Nathan Fox

CS 460

February 19, 2015

Transport Layer Homework

1)

a.

01011100

01100101

--------

11000001 => 00111110

b.

11011010

01100101

--------

00111111 => 11000000

c.

If you flip the left-most bit of each byte, the 1's complement of the sum is unchanged:

11011100

11100101

--------

11000001 => 00111110

2)

In the case outlined below, the first time the Receiver sends ACK 0 (A0), it takes too long and the Sender times out. The Sender resends D0 and receives A0. The Sender sends D1 and receives A1. The Sender sends D0 and receives the first A0 that the Receiver sent, and the Sender thinks that the Receiver is acknowledging the most recent D0, when in reality the most recent D0 was dropped. The Sender sends D1 and the Receiver never receives the D0 that was dropped.

S R

Send D0 |D0------- |

| | \ |

Timeout | ------->|Receive D0

| | -A0|Send A0

Send D0 |D0------- / |

| \ / |

| --/---->|Receive D0

| --/----A0|Send A0

| / / |

Recv A0 |<------ / |

Send D1 |D1------/-- |

| / \ |

| / ----->|Receive D1

| / ------A1|Send A1

| / / |

Recv A1 |<--/----- |

Send D0 |--/---- |

| / \ |

Recv A0 |< ---(x) |

Send D1 |D1--- D0 Dropped|

| \ |

| ----------->|Receive D1

| |

3)

When the Sender sends data infrequently, I would prefer to use a NAK-only protocol over an ACK-only protocol. If I send just small amounts of data infrequently, I don't need everything I send to be acknowledged quickly, because I don't

have any more data to send once it gets acknowledged. I would want to just send the small amount of data I have, set a timer, and assume that the Receiver got it if I don't receive a NAK within the timer. If I do receive a NAK, I would

just resend the data and restart the timer.

When the Sender has a lot of data to send, I would prefer to use an ACK-only protocol. With a NAK-only protocol, I would have to send all the data in my window, set a timer, and I would have to wait for the timer to go off without receiving a NAK in order to slide my window and send more data. I don't want to have to wait for the timer to go off; I would rather wait to receive an ACK and then slide the window and send more data. Because the connection experiences few losses, I would only occasionally have to wait for timers when I don't receive the ACK I expect to in time. Even this could be fixed if I used fast retransmit.

4)

a. True. The Receiver may re-acknowledge packets below the window base.

b. True. The Receiver may re-acknowledge the packet just below the window base. This happens in the case where data is sent and acknowledged, the window slides, new data is sent but not received in order, so the cumulative ACK that comes back is for the packet just below the window base.

c. False. SR uses increasing sequence numbers, while the alternating-bit protocol only uses a single bit for the sequence number

c. False. GBN uses increasing sequence numbers, while the alternating-bit protocol only uses a single bit for the sequence number

5)

a. TCP slow start (1-6) (23-26)

b. TCP congestion avoidance (6-16) (16-22)

c. 3 duplicate ACKs, because it decreases by 1/2

d. Timeout event, because the congestion window resets to 1 MSS

e. Round 1, ssthresh = 32

f. Round 18, ssthresh = 21

g. Round 24, ssthresh = 14

h. 70th segment sent during Round 7

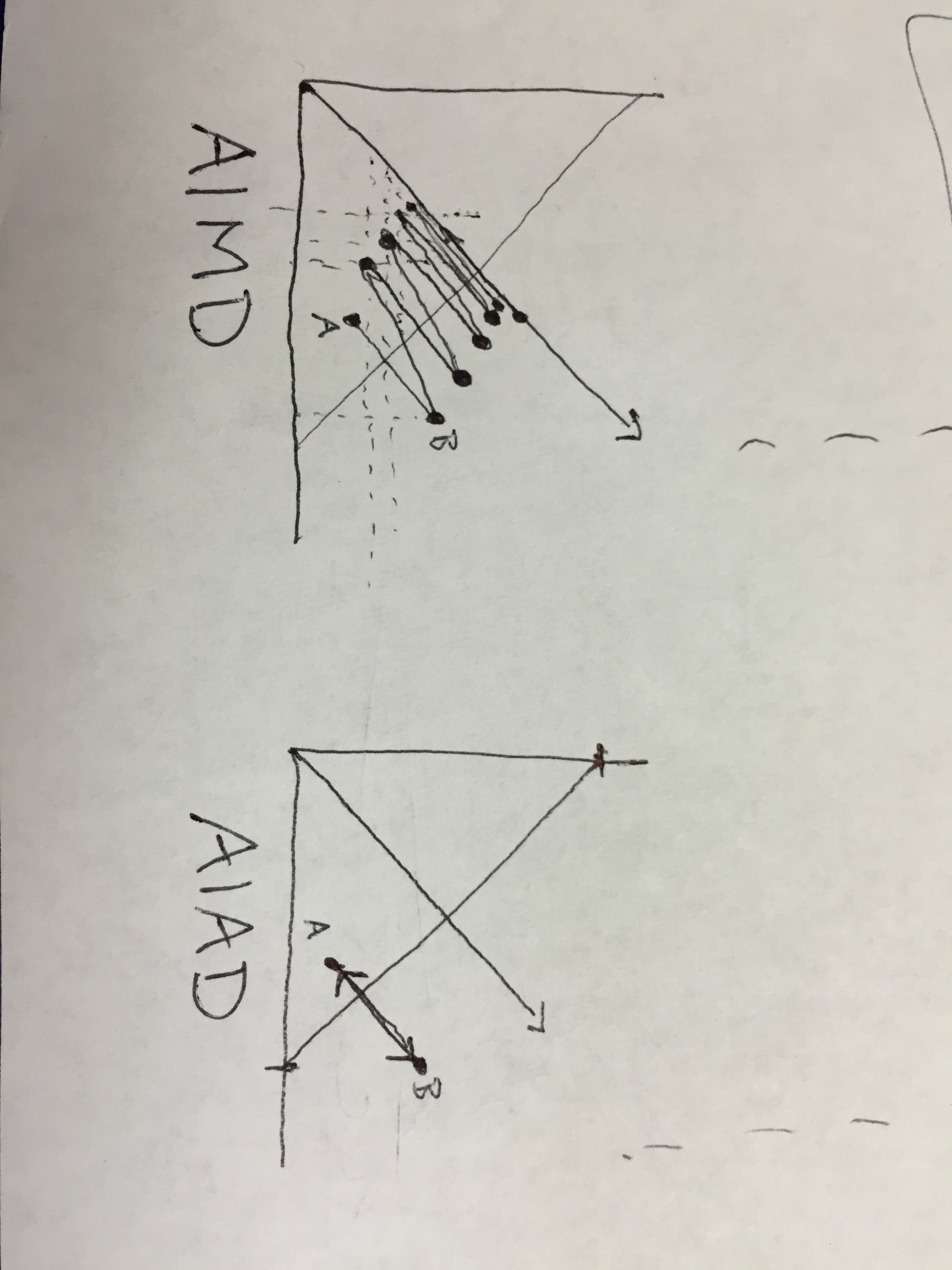
i. ssthresh = 4, cwnd = 7

j. ssthresh = 21, cwnd = 4

k. 1 + 2 + 4 + 8 + 16 + 1 = 32

6)

An AIAD algorithm would not converge to an equal share. The AIMD algorithm converges to an equal share because when the utilization is too much for the bandwidth, both throughputs are cut in half (multiplicative decrease), which leads to a line that slopes towards the equal bandwidth share line, regardless of where they are in the two-dimensional space. An AIAD algorithm would decrease the throughputs linearly, which leads to a line with a slope of 1 (a 45-degree line). The linear increase would retrace that 45-degree line, and the linear decrease would retrace it back. It would oscillate back and forth on the same 45-degree line forever.



7)

The ssthresh and cwnd values from time t1 represent how TCP had to respond to the network state as it was sending data up to time t1. At time t2, the network state might be totally different, so using the old ssthresh and cwnd values from time t1 doesn't really make sense. Maybe the network had only a few users at time t1, so ssthresh and cwnd were set pretty high. If the network is a lot more congested at time t2, using the old ssthresh and cwnd could result in a lot of packet loss. It would only be advantageous to use the t1 values if the network hadn't changed much between t1 and t2. As an alternative, I would suggest resetting ssthresh and cwnd to their initial defaults at time t2, and then as data is sent those values can adjust to the current network state.