Flow control and loss recovery

Introduction

Flow control is a concept in networking that controls that the speed of a sender does not exceed a receiver's capacity to receive data as quickly as it arrives. As a general concept, it can be found on several layers of the network stack, on layer 2 or any higher layer. On layer 2, for example, flow control ensures that a node does not send faster than its directly attached neighboring node can receive, on layer 4, flow control ensures that a sending end system does not send faster than a receiving end system can receive and deliver to the application layer. Not all protocols include flow control, layer N must implement it if it offers flow controlled service to layer N+1.

Loss recovery is another concept that a protocol on layer 2 or any higher layer may implement. The term refers to the loss of packets which a layer N must correct if it offers reliable service to the higher layer N+1. To do this, the protocol a layer N must implement a repair mechanism, and in many cases, this is done by detecting the loss and retransmitting the lost packet.

Flow control and loss recovery are usually very tightly integrated with each other. In many textbooks, retransmission for loss recovery is actually described as a task of flow control.

On layer 4 of the Internet, TCP offers a service with both flow control and loss recovery, whereas UDP offers a service without either flow control or loss recovery.

The Task

There are applications that are not satisfied with the services of either TCP or UDP. In many cases, they are looking for reliable, flow-controlled service but they require that packets are delivered to the receiving application exactly as the sending application sends them. For this reason, they add a new layer that provides the missing services on top of either TCP or UDP. One example is QUIC, which is used for Google Chrome, and which implements a "better" transport layer service on top of the transport layer service UDP.

In this assignment, you are going to write a client-server application that compares images that are stored on two end systems.

- The client reads several images from disk, then opens a connection to the server and sends the images to it. Each image is transferred in a single packet. The provided images are small enough to fit in a single ethernet frame.
- When the server receives a packet from the client, it extracts the image data from the packet's payload and compares it with all of the images that are stored locally on the server. It writes the name of the file that matches the image to a local file. This file can then be used to check that your implementation works as expected.

Obviously, because the server has to do a lot of work for every image, it will not be ready for receiving new packets all the time. Flow control is required to keep the client from sending too quickly.
Also, you must handle loss on the path from the client to the server. The path will be lossy because you must use our function `send_packet()` as a replacement for the C function `send()`. The loss probability should be set by calling `set_loss_probability( )`.

You must write a layer on top of UDP that provides flow control and loss recovery. In this layer, you will implement the Sliding Window algorithm with a maximum window size of 7 packets.

The packets

The protocol format of your transport layer must be the following:

- (int) total length of the packet including the payload in bytes
- (unsigned char) sequence number of this packet
- (unsigned char) sequence number of the last received packet (ACK)
- (unsigned char) flags
    - 0x1 : 1 if this packet contains data
    - 0x2 : 1 if this packet contains an ACK
    - 0x4 : 1 if this packet closes the connection
    - All other bits: 0
- (unsigned char) unused, must always contain the value 0x7f
- (bytes) payload

There should be no data in-between the header variables, and between the header and the payload.
A packet with the flag set to 0x1 is a data packet. It should have a payload.
A packet with the flag set to 0x2 is an ACK. It must not have a payload.
A packet with the flag set to 0x4 is a termination packet. It must not have a payload. It is used by the client to terminate the server, and it is not ACKed.
Only one of the flags 0x1, 0x2 or 0.4 can be 1 in a valid packet.

The application sends payloads containing the following:

- If the application sends a file
    - (int) unique number of the request (this is a unique number of your choice that is intended for debugging, not a copy of the sequence number)
    - (int) length of the filename in bytes (including final 0)
    - Filename (including final 0)
    - Bytes of the images
- ACK packets and termination packets do not carry the payload.

The filename should not include directories. You can use the function `basename`.

The Client

The client should accept the following command-line arguments:

- The IP-address or hostname of the machine where the server application runs.
- The port number at which the server application should receive packets.
- A filename that contains a list of image filenames. En eksempelfil er gitt.
- A drop percentage. This should be in the range 0 to 20.

You cannot assume that any of the filenames (both the one given and those in the given file) are valid, and must therefore check that they exist. If a filename is invalid the client should print out an appropriate message and terminate. You can assume that the other command-line arguments are correct.

Payload containing one complete image-file should be put into one packet and sent to the server. The first packet should have the sequence number 0 (this is an assumption in the server). The sequence number is incremented by one; The second packet has sequence number 1, the third packet has sequence number 2, and so forth.

Packet loss is something that cannot be reliably detected. The best we can do is to infer packet loss from absence of an acknowledgement. The client should deem a packet lost if it has not received an acknowledgement for the packet within 5 seconds, it should use a timeout to detect this. For simplicity, you can track only the time of the oldest packet (lowest sequence number not received acknowledgement for).

If a packet is lost, it has to be resent. It is the job of the client to do so. The client must keep a linked list of sent, but not acknowledged, packets for this purpose. You must allocate the packets in this list on the heap. Each entry in the linked list should contain enough information to be able to resend and remove packets once a timeout or reception of acknowledgement occurs.

Whenever a timeout occurs all packets in the current window should be resent and the 5-second clock should be reset.

Whenever an acknowledgement is received its acknowledgement number should be compared to the sequence number of the oldest packet. If they match, the acknowledgement is accepted; The packet can be removed from the list and the window can be advanced by one packet.
If they do not match, the acknowledgement is rejected; The client does nothing.
Acknowledgements are not cumulative, as they are in TCP, in this task. This is inefficient, but it simplifies the implementation.

When the client has sent its image-files it should send a termination packet to the server. This packet does not have to be sent reliably. Our lossy function (`send_packet`) does not drop termination packets. In the unlikely case that UDP drops the termination packet, the server has to be terminated manually.

The Server

The server should accept the following command-line arguments:

- The port number at which the server application should receive packets
- A directory name containing image-files.
- Filename for printing matches

The server should listen to incoming packages on the specified port. When the server receives a packet it has to check that the sequence number of that packet matches the sequence number that is expected. The first packet is assumed to have sequence number 0.

Whenever a packet that carries the expected sequence number is received, the server should do the following. Perform the steps of the Go-back-N protocol for receiving packets and pass the payload to the application layer for processing.
The payload contains a filename and an image. The image should be compared to the content of each of the files in the given directory. If an identical file is found, the server should append a string like the following to the file for printing matches:

“<Payload’s filename> <Server’s filename>\n”

If no match is found, the server should append a string like the following:

“<Payload’s filename> UNKNOWN\n”

You can compare images in two ways:

- Either you use the function `Image_create` to create image structures of type `struct Image` and then compare the images using `Image_compare`. Do not forget to release the memory with `Image_free`. We hand out these functions in the files pgmread.c and pgmread.h.
- Or you compare the images byte-for-byte yourself.

The server should terminate when it receives a termination packet.

Memory management

We expect that there are no memory leaks under a normal run of the applications. A normal run is a run where command-line arguments and filenames are all valid, and UDP has no packet loss. All dynamically allocated memory must be explicitly deallocated. All opened file-descriptors must be explicitly closed.

We expect that valgrind doesn’t display any warnings under a normal run of the applications. If you are convinced that a warning is false, which rarely happens, you must state this in the code or in a separate document.

Failing to meet these expectations will affect the evaluation of your assignment negatively.

Submission

You must submit all your code in a single TAR, TAR.GZ or ZIP archive.
If your file is called `<candidatenumber>.tar` or `<candidatenumber>.tgz` or `<candidatenumber>.tar.gz`, we will use the command `tar` on login.ifi.uio.no to extract it. If your file is called `<candidatenumber>.zip`, we will use the command `unzip` on login.ifi.uio.no to extract it. Make sure that this works before uploading the file.

Your archive must contain Makefile which will have at least these options
`make` - compiles both your programs resulting in executable binaries "client" and "server"
`make all` - does the same as make without any parameter
`make clean` - deletes the executables and any temporary files (eg. *.o)

About the evaluation

The home exam requires an understanding of various functions that operating systems and network protocols must offer to applications, in particular memory management, file handling and networking. It is a good idea to solve parts of this home exam in separate programs first and combine them into a client and a server later.

We propose to address the sub-tasks as follows:

- Memory management
    a. Do not forget to free all of the memory that you have allocated. In operating systems and networks, you are always personally responsible for memory handling. Knowing exactly which memory you are using, how long you are using it and when you can free it, is one of the most important tasks in operating systems and networks. We use C in this course to make sure that you understand how difficult that task is and how you can solve it. Doing this right (with help from `valgrind`) is very important for the home exam.
    b. We do not expect that your server frees all memory until you implement Networking sub-task (g) - but we expect that you do not forget memory that you have allocated.
    c. We do not expect that your client and server free all memory in case of crashes or when you have to press Ctrl-C because of another problem.
- Networking
    a. Sending UDP packets from a client to server using the functions sendto and recvfrom, and make sure that the client frees all memory before terminating.
    b. Implement the Packet format that is required by the assignment.

c. Add the Stop-and-wait functionality with just 2 sequence numbers (you can implement Go-back-N directly, but this is safer).
d. Replace the function sendto with our function send_packet, use `select` to implement timeouts and set the loss probability on the client side to a non-zero percentage. Make sure that stop-and-wait works.
e. Replace Stop-and-wait with Go-back-N with a maximum window size of 7 packets.
f. Make sure that the send window is implemented as a link list of structs that are allocated in heap memory (probably already done in (e)).
g. Implement the “close connection” functionality, and make sure that the server frees all memory correctly before terminating.

- Files
  a. Read a list of filenames from a file.
  b. Read a complete file into heap memory.
  c. Check if two buffers that are stored in heap memory are identical.
  d. Write a file that contains two filenames in each line if the buffers are identical, or one filename and the word UNKNOWN if the buffers are not identical.
  e. Discover the content of a directory (using `readdir`), determine which of those are files (using `lstat` or `fstat`), and use that list as a list of filenames on the server.
- Creating a complete client and server.
  You must solve several Networking sub-tasks and several Files sub-tasks to pass the home exam. A sloppy solution for many sub-tasks is equally valuable as a very good solution for a few sub-tasks.
  a. It may be a good idea to combine Networking (a)+(b) and Files (a)+(b)+(c) for a very simple first version of your client and server.
  b. It is a very good idea to create a working client-server solution that combines Networking (a)+(b)+(c)+(d) and Files (a)+(b)+(c) before solving more sub-tasks.
  c. Extend this by solving the remaining sub-tasks.

Useful and relevant functions

- For sockets
  - `socket`
  - `bind`
  - `sendto`
  - `recvfrom`
  - `select`
  - `perror`
  - `getaddrinfo`
- For files
  - `fread`
  - `fwrite`

    - `fopen`
    - `fclose`
    - `basename`
    - `lstat` or `fstat`
- For directories
    - `opendir`
    - `readdir`
    - `closedir`

Handouts

https://www.uio.no/studier/emner/matnat/ifi/IN2140/v20/undervisningsmaterial/h1-2020-handout-v1.tgz