Caso 3|Sistema de mensajería con filtros y cuarentena

Jeronimo López — 202320969 (j.lopezm234@uniandes.edu.co)
Daniel Espitia — 201716251 (da.espitiab@uniandes.edu.co)

¿Qué construimos?

Implementamos una tubería concurrente con cuatro etapas: generadores de mensajes (clientes), filtros de spam, cuarentena con retardo y servidores de entrega. Los clientes publican en un búfer de *entrada* con capacidad fija; los filtros consumen, clasifican y envían a *entrega* o a *cuarentena*; el manejador libera desde cuarentena cuando se cumple el tiempo; los servidores consumen de *entrega*. El cierre usa mensajes `FIN` y un **Coordinador** que evita terminar antes de tiempo.

Clases y rol de cada una de ellas

Mensaje. Campos: `tipo` (`INICIO`, `DATO`, `FIN`), `idCliente`, `secuencial`, `esSpam`. El `toString()` facilitó leer los logs.

BuzonLimitado. Monitor con cola acotada. Dos estilos de acceso: (i) `ponerBloqueante`/`tomarBloqueante` con `wait()`/`notifyAll()` y (ii) `intentarPoner`/`intentarTomar` que devuelven `false`/`null`; en ese caso el hilo cede CPU con `Thread.yield()` y reintenta.

BuzonCuarentena. Guarda pares *(mensaje, segundosRestantes)*. Cada segundo, `ticYRecolectarListos` reduce contadores y devuelve los que ya cumplieron el tiempo. Para simular mensajes maliciosos usamos un número uniforme de 1 a 21; si es múltiplo de 7 se descarta.

ClienteEmisor. Publica `INICIO`, luego `DATO` (algunos marcados como spam al azar) y al final `FIN`. Actualiza contadores en el **Coordinador**.

FiltroSpam. Consumidor pasivo de *entrada*. Si el mensaje es spam, lo pone en cuarentena con retardo aleatorio (10–20 s); si no, lo envía a *entrega* en modo semi–activo. Cuando no quedan pendientes y todos los clientes terminaron, pide permiso al **Coordinador** para emitir `FIN` a *cuarentena* y luego a *entrega*.

ManejadorCuarentena. Hilo temporizado: `sleep(1000)` y luego `ticYRecolectarListos`. Reinyecta los listos a *entrega*. Termina cuando vio `FIN` y la lista quedó vacía.

ServidorEntrega. Consumidor semi–activo de *entrega*. Simula latencia de 0.5–2.0 s por mensaje y termina al recibir `FIN`.

Coordinador. Métodos `synchronized` para contadores (`iniciosVistos`, `finesVistos`) y banderas que evitan emitir `FIN` dos veces. Expone `puedeEnviarFinEntrega()` y `puedeEnviarFinCuarentena()`.

Principal. Crea hilos, hace `join()` y al final agrega tantos `FIN` extra como servidores para que todos vean la señal.

Diagrama de clases. Usamos el archivo `Caso3UML.pdf`.

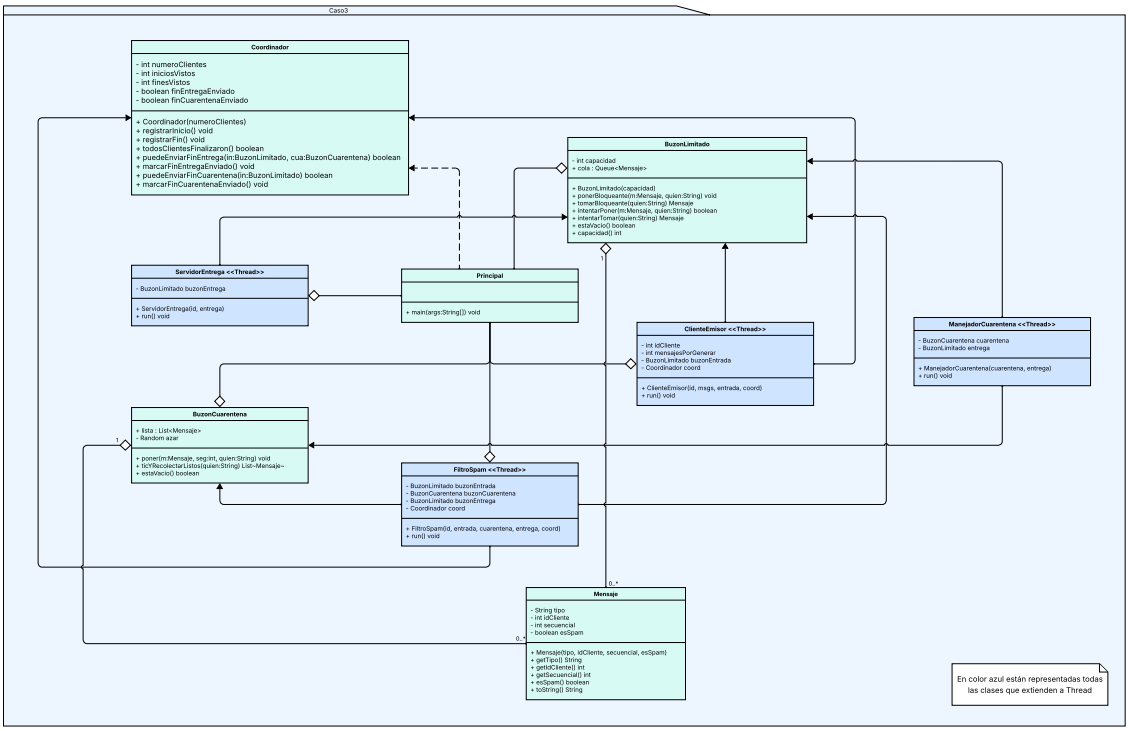

Figura 1: Diagrama de clases (métodos relevantes de sincronización).

Sincronización por parejas

Pareja	Tipo	Mecánica exacta
Cliente → Buzón de entrada	Pasiva	`ponerBloqueante`: si la cola está llena, el hilo hace `wait()`; al insertar se invoca `notifyAll()`. La cola nunca supera la capacidad.
Filtro ← Buzón de entrada	Pasiva	`tomarBloqueante`: si la cola está vacía, el hilo hace `wait()`; al retirar se llama `notifyAll()` para despertar a productores.
Filtro → Buzón de entrega	Semi–activa	`intentarPoner`: si el búfer está lleno, devuelve `false`; el hilo cede CPU con `Thread.yield()` y reintenta.
Servidor ← Buzón de entrega	Semi–activa	`intentarTomar`: si está vacío, retorna `null`; el hilo cede CPU con `yield()` hasta que aparezcan datos.
Filtro → Cuarentena	Monitor	Inserción sincronizada del par *(mensaje, segundos)*; el estado interno queda protegido por el monitor.
Manejador ↔ Cuarentena	Temporizado	El manejador duerme 1000 ms y luego invoca `ticYRecolectarListos`. Los mensajes listos van a salida; si el número aleatorio es múltiplo de 7, se descartan como "maliciosos".
Manejador → Entrega	Semi–activa	`intentarPoner` con reintentos y `yield()` si el búfer de salida está lleno.
Filtros ↔ Coordinador	Exclusión	Lectura/actualización de contadores y banderas mediante métodos `synchronized` (`puedeEnviarFinEntrega` y `puedeEnviarFinCuarentena`).
Principal → Servidores	Cierre	Se agregan `FIN` extra (uno por servidor) para garantizar que todos los consumidores observen la señal de término.

¿Cómo es el proceso?

1. Los clientes publican `INICIO` y sus `DATO`. Si *entrada* se llena, quedan en `wait()` hasta que un filtro retire elementos.
2. Los filtros consumen de *entrada*. Si no es spam, pasan a *entrega* con `intentarPoner`; si es spam, lo envían a *cuarentena* con retardo aleatorio.
3. El manejador descuenta segundos y libera los listos a *entrega*. Algunos se descartan por la regla del múltiplo de 7.
4. Los servidores consumen de *entrega* con `intentarTomar`, simulan latencia y reportan la entrega.
5. Con todos los clientes finalizados y *entrada* vacía, un filtro autorizado por el **Coordinador** emite `FIN` a *cuarentena* y, cuando esta queda vacía, `FIN` a *entrega*.

6. **Principal** añade #servidores `FIN` extra para que ninguno quede esperando.

Validación con tres corridas

Guardamos los insumos en `Pruebas/` (`prueba1.txt`, `prueba2.txt`, `prueba3.txt`) y las salidas `salida_programaX.txt`.

Programa 1 (3, 4, 2, 2, 5, 3). Caso base. Los clientes muestran "espera: búfer lleno" cuando procede; los servidores consumen con `(tryTake)`; el cierre usa exactamente dos `FIN`. No hubo descartes desde cuarentena.

Programa 2 (4, 10, 3, 3, 5, 3). Más tráfico y varios descartes (múltiplos de 7, 14 y 21). El Coordinador habilita `FIN` hacia cuarentena y luego a entrega en el orden correcto. Al final **Principal** agrega tres `FIN` y todos terminan.

Programa 3 (5, 20, 5, 5, 5, 5). Prueba de carga: muchas liberaciones desde cuarentena, descartes frecuentes y cinco `FIN` finales correctamente consumidos. No observamos bloqueos ni inanición en *entrega*.

Riesgos y cómo los evitamos

- **Señal de fin perdida**. Se agregan `FIN` extra (= número de servidores).
- **Carreras en contadores/banderas**. El **Coordinador** expone solo métodos `synchronized`.
- **Consumo de CPU**. En *entrega* preferimos espera semi–activa con `yield()`.

Estructura del repositorio y ejecución

Estructura usada:

```
Caso3/
  BuzonCuarentena.java
  BuzonLimitado.java
  ClienteEmisor.java
  Coordinador.java
  FiltroSpam.java
  ManejadorCuarentena.java
  Mensaje.java
  Principal.java
  ServidorEntrega.java
  configuracion.txt
```

```
Enunciado/
  2025-2-1311-Caso3_v2.pdf

Informes/
  Caso3UML.pdf

Pruebas/
  prueba1.txt
  prueba2.txt
  prueba3.txt

README.md
```

Comandos (la app lee `Caso3TIC/Caso3/configuracion.txt`, por eso copiamos cada prueba allí antes de correr):

```
# Compilar
javac -d out -sourcepath . Caso3/*.java

# Crear ruta esperada por Principal
mkdir -p Caso3TIC/Caso3

# Programa 1
cp Pruebas/prueba1.txt Caso3TIC/Caso3/configuracion.txt
java -cp out Caso3.Principal > Pruebas/salida_programa1.txt

# Programa 2
cp Pruebas/prueba2.txt Caso3TIC/Caso3/configuracion.txt
java -cp out Caso3.Principal > Pruebas/salida_programa2.txt

# Programa 3
cp Pruebas/prueba3.txt Caso3TIC/Caso3/configuracion.txt
java -cp out Caso3.Principal > Pruebas/salida_programa3.txt
```