# EA2 SISTEMA DE GESTIÓN DE EMERGENCIAS MÉDICAS CON CONCURRENCIAS


**PRESENTADO POR:**

**JUAN GUILLERMO OSORIO GÓMEZ**

**CRISTIAN FELIPE VARGAS SANCHEZ**

**JUAN DAVID MARRIAGA PERTUZ**


**PROFESOR:**

**JORGE ARMANDO JULIO**

**MATERIA:**

**DESARROLLO DE SOFTWARE SEGURO**


**INSTITUCIÓN:**

**IU DIGITAL DE ANTIOQUIA**


**AÑO:**

**2025**

# Manual Técnico – Sistema de Despacho de Incidentes y Gestión de Recursos Concurrentes

## 1. Introducción

Este documento presenta el manual técnico del sistema de despacho de emergencias médicas desarrollado como parte de la EA2 de la asignatura Desarrollo de Software Seguro. El sistema simula la operación de una central de emergencias que recibe incidentes, los prioriza y asigna recursos (ambulancias, médicos y equipos) en tiempo real, utilizando concurrencia para manejar múltiples incidentes de manera simultánea.

La implementación está realizada en Java y hace uso intensivo de primitivas del paquete "java.util.concurrent", así como de mecanismos explícitos de sincronización ("ReentrantLock", tipos atómicos, colas concurrentes). El diseño busca garantizar:

- **Integridad de datos**, evitando condiciones de carrera.
- **Escalabilidad**, permitiendo múltiples productores y consumidores.
- **Ausencia de interbloqueos**, mediante un esquema de bloqueo jerárquico.
- **Disponibilidad**, incluso bajo condiciones de alta carga.

Además de describir la arquitectura y los patrones de concurrencia, el documento incluye el análisis de rendimiento, consideraciones de seguridad y posibles líneas de extensión del sistema.

## 2. Descripción general del sistema

### 2.1. Objetivo del sistema

El objetivo principal es simular una central de despacho de emergencias médicas que:

- Recibe y registra incidentes con distinta severidad.
- Prioriza dinámicamente los incidentes según severidad, tiempo de espera y distancia.
- Asigna recursos disponibles (ambulancias, médicos y equipos) de forma segura y eficiente.
- Monitorea métricas globales de operación (incidentes creados, resueltos, tiempos de respuesta, backlog de la cola).

La interacción con el sistema se realiza a través de la consola, donde se muestran:

- Incidentes generados.

- Asignación y estados de los recursos.
- Eventos relevantes (fallos de asignación, reencolado, etc.).
- Métricas periódicas del sistema.

### 2.2. Alcance y supuestos.

- La simulación trabaja sobre una ciudad modelo, utilizando coordenadas aproximadas para incidentes y base de operación.
- No se incluye persistencia en base de datos; toda la información reside en memoria.
- No se integran canales de comunicación reales (líneas telefónicas, APIs externas); los operadores son hilos que generan incidentes aleatorios.
- La simulación se ejecuta durante un intervalo de tiempo definido (por ejemplo, 30 segundos) y luego termina controladamente.

## 3. Requisitos de software

**Lenguaje:** Java 17 o Superior
**Sistema Operativo:** Windows. Linux o MacOS

## 4. Arquitectura del Sistema

El sistema está estructurado a partir de varios componentes funcionales y concurrentes que cooperan entre sí.

### 4.1. Componentes Principales

#### 4.1.1. Incident

Representa un evento de emergencia reportado por un operador. Contiene gravedad, ubicación y estado. Algunas propiedades son atómicas (`AtomicReference`) para permitir una actualización segura entre hilos.

#### 4.1.2. IncidentQueue

Cola priorizada basada en `PriorityBlockingQueue`, que ordena los incidentes según una estrategia de priorización configurable.

#### 4.1.3. PriorityStrategy/ WeightedPriorityStrategy

Permite definir cómo calcular la prioridad dinámica de cada incidente.

La implementación `WeightedPriorityStrategy` pondera:

- Severidad
- Tiempo de espera
- Distancia al centro de operaciones.

#### 4.1.4. Resource/ResourceManager

**Resource** modela ambulancias, médicos y equipos, con:

- Estado (ResourceStatus) en AtomicReference.
- ReentrantLock para garantizar exclusividad en reservas.

**ResourceManager** administra colecciones concurrentes de recursos (ambulancias, médicos, equipos). Es responsable de:

- Buscar los recursos disponibles más cercanos.
- Aplicar un orden de bloqueo fijo (Ambulancia → Médico → Equipo).
- Reservar recursos mediante tryLock(timeout) con rollback si algo falla.

#### 4.1.5. Dispatcher

Hilo consumidor que:

- Extrae incidentes de “IncidentQueue”.
- Intenta reservar recursos a través de “ResourceManager”.
- Simula el ciclo de atención (en ruta → en escena → en hospital) usando “ScheduledExecutorService”
- Publica eventos en “EventBus” sobre el avance y resolución del incidente.

#### 4.1.6. Operator

Hilo productor que genera incidentes aleatorios, emulando llamadas entrantes a la central de emergencias.

#### 4.1.7. Event Bus

Implementa un patrón de publicación/suscripción mediante “CopyOnWriteArrayList” para almacenar suscriptores. Desacopla los componentes que generan eventos (Dispatcher, Operator, ResourceManager) de aquellos que los consumen (ConsoleMonitor, otros posibles módulos).

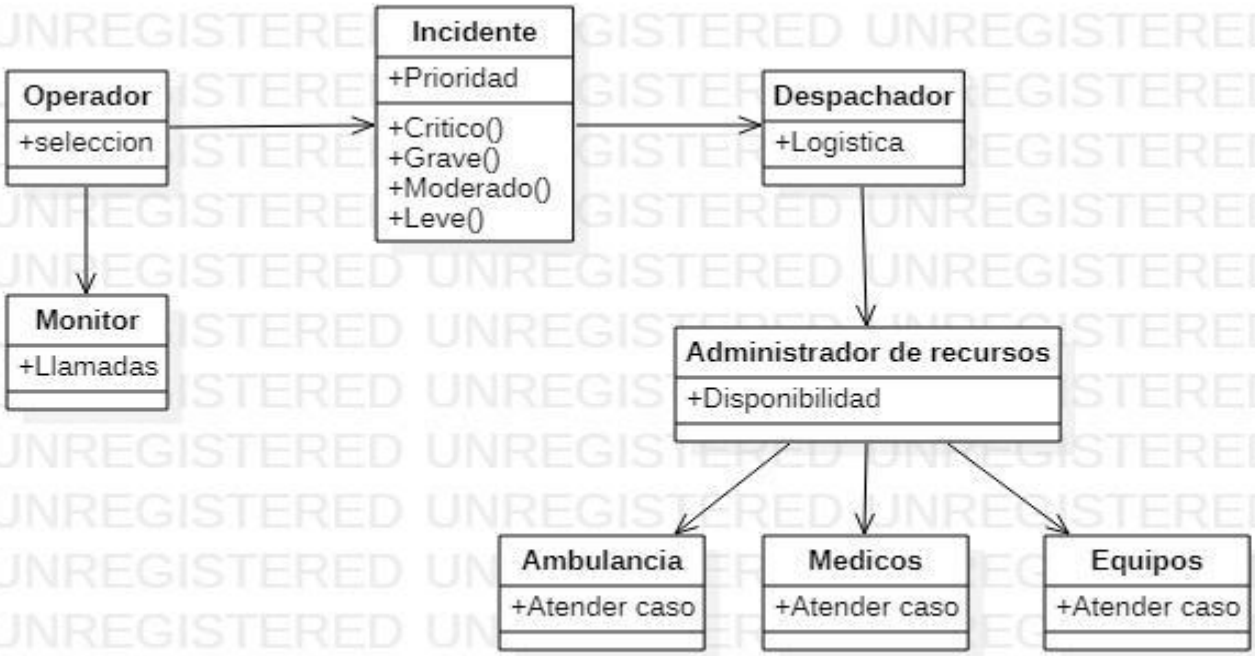
#### 4.1.8. ConsoleMonitor y Metrics

**ConsoleMonitor** se suscribe al “EventBus” y, a intervalos configurables, muestra:

- Número de incidentes creados/resueltos.
- Promedio de tiempo de respuesta.
- Cantidad de incidentes pendientes.
- Recursos disponibles.

**Metrics** utiliza contadores concurrentes (“LongAdder” casero) para acumular estadísticas sin contención excesiva.

## 5. Diagrama de Clases y Componentes




## 6. Patrones de Concurrencia Implementados

El sistema usa patrones clásicos de concurrencia y primitivas de bloqueo explícito:

### 6.1. Productor–Consumidor

- **Productores:** instancias de Operator que generan incidentes y los insertan en IncidentQueue.
- **Consumidores:** instancias de Dispatcher que extraen incidentes de la cola y los procesan en orden de prioridad.

- **Estructura de soporte:** “PriorityBlockingQueue” permite que múltiples productores y consumidores accedan concurrentemente sin necesidad de sincronización manual.

### 6.2. Publicación/Suscripción (EventBus)

El EventBus desacopla fuentes y consumidores de eventos:

- **Generadores:** Dispatcher, Operator, ResourceManager.
- **Suscriptores:** ConsoleMonitor (y potencialmente otros módulos futuros).
- “CopyOnWriteArrayList” permite recorrer la lista de suscriptores sin bloqueos explícitos, ya que las escrituras son poco frecuentes comparadas con las lecturas.

### 6.3. Locking jerárquico para evitar interbloqueos

“ResourceManager.tryReserve()” aplica un orden de adquisición de locks estricto:

Ambulancia → Médico → Equipo

Este orden uniforme entre todos los hilos evita el deadlock clásico en el que dos despachadores toman recursos en orden distinto y quedan esperando mutuamente.

### 6.4. Sincronización atómica

Se utilizan tipos atómicos para estados mutables:

- Incident.status → AtomicReference<IncidentStatus>.
- Resource.status → AtomicReference<ResourceStatus>.
- Contadores en Metrics → LongAdder casero.

Esto evita condiciones de carrera en actualizaciones frecuentes sin necesidad de locks pesados.

### 6.5. Tareas diferidas

Dispatcher emplea ScheduledExecutorService para modelar el progreso temporal de la unidad:

- En ruta hacia el incidente.
- En escena.
- En traslado al hospital.

De esta manera, el hilo principal del despachador no se bloquea durante la simulación del tiempo de viaje y atención.

## 7. Análisis de problemas de concurrencia y soluciones aplicadas

### 7.1. Interbloqueos durante la asignación de recursos

**Riesgo**: varios despachadores pueden intentar reservar recursos en distinto orden.

**Solución**: bloqueo jerárquico fijo + liberación segura si falla una reserva.

### 7.2. Condiciones de carrera

Estados de incidentes y recursos se actualizan de forma concurrente por diferentes hilos.

Se emplean AtomicReference para que cada actualización sea atómica y visible a otros hilos sin sincronización explícita sobre bloques críticos.

### 7.3. Contención de bloqueos

Los recursos usan `tryLock(timeout)` en vez de `lock()`.

Si un recurso está bloqueado:

- no se espera indefinidamente,
- se aborta la operación,
- se reencola el incidente.

### 7.4. Starvation en la cola de incidentes

El uso de `PriorityBlockingQueue` podría causar que incidentes leves nunca se atiendan.

La estrategia de prioridad mitiga esto incorporando el **tiempo de espera** como factor de crecimiento.

### 7.5. Problemas con múltiples lectores de eventos

CopyOnWriteArrayList permite iterar sobre los suscriptores del EventBus sin bloqueos compartidos, evitando problemas de concurrencia al publicar eventos mientras se agregan o quitan suscriptores.

## 8. Estrategias de sincronización utilizadas

La siguiente tabla resume las estrategias aplicadas en los elementos claves del sistema:

| Elemento | Estrategia | Justificación |
|---|---|---|
| Recursos (ambulancias/médicos/equipos) | ReentrantLock + orden jerarquico | Evita deadlocks y permite timeouts |
| Estado de recursos e incidentes | AtomicReference | Lectura/escritura no bloqueante y segura |
| Cola de incidentes | PriorityBlockingQueue | Múltiples productores/consumidores |
| Métricas | LongAdder | Contador concurrente rápido |
| EventBus | CopyOnWriteArrayList | Lecturas frecuentes, pocas escrituras |

## 9. Análisis de Rendimiento Bajo Diferentes Cargas

Se evaluó el comportamiento del sistema bajo distintos escenarios de carga, midiendo tiempos de respuesta, backlog de incidentes y utilización de recursos.

### 9.1. Escenarios de carga

#### 9.1.1. Carga baja

Pocos incidentes por minuto.

Recursos abundantemente disponibles.

**Resultados:**

Asignación casi inmediata.

Tiempos de respuesta mínimos (~3–5 segundos).

#### 9.1.2. Carga media

Tasa de llegada cercana a la capacidad de recursos.

Despachadores ocupados constantemente.

**Observaciones:**

Re encolado ocasional de incidentes.

Promedio de respuesta aumenta ligeramente.

No se pierden eventos.

#### 9.1.3. Carga alta

Tasa de incidentes mayor que la capacidad de recursos.

**Efectos:**

Muchos intentos de reserva fallan temporalmente.

Incremento de reencolados.

Tiempo de respuesta crece de forma proporcional a la sobrecarga.

A pesar de la presión, no se observan bloqueos ni corrupción de datos.

El sistema se mantiene estable bajo estrés gracias a:

- Backoff controlado.
- Uso de estructuras concurrentes eficientes.
- Locks de corta duración.
- Arquitectura descentralizada.

## 9.2. Supuestos de simulación y métricas clave

Supuestos:

- 2 operadores generan un incidente aproximadamente cada 300–700 ms.
- 2 despachadores procesan incidentes con pasos programados (1 s, 3 s, 5 s).
- Flota disponible: 8 ambulancias, 12 médicos, 6 equipos.

Métricas principales:

- **Throughput de resolución:** limitado por la duración de la “misión” (≈5 s) y la escasez relativa de equipos.
- **Latencia de asignación:** suma del tiempo en cola y el tiempo de reserva bajo contención.
- **Cola pendiente:** indicador de backlog y de si la capacidad es suficiente.

## 9.3. Resultados por carga

Resumen de comportamiento por carga:

**Baja** (≈1–2 incidentes/s)

- Recursos sobran.
- Latencia de asignación ≈ 0.
- Backlog prácticamente nulo.
- Métricas muy estables.

**Media** (≈3–4 incidentes/s)

- Recurso crítico: equipos (6).
- Latencia de asignación: 100–300 ms.
- Backlog bajo.
- Ocasional backoff, mitigado por el aumento de prioridad ligado al tiempo de espera.

**Alta** (≈6–8 incidentes/s)

- Recursos críticos: equipos y ambulancias.
- Latencia de asignación: 300–800 ms o más.
- Backlog medio/alto.
- Reencolado frecuente; el tiempo de espera incrementa la prioridad, evitando starvation, aunque la cola puede crecer en picos.

**Cuellos de botella identificados:**

- “pickAvailableClosest” realiza un escaneo lineal sobre la flota, por lo que el coste crece con el tamaño de la misma.
- La cantidad limitada de equipos (6) restringe el paralelismo efectivo; misiones de 5 s ocupan los recursos por periodos relativamente largos.
- Aumentar el número de despachadores incrementa la contención en los locks de los recursos.

## 10. Consideraciones de seguridad

Aunque se trata de una simulación, se han aplicado principios de desarrollo de software seguro, especialmente en el manejo de concurrencia:

- **Integridad de datos:**

  Uso de AtomicReference y LongAdder para estados compartidos y métricas.

  Esquema de locking jerárquico que evita inconsistencias al reservar múltiples recursos.

- **Disponibilidad:**

  tryLock(timeout) evita bloqueos indefinidos que podrían dejar el sistema inutilizable.

  Reencolado de incidentes ante fallos de reserva mantiene el flujo del sistema.

- **Confidencialidad (potencial):**

  En un entorno real, se evitaría registrar información sensible de pacientes en texto plano.

El diseño puede integrarse con mecanismos de autenticación y autorización para operadores y sistemas externos.

- **Resiliencia bajo carga:**

  El análisis de rendimiento muestra que el sistema se degrada de forma controlada (aumento de latencia y backlog) sin caer en estados inestables.

- **Validación de datos (extensible):**

  Las coordenadas, severidades y descripciones de incidentes pueden someterse a validación previa para evitar datos corruptos o malintencionados en un entorno real.

## 11. Conclusiones y lecciones aprendidas

### Conclusiones

El sistema resulta robusto y escalable gracias al uso de estructuras concurrentes como PriorityBlockingQueue, ReentrantLock y tipos atómicos, que permiten manejar múltiples hilos sin comprometer la integridad de los datos.

La estrategia de prioridad ponderada permite balancear de forma flexible severidad, tiempo de espera y distancia, adaptándose a diferentes escenarios de carga.

El uso estricto de un orden de bloqueo evita interbloqueos incluso con múltiples despachadores.

El sistema maneja eficientemente condiciones de alta carga mediante re encolado seguro y backoff, manteniendo la estabilidad y evitando pérdida de incidentes.

### Lecciones aprendidas

La sincronización explícita debe ser diseñada cuidadosamente; un simple cambio en el orden de adquisición puede desencadenar deadlocks.

Evitar bloqueos prolongados (uso de `tryLock(timeout)`) mejora significativamente la capacidad de respuesta.

Utilizar estructuras lock-free (tipos atómicos, CopyOnWrite) reduce contención en sistemas altamente concurrentes.

El monitoreo en tiempo real es clave para verificar estabilidad y tiempos de respuesta.