ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

FS 2016 Prof. R. Wattenhofer

Technische Informatik II

Klausur

Mittwoch, 24. August 2016, 15:00 - 16:30 Uhr

Die Prüfung dauert 90 Minuten und es gibt insgesamt 90 Punkte. Die Anzahl Punkte pro Teilaufgabe steht jeweils in Klammern bei der Aufgabe. Sie dürfen die Prüfung auf Englisch oder Deutsch beantworten. **Begründen Sie** alle Ihre Antworten und beschriften Sie Skizzen und Zeichnungen verständlich. Was wir nicht lesen können, können wir auch nicht bewerten!

Schreiben Sie zu Beginn Ihren Namen und Ihre Legi-Nummer in das folgende dafür vorgesehene Feld und beschriften Sie **jedes Zusatzblatt** ebenfalls mit Ihrem Namen und Ihrer Legi-Nummer.

Name	**Legi-Nr.**

Punkte

Aufgabe	**Erreichte Punktzahl**	**Maximale Punktzahl**
1		24
2		12
3		10
4		16
5		12
6		16
Summe		**90**

1 Multiple Choice (24 Punkte)

Geben Sie bei jeder Aussage an, ob sie wahr oder falsch ist. Jede korrekte Antwort gibt 1 Punkt, **jede fehlerhafte Antwort -1 Punkt**. Jede unbeantwortete Aussage gibt 0 Punkte. Wenn die Summe der Punkte für alle Aussagen negativ ist, dann wird die Aufgabe insgesamt mit 0 Punkten bewertet.

Lösungen

Aussage	wahr	falsch
Bei POST requests werden die request-Daten nicht in der URL mitgeschickt. Deswegen können POST requests nicht als Lesezeichen im Browser gespeichert werden. *Begründung: Nur URLs können gespeichert werden als Lesezeichen. Macht also nur bei GET requests Sinn, da die Daten dort direkt in der URL stehen.*	✓	
E-Mails werden mit SMTP versendet. SMTP basiert auf HTTP, wobei HTTPS für verschlüsselte E-Mails verwendet wird. *Begründung: SMTP weist zwar Ähnlichkeiten zu HTTP auf, basiert aber nicht darauf. SMTP baut direkt auf TCP auf.*		✓
POP3 erlaubt es, die E-Mails auf dem Server zu belassen wenn sie abgerufen werden. *Begründung: POP3 hat eine DELE und eine RETR Funktion. Wird also nur RETR aufgerufen, so bleibt die E-Mail auf dem Server. Mail-Clients wie Thunderbird geben Benutzern die Wahl.*	✓	
Pipes sind nicht für die Kommunikation zwischen zwei Rechnern geeignet. *Begründung: Pipes sind systeminterne Konstrukte. Zur Kommunikation zwischen verschiedenen Systemen muss man Konstruke wie bspw. Sockets verwenden.*	✓	
Die meisten Systemabstürze von Windows werden von Gerätetreibern verursacht. *Begründung: Wurde mehrfach vom Professor in der Vorlesung gesagt. Konkret passieren Abstürze durch Kernelmodecode, und Gerätetreiber laufen im Kernelmode. Ansonsten läuft dort nur Windows selbst, das weitgehend absturzfrei ist.*	✓	
In einer max-min-fairen Zuweisung (allocation) haben zwei Flüsse immer die gleiche Rate, wenn sie die gleiche Nachfrage (demand) haben und über mindestens eine gemeinsame Kante verlaufen. *Begründung: Gegenbeispiel: die Rate des einen Flusses könnte durch eine Kante mit sehr kleiner Kapazität, die der andere Fluss nicht benutzt, beschränkt sein.*		✓
In AIMD wird ungefähr gleich oft erhöht (increase) wie reduziert (decrease). *Begründung: Die additiven Schritte treten öfter auf, da sie kleiner sind.*		✓
In TCP hängen die Timeouts immer von der Round Trip Time ab. *Begründung: Stimmt so, vgl. Skript: "How long a sender should wait for the acknowledgement of a sent packet depends on the RTT."*	✓	
Wenn eine irreduzible, endliche Markov-Kette mit mindestens 2 Zuständen deterministisch ist, also wenn es zu jedem Zustand der Markov-Kette einen Folgezustand gibt, der mit Wahrscheinlichkeit 1 eintritt, dann ist die Markov-Kette nicht ergodisch. *Begründung: Der gerichtete Graph dieser Markov-Kette ist ein (gerichteter) Kreis und damit ist die Markov-Kette nicht aperiodisch, also auch nicht ergodisch.*	✓	

In einer Datenbank legen wir eine Tabelle für jede Entity an, hingegen nie für eine Relationship. *Begründung: Wir erstellen eine eigene Tabelle für jede n-to-n Relationship.*	✓
Gegeben zwei Tabellen T_1 und T_2, die eine Grösse (Anzahl der Tupel) von s_1 bzw. s_2 haben, so ist die Grösse von $join(T_1, T_2)$ höchstens $s_1 \times s_2$. *Begründung: Je eine Zeile von T_1 und T_2 ergeben höchstens eine Zeile im join, egal welche Art von join wir verwenden.*	✓
Zwei verschiedene Datenbanken haben immer unterschiedliche ER-Diagramme. *Begründung: Zwei verschiedene Datenbanken können laut Skript identische ER-Diagramme haben; beispielsweise könnte eine Datenbank nur eine Teilmenge der Datensätze (rows) der anderen Datenbank haben.*	✓
B+-trees können nur für Attribute verwendet werden, auf denen eine Ordnung definiert ist. *Begründung: B+-trees sind so gebaut, dass sie die verwalteten Elemente geordnet bereithalten; also müssen die Elemente anordenbar sein.*	✓
Eine Markov-Kette, die in der Übergangsmatrix (transition matrix) keine 0 als Eintrag enthält, ist aperiodisch. *Begründung: Jeder Zustand hat einen self-loop, der positive Wahrscheinlichkeit hat.*	✓
Für jede quadratische Matrix W, in der jede Zeile und jede Spalte mindestens einen von 0 verschiedenen Eintrag enthält, gibt es eine Markov-Kette mit Übergangsmatrix W. *Begründung: Die Summe der Einträge in jeder Zeile muss 1 sein.*	✓
In der stationären Verteilung eines einfachen Random Walks auf einem Pfad (mit mindestens 3 Knoten) haben alle Zustände die gleiche Wahrscheinlichkeit. *Begründung: Gegenbeispiel: bei einem Pfad mit 3 Knoten haben die beiden Enden eine Wahrscheinlichkeit von 25% und der Knoten in der Mitte eine von 50%.*	✓
Wenn es nur zwei Prozessoren gibt, dann sind ALock und das TTAS-Lock ähnlich effizient, weil es keine Invalidation Storms gibt. *Begründung: Invalidation Storms treten auf wenn mehrere Prozessoren gleichzeitig versuchen das TAS-Lock zu erlangen.*	✓
Sowohl das TTAS-Lock als auch das ALock sind FiFo-fair. *Begründung: Das TTAS Lock ist nicht FiFo-fair.*	✓
Einen Maximum Spanning Tree und einen Minimum Spanning Tree zu finden dauert gleich lange. *Begründung: Reduktion von Maximum auf Minimum: multipliziere alle Gewichte zuerst mit -1, dann finde einen Minimum Spanning Tree, dann wieder alle Gewichte mal -1 nehmen; ergibt einen Maximum Spanning Tree.*	✓
Wenn Sie ein Paket von einem Webserver erhalten, enthält es die MAC-Adresse der Webservers. *Begründung: Die Quell-MAC-Adresse wird auf den zwischenliegenden Hops auf dem Networklayer ersetzt.*	✓

In einem lokalen Netzwerk (LAN) können Sie die MAC-Adresse Ihres Geräts verstecken, indem Sie einen Switch dazwischenschalten. *Begründung: Switches geben Pakete unverändert weiter.*		✓
ARP eignet sich nicht für drahtlose Verbindungen. *Begründung: Doch.*		✓
Das Exposed Terminal Problem könnte mittels RTS/CTS gelöst werden, wenn die Wireless-Knoten zugleich senden und empfangen könnten. *Begründung: Nein, auch dann nicht.*		✓
Ein Nachteil von Lock-Free Synchronisation ist, dass es schwer ist, diese korrekt zu implementieren. *Begründung: Laut Folien zu Kapitel 8, Seite 9 ist das der Fall.*	✓	

2 Synchronisation (12 Punkte)

In dieser Aufgabe betrachten wir die Synchronisations-Varianten, die wir in der Vorlesung für ein Listen-basiertes Set verwendet haben. Wir möchten die Laufzeit der verschiedenen Varianten theoretisch analysieren.

a) [6] Ein Nutzer will eine *delete*-Operation durchführen. Wie lange benötigt dies, wenn der Nutzer einen einzigen Thread verwendet und

(i) coarse grained

(ii) fine grained

(iii) optimistic

Locking verwendet?

Die Grösse des Sets ist 2 000 und das 1 000ste Element soll gelöscht werden. Ein Lock anfragen und releasen dauert jeweils 20 ns; einem Zeiger folgen 5 ns. Alle anderen Operationen dauern 0 ns.

b) [6] Nehmen wir nun an, dass mehrere Threads verwendet werden. Beschreiben Sie je ein Zugriffs-Szenario, unter welchem

(i) coarse grained

(ii) fine grained

(iii) optimistic

Locking am langsamsten ist.

Lösungen

a) delete

(i) coarse grained: 20ns um lock zu beantragen, $1000 \cdot 5ns$ bis zum Element, 20ns für unlock = 5'040ns (dummy head kann ignoriert werden)

(ii) fine grained: Jedes Element wird gelockt/unlockt: (20+5+20)ns x 1000 = 45'000ns

(iii) optimistic: Erste Iteration 1000x5ns, zwei Locks beantragen: 2x20ns, zweite Iteration 1000x5ns, zwei Locks freigeben 2x20ns: = 10'080ns

b) langsam, wenn

(i) coarse grained: ganz viele Threads verwendet werden, aber die Zugriffe auf viele verschiedene Elemente verteilt sind

(ii) fine grained: nur wenige Threads verwendet werden (analog a))

(iii) optimistic: häufig benachbarte Elemente verändert werden, insbesondere am Ende der Liste. (Zum Beispiel, wenn Elemente eingefügt werden sollen, deren Vorgänger ungefähr gleichzeitig gelöscht werden soll.) Dann kann das Validieren oft fehlschlagen, was wiederholtes Traversieren der Liste nötig macht.

3 Public Key Krypto (10 Punkte)

Gegeben sei eine öffentliche "collision resistant" Hashfunktion h, und ein sicheres Public Key Cryptosystem mit öffentlichem Schlüssel k_p von Alice, und einem geheimen Schlüssel k_s, den nur Alice kennt. Für eine Nachricht m gilt: $k_p(m) = c$ und $k_s(c) = m$.

a) Bob schickt $k_p(m), h(m)$ an Alice. Betrachten Sie zwei Fälle, bei denen eine Eavesdropperin Eve mithört. Kann Eve jeweils die Nachricht entschlüsseln?

(i) `[2 Punkte]` Eve erhält $k_p(m)$

(ii) `[2 Punkte]` Eve erhält $k_p(m)$ und $h(m)$

b) Bob schickt nun Nachrichten m_i, $i = 1, 2, 3 \ldots$, via $k_p(m_i), h(k_p(m_i))$ an Alice. Eve fängt die Nachrichten ab und kann sie verändert an Alice weiterschicken. Was gilt für die Vorgehensweise von Bob?

(i) `[2 Punkte]` Sie ist Malleable

(ii) `[2 Punkte]` Sie erfüllt Forward Secrecy

(iii) `[2 Punkte]` Sie ist sicher gegen Replay-Attacken

Lösungen

a) Bob schickt $k_p(m), h(m)$ an Alice. Betrachten Sie drei Fälle, bei denen eine Eavesdropperin Eve mithört. Kann Eve jeweils die Nachricht entschlüsseln?

(i) `[2 Punkte]` und (ii) `[2 Punkte]` Beides nein: Aus $k_p(m)$ erhalten wir keine Informationen (sicheres PKC), und h ist eine one-way Hashfunktion, da h collision resistant ist – erlaubt also auch keinen Rückschluss auf m.

b) Bob schickt nun Nachrichten m_i, $i = 1, 2, 3 \ldots$, via $k_p(m_i), h(k_p(m_i))$ an Alice. Eve fängt die Nachrichten ab und kann sie verändert an Alice weiterschicken. Was gilt für die Vorgehensweise von Bob?

(i) `[2 Punkte]` Ja, da der öffentliche Schluessel k_p und h öffentlich ist, kann die Nachricht verändert werden - Eve verschlüsselt und hashed ihre eigene Nachricht, welche sie dann an Alice schickt.

(ii) `[2 Punkte]` Nein, der geheime Schlüssel k_s ist für alle Nachrichten identisch.

(iii) `[2 Punkte]` Nein, Bob könnte auch zweimal die gleiche Nachricht schicken - dann kann Alice nicht unterscheiden ob Eve "replayed" oder ob Bob noch einmal sendet.

4 Locking (16 Punkte)

Wir haben eine Klasse geschrieben, die genaue Zeitwerte (zwei 32bit Integer lang) anderen Programmteilen zur Verfügung stellt. Die `read()` Methode soll dabei den zuletzt geschriebenen Zeitwert zurückliefern. Ausserdem soll dies auch korrekt funktionieren wenn mehrere Threads parallel die `read()` und `write()` Methoden aufrufen. Korrekt heisst in diesem Fall, dass die `read()` Methode nur Werte zurück liefert die in einem einzigen Aufruf von `write()` geschrieben wurden.

```
1  class Timer64 {
2    private int[2] mTime;
3    private int mCounter;
4
5    public Timer64(){
6      mTime = {0,0};
7      mCounter = 2;
8    }
9
10   public read( int[2] time ){
11     while( true ){
12       int c1 = mCounter;
13       time[0] = mTime[0];
14       time[1] = mTime[1];
15       int c2 = mCounter;
16       if( c1 == c2 && c1 odd )
17         return;
18     }
19   }
20
21   public write( int[2] time ){
22     while( true ){
23       int c = mCounter;
24       if( c even ){
25         if( mCounter.compareAndSwap( c, c+1 ) == c ){
26           mTime[0] = time[0];
27           mTime[1] = time[1];
28           mCounter = mCounter + 1;
29           return;
30         }
31       }
32     }
33   }
34 }
```

a) `[4 Punkte]` Wieso bestehen sowohl die `read()` und `write()` Methoden nicht ausschliesslich aus den Zuweisungen der zwei Array Elemente?

b) `[4 Punkte]` Ist es möglich, dass mehrere Threads gleichzeitig die Werte von `mTime[0]` und `mTime[1]` verändern?

c) Ein kurzer Test zeigt, dass der Algorithmus nicht richtig funktioniert. Die `read()` Methode liefert ab und zu Werte `mTime[0]` und `mTime[1]` zurück, die nicht in einem einzigen Aufruf von `write()` geschrieben wurden.

(i) `[3 Punkte]` Wie kann es zu diesem Verhalten kommen?

(ii) [3 Punkte] Welche Zeile müssen Sie im Code anpassen, damit der Algorithmus korrekt funktioniert?

d) [2 Punkte] Angenommen Sie haben den Algorithmus korrigiert und dieser funktioniert nun korrekt. Ist es möglich, dass eine `write()` operation niemals terminiert (writer starvation)?

Lösungen

a) Die Aufgabenstellung fordert, dass read() nur Werte zurückliefert die in einem einzigen Aufruf von write() geschrieben wurde. Das wäre in diesem Fall nicht gegeben.

b) Nein, das ist nicht möglich. Nur falls mCounter von einem geraden auf einen ungeraden Werte geändert werden kann wird in mCounter geschrieben. Nur ein Prozess kann compareAndSwap(mCounter, c+1) ausführen welches den alten Wert c von mCounter zurückliefert.

c) Ein kurzer Test zeigt, dass der Algorithmus nicht richtig funktioniert. Die `read()` Methode `mTime[0]` und `mTime[1]` liefert ab und zu Werte zurück, die nicht in einem einzigen Aufruf von `write()` geschrieben wurden.

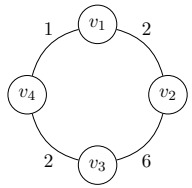
(i) Prozess A schreibt und wird zwischen Zeile 26 und 27 unterbrochen. Dann führt Prozess B ein read durch (dieses terminiert da mCounter odd ist).

(ii) Zeile 16 muss geändert werden zu: `if((c1 == c2) && c1 even)`

d) Der write Zugriff funktioniert unabhängig davon wie viele read Zugriffe gleichzeitig gemacht werden. Die einzelnen write Zugriffe werden zwar nicht fifo-fair abgearbeitet aber irgendwann kommt jeder Prozess zum Zuge.

5 Routing (12 Punkte)

Betrachten Sie ein Kreisnetzwerk mit n Knoten. Ein Kreisnetzwerk ist ein zusammenhängender Graph, in dem jeder Knoten Grad 2 hat, je einen Nachbarn im Uhrzeigersinn und im Gegenuhrzeigersinn. Den Kanten sind ganzzahlige nichtnegative Kosten jeweils kleiner gleich n^2 zugewiesen. Die IDs von v_1 bis v_n sind den Knoten im Uhrzeigersinn zugewiesen. Jeder Knoten hat eine Routingtabelle, mittels derer er ein Paket auf dem kostengünstigsten Weg in Richtung seines Ziels weiterschicken kann, wobei das Ziel jeder der n Knoten sein kann. Abbildung 1 zeigt ein Beispielnetzwerk.



Ziel	Nächster Knoten
v_1	Ziel erreicht
v_2	v_2
v_3	v_4
v_4	v_4

Abbildung 1: Ein Beispiel eines $n = 4$ Kreisnetzwerkes mit Routingtabelle für v_1.

Der Einfachheit halber können Sie annehmen, dass es $1ms$ dauert, um eine Nachricht über eine Kante zu senden, unabhängig von den Kosten der Kante. Lokale Berechnungen, die die Knoten vornehmen, kosten in diesem Modell keine Zeit.

a) [2] Für Kreisnetzwerke, vergleichen Sie das Distance-Vector- und das Link-State-Protokoll im Hinblick auf den pro Knoten verwendeten Speicherbedarf.

b) [6] Für Kreisnetzwerke, vergleichen Sie das Distance-Vector- und das Link-State-Protokoll im Hinblick auf den im schlimmsten Fall erforderlichen Aufwand (Zeit, Anzahl gesendeter Nachrichten) um alle Routingtabellen zu aktualisieren, nachdem sich ein einziges Kantengewicht ändert.

c) [4] Nehmen wir an, dass bekannt ist, dass sich in einem Kreisnetzwerk Kantengewichte nicht ändern können. Entwerfen Sie ein möglichst speichereffizientes Konzept für die Routingtabellen und erklären Sie, wie Sie Routingentscheidungen treffen.

Lösungen

a) Beide Protokolle brauchen gleich viel Speicher pro Knoten, nämlich $O(n)$ Einträge in die Routingtabelle. DV speichert pro Eintrag zwar zusätzlich noch ein Gewicht ab, aber LS muss die gesamte Topologie speichern, was für Kreise aber auch $O(n)$ Einträge braucht.

b) LS:

- Zeitbedarf T_m um alle Knoten über die Änderung zu informieren: $T_m = n/2 \implies O(n)$
- Zeitbedarf T_r um an jedem Knoten gleichzeitig(!) die neue Routing-Tabelle zu berechnen, z.B. mit dem Algorithmus aus dem Skript: $T_r = O(n \log(n))$
- Anzahl Nachrichten N_m die geschickt werden müssen: $N_m = n \implies O(n)$

DV:

- Count-to-Infinity: Im schlimmsten Fall ändert sich ein Kantengewicht von 1 zu n^2. In diesem Fall zählen die angrenzenden Knoten bis n^2 (Zeit und Nachrichtenkomplexität von $O(n^2)$), was schon deutlich schlechter ist als LS.
(*Erklärung*: Man muss gar nicht alle Knoten in die Analyse einbeziehen. Es reicht diese Lower-Bound zu zeigen. Eine genauere Analyse zeigt, dass in die Zeitkomplexität $O(n^2)$ und die Nachrichtenkomplexität $O(n^3)$ ist. Diese Einsicht war jedoch nicht nötig um die volle Punktzahl zu erreichen.)

c)

- Jeder Knoten halbiert für sich das Netzwerk anhand der Kantengewichte
- Die Grösse der Routingtabelle ist also konstant ($O(1)$). Man speichert nur die Grenzkante ab.
- Anhand der ID des Zielknotens und der Grenzkante entscheidet man dann ob man links oder rechts routet
- Dabei muss man den "wrap-around" bei Kante $(n, 1)$ beachten

6 Hashing (16 Punkte)

a) [4] Gegeben seien eine Hashtabelle der Grösse m und ein Universum U mit $|U| > m$. Zeigen Sie, dass jede universelle Familie von Hashfunktionen mindestens m Funktionen enthält.
Hinweis: Eine Familie von Hashfunktionen $\mathcal{H}$ ist genau dann universell, wenn für jedes Paar von zwei verschiedenen keys k_1, k_2 gilt, dass bei uniformer Auswahl einer Hashfunktion $h \in \mathcal{H}$ gilt, dass $\Pr[h(k_1) = h(k_2)] = \frac{1}{m}$.

Wir betrachten jetzt eine konkrete Hashfunktion. Sei $U = \{0, \ldots, 2^w - 1\}^l$ für $w, l \geq 4$, l und w gerade, $l \leq 2^w$; das Universum besteht aus Tupeln der Länge l von Bitstrings der Länge w. Sei $m = 2^w$ die Grösse einer Hashtabelle. Für $k = (k_1, \ldots, k_l) \in U$ sei der Wert der Hashfunktion $h_\oplus : U \to \{0, \ldots, m-1\}$ an der Stelle k definiert als:

$$h_\oplus(k_1, \ldots, k_l) = k_1 \oplus \ldots \oplus k_l$$

wobei $\oplus$ bitweises XOR ist. Beispiel: $0011 \oplus 0101 = 0110$.

b) [6] Geben Sie explizit eine Menge von mindestens 2^l keys an, die unter $h_\oplus$ den selben Hashwert haben.

c) [6] Wenn n keys unabhängig voneinander uniformverteilt aus U ausgewählt werden, was ist die erwartete Anzahl an Kollisionen, die unter $h_\oplus$ entstehen?

Lösungen

a) Wenn $\mathcal{H}$ universell ist, dann gilt für jedes Paar von verschiedenen Schlüsseln k_1 und k_2, dass k_1 und k_2 unter genau $\frac{|\mathcal{H}|}{m}$ vielen Hashfunktionen in $\mathcal{H}$ kollidieren. Da die Anzahl von Funktionen, unter denen k_1 und k_2 kollidieren eine ganze Zahl sein muss, muss $\frac{|\mathcal{H}|}{m} \geq 1$ gelten. Wenn aber $|\mathcal{H}| < m$, dann gilt $\frac{|\mathcal{H}|}{m} < 1$. Also muss $|\mathcal{H}| \geq m$.

b) Sei $k = (k_1, \ldots, k_l)$ die Binärdarstellung von $(0, 1, 2, \ldots, l-1)$. Wir können nun k beliebig permutieren, um einen neuen, anderen key zu erhalten, der unter $h_\oplus$ den selben Hash wie k hat. Wir erhalten so $l!$ viele keys mit dem selben Hash, und weil $l \geq 4$ gilt $l! \geq 2^l$.

c) Jeweils $\frac{|U|}{2^w} = \frac{|U|}{m}$ viele keys haben unter $h_\oplus$ den selben Hash, Beweis: Nimm für einen beliebigen key $k = (k_1, \ldots, k_l)$ die XOR-Summe der ersten $l-1$ Komponenten, dann entscheidet k_l die Gesamtsumme. Weil es 2^w Möglichkeiten für k_l gibt und jede davon eine andere Gesamtsumme erzeugt, haben alle diese 2^w keys einen anderen Hash.

Wenn wir zwei keys uniformverteilt aus U ziehen, haben wir also eine Chance von $\frac{1}{m}$ zwei keys zu ziehen, die unter $h_\oplus$ kollidieren. Mit Linearität der Erwartung ergibt sich damit $\mathbb{E}[\textit{Anzahl Kollisionen}] = \binom{n}{2}\frac{1}{m}$.