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:

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. The server is expected to stay online at all times, so it is not expected to initiate a connection ending signal to the clients.

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 with connecting to the server due to a wrong address or other errors, in which case there will be an error message output into the terminal and the program exits in a controlled manner.

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

4. Explanation of how transparency, scalability, and failure handling have been catered in the solution (you may comment on that in the video) (3)

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.