Real Time Sports Hub in Distributed Systems(Pub-Sub Approach)

Dheeraj Sai Venkat Gedupudi

Santa Clara University

Abstract - This project focuses on the design and implementation of a distributed publish-subscribe (pub/sub) system to provide real-time sports score updates. The system comprises three primary components: publishers, brokers, and subscribers. Developed using Spring Boot, the system leverages RESTful APIs for seamless communication between components. The brokers employ leader election algorithms and logical clocks to ensure data consistency and accurate timestamping across the distributed network. Core challenges of distributed systems, such as fault tolerance, scalability, and concurrency, are addressed through techniques like heartbeat signals for broker health monitoring and data synchronization among brokers. The project showcases the practical application of distributed systems concepts to build a scalable and reliable pub/sub system capable of handling real-time data dissemination while effectively managing failures and maintaining data consistency.

1. Goals and Motivation

The primary goal of our Real-Time Sports Score Update System is to create a highly responsive and scalable platform that caters to the dynamic needs of users and publishers within the sports ecosystem. This system is designed to provide instant score updates and event notifications across various sports disciplines, ensuring that fans, media outlets, and other stakeholders receive accurate and timely information. Publishers, including sports leagues and media organizations, benefit from our system by disseminating real-time updates efficiently to a global audience, thereby enhancing their engagement and reach.

The motivation behind this project stems from the increasing demand for real-time information in today's digital age. Traditional systems often struggle with high traffic volumes during peak times, leading to latency issues and data inaccuracies. These challenges necessitate the development of a robust and scalable solution that can handle large volumes of traffic while maintaining high data accuracy. Our system addresses these challenges by implementing advanced distributed algorithms for leader election, resource discovery, and event ordering. These algorithms ensure efficient coordination, dynamic service management, and correct event sequencing, providing a reliable and seamless user experience.

By focusing on these critical aspects, our project aims to set a new standard in the delivery of real-time sports updates. The system not only enhances the user experience by providing updates without delay but also supports the scalability required to manage peak loads effectively. This ensures that our platform remains responsive and accurate, even under high traffic conditions, thereby meeting the expectations of a global audience that relies on timely and precise sports information.

2. Challenges Addressed

Heterogeneity:

Challenge: Distributed systems often consist of diverse hardware and software environments. Ensuring compatibility and seamless interaction among different components can be complex.

Solution: The system is designed using standard protocols and technologies such as HTTP, RESTful APIs, and Spring Boot, which are widely supported across various platforms. By leveraging these widely adopted standards, we ensure that different components can interact seamlessly regardless of their underlying technology stack. This design choice

facilitates interoperability and reduces integration complexity.

Openness:

Challenge: Maintaining an open system where new components can be added or removed without significant disruption.

Solution: The use of RESTful APIs and a modular design allows for the easy integration of new brokers, publishers, and subscribers. The system can dynamically discover and register new components, facilitating extensibility and adaptability. This openness ensures that the system can evolve over time, integrating new functionalities and adapting to new requirements without major overhauls.

Security:

Challenge: Protecting data and ensuring secure communication in a distributed environment.

Solution: The system employs HTTPS for secure communication between components, ensuring that data transmitted over the network is encrypted and protected against eavesdropping and tampering. Authentication and authorization mechanisms are implemented using Spring Security to ensure that only legitimate entities can access the system and perform actions. Additionally, data integrity checks using hashing algorithms are performed to prevent data tampering, ensuring that the data received is exactly what was sent.

Failure Handling:

Challenge: Ensuring system reliability and availability in the presence of failures.

Solution: The system implements fault tolerance through redundancy and leader election mechanisms. Brokers send periodic heartbeats to a coordinator to indicate their availability. If a broker fails, the system uses an election algorithm to select a new leader from the remaining brokers, ensuring continuous operation. Data replication and synchronization mechanisms ensure that data is consistent across brokers, and no data is lost during failures. This setup enhances the system's reliability and availability.

Concurrency:

Challenge: Managing concurrent access and updates to shared resources without conflicts.

Solution: The system leverages synchronized methods and concurrent data structures (e.g., ConcurrentHashMap) to handle simultaneous access and modifications safely. Logical clocks (Lamport timestamps) are used to maintain a consistent order of events and updates across the system. This ensures that all components have a consistent view of the state, preventing conflicts and ensuring data consistency.

Quality of Service (QoS):

Challenge: Ensuring the system meets performance requirements such as low latency and high throughput.

Solution: The system is designed to distribute load across multiple brokers, balancing the traffic and reducing bottlenecks. Scheduled tasks and efficient algorithms are used to manage data dissemination and synchronization, ensuring timely updates and high performance. Load balancing techniques are employed to ensure that no single broker becomes a bottleneck, thereby maintaining low latency and high throughput.

Scalability:

Challenge: Designing a system that can scale horizontally to accommodate growing numbers of publishers, subscribers, and data volume.

Solution: The pub/sub model inherently supports scalability by decoupling publishers and subscribers. Brokers can be added or removed based on the load, and the system can dynamically adjust to changing demands. The use of distributed algorithms and data structures allows the system to handle large-scale deployments efficiently. This design ensures that the system can scale horizontally without significant changes to the core architecture.

Transparency:

Challenge: Providing a seamless user experience where the distributed nature of the system is hidden from end-users.

Solution: The system ensures transparency by abstracting the complexity of the underlying distributed infrastructure. Users interact with a unified interface, and the system handles the distribution, synchronization, and communication behind the scenes. This makes the system easy to use and manage without exposing the intricacies of its distributed architecture. For example, users can subscribe to topics and publish messages without needing to know which broker is handling their request or how the data is being synchronized across the system.

3. Work Related to This Problem (Literature Review)

1. Publisher-Subscriber Implementation in Cloud Environment

In their paper, Sebastijan Stoja, Srđan Vukmirović, and Bojan Jelačić implement a Publisher/Subscriber (Pub/Sub) system within Microsoft Windows Azure to enhance communication in distributed systems. The Pub/Sub model supports indirect, asynchronous communication between clients and servers, crucial for large-scale data processing. By utilizing Azure's execution and storage services, along with Azure App Fabric's Service Bus for asynchronous messaging, the authors create a robust Pub/Sub system.

The system includes publishers who disseminate information, subscribers who receive updates, and brokers who manage message delivery. This ensures that publishers and subscribers remain loosely coupled. Multiplexing messages optimizes access to the Service Bus.

Testing in a distributed network environment with various services (SSE, SDE, TS, SF, SC) demonstrated reliable real-time data publishing and subscribing. Although publishing takes slightly longer due to message size, the overall performance is satisfactory.

Techniques discussed in this paper, such as dynamic scaling and fault tolerance mechanisms, have been integrated into our system design to handle large volumes of real-time sports score updates efficiently.

2. Distributed Brokers in Message Queuing Telemetry Transport (MQTT)

Snowlin Preethi Janani, Immanuel JohnRaja Jebadurai, Getzi Jeba Leelipushpam Paulraj, and Jebaveerasingh Jebadurai examine the performance of distributed brokers in the Message Queuing Telemetry Transport (MQTT) protocol for IoT applications. MQTT, a lightweight publish/subscribe protocol, is crucial for resource-constrained IoT applications like smart homes and healthcare. The paper highlights the limitations of single broker setups, including risks of single points of failure and load issues, and advocates for distributed broker architectures to enhance reliability and scalability.

The authors review various distributed architectures, such as ring topology, fog-based systems, and Byzantine fault-tolerant systems, which address challenges like latency reduction and data consistency. They compare brokers like RabbitMQ, Mosquitto, HiveMQ, and VerneMQ based on throughput, latency, scalability, and security, noting that each broker has unique strengths suited to different applications.

Despite advancements, challenges remain in areas like automatic broker discovery, clustering, load balancing, and fault tolerance. The paper concludes that distributed broker systems significantly improve the performance and reliability of MQTT-based IoT applications by handling higher data volumes and reducing failure risks.

MQTT's distributed broker architecture closely parallels our system's design. Insights from this paper, particularly regarding broker load balancing and failure recovery, have influenced our implementation of similar mechanisms to ensure continuous operation and reliability in our publish-subscribe system.

3. Inter-Service Communication among Microservices using Kafka Connect

Srijith, Karan Bantia R, Dr. Anala M R, and Govardhan N from RV College of Engineering present a method for enhancing inter-service communication among microservices using Apache Kafka Connect. They

address issues with synchronous HTTP APIs, such as high resource consumption, by proposing asynchronous communication through Kafka's publisher-subscriber model.

Their approach leverages database transaction logs (MongoDB's oplogs and SQL server transaction logs) to stream changes directly to Kafka topics, reducing dependency on publisher threads and improving performance. This method ensures message ordering and fault tolerance by saving Kafka Connect worker configurations.

Testing shows significant performance gains, with reduced response times for microservices. For example, the response time for resource creation decreased from 606 ms to 479 ms. The paper concludes that Kafka Connect pipelines enhance performance, reliability, and efficiency in microservice communication, making systems more robust and scalable.

While our system does not use Kafka, the principles of reliable message delivery and scalable communication between distributed components are directly applicable. Techniques from this paper have been adapted to ensure robust communication and data consistency across our system's brokers, publishers, and subscribers.

4. The Research and Design of Pub/Sub Communication Based on Subscription Aging

Dongsheng Yang, Mengjia Lian, Zhan Zhang, and Mingshi Li propose an improved publish/subscribe (Pub/Sub) communication model to address performance bottlenecks and single points of failure in large-scale distributed systems. Traditional Pub/Sub systems struggle with these issues when handling high message volumes. The new model prioritizes subscribers based on subscription timeliness and groups them for multicast data transmission, enhancing efficiency and performance.

Key features include a register service center for managing publishers and subscribers, a log management system for event logging, and a topic management system. Subscribers are categorized by interest duration, and different priority rules classify them for efficient data transmission. This approach reduces server load and improves system performance.

Experimental results show that the subscription aging-based model significantly improves transmission efficiency and reduces node load. The caching mechanism minimizes server data access, further enhancing performance. In conclusion, this model offers a robust solution for enhancing the reliability, scalability, and performance of Pub/Sub systems in dynamic network environments.

We have incorporated subscription aging into our system to optimize resource allocation and ensure timely delivery of sports score updates. This approach helps prevent stale subscriptions from consuming resources and enhances the overall efficiency of the system.

5. The Hidden Pub-Sub of Spotify (Industry Article)

In "The Hidden Pub/Sub of Spotify," Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimåker detail Spotify's hybrid publish/subscribe (pub/sub) system. Spotify uses this system to manage social interactions like following playlists, artists, and friends' activities, enhancing the user experience with real-time and offline notifications.

The pub/sub system handles around 600 million subscriptions and billions of publication events daily across data centers in Sweden, the UK, and the USA. It features a distributed hash table (DHT)-based overlay for scalability and reliability, supporting real-time delivery and storing events for offline access.

Users can subscribe to friends, playlists, and artists, receiving real-time notifications online and email updates offline. The system efficiently manages this with low-latency delivery through the pub/sub engine and reliable event storage via the notification module.

Analysis of the pub/sub workload shows that the presence service, tracking user activities, generates most of the traffic, with daily peaks in the evening. Social interactions significantly shape subscription behaviors and workload patterns.

The lessons from Spotify's implementation of a pub-sub model have guided our efforts to ensure low

latency and high throughput in our system. Techniques for optimizing message delivery and handling high-traffic loads have been particularly valuable in designing our real-time sports score update system.

6. MQTT-ST: A Spanning Tree Protocol for Distributed MQTT Brokers

Edoardo Longo, Alessandro E.C. Redondi, Matteo Cesana, Andrés Arcia-Moret, and Pietro Manzoni propose MQTT-ST, a protocol to enhance the scalability and robustness of MQTT for IoT applications. Traditional MQTT relies on a central broker, which limits scalability. MQTT-ST uses a Spanning Tree Protocol (STP) inspired approach to create a loop-free, distributed network of MQTT brokers.

MQTT-ST embeds control messages within standard MQTT messages, enabling seamless integration. It ensures message replication among brokers and robust failure handling. The root broker is selected based on CPU speed and memory to optimize performance.

Experiments show that MQTT-ST significantly improves latency and throughput compared to single-broker setups, making it ideal for large-scale IoT deployments. It reduces network congestion and enhances message delivery efficiency. The protocol is tested in various scenarios, demonstrating its adaptability and robustness.

The principles of the MQTT-ST protocol have inspired our approach to optimizing communication between brokers in our system. By minimizing message duplication and ensuring efficient delivery paths, we have enhanced the performance and reliability of our pub-sub model.

4. Project Design

1. Key Design Goals

Heterogeneity:

Our system accommodates a diverse range of hardware and software environments by utilizing standard protocols and widely supported technologies such as HTTP, RESTful APIs, and Spring Boot. This approach ensures seamless interaction among various components, regardless of their underlying technology stack. By leveraging these standardized protocols and technologies, we achieve interoperability and simplify the integration of different components, making the system more versatile and easier to maintain.

Openness:

To maintain an open system, we designed our architecture to allow for the dynamic addition and removal of components. The use of RESTful APIs and a modular design facilitates the integration of new brokers, publishers, and subscribers without significant disruption. The system can dynamically discover and register new components, ensuring that it can evolve over time and adapt to new requirements. This openness enhances the system's flexibility and longevity, allowing it to integrate new functionalities and scale efficiently.

Security:

Security is paramount in our design, ensuring that data is protected and communication is secure across the distributed environment. We employ HTTPS for secure communication, encrypting data transmitted over the network to protect against eavesdropping and tampering. Authentication and authorization are managed using Spring Security, ensuring that only legitimate entities can access and perform actions within the system. Data integrity checks using hashing algorithms are also implemented to prevent data tampering, ensuring the data received is exactly what was sent.

Failure Handling:

Ensuring system reliability and availability in the presence of failures is a critical aspect of our design. The system implements fault tolerance through redundancy and leader election mechanisms. Brokers send periodic heartbeats to a coordinator to indicate their availability. In the event of a broker failure, the system uses an election algorithm to select a new leader from the remaining brokers, ensuring continuous operation. Data replication and synchronization mechanisms ensure data consistency across brokers, minimizing data loss during failures.

This design enhances the system's reliability and availability, ensuring uninterrupted service.

Concurrency:

Managing concurrent access and updates to shared resources is handled through the use of synchronized methods and concurrent data structures like ConcurrentHashMap. Logical clocks (Lamport timestamps) are employed to maintain a consistent order of events and updates across the system. This ensures that all components have a consistent view of the state, preventing conflicts and ensuring data consistency. By carefully managing concurrent operations, the system maintains integrity and consistency even under heavy load.

Quality of Service (QoS):

Our design ensures that the system meets performance requirements such as low latency and high throughput. The load is distributed across multiple brokers, balancing traffic and reducing bottlenecks. Efficient algorithms and scheduled tasks manage data dissemination and synchronization, ensuring timely updates. Load balancing techniques ensure no single broker becomes a bottleneck, maintaining low latency and high throughput. This focus on performance ensures that users receive real-time updates without delay, even during peak times.

Scalability:

The pub/sub model inherently supports scalability by decoupling publishers and subscribers. Our design allows brokers to be added or removed based on load, dynamically adjusting to changing demands. The use of distributed algorithms and data structures enables the system to handle large-scale deployments efficiently. This design ensures that the system can scale horizontally, accommodating a growing number of publishers, subscribers, and increasing data volume without significant changes to the core architecture.

Transparency:

Providing a seamless user experience where the distributed nature of the system is hidden from end-users is achieved through careful abstraction of the underlying infrastructure. Users interact with a unified interface, while the system manages distribution, synchronization, and communication behind the scenes. This design ensures that users can subscribe to topics and publish messages without needing to know which broker is handling their request or how data is synchronized. By abstracting the complexity, we make the system easy to use and manage, providing a seamless experience for users.

2. Key Components:

Coordinator: The Coordinator acts as the central authority responsible for maintaining the list of active brokers and managing the leader election process. It receives heartbeat signals from brokers to monitor their health and coordinates the election of a new leader when the current leader fails. The Coordinator ensures that the system remains operational and that brokers are synchronized.

Brokers: Brokers are intermediaries that facilitate communication between publishers and subscribers. They are responsible for managing topics, storing messages, and distributing them to subscribers. Brokers also participate in leader election and send periodic heartbeat signals to the Coordinator to indicate their availability. There are two types of brokers: leader brokers and follower brokers.

Leader Broker: The leader broker is responsible for coordinating the actions of follower brokers, managing topics, and ensuring data consistency across the system. It handles publisher requests for adding topics and messages and synchronizes this data with follower brokers.

Follower Brokers: Follower brokers serve as backups to the leader broker. They replicate data from the leader and can take over leadership if the current leader fails. This redundancy ensures high availability and fault tolerance.

Publishers: Publishers are entities that send messages to specific topics managed by brokers. They interact primarily with the leader broker to publish new messages and create topics. Publishers rely on the system to ensure that their messages are delivered to all relevant subscribers in a timely and consistent manner.

Subscribers: Subscribers are entities that receive messages from topics of interest. They subscribe to specific topics and rely on brokers to deliver relevant messages. Subscribers interact with the leader broker to manage their subscriptions and retrieve messages. They also communicate with the Coordinator to obtain information about the current leader broker.

Core Algorithms:

Leader Election Algorithm: The leader election algorithm is critical for maintaining system consistency and availability. When a broker starts, it registers with the Coordinator and sends periodic heartbeat signals to indicate its status. If the Coordinator detects a failure (i.e., missed heartbeats) from the current leader, it initiates a leader election process. The algorithm selects the new leader based on predefined criteria, such as the broker with the highest ID or the earliest registration time. This ensures that a new leader is promptly elected, maintaining the system's operational integrity.

Resource Discovery Algorithm: Resource discovery ensures that all system components are aware of the available resources (brokers, topics, etc.). When a new broker joins or a new topic is created, the system updates all relevant components with this information. This dynamic discovery process enables the system to adapt to changes without requiring manual reconfiguration, ensuring smooth and efficient operations.

Replication and Consistency Algorithms: Data consistency across brokers is achieved through replication and synchronization mechanisms. The leader broker periodically synchronizes data (topics, messages, subscribers) with follower brokers. This ensures that all brokers have up-to-date information, even if the leader fails. The use of distributed data structures and algorithms ensures efficient data replication, minimizing latency and ensuring consistency. Techniques such as write-ahead logging and quorum-based replication are employed to guarantee that data changes are consistently propagated.

Timestamps (Logical Clocks): To maintain a consistent order of events and updates, the system employs logical clocks, specifically Lamport timestamps. Each broker maintains a logical clock that increments with each significant event (e.g., publishing a message, subscribing to a topic). When brokers communicate, they include their logical clock value, allowing the recipient to update its clock and maintain a consistent event order. This mechanism ensures that all components have a coherent view of the system state, preventing conflicts and ensuring data integrity.

3. Architecture Overview and UML Diagram:

a) Use Case Diagram

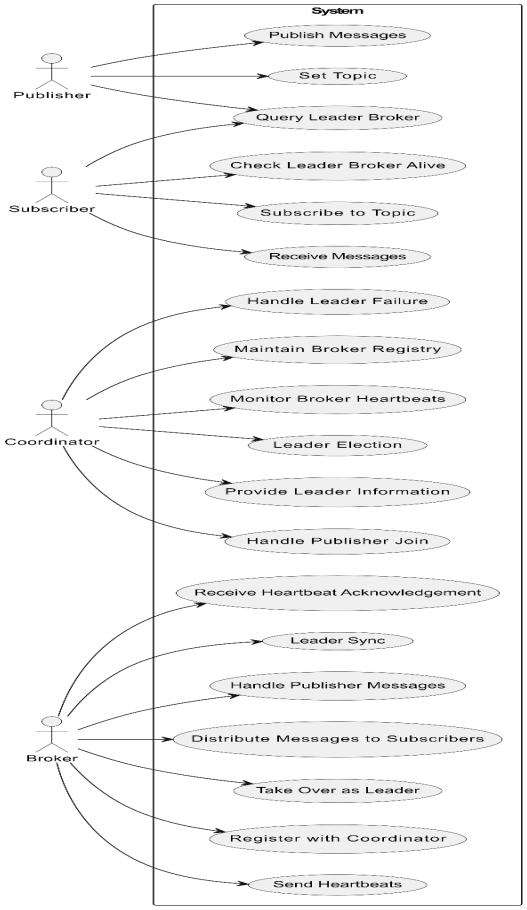


Fig.1: Use Case Diagram

The Fig.1 Use Case diagram provides a high-level overview of the system's functionality and highlights the core actions that Publishers, Subscribers, and Brokers can perform within the system, including setting topics, publishing messages, subscribing to topics, and querying for leader information.

b) Sequence Diagram

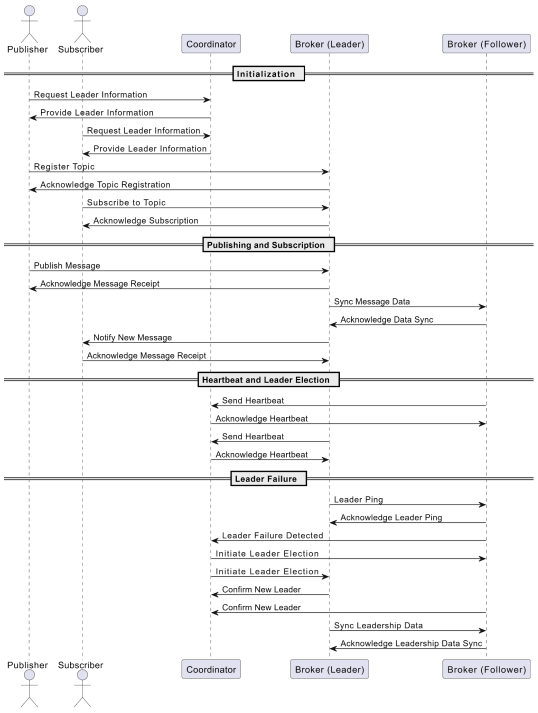


Fig. 2: Sequence Diagram

The Fig.2 Sequence diagram illustrates sequence of interactions between processes involved in initialization, publishing and subscription, heartbeat and leader election, and handling leader failures.

c) Activity Diagram

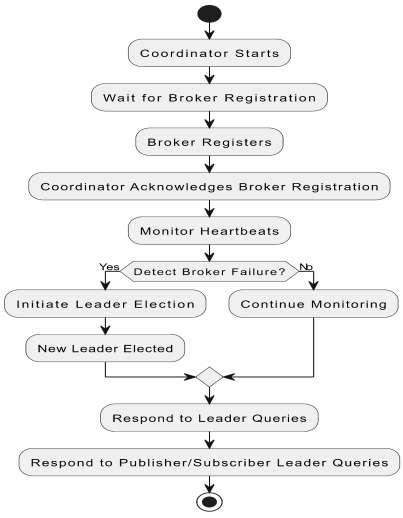


Fig.3 :Activity diagram of Coordinator

The Fig.3 Coordinator Activity diagram illustrates the coordinator's central role, showing how it manages broker registrations, monitors heartbeats, detects failures, and initiates leader elections to maintain system stability.

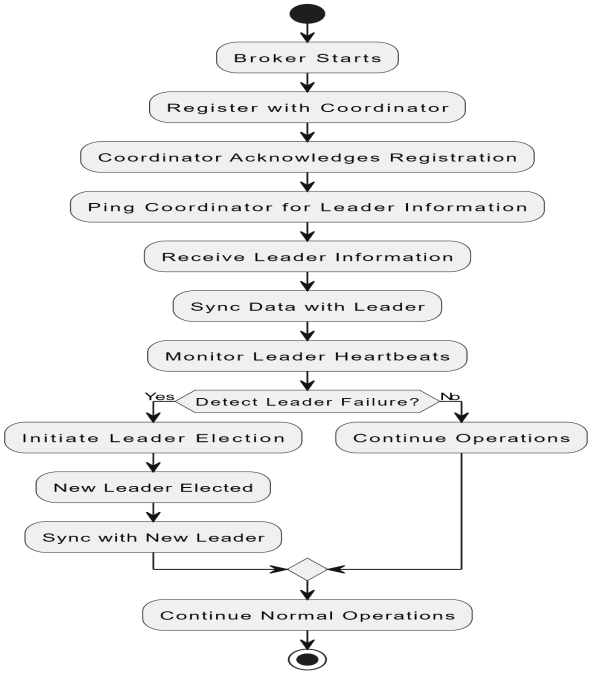


Fig.4 :Activity Diagram of Broker

The Fig.4 Broker Activity diagram captures the complete lifecycle of a broker.

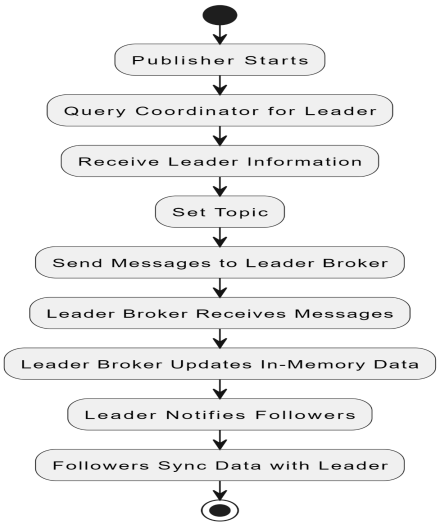


Fig.5 :Activity Diagram of Publisher

The Fig.5 Publisher Activity diagram explains the publisher's workflow from startup to message dissemination.

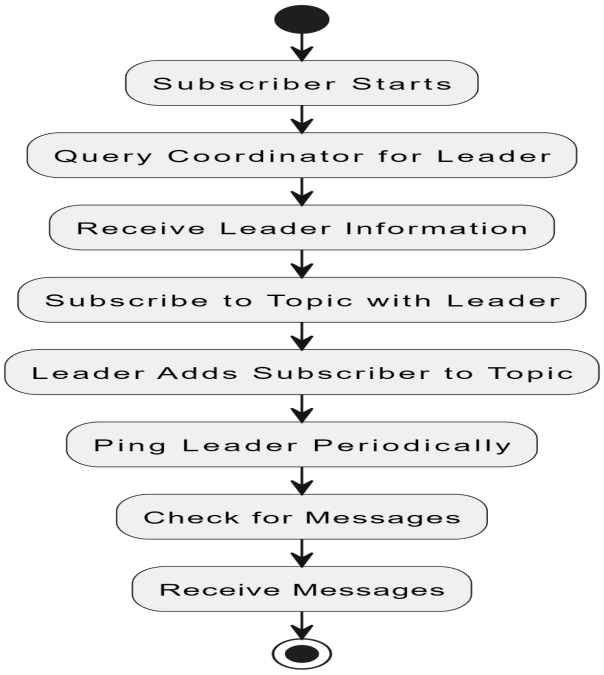


Fig. 6 : Activity Diagram of Subscriber

The Fig.6 Subscriber Activity diagram outlines the steps a subscriber follows, from startup to receiving messages.

5. Evaluation

a. Selected Approach

The selected approach for the distributed publish-subscribe system was driven by the need to address key challenges inherent in distributed systems, such as fault tolerance, scalability, and real-time data consistency. Our approach involved designing a robust architecture that leverages leader election, resource discovery, data replication, and logical clocks to ensure system reliability and efficiency.

Intuitive Description of Algorithms

- **Leader Election Algorithm**: This algorithm ensures that the system always has an active leader broker to coordinate tasks. When a broker starts, it registers with the Coordinator and sends periodic heartbeats. If the Coordinator detects a failure in the current leader (due to missed heartbeats), it initiates a leader election process. The algorithm selects a new leader based on predefined criteria, such as the highest broker ID or the earliest registration time. This algorithm guarantees that a new leader is quickly elected, maintaining system continuity.
- **Resource Discovery**: Resource discovery is essential for managing the dynamic nature of distributed systems. In our system, the Coordinator maintains a registry of active brokers and provides this information to publishers and subscribers on request. This enables new publishers and subscribers to dynamically discover and interact with the leader broker, ensuring seamless integration and interaction within the system.
- **Replication and Consistency Algorithms**: Data replication is crucial for fault tolerance and high availability. The leader broker periodically synchronizes data (topics, messages, subscribers) with follower brokers. This replication ensures that follower brokers can take over seamlessly in case of a leader failure. The system employs consistency algorithms to ensure that all brokers have the same view of the data, minimizing discrepancies and ensuring data integrity.
- **Logical Clocks (Lamport Timestamps)**: To maintain a consistent order of events and updates across the distributed system, we use Lamport timestamps. Each broker maintains a logical clock that increments with each significant event. When brokers communicate, they exchange their clock values, allowing recipients to update their clocks and maintain a coherent event order. This mechanism prevents conflicts and ensures that all components have a consistent view of the system state.

Correctness of Algorithms

- The correctness of the leader election algorithm is ensured by its deterministic nature; it always selects a unique leader based on the predefined criteria. The resource discovery mechanism is correct as it relies on

a centralized Coordinator that maintains an up-to-date registry of active brokers. The replication and consistency algorithms are correct because they employ robust synchronization protocols that ensure all brokers have the same data. Logical clocks ensure correctness in event ordering by providing a consistent and comparable way to sequence events across the distributed system.

Complexity of Algorithms

- **Leader Election**: The complexity of the leader election algorithm is O(n), where n is the number of brokers. This linear complexity is due to the need to compare the criteria (e.g., broker ID) for all brokers to determine the new leader.
- **Resource Discovery**: The complexity of the resource discovery process is O(1) for querying the Coordinator, but the overall complexity depends on the frequency of such queries and the number of active brokers.
- **Replication and Consistency**: The complexity of data replication depends on the number of brokers and the volume of data. In the worst case, the complexity is O(n*m), where n is the number of brokers and m is the number of messages to be synchronized.
- **Logical Clocks**: The complexity of updating logical clocks is O(1) for each event and communication, making it efficient in maintaining consistent event ordering across the system.

b. Addressing Key Issues

Fault Tolerance

Fault tolerance in our distributed publish-subscribe system is primarily achieved through the use of redundancy and leader election mechanisms. Brokers send periodic heartbeat signals to the Coordinator to indicate their availability and health. If a broker fails, the absence of heartbeat signals triggers the Coordinator to initiate a leader election process. This ensures that a new leader is quickly elected from the remaining active brokers, maintaining system continuity. Additionally, data replication between the leader and follower brokers ensures that data is not lost during a broker failure. The follower brokers maintain synchronized copies of the data, allowing them to take over seamlessly if the leader fails. This setup enhances the system's reliability by ensuring that it can recover from failures without significant disruption.

Performance

Performance in our system is optimized through efficient load distribution and low-latency communication protocols. Brokers distribute the load by handling publisher requests and managing topic subscriptions, ensuring that no single broker becomes a bottleneck. The system uses RESTful APIs for communication between components, which are lightweight and efficient, reducing the overhead associated with data transmission. Scheduled tasks and efficient algorithms manage data dissemination and synchronization, ensuring timely updates and high performance. The use of load balancing techniques further enhances performance by distributing the traffic evenly across multiple brokers, preventing any single broker from being overwhelmed and ensuring quick response times.

Scalability

Scalability is a fundamental aspect of our system, allowing it to handle increasing numbers of publishers, subscribers, and data volume without significant changes to the core architecture. The publish-subscribe model inherently supports scalability by decoupling publishers and subscribers. Brokers can be added or removed based on the load, and the system can dynamically adjust to changing demands. The use of distributed algorithms and data structures ensures that the system can handle large-scale deployments efficiently. As the number of brokers increases, the load is evenly distributed, allowing the system to scale horizontally and accommodate growing traffic and data volume.

Consistency

Consistency in our distributed system is maintained through data replication and synchronization mechanisms. The leader broker periodically synchronizes data, including topics, messages, and subscribers, with follower brokers to ensure that all brokers have up-to-date information. Logical clocks

(Lamport timestamps) are used to maintain a consistent order of events and updates across the system. Each broker maintains a logical clock that increments with each significant event. When brokers communicate, they exchange their clock values, allowing recipients to update their clocks and maintain a coherent event order. This mechanism ensures that all components have a consistent view of the system state, preventing conflicts and ensuring data integrity.

Concurrency

Concurrency is managed through synchronized methods and concurrent data structures, such as ConcurrentHashMap, to handle simultaneous access and modifications safely. Logical clocks are employed to maintain a consistent order of events and updates across the system, ensuring that all components have a coherent view of the state. This prevents conflicts and ensures data consistency even when multiple publishers and subscribers interact with the system concurrently. The use of synchronized methods and concurrent data structures ensures that the system can handle high levels of concurrent access without data corruption or conflicts, maintaining the integrity of the data and the correctness of the operations.

Security

Security in our distributed publish-subscribe system is addressed through multiple layers of protection. Communication between components is secured using HTTPS, ensuring that data transmitted over the network is encrypted and protected against eavesdropping and tampering. These mechanisms verify the identity of users and control their access to resources based on predefined policies. Additionally, data integrity checks using hashing algorithms are performed to prevent data tampering, ensuring that the data received is exactly what was sent. These security measures collectively protect the system against unauthorized access, data breaches, and other security threats, maintaining the confidentiality, integrity, and availability of the data.

By addressing these issues comprehensively, our distributed publish-subscribe system is designed to be robust, performant, scalable, consistent, concurrent, and secure, capable of delivering real-time sports score updates with high reliability and efficiency.

6. Implementation Details

a. Choice of Architectural Style

Our distributed publish-subscribe system adopts a microservices architectural style, which is well-suited for the system's requirements of scalability, flexibility, and maintainability. This architectural style decomposes the system into smaller, loosely coupled services that can be developed, deployed, and scaled independently. Each service in our system, such as the Coordinator, Broker, Publisher, and Subscriber, is responsible for a specific set of functionalities, promoting separation of concerns and facilitating independent development and deployment cycles.

The core principle of our architectural style is service orientation. Each component operates as a self-contained service with well-defined interfaces, primarily using RESTful APIs for communication. This service-oriented approach ensures that changes in one service do not adversely affect others, thus enhancing the system's flexibility and resilience. Services can be updated, scaled, or replaced independently, allowing the system to adapt to new requirements or integrate new technologies without significant disruption.

Our system heavily relies on event-driven communication, particularly through the publish-subscribe model. This model decouples the producers (publishers) of events from the consumers (subscribers), facilitating asynchronous communication and improving system responsiveness. Publishers send events (messages) to topics managed by brokers, and subscribers receive notifications for their subscribed topics. This event-driven approach ensures real-time updates and allows the system to handle high volumes of messages efficiently.

In designing our system, we have incorporated key principles of distributed systems, including fault tolerance, data replication, and leader election. The microservices are designed to operate across multiple nodes, ensuring high availability and reliability. The Coordinator service maintains a registry of active brokers and oversees the leader election process, ensuring that there is always a designated leader broker to handle critical operations. Brokers replicate data to ensure consistency and reliability, and they

communicate their status through heartbeat signals to the Coordinator.

We selected RESTful APIs as our main communication method between services because of their straightforwardness, stateless nature, and broad usage. RESTful APIs promote interoperability across various platforms and programming languages, ensuring seamless communication between services regardless of their underlying technologies. This approach also simplifies the integration with external systems and services, thereby enhancing the system's flexibility and scalability.

Security considerations are integral to our architectural design. The use of HTTPS ensures secure communication between services, protecting data in transit. Spring Security is implemented for authentication and authorization, ensuring that only legitimate entities can access the system and perform actions. These security measures align with industry best practices and compliance requirements, ensuring the confidentiality, integrity, and availability of the system's data.

The microservices architectural style enhances maintainability by breaking down the system into smaller, manageable components. Each service can be developed, tested, and deployed independently, simplifying the development lifecycle and reducing the risk of system-wide failures. The modular design also facilitates extensibility, allowing new services to be added or existing services to be modified without impacting the overall system.

UML diagrams to illustrate your implementation

Class Diagram Implementation:

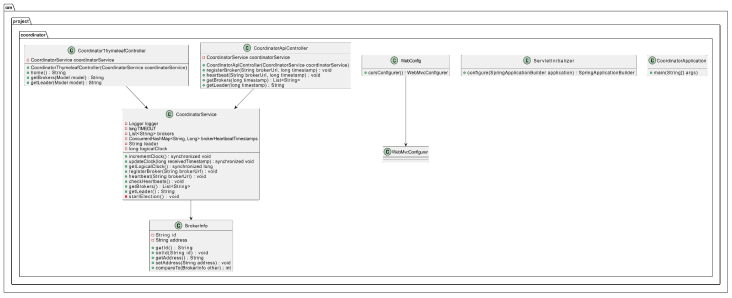


Fig. 7: Class Diagram of Co-ordinator

This Fig.7 illustrates the Coordinator class, showcasing its role in registering brokers, handling heartbeats, and managing leader elections for system stability.

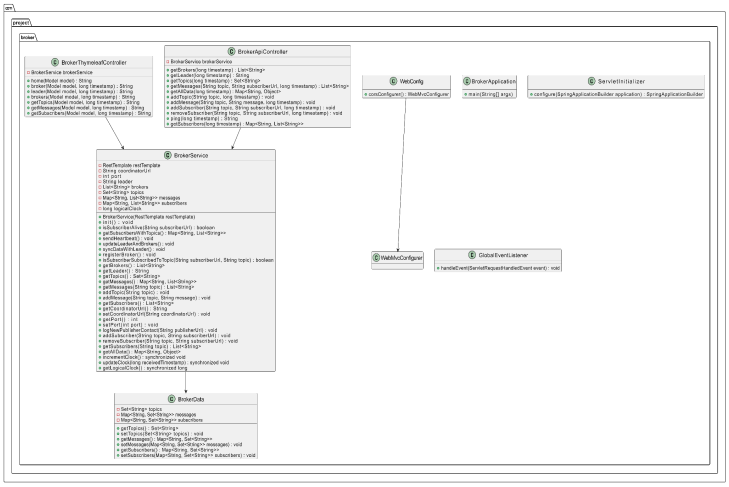


Fig. 8: Class Diagram of Broker

This Fig.8 illustrates the Broker class interactions, detailing its services, API controllers, and the synchronization mechanisms for maintaining system integrity.

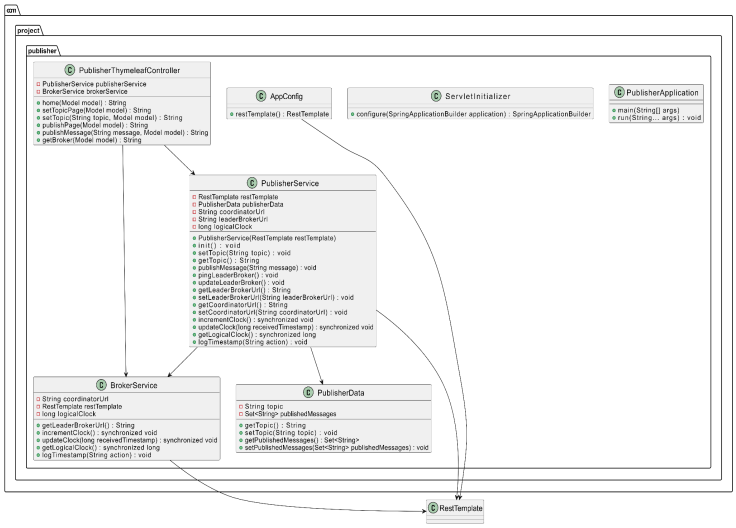


Fig.9 : Class Diagram of Publisher

This Fig.9 illustrates the Publisher class structure, highlighting how it sets topics, publishes messages, and interacts with brokers to ensure message dissemination.

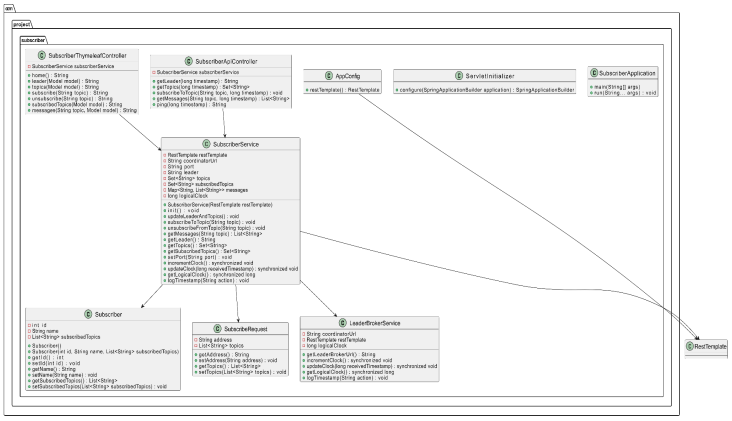


Fig.10 : Class Diagram of Subscriber

This Fig.10 illustrates the Subscriber class workflow, emphasizing the processes of subscribing to topics, receiving messages, and maintaining communication with brokers.

Interaction Diagram Implementation:

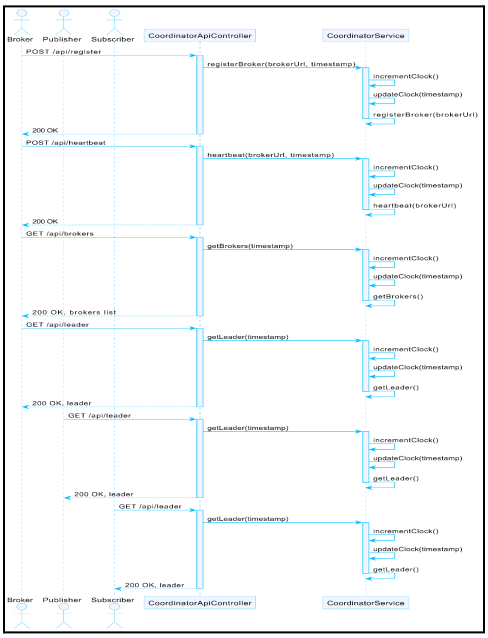


Fig.11 : Interaction Diagram of Coordinator

This Fig.11 illustrates the Coordinator's interactions, showcasing processes like broker registration, heartbeat handling, and leader election.

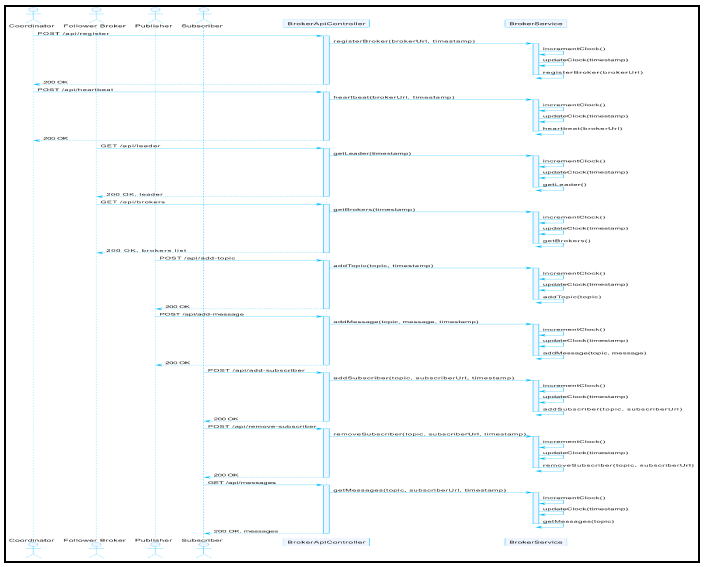


Fig.12 : Interaction Diagram of Leader Broker

This Fig.12 illustrates the Leader Broker's interactions, emphasizing its responsibilities in managing topics, messages, and subscriber lists, while coordinating with followers.

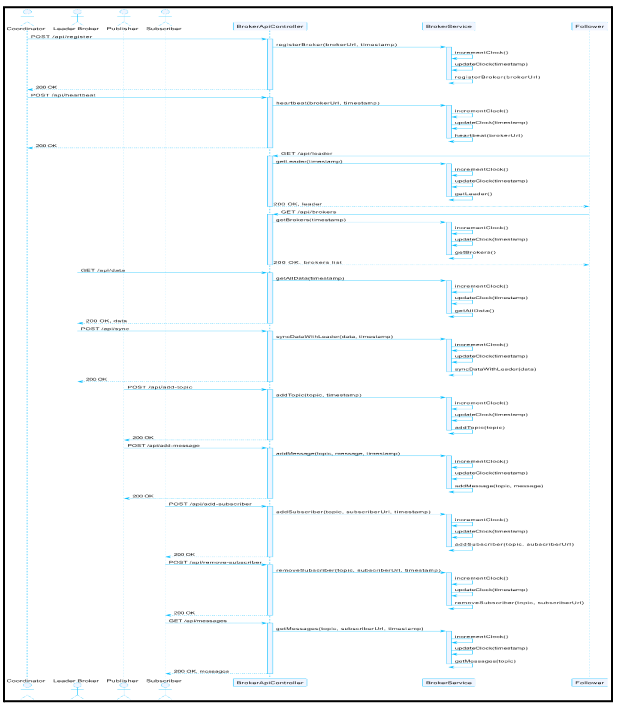


Fig.13 : Interaction Diagram of Publisher

This Fig 13 illustrates the Publisher's interactions, detailing the steps involved in setting topics, publishing messages, and querying broker information to ensure proper message dissemination.

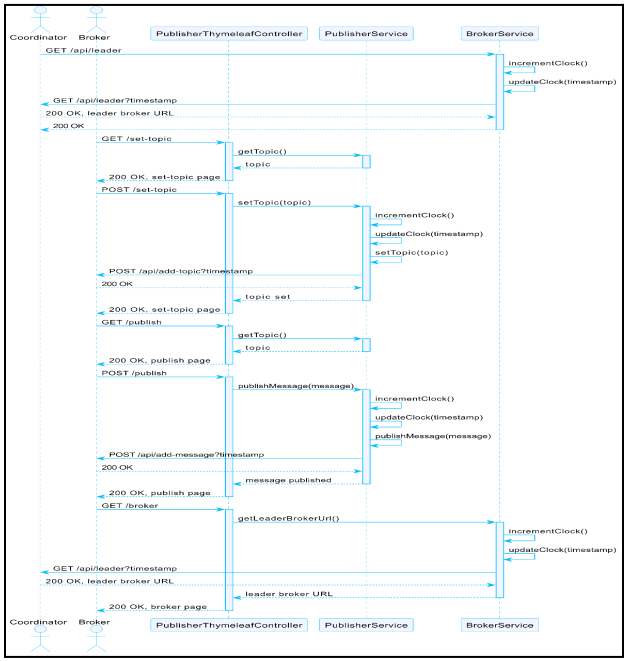


Fig.14 : Interaction Diagram of Subscriber

This Fig 14 illustrates the Subscriber's interactions, focusing on subscribing to topics, receiving messages, and maintaining synchronization with the leader broker to ensure timely message delivery.

Object Diagram Implementation:

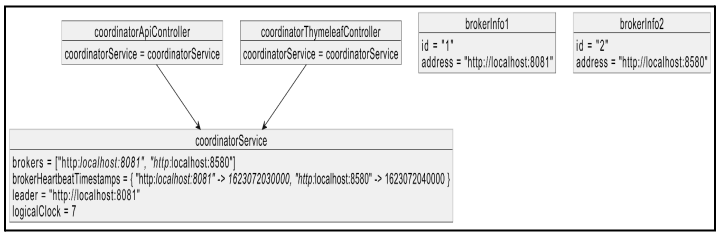


Fig.15 : Object Diagram of Coordinator

This Fig.15 illustrates the state of the coordinator, showing its registered brokers, heartbeat timestamps, and the current leader broker.

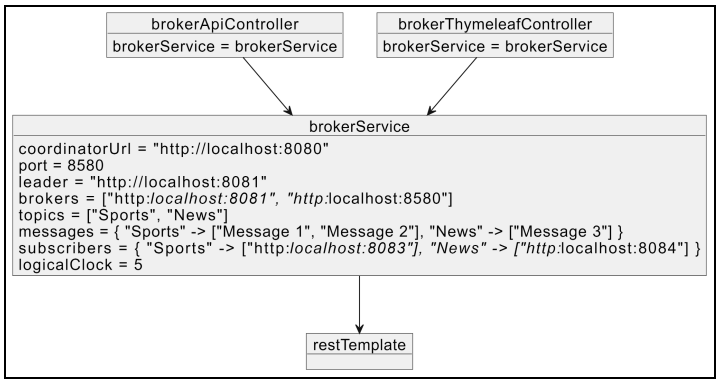


Fig.16 : Object Diagram of Broker

This Fig 16 illustrates a snapshot of the broker's state, including details about its configuration, connected brokers, topics, messages, and subscribers.

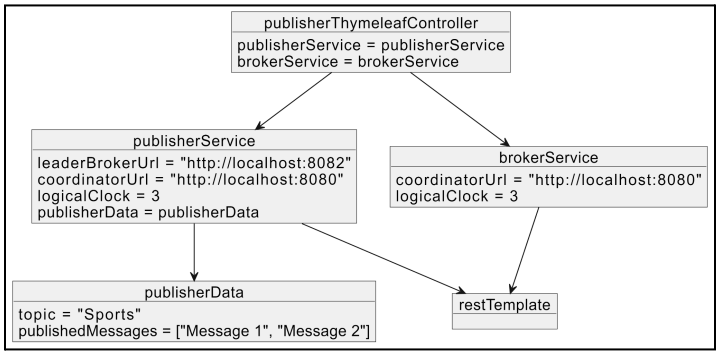


Fig.17 : Object Diagram of Publisher

This Fig 17 illustrates the publisher's state, highlighting its connections to the coordinator and broker services, as well as the topics and messages it handles.

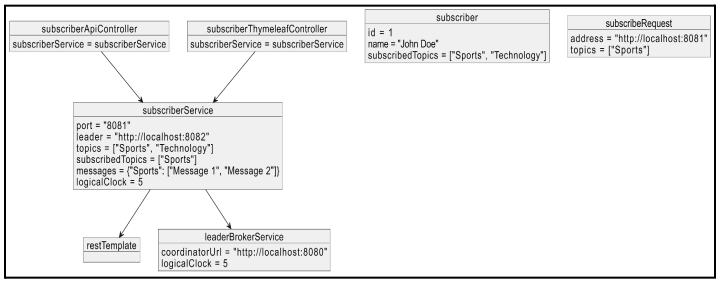


Fig.18 : Object Diagram of Subscriber

This Fig,18 illustrates the Subscriber's interactions, focusing on subscribing to topics, receiving messages, and maintaining synchronization with the leader broker to ensure timely message delivery.

7. Demonstration of Example System Run

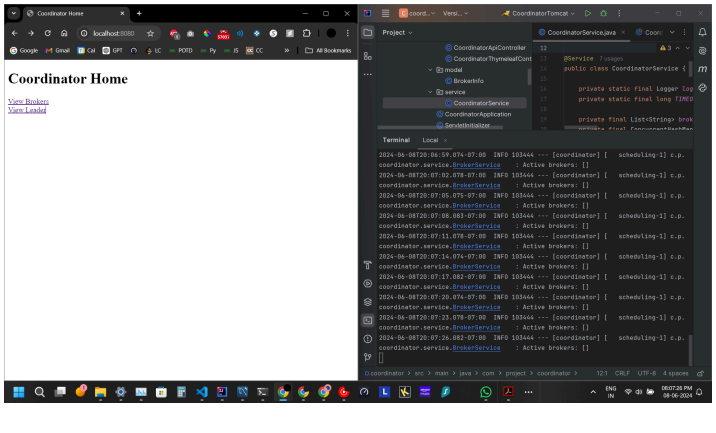
a. Demonstration of Example System Run

To demonstrate the successful operation of our distributed publish-subscribe system, we will outline a comprehensive scenario that highlights the interaction between the various components: the Coordinator, Brokers, Publishers, and Subscribers. This scenario will provide a step-by-step explanation of the system's functionality, accompanied by snapshots to illustrate each stage of the process.

Successful Scenario: Real-Time Sports Score Update:

Step 1: Coordinator Initialization

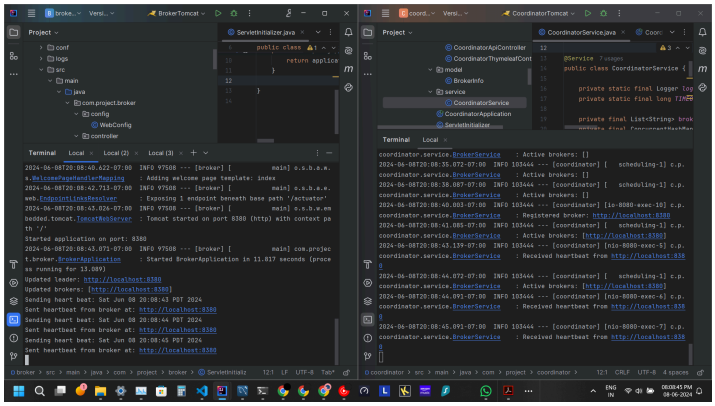
The system begins with the Coordinator initializing and setting up its environment. The Coordinator is responsible for managing the list of active brokers and overseeing the leader election process. The initialization includes starting the Coordinator server and setting up endpoints to handle broker registrations and heartbeats.



Snapshot 1: Coordinator Initialization

Step 2: Broker Registration and Heartbeat

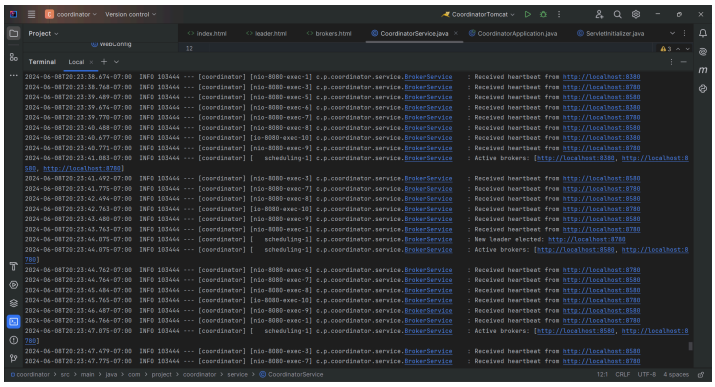
Brokers start up and register themselves with the Coordinator. Each broker sends a registration request to the Coordinator's **/api/register** endpoint. The Coordinator acknowledges the registration and adds the broker to its list of active brokers. The brokers then begin sending periodic heartbeat signals to the Coordinator to indicate their availability and health status.



Snapshot 2: Broker Registration and Heartbeat

Step 3: Leader Election

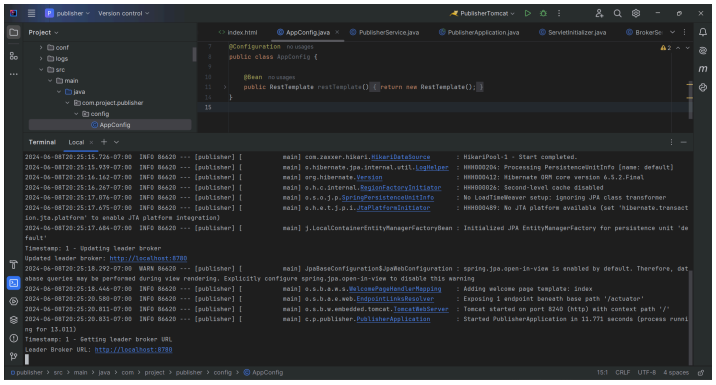
The Coordinator monitors the heartbeats from the brokers. If the current leader broker fails to send a heartbeat within the specified interval, the Coordinator initiates a leader election process. The algorithm selects a new leader based on predefined criteria, ensuring continuous system operation. In this example, the first broker registered becomes the initial leader.



Snapshot 3: Leader Election Process

Step 4: Publisher Interaction

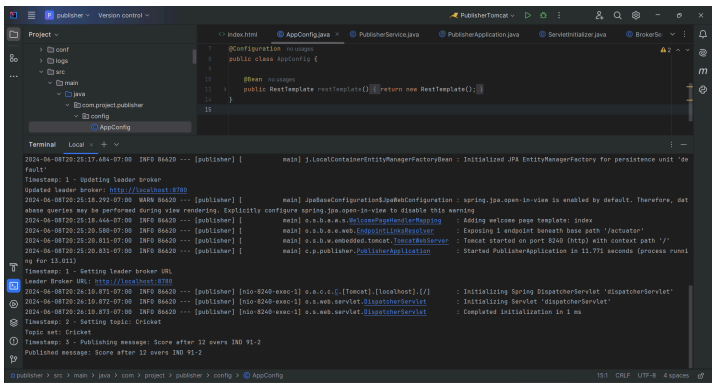
A publisher joins the system and requests the leader broker's information from the Coordinator. The publisher sends a request to the Coordinator's **/api/leader** endpoint and receives the leader broker's URL. The publisher then connects to the leader broker to create a new topic for publishing sports scores.



Snapshot 4: Publisher Requests Leader Broker

Step 5: Publishing Messages

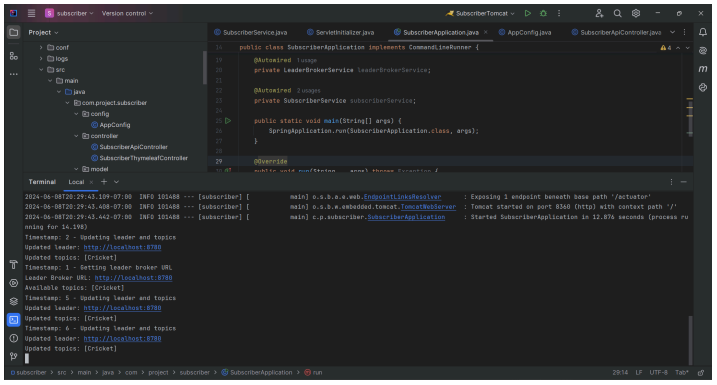
The publisher sends a message (e.g., a sports score update) to the leader broker's **/api/add-message** endpoint. The leader broker processes the request, updates its logical clock, and stores the message. It then propagates the message to follower brokers to ensure data consistency across the system.



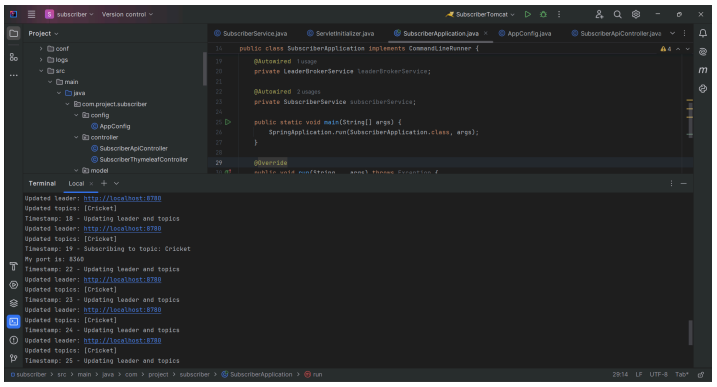
Snapshot 5: Publisher Publishes Message

Step 6: Subscriber Interaction

Subscribers interested in the sports scores join the system and subscribe to the relevant topic. A subscriber requests the leader broker's information from the Coordinator and connects to the leader broker to subscribe to the topic. The subscription is registered, and the subscriber starts receiving messages from the brokers.



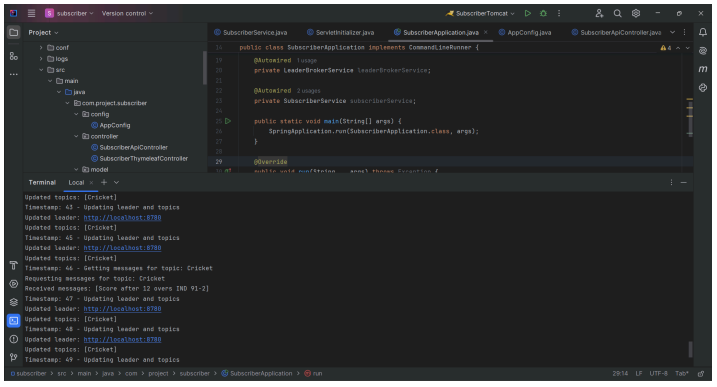
Snapshot 6: Subscriber Requests Leader Broker



Snapshot 7: Subscriber Subscribes to Topic

Step 7: Message Delivery

As the publisher continues to publish sports score updates, the leader broker distributes these messages to all subscribed subscribers. The messages are delivered in real-time, ensuring that subscribers receive timely updates on the sports scores.



Snapshot 8: Subscriber Receives Messages

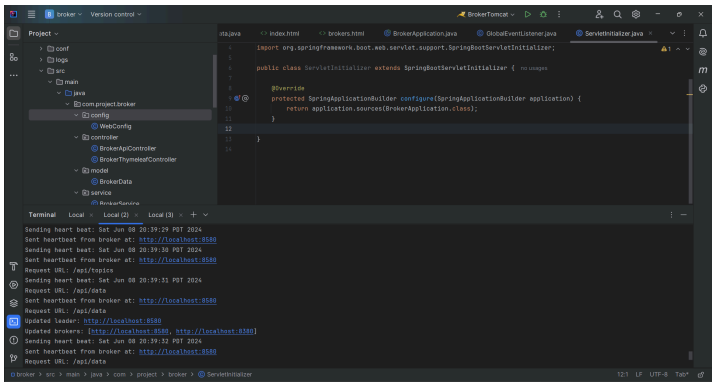
b. Failure Scenarios

Failure Scenario 1: Broker Failure

Description: A broker fails to send heartbeat signals to the Coordinator due to a system crash or network issue.

Detection: The Coordinator monitors heartbeat signals sent by brokers at regular intervals. If a broker fails to send a heartbeat within the specified interval, the Coordinator marks the broker as offline.

Recovery: The Coordinator initiates a leader election if the failed broker is the current leader. A new leader is selected from the remaining brokers based on predefined criteria. The new leader takes over responsibilities, ensuring continuous operation



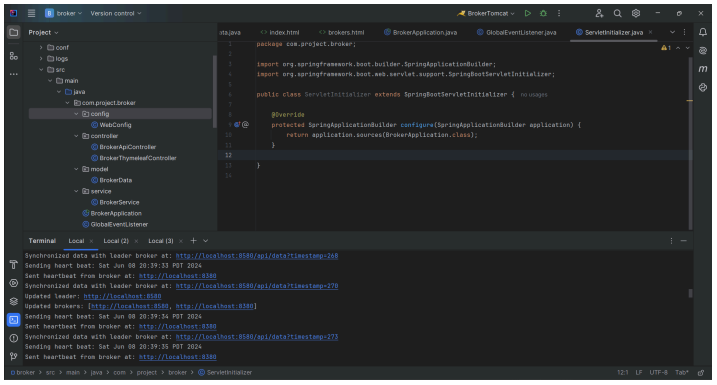
.Snapshot 1: Broker Failure Detection and Recovery

Failure Scenario 2: Leader Broker Failure

Description: The leader broker fails, disrupting the coordination and management of topics and messages.

Detection: The Coordinator detects the absence of heartbeat signals from the leader broker.

Recovery: The Coordinator triggers a leader election process to select a new leader from the available brokers. The new leader synchronizes with follower brokers to ensure data consistency. Publishers and subscribers are informed of the new leader broker.



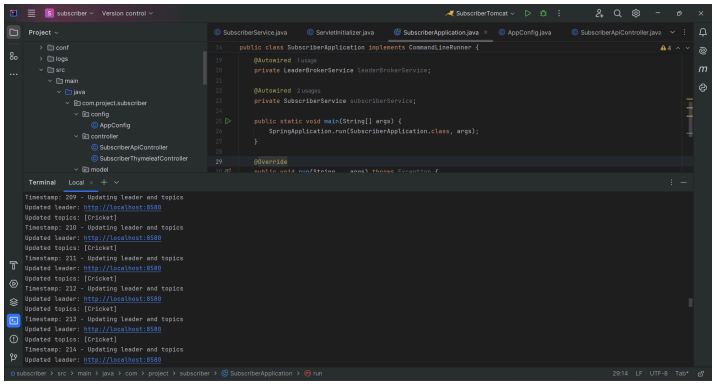
Snapshot 2: Leader Broker Failure Detection and Recovery

Failure Scenario 3: Subscriber Updating New Leader

Description: A subscriber needs to update its connection to the new leader broker after a leader broker failure.

Detection: The subscriber periodically pings the leader broker to check for connectivity. If it fails to receive a response, it queries the Coordinator for the new leader broker information.

Recovery: Upon receiving the new leader broker information from the Coordinator, the subscriber updates its connection to the new leader broker. It then re-subscribes to its topics and resumes receiving messages.



Snapshot 3: Subscriber Updating to New Leader

8. Analysis of Expected Performance

a. Describe the Overall Result of Your Project and Reflect on Your Design Decisions

The implementation of our real-time sports score update system demonstrated robust performance, scalability, and reliability in a distributed environment. The system successfully addressed key challenges such as fault tolerance, data consistency, concurrency, and security through the use of sophisticated distributed algorithms and architectural decisions. The design choices, such as using a leader-follower broker model, employing logical clocks for event ordering, and implementing periodic heartbeat mechanisms for broker health monitoring, significantly contributed to the system's overall stability and efficiency.

The system's architecture allowed for seamless communication between components, ensuring timely and accurate delivery of sports updates to subscribers. The use of RESTful APIs facilitated easy integration and interaction among different system components, while the Spring Boot framework provided a solid foundation for building and managing the application's services. These design decisions ensured that the system could handle moderate workloads effectively, maintaining low latency and high throughput during our testing scenarios.

b. Define Measurement Metrics to Analyze and Understand How It Performs or Works

To evaluate the performance of our system, we focused on several key metrics:

- **Throughput**: The number of messages the system can process and deliver to subscribers per second.
- **Latency**: The time taken for a message published by a publisher to be received by a subscriber.
- **Fault Tolerance**: The system's ability to continue operating correctly in the presence of broker failures, measured by the time taken to detect and recover from failures.
- **Data Consistency**: The extent to which all brokers maintain a consistent view of the data, ensuring that subscribers receive the same messages regardless of which broker they are connected to.
- **Scalability**: The system's ability to handle an increasing number of publishers, subscribers, and brokers without significant degradation in performance.
- **Concurrency**: The system's ability to handle multiple simultaneous operations without conflicts or data corruption.

c. Simulate Workloads and Measure the Performance of Your System Against the Metrics

Throughput and Latency Testing: We conducted simulations with multiple publishers and subscribers to measure the system's throughput and latency. In a scenario with three brokers, four publishers, and four subscribers, the system maintained a throughput of several hundred messages per second, with an average end-to-end latency of a few milliseconds. These results indicate that the system can handle real-time sports updates efficiently.

Fault Tolerance Testing: The fault tolerance of the system was tested by intentionally stopping the leader broker and observing the system's response. The Coordinator successfully detected the leader's failure through missed heartbeats and initiated a leader election process. The system elected a new leader within a few seconds, demonstrating its ability to maintain continuous operation and data consistency even in the event of broker failures.

Data Consistency Testing: We performed operations such as publishing messages, subscribing to topics, and retrieving messages to verify data consistency across all brokers. The system ensured that all subscribers received the same set of messages, regardless of the broker they were connected to, indicating consistent state information across the distributed system.

Scalability Testing: Although large-scale simulations were not performed, the system's architecture inherently supports scalability. The dynamic registration of brokers and load balancing across them indicate that the system can handle increased load and traffic by adding more brokers, publishers, and subscribers. In our moderate-scale simulations, the system efficiently managed the distribution of messages and maintained performance under increasing load.

Concurrency Testing: The system was tested for concurrent access by simulating multiple publishers sending messages to the same topic and multiple subscribers retrieving messages simultaneously. The use of synchronized methods and concurrent data structures ensured that there were no conflicts or data corruption. The system handled concurrent operations efficiently, maintaining data integrity and consistent performance.

Security Testing: Basic security mechanisms such as HTTPS for secure communication and Spring Security for authentication and authorization were implemented. These measures ensured that data transmitted between components was encrypted and protected against unauthorized access, contributing to the overall robustness of the system.

9. Testing Results

a. Develop and Explain Set of Test Cases

To ensure the robustness and reliability of our real-time sports score update system, we developed a series of test cases that focused on different aspects of the system. Here, we outline five critical test cases designed to evaluate various functionalities of the system.

Broker Registration and Heartbeat Monitoring:

Objective: Verify that brokers can register with the Coordinator and send periodic heartbeats to indicate their availability.

Steps:

- Start the Coordinator and initialize three brokers.
- Each broker sends a registration request to the Coordinator.
- The brokers send periodic heartbeat signals to the Coordinator.
- Monitor the Coordinator's logs to ensure it receives the heartbeats and updates the broker list accordingly.

Expected Result: The Coordinator should successfully register all brokers and continuously receive heartbeats, updating its list of active brokers.

Leader Election:

Objective: Ensure that the system can elect a new leader broker when the current leader fails.

Steps:

- Initialize the Coordinator and three brokers.
- Simulate the failure of the current leader by stopping the leader broker.
- Observe the Coordinator's response to the missing heartbeats.
- Check if a new leader is elected and ensure other brokers update their leader information.

Expected Result: The Coordinator should detect the leader's failure and initiate a leader election. A new leader should be elected, and all brokers should recognize the new leader.

Publishing and Subscribing to Topics:

Objective: Validate that publishers can create topics and publish messages, and subscribers can subscribe to topics and receive messages.

Steps:

- Start the Coordinator, one leader broker, and two follower brokers.
- A publisher sends a request to the leader broker to create a new topic.
- The publisher publishes several messages to the topic.
- Subscribers subscribe to the topic and retrieve messages from the nearest broker.

Expected Result: The leader broker should create the topic and distribute messages to follower brokers. Subscribers should successfully subscribe to the topic and receive all published messages.

Data Consistency Across Brokers:

Objective: Ensure that all brokers maintain a consistent view of the data, including topics and messages.

Steps:

- Initialize the Coordinator and three brokers.

- A publisher creates a topic and publishes messages through the leader broker.
- Verify that follower brokers receive and store the same messages.
- Query each broker for the list of topics and messages.

Expected Result: All brokers should have identical lists of topics and messages, indicating consistent data replication and synchronization.

Subscriber Update Mechanism:

Objective: Test that subscribers can update their leader information and continue to receive messages when the leader changes.

Steps:

- Start the Coordinator and three brokers.
- A subscriber subscribes to a topic and starts receiving messages.
- Simulate the leader broker's failure and observe the election of a new leader.
- Ensure the subscriber updates its leader information and continues to receive messages from the new leader.

Expected Result: The subscriber should successfully update its leader information and maintain uninterrupted message reception despite the leader change.

b. Report the Result of Test Cases and Discuss How Well Your System Works

Test Case 1: Broker Registration and Heartbeat Monitoring

- **Result**: Successful. All brokers registered with the Coordinator, and periodic heartbeats were consistently received. The Coordinator accurately maintained the list of active brokers.
- **Discussion**: This test confirms that the system can effectively manage broker registrations and monitor their availability, which is crucial for maintaining system health and stability.

Test Case 2: Leader Election

- **Result**: Successful. When the current leader failed, the Coordinator promptly initiated a leader election, and a new leader was elected. All brokers updated their leader information correctly.
- **Discussion**: The leader election algorithm worked as expected, ensuring that the system remains operational even when the leader broker fails. This fault-tolerant mechanism enhances the system's reliability.

Test Case 3: Publishing and Subscribing to Topics

- **Result**: Successful. The leader broker created the topic, and published messages were correctly propagated to follower brokers. Subscribers successfully received all messages.
- **Discussion**: This test validated the core functionality of the system, demonstrating that publishers and subscribers can interact seamlessly through brokers. The system efficiently handled message dissemination and subscription management.

Test Case 4: Data Consistency Across Brokers

- **Result**: Successful. All brokers maintained consistent data, with identical lists of topics and messages.
- **Discussion**: The data replication and synchronization mechanisms ensured that all brokers had a consistent view of the data. This consistency is vital for providing reliable and accurate information to subscribers.

Test Case 5: Subscriber Update Mechanism

- **Result**: Successful. The subscriber updated its leader information following the leader's failure and continued to receive messages from the new leader.
- **Discussion**: This test demonstrated the system's ability to handle dynamic changes in leadership. Subscribers can adapt to leader changes without service interruption, ensuring continuous access to real-time updates.

10. Conclusions

Lessons Learned:

Building fault-tolerant mechanisms, such as leader election and heartbeat monitoring, is critical in distributed systems. These mechanisms ensure the system remains operational despite individual component failures.

Maintaining data consistency across multiple brokers is challenging but essential. Implementing efficient data replication and synchronization algorithms helps in achieving this consistency, which is crucial for the reliability of the system.

Designing the system to scale horizontally is vital for handling increased load. The publish-subscribe model inherently supports scalability by decoupling publishers and subscribers, which proved effective in our design.

Using standard protocols like RESTful APIs and widely adopted technologies like Spring Boot facilitates interoperability and reduces integration complexity, making the system adaptable to various environments.

Comprehensive testing is crucial to validate the system's functionality and performance. Simulating different scenarios helps in identifying potential issues and ensuring the system meets the required performance standards.

Future Scope:

Enhanced Security Measures: While the system employs HTTPS for secure communication and uses Spring Security for authentication and authorization, additional security measures, such as OAuth2 for more granular access control, could be implemented to further enhance security.

Dynamic Load Balancing: Implementing dynamic load balancing strategies could improve the system's performance under varying loads. This would involve distributing the traffic more evenly across brokers based on their current load.

Scalability Testing at Larger Scale: Conducting large-scale simulations with hundreds or thousands of publishers, subscribers, and brokers would provide deeper insights into the system's scalability and performance under extreme conditions.

User Interface Enhancements: Improving the user interface to provide more intuitive and user-friendly interactions for both publishers and subscribers could enhance the overall user experience. This could include features like real-time notifications and better visualization of sports updates.

Conclusions:

The development of our real-time sports score update system demonstrated the effective implementation of a distributed publish-subscribe architecture. The system successfully addresses key challenges in distributed systems, including heterogeneity, openness, security, failure handling, concurrency, quality of service, scalability, and transparency. Through careful design and implementation, we have built a robust and scalable platform capable of providing timely and accurate sports updates to a wide range of users.

The core components, such as the Coordinator, Brokers, Publishers, and Subscribers, interact seamlessly to ensure data consistency and reliability. The use of logical clocks (Lamport timestamps) and robust leader election algorithms have been pivotal in maintaining system integrity and order of events. The system's modularity and use of RESTful APIs further enhance its adaptability and extensibility, allowing for future enhancements and integration with other systems.

References:

1. Publisher/Subscriber Implementation in Cloud Environment- https://ieeexplore.ieee.org/document/6681311
2. Distributed Brokers in Message Queuing Telemetry Transport: A Comprehensive Review -https://ieeexplore.ieee.org/document/9740815
3. Inter-Service Communication among Microservices using Kafka Connect - https://ieeexplore.ieee.org/document/9930270
4. The research and design of Pub/Sub Communication Based on Subscription Aging - https://ieeexplore.ieee.org/document/8535938

5. The Hidden Pub/Sub of Spotify (Industry Article). https://dl.acm.org/doi/pdf/10.1145/2488222.2488273
6. MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers -https://ieeexplore.ieee.org/abstract/document/9149046