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 a proxy that simulates a network. 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 2.

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 proxy program acts as a transport service, passing messages between sender and receiver to simulate a network.

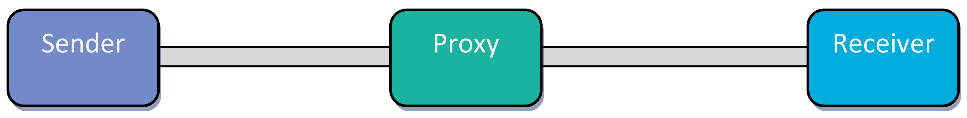
Protocol *rdt3.0* is robust in the presence of errors and lost messages. Sequence numbers and acks are required as specified by the protocol. The sequence and ack numbers will either have a value of 0 or 1, and the only buffering required is of the last message sent. To demonstrate that your program 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 (header & payload together make up 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 two parts A and B.

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 proxy should bind their sockets to a user-specified port and print the host name and port number on the screen. When the proxy is executed, the host name and port number of the receiver should be passed in as parameters to `main()`. Likewise, when the sender is executed, the host name and port number of the proxy should be passed in as parameters to `main()`. The second part below includes an input parameter specification for all three programs. With the provided information, all three programs can setup a communication channel for sending messages back and forth. The figure below illustrates the communication channel involving a sender, receiver, and a proxy that serves as a message transporter between sender and receiver.



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 proxy 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 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 proxy 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 proxy'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 proxy 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 proxy 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 proxy.

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 packet to indicate that your packets 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 packet including acknowledgments. Corrupt packets must also occur at a set percentage rate as above. Similarly, the percentage of corrupted packets must be accepted as an input parameter in `main()` to the proxy, which handles the simulation of damaged data.

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

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

```
receiver port

sender port proxyHostname proxyPort

proxy port rcvHostname rcvPort lostPercent delayedPercent errorPercent
```

The parameter `port` specifies a valid port number for a program to bind a socket. The parameter `rcvPort` and `proxyPort` specify the corresponding port numbers for communicating with a receiver and a proxy. The parameters `proxyHostname` and `rcvHostname` represent the host names for the proxy 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%.

IMPLEMENTATION SUGGESTIONS

For your implementation consider the following system calls:

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

DELIVERABLES & EVALUATION

Submit your complete solution as a single zip file (only zip files will be accepted) containing A) source code, B) a single makefile to compile the three different programs, C) a report in PDF format, and D) a README file (if applicable) to the corresponding dropbox in *eLearning*.

The report must describe briefly the respective algorithms of the sender, receiver, and proxy and the format of messages send between the three peers. 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.

You must follow the *Project Submission Instructions* posted in *eLearning* under Course Materials. The submission requirements are part of the grading for this assignment. If you do not follow the requirement, points will be deducted from your project grade. Keep in mind that documentation of source code is an essential part of programming. 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 do not refactor your code or if I encounter memory leaks in your program while testing it.

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 proxy. 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.

DUE DATE

The project is due March 27th by 11:00 pm. Late submissions will not be accepted and I will not accept any emailed solutions. The syllabus contains details in regards to late submissions.

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	Attempted	Deficient	
eLearning	2 points Deliverable zipped in folder 'Student1Student2proj#'	1 point	0 points	
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	4 points make & make clean	3 points	2 points	0 points
Compiles	10 points	7 points Some warnings	3 points Some errors	0 points
Documentation & Style	**Perfect**	**Good**	**Attempted**	**Deficient**
Documentation & Style	5 points Functions documented. Student-written files have file comment header: Student Name, Filename, Assignment Number, Class. Whitespace used properly.	3 points	2 points	0 points
Sender	**Perfect**	**Good**	**Attempted**	**Deficient**
Prompts user for message and segments it.	10 points	7 points	3 points	0 points
Implements sender protocol with time-out.	12 points	8 points	4 points	0 points
Receiver	**Perfect**	**Good**	**Attempted**	**Deficient**
Assembles segments into complete message and prints it out.	10 points	7 points	3 points	0 points
Implements receiver protocol.	12 points	8 points	4 points	0 points
Proxy	**Perfect**	**Good**	**Attempted**	**Deficient**
Transports messages between sender and receiver.	10 points	7 points	3 points	0 points
Simulates drop of network packets.	5 points	4 points	1 point	0 point
Simulates corruption of network packets.	5 points	4 points	1 point	0 point
Simulates delay of network packets.	5 points	4 points	1 point	0 point
Report	**Perfect**	**Good**	**Attempted**	**Deficient**
Describes algorithms.	5 points	4 points	1 point	0 point
Describes format of messages exchanged between sender, receiver, and proxy.	5 points	4 points	1 point	0 point

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.