COMPUTER SCIENCES AND SOFTWARE ENGINEERING
AUBURN UNVERSITY

COMP 4320

Introduction to Computer Networks

Spring 2017

Project 2

Implementation of a Reliable Web Service Using Go Back N (GBN) Protocol over the UDP Transport Service

Due: 11:55pm April 19, 2017

Objective

The purpose of this assignment is to implement a Reliable Web Service using Go Back N (GBN) Protocol over the UDP transport service. You will write the reliable Web client and server programs based on the (GBN) Protocol that will communicate over the College of Engineering LAN. You will also write a gremlin function that will simulate unreliable networks which will corrupt, lose *and delay* packets. You will also learn other important functions in computer networks: (1) implementation of segmentation and re-assembly of long messages, (2) detecting errors in the received packets, (3) guaranteeing reliable data transfer by recovering from failures, such as packet corruption, loss or delays, and (4) emulation of packet errors generation and detection.

Overview

Again, ***you must implement the reliable Web client and server programs using Java*** and they must execute correctly in the COE tux Linux computers. In this project you will implement a reliable web client application, a reliable web server application, the segmentation and re-assembly function, an error detection function and a gremlin function (that can corrupt, lose and delay packets with specified probabilities). The overview of these software components is shown in Figure 1 below.

The reliable Web client and server program must have the following features, including the Go Back N (GBN) Protocol to ensure that the packets and received reliably.

The Web client initiates the communication by sending an HTTP request to the Web server. The HTTP request message is supposed to be of the form:

```
GET TestFile.html HTTP/1.0
```

This outgoing HTTP request is not processed by the segmentation and re-assembly, error detection or the Gremlin function. The HTTP request is sent through the transport UDP datagram socket to the Web server. Modify the client program so that it will send a HTTP request to a simple Web server to retrieve a data file.

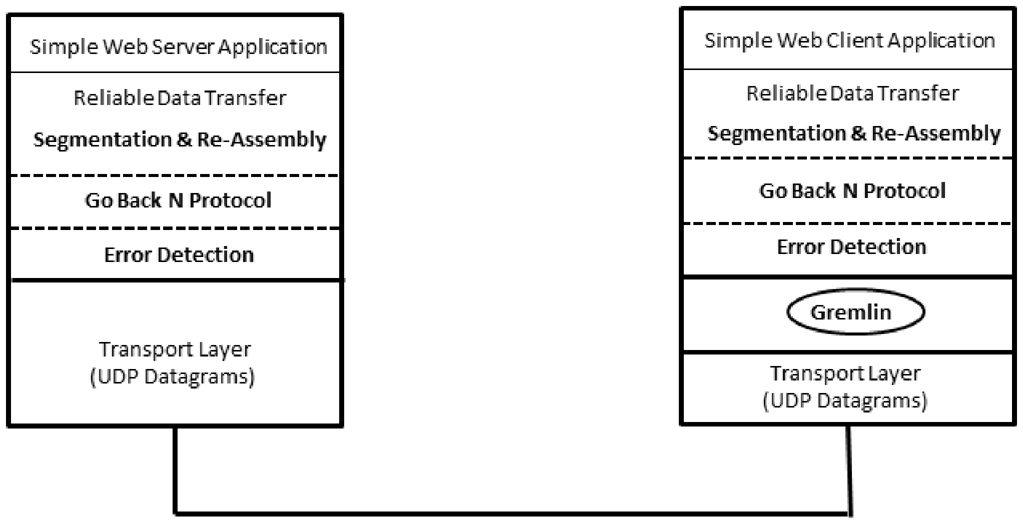
Simple Web Server Application
Reliable Data Transfer
Segmentation & Re-Assembly
Go Back N Protocol
Error Detection
Transport Layer (UDP Datagrams)
Simple Web Client Application
Reliable Data Transfer
Segmentation & Re-Assembly
Go Back N Protocol
Error Detection
Gremlin
Transport Layer (UDP Datagrams)

Figure 1. Overview of the Software Components

The file, `TestFile`, to be transferred is originally stored in the local secondary storage of the *server* host. After receiving the `GET TestFile.html` command from the client, the *server* first reads the file and put them in a buffer and sends the content of the buffer to the Web *client*. The test file is an ASCII file and must be at least *80 Kbytes* in size. 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. The packet must contain a header that contains information for error detection and other protocol information. You may design your own header fields that are of reasonable sizes.

The Web 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 4 header lines are supposed to be of the form:

```
HTTP/1.0 200 Document Follows\r\n
Content-Type: text/plain\r\n
Content-Length: xxx\r\n
\r\n
Data
```

(note: `\r` is a carriage return, `\n` is a line feed, `xxx` is the number of bytes in the HTML file being sent and data is the requested HTML file)

At the Web server host, the HTTP request is also not processed by those functions at the Web server host.

The server then reads the requested HTML file (an ASCII file, must be at least 80 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.

Since the requested file may be large, the Web 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. The packet must contain a header that contains information for error detection 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 *server* will use the Go Back N protocol with positive/negative acknowledgement and retransmission (PAR). You must use the window size, N = 32. The sequence number must be *modulo 64*. After it sends *32 packets*, it will wait for a positive or negative acknowledgement from the *client* before it sends the next window size of packets. The last packet will be padded with NULL character if the remaining data of the file is less than 512 bytes. 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.

The packet is 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.

Add print statements in the client program to print the sequence numbers, ACK/NAK (acknowledgement) and data to indicate that it is sending and receiving the packets correctly, i.e. print each packet (say, only the first 48 bytes of data) that it sends and receives.

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, lose some packets and delay some packets, depending on the corrupt, loss, and delay probabilities. This will emulate errors that may be generated by the network links and routers. The packet is then processed he packet is then processed by the Go Back N protocol. Then, the packet is processed by 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.

The Web *client* will receive data of the test file in 512-byte packets, i.e. the client will receive each 512-byte packet in a loop and writes them into a local file sequentially. After it receives a packet and verifies that it is correct, it will send an acknowledgement. Each packet is processed by the error detection function that will detect possibility of error based on the checksum. The packet is then processed by the segmentation and re-assembly function that re-assembles all the segments of the file from the packets received into the original file. When it receives a 1-byte message with a NULL character, then it knows that the last packet has been received and it closes the file.

Go Back N (GBN) Protocol

You are to design and implement a Go Back N protocol where the window size is 32 and the sequence number is modulo 64. Your Go Back N protocol will deal with errors, loss and delays in packets.

The Go Back N protocol follows the pipeline principle as follows. After the sender sends packet 0, the sender then sends 31 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 sender is still busy sending packets into the channel. Handling of packet 0 will then be done while the handling of packet 1 and subsequent packet is already underway. Thus, Go Back N pipelines the processing of packets before the completion of previous packet transmission to keep the channel busy.

In your Go Back N protocol, the receiver must handle packet errors, lost packets and delayed packets the following ways.

1. When the receiver receives a packet, it must use the coding method of your choice to check for checksum errors. The sender also uses the same coding method. If the packet is free of error and in the right order, it sends back an ACK to the sender. The sequence number of the ACK should be the next packet number that the receiver is expecting. For example, if it receives a packet 2, then it sends an ACK with sequence number 3.
2. When the receiver detects an error in a packet, it *must* send back a NAK to the sender. The sequence number should be the next packet that the receiver is expecting. For example, if the receiver has received packets up to and including packet 3 and it then receives a packet that contains an error, then it should send a NAK with sequence number 4. If the receiver detects an error in the packet, **it must print out in its output trace that the packet (with sequence number) has errors**. The receiver will then drop the packet and not pass it to the function that reassembles the data stream.
3. When the receiver receives a packet out of order (possibly due to a lost packet), it must ignore the out-of-order packets and *must* send back an ACK for the last packet that it received correctly. The function that receives the packet must check if it contains the expected sequence number. When it receives an out-of-order packet, **it then prints the sequence number in the output trace** and indicate if there are lost or delayed packets.

ACK and NAK are never lost or damaged.

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

1. When the sender receives an ACK, it must advance it window forward. For example, after the sender with send window [0-31] transmits packets 0 to 31, it receives an ACK with sequence number 8. Then the sender must advance its send window to [8-39] and sends the new packets 16-39.
2. When the sender receives a NAK with sequence number 8, it will retransmit packets [8-39]. A NAK with sequence number 8 indicates that the receiver has not received packet 8 correctly and has rejected all subsequent packets.
3. When the receiver does not send back either an ACK or NAK, then the sender will timeout on the earliest packet that has not been ACKed, e.g. packet 2 timeouts. It then retransmits all packets 2 to 31.

In your experiments, the following interesting interaction may occur. If the sender 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 sender can then avoid retransmitting those packets that are being ACKed. In this project, you may choose either implementation.

Another important implementation issue is how to set the timer for all the 32 packets that have outstanding ACKs. You need to set the timeout to a value less than 20 millisecond and based on the estimated round-trip time. Since there is only 1 timer in each computer, you need to implement the earliest timeout using the real timer and the remaining 31 timers implemented in software as follows. After first setting the timer for packet 0, and after transmitting packet 1, the GBN protocol should record the start time for packet 1 timeout relative to that of packet 0. When the ACK for packet 0 is received, then the real timer is set for packet 1 timeout. This can be repeated for all subsequent packets.

If packet 0 times out, then all subsequent packets must be retransmitted. A new timeout for packet 0 is set and all the software timers may also be reset at that time.

When the sender receives a NAK with sequence number *n*, it must stop the timer and immediately retransmit packet *n* and all subsequent packets. It also erases all software timers and resets the timer for packet n and re-creates the software timers for all subsequent packets.

For each packets transmitted, the timeout value must not be more than three 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 15 millisec.

When the simple file transfer application send a stream of input data to the GBN program, GBN will break the input data stream into data packets of 512 bytes. If the total length is not a multiple of 512 bytes, pad the last packet with 0s.

Each packet should also have a 16-bit checksum attached to the header of the packet, using any coding scheme of your choice. The sender computes the checksum value and attached it to the packet. The same coding scheme must be used by the receiver to check the checksum value.

The packet header should contain a sequence number, although your implementation can use a one-byte field in the packet structure.

Gremlin Function

Your program must allow the probabilities of damaged, lost and *delayed* packets to be input as arguments when the program is executed. For delayed packets, your program must also allow the user to input the delay time in milliseconds. These parameters for packet damage, lost and *delay* probabilities and the delay time are passed to your Gremlin function. You will implement a gremlin function to simulate *three* possible scenarios in the transmission line: (1) transmission error that cause packet corruption, (2) packet loss*, (3) packet delay* and (4) correct delivery. Corrupted packets and lost packets are processed as in Project 1. When the delayed packet probability and delay time are given, e.g. 0.3 and 4 millisecond, then three out of ten packets will be delayed for 4 milliseconds before being transmitted.

Error Detection Function

Error detection is implementation as in Project 1.

Testing

Run the Web UDP Java client and Web UDP Java server programs with the GBN protocol for reliable data transfer. Other software, such as segmentation and re-assembly, error detection and gremlin functions must also function correctly. The Web client and server programs must execute correctly on ***different*** tux Linux computers. Capture the execution trace of the programs. In your tests, the requested HTML file (an ASCII file) must be at least 80 Kbytes. In Linux, use the `script` command to capture the trace of the execution of the Web client and Web server programs. The trace must contain information when packets and ACK/NAK are sent or received, when 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 executions of the programs in Canvas on or before the due date. You will also demo your programs to the T.A. to verify that your programs execute correctly.