Solutions for Homework #2[1]

EE122: Introduction to Communication Networks

(Fall 2007)

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

Vern Paxson / Lisa Fowler / Daniel Killebrew / Jorge Ortiz

1. Problems from Kurose & Ross.

   (a) **Chapter 2 P4**

   i. The object requested by the broweser is `/cs453/index.html`. The `Host:` field indicates the server's name, and thus we can infer that the original URL was `http://gaia.cs.umass.edu/cs453/index.html`.

   ii. The browser is running HTTP version 1.1, as indicated just before the first <cr><lf> pair.

   iii. The browser is requesting a persistent connection, as indicated by the `Connection: keep-alive` header.

   iv. This is a trick question. This information is not contained in an HTTP message anywhere. So there is no way to tell this from looking at the exchange of HTTP messages alone. One would need information from the IP datagrams (that carried the TCP segment that carried the HTTP GET request) to answer this question.

   (b) **Chapter 2 P5**

   i. The status code of `200` and the phrase `OK` indicate that the server was able to locate the document successfully. The reply was provided on `Tuesday, 07 Mar 2006 12:39:45GMT` (Greenwich Mean Time).

   ii. The document `index.html` was last modified on `Saturday 10 Dec 2005 18:27:46 GMT`.

[1] Version 2, Oct 11 2007: Fixes an incorrect solution for Problem 3(d). Our apologies, and kudos to Rick Schmidt for spotting the error.

iii. There are 3,874 bytes in the document being returned, as indicated by the `Content-Length:` header.

iv. The first five bytes of the returned document are `<!doc`, as this is what comes after the <cr><lf><cr><lf> pair that terminates the reply headers.
The server agreed to a persistent connection, as indicated by the `Connection: Keep-Alive` header.

(c) **Chapter 5 P17**

i. We want to maximize the distance, such that

$$\frac{1}{1+5\left(\frac{d_{prop}}{d_{trans}}\right)} = 0.5$$

Simplifying, we have $\frac{d_{prop}}{d_{trans}} = 0.2$.
A 64-byte frame is 512 bits in size. However, we're also directed to take into account the preamble, which is another 8 bytes, for a total of 576 bits. Given this, we calculate:

$$d_{trans} = (576 \text{ bits})/(10^8 \text{ bits/sec}) = 5.76\mu\text{sec}.$$

From these two expressions we have

$$d_{prop} = 0.2 \cdot 5.76\mu\text{sec} = 1.15\mu\text{sec}.$$

The expression for d_{prop} represents the maximum time it takes for a signal to propagate between any two nodes. Given the speed of propagation along the link of $1.8 \cdot 10^8$ m/sec, this means that the distance the signal can propagate is:

$$1.8 \cdot 10^8\text{m/sec} \cdot 1.15\mu\text{sec} = 207 \text{ m}.$$

The problem, however, asks for the maximum distance beteen a node and the *hub*. Since a hub is a layer-1 device, the constraint on total size means that nodes must be no further than 103.5 m from the hub, to ensure that the total distance between nodes does not exceed 207 m.
This value is less than the 100 Mbps Ethernet standard of 200 meters, but the total propagation distance slightly exceeds the maximum allowed by the standard.

ii. For a transmitting node *A* to detect another node transmitting during *A*'s transmission, d_{trans} must be greater than $2 \cdot d_{prop}$. This is enough time to ensure that if *B* starts transmitting in the middle of *A*'s message (before *A*'s message has reached *B*), *A* will still see *B*'s signal before *A* has finished transmitting, and thus will detect the collision. $2 \cdot d_{prop} = 2.3\mu$sec. Because $2.3\mu\text{sec} < d_{trans} = 5.76\mu$sec, *A* will detect *B*'s signal before *A* finishes transmitting.

2. **The Design of IP.**

The information about the Record Route option can be found on page 20 of RFC 791.

(a) The type code is 7.

(b) Other information present in the option is the option length and the pointer. The length indicates the total number of bytes taken up by the Record Route option, starting with the type code and ending with the last byte of route data (inclusive). The pointer indicates at which byte the next route address should be stored (where the next router should place its IP address). The value is relative to the start of the whole Record Route, so the minimum value of the pointer field is 4. The type code, length, and pointer take up 3 bytes.

(c) The length of an IP header in 4-byte words is specified in the 4-bit "header length" field (termed **IHL** on page 11 of RFC 791). The maximum 4-bit value this field can express is 15, so the largest possible IP header is 60 bytes in length.

Because IP headers include at least 20 bytes of non-optional information, there are 40 bytes remaining for the Record Route option. 3 bytes are consumed by type code, length, and pointer, leaving 37 bytes for the actual route data. At 4 bytes per IP address, we can record at most 9 router addresses.

(d) If the route is too long, the Record Route will be full, and the router will forward the packet without inserting its address. If there is room for only part of an address, then RFC 791 states that the datagram is considered to be ill-formed and the router discards it. In either case, the router has the option to send an ICMP message (not yet covered in lecture) to the originator to inform it of the problem.

3. **Comparison of Stop-and-Wait and Sliding Window.**

First, some preliminaries:

- Each data packet holds 1,460 bytes of payload, so to transfer 122,000 bytes of payload will require 83 full-sized packets plus one 820-byte packet ($83 \cdot 1460 + 820 = 122000$).
- For a full-sized packet, the time to send it from Berkeley to Los Angeles is:

$$T_{\text{data}} = 12 \text{ ms} + \frac{(1460 + 40) \text{ bytes} \cdot 8 \text{ bits/byte}}{12 \text{ Mbps}} = 13 \text{ ms}$$

- The same calculation for the 820-byte packet yields a latency of $T_{\text{partial}} = 12.573$ ms.
- Similarly, the time to send back a 40-byte acknowledgement is $T_{\text{ack}} = 12.027$ ms.

- The round-trip time (RTT) for a full-sized packet is therefore:

$$\text{RTT} = T_{\text{data}} + T_{\text{ack}} = 13 \text{ ms} + 12.027 \text{ ms} = 25.027 \text{ ms}$$

Given these, we proceed as follows:

(a) To send the file with Stop-and-Wait, each full-sized round requires time RTT $= T_{\text{data}} + T_{\text{ack}}$. We include the final partial packet to this to get:

$$T_{\text{stop-and-wait}} = (83 * 25.027) + (12.573 + 12.027) = 2101.8 \text{ ms}$$

(b) The throughput is the amount of data transferred divided by the time it took to send it, so:

$$\text{Throughput} = 122000\text{B}/2.1018\text{s} = 58.045 \text{ KBps}$$

(c) An important insight is that each acknowledgment immediately allows us to send another packet (because the window slides forward). Therefore, for a window of 8 packets, packets #1, #9, #17, and so on are each sent with an interval of $T_{\text{data}} + T_{\text{ack}}$ between them. Similarly for #2, #10, #18, etc., and #3, #11, #19, etc..

Let us now consider the last packet we send, which is #84 (and is a partial packet rather than a full one like the others). It is part of the series #4, #13, #21, etc..

Packet #4 is sent just after transmitting #1-3, so at time:

$$T_4 = \frac{3 * (1460 + 40) \text{ bytes} \cdot 8 \text{ bits/byte}}{12 \text{ Mbps}} = 3 \text{ ms}$$

Packet #13 is sent at $T_4 +$ RTT; #21 at $T_2 + 2 \cdot$ RTT, etc.

In particular, packet #84 is sent at

$$T_4 + 10 \cdot \text{ RTT} = 3 + 10 \cdot 25.027 = 253.27 \text{ ms} \quad (1)$$

Since this is the last packet, we are done when its acknowledgment arrives, which takes

$$T_{\text{partial}} + T_{\text{ack}} = 12.573 + 12.027 = 24.600 \text{ ms} \quad (2)$$

Therefore, the entire transmission takes $253.27 + 24.6 = 277.87$ ms. The corresponding throughput is

$$\text{Throughput} = 122000\text{B}/0.27787\text{s} = 439.06 \text{ KBps}$$

This reflects a gain of nearly 8 times—not quite a full factor of 8 because the window doesn't allow us to fully parallelize transmission, but rather to pipeline it.

(d) The bandwidth-delay product of this path is

$$12 \text{ Mbps} * \text{RTT ms} = 300.32 \text{ Kb} = 37.5405 \text{ KB}$$

For 1500 byte packets—which again each carry 1,460 bytes of data—a window size of 26 packets totals 39,000 bytes. (Note that an understandable minor error here would be to instead use a window size of 25 packets, which is only 40 bytes below the full bandwidth-delay product.)

For a 122,000 byte file, this results in 3 full window's worth of packets, followed by 6 more packets (the last of which is a partial packet, again of size 820 bytes).

We might employ the method used above to calculate the total transfer time and corresponding throughput. The 84^{th} packet belongs to the sequence #6, #32, #58, #84. However, a subtlety arises, as follows. When the ACK for packet #1 arrives, it allows us to send packet #27 (since the window is 26 packets in size). Note that this occurs *while we are still in the process of transmitting packet #26*, because 26 packets represent a bit *more* data than required to keep the forward path continuously busy.

We can see this effect by considering that we start sending packet #26 at a time 25 msec after we began sending packet #1 (since each packet has a transmission time of 1 msec). We will finish transmitting packet #26 at time 26 msec. However, the RTT for this path is 25.027 msec. Thus, the ACK arrives shortly after we have *begun* sending packet #26, and before we have finished doing so.

This means that we send packet #27 directly after we finish sending packet #26, so at time 26 msec, rather than at time we would calculate based on the RTT.

Therefore, a window of 26 packets means that every packet goes out 1 msec after its predecessor. In particular, we start sending packet #84 at time 83 msec. The time required to transmit #84 and receive its ACK added to the send time of #84 gives us the total transmission time:

$$T_{84} + T_{\text{partial}} + T_{\text{ack}} = 83 + 24.6 = 107.6\text{ms}$$

Throughput is therefore $122000\text{B}/107.6 \text{ ms} = 1.133829 \text{ MBps} = 9.070632 \text{ Mbps}$. It's not the full 12 Mbps because we spend about half an RTT waiting for the first data packet to arrive at the receiver, and another half an RTT waiting for the final ack to arrive back at the sender.

The general observation is that a window size greater or equal to the bandwidth-delay product allows us to fully use the capacity of the network path. There is no "down time" spent waiting for acknowledgments to advance the window so we can send more; we're always able to send at the rate the network can sustain.

(e) Setting the window any higher won't change the throughput we can achieve, since the window computed using the bandwidth-delay product already allows us to send at the full rate the path can support.

More generally, using the bandwidth-delay product for the window allows us to completely "fill the pipe," namely to keep the forward transmission path continuously occupied. Any larger window can't enable us to go any faster, since we're already going as fast as the network can possibly let us go. (The larger window *can* lead to larger "queues" inside the network, since we give the network more load to process in a given amount of time. Ironically, this can lead to some of our packets being dropped due to exhausted queue space, which, as we'll see when we study congestion control, can actually cause the transfer to go much *slower*.)

4. **Encoding.**

   (a) Below is a table of all 5-bit codes with at most one leading and at most one trailing 0. `X`'s represent a 0 or 1:

code pattern	number of possibilities
`01X10`	2
`01XX1`	4
`1XX10`	4
`1XXX1`	8

   $\Rightarrow$ 18 possible sequences.

   (b) Because there are only $2^4 = 16$ possible 4-bit sequences, it is possible to map all 4-bit sequences to the aforementioned 5-bit sequences (since $18 > 16$).

5. **Parity.**

   (a) Let M be the set of all 7-bit messages with a single 1 bit.

$$\begin{aligned} M \quad = \{ \quad & 0000001, 0000010, 0000100, 0001000, \\ & 0010000, 0100000, 1000000 \quad \} \end{aligned}$$

   Any message $m_1 \in M$ can be transmuted into any other message $m_2 \in M (m_1 \neq m_2)$ with a 2-bit error—the first bit error sets m_1's lone 1-bit to 0, and the second bit error then sets the long 1-bit present in m_2.

   To detect a 2-bit error, each message in M must have a different error code. However, with 2 bits for error detection we can only express $2^2 = 4$ error codes. Therefore, there must exist $m_1, m_2 \in M, m_1 \neq m_2$, such that m_1 and m_2 share the same error code. If m_1 is transmuted to m_2 due to a 2-bit error, since their error codes are the same, this 2-bit error cannot be detected.

Therefore, there is no error detection code of size 2 bits that can detect all 2-bit errors for 7-bit messages.

(b) Because we are not finding the minimal N, we can make the problem easier through some simplifications.

i. First, here's an approach based on pretty much brute force:
Let M be the set of all N-bit messages with a single 1 bit. There are N elements in M. Any message $m_1 \in M$ can be transmuted into any other message $m_2 \in M(m_1 \neq m_2)$ with a 2-bit error. For example, assume 0000....0001 had bit flips at the last two bits, then it becomes 0000....0010. Since this is a 2-bit error, any code that detects up to 8-bit errors must detect it.
To detect it, however, each message in M needs to have a different error code (same as in the previous problem). However, with 32 bits we can only generate 2^{32} different error codes. Therefore, if $N > 2^{32}$, there must exist $m_1, m_2 \in M(m_1 \neq m_2)$ such that m_1 and m_2 share the same error code. When m_1 is transmuted to m_2 with a 2-bit error, since they have the same error code, we cannot detect the 2-bit error.
This then shows that if $N = 2^{32} + 1$, there is no error detection code of size 32 bits that can detect all errors altering up to 7 bits, since such codes cannot even detect all 2-bit errors.

ii. Here's an approach that arrives at a much tighter bound:
Consider the set S composed of all N-bit messages with 3 bits set (3 bits are 1, the rest are 0). Clearly, we can transform any member of S into any other member using a 6-bit error, just as two bits sufficed for part (a) above.
We can calculate the cardinality of S using the combinatoric "choose" function, because we have N different bits, and we are choosing exactly 3 of them to be 1 (those unfamiliar with "choose" can read `http://en.wikipedia.org/wiki/Choose`). Now we want to pick N such that the size of S exceeds 2^{32}:

$$\left(\begin{array}{c} N \\ 3 \end{array} \right) > 2^{32}$$

Solving for N, we arrive at a minimal value $N = 2955$ bits. Because there are more messages in *S* than there are possible 32-bit error codes, a 32-bit error code cannot detect all errors altering up to 6-bits in a 2955-bit block. If our 2955-bit block is so large that a 32-bit error code is not large enough to find all 6-bit errors, then it is definitely not large enough to find all 7-bit errors.

6. **Email.**

(a) The commands used are as follows:

```
HELO imail.eecs.berkeley.edu
MAIL FROM:<ee122-tb@imail.eecs.berkeley.edu>
RCPT TO:<dank@eecs.berkeley.edu>
DATA
From: Golden Bear <ee122-tb@imail.eecs.berkeley.edu>
To: EE122 <dank@eecs.berkeley.edu>
Subject: My Answer To Homework #2, Problem #4
Hi.
I am Golden Bear.
.
QUIT
```

(b)
```
RCPT to:<vern@icir.org>
```

returns

```
550 5.7.1 <vern@icir.org>... Relaying denied
```

calmail.berkeley.edu does not allow arbitrary hosts to relay through it. (Note, it's possible that you used a host and/or <From> address for which it actually would allow the relaying. We did not mark off in this case.)

7. **DNS.**

(a) The namer servers visited:

i. *a.root-servers.net* as the initial root server

ii. *m3.nstld.com* (or another TLD serer) to resolve names in `edu`.

iii. *adns1.berkeley.edu* (or another of UCB's servers: *ucb-ns.nyu.edu*, *ns.v6.berkeley .edu*, *dns2.ucla.edu*, *adns2.berkeley.edu*, *phloem.uoregon.edu*) to resolve names in `berkeley.edu`.

iv. At this point, the reply from `adns1.berkeley.edu` already includes an Answer. However, if you chose another of the servers, it may or may not have provided an answer. (E.g., `ucb-ns.nyu.edu` does, but `dns2.ucla.edu` does not.)

If not, then you proceed one more level, choosing say *cgl.ucsf.edu* (or another of EECS's servers: *ns.eecs.berkeley.edu*, *ns.cs.berkeley.edu*, *vangogh.cs.berkeley .edu*, *adns1.berkeley.edu*, *adns2.berkeley.edu*, *vangogh.cs.berkeley.edu*) to resolve names in `cs.berkeley.edu`.

(b) The Answer returns an A record with a value of 128.32.42.35.

(c) The same sequence as above occurs, except at step 3 both no Answer is returned nor is any further NS record. (The server will return an SOA [Source of Authority] record, which we haven't discussed, but this is basically its way of saying that the name definitely does not exist.)

(d) We perform the command `dig +norecurse -x 128.32.42.35` and find that yes, the reverse mapping for sphere.cs.berkeley.edu's address does match its hostname.

(e) *ns.cs.berkeley.edu* manages the RR (or another of EECS's servers: *ns.eecs.berkeley.edu*, *vangogh.cs.berkeley.edu*, *adns1.berkeley.edu*, *adns2.berkeley.edu*, *vangogh.cs.berkeley.edu*).

The zone is `42.32.128.in-addr.arpa`.