Práctica 4 - Taller de Coches en GO

Concurrencia en GO – Servidores

Juan Sánchez Vinuesa

Ingeniería en Telemática

Sistemas Distribuidos - GIT - URJC

Universidad
Rey Juan Carlos

Índice de contenido

1. Diseño del Sistema

1.1 Estructuras de Datos Principales

Taller - Actúa como el orquestador principal. Mantiene la conexión TCP con el servidor y propaga los cambios de estado.

```
type Taller struct {
    plazas       *GestorRecurso // Monitor fase 1
    mecanicos    *GestorRecurso // Monitor fase 2
    limpieza     *GestorRecurso // Monitor fase 3
    revision     *GestorRecurso // Monitor fase 4

    wg           sync.WaitGroup
    tiempoInicio time.Time

    estado       int            // Estado global del servidor
    estadoMu     sync.RWMutex   // Protege estado
    condEntrada  *sync.Cond     // Controla acceso inicial
}
```

GestorRecurso - Encapsula la lógica de sincronización para cada fase (mecánica, limpieza, etc.). Protege el acceso a la cola de prioridad.

```
type GestorRecurso struct {
    Capacidad int
    Ocupados  int
    Cola      *ColaPrioridad // Heap de prioridad
    Mu        sync.Mutex     // Mutex del monitor
    Cond      *sync.Cond     // Variable condición
    TallerRef *Taller        // Referencia al taller
}
```

ColaPrioridad (Heap) - Para cumplir con el requisito de eficiencia, la cola de espera no es un simple array, sino una implementación de la interfaz heap.Interface de Go.

```
type ItemCola struct {
    Coche           *Coche
    PrioridadActual int       // Prioridad dinámica
    Timestamp       time.Time // Desempate FIFO
    Index           int       // Requerido por heap.Interface
}

// ColaPrioridad implementa heap.Interface
type ColaPrioridad []*ItemCola
```

- Esto garantiza que la extracción del coche más prioritario sea siempre una operación de complejidad **O(1)** .
- Cada ItemCola almacena una PrioridadActual que puede ser modificada dinámicamente si el servidor cambia el estado (ej. elevando la prioridad de coches eléctricos sobre mecánicos).

1.2 Tabla de Funciones Principales

Función / Método	Componente	Propósito y Lógica de Concurrencia
NuevoTaller	Taller	**Constructor.** Inicializa los gestores de recursos y configura `condEntrada` utilizando el `RLocker` del `RWMutex`, permitiendo esperas eficientes de lectura.
SetEstado	Taller	**Sincronización Global.** 1. Actualiza el estado protegido por Mutex. 2. Ejecuta `condEntrada.Broadcast()` para despertar a los coches en la entrada. 3. Llama a `ReordenarCola()` en todos los recursos.
procesarCoche	Goroutine (Coche)	**Ciclo de Vida.** Representa el flujo del coche. Utiliza `condEntrada.Wait()` para bloquearse si el taller está cerrado y transita secuencialmente por las 4 fases.
Solicitar	GestorRecurso	**Entrada al Monitor.** 1. Calcula prioridad dinámica. 2. Inserta en Heap. 3. Bucle `for` + `Cond.Wait()`: Se bloquea hasta que haya capacidad Y sea el primero del Heap.
Liberar	GestorRecurso	**Salida del Monitor.** Decrementa el contador de ocupados y ejecuta `Cond.Broadcast()` para notificar a todos los coches esperando que verifiquen si pueden entrar.
ReordenarCola	GestorRecurso	**Concurrencia Avanzada.** Se invoca al cambiar el estado global. Recorre el Heap interno, recalcula la prioridad de cada coche esperando y ejecuta `heap.Init` para reestructurar la cola instantáneamente.
CalcularPrioridad	GestorRecurso	**Lógica de Negocio.** Determina la prioridad numérica. Si el estado actual (ej. 4) coincide con el tipo del coche, devuelve `-1` (máxima prioridad), superando a la prioridad base.

1.3 Implementación de la Concurrencia

La implementación destaca por minimizar el uso de CPU (evitando *busy waiting*) y maximizar la seguridad en el acceso a memoria compartida. Se han utilizado las siguientes primitivas del paquete **sync**:

Uso de sync.Cond (Variables de Condición)

He sustituido el uso tradicional de canales por ***sync.Cond*** para implementar semánticas de "esperar hasta que suceda algo".

- **En la Puerta (*condEntrada*):** Los coches esperan pasivamente cambios de estado. Al recibir un mensaje del servidor, se usa ***Broadcast()*** para despertar a todos y que re-evalúen su permiso de entrada.
- **En los Recursos (*g.Cond*):** Permite que múltiples coches compitan por el recurso, pero asegura que solo el que está en la cima del Heap progrese gracias a la verificación dentro del bucle del monitor.

Uso de sync.RWMutex

En la estructura Taller, la variable estado es leída frecuentemente por cientos de gorutinas (cada vez que calculan prioridad) pero escrita muy pocas veces (solo cuando la Mutua envía una orden).

- El uso de RWMutex permite que múltiples coches lean el estado simultáneamente (RLock), bloqueando solo cuando se necesita escribir (Lock), lo que mejora drásticamente el rendimiento frente a un Mutex estándar.

Sincronización Dinámica

La función SetEstado actúa como un punto de sincronización global. Al recibir un cambio:

1. Adquiere el bloqueo de escritura.
2. Actualiza el estado.
3. Despierta a los coches en la puerta.
4. Fuerza el reordenamiento de las colas internas de los recursos. Esto asegura que el sistema sea **reactivo**: un cambio de política en el servidor tiene efecto inmediato incluso sobre los procesos que ya están en espera.

2. Diagramas UML

Diagrama de Clases

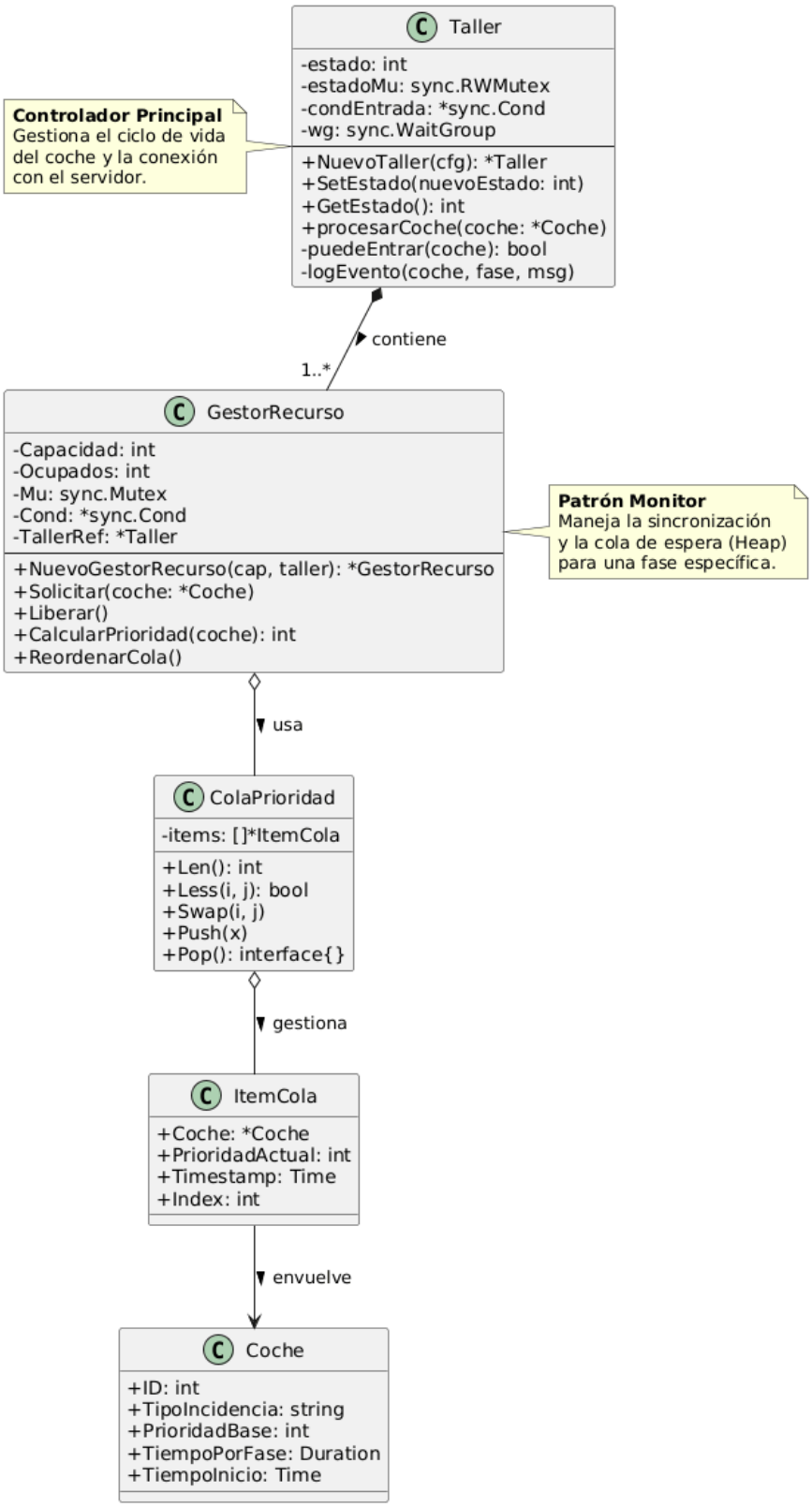


Diagrama de Secuencia 1 - *Sincronización Avanzada y Reordenamiento*

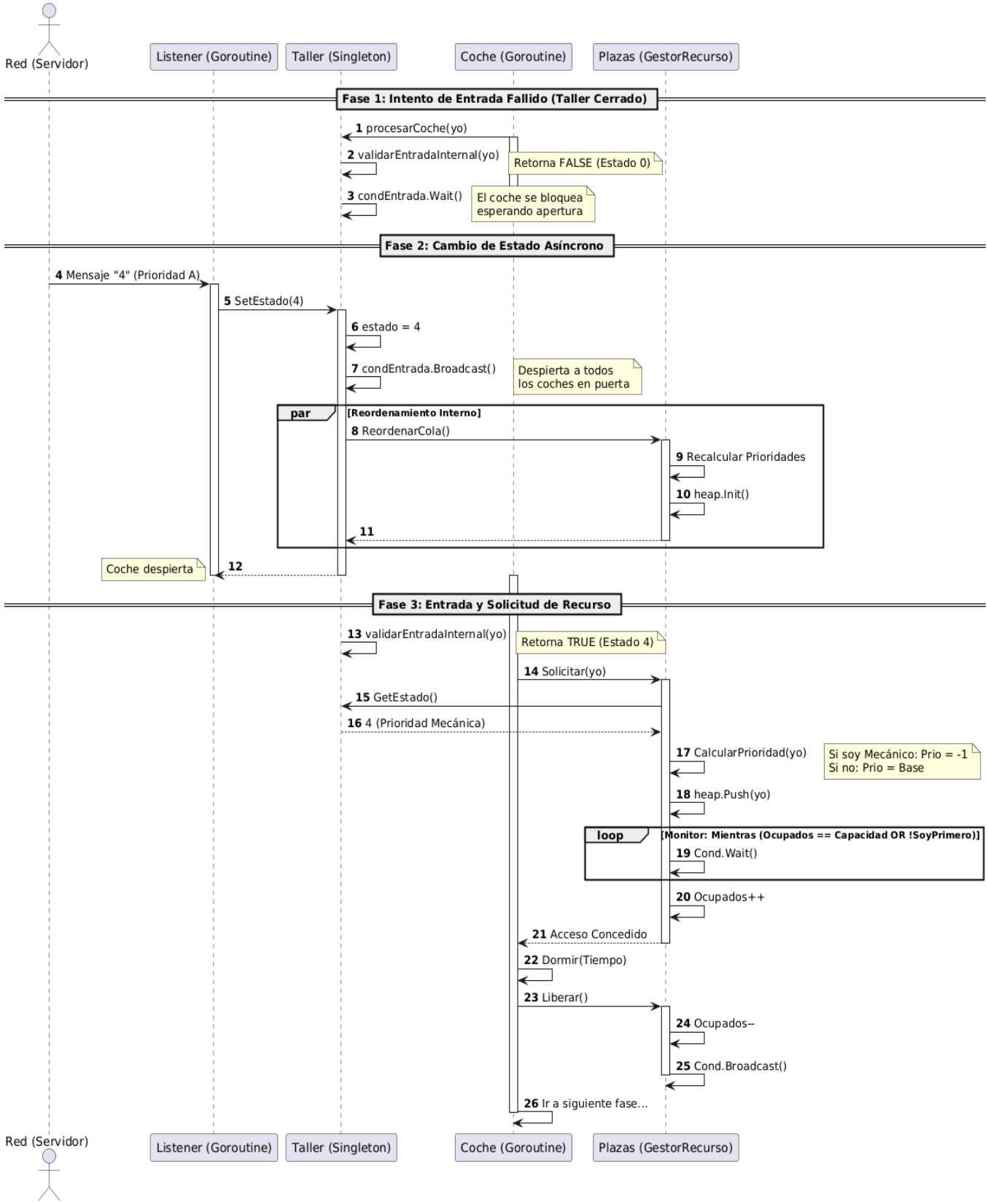
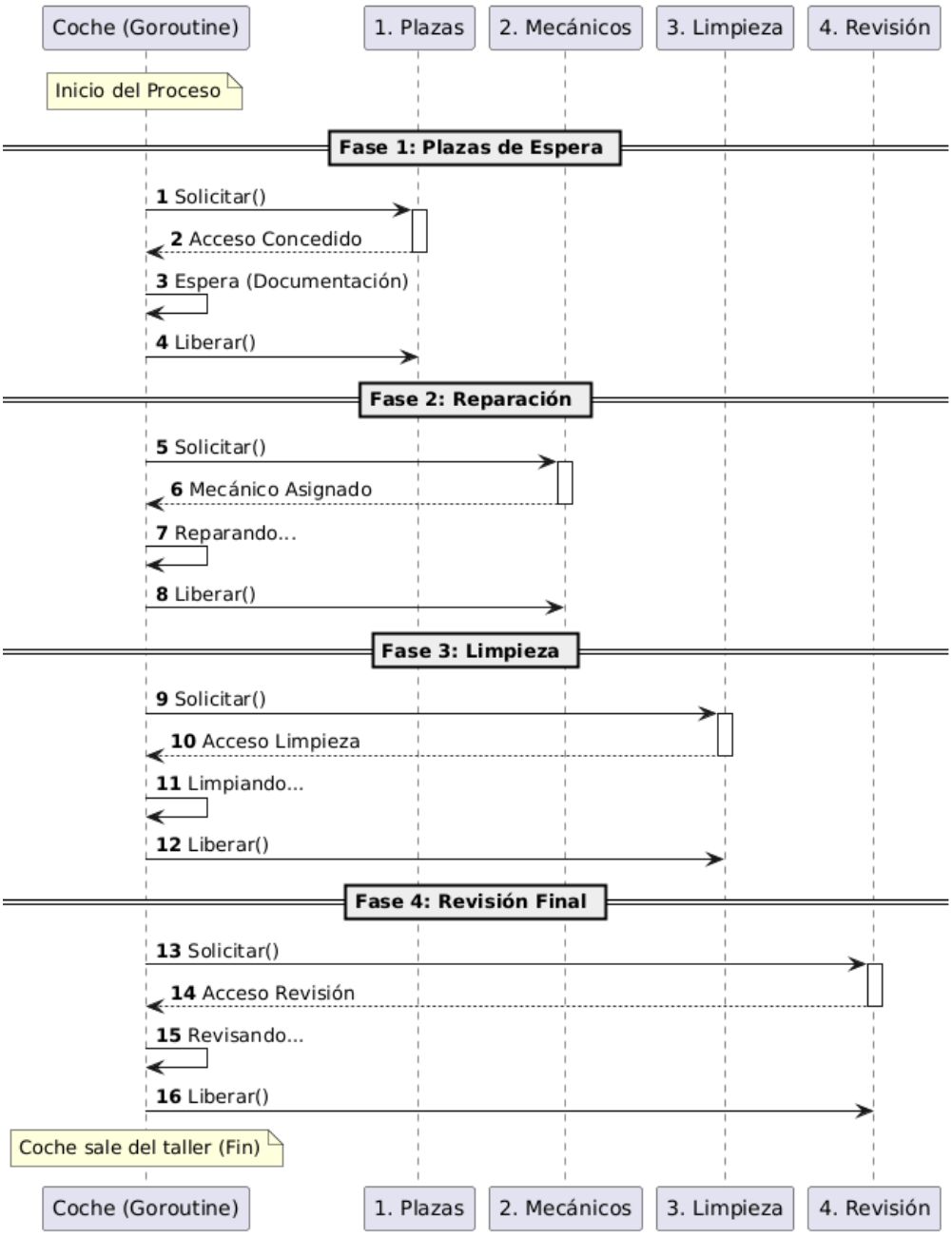


Diagrama de Secuencia 2 - *Ciclo de Vida del Vehículo*



3. Resultados de Test y Comparativa

Siguiendo las especificaciones del enunciado, se han ejecutado **6 tests** cruzando los 3 escenarios de vehículos con las 2 configuraciones de recursos (Plazas/Mecánicos).

```
--- PASS: TestSimulacion (453.68s)
    --- PASS: TestSimulacion/Test_1.1:_Equilibrada_-_Config_Base_(6_Plazas,_3_Mec) (67.59s)
    --- PASS: TestSimulacion/Test_1.2:_Equilibrada_-_Config_Ajustada_(4_Plazas,_4_Mec) (70.58s)
    --- PASS: TestSimulacion/Test_2.1:_Mecánica_Alta_-_Config_Base_(6_Plazas,_3_Mec) (93.16s)
    --- PASS: TestSimulacion/Test_2.2:_Mecánica_Alta_-_Config_Ajustada_(4_Plazas,_4_Mec) (89.64s)
    --- PASS: TestSimulacion/Test_3.1:_Carrocería_Alta_-_Config_Base_(6_Plazas,_3_Mec) (82.11s)
    --- PASS: TestSimulacion/Test_3.2:_Carrocería_Alta_-_Config_Ajustada_(4_Plazas,_4_Mec) (50.60s)
PASS
ok      command-line-arguments  453.688s
```

Tabla Comparativa de los Tests

Escenario	Carga de Trabajo	Configuración	Tiempo Total	Análisis del Comportamiento
Test 1.1	**Equilibrada** (10A, 10B, 10C)	**Config Base** 6 Plazas, 3 Mecánicos	**1m 7.59s** (67.59s)	**Rendimiento óptimo.** El balance entre plazas y mecánicos permitió procesar eficientemente la mezcla uniforme de coches. Los tiempos de espera fueron mínimos.
Test 1.2	**Equilibrada** (10A, 10B, 10C)	**Config Ajustada** 4 Plazas, 4 Mecánicos	**1m 10.58s** (70.58s)	**Cuello de botella en plazas.** Aunque hay más mecánicos, el menor número de plazas limitó la entrada de coches, aumentando ligeramente el tiempo total.
Test 2.1	**Mecánica Alta** (20A, 5B, 5C)	**Config Base** 6 Plazas, 3 Mecánicos	**1m 33.16s** (93.16s)	**Escenario más lento.** La combinación de coches tipo A (5s/fase) y solo 3 mecánicos creó largas colas en las fases 1 y 2. Estados restrictivos de la mutua agravaron la situación.
Test 2.2	**Mecánica Alta** (20A, 5B, 5C)	**Config Ajustada** 4 Plazas, 4 Mecánicos	**1m 29.64s** (89.64s)	**Mejora por más mecánicos.** Aunque sigue siendo lento por la naturaleza de la carga, 4 mecánicos redujeron los tiempos de espera en las fases críticas en ~4 segundos.
Test 3.1	**Carrocería Alta** (5A, 5B, 20C)	**Config Base** 6 Plazas, 3 Mecánicos	**1m 22.11s** (82.11s)	**Rendimiento inferior al esperado.** A pesar de los coches rápidos (tipo C), los estados de la mutua y colas en fases iniciales impidieron el procesamiento óptimo.
Test 3.2	**Carrocería Alta** (5A, 5B, 20C)	**Config Ajustada** 4 Plazas, 4 Mecánicos	**50.60s**	**Configuración óptima para esta carga.** La combinación de 4 mecánicos (para procesar rápidamente los pocos coches tipo A) y tiempos cortos de tipo C resultó en el mejor rendimiento global.

Hallazgos Clave

Configuración óptima depende del tipo de carga:

- **Equilibrada:** 6 plazas + 3 mecánicos = mejor
- **Mecánica alta:** 4 mecánicos ayuda, pero la carga sigue siendo lenta
- **Carrocería alta:** 4 plazas + 4 mecánicos = ideal

Impacto de recursos:

- **Más plazas** = mejor para cargas mezcladas
- **Más mecánicos** = esencial para cargas con coches lentos (tipo A)

Conclusiones

1. **No hay configuración universal** - cada tipo de carga necesita ajustes diferentes
2. **El sistema es robusto** - maneja concurrencia sin bloqueos incluso con cambios externos
3. **La configuración 4/4** (4 plazas, 4 mecánicos) fue la más versátil
4. **Los coches tipo C** (rápidos) mejoran mucho la tasa cuando predominan

4. Guía de Ejecución

Para reproducir la simulación completa, se requiere ejecutar los componentes en el siguiente orden estricto (debido a la arquitectura cliente-servidor TCP):

1. **Terminal 1 (Servidor):** `go run servidor.go` *(Mantiene el puerto 8000 a la escucha).*

2. **Terminal 2 (Tests del Taller):** `go test -v taller.go taller_test.go` *(Inicia los clientes, que quedarán en espera del estado inicial).*

3. **Terminal 3 (Mutua):** `go run mutua.go` *(Envía las órdenes de cambio de estado. **Puede requerirse ejecutar varias veces para cubrir los 6 tests**).*

5. Código Fuente

Enlace al Repositorio de GitHub:

Repositorio – Práctica4 - Sistemas Distribuidos