COMPUTER SCIENCES AND SOFTWARE ENGINEERING
AUBURN UNVERSITY

COMP 4320

Introduction to Computer Networks

Fall 2021

Project 2

Implementation of a Selective Repeat Reliable Data Transfer for a Web Service over the UDP Transport Service

Due in Canvas: 11:55pm December 2, 2021

Objective

The purpose of this assignment is to implement a *Selective Repeat* reliable data transfer for Web service over the UDP transport service. Your Selective Repeat protocol must correctly transmits segments even though the transport layer uses a channel that may cause some packets to be lost and some packets to contain bit errors. In this project, you will understand the sliding window protocol better in terms of its flow control and error-detection/correction features. You will implement the Selective Repeat protocol for ensuring reliability in your Web client and server programs.

Overview

In this project, you will implement a Selective Repeat protocol for ensuring reliable data transfer for the web client and server applications. As in Project 1, these applications ***must be written in C or C++*** and execute correctly either in your own computer(s) or in the COE tux Linux computers. Other related software modules that you will implement are the segmentation and re-assembly function, an error detection function and a gremlin function (that can corrupt *and* lose packets with specified probabilities). The overview of these software components is show in Figure 1 below.

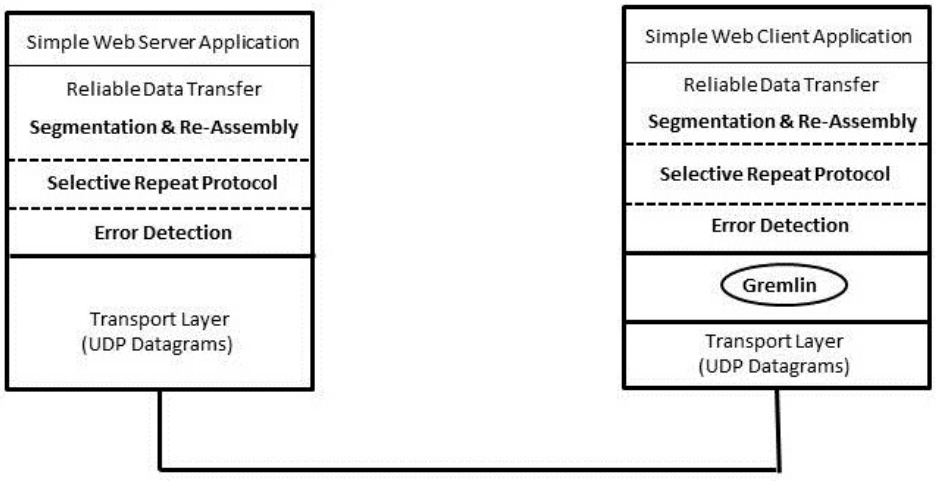

Figure 1. Overview of the Selective Repeat Protocol and other Software Components

As in the first project, the Web client initiates the communication by sending an HTTP request to the Web server. This outgoing HTTP request is *not* processed by the segmentation and re-assembly, selective repeat protocol, error detection or the Gremlin function. In this project, you will only enforce reliability in the transfer of data from the Web server to the client. The HTTP request is sent through the transport UDP datagram socket to the Web server.

At the Web server host, the HTTP request is also not processed by those functions at the Web server host. The Web server will then process the request, e.g. GET command, by reading the file requested by the Web client. Since the requested file may be large, the server application will use the segmentation function to partition the file into smaller segments that will fit into a packet of size allowable by the network. Each segment is then placed into a 512-byte packet that is allowed by the network. Each packet must contain a header that contains information for the selective repeat protocol, error detection, segmentation and other protocol information. You may design your own header fields that are of reasonable sizes. Another field that must be in the header is a sequence number. The packet is then passed to the selective repeat protocol that will use the sequence number and acknowledgements to guarantee reliable data transfer. The packet then passed to the error detection function which, at the server (sending process), will compute the checksum and place the checksum in the header. The packet is finally sent via the UDP socket to the Web client.

When the packet is received by the Web client host UDP socket, the packet will be processed by the Gremlin function which may randomly cause errors in some *packets or lose some packets.* This will emulate errors that may be generated by the network links and routers. The packet is then processed by the error detection function that will detect possibility of error based on the checksum. The packet is then processed by the selective repeat protocol by checking the sequence number and process the acknowledgement accordingly. Correct packets are then passed to the segmentation and re-assembly function that re-assembles all the segments of the file from the packets received into the original file. The file is then displayed by the Web client application using a display software or browser.

Descriptions

The Selective Repeat protocol is a reliable transmission protocol that is fairly efficient. The main purpose of the protocol is to convert an unreliable channel into a reliable one by detecting and recovering from errors in the packets as well as lost packets.

You may reuse the functionalities implemented in the first project for segmenting and reassembling packets, lost and scrambled. The unreliable channel is emulated by the gremlin function you implemented that will cause some bits in the packets to be flipped in addition to losing some packets. The middle sub-layer contains the Selective Repeat protocol that you will implement in this project. The top sub-layer contains the

segmentation and re-assembly function you implemented in the first project that will segment and reassemble the large data segment (file) that is being transmitted.

The Web application programs will transmit the contents of an ASCII data file. The Web server passes the data in the file to the segmentation and re-assembly module which breaks the data stream into smaller 512-bytes packets (including the flags and header). It then sends the first packet to the Web client. The error detection (ED) module generated and inserts the checksum for error checking. Finally, the Selective Repeat protocol will send several packets in a pipeline and waits for the acknowledgements from the receiver. If the sender times out before receiving the ACK or detects an error in a packet, it will assume that the packets are lost and retransmit the packets.

You will design and implement the Selective Repeat protocol with positive/negative acknowledgement and retransmission (PAR). You must use the window size, **N = 8** for both the sender and receiver. The sequence number must be modulo **24**. Your Web application will use this Selective Repeat protocol to transmit the contents of an ASCII data file. The Web server application receive a request from the Web client (e.g. GET) and reads data from an ASCII file. The SAR function breaks the large messages into smaller packets. It then sends these packets to the Web client application through the Selective Repeat protocol. When the client FTP application correctly receives all the packets in the right order from the server, it will reassemble the file that is transferred.

You will have considerable freedom in designing the service interfaces, the packet-header formats, and the implementation mechanisms for these protocols.

Selective Repeat Protocol (SR)

You are to design and implement a Selective Repeat protocol where the window sizes for both the sender and receiver are 8 and the sequence number is modulo 24. Your Selective Repeat protocol must deal with errors in packets and lost packets.

The Selective Repeat protocol follows the pipeline principle as follows. After the transmitter sends packet 0, the transmitter then sends seven additional packets into the channel, optimistic that packet 0 will be received correctly and not require retransmission. If that turns out to be right, then the ACK for packet 0 will arrive while the transmitter is still busy sending packets into the channel. Handling of packet 0 will then be done while the handling of packet 1 and subsequent packets is already underway. Thus, Selective Repeat pipelines the processing of packets before the completion of previous packet transmission to keep the channel busy.

In your Selective Repeat protocol, the receiver must handle packet errors and lost packets the following ways.

1. When the receiver receives a packet, it must use the same Error Detection algorithm as used by the transmitter to check for checksum for errors. If the packet is free of error and can be placed in the right order in the receiver's buffer, it sends back an ACK for this packet to the transmitter. This ACK is sent in a

vector along with ACKs and NAKs of 7 other packets, since the receiver window is 8.
2. When the receiver detects an error in a packet, it *must* send back a NAK to the transmitter, in a vector along with ACKs and NAKs of 7 other packets,

ACK and NAK are never lost or damaged by the gremlin process.

For each of the three corresponding cases above, the transmitter must respond as follows:

1. When the transmitter receives an ACK, it must store that information in its buffer. If the ACKs are for the first few packets in the window, then it must advance its window forward. For example, after the transmitter with send window [0-7] transmits packets 0 to 7, it receives a vector with ACKs for packet 0 and packet 2. Then the transmitter must advance its send window to [1-8] and sends the new packet 8.
2. When the transmitter receives a NAK with sequence number 1 and 3, it will retransmit packets [1,3]. A NAK with sequence number 1 indicates that the receiver has not received packet 1 correctly but may still accept all subsequent packets.
3. When the receiver does not send back either an ACK or NAK, then the transmitter will timeout on the earliest packet that has not been ACKed, e.g. packet 1 timeouts. It then retransmits only packet 1.

There is an interesting interaction between responses 1, 2 and 3 above. For example, if the transmitter timeouts and is in the process of retransmitting a window size of packets and a new ACK arrives. A simpler implementation is to complete the retransmission of those packets first before servicing the receiving of the ACK. Although this implementation is simpler, it is inefficient because the incoming ACK may indicate that the receiver has received some of the packets that are being retransmitted. A more efficient implementation is to cause the incoming ACK to interrupt the retransmission and the transmitter can then avoid retransmitting those packets that are being ACKed. Although the second method is more efficient, it is also more difficult to implement and the saving in efficiency may not be worth the complexity. In this project, you may choose either implementation.

When the sender receives a NAK with sequence number *n*, it must stop the timer and immediately retransmit packet *n*. It also updates all the other software timers.

For each packet transmitted, the timeout value must not be more than four times the round trip time. If you assume the round trip time to be typically about 5 millisec, then the timeout value should not be more than ***20 millisec***.

Each packet should also have a checksum inserted into a pre-specified position in the packet. The sender computes the values of the checksum and inserts them into the packet. The receiver uses the same algorithm to check the checksum values. You can use any algorithm to check for error, so long as the receiver and sender use the same algorithm. If

the client detects an error in the packet, **it must print out in its output trace that the packet has errors with the packet sequence number**.

The packet header should contain a one-byte sequence number. As each data packet is composed, the sending process sends it to the receiving process using UDP. When the client receives packets, it must check the sequence number of the packets to make sure that the sequence number is within the receive window. **It prints the sequence number in the output trace** and indicates if there are lost packets or corrupted packets.

Gremlin Function

Your program must allow the probability of damaged *and lost* packets to be input as arguments when the program is executed. This packet damage *and lost* probabilities is passed to your Gremlin function. You will implement a gremlin function to simulate *three* possible scenarios in the transmission line: transmission error that cause packet corruption, *packet loss* and correct delivery. When the receiving process receives each packet, it first pass the packet to a gremlin function which will randomly determine whether to change (corrupt) some of the bits or pass the packet as it is to the receiving function. It will also decide whether some packets will be dropped based on the loss probability. The gremlin function uses a random-number generator to determine whether to damage a packet or pass the packet as it is to the receiving function.

If the gremlin decides to *lose a packet*, then the client's selective repeat protocol will not send an ACK back to the server. For example, a packet's loss probability of 0.2 means that two out of ten packets are dropped.

If it decides to damage a packet, it will decide on how many and which byte to change. The probability that a given packet will be damaged, P(d), is entered as an argument at runtime. If the probability of damaging a packet is .3, then three out of every ten packets will be damaged. If the packet is to be damaged, the probability of changing one byte is .5, the probability of changing two bytes is .3, and the probability of changing 3 bytes is .2. Every byte in the packet is equally likely to be damaged. The packet is then passed from the gremlin function to the error detection function that will check for errors in the packet.

Error Detection Function

The sending process, e.g. the Web Server, will compute the checksum for the packet that is to be sent. The checksum is calculated by simply summing all the bytes in the packet. The checksum is then inserted into the checksum header field of the packet.

The receiving process, e.g. the Web client, will then use the same algorithm for computing the checksum that the sending process used. It will calculate the checksum by summing all the bytes in the received packet. It then compares the computed checksum with the checksum received in the packet. If the two checksums match, then it assumes that there is no error, otherwise there is at least an error in the packet.

When the receiving process detects an error in a packet, it will print out a message indicating the packet's sequence number and that there is an error in the packet.

In this project, the receiver of your selective repeat protocol will handle errors in packets by sending a NAK packet with the sequence number. The sender will then retransmit the packets that were NAKed.

Testing

To test the selective repeat protocol, integrate it into your *simple Web service,* with the segmentation and re-assembly, error detection and gremlin functions, implemented in the first project. Your programs must execute correctly in your own computer *or* in our College of Engineering tux Linux computers. In both cases, *all communication between the client and the server must be through the socket API with real IP addresses, i.e.* ***you cannot use the 127 loopback IP addresses, e.g. 127.0.0.1****. Your computer must be connected to the Internet and you must be able to find the real IP address assigned to your computer. Use that real IP address for creating your sockets and for all communication between the client and the server.*

If you use Auburn University's tux Linux workstations you should use different tux Linux computers for the client and the server processes. The tux computers that are available for you use through remote access are tux050-tux065 and tux237-tux252. Altogether, there are 32 tux machines. If you are using the tux computers, send me email with all your group members and I'll assign to you open port numbers for the tux computers so that you can avoid interfering with each other's message transmission.

Implement the client program so that it will send a HTTP request to a simple Web server to retrieve a data file. The server will send HTTP response messages in 512-byte packets until the end of the file (see below). Add print statements in the client program to indicate that it is *sending* and *receiving* the packets correctly, i.e. print the messages that it sends and receives. The server program responds to clients' HTTP requests. The server constructs HTTP response messages by putting header lines before the object itself that is to be sent. The server reads the requested HTML file (an ASCII file, must be at least **64 Kbytes**), put them in a buffer and sends the content of the buffer to the Web client who made the request. The HTTP response messages are sent in 512-byte packets until the end of the file. At the end of the file, it transmits 1 byte (NULL character) that indicates the end of the file. It will then close the file. Add print statements in the server program to indicate that it is *receiving* and *sending* the packets correctly, i.e. print the messages that it receives and sends.

Run the modified UDP C/C++ client and UDP C/C++ server programs with the Selective Repeat protocol for reliable data transfer. Other software, such as the segmentation and re-assembly, error detection and gremlin functions must also function correctly. The C/C++ client and server program must execute on different tux Linux computers if you are using the COE tux Linux computers. If you are using your own computer, you can execute both the C/C++ client and server program on the same computer but using

different port numbers. Capture the execution trace of the programs. In Linux, use the `script` command to capture the trace of the execution of the UDP C/C++ client and UDP C/C++ server programs. The trace must contain information when data packets and ACK/NAK are sent or received, which packets are corrupted, and when packets are lost. Sequence numbers and other relevant information on the packets must be printed.

Print the content of the input file read by the server program and the output file received by the client program.

Submission

Submit your source codes and the script of the execution traces of the programs, the file that was sent by the server and the file that was received by the client. Submit these in Canvas on or before the due date.