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 instead. However, SIMP needs some functionality for ensuring that messages are delivered – otherwise, user messages could get lost on its way and never be recovered. The protocol should operate 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 datagrams

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

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

Header format

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

1. `Type` (1 byte): the type of the datagram as outlined before. Possible values:
   - `0x01` = control datagram.
   - `0x02` = chat datagram.
2. `Operation` (1 byte): indicates the type of operation of the datagram. Possible values:
   - If `Type == 0x01` (control datagram):
     - `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 datagram): 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 datagrams.
4. `User` (32 bytes): user name encoded as an ASCII string.
5. `Length` (4 bytes): length of the datagram payload in bytes.
6. `Payload`: depending on the field `Type`:
   - If `Type == 0x01` (control datagram) and `Operation == 0x01` (error): a human-readable error message as an ASCII string.
   - If `Type == 0x02` (chat datagram): the contents of the chat message to be sent.

Operation

The protocol begins by establishing a connection using the three-way handshake algorithm as seen in class:

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

Implementation

A SIMP implementation is composed of a **daemon** and a **client**:

- The **daemon** is a program that runs in the background and waits for incoming connections. Before being able to run a chat, every user needs to have the SIMP daemon up and running. The daemon will be started using the IP address of the server as a command line parameter. Communication using the SIMP protocol as described previously **always** works from daemon to daemon. The port number used for daemon-to-daemon communication is standarized and always set to 7777. The port number used for communication between client and daemon is always 7778.
- The **client** is a text-based program that is used by the end-users to chat. It will accept the IP address of the daemon as a command line parameter and, if the daemon is correctly running, it will ask the user for a username. Then, one of the following conditions will hold:
  1. The daemon responds that a conversation request is pending. In this case, the client will show the IP address and the username of the user requesting the chat and will ask the user to decide whether to accept or decline the chat request (IP address and username of the remote user will be provided by the daemon to the client).
  2. If there is no chat request pending, the user will be asked if they want to start a new chat or wait for chat requests. In the first case, the client will request the IP address of the user to send the chat request to. In the latter case, the client will enter an idle state where it waits for incoming chat requests. As soon as a chat request arrives, the client will proceed as in case 1.
  3. The user will always have the option to quit (disconnect from the daemon) by pressing the key "q".

Remark 1: Note that the client and the daemon are *separate* programs that run independently. Therefore, you will need to implement your own communication protocol between the client and the daemon that implements the following operations: `connect` (to establish a connection between client and daemon), `chat` (to send chat messages to the daemon) and `quit` (disconnect from the daemon) operations.

Remark 2: Remember that in this protocol the chat operation is synchronous. Once a chat message is sent, the sender waits for a reply from the other user indefinitely.

Submission and details

This project will be completed in groups of two people and delivered on **19/12/2024 23:59** at the latest.

All projects will be submitted via MS-Teams as a ZIP-file containing:

1. Implementation in Python:
   - `simp_daemon.py`: implementation of the daemon as described previously.
   - `simp_client.py`: implementation of the client as described previously.
   - `requirements.txt`: requirements file that can be used with `pip` to install all dependencies.
   - Any other auxiliary files/modules needed.
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): 15 points.
- Correct implementation of three-way handshake: 10 points.
- Correct implementation of stop-and-wait: 10 points.
- Correct implementation of the communication between daemon and client: 10 points.
- Clean code and clear documentation: 5 points.