## Overview

This report describes the design and implementation of a simple publish-subscribe messaging system in Python. The system aims to provide a basic framework for decoupled communication between message producers (publishers) and message consumers (subscribers). It addresses several challenges in distributed messaging, including dynamic topic management, and fault tolerance. The system is designed to be scalable to handle a throughput of 1000 messages per second. All the requirements (R1 – R10) have been implemented and match the requirements of the specification.

## Design and Implementation

### System Design

The system comprises of four core components: Message, MessageQueue, PubSubSystem, and Subscriber. These components correspond to a respective python file which contain corresponding code.

The Message file is a data structure to represent what a message should look like. The structure represents a message with a unique identifier (id), topic, payload, timestamp, and attempt count. The unique ID is crucial for identifying and handling duplicate messages. The timestamp is used for tracking message order. The attempts counter is used for re-sending dropped messages. This class stands purely as a blueprint for creating message objects by defining their structure and behavior.

MessageQueue is the second component of the project. This file manages a queue of messages for a specific topic. It uses a Python list, self.queue, to store messages and a threading.Lock() for thread-safe access. The crashed attribute indicates the state of the queue.

PubSubSystem is the third component of the system and contains most of the logic to make the system work. It manages all message queues in a dictionary, self.queues, where the key is the topic name, and the value is the MessageQueue object. It uses a threading.Lock() to protect access to the queue dictionary. It also maintains a dictionary of subscribers.

Finally, Subscriber is the fourth and last component of the system. It represents a subscriber to a topic. It has a name and a list of received messages.

**Communication/Interaction Patterns**

A publisher creates a Message object and calls the PubSubSystem.publish() method, specifying the topic and payload. The PubSubSystem then adds the message to the corresponding MessageQueue. A subscriber calls the PubSubSystem.subscribe() method, specifying the topic it wants to subscribe to and passing a reference to itself. The PubSubSystem adds the subscriber to the subscriber list of the corresponding MessageQueue. The PubSubSystem runs a delivery thread that continuously dequeues messages from the MessageQueues and delivers them to the associated subscribers by calling the subscriber's receive() method. The system incorporates mechanisms to handle temporary interruptions, queue crashes, and subscriber crashes. Simulated network delays and message drops are also included as separate methods which can be called in testing to show functionality.

**Requirement Implementations**

All the requirements have been implemented and highlighted below:

R1: Subscribing and publishing events:

Understanding: This involves implementing the basic functionality for publishers to send messages to topics and for subscribers to register their interest in specific topics to receive those messages.

Design Assumptions: Assumes that topics are strings and messages contain a payload of arbitrary data.

Solution: The PubSubSystem.publish() method adds messages to the appropriate queue, and the PubSubSystem.subscribe() method registers subscribers with the queue.

Testing Results: Verified through the test_pubsub() function by publishing messages to a topic and asserting that the subscribers receive them.

Reflection: Basic pub-sub functionality is implemented correctly.

R2: Lookup, discovery, and access of event channels:

Understanding: This requires a mechanism for publishers and subscribers to find and access the message queues (channels) associated with specific topics.

Design Assumptions: Topics are represented by strings, which serve as keys to access the corresponding message queues.

Solution: The PubSubSystem class maintains a dictionary (self.queues) that maps topics to MessageQueue objects. The PubSubSystem.get_queue() method provides access to these queues.

Testing Results: The test_pubsub() function verifies that a queue can be retrieved for a valid topic and that None is returned for a non-existent topic.

Reflection: Topic lookup is implemented efficiently using a dictionary.

R3: Dynamic message queue management:

Understanding: This involves the ability to create and manage message queues dynamically, without requiring predefined configurations.

Design Assumptions: Queues can be created at any time during the execution of the system.

Solution: The PubSubSystem.create_queue() method creates a new MessageQueue object and adds it to the self.queues dictionary.

Testing Results: The test_pubsub() function creates a new queue and verifies that it is successfully created.

Reflection: Dynamic queue creation is supported.

R4: Temporary interruptions of connections:

Understanding: This refers to handling brief network outages or delays that might occur during message transmission.

Design Assumptions: Network interruptions are simulated with a short delay.

Solution: The simulate_network_delay() function simulates a delay using time.sleep(). This delay represents a temporary interruption.

Testing Results: The test_pubsub() function uses simulate_network_delay() to introduce a delay and checks if the system continues to function.

Reflection: The system can handle simulated temporary interruptions.

R5: Crashing queues:

Understanding: This involves handling situations where a message queue becomes unavailable due to a software or hardware failure.

Design Assumptions: Queue crashes are simulated by setting a crashed flag in the MessageQueue object. Messages sent to a crashed queue are lost.

Solution: The MessageQueue.crash() method sets the crashed flag, and the MessageQueue.recover() method resets it. The enqueue() and dequeue() methods check this flag.

Testing Results: The test_pubsub() function simulates a queue crash, sends a message during the crash, and verifies that the message is not delivered.

Reflection: The system handles queue crashes by discarding messages sent during the downtime.

R6: Crashing customers:

Understanding: This refers to handling situations where a subscriber becomes unavailable.

Design Assumptions: Subscriber crashes are simulated by unsubscribing and resubscribing the subscriber.

Solution: The simulate_subscriber_crash() function simulates a subscriber crash by unsubscribing and resubscribing the subscriber.

Testing Results: The test_pubsub() function simulates a subscriber crash and verifies that the subscriber recovers and receives messages after recovery.

Reflection: The system handles subscriber crashes by allowing them to recover and rejoin the subscription.

R7: Long delays in network traffic:

Understanding: This is similar to R4, but with longer delays.

Design Assumptions: Network delays are simulated using time.sleep().

Solution: The simulate_network_delay() function is used with a larger delay value.

Testing Results: Tested implicitly within test_pubsub() by using a sleep.

Reflection: The system can handle long delays.

R8: Dropped messages:

Understanding: This involves handling scenarios where messages are lost during transmission.

Design Assumptions: Message drops are simulated randomly with a probability.

Solution: The simulate_message_drop() function simulates message drops with a given probability.

Testing Results: The test_pubsub() function uses simulate_message_drop() and verifies that the number of received messages is less than or equal to the number of sent messages.

Reflection: The system can simulate dropped messages.

R9: Out of order messages:

Understanding: This refers to messages arriving at the subscriber in a different order than they were sent.

Design Assumptions: The system can track message order using timestamps, but does not guarantee ordered delivery.

Solution: The Message class includes a timestamp attribute. While the system doesn't reorder messages, the timestamp can be used by the subscriber to detect out-of-order delivery.

Testing Results: The test cases do not explicitly test for reordering, but the timestamp is present in the delivered messages.

Reflection: The system provides the means to detect out-of-order messages but does not guarantee ordered delivery.

R10: Duplicated messages:

Understanding: This involves handling situations where a subscriber receives the same message multiple times.

Design Assumptions: The system can track message duplication using message IDs, but does not prevent it.

Solution: The Message class includes a unique id attribute. Subscribers can use this ID to detect and discard duplicate messages.

Testing Results: The test cases do not explicitly test for duplicate message handling, but the unique ID is present in the delivered messages.

Reflection: The system provides the means to detect duplicate messages but does not prevent their delivery.

## Conclusion

This project implemented a pub-sub messaging system in Python that addresses several important challenges in distributed communication. The system supports dynamic topic management, and fault tolerance. The core functionality such as publishing, subscribing, and topic management was all properly implemented into the system. There are also addresses to fault tolerance for temporary interruptions, queue crashes, and subscriber crashes. The system can simulate network delays and message drops. Furthermore, messages include IDs and timestamps for message tracking. The system is very modular and extensible in design. On top of

all the specification requirements, I also implemented a CLI interface to run the publish and subscriber model which enables smoother testing. While implementing this practical I found several challenges. Ensuring thread safety and preventing race conditions proved to provide several issues at the start. Once I was able to add thread locking this issue was prevented. If I was given more time there are several features that I would implement such as adding support for message filtering and routing as well as adding a better message ordering system so that messages send in the correct order according to their ID.

## Instructions

Proper documentation to run and test the application can be found in the README file. Ensure that Python 3.x is installed on the machine where the code will be tested. There are no external libraries required to run the program. There are two ways to run the program. Once you have cd into the src directory, running ‘python3 test.py’ will run an automated stacscheck of sorts which tests all the requirements of the practical. Opening the test file will show comments corresponding to what is being tested. Alternatively you could run the program by opening two separate terminals and running the publisher and subscriber separately which the commands outlined in the README.