RELIABLE DATA TRANSFER (RDT 3.0) PROTOCOL

OVERVIEW

This assignment requires you to implement the 1-bit Sliding Window Protocol *rdt 3.0* (Alternating Bit protocol) as described in class. You must write three programs written in ANSI C to model the *sender*, the *receiver*, and the *network* that simulates a network connecting a *sender* with a *receiver*. All three programs must use UDP message passing and only use one UDP socket for communication. You may use code from the UDP client server programs you implemented in project 1.

The *sender* is waiting for 3 events in *rdt3.0*: data to send, timer to go off, and acks from the network. For the *sender* to obtain data to send from the layer above (e.g., user), your program must prompt the user for input. Upon entry of a message to send, your *sender* must segment the data and send messages in individual segments, one at a time according to the *rdt 3.0* sender protocol. The *receiver* must assemble the individual segments in the correct order to recreate the original message. It implements the *rdt 3.0* receiver protocol. The *network* program acts as a transport service, simulating a network that passes messages between *sender* and *receiver*.

Protocol *rdt3.0* is robust in the presence of errors and lost messages. Sequence numbers and acks are required as specified by the protocol. The sequence and ack numbers will either have a value of 0 or 1, and the only buffering required is of the last message sent. To demonstrate that your program is robust the *receiver* must assemble the individual segments in the payload and print the resulting message to standard out. The message the user typed in as input in the *sender* must be identical with the message received by the *receiver* for the transmission to be successful. The message must be sent in multiple segments to the recipient based on the segment's size. For this project, you must use a segment size of 10 bytes (transport header & payload together make up exactly 10 bytes). This means that if the complete message to be delivered is 50 bytes long, then you will have more than 5 segments that need to be sent to the recipient. In tests, I will have messages that will be longer than 10 bytes.

This project is divided into parts A, B, and C each dealing with different aspects of the project.

PART A: IMPLEMENTING RDT3.0

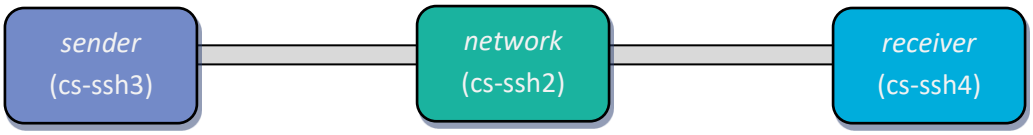
In this part, you must correctly implement the protocol in both the *sender* and *receiver* program. The `main()` function in each program should be a driver program accepting input from the user and/or command line parameters and then calling the appropriate function to send or receive a message. All variables should be declared in `main()` and passed to the appropriate functions. You may not have global variables! Furthermore, you must use the provided header files `rdtSender.h` and `rdtReceiver.h` posted in *eLearning* to implement your code. The driver program for the *sender* and *receiver* will make the calls of the functions provided in the header files to send and receive messages.

The *sender*, *receiver*, and *network* should bind their sockets to a user-specified port and print the host name and port number on the screen. When the *sender* is executed, the host name and port number of the *network* and the host name and port number of the *receiver* should be passed in as parameters to `main()`. Part B below describes the exact input parameters for all three programs. When the *network* is executed it receives packets from the *sender* and the *receiver* and forwards them to the corresponding destination address using information in the network header of the data packet exchanged between all three programs. The network header must be created by the *sender* and *receiver* and read by the *network* to direct traffic. It is independent from the actual network header

implemented by the computer's network layer service. The "simulated" network header is used by the *network* program to determine the destination address of the packets sent to the different programs via the UDP protocol. The diagram below illustrates the network header that must be implemented by this project. Note that the 10 byte segment length used by RDT is included in the entire 54 byte long UDP packet. Destination and source IP addresses and port information of the sender and receiver is represented as a string and not an actual binary number.

network layer				transport layer
src IP (16 bytes)	src port (6 bytes)	dest IP (16 bytes)	dest port (6 bytes)	segment
0 15	16 21	22 37	38 43	44 53

With the information provided at startup and the information that is being included in the network header all three programs can setup a UDP communication channel for sending messages back and forth. The figure below illustrates the communication channel involving a *sender*, *receiver*, and a *network* that serves as a message transporter between *sender* and *receiver*. Note, all three programs run on different machines.



The sender must use UDP `sendto()` to transmit a UDP packet to the known location of the *network* program. However, the *network* program must use the information in the UDP packet's header to forward the packet to the correct receiver location. The receiver after performing a `recvfrom()` will know the location of the network program. The receiver's reply must be transmitted back to the network. The receiver must reverse the source and destination address information in the network header of the UDP packet so that the network can forward the message to the correct destination, which will be the sender.

The time to send and receive a packet in our physical network is very short. Therefore, you will need to make use of the `sleep()` function in the *network* program to test whether your timer in the *sender* is working properly. To handle the timer event and the arrival of an acknowledgement event, you must use the `select()` system call. The manual page for `select()` has the necessary information for the project. I have uploaded a handout in *eLearning* for this week's unit with a more detailed description of `select()`.

PART B: SIMULATING LOST, CORRUPT, OR DELAYED MESSAGES

In our physical network, it is highly unlikely timeouts will occur or packets will be lost. Recall that a timeout occurs if a packet is delayed or lost. Your system must simulate lost packets using the *network* that delivers data to the *sender* and *receiver*. Random drop of packets is easily accomplished by using random numbers to decide when to drop a packet. The *network*'s send function that is sending a packet to the *receiver* must use a random number to determine whether to call `sendto()` or not, to simulate a lost packet on the network.

The *network* also simulates slow packet transport within the network by delaying transport of a packet to its destination using a fixed time delay. This time delay should be between 1.5 and 2 times the waiting time for the *sender* to resend packets. To accurately simulate slow data transport, the process delaying the transport should be able to continue execution. This means that simply calling `sleep()` to delay sending a packet will block the process preventing it to accept any new packets or sending new packets. Instead, you need to create a new thread within

the *network* program that will be tasked with sending the packet with some delay using `sleep()`. You must simulate the probability of a lost or delayed packet as some percentage such as 60%. The two separate parameters for specifying the delayed and lost packet probabilities must be accepted as an input parameter in `main()` to the *network*.

On our internal network, it is unlikely that your packets will be corrupt. You will need to simulate damaged data using a tag in the segment to indicate that your segments are either free of error or corrupt – no checksum computation is needed. Randomly select the tag to be either corrupt or not corrupt in each segment of your packet including acknowledgments. Corrupt segments must also occur at a set percentage rate as above. Similarly, the percentage of corrupted segments must be accepted as an input parameter in `main()` to the *network*, which handles the simulation of damaged data.

Your implementation of rdt3.0 should be as robust as possible in the presence of lost, delayed, and damaged data. Remember both data and ack messages can be lost, delayed, or corrupted! This means that you must simulate data and ack messages being lost, delayed, or corrupted.

The argument parameters for *receiver*, *network*, and *sender* must follow this format:

`receiver` *`port`*

`sender` *`port rcvHost rcvPort networkHost networkPort`*

`network` *`port lostPercent delayedPercent errorPercent`*

The parameter *`port`* specifies a valid port number for a program to bind a UDP socket. The parameter *`rcvPort`* and *`networkPort`* specify the corresponding port numbers for communicating with a *receiver* and a *network* program. The parameters *`networkHost`* and *`rcvHost`* represent the host names for the *network* and the *receiver*. Finally the parameters *`lostPercent`*, *`delayedPercent`*, and *`errorPercent`* represent the corresponding percentages for lost, delayed, and corrupt packages as integers in the range of 0 to 100, not as a fraction of 0.0 to 1.0. For example, a value of 60 as percentage parameter would represent 60%.

PART C: RECORDING NETWORK TRAFFIC

The network program simulates a network that transports packages from a sender to a receiver and vice-versa using the information in the network header of the packet. Furthermore, as described above, the program simulates lost, corrupt, and delayed transport of packets. In this part, you must extend the features of the *network* program to record the traffic the program observes during the simulation. It must record the number of packets that it receives from a sender and the number of replies it receives from a receiver. As senders are the initiator of communication, the network "knows" that any packet it received from a new host must be the sender. As information about traffic is collected, the network must periodically print out statistics about the traffic including the number of packets transmitted between sender and receiver, as well as the number of delayed, dropped, and corrupt packets. For example, in your simulation, the network might print out statistics about the traffic for every other packet that it receives. It is up to you to decide on a format that makes analysis of traffic easy to inspect for the user.

IMPLEMENTATION SUGGESTIONS

For your implementation consider the following system calls:

socket()	**gethostbyname()**	**gethostname()**	**bind()**
sendto()	**recvfrom()**	**close()**	**select ()**

DELIVERABLES & EVALUATION

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You should have at a minimum the following files for this assignment:
    a. source code for `sender`, `network`, and `receiver`
    b. a single Makefile to compile the three programs,
    c. a protocol document
    d. a README file (optional if project wasn't completed)

The protocol document must describe briefly the respective algorithms of the *sender* and *receiver* program and the format of the transport-layer segment. The README file should only be included if you submit a partial solution. In that case, the README file must describe the work you did complete.

Your program will be evaluated according to the steps shown below. Notice that I will not fix your syntax errors. However, they will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *eLearning*.
    - If errors occur during compilation, the instructor will not fix your code to get it to compile. The project will be given zero points.
    - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Program documentation and code structure.
    - The source code must be properly documented and the code must be structured to enhance readability of the code.
    - Each source code file must include a header that describes the purpose of the source code file, the name of the programmer(s), the date when the code was written, and the course for which the code was developed.
3. Program evaluation using several test runs with input of the grader's own choosing. At a minimum, the test runs address the following questions.
    - Will the *sender*, *receiver*, and *network* communicate?
    - Will the *sender* and *receiver* execute the protocol to transport large messages?
    - Will the *network* simulate delayed, dropped, and corrupt messages?
    - Will the *network* print statistics about the traffic?

Keep in mind that documentation of source code is an essential part of computer programming. In fact the better your code is document the better it can be maintained and reused. If you do not include comments in your source code, points will be deducted. I also require you to refactor your code to make it more manageable and to avoid memory leaks. Points will be deducted if you don't refactor your code or if we encounter memory leaks in your program during testing.

DUE DATE

The project is due as indicated by the Dropbox for project 2 in *eLearning*. Upload your complete solution to the dropbox and the shared drive (if available). I will not accept submissions emailed to me or the grader. Upload ahead of time, as last minute uploads may fail.

TESTING

Your solution needs to compile and run on the CS department's SSH servers. I will compile and test your programs running several undisclosed test cases and using three servers to run the *sender*, *receiver*, and *network*. Therefore, to receive full credit for your work it is highly recommended that you test & evaluate your solution on the servers to make sure that I will be able to run your programs and test them successfully. You may use any of the five SSH servers (cs-ssh1.cs.uwf.edu, ..., cs-ssh5.cs.uwf.edu) available to you for programming, testing, and evaluation. For security reasons, the majority of ports on the servers have been blocked from communication. Ports that are open for communication between the public servers range in values from 60,000 to 60,099.

GRADING

This project is worth 100 points in total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, crashes, or is otherwise, poorly documented or organized. The points will be given based on the following criteria:

Submission	Perfect	Deficient		
eLearning	5 points individual files have been uploaded	0 points files are missing		
shared drive (if available)	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes clean rule	3 points missing clean rule	2 points missing rules; does not compile project	0 points make file is missing
compilation	10 points no errors, no warnings	7 points some warnings	3 points many warnings	0 points errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	5 points follows documentation and code structure guidelines	3 points follows mostly documentation and code structure guidelines; minor deviations	2 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines

Sender	Perfect	Good	Attempted	Deficient
Prompts user for message and segments it.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Implements sender protocol with time-out.	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Receiver	**Perfect**	**Good**	**Attempted**	**Deficient**
Assembles segments into complete message and prints it out.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Implements receiver protocol.	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Network	**Perfect**	**Good**	**Attempted**	**Deficient**
Transports messages between sender and receiver.	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Simulates drop of network packets.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Simulates corruption of network packets.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Simulates delay of network packets with separate threads.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Records and prints traffic sent between sender and receiver.	5 points correct, completed	4 points minor errors	1 points incomplete	0 points missing or does not compile
Report	**Perfect**	**Good**	**Attempted**	**Deficient**
Describes the algorithm implemented by the sender and receiver.	5 points fully described	4 points mostly described	1 point critical information is missing	0 point incomplete or missing
Describes the format of messages exchanged between sender, receiver, and proxy.	5 points fully described	4 points mostly described	1 point critical information is missing	0 point incomplete or missing

Not shown above are deductions for memory leaks and run-time issues. Up to 15 points will be deducted if your program has memory leaks or run-time issues such as crashes or endless loops when tested on the SSH server.