Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Diseño e Implementación de Tolerancia a Fallos con Baja Latencia en Simulación Distribuida

Autor:

Javier Vela Tambo

Directores:

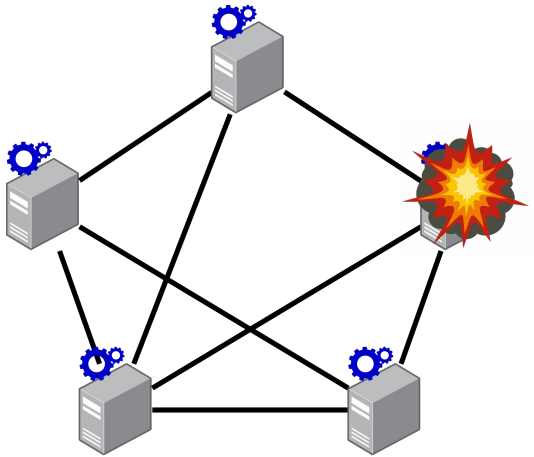
Unai Arronategui Arribalzaga

José Ángel Bañares Bañares

Tolerancia a Fallos en Simulación Distribuida

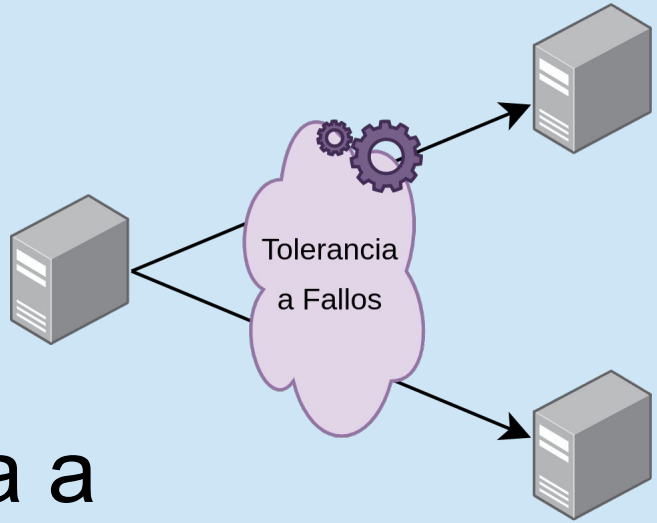
Contexto

- Simulación: herramienta para analizar sistemas de **grandes dimensiones**.
- Limitaciones en la simulación centralizada:
  - **Memoria**
  - Procesamiento
- La **simulación distribuida** es la solución.
- **Escala** en distribuido implica **fallos**.
- Necesidad de incluir mecanismos de tolerancia a fallos.



Objetivo:

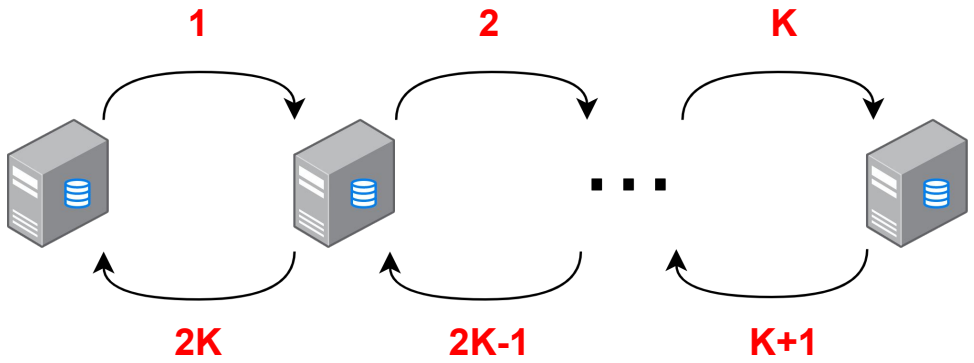
Reducir la latencia introducida por los mecanismos de tolerancia a fallos para **preservar la consistencia** en situaciones **sin fallos**.



Baja Latencia en Comunicación de Eventos

Motivación

- Un aumento en la latencia de comunicación de eventos **ralentiza la simulación**.
- Modelo **TF tradicional** no es la solución.



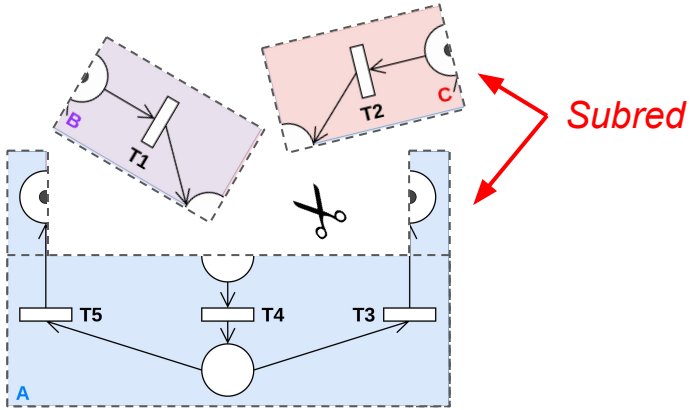
Conceptos de Referencia

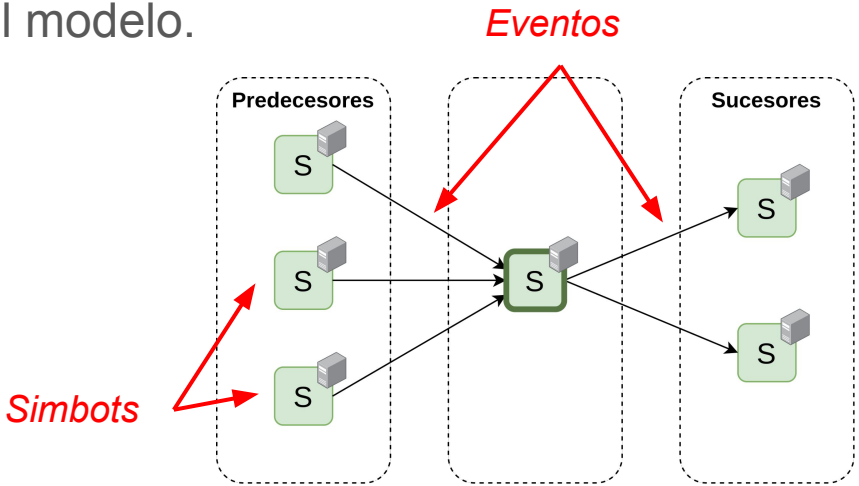
1. Introducción
2. **Conceptos de Referencia**
3. Análisis y Diseño de Tolerancia a Fallos
4. Diseño e Implementación del Simbot
5. Resultados
6. Conclusiones

Nuestro Simulador: *Simbot Swarm*

Conceptos de Referencia

- Simulador distribuido de **sistemas de eventos discretos** (DES).
- Modelado mediante **Redes de Petri** con tiempo.
- **Partición** genera dependencias del modelo.

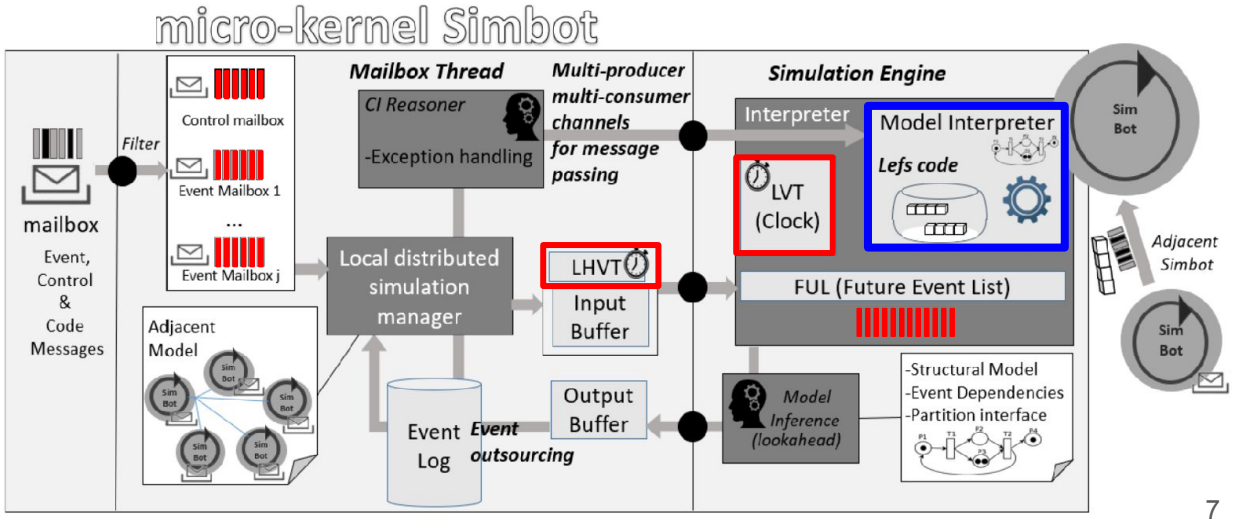




Arquitectura del *Simbot*

Conceptos de Referencia

- Los *simbots* utilizan diversas **colas de eventos**.
- Muy importante **mantener la causalidad** de eventos. Mediante estrategias:
  - **Conservativa**
  - Optimista
- Estado del *Simbot*
  - Modelo y Sensibilización
  - Colas de eventos
  - Relojes locales de Tiempo de Simulación



Análisis y Diseño de Tolerancia a Fallos

1. Introducción
2. Conceptos de Referencia
3. **Análisis y Diseño de Tolerancia a Fallos**
4. Diseño e Implementación del Simbot
5. Resultados
6. Conclusiones

Análisis de Tolerancia a Fallos

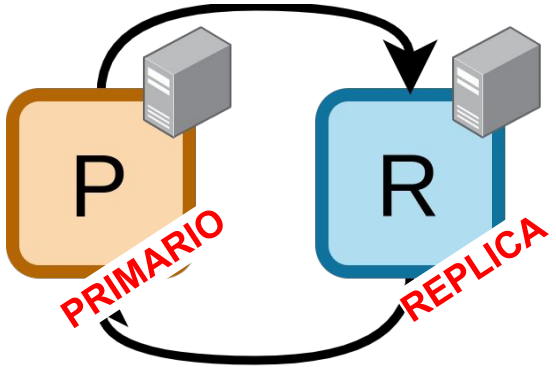
Análisis

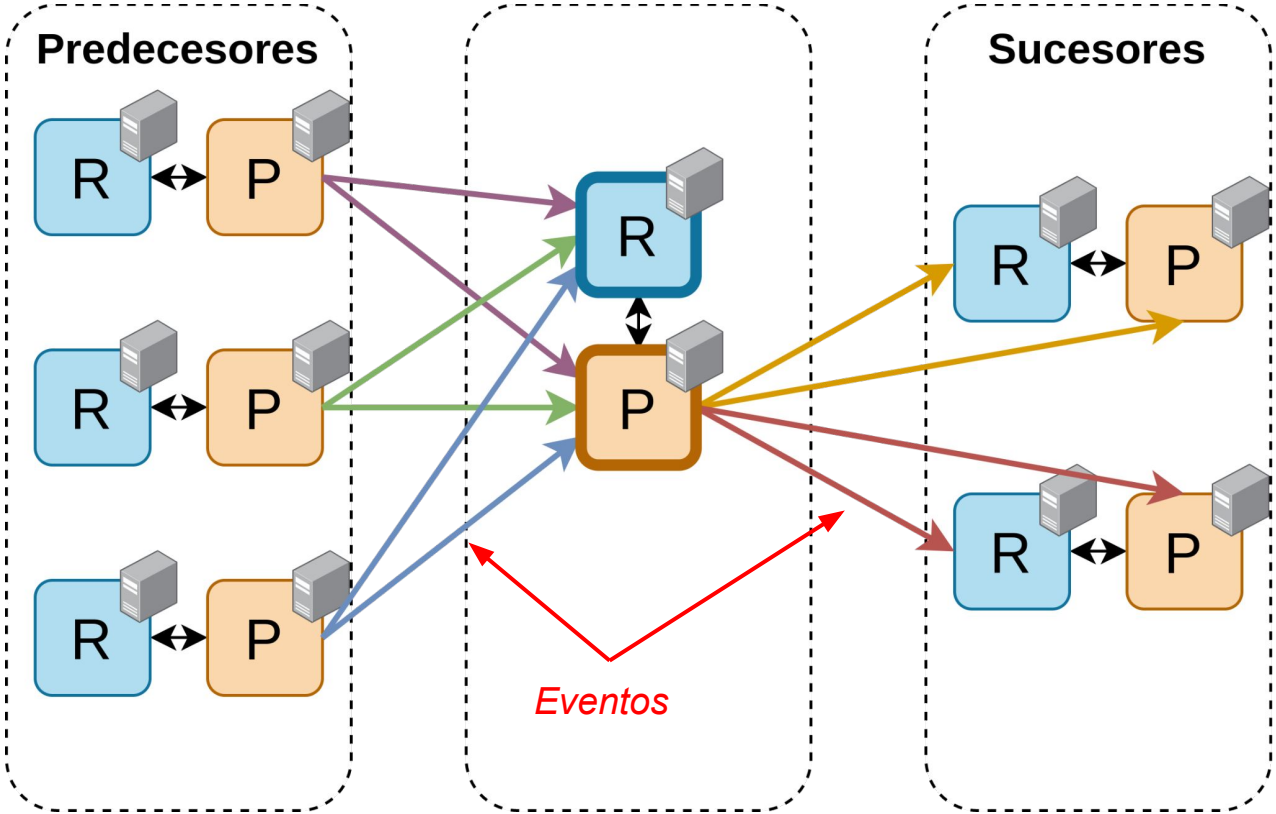
- Tolerar **1 fallo por subred** (partición del modelo).
- Mantener **consistencia de causalidad**: no perder **eventos**, ni su **orden**.
- Cobertura de todas las **situaciones complejas** en las que ocurre un fallo.
- **Baja latencia** en situaciones **sin fallos**:
  - Reducir **mensajes** necesarios.
  - Minimizar **coordinación** y **sincronización**.

Replicación Adaptada

Diseño

- Modelo de **máquina de estados replicada** adaptada a la simulación.
  - **Factor replicación k = 2**
- *Simbot* **primario modifica el estado** de la simulación.
- El estado de réplicas evoluciona por igual.
- **Uso de mensajes y tiempos de simulación.** KEY IDEA



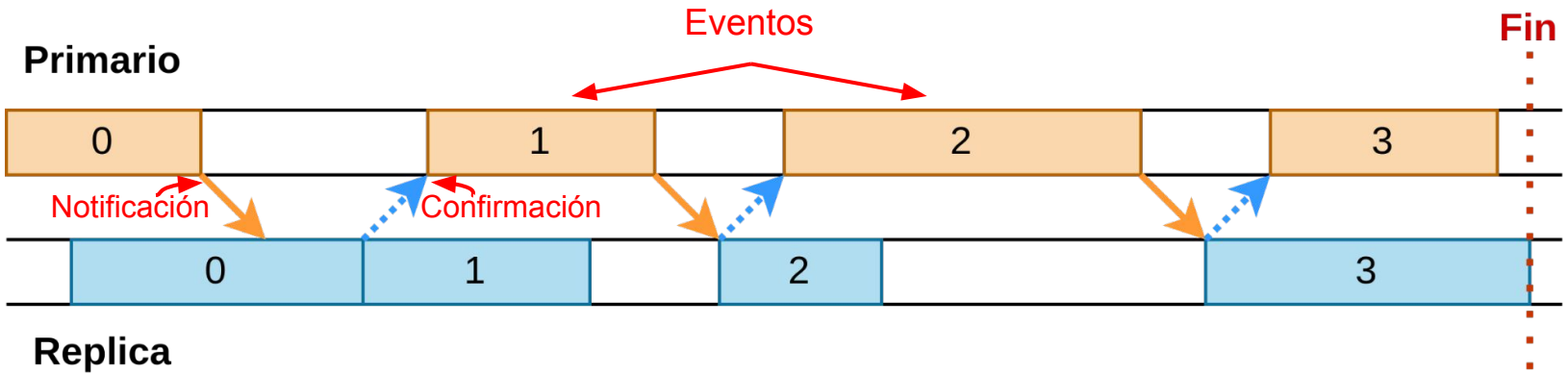


Estructura de ***Simbots*** Vecinos **Replicados**

Consistencia de Réplicas

Análisis

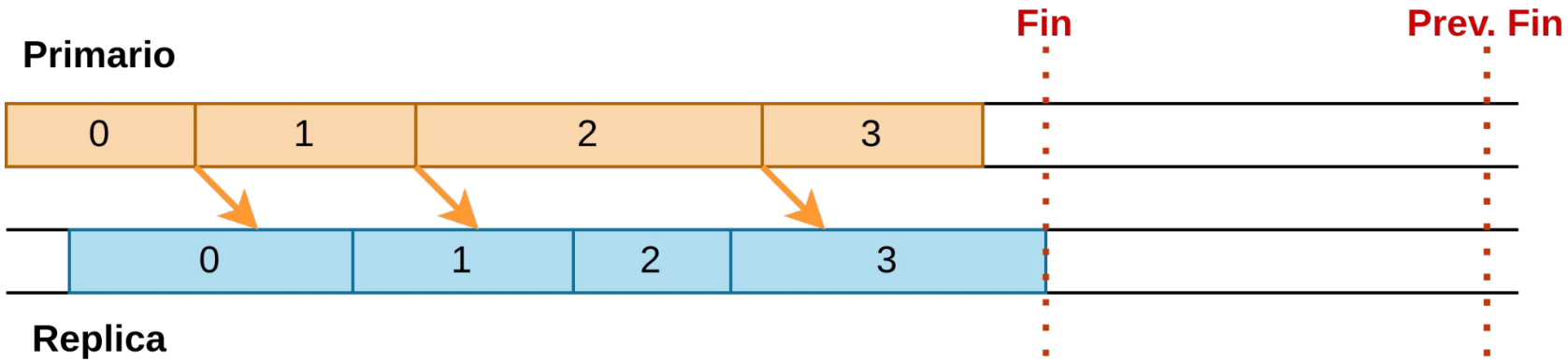
- **Consistencia** del estado y comportamiento **de réplicas**.
- La consistencia **estricta** conlleva aumento de mensajes y sincronización.



Desacoplamiento de Réplicas

Análisis / Diseño

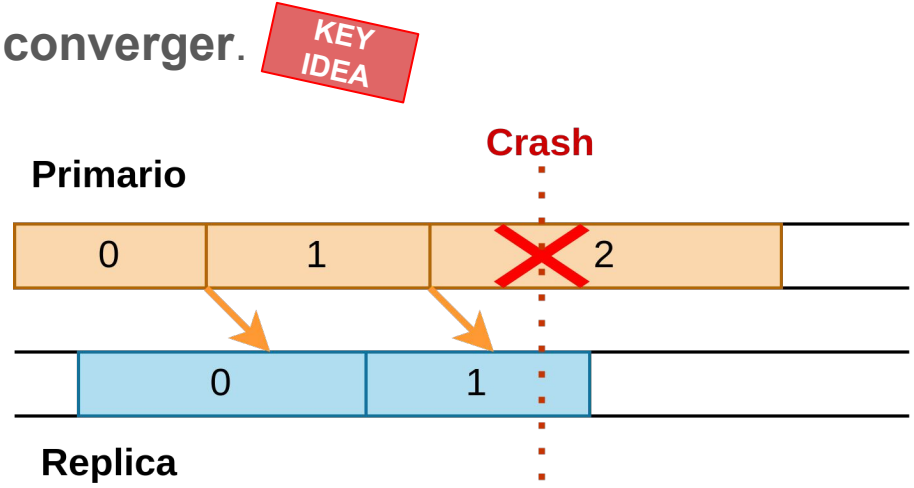
- **Desacoplamiento de la ejecución** de las réplicas. KEY IDEA
- **Consistencia causal**:
  - Coordinación débil: Notificaciones en un único sentido.
  - Reduce tiempos de espera.

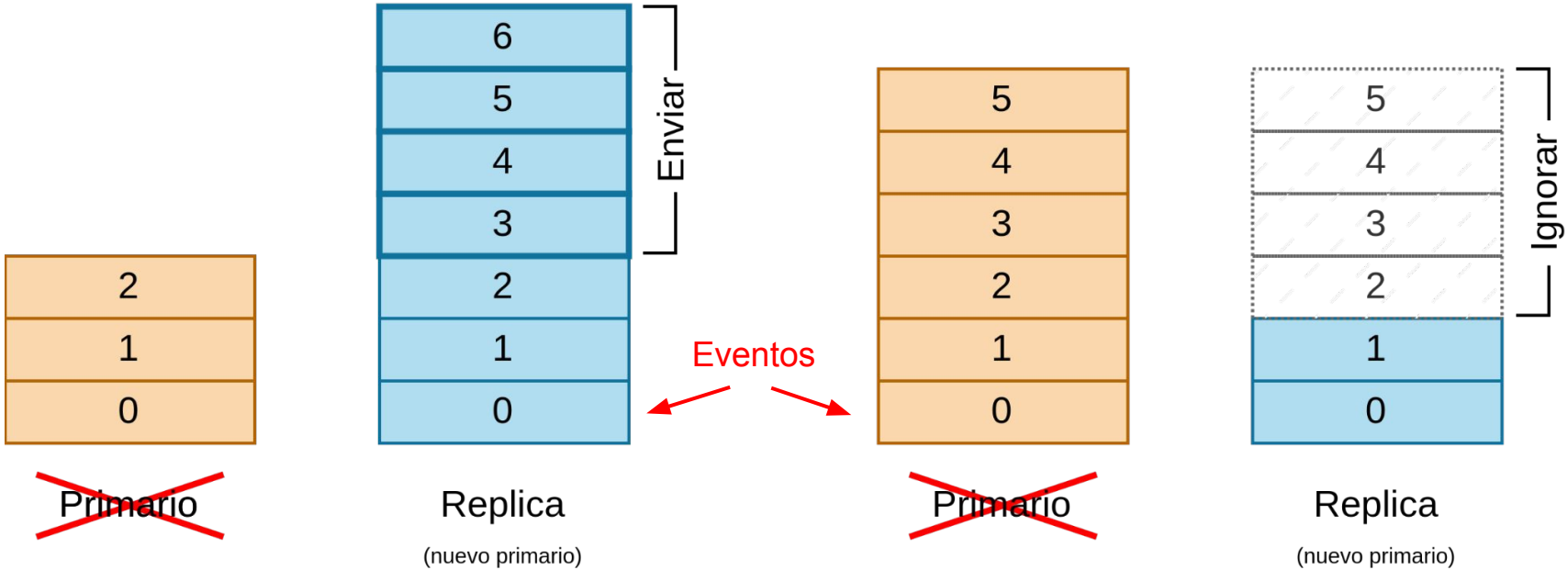


Recuperar la Consistencia ante un Fallo

Análisis

- Fallo de réplica:
  - Primario continúa
- Tras un fallo, la **consistencia** debe **converger**.
- Fallo de primario:
  - Réplica más avanzada
  - Réplica más retrasada
  - Réplica y primario igual



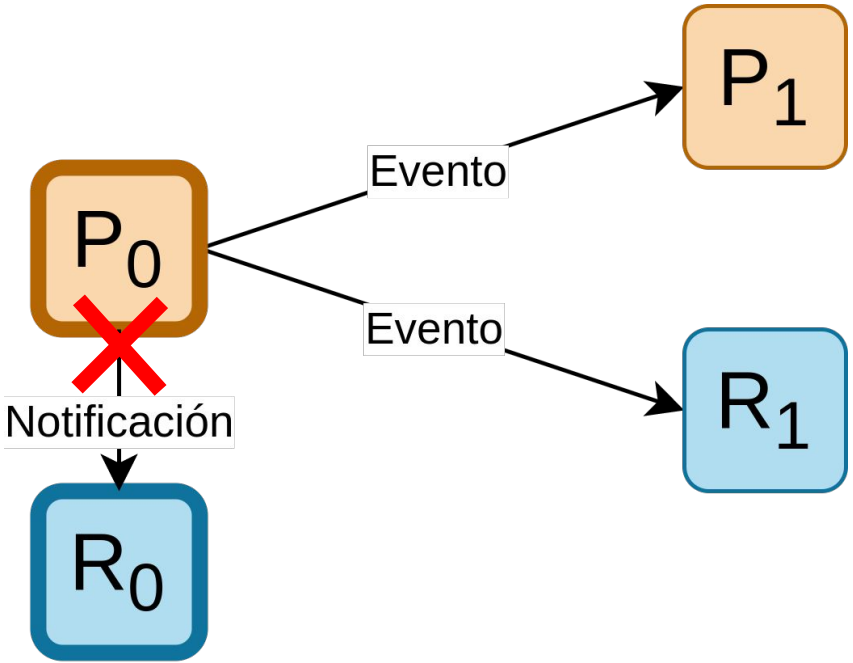


Convergencia de la Consistencia en **Promoción de Réplica**

Situaciones Complejas de Fallo

Análisis / Diseño

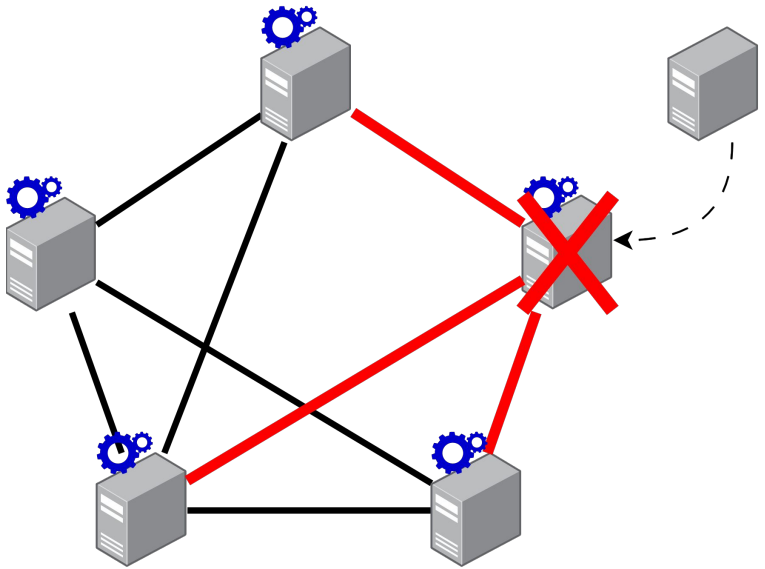
- Fallo entre envío de notificaciones puede llevar a **eventos repetidos**.
- Es necesario un mecanismo para **invalidar mensajes**.
  - Mediante un **identificador único**.

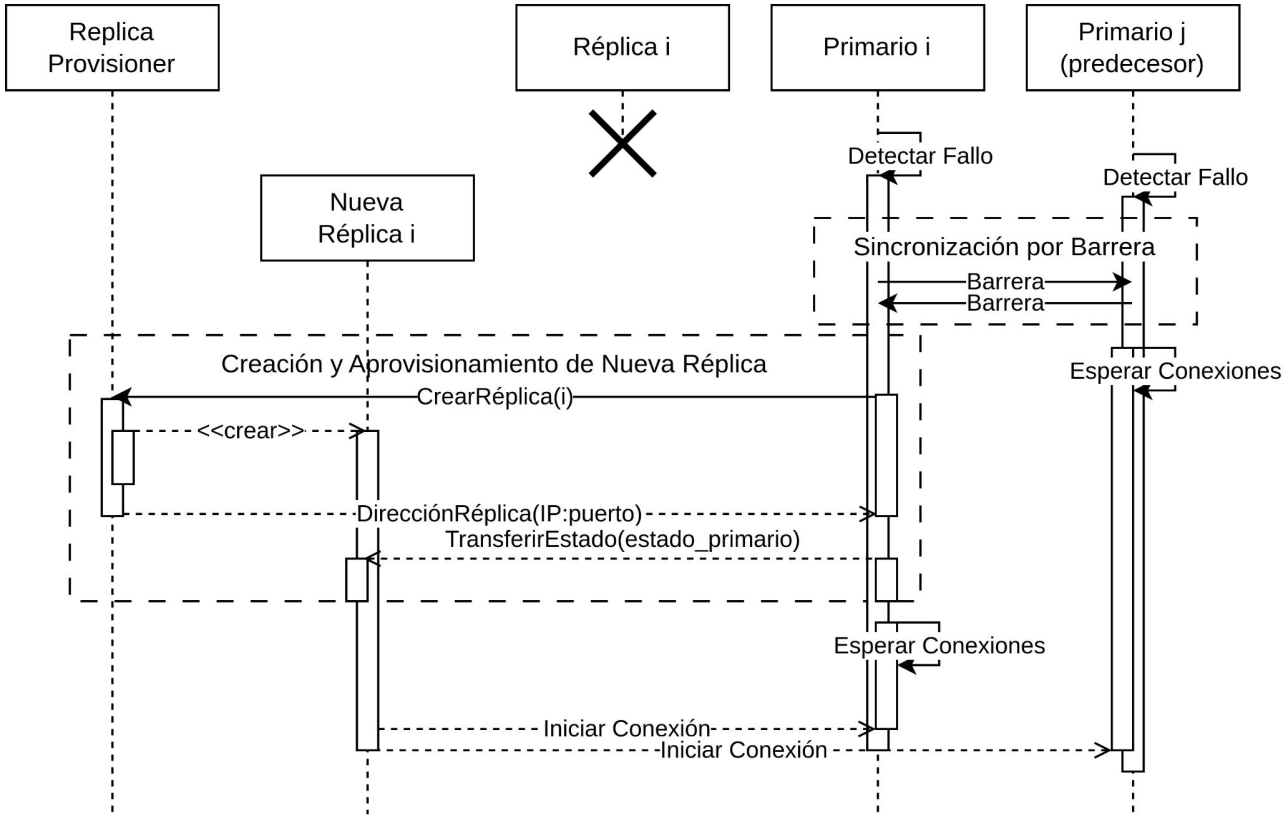


Detección y Recuperación de Fallos

Diseño

- **Detección de fallos** de *simbots* de subredes vecinas y propia.
- Mecanismos de **recuperación**:
  a. Fallo de **Réplica** → Petición nueva réplica
  b. Fallo de **Primario**→ Promoción
  c. Fallo de **Sucesor** → Sincronización
- **Incorporación** dinámica de *simbots*.





Petición de **Nueva Réplica** tras Fallo y **Copia de Estado**

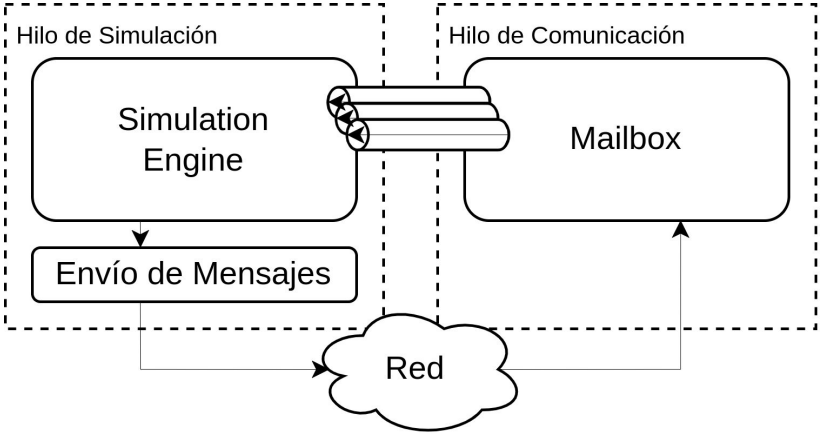
Diseño e Implementación del Simbot

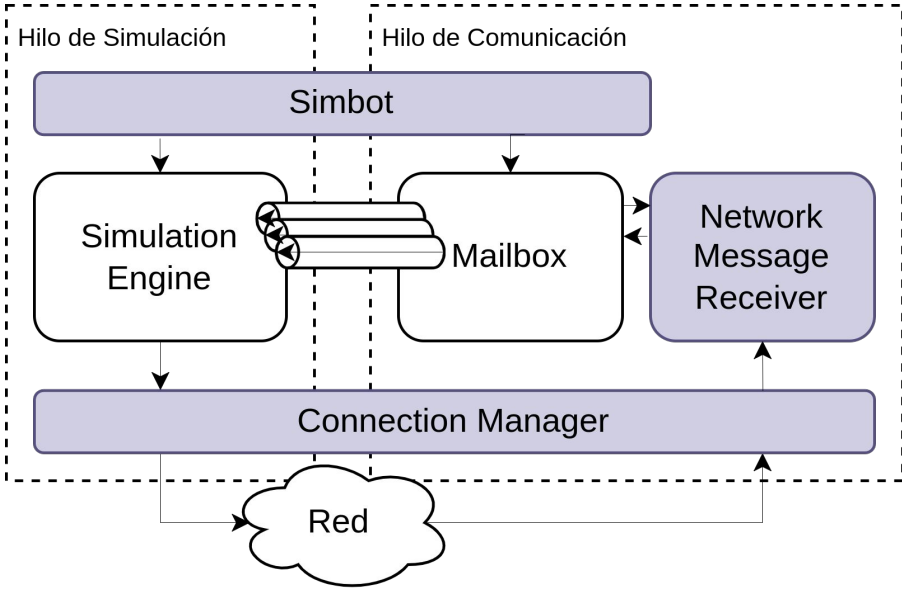
1. Introducción
2. Conceptos de Referencia
3. Análisis y Diseño de Tolerancia a Fallos
4. **Diseño e Implementación del *Simbot***
5. Resultados
6. Conclusiones

Simbot

Implementación

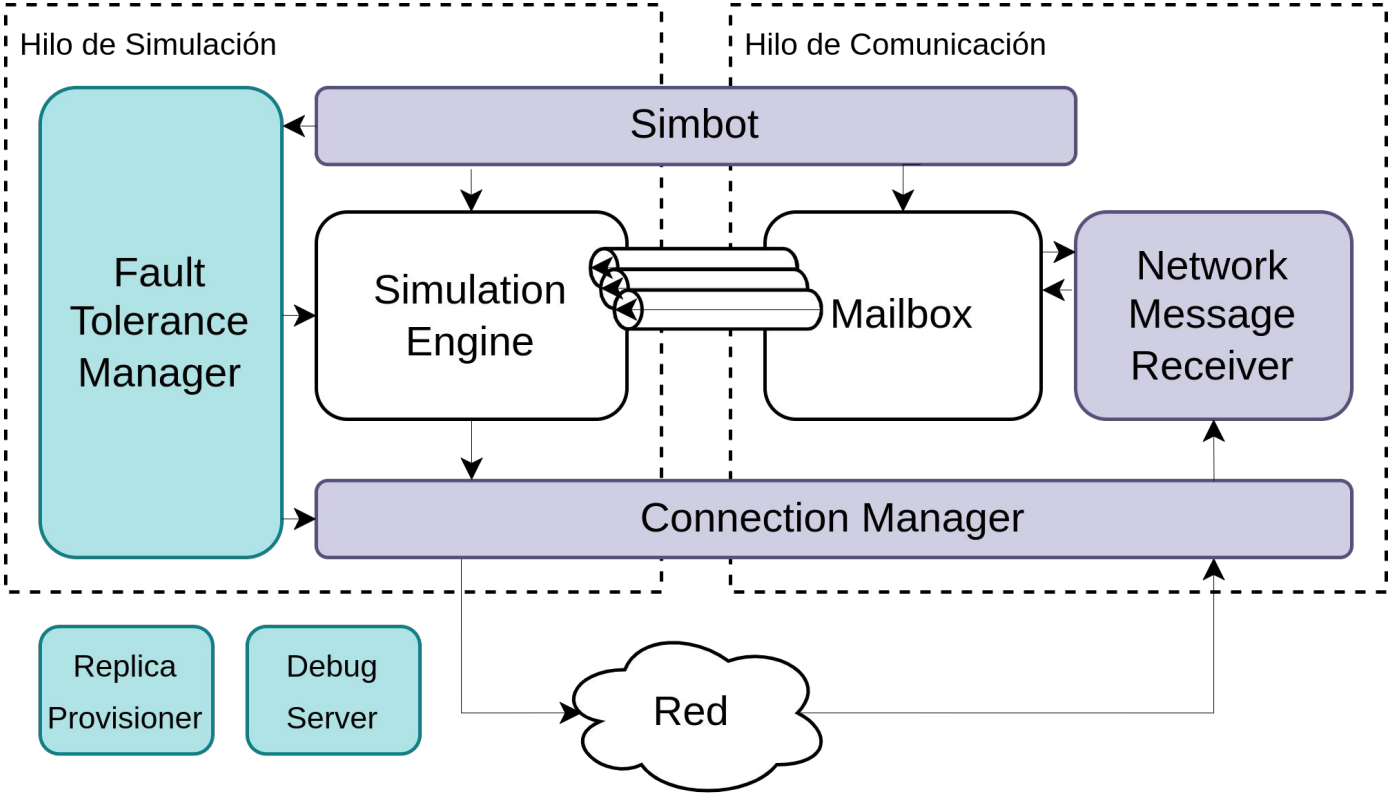
- **Rust**: lenguaje **eficiente y seguro** en memoria y concurrencia por compilación.
- Necesidad de adaptar el diseño **previo**.
- Dos versiones: **sin** y **con tolerancia a fallos**.
- Dos servicios auxiliares:
  - ***Debug Server***
  - ***Replica Provisioner***



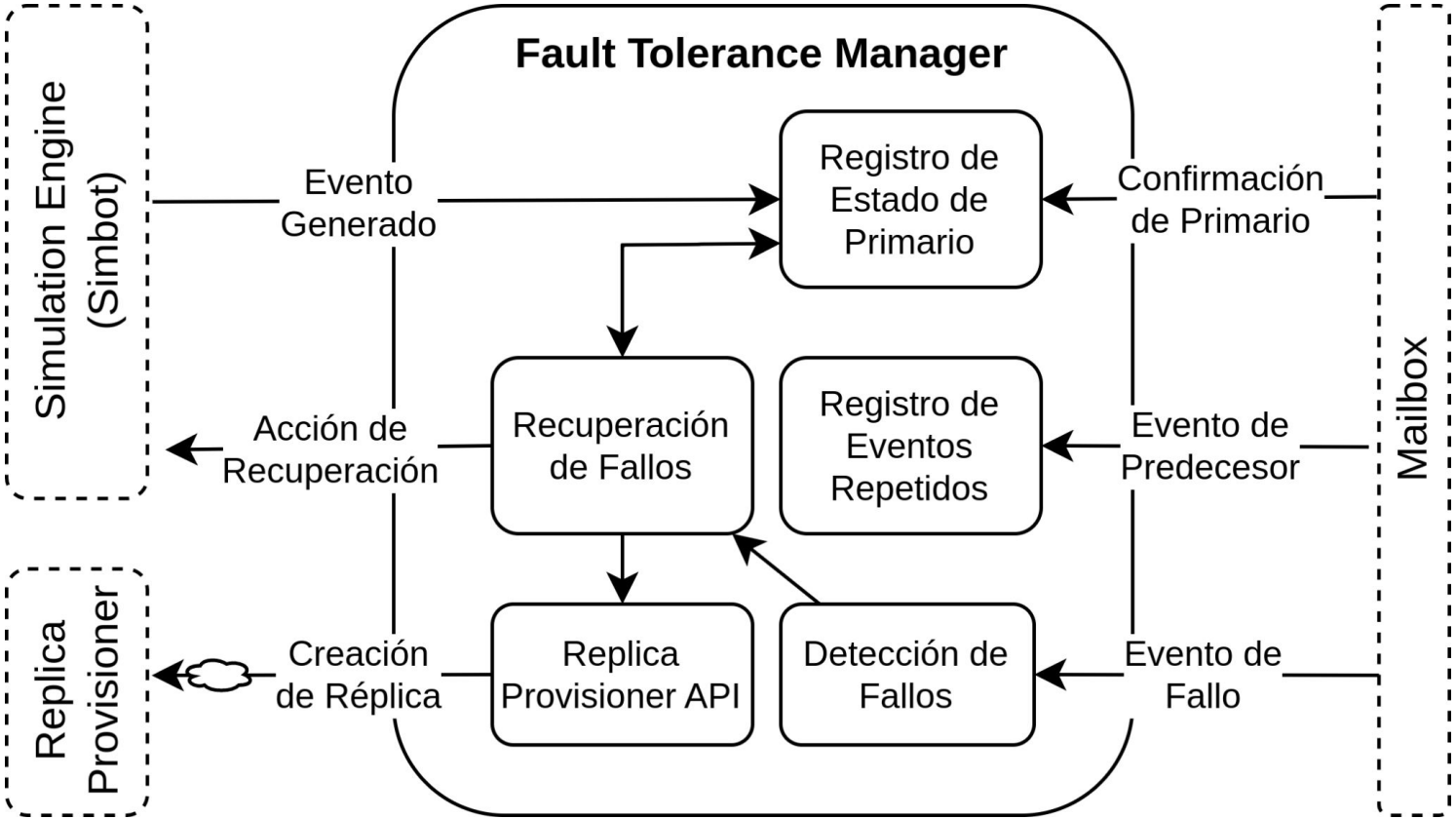


Arquitectura *Simbot* (previo)

Arquitectura *Simbot* (sin tolerancia a fallos)



Arquitectura *Simbot* (con tolerancia a fallos)



Componentes del ***Fault Tolerance Manager***

Resultados

1. Introducción
2. Conceptos de Referencia
3. Análisis y Diseño de Tolerancia a Fallos
4. Diseño e Implementación del Simbot
5. **Resultados**
6. Conclusiones

Mejora frente a Implementación Previa

Resultados

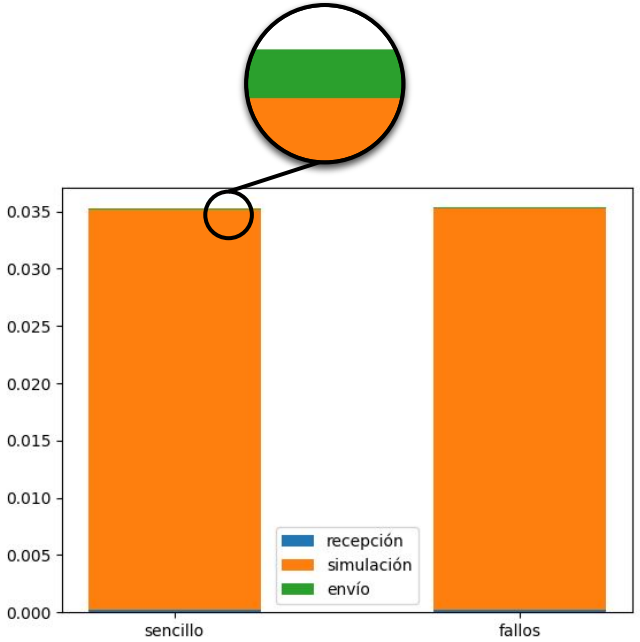
- Nueva versión del simulador sin tolerancia a fallos **21** veces más rápida.
  - En simulación de **100.000 ciclos**
- Optimización de:
  - Conexiones TCP
  - Estructuras de datos
  - Uso de memoria

Ciclo Final	Tiempo de Ejecución	
	previa	nueva
20	0.0131	0.0067
1000	0.461	0.162
10000	20.437	1.643
100000	372.88	17.655

Latencia de Simulación con Tolerancia a Fallos

Resultados

- Ejecución **con carga de simulación**:
  - Carga **mínima rentable** para la simulación distribuida frente a centralizada (0.035 segundos).
  - **Aumento** de tiempo de ejecución **despreciable** (0,269 %).
- Ejecución **con carga de comunicación**:
  - El aumento del tiempo de envío **no se transfiere** al resto de la simulación.



Conclusiones

1. Introducción
2. Conceptos de Referencia
3. Análisis y Diseño de Tolerancia a Fallos
4. Diseño e Implementación del Simbot
5. Resultados
6. **Conclusiones**

Objetivos Cumplidos

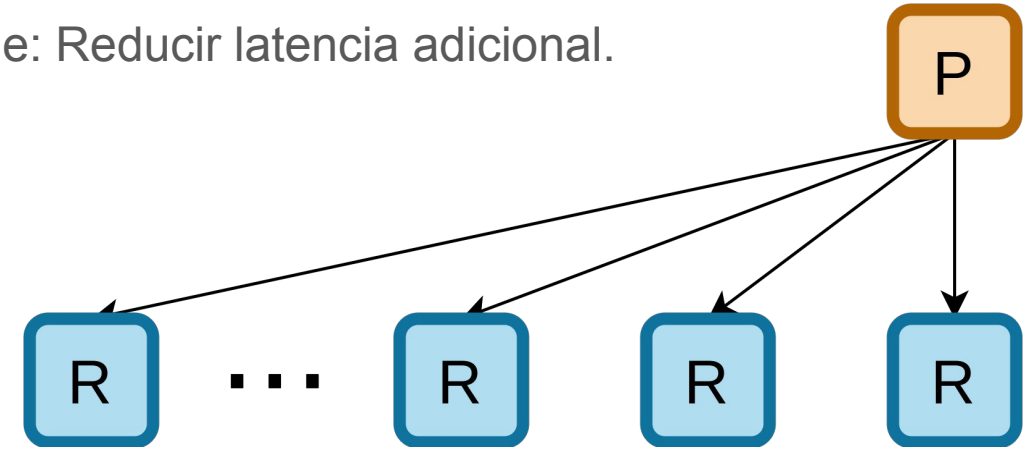
Conclusiones

- Análisis de tolerancia a fallos en **simulación distribuida** escalable
- Diseño de **mecanismos** de tolerancia a fallos basados en **replicación adaptada**
- Simulación en situaciones sin fallos **sin aumento significativo de latencia**

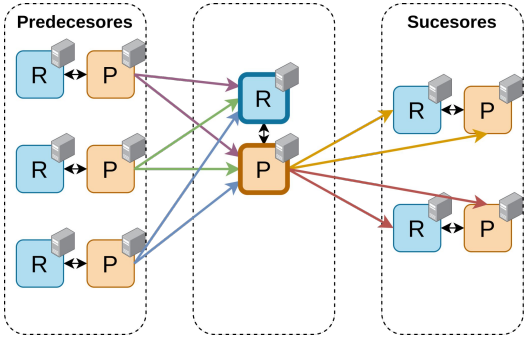
Trabajo Futuro

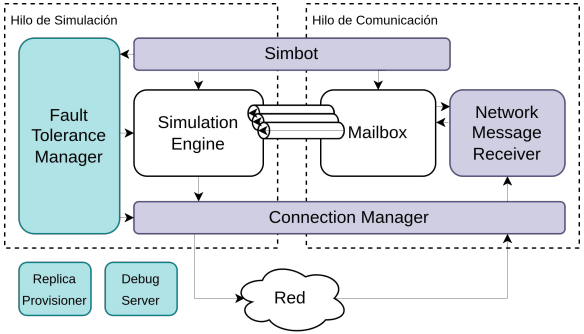
Conclusiones

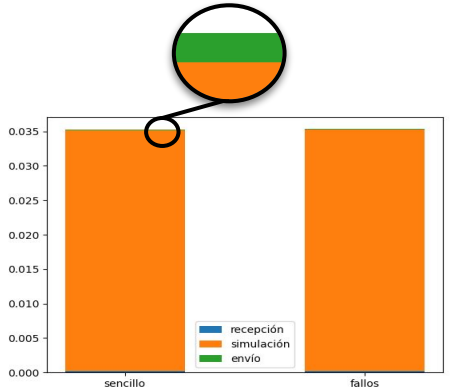
- **Aumentar la tolerancia a fallos del simulador**.
- **Múltiples réplicas**, extender comportamiento actual.
- Mediante **IP *Multicast*** fiable: Reducir latencia adicional.



Diseño e Implementación de Tolerancia a Fallos con Baja Latencia en Simulación Distribuida







Javier Vela Tambo (775593@unizar.es)

Anexos

Algorithm 1 Procedimientos de Registro de Estado del Primario en Réplica

// Diferencia de estado entre Primario y Réplica. Inicialización: Lista vacía

$Q^R \leftarrow []$

// Último mensaje generado por Réplica. Inicialización: ID Mensaje nulo

$M^R \leftarrow 0$

// Último mensaje confirmado por Primario. Inicialización: ID Mensaje nulo

$M^P \leftarrow 0$

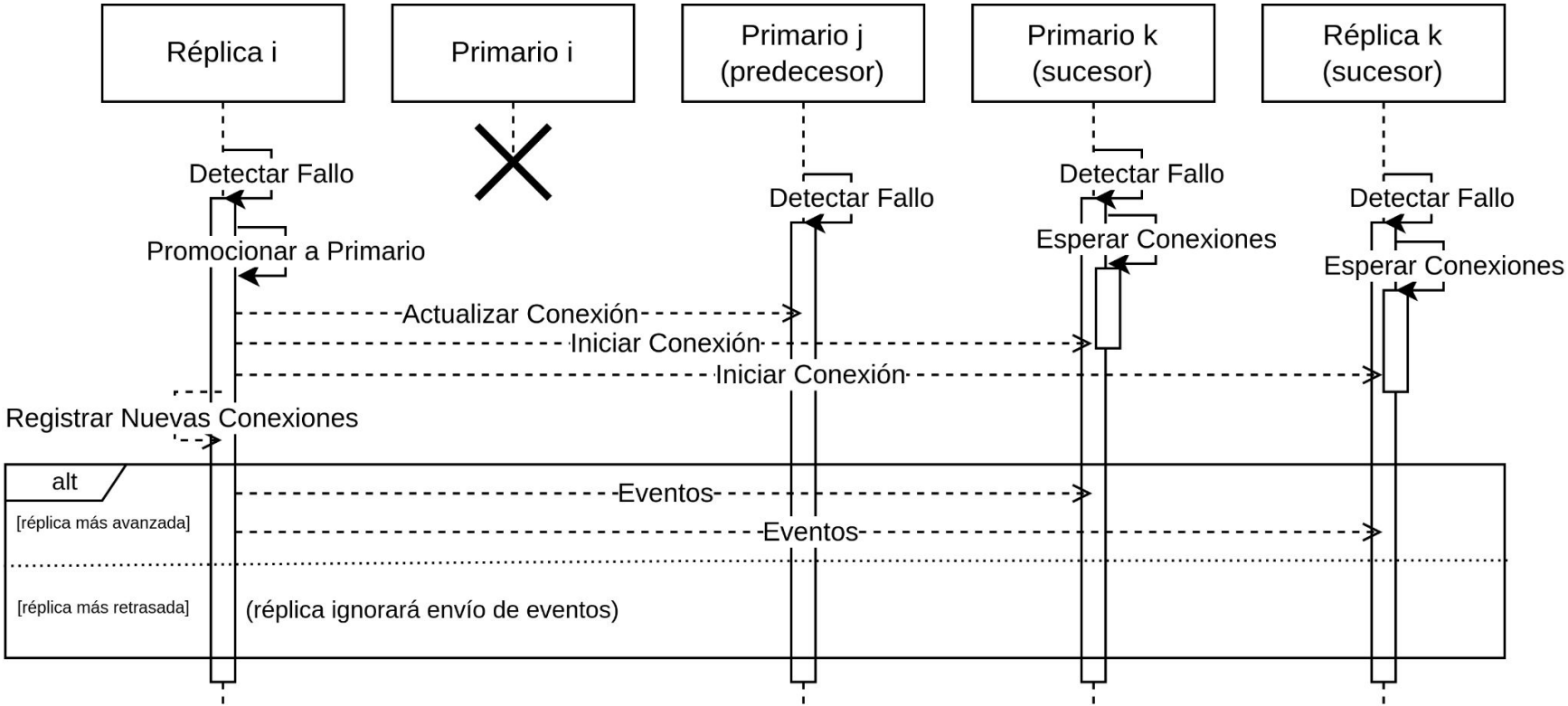
Algoritmo de Registro del Estado

```
// Recepción de Confirmación de Evento del Primario
procedure registrarConfirmaciónRecibida(M : Confirmación)
    M^P ← M_ID
    if M^R >= M^P then
        // Réplica va más avanzada o a la par con el Primario
        Q^R.eliminarPrimero()
    else
        // (M^R < M^P) Réplica va más retrasada
        no − op
    end if
end procedure
```

Algoritmo de Registro del Estado (Continuación)

```
// Generación de Evento de la Réplica
procedure registrarEventoGenerado(M : Mensaje)
    M^R ← M_ID
    if M^R > M^P then
        // Réplica va más avanzada
        Q^R.insertarUltimo(M)
    else
        // (M^R <= M^P) Réplica va más retrasada o a la par con el Primario
        no − op
    end if
end procedure
```

Algoritmo de Registro del Estado (Continuación)



Promoción de Réplica Tras Fallo de Primario

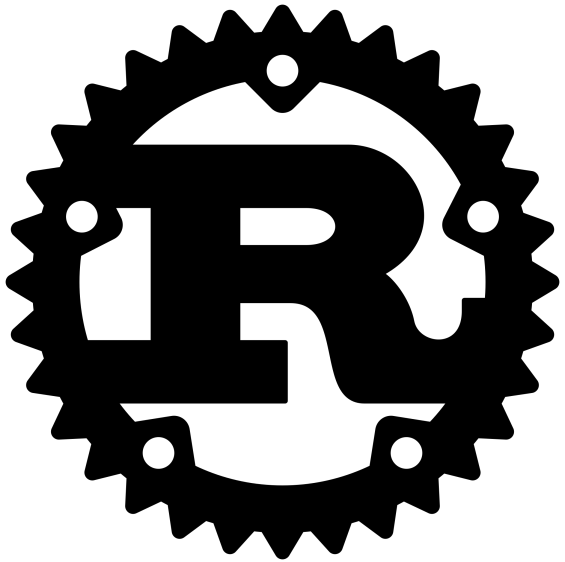
Ideas Clave del Diseño

Diseño

Rust

Implementación

- Seguro en memoria y concurrencia.
- Librerías:
  - Canales de mensajes: *crossbeam_channel*
  - Memoria compartida: *Arc* y *RwLock*
  - Conexiones no-bloqueantes y *Poll: mio*
  - Codificación binaria: *bincode*



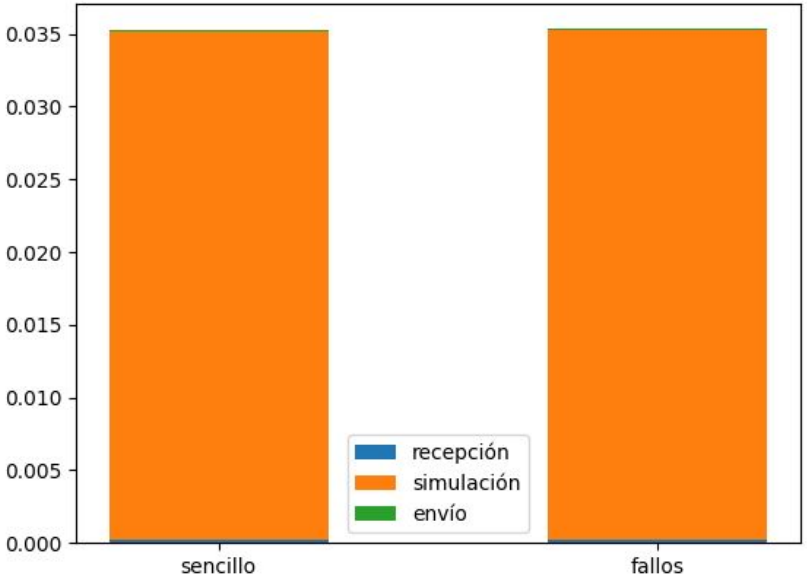
- El simulador es capaz de procesar $P = 7522936$ [4] eventos por segundo.
- La latencia de red del entorno de simulación es $\tau = 350$ microsegundos.
- La densidad de eventos del modelo es $E = 1$ evento por *simsecond*.

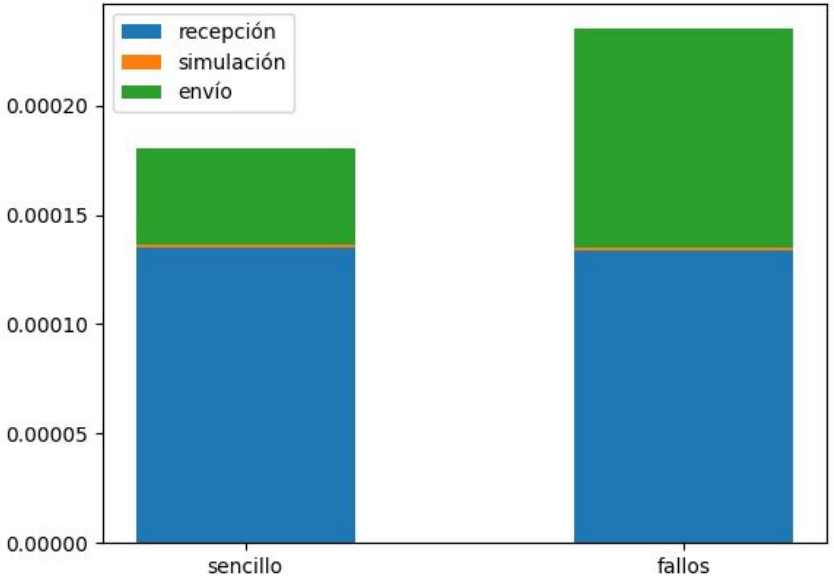
$$\lambda = \frac{L \times E}{\tau \times P}$$

$$100 >= \frac{L \times E}{350 \times 10^{-6} \times 7{,}5 \times 10^{6}}$$

$$L \times E >= 100 \times 350 \times 10^{-6} \times 7{,}5 \times 10^{6} = 262500$$

Cálculo de Carga de Simulación Mínima Rentable





Resultados de Experimentos

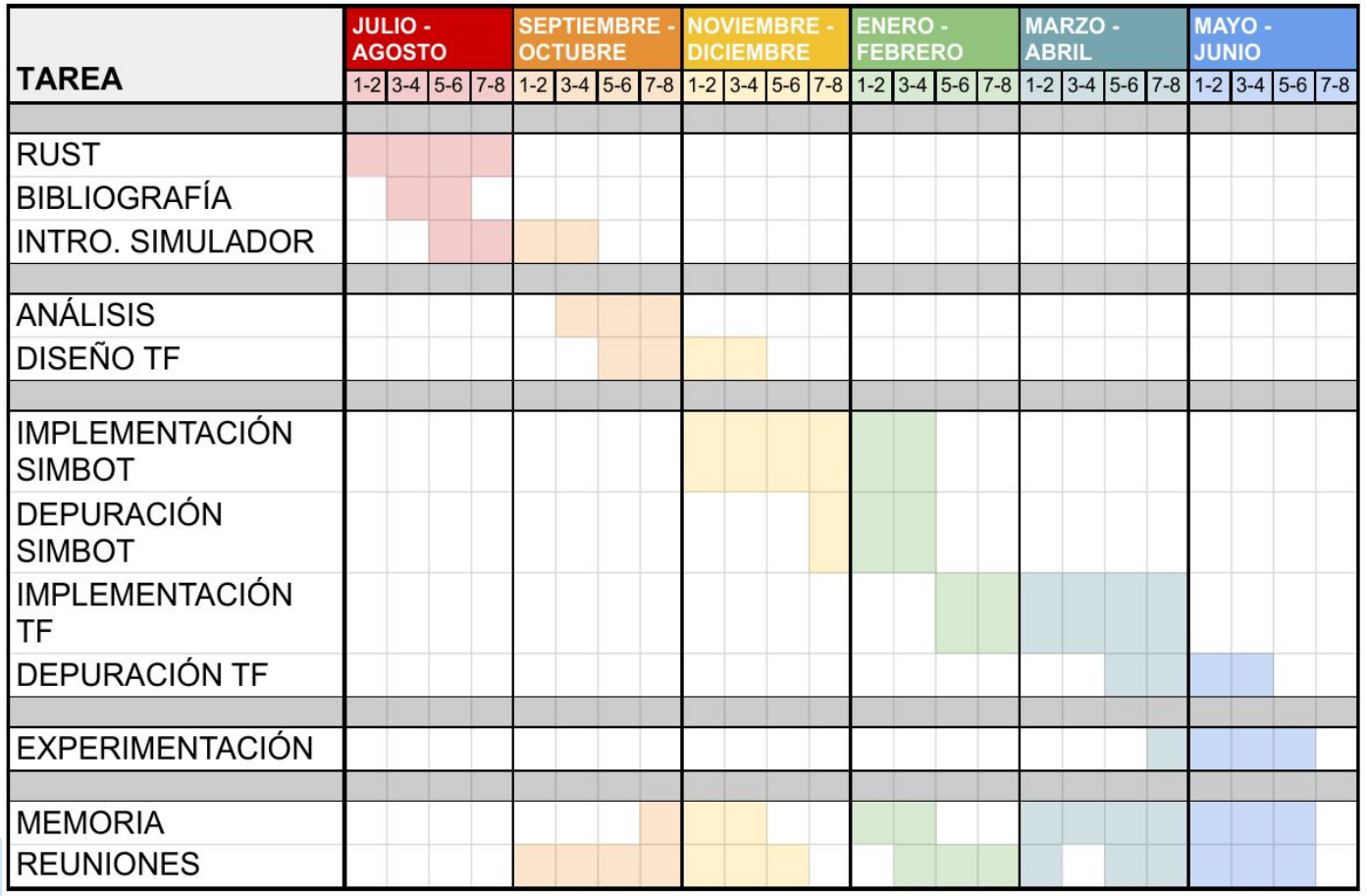


Diagrama de Gantt del Proyecto

Tarea Desarrollada	**Tiempo** (horas)
Revisión de la bibliografía	10
Introducción al lenguaje Rust	15
Introducción al código del simulador	20
Análisis de fallos y diseño de la solución	50
Diseño, implementación y depuración del *simbot*	110
Implementación de tolerancia a fallos en el *simbot*	50
Experimentación	75
Reuniones	45
Redacción de la memoria	60
Horas totales	**435 horas**

Tabla de Control de Horas del Proyecto