Laurenz Lazarus, BSc

Studo Chat & Wiki

Attempt to improve a growing chat application

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Frank Kappe

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Kainbach bei Graz, October 2020

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Studo Chat is a chat application within the mobile Studo app. It aims to automatically connect the students who benefit the most from each other. This thesis aims to improve the chat in order to further increase the benefit for the students. In contrast to other theses, this thesis tries to do this by two different approaches: improving the already existing Chat Wiki and implementing the new Chat Search function. Another distinctive feature of this work is that the evaluation of the two improvement attempts is not evaluated by user surveys as usual but follows a data-driven approach. Since Chat Wiki was enhanced, an A/B-Test was used for evaluation. For the new Chat Search feature a comparison of the user interactions before and after the feature was released, were used for the evaluation. Furthermore, the thesis gives insight into the structure and functioning of the chat, regarding User Interface (UI) and User Experience (UX) as well as certain software design decisions and implementation details.

Contents

List of Figures

List of source codes

List of Tables

1 Introduction

At the beginning there is always a goal, the goal of this work is to improve the Studo Chat to enhance the experience of the users. This thesis will guide through the journey with each of the following chapters. Chapter 2 introduces the reader into the world of network layers, with a special focus on the transport layer and the application layer. The goal of this chapter is to explain why and how WebSockets revolutionized not only the world wide web but also how other client-side applications communicate with servers and why Studo Chat relies heavily on them. Chapter 3 is all about the Studo Chat, how the chat and its UI is structured and organized as well as providing some implementation details. The chapter provides fundamental knowledge about the chat and how users can use it, this knowledge is needed to understand the chat and the following improvement attempts. Chapter 4 explains why and how it was tried to enhance the Chat Wiki, how the A/B-Test was set up, and how the collected data was analyzed. The chapter 5 describes how and why the new feature called Chat Search was implemented and how a data-driven approach was taken to evaluate its impact on user behaviour. The last chapter, chapter 6, provides the answer to the question if and how the two improvements attempts did improve anything as well as providing a short dive into what else could be improved in the future.

2 Network Layer Theory

This chapter is all about technical theory, standards, and a short dive into history of the network layer theory. This will help to understand why certain technologies were used instead of others as well as further deepen the knowledge of the reader.

2.1 Transport Layer

As stated by the International Organization for Standardization (1994, pages 28–29), the International Organization for Standardization (ISO)-Open Systems Interconnection (OSI)-model consists of 7 network layers. For this work, four of them are particularly interesting, the fourth layer called Transport Layer, the fifth layer called Session Layer, the sixth layer called Presentation Layer, and the seventh layer called Application Layer. For simplicity and to better match the requirements of this work, layer 5 to layer 7 are combined into one layer called Application Layer. As Serpanos and Wolf (2011, pages 141–142) say in contrast to the Internet Protocol (IP) whose main purpose is to connect systems over the internet, the main purpose of the Transport Layer is to connect multiple applications that run on these systems with each other. A very common example for this would be multiple systems, all running Web browsers whose downloading a different Website from a single Web server. In this section, the two most important protocols of the Transport Layer called User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are discussed.

2.1.1 UDP

UDP is a simple and fast transport protocol which uses datagrams to send messages. A datagram consists of two parts: header and payload. The header contains source port, destination port, length as well as an optional checksum, the payload contains the data to be transmitted. It does not require any connection procedure whatsoever, therefore it is possible to send packages right away. However, this comes at the price of missing reliability and error correction. There is no functionality to counter package loss, reordering, or delaying. Even if the checksum is given but does not match, which points to a network transmission error, the package is declared lost and discarded. It applies best on applications where the newest messages are more important than the previous ones and lost messages are therefore tolerated. Real-time audio/video or video games are common use cases for UDP (Postel, 1980; Serpanos and Wolf, 2011, pages 141–143; Kumar and Rai, 2012, pages 21–22).

2.1.2 TCP

In comparison to UDP, TCP is a more advanced transport protocol. A TCP message consists of a header and a data payload. The header contains the source port, destination port, sequence number, acknowledgement number, data offset, reserved bits, control bits, window, checksum, urgent pointer, options, and padding. Before a data transmission is possible, a connection must be negotiated. Therefore a three-way handshake must be successfully completed. This makes TCP slower than UDP however, TCP not only ensures that all messages arrive the receiver's application with intact integrity but also in the correct order. This is possible through the use of the sequence number, acknowledgement number, and checksum. A transmitter will send a specific message with a specific sequence number to the receiver until the receiver responds with a successful acknowledgement for this message with the specific sequence number. This guarantees that data sent from a server to a client will be transmitted correctly, therefore the best applications for TCP are were the integrity and completeness of the transmitted data is more important than their up-to-dateness. This also provides the main reason

why TCP is the base transport protocol for many application protocols like Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Secure Shell (SSH), Internet Message Access Protocol (IMAP), etcetera (Postel, 1981; Serpanos and Wolf, 2011, pages 141–143; Kumar and Rai, 2012, pages 22–24).

2.2 Application Layer

The ISO OSI 7 layer network model contains separate layers for the session, presentation, and application International Organization for Standardization (1994, pages 28–29). In this work, these three layers are combined into one layer called the Application Layer because many modern application protocols handling the aspects of sessions and presentation by themselves, like the HTTP which gets discussed later on. This work, therefore, follows the communication layers of the internet architecture described in Network Working Group, Internet Engineering Task Force, and R. Braden (1989, pages 1–10). In this section, the history behind HTTP two way client-server communication is discussed to provide knowledge of why Hypertext Transfer Protocol Polling (HTTP Polling) and Hypertext Transfer Protocol Long Polling (HTTP Long Polling) were invented and why these technologies have been replaced by the WebSocket technology.

2.2.1 HTTP Polling and HTTP Long Polling

HTTP is a stateless protocol that was originally designed for one use case: retrieving Hypertext Markup Language (HTML) documents from a web server (World Wide Web Consortium, 1991). The protocol was further developed and gained more functionality nevertheless, HTTP kept its original base which means that the server only sends data to the client if the client asks for it, or more technically: if the client sends a request for a certain piece of data to the server (Network Working Group, Fielding, et al., 1999, pages 12-13).

However, technologies further evolved and a need for server-initiated communication from the server to a client was developed. To satisfy the need, HTTP Polling and HTTP Long Polling were created, with them it is possible to update information on a website without the need of the user sending a request manually. Therefore it was possible to create real-time web applications like a stock ticker, chats, or similar. The way HTTP Polling works is that the client continuously sends requests to the server, every request with a certain time delta between them. The server responses to every request either with new information if one is available or with an empty response otherwise. HTTP Long Polling works similarly, with the difference that the server does not respond immediately with an empty response if no new information is available. Instead, the server waits until either new information is available or the timeout runs out in which case the server sends an empty response. If the client gets a response from the server, the client immediately sends another request to the server (Pimentel and Nickerson, 2012, pages 45–46).

The fundamental problem with these techniques is that they do not conform with the initial idea of the HTTP or as Internet Engineering Task Force, S. Loreto, et al. (2011, page 1) says:

> "On today's Internet, the Hypertext Transfer Protocol (HTTP) is often used (some would say abused) to enable asynchronous, 'server- initiated' communication from a server to a client as well as communication from a client to a server."

This leads to problems with scalability as well as efficiency for two main reasons (Internet Engineering Task Force, Fette, et al., 2011):

1. The server needs to create multiple TCP connections for every client, basically one new connection for every information transmission.
2. There is a lot of overhead due to the fact that with every request the client sends and every response the server sends, there is the HTTP-header send as well.

By taking a closer look at Network Working Group, Fielding, et al., 1999, it can be seen what a minimum request looks like:

```
Request = Request-Line
```

Where `Request-Line` is defined by:

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

So in theory the absolute minimum for a HTTP 1.1 request header would look similar to this example:

```
GET http://a.co HTTP/1.1
```

Which results in a total of 26 bytes (including the 2 bytes for `CR` and `LF` at the end).

This however is very theoretical for multiple reasons:

1. Hypertext Transfer Protocol Secure (HTTPS) is nowadays standard and successfully took over from HTTP with more than 50% global usage in Google Chrome and Mozilla Firefox since 2017, and increasing (Felt et al., 2017). This is a positive trend for security but also adds further weight to the overhead.
2. Usually, the content of the root directory is not requested for your real-time application. In theory, a dedicated domain could be registered but this adds a lot of effort and complexity. It is much easier to simply use another path in the already existing domain.
3. Most applications need some kind of state, e.g.: chat applications need at least a session and probably some sort of authentication. To achieve this you need even more HTTP header fields like the `Cookie` header field.
4. There are several other HTTP header fields that are needed to ensure that everything works as indented for example: making sure that the right encoding/decoding is used.

All of this usually leads to much bigger sizes than just 26 bytes.

For the response the minimum would be:

```
Response = Status-Line
```

Where Status-Line is defined by:

```
Status-Line =
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

So again, in theory the absolute minimum for a HTTP 1.1 response header would look similar to this example:

```
HTTP/1.1 200 OK
```

Which would lead to a minimum of 17 bytes including CRLF. Just like the request, this is very theoretical for the same reasons as for the request header size mentioned above and would be much larger in a real-world application.

So this leads to an absolute minimum overhead of 43 bytes per update, this may not seem much but again, in a real-world application this would be much higher. It is very dependent on the use case as well as the exact implementation but as google.com, 2009 states:

> "Request headers today vary in size from ~ 200 bytes to over 2KB. As applications use more cookies and user agents expand features, typical header sizes of 700-800 bytes is common."

To perform a calculation that has more relevance in real-world applications, it can be assumed that an example implementation would have a typical request and response header size of $2 * 700 = 1400$ bytes. This is combined with the fact that Studo Chat faces up to 116.8 Million such requests and responses per week. So it totals at

$$1400 * 116.8 * 10^6 = 163.53 \; Gigabytes$$

of raw overhead traffic, in a single week.

It should be noted that this calculation does not reflect the fact that neither HTTP Polling nor HTTP Long Polling is capable of "real" two-way communication. So there is additional overhead when the client asks for new data but the server does not have anything new yet. This additional overhead may not be much for applications with a static data stream like stock tickers but for others like chat application, this can be quite significant.

2.2.2 WebSockets

To resolve these issues which arrive with HTTP Polling and HTTP Long Polling, the Internet Engineering Task Force (IETF) published the document RFC6455 in December 2011 which contains the specifications to the WebSocket Protocol. The main goal of the WebSocket protocol is to replace abusive techniques like HTTP Polling with a protocol that enables full-duplex two-way communication between client and server without the need for multiple TCP connections (Internet Engineering Task Force, Fette, et al., 2011, page 1–4).

The WebSocket protocol consists of two parts: the initial handshake and the following data transfer. The design behind the handshake keeps the compatibility with HTTP servers in mind, therefore the handshake is an HTTP GET request which tells the server that the client wants to upgrade the current HTTP connection to a WebSocket connection, a partial example of such a request could look like this:

```
GET /some/path HTTP/1.1
Host: www.some-host.com
Connection: upgrade
Upgrade: websocket
(...)
```

The server then sends a request in which the server confirms the upgrade by stating that a protocol change is about to happen. Similar to the request, the response of the server is an HTTP response as well, with the first line being the HTTP Status-Line, with the status code being 101. Any other code than 101 would indicate that the WebSocket handshake was not successful and the upgrade was canceled by the server. A partial example of such a response could look like this:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: upgrade
(...)
```

After successful completion of the handshake, both the server as well as the client can independently send information at will. This handshake design results into the benefit that a client can communicate to the server via the WebSocket protocol by sending messages to the same server port, the client uses to send HTTP requests too (Internet Engineering Task Force, Fette, et al., 2011, pages 5–9).

The data transfer can start right after the successful handshake, although during the data transfer not only data is sent back and forth between client and server. Conceptually, one submission from client to server or vise versa is called a message. A message is made of one or more frames and every frame has its own type which for example can be one of the following:

- Textual type
- Binary data
- Control frames

Textual type content is typically encoded in Unicode Transformation Format (UTF)-8 and has the advantage that, depending on the implementation, it could be directly displayed by the application without further processing. Control frames are required by the WebSocket protocol itself to function properly. They do not carry any data for the destination application but carrying signals used by the WebSocket protocol itself to for example signal that the connection should be closed. Binary data is not decoded by the WebSocket protocol but by the application itself (Internet Engineering Task Force, Fette, et al., 2011, pages 5–6). Consequently, this option offers a lot of optimization potential for example in terms of the size of the data to be transmitted and/or decoding time, achieved by the use of custom compression algorithms.

Liu and Sun (2012, page 800) state that:

> "The efficiency is the key issue of real-time data transmission; it's also the important standard for evaluating whether a protocol is suitable for real-time data transmission."

The design of the WebSocket protocol does not only reduce the amount of network transmission overhead as shown previously, but it does reduce latency as well. As shown by Pimentel and Nickerson (2012, pages 49–52) HTTP Polling and HTTP Long Polling can perform as well as WebSockets at

over-the-internet applications with known static refresh intervals. Although as soon as the underlying network latency is equal or more than half of the refresh interval, HTTP Polling and HTTP Long Polling starts struggling and WebSockets standing above.

By comparing HTTP Long Polling with WebSockets within a real-time collaboration application over the internet Marion and Jomier (2012) showed that WebSockets achieved an over 50% lower average latency compared to HTTP Long Polling. Furthermore, in the experiment, it was also shown that WebSockets could handle update rates of 60 messages per second whereas HTTP Long Polling only a little under 6 messages per second. In an additional internal benchmark test, it was demonstrated that the WebSocket implementation was capable of over 1000 messages per second.

Puranik, Feiock, and Hill (2013, pages 114-116) stated that the increase of the main memory consumption of the client varied between 10% to 20% if the client uses HTTP Long Polling in comparison to using WebSockets. This result was achieved within a test in which the client did not process the data in any way, so the increased main memory consumption was only caused by the different transfer protocols. Furthermore, it could be observed that the overall bandwidth consumption with HTTP Long Polling was around 40% higher than with WebSockets. This was explained with the facts that firstly with the HTTP Long Polling implementation the minimum size of messages equals 256 bytes and secondly, as already discussed, HTTP Long Polling suffers from the bigger HTTP header overhead in comparison to WebSockets. In addition to this it was shown that with WebSockets, the average package size was 31.7% lower in comparison with HTTP Long Polling. Lastly, it was pointed out that due to the increased network overhead and latency of HTTP Long Polling, 99.18% of the sent samples were not received by the client in time.

When comparing HTTP Long Polling with WebSockets in regards of energy consumption, which is especially relevant on mobile platforms, then it turns out that WebSockets compared to HTTP Long Polling can reduce the amount of energy consumption by 51.02% when the client is connected to the internet via Enhanced Data Rates for Global System for Mobile Communications (GSM) Evolution (EDGE). If the client is connected to the internet via Third Generation Telecommunication Networks (3G), WebSockets compared to

HTTP Long Polling can reduce the amount of energy consumption by 49.70%. A potential reason for the high difference in energy consumption could be caused by the fact that with HTTP Long Polling the client needs to disconnect and reconnect every time the server responds to a request. This could cause the corresponding radio module (EDGE, 3G) to shut down and boot right up again, which would cause a lot of lost energy. In case the client is connected to the internet via Wireless Local Area Network (LAN) (WLAN) the difference in energy consumption is negligible. This is caused by the well-implemented energy-saving functionality of IEEE 802.11 (Herwig, Fischer, and Braun, 2015, pages 344-346).

3 Studo Chat

The principles and design of Studo Chat were crafted with efficient communication between students in mind. This chapter guides through the UI of Studo Chat and explains how it achieves to connect the students who benefit the most from each other. As well as how it provides a platform, on which students can share their knowledge and experience. Although all example figures show iOS devices, all explanations are valid for the Android ecosystem as well.

3.1 User Interface

Studo Chat is designed to allow students a quick and easy way to communicate. Its UI is optimized for mobile usage to best fit the modern needs of students. Figure 3.1 shows the Channels View, here the user can select all channels which the user is currently subscribed to. At the time of writing, Studo Chat supports three main categories of channels which users can subscribe or will automatically be subscribed to:

- Region-based channels
- Course-based channels
- Private channels

Region-based channels are channels that every student gets automatically subscribed to because of several region-based criteria like coun-

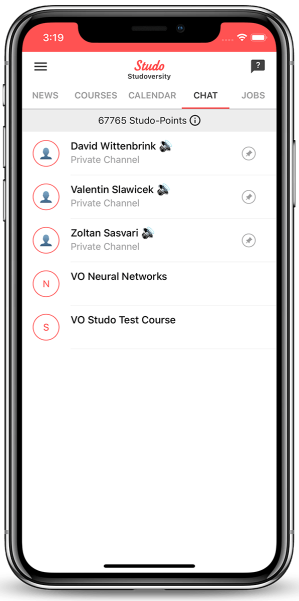

Figure 3.1: Chat Channels View

Figure 3.2: TopicsView of the Graz channel

Figure 3.3: MessagesView of a topic within the Graz channel

try, city, university, or study. So, for example, all students of the Graz University of Technology which are enrolled in computer science master study, are automatically subscribed to the channels "Technische Universität Graz", "Graz" and "MA Computer Science". The automatic subscription logic of course based channels is very similar to the region based channels, except that the students will be automatically subscribed if they enroll a course. An example here would be "VO Neural Networks" or "KU Machine Learning". Private channels are very different from all other channels, these channels have more of an Instant Messaging (IM) character like common messaging apps for example Telegram, WhatsApp, etcetera. In a private channel, two users can privately communicate with each other. If the user (a) subscribes to a private channel with the user (b), the user (b) will get automatically subscribed to the private channel as well. However, the user (b) can then decide if the private channel with the user (a) is accepted or rejected. If accepted, the communication can start, otherwise, the channel gets closed.

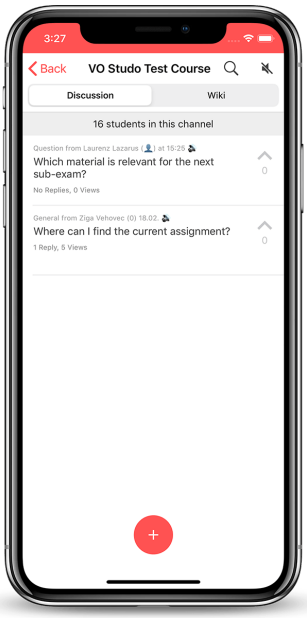

Figure 3.4: Discussion section of Studo Test Course

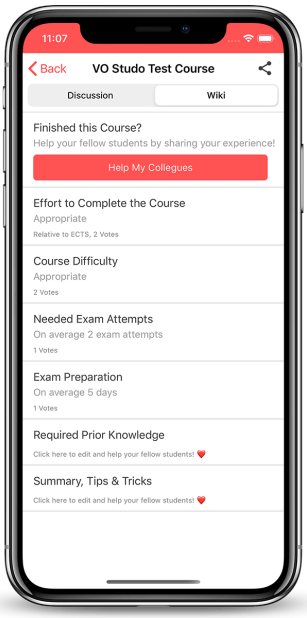

Figure 3.5: Wiki View section of Studo Test Course

Region-based channels offer a discussion platform for users, where they can create new topics which are conceptually similar to conversation threads of a forum. A typical example of a region-based channel would be the channel of the city Graz as it can be seen in figure 3.2. Within a topic, users can add messages to the topic and thus reply to questions or discuss the topic. It is also possible to share documents like images, Portable Document Format (PDFs), etcetera to further improve communication capabilities. For example, it is a lot easier to discuss a mathematical problem if you can use a pen and paper to write the problem down, take a quick picture of it and share the picture with your colleges than it is if you only can send plain-text messages. There is also a mechanic to up-vote certain topics/messages, so users can indicate which questions/answers are of particular interest. If a user adds a new topic, this user can then mark one of the answers of the topic as the "Accepted Answer" of the topic, this message will then be highlighted so other users can easily identify it, an example of this can be seen in figure 3.3.

As can be seen in figure 3.4 course based channels are special because they not only contain a discussion section, like the region based channels, but also a wiki section. The Chat Wikis ambition is to provide a more structured platform where students can share their experience about a particular course and therefore help other students to better estimate the amount of work a particular course will take. The knowledge of how much effort every course will take is needed to structure a semester most efficiently. The Wiki also provides knowledge about the effort a certain exam will take, what prior knowledge other students find useful, and some general tips and tricks for this particular course. An example of the Studo Chat Wiki can be found in figure 3.5.

As already mentioned above, private channels are very different, they do not contain any discussion or wiki sections. Figure 3.6 shows an example of such a private channel, it features a private communication channel between 2 users, which can share text messages as well as documents. Basically, it is a topic which only 2 users have access to.

Therefore private channels offer a good way for students to discuss a certain topic or problem in detail without disturbing other users with the conversation. Private channels also enable users to discuss solutions to a problem in detail that could not be discussed publicly, like the solution of certain programming homework which would have a high potential of plagiarism.

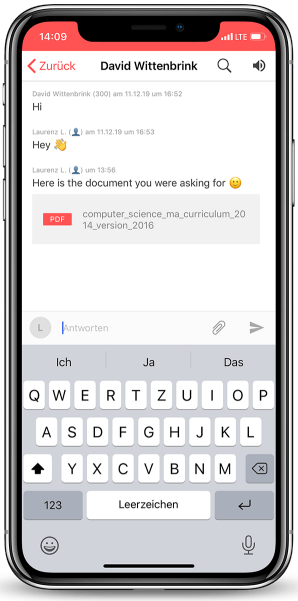

Figure 3.6: Private Channel view

3.2 Layout

Figure 3.7 shows the Studo Chat Wiki view, here Students can get information about several aspects of the selected course as already discussed in section 3.1. The first marked section (1) of the figure shows the Navigation-Bar which contains three elements:

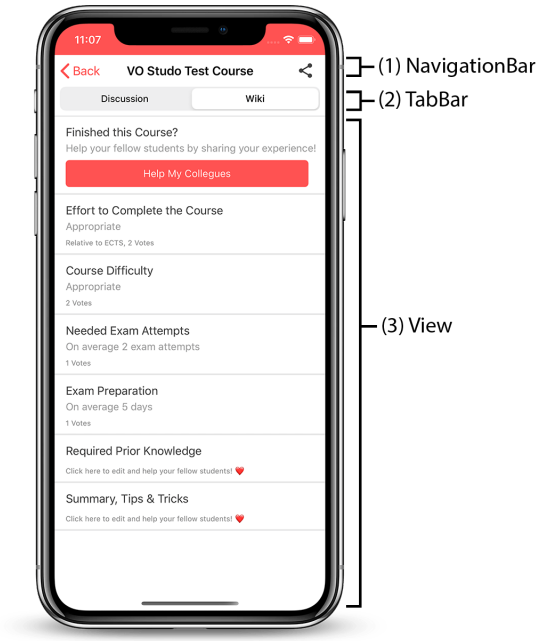

Figure 3.7: Explanation of the basic layout

- Back Button: `< Back`
- Title of the view: `Vo Studo Test Course`
- NavigationBar Buttons: In this case a single button which provides share actions to the user

If a user touches the back button, an animated back-navigation starts which presents the previous view. In this example, the app shows the content of a channel, the VO Studo Test Course channel, so the previous view would be the Channels View. This holds for all channels because all channels are accessed from the Channels View.

The course "VO Studo Test Course" is a Studo internal course, with this test course developer can test all aspects of the Chat and Wiki in a fully deployed production environment, which means on production server infrastructure while all other users are using it as well, however without any user noticing anything. With this strategy, it is also possible to test new features on production servers without any user noticing it.

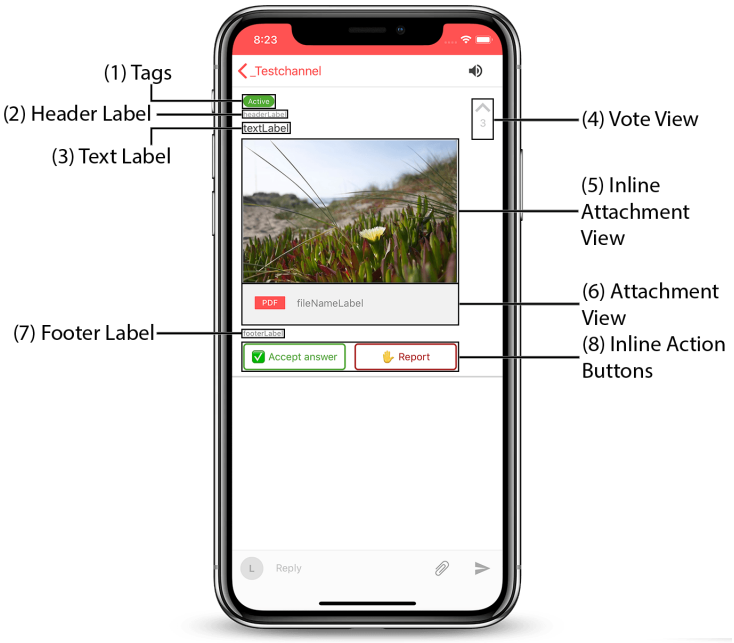

Figure 3.8: Explanation of the message view layout

The second marked section (2) of figure 3.7 shows the TabBar. It holds a selection element with its help users can switch between different views. In this particular example, the user can switch between the discussion view and the wiki view of the Vo Studo Test Course, and currently, the wiki view is selected. The TabBar is special because it is only available on course-based channels. All other channel types do not feature more than one view and therefore do not need a way of switching between different views.

Section (3) which also happens to be the last section, presents always the view which is selected by the TabBar in (2). This figure shows the wiki view of the VO Studo Test Course channel because this view is currently selected in (2).

Figure 3.8 shows the layout of the message view, which is the view with the most information for the user. The first element of the message view are the (1) Tags, tags are used to inform the user about special attributes of a certain message. In the example shown in figure 3.8, only one "Active" tag is shown but one message can have multiple tags. Possible examples

for tags are "Accepted" which indicates to users that the message with the tag is the accepted answer to the question of the current topic. Another example are the "In Planning" tags which are used by the Studo team to indicate that a certain feature request of the users which is located in the `Studo Chat Feedback`-Channel, has made it into the development backlog.

`(2) Header Label`, `(3) Text Label`, and `(7) Footer Label` are the 3 labels of the message view, which are used to display text information. It depends on the style and use-case of the message which label shows which information or if a label is shown at all. In figure 3.6 for instance, the `(2) Header Label` is used to display the sender, the senders Studo Points, and the send-time and -date of the message. The `(3) Text Label` shows the actual message the sending user did write and `(7) Footer Label` is not used and therefore hidden. But in figure 3.7, the second to fifth message use the `(2) Header Label` as a description of the message, the `(3) Text Label` displays the information and the `(7) Footer Label` shows some more information about the number of votes for each message. It also has to be noted that these messages using a different style than the messages of figure 3.6, message styles are discussed in section 3.3 in more detail.

In the upper right corner of the message view is the `(4) Vote View` placed. With this view, the users are able to vote on a certain message and therefore increase or decrease the vote count of the message. The higher the vote count of a message the higher is the informational value of the message. The view also informs the user about the current vote count, in the example seen in figure 3.8 the current vote count is 3. Currently, there are three supported voting modes:

- `Up and Down`
- `Up`
- `None`

In figure 3.8 an example of the `Up` mode is shown, where the user is only allowed to place a positive vote on the message. In the case of the `Up and Down` mode, an additional negative vote button in form of a down-pointing arrow is placed under the vote counter. Consequently, the users are able to place a positive or a negative vote onto a message, with that the vote counter gets increased or decreased by one. Is the mode set to `None` the

whole `(4) Vote View` will be removed from the message view and therefore disabling the voting feature. When a user has voted on a message, this is visually indicated by coloring the corresponding arrow as well as the vote counter. So if the user up-voted the message, the up-pointing arrow, and the vote counter will be colored red.

If a user sends a message with an image, like a photo or a screenshot etcetera, attached to it the `(5) Inline Attachment View` is visible and shows the attached image. If the user clicks on the `(5) Inline Attachment View`, a new view called attachment view is presented which shows the attachment image in full size. In this attachment view, the user is also able to perform several actions with the image like save it to the device, print it, etcetera.

In case a user attaches a file to a message which is not an image, for example, a PDF or a word processor file like a DOCX, the `(6) Attachment View` is visible instead of the `(5) Inline Attachment View`. The `(6) Attachment View` shows the file type as well as the file name of the attachment and like the `(5) Inline Attachment View`, it also presents the attachment view if the user clicks it.

Both `(5) Inline Attachment View` and `(6) Attachment View` can also be visible simultaneously to for example also present the file name and the file extension of an image. In the future, there might also be the possibility to let the server automatically create a preview image of certain uploaded attachments like PDFs, DOCXs, XLSXs, etcetera which then allows users to get a feeling for the content of an attachment without the need to download and open the file beforehand.

At the very bottom of the message view, are the `(8) Inline Action Buttons` located. The example shown in figure 3.8 displays two actions called ”Accept answer” and ”Report” which allows the user to accept or report the message. These are server defined actions that a user can perform on a certain message. There are even more actions a user can perform if the user long clicks a message, then a selection of actions that can be performed gets displayed. An example of this can be seen in figure 3.9. These actions as well as the `(8)Inline Action Buttons` and their difference will be discussed in more detail in section 3.3.

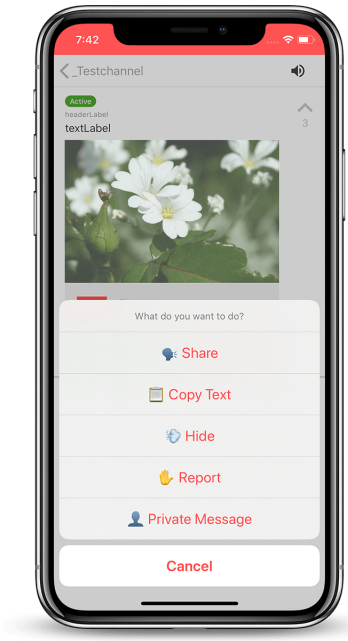

Figure 3.9: A selection of message actions a user can perform

3.3 Server driven Rendering

Studo Chat was designed with the highest possible efficiency and flexibility in mind. To achieve this, it was decided that the client basically just executes the commands of the server and has no business logic whatsoever. So the server not only decides what the client displays but also how the client displays the information and how the user can interact with it.

One example of this flexibility are the message items, these items are stored as `ClientMessage` objects in realm on the client-side, as it can be seen at listing 1. A `ClientMessage` contains a lot of properties that are all defined by the server. Since the number of properties is not insignificant, they are divided into groups to simplify the process of understanding their functionality.

The first group is about database Identifiers (IDs) used to relate and identify objects, like in most databases. It contains two properties:

```
class ClientMessage: Object {
  var id = ""
  var topicId = ""
  var header = ""
  var text = ""
  var htmlText = false
  var downloadUrl = ""
  var fileName = ""
  var footer = ""
  var searchTerm = ""
  var votes = 0
  var hidden = false
  var sortScore = 0.0
  var voteState = VoteType.NONE
  var allowedActionIds = [String]()
  var allowedInlineActionIds = [String]()
  var tagIds = [String]()
  var actionParameters = [String: String]()
  var style = ChatPayloadStyle.STANDARD
  var inlineImageUrl = ""
  var inlineImageScale = 1.0 // width / height
  var inlineImageMaxWidth = 0
  var votes = 0
  var showAttachmentLayout = false
}
```

Listing 1: ClientMessage realm object declaration in Swift

- `id`
- `topicId`

They are the usual IDs used to identify every single message object.

The second group is all about content and information which the user can read. This group consists of:

- `header`
- `text`

- footer
- fileName
- tagIds
- inlineImageUrl
- inlineImageScale
- inlineImageMaxWidth
- showAttachmentLayout

The first three properties are straight forward, the header contains the text of the (2) Header Label, the text contains the text of the (3) Text Label and the footer contains the text of the (7) Footer Label shown in figure 3.8.

The fileName member field is used to store the file name and file extension of the attachment of the message for example ”someFile.pdf” or ”someImage.jpg”. It is used by the (6) Attachment View seen in figure 3.8 to present the name of the file and its extension.

The String-Array called tagIds is used to store all the IDs of the ClientChatTags contained by the message. The tagIds member field is used by the (1) Tags view seen in figure 3.8. Although this example shows one tag only, multiple tags are supported.

```
class ClientChatTag: Object {
 var id = ""
 var text = ""
 var color = ""
 var filled = true
}
```

Listing 2: ClientChatTag realm object declaration in Swift

As it can be seen in listing 2, a ClientChatTag contains besides the id member field, multiple other member fields to customize the appearance of the tag. Namely, the text which stores the text shown by the tag, color which defines the color of the tag and filled, a Boolean which defines if the background of the tag is filled or not. This functionality is shown in figure 3.10, where the left tag is filled and the right tag is not filled. In the

Figure 3.10: Explanation of the message view layout

case of the tag being filled, the color of the text is white and if the tag is not filled, the text has the same color as the background of the tag.

So tags are one of many examples of how the server-side rendering works in Studo Chat. All tags and their visual representation are defined by the server and the client presents the UI to the users. The reason behind the separate `ClientChatTag` class in comparison to simply inline the data into the `ClientMessage` lies in data reduction. There is a very limited set of different tags, currently less than ten. But these ten tags are present in thousands of messages. And since these messages are sent from the server to the client, this not only reduces storage on the client-side but also reduces the needed network bandwidth. Another benefit of this would be that in case one of the tags will get changed, for example to another color, only one database object needs to be changed.

With the next three member fields called `inlineImageUrl`, `inlineImageScale` and `inlineImageMaxWidth` are the visuals of the `(5) Inline Attachment View`, which is displayed in figure 3.8, defined. `inlineImageUrl` contains the Uniform Resource Locator (URL) of the image preview that should be displayed, `inlineImageScale` defines the ratio of width and height of the image preview and `inlineImageMaxWidth` defines the maximum width of the image preview. These fields are helping the client to calculate the dimensions of the `(5) Inline Attachment View`. The preview images presented by the `(5) Inline Attachment View` are heavily compressed Joint Photographic Experts Group (JPEGs) with their resolution reduced to a maximum of 600 pixel width. This helps not only to reduce the size of the image cache of the clients but also reduce the needed network bandwidth which also improves the UX since the images are loading a lot faster.

In order to be able to inform the user about the current vote count, the member field `votes` is used. It contains the number of positive votes minus the number of negative votes. This number is then displayed by the vote

label placed within the `(4) Vote View` shown in figure 3.8. In this example, the current vote count is 3.

The last member field of the second group is called `showAttachmentLayout`. As the name already suggests, it controls if the `(6) Attachment View` displayed in figure 3.8 is shown or not. Consequently, the `(6) Attachment View` is only visible if `showAttachmentLayout` equals `true` and the `fileName` member field is set. This allows the server to define if an attachment needs the file name and file extension to be displayed or not. In the case of an image, the file name and file extension are not relevant and therefore the `showAttachmentLayout` is usually set to `false` for image type attachments. In the case of PDFs attachments, there is currently no preview image since the rendering of PDFs is not yet implemented on the server-side. Therefore only the `(6) Attachment View` is shown. But when the rendering of PDFs is supported, `showAttachmentLayout` will stay `true` since for document types like PDFs, the name of the document and its extension are important to the user.

In contrast to the second group which is about the information that is presented to the users, the third and last group is about how the message view and the information it carries are presented to the users and about how the users are able to interact with it. This group is made of the following properties:

- `htmlText`
- `downloadUrl`
- `hidden`
- `sortScore`
- `voteState`
- `allowedActionIds`
- `allowedInlineActionIds`
- `actionParameters`
- `style`

With the Boolean called `htmlText` the server controls if the `(3) Text Label` shown in figure 3.8 renders the String stored in the `text` member field as HTML or as plain text. More on why and how this functionality is used can be found in section 5.2.

In case the message has a file attached to it, the `downloadUrl` member field contains the URL of the file. If the file is an image, the `downloadUrl` points to the full resolution image, in contrast to the `inlineImageUrl` which always points to a heavily compressed preview version. If the user clicks on either `(5) Inline Attachment View` or `(6) Attachment View`, then the file which the `downloadUrl` points to is downloaded and presented to the user.

If the member field called `hidden` is set to `true`, then the message will not be displayed. One example of its use would be if user A reports a certain message written by another user B, then the message of user B will be set to `hidden` equals `true` on all clients of user A. This provides a number of benefits. First, if a user reports a message then it is obvious that the user does, for whatever reason, not like the message and does not want to look at it. So by removing it instantaneously, an obvious pain point of the user has been removed. Secondly, by instantly hiding the reported message, the ability of re-reporting the same message by the same user has been removed. This also saves the Studo team time for the evaluation and processing of the reports.

The member field called `sortScore` is from type Float and the client uses it to sort the messages as it can be seen in listing 3. The `sortScore` of each message is calculated by the server. This allows the server to decide which message should be at which position. Most of the time the messages are in chronological order but this is not always the case. The Course Wiki and the Chat Settings would be a good example of this. The same `sortScore` is used for the channels and the topics, which for example allows the users to pin a certain channel or it would allow the server to sort topics with an intelligent algorithm based on creation date, vote count, and other interaction criteria.

```
let messages = realm.objects(ClientMessage.self)
  .filter("topicId == %@", topicId)
  .filter("hidden == false")
  .sorted(byKeyPath: "sortScore", ascending: false)
```

Listing 3: Query the messages of a certain topic

Listing 3 shows how messages are retrieved from the local realm database.

The query filters all messages with a specific `topicId` as well as all hidden messages and sorts the result by the server defined `sortScore` in a descending way. In this example, the realm own methods for filtering and sorting were used instead of the Swift ones because they yield better performance. Especially the sorting method can make a huge difference in performance.

With the member field called `voteState`, which is of the enum type `VoteType` whose definition can be found in listing 4, the `(4) Vote View` from figure 3.8 knows in which of the three states defined in listing 4 it is currently in. So if a user has up/down-voted a message, the `voteState` would be `UP`/`DOWN` or `NONE` otherwise.

```
enum VoteType: String {
  case UP
  case DOWN
  case NONE
}
```

Listing 4: Definition of enum VoteType

The reason why `voteState` and `VoteType` were named differently is due to the scopes they were created in. `voteState` was created on the client-side to describe the current state of the `(4) Vote View`. `VoteType` on the other hand was created on the server-side and describes the type of the vote a user has placed.

The member field `allowedActionIds` is a String-Array containing the IDs of `ClientChatActions` which the user is allowed to perform onto the message. These actions can be performed by the user by simply long pressing the message view, an example of this can be seen in figure 3.9. The `ClientChatAction` class is defined in listing 5. This allows the server to be in full control of which user can perform which action on which message, since the actions are defined per user, per message. One use case of this flexibility are the actions a user can perform on his own messages only, like "Edit" and "Delete". Another possibility would be moderator actions like "Move to other Topic", which a moderator could perform on all messages which the moderator has moderator rights on, like all messages of a certain channel to name an example.

```
class ClientChatAction: Object {
  var id = ""
  var text = ""
  var nextActionIds = [String]()
  var optimisticUiActions = [OptimisticUiAction]()
  var actionChainHeadline = ""
  var filled = false
  var color = ""
  var lineSortScore = 0
  var type = ""
  var textInputRegex = ""
  var textInputPlaceholder = ""
}
```

Listing 5: Definition of ClientChatAction

`allowedInlineActionIds` is very similar to `allowedActionIds`, it also is a String-Array which contains the IDs of `ClientChatActions`. But in contrast to `allowedActionIds`, the `allowedInlineActionIds` are directly visually represented in the message view by the `(8) Inline Action Buttons` shown in figure 3.8. This visual representation informs the users about a selected set of actions they can perform as well as indirectly prompt them to perform an action, which allows a variety of use cases. An example of the indirectly prompting the user can be found in section 4.1 in figure 4.1. At `(1)` it can be seen that the indirect prompting is performed by presenting a "Accept answer"-inline action button to each answer posted to the user's question. Note that these "Accept answer"-inline action buttons are only visible to the user who wrote the question. Another example of a use case for the `allowedInlineActionIds` would be a poll type message as it can be seen in figure 3.11. Here the users can place their vote by simply tapping one of the inline action buttons. This functionality already provides a lot of opportunities but to really understand the power in which the `ClientChatAction` class provides, a closer look at its definition has to be made 5.

First up the UI specific member fields of `ClientChatAction` namely `text` which holds the text of the action, the `color` field stores the color as hex

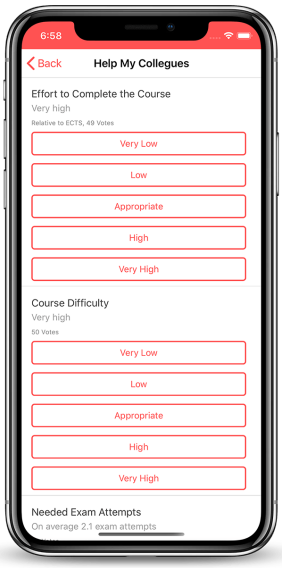

Figure 3.11: Example of a poll using inline action buttons

string. With the `filled` member field can be defined if the background of the action is colored or transparent. This functionality is very similar to the previously discussed `ClientChatTag`, an example of this can be seen in figure 3.10. The `lineSortScore` member field defines in which line the inline action button should be located. So if the `lineSortScore` equals for example 0, the inline action button is located in the first line, its located in the second line if `lineSortScore` equals 1, and so on. An example of this can be seen in figure 3.11, here the `lineSortScore` of the "Very Low" action equals 0 and of the "Very High" action equals 4. In case two or more actions have the same `lineSortScore` value, the space between them is divided equally, as it can be seen at `(8) Inline Action Buttons` in figure 3.8.

It is also possible to trigger client-side actions additionally to the server-side actions like tagging a message. This is possible with the `optimisticUiActions` member field which is an Array made of `OptimisticUiActions`. These `OptimisticUiActions` are triggered as soon as the user taps the corresponding `ClientChatAction`, regardless of the `ClientChatAction` being present in the `allowedActionIds` or `allowedInlineActionIds` member field. Consequently, these `OptimisticUiActions` are executed before the server responds

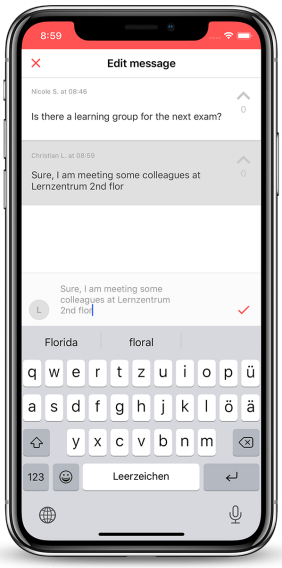

Figure 3.12: Editing a message

and therefore it does not matter if the server responds at all, hence the "optimistic" in the `OptimisticUiActions` naming. There are several example usages of `OptimisticUiActions` like a simple "Delete" action which is used if a user wants to delete one of its own messages. If executed the "Delete" action, removes the corresponding message from the client realm database. A bit more complicated is the "Edit" action, it is triggered if a user wants to edit one of its own messages. This triggers a bunch of UI adjustments as can be seen in figure 3.12. Firstly in the `NavigationBar` the "Back"-button is replaced with a close button and the title is replaced with the "Edit message". Secondly, the message which is now in edit mode is highlighted with a light gray background color. Thirdly the `ChatInputView` is activated, its text field contains the text of the selected message and the send button is replaced with an accept button. By pressing the accept button the user sets the new text of the message. But the most powerful `OptimisticUiAction` is the "Deeplink" action which can be used to trigger deep links to all views of the Studo app. A more detailed explanation of deep links can be found at 4.3. This allows a `ClientChatAction` to trigger a navigation action to another chat view or every other non-chat view within the Studo app. One example of its use can be found in section 4.3 in figure 4.3. Here the inline

action button called ”Help My Colleagues” triggers a deep link to the Wiki Addview. There are even more `OptimisticUiActions` like ”Copy Text” to copy the text of the selected message to the clipboard or ”Share” which spawns the system native sharing view to share the link to the selected message.

Another powerful feature of `ClientChatAction` is the ability to create an action chain consisting of multiple `ClientChatActions`. As it can be seen in figure 3.13, if users do long press a message, a selection of message actions is available. In this example, the user decides to file a ”Report” onto the message and this is where the action chain comes into play. Now a new selection of message actions is available, in this example, the user reports the message for ”Other Reason” and is then prompted with a dialog where the user can specify how exactly the message violates the Studo Chat Guidelines.

The action chaining is implemented with the `nextActionIds` member field of `ClientChatAction`, which is a String-Array that holds all IDs of the following `ClientChatActions`. In the example visible in figure 3.13, the `nextActionIds` of the ”Report” action contains the IDs of the actions visible in the view at the bottom left, namely ”Spam”, ”Offence”, ”Racism & Discrimination”, etcetera. Consequently, the action chain does end as soon as the user selects an action that has an empty `nextActionIds` array. `actionChainHeadline` is used to set the headline of the next action selection. In the example displayed in figure 3.13 the `actionChainHeadline` of the ”Other Reason” action is set to ”Why should this post be removed?”. This is very useful to add more context to each step.

In order to realize the dialog visible at the bottom right of figure 3.13 it was necessary to implement the `type` member field which defines the type of the `ClientChatAction`. This is not the cleanest way to model the data structure but unfortunately, the client realm database does not support inheritance. Currently, there are two supported `types` namely ”BUTTON” and ”TEXTINPUT”. All actions displayed at the top right of figure 3.13 are of `type` ”BUTTON”. The actions displayed as dialog at the bottom right of figure 3.13 are mixed. The action displayed as text field is of `type` ”TEXTINPUT” and the button called ”OK” is of `type` ”BUTTON”. A single dialog supports multiple actions of `type` ”TEXTINPUT” and ”BUTTON”.

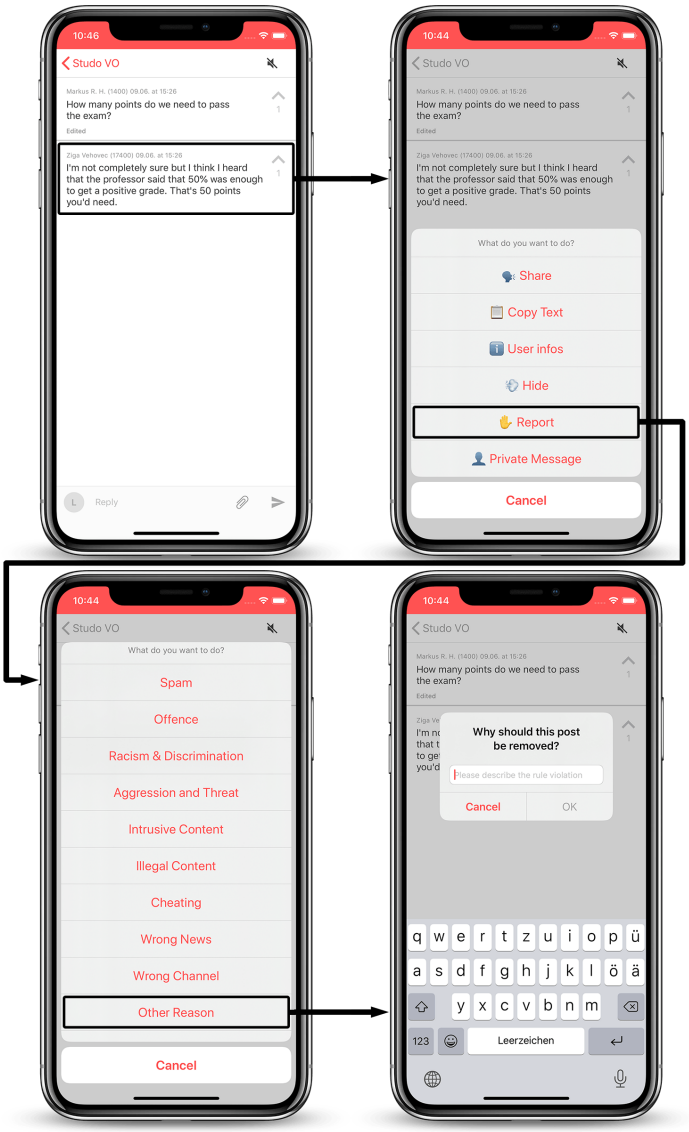

Figure 3.13: Flow of reporting a message

With the `textInputPlaceholder` a placeholder text for the text fields of the ”TEXTINPUT” actions can be defined. The `textInputRegex` is used to define a pre-filtering of the text that the users input into the text fields of the ”TEXTINPUT” actions. So as long as not all ”TEXTINPUT” actions fulfill their corresponding `textInputRegex`, all the ”BUTTON” actions are not clickable and grayed out. An example of the behaviour can be seen at the bottom right of figure 3.13, here the ”OK” action is not active since the text field is empty and the `textInputRegex` requires the text field to be not empty. As soon as the user types a character in, the ”OK” button will get active and not grayed out. So `ClientChatActions` are very powerful and are fully configurable by the server and therefore adding to the server-side rendering feature set.

Another member field of the `ClientMessage` is called `actionParameters`, it is a Map with a String as an index and a String as value. The index is the ID of the corresponding `ClientChatAction` and the value is the value carries the information needed to perform the `ClientChatAction`. In the case of the ”Share” action the entry would look like

```
"SHARE" : "Some server defined text usually the first few
words of the message ... - https://studo.co/chat/shared?
channel=816b7bd6856a08e577f177e18b129&..."
```

The first String `"SHARE"` is the index as well as the ID of the ”Share”-`ClientChatAction`. Followed by the ”:” is the value String which contains some server defined text as well as the URL to the to be shared message. Another example usage of the `actionParameters` are the deep links that are used to start a new or navigate to an already existing private channel, etcetera.

Last but by far not least is the `style` member field of the `ClientMessage` class. It is used by the server to define one of three supported message styles defined by the `ChatPayloadStyle` enum, whose definition can be found at listing 6. The `STANDARD` style is, as the name already says, the default style used for all user-created messages. The informational focus of the `STANDARD` style is at the `(3) Text Label`, with the `(2) Header Label`

```
enum ChatPayloadStyle: String {
  case STANDARD
  case BIG_HEADER
  case SECTION_HEADER
}
```

Listing 6: Definition of ChatPayloadStyle

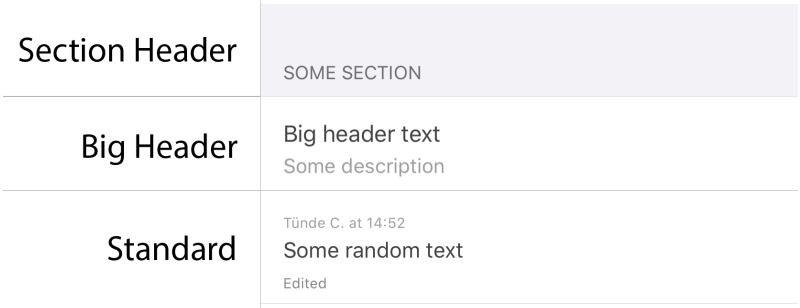

Figure 3.14: Different Chat Message styles

and `(7) Footer Label` providing only additional information. The definition of the three different labels can be found in figure 3.7. `BIG_HEADER` styled messages can be found in Course Wikis. They are focused at the `(2) Header Label` with the `(3) Text Label` providing an additional description. The `(7) Footer Label` is usually unused. With the `SECTION_HEADER` style, it is possible to create sections, this is currently used in the Studo Chat Settings view. The `SECTION_HEADER` style usually only uses the `(2) Header Label` in an upper-cased font style. The background is changed as well to further visually distinguish the message from the surrounding. An example of all three styles can be found in figure 3.14.

The server has control over what the client displays, how the client displays it, and how the users can interact with the client. This is called server-side rendering because the server defines (”renders”) the UI beforehand and the client only executes. This gains great flexibility for future improvements.

4 Chat Wiki navigation redesign

The following chapter guides through the layout design of a Studo Chat View to help the reader to better understand why a redesign of the Wiki navigation was necessary. It explains how the old navigation design was implemented and why it was implemented this way. The chapter shows what the problems of the old design are and how they might impact user behaviour. Furthermore, it presents the solution and new implementation and tells how an A/B Test was set up to measure the impact of the new solution. It also shows how the data was extracted and prepared to be analyzed.

4.1 The Old Implementation

It is one of the continuous goals of Studo to further improve the overall app experience for the users (or in this context more specifically: students), to assist them by their daily challenges and help them to successfully complete their studies. One of the newer sub-goals is to enhance the experience in Chat as well as its subcategory the Chat Wiki. Newer because Chat is the newest addition to the big feature set which the Studo app offers. The objective of Chat Wiki is to provide a platform for students to share their experiences, which they gathered during a class, with their colleges. This can enable students to better optimize their class schedule for every semester.

Therefore it is necessary to make the process of sharing experiences and knowledge as friction-less as possible to motivate as many students to share their knowledge and therefore to help as many students as possible. After reviewing the part of Chat Wiki in which the students can share the experience, a design flaw was discovered which potentially could prevent

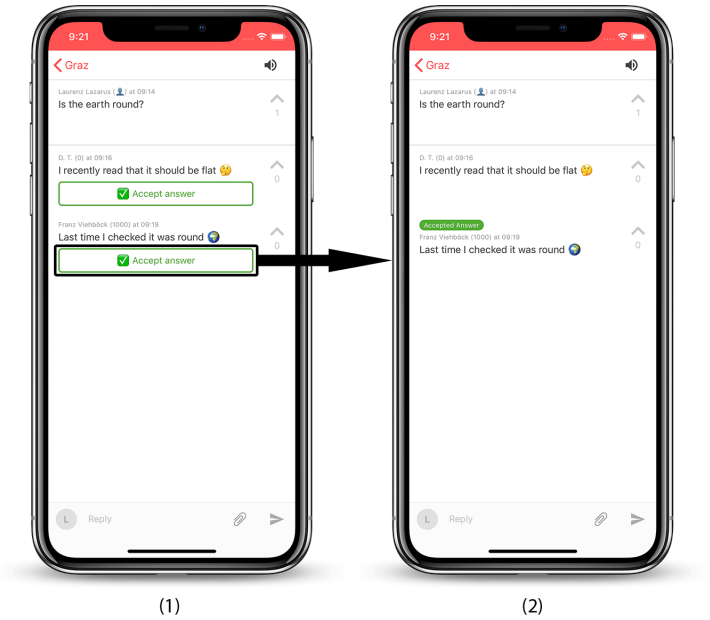

Figure 4.1: Flow of "Accept Answer" inline button

students to share their experience more than once or in the worst case, even prevent students to share their experience at all.

But first, an explanation of how it is implemented and why it was implemented this way. If users open a channel, their client sends a request to the server, the client, therefore, subscribes to this channel and the server responds with the corresponding data. That might be topics, messages, and other relevant information, depending on the type of channel. Due to the subscription, the client will receive all updates from the server instantaneously, as long as the user stays active in this channel. The subscription logic is only possible because of the usage of the WebSocket protocol which keeps an open TCP connection between client and server (as discussed in chapter 2) as long as the user stays active in the chat. This enables the UI to be very responsive, even during actions were Optimistic UI is not possible because the behaviour/response of the server can not be known before.

This responsiveness allowed the implementation of inline buttons, they are embedded into message items, trigger server requests and the corresponding

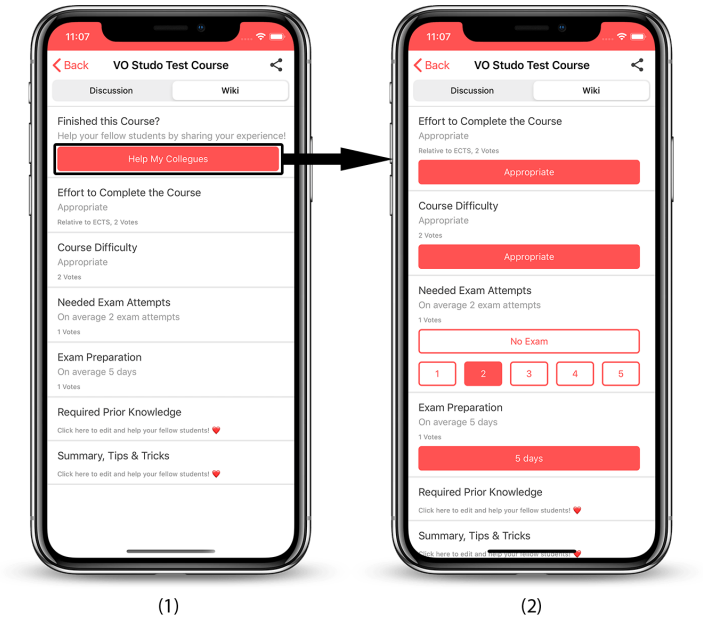

Figure 4.2: Navigation from Wiki View to Wiki Addview

server responses update the UI, even if no Optimistic UI is possible. One implemented example is the already mentioned ability of users to mark a message as "Accept Answer". So if a user creates a new question-topic, every answer to this topic will get an inline button called "Accept Answer", of course, these buttons are only visible for the creator of the question-topic, this can be seen at (1) of figure 4.1. If the user wants to mark a message as the "Accepted Answer", the user presses the button of the message, this way a request is sent to the server which marks the message as the accepted answer of this topic and responses with an update to the appearance of the "Accepted Answer"-message for all users which are currently actively subscribed to this topic, as it is illustrated in (2) of figure 4.1. The "Accepted Answer" message will then be visually marked by an "Accepted Answer" badge.

As Chat Wiki was implemented, it seemed to be a clean and easy solution to use an inline button for the navigation to the wiki add view. So if the user clicks on the "Help my Colleagues" inline button, as seen at (1) of figure

4.2, the server responds with an update for all message items in the current wiki, as illustrated in (2) of figure 4.2. In the wiki add view users can then add their knowledge and experiences. The elegance of this implementation design lies in the fact that this way no client app update is required and the server is in control of basically everything, which provides the following benefits:

- No waiting times for new feature and adaptions deployment:
  Client-side app updates take days until rolled out to 80% of our user base, server-side updates are rolled out to 100% in a matter of minutes. This makes it possible to quickly react to usage spikes or other time-bounded issues.
- Less implementation effort:
  Everything has to be done just once on the server. No double implementations on the client-side are needed. Double implementations because of the two supported mobile platforms Android and iOS.
- Fewer developer dependencies:
  Due to the fact that not every developer has the same skill set it usually needs 2-3 developers to implement a new feature: 1 server-side developer and 1-2 client-side developer. By implementing a server-side only feature, there are no dependencies and a single server-side developer can implement the feature on its own.
- All users have the same experience:
  Not all users do app updates every day, week, or even month, which results in the fact that the newest app update is never to 100% rolled out to all users. So this design results in the ability to roll out new fixes and even features to all users simultaneously.

On the server-side usually, each type of content source (for example Channel, Topic, etcetera) holds its own data source, Wiki is different. Wiki View and Wiki Addview share the same data source since they basically provide the same content with the difference being that on Wiki View all the aggregated entries of the users are just shown and can not be modified. With Wiki Addview the users can add their own experiences. By sharing the same data source it was possible to achieve the following advantages:

- No redundancies:
  By sharing the same data source, there is no need of storing all wiki

entries twice in the database. As it can be seen in figure 4.2, the "Course Difficulty"-item has the same content on Wiki View and Wiki Addview.
- No need for data synchronization:
  If there is only one database entry, there is also no need to keep the data synchronized. Otherwise, this would add a lot of complexity.

4.2 The Problem

As discussed in subsection 4.1 the old implementation design has a lot of benefits from an implementation point of view but as already mentioned it has a big design flaw from a UX point of view. To understand what the problem is and how negatively impactful it can be, it is necessary to keep the navigation pattern mention in section 3.2 in mind. Because usually the user clicks on a UI element, for example, a button, and the app responds with a navigation transition to a new view. The user then can undo this navigation with the back-button in the upper left corner of the screen, so that the app shows again the initial view. On Android, the back navigation is sometimes also possible with the dedicated hardware back-button.

Just like every other UI navigation action, the old wiki implementation requires the user to click on a UI element, in this case, a button. Though in contrast, the button click does not result in a navigation transition, instead it results in an update of the message items of the Wiki View, as discussed in section 4.1 and as it can be seen in figure 4.2. This leads to a variety of problems:

- Inconsistency in navigation animations:
  All mobile platforms have their own patterns and standard animations, including animations for UI navigation transitions. The old wiki implementation violates these standards, this results in inconsistency. This inconsistency may cause users neither to understand what just happened nor what they are supposed to do next.
- There is no way back:
  Since just the view is updated, the NavigationBar, as well as the TabBar, retain the same appearance and functionality, which further violates

the navigation patterns. The user expects to be able to revert every navigation action at any point. In the old wiki implementation the user is not able to do that because if the user clicks on the `< Back` -button, the app navigates back to the Channels View, not the from the user expected behaviour. Furthermore, if the user then navigates again to the Wiki View of the same channel, the Wiki Addview is presented again. So in fact the user has no way back to the Wiki View.

- The app is broken:
  These violations lead to the user's impression that the app is not behaving as expected and that the app must be broken. Because the users think that there is no way that the inconsistent behavior is indented from the developers.
- Losing the trust of the user:
  Apps that are obviously broken are not trustworthy to the user and thus not worth their time and effort or worse, the user may think since the app is not behaving correctly, it may also not be able to correctly save their efforts. This is fatal for a community-driven feature like Chat Wiki.

The cause of there being no way back and why this is not entirely true lies in the implementation design. If a client sends the "User wants to enter the Wiki Addview for wiki `xyz`"-request to the server, the server notes that this specific user gets now the Wiki Addview content of wiki `xyz` instead of the Wiki View content for the next 5 minutes. After this time period, the user gets the Wiki View content served again. But since there is no obvious way for the user to know what is going on, it is basically the same as if there is no way back at all.

4.3 The Solution - New Wiki Implementation

In theory, the solution is straight forward: splitting up the Wiki View content and Wiki Addview content in two separate views, this way the app performs a standard navigation transition which would remove all inconsistencies and allow the user to navigate between these two views at any time, as it can be seen at figure 4.3. The new solution should provide the same

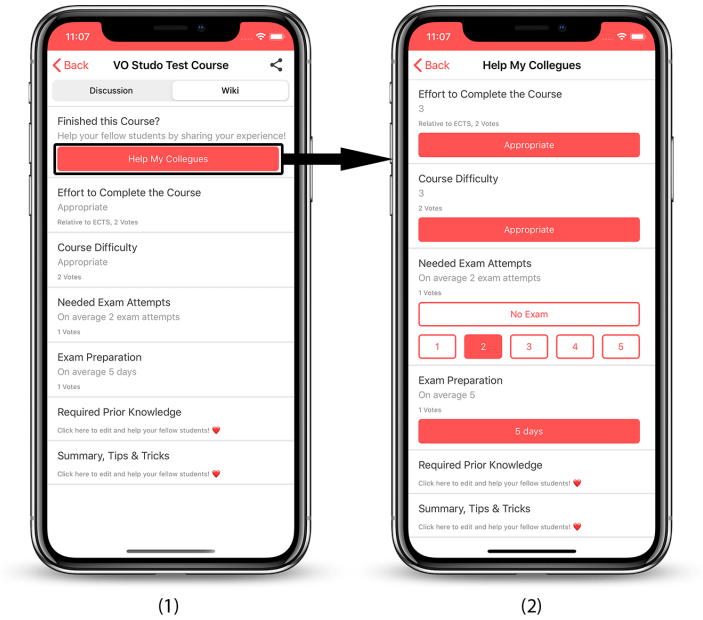

Figure 4.3: New Implementation of Navigation from Wiki View to Wiki Addview

advantages as the old implementation though, which leads to the following three problems:

1. How can the shared data source design and its advantages be kept?
2. How should the server tell the client to navigate to a new view?
3. And how could this be accomplished without the need of a client-side update?

1. How can the shared data source design be kept? The new implementation design requires the Wiki Addview topic (plus all related messages) have different IDs in comparison to the Wiki View topic + related messages of the same Channel. Otherwise, the topics and messages would be overwritten all the time in the client-side database which would cause the behaviour that if the user navigates back from Wiki Addview to Wiki View, the Wiki View would then display the messages of Wiki Addview until the Wiki Addview topic + messages would be overwritten by the server. This is unacceptable.

So instead of memorizing which user is in the "Wiki Addview Mode", the new implementation of wiki fakes all the Wiki Addview content IDs for the client. As a result, on the server-side only the Wiki View content exists. So if the server sends a Wiki Addview content update to the client, the server gathers the Wiki View content from the database, modifies the IDs so that no ID collision occurs on the client-side, and sends the modified data to the client. On the other way around, if the client sends a Wiki Addview request to the server, for example, the user did add some knowledge, the server first thing the server does is to revert the ID modifications before the server stores the new data to the database. So in fact for the client, it appears as if Wiki View and Wiki Addview were completely unrelated, but for the server, it is the same. This pattern allows keeping all the benefits, which were previously achieved by the old implementation design.

2. How should the server tell the client to navigate to a new view? 3. And how could this be accomplished without the need for a client-side update? To tell the client to perform a certain navigation action is usually trivial, but unfortunately in this case there was a deadline which prevented the ability to roll out a new app version in time. And suddenly a trivial task becomes a difficult task. The solution to this problem was to use the basic but already existing implementation of in-app deep links.

On mobile platforms like Android and iOS, apps can define custom URL Schemes which then get registered in the operating system. If a user opens such a URL, the operating system runs the app which registered the corresponding URL Scheme with the URL as a parameter. This way it is possible to transfer data between two different apps or to quickly transfer a state from a website to an app. For example, this allows to open a specific conversion on a messenger app if the user is currently in the same conversation but via the web version of the messenger on the smartphone. These URLs are called deep links.

These deep links can also be triggered by the same app which is registered for it. Combined with the ability of Studo Chat's inline buttons to open URLs which were defined by the server allows server-side defined navigation actions. Therefore it is possible for the server to tell the client to perform

a navigation action to a certain view without the need for a client-side update.

4.4 Setting up an A/B Test

To analyze if and how the new design influenced the user behaviour, it was decided to perform an A/B test, while being under the time pressure of a deadline. This was only possible because of the elegant design which allows the server to dictate the client what to display to the user as well as the fact that all clients were compatible with both implementations. So all preparations for the A/B test could be performed by server-side changes only. To set up the A/B test, all users were split up in 2 equally sized halves: One half with the old implementation and the other half with the new implementation. The splitting was achieved by taking the IDs of the users, calculating a hash of every ID, and looking at the integer representation of it. If the integer representation of the user-ID hash was even, the user was in the old-implementation group or in the new-implementation-group otherwise, as it can be seen in listing 7.

```
if (client.userId.hashCode() % 2 == 0) {
    // Old implementation group
} else {
    // New implementation group
}
```

Listing 7: Defining Group A and B

In case the user was in the old implementation group, the action of the "Help My Colleges" inline button of the Wiki View (as seen in figure 4.2), was the old "User wants to enter the Wiki Addview"-request. Otherwise, the action of the "Help My Colleges" inline button of the Wiki View was the new deep link. Of course, this means that during the A/B test, both implementations needed to co-exist on the server-side.

4.5 Analysis of the acquired data

Since the Chat Wiki Addview is basically a channel with just one topic, every section which can be seen in Wiki Addview is a message. This means that for example in figure 4.3 the sections "Effort to Complete the Course" or "Course Difficulty" are two separate messages and are stored as two separate WikiAverage messages in the database. Each WikiAverage message has a list of Votes stored, which can be seen in the database structure declaration at listing 8.

```
class WikiAverage : MongoMainEntry() {
  class Vote(
      var userId: String,
      var date: Date,
      var vote: Int
  ) : MongoSubEntry()

  // var _id: String defined in the MongoMainEntry super class
  var votes: List<Vote> = emptyList()
}
```

Listing 8: WikiAverage declaration

So if the votes need to be extracted, it results in a list of lists. More precisely: a list of WikiAverages where every WikiAverage holds a list of Votes. This data structure made it necessary to perform several cleaning and transformation steps in order to be able to extract the needed information.

4.5.1 Extracting relevant data from remote server database

Since the majority of WikiAverages stored in the remote server database were of no interest in this analysis and exporting all WikiAverages would have taken a lot of time and wasted database server compute power, the first step was to select a suiting time range. The A/B-Test was live from 2020.01.27 18:36 to 2020.02.18 11:00. On 2020.02.01 at 17:30 a push notification

was sent out, which asked the users to share their experiences in Chat Wiki. To let the users enough time to add their knowledge to Chat Wiki, a time period of 2 weeks was considered. Therefore a suitable time range would be from 2020.02.01 17:30 to 2020.02.15 17:30.

The second step was to export all WikiAverages from the remote server database, which held at least one Vote that was within the time range at which the A/B-Test was live, to a local database. The exportation of the data from the live database to a local testing database is not necessary but it prevents accidental data loss of the production database and provides a performance benefit while performing the data analysis. The result of the mongo database query, displayed in listing 9, was exported. This resulted

```
{ "votes.date":
    {
        $gt: ISODate("2020-01-28T00:00:00.000+0000"),
        $lt: ISODate("2020-02-18T08:00:00.000+0000")
    }
}
```

Listing 9: Mongo database query

in a total of 22,710 WikiAverages imported into the local mongo database.

4.5.2 Processing local data with Kotlin

To further ease the process of data transformation, the local mongo database was accessed via the chat server Kotlin-project. This allows the usage of functional programming and already existing helper functions. As a result, all the next steps were performed in the Kotlin programming language.

The next step was to ”flatten” the data structure to make it easier to handle the data. In order to do this, it was first required to determine the needed member-fields. By taking a look at listing 8 it is clear that the following three member-fields are needed: `userId`, `data`, and `_id`. To aid the readability of the future code the AnalysisVote helper class was created, which can be seen at listing 10.

```
data class AnalysisVote(
    val wikiAverageId: String,
    val date: Date,
    val userId: String
)
```

Listing 10: AnalysisVote declaration

Where `wikiAverageId` is the ID of the corresponding WikiAverage, and `date` and `userId` are the members of the corresponding Vote.

```
val partialVotes = getCollection<WikiAverage>()
    .find()
    .flatMap { wikiAverage ->
      wikiAverage.votes.mapNotNull { partialVote ->
        if (partialVote.date > "2020-02-1T17:30:00".toDate() &&
            partialVote.date < "2020-02-15T17:30:00".toDate()) {
          AnalysisVote(
              wikiAverageId = wikiAverage._id,
              date = partialVote.date,
              userId = partialVote.userId
          )
        } else {
          null
        }
      }
    }
```

Listing 11: Flatting the data structure

Listing 11 shows how all WikiAverages are obtained from the local database, which has been already pre-filtered as mentioned in Subsection 4.5.1. Then getting filtered by `date` to match the previously selected 2 Week period between 2020-02-1 17:30 and 2020-02-15 17:30 and finally stored into a list of AnalysisVotes called `partialVotes`. So the original data structure (a list of WikiAverages where each WikiAverage contains a list of Votes) was transformed/flattened into a list of AnalysisVotes. Note that each

Course Wiki has four WikiAverage entries, the reasons for this can be found in section 4.5. Thus a user can have up to four Vote entries per Course Wiki and therefore up to four AnalysisVote entries per Course Wiki in the `partialVotes` list.

Thanks to the simple design of the group separation discussed in section 4.4, it was little effort to verify that users are equally divided into groups A and B. The list of AnalysisVote created in listing 11 called `partialVotes`, was filtered by the parity of the integer representation of the hashed `userId`. This resulted in two lists namely `partialVotesOld` and `partialVotesNew` which contained all partial votes of the old method group and the new method group respectively. To obtain the user count of the two groups, the corresponding partialVote list was grouped by the `userId` and then counted, as it can be seen in listing 12.

```
val partialVotesOld = partialVotes
    .filter { partialVote ->
      partialVote.userId.hashCode() % 2 == 0
    }
val partialVotesNew = partialVotes
    .filter { partialVote ->
      partialVote.userId.hashCode() % 2 != 0
    }

val oldMethodUserCount = partialVotesOld
    .groupBy { it.userId }
    .count()
val newMethodUserCount = partialVotesNew
    .groupBy { it.userId }
    .count()
```

Listing 12: Calculating group sizes

To obtain which user voted on which Course Wiki, it was necessary to transform the `partialVotesOld` and `partialVotesNew` lists to a map type which has the `courseWikiId + userId` as key and the list of corresponding `partialVotes` as value. As listing 13 shows, this has been achieved

by grouping the lists by a string created from the `courseWikiId` and the `userId`. The `courseWikiId` was extracted from the `wikiAverageId`, which is possible because of the special design of the IDs in Studo Chat. Here the `wikiAverageId` consists of the `courseWikiId + "_" + wikiAverageType`, where `courseWikiId` again consists of several other parts containing two `"_"`. So by splitting the `wikiAverageId` by `"_"` and rejoining the first 3 elements yields the `courseWikiId`. This design decision allows on the server-side to obtain IDs without the need for database queries which holds major performance improvements in certain situations but also increases code complexity and reduces code readability. The results of the grouping shown in listing 13 are two maps called `wikiUserIdToPartialVotesOld` and `wikiUserIdToPartialVotesNew` which holding the `courseWikiId + "-" + userId` as key and the corresponding list of `partialVotes` as value.

```
val wikiUserIdToPartialVotesOld = partialVotesOld
    .groupBy { partialVote ->
      val courseWikiId = partialVote.wikiAverageId
          .split("_")
          .take(3)
          .joinToString("_")
      courseWikiId + "-" + partialVote.userId
    }
val wikiUserIdToPartialVotesNew = partialVotesNew
    .groupBy { partialVote ->
      val courseWikiId = partialVote.wikiAverageId
          .split("_")
          .take(3)
          .joinToString("_")
      courseWikiId + "-" + partialVote.userId
    }
val votesCountOld = wikiUserIdToPartialVotesOld.count()
val votesCountNew = wikiUserIdToPartialVotesNew.count()
```

Listing 13: Transforming list of AnalysisVote to map of courseWikiId+userId to list of AnalysisVote

Listing 14 shows how the count of users which voted on more than

one Course Wiki is calculated and stored in the two variables called `multiVoteUserCountOld` and `multiVoteUserCountOld`. At first all elements of `wikiUserIdToPartialVotesOld` whose `partialVote` list did not have 4 elements where filtered. This filtering step was performed because only completed votes were taken into consideration as they provide the most value. Then the keys of the filtered `wikiUserIdToPartialVotesOld` were grouped by the `userId`. The `userId` was extracted from the key by splitting the key by `"-"` and taking the second element. This was possible because of the way the key was created in listing 13. The grouping produced a new temporary map that has the `userId` as key and the old key as value, so it makes it easy to get the number of Course Wikis a specific user has voted on. The last filter statement removes all entries where a user voted on one Course Wiki only. This is done because if users enter the old Course Wiki Add View, the users will not notice that anything is wrong, just after the users are trying to finish the entry of the Course Wiki the users could get irritated by the non-conforming UX which might reduce the motivation of the users to add their knowledge to more Course Wikis.

```
val multiVoteUserCountOld = wikiUserIdToPartialVotesOld
    .filter { it.value.count() == 4 }
    .keys
    .groupBy { it.split("-")[1] }
    .filter { it.value.count() > 1 }
    .count()
val multiVoteUserCountNew = wikiUserIdToPartialVotesNew
    .filter { it.value.count() == 4 }
    .keys
    .groupBy { it.split("-")[1] }
    .filter { it.value.count() > 1 }
    .count()
```

Listing 14: Calculating how many users voted on more than one Course Wiki

To get the number of users who voted on more than one Course Wiki in the given A/B-Testing time frame and had voted on a Course Wiki before the A/B-Testing time frame, it was necessary to gather all WikiAverage entries

from the production database. To speed things up it was decided, instead of exporting from the production database to just download the lasted backup of the production database, from the Web Interface of the Service Provider used by Studo Chat, to then partially import the backup into the local database. Partially because just the WikiAverage objects were needed and the backup contained all objects of the production database. After the required data was successfully imported, it was possible to compute a list of all userIDs which have already voted on a Course Wiki before the A/B-Test. This list was called `usersWhichVotedBeforeABTest`. As listing 15 shows, `usersWhichVotedBeforeABTest` was generated by querying all WikiAverages from the local database and mapping the list of WikiAverages to a list of Votes, which then contained all partial Votes ever made on Studo Chat. Now all the partial votes could be filtered by date so that only the partial votes which were created before the start of the A/B-Test were preserved. The last two statements convert the list of partial votes into a set of userIds. Since `usersWhichVotedBeforeABTest` was created, the number of users who voted on more than one Course Wiki in the given A/B-Testing time frame and had voted on a Course Wiki before the A/B-Testing time frame could be calculated. Therefore basically the same statement of listing 14 was used again but with one addition. The additional filter statement at the end filters all userIds which did not vote on a Course Wiki before the A/B-Test time frame. The result was stored in the variables called `userCountWhichVotedBeforeOld` and `userCountWhichVotedBeforeNew`.

In order to be able to further analyze the results of the A/B-Test, it was considered to evaluate the percentages of first-time-vote users and not-first-time-vote users within the two groups A and B. Therefore the `userCountWhichVotedBeforeOld/New` was divided by `multiVoteUserCountOld/New` which gives the percentages of not-first-time-vote users of both groups in a percentage representation of 0 to 1. This was multiplied by 100 to get a percentage representation of 0 to 100, the resulting number was then stored into the variables `percOldUsersVotedBefore` and `percNewUsersVotedBefore`. The first-time-vote user percentages were calculated by simply subtracting the just calculated `percOldUsersVotedBefore` and `percNewUsersVotedBefore` from 100, the results were stored into `percOldUsersNotVotedBefore` and `percNewUsersNotVotedBefore` respectively. The kotlin code of these calculations can be seen in listing 16.

```
val usersWhichVotedBeforeABTest = getCollection<WikiAverage>()
    .find()
    .flatMap { it.votes }
    .filter { it.date < "2020-02-1T17:30:00".toDate() }
    .map { it.userId }
    .toSet()

val userCountWhichVotedBeforeOld = wikiUserIdToPartialVotesOld
    .filter { it.value.count() == 4 }
    .keys
    .groupBy { it.split("-")[1] }
    .filter { it.value.count() > 1 }
    .filter { usersWhichVotedBeforeABTest.contains(it.key) }
    .count()

val userCountWhichVotedBeforeNew = wikiUserIdToPartialVotesNew
    .filter { it.value.count() == 4 }
    .keys
    .groupBy { it.split("-")[1] }
    .filter { it.value.count() > 1 }
    .filter { usersWhichVotedBeforeABTest.contains(it.key) }
    .count()
```

Listing 15: Calculating how many users already voted before

```
val percOldUsersVotedBefore = userCountWhichVotedBeforeOld /
    multiVoteUserCountOld * 100
val percOldUsersNotVotedBefore = 100 - percOldUsersVotedBefore

val percNewUsersVotedBefore = userCountWhichVotedBeforeNew /
    multiVoteUserCountNew * 100
val percNewUsersNotVotedBefore = 100 - percNewUsersVotedBefore
```

Listing 16: Calculating percentages of first-time-vote users

5 Chat Search

Since the Studo Chat platform initially launched in 2017 it has come a long way, in both user base growth as well as feature set improvements. So scalability is always a key feature if it comes to Studo Chat or as Kappe (1995, page 85) says:

> "In the Internet environment, scalability is a very important aspect of system design, since the values of these variables are already high and are continuing to grow extremely fast."

Therefore one of the major challenges is to not only maintain the feature set that already exists for a continuously growing monthly- and daily-active-user base but further improve the feature set. One of these feature set improvements is called Chat Search. This chapter is about the reason for its existence, the implementation challenges as well as the question if the feature did change the user behaviour in any way.

5.1 Reason

In the three years Studo Chat is in production, not only the user base did grow significantly but also the interactions like new topics, messages, votes, etcetera. For example, the monthly active user figure did increase by 114.02% from 2017 to 2018 and again by 39.29% from 2018 to 2019. This growth did lead to several issues, one of them being that topics are far less interacted with again after they are not one of the newest topics in the channel anymore. The same behaviour can be observed at other places within the Studo app as well, the news section in the Studo app for example. This is most likely caused by the fact that items that can be interacted with, without the need to scroll are the ones which get the most interaction. The

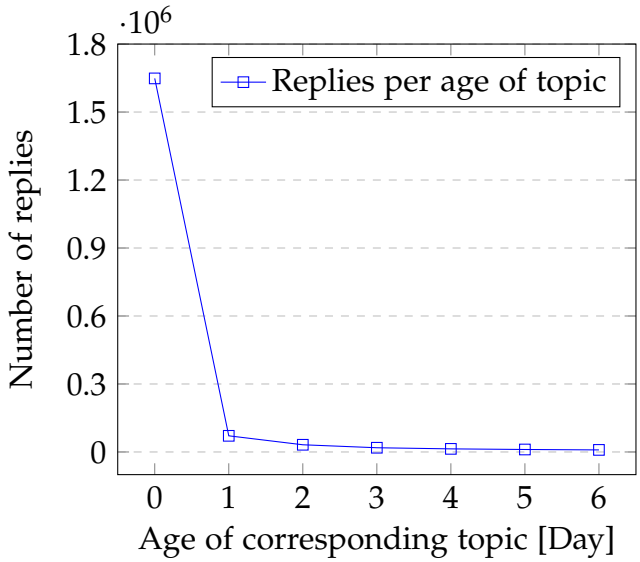

Figure 5.1: Topic replies per topic age in days of the year 2019

further a user has to scroll to get to the item, the less the likeliness that the user is going to interact with the item. Depending on the size and resolution of the screen of the device the user uses, of course, approximately four to eight topics are displayed at a time. Therefore the first four to eight topics are visited and interacted with the most. An example of the decrease in interaction on topics can be seen in figure 5.1. Here it can be seen that on day 0, meaning the same day as the topic was created, by far the most replies and therefore interactions happen. One day after the topic was created, day 1 at the chart, only has on average only $\sim$ 4% of the replies that the first day, day 0 at the chart, had.

This leads to two main problems:

- Knowledge loss
- Redundant information / information fragmentation

The problem of knowledge loss is especially problematic in Course Channels. These channels often contain answers to questions that recur every semester because the subject matter of a class usually does not change every semester or in some cases even decades. But depending on the activity in the Course

Channel it is possible that a good answer does not even stay at the magic top ten for a week and is therefore ”lost” forever because hardly any user is willing to scroll through hundreds of topics to eventually find an answer to a question. It is particularly problematic if users are willing to sacrifice a considerable amount of time to create a useful document like a substance summary or elaboration just to find out that no one can find it anymore after a week.

Another problem is redundant information, this applies to Region Channels as well as to Course Channels. Due to the problem with ”lost” topics, there are many duplicate questions, which not only leads to redundant information but to a fragmentation of knowledge as well. For example, if there are multiple question topics asking about the best Döner-Kebap-Restaurant in Graz, there might be answers suggesting Restaurant-A in each of those questions topics, causing redundant information. But there might also be an answer suggesting Restaurant-B which is present in one question topic only, leading to a fragmentation of information which makes it even harder for users to find the information needed.

These problems made it necessary to implement a way for users to find the information they need in an easy way and therefore to counter the knowledge loss, redundant information, and information fragmentation.

5.2 Implementation

To counter the problems discussed in section 5.1 it was decided to implement a search feature to provide users the ability to search for questions, answers, and documents within a channel. The user can access the SearchView 5.2 via the search icon which can be found in the NavigationBar of each channel that supports the search feature. An example of this can be found in figure 3.2. The decision about the way users can access the search feature was based on two main criteria, namely server control and UI equality of both client-side platforms. The NavigationBar Buttons provide different actions like notification toggle, share, and search. These functionalities are controlled by the server which allows to enable and disable the search feature for all or just one channel at any given time, which fulfills the first criterion. This is

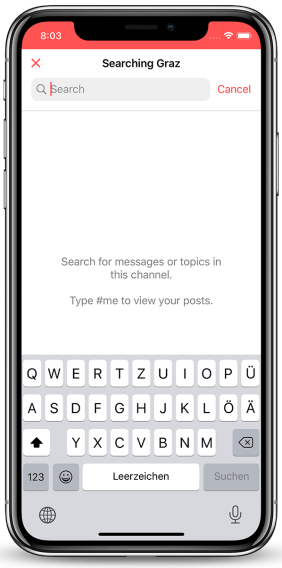

Figure 5.2: Searchview

not only useful at the launch of the feature but at the launch of every update of the search feature. So in case, an update causes an error or even a crash, the server can simply disable the feature for all clients. But this is not the only use case, since it is possible to disable search for just a single channel, it can also be used to counter a load spike at a specific channel. So if for example there is an event in Graz or related to Graz which causes a lot of chat activity within the Graz channel, it is possible to save up some server resources by disabling the search feature. The standard UI implementations of search in Android and iOS are quite different, by using a simple button in the NavigationBar the UI of the Android and iOS implementation are the same, which fulfills the second criterion.

As can be seen in figure 5.2 the UI of the SearchView is kept simple. The NavigationBar holds a close button to close the SearchView, as well as a title that notifies the user about which channel is currently searched, in this example the ”Graz” channel is searched. Directly underneath the NavigationBar, is the SearchBar located which provides the TextField in which the users type their search terms. Between SearchBar and Keyboard sits the ConentView, it shows all results of the search terms. In case no search

Figure 5.3: Searchview with results

term is entered yet, the ContentView shows a help text as the example figure 5.2 shows. If a search term in which a user enters into the SearchBar does not return any results, a ”No results found” text is displayed instead of the help text.

Figure 5.3 shows the SearchView with a number of results. Each result is a message which contains a match. Their UI is almost the same as if they were within a MessagesView but with the matching search term highlighted with a bold font style. The highlighting is possible because of the `ClientMessage.htmlText` member field discussed in section 3.3. If a user taps on one of the results, a navigation action to the corresponding topic is triggered, so the user can interact as usual with the topic and its messages.

The software implementation of the search feature is in theory quite basic, the user types in some search terms, the client sends a request to the server and the server responds with a number of results. But as always, the challenges lie in the details. A number of design decisions had to be taken, the first one being about the network protocol. Usually, a Representational state transfer (REST)-Application programming interface (API) over HTTPS would be a good choice for a use case like this. But as already discussed

in section 4.1 Studo Chat transmits data via WebSocket protocol, so it was decided to use the same protocol for the search feature. This allowed reducing the complexity on the server-side as well as the reuse of a lot of code on the server- and client-side. One example of this reuse of code is that the results send from the server are just a list of the matching messages, so the SearchView implementation code can be partly shared with the code of MessagesView.

Another design decision was about the UX of searching. The goal was to create an experience that feels modern, fast, and is easy to use. So the results should appear as the user types the search terms into the TextField, without the need of pressing an additional button to send the search request to the server.

These decisions led to two problems which were combined even more problematic. Firstly, to provide the planned UX the client needs to send a request to the server for every character the user typed in. This means for a simple search like "döner", the client would need to send five requests, one for every character. These can lead to severe server load issues since in the current state of the server a search request can take seconds to process. So a concept has to be made to reduce the number of requests and therefore reduce the server load and increase the scalability.

Secondly, the WebSocket protocol provides a bidirectional data transfer but without any mechanism to refer each response to each request. This is problematic because the time it takes the server to respond to a request can vary widely, from milliseconds to seconds. Consequently, a custom solution that allows referring each request to each response must be implemented. Otherwise, it can occur that the last request is handled faster than the second last request, meaning that on the client-side the last response is replaced by the second last response. This behaviour is visually represented by the sequence diagram shown in figure 5.4. Therefore the wrong results would be displayed to the user which would be a catastrophic UX.

As already mentioned, each server response to a client search request consists of a number of messages. These messages differ in a few characteristics from their "original" ones. One difference is the `topicId`, each search result message has the same static `topicId = search_topic`. This allows the SearchView to use an easy query to present all search result messages, as

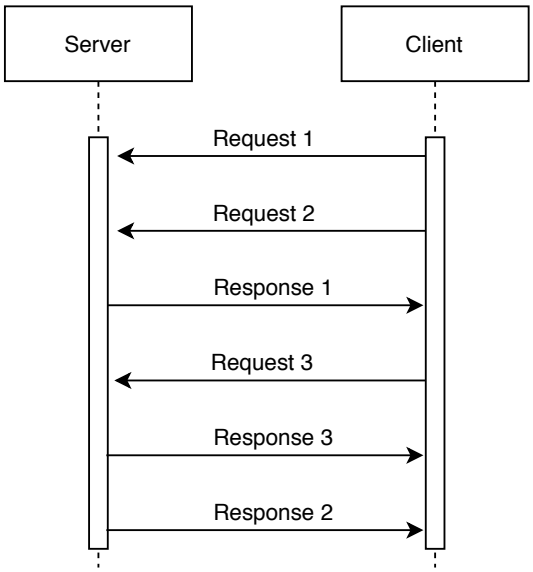

Figure 5.4: Sequence diagram of problematic chat search network communication

it can be seen at listing 17. Furthermore can be seen that the `sortScore` is here used as well, so the server can tell the client which results are the most important and should be displayed on top.

```
let allSearchResults = realm.objects(ClientMessage.self)
      .filter("topicId = %@", "searchTopic")
      .sorted(byKeyPath: "sortScore", ascending: false)
```

Listing 17: Search result messages realm query

To counter the problem where the client would receive the results in the wrong order and therefore display the wrong results to the users, the following concept was implemented. In contrast to all other topics where the server not only sends the new messages to the client but also tells the client which messages should be deleted, the search-topic on client-side never receives a delete request from the server. The server only sends the requested search result messages and like all other messages, the search result messages are stored to realm as well. To display the search results, the client queries all search result messages as discussed before, and shown

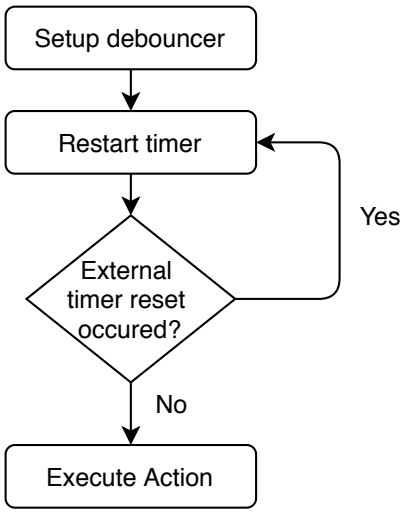

Figure 5.5: Flowchart of a debouncer

in listing 17. But to only show the results of the current search term the user requested, a new member field was introduced to the `ClientMessage` class, called `searchTerm`. With the `searchTerm` member field, the server can tell the client which search result message belongs to which search request. Therefore it does not matter how long each request takes, only the results of the current search term the user typed into the SearchBar are displayed. To counter a growing client database as well as presenting the user "old" search results, because a user may search for the same term a month later but the search results may change quite significantly after a month, the SearchView always deletes all search result messages as a new instance gets created.

Since every character the user types into the SearchBar triggers a search request for the server, even a simple search term produces a number of network requests. Namely, exactly as many network requests as the search term has characters. To reduce the server load but still preserve a good UX for the users while searching it was decided to implement a debouncer. A debouncer has a certain action to execute but does not execute the action right away. The debouncer has a timer that triggers the execution after a specified amount of time. But the clue is that the timer can be restarted and therefore the execution of the action is delayed. For a better understanding,

the visual representation of this can be found in figure 5.5. So the action the debouncer should execute is the search request with the current search term and the timeout of the timer is set to 300ms. Every time the user changes the search term by adding or removing a character, the timer of the debouncer gets restarted. So as long as the users do not need more than 300ms between their virtual keystrokes, only one search request is sent to the server.

On the server-side, it was decided to keep the implementation simple because it was not known how well the user base would accept the feature. Therefore the text search features of mongoDB were used for a first implementation, although they are limited in functionality in comparison to other search engines, like Elasticsearch (Elasticsearch B.V., 2020, page 1) to name an example.

But by using the text search feature of mongoDB it was not necessary to implement another search engine and to keep the data sync between the search engine and mongoDB, which reduces implementation cost and overall complexity of the feature.

5.3 Preparing data for evaluation

To evaluate if Chat Search did impact the behaviour of the user base, interaction data with a corresponding timestamp was needed, so data before and after the Chat Search release could be compared. The only data set that met the requirements which was also big/old enough to provide meaningful data was the messages collection. So the idea was to take all messages which were posted between the first of March and the first of August of 2019 and 2020 respectively. Each message has a timestamp which provides the creation date of the message. This timestamp is then compared to the creation timestamp of the corresponding topic. With this data, a chart can be created which tells how many messages are posted related to the age of the topic in which these messages were posted.

All code examples were written in Kotlin and were applied to gather the message data from 2019 as well as 2020. As it can be seen in listing 18, all messages from type ”Course Discussion” which were created between

March the first and August the first were queried and then grouped into a map which has the `topicId` as key and the list of corresponding messages as value. As the last step, all topics with no replies were filtered.

```
val topicIdsToMessages = getCollection<Message>()
    .find(Message::_id startsWith "course_discussion_",
        Message::date >= "2019-05-01T00:00:00".toDate()!!,
        Message::date < "2019-08-01T00:00:00".toDate()!!
    )
    .groupBy { it.topicId }
    .filter { it.value.count() > 1 }
```

Listing 18: Query and group messages from mongoDB

Listing 19 shows how the number of replies related to the age of the topic was calculated. It was iterated over each entry of the previously defined `topicIdsToMessages`. In each iteration, the `topic` and the corresponding `firstMessage` were queried from mongoDB to get the creation date of the topic. Before the list of messages was iterated, it was filtered by the `firstMessage` because the first message of a topic is the message representation of the topic and therefore is not a reply. For every reply message, the time difference in days was calculated and the corresponding entry of the `tAgeToReplies` map was incremented by one. The resulting map called `tAgeToReplies` has the age of the topic in days and the number of replies as value. So for example the expression `tAgeToReplies[0]` would return the number of replies which were created on the same day as the corresponding topic.

To calculate the replies per day in percent it was necessary to sum up all messages first and store the result into `numberOfMessages` afterwards, as it can be seen in listing 20. Then all values of `tAgeToReplies` were divided by `numberOfMessages` and the results were stored into `tAgeToPercentReplies`.

As it was considered useful to accumulate the data per month instead of per day the same calculation took place but instead of calculating and using `diffInDays` from listing 19, `diffInMonths` is calculated and used as it can be seen in listing 21.

```
val tAgeToReplies = mutableMapOf<Int, Int>()
topicIdsToMessages.forEach { (topicId, messages) ->
  val topic = getCollection<Topic>()
      .findOne(Topic::_id equal topicId)!!
  val firstMessage = getCollection<Message>()
      .findOne(Message::_id equal topic.firstMessageId)!!
  messages
      .filter { it._id != firstMessage._id }
      .forEach { message ->
        val diffInMillies = Math
            .abs(message.date.time - firstMessage.date.time)
        val diffInDays = TimeUnit.DAYS
            .convert(diffInMillies, TimeUnit.MILLISECONDS)
            .toInt()
        tAgeToReplies[diffInDays] = tAgeToReplies[diffInDays]+1
      }
}
```

Listing 19: Calculate replies per topic age in days

In order to be able to better visualize the reply history of up to 18 months old topics it was decided to calculate the log of the percentile reply counts as it can be seen in listing 22. But due to the small nature of the percentile reply counts all results were negative. Furthermore, the *tAgeToPercentReplies* contains several zeros which would result in not-a-number or negative infinity.

To counter these mathematical effects so the data could be better visualized the log calculation was slightly adapted, as it can be seen in listing 23. Before the log10 was taken from the percentages, 0.00001 was added to counter the "Not a Number" results produced by taking the log of 0. To let the y-axis of the chart range from 0 to +5, the result of the log was added with 5.

```
val tAgeToPercentReplies = mutableMapOf<Int, Int>()
val numberOfMessages = tAgeToReplies.values.sum()
tAgeToReplies.forEach { (age, replyCount) ->
  tAgeToPercentReplies[age] = replyCount / numberOfMessages
}
```

Listing 20: Calculate reply percentages per topic age

```
val diffInMonths = TimeUnit.DAYS
    .convert(diffInMillies, TimeUnit.MILLISECONDS)
    .toInt() / 30
```

Listing 21: Calculate reply percentages per topic age in months

```
tAgeToReplies.forEach { (age, replyCount) ->
  tAgeToPercentLogReplies[age] =
      log10(tAgeToPercentReplies[age]!!)
}
```

Listing 22: Calculate reply log percentages per topic age in months

```
tAgeToReplies.forEach { (age, replyCount) ->
  tAgeToPercentLogAdaptReplies[age] =
      log10(tAgeToPercentReplies[age]!! + 0.00001) + 5
}
```

Listing 23: Calculate reply adapted log percentages per topic age in months

6 Results

This chapter discusses the results of the two improvement attempts, namely Chat Wiki navigation redesign and the new Chat Search feature. It shows the numbers and provides an interpretation of their meaning. Additionally, the results were checked for their statistical relevance.

6.1 Wiki navigation redesign

This section discusses the results of the A/B-Test of Wiki navigation redesign. All variables used in table 6.1 were previously defined in section 4.5, which also explains in detail how each number was calculated. The time frame of the A/B-Test was between 2020.02.01 17:30 and 2020.02.15 17:30.

Variable Name	Group A (Old)	Group B (New)
partialVotes	34,132	
partialVotesOld/New	16,724	17,408
old/newMethodUserCount	1,688	1,689
votesCountOld/New	4,883	4,966
multiVoteUserCountOld/New	685	738

Table 6.1: Results of A/B-Test

The variable `partialVotes` represents the total number of partial votes all users have placed within the A/B-Test time frame. A single user can place up to 4 votes at a single Course Wiki, consequently, these votes are called partial votes. More details on this can be found in subsection 4.5.2. The variables `partialVotesOld` and `partialVotesNew` holding the numbers of partial votes of the corresponding test group. `oldMethodUserCount` and

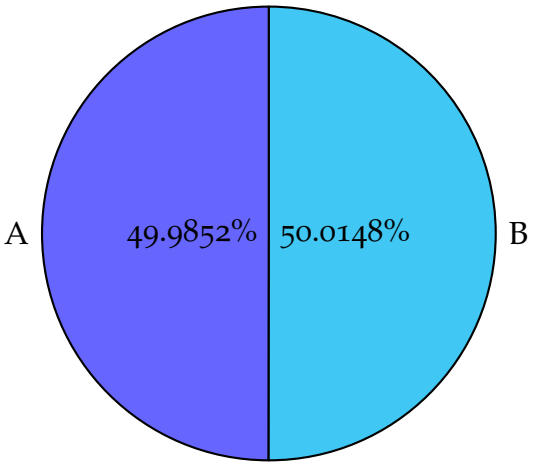

Figure 6.1: Distribution of groups A/B

`newMethodUserCount` representing the number of users in the corresponding group. `votesCountOld` and `votesCountNew` showing the sum of the number of Course Wikis each user has voted on, meaning a user has placed at least one partial vote on a Course Wiki. `multiVoteUserCountOld` and `multiVoteUserCountNew` are showing the numbers of users who did vote on more than one topic with exactly four partial votes.

As visualized by figure 6.1, the separation of the users in the two groups A and B did work well since there is a small difference of one user and a percentile difference of 0.0148%.

`multiVoteUserCountOld/New` shows that on group B, 7.74% more users did vote on more than one Course Wiki. To validate the statistical significance of the result, the p-value was calculated via the chi-square function from the scipy python library (The SciPy community, 2020a, page 1).

```
chisq, p-value = chisquare(
    f_obs=[685, 738],
    f_exp=[711.5, 711.5]
)
```

Listing 24: Calculating p-value with chi-square test

Listing 24 displays how the chi-square calculation was performed. `f_obs` are the observed values, in this case, the values of `multiVoteUserCountOld/New`. `f_exp` are the expected values, in this case, it was expected that the groups A and B had the same values, so it was expected that group A and B would perform exactly the same. 711.5 is the result of $\frac{685+738}{2}$. As a result the `chisquare`-function returns the chi-squared test statistic, called `chisq` and the `p-value`.

chisq	p-value
1.9739985945186227	0.16002407260447196

Table 6.2: Results of chi-square

As can be seen in table 6.2, the p-value is 16% which is significantly higher than the general accepted $\alpha = 5\%$. Therefore can the null hypothesis not be rejected. As a consequence, no clear statement can be made because either the new implementation method (group B) is no better than the old implementation method (group A) or group B is better than group A but only ever so slightly.

However, it should be noted that further investigations revealed an imbalance of first-time-vote users and non-first-time-vote users in groups A and B. As can be seen in figure 6.2, first-time-vote users and non-first-time-vote users are not evenly distributed.

When studying figure 6.2, it is clear that the majority of the users of group A had already voted on a Course Wiki before the A/B-Test time frame. Therefore over 60% of group A's users were already familiar with the old implementation of the Wiki Addview and might already have been "used" to the design and behaviour of the old implementation of Wiki Addview. This could suggest that these users influenced the results of the A/B-Test in favour of group A since it does not matter if a user has voted before in group B but it could matter in group A.

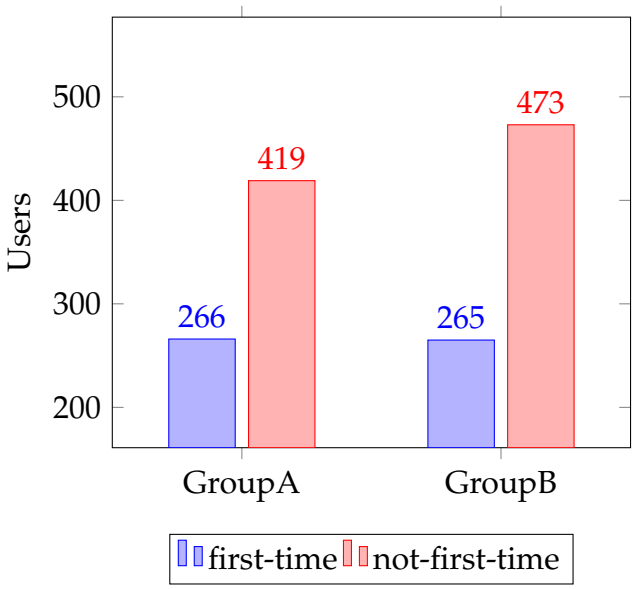

Figure 6.2: Comparison of first-time to non-first-time voting users in groups A/B

6.2 Chat Search

In this section, the results of the data previously prepared in section 5.3 will be evaluated and discussed. The first chart 6.3 shows the number of replies a topic has received on each day from its creation day to a week afterward. Since the user base of Studo as well as Studo Chat did grow from 2019 to 2020, an increase of interactions, in this case, new messages, especially at day 0 is clearly visible. The chart also reveals that a topic gets by far the most replies on the same day it was created.

To better compare the data of 2019 and 2020, chart 6.4 shows the same data in percentages, the exact calculation can be found in section 5.3 at listing 20. The chart shows that in both 2019 and 2020 each topic received about 90% of its replies on the first day. Furthermore, the chart shows that the reply behaviour in 2020 did not change in a significant way in the first week, compared to 2019.

Due to the large difference in the size of the data, it was necessary to take a log in order to be able to visualize one and a half years in one chart, as it can

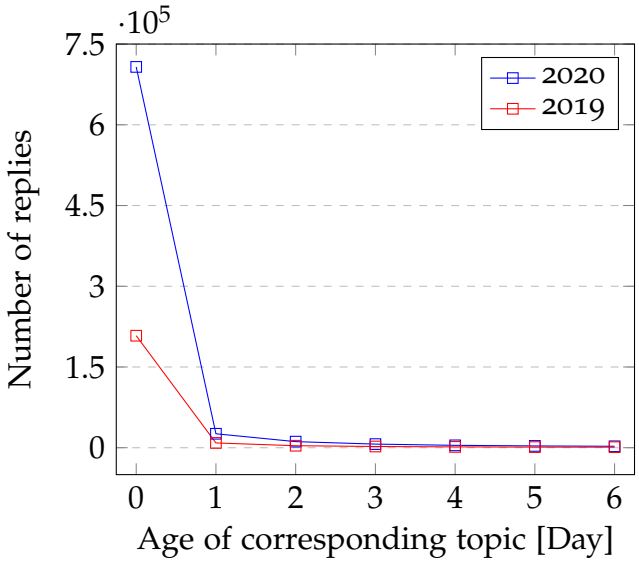

Figure 6.3: Comparison of reply counts per day of first week

be seen in listing 22. Unfortunately, the data contains several zeros which resulted in not-a-number or negative infinity results, they were replaced with 0 to be able to represent them in a chart. The result of these calculations can be found in figure 6.5. It can already be seen that the first month was very similar both in 2019 and 2020, but interestingly between the second and the fourth month (1 to 3 in the chart), 2019 seems to have received more replies. But in the fifth month, the reply ration is again very similar and 2020 appears to have a higher ratio beginning with the sixth month but to better visualize the data, some adaptions had to be made.

The goal was to ”prettify” the data so it could be interpreted more easily. The exact calculations and reasoning can be found in section 5.3 at listing 23, the visual result can be found in figure 6.6. The chart leads to the hypothesis that there might be significant differences in the data between the seventh and the eighteenth month of 2019 and 2020.

To confirm the hypothesis, a closer look was taken at the time range of interest, which can be seen in figure 6.7. Except for months 7 and 14, all 2020 reply percentages are above or equal the ones of 2019. This could indicate that Chat Search leads to a higher reply activity to older topics. Older topics

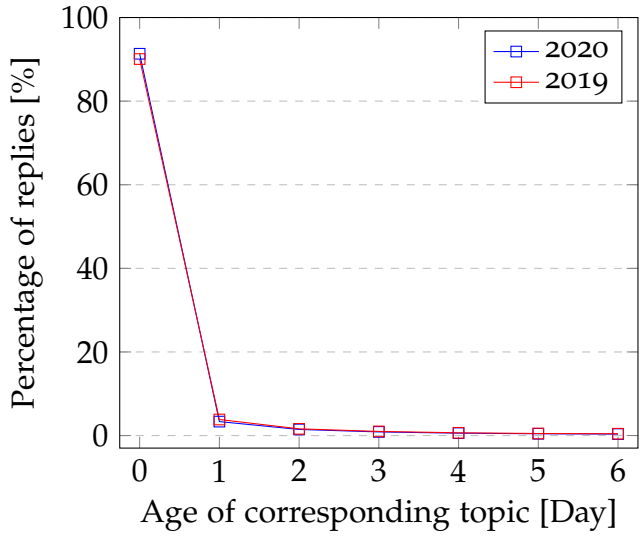

Figure 6.4: Comparison of reply percentages per day of first week

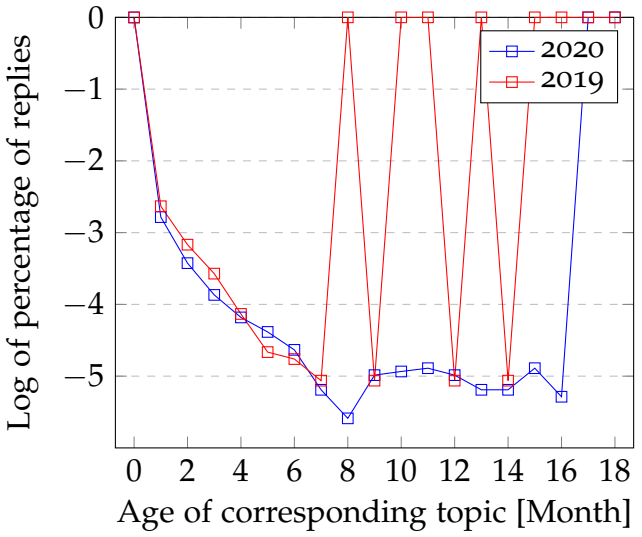

Figure 6.5: Comparison of reply log percentages per month of 18 months

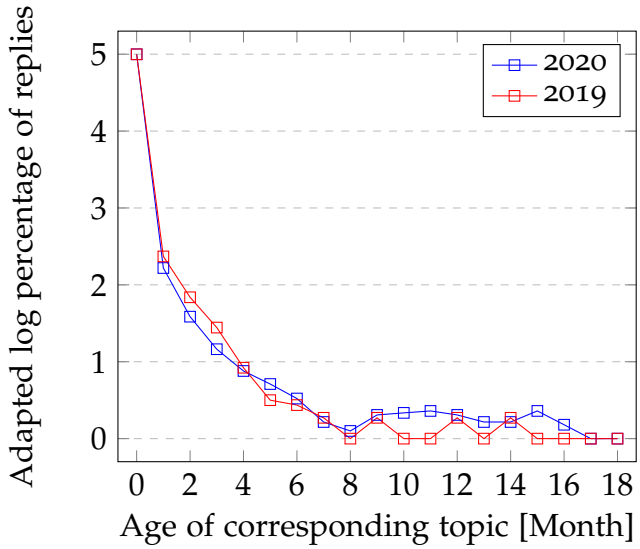

Figure 6.6: Comparison of adapted reply log percentages per month of 18 months

being in this case between 6 and 18 months old.

To prove that the difference between 2019 and 2020 is significant, the t-test from the scipy python library (The SciPy community, 2020b, page 1) was used. The python code of this function call can be found at listing 25. The null hypothesis of this test states that the means of the two data sets are equal. The results of the t-test can be found in table 6.3.

```
statistic, p-value = stats.ttest_ind(
    a=data19,
    b=data20
)
```

Listing 25: Calculating p-value with t-test

As table 6.3 shows, is the p-value equals 2.987% and is, therefore, smaller as the general accepted $\alpha = 5\%$. Consequently, the null hypothesis can be rejected and accept the alternative hypothesis which states that the means of the two data sets are not equal. Therefore is the increase of replies of topics whose age is between 7 and 18 months in the year 2020 compared to

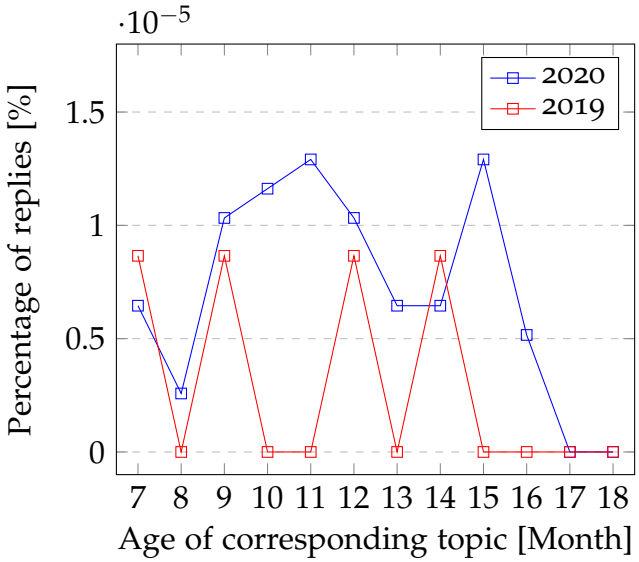

Figure 6.7: Comparison of reply percentages per month between 7th and 18th month

the year 2019 significant. This leads to the conclusion that Chat Search did impact the behaviour of the users and older topics are getting now more interactions than before.

statistic	p-value
-2.322234238817402	0.029868210927147056

Table 6.3: Results of t-test

6.3 Conclusion and Future Work

The goal of the thesis was to improve the Chat to be able to support students even better. To achieve this goal two different approaches were chosen, one was to improve the already existing Wiki, called Wiki navigation redesign, and the other was to implement a new feature, called Chat Search. Although the results of the Wiki navigation redesign were not conclusive, they showed that the attempt to improve the Wiki did not make it worse, maybe even slightly better. The results of Chat Search clearly showed an improvement in the usage of the Chat which leads to the conclusion that the goal of the thesis to improve the Chat was achieved overall.

It is always tricky to improve an application that already has a rich feature set. There is always the danger that a piece of software has so many features that it can be used for everything and nothing at the same time. To avoid this it is necessary to always keep a clear direction in mind at which the application should head. For Studo Chat this direction is very clear: the chat aims to improve the communication and the sharing of knowledge between students. One possibility to improve communication in the future could be the implementation of private group chats. They could be like the already existing private channels but with more than the current maximum of two participants. This way, students could easily create a group chat to for example handle all needed communication for group work, without the struggle to find a common communication service. Another future improvement regarding Chat Wiki could be to add additional space for attachments/files. This way students could more easily share and find important files like lecture summaries and elaborations. And there are many more ideas and possibilities but all have one thing in common: the objective of helping students achieve their goals and successfully graduate from their studies.

Appendix

Index of Abbreviations

Bibliography

Elasticsearch B.V. (Sept. 16, 2020). *Open Source Search: The Creators of Elasticsearch, ELK Stack & Kibana — Elastic*. URL: https://www.elastic.co/ (visited on 09/16/2020) (cit. on p. 61).

Felt, Adrienne Porter et al. (Aug. 2017). "Measuring HTTPS Adoption on the Web." In: *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC). USENIX Association, pp. 1323–1338. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/felt (cit. on p. 7).

google.com (Nov. 12, 2009). *Spdy: An Experimental Protocol For a Faster Web*. URL: https://sites.google.com/a/chromium.org/dev/spdy/spdy-whitepaper (visited on 03/24/2020) (cit. on p. 8).

Herwig, Volker, René Fischer, and Peter Braun (Sept. 24, 2015). "Assessment of REST and WebSocket in regards to their energy consumption for mobile applications." In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. IEEE Internet Computing, pp. 342–347. ISBN: 978-1-4673-8361-5. DOI: 10.1109/IDAACS.2015.7340755 (cit. on p. 12).

International Organization for Standardization, (ISO) (Nov. 15, 1994). *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip (visited on 03/24/2020) (cit. on pp. 3, 5).

Internet Engineering Task Force, (IETF), I. Fette, et al. (Dec. 2011). *The WebSocket Protocol*. URL: https://www.rfc-editor.org/rfc/rfc6455.html (visited on 03/24/2020) (cit. on pp. 6, 9, 10).

Internet Engineering Task Force, (IETF), Ericsson S. Loreto, et al. (Apr. 2011). *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. URL: https://www.rfc-editor.org/rfc/rfc6202.html (visited on 03/24/2020) (cit. on p. 6).

Kappe, Frank (1995). "A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems." In: J. UCS, pp. 84–104. DOI: 10.1007/978-3-642-80350-5_8 (cit. on p. 53).

Kumar, Santosh and Sonam Rai (May 2012). "Survey on Transport Layer Protocols: TCP & UDP." In: vol. 46. International Journal of Computer Applications, pp. 20–25. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.734.7346&rep=rep1&type=pdf (cit. on pp. 4, 5).

Liu, Qigang and Xiangyang Sun (Dec. 2012). "Research of Web Real-Time Communication Based on Web Socket." In: International Journal of Communications, Network and System Sciences, pp. 797–801. DOI: 10.4236/ijcns.2012.512083 (cit. on p. 10).

Marion, Charles and Julien Jomier (Aug. 2012). "Real-Time Collaborative Scientific WebGL Visualization with WebSocket." In: *Proceedings of the 17th International Conference on 3D Web Technology*. Association for Computing Machinery, pp. 47–50. ISBN: 9781450314329. DOI: 10.1145/2338714.2338721 (cit. on p. 11).

Network Working Group, (NWG), R. Fielding, et al. (June 1999). *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://www.rfc-editor.org/rfc/rfc2616.html (visited on 03/24/2020) (cit. on pp. 5, 6).

Network Working Group, (NWG), (IETF) Internet Engineering Task Force, and Editor R. Braden (Oct. 1989). *Requirements for Internet Hosts – Communication Layers*. RFC Editor. URL: https://www.rfc-editor.org/rfc/rfc1122.html (visited on 03/25/2020) (cit. on p. 5).

Pimentel, V. and B. G. Nickerson (July 2012). "Communicating and Displaying Real-Time Data with WebSocket." In: IEEE Internet Computing, pp. 45–53. DOI: 10.1109/MIC.2012.64 (cit. on pp. 6, 10).

Postel, Jon (Aug. 28, 1980). *User Datagram Protocol*. RFC Editor. URL: https://www.rfc-editor.org/rfc/rfc768.html (visited on 03/25/2020) (cit. on p. 4).

Postel, Jon (Sept. 1981). *TRANSMISSION CONTROL PROTOCOL*. RFC Editor. URL: https://www.rfc-editor.org/rfc/rfc793.html (visited on 03/25/2020) (cit. on p. 5).

Puranik, Darshan G., Dennis C. Feiock, and James H. Hill (Apr. 22, 2013). "Real-Time Monitoring using AJAX and WebSockets." In: *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. IEEE Internet Computing, pp. 110–118. ISBN: 978-0-7695-4991-0. DOI: 10.1109/ECBS.2013.10. URL: https://citeseerx.ist.psu.

edu/viewdoc/download?doi=10.1.1.830.3793&rep=rep1&type=pdf (cit. on p. 11).

Serpanos, Dimitrios and Tilman Wolf (Jan. 2011). *Architecture of Network Systems*. 1st ed. Morgan Kaufmann Publishers Inc. ISBN: 9780080922829 (cit. on pp. 3–5).

The SciPy community, (TSC) (July 4, 2020a). *scipy.stats.chisquare*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chisquare.html (visited on 07/10/2020) (cit. on p. 66).

The SciPy community, (TSC) (Sept. 9, 2020b). *scipy.stats.ttest_ind*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html (visited on 09/09/2020) (cit. on p. 71).

World Wide Web Consortium, (W3C) (1991). *The Original HTTP as defined in 1991*. URL: https://www.w3.org/Protocols/HTTP/AsImplemented.html (visited on 03/25/2020) (cit. on p. 5).