



UNIVERSIDAD NACIONAL DE CÓRDOBA
FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES
PROGRAMACIÓN CONCURRENTES

TRABAJO PRÁCTICO N° 1

"Simulación concurrente del flujo de pedidos en un centro logístico"

Grupo: Fairness con coca

Alumnos:

García, Lautaro Misael	44.192.548
Martinez, Facundo Jesús	40.681.858
Renaudo Gaggioli, Valentino	45.379.952
Sánchez, Leonardo Ariel	38.002.962

versión: 1.1.0

Mayo / 2025

Índice

1. Introducción.....	4
1.1 Objetivo del trabajo práctico.....	4
1.2 Descripción general del problema a resolver.....	4
1.3 Enfoque adoptado para la resolución.....	4
2. Desarrollo.....	6
2.1 Modelado y Diseño.....	6
2.2 Implementación.....	7
2.3 Herramientas Utilizadas.....	8
3. Plan de Versiones en el Repositorio.....	9
4. Resultados y análisis.....	11
4.1 Comparación de tiempo de ejecución con una sola instancia por etapa.....	12
5. Decisiones de diseño.....	13
6. Conclusiones.....	14
7. Anexo: Historial de cambios.....	15

1. Introducción

1.1 Objetivo del trabajo práctico

Este trabajo práctico tiene como objetivo principal afianzar los conocimientos adquiridos sobre programación concurrente, fundamentales en el desarrollo de sistemas que requieren la ejecución simultánea de múltiples tareas. A través de la implementación de procesos que operan en paralelo, se busca comprender y aplicar conceptos clave como hilos de ejecución, sincronización, condiciones de carrera, estructuras de datos seguras para múltiples hilos, y el manejo de recursos compartidos.

1.2 Descripción general del problema a resolver

El problema planteado consiste en simular el circuito logístico de gestión de pedidos dentro de una empresa de distribución. La tarea implica modelar los distintos procesos involucrados de manera que puedan funcionar en paralelo, representando así un sistema realista donde múltiples pedidos son procesados simultáneamente. Estos procesos deben interactuar entre sí mediante estructuras que garanticen la integridad de los datos y eviten errores comunes en entornos concurrentes. El sistema también debe ser capaz de registrar el estado de los pedidos y producir logs con fines estadísticos.

1.3 Enfoque adoptado para la resolución

Para abordar este problema, se diseñó una arquitectura en la que cada proceso del circuito logístico es representado por uno o más hilos independientes. Los pedidos fluyen a través del sistema mediante una serie de colas que conectan las diferentes etapas del proceso y permiten una comunicación segura entre hilos. Cada clase responsable de un rol logístico (por ejemplo, Preparador, Despachador, Entregador, Verificador) opera sobre estas colas, consumiendo y produciendo pedidos según reglas específicas. Este enfoque modular y concurrente facilita la escalabilidad del sistema y permite simular con mayor precisión las condiciones del mundo real en un entorno controlado y extensible.

2. Desarrollo

2.1 Modelado y Diseño

El sistema fue modelado como un conjunto de etapas que representan el ciclo de vida de un pedido dentro del centro de almacenamiento: preparación, despacho, entrega y verificación. Cada etapa es gestionada por una clase específica (Preparador, Despachador, Entregador y Verificador), que extiende una clase base llamada Distribuidor. Estas clases funcionan como hilos independientes que procesan pedidos de forma concurrente. Los pedidos se trasladan entre etapas a través de colas compartidas, que actúan como buffers entre hilos. Además, se utiliza una clase llamada CentroDeAlmacenamiento, que contiene una matriz de casilleros, donde se almacenan los pedidos durante el proceso. Cada pedido puede tener éxito o fallar en una etapa, simulando condiciones reales mediante probabilidades de fallo. El sistema también registra estadísticas sobre el proceso y el estado de los casilleros, lo que permite evaluar el desempeño general.

El diseño del sistema se basó en una estructura modular, orientada a objetos, que permite representar de forma clara cada rol dentro del proceso de distribución de pedidos. Las clases principales fueron organizadas para reflejar tanto la lógica funcional como las necesidades de la programación concurrente:

La clase Pedido representa la unidad básica del sistema. Contiene atributos como su estado actual y su posición dentro del centro de almacenamiento.

Las clases Despachador, Entregador y Verificador extienden a la clase abstracta Distribuidor, la cual encapsula el comportamiento común de los hilos procesadores. Este diseño permite el uso de herencia para reducir la duplicación de código y facilitar la extensión del sistema.

Distribuidor implementa Runnable y gestiona la lógica de procesamiento: extracción de pedidos de la cola de origen, simulación de éxito o fallo, y envío a las colas correspondientes (exitosos o fallidos).

Preparador implementa Runnable y representa uno de los hilos encargados de tomar pedidos nuevos, obtener un casillero disponible desde el CentroDeAlmacenamiento, asignarlo al pedido, y pasarlo a la cola de

pedidos preparados. Accede a las colas compartidas (pedidosNuevos y pedidosPreparados). La extracción del pedido se realiza de forma aleatoria para simular desorden en el procesamiento. En caso de no haber casilleros disponibles, espera brevemente antes de reintentar. El tiempo de procesamiento se simula con una pausa controlada por una distribución gaussiana.

CentroDeAlmacenamiento modela una matriz de casilleros donde se puede ubicar cada pedido. Esta clase se accede de manera sincronizada para garantizar que dos hilos no asignen el mismo casillero al mismo tiempo.

Este enfoque permite un flujo de trabajo concurrente ordenado, donde cada etapa puede trabajar en paralelo sin interferencias, y se asegura la integridad de los datos mediante sincronización explícita y el uso de clases concurrentes.

2.2 Implementación

Main: Clase principal que se encarga de inicializar el entorno, generar los pedidos y lanzar los hilos correspondientes. Es el núcleo organizador de la simulación. Registra en archivos .txt y .csv el estado del sistema cada 200 ms.

CentroDeAlmacenamiento: Administra el conjunto de casilleros mediante una matriz. Provee casilleros vacíos aleatorios mediante obtenerCasillero().

Pedido: Representa un pedido individual. Contiene atributos como ID y estado. Es la unidad que se transmite a lo largo del sistema.

Casillero: Simula un espacio de almacenamiento para un pedido. Contiene métodos sincronizados para depositar o retirar pedidos, y estadísticas internas sobre uso y fallos.

Distribuidor: Clase base para los hilos que participan en la cadena (Despachador, Entregador, Verificador). Define estructura común y comportamiento general.

Preparador, Despachador, Entregador, Verificador: Hilos que simulan cada etapa del proceso. Cada uno toma pedidos de una cola, ejecuta

una tarea (con posible falla), y los pasa a la siguiente etapa. Implementan lógica concurrente y aleatorización de tiempo/falla.

En el caso de la asignación de pedidos a casilleros, se implementó una selección aleatoria de los casilleros disponibles. Esto se logró generando un índice aleatorio con la clase Random y accediendo a dicho casillero dentro del arreglo. De este modo, se garantiza una distribución uniforme de pedidos entre los casilleros y se simula un comportamiento no determinista que favorece el testeado de concurrencia y rendimiento del sistema.

Para simular el comportamiento realista del procesamiento de pedidos, se utilizó una distribución gaussiana (normal) para determinar los tiempos de espera entre tareas. Esta distribución permite que la mayoría de los tiempos se agrupen alrededor de un valor promedio, con pequeñas variaciones, lo cual imita con mayor precisión los tiempos variables que se dan en la práctica. Se establecieron límites mínimos y máximos para evitar valores extremos no deseados. Esta elección busca lograr un equilibrio entre realismo, fluidez de simulación y control del rendimiento del sistema.

2.3 Herramientas Utilizadas

- Lenguaje: Java
- Entorno de desarrollo: Visual Studio Code e IntelliJ IDEA
- Control de versiones: Git y GitHub
- Gestor de dependencias: Maven
- Para gráficos: LibreOffice Calc

3. Plan de Versiones en el Repositorio

A lo largo del desarrollo del proyecto, se siguió una estrategia incremental basada en versiones, lo cual permitió integrar y verificar nuevas funcionalidades de forma progresiva. A continuación se detalla el plan de versiones.

Versión 1 - Estructura inicial del proyecto

Clases: Casillero, Estado, CentroDeAlmacenamiento.

Inicialización de matriz de casilleros.

Método `toString()` para visualizar el estado de la matriz de casilleros.

Versión 2 - Preparación de pedidos y prueba de hilos

Clase Pedido.

Lista inicial de 500 pedidos.

Se desarrolló el módulo de preparación de pedidos, con los hilos Preparador.

Se realizaron pruebas para verificar la correcta interacción entre hilos y estructuras concurrentes.

Versión 3 - Flujo completo y manejo de fallos

Se añadieron las clases Despachador, Entregador y Verificador, completando el ciclo de procesamiento de pedidos.

Se implementó el manejo de fallos con registro en listas separadas.

El sistema ya permite procesar 500 pedidos de forma concurrente.

Se implementan `LinkedBlockingDeque` para el acceso seguro a las colas de pedidos.

Versión 4 - Log y generación de archivos estadísticos

Se agregó el módulo de logging dentro del main, con escritura periódica en archivos .txt y .csv.

Se introdujeron esperas aleatorias y configurables para simular las condiciones realistas.

Se incorporó cálculo del tiempo de ejecución y la impresión del estado final del almacenamiento.

Versión 5 - Documentación, dependencias y diagramas

Inclusión de los diagramas de clases y de secuencia en alta calidad.

Migración del proyecto a Maven para el manejo de dependencias.

Revisión final del código y documentación para asegurar la portabilidad.

Versión 6 - Revisión

Se independizó del main la lógica del logging, creando una estructura que se ejecuta en otro hilo.

Se reemplazó la LinkedBlockingDeque.

Se corrigieron problemas de concurrencia preexistentes.

4. Resultados y análisis

Durante la ejecución del programa, se generaron dos archivos de log: uno en formato **.txt** y otro en **.csv**. Ambos contienen información sobre la cantidad de pedidos verificados y fallidos en intervalos regulares de tiempo (cada 200 ms). Estos datos permiten observar la evolución del sistema y detectar posibles cuellos de botella o desequilibrios en la concurrencia.

Ejemplos de líneas del log .txt:

[1 ms] Verificados: 0, Fallidos: 0

[217 ms] Verificados: 0, Fallidos: 0

[419 ms] Verificados: 2, Fallidos: 3

[619 ms] Verificados: 4, Fallidos: 4

[822 ms] Verificados: 5, Fallidos: 4

[1024 ms] Verificados: 9, Fallidos: 5

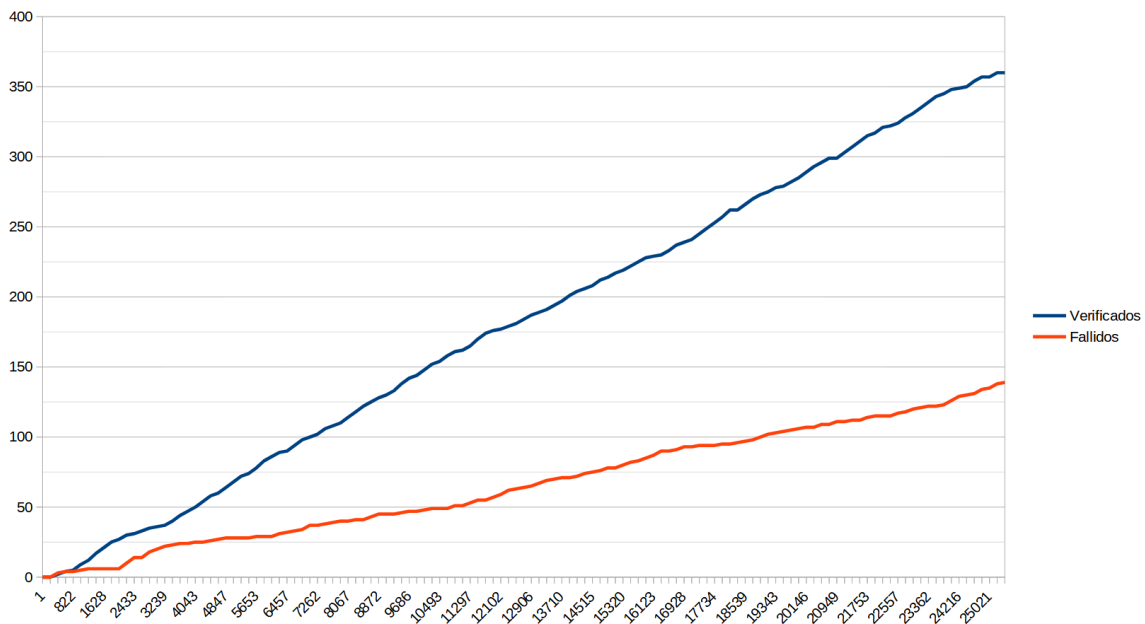
Ejemplos de líneas del log .csv:

Timestamp(ms)	Verificados	Fallidos
[1 ms]	0	0
[217 ms]	0	0
[419 ms]	2	3
[619 ms]	4	4
[822 ms]	5	4
[1024 ms]	9	5
...

Estos resultados se obtuvieron trabajando con:

3 hilos preparadores, 2 hilos despachadores, 3 hilos entregadores y 2 hilos verificadores.

A partir del archivo CSV, se generó un gráfico utilizando **LibreOffice Calc**, lo que permitió visualizar de forma clara la evolución del sistema:



4.1 Comparación de tiempo de ejecución con una sola instancia por etapa

Con el fin de contrastar el rendimiento del sistema concurrente, se realizó una simulación alternativa utilizando solo un hilo por cada etapa del circuito logístico: un Preparador, un Despachador, un Entregador y un Verificador. Esta configuración reduce la paralelización del sistema, ya que cada proceso cuenta con una única instancia activa.

Los resultados obtenidos evidencian diferencias marcadas:

- Simulación concurrente (múltiples hilos por etapa): Tiempo total \approx 20 segundos.
- Simulación limitada (un hilo por etapa): Tiempo total \approx 41 segundos.

Este contraste muestra que el sistema gana eficiencia cuando las etapas del proceso pueden ejecutarse con más de un hilo.

5. Decisiones de diseño

Separación por responsabilidades: una clase por rol

Cada etapa del proceso (preparador, despachador, entregador y verificador) fue modelada mediante una clase específica que hereda de una clase abstracta común. Esta separación favorece la legibilidad, el mantenimiento del código y permite la escalabilidad del sistema.

Randomización de tiempos de procesamiento

Uno de los objetivos principales del trabajo es simular comportamientos realistas bajo condiciones inciertas. Para esto, se incorporó una lógica de fallos aleatorios en cada etapa del procesamiento de pedidos. A su vez, los tiempos de procesamiento de cada hilo también fueron randomizados utilizando la distribución gaussiana (normal), a través de la función `random.nextGaussian()`. Esto permite que los delays se distribuyan alrededor de un valor medio, reflejando la variabilidad habitual que puede encontrarse en sistemas reales, donde los tiempos de atención no son constantes pero tienden a una media.

Persistencia del log en formatos aptos para lectura y graficación

Se implementó un registro periódico de estadísticas del sistema en archivos `.txt` y `.csv`. Esto facilita el análisis posterior de resultados y la visualización mediante herramientas externas como hojas de cálculo o software de gráficos.

6. Conclusiones

Este trabajo nos permitió llevar a la práctica los principales conceptos de programación concurrente, viendo de forma concreta cómo se comportan los hilos cuando comparten recursos y trabajan en paralelo.

Uno de los principales desafíos fue el manejo correcto de los recursos compartidos entre hilos, como las colas de pedidos y la asignación de casilleros. Estos fueron resueltos mediante el uso de estructuras thread-safe y mecanismos de sincronización, lo cual garantiza la integridad de los datos y evitó condiciones de carrera.

Además, se pudo observar la diferencia de rendimiento entre un enfoque concurrente y uno secuencial. La versión concurrente logró completar la simulación en aproximadamente 20 segundos, mientras que la versión limitada a un hilo por etapa requirió cerca de 41 segundos. Esta comparación refuerza la ventaja concreta que ofrece la concurrencia en términos de eficiencia, escalabilidad y realismo.

El trabajo no solo permitió poner en práctica conceptos teóricos sobre concurrencia, sincronización y diseño orientado a objetos, sino también entender su aplicación concreta en escenarios reales. Trabajando con fallos aleatorios y tiempos de procesamiento variables, se pudo observar cómo se comporta el sistema frente a la incertidumbre y la no determinación.

En resumen, aprendimos cómo diseñar un sistema concurrente con múltiples etapas, cómo coordinar hilos que comparten datos, cómo registrar el estado del sistema a lo largo del tiempo y cómo manejar su finalización. Todo esto reforzó los conceptos teóricos vistos en clase con una aplicación concreta.

7. Anexo: Historial de cambios

Este apartado detalla las modificaciones realizadas al código como respuesta a observaciones y correcciones indicadas por el equipo docente, tras la entrega inicial del trabajo. Durante el proceso de revisión, se detectaron errores conceptuales, problemas de encapsulamiento, falta de responsabilidades bien definidas en algunas clases, entre otros aspectos.

A partir de dichas devoluciones, se llevaron a cabo una serie de refactorizaciones y mejoras estructurales para cumplir con los requisitos establecidos por la consigna, garantizar un diseño más sólido y mejorar la legibilidad.

Corrección de la concurrencia

`Casillero`

Se identificó a los objetos de la clase `Casillero` como recursos compartidos entre múltiples hilos. Por esta razón, se aplicó sincronización directamente sobre sus métodos, ya que fue en esta clase (y no en `CentroDeAlmacenamiento` como se había considerado inicialmente) donde efectivamente se manifestaban los problemas de concurrencia.

`CentroDeAlmacenamiento`

Se corrigió el enfoque erróneo en el que se consideraba a esta clase como el recurso compartido. Se eliminaron las sincronizaciones innecesarias en sus métodos, permitiendo el acceso concurrente a la matriz de casilleros sin bloqueos innecesarios.

`SynchronizedList`

Se creó esta clase para encapsular una lista común (no concurrente) y protegerla mediante bloques sincronizados. De este modo, se reemplazó el uso de estructuras de la librería `java.util.concurrent`, como `LinkedBlockingDeque`, cumpliendo con los requisitos de la consigna.

Además de encapsular la protección sincronizada de listas comunes, se incorporó el método `getRandom()`, que permite obtener y eliminar un elemento aleatorio de forma atómica y segura. Este método reemplaza una

implementación anterior en los hilos, donde se evaluaba primero si la lista no estaba vacía (`if`) y luego se accedía con `remove(indiceAleatorio)`, lo cual exponía el código a condiciones de carrera. El nuevo enfoque resuelve esta situación al encapsular ambas operaciones (verificación y extracción) dentro de un único método sincronizado.

En caso de estar vacía, la lista sigue lanzando una excepción, por lo que el comportamiento sigue siendo controlable desde la lógica del hilo.

Preparador

Se eliminaron las sincronizaciones en sus métodos, dado que la protección de las estructuras ya estaba garantizada por `SynchronizedList`, evitando así redundancia y sobreprotección.

Redistribución de responsabilidades

Main

Se eliminó la responsabilidad de manejo de archivos y registro de eventos. Esta tarea fue delegada a una nueva clase específica, promoviendo un enfoque más acorde a los principios de responsabilidad única.

Distribuidor

Se modificó su implementación para que utilice la interfaz `Runnable` en lugar de extender directamente de `Thread`, permitiendo una mayor flexibilidad y mejor adhesión a buenas prácticas de diseño orientado a objetos.

`LoggerManager`, `AbstractLogWriter`, `ConsoleLogger`,
`CsvLogger`, `TxtLogger`

Se crearon estas nuevas clases para abstraer y distribuir las tareas relacionadas con el loggeo de información en diferentes formatos (consola, archivos TXT y CSV), centralizando dicha funcionalidad fuera de `Main`.

Actualización de diagramas

Se actualizaron los **diagramas de clases** y **diagramas de secuencia** para reflejar la nueva estructura del sistema, con las clases auxiliares y sus relaciones.