CS 655: Introduction to Computer Networks
Fall 2020
Solutions to Homework 2

1. (15 points)

a) The time to transmit an object of size L over a link of rate R is L/R. The average time is the average size of the object divided by R:

Δ = (850,000 bits)/(15,000,000 bits/sec) = .0567 sec

The traffic intensity on the link is given by $\beta\Delta$=(16 requests/sec)(.0567 sec/request) = 0.907. Thus, the average access delay is (.0567 sec)/(1 - .907) ≈ .6 seconds. The total average response time is therefore .6 sec + 3 sec = 3.6 sec.

b) The traffic intensity on the access link is reduced by 60% since the 60% of the requests are satisfied within the institutional network. Thus the average access delay is (.0567 sec)/[1 – (.4)(.907)] = .089 seconds. The response time is approximately zero if the request is satisfied by the cache (which happens with probability .6); the average response time is .089 sec + 3 sec = 3.089 sec for cache misses (which happens 40% of the time). So the average response time is (.6)(0 sec) + (.4)(3.089 sec) = 1.24 seconds. Thus the average response time is reduced from 3.6 sec to 1.24 sec.

2. (15 points)

Note that each downloaded object can be completely put into one data packet. Let Tp denote the one-way propagation delay between the client and the server.

First consider parallel downloads using non-persistent connections. Parallel downloads would allow 10 connections to share the 150 bits/sec bandwidth, giving each just 15 bits/sec. Thus, the total time needed to receive all objects is given by:

(200/150+Tp + 200/150 +Tp + 200/150+Tp + 100,000/150+ Tp)
+ (200/(150/10)+Tp + 200/(150/10) +Tp + 200/(150/10)+Tp + 100,000/(150/10)+ Tp)
= 7377 + 8*Tp (seconds)

Now consider a persistent HTTP connection (with pipelining). The total time needed is given by:

(200/150+Tp + 200/150 +Tp + 200/150+Tp + 100,000/150+ Tp)
+ (200/150)+Tp + 10*(100,000/150)+ Tp
=7339 + 6*Tp (seconds)

Assuming the speed of light is $300*10^6$ m/sec, then $Tp=10/(300*10^6)=0.03$ microsec. Tp is therefore negligible compared with transmission delay.

[Note: under persistent with pipelining, the requests for the ten embedded objects are sent back-to-back, and once the first request arrives, the first embedded object is transmitted, which takes 100,000/150= 666.66 seconds, and meanwhile all remaining 9 requests arrive as it takes 9* (200/150)+Tp ~ 12 seconds and so the transmission of the 9 remaining embedded objects continues after the transmission of the first embedded object is complete.]

Thus, we see that persistent HTTP is not significantly faster (less than 1 percent) than the non-persistent case with parallel download.

3. (10 points)

For calculating the minimum distribution time for client-server distribution, we use the following formula:

$$D_{cs} = max \{NF/u_s, F/d_{min}\}$$

Similarly, for calculating the minimum distribution time for P2P distribution, we use the following formula:

$$D_{P2P} = max\{F/u_s, F/d_{min}, NF/(u_s + \sum_{i=1}^{N} u_i)\}$$

Where, F = 15 Gbits = 15 * 1024 Mbits
u_s = 30 Mbps
$d_{min} = d_i$ = 2 Mbps

Note: The following calculations take all 1K = 1024. Your numbers may slightly differ, for example, if you take 1Kbps to be 1000 bps.

Client Server

			N	
		10	**100**	**1000**
	300 Kbps	7680	51200	512000
u	**700 Kbps**	7680	51200	512000
	2 Mbps	7680	51200	512000

Peer to Peer

			N	
		10	**100**	**1000**
	300 Kbps	7680	25904	47559
u	**700 Kbps**	7680	15616	21525
	2 Mbps	7680	7680	7680

4. (10 points)

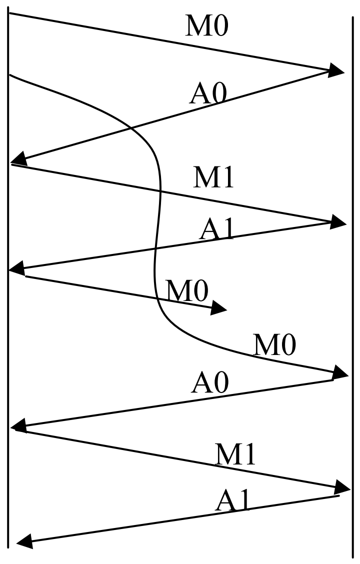
Yes. His first claim is possible, as long as there are enough peers staying in the swarm for a long enough time. Bob can always receive data through optimistic unchoking by other peers.

His second claim is also true. He can run a client on each host, let each client "free-ride," and combine the collected chunks from the different hosts into a single file. He can even write a small scheduling program to make the different hosts ask for different chunks of the file. This is actually a kind of Sybil attack in P2P networks.

5. (10 points)

Suppose the sender is in state "Wait for call 1 from above" and the receiver (the receiver shown in the homework problem) is in state "Wait for 1 from below." The sender sends a packet with sequence number 1, and transitions to "Wait for ACK or NAK 1," waiting for an ACK or NAK. Suppose now the receiver receives the packet with sequence number 1 correctly, sends an ACK, and transitions to state "Wait for 0 from below," waiting for a data packet with sequence number 0. However, the ACK is corrupted. When the rdt2.1 sender gets the corrupted ACK, it resends the packet with sequence number 1. However, the receiver is waiting for a packet with sequence number 0 and (as shown in the home work problem) always sends a NAK when it doesn't get a packet with sequence number 0. Hence the sender will always be sending a packet with sequence number 1, and the receiver will always be NAKing that packet. Neither will progress forward from that state.

6. (10 points)



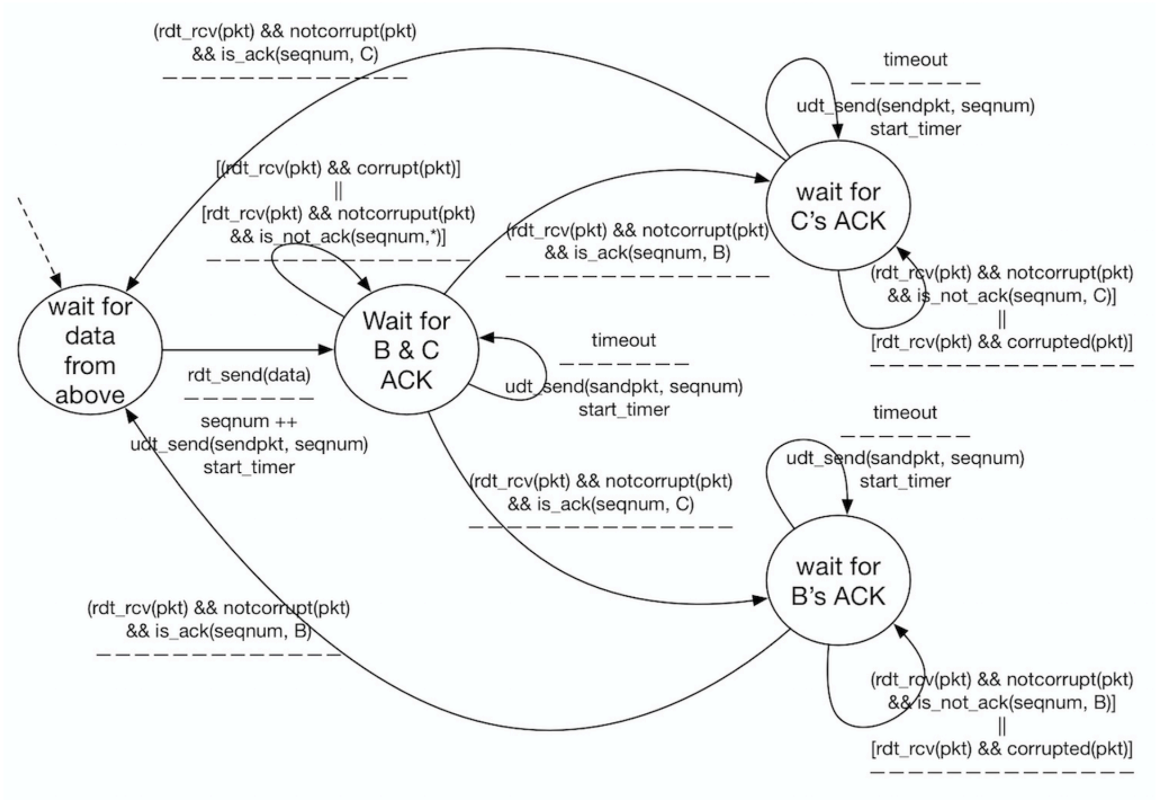
old version of M0 accepted!

7. (15 points)

This problem is a variation on the simple stop and wait protocol (rdt3.0). Because the channel may lose messages and because the sender may resend a message that one of the receivers has already received (either because of a premature timeout or because the other receiver has yet to receive the data correctly), sequence numbers are needed. As in rdt3.0, a 1-bit sequence number will suffice here.

The sender and receiver FSM are shown in Figure 3. In this problem, the sender state indicates whether the sender has received an ACK from B (only), from C (only) or from neither C nor B. The receiver state indicates which sequence number the receiver is waiting for.

sender


(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_ack(seqnum, C)
timeout
udt_send(sendpkt, seqnum)
start_timer
[(rdt_rcv(pkt) && corrupt(pkt)]
||
[rdt_rcv(pkt) && notcorruput(pkt)
&& is_not_ack(seqnum,*)]
wait for C's ACK
(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_ack(seqnum, B)
(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_not_ack(seqnum, C)]
||
[rdt_rcv(pkt) && corrupted(pkt)]
wait for data from above
Wait for B & C ACK
timeout
udt_send(sandpkt, seqnum)
start_timer
rdt_send(data)
seqnum ++
udt_send(sendpkt, seqnum)
start_timer
timeout
udt_send(sandpkt, seqnum)
start_timer
(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_ack(seqnum, C)
wait for B's ACK
(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_ack(seqnum, B)
(rdt_rcv(pkt) && notcorrupt(pkt)
&& is_not_ack(seqnum, B)]
||
[rdt_rcv(pkt) && corrupted(pkt)]

receiver B

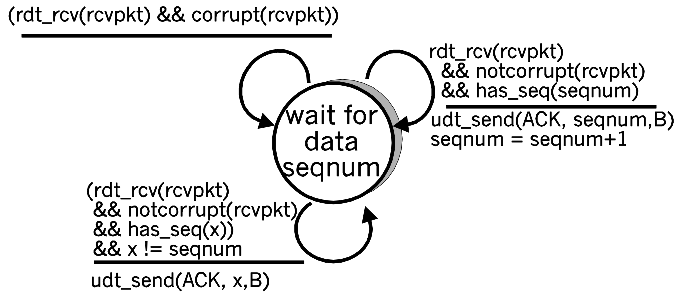


Figure 3. Sender and receiver for Problem 3.19(Problem 19)

8. (15 points)

Because the A-to-B channel can lose request messages, A will need to timeout and retransmit its request messages (to be able to recover from loss). Because the channel delays are variable and unknown, it is possible that A will send duplicate requests (i.e., resend a request message that has already been received by B) due to premature retransmissions. To be able to detect duplicate request messages, the protocol will use

sequence numbers. A 1-bit sequence number will suffice for a stop-and-wait type of request/response protocol.
A (the requestor) has 4 states:

- **"Wait for Request 0 from above."** Here the requestor is waiting for a call from above to request a unit of data. When it receives a request from above, it sends a request message, R0, to B, starts a timer and makes a transition to the "Wait for D0" state.

- **"Wait for D0"**. Here the requestor is waiting for a D0 data message from B. A timer is always running in this state. If the timer expires, A sends another R0 message, restarts the timer and remains in this state. If a D0 message is received from B, A stops the timer and transits to the "Wait for Request 1 from above" state (after delivering the data to the layer above).

- **"Wait for Request 1 from above."** Here the requestor is again waiting for a call from above to request a unit of data. When it receives a request from above, it sends a request message, R1, to B, starts a timer and makes a transition to the "Wait for D1" state.

- **"Wait for D1".** Here the requestor is waiting for a D1 data message from B. A timer is always running in this state. If the timer expires, A sends another R1 message, restarts the timer and remains in this state. If a D1 message is received from B, A stops the timer and transits to the "Wait for Request 0 from above" state (after delivering the data to the layer above).

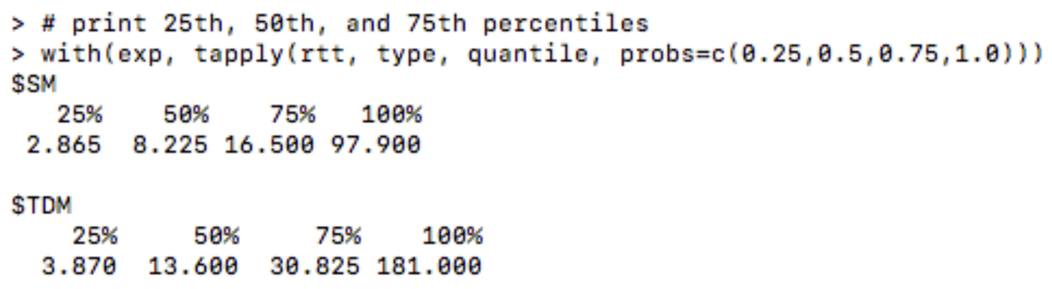
The data supplier (B) has only two states:

- **"Wait for R0."** In this state, B ignores any received R1 messages and stays in this state. If B receives an R0 message, then it sends its D0 message, which will be received correctly over the B-to-A channel, and transits to the "Wait for R1" state.

- **"Wait R1."** In this state, B ignores any received R0 messages and stays in this state. If B receives an R1 message, then it sends its next D1 message, which will be received correctly over the B-to-A channel, and transits to the "Wait for R0" state.

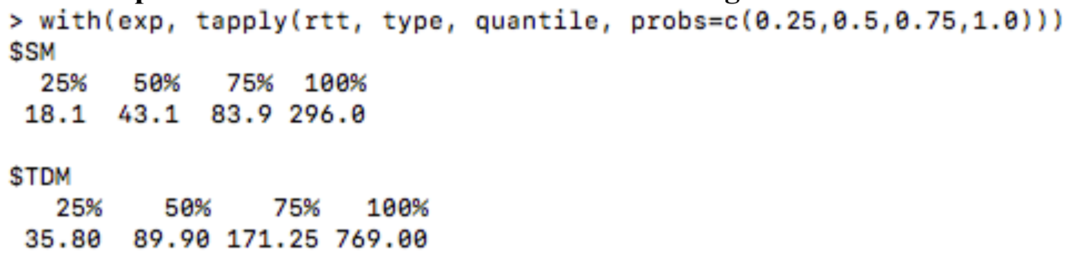
9. **(50 points)**

M/M/1 Experimental results should look like following:



```
> # print 25th, 50th, and 75th percentiles
> with(exp, tapply(rtt, type, quantile, probs=c(0.25,0.5,0.75,1.0)))
$SM
   25%    50%    75%   100%
 2.865  8.225 16.500 97.900

$TDM
    25%     50%     75%    100%
  3.870  13.600  30.825 181.000
```

M/D/1 Experimental results should look like following:

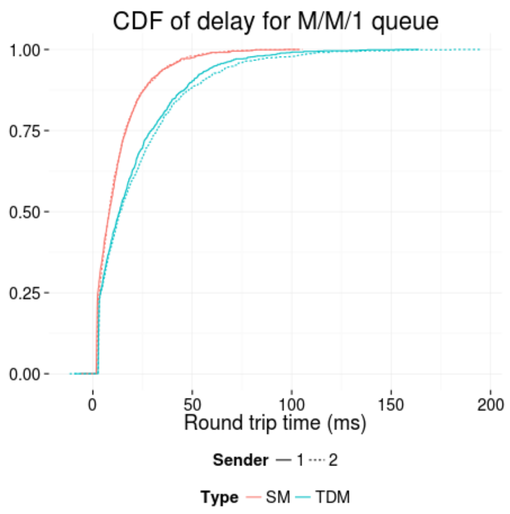


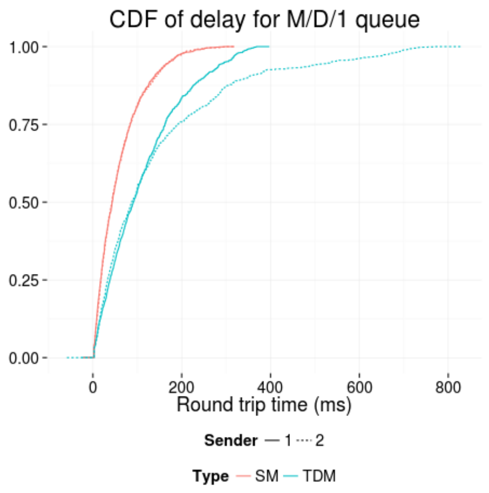
```
> with(exp, tapply(rtt, type, quantile, probs=c(0.25,0.5,0.75,1.0)))
$SM
  25%   50%   75%  100%
 18.1  43.1  83.9 296.0

$TDM
   25%    50%    75%   100%
 35.80  89.90 171.25 769.00
```

Observations:

1. Experimental results, under packet rate of 110 and average packet size of 512, show that M/D/1 is performing worse than M/M/1. However, analytically, M/D/1 should perform better than M/M/1 due to less variability in packet transmission times. Substituting into the analytical delay equations, we have:
    a. For SM, average packet delay under M/D/1 = 22.5ms, whereas under M/M/1, the delay = 41.4ms.
    b. For TDM, average packet delay under M/D/1 = 45ms, whereas under M/M/1, the delay = 82.8ms.
2. TDM performs almost twice as bad as SM for both M/M/1 and M/D/1.
3. Experimental results don't align very well with the analytical model [this is because the network drops some of the very large packets generated using the exponential distribution for M/M/1 due to MTU (max packet size). There are also other network delays that are not taken into account in the analytical models. The results SHOULD align better under packet rate of 220 and average packet size of 256 when the probability of very large packets is smaller!]

[Optional] CDF:


CDF of delay for M/M/1 queue
1.00
0.75
0.50
0.25
0.00
0
50
100
150
200
Round trip time (ms)
Sender — 1 ···· 2
Type — SM — TDM


CDF of delay for M/D/1 queue
1.00
0.75
0.50
0.25
0.00
0
200
400
600
800
Round trip time (ms)
Sender — 1 ···· 2
Type — SM — TDM