Verteilte Systeme

Synchrone verteilte Systeme

Synchrone verteilte System

Ein synchrones VS auf den Prozessen $p \in P$ und dem Nachrichtenalphabet M besteht aus:

$Stati_p$ - Menge der Zustände (ggf. nicht endlicht!)

$Start_p$ - Teilmenge $Stati_p$ der Startzustände

$Msgg_p$ - Nachrichtengeneratorfunktion

$$Stati_p \times Neigh_{send}(p) \longrightarrow M \cup \{null\}$$

$Trans_p$ - Zustandsübergangsfunktion

$$M \cup \{null\} \times Neigh_{recv}(p) \times Stati_p \longrightarrow Stati_p$$

wobei: $Neigh_{send}(p)$ Kanäle, über die p senden kann

$Neigh_{recv}(p)$ Kanäle, über die p empfangen kann

Im Falle von symmetrischen Kanälen ist $Neigh_{send}(p) = Neigh_{recv}(p)$

Synchrone verteilte System

Ablauf einer Runde im synchronen verteilten System:

1. Nachrichtengeneratorfunktion wird auf alle Prozesse angewandt, die Nachrichten wenden an die Nachbarn über die entsprechenden Kanäle versandt.

2. Die Zustandsübergangsfunktion wird auf den aktuellen Zustand und die eingegangenen Nachrichten angewandt.

Synchrone verteilte System

Varianten:

- Haltezustände:
  - Zustände, die keine weiteren Nachrichten versenden und die ein Prozess nicht mehr verlässt
  - nicht zu verwechseln mit akzeptierenden Zuständen
- Variable Anfangszeiten:
  - Knoten fangen erst ab einer gewissen Runde an am Algorithmus zu partizipieren.
  - Realisiert über ausgezeichneten Prozess, der Start-Nachrichten sendet und Startzustände, die keine Nachrichten generieren.
  - ggf start durch den Empfang von Nachrichten von anderen Prozessen.

Synchrone verteilte System

Prozess-Fehler:

Ein Prozess hält in einem Zustand und versendet keine weiteren Nachrichten.

Kanalfehler:

Ein Kanal verliert Nachrichten.
{null} statt der gesendeten Nachricht

Byzantinische Fehler:

Ein Prozess versendet beliebige Nachrichten aus M x {null} unabhängig von $Msgg_p$

Verteilte Systeme

Auswahlalgorithmen II
(leader election algorithms)

LCR-Algorithmus [LeLann 1977, Chang/Roberts 1979]

bestimmt Station mit höchster ID als Koordinator
in einem Synchronen Ring

Vor.:

- Stationen sind ausfallsicher (!), bzw. werden bei Ausfall automatisch übersprungen

LCR-Algorithmus [LeLann 1977, Chang/Roberts 1979]

```
Initialisierung:
  status = unknown
  send = ID

Nachrichtengeneratorfunktion:
  sende send an nie nächste Station im Ring,
  die die Nachricht dann aus recv lesen kann.

Zustandsübergangsfunktion:
  send := null
  case recv:
    | recv > ID: send := recv
    | recv = ID: status := leader
    | recv < ID: skip
```

LCR-Algorithmus [LeLann 1977, Chang/Roberts 1979]

Kommunikationskomplexität:

- Best-Case: O(n)
- Worst-Case: $O(n^2)$

Zeitkomplexität:

- nach n Runden ist die Nachricht bei ID_{max}: O(n)

Time-Slice-Algorithmus

Idee:

Wir nutzen das Zeitverhalten aus

p_{min} Phasen mit n Runden

in Phase i dürfen nur Nachrichten mit ID i zirkulieren

Algorithmus:

Wenn in Phase i der Knoten mit ID i noch keine Nachricht erhalten hat, erklärt er sich zum Koordinator und sendet ID den Ring entlang

Kommunikationskomplexität: n

Zeitkomplexität: $n*ID_{min}$

nicht notwendigerweise beschränkt (!)

LCR-Algorithmus (async)

bestimmt Station mit höchster ID als Koordinator

modifiziert für einen asynchronen Ring

Vor.:

- Stationen sind ausfallsicher (!), bzw. werden bei Ausfall automatisch übersprungen

Modifikation:

- Send ist ein ein FIFO-Puffer, der das älteste Element bei recv auf der anderen Kanalseite zurückgibt

LCR-Algorithmus (async)

```
Initialisierung:
  status = unknown
  send = ID

Nachrichtengeneratorfunktion:
  sende send an nie nächste Station im Ring,
  die die Nachricht dann aus recv lesen kann.

Zustandsübergangsfunktion:
  send := null
  case recv:
    | recv > ID: send := recv
    | recv = ID: status := leader
    | recv < ID: skip
```

LCR-Algorithmus (async)

Kommunikationskomplexität:

Best-Case: O(n)

Worst-Case: $O(n^2)$

Zeitkomplexität:

Annahmen: l = obere Schranke der Bearbeitungszeit

d = obere Schranke der Nachrichten-übermittlungszeit

faires Scheduling

die Nachricht ist nach n(l+d) ist bei ID_{max}

Peterson Leader Election Algorithm

wählt eine beliebige Station als Koordinator
in einem unidirektionalen asynchronen Ring.

Idee:

Pro Phase reduzieren wir die Anzahl der “aktiven” Stationen um mindestens die Hälfte.

Initialisierung: $uid_0 := ID_p$

Algorithmus:

Jede Station hat eine vergleicht seine Temporäre uid_0 mit der seiner beiden Vorgänger uid_1, uid_2.
Ist $uid_1 > max\{uid_0, uid_2\}$, dann bleibt er aktiv und setzt $uid_0 := uid_1$, sonst wird er relay.
Wenn $uid_0 == uid_1$ empfangen wird, wird uid_0 Koordinator.

Peterson Leader Election Algorithm

Kommunikationskomplexität: O(n*log(n))

- Pro Phase sendet jeder Prozess maximal 2 Nachrichten
- nach log(n)+1 Phasen steht der Koordinator fest

=> 2n*(log(n)+1) = O(n*log(n))

Zeitkomplexität: O(n*log(n))

Annahmen: l = obere Schranke der Bearbeitungszeit

d = obere Schranke der Nachrichten-übermittlungszeit

faires Scheduling

Verteilte Systeme

Sperrsynchronisation (mutual exclusion)

Sperrsynchronisation

Problem:

- Zugriff auf eine gemeinsame Ressource, der gegenseitigen Ausschluss erfordert (vgl. ALP IV)

- Realisierung von Semaphoren oder Monitoren durch atomare Operationen (z.B. “test and set”) oder gemeinsamen Speicher nicht möglich

Sperrsynchronisation

Lösung: Verteilter Algorithmus, der folgende Eigenschaften garantiert:

1. Sicherheit (safety) :
   Höchstens ein Prozess darf sich im kritischen Abschnitt befinden

2. Lebendigkeit (liveness) :
   Jeder Prozess kann irgendwann in den kritischen Abschnitt eintreten (keine Verklemmung/ kein Verhungern)

3. Ordnung (ordering): Eintritt in den kritischen Abschnitt kausal geordnet.

Koordinator

Eine ausgezeichnete Station s fungiert als Koordinator und stellt den anderen einen Semaphor bereit:

```
send request to s
recv grant from s
begin critical
   ...
end critical
send release to s
```

Erfüllt Sicherheit und Lebendigkeit, aber nicht Ordnung!

Vorteil: Einfach, wenig Kommunikationsaufwand O(1)

Nachteil: Single point of failure

Simulation von gemeinsamem Speicher

Gegenseitiger Ausschluss wird auf virtuellem verteiltem Speicher realisiert (z.B. mit dem Bakery-Algorithmus).

Erfüllt Sicherheit und Lebendigkeit und Ordnung

Vorteil: analoge Implementierung zu lokalem Ausschluss

Nachteile:

- hohe Gesamtkomplexität
- hohes Nachrichtenaufkommen

Virtueller Token-Ring

Der Besitz des Token erlaubt eintritt in den kritischen Abschnitt.

Erfüllt Sicherheit und Lebendigkeit, aber nicht Ordnung!

Vorteile:

+ einfach
+ geringes Nachrichtenaufkommen wenn Ressource häufig von allen benötigt

Nachteile:

- Ordnung / Fairnesseigenschaften problematisch
- fehleranfällig
- hohes Nachrichtenaufkommen wenn Ressource selten benötigt

Logische Uhren [Lamport 1978, Ricart/Agrawala 1981/83]

Idee:

Bei Konflikten wird die Eintrittsreihenfolge in den kritischen Abschnitt wird durch eine logische Uhr festgelegt.

Erfüllt Sicherheit und Lebendigkeit und Ordnung!

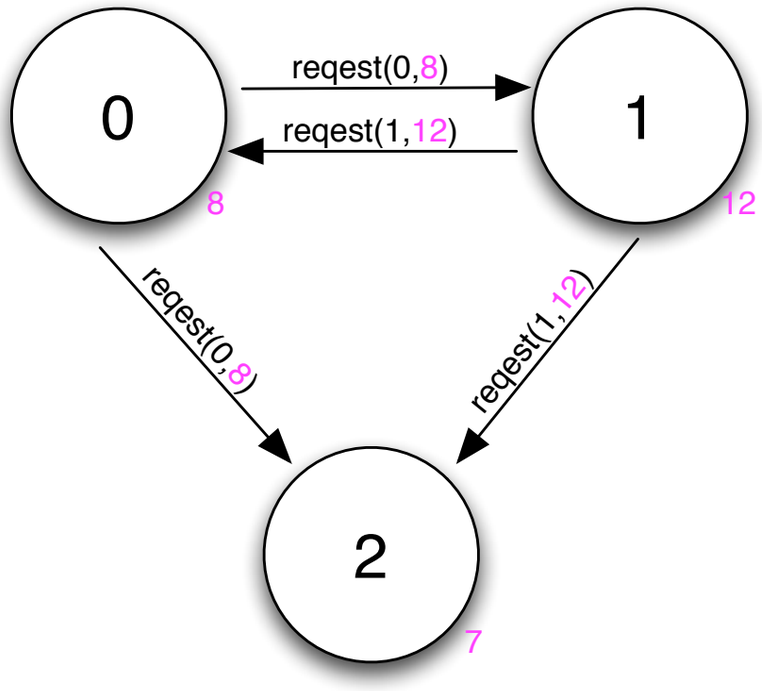
Voraussetzungen:

- Zuverlässige FIFO Gruppenkommunikation
- Anzahl n der Stationen bekannt

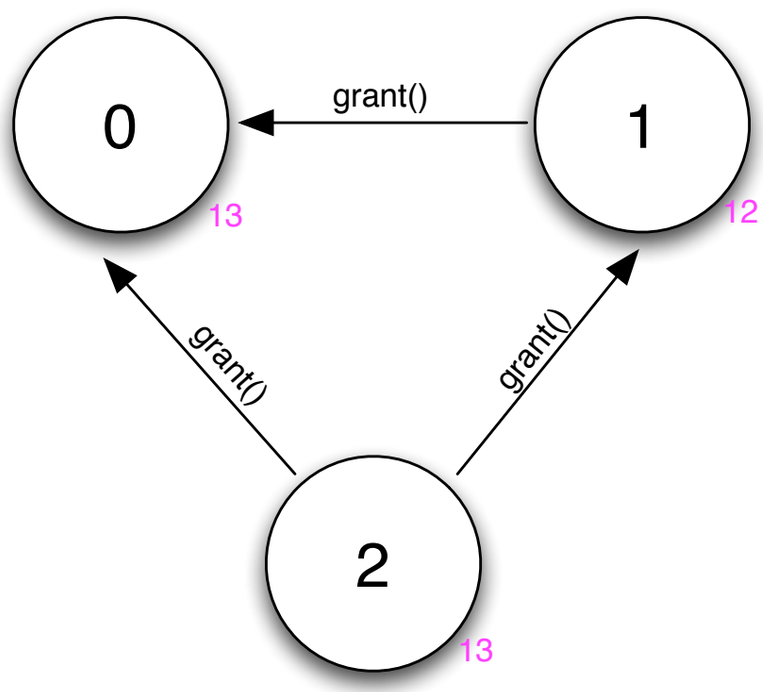
Logische Uhren [Lamport 1978, Ricart/Agrawala 1981/83]

```
monitor Station()
  clock := 0 // Skalarzeit
  state := RELEASED;
  granted := 0;

  proc enter_critical()
    state := REQ;
    rclock := clock;
    send request(rclock, this) // Broadcast
    await (granted = (n-1));
    state := CRIT;

  port request(T time, Station s)
    if (state = CRIT or (state = REQ and (rclock, this) < (time, s)))
    then queue.append(s);
    else send grant() to s; // OK - Eintrittswunsch stattgegeben

  port grant()
    granted := granted + 1

  proc leave()
    state := RELEASED
    forall s in queue do send grant() to s
```

Logische Uhren [Lamport 1978, Ricart/Agrawala 1981/83]

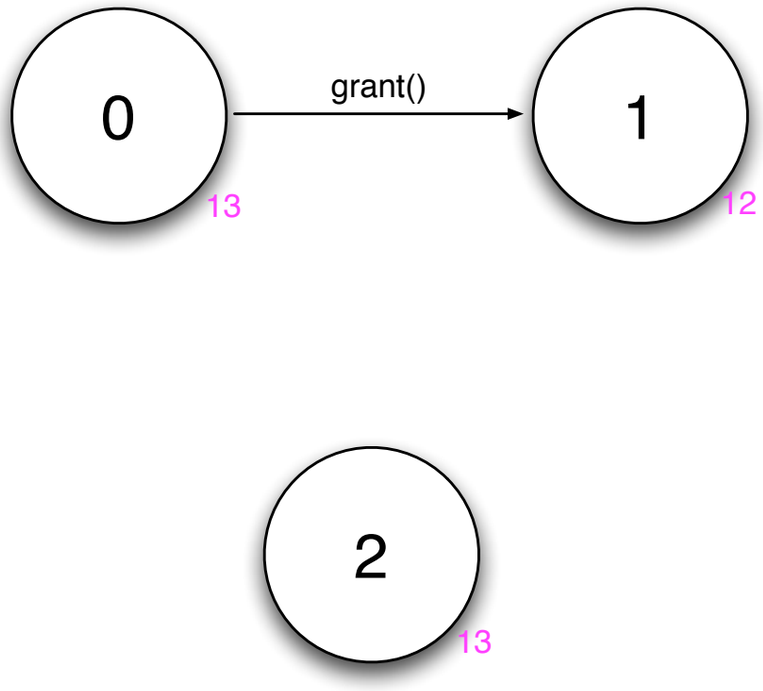


0 und 1 wollen in den kritischen Abschnitt eintreten



Skalarzeit von 0 ist kleiner als die von 1

0 tritt in den kritischen Abschnitt ein



0 verlässt den kritischen Abschnitt

1 tritt in den kritischen Abschnitt ein

Logische Uhren [Lamport 1978, Ricart/Agrawala 1981/83]

Kommunikationsaufwand:

2(n-1) Nachrichten pro Eintritt

(ohne multicastfähige Hardware)

Vorteile

- relativ einfach
- erhält kausalität
- gute Fairnesseigenschaften

Nachteile:

- Bei Stationsausfall droht Verklemmung!
- relativ hoher Kommunikationsaufwand

Übungsaufgabe zum 17.5.2011

- Erweitern Sie Ihre UdpChannel Implementierung um die nichtblockierende nread()-Methode

- Bauen Sie sich ein kleines Framework, um synchrone Algorithmen auf unserer Channel-Abstraktion ausführen zu können.

- Implementieren Sie den Time-Slice-Algorithmus in Ihrem Framework.
  Die main-Methode der Abgabe in soll in einer Klasse namens TimeSlice.java liegen. Die Implementierung soll die einzelnen Nachrichtenm die Runden- und Phasen-Nummern und den gewählten Koordinator ausgeben.