Report: Distributed Publisher-Subscriber System

Student Name: Qiyue Mei

Student ID: 1554024

---

1. Overall Class Design and Interaction Diagram

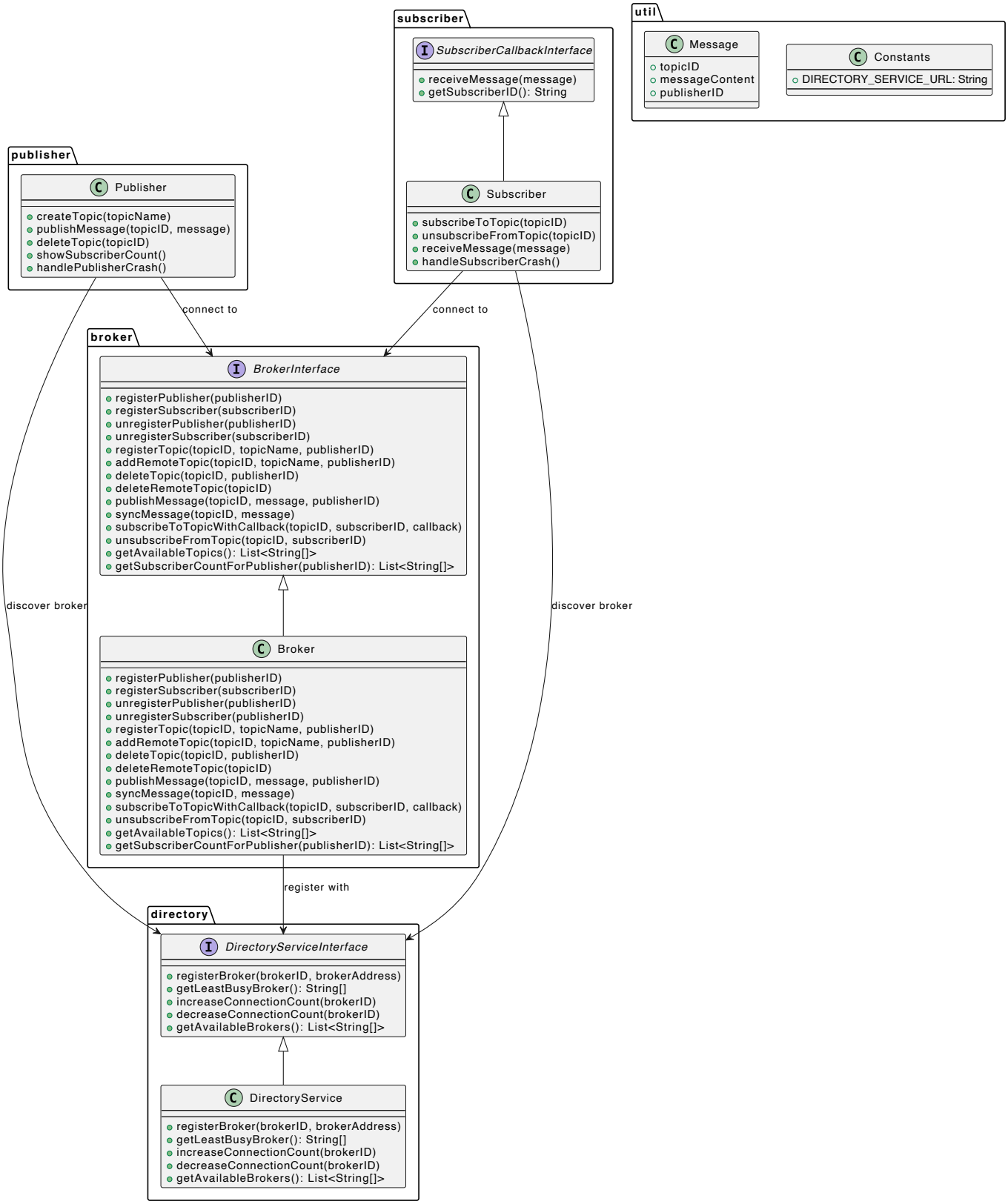
The **Distributed Publisher-Subscriber System** is composed of several distinct components interacting to provide real-time message distribution between publishers and subscribers. The key classes in the system are:

- **Directory Service**: Manages the registry of brokers, helps publishers and subscribers connect to the broker.
- **Broker**: Acts as an intermediary between publishers and subscribers. Brokers manage topics, handle subscriptions, and route messages across the network of brokers.
- **Publisher**: Publishes messages on specific topics to its connected broker.
- **Subscriber**: Subscribes to topics via a broker and receives real-time messages on those topics.

1.1 Class Diagram Overview:

- `diretorcy` Package
  - **DirectoryServiceInterface**: Provides the necessary methods for managing brokers.
  - **DirectoryService**: Implements the directory logic for managing broker registrations and connection counts.
- `broker` Package
  - **BrokerInterface**: Defines the operations available on the broker, including publishing, subscribing, and inter-broker communication.
  - **Broker**: Implements the broker logic, including topic management, message routing, and subscriber notifications.
- `publisher` Package
  - **Publisher**: Allows users to create topics, publish messages, and manage their topics.
- `subscriber` Package
  - **SubscriberCallbackInterface**: Provides a callback mechanism for subscribers to receive real-time messages.
  - **Subscriber**: Implements the subscriber logic, allowing users to list topics, subscribe to them, and receive messages.
- `util` Package
  - **Message**: Encapsulates the message exchange protocol.

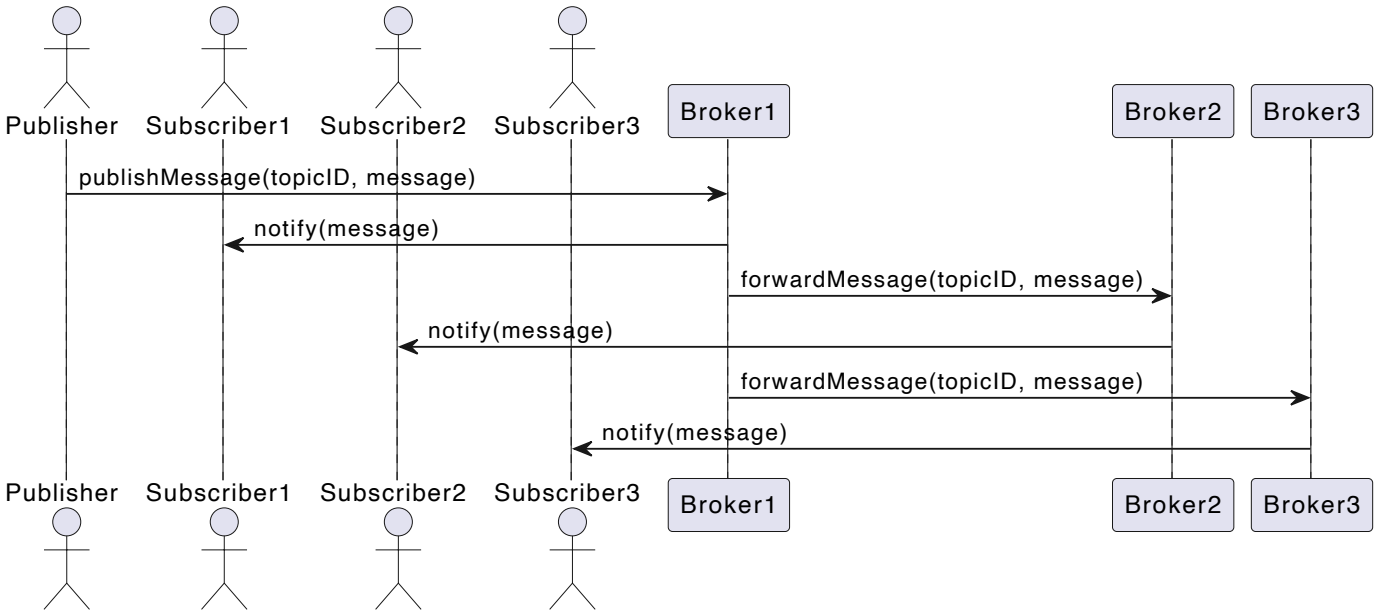
- **Constants**: Defines various constants used throughout the publish-subscribe system, such as the Directory Service URL and other configuration parameters.



1.2 Interaction Diagram:

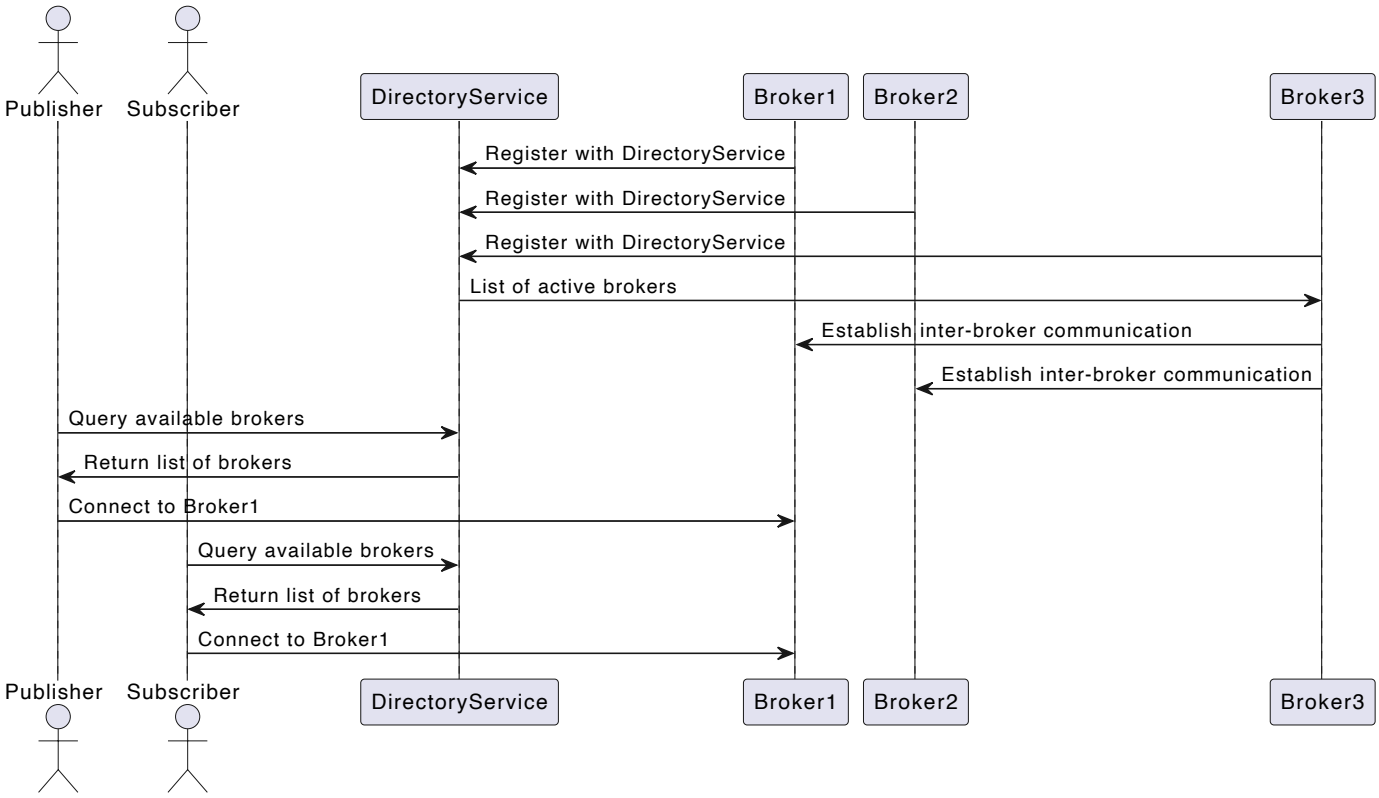
1. **Publisher publishes a message**:
    - The publisher sends a message to its connected broker.

- The broker receives the message, and if the topic has subscribers, it forwards the message to all brokers (inter-broker communication).
- Each broker then notifies the subscribers connected to it.



2. **Dynamic Broker Discovery**:

- Broker Startup:
  - New brokers register with the Directory Service, which maintains an updated list of all active brokers.
  - The Directory Service provides new brokers with the list of existing brokers for establishing inter-broker communication via RMI.
- Broker Interconnection:
  - New brokers connect to existing brokers, and all brokers establish a fully connected network.
- Publisher/Subscriber Discovery:
  - Publishers and subscribers query the Directory Service to discover available brokers, select one, and register with it.
  - The selected broker informs the Directory Service of the new connection.



2. System Components

2.1 Directory Service

- **Role**: Acts as a registry for brokers, allowing clients to dynamically discover brokers.
- **Key Functions**:
  - **registerBroker()**: Registers brokers in the system.
  - **getLeastBusyBroker()**: Returns the broker with the lowest connection count.
  - **getAvailableBrokers()**: Lists available brokers for client selection.
  - **increaseConnectionCount() & decreaseConnectionCount()**: Updates broker connection counts when clients connect or disconnect.

2.2 Broker

- **Role**: Manages topics, message routing, and subscriber notifications, while facilitating inter-broker communication.
- **Key Functions**:
  - **registerPublisher() & unregisterPublisher()**: Manages publisher registration, unregistration, and crash handling.
  - **registerSubscriber() & unregisterSubscriber()**: Manages subscriber registration, unregistration, and crash handling.
  - **registerTopic() & deleteTopic()**: Handles topic creation, deletion, and synchronization across brokers.
  - **handleCrash()**: Handles unexpected crashes.

- **publishMessage()**: Routes messages to local and remote subscribers.
- **syncMessage()**: Synchronizes messages between brokers.

2.3 Publisher

- **Role**: Creates topics and publishes messages through a connected broker.
- **Key Functions**:
  - **connectToBroker()**: Dynamically connects to brokers via the Directory Service.
  - **registerTopic()**: Creates topics on the broker.
  - **publishMessage()**: Publishes messages to topics.
  - **deleteTopic()**: Deletes topics and notifies subscribers and brokers.

2.4 Subscriber

- **Role**: Subscribes to topics and receives real-time messages via RMI callbacks.
- **Key Functions**:
  - **connectToBroker()**: Dynamically connects to brokers via the Directory Service.
  - **subscribeToTopicWithCallback()**: Subscribes to topics for real-time notifications.
  - **receiveMessage()**: Receives messages when publishers post to subscribed topics.
  - **unsubscribeFromTopic()**: Unsubscribes from topics and receives confirmation.

3. Critical Analysis and Design Choices

3.1 Use of RMI for Communication

- **Justification**: The system uses Java's Remote Method Invocation (RMI) to facilitate communication between publishers, subscribers, and brokers, and to handle **RMI callback mechanisms** for real-time message delivery to subscribers.
- **Advantage**:
  - **High-level abstraction**: RMI abstracts away much of the complexity involved in setting up network communication between distributed systems. The automatic serialization and deserialization of objects simplify message transmission between brokers, publishers, and subscribers. This reduces the manual overhead of managing low-level sockets or raw data streams, allowing developers to focus on core business logic.
- **Improvement Consideration**:
  - **Expanded registry for non-localhost environments**: Currently, the system might be limited to `localhost` or single-node setups. To scale the system across multiple machines, the RMI registry needs to be expanded to handle external network addresses and possibly distributed registries.
  - **Load balancing enhancements**: While the system currently uses **connection count** as the primary metric for distributing publisher and subscriber workloads, this could be improved. Load balancing algorithms could consider network latency, CPU usage, memory consumption, and network performance across brokers to more effectively distribute workloads.

3.2 Handling Concurrency with Synchronized Methods

- **Justification**: The use of **synchronized methods** in critical parts of the system, such as topic management and subscriber registration, ensures that shared resources are properly managed in a multi-threaded environment, preventing data races and ensuring consistency.
- **Advantage**:
  - **Simplicity and effectiveness**: Using synchronized blocks or methods is a straightforward way to ensure thread safety when multiple threads are accessing shared data. It prevents race conditions and ensures that only one thread modifies the shared state at a time.
- **Improvement Consideration**:
  - **Use of more scalable data structures**: As the number of concurrent operations grows, reliance on synchronized methods might introduce performance bottlenecks due to thread contention. A **ConcurrentHashMap** or **ConcurrentHashSet** can provide a lock-free or finer-grained locking alternative, improving system scalability and performance.
  - **Explicit locks**: In some cases, using `ReentrantLock` or other advanced locking mechanisms may provide better flexibility and performance, especially in scenarios where locks need to be acquired and released under specific conditions.

3.3 Inter-Broker Communication for Message Routing

- **Justification**: The system employs a fully connected topology where each broker communicates with every other broker in the network. This ensures that when a message is published to one broker, it is propagated to all other brokers, and subscribers connected to any broker can receive the message.
- **Advantage**:
  - **Reliable message propagation**: The fully connected architecture ensures that messages can be propagated across the network of brokers without needing a central coordinator, reducing single points of failure and enhancing system reliability.
- **Improvement Consideration**:
  - **Advanced routing algorithms**: The current approach can lead to inefficiencies as the network scales. Implementing advanced routing algorithms, such as **flooding** (broadcasting the message to multiple brokers) or **multicast routing** (sending messages only to brokers with subscribers interested in the topic), could improve message propagation efficiency and reduce unnecessary network traffic.

3.4 Fault Tolerance

- **Justification**: The system differentiates between graceful exits and crashes of both publishers and subscribers. If a publisher crashes, all associated topics are deleted, and subscribers are unsubscribed. If a subscriber crashes, only the subscriptions are removed, leaving the topics intact.
- **Advantage**:
  - **Robust handling of crashes**: By differentiating between **unexpected crashes** and **graceful exits**, the system can appropriately clean up resources to maintain consistency and prevent memory leaks or orphaned data. It also ensures that system states are updated accurately depending on the type of shutdown.

- **Improvement Consideration**:
    - **Improved fault tolerance with security**: To enhance the fault-tolerance mechanism, additional security measures could be introduced to verify the integrity of publishers and subscribers. For instance, **heartbeat mechanisms** could be employed to continuously check the health of clients, or **authentication mechanisms** could ensure that only authorized clients can reconnect or restart after a crash.
    - **Recovery mechanisms**: Consider implementing **topic persistence** or **subscriber session recovery** to allow subscribers and publishers to resume their activities after a crash, without having to re-subscribe or re-create topics.

4. Conclusion

This final Distributed Publisher-Subscriber System in this project is a well-architected, scalable, and fault-tolerant system that leverages Java RMI to handle real-time message distribution. It demonstrates effective use of a broker network to manage publishers and subscribers and provides a strong foundation for building larger, more complex systems.