CSCE 463/612: Networks and Distributed Processing

Homework 3 Part 1 (25 pts)

Due date: 3/25/25

1. Purpose

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.

2. Description

This part implements only the connection handshake (SYN) and termination (FIN).

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 (see below).

2.1. Code (25 pts)

The homework must accept as input seven command-line parameters of the transfer – destination server (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 s3.irl.cs.tamu.edu 24 50000 0.2 0.00001 0.0001 100
```

requests communication with s3.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 `s3.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 (int argc, char **argv)
{
	// 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 s3.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-05) 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 sends 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 s3.irl.cs.tamu.edu in 2.237 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 s38.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. 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 given by class `SenderSynHeader`:

```
#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;
};
```

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

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

The Flags header contains five fields:

```
#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.*

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 d) 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 – you should 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 the course website and run it locally for debugging purposes (make sure to set your destination to 127.0.0.1 instead of s3.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 that omits checksums, bypasses router emulation, and allows (9000-28)-byte packets. You could get over 10 Gbps with this version.

463/612 Homework 3 Grade Sheet (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: _____