**A close-up of a logo

Description automatically generated**

|  |
| --- |
| **Real-Time Collaboration Platform Using WebSockets and Event-Driven Architecture** |

**Christos Stylidis**

SID: 205364504177

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

*Master of Science (MSc) in Information and Communication Systems*

JANUARY 2025  
THESSALONIKI – GREECE

****

|  |
| --- |
| **Real-Time Collaboration Platform Using WebSockets and Event-Driven Architecture** |

**Christos Stylidis**

SID: 205364504177

|  |  |
| --- | --- |
| Supervisor: | Prof. Leonidas Akritidis |
| Supervising Committee Members: | Assoc. Prof. Panayiotis Bozanis  Assist. Prof. Dimitrios Karapiperis |

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

*Master of Science (MSc) in Information and Communication Systems*

JANUARY 2025  
THESSALONIKI – GREECE

Abstract

This dissertation explores the development of a **real-time ticketing system** leveraging **WebSockets and an event-driven architecture**. The platform facilitates instant updates, allowing users, agents, and admins to seamlessly interact with ticket data. The focus of this project was to address challenges in real-time data synchronization, ensuring scalability and maintaining role-base access control.

The system was implemented using **Node.js, Express.js, MongoDB, and Kafka, Authentication Setup** (JWT) to enable real-time updates for ticket creation, assignment, and status changes. Websocket integration ensures live feedback for dashboard operation, while Kafka serves as the backbone for reliable message queueing between components. The codebase includes role-based dashboards for users, agents, and admins, supporting dynamic operations like viewing, creating and updating tickets.

Acknowledgment is extended to my supervisor, **Leonidas Akritidis**, and committee members **Panayiotis Bozanis** and **Dimitrios Karapiperis** for their valuable insights and support. Gratitude is also expressed to the **IHU faculty and peers** for their feedback encouragement throughout this project.

Christos Stylidis

23.12.2024

Table of Contents

Table of Contents

[Abstract 3](#_Toc186921207)

[Table of Contents 4](#_Toc186921208)

[1 Introduction 5](#_Toc186921209)

[1.1 Overview 5](#_Toc186921210)

[1.2 Motivation 6](#_Toc186921211)

[1.3 Objectives 7](#_Toc186921212)

[1.4 Structure 8](#_Toc186921213)

[2 Literature Review 9](#_Toc186921214)

[2.1 Overview 9](#_Toc186921215)

[2.2 Related Work 10](#_Toc186921216)

[2.3 Critical Analysis 10](#_Toc186921217)

[**2.3.1** **Strength of Existing Systems** 10](#_Toc186921218)

[**2.3.2** **Weaknesses of Existing Systems** 12](#_Toc186921219)

[**2.3.3** **Proposed Improvements** 14](#_Toc186921220)

[**2.4** **Summary and Conclusion** 16](#_Toc186921221)

[3 System Design and Architecture 17](#_Toc186921222)

[3.1 Overview 17](#_Toc186921223)

[3.2 Architectural Design 18](#_Toc186921224)

[3.3 Detailed Workflow and Interactions 19](#_Toc186921225)

# **Introduction**

## **Overview**

The growing reliance on real-time communication systems in various domains has emphasized the need for robust, scalable, and efficient solutions. Real-time systems have become critical in ensuring seamless interactions, particularly in industries requiring instantaneous updates, such as customer support, logistics, and collaborative platforms. These systems aim to enhance user experiences by delivering immediate feedback and reducing delays.

Despite advancements, traditional systems often face challenges in managing high-volume, bi-directional communication and ensuring consistency across different user roles. These limitations become particularly evident in scenarios involving multiple stakeholders, such as users, agents, and administrators, working collaboratively within a system. The need for a reliable solution synchronize data, ensure real-time updates, and maintain role-based access has never been more crucial.

This dissertation system designed to address these challenges. By leveraging WebSockets and an event-driven architecture, the system ensures instantaneous updates for ticket creation, status changes, and assignment operations. Built using Node.js, Express.js, MongoDB, and Kafka, the platform incorporates scalable and efficient communication mechanisms while maintaining data integrity. The proposed solution aims to offer an intuitive responsive user experience across dashboards for users, agents, and administrators.

## **Motivation**

The increasing reliance on real-time systems across industries such as e-commerce, customer support, logistics, and collaborative platforms has highlighted the critical need for systems that an deliver instantaneous updates and seamless interactions. Traditional communication systems often struggle to meet these demands due to latency issues, lack of scalability, and difficulty in maintaining data consistency among various stakeholders. As businesses increasingly shift towards digital and interconnected operations, the need for reliable and scalable real0time communication platforms has become more pressing than ever.

Despite advancements in technology, many existing solutions fail to effectively address challenges like role-based access control, real-time data synchronization, and handling high-volume data streams efficiently. Users often face delays in receiving updates, and administrators encounter difficulties in managing operations dynamically. These limitations can lead to customer dissatisfaction, operational inefficiencies, and ultimately, business losses.

The dissertation is motivated by the opportunity to bridge these gaps by leveraging modern technologies like WebSockets, Kafka, and an event-driven architecture to develop a robust real-time ticketing system. The proposed system not only addresses the technical limitations of existing solutions but also provides a scalable and intuitive platform that ensures consistent and secure interactions for all users-be they customers, agents, or administrators. The project aims to contribute to the ongoing evolution of real-time communication systems, offering innovative approaches to tackle long0standing challenges and setting a foundation for future advancements in this field.

## **Objectives**

The primary objective of this dissertation is to develop a robust and scalable real-time ticketing system that addresses the challenges of modern communication and operational needs in various domains. Be leveraging advanced technologies such as WebSockets, Kafka, and an event-driven architecture, the system aims to provide instantaneous updates. Seamless interactions, and enhanced role-based access for users, agents, and administrators.

Specific objectives include:

1. **Real-Time Updates:** To enable instantaneous updates for ticket creation, assignment, and status changes, ensuring a smooth and uninterrupted flow of information among stakeholders.
2. **Data Consistency and Synchronization:** To ensure that all users have access to the most up-to-date information by maintaining real-time data synchronization across the system.
3. **Role-Based Access Control:** To implement secure and efficient access controls that differentiate user privileges and ensure that each role-whether a customer, agent, or administrator-can perform its designated tasks without interference.
4. **Scalability:** To design the system to handle high volumes of concurrent users and data streams without compromising performance or reliability.
5. **User Experience:** To deliver an intuitive and responsive interface for all users, enabling effortless navigation and task completion.
6. **System Reliability and Security:** To build a system that ensures data integrity, prevents unauthorized access, and provides fault-tolerant mechanisms to handle potential systems failures.

This dissertation aims not only to address existing limitations in real-time communication systems but also to set a foundation for future enhancements, supporting scalable and innovative solutions in the field.

## **Structure**

This dissertation is organized into several chapters, each addressing specific aspects of the development and implementation of real-time ticketing system:

1. **Chapter 1:** **Introduction**

This chapter provides an overview of the project, its motivation, and objectives. It sets the foundation for understanding the problem domain and outlines the structure of the dissertation.

1. **Chapter 2: Literature Review**

This chapter surveys existing research and technologies related to real-time communication systems, event-driven architectures, and role-based access control. It highlights the strengths and limitations of current solutions and positions this project within the broader research landscape.

1. Chapter 3: System Design and Architecture

This chapter delves into the architectural design of the system, detailing the use of technologies such as WebSockets, Kafka, and MongoDB. It explains the rationale behind the chosen design and how it addresses the identified challenges.

1. Chapter 4: Implementation

This chapter provides a comprehensive discussion of the implementation process, including the development of key system features like real-time updates, fault-tolerant mechanisms, and user role management.

1. Chapter 5: Results and Evaluation

This chapter evaluates the system’s performance, scalability, and reliability. It includes testing results and discusses how the system meets the objectives outlined earlier.

1. Chapter 6: Conclusions and Future Work

This chapter summarized the contributions of the dissertation and suggests potential direction for future research and development

1. Appendices

Additional materials, including code snippets, diagrams, and datasets, are provided here for reference.

# **Literature Review**

## **Overview**

The rapid advancements in technology have reshaped the landscape of communication systems, making real-time data processing and exchange an essential component of modern applications. From e-commerce platforms to collaborative tools and customer support systems, the demand for seamless and instantaneous communication has become a cornerstone for businesses and organizations seeking to enhance user experiences and operational efficiency.

Traditional communication systems, despite their widespread adoption, often fall short in addressing the complexities of real-time data exchange. These systems are typically hindered by latency issues, scalability challenges, and a lack of mechanism to ensure data consistency and reliability. As industries become increasingly communication has grown significantly.

This dissertation builds upon the foundation of existing research and leverages cutting-edge technologies such WebSockets, Kafka, and event-driven architectures to develop a real-time ticketing system. By addressing key challenges such as role-based access control, real-time updates, and high-volume data management, this work aims to contribute to the ongoing evolution of real0time communication systems. The subsequent sections will explore existing literature in this domain, critically analyzing related work and identifying gaps that this project seeks to address.

## **Related Work**

The purpose of this section is to explore existing research and systems relevant to real-time communication platforms, focusing on the technologies and methodologies that address challenges such as scalability, latency, and role-based control.

By critically examining related work, this dissertation aims to identify the strengths and limitations of existing solutions, providing a foundation for understanding how the proposed system contributes to the field.

This section will focus on systems and research that utilize WebSockets, Kafka, and event-driven architectures, discussing their applications, performance, and shortcomings in real-time communication.

## **Critical Analysis**

This section evaluates existing real-time communication systems and methodologies by analyzing their strengths and limitations. The analysis focuses on key aspects such as scalability, latency, fault tolerance, and role-based access control, which are critical to modern real-time applications. By identifying gaps in current approaches, this section sets the foundation for justifying the need for the proposed solution outlined in this dissertation.

### **Strength of Existing Systems**

**Scalability**

Scalability is a critical aspect of real-time systems, enabling them to handle increasing loads without compromising performance. It ensures that systems can grow efficiently to meet users demands, whether through hardware upgrades or distributed architectures. Technologies like Kafka exemplify scalability through features such as horizontal scaling, where additional brokers can be added to the cluster to distribute workloads effectively.

For instance, Kafka’s message partitioning mechanism allows data to be segmented and processed in parallel across multiple nodes, ensuring high throughput even under heavy traffic. This capability is particularly valuable for systems like e-commerce platforms and ticketing systems that experience significant traffic surges during peak periods. By supporting seamless scalability, these systems can maintain responsiveness and reliability, providing consistent user experiences regardless of demand fluctuations.

**Latency**

Latency is another key strength of modern real-time systems, ensuring minimal delays in data transmission and processing. Technologies like WebSockets are designed to reduce latency by establishing persistent, bi-directional communication channels between clients and servers. Unlike traditional HTTP request-response mechanisms, WebSockets allow for instant data exchange, eliminating the overhead of repeatedly establishing connections.

For example, in collaborative platforms or ticketing systems, low latency enables near-instantaneous updates, such as notifying agents about new tickets or updating users on ticket status changes. This responsiveness is essential for maintaining user satisfaction and operational efficiency, especially in scenarios requiring immediate feedback and decision-making.

**Fault Tolerance**

Fault tolerance is a critical aspect of robust real-time systems, ensuring reliability and uninterrupted operations even in the event of component failures. Technologies like Kafka inherently support fault tolerance by utilizing distributed architecture and replication. This design ensures that data remains accessible and consistent even if one or more brokers fail, maintaining the integrity of real-time data streams.

For example, in high-stakes applications like real-time ticketing systems, fault tolerance ensures that tickets are not lost during transmission or processing due to system failures. By leveraging Kafka’s replication capabilities and failover mechanisms, real-time systems can handle unexpected disruptions, providing a seamless experience for users and administrators alike.

**Role-Based Access Control (RBAC)**

Role-Based Access Control (RBAC) plays a crucial role in ensuring security and operational efficiency in real-time systems. By assigning permissions and access levels based on user roles, such as administrators, agents, and end-users, RBAC ensures that sensitive operations are only accessible to authorized individuals. This minimizes the risk of unauthorized actions and protects the integrity of the system.

Existing systems effectively utilize RBAC to streamline workflows and enforce security policies. For instance, in ticketing platforms, administrators can manage and monitor tickets, agents can handle assigned tasks, and users can create and track tickets within their permissions. This structured access control enhances user accountability and prevents accidental or malicious misuse of the system, fostering a secure and well-organized environment.

* + 1. **Weaknesses of Existing Systems**

**Scalability Issues**

Although significant progress has been made in developing real-time communication systems, scalability continues to be a pressing challenge. As the number of users and the volume of data increase, many traditional systems struggle to maintain efficiency, leading to performance bottlenecks, slower response times, and operational disruptions during peak usage.

Furthermore, the inability of some systems to dynamically allocate resources in response to sudden spikes in activity exacerbates scalability challenges. This limitation often results in degraded user experiences, particularly in industries like e-commerce, logistics, and collaborative platforms, where real-time communication is critical.

**Latency and Performance Bottlenecks**

While modern communication systems such as those leveraging WebSockets and Kafka offer reduced latency compared to traditional architectures, performance bottlenecks can still emerge in specific scenarios. Systems relying on real-time data processing must ensure minimal delay to maintain user satisfaction, yet challenges like high server loads, inefficient resource allocation, or suboptimal network conditions can introduce significant latencies.

For example, WebSocket-based systems depend heavily on maintaining active connections, which can degrade in performance in proper scaling mechanisms, such as distributed servers or load balancers, are not in place. Similarly, Kafka, while efficient in handling large data streams, may encounter delays if producers or consumers are not optimized for high-throughput environments or if disk I/O becomes a bottleneck.

Addressing latency issues requires a multi-faceted approach, including the optimization of network protocols, the introduction of caching mechanisms, and the effective distribution of computational resources. Such measures are critical in ensuring that performance bottlenecks do not undermine the reliability of real-time systems.

**Fault Tolerance Limitations**

Fault tolerance is a critical aspect of real-time communication systems, ensuring uninterrupted service and data integrity even in the presence of failures. Systems like Kafka and WebSockets are inherently designed to handle faults to some extent, but challenges still arise when failures occur at scale or across multiple components.

For instance, Kafka’s replication mechanism allows data to persist even if one or more brokers fail. However, if replication factors are not adequately configured or if the leader election process becomes delayed, it can impact the overall system availability. Similarly, in WebSocket-based systems, a dropped connection due to server failure can disrupt real-time communication unless reconnection mechanism and failback strategies are implemented effectively.

Modern fault-tolerant systems also need to address challenges like:

* **Partition loss** in distributed setups.
* Ensuring **state consistency** across nodes.
* Implementing robust **retry mechanisms** to recover from transient failures.

By identifying and addressing these issues, the proposed solution in this dissertation aims to ensure resilience, minimizing downtime and maintaining consistent service levels in real-time environments.

**Role-Based Access Control Constraints**

Role-Based Access Control (RBAC) mechanisms are widely employed in modern systems to ensure secure and restricted access to resources based on user roles and permissions. While RBAC systems provide a structured framework for managing user access, they are not without limitations, especially in the context of real-time applications.

One of the key challenges is RBAC implementations lies in **scalability**. As the number of users and roles increases, the complexity of managing permissions and ensuring consistent enforcement grows exponentially. Systems with frequent role or permission updates may face delays in propagating changes, leading to potential security gaps or operational inefficiencies.

Another limitation is the **lack of flexibility** in adapting to dynamic access needs. Real-time communication systems often involve ad hoc collaboration or temporary access requirements, which traditional RBAC frameworks struggle to accommodate efficiently.

Moreover, **integration issues** arise when combining RBAC with real-time communication technologies like WebSockets. Maintaining synchronized role-based access in distributed architectures, especially during network interruptions or system failures, presents significant challenges. These constraints can hinder the system’s ability to provide uninterrupted service and maintain data security.

By addressing these constraints, the proposed solution in this dissertation aims to enhance the effectiveness of RBAC mechanisms in real-time environments. It seeks to integrate dynamic role updates, minimize propagation delays and ensure seamless operation across distributed nodes.

### **Proposed Improvements**

**Enhancing Scalability**

To address the scalability challenges identified in existing systems, the proposed solution employs a distributed architecture leveraging Kafka and WebSockets. By distributing workloads across multiple nodes, the system ensures efficient resource allocation and minimizes bottlenecks during peak usage.

Dynamic resource allocation mechanisms are implemented to adapt to sudden spikers in user activity. This involves leveraging Kafka’s partitioning capabilities to distribute messages evenly brokers and scaling WebSocket serves to handle concurrent connections effectively.

Additionally, the system incorporates real-time monitoring and load-balancing strategies to identify and mitigate performance bottlenecks proactively. By continuously analyzing system performance and reallocating resources, the proposed solution achieves greater scalability and reliability in handling high-volume data streams.

**Improved Fault Tolerance**

To ensure resilience and data integrity, the proposed solution utilizes Kafka’s replication mechanism to minimize the impact of broker failures. By distributing data across multiple brokers and implementing a suitable replication factor, the system ensures data availability and fault tolerance in case of node failures.

Advanced fault-tolerance strategies, such as leader election optimization and dynamic reallocation of resources, can further enhance system reliability. For instance, improving the responsiveness of leader election mechanisms during broker failures reduces downtime and prevents disruptions to real-time communication.

Additionally, WebSocket-based systems can incorporate retry and reconnection mechanisms to handle transient network failures effectively. By proactively addressing potential points of failure, these improvements ensure seamless real-time interactions, even under adverse conditions.

The proposed solution aims to address critical challenges such as:

* **Partition recovery** in distributed environments.
* **Leader election stability** to minimize failover latency.
* **Optimized resource utilization** to handle high-volume workloads during failures.

By implementing these enhancements, the system strengthens its ability to deliver consistent and reliable service across real-time environments.

**Refining Role-Based Access Control**

To address the limitations identified in the existing RBAC implementations, the proposed solution introduces dynamic role management and real-time permission updates. By leveraging a distributed architecture, the system ensures that role-based access policies remain consistent across multiple nodes, even during high-volume operations.

Key improvements include:

* **Dynamic Role Updates:** Incorporating mechanisms to propagate role and permission changes in real time, reducing delays and ensuring seamless enforcement of access policies.
* **Adapting to Temporary Roles:** Enabling the creation of ad hoc or temporary roles for specific tasks or collaborations without compromising system security.
* **Integration with Real-Time Systems:** Ensuring synchronization between RBAC policies and real-time communication technologies such as WebSockets to maintain uninterrupted and secure interactions/

By implementing these enhancements, the proposed solution fosters a secure and adaptable environment, capable of meeting the demands of modern real-time communication systems.

### **Summary and Conclusion**

The literature review highlighted the rapid advancements in real-time communication systems and their increasing importance across various industries. Technologies such WebSockets, Kafka and event-driven architectures have proven to be effective in addressing many challenges, including scalability, latency, fault tolerance, and secure access control through RBAC.

Despite these advancements, several limitations persist:

* Scalability issues during peak loads sudden spikes in user activity.
* Latency and performance bottlenecks in high-throughput environments.
* Fault tolerance challenges in handling broker failures and ensuring data consistency.
* Constrains in RBAC systems related to scalability, flexibility, and integration with real-time technologies.

The proposed solution addresses these gaps by leveraging dynamic resource allocation, ad vanced fault-tolerance mechanism, and refined role-based access control strategies. By incorporating these improvements, the system aims to provide a robust, scalable, and secure real-time communication platform.

This chapter forms the foundation for the subsequent chapters, which will detail the design, architecture, implementation, and evaluation of the proposed system, showcasing how it addresses the identified gaps and contributes to advancing real-time communication systems.

# **System Design and Architecture**

## **Overview**

The system design of the real-time ticketing platform is centered around a robust architecture that integrates multiple components to ensure scalability, reliability, and seamless real-time communication. By leveraging modern technologies like WebSockets, Kafka, MongoDB, and Node.js, the system is built to handle high user concurrency while maintaining data integrity and responsiveness.

At the core of the architecture, WebSockets provide a persistent, bi-directional communication channel between clients and the server, enabling instantaneous updates for ticket creation, assignment, and status changes. Kafka serves as the backbone for reliable message queuing and data streaming, ensuring scalability and fault tolerance in handling high-volume transactions. MongoDB is employed as the primary database, offering flexibility in managing dynamic ticketing data and ensuring high performance during read and write operations.

This chapter provides a detailed explanation of the system’s architecture outlining how its components interact to achieve the objectives of real-time data synchronization, role-based access control, and fault tolerance. The following sections will delve into the architectural design principles, the choice of technologies, and their integration within the system to address the challenges identified in earlier chapters.

## **Architectural Design**

The architectural design of the real-time ticketing system is based on a modular and distributed approach, ensuring scalability, fault tolerance, and efficient real-time communication. The system architecture consists of the following core components:

1. **Client-Side Interface:**

* The frontend is built using React, providing users, agents, and administrators with intuitive dashboards for ticket creation, management, and monitoring.
* The client communicates with the backend through REST APIs certain operations and WebSocket connections for real-time updates.

1. **Backend Server:**

* Built using Node.js and Express.js the backend handles core business logic, authentication, and communication between the frontend, database, and message broker.
* WebSocket integration ensures live updates for ticket operations, such as creation, assignment, and status changes.

1. **Message Broker (Kafka):**

* Kafka serves as the event-streaming platform, facilitating reliable message queuing between system components.
* Although Kafka supports advanced features such as message partitioning and replication for fault tolerance, this implementation focuses on its core capability of ensuring reliable communication and event delivery.

1. **Database (MongoDB):**

* MongoDB stores ticket data, user roles, and systems logs. Its NoSQL structure allows for flexible data modeling and efficient handling of dynamic ticketing operations.
* The database ensures data consistency and support high-speed read/write operations.

1. **Role-Based Access Control (RBAC):**

* A structured RBAC system ensures secure access to the platform. Roles include users, agents, and administrators, each with specific permissions.
* Token-based authentication using JSON Web Tokens (JWT) provides secure and scalable session management.

1. **Real-Time Communication:**

* WebSocket connections facilitate bi-directional communication between the server and clients, enabling instantaneous updates for ticket activities.

1. **Fault Tolerance Mechanisms:**

* The system leverage Kafka’s inherent reliability for event delivery. Advanced features like replication failover mechanisms, while supported by Kafka, are not configured in this implementation.
* Basic retry and reconnection mechanism ensure resilience during transient network disruptions.

The architectural design emphasizes modularity, making it easier to extend or modify components without disrupting the overall system. The next section will delve into the detailed workflows and interactions between these components.

## **Detailed Workflow and Interactions**

**Overview of Workflows:**

The real-time ticketing system ensures seamless interaction among users, agents, and administrators by employing a modular workflow that facilitates ticket creation, assignment, and status updates. Each workflow is designed to leverage the capabilities of WebSockets for real-time communication and Kafka for reliable message delivery.

For instance, when a user creates ticket, the request flows from the client-side interface to the backend server, where it is validated, processed, and stored in the MongoDB database, Simultaneously, Kafka’s event-streaming mechanism ensures that updates are broadcast to all relevant stakeholders, enabling instantaneous visibility across the system.

**Interaction Between Components**

* **Client-Side Interaction:**
  + 1. Users initiate actions such as ticket creation, updates, or queries through the React-based dashboard.
    2. WebSocket connections maintain a persistent, bi-directional link between the client and server, enabling real-time updates for all subscribed clients without the need for repeated HTTP requests.
* **Backend Server Operations:**
  + 1. The backend server, built with Node.js and Express.js, processes incoming requests by interacting with MongoDB for data storage and the WebSocket server for real-time communication.
    2. It handles role-based permissions, ensuring that each user can only perform actions within their designated access level.
* **Kafka as a Message Broker:**
  + 1. Kafka is integrated into the architecture as a basic event-streaming platform for message queuing.
    2. The current implementation focuses on sending messages to a single topic (tickets) and consuming those messages for processing. Advanced features like partitioning and multi-consumer setups are not utilized.
* **Database Operations**
  + 1. MongoDB servers as the system’s data repository, storing ticket details, user roles, and log information.
    2. Queries are optimized to ensure high-speed data retrieval for real-time operations, particularly during search and filtering tasks.

**Role-Based Interactions**

* **Users:**
  + 1. Users can create new tickets, view existing tickets, and track the progress of their requests in real time.
    2. Upon ticket creation, the system assigns a unique identifier to the ticket and broadcasts updates to relevant agents and administrators using WebSocket events.
* **Agents:**
  + 1. Agents receive ticket assignments through real-time WebSocket updates and can update ticket statuses.
    2. The system ensures that agents only have access to tickets assigned to their categories or expertise, as determined by ticket properties and agent roles.
* **Administrators:**
  + 1. Administrators can create and delete tickets, update ticket statuses, and monitor ticket operations across all categories.
    2. They have global access to all tickets and are responsible for overseeing the overall system’s functionality.

**Illustrative Diagrams**

* **System Architecture Overview:**

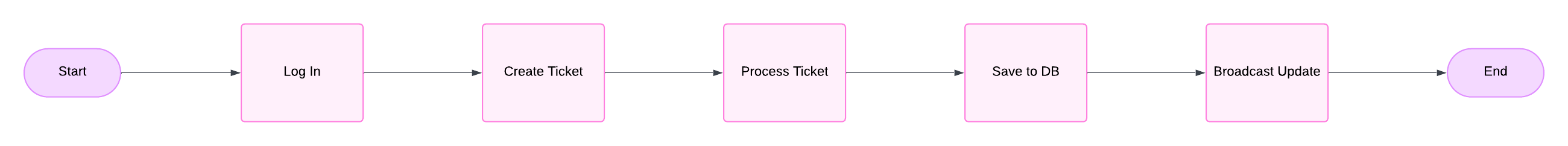
This diagram represents the overall architecture of the ticketing system. It showcases the flow of data and interactions between the front-end dashboards, backend core, WebSocket for real-time updates, Kafka for message queuing, and MongoDB for data storage. The architecture supports ticket creation, updates, and notifications across the User, Agent, and Admin dashboards.

A diagram of a company

Description automatically generated

* **User Dashboard Workflow**

This diagram illustrates the workflow for the User Dashboard in the ticketing system. It shows how users interact with the dashboard to crate tickets, view ticket details, and track ticket statuses. The workflow highlights the integration with the backend, MongoDB for ticket storage, and WebSocket for real-time updates.



* **Agent Dashboard Workflow**

A diagram of a program

Description automatically generatedThis diagram illustrates the functionality of the Agent Dashboard. It highlights how agents receive ticket assignments, update statuses, and monitor changes in real time using WebSocket updates.

* **Admin Dashboard Workflow**

This diagram illustrates the workflow for the Admin Dashboard in the ticketing system. It highlights ability to view all tickets, create new tickets, update ticket statuses, and delete tickets. The diagram also showcases the integration with the backend for processing updates and broadcasting changes via WebSocket. This ensures all connected dashboards (User, Agent, and Admin) are synchronized in real-time with the latest ticket information.

A diagram of a diagram

Description automatically generated

**Ticket Creation Process**

This subsection describes the detailed process for creating tickets in the system. It covers how user requests are handled from the frontend to the backend and the database, followed by broadcasting real-time updates using Kafka and WebSocket.

* **Frontend Validation and Data Submission**

When a user creates a ticket, the process begins on the User Dashboard. The frontend provides a form with fields for the ticket title, description, and category. Before submission, client-side validation ensures all required fields are completed, adhere to the expected formats, and meet predefined criteria. Once the validation succeeds, the frontend sends the ticket data to the backend via a POST request to the /create-ticket endpoint. This request includes the user’s session token (JWT) in the headers, which the backend uses to authenticate the user and verify their authorization to create tickets. This ensures only legitime users can perform this operation, maintaining the integrity and security of the system.

* **Backend Processing**

Upon receiving the request, the backend performs additional validation to verify the data’s integrity and ensure the user is authorized to create tickets. Using Kafka’s producer functionality, the backend sends a “new-ticket” message containing the ticket details (e.g., title, description category, and initial status: Waiting) to the designated topic. A Kafka consumer, subscribed to this topic, processes the event and saves the ticket data to the tickets collection in MongoDB. This stored document includes key details such as the ticket’s creator, category, and status.

* **Real-Time Updates**

After a ticket creation request is processed, the backend integrates Kafka and WebSocket to ensure real-time communication across all dashboards. First, the system uses Kafka’s producer functionality to send a “new-ticket” message to the designated topic. The message includes all necessary ticket details, such as the little, description, category, and initial status (waiting). A Kafka consumer, subscribed to this topic, processes the event and saves the ticket data to the MongoDB database, ensuring asynchronous and reliable data handling.

Once the Kafka consumer confirms that the ticket has been stored successfully, the backend emits a “ticket-created” event via WebSocket. This event is broadcast to all connected clients, including the User, Agent, and Admin Dashboards, ensuring that all relevant stakeholders are updated in real time. This two-step process of Kafka message handling and WebSocket broadcasting ensures that the system remains both scalable and responsive, with room for future enhancements such as multi-consumer setups or external integrations.

**Ticket Management Workflows**

* **User Interaction with Created Tickets**

Once a ticket is created, users can view its details and monitor its progress in real time through the User Dashboard. The dashboard fetches ticket data directly from the tickets collection in MongoDB, using the user’s unique identifier to ensure that only tickets associated with their account are displayed. This is achieved via a secure API call to the /get-tickets endpoint, which validates the user’s session token (JWT) to authorize access.

Real-time updates are facilitated through WebSocket events such as “status-updated”. For example. When an agent modifies the status of a ticket (e.g., from waiting to in progress), the User Dashboard receives as instant notification and updates the displayed ticket details without requiring a page refresh. This seamless integration between the backend, database, and WebSocket ensures that users are always informed about the latest developments in their tickets.

* **Agent Interaction with Tickets**

Agents play a key role in managing tickets within the system. Once a ticket is created and assigned to an agent based on its category, the Agent Dashboard provides real-time access to all assigned tickets. The dashboard retrieves ticket data via a secure API call to the /get-tickets endpoint, using the agent’s session token (JWT) to validate the identify and authorization.

Agents can view ticket details, including the ticket’s status (waiting, in progress, or resolved), title, and description. If required, agents can update a ticket’s status by interacting with the dashboard. When an agent modifies a ticket (e.g., updating its status to in progress), a PUT request is sent to the /update-ticket endpoint. The backend processes the request, validates the agent’s authorization, and updates the ticket’s status in MongoDB.

Real-time notifications are then triggered via a “status-updated” WebSocket event, which is broadcast to all relevant dashboards. This ensures that users are instantly notified of updates to their tickets, and other agents or admins have the latest information available. The seamless in integration of WebSocket and database updates enhances coordination among agents and improves response times.

* **Admin Role in Ticket Management**

Admins have full control over ticket operations and management within the system, granting them the highest level of access across all dashboards. Using the Admin Dashboard, they can monitor, create, update, and delete tickets, ensuring efficient oversight of the platform’s operations.

* **View and Monitor Tickets**

Admins can view and monitor tickets across all categories using the Admin Dashboard. The dashboard retrieves tickets via the common /get-tickets endpoint, where the backend dynamically filters tickets based on the user’s role. For admins, all tickets are retrieved, while agents and users only see tickets relevant to their assigned categories or those they created, respectively. The ticket details displayed include the title, description, category, status, and creator. This unified approach ensures that all roles can access their relevant data while maintaining security and efficiency.

* **Create Tickets**

Like user functionality. Admins can create tickets by submitting a form with details such as title, description, and category. The frontend sends the data to the backends’ /create-ticket endpoint, where it is processed and stored in the database. Real-time updates ensure the newly created ticket is visible to all relevant dashboards (e.g., Agent and User Dashboards).

* **Update Ticket Status**

Admins can update ticket statuses, like agents, through the Admin Dashboard. For example, they can mark a ticket as resolved or in progress depending on the current state of operations. These updates are processed by the /update-ticket endpoint, and corresponding WebSocket events “status-updated” notify all connected clients of the change.

* **Delete Tickets**

Admins have the exclusive ability to delete tickets. When ad admin deletes a ticket through the dashboard, a DELETE request is sent to the /delete-ticket endpoint. The backend processes the request by verifying the admin’s authorization, removing the ticket from MongoDB, and broadcasting a “ticket-deleted” WebSocket event to ensure that all dashboards are updated in real time. This capability allows admins to manage obsolete or duplicate tickets efficiently.

**System-Level Workflows and Communication**

* **Kafka Producer / Consumer Workflow**

Kafka servers as a reliable event-streaming platform in the system, facilitating asynchronous communication between components. Its producer / consumer model ensures scalability and fault tolerance by decoupling the ticket creation process into distinct steps.

The **Kafka Producer**, initialized in the backend, plays a key role during ticket creation. Once the user submits ticket details, the backend’s /create-ticket endpoint sends a “new-ticket” event to the Kafka topic named tickets using the sendMessage function. This event includes essential ticket data, such as the title, description, category, and initial status.

Simultaneously, the **Kafka Consumer**, also initialized in the backend, listens to the tickets topic. When a “new-ticket” event is detected, the consumer processes the message by saving the ticket details to the MongoDB database. This separation ensures that data is reliably stored even if downstream services experience temporary issues.

Both the producer and consumer are connected to the Kafka cluster through a shared configuration. For example, the clientId is set to “ticketing-system”, and the Kafka broker is hosted locally at localhost:9092. This lightweight setup effectively integrates Kafka into the system, ensuring seamless communication and data processing.

A diagram of a network

Description automatically generated with medium confidence

* **WebSocket Event Flow**

**Introduction**

WebSockets enable real-time communication between the server and connected clients. Unlike traditional HTTP requests, WebSockets maintain a persistent, bi-directional connection, allowing instant event-based communication. This functionality is critical for synchronizing the User, Agent, and Admin Dashboards in the real-time ticketing system.

**Step 1: Connection Establishment**

The WebSocket connection begins with the initialization of the server. The backend uses the socket.io library to setup up a WebSocket server, which listens for incoming client connections. Each client (e.g., User Dashboard, Agent Dashboard) establishes a connection to the WebSocket server upon logging in. The server assigns a unique socket ID to every connected client, ensuring individual sessions for real-time communication.

**Step 2: Event Emission and Listening**

After the WebSocket connection is established, the server and clients communicate via specific event channels. These events allow for real-time updates across all connected dashboards.

**1. Backend Event Emission:**

The backend emits events when critical actions occur, such as:

* + **“ticket-created”:** Sent when a new ticket is created. This event carries ticket details like title, description, and category.
  + **“status-updated”:** Sent when a ticket’s status changes (e.g., from “waiting” to “in-progress”).
  + **“ticket-deleted”:** Sent when an admin deletes a ticket. The event contains the ticket ID for client synchronization.

**2. Client Event Listeners:**

Each dashboard (User, Agent, Admin) listens to relevant events using the socket.on() method:

* + **User Dashboard**: Listens for “ticket-created,” “status-updated,” and “ticket-deleted” to display real-time changes.
  + **Agent Dashboard**: Similar to the User Dashboard but focuses on tickets assigned to the agent’s category.
  + **Admin Dashboard**: Primarily listens for general updates and handles admin-specific functionalities like deletion locally.

**Step 3: Synchronization and Updates:**

Synchronization ensures that any updates made to tickets are reflected in real-time across all connected dashboards (User, Agent, Admin). This step highlisghts how the system achieves consistency and responsiveness:

**1. Broadcasting Events:**

When a ticket event occurs (e.g., creation, status update, or deletion), the backend broadcasts the corresponding event to all connected clients using the io.emit() method. This ensures that every dashboard receives the update without the need for refreshing or polling.

**2. Dashboard Updates:**

Each dashboard listens to specific events and updates its interface accordingly:

* **User Dashboard:** Automatically updates the ticket list and status details when events like “status-updated” or “ticket-deleted” are received.
* **Agent Dashboard:** Refreshes the agent’s assigned ticket list relevant events occur.
* **Admin Dashboard:** Handles updates efficiently, ensuring administrative actions like ticket deletion reflect in the system instantly.

**3. Ensuring Consistency:**

To prevent discrepancies, the system validates incoming events and updates only the relevant tickets. For example:

* + If a “**ticket-deleted**” event is received, only the corresponding ticket is removed from the interface.
  + If a “**status updated**” event is received, the specific ticket’s status is updated while preserving other details.

**Step 4: Error Handling and Reconnection:**

The WebSocket implementation in the real-time ticketing system ensures stable connections between the server and clients. While error handling is basic, it covers essential scenarios to maintain connectivity.

**1. Socket Connection Initialization:**

* Each client (e.g., User, Agent, or Admin Dashboard) establishes a WebSocket connection upon loading the dashboard.
* A unique socket.id is generated and logged on the backend to track active connections

**2. Automatic Reconnection:**

* When a WebSocket connection is interrupted (e.g., due to a network issue or browser refresh), the socket.io library automatically attempts to reconnect to the server.
* This ensures that clients rejoin the real-time communication seamlessly without manual intervention.

**3. Frontend Error Handling:**

* Error events such as connection loss or failed reconnection are handled using basic console logging for debugging purposes.
* While the application doesn’t display user-facing error messages (e.g., “Connection lost”), these errors are monitored in the browser developer tools for troubleshooting.

**4. Backend Logging:**

* The backend logs disconnection events, including the socket.id of disconnected clients, to track and monitor connection statuses.

**5. Room for Improvement:**

* Currently, no visual indicators or notifications are implemented for users experiencing connection issues. However, this could be a potential enhancement to improve user experience.

**Step 5: Scalability and Performance Optimization**

The WebSocket implementation in the ticketing system is designed to handle multiple connections and real-time updates efficiently. While it is a lightweight setup, it incorporates several elements that contribute to scalability and performance:

**1. Efficient Use of socket.io:**

* The socket.io library supports real-time communication for multiple users without compromising performance.
* Its capability to manage event-driven architecture ensures the system is lightweight and adaptable to growing demands.

**2. Backend Resource Management:**

* WebSocket connections are dynamically handled by the backend.
* Disconnection cleanup ensures server resources are freed immediately.

**3. Horizontal Scaling with Load Balancers:**

* The architecture allows the use of multiple WebSocket servers for scalability.
* Load Balancers can ensure users stay connected to the correct server for real0time updates.

**4. Integration with Kafka:**

* Kafka acts as the backbone for message processing, ensuring large-sclae data streams are managed efficiently.
* The Kafka producer / consumer workflow supports high volumes of real-time events without overloading the system.

**5. Database Optimization:**

* MongoDB’s indexing and schema design improve query performance for real-time updates.
* Optimized write operations ensure tickets are updated and retrieved with minimal delay.

**6. Potential Future Enhancements:**

* **Rate Limiting:** Prevent overloading the WebSocket server with excessive updates.
* **Monitoring Tools:** Implement tools to track WebSocket and Kafka performance metrics.

**Database Design and Operations**

* **Database Schema and Relationships:**

The Real-Time Ticketing System leverages MongoDB for efficient storage and management of data. The database consists of two main collections: Account and Tickets. These collections are structured to ensure scalability and easy access to information.

* **Accounts Collection:**
  + Fields:
    - \_id (ObjectId): A unique identifier for each account.
    - username (String): The display name of the user.
    - email (String): The user’s email address
    - password (String): A hashed version of the user’s password for secure authentication.
    - role (String): Specifies the user’s role (e.g., admin, agent, or user).
    - department (String): Applicable only for agents, specifying their department (e.g., Technical Support, Sales).
  + A close-up of a sign

    Description automatically generatedExample Document:
* **Tickets Collection:**
  + Fields:
    - \_id (ObjectId): A unique identifier for each ticket.
    - title (String): A short title describing the ticket.
    - description (String): Detailed information about the issue.
    - status (String): Tracks the ticket’s progress (e.g., Open, Resolved).
    - category (String): Specifies the type of issue e.g., Technical Support, General Inquiry)
    - createdBy (ObjectId): Reference the \_id of the user who created the ticket
  + A white background with red text

    Description automatically generatedExample Document:
* **Relationships:**

The createdBy field in the Tickets collection references the \_id field in the Accounts collection. This relationship allows the system to associate each ticket with the user who created it, enabling role-based interaction and efficient data querying.

* **Database Operations and Query Optimization**

Efficient database operations are crucial for ensuring the Real-Time Ticketing System performs as expected. Below, the current CRUD operations implemented in the system are described, followed by suggestions for potential enhancements in query optimization.

**CRUD Operations**

The Ticketing System supports the full range of CRUD (Create, Read, Update, Delete) operations for managing tickets. These operations are defined in the ticketRoute.js file and interact with the MongoDB collections via Mongoose.

**1. Create:**

* + **Endpoint**: POST /tickets
  + Inserts a new ticket into the database using the data provided in the request body.
  + Field include:
    - title
    - description
    - status
    - category
    - createdBy (a reference to the Account collection)

**2. Read:**

* + **Endpoint**: GET /tickets.
  + Retrieves all tickets from the database.

**3. Read Ticket by ID:**

* + **Endpoint:** GET /tickets/:id
  + Fetches a specific ticket by its unique identifier.

**4. Update Ticket by ID:**

* + **Endpoint:** PUT /tickets/:id
  + Updates the status field of a ticket by its ID.

**5. Delete:**

* + **Endpoint:** DELETE /tickets/:id
  + Removes a ticket from the database

**Potential Query Optimization Techniques**

Currently, the application does not include advanced query optimization techniques like indexing, pagination, or lean queries. The following optimizations are suggested for future implementation to improve database efficiency:

**1. Indexing:**

* + - Create custom indexes for frequently queried fields, such as status and createdBy, to speed up query performance.

**2. Pagination:**

* + - Implement pagination using limit and skip parameters for the GET /tickets endpoint to handle large datasets efficiently.

**3. Lean Queries:**

* + - Use Mongoose’s .lean() method for read operations to return plain JavaScript objects instead of full Mongoose documents, reducing memory overhead.

**4. Cashing:**

* + - Add caching mechanisms for repetitive queries, such as retrieving all tickets, to reduce database load.

**5. Batch Updates:**

* + - Optimize write operations by implementing batch updates for scenarios where multiple tickets need updates simultaneously.

**System Security**

* **JWT Authentication**

The Real-Time Ticketing System implements JSON Web Tokens (JWT) to secure user authentication and control access to the system’s features. This approach ensures that sensitive data, such as user credentials, remains protected while facilitating seamless communication between the frontend and backend.

**Token Generation**

* + JWTs are generated during the login process within the authController.js file.
  + When a user logs in, the system creates a token by combining the user’s email, account ID, and randomly generated JWT secret stored in the /env file.
  + The token is set to expire after one hour, ensuring that expired tokens cannot be reused.

**Access Control**

* + The token is sent to the client after successfully authentication and is included in subsequent requests as part of the headers.
  + The system verifies the token’s validity for every incoming request to protect against unauthorized access.
  + Role-based access control (RBAC) is enforced at the route level. Only users with appropriate roles can access specific dashboards (e.g., Admin, User, or Agent dashboards).

**Implementation Highlights**

* + **Randomized Security**: The use of a unique secret key for each session ensures tokens are unpredictable and secure.
  + **Expiration Mechanism:** Token expiration minimizes the risk of misuse in case of credential compromise.
  + **Role Verification:** Routes include middleware checks to verify both the token and user role, ensuring that access permissions are strictly enforced.

Password Hashing

Role-Based Access Control (RBAC)

**Testing and Validation**

Tests text

## **Chapter Conclusion**