ChatStream: Real-Time Livestream Chat System Using WebRTC, ActionCable, and WebSockets

Tatiya Seetharatkul
Department of Computer Science
and Information Management
Asian Institute of Technology
st124875@ait.asia

***Abstract*—Seamless real-time communication is essential for university platforms to support collaboration and live events. Many existing tools lack academic authentication and are not optimized for low-latency, browser-based usage. To address this, ChatStream was developed as a secure, scalable livestream chat system using Ruby on Rails with a PostgreSQL backend. At the Application Layer, WebRTC handles video streaming and WebSockets enable real-time messaging. UDP and TCP are used at the Transport Layer for video and chat, respectively, with IPv4 routing at the Network Layer. Authentication is managed with Devise, and ActionCable powers real-time updates. Each chat room supports one streamer broadcasting to authenticated viewers, who can interact instantly.**

***Index Terms*—WebRTC, WebSocket, ActionCable, TCP, UDP**

I. Introduction

The increasing reliance on browser-based communication tools has emphasized the role of real-time livestreaming technologies in contemporary education. As universities adopt hybrid learning models, the demand for low-latency video and chat systems that operate efficiently across a wide range of devices and networks has grown significantly.

This study examines the development of a university-oriented livestream chat platform through the framework of the Internet Protocol Suite, which categorizes communication functions into five abstract layers. At the **Application Layer**, protocols such as WebRTC and WebSockets facilitate video transmission and real-time messaging within web browsers. The **Transport Layer** ensures end-to-end data delivery, employing UDP for latency-sensitive media streaming and TCP for reliable message transmission. The **Network Layer** governs packet routing between distributed systems, utilizing the Internet Protocol (IPv4). The **Data Link Layer** is responsible for local frame delivery over physical media, such as Ethernet or Wi-Fi. Finally, the **Physical Layer** transmits raw binary data through hardware components, including network cables, routers, and wireless transceivers.

A thorough understanding of data transmission across these layers is essential for the design, debugging, and optimization of responsive and robust web-based communication systems. This layered perspective has guided the technical decisions made throughout the development process.

This project was developed as part of the AT70.15 Advanced Topics in Internet Technology course at the Asian Institute of Technology for the January 2025 semester.

II. Objective

The objective of this project is to design and implement a browser-based livestream chat system tailored for academic use. The system aims to enable real-time, one-way video broadcasting from a designated streamer to multiple authenticated viewers, accompanied by interactive text-based messaging. Emphasis is placed on ensuring secure access control, low-latency performance, and seamless user experience across modern web browsers.

By integrating WebRTC for media transmission, ActionCable for real-time messaging, and Devise for authentication, the project seeks to demonstrate how contemporary web technologies can be orchestrated to support scalable and responsive communication within a university environment. The system also serves as a practical application of the five-layer Internet Protocol model, reinforcing concepts of protocol interaction, network architecture, and end-to-end data delivery.

III. Target Users

The primary target users of this system are university students, faculty members, and academic staff who require a lightweight, browser-based platform for one-way livestreaming and real-time chat. Designed specifically with academic environments in mind, the system facilitates interactive lecture delivery, seminar broadcasting, and virtual study sessions, particularly in scenarios where low-latency video and responsive communication are essential.

Students can join chat rooms as authenticated viewers, receive real-time video from the designated streamer (e.g., a lecturer), and actively participate in discussions through the integrated chat interface. Faculty members or teaching assistants can serve as streamers, using the system to broadcast lectures or announcements without requiring external software installations.

By prioritizing ease of access, authentication, and low resource consumption, the platform is also suitable for use in resource-constrained settings, such as small classrooms, departmental webinars, or collaborative group meetings. All that is required to participate is a modern web browser, making the system inclusive and platform-independent.

IV. Literature Review

Real-time communication systems have evolved significantly, progressing from basic client-server models to modern, browser-native peer-to-peer communication frameworks. A foundational technology in this field is WebRTC, which enables low-latency audio and video transmission directly between users. According to Loreto and Jennings [1], WebRTC provides built-in support for NAT traversal using protocols such as ICE, STUN, and TURN, making it highly effective for real-time communication over constrained networks.

For backend development, Ruby on Rails is widely used due to its support for the Model-View-Controller (MVC) architecture and its integration with ActionCable for real-time communication. ActionCable allows developers to implement publish-subscribe patterns using WebSocket underpinnings, enabling responsive chat features and live data streams [2].

Containerization through Docker has transformed deployment practices in software development. As Merkel highlighted, Docker offers lightweight Linux containers that promote consistent deployment environments, reduce compatibility issues, and facilitate scalability [3]. These advantages are especially relevant in systems requiring real-time responsiveness and minimal latency.

The architecture of such systems often combines HTTP/HTTPS for serving static assets and WebSocket for persistent, bidirectional updates. This hybrid pattern is aligned with the principles of network-based software architecture as discussed by Fielding, who emphasized RESTful design and layered system interactions to enhance scalability and performance [4].

Security considerations are also crucial. Modern web browsers require secure contexts for accessing media devices. Mozilla [9], W3C [10], and Chromium [11] all enforce HTTPS for APIs like `getUserMedia()`, reinforcing the importance of encrypted transport even in local development. Local testing across devices on the same LAN, using IP addresses instead of `localhost`, helped validate these constraints while confirming functionality across OSI layers below the network level.

Several existing platforms and studies have explored similar applications of WebRTC and real-time communication. BigBlueButton, an open-source web conferencing system designed for online education, incorporates WebRTC and messaging to support virtual classrooms [5]. Jitsi Meet is another open-source platform that offers secure video conferencing using peer-to-peer WebRTC and signaling over XMPP or WebSocket [6]. These systems validate the feasibility of browser-based streaming with scalable backends and highlight design decisions such as TURN fallback and bandwidth adaptation.

From a research perspective, Jain and Gupta [7] demonstrated the development of a real-time video conferencing platform using WebRTC, detailing architecture layers, signaling methods, and performance optimizations. Their study supports the architectural choices made in this project, particularly the integration of WebSocket for signaling and UDP for media transport. Similarly, the Google WebRTC team has released best practices and design guidance for implementing secure and efficient real-time media applications in the browser [8].

This literature collectively informs the architecture, protocols, and design principles behind the implemented system, highlighting the importance of secure transport, peer-to-peer optimization, and modular backend development.

V. System Overview

The ChatStream system is structured around a centralized Rails backend that coordinates real-time video streaming and chat communication between a designated streamer and multiple viewers. When a chat room is created, only the room's owner is allowed to initiate a livestream using the WebRTC API. The media stream is transmitted directly from the streamer's browser to viewers via peer-to-peer WebRTC connections.

Signaling for the WebRTC session, including SDP offers/answers and ICE candidate exchange is handled through WebSocket channels powered by ActionCable, enabling low-latency message passing. Meanwhile, authenticated users can send and receive chat messages in real time within the same WebSocket infrastructure.

The frontend is rendered using Rails views with embedded JavaScript, ensuring a tightly coupled monolithic experience. Bootstrap is used for layout and responsiveness, and JavaScript handles UI interactivity and media control. Docker is used to containerize the entire stack, including PostgreSQL, Redis, and the Rails application, ensuring consistent behavior across development environments.

A simplified system workflow involves the streamer clicking a 'Start Streaming' button, triggering media capture and peer connection setup. Viewers who join the room automatically establish WebRTC connections and begin receiving the video stream while participating in the integrated chat interface.

VI. System Architecture

The system is implemented using Ruby on Rails as the backend framework, and HTML, CSS (utilizing the Bootstrap framework), and JavaScript for frontend development. WebRTC is employed to facilitate real-time video streaming, while ActionCable supports WebSocket-based communication, enabling interactive chat functionality and signaling between users.

A. Key Technologies

The system leverages a combination of modern web technologies to support real-time streaming, messaging, and user interaction. At its core, **Ruby on Rails** serves as the primary backend framework, adopting the Model-View-Controller (MVC) architectural pattern. It manages HTTP routing, abstracts database interactions via ActiveRecord, and integrates with ActionCable to enable WebSocket-based communication.

On the frontend, the user interface is constructed using a combination of **HTML**, **CSS** (with Bootstrap for responsive design), and **JavaScript**. JavaScript plays a critical role in

client-side interactivity, particularly in initializing WebRTC sessions and dynamically updating the chat interface during live interactions.

WebRTC is employed to enable peer-to-peer video streaming from a designated streamer to one or more viewers. Internally, WebRTC utilizes both **UDP** which is primarily for NAT traversal and media delivery through the use of ICE/STUN and **TCP** as a fallback transport protocol to ensure connectivity under varying network conditions.

ActionCable, a WebSocket abstraction provided by the Rails framework, facilitates real-time bidirectional communication between clients and the server. It is used to handle signaling for WebRTC connections as well as the transmission of chat messages. Underlying this, **WebSocket** technology maintains low-latency, persistent connections over **TCP**, allowing for efficient, real-time message exchange and session coordination.

Lastly, **Docker** is used to containerize the entire application stack, including all services and dependencies. This containerization ensures consistency across development and production environments, streamlines setup and deployment processes, and provides isolated execution of services for enhanced portability and reliability.

B. Database Design

The system's data layer comprises several relational tables that manage user authentication, room configurations, and real-time messaging. The core tables are described as follows:

The **users** table is responsible for storing user credentials and related metadata. Authentication is managed through the Devise gem, which ensures secure handling of login and session management. Key attributes of this table include the `id`, which serves as the primary key, and the `email`, which uniquely identifies each user in the system. The `encrypted_password` field stores a securely hashed version of the user's password to maintain confidentiality. Additionally, Rails-generated timestamps `created_at` and `updated_at` are included to record when each user record is created and last modified.

The **chat_rooms** table represents individual streaming or messaging spaces within the application. Each record in this table contains an `id` as its unique identifier, a `name` representing the room's display name, and a `streamer_id`, which acts as a foreign key referencing the user who owns or streams in the room. This association forms a one-to-many relationship between users and chat rooms, where a single user can create multiple chat rooms, but each chat room is associated with exactly one user. Timestamp fields `created_at` and `updated_at` are included for tracking changes to room records. The overall relationship structure is illustrated in Figure 1.

The **messages** table stores all chat messages exchanged within the platform's chat rooms. Each message record includes an `id` as the primary key and a `content` field to store the textual body of the message. It also contains two foreign keys: `chat_room_id`, which links the message to a specific chat room, and `user_id`, which identifies the user who sent the message. Standard Rails timestamps `created_at` and `updated_at` are used to track message creation and modification. This structure enforces a many-to-one relationship in both directions: a single chat room can contain multiple messages, and a user can send many messages across different rooms. These relationships are also depicted in Figure 1.

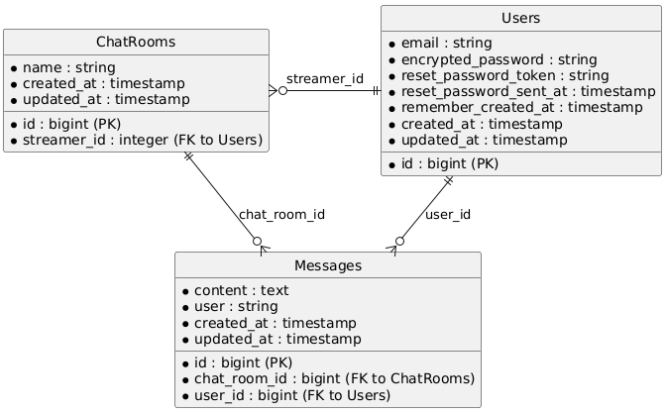

Fig. 1. Entity-Relationship Diagram of the system.

During the early stages of development, the system was designed to support two-way video streaming, where both the streamer and viewers could transmit media. To accommodate this, a join table named `chat_room_users` was introduced to model a many-to-many relationship between users and chat rooms. However, the design was later simplified to a one-way streaming architecture, in which only the designated streamer transmits video. Consequently, the `chat_room_users` table remains unused in the current implementation, as user participation is now implicitly tracked through message associations and stream ownership.

C. Controllers

The system adheres to the Model-View-Controller (MVC) architecture provided by Ruby on Rails. In this design, controllers serve as the intermediaries that coordinate user interactions, enforce access control, and prepare data to be rendered by views.

The **ChatRoomsController** is responsible for managing the lifecycle of chat rooms, including listing, creation, editing, updating, and deletion. Access to all actions is restricted to authenticated users, and only the room owner (designated as the streamer) is permitted to modify or delete the room. The `index` action is used to retrieve and display all available chat rooms, while the `show` action renders the livestreaming interface, which includes both video streaming and real-time chat. Creation, updating, and deletion of rooms are handled by the `create`, `update`, and `destroy` actions, respectively. Access control is enforced through the `authorize_owner!` method to ensure that only the designated owner can make changes. Additionally, the controller implements a `handle_not_found` method to gracefully handle exceptions such as invalid room access.

The **MessagesController** facilitates the management of real-time messaging within individual chat rooms. The `create` action is used to associate a new message with both the current user and the corresponding chat room. Upon successful creation, the message is broadcast using Turbo Stream to ensure immediate updates on the frontend. While standard RESTful actions such as `index`, `show`, `edit`, `update`, and `destroy` are also defined, the majority of interactions occur dynamically through WebSocket-based communication. Authentication is strictly enforced to ensure that only verified users are permitted to send or manage messages.

The **HomeController** provides a single `index` action, which is responsible for rendering the default landing page of the application. It typically serves as an entry point to the system and may include static content or navigational elements for the user interface.

VII. Methodology

The development of the livestream chat platform followed an incremental, modular approach using Ruby on Rails as the backend framework and JavaScript with WebRTC for real-time video communication. The application was containerized using Docker to ensure consistency across development and deployment environments.

A. Environment Setup

The project environment was initialized by setting up a PostgreSQL database and configuring access through PgAdmin. Docker Compose was used to orchestrate the necessary services, including Rails, Redis, and PostgreSQL, ensuring a consistent and reproducible environment across development and production.

B. User Authentication

User authentication was implemented using the Devise gem. Routes for sign-up, login, and logout were configured to manage user sessions securely. Access control was enforced using controller filters, allowing only authenticated users to join chat rooms or interact with streaming features.

C. Backend Implementation

The backend was built using Ruby on Rails, leveraging the Model-View-Controller (MVC) architecture. A 'chat_rooms' resource was created to manage individual chat rooms, each associated with a specific user (the streamer). Users could create, view, and delete their own rooms. The messaging system was implemented using a 'messages' table, with ActionCable configured via Redis to support real-time bidirectional messaging. Turbo Stream was used to dynamically update chat interfaces without requiring full page reloads. For video streaming, the 'getUserMedia()' API was used to capture the streamer's media input, which was transmitted to viewers via WebRTC peer connections. ICE, STUN, and TURN protocols were utilized for NAT traversal and peer discovery. A custom signaling system built with ActionCable managed the exchange of WebRTC session descriptions and ICE candidates.

D. Frontend Implementation

The frontend interface was built using HTML, CSS with Bootstrap for responsive design, and JavaScript for client-side interactivity. JavaScript was responsible for initiating WebRTC sessions, capturing media, and dynamically updating the UI during live interactions. Viewers received the streamer's video feed rendered on a 'canvas' element, which was sourced from a hidden 'video' tag. Real-time chat messaging was enhanced with JavaScript MutationObservers to handle dynamic message appending and auto-scrolling. Stimulus controllers were used to streamline DOM updates and interactivity. The interface was further refined with features such as the 'Start Streaming' button, Enter-to-send message support, and consistent visual feedback for user actions.

E. Testing and Validation

The system was thoroughly tested within a local area network (LAN) environment. Multiple users accessed the application using IP addresses over the same Wi-Fi network to validate real-time video streaming and chat functionalities. Testing focused on ensuring reliable peer-to-peer media delivery, seamless chat updates, and correct enforcement of authentication and room ownership. Edge cases such as unauthorized access, incorrect user roles, message truncation, and reinitialization of video streams were also examined to confirm the system's robustness and responsiveness in local deployment conditions.

VIII. Network Layer and Protocol Stack

The system architecture adheres to the layered structure of the OSI model, integrating a range of protocols and technologies to support real-time communication, signaling, and multimedia transport.

At the **Application layer**, several components play key roles. HTTP and HTTPS are used for serving static assets and initiating client-server interactions. WebSocket technology enables full-duplex, low-latency connections by upgrading HTTP(S) sessions to persistent TCP connections, thereby facilitating real-time signaling and chat features. Within the Ruby on Rails framework, ActionCable extends WebSocket functionality by managing channels and broadcasting messages, which is instrumental in enabling live chat capabilities. For peer-to-peer video streaming, the system employs WebRTC, which encapsulates several sub-protocols: the Session Description Protocol (SDP) is responsible for negotiating media capabilities such as codecs and bitrate; Interactive Connectivity Establishment (ICE) coordinates the candidate selection process for forming peer-to-peer links; STUN (Session Traversal Utilities for NAT) assists in discovering public-facing IP addresses; and TURN (Traversal Using Relays around NAT) provides a relay fallback mechanism when direct connections are not feasible.

At the **Transport layer**, both TCP and UDP are employed to serve distinct purposes. TCP is leveraged by WebSocket and ActionCable to ensure reliable, ordered, and error-checked data transmission between clients and the backend. In contrast,

UDP is preferred by WebRTC for transmitting real-time audio and video due to its lower latency. WebRTC further enhances UDP transmission with optional Forward Error Correction (FEC) and retransmission strategies to mitigate packet loss.

Moving to the **Network layer**, the system relies on IPv4 addressing to route traffic across local and public networks. During local development, devices communicate over private IP ranges (e.g., 192.168.x.x), whereas production environments utilize public-facing IP addresses provided through services like Ngrok tunnels. ICE candidate gathering plays a critical role in identifying the best possible connection paths by collecting local, reflexive, and relay IP candidates.

At the **Link layer**, communication is conducted via Wi-Fi or Ethernet interfaces. During the development and evaluation stages, the system was tested extensively on a shared Wi-Fi network, where laptops and mobile devices accessed the platform to validate livestream and chat features. This layer is responsible for frame-level transmission within the local area network (LAN). MAC-level addressing facilitates communication between devices before packets are routed using IP, particularly in scenarios where users access the system using the server's LAN IP rather than localhost. These local tests demonstrated the application's reliability and consistency across different devices on the same network without relying on external internet routing.

Lastly, at the **Physical layer**, the system depends on standard transmission media such as Wi-Fi radio waves or Ethernet cables to transmit raw binary data between connected devices. While this layer forms the foundation for all higher-level communication, the project did not involve direct development or manipulation of the physical layer. Its functionality is abstracted and managed by underlying hardware components and operating system-level drivers, making it an invisible yet essential part of the system infrastructure.

IX. Results

The system provides a fully functional livestream and chat application, implemented through various view templates under the Rails 'app/views/' directory. Each view corresponds to a specific user interface or real-time functionality within the platform.

A. Chat Room Views

The 'chat_rooms/' folder contains views related to room management and video streaming:

- **index.html.erb**: Displays a list of all chat rooms (Fig. 2).
- **show.html.erb**: Main livestreaming interface that renders video and chat (Fig. 3).
- **new.html.erb**: Forms for creating rooms (Fig. 4).
- **edit.html.erb**: Forms for updating rooms (Fig. 5).

B. Message Views

The 'messages/' folder does not contain standalone views; instead, it provides a partial form used for real-time message input within the livestreaming interface. This form is rendered directly in the 'chat_rooms/show.html.erb' view, allowing users to submit messages seamlessly without page reloads.

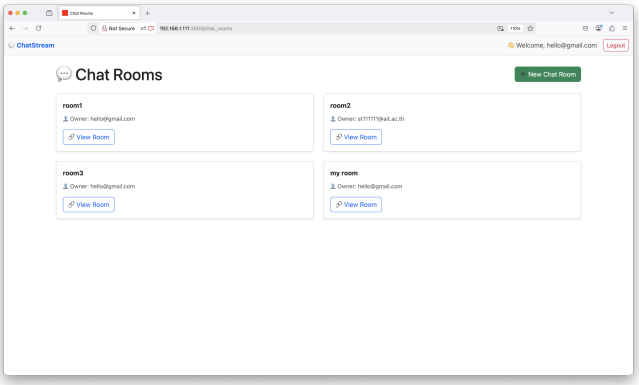

Fig. 2. chat_rooms#index view.

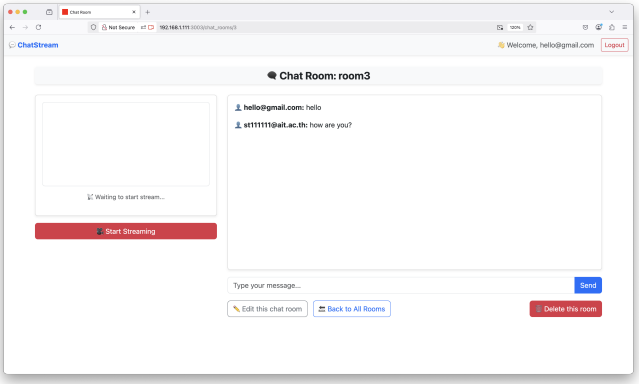

Fig. 3. chat_rooms#show view.

C. Authentication Views

The 'devise/' directory handles user registration, login, password recovery, and session management. Each subfolder corresponds to a Devise module.

- **devise/sessions/new.html.erb**: Manages the Login funcionality (Fig. 6).
- **devise/registrations/new.html.erb**: Manages the Signup funcionality (Fig. 7).

D. Home Views

- **home/index.html.erb**: Default homepage with general UI and navigation (Figure 8).

This modular organization ensures that each component—streaming, chat, and authentication—is clearly separated and maintainable. It also supports future extensibility, such as adding features for screen sharing, multi-user streaming, or event-based chat moderation.

X. Discussion

The system was tested in real-time within a local area network (LAN) environment. WebRTC consistently established stable peer-to-peer streams, while ActionCable maintained low-latency communication for live chat. Performance was optimal under local conditions, with minimal lag and reliable

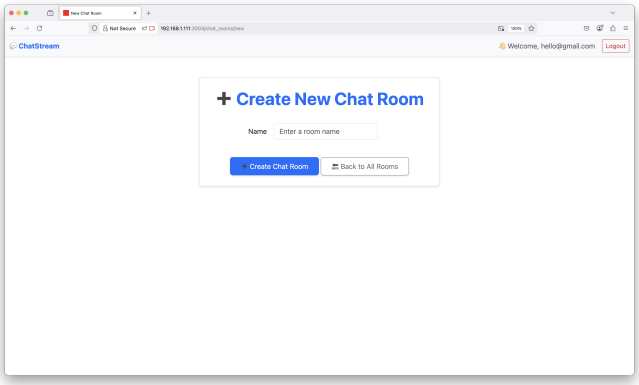

Fig. 4. chat_rooms#create view.

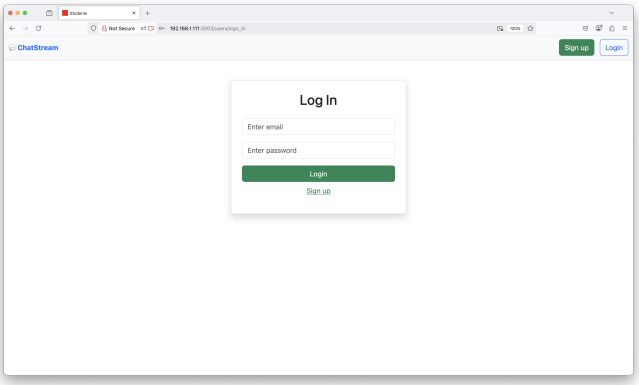

Fig. 6. sessions#new view.

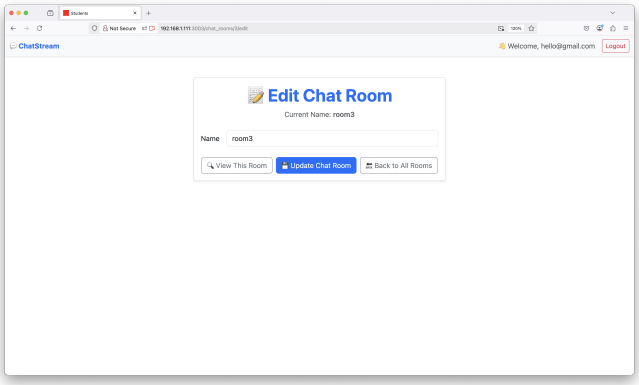

Fig. 5. chat_rooms#edit view.

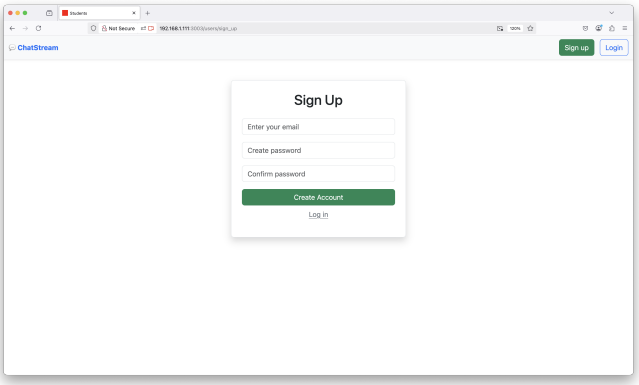

Fig. 7. registrations#new view.

media delivery across devices connected to the same Wi-Fi network.

The choice of **Ruby on Rails** as the backend framework was driven by its maturity, clear MVC structure, and built-in support for real-time features through ActionCable. It aligned well with the goal of developing a monolithic web application (MPA), where both server-side logic and views are handled within a single, cohesive framework. This architecture simplified development by reducing the need to manage separate frontend-backend communication layers, allowing for tighter integration of features such as routing, database access via ActiveRecord, and WebSocket handling through ActionCable.

HTML, **CSS**, and **JavaScript** were selected for frontend development due to their widespread support and flexibility. Bootstrap enhanced responsiveness across devices, while JavaScript was essential for managing client-side interactions, especially for initializing and controlling WebRTC media streams and updating chat content dynamically.

WebRTC was chosen for video streaming due to its native browser support, real-time peer-to-peer capabilities, and minimal overhead. Its ability to negotiate connections using ICE, STUN, and TURN ensures compatibility across varied local network setups. Using **UDP** by default for media delivery provided low-latency streaming, while falling back to **TCP** ensured robustness when UDP traversal was not possible.

ActionCable was employed to unify WebSocket integration with the Rails stack, providing seamless channels for signaling and chat delivery. Its tight coupling with Rails' infrastructure allowed for synchronized updates and simplified session management, making it ideal for real-time use cases.

Finally, **Docker** was adopted to encapsulate the full application environment. This decision ensured that development and testing remained consistent across machines, minimizing configuration drift. Containerization also supported isolated service execution and simplified dependency management, which proved crucial during local network testing.

These design choices collectively enabled the system to function reliably under local conditions and fulfilled the goals of delivering scalable, real-time video and messaging features in a browser-based environment.

Additionally, while the selected technologies offered significant advantages, some limitations were observed when compared to alternatives. For instance, Ruby on Rails facilitated rapid development and integrated WebSocket support via ActionCable; however, it lacks native support for advanced frontend interactivity compared to modern Single Page Application (SPA) frameworks like React or Vue.js. WebRTC provided efficient peer-to-peer media delivery, but configuring

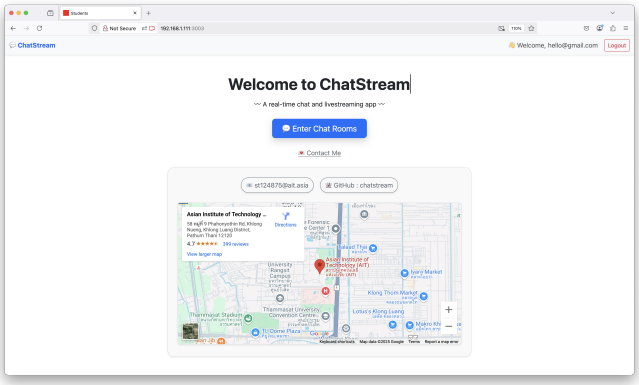

Fig. 8. home#index view.

STUN/TURN servers and ensuring browser compatibility introduced additional setup complexity. Similarly, ActionCable, though effective within the Rails ecosystem, may not scale as efficiently as standalone messaging brokers like RabbitMQ or external signaling servers in high-load environments. Despite these trade-offs, the chosen stack provided a well-balanced solution for a lightweight academic livestreaming platform deployed within a LAN context.

XI. Challenges and Limitations

One of the main challenges during development was enabling camera and microphone access through the getUserMedia() API when the application was not served over a secure context. Modern browsers, including Chrome, Edge, and Brave, block access to media devices on non-HTTPS connections or when accessed via local IP addresses such as 192.168.x.x. This restriction hindered testing across devices within a local network.

This limitation is imposed at the Application Layer of the Internet Protocol Suite, even though the underlying communication still operates over TCP at the Transport Layer and IP at the Network Layer. Web APIs like getUserMedia(), getDisplayMedia(), and enumerateDevices() are explicitly restricted to secure contexts to prevent misuse and ensure user privacy [10].

Unlike Chromium-based browsers, Firefox provides flexibility during development. By navigating to about:config and enabling the flags media.devices.insecure.enabled and media.getusermedia.insecure.enabled, developers can bypass the secure context restriction for local IP access [9].

To overcome this limitation, the system was primarily tested on localhost, which browsers inherently treat as a secure context. Additionally, tools such as Ngrok were employed to expose the local server over HTTPS, allowing access from other devices within the same network. This limitation is formally recognized by the Chromium team, emphasizing secure contexts for media APIs to enhance privacy and prevent abuse [11].

XII. Conclusion

This project successfully delivered a browser-based livestream chat system tailored for academic settings, achieving the core objective of enabling real-time, one-way video broadcasting from a designated streamer to multiple authenticated viewers, alongside interactive text-based messaging. By integrating WebRTC for efficient media transmission, ActionCable for low-latency communication, and Devise for secure authentication, the system demonstrated how modern web technologies can be effectively orchestrated within a monolithic web application architecture.

Emphasis was placed on secure access, seamless browser compatibility, and responsive performance, all of which were validated through local testing across devices on the same network. Furthermore, the system's layered design aligns with the five-layer Internet Protocol model, providing a tangible example of how protocols from the physical to application layers cooperate in real-time communication. Future enhancements may explore advanced features such as screen sharing, session recording, or support for multi-streamer environments to extend the system's utility and robustness.

XIII. Future Work

To further align the platform with academic needs, several enhancements can be considered in future development.

Scheduled Classroom Events: Integration of a calendar system to allow lecturers to schedule livestream sessions in advance, enabling automatic room activation and notifications to enrolled students.

TA-Only Chat Mode: A moderated chat mode where only authorized teaching assistants (TAs) can respond publicly to questions, reducing noise during large-scale lectures.

Quiz and Poll Integration: Real-time interactive components such as quizzes, polls, or Q&A modules can be added to improve engagement and learning outcomes during live sessions.

Exportable Chat Logs: Enabling the export of chat transcripts for academic recordkeeping, attendance validation, or post-session review by instructors and administrators.

These additions would enhance the system's utility for both instructors and students, transforming it from a basic communication tool into a full-featured academic livestreaming platform.

Acknowledgment

The author would like to express sincere gratitude to Dr. Adisorn Lertsinsrubtavee for his invaluable guidance and instruction throughout the project. Special thanks are also extended to Dr. Preechai Mekbungwan for his dedicated teaching and continued support during the course.

The author also wishes to acknowledge the contributions of various guest staff members whose assistance during hands-on sessions was greatly appreciated.

This work was supported by the Internet Education and Research Laboratory (intERLab), which provided the technical

environment and infrastructure essential for implementation and testing.

PROJECT REPOSITORY

The complete source code and configuration files for this project are available on GitHub at this repository[1].

REFERENCES

[1] S. Loreto, C. Jennings, "WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web," O'Reilly Media, 2014.

[2] DHH et al., "Ruby on Rails Guide: Action Cable Overview," rubyonrails.org. Available: https://guides.rubyonrails.org/action_cable_overview.html

[3] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, 2014. Available: https://dl.acm.org/doi/10.5555/2600239.2600241

[4] R. T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," Doctoral Dissertation, University of California, Irvine, 2000. Available: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[5] F. Dixon and R. Schroder, "BigBlueButton: Open Source Web Conferencing," in *Proceedings of the Open Source Systems Conference*, 2010. Available: https://bigbluebutton.org/

[6] E. Ivov et al., "Jitsi: The Open Source Platform for Secure Video Conferencing," Jitsi.org, 2014. [Online]. Available: https://jitsi.org

[7] N. Jain and S. Gupta, "Design and Implementation of a Real-Time Video Conferencing System Using WebRTC," in *IEEE Conference on Recent Advances in Electronics*, 2016. Available: https://www.researchgate.net/publication/367536295_Design_and_Implementation_of_WebRTC_Video_Conference_System_Structure_Compatible_with_GBT28181_Devices

[8] Google WebRTC Team, "Real-Time Communication in Web Applications," 2015. [Online]. Available: https://webrtc.org

[9] Mozilla, "Camera and microphone require HTTPS," [Online]. Available: https://blog.mozilla.org/webrtc/camera-microphone-require-https-in-firefox-68/

[10] W3C, "Secure Contexts," [Online]. Available: https://w3c.github.io/webappsec-secure-contexts/#localhost

[11] G. Urdaneta, "Intent to Remove: Nonsecure usage of MediaDevices, getUserMedia(), getDisplayMedia(), enumerateDevices() and related types," [Online]. Available: https://groups.google.com/a/chromium.org/g/blink-dev/c/SGWYHfR5CyY/m/uyUBJZAqBwAJ

[1] https://github.com/werrnnnwerrrnnnnnn/chatstream