CSCE 463/612: Networks and Distributed Processing

Homework 3 (100 pts)

Due dates: 3/28/17 (part 1), 4/11/17 (part 2), 4/20/17 (part 3)

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.tamu.edu 24 50000 0.2 0.00001 0.0001 100
```

requests communication with s8.irl.cs.tamu.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.tamu.edu` never intentionally reorders 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):

```
// 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 {
        ...
```

```
};

// main.cpp
#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 handing: print status and quit
		off += bytes;
	}

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

Successful operation follows this example:

```
Main:   sender W = 10, RTT 0.200 sec, loss 1e-05 / 0.0001, link 100 Mbps
Main:   initializing DWORD array with 2^20 elements... done in 0 ms
[ 0.002] --> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.82
[ 0.223] <-- SYN-ACK 0 window 1; setting initial RTO to 0.663
Main:   connected to s8.irl.cs.tamu.edu in 0.221 sec, pkt size 1472 bytes
[ 0.233] --> FIN 0 (attempt 1 of 5, RTO 0.663)
[ 0.457] <-- FIN-ACK 0 window 0
Main:   transfer finished in 0.010 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., → outgoing and ← 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., 221 * 3 = 663 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 `ss.Open()` and reports that to the screen, together with `MAX_PKT_SIZE` it plans to use. The next two lines come from `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 `ss.Open()`and calling `ss.Close()`. Using the trace above, the delay is 233 – 223 = 10 ms.

For higher packet loss, the SYN/SYN-ACK or FIN/FIN-ACK pairs may get lost. Here is an example that requests 40% loss in the forward direction and requires multiple retransmissions:

```
Main:   sender W = 10, RTT 0.200 sec, loss 0.4 / 0.0001, link 100 Mbps
Main:   initializing DWORD array with 2^20 elements... done in 0 ms
[ 0.002] --> SYN 0 (attempt 1 of 3, RTO 1.000) to 128.194.135.82
[ 1.014] --> SYN 0 (attempt 2 of 3, RTO 1.000) to 128.194.135.82
[ 2.028] --> SYN 0 (attempt 3 of 3, RTO 1.000) to 128.194.135.82
[ 2.239] <-- SYN-ACK 0 window 1; setting initial RTO to 0.631
Main:   connected to s8.irl.cs.tamu.edu in 0.210 sec, pkt size 1472 bytes
[ 2.247] --> FIN 0 (attempt 1 of 5, RTO 0.631)
[ 2.886] --> FIN 0 (attempt 2 of 5, RTO 0.631)
[ 3.097] <-- FIN-ACK 0 window 0
Main:   transfer finished in 0.007 sec
```

Before the first RTT is measured, the RTO starts at its default value of 1 second. 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 `SocketSender` functions should be able to produce is given by:

```
// 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 TIMEOUT               5       // timeout after all retx attempts are exhausted
#define FAILED_RECV           6       // recvfrom() failed in kernel
```

Examples (100xx errors come from `WSAGetLastError`):

```
Main:   sender W = 1, RTT 0.200 sec, loss 0.9 / 0.0001, link 100 Mbps
Main:   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 1.000) to 128.194.135.82
[2.032] --> SYN 0 (attempt 3 of 3, RTO 1.000) to 128.194.135.82
Main:   connect failed with status 5

Main:   sender W = 1, RTT 0.200 sec, loss 0.1 / 0.0001, link 100 Mbps
Main:   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:   connect failed with status 4

Main:   sender W = 1, RTT 0.200 sec, loss 0.9 / 0.0001, link 100 Mbps
Main:   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:   connect failed with status 6

Main:   sender W = 8000, RTT 0.010 sec, loss 0 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^30 elements... done in 1076 ms
[0.002] --> target s88.irl.cs.tamu.edu is invalid
```

```
Main:   connect failed with status 3
```

2.2. Packet Headers

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

```
#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:

```
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:

```
#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:

```
class ReceiverHeader {
public:
        Flags                   flags;
        DWORD                   recvWnd;        // receiver window for flow control (in pkts)
        DWORD                   ackSeq;         // ack value = next expected sequence
};
```

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`:

```
LinkProperties lp;
lp.RTT = atof (...);
lp.speed = 1e6 * 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:

```
ReceiverHeader rh;
if ((bytes = 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. You could get over 10 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] B     18 (  0.0 MB) N     19 T 0 F 0 W 1 S 0.105 Mbps RTT 0.102
[ 4] B     36 (  0.1 MB) N     37 T 0 F 0 W 1 S 0.105 Mbps RTT 0.102
[ 6] B     55 (  0.1 MB) N     56 T 0 F 0 W 1 S 0.111 Mbps RTT 0.102
[ 8] 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 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():

```
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
[ 2] B      9 (  0.0 MB) N     10 T 4 F 0 W 1 S 0.053 Mbps RTT 0.102
[ 4] B     20 (  0.0 MB) N     21 T 9 F 0 W 1 S 0.064 Mbps RTT 0.102
[ 6] B     29 (  0.0 MB) N     30 T 14 F 0 W 1 S 0.053 Mbps RTT 0.102
[ 8] B     42 (  0.1 MB) N     43 T 18 F 0 W 1 S 0.076 Mbps RTT 0.102
[10] B     60 (  0.1 MB) N     61 T 18 F 0 W 1 S 0.105 Mbps RTT 0.102
[12] B     69 (  0.1 MB) N     70 T 21 F 0 W 1 S 0.053 Mbps RTT 0.102
```

```
[14] B     84 (  0.1 MB) N     85 T 23 F 0 W 1 S 0.088 Mbps RTT 0.102
[15.11] <-- FIN-ACK 90 window FC694CF3
Main:   transfer finished in 14.894 sec, 70.40 Kbps, checksum FC694CF3
Main:   estRTT 0.102, ideal rate 114.46 Kbps
```

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
<-- ACK 1                           // expects packet 1
--> data 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:

```
// 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 `devRTT = 0` and `RTO = estRTT`, which will lead to frequent spurious timeouts.

4. Description (Part 3)

The final protocol includes additional features of TCP – cumulative ACKs with pipelining, fast retransmit, and flow control (note that a single timer for the base of the window remains the same as in Part 2). 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 15 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.219 sec, pkt size 1472 bytes
[  2] B    184 (  0.3 MB) N    368 T 0 F 1 W 184 S 1.077 Mbps RTT 0.219
[  4] B   2821 (  4.2 MB) N   5642 T 1 F 5 W 2821 S 15.408 Mbps RTT 0.218
[  6] B  11492 ( 16.9 MB) N  16492 T 2 F 12 W 5000 S 50.741 Mbps RTT 0.218
[  8] B  18075 ( 26.6 MB) N  23075 T 5 F 18 W 5000 S 38.522 Mbps RTT 0.218
[ 10] B  24680 ( 36.3 MB) N  29680 T 5 F 23 W 5000 S 38.651 Mbps RTT 0.218
[ 12] B  30291 ( 44.6 MB) N  35291 T 7 F 28 W 5000 S 32.835 Mbps RTT 0.218
[ 14] B  33904 ( 49.9 MB) N  38904 T 7 F 32 W 5000 S 21.142 Mbps RTT 0.218
[ 16] B  37389 ( 55.0 MB) N  42389 T 8 F 37 W 5000 S 20.393 Mbps RTT 0.218
[ 18] B  48062 ( 70.7 MB) N  51849 T 9 F 42 W 5000 S 62.459 Mbps RTT 0.248
[ 20] B  52999 ( 78.0 MB) N  57999 T 11 F 50 W 5000 S 28.890 Mbps RTT 0.248
[ 22] B  63145 ( 92.9 MB) N  68145 T 13 F 58 W 5000 S 59.371 Mbps RTT 0.216
[ 24] B  75527 (111.2 MB) N  80206 T 14 F 67 W 5000 S 72.458 Mbps RTT 0.226
[ 26] B  81802 (120.4 MB) N  86802 T 14 F 76 W 5000 S 36.683 Mbps RTT 0.226
[ 28] B  90697 (133.5 MB) N  91679 T 16 F 82 W 5000 S 52.051 Mbps RTT 0.226
[29.065] <-- FIN-ACK 91679 window FC6FB7CB
Main:   transfer finished in 28.626 sec, 37509.93 Kbps, checksum FC6FB7CB
Main:   estRTT 0.226, ideal rate 258854.99 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, …, 2^{10} and keeping the x-axis on a log-scale. Your peak rate will be around 24 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, …, 5120 ms. Plot stable rate r(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 9-KB packets (see above)?

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 and you need provide its upper bound as part of the answer.

6. (extra credit, 10 pts): achieve performance similar to that in section 4.11 using ts.cs.tamu.edu or ts2.cs.tamu.edu.

4.3. Program Structure

It is recommended to structure your program around the three threads 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.

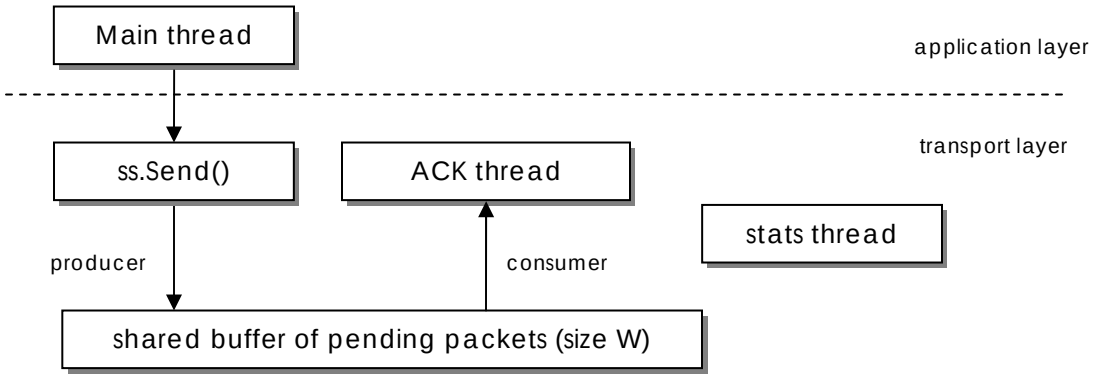

Figure 1. Organization of the program.

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 unblock `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 retransmissions among the packets with sequence numbers in $[x, x + y - 1]$. Finally, when moving the window 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:

```
int kernelBuffer = 20e6;              // 20 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 may need to 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):

```
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:

```
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)

        // how much we can advance the semaphore
        newReleased = sndBase + effectiveWin - lastReleased
        ReleaseSemaphore (s, newReleased)
        lastReleased += newReleased
    }
```

```
}
```

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 0.115 sec, pkt size 1472 bytes
[  2] B    126 (  0.2 MB) N    126 T 0 F 0 W 10 S 0.740 Mbps RTT 0.116
[  4] B    286 (  0.4 MB) N    296 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[  6] B    446 (  0.7 MB) N    456 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[  8] B    606 (  0.9 MB) N    616 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[ 10] B    766 (  1.1 MB) N    776 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[ 12] B    926 (  1.4 MB) N    936 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[ 14] B   1086 (  1.6 MB) N   1096 T 0 F 0 W 10 S 0.936 Mbps RTT 0.116
[ 16] B   1246 (  1.8 MB) N   1256 T 0 F 0 W 10 S 0.936 Mbps RTT 0.115
[ 18] B   1416 (  2.1 MB) N   1417 T 0 F 0 W 10 S 0.995 Mbps RTT 0.115
[ 20] B   1576 (  2.3 MB) N   1586 T 0 F 0 W 10 S 0.936 Mbps RTT 0.115
[ 22] B   1736 (  2.6 MB) N   1746 T 0 F 0 W 10 S 0.936 Mbps RTT 0.115
[ 24] B   1896 (  2.8 MB) N   1906 T 0 F 0 W 10 S 0.936 Mbps RTT 0.113
[ 26] B   2046 (  3.0 MB) N   2056 T 0 F 0 W 10 S 0.877 Mbps RTT 0.127
[ 28] B   2186 (  3.2 MB) N   2196 T 0 F 0 W 10 S 0.819 Mbps RTT 0.127
[ 30] B   2336 (  3.4 MB) N   2346 T 0 F 0 W 10 S 0.878 Mbps RTT 0.127
[ 32] B   2476 (  3.6 MB) N   2486 T 0 F 0 W 10 S 0.819 Mbps RTT 0.127
[ 34] B   2616 (  3.9 MB) N   2626 T 0 F 0 W 10 S 0.819 Mbps RTT 0.127
[ 36] B   2766 (  4.1 MB) N   2776 T 0 F 0 W 10 S 0.877 Mbps RTT 0.126
[37.837] <-- FIN-ACK 2865 window 5B0360D
Main:   transfer finished in 37.594 sec, 892.54 Kbps, checksum 5B0360D
Main:   estRTT 0.126, ideal rate 931.37 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 811 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.123 sec, pkt size 1472 bytes
[  2] B  14559 ( 21.4 MB) N  22094 T 0 F 0 W 12000 S 85.553 Mbps RTT 0.130
[  4] B  85837 (126.4 MB) N  97837 T 0 F 6 W 12000 S 416.825 Mbps RTT 0.207
[  6] B 170753 (251.3 MB) N 182753 T 2 F 12 W 12000 S 496.914 Mbps RTT 0.187
[  8] B 249672 (367.5 MB) N 259675 T 3 F 19 W 12000 S 461.824 Mbps RTT 0.175
[ 10] B 325519 (479.2 MB) N 337519 T 5 F 29 W 12000 S 443.425 Mbps RTT 0.116
[ 12] B 394961 (581.4 MB) N 405233 T 7 F 40 W 12000 S 406.378 Mbps RTT 0.103
[ 14] B 472303 (695.2 MB) N 484303 T 7 F 47 W 12000 S 452.591 Mbps RTT 0.173
[ 16] B 545545 (803.0 MB) N 557545 T 7 F 55 W 12000 S 428.601 Mbps RTT 0.193
[ 18] B 628755 (925.5 MB) N 636984 T 8 F 61 W 12000 S 486.933 Mbps RTT 0.101
[ 20] B 694104 (1021.7 MB) N 706104 T 9 F 68 W 12000 S 382.410 Mbps RTT 0.161
[21.170] <-- FIN-ACK 733431 window E8F5B708
Main:   transfer finished in 20.902 sec, 410954.64 Kbps, checksum E8F5B708
Main:   estRTT 0.127, ideal rate 1110625.81 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 0.022 sec, pkt size 1472 bytes
[  2] B    164 (  0.2 MB) N    174 T 4 F 16 W 10 S 0.962 Mbps RTT 0.036
[  4] B    363 (  0.5 MB) N    373 T 8 F 32 W 10 S 1.165 Mbps RTT 0.036
[  6] B    600 (  0.9 MB) N    601 T 12 F 43 W 10 S 1.387 Mbps RTT 0.036
[  8] B    774 (  1.1 MB) N    784 T 18 F 56 W 10 S 1.018 Mbps RTT 0.036
[ 10] B    953 (  1.4 MB) N    955 T 25 F 69 W 10 S 1.047 Mbps RTT 0.035
[ 12] B   1107 (  1.6 MB) N   1117 T 32 F 83 W 10 S 0.900 Mbps RTT 0.032
[ 14] B   1298 (  1.9 MB) N   1308 T 39 F 97 W 10 S 1.118 Mbps RTT 0.032
[ 16] B   1518 (  2.2 MB) N   1528 T 42 F 115 W 10 S 1.287 Mbps RTT 0.031
[ 18] B   1740 (  2.6 MB) N   1750 T 50 F 131 W 10 S 1.298 Mbps RTT 0.029
[ 20] B   2001 (  2.9 MB) N   2011 T 55 F 149 W 10 S 1.527 Mbps RTT 0.029
[ 22] B   2248 (  3.3 MB) N   2258 T 62 F 170 W 10 S 1.445 Mbps RTT 0.028
```

```
[ 24] B   2513 (  3.7 MB) N   2523 T 67 F 190 W 10 S 1.551 Mbps RTT 0.028
[ 26] B   2773 (  4.1 MB) N   2783 T 73 F 208 W 10 S 1.521 Mbps RTT 0.028
[26.730] <-- FIN-ACK 2865 window 5B0360D
Main:   transfer finished in 26.675 sec, 1257.91 Kbps, checksum 5B0360D
Main:   estRTT 0.028, ideal rate 4199.75 Kbps
```

4.9. Bottlenecked by Win/RTT

```
Main:   sender W = 300, RTT 0.100 sec, loss 0.001 / 0, link 1000 Mbps
Main:   initializing DWORD array with 2^24 elements... done in 15 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.118 sec, pkt size 1472 bytes
[  2] B    984 (  1.4 MB) N   1284 T 0 F 4 W 300 S 5.792 Mbps RTT 0.118
[  4] B   5008 (  7.4 MB) N   5008 T 0 F 11 W 300 S 23.548 Mbps RTT 0.121
[  6] B   9384 ( 13.8 MB) N   9482 T 0 F 13 W 300 S 25.594 Mbps RTT 0.125
[  8] B  13847 ( 20.4 MB) N  14147 T 0 F 15 W 300 S 26.117 Mbps RTT 0.137
[ 10] B  17409 ( 25.6 MB) N  17709 T 0 F 21 W 300 S 20.844 Mbps RTT 0.137
[ 12] B  21437 ( 31.6 MB) N  21438 T 0 F 27 W 300 S 23.571 Mbps RTT 0.121
[ 14] B  25803 ( 38.0 MB) N  26092 T 0 F 30 W 300 S 25.524 Mbps RTT 0.118
[ 16] B  29561 ( 43.5 MB) N  29861 T 0 F 33 W 300 S 21.980 Mbps RTT 0.121
[ 18] B  33545 ( 49.4 MB) N  33655 T 0 F 38 W 300 S 23.313 Mbps RTT 0.123
[ 20] B  37275 ( 54.9 MB) N  37382 T 0 F 43 W 300 S 21.827 Mbps RTT 0.124
[ 22] B  41911 ( 61.7 MB) N  42139 T 0 F 45 W 300 S 27.129 Mbps RTT 0.118
[ 24] B  45604 ( 67.1 MB) N  45840 T 0 F 51 W 300 S 21.610 Mbps RTT 0.117
[24.680] <-- FIN-ACK 45840 window 85A854D4
Main:   transfer finished in 24.430 sec, 21976.30 Kbps, checksum 85A854D4
Main:   estRTT 0.117, ideal rate 29960.34 Kbps
```

4.10. Surviving Heavy Loss

```
Main:   sender W = 10, RTT 0.010 sec, loss 0.5 / 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 0.023 sec, pkt size 1472 bytes
[  2] B     20 (  0.0 MB) N     30 T 19 F 1 W 10 S 0.117 Mbps RTT 0.025
[  4] B     44 (  0.1 MB) N     54 T 40 F 1 W 10 S 0.140 Mbps RTT 0.025
[  6] B     69 (  0.1 MB) N     79 T 59 F 3 W 10 S 0.146 Mbps RTT 0.025
[ 7.777] <-- FIN-ACK 90 window FC694CF3
Main:   transfer finished in 7.721 sec, 135.81 Kbps, checksum FC694CF3
Main:   estRTT 0.025, ideal rate 4591.14 Kbps
```

4.11. Extra Credit

```
Main:   sender W = 3000, RTT 0.010 sec, loss 0 / 0, link 10000 Mbps
Main:   initializing DWORD array with 2^29 elements... done in 1466 ms
Main:   connected to s8.irl.cs.tamu.edu in 0.030 sec, pkt size 1472 bytes
[  2] B 112975 (166.3 MB) N 115121 T 0 F 0 W 3000 S 664.167 Mbps RTT 0.030
[  4] B 237874 (350.2 MB) N 240078 T 0 F 8 W 3000 S 730.539 Mbps RTT 0.024
[  6] B 353291 (520.0 MB) N 354036 T 0 F 24 W 3000 S 675.164 Mbps RTT 0.011
[  8] B 486832 (716.6 MB) N 489511 T 0 F 31 W 3000 S 781.460 Mbps RTT 0.033
[ 10] B 623097 (917.2 MB) N 625143 T 0 F 39 W 3000 S 797.056 Mbps RTT 0.022
[ 12] B 750866 (1105.3 MB) N 752293 T 0 F 55 W 3000 S 747.299 Mbps RTT 0.018
[ 14] B 885112 (1302.9 MB) N 885947 T 0 F 71 W 3000 S 785.212 Mbps RTT 0.014
[ 16] B 1017479 (1497.7 MB) N 1020479 T 0 F 86 W 3000 S 774.231 Mbps RTT 0.034
[ 18] B 1154644 (1699.6 MB) N 1157268 T 0 F 97 W 3000 S 802.668 Mbps RTT 0.035
[ 20] B 1289677 (1898.4 MB) N 1292677 T 0 F 112 W 3000 S 790.195 Mbps RTT 0.035
[ 22] B 1417353 (2086.3 MB) N 1419904 T 0 F 127 W 3000 S 747.138 Mbps RTT 0.031
[23.074] <-- FIN-ACK 1466861 window EF5F9797
Main:   transfer finished in 23.012 sec, 746560.18 Kbps, checksum EF5F9797
Main:   estRTT 0.026, ideal rate 1348868.42 Kbps
```

CSCE 463/612 Homework 3 Part 1

Name: _____

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

Total points: _____

CSCE 463/612 Homework 3 Part 2

Name: _____

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

Total points: _____

CSCE 463/612 Homework 3 Part 3

Name: _____

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

Total points: _____