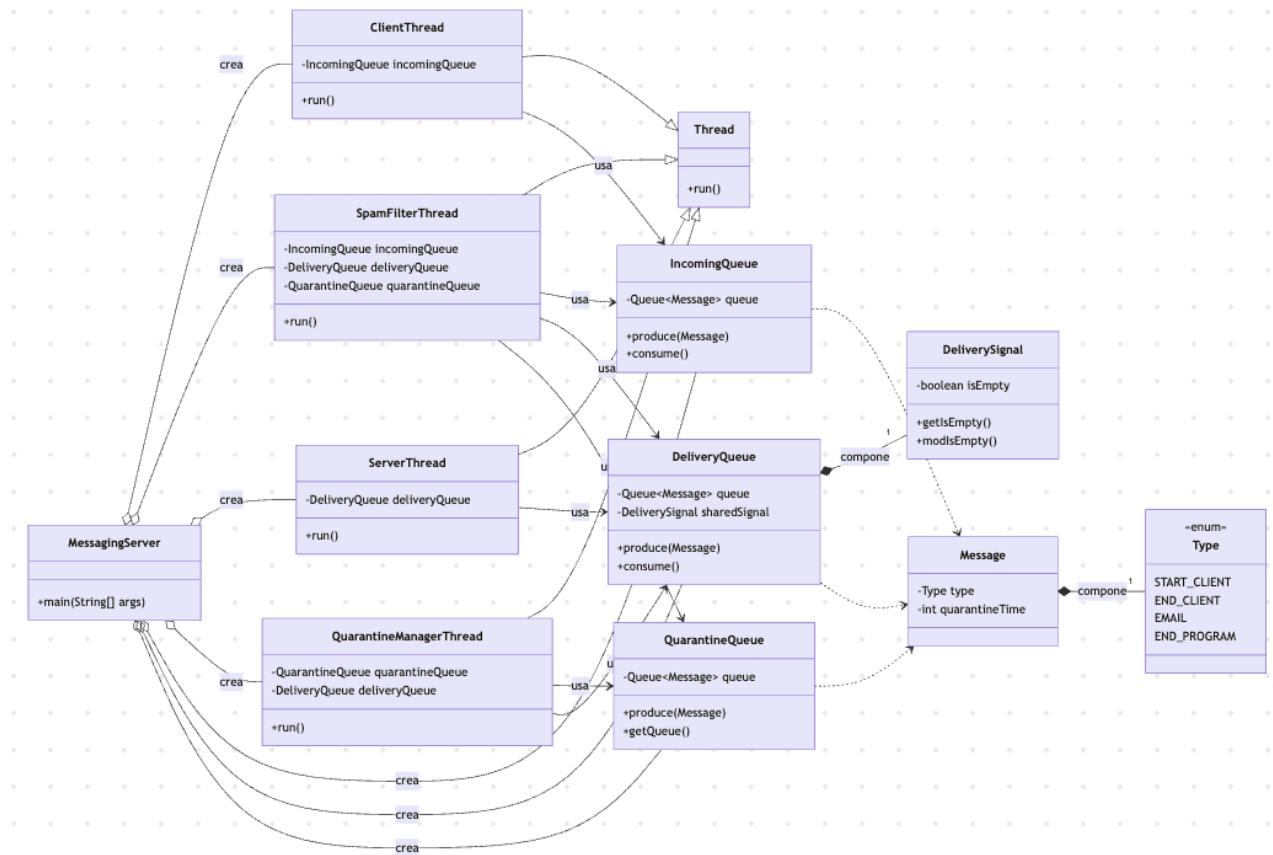


Simon Pineda 202410683

Daniel Colmenares 202410139

UML del programa:

Agregar otros atributos (capacity, statics, nums)



Funcionamiento general de la aplicación:

Como se muestra en el diagrama, la implementación del centro de mensajería se basa en threads (productores/consumidores) y colas (buffers compartidos) que interactúan entre ellos para comunicar información desde los clientes hasta los servidores de entrega final.

El flujo básico del programa es el siguiente:

- La clase `MessagingServer` (main) lee los parámetros y crea los respectivos threads clientes, filtros de spam y servidores de entrega y las colas de entrada, entrega y cuarentena. Adicionalmente se crea sólo un thread manejador de cuarentena. Todos los threads son empezados (start).
 - Los clientes producen sus mensajes y los envían a `IncomingQueue`. Los filtros de spam consumen estos mensajes, los analizan y los envían a `DeliveryQueue` o `QuarantineQueue`.
 - El manejador de cuarentena consume de `QuarantineQueue` y procesa los mensajes. Estos pueden ser descartados o enviados a `DeliveryQueue`.
 - Finalmente los servidores de entrega consumen los mensajes de `DeliveryQueue`.

Los filtros de spam llevan la cuenta de los mensajes de inicio y final de los clientes mediante variables estáticas. Una vez estas variables sean iguales, todos los filtros habrán procesado todos los mensajes y saben que no se producirán más. Solo uno de ellos envía un mensaje de terminación (`END_PROGRAM`) a `QuarantineQueue` y `DeliveryQueue` para que sus consumidores lo lean y puedan terminar.

En el caso de `DeliveryQueue`, cuando recibe `END_PROGRAM`, en un ciclo agrega una copia de este por cada servidor de entrega, por eso la cola debe saber cuántos servidores hay por medio de un atributo. Después de que todos los threads finalizan, el main imprime que los mensajes fueron enviados y el programa finaliza.

Interacciones entre clases:

- `ClientThread` (Productor) - `IncomingQueue` (Monitor):

Esta relación implementa el patrón de Productor-Consumidor con **espera pasiva** en un buffer limitado. El método `produce()` es sincronizado. Si la cola está llena (`maxSize == queue.size()`), el `ClientThread` llama a `wait()`, liberando el monitor y quedando bloqueado hasta que un filtro consuma un mensaje. Al terminar de producir, el cliente llama a `notify()` para despertar a un `SpamFilterThread` que esté esperando por mensajes.

- `SpamFilterThread` (Consumidor) - `IncomingQueue` (Monitor):

Este es el complemento de la relación anterior, que también implementa **espera pasiva**. El método `consume()` es sincronizado. Si la cola está vacía (`queue.isEmpty()`), el `SpamFilterThread` llama a `wait()`, liberando el monitor y quedando bloqueado hasta que un cliente produzca un mensaje. Al terminar de consumir, el filtro llama a `notify()` para despertar a un `ClientThread` que esté esperando por espacio disponible en la cola.

- `ServerThread` (Consumidor) - `DeliveryQueue` (Monitor):

A diferencia de las anteriores esta relación implementa **espera activa**, porque el caso así lo pide. El consumidor gira en un bucle mientras `sharedSignal.isEmpty()` sin ceder el procesador, consumiendo CPU hasta que la señal `DeliverySignal` es modificada por un productor. Una vez que la señal cambia, el hilo entra en un bloque `synchronized` separado para garantizar que la operación `queue.remove()` sea atómica y evitar condiciones de carrera.

- `SpamFilterThread` (Productor) - `DeliveryQueue` (Monitor):

Aquí se implementa una **espera semiactiva**. El `SpamFilterThread` comprueba si la cola está llena (`maxSize == queue.size()`) en un bucle. Si está llena, el hilo llama a `yield()`, cediendo voluntariamente su turno de CPU a otros hilos. Una vez que hay espacio (detectado en la siguiente iteración del bucle), entra en un bloque `synchronized` para añadir el mensaje de forma segura a la cola.

- `QuarantineManagerThread` (Productor) - `DeliveryQueue` (Monitor):

Esta relación es idéntica a la del `SpamFilterThread` con `DeliveryQueue`, utilizando **espera semiactiva**. El `QuarantineManagerThread`, al intentar mover un mensaje de cuarentena a entrega, comprueba si la cola de entrega está llena. Si lo está, llama `yield()` en un bucle hasta que se libere espacio, después entra a un bloque `synchronized` para añadir el mensaje.

- `SpamFilterThread` (Productor) - `QuarantineQueue` (Monitor):

Puesto que en el caso especifica que la cola de cuarentena tiene espacio ilimitado, no se requiere espera. La sincronización se limita a garantizar el **acceso mutuamente excluyente**. El otro método sincronizado es el `produce()` de `QuarantineQueue`, este método añade un elemento a la cola.

- `QuarantineManagerThread` (Consumidor) - `QuarantineQueue` (Monitor)

Esta es una sincronización que usa un **bloqueo de monitor externo**. El `QuarantineManagerThread` debe revisar toda la cola cada segundo. El mánager bloquea el monitor `quarantineQueue` usando un bloque `synchronized(quarantineQueue)` desde fuera de la clase. Una vez que posee el candado, puede invocar de forma segura a `getQueue().iterator()` para iterar sobre la colección, decrementar contadores y remover mensajes con `it.remove()`.

- `SpamFilterThread` (Hilos) - `SpamFilterThread` (Hilos):

Los hilos `SpamFilterThread` usan dos monitores (`Object`) estáticos para coordinarse entre sí. El `lock` protege los contadores compartidos (asegurando que el conteo de los mensajes de inicio y fin sea atómico y sin errores). El `endLock` asegura que solo **un** filtro envíe el mensaje final `END_PROGRAM`, evitando envíos duplicados.

Validación del programa:

Para verificar que la implementación cumple con todos los requisitos funcionales y de concurrencia, se diseñaron y ejecutaron múltiples pruebas. La validación se basó en el análisis de las trazas de consola (`System.out.println`) generadas durante la ejecución del programa.

```

● simonpine@Simons-MacBook-Pro-4 Caso3 % cd /Users/simonpine/Desktop/ProyectosActuales/Caso3 ; /usr/bin/env /Library/Java/JavaVirtualMachines/jdk-23.jdk/Contents/Home/bin/java -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/simonpine/Library/Application Support/Code/User/workspaceStorage/6de860e5ac57aa53152551d3913a94e/redhat.java /jdt_ws/Caso3_4ccf7762/bin MessagingServer
[Client-0]: Produced: Client-0-START:[Client-0-START]
[Filter-0]: Consumed: Client-0-START:[]
[Client-2]: Produced: Client-2-START:[Client-2-START]
[Client-3]: Produced: Client-3-START:[Client-2-START, Client-3-START]
[Client-4]: Produced: Client-4-START:[Client-2-START, Client-3-START, Client-4-START]
[Client-1]: Produced: Client-1-START:[Client-2-START, Client-3-START, Client-4-START, Client-1-START]
[Filter-1]: Consumed: Client-2-START:[Client-3-START, Client-4-START, Client-1-START]
[Client-3]: Produced: Client-3-0:[Client-3-START, Client-4-START, Client-1-START, Client-3-0]
[Client-0]: Produced: Client-0-0:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0]
[Client-1]: Produced: Client-1-0:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0]
[Client-4]: Produced: Client-4-0:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0]
[Client-2]: Produced: Client-2-0:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0]
[Client-4]: Produced: Client-4-1:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1]
[Client-3]: Produced: Client-3-1:[Client-3-START, Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1, Client-3-1]
[Filter-0]: Consumed: Client-3-START:[Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1, Client-3-1]
[Client-1]: Produced: Client-1-1:[Client-4-START, Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1, Client-3-1, Client-1-1]
[Filter-1]: Consumed: Client-4-START:[Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1, Client-3-1, Client-1-1]
[Client-0]: Produced: Client-0-1:[Client-1-START, Client-3-0, Client-0-0, Client-1-0, Client-4-0, Client-2-0, Client-4-1, Client-3-1, Client-1-1, Client-0-1]

```

Se modificó el archivo `parameters.txt` con múltiples configuraciones. Las trazas de consola confirman que el número correcto de hilos `Client`, `Filter` y `Server` se inician correctamente. Se observa cómo los clientes (`Client-0`, `Client-2`, etc.) y filtros (`Filter-0`, `Filter-1`) comienzan a interactuar con el `IncomingQueue` de inmediato, validando el arranque del sistema.

```

[Filter-0]: Produced: Client-4-2:[Client-4-2]
[Filter-1]: Consumed: Client-2-END:[Client-4-3, Client-4-4, Client-4-5, Client-4-6, Client-4-7, Client-4-8]
[Server-0]: Consumed: Client-4-2:[]
[Client-4]: Produced: Client-4-9:[Client-4-3, Client-4-4, Client-4-5, Client-4-6, Client-4-7, Client-4-8, Client-4-9]
[Client-4]: Produced: Client-4-END:[Client-4-3, Client-4-4, Client-4-5, Client-4-6, Client-4-7, Client-4-8, Client-4-9, Client-4-END]
Client-4 FINISHED
[Filter-1]: Consumed: Client-4-3:[Client-4-4, Client-4-5, Client-4-6, Client-4-7, Client-4-8, Client-4-9, Client-4-END]
[Filter-1]: Produced: Client-4-3:[Client-4-3]
[Server-0]: Consumed: Client-4-3:[]
[Filter-0]: Consumed: Client-4-4:[Client-4-5, Client-4-6, Client-4-7, Client-4-8, Client-4-9, Client-4-END]
[Filter-0]: Produced: Client-4-4:
[Filter-0]: Consumed: Client-4-5:[Client-4-6, Client-4-7, Client-4-8, Client-4-9, Client-4-END]
[Filter-0]: Produced: Client-4-5:[Client-4-5]
[Server-1]: Consumed: Client-4-5:[]
[Filter-1]: Consumed: Client-4-6:[Client-4-7, Client-4-8, Client-4-9, Client-4-END]
[Filter-1]: Produced: Client-4-6:[Client-4-6]
[Server-0]: Consumed: Client-4-6:[]
[Filter-0]: Consumed: Client-4-7:[Client-4-8, Client-4-9, Client-4-END]
[Filter-0]: Produced: Client-4-7:
[Filter-1]: Consumed: Client-4-8:[Client-4-9, Client-4-END]
[Filter-1]: Produced: Client-4-8:
[Filter-0]: Consumed: Client-4-9:[Client-4-END]
[Filter-0]: Produced: Client-4-9:

```

Se realizó un seguimiento visual del flujo de mensajes a través de las trazas.

- Producción de Clientes: Se verificó que los clientes generan sus mensajes en el orden correcto (`START`, `EMAIL`, `END`). Por ejemplo en la imagen anterior, se observa cómo `Client-4` produce `Client-4-9` y, posteriormente, `Client-4-END`.
- Procesamiento de Filtros: Se constató que los filtros consumen mensajes del `IncomingQueue` y los enrutan correctamente a `DeliveryQueue` (mensajes válidos) o `QuarantineQueue` (mensajes spam).

```

[manager]: Produced: Client-0-4:[Client-0-4]
[Server-1]: Consumed: Client-0-4:[]
[Manager]: Discarded: Client-2-4
[Manager]: Discarded: Client-4-9
[Manager]: Produced: Client-3-1:[Client-3-1]
[Server-1]: Consumed: Client-3-1:[]
[Manager]: Produced: Client-1-7:[Client-1-7]
[Manager]: Produced: Client-3-9:[Client-1-7, Client-3-9]
[Server-0]: Consumed: Client-1-7:[Client-3-9]
[Server-1]: Consumed: Client-3-9:[]
[Manager]: Produced: Client-3-0:[Client-3-0]
[Server-1]: Consumed: Client-3-0:[]
[Manager]: Produced: Client-1-1:[Client-1-1]
[Server-0]: Consumed: Client-1-1:[]
[Manager]: Produced: Client-0-1:[Client-0-1]
[Server-1]: Consumed: Client-0-1:[]
[Manager]: Discarded: Client-0-3
[Manager]: Produced: Client-0-6:[Client-0-6]
[Manager]: Discarded: Client-1-6
[Server-0]: Consumed: Client-0-6:[]
[Manager]: Produced: Client-2-5:[Client-2-5]
[Server-0]: Consumed: Client-2-5:[]
[Manager]: Produced: Client-4-8:[Client-4-8]
[Server-1]: Consumed: Client-4-8:[]
[Filter-1]: Produced: END:
[Filter-1]: Produced: END:[END]
[Server-0]: Consumed: END:[END]
Filter-1 FINISHED
Filter-0 FINISHED
Server-0 FINISHED
[Server-1]: Consumed: END:[]
Server-1 FINISHED
Manager FINISHED
Messages sent

```

Se validaron las dos funciones principales del manejador analizando las trazas como en la imagen anterior:

- **Procesamiento de Cuarentena:** El `Manager` solo comienza a producir mensajes (`[Manager]: Produced...`) hacia el final de la simulación. Esto confirma que la lógica de `Thread.sleep(1000)` y el decrecimiento de contadores funciona, reteniendo los mensajes durante el tiempo de cuarentena especificado (10-20 segundos).
- **Descarte de Mensajes:** En la misma imagen, se observan líneas como `[Manager]: Discarded: Client-2-4`, lo que valida que la lógica de descarte (múltiplo de 7).

Prueba de estrés:

- Detección de `ConcurrentModificationException`: Al ejecutar con alta carga, se detectó una `ConcurrentModificationException` en `DeliveryQueue`. Esto ocurría porque un productor modifica la cola (`.add()`) mientras un consumidor (`ServerThread`) la iteraba para el `println`. Se corrigió moviendo los bucles de espera (activa y semiactiva) fuera de los bloques `synchronized`.
- Detección de Condición de Carrera en Terminación: Se observó que los `SpamFilterThread` terminaban prematuramente. Se corrigió la lógica de terminación para asegurar que un filtro solo termina cuando todos los clientes han finalizado (`allClientsFinished`) y el `IncomingQueue` está completamente vacío.

Con la siguiente configuración se generó la última prueba de estrés, después de haber solucionado los problemas encontrados. Este último resultado se guardó en el archivo estrés.txt, ahí es posible apreciar el correcto funcionamiento del programa.

```
ces > ≡ parameters.txt
```

```
20
```

```
100
```

```
2
```

```
5
```

```
5
```