Sistema Distribuido para Simulación Montecarlo con RabbitMQ

Aguilar Serrano Diego Fernando[1], Guevara Cano Cristhian Daniel[1]

Benemérita Universidad Autónoma de Puebla, Puebla, México
{diego.aguilarse,cristhian.guevaraca}@alumno.buap.mx

Abstract. Este trabajo presenta el diseño e implementación de un sistema distribuido para realizar simulaciones Montecarlo utilizando el paradigma de paso de mensajes. El sistema emplea RabbitMQ como middleware de mensajería, permitiendo la coordinación entre un productor que genera escenarios únicos, múltiples consumidores (workers) que procesan dichos escenarios de forma distribuida, y un visualizador que presenta estadísticas en tiempo real. La arquitectura implementada garantiza escalabilidad, tolerancia a fallos y la capacidad de cambio dinámico de modelos sin interrumpir la operación del sistema. Se muestra que el uso de un middleware de mensajería como RabbitMQ es fundamental para lograr una coordinación efectiva en sistemas distribuidos de simulación, simplificando la tarea de comunicación entre procesos y servicios.

Keywords: sistemas distribuidos · simulación Montecarlo · paso de mensajes · RabbitMQ · middleware · AMQP

1 Introducción

Las simulaciones Montecarlo son técnicas computacionales que utilizan muestreo aleatorio repetitivo para obtener resultados numéricos [1]. Su aplicación abarca desde finanzas y física hasta ingeniería y ciencias sociales. Sin embargo, estas simulaciones suelen requerir un gran número de iteraciones para alcanzar resultados estadísticamente significativos, lo que demanda considerable capacidad computacional.

Los sistemas distribuidos ofrecen una solución natural a este problema al permitir el procesamiento paralelo de escenarios en múltiples nodos computacionales [3]. El paradigma de paso de mensajes, específicamente, proporciona un modelo de programación donde los procesos se comunican mediante el envío explícito de mensajes, evitando el uso de memoria compartida y sus problemas asociados [4].

La necesidad de realizar simulaciones Montecarlo de manera eficiente y escalable motiva el desarrollo de sistemas distribuidos especializados. Los desafíos principales incluyen:

– **Coordinación**: Garantizar que múltiples workers procesen escenarios únicos sin duplicación

- **Escalabilidad**: Permitir agregar o remover workers dinámicamente
- **Flexibilidad**: Cambiar modelos de simulación sin reiniciar el sistema
- **Monitoreo**: Visualizar el progreso y estadísticas en tiempo real

El documento se organiza como sigue: la sección 2 presenta los conceptos fundamentales. La sección 3 describe la arquitectura del sistema. La sección 4 detalla el diseño de componentes. La sección 5 presenta la implementación del sistema. La sección 6 analiza la integración del middleware. Finalmente, la sección 7 presenta conclusiones.

2 Marco Teórico

2.1 Simulación Montecarlo

La simulación Montecarlo es un método numérico que utiliza muestreo aleatorio para resolver problemas que pueden ser determinísticos o estocásticos [2]. El proceso general consiste en:

1. Definir un dominio de entradas posibles
2. Generar muestras aleatorias del dominio según distribuciones de probabilidad
3. Realizar cálculos determinísticos sobre cada muestra
4. Agregar los resultados para obtener estimaciones estadísticas

Matemáticamente, si queremos estimar $E[f(X)]$ donde X es una variable aleatoria, el estimador Montecarlo es:

$$\hat{\mu}_N = \frac{1}{N} \sum_{i=1}^{N} f(X_i) \tag{1}$$

donde $X_1, X_2, ..., X_N$ son muestras independientes de X.

2.2 Sistemas Distribuidos

Un sistema distribuido es una colección de elementos computacionales autónomos que aparecen ante sus usuarios como un sistema único y coherente [3]. Las características principales incluyen:

- **Concurrencia**: Múltiples procesos ejecutándose simultáneamente
- **Ausencia de reloj global**: No existe sincronización de tiempo perfecta
- **Fallos independientes**: Componentes pueden fallar de manera independiente

2.3 Paso de Mensajes

El paradigma de paso de mensajes permite la comunicación entre procesos mediante el envío explícito de mensajes [4]. Las operaciones primitivas son:

- `send(destino, mensaje)`: Enviar mensaje a un destino
- `receive(fuente, mensaje)`: Recibir mensaje de una fuente

Este paradigma evita los problemas de sincronización de memoria compartida y facilita la construcción de sistemas distribuidos escalables.

2.4 Middleware de Mensajería

Un middleware de mensajería actúa como intermediario entre productores y consumidores de mensajes [5], proporcionando servicios como:

- **Desacoplamiento**: Productores y consumidores no necesitan conocerse
- **Persistencia**: Mensajes pueden almacenarse para garantizar entrega
- **Enrutamiento**: Mensajes se dirigen a consumidores apropiados
- **Balanceo de carga**: Distribución automática de trabajo

3 Arquitectura del Sistema

3.1 Componentes

La arquitectura del sistema sigue el patrón productor-consumidor distribuido (Fig. 1). Los componentes principales son:

- **Productor**: Genera escenarios únicos basados en un modelo
- **Workers**: Procesan escenarios y calculan resultados
- **Dashboard**: Coordina el sistema y visualiza resultados
- **Middleware (RabbitMQ)**: Facilita la comunicación mediante colas

3.2 Colas de Mensajes

El sistema utiliza cuatro colas de mensajes especializadas (Tabla 1):

Table 1. Colas de mensajes del sistema y sus características

Cola	Propósito	TTL
`modelo`	Distribuir modelo de simulación a workers	Configurable
`escenarios`	Distribuir escenarios generados	Sin TTL
`resultados`	Recolectar resultados procesados	Sin TTL
`comandos`	Enviar comandos de control	Sin TTL

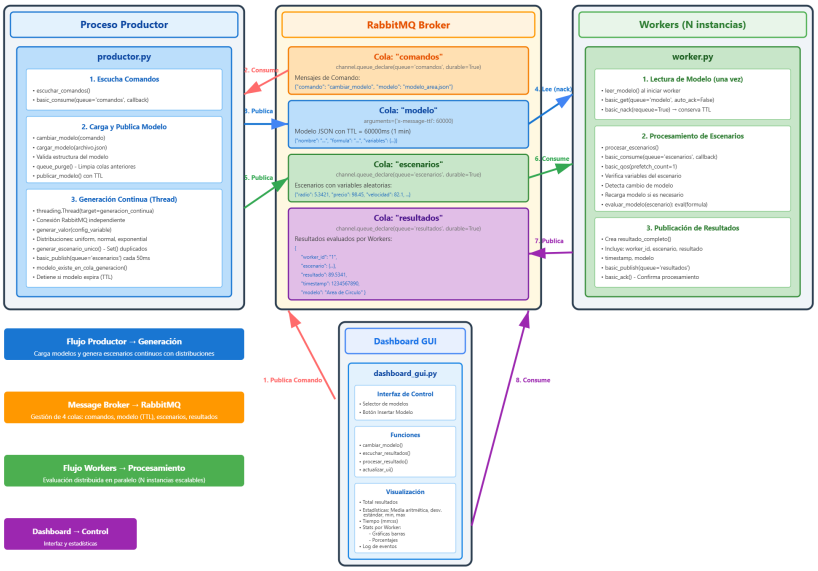

Fig. 1. Arquitectura general del sistema distribuido. Los componentes se comunican exclusivamente a través del middleware RabbitMQ, el cual gestiona cuatro colas especializadas para comandos, modelos, escenarios y resultados.

3.3 Flujo de Operación

El flujo de operación del sistema sigue estos pasos:

1. El Dashboard envía un comando de "cambiar modelo" a la cola de comandos
2. El Productor recibe el comando, carga el modelo y lo publica en la cola de modelo con TTL
3. El Productor inicia la generación continua de escenarios únicos
4. Los Workers leen el modelo (una sola vez) de la cola usando NACK+requeue
5. Los Workers consumen escenarios, evalúan el modelo y publican resultados
6. El Dashboard consume resultados y actualiza estadísticas en tiempo real
7. Al expirar el TTL, el modelo desaparece automáticamente de la cola

3.4 Escalabilidad

La arquitectura permite escalabilidad mediante:

- **Workers dinámicos**: Se pueden agregar o remover workers sin afectar el sistema
- **Colas durables**: Los mensajes persisten aunque no haya consumidores activos
- **QoS configurable**: Cada worker procesa mensajes según su capacidad
- **Distribución automática**: RabbitMQ balancea carga entre workers disponibles mediante el algoritmo Round-Robin [7]

4 Diseño de Componentes

En esta sección se profundiza la estructura de cada uno de los componentes del sistema.

4.1 Productor

El productor es responsable de generar escenarios únicos continuamente hasta que el modelo expire.

Estructura de Datos

- `modelo_actual`: Estructura JSON con fórmula y definición de variables
- `escenarios_generados`: HashSet para garantizar unicidad mediante hashing
- `connection_comandos`: Conexión RabbitMQ para recibir comandos del dashboard
- `connection_generacion`: Conexión RabbitMQ independiente para publicar escenarios

Thread-Safety El productor utiliza dos conexiones independientes a RabbitMQ para evitar conflictos de concurrencia:

- **Thread Principal**: Escucha comandos mediante operación bloqueante
- **Thread Generación**: Publica escenarios continuamente en loop

Esta separación es necesaria porque las bibliotecas cliente de RabbitMQ generalmente no son thread-safe para operaciones concurrentes en el mismo canal.

4.2 Worker

Los workers procesan escenarios de manera completamente independiente, sin coordinación explícita entre ellos.

Ciclo de Vida

1. Conectar a RabbitMQ
2. Leer modelo usando `basic_get()` + NACK
3. Procesar escenarios en loop infinito
4. Detectar cambio de modelo comparando variables
5. Recargar modelo automáticamente si es necesario

Detección de cambio de modelo: Los workers detectan cambios de modelo sin comunicación explícita mediante comparación de variables:

Algorithm 1 Detección automática de cambio de modelo

Require: escenario, modelo_actual
1: $vars\_escenario \leftarrow$ keys(escenario)
2: $vars\_modelo \leftarrow$ keys(modelo_actual.variables)
3: **if** $vars\_escenario \neq vars\_modelo$ **then**
4: $nuevo\_modelo \leftarrow$ leer_modelo()
5: $modelo\_actual \leftarrow nuevo\_modelo$
6: reiniciar_contador()
7: log("Modelo actualizado")
8: **end if**

Se comparan las variables del modelo cargado en el worker con las variables del escenario que está consumiendo. Si no coinciden, consulta a la cola de modelos para actualizar el nuevo modelo.

4.3 Dashboard

El dashboard coordina el sistema mediante comandos y visualiza resultados en tiempo real.

Arquitectura de Threads

- **Thread GUI**: Interfaz gráfica (Tkinter), envía comandos a la cola
- **Thread Resultados**: Consume resultados y actualiza estadísticas

Cada thread mantiene su propia conexión a RabbitMQ para garantizar thread-safety.

Detección de cambio de modelo: El dashboard detecta cambios leyendo el campo `modelo` incluido en cada resultado:

Algorithm 2 Actualización de estadísticas al cambiar modelo

Require: resultado
1: $modelo\_nuevo \leftarrow resultado.modelo$
2: **if** $modelo\_nuevo \neq modelo\_actual$ **then**
3: reiniciar_estadisticas()
4: actualizar_UI($modelo\_nuevo$)
5: log("Modelo cambiado: " + $modelo\_nuevo$)
6: **end if**
7: agregar_resultado(resultado)
8: actualizar_graficas()

Se compara el modelo elegido desde la GUI con el modelo precargado anterior. Si son diferentes, reinicia todas las estadísticas.

5 Implementación del sistema

La implementación del sistema se puede sintetizar en cuatro algoritmos principales

5.1 Generación de Escenarios Únicos

El productor garantiza unicidad absoluta mediante hashing:

Algorithm 3 Generación de escenario único con verificación

```
Require: modelo
escenarios_generados ← ∅
while generando and modelo_valido() do
    escenario ← {}
    for cada var en modelo.variables do
        escenario[var] ← generar_valor_aleatorio(var)
    end for
    hash ← calcular_hash(escenario)
    if hash ∉ escenarios_generados then
        escenarios_generados ← escenarios_generados ∪ {hash}
        publicar_mensaje(COLA_ESCENARIOS, escenario)
        total ← total + 1
    end if
    if total mod 100 = 0 then
        if not modelo_existe_en_cola() then
            log("Modelo expirado por TTL")
            break
        end if
    end if
    dormir(INTERVALO)
end while
```

Crea un escenario con variables aleatorias y mediante hashing busca si el escenario fue previamente generado. Si ya fue generado, se crean escenarios hasta alguno nuevo y se agrega a los escenarios creados. Esto nos permite garantizar que dos escenarios no se repitan y evitar computo repetido.

5.2 Procesamiento de Escenarios

Los workers evalúan el modelo de forma segura utilizando contextos aislados:

Algorithm 4 Procesamiento seguro de escenario

```
Require: escenario, modelo
verificar_compatibilidad(escenario, modelo)
contexto ← copiar(escenario)
resultado ← evaluar(modelo.formula, contexto)
mensaje ← {
        worker_id: MI_ID,
        resultado: redondear(resultado, 4),
        modelo: modelo.nombre,
        timestamp: tiempo_actual()
}
publicar_mensaje(COLA_RESULTADOS, mensaje)
confirmar_procesamiento()
```

La lectura del escenario es crucial para la simulación Montecarlo. Con el modelo previamente cargado, podemos realizar las operaciones y publicar los resultados para su posterior tratamiento.

5.3 Mecanismo de Cambio de Modelo con TTL

El cambio de modelo por parte del productor corresponde al siguiente algoritmo:

Algorithm 5 Cambio de modelo con TTL automático

```
Require: nuevo_modelo, TTL_milisegundos
limpiar_cola(COLA_ESCENARIOS)
mensaje ← serializar(nuevo_modelo)
propiedades ← {
        delivery_mode: PERSISTENTE,
        expiration: convertir_a_string(TTL)
}
publicar_mensaje(COLA_MODELO, mensaje, propiedades)
log("Modelo publicado con TTL: " + TTL + "ms")
iniciar_thread_generacion()
```

Para que un modelo expire, debe ocurrir que el TTL del modelo haya pasado o que un nuevo modelo haya sido insertado a la cola (mediante solicitud del dashboard). En cualquier caso, es importante limpiar la cola de escenarios para evitar remanentes que no correspondan a los cálculos deseados.

5.4 Lectura de Modelo sin Consumo

Algoritmo que permite a los workers leer el modelo sin eliminarlo de la cola:

Algorithm 6 Lectura no destructiva con NACK

```
(frame, properties, body) ← basic_get(COLA_MODELO)
if frame = None then
    return None
end if
modelo ← deserializar(body)
basic_nack(frame.delivery_tag, requeue=True) {Preserva TTL}
return modelo
```

El parámetro `requeue=True` es crítico: por medio de un NACK, garantiza que el mensaje vuelva a la cola manteniendo su TTL original, permitiendo que otros workers también lo lean.

6 Integración del Middleware

La decisión de integrar RabbitMQ como middleware en la arquitectura se fundamenta en los siguientes criterios:

Desacoplamiento: RabbitMQ proporciona desacoplamiento completo entre productores y consumidores:

- **Espacial**: Los componentes no necesitan conocer las direcciones de red de otros componentes. Un worker puede ejecutarse en cualquier máquina con acceso al broker.
- **Temporal**: Los productores pueden enviar mensajes aunque los consumidores no estén activos. Los mensajes se almacenan hasta que un consumidor esté disponible.

Sin middleware, se requeriría implementar manualmente:

- Gestión de conexiones punto a punto entre cada par de componentes
- Mecanismos de descubrimiento de servicios para localizar componentes dinámicamente
- Buffers de mensajes y políticas de reintento en caso de fallos temporales
- Sincronización compleja entre componentes heterogéneos

Persistencia y garantías de entrega: RabbitMQ ofrece colas durables con persistencia en disco, garantizando que los mensajes no se pierdan ante fallos del sistema. Debido a la importancia de computar todos los escenarios del modelo publicado sin pérdida de información, es importante garantizar la entrega de cada uno junto con los resultados obtenidos.

Time-To-Live (TTL): La funcionalidad de TTL es crítica para el mecanismo de cambio dinámico de modelo:

- El modelo se publica con TTL configurable (ej: 300 segundos)
- Después del TTL, RabbitMQ elimina automáticamente el mensaje
- Workers nuevos no pueden leer modelos expirados
- Workers existentes continúan con el modelo en memoria

Implementar TTL sin middleware requeriría:

- Agregar timestamps a cada mensaje manualmente
- Verificación periódica de expiración en cada componente
- Limpieza manual de mensajes viejos en estructuras de datos
- Sincronización de relojes entre nodos distribuidos

Quality of Service (QoS) RabbitMQ permite configurar QoS por consumer mediante el parámetro `prefetch_count`. Esto garantiza que:

- Cada worker procesa solo el número configurado de mensajes simultáneamente
- Workers lentos no se sobrecargan con mensajes que no pueden procesar
- Balanceo de carga automático basado en capacidad real de cada worker

En particular, para la arquitectura, se define `prefetch_count=1` para garantizar que, hasta que el worker haya procesado su escenario, puede recibir otro más.

Reconocimiento de Mensajes (ACK/NACK) El protocolo AMQP [8] implementado por RabbitMQ proporciona:

- **ACK**: Confirmar procesamiento exitoso, mensaje se elimina de la cola
- **NACK + requeue**: Devolver mensaje a la cola sin eliminarlo, preservando TTL
- **NACK sin requeue**: Rechazar mensaje permanentemente (dead-letter)

Esto permite el mecanismo de "leer modelo una vez sin consumirlo":

Algorithm 7 Lectura de modelo preservando TTL

```
mensaje ← cola_modelo.get()
modelo ← deserializar(mensaje)
cola_modelo.nack(mensaje, requeue=True) {Mensaje vuelve con TTL intacto}
return modelo
```

El uso de RabbitMQ como middleware ha ayudado a simplificar el proceso de comunicación, aportando a la arquitectura del sistema beneficios como:

- **Reducción de acoplamiento**: Los componentes solo necesitan conocer nombres de colas, no direcciones de otros componentes
- **Escalabilidad mejorada**: Agregar workers es trivial (conectar al broker y comenzar a consumir)
- **Tolerancia a fallos**: Si un worker falla, sus mensajes no confirmados se redistribuyen automáticamente
- **Flexibilidad**: Fácil agregar nuevos tipos de componentes (ej: logger, archivador)

Por estas razones, RabbitMQ no es simplemente una conveniencia sino un componente arquitectónico fundamental que define las capacidades del sistema.

7 Conclusiones

Este trabajo presentó un sistema distribuido completo para simulaciones Montecarlo basado en paso de mensajes. Las principales conclusiones son:

1. **Middleware es esencial**: El uso de RabbitMQ proporciona garantías de entrega confiables y facilita escalabilidad sin código adicional. No es una conveniencia sino un componente arquitectónico fundamental.
2. **TTL simplifica diseño**: El mecanismo de TTL de RabbitMQ permite cambio dinámico de modelos de manera conveniente, evitando complejidad de sincronización de relojes y verificación manual de expiración.
3. **Detección implícita robusta**: La detección de cambio de modelo por comparación de variables es más robusta y fácil de implementar que canales explícitos de notificación, eliminando un punto de fallo.
4. **Thread-safety para servicios separados**: Uso de conexiones separadas por thread es esencial para correctitud en sistemas con operaciones bloqueantes concurrentes.

El papel del middleware en sistemas distribuidos es importante al momento de comunicarnos entre componentes del sistema. Sin embargo, su implementación suele ser compleja, por lo que diseñarlo puede consumir mucho más tiempo, aunado a los servicios que también se deben programar. El uso de RabbitMQ como middleware de mensajería facilita la comunicación en sistemas distribuidos, permitiendo resolver el problema de comunicación y ocupándonos del diseño de los servicios y la correcta orquestación del sistema.

References

1. Metropolis, N., Ulam, S.: The Monte Carlo method. Journal of the American Statistical Association **44**(247), 335–341 (1949)
2. Kalos, M.H., Whitlock, P.A.: Monte Carlo Methods, 2nd edn. John Wiley & Sons (2008)
3. Tanenbaum, A.S., Van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Pearson Prentice Hall (2007)

4. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin/Cummings Publishing (1991)
5. Videla, A., Williams, J.J.: RabbitMQ in Action: Distributed Messaging for Everyone. Manning Publications (2012)
6. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design, 5th edn. Addison-Wesley (2011)
7. RabbitMQ Documentation. Consumer. Recuperado el 27 de noviembre de 2025 de https://www.rabbitmq.com/docs/consumers
8. RabbitMQ Documentation. AMQP 0-9-1 Model Explained. Recuperado el 27 de noviembre de 2025 de https://www.rabbitmq.com/tutorials/amqp-concepts.html