Verteilte Softwaresysteme

Verteilte Algorithmen

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 28.11.2018 13:35

Inhaltsverzeichnis

Verteilt versus zentralisiert

Unterschiede

- ein verteiltes System unterscheidet sich von einem zentralisierten System durch:
  1. Es gibt keinen globalen Zustand.
  2. Es gibt keinen globalen Zeitrahmen.
  3. Es gibt kein deterministisches Verhalten.
- die drei Punkte werden im Folgenden noch etwas näher beleuchtet
- anschließend betrachten wir verschiedene Algorithmen um mit diesen Problemen umgehen zu können

Kein globaler Zustand

- bei zentralem Algorithms:
  - Entscheidungen basieren auf dem Zustand des Systems
  - Werte von Variablen werden ermittelt und auf deren Basis wird eine Entscheidung getroffen
  - zwischen der Inspektion und der Entscheidung werden keine Daten modifiziert
- Knoten in einem verteilten System haben nur Zugriff auf eigenen Zustand
- Idee könnte sein, dass Knoten Zustand der anderen Knoten einholt
  - es gibt aber keine Garantie, dass sich die Daten auf den anderen Knoten nicht bereits zwischen der Inspektion und der Entscheidung ändern
  - Entscheidung würde somit auf Basis von alten (jetzt ungültigen) Daten gefällt

Kein globaler Zeitrahmen

- Ereignisse bei zentralem Algorithmus sind total geordnet durch zeitliche Reihenfolge
  - für je zwei Ereignisse kann entschieden werden, welches davon *vor* dem anderen statt fand
- Zeitrelationen bei verteiltem Algorithms sind nicht total
- für Ereignisse auf einem Knoten kann entschieden werden, welches vor dem anderen lag
- bei Ereignissen auf zwei verschiedenen Knoten, die nicht in *Ursache-Wirkungsrelation* zueinander stehen, kann nicht entschieden werden, welches zeitlich vor dem anderen lag

Kein deterministisches Verhalten

- bei zentralem System ist die Berechnung basierend auf den Eingabewerten eindeutig
  - bei gegebenem Programm und Eingabe ist nur eine Berechnung möglich
- bei verteiltem System ist die globale Reihenfolge der Ereignisse nicht deterministisch
  - Ausnahme: Synchronisationsoperationen

Logische Ordnung von Ereignissen

Lamport-Zeit

- oft ausreichend, dass alle Knoten sich auf gemeinsame Zeit einigen, die nicht mit der realen übereinstimmen muss
  - **logische Uhren (*logical clocks*)**
- wird dazu noch gefordert, dass sich die Zeit nur um eine kleine Differenz von der realen unterscheidet:
  - **physikalische Uhren (*psysical clocks*)**
- zur logischen Uhrsynchronisation reicht die Bestimmung, in welcher zeitlichen Reihenfolge zwei Ereignisse miteinander stehen

Die Relation liegt vor (happens before)

- für zwei Ereignisse a und b definieren wir:
  - Es gilt $a \to b \Leftrightarrow a$ tritt zuerst auf und dann b
- die Relation $\to$ heisst dann **liegt vor (*happens before*)**
- es gilt:
  1. Sind a und b Ereignisse des gleichen Prozesses und tritt a vor b auf, dann gilt $a \to b$.
  2. Ist a das Senden einer Nachricht und b das Empfangen dieser Nachricht, dann gilt $a \to b$.
- $\to$ ist transitiv: $a \to b \wedge b \to c \Rightarrow a \to c$
- $\to$ hat irreflexive, partielle Ordnung
- zwei Ereignisse a und b für die weder 1. noch 2. gilt, stehen nicht in der liegt-vor-Relation
  - d.h. es gilt weder $a \to b$ noch $b \to a$
  - a und b sind dann **konkurrent**

Zeitstempel

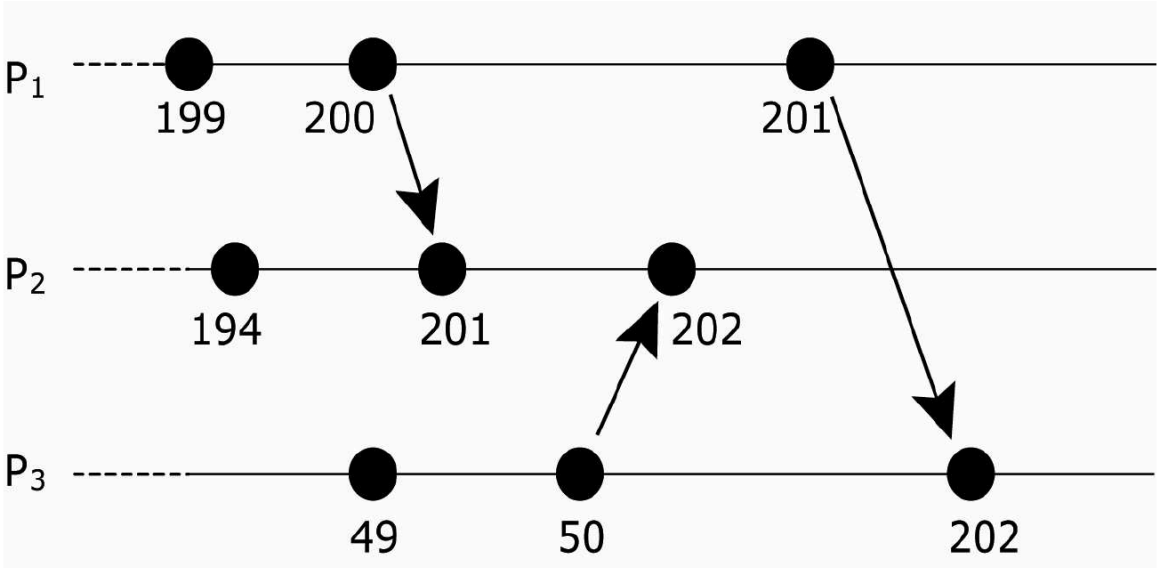
- um die Relation aufzustellen, brauchen wir keine synchronisierten Uhren
- mit jedem Ereignis a assoziieren wir einen **Zeitstempel** oder **Zeitwert** $C(a)$
- für alle Ereignisse a und b mit $a \to b$ muss gelten $C(a) < C(b)$
- zusätzlich gilt: der Zeitstempel muss immer anwachsen, kann also nie dekrementiert werden
- Zeitkorrekturen können nur durch Addition von positiven Werten vorgenommen werden

Logische Uhren

- um den Ereignissen Zeiten zuordnen zu können
  - ordnen wir jedem Prozess P_i eine logische Uhr C_i zu
- die logische Uhr C_i kann ein einfacher Zähler sein, der bei jedem Ereignis im Prozess P_i inkrementiert wird
  - jedem Ereignis wird eine eindeutige Zahl zugeordnet
- liegt a vor b muss gelten $C_i(a) < C_i(b)$
- die Ereignisse *innerhalb* eines Prozesses P_i haben damit eine **globale Ordnung**

Zeitstempel für mehrere Prozesse

- nachdem jeder Prozess eine eigene Uhr hat, kann es natürlich zu folgendem Beispiel kommen:
  - P_1 sendet zum Zeitpunkt $C_1(a) = 200$ Nachricht an P_2
  - P_2 empfängt die Nachricht zum Zeitpunkt $C_2(b) = 195$
  - das verletzt die Bedingung: $a \to b \Rightarrow C_1(a) < C_2(b)$
- Lösung: Wir stellen die Uhr des empfangenden Prozesses vor:
  - empfängt ein Prozess P_i mit de Ereignis b eine Nachricht mit Zeitstempel t
  - und es gilt $C_i(b) < t$
  - dann stellen wir die Uhr von P_i vor, so dass gilt $C_i(b) = t + 1$
- im obigen Beispiel gilt $C_2(b) = 200 + 1 = 201$



Quelle: Bengel et al., Masterkurs Parallele und Verteilte Systeme

Erweiterung auf totale Ordnung

- die partielle Ordnung $\to$ kann wie folgt auf eine totale Ordnung erweitert werden
- ist a ein Ereignis in P_i und b ein Ereignis in P_j dann gilt a liegt vor b, wenn

  (i) $C_i(a) < C_j(b)$ oder
  (ii) $C_i(a) = C_j(b)$ und $P_i < P_j$

Eine Zeit für das gesamte verteilte System

- durch die Erweiterung auf eine totale Ordnung, haben wir allen Ereignissen eine Zeit zugeordnet, für die gilt:
  1. liegt a vor b im gleichen Prozess, gilt $C(a) < C(b)$
  2. ist a das Senden einer Nachricht und b das Empfangen dieser Nachricht, gilt $C(a) < C(b)$
  3. für alle Ereignisse a und b gilt $C(a) \neq C(b)$
- **Lamport-Zeit**, benannt nach Leslie Lamport (u.a. Entwickler von LaTeX)
- Einsatz z.B. bei
  - Monitoring- und Debuggingsystem für verteilte Systeme
  - Lösung des Konkurrenzproblem bei Transaktionen
  - verteiltem Algorithmus für wechselseitigen Ausschluss

Vektoruhren

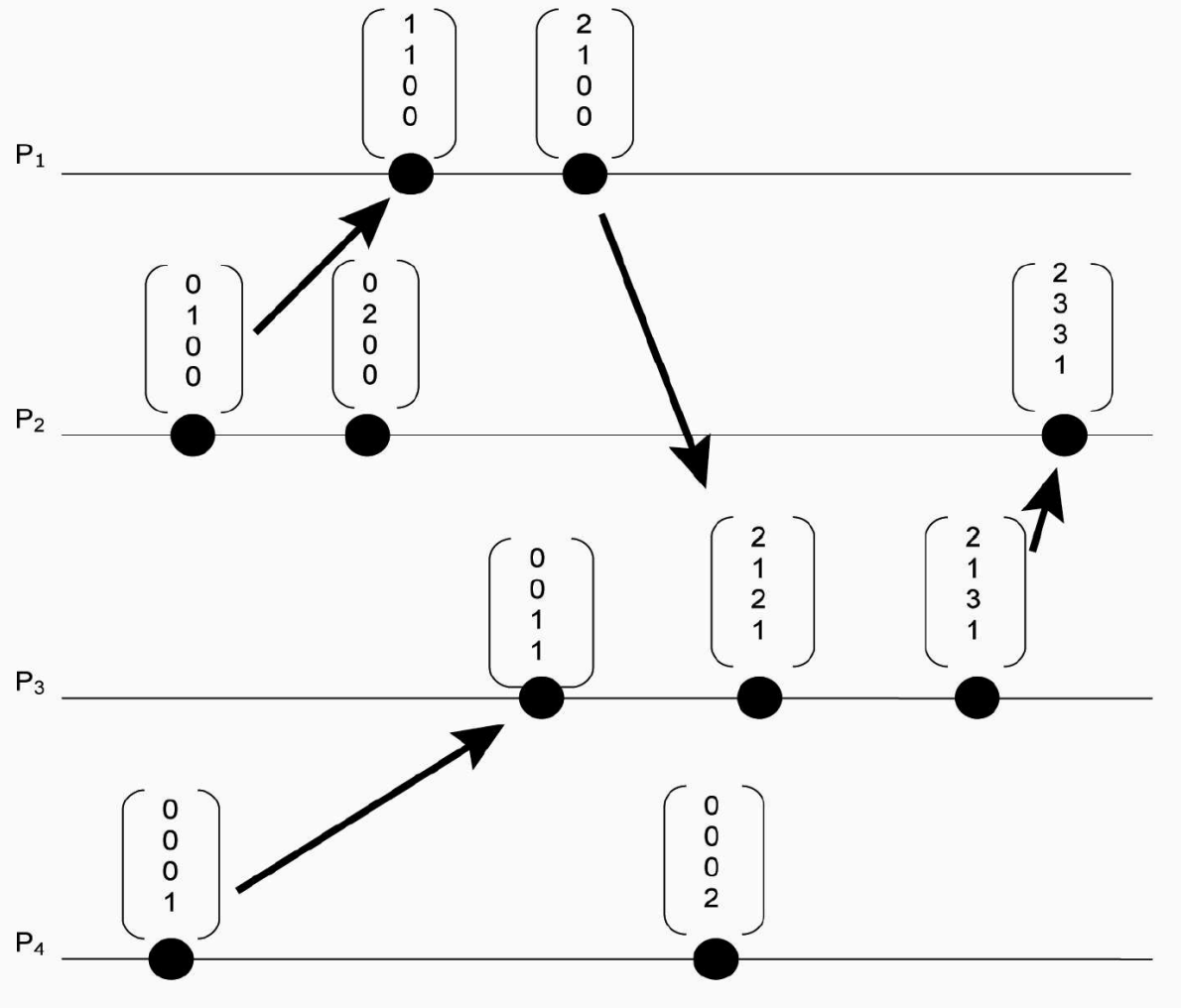
- die Lamport-Zeit hat eine gravierenden Nachteil
  - aus $C(a) < C(b)$ kann nicht auf $a \rightarrow b$ geschlossen werden
  - das liegt an den konkurrenten Ereignissen mit dem selben Zeitstempel, die über den Prozessindex geordnet werden
- zur Vermeidung dieses Nachteils **Vektoruhren** von Mattern
- jeder Prozess P_i besitzt eine einfache Uhr C_i
- ein idealisierter externer Beobachter hat Zugriff auf alle lokalen Uhren
- die Zeiten für alle Prozesse lassen sich zu einem Vektor zusammenfassen
- die Vektoruhr eines Prozesses repräsentiert also seine lokale Zeit und eine Abschätzung der lokalen Zeiten der anderen Prozesse
- bei n Prozessen gelten für die Vektoruhren folgende Regeln:
  1. jedes Ereignis von P_i inkrementiert die lokale Uhr C_i und damit das i-te Element des Vektors V_i

     $$V_i = (V_i[1], ..., V_i[i-1], V_i[i]+1, V_i[i+1], ..., V_i[n])$$

  2. Sendet ein Prozess P_i eine Nachricht an P_j
     - inkrementiert er seine lokale Uhr C_i
     - und sendet zusätzlich den Stand seiner lokalen Uhren

     P_j setzt jede Uhr dann auf das Maximum vom gespeicherten und dem empfangenen Wert.

Beispiel für $n = 4$:



Quelle: Bengel et al., Masterkurs Parallele und Verteilte Systeme

Ordnung der Ereignisse

- Vergleich der Zeitstempel:

  1. $V_1 = V_2 \Leftrightarrow V_1[i] = V_2[i], \forall i = 1, ..., n$
  2. $V_1 \leq V_2 \Leftrightarrow V_1[i] \leq V_2[i], \forall i = 1, ..., n$
  3. $V_1 < V_2 \Leftrightarrow V_1 \leq V_2 \wedge V_1 \neq V_2$

- für zwei Ereignisse a und b gilt:

  1. $a \rightarrow b \Rightarrow V(a) < V(b)$
  2. $\neg(V(a) < V(b)) \wedge \neg(V(b) < V(a)) \Rightarrow a$ und b sind konkurrent

  im Gegensatz zu Lamportuhren gilt auch:

3. $V(a) < V(b) \Rightarrow a \rightarrow b$

- mit Vektoruhren lässt sich also auch feststellen ob zwei Ereignisse gleichzeitig, also parallel aufgetreten sind

Auswahlalgorithmen

Problem

- viele verteilte Algorithmen basieren auf einem Server(prozess) und vielen Clients
- fällt der Server aus, steht das verteilte System
- daher repliziert man den Server und lässt einen anderen Server bei Ausfall die Funktion übernehmen
- benötigen Verfahren um einen neuen Server auszuwählen
- Voraussetzung:
  - auf n Rechnern läuft Kopie des Serveralgorithmus
  - aber nur ein Prozess ist tätig (*Master*)
  - alle anderen leiten Anfragen an diesen weiter
- Ausfall wird durch Timeout entdeckt
- **Auswahlalgorithmus** / **Leader Election** bestimmt neuen Master

Bully-Algorithmus

- verteilter Auswahlalgorithmus
- wählt von n Prozessen den mit dem höchsten Index als Master aus

$$P_m \text{ mit } m = max\{i | 1 \leq i \leq n, P_i \text{ist aktiv}\}$$

  - bestimmt den *bulligsten* Prozess

Ablauf

- bemerkt ein Prozess P_i, dass Master ausgefallen, startet er eine Wahl:
  1. P_i schickt Wahlnachricht an alle P_j mit $j > i$ und wartet ein Zeitintervall T auf Antwort.
     - Nachricht wird auch an alten Master geschickt, der evtl. nur überlastet ist
  2. Erhält P_j diese Nachricht, beendet er die Wahl von P_i und startet selbst eine Wahl

3. erhält P_i innerhalb Intervall keine Antwort, bestimmt er sich selbst zum neuen Master
   - informiert alle Prozesse $P_j, j < i$ durch Koordinationsnachricht
4. erhält er Nachricht, wartet er weiteres Zeitintervall T' ob neuer Master gefunden wurde
   - wenn nicht startet er neue Wahl

Aufwand

- Bully-Algorithmus erhöht Fehlertoleranz
- bezahlt aber mit hohem Kommunikationsaufwand
- worst case:
  - Master fällt aus, wird von P_1 bemerkt
  - P_1 sendet $n-1$ Nachrichten an $P_2, ..., P_n$
  - $P_2, ..., P_{n-1}$ starten neue Wahl
  - P_{n-1} wird schließlich neuer Master
  - Summe Auswahlnachrichten: $\sum_{i=1}^{n-1} n - i$
  - Summe Rückantworten: $\sum_{i=1}^{n-1} i - 1$
  - Koordinationsnachrichten: $n-2$

$$(\sum_{i=1}^{n-1} n - 1) + (n-2)$$

- nur sinnvoll, wenn sehr selten Fehlerfälle

Ring-Algorithmus

- basiert auf logischem Ring von zusammengeschlossenen Prozessen
- Ring in einer Richtung ausgelegt
- Nachrichten immer nur in eine Richtung gesendet
- jeder kennt seinen Nachfolger
- ist Nachfolger ausgefallen, sendet er an dessen Nachfolger,…

Ablauf

- bemerkt P_i, dass der Master ausgefallen ist, startet er Wahl

1. P_i legt Aktivenliste an und trägt sich ein

- sendet diese Liste an seinen Nachfolger
- erhält P_j Aktivenliste sind zwei Fälle zu unterscheiden

1. P_j erhält Aktivenliste zum ersten Mal, befindet sich nicht darin
   - trägt sich ein und sendet weiter
2. P_j steht bereits in der Aktivenliste
   - er kann Wahl beenden, Master auswählen (z.B. höchster Index)
   - sendet Nachricht mit neuem Koordinator in den Ring

2. empfängt P_j zum zweiten Mal die Koordinationsnachricht, ist Auswahl beendet

- im Worst Case $2n$ Nachrichten

Übereinstimmungsalgorithmen

Problem

- Menge von Prozessen müssen sich auf einen "gemeinsamen Wert" einigen können
- Übereinstimmung bei Fehlerfreiheit einfach, aber:

1. Kommunikationsmedium kann fehlerhaft sein

- Nachrichten gehen verloren
- Nachrichten werden korrumpiert

2. Prozesse können fehlerhaft sein

- im besten Fall stoppt der Prozess: **fail-stop failure**
- im schlechtesten Fall **byzantine failure**
  - Prozess arbeitet weiter, liefert aber fehlerhaftes Ergebnis
  - sendet inkorrekte Nachrichten
  - im Extremfall: kooperiert mit anderen fehlerhaften Prozessen

Byzantine Generals Problem

- mehrere Divisionen der byzantinischen Armee umgeben fendliches Lager
- ein General kommandiert jede dieser Divisionen
- alle Generäle müssen zu einer Übereinstimmung kommen
  - Angriff von nur wenigen Divisionen wird zur Niederlage führen
- Kommunikation ist nur durch Botschafter möglich, die von einem Divisionslager zum anderen laufen
- Probleme
  1. Botschafter können vom Feind gefangen genommen werden

   - Nachricht geht verloren

2. Unter den Generälen können Verräter sein, die eine Übereinstimmung verhindern wollen

Unzuverlässige Kommunikation

- Annahme: wenn ein Prozess ausfällt, stoppt er und arbeitet nicht mehr weiter (**fail-stop failure**)
- Voraussetzung: unzuverlässige Kommunikation zwischen den Prozessen
- Szenario: Prozess P_i auf Maschine A sendet Nachricht an P_j auf B
- P_i will wissen ob die Nachricht bei P_j angekommen ist
   - z.B. weil P_i verschieden weiter arbeiten will, je nachdem ob die Nachricht angekommen ist oder nicht

Time-out-Schema

- zur Entdeckung von Übertragungsfehlern: time-out-Schema:
- mit Senden der Nachricht startet P_i ein Zeitintervall in dem er eine Quittung haben möchte
- P_j empfängt und sendet sofort Quittung an P_i
- wenn P_i die Quittung innerhalb des Zeitintervalls empfängt, weiß er, dass P_j die Nachricht empfangen hat
- wenn er im Intervall keine Quittung bekommen hat, sendet er erneut
- das wird solange wiederholt, bis entweder Quittung oder Info von dem System auf B, dass P_j nicht mehr läuft

- nehmen wir weiter an, P_j will wissen ob seine Quittung angekommen ist
- also beide wollen z.B. eine Funktion `Success` genau dann aufrufen, wenn sie übereinstimmen
- bei unzuverlässiger Kommunikation ist diese Aufgabe nicht zu lösen!

Problemstellung allgemein

- bei einer verteilten Umgebung
- mit unzuverlässiger Kommunikation
- ist es **nicht möglich** für Prozesse P_i und P_j,
- dass sie über Ihre gegenwärtigen Zustände **übereinstimmen**
- Beweis:
   - Annahme: es existiert eine minimale Sequenz von Nachrichtenaustauschen, so dass danach beide Prozesse übereinstimmen, dass Sie `Success` ausführen

  - sei m' die letzte Nachricht die P_i an P_j sendet
  - da P_i nicht weiß ob seine Nachricht bei P_j angekommt, führt P_i die Funktion `Success` aus, unabhängig vom Ergebnis des Nachrichtenversandes
  - somit kann m' auch weg gelassen werden, ohne die Entscheidungsprozedur zu beeinflussen
  - d.h. ursprüngliche Sequenz war nicht minimal. ⇒ **Widerspruch!**

Byzantinische fehlerhafte Prozesse

- Annahme: Kommunikationssystem ist zuverlässig
- aber: Prozesse sind fehlerhaft und arbeiten mit unvorhersehbarem Verhalten (**Byzantine Failure**)
- gegeben: System mit n Prozessen, bei denen höchstens f fehlerhaft sind
- jeder Prozess P_i besitzt einen privaten Wert V_i
- wir benötigen Algorithmus, der für jeden nicht fehlerhaften Prozess P_i
  - einen Vektor $X_i = (A_{i1}, A_{i2}, ..., A_{in})$ konstruiert mit:

1. ist P_i fehlerfrei, so ist $A_{ij} = V_j$ und
2. sind P_i und P_j fehlerfrei, so gilt $X_i = Xj$

Eigenschaften der Lösungen

1. ein Lösungsalgorithmus zur Übereinstimmung kann nur gefunden werden, wenn gilt $n \geq 3 * f + 1$
   - d.h. es gibt keine Lösung für $2 \leq n \leq 3f$
2. die Anzahl der Kommunikationsrunden zur Erreichung einer Übereinstimmung ist proportional zu $f + 1$
3. die Anzahl der Nachrichten zur Erreichung einer Übereinstimmung ist hoch
   - keinem einzelnen Prozess kann getraut werden, so dass alle die gesamte Information sammeln müssen

Beispiel für $n = 4$ und $f = 1$

- Algorithms benötigt zwei Kommunikationsrunden

1. jeder Prozess sendet seinen eigenen Wert zu allen anderen Prozessen
2. jeder Prozess sendet seine in der ersten Runde enthaltenen Informationen zu allen anderen Prozessen

- wenn fehlerhafter Prozess keine Information sendet, wählt ein fehlerfreier Prozess irgendeinen willkürlichen Wert

- ein fehlerfreier Prozess P_i konstruiert seinen Vektor $X_i = (A_{i1}, A_{i2}, A_{i3}, A_{i4})$ wie folgt:
- $A_{ii} = V_i$
- für $j \neq i$:
  - wenn wenigstens zwei von den drei von Prozess gesendeten Werten übereinstimmen, dann wird die Majorität der zwei Werte genommen
  - sonst wird ein Defaultwert genommen, z.B. `nil`