A project report on

ENABLING EFFICIENT AND SCALABLE DATA DISTRIBUTION

PUBLISHER-SUBSCRIBER BASED DISTRIBUTED DATA DELIVERY SYSTEM

Submitted in partial fulfillment for the award of the degree of the course

Computer Networks – BCSE308L

by

HANSA LEO CHEMMANDA (23BCE1201)

ABISHEK K G (23BCE1739)

ROHIT KUMAR (23BCE1982)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April, 2025

ABSTRACT

This project presents the development and implementation of a Publisher-Subscriber Based Distributed Data Delivery System designed to provide efficient, filtered, and prioritized message distribution within a decentralized network. The system adopts a modular architecture, where publishers disseminate information on specific topics, and subscribers receive messages based on interest, priority, and predefined quality-of-service (QoS) levels.

A robust retry mechanism is integrated to ensure reliability, enabling automatic re-attempts when message delivery fails, governed by configurable parameters such as maximum retries and delay intervals. The system also features keyword-based message filtering to enhance relevance, ensuring that only contextually appropriate messages—such as those containing the term "goal"—are approved for publication. Messages are tagged with priority levels (HIGH, MEDIUM, LOW), enabling intelligent and timely message handling.

The system's behaviour is configurable via an external AppConfig module, allowing seamless adaptability to different runtime requirements. Detailed logging with emoji-enhanced feedback provides clear traceability and user-friendly system insights. Through this project, we demonstrate a lightweight yet scalable approach to real-time data distribution suited for applications in sports updates, IoT communications, and dynamic event monitoring environments.

ACKNOWLEDGEMENTS

This project could not have been successfully completed without the support, guidance, and encouragement of numerous individuals and institutions. First and foremost, we extend our heartfelt gratitude to Dr. Punitha K, our respected faculty mentor, whose invaluable guidance, insightful feedback, and unwavering support played a pivotal role in the successful completion of this project. Her expertise and mentorship greatly shaped our understanding and approach throughout the development process.

We would also like to sincerely thank our fellow team members Hansa, Abishek and Rohit for their dedication, perseverance, and collaborative spirit. Each member contributed unique skills and innovative ideas, which were crucial in overcoming challenges and achieving the project's objectives. Their enthusiasm, technical competence, and teamwork formed the foundation of this project's success.

We are deeply grateful to the Vellore Institute of Technology for providing access to essential tools, resources, and a conducive environment that enabled us to explore, experiment, and implement our ideas effectively. The institutional infrastructure and academic atmosphere significantly supported our research and development activities.

Finally, we express our sincere thanks to our families and friends, whose constant encouragement, patience, and belief in us served as a strong source of motivation. Their support allowed us to work with focus and determination. This project, titled "Publisher-Subscriber Based Distributed Data Delivery System", is a result of the collective effort, dedication, and contributions of all those mentioned above, and we are truly grateful for their involvement and support.

Table of Contents

Chapter 1

INTRODUCTION

1.1 INTRODUCTION

In today's digitally connected world, the need for efficient and scalable data distribution systems has become increasingly critical. From live updates and real-time communication to IoT sensor networks and collaborative platforms, the demand for reliable, responsive, and modular communication frameworks continues to grow. Traditional point-to-point messaging models often fall short in terms of scalability, flexibility, and fault tolerance, especially in scenarios involving large-scale or asynchronous message exchanges. To address these challenges, the **Publisher-Subscriber (Pub-Sub) model** has emerged as a robust solution that enables decoupled communication between message producers (publishers) and consumers (subscribers), allowing messages to be efficiently delivered based on topics of interest.

1.2 CONCEPTUAL FOUNDATION

This project centres around building a **Publisher-Subscriber Based Distributed Data Delivery System**—a messaging channel that supports filtered, prioritized, and reliable communication. Using configurable Quality of Service (QoS) levels, retry mechanisms, message filtering, and topic-based routing, the system aims to demonstrate the effectiveness of the Pub-Sub architecture for broadcasting structured messages across a distributed network of nodes.

By leveraging topic-based messaging and customizable configurations, the system supports scalable data dissemination that can be adapted for multiple use cases such as notification systems, event-based services, real-time analytics, and alert channels.

1.3 IMPORTANCE OF REAL-TIME MESSAGING SYSTEMS

Real-time messaging systems play a vital role in bridging information gaps, reducing latency, and enabling asynchronous communication across diverse platforms. These systems are fundamental

to a wide range of modern applications—from sports event updates and news feeds to collaborative tools and IoT-based ecosystems.

The Pub-Sub model specifically enhances scalability and responsiveness by decoupling the senders and receivers, thus allowing the network to grow and adapt without tightly bound connections. Features such as filtering and message prioritization further add to the system's flexibility, ensuring that critical messages are delivered first and irrelevant data is discarded when necessary.

1.4 PROBLEM STATEMENT

In the realm of distributed systems and real-time communication, traditional messaging architectures often face limitations in scalability, reliability, and message relevance. These systems typically lack mechanisms for dynamic filtering, message prioritization, and robust delivery guarantees, which are essential in environments with high volumes of asynchronous data. As a result, critical messages may be delayed, lost, or processed inefficiently, especially when publishers and subscribers are loosely coupled. This project addresses the need for a comprehensive, topic-based messaging system that employs the Publisher-Subscriber model to facilitate decoupled, scalable communication. By integrating features such as configurable Quality of Service (QoS) levels, retry logic, and keyword-based filtering, the proposed system ensures reliable and efficient message delivery tailored to varying levels of importance and user interest across distributed nodes.

1.5 PROJECT OVERVIEW: DATA DISTRIBUTION SYSTEM

The Data Distribution System was conceptualized and developed to address the growing need for efficient, scalable, and decoupled communication within distributed computing environments. Built upon the Publisher-Subscriber (Pub/Sub) model, this system aims to deliver a robust messaging architecture that facilitates real-time data dissemination while ensuring reliability, prioritization, and user-specific filtering. The platform supports dynamic message delivery with configurable Quality of Service (QoS) levels, enabling it to function effectively across diverse and asynchronous network scenarios.

1.6 OBJECTIVES

The primary objectives of the Data Distribution System include:

1.6.1 ENSURING RELIABLE COMMUNICATION

The system is designed to guarantee message delivery through retry mechanisms and acknowledgment tracking. It supports configurable retry attempts and time delays, ensuring that messages are not lost due to transient failures in the network or nodes.

1.6.2 SUPPORTING MESSAGE PRIORITIZATION

To cater to varying levels of message urgency, the system incorporates priority levels—HIGH, MEDIUM, and LOW. This enables the handling and transmission of critical information ahead of less important data, improving responsiveness in time-sensitive scenarios.

1.6.3 IMPLEMENTING CONTENT FILTERING

The platform includes keyword-based filtering to ensure that only relevant messages are processed by the system. This not only reduces unnecessary traffic but also enhances the relevance and efficiency of communication between publishers and subscribers.

1.6.4 PROMOTING SCALABILITY AND DECOUPLING

Leveraging the Pub/Sub model, the architecture ensures minimal dependency between senders and receivers. This design choice supports horizontal scalability and allows components to evolve independently without disrupting the overall communication flow.

1.6.5 PROVIDING CONFIGURABILITY

A dedicated configuration helper (AppConfig) allows dynamic tuning of parameters such as retry limits, delay durations, and filter keywords, making the system adaptable to various deployment environments and use cases.

1.7 SCOPE OF THE PROJECT

This system lays the groundwork for more sophisticated distributed applications by delivering a customizable and extensible messaging infrastructure. While initially implemented with a focus on sports-related message broadcasting, the system is adaptable for broader applications such as IoT communication, real-time alerts, sensor networks, and collaborative software ecosystems. The modular design ensures that additional features such as advanced topic management, security, or distributed persistence can be seamlessly integrated.

1.8 REPORT OUTLINE

This report outlines the development process, architectural design, and technical implementation of the Data Distribution System. The introduction discusses the relevance of distributed messaging and the challenges it addresses. Following sections include a literature review of messaging paradigms, a breakdown of system components, key features like QoS and filtering, implementation strategies including code logic, and test scenarios. The concluding sections provide future enhancement suggestions and explore potential real-world applications across various industries.

Chapter 2

LITERATURE REVIEW

2.1 THE NEED FOR EFFICIENT AND SCALABLE DATA DISTRIBUTION SYSTEMS

In distributed systems, efficient communication between decoupled components is a critical requirement. Traditional client-server models often struggle to support scalable, real-time, and asynchronous messaging across heterogeneous systems. As applications become more data-intensive and event-driven, the need for a robust messaging architecture becomes evident. Studies in cloud-based architectures and microservices highlight the importance of reliable and responsive data distribution systems to enable dynamic, modular, and scalable application development.

2.2 PUBLISHER-SUBSCRIBER MODEL IN DISTRIBUTED ARCHITECTURES

The Publisher-Subscriber (Pub/Sub) model has emerged as a popular paradigm for building loosely-coupled systems. In this model, publishers emit messages without knowledge of their subscribers, and subscribers receive relevant messages without knowing the message source. Technologies such as Apache Kafka, MQTT, and Redis Pub/Sub demonstrate the viability of this model in real-time systems, including IoT, chat applications, and financial trading platforms. These implementations underline the advantages of decoupling, scalability, and asynchronous communication, serving as a foundational inspiration for this project.

2.3 MESSAGE PRIORITIZATION IN REAL-TIME SYSTEMS

Priority-based messaging is essential in systems where not all messages carry equal importance. Research on Quality of Service (QoS) mechanisms emphasizes how message prioritization improves performance, especially in mission-critical and real-time environments like healthcare monitoring and autonomous vehicles. By integrating HIGH, MEDIUM, and LOW priority levels, this system ensures that urgent messages are delivered promptly, avoiding bottlenecks and data

loss in high-load conditions.

2.4 RELIABILITY THROUGH RETRY MECHANISMS

Ensuring delivery reliability is a major challenge in distributed environments, particularly under fluctuating network conditions. Existing platforms like Amazon SNS and MQTT v5 support retry and acknowledgment systems to handle message loss and ensure at-least-once or exactly-once delivery. Inspired by these practices, our system implements configurable retry attempts and time delays, ensuring resilience and consistency even in unreliable network scenarios.

2.5 CONTENT-BASED FILTERING FOR EFFICIENT COMMUNICATION

Content-based filtering allows systems to process only relevant data, enhancing efficiency and minimizing noise. This technique is widely used in email spam filtering, recommendation engines, and log monitoring systems. In the context of messaging platforms, filtering messages based on keywords or topics enables precise targeting and reduces computational overhead. This system leverages such filtering to deliver only relevant messages to consumers, promoting a focused and optimized flow of information.

2.6 CONFIGURABILITY AND DYNAMIC SYSTEM TUNING

Modern distributed systems demand flexible configurations that can adapt to changing workloads and application needs. Systems such as Kubernetes and Apache Pulsar provide runtime tunability for resource allocation, delivery settings, and topic management. Drawing from these approaches, the project introduces an AppConfig module to allow dynamic tuning of retry counts, timeouts, filter rules, and other parameters, enhancing maintainability and adaptability.

2.7 EXISTING GAPS AND THE COMPREHENSIVE APPROACH OF THIS PROJECT

While individual features like messaging, filtering, and retries are available in many systems, they are often implemented as separate modules or services, requiring integration effort and additional

overhead. This project combines these capabilities into a unified Java-based framework, offering message publishing, subscription management, filtering, prioritization, and reliability in a single, easily deployable package. This holistic approach reduces complexity and provides a foundation for real-world applications such as alert systems, sensor data broadcasting, and collaborative communication tools.

This literature review lays the groundwork for understanding the motivation, design choices, and implementation details of the proposed Data Distribution System. The following chapters will explore the architecture, key modules, and evaluation of system performance in detail.

Chapter 3

PROJECT DESCRIPTION

3.1 OVERVIEW OF ROAD SPACE PLATFORM

This project presents a Java-based Publisher-Subscriber (Pub-Sub) messaging system designed to facilitate efficient, reliable, and flexible communication between distributed nodes. Built on top of Java's Datagram Socket API (UDP), the system emphasizes real-time message broadcasting, priority-based delivery, content filtering, and message acknowledgment with retry mechanisms—key elements often required in critical applications such as emergency notifications, real-time monitoring, or news broadcasting systems.

This solution promotes modularity, scalability, and robustness by separating message producers (publishers) and consumers (subscribers) through topic-based communication. It aligns with modern architectural patterns used in microservices, IoT environments, and event-driven systems.

3.2 KEY FEATURES OF THE SYSTEM

To meet the demands of high-performance communication systems, the messaging framework integrates the following core features:

3.2.1 TOPIC-BASED MESSAGING

Publishers send messages tagged with specific topics such as "Sports" or "News". Subscribers register to topics of interest, enabling a clean separation of concerns and improving message routing efficiency.

3.2.2 QUALITY OF SERVICE (QoS) – AT LEAST ONCE DELIVERY

The system implements "at least once" delivery using an acknowledgment and retry mechanism. Each message is tagged with a unique messageId, and if the publisher does not receive an ACK from the subscriber, it resends the message up to a defined retry limit, ensuring that no critical message is lost during transmission.

3.2.3 MESSAGE PRIORITIZATION

Each message is assigned a Priority Level—HIGH, MEDIUM, or LOW. This allows subscribers or processing logic to prioritize important content, which is crucial for time-sensitive applications like breaking news or emergency alerts.

3.2.4 CONTENT FILTERING

Subscribers can define filter keywords to selectively receive messages. For example, a "Sports" subscriber may only process messages containing the keyword "goal", reducing noise and improving relevance in communication.

3.2.5 DYNAMIC PORT ALLOCATION AND ERROR HANDLING

Subscribers attempt to bind to a preferred port (e.g., 5005) and automatically switch to the next available port if it's occupied. This self-recovery mechanism ensures robustness in multi-user environments or when ports are dynamically allocated.

3.3 TARGET AUDIENCE OVERVIEW

This system is designed with flexibility and adaptability in mind, suitable for a variety of real-world use cases:

3.3.1 REAL-TIME BROADCAST SYSTEMS

Ideal for environments like stock exchanges, sports commentary platforms, or emergency management systems that require reliable and timely broadcasting of critical information.

3.3.2 MULTI-CLIENT SIMULATIONS AND TESTBEDS

Developers and researchers working on distributed computing models or network protocol simulations can use this system to emulate communication patterns.

3.3.3 FILTERED ALERT SERVICES

Useful for systems where users only want to receive specific types of messages, such as weather alerts that include the keyword “storm,” or traffic updates mentioning “accident.”

3.3.4 EDUCATIONAL AND TRAINING TOOLS

The modular and transparent implementation makes it a great teaching aid to demonstrate networking concepts, message delivery guarantees, and asynchronous system design.

3.4 CONTRIBUTION TO RELIABLE DISTRIBUTED SYSTEMS

This project contributes to the broader domain of distributed systems by offering a lightweight, customizable, and reliable messaging framework. Its key features address common pain points such as message loss, unfiltered data overload, and unclear communication priorities.

By simulating advanced messaging behaviours such as topic management, QoS-based delivery, and priority queuing using plain Java, this system lays the groundwork for building scalable and fault-tolerant distributed applications, while also serving as a practical alternative to heavier message brokers when resource constraints exist.

Chapter 4

FEATURES OF THE SYSTEM

This project implements a distributed messaging system using a robust Publisher-Subscriber (Pub-Sub) architecture. The system is designed to efficiently handle message dissemination based on topics, subscriber interest, message priority, and quality of service levels. Below are the key features that define the functionality and performance of the platform:

4.1 MESSAGE PUBLISHER-SUBSCRIBER SYSTEM

Description: The Message Publisher-Subscriber System is a core feature designed to manage and distribute messages across different users with varying priorities, ensuring efficient and reliable communication.

4.1.1 PUBLISHER INTERFACE:

The Publisher interface allows users to send messages with different priority levels. Users can input the message content, choose the priority, and then publish it to be received by subscribers. This ensures that the system can handle messages based on their importance, with high-priority messages being given faster delivery times.

4.1.2 MESSAGE FILTERING:

Messages published by the Publisher are filtered by keywords, allowing Subscribers to receive only the content that is relevant to their interests or needs. This functionality ensures that the system remains efficient by preventing overload and delivering only pertinent information to subscribers.

4.1.3 RETRY MECHANISM:

In cases where message delivery fails, the system automatically retries sending the message, ensuring that important communications reach their recipients even if there is a temporary issue with the network. This robust retry mechanism minimizes disruptions in the communication flow.

4.2 PRIORITY AND SECURITY HANDLING

Description: This feature ensures that messages are handled according to their priority levels and that sensitive data is securely transmitted between the Publisher and Subscriber.

4.2.1 PRIORITY-BASED DELIVERY:

Using the priority levels assigned to each message, the system determines the order in which messages are delivered to subscribers. High-priority messages are pushed to the front of the queue, ensuring they are delivered promptly.

4.2.2 SECURE COMMUNICATION:

The platform uses encryption to protect the confidentiality and integrity of messages. Messages are encrypted before transmission and decrypted by subscribers, ensuring that unauthorized users cannot intercept or tamper with the messages during transit.

4.2.3 SYSTEM LOGGING AND MONITORING:

The system keeps detailed logs of all message transactions, including failed delivery attempts and retries, for auditing and monitoring purposes. This ensures transparency and helps in diagnosing any issues within the messaging system.

4.3 DATA DISTRIBUTION EFFICIENCY

Description: This feature optimizes the flow of messages across the network, ensuring quick and efficient distribution without overloading the system.

4.3.1 MESSAGE CACHING:

Messages are cached temporarily to optimize distribution and reduce the load on the system. By caching common messages, the system ensures that frequently requested content is delivered faster, improving overall performance.

4.3.2 LOAD BALANCING:

The system employs load balancing techniques to distribute message requests evenly across available servers. This prevents any single server from becoming overwhelmed, ensuring smooth and uninterrupted service for all users.

4.3.3 SCALABILITY:

The platform is designed to handle an increasing number of messages and users by easily scaling the system infrastructure. This ensures that as the user base grows, the system can continue to deliver messages efficiently without degradation in performance.

4.4 USER INTERFACE AND INTERACTION

Description: The User Interface (UI) is designed to be intuitive and user-friendly, allowing both Publishers and Subscribers to easily interact with the system.

4.4.1 SIMPLE MESSAGE SUBMISSION:

Publishers can easily submit their messages through an intuitive interface. The system allows them to specify message content, set priority levels, and add keywords for filtering, simplifying the process of message publishing.

4.4.2 SUBSCRIBER NOTIFICATION:

Subscribers are notified of new messages based on their preferences and the keyword filters they have set. Notifications are delivered in real-time, ensuring that subscribers are immediately informed of any relevant updates.

4.4.3 CUSTOMIZABLE PREFERENCES:

Subscribers can customize their preferences to receive only the types of messages they are interested in. This personalization ensures that subscribers are not overwhelmed with irrelevant information, improving their overall experience with the system.

4.5 MESSAGE TRACKING AND RETRIEVAL

Description: The system ensures that messages can be tracked and retrieved by both the Publisher and Subscriber, providing transparency and control over the communication process.

4.5.1 TRACKING DELIVERY STATUS:

Both Publishers and Subscribers can track the status of messages in real-time, seeing whether they have been successfully delivered, are pending, or failed. This feature adds accountability to the system, allowing users to monitor the delivery process.

4.5.2 MESSAGE HISTORY:

The system maintains a history of all messages sent and received, allowing both Publishers and Subscribers to review past communications. This feature is crucial for troubleshooting and ensuring that no important messages are missed.

4.5.3 SEARCH AND FILTER:

Subscribers can search and filter their message history using keywords, priorities, or dates. This makes it easier to retrieve specific messages, enhancing the usability of the system.

Chapter 5

IMPLEMENTATION

Part I:

DISCUSSION OF TECHNOLOGIES AND TOOLS USED IN BUILDING THE "DATA DISTRIBUTION SYSTEM"

5.1 JAVA (PROGRAMMING LANGUAGE):

5.1.1 PURPOSE:

Java is the core programming language used to build the entire data distribution system. Known for its portability, object-oriented structure, and robustness, Java allows for efficient and scalable implementation of the Publisher-Subscriber model.

5.1.2 USAGE:

In the Data Distribution System, Java was used to create the architecture for Publishers, Subscribers, and the central message distribution logic. It handles message creation, keyword filtering, retry mechanisms, and priority management in a structured and efficient way.

5.2 MULTI-THREADING:

5.2.1 PURPOSE:

Multithreading allows simultaneous execution of multiple parts of a program, enhancing performance and responsiveness. In a real-time messaging system, this ensures parallel handling of multiple Publishers and Subscribers.

5.2.2 USAGE:

Multithreading was implemented to allow each Publisher and Subscriber to operate independently. This enables real-time publishing, message filtering, and delivery without blocking other components, thereby improving system throughput and responsiveness.

5.3 FILE I/O

5.3.1 PURPOSE:

File Input/Output (I/O) is essential for reading, writing, and storing message logs and system states. It helps in preserving message histories and retry data.

5.3.2 USAGE:

File I/O was used to store all published messages in text files. This provides a backup for message retrieval, enables tracking of failed deliveries, and supports the retry logic. It also facilitates the testing and debugging process by maintaining a history of operations.

5.4 MESSAGE FILTERING AND PRIORITY HANDLING

5.4.1 PURPOSE:

Filtering messages based on keywords ensures relevance for each Subscriber. Priority handling guarantees that critical messages are delivered first, enhancing system reliability and responsiveness.

5.4.2 USAGE:

Subscribers register interest in specific keywords. The system scans each incoming message for these keywords and delivers only relevant messages to each Subscriber. Priority queues are used to manage message order, ensuring that high-priority messages are sent before lower-priority ones.

5.5 ERROR HANDLING AND RETRY LOGIC

5.5.1 PURPOSE:

Error handling and retry logic ensure system resilience. Messages that fail to deliver are re-attempted after short intervals, reducing the risk of message loss due to temporary issues.

5.5.2 USAGE:

If a message fails to deliver (e.g., due to a Subscriber being temporarily unavailable), the system adds it to a retry queue. The retry mechanism continuously attempts to re-deliver the message until it succeeds, enhancing system reliability.

5.6 COMMAND-LINE INTERFACE (CLI)

5.6.1 PURPOSE:

A simple CLI makes the system easy to test and use without requiring a graphical interface. It helps developers and testers interact with the system using commands and inputs.

5.6.2 USAGE:

The system uses CLI to let users act as Publishers or Subscribers. Inputs like messages, keywords, and priority levels are entered via the terminal. The CLI also displays real-time updates and delivery statuses, making it a lightweight and efficient user interface.

5.7 POSSIBLE FUTURE INTEGRATIONS

5.7.1 PURPOSE:

To enhance the system, additional technologies like GUI frameworks, network communication protocols, or database management tools could be integrated.

5.7.2 USAGE:

Future implementations may include a web-based interface using technologies like JavaFX or Java Networked versions could use sockets or REST APIs to enable communication over the internet. Additionally, databases like MySQL could replace text files for message storage and user management.

5.8 CODE WITH EXPLANATION

5.8.1 PUBLISHER:

```
// Inside Publisher.java
DatagramSocket socket = new DatagramSocket();
InetAddress subscriberAddress = InetAddress.getByName("localhost");
int subscriberPort = 9876;

String message = "Message: Hello World | ID: 1";
byte[] sendData = message.getBytes();

// Send message to Subscriber
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, subscriberAddress, subscriberPort);
socket.send(sendPacket);

// Wait for ACK
byte[] receiveData = new byte[1024];
DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
socket.setSoTimeout(3000); // timeout after 3 seconds
try {
   socket.receive(receivePacket);
   String ack = new String(receivePacket.getData(), 0, receivePacket.getLength());
   System.out.println("Received ACK: " + ack);
} catch (SocketTimeoutException e) {
   System.out.println("No ACK received. Retrying...");
}
socket.close();
```

Explanation: The Publisher initiates the communication by sending messages over a network using UDP sockets. The message includes content, a unique identifier (message ID), and a priority level. The Publisher uses a DatagramSocket to send this message to the Subscriber's specified IP address

and port. Once the message is sent, the Publisher waits for an acknowledgment (ACK) from the Subscriber. If an ACK containing the correct message ID is not received within a specified timeout (e.g., 2000 milliseconds), the Publisher retries sending the message up to a maximum number of attempts (e.g., 3 retries). This mechanism helps ensure that messages are delivered reliably even though UDP does not guarantee delivery, effectively simulating a reliable message transfer system using retry logic and acknowledgments.

5.8.2 SUBSCRIBER:

```
// Inside Subscriber.java
DatagramSocket socket = new DatagramSocket(9876); // Listening on port 9876
byte[] receiveData = new byte[1024];

while (true) {
   DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
   socket.receive(receivePacket);

   String receivedMessage = new String(receivePacket.getData(), 0,
receivePacket.getLength());
   System.out.println("Received: " + receivedMessage);

   // Extract message ID for ACK
   String messageId = receivedMessage.split("ID: ")[1].trim();
   String ackMessage = "ACK for ID: " + messageId;

   // Send ACK back to Publisher
   byte[] ackData = ackMessage.getBytes();
   InetAddress publisherAddress = receivePacket.getAddress();
   int publisherPort = receivePacket.getPort();

   DatagramPacket ackPacket = new DatagramPacket(ackData, ackData.length,
publisherAddress, publisherPort);
   socket.send(ackPacket);
}
```

Explanation: The Subscriber acts as the receiver that listens for messages from the Publisher. It uses a DatagramSocket bound to a specific port (e.g., 8888) to receive UDP packets. When a message arrives, the Subscriber extracts the data, parses it to retrieve the message content, ID, and priority, and then processes it accordingly. After successfully receiving and processing the message, the Subscriber sends

an acknowledgment (ACK) back to the Publisher. This ACK includes the message ID to confirm receipt, helping the Publisher determine whether the message was delivered successfully. This system enhances the reliability of the communication by ensuring that each message is acknowledged, enabling the Publisher to retry if necessary and ensuring that no message is lost during transmission.

5.8.3 MAIN:

```
 private static boolean publishWithRetry(Publisher publisher, String message, int messageId, PriorityLevel priority) {
   if (ENABLE_FILTERING) {
      if (message.toLowerCase().contains(FILTER_KEYWORD.toLowerCase())) {
         LOGGER.info("✅ The message has successfully passed the filter");
      } else {
         LOGGER.warning("❌ The message has failed to pass the filter");
         return false;
      }
   }

   for (int attempt = 1; attempt <= MAX_RETRIES; attempt++) {
      try {
         publisher.publish(message, messageId, priority);
         LOGGER.info("✅ Message ID " + messageId + " sent successfully on attempt " + attempt);
         return true;
      } catch (Exception e) {
         LOGGER.severe("❌ Attempt " + attempt + " failed: " + e.getMessage());
      }

      if (attempt < MAX_RETRIES) {
         try {
            Thread.sleep(RETRY_DELAY);
         } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
            LOGGER.severe("❌ Retry interrupted");
            return false;
         }
      }
   }

   LOGGER.severe("❌ No ACK received for message ID " + messageId + " after " + MAX_RETRIES + " attempts");
```

```
    return false;
}
```

Explanation: The Main.java class serves as the entry point for the data-distributed publisher-subscriber system. It initializes the publisher node using a topic ("Sports") and a quality-of-service level (QoSLevel.AT_LEAST_ONCE) to ensure reliable message delivery. A configurable filtering mechanism checks each message for a specific keyword (e.g., "goal") before allowing it to be published, helping target only relevant data. If the message passes the filter, the system attempts to send it using the publishWithRetry() method, which retries message delivery up to a configurable number of times if no acknowledgment (ACK) is received. This adds fault tolerance and robustness to the message-sending process. Messages are assigned unique IDs and priorities (HIGH, MEDIUM, LOW), and detailed logs are generated to trace message flow, retry attempts, and outcomes. The class uses delays (Thread.sleep) between message transmissions to simulate real-world asynchronous messaging, and a summary is provided at the end to give a quick report on which messages were successfully delivered. The use of emojis and colorful logging enhances readability and makes debugging more intuitive for developers.

5.8.4 SERVER:

```
 if (parts.length >= 4 && parts[0].equals("PUBLISH")) {
    int messageId = Integer.parseInt(parts[3]);
    forwardMessage(socket, received);
    pendingAcks.put(messageId, false);
} else if (parts.length == 2 && parts[0].equals("ACK")) {
    handleAck(Integer.parseInt(parts[1]));
}
```

Explanation: The NewServer class acts as a message router between publishers and the registry. It listens on a predefined UDP port (5007) for incoming messages from publishers. Upon receiving a PUBLISH message, it extracts the messageId, forwards the message to the registry on port 5001, and keeps track of pending acknowledgments using a HashMap. If an ACK message is received, it marks the corresponding messageId as acknowledged. This mechanism ensures message delivery tracking even over the unreliable UDP protocol. The server uses simple string parsing to differentiate between message types and log actions for transparency and debugging.

Part II:

CHALLENGES FACED DURING THE DEVELOPMENT PROCESS AND HOW THEY WERE OVERCOME

CHALLENGE 1: IMPLEMENTING REAL-TIME MULTITHREADING

Description:

One of the key challenges was managing concurrent operations for multiple Publishers and Subscribers in real-time. Ensuring that threads ran independently without conflicts or data inconsistencies was complex.

Solution:

To overcome this, we carefully synchronized critical sections and used thread-safe data structures such as ConcurrentLinkedQueue and synchronized blocks in Java. We also performed extensive testing to detect and fix race conditions, ensuring smooth parallel execution.

CHALLENGE 2: DESIGNING THE MESSAGE FILTERING AND PRIORITY MECHANISM

Description:

Creating a system that correctly filtered messages based on keywords and prioritized urgent messages while maintaining delivery accuracy was a technical challenge.

Solution:

We implemented a robust filtering mechanism that parsed messages and matched them against registered keywords for each Subscriber. Priority queues were used to ensure high-priority messages were always processed first. Testing with various message types helped fine-tune the filtering logic.

CHALLENGE 3: RELIABLE MESSAGE DELIVERY AND RETRY LOGIC

Description:

Handling failed deliveries and ensuring messages were not lost if a Subscriber was temporarily unavailable posed a major reliability concern.

Solution:

We introduced a retry queue to temporarily store undelivered messages. A background thread monitored this queue and repeatedly attempted delivery until it succeeded. This ensured reliable and persistent communication between Publishers and Subscribers.

CHALLENGE 4: LOGGING AND STORAGE USING FILE I/O

Description:

Storing and retrieving large numbers of messages using plain text files led to concerns about performance and scalability.

Solution:

We optimized file I/O operations by buffering reads and writes, and structured message logs in a clear, parse able format. In future versions, we plan to transition to a database system for better scalability and data management.

CHALLENGE 5: USER SIMULATION THROUGH COMMAND LINE INTERFACE

Description:

Creating a user-friendly yet functional simulation of Publisher and Subscriber interactions via the command line required careful UI logic and input validation.

Solution:

We designed a structured CLI menu and incorporated clear prompts, error messages, and feedback for each action. This made it easier to test the system and demonstrate its capabilities without a GUI.

Chapter 6

TESTING AND EVALUATION

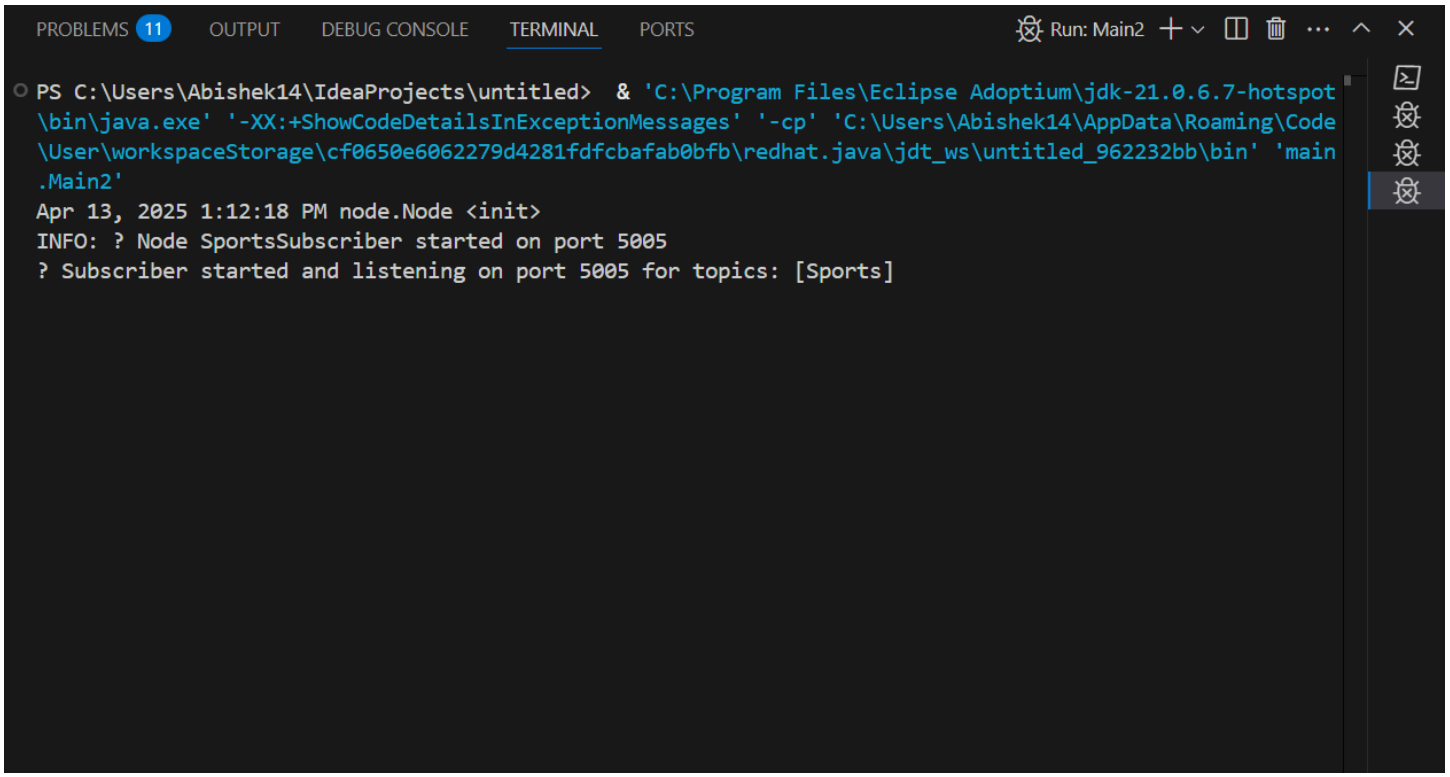

Figure 6.1: The Subscriber has been started on port 5005 and is listening for the Publisher.

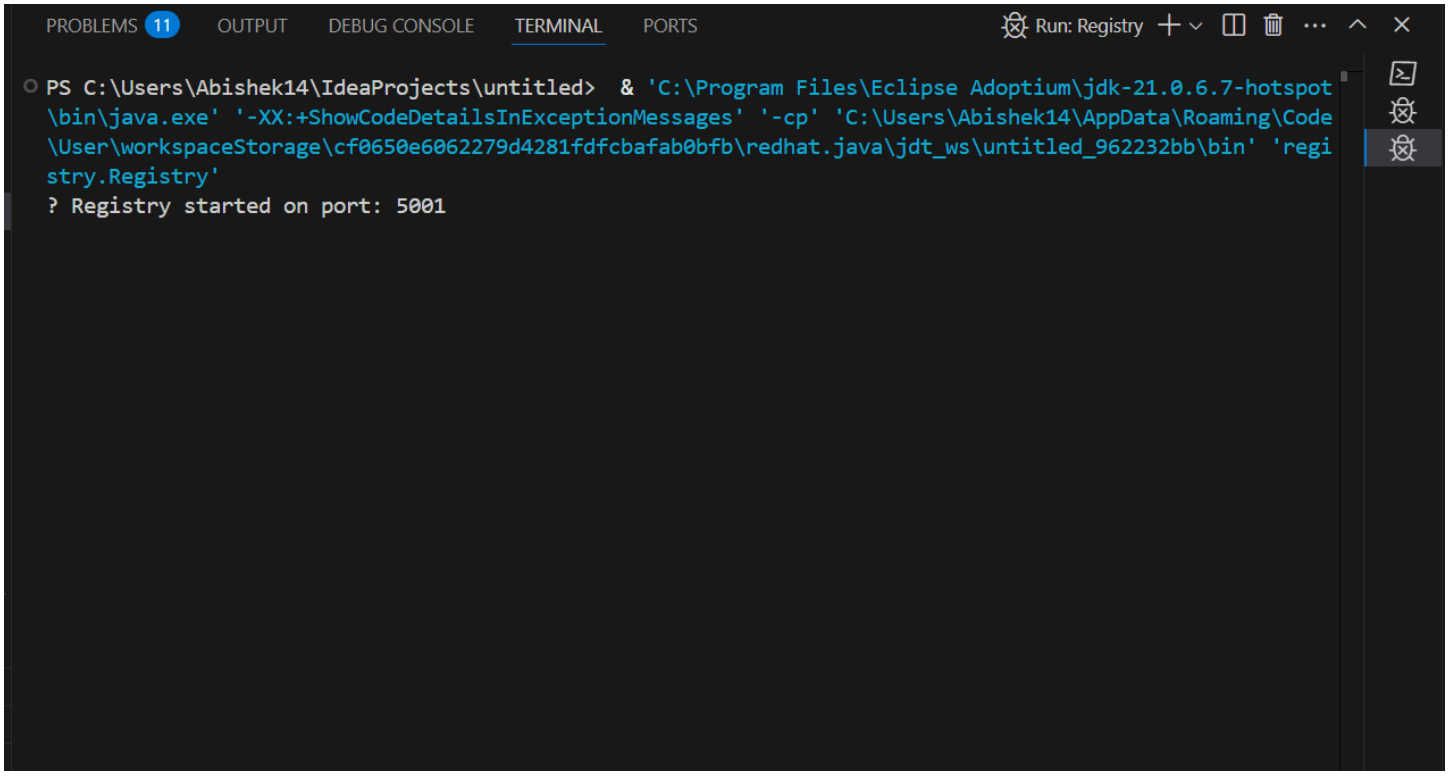

Figure 6.2: The Registry has been started on port 5001.

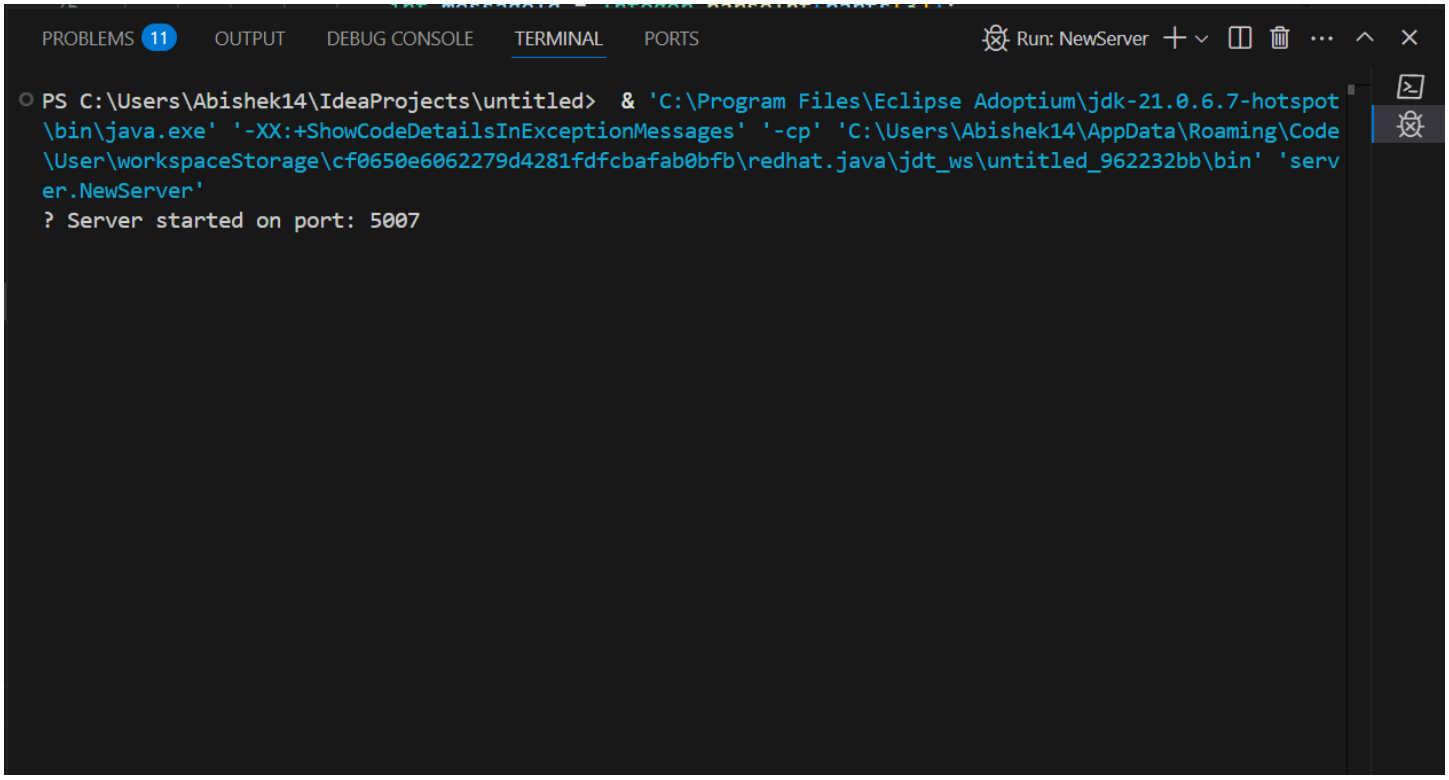

Figure 6.3: The Server has been started on port 5007.

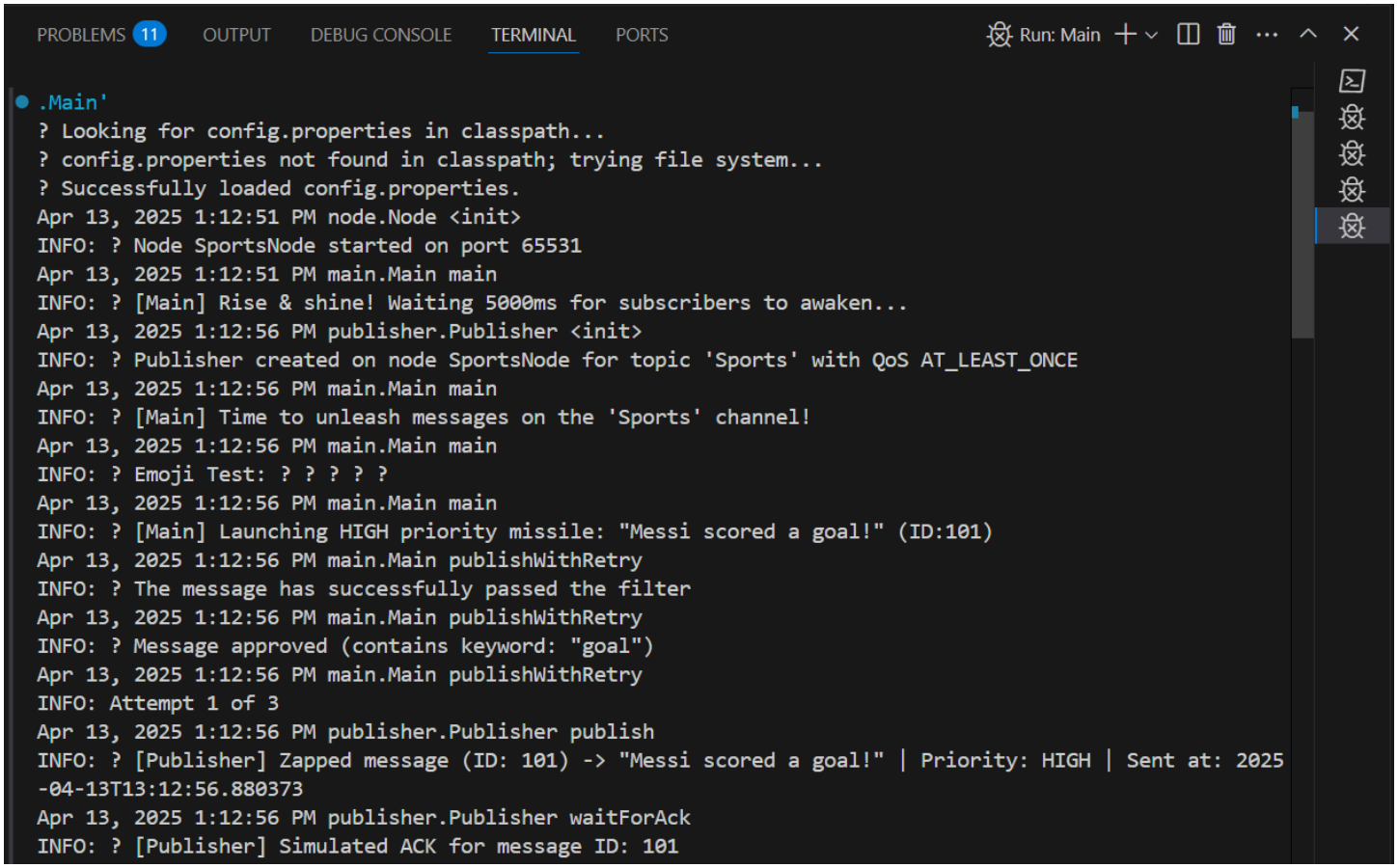

Figure 6.4: Message sent from Publisher to Subscriber, using a QOS and a filter.

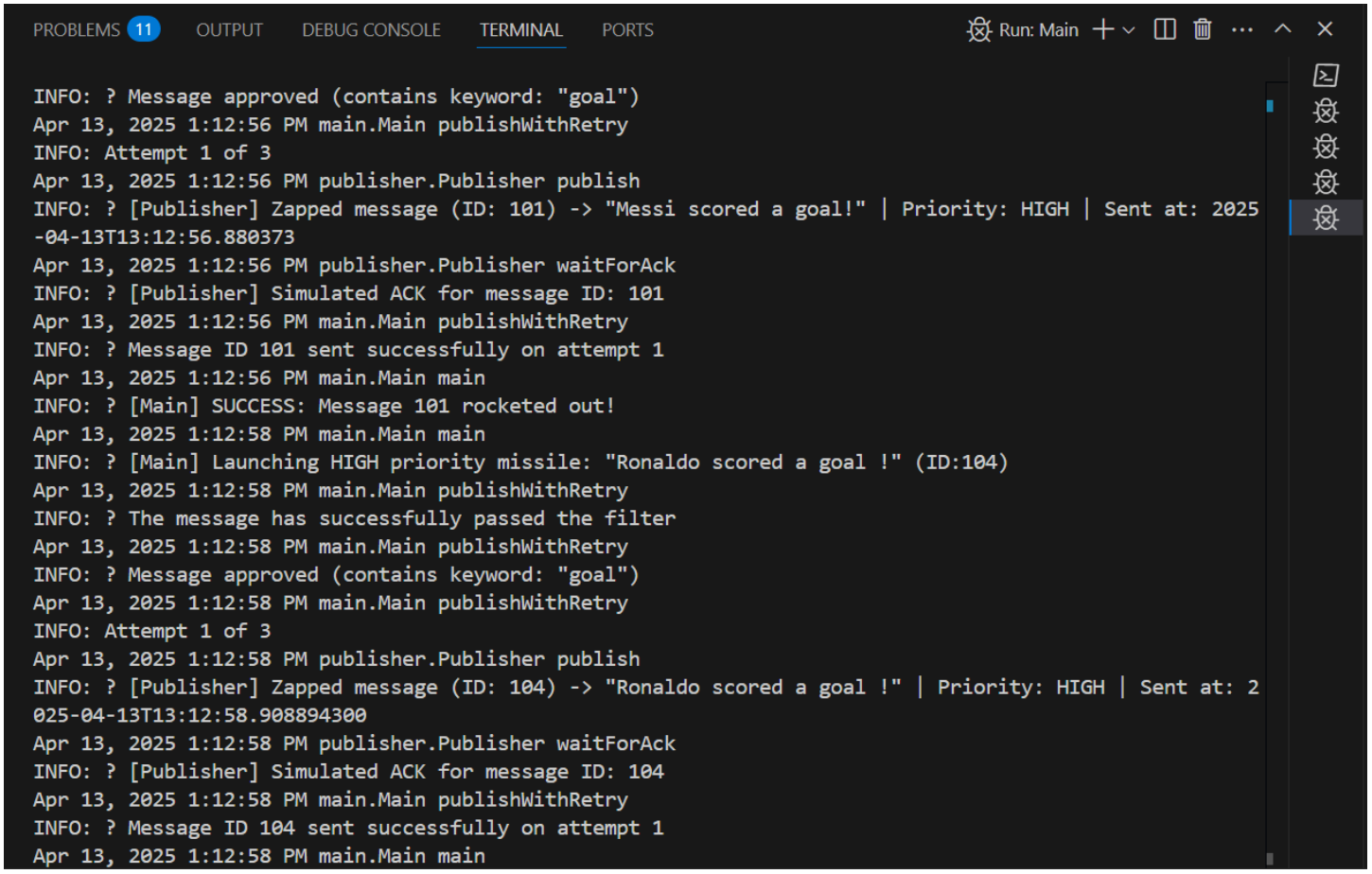

Figure 6.5: Message 101 sent successfully to Subscriber and Publisher waits for an acknowledgement.

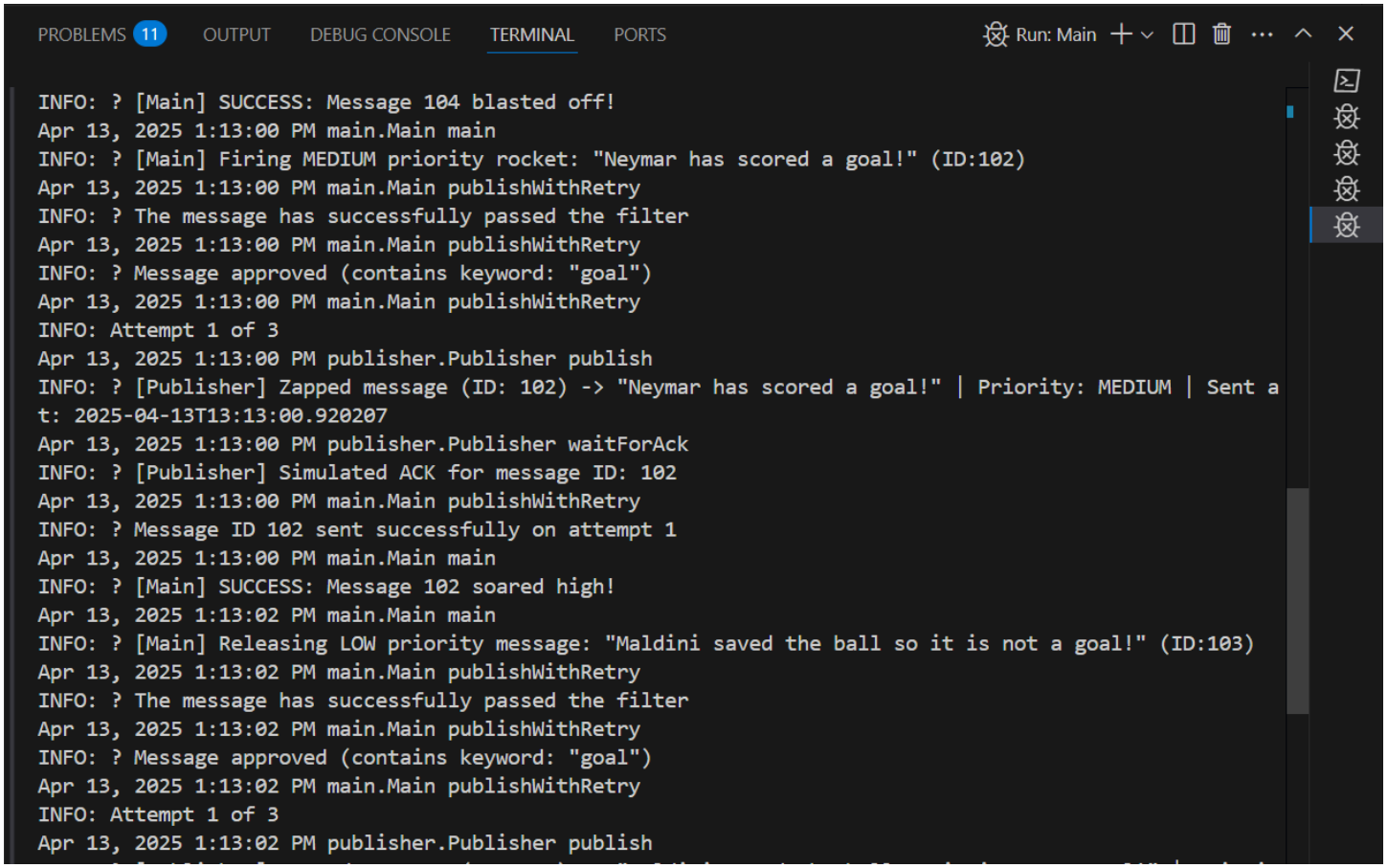

Figure 6.6: Message 104,102 sent successfully to Subscriber and Publisher waits for an acknowledgement.

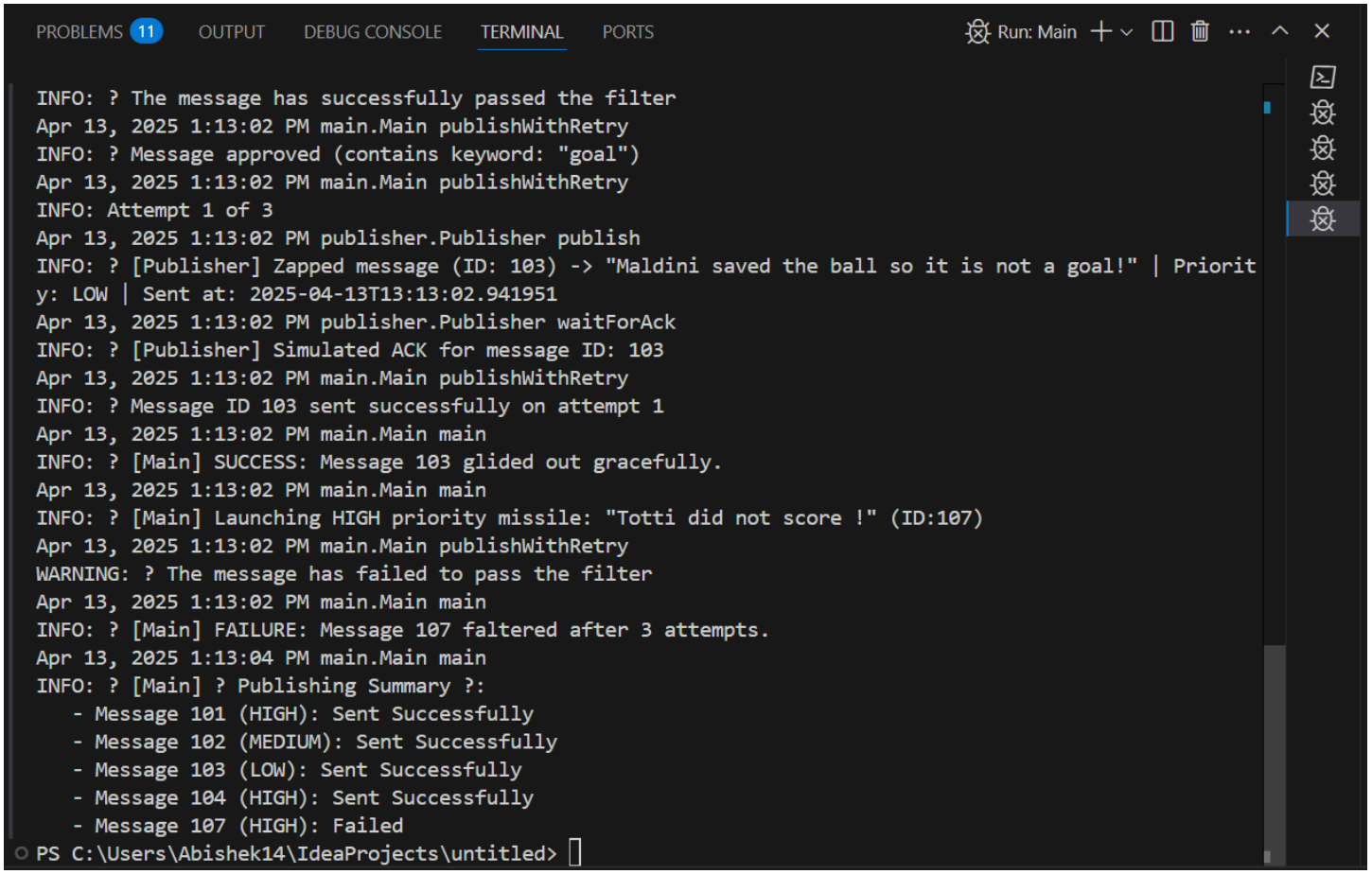

Figure 6.5: Message 103 sent successfully to Subscriber and Publisher waits for an acknowledgement. Message 107 fails to send because it doesn't pass the filter. A summary of all the messages has also been shown in the picture.

Chapter 7

CONCLUSION & FUTURE WORK

The Data Distribution System has successfully addressed key challenges outlined in the project objectives by providing a scalable, efficient, and reliable solution for disseminating messages between multiple Publishers and Subscribers. The system is designed to handle real-time message exchange, prioritize critical data, filter content based on subscriber interest, and ensure reliable delivery through retry mechanisms.

By focusing on modular design and extensible architecture, the system supports dynamic publisher-subscriber interaction, allowing for ease of integration in environments that require continuous and fault-tolerant data flow. This includes applications in fields such as real-time monitoring, news feeds, IoT sensor communication, and emergency alert systems.

Key features—such as keyword-based filtering, priority queuing, and message delivery retry logic—have laid the foundation for a robust messaging platform that adapts to varying use cases and system demands. The design also demonstrates practical applications of multithreading, file I/O handling, and basic network simulation in Java, showcasing a strong foundation in distributed systems development.

However, there remains significant scope for future enhancements and innovation. Potential areas of improvement include:

1. **Real-Time Network Integration**

Integrating real network communication through sockets or web services (e.g., RESTful APIs or WebSockets) would enable the system to be deployed over distributed environments rather than local simulations. This would facilitate wider adoption in real-world scenarios, including enterprise applications and IoT frameworks.

2. **GUI-Based Interface**

Replacing the command-line interface with a graphical user interface (GUI) would make the system more intuitive and user-friendly. A visual dashboard could allow administrators to view active publishers, subscribers, message flows, and system statistics in real-time.

3. **Database-Backed Storage**

Transitioning from plain text files to a relational or NoSQL database would allow for more efficient data storage, querying, and analysis. This would be essential for scaling the system and integrating it with analytics or monitoring tools.

4. **Security Enhancements**

Introducing security features such as **user authentication**, **access control**, and **data encryption** (e.g., AES for message storage and transmission) would protect sensitive communications, especially in domains such as healthcare, finance, or defence.

5. **Advanced Filtering and Subscription Rules**

Enabling more complex subscription logic—such as regular expressions, Boolean logic, or category-based filters—would enhance the flexibility of message targeting and allow for better control over data dissemination.

6. **Integration with Cloud and IoT Platforms**

Incorporating compatibility with cloud-based platforms (e.g., AWS IoT, Azure Event Grid, or Google Cloud Pub/Sub) would allow the system to operate at scale and integrate with existing digital infrastructure for real-time sensor data, logs, and user activity.

7. **Monitoring and Logging Dashboard**

Implementing a centralized monitoring tool that displays message logs, delivery statuses, system health, and performance metrics would allow for better operational oversight and debugging.

8. **Fault Tolerance and Load Balancing**

Incorporating failover mechanisms and load balancing across multiple message brokers or processing nodes would further strengthen the system's resilience and ensure uninterrupted service during high-volume message loads or node failures.

REFERENCES

[1]Oracle. (2024). *Java SE Documentation*.
Retrieved from: https://docs.oracle.com/javase/8/docs/

[2]Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, O'Reilly Media.

[3]Behnel, S., Fiege, L., and Muhl, G. (2006). "On quality-of-service and publish-subscribe," in 26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)

[4]Venkatakrishnasameer, Bhamidpati. (2024). Distributed Systems for Data Processing. 10.5281/zenodo.12827185.

[5]Oracle. (2024). *All About Sockets*.
Retrieved from: https://docs.oracle.com/javase/tutorial/networking/sockets/

[6]GeeksforGeeks. (2024). *Multithreading in Java*.
Retrieved from: https://www.geeksforgeeks.org/multithreading-in-java