# COMS3200/7201 Assignment 2

Due: Wednesday 1st May 2019, 20:00

100 marks total (scaled to 15% of final grade)

## Part A (25 marks)

Part A will involve a few short questions about the QUIC experimental transport layer protocol. To answer these questions you will have to do some research about this protocol, which will include reading sections of the IETF draft for this protocol (`https://datatracker.ietf.org/doc/draft-ietf-quic-transport/`). You should make sure to read v19 of the draft as there have been changes to the protocol between drafts. Keep in mind that this draft goes into more technical detail than is required for these questions.

You should make sure to read the following sections of the draft, although you may have to do further reading to answer all the questions.

- Introduction (section 1)
- Frame types and formats (sections 12 and 19)
- Variable length encoding (section 16)
- Error handling (section 11)

You should submit your answers via the quiz on Blackboard. The answer to each question should be as concise as possible (while still fully answering the question). All questions will be manually marked by a tutor.

1. *Identify* and *describe* three advantages of QUIC over TCP. (6 marks)
2. *Define* a stream, a connection, and a frame in the context of QUIC. (3 marks)
3. Is it possible for two different QUIC packets over a single connection to have a packet number of $10^{20}$? *Explain* why/why not. (2 marks)
4. According to the draft, what sort of error could result in a RESET_STREAM frame being sent as opposed to a CONNECTION_CLOSE frame? (2 marks)
5. If an endpoint receives an invalid CONNECTION_CLOSE frame, how should it signal this error to its peer? (1 mark)

Consider the following hexadecimal sequences representing one or more QUIC frames (these are explained in section 19 of the draft). For each sequence, state what QUIC frame(s) the sequence represents (according to the draft) and explain the purpose of the frame type(s). For example, the hex sequence 0x01 represents a PING frame, which is used to check reachability of a peer.

6. 0x00000000 (3 marks)
7. 0x05010100 (4 marks)
8. 0x106000 (4 marks)

# Part B (25 marks)

This question requires you to analyse the Wireshark file `file-download.pcapng`. This packet capture shows a client downloading a large file from a server, using TCP as a transport layer protocol.

You will need to read some of the relevant IETF RFCs to understand some of the following questions. In particular:

- RFC793 is the base TCP spec
- RFC2018 describes the TCP SACK option
- RFC7323 describes the Window Scale option

Answer each of the following questions in the associated quiz on blackboard, following the specified instructions. All questions will be automatically marked. Recall that Wireshark displays TCP sequence numbers as a value relative to the first sequence number. You will need to disable this to answer any questions that ask for a *raw* sequence number. It is highly recommended that you turn of TCP reassembly to understand the order of packet transmission.

1. What is the raw TCP sequence number of...
   (a) The packet initiating the TCP connection? (1 mark)
   (b) The acknowledgement from the server for the above packet? (1 mark)

Frame 10 shows a HTTP GET request from the client to the server. Questions 2 and 3 focus on this frame.

2. What are the values of the 8 TCP flags (CWR to FIN) as a single 8-digit binary number? For example, if FIN was set and the others unset, you would write 00000001. (2 marks)
3. What is the length of the TCP header (in bytes)? (1 mark)
4. The server starts its response with a burst of packets. Starting with frame 12, how many frames are sent before the sender stops and waits for an acknowledgement? (HINT: there will be at least a 20ms delay between receiving consecutive packets when this occurs). (2 marks)

The following questions relate to window scaling. You should read RFC7323 before attempting these questions.

5. What is the initial value of the window scale shift count indicated by...
   (a) The client? (1 mark)
   (b) The server? (1 mark)
6. Suppose the client sent a packet to the server (across the same TCP connection as the one in the Wireshark trace) with an advertised window size of 101. What is the true window size in bytes? (3 marks)
7. In what frame does the client first indicate a change to the window size (*after* the server begins transmitting the file), and how much does it increase/decrease by when this occurs? (HINT: you may want to generate a graph of the window size to assist in finding this) (4 marks)

Around packet 1037, some frames are lost. Questions 8-10 focus on these lost frames.

8. How many bytes are lost? (4 marks)
9. In which frame is the first lost frame retransmitted? (2 marks)
10. In which frame does the receiver indicate that all missing frames have been received? (3 marks)

# Part C (50 marks)

The RUSH protocol (Reliable UDP Substitute for HTTP) is a HTTP-like stop-and-wait protocol that uses UDP in conjunction with the RDT rules. You have recently been hired by the multinational tech giant COMS3200 Inc, who have identified your deep knowledge in the field of transport-layer protocols. The CEO of COMS3200 Inc, Dan Kim, has asked you to develop a network server capable of sending RUSH messages to a client. It is expected that the RUSH protocol is able to handle packet corruption and loss according to the RDT rules. Your server program must be written in Python, Java, C, or C++.

| **ASIDE:** The RUSH protocol is not a real networking protocol and has been created purely for this assignment |
|---|

## Program Invocation

Your program should be able to be invoked from a UNIX command line as follows. It is expected that any Python programs can run with version 3.6, and any Java programs can run with version 8.

### Python

```
python3 assign2.py
```

### C/C++

```
make
./assign2
```

### Java

```
make
java Assign2
```

## RUSH Packet Structure

A RUSH packet can be expressed as the following structure (`||` is concatenation):

```
IP-header || UDP-header || RUSH-header || payload
```

The data segment of the packet is a string of ASCII plaintext. Single RUSH packets must be no longer than 1500 bytes, including the IP and UDP headers (i.e. the maximum length of the data section is 1466 bytes). Packets that are smaller than 1500 bytes should be padded with 0 bits up to that size. The following table describes the header structure of an RUSH packet:

| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Sequence Number | | | | | | | | | | | | | | | |
| 16 | Acknowledgement Number | | | | | | | | | | | | | | | |
| 32 | Flags | | | | | | | | Reserved (should be 0) | | | | | | | |

The following sections describe each header in this packet further.

### Sequence and Acknowledgement Numbers

Sequence numbers are independently maintained by the client and server. The first packet sent by either endpoint should have a sequence number of 1, and subsequent packets should have a sequence number of 1 higher than the previous packet (note that unlike TCP, RUSH sequence numbers are based on the number of packets as opposed to the number of bytes). When the ACK flag is set, the acknowledgement number should contain the sequence number of the packet that is being acknowledged. When a packet is retransmitted, it should use the *original* sequence number of the packet being retransmitted. Any packet that isn't a retransmission (including NAKs) should increment the sequence number.

### Flags

The Flags section of the header is broken down into the following:

| Bit | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | ACK | NAK | GET | DAT | FIN |

The purpose of these flags is described in the example below.

## RUSH Example

The following situation describes a simple RUSH communication session. Square brackets denote the flags that are set in each step (for example [FIN/ACK] denotes the FIN and ACK flags having the value 1 and the rest having the value 0). Note that RUSH, unlike TCP, is not connection-oriented. There is no handshake to initialise the connection, but there is one to close the connection.

1. The client sends a request packet to the server [GET]
    - The sequence number of this packet will always be 1
    - The data section of this packet will contain the name of a resource (eg. `file.txt`)
2. The server transmits the requested resource to the client over (possibly) multiple packets [DAT]
    - The first packet should have a sequence number of 1
3. The client acknowledges having received each data packet [DAT/ACK]
    - The acknowledgement number of this packet should be the sequence number of the packet being acknowledged
4. After receiving the last acknowledgement, the server signals the end of the connection [FIN]
5. The client acknowledges the connection termination [FIN/ACK]
6. The server acknowledges the client's acknowledgement and terminates the connection [FIN/ACK]

## Your Task

### Basic Server Functionality (10 marks)

To receive marks in this section you need to have a program that is able to:

- Listen on an unused port for a client's message
- Successfully close the connection

When invoked, your program should choose an unused localhost port and listen on that port. It should output that port as a raw base-10 integer to stdout. For example, if Python was used and port 54321 was selected, your program invocation would look like this:

```
python3 assign2.py
54321
```

Any lines in stdout after the port number can be used for debugging. For this section, your program does not need to respond to the GET request. Upon hearing from a client, your program can immediately signal the end of the connection (as described in the example). Once the FIN handshake has been completed, your program should terminate.

You may always assume that only one client will connect to the server at a time. For this section and the next you may also assume that no packets are corrupted or lost during transmission.

### File Transmission (10 marks)

To receive marks in this section you need to have a program that is able to:

- Perform all features outlined in the above section
- Successfully transmit a requested file over one or more packets
- Receive (but not handle) ACKs from the client during transmission

When your server receives a GET packet, it should locate the file being requested and return the file's contents over one or more DAT packets. When complete, the server should close the connection (as in the above section). You may assume that the file being requested always exists (it is expected that this file is stored in your program's working directory).

### Retransmission (15 marks)

To receive marks in this section you need to have a program that is able to:

- Perform all features outlined in the above sections
- Retransmit any packet on receiving a NAK for that packet
- Retransmit any packet that has not been acknowledged within 3 seconds of being sent

A client will send a NAK packet should it receive a corrupted packet or a packet with a sequence number it wasn't expecting (the NAK packet's acknowledgement number will contain the sequence number it *was* expecting). In this case your program should retransmit the packet with that sequence number.

If a data packet or an ACK gets lost during transmission your program should retransmit it after 3 seconds without acknowledgement. How you choose to handle timeouts is up to you, however it must work on a UNIX machine (moss). Achieving this through multithreading, multiprocessing, or signals is fine provided you only use standard libraries.

**Packet Corruption (15 marks)**

To receive marks in this section you need to have a program that is able to:

- Perform all features outlined in the above sections
- Gracefully ignore corrupt, invalid, or unexpected packets

When a corrupt, invalid, or unexpected packet is received, your program should ignore it and continue to run without error. Part of this task is determining what would constitute as an invalid or unexpected packet. You don't have to worry about checking the UDP checksum - only the contents of the RUSH header and data.

## Tips for Success

- Revisit the lectures and labs on reliable data transfer and TCP, ensuring you are familiar with the fundamentals
- Frequently test your code on moss
- Ensure your base functionality is working before attempting the more difficult tasks
- Start early - there will be limited help during the midsemester break so any questions will need to be asked in labs beforehand

## Library Restrictions

- The only communication libraries you may use are standard socket libraries which open UDP sockets
- You can't use any libraries that aren't considered standard for your language (i.e. if you have to download a library to use it it would be considered as non-standard)
- If you are unsure about whether you may use a certain library, please ask the course staff on Piazza

## Submission

Submit all files necessary to run your program. At a minimum, you should submit a file named `assign2.py`, `assign2.c`, `assign2.cpp` or `Assign2.java`. If you submit a C/C++ or Java program, you should also submit a makefile to compile your code into a binary named `assign2` or a .class file named `Assign2.class`.

**IMPORTANT:** If you do not adhere to this (eg. submitting a C/C++/Java program without a Makefile, or a .class file instead of a .java file), *you will receive 0 for this part of the assignment.*

## Marking

Your code will be automatically marked on a UNIX machine, so it is essential that your program's behaviour is exactly as specified above. Your program should complete all tasks within a reasonable time frame (for example a single packet should not take more than one second to construct and send) - there will be timeouts on all tests and it is your responsibility to make sure your code is not overly inefficient. It is expected that you will receive a small sample of tests and a basic RUSH client program before the submission deadline.

There are no marks for coding style.

## Academic Misconduct

Students are reminded of the University's policy on student misconduct, including plagiarism. See the course profile and the School web page `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

## Late Submission and Extensions

Please view the ECP for policies on late submissions and extensions.

## Version History

### v1 (08/04/2019)

- Released assignment

### v2 (14/04/2019)

- Fixed the example binary number in part B
- Added a note to part B about TCP reassembly
- Clarified that the UDP checksum can be ignored for part C