Midterm — Oct. 12, 2007

EE122: Introduction to Communication Networks

Fall 2007

SOLUTIONS

Prof. Paxson
Department of Electrical Engineering and Computer Sciences
College of Engineering
University of California, Berkeley

Problem	1	2	3	4	5	6	7	8	Total
Mean Score	17.0	10.4	11.4	10.2	17.3	9.7	13.1	13.4	102.5
Max Points	20	15	20	20	20	15	20	20	150

High score: 140
90th percentile: 126
75th percentile: 116
Median score: 108.5
25th percentile: 89
Low score: 40

1. **Alphabet soup.**

   For each of the concepts given below, list which of the following acronyms apply:

   > 802.3, ARQ, CDN, CIDR, CNAME, CRC, CSMA, CSMA/CD, DF, FDMA, FTP, HTTP, ICANN, IMAP, IPv4, IPv6, LAN, MIME, MTU, MX, Mbps, NIC, NRZ, NRZI, NS, PTR, RFC, RIR, RTT, SMTP, SONET, TCP, TDMA, TLD, TTL, UDP, URI

   Not all acronyms are used. Unless otherwise stated, there is one acronym per concept. (1 point per item)

   Mentioning any additional acronyms along with the correct one generally resulted in no credit for the given item, except when the additional acronym also closely related to the concept.

(a) Used to provide a web site's content from a large number of distributed servers: **CDN**. One common error was to list **HTTP** instead. However, HTTP does not by itself entail "a large number of distributed servers."

(b) A way of interpreting IP addresses as having an initial (variable-length) network prefix, plus the remaining bits identifying a host within that network: **CIDR**

(c) An Internet standards document: **RFC**

(d) The IEEE standardization of Ethernet: **802.3**

(e) 2 types of layer-4 transport protocols: **TCP, UDP**. You needed to specify both to get credit.

(f) A DNS resource record used to identify email servers associated with a domain: **MX**

(g) An application-layer protocol that uses an additional connection between the same client and server for each data object transferred: **FTP**. Some put down **HTTP** instead. However, only version 1.0 of HTTP uses a separate connection for each object. HTTP 1.1 added persistent connections.

(h) A term referring to a host's network adaptor: **NIC**. A common error was to list **LAN** instead. That, however, refers to the entire local-area network, not particularly to the host's adaptor.

(i) 4 types of DNS resource records: **NS, MX, PTR, CNAME**. You needed to specify 3 of the 4 to get credit.

(j) The largest sized packet that can be sent across a link (or a network path) without requiring fragmentation: **MTU**. Some instead listed **DF**. While the IPv4 DF header flag is used to prohibit fragmentation, by itself it does not refer to the size of packets that can be sent without fragmentation.

(k) The style of MAC (Media Access Control) protocol used by Ethernet: **CSMA/CD**. No credit was given for instead (or also) mentioning **CSMA**, since a key facet of Ethernet's MAC protocol is its *collision detection*.

(l) A scheme for computing a checksum value over a block of data in order to detect bit errors: **CRC**

(m) A standardized scheme for encoding different types of email (and Web) content: **MIME**. A common error was to specify **HTTP** instead. However, HTTP is not involved in email, and also by itself isn't a scheme for *encoding* content, though it includes mechanisms for specifying these.

(n) An entity responsible for allocating Internet address blocks for a large region: **RIR** or **ICANN**. I was looking for **RIR** in particular, but **ICANN**—which ultimately controls *all* address blocks, and allocates them to a given **RIR** for each large region, was also a reasonable answer.

(o) The time it takes to send a packet to a destination and hear a response back from it: **RTT**

(p) The main protocol used to transmit email: **SMTP**. One error here was to list **IMAP** instead. However, IMAP is only one of several protocols used to read email once it has been transmitted; it's not used to transmit the email. Another error was to give **MIME**, which concerns how mail is *encoded*, but not how it is *transmitted*.

(q) The term for primary DNS zones such as **.com** or **.uk** : **TLD**. A common error was to instead give **NS**. That term refers to a particular type of DNS Resource Record, rather than the primary DNS zones.

(r) A counter in the IP header that is decreased at each hop; if it reaches 0, the packet is discarded (also refers to how long to keep a DNS response in a local cache): **TTL**

(s) A class of mechanisms used for transport protocols that achieve reliability by retransmitting missing data. Particularly used in reference to simple reliable schemes such as Stop-and-Wait: **ARQ**. Some put down **TCP** instead. TCP is one particular transport protocol. The question is instead asking about the name of a set of mechanisms used by a number of transport protocols, one of which is TCP (though only partially, as we will see.

(t) A way of sharing a link's capacity among a group of senders in which each sender is assigned its own frequency to use when transmitting, which it can use to transmit whenever it pleases: **FDMA**

2. **MAC Protocols.**

For each of the following types of media access control protocols, place an X if the given attribute applies to it. (15 points)

Attribute	Ethernet	Slotted Aloha	Token Passing	TDMA	FDMA
Uses Collision Detection	✗	✗			
Uses Carrier Sense	✗				
Uses Exponential Backoff	✗				
A single node can potentially use close to all of the capacity	✗	✗	✗		
Nodes have to wait for it to be their turn to send			✗	✗	
Operates efficiently when many nodes all have data to send			✗	✗	✗
Vulnerable to failure of a single node			✗		
Uses randomness to avoid synchronization	✗	✗			

Scoring was +1 point for each box correctly checked, -1 point for each box incorrectly checked, and a minimum of 0 if the latter outweighed the former.

Common problems:

- While Slotted Aloha uses a form of backoff when it detects a collision, it is *not* exponential (doesn't back off further and further on each subsequent collision).
- Token passing allows a single node to potentially use close to all of the capacity, since passing around the token occurs considerably more quickly than the time it takes to transmit a large frame.
- Ethernet does not operate efficiently when *many* nodes all have data to send, as these senders will very often generate collisions.
- Ethernet uses randomness during its exponential backoff procedure in order to make it unlikely that nodes pick the same backoff interval
- With both token passing and TDMA, a node must wait until its designated turn for transmitting. With FDMA, nodes do *not* need to wait, since they are free to use their assigned frequency whenever they wish.
- Token passing operates efficiently when many nodes have data to send since each transfer of the token to a new sender generally allows the sender to then transmit and make effective use of the network.

- The phrase “avoid synchronization” refers to senders trying to send at the same time. It does not refer to clock synchronization.

3. **Multiple Data Transfers.**

Using a Web browser, you visit the web site for `www.hamburger.com`. The base HTML page for the main page `www.hamburger.com` is 30,000 bits. Once the base HTML page is fetched, it contains URL references for the following embedded images:

`http://www.hamburger.com/burger_banner.jpg`	15,000 bits
`http://www.hamburger.com/lettuce.jpg`	5,000 bits
`http://www.hamburger.com/mmm_bacon.jpg`	10,000 bits
`http://www.hamburger.com/veggie.jpg`	10,000 bits
`http://www.hamburger.com/disclaimer.txt`	5,000 bits
`http://www.hamburger.com/royale_with_cheese.jpg`	35,000 bits

Your Web browser uses the HTTP protocol to download the base page and the embedded objects. Make the following assumptions:

- At most 10,000 bits of data fits into a single packet. You can ignore the overhead of any headers or framing.
- You must first download the entire base page before you can start fetching the embedded images.
- HTTP requests are 1,000 bits in size.
- Any new connection to a machine requires a connection-establishment handshake.

  For this problem, you do not need to worry about closing connections, and you can ignore the delay introduced in acknowledging the final data packet sent by the server to your browser.
- All senders use windows of 20,000 bits.
- No packets are lost.

Note: many students found this problem difficult.

(a) For the initial transfer of the home page, how many RTTs are required, and what occurs during each of them? (5 points)

Answer: 3 RTTs.

i. Establish the connection.

ii. Send over the request and receive 20,000 bits of the home page data in reply. 20,000 bits = two maximum-sized packets, as permitted by the sender’s window.

iii. **Send an acknowledgment for this data, which then slides the window, and receive the remaining 10,000 bits (one packet) in reply.**

Common problems:

- An RTT reflects a round-trip of traffic, i.e., packets from one end to the other and responses to those packets (either acknowledgments and/or data) coming back. A number of answers instead counted each such exchange as two RTTs.
- Given that senders have *windows* allowing transfer of more than one packet, the transfers do *not* proceed by Stop-and-Wait (one data packet at a time).
- A number of answers had an RTT for just the initial request from the client (plus and acknowledgment of it by the server), rather than the server replying to this request with the first 20,000 bits of data. As soon as the server receives the request, it can send out data in response.
- Some answers neglected to include an RTT for the initial connection establishment.
- The problem states (and was further clarified during the exam) that the question pertains to downloading of the base page only, not also the embedded images.

(b) How quickly (in terms of RTTs) can your browser download the base page for `www.hamburger.com` and all embedded objects if the browser uses:

Note: if your answers in this part were incorrect but were consistent with your answer for part (a), then you received credit if you reused the same logic (for example, you did your computations assuming Stop-and-Go rather than sliding window). However, this required that your reasoning remained consistent through the problem.

i. One connection per item, with up to 4 concurrent connections. (5 points)

Answer: either 7 RTTs or 8 RTTs, depending on whether your browser starts fetching `royale_with_cheese.jpg` immediately or only after the first 4 retrievals.

A. It takes 3 RTTs to fetch the base HTML page, per the previous problem.

B. It takes 2 RTTs to fetch each of the embedded images other than the last: one RTT to establish a new connection and one to retrieve it, since it fits within the sender's window.

C. It takes 3 RTTs to fetch `royale_with_cheese.jpg` since it's too big for a single window.

D. The problem asks for how quickly the browser can fetch everything. To do so, once it received the base page it would start off fetching `royale_with_cheese.jpg` and three of the others. Once those others finished, it would fetch the two remaining ones. While doing so, `royale_with_cheese.jpg` would complete, and the two remaining ones. Thus, the total time for the embedded images is 4 RTTs.

E. It's reasonable to assume the browser wouldn't necessarily figure out it should start working on `royale_with_cheese.jpg` initially (the browser likely doesn't know its size, and therefore that it's worth getting an early start). In this case, it would take 5 RTTs to fetch the embedded images: the first 4 concurrently (2 RTTs) plus then the last two (3 more RTTs).

Common problems:

- The most important facet of this problem is that the answer reflects the time taken by up to 4 connections executing *at the same time*.
- An incorrect assumption that because the images total 80,000 bits, then they can fit into either 8 packets or (a bit better) 4 windows is appropriate for using a *pipelined* connection, but not for *concurrent* connections. The latter require their own connection establishment, and cannot share space across multiple items (such as the 15,000 bit item and the 5,000 bit item together fitting into 2 packets).
- Some answers overlooked that there could only be *four* simultaneous connections, rather than all 6.
- Some answers overlooked that the question included the time to download the base HTML page too.
- The term "concurrent connections" refers to multiple transport-level (TCP) connections, not multiple requests within a single TCP connection.
- It's important to recognize that one of the items (namely `royale_with_cheese.jpg`) takes two RTTs to transfer its data.

ii. A single persistent, non-pipelined connection. (5 points)

Answer: 10 RTTs.

- **Again it takes 3 RTTs to fetch the base HTML page.**
- **With a persistent, non-pipelined connection, the browser can only fetch one item at a time, waiting to receive all of it before requesting another item. However, these items do *not* require any further connection establishments.**

- **All of the embedded images except `royale_with_cheese.jpg` can be transferred in a single RTT, since they fit within the server's sending window of 20,000 bits.**
- **`royale_with_cheese.jpg` takes 2 RTTs.**
- **Therefore the embedded images require 1+1+1+1+1+2 = 7 RTTs.**

Common problems:

- Non-pipelined does not mean one packet per RTT; it means one *request* at a time. So while it is *analogous* to transport-level Stop-and-Go, it still uses sliding window (up to 2 packets per RTT, in this case).
- Some partial credit was allowed if this problem was solved in a manner consistent with a *pipelined* HTTP connection, and the next problem in a manner consistent with a *non-pipelined* connection.
- When using a persistent connection, there is only a single RTT of connection establishment overhead (at the beginning).

iii. A single pipelined connection. (5 points)

Answer: 7 RTTs.

- **Again, 3 RTTs for the base HTML page.**
- **With pipelining, the browser can send over requests for all of the embedded images at once.**
- **The server returns these requests at a rate of 20,000 bits/RTT, since that's its window size. In particular, with pipelining it can send data for multiple replies in a single packet.**
- **Accordingly, the 80,000 bits' worth of embedded images require 8 packets = 4 windows, so 4 more RTTs to transfer.**

Common problems:

- The base HTML page cannot be pipelined with any of the embedded items since it must first be received in its entirety before the browser knows what items to fetch.
- Similarly to the above problem, HTTP pipelining is *analogous* to Sliding Window in terms that multiple messages are underway at one time. Its use does not mean that the transport protocol (TCP) starts using Sliding Window. In this whole problem (all four parts), the transport protocol is *always* using Sliding Window (up to 2 packets per RTT).

4. **Performance of reliable data transfer.**

Consider two nodes, A and B. Suppose the network path from A to B has a bandwidth of 5 KB/s (5,000 bytes per second) and a propagation time of 120 msec. The path in the reverse direction, from B to A, has a bandwidth of 10 KB/s and a propagation time of 80 msec.

Let data packets have a size (including all headers) of 500 bytes and acknowledgment packets a size of 100 bytes.

Note: many students found this problem difficult.

(a) Give a numeric expression for the throughput A can achieve in transmitting to B using Stop-and-Wait. You can treat a 500-byte data packet as transferring 500 bytes of useful data (that is, ignore that it's a bit less due to the headers). (5 points)

Answer:

The transmission time of a data packet from A to B is (500 B)/(5 KB/s) = 100 msec.

The transmission time of an acknowledgment from B to A is (100 B)/(10 KB/s) = 10 msec.

The total propagation time is 80+120 = 200 msec.

Thus an RTT for sending a data packet and receiving an acknowledgment for it is 200+100+10 = 310 msec.

With Stop-and-Wait, A sends one data packet per RTT, so the throughput it can achieve is (500 B)/(310 msec) = 500/0.310 B/s ($\approx$ 1613 B/s).

Common problems:

- Throughput refers to useful data sent over time. Acknowledgments are not "useful data," so their size does not figure into how much data is sent over time. However, their propagation time and transmission time does, since the arrival of acknowledgments governs when the sender can send new data.
- Since throughput is computed over *elapsed time*, you need to figure in not only the propagation time of the path but also the transmission time of both the data packets and the acknowledgments.
- Another common error regarding the time interval was to consider only the time elapsed in the forward direction. However, as mentioned above, since the receipt of acknowledgments is required to continue sending data, the time they take to come back also must be figured in.
- By definition, Stop-and-Wait entails sending just one data packet per round-trip.

(b) Give a numeric expression for the size of the window, in terms of number of data packets, that A must use in order to transfer its data as fast as possible, if A instead uses Sliding Window. (5 points)

Answer: to go as fast as possible, A must use a window that is at least as large as the bandwidth-delay product, which is 5 KB/s $\cdot$ 310 msec = 1,550 bytes.

The question asks for how many data packets, which is given by:

$$\left\lceil \frac{1550\ B}{500\ B/pkt} \right\rceil = 4 \textbf{ pkts}$$

where $\lceil \cdots \rceil$ denotes the "ceiling" operator (round up to nearest integer).

Common problems:

- The problem called for computing the size of the window in terms of data packets, not bytes.
- When computing the size in packets, you need to round up to ensure that the window is large enough to completely fill the available capacity.
- It was not enough to simply mention the bandwidth-delay product; the problem asked for a *numeric expression*.
- The "delay" term in the bandwidth-delay product corresponds to how long it takes to receive an acknowledgment for a full-sized data packet. Thus as well as the round-trip propagation time, it must take into account the transmission times for both a full-sized data packet and its acknowledgment.
- The "bandwidth" term in the bandwidth-delay product refers to the forward path's capacity, *not* the throughput achievable by Stop-and-Wait.
- The bandwidth in the reverse direction does not come into play *unless* it is so small that the acknowledgments get spread out and they can't slide the window quickly enough (see part (d)).
- Note, if you miscomputed the RTT in part (a) but then used that value again here in an otherwise correct fashion, you received full credit.

(c) What is the maximum rate A can achieve? (5 points)

Answer: when using a window greater or equal to the bandwidth-delay product, a sender can achieve a rate—i.e., how fast it can transmit over time—equal to the bandwidth of the path. (A minor consideration here is overhead due to packet headers, but the problem specifically tells us to ignore that.)

Therefore, the window of 4 pkts suffices for A to achieve a rate of 5 KB/s.

Common problems:

- The problem asks for a *rate*. Thus, simply giving the bandwidth-delay product (which is a *size*) does not suffice.
- Because the problem asks for the maximum rate, simply stating the window size divided by the RTT was not enough for full credit. The point of the problem is to recognize that the rate will be the full 5 KB/s.

(d) If the bandwidth of the path from B to A drops to 100 bytes/sec, can A still achieve this rate? If so, roughly how much smaller or bigger is the new window size? If not, what is the new limit on the rate A can achieve? (5 points)

Answer: this question is subtle, and goes to the heart of just how sliding window works.

When the bandwidth in the reverse direction drops so drastically, not only does the transmission time of the acknowledgments go way up (to 1000 msec), but even more critically, *the window can only advance once per 1000 msec*, since that's the minimum spacing between two acknowledgments. Therefore A can only send one data packet per second, and has its maximum rate reduced to 500 B / 1000 msec = 500 B/s.

Common problems:

- The key to this problem is recognizing that it is not simply a matter of computing the larger RTT (and thus increased window size). However, solutions that only did this latter received partial credit if correctly done. This included needing to indicate the new window size.
- As asked for in the problem, solutions that stated A could not still achieve the same rate needed to also indicate what would be the new limit on the rate.
- Some misread this problem as asking if the rate would change *given no change in the window size*. This is clearly not the intent of the problem, since it asks for how the window would change if A can still go as fast, but received partial credit if correctly analyzed given that misinterpretation.
- Very few students got this problem fully correct.

5. **Encoding.**

Consider a link which has two levels, **hi** or **lo**. We wish to transmit the bit sequence **1110** across this link. Assume that the bit we sent most recently was **0**, and when we finished transmitting it the link was at the **lo** level.

(a) For the following patterns of signals, indicate whether they correspond to NRZ, NRZI, Manchester, or 4-bit/5-bit encoding. (2 points each)

i. **hi**, **lo**, **hi**, **hi**, **hi**.

Answer: this must be a 4-bit/5-bit encoding, since that's the only one of the encodings that requires 5 signal levels to represent 4 bits.

ii. **hi**, **hi**, **hi**, **lo**.

Answer: NRZ, which represents a 1-bit with a *hi* signal level and a 0-bit with a *lo* signal.

iii. **hi**, **lo**, **hi**, **lo**, **hi**, **lo**, **lo**, **hi**.

Answer: Manchester, the only one of the 4 encoding schemes that requires two signal levels per bit. (In particular, it represents a 1-bit as a *hi-lo* transition and a 0-bit as a *lo-hi* transition.

iv. **hi**, **lo**, **hi**, **hi**.

Answer: NRZI, which represents a 1-bit as a transition and a 0-bit as no transition. Recall that the problem states that the link was left as *lo*, so we have a transition (to *hi*), another (to *lo*), another (to *hi*), and then no transition (so remaining at *hi*).

Most students did fine on all of these.

(b) Consider an alternate encoding scheme that represents a **1** bit using a single transition and a **0** bit using two transitions.

i. Write down the **hi**/**lo** representation of **1110**. (4 points)

Answer: with this scheme, we need <one transition> followed by <one transition> followed by <one transition> followed by <two transitions>. Since the signal starts of *lo*, the encoding would be *hi* (one transition), *lo* (one transition), *hi* (one transition), *lo*, *hi* (two transitions).
There are other ways to interpret the encoding scheme. For example, if we consider that the signal will be inspected two times per cycle (in order to see whether the first half of the cycle has a transition, and then whether the second half does), then we might express the encoding as: *hi+hi, lo+lo, hi+hi, lo+hi*.

Common problems:

- Encodings that exhibited a single erroneous signal level, or included an unnecessary transition, received half credit. Those for which I was unable to determine how they related to one-versus-two transitions received one point.

ii. What advantage does this scheme offer over NRZI? (4 points)

Answer: NRZI has the problem that it does not exhibit any transitions in the presence of a string of 0-bits, which can then lead to problems with clock recovery. This scheme, however, always includes transitions, and so avoids this problem.

Common problems:

- The key for this problem was mentioning clock recovery, or, equivalently, avoiding clock drift.
- Some answers stated that this encoding allows detection of *errors*. However, clock recovery is about avoiding bit slips, not detecting them.

- Answers expressed in terms of idle time, or just “handling” of long strings of zeroes, received only partial credit.
- Answers that stated the encoding provides transitions for long strings of zeroes, without explaining why that’s an advantage (clock recovery), received 3 points.
- Some answers talked about long strings of both zeroes and ones. For NRZI, only long strings of zeroes are problematic.

iii. What disadvantage does this scheme have compared to NRZI? (4 points)

Answer: this scheme can require two signals to encode a single bit (necessary for representing two transitions, and thus a 0-bit). Therefore, like with Manchester, this scheme requires a higher clock rate, or, equivalently, results in lower efficiency.

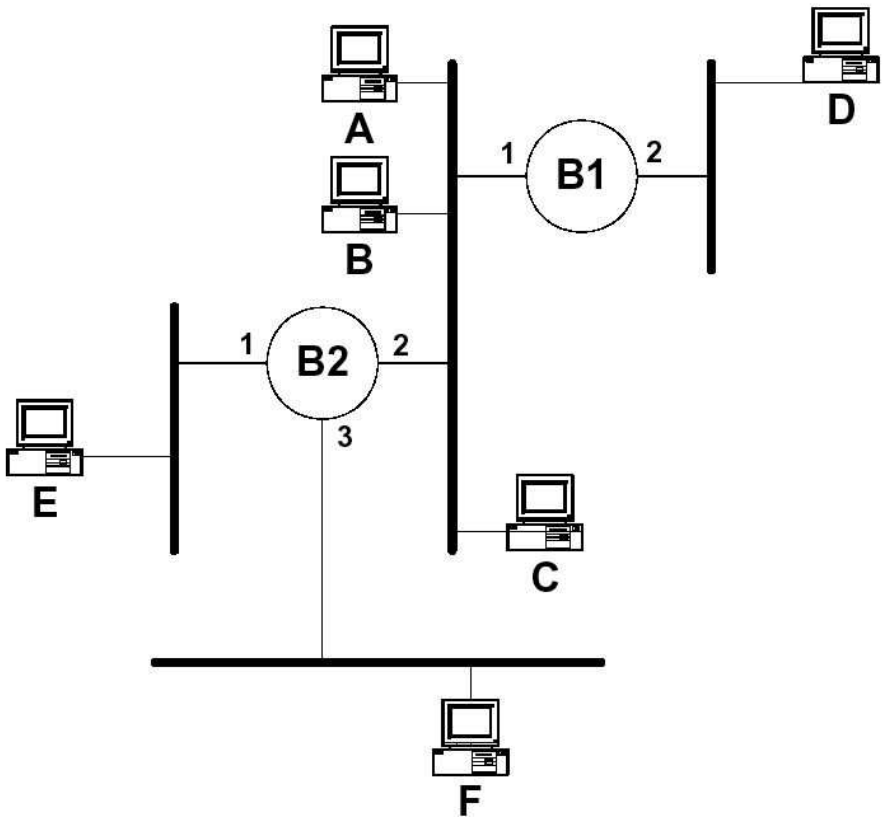
I allowed full credit for other disadvantages when adequately supported. These include: (1) ambiguity in distinguishing an encoded zero from two encoded ones (which might or might not be the case, depending on the clock framing), and (2) the requirement to use a fluctuating clock rate (which the encoding might need, depending on how one interprets what it means to exhibit one versus two transitions).

Common problems:

- Simply stating that the encoding is more complicated to implement was worth partial credit.

6. **Bridges / Switches.**

Consider the network of learning bridges (switches) shown in the following figure:



Show the forwarding table in each of the bridges after the following transmissions (which occur in the given order), assuming each table starts out empty:

(a) C sends to A

(b) F sends to E

(c) E sends to F

(d) D sends to B

Give a table for each bridge, each with two columns: destination and port number, showing how the bridge would forward traffic. (15 points)

Answer:

When C sends to A, B1 receives the frame on its port #1, and B2 on its port #2. Recall that bridges (and switches) learn the direction towards the *source* of a frame rather than its destination. So after seeing this frame, their forwarding tables look like:

Bridge B1

Destination	*Port number*
C	1

Bridge B2

Destination	*Port number*
C	2

In addition, upon receiving the frame, because neither has an entry in its forwarding table for the destination A, they both flood the frame, with B1 forwarding it out its port #2, and B2 forwarding it out its ports #1 and #3.

When F sends to E, B2 sees the frame on its port #3. It has no forwarding entry for E, so it floods the frame out its ports #1 and #2. Because of this latter forwarding, B1 also sees the frame, and the resulting tables look like:

Bridge B1

Destination	*Port number*
C	1
F	1

Bridge B2

Destination	*Port number*
C	2
F	3

Next comes the frame from E to F. B2 sees this on its port #1. However, in this case it *does* have a forwarding entry for the destination. Therefore it forwards the frame *only* to its port #3, and not its port #2. Consequently, B1 does *not* see the frame and thus won't learn the direction to forward towards E. The tables now look like:

Bridge B1

Destination	*Port number*
C	1
F	1

Bridge B2

Destination	*Port number*
C	2
F	3
E	1

Finally, D sends to B. This frame shows up on B1's port #2. It has no entry for D, so it floods the frame, which simply means it forwards it out its port #1. B2 therefore sees the frame as appearing on its port #2, and learns that that is the port to use to forward towards D.

The final tables therefore look like:

Bridge B1

Destination	*Port number*
C	1
F	1
D	2

Bridge B2

Destination	*Port number*
C	2
F	3
E	1
D	2

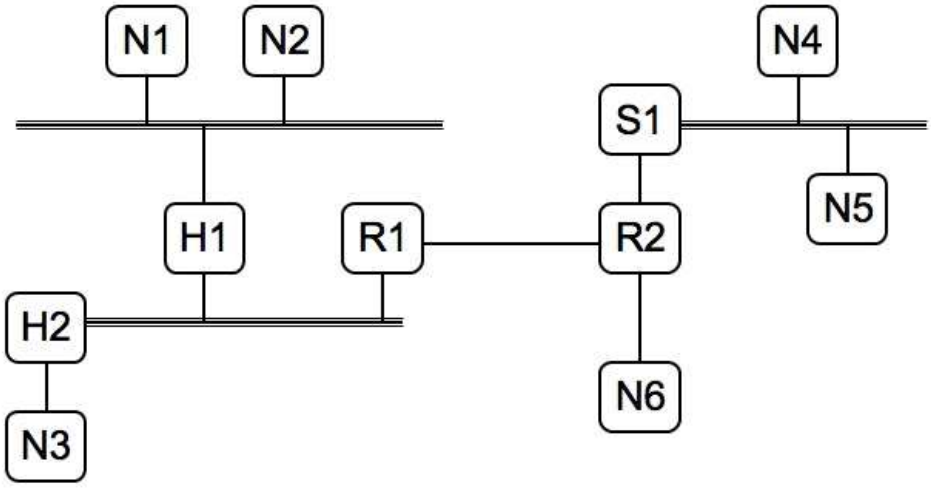
Scoring was: +2 for each correct table entry, except B2's entry for D was worth +3. -1 for each incorrect table entry.

Common problems:

- B1 does not learn about destination E, since B2 already has a forwarding entry for F, and therefore doesn't flood the frame with source MAC address E onto the link it shares with B1.
- If a solution appeared to make more sense with the tables for B1 and B2 swapped, then it was graded such, with -2 deducted for the error.
- Brute-forcing the entire table (filling it for every destination) was worth 5 total points.
- Some answers overlooked that a bridge *floods* any frame for which it doesn't have a forwarding entry.
- Some answers were in terms of bridges learning *destination* MAC addresses rather than sources. This received little credit, due to the circularity it requires.
- Any frame showing up on a link attached to a bridge gets learned. Thus, the frame from D to B gets learned by both B1 and B2, and in fact B2 will flood the frame out ports #1 and #3, since it doesn't have a forwarding entry for B.
- Port numbers are relative to each bridge. For example, C is on port 1 for B1 but port 2 for B2.
- Bridges don't learn about each other during regular forwarding. (They do when executing the Spanning Tree algorithm, but that's not included in this problem.)
- Entries in forwarding tables simply list MAC addresses and their associated ports to use in forwarding. They do not list source/destination pairs, such as C $\rightarrow$ A.
- Similarly, the ports in forwarding tables are simply the ones associated with forwarding towards the given MAC address. Bridges do not learn pairs of ports associated with where a frame came from and where it goes to.

7. **Error detection.**

Consider the following topology, where a node labeled with **N** denotes an end system, **H** denotes a hub, **S** denotes a switch and **R** denotes a router:



All links are Ethernet.

Suppose **N1** uses TCP to send a 1 KB message to **N5**. The TCP connection has already been established, so this message is sent in a single packet.

(a) When the frame holding the packet arrives at **N5**: (8 points total)

i. Does it have the same Ethernet checksum as the frame holding the packet had when **N1** sent it? If not, why not?

Answer: the Ethernet MAC addresses will be different at N5 than at N1. In particular, at N1 the source MAC address will be N1's and the destination MAC address will be R1's; while at N5, the source MAC address will be R2's and the destination MAC address will be N5's. Because the Ethernet checksum covers the MAC addresses (and also the IP header, which changes too, as discussed below), it will be *different.*
Another valid answer was to observe that elements of the IP header will have changed (see next problem part), so that will entail the Ethernet checksum changing too.
This part of the problem was worth 3 points.

Common problems:

- Simply stating that the checksum would be different without explaining why was worth 1 point.
- There was some confusion about the Ethernet **CRC** versus the term "checksum." The CRC is one *type* of checksum.

ii. Does it have the same IP checksum as the original did? If not, why not?

Answer: the IP checksum covers the IP header. While most of the IP header fields stay the same (including the source and destination IP addresses), the TTL is decremented at each hop (once at R1 and once at R2). Each router as it decrements it also recomputes the IP checksum to match the new header contents. Thus, the IP checksum at N5 will again be *different* from that in the packet when sent by N1.
This part of the problem was worth 3 points.

iii. Does it have the same TCP checksum as the original did? If not, why not?

Answer: the TCP checksum covers the TCP header and the application payload (as well as a few fields in the IP header, which we haven't discussed yet; but *not* the TTL or IP checksum). None of these value changes, so the TCP checksum will be the same at N5 as it was at N1.
This part of the problem was worth 2 points.

(b) Suppose when **H1** processes the frame it introduces a single bit error. At which nodes (i.e., any of the end systems, hubs, switches, or routers) will the errored packet appear? Here, "appear" means the frame arrives at the node's adapter, whether or not the adapter will then accept the frame. (3 points)

Answer: the Ethernet CRC checksum will detect any single-bit errors, so the damage caused by H1 will cause the frame to be discarded by any recipient after it verifies the checksum. However, the damaged frame will still *appear* at nodes H2, R1, and N3. (It appears at N3 because H2 will forward it even though it is errored, because H2 operates at layer 1 rather than layer 2.) It will *not* appear in its errored form at N2, since hubs do not resend frames onto the same segment as that from which they received them.

Since the CRC checksum fails, R1 will not further process the frame, and in particular will not forward the encapsulated packet onto R2.

Common problems:

- Simply stating that the errored packet would appear at R1 was worth 2 points. For a full 3 points, you needed to include *all* of the nodes, i.e., also H2 and N3. However, if you just included N3, I allowed credit, since H2 is implicit in N3.
- The *errored* packet will not appear at N2. Hubs do not rebroadcast packets onto the same segment as that from which they received them.

- The CRC error will already be caught at R1, *not* only by S1. This is because R1 needs to receive the Ethernet frame and decapsulate the embedded IP packet in order to do its processing. Doing so necessarily entails validating the Ethernet checksum.

(c) Suppose instead that after **N1**'s kernel constructs the TCP header, but before it constructs the IP header, a single bit in the 1 KB message gets flipped in error. At which nodes will the errored packet appear? (3 points)

Answer: the error in the application data will not be caught until N5 validates the TCP checksum. (In particular, it will not be caught by the Ethernet CRC checksum because when N1 sends the Ethernet frame it will compute the CRC over the already-corrupted message.)

The errored packet thus appears at every node in the diagram except N6 (being a router, R2 does not flood traffic, and so will only forward the packet to S1). It was optional to state that it "appears" at N1, since indeed the errored packet does show up there.

Common problems:

- As in the previous part of this problem, stating that the packet would appear at N5 was worth 2 points. For the full 3 points, you needed to state that it would appear at all the other nodes (or close; for example, omitting N4 was okay) except for N6.

(d) Suppose instead that after **N1**'s kernel constructs both the TCP and IP headers, but before it constructs the Ethernet header and trailer, a single bit in the IP header gets flipped in error. At which nodes will the errored packet appear? (3 points)

Answer: the error in the IP header will be caught when R1 validates the IP checksum after receiving the packet, so the damaged packet will show up at R1, N2, H1, H2, and N3. R1 will not forward it further, however.

Again, the packet will have a valid Ethernet CRC, though in this case that won't make a difference, other than if it had a bad CRC, then R1 would discard it a bit earlier in its processing of the incoming packet.

Common problems:

- As above, stating that it appears at R1 was good for 2 points; 3 points required stating nearly all of the other nodes, too.

(e) For this last case (**N1**'s kernel processing flips a single bit in the IP header), suppose that in addition when **H1** processes the frame it also flips a single bit in the IP header. Under what circumstances, if any, can the packet arrive at **N5**? (3 points)

Answer: the packet cannot arrive at N5. The reason is that if H1 flips a single bit, that will invalidate the Ethernet CRC, so R1 will discard the frame upon receipt.

(Recall that until H1 receives the frame, it has a valid Ethernet CRC, since the Ethernet header and trailer were constructed *after* the first bit error was caused in the IP header.)

I allowed 2 points for answers that overlooked this issue and focused on the question of what sort of second bit errors would evade detection by the IP header checksum. One form is an error that inverts the *same* bit as was previously flipped, thus inadvertently undoing the original damage, and making the IP header checksum valid again. Another form of error would be one of the types of two-bit errors that the IP header checksum is too weak to detect, such as the same bit position in two separate 16-bit words.

Common problems:

- Neither the Ethernet CRC nor the Internet checksum provide any form of *error correction*, nor do they use *parity*.

8. **Framing.** In networking, being able to determine the beginning and ending of a message is termed *framing*. Framing issues come up at both lower layers of the networking stack and higher layers.

   (a) One style of framing is to precede a message with a *count* of its length. (4 points)

      i. When using counts for link-layer framing, what problem can arise?

      Answer: if the count becomes corrupted, then not only does the current frame get misrecovered, but ensuing frames too: the receiver will locate an incorrect length count for the next frame, and thus for the frame after it, and so on.
      Another valid answer was that the formatting of the length field (for example, if it is restricted to a certain number of bits) can impose limits on the size of link-layer frames.

      Common problems:

      - Simply stating that errors can occur was worth 1 point.
      - Stating that the length might exceed what the physical medium supports was worth 1 point, since this problem should not arise in practice.

      ii. When using this style to send application-layer messages over TCP, does this problem still arise? Why or why not?

      Answer: because TCP provides reliable delivery, this problem does *not* arise for framing *application-layer* messages.
      It sufficed to mention that TCP protects the application-layer data (not just its header) with a checksum.

      Common problems:

- Note that the question is about the framing of *application-layer messages*. These use the (reliable) service provided by TCP. Some answers instead discussed what issues arise for TCP's own internal framing (which relies on IP's per-packet framing).
- Some interpreted the question as asking whether such a problem occurring at the link-layer would cause trouble for TCP too. If the explanation given was correct, this was given half credit.
- Stating that a corruption might occur that happens to pass TCP's checksum was worth 1 point, since this is unlikely to occur in practice.

(b) Another style of framing is to use a *sentinel* value. (6 points)

i. When using sentinels for link-layer framing, what problem can arise?

Answer: the value used for the sentinel might appear in the frame's data and become confused with the indicator of the end of the frame.

Common problems:

- Stating that the sentinel could become corrupted and thus missed was worth 1 point, as it doesn't go to the heart of the general issue with using sentinels.

ii. Briefly describe a solution for this problem.

Answer: some form of *escaping* needs to be used in order to tell embedded instances of the sentinel value from the true instance that comes at the end of the frame.
Another valid answer was to discuss use of encodings that have unused values (such as 4-bit/5-bit); one of the otherwise unused values can then serve as the sentinel value.

Common problems:

- Simply giving a specific example of an escaping scheme without identifying the general concept of escaping was worth 1 point.

iii. When using this style to send application-layer messages over TCP, does this problem still arise? Why or why not?

Answer: the problem *does* arise, since TCP provides a pure "byte stream" to the application layer. This service does not include any built-in form of escaping or unused values.

Common problems:

- Answers stating that TCP delivers *packets* that the application still needs to reassemble received half credit, since TCP instead delivers a byte stream (no packet boundaries), though the application has to recover its message framing from the byte stream by itself.

- The answer needed to be in terms of application-layer messages, so comments such as "use unmapped 4-bit/5-bit encoding values for the sentinel" are inapplicable.

(c) Consider internetwork-layer datagrams sent using IP. (6 points)

i. What style of framing does IP use for the datagrams it transmits?

Answer: the IP header includes a *count* reflecting the length of the datagram. Another valid answer was to discuss the framing IP uses for fragmentation.

ii. What solution, if any, does it use to address the problem you identified above that can arise for this style?

Answer: IP protects the integrity of its header using a checksum. Thus, if a length value becomes corrupted, it knows to discard the packet.

Common problems:

- IP's checksum is *not* a CRC.

iii. How does a host that receives an IP datagram know what type of transport protocol information is inside the datagram?

Answer: the IP header includes a *protocol* field that codifies what type of transport protocol header follows the IP header.

Common problems:

- The header field is *not* the "Type of Service" field.

(d) Application-layer protocols are free to use a wide range of framing techniques. (4 points)

i. What sort of framing does SMTP use to figure out where an email message being transferred ends?

Answer: SMTP uses a *sentinel* in the form of a line consisting of a single period (".")

ii. How does FTP indicate when it has finished transferring a file?

Answer: FTP closes the data connection to indicate it is done with the transfer. Another valid answer was to note that the FTP server will return a status message.

Common problems:

- Since FTP's termination is somewhat atypical, it did not suffice to simply state that it uses a sentinel.
- While FTP does state the length of the file when beginning a transfer (if known), this is not used for its framing, and occurs prior to the transfer, and thus is not the same as the status message

iii. What sort of framing does HTTP use for header information in requests and responses?

Answer: HTTP uses sentinels to frame its header information. It marks the end of each header using a CRLF, and the end of all the headers by an empty line terminated by a CRLF.

Common problems:

- It was not sufficient to simply give the syntax of a GET request or a status reply, since the problem asked about *header* information.

iv. Give an example of a type of framing HTTP uses for figuring out where an *item* being transferred ends. (Here an "item" refers to the object returned in response to a `GET` request, rather than the headers returned by the server.)

Answer: HTTP can use a length field (the `Content-Length` header, or for "chunking"), or a sentinel (the server closing the connection after transferring the item, for non-persistent connections).

Common problems:

- The problem specifically asks about items rather than headers. The end of items are *not* framed using CRLF.
- While items that happen to be HTML documents will tend to end with `</HTML>`, the presence of this token is not used by *HTTP* for its framing (and can't be, because items might be of some type other than HTML).