Verteilte Systeme

Oktober - Dezember 2022

5. Vorlesung – 07.11.2022

Kurs: TINF20AI1

Dozent: Tobias Schmitt, M.Eng.

Kontakt: d228143@
student.dhbw-mannheim.de

Wiederholungsfragen

- Inwiefern kann Broadcast zum Finden einer Ressource genutzt werden? Und welche Grenzen gibt es bei diesem Verfahren?

- Welche Ansätze hinsichtlich mobiler Entitäten kennen Sie?

- Welche Ansätze hinsichtlich dem Auffinden von Entitäten kennen Sie in Peer-to-Peer-Systemen?

- Welche Anwendungsbereiche kennen Sie für hierarchische Benennungsschema?

- Wie unterscheiden sich TAI und UTC?

Themenüberblick

- **Synchronisierung**
  - **Logische Uhren**
  - Gegenseitiger Ausschluss
  - Wahlalgorithmen
- Konsistenz und Replikation

Logische Uhren

- Beobachtung
  - Uhrzeitsynchronisierung zwar möglich, aber kein absolutes Muss
  - Bei nicht-wechselwirkenden Prozessen: fehlende Synchronisierung nicht beobachtbar und kein Resultat von Fehlern
  - Aber wichtig: Einigung auf eine Reihenfolge von Ereignissen

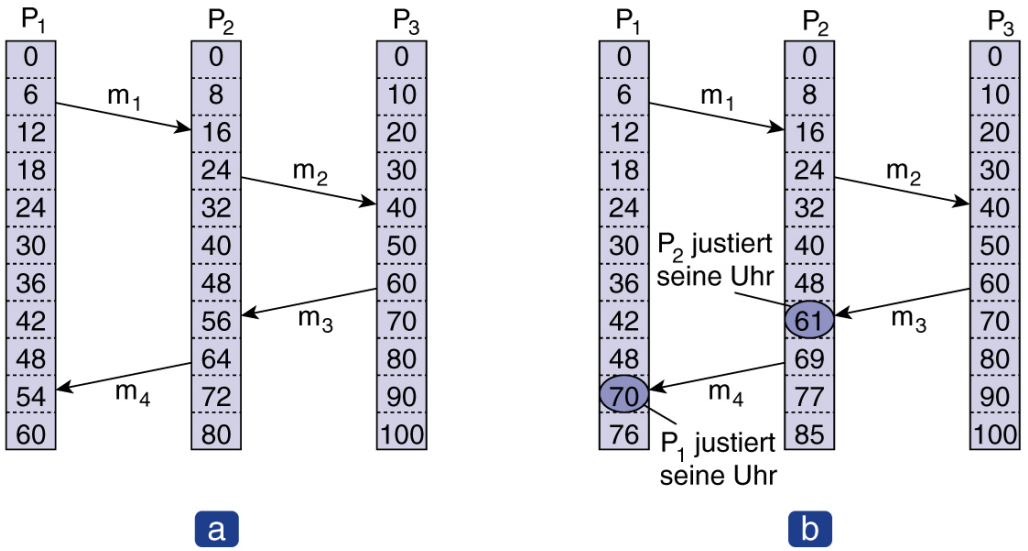
- Frage: Wie kann man die Reihenfolge von Ereignissen in verteilten Systemen sicherstellen?

Logische Uhren

- **Lamport-Zeit / Logische Uhr von Lamport**
  - Bezeichnung: a→b
  - Bedeutung: „a passiert vor b“ („*happens-before*“)
  - Beobachtung:
    - Im selben Prozess passiert a vor b, dann a→b
    - a entspricht Senden einer Nachricht, b entspricht Empfang einer Nachricht, dann a→b
  - Hinweis: Happens-before-Relation ist transitiv
  - Wenn a→b und b→c, dann gilt auch a→c
  - Einführung eines Zeitwertes C(a),
  - wenn a→b, dann C(a) < C(b)

Logische Uhren

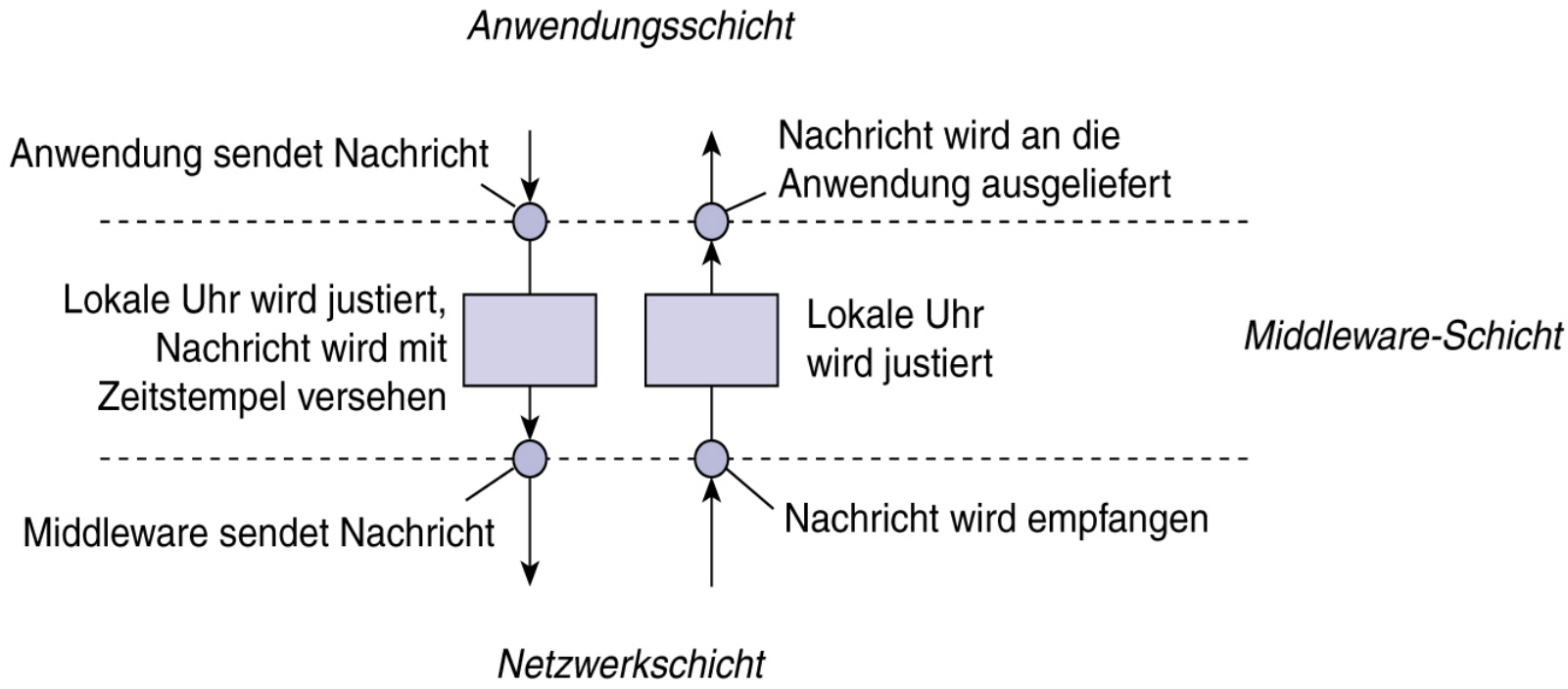
- **Lamport-Zeit / Logische Uhr von Lamport** – Teil 2
  - Bedingungen: Ist a ein Ereignis in Prozess P_i und b ein Ereignis in Prozess P_j, dann gilt a→b wenn
  - (i) $C_i(a) < C_j(b)$ oder
  - (ii) $C_i(a) = C_j(b)$ und $P_i < P_j$
  - (Berücksichtigung der
  - Prozessnummern)
  - Prinzip: Anpassung der
  - jeweiligen Uhren



(a) Drei Prozesse, von denen jeder eine eigene Uhr hat. Die Uhren laufen mit unterschiedlichen Geschwindigkeiten.
(b) Der Algorithmus von Lamport korrigiert diese Uhren.

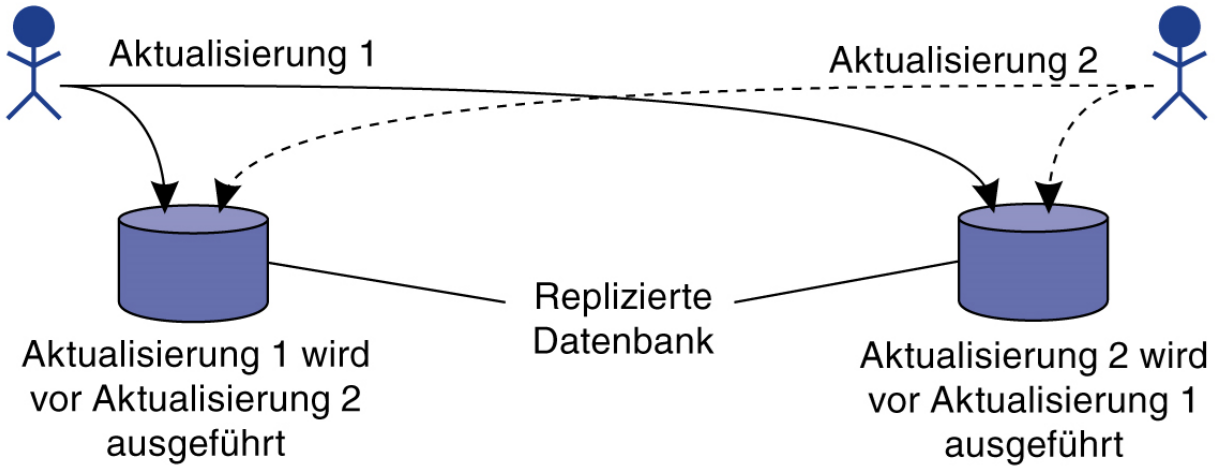
Logische Uhren

- **Lamport-Zeit / Logische Uhr von Lamport** – Teil 3
  - Lokalisierung der logischen Uhren von Lamport in der Middleware-Schicht



Logische Uhren

- Anwendung: **Vollständig geordnetes Multicasting**
- Beispiel: Kontoguthaben 1000€
  - Aktualisierung 1: Einzahlung von 100€
  - Aktualisierung 2: Gutschrift von Zinsen zu 1%
- Problem: Reihenfolge muss auf allen Datenbanken gleich sein



Logische Uhren

- Anwendung: **Vollständig geordnetes Multicasting**
- Annahmen:
  - Keine Nachrichten gehen verloren
  - Eintreffende Nachrichten kommen auch in der Sendereihenfolge eines Sender an

Logische Uhren

- Anwendung: **Vollständig geordnetes Multicasting**
- Lösung:
  - Jede Nachricht mit aktuellem (logischen) Zeitstempel des Senders versehen an alle Knoten (inklusive sich selber)
  - Empfänger
    - Einsortierung in Warteschlage gemäß dem Zeitstempel
    - Senden einer Bestätigung via Multicast
  - Resultat: Alle Prozesse haben gleiche Kopie der lokalen Warteschlange.

Logische Uhren

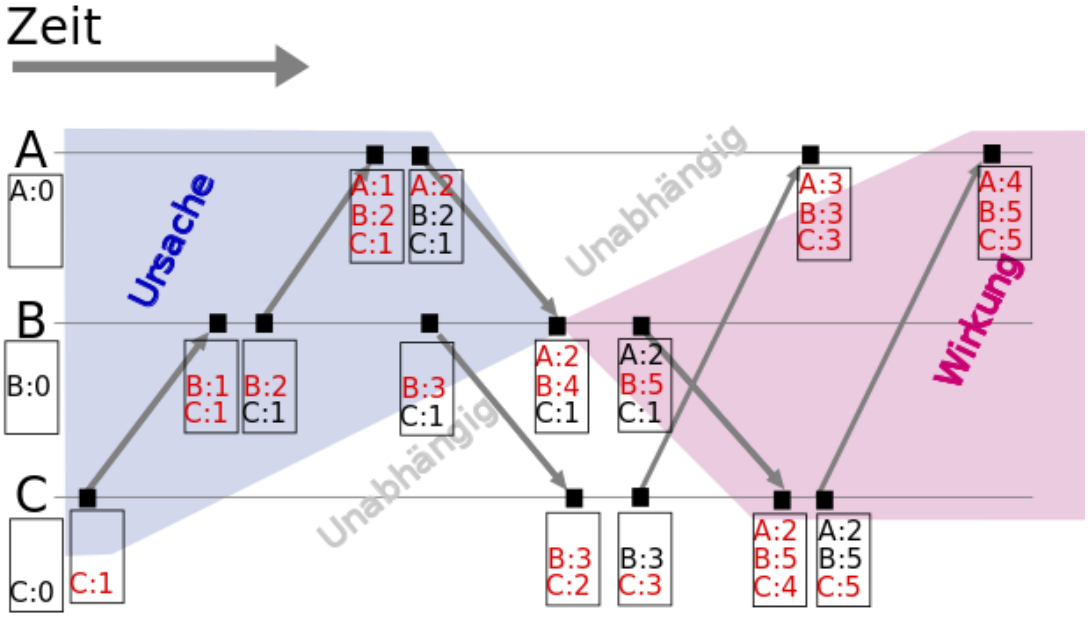
- **Vektoruhren**
  - Bei logischer Uhr von Lamport → Vollständige Ordnung (gemäß der Bedingungen)
  - Nachteil bei Lamport: Keine Aussage über Kausalität
  - Ziel:
    - Kausalzuordnung möglich
    - Nebenläufigkeit von Ereignissen ermittelbar

Logische Uhren

.Vektoruhren – Teil 2

- Idee: Vektor mit Zeit für jeden Prozess wird geführt
- Aktualisierung der eigenen Zeit
- Versenden des gesamten Vektors

- Anwendbarkeit:
- Monitor- und
- Debugsysteme



(Quelle: https://de.wikipedia.org/wiki/Vektoruhr#/media/Datei:Vektoruhren.svg)

Themenüberblick

- **Synchronisierung**
  - Logische Uhren
  - **Gegenseitiger Ausschluss**
  - Wahlalgorithmen
- Konsistenz und Replikation

Gegenseitiger Ausschluss

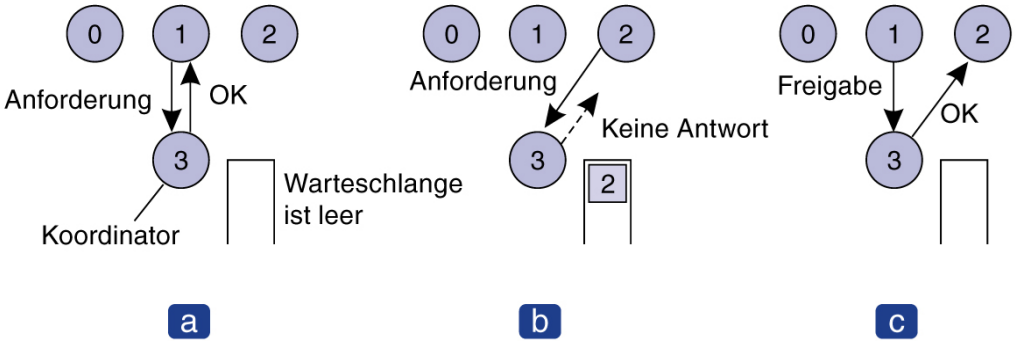
- Fragen:
  - Was verstehen Sie unter einer Race-Condition? Und was mein der kritische Bereich in diesem Zusammenhang? (Nennen Sie ggf. ein Beispiel für eine RC.)
  - Was verstehen Sie unter wechselseitigem Ausschluss?
  - Wie lässt sich der wechselseitige Ausschluss realisieren?
  - (Was war die Lösung aus der Betriebssysteme-Vorlesung?)
  - Was verstehen Sie unter Verhungern eines Prozesses?
  - Was verstehen Sie unter einem Deadlock?

Gegenseitiger Ausschluss

- Wünsche hinsichtlich des gegenseitigen Ausschlusses
  - Keine Deadlocks
  - Kein Verhungern
  - Fairness
  - Fehlertoleranz
- Lösungsansätze
  - Token-basierte Ansätze
  - Quorum-basierte Ansätze (Mehrheits-basiert)
  - Nicht-Token-basierte Ansätze (Laufzeit-basiert)

Gegenseitiger Ausschluss

- **Zentralisierter Algorithmus**
  - Einführung eines Koordinators
  - Prozess-Anfrage an Koordinator
  - Wenn kein Prozess Ressource belegt
    - Erteilung der Freigabe
  - Wenn aber ein Prozess Ressource belegt
    - Einsortierung in Warteschlange
    - Mit Freigabe der Ressource → Nachricht an wartenden Anfrageprozess



Gegenseitiger Ausschluss

- **Dezentraler Algorithmus**
  - Erweiterung des zentralen Koordinators
  - Ressourcen werden n-mal repliziert und jede Kopie erhält eigenen Koordinator
  - Zugriff auf Ressource verlangt Mehrheitsvotum

$$m > n/2 \text{ Koordinatoren}$$

Gegenseitiger Ausschluss

- **Dezentraler Algorithmus**
  - Vorteil:
    - Weniger anfällig bzgl. Ausfällen eines Koordinators
    - Algorithmus funktioniert auch, wenn sich der Koordinator regelmäßig zurücksetzt (alle bisherigen Freigaben vergisst)
  - Nachteil:
    - Problematisch, wenn viele Knoten auf Ressource zugreifen wollen (ggf. bekommt keiner die Mehrheit)

Gegenseitiger Ausschluss

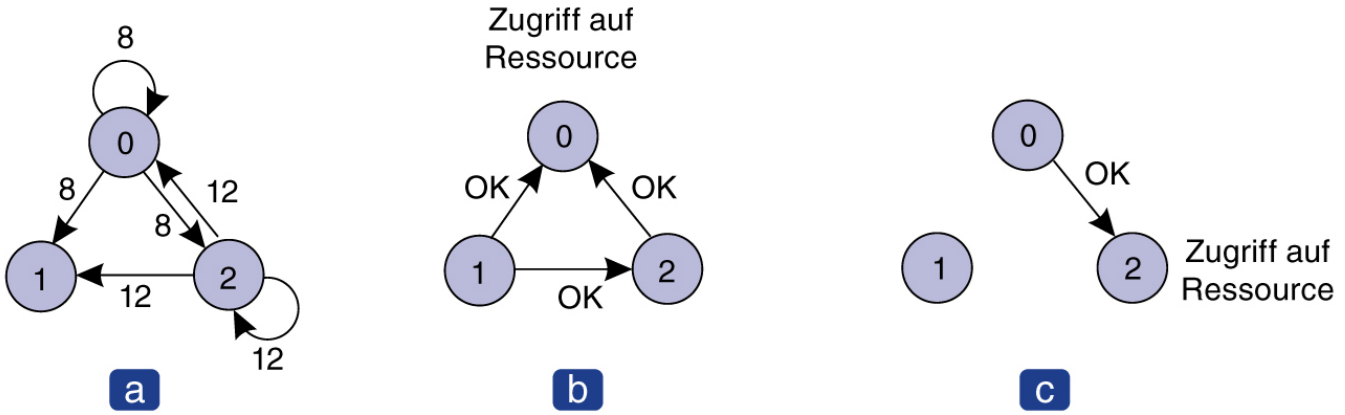
- **Verteilter Algorithmus**
  - Ausgangspunkt: Verteiltes System mit vollständiger Ordnung aller Ereignisse im System (vgl. logische Uhr von Lamport)
  - Vorgehensweise eines an Ressource interessierten Prozesses
    - Erstellt Nachricht mit Ressourcenbezeichnung, Prozessnummer und aktueller (logischer) Zeit
    - Senden der Nachricht an alle (Annahme: kein Verlust von Nachrichten)
    - Warten auf die Erlaubnis von allen

Gegenseitiger Ausschluss

- **Verteilter Algorithmus**
  - Reaktion der Empfänger
    - Kein Zugriff auf Ressource, dann OK an Sender
    - Aktueller Zugriff, dann Schweigen und Einsortierung in Warteschlange (OK an Sender nur nach Freigabe der Ressource)
    - Bei geplantem Zugriff in Zukunft: Zeitstempelvergleich (OK an Sender oder Einsortierung in Warteschange)

Gegenseitiger Ausschluss

.Verteilter Algorithmus – Teil 2



(a) Zwei Prozesse möchten gleichzeitig auf eine gemeinsam genutzte Ressource zugreifen.

(b) Prozess 0 hat den niedrigeren Zeitstempel, sodass er gewinnt.

(c) Wenn Prozess 0 fertig ist, sendet er auch ein *OK*, sodass 2 weitermachen kann.

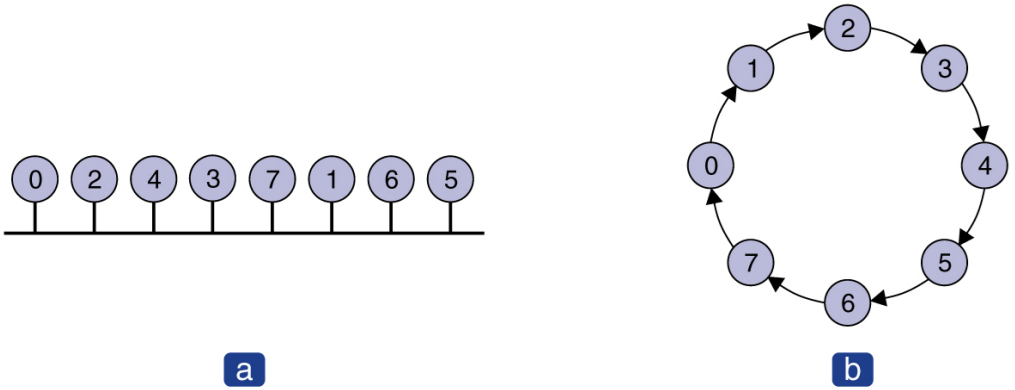
.Probleme:

- Nicht nur singuläre Fehlstelle (SPOF, Single Point of Failure), sondern n Fehlstellen
- Absturz eines Prozess: Interpretation als fehlende Freigabe
  - Verbesserung durch Arbeit mit Empfangsbestätigungen
- Arbeit mit Multicast-Kommunikation oder alle Gruppenmitgliedschaften müssen bekannt sein

Gegenseitiger Ausschluss

- **Token-Ring-Algorithmus**
  - Erstellung eines logischen Rings durch Software
    - Beispiel: numerische Ordnung der Netzwerkadresse
  - 1 Token im Ring → Weitergabe des Tokens nach der Prüfung, ob Ressource benötigt wird
  - Zirkulierung des Tokens im Ring
  - Nachteile:
    - Verlust des Tokens im System (Problem langes Ausbleiben des Tokens ≠ Verlust des Tokens)
    - Absturz eines Prozesses
      - Kompensation, wenn mit Empfangsbestätigungen gearbeitet wird

Gegenseitiger Ausschluss



(a) Eine ungeordnete Gruppe von Prozessen in einem Netzwerk;

(b) ein mit Software erstellter logischer Ring

Gegenseitiger Ausschluss

- **Vergleich der Algorithmen**

Algorithmus	Nachrichten pro Eintritt/ Austritt	Verzögerung vor dem Eintritt (in Nachrichtenzeiten)	Probleme
Zentralisiert	3	2	Absturz des Koordinators
Dezentral	3mk, k = 1,2,...	2m	Verhungern, niedrige Leistungsfähigkeit
Verteilt	2(n − 1)	2(n − 1)	Absturz irgendeines Prozesses
Token Ring	1 bis ∞	0 bis n − 1	Verlorenes Token, Absturz eines Prozesses

Themenüberblick

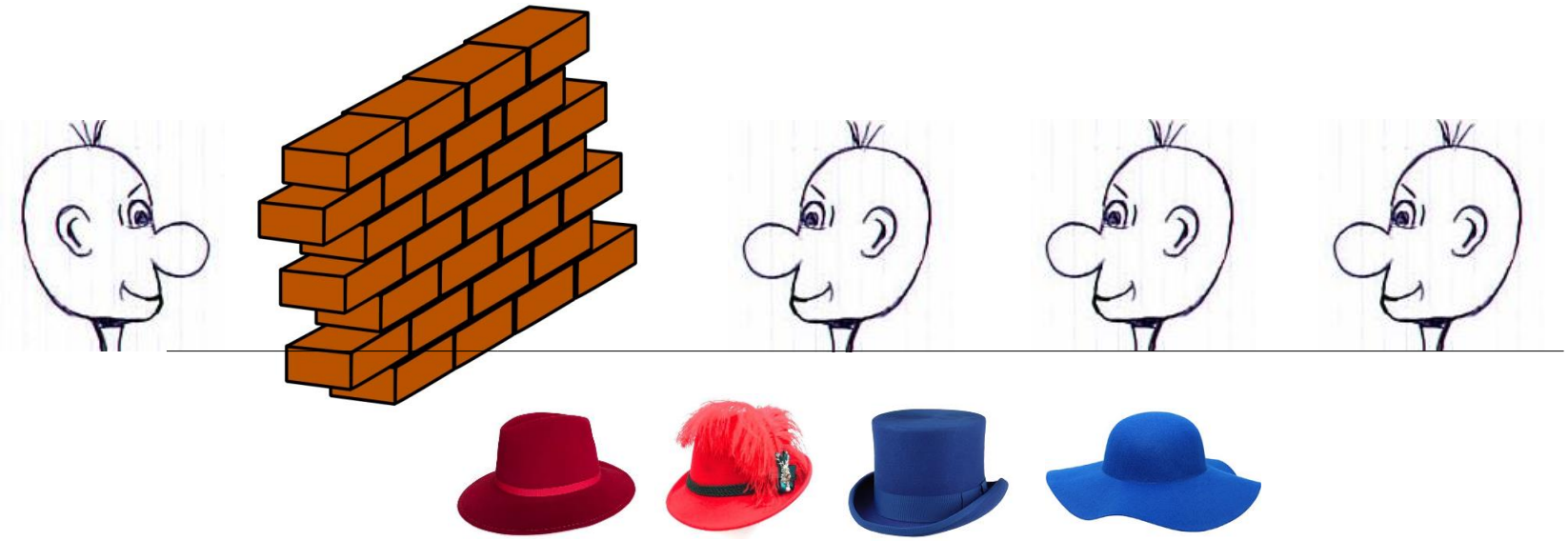
- **Synchronisierung**
  - Logische Uhren
  - Gegenseitiger Ausschluss
  - **Wahlalgorithmen**
- Konsistenz und Replikation

Interludium - Knobelaufgabe

- 4 Entwickler wegen Unkenntnis bzgl. verteilter Systeme vom Chef bestraft
  - Im Sandkasten der Stadt bis auf den Kopf im Sand vergraben
  - … Anordnung siehe unten
  - Chef verteilt 2 rote und 2 blaue Hüte
  - Für die Rettung aller: **Nur einer sagt seine richtige Hutfarbe!**
  - Bedingungen:
    - Keine Absprache oder sonstige Zeichen erlaubt
    - Chef gibt nur eine Minute Zeit.
  - Aufgabe: Wer rettet die armen Entwickler? Und was (bzw. welche Hutfarbe) muss er sagen?

Hinweis: Es scheint keine Sonne, so dass Sie den Schattenwurf nicht berücksichtigen können.

Interludium - Knobelaufgabe



Hinweis: Es scheint keine Sonne, so dass Sie den Schattenwurf nicht berücksichtigen können.

Herkömmliche Wahlalgorithmen

- Problemstellung
  - Manche verteilte Algorithmen verlangen **einen** Prozess als Koordinator
  - Bei Gleichheit aller Prozesse: Wer wird als Koordinator ausgewählt?
- Annahmen:
  - Jeder Prozess hat eindeutige Nummer (z.B. Netzwerkadresse)
  - Jeder Prozess kennt die Nummern aller anderen.
  - Unwissenheit über Aktivität der anderen Prozesse

Herkömmliche Wahlalgorithmen

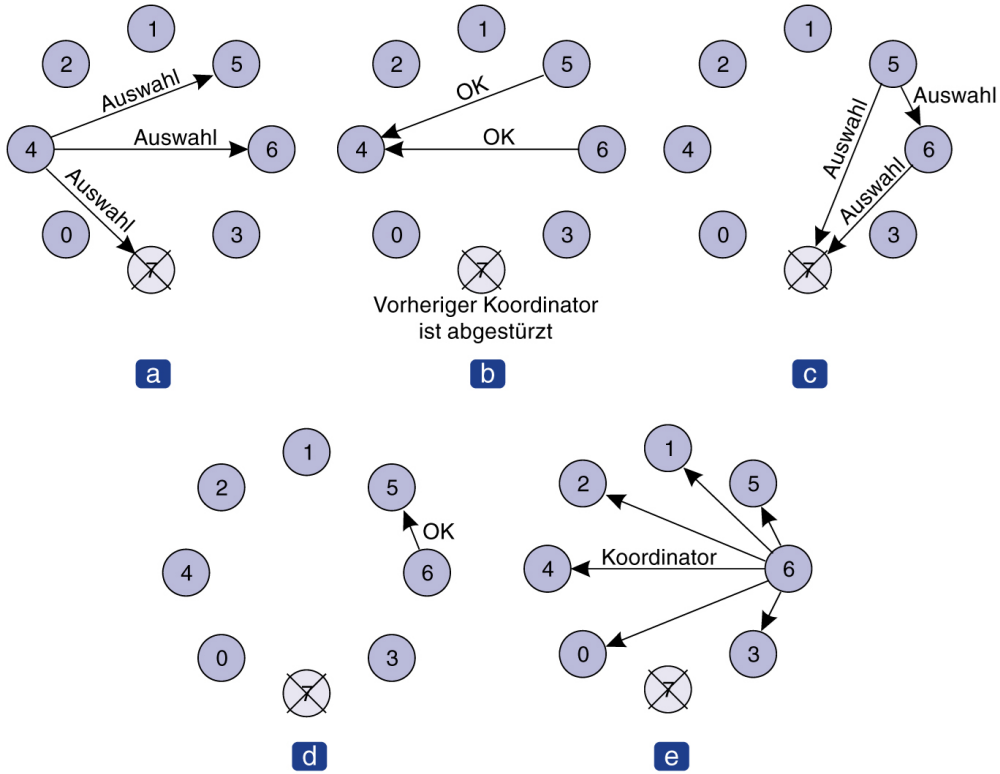
- Ziel:
  - Nach Abschluss des Wahlalgorithmus sind sich alle einig, wer der neue Koordinator ist.
- Aufgabe:
  - **Wie lässt sich bei vertretbarem Aufwand der aktive Prozess mit der höchsten Nummer ermitteln und allen anderen Prozessen kommunizieren?**

Herkömmliche Wahlalgorithmen

- **Der Bully-Algorithmus** (Bezug darauf, dass der „größte“ Prozess gewinnt.)
  - Prozess P stellt fest, dass Koordinator nicht mehr antwortet
  - P führt die Wahl durch
    - P sendet WAHL-Nachricht an alle Prozesse mit höheren Nummern
    - Variante 1: Keiner antwortet => P gewinnt die Wahl
    - Variante 2: Antwort von Prozess Q, der die Wahl dann übernimmt … und P schweigt.
  - Neuer Koordinator sendet Nachricht an alle über den Gewinn der Wahl.
  - Bei Respawn des alten Koordinators:
    - Abhalten einer neuen Wahl (initiiert durch neuen Prozess)

Herkömmliche Wahlalgorithmen

- **Der Bully-Algorithmus** – Teil 2



(a) Prozess 4 führt eine Wahl durch.

(b) Die Prozesse 5 und 6 antworten und teilen 4 mit, dass er aufhören soll.

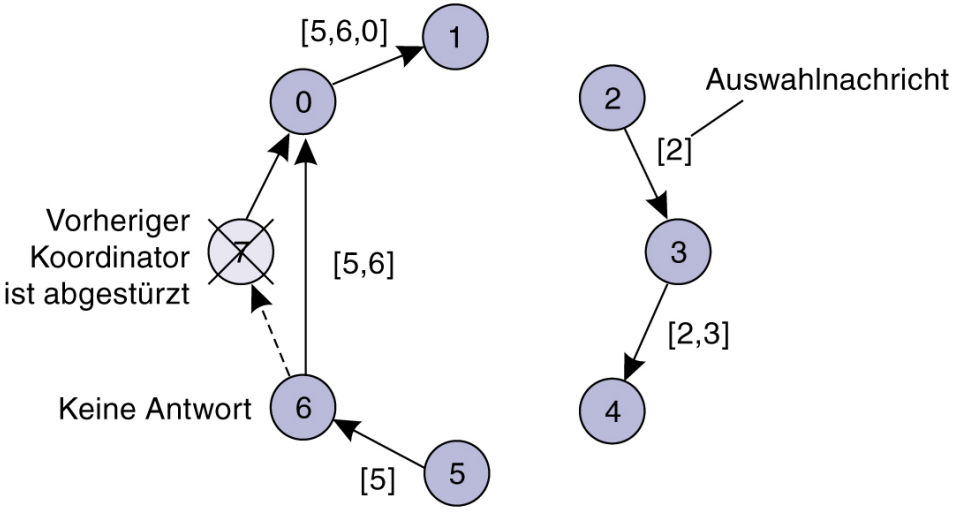
(c) Nun halten die Prozesse 5 und 6 beide eine Wahl a

(d) Prozess 6 sagt 5, dass er aufhören soll.

(e) Prozess 6 gewinnt und sagt allen Bescheid.

Herkömmliche Wahlalgorithmen

- **Ein Ringalgorithmus (aber ohne Token)**
  - Annahme: Nachfolger für jeden Prozess bekannt (aufgrund physischer oder logischer Anordnung)
  - Prozess P stellt fest, dass Koordinator nicht mehr antwortet
  - P startet die Wahl
    - Erstellung einer Nachricht, die Prozessnummer von P enthält

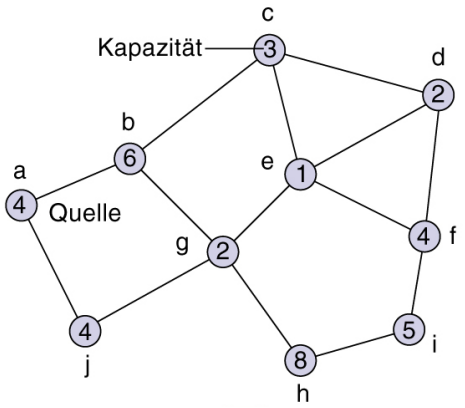


Herkömmliche Wahlalgorithmen

- **Ein Ringalgorithmus (aber ohne Token)**
  - Nachrichtenweiterleitung an Nachfolger
  - Variante 1: Nachfolger antwortet, Nachfolger wiederholt Eintragung seiner Prozessnummer und Weiterleitung
  - Variante 2: Nachfolger antwortet nicht, Weiterleitung an übernächstes Mitglied
  - P erreicht die Nachricht erneut
    - Bestimmung des Koordinators
    - (Prozess mit der höchsten Nummer)
    - Nachrichtentyp ändern
    - Eigene Prozessnummer löschen
    - Weiterleitung an Nachfolger

Wahlalgorithmen in drahtlosen Umgebungen

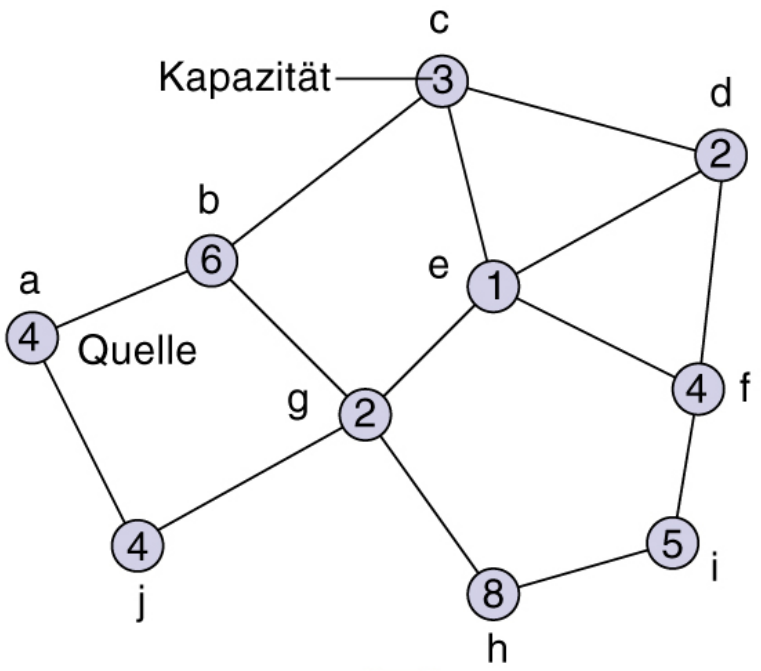
- Herkömmliche Wahlalgorithmen in drahtlosen Umgebungen nicht realistisch, da
  - Nachrichtenübertragung nicht zuverlässig
  - Topologie veränderbar



- Problemstellung:
  - Ad-Hoc-Netzwerk (aber Veränderlichkeit ignorieren)
- Ziel: Auswahl des besten Knotens als „Anführers“ hinsichtlich
  - Lebensdauer der Batterie
  - Ressourcenverfügbarkeit oder anderer Kriterien

Wahlalgorithmen in drahtlosen Umgebungen

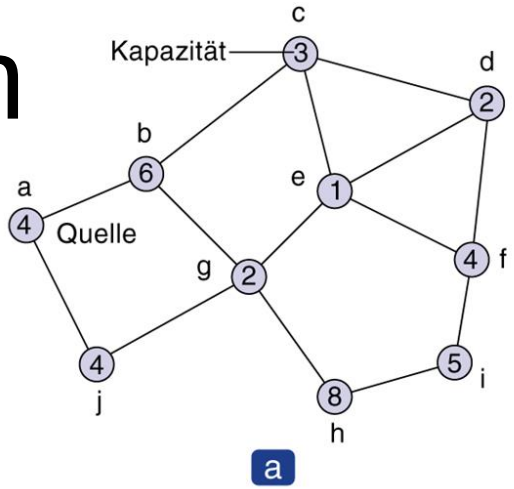
- Frage: Wie könnte der Wahlalgorithmus hier aussehen?

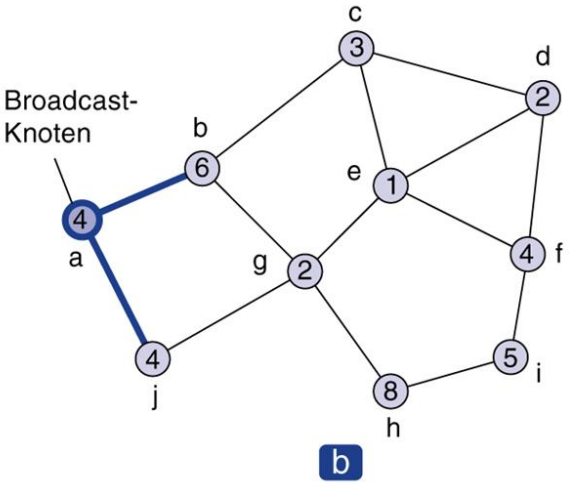


Wahlalgorithmen in drahtlosen Umgebungen

- Lösungsidee
  - Knoten Q (Quelle) sendet Wahl-Nachricht an alle unmittelbaren Nachbarn
  - Knoten P - Erhalt der Nachricht zum 1ste Mal
    - Weiterleitung an alle Nachbarn außer Eingangs-/Elternknoten (und noch keine Rückantwort)
  - Knoten P - Erhalt der Nachricht erneut
    - Sofortige Rückmeldung an Sender
  - Knoten R ist Blattknoten, wenn Rückmeldung aller Nachbarn, dass Wahlnachricht bereits erhalten wurde
    - Rückmeldung an Eltern samt Zusatzinfos / Statusinfos
  - Zusatzinfos / Statusinfos werden gesammelt und bis zu Q weitergereicht
    - Knoten Q bestimmt „besten“ Knoten

Wahlalgorithmen in drahtlosen Umgebungen



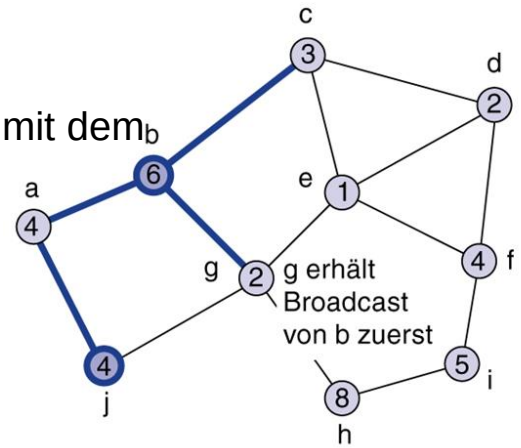


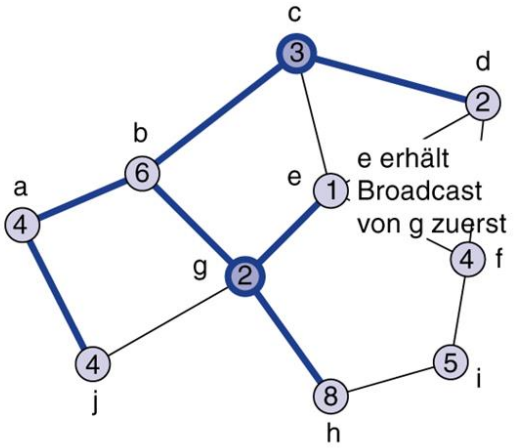
Wahlalgorithmus in einem drahtlosen Netzwerk mit dem Knoten *a* als Quelle;

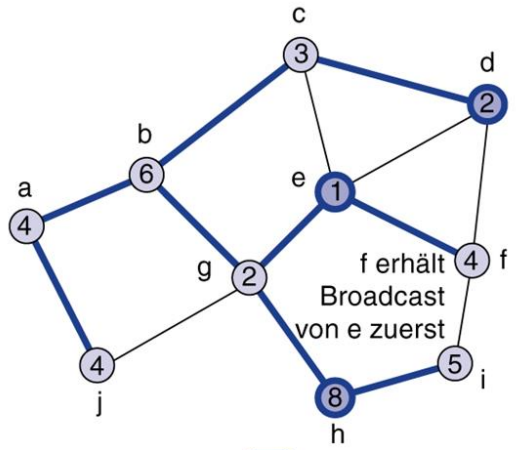
(a) ursprüngliches Netzwerk;

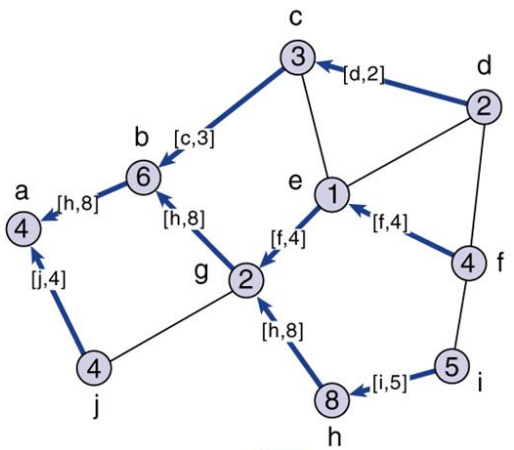
(b)–(e) die Phase der Baumerstellung (der letzte Broadcast durch die Knoten *f* und *i* ist nicht gezeigt);

(f) Mitteilen des besten Knotens an die Quelle







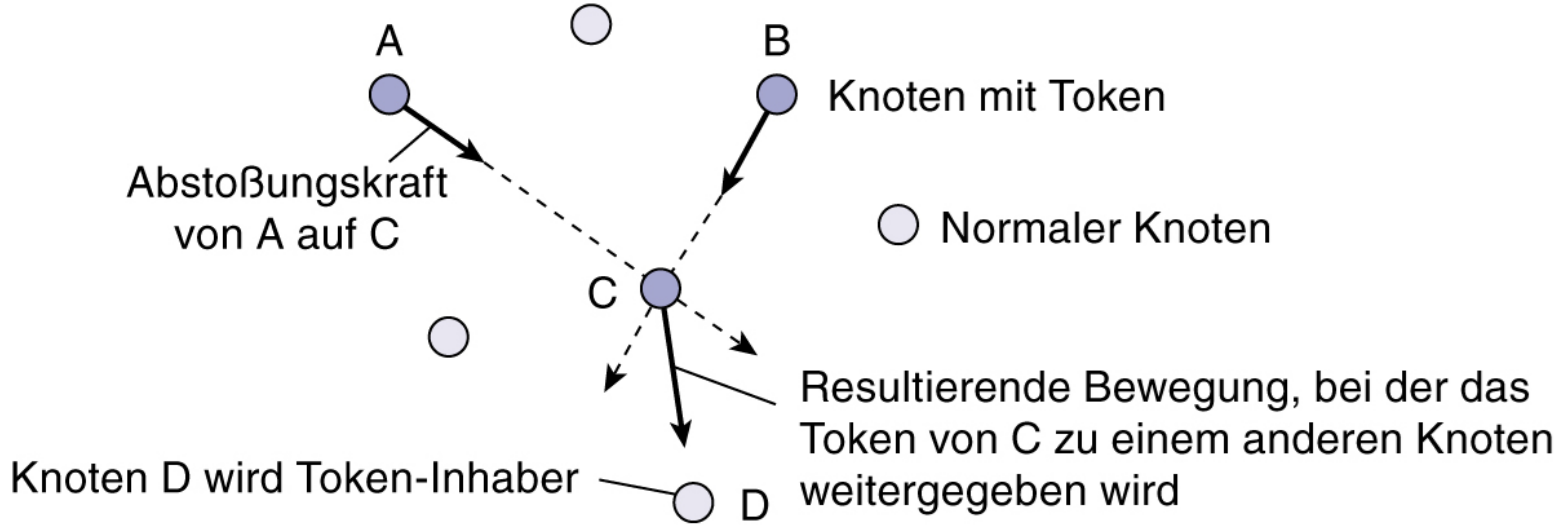


Wahlalgorithmen in großmaßstäblichen Systemen

- Bisherige Einschränkungen
  - Relativ kleine verteilte Systeme
  - Auswahl eines Knotes
- Problem bei Peer-to-Peer-Systemen
  - Mehrere Superpeers möglich
- Bedingung an die Superpeers
  - Zugang normaler Knoten zu Superpeers mit niedriger Latenz
  - Superpeerverteilung gleichmäßig über das Overlay-Netzwerk
  - Anzahl der Superpeers abhängig von Gesamtzahl der Knoten
  - Superpeer dient nur für vorgegebene Anzahl an Knoten

Wahlalgorithmen in großmaßstäblichen Systemen

- Token-Ansatz
  - N Token werden zufällig auf N Knoten verteilt
  - Kein Token kann mehr als einen Knoten haben
  - Jeder Token stellt abstoßende Kraft für andere Token dar
  - Ziel: Gleichmäßige Verteilung der Tokens im geometrischen Raum



Themenüberblick

- Synchronisierung
  - Logische Uhren
  - Gegenseitiger Ausschluss
  - Wahlalgorithmen
- **Interludium - Praxisbeispiel**
  - **Problemstellung: Race-Condition in einer Web-Anwendung**
- Konsistenz und Replikation

Interludium – Race-Condition in einer Web-Anwendung

- Hintergrund – ERP-Software (Enterprice-Ressource-Planning) als Web-Anwendung
  - Basistechnologien: PHP, MySQL, Javascript, HTML, CSS
  - Client-Server-Architektur
  - Kontext: Auftragsmodul – Änderung von Auftragspositionen
    - Artikelinfos: Artikelnr., Menge, Preis
    - Kundeninfos: Kundenart.nr., Kundenpos.nr., Kundenartikelbezeichung
    - Termindaten: Wunschtermin, Liefertermin
    - Fertigungsinfos: Produktionsbeginn, Sollzeit, Status, …
  - Hinweis: Alle Auftragspositionsdaten in einer Tabelle

Interludium – Race-Condition in einer Web-Anwendung

- Problem – Auftragsbesprechung
  - Absprache mehrere Abteilungen zu Auftragspositionen
  - Arbeit mit 2 Rechnern, Änderungen eines Datensatzes von 2 Clients
  - Ablauf:
    - Client A ruft Formular auf (Datensatz zum Stand 1)
    - Client B ruft Formular auf (Datensatz zum Stand 1)
    - Client A speichert seine Daten (neuer Stand 2)
    - Client B möchte seine Daten abspeichern (neuer Stand 3?)
  - Achtung: Race-Condition – Stand 2 würde überschrieben werden

- **Aufgabe:** Überlegen Sie sich ein mögliches Vorgehen um diese Situation zu lösen.

Interludium – Race-Condition in einer Web-Anwendung

- Realisierte Lösung – Schritt 1
  - Historisierung der Datensätze
    - Kein Löschen eines Datensatzes
    - Pflege von 3 Datenbankfeldern
      - Datum der Änderung (Ab wann war der Datensatz gültig?)
      - PersonenID des Änderers (Wer hat die Änderung veranlasst?)
      - Datum der Deaktivierung (Bis wann ist der Datensatz gültig?)
      - (Default-Wert = 0 und kennzeichnet aktiven Datensatz)
    - Vorteil: Änderungshistorie möglich
      - Ziel: Wer hat welche Änderungen veranlasst?

Interludium – Race-Condition in einer Web-Anwendung

- Realisierte Lösung – Schritt 2
  - Idee: Datensatzdatum mitsenden und Vergleichsansicht
  - Ablauf:
    - Client A ruft Formular auf (Datensatz zum Stand 1)
    - Client B ruft Formular auf (Datensatz zum Stand 1)
    - Client A sendet seine Daten (neuer Stand 2, mit Referenz auf Stand 1)
      - Server sollte Tabellen sperren und speichert die neuen Daten
    - Client B sendet seine Daten (neuer Stand 3, mit Referenz auf Stand 1)
      - Server stellt fest, dass bereits Stand 2 vorliegt ($T_2 > T_1$)
    - Client B erhält Differenzansicht
      - Client B wählt gültige Änderungen aus
      - Übergabe sämtlicher Formulardaten + Referenz auf Stand 2
      - Server speichert Daten (außer Stand 2 ist veraltet)

Themenüberblick

- Synchronisierung
- Interludium - Praxisbeispiel
- **Konsistenz und Replikation**
  - Datenzentrierte Konsistenzmodelle
  - ...

Konsistenz und Replikation

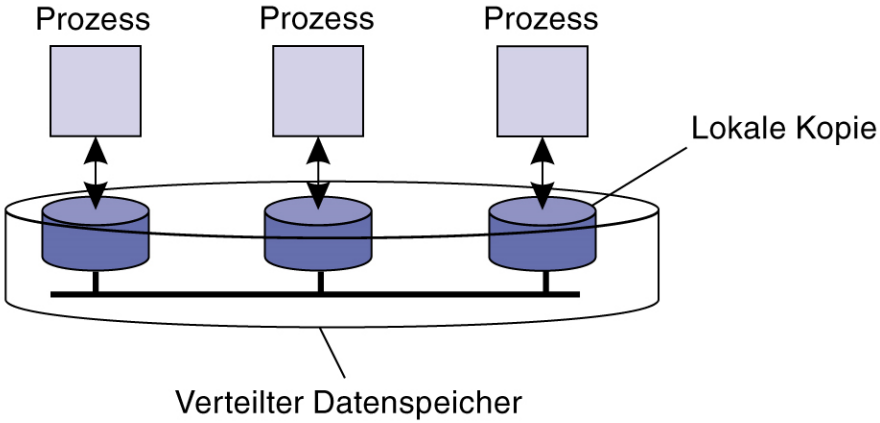
- Welche Gründe kann es für Replikation geben?
- Was kann gegen das Replizieren von Daten sprechen?
- Was bedeutet Konsistenz? Welche Vorstellung verbinden Sie damit?
- Ist Konsistenz messbar?
- Was könnten Sie sich Vorstellen, was datenzentrierte und clientzentrierte Konsistenz bedeutet?

Konsistenz und Replikation

- Gründe für Replikation
  - Verlässlichkeit
    - Im Fehlerfall – Umschalten auf anderes Replikat
    - Schutz vor beschädigten Daten
  - Systemleistung
    - Zahlenmäßige Skalierung → Zugriff von mehr Prozessen
    - Geografische Skalierung → Verkürzung der Datenzugriffszeit

- Gegenindikationen
  - Replikate aktuell zu halten bedarf Bandbreite
  - Konsistenzprobleme

Datenzentrierte Konsistenzmodelle



- Modell des logischen Datenspeichers
- Konsistenzmodell
  - Vertrag zwischen Prozessen und Datenspeicher
  - Korrektheit des Speichers, solange Einhaltung gewisser Regeln
  - Datenspeicher spezifiziert genau, was das Ergebnis von Lese- und Schreiboperationen im Rahmen der parallelen Ausführung sein wird

- Idee: Leseoperation liefert Resultat des letzten Schreibvorganges
- Problem: Festlegung schwierig bei Fehlen einer globalen Uhr
  - Modelle schränken Rückgabemöglichkeiten von Leseoperationen ein
- Aber: Je stärker die Konsistenzanforderung, desto höher der Aufwand für die Aktualisierung

Datenzentrierte Konsistenzmodelle

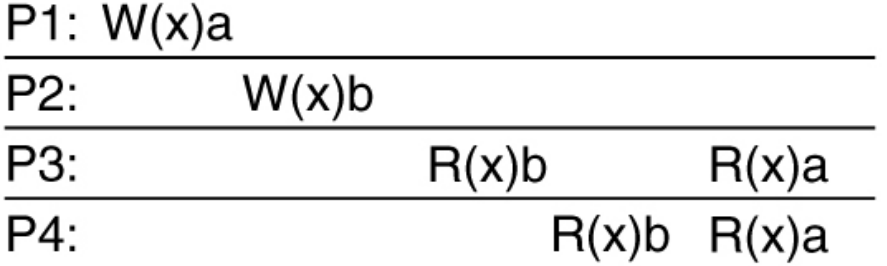
- Definition von Inkonsistenzen im Rahmen stufenloser Konsistenz:
- Abweichung von Replikaten hinsichtlich
  - numerischer Werte
    - z.B. Aktienkurse → Definition einer absoluten oder relativen numerischen Abweichung
    - Oder auch die Anzahl auf ein Replikat angewendete Aktualisierungen
  - Veralterungsgrad
    - Bezug auf den Zeitpunkt der letzten Aktualisierung
    - z.B. Daten vom Wetterbericht veralten nicht innerhalb weniger Stunden
  - Reihenfolge von Aktualisierungsoperationen
    - z.B. nur vorläufige Aktualisierung bis Zustimmung hinsichtlich der Reihenfolge aller Replikate → ggf. Zurücknahme der Aktualisierungen

Datenzentrierte Konsistenzmodelle

- Messung von Inkonsistenzen im Rahmen stufenloser Konsistenz
  - Definition einer Konsistenzeinheit (Consistency Unit) – kurz „Conit“
  - Beispiele für fein- und grobkörnige Conits:
    - Aufzeichnung hinsichtlich einer Aktie
    - Einzelner Wetterbericht
    - Komplette Datenbank
  - Conit-Größe
    - Kleinere Conits → mehr Verwaltung
    - Größere Conits → schneller im Inkonsistenz-Zustand
  - Wunsch: Klare Angabe der Konsistenzanforderung

Konsistente Anordnung von Operationen

- Sequentielle Konsistenz
  - Das Ergebnis jeder Ausführung ist dasselbe, als wären die Operationen von allen Prozessen in einer sequentiellen Reihenfolge ausgeführt worden und die Operationen jedes einzelnen Prozesses erscheinen in dieser Sequenz in der von seinem Programm vorgegebenen Reihenfolge.
  - Alle Prozesse beobachten dieselbe Verzahnung und jede Verzahnung von Operationen ist gültig, solange alle Prozesse die selbe Verzahnung sehen.
  - Hinweis: Keine Aussage über die Zeit.
  - Beispiele:



Erklärung:
P1: W(x)a → Prozess 1 schreibt in die Variable x den Wert a.
P2: R(x)b → Prozess 2 liest die Variable x und erhält den Wert b.

Konsistente Anordnung von Operationen

- Sequentielle Konsistenz
  - Weiteres Beispiel:
    - Operationen von 3 gleichzeitig ausgeführten Prozessen (Hinweis: print(a,b) meint Ausgabe von a und b.)

Prozess P1	Prozess P2	Prozess P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

    - Ausführung ergibt 6-Bit-Zeichenkette
    - Nicht alle 6-Bit-Zeichenketten sind im Rahmen der sequenziellen Konsistenz erlaubt! … siehe folgende Beispiele:

x ← 1;	x ← 1;	y ← 1;	y ← 1;
print(y, z);	y ← 1;	z ← 1;	x ← 1;
y ← 1;	print(x, z);	print(x, y);	z ← 1;
print(x, z);	print(y, z);	print(x, z);	print(x, z);
z ← 1;	z ← 1;	x ← 1;	print(y, z);
print(x, y);	print(x, y);	print(y, z);	print(x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111

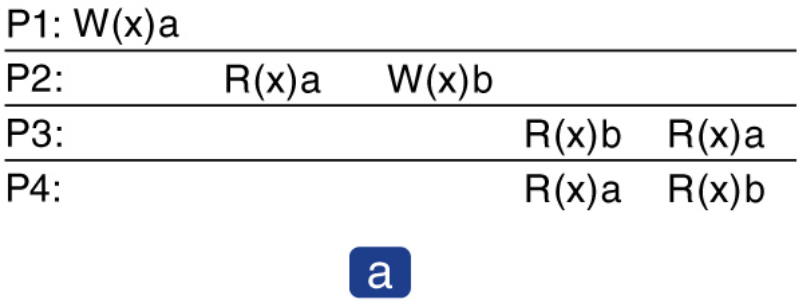
Konsistente Anordnung von Operationen

- Kausale Konsistenz
  - Schreibvorgänge, die möglicherweise in kausaler Beziehung zueinander stehen, müssen von allen Prozessen in derselben Reihenfolge wahrgenommen werden. Parallele Schreibvorgänge können auf unterschiedlichen Rechnern in unterschiedlicher Reihenfolge gesehen werden.
  - Hinweis: Tritt eine Leseoperation gefolgt von einer Schreiboperation ein, dann sind die Ereignisse potentiell kausal verknüpft.
  - Kausale Konsistenz ist schwächer als sequentielle Konsistenz.
- Beispiel:

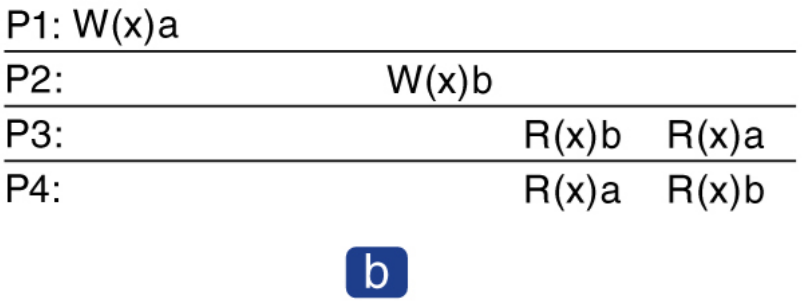
P1:	W(x)a				W(x)c	
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

Konsistente Anordnung von Operationen

- Kausale Konsistenz
  - Frage: Welcher der folgenden Abläufe ist aus Sicht der kausalen Konsistenz falsch?



a



b

Konsistente Anordnung von Operationen

- Gruppieren von Operationen
  - Arbeit mit Synchronisationsvariablen für wechselseitigen Ausschluss
  - Grund: Abfolge von Lese- und Schreiboperationen wird atomar
  - Synchronisationsvariable steht für bestimmen Satz an Daten (einzelne Datenelemente bis hin zur Gesamtheit der gemeinsam genutzten Daten)
  - Besitzer der Sync.-Variablen: Letzter Nutzer
  - Aneignung über Nachrichten an Besitzer
  - Kriterien an die Sync.-Variablen
    - Zugriff auf Sync.Variablen muss sequentiell konsistent sein
    - Kein Zugriff auf Sync.Variablen ist erlaubt, bis alle vorangegangen Aktualisierungen ausgeführt sind
    - Kein Daten-Zugriff ist erlaubt, bis alle vorangegangenen Zugriffe durch Sync.Variablen erfolgt sind

Beispiel: