UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PATRONES DE DISEÑO PARA PREVENCIÓN Y CORRECCIÓN DE ERRORES EN COMPONENTES DE COMUNICACIÓN DE PROGRAMAS PARALELOS

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
RAÚL HERNÁNDEZ TOLEDO

TUTOR PRINCIPAL:
DR. JORGE LUIS ORTEGA ARJONA
FACULTAD DE CIENCIAS - UNAM

MÉXICO, D.F. DICIEMBRE DE 2012

Dedicatorias

- A toda la familia Cruz Arreola y en especial a Marta Arreola, llegar a esta meta fue en gran parte gracias a su invaluable apoyo e incondicional cariño. Dios la bendiga.
- A Liliana, por ser esa persona que me ha ayudado a superarlo todo: a tí te debo lo que ahora soy. Te refrendo mi amor con la frase del poeta: *El amor, con amor se paga.*
- A Sofía, que éste logro sea un motivo de orgullo para tí y represente siempre el compromiso de tus padres por ofrecerte un mundo mejor.
- Al Dr. Jorge Ortega Arjona, gracias por todo su apoyo durante el desarrollo de este trabajo y por compartir sus historias y experiencias. Pero gracias sobre todo, por ser una persona comprometida con su trabajo.
- A mis padres y hermanos, los amo con todo mi corazón, espero que este trabajo sea un motivo de orgullo.
- A mi amada UNAM, a la que le debo este sentimiento que nace de mi corazón, se convierte en un nudo en mi garganta y sale de mi boca como un grito de guerra: *México, Pumas, Universidad. Goya Goya ...*

Índice general

Resumen xiii

1. Introducción 1

1.1. El contexto 1
1.2. El problema 2
1.3. La hipótesis 3
1.4. La propuesta 3
1.5. Contribución 3
1.6. La estructura de la tesis 4

2. Antecedentes 7

2.1. La comunicación y sincronización entre procesos 7
2.1.1. Mecanismos de sincronización 7
2.1.2. Tipos de comunicación 8
2.1.3. La comunicación en procesos distribuidos 8
2.2. Introducción a las fallas en sistemas 10
2.2.1. Definición de clases de fallas 10
2.2.2. Seguridad, vitalidad y formas de tolerancia a fallas 12
2.2.3. Clasificación de las fallas 13
2.2.4. División del estudio de la tolerancia a fallas 15
2.2.5. Relación entre faults, errors y failures 15
2.3. Introducción al procesamiento paralelo 16
2.3.1. Paralelización de un programa 17
2.3.2. Niveles de paralelismo 18
2.4. Patrones de software 19
2.4.1. Qué es un patrón 20
2.4.2. Cómo se describe un patrón de software 21

2.4.3. Categorización de los patrones de software 22
2.5. Resumen 24

3. Trabajo Relacionado **25**
3.1. Un sistema de patrones de software para tolerancia a fallas 25
3.1.1. Fail-Stop Processor (Procesador falla-parada) 27
3.1.2. Acknowledgment (Reconocimiento) 27
3.1.3. I Am Alive (Estoy vivo) 29
3.1.4. Are You Alive (¿Estas vivo?) 30
3.1.5. Roll Forward (Puesta al día) 31
3.1.6. Rollback (Retroceso) 32
3.1.7. Passive Replication (Replicación Pasiva) 34
3.1.8. Semi-Passive Replication (Replicación Semi-Pasiva) 35
3.1.9. Semi-Active Replication (Replicación Semi-Activa) 37
3.1.10. Active Replication (Replicación Activa) 37
3.2. Patrones de diseño para componentes de comunicación de programas paralelos 39
3.2.1. Message Passing Pipe (Tubería de Paso de Mensajes) 40
3.2.2. Message Passing Channel (Canal de Paso de Mensajes) 41
3.3. Resumen 44

4. Propuesta de patrones de software **45**
4.1. Descripción de la clase de fallas 45
4.1.1. Falla por error en la asignación de memoria en el buffer 46
4.1.2. Falla por error en el link de comunicación 49
4.1.3. La clase de fallas. 50
4.2. Descripción de los patrones de software 51
4.2.1. Paso de Mensajes Paralelo con Buffer no Delimitado 51
4.2.2. Monitor de Paso de Mensajes Paralelo 58
4.2.3. Monitor de Canal de Paso de Mensajes Paralelo 65
4.2.4. Resumen 71

5. Evaluación experimental de los patrones propuestos **73**
5.1. Evaluación del patrón de Paso de Mensajes Paralelo con Buffer no Delimitado 74
5.1.1. Descripción de la implementación de los participantes en el patrón 74
5.1.2. Ejecución del programa paralelo 79
5.2. Evaluación del patrón Monitor de Paso de Mensajes Paralelo 82
5.2.1. Descripción de la implementación de los participantes en el patrón. 83
5.2.2. Ejecución del programa paralelo 88
5.3. Evaluación del patrón Monitor de Canal de Paso de Mensajes Paralelo 93
5.3.1. Descripción de la implementación de los participantes en el patrón 93
5.3.2. Ejecución del programa paralelo 98
5.3.3. Resumen 102

6. Conclusiones **103**
6.1. Resultados del trabajo de investigación 103
6.1.1. Conclusiones que reflejan sus virtudes 104
6.1.2. Conclusiones que reflejan sus defectos. 105
6.2. Trabajo futuro 105

Bibliografía **107**

Índice de figuras

2.1. Comunicación síncrona. 9
2.2. Comunicación asíncrona. 9
2.3. Un simple programa tolerante a fallas. 11
2.4. Técnicas de la tolerancia a fallas. 16
2.5. El camino de un servicio correcto a un servicio incorrecto. 16
2.6. Pasos para la paralelización de un programa secuencial. 18
2.7. Clasificación de los patrones de software según el nivel de abstracción. 23

3.1. Estructura y diagrama de actividad del patrón Fail-Stop Processor . . 28
3.2. Estructura y diagrama de actividad del patrón Acknowledgment . . . 29
3.3. Estructura y diagrama de actividad del patrón I Am Alive. 30
3.4. Estructura y diagrama de actividad del patrón Are You Alive. 32
3.5. Estructura y diagrama de actividad del patrón Roll Forward. 33
3.6. Estructura y diagrama de actividad del patrón Rollback. 34
3.7. Estructura y diagrama de actividad del patrón Passive Replication. . 35
3.8. Estructura y diagrama de actividad del patrón Semi Passive Replication. 36
3.9. Estructura y diagrama de actividad del patrón Semi Active Replication. 38
3.10. Estructura y diagrama de actividad del patrón Active Replication. . . 39
3.11. Diagrama de colaboración del patrón Message Passing Pipe. 41
3.12. Diagrama de secuencia del patrón Message Passing Pipe. 42
3.13. Diagrama de colaboración del patrón Message Passing Chanel 43
3.14. Diagrama de seuencia del patrón Message Passing Chanel 43

4.1. Falla en la comunicación por error de asignación de memoria al buffer. 48
4.2. Falla por un error en el link de comunicación. 49
4.3. Diagrama de colaboración del patrón Paso de Mensajes Paralelo con Buffer No Delimitado. 53

4.4. Diagrama de secuencia del patrón Paso de Mensajes Paralelo con Buffer no Delimitado. 55
4.5. Diagrama de flujo para el monitor del buffer. 56
4.6. Falla en la transmisión del mensaje. 59
4.7. Diagrama de colaboración del patrón Monitor de Paso de Mensajes Paralelo. 61
4.8. Diagrama de secuencia del patrón Monitor de Paso de Mensajes. . . . 62
4.9. Un AC que simula la propagación de fuego en un bosque. 65
4.10. Distribución del AC en un array de procesadores. 66
4.11. Estructura del patrón Monitor de Canal de Paso de Mensaje Paralelo. 68
4.12. Diagrama de secuencia del patrón Monitor de Canal de Paso de Mensaje Paralelo. 69

5.1. El diagrama de clases del programa paralelo. 80
5.2. Ejecución del programa paralelo sin monitor de buffer. 81
5.3. Consumo de la memoria del proceso PipeClassB. 82
5.4. Ejecución del programa paralelo con monitor de buffer. 83
5.5. El diagrama de clases de la implementación del patrón Monitor de Paso de Mensajes Paralelo. 88
5.6. Escenarios para el envío de mensajes. 89
5.7. Ejecución del programa sin errores de comunicación. 89
5.8. Ejecución del programa paralelo con perdida de mensaje. 90
5.9. Ejecución del programa paralelo con reenvío de mensajes. 91
5.10. Ejecución del programa paralelo con un error en el envío del ACK. . . 91
5.11. El mensaje ACK no llega a su destino y provoca reenvío de datos. . . 92
5.12. El diagrama de clases de la implementacion del patrón Monitor de Canal de Paso de Mensajes Paralelo. 94
5.13. Diagrama de flujo de la clase pipeClassA. 100
5.14. Ejecución del programa sin errores. 101
5.15. Ejecución del programa paralelo con un error en el envío del mensaje. 101

Índice de tablas

2.1. Cuatro modelos de tolerancia a fallas. 13

3.1. Clasificación del sistema de patrones para tolerancia a fallas. 26
3.2. Clasificación de los patrones de diseño para componentes de comunicación. 40

4.1. Resumen de la clase de fallas. 50

Resumen

La comunicación entre dos o más procesos que trabajan con un objetivo en común es parte fundamental en el procesamiento paralelo. Diseñar un mecanismo que permita la comunicación entre procesos requiere de un análisis detallado del problema, así como de la plataforma de hardware en la cual se va a implementar la solución.

La presencia de errores durante la comunicación puede provocar una falla que desvíe al programa de su funcionamiento correcto, o incluso detener su ejecución. Es por ello que es importante contemplar mecanismos que prevengan o corrijan errores, con el fin de garantizar en lo más posible una comunicación acorde a la especificación del programa.

Se presenta una serie de patrones de software que describen mecanismos de prevención y corrección de errores en el diseño de componentes de comunicación. Estos patrones son una variante a algunas soluciones que ya existen en la literatura, pero que no contemplan la presencia de errores en el caso del diseño de componentes de comunicación, o no fueron diseñados para procesos paralelos, en el caso del tratamiento de fallas.

Por último se presenta una evaluación de los mecanismos propuestos en los patrones presentados en un lenguaje de programación orientado a objetos.

Capítulo 1

Introducción

Sale el sol, y se pone el sol, y se apresura
a volver al lugar de donde se levanta.
El viento tira hacia el sur, y rodea al norte;
va girando de continuo,
y a sus giros vuelve el viento de nuevo.
Eclesiastés 1:5-6

1.1. El contexto

Un programa paralelo, visto desde una enfoque arquitectónico, es la unión de dos o más componentes de software cuyas funciones son: *(1)* realizar el cómputo encargado de procesar la información que resuelve el problema y *(2)* hacer posible el intercambio de información entre procesos, es decir, de establecer la comunicación entre ellos. Los primeros se conocen como componentes de procesamiento y los segundos como componentes de comunicación [1].

El diseño de ambos componentes está sujeto al tipo de paralelismo intrínseco en el problema a resolver y la organización de memoria en la que se desea implementar la solución [1]. Independientemente de las combinaciones que surjan entre estos dos factores, si el proceso de comunicación no se realiza conforme a la especificación del programa, no se tiene garantía de un resultado correcto, o incluso, puede provocar el término prematuro del mismo.

Esta tesis se enfoca sólo en el componente de comunicación como una pieza de software que no está exenta a la presencia de errores que puedan provocar una falla en su servicio. Evitar una falla depende de la capacidad del componente de comunicación para prevenir o corregir errores durante su funcionamiento.

Proveer la capacidad de prevención o corrección de errores es una tarea que requiere de la implementación de mecanismos que tomen en cuenta los elementos que conforman un componente de comunicación de un programa paralelo y la dinámica existente entre ellos.

1.2. El problema

La presencia de un error en algún proceso del componente de comunicación puede derivar en una falla que provoque que la comunicación no se realize de forma correcta o sea interrumpida. Si tal es el caso, existen dos posibles escenarios: un resultado final incorrecto, o la propagación de la falla al resto de los componentes y la terminación del programa paralelo. Ambos casos son indeseables y costosos en un procesamiento paralelo.

Supóngase el caso de un procesamiento que requiere de largos periodos de tiempo de cómputo, y un problema en la comunicación que obliga a desechar todo el cómputo [2]; o el caso en que un resultado correcto de un programa paralelo es crítico y una falla en la comunicación provoca un resultado incorrecto del programa.

La dificultad de diseñar software tolerante a fallas se debe a la variedad de errores que pueden surgir y las diferentes consecuencias que pueden provocar. Un programa paralelo incorpora procesos que no se encuentran en un programa secuencial y que requieren de un esfuerzo de diseño adicional [3], por lo que a la complejidad de un mecanismo para prevenir o corregir errores se le suma la complejidad de no alterar el funcionamiento de estos procesos adicionales.

En el caso particular de un componente de comunicación de un programa paralelo, un diseño que incorpore mecanismos de prevención o correción de errores no debe romper con el objetivo principal del componente: el intercambio síncrono o asíncrono de datos entre componentes de procesamiento.

1.3. La hipótesis

La hipótesis de esta tesis es la siguiente:
"¿Es posible especificar, mediante patrones de software, las acciones necesarias a realizar ante la presencia de errores en la comunicación entre componentes de procesamiento de un programa paralelo, que utiliza una organización de memoria distribuida, con el fin de asegurar la comunicación entre los componentes implementados para la comunicación?"

1.4. La propuesta

En un programa paralelo distribuido es indispensable que la comunicación entre componentes de procesamiento esté garantizada. La importancia de que la comunicación se realize aún ante la presencia de errores, radica en evitar que ocurran fallas que provoquen el desperdicio de todo el cómputo realizado.

La propuesta de esta tesis es establecer una solución, expresada en patrones de software, que proporcione un diseño para la construcción de componentes de comunicación de programas paralelos distribuidos, capaz de prevenir o corregir errores en la comunicación.

La manera de poner a prueba la efectividad de la solución propuesta es mediante la implementación, en un lenguaje orientado a objetos, de los patrones propuestos, en la cual se simulan los errores para los cuales ofrecen una solución.

1.5. Contribución

El objetivo principal de esta tesis es describir una serie de patrones de software que describan los mecanismos que puedan garantizar la comunicación entre procesos de programas paralelos distribuidos.

Bajo esta consideración, las contribuciones principales son las siguientes:

1. La descripción de patrones de software para el diseño de componentes de comunicación de programas paralelos distribuidos, que incorporan mecanismos de prevención y correción de errores en la comunicación entre componentes de procesamiento.

2. El desarrollo de un programa paralelo distribuido basado en los patrones propuestos, que integre un componente que simule los errores para los cuales los patrones ofrecen una solución.

Para la descripción de los patrones de software se utiliza la forma POSA, descrita en [4], y para el desarrollo del programa paralelo distribuido se utiliza el lenguaje orientado a objetos Java.

1.6. La estructura de la tesis

La estructura de la tesis es la siguiente:

- **Capítulo 2. Antecedentes.** En este capítulo se da una introducción a la temas relacionados con esta tesis:
  - Los tipos de comunicación existentes entre los procesos que conforman un programa paralelo y su relación con la organización de memoria utilizada.
  - Una introducción a la programación paralela, el objetivo de un programa paralelo, los pasos a seguir para paralelizar un programa secuencial y la clasificación de los tipos de paralelismo.
  - Se introduce al concepto de tolerancia a fallas en sistemas de cómputo, el origen y la clasificacion de las fallas, la división del estudio de la tolerancia a fallas, así como las caracteristicas que debe presentar un sistema distribuido considerado un sistema tolerante a fallas.
  - Una introducción a los patrones de software, la forma en que son descritos y su clasificación.
- **Capítulo 3. Trabajo Relacionado.** En este capítulo se presenta un resumen del trabajo relacionado al tema de interés de esta tesis. Los patrones de software que se han desarrollado para el diseño de componentes de comunicación de programas paralelos distribuidos y aquéllos orientados a la tolerancia a fallas en sistemas distribuidos.
- **Capítulo 4. La descripción de los patrones de software.** En este capítulo se describe una conjunto de errores que delimitan el alcance de las soluciones

propuestas. Con base en este conjunto de errores se realiza la descripción de los patrones de software propuestos.

- **Capítulo 5. Implementación de los patrones de software.** En este capítulo se realiza una implementación de cada uno de los patrones de software propuestos, así como la ejecución de cada una de estas implementaciones en escenarios libres de errores y en escenarios en donde se simulan los errores, para los cuales ofrecen una solución.

- **Capitulo 6. Conclusiones.** En este capítulo se presenta una discusión y análisis de los resultados obtenidos, así como del trabajo futuro a realizar en el tema.

Capítulo 2

Antecedentes

We are accidents
waiting waiting to happen.
There There, Radiohead's song

El objetivo de este capítulo es presentar: *(1)* una introducción a los tipos de comunicación entre procesos de un programa paralelo; *(2)* una descripción y clasificación de fallas en sistemas; *(3)* una introducción a la programación paralela; y *(4)* una introducción a los patrones de software.

2.1. La comunicación y sincronización entre procesos

Una comunicación entre procesos concurrentes o paralelos es el intercambio de información que permite un trabajo cooperativo con el fin de resolver una tarea en común [5]. Mediante este intercambio de información, se realiza la coordinación de los procesos, la cual sólo es posible mediante un mecanismo de sincronización.

2.1.1. Mecanismos de sincronización

En 1968, E. W. Dijsktra propone un primer mecanismo de sincronización en [5]. Este mecanismo, conocido como semáforo, plantea el uso de variables compartidas, las cuales son modificadas de forma no determinística por todos y cada uno de los procesos en ejecución mediante primitivas. El problema de establecer qué proceso

tiene acceso a la variable compartida y al mismo tiempo bloquearla para el resto de los procesos, dio lugar a problemas clásicos como la exclusión mutua.

El mecanismo de sincronización mediante semáforos es retomado en [6], donde se propone una notación estructurada del concepto de sección crítica, modularizando el código referente a la sección crítica y redefiniéndola como región crítica. Tan solo dos años después, en [7] se presenta la versión orientada a objetos: los monitores. En esta base teórica, se consideran comunicaciones entre procesos con una organización de memoria compartida. Es en [8] en donde se especifica la sincronización de procesos en una organización de memoria distribuida.

2.1.2. Tipos de comunicación

Una comunicación entre dos procesos puede llevarse a cabo de dos distintas maneras: de forma *síncrona* o de forma *asíncrona* [9, 10].

Las caracteristicas de ambas son las siguientes [10]:

- **Comunicación síncrona.** En este modelo de comunicación, el proceso servidor que envía un mensaje es bloqueado hasta que el proceso cliente haya recibido el mensaje. Por tanto, para un intercambio de información (por medio de un mensaje), ambos procesos deben estar sincronizados.

- **Comunicación asíncrona.** En este modelo, el proceso servidor no tiene que esperar una respuesta del proceso cliente. Si éste no puede atender la petición, entonces es almacenada temporalmente en un buffer. Mientras tanto, el proceso servidor puede seguir enviando mensajes a otros procesos.

En las figuras 2.1 y 2.2 se observa gráficamente la diferencia entre estos tipos de comunicación.

Al implementar una comunicación asíncrona, se puede obtener un grado más alto de paralelismo con respeto al síncrono [11], con la desventaja de tener que crear un mecanismo complejo de administración del buffer.

2.1.3. La comunicación en procesos distribuidos

En los procesos distribuidos, existen dos mecanismos para establecer una comunicación: paso de mensajes y llamadas a procedimientos remotos (RPC) [10].

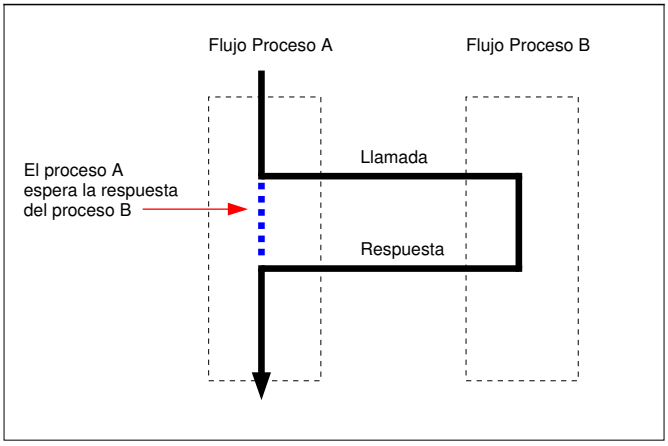

Figura 2.1: Comunicación síncrona.

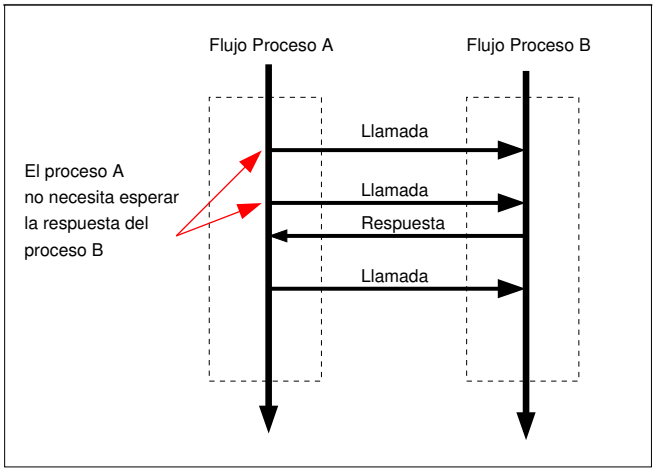

Figura 2.2: Comunicación asíncrona.

- **Paso de mensajes.** La información es transmitida punto a punto de un procesador a otro por medio de operaciones de envío y recepción. Estas operaciones están definidas dentro de una biblioteca, por lo que si dos procesos necesitan intercambiar información, solo es necesario que activen la operación específica de comunicación [12].

  Las bibliotecas podrían estar ligadas a un hardware o plataforma específica, lo cual se traduce a problemas de portabilidad, razón por la cual se crea el estándar para el paso de mensajes conocido como Interfaz de Paso de Mensajes (MPI), especialmente diseñada para aplicaciones paralelas [10].

  Las operaciones comúnmente utilizadas para implementar una comunicación mediante paso de mensajes son: la operación *send*, en la que se especifica el

proceso o máquina destino del mensaje, y la operación *receive*, que obtiene los datos recibidos y los almacena en un buffer local [13].

- **Llamadas a Procedimientos Remotos (RPC).** La comunicación se establece haciendo llamadas a procedimientos que se ubican en otras computadoras conectadas por una red. Estas llamadas funcionan de manera similar a un llamada a un procedimiento local: la información es enviada dentro de los parámetros y regresa en un procedimiento resultante [10].

  Las llamadas a procedimientos remotos implementan una comunicación síncrona, lo cual funciona bien dentro de ciertos escenarios, como por ejemplo, en el desarrollo de sistemas de 3 capas o n-capas. Pero su naturaleza fuertemente acoplada es un obstáculo en procesamientos de sistema a sistema [14].

2.2. Introducción a las fallas en sistemas

Un sistema tolerante a fallas se puede definir como un sistema que: *(1)* es capaz de detectar fallas; *(2)* una vez detectadas, sea capaz de contenerlas; y *(3)* es capaz de corregirlas en un futuro [15]. Una falla es cualquier cosa que sucede en un sistema que lo desvía de un comportamiento correcto, y que puede originarse por diversos motivos: fenómenos naturales, fallas en el hardware, e incluso, por acciones humanas accidentales o intencionales [16].

Lograr que un sistema en general sea capaz tolerar fallas es una tarea compleja y difícil [11]. Requiere un esfuerzo extra del que conlleva la solución misma del problema, debido a que debe de llevar el control de sus propias actividades y también de las complejas situaciones que pueden ocurrir cuando alguno de sus componentes presenta una falla [17].

2.2.1. Definición de clases de fallas

La detección de errores y el conocimiento de su causa se considera una parte esencial dentro del computo tolerante a fallas [18]. Durante el diseño de un sistema tolerante a fallas, un primer requisito es especificar una *clase de fallas* que contiene aquellos errores que es capaz de tolerar el sistema.

Formalmente una *clase de fallas* es definida de la siguiente manera [19]:

DEFINICIÓN **1** *Una clase de fallas para* $\mathtt{p}$ *es un conjunto de acciones sobre las variables de* $\mathtt{p}$.

donde p es cualquier programa definido como un conjunto de variables y un conjunto finito de acciones. Para ejemplificar esta definición se muestra en la figura 2.3 el ejemplo descrito en [18].

process from example
var $x \in \{0, 1, 2, 3\}, x := 1$
  begin
    {* acciones normales del programa: *}
    $x = 1 \rightarrow x := 2$
  [] $x = 2 \rightarrow x := 1$
    {* fallas que deben ser toleradas: *}
  [] **true** $\rightarrow x := 0$
    {* mecanismo protector *}
  [] $x = 0 \rightarrow x := 1$
  end

Figura 2.3: Un simple programa tolerante a fallas.

En el ejemplo de la figura 2.3 la clase de fallas está conformada por la sentencia:

$$true \rightarrow x := 0$$

donde *true* puede ser reemplazado por una sentencia lógica que compruebe que ocurre un error. Por ejemplo, si el programa indica que x siempre toma el valor de 1 ó 2, y en algún momento adquiere otro valor (suponiendo que se agregan más líneas de código que interactúan con la variable), se considera un error. Por lo que la sentencia puede ser reemplazada por:

$$\mathsf{if}\ (x! = 1 \wedge x! = 2) \rightarrow x = 0.$$

Siguiendo con el ejemplo de la figura 2.3, la línea de código $x \in \{0, 1, 2, 3\}$ representa una *afirmación invariante*, la cual es una propiedad de todos los estados del sistema que siempre es cierta para cualquier ejecución [20]. Con estos elementos definidos,

se puede mencionar una definición formal de un sistema que tolera cierta clases de fallas [18, 19]:

Otra definición es la siguiente [18]:

DEFINICIÓN **2** *Un programa distribuido* `A` *se dice que es tolerante a fallas de una clase de fallas* `F` *para una invariante* `P` *si y solo si existe un predicado* `T` *para los siguientes tres requerimientos:*

- *Cualquier configuración donde* `P` *está contenido, también está contenido* `T` *(es decir,* $P \Rightarrow T$*).*
- *Iniciando en cualquier estado donde* `T` *está contenido, si cualquier acción de* `A` *o* `F` *es ejecutada, el estado resultante siempre es uno que esté contenido en* `T` *(es decir,* `T` *es cerrado en* `A`*, y* `T` *es cerrado en* `F`*).*
- *Iniciando en cualquier estado en* `T`*, cada cómputo que ejecute solo acciones de* `A`*, eventualmente alcanza un estado donde* $\boldsymbol{P}$ *está contenido.*

En resumen, la definición significa que si un programa `A` es tolerante a las fallas de la clase de fallas `F` para una invariante `P`, se afirma entonces que `A` es `F`-tolerante para `P`.

Para el ejemplo de la figura 2.3, las tres componentes son:

1. La invariante $\texttt{P} \equiv x \in \{1, 2\}$.
2. La clase de fallas es $\texttt{F} = \{true \rightarrow x := 0\}$.
3. El predicado es $\texttt{T} \equiv x \in \{0, 1, 2\}$.

2.2.2. Seguridad, vitalidad y formas de tolerancia a fallas

En 1977 Leslie Lamport hace referencia a una necesidad de verificar la *corrección* de programas multiproceso, debido a la prevalencia de errores. Para ello, propone probar esencialmente dos diferentes tipos de propiedades de un programa: *safety (seguridad)* y *liveness (vitalidad)* [21].

La propiedad *safety* significa que *algo no pasará* [21] o *algo malo no pasará* [18] dentro del sistema. *Liveness* significa que *algo debe pasar* [21] o *algo bueno sucederá* [18] durante la ejecución del sistema.

Bajo la premisa de cumplir esas dos propiedades, existen cuatro modelos en que un sistema distribuido puede cumplir con la tolerancia a fallas [18]. Estas formas están descritas en la tabla 2.1.

	live	**not live**
safe	masking	fail-safe
not safe	non-masking	none

Tabla 2.1: Cuatro modelos de tolerancia a fallas.

De la tabla anterior, si un sistema cumple con la propiedad *safe* y *live* bajo la presencia de una clase de fallas `F`, se dice el sistema está *enmascarando (masking)* las fallas. El término *fail-safe* indica que el programa cumple con la propiedad *safety*, pero el sistema no continúa ejecutándose. El término *non-masking* indica un comportamiento erróneo del sistema, pero continúa ejecutándose. Finalmente *none* indica que no hay ninguna garantía ante la presencia de fallas.

2.2.3. Clasificación de las fallas

En el diseño de un sistema tolerante a fallas, un arquitecto o desarrollador de software debe ser capaz de identificar el tipo de fallas que pueden presentarse. Por tal motivo, para poder realizar un sistema fiable, es decir, que sea disponible, confiable, seguro y altamente mantenible, es necesario conocer los modelos de fallas que existen [10].

Existen dos grandes clasificaciones de fallas: respecto a su duración y respecto a su tipo.

Clasificación con respecto a su duración

En [16] y en [10] se menciona la clasificación de las fallas con respecto a su duración:

- **Transitorias.** Algún evento aislado causa la falla.
- **Intermitentes.** Fallas que son recurrentes no necesariamente en un periodo regular de tiempo.
- **Permanentes.** Son fallas que ocurren y persisten hasta que el desperfecto es reparado.

Clasificación con respecto a su tipo

Existen varios enfoques en ésta clasificación de fallas. La propuesta en [11] establece

el concepto de servidor, como el hardware o software, capaz de ofrecer un servicio. Teniendo en cuenta este concepto, la clasificación es la siguiente:

- **Falla de congelación.** El servidor simplemente deja de funcionar, no recibe ni emite respuesta. Generalmente la única solución a esta falla es reiniciar el servidor.

- **Falla de omisión.** El servidor omite el envío de una petición o la respuesta de la misma. Es posible que el servidor nunca haya obtenido la solicitud o en caso contrario que haya realizado su trabajo pero algo resulto mal al enviar la respuesta.

- **Falla de tiempo.** La respuesta se genera después del intervalo de tiempo esperado. En este caso, el servidor responde tarde debido a un pobre desempeño.

- **Falla de respuesta.** La respuesta del servidor es incorrecta. Es posible que el valor sea incorrecto o el estado de transición que toma es incorrecto.

- **Falla arbitraria.** El servidor produce respuestas arbitrarias en cualquier momento. Esta falla también se le conoce como una falla bizantina.

Otra clasificación se encuentra en [22], la cual coincide con la anterior en cuanto a las fallas de omisión y las arbitrarias, pero difiere en las siguientes:

- **Fallas fail-stop.** Provocan que la ejecución de un sistema cese sin producir ninguna salida y el entorno es capaz de detectarla.

- **Fallas crash.** Provocan que la ejecución cese sin producir ninguna salida, sin embargo, la falla no es detectable por el entorno.

Aunque la clasificación presentada es clara respecto al origen de la falla, no es una tarea trivial poder distinguir la una de la otra. En la mayoría de las veces no importa qué tipo de falla se presenta, sino sólo importa que el sistema sea capaz de tolerarla [23].

2.2.4. División del estudio de la tolerancia a fallas

Para construir un sistema fiable, es decir, tolerante a fallas, existen cuatro aspectos fundamentales necesarios de estudio [23]:

1. *Detección de errores.* Es la capacidad de un sistema de poder detectar un error, la cual es fundamental, ya que si no la posee el sistema simplemente deja de funcionar.

2. *Recuperación ante fallas.* Una vez que el sistema detecta un error, deber cubrir o enmascarar los efectos del mismo.

3. *Diagnósticos de fallas.* Es la capacidad de localizar el dónde y cómo se originó una falla.

4. *Auto-Reparación.* Una vez diagnosticada la falla, el sistema debe ser capaz de poder auto-recuperarse o auto-reconfigurarse y seguir con un funcionamiento correcto.

Cada uno de estos aspectos es un amplio campo de estudio. Los aspectos más importantes para desarrollar un sistema tolerante a fallas son la detección y la recuperación de errores. Un sistema más robusto es capaz de diagnosticar fallas y auto-repararse. En [24] se resume de manera clara las técnicas que cubren el estudio de la tolerancia a fallas y se resume en la figura 2.4.

2.2.5. Relación entre faults, errors y failures

A continuación se muestran una serie de definiciones que complementan el enfoque sobre tolerancia a fallas de este trabajo. Las siguientes definiciones se encuentran en [24].

- **Servicio correcto (Correct service).** Un servicio correcto es entregado cuando el servicio implementa la función del sistema, entendiendo por función todo aquéllo que el sistema intenta *hacer* y que está descrito en una especificación funcional.

- **Servicio fallido (Service Failure).** Es la transición de un servicio correcto a un servicio incorrecto. El evento que lleva de un estado a otro se llama *failure*.

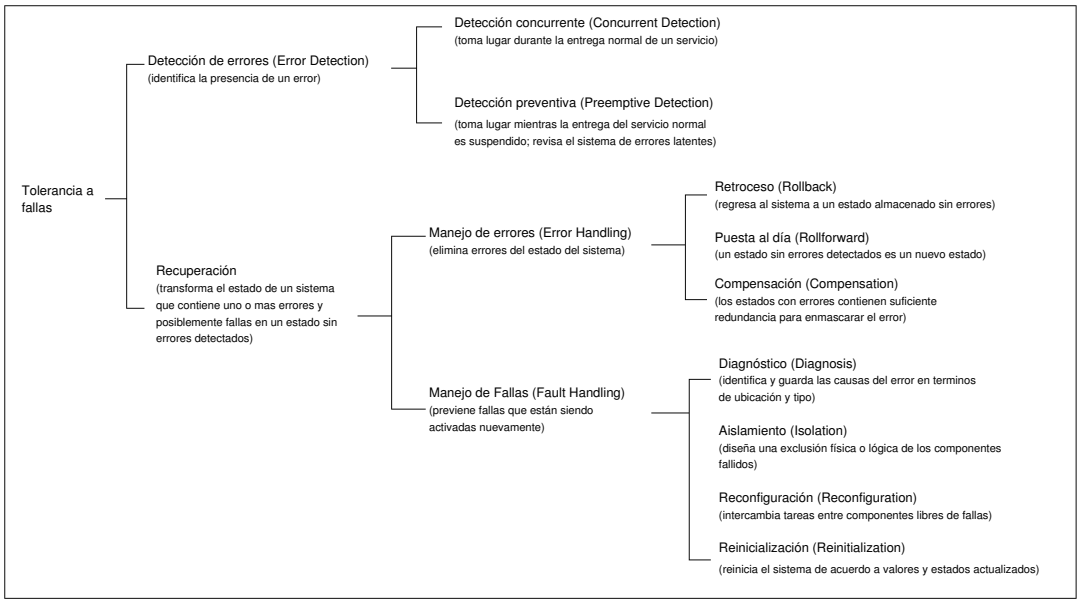

Figura 2.4: Técnicas de la tolerancia a fallas.

- **Error.** Dado que un sistema es una secuencia de estados y que un servicio fallido significa que al menos alguno de esos estados se desvía de un servicio correcto, entonces a esa desviación se le conoce como *error*.
- **Fault.** La causa juzgada o la hipótesis de un error se conoce como *fault*. Es decir, es aquello que origina el error.

De manera gráfica, en la figura 2.5 se muestra el camino de un servicio correcto a un servicio incorrecto.

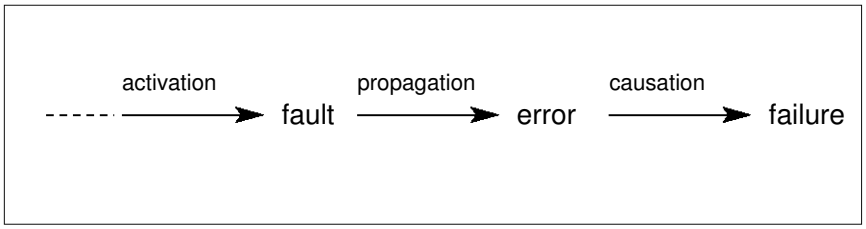

Figura 2.5: El camino de un servicio correcto a un servicio incorrecto.

2.3. Introducción al procesamiento paralelo

El procesamiento paralelo es la descomposición del cómputo de una aplicación en dos o más tareas que se ejecutan de manera simultánea en dos o más procesadores [25].

Con la aparición de procesadores con varios núcleos o *cores*, es necesario agregar a esta definición que el procesamiento paralelo también puede ejecutarse sobre un solo procesador con dos o más cores [12].

2.3.1. Paralelización de un programa

El objetivo de un programa paralelo es solucionar un problema de una manera más rápida con respecto a su solución secuencial. La transformación de un programa secuencial a un programa paralelo se conoce como *paralelización*. En [26] se enumeran tres pasos para lograrla de una manera sistemática:

1. **Descomposición de la computación.** El objetivo de este paso es descomponer un algoritmo secuencial en unidades pequeñas de paralelismo, conocidas como *tareas*. Dependiendo de la organización de la memoria, estas tareas interactúan mediante el acceso a espacios de memoria (memoria compartida) o mediante paso de mensajes (memoria distribuida). La creación de estas tareas puede realizarse de forma estática al inicio del procesamiento o dinámicamente durante la ejecución del proceso.

   El tiempo de ejecución de una tarea se conoce como **granularidad**, el cual está ligado totalmente a la naturaleza del problema. Es responsabilidad del programador tener en cuenta la granularidad del problema y decidir el número de tareas que se generan, ya que un número excesivo de tareas puede generar un tiempo mayor de ejecución comparado con la solución secuencial y, por otro lado, en un número escaso de tareas es posible que no se aproveche del todo la paralelización.

2. **Asignación de tareas a procesadores o hilos.** La asignación de tareas se debe diseñar teniendo en cuenta el número de accesos a memoria y la comunicación entre las tareas. El objetivo es lograr un buen balanceo de cargas, es decir, que cada proceso o hilo de ejecución tenga la misma carga de cómputo a ejecutar.

3. **Mapeo de procesos o hilos a procesadores físicos o cores.** El mapeo de procesos es necesario cuando el número de procesos o hilos es mayor al número de procesadores y tiene como objetivo hacer un uso igualitario de los procesadores o cores disponibles.

Un paso intermedio entre la asignación de tareas a procesadores o hilos y el mapeo se menciona en [13]: la **orquestación**. Como su nombre lo indica, significa orquestar u organizar los accesos a los datos, las comunicaciones y la sincronización de todos los procesos.

El objetivo de una buena orquestación es la de reducir el costo de la comunicación y sincronización. Por tanto, el diseñador debe elegir las primitivas apropiadas para obtener una simple, eficiente y exitosa orquestación. En la figura 2.6 se describe el proceso completo [13].

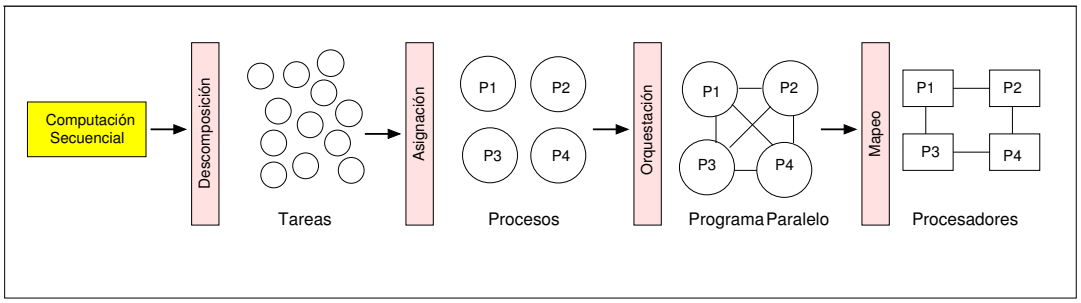

Figura 2.6: Pasos para la paralelización de un programa secuencial.

2.3.2. Niveles de paralelismo

La naturaleza de un problema establece las oportunidades que tiene el programador para paralelizar la solución [3]. A través de un análisis minucioso del problema, es posible distinguir qué nivel de paralelismo existe en el problema. A continuación se menciona una clasificación de los niveles de paralelismo [12]:

1. **Paralelismo a Nivel de Instrucción.** Las instrucciones de un programa pueden ser ejecutadas en forma paralela sí y solo sí son independientes entre ellas. Es decir, durante la ejecución de las múltiples instrucciones no existe dependencia de datos, por ejemplo, que una instrucción no necesite del resultado de otra para continuar su ejecución.

   En caso de haber dependencia de datos, no es posible utilizar este nivel de paralelismo. En [26] se muestran tres tipos se dependencia de datos:

- **Dependencia de Flujo.** Una variable o registro utilizado por un proceso es utilizado como operando en otro proceso.
- **Anti-Dependencia.** Significa que una variable que es utilizada como un operando por un proceso es utilizada para almacenar el resultado de otro proceso.
- **Dependencia de Salida.** Dos procesos utilizan la misma variable para almacenar el resultado.

2. **Paralelismo de datos.** El paralelismo de datos implica distribuir los datos a través de múltiples procesos, siempre y cuando las operaciones que se apliquen a los datos sean iguales y no haya dependencia entre ellas.

3. **Paralelismo Funcional.** En este nivel, se tienen partes del programa que son independientes y que pueden ejecutarse al mismo tiempo. Estas partes independientes pueden ser desde simple sentencias, bloques de código, ciclos, hasta llamadas a funciones.

En [3] se describe otra clasificación del nivel de paralelismo: **paralelismo de actividad**. Este nivel de paralelismo es una combinación del paralelismo de datos y el paralelismo funcional, es decir, el problema puede ser particionado tanto por los datos como por el algoritmo.

2.4. Patrones de software

Un patrón de software es un esquema de solución que proporciona conocimiento sobre problemas comunes en el desarrollo de software. El concepto de patrón de software toma prestado los fundamentos de patrones desarrollados para la arquitectura, integrándolos al ámbito del desarrollo de software.

Debido a que los patrones representan una evolución importante en la abstracción y reutilización del software [27], los mismos objetivos buscados por la programación orientada a objetos, su uso se ha popularizado entre los diseñadores y desarrolladores de software, quienes han buscado estandarizar cada vez más todo tipo de problemas relacionados con el desarrollo de software.

Es importante entonces conocer qué es un patrón de software, la forma en que éste se documenta y su clasificación.

2.4.1. Qué es un patrón

Un patrón es la especificación de una solución exitosa. Esta solución es documentada con el objetivo de compartir el conocimiento obtenido de experiencias anteriores, esperando sea útil en la resolución de problemas similares.

Es necesario que esta especificación sea una generalización de la solución, independiente de una tecnología en particular, de tal manera que el resultado sea una estandarización que ofrezca soluciones generales.

El concepto de patrones de software tiene como fundamento la definición proporcionada por el catedrático Christopher Alexander a finales de los sesenta [28], la cual indica que:

Definición de patrón 1 *Un patrón es una regla de tres partes que expresan una relación entre un contexto, la descripción de la situación en que el problema ocurre; un problema, la especificación de lo que se quiere resolver; y una solución, que es el cómo resuelve el problema.*

Una segunda definición es la siguiente [29]:

Definición de patrón 2 *Un patrón es la abstracción de una forma concreta que se sigue recurriendo en determinados contextos no arbitrarios.*

Una tercera definición es la siguiente [30]:

Definición de patrón 3 *Un patrón describe el diseño recurrente de un problema particular que surge en contextos específicos de diseño y presenta una bien probada solución para el problema. La solución es especificada por la descripción de sus participantes, sus responsabilidades y relaciones, y la manera en el que colaboran .*

En las tres definiciones citadas, es claro observar que todas giran alrededor de tres conceptos: un *problema*, un *contexto* y una *solución*.

Como herramienta en el desarrollo de software, el uso de patrones de software se puede considerar como una conexión entre un problema en un cierto contexto y una solución general al mismo. Sin embargo, aunque el patrón es útil para establecer una solución a partir de un problema, éste no proporciona a detalle el diseño completo de un sistema.

2.4.2. Cómo se describe un patrón de software

Un patrón se describe a través de una plantilla o formulario. Una primera plantilla para describir un patrón la proporciona C. Alexander [28], la cual establece las bases para las variantes creadas específicamente para el desarrollo de software. Entre las más populares se encuentra la forma *GoF* [31] y la forma *POSA* [4].

Aunque las formas difieren en la cantidad de elementos que describen un patrón, en esencia ambas tienen como objetivo: *(1)* describir el problema que se pretende resolver; *(2)* contextualizar el problema; *(3)* describir la solución y su forma de implementarla, y finalmente *(4)* describir los beneficios y desventajas del patrón. Esta tesis utiliza la forma POSA [4], que se describe a continuación:

- **Nombre** El nombre y un resumen corto del patrón.
- **También conocido como** Otro nombre para el patrón, si hay alguno conocido.
- **Ejemplo** Un ejemplo del mundo real que demuestra la existencia del problema y la necesidad del patrón.
- **Contexto** La situación en la que se aplica el patrón.
- **Problema** El problema que resuelve el patrón incluyendo una discusión de las fuerzas asociadas.
- **Solución** El principio fundamental de la solución que sirve como base al patrón.
- **Estructura** Una especificación detallada de los aspectos estructurales del patrón.
- **Dinámica** Escenarios típicos que describen el comportamiento en tiempo de ejecición del patrón. Los escenarios se ilustran con diagramas de secuencia entre objetos.
- **Implementación** Guía para la implementación del patrón. Son solo sugerencias, no reglas inmutables.

- **Ejemplo resuelto** Discusión sobre cualquier aspecto importante para resolver el ejemplo que no se haya cubierto en las secciones de solución, estructura, dinámica e implementación.
- **Variantes** Una breve descripción de las variantes o especializaciones del patrón.
- **Usos conocidos** Ejemplos de usos del patrón tomados de sistemas existentes.
- **Consecuencias** Los beneficios que provee el patrón, y cualquier desventaja potencial.
- **Véase también** Referencias a patrones que resuelven problemas similares, y a patrones que ayudan a refinar el patrón que se está describiendo.

Aunado a este formulario, se utilizan notaciones gráficas como UML: diagramas de clase, diagramas de objetos y diagramas de secuencia.

2.4.3. Categorización de los patrones de software

Los patrones de software se pueden clasificar con respecto a su nivel de *abstracción* [4] y respecto a su *propósito* [31]. Aunque estas dos clasificaciones no son las únicas, sí son las más conocidas, ya que fueron las primeras que se crearon en el diseño de patrones.

La clasificación respecto al nivel de abstracción es el siguiente:

1. **Patrones Arquitectónicos.** Expresa un esquema estructural fundamental de la organización del sistema. Provee también un conjunto predefinido de subsistemas, especifica sus responsabilidades, e incluye las reglas y directrices para organizar las relaciones entre ellos [4].
2. **Patrones de Diseño.** Provee un esquema para refinar los componentes de un sistema, la relación y comunicación entre ellos [31].
3. **Idiomas.** Es un patrón de bajo nivel que describe cómo resolver una implementación de un problema específico en un determinado lenguaje de programación [4].

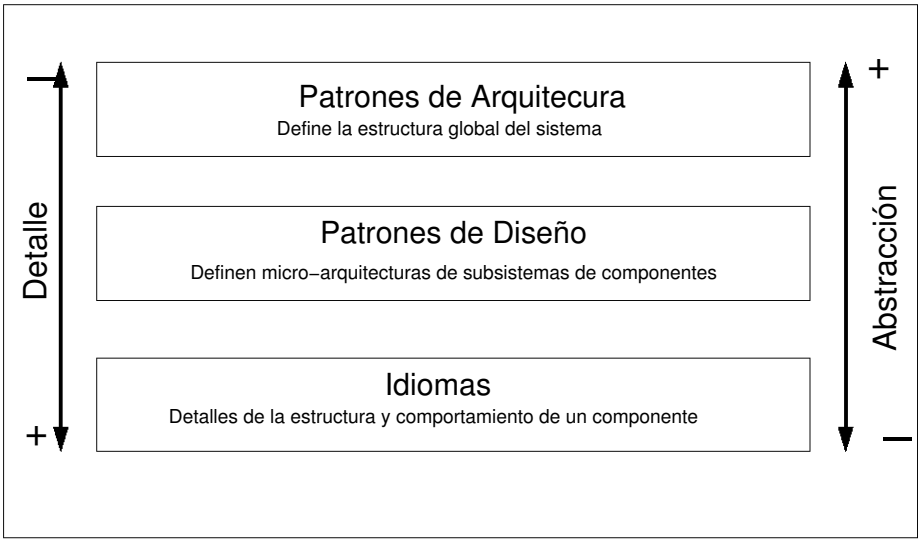

Figura 2.7: Clasificación de los patrones de software según el nivel de abstracción.

Los patrones de software tienen interdependencia, es decir, no existen en un aislamiento total. Una colección de patrones para arquitectura de software, junto con las directrices para su implementación se conoce como un **sistema de patrones** [4].

Un sistema de patrones presenta las siguientes características [32]:

- Provee una base de patrones suficiente para direccionar-resolver problemas de diseño en el dominio donde son aplicados.
- Describe todos sus los patrones uniformemente.
- Expone todas las relaciones sobre todos sus patrones.
- Provee un esquema organizacional para todos sus patrones.
- Apoya el desarrollo de sistemas de software con un conjunto de instrucciones acerca de cómo son implementados y combinados todos sus patrones.
- Apoya su propio crecimiento, permitiendo la integración de nuevos patrones y consecuentemente la adaptación del sistema de patrones a las nuevas tecnologías.

Un sistema de patrones no ofrece todas las soluciones a los problemas de diseño en un dominio particular. Solo describe ciertos aspectos de la construcción de sistemas de software, por lo que es responsabilidad del diseñador completar la arquitectura del software a desarrollar.

2.5. Resumen

En este capítulo se presenta una introducción a la forma en que se comunican procesos que trabajan de forma conjunta con un objetivo en común. Se mencionan las propiedades necesarias para lograr una comunicación: sincronización y coordinación, y los mecanismos creados para lograrlos: semáforos, regiones críticas y monitores. Dependiendo de las necesidades del problema a resolver, existen dos principales formas de establecer una comunicación: síncrona y asíncrona.

Se presenta una introducción a la tolerancia a fallas en sistemas mediante la clasificación de las fallas respecto a su naturaleza y duración. Las clasificaciones permiten indicar que una falla puede ocurrir por un problema en el hardware o en el software, debido a un error humano (accidental o no) y hasta por fenómenos naturales.

Se mencionan las características que debe tener un sistema considerado tolerante a fallas, es decir, un sistema que es capaz de detectar, enmascarar y corregir fallas. Para lograr tal objetivo, el estudio de la tolerancia a fallas se divide en cuatro áreas: la detección de errores, la recuperación del sistema ante la presencia de fallas, el diagnóstico de fallas y la auto-reparación.

También se menciona una introducción a la programación paralela, describiendo el objetivo de paralelizar la solución de un problema, los pasos necesarios para pasar de un programa secuencial a un programa paralelo, y los niveles de paralelismo inherentes al problema a solucionar.

Y finalmente se hace una introducción a los patrones de software: su definición, la forma en que son descritos, su clasificación respecto a su nivel de abstracción y la introducción al concepto de sistema de patrones.

Capítulo 3

Trabajo Relacionado

Es mucho más fácil detectar el error que descubrir la verdad;
el primero se halla en la superficie y no cuesta demasiado dar con él;
la segunda reposa en las profundidades y explorarla no está al alcance de cualquiera.
Johann Wolfgang von Goethe

Este capítulo presenta los trabajos relacionados al desarrollado en esta tesis, los acercamientos en cuanto a *(1)* patrones de software para tolerancia a fallas y *(2)* patrones de diseño para componentes de comunicación de programas paralelos.

3.1. Un sistema de patrones de software para tolerancia a fallas

Un sistema que se considere tolerante a fallas debe cubrir por lo menos los siguientes aspectos: la detección de fallas, el enmascaramiento de fallas y la recuperación ante ellas. El sistema de patrones propuesto en [32] cubre estos tres aspectos para fallas del tipo crash y fallas bizantinas. La clasificación propuesta se muestra en el cuadro 3.1.

Este sistema de patrones aporta un marco de trabajo para realizar un diseño de sistemas con tolerancia a fallas, implementando uno o varios de los patrones propuestos. Por ejemplo, si el contexto del problema requiere realizar el diseño de un sistema que trate con fallas bizantinas, es posible combinar el patrón **Fail-Stop Procesor** para detectar una falla, el **Pasive Replication** para enmascararla y finalmente el **Rollback** para la recuperación del sistema [32].

En las siguientes secciones se presenta un breve resumen de cada uno de los patrones que conforman el sistema de patrones.

<table>
<tr><th>Patrón</th><th>Tipo de Falla</th><th>Aspecto de la tolerancia a fallas</th></tr>
<tr><td>Fail-Stop Procesor
(Procesador falla-parada)</td><td>bizantina</td><td rowspan="4">Error Detection</td></tr>
<tr><td>Acknowledgment
(Reconocimiento)</td><td rowspan="3">crash</td></tr>
<tr><td>I Am Alive
(Estoy vivo)</td></tr>
<tr><td>Are you Alive
(¿Estas vivo?)</td></tr>
<tr><td>Roll Forward
(Puesta al día)</td><td rowspan="2">crash</td><td rowspan="2">Error Recovery</td></tr>
<tr><td>Rollback
(Retroceso)</td></tr>
<tr><td>Pasive Replication
(Replicación pasiva)</td><td rowspan="4">crash

bizantina</td><td rowspan="4">Error Masking</td></tr>
<tr><td>Semi-Pasive Replication
(Replicación semi-pasiva)</td></tr>
<tr><td>Semi-Active Replication
(Replicación semi-activa)</td></tr>
<tr><td>Active-Replication
(Replicación activa)</td></tr>
</table>

Tabla 3.1: Clasificación del sistema de patrones para tolerancia a fallas.

3.1.1. Fail-Stop Processor (Procesador falla-parada)

Un sistema que exhibe fallas bizantinas puede provocar errores de salida, ninguna salida o incluso duplicar la salida (correcta o errónea), entre otros posibles errores. Tratar con fallas bizantinas es costoso, por lo que es preferible transformar esas fallas en otras más tratables o amigables [32].

El contexto.

El patrón aplica en sistemas que tienen las siguientes características [32]:

- El sistema es determinístico.
- Los errores que el sistema puede experimentar son transitorios.
- Los errores no son debido a errores de *entradas*.
- Los errores que puede experimentar son debido a fallas bizantinas.

El problema.

El patrón resuelve el problema de transformar fallas bizantinas a fallas fail-stop, balanceando las siguientes fuerzas [32]:

- El error se confina dentro del sistema fallado y no infecta su entorno.
- El error es detectado por el entorno.
- El tiempo de sobrecarga en la ejecución de un sistema libre de error llega a ser bajo.

La solución.

Se basa en la replicación del sistema y en la comparación de las salidas de las replicas. Hay que tener en cuenta que si existe un error en la entrada de las réplicas, el error se propaga de igual manera en todas las réplicas. La complejidad del diseño de este patrón es baja, debido a que solo se necesita un distribuidor y un comparador. En la figura 3.1 se muestra la estructura (a) y el diagrama de actividad (b) del patrón Fail-Stop Processor (Procesador falla-parada) [32].

3.1.2. Acknowledgment (Reconocimiento)

Una manera de detectar fallas del tipo *crash* es tener un subsistema que notifique al remitente, en un intervalo de tiempo, la recepción de sus entradas [32].

El contexto.

El patrón aplica en sistemas que tienen las siguientes características [32]:

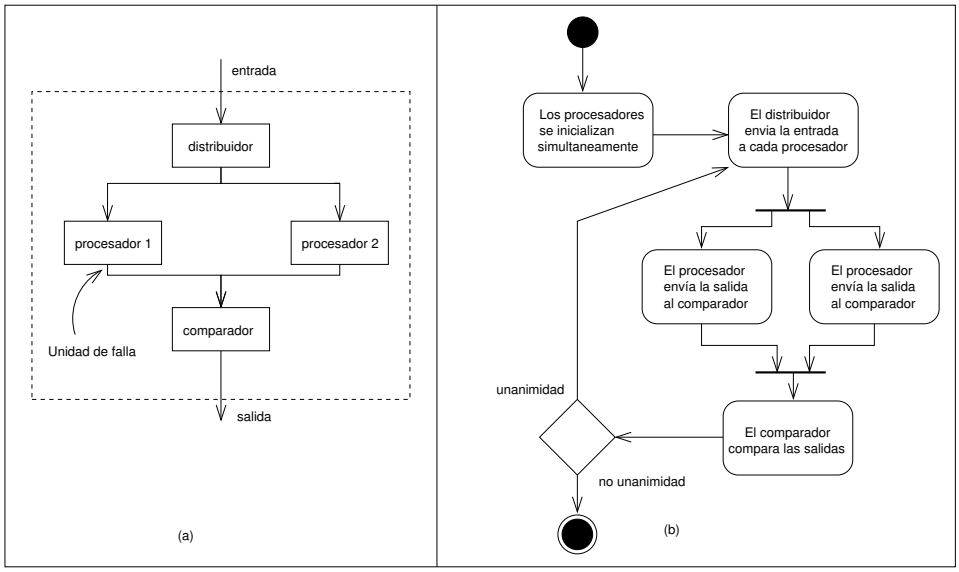

Figura 3.1: Estructura y diagrama de actividad del patrón Fail-Stop Processor

- Los errores que puede experimentar el sistema supervisado son debido a fallas del tipo *crash* o de omisión.
- La frecuencia de interacciones entre el sistema supervisado y el supervisor puede llegar a ser mucha.
- El tiempo que le toma al sistema supervisor contactar al sistema supervisado está delimitado y es conocido.

El problema.
El patrón resuelve el problema de detectar errores en un sistema, balanceando las siguientes fuerzas [32]:

- El tiempo de sobrecarga del mecanismo de detección debe ser mínimo.
- La información que el sistema supervisado ha fallado sólo es de importancia cuando ese sistema está en uso (es decir ha recibido alguna entrada y se espera que produzca la salida correspondiente).
- La comunicación entre el sistema supervisado y el supervisor no debe incrementarse innecesariamente.

La solución.
La solución está basada en el *conocimiento* de la recepción de una entrada del sistema supervisado por parte del sistema supervisor. Se necesita que el sistema supervisor tenga

un tiempo predefinido de espera del sistema supervisado, ya que si lo sobrepasa ese tiempo, se considera como un error. En la figura 3.2 se muestra la estructura (a) y el diagrama de actividad (b) del patrón Acknowledgment (reconocimiento) [32].

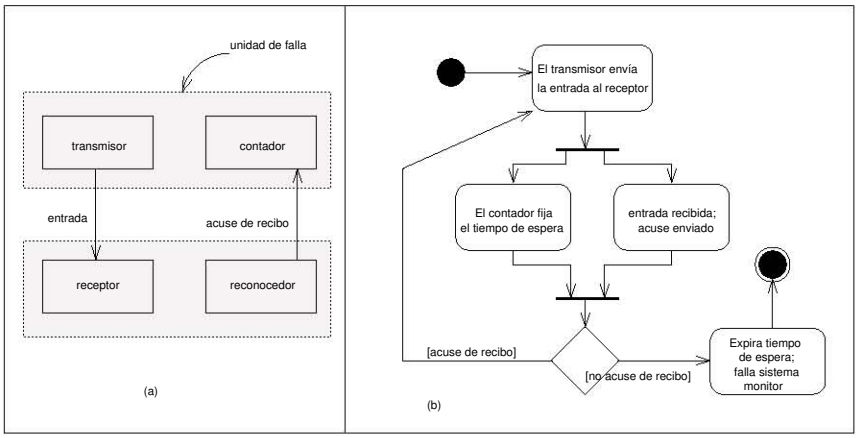

Figura 3.2: Estructura y diagrama de actividad del patrón Acknowledgment

3.1.3. I Am Alive (Estoy vivo)

Una variante para detectar fallas del tipo crash es recibir notificaciones en intervalos de tiempo por parte del sistema supervisado de que está vivo. La recepción de estas señales dentro del tiempo de espera, indican que el sistema supervisado se encuentra en buen estado [32].

El contexto.

El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que puede experimentar el sistema supervisado pueden ser debido a fallas del tipo crash.
- La frecuencia de interacciones entre el sistema supervisado y el supervisor puede llegar a ser mucha.
- La frecuencia de las comunicaciones entre el sistema supervisor y el supervisado está claramente delimitado por el límite de saturación de la red.
- El intervalo de tiempo entre las sucesivas salidas del sistema supervisado no está delimitado o no se conoce.
- El tiempo que toma el sistema supervisado en contactar al sistema supervisor está delimitado y es conocido.

El problema.
El patrón resuelve el problema de detectar errores en un sistema balanceando las siguientes fuerzas [32]:

- La detección de un error en el sistema supervisado se debe realizar tan pronto como sea posible, incluso antes que el entorno necesite comunicarse con el sistema supervisado.
- El sistema supervisor debe tener una actualización regular del conocimiento respecto a la ocurrencia de errores en el sistema supervisado.

La solución.
Se basa en una notificación regular del buen estado del sistema supervisado, en orden de minimizar el tiempo necesario para detectar un error en caso de largos periodos ociosos de comunicación. Para su implementación se necesita de un supervisor encargado de reconocer el buen estado del sistema, un cronómetro para medir los intervalos de tiempo y una entidad que envíe las señales de vida. En la figura 3.3 se muestra la estructura (a) y el diagrama de actividad (b) del patrón I Am Alive (Estoy vivo) [32].

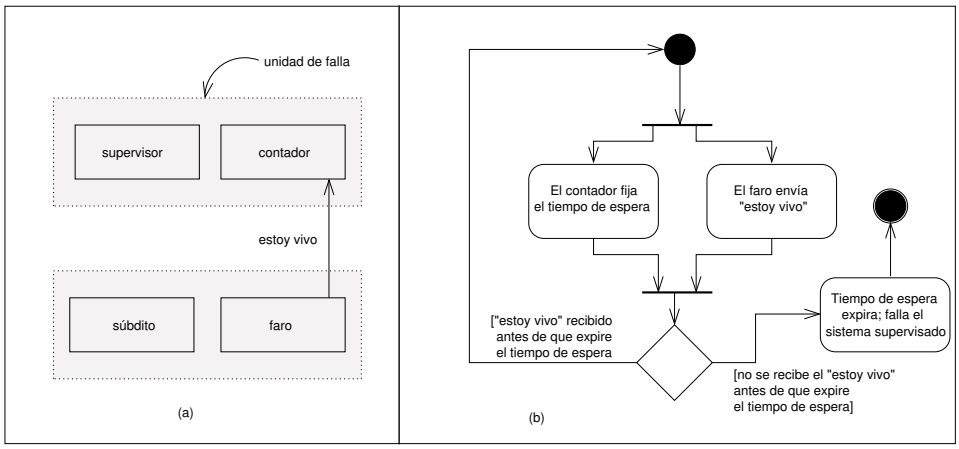

Figura 3.3: Estructura y diagrama de actividad del patrón I Am Alive.

3.1.4. Are You Alive (¿Estas vivo?)

Una variante al patrón I Am Alive, es tener un sistema supervisor que indague el buen estado del sistema supervisado [32].

El contexto.
El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que puede experimentar el sistema supervisado pueden ser debido a fallas del tipo crash.
- La frecuencia de interacciones entre el sistema supervisado y el supervisor puede llegar a ser mucha.
- La frecuencia de las comunicaciones entre el sistema supervisor y el supervisado está claramente delimitado por el límite de saturación de la red.
- El intervalo de tiempo entre las sucesivas salidas del sistema supervisado no está delimitado o no se conoce.
- El tiempo que toma el sistema supervisado en contactar al sistema supervisor está delimitado y es conocido.

El problema.
El patrón resuelve el problema de detectar errores en un sistema balanceando las siguientes fuerzas [32]:

- Los errores ocurridos en el sistema supervisado deben ser detectados en el periodo más corto para conveniencia del sistema supervisor.
- La sobrecarga de comunicación introducida debe ser relativamente baja.

La solución.
Se basa en un sistema supervisor que espera N periodos consecutivos de tiempo para enviar una señal *Are you alive*. Solo si en el n-ésimo periodo el tiempo expira sin recibir respuesta a la señal, el sistema supervisor detecta un error. En la figura 3.4 se muestra la estructura (a) y el diagrama de actividad (b) del patrón Are You Alive (¿Estas vivo?) [32].

3.1.5. Roll Forward (Puesta al día)

La recuperación de un sistema ante una falla consiste en restablecer el último estado correcto que se conoce. Una manera de lograr la recuperación es implementar un sistema tolerante a fallas consistente en dos réplicas. Si la primera réplica produce una salida exitosa, la segunda es puesta al día con el nuevo estado correcto, en caso contrario, la réplica es descartada y el estado correcto es tomado de la segunda réplica [32].
El contexto
El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que el sistema puede experimentar son *detectables*.

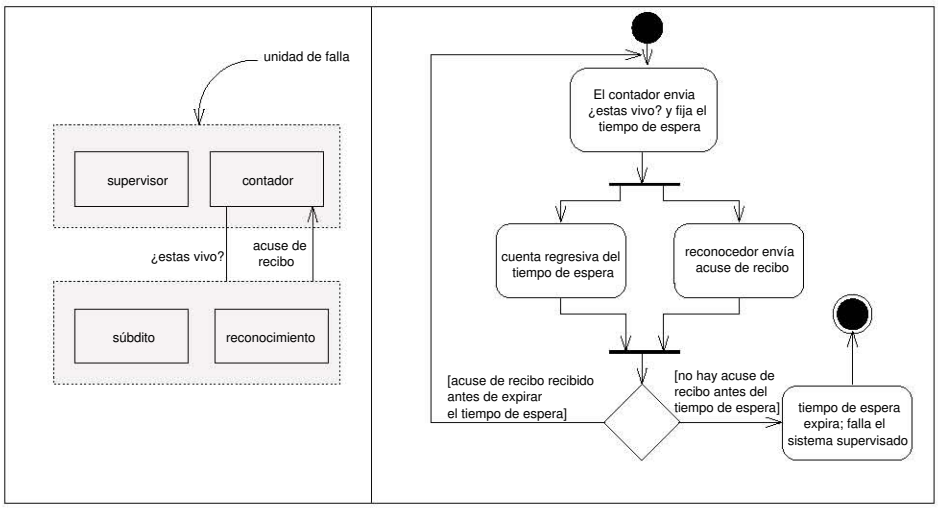

Figura 3.4: Estructura y diagrama de actividad del patrón Are You Alive.

- Los errores que el sistema puede experimentar no se deben a *errores en la entrada*.
- Las ejecuciones sin errores del sistema están claramente por debajo de todas las limitaciones de tiempo impuestas sobre ellas.
- El sistema es capaz de exportar su estado actual e importar un nuevo estado.

El problema

Sobre este contexto, el patrón Roll Forward (Puesta al día) resuelve el problema de recuperación de un error en un sistema balanceando las siguientes fuerzas [32]:

- El tiempo para recuperarse de un error se debe mantener mínimo.
- La ejecución libre de errores del sistema no debe violar ninguna de las limitaciones de tiempo impuestas.

La solución

La solución propuesta por el patrón consiste en la implementación de dos réplicas del sistema. Una de ellas procesa las nuevas entradas y si no ocurre un error, exporta el nuevo estado a la otra réplica. En caso contrario, si la réplica encargada de las nuevas entradas experimenta un error, se descarta e importa el último estado correcto de la réplica 2 [32].

3.1.6. Rollback (Retroceso)

El patrón Rollback (Retroceso) presenta otra manera de recuperarse de un error usando réplicas. Una réplica recibe y procesa las entradas y en cierto momento puede generar un

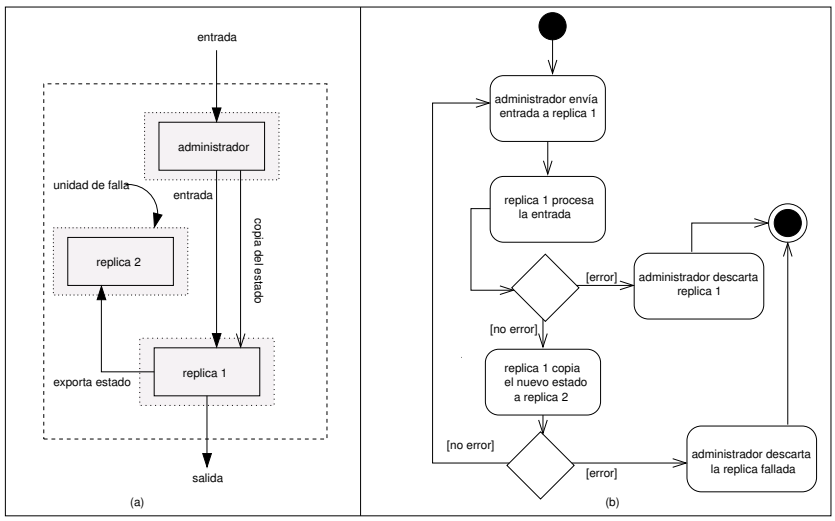

Figura 3.5: Estructura y diagrama de actividad del patrón Roll Forward.

punto de control. Cuando ocurre un error, la segunda réplica usa el punto de control para restaurar algún estado libre de errores [32].

El contexto

El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que el sistema puede experimentar son *detectables*.
- Los errores que el sistema puede experimentar no se deben a *errores en la entrada*.
- El sistema es capaz de exportar su estado actual e importar un nuevo estado.

El problema

Sobre este contexto, el patrón Rollback (Retroceso) resuelve el problema de recuperación de un error en un sistema balanceando las siguientes fuerzas [32]:

- El tiempo de sobrecarga de una ejecución libre de errores debe ser mantenido mínimo.
- El estado libre de errores restaurado después de un error es lo más cerca posible al último estado libre de errores que había llegado la réplica antes de experimentar el error.

La solución

La solución propuesta es establecer dos réplicas encargadas de establecer puntos de control. Si una réplica falla, es descartada y la segunda réplica es la encargada de exportar el estado libre de errores a partir del punto de control [32].

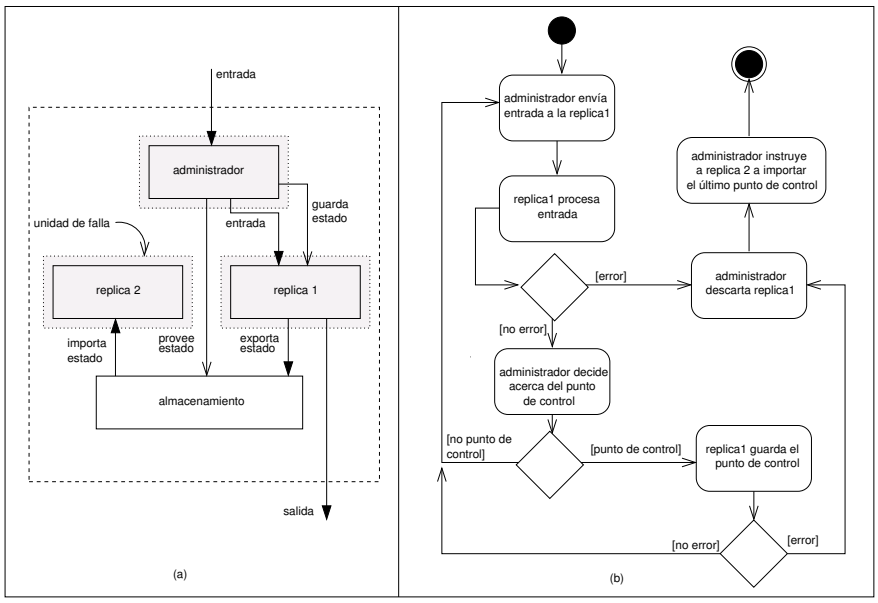

Figura 3.6: Estructura y diagrama de actividad del patrón Rollback.

3.1.7. Passive Replication (Replicación Pasiva)

Los patrones Roll Forward (Puesta al día) y Rollback (Retroceso) proveen soluciones para recuperarse de un error en el sistema. Sin embargo, la información procesada hasta antes del momento de la falla se pierde. En muchos casos, esta pérdida de información no es conveniente. Es por ello que se necesita una técnica de enmascaramiento de errores [32].

El contexto

El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que el sistema puede experimentar son *detectables*.
- Los errores que el sistema puede experimentar no se deben a *errores en la entrada*.
- El sistema es capaz de exportar su estado actual e importar un nuevo estado.

El problema

Sobre este contexto, el patrón Passive Replication (Replicación Pasiva) resuelve el problema de enmascarar un error en el sistema balanceando las siguientes fuerzas [32]:

- La entrada recibida por el sistema debe ser procesada y la salida designada debe ser entregada independientemente si un error ocurre en el sistema.
- El tiempo de sobrecarga de un sistema libre de errores se debe mantener mínimo.

- Si un error ocurre en el sistema, el entorno es capaz de tolerar una salida retardada del sistema que toma considerablemente más que el retardo usual en una ejecución libre de error.

La solución

La solución está basada en la sugerida por el patrón Rollback (Retroceso). Se crean dos réplicas, una que recibe y procesa la entrada (llamada *primario*), y otra réplica (llamada *respaldo*) que permanece inactiva bajo una ejecución libre de errores del sistema. Cada nueva entrada enviada a *primario* es anotada en el *registro*. El administrador decide cuándo *primario* debe crear un punto de control en *almacenamiento* (antes de registrar la entrada, es enviada a *primario*). El *registro* envía a *primario* la entrada y este último la procesa. Si un error ocurre en *primario* mientras se procesa la entrada, el *respaldo* es activado, importa el último punto de control, recibe del registro la última entrada y comienza su procesamiento, en sustitución de *primario* que ha fallado [32].

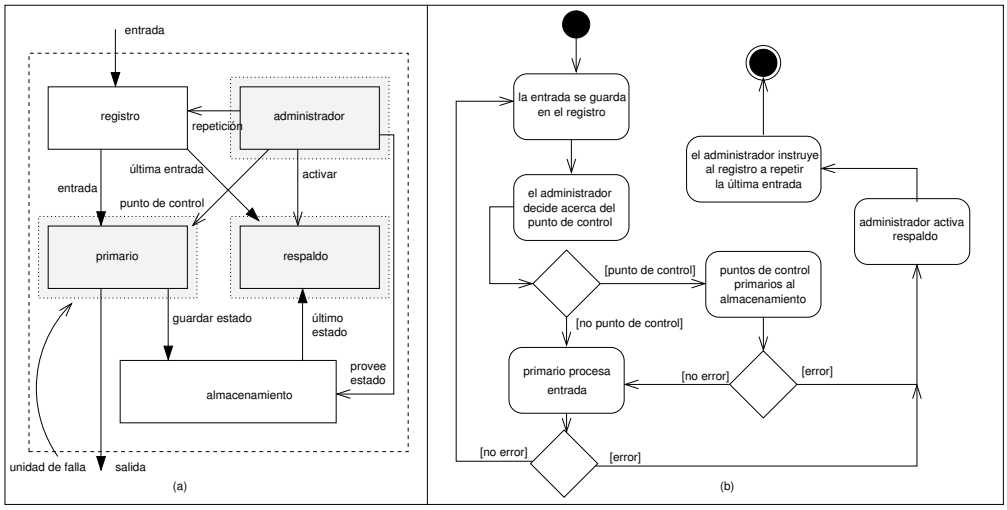

Figura 3.7: Estructura y diagrama de actividad del patrón Passive Replication.

3.1.8. Semi-Passive Replication (Replicación Semi-Pasiva)

Otra manera de enmascarar errores sin tener que pagar retrasos de tiempo cuando ocurre un error, es seguir un esquema similar al del patrón Passive Replication (Replicación Pasiva), a diferencia de que este patrón exporta el estado directamente a la entidad de respaldo [32].

El contexto

El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que el sistema puede experimentar son *detectables*.
- Los errores que el sistema puede experimentar no se deben a *errores en la entrada*.
- El sistema es capaz de exportar su estado actual e importar un nuevo estado.

El problema

Sobre este contexto, el patrón Semi-Passive Replication (Replicación Semi-Pasiva) resuelve el problema de enmascarar un error en el sistema balanceando las siguientes fuerzas [32]:

- La entrada recibida por el sistema debe ser procesada y la salida designada debe ser entregada independientemente si un error ocurre en el sistema.
- La ejecución libre de errores del sistema debe sufrir sanciones de tiempo mínimo.
- El retraso que introduce la solución en la presencia de errores debe mantenerse baja.

La solución

La solución es similar a la sugerida en el patrón Passive Replication (Replicación Pasiva) con la diferencia de que no se necesita una entidad de almacenamiento. Se crean dos réplicas, una que recibe y procesa la entrada (llamada *primario*) y otra (llamada *respaldo*) que importa el estado de *primario* cada vez que éste realiza un punto de control. El resto del proceso es similar al patrón Passive Replication (Replicación Pasiva) [32].

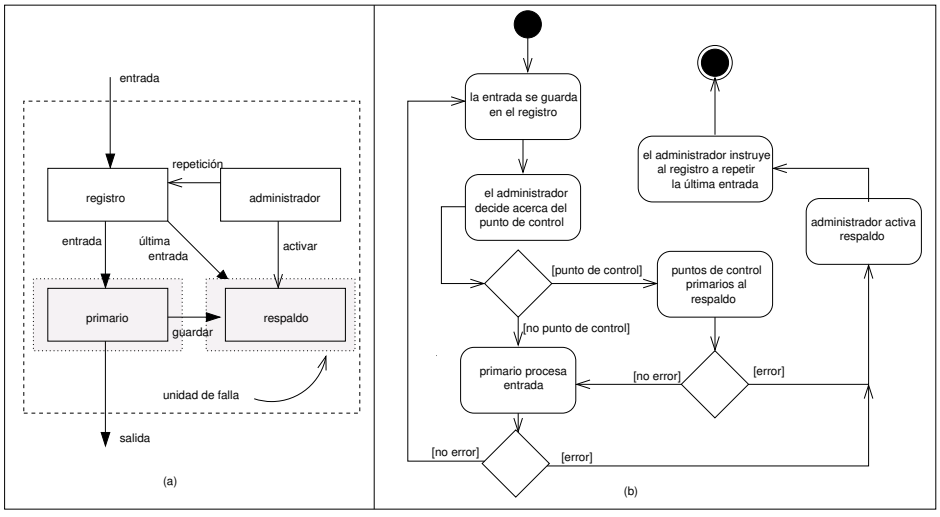

Figura 3.8: Estructura y diagrama de actividad del patrón Semi Passive Replication.

3.1.9. Semi-Active Replication (Replicación Semi-Activa)

Para evitar la complejidad relacionada con la exportación e importación del estado de un sistema, el patrón Semi-Active Replication (Replicación Semi-Activa) proporciona una forma alternativa para enmascarar errores. En lugar de tener sólo una réplica de forma activa y el resto de forma pasiva esperando un error, en este patrón todas las réplicas están activas procesando las entradas de forma paralela. Si se produce un error en una de las réplicas, los demás son capaces de entregar sin demora la salida designada [32].

El contexto

El patrón aplica en sistemas que tienen las siguientes características [32]:

- Los errores que el sistema puede experimentar son *detectables*.
- Los errores que el sistema puede experimentar el sistema no se deben a *errores en la entrada*.

El problema

Sobre este contexto, el patrón Semi-Active Replication (Replicación Semi-Activa) resuelve el problema de enmascarar un error en el sistema balanceando las siguientes fuerzas [32]:

- La entrada recibida por el sistema debe ser procesada y la salida designada debe ser entregada independientemente si un error ocurre en el sistema.
- La ejecución libre de errores del sistema debe sufrir retrasos de tiempo mínimo.
- El retraso que introduce la solución en la presencia de errores debe mantenerse baja.

La solución

La solución se basa en un grupo de dos réplicas idénticas del sistema que procesan en paralelo las entradas al sistema tolerante a fallas. Una de las réplicas (llamada *coordinador*) es el líder del grupo y la otra réplica (llamada *seguidor*) que supervisa al coordinador de errores. En una ejecución libre de errores, el *coordinador* proporciona la salida al entorno. En este caso el *seguidor* mantiene la salida que se ha producido hasta que esté seguro que el *coordinador* ha emitido una salida al entorno y luego la descarta. Si se detecta un error en el *coordinador*, entonces el *seguidor* toma la responsabilidad de emitir una salida al entorno [32].

3.1.10. Active Replication (Replicación Activa)

La técnica más poderosa para el enmascaramiento de errores es la sugerida por el patrón Active Replication (Replicación Activa). A diferencia del patrón Semi-Active Replication

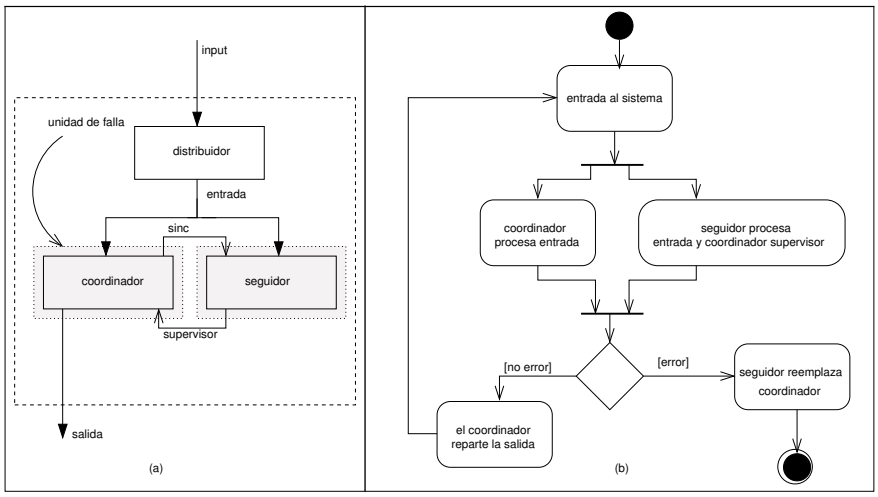

Figura 3.9: Estructura y diagrama de actividad del patrón Semi Active Replication.

(Replicación Semi-Activa), en este patrón todas las réplicas entregan su salida sin supervisarse las unas con las otras, o mantener su salida hasta que un miembro del grupo entregue su salida al entorno [32].

El contexto

El patrón se aplica en sistemas que tienen las siguientes características [32]:

- El sistema es determinista, es decir, su producción se define sólo por su estado inicial y la secuencia de entradas que se han procesado al momento actual.
- Los errores que el sistema puede experimentar no se deben a *errores en la entrada*.
- Los errores que el sistema puede experimentar son causados a fallas bizantinas.

El problema

Sobre este contexto, el patrón Active Replication (Replicación Activa) resuelve el problema de enmascarar un error en el sistema balanceando las siguientes fuerzas [32]:

- La entrada recibida por el sistema debe ser procesada y la salida designada debe ser entregada independientemente si un error ocurre en el sistema.
- La ejecución libre de errores del sistema debe sufrir retrasos de tiempo mínimo.
- El retraso que introduce la solución en la presencia de errores debe mantenerse baja.
- El sistema supervisor es determinístico.

La solución

La solución sugerida por este patrón está basada en la creación de un grupo de réplicas.

De manera similar al patrón Semi-Active Replication (Replicación Semi-Activa), todas las réplicas reciben la misma entrada (el mismo contenido en el mismo orden), pero en este caso todos los miembros del grupo entregan su salida a un *comparador*. Este último es una extensión del utilizado en el patrón Fail-Stop Processor (Procesador falla-parada), que es el responsable de decidir (por mayoría de votos) cúal es la salida correcta.

En una ejecución libre de errores, el comparador debe recibir idénticas salidas de todas las replicas. Si un error ocurre en una de las replicas, la salida que la réplica produce es diferente de las otras que no tuvieron el mismo error. Usando un sistema de tres réplicas, el *comparador* es capaz de detectar cúal es la salida errónea [32].

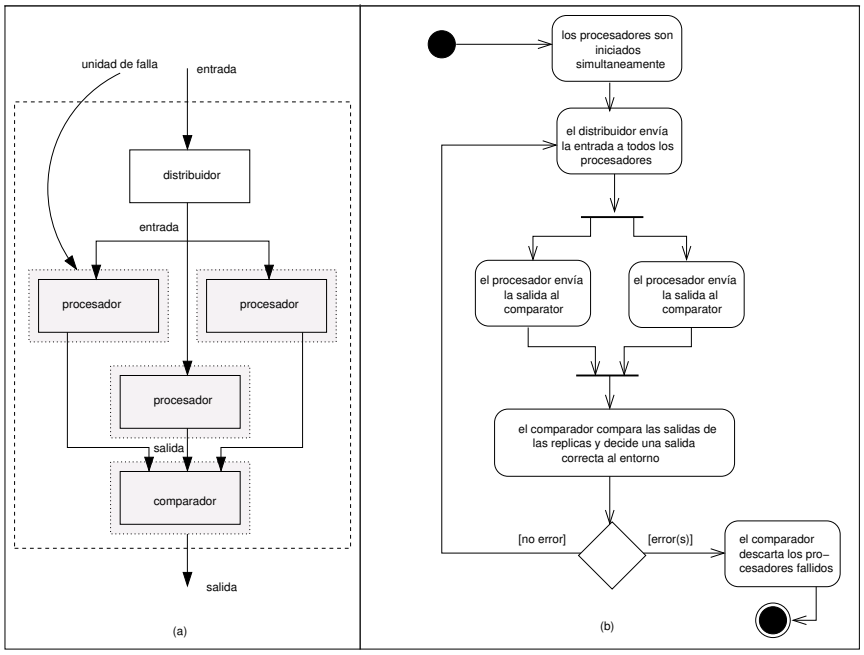

Figura 3.10: Estructura y diagrama de actividad del patrón Active Replication.

3.2. Patrones de diseño para componentes de comunicación de programas paralelos

La comunicación entre componentes de procesamiento es parte fundamental de cualquier procesamiento paralelo. En un programa paralelo, los datos deben ser transferidos entre tareas para poder realizar la computación. Este flujo de información es especificado en la fase del diseño de las comunicaciones [33]. Un conjunto de patrones de diseño para componentes de comunicación de programas paralelos es presentado en [1].

Este conjunto de patrones de diseño están clasificados acuerdo a *(1)* el tipo de paralelismo: funcional, de dominio o de actividad; *(2)* a la organización de memoria de la plataforma de hardware y *(3)* al tipo de sincronización. En la tabla 3.2, se presenta la clasificación de los patrones propuestos en [1].

	Paralelismo	**Organizacion de la memoria**	**Sincronización**
Shared Variable Pipe (Tubería de Variable Compartida)	Funcional	Memoria Compartida	Asíncrona
Multiple Local Call (Múltiple Llamada Local)			Síncrona
Message Passing Pipe (Tubería de Paso de Mensajes)		Memoria Distribuida	Asíncrona
Multiple Remote Call (Múltiple Llamada Remota)			Síncrona
Shared Variable Channel (Canal de Variable Compartida)	Dominio	Memoria Compartida	Asíncrona
Message Passing Channel (Canal de Paso de Mensajes)		Memoria Distribuida	
Local Rendezvous (Cita Local)	Actividad	Memoria Compartida	Síncrona
Remote Rendezvous (Cita Remota)		Memoria Distribuida	

Tabla 3.2: Clasificación de los patrones de diseño para componentes de comunicación.

3.2.1. Message Passing Pipe (Tubería de Paso de Mensajes)

Este patrón *Message Passing Pipe (Tubería de Paso de Mensajes)* describe el diseño de un componente *pipe* basado en paso de mensajes que implementa operaciones de envío y recepción en un sistema paralelo de memoria distribuida [1].

Contexto

El patrón se aplica cuando se construye un programa paralelo usando el patrón arquitectónico *Parallel Pipes and Filters (Filtros y Tuberías Paralelas)* presentado en [34], el cual es una extensión que incluye aspectos de paralelismo funcional del patrón *Pipes and Filters (Filtros y Tuberías)* descrito en [4].

El patrón *Pipes and Filters (Filtros y Tuberías)* provee una estructura para sistemas que procesan un flujo de datos. Cada paso del proceso se encapsula en componentes autónomos (filtros) y los datos pasan a través de *tuberías* entre filtros adyacentes [4].

El problema

Una colección de filtros paralelos necesitan comunicarse para intercambiar mensajes, siguiendo una sola dirección de flujo de datos, donde cada dato es procesado dentro de un filtro. El patrón resuelve este problema balanceando las siguientes fuerzas [1]:

- Mantener el orden de la transferencia de datos a través de las tuberías usando una política FIFO.
- La comunicación debe ser punto a punto y unidireccional.
- La implementación debe considerar una plataforma de memoria distribuida.
- Los datos deben ser transferidos asíncronamente.

La solución

La estructura del software se compone de *puntos finales* de comunicación (por ejemplo sockets), algún mecanismo de sincronización y un par de secuencias de datos. Los componentes se unen en orden al fin de lograr una comunicación unidireccional entre filtros que se ejecutan en memoria distribuida en diferentes procesadores o computadoras.

En la figuras 3.11 y 3.12 se muestran los diagramas de colaboración y secuencia respectivamente [1].

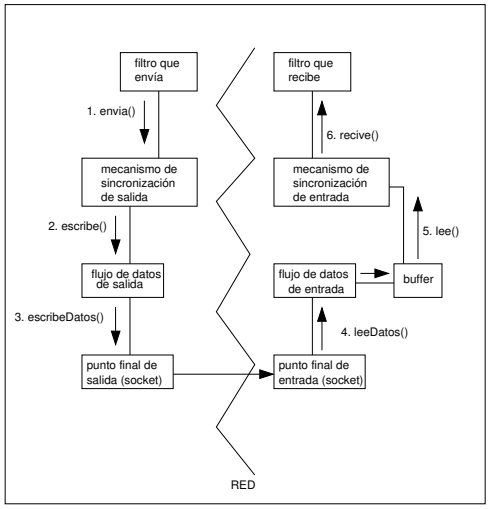

Figura 3.11: Diagrama de colaboración del patrón Message Passing Pipe.

3.2.2. Message Passing Channel (Canal de Paso de Mensajes)

El patrón *Message Passing Channel (Canal de Paso de Mensajes)* describe el diseño de un componente *canal* basado en paso de mensajes, implementando operaciones de *envio* y *recepción* para las comunicaciones del componente *canal* en un sistema paralelo de memoria distribuida [1].

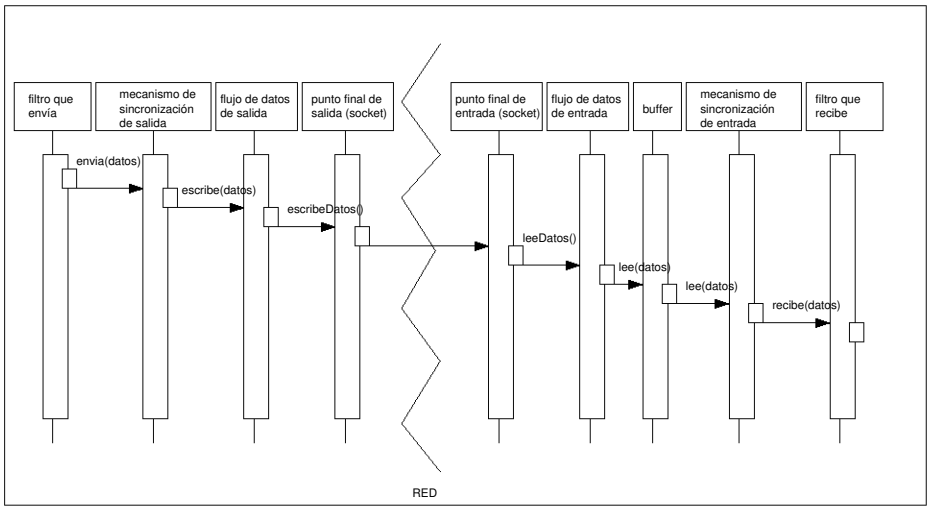

Figura 3.12: Diagrama de secuencia del patrón Message Passing Pipe.

Contexto

El patrón se aplica cuando se construye un programa paralelo usando el patrón arquitectónico *Communicating Sequential Elements (Elementos de Comunicación Secuencial)* [1]. Este es un patrón para paralelismo de dominio en que cada componente está diseñado para ejecutar las mismas operaciones con diferentes piezas regulares de datos. Las operaciones en cada componente dependen del resultado parcial de los componentes vecinos [35].

El problema

Un elemento necesita intercambiar valores con sus elementos vecinos. Todos los datos están guardados en elementos, que son los responsables de procesar los datos. El patrón resuelve este problema balanceando las siguientes fuerzas [1]:

- Mantener el orden en la transferencia de datos a través del canal.
- La comunicación debe ser uno a uno y bidireccional.
- La implementación debe considerar una plataforma de memoria distribuida.
- La transferencia de datos se debe llevar a cabo de forma asíncrona.

La solución

Diseñar un componente de canal como una estructura de software distribuido conectando elementos que se ejecutan en dos procesadores o computadoras diferentes, cuya estructura está compuesta de puntos finales de comunicación (usualmente sockets), algún mecanismo de sincronización y flujos de datos. Los componentes se unen para lograr una comunicación bidireccional, con componentes en memoria distribuida que se ejecutan en diferentes

procesadores o computadoras [1]. En la figuras 3.13 y 3.14 se muestran los diagramas de colaboración y secuencia, respectivamente [1].

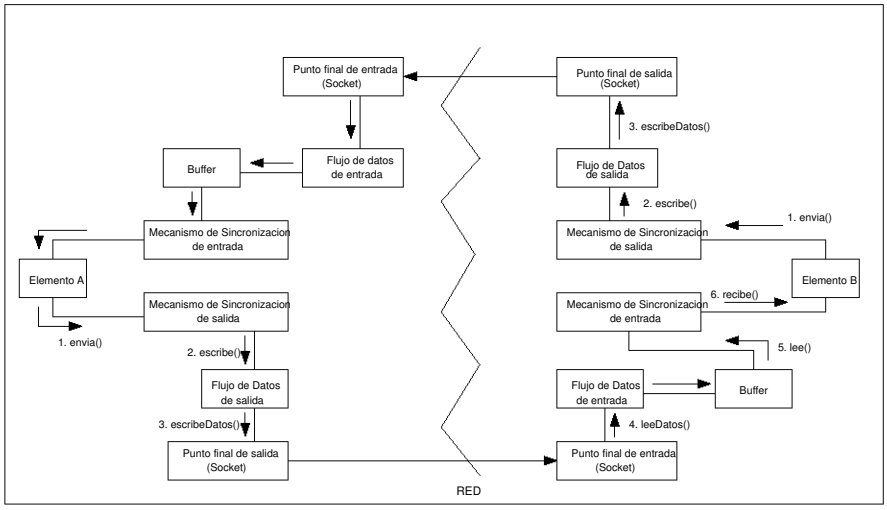

Figura 3.13: Diagrama de colaboración del patrón Message Passing Chanel

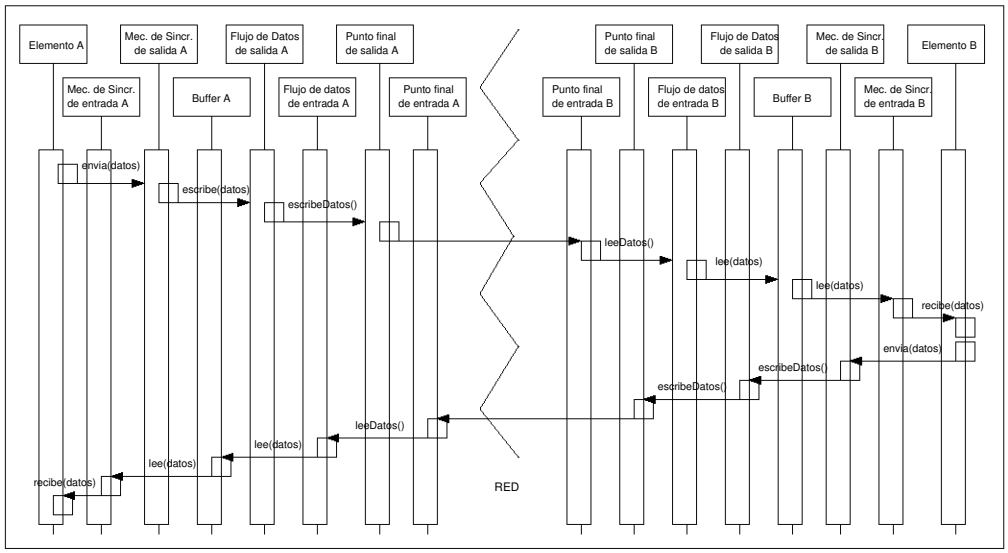

Figura 3.14: Diagrama de seuencia del patrón Message Passing Chanel

3.3. Resumen

En este capítulo se muestran dos catálogos de patrones de diseño. El primero sugiere técnicas para la tolerancia a fallas como: *(1)* redundancia, a través de la replicación de módulos del sistema; *(2)* puntos de control, que almacenan un estado correcto previo a un proceso consecuente que puede fallar y *(3)* módulos supervisor-supervisado, para conocer el estado actual tanto del módulo que emite el mensaje como del que lo recibe. Con estos tres mecanismos, el sistema de patrones propuesto en [32] ofrece una solución a problemas de detección, recuperación y enmarcaramiento de errores.

El segundo trabajo expuesto, consiste en una propuesta de componentes de comunicación para programas paralelos. Estos patrones de software están diseñados de acuerdo al tipo de paralelismo del problema a resolver y al tipo de memoria del hardware. El objetivo de esta tesis se delimita a arquitecturas de memoria distribuida, por lo que solo se muestran cuatro patrones de diseño. En [1] se puede consultar el trabajo completo.

En el contexto de este trabajo, ninguno de los dos trabajos da solución al problema que se plantea. Por un lado, el primer sistema de patrones describe soluciones para el manejo de errores en sistemas que no realizan procesamiento paralelo, por el otro, se describen soluciones para la comunicación de programas paralelos, pero sin la consideración de que puedan presentarse errores en la misma. Sin embargo, ambos trabajos sirven como base para la propuesta de solución que propone ésta tesis y que se describe a detalle en el siguiente capítulo.

Capítulo 4

Propuesta de patrones de software

El medio es el mensaje
Marshall McLuhan

Este capítulo tiene como objetivo: *(1)* definir una clase de fallas sobre la cual están diseñados los mecanismos de prevención y corrección de errores y *(2)* la descripción de los patrones de diseño que integran los mecanismos propuestos de prevención y corrección de errores.

En los patrones de diseño para componentes de comunicación descritos en la sección 3.2, se da por hecho que la comunicación se logra y no existen errores durante la transmición de datos o en alguno de los componentes que lo integran. En los mecanismos propuestos aquí, el contexto es similar al que da origen a tales patrones, pero se considera el hecho de la presencia de fallas que pueden interrumpir la comunicación correcta del programa paralelo.

Se considera también como referencia el sistema de patrones descrito en la sección 3.1, en los cuales las técnicas propuestas de tolerancia a fallas no están descritas considerando un contexto en donde el procesamiento se realiza de forma paralela.

Finalmente, la descripción de los patrones de software se realiza utilizando la forma POSA [4] [Sección 2.4.2], una de las plantillas de descripción de patrones de software más populares y utilizadas en la literarura de los patrones de software.

4.1. Descripción de la clase de fallas

La presencia de una falla en un componente de comunicación puede deberse a diferentes motivos: a un error en el disco duro, en la memoria, en un nodo de red, e incluso se

pueden considerar errores por fenómenos naturales [Sección 2.2]. La presencia de una falla que interrumpe la comunicación tiene como consecuencia una ejecución incompleta o un resultado incorrecto. En ambos casos, significa el desperdicio de todo el cómputo. Este escenario resulta muy costoso en el caso de un programa paralelo que necesita largos periodos de cómputo para obtener una respuesta [2, 36] .

El alcance de este trabajo se limita a un conjunto de fallas, llamado desde ahora *clase de fallas* [Sección 2.2], sobre las cuales los patrones propuestos ofrecen un mecanismo de prevención o corrección de errores.

Desde un enfoque arquitectónico, todos los componentes que integran cualquier sistema también pueden ser considerados como un sistema por sí mismos [24]. Con base en ello, es posible *ver* como un sistema cada uno de los componentes que permiten la comunicación entre componentes de procesamiento de un programa paralelo, y así, analizarlos de una forma independiente.

Los participantes en un componente de comunicación en general son los siguientes [25]:

- Un mecanismo de sincronización.
- Un buffer.
- Un mecanismo de envío y recepción de datos a través de la red.
- Un mecanismo de serialización y deserialización de datos para su envío a través de la red (en el caso de una arquitectura distribuida).

La dinámica entre estos componentes está definida por el tipo de paralelismo que presenta el problema a resolver y la arquitectura de memoria en la cual se va a implementar la solución.

Las fallas que se describen a continuación indican cuál es el problema observado, así como las consecuencias que tiene cada una de ellas. El objetivo de definir cada una de ellas a detalle es delimitar de forma puntual el contexto en el que se presenta.

4.1.1. Falla por error en la asignación de memoria en el buffer

El objetivo de implementar un buffer, es decir, una estructura de datos con políticas FIFO, dentro de un componente de comunicación, es servir como un repositorio de datos. Un componente de procesamiento deposita datos y continúa con su tarea, mientras otro extrae los mismos conforme a sus necesidades, permitiendo así establecer un intercambio de datos asíncrono [Sección 2.1.2] entre componentes de procesamiento de un programa paralelo.

La presencia de una falla por un error en la asignación de memoria interrumpe el proceso de almacenamiento de datos, provoca una pérdida de datos y una comunicación no acorde a la especificación funcional del programa [Sección 2.2.5].

Esta falla se define de la siguiente manera:

Definición de Falla 1 ***(Falla por error en la asignación de memoria en el buffer.)****: No es posible asignar más memoria al buffer, lo que provoca que no pueda seguir almacenando datos.*

La falla puede presentarse en un escenario como el que se describe a continuación [13]:

"Considere el problema de la contención de un buffer de entrada...cada transacción es colocada en el siguiente espacio libre de la cola... sin embargo es posible que un largo numero de procesadores haga peticiones al mismo módulo al mismo tiempo, comprometiendo excesivamente el buffer...".

Una solución a este problema es incrementar el tamaño del buffer de una forma tal que pueda evitarse este problema. Pero en este escenario el problema es determinar *cuánto* espacio es necesario reservar para evitar la falla de asignación de memoria.

La solución que se propone en [13] es detener la recepción de mensajes, lo que con lleva a un "contrato" entre los nodos de procesamiento y la red: "si la red acepta una transacción, ésta garantiza que llega a su destino; en caso contrario se informa al nodo que intente después".

La estrategia mencionada en el parrafo anterior implica que el componente de procesamiento se detenga a la espera de una señal que le indique que ya es posible almacenar los datos que esta generando. Esto implica la pérdida de simultaneidad en el proceso, escenario que se desea evitar.

En la figura 4.1 se muestra el diagrama de secuencia donde se muestra el problema comentado.

Si la estructura de datos utilizada para implementar un buffer implementa un mecanismo que asegura la vitalidad del programa [Sección 2.2.2], por ejemplo con un bloque try/catch, no es posible asegurar que la comunicación se realiza conforme a su especificación debido a que las nuevas entradas son omitidas hasta que sea posible asignar memoria al buffer.

En [37] se muestra una solución en lenguaje Java al problema, el cual consiste en atrapar el error e inmediatamente utilizar el recolector de basura:

```
1 ...
2 Vector messages = new Vector();
3 int count = 0;
```

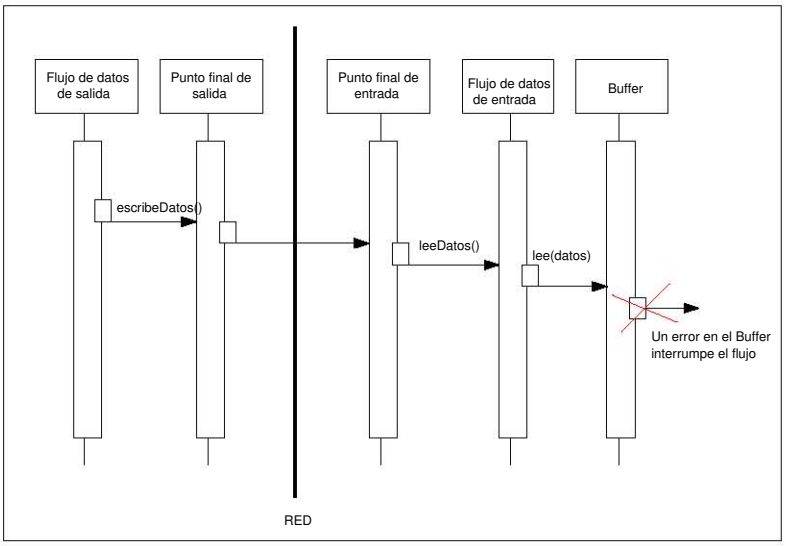

Figura 4.1: Falla en la comunicación por error de asignación de memoria al buffer.

```
4  //increment in megabyte chunks initially
5  int size = 1048576;
6
7  while(size > 1){
8
9       try{
10           for( ; true;count++){
11                v.addElement(new byte[size]);
12           }
13      }catch(OutOfMemoryError bounded) { size = size/2; }
14 }
15 v=null;
16 //and ask for a new Vector as a new small object
17 //to make sure garbage collection kicks in before
18 //we exit the method
19 v = new Vector();
20 ...
```

En el contexto de una aplicación paralela, utilizar esta técnica implica eliminar todos los datos almecenados hasta el momento del error, lo cual resulta catastrófico. Otra solución propuesta en [37] implica la creación de hilos monitores que utilizan variables banderas para prevenir el error. Sin embargo, esta solución puede "no ser factible en un ambiente en donde existen muchos hilos ejecutandose" [37].

Este error no es propio del lenguaje Java. Es un error propio de creación de objetos y recolección de basura (garbage collection). En [38] se encuentra la descripción del error en el caso del lenguaje C++.

4.1.2. Falla por error en el link de comunicación

Un error durante el envío o la recepción de datos a través de la red puede ocurrir por diversos motivos: pérdida o colisiones de paquetes, un retardo en la entrega del paquete, un problema de ruteo, etc. Para esta tesis, se consideran dos fallas independientemente del error que las origina: falla por la pérdida del mensaje o falla por el retardo del mensaje.

La pérdida o el retardo de mensajes, es decir, los datos enviados mediante operaciones de comunicación establecidas de un punto a otro [Sección 2.1.3], provoca que la comunicación se vea comprometida en cuanto a la integridad en los datos, teniendo como consecuencia un resultado incorrecto al final del cómputo o incluso la terminación inesperada del programa paralelo.

La falla se define de la siguiente manera:

Definición de Falla 2 ***(Falla por error en el link de comunicación)****: Un error durante la transmisión del mensaje a través una red provoca pérdida o retardo de mensajes, afectando con ello la correcta comunicación entre componentes de procesamiento.*

En la figura 4.2 se muestra un ejemplo en el que dos componentes de comunicación intercambian información de manera normal, hasta que uno de los mensajes enviados no llega a su destino.

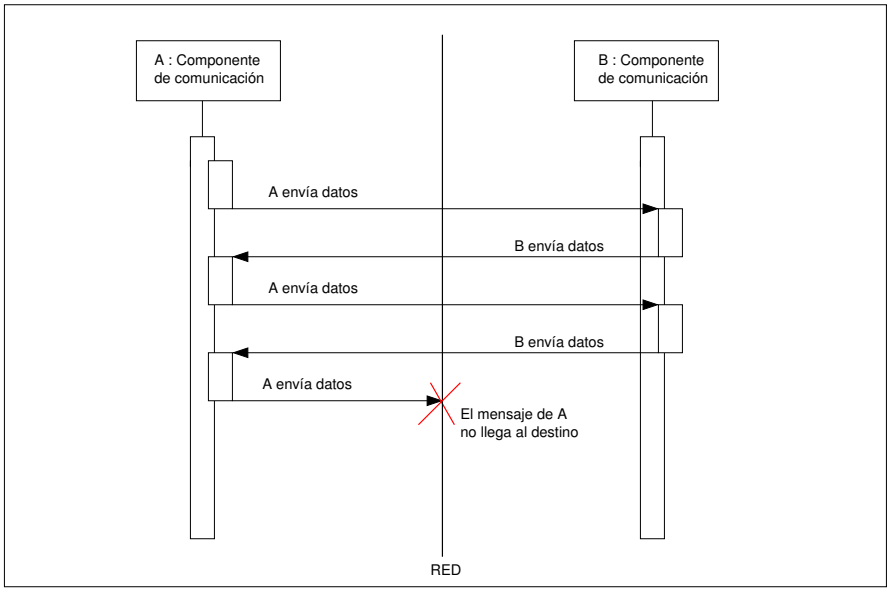

Figura 4.2: Falla por un error en el link de comunicación.

Una caracteristica principal de una comunicación que se llava a cabo mediante paso de mensajes [Sección 2.1.3] es que realiza el envío y recepción de mensaje de una manera

asíncrona [Sección 2.1.2]. Esto implica que el componente de procesamiento del programa paralelo no interrumpe su ejecución si el mensaje no fue enviado o recibido satisfactoriamente en tanto tenga datos que procesar.

Los errores que se presentan durante la transmisión del mensajes se limitan a aquellos que se presentan de manera transitoria o intermitente [Sección 2.2.3] y de una manera limitada, es decir, no todos los mensajes enviados se pierden durante su envío.

En el momento de presentarse un error en el link de comunicación, el componente de comunicación no *sabe* de la gravedad del problema: si se debe a un problema temporal en la red o si ha ocurrido un problema del que no es posible recuperarse. Una manera de poder obtener esa información es establecer un protocolo entre el emisor y el receptor, en el que el primero envía un mensaje con la intención de tener un acuse de recibo por parte del receptor, indicando que el mensaje fue recibido satisfactoriamente.

Existen varios protocolos para obtener información acerca de la entrega satisfactoria de un mensaje. Entre ellos, se encuentra el protocolo **Stenning** o el protocolo del **Bit Alternativo (Alternative Bit)**, ambos descritos en [20]. Sin embargo, en ambos protocolos el mensaje es enviado repetidamente hasta que el receptor envíe una señal de que ha recibido tal mensaje, lo cual genera tráfico extra en la red y la necesidad de analizar todos los mensajes recibidos por parte del receptor, aún cuando ya haya enviado el acuse de recibo.

4.1.3. La clase de fallas.

Con base en las definiciones de fallas presentadas en la secciónes anteriores, en la presente sección se define la clase de fallas para las cuales ésta tesis propone una solución expresada en forma de patrones de diseño. En el cuadro 4.1 se presenta la caracterización de las fallas en base a la clasificación presentada en la sección 2.2.3.

Descripción de la Falla	**Detectabilidad**	**Respecto a su duración**	**Respecto a su tipo**
Falla por error en la asignación de memoria al buffer	Sí	Transitoria	Crash ó Bizantina
Falla por error en el link de comunicación	Sí	Transitoria ó Intermitente	Crash ó Bizantina

Tabla 4.1: Resumen de la clase de fallas.

Sobre la tabla anterior se tienen las siguientes consideraciones:

En la tabla se observa que la presencia de cualquiera de las fallas puede provocar que la comunicación se interrumpa (crash) o en caso de continuar, provocar un estado de inconsistencia en el programa paralelo (falla bizantina) de tal menera que los siguientes estados

del sistema o la regla de generación de mensajes se efectúen de manera arbitraria [20], o incluso, es posible un escenario en que los demás componentes del programa presenten un funcionamiento correcto [24].

4.2. Descripción de los patrones de software

En la sección anterior se describe una clase de fallas sobre el conjunto de procesos que integran un componente de comunicación de un programa paralelo. En las secciones siguientes se establece la solución propuesta en esta tesis para cada uno de los elementos de la clase de fallas.

4.2.1. Paso de Mensajes Paralelo con Buffer no Delimitado

El patrón describe el diseño de un componente de comunicación para un programa paralelo basado en el patrón Message Passing Pipe [25] [Sección 3.2.1] con una variante en el componente que implementa el buffer. Esta variante evita la presencia de una falla en la comunicación originada por un error en la asignación de memoria al buffer para un nuevo elemento a almacenar [Sección 4.1.1].

Ejemplo

Un buffer es implementado utilizando una estructura de datos que permite preservar el orden en que los datos son recibidos mediante una política FIFO. Supóngase la siguiente estructura de un buffer no delimitado similar a la descrita en [12, 39, 40]:

```
1  ...
2  private Vector messages = new Vector();
3  private int numMessages = 0;
4  ...
5
6  public synchronized void send(Object c){
7     if (c == null)
8        throw new NullPointerException();
9
10    try{
11        mensages.addElement(c);
12        numMensajes++;
13    }catch(Exception e){...}
14 }
```

La extracción de datos del buffer depende de la velocidad con que el componente de procesamiento está consumiéndolos. Si la velocidad con la que se agregan elementos al buffer

es mayor a la velocidad con que se extraen, es posible que se llegue a un punto en que no pueda asignar más memoria a los nuevos datos que genera el componente de procesamiento.

En el momento de presentarse un error de asignación de memoria, el código try/catch evita que el programa termine de manera inesperada, con la desventaja que los datos que llegan a partir de ese momento se pierden al no poder ser almacenados, lo que origina una falla en la comunicación.

Por lo tanto no es suficiente con atrapar el error: es necesario evitar la perdida de datos cuando se presenta este escenario.

Contexto

Se desarrolla un componente de comunicación para un programa paralelo, basándose en el patrón Message Passing Pipe [25] [Sección 3.2.1], en el cual es posible la presencia de una falla provocada por un error en la asignación de memoria al buffer [Sección 4.1.1], la cual provoca que la comunicación no se realice de acuerdo a la especificación funcional del programa paralelo.

Problema

El buffer de un componente de comunicación de un programa paralelo no puede seguir almacenando datos debido a que no es posible asignarle más memoria a la estructura de datos que lo implementa.

La acumulación de datos se origina por una diferencia entre la velocidad en que se generan los datos y la velocidad en la que son consumidos, por lo que al llegar al punto donde no es posible asignar más memoria los datos que lleguan al buffer no pueden ser almacenados.

Fuerzas

Las fuerzas a considerar en la solución propuesta en este patrón son las siguientes:

- El mecanismo debe implementar una estructura auxiliar al buffer en tiempo de ejecución, por lo que es necesario anticipar la falla.
- El orden de los datos se debe conservar en todo momento. La transición de datos del buffer auxiliar al buffer principal deber respetar el orden de los datos.
- El mecanismo debe proporcionar conocimiento acerca del estado del buffer principal, de tal manera que sea posible transferir los datos almacenados en la estructura auxiliar al buffer principal.

- El mecanismo no debe modificar la recepción y el consumo de datos asíncrono del programa paralelo.
- La implementación de un buffer auxiliar debe ser intransendente para el mecanismo extractor de datos del buffer.

Solución

El patrón soluciona el problema implementando un componente encargado de identificar un error en la asignación de memoria al buffer. Una vez atrapado el error, el componente es encargado de crear una estructura auxiliar para almacenar los datos entrantes en tanto el buffer original no tenga disponibilidad de memoria.

El nuevo componente monitorea la capacidad del buffer principal, y en el caso de encontrar disponibilidad de memoria para nuevos datos, es encargado de copiar los datos almacenados en el buffer auxiliar.

Estructura

En la figura 4.3 se muestra la estructura del patrón.

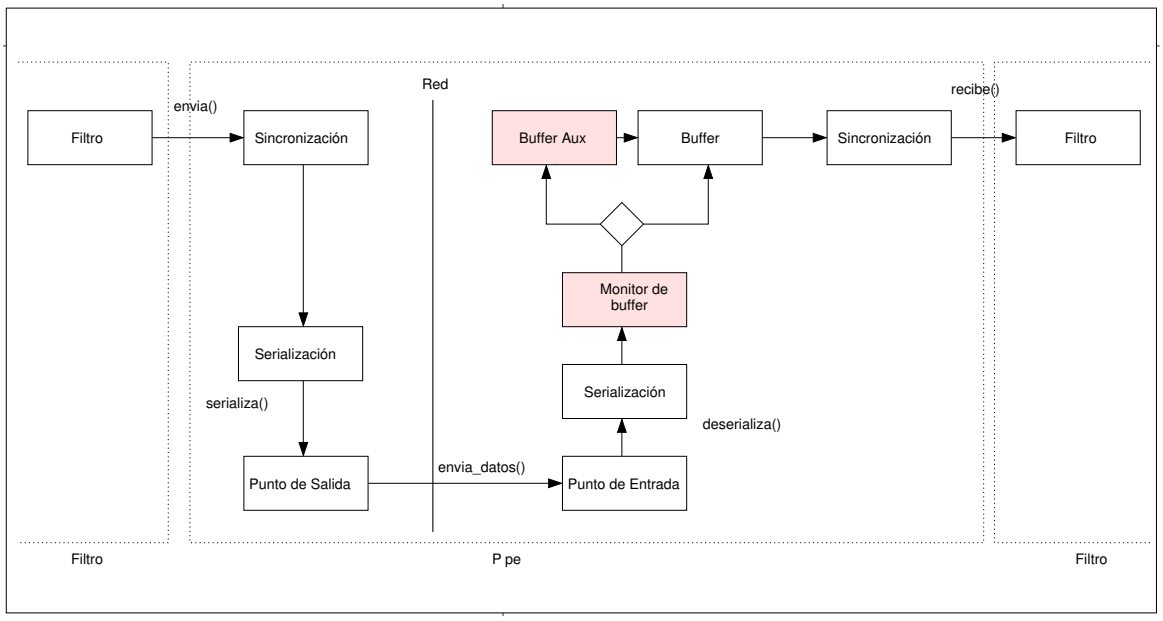

Figura 4.3: Diagrama de colaboración del patrón Paso de Mensajes Paralelo con Buffer No Delimitado.

Participantes

Los participantes son los mismos que se presentan en el patrón Message Passing Pipe [25] [Sección 3.2.1] más los siguientes:

- **Un monitor del buffer.** Este componente es el encargado de monitorear la capacidad del buffer y evitar una falla por error de asignación de memoria. Su responsabilidad consiste en detectar cuándo el buffer está en su límite de su capacidad, y transferir el flujo de mensajes a un buffer auxiliar.

  También tiene como responsabilidad transferir, si es posible, los elementos almacenados en el buffer auxiliar al buffer principal preservando el orden de los mismos.

- **Un buffer auxiliar.** Este componente tiene la responsabilidad de almacenar los datos que ya no son posibles almacenar en el buffer principal. Su implmenentación debe ser idéntica a la implementación del buffer principal.

  Este componente puede crearse de manera dinámica o estática, y entrar en funcionamiento al momento en que se detecta un error en la asignación de memoria en el buffer principal.

Dinámica

La dinámica que presenta el patrón es similar a la descrita en el patrón Message Passing Pipe [25] [Sección 3.2.1]. Sin embargo, el monitor del buffer incluido en el componente modifica la dinámica original.

En la figura 4.4 se muestra gráficamente la nueva dinámica del patrón.

La dinámica del patrón es la siguiente:

- El componente que implementa el buffer recibe un nuevo mensaje que debe almacenarse en la estructura de datos que implementa el buffer.

- El componente monitor del buffer verifica si existen elementos en el buffer auxiliar. De ser cierto, ésto indica que en la recepción del elemento anterior no ha llegado a un límite de la capacidad del buffer principal.

- Si existen elementos en el buffer auxiliar, estos se transfieren al buffer principal de ser posible. En caso contrario, se siguen almacenando en el buffer auxiliar.

- Si no existen elementos en el buffer auxiliar, es posible que aún se tenga espacio en el buffer principal por lo que se intenta almacener ahí mismo.

  Sin embargo es posible que el buffer principal esté en el límite de capacidad y ya no pueda almacenar más elementos, por lo que se atrapa la excepción y el elemento es direccionado al buffer auxiliar.

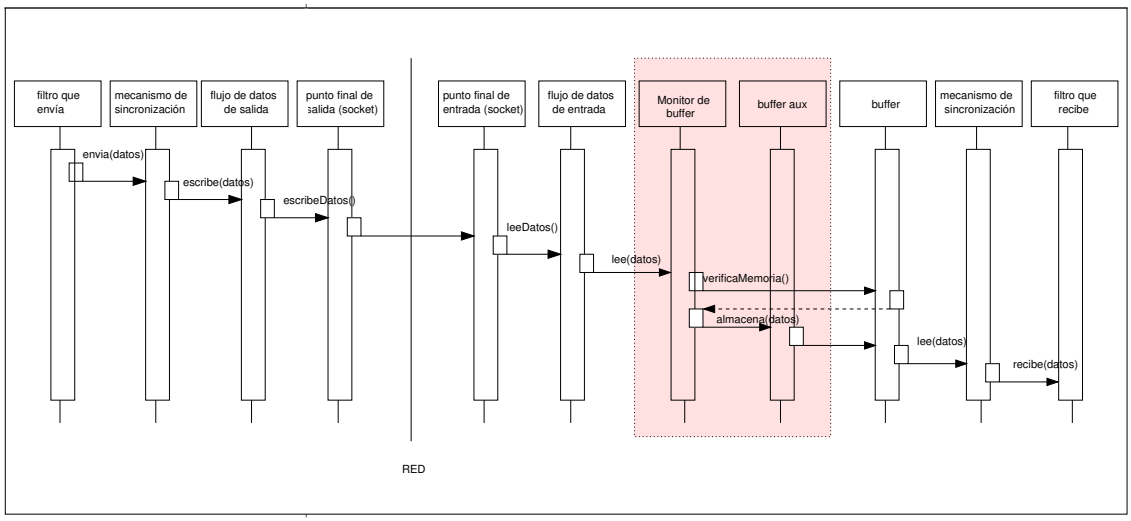

Figura 4.4: Diagrama de secuencia del patrón Paso de Mensajes Paralelo con Buffer no Delimitado.

- Los datos se siguen almacenando en el buffer auxiliar en tanto el buffer principal no pueda almacenarlos.

Implementación

El patrón requiere de la implementación de mecanismos de sincronización, de serialización de datos y puntos finales de entrada y salida de la misma forma en que se implementan en el patrón Message Passing Pipe [25] [Sección 3.2.1].

La implementación del patrón requiere además lo siguiente:

- El monitor del buffer debe estar implementado dentro de un bloque de código que permita atrapar el error de asignación de memoria y continuar con la ejecución del programa.

- El buffer auxiliar debe ser implementado de la misma forma que el buffer principal: mediante un arreglo o vector de algún tipo de objeto o tipo de dato primitivo.

- El buffer auxiliar debe implementar a las mismas operaciones de escritura y lectura bajo el mismo mecanismo de sincronización que implementa el buffer principal.

- El monitor del buffer debe implementar atrapar el error por asignación de memoria del propio buffer auxiliar.

Ejemplo Resuelto

Considerando la estructura general de un buffer presentado en [12, 39, 40], el funcionamiento del mecanismo propuesto se presenta en la figura 4.5.

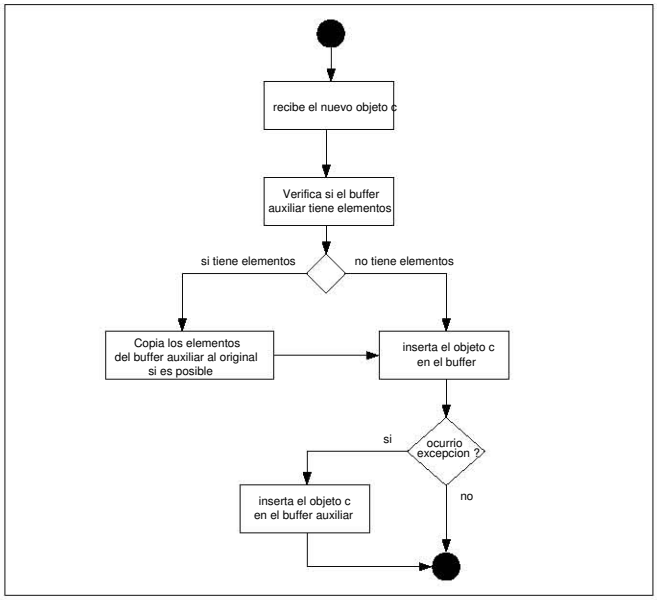

Figura 4.5: Diagrama de flujo para el monitor del buffer.

En la recepción de cada nuevo elemento, es necesario saber si existe un error en el momento de insertar el elemento anterior. Para obtener tal estado, es necesario saber el tamaño del buffer auxiliar: si el tamaño es cero, significa que el buffer principal no ha caido en un desbordamiento. En caso contrario, significa que ha llegado a su límite de capacidad al agregar el elemento.

Si el buffer auxiliar no está vacío, un ciclo *while* se ejecuta para transferir sus elementos al buffer principal. Sin embargo, no es posible asegurar que todos puedan transferirse exitosamente, debido a que el buffer principal puede llegar a su límite nuevamente. Si éste es el caso, el elemento pendiente es almacenado en el buffer auxiliar.

En el cuadro siguiente se muestra el pseudocódigo del buffer que implementa el flujo presentado en la figura 4.5.

```
1  ...
2  private Vector messages = new Vector();
3  private int numMessages = 0;
4  ...
5
6  public synchronized void send(Object c){
7     if (c == null)
8        throw new NullPointerException();
```

```
9
10      try{
11
12          if(mensagesAux.size() > 0){
13              ...
14              while(mensagesAux.size() > 0){
15                  mensages.addElement(mensagesAux.firstElement());
16                  mensagesAux.removeElementAt(0);
17              }
18              ...
19          }else{
20                   mensages.addElement(c);
21                   numMensajes++;
22          }
23
24      }catch(OutOfMemoryError e){
25
26              try{
27                  ...
28                  mensagesAux.addElement(c);
29                  numMensajes++;
30                  ...
31
32              }catch(OutOfMemoryError f){...}
33
34      }
35  }
```

Consecuencias

1. **Beneficios**

   - *El patrón puede evitar la presencia de una falla en la comunicación de forma sencilla.* Tan solo es necesario replicar el buffer para reducir las posibilidades de falla debido a un desbordamiento en la capacidad del buffer.
   - *El monitoreo de la capacidad del buffer es dinámico.* El monitoreo no se realiza mediante un contador y su comparación a un número determinado. Se sabe del desbordamiento cuando el buffer auxiliar tiene al menos un elemento almacenado, lo que representa una gran ventaja en la implementación en diferentes plataformas y lenguajes en donde la asignación de memoria a un objeto puede variar.
   - *El desempeño del programa no se ve afectado.* Dado que en cada recepción de un nuevo mensaje es necesario realizar a lo más dos comparaciones, la complejidad

puede considerarse como constante, por lo que el desempeño del programa no se ve afectado.

2. **Inconvenientes**

   - *El buffer auxiliar también puede presentar un desbordamiento en su capacidad de almacenamiento.* Si se presenta tal situación, es decisión del diseñador implementar otro buffer auxiliar adicional.

4.2.2. Monitor de Paso de Mensajes Paralelo

El patrón describe el diseño de un componente de comunicación basado en el patrón Message Passing Pipe [25] [Sección 3.2.1] con un componente adicional para monitorear el envío y recepción de los mensajes enviados por la red. El patrón incorpora las técnicas descritas en los patrones Acknowledgment, I Am Alive, Are you Alive [32] [Sección 3.1] para implementar el monitor de comunicaciones.

Este patrón implementa un mecanismo para prevenir fallas en la comunicación originadas por un errores transitorios o intermitentes durante la transmisión del mensaje a través de un canal de comunicación [Sección 4.1.2].

Ejemplo

Supóngase un programa paralelo distribuido que realiza una comunicación a través de un canal no confiable mediante las funciones *send* y *receive* propuestas en [39]:

```
1   public void send(Object m) {
2         ...
3         try {
4            ...
5            outObj.writeObject(m);
6            outObj.flush();
7            ...
8         } catch (IOException e) {
9            ...
10        }
11  }
12
13  public Object receive() {
14        Object o = null;
15
16        try {
17           o = inObj.readObject();
18        } catch (Exception e) {
19           ...
```

```
20            }
21        }
22        return o;
23    }
```

Tanto en el envío como en la recepción de los datos pueden presentarse errores durante las transmisión de datos [Sección 4.1.2]. Este comportamiento se muestra de forma gráfica en la figura 4.6.

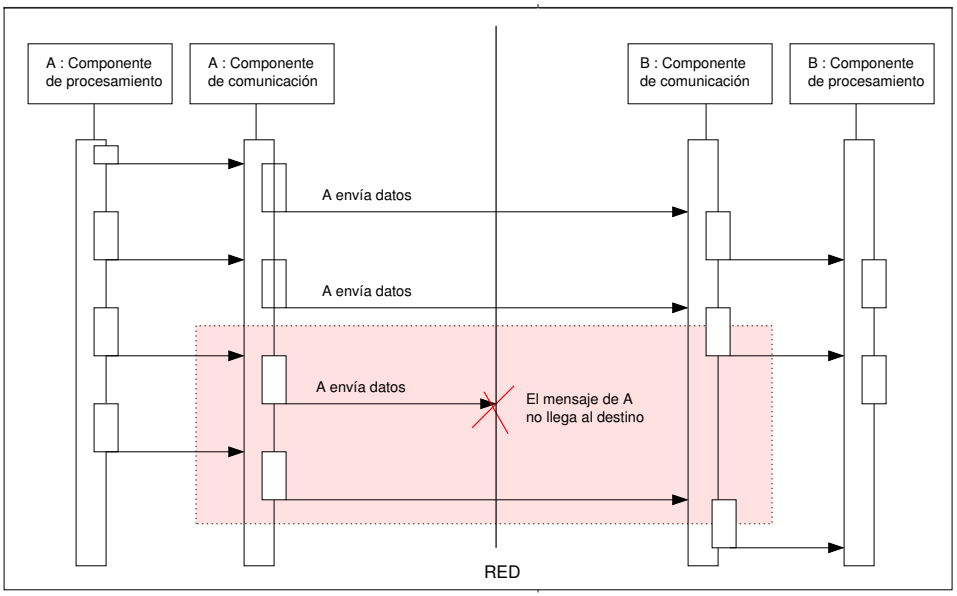

Figura 4.6: Falla en la transmisión del mensaje.

El componente A envía un mensaje, el cual no llega a su destino por alguna razón. Si el error que ocasiona la falla es atrapado por un bloque de excepción, la ejecución continúa de manera normal, pero se ha perdido información.

Esta pérdida de información tiene dos posibles consecuencias: que el programa continúe con el cómputo y al final de tenga un resultado incorrecto, o que se llegue a una interrupción completa del cómputo.

Contexto

Se desarrolla un componente de comunicación para un programa paralelo implementado en una arquitectura de memoria distribuida basándose en el patrón de diseño Message Passing Pipe [25] [Sección 3.2.1] y se utiliza un canal de comunicación que no ganantiza el envío o recepción correcto de los mensajes. Se desea entonces asegurar el envío y recepción de mensajes con el objetivo de tener un resultado correcto.

Problema

Una falla por un error durante la transmisión de un mensaje a través de un canal de comunicación [Seccién 4.1.2] provoca que el intercambio de información entre componentes de procesamiento de un programa paralelo no se realice de acuerdo a la especificación del programa.

El error durante la transmisión puede presentarse de manera transitoria o intermitente, teniendo como consecuencia una falla del tipo bizantina o incluso una falla del tipo crash, lo cual en ambos casos significa el desperdicio de todo el cómputo.

Fuerzas

El patrón resuelve el problema balanceando las siguientes fuerzas:

- El componente de comunicación necesita de un mecanismo que informe el estado de la entrega de un mensaje.
- Un problema de ruteo o de tráfico de red puede provocar un retraso en la entrega del mensaje, el cual puede ser interpretado por el componente de comunicación como un error.
- La comunicación entre componentes de procesamiento puede incrementarse de tal manera que sature el canal de comunicación.
- El componente de comunicación deber preservar el orden de los datos y verificar la duplicidad de los mismos.
- Se debe garantizar el paralelismo del sistema respetando el envío asíncrono de mensajes.
- Se debe establecer un tiempo de espera para la recepción de mensajes antes de considerar que existe un error en la comunicación.

Solución

El patrón soluciona el problema agregando un componente extra a los ya definidos en el patrón Message Passing Pipe [25] [Sección 3.2.1] . Este componente consiste en dos partes, en un monitor de envío de mensajes y un monitor de recepción, los cuales son encargados de identificar si existen errores durante la transmisión de los mensajes.

El componente está basado en los patrones Acknowledgment, I Am Alive, Are you Alive [32] [Sección 3.1], con la diferencia que este componente no genera una señal cada cierto tiempo, sino que utiliza los mismos datos como señal de reconocimiento en donde el

intervalo de tiempo está sujeto a la generación de nuevos datos por parte del componente de procesamiento.

A fin de garantizar el paso asíncrono de mensajes y con ello el paralelismo, el componente también agrega un buffer en el componente monitor, el cual almacena los datos mientras se recibe el acuse de recibo del dato enviado.

Estructura

En la figura 4.7 se muestra la estructura del patrón.

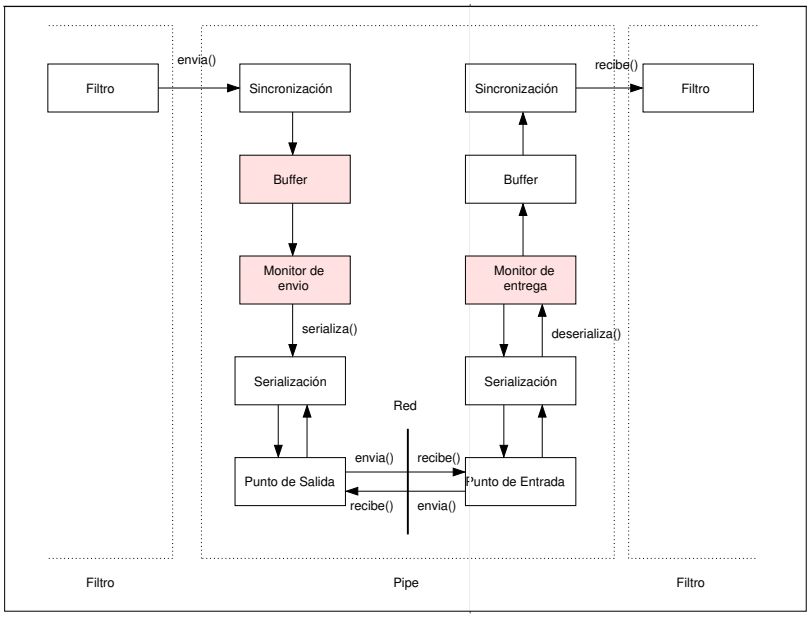

Figura 4.7: Diagrama de colaboración del patrón Monitor de Paso de Mensajes Paralelo.

Participantes

Los participantes que conforman el patrón son los mismos descritos en el patrón Message Passing Pipe [25] [Sección 3.2.1] más los siguientes:

- **Monitor de envío.** Este monitor es el encargado de generar un identificador único para cada mensaje, y recibir el acuse de recibo del monitor de entrega dentro de un rango de tiempo predefinido.
- **Monitor de recepción.** Este monitor es el encargado de recibir el mensaje, validar el id anexo al mensaje y generar el acuse de recibo en el caso de no haberlo recibido antes o ignorarlo en caso contrario.

- **Un buffer de salida.** Su estructura es similar al que ya implementa el patrón Message Passing Pipe [25] [Sección 3.2.1], y su tarea es permitir se sigan recibiendo datos del componente de procesamiento, mientras se espera el acuse de recibo del monitor de recepción.

Dinámica

La dinámica original del patrón Message Passing Pipe [25] [Sección 3.2.1] se ve modificada debido a la inclusión del componente monitor. En la figura 4.8 se muestra la dinámica resultante.

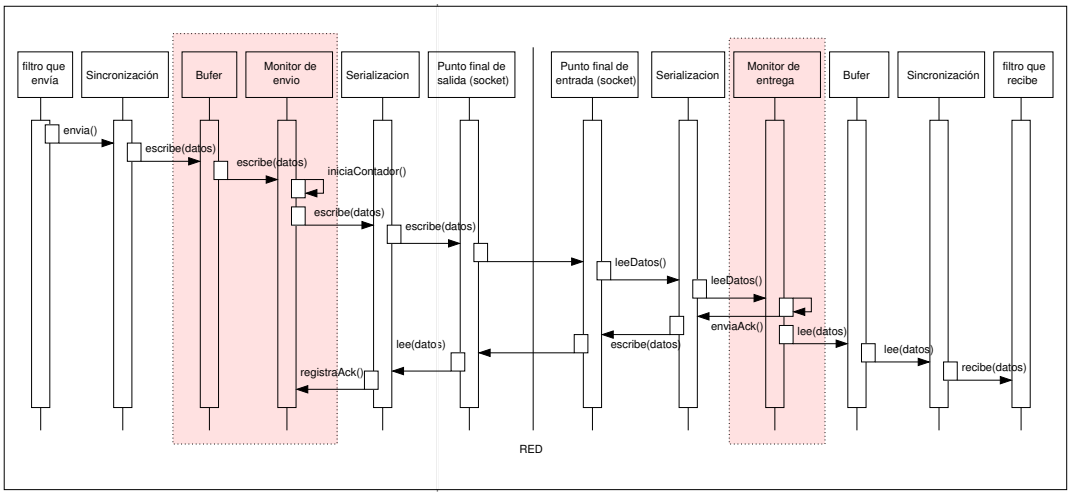

Figura 4.8: Diagrama de secuencia del patrón Monitor de Paso de Mensajes.

- **Monitor de envío.** Una vez recibido el mensaje, el monitor de envío genera un identificador único e irrepetible durante la ejecución del programa para cada mensaje. Enseguida de enviar el mensaje, inicia el temporalizador que define el tiempo de espera para recibir el acuse de recibo por parte del monitor de recepción.

  El monitor de envío considera que existe algún error en el envío del mensaje cuando el tiempo de espera expira y no haya registrado un acuse de recibo para el identificador único generado.

- **Monitor de entrega.** Al momento de recibir el mensaje obtiene el identificador anexo al mismo y verifica si el mensaje ya a sido recibido o no. En cualquier caso el monitor genera un acuse de recibo con el mismo identificador.

  Si el mensaje no se ha recibido antes, el monitor envía el mensaje al siguiente componente. En caso contrario simplemente se descarta el mensaje.

Implementación

El patron requiere implementar los mismos componentes del patrón Message Passing Pipe [25] [Sección 3.2.1] más dos componentes monitores que se describen a continuación:

- El monitor de envío requiere:
  - **Un generador de identificadores únicos y consecutivos.** La eleccion de un identificador es una decisión del diseñador del sistema. Puede consistir desde un dato primitivo hasta un objeto específico.
  - **Una función que anexe el identificador a los datos a enviar.** Tanto el identificador como los datos a enviar deben integrarse de forma tal que formen una sola entidad, de tal manera que sea posible serializarlos como un solo objeto y deserializarlo como tal.
  - **Una función que calcule un cierto tiempo de espera para la recepción del acuse de recibo.** Al finalizar el tiempo calculado por esta función, la misma debe verificar si el acuse de recibo ha llegado durante ese periodo de tiempo.
  - **Un buffer de salida.** Se implementa con cualquier estructura de datos que mantenga una politica FIFO.
- El monitor de recepción requiere de:
  - **Una función que separe el identificador del mensaje de los datos.** Una vez deserializado el mensaje, se debe obtener el identificador del mismo, con el fin de poder compararlo con el histórico de identificadores.
  - **Una estructura de datos para almacenar el histórico de identificadores de cada mensaje recibido.** La razón de implementar un histórico de identificadores de debe a que no es posible utilizar el buffer para verificar si un mensaje ya ha sido recibido, ya que los mensajes del buffer están extrayendose en todo momento.
  - **Una función que sea capaz de decidir si un mensaje es correcto.** Resultado de la comparación del identificador recibido y su comparación con el histórico de identificadores.
  - **Una función que genere un mensaje de respuesta utilizando el mismo identificador anexo al mensaje.** Es el acuse de recibo que indica que el mensaje ha sido recibido satisfactoriamente.

Consecuencias

1. **Beneficios**

   - El sistema de comunicación puede sobreponerse a error durante la transmision del mensaje, evitando un falla que provoque un resultado incorrecto o el desperdicio de todo el cómputo.
   - El mecanismo es ideal en programas paralelos con granularidad gruesa.
   - La implementación del monitoreo de mensajes utilza funciones simples de comparación de identificadores, lo cual se realiza en tiempo constante.
   - La solución respeta la comunicación asíncrona de mensajes. Gracias al buffer de envío, los componentes de procesamiento pueden seguir procesando independientemente de que el componente de comunicación esté esperando el acuse de recibo.
   - Debido a que el siguiente mensaje solo es enviado hasta el aseguramiento del anterior, se evita el problema de la recepción de mensajes traslapados.
   - La solución puede ser adaptada para identificar otros tipos de fallas, por ejemplo, si el número de intentos de envío de un mensaje es agotado, puede tratarse de otra falla y tomar las medidas necesarias.

2. **Desventajas**

   - El número de comunicaciones para lograr el monitoreo de los mensajes se duplica con respecto al patrón original.
   - La latencia de la red puede impactar en el desempeño del programa paralelo. Una red con mucho tráfico puede originar muchos retardos y como consecuencia el reenvio de mensajes sin que sean necesarios.
   - El mecanismo propuesto no funciona para fallas del tipo crash en la red de comunicación o para fallas intermitentes con una larga duración (larga duración respecto al cálculo de tiempo estimado para el envío o recepción de un mensaje en condiciones normales).
   - El incremento del tiempo de comunicación resultado de la espera del acuse de recibo puede provocar un error de asignación de memoria en el buffer de salida.

4.2.3. Monitor de Canal de Paso de Mensajes Paralelo

El patrón describe el diseño de un componente de comunicación para un programa paralelo que necesita un canal bidireccional de paso de mensajes para el intercambio de informacion entre componentes de procesamiento. El diseño se basa en el patrón Message Passing Channel [25] [Sección 3.2.1] con un componenete adicional para monitorear el envío y recepción de los mensajes enviados por la red con el objetivo de prevenir fallas debido a errores durante la transmisión de datos a través de la red [Sección 4.1.2].

Ejemplo

Supóngase un programa paralelo que simula un autómata celular (AC) que modela la propagación de fuego en un bosque. Cada célula representa un árbol, cuyo estado actual se calcula en base a su propio estado y el estado de sus vecinos. En la figura 4.9 se muestra el modelo del AC y el tipo de comunicación existente.

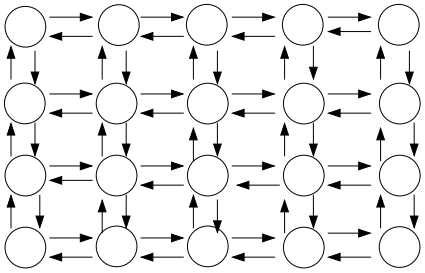

Figura 4.9: Un AC que simula la propagación de fuego en un bosque.

El AC puede implementarse en un arreglo de procesadores como lo propone Brinch Hansen [41] , en la que la malla completa se divide en varias submallas y cada una de ellas es asignada a un procesador.

Para cada paso de tiempo, cada submalla debe comunicar el estado actual de sus celulas frontera y al mismo tiempo recibir la misma información de sus vecinos.

En la figura 4.10 se muestra la distribución del modelo en un arreglo de procesadores en una arquitectura de memoria distribuida.

Si existe la presencia de un error durante el envío o recepción de un mensaje que impida que los datos lleguen a su destino, se tiene como consecuencia que una célula no pueda calcular su nuevo estado, y por lo tanto, no pueda enviar la nueva información generada. Este error origina una falla en la comunicación provocando un resultado incorrecto o la interrupción del procesamiento. En ambos casos significa el desperdicio del cómputo.

Contexto

Se diseña un programa paralelo distribuido que necesita implementar un canal bidirecional

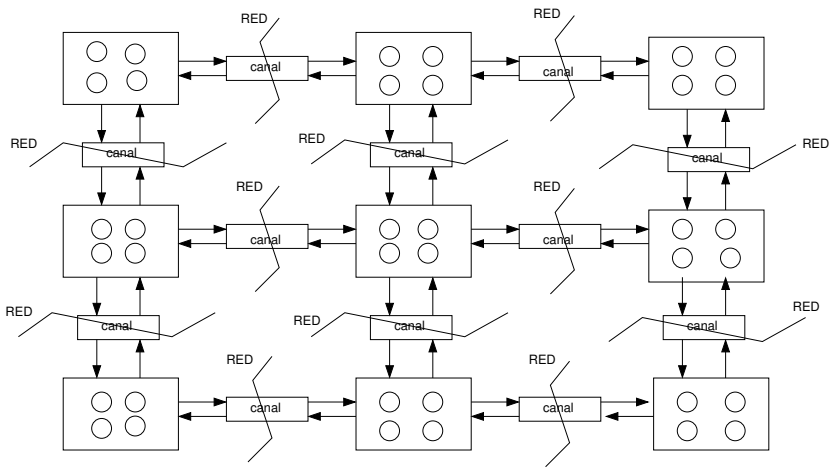

Figura 4.10: Distribución del AC en un array de procesadores.

de comunicaciones asíncrono para el paso de mensajes entre componentes de procesamiento. El componente de comunicación se diseña en base al patron Message Passing Channel [25] [Sección 3.2.1] y se implementa en una red de comunicación en donde no está garantizada la entrega de mensajes debido a errores durante la transmición de datos. Los errores que se presentan en la red de comunicación de presentan de una forma transitoria o intermitente.

Problema

Un error transitorio o intermitente durante el envío de datos entre componentes de procesamiento de un programa paralelo de memoria distribuida provocan que la comunicación no se realice conforme a la especificación funcional del programa. El error que se presenta puede originar una falla bizantina o del tipo crash en el componente de comunicación, y propagarse al resto del programa.

Los componentes de comunicación desconocen si la entrega del mensaje es satisfactoria o no, por lo que de continuar con su ejecución pueden provocar desde un resultado incorrecto hasta la interrupción completa del cómputo.

Fuerzas

El patrón resuelve el problema balanceando las siguientes fuerzas:

- El componente de comunicación necesita conocer el estado de la entrega de cada mensaje.
- Un retardo en la entrega del mensaje puede ser considerado como un error en la comunicación, aunque no se trate de un error en sí. El retardo puede ser provocado por un problema de ruteo y por el tráfico mismo de la red.

- La comunicación entre componentes de procesamiento puede incrementarse de tal manera que sature el canal de comunicación.
- El componente de comunicación deber preservar el orden de los datos y verificar la duplicidad de los mismos.
- Se debe garantizar el paralelismo del sistema respetando el envío asíncrono de mensajes.
- Se debe establecer un tiempo de espera para la recepción de mensajes antes de considerar que existe un error en la comunicación.

Solución

El patrón soluciona el problema de prevenir una falla en la comunicación, implementando un componente de monitoreo de mensajes a los componentes existentes en el patrón Message Passing Channel [25] [Sección 3.2.1].

El componente consta de dos partes: una encargada de monitorear el envio de mensajes, y otra que es encargada de recibir el mensaje y responder con un acuse de recibo al primero. Esta técnica es similar a la presentada en los patrones Acknowledgment, I Am Alive, Are you Alive [32] [Sección 3.1], pero con variantes que permiten la comunicación asíncrona de mensajes y en donde la señal de reconocimiento son los mismos datos enviados.

Debido a que el patrón realiza un envío bidireccional de mensajes, el componente de monitoreo es replicado para ofrecer un monitoreo en ambas direcciones.

Estructura

En la figura 4.11 se muestra la estructura del patrón.

Participantes

Los participantes que conforman este patrón son los mismos que se describen en el patrón Message Passing Channel [25] [Sección 3.2.1] más la incorporacion de los siguientes:

- **Monitor de envío.** Este monitor es el encargado de generar un identificador único para cada mensaje, y recibir el acuse de recibo del monitor de entrega dentro de un rango de tiempo predefinido.
- **Monitor de recepción.** Este monitor es el encargado de recibir el mensaje, validar el id anexo al mensaje y generar el acuse de recibo en el caso de no haberlo recibido antes o ignorarlo en caso contrario.

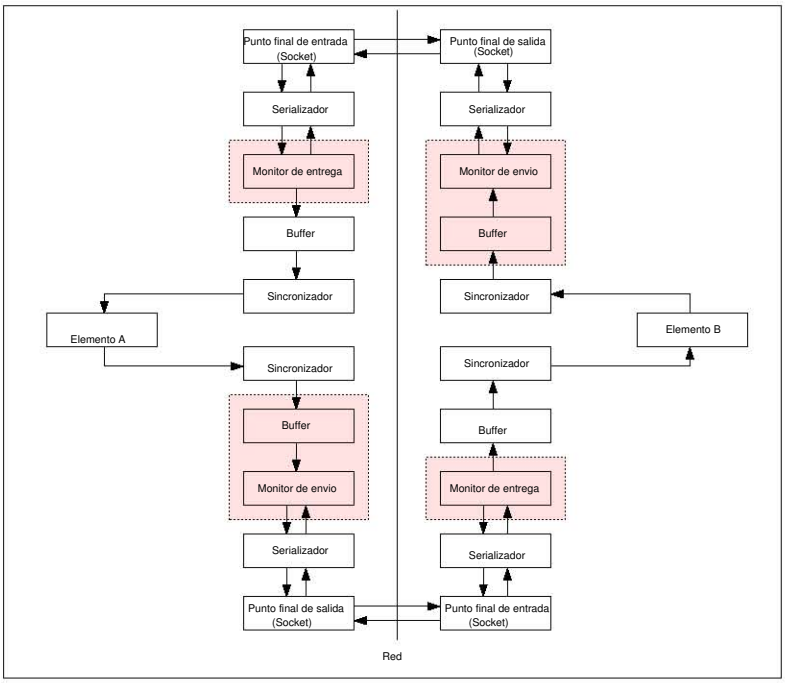

Figura 4.11: Estructura del patrón Monitor de Canal de Paso de Mensaje Paralelo.

- **Un buffer de salida.** Su estructura es similar al que ya implementa el patrón Message Passing Channel [25] [Sección 3.2.1] y su tarea es permitir seguir recibiendo datos del componente de procesamiento mientras se espera el acuse de recibo del monitor de recepción.

Los participantes descritos son los necesarios para la comunicación en una dirección de los datos. Para establecer la comunicación bidireccional, es necesario replicarlos para establecer la comunicación en la otra dirección.

Dinámica

Con la inclusión de los monitores al patrón Message Passing Channel [25] [Sección 3.2.1] la dinámica original se modifica, resultando la dinámica que se muestra en la figura 4.12.

- Los datos son recibidos del componente de procesamiento y depositados en el **buffer de salida** mediante un mecanismo de sincronización.
- El **monitor** de envío toma los datos del buffer y genera un número consecutivo para identificarlos. El número identificador es anexado a los datos e inmediatamente despues se inicia el conteo del tiempo de espera para recibir el acuse de recibo.
- Los datos son serializados y enviados a través de la red. En el punto de recepción, los datos son deserializados y enviados al **monitor de recepción**.

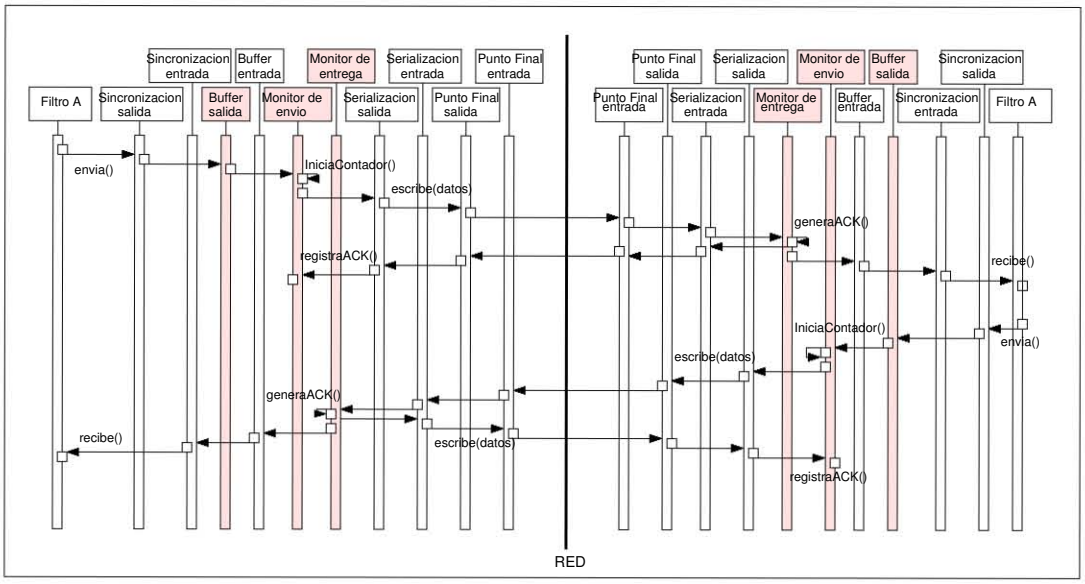

Figura 4.12: Diagrama de secuencia del patrón Monitor de Canal de Paso de Mensaje Paralelo.

- El **monitor de recepción** se encarga de obtener el número identificador de los datos, analizar si ese identificador ya ha sido recibido, y finalmente regresar el acuse de recibo al **monitor de recepción** enviando los datos al componente siguiente.

La dinámica es similar para el envío de mensajes en la dirección contraria utilizando los monitores correspondientes.

Implementación

El patrón requiere de la misma implementación necesaria para el patrón Message Passing Channel [25] [Sección 3.2.1] más lo siguiente:

- El monitor de envío requiere de:
  - **La contrucción de un generador de números consecutivos.** Mediante algún tipo de dato primitivo u objeto se utilizan como identificadores para los mensajes.
  - **La construcción de una función que integre el identificador a los datos del mensaje.** De una manera que sea posible separar dichos datos al momento de la recepción del mensaje.
  - **Una función que calcule el tiempo de espera para la recepción del acuse de recibo.** Iniciando el cálculo en el momento del envío del mensaje.

- **Un buffer de salida.** Que implemente una politica FIFO.

- El monitor de recepción requiere de:

  - **Una función que separe el identificador del mensaje de los datos.** El cual es utilizado en la validación del mensaje.
  - **Una estructura de datos que almacene el historial de los identificadores de los mensajes recibidos.** De tal manera que sea posible saber si un mensaje debe ser descartado.
  - **Un comparador que decida si un mensaje es válido.** Comparando su identificador con respecto al historial de identificadores.
  - **Una función que genere un acuse de recibo.** En base al identificador anexo al mensaje.

- Para la implementación de un flujo bidireccional, cada componente de comunicación debe implementar un monitor de envío y un monitor de recepción de forma independiente.

- Cada componente de comunicación debe implementar un buffer de salida para generar un flujo bidireccional.

Consecuencias

- **Beneficios**

  - El sistema de comunicación puede sobreponerse a error durante la transmisión del mensaje, evitando una falla en la comunicación reenviando un número predefinido de veces el mismo mensaje.
  - El tiempo de espera para el acuse de recibo es totalmente configurable, así como el número de intentos de envío.
  - Se conserva la comunicación asíncrona a pesar de la espera del acuse de recibo de cada mensaje.
  - Dado que el paso de mensajes es bidireccional, solo es necesario desarrollar una vez los componentes propuestos y replicarlos cuantas veces sea necesario.
  - Debido a que los mensajes solo son enviados si el mensaje anterior fue asegurado, se evita el problema de recepción de mensajes fuera de orden.

- La solución puede ser adaptada para identificar otros tipos de fallas, por ejemplo, si el número de intentos de envío de un mensaje es agotado, puede tratarse de otra falla y tomar las medidas necesarias.

- **Desventajas**

  - El número de comunicaciones para lograr el monitoreo de los mensajes se duplica con respecto al patrón original. Esto afecta directamente al desempeño del programa.
  - La latencia de la red puede impactar en el desempeño del programa paralelo debido al reenvio innecesaro de mensajes.
  - El mecanismo propuesto no funciona para fallas del tipo crash en la red de comunicación. En este caso, el cómputo se interrumpe inevitablemente.
  - El incremento del tiempo de comunicación resultado de la espera del acuse de recibo puede provocar un error de asignación de memoria en el buffer de salida.

4.2.4. Resumen

La serie de patrones propuestos y descritos en este capítulo ofrecen una solución de prevención o corrección de errores con el objetivo de evitar la presencia de fallas en un componente de comunicación de un programa paralelo. La solución propuesta en estos patrones está diseñada de tal forma que no rompe con la dinámica de los patrones ya existentes para componentes de comunicación.

La estrategia del envío de una señal de reconocimiento propuesta en el sistema de patrones de la sección 3.1 es utilizada en los patrones Monitor de Paso de Mensajes Paralelos y el patrón Monitor de Canal de Paso de Mensajes Paralelo. Sin embargo, la señal de reconocimiento no es empleada con el objetivo de verificar el buen estado de un componente del sistema, más bien se utiliza de tal forma que proporciona información acerca de la entrega y recepción de los mensajes enviados por la red de un componente de procesamiento a otro.

Para el patrón Paso de Mensajes Paralelo con Buffer no Delimitado, se utiliza la replicación de componentes, de manera similar como lo emplean los patrones Passive Replication, Semi-Pasive Replication, Semi- Active Replication y Active Replication 3.1. La diferencia radica en que el componente replicado (el buffer, en este caso) es empleado para evitar que el componente principal deje de dar un servicio correcto, contrario a la replicación que se utiliza en los patrones mencionados, en donde el componente replicado entra en escena

cuando el componente principal ha dejado de funcionar y deja de proporcionar un servicio correcto.

En la literatura de patrones de software, la implementación completa de la solución propuesta no se integra dentro de la descripción misma del patrón. El motivo de tal situación se debe a que un patrón solo es la descripción de una solución dentro de un contexto y se deja que el diseñador o programador utilice las herramientas que sean convenientes para su implementación, es decir, el lenguaje o estructuras de datos por mencionar dos ejemplos.

En el siguiente capítulo se muestra la implementación completa de cada uno de los patrones propuestos en este capítulo, con el objetivo de demostrar que la solución propuesta en los patrones es posible llevarla a cabo en un ejemplo práctico. Para cumplir tal objetivo es necesario inducir los errores sobre los cuales se está ofreciendo una solución.

Capítulo 5

Evaluación experimental de los patrones propuestos

Hay dos maneras de diseñar software:

una es hacerlo tan simple que sea obvia su falta de deficiencias,

y la otra es hacerlo tan complejo que no haya deficiencias obvias.

C.A.R. Hoare

Este capítulo tiene como objetivo realizar una evaluación de los patrones de software propuestos en el capítulo anterior. En cada implementación se simulan errores que se debe prevenir o corregir, según sea el caso, con el objetivo de realizar una evaluación experimental de los patrones propuestos. La simulación de los errores se limita a aquellos errores descritos en la sección 4.1 para los cuales se propone una solución.

Se utiliza el lenguaje de programación Java, el cual es un lenguaje orientado a objetos y multiplataforma, lo cual permite crear un sistema distribuido heterogéneo gracias a la portabilidad que ofrece la máquina virtual.

El envío de datos se realiza mediante el protocolo UDP, que entre sus características se encuentra la ausencia de mecanismos de pérdida de mensajes o duplicación de los mismos delegando esa responsabilidad a las aplicaciones que lo utilizan. Debido a las característica mencionada, es el contexto en el cual los patrones descritos en el capítulo anterior tienen mayor sentido.

5.1. Evaluación del patrón de Paso de Mensajes Paralelo con Buffer no Delimitado

La evaluación del patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 4.2.1] consiste en un programa paralelo en el cual un componente de procesamiento genera un arreglo de números aleatorios, y otro componente de procesamiento realiza el cálculo de la media de todos ellos.

Debido a que en la presente tesis interesa la parte de la comunicación, no es de importancia lo que los componentes de procesamiento realizan. El objetivo es mostrar el funcionamiento de los componentes de comunicación diseñados en base al patrón propuesto ante una falla por asignación de memoria [Sección 4.1.1].

En primer lugar se describe la implementación en un lenguaje de alto nivel cada uno de los participantes que integran el patrón. Enseguida se describen las ejecuciones del programa paralelo, las cuales se llevan a cabo tomando en cuenta dos escenarios: sin el mecanismo de corrección de errores y con el mecanismo de corrección de errores, de tal manera que sea visible el funcionamiento del mecanismo de corrección.

5.1.1. Descripción de la implementación de los participantes en el patrón

La implementación de cada uno de los participantes se describe a continuación:

- **Sincronización.** La recepción y envío de mensajes se realiza mediante funciones enviar y recibir, las cuales tienen la propiedad *synchronized* [1] de Java que implementa la sincronización de accesos a dichos metodos. Aunque la estrategia utilizada en Java al emplear la propiedad *synchronized* es efectiva, puede presentar problemas de deadlook o starvation [42], sin embargo es suficiente para los objetivos de ésta tesis.

```
1  public synchronized Object recibir(){
2       Object dato = null;
3       try{
4            if(messages.isEmpty()){
5               dato = null;
6            }else{
7              dato = messages.firstElement();
8              messages.remove(0);
```

[1] El lenguaje Java provee la palabra reservada *synchronized*, la cual permite al programador accesar a un recurso muy similar a la exclusión mutua de bloqueo, previniendo que dos o más hilos llamen métodos de un mismo objeto de manera simultánea [40]

```
9           }
10
11      }catch(Exception e){}
12          return dato;
13  }
14
15  public synchronized void enviar(Object obj,DatagramSocket socket){
16        try{
17            byte[] datoEnBytes = (byte[]) obj;
18            DatagramPacket datoMSG = new DatagramPacket(...);
19
20            socket.send(datoMSG);
21        }catch(Exception w){}
22  }
```

En la función *recibir()*, si el buffer se encuentra vacío se regresa el valor *null* a la función que la invoca. Esta no es la única forma de actuar en el caso de un buffer vacio: si se trata de un sistema multihilos, es posible detener por un momento la ejecución del hilo mientras un dato arriba al buffer [25, 39, 40]. Tal mecanismo se conoce como *Wait-and-Notify* [40].

La función *enviar(Object obj, DatagramSocket socket, int IdMsg)* recibe como parámetros el objeto que va a ser enviado a través de la red y el objeto socket por el cual se envía a la red.

- **El serializador.** La serialización de datos se implementa mediante las funciones *write()* y *read()*, las cuales implementan las funciones *writeObject()* y *readObject()* de la interface *Serializable*.

```
1   public void write(){
2       try{
3           ByteArrayOutputStream bytes = new ByteArrayOutputStream();
4           ObjectOutputStream os = new ObjectOutputStream (bytes);
5           os.writeObject(this);
6           os.close();
7           cadena = bytes.toByteArray();
8       }
9       catch(Exception e){
10      }
11  }
12
13  public Object read (byte [] bytes){
14      try{
15          ByteArrayInputStream byteArray = new ByteArrayInputStream(bytes);
16          ObjectInputStream is = new ObjectInputStream(byteArray);
17          Object obj = is.readObject();
18          is.close();
19
```

```
20              return obj;
21          }catch(Exception e){
22              return null;
23          }
24      }
```

La interface *Serializable* proporcionada en el JDK de Java implementa la funcionalidad de la serialización de datos.

- **El buffer**. La implentación de un buffer consiste en crear una estructura de datos que implementa una política FIFO. Java proporciona diferentes colecciones implementar dicha estructura. Entre ellas, se encuentra la clase *Vector*, que satisface las necesidades de este programa:

```
1       public Vector messages = new Vector();
```

La clase Vector posee los metodos: *isEmpty()*, que verifica si el buffer tiene elementos almacenados o no; *firstElement()*, que obtiene el primer elemento del buffer; *addElement()*, que agrega un elemento al buffer y finalmente el método *remove(int n)*, en donde la variable *n* indica la posición del objeto a remover del vector.

- **El monitor del buffer.** El monitor de buffer necesita una estructura con las mismas propiedades del buffer descrito para almacenar los datos que no puedan almacenarse debido a un error de asignación de memoria [Sección 4.1.1].

```
1           while(true){
2             try{
3               if(messagesAUX.size()>0){
4                     while(messagesAUX.size() > 0){
5                        messages.addElement(messagesAUX.firstElement());
6                        messagesAUX.remove(0);
7                     }
8               }else{
9                     DatagramPacket dato = new DatagramPacket (...);
10                    socket.receive(dato);
11                    Serializador deserializa = new Serializador();
12                    Object obj = deserializa.read(dato.getData());
13                    messages.addElement(obj);
14              }
15            }catch(OutOfMemoryError e){
16                try{
17                   messagesAUX.addElement(datoRecibido);
18                }catch(Exception w){}
19            }catch(Exception f){}
20          }
```

La estructura auxiliar que utiliza el monitor del buffer es el objeto *messagesAUX*, el cual almacena de forma temporal los datos que no puede almacenar el buffer original.

- **El generador de retardos.** El error por asignación de memoria surge de una diferencia entre la velocidad en que se generan los datos y la velocidad en que son consumidos [Sección 4.1.1].

  Para simular este comportamiento se implementa un método que genera un retraso en el componente de procesamiento. Aunque no es un participante en el patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 4.2.1], es necesario describirlo para efectos de reproducir el error de asignación de memoria.

```
1  public static void generaRetraso(int n, int m){
2     try{
3        Thread.sleep((int) Math.floor(Math.random() * (n-m+1) + m));
4     }catch(Exception e){}
5  }
```

  Los parámetros del método indican el intervalo de tiempo en milisegundos en el que un retraso puede ser generado.

- **El componente de procesamiento.** Este componente es el encargado de generar los datos que se envían mediante el componente de comunicación. Se describe a continuación la función que genera los datos y la función que los consume.

```
1
2     public static Object ModuloProcesamientoA(){
3
4         //Esta funcion representa el componente de procesamiento
5         //en ella se genera la informacion a enviar
6         double[] msg = new double[8000];
7
8         for(int m=0;m<msg.length;m++)
9             msg[m] = Math.random();
10
11        Object obj = msg;
12        return obj;
13    }
14
15
16    public static void ModuloProcesamientoB(Object obj){
17          ...
18
19          try{
20             for(int m=0;m<datos.length;m++){
21                 suma += datos[m];
22                 cont++;
```

```
23              }
24              prom = suma / cont;
25              //Se inyecta el retraso
26              generaRetraso();
27           }catch(Exception e){}
28      }
```

Los que realiza el primer método es generar un arreglo de 8000 números aletarorios y asignar este arreglo a un objeto. Cuando el objeto es recibido, el siguiente componente de procesamiento realiza el cálculo del promedio de cada arreglo recibido.

Una vez descritos los participantes del patrón, resta describir las funciones que los implementan.

- **La clase PipeClassA.** Es el inicio del programa.

```
1  public class PipeClassA {
2      ...
3
4     public static void main(String[] args){
5
6           ...
7           try{
8
9             PipeClassA pipeA = new PipeClassA();
10            pipeA.socket = new DatagramSocket(...);
11
12            for (int i = 0; i < 5000; i++){
13
14              //Se obtienen los datos del componente de procesamiento.
15              Object dato = ModuloProcesamientoA();
16
17              //Se serializan los datos
18              Serializador serializa = new Serializador(dato);
19              serializa.write();
20
21              //finalmente se envian los datos
22              PipeClassA.envia(serializa.cadena,pipeA.socket);
23            }
24
25          }catch(Exception w){}
26
27    }
28    ...
29 }
```

- **La clase PipeClassB.** Es la clase que se sitúa en el otro punto de la red. En cualquier otra implementación solo es necesario replicar esta clase cuantas veces sea necesario, sobre escribiendo el código del componente de procesamiento.

```
1
2  public class PipeClassB implements Runnable{
3     ...
4
5    public static void main(String[] args){
6      try{
7
8          //Se crea un hilo de ejecucion para recibir
9          //los mensajes de forma ininterrumpida
10         PipeClassB datoFromNet = new PipeClassB();
11
12         while(true){
13
14           //Se obtiene el dato proveniente del buffer.
15           Object dato =  PipeClassB.recibir();
16
17           if(dato != null){
18              //Aqui se envia el valor al siguiente modulo de procesamiento
19               moduloProcesamientoB(dato);
20           }
21         }
22     }catch(Exception w){}
23   }
24   ...
25 }
```

La primera acción del la clase PipeClassB es lanzar un hilo encargado de implementar el monitor del buffer anteriormente descrito. De esta forma, se garantiza la recepción asíncrona de mensajes.

Si el buffer está vacío, entonces no se ejecuta el modulo de procesamiento B. En el caso de un ambiente multihilos, en recomendable implementar los metodos *wait* y *notify* [25, 39, 40].

En la figura 5.1 se muestra el diagrama de clases que resume lo descrito.

5.1.2. Ejecución del programa paralelo

El objetivo de esta sección es mostrar la ejecución del programa paralelo en un escenario en el que no existe un mecanismo de correción de errores y un escenario en donde se implementa el mecanismo propuesto en el patrón.

Para el primer escenario, se realiza una ejecución del programa paralelo solo con los elementos descritos en el patrón Message Passing Pipe [25] [Sección 3.2.1]. El componente de software que implementa la lectura de datos y almacenamiento de datos se presenta a continuación:

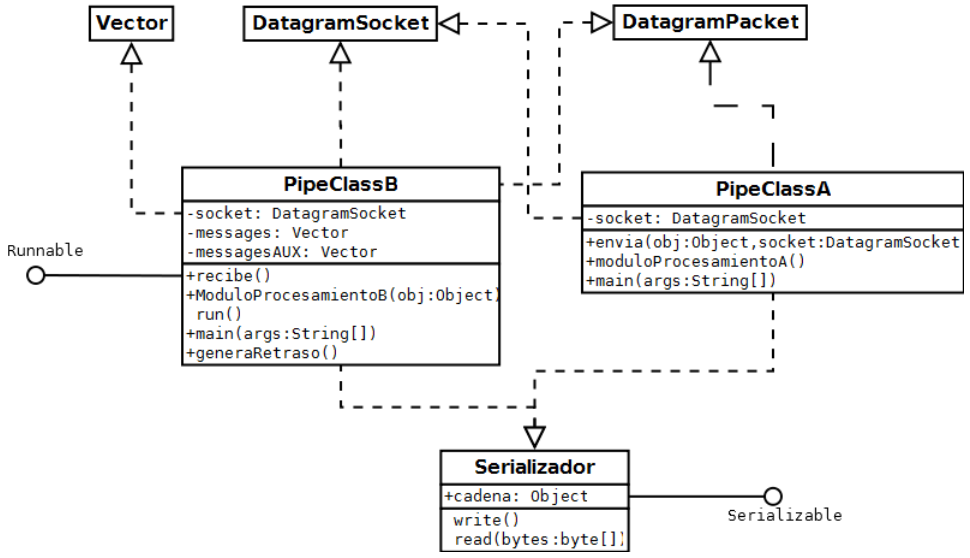

Figura 5.1: El diagrama de clases del programa paralelo.

```
1    while(true){
2        try{
3                ...
4                DatagramPacket dato = new DatagramPacket (...);
5                socket.receive(dato);
6                Serializador deserializa = new Serializador();
7                Object obj = deserializa.read(dato.getData());
8
9                messages.addElement(obj);
10       }catch(Exception e){}
11   }
```

El componente despliega en pantalla el tamaño del buffer en cada lectura de datos que realiza. Debido a que la velocidad con la que se consumen los datos es menor a la velocidad con que se reciben de la red, llega un momento en que no es posible almacenar más elementos y entonces se presenta el error de asignación de memoria. La presencia del error se muestra en la figura 5.2.

En el momento en que se presenta el error, éste es atrapado mediante la estructura try/catch que evita que el programa se interrumpa en ese momento. Sin embargo la funcionalidad del componente de comunicación se ve degradada al no poder recibir los mensajes que se siguen enviando del primer componente.

En la figura 5.2 se muestra la gráfica del consumo de memoria que tiene el proceso que ejecuta la clase PipeClassB, obtenida por la herramienta *jconsole* incluida en la distribución de Java.

En la gráfica se observa el incremento del consumo de memoria heap [2] de la máquina

[2]La memoria heap es el espacio de memoria usado por la Máquina Virtual de Java para todos

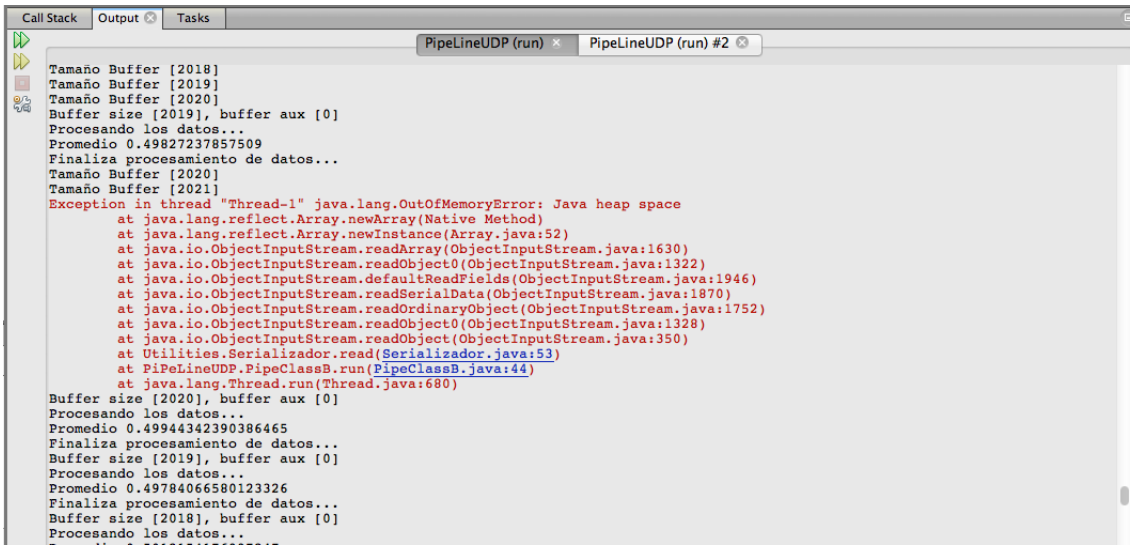

Figura 5.2: Ejecución del programa paralelo sin monitor de buffer.

virtual hasta que llega un momento en que ya no es posible asignar más memoria al proceso.

Para el segundo escenario, se ejecuta el mismo programa pero ahora con los elementos descritos en el patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 4.2.1]. En la figura 5.4 se muestra el mismo punto en donde el programa anterior presento la falla.

El funcionamiento del monitor se describe a continuación:

1. El programa paralelo activa el mecanismo de corrección del error cuando intenta almacenar el elemento número 2022. En primera instancia, el error es atrapado mediante el bloque *try* por lo que el flujo de control se dirige a ejecutar lo que se encuentre en el bloque *catch*.

2. El bloque catch indica que el elemento leído se almacene en el buffer auxiliar. Con esta acción, se evita que el dato se pierda y se pueda seguir con la lectura del siguiente dato proveniente de la red.

3. Para el siguiente dato, el monitor pregunta sobre el número de elementos que tiene el buffer auxiliar. Si la respuesta indica que su tamaño es mayor a cero, el monitor *sabe* que en la lectura anterior existe un error. En este punto hay dos posibles escenarios: que el componente de procesamiento haya extraido elementos del buffer principal y ya esté en la posibilidad de seguir almacenando datos, o que continue en el límite de su capacidad.

los objetos generados con la palabra *new* en una aplicación Java [43].

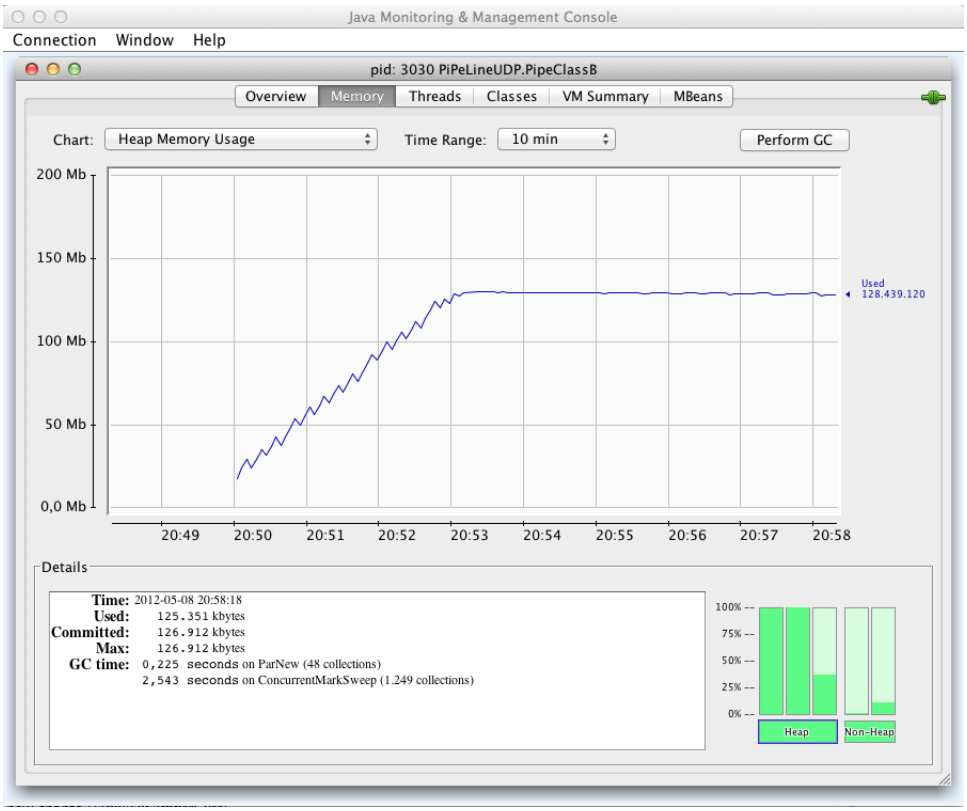

Figura 5.3: Consumo de la memoria del proceso PipeClassB.

4. El monitor procede a migrar los elementos que tiene el buffer auxiliar al buffer principal, independientemente del escenario presentado en el punto anterior. Si es posible lo realiza la migración de datos o en caso contrario se encuentra una vez más con el error de asigación de memoria.

5. Dependiendo del resultado del punto anterior el nuevo dato es almacenado en el buffer principal o en el auxiliar.

5.2. Evaluación del patrón Monitor de Paso de Mensajes Paralelo

La evaluación del patrón consiste en un programa paralelo de memoria distribuida que genera un arreglo de números aleatorios en un componente de procesamiento y el promedio de los mismos en el siguiente componente de procesamiento. El envío de datos a través de la red se realiza mediante componentes de comunicación diseñados conforme al patrón Monitor de Paso de Mensajes Paralelo [Sección 4.2.2].

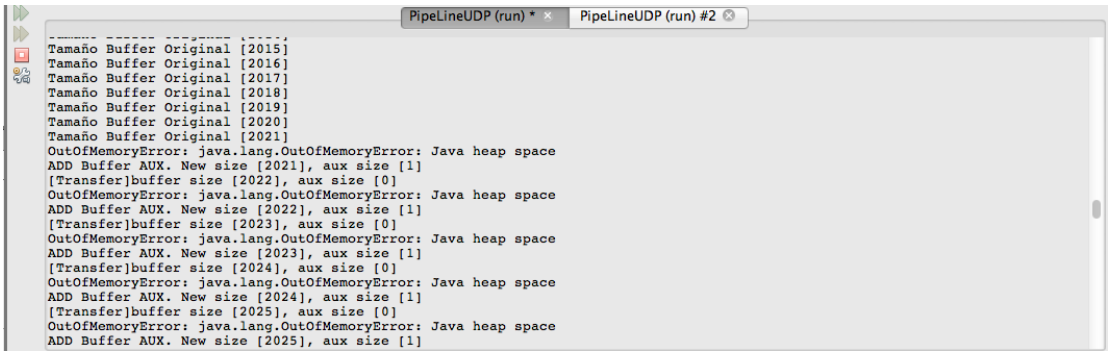

Figura 5.4: Ejecución del programa paralelo con monitor de buffer.

En programa paralelo tiene un componente de software extra, encargado de introducir de manera deliberada errores en la comunicación. El objetivo de simular errores en la comunicación mediante este componente es demostrar la utilidad de la solución propuesta en el patrón de diseño.

5.2.1. Descripción de la implementación de los participantes en el patrón.

A continuación se describe la implementacion de los participantes en el patrón.

- **Sincronización.** Este participante es idéentico al descrito en la implementación del patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 5.1].
- **El serializador.** La serialización es similar al descrito en la implementación del patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 5.1].
- **Buffer de salida y recepción.** La implementación de un buffer requiere cualquier estructura capaz de almacenar datos mediante una política FIFO. En el caso del lenguaje Java, un objeto de la clase Vector implementa tal política.
- **Monitor de envío.** El monitor de envío se implementa con las siguientes funcionalidades:

  1. Una función encargada de extraer los objetos almacenados en el buffer de salida.

```
1  public synchronized Object extraeDatoBuffer(){
2        Object dato = null;
3        try{
4             if(messages.isEmpty()){
5                dato = null;
6             }else{
```

```
7               dato = messages.firstElement();
8               messages.remove(0);
9             }
10        }catch(Exception e){}
11        return dato;
12    }
```

2. Un generador de identificadores para cada mensaje que se envía. El generador puede implementarse conforme a las necesidades de diseño. En este caso, un generador de números consecutivos es suficiente.

```
1  //crea el identificador unico
2  IdMsg++;
3  Hashtable hastable = new Hashtable();
4  hastable.put(IdMsg,dato);
```

El mensaje y el identificador son acoplados en un objeto de la clase Hashtable, el cual consiste de un objeto y una llave para accesar al mismo.

3. Una función que recibe el acuse de recibo del componente de comunicación receptor.

```
1     public synchronized Object recibeACK(DatagramSocket socket){
2
3         Object obj = null;
4         //Parametro configurable
5         int timeout = 500;
6
7         try{
8             DatagramPacket dato = new DatagramPacket (...);
9             socket.setSoTimeout(timeout);
10            socket.receive(dato);
11
12            Serializador deserializa = new Serializador();
13            obj = deserializa.read(dato.getData());
14
15        }catch(Exception e){
16            socket.close();
17        }
18        return obj;
19    }
```

La función implementa una variable que representa el tiempo en milisegundos de espera para la recepción del acuse de recibo. La propiedad *setSoTimeout* del objeto *socket* permite especificar el timeout del mismo pasándole como parametro dicha variable.

- **Monitor de entrega.** Este monitor se implementa con las siguientes funcionalidades:

  1. La extracción del identificador el mensaje incluido en el objeto y su almacenamiento en un tipo de dato que implementa una politica LIFO.

```
1   ...
2   Enumeration e = hastable.keys();
3   Object id = null;
4   while(e.hasMoreElements())
5           id = e.nextElement();
6
7   ...
8   stackHist.push(id);
9   ...
```

     Para esta funcionalidad se utiliza un objeto de la clase Stack que provee el lenguaje Java. Antes de agregar un nuevo elemento al stack es necesario verificar si el identificador del mensaje recibido es el correcto.

  2. Una función que genera el acuse de recibo enviando como mensaje el identificador del mensaje recibido.

```
1  public final synchronized void enviaACK(Object id,DatagramSocket socket){
2      try{
3       Serializador serializa = new Serializador(id);
4       serializa.write();
5       byte[] datoEnBytes = (byte[]) serializa.cadena;
6
7       DatagramPacket datoMSG = new DatagramPacket(...);
8       socket.send(datoMSG);
9
10     }catch(Exception w){}
11 }
```

Una vez descritos los participantes del patrón, resta describir las funciones que los implementan.

- **La clase PipeClassA.** Esta clase crea dos hilos de ejecución, uno de ellos representa el componente de procesamiento encargado de la generación de datos y el otro encargado del envío de los datos.

```
1  public class PipeClassA implements Runnable{
2
3      public void run(){ ... }
4
```

```
5     public static void main(String[] args){
6
7     try{
8         PipeClassA pipeA = new PipeClassA();
9         pipeA.socket = new DatagramSocket(...);
10        pipeA.socketACK  = new DatagramSocket(...);
11        new Thread(pipeA).start();
12
13        while(true){
14            Object dato = extraeDatoBuffer();
15            if(dato != null){
16                //crea el identificador unico
17                IdMsg++;
18                Hashtable hastable = new Hashtable();
19                hastable.put(IdMsg,dato);
20
21                //Serializa los datos
22                Object objenv = hastable;
23                Serializador serializa = new Serializador(objenv);
24                serializa.write();
25                 ...
26                pipeA.envia(serializa.cadena,pipeA.socket,IdMsg);
27                 ...
28                Object objACK = recibeACK(pipeA.socketACK);
29
30                if(objACK != null){
31                   //Se recibio el ack
32                }else{
33                    //Timeout: No se recibio el ack del mensaje, reenvia
34                    ...
35                    if(objACK != null){
36                       //se recibio el ACK
37                        ...
38                    }else{
39                      //No se recibio el ack del mensaje, reenvia.
40                      ...
41                    }
42                }
43            }
44        }
45      }catch(Exception w){}
46    }
47 }
```

- **La clase PipeClassB.** Es la clase que recibe los mensajes provenientes de la red. Esta clase también genera dos hilos de ejecución: una para la lectura de mensajes y otra para el procesamiento de los mismos.

```
1  public class PipeClassB implements Runnable{
2
```

```
3   public void run(){
4           ...
5           while(true){
6             try{
7               ...
8               socket.setSoTimeout(...);
9               DatagramPacket dato = new DatagramPacket (...);
10              socket.receive(dato);
11              Serializador deserializa = new Serializador();
12              obj = deserializa.read(dato.getData());
13
14              //Se separa el identificador del objeto
15              ...
16              while(e.hasMoreElements()){
17                  id = e.nextElement();
18              }
19
20              idMsg = Integer.parseInt(id.toString());
21              try{
22                lastId = stackHist.peek();
23                last   = Integer.parseInt(lastId.toString());
24              }catch(Exception p){}
25
26              if((idMsg - last) > 1)
27                 continue;
28
29              if(idMsg == last){
30                  enviaACK(id,socketACK);
31              }else{
32                  stackHist.push(id);
33                  enviaACK(id,socketACK);
34                  messages.addElement(hastable.get(id));
35              }
36            }catch(Exception e){
37                socket.close();
38            }
39          }
40  }
41
42   public static void main(String[] args){
43      try{
44
45          PipeClassB pipeB = new PipeClassB();
46          new Thread(pipeB).start();
47
48          while(true){
49
50            //Se obtiene el dato proveniente del buffer.
51            Object dato =  pipeB.recibir();
52
53            if(dato != null){
54               //Aqui se envia el valor al siguiente modulo de procesamiento
```

```
55                  ModuloProcesamientoB(dato);
56              }
57            }
58        }catch(Exception w){}
59    }
60 }
```

En la figura 5.5 se muestra el diagrama de clases que resume lo descrito.

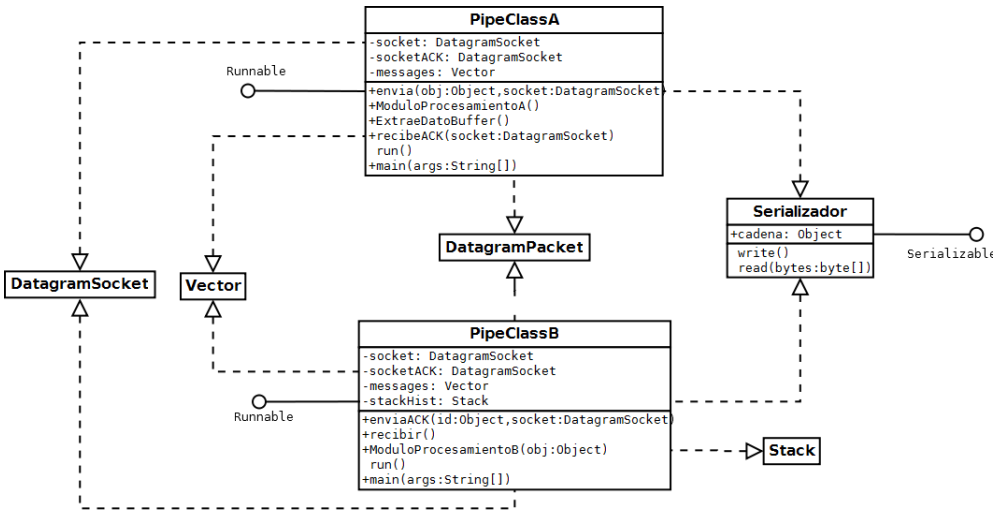

Figura 5.5: El diagrama de clases de la implementación del patrón Monitor de Paso de Mensajes Paralelo.

5.2.2. Ejecución del programa paralelo

La ocurrencia de un error en la comunicación puede presentarse de las siguientes maneras: *(a)* el mensaje es enviado al siguiente componente de comunicación, pero no llega a su destino; *(b)* el mensaje de datos o el mensaje de acuse de recibo arriban al componente de comunicación despues del timeout establecido ó *(c)* el mensaje arriba al componente de comunicación y emite el acuse de recibo, pero éste nunca llega a su destino [3]. En la figura 5.6 se muestraa gráficamente los casos mencionados.

Antes de describir a detalle la ejecución del programa bajo los errores descritos en el párrafo anterior, se muestra en la figura 5.7 la ejecución del programa paralelo sin la presencia de errores de comunicación.

La función de la clase PipeClassA es enviar un mensaje con un *id* único a la clase pipeClassB, y enseguida, esperar la recepción del acuse de recibo. En el caso de la clase

[3]Para los tres casos mencionados, se da por hecho que el componente de comunicación *sí* envía el mensaje, ya que de forma contraria se considera un error diferente, fuera del alcance de esta tesis.

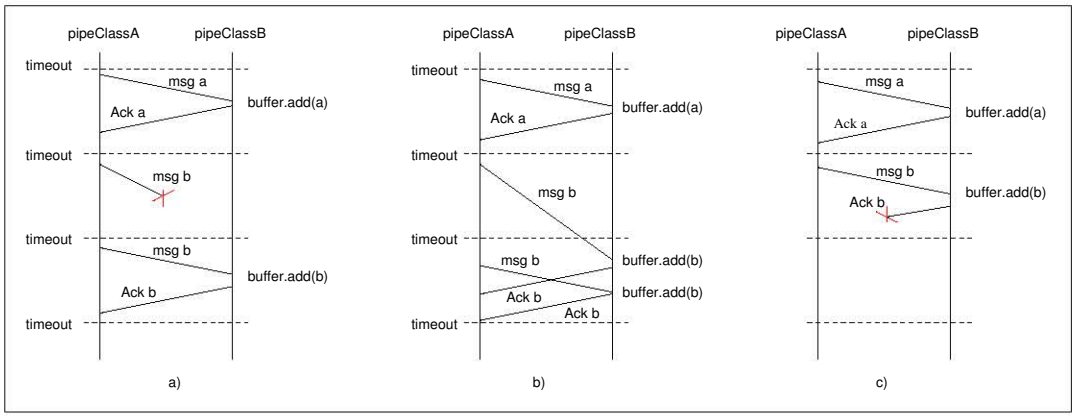

Figura 5.6: Escenarios para el envío de mensajes.

pipeClassB, su objetivo es obtener el *id* del mensaje y validar si no ha sido recibido con anterioridad. Si el id no ha sido recibido con anterioridad, se genera el acuse de recibo utilizando el mismo *id* recibido en el mensaje.

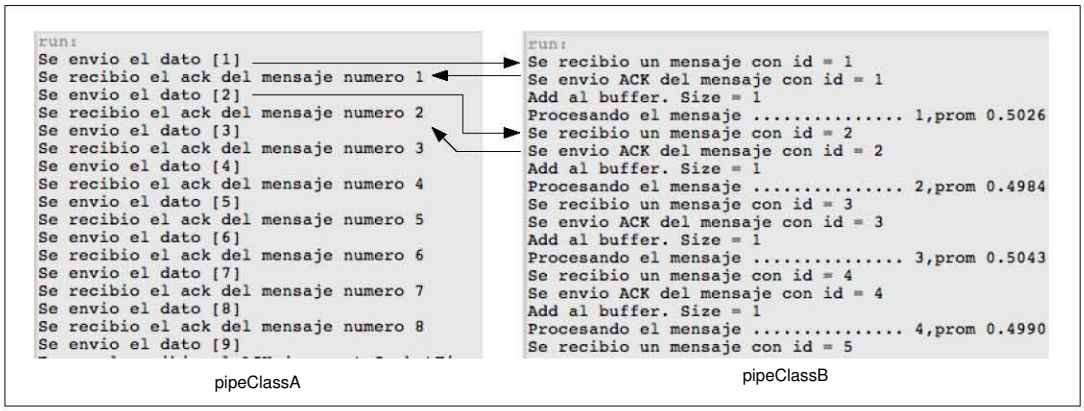

Figura 5.7: Ejecución del programa sin errores de comunicación.

El paso siguiente es mostrar la ejecución del programa paralelo ante la presencia de errores en la comunicación. Como se describe al inicio de la sección, existen tres casos:

- **El mensaje enviado por el componente de comunicación A no es recibido por el componente de comunicación B.** Para simular este error, se genera un mecanismo aleatorio que suspende la actividad del hilo encargado de la lectura del mensaje, provocando un error por timeout.

  En el código siguiente se muestra la implementación del simulador de errores:

```
1 while(true){
2
```

```
3       try{
4               ...
5               if(Math.random() >.95){
6                   Thread.sleep(1000);
7                   socket.receive(dato);
8                   continue;
9               }
10              socket.setSoTimeout(500);
11              socket.receive(dato);
12              Serializador deserializa = new Serializador();
13              obj = deserializa.read(dato.getData());
14              ....
15
16  }
```

La consecuencia de este evento es la pérdida de mensajes, por lo que el orden de los datos queda comprometido. En la figura 5.8 se muestra una ejecución de lo descrito.

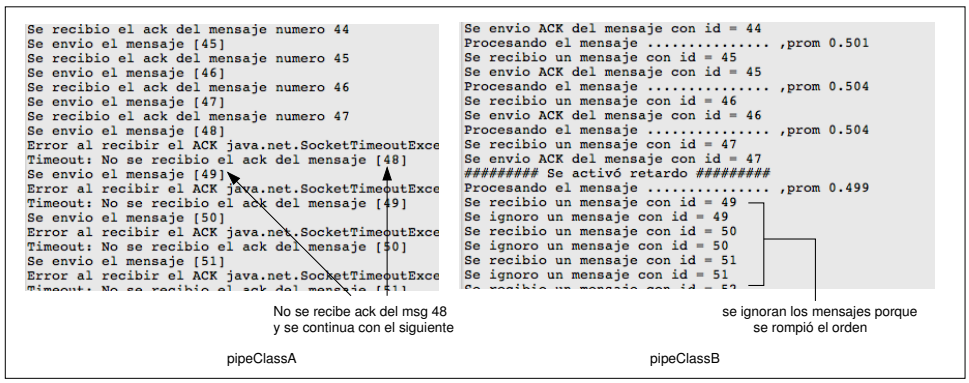

Figura 5.8: Ejecución del programa paralelo con perdida de mensaje.

En la ejecución se puede observar un comportamiento correcto hasta el mensaje número 47. El siguiente mensaje tiene un retraso mayor al timeout establecido para su recepción, por lo que no es recibido por el componente de comunicación.

Debido a que el mensaje no fue recibido, el componente continúa recibiendo mensajes, pero reconoce que falta un mensaje por lo que ignora los mensajes entrantes. El patrón especifica que en este escenario, el componente de comunicación A debe reintentar un predefinido número de veces el envío.

En la figura 5.9 se muestra la ejecución del programa integrando la solución propuesta por el patrón. Con el reenvío de mensajes es posible continuar con el funcionamiento correcto del programa paralelo. El número de reenvíos posibles es una desición de diseño que depende principalmente de la naturaleza del programa paralelo.

- **El componente de comunicacion A no recibe dentro del timeout el acuse**

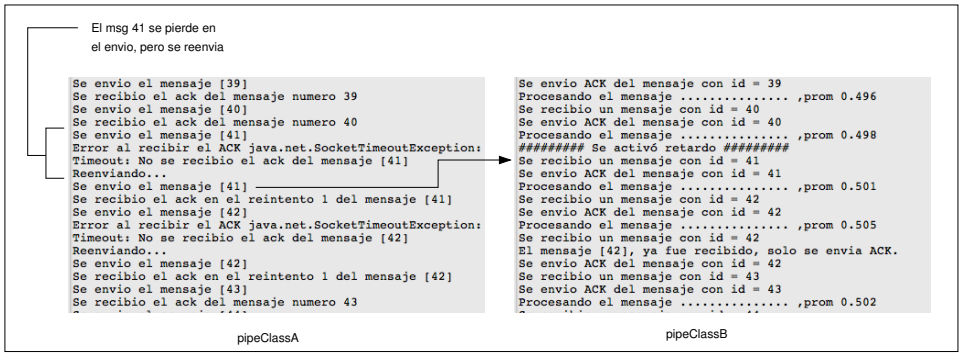

Figura 5.9: Ejecución del programa paralelo con reenvío de mensajes.

de recibo por parte del componente de comunicación B. Este segundo caso consiste en un error durante el envío del acuse de recibo por parte del componente de comunicación B. El componente A envía el mensaje y el componente B lo recibe satisfactoriamente, generando el acuse de recibo el cual es enviado a través de la red, pero no llega a su destino.

En la figura 5.10 se muestra el funcionamiento del programa paralelo ante un error de comunicación que impide que el acuse de recibo llegue a su destino.

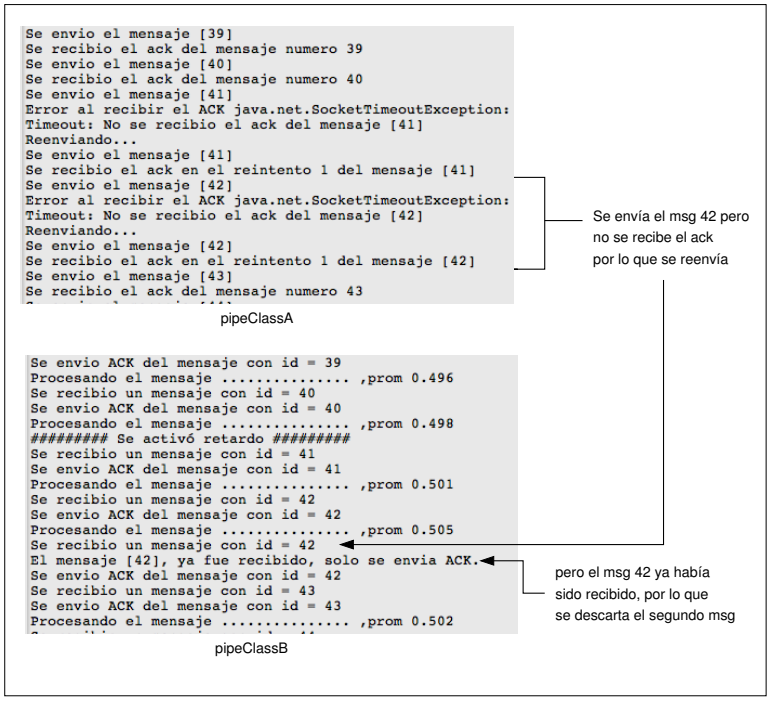

Figura 5.10: Ejecución del programa paralelo con un error en el envío del ACK.

El componente de comunicación que recibe el mensaje detecta que el mensaje recibido

ya ha sido recibido con anterioridad. La única forma de caer en este escenario es que haya habido un problema con la recepción del acuse de recibo, por lo que no almacena el valor recibido y se limita solamente a enviar nuevamente el acuse de recibo.

- **El mensaje enviado sufre un retardo en su arribo al siguiente componente, lo que provoca duplicación de datos.** Esta situación puede ocurrir debido a que la ruta que toma el mensaje para llegar a su destino no siempre es la misma. Otro factor que puede considerarse para este tipo de escenario es el tráfico en la red, en donde se realiza la implementación del programa paralelo.

  En la figura 5.11 se muestra gráficamente esta situación. El acuse de recibo sufre un retraso en su arribo al componente de comunicación, por lo que éste considera que existe un error y reenvia el mensaje.

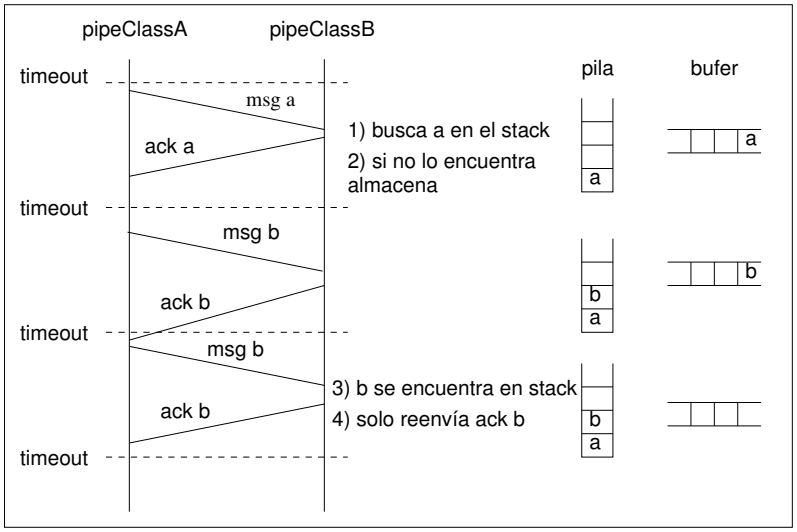

Figura 5.11: El mensaje ACK no llega a su destino y provoca reenvío de datos.

La diferencia entre el escenario anterior y éste radica en que en el escenario anterior el acuse de recibo nunca llega, y en este caso, el acuse si llega, pero a destiempo.

Para evitar este escenario de duplicación de datos, el patrón menciona como una de las fuerzas a resolver la validación de cada mensaje. El siguiente código implementa la funcionalidad que se observa en la figura 5.11.

```
1  while(true){
2       ...
3       //obtiene el Id anexo al mensaje
4       idMsg = Integer.parseInt(id.toString());
5       Object lastId = null;
6
```

```
7       try{
8          //"ve" el tope de la pila
9          lastId = stackHist.peek();
10         last  = Integer.parseInt(lastId.toString());
11      }catch(Exception p){}
12
13      if((idMsg - last) > 1){
14          System.out.println("Se ignoro un mensaje con id = " + id);
15          continue;
16       }
17       if(idMsg == last){
18          //significa que hubo un problema en la recepcion del ACK,
19          //solo enviar ACK.
20          enviaACK(id,socketACK);
21       }
22       ...
23  }
```

5.3. Evaluación del patrón Monitor de Canal de Paso de Mensajes Paralelo

El patrón Monitor de Canal de Paso de Mensajes Paralelo [Sección 4.2.3] se implementa de manera similar al patrón Monitor de Paso de Mensajes [Sección 4.2.2] con la difencia que utiliza un canal bidireccional para el paso de mensajes.

Para ilustrar la implementación del patrón, se utiliza la misma lógica del programa paralelo descrito en los patrones anteriores, con las modificaciones necesarias. En la figura 5.12 se muestra el diagrama de clases del programa paralelo.

5.3.1. Descripción de la implementación de los participantes en el patrón

La descripción de la implementación de los participantes es la siguiente:

- **Sincronización.** Este participante es idéntico al descrito en la implementación del patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 5.1].
- **El serializador.** La serialización se realiza de manera idéntica a la descrita en la implementación del patrón Paso de Mensajes Paralelo con Buffer no Delimitado [Sección 5.1].
- **Buffer de salida y recepción.** La implementación de un buffer requiere cualquier estructura capaz de almacenar datos mediante una política FIFO. En el caso del

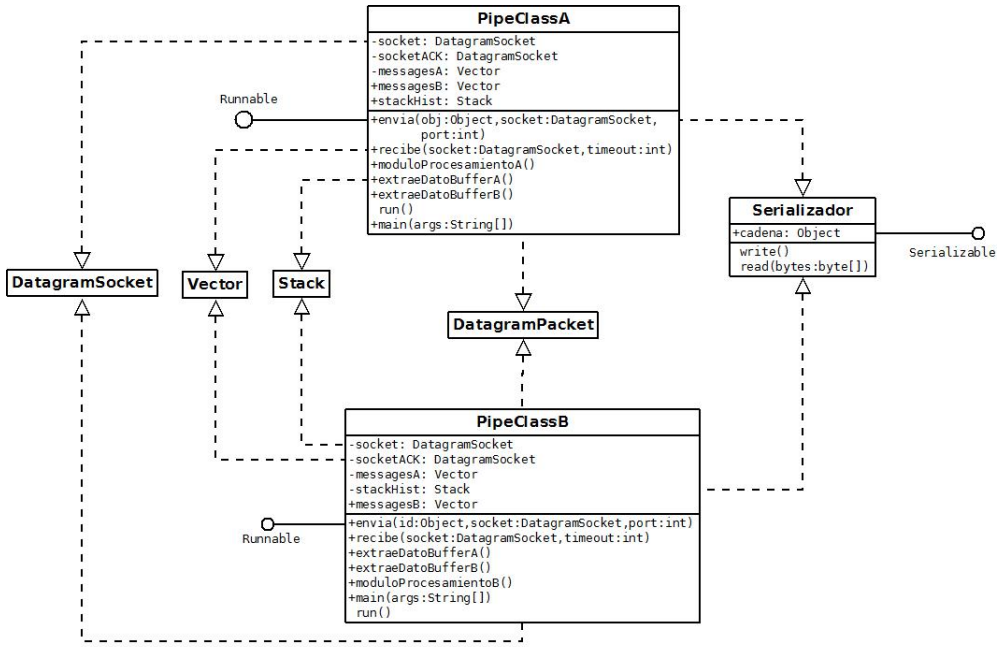

Figura 5.12: El diagrama de clases de la implementacion del patrón Monitor de Canal de Paso de Mensajes Paralelo.

lenguaje Java, se utiliza el tipo de dato Vector. Para este patrón se necesita un par de buffers en cada componente de comunicación por canal de comunicación, es decir, por cada componente de comunicación con el que intercambie datos.

- **Monitor de envío.** La implementación de este componente es similar a la implementación descrita para el monitor de envío del patrón Monitor de Paso de Mensajes Paralelo [Sección 5.2.1]: *(1)* una función encargada de extraer los objetos del buffer; *(2)* un generador de identificadores únicos para cada mensaje y *(3)* una función encargada de la recepción del mensaje.

- **Monitor de entrega.** Este participante se implementa de manera similar al monitor de entrega del patrón Monitor de Paso de Mensajes Paralelo [Sección 5.2.1]: *(1)* una funcionalidad para validar los mensajes recibidos [4] y *(2)* una funcionalidad para la generación del acuse de recibo.

Las funciones que implentan los participantes son las siguientes:

- **La clase PipeClassA.** Es el inicio del programa paralelo. La función *main* se encarga de asignar el valor inicial del primer mensaje enviado al siguiente componente

[4] Se trata de una validación en cuando a la secuencialidad de los identificadores de los mensajes recibidos, no en cuanto al contenido de los mismos.

de comunicación. Este valor es el estado inicial del componente de procesamiento, el cual necesita del estado siguiente del sistema (proveniente del componente de procesamiento con el que se está comunicando) para actualizar este valor. El código es el siguiente:

```
1  public class PipeClassA implements Runnable{
2      ...
3      public PipeClassA(){}
4
5      public void run(){
6          ...
7          while(true){
8             try{
9                ...
10               //estable el timeout
11               pipeRecibe.socket.setSoTimeout(500);
12               DatagramPacket dato = new DatagramPacket (new byte[65536], 65536);
13               pipeRecibe.socket.receive(dato);
14               Serializador deserializa = new Serializador();
15               obj = deserializa.read(dato.getData());
16
17               //Se separa el identificador del objeto
18               ...
19              hashtable    = (Hashtable) deserializa.cadena;
20
21               Enumeration e = hashtable.keys();
22               while(e.hasMoreElements()){
23                   id = e.nextElement();
24               }
25
26               idMsg = Integer.parseInt(id.toString());
27               try{
28                  lastId = stackHist.peek();
29                  last  = Integer.parseInt(lastId.toString());
30               }catch(Exception p){}
31
32               // ignora el mensaje si no es el consecutivo
33               if((idMsg - last) > 1){
34                  continue;
35               }
36
37               if(idMsg == last){
38                    ...
39                   pipeRecibe.envia(id, pipeRecibe.socketACK,5542,idMsg);
40               }else{
41                   //mantiene el historial de id's
42                   stackHist.push(id);
43                   Serializador serializa = new Serializador(id);
44                   serializa.write();
45
```

```
46                  pipeRecibe.envia(serializa.cadena,
47                                                  pipeRecibe.socketACK,5542,idMsg);
48                  messagesB.addElement(hastable.get(id));
49              }
50              //ejecuta modulo procesamientoA()
51              messagesA.add(ModuloProcesamientoA());
52            }catch(Exception e){ ... }
53          }
54      }
55
56      public static void main(String[] args){
57        try{
58            PipeClassA pipeEnvia = new PipeClassA();
59            ...
60          //crea un hilo de ejecucion para atender los mensajes
61          //del siguiente componente
62          new Thread(pipeEnvia).start();
63
64          //valor inicial del programa====
65          messagesA.add(doubleAct);
66
67          while(true){
68                ...
69          }
70        }catch(Exception w){...}
71      }
```

- **La clase PipeClassB.** Es la clase que se sitúa en el otro punto de la red. Es la encargada de calcular el nuevo valor utilizando el valor recibido por el componente de comunicación y el valor inicial del componente de procesamiento.

```
1   public PipeClassB(){}
2
3     public void run(){
4           ...
5         PipeClassB pipeRecibe = new PipeClassB();
6         ...
7          while(true){
8             try{
9                ...
10               pipeRecibe.socket.setSoTimeout(500);
11               ...
12               //finalmente al buffer
13               messagesA.addElement(hastable.get(id));
14               ...
15               messagesB.add(ModuloProcesamientoB());
16             }catch(Exception e){...}
17          }
18    }
19
```

```
20      public static void main(String[] args){
21        try{
22           int IdMsg = 0;
23           //Se crea un hilo de ejecucion para recibir mensajes
24           PipeClassB pipeEnvia = new PipeClassB();
25           new Thread(pipeEnvia).start();
26           ...
27           while(true){
28             //Se obtiene el dato proveniente del buffer.
29             datoNew =  pipeEnvia.extraeDatoBufferB();
30
31             if(datoNew != null){
32                //crea el identificador unico
33                IdMsg++;
34                Hashtable hastable = new Hashtable();
35                hastable.put(IdMsg,datoNew);
36
37                //Serializa los datos
38                Object objenv = hastable;
39                Serializador serializa = new Serializador(objenv);
40                serializa.write();
41
42                //Finalmente envia por el segundo canal
43                ...
44                pipeEnvia.envia(serializa.cadena,pipeEnvia.socket,5540,IdMsg);
45
46                //Intenta la lectura del ACK una vez enviado el mensaje
47                ...
48                Object objACK = recibe("Canal 2 [PipeB] (ACK)",
49                                       pipeEnvia.socketACK,timeout);
50
51                if(objACK != null){
52                   ...
53                }else{
54
55                   ...
56                   pipeEnvia.envia(serializa.cadena, pipeEnvia.socket,5540,IdMsg);
57                   objACK = recibe(pipeEnvia.socketACK,timeout);
58
59                   if(objACK != null){
60                      ...
61                   }else{
62                      ...
63                   }
64                   ...
65   }
```

En este patrón se utilizan dos pares de monitores envío-entrega para establecer un canal bidireccional de envío de datos, cada uno de ellos ejecutándose en un hilo.

5.3.2. Ejecución del programa paralelo

La funcionalidad del programa paralelo consiste en enviar y recibir valores mediante un canal de comunicación bidireccional. En este caso, los módulos de procesamiento obtienen el valor recibido, lo disminuyen en una unidad y asignan ese valor como su estado actual que es enviado de regreso. Sin embargo, los componentes de procesamiento pueden ser sobrescritos para resolver cualquier problema que necesite una comunicación bidireccional.

El programa consiste en cuatro hilos de ejecución: dos para el componente de comunicación A (pipeClassA) y dos para el componente de comunicación B (pipeClassB). Cada uno de ellos utiliza un buffer para garantizar la asincronía del envío y recepción de mensajes.

El hilo encargado del envío de mensajes del componente de comunicación A obtiene el valor inicial del buffer de envío, y procede a su envío a través de la red. El hilo encargado de la lectura de mensajes del componente de comunicación B obtiene el valor y lo almacena en el buffer de recepción.

El componente de procesamiento B extrae el dato del buffer de recepción, realiza los cálculos necesarios, para inmediatamente despues insertar el resultado en el buffer de envío. El hilo encargado del envío de mensajes del componente de comunicación B detecta que hay un valor en su buffer, por lo que procede a enviar el mensaje.

En el diagrama de flujo que se muestra en la figura 5.13 se puede ver gráficamente lo que ocurre en el componente de comunicación implementado por la clase pipeClassA.

La ejecución del programa paralelo sin la presencia de un error en la comunicación se muestra en la figura 5.14.

La aparición de un error en la comunicación puede ocurrir en cualquiera de los canales implementados. En esta implementación se trata de un solo canal bidireccional que consiste en 4 puntos finales de comunicación, dos para el envío de mensajes y dos para la recepción de los mismos.

El mecanismo de correción de errores funciona de manera similar a la mostrada en la implementación del patrón Monitor de Paso de Mensajes Paralelo [Sección 5.2]: se implementan para cada uno de los puntos de salida un timeout acorde a las necesidades de diseño de cada programa paralelo.

En la figura 5.15 se muestra la ejecución del programa con la simulación de un error en la comunicación al momento de enviar el acuse de recibo de un mensaje.

De la figura 5.15 se pueden hacer las siguientes observaciones:

- El retardo en el envío del acuse de recibo provoca un error de timeout en la clase pipeClassA, por lo que se realiza un reenvío del mensaje.
- El mensaje es recibido por la clase pipeClassB, pero como ya habia sido recibido, solo envía el acuse de recibo y se ignora el mensaje.

- La clase pipeClassA solo tiene conocimiento de la entrega correcta hasta el segundo reintento de envío.
- En tanto que no haya una certeza en la entrega el mensaje, el canal 2 queda en espera de la resolución de mensajes del canal 1.

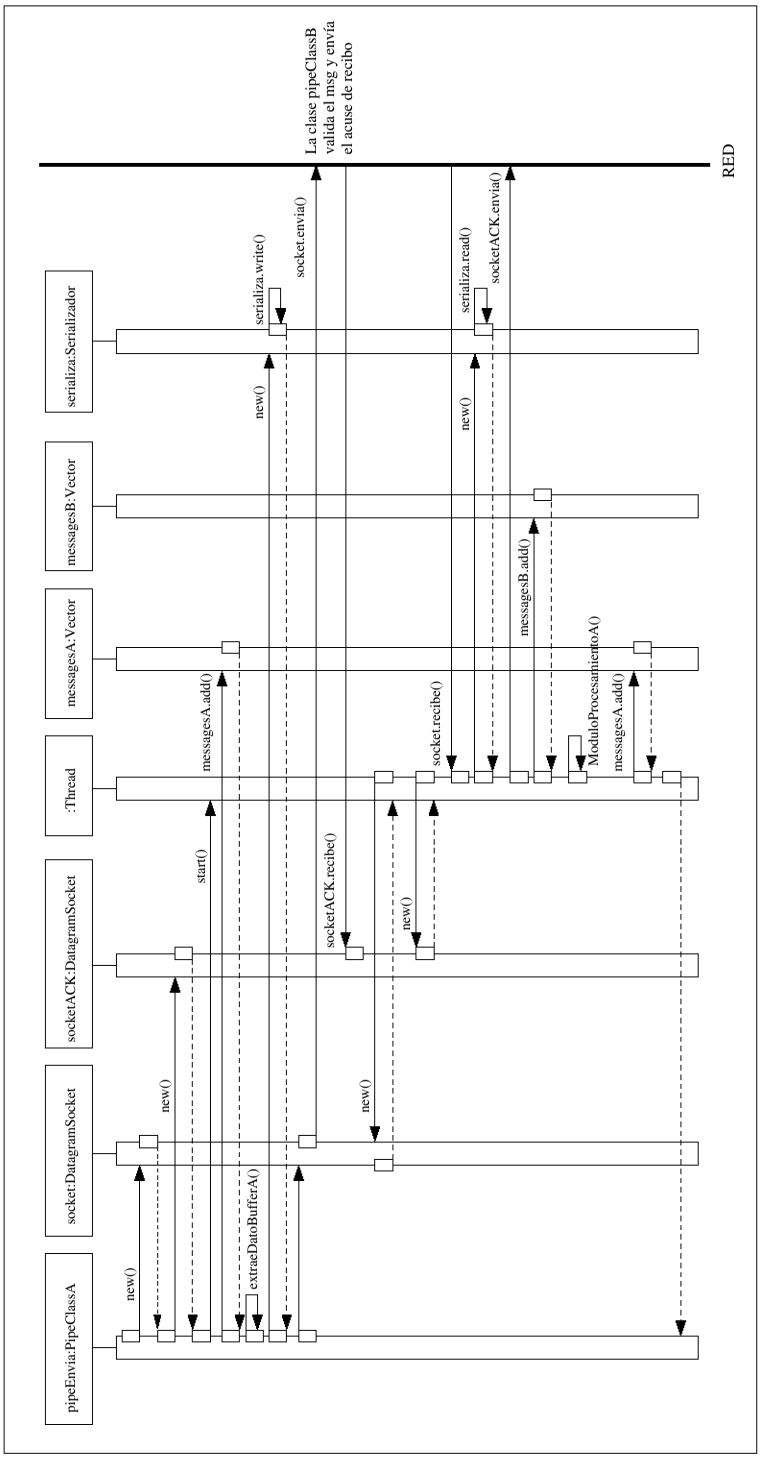

Figura 5.13: Diagrama de flujo de la clase pipeClassA.

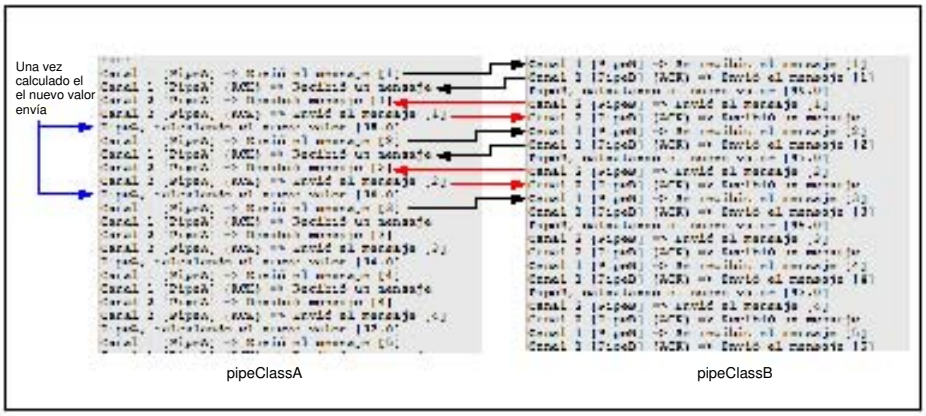

Figura 5.14: Ejecución del programa sin errores.

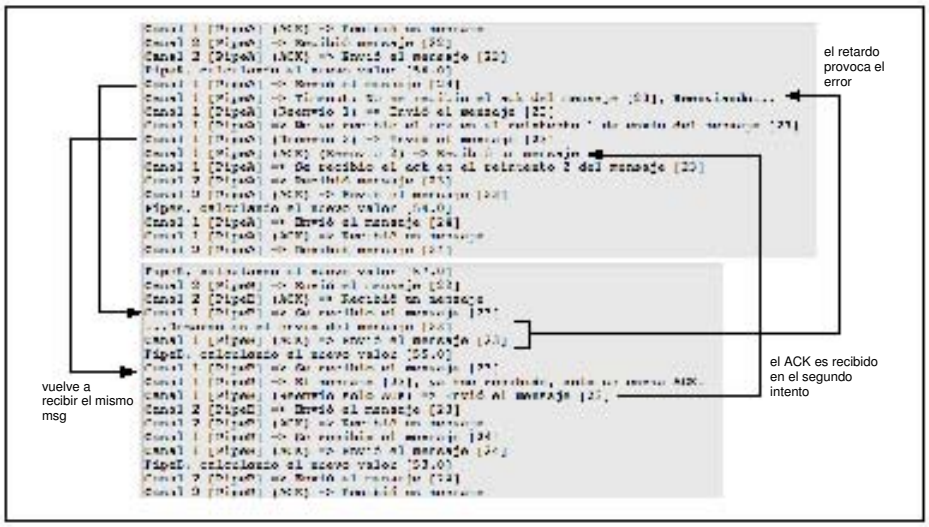

Figura 5.15: Ejecución del programa paralelo con un error en el envío del mensaje.

5.3.3. Resumen

Las evaluaciones experimentales desarrolladas en éste capítulo muestran la funcionalidad de las soluciones propuestas por los patrones de diseño. En cada uno de los programas desarrollados, se describe a detalle todos y cada uno de los participantes y la forma cómo interactúan entre ellos.

Es preciso mencionar que los programas presentados no son la única o la mejor forma de implementar los patrones propuestos. De hecho, es posible utilizar cualquier otro lenguaje orientado a objetos que provea diferentes herramientas de sincronización o de implementación de sockets, solo por mencionar dos ejemplos.

Debido a que el objetivo de este trabajo se enfoca solo en la parte de la comunicación entre componentes de procesamiento, no es de importancia el procesamiento que éstos componentes realicen, por lo que su funcionalidad de reduce a una cuantas operaciones sencillas. Esta decisión se basa en que el objetivo de la presente tesis no es resolver un problema en específico de cómputo paralelo, sino hacer incapié en la manera en que se realiza la comunicación dentro del contexto de la programación paralela en la que pueden presentarse errores durante la misma.

Tambien hay que destacar que esta evaluación experimental no está enfocada en realizar un análisis respecto al tiempo de ejecución de un programa paralelo. La evaluación presentada se limita únicamente a mostrar la forma en que las soluciones propuestas funcionan evitando posibles fallas que provoquen una comunicación incorrecta.

En el caso del patrón Paso de Mensajes Paralelo con Buffer no Delimitado, es puede observar que el programa puede seguir funcionando a pesar de que ya no sea posible asignar memoria al buffer, esto sin interrumpir en ningún momento el funcionamiento de los componentes de procesamiento. En el caso de los patrones Monitor de Paso de Mensajes Paralelo y Monitor de Canal de Paso de Mensajes Paralelo, es posible corregir una comunicación que puede llegar a ser fallida por la perdida de un mensaje mediante el reenvío del mismo, sin afectar de igual manera el funcionamiento de los componentes de procesamiento.

En el capítulo siguiente se presentan las conclusiones al presente trabajo, las cuales describen las ventajas y desventajas que presentan los patrones de software propuestos, así como el trabajo futuro en el tema.

Capítulo 6

Conclusiones

Aún siendo verdad que yo haya errado,
Sobre mí recaería mi error.
Libro de Job 19:4

En este capítulo se presenta: *(1)* las conclusiones del trabajo de investigación realizado en esta tesis, partiendo de la hipótesis enunciada; *(2)* las contribución del trabajo al área de estudio y *(3)* el trabajo futuro en esta área de investigación.

6.1. Resultados del trabajo de investigación

El objetivo principal de este trabajo es proveer mecanismos de prevención y corrección de errores al diseño de componentes de comunicación de programas paralelos e integrarlos en un patrón de diseño. Esta idea se refleja en la hipótesis descrita en el capítulo 1 y que de forma textual dice lo siguiente:

"¿Es posible especificar mediante patrones de software, las acciones necesarias a realizar ante la presencia de errores en la comunicación entre componentes de procesamiento de un programa paralelo que utiliza una organización de memoria distribuida, con el fin de asegurar la comunicación entre los componentes implementados para la comunicación?"

Este trabajo demuestra que es posible implementar mecanismos de prevención y corrección de errores. Sin embargo, es necesario puntualizar las virtudes y defectos encontrados en estas soluciones, de manera que sea posible tener una mejor valoración de las mismas.

6.1.1. Conclusiones que reflejan sus virtudes

A continuación se describen las conclusiones que reflejan sus virtudes:

- **Los patrones de diseño propuestos resuelven un fenómeno recurrente.** El error de asignación de memoria en un buffer y la perdida-retardo de mensajes que viajan a través de una red de comunicación, son fenómenos recurrentes en los sistemas de cómputo. Los patrones descritos en esta tesis proponen una solución efectiva a estos problemas en el contexto de la programación paralela y una arquitectura distribuida.
- **Los patrones de diseño propuestos no rompen con la dinámica de los patrones de diseño en los cuales están basados.** Esto se debe a que las soluciones consisten en la modificación interna de los participantes o en la replicación de los mismos, y no en la estructura y dinámica general de los patrones.
- **Los patrones encapsulan y abstraen un problema bien definido.** Esto se logra gracias al planteamiento de una clase de fallas sobre la cual los patrones ofrecen una solución. De no ser así, las soluciones propuestas no pueden ser consideradas patrones, ya que el problema a resolver no esta bien definido o delimitado.
- **Los patrones ofrecen apertura y viabilidad.** Los patrones pueden implementarse de distintas maneras y conforme a las necesidades del diseñador. Por ejemplo:
  - En el caso de la validación de mensajes por parte del receptor, se utiliza un objeto de la clase Stack para comprobar el orden de los identificadores. En este caso, el diseñador o programador puede utilizar cualquier otro mecanismo que le parezca más eficiente o más sencillo de utilizar.
  - En el caso del monitor del buffer, también es posible implementar los buffers utilizando otro tipo de datos o tipos de objetos, siempre y cuando mantengan la política FIFO.
  - Es posible integrar el mecanismo propuesto en el patrón Paso de Mensajes Paralelo con Buffer no Delimitado en los otros dos patrones, con solo modificar la implementación de los buffers.
  - Es posible utilizar un mecanismo de sincronización que ofrezca el lenguaje mismo de programación o la construcción de uno, por ejemplo semaforo o regiónes crítica. En la implemetación realizada en esta tesis, es utiliza la sincronización basada en monitores que ofrece el lenguaje Java, a través del uso de métodos *synchronized*.

6.1.2. Conclusiones que reflejan sus defectos.

A continuación se describen las conclusiones que reflejan sus defectos:

- **El cálculo del tiempo de espera de un acuse de recibo no es tarea trivial.** En el caso del patrón Monitor de Paso de Mensajes Paralelo y el patrón Monitor de Canal de Paso de Manejes Paralelo, es necesario establecer un tiempo de espera durante el cual el componente de comunicación espera recibir el acuse de recibo. El cálculo de este tiempo es la columna vertebral de la solución ofrecida por estos patrones, por lo que es necesario ser muy cuidadoso al respecto.

- **Se duplica el número de mensajes.** En el caso del patrón Monitor de Paso de Mensajes Paralelo y el patrón Monitor de Canal de Paso de Manejes Paralelo, el número de mensajes se duplica con respecto al número de mensajes sin el mecanismo de corrección de errores. Este problema se agrava en los casos en que se utiliza una red muy saturada, o la granularidad del programa paralelo es fina.

- **El buffer auxiliar puede tambien llegar al límite de su capacidad.** En el caso del patrón Paso de Mensajes Paralelo con Buffer no Delimitado, la implementación de un buffer auxiliar que evite la pérdida de mensajes y permita la continuación del procesamiento puede llegar al límite de su capacidad también. Aunque este caso puede resolverse agregando un tercer buffer al componente de comunicación, la intuición de la necesidad de una refactoriazación del programa paralelo se hace más evidente.

6.2. Trabajo futuro

El trabajo futuro en el tema que esta tesis aborda se describe en los siguientes puntos:

- **Una análisis comparativo del desempeño de las comunicaciones con y sin los mecanismos de prevención y corrección de errores propuestos.** En base a este análisis es posible ayudar al diseñador o programador del sistema a tomar una decisión respecto a la implementación los patrones propuestos en esta tesis, basado en un análisis de costo-beneficio.

- **Un análisis que permita seleccionar el tiempo de espera tomando en cuenta la granularidad del programa y la topología de la red en la que se implementa.** Si la necesidades del diseño del sistema paralelo son acordes al contexto y problema que mencionan los patrones de diseño, este análisis es de utilidad en la elección de un timeout apropiado para implementar el sistema.

- **Integrar estos patrones con otros basados en replicación.** En el caso de que los mecanismos de prevención y correción no sean suficientes para corregir un error, es posible integrar aspectos de replicación y extender la capacidad de prevención y correción de errores.

Bibliografía

[1] Jorge L. Ortega Arjona. Design patterns for communication components of parallel programs. *12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)*, 2007.

[2] Gopinatha Jakadeesan and Dhrubajyoti Goswami. A classification-based approach to fault-tolerance support in parallel programs. *Parallel and Distributed Computing Applications and Technologies, International Conference on*, 0:255–262, 2009.

[3] Cherri M. Pancake. Is parallelism for you? *IEEE Comput. Sci. Eng.*, 3:18–37, June 1996.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[5] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[6] Per Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15:574–578, July 1972.

[7] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[8] Per Brinch Hansen. Distributed processes: a concurrent programming concept. *Commun. ACM*, 21:934–941, November 1978.

[9] S. M. Shatz. Communication mechanisms for programming distributed systems. *Computer*, 17:21–28, June 1984.

[10] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[11] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34:56–78, February 1991.

[12] T. Rauber and G. Runger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.

[13] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[14] R. Monson-Haefel and D.A. Chappell. *Java Message Service*. Java Series. O'Reilly, 2009.

[15] Drasko Sotirovski. Towards fault-tolerant software architectures. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '01, pages 7–, Washington, DC, USA, 2001. IEEE Computer Society.

[16] N. Santoro. *Design and analysis of distributed algorithms*. Wiley series on parallel and distributed computing. Wiley-Interscience, 2007.

[17] A.D. Kshemkalyani and M. Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.

[18] Felix C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31:1–26, March 1999.

[19] A. Arora and S Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *In Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, 1998.

[20] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[21] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, SE-3 Issue:2:125 – 143, March 1977.

[22] Sape Mullender, editor. *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[23] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Morgan and Claypool Publishers, 2009.

[24] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[25] J.L. Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley Software Patterns Series. John Wiley & Sons, 2010.

[26] Albert Y. H. Zomaya, editor. *Parallel and distributed computing handbook*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[27] Stephen Stelting and Olav Maassen. *Patrones de diseño aplicados a java*. Prentice Hall, 2003.

[28] C. Alexander. *The Timeless Way of Building*. Center for Environmental Structure Series. Oxford University Press, 1979.

[29] Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development. *Theor. Pract. Object Syst.*, 2(1):3–13, November 1996.

[30] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[32] Titos Saridakis. A system of patterns for fault tolerance. *Originally published in Proceedings of the* 7^{th} *European Conference on Pattern Languages of Programs*, 2002.

[33] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[34] Jorge L. Ortega-Arjona. The pipes and filters pattern. a functional parallelism architectural pattern for parallel programming. *Proceedings of the* 10^{th} *European Conference On Pattern Languages of Programming and Computing, EuroPLoP*, 2005.

[35] Jorge L. Ortega-Arjona. The communicating sequential elements pattern. an architectural pattern for domain parallelism. *Proceedings of the* 7^{th} *Conference on Pattern Languages of Programming, PLoP 2000*, 2000.

[36] Michael Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *CoRR*, abs/cs/0501002, 2005.

[37] Jack Shirazi. Catching outofmemoryerrors to preserve monitoring and server processes.

[38] S. Prata. *C++ Primer Plus*. Developer's Library. Addison-Wesley, 2011.

[39] S. Hartley. *Concurrent Programming: The Java Programming Languaje*. Oxford University Press, Inc., 1998.

[40] S. Oaks and H. Wong. *Java threads*. The Java series. O'Reilly, 2004.

[41] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency, Practice and Experience*, 5(5):425–448, 1993.

[42] http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html Oracle Corporation. The java tutorial.

[43] C. Hunt, P. Hohensee, and B. John. *Java Performance*. Java Series. Prentice Hall, 2011.