Nombre del curso:

Programación paralela y distribuida

Nombre del trabajo:

Avance #2 Proyecto Final

Estudiantes:

- Byron Chacón Jiménez
- Kevin Latino Marín
- Santiago Lobo Ulloa
- Alejandro Quesada Ruiz

Profesor:

Jorge Bastos Garcia

Fecha de entrega:

23 de junio

2. Introducción

Objetivo del programa

El programa implementa un **Simulador de Control de Autobuses** que modela la ejecución concurrente de múltiples hilos, cada uno representando un autobús recorriendo la ruta San José – Cartago – San José. La simulación permite comprender cómo interactúan los hilos en un entorno compartido, cómo se coordinan los accesos a recursos y cómo se gestionan situaciones de conflicto en el movimiento de los autobuses.

El objetivo principal es ofrecer una herramienta educativa que permita visualizar el funcionamiento interno de la concurrencia en Java, mostrando el comportamiento de los hilos, la sincronización y los eventos generados durante la ejecución del sistema.

Alcance

El simulador desarrollado incluye:

- Inicialización de la ruta con un conjunto de paradas fijas.
- Creación de una flota de autobuses, cada uno ejecutándose en su propio hilo.
- Movimiento continuo de los autobuses en ambas direcciones del recorrido.
- Gestión de conflictos cuando dos autobuses intentan acceder simultáneamente a la misma parada.
- Visualización gráfica en tiempo real mediante una interfaz basada en Swing.
- Generación de métricas como tiempos de viaje, tiempos de espera y eventos de sincronización.

El simulador **no incluye**:

- Tráfico real dinámico.
- Persistencia en base de datos.

- Inteligencia artificial aplicada a la movilidad.

- Comunicación en red o integración externa.

Público objetivo

Este proyecto está dirigido a:

- Estudiantes del curso de programación paralela y distribuida.

- Desarrolladores con interés en aprender sobre hilos y sincronización.

- Profesores que deseen utilizar un simulador visual para explicar concurrencia.

- Personas interesadas en sistemas de simulación y movilidad.

3. Requisitos del sistema

Hardware

Para ejecutar el simulador se recomienda:

- Procesador de doble núcleo a 2.0 GHz o superior.
- 4 GB de memoria RAM (8 GB recomendado).
- 500 MB de espacio disponible en disco.
- Gráficos integrados suficientes para ejecutar Java Swing.

Software

- Windows, macOS o Linux.
- Java JDK 8 o superior.
- IntelliJ IDEA, Eclipse o NetBeans.
- Librerías estándar de Java (Swing, manejo de hilos y colecciones).

- Opción de usar Git para control de versiones.

4. Conceptos básicos

Hilos

¿Cómo se crean y gestionan los hilos?

Según la documentación oficial de Oracle sobre Java SE 21 (2023), los hilos pueden crearse extendiendo la clase Thread o implementando la interfaz Runnable, lo que permite encapsular la lógica concurrente. La gestión del hilo implica iniciarlo mediante start(), supervisar su ejecución, reaccionar a solicitudes de interrupción y manejar las transiciones entre estados como *new*, *runnable*, *blocked*, *waiting* y *terminated* durante todo su ciclo de vida.

¿Qué tareas realiza cada hilo?

Cada autobús ejecuta un hilo encargado de:

- Avanzar entre paradas.

- Actualizar su estado en tiempo real.

- Simular tiempos de espera.

- Coordinar el acceso a una parada antes de ocuparla.

¿Cómo se sincronizan los hilos?

Según *Java Concurrency in Practice Updated Notes 2023*, la sincronización entre hilos en Java se logra mediante mecanismos como synchronized, Semaphore y ReentrantLock, los cuales permiten controlar el acceso exclusivo a recursos compartidos. En un simulador de transporte, esto asegura que solo un autobús pueda usar una parada en un momento dado, evitando condiciones de carrera y comportamientos inconsistentes.

UDP

¿Para qué se utiliza UDP?

Según un estudio de Performance Evaluation of UDP‑Based Data Transmission with Acknowledgment for Various Network Topologies in IoT Environments (2023), el protocolo User Datagram Protocol (UDP) es frecuentemente seleccionado para implementaciones de Internet de las Cosas (IoT) debido a su simplicidad y bajo overhead; sin embargo, destaca que esa misma ligereza implica que su fiabilidad decae en entornos dinámicos o con recursos limitados, lo que limita su idoneidad para aplicaciones donde la integridad de datos sea crítica.

¿Cómo se establecen las conexiones UDP?

Según el artículo de GeeksforGeeks (2025), al programar en Java con la clase DatagramSocket para comunicarse vía UDP, se observa que no es necesario establecer una conexión formal entre cliente y servidor — los paquetes se envían y reciben sin negociación previa, lo cual favorece escenarios donde prima la latencia mínima frente a la entrega garantizada.

¿Cómo se envían y reciben datos?

Según la guía técnica de Baeldung sobre programación UDP en Java (2024), el intercambio de datos en este protocolo se realiza mediante el envío de datagramas: para transmitir información se utiliza socket.send(packet), mientras que para recibirlos el programa emplea socket.receive(packet). Ambos métodos operan sin necesidad de establecer una conexión persistente, ya que UDP trabaja de forma no orientada a conexión y trata cada paquete como una unidad independiente.

TCP

¿Para qué se utiliza TCP?

Según *Computer Networking: A Top-Down Approach* (Kurose & Ross, edición 2023), el protocolo TCP se utiliza en situaciones donde es indispensable asegurar que los datos

lleguen completos, en el orden correcto y sin pérdidas, ya que incorpora mecanismos de control de flujo, control de congestión y confirmación de entrega.

¿Cómo se establece la conexión?

Según *Java Network Programming* de O'Reilly (edición 2023), una conexión TCP se establece cuando el servidor crea un ServerSocket que queda a la espera de solicitudes entrantes, y el cliente inicia la comunicación creando un Socket para conectarse. Una vez que ambos extremos completan el handshake de tres vías característico de TCP, queda formado un canal confiable y persistente entre cliente y servidor.

¿Cómo se envían y reciben datos?

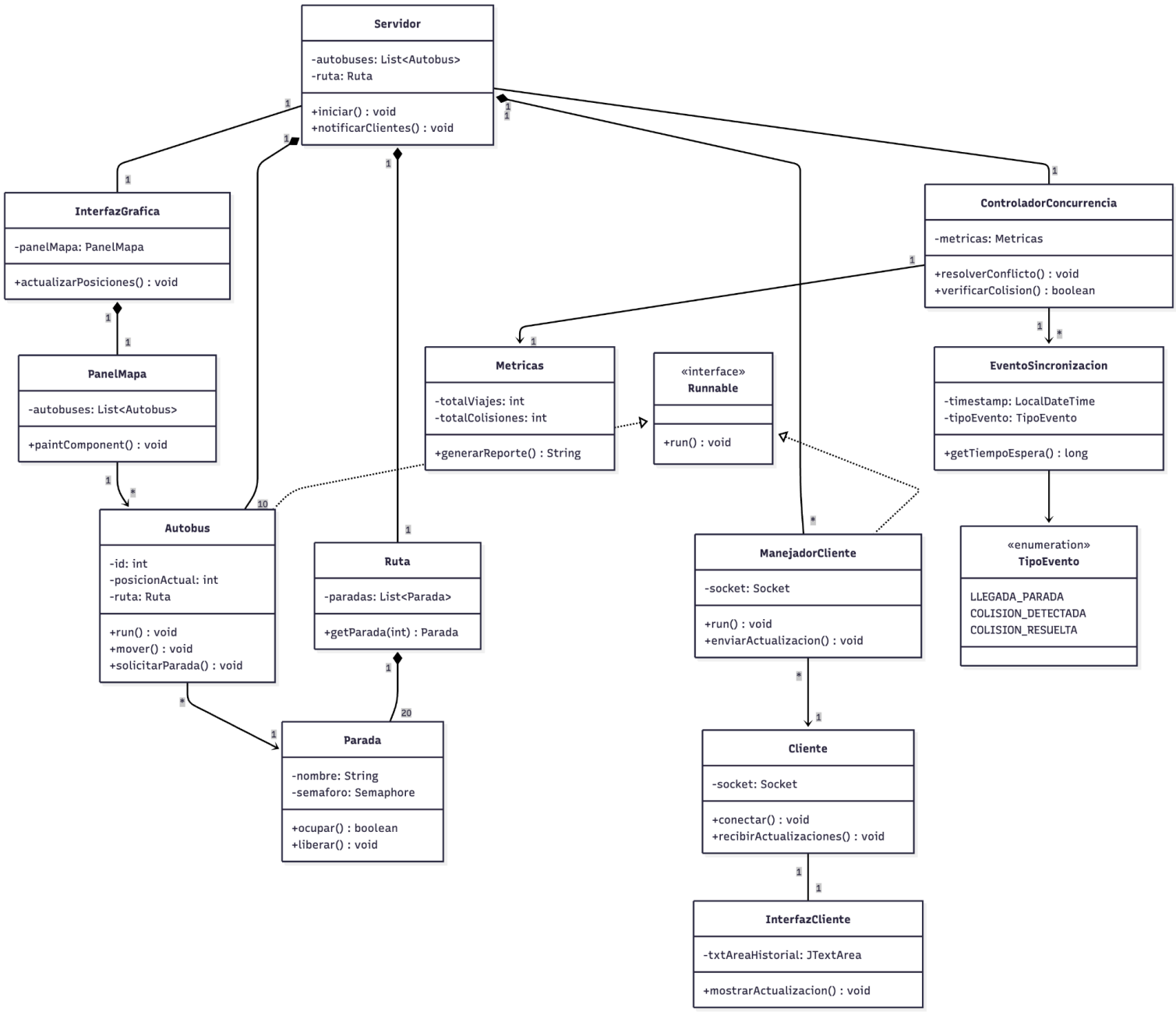
Según la documentación oficial de Oracle para Java SE 23 (2023), una vez establecida la conexión TCP, el cliente y el servidor intercambian información mediante flujos: el envío de datos se realiza a través de un **OutputStream**, mientras que la recepción se efectúa

usando un **InputStream**, lo cual garantiza que los datos se transfieran de forma ordenada y confiable.

5. Arquitectura del programa

5.1 Diagrama de clases o componentes

El simulador de control de autobuses está diseñado siguiendo el patrón Cliente-Servidor con programación concurrente mediante hilos. A continuación se presenta el diagrama de clases UML que visualiza la estructura del programa y las relaciones entre sus componentes:


Servidor
-autobuses: List<Autobus>
-ruta: Ruta
+iniciar() : void
+notificarClientes() : void
InterfazGrafica
-panelMapa: PanelMapa
+actualizarPosiciones() : void
PanelMapa
-autobuses: List<Autobus>
+paintComponent() : void
ControladorConcurrencia
-metricas: Metricas
+resolverConflicto() : void
+verificarColision() : boolean
Metricas
-totalViajes: int
-totalColisiones: int
+generarReporte() : String
«interface»
Runnable
+run() : void
EventoSincronizacion
-timestamp: LocalDateTime
-tipoEvento: TipoEvento
+getTiempoEspera() : long
Autobus
-id: int
-posicionActual: int
-ruta: Ruta
+run() : void
+mover() : void
+solicitarParada() : void
Ruta
-paradas: List<Parada>
+getParada(int) : Parada
ManejadorCliente
-socket: Socket
+run() : void
+enviarActualizacion() : void
«enumeration»
TipoEvento
LLEGADA_PARADA
COLISION_DETECTADA
COLISION_RESUELTA
Parada
-nombre: String
-semaforo: Semaphore
+ocupar() : boolean
+liberar() : void
Cliente
-socket: Socket
+conectar() : void
+recibirActualizaciones() : void
InterfazCliente
-txtAreaHistorial: JTextArea
+mostrarActualizacion() : void

5.2 Descripción de módulos

Módulo Principal (Servidor)

Clase: Servidor

- **Funcionalidad**: Gestiona la lógica central del simulador. Crea y coordina los hilos de autobuses, mantiene el estado de la ruta y maneja las conexiones de clientes.
- **Responsabilidades**:
    - Inicializar la ruta con las 20 paradas
    - Crear y lanzar los 10 hilos de autobuses
    - Aceptar conexiones de clientes mediante sockets TCP
    - Notificar a los clientes sobre actualizaciones de estado
    - Coordinar el sistema de sincronización

Interrelaciones:

- Crea y gestiona instancias de Autobús
- Mantiene una referencia a Ruta
- Crea instancias de ManejadorCliente por cada conexión
- Utiliza ControladorConcurrencia para resolver conflictos

Módulo de Autobuses

Clase: Autobus (implementa Runnable)

- **Funcionalidad**: Representa un autobús individual que se mueve por la ruta. Cada instancia se ejecuta en su propio hilo.
- **Responsabilidades**:
    - Moverse entre paradas de forma autónoma
    - Solicitar acceso exclusivo a las paradas
    - Simular tiempos de viaje variables (usando Random y Thread.sleep())
    - Notificar su posición actual al servidor
    - Detectar cuando completa un ciclo completo

Interrelaciones:

- Interactúa con objetos Parada para solicitar acceso
- Consulta Ruta para conocer la siguiente parada
- Es observado por el Servidor

Módulo de Ruta y Paradas

Clase: Ruta

- **Funcionalidad**: Contiene y gestiona la lista ordenada de las 20 paradas de la ruta San José-Cartago-San José.
- **Responsabilidades**:
    - Almacenar información de todas las paradas
    - Proporcionar acceso a las paradas por índice
    - Mantener las distancias entre paradas

    - Permitir navegación secuencial por la ruta

Clase: Parada

- **Funcionalidad**: Representa un punto específico en la ruta donde los autobuses se detienen.
- **Responsabilidades**:
    - Controlar acceso exclusivo mediante Semaphore (binario)
    - Registrar qué autobús la está ocupando
    - Permitir que autobuses soliciten y liberen el acceso
    - Prevenir condiciones de carrera

Interrelaciones:

- Ruta contiene una lista de objetos Parada
- Autobus solicita acceso a instancias de Parada
- Utiliza sincronización mediante java.util.concurrent.Semaphore

Módulo de Sincronización

Clase: ControladorConcurrencia

- **Funcionalidad**: Gestiona los eventos de sincronización y resolución de conflictos cuando múltiples autobuses intentan acceder a la misma parada.
- **Responsabilidades**:
    - Detectar colisiones (dos autobuses en la misma parada)

    - Aplicar política de resolución: mover un lugar adelante al autobús retrasado
    - Registrar todos los eventos de sincronización
    - Generar métricas de desempeño
    - Mantener estadísticas de tiempos de espera

Clase: EventoSincronizacion

- **Funcionalidad**: Representa un evento específico de sincronización entre autobuses.
- **Responsabilidades**:
    - Registrar timestamp del evento
    - Identificar autobuses involucrados
    - Almacenar tipo de evento (colisión, espera, etc.)
    - Documentar la resolución aplicada

Interrelaciones:

- Es utilizado por Servidor y Autobus para coordinar accesos
- Crea instancias de EventoSincronizacion
- Actualiza objeto Metricas

Módulo Cliente-Servidor

Clase: Cliente

- **Funcionalidad**: Aplicación cliente que se conecta al servidor para monitorear un autobús específico.

- **Responsabilidades**:
    - Establecer conexión TCP con el servidor
    - Solicitar monitoreo de un autobús específico
    - Recibir actualizaciones en tiempo real
    - Mostrar historial de paradas visitadas
    - Manejar desconexiones

Clase: ManejadorCliente (implementa Runnable)

- **Funcionalidad**: Gestiona la comunicación con un cliente específico en el servidor.
- **Responsabilidades**:
    - Recibir solicitudes del cliente
    - Enviar actualizaciones de estado del autobús monitoreado
    - Mantener la conexión activa
    - Notificar cuando el día de servicio finaliza

Interrelaciones:

- Servidor crea una instancia de ManejadorCliente por cada conexión
- Se comunica mediante sockets TCP
- Lee de BufferedReader y escribe a PrintWriter

Módulo de Interfaz Gráfica

Clase: InterfazGrafica

- **Funcionalidad**: Ventana principal del servidor que muestra el estado de todos los autobuses en tiempo real.
- **Responsabilidades**:
    - Mostrar el mapa de la ruta
    - Visualizar la posición de cada autobús
    - Mostrar eventos de sincronización
    - Proporcionar controles de inicio/detención
    - Mostrar log de eventos

Clase: PanelMapa (extiende JPanel)

- **Funcionalidad**: Componente personalizado que dibuja el mapa y los autobuses.
- **Responsabilidades**:
    - Renderizar la imagen del mapa base
    - Dibujar iconos de autobuses en sus posiciones actuales
    - Marcar las 20 paradas en el mapa
    - Actualizar la visualización periódicamente
    - Usar colores diferentes para cada autobús

Clase: InterfazCliente

- **Funcionalidad**: Ventana de la aplicación cliente.
- **Responsabilidades**:
    - Permitir seleccionar qué autobús monitorear
    - Mostrar historial de paradas

    - Indicar estado de conexión
    - Mostrar timestamp de cada actualización

Interrelaciones:

- InterfazGrafica contiene instancia de PanelMapa
- Ambas interfaces utilizan componentes Swing (JFrame, JPanel, JTextArea, etc.)
- Se actualizan mediante el patrón Observer o callbacks

Módulo de Métricas

Clase: Metricas

- **Funcionalidad**: Recopila y almacena estadísticas sobre el desempeño del sistema.
- **Responsabilidades**:
    - Contar total de viajes completados
    - Registrar número de colisiones detectadas
    - Calcular tiempos de espera promedio
    - Generar reportes estadísticos
    - Mantener métricas por autobús

Interrelaciones:

- Es actualizada por ControladorConcurrencia
- Consultada por InterfazGrafica para mostrar estadísticas

5.3 Flujo de datos

Esta sección describe cómo se mueven los datos entre los diferentes componentes del sistema durante la ejecución del simulador.

FLUJO PRINCIPAL DE EJECUCIÓN

1. Inicialización

El programa inicia en el método main() que crea el Servidor. El Servidor inicializa la Ruta con las 20 paradas, crea los 10 objetos Autobus y lanza sus hilos. Finalmente, muestra la InterfazGrafica.

2. Ciclo de vida de un Autobús

Cada hilo de Autobus ejecuta un ciclo continuo:

- Calcula una velocidad aleatoria para variar tiempos de viaje
- Se mueve a la próxima parada en la ruta
- Solicita acceso a la parada usando el método ocupar() que internamente usa el semáforo
- Si hay colisión detectada, el ControladorConcurrencia resuelve el conflicto
- Espera un tiempo simulado en la parada usando Thread.sleep()
- Libera la parada llamando al método liberar() que libera el semáforo
- Notifica al Servidor de su nueva posición
- Repite el ciclo hasta completar el recorrido completo

3. Comunicación Cliente-Servidor

El Cliente se conecta al Servidor. El Servidor acepta la conexión y crea un ManejadorCliente que se ejecuta en su propio hilo. El Cliente solicita monitorear un autobús específico. El ManejadorCliente envía actualizaciones cada vez que ese autobús cambia de parada. El Cliente recibe las actualizaciones y la InterfazCliente las muestra en pantalla.

4. Resolución de conflictos

Cuando el Autobus1 solicita la Parada N y logra ocuparla exitosamente. Luego el Autobus2 intenta solicitar la misma Parada N pero queda bloqueado esperando el semáforo. El ControladorConcurrencia detecta la colisión. El ControladorConcurrencia resuelve el conflicto moviendo al Autobus2 a la Parada N+1. Se crea un EventoSincronizacion para registrar lo ocurrido. La InterfazGrafica muestra el evento en el log.

FLUJO DE DATOS ENTRE COMPONENTES

Los Autobus envían su posición actual al Servidor.

El Servidor envía notificaciones a los ManejadorCliente.

Los ManejadorCliente envían actualizaciones a sus respectivos Cliente.

Los Cliente muestran la información en su InterfazCliente.

El Servidor también actualiza la InterfazGrafica con las posiciones.

La InterfazGrafica actualiza el PanelMapa para redibujar los autobuses.

Los Autobus solicitan acceso a las Parada.

Las Parada reportan su estado al ControladorConcurrencia.

El ControladorConcurrencia crea EventoSincronizacion cuando detecta conflictos.

Los EventoSincronizacion se acumulan en una lista.

El ControladorConcurrencia actualiza las Metricas del sistema.

SINCRONIZACIÓN DE DATOS

- El acceso a las Paradas está protegido mediante Semaphore. Cada semáforo tiene 1 permiso, garantizando acceso exclusivo.

- La lista de autobuses en el Servidor se accede de forma sincronizada usando synchronized o Collections.synchronizedList() para evitar condiciones de carrera.

- Los eventos de sincronización se almacenan en una cola thread-safe como ConcurrentLinkedQueue.

- La actualización de la interfaz gráfica se realiza mediante SwingUtilities.invokeLater() para garantizar que solo el Event Dispatch Thread modifica componentes Swing.

PROTOCOLO DE COMUNICACIÓN TCP

El servidor y los clientes se comunican mediante mensajes de texto con un formato específico.

Formato de mensajes del Servidor hacia el Cliente:

TIMESTAMP|AUTOBUS_ID|PARADA_ACTUAL|NOMBRE_PARADA|ESTADO

Ejemplo:

2024-06-23T10:30:45|3|12|Escuela República Dominicana|EN_MOVIMIENTO

Comandos que el Cliente puede enviar al Servidor:

MONITOREAR:3 - Solicita seguimiento del autobús número 3

DESCONECTAR - Cierra la conexión con el servidor

ESTADO - Solicita el estado actual de todos los autobuses

5.4 Patrones de diseño utilizados

1. **Singleton**: La clase Servidor puede implementarse como singleton para garantizar una única instancia.
2. **Observer**: Los clientes observan cambios en el estado de los autobuses.
3. **Producer-Consumer**: Los autobuses producen eventos que son consumidos por el controlador de concurrencia.

4. **Thread Pool**: Aunque cada autobús tiene su propio hilo, los ManejadorCliente podrían usar un ExecutorService.
5. **Factory**: Creación de autobuses y paradas mediante métodos factory.

5.5 Consideraciones de concurrencia

- **Hilos**: 10 hilos para autobuses + 1 hilo servidor + N hilos para clientes
- **Sincronización**: Semaphore para paradas, ReentrantLock para estructuras compartidas
- **Evitar deadlocks**: Orden consistente de adquisición de locks, timeouts en semáforos
- **Thread-safety**: Todas las colecciones compartidas son thread-safe
- **Actualización GUI**: Solo el Event Dispatch Thread modifica componentes Swing

6. Guía del usuario

Instalación

Requisitos previos

- Tener instalado Java (JDK 8 o superior).

- NetBeans o cualquier IDE compatible.

- Opcional: Maven.

Instalación en Windows / macOS / Linux mediante NetBeans

1. Abrir NetBeans.

2. Seleccionar **File > Open Project…**

3. Elegir la carpeta del proyecto.

4. Ejecutar **Clean & Build**.

5. Presionar **Run Project**.

Instalación por línea de comandos

cd proyecto

mvn clean package

java -cp target/ProyectoFinal.jar com.mycompany.proyectofinal.ProyectoFinal

Interfaz de usuario

La interfaz está compuesta por:

Encabezado

- Barra superior con el título *"Simulador San José – Cartago"*.

Panel principal (izquierda)

- Muestra el mapa.

- Muestra los 20 puntos de parada.

- Visualiza la posición y dirección de cada autobús, con su icono e identificación.

Panel lateral (derecha)

Dividido en dos secciones:

- **Estado de los buses:** lista con:

    - Estado (EN_PARADA / EN_TRANSITO / ESPERANDO)
    - Parada actual
    - Dirección

- **Eventos recientes:**

  Registro de conflictos y acciones relevantes.

Botonera inferior

- **Iniciar:** activa los hilos de los autobuses.
- **Detener:** detiene la simulación y reinicia posiciones.

Tutoriales

Iniciar la simulación

1. Abrir el proyecto.
2. Ejecutar la aplicación.

3. Presionar **Iniciar**.

4. Observar el movimiento de los autobuses en tiempo real.

Detener la simulación

1. Presionar **Detener**.

2. Los hilos terminan correctamente.

3. La vista vuelve al estado inicial.

Visualizar conflictos

1. Observar el panel “Eventos recientes”.

2. Cuando dos buses buscan la misma parada, aparece un mensaje indicando espera.

3. Útil para explicar sincronización.

7. Referencia de la Aplicación

Clase Autobus

Atributos principales

- id, nombre, color, ruta, posicionActual, enIda, estado, icono.

Métodos

- run(): controla el ciclo de vida del autobús.
- mover(): avanza a la siguiente parada.
- getParadaActualNombre(): devuelve la parada actual.
- getDireccion(): indica ida/vuelta.
- getEstado(): estado actual del hilo.

Clase Ruta

- Lista ordenada de paradas.
- Métodos: getParada(), size().

Clase Parada

- Atributos: nombre, x, y.

- Métodos: getters.

Clase SimuladorControl

- Controla la ejecución general.

- Métodos: iniciarSimulacion(), detenerSimulacion(), getBuses().

Clase MapaPanel

- Dibuja mapa, autobuses y paradas.

- Método clave: paintComponent().

Clase InfoPanel

- Actualiza estados y registra eventos.

- Métodos: actualizarEstados(), registrarEvento().

Clase SimuladorFrame

- Ventana principal con los paneles y botones.

8. Consideraciones de rendimiento

Análisis de algoritmos

Movimiento de autobuses

- Complejidad O(n) por autobús donde n es el número de paradas.

- En total, O(10n).

Repintado gráfico

- Mapa: O(1)

- Paradas: 20

- Autobuses: 10

- Carga mínima para cualquier equipo moderno.

Sincronización

La contención es baja porque los autobuses están distribuidos en la ruta.

Optimizaciones sugeridas

- Limitar repintado a ~30 FPS.

- Parametrizar velocidades para pruebas.

- Usar ExecutorService en vez de hilos directos.

- Mantener imágenes cargadas en memoria.

9. Depuración y solución de problemas

Mensajes de error comunes

InterruptedException

Ocurre al detener la simulación. Es normal.

Autobuses detenidos

Si un autobús queda “esperando”, significa que otro ocupa su siguiente parada.

Índice fuera de rango

Sucede si se manipulan posiciones manualmente.

Procedimientos de depuración

- Agregar mensajes de consola para seguir el flujo.
- Activar un modo DEBUG.
- Utilizar herramientas del IDE como breakpoints.

- Monitoreo de hilos con VisualVM.

Problemas visuales comunes

- Autobuses superpuestos → conflicto real.

- Eventos duplicados → accesos simultáneos genuinos.

- Estados sin actualizar → refrescar InfoPanel.

BIBLIOGRAFÍA

- Masriap, M.; Amaran, M. H.; Yussoff, Y.; Ab Rahman, R.; Hashim, H. (2023). *Performance Evaluation of UDP-Based Data Transmission with Acknowledgment for Various Network Topologies in IoT Environments*. Electronics, 13(18): 3697. Disponible en: https://www.mdpi.com/2079-9292/13/18/3697

- GeeksforGeeks. (2025, 23 julio). *Working with UDP DatagramSockets in Java*. Disponible en: https://www.geeksforgeeks.org/java/working-udp-datagramsockets-java/
- Kurose, J. & Ross, K. (2023). *Computer Networking: A Top-Down Approach* (8th ed.). Pearson.
- Oracle. (2023). *Java Platform, Standard Edition 23 – Networking (TCP Streams)*. Disponible en: https://docs.oracle.com/en/java/javase/23/
- Harold, E. R. (2023). *Java Network Programming* (5th ed.). O'Reilly Media.
- Baeldung. (2024). *A Guide to UDP in Java*. Disponible en: https://www.baeldung.com/udp-in-java
- Oracle. (2023). *Java Platform, Standard Edition 21 — Concurrency and Threads*. Disponible en: https://docs.oracle.com/en/java/javase/21/
- Goetz, B. (2023). *Java Concurrency in Practice — Updated Notes*. Addison-Wesley.