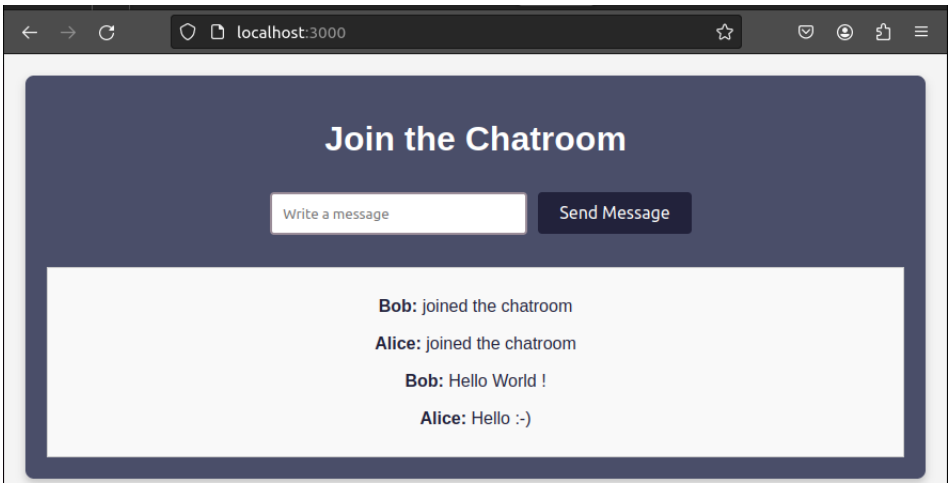
Chatroom Application using Erlang

IN.5022 Concurrent and Distributed Computing

Antoine Tissot and Milena Kovatsch

December 21, 2023



1 Introduction

The rapid evolution of chatroom applications, exemplified by platforms like Discord, Microsoft Teams, WhatsApp, Telegram etc. highlights the increasing demand for real-time communication, in both professional and private contexts. Understanding the implementation of such services is thus valuable in the computer science education as it offers a hands-on exploration of concurrent programming and real-time communication protocols.

Against the backdrop of these developments, the final project in the lecture on distributed and concurrent computing offers a useful opportunity to learn more about this topic. The knowledge of the Erlang programming language acquired in this course can be put to good use and further linked with additional technologies. When implementing a chatroom application, the basic structure with a frontend and backend requires dealing with various necessary components.

In this report, the implementation and the results of our chatroom application based on Erlang are discussed. The first section provides a brief outline of current developments in chat applications and existing technologies that are useful for implementation. The second section shows the implemented solution. According to the structure of our chatroom application, the use of the different pieces is discussed. Subsequently, the implemented solution is evaluated in a further section and discussed with regard to possible extensions and improvements. Finally, the whole project is outlined and reflected upon in a last concluding section.

2 State-of-the-Art

In recent years, the landscape of chatroom applications has quickly developed, driven by the growing need for real-time communication in different contexts. Notable examples include Slack, Discord, and Microsoft Teams, which have become integral in professional collaboration. Additionally, WhatsApp, Telegram, and Signal have redefined personal messaging with a focus on security and privacy.

Our project aimed to explore the essentials of application structure and communication components. We chose Erlang for its strengths in concurrent systems and real-time communication, creating a basic chatroom application. With regard to an Erlang-based chatroom, we can fall back on features that have already been developed. The Cowboy HTTP server was developed 10 years ago and has been updated ever since. Cowboy also supports the Websocket protocol, which has also been around for more than 10 years and enables a persistent bidirectional connection. This feature makes the Websocket protocol particularly interesting for applications that have to run in real time. Looking at the other side, frontends are the face of digital experiences. When it comes to building user interfaces, React stands out for its simplicity and flexibility. React was developed by facebook and was introduced as well 10 years ago.

Building on these existing tools, the creation of a chatroom with an Erlang backend offers a good opportunity to get to know the structure of an application with frontend and backend.

3 Presentation of the solution

In order to realize our chatroom, we had to deal with different functionalities and link the specific components with each other. The diagram presented in Figure 1 provides a visual representation of the architecture of our chatroom application. In the backend

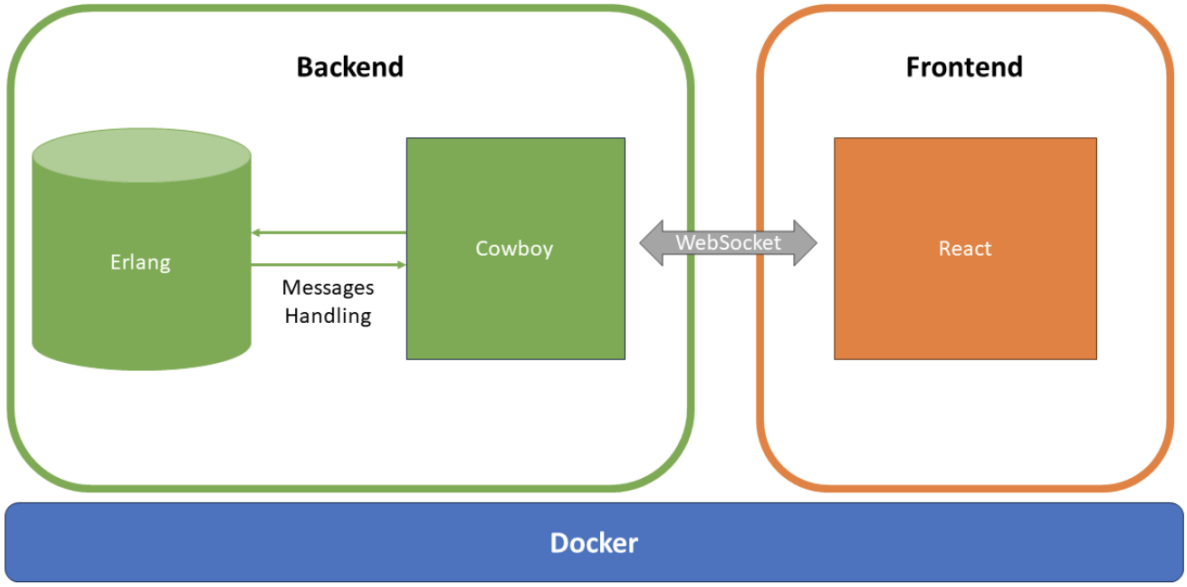


FIGURE 1 – Chatroom Application - Design Diagram

we handle the data in Erlang. The inherent strengths of Erlang, such as lightweight processes and efficient message passing, makes it particularly valuable for developing chatroom applications. An HTTP server developed for Erlang - Cowboy - enables us to implement the WebSocket protocol for the interface. The front end of our application was constructed using React, a JavaScript library developed by Facebook. To facilitate the execution of our application amongst different users we further rely on Docker. A more detailed description of these components and their concrete implementation up to the complete chatroom is described below.

3.1 Backend

Rebar3

At the beginning of our implementation we set up our backend project structure using Rebar3 [**1**]. Rebar3 is a dependency management tool and it took over two main functions during our project work. Primarily, it stands out as a helpful tool for managing dependencies, simplifying the handling of project requirements. Secondly, Rebar3 automates the compilation of code, which simplifies the workflow. To enable this simplification, the correct configurations must be stored in the *rebar.config* file.

```
{deps, [
  {cowboy, "2.9.0"},
  {jsx, "3.1.0"},
  {gproc, "0.8.0"}
]}.
```

In our case Rebar3 helped us to easily integrate Cowboy in our Erlang project. Furthermore we use the library `jsx` to handle JSON data, the library `gproc` to progressively handle processes and `ETS` tables to build a key-value store for keeping track of our chatroom users.

ETS table management

The backend of our application is based on Erlang. At its core lies the effective coordination of concurrent user connections through the utilization of an ETS table (Erlang Term Storage). ETS tables in Erlang provide a high-performance, in-memory storage mechanism for handling large volumes of data across concurrent processes. These tables offer efficient read and write operations.

We organize the management of the different user sessions in a separate Erlang module : *session_handler.erl*. This module provides definitions for functions to create, update, delete and get the values of the different user sessions using key-value store defined in ETS tables.

Cowboy and WebSocket

Cowboy is an open-source HTTP server designed for Erlang [**2**]. Cowboy is ideal for web services and stands out for its efficient handling of concurrent connections and lightweight design. It serves as a robust solution for managing HTTP requests and WebSocket connections.

In the case of real-time chatroom applications, continuous communication is essential. Based on this requirement we decided to work with the WebSocket protocol to let the frontend and backend communicate with each others [**3**]. WebSocket technology serves as a communication protocol, facilitating real-time, bidirectional data exchange between a client and a server through a persistent connection. As displayed on Figure 2, a client

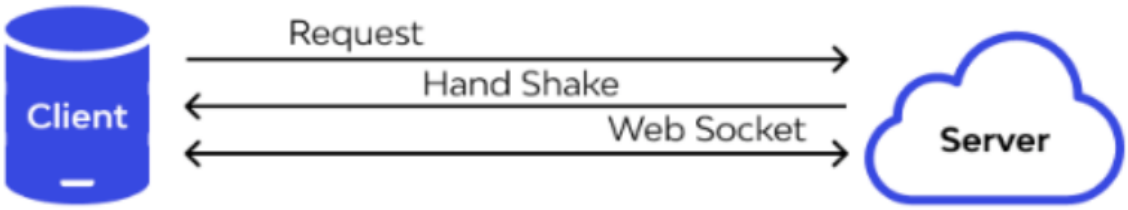


FIGURE 2 – WebSocket connection

establishes at first a HTTP connection with the server to initiate the handshake. Once the handshake is complete, the communication upgrades to the WebSocket protocol, providing a persistent channel for real-time data exchange. This is a major difference to REST, which follows a stateless request-response pattern, suitable for discrete operations like data retrieval or modification, using standard HTTP methods. In contrast the WebSocket protocol allows for instant communication between the frontend and backend, making it ideal for dynamic and interactive applications. By establishing a continuous connection, WebSocket eliminates the need for repeated requests, reducing overhead and

ensuring quick data transmission. This approach hence enhances the responsiveness of a chatroom, enabling users to receive and send messages without noticeable delays.

In order to work with the Websocket protocol, Cowboy provides various functions that can be used to establish the connection and process messages. At the beginning of each connection, a request is sent to the Cowboy `init/2` callback, which forwards the request to the `cowboy_websocket` module. The `websocket_` callbacks associated with this module take care of the correct handling of the websocket protocol once the connections have been successfully established. Each connection corresponds to an Erlang process to which the associated PID is assigned via the `websocket_init()` function. The subsequent processing of the messages can then be carried out using the provided `websocket_handle()` and `websocket_info()`. `websocket_handle()` is called when a message arrives from the client, while `websocket_info()` is used for processing Erlang messages. At the end of a connection `websocket_terminate()` function is used.

By supporting the WebSocket protocol, Cowboy plays an important role in facilitating the server-client communication in our chatroom application.

Multicasting and gproc

To distribute the messages to the connected users the process PID of each connection provides a useful available "address". However the implementation of the base case of the e.g. `websocket_handle` function provides not automatically for the mandatory addressing of all recipients.

```
1 websocket_handle({text, Message}, State) ->
2 %% parsing and de-/encoding of messages/response
3 {[{text, Response}], State}.
```

In the case of a chatroom the simultaneous distribution of messages is yet absolutely necessary. This is also referred to as multicasting. Multicasting describes the simultaneous distribution of a single message from a sender to multiple recipients. In the context of our chatroom application, multicasting is crucial for efficiently broadcasting messages to all participants in real-time. This enables a responsive chat experience as users can instantly receive updates and engage in group conversations without the need for individualized message distribution.

On the one hand, comprehensive addressing of messages would be possible based on our ETS table. However, the already developed gproc process registry is a very useful tool here [**4**]. gproc allows for efficient management of processes among connected users by associating them with unique global keys or names. This way multicasting can be achieved with just two lines of code. In order to register processes under a global name one can use the `reg()` function, when establishing the websocket connection.

```
1 gproc:reg({p, l, my_chatroom})
```

All the registered websocket processes can then be retrieved in one catch by calling the gproc `send()` function, which creates a package of all "adresses"and the response message :

```
1 websocket_handle({text, Message}, State) ->
```

```
%% parsing and de-/encoding of messages/response
gproc:send({p, l, my_chatroom}, Response)
{ok, State};
```

Implementation and assembly of the individual components

In our implementation the parts described above were needed to be assembled together. The use of the different components is particularly evident in the functions for managing the exchange of messages between the front- and backend. Following the arrival of a new message from the client, a case distinction is first made. This is either the creation of a new user or the transmission of a message for the chatroom. If a new user has joined the chatroom, the message is first decoded from the JSON format. In the following code, the ETS list is used to register the data of the participants. After parsing the message back into JSON format, the message is addressed and transmitted using gproc. The distribution of the chat messages also takes place accordingly.

```
websocket_handle({text, Message}, State) ->
    ParsedBody = jsx:decode(Message),
    case ParsedBody of
    #{<<"type">> := <<"user-created">>, <<"username">> := Username} ->
        session_handler:add_participant(Username), % Add participant to ETS list
        Participants = session_handler:get_participants(),
        Response = jsx:encode(#{<<"type">> => <<"user-joined">>,
            <<"username">> => Username,
            <<"message">> => <<"joined the chatroom">>,
            <<"participants">> => Participants}),
        io:format("User created: ~s~n", [Username]),
        io:format("Response: ~s~n", [Response]),
        io:format("Values in my_chatroom: ~p~n", [gproc:lookup_values({p, l,
            my_chatroom})]),
            gproc:send({p, l, my_chatroom}, Response),
            {ok, State};
    #{<<"type">> := <<"send-message">>, <<"username">> := Username, <<"message">> :=
        MessageText} ->
        io:format("Received message from ~s: ~s~n", [Username, MessageText]),
        Response = jsx:encode(#{<<"type">> => <<"chat-message">>,
            <<"username">> => Username,
            <<"message">> => MessageText}),
        io:format("Response: ~s~n", [Response]),
        io:format("Values in my_chatroom: ~p~n", [gproc:lookup_values({p, l,
            my_chatroom})]),
            gproc:send({p, l, my_chatroom}, Response),
            {ok, State}
```

This code section of the `websocket_handle()` shows how the different components must be linked in order to enable efficient communication, process the data efficiently and make it available to the participants in a user-friendly way.

3.2 Frontend in React

For the frontend aspect of the chatroom application, we chose React due to its comprehensive capabilities for building user interfaces and its support for real-time updates [**5**]. Our implementation primarily revolves around three key files : *App.js*, *CreateSession.js*,

and *WebSocketManager.js*.

App.js serves as the root component, orchestrating the overall layout and navigation logic. *CreateSession.js* manages the initiation of user sessions and the handling all the messages, while *WebSocketManager.js* is responsible for managing WebSocket connections to enable real-time message exchange.

The *CreateSession.js* script is central to our application, handling both the transmission and reception of messages. It facilitates user interactions such as starting chat sessions and exchanging messages. Within this script, `handleCreateUser` and `handleSendMessage` methods manage the dispatch of session initiation requests and chat-messages to the backend. The `handleMessage` method then processes the backend responses by parsing the JSON messages for accurate display in the user interface.

```
1  const handleMessage = (event) => {
2     try {
3       const data = JSON.parse(event.data); # Parsing data from the received JSON
            message
4       if (data.type === 'chat-message') { # Logic for 'chat-message' messages
5         setMessages(prevMessages => [...prevMessages, { username: data.username,
              message: data.message }]);
6       } else if (data.type === 'user-joined') { # Logic for 'user-joined' messages
7         setParticipants(data.participants); # Update the participant lists when a new
              user joins the chat
8         setMessages(prevMessages => [...prevMessages, { username: data.username,
              message: data.message }]);
9       }
10    [...]
11    };
```

This approach ensures dynamic updates of the chatroom's content and enables interactive participation of users.

3.3 Docker

Docker was a valuable addition to our project, simplifying the deployment of our application's components. It allowed us to run the Erlang backend and React frontend as distinct microservices in separate containers, ensuring all dependencies were included. This method improved the consistency and reliability of our application across different environments.

Our Docker setup involved two distinct *Dockerfiles* for the backend and frontend, alongside a *docker-compose.yml* file. The *docker-compose.yml* was configured to initiate two frontend instances. However, its flexible design allows for the easy addition of more frontends and therefore more users.

A key advantage of Docker was its ability to resolve compatibility issues, or 'works on my machine' problems. By using Docker, we could easily build and run our application in various environments with a single command, `docker-compose up --build`, ensuring a smooth and efficient deployment process. This process highlighted how Docker simplifies developing and deploying projects with multiple technologies.

4 Evaluation of the solution

The solution developed effectively achieves its intended functionality. When Docker is executed, both the backend and frontend microservices activate without issues. Users can access the application by visiting localhost on ports 3000 and 3001, where they have the ability to create user accounts and engage in the chat. The application efficiently facilitates message exchanges between different instances, and displays a list of participants in a dedicated section on the interface. Despite its simplicity, the application offers a straightforward and practical user experience.

The integration of various technologies, including Erlang, Cowboy, and React, proved highly effective for our project. Erlang's compatibility with Cowboy was particularly advantageous for backend development, while React emerged as a robust choice for the frontend. Additionally, Docker served as the perfect tool to unify these components, offering a practical solution for consistent deployment across different environments.

While the system effectively meets basic requirements, there are areas that could be improved. Presently, the application doesn't efficiently manage user disconnections and reconnections, resulting in departed users remaining visible on the participant list until the application is restarted. Also, there is no message history preservation ; messages are transmitted in real-time without any storage. These issues present opportunities for future enhancements. Potential improvements include implementing an ETS table for storing message history and improving the mechanism for updating user statuses. These enhancements could significantly expand the application's capabilities and improve the overall user experience.

Key lessons learned include :

- Leveraging existing solutions and building upon them, rather than attempting to reinvent.
- The importance of designing a simple, clear workflow for the application from the outset.
- Starting with basic functionalities and progressively expanding the application's capabilities.

Finally, a significant learning outcome was navigating the steep learning curve associated with Erlang's unique syntax and functional paradigm, which posed challenges but also offered valuable insights.

5 Conclusion

The development of our chatroom application served as an insightful exploration into the integration of concurrent and distributed systems within a full-stack environment. The project successfully combined the strengths of Erlang and Cowboy for the backend with React for the frontend, all within a Dockerized framework. This blend of technologies demonstrated not only their individual capabilities but also their potential when combined effectively. The project underlined the significance of utilizing existing technological

solutions, emphasizing the advantages of a systematic development approach and the necessity of incremental improvements.

In the course of the project, we identified areas for further enhancement, such as more efficient handling of user disconnections and the storage of message history. These areas present opportunities for future development, aiming to refine the application and enhance the user experience. Additionally, navigating the unique syntax and functional paradigm of Erlang presented a notable challenge, but it was one that ultimately contributed to a deeper understanding of distributed systems and enriched our programming skills.

In summary, this project was not just a technical endeavor but also a profound learning experience. It highlighted the importance of blending theory with practical application, and underscored the critical aspects of adaptability and scalability in software development. The experience and insights gained from this project lay a solid foundation for future explorations in the field of concurrent and distributed systems.

6 Bibliography

[1] `https://rebar3.org` version 2023 Last visited : 19.12.2023.

[2] `https://ninenines.eu` version 2012-2018 Last visited : 19.12.2023.

[3] `https://datatracker.ietf.org/doc/html/rfc6455` version 2011 Last visited : 19.12.2023.

[4] `https://github.com/uwiger/gproc` version 2022 Last visited : 19.12.2023.

[5] `https://react.dev` version 2023 Last visited : 19.12.2023.

7 Appendix

How to deploy the code and use the Chatroom application.

1. Ensure Docker is installed and running on your system.
2. Open a terminal and navigate to the root directory of the project.
3. Run the command `docker-compose up --build` to build and start the backend and frontend containers.
4. Open your favorite web browser.
5. Access `http://localhost:3000/` and `http://localhost:3001/` in two separate tabs or windows.
6. Interact with the Chatroom Application via the frontend. Choose different usernames in each session to simulate multiple users.
7. Start chatting and observe the interaction between the two chat sessions.