University of New South Wales
School of Electrical Engineering and Telecommunications
Term 1, 2021

ELEC4123 Elective Design Topic Cover Sheet

Lecturer: Prof. David Taubman

Assigned Team Number: (e.g., AM Team-1, PM Team 3, …) PM Team 9

Student Number	**Family Name**	**First Name**	**Signature**
Z5116634	**Hua**	**Simon**	**Simon Hua**
Z5118814	**Liu**	**Jennifer**	**Jennifer Liu**
Z5116071	**Zhang**	**Clinton**	**Clinton Zhang**
Z5133975	**Zhen**	**Stephanie**	**Stephanie Zhen**

You must tick one of the following boxes before submitting your report:

1. We undertook one of the Communications design topics: ☐
2. We left our E-Cores (and ultracapacitor) in the Lab on Friday of Week 10: ☐
3. We returned our E-Cores (and ultracapacitor) to the staff in EE-G15: ☐

ELEC4123 - Network Communications – Team 9

1. Problem Statement

The aim of the program is to accurately recover as many messages as quickly as possible before the timeout with a minimal number of failures. The key challenges identified early on are:

- Rate limiting snoop requests.
- Reconstructing message from packets.
- Communicating / controlling multiple clients.
- Sequencing / timing of operations.
- Communicating using three computers.

2. Selected design concept reasoning (system design)

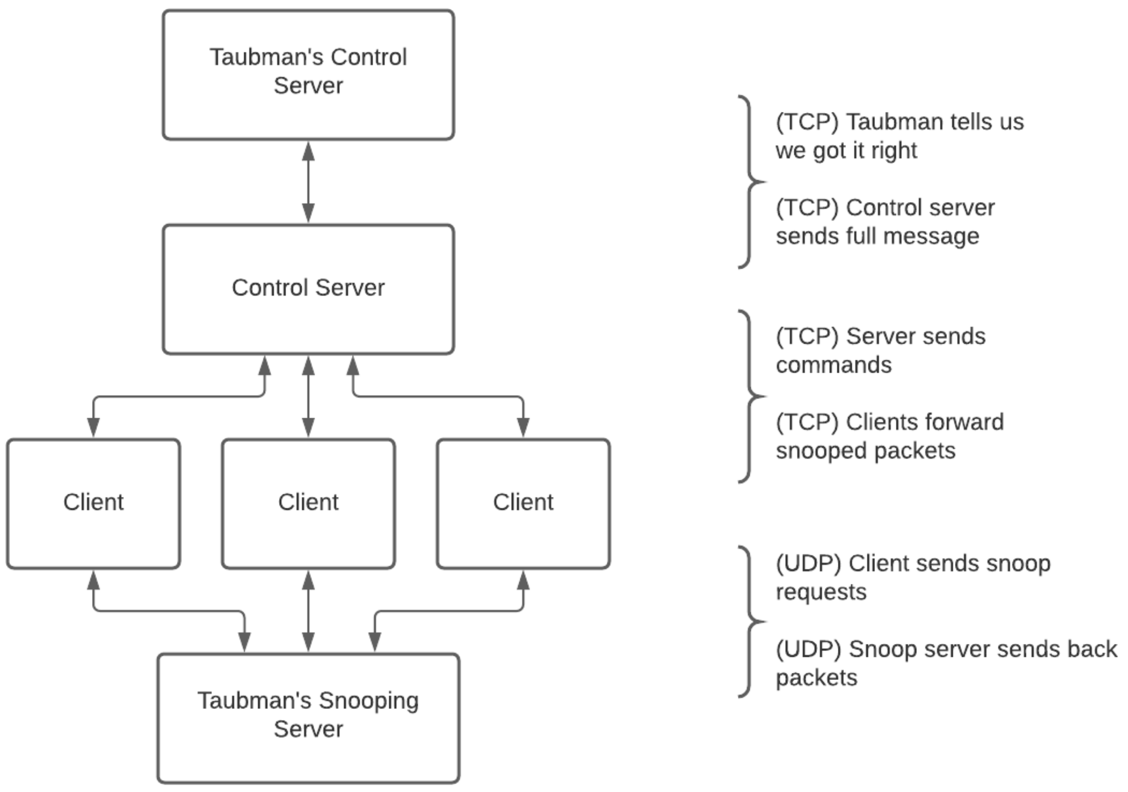
Early on, we recognised that using multiple clients would allow us a greater snooped packet throughput, which would assist in achieving the specified snooping time for the faster message speeds.

These clients would need to be able to communicate and cooperate with each other to work towards constructing the message. As such, we had several options for how the clients could be designed to communicate including a ring, peer to peer, or with a centralised control server. We realised that assembling the message would be non-trivial and as such, we decided to use a centralised control server to simplify the design and implementation of such an algorithm. This would also come with the advantage that the requisite timing for the request packets would be easier to manage when done from a central server.

In deciding to focus most of the complexity into the control server, we came up with an overall system design that had clients that would, upon receipt of a command from the control server, send the snooping request before forwarding the snooped packet back to the control server. The control server would handle the timing logic and assembly of the message as well as the transmission of the final, assembled message to the message server.

Because of difficulties in managing the delay between messages, especially since messages are of variable length, we decided to snoop as fast as possible and eventually we should pick up every packet in the message. We can then assemble the whole message and send it off to the message server.

3. <u>High level block diagram</u>



Taubman's Control Server
Control Server
Client
Client
Client
Taubman's Snooping Server
(TCP) Taubman tells us we got it right
(TCP) Control server sends full message
(TCP) Server sends commands
(TCP) Clients forward snooped packets
(UDP) Client sends snoop requests
(UDP) Snoop server sends back packets

Figure 1 - Block diagram overview (created using Lucidcharts.)[1]

<u>Allocation of tasks to team members</u>

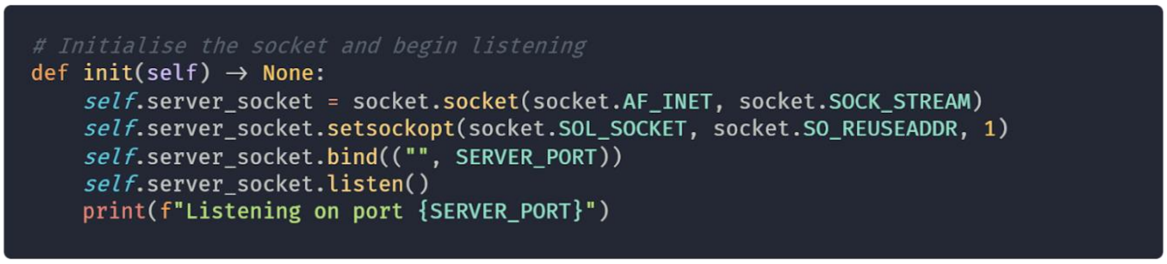
- Establishing communication- Simon
  - Communication between the control server and the three clients as well as between the three clients and Taubman's snooping server will be established. Main deliverable is the control server being able to request a snoop from a client and eventually receiving the snooped packet forwarded from the snooping server via the client.
- Control schema for clients (delay, commands) - Stephanie
  - Ensuring smooth communication between clients and control server using delays. Main deliverable is correct delay operation (process and duration).
- Message reconstruction - Jennifer
  - Reconstruct the message to be forwarded to Taubman's control server for validation. Main deliverable is identifying and implementing mathematical or otherwise method of message reconstruction.
- Send result to Taubman's control server - Clinton
  - Prepare the message for Taubman's control server as well as retrieve validation results. Main deliverable is implementation of POST as delivery method, as per specification.

4. <u>One section for each design task (written by the identified member)</u>

<u>Establishing communication</u>
<u>Simon Hua</u>

The main challenge of this task was to be able to effectively control and receive packets from the clients. Since we decided to focus most of the complexity into the control server, our implementation for the clients was relatively simple. In the interest of minimising latency, we chose to write the client in C. Additional latency would have interfered with any timing schema we could have chosen and would have reduced the performance of our system. However, the complexity of the algorithm and ease of development made us choose Python for the control server.

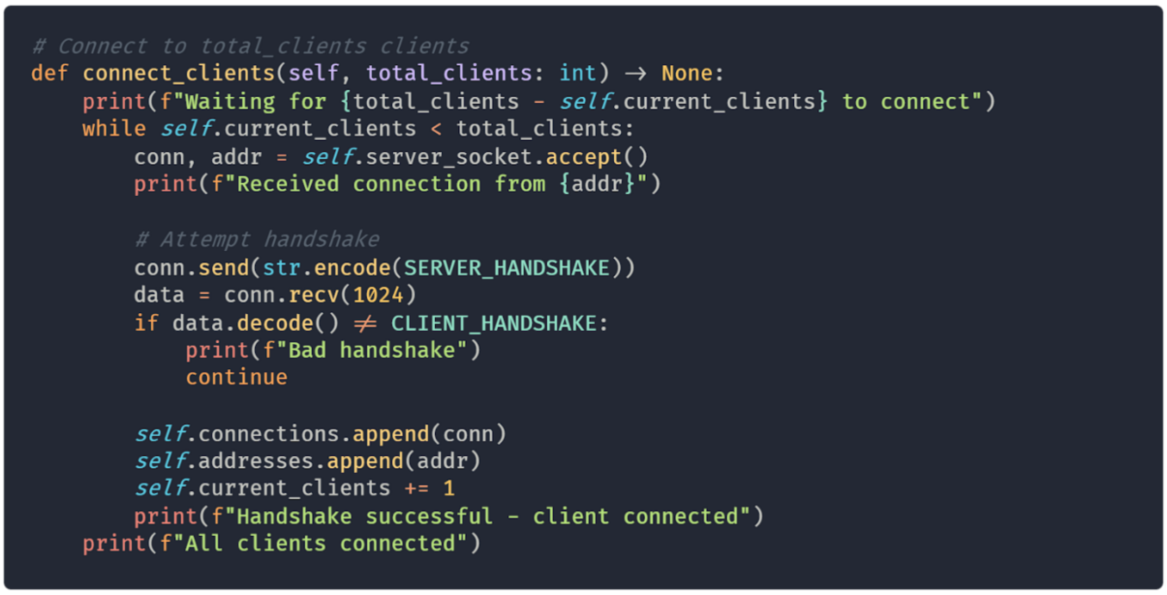
To aid in compartmentalising the code, the client-server communications are handled by a separate server class.



```python
# Initialise the socket and begin listening
def init(self) -> None:
    self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.server_socket.bind(("", SERVER_PORT))
    self.server_socket.listen()
    print(f"Listening on port {SERVER_PORT}")
```

Figure 2 - Control server initialising and listening for client connections.

Upon being initialised, the server begins to listen on the specified client-server TCP port. When started, the client will attempt to connect to a specified IP and the known client-server port. The server, upon receiving the connection, will accept the request and send a known handshake string. The client then confirms the string is correct and replies with its own handshake. If it can confirm the client handshake string, the server then stores the connection details, and the client and server are considered connected. If any step fails, the connection process for that client gets reset and needs to be restarted. The handshaking protocol ensures that the clients and server created valid connections instead of to some other running program.



```python
# Connect to total_clients clients
def connect_clients(self, total_clients: int) -> None:
    print(f"Waiting for {total_clients - self.current_clients} to connect")
    while self.current_clients < total_clients:
        conn, addr = self.server_socket.accept()
        print(f"Received connection from {addr}")

        # Attempt handshake
        conn.send(str.encode(SERVER_HANDSHAKE))
        data = conn.recv(1024)
        if data.decode() != CLIENT_HANDSHAKE:
            print(f"Bad handshake")
            continue

        self.connections.append(conn)
        self.addresses.append(addr)
        self.current_clients += 1
        print(f"Handshake successful - client connected")
    print(f"All clients connected")
```

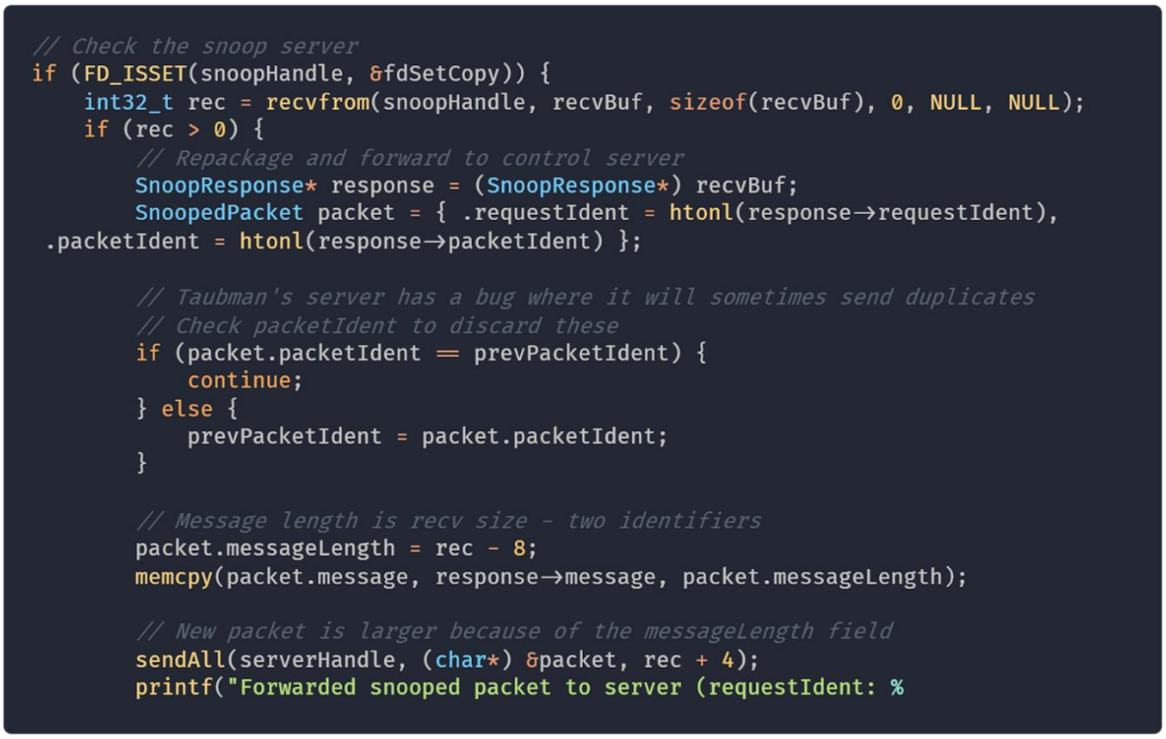
Figure 3 - Control server handshaking and connecting to clients.

When the specified number of clients has connected, the server stops accepting new connections and instead goes into configuration mode. The IP and port for the snooping server are transmitted to each client, which then each create a UDP connection to the snooping server.

With communications established, the rest of the control server program begins to execute.

The control server can request a snoop from a client by calling the appropriate method in the server class. Upon doing so, a TCP packet is transmitted over the already established connection to the specified client with the request identifier (Pr) and request number (Sr).

Each client at this point is waiting for incoming data from two connections - the control server and the snooping server. Upon receipt of the snooping request from the control server, the client generates an appropriate packet and transmits it over the UDP connection to the snooping server. Upon receipt of a snooped packet from the snooping server, the client then repackages it with an additional message length field before forwarding the new packet to the control server via TCP.



```c
// Check the snoop server
if (FD_ISSET(snoopHandle, &fdSetCopy)) {
    int32_t rec = recvfrom(snoopHandle, recvBuf, sizeof(recvBuf), 0, NULL, NULL);
    if (rec > 0) {
        // Repackage and forward to control server
        SnoopResponse* response = (SnoopResponse*) recvBuf;
        SnoopedPacket packet = { .requestIdent = htonl(response->requestIdent),
 .packetIdent = htonl(response->packetIdent) };

        // Taubman's server has a bug where it will sometimes send duplicates
        // Check packetIdent to discard these
        if (packet.packetIdent == prevPacketIdent) {
            continue;
        } else {
            prevPacketIdent = packet.packetIdent;
        }

        // Message length is recv size - two identifiers
        packet.messageLength = rec - 8;
        memcpy(packet.message, response->message, packet.messageLength);

        // New packet is larger because of the messageLength field
        sendAll(serverHandle, (char*) &packet, rec + 4);
        printf("Forwarded snooped packet to server (requestIdent: %
```

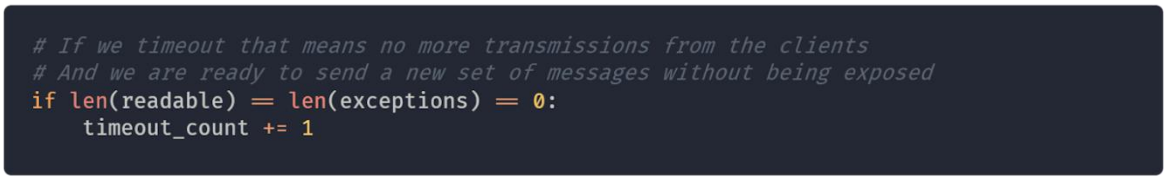
Figure 4 - Client receiving packet from snooping server, repackaging it, and forwarding to control server.

The UDP packet from the snooping server does not contain any message length details. We therefore assume that each packet we receive on the client as complete. However, because TCP does not guarantee that the packets will not be sent in multiple parts, and it is possible for us to send multiple packets before the control server performs a single recv, we add an extra message length unsigned integer to the beginning of the packet. This allows the control server to infer if the received message is incomplete (and hence decide to recv again) or if multiple packets have been received and need to be split.

The server class has an additional method to receive any snooped packets. When called, it deciphers the snooped packets sent by the client into a Python object and returns a list containing each of the snooped packets.

Control schema for clients (delay, commands)
Stephanie Zhen

The key timing problem is how to delay the clients appropriately so that we do not get exposed. Fortunately, we are given the message rate and can pass it in as an argument. Since we know the inter packet time required to not be exposed is 50 characters, we can naively wait 50-character intervals after the last packet is received from any of the clients before transmitting the next set of requests.



```
# If we timeout that means no more transmissions from the clients
# And we are ready to send a new set of messages without being exposed
if len(readable) == len(exceptions) == 0:
    timeout_count += 1
```

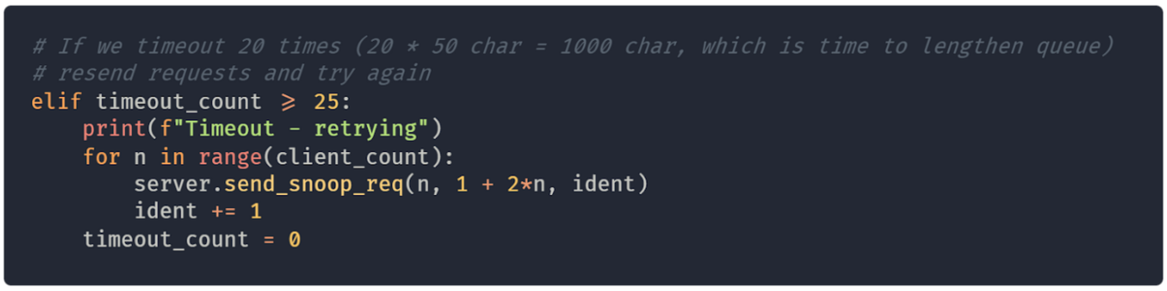
Figure 5 - Waiting for packets from the clients or timeout. In testing, we found that the timer was often not precise enough and we sometimes got exposed with 50 characters, so we used 60-character intervals to be safe.

To ensure that we do not get stuck into a loop, constantly receiving the same message, we deliberately add some random jitter into this delay. Since we do not target any missing messages specifically, this should help us break out of any loops and assist in snooping every message.

However, if we have issue with the timing, we want to ensure that our program is failsafe and does not get stuck. If, for example, we fill up a queue, then additional snoop requests will be rejected, and if we naively wait for three responses before continuing, our program will get stuck since we never receive the third response.

To handle this, we use the timeout parameter for the select function to handle our delays. This ensures that we always delay at least 50 characters worth of time from receipt of the latest transmission, even if a channel retransmits or fails to transmit. However, if we detect that any channel has failed to respond within 50 characters worth of time after a transmission, we then assume we have filled up the queue and the snooping server is no longer responding. At that point, we wait for 1000 characters worth of time, which is enough for at least one queue position to regenerate in each channel and try again.

However, it is possible that if the latency is particularly high, we could go into this waiting mode unnecessarily. Since the message length is a maximum of 20 characters and we can always send a request with position (Sr) 1 on the same host as the server, we are unlikely to run into this issue. Fortunately, testing has shown this to be the case and even if we were to go into this mode, we would not lose any messages since the packets would still be within the connection buffer.



```python
# If we timeout 20 times (20 * 50 char = 1000 char, which is time to lengthen queue)
# resend requests and try again
elif timeout_count ⩾ 25:
    print(f"Timeout - retrying")
    for n in range(client_count):
        server.send_snoop_req(n, 1 + 2*n, ident)
        ident += 1
    timeout_count = 0
```

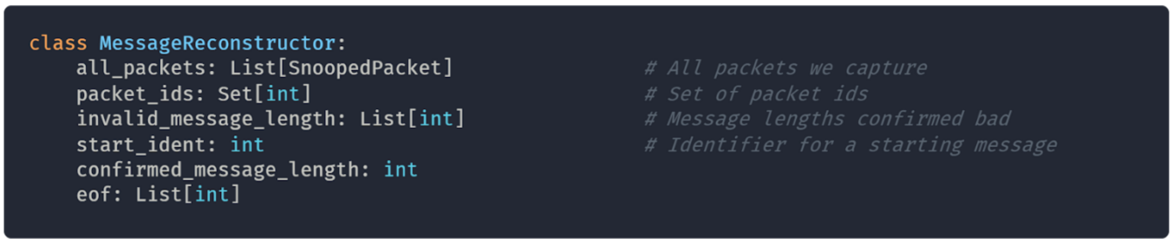
Figure 6 - If we time out enough times, we reset and try again.

Message reconstruction
Jennifer Liu

The key behind the reconstruction algorithm is recognising that the EOF character is the only reliable indexer of the message, in that it will always be contained in the final packet of a message. Therefore, to determine the length of the message to ensure that it has been reconstructed correctly, we need to detect at least two packets with the EOF character within them and subtract to find all differences.

Note that this difference is not necessarily the length of the message; rather, it is an integer multiple (n>=1) of the message length.

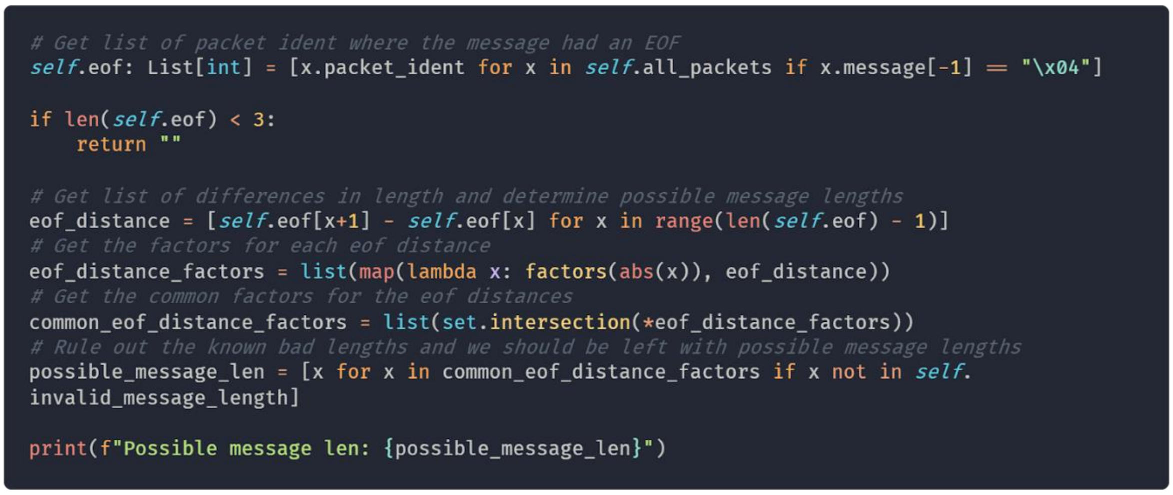
We compartmentalised all the message reconstructing code into its own class, within which we store data such as all the snooped packets and the packet ID of a packet from the beginning of the message (for message indexing), a list of known invalid message lengths, a list of previously tried invalid message lengths, and if found, the message length.



```python
class MessageReconstructor:
    all_packets: List[SnoopedPacket]                # All packets we capture
    packet_ids: Set[int]                            # Set of packet ids
    invalid_message_length: List[int]               # Message lengths confirmed bad
    start_ident: int                                # Identifier for a starting message
    confirmed_message_length: int
    eof: List[int]
```

Figure 7 - Message Reconstructor class.

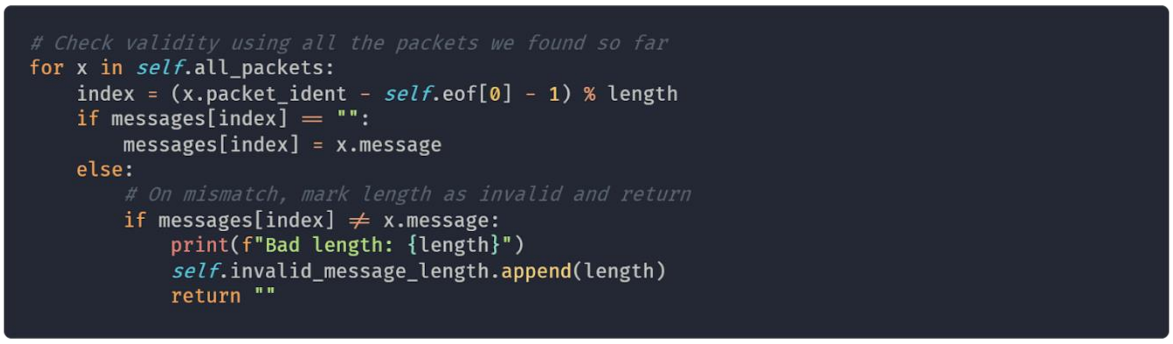
To reconstruct the message, the algorithm first collects all packets that contain EOF characters as well as the differences in the packet identifiers for these EOF packets. The message length will be one of the common factors of these calculated differences as stated above. For each common factor H, we attempt the reassembly of the message from our packets. If H is indeed the true message length, the modulus result between H and the sequence numbers of these messages will be in sequential order and wrap around as the message packets are sent repeatedly.



```python
# Get list of packet ident where the message had an EOF
self.eof: List[int] = [x.packet_ident for x in self.all_packets if x.message[-1] == "\x04"]

if len(self.eof) < 3:
    return ""

# Get list of differences in length and determine possible message lengths
eof_distance = [self.eof[x+1] - self.eof[x] for x in range(len(self.eof) - 1)]
# Get the factors for each eof distance
eof_distance_factors = list(map(lambda x: factors(abs(x)), eof_distance))
# Get the common factors for the eof distances
common_eof_distance_factors = list(set.intersection(*eof_distance_factors))
# Rule out the known bad lengths and we should be left with possible message lengths
possible_message_len = [x for x in common_eof_distance_factors if x not in self.
invalid_message_length]

print(f"Possible message len: {possible_message_len}")
```

Figure 8 - Identifying possible message lengths by using EOF packets.

Let the packet ID for a received packet be P and the packet ID for a packet containing EOF be S. Packet ID S+1 therefore points to a packet at the beginning of the message. This means the modulus of P-(S+1) with H will be the index of the packet within the whole message. If we repeat for all packets, we can detect inconsistencies in the message (e.g. differing messages in the same index) and thus determine invalid message lengths.

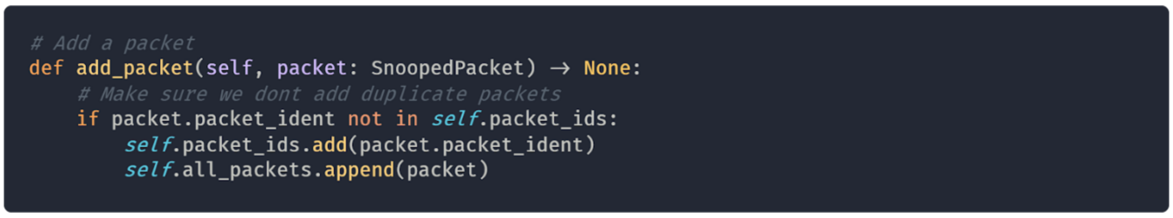


```python
# Check validity using all the packets we found so far
for x in self.all_packets:
    index = (x.packet_ident - self.eof[0] - 1) % length
    if messages[index] == "":
        messages[index] = x.message
    else:
        # On mismatch, mark length as invalid and return
        if messages[index] != x.message:
            print(f"Bad length: {length}")
            self.invalid_message_length.append(length)
            return ""
```

Figure 9 - Checking if a message length is invalid.

Using this method, we can loop through our common factors and remove known invalid message lengths from contention until we are left with a single message length. Afterwards, all that remains is to wait until all the packets have come in to assemble the whole message based on the calculated index and return the result.

Care needs to be taken to make sure that we do not pick up duplicate packets, especially EOF packets. It is possible that if we ask for messages 1, 2, and 3 from the three channels, two of the channels can snoop the same packet since the requests are sent sequentially, especially if they are on relatively high latency connections, the server is running at a high character rate, or the server is about to cycle a message out. To minimise this risk, we skip a position in the message for subsequent channels. Furthermore, we track the packet ids of each packet we snoop and only store new packets.



```python
# Add a packet
def add_packet(self, packet: SnoopedPacket) -> None:
    # Make sure we dont add duplicate packets
    if packet.packet_ident not in self.packet_ids:
        self.packet_ids.add(packet.packet_ident)
        self.all_packets.append(packet)
```

Figure 10 - Tracking duplicate packet ids with a set. Sets are extremely performant to test for the presence of a value so the overhead should be minimal.

Send result to Taubman’s control server
Clinton Zhang

Message reconstruction is attempted once all the snooped packets in an iteration have been received. If we successfully construct a message, we then send it to the HTTP server, but otherwise we continue into the next iteration by sending out a new set of snoop requests.

```python
# Add jitter into the timeout to make sure we dont just loop over the same messages
timeout = (60 + random.randrange(0, 100))/bitrate

# Send snoop requests to all clients
# Make sure we send requests with offset queue positions
for n in range(client_count):
    server.send_snoop_req(n, 1 + 2*n, ident)
    ident += 1
```

Figure 11 - Adjusting the timeout and sending out a new set of snoop requests after we fail to assemble the message.

When a message is successfully assembled, it gets passed along to a send_message function. This function establishes a TCP connection to the HTTP message server and transmits a HTTP/1.1 message containing our snooped message. We read the response - if it is a “200” code, then we are successful but there are additional messages to decode. “205” indicates we have successfully decoded the last message and we can exit. Any other code indicates that we have failed to send the correct message and we should try to ascertain the message again.

```python
# Send complete message to server
# Return true if accepted, exit if all messages sent successfully
def send_message(message: str) -> bool:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((SNOOP_SERVER_IP,HTTP_SERVER_PORT))

    header = "POST /session HTTP/1.1\r\n"
    host = "Host: " + SNOOP_SERVER_IP + ":8155\r\n"
    contentLength = "Content-Length: " + str(len(message)) + "\r\n\r\n"

    data = header + host + contentLength + message
    s.sendall(str.encode(data))

    rec = s.recv(4096)

    if "200" in rec.decode():
        return True
    elif "205" in rec.decode():
        exit()
    else:
        return False
```

Figure 11 - POSTing assembled message to HTTP server.

5. Risk mitigation strategies

The main risks we identified were firewall problems and software compatibility. We and our tutor were unsure about what hardware or operating systems our software was to run on. To mitigate the risk of encountering compatibility issues, we wrote the clients to be compatible with both Windows and Linux and used Python for our control server with the understanding that Python's implementation of sockets is platform agnostic.

There were also some concerns about connectivity through firewalls, especially those on campus. To mitigate this risk, we made sure to test on various computers and virtual machines throughout development.

6. <u>Reflection upon the design and achieved performance</u>

<u>Improvements</u>

The most significant improvement that could be made would be to add functionality to “search” for missing messages. Once we have determined message length, our current strategy is to wait and keep trying to pick up messages until the gaps have been filled in. While this works and is very simple to implement, it is possible for us to specifically determine when the missing messages are going to be played next (since we have the lengths of all the previous messages, and we know the index of the last message that just got played by the server). Specifically targeting those gaps would significantly improve the performance of our system.

The main difficulty of implementing this would be variable latency between our control server and the clients. Testing has shown connections to clients, especially those on an external network, can have highly variable latencies. The easiest way to handle this would possibly be to measure the average latency or use the control server program itself to start snooping (least variance in latency) but it could still prove problematic. A scattergun approach of grabbing the packets before and after the target and deliberately being exposed could also help.

The processing of sets of packets from each client is also delayed by the other clients unnecessarily. When the snoop requests are sent, the system waits until all the snoop responses have been gathered before performing processing and sending out a new set of requests. This means that with three clients, there is an additional delay of at least two message packets for the first channel as it now needs to wait for the second and third channels to be served by the message server on top of our enforced exposure dodging delay and processing time. When the inter packet delay due to exposure is 50 characters, or around 3 packets, this represents a significant reduction in throughput. Removing this excessive delay would require some form of a multithreading model but would result in an improvement in the performance of the system with a large number of clients.

Another potential performance improvement is with regards to our message assembling algorithm. We currently check the entire set of packets for valid message lengths / whole messages. However, new packets do not affect the previous ones, which means we could buffer the new packets separately and perform processing only on those packets, which could cause the complexity of that part to fall from O(n^2) to O(n).

Furthermore, the processing time required to check for a complete message also increases as we receive more messages since we need to parse a larger list of packets. We currently delay snooping requests by just enough to not be detected by the server after we receive the last snooped packet and before we begin processing. This fixed delay does not consider the processing time for our message assembling algorithm and we therefore delay unnecessarily. By measuring the processing time, we could increase the rate at which we snoop packets.

Furthermore, our control server is written in Python, with the main intention of allowing quick and easy development. Vectorising some of the operations, rewriting parts of it in a more performant language, or using libraries like PyPy or Numba to JIT compile the control server code could improve the performance. However, profiling has shown that the processing time on the control server has a minimal impact on overall performance and as such, there are probably better improvements we could make instead.

Limitations

The communication between client and control server occurs on a single port. This prevents multiple snooping clients from running on the same device.

There should not be any limitations with regards to maximal message rate or length - while we may not be able to reach optimal message retrieval speeds, the message should eventually be completely retrieved. However, there is a maximal packet length caused by the fixed size buffer allocated within the snooping client for communications.

Because of the random nature of our snooping, it is possible for the snooping server to sometimes infer an incorrect packet size and transmit the wrong message. This occurs when the same subset of messages is snooped repeatedly, and it happens that the offset of each packet lines up with a message length equal to a factor of the real message length. For example, if the whole message is 4 packets long and we get unlucky with the randomness in the delay, we could pick up the first and last packet repeatedly. The control server then infers that the length of the message is 2 packets since there is no issue with the offsets if we use that length and sends it off. Our server needs to pick up the final packet at least 3 times before it will begin processing so this only happens with extremely short messages, and even then, only if we are very unlucky.

Finally, there should not be a limit on the number of clients beyond what is imposed by the OS or hardware. The clients can run on Windows or Linux without an issue. However, because of the timing issue outlined in the improvements above, we get diminishing returns with additional clients.

Appendix

[1] Code can be found at: https://github.com/NotSimone/Network-Elective
[2] Lucidcharts - https://lucid.app/lucidchart/invitations/accept/inv_51038e39-8aff-44e8-a7da-6eb80e65fda8?viewport_loc=-10%2C-10%2C2389%2C1221%2C0_0