Technical Documentation

SMS Alert Simulation System

Overview

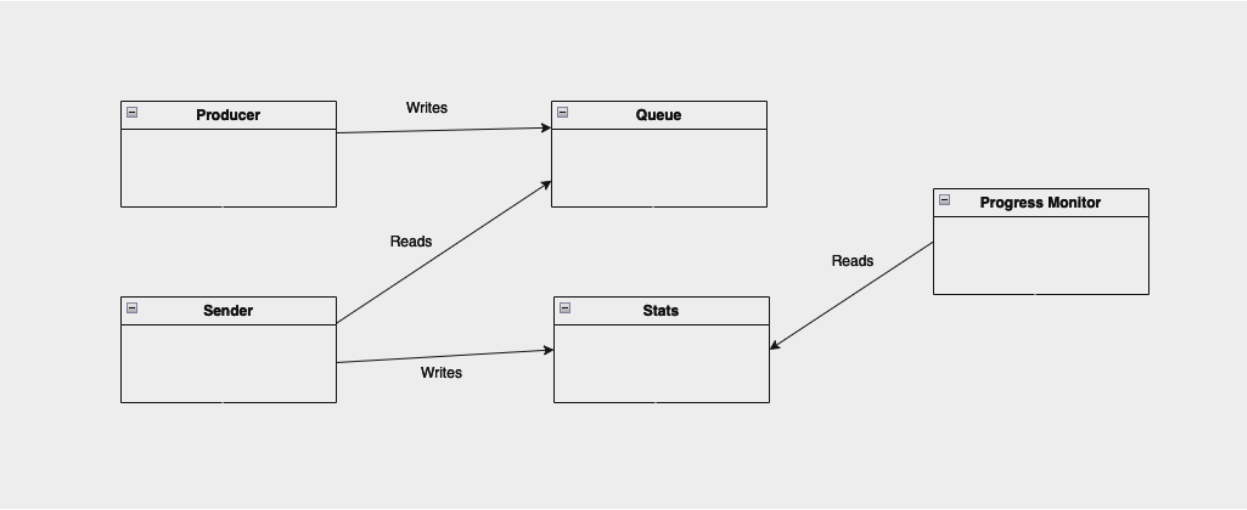
The SMS Alert simulation system is a scalable system for simulating SMS alert processing with configurable producer, senders, and monitoring. It is designed for concurrency and scalability. The system consists of 5 Major components:

- Producer, generates messages
- Sender, simulates sending the message
- Queue, shared object to store the messages
- Stats, shared object to store the simulation statistics
- Progress Monitor, displays the simulation progress

System Architecture

This assignment is implemented in Java and uses the concepts of Multithreading.

To briefly go over how these components connect with each other, here's a high level design of the system.



The figure above roughly represents the system.

- The Producer writes to a shared object between the Producer and Senders which is the Queue, and concurrently multiple instances of Senders read from the same Queue.
- The Senders simultaneously write to another shared object Stats which stores the current progress of the simulation.
- This Stats object is shared by the Senders and the Progress Monitor.
- The Progress Monitor periodically fetches the Stats to display to the user.

Components

There are 7 total components in my design which are Producer, Sender, ProgressMonitor, MessageAlertSim, MessageStats, BlockingMessageQueue and Message.

1. Class Producer

- Producer class implements IProducer and Runnable interfaces.
- It's constructor takes in a shared BlockingMessageQueue object and the number of messages it has to produce.
- Available Methods:
  - generateMessage(): It's functionality is to randomly generate message with length in the range of 1 to 100 both inclusive.
  - run(): This method generates the required number of message which are then added to the BlockingMessageQueue sequentially. After each addition there's a small delay of 10ms to simulate real world scenario since sending messages in real life require machine to machine communication which takes non trivial amount of time.
- There's only one thread of producer running throughout the simulation.

2. Class Sender

- Sender class implements ISender and Runnable interfaces.
- It's constructor takes in:

    - Shared BlockingMessageQueue instance
    - Failure rate, the probability with which a message should fail which ranges between 0.0 and 1.0, excluding 1.0 because it's highly unlikely in real world that all the messages supposed to be sent fail.
    - Mean Delay, the Average delay of all the messages, which is a non zero positive value
    - Shared instance of MessageStats
- Available Methods:
    - run():
        - It dequeues Message from the BlockingMessageQueue. If another Sender thread has the lock to the queue, this thread waits until it gets the lock.
        - Once the message is dequeued, it simulates sending by waiting for a random time in the range of 1 to 2 times the Mean Delay, this gives us a uniform distribution of delays across all the messages.
        - With a probability of specified failure rate, the message is flagged as failed and the completion time for the message is set.
        - Updates the Stats object and continues running until the flag is set to false.
    - stop(): Sets the running flag to false, to stop the thread.
- There are multiple sender threads running concurrently through out the simulation.

3. Class Progress Monitor

- ProgressMonitor implements IProgressMonitor and Runnable interface
- Constructor takes MessageStats and update interval
- Available Methods:
    - run(): Periodically prints current statistics
    - printStats(): Formats and displays current stats

    - stop(): Gracefully stops the monitor thread
- Single monitor thread runs throughout simulation

4. Class Message

- Message class represents an SMS message entity.
- Constructor takes in a string content for the message, initializes a UUID for the message and sets creationTime to object initialization time in milliseconds
- Fields:
    - content (String): The actual message content
    - creationTime (long): Time when message was created, which is set to message object initialization time in milliseconds
    - failed (boolean): Indicates if message processing failed
    - sentTime (long): Time when message processing completed in milliseconds
- Available Methods:
    - getters and setters for all fields
    - Message is immutable except for failed and sentTime status.
    - setSentTime() checks if the provided time is before the creationTime, if true throws an IllegalArgument exception

5. Class BlockingMessageQueue

- BlockingMessageQueue implements IMessageQueue, It's a wrapper class for Java's LinkedBlockingQueue which is thread-safe.
- Provides separate locking mechanism for adding and removing Messages
- This Queue's constructor takes in a capacity value which is provided as 2 times the number of senders in the simulation. This ensures that the Queue doesn't cause an OutOfMemory exception.
- Once the size limit is reached, the Producer thread gets blocked and waits until the Senders process the messages in the Queue. This makes the program

scaleable.

- Available Methods:
  - add(): Adds message to queue (blocks if full)
  - remove(): Removes and returns message (blocks if empty)
  - size(): Returns current queue size
  - isEmpty(): Checks if queue is empty

6. Class MessageStats

- MessageStats handles message statistics tracking
- Uses atomic operations for thread safety
- Available Methods:
  - incrementSent(): Increments successful message count
  - incrementFailed(): Increments failed message count
  - addProcessingTime(): Adds processing time for a message
  - getters for counts and average processing time
- All operations are thread-safe using AtomicInteger/AtomicLong

7. Class MessageAlertSim

- Main simulation orchestration class
- Constructor takes configuration parameters:
  - messageCount: Total messages to process
  - senderCount: Number of sender threads
  - failureRate: Probability of message failure
  - meanDelay: Average processing time
  - monitorInterval: Stats update frequency
- Available Methods:
  - go(): Initializes and starts all components

- initializeProducer/Senders/Monitor(): create the respective threads and starts them
- waitForCompletion(): Waits for simulation to finish
- shutdown(): Gracefully stops all the sender threads and the monitor thread
- getFinalStats(): Prints final simulation statistics on to the console

My Design

- My core design decision was for the data structure that would store the messages. I knew it should be some kind of a thread safe queue which blocks adding and removing operations separately. I came across two types of queues in java, BlockingLinkedQueue and ConcurrentLinkedQueue
  - BlockingLinkedQueue, uses locking mechanism but with 2 locks, one for adding operation and one for removing operation. Which is perfect for my scenario.
  - ConcurrentLinkedQueue, does not use locks uses compare and swap. but the problem is it's unbounded which may cause us to run into memory issues. And it does not guarantee ordered processing, which means that few messages in the beginning of the queue may be 'ignored' which will result in inconsistent processing times for messages.
  - I initially tried implementing my own thread safe queue but chose to replace it with BlockingLinkedQueue since it uses two locks and is more efficient with the locking mechanism.
- Added a max capacity to the queue, because we only have one producer, which doesnt compete with any other thread to put messages in the queue. If the rate of production of messages is faster than the senders' consumption rate, given a high number of messages we may run out of memory which will break the program or in general it's not scalable to have a queue with no size limit which can be unpredictable for various input combinations.
- Messages when created by the producer and added to the queue, their timer starts. It's similar to when user hits send, thats when we start the 'timer' to calculate the time taken to send the message.

- Messages fail with the ‘send’ delay, in order to simulate the real world setting where sending a message may consist of multiple steps and failure may occur at any of these steps.

Note

I have used claude.ai to help me think about test case scenarios for some of the classes and writing precise comments and javadocs. I used it to make sure that I don’t miss any testing scenarios. I used it to help me understand some of the requirements like the mean delay.

I referred to the following external resources to help me with the assignment:

- https://jenkov.com/tutorials/java-concurrency/index.html
- https://www.youtube.com/watch?v=WIdMTtUWqTg