IT559 Distributed Systems – Lab Manual

Lab 8 – Message Passing and Publish-Subscribe Model

Write a program to implement **Message Passing** and **Publish-Subscribe (Pub-Sub) Model** in a distributed system. After executing each program, capture the output in a text file format.

Program files should be named as:

- `StudentID_MessagePassing_x.py` (for Python) or `StudentID_MessagePassing_x.java` (for Java)

Text files with captured output should be named as:

- `StudentID_MessagePassing_x.txt`

Introduction to Message Passing and Pub-Sub Model

Message Passing is a fundamental mechanism in distributed systems that allows processes to communicate with each other over a network. The **Publish-Subscribe (Pub-Sub) Model** is a popular messaging pattern where publishers send messages without knowing the receivers, and subscribers receive messages based on their subscriptions.

Key Concepts:

1. **Message Passing:**
    - Processes communicate by sending and receiving messages.
    - Can be implemented using **sockets, message queues, or RPC.**
    - Two main types: **Synchronous (blocking)** and **Asynchronous (non-blocking).**
2. **Publish-Subscribe Model:**
    - Publishers send messages to a **broker or topic.**
    - Subscribers receive messages based on their subscriptions.
    - Decouples senders from receivers, making communication scalable.

Comparison of Message Passing and Pub-Sub

Feature	Message Passing	Publish-Subscribe
Direct Communication	Yes	No (Uses Broker)
Scalability	Limited	High
Message Targeting	Specific Process	Multiple Subscribers
Examples	MPI, IPC	Kafka, RabbitMQ

For more details, refer to:

- Message Passing in Distributed Systems
- Publish-Subscribe Model

Problem 1: Implementing Basic Message Passing

Write a program to implement direct **Message Passing** between multiple processes.

Implementation Steps:

1. **Create multiple processes** (e.g., P1, P2, P3).
2. **Implement a communication mechanism:**
    - Use **sockets** or **message queues** for sending and receiving messages.
3. **Demonstrate synchronous and asynchronous messaging.**
4. **Display message exchange logs.**

Expected Output Example:

P1 sends message to P2: "Hello P2!"
P2 receives: "Hello P2!"
P2 sends message to P3: "P2 to P3 communication."
P3 receives: "P2 to P3 communication."

Problem 2: Implementing Publish-Subscribe Model

Extend Problem 1 to implement a **Pub-Sub Model** where multiple subscribers receive messages from a central broker.

Implementation Steps:

1. **Set Up a Message Broker:**
    - Implement a broker that maintains different topics and manages subscriptions.
    - Store subscriber information in a data structure such as a dictionary (`topic -> list of subscribers`).
2. **Implement the Publisher:**
    - A publisher sends messages to a specific topic.
    - The message is forwarded to the broker, which distributes it to all active subscribers of that topic.
3. **Implement the Subscriber:**
    - A subscriber registers itself with the broker by subscribing to a topic.
    - The broker maintains a record of which subscribers are linked to which topics.
    - Whenever a new message arrives for a subscribed topic, the broker pushes it to the relevant subscribers.
4. **Handling Multiple Subscribers:**
    - Ensure multiple subscribers can receive messages for the same topic.
    - Implement a notification mechanism where messages are forwarded asynchronously to subscribers.
5. **Testing the Pub-Sub Model:**
    - Create multiple publishers and subscribers and test the flow where multiple publishers send messages to different topics.
    - Verify if all relevant subscribers receive messages correctly.

Expected Output Example:

Publisher publishes: "New update available!" to Topic: Updates
Subscriber 1 receives: "New update available!"
Subscriber 2 receives: "New update available!"

Problem 3: Implementing Scalable Pub-Sub with Multiple Topics

Modify Problem 2 to support multiple topics and multiple publishers.

1. Allow **dynamic subscription** where subscribers can subscribe/unsubscribe at runtime.
2. Allow **multiple publishers** to publish messages to different topics.
3. Ensure **thread-safe and efficient message delivery.**
4. Use **multithreading or multiprocessing** to simulate real-world messaging systems.

Optional Assignment: Implementing Pub-Sub with Kafka or RabbitMQ

- Modify your program to use **Kafka** or **RabbitMQ** for handling message publishing and subscription.
- Implement a **real-world example** such as **live news updates** or **chat system** using the pub-sub model.

Submission Instructions

Submit a compressed `.zip` file named `StudentID_MessagePassing.zip`, containing the following files:

- `MessagePassing.py` (or `.java` equivalent)
- `PubSub.py` (or `.java` equivalent)
- Corresponding output text files.

Ensure that your programs are well-commented and handle errors properly.