



Práctica 2 - Taller de Coches en GO

Implementación usando *goroutines* y *channels*

Juan Sánchez Vinuesa

Ingeniería en Telemática

Sistemas Distribuidos - GIT - URJC

Índice de contenido

■ <u>1. Diseño del Sistema</u>	2
■ <u>Estructuras de Datos Principales</u>	2
■ <u>Funciones Principales</u>	3
■ <u>Diagramas de Secuencia UML</u>	4
■ <u>Implementación de GoRoutines y Channels</u>	7
■ <u>Guía de Ejecución</u>	9
■ <u>2. Análisis Comparativo de Resultados de los Tests</u>	11
■ <u>3. Código Fuente</u>	12

1. Diseño del Sistema

Estructuras de Datos Principales

🚗 **Coche** - Representa cada vehículo que llega al taller con su incidencia específica.

```
type Coche struct {
    Matricula      string
    ID             string
    TipoIncidencia TipoIncidencia
    TiempoAtendido time.Duration
    ChanTerminado chan bool
    TiempoLlegada time.Time
}
```

🔧 **Mecánico** - Cada mecánico es una goroutine independiente que procesa coches

```
type Mecanico struct {
    ID           string
    Especialidad TipoIncidencia
    Ocupado      bool
    ChanTrabajo  chan *Coche
    Trabajando   bool
    taller       *Taller
}
```

💻 **Taller** - Coordina todas las operaciones del sistema y gestiona el estado global.

```
type Taller struct {
    Cola          *Cola
    Mecanicos     [] *Mecanico
    ChanDetener   chan bool
    Stats         *Estadisticas
    running       bool
    mensajesBuffer []string
}
```

Cola de Espera - Gestiona la cola de espera con notificaciones eficientes.

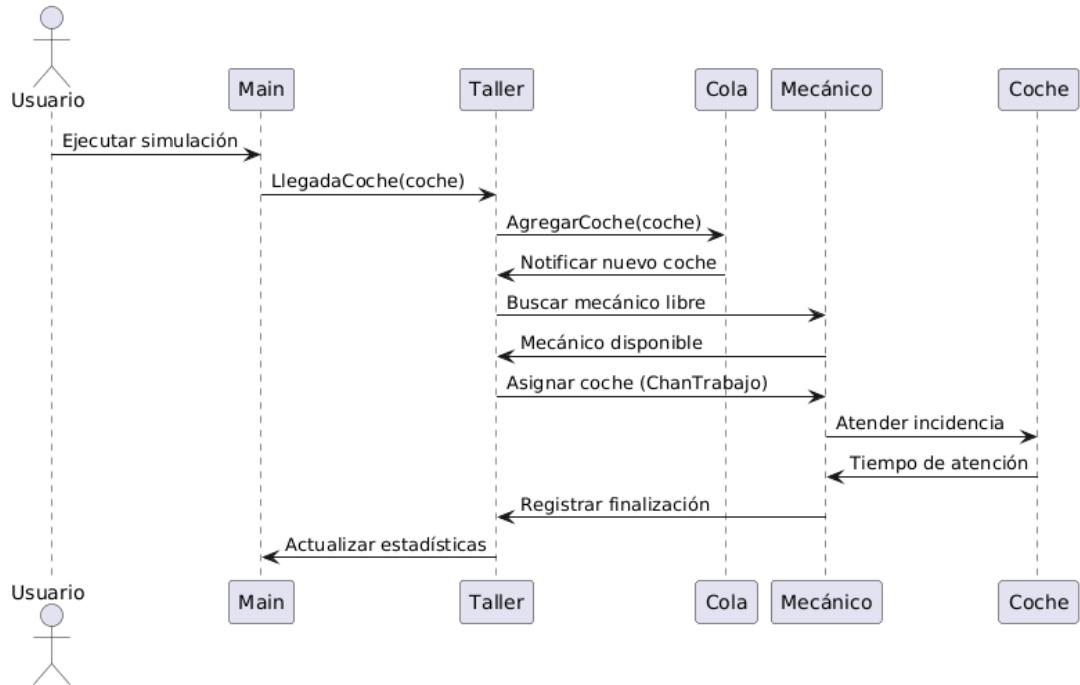
```
type Cola struct {
    coches    []*Coche
    mutex     sync.Mutex
    cerrada   bool
    notify    chan struct{}`}
```

Tabla de Funciones Principales

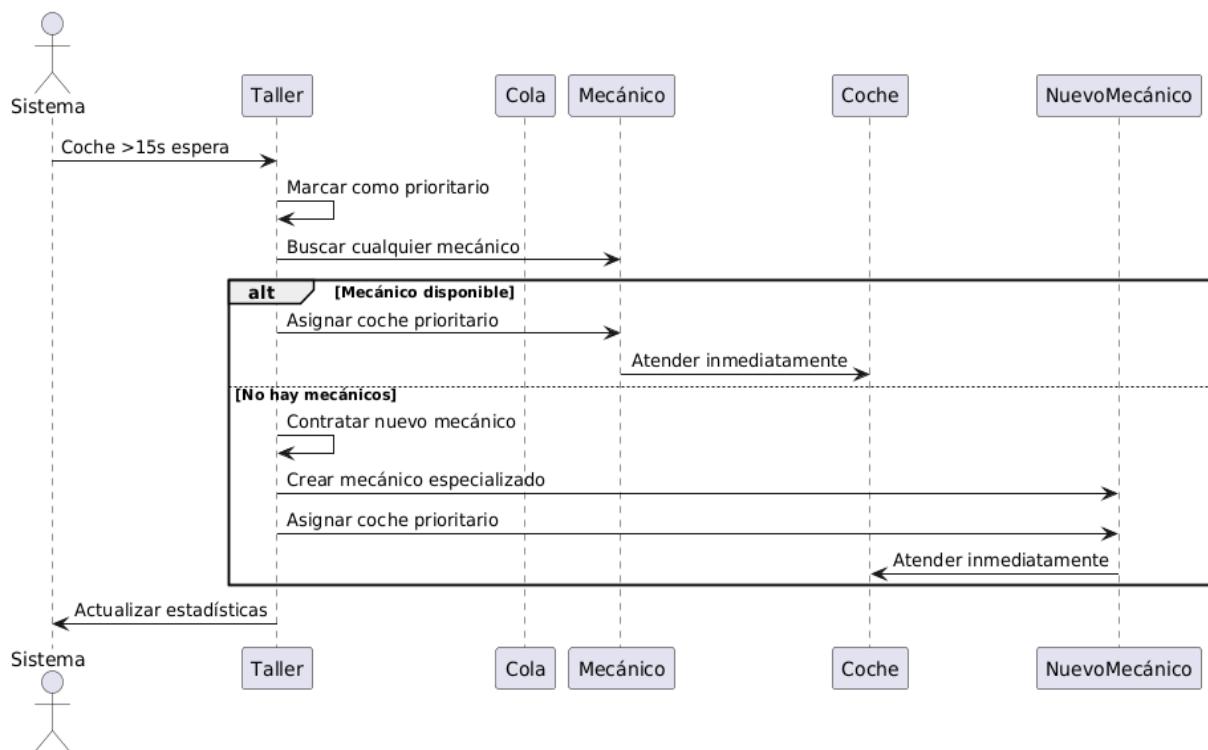
Función	Tipo	Propósito
NuevoTaller()	Constructor	Inicializa todo el sistema concurrente
(t *Taller) Iniciar()	Control	Lanza todas las goroutines del sistema
(t *Taller) coordinator()	Goroutine	Cerebro que coordina todo el flujo
(t *Taller) Detener()	Control	Cierre graceful de todas las goroutines
(c *Cola) ObtenerCoche()	Concurrencia	Extrae coches con patrón observer eficiente
(t *Taller) atiendeCochePrioritario()	Emergencia	Garantiza cero esperas indefinidas
(m *Mecanico) Iniciar()	Goroutine	Worker especializado concurrente
EjecutarSimulacion()	Simulación	Ejecuta pruebas automáticas completas
CrearConfiguracionAutomatica()	Configuración	Genera escenarios predefinidos para tests

Diagramas de Secuencia UML

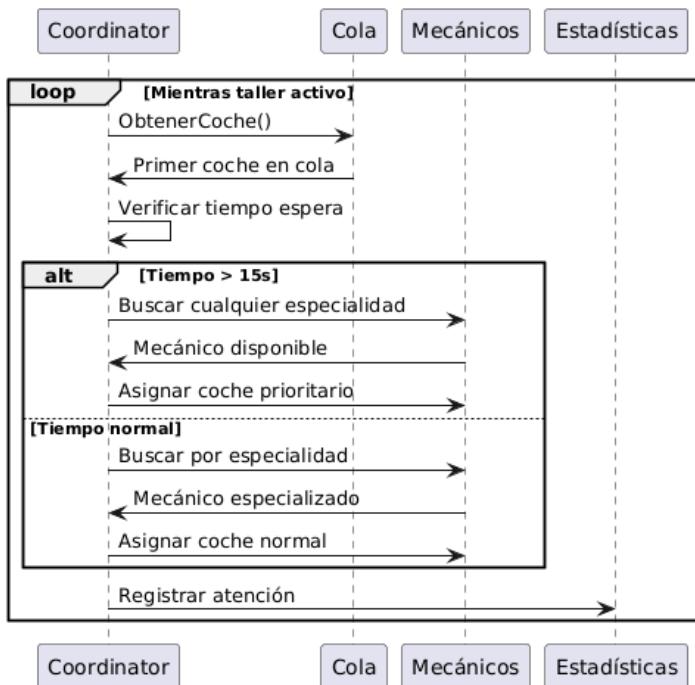
Llegada y Atención de Coche Normal



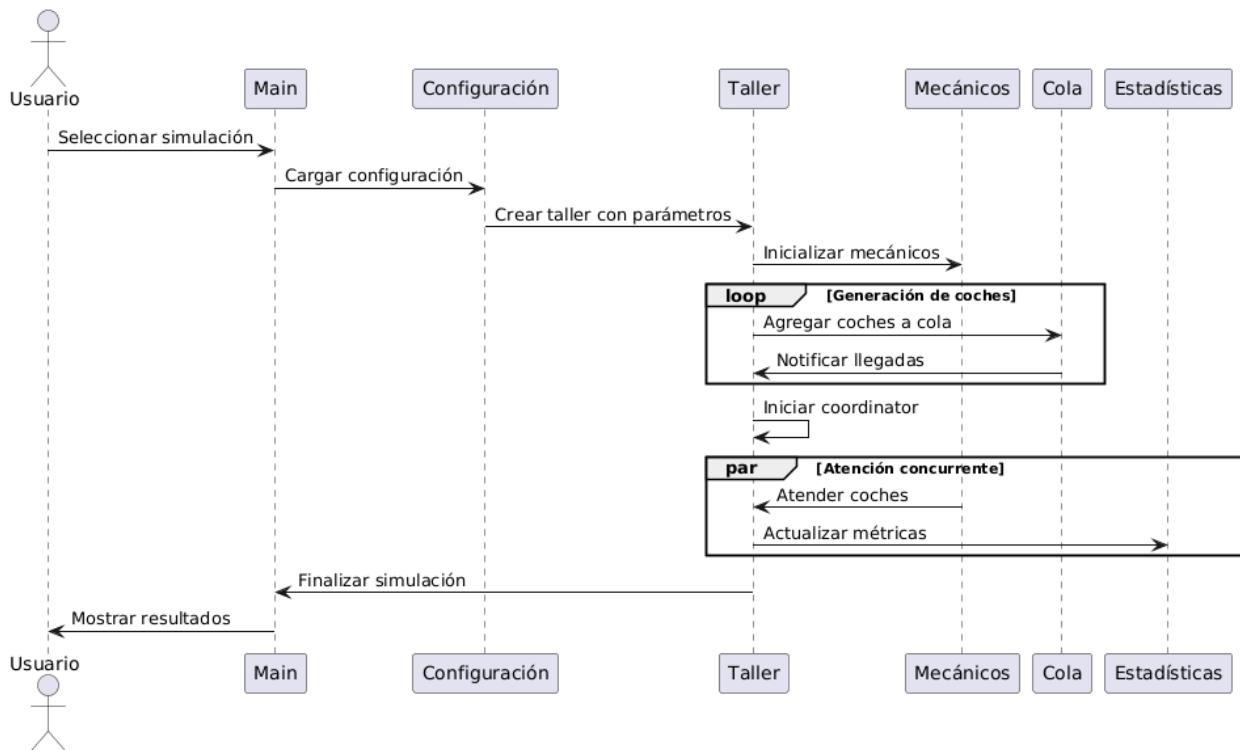
Atención Prioritaria y Contratación



Gestión de la Cola de Espera



Simulación Completa



Goroutines Implementadas

A) Goroutine del Coordinator:

```
// En taller.go
func (t *Taller) coordinator() {
    for t.running {
        coche := t.Cola.ObtenerCoche()
        if coche == nil { return }
        // Lógica de asignación...
    }
}
```

B) Goroutines de los Mecánicos:

```
// En mecanico.go
func (m *Mecanico) Iniciar(taller *Taller) {
    go func() {
        for coche := range m.ChanTrabajo {
            // Procesar coche...
            tiempoAtencion := coche.TiempoAtencion()
            time.Sleep(tiempoAtencion)
            // Registrar finalización...
        }
    }()
}
```

C) Goroutines Auxiliares para Re-encolado:

```
// En atiendeCocheNormal()
go func(c *Coche) {
    time.Sleep(waitTime)
    if t.running {
        t.Cola.AgregarCoche(c)
    }
}(coche)
```

Channels Implementados

A) Channel de Trabajo por Mecánico:

```
type Mecanico struct {
    ChanTrabajo chan *Coche // Channel buffered (tamaño 1)
}

// Uso: Asignar coche a mecánico
mecanico.ChanTrabajo <- coche
```

B) Channel de Notificación de Cola:

```
type Cola struct {
    notify chan struct{} // Channel para notificar nuevos elementos
}

// Uso: Notificar cuando hay coche nuevo
select {
case c.notify <- struct{}{}:
default: // Evita bloqueo si ya hay notificación
}
```

C) Channel de Control de Parada:

```
type Taller struct {
    ChanDetener chan bool // Para señalizar parada
}
```

Sincronización y Control

Inicio del Sistema:

```
func (t *Taller) Iniciar() {
    go t.coordinator() // Lanzar goroutine coordinador
    for _, m := range t.Mecanicos {
        m.Iniciar(t) // Lanzar goroutine por cada mecánico
    }
}
```

Parada Controlada:

```
func (t *Taller) Detener() {
    t.running = false
    t.Cola.Cerrar() // Cerrar cola primero
    close(t.ChanDetener) // Señalizar parada
    for _, m := range t.Mecanicos {
        m.Detener() // Cerrar channels de mecánicos
    }
}
```

6. Gestión de Concurrencia en la Cola

```
func (c *Cola) ObtenerCoche() *Coche {
    for {
        c.mutex.Lock()
        if len(c.coches) > 0 {
            coche := c.coches[0]
            c.coches = c.coches[1:]
            c.mutex.Unlock()
            return coche
        }
        c.mutex.Unlock()
        <-c.notify // Espera bloqueante hasta notificación
    }
}
```

Guía de Ejecución

El sistema ofrece múltiples formas de interacción: **gestión manual** para uso realista, **simulación automática** para análisis rápido, y **tests individuales** para desarrollo.

Menú Inicial - Terminal

```
==== TALLER MECÁNICO - PRÁCTICA 2 ====
1. Gestión Manual (Clientes, Vehículos, Incidencias, Mecánicos)
2. Ejecutar Simulación Automática
3. Simulación con Datos Actuales
4. Estado Actual del Taller
5. Ejecutar Tests
0. Salir

Seleccione opción: █
```

OPCIÓN 1: Gestión Manual (20-30 minutos)

1. Ejecutar el programa:

```
go run main.go
```

2. Seleccionar opción 1: "Gestión Manual"

3. Navegar por los submenús:

- a. **Clientes**: Crear, visualizar, modificar, eliminar
- b. **Vehículos**: Gestionar vehículos y asociar incidencias
- c. **Incidencias**: Gestionar problemas con tipo y prioridad
- d. **Mecánicos**: Gestionar especialistas y sus plazas

4. Volver al menú principal (opción 0)

5. Seleccionar opción 3: "Simulación con Datos Actuales"

- a. El sistema usará TUS mecánicos creados
- b. Procesará TUS vehículos con sus incidencias reales
- c. Simulará el taller con TU configuración personalizada

Automatización de las pruebas

Tras probar manualmente el sistema 2–3 veces, vi que **dedicaba más tiempo a preparar datos que a analizar resultados**. Para solucionarlo, implementé `simulacion.go` y la función `CrearConfiguracionAutomatica()`, que generan toda la configuración a partir de un número de escenario. Así **pasé de tardar más de 30 minutos en ejecutar los 5 tests a menos de 3 minutos**.

OPCIÓN 2: Simulación Automática (40 segundos / escenario)

1. Ejecutar el programa:

```
go run main.go
```

2. Seleccionar opción 2: "Ejecutar Simulación Automática"

- a. El sistema ejecutará automáticamente los 5 escenarios predefinidos
- b. No requiere ninguna entrada manual
- c. Genera métricas completas de rendimiento

OPCIÓN 3: Tests Individuales (40 segundos / escenario)

Método A: Desde VS Code

1. Abrir el archivo `taller_test.go`
2. **Buscar las funciones de test** (cada escenario tiene su propia función)
3. **Hacer clic en el ícono** que aparece al lado de cada función / test

Método B: Desde Terminal

```
# Ejecutar TODOS los tests
go test -v

# Ejecutar UN test específico
go test -v -run TestEscenario1_ConfiguracionBase
```

2. Análisis Comparativo de Resultados de los Tests

Punto 1 - "Comparativas para cuando la cantidad máxima de coches con una sola incidencia del mismo tipo se duplica."

→ Esto requiere 2 escenarios: **Base** y **Doble Carga**

Punto 2 - "Comparativas para cuando duplicamos la plantilla de 3 mecánicos (uno de cada especialidad) a 6."

→ Esto requiere 2 escenarios: **Base** y **Doble Plantilla**

Punto 3 - "Comparativas para cuando se tiene 3 mecánicos de especialidad mecánica por cada uno de eléctrica y uno de carrocería, y para cuando se tienen 1 mecánico de especialidad mecánica por cada tres de eléctrica y tres de carrocería."

→ Esto requiere 2 escenarios: **3M-1E-1C** y **1M-3E-3C**

Resultados de los Tests

Escenario	Configuración	Duración	Mecánicos Extra	Prioritarios
Caso Base	3 mec (1M/1E/1C)	36.44s	2	4
Doble Carga	3 mec (1M/1E/1C)	39.67s	8	15
Doble Plantilla	6 mec (2M/2E/2C)	20.92s	0	0
Distribución 3M-1E-1C	5 mecánicos	36.49s	0	1
Distribución 1M-3E-3C	7 mecánicos	32.43s	0	0

Caso Base vs Doble Carga

- **Demand:** +100% vehículos (8 → 16)
- **Rendimiento:** Solo +9% de tiempo adicional (39.67s vs 36.44s)
- **Adaptabilidad:** 8 contrataciones automáticas bajo demanda
- **Conclusión:** El sistema maneja carga duplicada con impacto temporal mínimo, demostrando excelente escalabilidad

Caso Base vs Doble Plantilla

- ⌚ **Velocidad:** +70% más rápido (20.92s vs 36.44s)
- ⚠️ **Estabilidad:** Eliminación total de congestiones (4 → 0 prioridades)
- 💰 **Eficiencia:** Cero contrataciones extra necesarias
- 📈 **Conclusión:** Duplicar la plantilla reduce tiempo en 43% y elimina completamente los cuellos de botella

🏆 Ranking de Escenarios

Posición	Escenario	Justificación
1	Doble Plantilla	Más rápido (20.92s) + cero prioridades + cero contrataciones extra
2	Distribución 1M-3E-3C	Balance perfecto: rápido (32.43s) + cero congestiones + especialización eficiente
3	Distribución 3M-1E-1C	Similar tiempo al caso base pero mejor gestión (solo 1 prioridad vs 4)
4	Caso Base	Configuración mínima funcional, pero con 4 prioridades y 2 contrataciones
5	Doble Carga	Estrés máximo: más tiempo, 15 prioridades y 8 contrataciones necesarias

3. Código Fuente

Enlace al Repositorio de GitHub:



[Repositorio – Práctica2 - Sistemas Distribuidos](#)