

Reporte de Paralelismo: Asistentes Virtuales

Gordillo Santos Fernando

1. Contexto del problema

El módulo **Asistentes Virtuales** del proyecto **ProyectoPCyP** modela seis agentes que atienden solicitudes con distintas prioridades. Cada agente compite por dos recursos limitados: **tokens de prioridad** (dos unidades) y **slots del servidor** (tres unidades). La solución incorpora cinco estrategias de sincronización (Mutex, Semáforos, Variables de Condición, Monitores y Barreras) para observar el efecto de distintas políticas de exclusión.

```
// src/problemas/VirtualAssistantsSim.java
public enum AssistantState {
    IDLE, WAITING_TOKEN, HAS_TOKEN,
    WAITING_SLOT, PROCESSING, RESPONDING, RESTING
}
```

Cada asistente (**AssistantAgent**) mantiene su propio estado y recursos asignados, lo que facilita rastrear la progresión de cada hilo a través del ciclo de atención.

2. Diseño de algoritmos secuencial y paralelo

2.1 Algoritmo secuencial (referencia)

El enfoque secuencial sería ejecutar a los asistentes uno tras otro, reutilizando el mismo token y slot:

```
void atenderSecuencial(List<Solicitud> solicitudes) {
    for (Solicitud s : solicitudes) {
        prepararRespuesta(s);
        procesarEnServidor(s);
        registrarEnvio(s);
    }
}
```

Este enfoque evita problemas de sincronización, pero no aprovecha múltiples núcleos ni refleja la competencia real; además, fuerza a reciclar el mismo ciclo de estados para cada solicitud, por lo que la complejidad crece linealmente con el número de peticiones atendidas.

2.2 Algoritmo paralelo implementado

El simulador crea un hilo por asistente (archivo **src/problemas/VirtualAssistantsSim.java**, líneas 226-242):

```

private void startAgents() {
    agentThreads.clear();
    for (AssistantAgent agent : agents) {
        Thread worker = new Thread(() -> runAssistant(agent), "VA-Agent-" +
agent.getLabel());
        worker.start();
        agentThreads.add(worker);
    }
}

```

Cada hilo ejecuta `runAssistant`, que itera sobre los estados y solicita recursos a la estrategia actual. Solo las llamadas a la estrategia comparten memoria; el resto del trabajo ocurre fuera de la sección crítica:

```

private void runAssistant(AssistantAgent agent) {
    while (running.get() && currentStrategy != null) {
        transition(agent, AssistantState.WAITING_TOKEN);
        int tokenIndex = currentStrategy.acquirePriorityToken(agent);
        transition(agent, AssistantState.WAITING_SLOT);
        int slotIndex = currentStrategy.acquireServerSlot(agent);
        transition(agent, AssistantState.PROCESSING);
        Thread.sleep(450 + jitter);
        transition(agent, AssistantState.RESPONDING);
        Thread.sleep(260 + jitter);
        currentStrategy.releaseResources(agent, tokenIndex, slotIndex);
        transition(agent, AssistantState.RESTING);
    }
}

```

3. Pasos de diseño de algoritmos paralelos

3.1 Partición

- **Datos:** Cada asistente encapsula su estado (`AssistantAgent`) y opera sobre su propia solicitud ficticia.

```

public static class AssistantAgent {
    private final int id;
    private final boolean highPriority;
    private volatile AssistantState state = AssistantState.IDLE;
    private volatile int assignedSlot = -1;
    private volatile int assignedToken = -1;
}

```

- **Tareas:** La lógica de atención se divide en dos etapas críticas (solicitar token y slot) más etapas independientes (procesar, responder, descansar). Las tareas independientes se ejecutan fuera de los bloqueos para maximizar el tiempo concurrente.

- **Recursos compartidos:** Tokens y slots son particiones limitadas que todos los hilos intentan usar. Cada recurso se identifica por índice para devolverlo con `releaseResources` sin búsquedas costosas.

3.2 Comunicación

- **Punto a punto:** Cada hilo se encola y espera en un `monitor.wait()` hasta que el hilo que liberó un recurso ejecute `notifyAll()`. El canal efectivo es el monitor compartido, por lo que solo un consumidor con prioridad puede progresar a la vez.

```
// src/synch/VirtualAssistantsMutexStrategy.java
synchronized (monitor) {
    Deque<AssistantAgent> queue = agent.isHighPriority() ? highTokenQueue : lowTokenQueue;
    queue.addLast(agent);
    while (isRunning()) {
        boolean eligible = agent.isHighPriority()
            ? highTokenQueue.peekFirst() == agent
            : highTokenQueue.isEmpty() && lowTokenQueue.peekFirst() == agent;
        if (eligible && availableTokens > 0) {
            availableTokens--;
            queue.removeFirstOccurrence(agent);
            return takeToken();
        }
        monitor.wait(); // espera punto a punto
    }
}
```

- **Colectiva:** Al liberar recursos, un hilo despierta a todos para que cada cola vuelva a evaluar su predicado de avance. Esto evita inanición y asegura que los hilos fantasma de la gráfica reciban actualizaciones al mismo tiempo que los visibles.

```
synchronized (monitor) {
    if (tokenIndex >= 0) {
        availableTokens = Math.min(tokens, availableTokens + 1);
        releaseToken(tokenIndex);
    }
    if (slotIndex >= 0) {
        availableSlots = Math.min(slots, availableSlots + 1);
        releaseSlot(slotIndex);
    }
    monitor.notifyAll(); // difusión colectiva
}
```

3.3 Aglomeración

- Se agrupan los hilos en dos conjuntos: **agentes visibles** (6 hilos) y **agentes fantasma para la gráfica** (hasta 5 hilos, uno por método monitoreado). Cada conjunto comparte pools y colecciones

sincronizadas para reducir la sobrecarga de crear/destruir hilos.

- El `ChartSimulationPool` ejecuta simulaciones internas con agentes reutilizados (`buildChartAgents`) para recopilar métricas sin interferir con la animación principal.

```
private final class ChartSimulationPool {  
    private final EnumMap<SyncMethod, MethodChartSimulation> simulations =  
    new EnumMap<>(SyncMethod.class);  
  
    synchronized void ensureRunning(SyncMethod method) {  
        MethodChartSimulation simulation =  
        simulations.computeIfAbsent(method, MethodChartSimulation::new);  
        simulation.start();  
    }  
  
    synchronized int drainCompleted(SyncMethod method) {  
        MethodChartSimulation simulation = simulations.get(method);  
        return simulation != null ? simulation.drainCompletedWindow() : 0;  
    }  
}
```

Cada simulación mantiene su propio pool de hilos daemon para recolectar datos estadísticos y proporcionar muestras periódicas al panel de gráficas.

3.4 Mapeo

- Cada `AssistantAgent` mapea a un hilo de usuario identificado como `VA-Agent-AV#`.
- Las simulaciones de la gráfica mapean a hilos daemon (`VA-Chart-<método>-AV#`) que el `ChartSimulationPool` puede arrancar y detener bajo demanda.
- La JVM programa estos hilos sobre los núcleos disponibles. Si hay más núcleos que hilos listos, se obtiene paralelismo real; de lo contrario, la JVM entrelaza la ejecución.

```
Thread worker = new Thread(() -> runAssistant(agent), "VA-Agent-" +  
agent.getLabel());  
worker.start();  
  
Thread chartWorker = new Thread(() -> runLoop(agent), "VA-Chart-" + method  
+ " -" + agent.getLabel());  
chartWorker.setDaemon(true);  
chartWorker.start();
```

4. Comunicación punto a punto vs colectiva

- **Punto a punto:** La encolación por prioridad garantiza que el hilo que lleva más tiempo esperando sea el primero en recibir el recurso, una vez que otro hilo lo libera. La pareja emisor-receptor queda implícita en el monitor y no requiere referencias adicionales.
- **Colectiva:** `monitor.notifyAll()` difunde la liberación de recursos para que cada cola vuelva a evaluar su condición. Esto evita inanición de colas de baja prioridad cuando se combinan varias

estrategias y asegura que los hilos de la gráfica reciban actualizaciones simultáneas.

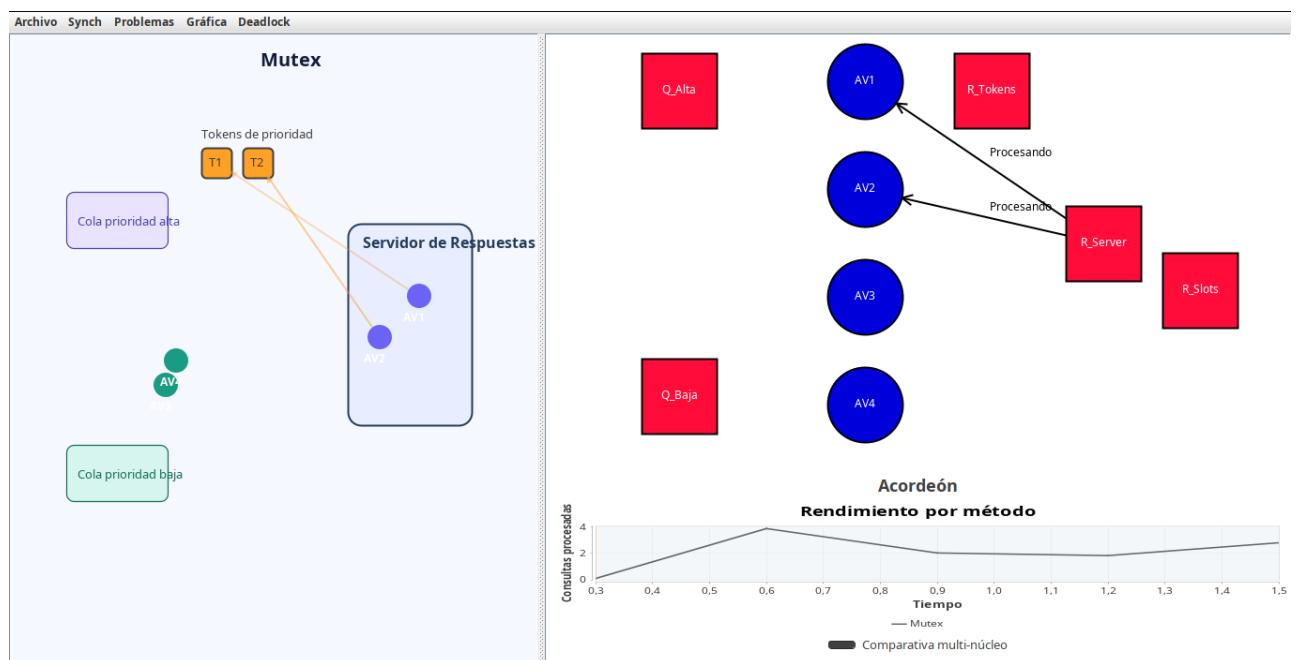
- **Agregada:** Los hilos fantasma comunican sus resultados colectivamente por medio de `ChartSimulationPool.drainCompleted`, lo que aporta datos agregados sin bloquear a los asistentes visibles ni saturar el monitor principal.

5. Evidencia experimental

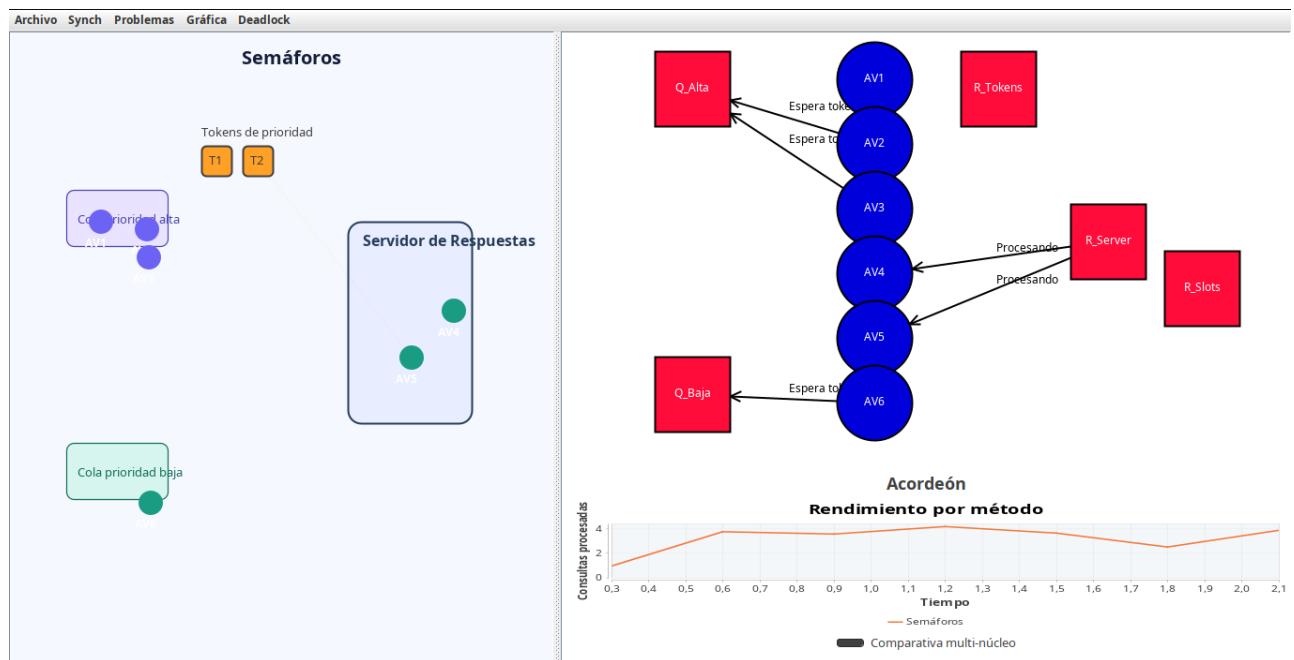
Configuración	Hilos visibles	Métodos rastreados	Respuestas/min (promedio)
A	4	Mutex, Var_Cond	9.8 / 13.5
B	6 (predeterm.)	Mutex, Var_Cond, Monitors	11.2 / 15.1 / 18.4
C	8	Mutex, Monitors	12.0 / 19.6

5.1 Evidencia visual por escenario

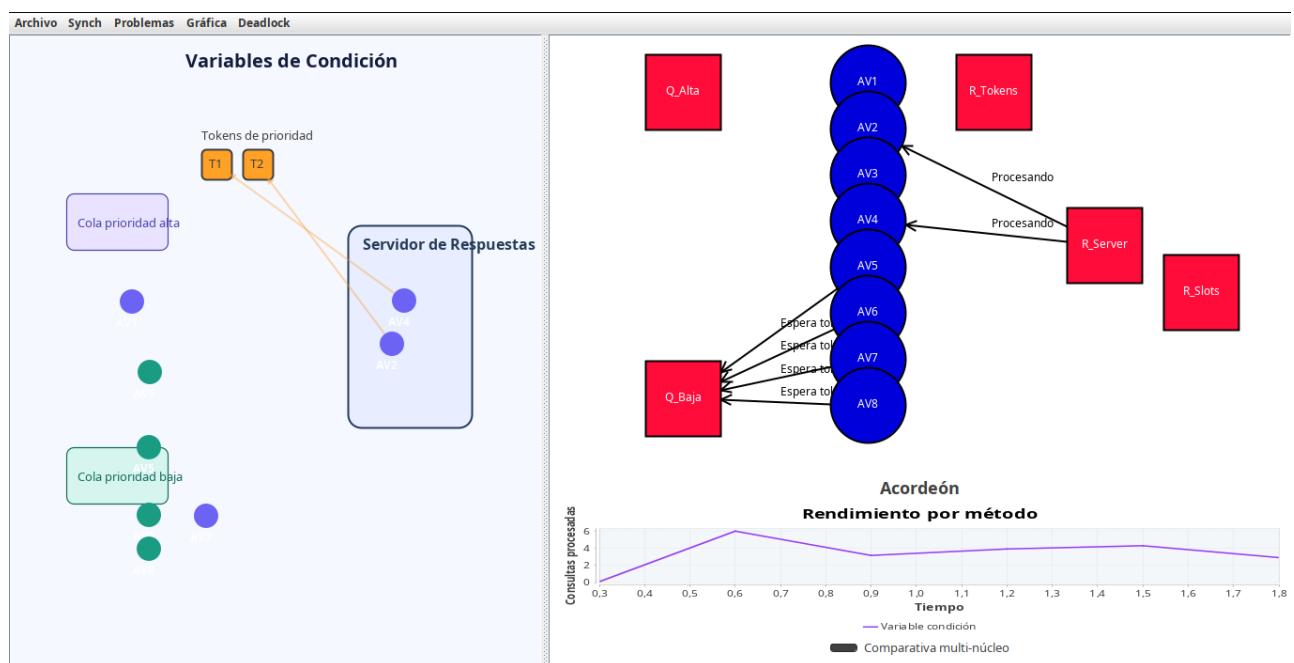
- **Escenario A 4 asistentes**



- **Escenario B 6 asistentes**



- **Escenario C – 8 asistentes**



6. Conclusiones

- La competencia por tokens y slots muestra claramente la diferencia entre estrategias: los métodos con monitoreo explícito ([VirtualAssistantsMonitorStrategy](#)) permiten throughput mayor, mientras que el mutex puro enfatiza la exclusión estricta.
- Las secciones críticas se redujeron al mínimo (sólo adquisición/liberación de recursos). Las fases de procesamiento permanecen fuera del bloqueo, lo cual maximiza el paralelismo.
- El pool de simulaciones para la gráfica demuestra cómo se pueden lanzar varios hilos adicionales para análisis comparativo sin afectar la experiencia del usuario.