17th International Conference on Space Operations, Dubai, United Arab Emirates – 6-10 May 2023.

SpaceOps-2023, ID # 205

Role-based Multi-Chat System for Space Mission Control

Falk R. Schiffner[a]*, Maximilian Burr[b]

[a] *German Aerospace Center (DLR e.V.), Space Operations & Astronaut Training, Berlin, Germany*, falk.schiffner@dlr.de

[b] *Technische Universität Berlin, Distributed Security Infrastructures, Berlin, Germany,* maximilian.burr@campus.tu-berlin.de
* Corresponding Author

Abstract

Flawless voice communication is crucial for space mission control. Nevertheless, communication via a voice communication system (VoCS) has a major drawback; speech is fleeting and gone by the second. This can be addressed by replaying the recorded speech, but finding the correct timestamp of an utterance can be time-consuming. Therefore, an additional communication channel is envisioned for future space mission communication systems. This paper reports the conceptualization of the role-based multi-chat system for space mission control.

The paper is subdivided into two parts. The first part reports the concept. We will clarify the distinction between our approach to existing systems used outside the space flight domain. We will show that the role-based approach is also advantageous for a chat application. The multi-chat concept allows participants to take part in several chat loops simultaneously. This is especially interesting since space missions are more and more endeavors of several agencies with industry and private partners.

The developed concept foresees that the chat system can operate by itself or supplement a VoCS as an additional feature. This is meant to support the flexibility of agencies and enterprises to implement it into their workflow.

Among other things, we will detail the role-based access to several dedicated chat loops. Here, we explain what a role is and what that means for the chat loop setup. Further, we will explain the three states of a chat loop. These are namely "Read," "Write+Read," and "NotAllowed."

This paper continues with the second part, where we explain the development of the first prototype. Here, we will go into more detail on how the user can interact with the system. For example, we neglect the idea of notifications or other interruptions like audible signals to avoid user distraction. Moreover, we will explain in detail how the user interface is designed. For example, we chose a reduced interface design with a color scheme to enable the user to check each chat loop's state quickly. Finally, the paper will close with an outlook on future topics regarding integrating the prototype into existing VoCS and the potential beneficial combination with Speech-To-Text and Text-To-Speech engines that fully fuse the two communications channels.

We firmly believe that a role-based multi-chat system would benefit companies and space agencies alike in a more and more interconnected world. Even today's operators are familiar with different chat systems and expect chat functionality from a modern communications system.

Keywords: Chat System, Role-based Chat Communication, Voice Communication System, Space Mission Operation

Acronyms/Abbreviations

VoCS	=	Voice Communication System
GSOC	=	German Space Operation Center
ESA	=	European Space Agency
POC	=	Proof of Concept
HTTPS	=	Hypertext Transfer Protocol Secure
JSON	=	JavaScript Object Notation
UUID	=	Universally Unique Identifier
CSV	=	Comma-separated values
CSS	=	Cascading Style Sheets

DLR = Deutsches Zentrum für Luft- und Raumfahrt e.V. (German Aerospace Center)
UDP = User Datagram Protocol

1. Introduction

In space operations, coordination through communication facilitates successful missions. A direct, structured method of communication is voice communication through voice loop systems [1]. Through voice loop systems operators can stay aware of ongoing activities and mission-related events without the necessity of being fully involved in the conversation. Voice loop systems have proven to be an efficient communication tool for space mission control [2]. Thus, voice communication is an essential communication path for operators.

However, voice communication has one major drawback, which is its volatility. Therefore, if an operator misses essential information, it is gone and must be repeated. Voice loop systems can offer to record and replay on request [1]. Nevertheless, searching through a database of recordings for the correct utterance can be time-consuming.

This paper introduces a novel concept for communication in space mission control. An additional communication path is proposed, thus making information exchange readable and persistent. The chat system offers various features, such as multiple chats and selective reload of stored messages. They enable an operator to participate in multiple conversations and decide which chats are loaded for re-reading.

For rights management, the approach of role-based access is considered analogous to a typical voice loop system. In addition, a proof-of-concept implementation allows the evaluation of the concept in theoretical and practical terms.

2. Concept

As outlined before, communication in space mission control is dominated by voice communication. However, augmenting the current voice communication with chat-based communication might be beneficial.

Today, chat systems offer a myriad of features for their users. But only some features are relevant for space mission control. Some features might even be counterproductive because they overload an operator. In space mission control, this can be a significant safety risk. The requirements derived from the mission control setting define vital points of the chat system.

This section introduces the concept and will begin with the architecture of the chat system. Further, the section continues with the conceptualization of the user interface. Finally, the section closes with a section about the other domain-specific conceptual decisions.

2.1 Architecture

For a chat system, the goal is to deliver messages from the sender to a receiver or, in the case of group messaging, to many receivers. The task of the architecture is to facilitate message exchange through efficient routing and fast delivery. If the delivery is not fast enough or messages get lost, the user experience suffers, or worse, delayed or missing information can lead to bad decision-making. Thus, the architecture must ensure that messages are routed efficiently and without loss.

In Fig. 1, the concept of the architecture for the chat system is depicted. Conceptually, the architecture is based on a central server where clients connect. The central server, depicted as a large blue circle, is responsible for authenticating clients, maintaining client connections, routing, and storing messages. Each client, presented as a small black circle, maintains a bi-directional connection with the central server. The clients send and receive messages through the established bi-directional client-server connection.

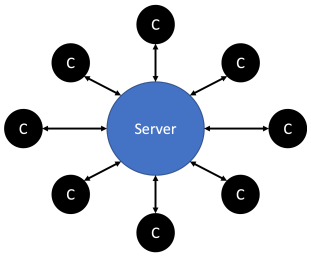

Fig. 1. Conceptual architecture for the chat system with a central server and multiple connected clients (black).

In a voice loop system, speech facilitates information exchange and can be regarded as a synchronous communication path. On the other hand, the conceptualized chat system introduces an asynchronous communication path to space flight operations. In general, an operator can send and receive messages at all times. However, a receiving operator can read and answer at a later point in time.

The chat system features a selective message reload feature to enable the operator to re-read older messages if needed. This is explained in detail in the following section. Messages may be reloadable up to a certain point in time. The storage period depends on the configuration made in the system setup.

The asynchronous message exchange has a major advantage compared to voice communication. For example, if an operator is away from his desk, another operator who wants to get in contact can leave a message in the desired chat loop. Eventually, when the operator returns to his desk, he will notice the message and can respond. In a voice-only system, it is not possible to leave a message. Thus, the contacting operator has to repeat his message over and over again until the receiving operator is back at his desk to answer the request.

2.2 Server Services

The server is the central element in the system. As described beforehand, all clients connect to a central server to send and receive messages. Additionally, the server provides services that further enhance the user experience.

In space flight operations, storage is an essential component. For example, in voice loop systems, voice communication is stored and available for post-mission analysis and training purposes. Therefore, the chat system must provide the same functionality to permanently store all messages transferred through the chat system in a database.

For the chat system, the messages should be stored on a per-loop basis. This means that for every chat loop, the messages should be stored separately so that they can be easily distinguished. For example, in post-mission evaluation, a team communicating on a specific chat loop can retrieve their data from the database and evaluate the communication.

Initially, the storage was only envisioned for post-mission analysis and training purposes. While it could be used to provide old messages to the user during operation, it might overstrain users if too many messages are loaded in the setup phase. Users might be overloaded with messages that have been sent during their absence. Thus, if they join a new loop, an operator might be forced to read many messages before continuing his work. Therefore, not loading old messages mitigates the overload and allows the operators to focus on a new session without being bothered by old messages.

However, after an expert workshop with trainers and operators from GSOC, the envisioned storage had to be extended to provide old data during the active session. Users of a chat system expect that they can read older messages for each chat loop. For example, if an operator did not monitor a chat loop for the last few minutes but messages arrived, he can retrieve them. Another example could be that the session before discussed important information which is now necessary. Thus, by retrieving the old messages from storage, the user can re-read the messages.

Therefore, the chat system incorporates a mechanism to retrieve old messages which are currently not displayed. However, the system should not load old messages by itself but rather on demand. This prohibits an overload after joining a loop. Additionally, the system should only provide a limited number of messages, which allows for keeping the number of messages low. Furthermore, this limit minimizes the risk of using the chat system to drop notes for later use.

The system only delivers messages from the last 24 hours, messages older are not delivered. The final time frame can be adjusted on a case-by-case approach. Therefore, an evaluation is necessary to balance the trade-off between loading messages and not overloading an operator.

Additionally, it is stored when a user has joined and monitored a loop during a certain time frame. Therefore, the chat system should include a log containing all entries and all exits to a chat loop by users. The log could be used in the chat system as well. For example, every time a user joins or leaves a chat, a log message is sent to all other users and displayed directly in the chat. This could be useful to know if a specific user is still present.

Nevertheless, the downside of a log message in the chat is that it is an additional message. Operators in space flight operations already have lots of tasks to perform at the same time. Thus, it is important to reduce the information load to a minimum. Therefore, the server should only store, not relay, log information to other participants.

The log for the chat system should be stored separately from other data, such as messages or user accounts. Furthermore, the log should be easily retrievable by the system administrators. Each log entry should contain information to identify the user, the loop, and the time at which the action happened.

However, the user does not notice the architecture when connecting to the server. Therefore the next section introduces the concept of the user interface for the chat system.

2.3 User Interface

The user interface defines how the user can interact with the chat system. One of the most important elements of the chat system is the input field. The input field allows a user to send messages in a chat loop. Many messengers, such as Signal or Element, which is a Matrix client implementation, place their input field directly underneath the ongoing chat [3, 4, 5]. In fact, the widespread usage of chat systems [6, 7] leads to a specific user expectation, and therefore, it makes sense to place the input field underneath as well. Therefore, their familiarization phase is expected to be short.

The usual send process is that the user types a message into the input field and then initiates the transmission through a predefined key, such as the "Enter"-key. This could lead to sending accidentally or incomplete messages.

In the space flight domain, sending messages accidentally on a key press is an unintended action. Therefore, an operator should only be able to send messages through intentional action.

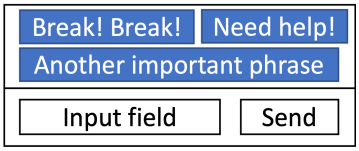

Fig. 2. Conceptual input field with explicit send button.

Furthermore, every operator's action on the chat system should be intentional. As shown in Fig. 2, a dedicated send button is another method to initiate the transmission. If the button has to be pressed, it cannot be initiated unintentionally while typing a message into the input field.

While it is still possible that the mouse is unintentionally placed on the button and pressed, it is much more unlikely compared to a key press. Thus, cluttering the chat with unintended messages or additional messages is greatly reduced by using a button.

The input field and the send button are related in use; thus, keeping these elements close together makes sense. As depicted in Fig. 2, the button could be placed to the right of the input field to allow fast and easy access after typing a message.

Essential to a chat system is to display the sent and received messages. Each message, from loops the user is engaged with, should immediately appear on the user's screen. The chat system should display all messages as soon as they are written or received. For easy readability, the chat system must keep a chronological order, where messages are sorted after creation time. Other chat systems, such as Signal, offer a separate space for each chat that the user is engaged in [3]. The advantage of having a separate space is that all the messages are separated from each other. Only messages related to the specific chat are displayed while the user is inside each specific chat.

While separating the different chats is desirable because it reduces the number of messages per chat space, operators must be able to read multiple chats simultaneously. But, if the user wants to interact with a high number of chats at the same time, it is impractical to separate the chat spaces.

For a large number of chats, the different open chat spaces will only fit on the screen, decreasing the size of each chat space or overlapping them. Even then, it overloads the screen with different spaces, which can overwhelm a user. Thus, an accumulated representation in one window is our proposed solution to this problem. In this case, all messages from all the chats the user is engaged in will be represented in a single chat space.

In an expert conversation with members of the GSOC and ESA, it became clear that an operator in space flight operations does not have the time to interact with multiple different chat spaces and needs a fast overview of all the messages. Therefore, the concept foresees a single chat space that displays all incoming messages.

Sometimes, it might be inconvenient to use a chat system, and thus, a chat space with incoming messages might annoy a user. However, closing the chat altogether is not an option because the chat system can become relevant at any second. Therefore, the chat space in the chat system should be retractable. A retracted chat space allows free space for other components of the chat system.

Fig. 3 shows a conceptual visualization of the chat space and the input field. For the chat system, it is envisioned to use a similar placement for the chat space as in other chat systems. For example, in Signal and Element, the chat

space is placed directly above the input field and the send button [3, 4]. Maintaining the same layout as in other chat systems is geared to reduce the adaptation time of operators.

The messages in the chat space, as seen in Fig. 3, are separated depending on who created them. Messages created by the operator should be placed on the right side, and received messages should be placed on the left side of the chat space. This allows for easy differentiation between sent and received messages

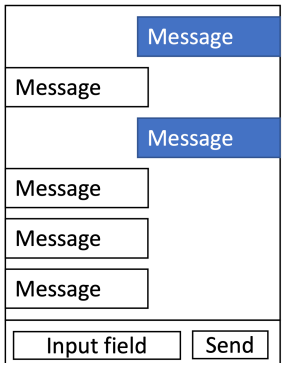

Fig. 3. Conceptual chat window with several messages and an input field at the bottom.

Further, the usage of color enhances this separation. This separation offers a clear overall view of incoming and outgoing messages.To distinguish the messages from different chat loops received in one chat window, it is necessary to add visual tags to messages to discern them. The message and its tag should be visually connected so that it is apparent which tag belongs to which message. The amount of information in the tag can differ between sent and received messages.

For sent messages, it could be sufficient to add the chat they were sent in as part of the tag, together with sending time. Received messages need tags with more information. The time and the chat loop are elementary information and must be included. However, the received messages additionally must include the sender of a message. Otherwise, it would be impossible to discern the received messages.

As depicted in Fig. 3, the chat space has a fixed size. Thus, scrolling behavior is necessary to display longer conversations with multiple messages. Scrolling enables the user to reach old messages. Re-reading old messages is one of the big advantages of a chat system compared to a voice communication system.

The scroll bar should be non-obtrusive to focus clearly on the message exchange. During scrolling, the user should be able to stop at any given point.

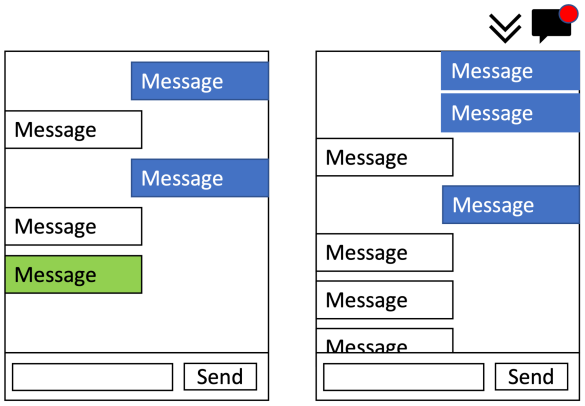

Fig. 4. Appending new messages (green) without and with scrolling behavior.

As shown in the chat window on the left in Fig. 4, the new messages, marked green, should be placed at the bottom of the chat space. Accordingly, old messages move to the top of the chat space. Therefore, the chat system must automatically scroll to the bottom to fully display the message.

However, if a user has scrolled to a different position before or during the arrival of a new message, the chat window should not move, and the message should still be appended at the bottom. As depicted on the right in Fig. 4, the user should still be notified of the new message through a visual cue. Additionally, as in Fig. 4, a button can help jump to the bottom of the chat space immediately. Thus, enabling a user to read and react to the newest message if desired.

Extensive usage of the chat can crowd the chat space with messages. An overcrowded chat space can become a problem for users in space flight operations. Especially during pressure phases, it is beneficial if the chat space does not fill up with messages because focus might be needed for other tasks. Therefore, the chat system should have mechanisms to keep the number of messages loaded at a minimum. For example, clearing the chat space from messages during operations could be beneficial. This could be achieved by deactivating a chat loop through a dedicated deactivation button. Through deactivation, the user leaves the chat loop, and no new messages are received on that loop. Further, this removes all content from that chat loop from the chat space.

Each chat loop represents a chat with multiple participants in the chat system. Visually each loop should look similar to a voice loop system. As described by Peinado et al., the current voice loop system uses a layout where a tile-like interface represents the loops [8]. Thus, the chat system adopts this layout for its loops, as shown in Fig. 5.

The tile-like space should contain the name of the loop for identification. The middle of the tiles is used for a small description of the loop.

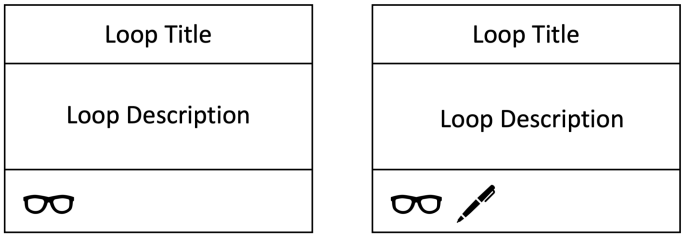

Fig. 5. Visual representations of loops with different rights (left side - read access, right side - read & write access).

The description field might be useful if new loops appear for an operator to know what they are for. However, other use cases for the description field can be imagined. For example, it could be used to show all operators who are assigned to the loop or have the writing state activated. To specify an advantageous use of this tile space, an investigation should be conducted with expert operators in the future.

In a voice loop system, each voice loop has a specific state. The specified states are "Off," "Monitoring," or "Speaking." The chat system should have similar states for its loops to reflect the hierarchical command structure and mimic the advantageous fast workflow of the voice loop setup. Therefore, the chat system has four distinct states for each loop.

- Not Allowed - Not every loop is accessible for every user. Thus, the chat system must ensure that users cannot interact with loops they are not assigned to. If a user is Not Allowed in a chat loop, this loop is not shown to the user.
- Off - This is the initial loop state for loops a user is assigned to in the chat system on startup. Here, the loop is unused, and messages are not displayed. The "Off" state should be reachable through a dedicated button, allowing the deactivation of a loop immediately.
- Monitoring - A loop in the "Monitoring" state is read-only. Thus, all messages that are written are displayed in the chat space. This may be the only available state for some loops if a user does not have write permission. Thus, checking if a user is allowed to write is necessary. Otherwise, written communication should be prohibited on that chat loop. If write access is given on a loop, a user can trigger another state change from "Monitoring" to "Writing" by clicking once more on the desired loop.
- Writing - Only one loop can be in the "Writing" state. If writing to several loops were allowed, it could lead to unintended messages flooding the system. If another loop is switched to "Writing" state, the current loop is changed to the "Monitoring" state.

The chat system should include a visual representation of each state. This is done by changing the colors of the loops on the screen to show the user that a state change has occurred. Using different colors to show the current state

of the loop is also used in the voice loop systems [2]. Here, the state switch is represented by green for TALK and blue for MONITOR. Moreover, the rights are shown by using small icons (headphones and microphone). This idea was transferred to the chat system using glasses for reading access and a pen for writing access.

After the description of the user interface, which defines how the user can interact with the chat system, the following section describes the security features of the concept.

2.4 Security Considerations

Since the chat system will eventually be deployed in a highly secured network, as is common in space operations, one might think the chat system does not need any additional security features. For example, a malicious entity could gain access to the network through a weakness and read all messages. Additionally, security might be required in other use cases of the chat system. For example, if the chat system is used in a non-secured network and the users still want secure communication, it is beneficial if security is built in. Nevertheless, security concepts such as encryption exceed the scope of this paper.

An essential security aspect is the management of users and their respective rights. Therefore, using role-based rights management is beneficial in a space mission environment.

The chat system uses role-based rights management, where the user gains the rights of the role through the assignment by an administrator. However, the rights themselves are assigned to roles. For example, each role contains a certain set of loops that can be accessed. If a new user is added to the database, not all the necessary rights for the loops must be assigned. Instead, only the respective role must be assigned to the user to gain the rights.

After discussions with experts from GSOC, the peculiarity of using positions as an additional abstraction layer was detailed. As shown in Fig. 6, each position has a single role. The users themselves are assigned the positions, which enables administrators and other personnel to differentiate between users even if they are assigned the same role.

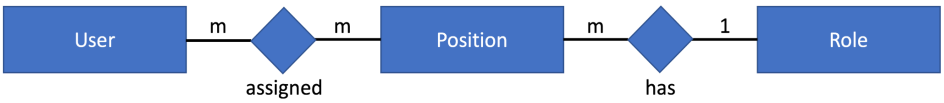

Fig. 6. Entity-relationship model for user, position, role.

By supporting positions and roles, the system can be used in environments with only roles, with minor changes in the structure.

Various security checks should enforce the rights throughout the chat system. Therefore, even if one security check is bypassed through intended or unintended behavior, other checks can still uphold the integrity of the chat system. Therefore, enforcing the access rights set by the system administrators. For rights management, it is important to have authentication of the users. A respective mechanism is necessary to identify the user.

3. Implementation

The following section describes the implementation of the chat system, which is tailored to usage in space mission control. The implementation should be regarded as a proof of concept (POC) and is not ready for production use.

All the code files and instructions on executing the implementation can be found in the repository[1].

3.1 Architecture

As described in the concept, the architecture is based on a central server, which runs the application. Several clients can connect to the central server to interact with the chat system. The chat system is developed as a browser-based web application, so a user has access through a web browser. Any major web browser can be used for the interaction, and this opens a wider user range.

[1] https://github.com/max-i-am/mmocs

The central element of the implemented chat system is the server. The server is implemented in Python 3.10 [9]. For Python, different web application frameworks are available, which allow building web applications. The Flask [10] web application framework is used for this project.

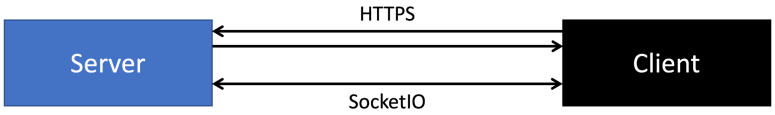

Fig. 7. Connection between the server and a client as implemented in the chat system.

As depicted in Fig. 7, each client uses an HTTPS connection to access the chat system on the server. The client sends an HTTPS request, and the server answers with an HTTPS response. Through HTTPS communication, the client can retrieve the web application and load the application in the web browser. All functions running on the clients of the web application are implemented in JavaScript [11].

As seen in Fig. 7, the HTTPS connection is a communication path that can only be initiated by the client. However, a communication channel is necessary for exchanging messages in the chat system where both parties can send and receive messages. So, the server must be able to send data to the client without a preceding request.

A library, which offers support for low-latency, bi-directional communication is SocketIO [12]. SocketIO uses the WebSocket protocol at its core. Using the SocketIO library creates a WebSocket connection between the server and the client. But, SocketIO is not compatible with plain WebSockets because SocketIO sends additional metadata with each packet.

For the chat system, it would be possible to use plain WebSocket connections to create the bi-directional communication channel. However, SocketIO has a few additional features, such as a fallback to HTTPS long-polling, automatic re-connection, and broadcasting [12].

Through automatic re-connection, SocketIO makes sure that the connection is reconstructed after the link has been broken [12]. A heartbeat mechanism checks the status of the connection. If the client disconnects because of a broken link, it automatically re-connects the client to the server.

Broadcasting with SocketIO allows sending data to a set of clients [12]. This is useful for the chat system because the chat loops are group chats. Therefore, the messages that are sent to a loop must be disseminated to all connected clients currently in the loop.

A drawback of SocketIO is that it introduces an overhead to each message [12]. This can be neglected because it is only a minor overhead compared to plain WebSockets. The second drawback is that the SocketIO client library must be transmitted to a client's browser before it can be executed. In comparison, a plain WebSocket connection is built into modern browsers. For this implementation, the advantages of using WebSockets through SocketIO outweigh the drawbacks.

3.2 Chat Communication

Before messages can be exchanged, the users participating must join the loop. The client can join a loop by sending a join request. If the user is allowed to join the loop, the user is registered in the SocketIO broadcast mechanism. Otherwise, the user is rejected and will not be connected to the loop and, thus, will not receive messages from that chat loop.

After successful registration, the chat messages are exchanged through the broadcast mechanism. Only the subset of people currently registered for the loop receive the message. For the correct delivery of messages to the selected loop, additional metadata is necessary. The sender sends the following metadata with every message:

- Loop - A string defining the loop the message should be sent on
- Message id - An id, which is unique for every message of a client
- Message - Actual text message

The message and its metadata are sent to the server in a JSON string, which can be retrieved on the server [12]. After retrieving the message, the server checks whether the client has specified a loop and whether the rights are

sufficient to write on that loop. Only if the user has the right the message is broadcasted to the currently registered users for the specified loop.

For an outgoing message, additional metadata is specified, which details the sender's position. Thereby enabling the receivers of the message to know who sent the message. As explained in the concept, the position is an additional abstraction layer to the role. Including the position, the following metadata is sent to the receiving clients

- Position - The hierarchical position of the sender
- Loop - A string defining the loop the message should was sent on
- Message id - An id, which is unique for every message on the server
- Message - Actual text message

Besides the position, the sent metadata also differs in the message id. While each message still has a unique message id for identification, the message id created by the sender is unique per sender. However, this is not sufficient for all messages on a server.

Thus, a new message id is calculated on the server for each received message. For a unique identifier of the messages, UUID Version 4 is used. The UUID has been specified in RFC 4122 [13] to create a 128-bit long unique identification.

The newly calculated UUID is sent to the receiving clients of the message. Additionally, the sender gets an acknowledgment message with the newly calculated UUID to replace the message id calculated before with the unique UUID created by the server. By replacing the message id on the sender side, all clients have the same unique message id for a single message.

3.3 Data Storage

Data storage is essential for the chat application. For example, it is necessary for the authentication of user accounts, rights management, and storing chat history.

The implementation uses an addon called Flask-SQLAlchemy [14] to create and interact with the database. Through that, the database can be administered by writing Python code.

The database used in the implementation is an SQLite database, which offers on-device storage of data without any configuration [15].

In the database, each user has a unique username and password. The passwords are stored as hashes. Therefore, preventing the storage in plain text where anybody with access can read the passwords. The combination of username and password is essential to provide a secure login to the web application.

User accounts in the database have a relation to the positions, and each position has a relation to a single role. Thus, role-based access control in the database is realized by creating relations between a user and their position.

Through modification of the database, loops can be assigned to several roles. The relation between loops and roles stores the access right the role possesses for a specific loop.

Additionally, the database stores the chat history, and after the server broadcasts the messages to the receiving clients, the message is stored in the database. Afterward, the messages stored can be retrieved on a per-loop basis.

The database query is only executed to check if the user is allowed to join a loop. Then, the application loads the rights for a chosen role in a session object. A session object is a content store in Flask, which can store information specific to a user [16]. The content store requires cookies, which must be enabled in the browser. The server can access the session object and retrieve the stored content during the message exchange. With the session object's content, the server can verify that the user is allowed to send a message without requiring a database query.

3.4 User Logging

The chat system implements a mechanism to record whether a user has joined or left a loop. The server writes a log entry when entering or after leaving a loop. If the client does not properly leave a loop, the log is still written because SocketIO recognizes the disconnection. The disconnection triggers the log function for every loop in which the user was last active, writing a log entry for each loop. A log entry in the implementation contains the following data:

- Username - A string identifying the account
- Position - A string with the position which executed the action

- Loop - A string with the loop on which the action has happened
- Time - Current UTC time as a timestamp string
- Action - A string containing the action "Join" or "Leave"

The stored data allows for identifying the action that was performed as well as the loop and by whom. Therefore, enabling an overview of who was active during which times and on which loop. This data can be useful for post-mission analysis or incident investigation. The log of the chat system is not as critical as the other stored data in the chat system and, thus, can be implemented by extending a CSV file. The log can be exported and used for analysis by writing to a file. The data does not have to be extracted from the database.

3.5 Message Reload

The chat system should provide a message reload to enable users to read past messages. In the implementation, past messages are loaded on demand from the database and sent to the client. All data, which is necessary for loading past messages, is sent through the low-latency SocketIO connection.

The message-reload is triggered on the client side by sending a request to the server. The server can selectively load the messages of currently relevant loops. Loops that are "Off" are not loaded to save resources and avoid flooding the chat space with unintended messages.

As described in the concept, only the last 24 hours of messages are delivered, including their metadata. Therefore, the database is only queried for the last 24 hours of messages. All messages older than the past 24 hours remain in the database. Each message is identified with a unique message ID. By comparing message IDs with IDs from messages already in the chat space, the client finds duplicates. The duplicates are then discarded, and only the new messages are displayed.

3.6 User Interface

The front end is where the application becomes visible to the user, and they interact with the system. The web browser displays the front end of the chat system.

Different toolkits are used for a visually appealing front end of the web application. The first toolkit used is called Bootstrap [17]. Bootstrap allows building the front end of an application by using pre-built components. All the icons the chat system uses throughout the project are sourced from Font Awesome [18]. Both toolkits provide JavaScript and CSS files. The project includes these files, and the web server delivers them as part of the HTTPS response.

The chat system consists of three different pages, as shown in the screenshots. All pages are built using the same template. The black bar at the top of each page is called the navigation bar. The navigation bar contains links to other web application pages for fast access.

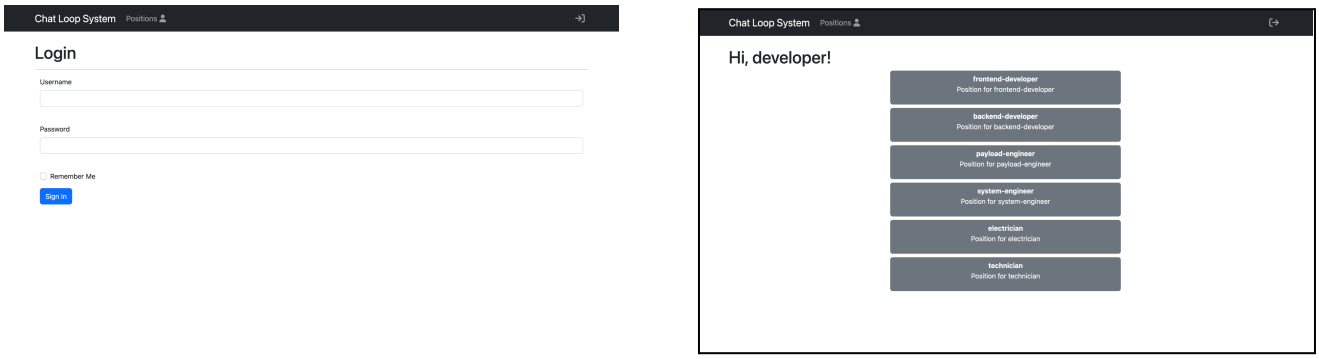

Fig. 8. System screenshots (left side login page, right side - selection menu for roles/positions).

The first page greeting a user when trying to access the web application is the login page, as displayed in Fig. 8. The login page is a minimalist page with two input fields for username and password. It starts the authentication process with the web server. If authentication is successful, the user is redirected to the positions page. The positions

page, as depicted in Fig. 8, lists the positions available to the respective user. The user can select the position for his current activity by clicking on the rectangle containing the position.

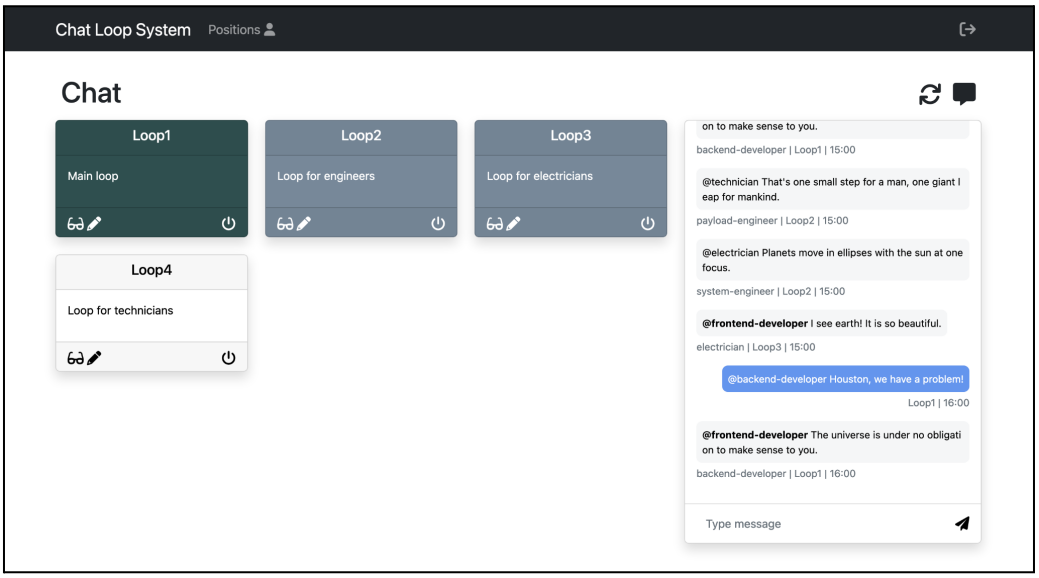

Fig. 9. System Screenshot (chat loop tiles and chat space)

After that, the user is redirected to the chat page, as depicted in Fig. 9. On the left of the chat page; the user is presented with his loops. On the right, the chat window displays the communication on the different loops.

Each loop is implemented as a Bootstrap component called cards. Cards separate content in different containers [19]. Every card representing a loop is clickable to change the loop's state. The whole area of the card is clickable, offering a larger clickable area. This can be beneficial for use with a touchscreen.

A card includes different options, such as a header, a body, a footer, and different background colors [19]. The footer of the loop contains three different icons. Up to two icons are displayed on the left, showing the rights on a particular loop. For example, the "glasses"-icon shows that a user has monitoring rights on a loop and, thus, is always displayed. Additionally, the "pen"-icon can appear on a card showing that a user holds write access on the loop.

The third icon in the footer is a clickable "off"-icon. It deactivates a loop, and the user leaves the conversation (see Fig. 10). Furthermore, turning a loop off removes all its messages from the chat window. Therefore, the chat window only contains the active loops. Again, if the messages are needed, they can be retrieved using the message loading functionality.

On initial access, the chat page only displays the loops; thus, no chat window is visible. A button with a "chat"-icon is located on the top right of the page. By clicking, the chat window becomes visible. The chat window can be retracted by clicking the "chat"-icon again.

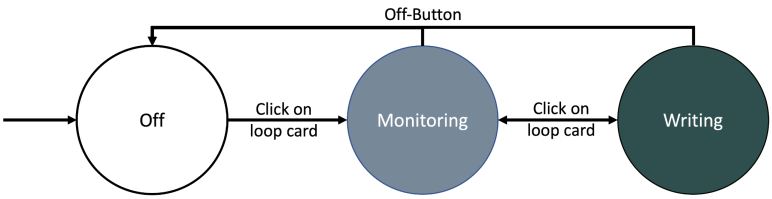

Fig. 10. A state chart showing the loop state changes with the corresponding colors from the chat system.

The chat window is implemented as a large Bootstrap card without a header. In the card body, the chat messages are displayed. The chat system differentiates between sent and received messages. The received messages have a gray background color and are oriented to the left of the card body. The sent messages have a blue background color

and are oriented to the right of the card body. This allows users to easily differentiate between sent and received messages, as described in the concept.

All messages have a small information badge below-containing metadata. The metadata differs for sent and received messages. The order of the metadata is chosen as position, loop, and time to resemble the call structure of voice loop systems at the DLR.

As shown in Fig. 9, some messages contain bolt position names. This is because the chat system features mentions through which specific users can be addressed. This is implemented by checking if an @ was prepended to a message. If the word following the @ is identical to the current position, the @ and the position is displayed in the bolt, so a user knows if he is addressed.

Next to the "chat"-icon, the reload button is located. The reload button triggers the message reload, as explained in the subsection above.

The footer of the chat window contains the text input and the send button. The text input is implemented through a text area to allow for the input of large text messages. The message can be sent by clicking on the "send"-icon placed next to the text area. Since a user should only be able to send a chat message if a loop is in the writing state, the text area and the send button are deactivated if no loop is in the writing state.

4. Future Work

The chat system should be used to conduct further research on how a chat system can be used in space mission control. The envisioned use case described in this work has to be verified through real-world testing in a control room environment.

The testing of the POC implementation was done in a laboratory setting with only two clients and a single server. However, in a real-world test, more sophisticated testing should be conducted. In a mission control center, the chat system can be tested with many more clients to see whether it still performs as reliably. Additionally, the chat system should be deployed in a network environment with different load and capabilities. When testing in a mission control center, the chat system can also be tested with real operators. Therefore, their usage of the chat system can be evaluated. The gathered data can then be used to tune the concept of the chat system.

Additionally, different transport protocols can be tested. For example, WebTransport can be tested as an alternative to WebSockets. Thus, checking whether a chat system with a UDP-based protocol can deliver performance benefits.

As detailed before, the chat system might be useful in other use-cases where a clear hierarchical structure is necessary. For future work, it might be necessary to evaluate the capabilities of the chat system concerning other use cases. For example, disaster relief or military use cases could be considered and evaluated.

The concept of the chat system is only implemented as POC. The language and the packages used for the implementation might not be suitable for a production chat system, and need to be evaluated, e.g., it could be beneficial to use a faster programming language.

The chat system is developed at DLR. A potential combination with the new voice communication system openvocs [20] is envisioned. In a future iteration of a multimodal communication system for space mission control, a fusion of the two modalities, namely the voice channel and the text channel, could be targeted. More precisely, the vision enables the conversion of the written text in a chat loop to be played out on a voice loop via state-of-the-art Text-To-Speech algorithms and vice versa. So, spoken utterances unanswered in a voice loop can be transferred via Speech-To-Text algorithms into text and sent to the respective chat loop. Even though that is a future vision, the technologies are already used in different domains and applications. To tweak these and use them for space mission control seems promising.

References

[1] E. S. Patterson, D. D. Woods, and others, “Voice loops as coordination aids in space shuttle mission control,” *Computer Supported Cooperative Work (CSCW)*, vol. 8, no. 4, pp. 353–371, 1999.

[2] T. Tomlinson and B. Rolet, “A New Era for Voice Conferencing: Advancement in Technology and Capabilities,” in *SpaceOps 2006 Conference*, Rome, Italy, Jun. 2006. doi: 10.2514/6.2006-5610.

[3] “Signal Messenger: Speak Freely,” *Signal Messenger*. https://signal.org/de/index.html (accessed Jul. 18, 2022).

[4] “Secure messaging app | End-to-end encrypted messenger.” https://element.io/personal (accessed Jul. 08, 2022).

[5] “Matrix.org,” *Matrix.org*. https://matrix.org (accessed Dec. 28, 2022).

[6] “Messaging App Revenue and Usage Statistics (2022),” *Business of Apps*, Dec. 21, 2021. https://www.businessofapps.com/data/messaging-app-market/ (accessed Jan. 02, 2023).

[7] “Most popular messaging apps 2022,” *Statista*. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/ (accessed Jan. 02, 2023).

[8] O. Peinado and D. Feeney, “Comparison of voice systems for Human Space Flight and Satellite Missions,” in *2018 SpaceOps Conference*, Marseille, France, May 2018. doi: 10.2514/6.2018-2361.

[9] “Welcome to Python.org,” *Python.org*. https://www.python.org/ (accessed Dec. 28, 2022).

[10] “Flask,” *Pallets*. https://palletsprojects.com/p/flask/ (accessed Dec. 28, 2022).

[11] “JavaScript | MDN.” https://developer.mozilla.org/en-US/docs/Web/JavaScript (accessed Dec. 28, 2022).

[12] “Introduction | Socket.IO.” https://socket.io/docs/v4/ (accessed Aug. 30, 2022).

[13] P. J. Leach, R. Salz, and M. H. Mealling, “A Universally Unique IDentifier (UUID) URN Namespace,” Internet Engineering Task Force, Request for Comments RFC 4122, Jul. 2005. doi: 10.17487/RFC4122.

[14] “Flask-SQLAlchemy — Flask-SQLAlchemy Documentation (2.x).” https://flask-sqlalchemy.palletsprojects.com/en/2.x/ (accessed Dec. 28, 2022).

[15] “Appropriate Uses For SQLite.” https://www.sqlite.org/whentouse.html (accessed Sep. 20, 2022).

[16] “Quickstart — Flask Documentation (2.2.x).” https://flask.palletsprojects.com/en/2.2.x/quickstart/#sessions (accessed Sep. 21, 2022).

[17] M. O. contributors Jacob Thornton, and Bootstrap, “Bootstrap.” https://getbootstrap.com/ (accessed Dec. 28, 2022).

[18] “Font Awesome.” https://fontawesome.com (accessed Dec. 28, 2022).

[19] Contributors, M. Otto, and J. Thornton, “Cards.” https://getbootstrap.com/docs/5.2/components/card/ (accessed Sep. 01, 2022).

[20] F. Schiffner, A. Bertard, M. Beer, and M. Töpfer, “Openvocs, a light-weight Voice Communication System for Space Mission Control,” in *SpaceOps 2023 Conference*, Dubai, UAE, Mar. 2023.