Multiplexing and Loss Recovery

Introduction

An important task of the transport layer is multiplexing. Whereas the network layer is responsible for delivering packets to an end system, the transport layer is responsible for delivering these packets to the correct process on that end system. This is, however, a simplification. In many cases, a single process must offer many connections, and the transport layer must provide multiplexing between all of these connections inside the same process. A very typical example of this is a web server, which must serve files to several web browsers at the same time.

If the transport layer offers reliable services to the next higher layer, it may also have to implement reliability if the underlying network does not. In the case of the Internet, TCP provides multiplexing and reliability on top of the network layer IP.

In this task, you will be using the UDP protocol to emulate a packet-oriented, unconnected, unreliable network layer protocol, and you create your own transport layer that implements multiplexing for several concurrent connections to the same process on a single server.

You will implement two applications: a client and a server. These applications will implement both transport layer functionality and application layer functionality.

The Task

This task is inspired by a protocol that is rarely used today: the file service protocol FSP. It transfers large files (such as an operating system kernel) over UDP and is sometimes used to deliver large files to many blade computers in a compute cluster when the cluster is powered up. These blade computers do not have any permanent storage of their own; FSP is used to retrieve their operating systems during boot. FSP was developed because it is a very light-weight protocol on top of UDP, but its role is very limited because it cannot provide reliable data transfer. If packets get lost, FSP must restart the transfer.

In this task, you will create a new transport layer, RDP, the Reliable Datagram Protocol, which provides multiplexing and reliability, a NewFSP client that retrieves files from a server, and a NewFSP server that uses RDP to deliver large files to several NewFSP clients reliably.

RDP

Unfortunately, you cannot implement your RDP directly above IP because that requires root privileges. Instead, you will use UDP to emulate a connectionless, unreliable network layer protocol underneath the transport protocol RDP. Due to the limitations of IFI's installation, you must support IPv4 addresses. You may also support IPv6 addresses.

Your transport protocol RDP, residing on top of this "network protocol", will provide multiplexing and reliability. Your RDP implementation will use one single UDP port to receive packets from all clients, and it will answer them over this port as well.

Reliability: Your RDP protocol is a connection-oriented protocol, because providing reliability is easier with connection-oriented protocols than connectionless ones. Your application is supposed to call the function `rdp_accept()` to check if a new connection request has arrived and accept it if that is the case. `rdp_accept()` returns a `NULL`-pointer if that is not the case and a pointer to **a dynamically allocated data structure** that represents an RDP connection if a connection is accepted. RDP is not as complex as TCP; it uses a simple stop-and-wait protocol to achieve reliable data transfer. Every packet that is sent by an RDP sender must be acknowledged by the receiver before it can send the next packet (stop-and-wait). If no acknowledgement is received within 100 milliseconds, RDP retransmits the packet. The RDP sender does not accept another packet from the application until the previous packet has been delivered (that means it returns 0 when the application calls `rdp_write()`).

Multiplexing: The multiplexing that is required will be implemented by you in your RDP protocol. You will use connection IDs that identify the correct connection on the server and on the client. When a client establishes a connection, it chooses its own connection ID randomly, and includes a connection ID 0 for the server when requesting a connection. If another client with the same connection ID is already connected to the server, it must reject that new connect request by sending a packet with `flags==0x20`.

NewFSP

You will implement two programs: a NewFSP server and a NewFSP client.

The NewFSP client will use RDP to connect to a NewFSP server and retrieve a file. The client does not know the name of this file; it will simply establish an RDP connection to the server and receive packets. The NewFSP client will store these packets in a local file called `kernel-file-<random number>`. When the NewFSP server indicates that the file transfer is complete by sending an empty packet, the NewFSP client will close the connection.

The NewFSP server will use RDP to receive connect requests from NewFSP clients and serve them. The NewFSP server is a single-threaded process that can serve several NewFSP clients at the same time. It has one main loop, in which it

1. checks if a new RDP connect request has arrived,
2. tries to deliver the next packet (or the final empty packet) to each connected RDP client,
3. checks if an RDP connection has been closed.
4. If nothing has happened in any of these three cases (no new connection, no connection was closed, no packet could be sent to any connection), you must wait for a period. There are two implementation options:
    a. Waits for one second before trying again.
    b. Use `select()` to wait until an event occurs.

When a client has connected successfully, the NewFSP server will transfer a large file to that client. When it has transmitted the entire file, it sends an empty packet to indicate that the transfer is complete.

It sends the same large file to all of these clients. You can choose if the NewFSP server reads the file from disk several times or keeps it in memory. It uses RDP's multiplexing provided to achieve this, but it means that it must be able to accept new connections from a NewFSP client in between sending packets to other, already connected NewFSP clients.

The Packets

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

Data Type	Name (suggestive)	Description
unsigned char	`flags`	Flags defining different types of packets
• If `flags==0x01`, then this packet is a connect request. • If `flags==0x02`, then this packet is a connection termination. • If `flags==0x04`, then this packet contains data. • If `flags==0x08`, then this packet is an ACK. • If `flags==0x10`, then this packet accepts a connect request. • If `flags==0x20`, then this packet refuses a connect request.		
unsigned char	`pktseq`	Sequence number of packet
unsigned char	`ackseq`	Sequence number ACK-ed by packet
unsigned char	`unassigned`	Unused, that is, always 0
int	`senderid`	Sender´s connection ID in network byte order

int	`recvid`	Receiver´s connection ID in network byte order
int	`metadata`	Integer value in network byte order whose interpretation depends on the value of flags
• If `flags==0x04`, then `metadata` gives the total length of the packet, including the payload, in bytes. • If `flags==0x20`, then `metadata` gives an integer value that indicates the reason for refusing the connect request. You choose the meaning of these error codes.		
bytes	`payload`	Payload, that is, the number of bytes indicated by the previous integer value, but never more than 1000 bytes. Payload is only included in the packet if `flags==0x04`.

Notes:

- In every packet, only one of the bits in `flags` can be set to 1. If you receive a packet where this is not true, discard it.
- A packet can only have payload if `flags==0x04`.
- If `flags==0x01`, the server's connection ID is unknown and must contain 0.
- The first data packet in a connection has the sequence number 0.
- An ACK packet contains the sequence number of the last packet that it has received (unlike TCP).

The application sends datagrams that have a payload size between 0 and 1000 bytes (>=0 and <1000). The application relies entirely on RDP to guarantee that all bytes arrive in the right order and that none are lost. The NewFSP application interprets that a data packet with no payload indicates the end of a file transfer—for RDP an empty data packet is just a normal data packet.

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 UDP port number used by the server.
- The loss probability `prob` given as a floating point value between 0 and 1, where 0 is no loss and 1 is always loss.

The client calls the function `set_loss_probability(prob)` at the start of the program.

The client attempts to connect to the server. If the server does not answer the connect request in one second, the client prints an error message and terminates. When the client has connected successfully, it prints the connection IDs of both client and server on the screen.

When the client has successfully received and stored the complete file transferred by the server and the RDP connection is closed, it prints the name of the file that it has written (i.e. `kernel-file-<random number>`) to the screen and terminates.

The Server

The server should accept the following command-line arguments:

- The UDP port number at which the server application should receive packets
- The filename of the file that it sends to all connecting NewFSP clients
- The number N of files that the server should serve before it terminates itself.
- The loss probability `prob` given as a floating point value between 0 and 1, where 0 is no loss and 1 is always loss.

The server calls the function `set_loss_probability(prob)` at the start of the program.

Whenever a client connects to the server, the server responds with a packet with `flags=0x10` and prints the connection IDs of both the server and the client to the screen: `CONNECTED <client-ID> <server-ID>` . However, if the server is already serving the Nth file or it has already served it, it does not accept the connection, it responds with a packet with `flags==0x20` and prints: `NOT CONNECTED <client-ID> <server-ID>`

Whenever a client disconnects from the server, the server prints the connection IDs of both the server and the client to the screen: `DISCONNECTED <client-ID> <server-ID>`

The server should terminate when it has successfully delivered N files.

Packet Loss

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 `sendto()`. It takes the same arguments in the same order.

The provided files `send_packet.c` and `send_packet.h` compile on IFI's login machines without any `-std` compilation flags. It does also compile with `-std=gnu11` (the default), even with `-Wall -Wextra`. Other C standards, in particular `-std=c11` will result in a warning, and the function may not work correctly.

The packet loss probability is given as a number between 0 and 1. If you implement the packet format as requested in this assignment, only packets that contain data (`flags==0x04`) or ACKs (`flags==0x08`) will be dropped. This is for your convenience, but you can remove that

condition if you also want to experience the loss of connection requests and connection terminations.

Both your client and server must set the loss probability by calling the function `void set_loss_probability( float x )` that is provided in `send_packet.c`.

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. All dynamically allocated memory must be explicitly deallocated. All opened file descriptors must be explicitly closed.

We expect that Valgrind does not 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. It is also prudent to download and test the code after delivering it.

Your archive must contain a `Makefile`, which will have at least these options:

- `make` - compiles both your programs into executable binaries `client` and `server`
- `make all` - does the same as `make` without any parameter
- `make clean` - deletes the executables and any temporary files (e.g. `*.o`)

About the Evaluation

The home exam will be evaluated on the computers of the `login.ifi.uio.no` pool. The programs must compile and run on these computers. We will attempt to run NewFSP client and server on different computers in the pool.

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 subtasks 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 this 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 subtask f, but we do expect that you remember the 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. Send UDP packets between client and server using the functions `sendto()` and `recvfrom()`. Make sure that both client and server free all memory before terminating.
    b. Implement the packet format that is required by the assignment.
    c. Replace the function `sendto()` with our function `send_packet()` and set the loss probability on the client side to a value between 0 and 1.
    d. Make sure that packet retransmission works.
    e. Make sure that multiplexing works by connection with several clients at the same time. You must use the provided `send_packet()` function with some packet loss probability. File transfers may be too fast to start several concurrent NewFSP client if you do not do that.
    f. Implement the “close connection” functionality and make sure that the server frees all memory correctly before terminating.
- File on the server
    a. Check if the file indicated on the command line of the NewFSP server exists. Terminate with an error message if the file is not found.
    b. You may open the file for sending once per connection from the NewFSP client, but you may also read it once and keep it in memory until the NewFSP server terminates.
- File on the client
    a. Check that the file `kernel-file-<random number>` does not exist and that you can open it.
    b. Write the packets that the client receives into this file.

- Creating a complete client and server
  You must solve several networking subtasks and several files subtasks to pass the home exam. A sloppy solution for many subtasks is equally valuable as a very good solution for a few subtasks.

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()`