Project Report

on

Websocket as a Service (WaaS)

Submitted by

Project Members

Hrishikesh Vaze	1032190087
Om Patel	1032190080
Aniruddha Shende	1032190079
Vartika Katiyar	1032190212

Under the Internal Guidance of

Prof. Sarika Bobde

School of Computer Engineering and Technology

MIT World Peace University, Kothrud,

Pune 411 038, Maharashtra - India

2022-2023

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

CERTIFICATE

This is to certify that

Hrishikesh Vaze, Om Patel, Aniruddha Shende, Vartika Katiyar

of BTech.(Computer Science & Engineering) have completed their project titled "Websocket as a Service" and have submitted this Capstone Project Report towards fulfillment of the requirement for the Degree-Bachelor of Computer Science & Engineering (BTech-CSE) for the academic year 2022-2023.

Prof. Sarika Bobde	**Dr. Vrushali Kulkarni**
Project Guide	Program Head
School of CET	School of CET
MIT World Peace University, Pune	MIT World Peace University, Pune

Internal Examiner:

External Examiner:

Date: 23-05-2023

Acknowledgement

Prof. Sarika S. Bobde's outstanding assistance in the planning and production of this research study is acknowledged with our sincere gratitude. Her knowledge, direction, and steadfast support have been crucial in assuring the correctness and high caliber of this job.

This report has tremendously benefited from Prof. Sarika's astute observations, critical criticism, and helpful suggestions. Her significant research expertise and experience have served as a constant source of motivation and inspiration for us. We consider ourselves extremely fortunate to have had the chance to collaborate with her, and we appreciate her kindness, tolerance, and commitment to our success. We gratefully recognise her contributions to this study paper and are eager to benefit from her knowledge going forward.

Name of the Students

Hrishikesh Vaze

Om Patel

Aniruddha Shende

Vartika Katiyar

Abstract

Real-time communication has become an essential part of many applications and systems in today's fast-paced, data-driven environment. Real-time communication is crucial for delivering notifications, updates, and other sorts of data, as well as for creating a seamless and interesting user experience. However, real-time communication infrastructure construction and upkeep can be difficult, particularly for enterprises and organizations that lack the knowledge or resources to handle it themselves. The demands of contemporary real-time applications cannot be met by the conventional HTTP request-response mechanism. Communication between the client and server is frequently sluggish, ineffective, and unreliable as a result. Our goal is to provide a real-time communication platform that enables any application, including online and mobile applications, to send and receive messages and updates in real-time with little to no latency while also ensuring scalability, dependability, a high number of concurrent users, and security. Real-time data transmission services have become less common, but certain of their capabilities, such Custom Authentication, End-to-End Encryption, Flexible APIs, or Event replay, are always vulnerable.

List of Figures

List of Tables

Table of Contents

Chapter 1 Introduction

In today's fast-paced, data-driven world, real-time communication has become a critical component of many applications and systems. Whether it's delivering notifications, updates, or other types of data, real-time communication is essential for providing a seamless and engaging user experience. However, building and maintaining real-time communication infrastructure can be challenging, especially for businesses and organizations that lack the expertise or resources to manage it themselves.

Realizing the need for a platform that provides a scalable and reliable infrastructure for real-time communication, the ExpressWaaS team developed a WebSocket as a service application. The goal of this project is to provide a fully managed WebSocket platform that can be integrated easily into any application or system, enabling real-time communication between clients and servers. Real-time communication may be easily added to a variety of applications with the help of our system ExpressWaaS. The platform is dependable, and secure, making it the perfect option for companies and organizations that must provide users with real-time data. ExpressWaaS has the infrastructure and resources you need to launch rapidly, whether you're developing a chat app, a collaboration tool, or a data-driven dashboard. By utilizing the service, businesses and organizations can focus on developing their applications and systems, without the added burden of managing the real-time communication infrastructure themselves. With the reliable and scalable infrastructure provided by ExpressWaaS, developers can deliver real-time data to their users seamlessly and efficiently, providing a better overall user experience.

Due to the rising demand for real-time communication in web applications, WebSockets are becoming more and more well-liked among web developers. However, managing the WebSockets infrastructure can be difficult and expensive, especially for applications that need to be extremely scalable and dependable. WaaS (WebSockets as a Service) is useful in this situation. Developers are relieved of the responsibility of managing the underlying infrastructure by WaaS, which enables them to outsource the management of WebSockets infrastructure to a third-party supplier. Nevertheless, there is a substantial knowledge gap about WebSocket as a Service's (WaaS) capabilities, limitations, and potential use cases.

This application will utilize modern web technologies and provide support for JSON data payload. It will be designed to handle multiple simultaneous connections. The application will be hosted on a cloud-based platform, ensuring high availability and scalability.

Chapter 2 Literature Survey

2.1 Communication Protocols

Communication protocols are the backbone of modern communication systems and are crucial for the proper functioning of any application that relies on data exchange between different devices or systems. The choice of communication protocol depends on several factors, such as the type of application, the devices involved, the environment in which the communication takes place, and the data transfer requirements. There is no one-size-fits-all protocol that can cater to the diverse needs of all applications.

Peer-to-peer connections are one of the most popular communication protocols that provide low latency, fast data transfer rates, and the ability to tolerate some packet loss. Peer-to-peer connections are typically used in applications that require real-time communication, such as online gaming, video conferencing, and voice-over internet protocol (VoIP). In peer-to-peer connections, the devices communicate directly with each other without going through a central server. This reduces the latency and improves the speed of data transfer.

However, not all applications require peer-to-peer connections. Some programs do not require communication from a distinct peer and can poll the server as needed. For example, email clients and web browsers are programs that do not require peer-to-peer connections. These applications can communicate with the server through a client-server communication protocol. In a client-server protocol, the client requests data from the server, and the server responds with the requested data.

Reliable real-time communication is crucial for some applications, such as financial trading systems and emergency response systems. In such applications, even a few milliseconds of delay in data transfer can have severe consequences. For reliable real-time communication, the communication protocol must ensure that all data is received and processed in the correct order. Examples of communication protocols that provide reliable real-time communication include TCP (Transmission Control Protocol) and SCTP (Stream Control Transmission Protocol).

When considering a protocol, it is important to understand that no protocol is better or worse than others. The protocol will depend on the type of application being built. Some protocols are designed for specific use cases and cannot be used in other applications. For example, the Modbus protocol is widely used in industrial automation systems, while the MQTT protocol is used in the Internet of Things (IoT) devices.

environment in which the communication takes place is also an important factor to consider when selecting a communication protocol. Wireless communication protocols, such as Bluetooth and Wi-Fi, are designed for communication between devices in close proximity. These protocols are not suitable for long-range communication. In contrast, communication protocols like GSM (Global System for Mobile Communications) and LTE (Long-Term Evolution) are designed for long-range communication and are commonly used in cellular networks.

factor to consider when selecting a communication protocol is the data transfer requirements. Some applications require high-speed data transfer, while others require low-speed data transfer but with high reliability. For example, video streaming applications require high-speed data transfer, while industrial control systems require low-speed data transfer but with high reliability.

in short selecting the right communication protocol for an application is crucial for its proper functioning. The choice of protocol depends on several factors, such as the type of application, devices involved, environment, and data transfer requirements. There is no one-size-fits-all protocol, and each protocol has its own strengths and weaknesses. Therefore, it is important to carefully evaluate the requirements of the application and select the appropriate protocol that can cater to those requirements.

2.2 HTTP in Real Time Communication

HTTP (Hypertext Transfer Protocol) is a widely used protocol that facilitates the transfer of data over the Internet. It was first introduced in the early 1990s as a means of sharing hypertext documents between computers on the internet and has since become the foundation of the World Wide Web. HTTP is used to transfer data between web servers and clients and has undergone several revisions to support new features and technologies.

Initially designed for transferring static content, HTTP has since evolved to support real-time applications as well. One of the most significant advancements in this regard has been the introduction of WebSockets, which allows developers to establish a persistent connection between the client and server. This enables bi-directional communication and real-time updates, making HTTP a popular choice for real-time applications such as chat systems, online gaming, stock trading platforms, and monitoring systems.

One of the key benefits of using HTTP for real-time applications is its reliability and scalability. HTTP is a tried and tested protocol that has been used in the industry for decades and has proven to be a reliable solution for transferring data over the Internet. Additionally, the use of HTTP allows for easy integration with existing web infrastructure and tools, making it an accessible choice for developers. For example, many web developers are already familiar with HTTP and can leverage their existing knowledge and tools to build real-time applications.

Furthermore, the use of HTTP also provides benefits in terms of security and performance. HTTP can be used with Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to encrypt data transmission, ensuring that sensitive data is protected from prying eyes. In terms of performance, HTTP can support low latency and high performance, allowing for real-time updates and rapid data transfer.

However, there are also some limitations to using HTTP for real-time applications. One of the main challenges is that HTTP is based on a request-response model, which means that the client must initiate a request before the server can respond. This can introduce latency and impact real-time responsiveness, particularly for applications that require frequent updates. Additionally, HTTP is not a true real-time protocol, as it relies on polling or long-polling mechanisms to achieve real-time communication.

Despite these limitations, HTTP remains a popular choice for real-time applications, particularly for applications that require low latency and high performance. To further enhance the capabilities of HTTP for real-time applications, various extensions and frameworks have been developed. For example, some frameworks provide a simplified programming model for building real-time applications over HTTP, while others provide additional features such as message queuing, broadcasting, and clustering.

HTTP is a widely used protocol that provides a reliable and scalable solution for transmitting data over the Internet. With the introduction of WebSockets, HTTP can now also be used for real-time communication, making it a popular choice for real-time applications such as chat systems, online gaming, stock trading platforms, and monitoring systems. While there are limitations to using HTTP for real-time applications, its reliability, security, and performance benefits make it an attractive choice for developers. As the needs of real-time applications continue to evolve, it is likely that HTTP will continue to adapt and evolve to meet these needs.

2.3 WebSocket in Real Time Application

Real-time applications have become increasingly popular in recent years, particularly with the rise of WebSockets. WebSockets offer several advantages over conventional HTTP queries, which make them an ideal choice for real-time communication. In this article, we will explore the benefits of WebSockets and the various studies conducted in this field.

One of the main advantages of WebSockets is their ability to reduce latency. Latency refers to the delay between a user's action and the server's response. In real-time applications such as chat programs and online games, low latency is essential to ensure a smooth user experience. WebSockets maintain an open connection between the client and server, allowing data to be transmitted quickly and efficiently, in contrast to HTTP requests, which require the establishment of a new connection for each request. This persistent connection reduces the latency, making WebSockets ideal for real-time applications.

Another advantage of WebSockets is their ability to enable two-way communication between the client and server. With WebSockets, both the client and server can send and receive data, allowing for real-time data exchange. This is particularly useful for applications such as chat programs, where users need to see messages from other users immediately. Additionally, WebSockets have less overhead than conventional HTTP requests, allowing for the transmission of more data and an improved user experience.

have been the subject of several studies, each focusing on different aspects of the technology. In their study, Ogundeyi and Yinka-Banjo provide a comprehensive overview of the use of WebSockets in real-time applications. The authors explain the WebSocket protocol, its advantages, and disadvantages, and highlight various real-time applications that can benefit from its use. They also provide a practical example of implementing WebSockets in a real-time chat application and discuss the security implications of using WebSockets.

Similarly, in their study, Darshan G. Puranik et al. discuss the implementation of real-time monitoring systems using AJAX and WebSockets and propose a solution that combines the best of both technologies. The authors provide a detailed description of the system architecture and include a case study that demonstrates the effectiveness of the proposed solution in a real-world scenario.

Another study by Rachna Agrawal and colleagues provides insights into the technical and privacy aspects of WhatsApp's end-to-end encryption mechanism. The study covers various aspects of WhatsApp encryption, such as key generation, key exchange, and message encryption, and highlights the privacy implications of WhatsApp's end-to-end encryption.

Additionally, a study by Murley et al. explores the adoption of WebSocket technology and its impact on the real-time web. The authors analyze data from a measurement study of Alexa's top 10,000 websites to understand the prevalence of WebSocket adoption and discuss how WebSocket can enable new types of real-time web applications.

Finally, a study on the presence of WebSockets in various programming languages and libraries provides a comprehensive analysis of the technology's performance and practical implications for developers. The study found that the choice of programming language and library can significantly impact the performance of WebSockets.

In terms of communication protocols used with WebSockets, there are several options available. The most commonly used protocols are the WebSocket Protocol (RFC 6455) and the Socket.IO protocol. The WebSocket Protocol is the standard protocol used for WebSockets and is supported by most modern web browsers. The Socket.IO protocol is a higher-level protocol built on top of WebSockets that provides additional features such as room management and automatic reconnection. WebSockets offer several advantages over conventional HTTP queries for real-time applications. Their ability to reduce latency, enable two-way communication, and have less overhead makes them an ideal choice for applications such as chat programs, online gaming, stock trading platforms, and monitoring systems.

2.4 WebSockets in various programming languages and libraries

WebSockets provide a powerful mechanism for real-time communication between clients and servers on the web. When implementing WebSocket functionality, developers have a variety of programming languages and libraries to choose from. Each language and library has its own strengths and considerations, such as performance, ease of use, and integration capabilities. Evaluating the specific requirements of a project and understanding the features and performance characteristics of different WebSocket implementations help us to make an informed decision.

When considering WebSocket libraries, it is important to evaluate their performance characteristics. Several factors, such as connection establishment success rate, time taken to connect to a large number of WebSocket clients, response time to a high volume of requests, and round-trip time, contribute to performance metrics. Following are some of them:

1. C / Libwebsockets:
    - Libwebsockets is a lightweight C library that provides a simple and efficient way to implement WebSocket functionality.
    - It offers a range of features, including SSL/TLS encryption, event-driven architecture, and support for various WebSocket protocols.
    - However, due to its low-level nature, using Libwebsockets requires a good understanding of C programming and socket handling.
2. C++ / uWebSockets:
    - uWebSockets is a C++ library that provides a high-performance WebSocket implementation.
    - It offers a developer-friendly API, supports SSL/TLS encryption, and has built-in support for pub/sub patterns and HTTP upgrades.
    - The library is known for its excellent performance and low memory footprint, making it suitable for resource-constrained environments.
3. C# / Fleck:
    - Fleck is a popular WebSocket library for C# that simplifies WebSocket implementation in .NET applications.
    - It provides an easy-to-use API, supports SSL/TLS encryption, and integrates well with ASP.NET and other frameworks.

- Fleck offers features like message framing, event-driven programming, and client and server implementations.

4. Go / Gorilla:
    - Gorilla WebSocket is a Go package that offers a complete and robust WebSocket implementation.
    - It provides a clean API, supports SSL/TLS encryption, and includes features like WebSocket compression and subprotocol support.
    - Gorilla WebSocket is widely adopted in the Go community and is known for its excellent performance and scalability.
5. Java / Java-WebSocket:
    - Java-WebSocket is a Java library that enables WebSocket communication in Java applications.
    - It provides a simple API, supports SSL/TLS encryption, and offers features like message compression and per-message deflate.
    - Java-WebSocket is easy to use and integrates well with existing Java frameworks, making it a popular choice for Java developers.
6. Node.js / uWebSockets:
    - uWebSockets is a high-performance WebSocket library for Node.js that is built on top of the uWebSockets C++ library.
    - It provides a simple and efficient API, supports SSL/TLS encryption, and offers features like pub/sub and HTTP upgrade support.
    - uWebSockets for Node.js combines the performance benefits of the underlying C++ library with the ease of use and flexibility of JavaScript.
7. PHP / Ratchet:
    - Ratchet is a PHP library that facilitates WebSocket implementation in PHP applications.
    - It offers a simple and intuitive API, supports SSL/TLS encryption, and provides features like broadcasting, asynchronous messaging, and per-connection event handlers.
    - Ratchet integrates well with popular PHP frameworks and is widely used for real-time applications in the PHP ecosystem.
8. Python / websockets:
    - The websockets library in Python provides a complete implementation of the WebSocket protocol.

- It offers a straightforward API, supports SSL/TLS encryption, and includes features like message framing and per-message deflate compression.
- The websockets library is well-documented and widely used in the Python community for WebSocket-based applications.

9. Rust / rust-web socket:
    - The rust-websocket crate is a WebSocket library for the Rust programming language.
    - It provides a clean and ergonomic API, supports SSL/TLS encryption, and offers features like per-message compression and subprotocol support.
    - Rust developers appreciate the safety and performance benefits that the language provides, making rust-websocket a popular choice for WebSocket implementation in Rust projects.

In a comparative analysis of various WebSocket libraries, Node.js emerged as the top performer. Node.js demonstrated superior performance across multiple metrics, including connection success rate, response time, and round-trip time. Its event-driven, non-blocking architecture allows it to handle a large number of concurrent connections efficiently. Additionally, the simplicity and ease of use of Node.js make it an optimal choice for WebSocket projects.

2.5 Data Encryption

In an era of increasing digital threats and privacy concerns, data encryption plays a critical role in safeguarding sensitive information. Encryption technologies serve different purposes, offering varying levels of protection to data during transit and at rest. This article explores several encryption technologies, including end-to-end encryption (E2EE), the Signal Protocol, and AES, shedding light on their functionalities, benefits, and applications.

End-to-End Encryption (E2EE):

End-to-end encryption is a secure communication method that protects data being transported from one endpoint to another from prying eyes. Unlike traditional server-based messaging frameworks, which often lack E2EE encryption, this approach ensures the confidentiality and integrity of message exchanges between clients, eliminating the need to trust third-party servers with the original messages. By encrypting data at the source and decrypting it only at the destination, E2EE reduces the number of parties who may potentially interfere with or break the encryption, enhancing overall security.

Signal Protocol:

The Signal Protocol is a cryptographic protocol designed to provide end-to-end encryption for instant messaging, voice and video calls, and file transfers. It focuses on safeguarding the confidentiality and security of communication between multiple parties. The Signal Protocol employs a combination of symmetric and asymmetric encryption, along with perfect forward secrecy, which ensures that messages remain secure and confidential even if encryption keys are intercepted by unauthorized parties. By utilizing advanced encryption techniques, the Signal Protocol offers a robust defense against eavesdropping, data tampering, and other security threats.

AES (Advanced Encryption Standard):

AES is a symmetric-key encryption method widely recognized for its strength and security. It is commonly used to protect data both at rest and in transit. AES operates on fixed-length blocks of data and uses a symmetric key for both encryption and decryption. With different key sizes (128, 192, and 256 bits), AES provides varying levels of security. It is a trusted encryption standard used to secure sensitive information such as files, emails, and confidential documents. Additionally, AES is widely employed to safeguard internet communications through protocols like SSL/TLS, ensuring secure connections between clients and servers.

2.6 Research Gaps

The rapid growth of technology has brought about new opportunities and challenges that require constant research and investigation. One such area of study is real-time applications and the use of Web Sockets as a means of communication. While there have been significant advancements in this area, there are still gaps in the literature that need to be addressed.

One area that requires further investigation is the scalability of Web Sockets in large-scale applications. While Web Sockets have been shown to improve performance and reduce latency in real-time applications, it is important to explore their scalability and compare them with other real-time communication protocols like Server-Sent Events and Long Polling. Scalability is a critical aspect of any technology, and understanding the limitations and challenges of Web Sockets is essential in building efficient and responsive real-time applications.

Another area that requires further research is the security implications of using Web Sockets in real-time applications. While Web Sockets allow for real-time data exchange, the transmission of sensitive data over Web Sockets can pose security risks. Therefore, it is essential to explore the security implications of Web Sockets and develop secure mechanisms for transmitting sensitive data over Web Sockets.

Moreover, the proposed solutions for real-time applications like monitoring systems need to be thoroughly evaluated for their scalability, security, and performance in large-scale applications. The paper on real-time monitoring using AJAX and WebSockets provides practical guidance on combining the two technologies for efficient and responsive monitoring systems. However, there is still a need to explore the scalability, security, and performance of the proposed solution in large-scale monitoring systems. Additionally, the limitations and challenges associated with end-to-end encryption, as well as usability issues, need to be addressed to ensure secure and user-friendly communication.

Furthermore, researchers should consider conducting studies that are comprehensive, inclusive, and diverse, taking into account different programming languages, libraries, and sectors. Overall there is a need for further research in the field of real-time applications and the use of Web Sockets as a means of communication. Addressing the gaps in the literature can help in building a better understanding of the issues and provide valuable insights for practitioners and policymakers. Researchers should conduct comprehensive, inclusive, and diverse studies that address the limitations and challenges of the technologies they investigate. By doing so, they can develop innovative solutions to real-world problems, which can have a significant impact on society.

Chapter 3 Problem Statement

The traditional HTTP request-response model is not sufficient to meet the requirements of modern real-time applications. It often results in slow, inefficient, and unreliable communication between the client and server. A solution is to create a real-time communication platform that allows web and mobile applications, or any other application to send and receive messages and updates in real time with low latency while ensuring reliability, a large number of concurrent users, and security. There have been fewer services offering real-time data communication, yet some features like Custom Authentication, End-to-End Encryption, Flexible APIs, or Event replay are always compromised.

3.1 Project Scope

- The WaaS application should provide low latency connection and data transfer between servers and clients.
- The application should be able to handle a large number of concurrent users with no major impact on its performance.
- The application should also have an HTTP medium of data transfer along with WebSocket.
- The application should facilitate its client with a custom authentication feature for their clients.
- The application should also offer data transfer with end-to-end encryption to their clients.
- The application should be able to record and replay the events.
- The application must support the transfer of JSON data only.
- The application should be able to make HTTP-type requests over Websocket.

3.2 Project Assumptions

- The network infrastructure of the client and server will support WebSocket protocol.
- The application assumes that browser client machine will have a modern web browser that supports modern JavaScript.
- The SDK assumes that the data being transferred will be in the JSON format.
- The platform will be able to handle a large number of simultaneous WebSocket connections.
- The platform will provide real-time monitoring and analytics of WebSocket connections and messages.

3.3 Project Limitations

- The application is limited to supporting JSON data payloads only. Other data formats such as XML or binary data may not be supported. Clients have to wrap other data into JSON and then they can transfer it.
- The application is limited to providing WebSocket connection support for real-time data transfer only. Though the transfer of large files or video streaming or streaming of any data is still not yet supported by the SDK. In this case, clients have to write their algorithms (i.e. dividing data into small chunks and managing sequence) to make things possible.
- When any client or server is restarted it regenerates the RSA key pair. When recording and replaying the events, we cannot store the events while E2EE is on. This is because clients may have regenerated their key pair, making them unable to decrypt the old encrypted data while replaying.

3.4 Project Objectives

- To explore the requirements of developers.
- To study different algorithms for implementing E2EE.
- To develop an SDK for communicating with the platform.
- To develop documentation for both SDK and Rest API.
- To create a user interface for monitoring real-time data.
- To evaluate the performance of the system.

Chapter 4 Project Requirements

4.1 Resource

4.1.1 Reusable Software Components

- Pusher Module
- HTTPOverWS
- Node SDK
- Encryption Module

4.1.2 Software Requirement

- The system must support the latest version (atleast ES6) of Node.js and Socket.IO.
- The system must be able to handle a high volume of WebSocket connections simultaneously.
- The system must have a secure authentication mechanism to prevent unauthorized access.
- The system must support data encryption for end-to-end security.
- The system must provide a simple and user-friendly API for client applications to use.
- The system must be able to monitor and log errors for debugging purposes.
- The system must support multiple operating systems including Linux, macOS, and Windows.
- The system must provide real-time statistics and analytics for clients to monitor their usage.

4.1.3 Hardware Requirements

- At Least 8 GB Ram
- At Least 30 GB storage
- At Least 1 Mbps of network bandwidth
- 4 core CPU

4.2 Risk Management

Risk Management Factor	Description	Priority
Testing	Insufficient testing leading to bugs in the production environment	High
Technical Complexity	Lack of expertise or experience in WebSocket can lead to delays, cost overruns, or even failure.	High
Security	WebSocket applications can be vulnerable to attacks, such as cross-site scripting, injection attacks, and session hijacking. Proper security measures must be implemented.	High
Scalability	Designing for scalability can be complex. If the application is not designed for scalability from the beginning, it may not perform well under heavy loads, causing downtime or poor user experience	High
Integration	The WebSocket application must be integrated with other systems and technologies, such as databases, APIs, and web servers. Any integration issues or incompatibilities can cause delays and increased development costs.	Medium

Risk Management Factor	Description	Priority
User Experience	The WebSocket application must provide a smooth and intuitive user experience. Any issues with usability or functionality can lead to dissatisfied users and decreased adoption rates.	Medium
Compatibility	WebSocket is not supported by all browsers and devices, which can limit the application's reach. Compatibility issues must be addressed during development and testing to ensure the application works as expected across all supported platforms.	Medium
Regulatory Compliance	Depending on the nature of the application and the data it processes, there may be regulatory compliance requirements that must be met. Failure to meet these requirements can result in legal and financial penalties.	Low

Table 1 Risk Management Factors

4.3 Functional Specification

4.3.1 Interface

- **External interfaces**
  - RESTful API: The WebSocket as a service application should provide a RESTful API that allows developers to send and receive data payloads over this connection.

    - ➢ Webhooks: The application should also support webhooks to notify subscribed clients of relevant events and data updates.
- **Internal interface:**
    - ➢ **Data storage interface:** The application should have an interface for storing WebSocket connection data, such as connection IDs, and metadata
    - ➢ **Event handler interface:** The application should have an interface for handling incoming events and forwarding them to the appropriate clients.
- **Communication interfaces:**
    - ➢ **Pusher Module:** This interface should be able to transfer data according to the metadata attached to it.
- **Graphical User Interfaces:**
    - ➢ **Dashboard:** The application should provide a graphical user interface for managing clients' servers and clients, recordings, etc.

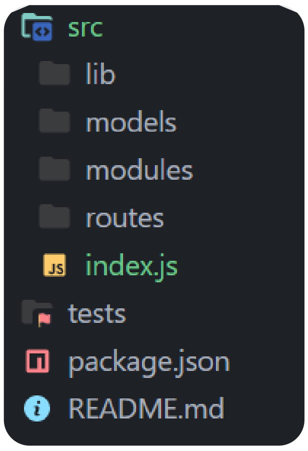
4.3.2 Interactions

- **Sustainability:**
    - ❖ The application should be designed such that new features can be added easily without changing the previous code.
    - ❖ The application should also be easy to deploy and maintain, with a focus on minimizing downtime and optimizing resource usage.
- **Quality management:**
    - ❖ The application should be thoroughly tested to ensure that it meets the functional and non-functional requirements and that it is robust and reliable.
    - ❖ The application should also be monitored and logged to enable efficient troubleshooting and issue resolution.
- **Security:**
    - ❖ The application should support SSL/TLS encryption to ensure that all communication between clients and servers is secure and protected against eavesdropping.
    - ❖ The application should also implement access control measures, such as API keys, to restrict access to authorized clients and prevent unauthorized access and misuse of the WebSocket connections and data.

Chapter 5 System Analysis Proposed Architecture/ HLD of Project

5.1 Design Considerations

The folder structure should be consistent throughout the project and follow a standard naming convention. It should be modular so that different modules of the project can be developed independently. It should be easily accessible to all team members and should be well-documented. So we followed the following folder structure in all our applications.



```
src
  lib
  models
  modules
  routes
  index.js
tests
package.json
README.md
```

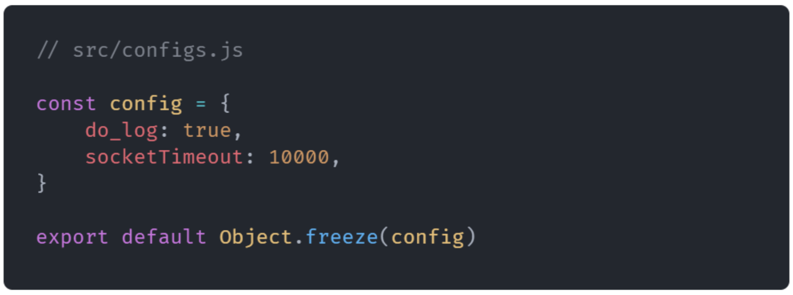
Figure 1 Default folder structure of our all repositories

We have also followed the SOLID principles of Object-Oriented Programming, which includes Single responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion. Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code. We have implemented the code keeping some patterns in consideration. This is what makes our application more scalable.

5.1.1 Singleton pattern

This pattern solves two common problems. It ensures that a class has just a single instance and provides a global access point to that instance. It can be easily done by making the class constructor private and creating a static method to return a single instance.

In our application, we have used this pattern in exporting system global settings/configurations. we implemented this with the help of javascript's Object.freeze method. Below is an example code for it.



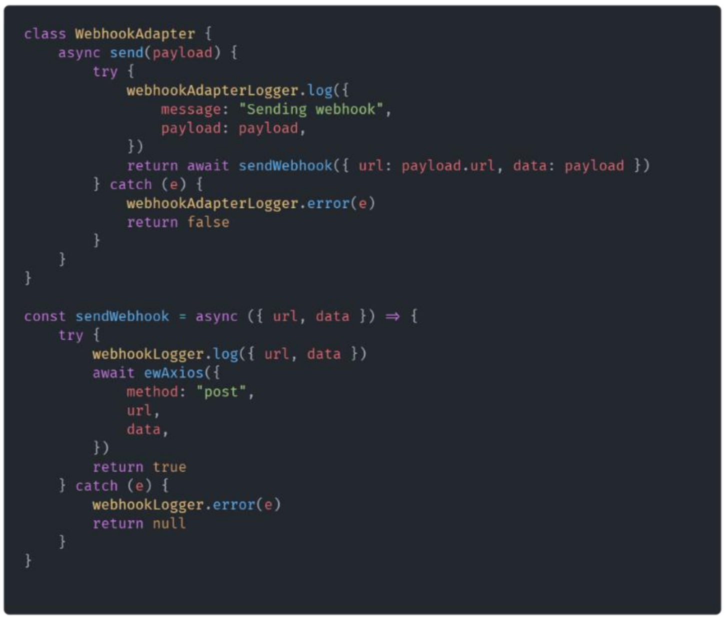
```
// src/configs.js

const config = {
    do_log: true,
    socketTimeout: 10000,
}

export default Object.freeze(config)
```

Figure 2 Example of Singleton Pattern used in global configuration in our Application

5.1.2 Adapter pattern

This pattern is useful when two or more interfaces are incompatible with each other or when a large number of interfaces with common functionality must be implemented. The adapter pattern allows classes with incompatible interfaces to collaborate by creating a wrapper object that translates one class's interface to another class's expected interface.

The pusher module in our application is responsible for routing incoming packets to their respective clients. This is accomplished through the use of metadata attached to each packet. Some packets are to be routed via WebSockets, while others are to be routed via Webhooks. Because our application is a WebSocket as a Service, we will concentrate on Websockets. As a result, the webhook interface is incompatible with the pusher. So we implemented webhookAdapter. Below is the example code.



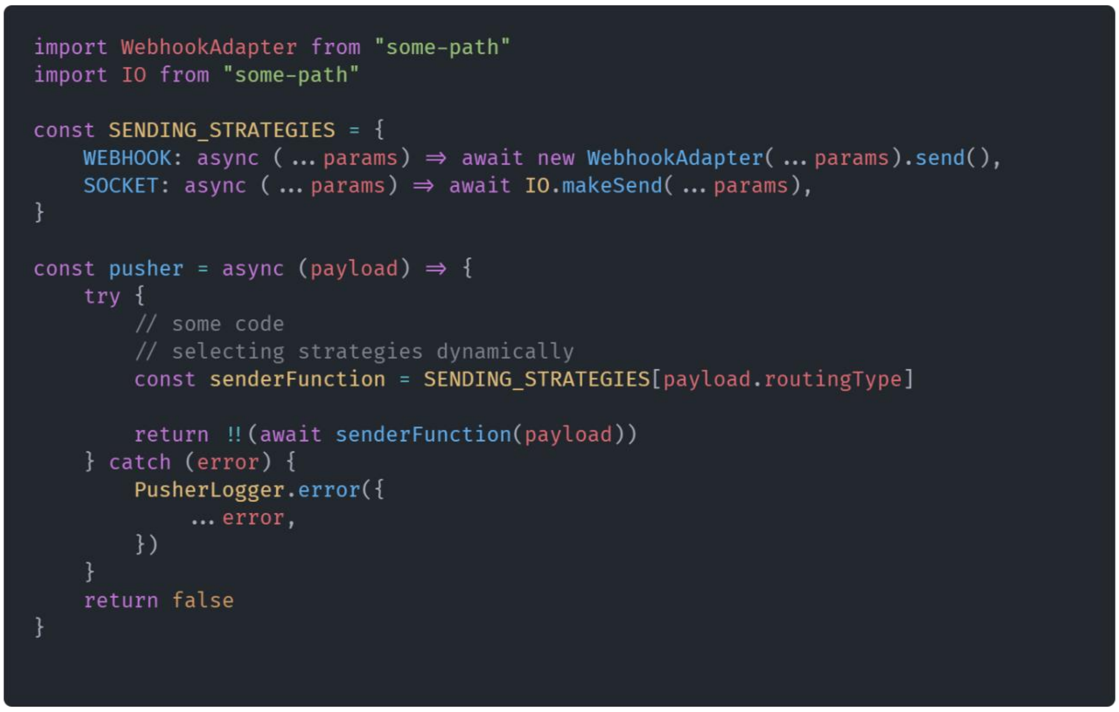
```
class WebhookAdapter {
    async send(payload) {
        try {
            webhookAdapterLogger.log({
                message: "Sending webhook",
                payload: payload,
            })
            return await sendWebhook({ url: payload.url, data: payload })
        } catch (e) {
            webhookAdapterLogger.error(e)
            return false
        }
    }
}

const sendWebhook = async ({ url, data }) => {
    try {
        webhookLogger.log({ url, data })
        await ewAxios({
            method: "post",
            url,
            data,
        })
        return true
    } catch (e) {
        webhookLogger.error(e)
        return null
    }
}
```

Figure 3 Example of Adapter Pattern used in our Application

5.1.3 Strategy pattern

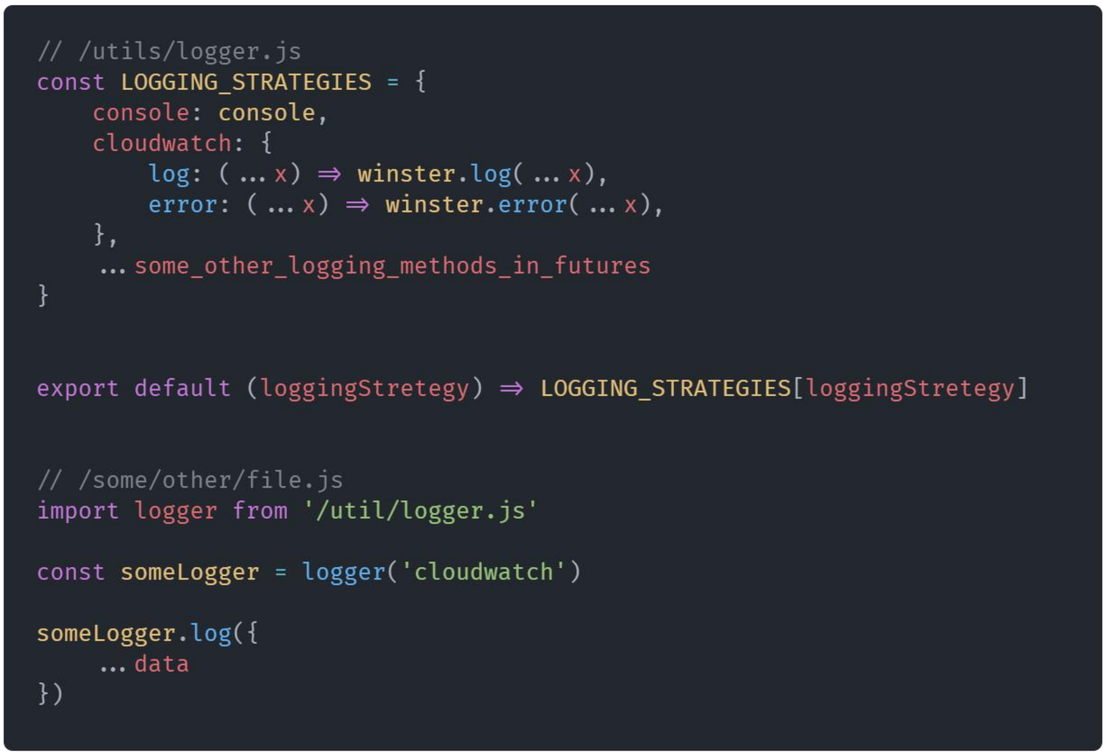
This pattern is used when we have different strategies or ways or methods or techniques to do the same thing. Generally, the outputs of all these methods are the same. This pattern ensures that we can add a new method or algorithm for the same thing easily in the future without affecting the old methods. Clients using old methods will have no conflicts with the addition of new methods.

In our application, this pattern best suits the pusher module, and the global logger class made. Below is the example code.



```
import WebhookAdapter from "some-path"
import IO from "some-path"

const SENDING_STRATEGIES = {
    WEBHOOK: async (...params) => await new WebhookAdapter(...params).send(),
    SOCKET: async (...params) => await IO.makeSend(...params),
}

const pusher = async (payload) => {
    try {
        // some code
        // selecting strategies dynamically
        const senderFunction = SENDING_STRATEGIES[payload.routingType]

        return !!(await senderFunction(payload))
    } catch (error) {
        PusherLogger.error({
            ...error,
        })
    }
    return false
}
```

Figure 4 Example of Strategy Pattern used in Pusher module



```
// /utils/logger.js
const LOGGING_STRATEGIES = {
    console: console,
    cloudwatch: {
        log: (...x) ⇒ winster.log(...x),
        error: (...x) ⇒ winster.error(...x),
    },
    ...some_other_logging_methods_in_futures
}


export default (loggingStretegy) ⇒ LOGGING_STRATEGIES[loggingStretegy]


// /some/other/file.js
import logger from '/util/logger.js'

const someLogger = logger('cloudwatch')

someLogger.log({
    ...data
})
```

Figure 5 Example of Strategy Pattern for using different logging strategies in our Application

5.2 Assumption and Dependencies

We assume that our clients are familiar with Node.js, but we are considering expanding language support in the future. We assume that clients are familiar with the event-based architecture. However, we will provide complete service documentation. Still, it is assumed that they understand how JWT or token-based authentication works. Clients are expected to primarily use WebSockets for real-time data transfer rather than webhooks.

For storing data, managing connections, and processing events, the application may also rely on third-party services and platforms such as cloud hosting providers, message brokers, and database management systems. Furthermore, the application may be dependent on the availability and compatibility of third-party client libraries and frameworks that developers may use to integrate WebSocket functionality into their applications and systems.

5.3 General Constraints

The application may be constrained by network bandwidth limitations, which may affect clients' ability to establish and maintain web socket connections with us, or the server's ability to handle large volumes of incoming and outgoing WebSocket messages. The application may also be limited by computing resources such as CPU, memory, and storage, which may limit the number of concurrent WebSocket connections and the amount of data that the server can process and store.

The application may also be limited by the WebSocket protocol's requirements, which impose limitations on the structure and content of WebSocket messages, as well as the maximum size of WebSocket payloads that can be sent or received.

The application may be subject to various legal and regulatory constraints, such as data privacy laws, intellectual property rights, and export control regulations, which may limit the types of data that can be exchanged over WebSocket connections as well as the countries or regions where the service is available.

5.4 System Architecture

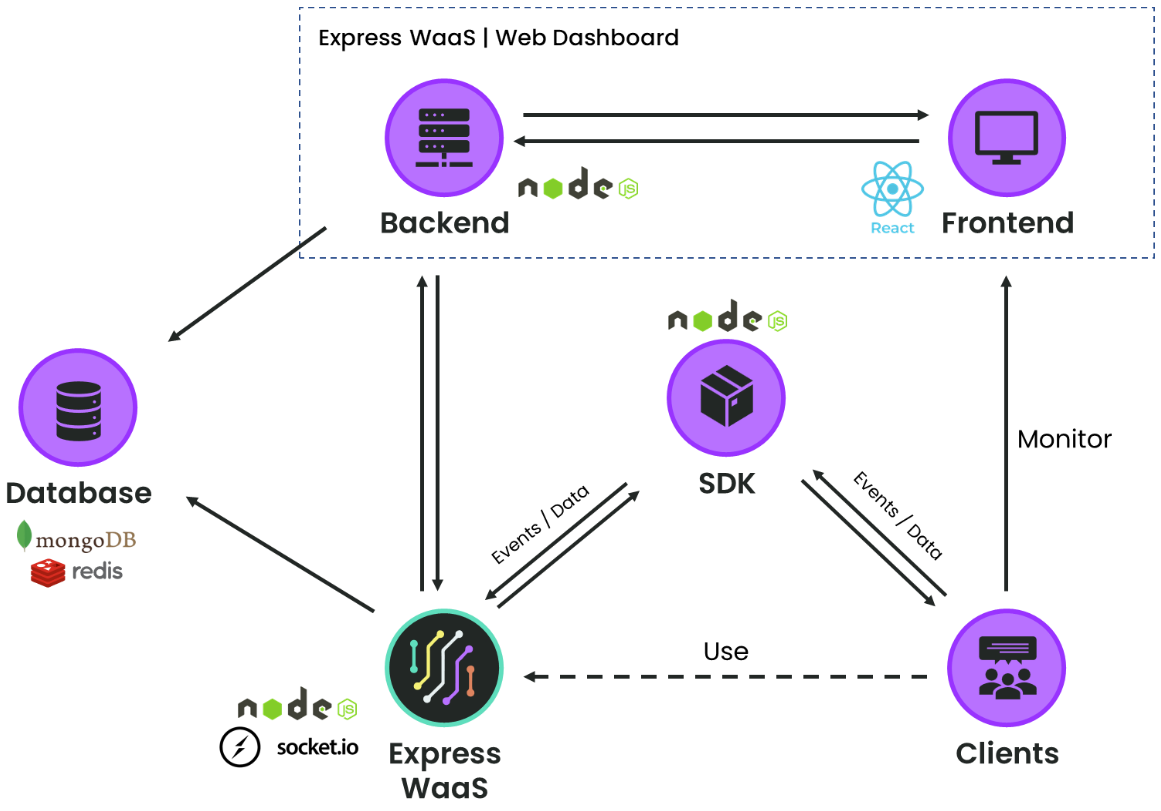


Figure 6 System Architecture Diagram

The system is divided into several parts, including the backend, frontend, database, and SDK. Node.js is used to build the backend, SDK, and service (ExpressWaaS). The front end is built with

React, a popular JavaScript library for creating user interfaces. We have also published SDK (under the name ew-sdk) on NPM (Official Node Package Manager) making it easy for our clients to interact with our APIs.

The system makes use of two databases: MongoDB as the primary database, and Redis as an in-memory database for ultra-fast IO.

Socket.IO is the base WebSocket library used by the service application. It provides functionality to end users by utilizing a variety of events and data. Aside from that. The backend is linked to the service via Socket.IO to transfer important data between them.

We, as developers, use PM2 (Advanced Process Manager built with Node.js) for monitoring, logging, and deploying our applications.

5.5 Modules of the Project

There are four main modules of the project:

1. EW-Service

   The Service module of the project is built using Node.js, and Socker.IO server. It uses MongoDB as a database and Redis as an in-memory data store.
2. EW-Frontend

   The frontend module of our application is built using ReactJS. The purpose of this module is to provide the user interface for our application and to enable real-time communication between the clients and the application.
3. EW-backend

   The backend module of our application is built using Node.js and Socket.IO server.
4. EW-NodeSDK

   The NPM module, built on javascript. It exposes some useful classes making it easier for clients to interact with our APIs.

5.6 UML Diagrams/Agile Framework

5.6.1 Package Diagram

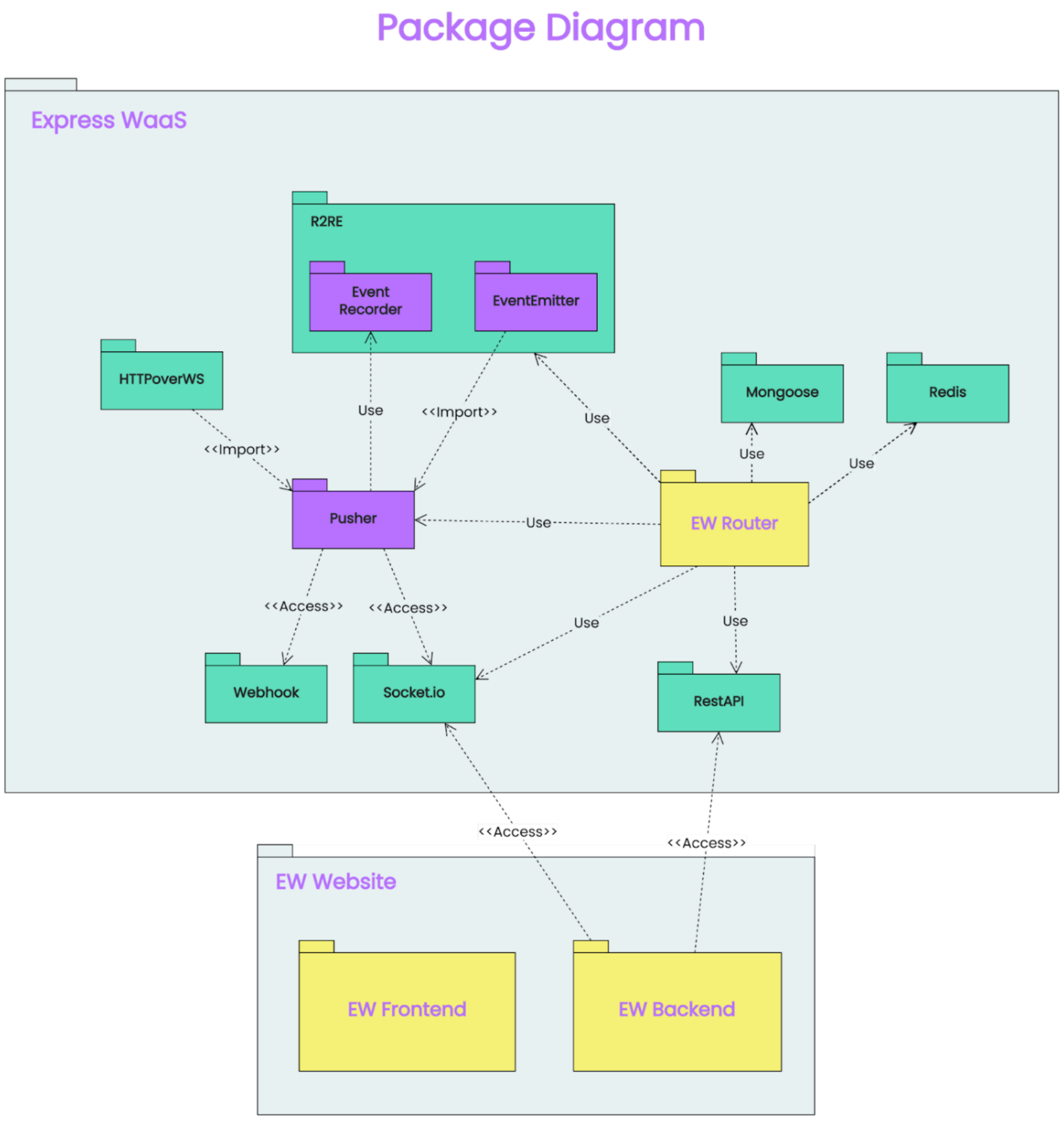


Figure 7 Package Diagram

The package diagram shows the different components and modules that make up the system.

1. **EWRouter:** Responsible for routing incoming client requests to appropriate handlers, ensuring messages are delivered to the correct endpoint.
2. **Pusher:** The main heart of the application that handles the routing functionality.
3. **Webhook:** Allows developers to receive real-time notifications through HTTP protocol when specific events occur, enabling real-time notifications for the client.

4. **Socket.io:** Enables real-time, bi-directional communication between web clients and servers, providing real-time communication between the client and the server.
5. **Mongoose:** A library that provides a way to interact with MongoDB databases, used in this package diagram to interact with the database.
6. **Redis:** An in-memory data structure store used as a database, cache, and message broker to provide caching and messaging functionality.
7. **RestAPI:** Handles RESTful API requests from clients, returning JSON responses.
8. **HTTPoverWS:** A very well-designed module that makes HTTP-type requests over Websocket protocol. These requests are faster than HTTP requests.
9. **Event Recorder:** Records and logs events for replaying and debugging purposes.
10. **EventEmitter:** This enables the application to replay the recorded events.
11. **EW Frontend:** The user interface that interacts with the client.
12. **EW Backend:** This mainly serves EW Frontend.

5.6.2 Activity Diagram

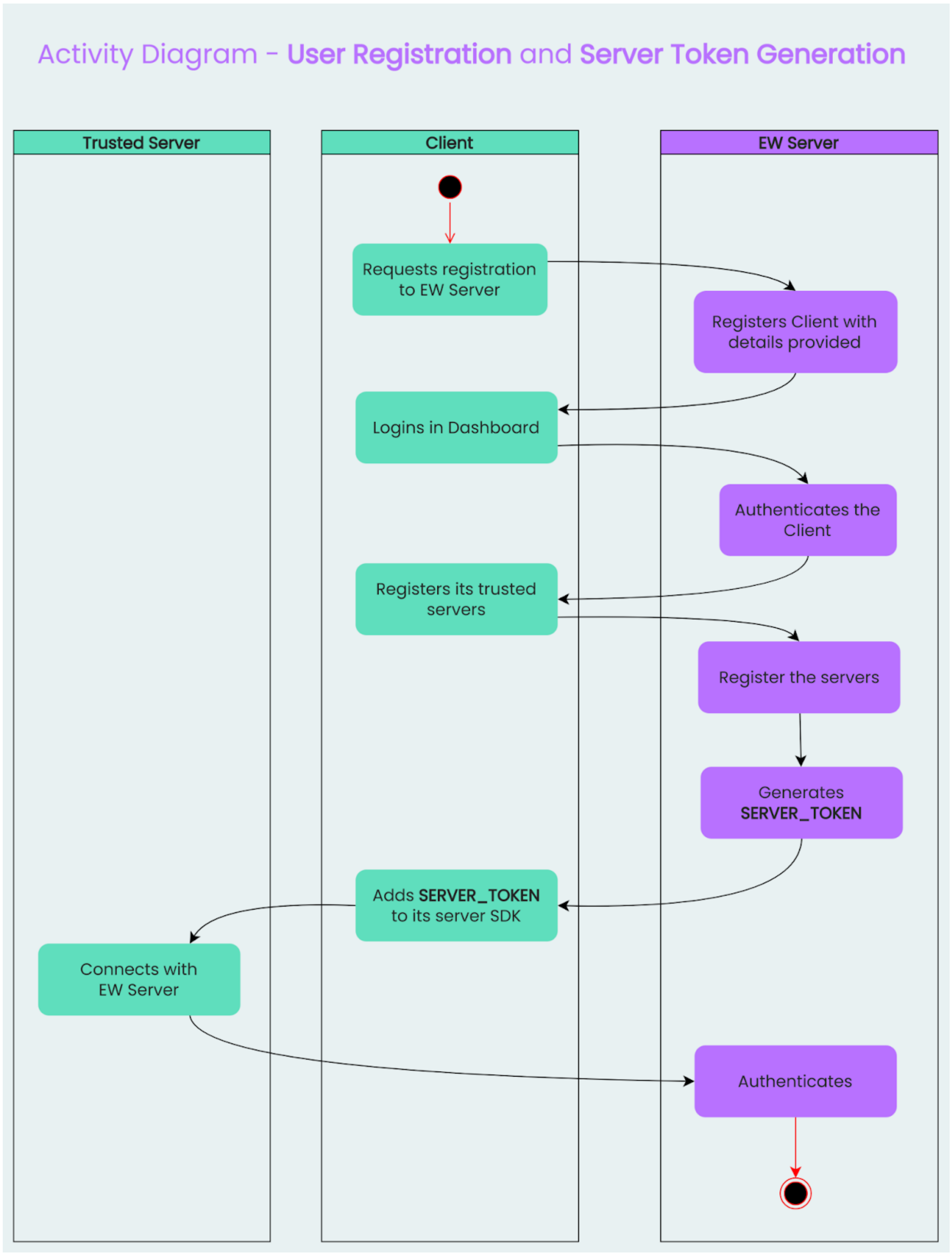

Activity Diagram - User Registration and Server Token Generation
Trusted Server
Client
EW Server
Requests registration to EW Server
Registers Client with details provided
Logins in Dashboard
Authenticates the Client
Registers its trusted servers
Register the servers
Generates SERVER_TOKEN
Adds SERVER_TOKEN to its server SDK
Connects with EW Server
Authenticates

Figure 8 User Registration Activity Diagram

The activity diagram outlines the process of user registration and server token generation in a system that involves a trusted server, a client, and an EWServer. The process begins with the user requesting registration to the EW server, which is followed by the user logging into the dashboard. Once logged in, the user can register their trusted servers with the system and proceed to register the client with the details provided.

After the client is registered, the system authenticates the client to ensure that it is authorized to access the server. Once the client is authenticated, the server generates SERVER_TOKENS, which are used to securely connect the client with the EW server. The SERVER_TOKENS are then added to the server SDK and the server connects with the EW server.

In more detail, the user begins by requesting registration to the EW server. Once done, they can log in to the system.

After logging in, the user can register their trusted servers with the system. To do this, the user must provide the server name, a unique address (or namespace), and a description of the server. The server is then added to the list of trusted servers in the system.

Clients can use generate token functionality from the dashboard to generate a unique token for their trusted server. The client then uses SERVER_TOKEN to authenticate with the EW server. The authentication process involves sending the SERVER_TOKEN to the EW server, which verifies the tokens and authenticates the client. Once the client is authenticated, it can access the services provided by the EW server.

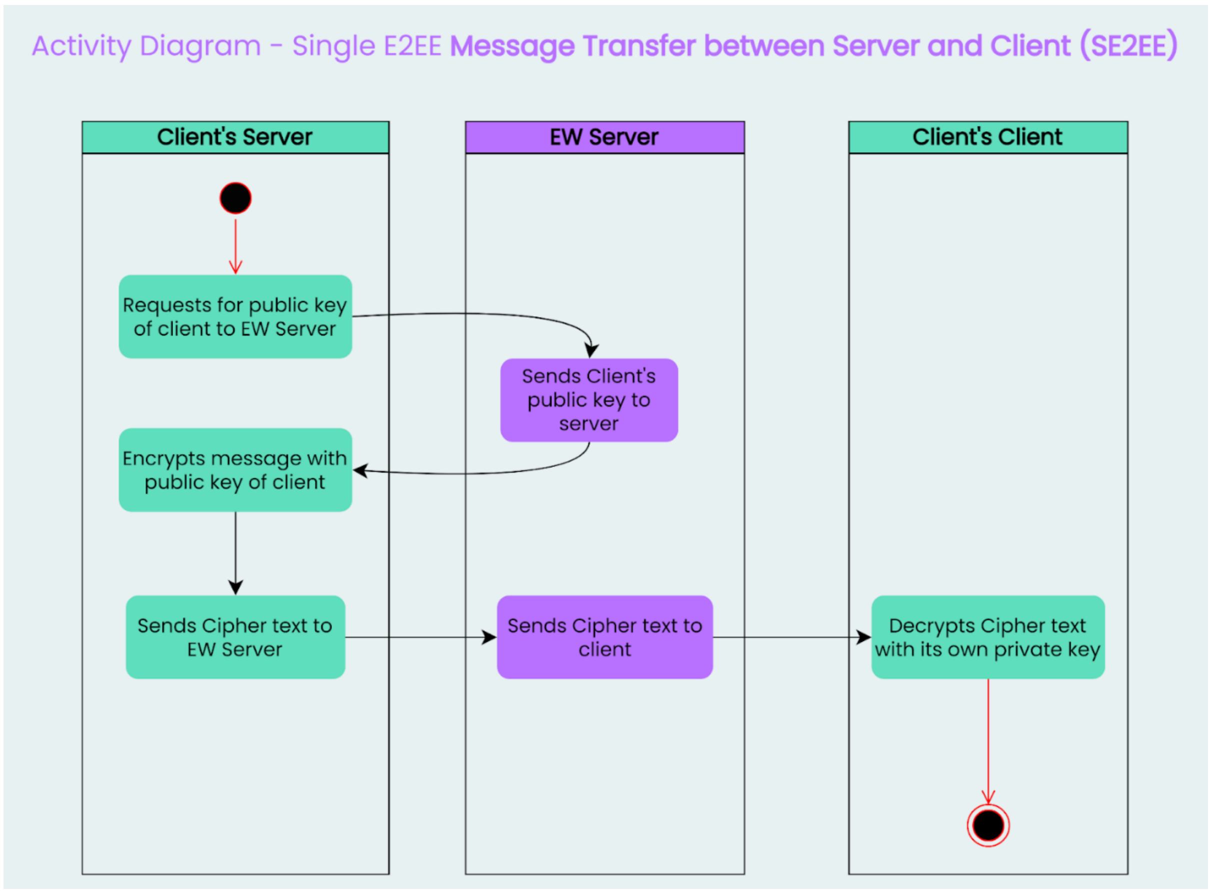

Activity Diagram - Single E2EE Message Transfer between Server and Client (SE2EE)
Client's Server
EW Server
Client's Client
Requests for public key of client to EW Server
Sends Client's public key to server
Encrypts message with public key of client
Sends Cipher text to EW Server
Sends Cipher text to client
Decrypts Cipher text with its own private key

Figure 9 Single E2EE Message Transfer b/w Server and Client Activity Diagram

This activity diagram outlines the process of a single E2EE message transfer between a client and a server in a system that involves the EW server. The process starts with the client's server requesting the public key of the client from the EW server. Once the EW server receives the request, it sends the public key of the client to the client's server.

The client's server then encrypts the message with the public key of the client. This ensures that only the client can decrypt the message. Once the message is encrypted, the client's server sends the client's public key to the server along with the cipher text.

The server then sends the cipher text to both the EW server and the client's client. The EW server receives the cipher text and forwards it to the client's client for decryption.

Finally, the client's client decrypts the cipher text with its private key. This completes the process of a single E2EE message transfer between the client and the server.

5.6.3 Sequence Diagram

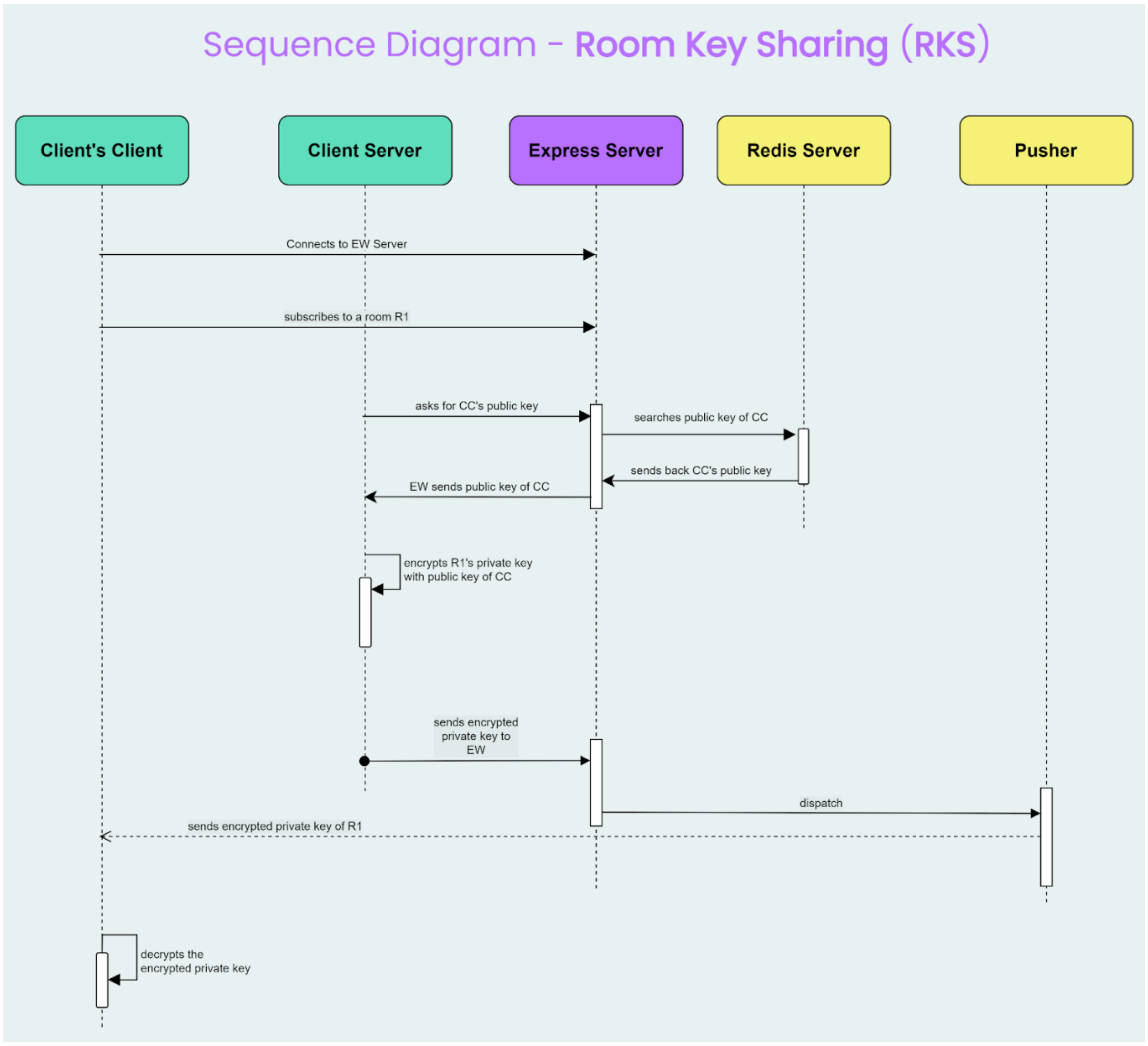

Sequence Diagram - Room Key Sharing (RKS)
Client's Client
Client Server
Express Server
Redis Server
Pusher
Connects to EW Server
subscribes to a room R1
asks for CC's public key
searches public key of CC
sends back CC's public key
EW sends public key of CC
encrypts R1's private key with public key of CC
sends encrypted private key to EW
dispatch
sends encrypted private key of R1
decrypts the encrypted private key

Figure 10 Room Key Sharing Sequence Diagram

The sequence diagram for Room Key Sharing (RKS) outlines the steps involved in sharing a private key for a room between a client and a server. The process starts with the client's client connecting to the EW server and subscribing to a specific room R1. The client's server then requests the public key of the client's client (CC) from the EW server.

Once the EW server receives the request, it searches for the public key of the CC and sends it to the client's server. The client's server then encrypts the private key of R1 with the public key of CC, ensuring that only the CC can decrypt it.

Next, the client's server sends the encrypted private key of R1 to the EW server, along with the encrypted private key and the public key of CC. The EW server forwards the encrypted private key of R1 to the express server, which in turn stores it in the Redis server.

The pusher service is then used to notify all subscribers of room R1 that the private key has been updated. The client's client receives the notification and requests the encrypted private key of R1 from the EW server.

Once the encrypted private key of R1 is received, the client's client decrypts it using its private key, which was previously obtained from the EW server. This completes the process of sharing a private key for a room between the client and the server.

The process involves the client's client connecting to the EW server, subscribing to a specific room R1, and requesting the public key of the CC. The client's server then encrypts the private key of R1 with the public key of CC, and the EW server stores and forwards the encrypted private key of R1 to the express server and the client's client. The client's client then decrypts the encrypted private key using its private key, completing the process of sharing the private key for room R1. The sequence diagram provides a visual representation of the steps involved in the process, making it easier to understand and implement the system.

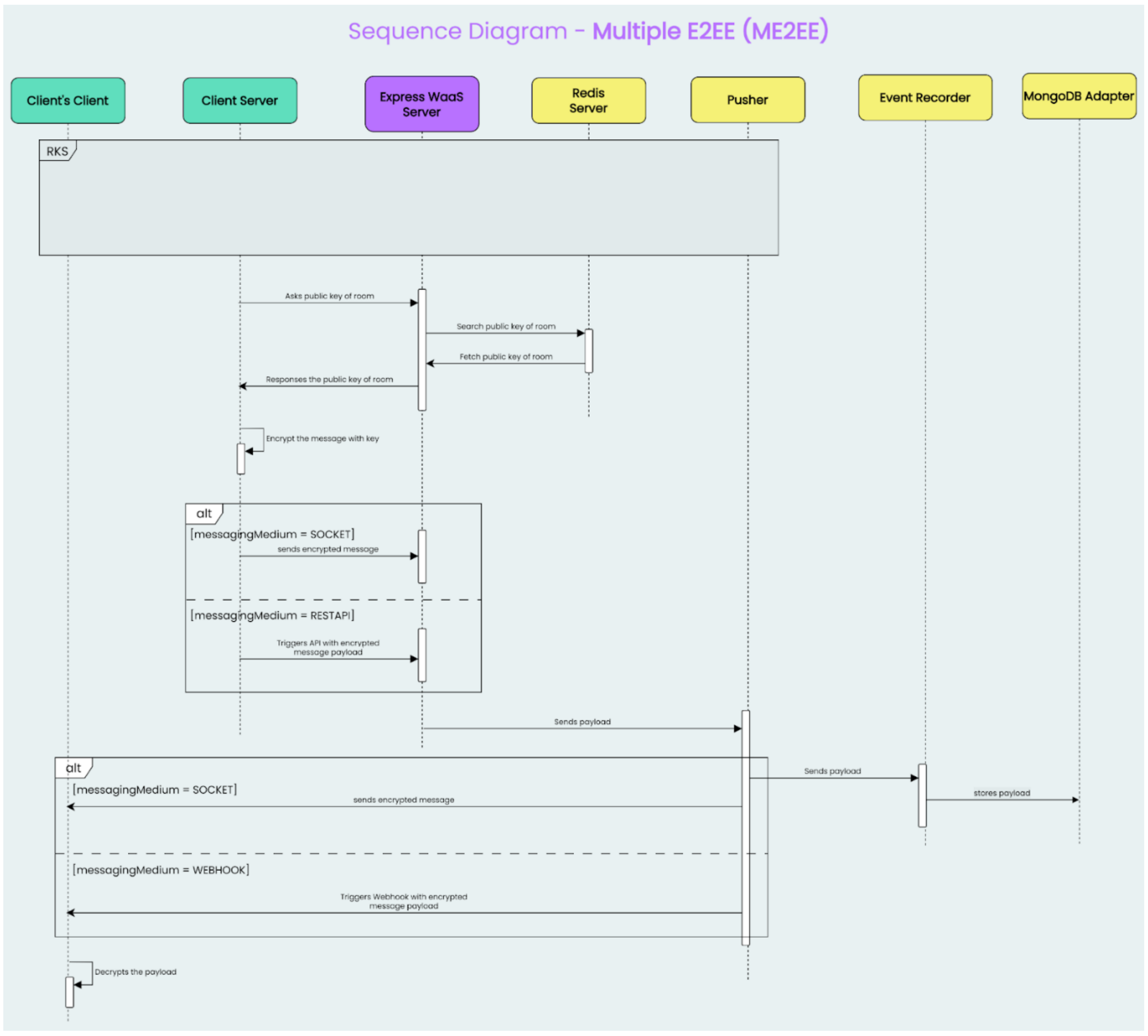

Sequence Diagram - Multiple E2EE (ME2EE)
Client's Client
Client Server
Express WaaS Server
Redis Server
Pusher
Event Recorder
MongoDB Adapter
RKS
Asks public key of room
Search public key of room
Fetch public key of room
Responses the public key of room
Encrypt the message with key
alt
[messagingMedium = SOCKET]
sends encrypted message
[messagingMedium = RESTAPI]
Triggers API with encrypted message payload
Sends payload
alt
[messagingMedium = SOCKET]
sends encrypted message
Sends payload
stores payload
[messagingMedium = WEBHOOK]
Triggers Webhook with encrypted message payload
Decrypts the payload

Figure 11 Multiple E2EE Sequence Diagram

The sequence diagram for Multiple E2EE (ME2EE) outlines the steps involved in the end-to-end encryption of messages between multiple clients and servers. The process starts with the client's client requesting the public key of the room from the client's server. The client's server then searches for and fetches the public key of the room from the Redis server and responds to the client's client with the public key.

Once the client's client receives the public key, it encrypts the message with the key and sends the encrypted message using one of the three messaging mediums: SOCKET, RESTAPI, or WEBHOOK.

If the messaging medium is SOCKET, the encrypted message is sent directly to the server using the messaging socket. If the messaging medium is RESTAPI, the encrypted message is sent as a payload in a RESTAPI trigger request. If the messaging medium is WEBHOOK, the encrypted message is sent as a payload in a webhook trigger request.

Once the encrypted message is received by the server, it is stored in MongoDB using the MongoDB Adapter, and a record of the event is created using the Event Recorder. The server then retrieves the encrypted message payload and decrypts it using the private key associated with the room. The decrypted message is then sent to all the subscribed clients in the room using the Pusher service.

The Room Key Sharing (RKS) process is used to securely share the private key of the room between the clients and servers involved in the encryption and decryption of the messages. The process involves requesting and fetching the public key of the room from the Redis server, encrypting the message using the room's public key, and sending the encrypted message using one of the two messaging mediums: SOCKET, or WEBHOOK. The server then stores and decrypts the encrypted message using the private key associated with the room and sends the decrypted message to all the subscribed clients in the room using the Pusher service. The Room Key Sharing (RKS) process is used to securely share the private key of the room between the clients and servers involved in the encryption and decryption of the messages. The sequence diagram provides a visual representation of the steps involved in the process, making it easier to understand and implement the system.

5.7 System Layer Diagram/Block diagram

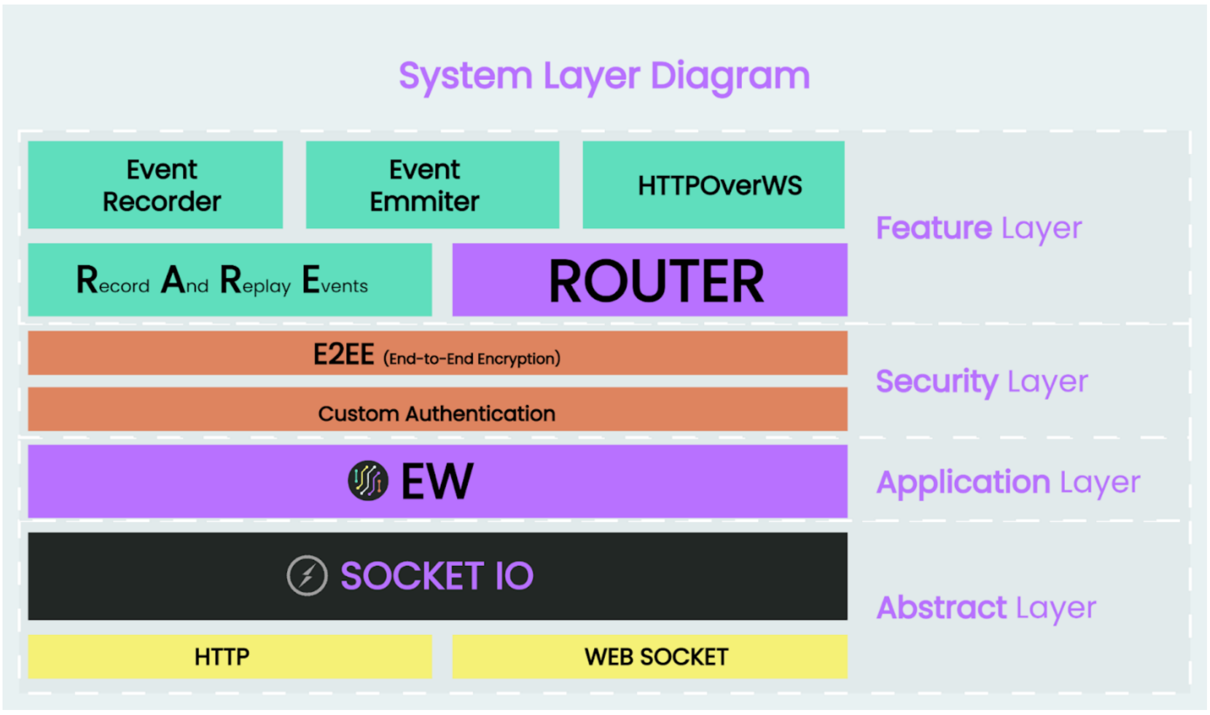
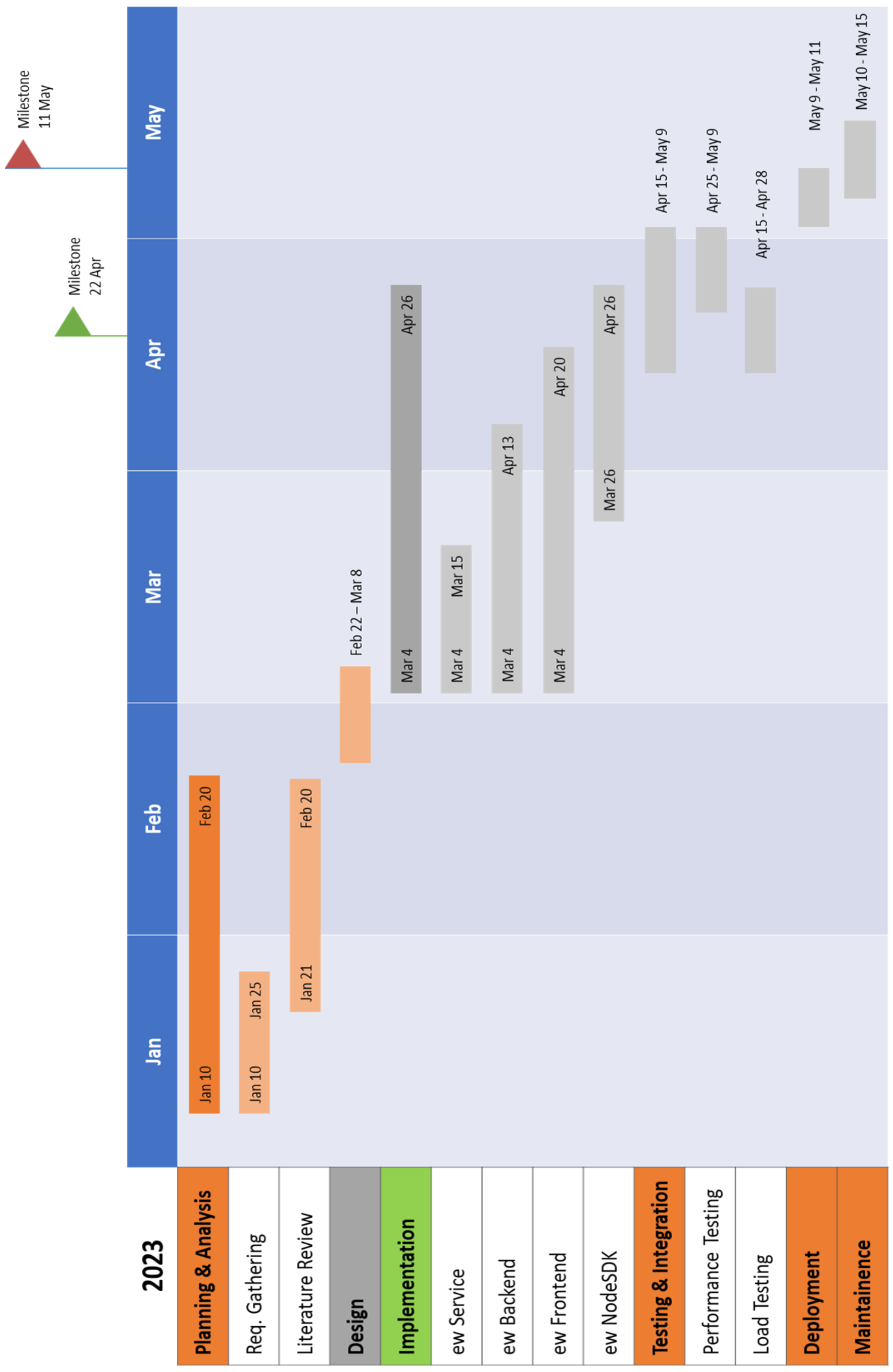


Figure 12 System Layer Diagram

The System Layer diagram depicts the overall layers of our server application. HTTP and WebSocket make up the bottom layer. These are the two mediums at the base that we use to transfer data. On top of that, the Socket.IO library is used to manage the WebSocket connection and event loop. Because they are hidden from end users, these two are referred to as the Abstract layer. Our EW services are available at the application level. On top of that, we've added custom authentication and end-to-end encryption. These act as a security layer when combined. On top of that, there is a feature layer. RARE (Record And Replay Events), HTTPOverWS and Pub/Sub Features are the most important. In short, data flows from top to bottom when sent and from bottom to up when received.

Chapter 6 Project Plan



Chapter 7 Implementation

7.1 Methodology

7.1.1 Technology Stack

We use Node.js for developing the whole system. For the database, we use MongoDB. MongoDB is a popular NoSQL database. To store key pairs, Redis in-memory database is used.

For developing the frontend part of the dashboard, we use the React.js library. The system also uses a popular library Socket.io for implementing web sockets. The system uses Nginx's port forwarding. Below are the given applications/libraries used to build our application.

1. ReactJS v18.0.0^
2. Node Js v18.15.0 LTS
3. NPM v9.6.2^
4. Socket.io v4.6.0^
5. MongoDB 6.0
6. Redis 6.4.0

7.1.2 Web Applications

The system consists of three applications

1. Service Application
2. Frontend Application
3. Backend Application

The service app takes care of all services given to clients. All three applications are built on Node.js on top of the socket.io library. Other than this an NPM SDK is published to communicate with our applications.

7.1.3 Packages used in each Module

1. EW-Service
    a. Express Framework
    b. Socket.IO server
    c. Mongoose
    d. RedisAdapter

   e. Axios
   f. Webhook Adapter (application module)
   g. Pusher (application module)
   h. HTTPOverWS (application)
   i. AutoHOW (application module)
2. EW-Frontend
   a. ReactJS
   b. Socket.IO client
   c. Axios
3. EW-Backend
   a. Axios
   b. Express Framework
   c. Socket.IO server
   d. Mongoose
4. EW-sdk
   a. HTTPOverWS (application module)
   b. E2EE (application module)

7.1.4 System Architecture

As discussed earlier, the system consists of three applications. The dashboard consists of a front-end application and a back-end application. The communication between them will happen either through RestApi or through web sockets for real-time updates. The backend will be communicating with service applications for some real-time updates to be shown on the dashboard. This communication happens over the web socket.

7.1.5 Collections in MongoDB

We identified the following collections to be used in the system:

- Clients
- ClientUsers
- Events
- Records

A Client object represents an organization.

A ClientUser object represents the individual servers or clients that are associated with that organization and can be uniquely identified.

7.1.6 Communication

There are two methods for communicating: WebSocket and Webhook. Our service is based on a publisher-subscriber model. Clients must subscribe to a specific topic, and topic updates are published by the server. The system uses the socket.io library as the foundation for the real-time data transfer service. Socket.IO is a JavaScript library that enables real-time, bidirectional, and event-based communication between the browser and the server. It provides a simple and efficient way to create real-time web applications that work across different platforms and devices. Socket.IO uses WebSockets, a protocol that allows for two-way communication between the client and server, to provide real-time updates and push notifications to users. It also has fallback mechanisms in place to ensure compatibility with older browsers and devices.

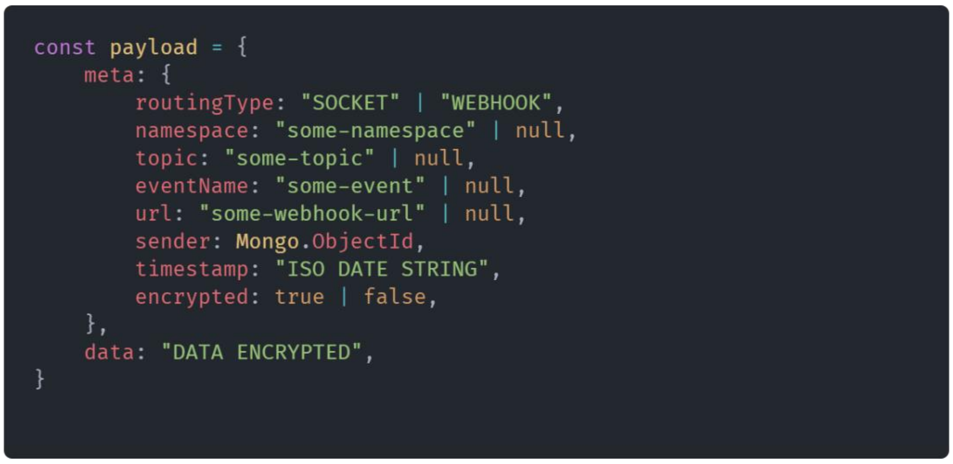
7.1.7 Authentication

The system uses JSON Web Token (JWT) for authenticating its clients. It is a popular stateless authentication mechanism that provides a secure and efficient way to handle user authentication and authorization in web applications. The JWT generated by the system consists of a unique Mongo ObjectID. So a client will need a JWT token in order to connect to the service or use the RESTApi.

There are two types of clients that will connect to the EW service - the client's servers and the client's clients. This implies that the client can be classified into two types - ServerType and ClientType.

7.1.8 Pusher Module

Pusher module is responsible for all the routing of packets happening in the service. It accepts input in the following payload format:



```
const payload = {
    meta: {
        routingType: "SOCKET" | "WEBHOOK",
        namespace: "some-namespace" | null,
        topic: "some-topic" | null,
        eventName: "some-event" | null,
        url: "some-webhook-url" | null,
        sender: Mongo.ObjectId,
        timestamp: "ISO DATE STRING",
        encrypted: true | false,
    },
    data: "DATA ENCRYPTED",
}
```

Figure 13 Payload format

See the above payload format. The meta field tells the communication medium. If routingType is defined as "SOCKET" then fields like "eventName", "topic", and "namespace" are used. Else in the case of routingType as "WEBHOOK" fields like "url" are used. The "encrypted" field tells if the data is encrypted or not.

7.2 Algorithm

We have used following algorithms within our application:

7.2.1 Room Key Sharing (RKS)

When a server needs to send a private key to a client, the Room Key Sharing algorithm will be used. This algorithm specifies how a server will send a client a private key.

When a Client's client is connected to the server, the public key of the client's client is shared with the server. To share the room key client's server encrypts the room key with the client's client public key. This encrypted key is then shared with the client's client and then it decrypts using its private key to get the room key.

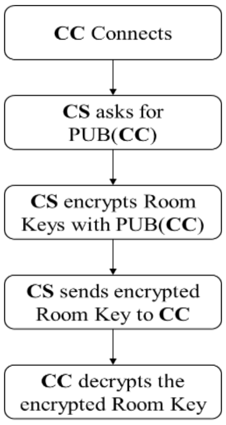

CC Connects
CS asks for PUB(**CC**)
CS encrypts Room Keys with PUB(**CC**)
CS sends encrypted Room Key to **CC**
CC decrypts the encrypted Room Key

Figure 14 Room Key Sharing Algorithm Block Diagram

7.2.2 End-to-End Encryption

End-to-end encryption is a security measure that ensures that only the sender and intended recipient can read the encrypted data, rendering it unreadable to everyone else, including service providers and hackers. This algorithm specifies how the system encrypts and decrypts the data at the sender's and receiver's end.

We use both RSA and AES encryption techniques to encrypt the data. RSA is an asymmetric encryption algorithm that is widely used for encryption. As a typical 4096 RSA can only encrypt data up to 512 bytes, AES encryption is also used. AES is a widely used symmetric encryption algorithm. The general idea is to encrypt the data symmetrically and then encrypt the symmetric keys with the RSA algorithm. The AES algorithm here is used along with Cipher Block Chaining (CBC) mode. Each plaintext block in CBC mode is combined with the previous ciphertext block before being encrypted. This ensures that identical plaintext blocks are encrypted into different ciphertext blocks, making identifying patterns in the data more difficult for an attacker. Below are the steps given to encrypt and decrypt the data.

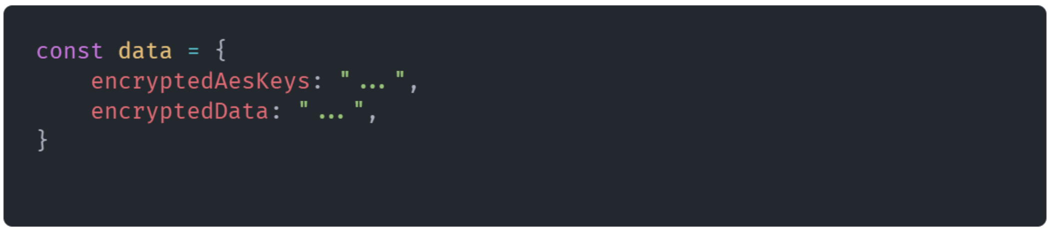
Encrypting the data:

1. Generate RSA key-pair with a key length of 1024 bits.
2. Generate a 256-bit key and 128 bot iv (initialization vector) for 256-bit AES encryption in CBC mode.
3. Encrypt data with AES. We call this CT (Cipher Text).
4. Encrypt AES key and iv using Pub(receiver)

Decrypting the data:

1. Decrypt AES key and iv with Pk(receiver)
2. Do AES decryption on CT.

Following is the encrypted data format:



```
const data = {
    encryptedAesKeys: "...",
    encryptedData: "...",
}
```

Figure 15 Data format after encryption

See the above format. The field "encryptedAesKeys" contains an AES key and an iv that have been encrypted using the RSA algorithm. The message is encrypted with the AES algorithm and is stored in the "encryptedData" field. Both of these fields must be transferred to the receiver in order for the message to be decrypted.

Chapter 8 Performance Evaluation and Testing

The main objective of our testing was to assess the performance and latency of the service under various load and payload conditions. To achieve this objective, we conducted several tests to determine the average latency with varying client counts, compare the latency of HTTP and HTTP over WebSocket protocols, and examine the effect of payload size on latency. Additionally, we included clients from the United States to compare the results with Indian clients, as our servers are hosted on Indian machines.

Our testing revealed that the performance and latency of the WebSocket-based service were significantly affected by the number of concurrent clients. As the number of clients increased, we observed a consistent increase in latency. This suggests that the service may experience performance degradation when subjected to a large number of concurrent users. The latency of the service is a critical factor in its performance and can significantly impact the user experience. Our findings indicate that optimizing the service to handle a higher number of concurrent clients could improve its reliability and scalability.

We also examined the effect of payload size on the service's latency. Our tests showed that as the payload size increased, so did the latency. This trend suggests that our WebSocket-based service may need to be more suitable for the real-time transmission of large data payloads. This is an important consideration since many modern web applications require real-time transmission of data with large payloads. Our testing indicates that optimizing the service to handle large data payloads could improve its performance significantly.

Furthermore, we compared the latency of HTTP and HTTP over WebSocket protocols. Our results showed that HTTP over WebSocket had much lower latency than HTTP-based communication, with an average latency of only 65% of that of HTTP-based communication. This finding suggests that using WebSocket-based communication can significantly improve the performance of web applications. This is especially true for applications that require real-time communication, such as chat applications, multiplayer games, and collaborative applications.

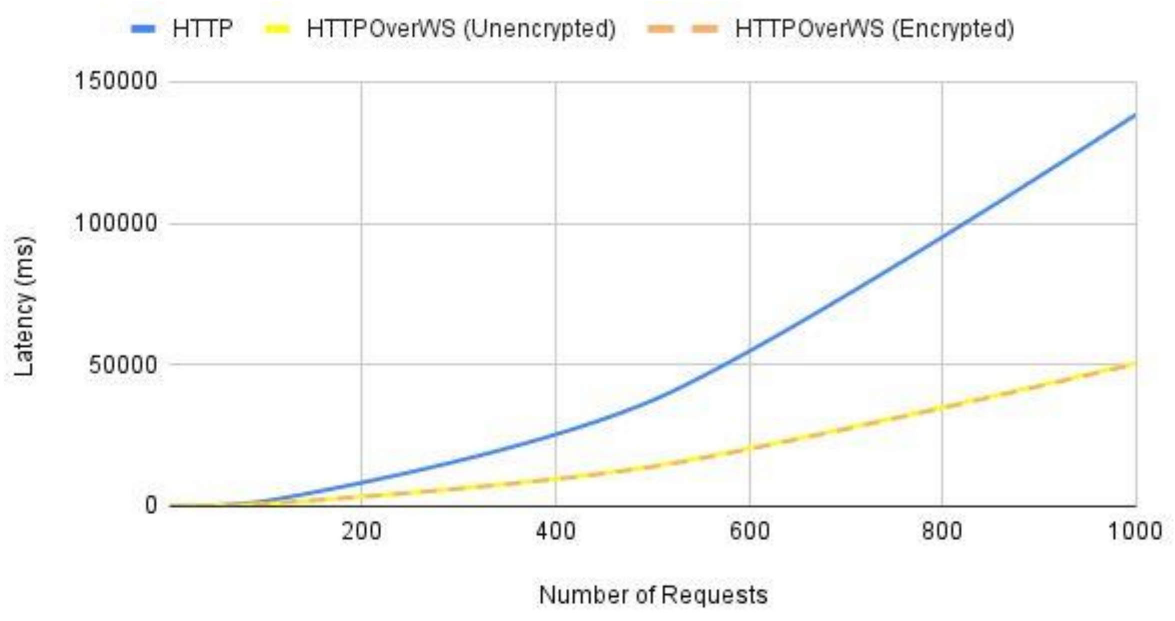


Figure 16 Latency Comparison: HTTP vs HTTPOverWS

The graph presented above illustrates the comparison of latency between HTTP Call and HTTPOverWS (HTTP over WebSocket) with and without E2EE. Our analysis revealed that HTTPOverWS exhibits significantly lower latency than traditional HTTP, with an average latency of only 65% compared to HTTP-based communication. This finding highlights the benefits of employing HTTPOverWS instead of HTTP when low-latency communication is essential.

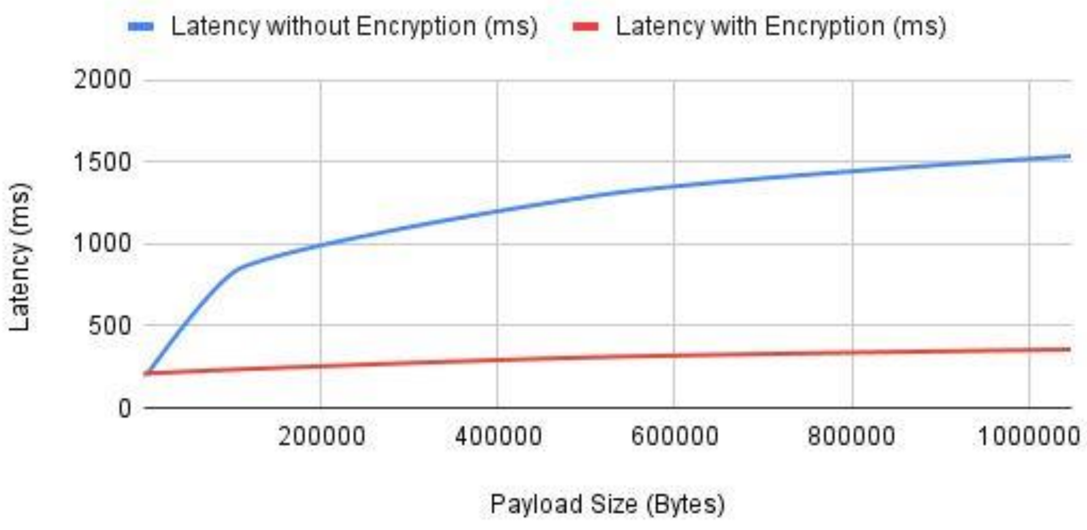


Figure 17 Average Latency vs Payload Size (India-based clients)

The graph presented above illustrates the latency for increasing payload size for clients located in India, with and without the E2EE feature enabled. Interestingly, we observed a significant increase in latency when we disabled E2EE, which came as a surprise.

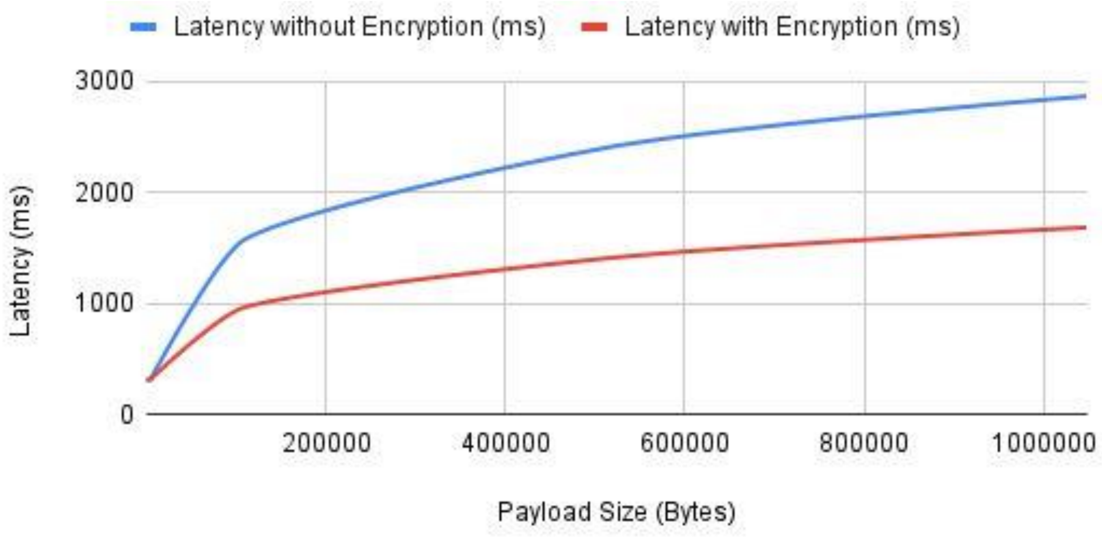


Figure 18 Average Latency vs Payload Size (US-based clients)

The graph showcases the impact of payload size on latency for clients located in the United States with and without E2EE. As anticipated, we observed an increase in latency as the payload size increased. Moreover, when we disabled E2EE, we discovered a noticeable increase in latency.

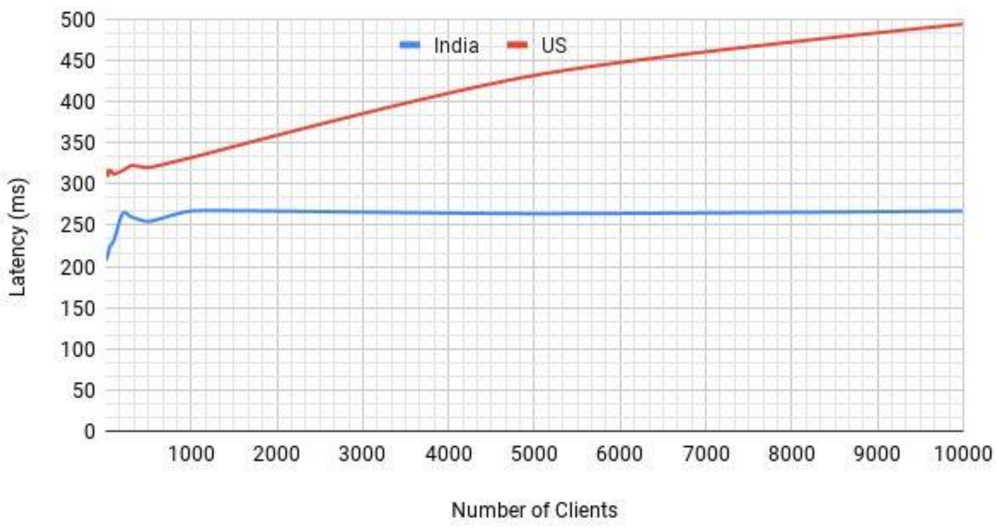


Figure 19 Average Latency v/s Number of Clients

To evaluate the performance and latency of our WebSocket-based service, we conducted testing with an increasing number of clients from two countries, namely India and the United States. Upon analyzing the average latency versus the number of clients, we noticed a consistent increase in latency as the number of clients increased. This indicates that the performance of our application may be affected when subjected to a high number of concurrent users.

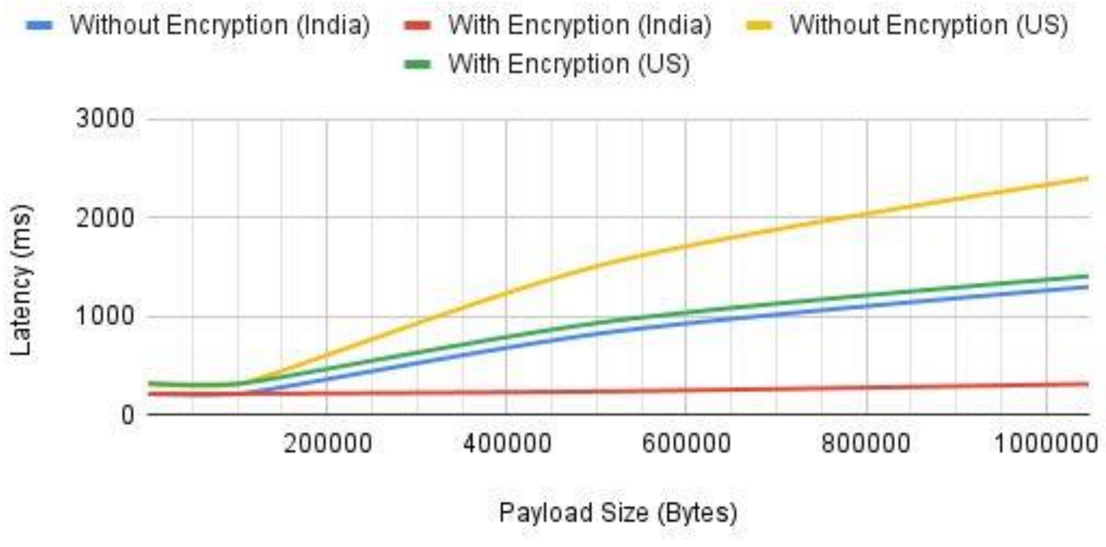


Figure 20 Comparing Latencies of US and India based Clients with increasing payload size

In the above graph, we have a plot of Latency(ms) vs Payload Size(bytes) in which we can observe that the latency is increasing at a faster rate for both the countries without encryption and the latency is increasing at a slower rate with encryption, which is much better as compared to without encryption.

Chapter 9 Deployment Strategies

We have three modules in this project which are required to run constantly to provide seamless service to our customers. Those modules are ew-Service, ew-frontend, and ew-backend. All these modules are connected to the Database (MongoDB) and an in-memory database (Redis).

There were multiple cloud providers available for this purpose such as GCP (Google Cloud Platform), AWS(Amazon Web Services), Azure, and Alibaba but we have selected Microsoft Azure as a cloud service provider because it is a popular cloud platform that offers a wide range of services and features which helped us build, deploy, and manage applications. It provided a flexible and reliable infrastructure that allows us to quickly deploy and scale our application. Github Student Account also offered Student Benefits on this cloud provider and hence we opted for this.

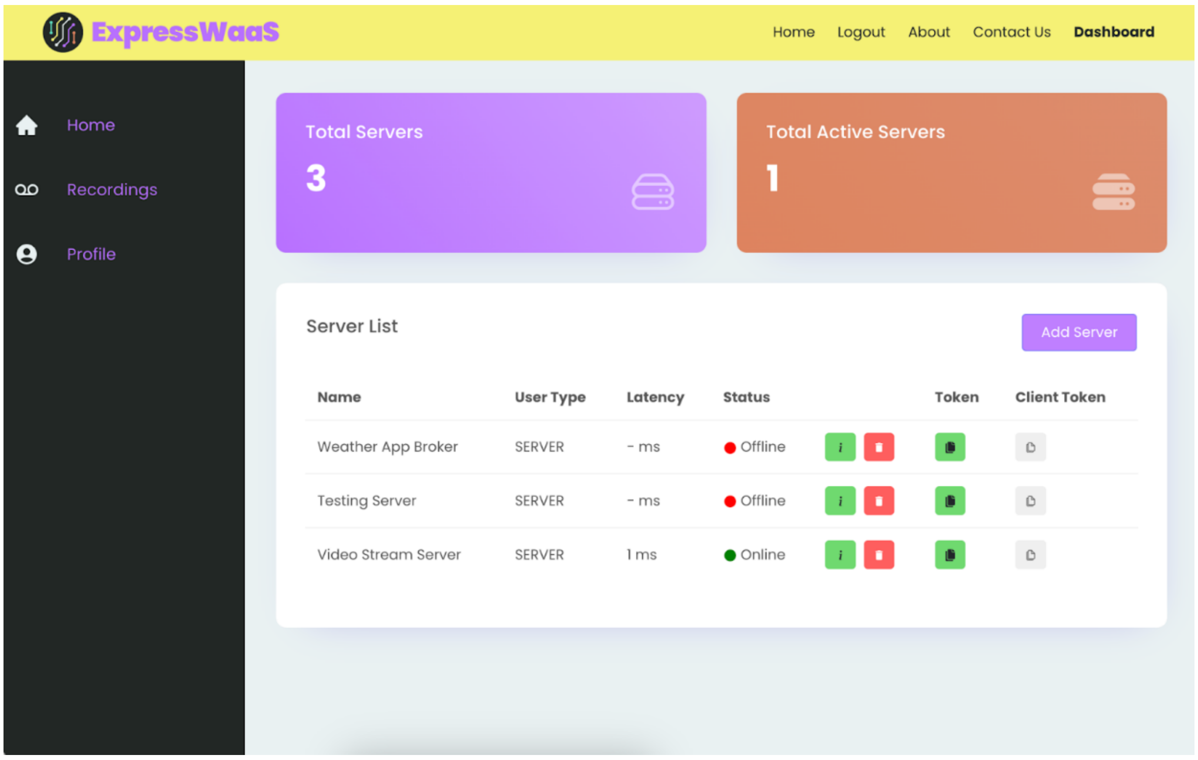
There were multiple VM types available on the Microsoft Azure platform but we have selected the VM with the following configuration:
CPU: 2vCP
Ram: 8 GB
Storage: 30 GB (SSD)

We have considered this configuration in order to provide our service to multiple clients simultaneously.

After getting our server credentials from the cloud provider we have used ssh-agent to connect to the remote server. We have performed the following steps to deploy our application on the server.

1. Installation of mongodb (version >= 4.4) database.
2. Securing the database with the password and creating collections.
3. Installing redis (version >= 6.4).
4. Installing & configuring nginx for reverse proxy as every application should be accessible via port 80 only.
5. Cloning the git repositories of ew-service, ew-backend and ew-frontend module.
6. Installing pm2 to run our complete stack of applications.

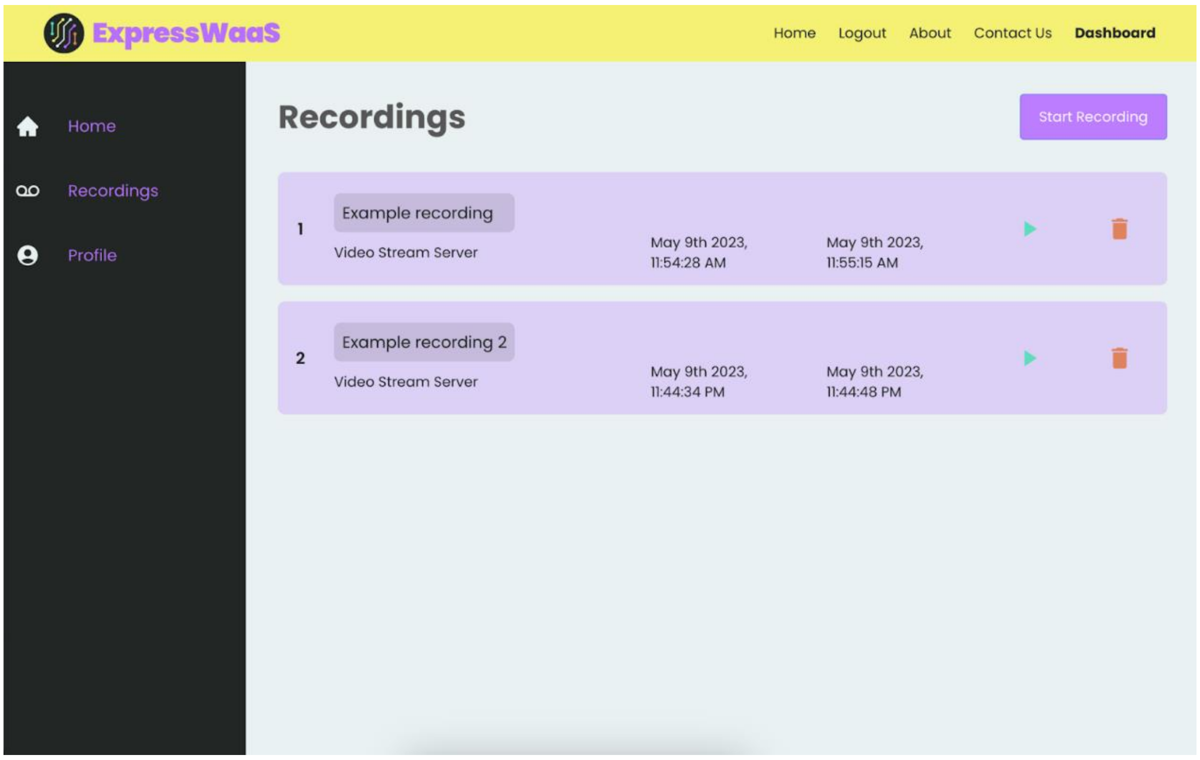
Chapter 10 Result and Analysis



Name	User Type	Latency	Status	Token	Client Token
Weather App Broker	SERVER	- ms	Offline		
Testing Server	SERVER	- ms	Offline		
Video Stream Server	SERVER	1 ms	Online		

Figure 21 Dashboard - Home Page

The dashboard of the ExpressWaas application provides a comprehensive overview of the application's status and allows users to easily manage and monitor the servers that are available in the application. Total Servers displays the total number of servers that are currently available in the application. Server List displays the list of available servers, along with their name and type. In this case, there are two servers available, namely, Weather App Broker and Testing Server. Total active servers represent the total number of servers that are currently active in the application.

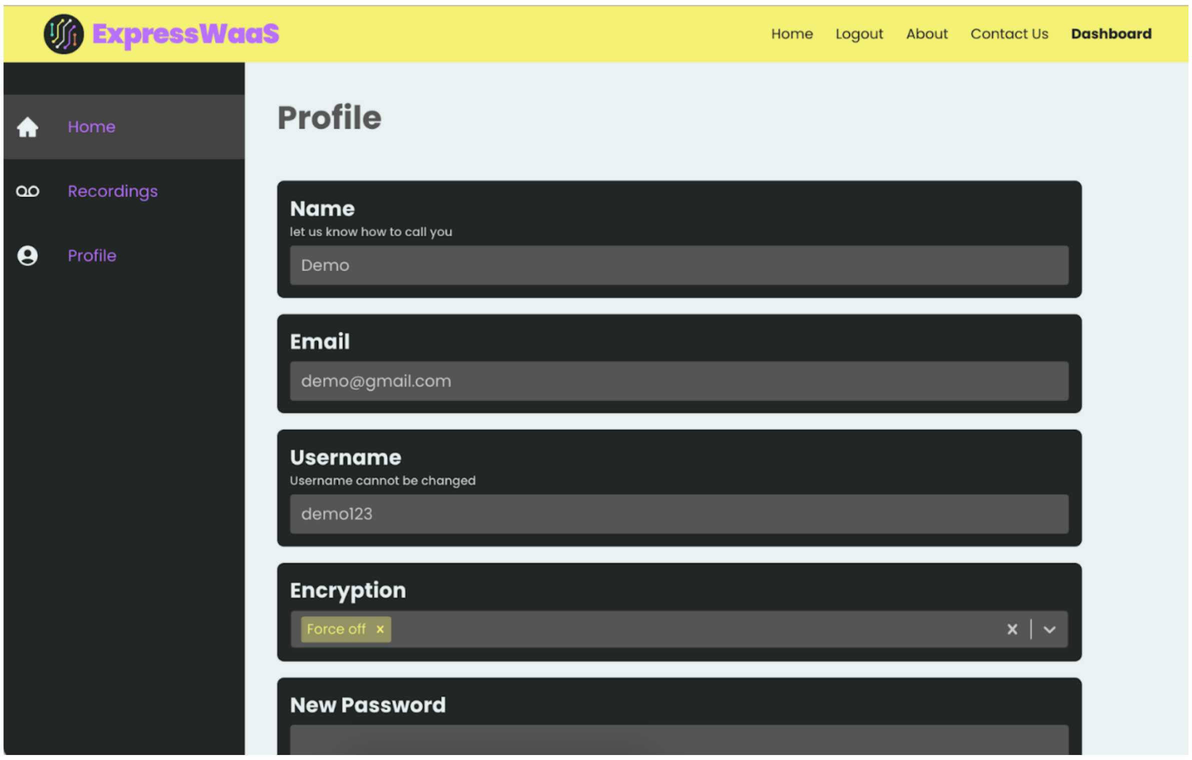


Recordings

1	Example recording Video Stream Server	May 9th 2023, 11:54:28 AM	May 9th 2023, 11:55:15 AM
2	Example recording 2 Video Stream Server	May 9th 2023, 11:44:34 PM	May 9th 2023, 11:44:48 PM

Figure 22 Event Recording Page

The Recordings section is the one that displays a list of previously recorded sessions. Users can review these recordings to analyze the performance of their servers or troubleshoot issues that may have occurred during a session. The "Example recording" and "Example recording 2" are two of the recordings that have been saved in the system. It allows users to keep track of server performance, troubleshoot issues, and take corrective action when necessary.



Profile

Name
let us know how to call you
Demo

Email
demo@gmail.com

Username
Username cannot be changed
demo123

Encryption
Force off

New Password

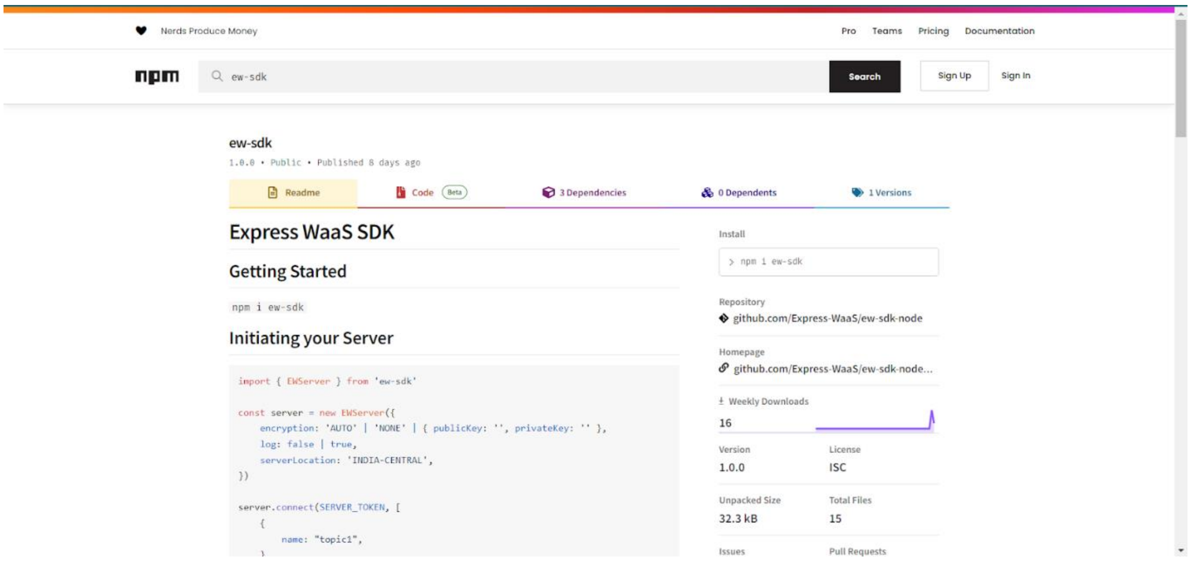
Figure 23 Dashboard - View and Edit Page

The profile section shows the details of the user and allows the user to update the specific fields related to the application.

NPM package - ew-sdk

We have also published the Node SDK package on NPM under the name "[ew-sdk](https://www.npmjs.com/package/ew-sdk)".

Link:- https://www.npmjs.com/package/ew-sdk



ew-sdk
1.0.0 • Public • Published 8 days ago

Express WaaS SDK

Getting Started

```
npm i ew-sdk
```

Initiating your Server

```
import { EWServer } from 'ew-sdk'

const server = new EWServer({
    encryption: 'AUTO' | 'NONE' | { publicKey: '', privateKey: '' },
    log: false | true,
    serverLocation: 'INDIA-CENTRAL',
})

server.connect(SERVER_TOKEN, [
    {
        name: "topic1",
    }
```

Install
> npm i ew-sdk

Repository
github.com/Express-WaaS/ew-sdk-node

Homepage
github.com/Express-WaaS/ew-sdk-node...

Weekly Downloads
16

Version	License
1.0.0	ISC
Unpacked Size	Total Files
32.3 kB	15
Issues	Pull Requests

Figure 24 Screenshot of NPM package page

Application

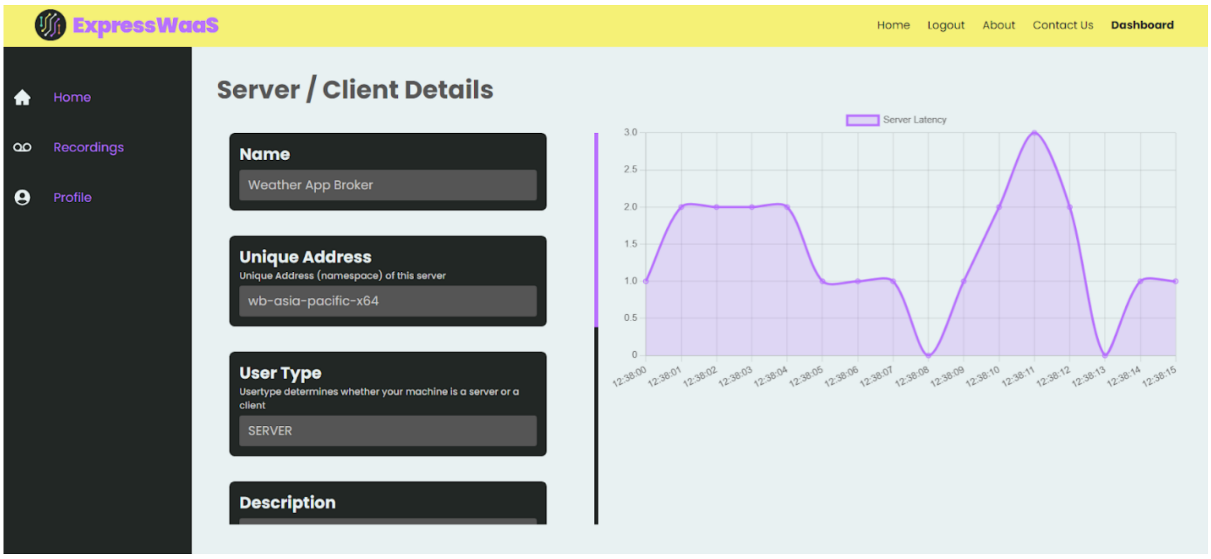
ExpressWaaS is a robust Websocket as a Service solution. It makes it simple for developers to incorporate real-time functionality into their applications. It includes features such as Custom Authentication, End-to-End Encryption, Record and Replay Events, and Pub/Sub for real-time data updates. With the help of the NPM SDK, users can easily integrate their applications with ExpressWaaS.

Demo Applications

To show the functionality of our application, we created two demo applications. Below are the details and screenshots of both the apps.

1. Weather Broker Application

   This demo application is a representation of the real IOT application. There is a broker server and three sensory nodes (Temperature node, Pressure node, and Humidity sensor) in the setup. In addition to that, we created a 'phone.js' file that depicts the weather app on a mobile phone. This phone script will be executed three times, each time subscribing to a different group of weather parameters. Screenshots with self-explanatory captions are attached below.



Server / Client Details

Name

Weather App Broker

Unique Address

Unique Address (namespace) of this server

wb-asia-pacific-x64

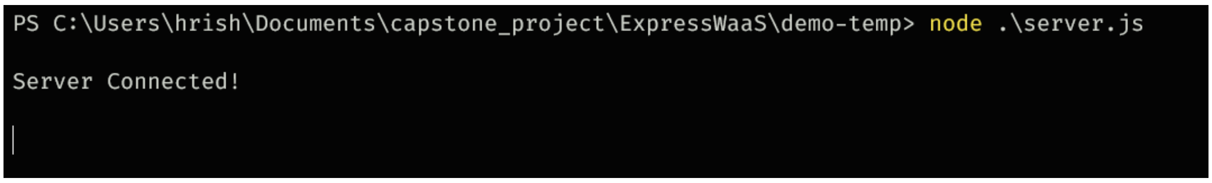
User Type

Usertype determines whether your machine is a server or a client

SERVER

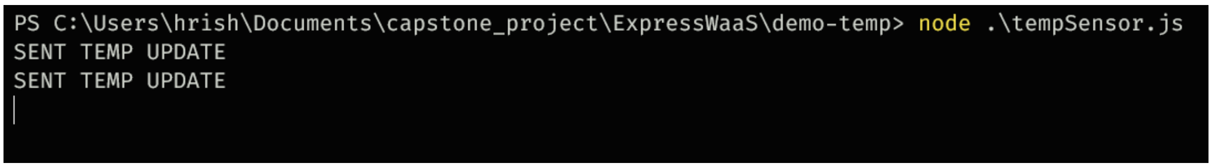
Description

Figure 25 Server Details Page of Weather App Broker



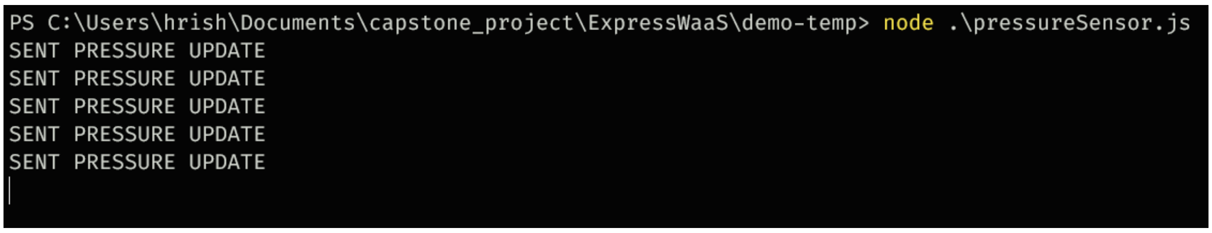
```
PS C:\Users\hrish\Documents\capstone_project\ExpressWaaS\demo-temp> node .\server.js

Server Connected!
```

Figure 26 Weather Broker Server Application



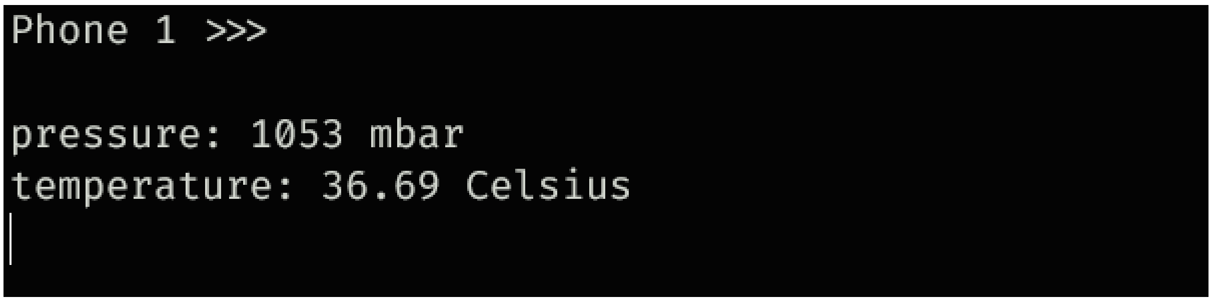
```
PS C:\Users\hrish\Documents\capstone_project\ExpressWaaS\demo-temp> node .\tempSensor.js
SENT TEMP UPDATE
SENT TEMP UPDATE
```

Figure 27 Temperature Publishing Node



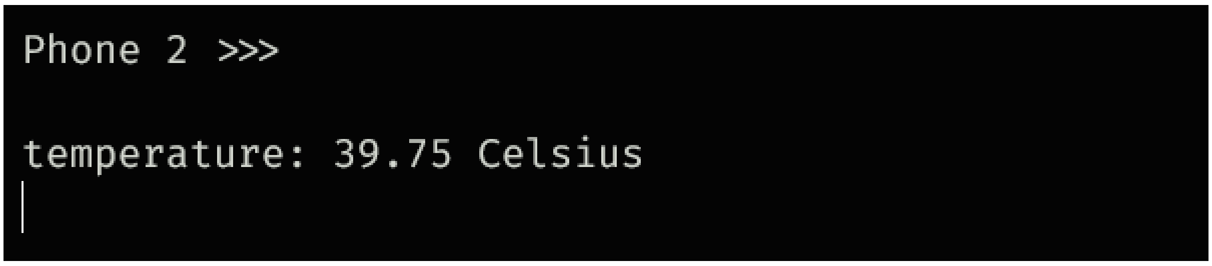
```
PS C:\Users\hrish\Documents\capstone_project\ExpressWaaS\demo-temp> node .\pressureSensor.js
SENT PRESSURE UPDATE
SENT PRESSURE UPDATE
SENT PRESSURE UPDATE
SENT PRESSURE UPDATE
SENT PRESSURE UPDATE
```

Figure 28 Pressure Publishing Node



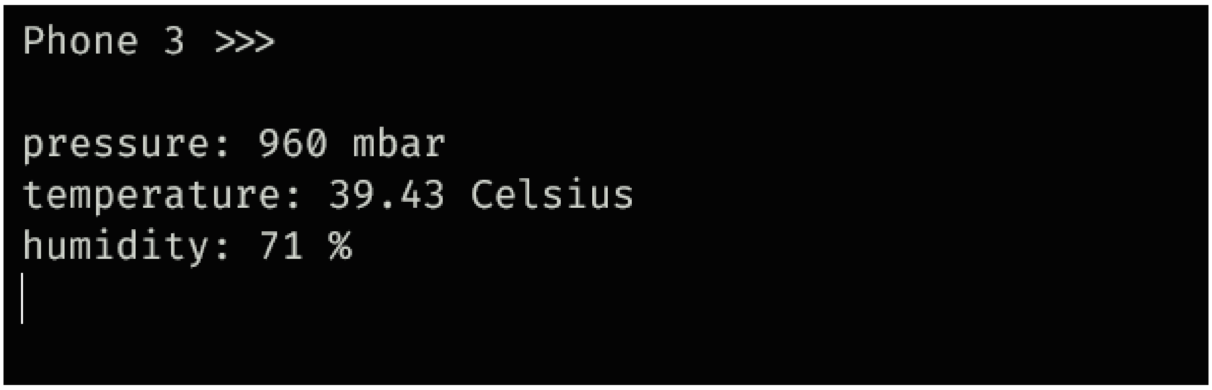
```
Phone 1 >>>

pressure: 1053 mbar
temperature: 36.69 Celsius
```

Figure 29 Phone subscribed to Pressure and Temperature



```
Phone 2 >>>

temperature: 39.75 Celsius
```

Figure 30 Phone subscribed to Temperature

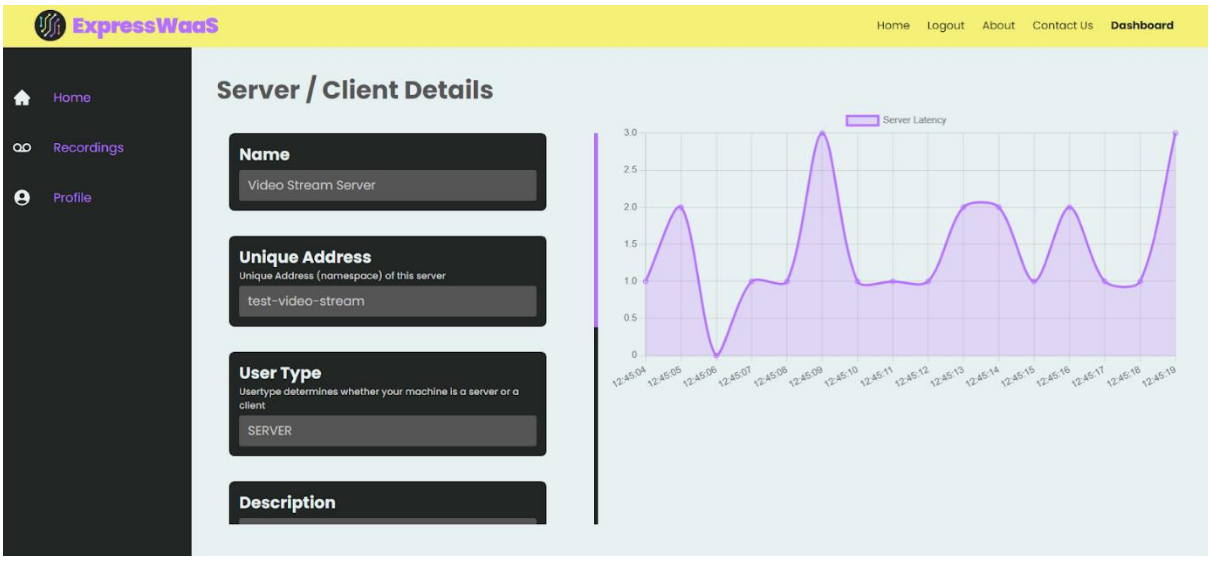


```
Phone 3 >>>

pressure: 960 mbar
temperature: 39.43 Celsius
humidity: 71 %
```

Figure 31 Phone subscribed to Temperature, Pressure and Humidity

2. Webcam Streaming Application

This sample application is a simple webcam-sharing application. It is a Node.js web application with a server and a single-page client. Clients will join different rooms based on the URL parameter. People in the same room will be able to see the person streaming's webcam. It demonstrates how efficiently our service can stream video with a few additional video streaming algorithms in addition to E2EE. Screenshots of this web application are attached below.



Server / Client Details

Name: Video Stream Server

Unique Address
Unique Address (namespace) of this server
test-video-stream

User Type
Usertype determines whether your machine is a server or a client
SERVER

Description

Figure 32 Server Details Page of Webcam Streaming Server

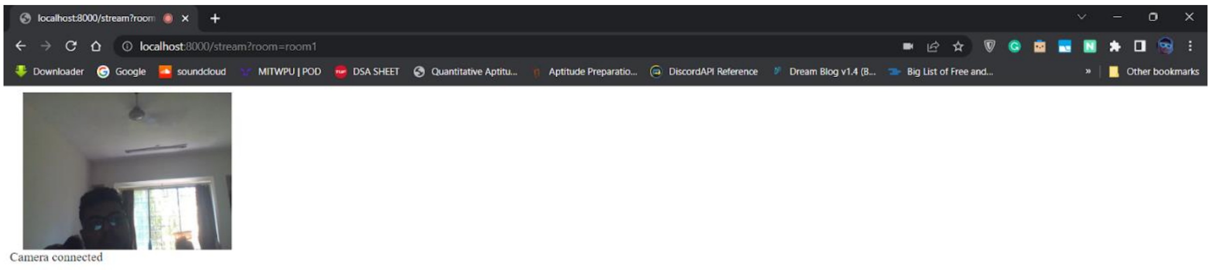


Figure 33 Webcam Streaming Page

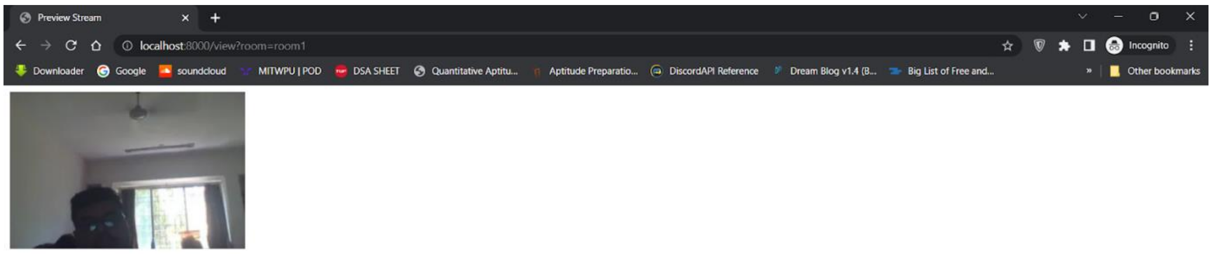


Figure 34 Webcam Viewing Page

Conclusion

Based on the results of the testing and analysis, we can conclude that the system we developed for real-time communication in web applications offers significant benefits. The use of WebSocket significantly reduces latency and improves the application's overall performance and scalability.

The results of the performance testing show that as the number of clients increases, the average latency of the system using WebSocket remains relatively low, whereas the system using HTTP rapidly increases in latency. The latency comparison between HTTP and HTTPOverWS highlights the benefits of using our Service, as it demonstrates that HTTPOverWS has significantly lower latency when compared to HTTP with persistent connections.

The system's limitation is that additional algorithms must be implemented when transferring large files or creating a video stream application. As a result, future work will include incorporating these algorithms into the SDK. Another restriction is that the SDK can only transfer JSON objects, so even if we only need to send a single string, we must wrap it in a JSON object. Another issue with the system is that when it is restarted, it regenerates the public and private key pairs. As a result, while recording events, we must still figure out how to store encrypted data so that it can be decrypted during replay even if the keys are regenerated. Another future work includes integrating with third-party services like Gmail and GitHub.

Future Prospects of Project

- Incorporating algorithms related to transferring large files or creating a video stream application into the SDK.
- Integrating with third-party services like Gmail and GitHub.
- Creating SDK with multiple languages hence increasing support for multiple languages
- Adding support for multiple WaaS Servers at different location reducing the latency for users.

References

[1] K. Ogundeyi and C Yinka-Banjo, "WebSocket in real time application," Nigerian Journal of Technology, no. Vol. 38 No. 4 (2019), p. 11, 2019.

[2] Q. Liu, G. Yang, R. Zhao and Y. Xia, "Design and Implementation of Real-time Monitoring System for Wireless Coverage Data Based on WebSocket," in 2018 IEEE 3rd International Conference on Cloud Computing and Internet of Things (CCIOT), 2018.

[3] M. Tomasetti, "An Analysis of the Performance of Websockets in Various Programming Languages and Libraries," SSRN E Journal, p. 7, 2021.

[4] P. Murley, Z. Ma, J. Mason, M. Bailey and A. Kharraz, "WebSocket Adoption and the Landscape of the Real-Time Web," in WWW '21: Proceedings of the Web Conference 2021, 2021.

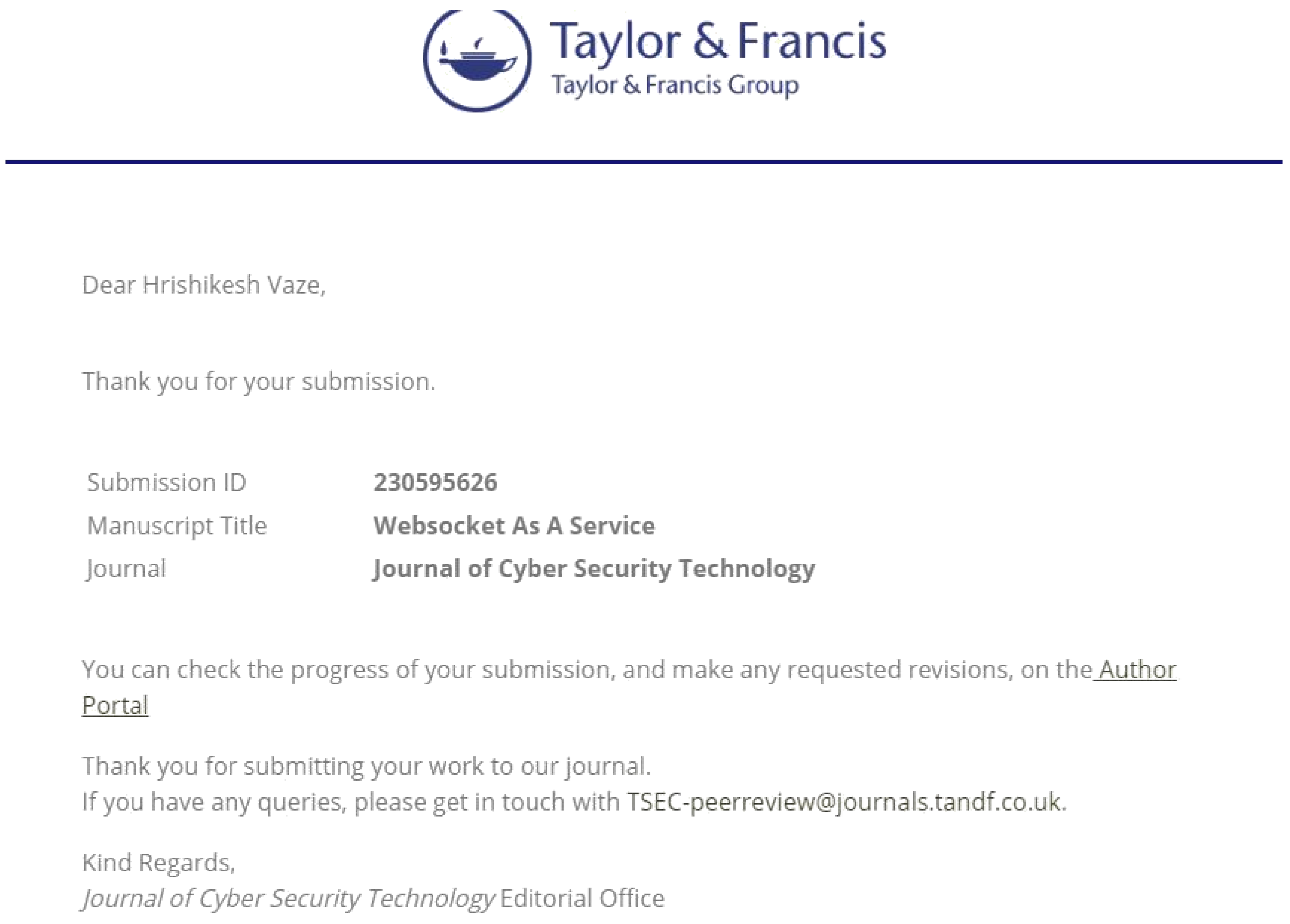
[5] D. G. Puranik, D. C. Feiock and J. H. Hill, "Real-Time Monitoring using AJAX and WebSockets," in International Conference and Workshop on Engineering of Computer-Based Systems, 2013.

[6] M. R. Agrawal, R. Rathore, S. Jain, SatyamYadav, V. Singh, Y. Mishra and N. P. Singh, "CASE STUDY ON WHATSAPP END TO END ENCRYPTION," International Research Journal of Modernization in Engineering Technology and Science , vol. 02, no. 04, p. 6, 2020.

[7] A. Rahmatulloh, I. Darmawan and R. Gunawan, "Performance Analysis of Data Transmission on WebSocket for Real-time Communication," in International Conference on QiR (Quality in Research), 2019.

Publication Details

We have published our research paper named **Websocket As A Service** in **Journal of Cyber Security Technology** on 22 May 2023.



Taylor & Francis
Taylor & Francis Group

Dear Hrishikesh Vaze,

Thank you for your submission.

Submission ID	**230595626**
Manuscript Title	**Websocket As A Service**
Journal	**Journal of Cyber Security Technology**

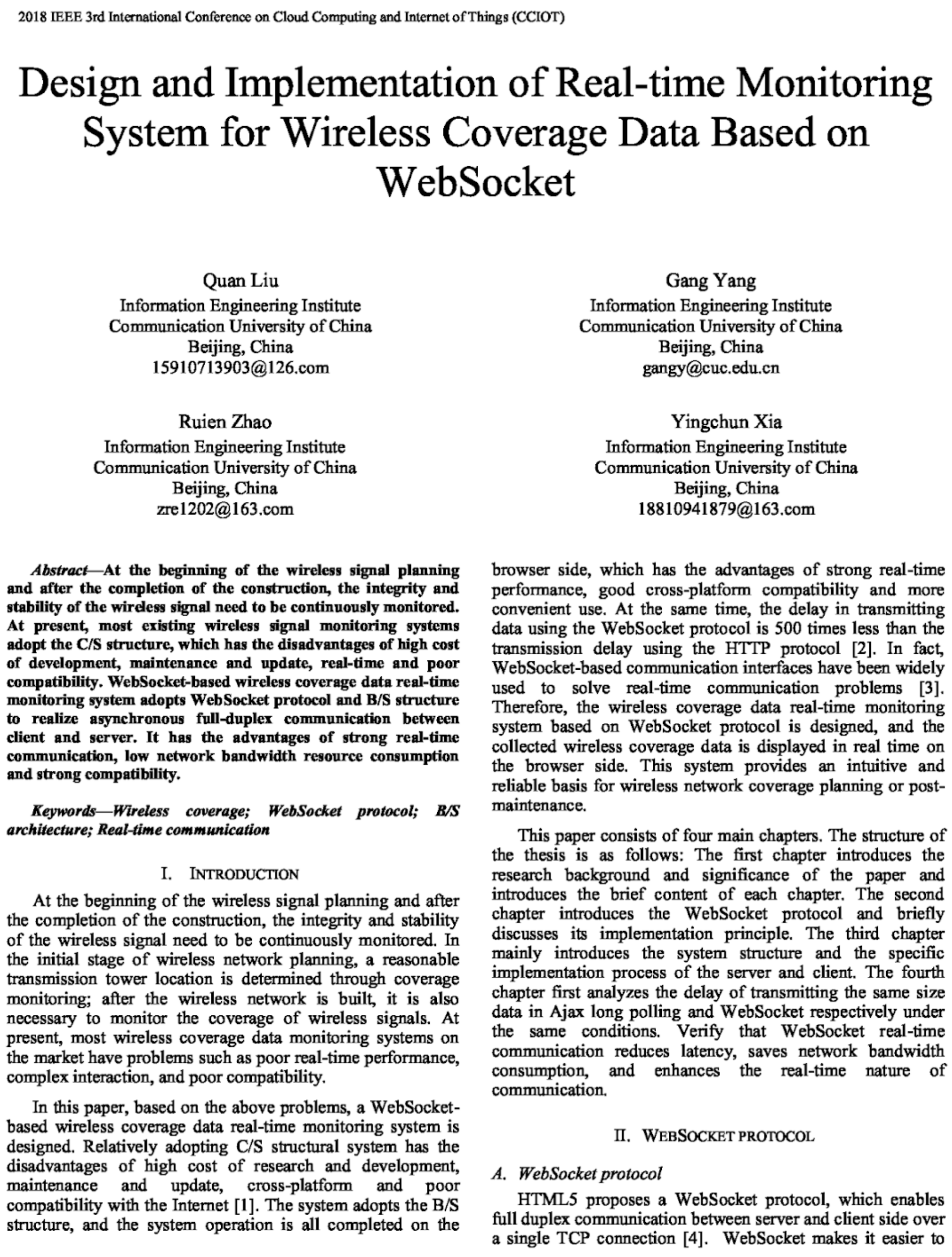
You can check the progress of your submission, and make any requested revisions, on the Author Portal

Thank you for submitting your work to our journal.
If you have any queries, please get in touch with TSEC-peerreview@journals.tandf.co.uk.

Kind Regards,
Journal of Cyber Security Technology Editorial Office

Appendices

Base Paper



2018 IEEE 3rd International Conference on Cloud Computing and Internet of Things (CCIOT)

Design and Implementation of Real-time Monitoring System for Wireless Coverage Data Based on WebSocket

Quan Liu
Information Engineering Institute
Communication University of China
Beijing, China
15910713903@126.com

Gang Yang
Information Engineering Institute
Communication University of China
Beijing, China
gangy@cuc.edu.cn

Ruien Zhao
Information Engineering Institute
Communication University of China
Beijing, China
zre1202@163.com

Yingchun Xia
Information Engineering Institute
Communication University of China
Beijing, China
18810941879@163.com

***Abstract*—At the beginning of the wireless signal planning and after the completion of the construction, the integrity and stability of the wireless signal need to be continuously monitored. At present, most existing wireless signal monitoring systems adopt the C/S structure, which has the disadvantages of high cost of development, maintenance and update, real-time and poor compatibility. WebSocket-based wireless coverage data real-time monitoring system adopts WebSocket protocol and B/S structure to realize asynchronous full-duplex communication between client and server. It has the advantages of strong real-time communication, low network bandwidth resource consumption and strong compatibility.**

Keywords*—Wireless coverage; WebSocket protocol; B/S architecture; Real-time communication

I. Introduction

At the beginning of the wireless signal planning and after the completion of the construction, the integrity and stability of the wireless signal need to be continuously monitored. In the initial stage of wireless network planning, a reasonable transmission tower location is determined through coverage monitoring; after the wireless network is built, it is also necessary to monitor the coverage of wireless signals. At present, most wireless coverage data monitoring systems on the market have problems such as poor real-time performance, complex interaction, and poor compatibility.

In this paper, based on the above problems, a WebSocket-based wireless coverage data real-time monitoring system is designed. Relatively adopting C/S structural system has the disadvantages of high cost of research and development, maintenance and update, cross-platform and poor compatibility with the Internet [1]. The system adopts the B/S structure, and the system operation is all completed on the browser side, which has the advantages of strong real-time performance, good cross-platform compatibility and more convenient use. At the same time, the delay in transmitting data using the WebSocket protocol is 500 times less than the transmission delay using the HTTP protocol [2]. In fact, WebSocket-based communication interfaces have been widely used to solve real-time communication problems [3]. Therefore, the wireless coverage data real-time monitoring system based on WebSocket protocol is designed, and the collected wireless coverage data is displayed in real time on the browser side. This system provides an intuitive and reliable basis for wireless network coverage planning or post-maintenance.

This paper consists of four main chapters. The structure of the thesis is as follows: The first chapter introduces the research background and significance of the paper and introduces the brief content of each chapter. The second chapter introduces the WebSocket protocol and briefly discusses its implementation principle. The third chapter mainly introduces the system structure and the specific implementation process of the server and client. The fourth chapter first analyzes the delay of transmitting the same size data in Ajax long polling and WebSocket respectively under the same conditions. Verify that WebSocket real-time communication reduces latency, saves network bandwidth consumption, and enhances the real-time nature of communication.

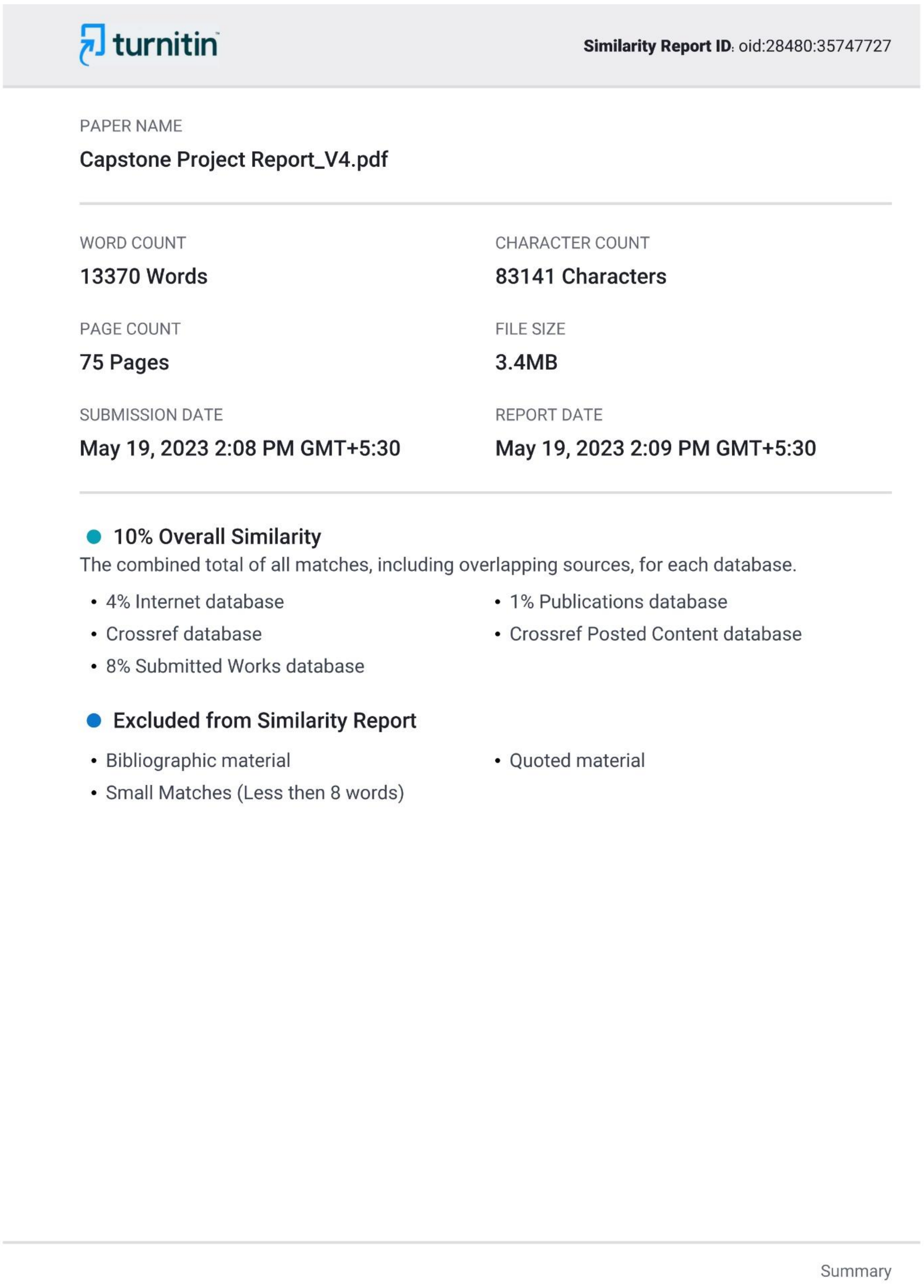
II. WebSocket Protocol

A. WebSocket protocol

HTML5 proposes a WebSocket protocol, which enables full duplex communication between server and client side over a single TCP connection [4]. WebSocket makes it easier to

Figure 35 Base Paper

Plagiarism Report from any Open Source


turnitin

Similarity Report ID: oid:28480:35747727

PAPER NAME

Capstone Project Report_V4.pdf

WORD COUNT	CHARACTER COUNT
13370 Words	83141 Characters
PAGE COUNT	FILE SIZE
75 Pages	3.4MB
SUBMISSION DATE	REPORT DATE
May 19, 2023 2:08 PM GMT+5:30	May 19, 2023 2:09 PM GMT+5:30

● **10% Overall Similarity**

The combined total of all matches, including overlapping sources, for each database.

- 4% Internet database
- 1% Publications database
- Crossref database
- Crossref Posted Content database
- 8% Submitted Works database

● **Excluded from Similarity Report**

- Bibliographic material
- Quoted material
- Small Matches (Less then 8 words)

Summary

Individual Contribution

Problem Statement: - The problem is the need to develop a module that can handle multiple client socket connections, record events, and provide encryption for data integrity. The module must efficiently handle high-traffic volumes, deliver messages on time, and ensure a seamless and secure experience for the users.

Name of Student: - Hrishikesh Vaze

Module Title: - ExpressWaaS Service, NODE SDK

Project's Module Objectives: -

The goal of developing this module for the application is to create a service that handles multiple client socket connections. The module will handle events like socket connections and disconnections, message broadcasting, and other related tasks. It should also be able to record these events for audit purposes and replay them as needed based on user preferences. The module should be able to handle high volumes of traffic while also ensuring that messages are delivered on time and efficiently. It should provide encryption to ensure the data's integrity.

Project's Module Scope: -

This module will provide a flexible solution for dealing with multiple socket clients while maintaining application performance. It will be able to encrypt messages, store event data in a database, and provide event data access via a RESTful API. It will be broken down into five modules: Event Recorder, Event Emitter, Webhook, RestAPI, and HttpOverWS. It will communicate with the front end via the back end. Every connection received by the module will be authenticated using JWT tokens.

Project's Module(s)

Software requirements:

- Node.js
- Socket.io
- Redis

Module Interfaces: Socket.io

Module Dependencies: ew-backend

Module Design:

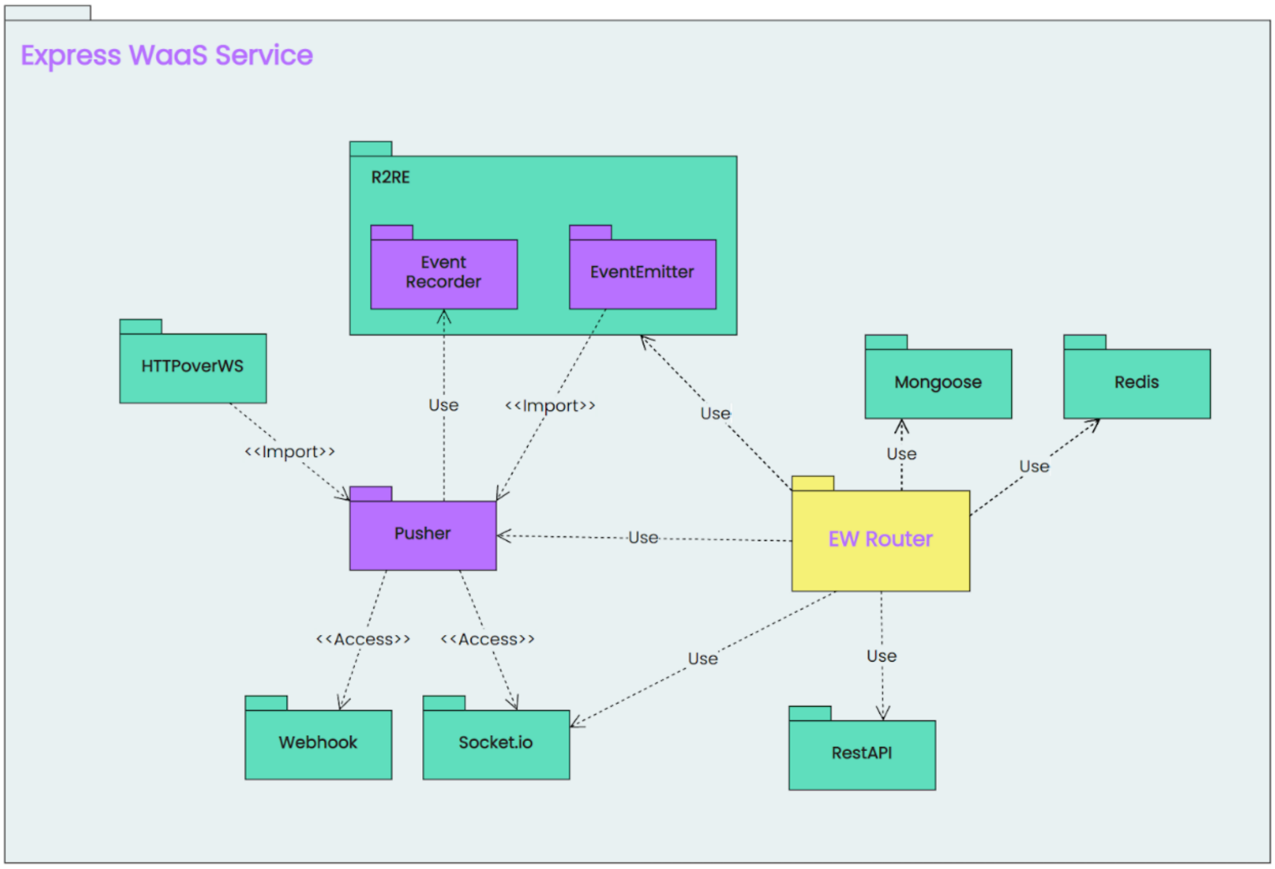


Figure 36 Package Diagram for Express WaaS Service

Module Implementation:

This module was built with Node.js and Socket.io. This module's development followed a traditional agile methodology. Event Recorder, HTTP over WS Module, Create Redis Adapter, Webhook Module, Event Emitter Module, RestAPI Module, Record and Replay, Custom Authentication, End to End Encryption, Pusher Module, and Socket.io Module were all broken down into small, manageable tasks. This module was implemented after all of the tasks were defined.

Problem Statement: - To create an interface that acts as middleware for the frontend and the service module. The module should have well-defined activities such as login, register, and other endpoints. The difficulty is to manage these duties effectively while also developing a scalable and robust module that satisfies the needs of the users and provides a consistent experience.

Name of Student: - Om Patel

Module Title: - ExpressWaaS Backend

Project's Module Objectives: -

The objective of creating this backend module for the application is to provide a layer of abstraction between the front end and services module, which allows improved maintainability of the overall application. The backend module serves as a gateway for communication between the front end and services, handling tasks such as authentication, and data management. By centralizing the handling of sensitive data of the ExpressWaaS service, it will become easier to implement security measures such as authentication and authorization. This will help in protecting data from unauthorized access, modification, or destruction.

Project's Module Scope: -

The goal of developing this backend module for the application is to provide a layer of abstraction between the front end and the services module, allowing for improved overall application maintainability. The backend module acts as a bridge between the front end and the services, handling tasks such as authentication and data management. By centralizing the ExpressWaaS service's handling of sensitive data, security measures such as authentication and authorization will be easier to implement. This will aid in data security by preventing unauthorized access, modification, or destruction.

Project's Module(s)

Software requirements:

- Socket.io
- Express
- Node.js

Module Interfaces: RestApi, Web-Socket

Module Dependencies: ew-service

Module Design:

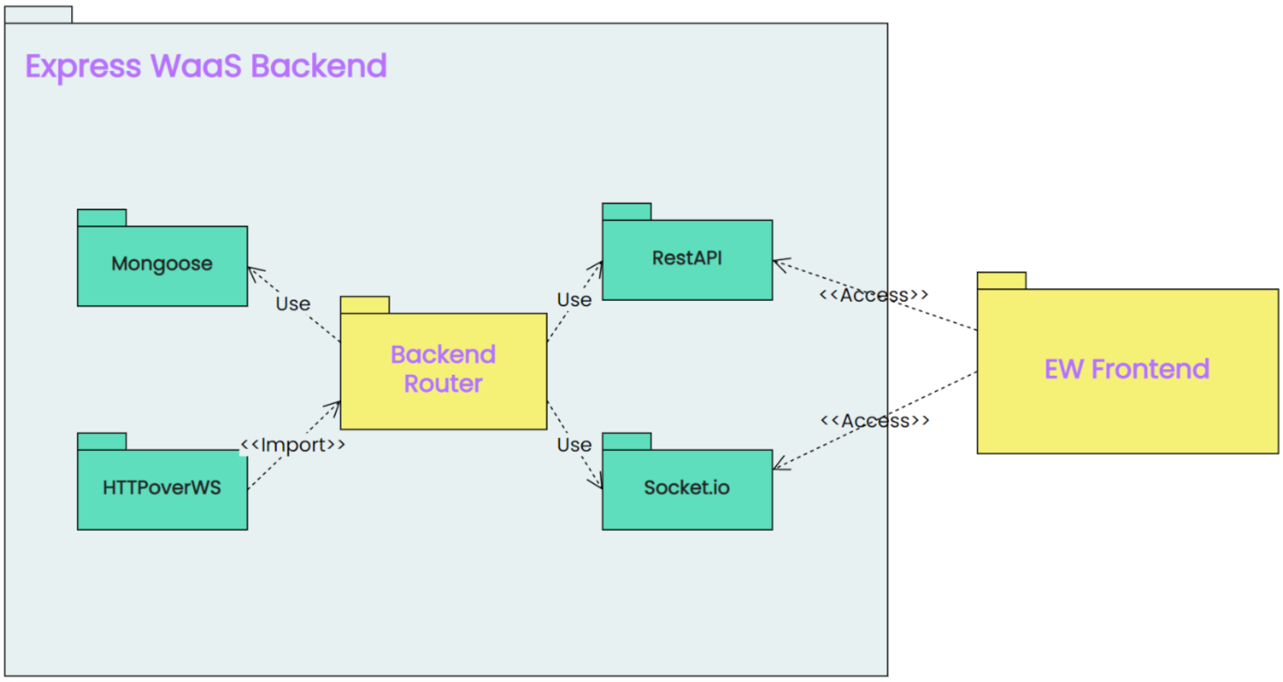


Figure 37 Package Diagram for Backend Module

Module Implementation:

This module was built with Node.js and Socket.IO. This module's development followed a traditional agile methodology. The module was divided into small, manageable tasks such as Initialising Backend Server, login, Register & Check Username Endpoint, Trusted Server/Client Endpoint, Dashboard Endpoint, and Record & Replay Endpoint. We began working on the tasks after they were all defined.

Problem Statement: - The problem is that there is no frontend module available for users to interact with our service, leading to usability and functionality issues, as well as user frustration. A frontend module is necessary to provide a user-friendly interface and enable users to access and use the service effectively.

Name of Student: - Aniruddha Shende

Module Title: - ExpressWaaS Frontend

Project's Module Objectives:

The goal of EW front-end development is to create an intuitive and responsive user interface that allows users to connect to the real-time communication platform in a seamless and efficient manner. Frontend development must prioritize providing an engaging and fluid user experience while leveraging ExpressWaaS's dependable and scalable infrastructure. The front end also allows users to personalize the service based on their preferences. It is intended to handle errors and report them to users in a clear and concise manner.

Project's Module Scope:

The front-end development scope includes designing and building user interfaces, integrating real-time communication features, ensuring cross-browser compatibility, and optimizing performance and responsiveness. Frontend development must also be consistent with the overall user experience and business goals of the application or system. To achieve the desired results, the development process must adhere to industry best practices and standards while utilizing cutting-edge technology and tools. Front-end development should be malleable and agile in order to accommodate future updates and changes to the real-time communication infrastructure.

Project's Module(s)

Hardware & Software requirements:

- ReactJS
- Socket.io

Module Interfaces: Dashboard, Machine Package

Module Dependencies: ew-backend

Module Design:

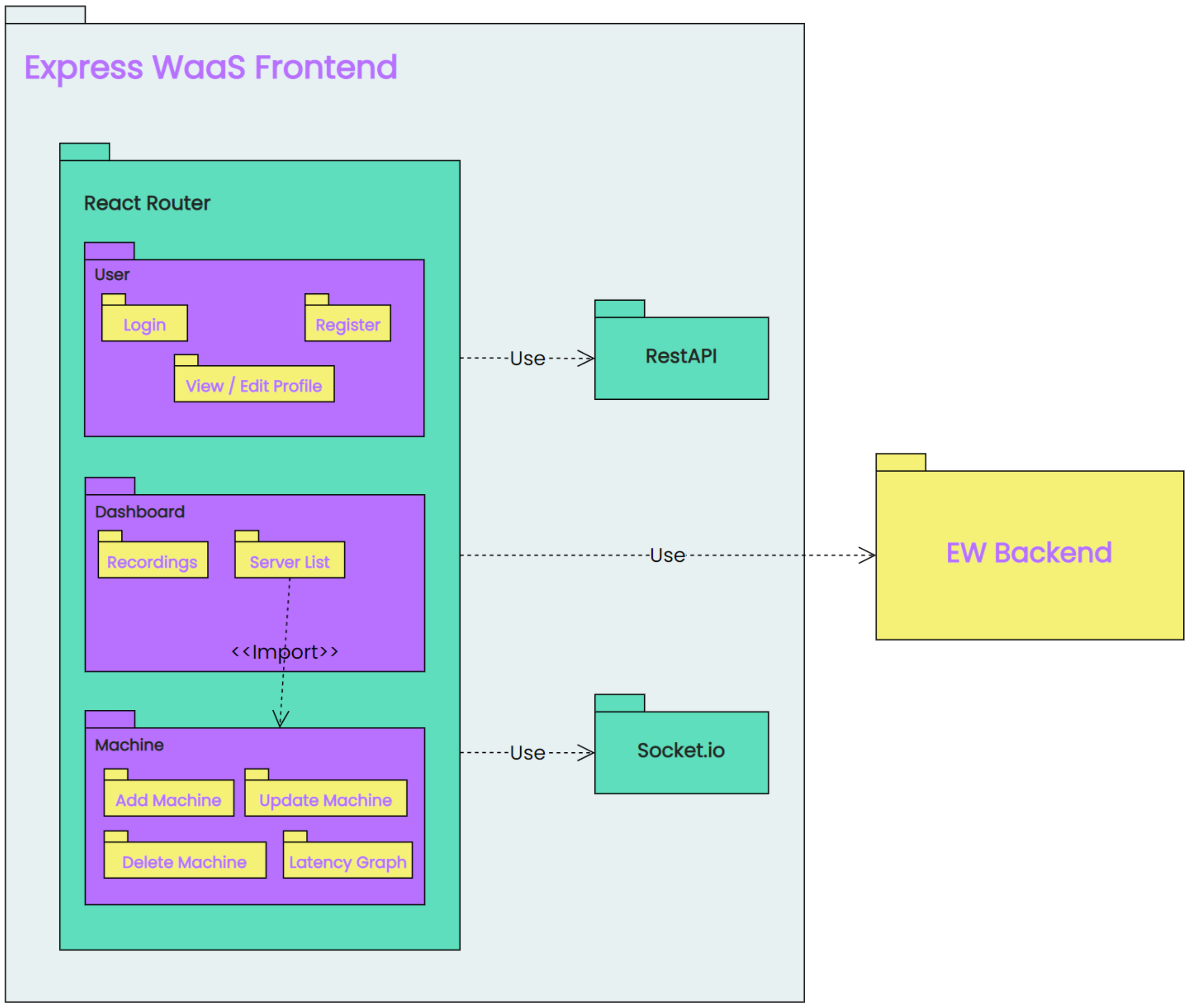


Figure 38 Package Diagram for Frontend Module

Module Implementation:

The Machines module oversees a system's machine setup and performance. It contains features such as adding, updating, and deleting machines. Users can customize the configurations of their servers/machines. The module also includes Latency Graphs for each machine. A software component that provides an overview of a system's performance and activity is the Dashboard module. It shows recordings (the records of events to be replayed later) and server lists (the list of servers or other network devices that are currently connected or available in the system). These features enable users to monitor and manage their system's resources, identify and diagnose problems, and optimize the performance and availability of their system. The Dashboard module increases a system's overall efficiency and effectiveness by improving the user experience.

Problem Statement: - The problem is that there is no frontend module available for users to interact with our service, leading to usability and functionality issues, as well as user frustration. A frontend module is necessary to provide a user-friendly interface and enable users to access and use the service effectively.

Name of Student: - Vartika Katiyar
Module Title: - ExpressWaaS Frontend

Project's Module Objectives:

The goal of EW front-end development is to create an intuitive and responsive user interface that allows users to connect to the real-time communication platform in a seamless and efficient manner. Frontend development must prioritize providing an engaging and fluid user experience while leveraging ExpressWaaS's dependable and scalable infrastructure. The front end also lets users personalize the service based on their preferences. It is intended to handle errors and report them to users in a clear and concise manner.

Project's Module Scope:

The front-end development scope includes designing and building user interfaces, integrating real-time communication features, ensuring cross-browser compatibility, and optimizing performance and responsiveness. Frontend development must also be consistent with the overall user experience and business goals of the application or system. To achieve the desired results, the development process must adhere to industry best practices and standards while utilizing cutting-edge technology and tools. Front-end development should be malleable and agile in order to accommodate future updates and changes to the real-time communication infrastructure.

Project's Module(s)
Software requirements:

- React JS
- Socket.IO

Module Interfaces: User package
Module Dependencies: ew-backend

Module Design:

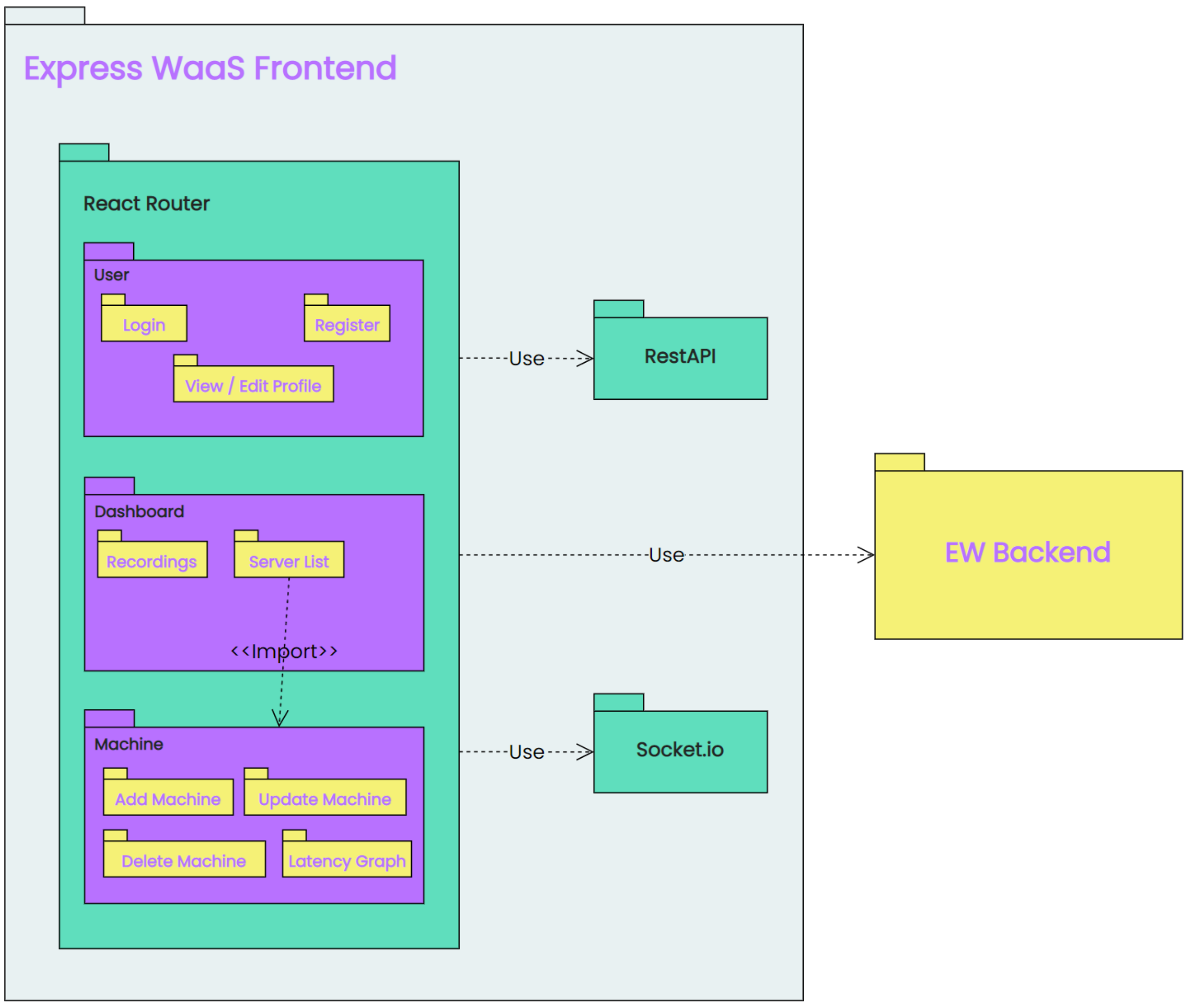


Figure 39 Package Diagram for Frontend Module

Module Implementation:

The user module is a piece of software that handles user accounts and profiles. It typically includes features such as login, registration, and view/edit profile. By entering their email address and password, users can securely access their accounts. The token system is used for authentication. The register feature allows new users to set up an account by providing personal information. The view/edit profile feature allows users to manage their profile information, such as contact information and preferences. The user module is a critical component of any software system that strives to provide its users with a personalized and secure experience.

Project to Outcome mapping

1. To explore the requirements of developers.
2. To study different algorithms for implementing E2EE.
3. To develop an SDK for communicating with the platform.
4. To develop documentation for both SDK and Rest API.
5. To create a user interface for monitoring real-time data.
6. To evaluate the performance of the system.

Sr. No.	PRN No.	Student Name	Project Student Specific Objective	Learning Outcomes mapped	Marks
1	1032190087	Hrishikesh Vaze	To create web sockect as service and sdk with features like Custom Authentication, E2EE, HTTPOverWs, and Record and Replay Event	1,2,3,4,5,6,7,8,9,10,11,12, 13,14,16,17,18	
2	1032190080	Om Patel	To create a backend application for handling frontend requests.	1,2,3,4,5,6,7,8,9,10,11,12, 13,14,16,17,18	
3	1032190079	Aniruddha Shende	To create dashboard module (frontend) for easing task of clients	1,2,3,4,5,6,7,8,9,10,11,12, 13,14,16,17,18	
4	1032190212	Vartika Katiyar	To create user module (frontend) for easing task of clients	1,2,3,4,5,6,7,8,9,10,11,12, 13,14,16,17,18	