1. Purpose
Design and implement reliable data transfer over unreliable channels.

2. Description (Part 1)
Your job is to write a C/C++ transport-layer service over UDP that can sustain non-trivial transfer rates (hundreds of megabits/sec) under low packet loss and avoid getting bogged down under heavy loss. Part 1 implements only the connection handshake (SYN) and termination (FIN).

2.1. Code (25 pts)
To allow you control over network loss/delay, the receiving server emulates two queues at its nearest router (in forward/reverse directions) to the specifications requested by the client. This includes the RTT, link speed, and average packet-loss rate. These parameters must be specified in the SYN in order to complete the connection setup. We discuss these next.

The homework must accept as input seven command-line parameters of the transfer – destination host (hostname or IP), a power-of-2 buffer size to be transmitted (in DWORDs), sender window (in packets), the round-trip propagation delay (in seconds), the probability of loss in each direction, and the speed of the bottleneck link (in Mbps). For example:

```
C:\> rdt.exe s8.irl.cs.tamus.edu 24 50000 0.2 0.00001 0.0001 100
```
requests communication with s8.irl.cs.tamus.edu, an input buffer of size $2^{24}$ DWORDs (i.e., $2^{26}$ bytes), $W = 50K$ packets in the sender window, 200-ms RTT, $10^{-5}$ loss rate in the forward direction, $10^{-4}$ loss in the return path, and 100-Mbps bottleneck-link speed. If the parameters are incorrect, the program should print usage information and quit. Note that the receiver running on s8.irl.cs.tamus.edu never intentionally reordered or corrupts packets. You thus do not need to deal with packet checksums and superfluous retransmissions arising from reordering.

To make sure the receiver detects packet loss and incorrect retransmission, it is important to initialize the buffer to values that cannot be easily interchanged without being detected. For this purpose, you have to allocate an array of DWORDs and then set each element to a unique value given below.

Your transport layer must consist of a class called SenderSocket that provides OS-like APIs Open, Send, and Close (highlighted in bold below):

```cpp
// SenderSocket.h
#define MAGIC_PORT 22345  // receiver listens on this port
#define MAX_PKT_SIZE (1500-28) // maximum UDP packet size accepted by receiver

class SenderSocket {  ...
```
#include "SenderSocket.h"

void main (void)
{
  // parse command-line parameters
  char *targetHost = ...;
  int power = atoi (...);    // command-line specified integer
  int senderWindow = atoi (...);    // command-line specified integer

  UINT64 dwordBufSize = (UINT64) 1 << power;
  DWORD *dwordBuf = new DWORD [dwordBufSize];  // user-requested buffer
  for (UINT64 i = 0; i < dwordBufSize; i++)  // required initialization
    dwordBuf [i] = i;

  SenderSocket ss;    // instance of your class
  if ((status = ss.Open (targetHost, MAGIC_PORT, senderWindow, ...)) != STATUS_OK)
    // error handling: print status and quit

  char *charBuf = (char*) dwordBuf;   // this buffer goes into socket
  UINT64 byteBufferSize = dwordBufSize << 2;  // convert to bytes
  UINT64 off = 0;     // current position in buffer
  while (off < byteBufferSize)
    {
      // decide the size of next chunk
      int bytes = min (byteBufferSize - off, MAX_PKT_SIZE - sizeof(SenderDataHeader));
      // send chunk into socket
      if ((status = ss.Send (charBuf + off, bytes)) != STATUS_OK)
        // error handling: print status and quit
        off += bytes;

    }

  if ((status = ss.Close ()) != STATUS_OK)
    // error handling: print status and quit
}

Successful operation follows this example:

Main: sender W = 50000, RTT 0.200 sec, loss 1e-005 / 0.0001, link 100 Mbps
Main: initializing DWORD array with 2^24 elements... done in 93 ms
[0.005] --> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.82
[0.207] <-- SYN-ACK 0 window 1; setting initial RTO to 0.606
Main: connected to s8.irl.cs.tamu.edu in 0.202 sec, pkt 1472 bytes
[0.209] --> FIN 0 (attempt 1 of 5, RTO 0.606)
[0.410] <-- FIN-ACK 0 window 0
Main: transfer finished in 0.002 sec

The first two lines, printed by main(), reiterate input parameters\(^1\). The next two come from ss.Open() as it sends a SYN and receives a SYN-ACK. Both rows start with the current time in seconds since the constructor of the SenderSocket class was called. This is followed by a tab and an arrow specifying the direction of the packet (i.e., \(\rightarrow\) outgoing and \(\leftarrow\) incoming). Each printout carries the corresponding sequence number (i.e., 0 in this case), with SYNs additionally reporting the attempt number, the retransmission timeout (RTO) before the packet was sent, and the destination IP. SYN-ACKs display the window size in the response and the new RTO based on the handshake round-trip time. This value is set to three times the RTT of the connection phase (i.e., 202 * 3 = 606 ms in the above trace).

\(^1\) To print fractional numbers in compact notation as in this example use \%g in printf. This will automatically switch the number between scientific (1e-005) and dotted (0.0001) formats depending on whichever is shorter.
Main() performs its own timing of \texttt{ss.Open()} and reports that to the screen, together with \texttt{MAX_PKT_SIZE} it plans to use. The next two lines come from \texttt{ss.Close()} as it send a FIN and receives a FIN-ACK. Finally, main() confirms a successful transfer and prints its duration. This is the time between returning from \texttt{ss.Open()} and calling \texttt{ss.Close()}. Using the trace above, the delay is 209 – 207 = 2 ms.

For higher packet loss, the SYN/SYN-ACK or FIN/FIN-ACK pairs may get lost. Here is an example that requests 50% loss in the forward direction and requires multiple retransmissions:

Main: \textit{sender W = 1, RTT 0.200 sec, loss 0.5 / 0.0001, link 100 Mbps}
Main: \textit{initializing DWORD array with 2^24 elements... done in 31 ms}
\[0.006\] -> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.82
\[1.012\] -> SYN 0 (attempt 2 of 3, RTO 2.000) to 128.194.135.82
\[3.024\] -> SYN 0 (attempt 3 of 3, RTO 4.000) to 128.194.135.82
\[3.225\] <= SYN-ACK 0 window 1; setting initial RTO to 0.603
Main: \textit{connected to s8.irl.cs.tamu.edu in 0.201 sec, pkt 1472 bytes}
\[3.227\] -> FIN 0 (attempt 1 of 5, RTO 0.603)
\[3.836\] -> FIN 0 (attempt 2 of 5, RTO 1.205)
\[5.052\] -> FIN 0 (attempt 3 of 5, RTO 2.410)
\[5.253\] <= FIN-ACK 0 window 0
Main: \textit{transfer finished in 0.001 sec}

Before the first RTT is measured, the RTO starts at its default value of 1 second. This gets doubled for each retransmission, which is known as the \textit{exponential timer backoff} in TCP. The rationale is to become more conservative during back-to-back loss of the same packet, which typically indicates severe states of network congestion. The FIN phase is similar, except it starts with the last known RTO. Set the maximum number of attempts to 3 for SYN packets and 5 for all others. After exceeding these thresholds, you should abort the connection.

The full list of errors that \texttt{SocketSender} functions should be able to produce is given by:

```c
// possible status codes from ss.Open, ss.Send, ss.Close
#define STATUS_OK  0 // no error
#define ALREADY_CONNECTED 1 // second call to ss.Open() without closing connection
#define NOT_CONNECTED  2 // call to ss.Send()/Close() without ss.Open()
#define INVALID_NAME  3 // ss.Open() with targetHost that has no DNS entry
#define FAILED_SEND  4 // sendto() failed in kernel
#define FAILED_RECV  6 // recvfrom() failed in kernel
```

Examples (100xx errors come from WSAGetLastError):

Main: \textit{sender W = 1, RTT 0.200 sec, loss 0.9 / 0.0001, link 100 Mbps}
Main: \textit{initializing DWORD array with 2^24 elements... done in 31 ms}
\[0.006\] -> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.82
\[1.020\] -> SYN 0 (attempt 2 of 3, RTO 2.000) to 128.194.135.82
\[3.032\] -> SYN 0 (attempt 3 of 3, RTO 4.000) to 128.194.135.82
Main: \textit{connect failed with status 5}

Main: \textit{sender W = 1, RTT 0.200 sec, loss 0.1 / 0.0001, link 100 Mbps}
Main: \textit{initializing DWORD array with 2^24 elements... done in 78 ms}
\[0.001\] -> SYN 0 (attempt 1 of 3, RTO 1.000) to 0.0.0.0
\[0.001\] -> failed sendto with 10049
Main: \textit{connect failed with status 4}

Main: \textit{sender W = 1, RTT 0.200 sec, loss 0.9 / 0.0001, link 100 Mbps}
Main: \textit{initializing DWORD array with 2^24 elements... done in 31 ms}
\[0.001\] -> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.1
\[0.003\] <= failed recvfrom with 10054
Main: \textit{connect failed with status 6}
2.2. Packet Headers

Make sure to #pragma pack all network structs to 1 byte. All packets begin with a Flags header:

```c
#define MAGIC_PROTOCOL 0x8311AA

class Flags {
public:
    DWORD reserved:5; // must be zero
    DWORD SYN:1;
    DWORD ACK:1;
    DWORD FIN:1;
    DWORD magic:24;

    Flags () { memset(this, 0, sizeof(*this)); magic = MAGIC_PROTOCOL; }
};
```

The reserved field must be zero, SYN/ACK/FIN are the same as in TCP, and the magic protocol number must be 0x8311AA as set in the constructor. The receiver runs on Windows and thus requires no conversion of fields to network byte order.

The header for outgoing data and FIN packets consists of the flags and the packet sequence number:

```c
class SenderDataHeader {
public:
    Flags flags;
    DWORD seq; // must begin from 0
};
```

Connection setup is performed by exchanging a pair of packets – SYN from the sender and SYN-ACK from the receiver. The format of SYN packets is:

```c
#define FORWARD_PATH  0
#define RETURN_PATH  1

class LinkProperties {
public:
    // transfer parameters
    float RTT; // propagation RTT (in sec)
    float speed; // bottleneck bandwidth (in bits/sec)
    float pLoss [2]; // probability of loss in each direction
    DWORD bufferSize; // buffer size of emulated routers (in packets)

    LinkProperties () { memset(this, 0, sizeof(*this)); }
};

class SenderSynHeader {
public:
    SenderDataHeader sdh;
    LinkProperties lp;
};
```

All response packets have the same structure and just consist of the header:

```c
class ReceiverHeader {
public:
    Flags flags;
};
```
Since your SenderSocket class needs the RTT, speed, and loss from main(), it makes sense to just pass a pointer to a LinkProperties structure to ss.Open:

```c
LinkProperties lp;
lp.RTT = atof (...);
lp.speed = le6 * atof (...);       // convert to megabits
lp.pLoss [FORWARD_PATH] = atof (...);
lp.pLoss [RETURN_PATH] = atof (...);
if ((status = ss.Open (targetHost, MAGIC_PORT, senderWindow, &lp)) != STATUS_OK)
```

Note that lp.bufferSize is not specified by the user and is the only parameter that ss.Open() needs to determine automatically based on the other variables of the system. The goal is to size the router buffer to never lose packets. If the window is \( W \) and the maximum number of retransmissions is \( R \), you should never have more than \( W + R \) packets in flight. Thus, you can safely set lp.bufferSize to this value.

### 2.3. Invalid Conditions

Note that incorrect flags and/or combinations of parameters will be rejected by the server silently. The following checks are performed by the receiver: a) link speed must be larger than zero and no more than 10 Gbps; b) the RTT must be non-negative and smaller than 30 seconds; c) both probabilities of loss must be in \([0, 1)\); and c) router buffer size must be between 1 and 1M packets. Additional reasons that cause packets to be discarded – FIN or data packets are received before a connection has been set up; incorrect magic protocol in the flags header or the reserved field is not zero; packet size is smaller than `sizeof(SenderDataHeader)`; FIN packets arriving before completion of the transfer (i.e., the receiver window has gaps); SYN packets with size smaller than `sizeof(SenderSynHeader)`; data packets with invalid sequence numbers; and bogus flag combinations (such as SYN + FIN).

### 2.4. UDP

Socket operation is identical to that in Homework 2 – open a UDP socket, bind it to port 0, and then utilize `sendto/recvfrom`. The receiver dispatches only compliant packets, which allows you to receive them directly into the corresponding struct:

```c
ReceiverHeader rh;
if (recvfrom(sock, &rh, sizeof(ReceiverHeader), ...) == SOCKET_ERROR)
    // report GetLastError() and return control to main()
```

### 2.5. Standalone Receiver

You can download the receiver from C:\463 on ts2 and run it locally for debugging purposes (make sure to set your destination to 127.0.0.1 instead of s8.irl.cs.tamu.edu). In Part 3, depending on your CPU, you may be able to achieve rates in excess of 2.5 Gbps on localhost. There is also a dummy receiver on ts2 that omits checksums, bypasses router emulation, and allows (9000-28)-byte packets. Using an appropriately overclocked i7, you could get 10.5 Gbps with this version.
3. Description (Part 2)

This part implements rdt 3.0 with dynamic RTOs, closely following the algorithm in the slides. The only exception is that your sequence numbers are 4 bytes rather than 1 bit. To verify the transfer is good, you need to compute a CRC-32 checksum across the sent buffer and compare that value to the one provided by the server.

3.1. Code (25 pts)

The seven input parameters are exactly the same as before. You should suppress per-packet printouts (unless debugging) and add output from a stats thread every 2 seconds:

```
Main:   sender W = 1, RTT 0.100 sec, loss 0 / 0, link 14 Mbps
Main:   initializing DWORD array with 2^15 elements... done in 1 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.102 sec, pkt 1472 bytes
[ 2.01] B  18 (  0.0 MB) N  19 T 0 F 0 W 1 S 0.105 Mbps RTT 0.102
[ 4.02] B  36 (  0.1 MB) N  37 T 0 F 0 W 1 S 0.105 Mbps RTT 0.102
[ 6.04] B  55 (  0.1 MB) N  56 T 0 F 0 W 1 S 0.111 Mbps RTT 0.102
[ 8.05] B  73 (  0.1 MB) N  74 T 0 F 0 W 1 S 0.105 Mbps RTT 0.102
[10.03] <-- FIN-ACK 90 window FC694CF3
Main:   transfer finished in 9.818 sec, 106.80 Kbps, checksum FC694CF3
Main:   estRTT 0.102, ideal rate 114.41 Kbps
```

The first stats line (in bold) shows that 2.01 seconds have elapsed since the SenderSocket constructor was called, the sender base is currently at 18 packets, 0.0 MB of data has been ACKed by the receiver, the next sequence number is 19, there have been 0 packets with timeouts and 0 with fast retransmission, the current effective window size is 1 (i.e., the minimum between sender and receiver window), the speed at which the application consumes data at the receiver (i.e., goodput) is 0.105 Mbps, and the estimated RTT is 102 ms. Note that the speed is averaged over the period since the last printout (i.e., by subtracting the base values, then multiplying the result by 8 * (MAX_PKT_SIZE - sizeof(SenderDataHeader)).

Upon receipt of a FIN-ACK, ss.Close() prints the FIN’s sequence number 90, which is the total number of data packets delivered in this exchange, and the CRC-32 checksum of the data provided by the receiver in the recvWnd header of the FIN-ACK packet (in all other cases, recvWnd is valid and should be used in flow control). The elapsed time reported by main is slightly different from Part 1 – it is the duration between the transmission of the first data packet and receipt of the last data ACK (i.e., both SYN and FIN phases are excluded). Since the application may call ss.Close() before the last packet has been acknowledged, this function must pause to collect all outstanding data ACKs (and let the sender perform retransmission as needed) before moving ahead with the FIN. It can then return the elapsed time to main():

```c
    double elapsedTime;
    if ((status = ss.Close (&elapsedTime)) != STATUS_OK)
```

The final line of the trace shows that the latest value of estimated RTT is 102 ms (which is updated using TCP formulas) and the ideal rate under no loss is 114 Kbps (i.e., window / estRTT).

Another example with moderate packet loss:

```
Main:   sender W = 1, RTT 0.100 sec, loss 0.2 / 0, link 14 Mbps
Main:   initializing DWORD array with 2^15 elements... done in 0 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.103 sec, pkt 1472 bytes
```
3.2. ACK Numbers

Note that ACKs are similar to those in TCP – they acknowledge the base of the receiver window, i.e., the next expected packet. Unlike TCP, however, SYN-ACK and FIN-ACK packets do not increment the sequence number. A valid exchange will proceed as following:

--> SYN 0
<<- SYN-ACK 0       // expects packet 0
--> data 0
<<- data 1         // expects packet 1
--> ACK 1
<<- ACK 2          // expects packet 2
--> FIN 2
<<- FIN-ACK 2 window = CRC32      // still expects seq 2

Use the following function to compute checksums on the buffer you’ve transmitted:

```cpp
// checksum.cpp
Checksum::Checksum ()
{
    // set up a lookup table for later use
    for (DWORD i = 0; i < 256; i++)
    {
        DWORD c = i;
        for (int j = 0; j < 8; j++)
        {
            c = (c & 1) ? (0xEDB88320 ^ (c >> 1)) : (c >> 1);
        }
        crc_table[i] = c;
    }
}
DWORD Checksum::CRC32 (unsigned char *buf, size_t len)
{
    DWORD c = 0xFFFFFFFF;
    for (size_t i = 0; i < len; i++)
    {
        c = crc_table [(c ^ buf[i]) & 0xFF] ^ (c >> 8);
    }
    return c ^ 0xFFFFFFFF;
}
// main.cpp
void main (void)
{
    // send the buffer, close the connection
    Checksum cs;
    DWORD check = cs.CRC32 (charBuf, byteBufferSize);
}
```

3.3. RTO

Set the RTO for the SYN packet to the maximum of 1 second and 2*lp.RTT. For RTO estimation during the rest of the transfer, prevent the RTT deviation from falling below 10 ms:

\[
RTO = estRTT + 4 * \max (devRTT, 0.010);
\]
This should keep the RTO at least 40 ms above the mean; otherwise, you are likely to end up with 
\( \text{devRTT} = 0 \) and \( \text{RTO} = \text{estRTT} \), which will lead to frequent spurious timeouts.

Every time you retransmit the same packet, double its RTO (i.e., use exponential timer backoff).
Once that packet is recovered, restore the previous RTO before this packet got lost.

**4. Description (Part 3)**

The final protocol includes several features of TCP – cumulative ACKs with pipelining, fast retransmit, flow control, and a single timer for the base of the window, coupled with an exponential timer backoff during loss. To deal with high-loss scenarios below, **set the maximum number of retx attempts for all packets to 50.**

**4.1. Code (25 pts)**

The input and output is the same as in Part 2, except now the window size can be above 1:

```
Main:   sender W = 5000, RTT 0.200 sec, loss 0.001 / 0.0001, link 1000 Mbps
Main:   initializing DWORD array with 2^25 elements... done in 141 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.201 sec, pkt 1472 bytes
  [2.01] B  256 { 0.4 MB} N 512 T 0 F 0 W 256 S 1.498 Mbps RTT 0.206
  [4.02] B  6688 { 9.9 MB} N 10118 T 1 F 5 W 5000 S 37.697 Mbps RTT 0.368
  [6.04] B 12590 {18.5 MB} N 17590 T 1 F 13 W 5000 S 34.479 Mbps RTT 0.368
  [8.05] B 20157 {29.7 MB} N 25157 T 1 F 21 W 5000 S 44.281 Mbps RTT 0.368
  [10.06] B 25917 {38.1 MB} N 28876 T 1 F 26 W 5000 S 33.707 Mbps RTT 0.368
  [12.07] B 31888 {46.9 MB} N 36888 T 1 F 34 W 5000 S 34.940 Mbps RTT 0.368
  [14.09] B 39427 {58.0 MB} N 44427 T 1 F 43 W 5000 S 44.117 Mbps RTT 0.368
  [16.10] B 44188 {65.0 MB} N 49188 T 1 F 48 W 5000 S 27.860 Mbps RTT 0.368
  [18.11] B 47897 {70.5 MB} N 52897 T 1 F 53 W 5000 S 21.705 Mbps RTT 0.368
  [20.12] B 55060 {81.0 MB} N 60060 T 1 F 60 W 5000 S 41.916 Mbps RTT 0.368
  [22.14] B 59711 {87.9 MB} N 64711 T 2 F 65 W 5000 S 27.217 Mbps RTT 0.368
  [24.15] B 63122 {92.9 MB} N 68122 T 2 F 70 W 5000 S 19.960 Mbps RTT 0.368
  [26.16] B 66526 {97.9 MB} N 71526 T 2 F 74 W 5000 S 19.920 Mbps RTT 0.368
  [28.17] B 78745 {115.9 MB} N 83745 T 2 F 80 W 5000 S 71.503 Mbps RTT 0.368
  [30.19] B 85496 {125.9 MB} N 90496 T 2 F 90 W 5000 S 39.506 Mbps RTT 0.368
  [31.40] <-- FIN-ACK 91679 window FC6FB7CB
Main:   transfer finished in 30.991 sec, 34646.57 Kbps, checksum FC6FB7CB
Main:   estRTT 0.368, ideal rate 159087.99 Kbps
```

**4.2. Report (25 pts)**

All transfers that determine speed must have sufficient user buffer size to reach steady-state dynamics (i.e., the rate becomes stable). Once the transfer speed has stabilized, record this value for the analysis below. Several questions to address:

1. Set packet loss \( p \) to zero in both directions, the RTT to 0.5 seconds, and bottleneck link speed to \( S = 1 \) Gbps. Examine how your goodput scales with window size \( W \). This should be done by plotting the steady-state rate \( r(W) \) for \( W = 1, 2, 4, 8, \ldots, 2^{10} \) and keeping the \( x \)-axis on a log-scale. Your peak rate will be 23.9 Mbps and, depending on your home bandwidth, usage of an on-campus server might be necessary. Using curve-fitting, generate a model for \( r(W) \). Discuss whether it matches the theory discussed in class.

2. Expanding on the previous question, fix the window size at \( W = 30 \) packets and vary the RTT = 10, 20, 40, \ldots, 5120 ms. Plot steady rate \( r(\text{RTT}) \), again placing the \( x \)-axis on a log-scale. Perform curve-fitting to determine a model that describes this relationship. Due to queuing/transmission delays emulated by the server and various OS kernel overhead, the
actual RTT may deviate from the requested RTT. Thus, use the measured average in your plots and comment on whether the resulting curve matches theory.

3. Run the dummy receiver on your localhost and produce a trace using $W = 8K$ (the other parameters do not matter as the dummy receiver ignores them, although they should still be within valid ranges). Discuss your CPU configuration and whether you managed to exceed 1 Gbps. How about 10 Gbps using large packets?

4. Use buffer size $2^{23}$ DWORDs, RTT = 200 ms, window size $W = 300$ packets, link capacity $S = 10$ Mbps, and loss only in the reverse direction equal to $p = 0.1$. Show an entire trace of execution for this scenario and compare it to a similar case with no loss in either direction. Does your protocol keep the same rate in these two cases? Why or why not?

5. Determine the algorithm that the receiver uses to change its advertised window. What name does this technique have in TCP? Hint: the receiver window does not grow to infinity, but it does get pretty large.

4.3. Program Structure

Your program will consist of three threads as show in Figure 1. When `ss.Open()` succeeds with the connection, it spawns the ACK and stats threads to run in the background and provide support for the remainder of the transfer. These threads can be signaled to quit in `ss.Close();` however, if the user deletes the socket class without calling `ss.Close()`, you may end up leaking memory or crashing. As a result, the destructor `~SenderSocket()` must check if these threads are still running and terminate them before deleting shared data objects inside the class (such as the queue of pending packets). The most common way of signaling termination is to set some shared event and then wait on both thread handles obtained from `CreateThread`. As a general rule, graceful termination of threads is the best method for avoiding unexpected crashes and various problems on exit.

![Figure 1. Organization of the program.](image)

Function `ss.Send()` interacts with the ACK thread through a shared queue of pending packets. This nicely maps to the bounded producer-consumer (PC) problem, where the fixed queue size is $W$ packets. The easiest way to implement this is to block `ss.Send()` on a semaphore that counts the number of empty slots in the buffer. For each received ACK that moves the base forward by $k$ packets, this semaphore gets released by the same $k$ slots. The buffer is a circular array of packets (each with `MAX_PKT_SIZE` bytes).
There are a few additional caveats. First, you need to ensure clean termination during timeouts. If
the ACK thread encounters the maximum number of timeouts on the same packet, it must un-
block ss.Send() and somehow notify it that the connection has failed. Second, when
ss.Close() is called, the function must block until the ACK thread has collected all outstanding
acknowledgments. Otherwise, the FIN packet may be rejected by the server and/or the transfer
may be incomplete. Third, upon receiving an ACK that moves the base from $x$ to $x + y$, an RTT
sample is computed only based on packet $x + y - 1$ and only if there were no prior retransmis-
sions among the packets with sequence numbers in [$x, x + y - 1$]. Finally, when moving the win-
dow forward, reset the timer to current time plus the most-recent RTO (retransmission timeout).

### 4.4. Winsock Issues

By default, the UDP sender/receiver buffer inside the Windows kernel is configured to support
only 8 KB of unprocessed data. You can achieve higher outbound performance and prevent
packet loss in the inbound direction by increasing both buffers, then setting your transport
threads to time-critical priority:

```c
int kernelBuffer = 2e6;   // 2 meg
if (setsockopt (sock, SOL_SOCKET, SO_RCVBUF, &kernelBuffer, sizeof (int)) == SOCKET_ERROR)
...
kernelBuffer = 20e6;   // 20 meg
if (setsockopt (sock, SOL_SOCKET, SO_SNDBUF, &kernelBuffer, sizeof (int)) == SOCKET_ERROR)
...
SetThreadPriority (GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
```

Additionally, for `sendto()` and `recvfrom()` to work concurrently from both threads, you must
first set the socket into non-blocking mode and then call `select()` before sending or receiving to
make sure each operation would complete immediately (otherwise, the call will fail with
`WSAEWOULDBLOCK`):

```c
u_long imode = 1;
if (ioctlsocket (sock, FIONBIO, &imode) == ...) // done once after creation
if (select (0, NULL, &fdWrite, ...) > 0) // before each sendto()
    sendto (...);
...
if (select (0, &fdRead, NULL, ...) > 0) // before each recvfrom()
    recvfrom (...);
```

### 4.5. Flow Control

The same semaphore between the send and ACK threads can be reused to easily accomplish flow
control using this general architecture:

```c
HANDLE s = CreateSemaphore (NULL, 0, W, NULL)
// after the SYN-ACK, inside ss.Open()
int lastReleased = min (W, synack->window);
ReleaseSemaphore (s, lastReleased);
// in the ACK thread
while (not end of transfer)
{
    get ACK with sequence y, receiver window R
    if (y > sndBase)
    {
        sndBase = y
        effectiveWin = min (W, ack->window)
```

---

10
To test that flow control works, set the RTT to 2 seconds and observe the effective window reported by your program. It should expand once per printout.

### 4.6. Small Window, No Loss

Main: sender W = 10, RTT 0.100 sec, loss 0 / 0, link 1000 Mbps
Main: initializing DWORD array with 2^20 elements... done in 0 ms
Main: connected to s8.irl.cs.tamu.edu in 0.103 sec, pkt 1472 bytes

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes Transferred</th>
<th>Window Size</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>146</td>
<td>156</td>
<td>0.859</td>
</tr>
<tr>
<td>4.01</td>
<td>326</td>
<td>336</td>
<td>1.053</td>
</tr>
<tr>
<td>6.03</td>
<td>516</td>
<td>526</td>
<td>1.112</td>
</tr>
<tr>
<td>8.04</td>
<td>696</td>
<td>706</td>
<td>1.053</td>
</tr>
<tr>
<td>10.05</td>
<td>886</td>
<td>896</td>
<td>1.112</td>
</tr>
<tr>
<td>12.06</td>
<td>1076</td>
<td>1076</td>
<td>1.053</td>
</tr>
<tr>
<td>14.08</td>
<td>1266</td>
<td>1256</td>
<td>1.053</td>
</tr>
<tr>
<td>16.09</td>
<td>1436</td>
<td>1446</td>
<td>1.112</td>
</tr>
<tr>
<td>18.10</td>
<td>1616</td>
<td>1626</td>
<td>1.053</td>
</tr>
<tr>
<td>20.11</td>
<td>1806</td>
<td>1816</td>
<td>1.112</td>
</tr>
<tr>
<td>22.12</td>
<td>1986</td>
<td>1996</td>
<td>1.053</td>
</tr>
<tr>
<td>24.14</td>
<td>2166</td>
<td>2186</td>
<td>1.112</td>
</tr>
<tr>
<td>26.15</td>
<td>2356</td>
<td>2366</td>
<td>1.053</td>
</tr>
<tr>
<td>28.16</td>
<td>2536</td>
<td>2546</td>
<td>1.053</td>
</tr>
<tr>
<td>30.17</td>
<td>2726</td>
<td>2736</td>
<td>1.112</td>
</tr>
</tbody>
</table>

Main: transfer finished in 31.665 sec, 1059.67 Kbps, checksum 5B0360D
Main: estRTT 0.102, ideal rate 1151.76 Kbps

### 4.7. Large Window, Low Loss

Main: sender W = 12000, RTT 0.100 sec, loss 0.0001 / 0, link 1000 Mbps
Main: initializing DWORD array with 2^28 elements... done in 626 ms
Main: connected to s8.irl.cs.tamu.edu in 0.101 sec, pkt 1472 bytes

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes Transferred</th>
<th>Window Size</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>14592</td>
<td>26592</td>
<td>85.952</td>
</tr>
<tr>
<td>4.00</td>
<td>91281</td>
<td>103281</td>
<td>451.527</td>
</tr>
<tr>
<td>6.00</td>
<td>166293</td>
<td>178293</td>
<td>441.653</td>
</tr>
<tr>
<td>8.00</td>
<td>254245</td>
<td>265999</td>
<td>517.841</td>
</tr>
<tr>
<td>10.00</td>
<td>347759</td>
<td>370815</td>
<td>749.568</td>
</tr>
<tr>
<td>12.00</td>
<td>427221</td>
<td>438733</td>
<td>538.513</td>
</tr>
<tr>
<td>14.00</td>
<td>507486</td>
<td>515736</td>
<td>472.582</td>
</tr>
<tr>
<td>16.00</td>
<td>588696</td>
<td>598259</td>
<td>478.146</td>
</tr>
<tr>
<td>18.00</td>
<td>655810</td>
<td>667810</td>
<td>395.005</td>
</tr>
<tr>
<td>20.00</td>
<td>727240</td>
<td>733431</td>
<td>420.563</td>
</tr>
</tbody>
</table>

Main: transfer finished in 20.140 sec, 426503.90 Kbps, checksum E8F5B708
Main: estRTT 0.102, ideal rate 1151.76 Kbps

### 4.8. Small Window, Moderate Loss

Main: sender W = 10, RTT 0.010 sec, loss 0.1 / 0, link 1000 Mbps
Main: initializing DWORD array with 2^20 elements... done in 0 ms
Main: connected to s8.irl.cs.tamu.edu in 0.013 sec, pkt 1472 bytes

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes Transferred</th>
<th>Window Size</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.01</td>
<td>429</td>
<td>439</td>
<td>2.510</td>
</tr>
<tr>
<td>4.02</td>
<td>871</td>
<td>881</td>
<td>2.587</td>
</tr>
<tr>
<td>6.04</td>
<td>1366</td>
<td>1376</td>
<td>2.897</td>
</tr>
<tr>
<td>8.05</td>
<td>1694</td>
<td>1704</td>
<td>1.861</td>
</tr>
<tr>
<td>10.06</td>
<td>2152</td>
<td>2162</td>
<td>2.739</td>
</tr>
<tr>
<td>12.07</td>
<td>2643</td>
<td>2653</td>
<td>2.873</td>
</tr>
</tbody>
</table>

Main: transfer finished in 13.396 sec, 2504.80 Kbps, checksum 5B0360D
Main: estRTT 0.011, ideal rate 10191.97 Kbps
### 4.9. Bottlenecked by Emulated Router

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes</th>
<th>Rate</th>
<th>Seq</th>
<th>Ack</th>
<th>Window</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>2275</td>
<td>13.376 Mbps</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0.100 sec</td>
</tr>
<tr>
<td>4.01</td>
<td>6584</td>
<td>25.223 Mbps</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0.108 sec</td>
</tr>
<tr>
<td>6.03</td>
<td>10937</td>
<td>25.471 Mbps</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>0.110 sec</td>
</tr>
<tr>
<td>8.04</td>
<td>16092</td>
<td>30.162 Mbps</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0.106 sec</td>
</tr>
<tr>
<td>10.05</td>
<td>20004</td>
<td>22.895 Mbps</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>0.115 sec</td>
</tr>
<tr>
<td>12.06</td>
<td>24833</td>
<td>28.248 Mbps</td>
<td>0</td>
<td>0</td>
<td>23</td>
<td>0.122 sec</td>
</tr>
<tr>
<td>14.08</td>
<td>29577</td>
<td>27.771 Mbps</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>0.103 sec</td>
</tr>
<tr>
<td>16.09</td>
<td>33571</td>
<td>23.372 Mbps</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>0.112 sec</td>
</tr>
<tr>
<td>18.10</td>
<td>38502</td>
<td>28.853 Mbps</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>0.103 sec</td>
</tr>
</tbody>
</table>

Main: transfer finished in 21.294 sec, 25212.74 Kbps, checksum 85A854D4

### 4.10. Surviving Heavy Loss

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes</th>
<th>Rate</th>
<th>Seq</th>
<th>Ack</th>
<th>Window</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.01</td>
<td>6</td>
<td>0.035 Mbps</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0.013 sec</td>
</tr>
<tr>
<td>4.02</td>
<td>6</td>
<td>0.000 Mbps</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0.013 sec</td>
</tr>
<tr>
<td>6.03</td>
<td>18</td>
<td>0.070 Mbps</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0.070 sec</td>
</tr>
<tr>
<td>8.05</td>
<td>41</td>
<td>0.135 Mbps</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0.070 sec</td>
</tr>
<tr>
<td>10.06</td>
<td>56</td>
<td>0.088 Mbps</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0.070 sec</td>
</tr>
<tr>
<td>12.07</td>
<td>84</td>
<td>0.164 Mbps</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0.070 sec</td>
</tr>
<tr>
<td>14.08</td>
<td>88</td>
<td>0.023 Mbps</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0.070 sec</td>
</tr>
</tbody>
</table>

Main: transfer finished in 14.141 sec, 74.15 Kbps, checksum FC694CF3

### 4.11. Saturating 1 Gbps

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes</th>
<th>Rate</th>
<th>Seq</th>
<th>Ack</th>
<th>Window</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>143553</td>
<td>845.572 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>4.00</td>
<td>30683</td>
<td>956.712 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>6.00</td>
<td>468775</td>
<td>957.736 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>8.00</td>
<td>631291</td>
<td>956.489 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>10.00</td>
<td>793714</td>
<td>956.310 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>12.00</td>
<td>1118910</td>
<td>957.322 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
<tr>
<td>14.00</td>
<td>1444185</td>
<td>957.473 Mbps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.032 sec</td>
</tr>
</tbody>
</table>

Main: transfer finished in 18.279 sec, 939892.48 Kbps, checksum EF5F9797

<table>
<thead>
<tr>
<th>Time</th>
<th>Bytes</th>
<th>Rate</th>
<th>Seq</th>
<th>Ack</th>
<th>Window</th>
<th>RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.00</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Main: transfer finished in 18.279 sec, 939892.48 Kbps, checksum EF5F9797
# CSCE 463/612 Homework 3 Part 1

Name: ______________________________

<table>
<thead>
<tr>
<th>Function</th>
<th>Points</th>
<th>Breakdown</th>
<th>Item</th>
<th>Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printouts</td>
<td>25</td>
<td>1</td>
<td>Incorrect summary (W, RTT, loss, speed)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect array initialization and delay</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Bad SYN (timer, seq, attempt, RTO, IP)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Bad SYN-ACK (seq, window, RTO)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Wrong main connect (host, delay, pkt size)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Bad FIN (seq, attempt, RTO)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Bad FIN-ACK (seq, window)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect final transfer delay</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Fails to retx on SYN timeout</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Fails to retx on FIN timeout</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Fails to print WSAGetLastError() on sendto</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Fails to print WSAGetLastError() on recvfrom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Main does not report status 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Main does not report status 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Main does not report status 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Main does not report status 6</td>
<td></td>
</tr>
</tbody>
</table>

Total points: ________________
CSCE 463/612 Homework 3 Part 2

Name: ______________________________

<table>
<thead>
<tr>
<th>Function</th>
<th>Points</th>
<th>Break down</th>
<th>Item</th>
<th>Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printouts</td>
<td>25</td>
<td>1</td>
<td>Fails to print the stat timer every 2 sec</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect base (B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect bytes sent</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect next sequence (N)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect timeouts (T)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect window (W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect speed (S)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect RTT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Incorrect FIN-ACK (seq, checksum)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>Incorrect summary (delay, rate, checksum)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect ideal rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Fails to perform as shown in the first example of section 3.1 (no loss)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Fails to perform as shown in the second example of section 3.1 (20% loss)</td>
<td></td>
</tr>
</tbody>
</table>

Total points: ________________
## CSCE 463/612 Homework 3 Part 3

Name: ______________________________

<table>
<thead>
<tr>
<th>Function</th>
<th>Points</th>
<th>Break down</th>
<th>Item</th>
<th>Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 4.6</td>
<td>5</td>
<td>1</td>
<td>Incorrect rate (not within 20%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect T (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect F (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect FIN-ACK sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Checksum does not match receiver</td>
<td></td>
</tr>
<tr>
<td>Test case 4.7</td>
<td>5</td>
<td>1</td>
<td>Incorrect rate (not within 20%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect T (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect F (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect FIN-ACK sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Checksum does not match receiver</td>
<td></td>
</tr>
<tr>
<td>Test case 4.8</td>
<td>5</td>
<td>1</td>
<td>Incorrect rate (not within 20%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect T (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect F (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect FIN-ACK sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Checksum does not match receiver</td>
<td></td>
</tr>
<tr>
<td>Test case 4.9</td>
<td>5</td>
<td>1</td>
<td>Incorrect rate (not within 20%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect T (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect F (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect FIN-ACK sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Checksum does not match receiver</td>
<td></td>
</tr>
<tr>
<td>Test case 4.10</td>
<td>5</td>
<td>1</td>
<td>Incorrect rate (not within 20%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect T (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect F (not within 10%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect FIN-ACK sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Checksum does not match receiver</td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td>25</td>
<td>5</td>
<td>Model of $r(W)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Model of $r(RTT)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Dummy receiver trace</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Reverse packet loss trace</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>Receiver window-resize algorithm</td>
<td></td>
</tr>
</tbody>
</table>

Total points: ________________