



HS 2010

Prof. Dr. Roger Wattenhofer
B. Keller, M. Kuhn, S. Welten, R. Eidenbenz

Prüfung Verteilte Systeme

Freitag, 11. Februar 2011
9:00 – 12:00**Nicht öffnen oder umdrehen bevor die Prüfung beginnt!**

Die Prüfung dauert 180 Minuten und es gibt insgesamt 180 Punkte. Die Anzahl Punkte pro Teilaufgabe steht jeweils in Klammern bei der Aufgabe. Sie dürfen die Prüfung 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
1		62
2		44
3		31
4		43
Total		180

1 Multiple Choice (20 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) Consistency und Commit Protokolle

Aussage	wahr	falsch
Eine real time partial order einer Execution definiert für jedes Paar von Operationen dieser Execution, welche von beiden Operationen früher beginnt.	<input type="checkbox"/>	<input type="checkbox"/>
Eine Execution, in welcher für jeden Client eine client partial order definiert ist, ist immer linearisierbar.	<input type="checkbox"/>	<input type="checkbox"/>
Eine Execution die sequentially consistent ist, ist immer linearisierbar.	<input type="checkbox"/>	<input type="checkbox"/>
Wenn nur ein Knoten, der am Two-phase commit Protokoll beteiligt ist, Byzantinisch ist, funktioniert das Protokoll immer noch.	<input type="checkbox"/>	<input type="checkbox"/>

B) Konsensus

Aussage	wahr	falsch
Wenn ein univalenter Zustand im Ausführungsbaum (execution tree) erreicht ist, kennen alle Prozesse die Entscheidung.	<input type="checkbox"/>	<input type="checkbox"/>
Wenn ein Algorithmus wechselseitigen Ausschluss (mutual exclusion) lösen kann, kann er auch Konsensus lösen.	<input type="checkbox"/>	<input type="checkbox"/>
Ein Algorithmus welcher bis zu f byzantinische Prozesse toleriert, funktioniert immer auch mit $2f$ Ausfällen (crash failures).	<input type="checkbox"/>	<input type="checkbox"/>
Der King Algorithmus toleriert zwar mehr Byzantiner als der Queen Algorithmus, benötigt aber auch mehr Runden.	<input type="checkbox"/>	<input type="checkbox"/>
Wenn wir das RMW-Register x dazu verwenden können das RMW-Register y zu simulieren und y hat Konsensus-Nummer c , dann hat x auch genau Konsensus-Nummer c .	<input type="checkbox"/>	<input type="checkbox"/>
Ein RMW Register heisst trivial, wenn kein Wert v existiert, so dass $v \neq f(v)$	<input type="checkbox"/>	<input type="checkbox"/>

C) Selfish Peers & P2P.

Aussage	wahr	falsch
Jedes Spiel besitzt mindestens ein pures Nash Equilibrium.	<input type="checkbox"/>	<input type="checkbox"/>
Ein P2P-System mit Price of Anarchy gleich 2 funktioniert immer noch mindestens halb so gut, wenn sich alle Peers eigennützig (=selfish) verhalten, wie wenn sich alle Peers kooperativ verhalten. Nehmen Sie an, dass die eigennützigsten Peers keine Koalitionen bilden.	<input type="checkbox"/>	<input type="checkbox"/>
In einem Spiel G mit zwei Spielern lässt es sich in $O(k_1^2 k_2 + k_2^2 k_1)$ Rechenschritten entscheiden, ob G ein dominantes Strategieprofil besitzt, wobei k_i die Anzahl verfügbarer Strategien von Spieler i seien.	<input type="checkbox"/>	<input type="checkbox"/>
In einem DHT Search Tree, bei dem neue Peers jeweils eine Random ID wählen, ist die benötigte Anzahl zu schickender Messages für eine Query garantiert in $O(\log n)$, wobei n die Anzahl Peers zum Zeitpunkt der Anfrage ist.	<input type="checkbox"/>	<input type="checkbox"/>
Die Netzwerktopologie in BitTorrent Schwärmen ist abgesehen von kleinen Abweichungen die gleiche wie jene in Chord.	<input type="checkbox"/>	<input type="checkbox"/>
Es kann keine DHT geben, in der die Suche nach einem Key garantiert weniger als $\Theta(\log n)$ Messages benötigt, wobei n die Anzahl Peers in der DHT ist.	<input type="checkbox"/>	<input type="checkbox"/>

D) Locking

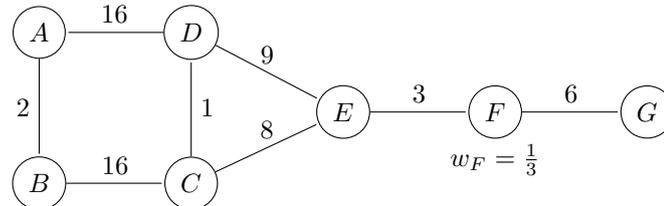
Aussage	wahr	falsch
In Multi-Prozessor Systemen belasten Test&Test&Set (TTAS) Locks die Cache Coherence Mechanismen.	<input type="checkbox"/>	<input type="checkbox"/>
Anderson Queue Locks (ALock) funktionieren besonders gut, wenn die Anzahl Threads unbekannt ist!	<input type="checkbox"/>	<input type="checkbox"/>
MCS Locks sind gut geeignet bei der Verwendung vieler Locks.	<input type="checkbox"/>	<input type="checkbox"/>
Die Effizienz von Backoff Locks wird durch die zugrundeliegenden Prozessor Plattform beeinflusst.	<input type="checkbox"/>	<input type="checkbox"/>

2 Selfish Peers (13 Punkte)

A) **Basics.** Beantworten Sie folgende Fragen mit 1-3 Sätzen:

- 1) (2 Punkte) Inwiefern können eigennützige (=selfish) Teilnehmer einem verteilten System schaden?
- 2) (2 Punkte) Was versteht man unter einem Nash Equilibrium?

B) (9 Punkte) **Selfish Caching.** Berechnen Sie den Price of Anarchy (PoA) und den optimistischen Price of Anarchy ($OPoA$) des folgenden Caching Netzwerks.



Für alle Peers u gelte $\alpha_u = 10$, und $w_u = 1$ wo nicht anders annotiert.

Zur Erinnerung: Die Caching Kosten α_u eines Peers u sind die Kosten, die für Peer u anfallen, falls er sich dazu entschliesst das File zu cachen. Der Demand w_u eines Peers u widerspiegelt die Häufigkeit, mit der u auf das File zugreifen möchte.

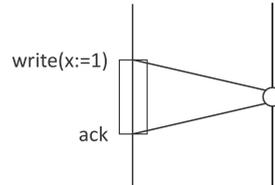
3 Paxos (10 Punkte)

- A) (4 Punkte) Wir nehmen an, dass wir bei Paxos 5 Proposer haben, welche sich initial alle einen Wert zwischen 1 und 5 überlegen, auf welchen sich am Ende alle Knoten einigen sollen. Stimmt die folgende Aussage: Wenn 3 der 5 Proposer sich für den Initialwert 5 entschieden haben, werden sich am Schluss alle Knoten auf den Wert 5 einigen, sofern das Protokoll terminiert. Begründen Sie Ihre Antwort!
- B) (6 Punkte) Wir wollen Paxos mit einem Quorumsystem an Stelle von Majorities umsetzen. Dazu verwenden wir ein Quorumsystem, bestehend aus vielen Knoten, in welchen die Schnittmenge zweier beliebiger Quoren genau zwei Knoten sind. Auf jedem der Knoten in diesem Quorumsystem läuft ein Paxos-Acceptor-Programm. Einer der Acceptor-Knoten arbeitet fehlerhaft: Er antwortet auf ein Prepare zum Teil korrekt mit $acc(x_{last}, n_{last})$ aber manchmal auch mit $acc(x_{old}, n_{old})$, wobei $acc(x_{old}, n_{old})$ eine Nachricht ist, die er schon früher gesendet hat. Ansonsten verhält sich dieser Knoten korrekt. Der Rest der Knoten führt das Paxos-Acceptor-Programm der Vorlesung aus. Zusätzliche, externe Knoten führen das Paxos-Proposer-Programm aus. Keiner der Knoten fällt aus (crasht). Funktioniert Paxos unter diesen Annahmen noch korrekt? Begründen sie ihre Antwort!

4 Consistency (14 Punkte)

In dieser Aufgabe führen wir ein Konsistenzmodell α ein, das in der Vorlesung nicht behandelt wurde.

Annahmen: Wir nehmen an, dass am Anfang alle Variablen auf 0 initialisiert sind. Alle Werte, die von Write-Operationen geschrieben werden, müssen eindeutig (unique) und grösser 0 sein. Ausserdem nehmen wir an, dass alle write-Operationen acknowledged sind. Das heisst, dass sie erst als beendet gelten, nachdem der Wert auf allen Replika geschrieben wurde. Weiter gilt, dass ein Client nur eine Operation gleichzeitig ausführen kann (siehe Bild).



Definition: Eine Execution ist α **consistent** falls für alle Clients X und für alle Clients Y gilt: X beobachtet nur mit Hilfe seiner Read-Operationen keine Reihenfolge von Write-Operationen von Y , die der tatsächlichen Ausführungsreihenfolge der Write-Operationen von Y widerspricht.

Notation: Verwenden sie in dieser Aufgabe die folgende Notation:

- **write(x:=val)** weist der Variable x den Wert val zu.
- **read(x)=val** ist eine Leseoperation auf der Variable x , die den Wert val zurück liefert.

Execution E1: Jede Zeile beschreibt die Operationen die der jeweilige Client ausführt und die Resultate dieser Operationen. Die Operationen sind pro Client von links nach rechts chronologisch geordnet (d.h. `read(x)=2` von Client1 kommt nach `write(x:=1)` und vor `write(x:=11)`, etc.).

- **Client1:** `write(x:=1)`, `read(x)=2`, `write(x:=11)`
- **Client2:** `write(x:=2)`, `write(x:=22)`, `read(x)=2`
- **Client3:** `write(x:=3)`, `read(x)=22`, `read(x)=1`

- A) (6 Punkte) Gegeben ist die Execution E1. Ist sie α **consistent**? Begründen sie ihre Antwort in 1-3 Sätzen!
- B) (8 Punkte) Gibt es eine Execution mit 2 Clients, die nicht sequentially consistent, aber α **consistent** ist? Falls nein, zeigen sie mit einem formellen Beweis, dass keine solche Execution existiert. Falls ja, geben sie eine solche an und zeigen sie, dass beide Vorgaben (nicht sequentially consistent aber α **consistent**) erfüllt sind.

5 Byzantines Quorum System (15 Punkte)

Die Rum produzierende Firma Quo stellt Sie ein, um ein Quorum System zu bauen. Sie stellt dabei die folgenden Anforderungen: Das Quorum System soll aus genau 3 Quoren bestehen und muss auch mit einem Byzantinischen Prozessor einwandfrei funktionieren. Sie haben dabei maximal 13 Prozessoren zur Verfügung und die Load muss kleiner als 1 sein.

- A) (6 Punkte) Definieren Sie mit Hilfe einer Skizze ein solches Quorum System mit möglichst wenig Prozessoren. Stellen Sie die Prozessoren dabei als Punkte dar und markieren Sie die Quoren.
- B) (5 Punkte) Warum kann man ein solches System nicht mit weniger Prozessoren realisieren?
- C) (4 Punkte) Was sind die Load, Work und die Resilience ihres Systems?

6 Konsensus (10 Punkte)

Wir erweitern das Konsensus Problem auf ein Beinahe-Konsensus Problem. Wir möchten erreichen, dass sich 3 Prozesse asynchron und warte-frei für maximal 2 verschiedene Werte entscheiden. Die folgende Validitäts-Kondition möchten wir erfüllen: **Wenn alle Prozesse mit dem gleichen Wert v starten, dann entscheiden sich alle korrekten Prozesse für den Wert v .** Wie beurteilen Sie das folgende Protokoll bezüglich der für Konsensus relevanten Kriterien? Löst es Beinahe-Konsensus? Begründen Sie Ihre Antworten!

Require: // default und inputs[] sind global definiert und bereits initialisiert. Sie können davon ausgehen, dass die Thread-Ids 0, 1 und 2 sind

```
1: default := 1;
2: inputs[] := ['?', '?', '?'];
3: procedure ALMOST_CONSENSUS()
4:   id := this.getThreadId();
5:   myInput := randomNumber();
6:   decision := myInput;
7:   // die Entscheidung
8:   inputs[id] = myInput;
9:   for (int i := 0; i ≤ 2; i++) do
10:     if ((inputs[i] ≠ '?') && (inputs[i] ≠ decision)) then
11:       decision := default;
12:     end if
13:   end for
14: end procedure
```

7 Spin Locks (8 Punkte)

Eine Barriere erlaubt einer gegebenen Anzahl von Prozessoren aufeinander zu warten um sich an einem gemeinsamen Punkt zu treffen. Dazu bietet die Barriere eine Methode `waitBarrier()` an. Die Methode blockiert den aufrufenden Prozessor so lange, bis jeder andere Prozessor die Methode ebenfalls aufgerufen hat. Anschliessend wird die Barriere aufgehoben und jeder Prozessor kann weiterarbeiten. Die Anzahl der Prozessoren ist konstant und zu Beginn bekannt.

Geben Sie eine Implementation der Methode `waitBarrier()` in Pseudo-Code an. Verwenden Sie dazu nur Spinning, eine shared Integer Variable und die CAS Operation. Weitere lokale Variablen innerhalb der Methode sind ok, aber nicht ausserhalb.