CT30A3401 Distributed Systems Assignment 1 documentation

Scalable chat application

Preface

The application designed and implemented in this assignment was a multi-user chat system using sockets over TCP. I implemented the assignment using Node $v.17.6.0$ without any additional npm-packages. Networking was implemented using Node's built-in net -module and vertical scalability with the cluster -module.

For the client-side program, I developed a lightweight CLI-application and focused more on the backend. Building a client with a good GUI is outside the scope of this assignment, and I suspect paying more attention to the context of distributed systems was the idea here.

The server-side program is a TCP Node server running on port 3000. Vertical scalability was implemented by spawning one master process that distributes requests to child processes called workers. The master process also handles operations that should apply between all processes (such as reserving nicknames for clients) and communication between the master and workers. A single worker can handle multiple client-server TCP connections due to the non-blocking nature of Node and the number of workers can be increased by adding more logical processors to the system thus making the server vertically scalable.

Assumptions

- Application security was not considered too much (i.e., there is no authentication, DDoS safeguards etc.). Some basic validation is performed, and malformed requests are ignored. Server also registers sockets with nicknames handed out after initializing the connection between the client and server and thus spoofing is not as trivial.
- User experience on the client side was not the main concern of this assignment.
- The system is not (and does not need to be) horizontally scalable at least as is.
- There is no need for persistent storage. Clients retain messages sent to them, and messages of channels they have subscribed to, only for the runtime duration.

Abstract architecture

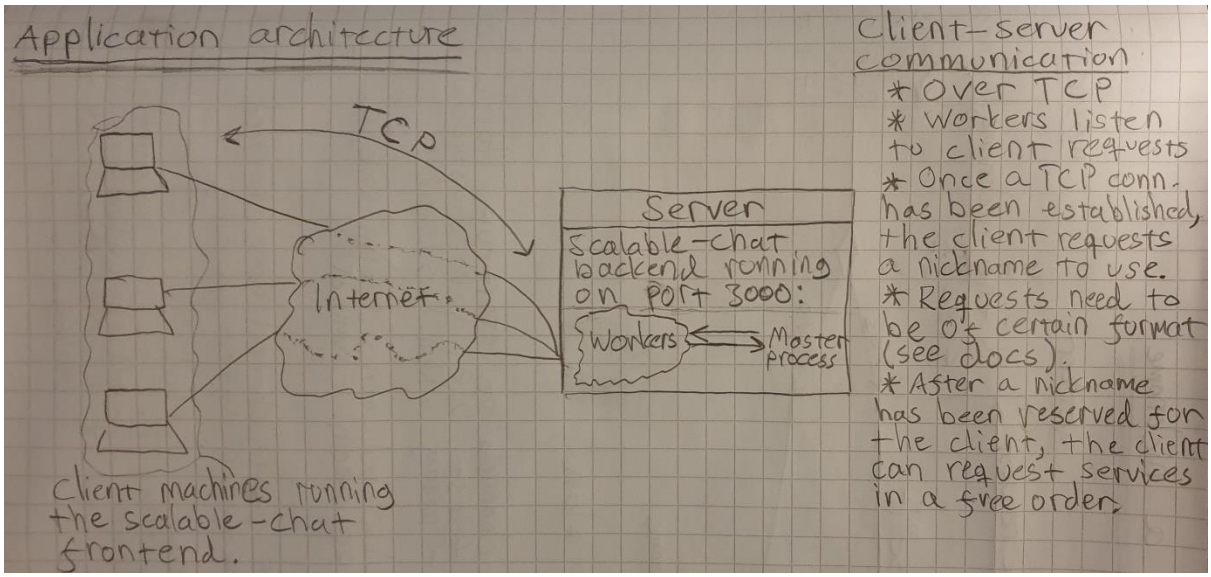

Figure 1: Overall application architecture

The overall abstract application architecture is depicted on Figure 1. The depiction is very simple, and such is the true architecture as well. The true multiprocess nature of the server is transparent to the client because the server's workers share the port 3000 and thus offer a uniform interface for the client program to connect to. Communication is done by sending JSON over the TCP connection between the client and the server (the JSON-strings need to be encoded with utf-8, see also later in the document how different services can be requested by the client).

Installation and usage

Installation is extremely easy, you only need to install Node $v17.6.0$ on your system. Download the repository's contents to a path of your choosing. Run either the client or server program with Node (for example on Windows, open up CMD, cd into the client directory and run the command $node\ client$ to start the client application).

See the video demonstration (check the readme of the repository) for usage instructions.

(Next page)

Initializing the connection

The client program initializes the connection with the server by first establishing a TCP connection between the client's arbitrary port and the server's port 3000. The server's IP-address is gotten as user input at the client program and hardcoded in the server source code (as a global variable $HOST\_IP$ at the top of the server.js file). After the TCP connection is set up, the client requests the server to determine whether the user's given nickname for the client is available on the server. If the server determines it is available, it reserves the nickname for the client if the client is connected.

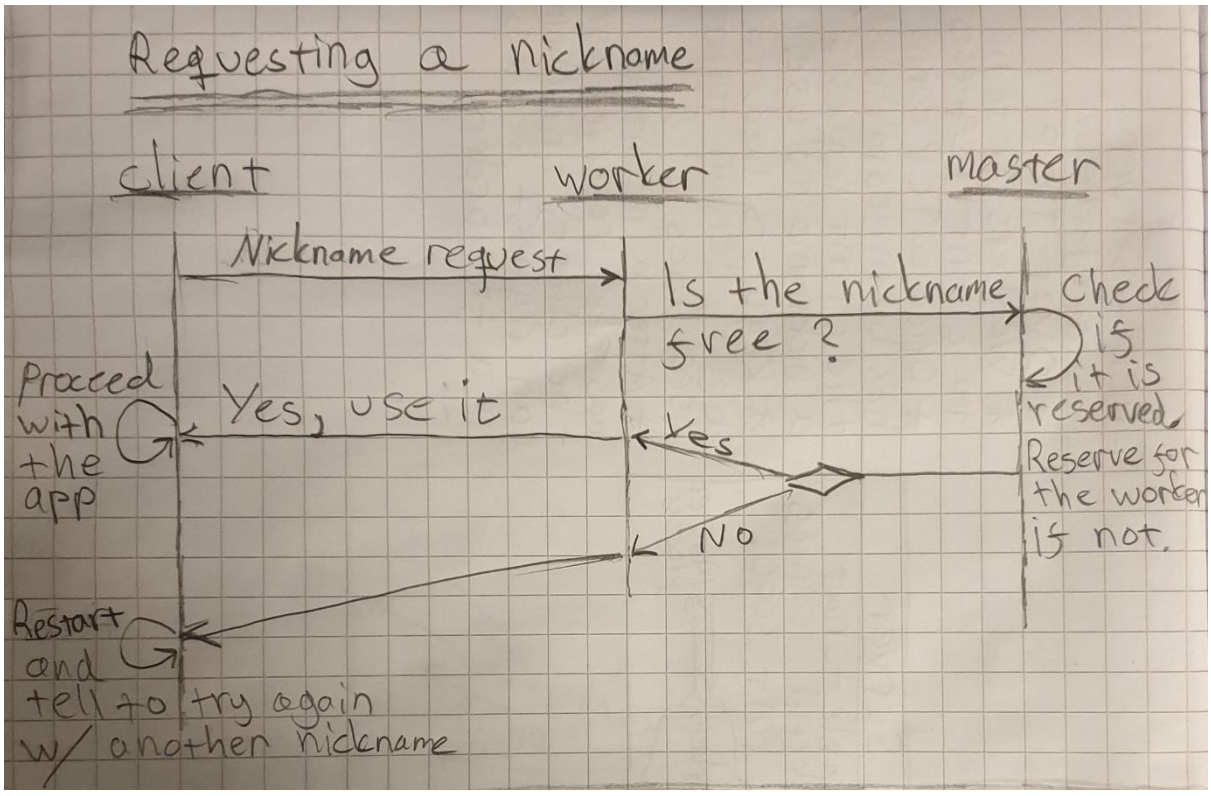

Figure 2: Sequence diagram of requesting a nickname from the server

Communication after initializing the connection

After initializing the communication, the client can ask the server to send private messages to other connected clients, subscribe to channels that the client has not already subscribed to (client program starts recording messages sent to the subscribed channels for the runtime's duration and the server starts sending them only after subscribing) and to send messages to channels the client has subscribed to.

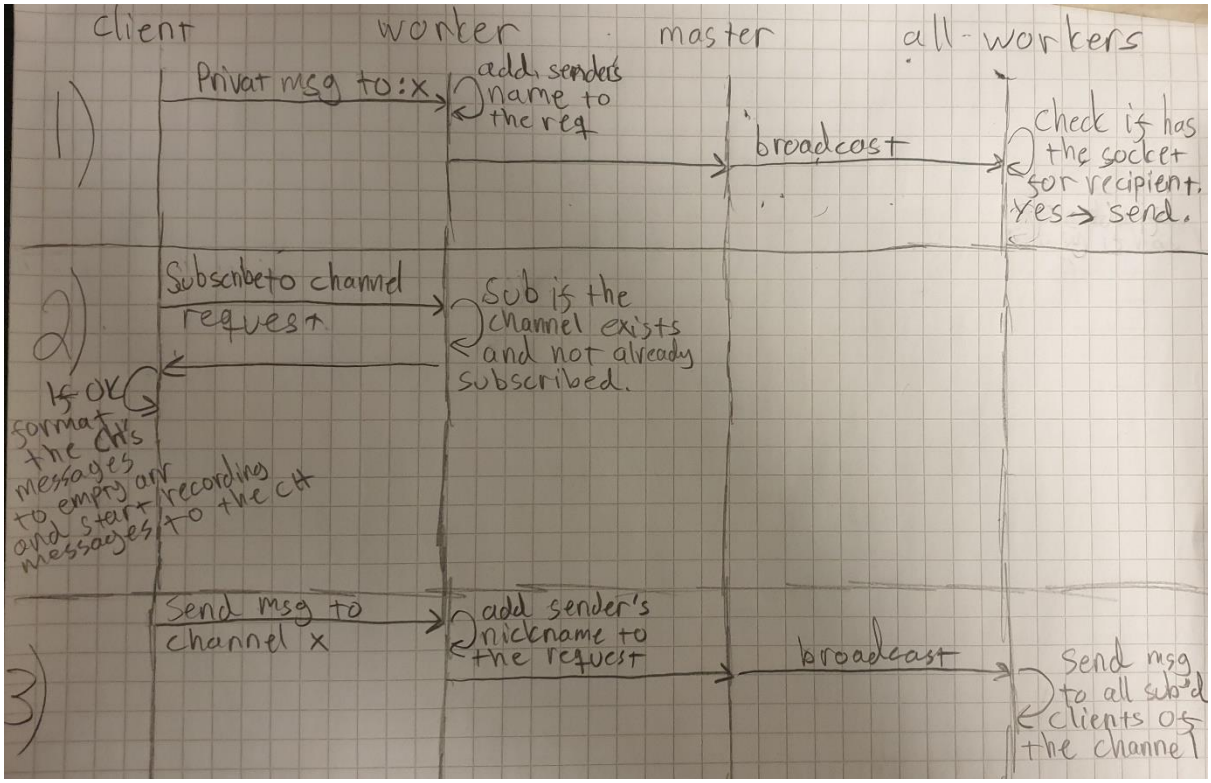

Figure 3: Communication after connection has been initialized

Transparency, scalability, and failure handling

As mentioned before, the server application can be considered transparent, as the multi-process nature of the backend offers a uniform TCP interface for the client program to connect to (i.e., the client sees the backend as a single system). Furthermore, the backend is vertically scalable as adding more physical memory (RAM) and logical processing units to the host machine will increase the performance of the application. As for failure handling, if one or more of the server's workers die, the whole system does not die. Clients may also loose connection abruptly and the server releases the resources it held for them and continues normally. If the server completely "dies" then obviously the system can not be used but one could setup backup servers with some clever configurations.