Department of Electrical Engineering and Computer Science

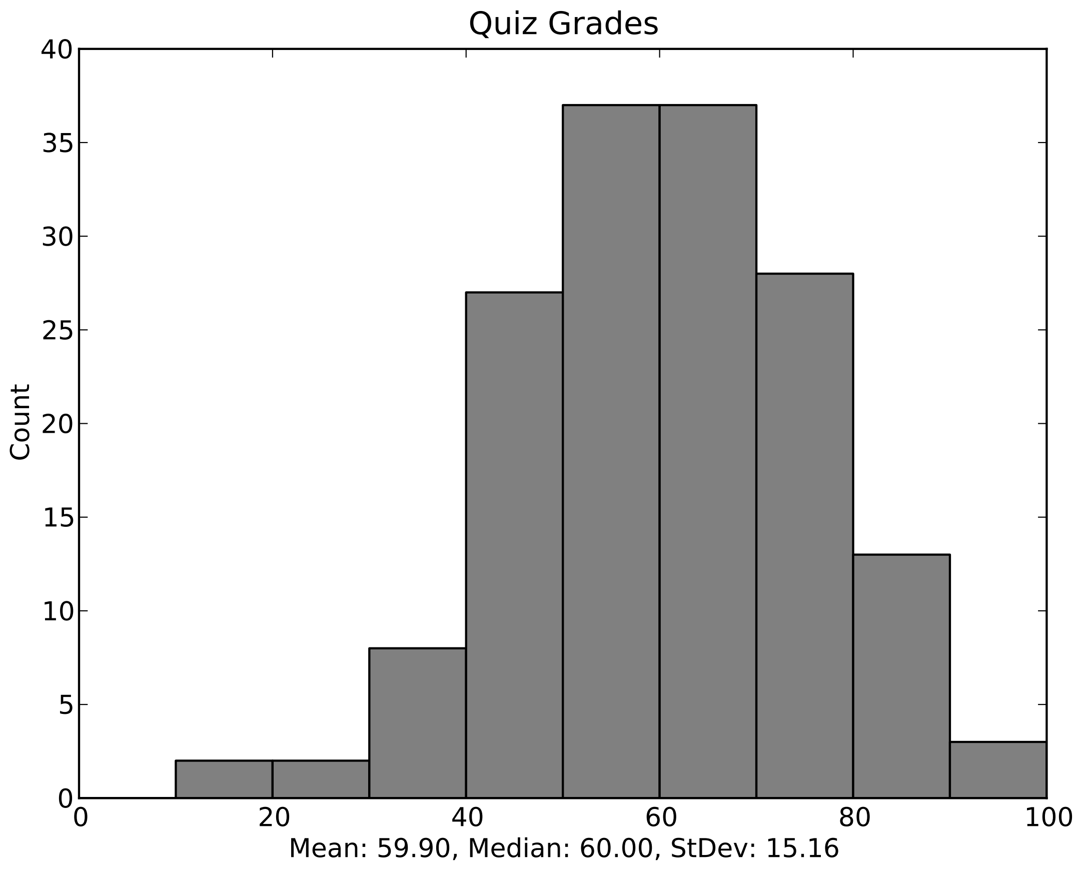
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.033 Computer Systems Engineering: Spring 2011

Quiz 2 Solutions

There are 11 questions and 12 pages in this quiz booklet. Answer each question according to the instructions given. You have **50 minutes** to answer the questions.

Grade distribution histogram:



I Reading Questions

1. [10 points]: Based on the paper "Do incentives build robustness in BitTorrent?", which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False A reference BitTorrent client will attempt to split its outgoing bandwidth equally between every peer it can connect to.

Answer: False. A BitTorrent client C only sends data to a peer P if P sends data to C at a rate fast enough to merit inclusion in the C's active set, or if C has optimistically unchoked P.

B. True / False BitTorrent permits content providers to shift bandwidth costs from their own ISP to those paid for by their content consumers.

Answer: True. A content provider publishes a torrent file and might initially seed the data, content consumers use BitTorrent's peer-to-peer protocol to further share data.

C. True / False The *torrent* file that a BitTorrent client downloads before joining a swarm contains the IP addresses of the *seed* nodes.

Answer: False. A torrent file contains the name and size of the file to be downloaded as well as SHA-1 fingerprints of the content blocks.

D. True / False The *BitTyrant* client uses unequal outgoing bandwidth allocations as one strategy to improve its download performance.

Answer: True. A BitTyrant client sets its upload contribution to a specific peer just high enough so that the peer reciprocates.

2. [10 points]: Based on the paper "A case for redundant arrays of inexpensive disks (RAID)" which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False A RAID array can improve reliability when individual disks fail independently from each other, but is less useful in the face of highly correlated failures.

Answer: True. If multiple disks fail at the same time, a RAID array will not be able to prevent data loss. However, if multiple disks fail over time, an administrator may be able to replace one failed disk (and allow RAID time to rebuild it) before the next one fails.

B. True / False The RAID paper predicted that disk seek times would improve at an annual rate similar to improvements seen in transistor density in microprocessors.

Answer: False. Seek times are limited by physical movement of the disk arm, and have not been improving much.

C. True / False If average disk capacities grow proportionally faster over time than sequential data transfer rates between disks and disk controllers, the MTTR (mean time to repair) of a RAID array will decrease.

Answer: False. Repairing a RAID array after a disk failure requires writing a disk's worth of data to the new disk. If the ratio of disk capacity to transfer speed grows, this will require more time, not less.

D. True / False If solid-state disks entirely replace spinning magnetic disks, none of the approaches described in the RAID paper will be needed anymore, since SSDs do not need to seek between different portions of the disk.

Answer: False. The RAID paper deals with failures of disks, and SSDs can still fail (albeit some of the reasons for failure are different).

II Border Gateway Protocol (BGP)

These questions are based on reading "An Introduction to Wide-Area Internet Routing".

3. [10 points]: Ben Bitdiddle was asked to debug a BGP problem. Help Ben to brush up his information about BGP. Which of the following statements are correct?

(Circle True or False for each choice.)

A. True / False To make money, an AS should announce the AS routes it learned from its customers to its peers.

Answer: True. These customers run services on their own servers and expect that any user on the Internet is able to reach their servers. Customers pay an AS to spread the reachability information of these servers to all routers on the Internet.

B. True / False To make money, an AS should not announce the AS routes it learned from its peers to its provider.

Answer: True. Two ASes establish a peering link to exchange traffic at no cost to each other. An AS, on the other hand, needs to pay its provider for any traffic the AS send to the provider or the provider sends to the AS. If an AS announces the routes it hears from its peers to its provider, the provider might route packets to the peers via the AS. The AS will in fact have to pay the provider for this incoming data.

C. True / False In BGP, an AS may learn many AS routes to an IP prefix.

Answer: True. One example is multi-homing. A customer may want to have two links to the Internet from two different providers. These different providers will announce routes to the same IP prefix up to their respective providers. At some point these announcements will reach a Tier 1 provider which will hear two routes to the same IP prefix.

D. True / False In BGP, the customer-provider relationships imply that if an AS has announced a route for a particular prefix to a neighbor, and later it learned a new route to that prefix, the AS will announce the new route to the neighbor.

Answer: False. Here is one scenario: the next hop in the old route uses a peer link, the new route's next hop uses a transit link. Financially, it is best for the AS to choose the old route over the new route; the AS will not advertise the new route.

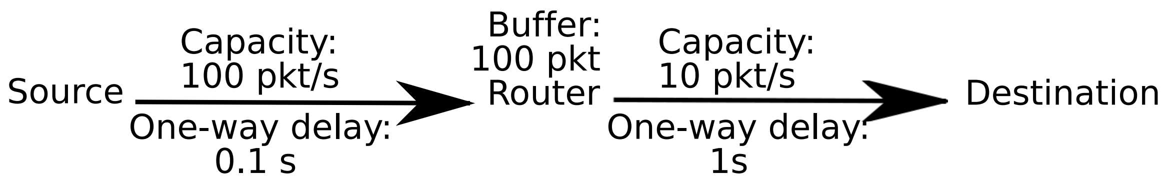
```
          Sprint
        /        \
   MIT AS         AS1
                    \
                     AS2
                       \
                        ...
                          \
                          AS10
                             \
                             AS EVIL
```

4. [10 points]: Consider the topology shown above. For any link, assume the higher node is a provider and the lower is a customer. MIT advertises prefix 18.* to Sprint. How can AS EVIL hijack most traffic destined to MIT from Sprint? (Describe briefly.)

Answer: It is no sufficient for AS EVIL to announce the 18.* prefix because Sprint will choose the shorter path to MIT and will not use AS EVIL's path. Instead, AS EVIL can take advantage of Longest Prefix Match to advertise prefixes that are more specific than 18.*. One scheme would announce 256 prefixes from 18.1.* to 18.256.*. Sprint's routers will choose the more specific prefix over MIT's 18.* prefix and send all traffic to AS EVIL.

Initials:

III Congestion control



Consider the above topology and assume the following:

- Link between Source and Router has a capacity of 100 packets/second and a propagation delay of 0.1 second in the forward direction.
- Link between Router and Destination has a capacity of 10 packets per-second and a propagation delay of 1 second in the forward direction. The transmission delays are 0.
- Acknowledgments from the destination to the source are delivered on a different path which has infinite capacity and zero delay, i.e., this means that acknowledgments will be delivered immediately from the destination to the source and will not suffer any drops.
- The Source and Destination can process packets very fast and the destination does not have any buffering limitations.
- The Router has a queue that can store a maximum of 100 packets.

5. [10 points]: Assume the Source uses a fixed window whose size is set to W. What is the maximum throughput in packet/sec that the source can deliver to its destination, if W=5, and then if W=20? (You can round your answer to the nearest integer.)

W = 5: _____________ pkt/s

Answer: The main point of this question is to realize that when the window is too small, it limits the throughput, but when the window is too large the throughput is limited by the capacity of the bottleneck link (given the system has enough buffering). Thus, when W = 5 packets, the delay inside the network is due to the propagation delay which is 0.1+1 = 1.1 second. The transmission rate = W/RTT = 5/1.1 packets/s. Since this is smaller than the bottleneck capacity of 10 packets/s, the throughput is 5/1.1 packets/s. A more accurate answer includes in the RTT computation the time to process a packet on each of the links, which is 0.01 seconds on the first link and 0.1 second on the second link. Hence, the RTT is 0.1+0.01+1+0.1 = 1.21 seconds. The throughput will be 5/1.21 packets/second. We accepted both answers since some routers can start forwarding the packet on the next link even before they receiver the last bit from the previous link.

W = 20: _____________ pkt/s

Answer: When W=20 packets, the throughput is 10 packets/second, which can be found immediately by noticing that in this scenario you are limited by the bottleneck capacity. The transmission rate is still given by W/RTT, however the throughput can never exceed the bottleneck capacity of 10 packets/second. In this case, the RTT will increase due to increased queue size at the router and the accompanying increase in delay, so that W/RTT is 10 packet/s.

Initials:

6. [10 points]: Assume that the source uses TCP. Focus on how TCP adapts its congestion window using AIMD as described in class. Ignore other details of TCP (e.g., slow start, fast retransmission and fast recovery). What is the maximum window size that the source's TCP will experience during AIMD?

Maximum window size: _____________ packets

Answer: The correct answer is 111 packets. Specifically, the TCP window will grow until the first drop, which will occur once the router's queue is full, and the bottleneck link has 10 packets along the link. This is total of 110 packets inside the network. Say the network has that number of packets in the router's queue and on the bottleneck link. The next RTT, when the TCP tries to increase its window by 1 more packet, i.e., W=111, the new packet will reach the router before the router can empty a spot in its queue and the drop will occur.

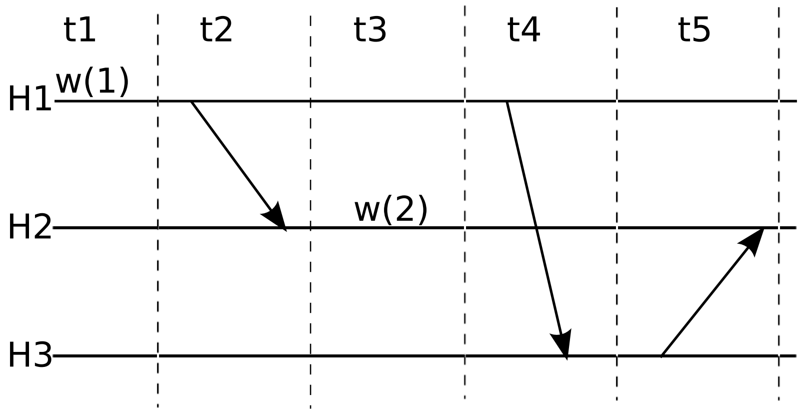
We accepted also 110 because this is the window at which the drop occurs. We also accepted 121 as a good answer because a TCP may get to 121 packets before it sees a drop if the window is initialized to 120. Specifically, the first link which has a capacity of 100 packets/s and a delay of 0.1 second, can keep 10 additional packets inside the network for a total of 120 packets in the network. This is however conditioned on starting with a large window and sending the packets over the first link in a burst.

We gave partial credit to students who realized that the router's queue has to be full before there is a drop and hence the TCP window will grow to 100 packets. These students missed the fact that there are some packets on the bottleneck link but got the buffer part. We also gave partial credits to students who have a correct argument but thought that TCP increases its window by 1 packet per ack (rather than 1 packet per RTT).

IV File reconciliation and time vectors

Ben designs a new tool for reconciling copies of a file on different computers. The scenario is a single user who has a number of computers, each of which may have a copy of a file f, and the user updates the copies of f independently from each other. For example, when the user is on an airplane, he can update only the copy on his laptop, and when the user gets to his office but forgets his laptop at home, he can update only the copy on his office machine. Ben's goal is that the different copies of f should behave as if there is a single copy of f.

To develop a design, he considers two update and reconciliation patterns. The first pattern for file f is as follows:



It shows 3 hosts: H1, H2, and H3. H1 writes (w) the value 1 to the file f, and then reconciles to H2 (i.e., changes on H1 are propagated to H2 and reconciled with changes on H2). Later H2 writes 2 to the file f and H1 reconciles to H3. Then, H3 reconciles to H2. To make it easy to talk about the different events, we have labeled them with real times t1 through t5, but you should assume that the clocks of the different computers are not synchronized and the computers have no agreement on this global time.

7. [5 points]: For the ideal outcome (i.e., f should behave as if there is a single copy), what value should f contain at H1, H2, and H3 after the above update and reconciliation pattern completes (i.e., at the end of t5)?

(Circle True or False for each choice.)

A. True / False H1: 1, H2: 2, H3: 2

Answer: False.

B. True / False H1: 1, H2: 2, H3: 1

Answer: True.

Reconciliations are unidirectional, as stated in the question and as indicated by the arrows.

At t2, H2 sees that H1 has a more recent value of f and set its own value of f to equal 1 because of the reconciliation. At t3, H2 changes its own local value of f to 2. At t4, H3 sees that H1 has a more

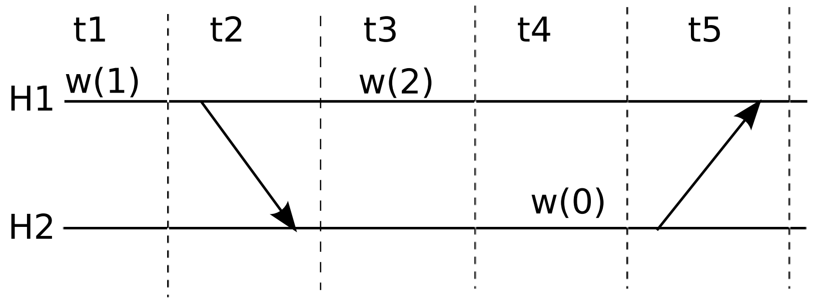
Initials:

recent value of f and set its own value of f to equal 1 because of the reconciliation. At t5, H2 ignores the value of f from H3 despite the reconciliation, because it has a more recent value of f. H2 has seen H1's write, and later made a change to f, so H2's value should not by overwritten by H3.

We end up with H1: 1, H2: 2, H3: 1.

Initials:

After reading the Unison paper, Ben realizes that it isn't always possible to come to agreement between nodes what the value of f should be. Consider the following second pattern:



H1 writes 1 into f and reconciles to H2. Then, H1 and H2 modify f in different ways, and then H2 reconciles to H1 at t5. At t5, H1 and H2 have a *conflict* because there is no way to order the updates at H1 and H2 at t3 and t4 that is consistent with what could have happened with a single copy of f. A user needs to get involved to determine what the right value is.

Ben's goal is to design a tool that reconciles updates to a file f correctly when it is possible, but raises a conflict when it is not possible. His design is based on time vectors (introduced in the lecture on time, but you don't need to recall the lecture to be able to answer this question). A time vector is a vector of integers, where entry i counts the updates to f on computer i. For example, the vector (0, 0) on host 2 signals that it hasn't seen updates for f from H1, and that H2 itself hasn't done any updates either.

8. [10 points]: For the pattern above (the second pattern), what is the value of the time vector for f at each node at the end of t1 through t4? (complete the table)

time	H1	H2
t1	(1, 0)	(0, 0)
t2	—	
t3		—
t4	—	

Answer:

time	H1	H2
t1	(1, 0)	(0, 0)
t2	—	*(1, 0)*
t3	*(2, 0)*	—
t4	—	*(1, 1)*

When H2 and H1 reconcile at t2, H2 will compare its vector (0, 0) with H1's vector (1, 0). Because H1's vector is more recent – (1, 0) $\geq$ (0, 0) – H2 will fetch the recent value of f from H1, and set its own vector to be (1, 0).

At t3, H1 makes a change to its copy of f, so it updates its vector to mark a write. H1's vector becomes (2, 0).

At t4, H2 makes a change to its copy of f, and it also updates its vector to mark a write. H2's vector becomes (1, 1).

Initials:

9. [5 points]: How can Ben's design detect that pattern 2 is a conflict? Be specific, explain using the time vectors in the examples above.

Answer: When H1 and H2 reconcile at t5, they will notice a conflict because of concurrent updates to f. Specifically, pattern 2 is a conflict because vectors v = (2,0) and w = (1,1) cannot be ordered with respect to one another. We can order v and w if $v[i] \geq w[i]$ for all i (because then $v \geq w$), or if $w[i] \geq v[i]$ for all i (because then $w \geq v$). Here, $v[0] > w[0]$ but $v[1] < w[1]$.

V Write-Ahead Logging

Ben hasn't solved enough problems in the last hour. Inspired by the lectures on atomicity and hands-on #5, he decides to build a database system for his bank. The database is a simple one: its cell storage is a table of account numbers and balances stored on disk, and it has only one operation: `credit(account, delta)` adjusts the balance of the specified account by the (possibly negative) amount `delta`.

Ben wants the database to support atomic transactions, so he adds the usual `begin` and `commit` operations, and stores a log on a separate disk. The operations are implemented as follows:

- `begin` appends a $\langle$BEGIN, $transactionid\rangle$ record to the log.
- `credit(account, delta)` updates the balance of the appropriate account in memory, but does *not* write the new balance to cell storage. It also appends to the log the record:
  $\langle$UPDATE, $transactionid$, $account$, $delta$, $oldbalance$, $newbalance\rangle$
- `commit` first appends to the log $\langle$OUTCOME COMMITTED, $transactionid\rangle$. It then writes out to cell storage any account balances modified by the transaction. When this is done, it appends to the log $\langle$END, $transactionid\rangle$

Ben knows that this protocol can support all-or-nothing atomicity across system crashes. He sketches a recovery procedure. After a crash, the system will scan the log and *redo* the effects of any transaction that logged an OUTCOME COMMITTED record but not an END record. But rather than implementing the recovery procedure, he puts the system into production immediately. After all, Worse Is Better, and he can always write that recovery procedure later...

Inevitably, Ben's system crashes before he gets around to writing that recovery procedure. The bank's customers are not amused. Panicked, Ben turns to you to help him recover the state of his database.

Initials:

The cell storage of the database at the time of the crash, and the last few entries of the log appear below. All earlier entries in the log correspond to transactions known to have completed in their entirety.

Cell Storage

AccountID	Balance
account1	$890
account2	$648
account3	$32
account4	$1500

Log

```
⟨BEGIN, T_1⟩
⟨UPDATE, T_1, account2, 200, 448, 648⟩
⟨OUTCOME COMMITTED, T_1⟩
⟨END, T_1⟩
⟨BEGIN, T_2⟩
⟨BEGIN, T_3⟩
⟨UPDATE, T_3, account1, 500, 390, 890⟩
⟨UPDATE, T_3, account4, −500, 1500, 1000⟩
⟨UPDATE, T_2, account3, 100, 32, 132⟩
⟨OUTCOME COMMITTED, T_3⟩
```

10. [10 points]: Fill in the values that will appear in the cell storage once recovery is completed using Ben's intended recovery scheme:

AccountID	Balance
account1	$890
account2	$648
account3	$32
account4	$1000

Answer: $T1$ has logged both a OUTCOME record and an END record, so its changes have already been installed into cell storage and the recovery procedure can ignore it.

$T3$ has logged an OUTCOME record but not an END record, so it is a *winner* and needs to be redone. At the time of the crash, its update to $account1$ had already been installed to cell storage, but its update to $account4$ had not. So the recovery procedure will change account4's balance to $1000.

$T2$ has not logged an OUTCOME record, so it had not reached the commit point at the time of the crash. It is deemed a *loser*, and the recovery procedure should not redo its changes. $account3$'s balance remains unchanged.

Initials:

11. [10 points]: Ben is worried that the write-ahead log will take up too much space. Given Ben's recovery procedure, which of the following options would reduce the size of the log, while still being able to persistently store data, and recover from crashes with all-or-nothing atomicity?

(Circle True or False for each choice.)

A. True / False eliminating the BEGIN record.

Answer: True. The BEGIN record is redundant. The existence of a transaction can be inferred from the first UPDATE record for that transaction.

B. True / False removing both the $oldbalance$ and $delta$ fields from the UPDATE record

Answer: True. As long as the $newbalance$ field is present, we can redo changes made by committed transactions. Ben's recovery procedure does not require undoing changes.

C. True / False removing both the $oldbalance$ and $newbalance$ fields from the UPDATE record

Answer: False. The $delta$ field alone is not enough to redo the effects of committed transactions, because adjusting the account balance by $delta$ is not an idempotent operation. A transaction might have committed but only written part of its changes to cell storage at the time of the crash (like $T3$ from the previous question). Without either the old or new balance, the recovery procedure has no way to know whether it needs to redo the change or not.

D. True / False periodically writing a CHECKPOINT record to the log which contains the ID of each uncommitted transaction, then discarding all log entries before the CHECKPOINT record.

Answer: False. Consider a transaction that begins before the checkpoint, and commits after the checkpoint. If the system crashes between when that transaction logs its OUTCOME and END records, the recovery procedure will need to redo its updates. But any UPDATE records made before the checkpoint will have been discarded.

End of Quiz II

Initials: