1. A video (at maximum 10 minutes) in which you explain how the code works, show the desired output, and explain how did you do that in the code.

2. A freely worded report (doc, pdf), which presents the system's architecture along with necessary explanations/assumptions considered during the design/development. The grading will be based on the following:

**Overview**

I made the chat messaging application using the JavaScript programming language and I used the socket.io library for creating the connection and communication between client and the server. Socket.io JavaScript library works using the WebSocket protocol and it is done using a TCP connection, which means that creating a communication lane requires the typical three-way handshake between the client and the server to establish a continuous connection, which will be used for all data transfers until the transmission is eventually ended abruptly, because of a server or client going offline, or alternatively in a controlled manner initiated by the client [2.]. The server is expected to stay online at all times, so it is not expected to initiate a connection ending signal to the clients.

**How node.js handles threads and concurrency**

Node.js usually runs programs in a single thread, unless multi-threading is implemented with worker threads. Node has a main event loop which processes all of the executed processes in a fixed sequence starting from the first process to the last and the order is fixed. Single-threaded processing is useful, since there is no need to do complicated programming of synchronized threads to do processes at the same time. According to Cieślar’s article on Node.js, there is a possibility to implement worker threads to do something similar to multi-threading for JavaScript [1], but I didn’t try implementing it as my program was still an entry level program for a simple communication app.

From what I understood from the [socket.io documentation](https://socket.io/docs/v4/), the clients and the server all run single-threaded in their own threads, so there is no multi-threading, but each device has their own thread which they will use for their processing [2]. The server will be the most loaded, as all of the clients will be sending messages to the server, and responses will need to be sent to each client. The clients only need to handle their own requests and messages the other clients are sending. This means that there will be a need to make sure that the server is able to handle all of the load and more processing power or memory will need to be allocated to it in case the present resources aren’t enough.

**How the connection is maintained**

Socket.io uses a TCP connection, which means that the connection between the clients and the server will stay alive as long as the connection termination isn’t initiated by one of the systems. After the three-way handshake to establish the connection, the client and server can communicate freely to send data between each other, and after the connection between them needs to end due to the user leaving, the TCP connection termination happens. The connection termination can be initiated by either of the parties by sending a data packet with the FIN-flag set, and there is a two-way termination to end the connection. Both of the parties need to acknowledge to each other that the connection has ended, so that data packets aren’t unnecessarily sent to a network and wasted since there will be no one receiving the packets.

**Transparency**

The distributed system’s access transparency is done by hiding information about how data is stored and transmitted from the user to the server, and all the operations that the user doesn’t need to see are hidden from them. For example, when sending a message, the user isn’t able to know in what form the message is sent to the server and where the data is handled and stored. The user receives messages from the server and the messages are printed out in formatted form in the terminal. All the commands create calls to different modules in the server socket, but the user doesn’t know about it since all of the operations are done behind the scenes.

In the system there is also location transparency, as the user doesn’t know in detail where the server or other clients reside. The user connects to the server by an IP-address, but the physical location of the server is hidden from the user and the server could be located anywhere and it can change location at any time.

**Scalability**

According to the socket.io documentation, the amount of memory used on the server scales linearly to the number of clients connected and the number of messages sent. This means that the server is able to maintain the amount of connections as long as there is enough memory on the server to serve all of the clients simultaneously. It is stated that using the socket.io library, there is a possibility to scale the server to multiple nodes so that the server can have more memory to serve more users. For my application I only created a single server to serve all of the clients, but you could create more as long as you have the necessary skills to implement them. I am still new to this library, so I didn’t look into creating a multi-server application for this assignment.

**Failure handling**

I implemented some error handling into the application in cases where an error has a possibility of occurring. There is most likely still other errors that could be handled but I created handlers for the ones that I could think of and encountered while programming the application.

In case the application isn’t run using the correct syntax, which includes the IP-address and a port number, there will be an error message output into the terminal with the correct syntax with the necessary information: ‘Correct usage: node client [IP-address] [port-number]’

There is error handling on the client side in case there is an error connecting to the server due to a wrong address and/or port, or some other errors, in which case there will be an error message output into the terminal and the program exits in a controlled manner.

There is also error handling for the user input if:

* Message sent into the text chat is empty, in which case it won’t be sent
* User tries to direct-message with a username that doesn’t match any currently connected to the server, or tries to direct-message themselves which isn’t permitted
* User tries to change their username to one that’s empty, or is the same one which they’re already using

In case the user exits the program in a different method from typing /leave into the chat, the exit of the user will still be registered on the server and a message is sent to everyone. This might happen because the user closes or exits the terminal, or they use the keyboardInterrupt input CTRL+C.

Description and explanation in the video about how the system works (2.5)

5. Quality and completeness of the report (design exercise) (2.5)

The TA can assist you the best when working with JavaScript or Python. TCP/UDP connections and multiprocessing are a part of virtually all programming languages, such as Java, C#, C, C++, but they may be easier to implement on some tools than on others. This task involves two core concepts: Multiprocessing and Network connections. With JavaScript, client-server architectures are typically handled with Node.js. Node.js handles threading differently than typical servers, so be prepared to explain on video how the node achieves this. Other programming languages, frameworks, and toolkits are also allowed. You must however explain on the video how the server manages multiple clients - via threads or otherwise - and how the connection is maintained. The video should also explain why the connection is TCP or UDP.

**Sources**

[1] <https://blog.logrocket.com/complete-guide-threads-node-js/>

[2] <https://socket.io/docs/v4/>