Scalable SMS Alert Simulation System

Table of Contents

1. Introduction
2. Technology Stack
3. System Design and Architecture
4. Implementation
5. Concurrency and Scalability
6. Unit Testing
7. Tradeoffs and Considerations
8. Conclusion

Introduction

This document provides an outline for the design and implementation of a scalable SMS alert simulation system. The system consists of three main components:

1. **Producer**: Generates a configurable number of random SMS messages.
2. **Senders**: A pool of configurable workers that consume messages and simulate sending them.
3. **Progress Monitor**: Tracks and displays the progress of message sending, including statistics like the number of messages sent, failed, and the average time per message.

The system is designed with scalability and concurrency in mind so that performance is efficient, even with a large amount of messages and senders. This model makes use of asynchronous programming to manage multiple concurrent tasks.

Technology Stack

- Programming Language: Python 3.9+
- Concurrency Model: `asyncio` for asynchronous operations
- Data Structures: `asyncio.Queue` for message passing
- Testing Framework: `pytest`
- Logging: Pythons built-in `logging` module

Reasons for choosing Python:

- Clear and concise syntax
- Good framework for asynchronous operations
- Global Interpreter Lock, eliminates need for multithreading
- Easy to scale with external libraries like Kafka or RabbitMQ
- Easy to troubleshoot

System Design and Architecture

Components Overview

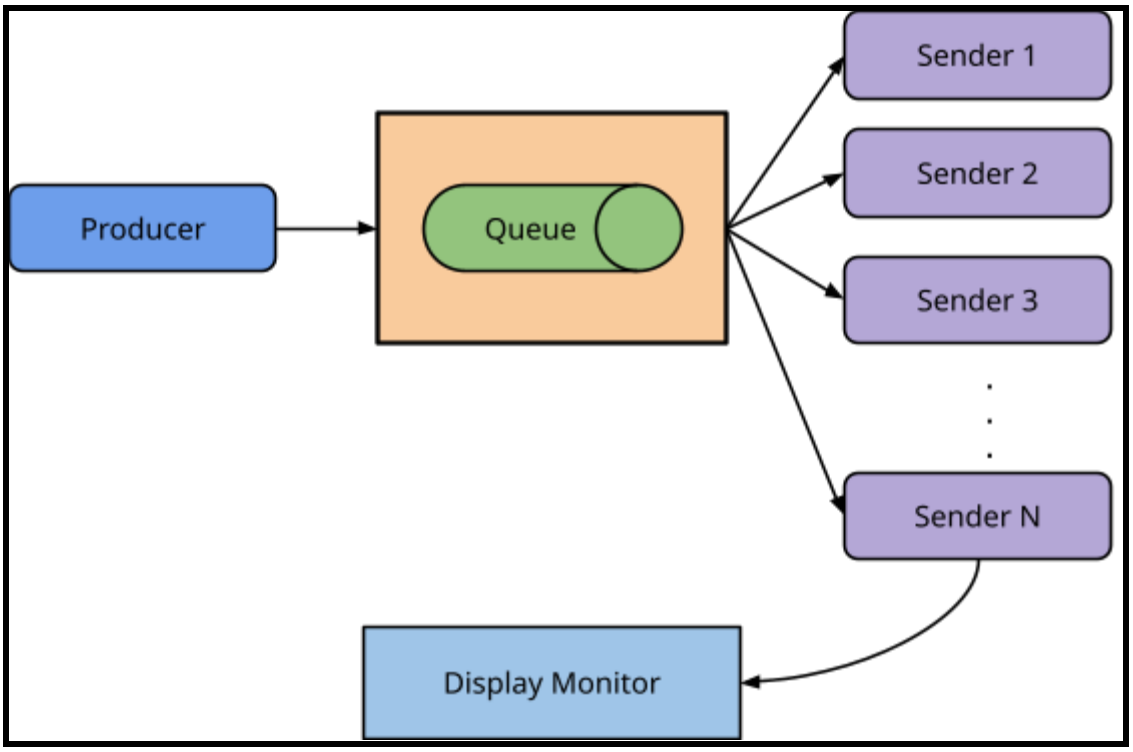
1. Producer Model:
    - Generates a configurable number of SMS messages with message ID and content
    - Each message contains up to 100 random characters
    - Messages are added to an asynchronous queue in batches of 1000 for senders to consume
    - Queue appends a sentinel value of `None` to indicate that no new messages will be produced
2. Sender Model:
    - Creates an configurable number of sender instances

    - Each sender retrieves a message from the queue and simulates sending the message by updating statistics accordingly
    - Each sender has a configurable failure rate and mean send-time that simulates failure and network delay
    - Senders are on standby to process messages until they receive a sentinel value
3. Display Monitor
    - Periodically outputs (with a configurable interval):
        - Time since program started running
        - Number of messages sent successfully
        - Average send time for all sent messages
    - Displays final statistics once all messages have been processed

Architecture

The architecture chosen to simulate an SMS alert system is a **message queueing system** which allows for asynchronous communication between producers and consumers (senders). The main operations of producing and sending messages can be blocking, so an asynchronous queue allows both producers and senders to context-switch accordingly. A message queue system is also able to scale well as messages generation can be distributed among multiple producers and message sending can be load-balanced by multiple senders.

Architectural Diagram:



Implementation

Project Structure

```
sms-simulation/
├── models/
│   ├── producer_model.py
│   ├── sender_model.py
│   └── display_monitor_model.py
├── tests/
│   ├── test_producer.py
│   ├── test_sender.py
│   └── test_display_monitor.py
├── docs/
│   └── technical_documentation.pdf
├── config.py
├── main.py
└── README.md
```

Configuration

The `config.py` file handles the configurable default parameters for producers, senders, and the display monitor

Producer Model

The `producer_model.py` file is responsible for the Message data class and handling the generation and enqueuing system for the asyncio Queue. There is a batch processing system to allow senders to start processing before the producer completes:

```
if msgs_in_batch >= self.batch_size:
                    msgs_in_batch = 0
                    logger.info(f"Produced batch of {self.batch_size} messages")
                    await asyncio.sleep(0.001)
```

Sender Model

The `sender_model.py` manages the creation of multiple senders with a unique ID and a configurable send time and failure rate. The `sender_message()` functionality simulates the network delay and failure while the `run()` function dequeues messages concurrently until a sentinel value is reached ：

```
async def run(self):
        """
        Main loop that continuously processes messages from the queue
asynchronously.
        Stops when a sentinel value of None is received.
        """
        self.running = True
        logger.info(f"Sender {self.id}: starting message processing")

        try:
            while self.running:
                message = await self.queue.get()
                if message is None:  # Sentinel value received
                    logger.info(f"Sender {self.id}: recieved sentinel")
                    self.queue.task_done()
```

```
                break
            await self.send_message(message)
            self.queue.task_done()
```

Display Monitor

The `display_monitor_model.py` keeps track of statistics in a centralized location and can be configured to update every N seconds.

```
while True:
        await asyncio.sleep(config.monitor_interval)
        counter +=1
        current_time = counter * config.monitor_interval #keep track of time elapsed

        sent = stats.get('sent', 0)
        failed = stats.get('failed', 0)
        total_time = stats.get('total_time', 0.0)
        avg_time = (total_time / sent) if sent > 0 else 0.0
        print(f"[Monitor] {current_time}s, Sent: {sent}, Failed: {failed}, Avg Time:
{avg_time:.4f} seconds")
```

Main

The `main.py` file is a runnable instance of the complete sms simulation and uses the components from the Producer Model, Sender Model, and Display Monitor. The asynchronous operations of producing and sending messages are initiated and then awaited in the correct order so the system works cohesively. An additional stat display is stored locally to show the final statistics of the program.

```
async def main():
    start_time = time.time()
    queue = asyncio.Queue() # main datastructure to handle messages

    stats = {
        'sent': 0,
        'failed': 0,
        'total_time': 0.0
    }

    #initialize producer (s)
```

```
    producer = ProducerModel(queue)
    producer_task = asyncio.create_task(producer.produce_messages())

    sender_tasks = []
    #initialize senders
    for i in range(config.num_senders):
        sender = SenderModel(i, queue, stats)
        new_task = asyncio.create_task(sender.run())
        sender_tasks.append(new_task)

    #initialize monitor
    monitor_task = asyncio.create_task(monitor_progress(stats))

    #============================Await Async Tasks============================

    await producer_task
    await queue.join() #wait for senders to finish process all messages in queue
    for task in sender_tasks: #extra check to make sure that tasks have also
finished
        await task

    monitor_task.cancel() #manully cancel monitor task
    try:
        await monitor_task
    except asyncio.CancelledError:
        Pass
```

Concurrency and Scalability

Concurrency

- Asyncio: Utilizes Python's `asyncio` library to handle concurrency through cooperative multitasking.
    - Advantages:
        - Efficiently manages a large number of I/O-bound tasks without the overhead of threading.
        - Simplies the implementation of asynchronous operations like simulated delays.
    - Limitations:

        - Requires careful structuring of coroutines to avoid blocking.
        - Asyncio queue is shared across models and not fault-tolerant

Scalability Considerations

- Producer:
    - Generates messages at the start; scalability is **primarily limited by memory** for holding messages in the queue.
    - Can be modified to produce messages at any moment for even larger scales.
    - Multiple producers can be created to scale accordingly
- Senders:
    - The number of senders is configurable, allowing the system to scale horizontally based on the requirements.
    - With `asyncio`, a large number of senders can be managed efficiently, as long as sender tasks are I/O-bound.
- Queue:
    - `asyncio.Queue` can handle large numbers of messages, but it is centralized and managed better with message broker systems like Kafka

---

Unit Testing

Implemented using Python's `pytest` framework to ensure that individual components function as expected.

Fixtures are used to use repeated models or functions during unit testing

Mocks are used to set up default parameters for sender, producer, and display monitor models.

Tests include:

- Basic functionality
- Concurrent operation
- Performance
- Error handling
- Edge-cases
- State management

Tradeoffs and Considerations

Choice of Concurrency Model

- Asyncio vs Multithreading vs Multiprocessing:
    - Asyncio: efficient in handling I/O-bound tasks (like message processing) and simple to manage coroutines
    - Multithreading: difficult to use with Python's Global Interpreter Lock (GIL) and complexity of race conditions
    - Multiprocessing: Offers parallelism but requires many instances of Python due to GIL and is good for CPU-bound tasks
- Message Passing Mechanism
    - In-memory queue
        - Pros: Fast and simple to implement for low memory systems
        - Cons: Limited to single process, high coupling across different objects
    - External Message Broker (Kafka)
        - Pros: Supports distributed system, queue as a broker, more fault-tolerant and scalable
        - Cons: complex to develop and maintain
- Modularity
    - The system is designed with modularity in mind, as different functions are divided into their respective models, and they can be easily scaled, and managed and expanded to real SMS data and API calls.

- However, the queue is not treated as its own model, as it would with the python `aiokafka` library.

Conclusion

The scalable SMS alert simulation system is designed using Python and leverages asynchronous operations to manage efficiency, concurrency and scalability. Python's `asyncio` library allows efficient handling of multiple senders and high throughput message processing. The system's modularity using different producer and sender models allows for scalability and maintenance, while comprehensive unit testing ensures reliability. This implementation serves as a foundation for simulating SMS alerts and can be expanded to a real-world scenario.