COMP90015 Distributed System Assignment 2 Report

Zhuoyang Hao 1255309

1 Problem Context

In distributed systems, efficient communication between components is key for scalability. The publisher-subscriber (pub-sub) model decouples message producers (publishers) from consumers (subscribers) through a central broker, ensuring dynamic and scalable messaging. This project implements a pub-sub system with a broker network, using Remote Method Invocation (RMI) for communication across distributed components, enabling reliable message delivery, topic management, and dynamic connectivity.

2 Components of the System

Figure 1 shows the components of the system and Figure 2 illustrates the overall class design:

2.1 Broker (broker/Broker.java)

The **Broker** acts as the central communication hub in the publisher-subscriber architecture. It manages publishers and subscribers, handling the registration of both, and forwards messages from publishers to the relevant subscribers. The broker is responsible for ensuring that each message published under a topic is delivered to all subscribers interested in that topic.

The **Broker** implements both the **IBroker** and **IPubSubInterface** interfaces. The **IBroker** interface handles communication and coordination between brokers in the network, facilitating inter-broker messaging and topic management. On the other hand, the **IPubSubInterface** is dedicated to interaction with publishers and subscribers, handling message forwarding and subscription management for these entities. The broker also manages topic creation, deletion and message publishing.

2.2 Topic (broker/Topic.java)

The **Topic** class represents a subject or category under which messages are published. It acts as the binding point between publishers and subscribers, where each topic can have multiple subscribers interested in receiving messages. This class manages a list of subscribers for each topic and ensures that messages are properly delivered to them when a publisher sends a message.

Importantly, each **Topic** instance is managed locally within each broker, meaning that topic and subscription information is decentralized. This ensures that there is no single point of failure in the system, as each broker maintains its own local state of topics and subscribers. When necessary, brokers communicate updates between each other, but they do not rely on a centralized node to maintain consistency, aligning with the decentralized design requirements of the system.

2.3 DirectoryService (directory/DirectoryService.java)

The **DirectoryService** acts as a lookup service that helps brokers, publishers, and subscribers discover each other. It provides an abstraction layer that simplifies network communication, allowing different components to find each other and establish connections without hardcoding addresses. This service registers and locates brokers, publishers, and subscribers, dynamically connecting them when needed.

2.4 Publisher (publisher/Publisher.java)

The **Publisher** class represents the producer of messages. It interacts with the broker to publish messages to a specific topic. Publishers register with brokers via the directory service and use the broker to forward messages to subscribers. The publisher does not need to manage the complexities of delivering messages directly to subscribers, which is handled by the broker.

2.5 Remote Interfaces (remote/)

- **IBroker**: Defines the operations that brokers must implement for inter-broker communication, such as topic management and message forwarding between brokers.
- **IDirectoryService**: Specifies the interface for the directory service, allowing components (brokers, publishers, subscribers) to discover each other and establish connections dynamically.
- **IPubSubInterface**: Handles interactions between the broker and publishers or subscribers. It manages operations such as publishing messages and managing subscriptions.
- **ISubscriber**: Defines the methods that subscribers must implement, including operations to receive messages and manage subscriptions.

2.6 Subscriber (subscriber/Subscriber.java)

The **Subscriber** class represents the consumer of messages. It subscribes to topics through the broker and receives messages whenever a publisher publishes to those topics. The subscriber listens for messages and processes them accordingly. It communicates with the broker to manage subscriptions and may receive notifications if topics are deleted or modified.

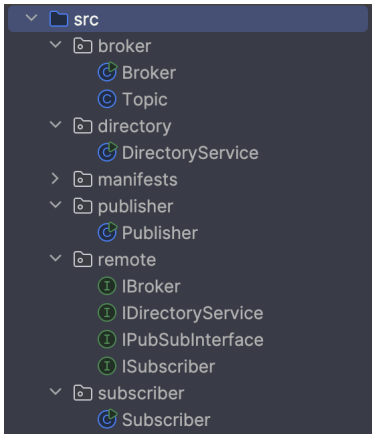

Figure 1: Components of System

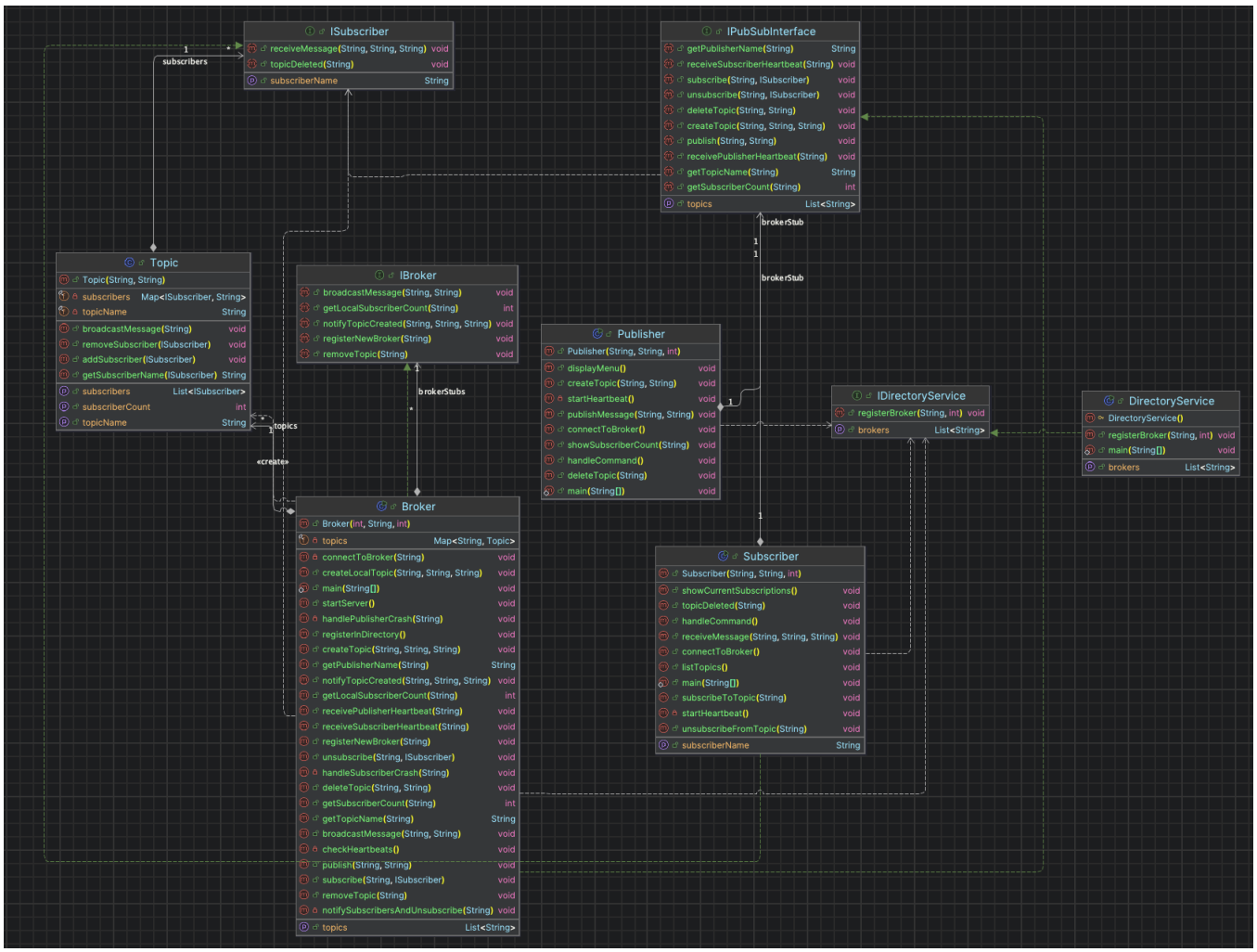

Figure 2: Overall Class Design

3 Interactions

Figure 3 shows the interaction of inital connections, topic creation, topic subscription, topic unsubscription, message publishing and topic deletion.

3.1 Initial Connection to the Directory Service

- The **Broker** first registers itself with the **DirectoryService** by sending its IP address and port, which allows other components to discover it when necessary.
- The **Publisher** and **Subscriber** request the list of available brokers from the **DirectoryService**. This enables both the publisher and subscriber to know which broker they can connect to in the system.
- The **DirectoryService** responds to both the **Publisher** and **Subscriber** with the broker information, allowing them to establish a connection with the broker.

3.2 Publisher Creates a New Topic

- Once connected to the broker, the **Publisher** initiates the creation of a new topic by calling `createTopic(publisherName, topicId, topicName)`. This message contains the publisher's name, the topic's unique ID, and the name of the topic.
- The **Broker** handles this request by creating a new **Topic** instance, which stores the necessary metadata and manages subscribers for that topic.
- After successfully creating the topic, the broker broadcasts the creation of the topic to all connected brokers.

3.3 Subscriber Subscribes to a Topic

- The **Subscriber** expresses interest in receiving messages from the newly created topic by calling `subscribe(topicId)` on the broker. This informs the broker that the subscriber wants to be notified whenever messages are published to this topic.
- The **Broker** forwards the subscription request to the **Topic** in **Broker**, adding the subscriber to the list of subscribers for that topic.

3.4 Publisher Publishes a Message to the Topic

- After the topic has been created, the **Publisher** can now publish messages to the topic. It sends a `publishMessage(topicId, message)` request to the **Broker**, containing the topic ID and the message to be delivered.
- The **Broker** forwards the message to the corresponding **Topic**, which is responsible for delivering the message to all the subscribers registered for that topic.
- The **Topic** delivers the message to each **Subscriber**.
- The **Broker** connected to the publisher then forwards the message to other brokers, who in turn broadcast it to their respective subscribers.

3.5 Subscriber Unsubscribes to a Topic

- When a **Subscriber** decides to stop receiving messages from a specific topic, it calls the `unsubscribeFromTopic(topicId)`.
- The **Broker** forwards this request to the corresponding **Topic** using `unsubscribe(topicId, subscriber)`.
- The **Topic** then removes the subscriber from its internal list of subscribers.

3.6 Delete Topic Process

- When a **Publisher** wants to delete a topic, it utilizes `deleteTopic(topicId)`.
- The broker retrieves the list of all subscribers of the topic by calling `getSubscribers()`.
- The **Topic** then calls `removeAll()` to remove all subscribers from the topic.
- After this, the broker uses `notifySubscriberAndUnsubscribe(topicId)` to notify other brokers in the network that the topic has been deleted, and unsubscribe the subscribers connected to other brokers accordingly.

4 Critical Analysis

4.1 RMI vs Socket

In this project, I implemented a pure RMI connection. In the context of this publisher-subscriber system with brokers, using RMI simplifies development by abstracting network communication and allowing direct method calls between distributed objects, which is beneficial for handling complex object interactions like topic management and message forwarding. RMI also handles object serialization and multi-threading internally, making it easier to manage concurrent subscribers and publishers. However, RMI introduces performance overhead due to its abstraction layers, which can affect message delivery speed, and its tight coupling to Java limits cross-platform flexibility. On the other hand, sockets provide greater control over communication, offering lower overhead and better performance, especially for real-time message passing between brokers and subscribers. Sockets also support different languages communications, making the system more scalable across different platforms. However, they require manual handling of connection management, serialization, and concurrency, which increases system complexity. For a Java-based system where simplicity and ease of use are key, RMI is likely more suitable, but for high-performance, scalable, and cross-platform needs, sockets would be a better option.

4.2 Concurrency Control

In the `Broker` and `Topic` classes, `ConcurrentHashMap` is used to handle concurrency by allowing multiple threads to safely access and modify shared data structures, such as topics, subscribers, and brokers, without requiring explicit synchronization. In the `Broker` class, it ensures thread-safe operations for managing topics, connected brokers, and heartbeats for publishers and subscribers, allowing multiple publishers and subscribers to interact with the broker concurrently. Similarly, in the `Topic` class, `ConcurrentHashMap` manages subscribers, enabling concurrent management including subscription and deletion and safe message broadcasting. This design avoids the need for manual locks, provides better performance through fine-grained locking, and ensures that the system remains scalable and responsive under high concurrency.

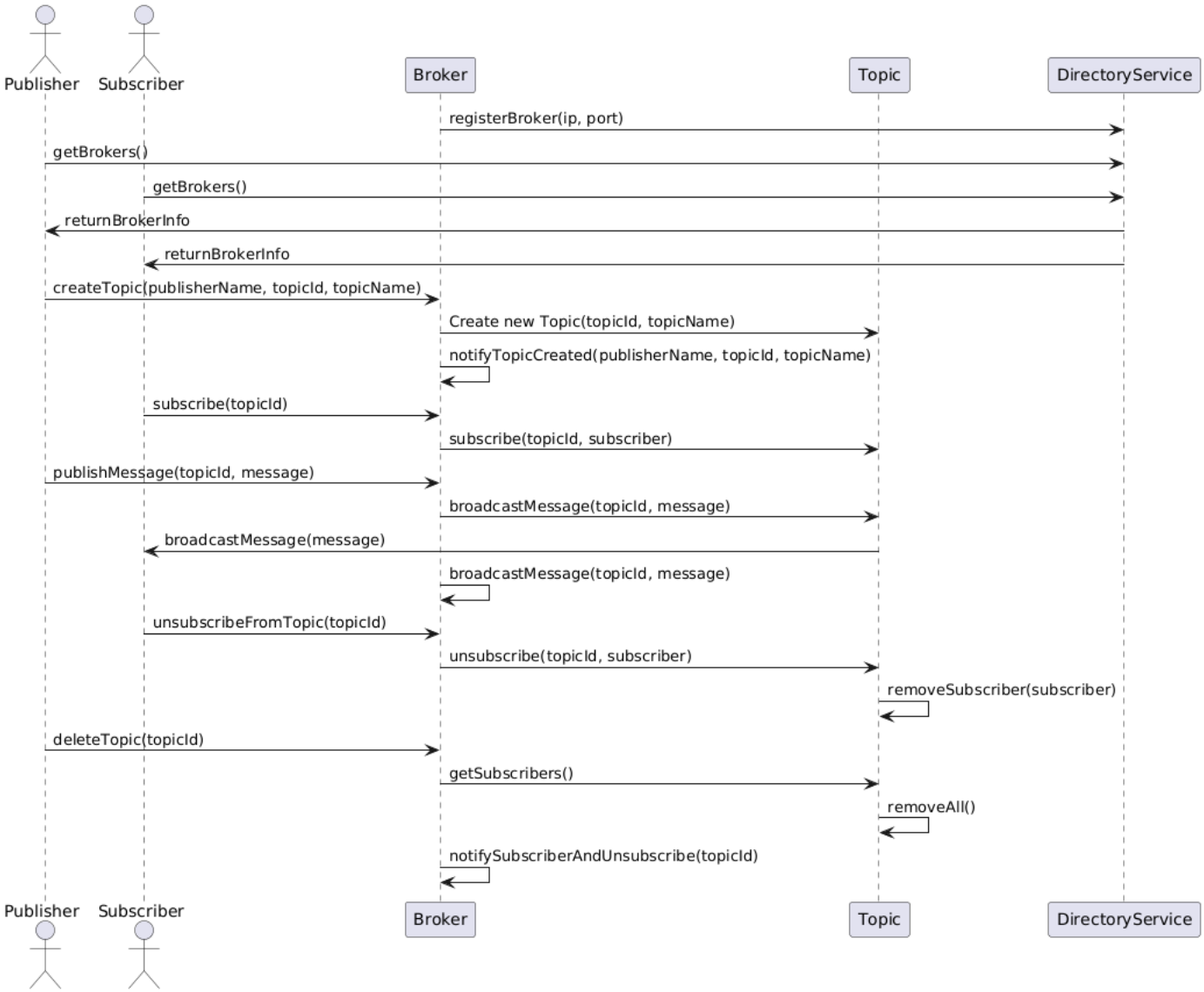

Figure 3: System Sequence Diagram

In the `DirectoryService` class, concurrency control is handled through the use of `synchronized` methods. The `registerBroker` and `getBrokers` methods are synchronized to ensure that only one thread can execute these methods at a time, preventing race conditions when modifying or accessing the list of active brokers. This guarantees that broker registration and retrieval operations are performed consistently, preventing issues such as duplicate entries or inconsistent broker lists. While `synchronized` methods limit parallelism by allowing only one thread to access them at a time, this approach provides a simple and reliable way to ensure thread safety, given the relatively lower frequency of broker registration operations. Overall, the combination of `ConcurrentHashMap` and `synchronized` methods allows the system to maintain high concurrency and responsiveness while ensuring thread safety and consistency.

4.3 Fault Tolerance

In this project, fault tolerance is achieved through a heartbeat mechanism between the publisher / subscriber and the broker. Both publishers and subscribers periodically send heartbeats to the broker every 2 seconds to indicate that they are still active and connected. The broker maintains separate Concurrent HashMap structures to store the most recent heartbeat timestamps for each publisher and subscriber. A scheduled task within the broker continuously checks these heartbeats, and if no heartbeat is received from a publisher or subscriber within a 5-second timeout period, the broker considers that entity to have crashed. If a publisher crashes, the broker deletes all topics associated with the failed publisher and unsubscribe all subscribers. Similarly, if a subscriber crashes, the broker removes the crashed subscriber from all subscribed topics. This mechanism ensures that the system remains robust by automatically detecting and handling node failures, allowing the broker to gracefully manage lost connections and maintain consistent state.

5 Conclusion

In this project, a distributed publisher-subscriber system with brokers was implemented using RMI to facilitate communication between components. The system effectively manages message distribution, topic creation, subscription management, and fault tolerance across multiple brokers. Key features such as concurrent handling of topics and subscribers were efficiently managed using `ConcurrentHashMap`, allowing for scalability and robust performance under high concurrency. The heartbeat mechanism between the broker, publisher, and subscriber ensures that the system is fault-tolerant by detecting and handling node failures gracefully.