Verteilte Algorithmen

- logische Ordnung von Ereignissen
- wechselseitiger Ausschluss (Konkurenzdienst)
- Wahlalgorithmen
- Konsensalgorithmen

Unterschiede zu zentralisiertem System

es gibt keinen gemeinsamen Zustand:

- Knoten im verteilten System kennen nur ihren eigenen Zustand und können keine Entscheidungen anhand des globalen Zustands fällen.
- es hilft nicht, zunächst alle Informationen aller Knoten zu sammeln: ein Knoten kann seinen Zustand bereits geändert haben, bis die Information beim Anfrager eintrifft. (aber: Variablen mittels Semaphore schützen!)

$\Rightarrow$ die gefällte Entscheidung beruht auf alten und somit ungültigen Daten

Unterschiede zu zentralisiertem System (2)

Abläufe sind unbestimmt:

- zenralisiertes System: bei einem gegebenen Programm und Eingabe ist nur eine Berechnung möglich
- verteiltes System: globale Reihenfolge der Ereignisse ist nicht deterministisch
- Beispiel: Server erhält viele Anfragen von unbekannter Anzahl Clients.
  * Server kann nicht warten, bis alle Anfragen eingetroffen sind, um sie dann in bestimmter Reihenfolge zu bearbeiten (Anzahl ist für ihn unbekannt)
  * Server bearbeitet Anfragen sofort, aber: die Reihenfolge, in der die Anfragen eintreffen, kann bei jedem Programmlauf unterschiedlich sein

Unterschiede zu zentralisiertem System (3)

es gibt keine gemeinsame Zeit:

- Ereignisse sind nicht total geordnet:
  * für Ereignisse auf einer Maschine kann entschieden werden, ob ein Ereignis vor einem anderen liegt.
  * bei Ereignissen auf zwei verschiedenen Maschinen, wobei die Ereignisse nicht in einer Ursache-Wirkung-Relation stehen, kann dies nicht entschieden werden

$\Rightarrow$ insgesamt: fehlerhafte verteilte Anwendungen

Wichtig: Synchronisationspunkte sowie ein Monitorsystem, das die zeitliche Reihenfolge der Aktionen und Ereignisse wiederspiegelt.

Synchrone verteilte Systeme

- die Ausführungszeit jedes Prozessschrittes hat bekannte untere und obere Grenzen
- jede übertragene Nachricht wird innerhalb einer bekannten, begrenzten Zeit empfangen
- der Abweichung der lokalen Uhren von der Echtzeit ist eine Grenze gesetzt
- alle Werte sind wahrscheinliche Grenzen
  * schwierige Festlegung von realistischen Werten
  * Probleme mit Zuverlässigkeit
  * Mechanismen zur Reservierung und Bereitstellung entsprechender Ressourcen notwendig

Asynchrone verteilte Systeme

- keine Beschränkung bzgl. Ausführungsgeschwindigkeit, Übertragungsverzögerung, Uhrabweichraten, ...
- Folgerungen: keine Annahme über Zeitintervalle erlaubt
  * Herstellen einer Abfolge von Ereignissen erfordert zusätzliche Konzepte: logische Zeit
  * exakte Abstimmung zwischen Ereignissen in diesem Modell nicht möglich
- Verteilte Systeme sind häufig asynchron
  * Prozesse müssen Prozessoren und Netzwerke gemeinsam nutzen
  * Angabe von Grenzen nicht möglich: unvorhersehbare Auslastung der Prozessoren und Netzwerke

Zeit in Verteilten Systemen

- Zeitstempel werden benötigt, um
  * Leistungsmessungen durchzuführen
  * Aktualität von Daten zu bewerten
  * Totalordnung von Objekten zu bewirken
- Probleme bei Verteilten Systemen:
  * Uhren sind mit Ungenauigkeiten behaftet: eine eindeutige, globale Zeit ist nicht verfügbar
  * selbst bei zentralen Zeitgeber: Ungenauigkeiten durch unterschiedliche Nachrichtenlaufzeiten

Beispiel: Abfolge von Ereignissen

Nachrichtenaustausch zwischen Benutzern X, Y und Z:

- Benutzer X sendet Nachricht mit dem Betreff `Meeting`
- Benutzer Y antwortet durch Senden einer Nachricht mit dem Betreff `Re: Meeting`
- Benutzer Z antwortet ebenfalls mit `Re: Meeting`, bezieht sich aber auf die Nachricht von X und von Y

Beispiel: Abfolge von Ereignissen (2)

Aufgrund voneinander unabhängiger Verzögerungen bei der Auslieferung der Nachrichten und unterschiedlicher lokaler Zeiten kann ein vierter Benutzer A folgende Reihenfolge bekommen

Eintrag	Von	Betreff
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

Problem: Benutzer A muss anhand der Texte die richtige Reihenfolge der Nachrichten ermitteln, um den Inhalt entsprechend zu verstehen

Logische Ordnung von Ereignissen

Idee: Rechner einigen sich auf gemeinsame Zeit: muss nicht mit realer Zeit übereinstimmen (logische Uhr)

Anforderung: Zeit auf allen Rechnern ist intern konsistent

physikalische Zeit: Uhren unterscheiden sich nur um einen kleinen, festen Betrag von der realen Zeit.

Logische Ordnung von Ereignissen (2)

- Ereignistypen:
  * Lokale Ereignisse
  * Sendeereignisse: Versenden einer Nachricht
  * Empfangsereignisse: Empfangen einer Nachricht
- Relation *liegt_vor*:
  * sind a und b Ereignisse des gleichen Prozesses, dann bedeutet $a \rightarrow b$: Ereignis a tritt vor Ereignis b auf, d.h. a *liegt_vor* b
  * ist a ein Ereignis des Sendens einer Nachricht und b das Ereignis des Empfangens in einem anderen Prozess, so gilt $a \rightarrow b$
  * *liegt_vor* ist eine transitive Relation (aus $a \rightarrow b$ und $b \rightarrow c$ folgt $a \rightarrow c$)

Logische Ordnung von Ereignissen (3)

- Zusammenfassung: Ereignisse, die in Relation *liegt_vor* zueinander stehen, treten entweder hintereinander im gleichen Prozess auf oder zwischen den beiden wird eine Nachricht gesendet
- Falls zwei Ereignisse a und b in verschiedenen Prozessen liegen und kein Nachrichtenaustausch vorliegt (auch nicht indirekt über einen dritten Prozess), dann stehen a und b nicht in *liegt_vor*-Relation
  * nebenläufige (konkurrente) Prozesse
  * keine Aussage über Auftreten der Ereignisse möglich

in einem solchen Fall helfen Zeitstempel

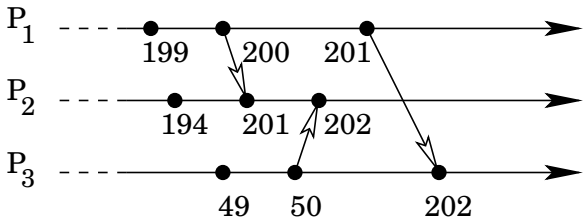
Zeitstempel

- Logische Uhr des Prozesses P_i: Monoton steigender SW-Zähler L_i
- Umwandeln des Zählers in Pseudozeit (Lamport-Zeit)
  * jedem Ereignis wird ein Zeitwert $L(a)$ assoziiert: $L(a)$ ist der **Zeitstempel** des Ereignisses a
  * falls $a \rightarrow b$, dann gilt $L(a) < L(b)$
  * **Zeitstempel können nur anwachsen**, d.h. Zeitkorrekturen ausschließlich durch Addition von positiven Werten möglich
- Unterscheidung:
  * $L_i(a)$: Zeitstempel von a im Prozess P_i
  * $L(a)$: globaler Zeitstempel von a im beliebigen Prozess

Zeitstempel (2)

Vorgehensweise zur Modellierung von *liegt_vor*:

1. $L_i = L_i + 1$, bevor a im Prozess P_i auftritt
2. Senden einer Nachricht m von P_1 an P_2:
   - P_1 sendet Wert $t = L_1$ zusammen mit Nachricht m
   - P_2 empfängt m und t und setzt $L_2 = \max(L_2, t)$. Dann $L_2 = L_2 + 1$, bevor $receive(m)$ durchgeführt wird.



Zeitstempel (3)

Also: die Uhr eines Prozesses wird vorgestellt, wenn er eine Nachricht erhält mit einem Zeitstempel größer als sein eigener Zeitstempel

Problem: identische Zeitstempel für zwei Ereignisse a und b in unterschiedlichen Prozessen P_i und P_j

totale Ordnung: beziehe Prozess-Identifikationsnummer mit ein: Ereignis a im Prozess P_i erhält als globalen Zeitstempel $L(a) = (L_i(a), P_i)$. Dann gilt

$$(L_i(a), P_i) < (L_j(b), P_j) \iff \begin{cases} L_i(a) < L_j(b) \\ L_i(a) = L_j(b) \wedge P_i < P_j \end{cases}$$

Zeitstempel (4)

Anwendung:

- bildet Grundlage für Monitor- und Debugging-System für verteilte Systeme
- verwendet in Algorithmus zur Lösung des Konkurrenzproblems für Transaktionen
- verwendet in verteiltem Algorithmus für wechselseitigen Ausschluss

Wechselseitiger Ausschluss

zur Koordination von Aktivitäten, z.B. beim Zugriff auf gemeinsam genutzte Ressourcen (siehe Kapitel Linux)

grundsätzliche Realisierungsmöglichkeiten:

- Server sammelt alle notwendigen Informationen und legt Entscheidung fest
  * Vorteil: Einfache Realisierung
  * Nachteil: Single Point of Failure
- Dynamische Festlegung in einer Gruppe von verteilten Komponenten
  * Vorteil: kein Single Point of Failure
  * Nachteil: komplexe Umsetzung

Wechselseitiger Ausschluss (2)

Systemsoftware: wechselseitiger Ausschluss wird realisiert mittels binärer Semaphore

verteilte Systeme: es existiert kein gemeinsamer Speicher → Lösung mit Hilfe von Nachrichten

einfache Lösung: verwenden eines zentralen Servers

- nur der Server gewährt Zutritt in kritische Bereiche
- Prozess sendet Nachricht (Eintrittswunsch)
- Eintrittswünsche werden in Warteschlange verwaltet
- nur einem Prozess wird Eintritt erlaubt (Rückantwort)
- Server benachrichtigen beim Verlassen des kritischen Abschnitts

Wechselseitiger Ausschluss (3)

Bewertung der Algorithmen:

- Bandbreite: Anzahl/Größe der gesendeten Nachrichten oder: Nachrichten pro Ein-/Austritt
- Clientverzögerungen: durch Warten auf Ein-/Austritt aus dem kritischen Bereich
- Systemdurchsatz: Geschwindigkeit, mit der die Gesamtheit der Prozesse auf den kritischen Bereich zugreifen kann unter der Voraussetzung, dass zwischen zwei aufeinanderfolgenden Prozessen eine Kommunikation notwendig ist → Synchronisationsverzögerung

Wechselseitiger Ausschluss (4)

Ring-basierter Algorithmus:

- Wechselseitiger Ausschluss zwischen n Prozessen ohne zusätzlichen Server-Prozess
- Prozesse sind als logischer Ring angeordnet: Prozess P_i mittels Kommunikationskanal mit Prozess $P_{(i+1) \bmod n}$ verbunden
- Vorgehensweise
  * Eintrittstoken wird als Nachricht weitergereicht
  * Prozesse, die nicht in k.A. eintreten wollen, leiten das Token einfach weiter
  * sonst: Prozess behält Token, nach Verlassen des kritischen Abschnitts Token weitergeben an Nachbarn

Wechselseitiger Ausschluss (5)

Bewertung:

- unnötiger Token-Wechsel, falls kein Prozess in den k.A. will $\rightarrow$ hohe Bandbreite
- fehleranfällig:
  * Ring muss immer geschlossen sein: Rekonfiguration nach Ausfall eines Prozesses
  * Token-Fehler (Duplikat oder Verlust) müssen erkannt und behoben werden $\rightarrow$ zentrale Überwachung
- Eintrittsverzögerung: 0 bis $n-1$
- Nachrichten pro Ein-/Austritt: 1 bis ∞

Wechselseitiger Ausschluss (6)

Abfragebasierter Algorithmus:

- Prozess will in k.A. eintreten: Anforderung senden als Multicast und auf Erlaubnis aller anderen Prozesse warten
- unterscheide zwei Fälle:
  * Prozess ist selber nicht an Eintritt in k.A. interessiert: Erlaubnis direkt verschicken
  * Prozess ist selber interessiert: Prozess, dessen Anfrage früher war, erhält die Erlaubnis (verwenden von Zeitstempeln)

Wechselseitiger Ausschluss (7)

Voraussetzungen des abfragebasierten Algorithmus:

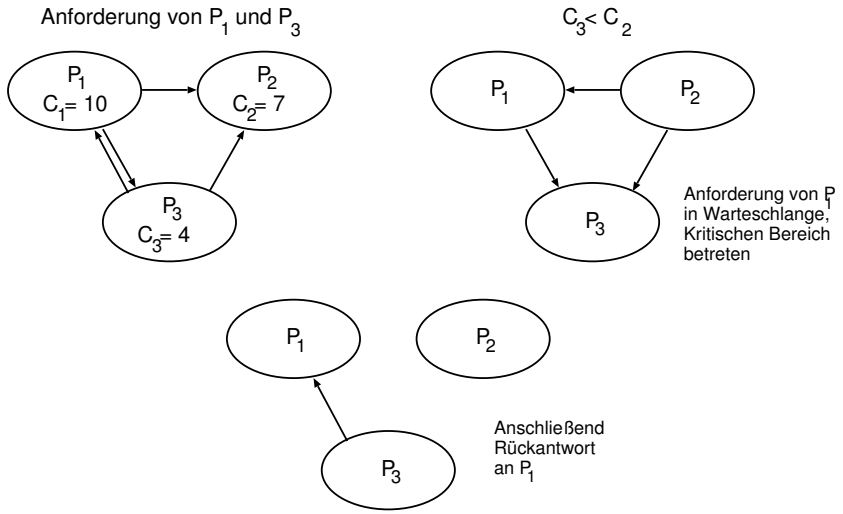
- jeder Prozess hat eine logische Uhr
- Anfragen bestehen aus Zeitstempel und Prozess-ID
- jeder Prozess kennt seinen Zustand
  * RELEASED: Außerhalb des kritischen Bereichs
  * WANTED: Zugang wird angefordert
  * HELD: Prozess ist im kritischen Bereich
- zuverlässige Kommunikation

Wechselseitiger Ausschluss (8)

Protokoll des abfragebasierten Algorithmus:

1. Anfrage verschicken
2. Antwort von Prozess P_i hängt von seinem Zustand ab:
   - RELEASED: Erlaubnis gewährt
   - HELD: keine Antwort, Anforderung in Warteschlange stellen
   - WANTED: falls Zeitstempel der Anfrage kleiner als eigener Zeitstempel, dann Antwort verschicken, sonst Anfrage in Warteschlange stellen
3. warte auf alle Antworten, dann eintreten in k.A.
4. verlassen des k.A.: Prozessen in Warteschlange Antwort schicken und aus Warteschlange entfernen

Wechselseitiger Ausschluss (9)



Wechselseitiger Ausschluss (10)

Bewertung:

- hohe Bandbreite: $\Theta(n^2)$, falls alle Prozesse ungefähr gleichzeitig in den k.A. wollen
- PID muss global eindeutig sein: · zentraler Manager · bei Prozesserzeugung Namen aller Prozesse erfragen und neuen Namen allen mitteilen · Struktur `IP:PID`
- fehleranfällig: Ausfall des Systems bei Ausfall eines Prozesses → überwachen durch zentralen Manager (informiert über Ausfall von Prozess P_x: Rückantwort von P_x muss nicht abgewartet werden, P_x muss nicht mehr um Erlaubnis gefragt werden)
- Prozesse, die nicht in k.A. wollen, werden mit Arbeit belastet (Anfrage annehmen und Erlaubnis verschicken)

Wahlalgorithmen

Algorithmen zur Wahl eines Prozesses, der dann eine bestimmte Aufgabe übernimmt

- typisch: Welcher Knoten wird zum Server deklariert?
- wichtig: alle Prozesse müssen sich auf die Wahl einigen

Replizieren der Server → bei Ausfall eines Servers kann ein anderer Server dessen Funktion übernehmen.

Voraussetzung:

- Kopie des Servers auf n Rechnern: nur ein Prozess ist tätig (Master), alle anderen dienen der Reserve
- Ausfall des Masters wird z.B. durch Timeout-Kontrolle von den untergeordneten Prozessen bemerkt

Wahlalgorithmen (2)

zu beachten:

- ein Prozess veranlasst eine Wahl (einzelner Prozess kann nur eine Wahl veranlassen)
- n Prozesse können n nebenläufige Wahlen veranlassen (jede Wahl muss auch bei Nebenläufigkeit eindeutig sein)
- ein Prozess P_i ist zu jedem Zeitpunkt entweder an der Wahl beteiligt oder nicht
- jeder Prozess erhält eindeutige ID, Prozess mit größter ID (o.B.d.A) gewinnt die Wahl
- jeder Prozess kennt den gewählten Prozess (spezielles Symbol für undefinierte Werte notwendig)

Wahlalgorithmen (3)

Bully-Algorithmus: P_i bemerkt Ausfall des Masters

1 P_i schickt Nachricht an alle P_k mit $k > i$ und wartet ein Zeitintervall T auf Antwort, außerdem Nachricht an bisherigen Master

2 erhält Prozess P_j ($j > i$) die Nachricht, schickt er eine Antwort und startet seinen eigene Auswahl

3a P_i erhält innerhalb von T keine Antwort: er bestimmt sich selbst zum neuen Master und schickt diese Info an alle Prozesse P_j mit $j < i$

3b P_i erhält mind. eine Antwort innerhalb von T: warte Zeitintervall T' auf Bestätigung, dass ein Prozess P_k mit $k > i$ als neuer Master bestimmt wurde $\rightarrow$ evtl. Wahlalgorithmus erneut starten

Wahlalgorithmen (4)

Anmerkungen:

- Timeout T: schätze maximale Übertragungszeit T_{trans} und maximale Verarbeitungszeit $T_{process}$

$$T = 2 \cdot T_{trans} + T_{process}$$

- T' muss viel größer als T gewählt werden
- Anzahl verschickter Nachrichten:
  * $\Theta(n^2)$, wenn P_n ausfällt und P_1 den Ausfall bemerkt
  * $\Theta(n^3)$, falls zusätzlich die Prozesse $P_{n-1}, P_{n-2}, \ldots$ ausfallen, bevor die Bestätigungsnachrichten geschickt werden können

Wahlalgorithmen (5)

Ring-Algorithmus: Prozesse bilden logischen Ring: jeder Prozess kennt seinen Nachbarn, Nachrichten werden nur in eine Richtung gesendet

(laut Bengel u. Tanenbaum/Steen: Ist Nachfolgeprozess ausgefallen, kann der Sender übergehen zu dessen Nachfolger, und falls der ausgefallen ist, auf dessen Nachfolger usw. $\rightarrow$ Kommunikationsstruktur Clique)

- Initialisierung:
  * Prozess P_i veranlasst die Wahl, markiert sich als Teilnehmer und sendet die Wahlnachricht
  * jeder empfangende Prozess vergleicht die Nachricht mit seiner eigenen ID

Wahlalgorithmen (6)

Ring-Algorithmus: (Fortsetzung)

- Vergleichsergebnisse:
  * empfangene ID größer und Empfänger noch kein Teilnehmer: Nachricht unverändert weiterreichen, Prozess markiert sich als Teilnehmer
  * empfangene ID kleiner und Empfänger noch kein Teilnehmer: eintragen der eigenen ID, weitergeben an Nachbarn, markieren als Teilnehmer
  * empfangene ID gleich eigener ID: aktueller Prozess hat größte ID, die Wahl ist entschieden, Markierung löschen und ID an Nachbarn weitergeben
  * empfangende Prozesse löschen die Markierung und reichen ID weiter

Wahlalgorithmen (7)

Anmerkungen:

- Anzahl verschickter Nachrichten: $3n - 1$ im worst-case
- Kommunikationsfehler oder Abstürze einzelner Komponenten führen zum Ausfall des gesamten Algorithmus
- geringerer Aufwand als Bully-Algorithmus, aber fehleranfällig
- warum bestimmt nicht jeder Prozess anhand der eigenen Informationen den neuen Master? Prozesse bemerken Ausfall des Masters zu unterschiedlichen Zeiten
  $\rightarrow$ Inkonsistenzen nach Neustart eines früheren, ausgefallenen Masters

Konsensalgorithmen

Konsens: Übereinstimmung bzgl. gemeinsamen Wertes

Motivation: mehrere Informationsquellen und steuernde Einheiten in Geräten $\rightarrow$ Steuereinheiten müssen sich auf Konsens einigen, ob Aktion durchgeführt/abgebrochen wird

Beispiele:

- Abbruch eines Startvorgangs z.B. bei Flugzeugen
- Einleiten einer automatischen Bremsung z.B. bei Zügen
- Überweisung: beide Rechner buchen gleichen Betrag ab
- Eintritt in k.A.: einigen auf gewählten Prozess

kein Problem bei fehlerfreien Kommunikationsmedien, Prozessen und Prozessumgebungen (Hardware)

Konsensalgorithmen (2)

Fehlerquellen:

- fehlerhafte Kommunikationskanäle: Verfälschung oder Verlust einer Nachricht
- fehlerhafte Prozesse: unvorhersehbares Verhalten
  * Fail-stop failure: Prozess stoppt nach Fehler $\rightarrow$ Entdeckung des Fehlers einfach
  * Byzantine failure: Prozess arbeitet fehlerhaft und erzeugt falsche Ergebnisse $\rightarrow$ inkorrekte Nachrichten und Gefahr für Integrität des Verteilten Systems

Konsensalgorithmen (3)

Ablauf: jeder Prozess P_i beginnt im nicht dedizierten Zustand und schlägt einen Wert v_i vor

- Prozesse kommunizieren und tauschen Werte aus
- jeder Prozess P_i setzt seine Entscheidungsvariable d_i auf den Wert, auf den sich die Prozesse geeinigt haben
- Übergang in dedizierten Zustand: d_i darf nicht mehr geändert werden

Konsensalgorithmen (4)

für jeden Konsensalgorithmus muss gelten:

- Terminierung: jeder korrekte Prozess setzt irgendwann die Entscheidungsvariable
- Einigung: Entscheidungswert aller korrekten Prozesse ist gleich
- Integrität: haben alle korrekten Prozesse denselben Wert vorgeschlagen, dann hat jeder korrekte Prozess im dedizierten Zustand diesen Wert gewählt

Konsensalgorithmen (5)

kein Algorithmus kann Konsens in asynchronem System garantieren,

- wenn Prozesse abstürzen können
  * Prozesse können zu beliebigen Zeitpunkten auf Nachrichten antworten $\rightarrow$ keine Unterscheidung zwischen langsamen/abgestürzten Prozessen möglich
  * Beweis der Nicht-Existenz von Fischer et al.
- wenn Kommunikation unzuverlässig
  * Kommunikation mittels Quittierungen: ein Prozess P_i schickt P_j seine Information, P_j bestätigt den Erhalt der Information, P_i bestätigt den Erhalt der Quittung
  * Problem: Quittung kann verloren gehen (unsicher)
  * Beweis durch Widerspruch: keine minimale Sequenz

Konsensalgorithmen (6)

Ansätze zur Umgehung des Unmöglichkeitsergebnisses:

- Verwenden partiell synchroner Systeme: abgeschwächte synchrone Verteilte Systeme, die aber strengeren Voraussetzungen als asynchrone Systeme unterliegen
- Fehlermaskierung: Prozessausfälle z.B. durch persistente Speicherung aller relevanten Informationen und Prozessneustart maskieren (wird bei Transaktionsverarbeitung eingesetzt)
- Fehlerdetektoren: Prozesse einigen sich darauf, dass ein Prozess, der eine bestimmte Zeit nicht geantwortet hat, als ausgefallen betrachtet wird und von Entscheidungen ausgeschlossen wird

Konsensalgorithmen (7)

Problem der byzantinischen Generäle:

- spezielle Variante des Konsensproblems
- Situation:
  * mehrere Divisionen der byzantinischen Armee umgeben ein Lager
  * jede der geografisch verstreuten Divisionen wird von einem General kommandiert
  * Generäle müssen übereinstimmen, ob im Morgengrau ein Angriff gestartet werden soll (Angriff mit nicht maximaler Stärke würde in Niederlage enden)

Konsensalgorithmen (8)

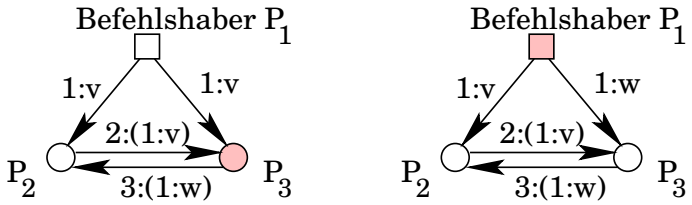
Problem der byzantinischen Generäle: (Fortsetzung)

- Kommunikation zwischen Divisionen mit Botschaftern
- Übereinstimmung der Generäle kann verhindert werden:
  * einige Botschafter werden gefangen genommen (unzuverlässige Kommunikation)
  * einige Generäle sind Verräter, die die Übereinstimmung verhindern wollen (fehlerhafte Prozesse)
- Voraussetzungen:
  * bis zu f der N Prozesse weisen beliebige Fehler auf: beliebige Nachricht mit beliebigem Wert senden
  * ausbleibende Nachrichten durch Timeouts erkennen
  * private Übertragungskanäle

Konsensalgorithmen (9)

zunächst: Befehl wird von einem Befehlshaber erteilt

fällt einer von 3 Prozessen aus, so gibt es keine Lösung (allgemein: Erweiterung für $N \leq 3 \cdot f$)



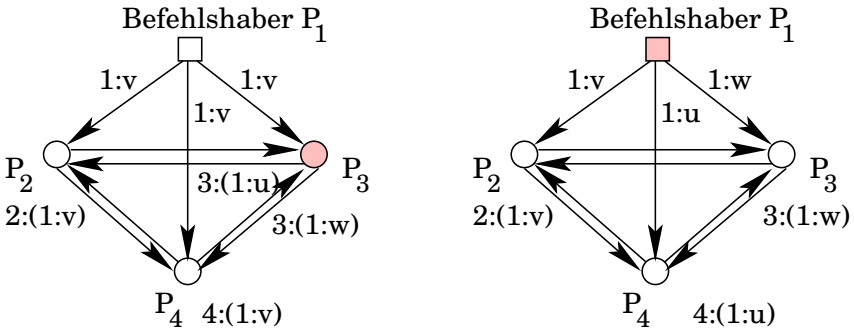
Problem: P_2 kann nicht erkennen, ob der Befehl des Befehlshabers oder des Untergebenen P_3 falsch ist

Konsensalgorithmen (10)

Lösung für $N = 4$ und einem fehlerhaften Prozess:

- die korrekten Generäle erzielen Einigung in zwei Nachrichtenrunden
  1. Befehlshaber sendet Wert an jeden Untergebenen
  2. jeder Untergebene sendet empfangenen Wert an die gleichgestellten Generäle
- jeder General empfängt einen Wert vom Befehlshaber und $N - 2$ Werte von gleichgestellten Generälen
  * Befehlshaber fehlerhaft: alle untergeordneten Generäle zeichnen die gleiche Wertmenge auf
  * Abweichende Wertmengen: General fehlerhaft, Bestimmung durch einfache Mehrheitsfunktion

Konsensalgorithmen (11)



P_2: `majority(v,-,u,v) = v`
P_4: `majority(v,v,w,-) = v`

P_2: `majority(v,-,w,u) = %`
P_3: `majority(w,v,-,u) = %`
P_4: `majority(u,v,w,-) = %`

Konsensalgorithmen (12)

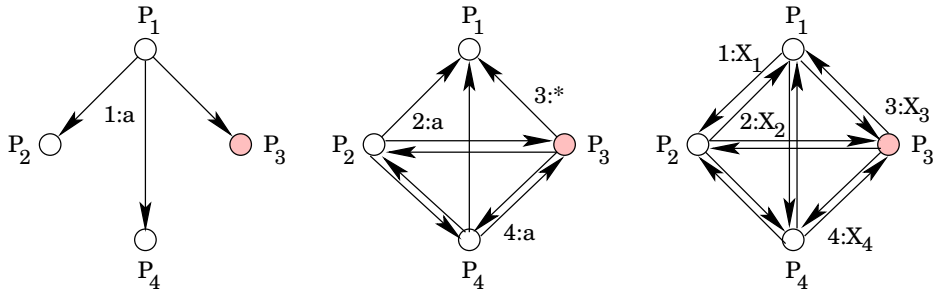
verallgemeinern: alle Generäle sind gleichberechtigt
$\rightarrow$ rekursiver Algorithmus

Algorithmus: für $f = 1$ werden nur 2 Runden benötigt

1. ein Prozess P_k beginnt die Wahl: senden seines Wertes d_k an alle anderen (Wert d_i vom Prozess P_i ist gleich d_k, falls Sender, Empfänger und Kommunikation o.k. sind)
2. jeder Prozess P_i sendet seinen Wert d_i allen anderen Prozessen
3. jeder Prozess sendet die in der ersten Runde erhaltenen Informationen zu allen anderen Prozessen

$\rightarrow$ Prozess speichert $n \times n$-Matrix anstelle eines Vektors

Konsensalgorithmen (13)



	init	**nach 1**	**nach 2**	**nach 3**
P_1	$d_1 = a$	$X_1 = (a, \cdot, \cdot, \cdot)$	$X_1 = (a, a, *, a)$	X_1, X_2, X_3, X_4
P_2	$d_2 = a$	$X_2 = (a, a, \cdot, \cdot)$	$X_2 = (a, a, *, a)$	X_1, X_2, X_3, X_4
P_3	$d_3 = *$	$X_3 = (*, \cdot, *, \cdot)$	$X_3 = (*, a, *, a)$	$*, *, *, *$
P_4	$d_4 = a$	$X_4 = (a, \cdot, \cdot, a)$	$X_4 = (a, a, *, a)$	X_1, X_2, X_3, X_4