ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

HS 2014 Prof. Dr. Roger Wattenhofer

# Prüfung
# Verteilte Systeme
# Teil 2

Mittwoch, 21. Januar 2015
9:00 – 12:00

Die Anzahl Punkte pro Teilaufgabe steht jeweils in Klammern bei der Aufgabe. Sie dürfen die Fragen englisch oder deutsch beantworten. Begründen Sie alle Ihre Antworten und beschriften Sie Skizzen und Zeichnungen verständlich. Schreiben Sie zu Beginn Ihren Namen und Ihre Legi-Nummer in das folgende dafür vorgesehene Feld.

| Name | Legi-Nr. |
|---|---|
| | |

## Punkte

| Frage Nr. | Erreichte Punkte | Maximale Punkte |
|---|---|---|
| 9 | | 25 |
| 10 | | 16 |
| 11 | | 17 |
| 12 | | 17 |
| 13 | | 15 |
| Total | | 90 |

# 9 Multiple Choice (25 Punkte)

Beurteilen Sie, ob die folgenden Aussagen richtig oder falsch sind, und kreuzen Sie die entsprechenden Felder an. Eine richtig beurteilte Aussage gibt 1 Punkt, eine nicht beurteilte Aussage 0 Punkte, eine nicht richtig beurteilte Aussage **-1 Punkt**. Die gesamte Aufgabe wird mit minimal 0 Punkten bewertet.

**A)** Consensus & Consistency

| **Aussage** | **Wahr** | **Falsch** |
|---|---|---|
| Jeder Algorithmus, der Compare-And-Swap (CAS) verwendet, hat eine Konsensusnummer von $\infty$. | ☐ | ☐ |
| Wenn man die Test-And-Set (TAS) Primitive 6-mal verwendet, kann man Konsensus mit 3 Prozessen erreichen. | ☐ | ☐ |
| Wenn man 16 Register gleichzeitig schreiben kann, kann man Konsensus für 4 Prozesse lösen. | ☐ | ☐ |
| Jeder korrekte Algorithmus, welcher Konsensus mit *byzantinischen* Fehlern löst, entdeckt auch welche Prozesse byzantinisch sind. | ☐ | ☐ |
| Anstatt Konsensus mit allen Servern zu lösen, können Quorum Systeme verwendet werden, um die Effizienz des Systems zu steigern. | ☐ | ☐ |
| Paxos benötigt keinen Primary der den Ablauf des Protokolls koordiniert. | ☐ | ☐ |
| Quorum Systeme mit Quoren, welche komplett disjunkt sind, sind besonders effizient, weil verschiedene Quoren gleichzeitig arbeiten können. | ☐ | ☐ |
| PBFT mit $n$ Teilnehmern toleriert bis zu $f = \lfloor \frac{n-1}{2} \rfloor$ crash failures. | ☐ | ☐ |
| Chubby implementiert Coarse-Grained Locking, das heisst es kann nur wenige Locks verwalten. | ☐ | ☐ |
| Bitcoin implementiert Eventual Consistency, d.h. Daten sind jederzeit in einem konsistenten Zustand. | ☐ | ☐ |

**B)** Locking

| **Aussage** | **Wahr** | **Falsch** |
|---|---|---|
| Um ein threadsicheres Lock implementieren zu können, wird mindestens eine $RMW$-Operation benötigt. | ☐ | ☐ |
| Test-And-Set-Locks (TAS) sind immer performanter als Test-And-Test-And-Set-Locks (TTAS), da weniger Operationen benötigt werden. | ☐ | ☐ |
| Test-And-Set-Locks sind nicht fair. | ☐ | ☐ |
| Mit Consensus-Algorithmen können Sie Mutual Exclusion (gegenseitigen Ausschluss) erreichen. | ☐ | ☐ |
| Mehrere Prozesse auf eine Änderung derselben Speicherzelle warten zu lassen ist weniger effizient als für jeden Prozess eine separate Speicherzelle zu verwenden. | ☐ | ☐ |

**C)** Game Theory

| **Aussage** | **Wahr** | **Falsch** |
|---|---|---|
| Wenn sich alle Spieler selfish verhalten, ist das Ergebnis des Spiels nie ein soziales Optimum, da Kooperation immer besser ist. | ☐ | ☐ |
| Wenn der Zustand eines Spiels ein soziales Optimum ist, so ändert kein Spieler mehr seine Entscheidung, weil sonst das Resultat insgesamt, d.h. auch für ihn, schlechter wird. | ☐ | ☐ |
| Das Spiel Stein-Schere-Papier hat kein reines Nash-Gleichgewicht. | ☐ | ☐ |
| In einer First-Price-Auction lohnt es sich manchmal *nicht* die Wahrheit zu sagen. | ☐ | ☐ |
| Wenn ein Spieler mindestens eine dominante Strategie hat, sollte er stets eine davon spielen. | ☐ | ☐ |

**D)** Network Updates

| **Aussage** | **Wahr** | **Falsch** |
|---|---|---|
| Die Hauptidee von *Software Defined Networks* (SDNs) beschreibt das Simulieren von Netzwerken in Software. | ☐ | ☐ |
| Wenn die Hälfte aller Kanten in einem Netzwerk einen Slack von 50% hat, dann kann immer ein *Capacity-Consistent Update* durchgeführt werden. | ☐ | ☐ |
| In *SDNs* kann in polynomieller Zeit vom Controller bestimmt werden, ob das Updaten einer Forwarding-Regel einen *Loop* erzeugt. | ☐ | ☐ |
| Beim Updaten von Präfix-basierten Forwarding-Regeln kann *Loop-Freedom* immer garantiert werden, wenn man die Präfix-Regeln temporär in beliebig kleine Präfixe auftrennen darf. | ☐ | ☐ |
| Durch Versions-Nummern kann in *Software Defined Networks* die Paket-Grösse verkleinert werden. | ☐ | ☐ |

| **Aussage** | **Wahr** | **Falsch** |
|---|---|---|
| Jeder Algorithmus, der Compare-And-Swap (CAS) verwendet, hat eine Konsensusnummer von $\infty$.<br>*Reason: Man kann das natuerlich ganz falsch anwenden... und Konsensusnr. 1 haben.* | | ✓ |
| Wenn man die Test-And-Set (TAS) Primitive 6-mal verwendet, kann man Konsensus mit 3 Prozessen erreichen.<br>*Reason: TAS hat Konsensusnummer 2.* | | ✓ |
| Wenn man 16 Register gleichzeitig schreiben kann, kann man Konsensus für 4 Prozesse lösen.<br>*Reason: Wie write(6). Im Allgemeinen kann man mit $n(n+1)/2(+1)$ Konsensus für $n$ Register lösen.* | ✓ | |
| Jeder korrekte Algorithmus, welcher Konsensus mit *byzantinischen* Fehlern löst, entdeckt auch welche Prozesse byzantinisch sind.<br>*Reason: Das ist nicht nötig, die meisten Algorithmen der Vorlesung machen das nicht.* | | ✓ |
| Anstatt Konsensus mit allen Servern zu lösen, können Quorum Systeme verwendet werden, um die Effizienz des Systems zu steigern.<br>*Reason:* | ✓ | |
| Paxos benötigt keinen Primary der den Ablauf des Protokolls koordiniert.<br>*Reason:* | ✓ | |
| Quorum Systeme mit Quoren, welche komplett disjunkt sind, sind besonders effizient, weil verschiedene Quoren gleichzeitig arbeiten können.<br>*Reason: Das wäre gar kein gültiges Quorumsystem.* | | ✓ |
| PBFT mit $n$ Teilnehmern toleriert bis zu $f = \lfloor \frac{n-1}{2} \rfloor$ crash failures.<br>*Reason: PBFT kann nicht mit $f > n/3$ funktionieren, egal ob crash oder byzantine. Weil $n/3 < f < n/2$ existieren kann stimmt das nicht. Alternativ: bei $n/2$ crash failures werden die $2f + 1$ thresholds nicht mehr getriggert.* | | ✓ |
| Chubby implementiert Coarse-Grained Locking, das heisst es kann nur wenige Locks verwalten.<br>*Reason: Coarse wrt time.* | | ✓ |
| Bitcoin implementiert Eventual Consistency, d.h. Daten sind jederzeit in einem konsistenten Zustand.<br>*Reason: Inkonsistenzen können bestehen bis diese aufgelöst werden.* | | ✓ |
| Um ein threadsicheres Lock implementieren zu können, wird mindestens eine *RMW*-Operation benötigt.<br>*Reason: Nope.* | | ✓ |
| Test-And-Set-Locks (TAS) sind immer performanter als Test-And-Test-And-Set-Locks (TTAS), da weniger Operationen benötigt werden.<br>*Reason: Caching makes TTAS more efficient.* | | ✓ |
| Test-And-Set-Locks sind nicht fair.<br>*Reason: Yup.* | ✓ | |
| Mit Consensus-Algorithmen können Sie Mutual Exclusion (gegenseitigen Ausschluss) erreichen.<br>*Reason: Jeder stimmt fürsich selbst, einer gewinnt.* | ✓ | |
| Mehrere Prozesse auf eine Änderung derselben Speicherzelle warten zu lassen ist weniger effizient als für jeden Prozess eine separate Speicherzelle zu verwenden.<br>*Reason: Verursacht eher einen "Invalidation Storm", wenn sich der Inhalt der Speicherzelle ändert.* | ✓ | |
| Wenn sich alle Spieler selfish verhalten, ist das Ergebnis des Spiels nie ein soziales Optimum, da Kooperation immer besser ist.<br>*Reason: blubb* | | ✓ |
| Wenn der Zustand eines Spiels ein soziales Optimum ist, so ändert kein Spieler mehr seine Entscheidung, weil sonst das Resultat insgesamt, d.h. auch für ihn, schlechter wird. | | ✓ |

| | | |
|---|---|---|
| *Reason: Schön wärs! Es kann sich immer noch ein Spieler individuell verbessern* | | |
| Das Spiel Stein-Schere-Papier hat kein reines Nash-Gleichgewicht.<br>*Reason: 1/3, 1/3, 1/3* | ✓ | |
| In einer First-Price-Auction lohnt es sich manchmal *nicht* die Wahrheit zu sagen.<br>*Reason: wenn ich hoechstens valaution habe* | ✓ | |
| Wenn ein Spieler mindestens eine dominante Strategie hat, sollte er stets eine davon spielen.<br>*Reason: deswegen sind sie dominant* | ✓ | |
| Die Hauptidee von *Software Defined Networks* (SDNs) beschreibt das Simulieren von Netzwerken in Software.<br>*Reason: Neee* | | ✓ |
| Wenn die Hälfte aller Kanten in einem Netzwerk einen Slack von 50% hat, dann kann immer ein *Capacity-Consistent Update* durchgeführt werden.<br>*Reason: Das reicht nicht aus* | | ✓ |
| In *SDNs* kann in polynomieller Zeit vom Controller bestimmt werden, ob das Updaten einer Forwarding-Regel einen *Loop* erzeugt.<br>*Reason: Ja, mit BFS/DFS.* | ✓ | |
| Beim Updaten von Präfix-basierten Forwarding-Regeln kann *Loop-Freedom* immer garantiert werden, wenn man die Präfix-Regeln temporär in beliebig kleine Präfixe auftrennen darf.<br>*Reason: Ja, dann ist es single - das geht.* | ✓ | |
| Durch Versions-Nummern kann in *Software Defined Networks* die Paket-Grösse verkleinert werden.<br>*Reason: Nein, wird groesser* | | ✓ |

# 10 Quorum Systeme (16 Punkte)

In dieser Aufgabe betrachten wir ein System bestehend aus einer Menge von Servern. Die Menge der Server ist $\mathbb{P} = \{P_1, \ldots, P_n\}$, und die Anzahl Server ist $|\mathbb{P}| = n$, wobei wir annehmen, dass $n > 3$ eine gerade Zahl ist.

***Definition* (Gruppe)** Eine *Gruppe* $\mathcal{G}$ ist eine Menge aus exakt $n/2 + 1$ Servern.
Formal: $\mathcal{G} \subset \mathbb{P}$ und $|\mathcal{G}| = n/2 + 1$.

***Beispiel*** Wenn $n = 4$ ist, dann ist $\mathcal{G} = \{P_1, P_2, P_3\}$ eine mögliche Gruppe.

In den folgenden Aufgaben versuchen wir ein Quorum System zu bauen. Als Quoren verwenden wir *Gruppen*, wobei wir jeweils $g$ verschiedene Gruppen als Quoren auswählen. Beachte dass alle Gruppen paarweise verschieden sein müssen, d.h. $\forall \mathcal{G}, \mathcal{G}' : \mathcal{G} \neq \mathcal{G}'$

**A)** (4 Punkte) Ist ein System, welches nur aus *einer einzigen* Gruppe besteht (d.h. $g = 1$), ein valides Quorum System?

**Falls nein:** Begründe warum es kein valides Quorum System ist.

**Falls ja:** Erkläre die Funktion derjenigen Server, welche nicht in der Gruppe (d.h. nicht im Quorum) sind.

**B)** (3 Punkte) Was ist die kleinste Anzahl an Gruppen $g_{\min}$, so dass für jede Auswahl von $g_{\min}$ verschiedenen Gruppen immer ein gültiges Quorum System entsteht? Was ist die grösste Anzahl $g_{\max}$?

**C)** (9 Punkte) Für diese Aufgabe betrachten wir ein System mit $g$ verschiedenen Gruppen, so dass $g_{\min} \leq g \leq g_{\max}$ gilt. Nehmen Sie an, dass für jede Auswahl von $g$ Gruppen ein valides Quorum System entsteht.

Betrachten wir nun die Systemeigenschaften *work*, *load* und *resilience*, wie sie in der Vorlesung definiert wurden.

Sind diese Eigenschaften *unabhängig* davon, *wie* die $g$ Gruppen konstruiert werden oder hängen sie von der Konstruktion der Gruppen ab?

**Falls unabhängig:** Geben Sie eine Formel (mit $g$ und $n$) für die Eigenschaft an.

**Falls abhängig:** Begründen Sie warum die Konstruktion der Gruppen den Wert beeinflusst. Wählen Sie z.B. ein $g$ und zeigen Sie, dass der Wert für zwei verschiedene Konstruktionen unterschiedlich ist.

**Beantworten Sie diese Frage für jede der drei Eigenschaften einzeln!**

Solutions

**A)** Ja, es ist ein gültiges Quorum System. (1 Punkt)

Die Funktion der anderen Server ist, die Daten zu "replizieren", d.h. sie sind *silent observers*. Sie sind sozusagen passive Datenspeicher, welche sich nicht an der Entscheidungsfindung beteiligen, sondern einfach nur die Daten zur Erhöhung der Redundanz speichern. (3 Punkte)

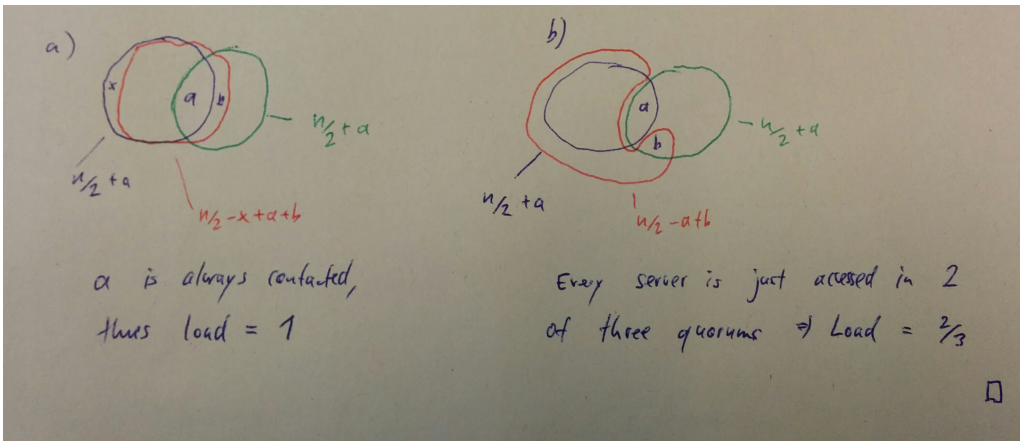
**B)** $g_{\min} = 1$, das haben wir schon in A) gezeigt. (0.5 Punkte für Anzahl, 0.5 Punkte für Referenz auf A), oder kurze Begründung analog zu A))

$g_{\max} = \binom{n}{n/2+1}$, da dies die maximale Anzahl an verschiedenen Gruppen ist. Weil sich alle Gruppen nach Konstruktion überlappen, ist es auch ein gültiges Quorum System. (1 Punkt Anzahl, 1 Punkt Begründung)

**C)** **Work** Unabhängig. Jedes mal müssen $n/2+1$ Server kontaktiert werden, d.h. work $= n/2+1$.

**Resilience** Abhängig von der Konstruktion. Im Gegenbeispiel a) ist die Resilience 0, weil sobald Server $a$ crasht kein Quorum mehr exisitert. In der Konstruktion b) ist die Resilience 1, weil kein Server in allen Quoren vorkommt, und man mind. 2 Server crashen muss.

**Load** Abhängig von der Konstruktion. Im Gegenbeispiel a) ist die load 1, weil $a$ immer kontaktiert wird. In b) ist die load 2/3, weil jeder Server nur in 2 von 3 Quoren vorkommt.




(Je 1 Punkt für abhänging/unabhängig, 2 Punkte für Begründung. Bei Gegenbeispiel (Resilience, Load) 1 Punkt für das Beispiel, 1 Punkt für Erklärungssatz zum Beispiel. Das Beispiel muss nicht gezeichnet werden, es kann auch in textuell erklärt sein)

# 11 Replicated Locking (17 Punkte)

Ihr Kollege hat die Aufgabe bekommen, einen FIFO-fairen Locking-Dienst zu implementieren, der ausserdem Mutual Exclusion (gegenseitigen Ausschluss) und Liveness garantiert. (*Liveness* bedeutet, dass jeder Prozess, der das Lock anfordert, das Lock früher oder später auch erhält.) Dazu hat er als Basis eine threadsichere Queue verwendet. Nachdem er das Lock implementiert hat, bittet Ihr Kollege Sie um Hilfe, denn es funktioniert nicht wie gewünscht.

Die Queue (Warteschlange) verwaltet eine Liste, auf die mit den folgenden 2 atomaren Operationen zugegriffen werden kann:

- ENQUEUE(Wert) hängt den gegebenen Wert hinten an die Liste an.
- DEQUEUE() gibt den Wert des ersten Elements zurück und entfernt es aus der Liste.

Wir gehen davon aus, dass Prozesse nur Crash-Failures haben können und dass, während ein Prozess das Lock hält, dieser Prozess nicht crasht.

**Algorithm 1** Faires Queue-Lock

```
1:  procedure FetchFirst( )                ▷ Hilfsfunktion zum Abrufen des ersten Elements
2:      p ← Dequeue()
3:      Enqueue(p)
4:      return p
5:  end procedure
6:
7:  procedure Lock(p)                      ▷ Prozess p will das Lock bekommen.
8:      Enqueue(p)
9:      while FetchFirst() ≠ p do
10:         Sleep()                        ▷ Vor dem nächsten Versuch kurz abwarten.
11:     end while
12: end procedure
13:
14: procedure Unlock(p)                    ▷ Prozess p besitzt das Lock und möchte es abgeben.
15:     Dequeue()
16: end procedure
```

**A)** (9 Punkte) Das in Algorithm 1 beschriebene Lock verhält sich nicht korrekt. Welche der 3 Eigenschaften *Mutual Exclusion*, *Liveness* und *FIFO-Fairness* werden verletzt?

**Falls erfüllt:** Erläutern Sie weshalb die Eigenschaft erfüllt ist.

**Falls verletzt:** Geben Sie eine verletzende Ausführung an und erläutern Sie das Problem.

**B)** (5 Punkte) Um das Lock zu reparieren, wird der Queue eine neue Operation *Peek()* hinzugefügt, die atomar das erste Element zurückgibt, ohne die Liste zu modifizieren. Der Aufruf an *FetchFirst()* wird durch einen an *Peek()* ersetzt. Welche der 3 Eigenschaften erfüllt das Queue-Lock nun?

**Falls erfüllt:** Erläutern Sie weshalb die Eigenschaft jetzt erfüllt ist.

**Falls verletzt:** Geben Sie eine verletzende Ausführung an und erläutern Sie das Problem.

**C)** (3 Punkte) Da der Locking-Dienst ein wichtiger Teil der Infrastruktur ist, soll er gegen Ausfälle gesichert werden. Der einzelne Server, auf dem der Dienst bisher lief, soll durch 5 Server ersetzt werden, die gemeinsam die Queue verwalten. Gehen Sie diesmal davon aus, dass die Prozesse, die sich in die Queue eintragen, niemals crashen. Geben Sie ein Protokoll zur Koordination der Server an, das auch bei Serverausfällen (Crashes) möglichst viele der 3 Eigenschaften erfüllt.

**A)** Alle 3 Eigenschaften werden verletzt:

**Gegenseitiger Ausschluss** Beispielausführung:

- Prozess 1 führt Zeilen 8, 2, 3, 9 aus und verlässt *Lock(p)* sofort. Queue: [1].
- Prozess 2 führt Zeilen 8, 2, 3, 9, 10, 2, 3, 9 aus und verlässt *Lock(p)* nach einem Schleifendurchlauf. Queue: $[1, 2]$.

**Liveness** Beispielausführung:

- Prozess 1 führt Zeilen 8, 2, 3, 9 aus (nimmt das Lock). Queue: [1].
- Prozess 2 führt Zeilen 8, 2, 3, 9, 10 aus. Queue: $[2, 1]$.
- Prozess 1 führt Zeile 15 aus (gibt das Lock ab). Queue: [1].
- Prozess 2 führt Zeilen 2, 3, 9, 10 immer wieder aus, bis er schwarz wird. Queue: [1].

**Fairness** Beispielausführung:

- Prozess 1 führt Zeilen 8, 2, 3, 9 aus (nimmt das Lock). Queue: [1].
- Prozess 2 führt Zeilen 8, 2, 3, 9, 10 aus. Queue: $[2, 1]$.
- Prozess 1 führt Zeile 15 aus (gibt das Lock ab). Queue: [1].
- Prozess 2 führt Zeilen 2, 3, 9, 10 immer wieder aus, bis er schwarz wird. Queue: [1].
- Prozess 3 führt Zeilen 8, 2, 3, 9, 10, 2, 3, 9 aus (nimmt das Lock). Queue: $[1, 3]$.
- Prozess 3 führt Zeile 15 aus (gibt das Lock ab). Queue: [3].
- Wiederhole mit beliebig vielen Prozessen, die alle das Lock kriegen; nur Prozess 2 erhält das Lock nie – unfair!

**FIFO-Fairness** Beispielausführung: (FIFO-Fairness zu widerlegen ist noch einfacher.)

**B)** Das Queue-Lock funktioniert nun schon viel besser:

**Gegenseitiger Ausschluss** Das Lock ist stets im Besitz des Prozesses, dessen ID in der Queue vorne steht. Ist die Queue leer, ist das Lock frei.

**Liveness** Crasht ein Prozess, der sich in die Queue eingetragen hat (aber noch nicht an der Reihe ist bzw. das Lock hält), so wird kein Prozess die ID des gecrashten Prozesses je aus der Queue entfernen. Somit wird kein Prozess, der sich nach dem gecrashten in die Queue einträgt, das Lock je erhalten.

**Fairness** Sofern es keine Crashs gibt, bekommt jeder Prozess, der es anfordert, das Lock, und zwar beliebig oft.

**FIFO-Fairness** Ebenfalls gegeben, sofern es keine Crashs gibt. Die Prozesse erhalten das Lock in genau der Reihenfolge, in der sie Zeile 8 in *Lock(p)* ausführen (allerdings nicht unbedingt in der Reihenfolge, in der sie *Lock(p)* betreten).

**C)** Three-Phase Commit erfüllt alle 3 Eigenschaften. Two-Phase Commit und Paxos erfüllen Liveness nicht, aber die anderen beiden.

# 12 Game Theory & SDN (17 Punkte)

Gegeben sei das folgende Netzwerk aus Abbildung 1. Jeder der Knoten $a, b, c$ und $d$ möchte Daten zu Knoten $e$ schicken. Betrachten Sie die Knoten $a, b, c$ und $d$ als Nutzer. Das Verschicken der Daten ist nicht kostenlos, sondern der $i$te Nutzer einer Kante (unabhängig davon in welche Richtung die Kante von den anderen Nutzern verwendet wird) muss für die Benutzung $i$ zahlen (die Kosten von Nutzern, die diese Kante bereits benutzen, wird durch weitere Nutzer nicht verändert). Jeder Nutzer muss für alle Kanten auf dem Pfad, auf dem er die Daten verschickt, zahlen und will seine Kosten minimieren. Die Daten eines Nutzers können *nicht* auf mehrere Pfade verteilt werden. Die Reihenfolge in der die Nutzer entscheiden, auf welchem Pfad sie die Daten verschicken, ist stets $a, b, c$ und dann $d$.

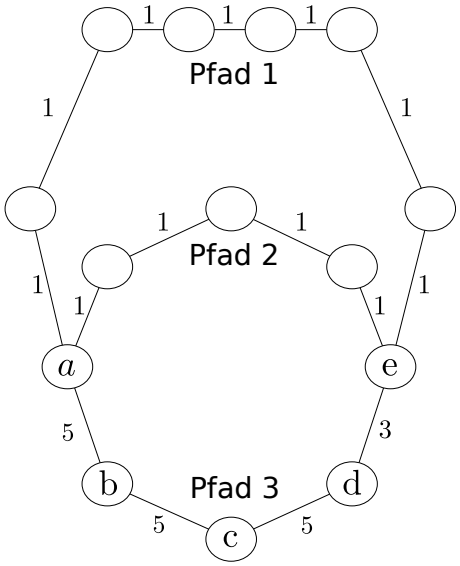



Abbildung 1: Netzwerk mit den Knoten $a, b, c, d, e$ und weiteren Knoten. Die Zahlen an den Kanten geben die Bandbreite der Kanten an.

Wir bezeichnen den oberen Pfad als Pfad 1, den mittleren Pfad als Pfad 2 und entsprechend den unteren Pfad als Pfad 3.

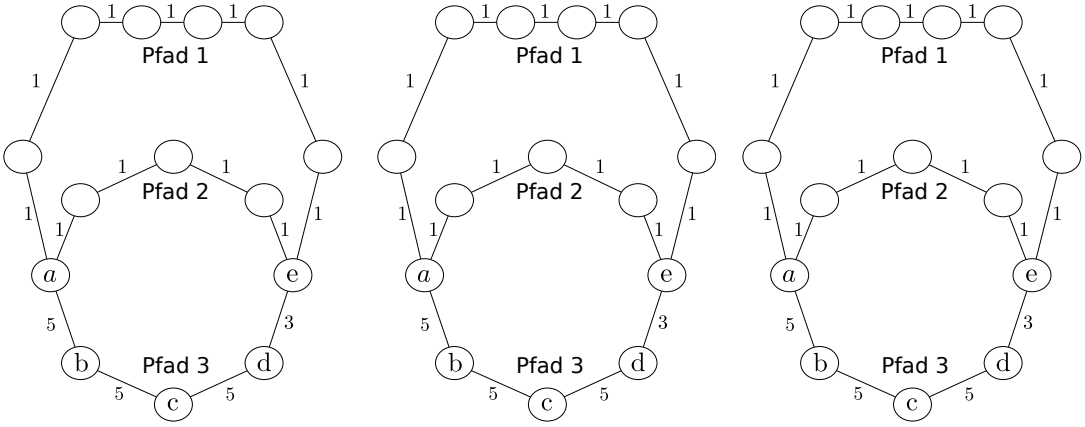
**Hinweis:** Sie können, müssen aber nicht, Ihre Lösung in die Graphen auf einer der folgenden Seiten einzeichnen.

**A)** (7 Punkte) Bestimmen Sie alle Nash-Gleichgewichte, ein Social Optimum und den Price of Anarchy.

**B)** (3 Punkte) Begründen oder widerlegen Sie die folgende Aussage: Wenn jeder Spieler seine Kosten minimieren will und die Spieler in einer fixen Reihenfolge erscheinen, dann ist das Resultat stets ein Nash-Gleichgewicht.

**C)** (4 Punkte) Nehmen Sie an, das Netzwerk sei ein SDN und Sie seien der Controller. Knoten $a, b, c$ schicken ihre Daten über Pfad 3, und Knoten $d$ über Pfad 2. Geben Sie eine konsistente Möglichkeit an (d.h. ohne Verletzung der Bandbreiten der Kanten), wie man zur folgenden Konfiguration migrieren kann: Knoten $a, b, d$ über Pfad 3, und Knoten $c$ über Pfad 2.

**D)** (3 Punkte) Nehmen Sie erneut an, das Netzwerk sei ein SDN und Sie seien der Controller. Die Knoten $a, b, c$ schicken ihre Daten immer noch über Pfad 3. Vom Knoten $d$ schicken jedoch nun *zwei* Nutzer ihre Daten: Einer über Pfad 1 und einer über Pfad 2. Begründen oder widerlegen Sie, ob Sie nun als SDN-Controller die Pfade ausgehend von den Knoten $a, b, c, d$ ändern können.

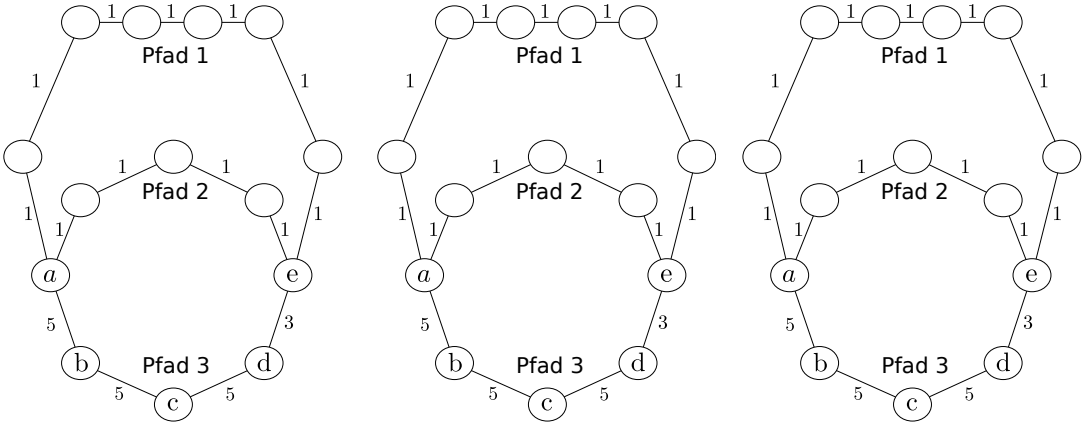



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Abbildung 2: Kopien des Graphen




(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Abbildung 3: Kopien des Graphen

Solutions

**A)** Es gibt 3 Nash-Gleichgewichte. Das erste lautet: Knoten $a$ entscheidet sich für Pfad 2, die Knoten $b, c, d$ verschicken ihre Daten über Pfad 3. Die Kosten sind $4 + 3 + 4 + 3 = 14$. Dies ist ebenfalls das Social Optimum.

Das zweite lautet: Knoten $a$ entscheidet sich für Pfad 3, die Knoten $b, c$ verschicken ihre Daten über Pfad 3, Knoten $d$ über Pfad 2. Die Kosten sind $4 + 6 + 6 + 13 = 29$.

Das dritte lautet: Knoten $a$ entscheidet sich für Pfad 3, der Knoten $b$ verschickt seine Daten über Pfad 2, Knoten $c, d$ über Pfad 3. Die Kosten sind $4 + 6 + 4 + 3 = 17$.

Damit ergibt sich ein Price of Anarchy von 29/14.

**B)** Die Behauptung ist wahr. Wir zeigen sie per Induktion über die Reihenfolge der Spieler. Die Induktionsbasis gilt, weil Spieler 1 sich zu Beginn in einem leeren Netzwerk für einen für ihn optimalen Pfad ausgewählt hat. Nun sind einige Kanten belegt, d.h. seine Kosten können nur steigen. Die Induktionshypothese lautet, dass keiner der ersten $i-1$ Spieler seine Strategie wechselt. Wir zeigen nun den Induktionsschritt von $i-1$ zu $i$. Da keiner der ersten $i-1$ Spieler gewechselt hat (I.H.) und Spieler $i$ als nur diese Spieler ihre Entscheidung schon getroffen haben, sich für ihn optimalen Pfad entschiedet hat, können sich seine Kosten nicht verringern, wenn er nun wechselt.

**C)** Eine Möglichket die Konfiguration zu migrieren lautet: Knoten $c$ verschickt über Pfad 1, dann erst kann Knoten $d$ über Pfad 3, und anschliessend Knoten $c$ über Pfad 2.

**D)** Das Netzwerk hat keinerlei Slack auf den Kanten inzident zu $e$ (nur $e$ ist ein Sink für alle Flüsse), d.h. eine konsistente Migrierung ist nicht möglich.

# 13 Clock Sync (15 Punkte)

In dieser Aufgabe analysieren und optimieren Sie ein bestehendes System für Clock Synchronization. Das System besteht aus $n$ Knoten, die wie eine Kette verbunden sind. Jeder Knoten besitzt eine eindeutige ID zwischen 1 und $n$ und Kommunikation ist jeweils möglich zwischen Knoten $i$ und $i+1$ für $i \in [1, n-1]$, in beiden Richtungen. Jeder Knoten besitzt eine eigene Clock, die mit $r_i = \pm 1000$ ppm driftet. Alle Clocks sind zu Beginn synchronisiert. Die Knoten können mittels `send(message, nodeId)` Nachrichten gezielt an einen ihrer Nachbarn mit ID=`nodeId` senden. Die durch die Übertragung entstehende Verzögerung $d$ verhält sich zufällig und ist in 80% der Fälle exakt 1ms und ansonsten länger als 1ms. Es wurden sogar schon Verzögerungen von 1s gemessen.

Im bestehenden Algorithmus sendet Knoten 1 alle 100 Sekunden seinen aktuellen Timestamp an Knoten 2. Wenn ein Knoten $i > 1$ einen Timestamp $t$ empfängt, dann wird folgende Funktion ausgeführt:

```
1: tLocal = aktueller Timestamp der lokalen Hardware Clock
2: function onTimestampReceived(timestamp)
3:     tLocal = timestamp
4:     send(tLocal, i + 1)
5: end function
```

**A)** (6 Punkte) Beschreiben Sie mindestens zwei Probleme des bestehenden Algorithmus.

**B)** (4 Punkte) Einer Ihrer Kollegen schlägt die folgende Änderung vor:

```
1: tLocal = aktueller Timestamp der lokalen Hardware Clock
2: function onTimestampReceived(timestamp)
3:    tLocal = max(tLocal,timestamp)
4:    send(tLocal, i + 1)
5: end function
```

Der neue Algorithmus wird in zwei Netzwerken (Topologie wie oben beschrieben, gleiche Anzahl Knoten, gleiche Verzögerungen) getestet. In einem der Netzwerke funktioniert der neue Algorithmus besser als der alte Algorithmus. Im anderen Netzwerk hingegen funktioniert der alte Algorithmus besser. Nennen Sie mögliche Gründe für das beobachtete Verhalten.

**C)** (5 Punkte) Wie würden Sie das Clock Synchronization Problem in dieser Situation lösen (ohne zusätzliche Hardware wie GPS)? Was sind die Vorteile Ihrer Lösung?

Solutions

**A)** Errors that are introduced on one link will propagate throughout the rest of the chain of nodes. The clocks of the nodes with higher ids will jump around a lot since the propagation delay is most likely arbitrary. There will be systematic errors since the transmission always takes at least 1 ms.

**B)** If node 1 has the fastest clock, this will work well since large transmission delays are countered by using the max function. However, if the fastest clock is not in the first node, clocks farther down the chain might never synchronize to it because tLocal ¿ timestamp because of the drift. (one can see this for $n = 2$ already)

**C)** Synchronize through request-response (ntp style) scheme. If the round trip is 2ms $\pm 2\mu$s we can synchronize very accurately knowing that the roundtrip is near perfectly symmetrical. After that, we can synchronize the next node. This means that clocks at the end of the chain will be much more accurate and less jumpy since we have an upper bound on the synchronization error between two nodes. Also we get rid of the systematic error introduced by the ¿=1ms propagation delay.