P2P Chat Application
System Functional Specification

CMP2204 Term Project Spring 2025

1 System Definition

Our P2P chat application consists of two major phases: *(i)* discovering all available users in the network, and *(ii)* initiating conversations with different users in the network. Figure 1 demonstrates the four processes that together make up the chat application; the left two processes correspond to the discovery of peers, and the right two processes correspond to the conversations among pairs.

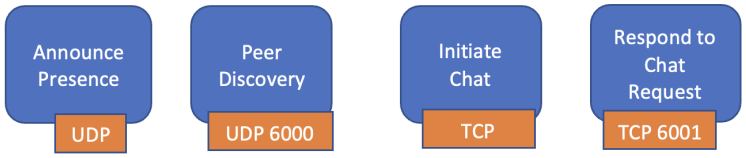

Figure 1: The four processes of the peer-to-peer chat application

1.1 Operational Scenarios

The following are the use cases supported by the P2P Chat Application:

Service Announcement - Peer Discovery: Upon connecting to the Local Area Network, every peer starts to *periodically, every 8 seconds* broadcast their presence. They insert their name in the message in JSON (JavaScript Object Notation) format. (Figure 2).

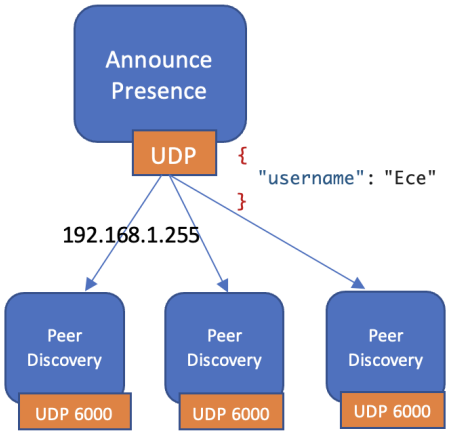

Figure 2: Peers broadcast their presence to all nodes in the network.

Upon connecting to the Local Area Network, every node also starts listening for peer announcements in the LAN. Upon hearing an announcement, the payload is parsed, and the ID of the peer is stored in a local dictionary, containing also the timestamp of when this user was last seen.

Viewing available users: The end user can view the list of available users in the network. The list only contains the users that were discovered in the last 15 minutes (900 seconds). If a user (*i.e.*, user's periodic announcements) have not been heard from within the last 10 seconds, the user's

name is displayed with "(Away)" notice next to it. Otherwise, the user's name will have "(Online)" mark next to it.

Initiating chat and ensuring message integrity: The end user specifies one username to chat with. The chat process (TCP client) looks up its local dictionary to retrieve the user's IP address. Then it (automatically, without user intervention) initiates a TCP session with the corresponding IP address, encrypts the user-typed message and sends it over this TCP connection. After this, TCP session is closed.

Receiving a message: When a new TCP connection request is received, this connection request is immediately accepted, the received message is decrypted and displayed. The sender's name (and not their IP address) must be displayed with the message, so that the end user can figure out whom the message is from.

Security: TCP sender and receiver can generate a shared secret key using the Diffie-Hellman key exchange algorithm, and use that key to encrypt exchanged messages.

Chat history: End user can view the chat history (timestamp, name of peer, IP address of peer, sent/received, text).

2 Requirements

Throughout this section, the term **shall** indicates an obligatory requirement that must be met to comply with the specification, whereas the term **may** indicates an item that is truly optional.

2.1 Service Announcement Requirements

Req. #	**Requirement**
2.1.0-A	When launched, Service_Announcer **shall** ask the user to specify their username, and must store it locally.
2.1.0-B	After storing the username, Service_Announcer **shall** start to periodically send broadcast UDP messages in the network to announce its presence. The period **shall** be once per **8 seconds**. For the broadcast IP address, please use 192.168.1.255
2.1.0-C	Service_Announcer 's periodic broadcasts **shall** contain a JSON including the username specified by the end user and the IP address. It is very important that the field name is exactly "username" and the format is a valid JSON format; otherwise you may have parsing issues when working with your peers. An example would look like: '{"username": "Ece"}'.

2.2 Peer Discovery Requirements

Req. #	Requirement
2.2.0-A	Peer_Discovery **shall** listen for UDP broadcast messages on port **6000**.
2.2.0-B	Upon receiving a message, Peer_Discovery module **shall** parse the message contents using a JSON parser in Python. The obtained *username* and the IP address retrieved in the UDP recvfrom() call **shall** be stored in a dictionary. The dictionary keys **shall** be the IP address (that you fetched using recvfrom()). This dictionary **shall** be shared with the Chat_Receiver process. You **may** store it in a local text file that is shared between the Peer_Discovery and Chat_Initiator components.
2.2.0-C	Upon receiving a broadcast message, if the user IP exists in the dictionary, then the corresponding timestamp (when this neighbor was last heard from) **shall** be updated. If the dictionary does not contain an entry for the sender's IP address, a new entry **shall** be added to dictionary.
2.2.0-D	Upon every insertion into the dictionary, Peer_Discovery module **shall** display the detected user on the console (*e.g.,* "Ece is online"), which would provide better user experience. This would also help you with debugging.

2.3 Chat Initiation Requirements

Req. #	Requirement
2.3.0-A	When launched, Chat_Initiator **shall** prompt the user to specify whether they would like to view online users, initiate chat, or view chat history. The user interactions and GUI (you can assume Terminal/console) that you prepare is up to you. User **shall** be able to specify "Users", "Chat", or "History".
2.3.0-B	When the end user specifies "Users", Chat_Initiator **shall** display the list of discovered users whose broadcast messages have been received within the last **15 minutes**. If a broadcast message from the user has been received within the past **10 seconds**, the GUI **shall** display *_username_ (Online)*, *e.g.* Alice (Online). If a broadcast message from the user has *not* been received within the past **10 seconds**, the GUI **shall** display *_username_ (Away)*, *e.g.* Bob (Away).
2.3.0-C	[**Secure Chat**] When end user specifies "Chat", Chat_Initiator **shall** ask the user to enter the name of the user to chat with. When the user specifies a username, Chat_Initiator **shall** ask the user whether they'll chat securely or not. If the user prompts secure, Chat_Initiator **shall** receive one number from the end user and **shall** initiate a TCP session with the IP address of the specified user and send the typed number. The message **shall** contain a JSON including the key, the JSON **shall** look exactly like this: '{"key": "*XXX*"}' (with *XXX* replaced by the number). The TCP session will persist (*i.e.,* do not close the TCP session after the number exchange. The end user will send another number as well; Chat_Initiator **shall** take that number and use these numbers to generate a shared key (using Diffie-Hellman key exchange mechanism) to encrypt exchanged text messages (use p=19 and g=2). The Chat_Initiator **shall** not do anything else, allowing the end user to type their message. When the user presses Enter, Chat_Initiator **shall** encrypt (using Python encrypt library) the entered message and send it to the end user, in a JSON message that looks like this: '{"encrypted_message": "*YYY*"}' (with *YYY* replaced by encrypted text). The Chat_Initiator **shall** then close the TCP session.
2.3.0-D	[**Unsecure Chat**] When end user specifies "Chat", Chat_Initiator **shall** ask the user to enter the name of the user to chat with. When the user specifies a username, Chat_Initiator **shall** ask the user whether they'll chat securely or not. If the user prompts unsecure, Chat_Initiator **shall** not do anything else, allowing the end user to type their message. When the user presses Enter, Chat_Initiator **shall** send the entered message to the end user, in a JSON message that looks like this: '{"unencrypted_message": "*YYY*"}' (with *YYY* replaced by encrypted text). The Chat_Initiator **shall** then close the TCP session.
2.3.0-E	Upon sending a message, Chat_Initiator **shall** log the sent message in a log under the same directory, along with timestamp and the username to whom the message was sent.
2.3.0-F	If Chat_Initiator attempts to initiate a TCP session with the end user but cannot initiate it, it **shall** display an error message indicating a connection with the end user cannot be established.
2.3.0-G	When end user specifies "History", Chat_Initiator **shall** display a log including timestamp, username, message content, and whether it was SENT or RECEIVED. To be able to do this, both Chat_Initiator and Chat_Responder **shall** write into a common log file.
2.3.0-H	After a TCP session is closed, Chat_Initiator **shall** persist; the service **shall not** terminate.

2.4 Chat Responder Requirements

Req. #	**Requirement**
2.4.0-A	Chat_Responder **shall** listen for TCP connections on port **6001**.
2.4.0-B	Chat_Responder **shall** accept TCP connection request before it times out.
2.4.0-C	Chat_Responder **shall** parse the JSON in the message to learn whether a *key* is being exchanged or a *message* (*encrypted/unencrypted*) is being received. For this, it **shall** parse the JSON message. *(i)* If the parsed message has a JSON key of "key", then the Chat_Responder **shall** generate its own key and send it. *(ii)* If the parsed message has a JSON key of "encrypted_message", Chat_Responder **shall** decrypt and display it on the console. *(iii)* If the parsed message has a JSON key of "unencrypted_message", Chat_Responder **shall** display the message contents on the console.
2.4.0-D	Upon receiving a message, Chat_Responder **shall** log the message in the log . Each entry **shall** specify timestamp, sender's username, and the message, marked as "SENT".
2.4.0-E	After a TCP session is closed, Chat_Responder **shall** persist; the service **shall not** terminate and **shall** continue to listen on port 6001.
2.4.0-F	Chat_Initiator and Chat_Responder **may** be implemented as different threads in the same process. In that case, end user may both write messages to send, and view messages received, in one (same) console window.