

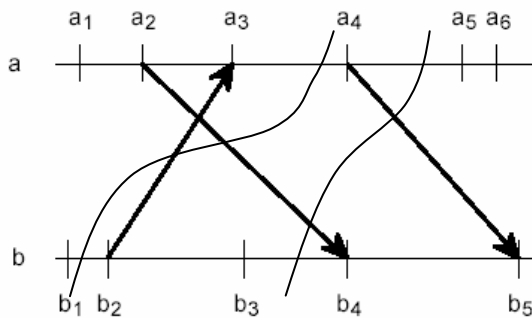
# Übung 4 Gruppe Hosebei

03.06.2002

Michele Luongo	migi@luongo.org	s99-713-190
Franziska Zumsteg	F.Zumsteg@access.unizh.ch	s99-717-084
Philip Iezzi	pipo@iezzi.ch	s99-714-354
Raphael Bianchi	saint.ch@dte.ch	s95-662-003

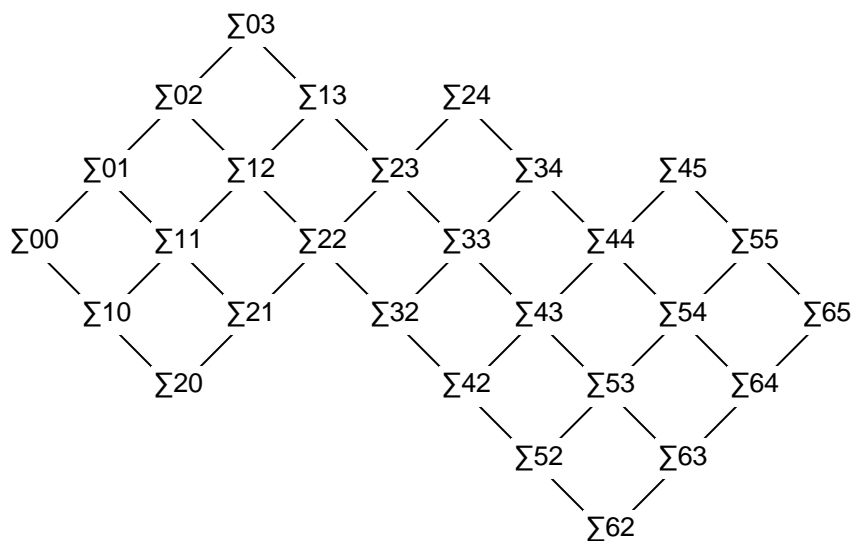
## Aufgabe 4.1

### a) konsistente/inkonsistente Sichten

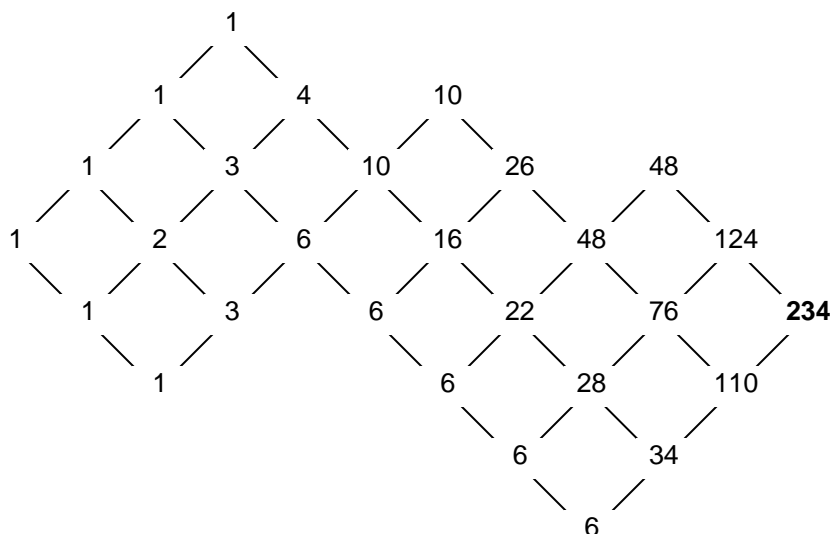


inkonsistente  $C = (3, 1)$       konsistente Sicht  $C = (4, 3)$

### b) geordnetes Gitter der globalen Zustände



### c) Anzahl konsistenter Runs



→ Total gibt es also **234 konsistente Runs**.

## Aufgabe 4.2

### Qualitäten

- "maybe" ein Packet kommt vielleicht nie am Ziel an.
- "at least once" mindestens ein Packet erreicht Ziel (Duplikate möglich).
- "at most once" höchstens ein Packet erreicht Ziel. (fehlende Pakete können nicht ausgeschlossen werden)
- "exactly once" idealisiertes "at most once" genau ein Packet kommt ins Ziel

### a) exactly-once

Exactly-once ist dadurch gekennzeichnet, dass eine Nachricht vom Empfänger genau einmal gelesen wird.

Mögliche Szenarien:

- ✓ Bestätigung des Empfängers geht verloren. Die Nachricht muss nochmals mit der gleichen Sequenznummer gesendet werden. Wenn der Empfänger merkt, dass er die Sequenznummer schon mal gesehen hat, dann verwirft er die Nachricht und schickt nochmals die Bestätigung an den Sender.
- ✓ Request/Nachricht vom Sender geht verloren. Der Sender bekommt dementsprechend keine Bestätigung und weiss, dass er die Nachricht nochmals schicken muss.
- ✓ Servercrash. Im ganzen Prozessablauf werden Rollbackpoints definiert. Die Nachrichten werden nach dem letzten Rollbackpoint nochmals geschickt.
- ✓ Flusskontrolle: Sender und Empfänger müssen sicherstellen, dass bei Überlastung des Senders der Empfänger dies feststellt und die Kommunikation drosselt.

### b) Amnesie-Crash

Beim Amnesiecrash verliert der Server einen Teil der Daten, die er vom Client bekommen hat. Ideal wäre, Rollbackpoints in den Prozessablauf einzufügen. Beim Amnesie-Crash sollen alle Daten nach dem letzten Rollbackpoint nochmals aufgerollt (geschickt) werden.

Folgende Qualitäten können erreicht werden:

- exactly-once: Der Aufwand, dies zu erreichen ist sehr gross. Wir benötigen ein Protokoll, das genau festlegt, was nach dem Checkpoint geschehen ist. Es muss genau festgehalten werden, was sich bereits im Speicher befindet und was nicht.
- at-least-once: Wir setzen z.B. 1 Minute vor dem Crash wieder ein. Dabei ist möglich, dass einzelne Prozesse doppelt ausgeführt werden.
- at-most-once: Der Empfänger wird wieder hochgefahren und fährt an derselben Stelle weiter. Somit werden bestimmt keine Prozesse doppelt ausgeführt, jedoch sind wohl einige verloren gegangen.
- FIFO: Da wir von einem Verlust jeglicher Datenzustände ausgehen zum Zeitpunkt des Absturzes und wir zuvor FIFO hatten, wird auch nach dem Crash FIFO garantiert.

## Aufgabe 4.3

### a) Bearbeitungszeit:

i) minimale Bearbeitungszeit:

$$T = b_A + \max \left( \begin{array}{l} \max(a_{AB} + b_B + a_{BE}, a_{AC} + b_C + a_{CE}) + b_E + a_{EF}, \\ \max(a_{AB} + b_B + a_{BD}, a_{AC} + b_C + a_{CD}) + b_D + a_{DF} \end{array} \right) + b_F$$

D und E sind je von B und C abhängig, also können sie erst starten, wenn B und C abgelaufen sind.

ii) maximale Bearbeitungszeit:

analog zu i), da alle Prozesse voneinander abhängig sind und erst starten können, wenn diejenigen der höheren Stufe abgelaufen sind.

### b) Anzahl beteiligter Knoten:

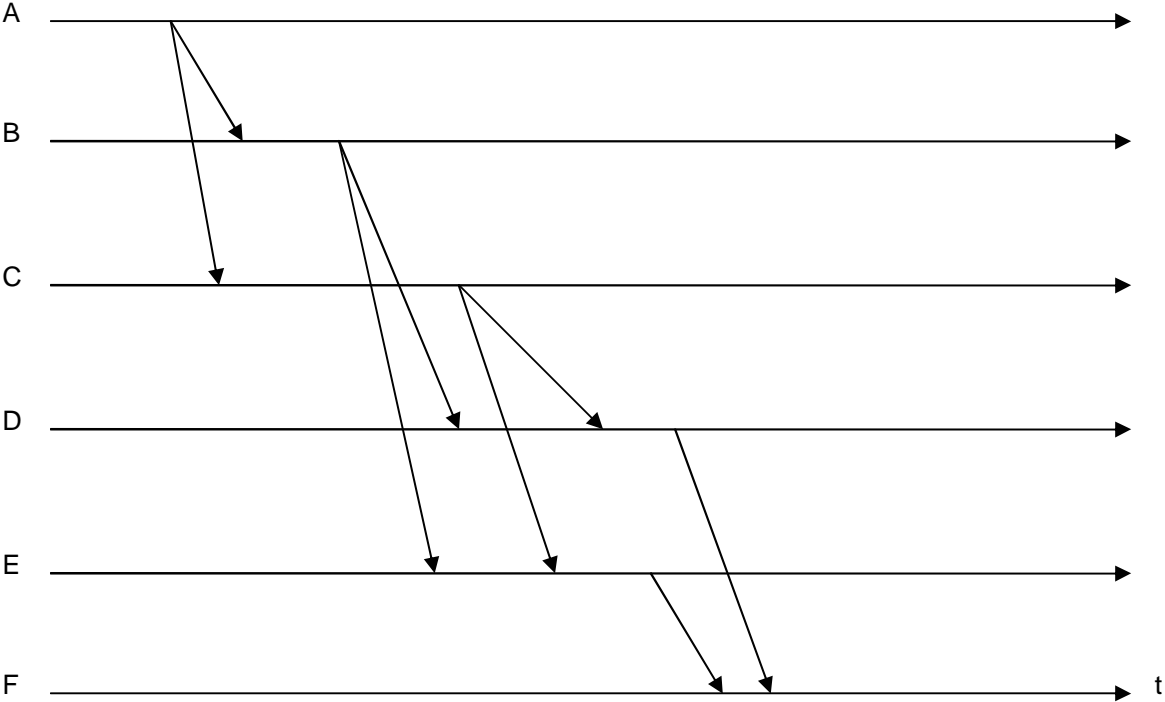
Für die minimale oder die maximale Bearbeitungszeit können immer nur max. 2 Knoten pro Zeitpunkt beteiligt sein. Die minimale Anzahl von beteiligten Knoten ist 1. Das Problem wiederum, dass B und C beide mit D und E kommunizieren, zwingt das System dazu, immer auf beide, B und C, zu warten, bevor E und D gestartet werden können. Diese Blockaden führen dazu, dass höchstens zwei Knoten aktiv sind. Im Falle des Minimums ist nur einer aktiv.

### c) durchschnittliche Belastung der Prozessoren

Annahme: jedes Teilprogramm wird von einem anderen Prozessor berechnet

$$\text{Belastung} = \frac{\left( (b_A + a_{AB} + a_{AC}) + (b_B + a_{AB} + a_{BD} + a_{BE}) + (b_C + a_{AC} + a_{CD} + a_{CE}) + \right. \\ \left. (b_D + a_{BD} + a_{CD} + a_{DF}) + (b_E + a_{BE} + a_{CE} + a_{EF}) + (b_F + a_{DF} + a_{EF}) \right)}{6}$$

**d) Zeit-Raum-Diagramm**



## Aufgabe 4.4

Ideales Netzwerk		
	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$	exactly-once FIFO
a) senden ohne empfangen (Verlust)		
i)	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $send(m:M) \equiv SKIP$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$	at-most-once FIFO
ii)	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$ INTERNAL $drop \equiv msgs \neq \emptyset \Rightarrow msgs := tail(msgs)$	at-most-once FIFO
b) mehrmals empfangen (Duplikate)		
i)	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow SKIP$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$	at-least-once FIFO
ii)	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$ INTERNAL $duplicate \equiv msgs \neq \emptyset \wedge m = last(msgs) \Rightarrow msgs := append(msgs,m)$	at-least-once FIFO
c) "fremde" Mitteilungen		
i)	VAR $msgs: SEQ[M] := \emptyset$ ACTIONS $send(m:M) \equiv msgs := append(msgs,m)$ $receive(m:M) \equiv msgs \neq \emptyset \wedge m = first(msgs) \Rightarrow msgs := tail(msgs)$ $receive(m:M) \equiv m=random()$	exactly-once FIFO

ii)	<p>VAR msgs: SEQ[M] := ∅</p> <p>ACTIONS send(m:M) ≡ msgs := append(msgs,m) receive(m:M) ≡ msgs ≠ ∅ ∧ m = first(msgs) ⇒ msgs := tail(msgs)</p> <p>INTERNAL <b>app-random ≡ m=random() ⇒ msgs:=append(msgs,m)</b></p>	exactly-once FIFO
<b>d) beliebige Reihenfolge</b>		
i)	<p>VAR msgs: SEQ[M] := ∅</p> <p>ACTIONS <b>send(m:M) ≡ msgs := insertrandom(msgs,m)</b> receive(m:M) ≡ msgs ≠ ∅ ∧ m = first(msgs) ⇒ msgs := tail(msgs)</p>	exactly-once pas de FIFO !
ii)	<p>VAR msgs: SEQ[M] := ∅</p> <p>ACTIONS send(m:M) ≡ msgs := append(msgs,m) receive(m:M) ≡ msgs ≠ ∅ ∧ m = first(msgs) ⇒ msgs := tail(msgs)</p> <p>INTERNAL <b>shuffle ≡ length(msgs) &gt; 1 ⇒ msgs := swaprando(msgs)</b></p>	exactly-once pas de FIFO !

## Aufgabe 4.5

### Spezifikation (ideales Netzwerk)

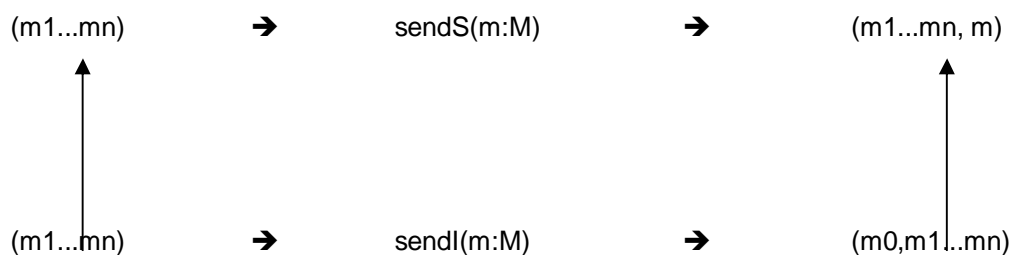
VAR msgS: SEQ[M] := ∅  
ACTIONS  
sendS(m:M) ≡ msgS:=append(msgS,m)  
receiveS(m:M) ≡ msgS ≠ ∅ ∧ m=first(msgS) => msgS:=tail(msgS)

### Implementation (undendlich langer Puffer = Stack)

VAR msgI: STACK[M] := ∅  
ACTIONS  
sendI(m:M) ≡ msgI := push (msgI,m)  
receiveI(m:M) ≡ msgI ≠ ∅ ∧ m=peek(msgI) => msgI:=pop(msgI)

### Abstraktion

f ≡ f: msgI → msgS  
           (m1...mn) → (m1...mn)  
 f ≡ Z<sub>I</sub> → Z<sub>S</sub>



## Aufgabe 4.6

---

Prozess ist bei Überlast **a) fähig, b) unfähig, auf Requests zu reagieren.**

- a) Monitor erhält alle Infos → Reply-Vektor berechnen → Überlastaussage möglich
- b) Monitor erhält keine Antwort → im Prinzip keine Aussage möglich (kann jedoch als Überlast interpretiert werden)

weitere Probleme:

- **temporale Probleme**  
Lastaussage nur möglich zwischen Versand des Multicast-Request und dem Eintreffen des letzten Replies, sonst Fehlschlag
- **unvollständiges Eintreffen der Replies → unklare Aussage**  
Ausnahme:  
66% melden Überlast → Zustand „gefährlich“  
66% melden KEINE Überlast → Zustand „normal“