Networking Technologies and Management Systems II Programming Project

In order to put the concepts learned in the course into practice, the programming project aims at implementing a simple messaging protocol for chat applications built on top of UDP. This protocol (that we will call Simple IMC Messaging Protocol, or SIMP for short) could in theory be used by a third-party to implement a chat program at the application layer level.

SIMP Specification

SIMP represents a lightweight protocol. As such, it won't need all connection-oriented functionalities provided by TCP. Therefore, it was decided to implement it on top of UDP. Nevertheless, SIMP needs some functionality for ensuring that messages are delivered – otherwise, user messages could get lost on its way. The protocol operates in the following way: first, any user may contact another user for starting a chat. Users are identified by their IP address and port number (of course, every user needs to run a SIMP implementation in order to get contacted in the first place). The user can accept or decline the invitation to chat. After accepting the invitation, both users can send messages to each other until one of them closes the connection. If a user already in a chat gets an invitation to chat from another user, that invitation will be automatically rejected by sending an error message (user is busy in another chat).

Types of messages

More specifically, the SIMP protocol is defined by the following message types:

- Control messages: these are used to establish and terminate a connection or resend messages after a timeout has occurred. These messages are identified by the `Type` field in the header (see next section) with a value of `0x01`.
- Chat messages: used for implementing the conversation itself between the users. These messages are identified by the `Type` field in the header (see next section) with a value of `0x02`.

Header format

Each message in the SIMP protocol is composed by a header and a payload. The header includes meta-information about the contents of the message itself, like type, sequence number, and other parameters. The payload is mainly used to carry the contents of the chat. **Note:** all strings will be encoded using ASCII characters.
The header is composed by the following fields:

1. `Type` (1 byte): the type of the message as outlined before. Possible values:
   - `0x01` = control message.
   - `0x02` = chat.
2. `Operation` (1 byte): indicates the type of operation of the message. Possible values:
   - If `Type == 0x01` (control message):
     - `0x01`: `ERR` (some error condition occurred).
     - `0x02`: `SYN` (used in sliding window algorithm).
     - `0x04`: `ACK` (used in sliding window algorithm and as general acknowledgement).
     - `0x08`: `FIN` (used to close the connection).
   - If `Type == 0x02` (chat): field `Operation` takes the constant value `0x01`.

3. `Sequence` (1 byte): a sequence number that can take the values `0x00` or `0x01` used to identify resent or lost messages.
4. `User` (32 bytes): user name encoded as an ASCII string.
5. `Length` (4 bytes): length of the message payload in bytes.
6. `Payload`: depending on the field `Type`:
   - If `Type == 0x01` (control message) and `Operation == 0x01` (error): a human-readable error message as an ASCII string.
   - If `Type == 0x02` (chat): the contents of the chat message to be sent.

Operation

The protocol begins by establishing a connection using the three-way handshake algorithm:

1. Sender begins by transmitting a `SYN` control message to the destination.
2. Receiver replies with another control message with a combination `SYN + ACK` (that's the result of an bitwise OR from the values of `SYN` and `ACK`).
3. Sender replies with `ACK`.

If the receiver wants to decline the connection, Step 2 is replaced by responding with a `FIN` control message. After this connection establishment phase, sender and receiver can begin with their conversation (type of message: chat). The sender of a given message will use the stop-and-wait strategy to wait until the message is acknowledged from the receiver. In case it does not receive a reply after a specified timeout (default: 5 seconds), it will retransmit the message with the same sequence number. After the `ACK` comes, it will transmit the next message with the next sequence number (0 or 1). `SYN` messages from other users will be declined by sending an `ERR` control message with the error message: "User already in another chat" and a `FIN` control message. If a participant wants to leave, it will send a `FIN` message to the other user, who will send an acknowledgement (`ACK`) before leaving the conversation.

Submission and details

This project will be completed in groups of two people and delivered on **18/01/2023 23:59** at the latest.

All projects will be handed in via MS-Teams and be composed of a ZIP-file containing:

1. Implementation in Python:
   - `simp_server.py`: will start the server using the IP address and port number given as parameters.
     - The user will enter their user name right after starting the server and then wait for incoming connections.
     - As soon as a connection arrives, the server will output the origin (IP, port) of the incoming connection and the contents of the message.
     - The user will then be able to enter a reply that will be sent to the other end of the conversation.
   - `simp_client.py`: will start the client that connects to the server using the IP and port number given as parameters.
     - The user will enter their user name right after starting the client.
     - If the connection with the server gets accepted, the user will then be able to type a message that will be sent to the server. Otherwise, the client will output the error message and exit.
     - The connection can be closed by both sides when typing the command `quit`.
   - Note that each participant has to wait for a reply in order to type the next message (i.e. it's a synchronous conversation).
2. Technical documentation.

Assessment

The assignment is worth **50 points** that will be given using the following criteria:

- Correct implementation of the message (header + payload): 25 points.
- Correct implementation of three-way handshake: 10 points.
- Correct implementation of stop-and-wait: 10 points.
- Clean code and clear documentation: 5 points.