr Speicher

Scale-out: Aufteilung der Last auf mehrere Server

Antwort 4

Welche Varianten von „In-Between"-Servern gibt es?

- Proxy
- Broker
- Trader
- Load-Balancer
- Agent

Antwort 5

Wie verhindern Sie einen unbefugten Zugriff auf eine Datenbank innerhalb eines „Three-Tier-Setup"?

- Wir könnten einen Security-Proxy vor die Datenbank schalten der unbefugte Zugriffe verhindert.
- DMZ verwenden
- Firewall verwenden
- Access-Tokens verwenden

Antwort 6

Was unterscheidet einen Proxy von einem Broker?

Ein Proxy ist ein vorgeschalteter Server / Vermittler für weitere dahinter geschaltete Application-Server. Eine Art Einfallstor.

Grundlegend kann der Broker intelligente Entscheidungen über die richtige Auswahl des Application-Servers treffen und somit eine Vielzahl von Fehlern maskieren.

Intuition: Broker = Proxy + intelligente Auswahl des Servers

Antwort 7

Was unterscheidet einen Broker von einem Trader?

Ein Trader ist eine Art Broker, aber hier erfolgt die Auswahl des Servers mit identischen Interfaces; die Wahl erfolgt also nach Maßstäben des Quality of Service.

Antwort 8

Welche Vor- und Nachteile gib es bei Einsatz eines Brokers?

Vorteile: Location und Replication Transparency

Nachteile: Bottleneck, Single-Point-Of-Failure

Antwort 9

Nennen Sie zwei Beispiele eines Handle-Driven Brokers.

- DNS
- Kerberos
- IP-Adressvergabe (DHCP)
- Content Distribution Network

Antwort 10

Erklären Sie „Balancer“ in Bezug zu einem VS.

Balancer (= Load-Balancer) verteilen Client-Requests nach aktuellem Workload der Server. Ziel ist eine gleichmäßige Belastung der Server und die Response-Time-Minimierung für die Clients.

Ein Balancer muss dafür jederzeit den Workload jedes beteiligten Servers kennen, ihn also kontinuierlich prüfen.

Antwort 11

Erklären Sie das Wort „Agent“ in Bezug zu einem VS.

Ein Agent ist dafür zuständig, komplexe Anfragen in simplere Anfragen zu zerlegen, diese an unterschiedliche Server weiterzuleiten und die Antworten zum Schluss zu einer einzelnen Antwort zusammenzufügen. (Bsp.: Reisebuchung)

Antwort 12

Welche Vor- und Nachteile haben Stateful-Server? Werden sie im Vergleich zu Stateless-Servern öfter eingesetzt?

Stateful-Server können aufgrund ihrer Daten User-Aktionen vorhersehen. Das Speichern und Berechnen sorgt für eine starke Auslastung der Rechner weshalb sie eher selten eingesetzt werden.

Stateless ist der defacto-Standard im Web. Client-Informationen werden häufig über Cookies gelöst.

Kapitel 3.2. Peer-to-Peer Systeme

(a) Merkmale von P2P-Systemen

- Gleichberechtigte Knoten ohne zentralen Master/Koordinator
- Alle Knoten (Peers) laufen mit dem gleichen Programm und interagieren kooperativ
- Typischerweise eine große Anzahl an Knoten im Netzwerk

(b) Kernkonzepte

- Selbstorganisation, kein zentrales Management
- Ressourcenverteilung
- Peers sind autonom, können kommen und gehen
- Adressierung findet auf Anwendungsebene statt
- Lookup-Mechanismus
- Dezentralisierter Speicher

(c) Vorteile/Nachteile

Vorteile

- Kein Single Point of Failure durch bessere Zuverlässigkeit und Erreichbarkeit.
- Daten sind einfacher zu veröffentlichen und damit auch einfacher aktuell zu halten.
- Anonymität bleibt gewahrt,
- Skalierbarkeit
- Geteilte Ressourcen wie z.B. Rechenleistung, Speicher und Bandbreite
- Keine Manipulation oder Zensur eines einzelnen Knoten.

Nachteile

- schwierig zu entwickeln und zu verwalten.
  - Programmieren
  - Debuggen
  - Security
  - Routing und Suche
  - Reproduzieren
- Es ist keine Zuverlässigkeit garantiert.
- teilnehmende Peers sind unter Umständen nicht vertrauenswürdig.

(d) P2P-Beispiel Gnutella

- Verbindungsaufbau über einige bekannte Peers
- Datensuche via query-flooding, wobei jede Anfrage eine TTL hat (default=7)
- Identische Anfragen werden abgelehnt
- Jede Message hat folgenden Header:

Field	Bytes	Beschreibung
ID	16	Zufällige Nummer zur Unterscheidung der Nachrichten
Descriptor	1	Typ der Message (Ping, Pong, Query, Query_Hit, Push)
TTL	1	Time to Live
Hops	1	Anzahl bereits beantragter Weiterleitungen
Length	3	Länge des Payloads

Tabelle 3.1: Header

- Grund für die TTL = 7 ist Milgram's Experiment: 2 Zufällige Personen auf dem Globus kennen sich im Schnitt über 6 Ecken

Kapitel 3.3. Publish/Subscribe Systeme

- Internet of Things, Big Data etc. verlangen den vermehrten Austausch von großen, nicht-statischen Datenströmen
- Oft sind Systeme, die auf einzelnen Queries basieren, dafür nicht ausgelegt (Client Server z.B)
- Subscription/Filter eignen sich mehr für Anwendungen wie:
  - Personalisierte News-Services
  - Automatisierte Bestandüberprüfung
  - Verteilte Auktionsplattformen (Ebay)
  - Sensoren-Netzwerke
  - Near-Real-Time- und IoT-Anwendungen

(a) Wichtige Merkmale

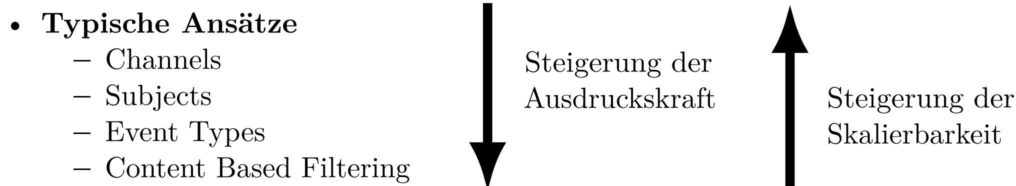
- Asynchrone Kommunikation
- Anonyme Adressierung (Loose Coupling, richtungslos)
- Vorteile für informations-fokussierten Anwendungen:
  - Kein ineffizientes Polling nötig
  - Geringe Latenz, da die Übertragung vom Producer initiiert wird
  - Die Verteilung an viele Nutzer ermöglicht Optimierungen

(b) Weitere Fragen bezüglich Pub/Sub-Systemen

- Wie wählt man nur Nachrichten von Interesse? $\leadsto$ Selektion der Notifications
- Wie werden Nachrichten nur an interessierte Benutzer gesendet? $\leadsto$ Ein Broker-Netzwerk wird benutzt um ein dezentralisiertes Notification Routing umzusetzen

(c) Notification Selection

Je nach Ansatz entsteht ein Tradeoff zwischen Skalierbarkeit(Scalability) und Ausdrucksstärke(Expressiveness)



- **Channels**
  Simpel, ohne Filterung
- **Topics/Subjects**
  - Notifications beinhalten einen Topic-Identifier
  - Topics können hierarchisch organisiert werden
- **Event Types**
  - Notifications sind nach Events kategorisiert
  - Normalerweise mit abstrakter Superklasse „Event“
  - Consumer abonnieren Unterklassen von der Event-Klasse
- **Content Based Routing**
  - Gezielte Filterung nach bestimmten Attributen
    z.B. `{Stock s | s.name = Microsoft ∧ s.price < 15}`

(d) Broker Network

- Broker koordinieren die Benachrichtigungen dezentralisiert
- Jeder Broker übernimmt eine Teilmenge der Clients
- Einfacher in azyklischer Topologie
- Broker kriegen die Notifications durch Flooding
  - Kann IP-Multicast ausnutzen
  - Keine Routing Tabelle nötig
  - Aber: Unnötiger Overhead durch ineffizientes Forwarding
  - Effektiv, wenn die meisten Notifications an den Großteil der Broker gehen sollen
- Routing Tables stellen oft eine Verbesserung dar

(e) Kafka - Beispiel für Service zur Datenstromverarbeitung

Aufbau

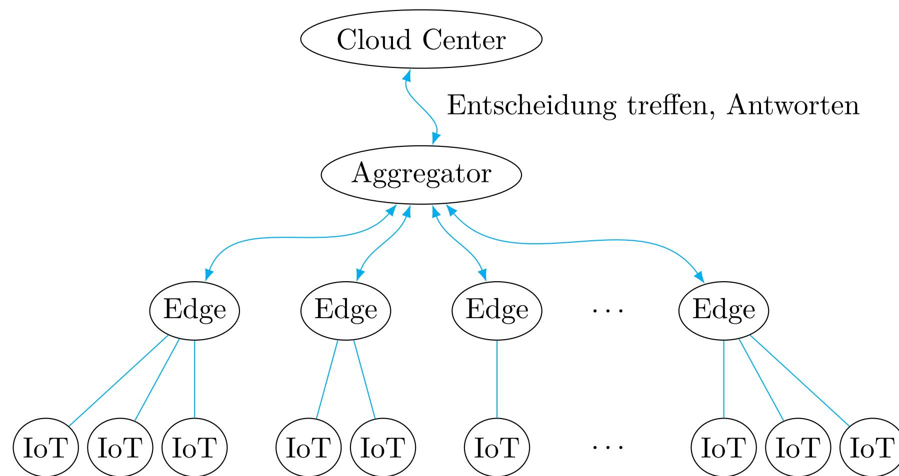


Abbildung 3.1: IoT und die Rolle von Kafka

(f) Hauptaufgaben

- Datenquellen zusammenführen
- Daten an den Abnehmer senden (Big Data Engines, Monitoring etc.)
- Ergebnisse in DB ablegen
- Daten klassifizieren

(g) Kafka vs. traditionelle Messaging Systeme (JMS, AMQ)

- Kafka priorisiert:
  - Scalability (TB an Daten)
  - Fault Tolerance (Replication, Partitioning)
  - Konnektivität über Netzwerkgrenzen hinweg
  - Simple Integration
- Kafka hat keine Acknowledgements, kümmert sich also nicht darum, dass Daten sicher ankommen

(h) Eigenschaften eines Streams

- Besteht aus (Key, Value, Time)-Tripeln
- Kafka speichert Streams in Kategorien (Topics)
- Können zuverlässig gespeichert werden
- Können möglichst latenzfrei verarbeitet werden

Kapitel 3.4. CAP-Theorem

(a) Microservices

> *The microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.*

(b) Kubernetes

- Plattform zur Verwaltung von partitionierten Workloads
- Kubernetes unterstützt unter Anderem:
  - Load Balancing
  - Organisation von logischen Speichereinheiten
  - Automatische Rollouts und Rollbacks
  - Selbstheilung: Neustart von gecrashten Containern

(c) Availibility

- Sicherheit durch Redundanz
- Steht im Konflikt mit Consistency

(d) Brewer's CAP-Theorem

In einem VS können niemals alle der folgenden 3 Ziele gleichzeitig erreicht werden:

- **Consistency**
  - Alle Replikas eines Daten-Elements sind konsistent untereinander
  - Falls ein Daten-Element eines Replikas geupdatet wird, dann werden alle anderen auch gleich geupdatet
  - Eine `read()` Operation gibt immer den Wert, den die jüngste `write()` Operation verursacht hat zurück
- **Availability**
  Das VS reagiert immer auf Anfragen und die Antwortzeiten sind akzeptabel
- **Partition Tolerance**
  Das System läuft korrekt, auch wenn Komponenten und/oder Kommunikationskanäle ausfallen

Fragen Kapitel 3.4

Frage 1

Was sind die Kernkonzepte eines Peer-to-Peer Netzwerks?

Frage 2

Welche Vor und Nachteile haben Peer-to-Peer Netzwerke?

Frage 3

Mit welcher Maßnahme verhindert man bei der Suche innerhalb eines VS endlose Weiterleitungen?

Frage 4

Nennen Sie eine Anwendung für Publish / Subscribe Systeme.

Frage 5

Erklären Sie die Struktur eines Broker Netzwerks.

Antworten Kapitel 3.4

Antwort 1

Was sind die Kernkonzepte eines Peer-to-Peer Netzwerks?

- Die Peers sind gleichberechtigt
- Die Peers erfüllen die gleiche Rolle und haben gleiche Verantwortlichkeiten
- Die Peers lassen das selbe Programm laufen
- Alle Peers haben das gleiche Interface
- Kooperative Interaktion
- Es gibt kein zentrales Management, das System ist selbstorganisierend
- Ressourcen werden geteilt
- Die Peers sind autonom
- Die Adressierung passiert auf dem Application Level
- Üblicherweise gibt es viele Peers
- Es gibt einen Lookup Mechanismus
- Die Übertragung und Speicherung von Daten passiert dezentral
- Das Netzwerk ist selbstskalierend

Antwort 2

Welche Vor und Nachteile haben Peer-to-Peer Netzwerke?

Vorteile

- Kein Single-Point-Of-Failure
- Die Upload Kapazitäten der Peers wird genutzt $\Rightarrow$ Höherer Throughput möglich
- Es gibt keinen zentralen Datenspeicher oder Server $\Rightarrow$ Erhöhte Privacy
- Selbstskalierendes Netzwerk

Nachteile

- Die Programmierung eines Peer-to-Peer Netzwerkes ist komplizierter als ein Client-Server-System
- Die Laufzeit der benötigten Lookup-Operation skaliert nicht gut
- Peers müssen nicht immer online sein, wenn wenige online sind kann die Download Rate schlecht sein bzw. Daten überhaupt nicht gedownloaded werden

Antwort 3

Mit welcher Maßnahme verhindert man bei der Suche innerhalb eines VS endlose Weiterleitungen?

Inkrementieren einer TTL. Wenn die TTL-Grenze erreicht wird, wird das Paket verworfen und nicht mehr weitergeleitet.

Antwort 4

Nennen Sie eine Anwendung für Publish / Subscribe Systeme.

- Chat-Gruppen in Messenger-Diensten
- Auslesen von Sensordaten aus Netzwerken.

Antwort 5

Erklären Sie die Struktur eines Broker Netzwerks.

Ein Broker Netzwerk ist ein Publish- / Subscribe System mit einem verteiltem Notification Service und besteht aus Clients und Brokern, die miteinander über ein Underlay Netzwerk verbunden sind.

Die Clients können sich bei den Brokern für ein Thema anmelden um dann Benachrichtigungen zu erhalten, falls es neue Nachrichten über das Thema gibt. Ein Client ist mit genau einem Broker verbunden, die Broker können aber mit mehreren Clients und Broker verbunden sein.

Die Broker leiten für die Clients relevanten Nachrichten an sie weiter. Die Broker kooperieren miteinander, bieten generell aber den gleichen Service an.

Kapitel 4

Programmiermodelle

Kapitel 4.1. Intro

Verteilte Systeme bestehen aus verschiedenen Komponenten, welche in verschiedenen Prozessen ausgeführt werden und miteinander kooperieren müssen. Dazu muss der „Distributed System Layer“ – oder auch Middleware genannt – eine Abstraktionsschicht schaffen, sodass die Anwendungen problemlos miteinander kommunizieren können.

(a) Remote Method Invocation

Es gibt die folgenden Komponenten

- **Remote Interface**

  Das Remote Interface beschreibt die Funktion, die auf dem Objekt zur Verfügung stehen und definiert damit das Verhalten.

- **Remote Object**

  Das Remote Object stellt das entfernte Objekt dar und liegt auf dem Server. Es implementiert das Remote Interface und das Verhalten.

- **Remote Object Reference**

  Die Remote Object Reference ist eine Referenz auf das entfernte Objekt. Sie ist global eindeutig.

Dies ist eine schematische Skizze, wie das Kommunizieren mit einem entfernten Objekt aussieht.

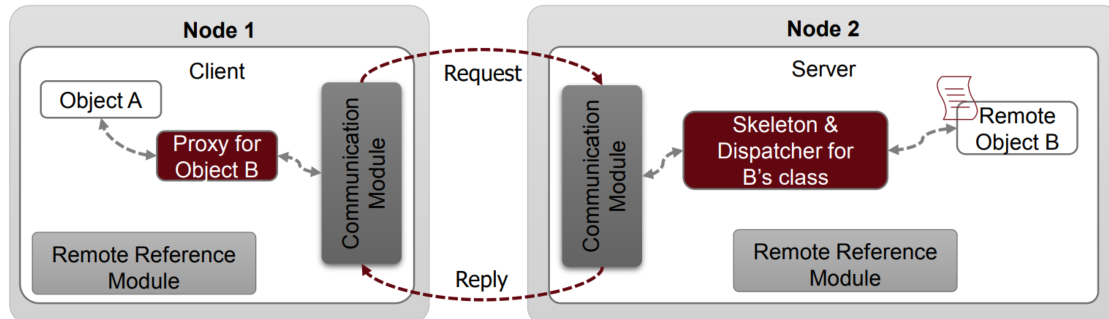


Abbildung 4.1: Kommunizieren mit einem entfernten Objekt
Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 4.1 Slide 10

Folgende Module können in der Skizze gefunden werden:

- **Communication Module**

  Das Communication Module kommuniziert mithilfe eines Request-Reply-Protokolls mit dem entfernten Objekt bzw. dessen Communication Module.

- **Remote reference module**

  Dieses Modul ist für das Übersetzen von lokalen und entfernten Adressen in die jeweils andere Adresse verantwortlich.

- **Proxy**

  Der Proxy übernimmt das (Un-)Marshalling und die Übertragung zum Communication Module. Dies geschieht transparent für den Nutzer.

- **Dispatcher**
  Der Dispatcher bekommt die Anfragen vom Communication module und ruft die entsprechende Methode im Skeleton auf um die Nachricht zu verarbeiten.
- **Skeleton**
  Implementiert die Methoden, welche durch das Remote Interface spezifiziert werden, sodass das Objekt die Anfrage verstehen kann.

Der Ablauf einer Method Invocation bei Verwendung von CORBA kann wie folgt dargestellt werden:

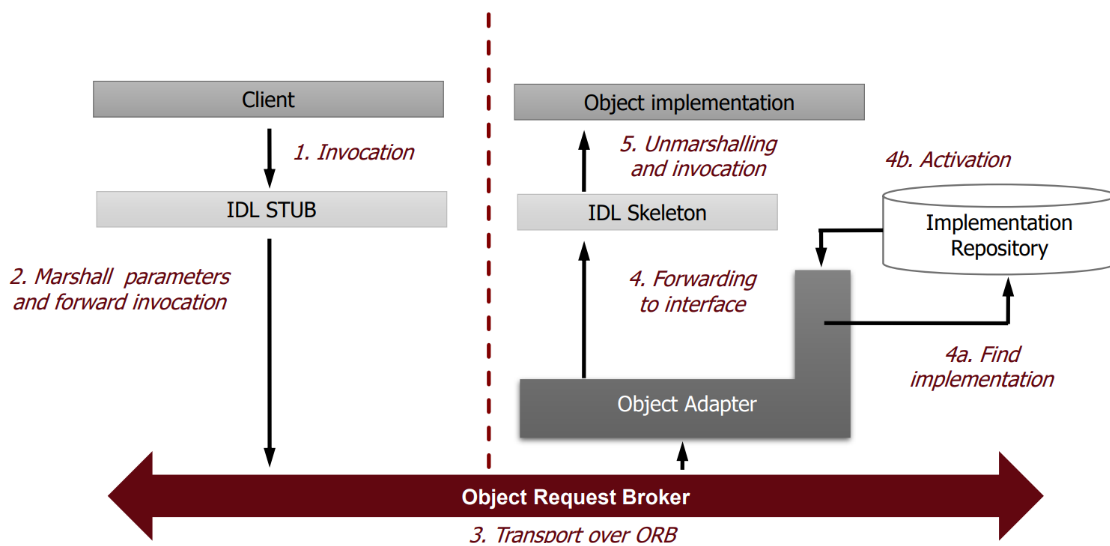


Abbildung 4.2: Static CORBA – Method Invocation
Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 4.1 Slide 15

Fragen Kapitel 4.1

Frage 1

Wie funktioniert RMI (Remote Method Invocation)?

Frage 2

Was ist eine Remote Object Reference?

Frage 3

Worauf muss bei der Vergabe der Remote Object References geachtet werden?

Frage 4

Was ist ein Remote Interface im Bezug auf RMI's?

Frage 5

Skizziere wie ein Client via RMI mit einem Server kommunziert. Skizziere den Proxy, das Remote Reference Module usw. ein.

Frage 6

Was ist ein „Remote Reference Modul“, „Skeleton“, „Dispatcher“, „Communication Module“?

Antworten Kapitel 4.1

Antwort 1

Wie funktioniert RMI (Remote Method Invocation)?

Wir nehmen an, dass eine Remote Object Reference gegeben ist. Dann ruft der Client eine Methode aus der Referenz auf, wobei diese durch das Remote Object Interface gegeben ist.

Dann führt das entfernte Objekt den Code aus und gibt das Ergebnis zurück.

Antwort 2

Was ist eine Remote Object Reference?

Eine Remote Object Reference ist eine global eindeutige Referenz auf ein Objekt, welches entfernt ist. Siehe Remote Method Invocation.

Antwort 3

Worauf muss bei der Vergabe der Remote Object References geachtet werden?

Es muss darauf geachtet werden, dass sie global eindeutig sind. Das kann erreicht werden indem man bereits global eindeutige Kennung in die Reference mit aufnimmt.

Antwort 4

Was ist ein Remote Interface im Bezug auf RMI's?

Ein Remote Interface spezifiziert für ein Remote Object - wie z.B. ein Java Interface - die Eingabe-Parameter und die Werte die die Methoden zurückgeben. Siehe Remote Method Invocation

Antwort 5

Skizziere wie ein Client via RMI mit einem Server kommunziert. Skizziere den Proxy, das Remote Reference Module usw. ein.

Siehe Figure 4.1

Antwort 6

Was ist ein „Remote Reference Modul", „Skeleton", „Dispatcher", „Communication Module"?

Siehe diese Erklärung

Kapitel 4.2. Service Oriented Architecture

- Das Ziel der SOA ist es aus einzelnen Prozeduren Services zu erstellen, welche in sich geschlossen sind und feste Interfaces besitzen.
- Entsprechend sind die einzelnen Komponenten unabhängig und lose gekoppelt.
- Sie bilden außerdem zusammen ein Verteiltes System.

(a) Workflows

Die folgenden Modelle sind Beschreibungen, wie eine Komposition aus mehreren Web-Services aussehen kann. Dabei werden beide Varianten asynchron ausgeführt – zumindest dort wo es möglich ist.

Web Service Orchestration

- Ein *zentraler* Controller komponiert verschiedene Services miteinander.
- Dabei gibt der Controller an, welche Aktionen von welchen Web-Services wann ausgeführt werden sollen.
- Es wird definiert welche Prozesse welche bereits existierenden Web-Services benutzen

Web Service Choreography

- Die Web-Services werden komponiert um die Kollaboration von Business-Partnern zu ermöglichen
- Es wir die Peer-To-Peer Kollaboration zwischen den Partnern definiert
- Kritisch ist die formale Definition des Nachrichten-Austauschs

(b) Microservices

Microservices sind ein Versuch einzelne Services noch kleiner zu machen: So klein, dass man sie nicht sinnvoll weiter aufteilen kann. Entsprechend ist jeder Microservice ein unabhängiger Prozess, welcher, zusammen mit anderen Prozessen, ein Verteiltes System bildet.

Dabei besitzen sie die folgenden Eigenschaften

- Die Aufteilung in Komponenten wird anhand der Services gemacht die bereitgestellt werden: Diskreter Service $\Rightarrow$ Diskrete Komponente
- „Smart Endpoints and Dumb Pipes“

  Das bedeutet, dass die zentrale Logik in einem Microprocess stattfindet und nicht in dem Messaging-Service
- Dezentralisiertes Datenmanagement
- Automatisierte Infrastruktur: „Continuous Integration / Continuous Delivery“ (CI/CD)
- Dezentralisierte Autorität

Fragen Kapitel 4.2

Frage 1

Wie funktioniert RMI bei CORBA? Skizziere dafür den Ablauf, wenn ein Client eine entfernte Methode aufruft.

Frage 2

Was ist der Unterschied zwischen SOA und Microservices?

Frage 3

Was ist der Unterschied zwischen Web Service Orchestration und Web Service Choreography?

Antworten Kapitel 4.2

Antwort 1

Wie funktioniert RMI bei CORBA? Skizziere dafür den Ablauf, wenn ein Client eine entfernte Methode aufruft.

Siehe Figure 4.2

Antwort 2

Was ist der Unterschied zwischen SOA und Microservices?

	Microservices	SOA
Architektur	Mehrere Services, welche unabhängig voneinander funktionieren	Die einzelnen Komponenten teilen sich Resourcen
Teilen von Komponenten	Nein	Ja
Granularität	Sehr granular	Weniger granular
Datenspeicherung	Jeder Service speichert seine eigenen Daten	Die Daten werden zwischen Services geteilt
Use-case	Kleinere Web-Anwendungen	Große, skalierungsbedürftige, Integrationen

Bei SOA ist das Ziel mehrere Anwendungen miteinander zu integrieren; bei Microservices hingegen werden Services benutzt um eine Anwendung zu strukturieren.

Antwort 3

Was ist der Unterschied zwischen Web Service Orchestration und Web Service Choreography?

Der Unterschied besteht darin, dass bei Web Service Orchestration ein zentraler Koordinator die Anfragen ausführt.

Bei Web Service Choreography koordinieren sich die Prozesse selbst.

Kapitel 4.3. Event Driven Architecture

(a) Komponenten

Erinnerung: In Definitionen haben wir ein Event definiert als eine einzelne Aktion, die ein Prozess ausführt.

Die EDA findet typischerweise zwischen zwei Parteien statt:

- **Event-Generator**
  Etwas, was ein Event generieren kann, z.B. in Sensor, ein User, etc.
- **Downstream Activity**
  Eine Aktivität, welche Events consumed.

Diese Parteien müssen nun vernetzt werden. Dies übernimmt der Event Processing Layer.

(b) Event Processing

Das Event Processing wird von dem Event Processing Layer (EPL) übernommen. Dieser wertet Events aus und initiiert Aktionen.

Teil des EPL ist die Event Engine:

- Sie übernimmt das Verarbeiten der Daten
- **Simpel**
  Jedes Event wird ohne Wissen von vorherigen Events unabhängig voneinander bearbeitet
- **Komplex**
  Das Wissen von vorherigen Events kann in das Verarbeiten des aktuellen Events mit einfließen

Dabei gibt es verschiedene Arten des Processings

- **Simple Event Processing**
  - Es gibt nur wichtige Events, welche Aktionen ausführen
  - *Nutzen*: Optimierung des Informationsflusses in dem System
- **Stream Event Processing**
  - Es gibt sowohl wichtige als auch unwichtige Events.
  - Diese werden unterschieden und wichtige Nachrichten an die Downstream activity weitergeleitet.
  - *Nutzen*: Ermöglicht das Fällen von Entscheidungen
- **Complex Event Processing**
  - Beim Bearbeiten des Events wird noch zusätzlicher Kontext der letzten Anfragen miteinbezogen
  - Entsprechend können Events aggregiert werden
  - *Nutzen*: Erkennen und Bearbeiten von Anomalien und Problematiken

Event Channel

Um die Downstream Activity über das Event zu informieren, benötigt der EPL außerdem noch einen Channel, über den er kommunizieren kann. Dazu werden meist Publish-Subscribe-Systeme oder Message-Queues verwendet.

Dabei kann das System wie folgt aussehen:

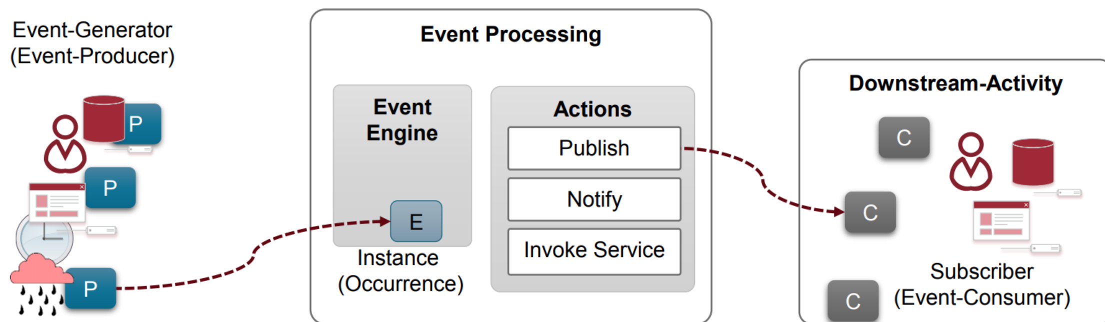


Abbildung 4.3: Beispiel der EDA
Quelle: © Distributed and Operating Systems (DOS) – Modul VS 2021 – VL 4.1 Slide 48

Fragen Kapitel 4.3

Frage 1

Was zeichnet eine Event-Driven Architektur aus?

Frage 2

Definiere die Worte Event Generators, Downstream Activity und Event Processing.

Frage 3

Welche Event Processing Styles gibt es? Was zeichnet sie aus?

Antworten Kapitel 4.3

Antwort 1

Was zeichnet eine Event-Driven Architektur aus?

Eine Event-Driven Architektur definiert verschiedene Komponenten: Event Generator(en), eine Event Processing Komponente und Downstream Activity(s).

Dabei werden Events durch die Generatoren produziert, welche von der Event Processing Komponente verarbeitet werden, und je nach Nachricht, an die entsprechenden Subscriber gesendet.

Antwort 2

Definiere die Worte Event Generators, Downstream Activity und Event Processing.

Siehe Komponenten.

Antwort 3

Welche Event Processing Styles gibt es? Was zeichnet sie aus?

Siehe Event Processing.

Kapitel 4.4. Enterprise Application Integration

Oftmals gibt es das Problem, dass einzelne Anwendungen miteinander vernetzt werden müssen. Dabei können die einzelnen Anwendungen sehr unterschiedlich sein: Verschiedene Programmiersprachen, verschiedene Systeme, etc.

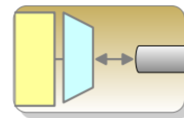
Oftmals möchte man Loose Coupling unter den einzelnen Komponenten erzielen. Das bedeutet, dass man möglichst wenige Annahmen über die Architektur machen möchte, was aber zur Folge hat, dass die Kommunikation ineffizienter wird.

(a) Patterns

Es gibt eine Vielzahl an Komponenten, welche in Patterns auftreten. Diese sind hier aufgeführt:

ChannelAdapter

Ein Channel ermöglicht es einer Anwendung mit einem Message Channel verbunden zu sein.



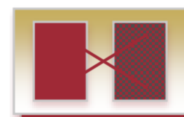
MessageEndpoint

Ein Message Endpoint verbindet eine Anwendung - ähnlich wie ein Channel Adapter - mit einem Channel.

Der Unterschied zu einem Channel Adapter ist, dass er direkt in der Anwendung verbaut ist, und nicht, wie der Channel Adapter, von außerhalb der Anwendung agiert.

MessageTranslator

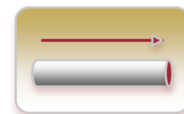
Ein Message Translator übersetzt Nachrichten in ein anderes Format – und zwar in ein Format, das die Ziel-Anwendung verstehen kann.



Message Channel / Point-to-Point Channel

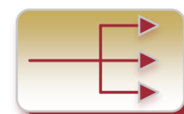
Ein Message Channel ist eine Verbindung zwischen einem Sender und möglicherweise mehreren Empfängern.

Dabei wird sichergestellt, dass die Nachricht ankommt. Falls mehrere Consumer involviert sind, so kann nur einer die Nachricht lesen.



Publish-Subscribe Channel

Eine Anwendung veröffentlicht eine Nachricht und eine beliebige Anzahl an Consumer können diese Nachricht lesen.



Aggregator

Ein Aggregator sammelt und speichert individuelle Nachrichten bis ein Satz an Nachrichten empfangen wurde. Diese werden dann zu einer einzelnen Nachricht zusammengefasst.

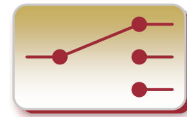
Dabei muss Folgendes beachtet werden:

- **Korrelation:**
  Welche Nachrichten gehören zusammen?
- **Vollständigkeit:**
  Wann sind wir fertig?
- **Aggregations Algorithmus**
  Wie sollen die Nachrichten miteinander kombiniert werden?

Content-Based Router

Ein Content-Based Router schickt jede Nachricht zu dem - abhängig vom Inhalt der Nachricht - korrekten Consumer.

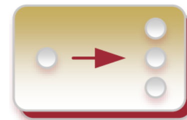
Entsprechend muss der Router die Nachricht verarbeiten und evaluieren.



Splitter

Ein Splitter unterteilt eine Nachricht in einzelne Nachrichten, damit diese durch andere Komponenten verarbeitet werden können.

Da die Nachricht am Ende zusammengefügt werden muss, entsteht dabei zusätzlicher Overhead, da bestimmte Metainformationen in jeder Nachricht enthalten sein müssen.



ContentEnricher

Ein Content Enricher ist ein spezialisierter Transformator. Das bedeutet, dass er eine externe Datenquelle anfragt um die Nachricht mit zusätzlichem Inhalt zu füllen.

Beispiele für Zusatz-Informationen:

- Berechnung
  - Correlation ID
- Umgebung
  - Zeit
- Ein anderes System
  - Ob eine Datei anwesend war

Fragen Kapitel 4.4

Frage 1

Warum brauchen wir EAI?

Frage 2

Was macht EAI komplex?

Frage 3

Welche Vorteile birgt Loose Coupling?

Frage 4

Warum sind EAI Patterns nützlich?

Frage 5

Welche fundamentalen Ansätze gibt es um EAI durchzuführen?

Frage 6

Welche Patterns wurden in der Vorlesung bzw. auf den Folien gezeigt. Beschreibe sie in jeweils einem Satz.

Antworten Kapitel 4.4

Antwort 1

Warum brauchen wir EAI?

Viele der vorhandenen IT-Systeme sind über die Zeit entstanden und hatten entsprechend verschiedene Anforderung. Deshalb können sie in verschiedenen Programmiersprachen und mit verschiedenen Programmierparadigmen geschrieben worden sein, was es sehr schwer macht diese mit einander zu vernetzen.

Um dennoch eine Lösung für das Problem zu finden gibt es die EAI. Diese bieten einen Weg verschiedene Anwendungen miteinander zu vernetzen, ohne dass eine Anwendung umgeschrieben werden muss.

Antwort 2

Was macht EAI komplex?

Da wir mit einer großen Anzahl unterschiedlicher Systeme konfrontiert sind und diese womöglich in verschiedenen Programmiersprachen geschrieben sind, ist es sehr schwer einen gemeinsamen Nenner zu finden.

Dabei treten Probleme wie Tight coupling, dass bestimmte Anwendungen nicht mehr verfügbar sind bzw. nur noch als binaries verfügbar sind oder, dass der Programmierer, welche die Kenntnis besitzt, nicht mehr da ist auf.

Entsprechend ist es sinnvoll vorhandene Patterns zu benutzen um die Lösungsfindung zu vereinfachen.

Antwort 3

Welche Vorteile birgt Loose Coupling?

Loose Coupling macht Verteilte Systeme skalierbarer und Komponenten eines VS leichter austauschbar.

Dadurch wird Space- und Time-Decoupling ermöglicht.

Außerdem wird mit Tight Coupling – dem Gegenteil von Loose Coupling – die Geschwindigkeit des Systems meist von der langsamsten Komponente diktiert. Wenn beispielsweise der zentrale Webserver die Anfragen nicht schnell genug verarbeiten kann, so müssen alle auf ihn warten.

Des Weiteren ist es durch Loose Coupling deutlich einfacherer die Komponente zu debuggen und zu testen, da man jede Komponente einzeln testen kann.

Antwort 4

Warum sind EAI Patterns nützlich?

Oftmals sind die Probleme, welchen wir begegnen, bereits gelöst worden. Das Rad jedes Mal neu zu erfinden bringt meistens nichts – meistens ist das sogar kontraproduktiv.

Des Weiteren können wir mit der Anwendung bewährter Patterns oftmals erhebliche Zeit und Kosten-Ersparnisse erzielen.

Antwort 5

Welche fundamentalen Ansätze gibt es um EAI durchzuführen?

- File-Transfer
- Shared Database

- Remote Procedure Invocation
- Message-Passing

Antwort 6

Welche Patterns wurden in der Vorlesung bzw. auf den Folien gezeigt. Beschreibe sie in jeweils einem Satz.

Siehe Patterns.

Kapitel 5

Sicherheit

Die Sicherheit in Verteilten Systemen ist stark gefährdet. Angreifer können die folgenden Mittel nutzen, um Angriffe durchzuführen:

- Beliebige Nachrichten zu beliebigen Prozessen senden.
- Beliebige Nachrichten mitlesen und abspeichern

(a) Angriffsmethoden

- **Eavesdropping**
  Das Zuhören und Mitschneiden von Nachrichten als dritte, unbefugte Person.
- **Denial of Service**
  Ein Angriff, bei dem der Service unerreichbar gemacht werden soll.
- **Masquerading**
  Die Benutzung von falschen Anmeldeinformationen.
- **Repetition**
  Das Wieder-Versenden von bereits gesendeten Nachrichten.

(b) Communication- und Computer Security

Computer Security

Computer Security beschreibt das Entdecken und Beheben von kompromittierten Knoten des Systems. Dabei können die Knoten Server, aber auch Komponenten wie Router sein.

Communication Security

Communication Security beschreibt das Verhindern von Angriffen gegen die Kommunikation zwischen zwei Knoten.

Dabei gilt es folgende Punkte zu beachten:

- **Authentication**
  - Beschreibt die Möglichkeit einem Kommunikationspartner einen Beweis der Identität zu geben.
  - Wird meist mit Shared Secrets realisiert.
- **Availability**
  - Die Möglichkeit mit dem Partner zu sprechen mit dem man sprechen möchte.
  - In Notfällen ist es eine Notwendigkeit, dass Availability gewährleistet ist.
- **Confidentiality**
  - Nur der zugehörige Empfänger kann die Nachricht entschlüsseln.
  - Wird durch (a)symmetrische Verschlüsselung realisiert.
- **Integrity**
  - Eine Nachricht ist nicht unterwegs verändert worden.
  - Wird durch Checksummen und Signaturen realisiert.

(c) Kryptographie

Die Ziele der Kryptographie sind es eine Nachricht M von Klartext in eine Form M' zu übersetzen, sodass nur autorisierte Person diese entschlüsseln kann.

Definitionen

- V Verschlüsselungsfunktion
- K_E Verschlüsselungs Key
- V^{-1} Entschlüsselungsfunktion
- K_D Entschlüsselungs Key

Es gilt: $V^{-1}(V(M)) = M$.

Symmetrische Verschlüsselung

Symmetrische Verschlüsselung – auch Secret-Key Cryptography genannt – ist eine Form der Verschlüsselung, wo der Sender und Empfänger ein gemeinsames Geheimnis kennen. Dabei ist das Geheimnis meist ein Schlüssel.

Dieser wird dann verwendet um die Nachricht zu ver- bzw. entschlüsseln.

Probleme:

- Man benötigt für eine Kommunikation zwischen beliebigen Knoten in einem vollständigen Netzwerk $\frac{n(n-1)}{2}$ Schlüssel.
- Wie kann man eine Kommunikation mit einer fremden Anwendung starten, wenn noch nie ein Schlüssel ausgetauscht wurde?

Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung gibt es einen privaten Schlüssel K_D, welcher nicht geteilt wird, und einen öffentlichen Schlüssel K_E, welcher geteilt wird. Dabei gilt

$$V_{K_D}^{-1}(V_{K_E}(M)) = M$$
$$V_{K_E}^{-1}(V_{K_D}(M)) = M$$

Das bedeutet, dass die Verschlüsselung in beide Richtungen gehen kann. Zum einen kann eine Checksumme der Nachricht mit dem privaten Schlüssel verschlüsselt werden um die Authentizität zu garantieren ($\rightarrow$ digitale Signatur), zum anderen kann eine Nachricht mit dem öffentlichen Schlüssel verschlüsselt und dann mit dem privaten Schlüssel entschlüsselt werden ($\rightarrow$ Verschlüsselung).

Da das Verschlüsseln auf dem asymmetrischen Weg deutlich mehr Zeit kostet, wird meist ein hybrider Ansatz benutzt, wo zunächst mithilfe des asymmetrischen Verfahrens ein Schlüssel für die Symmetrische Kommunikation ausgetauscht wird. Der symmetrische Schlüssel wird dann verwendet um die weitere Kommunikation zu verschlüsseln.

Use-Cases

- Vertraulichkeit von gesendeten / gespeicherten Informationen (Verschlüsselung).

- Beweis der Authentizität (digitale Signaturen oder Message Authentication Codes / MAC-Codes).
- Beweis der Integrität (verschlüsselte Checksummen)
- Beweis der Authentizität von Servern / Software (Zertifikate)

Integrität

Oftmals wollen wir auch sicherstellen, dass eine Nachricht nicht modifiziert worden ist. Dies wird durch eine sichere Hash-Funktion H (wie z.B. SHA) ermöglicht. Dabei gilt $H : A_1 \to A_2$ wobei A_1 und A_2 Alphabete sind. Funktionswerte von H können effizient berechnet werden, Funktionswerte der Inversen von H allerdings nicht.

Für die Hash Funktion wird außerdem noch ein geteilter, geheimer, Schlüssel $K_{A,B} \in A_3$ benötigt, damit kein Angreifer den Hash fälschen kann. Dafür definieren wir:

$$\text{MAC} : A_1 \times A_3 \to A_2$$

Message Authentication Codes

1. $K_{A,B}$ wird als geteilter Schlüssel zwischen A und B berechnet und ausgetauscht.
2. $E \coloneqq \text{MAC}(M, K_{A,B})$.
3. A fügt E an die bereits verschlüsselte Nachricht M an.
4. A sendet die Nachricht.
5. B bekommt die Nachricht M', die möglicherweise verändert ist.
6. B berechnet $E' \coloneqq \text{MAC}(M', K_{A,B})$.
7. Falls $E = E'$, so ist die Nachricht von A gesandt, und nicht unterwegs verändert worden.

Da MAC's auf symmetrischer Verschlüsselung basieren, können diese nicht für Digitalen Signaturen verwendet werden.

Public Key Infrastructure

Nun ist die Frage, wie man sicherstellen kann, dass eine Person, mit der man spricht, auch wirklich die Person ist, die sie zu sein scheint.

Um dieses Problem zu lösen verwenden wir einen hierarchischen Ansatz. Dabei ist die oberste Schicht die Certification Authority (CA). Diese stellt Zertifikate aus, welche folgende Informationen enthalten:

- Versionsnummer
- Seriennummer
- Signatur Algorithmus (ID)
- Name des Ausstellers
- Validitäts-Zeitraum
- Name

- Public Key Information

Solch ein Zertifikat wird durch die CA generiert. Damit dies möglich ist, muss sich der Nutzer bei der CA authentifizieren z.B. mit einem Ausweis.

Mithilfe des privaten Schlüssels der CA wird dann das Zertifikat signiert. Wenn nun ein Nutzer der CA traut, so traut er auch dem Zertifikat.

Die Services die die Root-CA zur Verfügung stellt, können folgendermaßen dargestellt werden:

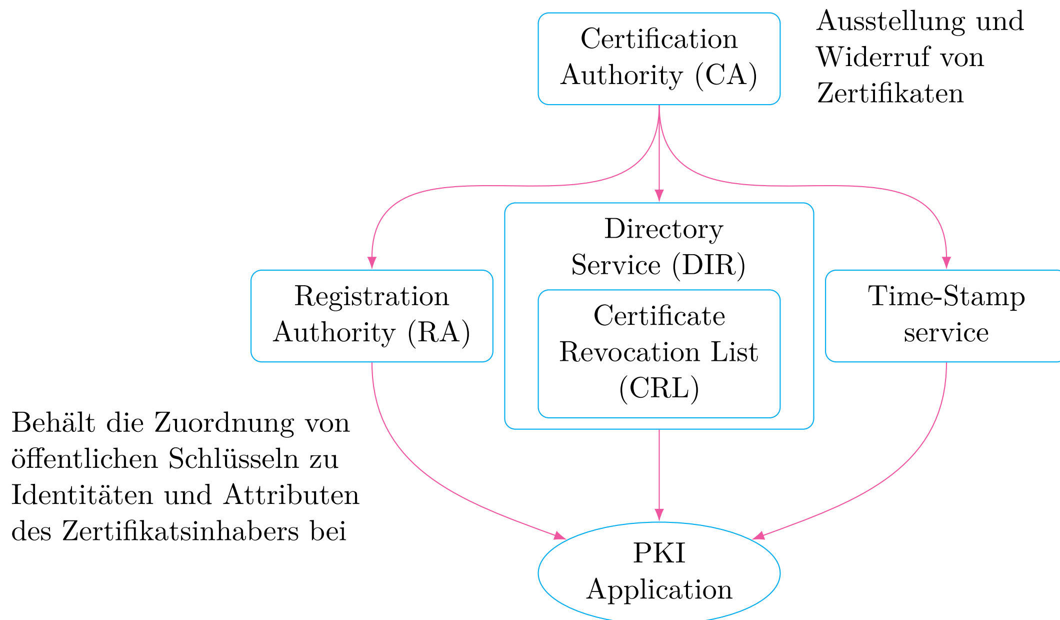


Abbildung 5.1: Ein Beispiel einer PKI

Dabei sind die CAs auch hierarchisch organisiert. Das bedeutet, dass es eine Root-CA gibt, welche dann Zertifikate für untergeordneten CA's ausstellt. Siehe Hierarchischer Aufbau der PKI.

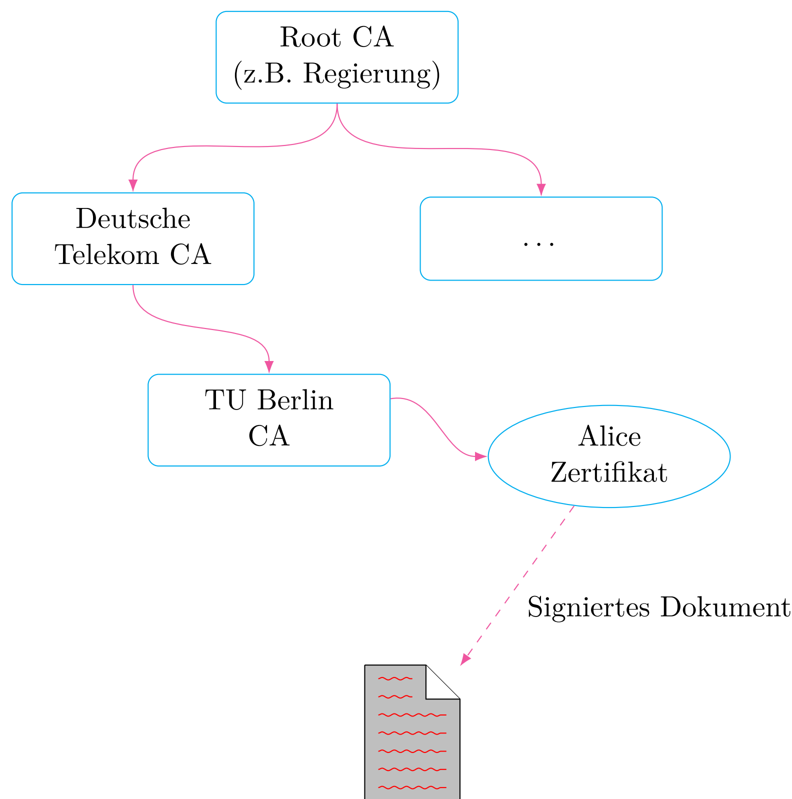


Abbildung 5.2: Hierarchischer Aufbau der PKI

(d) Authentifizierung auf einem lokalen System

In Intranets hat man oft das Problem, dass es eine zentrale Datenbank gibt, in der User-Daten gespeichert werden und jeder Service einen eigenen Authentifizierungs-Service aufgrund dieser Daten anbietet. Eine Methode dies zu realisieren wird vom **Needham Schröder Protokoll** umgesetzt.

Das Ziel ist der Austausch eines Schlüssels $K_{A\leftrightarrow B}$, welcher von A und B genutzt werden kann, um zu kommunizieren. Hier bezeichnet K_A den gemeinen Schlüssel von A und K_B den von B. Außerdem werden Nonces N_A und N_B von A und B generiert. Wenn etwas verschlüsselt ist, so wird es mit $\{\ldots\}_{K_X}$ notiert, wobei $X \in \{A, B\}$
Dann sieht der Ablauf wie folgt aus

1. *Von*: $A \to AS$, *Nachricht*: A, B, N_A
   A sendet eine Nachricht an AS für die Kommunikation mit B.
2. *Von*: $AS \to A$, *Nachricht*: $\left\{N_A, B, K_{A\leftrightarrow B}, \{K_{A\leftrightarrow B}, A\}_{K_B}\right\}_{K_A}$
   AS gibt eine Nachricht zurück, welche mit A's Schlüssel verschlüsselt ist. Diese besitzt ein Ticket, welches mit B's Schlüssel verschlüsselt ist.
3. *Von*: $A \leftrightarrow B$, *Nachricht*: $\{K_{A\leftrightarrow B}, A\}_{K_B}$
   A sendet das Ticket an B.
4. *Von*: $B \to A$, *Nachricht*: $\{N_B\}_{K_{A\leftrightarrow B}}$
   B entschlüsselt das Ticket und nutzt den Schlüssel $K_{A\leftrightarrow B}$ um das Nonce N_B zu verschlüsseln.

5. *Von*: $A \to B$, *Nachricht*: $\{N_B - 1\}_{K_{A\leftrightarrow B}}$
   A zeigt B, dass A der Sender der letzten Nachricht war indem gezeigt wird, dass A auch $K_{A\leftrightarrow B}$ besitzt.

Am Ende haben nun A und B einen Schlüssel, welcher benutzt werden kann um sicher zu kommunizieren.

Schwächen des Protokolls

Das Protokoll hat eine fundamentale Schwäche: N_A und N_B sollen „frisch" sein. Das dient dazu einen Replay-Angriff zu verhindern. Das Problem ist nun, dass ein Angreifer, sobald er den Schlüssel $K_{A\leftrightarrow B}$ bekommt, eine Kopie des Tickets $\{K_{A\leftrightarrow B}, A\}_{K_B}$ machen kann.

Dadurch kann er vorgaukeln A zu sein.

Dieses Problem kann umgangen werden, indem ein Zeitstempel t zur dritten Nachricht hinzugefügt wird, sodass die Nachricht statt $\{K_{A\leftrightarrow B}, A\}_{K_B}$ nun $\{K_{A\leftrightarrow B}, A, t\}_{K_B}$ ist.

(e) Kerberos

Bei Kerberos sind drei Parteien beteiligt

- **Der Client**
  Möchte auf einen Server zugreifen.
- **Der Server**
  Bietet ein Service an $\Rightarrow$ möchte vom Client genutzt werden.
- **Der Kerberos-Server**
  Bietet den Kerberos Dienst an, welcher Client und Server verbindet.

Ablauf

1. **Login**
   Zunächst authentifiziert sich der Client bei dem Authentifikationsserver, welcher am Ende ein TGT zurück sendet.
   1.1 C sendet den Username.
   1.2 Falls der User bekannt ist, so wird ein Ticket, verschlüsselt mit dem Passwort des Users, zurückgesandt. Dieses enthält die folgenden Elemente
      - Session Key
      - Nonce
      - Ticket für den Ticket Vergabeservice (TGT)

   Nun wird die Antwort mit dem Passwort von dem User entschlüsselt.
2. **Ticket Austausch**
   Nun benötigt der Client ein Ticket um auf den Server S zugreifen zu können. Dieses wird von T erledigt. Dabei werden nun alle Nachrichten mit dem Session Key verschlüsselt.
   2.1 Das TGT wird nun, verschlüsselt, zusammen mit der Information über den Server S, an T gesendet.

2.2 Der Schlüssel für die Kommunikation zwischen C und S wird zurückgesandt.

3. **Aktion ausführen**

   Nun fängt die Kommunikation zwischen Client und Sever an.

   3.1 Der Client wird mit der vorhin akquirierten Nachricht bei dem Server authentifiziert und kann den Service aufrufen.

   3.2 Die Antwort des Services

Der Ablauf kann mit folgendem Diagramm dargestellt werden

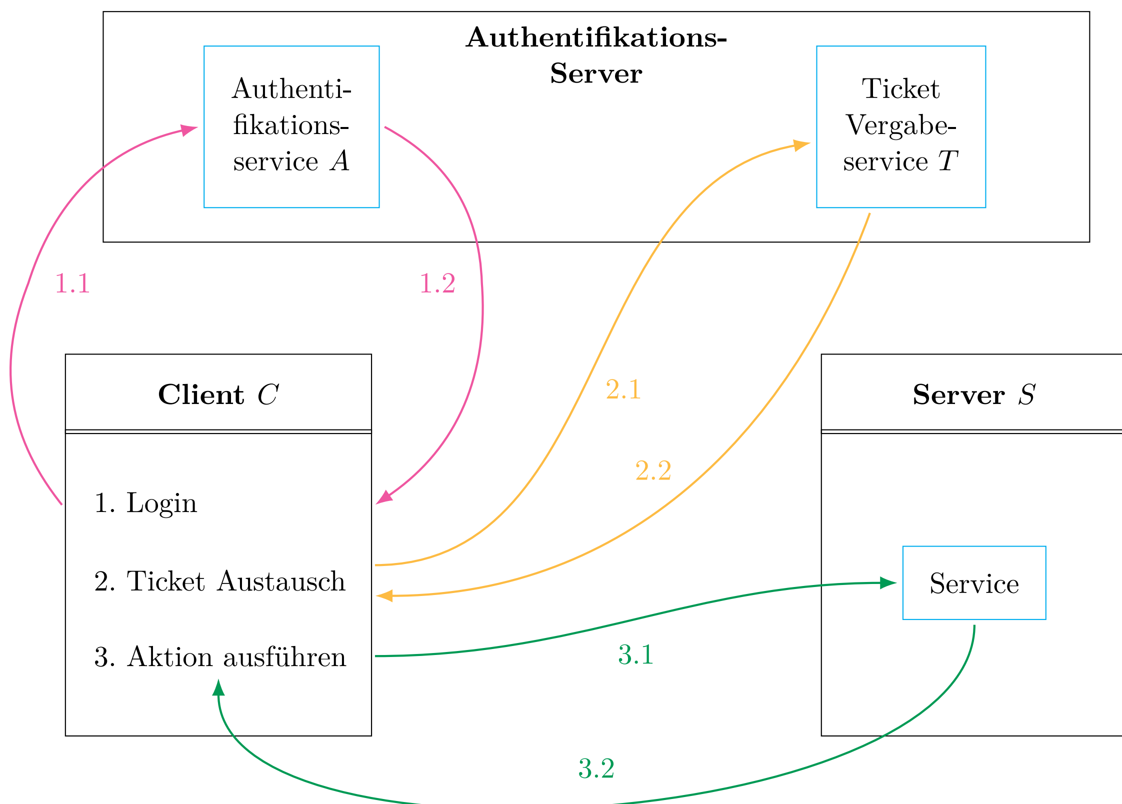


Abbildung 5.3: Ablauf von Kerberos

Fragen Kapitel 5

Frage 1

Warum sind VS schwer zu schützen?

Frage 2

Welche Angriffsvektoren gibt es? Was können wir unternehmen um uns gegen sie zu schützen.

Frage 3

Was ist der Unterschied zwischen Computer Security und Communication Security?

Frage 4

Welche Eigenschaften muss unser VS erfüllen um Communication Security zu gewährleisten?

Frage 5

Um ein VS zu schützen muss davon ausgegangen werden, dass der Angreifer gewisse Dinge tun kann um uns anzugreifen. Wovon müssen wir ausgehen? Was kann der Angreifer tun?

Frage 6

Wofür kann Cryptography genutzt werden, nenne die Use-Cases und welche Technologien dafür genutzt werden.

Frage 7

Was ist der Unterschied zwischen Secret-Key Cryptography und Asymmetric Cryptography?

Frage 8

Kann man Secret-Key Cryptography für digitale Signaturen verwenden? Begründe.

Frage 9

Beschreibe Step-by-Step wie A eine Nachricht M an B schicken kann, sodass Integrity, Autheticity und Confidentiality erfüllt ist.

Frage 10

Wie kann A verifizieren, dass die Digitale Signatur von B tatsächlich B's Signatur ist?

Frage 11

Beschreibe die Struktur der Public Key Infrastructure (PKI).

Frage 12

Wofür kann man das Needham-Schröder Protokoll gebrauchen?

Frage 13

Ein Client C will eine Anfrage an einen Server S schicken. Das Needham-Schröder Protokoll wurde verwendet um die Authentifizierung von Clients zu implementieren. Skizziere den Ablauf bis C die Anfrage an den S schickt.

Frage 14

Wofür werden Nonces gebraucht?

Frage 15

Wie funktioniert der Login eines Clients bei Kerberos?

Frage 16

Ein Verteiltes System benutzt Kerberos. Skizziere den Ablauf, bis (inklusive) ein Client C seine Anfrage an den Server S schickt.

Antworten Kapitel 5

Antwort 1

Warum sind VS schwer zu schützen?

- Interfaces sind offen
- Pakete können abgefangen und manipuliert werden
- Sicherheitskritische Algorithmen sind open-source, Angreifer können also frei nach Schwachstellen suchen

Antwort 2

Welche Angriffsvektoren gibt es? Was können wir unternehmen um uns gegen sie zu schützen.

Siehe Angriffsmethoden.

Antwort 3

Was ist der Unterschied zwischen Computer Security und Communication Security?

- Computer Security: Kompromittierte Knoten finden und wiederherstellen
- Communication Security: Schutz von Kommunikations-Kanälen und kommunizierenden Entitäten.

Antwort 4

Welche Eigenschaften muss unser VS erfüllen um Communication Security zu gewährleisten?

- **Integrity**

  Nachrichten kommen genau so wie sie abgeschickt wurden beim Empfänger an.
- **Confidentiality**

  Keine unauthorisierte Person kann eine Nachricht lesen.
- **Authenticity**

  Es kann erkannt werden wenn eine Nachricht angekommen ist die von einer unauthorisierten Entität geschrieben wurde, bzw. es gibt eine zuverlässliche Möglichkeit zu beweisen, dass man tatsächlich der Sender einer Nachricht ist.
- **Availability**

  Das VS kann zuverlässig auf Anfragen reagieren und die Komponenten des VS können zuverlässig miteinander kommunizieren.

Antwort 5

Um ein VS zu schützen muss davon ausgegangen werden, dass der Angreifer gewisse Dinge tun kann um uns anzugreifen. Wovon müssen wir ausgehen? Was kann der Angreifer tun?

Der Angreifer kann jegliche Nachricht lesen und manipulieren sowie beliebige Nachrichten an beliebige Prozesse senden.

Antwort 6

Wofür kann Cryptography genutzt werden, nenne die Use-Cases und welche Technologien dafür genutzt werden.

- **Confidentiality**
  Vertraulichkeit von Nachrichten und persistenten Daten sicherstellen: Verschlüsseling
- **Authenticity**
  Authentizität von Nachrichten sicherstellen über Digitale Signaturen.
  Authentizität von Servern / Software-Modulen sicherstellen über Zertifikate.
- **Integrity**
  Integrität von Nachrichten sicherstellen: Verschlüsselte Checksummen, MAC-Codes

Antwort 7

Was ist der Unterschied zwischen Secret-Key Cryptography und Asymmetric Cryptography?

- Secret-Key (symmetric): Geheimnis (Schlüssel) ist beiden Parteien vorher bekannt und wird zum Ver- und Entschlüsseln benutzt. Die Ver- und Entschlüsseling Operationen der Secret-Key Kryptographie sind schneller als die der asymmetrische Verschlüsselung.
- Asymmetric: Die Daten werden mit dem Public Key des Empfängers verschlüsselt. Eine effiziente Entschlüsselung ist nur mit dem Private Key des Empfängers möglich.

Antwort 8

Kann man Secret-Key Cryptography für digitale Signaturen verwenden? Begründe.

Nein. Da der Schlüssel der für die Ver- und Entschlüsseln benutzt wird gleich ist, könnte der Prüfer der digitalen Signatur auch derjenige sein der die Nachricht unterschrieben hat. Die notwendigen Bedingungen für digitale Signaturen, *Non-re-usability* und *Non-repudiation*, können bei Verwendung von symmetrischen Schlüsseln nicht erfüllt werden.

Antwort 9

Beschreibe Step-by-Step wie A eine Nachricht M an B schicken kann, sodass Integrity, Autheticity und Confidentiality erfüllt ist.

1. Berechne eine Checksumme der Nachricht
2. Verschlüssele die Checksumme mit dem Private Key des Senders
3. Verschlüssele die Nachricht, zusammen mit der Checksumme und Zertifikat mit dem Public Key des Empfängers
4. Versende dieses Paket
5. Der Empfänger entschlüsselt das Paket mit seinem Private Key
6. Er berechnet die Checksumme der Nachricht, entschlüsselt die Checksumme und vergleicht.
7. Falls die Checksummen gleich sind prüft er auch ob der Public Key mit dem die Checksumme verschlüsselt wurde tatsächlich zum erwarteten Sender gehört indem er das mitgesendete Zertifikat validiert.

Antwort 10

Wie kann A verifizieren, dass die Digitale Signatur von B tatsächlich B's Signatur ist?

1. Er prüft das Zertifikat das der Sender mitgesandt hat

2. Zuerst wird der Public Key vom Sender validiert indem die Digitale Signatur der Certification Authority (CA) auf dem Zertifikat des Senders mit dem Public Key auf dem Zertifikat der CA validiert wird.
3. Dann wird die Digitale Signatur auf dem Zertifikat der CA validiert indem der Public Key der CA die diese Unterschrift geschrieben hat benutzt wird.
4. Dies geht rekursiv so weiter bis der Root-CA Public Key erfolgreich zum validieren benutzt wurde. Dann sind wird fertig.

Antwort 11

Beschreibe die Struktur der Public Key Infrastructure (PKI).

Die PKI ist hierarchisch aufgebaut. Es gibt das Root-CA, das die Wurzel der Struktur ist und untergeordnete Schichten von CA's die zwar auch Zertifikate überprüfen können, aber keine eigenen Zertifikate ausstellen düfen. Zusätzlich gibt es eine Registration Authority, die eine Verbindung zwischen einem Public Key und dem Besitzer des Public Keys herstellt. Außerdem gibt es einen Directory Service der Auskunft über zurückgezogene Zertifikate gibt und einen Time-Stamp Service der eine zuverlässig Ordnung von Ereignissen herstellt.

Antwort 12

Wofür kann man das Needham-Schröder Protokoll gebrauchen?

Man kann es benutzen um die Authenfizierung in **Intranets** zu implementieren, ohne Geheimnisse in Klartext über das Netzwerk zu schicken.

Antwort 13

Ein Client C will eine Anfrage an einen Server S schicken. Das Needham-Schröder Protokoll wurde verwendet um die Authentifizierung von Clients zu implementieren. Skizziere den Ablauf bis C die Anfrage an den S schickt.

Siehe Authentifizierung auf einem lokalen System.

Antwort 14

Wofür werden Nonces gebraucht?

Nonces zeigen, dass die Nachricht eine Antwort auch eine bestimmte vorhergehende Nachricht ist.

Antwort 15

Wie funktioniert der Login eines Clients bei Kerberos?

1. Zuerst sendet der Client eine Login-Anfrage an den Authentication Server (AS).
2. Der AS sendet eine mit dem Public Key des Clients verschlüsselte Nachricht an den Client. Diese Nachricht enthält zum einen den symmetrischen Schlüssel den der Client braucht um verschlüsselt mit dem Ziel-Server zu kommunizieren, zum anderen enthält die Nachricht ein Ticket für die Kommunikation mit dem Ziel-Server.
3. Der Client entschlüsselt die Nachricht, extrahiert das Ticket und den symmetrischen Schlüssel.
4. Mit dem Schlüssel und dem Ticket kann der Client nun mit dem Ziel-Server kommunizieren.

Antwort 16

Ein Verteiltes System benutzt Kerberos. Skizziere den Ablauf, bis (inklusive) ein Client C seine Anfrage an den Server S schickt.

Siehe Ablauf von Kerberos