E-Greetings Card Protocol

GitHub Links

Georgia James (6623360) : https://github.com/Georgiaa-jamess/CompNetworkingCoursework

Eve Clements (6581764) : UDPCoursework/src at main · ec01049/UDPCoursework (github.com)

Seoeun Lee (6595203): https://github.com/sl980013/Computer_Networking

Marcus Tonge (6620849) : https://github.com/marcustonge/COM2022-Computer-Networking

Problem Description

The Electronic Greetings Card Protocol (EGCP) is intended for use in automating the process of sending greetings messages to people. Depending on the time of the day, the greeting will be either: “Good Morning”, “Good Afternoon” or “Good Evening”. The greeting can contain an optional additional message. Sending physical greetings cards is not good for the environment. Therefore, by sending an electronic greeting card, we can save trees by decreasing demand in paper usage for these simple greeting messages to be sent on.

The Electronic Greetings Card Protocol will need to be able to:
- Get the recipient's info such as their name to personalise the greeting. (e.g. Good Morning Marcus!)
- Check the time to send an appropriate greeting. (Good Morning, Good Afternoon or Good Evening) (will need to check between 24hr clock and AM/PM)
- Allow for customization the end of the greeting message to include a small user written message if they so choose.
- Encrypt the greetings message to protect user information and message contents. Even if sensitive content isn’t exchanged, it is nice to know the user's private messages aren’t easily read by others.
- Send the greeting message to a list of people and await their responses.

Design Decisions

Design And Architecture	Our protocol is mainly an Application, Transport and Network layer protocol. We have used layering in our protocol as layering provides us with modularity and clear interfaces. Layering also enables standards to be put in place and simply adapted as new hardware and software are

	developed. For example, say we want to change how the JSON objects are encoded, we would only have to change the application layer and we could leave the other layers untouched as it is a separate component of the transmission. At the application layer, JSON objects are encoded using ‘UTF-8’ encoding as it is currently the standard method for encoding text. A JSON object must have a ‘type’ property indicating what the data is, this has to be in cohesion with the aforementioned type convention. There is a checksum of the JSON object performed on the ‘type’ and ‘content’ properties using the CRC32 algorithm, which is then added to the JSON object under the property ‘checksum’. Checksums are a requirement of the protocol to ensure data integrity of the data we’re sending during communication, all data sent using this protocol must have a checksum attached to it. At the transport layer, the encoded JSON object is then broken down into packets ready to be sent to the receiver. We have chosen not to take a hands-on approach with packet ordering in our implementations as we have opted to focus on other features of UDP communication instead. Packets are then sent to the correct IP address at the network layer using UDP Datagram Sockets which fill the receiving buffer. From there on the socket handles the data link layer side of the transmission, sending and receiving packets of data, including sending and receiving acknowledgement packets to inform the recipient their data was received or to alert the recipient that the data we sent was received. Connection and session management is implementation specific, for the receiving server you may want to close connections with the sender after the ‘fin’ data type has been received in order to free system resources. For the sender, you may choose to close the connection with the recipient server if the connection times out a certain number of times so you can move on to sending your message to the rest of the recipients entered at the start of the program execution. We are using half-duplex communication as our protocol requires information to be transmitted between the sender and the recipient however not at the same time. This is because there is a certain communication sequence that must be followed or else the protocol won’t be able to be executed correctly.
Internetworking Complexities	Our protocol is compatible with any machine that supports JSON objects, UDP Datagram socket communication and RSA asymmetric encryption.

	Dealing with NATs by using methods such as NAT hole-punching should be supported in implementations of the protocol but it is not essential as the protocol should be able to run on a local area network. We have not chosen to use constant delivery networks as our protocol isn't handling lots of traffic and communications are short enough that the benefit of using CDNs would not be fully realised.
Data Integrity	It is up to the person implementing the protocol on how they deal with network failures such as timeouts, it is recommended that on the client side (the sender), that a default 1 second timeout be set for communications as modern day networks are fast enough to accommodate it and it prevents your implementation from stalling. Reliable data transfer mechanisms are dealt with inside the application layer of the protocol, it is recommended to account for different exceptions that can occur during data transmission over UDP. Such as timeouts, packets not being received or sent correctly, or other socket related issues. Checksums are required to be sent with each piece of data to ensure that the data that has been sent is intact and the data's integrity has been maintained during transport across the network. The checksums are computed using the CRC32 algorithm and stored inside the JSON objects that are sent. As modern day network transmissions are very reliable there most likely won't be much data loss across during transmission however it is still good to have as an added piece of protection for ensuring the data is not damaged or missing any parts. We have not actively considered multipath and routing failures as part of our protocol.
Throughput And Performance Optimisation	There is no enforced rule on how to deal with throughput issues in our protocol but as aforementioned, it is recommended that a 1 second timeout be set for transmission as modern day networks provide a reliable means of communication and there is very little data loss on smaller transmissions such as the ones sent in this protocol. The implementer may choose to cache all the messages they receive on the server side and output them when communication has been closed. It is recommended to have some form of flow control implemented to prevent against swamping the receiver with packets however it is not required in this protocol.

	There must be some form of error control and recovery so that if there is an issue during communication, an implementation of this protocol will attempt to recover and continue the communication.
Security	All messages (username and greetings messages) must be encrypted using asymmetric encryption with the recipient's public key. Public keys are exchanged after the 'sync' packet has been sent and received so that future messages can be encrypted and transmitted across a network securely. Details regarding the format in which to send the public key are mentioned above. Greetings messages must be signed with the sender's name but aside from that there is no form of authentication required. As we are not sending files there is a low chance of receiving malware during communication however it is best practice to sanitise all data received.

Our protocol was designed to be easy to understand with certain complexities added to make it secure and robust. We focused primarily on items 1, 3, and 5 from the checklist.
The protocol is designed to get user information it needs, once a connection has been established. The interface must allow a user to enter in a list of clients to send their greeting messages to. This message must be reliably transferred to the list of clients (unless the client is not listening).

Architecture

Our protocol sends JSON objects between the Client (the sender) and the Server (the recipient) in a specific sequence to transmit greetings between the users. Each object must have a 'type' property to indicate the data that is being transmitted and it must follow the typing convention as follows:

'sync' - indicates the start of a communication.
'fin' - indicates the end of a communication.
'ack' - an acknowledgement packet that indicates that the data that was sent has been received correctly by the recipient.
'message' - indicates that a message string is attached in the JSON property 'content'.
'request_username' - indicates the sender is requesting the username of the recipient.
'recipient_username' - indicates the recipient of the 'request_username' JSON object has responded with their username, this name can be set by the user if they choose so in their own implementation but it must be before a communication is started due to socket timeouts. The username is attached in the JSON object under the property 'content' and it is encrypted using the Client's public key.
'sender_public_key' - This indicates the client is sending their public key to the recipient, the key is encoded using the 'PEM' format and is decoded to a string so it can be entered into the JSON object. This key is stored inside the 'content' property of the object.
'recipient_public_key' - This indicates the server (recipient) is sending their public key to the client, the key is encoded using the 'PEM' format and is decoded to a string so it can be entered into the JSON object. This key is stored inside the 'content' property of the object.

Messages and usernames are encrypted using asymmetric encryption, the encrypted data is then encoded using Base64 encoding, these encoded bytes are then turned into a string format using “ISO/IEC 8859-1” character encoding.
The UDP port that the protocol will communicate on is 12000 only.

This communication sequence is as follows:
For the sender:

1. Start a connection with the recipient on port 12000
2. Send a JSON object with ’type’: ’sync’ to confirm initiation of connection with recipient
3. Send a JSON object with type: ‘sender_public_key’ and content: ‘your_asymm_encryption_public_key’ to send our public key to the recipient
4. Wait for a JSON object with ‘type’: ‘recipient_public_key’ to be received on the socket connection
5. Send a JSON object with ‘type’: ‘request_username’ to request the username of recipient
6. Wait for a JSON object with ‘type’: ‘recipient_username’ to be received on the socket connection
7. Decrypt and put the recipient username into the greeting message to be sent, (optional) put the custom message the user entered into the message too. Send a JSON object with ’type’: ‘message’ to recipient
8. Wait for a response JSON object with ‘type’: ‘message’ to be received on the socket connection
9. Send a JSON object with ’type’: ’fin’ to alert the ending of connection with recipient
10. Close the connection with recipient

For the receiver:

1. Wait for a connection to be initiated (leave the listener running on port 12000)
2. When a socket connection is started, receive a JSON object with ’type’: ‘sync’ that signals start of communication with client
3. Wait for a JSON object with ‘type’: ‘sender_public_key’ to be received on the socket connection
4. Send a JSON object with type: ‘recipient_public_key’ and content: ‘your_asymm_encryption_public_key’ to send our public key to the client
5. Receive a JSON object with type ‘request_username’
6. Send a JSON object with type ‘recipient_username’, content: ‘your_asymm_encrypted_username’ to the client
7. Receive a JSON object with type: ‘message’
8. Decrypt the message and display it to the user
9. Send a JSON object with type ‘message’, content: ‘your_asymm_encrypted_response message’ to the client
10. Receive a JSON object with type ‘fin’ to acknowledge end of connection
11. End the connection with the client

For example, the sender receives the recipients encrypted username, the sender should make sure the computed checksum of the received data matches the checksum that the data was sent with or else it will wait for the data to be resent. The username is decrypted and stored in the program, then an acknowledgment packet is created and transmitted back to the

We receive the server's identity during communication and send our own back at the end of our greeting message. There is no aspect of authentication in our protocol.

Interoperability & Conformance Testing Framework

We designed our interoperability and conformance testing framework using the keywords **SHOULD**, **MUST**, **SHOULD NOT** and **MUST NOT** as it encompasses the key points of a communication using this protocol:

ACK has been used as shorthand for 'acknowledgement' in the following description of the tests.

1. The client program **MUST** send a 'sync' packet to the server implementation on connecting, upon which the server should send an ACK packet in response. A message is output alerting the user on the client and server implementation that the connection has been initiated.
2. The client program **MUST** receive the name used by the server (typically the username) so it can be added to the message. This will be tested by checking whether the client implementation receives the username of other server implementations, this is displayed in the console.
3. Depending on the time, the greeting the server client starts with **MUST** be either: "Good Morning", "Good Afternoon" or "Good Evening". This will be tested by running the client programs at different times of the day and checking whether the correct greeting for the time is sent to the server implementation which should display this.
4. Clients **SHOULD** be able to enter an optional message to be attached to the body of the greeting and the greeting the server receives should display this too. To test this, the server will receive a greetings message from each client implementation and verify whether the greeting was received and displayed correctly.
5. The client program **MUST** be able to recover from an error in the case that server does not respond to it within the given timeout time.
6. Invalid IPs entered in the client implementation **MUST** be caught and throw exceptions.
7. ACK packets **MUST** be received from the server and client after data is sent. This will be tested by verifying if the client receives ACK packets from other server implementations and if the server receives ACK packets from other client implementations.
8. Checksums **MUST** be performed on the "type" and "content" properties of the JSON object. This will be tested by displaying a message regarding whether the checksum that the client receives matches the checksum that was inside the JSON object. If there is no checksum then the program will wait for the data to be sent again.

9. If a socket times out or there is a socket error with the server, the client **MUST** attempt to send the data again to ensure the data transfer mechanism is reliable.
10. The client program **SHOULD** send a ‘fin’ packet to the server implementation to signify the end of communication. Upon receiving this ‘fin’ packet, an ACK packet is received by the client from the server and the connection is then ended.

The tests will be referred to in the Conformance testing sections of our individual report sections by their test number.

Protocol Specification in RFC format

Electronic Greetings Card Protocol

Abstract

The Electronic Greetings Card Protocol (EGCP)is intended for use in automating the process of sending greetings messages to people. Depending on the time of the day, the greeting will be either: "Good Morning", "Good Afternoon" or "Good Evening". The greeting can contain an optional additional message.
This document describes the functions to be performed by the Electronic Greetings Card Protocol, the program that implements it, and its interface to programs or users that require its services.

April 2022

prepared for

University of Surrey
Department of Computer Science
Stag Hill, University Campus, Guildford GU2 7XH

Authors

Marcus Tonge, Seoeun Lee, Georgia James, Eve Clements

1. Introduction

1.1 Motivation

Sending physical greetings cards can be environmentally unfriendly as most people discard their greeting card after it has been opened. Allowing users to send an electronic greeting card instead we are more environmentally friendly by not needing to produce paper and card for these simple greeting messages to be sent on.

This protocol allows users to send greetings messages to a list of people without needing to individually write messages to the recipients.

1.2 Scope

The EGCP is intended to provide a means for users to transmit greetings messages securely and reliably in a networking environment. The EGCP is intended to be a client-server protocol that provides a structure for communication between client and server in a network environment.

1.3 Operation

As noted above, the primary purpose of EGCP is to provide a reliable way of exchanging greetings message between users.

The client program will act as a sender and take in a list of recipient IP addresses to send a greetings message to. These IP addresses are receiving servers. The client program can also take in a short, optional message that will be added to the body of the greeting when composed. The client program will then iterate through the list of IP addresses and perform the communication outlined later in this document to send the client’s greeting message to the recipient. It will continue this process until it has iterated through every address and will then close.

The server program will act as a receiver for greetings messages. It will listen for any incoming connections whilst running and when a connection is started it will follow the communication process detailed later in this document. It will receive a greetings message from the client and display it. Then the server will respond with its own message. The communication will then be terminated, and it will return to listening for any incoming connections.

2. Conventions used in this document

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be   interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Design

3.1 Data formatting

>> The EGCP transmits JSON objects between the client and the server to complete its communication. The JSON objects have the following properties: type, content, checksum. Each object MUST have a “type” property to indicate the data that is being transmitted and it must follow the message type convention in this section.

>> The ‘content’ property is not required in every object however it MUST be present when exchanging public keys, the server’s name, and any greetings messages.

>> The ‘checksum’ property MUST be present on all packets, including acknowledgement packets, for data integrity purposes.

>> The start of greetings messages sent by the client MUST follow the format of:
“Good Morning <Server Name>”,
“Good Afternoon <Server Name>”,
“Good Evening <Server Name>”.
The message chosen depends on the time of day.

The agreed upon timings for Morning, Afternoon and Evening are as follows:
Morning: 4AM onwards up to 11AM
Afternoon: 11AM onwards up to 6PM
Evening: 6PM onwards up to 4AM

>> The rest of the greetings message (the body of the message) should follow the format of:
“<optional additional short message>
From <Client Name>”

>> Messages sent by the server SHOULD follow the format:
“Thank you for your message”

>> Messages MUST be encoded and decoded using UTF-8 character encoding

>> Asymmetric encryption public keys must be encoded and decoded using “Base64” character encoding.

3.2 Error Control

>> The program implementations of this protocol MUST be robust and be able to recover from various errors that can occur during communication
>> The client program MUST wait for the server to re-send data if communication fails or there is an error with the data received.
>> The server program MUST wait for the client to re-send data if communication fails or there is an error with the data received.
>> If the sender of the message does not receive an acknowledgement packet within its own timeout after sending data it MUST attempt to resend the data.

3.3 Message Type Convention

The following are the types of JSON objects that are exchanged during a communication using EGCP and their meanings. These types are entered into the “type” property of the JSON objects

‘sync’ - indicates the start of a communication.
‘fin’ - indicates the end of a communication.
‘ack’ - an acknowledgement packet that indicates that the data that was sent has been received correctly by the recipient.
‘message’ - indicates that a message string is attached in the JSON property ‘content’.
‘request_username’ - indicates the sender is requesting the username of the recipient.
‘recipient_username’ - indicates the recipient of the ‘request_username’ JSON object has responded with their username, this name can be set by the user if they choose so in their own implementation but it must be before a communication is started due to socket timeouts. The username is attached in the JSON object under the property ‘content’ and it is encrypted using the client’s public key.

4. Communication

4.1 UDP

The EGCP communicates using the User Datagram Protocol (UDP) using IPv4. UDP wraps datagrams with a UDP header which contains four fields: Source port, destination port, length, and checksum. As the checksum field is optional in IPv4 communication we have decided to put the checksum inside the JSON object.

Communication will be handled using sockets which is a low-level networking interface.

>> The socket interface MUST provide access to the BSD socket interface. This is available on all modern Unix systems, Windows, MacOS and additional platforms.

>> The buffer size that is chosen to be used is optional. However, it is RECOMMENDED to use a buffer size of 4096

bytes to guarantee compatibility with other implementations.

An explanation of UDP can be found by viewing ‘What is UDP?’ [1]. An explanation of a UDP socket can be found at ‘Network Component: UDP Socket’[2].

An ASCII figure of the UDP Datagram header format:

An example JSON object that may be sent during communication:

```
{"type": "message", "content": encrypted_and_base64_encoded_message,
                    "checksum": 741078394}
```

4.2 The Communication Process

The following details the communication process an implementation of EGCP must follow to communicate properly. Acknowledgement packets are expected to be received whenever either the client or server sends data, else they must resend the data if an acknowledgement packet is not received promptly.

C: Generate a public and private key pair using the RSA asymmetric encryption algorithm.
S: Generate a public and private key pair using the RSA asymmetric encryption algorithm.
S: Wait for a connection to be initiated (leave the listener running)
C: Start the connection with the recipient
C: Send a JSON object with ’type’: ’sync’ to confirm initiation of connection with recipient
S: When a socket connection is started, receive a JSON object with ’type’: ‘sync’ that signals start of communication with client
C: Send a JSON object with ‘type’: ‘sender_public_key’ and ‘content’: <Your asymmetric encryption public key> to send our public key to the recipient
S: Wait for and receive a JSON object with ‘type’: ‘sender_public_key’
S: Send a JSON object with ‘type’: ‘recipient_public_key’ and ‘content’: <Your asymmetric encryption public key> to send our public key to the client

C: Wait for and receive a JSON object with ‘type’: ‘recipient_public_key’
C: Send a JSON object with ‘type’: ‘request_username’ to request the username of recipient
S: When JSON object with ‘type’ ‘request_username’ is received
S: Send a JSON object with ‘type’ ‘recipient_username’, ‘content’: <Your asymmetric encrypted username> which has been encoded using Base64 to the client
C: Wait for and receive a JSON object with ‘type’: ‘recipient_username’
C: Decode the username using Base64 then decrypt the username then put the recipient username into the greeting message to be sent, (optional) put the custom message the user entered into the message too
C: Send a JSON object with ’type’: ‘message’, ‘content’: <Your asymmetric encrypted greeting message> which has been encoded using Base64 to recipient
S: Wait for and receive a JSON object with type: ‘message’
S: Decode the message using Base64 then decrypt the message and print it to user
S: Send a JSON object with type ‘message’, content: <Your asymmetric encrypted response message’ which has been encoded using Base64 to the client
C: Wait for and receive a JSON object with ‘type’: ‘message’, decode the message using Base64 and decrypt the message then display it to the user
C: Send a JSON object with ’type’: ’fin’ to alert the ending of connection with recipient
S: Receive a JSON object with type ‘fin’ to acknowledge end of connection
C: Close the connection with server
S: End the connection with client

4.3 Internetworking

>> The implementer MAY choose to make their server public facing so it can receive messages from outside its local area network. However, this is not an essential feature and is mainly for those who want to use EGCP on a global scale.

5. Data Integrity

5.1 Acknowledgement packets

>> Acknowledgement packets are expected to be received whenever either the client or server sends data, else they MUST resend the data if an acknowledgement packet is not received promptly. This is to make the protocol more robust and reliable by ensuring messages are received by way of receiving an acknowledgement packet.

5.2 Checksums

Checksums allow the receiving device to verify the integrity of the data. Checksums are computed using the CRC32 algorithm. The data that is passed into this algorithm is the value of the JSON object properties "type" and "content". This is performed by taking the value associated with these properties (which will be in string format because it is in a JSON string), adding them together in the order "type" + "content" and then passing this value as bytes into the CRC32 algorithm. The resulting checksum is then added to the JSON object under the property "checksum".

>> If the checksum computed by the receiver does not match then it MUST wait for the data to be re-sent.

>> Checksums MUST be computed and sent within every JSON object to ensure data integrity.

6. Security

6.1 Asymmetric encryption

Asymmetric encryption must be used to encrypt data containing usernames or any greetings messages that are sent during the communication process. These asymmetric encryption keys must be generated using the RSA asymmetric encryption algorithm.

To exchange public keys, the key must be saved as bytes using the "PEM" format by using the PKCS #1 standard, this byte format of the key must be decoded into a string so it can be entered into the JSON object ready to be sent. When the public key is received it should be encoded back into a byte format using 'ISO/IEC 8859-1' encoding then loaded using the PKCS1 #1 standard to get the public key in a usable format.

>> All data sent containing usernames, or any greetings messages must be encrypted using the recipient’s public key. This is so the recipient can decode it using their private key.

An explanation of the RSA algorithm can be found in the references section [3].

6.2 Additional Security Measures

>> A person implementing this protocol MAY choose to add additional security procedures when receiving data.

They may choose to sanitise received messages to protect against code injection threats. They may choose to put in place measures protecting their server against denial-of-service attacks by disconnecting the socket listener if it receives too much data in a given amount of time. Another security concern to consider is IP spoofing and ensuring your data is not being sent to the wrong person.

However, these additional extensions are not mandatory for this protocol and are purely optional.

7. Conclusions

This protocol if implemented correctly provides a means for reliably sending greetings messages between clients and recipients (servers). This protocol could be extended further in a revision of it by allowing extra data to be transmitted along with the message such as documents, images, or other such files.

8. References

[1] Cloudflare, 2022, What is UDP?. Available at: https://www.cloudflare.com/en-gb/learning/ddos/glossary/user-datagram-protocol-udp/ (Accessed: 17th April 2022)

[2] ARM KEIL, 2022, Network Component: UDP Socket. Available at: https://www.cloudflare.com/en-gb/learning/ddos/glossary/user-datagram-protocol-udp/ (Accessed: 17th April 2022)

[3] GeeksForGeeks, January 2021, RSA Algorithm in Cryptography. Available at: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/ (Accessed: 20th April 2022)

Contribution to the RFC

Marcus Tonge wrote the Design, Communication, Data Integrity and some of the security sections of this RFC. Georgia James and Eve Clements wrote the introduction and conclusion sections. Additionally, Eve Clements contributed towards the Security section after researching how to implement asymmetric encryption in our implementations of this protocol and how to encode encrypted messages to be sent. Seoeun Lee proofread the document for any errors or contradictory pieces of the specification and helped research how to write the initial RFC.

Conformance Testing Results Summary For The Group:

A summary:

	Eve's Client	Seoeun's Client	Marcus' Client	Georgia's Client	Average
Eve's Server	10/10	10/10	10/10	10/10	1
Seoeun's Server	10/10	10/10	10/10	10/10	1
Marcus' Server	10/10	10/10	10/10	10/10	1
Georgia's Server	10/10	10/10	10/10	10/10	1
Average	1	1	1	1	

	Conformance Testing Marks
Eve	⅓(1 + 1 + 1) = 1
Seoeun	⅓(1 + 1 +1) = 1
Marcus	⅓(1 + 1 + 1) = 1
Georgia	⅓(1 + 1 + 1) = 1

Individual Contribution Report:

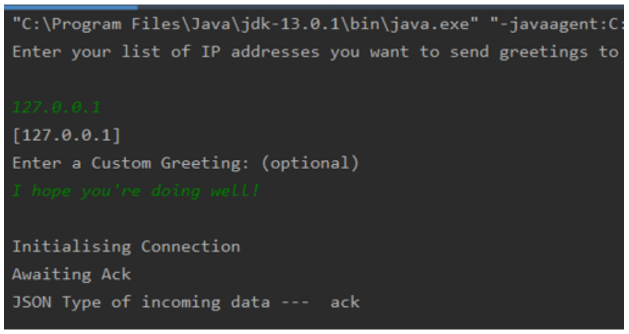
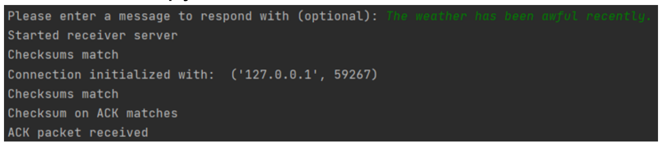
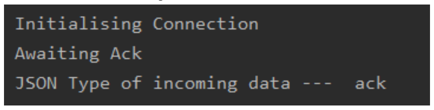
Eve Clements - 6581764

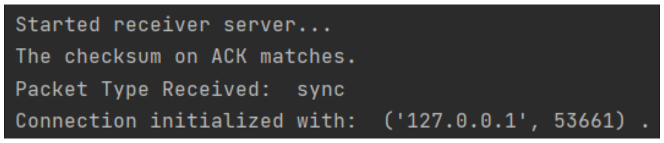
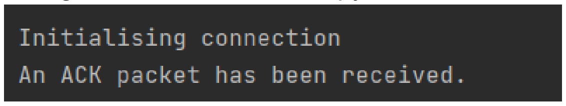
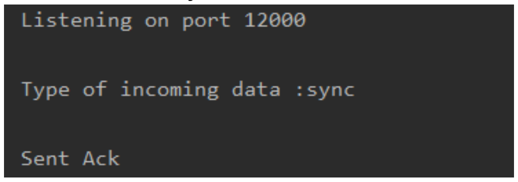
GitHub link to code: [UDPCoursework/src at main · ec01049/UDPCoursework (github.com)]

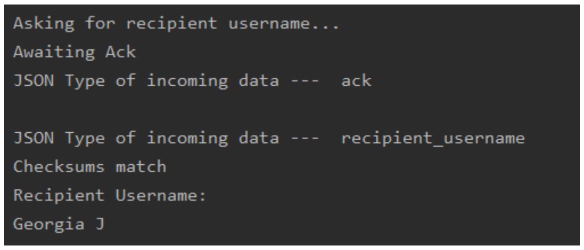
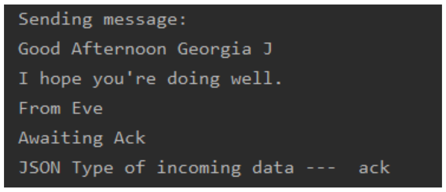
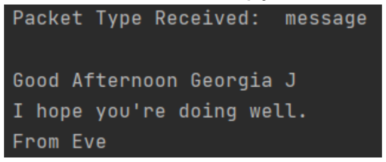
I implemented my UDP protocol using Java 13. Java's built in packages java.net and java.io were used to build the foundations of socket communication.
Additional libraries BouncyCastle (bouncycastle.org) and JSONSimple 1.1.1 have been used in order to support JSON Object transmission and RSA Asymmetric encryption in my implementation. Both must be imported.

Conformance Testing Results

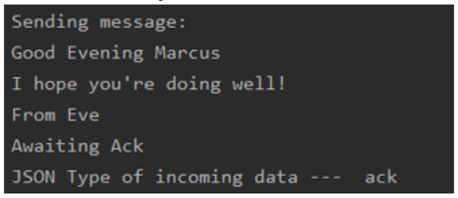
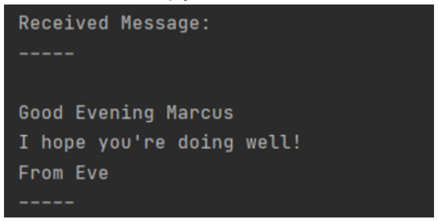
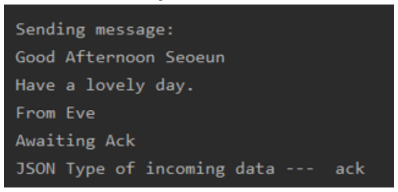
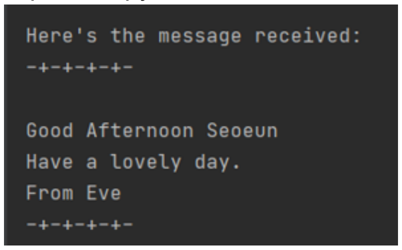
Eve's Conformance Tests	Results	Reasons

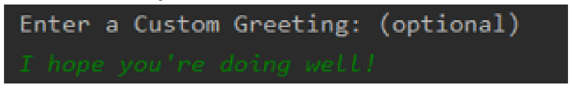
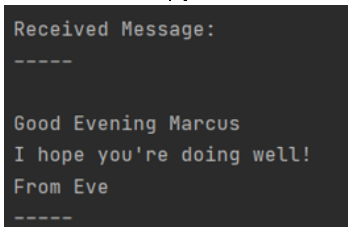
Test 1 - Client program **MUST** send a 'sync' packet to the server implementation on connecting, upon which the server should send an ACK packet in response. A message is output alerting the user on the client and server implementation that the connection has been initiated.	Passed	**Evidence of Passing**
		(Console Log of my ClientSender.java working with Marcus UDPReceiver.py):
		ClientSender.java
		
		`"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" "-javaagent:C:`
		`Enter your list of IP addresses you want to send greetings to`
		
		`127.0.0.1`
		`[127.0.0.1]`
		`Enter a Custom Greeting: (optional)`
		`I hope you're doing well!`
		
		`Initialising Connection`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		
		UDPReceiver.py
		
		`Please enter a message to respond with (optional): The weather has been awful recently.`
		`Started receiver server`
		`Checksums match`
		`Connection initialized with:  ('127.0.0.1', 59267)`
		`Checksums match`
		`Checksum on ACK matches`
		`ACK packet received`
		
		(Console Log of my ClientSender.java working with Georgia's UDPReceiverServer.py):
		ClientSender.java
		
		`Initialising Connection`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		UDPReceiverServer.py

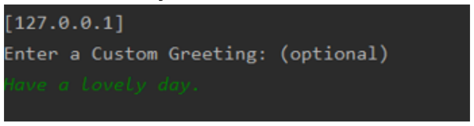
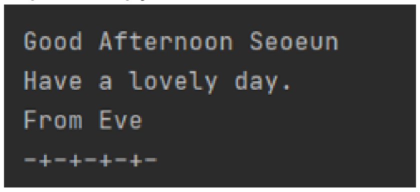
		
		`Started receiver server...`
		`The checksum on ACK matches.`
		`Packet Type Received:  sync`
		`Connection initialized with:  ('127.0.0.1', 53661) .`
		
		(Console Log of Georgia's UDPClientSender.py working with my ServerReceiver.java):
		
		Georgia's UDPClientSender.py
		
		`Initialising connection`
		`An ACK packet has been received.`
		ServerReceiver.java
		
		`Listening on port 12000`
		
		`Type of incoming data :sync`
		
		`Sent Ack`

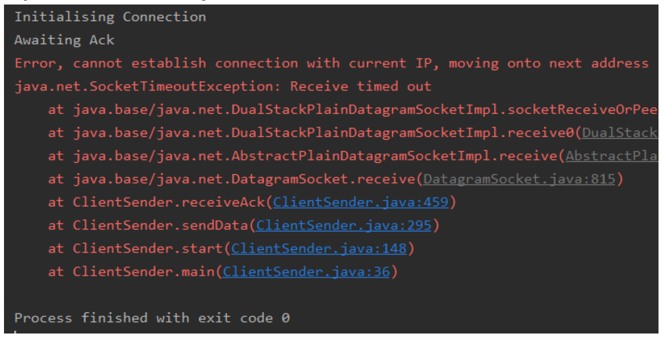
Test 2 -The client program **MUST** receive the name used by the server (typically the username) so it can be added to the message. This will be tested by checking whether the client implementation receives the username of other server implementations, this is displayed in the console.	Passed	**Evidence of Passing**
		(My ClientSender.java working with Georgia's UDPReceiverServer.py):
		
		
		`Asking for recipient username...`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		
		`JSON Type of incoming data ---  recipient_username`
		`Checksums match`
		`Recipient Username:`
		`Georgia J`
		
		`Sending message:`
		`Good Afternoon Georgia J`
		`I hope you're doing well.`
		`From Eve`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		UDPReceiverServer.py
		
		`Packet Type Received:  message`
		
		`Good Afternoon Georgia J`
		`I hope you're doing well.`
		`From Eve`

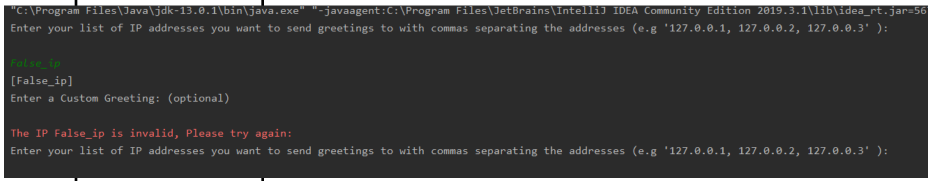
Test 3 - Depending on the time, the greeting the server client starts with **MUST** be either: "Good Morning", "Good	Passed	I tested my implementation between my own Server and Client at three points in the day to ensure the message sent was time-dependent. **Evidence of passing** **(My ClientSender.java working with Marcus'**

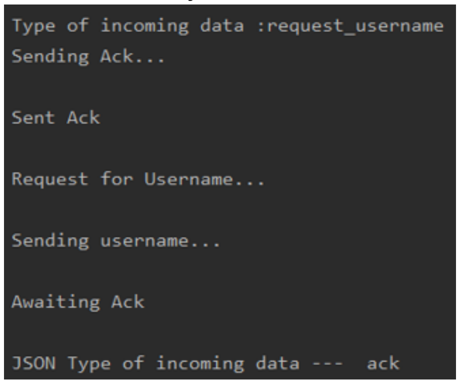
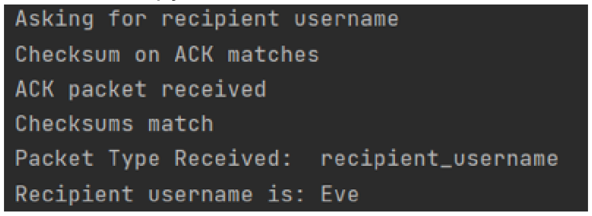
Afternoon" or "Good Evening". This will be tested by running the client programs at different times of the day and checking whether the correct greeting for the time is sent to the server implementation which should display this.		**UDPReceiver.py in the Evening)**
		ClientSender.java
		
		`Sending message:`
		`Good Evening Marcus`
		`I hope you're doing well!`
		`From Eve`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		UDPReceiver.py
		
		`Received Message:`
		`-----`
		
		`Good Evening Marcus`
		`I hope you're doing well!`
		`From Eve`
		`-----`
		
		(My ClientSender.java working with Seoeun's udpserver.py in the Afternoon)
		ClientSender.java
		
		`Sending message:`
		`Good Afternoon Seoeun`
		`Have a lovely day.`
		`From Eve`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		udpserver.py
		
		`Here's the message received:`
		`-+-+-+-+-`
		
		`Good Afternoon Seoeun`
		`Have a lovely day.`
		`From Eve`
		`-+-+-+-+-`

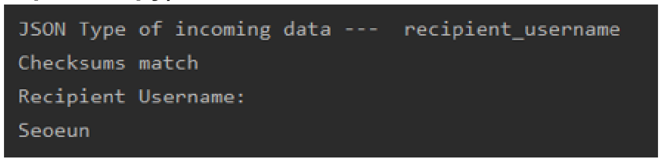
Test 4 - Clients **SHOULD** be able to enter an optional message to be attached to the body of the greeting and the greeting the server receives should display this too. To test this, the server will receive a greetings message from each client implementation and verify whether the greeting was received and displayed correctly.	Passed	**Evidence of passing**
		(My ClientSender.java working with Marcus' UDPReceiver.py)
		
		ClientSender.java
		
		`Enter a Custom Greeting: (optional)`
		`I hope you're doing well!`
		UDPReceiver.py
		
		`Received Message:`
		`-----`
		
		`Good Evening Marcus`
		`I hope you're doing well!`
		`From Eve`
		`-----`

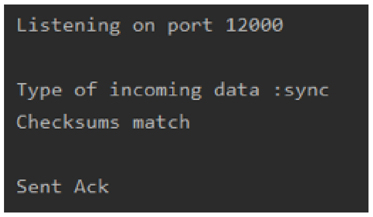
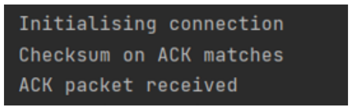
		Evidence of passing
		(My ClientSender.java working with Seoeuns udpserver.py)
		
		ClientSender.java
		
		`[127.0.0.1]`
		`Enter a Custom Greeting: (optional)`
		`Have a lovely day.`
		udpserver.py
		
		`Good Afternoon Seoeun`
		`Have a lovely day.`
		`From Eve`
		`-+-+-+-+-`

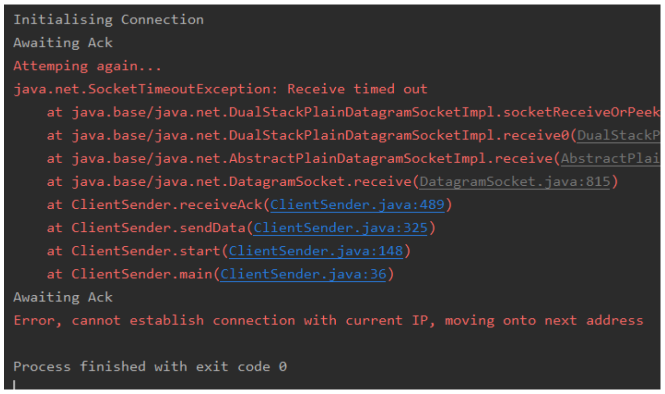
Test 5 - The client program **MUST** be able to recover from an error in the case that server does not respond to it within the given timeout time	Passed	I was able to test this by running my ClientSender.java with Georgia's UDPReceiverServer.py and terminating the server before the program had finished. This was achievable as running my java class with her python file was slower than my own classes working together.
		
		Evidence of Passing
		My ClientSender.java
		
		`Initialising Connection`
		`Awaiting Ack`
		`Error, cannot establish connection with current IP, moving onto next address`
		`java.net.SocketTimeoutException: Receive timed out`
		`    at java.base/java.net.DualStackPlainDatagramSocketImpl.socketReceiveOrPee`
		`    at java.base/java.net.DualStackPlainDatagramSocketImpl.receive0(DualStack`
		`    at java.base/java.net.AbstractPlainDatagramSocketImpl.receive(AbstractPla`
		`    at java.base/java.net.DatagramSocket.receive(DatagramSocket.java:815)`
		`    at ClientSender.receiveAck(ClientSender.java:459)`
		`    at ClientSender.sendData(ClientSender.java:295)`
		`    at ClientSender.start(ClientSender.java:148)`
		`    at ClientSender.main(ClientSender.java:36)`
		
		`Process finished with exit code 0`
		As shown, when the server was terminated, my client received a Socket TimeOut Exception and attempted to continue by closing the connection and moving on to the next IP address given.

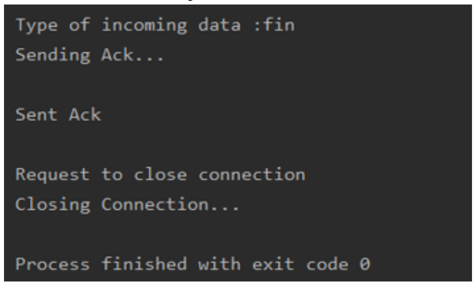
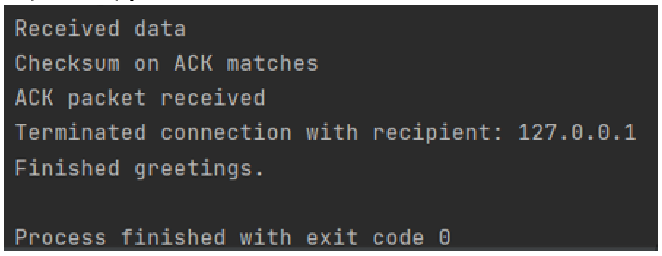
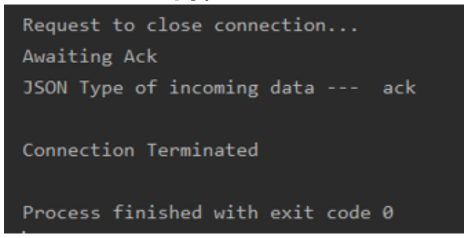
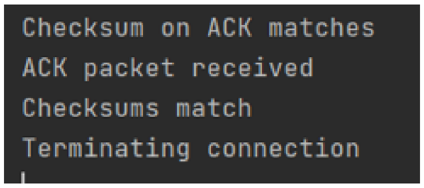
Test 6 - Invalid IPs **MUST** be caught & Throw Exceptions	Passed	I implemented my Client program to test whether the inputted IP Address was in a valid IPv4 format. If entered incorrectly, the user is alerted and given an opportunity to re-enter a valid IP. Communication is not attempted with an invalid IP.
		
		Evidence in my ClientSender.java:
		
		`"C:\Program Files\Java\jdk-13.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2019.3.1\lib\idea_rt.jar=56`
		`Enter your list of IP addresses you want to send greetings to with commas separating the addresses (e.g '127.0.0.1, 127.0.0.2, 127.0.0.3' ):`
		
		`False_ip`
		`[False_ip]`
		`Enter a Custom Greeting: (optional)`
		
		`The IP False_ip is invalid, Please try again:`
		`Enter your list of IP addresses you want to send greetings to with commas separating the addresses (e.g '127.0.0.1, 127.0.0.2, 127.0.0.3' ):`

Test 7 - ACK packets **MUST** be received after data is sent. This will be tested by verifying if my client receives ACK packets from other server implementations and if my server receives ACK packets from other client implementations.	Passed	This is shown in the above screenshots.
		
		Further Evidence (ServerReceiver.java working with Marcus' UDPSender.py):
		
		ServerReceiver.java
		
		`Type of incoming data :request_username`
		`Sending Ack...`
		
		`Sent Ack`
		
		`Request for Username...`
		
		`Sending username...`
		
		`Awaiting Ack`
		
		`JSON Type of incoming data ---   ack`
		UDPSender.py
		
		`Asking for recipient username`
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`Packet Type Received:  recipient_username`
		`Recipient username is: Eve`

Test 8 - Checksums **SHOULD** be performed on the "type" and "content" properties of the JSON object. This will be tested by displaying a message regarding whether the checksum that the client receives matches the checksum that was inside	Passed	**Evidence of passing**
		(My ClientSender.java working with Seoeun's udpserver.py):
		
		`JSON Type of incoming data ---   recipient_username`
		`Checksums match`
		`Recipient Username:`
		`Seoeun`
		
		(My ServerReceiver.java working with Marcus' UDPSender.py):

the JSON object. If there is no checksum then nothing will be displayed about it.		 `Listening on port 12000`
		
		`Type of incoming data :sync`
		`Checksums match`
		
		`Sent Ack`
		 `Initialising connection`
		`Checksum on ACK matches`
		`ACK packet received`

Test 9 - If a socket times out or there is a socket error with the server, the client **MUST** attempt to send the data again to ensure the data transfer mechanism is reliable.	Passed	When performing conformance testing, there were no Socket timeouts or errors. However, when another member temporarily removed some of their code which in turn terminated their program mid-run, I was able to test my protocols response.
		
		Evidence of passing: I ran my ClientSender.java with Georgia's UDPReceiverServer.py when it had a purposeful code removal in it.
		My client was awaiting an ack for the sent sync packet. Which was not received. Therefore, it attempted to resend the data.
		Due to the broken code, it failed again and my client gracefully shut down the connection, with the ability to move onto the next given IP.
		
		 `Initialising Connection`
		`Awaiting Ack`
		`Attemping again...`
		`java.net.SocketTimeoutException: Receive timed out`
		`    at java.base/java.net.DualStackPlainDatagramSocketImpl.socketReceiveOrPeek`
		`    at java.base/java.net.DualStackPlainDatagramSocketImpl.receive0(DualStackP`
		`    at java.base/java.net.AbstractPlainDatagramSocketImpl.receive(AbstractPlai`
		`    at java.base/java.net.DatagramSocket.receive(DatagramSocket.java:815)`
		`    at ClientSender.receiveAck(ClientSender.java:489)`
		`    at ClientSender.sendData(ClientSender.java:325)`
		`    at ClientSender.start(ClientSender.java:148)`
		`    at ClientSender.main(ClientSender.java:36)`
		`Awaiting Ack`
		`Error, cannot establish connection with current IP, moving onto next address`
		
		`Process finished with exit code 0`

Test 10 - The client program **SHOULD** send a 'fin' packet to the server implementation to signify the end of communication. Upon receiving this 'fin' packet, an ACK packet is received by the client from the server and the connection is then ended.	Passed	**Evidence of passing (My ServerReceiver.java working with Seoeun's udpclient.py):**
		
		ServerReceiver.java
		
		`Type of incoming data :fin`
		`Sending Ack...`
		
		`Sent Ack`
		
		`Request to close connection`
		`Closing Connection...`
		
		`Process finished with exit code 0`
		udpclient.py
		
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		`Terminated connection with recipient: 127.0.0.1`
		`Finished greetings.`
		
		`Process finished with exit code 0`
		(My ClientSender.java working with Marcus' UDPReceiver.py)
		
		`Request to close connection...`
		`Awaiting Ack`
		`JSON Type of incoming data ---  ack`
		
		`Connection Terminated`
		
		`Process finished with exit code 0`
		
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`Terminating connection`

Proprietary Extensions

- An extension I implemented was the use of RSA Asymmetric Encryption. This was added to protect private information such as a username or messages containing sensitive data. Overall, it provides security and trust in the communication between client and servers. Asymmetric Encryption was used over symmetric as it solves the inherent problem of needing to share a single key for both encryption and decryption. Therefore, it is much safer. On the other hand, it is usually much slower, however this is tolerable in our protocol as we are communicating with small amounts of data.
- The inclusion of checksums on all my datagram packets is useful as it allows my protocol to check incoming data for errors that could have occurred during transmission

or storage. This prevents my client and server programs from accepting data which is not in its original form. Reasons for the change in data include that it may have been maliciously replaced or a network connection was interrupted and a file did not finish downloading etc. All these problems can be mitigated via checksums.

- After having received a greeting, My server sends a default response: “Thankyou for your greeting”. I added this to enhance the user experience to reflect a real-life conversation. This also allows the user to follow the flow of the protocol. I would also have added an option for the user to customise the response content as this allows more flexibility in communication.
- UDP can be exploited for malicious purposes. As UDP does not require a handshake or permission to communicate, it is possible for attackers to ‘flood’ the server with UDP traffic. This is a type of denial-of-service attack which overwhelms the device's ability to process and respond. Therefore, If I had more time I would look into coding my implementation to include rate limiting. A server-level rate limiting such as Sliding Window Rate Limiting Algorithm would be implemented to accept a given number of requests within a specified time frame. It would measure the time between each request from each IP address. If the request limit was exceeded within the time frame, the IP address would be penalised and would not accept requests for a given time, or the socket would close.
- Another proprietary extension I would add to my protocol would be the use of Digital signatures. The inclusion of digital signatures would prove that a digital message had not been modified from the time it was signed.Digital signatures are useful as they develop trust between users and increase transparency of online communication.
  A hash function would allow a unique hash to be generated for each message being sent and when received, the recipient generates their own hash of the message to be compared against the decrypted sender's hash. Matching hases ensures the sender is authenticated and the message has not been modified.

I began testing my implementation between my own Client and Server programs to ensure all required JSON Types and messages were being sent, received and encrypted/decrypted correctly.
In order to test success, I used print statements to print every incoming and outgoing datagram packet type and content to the console so that the user can understand the ongoing communication. Upon success, I began testing my client program with my group members' server program. Being the only implementation not written in Python, I found there were differences in default encoding and decoding formats but we decided as a team that Base64 was a suitable option to ensure all our implementations worked together.
Finally, all the server and client programs for each group member were tested together one by one.

Design for Robustness

Postel’s Law: "be conservative in what you send, be liberal in what you accept".
My protocol should conform completely to the specifications, but should also be accepting non-conformant messages as long as the meaning is clear.

In order to obey Postel's Law, my protocol strictly follows the specification by only sending out JSON Objects with a specific Type property.
My implementation has been thoroughly designed to anticipate other's mistakes and be robust through the following:

- The continuous use of Try-Catch statements. When instantiating a datagram socket with a timeout and UDP datagram packet, the code has been surrounded by a try statement so that any arising Socket Errors will be catched. IO and UnknownHostExceptions are also catched and handled. This is also demonstrated when a user enters an Invalid IPv4 address.
- Setting a socket timeout. For testing purposes and to ensure packets reached the socket within the timeout, I set my timeout to 10 seconds. In the event of the server going offline or a Socket timeout Exception being thrown, the client would close the current connection and proceed to the next IP address listed.
- Packet type validation. When receiving JSONObjects, my protocol checks that the received packet type matches the one expected, from following the structure of the RFC. If not, an exception is thrown. My program will also not continue unless it receives an ACK type after sending data, to ensure that the data was received and not lost.
- The use of Checksums. Each datagram packet sent has a checksum value included for the type and content (if applicable) of the packet. Upon receiving a packet, my protocol will check whether the data has a checksum and if so, calculate the type and contents checksum and test they match. Upon failure, the datagram packet will not be accepted and request the packet to be resent. The protocol will ignore checksums if the sender has not included one in the packet.

Marcus Tonge - 6620849

I implemented this protocol using Python 3.0 and made use of various libraries but most importantly I utilised its "socket" library for communication. To use the server program properly, change the address the socket is bound to from "127.0.0.1" to the machine's network IPv4 address. For example:
socket.bind(('127.0.0.1', 12000)) - line 156 in the UDPReceiver.py, must be changed to:
socket.bind(('192.168.0.1', 12000)) given your machine's IP address is '192.168.0.1'.

The link to my code can be found here:
https://github.com/marcustonge/COM2022-Computer-Networking

Conformance Testing results
Summary score for my implementation (10 / 10 tests pass).

Conformance Tests	Results	Reasons
Test 1 (The client program MUST send a 'sync' packet to the server	All implementations passed this test.	Evidence of passing (console log of the UDP Sender client): (Console log of my UDP server working with Eve's

implementation on connecting, upon which the server should send an ACK packet in response. A message is output alerting the user on the client and server implementation that the connection has been initiated)		client. As the ‘sync’ packet was received by my server it displayed that a connection was initialized. As the output was the same with the rest of the implementations I have just shown the log from this one case): `Please enter a message to respond with (optional):` `Started receiver server` `Checksums match` `Connection initialized with:  ('127.0.0.1', 55578)` `Checksums match` `Checksum on ACK matches` `ACK packet received`
Test 2 (The client program MUST receive the name used by the server (typically the username) so it can be added to the message. This will be tested by checking whether the client implementation receives the username of other server implementations, this is displayed in the console)	All implementations passed this test	Evidence of passing: As the logs prove, the recipient username is received from all the server implementations by my client correctly and is output in the log. (Console log of my UDP Sender client working with Eve’s server): `Packet Type Received:  recipient_public_key` `Asking for recipient username` `Checksum on ACK matches` `ACK packet received` `Checksums match` `Packet Type Received:  recipient_username` `Recipient username is: Eve` (Console log of my UDP Sender client working with Georgia’s server): `Packet Type Received:  recipient_public_key` `Asking for recipient username` `Checksum on ACK matches` `ACK packet received` `Checksums match` `Packet Type Received:  recipient_username` `Recipient username is: Georgia` (Console log of my UDP Sender client working with Seoeun’s server): `Packet Type Received:  recipient_public_key` `Asking for recipient username` `Checksum on ACK matches` `ACK packet received` `Checksums match` `Packet Type Received:  recipient_username` `Recipient username is: Seoeun`
Test 3 (Depending on the time, the greeting the server client starts with must be either: “Good	All implementations had this functionality and passed the test	Evidence of passing: These tests were conducted at different times of the day to verify that the correct greeting message was displayed for the appropriate time. I have shown one case for each of the server

Morning”, “Good Afternoon” or “Good Evening”. This will be tested by running the client programs at different times of the day and checking whether the correct greeting for the time is sent to the server implementation which should display this)		implementations to demonstrate this however after testing it was verified that they all worked and displayed the correct greeting for the time.
		
		(Console log of my UDP Receiver working with Eve’s client):
		
		`Received Message:`
		`-----`
		`Good Evening Marcus`
		`I hope you’re doing well.`
		
		`From: Eve`
		`-----`
		
		`(Console log of my UDP Receiver working with Georgia’s client):`
		
		`Received Message:`
		`-----`
		`Good Evening Marcus!`
		
		`From: Georgia`
		`-----`
		
		`(Console log of my UDP Receiver working with Seoeun’s client):`
		
		`Received Message:`
		`-----`
		`Good Morning Marcus!`
		
		`From: Seoeun`
		`-----`

Test 4 (Clients **SHOULD** be able to enter an optional message to be attached to the body of the greeting and the greeting the server receives should display this too. To test this, the server will receive a greetings message from each client implementation and verify whether the greeting was received and displayed correctly.)	All implementations had this functionality and passed the test	Evidence of passing:
		I tested each client implementation and added an additional message to the greeting. I verified whether the additional message was present in the body of the greeting, which in all cases there was so the tests passed.
		
		(Console log of my UDP Receiver working with Eve’s client):
		
		`Received Message:`
		`-----`
		`Good Afternoon Marcus!`
		`This is an example message`
		
		`From: Eve`
		`-----`
		
		`(Console log of my UDP Receiver working with Georgia’s client):`
		
		`Received Message:`

		```
		-----
		Good Afternoon Marcus!
		Lovely weather we're having isn't it.

		From: Georgia
		-----

		(Console log of my UDP Receiver working with Seoeun's client):

		Received Message:
		-----
		Good Afternoon Marcus!
		Have a good day.

		From: Seoeun
		-----
		```

Test 5 (The client program **MUST** be able to recover from an error in the case that server does not respond to it within the given timeout time)	All implementations had this functionality and passed the test.	As each server implementation didn't cause any errors in testing, I tested this feature by closing the server implementation in each case and verifying how my client program reacted. The result was that for each case it didn't cause any issues which was due to my client program's extensive 'try except' approach surrounding the sending and receiving of data. An output of the console log that was displayed by my client when closing the server implementation mid transfer is shown below.
		
		(This is the output from my client console log when communication with Eve's server implementation is shut down during communication. As Eve's server was written in Java it ran slower on my machine so it was easier to close the server in time for these testing purposes as the implementations communicate very fast. As shown below, my client acknowledges the error in communication as there was a socket timeout. It alerts the user of this and attempts to resend the data to the receiver, however the communication fails and the client ends the connection with the server so it can continue to send the greetings message to the other servers in the list. The output is the same when there are errors in other server implementations.)
		```
		Enter your list of IP addresses you want to send greetings to with commas separating the addresses (e.g '127.0.0.1, 127.0.0.2, 127.0.0.3' ):
		['127.0.0.1']
		Please enter your custom message (optional):

		Initialising connection
		ACK packet received
		Exchanging public keys
		```

		`ACK packet received`
		`Socket timed out, trying again`
		`An Error occurred receiving data from the current IP, moving on to next address`
		
		`Finished sending greetings`

Test 6 (Invalid IPs entered in the client implementation **MUST** be caught and throw exceptions)	All implementations had this functionality and passed the test	This tests whether the client catches when an invalid IP address is entered (an address that is not in the IPv4 format). If this is the case the client should alert the user and user and skip over the address so a communication is not attempted to be established with it.
		
		(Console log from my client program when an invalid IP is entered, I have entered 2 for demonstrative purposes):
		`Enter your list of IP addresses you want to send greetings to with commas separating the addresses (e.g '127.0.0.1, 127.0.0.2, 127.0.0.3' ):`
		`not_an_ip, 192.`
		`['not_an_ip', '192.']`
		`Please enter your custom message (optional):`
		`IP is not valid`
		`IP is not valid`
		`Finished sending greetings`

Test 7 (ACK packets **MUST** be received from the server and client after data is sent. This will be tested by verifying if my client receives ACK packets from other server implementations and if my server receives ACK packets from other client implementations)	All implementations had this functionality and passed the test	(A console excerpt from my UDP Client when receiving a response back from Eve's server that shows that ACK packets are received, the same was outputted from Georgia's and Seoeun's server implementations. To not repeat myself I have shown one case but the log was no different in any other cases):
		
		`Enter your list of IP addresses you want to send greetings to with commas separating the addresses (e.g '127.0.0.1, 127.0.0.2, 127.0.0.3' ):`
		`['127.0.0.1']`
		`Please enter your custom message (optional):`
		
		`Initialising connection`
		`ACK packet received`
		`Exchanging public keys`
		`ACK packet received`

Test 8 (Checksums **MUST** be performed on the "type" and "content" properties of the JSON object.	All implementations had this functionality and passed the test.	This was tested by checking the console log produced when starting a communication with a server implementation. If there was a checksum in the JSON data was received it would compare the computed checksum with the one that arrived with

This will be tested by displaying a message regarding whether the checksum that the client receives matches the checksum that was inside the JSON object. If there is no checksum it will wait for the data to be sent again)	If an implementation did not use checksums in their approach my client has error handling for this case so it will be dealt with properly.	the data. If the checksums did not match it would output it and wait for the data to be resent, if they did match it would output that the checksums matched. If there was no checksum in the data it would display a message alerting the user the checksums do not match.
		
		(An excerpt of the console log of my client program communicating with Georgia's server implementation. As the console log was the same with Eve's and Seoeun's implementation I have not displayed the output from it because it was the same as with Georgia's code.):
		
		`Packet Type Received:  recipient_public_key`
		`Asking for recipient username`
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		
		(An excerpt of the console log of my client program communicating with Eve's server implementation.):
		
		`Please enter a message to respond with (optional):`
		`Started receiver server`
		`Checksums match`
		`Connection initialized with:  ('127.0.0.1', 57610)`
		`Checksums match`
		`Checksum on ACK matches`
		`ACK packet received`
		`sent our public key`
		`Checksums match`
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		
		`Received Message:`
		`-----`
		
		`Good Evening Marcus`
		`I hope you're doing well.`
		`From Eve`
		`-----`
		
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`Terminating connection`
		
		(An excerpt of the console log of my client program communicating with Eve's server implementation that I modified so that the checksum sent in the

		JSON object is not the checksum that is computed causing an error and awaiting the data to be re sent):
		`Please enter a message to respond with (optional):`
		`Started receiver server`
		`Checksums match`
		`Connection initialized with:  ('127.0.0.1', 57610)`
		`No checksum attached to the data, awaiting the data to be re-sent`

Test 9 (If a socket times out or there is a socket error with the server, the client **MUST** attempt to send the data again to ensure the data transfer mechanism is reliable)	All implementations had this functionality and passed the test	I had no communication issues with my client when testing with any of the other server implementations. When removing certain pieces of code from other server implementations to test this my program recovered in the same way as detailed below.
		(An excerpt of the console log from my client implementation when communicating with Eve's server. When running Eve's code my network had an error hence the additional communication error messages being displayed. However, my client recovered from this and proceeded to execute the rest of the communication protocol without any additional errors).
		
		`Initialising connection`
		`ACK packet received`
		`Exchanging public keys`
		`No ACK packet received`
		`No ACK packet received`
		`Error with connection - Resending Data`
		`ACK packet received`
		`Packet Type Received:  recipient_public_key`
		`Asking for recipient username`

Test 10 (The client program should send a 'fin' packet to the server implementation to signify the end of communication. Upon receiving this 'fin' packet, an ACK packet is received by the client from the server and the connection is then ended)	All implementations had this functionality and passed the test	The client should send a 'fin' packet at the end of communication to alert the server that the communication is now finished. It should receive an acknowledgement packet back then proceed to terminate the connection.
		
		(Excerpt from my client console log when ending a connection with other server implementations. As shown below, the ACK packet is received after sending the 'fin' packet then the connection is terminated. As this was the last server the client had to send greetings to it also displayed a message indicating it had finished sending the greetings message.):
		
		`Received Message:`
		`-----`

		`Thank you for your greeting`
		`-----`
		
		`ACK packet received`
		`Terminated connection with recipient -`
		`127.0.0.1`
		`Finished sending greetings`
		
		(Excerpt from my server console log when ending a connection with other client implementations. As shown below it will display a message alerting the user that the connection with the client has been terminated as it's finished communicating with it. I used a keyboard interrupt to stop the program from executing which displayed my cached greetings from the server running.):
		
		`Received Message:`
		`-----`
		`Good Evening Marcus`
		`This is a test`
		
		`From: Eve`
		`-----`
		
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`Terminating connection`
		`^C`
		
		`Cached Greetings:`
		
		`Good Evening Marcus`
		`This is a test`
		
		`From: Eve`
		
		`End of program`

The client was capable of sending it's greetings message to all 3 implementations at the same time when the server's were being ran on separate computers however this is difficult to show in testing. However, the test for the client to be able to send messages to multiple machines did pass.

Proprietary Extensions
I have added a profanity filter to my client and server programs so that any messages received containing profanity are filtered before being displayed to the user. I believe this is useful as profanity filters offer moderation for people that want to protect themselves from harassment or

abusive content. Profanity filters also help improve the internet by eliminating or replacing offensive words. This is also useful as younger, more impressionable people could communicate using this protocol and it is better to filter out abusive words so they aren't affected by it in any way.

I have added a message cache on the server program so that every time a greetings message is received it is added to a list of received messages. When the server is stopped using a keyboard interrupt (Control + C), the socket is closed and the messages are displayed to the user.

My server client allows the user to define the response message that is sent to the client at the end of communication. If the user chooses to leave this message blank it will default to the standard "Thank you for your message!". I thought this extension was more a user experience addition as it allows the user to personalise their responses.

I tried to implement a DoS protection feature on my server program however I could not get it to work. With enough time I would be able to. I would implement this feature by counting how many packets are being received and if it reaches a certain amount, say 50 requests, in 1 second I would close the socket temporarily then restart it after a minute had elapsed. Enabling a DoS protection feature can filter suspicious or unreasonable packets to prevent the network from being flooded with large amounts of fake traffic. This would be useful as it would stop system resources from being consumed where the program has to deal with bogus requests.

I tested my implementation by running my client program and attempting to communicate with my group members implementations of a server. Using print statements I was able to determine whether acknowledgement packets were being sent, if checksums matched and if there were any errors during communication and how they were being dealt with. Initially I tested my implementation by testing if the client and server programs I wrote could communicate with each other and would exchange greetings messages as they should. I also tested that based on the time of day, the correct greeting message was sent. For example, if it was 8AM a greeting of 'Good Morning' should be sent when the client program is executed, or if it was 9PM a greeting of "Good Evening" should be sent. Adding print statements after every core part of the communication allowed me to know exactly what was going on during the program execution and if my implementations were working effectively as they should. To test if my client program would send the same greetings messages to multiple servers (the IP addresses that are entered at the start of program execution), I ran 2 versions of the server with different IP addresses and validated whether the greetings messages were received by both servers.

I repeated this same testing approach with my server program and tested whether my group members' implementations of the client program would communicate with it as defined in the RFC. My client and server programs passed all the tests detailed in the interoperability and conformance testing framework.

Designing for robustness

The robustness principle (Postel's Law) defined by Jon Postel states: "Be conservative in what you do, be liberal in what you accept from others". This is the principle of robust communication, which is communication that can withstand intentional or unintentional disturbances of various kinds. In the context of computer networking a disturbance would be items such as logical errors and signal disturbances among other things.

My implementation of the protocol is very robust, and the programs I have coded anticipate many errors that may occur during protocol execution. I have surrounded each of my socket sending and receiving blocks with 'try and except' statements which allow my programs to catch any exceptions that may come up due to errors in the communication. This also allows my program to recover from the error and attempt to continue the communication unless either the client or server goes offline. This case of either of the programs going offline has been accounted for with a timeout. On my client program if the server stops responding the connection will timeout then the program will re-attempt communication.

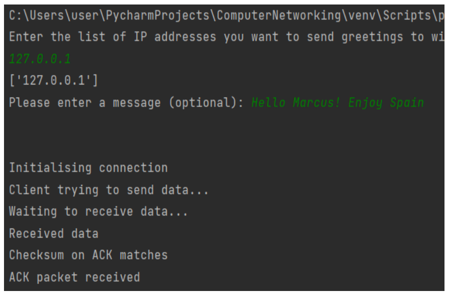
Both my client and server implementations have good data integrity. Every packet I send has checksums attached and will check for checksums on every packet that is received if there is one. If the checksum computed on the receiver side does not match the one that was attached with the data it won't send an acknowledgement packet and will wait for the data to be re-sent until the data is sent or a timeout occurs. As defined in the RFC, acknowledgement packets must be sent after receiving data and must be received after sending data which my program implements.

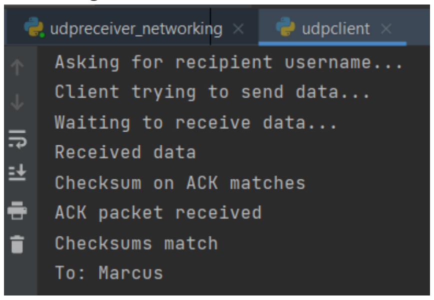
I have accounted for socket timeouts and after that an umbrella exception handler for any errors arising from the socket. Any exception that occurs outside of the socket library is dealt with as an unknown error. For the client program this means the communication with the current server will be ended and the socket connection is closed before moving on to the next server in the list. For the server program if a keyboard interrupt occurs the socket will close and the cached messages built up during the program runtime will be displayed before exiting the program. If any other exception occurs it will restart the communication process and wait for a new connection to be initiated by expecting a JSON object with "type": "sync" to come through.

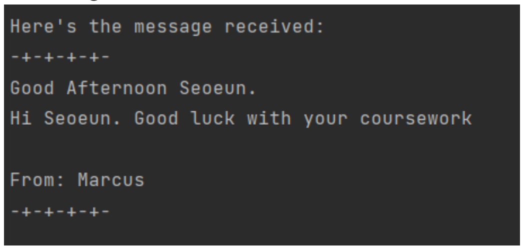
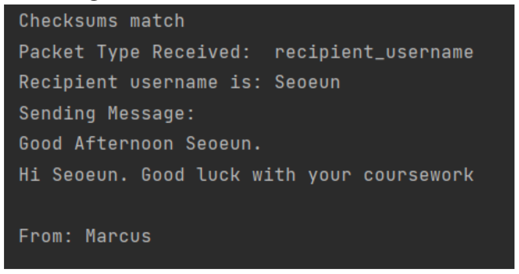
Seoeun Lee - 6595203

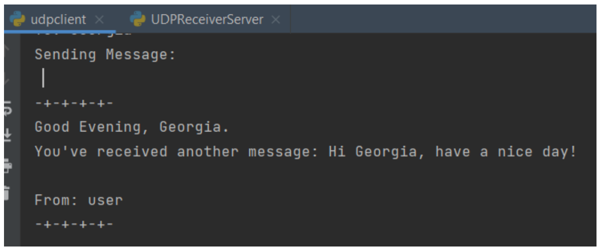
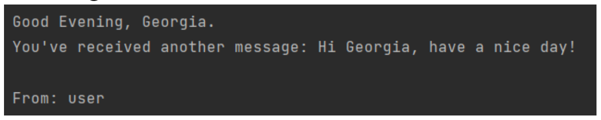
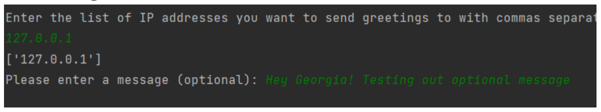
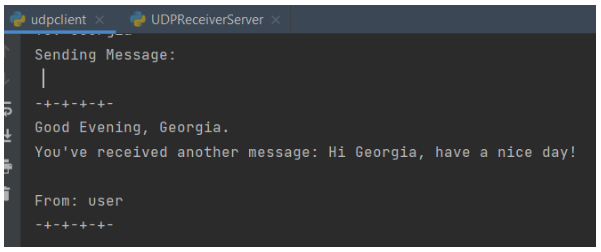
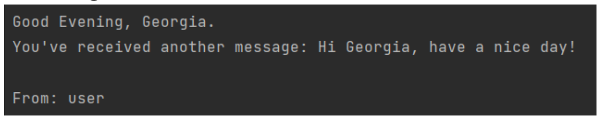
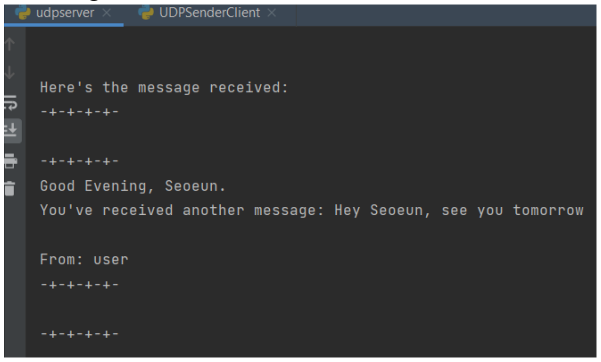
Link to the code: https://github.com/sl980013/Computer_Networking

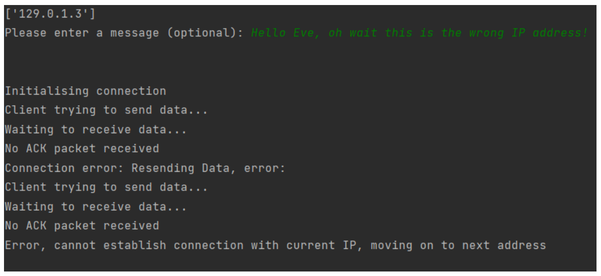
Seoeun's Conformance Test	**Results**	**Reasons**
Test 1 (The client program **MUST** send a 'sync' packet to the	Passed all tests	**Marcus' server- Seoeun's client test log:**

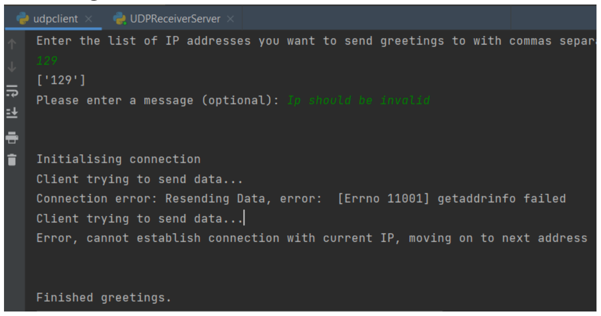
server implementation on connecting, upon which the server should send an ACK packet in response. A message is output alerting the user on the client and server implementation that the connection has been initiated)		
		`C:\Users\user\PycharmProjects\ComputerNetworking\venv\Scripts\p`
		`Enter the list of IP addresses you want to send greetings to wi`
		`127.0.0.1`
		`['127.0.0.1']`
		`Please enter a message (optional): Hello Marcus! Enjoy Spain`
		
		`Initialising connection`
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		>> Connection initiated

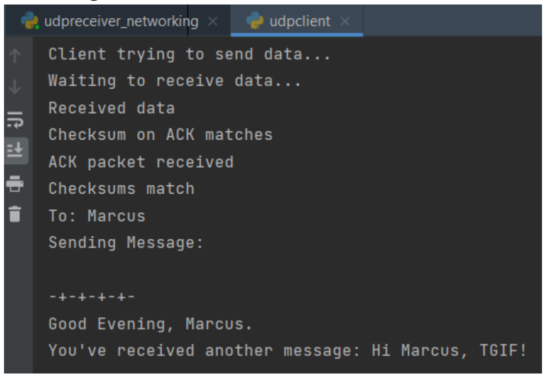
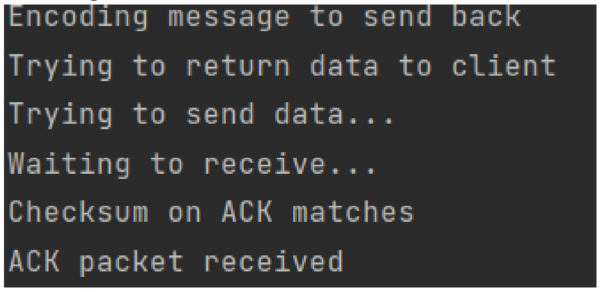
<u>Test 2</u>
(The client program **MUST** receive the name used by the server (typically the username) so it can be added to the message. This will be tested by checking whether the client implementation receives the username of other server implementations, this is displayed in the console.)
	Passed all tests	**Marcus' server- Seoeun's client test log:**
		Message sent from client:
		
		udpreceiver_networking udpclient
		`Asking for recipient username...`
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`To: Marcus`
		>> Successfully received receiver's name shown as "To: Marcus"

<u>Test 3</u>
(Depending on the time, the greeting the server client starts with **MUST** be either: "Good Morning", "Good Afternoon" or "Good Evening". This will be tested by running the client programs at different times of the day and checking whether the correct greeting for the time is sent to the server implementation which should display this.)
	Passed all tests	**Seoeun's server - Marcus' client test log:**
		Message received to server:
		
		`Here's the message received:`
		`-+-+-+-+-`
		`Good Afternoon Seoeun.`
		`Hi Seoeun. Good luck with your coursework`
		
		`From: Marcus`
		`-+-+-+-+-`
		Message sent from client:
		
		`Checksums match`
		`Packet Type Received:  recipient_username`
		`Recipient username is: Seoeun`
		`Sending Message:`
		`Good Afternoon Seoeun.`
		`Hi Seoeun. Good luck with your coursework`
		
		`From: Marcus`
		
		Seoeun's client - Georgia's server test:
		Message sent from client

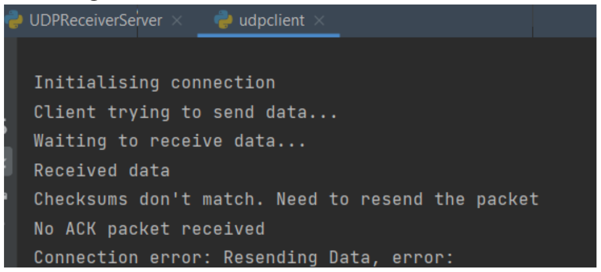
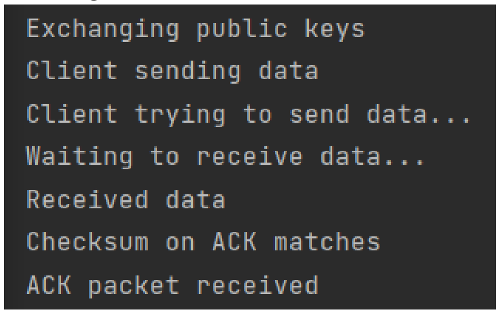
		 udpclient UDPReceiverServer Sending Message: | -+-+-+-+- Good Evening, Georgia. You've received another message: Hi Georgia, have a nice day! From: user -+-+-+-+- Message received to server:  Good Evening, Georgia. You've received another message: Hi Georgia, have a nice day! From: user >> Prints out different greetings at different times.
<u>Test 4</u> (Clients **SHOULD** be able to enter an optional message to be attached to the body of the greeting and the greeting the server receives should display this too. To test this, the server will receive a greetings message from each client implementation and verify whether the greeting was received and displayed correctly.)	Passed all tests	**Seoeun's client - Georgia's server test:** Message sent from the client:  Enter the list of IP addresses you want to send greetings to with commas separat 127.0.0.1 ['127.0.0.1'] Please enter a message (optional): Hey Georgia! Testing out optional message  udpclient UDPReceiverServer Sending Message: | -+-+-+-+- Good Evening, Georgia. You've received another message: Hi Georgia, have a nice day! From: user -+-+-+-+- Message received to the server:  Good Evening, Georgia. You've received another message: Hi Georgia, have a nice day! From: user **Georgia's client - Seoeun's server test:** Message received to the server:  udpserver UDPSenderClient Here's the message received: -+-+-+-+- -+-+-+-+- Good Evening, Seoeun. You've received another message: Hey Seoeun, see you tomorrow From: user -+-+-+-+- -+-+-+-+-
<u>Test 5</u> (The client program **MUST** be able to recover from an error in the case that	Passed all tests	**Eve's server- Seoeun's client test log:** Message from the client:

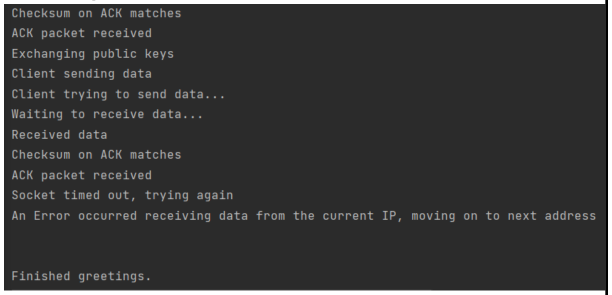
server does not respond to it within the given timeout time.)		
		`['129.0.1.3']`
		`Please enter a message (optional): Hello Eve, oh wait this is the wrong IP address!`
		
		`Initialising connection`
		`Client trying to send data...`
		`Waiting to receive data...`
		`No ACK packet received`
		`Connection error: Resending Data, error:`
		`Client trying to send data...`
		`Waiting to receive data...`
		`No ACK packet received`
		`Error, cannot establish connection with current IP, moving on to next address`

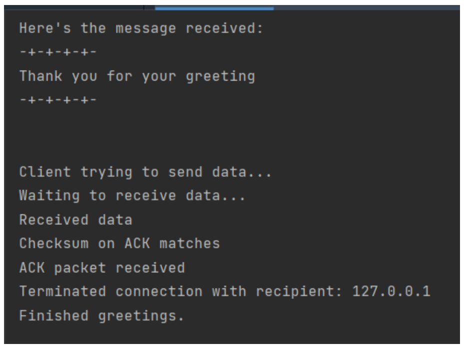
Test 6
(Invalid IPs entered in the client implementation **MUST** be caught and throw exceptions.)
	Passed all tests	**Seoeun's client - Georgia's server test:**
		Message sent from client:
		
		`Enter the list of IP addresses you want to send greetings to with commas separ`
		`129`
		`['129']`
		`Please enter a message (optional): Ip should be invalid`
		
		`Initialising connection`
		`Client trying to send data...`
		`Connection error: Resending Data, error:  [Errno 11001] getaddrinfo failed`
		`Client trying to send data...`
		`Error, cannot establish connection with current IP, moving on to next address`
		
		`Finished greetings.`

Test 7
(ACK packets **MUST** be received from the server and client after data is sent. This will be tested by verifying if the client receives ACK packets from other server implementations and if the server receives ACK packets from other client implementations.)
	Passed all tests	**Marcus' server- Seoeun's client test log:**
		Message sent from client:
		
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		`Checksums match`
		`To: Marcus`
		`Sending Message:`
		
		`-+-+-+-+-`
		`Good Evening, Marcus.`
		`You've received another message: Hi Marcus, TGIF!`
		
		Seoeun's server-Eve's client test log:
		Message received to the server:
		
		`Encoding message to send back`
		`Trying to return data to client`
		`Trying to send data...`
		`Waiting to receive...`
		`Checksum on ACK matches`
		`ACK packet received`

Test 8	Passed all tests	**Seoeun's client - Georgia's server test:**

(Checksums **MUST** be performed on the "type" and "content" properties of the JSON object. This will be tested by displaying a message regarding whether the checksum that the client receives matches the checksum that was inside the JSON object. If there is no checksum then the program will wait for the data to be sent again.)		Message sent from the client:
		
		
		`Initialising connection`
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksums don't match. Need to resend the packet`
		`No ACK packet received`
		`Connection error: Resending Data, error:`
		>> Ex1: Modified my udpclient file to deliberately unmatch checksum to print out there's no checksum match
		Seoeun's client - Eve's server test:
		Message sent from the client:
		
		`Exchanging public keys`
		`Client sending data`
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		>> Ex2: Print statement when checksums match

<u>Test 9</u>
(If a socket times out or there is a socket error with the server, the client **MUST** attempt to send the data again to ensure the data transfer mechanism is reliable.)
	Passed all tests	**Seoeun's client - Eve's server test:**
		Message received from the server:
		
		`Checksum on ACK matches`
		`ACK packet received`
		`Exchanging public keys`
		`Client sending data`
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		`Socket timed out, trying again`
		`An Error occurred receiving data from the current IP, moving on to next address`
		
		`Finished greetings.`

<u>Test 10</u>
(The client program **SHOULD** send a 'fin' packet to the server implementation to signify the end of communication. Upon receiving this 'fin' packet, an ACK packet is received by the client from the server and the connection is then ended.)
	Passed all tests	**Seoeun's client-Eve's server:**
		Message from client:
		
		`Here's the message received:`
		`-+-+-+-+-`
		`Thank you for your greeting`
		`-+-+-+-+-`
		
		`Client trying to send data...`
		`Waiting to receive data...`
		`Received data`
		`Checksum on ACK matches`
		`ACK packet received`
		`Terminated connection with recipient: 127.0.0.1`
		`Finished greetings.`

Proprietary extensions

My E-Greetings protocol is designed as a half-duplex as the server and client exchange the data one at a time not simultaneously.

Firstly, I've implemented checksums in the client and server programs. Checksum is crucial part of communication in the network. It checks whether all message packets of the E-Greetings protocol have safely been delivered or not. Hence, to ensure the data delivery and check data loss, checksums were implemented using the zlib library.

Secondly, the profanity filter has been implemented as another proprietary extension on both the client and server programs. It was done to protect users' mental health, as the issues on abusive comments leading to harmful behaviours are growing these days along with the increased usage of the internet. This was done by utilizing the better_profanity extension package on python using importing profanity library.

Thirdly, asymmetric encryption has been implemented. It was done to secure the user's messages from message leakage and hacking. This was done by using python's 'rsa' library. Exchanging the public keys between the server and the client, the programs ensure that the message deliveries don't get interrupted by external people.

Lastly, I implemented a caching function. Caching allows the users to remind their conversation for a natural flow with the greetings. It displays the history of messages. It's implemented by creating an empty list for cache and appending the messages sent and received within the server. After the socket close, the cached messages are shown.

The implementation was tested by running my client against mine and the groupmates' server programs. My client implementation requires a python environment with additional instalments for 'zlib', 'rsa', and 'better_profanity'. To test the programs, firstly, I entered some reply messages in the server program console. It then outputs the start of the server. Secondly, I run my client program and enter the IP addresses and optional messages. Then the two programs start exchanging json packets, print out the progress of it, and finish the execution. I ensured the json packet exchanges (acknowledgement, synchronisation, public key, username, message, and finish) by utilizing print statements after each packet exchange. When a user connects to the server and starts sending the message, it prints out specific packet exchange statements. For instance, to check that the implemented protocol meets the essential bases of the group's protocol, both programs print out 3 different greetings based on a time in the day (4 am - 11 am: Good Morning, from 11 am to 18 pm: Good Afternoon, From 18 pm to 4 am: Good Evening) with an optional message along with extra packet exchanging messages. Also, to check that all messages have been delivered safely, checksum validation messages get displayed. I ran different versions of the server with different IP addresses to test out if the implemented protocol works on multiple servers.

Designing for robustness

According to Postel's law, we should be conservative in what we do, be liberal in what we accept from others.
Both client and server programs are designed to be robust, anticipating various errors during the run of the protocol. The codes include 'try' and 'except' statements to check if there are any socket errors in every function and exchange of json object packets. It assists to move on to the next execution stage when there's an error. The programs try to recover from the error whenever there's a socket timeout or socket error. The programs do this by throwing try and except when they encounter socket.timeout and socket error, and repeat the failed operation reattempting the communication.
Moreover, socket timeouts, socket errors, and unknown errors get handled. In the case of a client program, it finishes the connections and closes the socket. In the case of the server program, it closes the socket and prints out the cache. As it's mentioned above, the programs expect to reattempt communication when they encounter other problems.
Finally, the implementations include a checksum. This gives the reliability and trustworthiness of data throughout their lifecycle. All json packets (acknowledgement, synchronization, public key, username, message, and finish) include checksums, and programs validate delivery of them with the print statements. Acknowledgement packets are supposed to be sent and received after receiving and sending data. When the checksums don't match, the program doesn't send an acknowledgement packet and waits for the second attempt of sending the data or a timeout error.

Georgia James - 6623360

https://github.com/Georgiaa-jamess/CompNetworkingCoursework

While implementing the E-Greetings Card Protocol, I tested my own implementation of the UDPClientSender and UDPSeverReceiver to ensure they worked as expected. It was essential that a user would be able to enter a list of IPs as well as a custom message, within the Pycharm environment I was able to see whether the prompts were displayed and data was collected correctly. Another important part of the protocol was to have the correct greeting displayed depending on the time of day - once I implemented this feature, I tested it in the morning, afternoon and evening to ensure the correct greeting was chosen. While testing my implementation as it was created, the use of print statements allowed me to see if the correct data was collected, generated and sent. Try and except statements were also very useful when trying to catch errors that would have prevented the program from running, this also provided useful when interacting with other implementations, as I talk about in Designing for robustness. Alongside my group members and their implementations, I tested my implementation with theirs by running my UDPSender client with my group member's implementation of the UDPReceiverServer, the print statement used in the development phase of my implementation once again informed me the correct data was collected, generated and sent, if ack packets were being sent and received, if checksums matches and any errors that were thrown . My group members informed me of how my UDPReceiverServer ran alongside their UDPSenderClient in terms of calculating the correct checksum, sending ack packets back to the client to confirm data was sent and if the Server was able to send a greeting back. My UDPSenderClient and UDPReceiverServer passed all the tests detailed in the Interoperability and Conformance Testing Framework.

Conformance Testing Results

Conformance Tests	Results	Reasons/Evidence
Test 1 (The client program MUST send a ‘sync’ packet to the server implementation on connecting, upon which the server should send an ACK packet in response. A message is output alerting the user on the client and server implementation that the connection has been initiated)	All implementations had this functionality and passed the test.	Once the sync packet was received by my server, the print statements showed the initialised connection with all clients. (Console log of my UDPServerReceiver working with Eve’s client): Started receiver server. Checksums match. Connection initialised with: ('127.0.0.1', 55578). Checksums match. ACK packet has been received. (Console log of my UDPServerReceiver working with Marcus’ client): Started receiver server. Checksums match. Connection initialised with: ('127.0.0.1', 55578). Checksums match. ACK packet has been received. (Console log of my UDPServerReceiver working with Seoeun’s client): Started receiver server. Checksums match. Connection initialised with: ('127.0.0.1', 55578). Checksums match. ACK packet has been received.
Test 2 (The client program MUST receive the name used by the server (typically the	All implementations had this functionality and passed the test.	The recipient’s username is successfully received from all the server implementations by my UDPSenderClient correctly and is output in the log. (Console log of my UDPSenderClient working with

username) so it can be added to the message. This will be tested by checking whether the client implementation receives the username of other server implementations, this is displayed in the console)		Eve's server): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match. Packet Type Received: recipient_username. Recipient username is: Eve. (Console log of my UDPSenderClient working with Marcus' server): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match. Packet Type Received: recipient_username. Recipient username is: Marcus. (Console log of my UDPSenderClient working with Seoeun's server): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match. Packet Type Received: recipient_username. Recipient username is: Seoeun.
Test 3 (Depending on the time, the greeting the server client starts with must be either: "Good Morning", "Good Afternoon" or "Good Evening". This will be tested by running the client programs at different times of the day and checking whether the correct greeting for the time is sent to the server implementation which should display this)	All implementations had this functionality and passed the test.	These tests were conducted at different times of the day to verify that the correct greeting message was displayed at the correct time. I have shown one case for each of the server implementations to demonstrate this however after testing it was verified that they all worked and displayed the correct greeting for the time. (Console log of my UDPServerReceiver working with Eve's client): Received Message: ----- Good Afternoon Georgia! From: Eve ----- (Console log of my UDPServerReceiver working with Marcus' client): Received Message:

		----- Good Evening Georgia! From: Marcus ----- (Console log of my UDP Receiver working with Seoeun's client): Received Message: ----- Good Morning Georgia! From: Seoeun -----
Test 4 (Clients **SHOULD** be able to enter an optional message to be attached to the body of the greeting and the greeting the server receives should display this too. To test this, the server will receive a greetings message from each client implementation and verify whether the greeting was received and displayed correctly.)	All implementations had this functionality and passed the test.	With each client implementation and added an additional message to the greeting and verified whether the additional message was present. (Console log of my UDPServerReceiver working with Eve's client): Received Message: ----- Good Afternoon Georgia! This is an example message… From: Eve ----- (Console log of my UDPServerReceiver working with Marcus' client): Received Message: ----- Good Afternoon Georgia! This is an example message… From: Marcus ----- (Console log of my UDP Receiver working with Seoeun's client): Received Message: ----- Good Afternoon Georgia! This is an example message… From: Seoeun

		-----
Test 5 (The client program **MUST** be able to recover from an error in the case that server does not respond to it within the given timeout time)	All implementations had this functionality and passed the test.	While running my UDPClientSender, I would close the server implementations and record the output on the console log. (Console log of my UDPClientSender working with Eve's client): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received. Request timed out - resending data. Unknown error occurred, moving on to next IP address. Finished sending greetings. (Console log of my UDPClientSender working with Marcus' client): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received. Request timed out - resending data. Unknown error occurred, moving on to next IP address. Finished sending greetings. (Console log of my UDPClientSender working with Seoeun's client): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection.

		ACK packet has been received. Exchanging public keys. ACK packet has been received. Request timed out - resending data. Unknown error occurred, moving on to next IP address. Finished sending greetings.
Test 6 (Invalid IPs entered in the client implementation **MUST** be caught and throw exceptions)	All implementations had this functionality and passed the test	When an IP is entered, a regex is used to check if it conforms to the IPv4 format, if not the client catches it, informs the user and moves to the next IP. (Console log from my UDPClientSender when an invalid IP is entered.): Enter your list of IP addresses you want to send greetings to with commas separating the addresses: Please enter your custom message (optional): IP you entered is not valid.
Test 7 (ACK packets **MUST** be received from the server and client after data is sent. This will be tested by verifying if my client receives ACK packets from other server implementations and if my server receives ACK packets from other client implementations)	All implementations had this functionality and passed the test.	(Console log of my UDPClientSender working with Seoun's server): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received. (Console log of my UDPClientSender working with Eve's server): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received.

		(Console log of my UDPClientSender working with Marcus' server): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message: Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received.
Test 8 (Checksums **SHOULD** be performed on the "type" and "content" properties of the JSON object. This will be tested by displaying a message regarding whether the checksum that the client receives matches the checksum that was inside the JSON object. If there is no checksum then nothing will be displayed about it)	All implementations had this functionality and passed the test.	(Console log of my UDPClientSender working with Eve's server): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match. (Console log of my UDPServerReceiver working with Marcus' client): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match. (Console log of my UDPServerReceiver working with Seoun's client): Packet Type Received: recipient_public_key. Requesting recipient username. ACK packet has been received. Checksums match.
Test 9 (If a socket times out or there is a socket error with the server, the client **MUST** attempt to send the data again to ensure the data transfer mechanism is reliable)	All implementations had this functionality and passed the test	The socket didn't time out often nor were there any socket errors but this console log comes from when there was an instance of a socket timeout with Eve's server. (Console log of my UDPClientSender working with Eve's server): Enter list of IP addresses to send greetings to with commas separating the addresses: Please enter your custom message:

		Initialising connection. ACK packet has been received. Exchanging public keys. ACK packet has been received. Request timed out - resending data. Unknown error occurred, moving on to next IP address. Finished sending greetings.
Test 10 (The client program should send a ‘fin’ packet to the server implementation to signify the end of communication. Upon receiving this ‘fin’ packet, an ACK packet is received by the client from the server and the connection is then ended)	All implementations had this functionality and passed the test	The client should send a ‘fin’ packet at the end of communication to alert the server that the communication is now finished. It should receive an acknowledgement packet back then proceed to terminate the connection. (Console log of my UDPClientSender when closing connections with other server implementations): Received Message: ----- Thank you for your greeting! ----- ACK packet received. Terminated connection with recipient - 127.0.0.1. Finished sending greetings. (Console log of my UDPServerReceiver when closing connections with other client implementations): Received Message: ----- Good Evening Georgia! Goodluck with the coursework! From: Eve ----- ACK packet received. Checksums match. Terminating connection. Cached Greetings: Good Evening Georgia! Goodluck with the coursework! From: Eve

Score: 30/30

Proprietary Extensions

Thirdly, asymmetric encryption has been implemented. It was done to secure the user's messages from the message leakage and hacking. This was done by using python's 'rsa' library. Exchanging the public keys between server and the client, the programs ensure that the message deliveries don't get interrupted by external people.

Lastly, I implemented a caching function. Caching allows the users to remind their conversation for a natural flow with the greetings. It displays the history of messages. It's implemented by creating an empty list for cache and appending the messages sent and received within the server. After the socket close, the cached messages are shown.

The first extension I thought necessary to add to the E-Greetings Card protocol was checksums. Checksums are used to check the integrity of received data - when sending greetings, it is essential that greetings are sent correctly. Checksums are added to all json packets that are sent to make sure that if data is intercepted maliciously or gets corrupted while sending, the correct data is received and if not, sent again. I implemented checksum in my program using zlib library in python.

The next extension I added was a profanity filter. With the option of user's being able to input a custom message, a profanity filter would block and filter any harmful content, protecting users. I implemented it using the better-profanity extension of python.

I also implemented asymmetric encryption to ensure user greetings that were sent or received were protected, although no protected information would be shared in the greetings, it was still effective to add to my program. To implement it, I used the rsa library in python.

The final extension I was able to implement was caching. My Greetings cache collects any received greetings in order to display them to the user once the socket connection is closed. Caching was important to implement in the protocol as after each message is received, it is overwritten by the next incoming message. It was implemented using a directory.

Extensions I would have liked to implement but could not due to time restrictions include: latency control and congestion control. Implementing latency control would work hand in hand with congestion control - ensuring that there is a minimum latency would first of all reduce congestion as more packets would be able to be sent, if congestion was to rise, the protocol would be able to slow the rate at which packets are sent, reducing the congestion.

Designing For Robustness (Postel's Law)

As stated in RFC 761, the robustness principle is how implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others. (Postel, 1980).

- Both my UDPClientSender and UDPServerReceiver anticipate errors that will arise and catches them using try and except statements. For example when attempting to send data to the UDPServerReceiver, if the socket times out or there is a socket error, the UDPClientSender will catch them, inform the user, try again and if it throws it again it will carry on to the next IP.
- Limiting what a user can input protects the program and data integrity. Where a user has to input something into the program, there are checks to ensure the data is suitable:
    - When the user inputs the list of IPs, each IP is checked to ensure it is valid.
    - When a user enters an optional message, it has limited characters and has a profanity filter applied to it.
- I Set a socket timeout to 1 second to ensure that if any errors arise with an IP address, the whole program is not affected, this is caught by the try except statements mentioned above and if the error continues, the program moves to the next IP address.
- Checksums are included in all json packets that are sent to ensure reliability of data. Successes and problems are communicated with the user using print statements. Checksums work hand in hand with ack packets - when the UDPReceiverServer receives a json packet where the checksum doesn't match, an ack packet is not sent, this is received by the UDPClientSender to resend the data.
- When a packet is received by either my UDPClientSender or UDPServerReceiver, the program confirms that the packet type received was the one expected, if not it throws an exception.

References

Postel, J., 1980. *RFC 761 - DoD standard Transmission Control Protocol*. [online] Datatracker.ietf.org. Available at: <https://datatracker.ietf.org/doc/html/rfc761#section-2.10> [Accessed 2 May 2022].

UDP Flood DDoS Attack | Cloudflare UK. (n.d.). *Cloudflare*. [online] Available at: https://www.cloudflare.com/en-gb/learning/ddos/udp-flood-ddos-attack/.

www.cisa.gov. *Understanding Digital Signatures | CISA*. [online] Available at: https://www.cisa.gov/uscert/ncas/tips/ST04-018#:~:text=Digital%20signatures%20work%20by%20proving.