Programming Assignment 2

Data Communication Networks – Fall 2021
Building in Reliability
Assigned on Sep.20, 2021

Due Date: **October 20, 2021 by 11:59pm CST (local time in Starkville)**

Assignments by undergraduate students are to be *done in pairs or individually*. No larger groups are allowed, and you can only work on code with that particular partner. ***Graduate students*** ***must do the assignment individually.***

Programs will be reviewed using plagiarism-detection software.

Carefully read and follow the instructions for submitting your solution. You may lose points if you deviate from these instruction.

Again: Your code must execute correctly on Pluto. Give yourself time to test on Pluto.

1. Assignment Objective

In the last assignment, you created a file-transfer protocol from client to server that was unreliable. The goal of this assignment is to incorporate some amount of reliability into your application.

To this end, you will implement a version of the **Go−Back−N (GBN)** protocol which facilitates the transfer of a text file from one host to another across an unreliable network. While we will **not be testing that your implementation succeeds over a lossy channel in this assignment**, we will be doing so in the next one; therefore, you should aim to get all of the functionality working correctly now.

Your protocol will remain unidirectional, i.e., data will flow in one direction (from the client to the server) and the acknowledgements (ACKs) in the opposite direction. To implement this protocol, you will write two programs: a client and a server, with the specifications given below. ***All communication will be done over datagram (UDP) UNIX sockets***.

1.1 Using C++ versus Java

The assignment description sometimes differentiates between implementations in C++ and those in Java. Be sure to use the correct supplied files depending on your choice of programming language. Below, I have summarized some the challenges facing you that are specific to C++ and those that are specific to Java.

C++: The packet class already has the serialization and deserialization functions written and they are fairly straightforward if you put in some effort to understand them.

Java: While Java is "cleaner", the serialization/deserialization commands require a little bit more effort to use. To smooth the way, I've provided you with links and actual commands that you must use.

1.2 Makefiles for C++ and Java

I am not providing makefiles this time. You can easily modify the one you used for PA1 to work for PA2, and you should submit your makefile with your source files (see Section 6).

2. Packet Format in Java and C++

A packet class is provided. All packets exchanged between the client and the server have the general structure as given by the code underneath. This class is similar for both C++ and Java, but I illustrate the Java class below (for both languages, be sure to understand the packet code before using it):

```
public class packet {
     .........
     private int type;
     private int seqnum;
     private int length;
     private String data;
     .........
}
```

The `type` field indicates the type of the packet: it is set to 0 if it is an ACK, 1 if it is a data packet, 2 if it is an end-of-transmission (EOT) packet from server to client, and 3 if it is an EOT packet from client to server.

For data packets, `seqnum` is the modulo 8 sequence number of the packet. That is, sequence numbers have values in {0, 1, 2, 3, 4, 5, 6, 7}. The sequence number of the first packet should be zero.

For ACK packets, `seqnum` is the sequence number of the packet being acknowledged.

The length field specifies the number of characters carried in the data field. It should be in the range of 0 to 30, taking the largest chunk of data per packet sent (i.e. take 30 characters if you can, less if you have reached the end of the file). As with out previous assignment, you can assume you are working with an ASCII text file and that spaces, exclamation points, newline symbols, etc. each count as a character.

For ACK packets, length should be set to zero and these packets should carry no data (i.e. data field should be set to NULL).

You may **not** modify this class (we will use our own packet class when testing).

3. Client Program (`client`)

You should implement a client program, named `client`. Its command line input includes the following:

```
<serverName: host name of the server>,

<serverPort: UDP port number used by the server to receive data from
the client>,

<fileName: name of the file to be transferred>
```

in the given order.

Upon execution, the client program should be able to read data from the specified file and send it using the GBN protocol to the server. The window size should be set to N=7. Recall that this means that, even though we have 8 sequence numbers, we are only using 7 of these sequence numbers at any given time.

After all contents of the file have been transmitted successfully to the server and corresponding ACKs have been received, the client should send an EOT packet to the server. The EOT packet is in the same format (and it has a sequence number) as a regular data packet, except that its type field is set to 3, its length is set to zero, and the data is set to NULL.

The client can close its connection and exit only after it has received ACKs for all data packets it has sent and received an EOT from the server. To keep the project simple, you can assume that EOT packets are never lost in the network.

To ensure reliable transmission, your program should implement the GBN protocol as follows:

- If the client has a packet to send, it first checks to see if the window is full — that is, whether there are *N* outstanding, unacknowledged packets. If the window is not full, the packet is sent and the appropriate variables are updated. If the window is full, the client will try sending the packet later.
- A timer of 2 seconds is started if it was not done before. The client will use only a single timer that will be set for the oldest transmitted−but−not−yet−acknowledged packet.
- When the client receives an acknowledgement packet with sequence number *n*, the ACK will be taken to be a cumulative acknowledgement, indicating that all packets with a sequence number up to and including *n* have been correctly received at the server.
- If a timeout occurs, the client resends all packets that have been previously sent but that have not yet been acknowledged. If an ACK is received corresponding to an un-acked packet within the window, but there are still additional transmitted−but−yet−to−be−acknowledged packets, the timer is restarted with 2 seconds. If there are no outstanding packets, the timer is stopped.
- There are two generic ways to structure the client when the window is filled. The first way is the client fills the window and then waits to receive **all** acks before checking the window again to see whether it can send. The second way is that your client can fill the window and then obtain an ack, check to see the window has space and send again, then check for acks again, etc. -- this always keeps the window full. ***Implement the second way.***

3.2 Interrupting the Timer

The recvfrom() call will cause the client to block until a packet is received. This may cause problems for the correctness of your program unless you interrupt the recvfrom() call.

For example, imagine the case where the last packet from the client is sent and the client then calls recvfrom() to obtain the acknowledgement that should be returned from the server. However, assume that this last packet from the client is lost in transit. No acknowledgement will be transmitted back from the server. The client, who is now waiting to receive an ack, will block forever.

You must interrupt the blocking call to check whether the timer has expired and, if it has, resend all outstanding packets as specified by GBN. It is up to you to decide how to do this.

3.1 Output from Client

For grading purposes, your client program will generate two log files, named as *clientseqnum.log* and *clientack.log*.

Whenever a packet is sent, its sequence number should be recorded in *clientseqnum.log*. This includes the EOT packet sent by the client.

The file *clientack.log* should record the sequence numbers of all the ACK packets and the EOT packet (from the server) that the client receives during the entire period of transmission.

Each instance the program is executed, these files should be *overwritten* (not appended to).

The format for these two log files is one number per line. You must follow this format to avoid losing marks as the TA will be using these log files to grade the correctness of your assignment.

4. Server Program (`server`)

You should implement the server program, named `server`. Its command line input includes the following:

```
<serverPort: UDP port number used by the server to receive data from the client>,

<fileName: name of the file into which the received data is written>
```

in the given order.

When receiving packets sent by the client, it should execute the following:

- check the sequence number of the packet.
- if the sequence number is the one that it is expecting, it should send an ACK packet back to the client with the sequence number of the received packet (indicating it has received all packets up to and including this packet).
- In all other cases, it should discard the received packet and resends an ACK packet for the most recently received in-order packet.
- After the server has received all data packets and an EOT from the client, it should send an EOT packet with the type field set to 2, and then exit.

Of course, the server must also write the received data from the client to the file `fileName`.

4.1 Output from Server

In addition to the file containing the text transferred from the client, the server program is also required to generate a log file, named as *arrival.log*. The file *arrival.log* should record the sequence numbers of all the data packets and the EOT packet (from the client) that the server receives during the entire period of transmission.

Each instance the program is executed, this file should be *overwritten* (not appended to). Similarly, in each instance the program is executed, `fileName` should be overwritten.

The format for the log file is one number per line. You must follow the format to avoid losing marks as the TA will be using these log files to grade the correctness of your assignment.

Serialization/Deserialization of Packets

5.1 Students using Java

Serialization and deserialization should be used to transmit and receive packets, respectively. Learn about the following commands:

```
ByteArrayOutputStream oSt = new ByteArrayOutputStream();
ObjectOutputStream ooSt = new ObjectOutputStream(oSt);
ooSt.writeObject(pkt);
ooSt.flush();
byte[] sendBuf = new byte[30];
sendBuf = oSt.toByteArray();
```

and the similar commands for deserialization. **You MUST do serialization this way.**

The following link seems like a good start (or read about it anywhere else you would like):

`http://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayOutputStream.html`

`and you can see examples here:`

`http://stackoverflow.com/questions/17940423/send-object-over-udp-in-java`

`http://stackoverflow.com/questions/3997459/send-and-receive-serialize-object-on-udp-in-java`

5.1 Students using C++

Serialization and deserialization functions have been provided to you in the packet class. Other methods of serializing/deserializing exist for C++, but they are complicated and they will not be used for this assignment.

I have provided example serialization and deserialization code that will help you understand how this works in the assignment.

5.2 How to Call Executables

For C++, you should call your executables in two separate terminals with the commands:

```
./server 6000 output.txt
./client localhost 6000 file.txt
```

Note that you should call server first, then the client. Of course, you may change the parameter values for ports etc. — in fact, I suggest you choose random port values so your don't conflict with other students while testing.

For Java, you should call your executables similarly. For example:

```
java server 6000 output.txt
java client localhost 6000 file.txt
```

Example executions for both languages are included in the files provided. My examples include additional output that is helpful. Your code need not do this; if you do decide to provide output to the screen/terminal, please be economical (do not spam).

6. Your Submitted Solution

6.1 Due Date

This assignment is due on Oct. 20, 2021 by 11:59pm CST (local time in Starkville).

6.2 Hand in Instructions

Deviating from these submission rules will lead to a **penalty of 5%** on the grade for PA2.

(i) Submit your files in a single compressed file (either .zip or .tar) using Canvas. This compressed file must be titled (do not include the square brackets):

[your NetID]-PA2.tar or [your NetID]-PA2.zip

for example, I would title mine: **my325-PA2.tar** or **my325-PA2.zip**

If you are working with a partner, put a hyphen between the NetIDs like so:

[NetID1-NetID2]-PA2.tar

(ii) Your source code ***MUST have the names of the author(s)*** on it! If you are working as a pair, ***only one student*** should submit the code through Canvas.

(iii) You must hand in the following TWO things:

- **(1) Source code for all files; that is your .cpp, .h, or .java files (including the packet files if you're using C++). Use proper names, NOT "Client.cpp" instead of client.cpp, for example.**

- **(2) Your makefile. Remember that your code must compile and link cleanly by typing ``make".**

Your implementation will be tested on the Pluto system and so it must run smoothly in this environment! See the syllabus for penalties for late assignments.

A sample file will be provided online for you to use for testing your client/server programs. However, it is your responsibility to test your code thoroughly and ensure it conforms to the requirements.

6.3 Documentation

There is no external documentation required for this assignment, you are expected to have a reasonable amount of internal code documentation.

7. Suggestions for Getting Started and Helpful Hints

I recommend the following general 2-step approach for implementation:

1. Forget about GBN for now. Get the client and server programs running using serialization and deserializing such that they can exchange data using the two different ports assigned. You may develop this locally, but remember to test ahead of time that this works on Pluto.

2. Review the details of GBN as given in the lecture slides and in our textbook (noting that we using the variation where the server sends an ACK after reception of each packet from the client). Focus on getting your client and server to run the GBN protocol. You may develop this locally, but remember to test this on Pluto.

Remember to use port numbers at 1024 or above to avoid reserved port numbers.

For C++ users, I highly recommend using the `time()` call for setting your timer. Personally, I prefer using `chrono`, but Pluto does not like it. For `time()`, take a look at:

https://www.tutorialspoint.com/c_standard_library/c_function_difftime.htm

Also, instead of using to_string to convert sequence numbers to strings, I recommend using the `snprintf()` function. Again, Pluto kicks up a fuss.

Finally, get started early. You have a lot time to complete this assignment, but ***do not*** leave this to the last minute (or even the last few days). I will tend to ignore last-minute requests for help, especially in regard to debugging code, since there is more than an adequate amount of time to complete this assignment.

To make your life easier, here is a complete list of the include statements for C++ users. If you use these and only these include statements, then I *think* Pluto will not fuss over your function calls. **However, you should still give yourself lots of time to test on Pluto.**

On the client side:

```
#include <stdlib.h>
#include <cstring>
#include <cstdlib>
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <iostream>
#include <fstream>
#include <arpa/inet.h>
#include "packet.h"
#include "client.h"
#include <math.h>
#include <time.h>
```

On the server side:

```
#include <stdlib.h>
#include <cstring>
#include <cstdlib>
#include<iostream>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <time.h>
#include <string.h>
#include <fstream>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include "packet.h"
#include <unistd.h>
```

Acknowledgements: I am grateful to Dr. Raouf Boutaba who brought this assignment topic to my attention.