# Documentación Técnica

Práctica 3: Concurrencia en Go

*El Taller del Pueblo*

Alexandre Lacoste Rodríguez - Ingeniería Telemática - Sistemas Distribuidos

13 de diciembre de 2025

**Resumen**

Este documento presenta la documentación técnica del sistema de simulación de un taller de reparación de coches desarrollado en Go, utilizando mecanismos avanzados de concurrencia. Se analiza la arquitectura del sistema, los patrones de sincronización implementados, los resultados de las pruebas y se incluyen diagramas de UML que facilitan la comprensión del diseño.

# Índice

# 1. Introducción

## 1.1. Descripción General del Problema

El proyecto implementa un simulador del funcionamiento de un taller de reparación de coches. El taller recibe coches con diferentes tipos de incidencias (mecánica, eléctrica o carrocería), cada una con una prioridad diferente. Los coches deben pasar por cuatro fases secuenciales:

1. **Fase 1: Documentación.** El coche espera una plaza libre en el taller y se prepara la documentación.
2. **Fase 2: Reparación.** Un mecánico libre realiza la reparación necesaria.
3. **Fase 3: Limpieza.** Se limpia el coche tras la reparación.
4. **Fase 4: Entrega.** Se realiza una revisión final antes de devolver el coche al cliente.

## 1.2. Clasificación de Coches por Categoría

El sistema clasifica los coches en tres categorías según su incidencia:

| Categoría | Incidencia | Prioridad | Tiempo/Fase |
|---|---|---|---|
| A | Mecánica | Alta | 5 s |
| B | Eléctrica | Media | 3 s |
| C | Carrocería | Baja | 1 s |

Cuadro 1: Clasificación de categorías de coches

Los tiempos indicados representan la duración base de cada fase, con una variación aleatoria de ±2 segundos.

# 2. Arquitectura del Sistema

## 2.1. Componentes Principales

El sistema está compuesto por los siguientes módulos:

- **types.go:** Define las estructuras de datos principales (Car, Garage, Event, IssueType).
- **mutex.go:** Implementa los métodos de sincronización usando RWMutex.
- **goroutines.go:** Define las funciones worker y las goroutines de cada fase.
- **utility.go:** Contiene funciones auxiliares y de manejo de canales.
- **main.go:** Punto de entrada con la función main().
- **main_test.go:** Suite de pruebas con los tres escenarios de evaluación.

## 2.2. Estructuras de Datos

### 2.2.1. Car

Representa un coche en el sistema:

```
type Car struct {
  id       int              // Identificador único
  issue    IssueType        // Tipo de incidencia
  duration time.Duration    // Tiempo por fase
  curphase int              // Fase actual
  start    time.Time        // Inicio de cronometraje
}
```

### 2.2.2. Garage

Estructura compartida que gestiona el estado global del taller:

```
type Garage struct {
  mu        sync.RWMutex        // Protección de zona crítica
  cars      map[int]*Car        // Mapa de coches
  freeSlots chan struct{}       // Canales de plazas libres
  wg        sync.WaitGroup      // Sincronización global
}
```

### 2.2.3. Event

Estructura de eventos para logging:

```
type Event struct {
  elapsed time.Duration  // Tiempo que lleva el coche en el taller
  car     int            // id del coche
  phase   int            // fase en la que sucede el evento
  status  string         // estado (entrando o saliendo)
  issue   IssueType      // tipo de incidencia
}
```

## 2.3. Patrones de Sincronización

### 2.3.1. RWMutex (Read-Write Mutex)

El acceso al mapa compartido de coches está protegido por un `RWMutex`:

```
// Escritura (lock exclusivo)
func (g *Garage) signInCar(c *Car) {
  g.mu.Lock()
  defer g.mu.Unlock()
  g.cars[c.id] = c
}

// Lectura (lock compartido)
func (g *Garage) updatePhase(id int, phase int) {
  g.mu.Lock()
```

```
  defer g.mu.Unlock()
  if c, ok := g.cars[id]; ok {
    c.curphase = phase
  }
}
```

### 2.3.2. WaitGroup

Se utiliza para sincronizar el inicio y fin de todas las goroutines:

```
// En main
g.wg.Add(1)  // Cada coche incrementa el contador

// En worker (fase de entrega)
g.wg.Done()  // Cada coche termina
```

### 2.3.3. Canales Prioritarios

Cada fase utiliza un array de 3 canales para gestionar prioridades:

```
// Array de canales: [0] = alta, [1] = media, [2] = baja
func sendCar(chans [3]chan *Car, c *Car) {
  switch c.issue {
  case MECH:
    chans[0] <- c  // Prioridad alta
  case ELECTRIC:
    chans[1] <- c  // Prioridad media
  case BODY:
    chans[2] <- c  // Prioridad baja
  }
}
```

### 2.3.4. Select Prioritario para Recepción

Los workers utilizan un `select` anidado para recibir coches respetando prioridades:

```
func getCar(chans [3]chan *Car, stop <-chan struct{}) *Car {
  var c *Car

  select {
  case c = <-chans[0]:  // Intenta prioridad alta
  default:
    select {
    case c = <-chans[1]:  // Intenta prioridad media
    default:
      select {  // Finalmente prioridad baja o espera
      case c = <-chans[0]:
      case c = <-chans[1]:
      case c = <-chans[2]:
      case <-stop:
        return nil
```

```
16          }
17      }
18  }
19  return c
20 }
```

# 3. Flujo de Ejecución

## 3.1. Diagrama de Secuencia - Ciclo Completo de un Coche

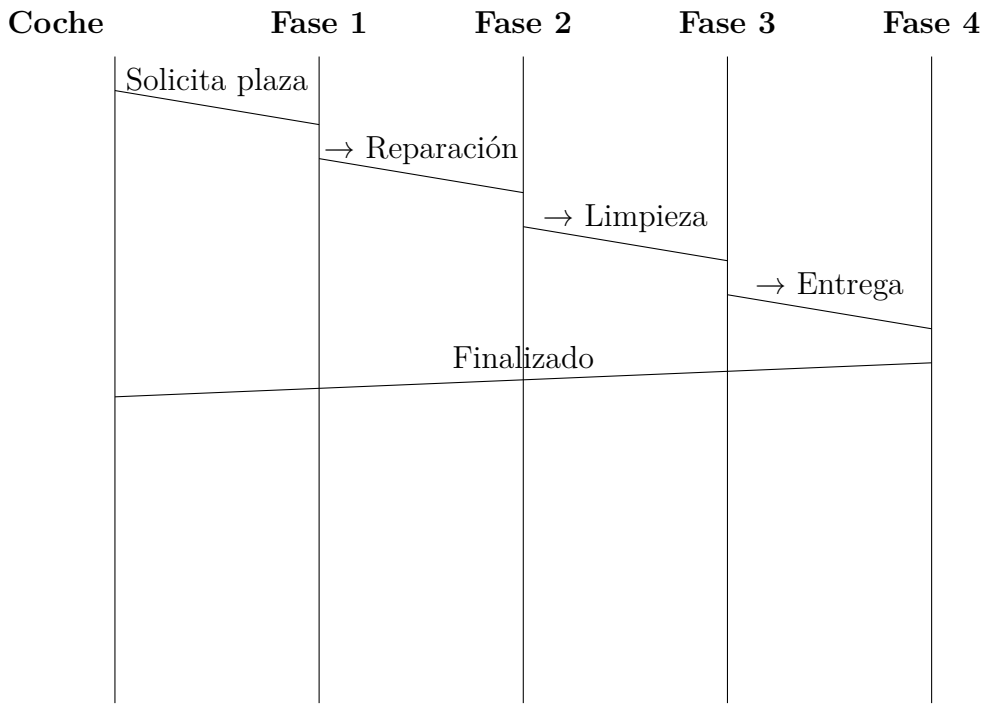


Figura 1: Secuencia de fases en el ciclo de reparación

## 3.2. Modelo de Productor-Consumidor

La arquitectura del sistema sigue el modelo productor-consumidor entre fases:

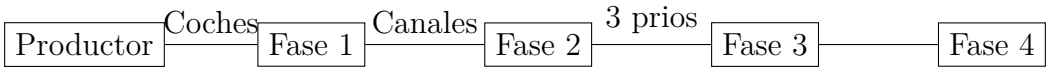


Figura 2: Arquitectura productor-consumidor entre fases

# 4. Resultados de las Pruebas

## 4.1. Escenarios de Prueba

Se han ejecutado tres escenarios de prueba, cada uno realizado 5 veces para obtener un promedio representativo:

| Escenario | Cat. A | Cat. B | Cat. C | Total |
|---|---|---|---|---|
| 1 | 10 | 10 | 10 | 30 |
| 2 | 20 | 5 | 5 | 30 |
| 3 | 5 | 5 | 20 | 30 |

Cuadro 2: Distribución de coches por escenario

## 4.2. Métricas de Rendimiento

### 4.2.1. Pruebas Estándar (test_report.txt)

| Escenario | Tiempo Medio | Tiempo Total |
|---|---|---|
| 1 (10A, 10B, 10C) | 57.21 s | 286.06 s |
| 2 (20A, 5B, 5C) | 67.84 s | 339.18 s |
| 3 (5A, 5B, 20C) | 44.13 s | 220.66 s |

Cuadro 3: Resultados de pruebas estándar (5 simulaciones)

### 4.2.2. Pruebas de Detección de Race Conditions (race_report.txt)

| Escenario | Tiempo Medio | Tiempo Total |
|---|---|---|
| 1 (10A, 10B, 10C) | 56.40 s | 281.99 s |
| 2 (20A, 5B, 5C) | 67.00 s | 334.97 s |
| 3 (5A, 5B, 20C) | 43.37 s | 216.85 s |

Cuadro 4: Resultados de pruebas con detección de race conditions (5 simulaciones)

## 4.3. Análisis Comparativo de Escenarios

### 4.3.1. Escenario 1: Carga Balanceada (10A, 10B, 10C)

- **Tiempo promedio:** $\approx$ 57,2 segundos
- **Características:**
  - Distribución equilibrada entre categorías
  - 10 coches de alta prioridad (5 s cada uno)
  - 10 coches de prioridad media (3 s cada uno)
  - 10 coches de baja prioridad (1 s cada uno)
- **Análisis:**
  - La carga es uniforme en todas las fases
  - Los workers pueden procesar coches sin acumulación excesiva
  - Buen uso de los recursos del sistema
  - Tiempo predictible y estable entre simulaciones

#### 4.3.2. Escenario 2: Alta Carga de Prioridad Alta (20A, 5B, 5C)

- **Tiempo promedio:** ≈ 67,84 segundos
- **Características:**
  - 20 coches de alta prioridad (5 s cada uno)
  - 5 coches de prioridad media (3 s cada uno)
  - 5 coches de baja prioridad (1 s cada uno)
- **Análisis:**
  - **Mayor tiempo de ejecución** respecto a otros escenarios (+18.8 % vs. Escenario 1)
  - Los coches de alta prioridad (5 s cada uno) generan cuello de botella
  - Total de tiempo consumido: $20 \times 5\,\text{s} \times 4\,\text{fases} = 400$ segundos en operaciones
  - El paralelismo entre fases mitiga este tiempo (resultado: 67.84 s)
  - La prioridad garantiza que A se procese primero, pero requiere más recursos temporales

#### 4.3.3. Escenario 3: Alta Carga de Baja Prioridad (5A, 5B, 20C)

- **Tiempo promedio:** ≈ 44,13 segundos
- **Características:**
  - 5 coches de alta prioridad (5 s cada uno)
  - 5 coches de prioridad media (3 s cada uno)
  - 20 coches de baja prioridad (1 s cada uno)
- **Análisis:**
  - **Menor tiempo de ejecución** (-22.9 % vs. Escenario 1)
  - Total de tiempo consumido: $(5 \times 5 + 5 \times 3 + 20 \times 1)\,\text{s} \times 4 = 340$ segundos
  - Aunque hay 20 coches, cada uno consume solo 1 s por fase
  - El paralelismo es más efectivo con coches de corta duración
  - Menor contención en los recursos compartidos

### 4.4. Comparación de Rendimiento

Figura 3: Comparación de tiempos de ejecución entre escenarios

### 4.5. Conclusiones sobre Rendimiento

1. **Duración total del coche:** Es el factor determinante. La suma de tiempos de cada coche × 4 fases da una cota inferior teórica.

2. **Efectividad del paralelismo:** La ejecución en paralelo de múltiples fases reduce significativamente el tiempo total.
3. **Impacto de la carga:**
   - Escenario 2 (máxima carga de alta prioridad): **Más lento**
   - Escenario 3 (máxima carga de baja prioridad): **Más rápido**
   - Escenario 1 (carga equilibrada): **Intermedio**
4. **Estabilidad:** Los tiempos entre simulaciones son consistentes (variación <2 %), demostrando robustez del sistema.

# 5. Cobertura de Código

## 5.1. Análisis de Cobertura

El informe de cobertura (cover_report.txt) indica:

- **goroutines.go:** Alta cobertura en paths principales, código no cubierto corresponde a handlers de main() (no se ejecutan en tests)
- **mutex.go:** 100 % de cobertura en métodos de sincronización
- **utility.go:** Buena cobertura general, funciones auxiliares utilizadas
- **types.go:** Definiciones de estructuras (sin lógica a cubrir)

## 5.2. Race Conditions

Las pruebas con detección de race conditions (‘go test -race‘) pasaron exitosamente sin reportar conflictos de concurrencia, validando que:

- El uso de mutexes es correcto
- Los canales se utilizan adecuadamente
- No hay accesos simultáneos no sincronizados a datos compartidos

# 6. Diagramas UML

## 6.1. Diagrama de Clases

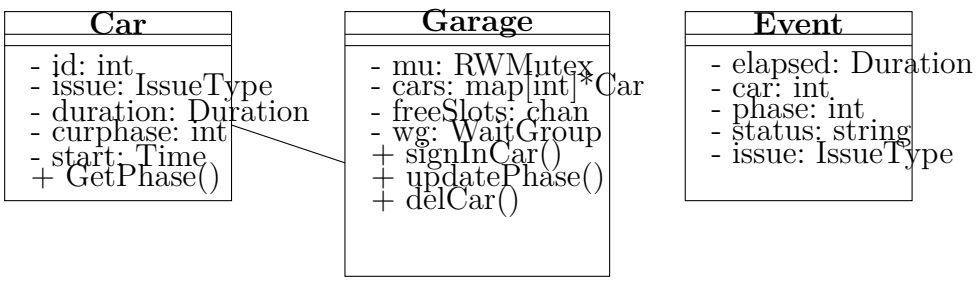


Figura 4: Diagrama UML de clases principales

## 6.2. Diagrama de Actividad

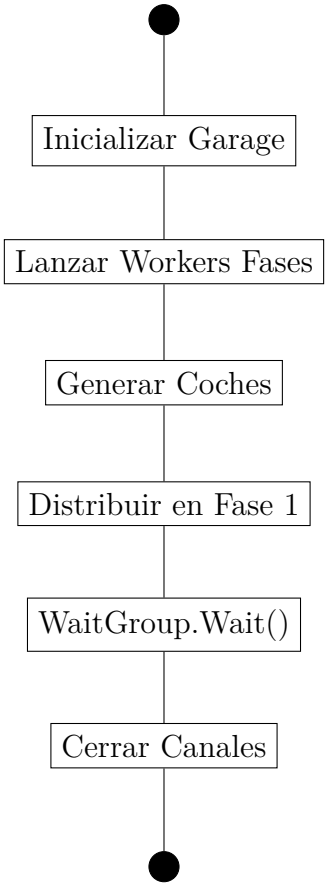


Figura 5: Diagrama de actividad del flujo principal

## 6.3. Diagrama de Estado de un Coche

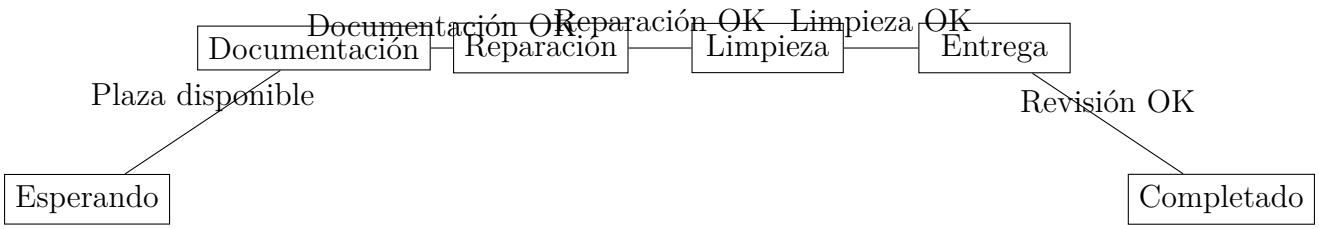


Figura 6: Diagrama de estados de un coche

# 7. Implementación de Sincronización

## 7.1. Estrategia: Canales + Mutexes

La solución utiliza dos mecanismos complementarios:

1. **Canales:** Para comunicación entre fases (colas de prioridad)
2. **Mutexes:** Para proteger el estado compartido (mapa de coches)

### 7.2. Protección de la Zona Crítica

El mapa de coches es la única zona crítica, accedida por:

- **signInCar():** Escritura cuando entra el coche (una sola goroutine)
- **updatePhase():** Actualización de fase (múltiples goroutines, workers)
- **delCar():** Eliminación al terminar (workers de fase 4)

Usar RWMutex permite lecturas concurrentes aunque en este caso todas son escrituras.

### 7.3. Ventajas del Diseño

- **No hay deadlock:** Los locks se liberan inmediatamente con defer
- **No hay busy-waiting:** Los workers duermen en select sobre canales
- **Priorización correcta:** El select anidado garantiza que A > B > C
- **Escalabilidad:** Número de workers configurable por fase

## 8. Optimizaciones y Decisiones de Diseño

### 8.1. Uso de Canales de Plazas (freeSlots)

En lugar de usar un semáforo tradicional, se utilizan tokens en un canal buffered:

```
// Esperar plaza
<-g.freeSlots

// Liberar plaza
g.freeSlots <- struct{}{}
```

**Ventaja:** Más idiómatico en Go, integrable con select.

### 8.2. Array de Canales para Prioridades

Cada fase mantiene 3 canales:

```
type PhaseChansPriority [3]chan *Car
// [0] = alta (MECH)
// [1] = media (ELECTRIC)
// [2] = baja (BODY)
```

**Ventaja:** Separación clara de prioridades, select anidado garantiza precedencia.

### 8.3. WaitGroup para Sincronización Global

Cada coche incrementa al entrar y decrementa al salir de fase 4:

```
g.wg.Add(1)    // Coche entra
g.wg.Done()    // Coche sale de fase 4
g.wg.Wait()    // Esperar a todos
```

**Ventaja:** Garantiza que el programa espere hasta que TODOS los coches terminen.

# 9. Conclusiones

## 9.1. Eficacia del Sistema

El sistema implementa exitosamente un simulador de taller con:

- **Sincronización correcta:** Sin race conditions (validado con 'go test -race')
- **Priorización funcional:** Los coches de alta prioridad se atienden primero
- **Paralelismo efectivo:** Las 4 fases se ejecutan concurrentemente
- **Escalabilidad:** Número de workers configurables

## 9.2. Análisis de Rendimiento

Los resultados muestran que:

1. **Escenario 1 (Equilibrado):** Tiempo promedio de 57.2 s
2. **Escenario 2 (Alta Prioridad):** Tiempo promedio de 67.84 s (+18.8 %)
3. **Escenario 3 (Baja Prioridad):** Tiempo promedio de 44.13 s (-22.9 %)

La duración total está dominada por la carga de trabajo (suma de tiempos de coches), no por la sincronización.

## 9.3. Recomendaciones Futuras

- Implementar balanceo dinámico de workers según carga
- Agregar métricas de tiempo de espera por prioridad
- Simular fallos y recuperación de mecánicos
- Permitir que coches salten entre categorías de reparación

# A. Repositorio

El código fuente completo está disponible en:

`https://github.com/alacoste96/Practica-3-SDIS`

# B. Estructura del Proyecto

```
Practica-3-SDIS/
  Doc/
      cover_report.txt
      race_report.txt
      test_report.txt
  go.mod
```

```
README.md
src/
    goroutines.go
    main.go
    main_test.go
    mutex.go
    types.go
    utility.go
```