2)  
(a) This is 125MB/sec; the sequence numbers wrap around when we send $2^{32} \times 4 = 4GB$. This would take $4GB/(125MB/sec) = 32$ seconds.
(b) Incrementing every 32 ms, it would take $32 \times 4 \times 10^9 ms$, or about four years, for the timestamp field to wrap.

3)  
Here is the table of the updates to the EstRTT, etc statistics. Packet loss is ignored; the SampleRTTs given may be assumed to be from successive singly transmitted segments. Note that the first column, therefore, is simply a row number, not a packet number, as packets are sent without updating the statistics when the measurements are ambiguous. Note also that both algorithms calculate the same values for EstimatedRTT; only the TimeOut calculations vary.

<table>
<thead>
<tr>
<th>SampleRTT</th>
<th>EstRTT</th>
<th>Dev</th>
<th>diff</th>
<th>new TimeOut</th>
<th>old TimeOut</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EstRTT+4×Dev</td>
<td></td>
<td></td>
<td>2×EstRTT</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.00</td>
<td>1.00</td>
<td>0.10</td>
<td>1.40</td>
<td>2.00</td>
</tr>
<tr>
<td>2</td>
<td>5.00</td>
<td>1.50</td>
<td>0.59</td>
<td>4.00</td>
<td>3.85</td>
</tr>
<tr>
<td>3</td>
<td>5.00</td>
<td>1.94</td>
<td>0.95</td>
<td>3.50</td>
<td>5.74</td>
</tr>
<tr>
<td>4</td>
<td>5.00</td>
<td>2.32</td>
<td>1.22</td>
<td>3.06</td>
<td>7.18</td>
</tr>
<tr>
<td></td>
<td>4.00</td>
<td>1.40</td>
<td>2.68</td>
<td>8.25</td>
<td>5.32</td>
</tr>
</tbody>
</table>

**New algorithm** (TimeOut = EstimatedRTT+ 4×Deviation):
There are a total of three retransmissions, two for packet 1 and one for packet 3. The first packet after the change times out at T=1.40, the value of TimeOut at that moment. It is retransmitted, with TimeOut backed off to 2.8. It times out again 4.2 sec after the first transmission, and TimeOut is backed off to 5.6. At T=5.0 the first ACK arrives and the second packet is sent, using the backed-off TimeOut value of 5.6. This second packet does not time out, so this constitutes an unambiguous RTT measurement, and so timing statistics are updated to those of row 1 above. When the third packet is sent, with TimeOut=3.85, it times out and is retransmitted. When its ACK arrives the fourth packet is sent, with the backed-off TimeOut value, $2 \times 3.85 = 7.70$; the resulting RTT measurement is unambiguous so timing statistics are updated to row 2. When the fifth packet is sent, TimeOut=5.74 and no further timeouts occur.

If we continue the above table to row 9, we get the maximum value for TimeOut, of 10.1, at which point TimeOut decreases toward 5.0.

**Original algorithm** (TimeOut = 2×EstimatedRTT):
There are five retransmissions: for packets 1, 2, 4, 6, 8. The first packet times out at T=2.0, and is retransmitted. The ACK arrives before the second timeout, which would have been at T=6.0.

When the second packet is sent, the backed-off TimeOut of 4.0 is used and we time out again. TimeOut is now backed off to 8.0. When the third packet is sent, it thus does not time out; statistics are updated to those of row 1.

The fourth packet is sent with TimeOut=3.0. We time out once, and then transmit the fifth packet without timeout. Statistics are then updated to row 2.

This pattern continues. The sixth packet is sent with TimeOut = 3.88; we again time out once, send the seventh packet without loss, and update to row 3.

The eighth packet is sent with TimeOut=4.64; we time out, back off, send packet 9, and update to row 4. Finally the tenth packet does not time out, as TimeOut=2×2.66=5.32 is larger than 5.0.

TimeOut continues to increase monotonically towards 10.0, as EstimatedRTT converges on 5.0.

4)  
(a) In slow start, the size of the window doubles every RTT. At the end of the $i$th RTT, the window size is $2^i$ KB. It will take 10 RTTs before the send window has reached $2^{10}$ KB = 1MB.
(b) After 10 RTTs, $1023\text{KB} = 1\text{MB} - 1\text{KB}$ has been transferred, and the window size is now $1\text{MB}$. Since we have not yet reached the maximum capacity of the network, slow start continues to double the window each RTT, so it takes 4 more RTTs to transfer the remaining $9\text{MB}$ (the amounts transferred during each of these last 4 RTTs are $1\text{MB}$, $2\text{MB}$, $4\text{MB}$, $1\text{MB}$; these are all well below the maximum capacity of the link in one RTT of $12.5\text{MB}$). Therefore, the file is transferred in 14 RTTs.

(c) It takes 1.4 seconds (14 RTTs) to send the file. The effective throughput is $(10\text{MB} / 1.4\text{s}) = 7.1\text{MBps} = 57.1\text{Mbps}$. This is only 5.7% of the available link bandwidth.

5)

Let the sender window size be 1 packet initially. The sender sends an entire window-full in one batch; for every ACK of such a window-full that the sender receives, it increases its effective window (which is counted in packets) by one. When there is a timeout, the effective window is cut into half the number of packets.

Now consider the situation when the indicated packets are lost. The window size is initially 1; when we get the first ACK it increases to 2. At the beginning of the second RTT we send packets 2 and 3. When we get their ACKs we increase the window size to 3 and send packets 4, 5 and 6. When these ACKs arrive the window size becomes 4.

Now, at the beginning of the fourth RTT, we send packets 7, 8, 9, and 10; by hypothesis packet 9 is lost. So, at the end of the fourth RTT we have a timeout and the window size is reduced to $4/2 = 2$.

Continuing, we have

<table>
<thead>
<tr>
<th>RTT</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent</td>
<td>9-10</td>
<td>11-13</td>
<td>14-17</td>
<td>18-22</td>
<td>23-28</td>
</tr>
</tbody>
</table>

Again the congestion window increases up until packet 25 is lost, when it is halved, to 3, at the end of the ninth RTT. The plot below shows the window size vs. RTT.
1)  
  a)  
  Time | Event                     |
  0    | ‘a’ is sent               |
  1    | ‘b’ is copied in the ‘send’ buffer |
  2    | ACK for a received, ‘b’ sent |
  3    | ‘c’ is copied in the ‘send’ buffer |
  4    | ACK for b is received, ‘c’ sent |
  .    |                           |
  .    |                           |
  .    |                           |
  .    |                           |
  16   | ACK for ‘h’ received, I sent |
  17   |                           |

b) Without the Nagel’s algorithm, the user will get small number of changes sent him. The user will see that the pointer changes some position and waits till it receives next position changes. With the Nagel’s algorithm, due to the buffering the changes get accumulated and are sent in batches. This causes the mouse movement to be not as responsive and too fast at the same time.