

Caso 3 - Concurrency y Sincronización de Procesos

Grupo 10

Sergio Soler Garzon - 202310103

Daniel Diab: 202321332

Tecnología e Infraestructura Computacional

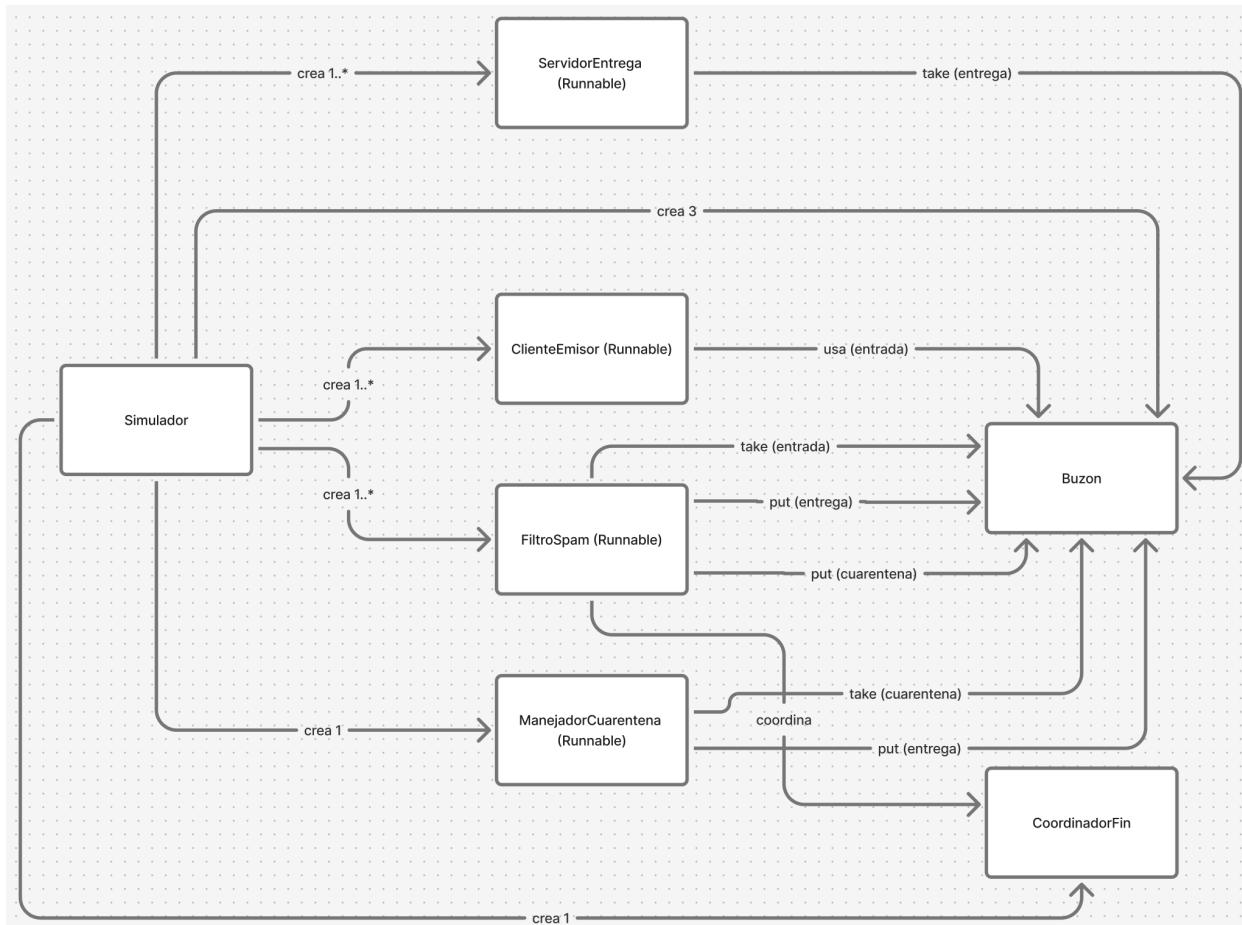
Ricardo Gómez

**Universidad de los Andes
Ingeniería de Sistemas y Computación
Bogotá D.C.
2025**

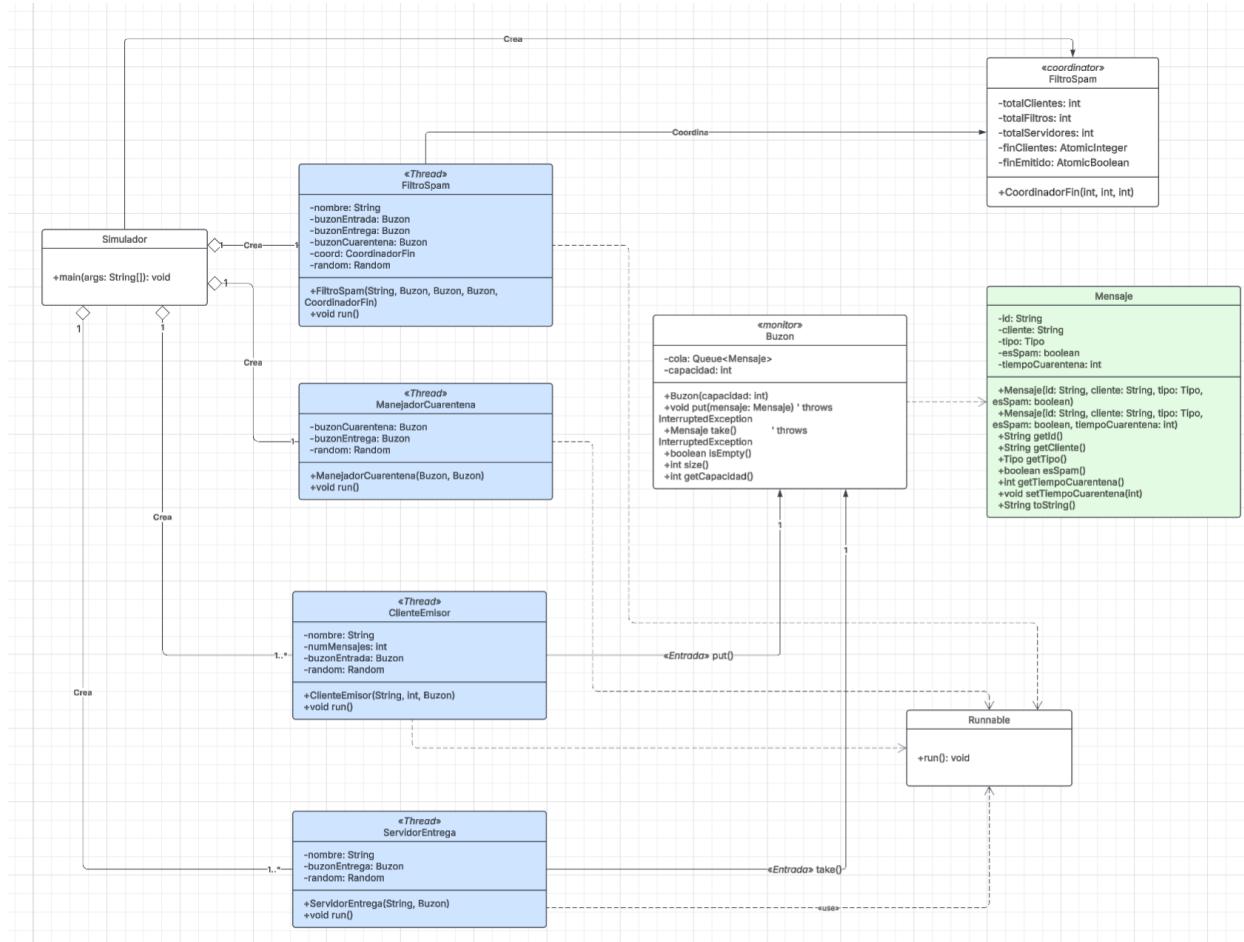
Diseño general

El modelo implementado está compuesto por siete clases principales: Mensaje, Buzon, ClienteEmisor, FiltroSpam (con su CoordinadorFin interno), ManejadorCuarentena, ServidorEntrega y Simulador. Cada clase cumple una función específica dentro del flujo de mensajes del sistema y refleja un rol en el modelo productor-consumidor. Por otro lado, el Simulador es el encargado de crear y coordinar las instancias necesarias para ejecutar la simulación. Al inicio, crea tres buzones compartidos: uno para la entrada, otro para la entrega y un tercero para la cuarentena, además de un objeto CoordinadorFin que controla el momento en que se debe finalizar todo el sistema. Cada ClienteEmisor mantiene una referencia al buzón de entrada, cada FiltroSpam accede a los tres buzones y al coordinador, el ManejadorCuarentena trabaja directamente con los buzones de cuarentena y entrega, y cada ServidorEntrega interactúa únicamente con el buzón de entrega.

Estas relaciones pueden describirse como asociaciones dentro de un diagrama de clases:



Uml:



- Simulador crea tres instancias de Buzon (entrada, entrega y cuarentena) y un CoordinadorFin.
- Cada ClienteEmisor tiene acceso al buzón de entrada.
- Cada FiltroSpam accede a los tres buzones y al coordinador.
- El ManejadorCuarentena utiliza los buzones de cuarentena y entrega.
- Cada ServidorEntrega toma mensajes del buzón de entrega.

En el diseño, las relaciones son de composición (porque el simulador crea las instancias) y de asociación (porque varios hilos comparten los mismos buzones). La clase Mensaje encapsula los datos de cada mensaje (id, cliente, tipo y estado de spam), mientras que Buzon implementa una cola bloqueante con mecanismos de sincronización (synchronized, wait, notifyAll) para coordinar productores y consumidores. Finalmente, Simulador lee los parámetros desde el archivo config.txt, crea los hilos, ejecuta toda la secuencia y al final utiliza join() para asegurar que todos los procesos terminen correctamente.

Funcionamiento global

Al iniciar la simulación, cada cliente emisor envía primero un mensaje de tipo INICIO, luego produce varios mensajes NORMAL (algunos pueden marcarse como spam de forma aleatoria) y finaliza con un mensaje FIN. Todos estos mensajes son depositados en el buzón de entrada. Además, los filtros de spam consumen mensajes de ese buzón. Si el mensaje es válido, lo envían al buzón de entrega; si es spam, lo colocan en el buzón de cuarentena con un tiempo aleatorio entre 10 y 20 segundos. Cada filtro lleva además un conteo de los mensajes FIN de los clientes. Cuando todos los clientes han terminado, el primer filtro que detecta la condición global se encarga de emitir el FIN global, enviando un mensaje FIN a los servidores, al manejador y a los demás filtros para garantizar una finalización ordenada.

Por otro lado, el manejador de cuarentena revisa constantemente los mensajes en cuarentena. Cada segundo reduce el contador de espera de cada mensaje y, cuando el tiempo llega a cero, decide si el mensaje se descarta o se libera al buzón de entrega. Este proceso continúa hasta que el manejador recibe un mensaje FIN, momento en el cual vacía su lista interna y termina. Por su parte, los servidores de entrega consumen del buzón de entrega, simulan el procesamiento de cada mensaje y finalizan cuando reciben un FIN. Al cierre, todos los hilos imprimen su mensaje de terminación y el Simulador informa que la simulación concluyó satisfactoriamente. De esta manera se logra una comunicación ordenada y sincronizada entre productores, filtros, manejadores y consumidores.

Sincronización por parejas

La sincronización entre las distintas clases se implementa mediante monitores internos de cada buzón, usando synchronized, wait() y notifyAll() para asegurar el acceso exclusivo y la coordinación entre hilos. Las principales interacciones son las siguientes:

- Cliente ↔ Buzón de entrada: El método put() del buzón es bloqueante y utiliza wait/notifyAll para que los clientes esperen de forma pasiva cuando la cola está llena.
- Filtro ↔ Buzón de entrada: Los filtros utilizan take() bloqueante, que impone un consumo ordenado y despierta a los productores cuando libera espacio.
- Filtro ↔ Buzón de entrega: El método put() se usa también de forma bloqueante, coordinando la disponibilidad de espacio y notificando a los servidores cuando hay nuevos mensajes.
- Filtro ↔ Buzón de cuarentena: Usa put() bloqueante (aunque la capacidad sea teóricamente ilimitada) para colocar mensajes de spam junto con su tiempo de retención.
- Manejador ↔ Buzón de cuarentena: Combina el uso del método sincronizado isEmpty() y take() para evitar bloqueos prolongados y procesar mensajes en lotes.
- Manejador ↔ Buzón de entrega: Al liberar mensajes, utiliza put() para despertarlos servidores que estaban esperando.
- Servidor ↔ Buzón de entrega: Emplea take() bloqueante hasta que recibe mensajes o un FIN.

- Filtros ↔ CoordinadorFin: Se usa AtomicInteger y AtomicBoolean para incrementar el contador y definir de manera segura qué filtro debe emitir el FIN global, evitando condiciones de carrera.

Filtros entre sí: Cooperan indirectamente mediante los buzones y los mensajes FIN que inyectan, asegurando que todos terminen sin quedar bloqueados.

Con este esquema, cada recurso compartido (los tres buzones y el coordinador) actúa como su propio monitor, evitando accesos simultáneos no controlados y garantizando la correcta sincronización entre los actores del sistema.

Validación y pruebas

Para comprobar el funcionamiento del sistema se realizaron varias pruebas con diferentes configuraciones, todas controladas desde el archivo config.txt.

En la configuración base con clientes = 3, mensajes = 5, filtros = 2, servidores = 2, capEntrada = 10 y capEntrega = 10, se verificó que cada hilo generara y procesara los mensajes en el orden correcto. Todos los buzones se vaciaron al finalizar y el simulador imprimió el mensaje de cierre sin hilos pendientes, en la prueba de saturación de entrada, se redujo capEntrada a 2 con tres clientes simultáneos. Se observó que los productores quedaban bloqueados cuando la cola se llenaba y se reanudaban tan pronto un filtro liberaba espacio, confirmando la correcta espera pasiva, en la prueba de saturación de entrega, se redujo capEntrega también a 2 y se generaron múltiples mensajes válidos. Los filtros esperaron al tener la cola llena y los servidores consumieron en orden, validando el comportamiento productor-consumidor.

Para una prueba de cuarentena intensiva, se aumentó la probabilidad de spam, lo que llenó el buzón de cuarentena. El manejador redujo los tiempos de los mensajes cada segundo, descartando aproximadamente 1 de cada 7 y liberando los demás hacia entrega, mostrando un comportamiento semi-activo y coherente con el diseño. Por último, en la prueba de finalización global, se contó el número de mensajes FIN emitidos y se comprobó que cada hilo imprimía su aviso de cierre. Con esto se confirmó que el CoordinadorFin evita hilos colgados y que la simulación termina siempre de manera controlada.

Resultado consola:

Observaciones

Durante las pruebas se identificó que el mensaje FIN global se emite tan pronto finaliza el último cliente, sin verificar si quedan mensajes pendientes en los buzones de entrada o cuarentena. Aunque esto no impide el funcionamiento correcto, no cumple del todo con la condición establecida en el enunciado, por lo que sería ideal agregar una verificación adicional antes de emitir el cierre. También se notó que los servidores usan una espera pasiva al consumir del buzón de entrega. Según la especificación, deberían aplicar una espera activa con sondeo periódico, pero esta alternativa aumentaría el uso de CPU. La decisión de mantener la espera pasiva se considera más eficiente y estable para la simulación.