Introduction

# Purpose:

The Priority message queue system make possible to communicate between multiple threads in a concurrent environment by providing a message queue by assigning priority. Threads can enqueue messages with specified priority, and the program ensure that highest priority messages are processed before lower priority ones. This document provide an architectural and design overview of the system, detailing its components,interactions and implementation details.

# Scope:

The scope of this document summarize the architecture and design considerations of the Priority Message Queue. It outlines the key components involved , their responsibilities,interactions and implementation details.Additionally, it discusses the design goals, assumptions, testing strategy, and potential future improvements.

Architectural Overview

# High Level description:

The Priority message queue is designed as a milti threaded application for facilitating communication between threads with thor priority. It consists of three main components:the priority message queue, sender threads, and receiver thread.

# Components and their work:

## PriorityMessageQueue:

Manages the message queue with priority

**Work:**

* Enqueue messages with specified priorities.
* Dequeue messages according to priority.
* Provide methods for peeking at the next message and checking if the queue is empty.

**Implementation details:**

* Utilizes “queue.PriorityQueue” for storing messages.
* Ensures thread safety using ‘threading.lock’.
* Synchronizes access to the queue using ‘threading.Condition’.

## Sender Threads

Threads responsible for sending messages to the message queue

Work:

* Send messages with specified priorities to the PriorityMessageQueue

Implementation:

* Accept parameters such as message content and priority
* Call ‘ enq\_message’ method of the Prioritymessagequeue to send messages

## Receiver thread

Thread responsible for receiving and processing messages from the message queue

Work:

* Continuously wait for messages from the Prioritymessagequeue
* Process received messages

Implementation details:

* Call the ‘dq\_message’ function of the priority message queue to receive messages.
* Print the message

# Interaction between components

* Sender threads send messages to the Priority message queue
* The receiver thread dequeue message from the priority message queue and process them

This architectural overview provides a high level understanding of the priority message queue, its components, responsibilities, and interactions. Subsequent sections will delve deeper into each components design and implementation details

Design Goals

# Functional requirements

### **Message Prioritization**:

**Goal**: Allow threads to enqueue messages with specified priorities

**Rationale**: prioritization ensures that higher priority messages are processed before lower priority ones, enabling critical messages to be handled promptly .

### Thread Safety

Goal: ensure safe concurrent access to the message queue

Rationale: Prevent race conditions and data corruption when multiple threads access the queue simultaneously,guaranteeing the integrity of message handling

### Message Delivery

Goal:ensure reliable delivery of messages to receiver threads

Rationale: Guarantee that all enqueued messages are eventually received and processed by the appropriate receiver thread, maintaining communication integrity

### Performance

Goal: Optimize message processing efficiency and throughput

Rationale:enable the system to scale gracefully as the workload increases, accommodating growing concurrency requirements.

# Non-functional Requirements

### Reliability

Goal: ensure system robustness and fault tolerance

Rationale: Handle errors gracefully recover from failures, and maintain message processing continuity to prevent data loss or system downtime

### scalability

Goal: support a large number of sender and receiver threads.

Rationale:Enable the system to scale gracefully as the workload increases, accommodating growing concurrency requirements

### Maintainability

Goal: Facilitate ease of system maintenance and updates.

Rationale: Employ modular design principles, clear code organization, and documentation practices to simplify troubleshooting, enhancements, and future modifications.

### Portability

Goal: Ensure compatibility across different environments and platforms.

Rationale: Design the system to be platform-independent and minimize dependencies on external libraries or system-specific features.

### Extensibility

Goal: Allow for easy extension of functionality.

Rationale: Design the system with flexibility to accommodate future requirements or enhancements without major architectural changes, promoting adaptability and longevityy.

Assumptions and Constraints

# Assumptions

### Concurrent Environment:

* The system operates in a concurrent environment where multiple threads may access the message queue simultaneously.

### Message Prioritization:

* Messages are enqueued with integer-based priorities, where lower values represent higher priorities.

### Thread Safety:

* Thread safety is maintained using locks and synchronization mechanisms to prevent race conditions and ensure data integrity.

### Python Environment:

* The system is designed and implemented in Python, leveraging its built-in threading and synchronization primitives.

## 

# Constraints

### Performance Overhead:

* Implementing thread safety mechanisms may introduce performance overhead, impacting the system's throughput and latency.

### Resource Limitations:

* The system's scalability may be limited by available system resources such as CPU cores, memory, and I/O bandwidth.

### Message Processing Speed:

* The processing speed of the receiver thread may affect the system's ability to handle a high volume of incoming messages efficiently.

### Platform Compatibility:

* The system's compatibility with different platforms and Python versions may be constrained by platform-specific features and dependencies.

### Message Content Size:

* The size of message content may be limited by memory constraints, affecting the system's ability to handle large messages efficiently.

Testing Strategy

# Unit Testing

### Priority Message Queue

### **Enqueue Message Test:** Verify that messages are enqueued correctly with the specified priorities.

### **Dequeue Message Test:** Ensure that messages are dequeued in the correct order based on priority.

### **Peek Message Test:** Validate that peeking at the next message does not remove it from the queue.

### **Empty Check Test:** Verify that the queue emptiness is correctly determined.

### Sender And Receiver Thread

* **Sender Thread Test:** Validate that sender threads can correctly enqueue messages with specified priorities.
* **Receiver Thread Test:** Ensure that the receiver thread can correctly dequeue and process messages from the queue.

# Integration Testing

**Sender-Receiver Interaction Test:** Validate the interaction between sender threads and the receiver thread by verifying that messages sent by sender threads are correctly received and processed.

**Concurrency Test:** Evaluate the system's behavior under various concurrency scenarios, ensuring thread safety and proper synchronization.

### 

# Performance Testing

**Throughput Test:** Measure the system's message processing throughput under different loads and concurrency levels.

**Latency Test:** Evaluate the time taken for a message to be enqueued, dequeued, and processed by the receiver thread under normal and peak loads.

**Scalability Test:** Assess the system's ability to scale with increasing numbers of sender and receiver threads.