Network Systems – Integration Project Report - Group 49

Group members:

- Andrei Bercea, s2170906
- Christopher Colomb, s2166429
- Maika Rabenitas, s2166410
- Patrick van Oerle, s2010267

1. System architecture and diagram *(20 points)*

The chat application allows users to converse via text with one another within a textual user interface. Only peer-to-peer messaging has been implemented. To create a reliable application which can send and receive messages, an efficient packet header structure is necessary to achieve this utility. The structure is as follows.

HEADER								
First Byte			Second Byte					
Source Address	Destination Address	Sequence/Ack Number	Unassigned	Ack Flag	Token Passing	Offset	Last Message
2 bits	2 bits	4 bits	2 bits	1 bit	2 bits	2 bits	1 bit

As two bytes are designated for the header, the remaining fourteen bytes of the packet are for the payload length. To maintain the efficiency of the chat application, the user is allowed to send a maximum of four packets for a single message. Therefore, the message itself consists of fifty-six bytes.

The class diagram can be found on the last page.

2. Description of system and its functionalities *(40 points)*

2.1. Medium access control

We tried to implement a token passing system. However, due to the lack of time and higher priorities, we were not able to implement this. In the beginning of the program, we did make a start by passing on the token. However, some simple checks whether the user that wants to send data actually has the token resulted in a lot of bugs, including nodes being out of range taking the token with them. For this reason, we decided to move on. At this moment, our access control can best be described as Aloha. A node just sends when it has data to send and hopes no collisions will take place.

2.2. Addressing

The addressing is implemented in a way where four users have a fixed ID. This makes the size of the header smaller, which gives more room for the payload length. We have also created an array of the users' names so that instead of identifying users by their ID numbers, a sender can use their own names. This way, when transmitting a message, the sender can address their recipients by their names, which feels more user-friendly.

2.3. Reliable data transfer

For each packet sent, the application checks to see if the message is intended for whatever host. If the host ID matches with the destination ID portion of the header in the received packet the packet will be digested to retrieve the message.

When a packet is ready to be sent (the header and the corresponding payload have been added), it is put into the sending queue. A host, at some point, retrieves that packet and starts the defragmentation process. At the same time, the receiver starts building the ACK packet corresponding to the DATA packet received (i.e. if the DATA packet is indeed meant for this host; otherwise, it is retransmitted). The source and destination IDs, together with the acknowledgement number, are kept in the ACK packet as well. This identification process of an ACK packet follows similar procedures as a DATA packet. In fact, the only difference is that an ACK packet will have a flag (acknowledgement flag) set on one bit of its second byte. This way, a host can distinguish between a DATA and an ACK packet (an ACK packet is also only two bytes long - the header -, but we considered the acknowledgement flag as an extra precautionary measure). In its second byte, the offset also gets copied from the DATA packet into the ACK (this is needed so that a host knows which part of the message it received/acknowledges).

2.4. Ad-hoc multi-hop routing & forwarding

Due to the ‘flooding’ nature of our Reliable Data Transfer Protocol (packets are broadcasted), a host retrieves a packet and checks its header. If the message is intended for him/her, then the packet is kept. Otherwise, it is put back into the sending queue, waiting to be retrieved by another host. This goes on until the packet is retrieved by the intended destination of the packet. The exceptional case (i.e. when the packet is a duplicate) is also considered, such that the packet is dropped (on the recipient’s side). This is done by setting sequence numbers in the header. When a packet with an already acknowledged sequence number is received, the packet is simply dropped.

3. Test setup and scenarios *(20 points)*

The first issue that needed to be addressed was formatting the first two bytes of a packet (the header of both DATA and ACK packets). Since a lot of bit shifting was involved in creating the headers (putting together the hosts’ IDs, sequence/ acknowledgement numbers, offset and flags), the code was very error-prone, so a lot of debugging was involved. To test this, we decided to make print statements after every bit operation.
The second issue was the Reliable Data Transfer Protocol. Solving the problems that arose with this protocol was a bit more tricky. In addition to print statements, we also

used the debugger embedded within Eclipse, but the web platform provided to us was the one that really helped us understand what was happening between the nodes.
In terms of latency and number of connections supported by a single chat session, our application can be considered as ‘naive’. The reasons why will be discussed in the below section.

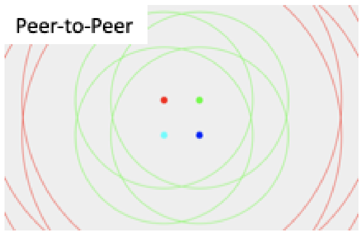
4. Test results *(20 points)*

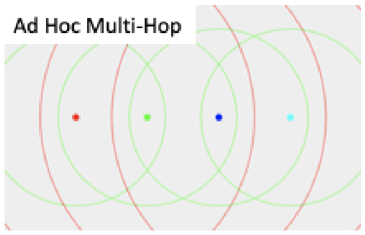
When testing/debugging the Reliable Data Transfer Protocol, the possible scenarios have been discerned:

- best case scenario: host 1 sends DATA packet; host 2 receives DATA packet; host 2 sends ACK packet; host 1 receives ACK packet;
- DATA packet lost: host 1 sends DATA packet; host 2 does not receives DATA packet; host 2 does not send ACK packet; host 1 retransmits DATA packet (the same one) after a timeout;
- ACK packet lost: host 1 sends DATA packet; host 2 receives DATA packet; host 2 sends ACK packet; host 1 does not received ACK packet; host 1 retransmits DATA packet; host 2 receives duplicate DATA packet, drops it and retransmits ACK packet;

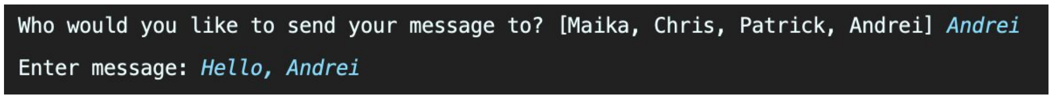
It is naive when latency is concerned mainly because all messages are broadcasted. This may at first seem like it has nothing to do with latency, but if the chance of packets getting lost is high, the system will go through a lot of timeouts, thus leading to significantly high value of latency. It may appear like this could be solved by reducing the timeouts, but if we were to do this, then ACK packets would not get back to the sender host in time, thus leading to further issues.
On the other hand, the app is ‘naive’ due to the static nature of how resources are allocated (i.e. an array of fixed length is constructed, with the IDs known beforehand). In our case, the array of IDs looks as follows: [“Maika”, “Chris”, “Patrick”, “Andrei”], with the corresponding ID given by the position in the vector (Maika - host 1, Chris - host 2, Patrick - host 3, Andrei - host 4).

The following hosts’ layouts were successfully tested.

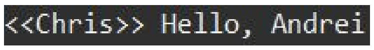




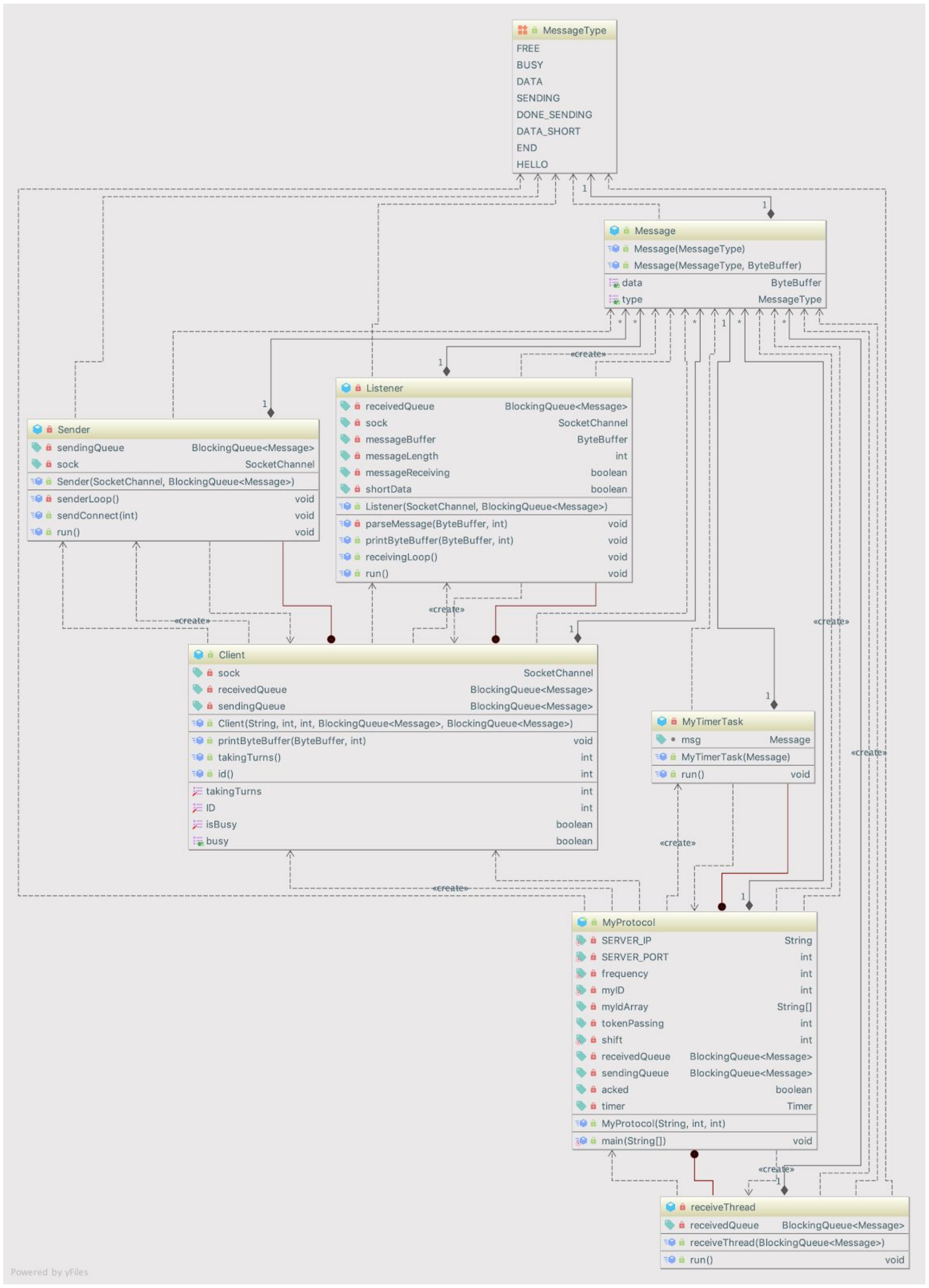
Console print of message being successfully delivered:



```
Who would you like to send your message to? [Maika, Chris, Patrick, Andrei] Andrei
Enter message: Hello, Andrei
```



```
<<Chris>> Hello, Andrei
```



Class Diagram of the Application