Problem 1

Suppose Bob joins a BitTorrent torrent, but he does not want to upload any data to any other peers (so called free-riding).

(a) Bob claims that he can receive a complete copy of the file that is shared by the swarm. Is Bob's claim possible? Why or why not?

(b) Bob further claims that he can further make his "free–riding" more efficient by using a collection of multiple computers (with distinct IP addresses) in the computer lab in his department. How can he do that?

Write your solution to Problem 1 in this box

(a) Yes, it is possible.
Bob joins a P-2-P connection. Therefore, as long as there are peers who want to share the requested file and stay in the swarm for enough time, Bob would be able to receive a complete copy of the file.
He can always receive unchoking data from other peers in the swarm through optimization

(b) Yes, this claim is also true
To achieve it, Bob can run a client on each machine, and join the swarm independency. He can "free-ride" on each server, and then combine all the pieces from each client into one single file.
(in P2P connection, each client can be a server)

Problem 2

Suppose you have a new computer just set up. `dig` is one of the most useful DNS lookup tool. You can check out the manual of `dig` at `http://linux.die.net/man/1/dig`. A typical invocation of `dig` looks like: `dig @server name type`.

Suppose that on April 19, 2017 at 15:35:21, you have issued "`dig google.com a`" to get an IPv4 address for `google.com` domain from your caching resolver and got the following result:

```
; <<>> DiG 9.8.3-P1 <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17779
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 4, ADDITIONAL: 4

;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             239     IN      A       172.217.4.142

;; AUTHORITY SECTION:
google.com.             55414   IN      NS      ns4.google.com.
google.com.             55414   IN      NS      ns2.google.com.
google.com.             55414   IN      NS      ns1.google.com.
google.com.             55414   IN      NS      ns3.google.com.

;; ADDITIONAL SECTION:
ns1.google.com.         145521  IN      A       216.239.32.10
ns2.google.com.         215983  IN      A       216.239.34.10
ns3.google.com.         215983  IN      A       216.239.36.10
ns4.google.com.         215983  IN      A       216.239.38.10

;; Query time: 81 msec
;; SERVER: 128.97.128.1#53(128.97.128.1)
;; WHEN: Wed Apr 19 15:35:21 2017
;; MSG SIZE rcvd: 180
```

(a) What is the discovered IPv4 address of `google.com` domain?

(b) If you issue the same command 1 minute later, how would "ANSWER SECTION" look like?

(c) When would be the earliest (absolute) time the caching resolver would contact one of the `google.com` name servers again?

(d) If the client keeps issuing `dig google.com A` every second, when would be the earliest (absolute) time the caching resolver would contact one of the .com name servers?

Write your solution to Problem 2 in this box

(a) it's 172.217.4.142
under the ANSWER SELECTION

(b) Everything stays the same, except for that
239 becomes 179

(c) After 239 seconds,
in terms of absolute time:
Wed Apr 19 15:39:20 2017

(d) Until the authority section cache cleans,
which is 55414 seconds.
in terms of absolute time
Wed Apr 20 06:58:55 2017

Problem 3

The sender side of *rdt3.0* simply ignores (that is, takes no action on) all received packets that are either in error or have the wrong value in the acknum field of an acknowledged packet. Suppose that in such circumstantces, *rdt3.0* were simply to retransmit the current data packet. Would the protocol still work ? (Hint: Consider what would happen if there were only bit errors; there are no packet losses but premature timeouts can occur. Consider how many times the *nth* packet is sent, in the limit as n approaches infinity).

Write your solution to Problem 3 in this box

Yes, this protocol would still work,

but very inefficient compared with the original rdt 3.0.

If the Ack is lost or corrupted, retransmittion would be the right solution. If acknum is not as expected, the retransmittion would only delay the following packets, but would not cause any logical error.

In terms of efficiency, the original rdt 3.0 only sends twice the packet that causes pre-mature timeout. But this protocal sends every packet after it twice.

Problem 4

Consider a reliable data transfer protocol that uses only negative acknowledgments. Suppose the sender sends data only infrequently. Would a NAK–only protocol be preferable to a protocol that uses ACKs? Why? Now suppose the sender has a lot of data to send and the end–to–end connection experiences few losses. In this second case, would a NAK–only protocol be preferable to a protocol that uses ACKs? Why?

Write your solution to Problem 4 in this box

For infrequent transmission, NAK-only protocol is not preferable

In NAK-only protocol. receiver only realizes packet loss after receiving the next packet, and thus send a NAK to the sender. (i-1, i+1, missed a packet → packet loss)

If the transmission is infrequent, the time to recover the lost packet is even longer. because the sender only retransmit packet i after receiver receives packet i+1. Therefore, NAK-only protocol is not preferred.

However, for frequent data transmission, the lost packets would be recovered quickly. Moreover for reliable data transfer, NAK signal is infrequently sent and Ack is never sent, which largely improves the efficiency, compared with Ack-only.

So in this case, NAK-only is preferable.

Problem 5

Suppose an application uses *rdt3.0* as its transport protocol. As the stop–and–wait protocol has very low channel utilization (shown in the lecture), the designers of this application let the receiver keep sending back a number (more than two) of alternating ACK 0 and ACK 1 even if the corresponding data have not arrived at the receiver. Would this application design increase the channel utilization? Why? Are there potential problems with this approach? Explain.

Write your solution to Problem 5 in this box

Yes, by replying before actually receiving the packets, it causes the sender to send pipelined packets to the receiver. Because, multiple packets might being transfered before the receiver receives the first packet.

Potential problem is the sender cannot retransmit in cases where the packets are corrupted or lost. Because the response are not technically ACK or NAK. It's also difficult to track the sequence of transmitted packets if the transmission speed is slow and multiple packets stay in the channel at the same time. Because this approach doesn't set up a buffer / window size.