JOHANNES KEPLER
UNIVERSITY LINZ

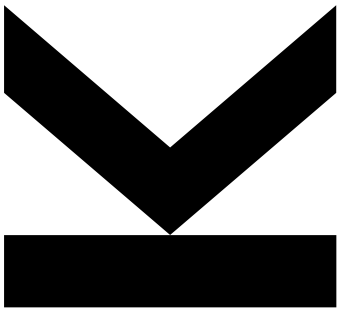
Author
Tobias Herber

Submission
Institute for System Software

Thesis Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc.

July 2023

Extending the Online Exam System Xaminer with Streaming Capabilities



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Extending the Online Exam System Xaminer with Streaming Capabilities

Dipl.-Ing. Dr. Markus Weninger, BSc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Student: Tobias Herber
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: December 2022

The exam system Xaminer is used by the Institute for System Software and other institutes to provide online exams for students. To improve the experience for both, lecturers and students, the system is constantly being improved and extended. Especially the need for a supplementary streaming service to provide video and audio connection between the lecturers and the students narrows the user experience.
Thus, the goal of this thesis is to develop a streaming service built into Xaminer. For this, the student has to perform the following steps:

- Evaluate:
  - the suitability of WebRTC (https://webrtc.org/) to transport the lecturer's video and audio (webcam + microphone) to students, as well as transporting up to 30 video and audio streams (one per student, webcam or desktop capture + microphone) to the lecturer. If WebRTC proves to be a non-optimal choice for this kind of task, the student should perform research on how to achieve this streaming goal.
    - If the chosen technology makes it necessary to implement server-side features (such as a websocket server), the goal is to achieve up to 300 concurrent streams (i.e., up to 10 lecturers with 30 students each). The thesis should contain at least theoretical thoughts on whether this goal is realistic.
    - As another subgoal, the student has to assess whether it is possible to force the students to share all screens, not only one specific screen if multiple screens are connected.
  - whether it is possible for a lecturer to display all streams at once, and if not, how many streams can be displayed at once and how easy these streams can be switched (for example, displaying 10 student streams at a time, and by clicking a button switching to the next 10 students).
    - This evaluation can be performed in a standalone prototype application – this application does not have to provide high user experience but focuses on assessing the technical feasibility.
- Integrate the streaming functionality into Xaminer. Currently, Xaminer offers lecturers a list of all connected students that states whether they already finished the exam or not. On this view, it would be desirable to display each student's video stream next to the student's list entry. A feature to display a given student's video in full screen has to exist.

Modalities:
The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.05.2023.

Abstract

More and more university exams are held online. This introduces the need for new tools. In addition to the examination tool itself, the students must be supervised using video calling software. While there are many good video calling tools on the market, most of them focus on corporate meetings. The aim of the project is to build a tool specifically for supervising students during online exams, while greatly reducing the overhead for students and instructors. This tool can be integrated into existing examination tools by building it as a web application using embeddable UI widgets.

The aim of our project is not only to implement the basic video calling functionality, but also to enrich it with features students and administrators need during an online exam. To make exams more pleasant for students, we will provide them with an easy interface to ask questions during the exam without disturbing other students. Additionally, we will make the ID check sessions in the beginning of the exam simpler for both the supervisors and the students; thereby eliminating the need to mark students on a printed out participant list.

We provide administration tools for supervisors to monitor the students during an exam in an intuitive way. Moreover, we perform logging to make it possible to check for inconsistencies and cheating after the exam.

Kurzfassung

Immer mehr Universitätsprüfungen finden online statt. Das erfordert die Entwicklung neuer Tools. Neben dem Prüfungstool selbst müssen die Studierenden mithilfe von Video-Calling-Software beaufsichtigt werden. Obwohl es viele gute Video-Calling-Tools auf dem Markt gibt, konzentrieren sich die meisten von ihnen auf Unternehmensmeetings. Ziel der Arbeit ist es, ein Tool speziell für die Überwachung von Studierenden während einer Online-Prüfung zu entwickeln, um die Durchführung solcher Prüfungen sowohl für Studierende als auch für Aufsichtspersonen zu vereinfachen. Dieses Tool kann in bestehende Online-Prüfungstools integriert werden, weil es als Webanwendung mit einbettbaren UI-Widgets implementiert ist.

Das Ziel unseres Projekts ist es nicht nur die grundlegende Video-Calling-Funktionalität zu implementieren, sondern sie auch mit den Funktionen auszustatten, die Studierende und Aufsichtspersonen während einer Online-Prüfung benötigen. Um Prüfungen angenehmer für Studierende zu gestalten, wird ihnen eine einfache grafische Oberfläche zur Verfügung gestellt, um während der Prüfung Fragen zu stellen, ohne andere Studierende zu stören. Darüber hinaus wird die Identitätsprüfungen zu Beginn der Prüfung sowohl für die Aufsichtspersonen als auch die Studierenden vereinfacht, wodurch die Notwendigkeit einer ausgedruckten Namensliste entfällt.

Des Weiteren werden Aufsichtspersonen mit Verwaltungstools ausgestattet, um die Studierenden während der Prüfung zu überwachen. Außerdem werden wir Logging durchführen, damit nach der Prüfung Unstimmigkeiten und Prüfungsbetrug überprüft werden können.

Table of Content

Contents

Abstract **i**

Kurzfassung **i**

1 Introduction **2**

2 Background **4**
2.1 Terminology 4
2.2 Video Calls in the Browser 5
2.3 Tech Stack 7

3 Overview **12**
3.1 Client Types 12
3.2 Communication Protocols 14
3.3 Communication Architectures 16
3.4 LiveKit 20
3.5 API and Video Streaming 21

4 Server **22**
4.1 Authentication 23
4.2 REST API 24
4.3 WebSocket Connections and State 25
4.4 Video Calling 27
4.5 Testing 28
4.6 Deployment 29

5 Client **30**
5.1 State 30
5.2 Components and Widgets 30
5.3 Video Calling 34
5.4 Testing 35
5.5 Demo Frontend 37
5.6 Integrating with Xaminer 37

6 Usage and Evaluation **40**
6.1 Students 40
6.2 Supervisors 41

7 Related Work **46**

8 Conclusion and Future Work **48**

Literature **50**

1 Introduction

More and more coding exams are held online to allow students to use tools representative of real programming. Since traditional online-learning systems are not well suited for these exams, the Institure for System Software at the Johannes Kepler University Linz built their own examination tool called Xaminer. Currently, this tool is meant to be used in conjunction with an external video calling provider, such as Zoom. This adds unnecessary overhead and limits us to the features offered by Zoom, which are mostly tailored towards company meetings, not supervising exams. To solve these problems, we decided to build our own exam-first video calling tool. To get an understanding of the featureset we need, the following list outlines the current procedure of an online exam.

1. Students enter the external video call ahead of time.
2. Supervisors provide some last minute information about the exam.
3. Before the start of the exam, the supervisors check the student's ID cards.
4. Students open the exam page.
5. Students work on the exam.
6. When students are done, they must to sign out by writing a message in the chat.

While this flow seems simple, there are some possible interruptions, such as:

1. Checking the ID of a student who joined late without disturbing the other students.
2. Answering questions of students, without interrupting the others.

However, it does not end there. For example, after the exam is finished, the supervisors need to look at the logs and make sure everything is in order and that no cheating has occured.

Zoom does not offer answers to many of these problems, such as ID checks or one-on-one questions. Our project aims to solve those problems by providing built-in support for these standard exam interactions. We decided to build our tool as a web-based service which can easily be integrated into the existing Xaminer tool, which is web-based as well. Through the use of modern real-time communication standards, such as WebRTC and WebSockets, we are able to run video calls in the browser. This completely eliminates the need for an external application.

In addition to improving the experience for students, we aim to improve the experience for the supervisors as well. Instead of letting the students choose their own names, we automatically fetch the students' IDs, thereby eliminating confusing default names such as "Alice's iPad". For the supervisors to check the orderly process of the exam, it is important to store detailed logs of the students' actions. Another goal is to eliminate analog parts in the process, for example using a printed participant list to tick off the students after their IDs have been checked.

2 Background

Throughout this project, we are using a wide variety of different technologies, tools and libraries. Before going into the specific details, we will provide an overview of these dependencies and phrases. Additionally, we will explain the very basics of how video calling works in the browser.

2.1 Terminology

In this section, we will introduce some important concepts and phrases used throughout this thesis.

Xaminer

Xaminer is an online exam tool developed by the Institute for System Software at the Johannes Kepler University Linz. The focus for Xaminer is to provide a great experience for coding exams. Unlike Moodle, the de-factor standard for online teaching, Xaminer has an integrated code editor which provides a much better experience for the students than a plain text field. Moreover, Xaminer has an admin interface which allows the supervisors to monitor an ongoing exam. The admin interface is also used by the lecturers to create the exam and to set the questions they want to ask. Currently, Xaminer is meant to be used in conjunction with Zoom. The goal of this project is to add integrated video calling.

Supervisor

A supervisor is a person who monitors an exam. They have the ability to watch and hear all students while they are working on the exam.

Room

To allow supervisors, to focus on a small set of students, it is common to split up the participants of an exam into groups where each group has at least one supervisor. For example, dividing an exam with 200 students into 4 groups would result in groups of 50 people with at least 4 supervisors, i.e., one per group.

A room is the main theoretical concept behind our video calling system. An ongoing exam has multiple rooms, i.e., one room per exam group set by the lecturer. Each room has at least one supervisor and multiple students connected to it. All members of a room (i.e., students and supervisors) can exchange media streams (e.g., camera or microphone). A room also has state which is shared across all clients connected to it. For example, if there is an ongoing ID check in the room, the state contains a value `"idCheck"=true`. This state is controlled by a central server and synchronized across all clients connected to it.

Question Room

Within a room, there can be multiple question rooms. A question room is a one-on-one video calling session between a supervisor and a student. While the supervisor and the student are in a question room, no other student or supervisor can hear them.

2.2 Video Calls in the Browser

Unlike text data, video streams have a set of complex requirements:

R1. They need to be streamed in real-time. This means that, instead of periodically pulling chunks of data, the receiver needs to receive data as soon as it is generated by the sender. Stream sources might be a camera or microphone.

R2. The communication must be full duplex. This means that both sides must be able to send and receive media streams, i.e., that all members of a call can send and receive audio and video data.

R3. The latency must be low. For a video call to feel natural, it must be possible for the parties of the call to communicate as they would in person. Even short delays are perceived by humans and make the experience worse.

R4. It must support a large number of clients. Our goal for this project is to support up to 30 clients in a single room and up to 10 full rooms at the same time.

R5. Our transmission protocol must support the large amounts of data that audio and video streams entail.

To achieve the video calling functionality we desire, we need a transmission technology that supports all of those requirements and is available in the browser. Strictly speaking, there are only two technologies that support all of those requirements: WebSockets and WebRTC. We are not able to use basic HTTP-requests, polling (i.e., sending requests over and over to fetch new data) or long polling (i.e., a more efficient version of polling, where connections are kept open until new data is generated) effectively. We will now take a closer look at both WebSockets and WebRTC to be able to choose the one that fits our project best.

WebSockets

To start a WebSocket connection, the client makes an HTTP request to a server. The server then tells the client to upgrade the connection to a WebSocket. After the connection is upgraded, both parties can exchange bi-directional messages in real-time. In addition to the more common text-based WebSocket messages, one can also send binary data. This is great for video and audio streams.

This definition leads us to the question of how WebSockets fulfill the requirements outlined before:

R1. and R2. WebSockets are full duplex real-time communication channels, so requirements one and two are fully supported.

R3. WebSockets are based on HTTP. Hence, they inherit the reliable message delivery but also the overhead that comes with it. For video streams, dropped packets are acceptable. A UDP-powered transmission protocol would be desirable, as it is faster and we don't need the reliability that comes with TCP.

R4. The number of clients that we can support depends on server resources and the performance of our server side code.

R5. WebSockets don't restrict us in the size of the messages we can transfer. To a certain extent, this again comes down to server resources, since larger messages consume more memory on the server.

WebRTC

WebRTC stands for web real-time communication. It is a collection of protocols that enable the transmission of large chunks of data in real-time. Unlike WebSockets, WebRTC can be used *peer-to-peer (P2P)*, i.e., using a direct connection between multiple non-server clients. This means that we might be able to run video calls without a costly and complex server. We examine the protocols WebRTC is based on in detail in Section 3.2. It is important to note that WebRTC was built with real-time video calls in mind. Hence, the browser provides us with APIs to send camera streams directly to a WebRTC peer without requiring our code to touch the data in any way.

Now that we have a basic understanding of WebRTC, we can contrast it with the requirements we outlined already.

R1. and R2. Similar to WebSockets, WebRTC supports real-time communication. WebRTC connections don't have to be full duplex, but the WebRTC APIs provide us with a simple way to add another connection in the other direction once we have established an initial one-way connection.

R3. WebRTC can use TCP or UDP for transmission and has very little protocol overhead. This effectively means that WebRTC has a lower latency than WebSockets.

R4. A WebRTC connection is always between two parties but we can create as many connections as we need. We will examine the use of a centralized server vs multiple peer-to-peer connections later. For now, we should remember that WebRTC supports both.

R5. Unlike WebSockets, WebRTC is built for the large chunks of data that video and audio streams entail.

Choosing our transmission technology

While both protocols have their upsides and downsides, we concluded that WebRTC is the better choice for our project. WebRTC's main disadvantage is that is is more complex and, to a certain extent, harder to implement than WebSockets. WebRTC was built for P2P connections. While P2P, i.e., opening individual connections between all clients, means that we don't have to run our own server, it also entails the complexity and bandwidth overhead of using many individual connections. Further, WebRTC uses UDP which can greatly reduce the latency.

2.3 Tech Stack

We will now examine the technologies and libraries used for this project.

Node.js and TypeScript

Node.js [1] is a runtime for JavaScript (JS). Insead of running JavaScript in a browser, Node.js enables us to run it on the server. Node.js is based on Google Chrome's V8 Engine [9]. Node.js replaces browser-focused APIs, such as DOM (i.e., the API that is used to render UI elements in a browser), with APIs more tailored towards a server environment, such as a file system interface, a built-in webserver and low level TCP and UDP connections which can be useful for accessing databases. Node.js implements those APIs using a high performance interface for calling native C/C++ libraries called Node-gyp [5]. Node-gyp can also be used to extend with other native addons.

JavaScript is loosely typed which can lead to easily avoidable typing problems. To combat this, we are using TypeScript [2] instead. TypeScript is a super-set of JavaScript enhanced with strict type safety, i.e., TypeScript's syntax is based on JavaScript but adds the ability to set the types of variables, functions and classes as well as the ability to define reusable types such as interfaces and enums.

Express and Warp

While Node.js's built-in HTTP Server is a great basis, it lacks important features, such as easy HTTP body parsing (i.e., parsing the data, in our case this is JSON, that is sent to the server), and routing (i.e., mapping URL paths to handler functions). Express [3] is a thin wrapper around Node.js's *HTTP module*. It extends Node.js with routing and body parsing functionality and adds some quality of life features to make development easier.

Warp [4] is a wrapper around Express. It adds decorator-based controller classes, logging, validation, and even more quality-of-life addons to Express.

WebSockets and Socket.IO

WebSockets (WS) are long-running connections based on HTTP [10]. Each WS starts with a standard HTTP request and is then upgraded to be a WebSocket connection, if both parties support it. WebSockets make it possible to have real-time connections, where both the client and the server can send messages (full duplex). It replaces polling and long polling, where the client continuously sends requests to the server to fetch new messages.

Socket.IO [5] is a high-level protocol for real-time communication between browsers and Node.js-based servers. It supports long polling and WebSockets and transparently finds the best possible communication method. While this used to be a vital feature in the past, all modern browsers support WebSockets. Since we use the rather modern WebRTC API, we can't support older browsers even if we wanted to [2]. Hence, we disabled the fallback to long polling and don't

[1] Node.js `https://nodejs.org/`
[2] TypeScript `https://www.typescriptlang.org/`
[3] Express `https://expressjs.com/`
[4] Warp `https://github.com/herber/warp`
[5] Socket.IO `https://socket.io/`

need the protocol negotiation step anymore. Socket.IO also ensures that not correctly closed connections, for example, when the Ethernet cable is pulled, are detected and closed. Additionally, Socket.IO provides us with the ability to assign clients to rooms, which we can use to broadcast messages to multiple clients connected to the same room with ease. Socket.IO transparently serializes JavaScript objects to JSON, which makes it easier to send structured data, without having to worry about serialization ourselves.

LevelDB

LevelDB [6] is a low level key-value store developed by Google. It allows us to persistently store and retrieve data without needing to spin up a whole database server. Since LevelDB is just a key-value store it does not support accessing the data using complex queries. The only way to access stored data is by its key. This is enough for our purposes since we don't need to be be able to query our data.

Preact and React

React [7] is a popular JavaScript rendering framework. React makes it possible to write reactive UIs which automatically get re-rendered when the state changes. It transparently renders the UI to the browser's window. What makes React different from other JS frameworks, such as Vue or Angular, is its use of a JavaScript-superset called JSX. JSX adds XML and HTML to JavaScript. It allows us to embed HTML-like code right within JS. This removes the need for external templates and makes it easier to dynamically render DOM nodes.

While React is great, it has a lot of functionality which we don't need for this project. All that functionality comes at the cost of bundle size. By using React, we would essentially send a lot of code that we don't need to the client. Preact [8] solves that problem. It very closely implements React's core functionality but at a much smaller bundle-size. While React and ReactDOM together, use 44.5kB minified and gzipped [7] [8], Preact uses only 4.2kB [6]. Such a choice has to be made carefully, since React and Preact are not fully compatible. This also means that we restrict ourselves in the libraries we are able to use.

Preact and React use a so-called virtual DOM (VDOM) to keep an internal model of the rendered component tree. Every time the application, or a part of the application, is re-rendered, the framework updates the VDOM first. Then it analyzes the updated VDOM tree and only changes the part of the actual DOM tree that updated. Using a second abstract DOM-model makes React fast and versatile.

Vite

Through the use of JSX and Typescript, we are writing non-standard JavaScript. We also spread our code across many files which import from one another. That means that we can't run the code we write in the browser natively, because browsers don't support TypeScript. This is a common problem for modern web applications. To solve it, we use so-called module bundlers. The bundler we are using is called Vite[9]. Module bundlers use a variety of plugins to parse and

[6] LevelDB `https://github.com/google/leveldb`
[7] React `https://react.dev/`
[8] Preact `https://preactjs.com/`
[9] Vite `https://vitejs.dev/`

transform non-standard JavaScript (e.g., TypeScript) to a version of JavaScript modern browsers support.

Testing Frameworks

For this project, we need to test the frontend and the backend. To run our tests, we are using the Jest [10] framework. Since Jest does not have any opinions about what or how we should test, it is very versatile.

On the backend, we test the API and the WebSocket endpoints. While it is common to use a library that emulates the requests for simple REST APIs, this is not easily possible since we want to test the WebSocket functionality and the REST endpoints in conjunction. To achieve this, we decided to run an embedded instance of our API service in the tests, i.e., each test starts a full instance of our API at a random unused port. In addition to that, we built a small set of helper functions to send HTTP requests and to communicate with our test server over WebSocket.

For the frontend, we are using the Preact Testing Library [11] to test our components, again using Jest as the test runner. Testing Library is a collection of testing primitives for many popular frontend frameworks. In addition to component tests, we are testing the individual functions of the frontend using unit tests run by Jest.

To test the frontend in a real-world scenario, we are using Playwright[12] by Microsoft. Playwright enables us to run our application in a real browser, such as Chromium (Chrome), Webkit (Safari) or Firefox, and to programmatically test the functionality, for example by playing out certain inputs and clicks.

LiveKit

LiveKit[13] is an open source real-time video streaming service. It provides everything we need to run WebRTC calls over a centralized server instead of using P2P. We will later discuss why this might be desirable.. LiveKit comes with SDKs for Node.js, the browser and many other runtimes and languages.

JSON Web Token

For this project, we are using JSON Web Tokens (JWT) for authentication. A JWT is a signed token containing an arbitrary JSON payload. Since the token contains all of its attributes, such as the permissions and the ID of the user it's associated with, the authentication is stateless, i.e., there is no need to store an authentication token in a server database. To ensure that the payload is not manipulated, each JWT contains a signature of its payload. If a user gets issued a token with the attribute `"role"="standard_user"` and changes the value to `"role"="admin_user"`, the signature won't match the payload anymore and the token is invalid. This approach makes the tokens tamper proof.

Since JWTs are not stored in a database, there is no built-in way to revoke them. This means that it is crucial that JWTs have an expiration date, so clients don't have access to resources forever.

[10] Jest `https://jestjs.io/`
[11] Testing Library `https://testing-library.com/`
[12] Playwright `https://playwright.dev/`
[13] LiveKit `https://livekit.io/`

JWTs can only be generated by the backend. To generate an authentication JWT, the client needs to send a request to an authentication endpoint with the required data (e.g., the `username` and `password`). The backend then validates that input data. If the input is valid, it generates and signs a JWT with the matching attributes. The client can then use this token to authenticate subsequent accesses to other backend endpoints, e.g., API calls.

3 Overview

In this section, we outline the architectural decisions we made and why we made them.

3.1 Client Types

For this project, we need two different types of clients, i.e., students and supervisors, each with their own permissions and actions they can take, e.g., students should not be able to access the exam settings. To make the terms more general, we opted to call students *standard clients* and supervisors *primary clients*.

Primary Clients

Primary clients are highly privileged users. This is the role exam supervisors get. Primary clients need the following functionality and privileges:

- They need to be able to change the settings of an exam, such as requiring students to share their screens or wether they are allowed to ask questions during the exam.
- They need to verify student's IDs. For this, they need to be able to enable or disable the ID check mode and they need to be able to tick off student after their IDs have been checked.
- They need to be able to receive questions and to answer them privately. While answering a question, they need to be in a one-on-one call with the student who asked the question.
- They need to see which students are online and view their camera and/or screenshare and hear their microphones. They should also be able to mute students.
- They need a log of all actions performed by the students. Moreover, they need to be able to export this log as a CSV file.
- They need to have an individual video feed for each student.

Primary clients only use Xaminer's admin UI. This is why we decided to build different UI layers for primary clients and standard clients. The primary client UI is split into three parts which are embedded into the existing Xaminer UI.

- A floating connection status widget as depicted in Figure 1, i.e., a UI element which has a fixed position in a corner, which tells the clients if they are connected to our WebSocket server and the video call. It includes a button which allows the supervisors to mute themselves.
- A settings widget, as shown in Figure 2, which they can use to update the settings of a room, view logs and connected students, answer questions and start the ID check.
- A student video widget, which displays the student's camera and screen video streams, as depicted in Figure 3. The video feeds can be maximized and a student can be muted. This widget will be embedded in Xaminer's student table, next to the attributes of a student, such as their name and the question they are currently at.

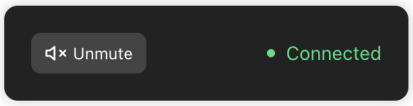


Figure 1: A connection status widget of a supervisor currently connected to a room.

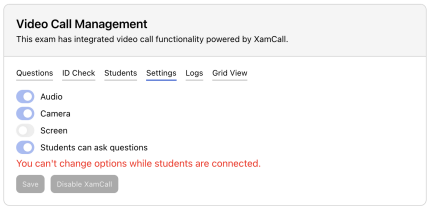


Figure 2: A settings widget for an ongoing exam.

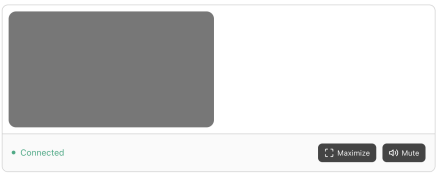


Figure 3: A student video widget with a camera stream.

Standard Client

Standard clients represent students. They are less privileged and there is information which we should, under all circumstances, hide from them. This includes the other students who are connected to the same exam. They can take the following actions:

- They need to be able to connect to all primary clients, i.e., supervisors, via WebRTC but they should not be able to connect to other students.
- They need to receive ID check status updates, if their ID has not been checked before. Upon ID check, we want to show the *ID check UI* only to those students whose IDs have not been checked before to avoid breaking the state of concentration for all other students.
- They need be able to ask questions and join a one-on-one question room with a primary client who answers their question. They need to be able to resolve the question, i.e., stop the conversation, either while they are in the one-on-one call or before, if assistance from a supervisor is no longer required. Similarly, primary clients can resolve a question as well, but only while they are answering it, i.e., while they are in a one-on-one call with a student.
- Students need to be able to receive the configuration, i.e., which media streams must be enabled and whether questions can be asked, of a room when the page loads. We need

this information to request the required permissions from the browser, before starting the video call.

- Student's interactions throughout the exam must be logged to be able to detect suspicious behaviour.

Xaminer has a specific UI for exam participants. We will embed our video calling functionality into this existing UI. Since the students can only take one action, which is asking a question, we can simply reuse the floating connection status widget from the primary client and add a button to ask a question.

Connection Structure

From the feature lists, we can extract the WebRTC connections that we want to have. Notably, we don't want standard clients, i.e., students, to be able to see or hear each other. However, we do want all primary clients, i.e., supervisors, to connect to all standard clients as shown in Figure 4. We also want all primary clients n the same room to connect to each other, in case there are multiple supervisors.

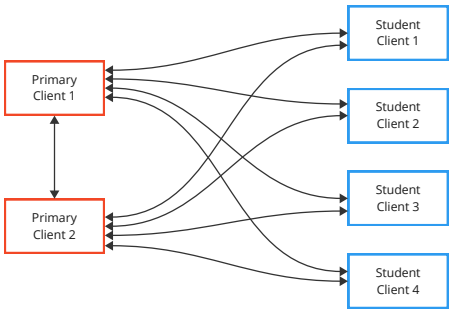


Figure 4: The WebRTC connection structure between multiple clients of different types in the same room.

3.2 Communication Protocols

WebRTC uses three protocols in conjunction: ICE, STUN and TURN. The main goal of these protocols is to establish a connection through complicated Network Address Translation (NAT) setups and to negotiate the capabilities that both sides have.

STUN - Session Traversal Utilities for NAT

STUN stands for *Session Traversal Utilities for NAT*. The STUN protocol is used to discover the configuration of a network and a client's IP address and port. Since many home and work networks use NAT (Network Address Translation), a single IP address is shared by multiple computers. NAT makes it so that individual computer don't have their own public IP addresses, instead, the network address translator, i.e., the router or firewall, has an IP address which can be used by all computers that are connected to it. The network address translator detects when a computer within the network wants to make a connection to the outside and opens a port on

the shared IP address for this connection. The network address translator then ensures that both sides can communicate through this shared IP address. However, this means that, in a peer-to-peer environment, if both sides use NAT, it is not possible to open a connection directly since a connection always needs to be opened from the inside of a network, to a concrete destination. If both sides use NAT there is no concrete destination to connect to.

As shown in Figure 5, STUN needs a publicly accessible STUN server. It is responsible for understanding the NAT configuration of a network, getting the public IP address and the public port or a client. This information can then be used to open a P2P connection. The port opened by the client to connect to the STUN server can, if the firewall supports/allows it, be reconfigured to be used for the P2P connection. If it wasn't for this centralized server, it would not be possible to open a port in a NAT network.

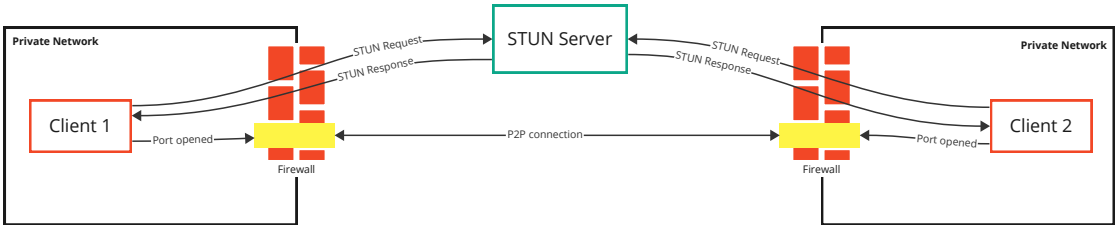


Figure 5: Opening a WebRTC connection using STUN for NAT traversal.

ICE - Interactive Connectivity Establishment

ICE stands for *Interactive Connectivity Establishment*. It is used to negotiate to best possible way for two peers to communicate with each other. ICE uses STUN to receive information about a client. As shown in Figure 6, ICE needs a centralized server, since both parties need to communicate before a P2P connection is established. Unlike STUN, ICE does not come with its own server protocol. Instead, the browser provides the application with so-called ICE candidates and it is the application's responsibility to exchange these candidate messages. In our project, we used our already existing Socket.IO server for ICE candidate communication.

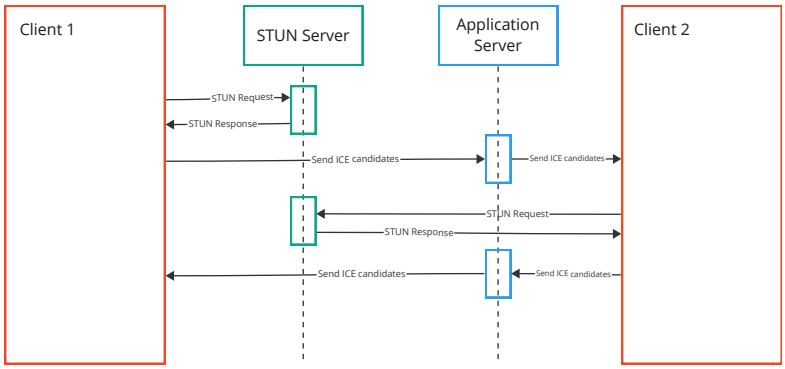


Figure 6: The flow of exchanging ICE information between two clients with a shared application server.

TURN - Traversal Using Relays around NAT

For most NAT configurations, ICE and STUN can start a peer-to-peer connection. However, some network address translators have a restriction called Symmetric NAT. With Symmetric NAT, a port opened by the network address translator can only receive data from parties to which the client has connected first. This means that a P2P connection is not possible, because for P2P, we first open a connection to the STUN server and then reuse this connection for the video call. To be able to support those clients, the browser detects this behavior and proxies the connections over a centralized server.

This is what WebRTC uses TURN (*Traversal Using Relays around NAT*) for. When a peer-to-peer connection is not possible, the browser automatically falls back to using a TURN server.

3.3 Communication Architectures

We have established that, in its essence, WebRTC is a peer-to-peer protocol. However, it is of course possible to implement WebRTC on a server and to use that as the peer. We will now examine different approaches for WebRTC connections.

P2P - Peer To Peer

P2P stands for *peer-to-peer*. This means that all members of a call who want communicate have to create individual connections between each other. Since the connections are direct, there are no servers required for sending the media streams. Say we have four clients as shown in Figure 7: Alice, Bob, Caitlin and David. They all want to connect to each other in one shared call. To achieve this with P2P, everyone of them would have to send their own media streams to the three others. In turn, they would receive one media stream from each other client.

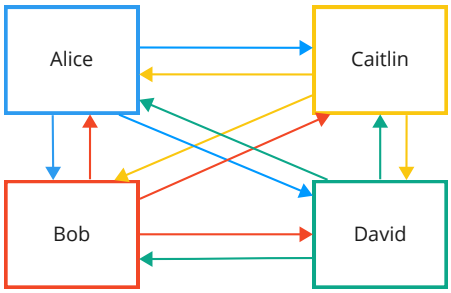


Figure 7: The connection structure between clients connected using P2P.

This results in the following benefits and drawbacks:

Benefits:

- **No centralized server required:** Video and audio streaming is quite bandwidth intensive. It would be beneficial, if we could save that infrastructure cost and keep our system simpler and easier to maintain.
- **Simpler to implement:** Instead of having to run and develop a WebRTC service, we only need to negotiate the connections between the clients. WebRTC is built for P2P which makes this quite an easy task.

Drawbacks:

- **A lot of small points of failure:** Each one of those connections can have problems which must be handled correctly. Since there are so many connections, the risk of a connection failing could be higher.
- **Clients need to send their outgoing media streams multiple times:** Having each client send the same stream multiple times can lead to large amounts of data being sent. We have to keep in mind that these clients are not servers in highly specialized datacenters. They might be mobile phones or computers in home networks with low bandwidth. To understand this, say we have a call with 31 people in a room. This means that one client, we will call them Alice, needs to send their camera, audio and screen share to 30 other clients. Let Alice's camera feed be *1.5 Mb/s*, their audio stream be *0.5 Mb/s*, and their screen stream be *2 Mb/s*. This means we can get their total upload bandwidth by calculating: $(1.5 + 0.5 + 2) * 30 = 120$. Alice would need an internet connection with at least *120 Mb/s* upload, which is unreasonable for residential internet connections in Austria.

SFU - Selective Forwarding Unit

SFU stands for *Selective Forwarding Unit* and uses a central server. An SFU tries to solve the problem of large amounts of data being sent from a client. Instead of having the client itself broadcasting its media streams, the data is sent to the SFU as shown in Figure 8. The SFU is then responsible for broadcasting it to each client who wants to receive it. Looking back at the example, Alice would only have to send *4 Mb/s* instead of *120 Mb/s*. The benefit of using an SFU is that it shifts the broadcasting to a server, which usually has much more bandwidth available. However, with an SFU each stream is still received individually by every client, instead of packing them into a single connection.

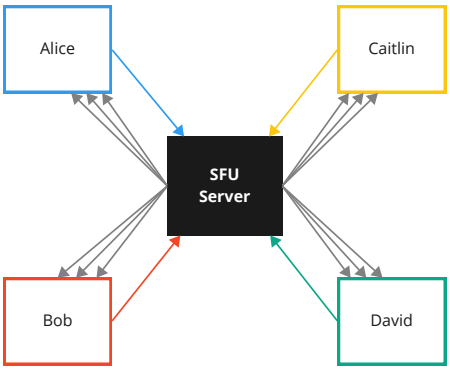


Figure 8: The connection structure between clients connected using an SFU.

Benefits:

- **Clients only need to send their media streams once:** This greatly reduces bandwidth and resource consumption in general. It is often not possible to send large amounts of data from home networks.
- **Streams are received individually:** We don't have to break data packets up into individual streams on the client side, which can be quite resource-intensive.

Drawbacks:

- **Need for a server:** For this approach we have to implement WebRTC on the server side. Such a server can be quite complex and may consume large amounts of bandwidth since it has to broadcast all media streams. A good implementation of such a service, called LiveKit, exists.

MCU - Multipoint Control Unit

Similar to SFU, an MCU (*Multipoint Control Unit*) uses a centralized server to relay the media streams between the clients. We have seen that an SFU sends the media streams a client has requested using individual connections for each stream. An MCU packages all of those streams into a single connection which the client then has to break up to get the individual media streams as shown in Figure 9

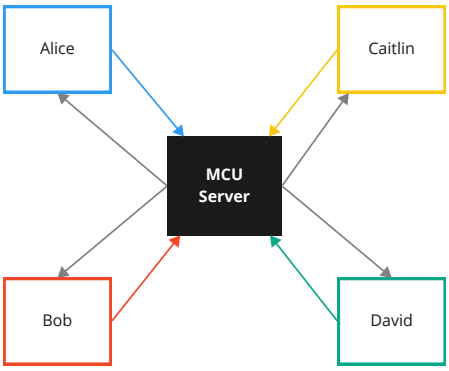


Figure 9: The connection structure between clients connected using an MCU.

Benefits:

- **Even lower bandwidth consumption:** An MCU further decreases the bandwidth we need, compared to an SFU, because we don't have the overhead of multiple connections.

Drawbacks:

- **Complicated server logic:** The server needs the logic and power required to package multiple streams into one. This makes the server's resource consumption higher and the implementation more complex.
- **Need to split the streams up:** With an MCU, the client has to split up the combined stream it receives into the individual ones. This leads to higher resource consumption and more complexity on the client side, i.e., inside the browser.
- **Marginal benefit:** While we would save some bandwidth, the large media streams still have to be sent. Sending them over a single stream does not make them smaller, it only reduces the protocol overhead. In the end we would have much more complexity and resource consumption for only a small decrease in bandwidth.

Choosing a transmission mode

While we knew the drawbacks of P2P initially, we choose to build an initial version of this project with it to be able to analyze the drawbacks in a real-world environment. While P2P was fine for the initial tests, we noticed that the CPU usage of the primary clients, which have to send their audio streams to all other clients, was quite high. This means that P2P is not only problematic with regards to bandwidth, it also puts a lot of strain on the sender's computer.

Further, during a test at Johannes Kepler University, we noticed that some members of the university's network had problems connecting over WebRTC. This was likely due to the fact that our P2P version lacked a TURN server. We were not able to investigate that further. For clients where the university's network is configured in a restrictive way, peer-to-peer connections seem to not be possible.

With all of that in mind, we made the choice to switch from P2P to using an SFU. We

decided to use LiveKit, as their product looks mature and reliable, and they offer great SDKs for JavaScript in the browser and on the backend (Node.js). This change forced us to re-architect some parts of this project but since LiveKit's SDKs are built to be easy-to-use, this migration was quite straight forward.

3.4 LiveKit

As discussed, we decided to switch from P2P to an SFU for the reduced upload bandwidth and the added features. Writing our own SFU was out of the scope of this project, so we searched for an off-the-shelf solution. While there are many companies offering hosted solutions, we needed one 1) that we could host ourselves, due to university restrictions, 2) which is trustworthy and well tested. LiveKit ticked all of those boxes and it is open source. It is very easy to set up on a virtual machine, using docker compose, a popular containerization system, and LiveKit's CLI.

LiveKit handles the WebRTC negotiations, with ICE, STUN and possibly TURN, on its own. It does that by running its own WebSocket service to offer real-time communication and negotiation capabilities.

LiveKit does not have any opinions about the products it can be used for. It simply serves as a primitive for sending and receiving media streams. This means that we need to tell LiveKit about the room, which media streams we want to send and which clients we want to connect to. The permissions of a client are controlled using a *JSON Web Token* (JWT) which is generated by our backend and sent back to the client. The client then passes that token on to LiveKit for authentication. The communication is showcased in Figure 10.

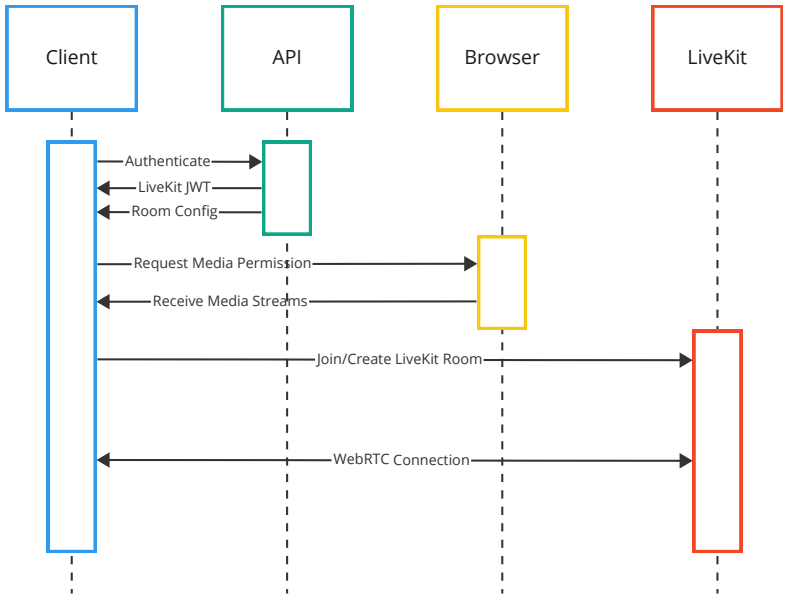


Figure 10: The handshake required to establish a connection using LiveKit.

Since LiveKit is an SFU, it can manipulate the media streams in ways which would not be possible with P2P connections in the browser:

- **Video simulcast** means that instead of publishing a video stream at a single resolution, LiveKit requests multiple different resolutions from the browser and publishes all of them simultaneously. This allows LiveKit to send each client the matching resolution depending on the size of the video element. In our case, small video previews don't need to be in HD, but it would be preferable to have a high resolution when a video feed is maximized. Achieving such a functionality with plain WebRTC would be hard.
- **Adaptive streaming** means that the client detects when a video element is not in view. When this is the case, it notifies LiveKit which then stops sending it actual video data and instead sends a 1x1 pixel pseudo-video just to keep the connection open and healthy. However, LiveKit continues to send the client's audio stream, so they can still be heard. Achieving something similar using plain WebRTC would be possible, by closing a P2P connection when it is not used. However, when it needs to be used again, e.g., when the video enters the viewport again by scrolling, we would have to open the connection and perform the whole negotiation flow all over again. This does take a noticeable amount of time and thus affects the user experience negatively. In addition to that come all of the risks and problems associated with continuously starting and stopping WebRTC connections.

Using these features, we see that not only the sender's bandwidth, if multiple clients are involved, is reduced, but we also reduce the receiving bandwidth especially for the supervisors who receive up to 30 video streams at once.

3.5 API and Video Streaming

While the video transmission technology is a vital part of our application, we also need a central API service which connects the clients together and allows them to perform the handshakes. This API server is written in TypeScript and runs using Node.js. Not only does it help us with handshakes and authentication but it also enables us to build additional features, such as ID checks and question rooms. To enable those features, the server has real-time capabilities as well.

4 Server

It is evident that we need a centralized server to facilitate connection negotiation and to generate LiveKit authentication JWTs. Aside from the core video streaming functionality, we also need a server to support additional features, such as ID check sessions and question rooms. In the following, will examine the features we anticipate having with this backend service to gain a better understanding of the implementation:

- **Authentication:** In Section 3.1, we outlined the capabilities that our primary clients (i.e., exam supervisors) and standard clients (i.e., students) should have. This makes it evident that we need some sort of authentication. Additionally, we need to integrate our authentication system with Xaminer's existing authentication.
- **ID check mode:** Since ID checks are a mandatory part of every exam, we decided to add a special UI for it. Supervisors should be able to start an ID check and tick-off students, while students should be notified of an ID check so they can show their ID cards to the camera.
- **Question rooms:** Students should be able to ask questions and supervisors should be able to answer them. It is crucial that the student and supervisor in a question room cannot be heard by anyone else. For this to work, we need to notify all other clients, that they should mute the clients in the question room.
- **Logging:** In order for supervisors to check for anomalies during and after the exam, we need a way to log certain actions such as when a student joins or leaves a room. It is necessary to have the ability to export the logs.
- **Information about the room:** The client needs to know certain information about the room before starting the call i.e., which media device permissions to request or whether questions are enabled.

This list indicates that we need real-time communication for our backend server. Since the communication is purely text-based, using WebSockets is the most sensible approach.

While Node.js and the browser have built-in APIs for WebSockets, these APIs lack certain functionality we need, such as the ability to send the same message to multiple clients at once or automatic detection of incorrectly closed connection (i.e., if a user unplugs their Ethertnet cable). Instead of implementing this ourselves, we opted to use the Socket.IO library. Socket.IO offers both a client-side and a server-side libraries for Node.js and the browser as well as many other platforms. Socket.IO provides us with important functionality, e.g., automatic detection of incorrectly closed connections, the ability to assign clients to rooms and the ability to broadcast to multiple clients at once.

Since not all actions need to be real-time, e.g., issuing authentication JWTs and changing the configuration of a room, we also developed a REST API. While all of that could have been sent over the WebSocket connection, we found that it is more comfortable to work with basic HTTP requests for those interactions. For the REST-API, we decided to use the Express-based Warp library. Warp provides us with an elegant interface for writing class-based controllers and built-in

support for authentication. For validation, we decided the use the Yup [14]. It is important to note that, while some WebSocket frameworks don't support sharing a port with a vanilla HTTP server, Socket.IO and Express make this very easy.

We have opted to use our backend API server to also serve the static assets for the client side. In a real-world project, we would use a highly optimized CDN-based static web service such as Vercel, Netlify or Cloudflare Pages. For this project, that would have been overkill.

4.1 Authentication

Since this system is not security critical and instant session invalidation, for example when someone logs out, is not necessary, we decided to use JSON Web Tokens (JWT). This saves us the work of having to implement a session database and makes the authentication stateless. The use of JWTs makes our system independent of Xaminer, which is great for testing. We also utilize this property in our demo frontend. However, this also means that we need a way to integrate our authentication system with Xaminer's.

As outlined in Section 2.3, JWTs are signed tokens with a JSON payload. This payload is set by our backend and contains the data we need to authenticate users. The tokens include standard attributes, such as the expiration date but also custom attributes such as the `clientId` (e.g., the student's unique ID number), the `clientLabel` (e.g., the student's name) and the `roomId` (i.e., unique ID of the exam combined with the ID of the exam group, the room is associated with).

Authentication over WebSockets

When a client connects to the WebSocket server, they are not immediately connected to any room and therefore cannot perform any actions initially. However, this behavior proves to be useful as it allows us to establish the WebSocket connection without requiring the client's authentication token, eliminating the need to wait for the connection to open later. A room might only be entered by users that authenticated themselves to take part in the given exam.

When the client is ready to connect to the room and has the authentication token, it sends an *init* message with its authentication token. If the token is valid, the server adds the client to the room and executes all associated procedures.

After the client has authenticated, the server remembers that the connection is authenticated and the client does not need to send its authentication token alongside its further messages.

Authentication for the REST API

Our REST API can be split into four parts:

- **Authentication Endpoints**: Used to generate a token.
- **Public Endpoints**: Are always publicly accessible, i.e., without authentication. An example for such a public endpoint is the "/ping" endpoint for health checks.
- **Admin Endpoints**: Authenticated endpoints which can only be access by primary clients, i.e., clients, who have `"kind"="primary"` in their authentication JWT.

[14] Yup: `https://github.com/jquense/yup`

- **Common Endpoints**: Authenticated endpoints which can be accessed by primary and standard clients, i.e., clients, who have `"kind"="client"`.

Authentication tokens are only required for the Admin Endpoints and the Common Endpoints. Clients can access these endpoints depending on the "kind" attribute in their JWT.

Issuing Tokens

Before being able to access the authenticated parts of the API, a client is required to fetch an authentication token from the `TokenController` located at `/token`.

If the client wants a student token, it is required to send its `examId` and `studentId`. The client can optionally pass a `groupId`. If no `groupId` is passed, the API makes a request to Xaminer's API to find out to which group the student belongs to. The data is then sent to Xaminer for verification. If the provided data is correct, i.e., if the student is registered for the given exam in the given group), the API generates, signs and returns a JWT with the student role.

If the client wants to retrieve a token for a primary client, it is required to send the supervisor's `privateKey` and a `groupId`. The client can optionally send the `examId` to make sure the `privateKey` matches the exam. The API then makes a request to Xaminer's API to validate the `privateKey`. If the `privateKey` is valid, a primary client token is generated.

The token issuing endpoints get called transparently by the frontend before making any other requests.

4.2 REST API

As outlined before, our REST API can be categorized into four parts: 1) Authentication Endpoints, 2) Public Endpoints, 3) Admin Endpoints, and 4) Common Endpoints. We already looked at the authentication endpoints in the previous section.

There currently is only one public endpoint: `/ping`. It can be used to monitor the health of the API service. If the API is healthy, it responds with a JSON payload and the 200 HTTP status code.

The common endpoints are used to fetch general information about a room. All common endpoints are part of the `ClientController`. To get an understanding of what a common endpoint does, it makes sense to look at the code for an endpoint. The `GET /client/room` endpoint, shown in Listing 1 serves as a good example.

Listing 1: `getRoom` API endpoint handler function

```
@Get('/room') (1)
@ClientAuthMiddleware() (2)
getRoom(@TokenParam() token: IAuthToken) { (3)
  let room = getData().getRoom(token.roomId); (4)

  return {
    room: presentRoom(room).config (5)
  };
}
```

1. The @Get(path) decorator from Warp is used to tell the framework about the route and method we want to serve this endpoint at.
2. The @ClientAuthMiddleware() decorator is used to validate the authentication token. It permits both primary and standard clients to access this route.
3. The result of the authentication middleware (i.e., provided by the @ClientAuthMiddleware() decorator) is accessible using the @TokenParam() decorator.
4. In the controller, we can use the roomId from the authentication token to get information about the client's room.
5. We use the "presentRoom" function to get an object containing the information of the room. We are using a presenter function (i.e., a function that further processes a value to be sent back to the client) instead of sending the data directly, to make sure that 1) the response has a common format which the client can rely on and 2) to remove values which are internal and should not be sent to the client, e.g., the private key of connected supervisors.

4.3 WebSocket Connections and State

The WebSocket part of our API handles the real-time communication required for some aspects of this project. To connect to a room, the client sends an "init" message with its authentication token. The server then validates 1) the token, 2) adds the client to the room and registers all required event listeners, 3) sends back the current state for the room depending on the client's permissions, and 4) informs the other clients about the new client so they can connect to it over WebRTC. The inverse procedure is called when the client disconnects from the WebSocket.

The use of WebSockets enables us to build real-time synchronization for reactive data-structures controlled by the server. For every action that is taken, the server is the single source of truth. It notifies the clients, depending on their role, about the data that has changed so they can update their local state. This, for example enables us to have a list of students that updates in real-time or to display the ID check UI in real-time.

A rather interesting real-time message flow is required for question rooms as shown in Figure 11.

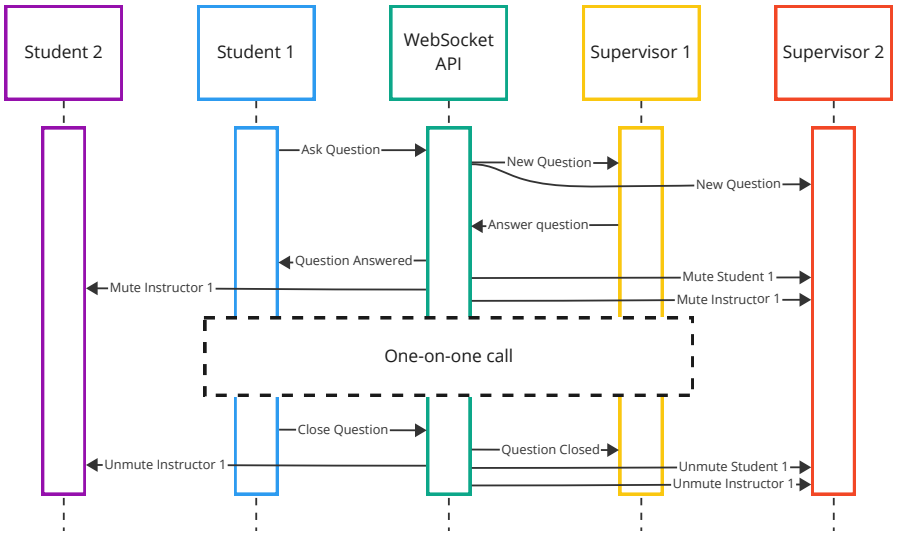


Figure 11: The communication flow when a client asks a question.

- After a student asks a question, all supervisors are informed that there is a new question.
- When a supervisor answers, the student who has asked the question gets notified that their question will now be answered and that they can establish the one-on-one call. The other student clients get notified that they should mute supervisor 1. The other supervisors get notified that they should mute student 1 and supervisor 1. (Side note: Students can never hear each other (i.e., they always mute each other) and thus do not have to explicitly mute student 1)
- When the student or the supervisor decides the close the question, the other question-member is notified that the question was closed. All non-question-members get notified that they should unmute the supervisor.

This neatly indicates that each client has its own state, but an action taken by a single client can affect some or all other clients. The API is responsible for coupling the individual states of all clients while respecting permissions and the current state of the clients. For this to work it is crucial that the server knows everything about everyone.

To avoid race conditions we opted to only access the data synchronously through the use of in-memory data structures, such as maps, sets and arrays. This is possible since we have an upper limit of about 300 clients, which can easily be handled by in-memory data structures and a single instance. If we needed support for multiple instances and did not want to store everything in memory, Redis, an in-memory key-value store with support for real-time updates, in conjunction with atomic operations would be a good fit.

4.4 Video Calling

As pointed out earlier, we initially implemented the video calling functionality in a peer-to-peer manner. During our tests, we noticed that P2P prohibited us from using helpful features, such as adaptive streaming while requiring the client to send large amounts of data. In this section we will examine both of those implementations from the perspective of the server.

P2P implementation

As a quick refresher: the STUN protocol is used the generate information about the NAT configuration and public IP address of a client. STUN has its own server implementation and there are public servers available, hence there was no need for us to implement this ourselves. However, the information generated by the STUN handshake needs to be shared between the clients. This is done using ICE candidates provided to our application by the browser. It is our application's job to exchange the ICE candiadates between the clients. It makes sense to use our existing Socket.IO server for this, as this information has to be shared between clients in real-time to ensure that establishing a connection does not take too long. When a client wants to open a P2P connection to another client, the exchange looks as showcased in Figure 12.

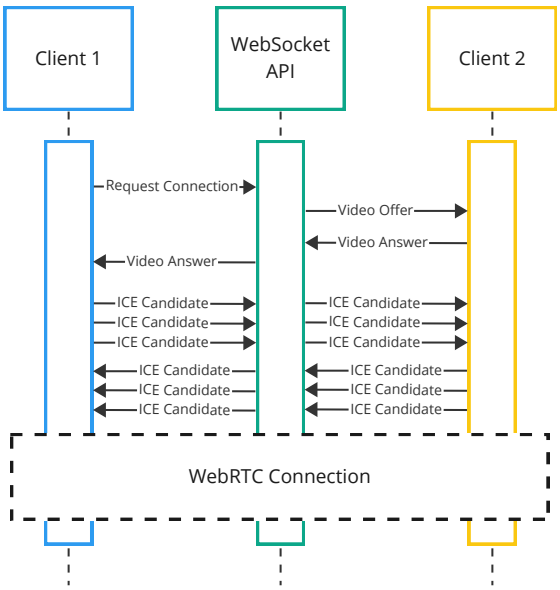


Figure 12: WebRTC connection setup handshake using P2P.

Implementation with LiveKit

When using LiveKit, the clients also need to exchange ICE candidates, but with a WebRTC proxy and not with each other. This is handled transparently by LiveKit. To achieve this, LiveKit runs its own WebSocket server which it uses to exchange the ICE candidates.

To link our application with LiveKit, all we have to do is generate authentication JWTs and forward them to our frontend. LiveKit provides a nice Node.js SDK which we can use to generate

the JWT. We implemented an API endpoint, which uses our existing authentication toksn to ensure that the client is who they say they are and then returns a matching LiveKit token. One could say that we have an endpoint which allows the client to upgrade their already issued API token to a LiveKit token. An important part of this token's payload is the `roomId` and the `identifier`. In our case the roomId is the same as the roomId we use to categorize exam rooms and the identifier is a client's unique ID.

4.5 Testing

Our REST and WebSocket APIs don't do anything complicated algorithmically, such as complex data structures or data processing algorithms. However, it has to ensure that handshakes and reactive state transitions work correctly. In the following, we will introduce how we designed our tests to validate that these flows work correctly.

API Tests

To test our REST and WebSocket APIs, we opted to write E2E tests which emulate real message flows with multiple clients over WebSockets and using our REST endpoints. We need to test both the WebSocket part run using Socket.IO and the REST endpoints in conjunction, since they are connected. For example, when the settings get updated by sending a request to a REST endpoint, all clients get notified about the change via a WebSocket message. To correctly test this behavior, i.e., making sure that an HTTP request to a given REST endpoint leads to the desired WebSocket messages being sent, only sending a request to the REST API does not suffice. Moreover, we also use this to test that WebSocket messages only get sent to the clients we expect. For example, updates to the list of connected students should only get sent to primary clients. To run the API tests, we are using the Jest test runner.

We wrote our own small testing utility module, which enables us to spin up instances of our API which run in the test runner. We also implemented some primitives which allow us to make requests over HTTP and to communicate over WebSocket. We then use assertions to validate the results. These test utilities also generate API tokens with the role we need.

Continuous Integration and Deployment

It is important to run tests automatically in a controlled environment. It is common to do this on every push to a Git repository using virtual machines which are configured programmatically. This is called continuous integration (CI). To run our CI workflows, we use GitHub Actions [15].

GitHub Actions can be configured to run arbitrarily complex workflows using multiple VMs which can run concurrently or sequentially. These workflows can be configured using YAML files contained in a special directory. When a test fails, all associated parties will be notified via email, and the logs can be accessed on GitHub's website.

[15] GitHub Actions `https://github.com/features/actions`

4.6 Deployment

To allow people to use our project, we need to run it on a server; this is called deployment.

LiveKit LiveKit offers a nice installer CLI to generate the configuration files needed. It runs its services in a container environment called Docker Compose. This allows the instances to be created, updated, deleted and rebuilt very easily using only a few commands. Since LiveKit uses a wide range of non-configurable ports for WebRTC and its own HTTP and WebSocket API, we opted to run LiveKit on its own virtual machine, without anything else running on it. This simplifies configuration and improves security, since we can assume that the VM is clean.

API For our API, we decided to reuse Xaminer's existing infrastructure. While this required us to change some parts of Xaminer's current setup it made sense due to limited server infrastructure. Xaminer's server was not yet set up using a containerization system such as Docker. While we would have liked to have such a system, we did not want to take the risk associated with rearchitecting a running production system. This means that we have to run the Node.js service alongside Xaminer's JVM service directly on the existing VM.

PM2 We want to 1) run the APIs Node.js process in the background, 2) have it start with the system and 3) have automatic restarts in case of a failure. While this could be implemented using a shell script being run as a daemon, we decided to go for a more sophisticated alternative, called PM2 [16]. PM2 is a process manager mainly designed for Node.js. It offers a command line interface to configure or restart the services it manages.

Caddy Xaminer's server ran an instance of Apache HTTPD which was configured to proxy Xaminer's Spring Boot API. It ensured that the API had an up-to-date SSL certificate issued by Let's Encrypt. While Apache HTTPD certainly is a great HTTP server, it is quite cumbersome to configure, which lead us to the decision to replace it with a more modern alternative called Caddy [17]. Caddy's headline features are that it 1) automatically generates SSL certificates using Let's Encrypt and 2) it is very easy to configure. In our case, we replaced five lengthy configuration files, with a single one with fewer then ten lines as shown in Listing 2.

Listing 2: `Caddyfile` configuration for our Caddy server

```
http://xaminer.jku.at, http://xamcall-api.herber.space {
        redir https://{host}{uri}
}
https://xaminer.jku.at {
        reverse_proxy :8080
}
https://xamcall-api.herber.space {
        reverse_proxy :3001
}
```

[16] PM2 `https://pm2.keymetrics.io/`
[17] Caddy `https://caddyserver.com/`

5 Client

Since our client-side UI is meant to be embedded in Xaminer's existing UI, we decided not to treat the client side as one application but rather as a collection of connected widget with some shared state. This approach is rather uncommon but made sense for us since we want to render different components at different locations in Xaminer's existing UI.

5.1 State

Using our widget-based approach means that we cannot use Preact's built-in state management directly. Instead, we developed our own reactive state classes, i.e., classes which hold some data (e.g., a list of connected students) and related actions (e.g., remove student) and notify listeners when the data changes. This state is then connected to Preact's internal state. When the state of one of our state classes changes, we inform Preact about it, which then re-renders the UI. What this approach provides us with is the ability to rely on Preact's reactive re-rendering functionality, while having our own state management code, separate from Preact.

Since the state is essentially controlled by the server, we need to listen for events over WebSocket using Socket.IO. Each state class registers the Socket.IO event listeners it needs and then updates its own state accordingly when it receives a message.

Some actions have to be sent to the server using Socket.IO and some using our REST API. To make this simple, we implemented our own SDK for the API. The SDK requests are performed by the state classes after some action is triggered. The server then sends back the updates to the state, which get integrated into to client's local state. Some actions, such as *start ID check* also manipulate the state of other clients, which get informed about the updates over the WebSocket connection as well.

5.2 Components and Widgets

While each widget provides different functionality, there are a lot of shared UI elements, such as buttons and spinners. To make the UI elements reusable, we put them into their own components. Components can be combined to form other components, e.g., we could have component with a list of students. Each student record could then be its own component. The student record could be made up of a box, a button and some text. Each of those items could be its own component.

Students

For students, we want to keep the UI parts to a minimum so they can focus on the exam. Looking at the features we have described previously, only if a question arises the students have to act proactively.

For the students, we are embedding a small floating widget, i.e., a widget which is fixed in the top-right corner of the browser window, in the student's exam view as depicted in Figure 13. This widget tells them that they are connected to our video calling system. It also has a button where students can ask a question.

In addition to this permanent widget, we also need some UI elements to show the students when the ID check is being performed as shown in Figure 14. To make it easy for the students

to check whether their ID card is in frame and clearly visible, we show a big preview of their camera stream along with a short notice that they should show their ID to the camera.

One very important aspect for us is to make the students aware that the exam has video-calling functionality built-in and that they need to enable camera, microphone and possibly screensharing to be able to take part in the exam. While the browser is responsible for asking for those permissions, the native prompts lack any explanation and feel intimidating. To combat this, we show our own dialog explaining the details to the students, before showing the native confirmation prompts as shown in Figure 15

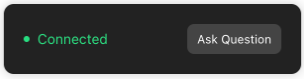


Figure 13: A student widget which shows that the student is connected

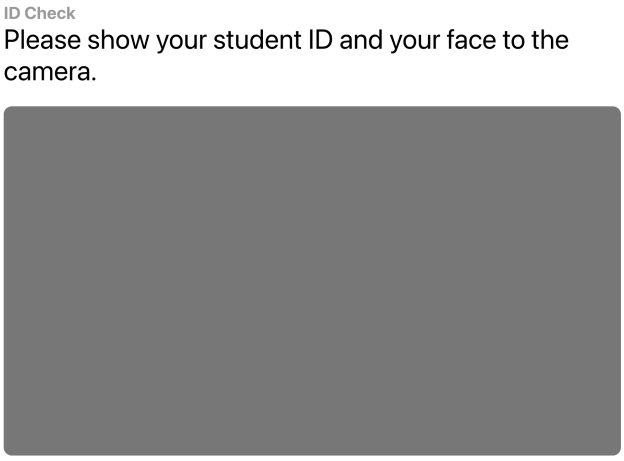


Figure 14: A student's view while the ID check is being performed

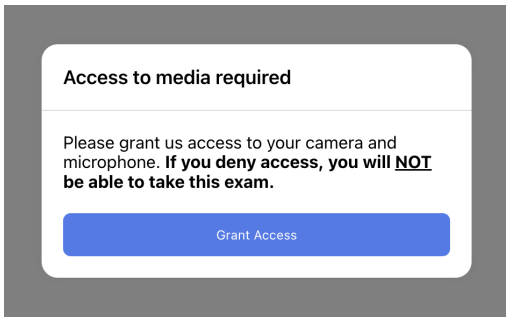


Figure 15: A student widget which shows that the student is connected

Supervisors

Contrary to the students exam view, where we opted for a floating widget in a corner, for the supervisors, we need to embed certain UI elements into the existing layout. This leads to the problem that we have to render parts of our Preact application into Xaminer's Vue.js [18] frontend. We proactively planned for this and implemented our own centralized state management which can be shared between multiple Preact applications. This means that each widget we render is its own Preact application but will be connected to our shared state. We will now describe the three different UI widget we need for the supervisors.

1. There is a floating widget for important actions (e.g., a mute button) and for displaying the connection status, similar to the one we have for the students as shown in Figure 1.
2. To be able to configure the properties of the video call, we have a settings widget. This settings widget is embedded in Xaminer's admin view as depicted in Figure 17.
3. Additionally, we have a widget which shows the video feed of a given student as shown in Figure 18. It is possible to maximize a student's video and to mute them for the supervisor. We embed one instance of this widget for each student in Xaminer's student table.

Figure 16: A supervisor connection widget.

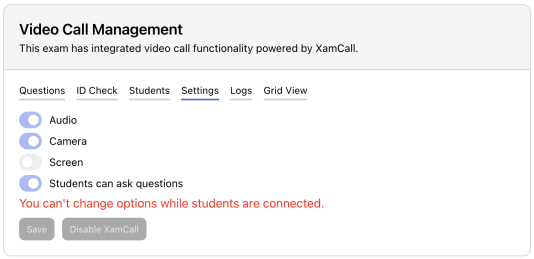


Figure 17: A supervisor settings widget.

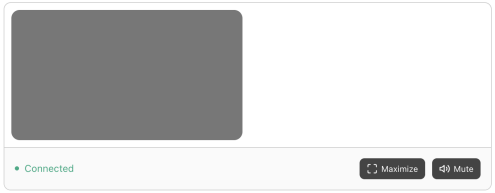


Figure 18: A student's video feed for a supervisor.

[18] Vue.js `https://vuejs.org/`

Preact in Vue: To render from Preact to Vue.js we are using a trick showcased in Figure 19 and Listing 3. First we render the Preact application into a standard DOM node which is returned from the render function. We are then using a Vue.js component, which calls the Preact render function and receives this node. The Vue.js component only has a div. We use Vue.js's ref-functionality [19] to receive a reference to the native DOM node of the Vue.js div. When we have both the Vue.js div and the Preact element, we use the standard DOM API to set the Preact application as a child node of the div in Vue.js. Thereby we have embedded Preact in Vue.js. Since Vue.js is very efficient and makes sure to reuse DOM nodes if possible, it likely won't re-render the component. However, if it does, we simple take the Preact-node and append it to the div in the new Vue.js component instance.

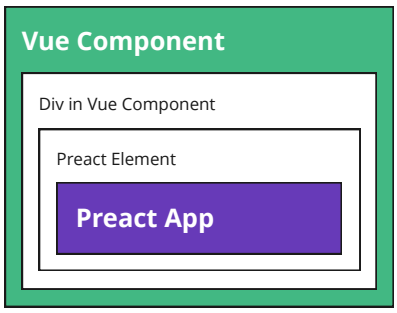


Figure 19: Embedding a Preact app in a Vue.js component.

Listing 3: A Vue component which renders a Preact application into itself

```
<template>
  <h1>Hello from Vue</h1>
  <div ref="el" /> <!-- Mark the ref of the DOM element -->
</template>

<script lang="ts">
@Component
export default class Widget extends Vue {
  @Ref('el') el!: HTMLElement; // Retrieve the DOM element using its ref.

  mounted () {
    // Render a Preact application into the ref we receive from Vue.
    render(<h1>Hello from Preact</h1>, this.el);

    // The result is a page displaying
    // "Hello from Vue" and after that "Hello from Preact".
  }
}
</script>
```

[19] Vue.js ref `https://vuejs.org/guide/essentials/template-refs.html`

5.3 Video Calling

Implementation using P2P

We already explained the handshake required to create a WebRTC connection in Section 4.4. Remember, that we use our own backend to exchange negotiation messages between the two clients. To implement this on the frontend, we first need to establish a connection to the WebSocket server to exchange the negotiation messages, e.g., ICE candidates, for WebRTC. The steps of the connection are listed below and visualized in Figure 20

1. For the students, we start a new `RTCPeerConnection` instance for all connected primary clients. This class uses event listeners to provide us with the messages it wants to exchange with the other side. When we receive a message, we pass it on to the target client using our WebSocket server.

2. The `negotiationNeeded` message is the first one we receive. It tells us that the browser is ready and wants to start the connection. We use the data of this message to create a `RequestConnection` event in our system. The WebSocket server then transforms the `RequestConnection` event to a `VideoOffer`.

3. When we receive a `VideoOffer` event on the other side, we also create a new `RTCPeerConnection` instance, and provide it with the `RTCSessionDescription` we received in the `VideoOffer`. The other side then send back a `VideoAnswer` with information about itself.

4. Now, both clients know about each other, but they still need to exchange their capabilities using ICE before the media streams can be sent. As explained in Section 4.4, ICE-messages are transmitted using our WebSocket API. When that is done, the video connection is established.

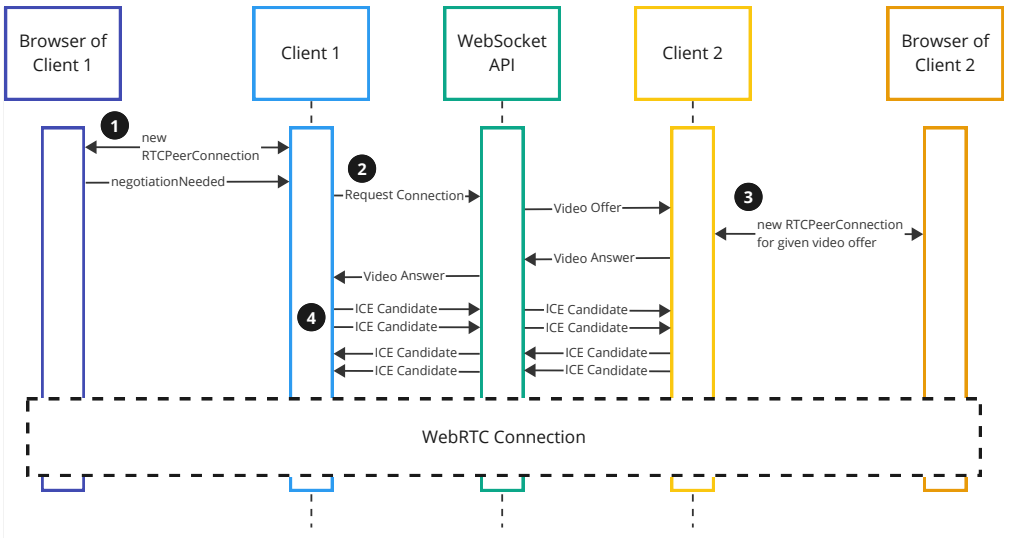


Figure 20: Embedding a Preact application in a Vue.js component.

To send data accross this connection, we need access to the user's media streams. To get a camera and microphone stream, we use the `getDisplayMedia` function provided by the browser. To get a screenshare, we use the `getUserMedia` function. Both of those functions prompt the user to allow us to use them. If the user blocks access, an exception is thrown which we handle by telling the user that they must provide us with access to be able to take part in the exam. We can pass some options to the `getUserMedia` and `getDisplayMedia` functions, e.g., the option to enable echo suppression or the option to set the resolution and frame rate we wish to receive.

Implementation with LiveKit

Internally, LiveKit does the same thing we have just described. Luckily, all of that is completely transparent to us. Once we have provided LiveKit with an authentication token (i.e., a JWT that holds information about the client and the room it is connecting to), it can connect the clients in a room. We need to remember that LiveKit uses a central WebRTC proxy server, so the connections are not directly between the client but rather between a client and the server.

To connect two clients, we need to tell the LiveKit server that it should pass the media streams from one client to the other. We listen to LiveKit's `TrackPublished` event which tells us about a new stream being published by another client in the same room. If the current client is a primary client, it will subscribe to all streams. This is because primary client are connected to all their students and all other primary clients. If the current client is a student, we only connect to primary clients, since students should not be able to see or hear each other.

5.4 Testing

Similarly to the backend, we are using the Jest test runner. Testing the client side is a bit more complicated than testing the backend. The API provided by the backend is built to be used by machines. However, the frontend is built to be used by humans. To test the frontend effectively, we employ three different types of tests.

Unit Tests

Unit tests test internal parts, such as functions, individually. They are rather easy to implement but can only be used to a limited extent for frontend applications, since they can not test the rendered UI.

Component Tests

To test components, we are using the Testing Library [20], a common component testing framework with support for a wide range of render libraries. The Testing Library emulates the browser in a Node.js environment. This makes it possible to render components and to perform assertions on the rendered result. Since no server instance is running during the component test, only certain parts of the frontend can be tested this way. Moreover, component tests are run in a Node.js environment which only emulates the browser, hence, certain behavior cannot be tested fully, such as requesting camera streams.

[20] Testing Library `https://testing-library.com/`

End-To-End Tests

So far we have only tested individual parts of our frontend in an emulated browser environment. This alone does not say much about our application as a whole. To test our application as a whole, in a real-world environment, we need to open it as a web page in an actual browser, with a running backend server. These types of tests are called end-to-end (E2E) tests, as they use a real instance of our frontend and backend to perform tests on.

Most browsers support being controlled programmatically, i.e., controlling browser behaviour such as clicks and page navigation thought code. To make this easy, e.g., accessible, Microsoft developed Playwight [21]. Playwright provides us with a standardized interface to control Apple Webkit (i.e., the open source version of Safari), Google Chromium (i.e., the open source version of Google Chrome) and Mozilla Firefox using Node.js.

In E2E tests, we write instructions telling the browser what to do, e.g., clicking on a button, typing some text or navigating to a page. After the action is performed by the browser, we can access all kinds of information about the page, such as the URL and even the whole DOM tree. We can use this state to assert that the action has been performed correctly. Playwright provides us with nice APIs to assert parts of the page, e.g., `page.getByText('XamCall room enabled').isVisible()`.

Since some updates in the browser take a few moments until the next frame is rendered or until something is ready, Playwright uses retries and timeouts for actions and information retrievals.

While Playwright is a great testing framework, it is rather uncommon to test WebRTC and hence we encountered some problems:

- In Apple Webkit, we were not able to get the tests working. Chromium and Firefox provide flags to tell the browser not to ask for permissions. In WebKit this did not seem to work. On Linux, WebKit just timed out when we needed access to camera streams. On MacOS, Webkit opened a permission dialog, even though it was configured to operate in headless mode. This means that we can not rely on automated tests using WebKit.

- In Google Chromium, we were able to disable the permission prompts and to serve a fake video stream. Using this functionality, we were able to get tests in Chromium working on Windows and MacOS system with physical cameras connected. Chromium never used those physical cameras but it seemed to need them to exist. On our Linux-powered CI system, with no physical cameras attached, Chromium did not allow us to use the media stream APIs.

- The only well-behaving browser in this case was Firefox. It worked on all systems we tested it on and it supports fake media streams, even without physical webcams connected.

With that in mind, it makes sense to test in Chromium locally and Firefox locally and in CI. Sadly, we are not able to test our project in Webkit.

CI/CD

For the frontend tests, we are again using GitHub actions.

[21] Playwright `https://playwright.dev/`

5.5 Demo Frontend

We intentionally kept our system highly independent of Xaminer. This decision allows us to use the video calling functionality in environments other than Xaminer. To test our project in a lightweight environment, without having to create test exams in Xaminer, we built a demo frontend.

The demo frontend is a React application built using Next.js [22]. Next.js is a full-stack, i.e., frontend and backend, web framework powered by React [23]. It allows us to define routes using the filesystem, i.e., each file in the `/pages` folder becomes a page with the same url as the files path within the `/pages` folder. Moreover, Next.js can also run backend code so we can emulate the backend parts we would expect from Xaminer.

The goal of the demo frontend is to make it easy for anyone to test our project in an environment that feels representative of Xaminer. When a user visits the demo frontend, they are asked if they want to create or join a room. When they click to join a room, they are prompted to enter the six digit room code. When they click to create a room, they receive a URL which they can use to join it. Before finally joining a room, the user can decide if they want to be a student or a supervisor. Using the join URL, multiple clients can join the same room with the role they desire.

For the students, we display a page that represents an exam, with our student widget in the top right corner. For the supervisors, we display a list of the video streams of all students who are connected to the same exam room. We also embed the management widget on the top of the page.

To render the widget, the demo frontend uses a very similar approach to the one used to embed the components in Xaminer's Vue.js frontend. React has references to the native DOM nodes as well. We use those to render the widgets into the React app. For authentication, we use the same token-based authentication system that we use with Xaminer. However, we disabled the integration to Xaminer's authentication system.

The demo frontend not only proved vital during the development of the frontend application, it is also used to serve the underlying web pages for the end-to-end tests.

5.6 Integrating with Xaminer

Xaminer is the existing exam tool which we want to integrate our video calling functionality into. Xaminer's frontend is a Vue.js application. We explained how we render our widgets into the Vue.js app Section 5.2, however due to the very flexible nature of our video calling frontend, there are some parts which we need to adapt to the page structure and data that Xaminer has, e.g., mapping examIds to roomIds and studentIds to clientIds. Instead of editing Xaminer's current code or changing our frontend, we built a stub script which adapts from Xaminer to our frontend, i.e., a JavaScript file, which implements functions which take the data that Xaminer has and perform the processing and HTTP requests required to transform it into the data that our frontend requires, after that transformation is done, the stub script calls the required methods exposed by our frontend. We opted for this architecture for the following reasons:

[22] Next.js `https://nextjs.org/`
[23] React `https://react.dev`

1. Integrating our React code directly into Xaminer, without our stub script, would have required some major changes to Xaminer. We want to remain on the safe side and only change a running system in minimal ways.

2. Even though Xaminer's bundle size is rather large, we still want to adhere to our philosophy of shipping a small bundle and not blocking the exam rendering. To achieve this, we are loading our frontend bundle asynchronously in the stub script.

3. For our frontend, some actions need to be performed after the other, i.e., we need to retrieve an authentication token before we can connect to a room. Moreover, some functions should only be called after the call is initialized, i.e., we need to initialize the call before rendering the UI widgets. Our frontend exposes APIs to check if certain parts are ready but integrating that directly in Xaminer would have required some additional changes. Instead, our stub script transparently handles the batching and waits until the required parts are ready before performing an action. It also mapps the data available in Xaminer to the data our system expects.

With this stub script, we only needed to add a few lines of code to Xaminer, without changing any of its structure, state or anything else as showcased in Listing 4. Yet, Xaminer did not have a way of determining the group of students who participate in the exam. Since our system needs this information, we added an endpoint to Xaminer's API which returns the group number a given student is assigned to in a given exam. We kept changes to Xaminer to an absolute minimum and ensured that our video calling system can also be used with other projects.

Listing 4: Initializaton of our stub script for a supervisor

```
let initializeXaminer = async () => {
  let key = ...
  let group = ...
  let exam = await fetch('/api/exam/...')

  // Initialize the video call for a supervisor
  window.callStub.startPrimary({
    xaminerAuthenticationKey: key,
    examId: exam.key,
    groupId: this.group.key
  });

  // Render the management UI widget
  window.xamCall.renderManagementUI(this.managementEl);
};
```

6 Usage and Evaluation

In this section we will examine the user interface of our project. As there are essentially two types of users, we will do this examination for both students and supervisors separately.

6.1 Students

We need to remember that students can only take minimal actions. They should be focused on the exam, not the video calling software. This aspect was crucial in the design of the student UI.

When students open an exam page with video calling enabled, they need to provide us with the permission to access their microphone and, depending on the settings selected by the supervisor, their camera and/or screen. If a student does not want to share their media devices, supervisors will not be able to grade their exam. Hence, it is important to educate the students about this rule.

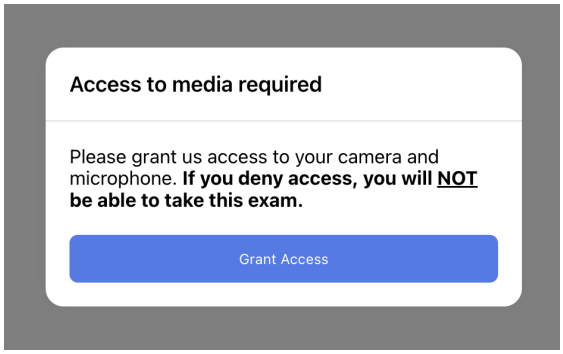


Figure 21: Media permission modal for students.

As shown in Figure 21, before launching the browser's native permission prompts, we show a dialog telling the students that they will be asked to give us access to their media devices. When they click *Grant Access*, the browser asks them for the permissions.

Even after the initial warning, students might still block access. When that happens, the browser notifies our frontend about it. In that case, we open a dialog telling the users that they need to go in their browser's settings and enable media access to participate in the exam. However, we opted not to lock students out of the exam completely since some students might have special agreements with their supervisors. Nevertheless, the supervisor is informed about the fact that a student declined to grant access to their media streams. In a future extension of the project, a setting could be added to make this behaviour customizable by the supervisor.

After our system has been granted the required permissions, the student can see their exam and our student video widget appears in the top-right corner. To make this pleasant to look at, we animate the widget to fade in from the top.

After all students have loaded the page and granted their permissions, the supervisors will start the ID check. Current of-the-shelf solutions, such as Zoom, only show a small preview of the user's own video. For ID checks it is better for students to see their own camera in full screen to make sure their student ID is clearly visible and that there are no reflections on it. Once the

supervisor has ended the ID check and marked the student as participating on their list, the big camera preview is closed and the exam is visible again.

When the ID check is done and the exam has started, students might want to ask questions. In the current Zoom-based workflow, questions are asked using the text chat, so other students are not disturbed. In our project, we implemented a custom question room feature. This allows students to enter one-on-one sessions to talk to their supervisors. This way, questions can be resolved quickly and other students remain focused on their work. To start a question room, students can click the *Ask Question* button, as shown in Figure 22

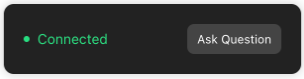


Figure 22: Connection widget for students.

After a student clicks on *Ask Question*, a dialog opens up informing the student about how question rooms work. In this dialog they can confirm that they want to ask the question or abort. When a student decides to ask the question, the supervisors will be notified. Supervisors now have to accept the question. While the student waits, the widget in the top-right updates to show that they are waiting for a supervisor as shown in Figure 23. Even before the question has been accepted, they can click on *Resolve Question* if they don't need assistance anymore.

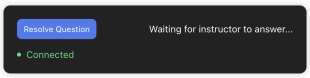


Figure 23: Student connection widget with an active question.

After a supervisor accepts the question, the one-on-one call will start. To make sure the student is not confused when the call starts, we again show a dialog. The widget in the top-right corner also updates to reflect that they are currently in a one-on-one call with a supervisor. The question can be resolved by the student or the supervisor by clicking on *Resolve Question*.

Even after we have been granted access to the student's media streams, it is still possible for the permissions to be revoked or the streams to fail. If this occurs, we try to access them again. If this is not possible, we update the UI to reflect that a media access error occurred. In future revisions, media access problems should be logged and reported to the supervisor. Possibly an integrated chat functionality could be added so the student and the supervisor can find a solution.

6.2 Supervisors

Now that we have an understanding of how the UI works on the student's side, we can explore the supervisor's side. Supervisors not only need to monitor the students, they also need to configure the settings of the exam and access the logs. It is important to emphasise that all parts of the supervisor's admin controls update in real time, even if multiple supervisors access the settings at the same time.

Our video calling system is integrated into all exams on Xaminer but some lecturers might want to keep using Zoom or a different video calling software for their exam. To achieve this, our system can be enabled or disabled. When disabled, we will hide the UI on the student's side and do not ask for any permissions.

On the supervisor's side, we will not show the settings widget and instead present them with a short description of our project and a toggle to *Enable Integrated Video Call Functionality* as shown in Figure 25. When a supervisor chooses to enable video calling, we start a connection to LiveKit and render the settings widget.

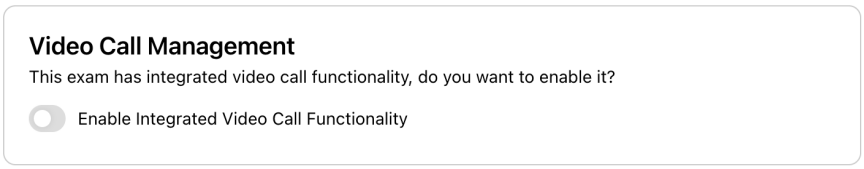


Figure 24: Management widget of an exam where our system is currently disabled.

The management UI is intentionally minimal as to not distract the supervisors from their duties. It is structured into six sections as shown in Figure 25.

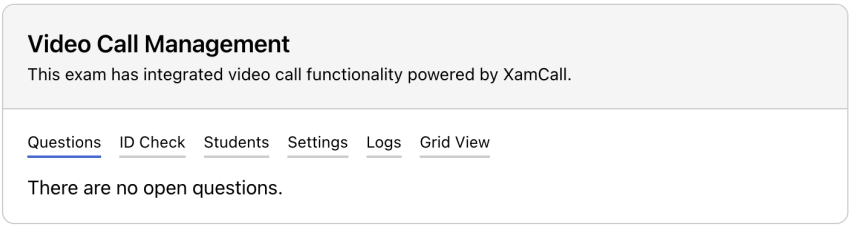


Figure 25: Exam management widget for supervisors.

1. The first one is for *Questions*. When there are unanswered questions, they will appear here. If a supervisor clicks on the *Answer* button next to a question, they will join a one-on-one call with the student. The questions are ordered from longest wait time to shortest. If a student has been waiting for more than five minutes the question turns red. Every time a student asks a question, the supervisors is notified about it by a short sound signal. This way, supervisors know when to look at the list of questions. A good future improvement would be if an indicator, i.e., a toast, appears when there are students waiting. When a supervisor answers a questions, the full screen video UI will open up. This UI shows a large preview of the student's video streams. A control menu is rendered on the bottom of the screen. Supervisors can mute themselves or the student or resolve the question, using the control menu. It also shows the student ID so the supervisors know who they are talking to.
2. The *ID check* tab tells a supervisor how many student's IDs have not been checked. When they click on *Start ID Check*, the ID check UI will open up for all unchecked students.

Importantly, student's whose IDs have already been checked will not see this UI. This behavior is useful if a student joins after the exam has already started, since we don't want to interrupt the other students.

When the ID check is started, the supervisor sees a two-by-two grid with the video feeds of the students whose IDs have not been checked yet as depicted in Figure 26. Traditionally, supervisors would tick off students on a sheet of paper. With our project, student can be ticked of directly in the browser.

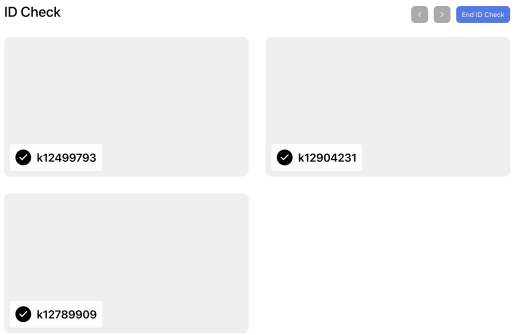


Figure 26: ID check UI shown during an ongoing ID check.

When a supervisor clicks on the checkmark next to student ID, the checkmark turns green. Notably, this is synchronized to all other supervisors, i.e., if one supervisor ticks off a student, they will appear as checked for all other supervisors. When all students are checked, the ID check can be ended by clicking on the *End ID Check* button. After this, the ID check UI will disappear for the students and the supervisors.

After student's IDs have been checked, a report can be downloaded as a CSV file. When more students join after the ID check, the UI will update to show that there are new unchecked students and the ID check can be started again.

3. The *Students* tab shows a list of all students, who are currently connected, with their student ID and the time when they joined. This list updates in real-time.

4. The *Settings* tab allows the supervisors to customize certain aspects of the video calling room, i.e., the media streams that must be shared and whether question rooms are enabled. These settings can only be changed while there are no students connected.

   The *Audio* option is always turned on and cannot be turned of since microphone streams must be enabled for all exams. Supervisors can choose if *Camera* or *Screen* access is required. Both options can be enabled at the same time to receive a student's camera stream and screen share. Moreover, it is possible to disable the ability for students to ask questions. On this page, the video calling functionality can also be disabled altogether. Clicking on *Disable XamCall* would bring the supervisor back the the enable UI. These settings can only be changed when there are no students currently connected.

5. On the *Logs* tab, the supervisor sees a list of all recent log entries and when they were created. This list can be downloaded as a CSV file by clicking *Download CSV*. Next to the download button we inform the supervisor that logs will only be stored for seven days. https://www.united-domains.de/qa-domain/

6. The *Grid View* tab opens up a UI which displays the student's video streams as a grid. Supervisors can navigate through multiple pages of students.

In addition to the management widget, the supervisors also have a floating connection widget, similar to the one the students have. It is located in the bottom-right corner off the screen. Supervisors can use the widget to mute themselves and to monitor their connection status.

Xaminer currently displays a table with a list of all students assigned to the exam as shown in Figure 27. It was a requirement for this project to embed a video feed next to each student's information in this table. This embedded widget displays the student's camera and screen share streams. The supervisor can click on *Mute* to mute the student's microphone. In this case, the student will only be muted for the supervisor who mutes them, not for the other ones. If a supervisor notices some suspicious activity, they can click on *Maximize* to open the student's video streams in full screen.

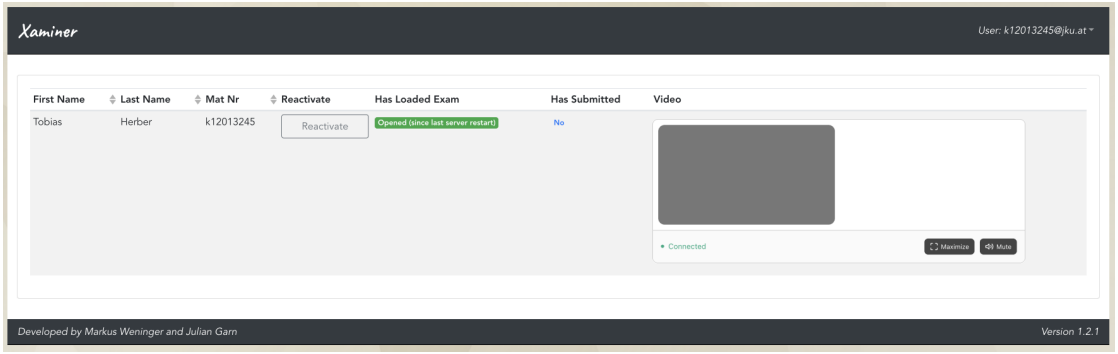


Figure 27: Xaminer supervisor page with a student's video stream.

7 Related Work

In this section, we will examine some other tools and services and how they handle real-time media streaming. Moreover, we will examine whether and how they could be used instead of our implementation.

Zoom Since Zoom is a proprietary software solution there is little information about it. At the time of writing, Zoom supports up to 1000 members in a single room [13]. This is a big competitive advantage and they understandably don't want to share too much about their systems. Zoom wrote a blog post summarizing their infrastructure in simple terms [12]. We will use this to, as much as possible, compare Zoom to our system. Zoom is based on a "Distributed architecture" this means that they can use different servers for streams in the same meeting. This is necessary for a large provider such as Zoom. However, our requirements are rather limited and hence we don't need multiple servers. Zoom uses what they call *Multimedia routing* to distribute streams. They say that it is not MCU. From how they describe it, it sounds like a more sophisticated version of SFU. LiveKit also uses SFU. Zoom supports Multi-bitrate encoding which is similar to LiveKit's Simulcast. Simulcast is the technology we use to dynamically send different video streams depending on the video element's size. While our system is obviously much simple than Zoom, their system has some noticeable similarities to ours. This confirms us in the choices we made.

Building our own SFU or MCU system While this certainly would have been an interesting endeavor, it would have far exceeded the scope of this thesis. Moreover, implementing something ourselves is not always the best decision, especially when it comes to a real production systems. LiveKit is not only an open source video streaming software, but also a company [4]. They state that they have thousands of customers, including some large corporations, such as Spotify and Typeform. This means that their system is heavily tested in the real world. Moreover, LiveKit being a company means that they need to make a profit with their product. To be able to sell such a system, a certain level of service and quality is required. If we built our own solution, that certainly would have not been to the standard that LiveKit offers.

Embedding Jitsi Jitsi is an open source video calling software [3]. It aims to be the open source equivalent to Zoom or Google Meet. Jitsi calls can be embedded onto any site. Choosing this approach would have made this thesis obsolete. However, it is safe to say that, with Jitsi, the feature set we have built would have not been possible. Jitsi does not have support for questions rooms or ID checks. Jitsi also does not support embedding individual video streams in a table.

Using a hosted solution Companies, such as Twilio and Agora, offer fully hosted video calling as a service solutions [11] [1]. They are similar products to LiveKit, but not open source. This means that we would be locked in and had to pay a service fee each month. Such a solution might be great for startups and companies wanting to keep their costs low and who don't want to manage their own servers. Since this is a university project, we have free access to high-performance servers. Hence, a hosted solution does not make sense for us.

8 Conclusion and Future Work

We have built a video calling tool specifically for exams. It reduces friction by being integrated into the existing Xaminer exam tool. We built administration tools, e.g., a grid view and logging right into our tool. Since we focused on exams, standard interactions, such as one-on-one question rooms and ID checks are integrated into our tool.

Frontend Since we built our UI using widgets, it was very easy to integrate into Xaminer. While choosing Preact instead of React was not necessary, the framework served us well. Using Socket.IO for WebSockets made building real-time interactions especially easy. Generally speaking, our technology stack served us very well and we have no regrets.

Backend Choosing Warp and Socket.IO to build both a REST API and a real-time interface made sense and served us well. Through the use of Node.js, the server was easy to deploy and is decently performant. The biggest regret is not using LiveKit from the start. On the other hand, building P2P video calling manually helped us learn and understand the interactions required to stream video in the browser.

In addition to the video calling functionality, we also built a great framework for reactive data. It uses a backend server as a single source of truth. Each client generates its own snapshot of data from that and updates it in real-time. All of that is enabled through the use of WebSockets and our permission scheme with primary and standard clients.

Future Enhancements The tool we have built is not only well suited for Xaminer, but for all other kinds of learning and examination tools. Prospective future iterations could be:

- Adding the ability to change settings while students are connected.
- Replacing LevelDB with a feature-rich database, such as MySQL or PostgreSQL.
- Storing logs in a database and not in memory.
- Adding our video calling functionality to more tools, such as Moodle.

List of Figures

1 A connection status widget of a supervisor currently connected to a room. 13
2 A settings widget for an ongoing exam. 13
3 A student video widget with a camera stream. 13
4 The WebRTC connection structure between multiple clients of different types in the same room. 14
5 Opening a WebRTC connection using STUN for NAT traversal. 15
6 The flow of exchanging ICE information between two clients with a shared application server. 15
7 The connection structure between clients connected using P2P. 16
8 The connection structure between clients connected using an SFU. 18
9 The connection structure between clients connected using an MCU. 19
10 The handshake required to establish a connection using LiveKit. 20
11 The communication flow when a client asks a question. 26
12 WebRTC connection setup handshake using P2P. 27
13 A student widget which shows that the student is connected 31
14 A student's view while the ID check is being performed 31
15 A student widget which shows that the student is connected 31
16 A supervisor connection widget. 32
17 A supervisor settings widget. 32
18 A student's video feed for a supervisor. 32
19 Embedding a Preact app in a Vue.js component. 33
20 Embedding a Preact application in a Vue.js component. 34
21 Media permission modal for students. 40
22 Connection widget for students. 41
23 Student connection widget with an active question. 41
24 Management widget of an exam where our system is currently disabled. 42
25 Exam management widget for supervisors. 42
26 ID check UI shown during an ongoing ID check. 43
27 Xaminer supervisor page with a student's video stream. 44

Listings

1 `getRoom` API endpoint handler function 25
2 `Caddyfile` configuration for our Caddy server 29
3 A Vue component which renders a Preact application into itself 33
4 Initializaton of our stub script for a supervisor 38

References

[1] Agora. [Accessed 25-Apr-2023].

[2] Can I Use — WebSocket API. https://caniuse.com/mdn-api_websocket. [Accessed 25-Apr-2023].

[3] Jitsi. [Accessed 25-Apr-2023].

[4] LiveKit — About. [Accessed 25-Apr-2023].

[5] node-gyp — Node.js Native Addon Build Tool. https://github.com/nodejs/node-gyp. [Accessed 25-Apr-2023].

[6] Preact@10.13.1 Bundle Size. https://bundlephobia.com/package/preact@10.13.1. [Accessed 25-Apr-2023].

[7] react-dom@18.2.0 Bundle Size. https://bundlephobia.com/package/react-dom@18.2.0. [Accessed 25-Apr-2023].

[8] react@18.2.0 Bundle Size. https://bundlephobia.com/package/react@18.2.0. [Accessed 25-Apr-2023].

[9] The V8 JavaScript Engine — nodejs.dev. https://nodejs.dev/en/learn/the-v8-javascript-engine/. [Accessed 25-Apr-2023].

[10] The WebSocket Protocol. https://tools.ietf.org/html/rfc6455. [Accessed 25-Apr-2023].

[11] Twilio. [Accessed 25-Apr-2023].

[12] Zoom Blog — Here's How Zoom Provides Industry-Leading Video Capacity. https://blog.zoom.us/zoom-can-provide-increase-industry-leading-video-capacity/. [Accessed 25-Apr-2023].

[13] Zoom Support — Hosting Large Meetings. https://support.zoom.us/hc/en-us/articles/201362823. [Accessed 25-Apr-2023].