ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

SISTEMA DE COMUNICACIÓN FIABLE CON CAÍDAS Y RECUPERACIONES DE EQUIPOS CON UNA SEGURIDAD MEDIANTE IDENTIFICACIÓN ANÓNIMA

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y REDES DE INFORMACIÓN

CRISTIAN OSWALDO RODRÍGUEZ SANTIAGO
cristian.rodriguez02@epn.edu.ec

DIRECTOR: JOSÉ ERNESTO JIMÉNEZ MERINO, PhD.
ernes@eui.upm.es

CODIRECTOR: ING. CARLOS ALFONSO HERRERA MUÑOZ, MSc.
carlos.herrera@epn.edu.ec

Quito, junio 2017

DECLARACIÓN

Yo, Cristian Oswaldo Rodríguez Santiago, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

_____

Cristian Oswaldo Rodríguez Santiago

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Cristian Oswaldo Rodríguez Santiago, bajo mi supervisión.

_____

José Ernesto Jiménez Merino, PhD.
DIRECTOR DEL TRABAJO DE TITULACIÓN

_____

Ing. Carlos Alfonso Herrera Muñoz, MSc.
CODIRECTOR DEL TRABAJO DE TITULACIÓN

AGRADECIMIENTOS

A Dios, por su guía, protección y fortaleza durante mi formación profesional.

A mi familia, especialmente a mi madre Adelina, por su paciencia, por su amor, y sus palabras de aliento en los momentos más difíciles.

A mi director, PhD. Ernesto Jiménez, por su paciencia, apoyo y dirección en el desarrollo de este trabajo.

A mis profesores, por impartir su conocimiento y ser parte vital de mi formación académica.

DEDICATORIA

A mi familia.

Cristian Rodriguez

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGO

RESUMEN

Los sistemas distribuidos tolerantes a fallos toman en cuenta algunos paradigmas imprescindibles para el desarrollo de múltiples aplicaciones. Uno de ellos es Consenso, el cual permite a un conjunto de procesos decidir un mismo valor en común a partir de sus valores iniciales. En la literatura se ha demostrado que Consenso no puede ser resuelto en sistemas asíncronos puros con caídas de procesos. Una forma de resolver este problema es hacer uso de Detectores de Fallos. Un Detector de Fallos es un dispositivo (en la mayoría de los casos software) que permite detectar equipos o procesos fallidos. Existen distintas clases como: P, ◊P, ◊S, entre otras. Cada una de estas clases se define en función de las características de completitud y precisión que presentan los distintos Detectores de Fallos. Estas clases permiten elegir la mejor opción en función del problema a resolver (en el caso de este trabajo es Consenso).

Los sistemas distribuidos anónimos ocultan la identidad de los involucrados en la comunicación. Sobre este tipo de sistema anónimo se realiza en este trabajo el diseño e implementación de un algoritmo de Detector de Fallos y otro de Consenso. Estos dos algoritmos diseñados van a permitir que el sistema anónimo resultante tenga ciertas propiedades como son: dinamismo y Caída-recuperación de equipos (procesos). El sistema operativo que se utiliza en este trabajo es FreeBSD y el lenguaje de implementación es C++.

PRESENTACIÓN

El presente trabajo tiene como objetivo la especificación, diseño e implementación de algoritmos de Detector de Fallos y Consenso.

En el primer capítulo se realiza un estudio del estado del arte, revisando los trabajos más importantes en la literatura. Primero se introduce el tema con el estudio de Detectores de Fallos Clásicos, su definición, características, clases e implementación. Seguidamente se aborda el tema relacionado con Detectores de Fallos Anónimos, su definición, clases, e implementación.

El segundo capítulo especifica el modelo del sistema sobre el cual se implementan los algoritmos. Además, se presenta los detalles de especificación, diseño e implementación de un nuevo algoritmo para detectar caídas de equipos (procesos) en sistemas anónimos. Este algoritmo tiene dos características: dinamismo y recuperación de equipos que han fallado.

En el tercer capítulo se presenta los detalles de especificación, diseño e implementación de un nuevo algoritmo de Consenso.

El cuarto capítulo presenta una aplicación que demuestra el funcionamiento de ambos algoritmos.

El quinto capítulo se muestra las pruebas del código implementado de cada uno de los algoritmos. Adicionalmente en el sexto capítulo se encuentran las conclusiones y recomendaciones seguidamente de los Anexos.

CAPÍTULO 1
ESTUDIO DEL ESTADO DEL ARTE

1.1. INTRODUCCIÓN

Un sistema distribuido es un conjunto de procesos que intercambian mensajes a través de enlaces de comunicaciones. Estos sistemas pueden ser síncronos, asíncronos o parcialmente síncronos. Un sistema distribuido síncrono define límites de tiempo conocidos, tanto para la ejecución de procesos como para el retardo de transmisión de los mensajes. En cambio, un sistema distribuido asíncrono no tiene restricciones de tiempo, ni de ejecución ni de retardo [1]. Por otro lado, un sistema parcialmente síncrono define, como el sistema síncrono, unos límites de tiempo, pero estos son desconocidos para los procesos que comparten el sistema [2]. Los servicios que ofrece el sistema distribuido se definen por una serie de operaciones conocidas como primitivas que, al ponerlas a disposición de un proceso, le permiten a este poder acceder al servicio.

Un fallo es una anomalía en la ejecución del proceso que repercute en la especificación para la cual fue programado. Debido a la importancia de un posible fallo, estos pueden clasificarse como: Bizantinos, por Omisión, Caída-parada (*crash-stop*), y Caída-recuperación (*crash-recovery*). Los fallos Bizantinos consisten en errores arbitrarios que se pueden dar en cualquier lugar de la implementación. El fallo de tipo Omisión se da cuando un proceso no envía o recibe los mensajes que se supone que debe enviar o recibir, en la práctica esto se da por desbordamiento de *buffers*, congestión, o pérdida de paquetes en la red. En el tipo Caída-parada, un proceso que falla deja de ejecutarse permanentemente y, por lo tanto, no se recupera a partir del fallo. Por último, en el tipo de fallos Caída-recuperación, el proceso que se detiene por sufrir un fallo, después de cierto tiempo, puede recuperarse y volver a ejecutarse con normalidad [3].

En un sistema distribuido tolerante a fallos con Caída-recuperación, un proceso correcto es aquel que nunca ha fallado o que, si ha fallado, al final se ha recuperado. Si el proceso está continuamente cayéndose y recuperándose no se le considera como correcto, si no como inestable.

Los enlaces de comunicación de un sistema distribuido se abstraen conceptualmente para representar a los componentes de la red. Debido a los fallos que pueden introducir estos componentes, se han modelado distintitos tipos de enlaces para describir su comportamiento al trasportar los mensajes. De todos los tipos existentes, se van a definir dos de ellos que se van a emplear ampliamente en el presente trabajo. Estos son: enlace fiable (*reliable link*) y enlace eventualmente fiable (*eventually timely*). También existe otro tipo de enlaces que resultan interesantes para ciertas soluciones, aunque no han sido utilizados en el presente trabajo, pero que se han empleado como ejemplo de trabajos existentes en la literatura para resolver ciertos problemas relacionados con el tema de este trabajo. Este tipo es: enlace con pérdida razonable (*fair-lossy*). Por una parte, un enlace fiable tiene un retardo acotado en los mensajes enviados, no introduce errores y no pierde el mensaje. En cambio, un enlace eventualmente fiable tiene un tiempo de estabilización previo antes de comportarse como enlace fiable, durante este tiempo de estabilización los mensajes transmitidos pueden perderse. Por último, un enlace con pérdida razonable puede extraviar mensajes, pero si el emisor continúa retransmitiendo a su receptor de forma permanente, tarde o temprano recibirá al menos una vez el mensaje [3].

Las aplicaciones (procesos) normalmente utilizan varios hilos para separar sus tareas. Estos hilos pueden compartir recursos físicos como, por ejemplo: procesador, memoria, interfaces o recursos lógicos como son las variables en código, con otros hilos del mismo proceso.

Para controlar la concurrencia a estos recursos se emplean algunos mecanismos como: Mutex (*Mutual exclusion*) y Semáforos. Cada uno de estos mecanismos definen una región critica del código, la cual consiste del recurso que es compartido y puede ser modificado por los hilos.

Mutex controla la concurrencia bloqueando el recurso a utilizar y solo puede ser liberado por el hilo que bloqueó el recurso. Un Semáforo define una capacidad de acceso al recurso y puede ser liberado por cualquier hilo; una vez que exista un espacio disponible. Esto se utiliza principalmente para limitar concurrencia [1].

1.2. DETECTOR DE FALLOS CLÁSICOS

1.2.1. DEFINICIÓN

Un Detector de Fallos es un dispositivo (software o hardware) que informa al sistema de los procesos que tienen falla. Los principales problemas que se pueden resolver con un Detector de Fallos son: elección de líder [4], difusión genérica (*generic broadcast*) [5], difusión atómica (*atomic multicast*) [6] y el que concierne al presente trabajo, Consenso [7].

Consenso se demostró en [8] que no se puede resolver en un sistema asíncrono en donde un proceso puede fallar. Para soslayar esta imposibilidad, una de las posibles alternativas para resolver Consenso es utilizar un Detector de Fallos además del sistema asíncrono [2].

1.2.2. CARACTERÍSTICAS

El concepto de Detector de Fallos no fiable fue introducido por primera vez por Chandra y Toueg [2]. El término no fiable se refiere a que el Detector de Fallos puede cometer inexactitudes señalando a un proceso correcto como proceso fallido. Un Detector de Fallos se caracteriza por dos propiedades: completitud (*completeness*) y precisión (*accuracy*).

Completitud es la propiedad que permite la detección de todos los procesos que efectivamente han fallado, mientras que la precisión restringe las inexactitudes que el Detector de Fallos puede cometer.

1.2.3. DEFINICIÓN DE CLASES

Chandra y Toueg propusieron en [2] distintos tipos de Detectores de Fallos en función de las propiedades de completitud y precisión. A cada uno de estos tipos de Detectores de Fallos se les denominó clase.

Para realizar esta distinción, Chandra y Toueg introdujeron dos valores de completitud (fuerte y débil) y cuatro valores de precisión (fuerte, débil, tarde o temprano fuerte, y tarde o temprano débil). Las ocho clases de Detectores de

fallos, resultantes de la combinación de estos seis tipos de valores en las propiedades, son las que aparecen en la Tabla 1.1. El término tarde o temprano (del inglés *eventual*) en las clases, hace referencia a que existe un tiempo finito en el cual el Detector de Fallos comenzará a cumplir la propiedad de precisión especificada.

Tabla 1.1. Clases de Detectores de Fallo

		Precisión			
		Fuerte	Débil	Tarde o temprano fuerte	Tarde o temprano débil
Completitud	Fuerte	P	S	◊P	◊S
	Débil	Q	W	◊Q	◊W

Una completitud fuerte implica que todos los procesos correctos tarde o temprano detectarán a todos los procesos fallidos. Por otro lado, una completitud débil implica que algún proceso correcto tarde o temprano detectará a todos los procesos fallidos.

En un Detector de Fallos con precisión fuerte ningún proceso puede ser considerado como fallido mientras se está ejecutando correctamente. En otras palabras, para que un proceso sea considerado fallido este debe haber dejado previamente de funcionar. En los Detectores de Fallos con precisión débil se pueden cometer inexactitudes a la hora de detectar procesos fallidos, siempre que exista la limitación de que hay al menos un proceso correcto al que no se le detectará nunca como fallido.

La Tabla 1.1 muestra que un de Fallos perfecto, denotado como P, cumple con una completitud fuerte y precisión fuerte, es decir, no comete falsas sospechas y detectará toda falla. Generalmente para poder asumir que existan Detectores de Fallos perfectos hace falta que el sistema que los implemente tenga límites de sincronismo estrictamente definidos (es decir, que sea un sistema síncrono). Por esta razón, se definen clases de Detectores con características que tarde o temprano serán fiables para sistemas parcialmente síncronos que no tienen estas restricciones de tiempo.

La motivación principal de las clases de los Detectores de Fallos radica en encontrar el más débil capaz de resolver un problema. Esto significa que, si el problema se soluciona con el más débil, podrá exitosamente ser resuelto con uno de características más fuertes. Esto se conoce como relaciones de equivalencia entre los Detectores de Fallos.

La clase del Detector de Fallos Omega (también llamada Ω) fue introducida como la clase más débil para resolver Consenso en sistemas asíncronos con el tipo de fallos de Caída-parada [9]. La particularidad de este Detector de Fallos es que tarde o temprano da a conocer la identidad de un proceso correcto llamado líder, en el que todos los demás procesos pueden confiar.

1.2.4. IMPLEMETACIONES DE Ω

De todas las clases de Detectores de Fallos presentadas en el apartado anterior, el presente trabajo se centra en la clase del Detector de Fallos Omega, ya que es la clase más débil para resolver el problema de Consenso [9].

Los trabajos más relevantes existentes en la literatura relacionados con Ω son: [10], [11], [12] y [13]. En [10] el autor propone un algoritmo que implementa Ω utilizando una cantidad de mensajes eficiente. El algoritmo requiere conocer la identidad de los procesos (membresía), no se pueden añadir nuevos procesos una vez iniciado el sistema (membresía estática), y con tipo de fallo Caída-parada.

En [11] y [12] se definen algoritmos de Ω que no requieren conocer la cantidad de procesos que van participar en el sistema, sin embargo, la membresía es estática y los procesos fallan según el modelo del tipo Caída-parada. La diferencia de ambos trabajos es que [11] considera enlaces fiables en la comunicación y [12] asume enlaces que pueden introducir errores (*fair-lossy*).

Estos dos artículos ([11] y [12]) son considerados seminales, debido a que fueron los primeros en proponer Detectores de Fallos que desconocen a priori la cantidad de procesos. Los Detectores de Fallos no solo se han investigado para

tipos de falla Caída-parada, sino que también en [13], por ejemplo, se explica que no se puede implementar Detectores de Fallos con procesos que fallan según el tipo Caída-recuperación, sin utilizar un medio de almacenamiento estable. También en [13] define un algoritmo que resuelve Consenso asumiendo un tipo de fallo Caída-recuperación, incluso si el número de procesos correctos es menor a la mitad de todos los procesos. El modelo del sistema consiste de enlaces confiables y membresía estática.

1.3. DETECTOR DE FALLOS ANÓNIMOS

1.3.1. MOTIVACIÓN PARA ANONIMATO

En muchos sistemas distribuidos es importante mantener a salvo la información que se manipula, para mantener la seguridad y privacidad de los usuarios, ya sean estos personas o procesos. Bajo este contexto, la seguridad se refiere a los mecanismos utilizados para proteger a la red y sus servicios de: acceso no autorizado, modificación, destrucción y divulgación de información. La privacidad es un apartado de la seguridad que se enfoca principalmente en el acceso no autorizado y en mantener en secreto a: usuarios, aplicaciones, y dispositivos [14].

La privacidad involucra dos aspectos: el primero es la privacidad de los datos. Esto consiste en proteger la información sensible del usuario durante la comunicación o durante el almacenamiento. Por ejemplo, una contraseña transmitida sobre SSL (*Secure Socket Layer*). El segundo aspecto consiste en la protección de la identidad de los usuarios. Por lo tanto, la seguridad en redes de comunicaciones involucra tanto la protección del contenido de un mensaje, como ocultar la identidad de quienes intervienen en la comunicación.

Un mensaje puede ser protegido con técnicas de codificación para que solo los destinatarios autorizados puedan acceder a la información. Un sistema anónimo oculta la identidad de los involucrados en la comunicación, mediante procesos y algoritmos que no necesitan de identificadores para ejecutar sus instrucciones. La principal motivación para el uso de sistemas anónimos es la ocultación de la

identidad y evitar ser rastreado. Entre los proyectos más relevantes que utilizan el anonimato se destacan: TOR (*The Onion Router*) [15], Secret [16], Whisper [17], Yik Yak [18].

Comúnmente el argumento más difundido en contra de estas herramientas es el uso inapropiado con fines ilegales, sin embargo, la conclusión es que los criminales ya tienen otros medios más efectivos y mejores para ocultar su identidad que la gente común no tiene. Algunos ejemplos son: teléfonos robados, computadoras infectadas con programas maliciosos, uso de enlaces cifrados *end-to-end*, entre otros. Con este tipo de sistemas anónimos se puede evitar el rastreo por parte de compañías de anuncios, acceder a servicios en Internet bloqueados por el ISP (*Internet Service Provider*), y comunicación en países donde la libre expresión está prohibida.

El presente trabajo se centra en ofrecer un sistema de comunicación donde el contenido del mensaje viaje en claro (es decir, sin importar su posible interceptación o captura por parte de entidades no autorizadas) pero la identidad de la fuente y el destino de la información quede oculta.

1.3.2. DEFINICIÓN

Un Detector de Fallos Anónimo es un dispositivo (software o hardware) que informa al sistema de los procesos que tienen falla, sin utilizar para ello identificadores de proceso.

1.3.3. DEFINCIÓN DE CLASES

Las clases de Detector de Fallos Anónimo más destacadas en la literatura son: AP y $A\Omega$, como versiones anónimas de los Detectores de Fallos P y Ω [19].

- Detector de Fallos AP: Provee a cada proceso de una variable entera, la cual describe el número aproximado de procesos fallidos. Esta variable siempre tiene que ser menor o igual al número de procesos con falla, pero existe un instante de tiempo a partir del cual el contenido de esa variable será exactamente igual que el número de procesos con falla [20]. Este

Detector de Fallos cumple con los mismos requisitos (propiedades de Completitud y Precisión) de P [19].

- Detector de Fallos $A\Omega$: Provee a cada proceso de una variable booleana, denotada como $leader$. Esta variable tarde o temprano permanecerá con el valor $true$ en un determinado proceso correcto (que es desconocido a priori), y con el valor $false$ en los demás procesos correctos.

Se demostró en [19] que la implementación de $A\Omega$ no es posible en ningún caso, incluso en un sistema síncrono. Es por esta razón que los autores en [21] utilizan la clase $A\Omega'$ para implementar la versión de Ω en sistemas anónimos. En [21] los autores presentan la primera versión implementable de $A\Omega'$ en un sistema con fallos Caída-parada. También demuestran en [21] que $A\Omega'$ es más débil que $A\Omega$. El Detector de Fallos $A\Omega'$ es el que se emplea en el presente trabajo, pero extendiendo el modelo de fallos a Caída-recuperación. Esta extensión en el modelo de fallos es totalmente novedosa, ya que no existe, a día de hoy, ningún trabajo en la literatura que lo haya estudiado o implementado cuando los procesos pueden recuperarse de las fallas.

1.3.4. DEFINICIÓN DE AΩ'

El Detector de Fallos $A\Omega'$ provee a cada proceso $p_i \in \Pi$ de dos variables: $leader_i$ y $quantity_i$. El subconjunto L corresponde a los procesos "líder". Un proceso correcto p_i es líder si tarde o temprano tiene la variable $leader_i = true$ permanentemente.

El subconjunto NL contiene los procesos "no líder". Un proceso correcto p_k es no líder si tarde o temprano tiene la variable $leader_k = false$ permanentemente. Por lo tanto, de lo anterior se infiere que $|L \cup NL|$ es igual al número de procesos correctos.

$A\Omega'$ debe satisfacer lo siguiente:

- Todo proceso correcto es tarde o temprano líder o no líder.
- En el sistema, tarde o temprano existe al menos un proceso líder.

- Después de un tiempo finito, todo proceso líder p_i tiene su variable $quantity_i = |L|$. Esto quiere decir que el Detector de Fallos también informa a cada proceso líder de la cantidad de líderes en el sistema.

1.3.5. IMPLEMETACIÓN DE AΩ'

El único trabajo existente actualmente en la literatura donde se implementa $A\Omega'$ es [21] con un modelo de fallos Caída-parada.

En [21] el sistema anónimo sobre el cual se define $A\Omega'$ es parcialmente síncrono, en el sentido de que los procesos ejecutan sus instrucciones en un tiempo desconocido pero acotado. El sistema se compone de un conjunto finito (llamado Π) de n procesos anónimos. Cada proceso perteneciente a Π se lo identifica mediante un subíndice, por ejemplo, $p_i \in \Pi$. El subíndice es solo para simplificar la explicación en los casos que sea necesario, y que esto no supone en ningún caso que los procesos puedan acceder a este identificador, ya que el sistema es anónimo. Todos los procesos están interconectados por enlaces eventualmente fiables. La caída de los procesos es permanente, es decir, utiliza un tipo de falla Caída-parada. Cada proceso $p_i \in \Pi$ utiliza una primitiva $broadcast(m)$ para enviar una copia del mensaje m a todos los demás procesos. El algoritmo que implementa $A\Omega'$ se muestra en el pseudocódigo de la Figura 1.1 [21]. En cada ejecución, tarde o temprano, existirá un conjunto de procesos líderes (con al menos un elemento en dicho conjunto). Es importante destacar que este algoritmo presenta eficiencia en la comunicación, debido a que solo los procesos líderes transmiten mensajes.

Un proceso p_i es líder solo si la condición de la línea 15 se satisface. Como se puede observar la única vez que la variable $leader_i$ es asignada como $false$ es en la línea 1. En otras palabras, una vez que un proceso es líder nunca dejará de serlo. En la etapa $Init$ de las líneas 1 a 3, además de iniciar la Tarea 1 (*Task 1*) y la Tarea 2 (*Task 2*) en distintos hilos, inicializan las siguientes variables:

- $timeout$: variable que introduce la condición de sistema parcialmente síncrono.

- $leader$: indica si un proceso es o no líder.
- seq: número de secuencia del proceso.
- $next\_ack$: siguiente número de secuencia.
- $quantity$: numero de líderes en el sistema, su valor es relevante si el proceso es líder.

Las instrucciones de las líneas 4 a 17 de la Tarea 1 se ejecutan indefinidamente. Cada proceso líder p_i difunde un mensaje ***heartbeat*** (HB, seq_i), siendo seq_i su número de secuencia (líneas 5 a 8). El proceso p_i en la línea 9 espera $timeout_i$ unidades de tiempo, después de lo cual revisa cuantos mensajes *ACK_heartbeat* ha recibido (líneas 10 a 16).

Si el proceso p_i es líder, almacena en el arreglo rec_i los mensajes (ACK_{HB}, s, s') que cumplan la siguiente condición: $s \leq seq_i \leq s'$ (línea 11). La variable $quantity_i$ es la cantidad de mensajes contenidos en rec_i (línea 12), siendo esta la cantidad de líderes en el sistema.

Si el proceso p_i no es líder, los mensajes $(ACK_{HB}, -, -)$ recibidos se almacenan en rec_i, si no existe ningún mensaje en rec_i significa que no hay líderes en el sistema y por lo tanto se considera a sí mismo un líder (línea 15).

La Tarea 2 se define en las líneas 18 a 24. Como se mencionó anteriormente, esta tarea debe ejecutarse paralelamente a la Tarea 1. Cuando un proceso líder p_i recibe un mensaje (HB, s_k) y cumple la condición $s_k \geq next\_ack_i$, quiere decir que el mensaje recibido no ha sido confirmado con *ACK_heartbeat* (líneas 18 - 19). Entonces, en la línea 20, se difunde la confirmación y se aumenta el valor de $next\_ack_i$ en $s_k + 1$ unidades.

Con las acciones anteriores se confirman todos los mensajes ***heartbeat*** en el rango $[next\_ack_i,\ s_k]$. La línea 24 aumenta el valor de la variable $timeout_i$ de un proceso p_i líder, si transmite mensajes (HB, seq_i) más rápido de lo que otro proceso líder p_k responde con (ACK_{HB}, s_k, s_k'). Por lo tanto, el proceso p_i tendrá que esperar más tiempo en la línea 9 antes de difundir nuevos mensajes.

1.4. CONSENSO

1.4.1. DEFINICIÓN DEL PROBLEMA

El problema de Consenso involucra a un conjunto de procesos conectados por enlaces de comunicación que deben decidir un mismo valor, partiendo de los valores propuestos por cada uno de ellos [2].

El problema de Consenso no puede ser resuelto en un sistema asíncrono propenso a fallos de los procesos [9]. Para solucionar este inconveniente, una de las formas es emplear alguna de las clases de Detector de Fallos [2]. De todas las clases existentes, se ha demostrado que el Detector de Fallos más débil para resolver Consenso es Ω [9].

Consenso puede ser resuelto utilizando un Detector de Fallos perfecto *P*, también puede resolverse con uno "imperfecto" que puede cometer inexactitudes. Un ejemplo es $\Diamond P$, el cual después de un cierto tiempo finito comienza a cumplir la propiedad de precisión.

A causa de la no fiabilidad del Detector de Fallos, un proceso correcto puede no tener éxito en acordar su valor propuesto con el resto del sistema, debido a que los demás procesos pudieron haber sospechado de él y obedecer a un líder diferente. Este tipo de algoritmos se conocen como indulgentes [23]. No obstante, siempre es interesante intentar resolver el problema (en el caso del presente trabajo es Consenso) con el Detector de Fallos más débil, es decir, con el que menos información proporciona sobre fallos (en este caso es Ω).

1.4.2. PROPIEDADES

En toda ejecución, cada proceso propone un valor y, pasado un tiempo, todo proceso debe decidir un valor, tal que este valor decidido debe satisfacer las siguientes propiedades:

- Validez (*validity*): Todo valor decidido tiene que ser propuesto por algún proceso del sistema.

Init:

1 $timeout_i \leftarrow 1;\ leader_i \leftarrow false; seq_i \leftarrow 0;$

2 $next\_ack_i \leftarrow 1; quantity_i \leftarrow 0;$

3 ***start*** $Tasks\ 1\ and\ 2$

Task 1:

4 ***while*** $true$ ***do***:

5 ***if*** $(leader_i)$ ***then***

6 $seq_i \leftarrow seq_i + 1;$

7 $broadcast(HB, seq_i);$

8 ***end if***;

9 ***wait until*** $timeout_i\ units;$

10 ***if*** $(leader_i)$ ***then***

11 ***let*** rec_i ***be*** $the\ set\ of\ (ACK_{HB}, s, s')$ $received$ ***such that*** $s \leq seq_i \leq s';$

12 $quantity_i \leftarrow |rec_i|;$

13 ***else***

14 ***let*** rec_i ***be*** $the\ set\ of\ new\ (ACK\_HB, -, -)\ received;$

15 ***if*** $(rec_i = 0)$ ***then*** $leader_i \leftarrow true$ ***end if***

16 ***end if***

17 ***end while***

Task 2:

18 ***upon reception of*** $message\ (HB, s_k)$ ***such that*** $(s_k \geq next\_ack_i)$ ***do***:

19 ***if*** $(leader_i)$ ***then***

20 $broadcast(ACK\_HB, next\_ack_i, s_k);$

21 $next\_ack_i \leftarrow s_k + 1;$

22 ***end if***

23 ***upon reception of*** $message(ACK\ HB, s_k, s'_k)$ ***such that*** $(s_k < seq_i)$ ***do***:

24 ***if*** $(leader_i)$ ***then*** $timeout_i \leftarrow timeout_i + 1$ ***end if***

Figura 1.1. Algoritmo Detector de Fallos AΩ' [21]

- Terminación (*termination*): Todo proceso correcto debe, tarde o temprano, decidir un valor.

- Acuerdo (*agreement*): Todo valor decidido tiene que ser el mismo en todos los procesos.

1.4.3. IMPLEMENTACIÓN

Los algoritmos indulgentes de Consenso presentados en [2], [24], [25] y [26] basan su funcionamiento en el Detector de Fallos $\Diamond S$ (que es equivalente a Ω [22]). El algoritmo que se muestra en el pseudocódigo de la Figura 1.2 es una instancia particular del protocolo genérico introducido por [27]. Este requiere de una mayoría de procesos correctos que satisfaga la condición de $f < n/2$ [23], siendo f el máximo número de procesos que pueden fallar, y n el número total de procesos. Los procesos p_i ejecutan sus instrucciones en rondas r_i, cada ronda se compone de dos tareas, y cada es liderada por un proceso p_c tal que $c = (r \ mod \ n) + 1$.

En la primera tarea ($Task\ 1$), siendo v_i el valor propuesto por p_i, la variable local est_i representa la decisión estimada de p_i. Durante una ronda r, el coordinador p_c trata de imponer su actual valor estimado de decisión. Para lograr esto, una ronda se hace en dos fases. Durante la primera fase ser realizan dos acciones, en la primera, p_c envía est_c a todos los demás procesos (línea 4), y en la segunda cualquier proceso p_i espera hasta recibir el valor estimado de p_c (línea 5). Una vez recibido el valor, el proceso p_i establece su variable auxiliar $aux_i = v = est_c$, o se iguala a un valor por defecto $\perp$ (línea 6).

Luego, la segunda fase de la ronda r se lleva a cabo transmitiendo el valor de aux_i (línea 7). Debido a que se asume que la mayoría de procesos son correctos, ningún proceso se bloquea en la línea 8.

Los mensajes se reciben y almacenan en rec_i, estos valores contenidos en rec_i pueden tener los siguientes valores $\{v\}, \{v, \perp\}, \{\perp\}$. Dependiendo el caso; se decide el valor v y luego difunde su decisión (línea 10), adopta v como su nuevo est_i y pasa a la siguiente ronda (línea 11), o simplemente pasa a la siguiente ronda sin modificar est_i (línea 12). Finalmente, la tarea 2 solo se limita a difundir la decisión tomada.

Function $Consensus(v_i)$

Task 1:

1 $r_i \leftarrow 0; est_i \leftarrow v_i;$

2 **while** $true$ **do**

3 $c \leftarrow (r_i \bmod n) + 1; r_i \leftarrow r_i + 1;$

$\%Phase\ 1\ of\ round\ r$

4 **if** $(i = c)$ **then** $broadcast\ PHASE1(r_i, est_i)$ **end if**;

5 **wait until** $(PHASE1(r_i, v)\ has\ been\ received)$

6 **if** $(PHASE1(r_i, v)\ received\ from\ p_c)$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**;

$\%Phase\ 2\ of\ round\ r$

7 $broadcast\ PHASE2(r_i, aux_i)$;

8 **wait until** $(PHASE2(r_i, aux)\ msgs\ have\ been\ received)$

9 **let** rec_i **be** $the\ set\ of\ values\ received\ by\ p_i$;

10 **case** $rec_i = \{v\}$ **then** $est_i \leftarrow v; broadcast\ DECISION(est_i);$ **stop** $Task\ 1$

11 $rec_i = \{v, \perp\}$ **then** $est_i \leftarrow v$

12 $rec_i = \{\perp\}$ **then** $skip$

13 **endcase**

14 **endwhile**

Task 2:

15 **when** $DECISION(est)\ is\ received: broadcast\ DECISION(est_i)$;

16 $return(est)$

Figura 1.2. Algoritmo de Consenso utilizando Detector de Fallos ◊S [27]

CAPÍTULO 2
ALGORITMO DE DETECCIÓN DE FALLOS

2.1. MODELO DEL SISTEMA

En este capítulo se detallan las características del modelo del sistema para el tipo de fallos Caída-recuperación que se utilizan en el presente trabajo en la implementación de los algoritmos de Detección de Fallos y Consenso. Estas son:

- Procesos: El sistema distribuido se compone de un conjunto de procesos (llamado Π) y su tamaño es n. Como se mencionó en el anterior capítulo, el uso de subíndices en los procesos es con fines de explicación. Los procesos son anónimos y no tienen identificadores.

- Anonimato: Los procesos son anónimos, por tanto, no existe manera de diferenciar dos procesos del sistema.

- Membresía desconocida: Los procesos desconocen para el caso del Detector de Fallos el tamaño de Π. En el caso de Consenso sí conocen el tamaño de Π.

- Tiempo: Los procesos son parcialmente asíncronos, es decir, un proceso p_i ejecuta sus instrucciones en un tiempo desconocido pero acotado.

- Fallos: Los procesos consideran el tipo de fallo Caída-recuperación, los cuales pueden presentar los siguientes comportamientos:

    - Permanentemente ejecutándose (*Permanently-up*): Nunca un proceso se ha caído.

    - Tarde o temprano ejecutándose (*Eventually-up*): Un proceso falla y se recupera un número finito de veces, pero tarde o temprano dejará de fallar.

- Permanentemente caído (*Permanently-down*): Un proceso falla y nunca se recupera.

- Tarde o temprano caído (*Eventually-down*): Un proceso falla y se recupera un número finito de veces, pero tarde o temprano dejará de recuperarse y seguirá caído por siempre.

- Inestable (*Unstable*): Un proceso falla y se recupera repetidas veces.

Los procesos permanentemente caídos, tarde o temprano caídos, o inestables conforman el subconjunto de los procesos incorrectos (llamado $Incorrectos$). Mientras que los procesos permanentemente ejecutándose, y tarde o temprano ejecutándose integran el conjunto de los procesos correctos (llamado $Correctos$). En el caso del Detector de Fallos, al menos un proceso debe ser correcto ($Correctos \subset \Pi$). En concreto, para el Detector de Fallos, de los Correctos al menos un proceso debe ser del tipo permanentemente ejecutándose. En el caso del Consenso, al menos la mayoría de procesos deben ser del tipo permanentemente ejecutándose. Obviamente, siempre debe cumplirse que $Incorrectos \cup Correctos = \Pi$.

- Modelo de comunicación: Un enlace bidireccional de tipo *fiable* conecta a todos los procesos del sistema.

- Primitivas: Los procesos pueden invocar a la primitiva $broadcast(m)$ para enviar una copia del mensaje m a todos los procesos.

- Estado del proceso luego de recuperarse: Cuando un proceso falla, los valores de las variables almacenadas en memoria volátil se pierden. Para recuperar el valor de las variables hasta antes de fallar, estas se guardan en almacenamiento estable. Las variables estables se van a implementar utilizando el disco duro como medio no volátil. Debido a que los tiempos de acceso y respuesta de las variables estables son considerablemente mayores que los de las variables volátiles, en el diseño de los algoritmos se van a

intentar minimizar el uso de las variables estables para mejorar los rendimientos.

- Dinamismo: Involucra el desconocimiento a priori del número de procesos que van a participar del sistema distribuido, y nuevos equipos se pueden ir añadiendo al sistema una vez que ha iniciado.

2.2. DISEÑO

El diseño del nuevo algoritmo de detección de fallos incluye dos características: dinamismo y recuperación de fallos. Como se ha comentado previamente, el dinamismo permite la entrada y salida de nuevos procesos en el sistema. La recuperación de fallos también se conoce como "introducción en caliente". Esta característica permite que si un equipo falla, este pueda reintroducirse al sistema sin necesidad de parar todo el sistema y volver a iniciarlo. Se destaca que las dos características mencionadas anteriormente son nuevas en toda la comunidad internacional para sistemas anónimos, siendo el presente trabajo pionero en su estudio e implementación.

2.2.1. LIBRERÍA PCAP

En la implementación se utiliza la librería PCAP (*Packet Capture Library*), la cual provee una interfaz de programación en leguaje C++ para captura y generación de paquetes y tramas. La librería utilizada pertenece a un proyecto de código abierto que tiene como característica la capacidad de correr sobre diferentes sistemas operativos, con soporte para el guardado de las tramas capturadas en ficheros para su posterior análisis. La descarga y documentación de esta librería está disponible en [28].

El esquema general de un programa que utiliza PCAP es el que se muestra en la Figura 2.1. La fase de inicialización involucra funciones para la obtención de información sobre las interfaces de red y su configuración. En el Código 2.1 se listan las principales funciones utilizadas para la fase de inicialización. La función de la línea

1 del Código 2.1 retorna el puntero a la primera interfaz de red válida, desde la cual las siguientes fases podrán generar o capturar tramas. Si existe algún error al obtener la interfaz de red la función retorna un valor `NULL`. Todas las funciones que manipulen el puntero `errbuf` lo emplearán como un *buffer* para dar a conocer los mensajes de error. Las funciones de las líneas 2 a 4 devuelven un valor igual a `-1` en caso de error.

En la línea 2 del Código 2.1 se muestra la función que devuelve la dirección de red y máscara por medio de las variables `netp` y `maskp` respectivamente. El puntero `device` contiene la dirección de memoria del arreglo de caracteres con el nombre de la interfaz de red a la cual se va consultar la dirección de red y máscara.

La función `pcap_findalldevs`, en la línea 3 del Código 2.1, obtiene todas las interfaces presentes en el sistema capaces de generar y capturar tramas o paquetes. Esta función almacena en el puntero de tipo `pcap_if_t` una lista que contiene todas las interfaces con la dirección de red y máscara.

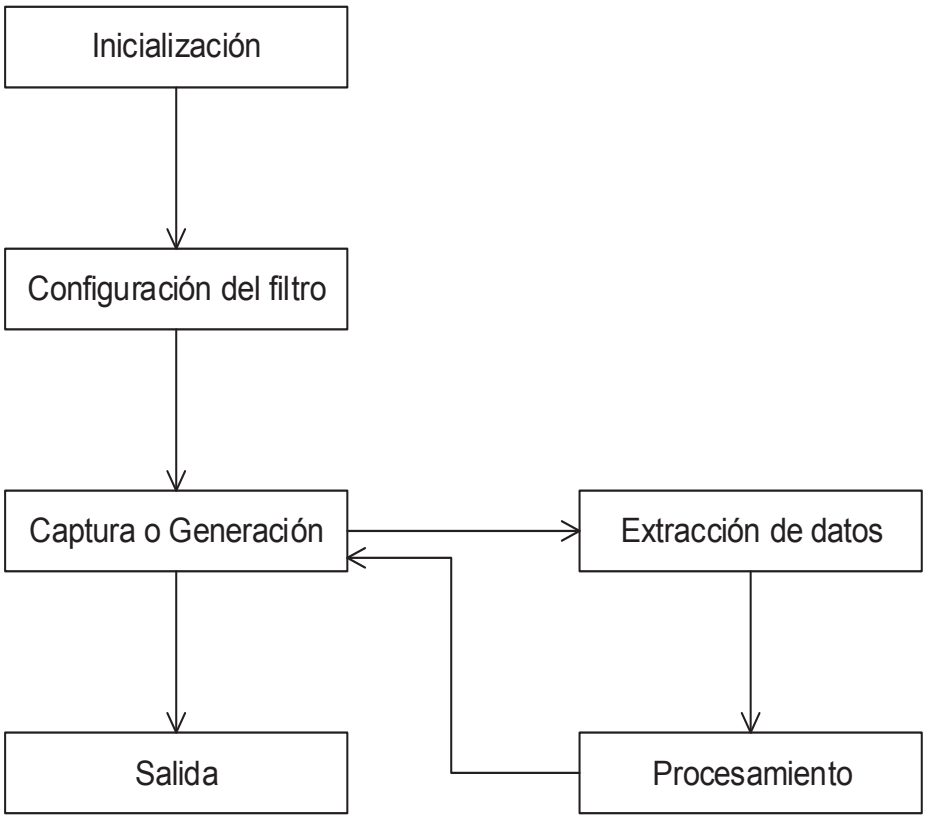

Figura 2.1. Esquema general de un programa con PCAP

```
1  char *pcap_lookupdev(char *errbuf);
2  int pcap_lookupnet(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf);
3  int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf);
4  int pcap_datalink(pcap_t *p);
```

Código 2.1. Funciones empleadas en la fase de inicialización

La Tabla 2.1 resume algunos de los protocolos que PCAP reconoce [28]. La columna DLT_name (*Data Link Type*) corresponde al tipo de enlace de la interface, siendo este el valor devuelto por la función `pcap_datalink`.

La fase de configuración del filtro se realiza en una capa baja del sistema operativo llamada zona *Kernel*, mientras que las funciones de la librería PCAP se ejecutan en el nivel o zona de usuario. La frontera delimitada por ambas zonas tiene que ser traspasada por cada trama o paquete recibido en la interfaz de red. Bajo este contexto, existe un método eficiente que permite aplicar un filtro en la zona *Kernel*, la cual solo permita el paso de paquetes de interés para la aplicación y la frontera no sea traspasada innecesariamente.

Tabla 2.1. Algunos de los protocolos reconocidos por la librería PCAP

DLT_name	Descripción
DLT_EN10MB	IEEE 802.3 Ethernet (10Mb, 100Mb, 1000Mb, y superior)
DLT_IEEE802	IEEE 802.5 *Token Ring*
DLT_SLIP	Protocolo SLIP (*Serial Line Internet Protocol*)
DLT_PPP	Protocolo PPP (*Point to Point Protocol*)
DLT_FDDI	FDDI (*Fiber Distributed Data Interface*)
DLT_PPP_ETHER	PPPoE (*Point to Point Protocol over Ethernet*)
DLT_RAW	IP (*Internet Protocol*) v4 o v6
DLT_FRELAY	*Frame Relay*
DLT_IEEE_802_15_4	IEEE 802.15.4 *Wireless Personal Area Network*
DLT_SDLC	Protocolo SDLC (*Synchronous Data Link Control*)

Cada sistema operativo define su método de filtrado como, por ejemplo: NIT (*Network Interface Tap*) para SunOS, *Ultrix Packet Filter* para DEC Ultrix, BPF (*Berkeley Packet*

Filter) para sistemas BSD, y LSF (*Linux Socket Filter*) para Linux [28]. Los filtros BPF son de interés del presente proyecto, ya que el sistema operativo sobre el cual se implementa es de tipo BSD.

El funcionamiento de un filtro BPF se compone de dos partes. La primera se conoce como *Network Tap* encargada de recopilar los paquetes desde el *driver* de la interfaz de red y entregar dichos paquetes a la aplicación. La segunda parte es *Packet Filter*, la cual implementa el filtro y decide si el paquete debe o no ser aceptado.

La Figura 2.2 muestra un esquema general del funcionamiento del filtro BPF. Como se puede observar, todas las acciones realizadas por el filtro BPF se ejecutan en zona de *Kernel*. Cuando un paquete llega a la interfaz de red, el driver primero entrega el paquete al filtro BPF. Si el paquete satisface las condiciones del filtro, una copia es entregada a los *buffers* de las aplicaciones. Más tarde, el paquete es devuelto al *driver* para que lo que entregue a la pila de protocolos que llevan el paquete a la aplicación correspondiente.

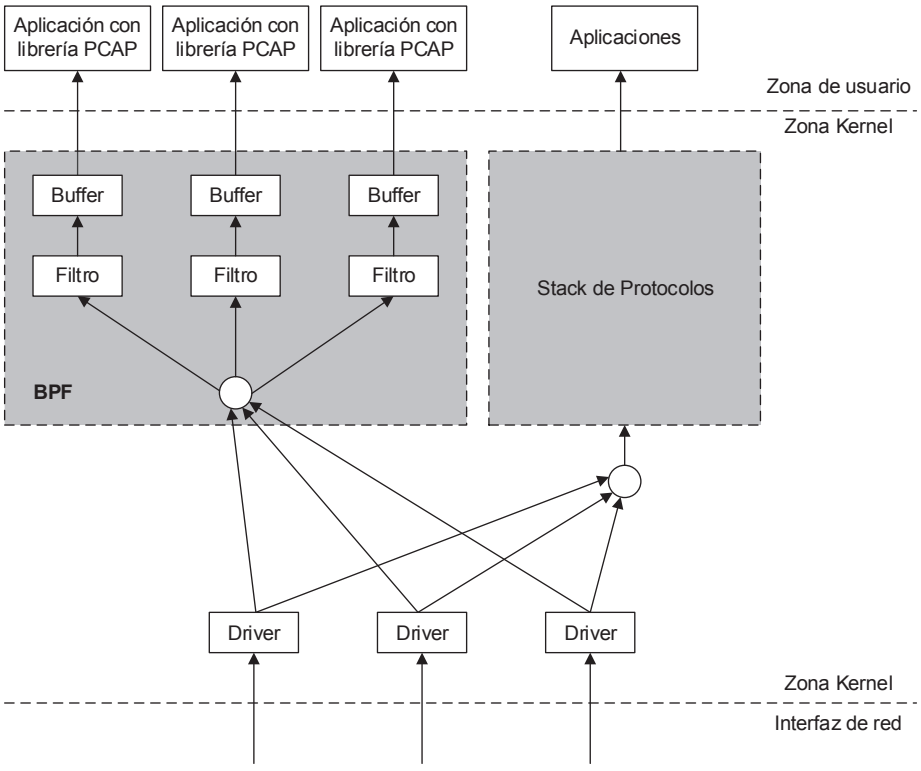

Figura 2.2. Esquema del funcionamiento del filtro BPF

La definición de un filtro BPF se la hace a través de primitivas de filtrado TCPDUMP [28]. En la Tabla 2.2 se describe 3 tipos de modificadores que el filtro puede tener.

Las expresiones con primitivas de filtrado pueden combinarse con paréntesis y operadores lógicos como: negación (`! not`), concatenación (`&& and`) y alternativa (`|| or`). En el Código 2.2 se muestran algunos ejemplos de filtros. La línea 1 captura el tráfico generado por el *host* con la dirección IP 192.168.1.1. La línea 2 captura todo el tráfico web, y la línea 3 captura todas las peticiones DNS (*Domain Name System*).

Tabla 2.2. Modificadores de primitivas de filtrador TCPDUMP

Modificador	Descripción
tipo	Define si es una dirección de *host*, dirección de red o un puerto.
dir	Determina orígenes y destinos de los datos con los modificadores src y dst.
proto	Identifica el protocolo a capturar.

```
1  src host 192.168.1.1
2  tcp and port 80
3  udp and dst port 53
```

Código 2.2. Ejemplos de filtros

Las funciones que se utilizan para la configuración del filtro se muestran en el Código 2.3. Dos de las funciones tienen el descriptor `pcap_t *p` obtenido mediante la función `pcap_open_live`, el funcionamiento de esta función se detalla más adelante. La línea 1 compila el filtro descrito en `srt` en formato TDPDUMP a su equivalente BPF que la función almacenará en `fp`. Para indicar al compilador la optimización del filtro se utiliza la variable `optimize`. Si el filtro es para IPv4 se especifica la máscara de subred en `netmask`.

Una vez compilado el filtro, se lo aplica con la función `pcap_setfilter` descrita en línea 2 del Código 2.3. Si existe error durante la ejecución de estas dos funciones, se puede llamar a `pcap_geterr()` para obtener la descripción del error. Una vez obtenida la información relacionada a la interfaz de red se procede a la compilación

del filtro. Este filtro requiere un descriptor de tipo `pcap_t`, el cual es obtenido mediante la función de la línea 1 del Código 2.4. Para ello se necesita especificar la interfaz mediante el puntero `device`, junto con la cantidad de bytes a capturar en la variable `snaplen`.

```
1  int pcap_compile(pcap_t *p, struct bpf_program *fp,char *str, int optimize, bpf_u_int32
   netmask);
2  int pcap_setfilter(pcap_t *p, struct bpf_program *fp);
3  pcap_geterr();
```

Código 2.3. Funciones utilizadas en la fase de configuración del filtro

Si se desea que la interfaz se abra en modo promiscuo, `promisc` debe tener un valor diferente de cero. La variable `to_ms` especifica el tiempo en milisegundos en donde el *Kernel* recolectará paquetes y los pasará a la zona de usuario. Esto evita el paso entre la zona de *Kernel* y la zona de usuario por cada paquete. Si se produce algún error la función devuelve `NULL` y una descripción del error en `errbuf`.

```
1  pcap_t *pcap_open_live(char *device, int snaplen,int promisc, int to_ms, char *errbuf);
2  int pcap_dispatch(pcap_t *p, int cnt,pcap_handler callback, u_char *user);
3  int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
4  u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h);
5  int pcap_inject(pcap_t *p, const void *buf, size_t size);
```

Código 2.4. Funciones utilizadas en la fase de configuración del filtro

La función de la línea 2 del Código 2.4 permite la captura de los paquetes. La variable `cnt` indica el número de paquetes a capturar, un valor igual a -1 origina una captura indefinida. El puntero `callback` especifica la función que será llamada cada vez que se recibe un paquete. Para que un puntero a función sea de tipo `pcap_handler` debe cumplir los requisitos listados en la Tabla 2.3. Si en el puntero `user` se especifica un valor, más adelante en el programa, se podrá identificar quién utiliza la función en caso de que sea una aplicación con múltiples capturas.

Las funciones de las líneas 2 y 3 del Código 2.4 son similares, ambas finalizan cuando han procesado `cnt` paquetes, pero la diferencia está en que `pcap_dispatch`

termina cuando `to_ms` expira. Es decir, para `pcap_dispatch` la variable `cnt` es el número máximo de paquetes a procesar antes de terminar, por lo que puede procesar menos paquetes. La función de línea 4 simplemente lee un único paquete y devuelve su contenido sin utilizar una función `callback`.

Tabla 2.3. Requisitos para que un puntero a función sea de tipo pcap_handler

Requisito	Descripción
Puntero `u_char`	Dirección de memoria donde se guardará el paquete.
Estructura `pcap_pkthdr`	Estructura que contiene una marca de tiempo, tamaño del paquete al ser capturado, y tamaño del paquete en el fichero.

Para generar paquetes con esta librería se utiliza la función `pcap_inject` mostrada en la línea 5 del Código 2.4. Recibe como parámetros el descriptor de la interfaz `p`, `buff` apuntando a los datos del paquete incluyendo sus cabeceras, junto con su longitud en bytes en la variable `size`.

Para extraer los datos de un paquete recibido, PCAP define una estructura de datos genérica que describen los campos de una trama Ethernet según el estándar IEEE 802.3. La definición de los campos está en el fichero `ethernet.h` de la librería PCAP, disponible en el directorio `/usr/includes/net/`. Un fragmento de este fichero se muestra en el Código 2.5.

Como se puede observar en la línea 5 del Código 2.5, la estructura que contiene la cabecera Ethernet se llama `ether_header`. En ella se encuentra la dirección destino (`ether_host`), la dirección origen (`ether_shost`) y el tipo de trama (`ether_type`). Los valores de tipo de trama se definen tomando en cuenta el RFC 5342 [30], en las líneas 11 a 14 se muestra algunos ejemplos. Las longitudes de los campos y la longitud máxima y mínima de la trama se declaran en las líneas 16 a 20, todos estos valores siguiendo el estándar IEEE 802.3. Una vez defina la estructura de la trama Ethernet, se puede acceder a todos sus valores mediante el puntero `u_char *packet` que las funciones `handler` deben implementar satisfaciendo las condiciones definidas en la Tabla 2.3.

La librería PCAP además ofrece la posibilidad de guardar las tramas capturadas en un fichero para procesarlos más tarde, para ello PCAP ofrece las funciones del Código 2.6. La función de la línea 1 del Código 2.6, crea un fichero de salida con el nombre contenido en `fname`. Esta función devuelve un descriptor de tipo `pcap_dumper_t`. Para escribir el paquete en el fichero creado, se utiliza `pcap_dump`, esta función recibe como parámetros: el usuario que hace uso de la función, la estructura `pcap_pkthdr` definida en la Tabla 2.3, y el paquete en sí contenido en `sp`. Al terminar la escritura del fichero se emplea la función `pcap_dump_close`.

```
1  struct ether_addr{
2          u_int8_t ether_addr_octet[ETH_ALEN];
3  } __attribute__ ((__packed__));
4
5  struct ether_header{
6          u_int8_t ether_dhost[ETH_ALEN];
7          u_int8_t ether_shost[ETH_ALEN];
8          u_int16_t ether_type;
9  }__attribute__(( packed ));
10
11 #define ETHERTYPE PUP 0x0200
12 #define ETHERTYPE IP 0x0800
13 #define ETHERTYPE ARP 0x0806
14 #define ETHERTYPE REVARP 0x8035
15 #define ETHER_ADDR_LEN ETH_ALEN
16 #define ETHER_TYPE_LEN 2
17 #define ETHER_CRC_LEN 4
```

Código 2.5. Fragmento de código del fichero ethernet.h

2.2.2. ALGORITMO

El punto de partida para el diseño e implementación del protocolo del presente trabajo ha sido el algoritmo de $A\Omega'$ [21] pero extendiendo el modelo de fallos al tipo Caída-recuperación. En el pseudocódigo de la Figura 2.3 se muestra el algoritmo desarrollado para este trabajo. Seguidamente se va explicar el algoritmo resultante comparándolo con la solución de [21].

```
when a process p_i initializes or recovers:
 1  TIMEOUT_i ← TIMEOUT_i + 1; % in stable storage
 2  wait TIMEOUT_i units of time;
 3  if ((HB, seq_i) received) then
 4          leader_i ← false;
 5  else
 6          leader_i ← true;
 7  end if;
 8  seq_i ← 0;
 9  next_ack_i ← 1; quantity_i ← 0;
10  start tasks 1 and 2

task 1:
11  while true do:
12          if (leader_i) then
13                  seq_i ← seq_i + 1;
14                  broadcast(HB, seq_i);
15          end if;
16          wait until TIMEOUT_i units;
17          if (leader_i) then
18                  let rec_i be the set of (ACK_HB, s, s')
                    received such that s ≤ seq_i ≤ s';
19                  quantity_i ← |rec_i|;
20          else
21                  let rec_i be the set of new (ACK_HB, −, −) received;
22                  if (rec_i = 0) then leader_i ← true end if
23          end if
24  end while

task 2:
25  upon reception of message (HB, s_k) such that (s_k ≥ next_ack_i) do:
26          if (leader_i) then
27                  broadcast(ACK_HB, next_ack_i, s_k);
28                  next_ack_i ← s_k + 1;
29          end if
30  upon reception of message(ACK HB, s_k, s'_k) such that (s_k < seq_i) do:
31                  if (leader_i) then TIMEOUT_i ← TIMEOUT_i + 1 end if
```

Figura 2.3. Algoritmo Detector de Fallos AΩ' con Caída-Recuperación

Se destaca que la diferencia entre la Figura 1.1 y la Figura 2.3 radica en que la variable $TIMEOUT$ se guarda en el disco duro (es decir, en almacenamiento estable), para que cuando un proceso se recupere pueda iniciar su estado a partir de los valores almacenados. En la línea 1 se recupera el valor de $TIMEOUT$, y se espera $TIMEOUT$ unidades de tiempo para recibir mensajes (línea 2). Si se recibe un mensaje

$heartbeat$ entonces quiere decir que existe un líder en el sistema y por lo tanto su variable $leader_i$ será $false$ (líneas 3 a 7). El funcionamiento del resto del algoritmo es exactamente igual al descrito en la Sección 1.3.5.

```
1  pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname);
2  void_pcap_dump(u_char *user, struct pcap_pkthdr *h,u_char *sp);
3  void pcap_dump_close(pcap_dumper_t *p);
```

Código 2.6. Funciones utilizadas en la fase de procesamiento

Como se puede ver en la Figura 2.4, el diagrama de actividad describe las líneas 1 a 10 del Figura 2.1. Se recupera la variable $TIMEOUT$ que está almacenada en disco, y se inicializan las variables temporales seq, $next\_ack$, $leader$ y $quantity$ para luego iniciar la Tarea 1 y Tarea 2 en diferentes hilos que se ejecutan en paralelo. Es importante mencionar que ambas tareas se ejecutan en bucles infinitos.

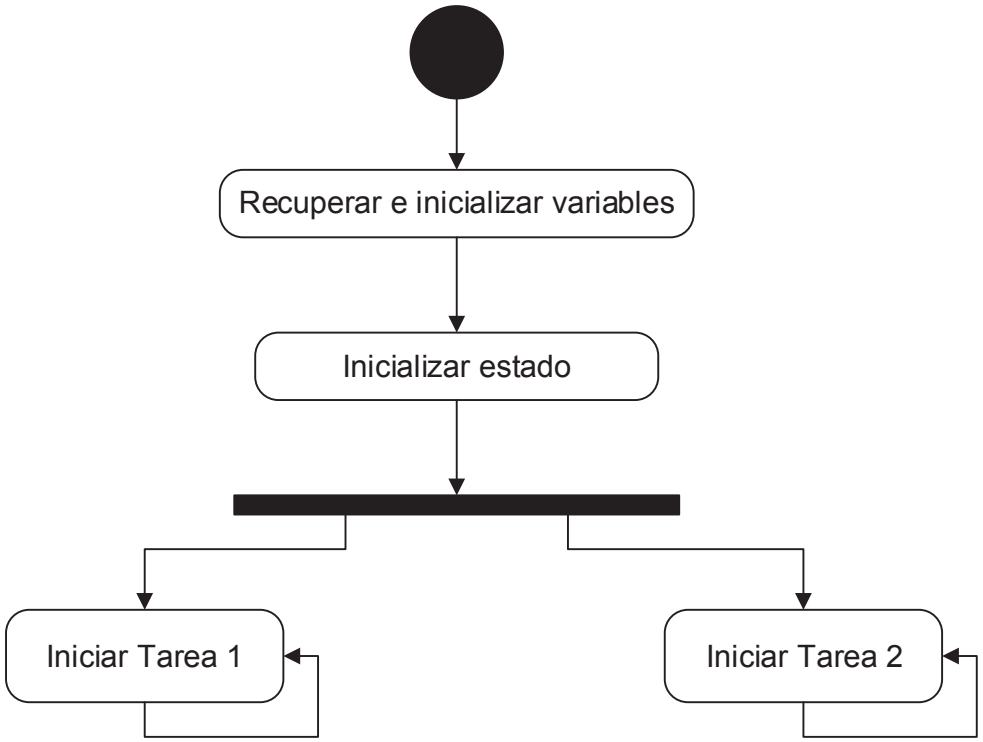

Figura 2.4. Diagrama de actividad del algoritmo Detector de Fallos

En la Figura 2.5 se muestra el diagrama de actividad de la Tarea 1, obedeciendo al algoritmo del Figura 2.1. Se evalúa dos veces si el proceso es o no es líder, ya que

en la Tarea 1 es donde se puede dar el caso de que un proceso cambie su estado de $leader = false$ a $leader = true$.

La Figura 2.6 detalla las actividades que realiza la Tarea 2. Como se puede observar la Tarea 2 evalúa los mensajes recibidos con la finalidad de aumentar la variable $TIMEOUT$ si es que un proceso va muy rápido con respecto a los demás. Esta acción introduce la característica de sistema parcialmente síncrono.

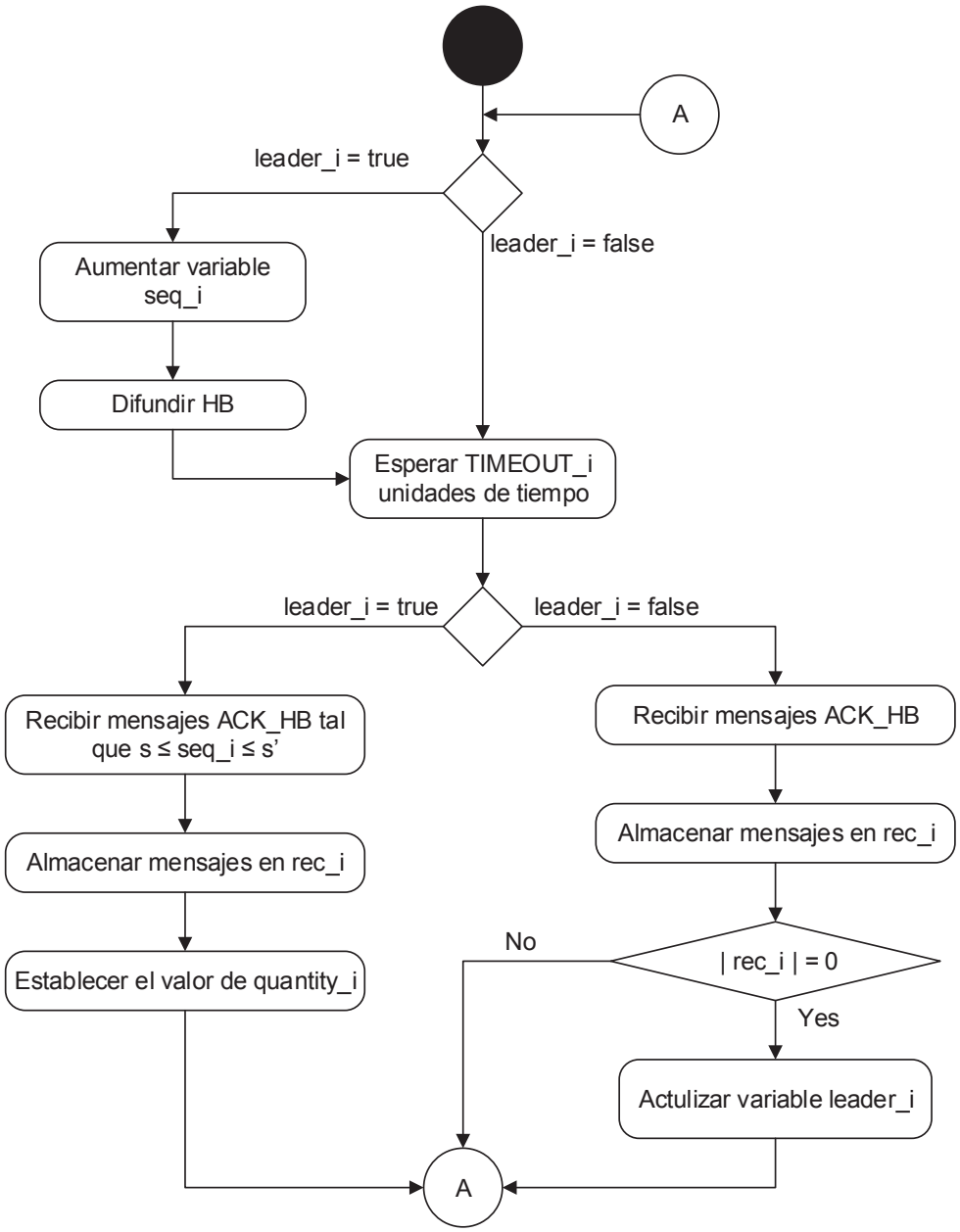

Figura 2.5. Diagrama de actividad de Tarea 1

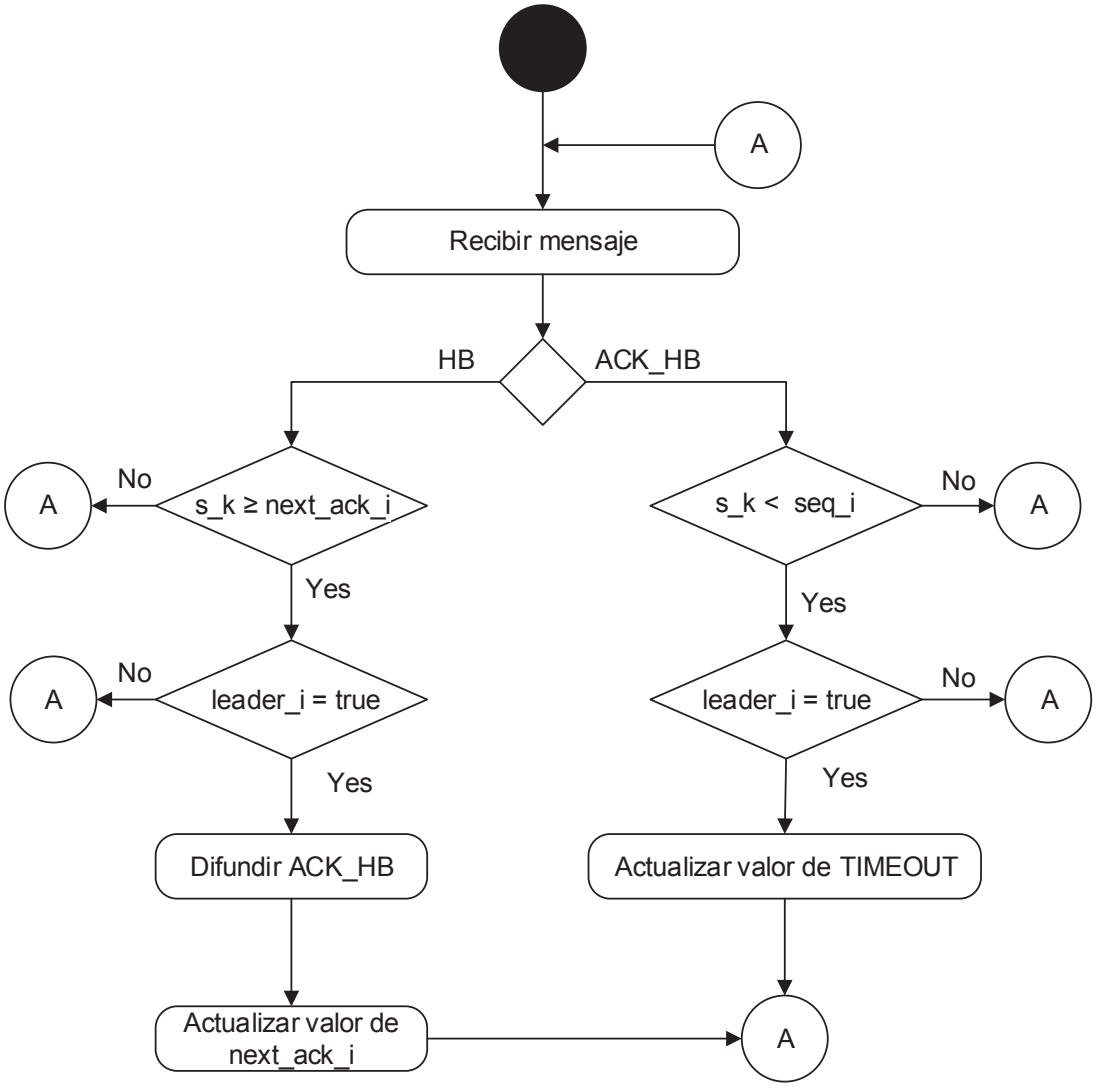

Figura 2.6. Diagrama de actividad de Tarea 2

2.2.3. DIAGRAMA DE CLASES

El diseño del Detector de Fallos sigue el patrón *Singleton*, es decir, todas las aplicaciones tendrán una única instancia del objeto Detector de Fallos. Para informar a las aplicaciones de los cambios que puede tener el Detector de Fallos se utiliza el patrón de diseño *Observer*.

Cualquier aplicación que requiera enterarse de los cambios del Detector de Fallos debe implementar la interfaz que la clase `FailureDetectorStatusListener` define y registrarse con el Detector de Fallos. Además, se emplean dos clases que ayudan al correcto funcionamiento del protocolo. La primera es la clase

`MessageFactory`, que es una factoría de mensajes. Esta clase registra todos los tipos de mensajes que manejan las aplicaciones. De esta manera, es posible realizar una serialización y deserialización en función del mensaje. La segunda clase es `CrashRecoveryManager`, la cual administra la recuperación de las variables y su escritura en disco.

2.2.3.1. Clase FailureDetector

En la Figura 2.7 se muestra la clase `FailureDetector`. Las aplicaciones que usen el Detector de Fallos requieren conocer dos valores: $leader$, para determinar si el proceso es líder o no, y $quantity$, para conocer el número de líderes que el Detector de Fallos estima que hay en el sistema.

Como se mencionó anteriormente, la instancia del Detector de Fallos es única, es decir, existe un solo objeto Detector de Fallos para todas las aplicaciones que requieran el servicio del Detector de Fallos. Esto se conoce como patrón de diseño *Singleton*. De esta manera, se asegura una única instancia de clase.

Por esta razón, el constructor `FailureDetector()` es privado, y las aplicaciones tendrán que utilizar el método estático `getInstance()`. Cuando este método es llamado por primera vez, instancia e inicializa un objeto de la clase, en las siguientes llamadas devolverá la referencia al objeto que fue instanciado en la primera vez.

Atributos privados:

- `communicator`: Utilizado para enviar y recibir mensajes.
- `crManager`: Administrador de Caída – recuperación.
- `leader`: Atributo correspondiente a la variable $leader$ del algoritmo de la Figura 2.3.
- `next_ack`: Atributo correspondiente a la variable $next\_ack$ del algoritmo de la Figura 2.3.
- `quantity`: Atributo correspondiente a la variable $quantity$ del algoritmo de la Figura 2.3.
- `seq`: Atributo correspondiente a la variable seq del algoritmo de la Figura 2.3.

- `timeout`: Atributo correspondiente a la variable $TIMEOUT$ del algoritmo de la Figura 2.3.
- `listenersMutex`: Control de concurrencia.
- `messageFactory`: Factoría de mensajes.
- `incomingMessages`: Cola para mensajes entrantes del comunicador.
- `newlyreceivedHBAckMessages`: Lista de nuevos mensajes ACK_HB.
- `receivedHBAckMessages`: Lista de mensajes ACK_HB.
- `receivedHBAckMessages`: Control de concurrencia.
- `statusListeners`: Listado de observadores.
- `stopTask1Thread`: Variable utilizada para detener el hilo de la Tarea 1.
- `stopTask2Thread`: Variable utilizada para detener el hilo de la Tarea 2.
- `Task1Thread`: Hilo de la Tarea 1.
- `Task2Thread`: Hilo de la Tarea 2.

Métodos públicos:

- `amLeader()`: Devuelve un valor booleano indicando si es líder o no el proceso.
- `currentTimeout()`: Devuelve el valor actual de la variable $TIMEOUT$.
- `getInstance()`: Instancia un objeto si es la primera llamada, de lo contrario devuelve la referencia al objeto.
- `getNextAck()`: Devuelve el valor actual de la variable $getNextAck$.
- `getSeq()`: Devuelve el valor actual de la variable seq.
- `numLeaders()`: Devuelve el número de líderes que el Detector de Fallos estima que hay en el sistema.
- `registerStatusListener()`: Registra una aplicación como observador.
- `unregisterStatusListener()`: Quita a una aplicación de la lista de observadores.

Métodos privados:

- `FailureDetector()`: Constructor de la clase.
- `~FailureDetector()`: Destructor de la clase.

- `FailureDetector(FailureDetector)`: Previene al compilador de generar copias innecesarias del objeto.
- `initialize_leader_status()`: Inicializa el estado del Detector de Fallos como proceso líder o no líder.
- `notifyAllListeners()`: Se invoca cada vez que existe un cambio en el Detector de Fallos.
- `recover()`: Rescata el valor de las variables almacenadas en disco duro.
- `remove_old_HB_ACK()`: Borra los mensajes HB_ACK que ya no se necesitan.
- `task1()`: Implementa la Tarea 1.
- `task2()`: Implementa la Tarea 2.

2.2.3.2. Clase CrashRecoveryManager

Esta clase permite la escritura en disco de las variables, así como la recuperación de sus valores después de un fallo (Figura 2.7). Se comporta como un Administrador de Caída-recuperación.

Atributos privados:

- `processTag`: Identificador de proceso.
- `recoveredProcess`: Indica si es no un proceso recuperado.
- `recoveryFile`: Descriptor del fichero de recuperación.
- `recoveryFilename`: Nombre del fichero de recuperación.
- `synchronizeMutex`: Control de concurrencia.
- `variables`: Lista de variables recuperadas.

Métodos públicos:

- `CrashRecoveryManager()`: Constructor de la clase.
- `~CrashRecoveryManager()`: Destructor de la clase.
- `discardStableVariables()`: Descarta información sobre las variables, en memoria y en disco duro.
- `isRecoveredProcess()`: Indica si se recuperaron las variables del proceso.
- `recoverVariable()`: Recupera el valor de una variable desde el disco duro.

- `registerVariable()`: Registra una variable con el Administrador de Caída-recuperación.
- `synchronize()`: Actualiza el valor de una variable en el disco duro.
- Métodos privados:
- `freeResources()`: Libera recursos, excepto el descriptor del ficheri de recuperación.
- `initializeRecoveryFile()`: Crea e inicializa un nuevo fichero de recuperación.
- `loadRecoveryData()`: Carga en memoria el valor de las variables recuperadas del fichero.
- `locateRecoveryFile()`: Localiza un fichero de recuperación y crea uno si no lo encuentra.
- `tryOpen()`: Intenta abrir el fichero de recuperación, se asegura que la información sea consistente.

2.2.3.3. Clase AnonymousCommunicator

Clase padre para la clase `EthernetAnonymousCommunicator`, la cual define los métodos que la clase hija debe implementar (Figura 2.7).

Métodos públicos:

- `~AnonymousCommunicator`: Destructor de la clase, cierra la comunicación y libera recursos.
- `broadcast()`: Difusión de un mensaje.

2.2.3.4. Clase EthernetAnonymousCommunicator

Clase utilizada para realizar la comunicación anónima. Encapsula los mensajes de la aplicación en tramas Ethernet (Figura 2.7).

Atributos privados:

- `IIEAC`: Instancia de `LowLevelEthernetAnonymousCommunicator`.
- `IIEACMap`: Arreglo de `LowLevelEthernetAnonymousCommunicatorMap`.
- `IIEACMapMutex`: Control de concurrencia.

- `messageBufferReceivingQueue`: Cola de recepción de mensajes.
- `messageFactory`: Factoría de mensajes.
- `messageReceivingQueue`: Cola de recepción.
- `networkInterfaceName`: Nombre de la interfaz por la cual se envían y reciben mensajes.
- `receivingThread`: Hilo de recepción.
- `serviceTag`: Identificador de protocolo.
- `stopReceivingThread`: Variable *booleana* utilizada para detener el hilo de recepción.

Métodos públicos:

- `EthernetAnonymousCommunicator`: Constructor de la clase.
- `~EthernetAnonymousCommunicator`: Destructor de la clase, cierra la comunicación y libera recursos.
- `broadcast()`: Realiza una difusión (*broadcast*) de un mensaje anónimo.

Método privado:

- `messageReceiver()`: Método que implementa el hilo de recepción.

2.2.3.5. Clase LowLevelEthernetAnonymousCommunicator

Clase para definir la comunicación de bajo nivel de Ethernet (Figura 2.7).

Atributos privados:

- `receivingQueues`: Cola de recepción.
- `receivingQueuesMutex`: Control de concurrencia.

Atributo protegido:

- `sendingQueue`: Cola de envío.

Métodos públicos:

- `addReceivingQueues()`: Añade una cola de recepción.
- `broadcast()`: Difusión de un mensaje.

- `distributePacket()`: Distribuye una trama a todas las colas.
- `dropReceivingQueue()`: Remueve una cola de recepción.
- `~LowLevelEthernetAnonymousCommunicator()`: Destructor de clase.

2.2.3.6. Clase PCAPEthernetAnonymousCommunicator

Implementa el comunicador para sistemas Linux (Figura 2.7).

Atributos privados:

- `pcapfd`: Instancia de PCAP.
- `receivingThread`: Hilo de recepción.
- `sendingThread`: Hilo de envío.
- `stopSendingThread`: Variable *booleana* utilizada para detener el hilo de envío.

Métodos privados:

- `packetReceiver()`: Implementa el hilo de recepción.
- `packetSender()`: Implementa el hilo de envió.

Métodos públicos:

- `PCAPEthernethAnonymousCommunicator()`: Constructor de la clase.
- `~ PCAPEthernethAnonymousCommunicator()`: Destructor de la clase.

2.2.3.7. Clase BPFLevelEthernetAnonymousCommunicator

Clase que implementa el comunicador para FreeBSD (Figura 2.7).

Atributos públicos:

- `bpfBufferLength`: Tamaño de *buffer*.
- `bpffd`: Instancia de PCAP.
- `bytesReceived`: Número de *bytes* recibidos.
- `receivingBpfBuffer`: *Buffer* de recepción.
- `receivingPos`: Define la posición en el arreglo de *bytes* del mensaje.
- `receivingThread`: Hilo de recepción.

- `sendingThread`: Hilo de envío.
- `stopReceivingThread`: Variable utilizada para detener el hilo de recepción de mensajes.
- `stopSendingThread`: Variable utilizada para detener el hilo de envío.

Métodos públicos:

- `BPFEthernetAnonymousCommunicator`: Constructor de la clase.
- `~BPFEthernetAnonymousCommunicator`: Destructor de la clase.
- `packetReceiver`: Implementa el hilo de envío.
- `packetSender`: Implementa el hilo de recepción.

2.2.3.8. Clase MessageFactory

Implementa la factoría de mensajes (Figura 2.7).

Atributos privados:

- `unpackers`: Lista de las funciones de deserialización.

Métodos públicos:

- `produce()`: Produce un nuevo mensaje.
- `registerType`: Registra un tipo de mensaje.

2.2.3.9. Clase Message

En la Figura 2.8 se muestra la clase `Message`. Cualquier mensaje que se defina debe heredar de la clase `Message` e implementar los métodos `pack()` y `unpack()` con la finalidad de serializar y deserializar los mensajes.

Los mensajes que se van a intercambiar en el Detector de Fallos son dos: `FDHeartbeatMessage` (HB) y `FDHeartbeatAckMessage` (ACK_HB).

2.2.3.10. Clase FDHeartbeatMessage

La Figura 2.8 muestra la clase `FDHeartbeatMessage`. Es una subclase de `Message` que define al mensaje HB del pseudocódigo de la Figura 2.3.

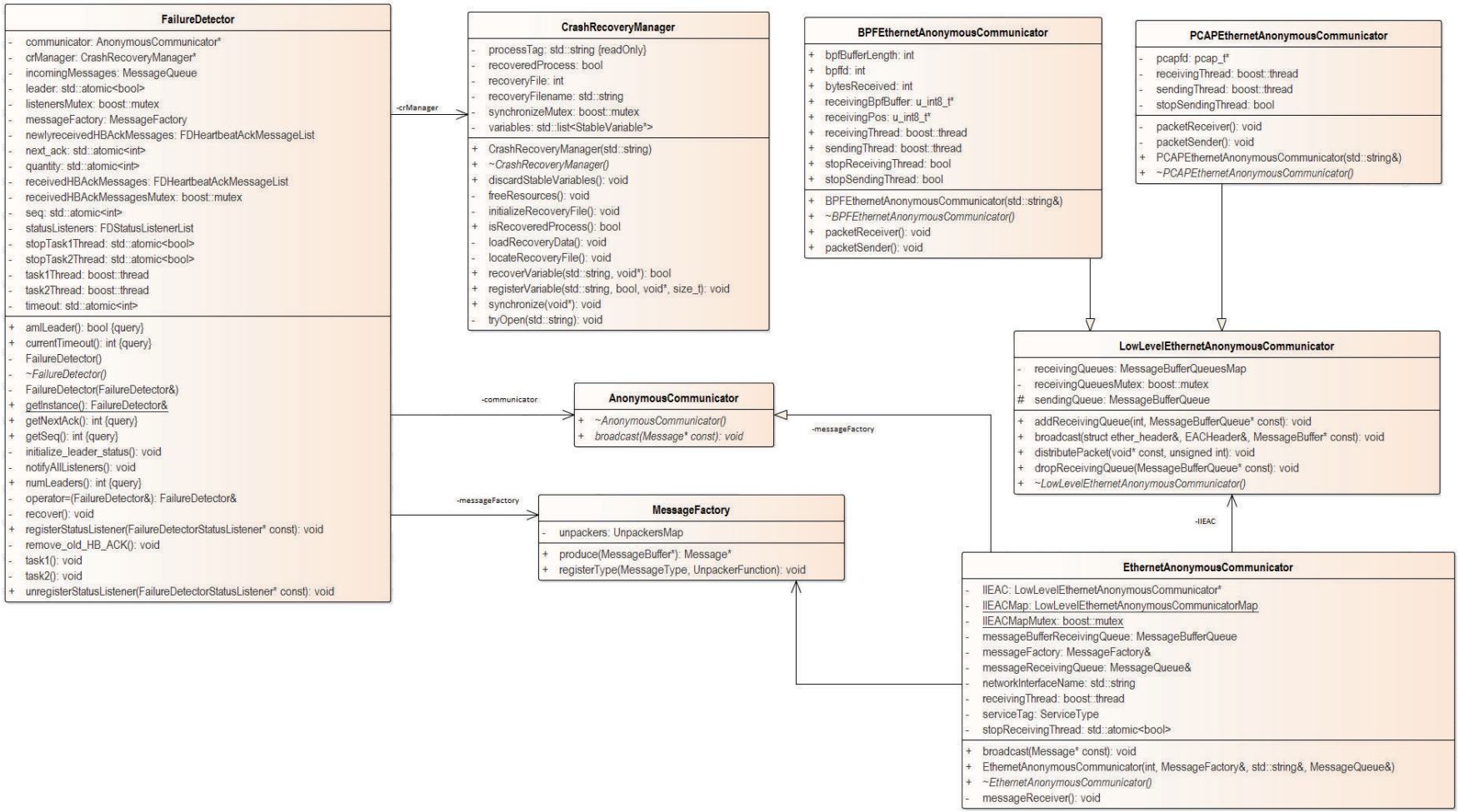

Figura 2.7. Diagrama de clases del Detector de Fallos

Atributos privados:

- `seq`: Indica el número de secuencia.

Atributos públicos:

- `type_id`: Identificador de tipo de mensaje.

Métodos públicos:

- `FDHeartbeatMessage()`: Constructor sin parámetros de entrada.
- `FDHeartbeatMessage(int)`: Constructor especificando el número de secuencia.
- `~FDHeartbeatMessage()`: Destructor de la clase.
- `pack()`: Método heredado para serializar el mensaje.
- `unpack()`: Método heredado para deserializar el mensaje.
- `getSeq()`: Método para consultar el valor del número de secuencia.

2.2.3.11. Clase FDHeartbeatAckMessage

La Figura 2.8 muestra la clase `FDHeartbeatAckMessage`. Esta clase es una subclase de `Message` que define al mensaje ACK_HB de la Figura 2.1.

Atributos privados:

- `seq`: Corresponde al número de secuencia.
- `next_ack`: Corresponde a next_ack.

Atributos públicos:

- `type_id`: Identificador de tipo de mensaje.

Métodos públicos:

- `FDHeartbeatAckMessage()`: Constructor sin parámetros de entrada.
- `FDHeartbeatAckMessage(int, int)`: Constructor especificando el número de secuencia y next_ack.
- `~FDHeartbeatAckMessage()`: Destructor.
- `pack()`: Método heredado para serializar el mensaje.

- `unpack()`: Método heredado para deserializar el mensaje.
- `getSeq()`: Método para consultar el valor del número de secuencia.
- `getNextAck()`: Método para consultar el valor del número de next_ack.

2.2.3.12. Clase TestMessage

En la Figura 2.8 se muestra la clase `TestMessage`. Es una subclase de `Message` que define un mensaje que se utiliza con fines de prueba.

Atributos privados:

- `number`: Indica el número de secuencia.
- `text`: Texto a enviar o recibir en el mensaje.

Atributos públicos:

- `type_id`: Identificador de tipo de mensaje.

Métodos públicos:

- `TestMessage()`: Constructor sin parámetros de entrada.
- `TestMessage(int, string)`: Constructor especificando el número de secuencia y cadena de caracteres.
- `~ TestMessage()`: Destructor.
- `pack()`: Método heredado para serializar el mensaje.
- `unpack()`: Método heredado para deserializar el mensaje.
- `getNumber()`: Método para consultar el valor de `number`.
- `getText()`: Método para consultar el valor de la cadena de caracteres de `text`.

2.2.3.13. Clase FailureDetectorStatusListener

En la Figura 2.9 se muestra la clase `FailureDetectorStatusListener`. El estado del Detector de Fallos puede cambiar en el tiempo, por ejemplo: la variable $quantity$ cambia de número o $leader$ de $false$ a $true$. Por esta razón, para que las aplicaciones que hacen uso del Detector de Fallos se enteren de los cambios que

este pueda tener, se utiliza el patrón de diseño *Observer*. Cualquier aplicación, que requiera enterarse de los cambios del Detector de Fallos, debe implementar la interfaz que esta clase define y registrarse con el Detector de Fallos. Esta clase se constituye como el sujeto del patrón de diseño *Observer*. Cada vez que suceda un cambio en el Detector de Fallos, esta clase utilizar el método `statusChanged()`.

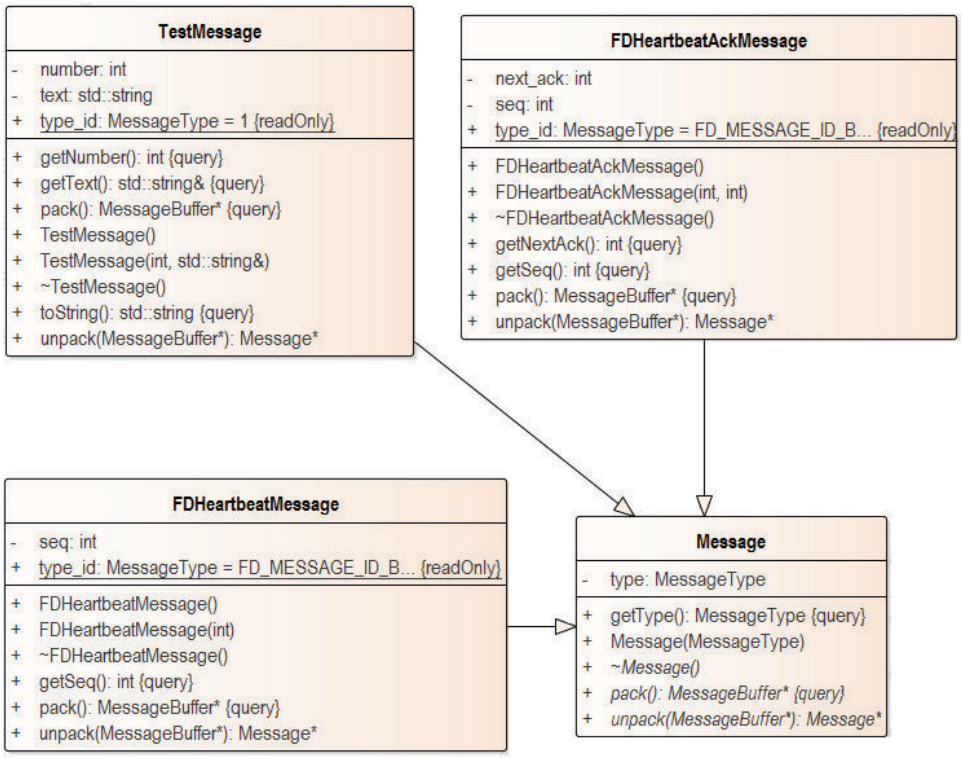

Figura 2.8. Diagrama de clases para tipos de mensajes del Detector de Fallos

Métodos públicos:

- `statusChanged()`: Se invoca cuando existe un cambio en el Detector de Fallos.
- `~FailureDetectorStatusListener()`: Destructor de la clase.

2.2.3.14. Clase BlockingQueue

Esta clase define una cola FIFO bloqueante, esto es para controlar la concurrencia de los hilos cuando accedan a una cola (Figura 2.9).

Atributos privados:

- `condition`: Permite el control de concurrencia.

- `mutex`: Definición de regiones críticas, con cerrojos, para control de concurrencia.
- `queue`: Cola.
- `stop`: Permite detener las operaciones `pop()` (extraer mensajes un arreglo).

Atributos públicos:

- `debug`: Indica si se desea o no realizar un seguimiento a la ejecución.

Métodos públicos:

- `BlockingQueue()`: Constructor de la clase.
- `~BlockingQueue()`: Destructor de la clase.
- `pop()`: Obtiene el primer elemento de la cola.
- `push()`: Introduce un elemento al final de la cola.
- `size()`: Obtiene el tamaño de la cola.
- `stopAll()`: Detiene todas las operaciones `pop()`.

2.2.3.15. Clase GlobalParameters

La clase `GlobalParameter` permite definir los parámetros de la aplicación (Figura 2.9).

Atributos públicos:

- `anonymousInterface`: Especifica la interface que se utiliza para el comunicador.
- Variables empleadas para depuración en la salida estándar de:
  - `debugConsensus`: Consenso.
  - `debugCRM`: Administrador de caída – recuperación.
  - `debugEAC`: Comunicador Ethernet.
  - `debugFD`: Detector de Fallos.
  - `debugRAC`: Comunicador anónimo.
  - `debugTwitter`: Aplicación de mensajería.
  - `EACFailureRate`: Porcentaje de falla del método `broadcast()` del comunicador.

- `initialSleep`: Tiempo de espera antes de iniciar.
- `numProcesses`: Número de procesos en el sistema.
- `racRebroadcastTimeoutMs`: Temporizador para reenvió de tramas.

Métodos públicos:

- `GlobalParameters()`: Constructor de la clase.
- `~GlobalParameters()`: Destructor de la clase.
- `parse()`: Convierte los argumentos de la línea de comandos a sus correspondientes tipos.
- `usage()`: Imprime un mensaje de uso en la salida estándar de error.

2.3. IMPLEMENTACIÓN

2.3.1. SISTEMA OPERATIVO

FreeBSD es el sistema operativo sobre el cual se realiza la implementación. Este sistema es de software libre cuyo código se distribuye bajo la licencia BSD. Su uso es muy común en servidores de alta demanda [29].

2.3.2. COMUNICADOR ANÓNIMO

La clase `EthernetAnonymousCommunicator` implementa el comunicador anónimo para el envío y recepción de tramas Ethernet. El anonimato se introduce en la capa enlace del modelo de referencia ISO/OSI (*International Organization for Standardization/Open System Interconnection*), al configurar la dirección MAC (*Media Access Control*) origen a `04:04:04:04:04:04` y la dirección MAC destino de *broadcast* `FF:FF:FF:FF:FF:FF` en la trama Ethernet. Para no afectar a otros servicios de red, el tipo de trama se configura como experimental `0x88b5` [30].

En el Código 2.7 se muestra la configuración de las direcciones MAC (líneas 2 - 4), mientras que en las líneas 1 y 4 se define el tipo de trama Ethernet.

En la Figura 2.10 se muestra el formato del mensaje de aplicación y su encapsulación en la trama Ethernet. El campo “Etiqueta” de protocolo se utiliza para identificar al

protocolo, y el comunicador anónimo entregue el mensaje a la aplicación correcta. El campo “Tipo de mensaje” se utiliza para definir los diferentes mensajes que la aplicación puede manejar, por ejemplo: HB y ACK_HB.

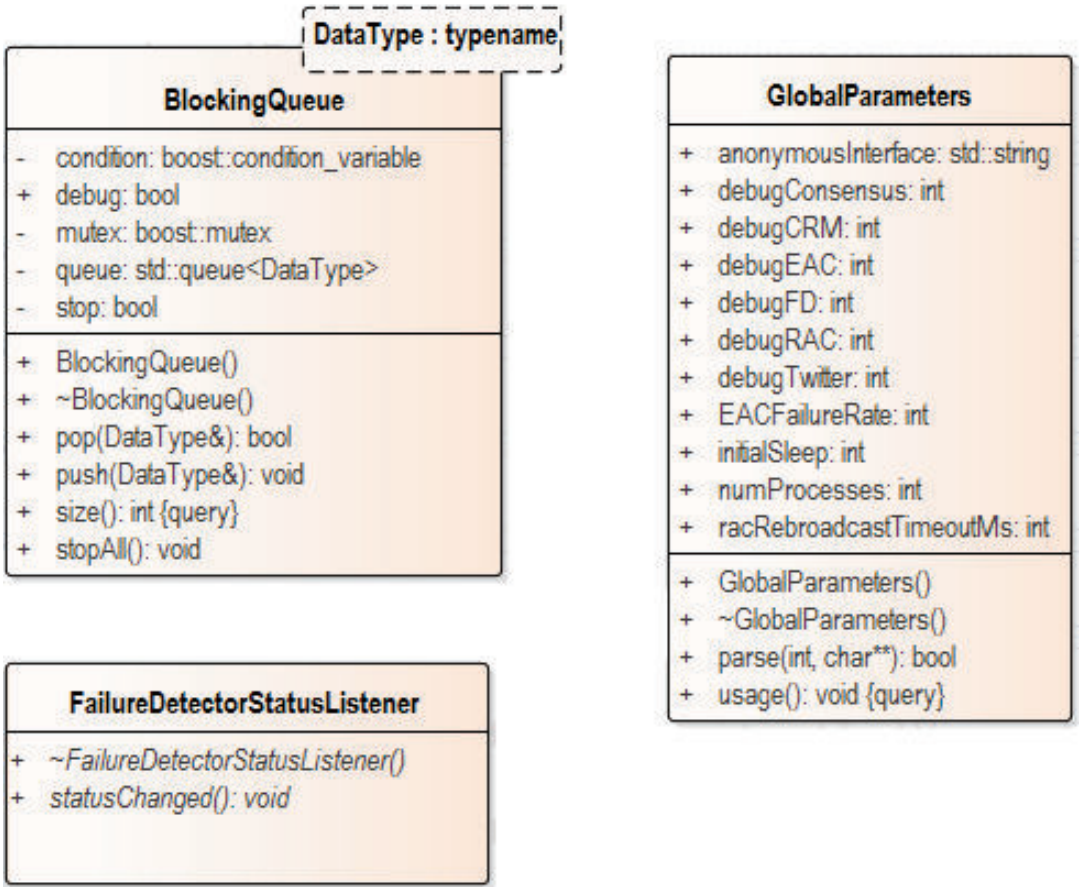

Figura 2.9. Diagrama de clases auxiliares

```
1  #define AC_ETHERTYPE 0x88b5
2  memset(&etherHeader.ether_dhost, 0xff, ETHER_ADDR_LEN);
3  memset(&etherHeader.ether_shost, 0x04, ETHER_ADDR_LEN);
4  etherHeader.ether_type = htons(AC_ETHERTYPE);
```

Código 2.7. Fragmento de código del método broadcast()

El comunicador anónimo implementa dos hilos para enviar mensajes: hilo de aplicación e hilo de envío (Figura 2.11). El hilo de aplicación es quien genera los mensajes, mientras que el hilo de envío recibe mensajes desde la aplicación e interactúa con PCAP para realizar el envío del mensaje a la red. Como se puede

observar el diagrama secuencial de la Figura 2.11 consta de dos fases: fase de inicialización y fase de envío.

```
1  EthernetAnonymousCommunicator(int serviceTag,
2  MessageFactory& messageFactory,
3  const std::string& networkInterface,
4  MessageQueue& messageReceivingQueue):
5  messageReceivingQueue(messageReceivingQueue) {
6  llEACUsers.llEAC = new
   PCAPEthernetAnonymousCommunicator(networkInterfaceName);
   receivingThread =
7  boost::thread(&EthernetAnonymousCommunicator::messageReceiver, this);
```

Código 2.8. Constructor de la clase EthernetAnonymousCommunicator

1. **Fase de inicialización**

1.1. Instancia un objeto `EthernetAnonymousCommunicator` llamando a su constructor (Código 2.8), el cual recibe como parámetros de entrada: el identificador de servicio (línea 1), factoría de mensajes (línea 2), la interfaz por la que se envía lo mensajes (línea 3) y el arreglo correspondiente a la cola para recepción de mensajes (línea 4).

1.2. Inicializa el arreglo `messageReceivingQueue` del Código 2.8 en la línea 5.

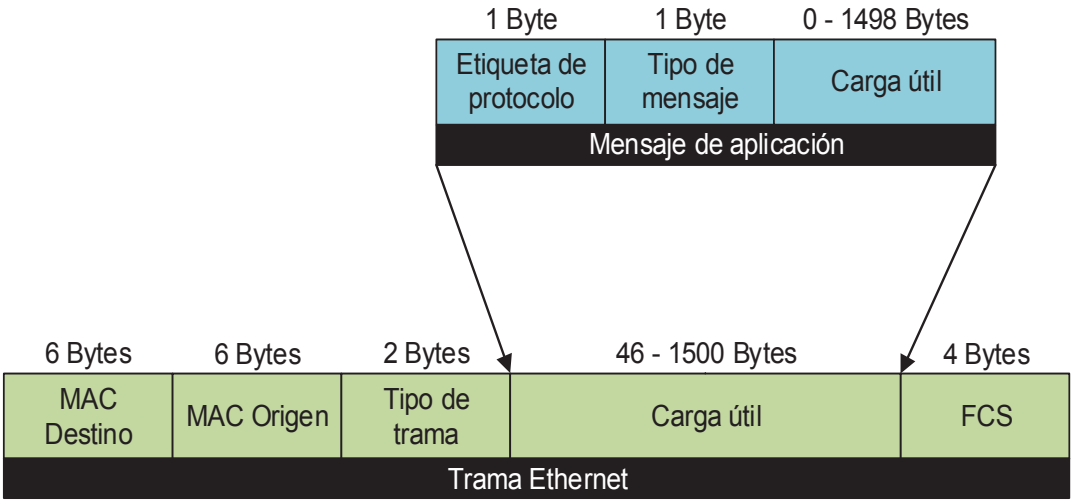

Figura 2.10. Formato de mensaje

1.3. Inicialización de PCAP al llamar a la función `pcap_open_live()` desde un objeto de la clase `BPFEthernetAnonymousCommunicator`, como se muestra en el Código 2.9.

1.4. Crea un hilo de recepción (Código 2.8, línea 7).

1.5. Cuando la aplicación lo requiera, se puede extraer un mensaje desde el arreglo `messageReceivingQueue` con el método `pop()`. Si el arreglo está vacío, se queda esperando hasta que la aplicación realice un llamado al método `broadcast()`.

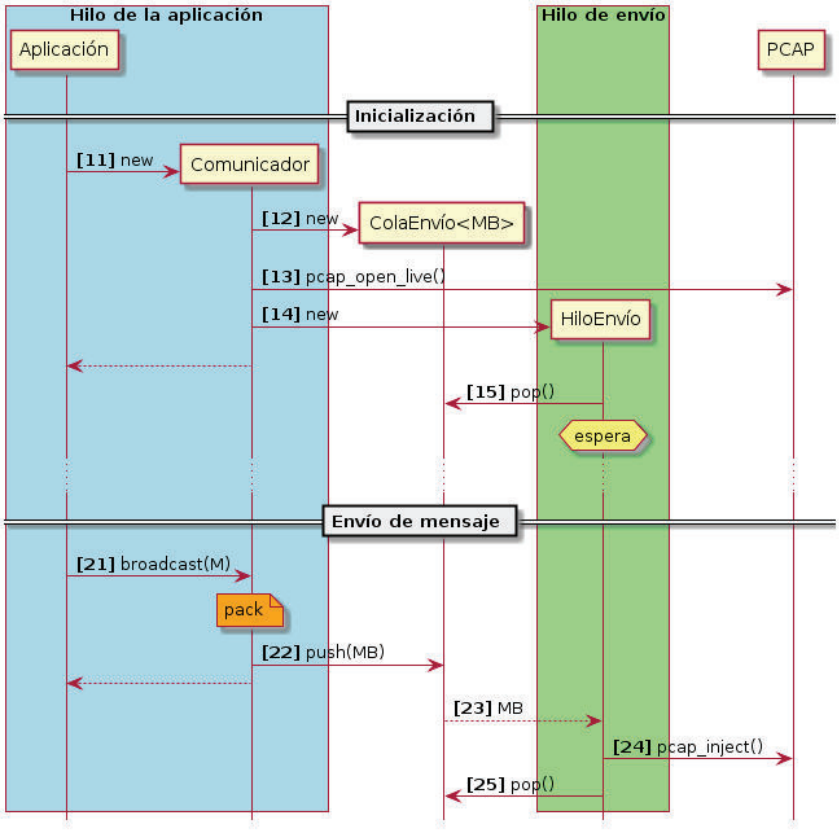

Figura 2.11. Diagrama secuencial de envío de mensajes

2. Fase envío de mensaje

2.1. En el Código 2.10 se implementa el método `broadcast()` de la clase `EthernetAnonymousCommunicator` para construir la cabecera y carga útil de la trama Ethernet. El comunicador serializa el mensaje con la función `pack()` (línea 2), y configura los campos de la trama (líneas 15), para

finalmente llamar al método `broadcast()` de la clase `LowLevelEthernetAnonymousCommunicator` y realizar la difusión del mensaje.

2.2. El mensaje serializado se guarda en el arreglo de envío del método `broadcast()` de la clase `LowLevelEthernetAnonymousCommunicator` (Código 2.11), y, una vez guardado el mensaje, la aplicación retoma el control.

2.3. El método `packetSender()` de la clase `PCAPEthernetAnonymousCommunicator` está a la espera de que llegue un mensaje al arreglo de envío, para luego realizar el llamado al método `pcap_inject()` (Código 2.11 líneas 4 - 5).

2.4. Inserta la trama Ethernet mediante la función `pcap_inject()` (Código 2.11, línea 6).

2.5. Se repite la secuencia desde 1.5 a la espera de un nuevo mensaje. De esta manera el hilo de envío funcionará indefinidamente.

```
1  if ((pcapfd =
   pcap_open_live(networkInterface.c_str(),BUFSIZ,1,0,pcap_errbuf)) == NULL) {
2  std::stringstream s;
3  s << "Error opening LIBPCAP: " << pcap_errbuf;
4  throw std::runtime_error(s.str());
5  }
```

Código 2.9. Inicialización de PCAP

La recepción de mensajes se ilustra en el diagrama de secuencia de la Figura 2.12, como se puede observar la recepción de mensajes emplea tres hilos.

La razón para utilizar tres hilos en lugar de dos como en el envío de mensajes, es que la llamada a `pcap_inject()` se realiza una única vez. En otras palabras, cualquier mensaje de aplicación es enviado una única vez por `pcap_inject()`. Al momento de recibir los mensajes, puede haber varios protocolos de aplicación y cada uno no puede llamar a `pcap_loop()` por separado. Por lo tanto, el hilo de PCAP será compartido por todos los protocolos de aplicación y cada uno de ellos implementa un

hilo de recepción. El diagrama secuencial de la Figura 2.12 consta de dos fases: fase de inicialización y fase de recepción de mensaje.

```
1  void EthernetAnonymousCommunicator::broadcast(const Message* const message){
2  const MessageBuffer* mb = message->pack();
3  struct ether_header etherHeader;
4  memset(&etherHeader.ether_dhost, 0xff, ETHER_ADDR_LEN);
5  memset(&etherHeader.ether_shost, 0x04, ETHER_ADDR_LEN);
6  etherHeader.ether_type = htons(AC_ETHERTYPE);
7  EACHeader eacHeader;
8  eacHeader.serviceTag = serviceTag;
9  eacHeader.payloadDataLength = 0;
10 const MessageBuffer* mb1 = mb;
11 while (mb1 != 0) {
12        eacHeader.payloadDataLength += mb1->length;
13        mb1 = mb1->prev;
14 }
15 eacHeader.payloadDataLength = htons(eacHeader.payloadDataLength);
16 llEAC->broadcast(etherHeader, eacHeader, mb);
```

Código 2.10. Fragmento de código del método broadcast()

```
1  void PCAPEthernetAnonymousCommunicator::packetSender(){
2  MessageBuffer* mb;
3  int ret;
4  do {
5        if (sendingQueue.pop(mb)) {
6              ret = pcap_inject(pcapfd, mb->data, mb->length);
```

Código 2.11. Fragmento de código del método packetSender()

1. Fase de inicialización

1.1. Instancia el arreglo de recepción `messageReceivingQueue` para luego pasarle como argumento de entrada al Código 2.8 en la línea 4.

1.2. Instancia el comunicador anónimo, llamando al constructor de `EthernetAnonymousCommunicator` (Código 2.8, línea 1).

1.3. El constructor del comunicador llama al método `pcap_open_live()` desde un objeto de la clase `BPFEthernetAnonymousCommunicator` (Código 2.9, línea 1).
1.4. Instancia un arreglo de recepción.
1.5. Instancia un hilo de recepción.
1.6. Instancia un hilo de PCAP.
1.7. El hilo de recepción espera hasta que llegue un mensaje al arreglo de recepción mediante la función `pop()`.

```
1   if (pcap_compile(pcapfd, &bpfProgram, "ether broadcast", 0,
2   PCAP_NETMASK_UNKNOWN) == -1) {
3         std::stringstream s;
4         s << "[EAC] pcap_compile(): " << pcap_geterr(pcapfd);
5         throw std::runtime_error(s.str());
6   }
7   if (pcap_setfilter(pcapfd, &bpfProgram) == -1) {
8         std::stringstream s;
9         s << "[EAC] pcap_setfilter(): " << pcap_geterr(pcapfd);
10        throw std::runtime_error(s.str());
11  }
12  if (pcap_loop(pcapfd, -1, lleac_pcap_handler, (u_char*) this) == -1) {
13        std::stringstream s;
14        s << "[EAC] pcap_loop(): " << pcap_geterr(pcapfd);
15        throw std::runtime_error(s.str());
16  }
```

Código 2.12. Llamado al método pcap_setfilter() y pcap_loop()

1.8. El hilo de PCAP espera por tramas con los métodos `pcap_setfilter()` y `pcap_loop()`. En el Código 2.12 se muestra la implementación de estos métodos; primero compila un filtro BPF, especificando recibir todas las tramas broadcast de Ethernet con "`ether broadcast`" (línea 1). A continuación lo instala (línea 7), y en la línea 12 se especifica a la función `pcap_loop()` la función `lleac_pcap_handler` que será la que se llama cada vez que se reciba una trama.

2. Fase de recepción de mensajes

2.1. La aplicación intenta extraer un mensaje desde el arreglo de recepción con el método `pop()`.

2.2. El Código 2.13 muestra las instrucciones ejecutadas cuando una trama llega. Primero se llama al método `lleac_pcap_handler` y verifica que el paquete haya llegado completo (líneas 2-3). Después realiza el llamado al método `distributePacket()`, para pasar la trama a todos los hilos de recepción que cada aplicación implementa (Código 2.14, líneas 1-10).

```
1  static void lleac_pcap_handler(u_char *user, const struct pcap_pkthdr
2  *header, const u_char *packet){
3        if (header->caplen != header->len) {
4              return;
5        }
6        PCAPEthernetAnonymousCommunicator* lleac =
         (PCAPEthernetAnonymousCommunicator*) user;
7        lleac->distributePacket(packet, header->caplen);
8  }
```

Código 2.13. Fragmento del método lleac_pcap_handler

2.3. En la línea 9, del Código 2.7, el hilo de PCAP inserta en la cola de recepción la carga útil de la trama Ethernet.

```
1  for (mbqli = receivingQueues[eacHeader->serviceTag]->begin(); mbqli !=
2  receivingQueues[eacHeader->serviceTag]->end(); mbqli++) {
3        MessageBufferQueue* q = *mbqli;
4        MessageBuffer* mb = new MessageBuffer;
5        mb->length = ntohs(eacHeader->payloadDataLength);
6        mb->data = new u_int8_t[mb->length];
7        memcpy(mb->data, payload, mb->length);
8        mb->prev = 0;
9        q->push(mb);
10 }
```

Código 2.14. Fragmento del método distributePacket

2.4. El hilo de recepción se desbloquea de la llamada `pop()` realizada en 1.7 y obtiene del arreglo de recepción el mensaje (Código 2.15, línea 4). Luego, realiza la decodificación del mensaje.

2.5. El hilo de recepción insertara el mensaje decodificado en el arreglo de recepción correspondiente.

2.6. La aplicación que realizó `pop()` en la secuencia 2.1 obtiene el mensaje.

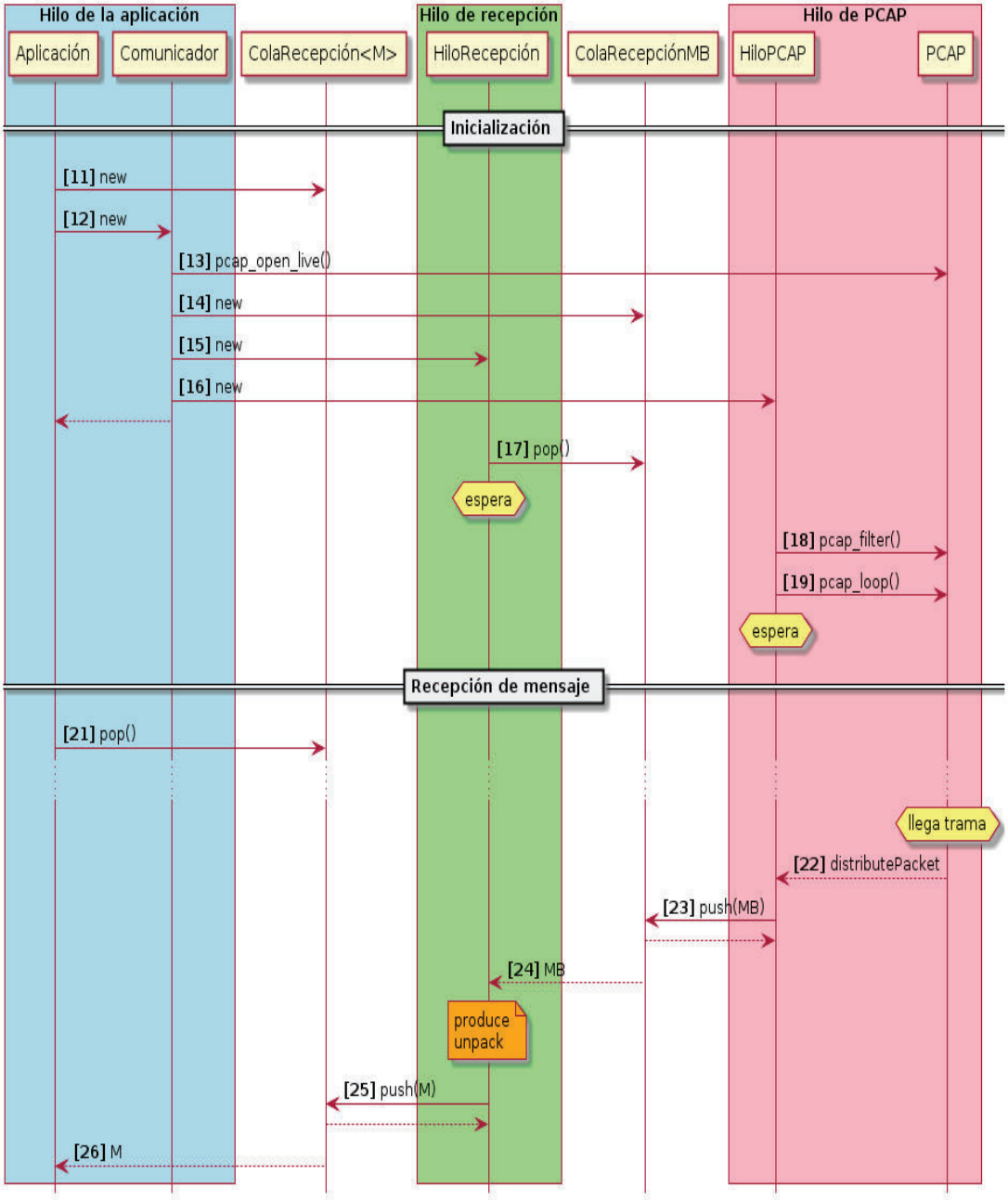

Figura 2.12. Diagrama secuencial de recepción de mensajes

```
1  void EthernetAnonymousCommunicator::messageReceiver(){
2          do {
3                  MessageBuffer* mb;
4                  if (messageBufferReceivingQueue.pop(mb)) {
5                          Message* m = messageFactory.produce(mb);
6                          if (m != 0)
7                                  messageReceivingQueue.push(m);
8                          delete (u_int8_t*) mb->data;
9                          delete mb;
10                 }
11         } while (! stopReceivingThread);
```

Código 2.15. Fragmento del método messageReceiver()

3.3.3. ALMACENAMIENTO EN DISCO DURO

La clase `CrashRecoveryManager` permite la recuperación y almacenamiento de las variables en el disco duro. El fichero que contiene las variables se almacena en el directorio `/tmp/` con la extensión `.rec`. También se utiliza un identificador para definir la cabecera del fichero con "`CRASH-RECOVERY01`" (Código 2.16, líneas 1-3).

```
1  #define STABLE_DIR_PATH "/tmp/"
2  #define STABLE_FILENAME_PATTERN "XXXXXXXXX.rec"
3  #define MAGIC_STRING "CRASH-RECOVERY01"
```

Código 2.16. Definición de información para almacenamiento

El constructor de la clase se muestra en el Código 2.17. En las líneas 1 y 2 se puede observar que recibe como parámetro de entrada `processTag` para identificar al proceso. Luego, en la línea 3, llama al método `locateRecoveryFile()` para localizar al fichero que contiene el valor de las variables. Si se encuentra el fichero, se llama a la función `loadRecoveryData`() para recuperar datos de las variables (líneas 4-7). De lo contrario, se crea un nuevo fichero con la función `initializeRecoveryFile()` (líneas 9-10).

El método `locateRecoveryFile()` localiza al fichero en el directorio `/tmp/` mediante el Código 2.18. Se utiliza `regex` para definir un filtro de búsqueda, en este

caso es el fichero terminado en `.rec` (línea 2). En la línea 3 se asigna el valor de -1 al descriptor del fichero `recoveryfile`. Luego, utilizando un iterador para recorrer todos los ficheros del directorio `/tmp/` (líneas 4 y 5), se analiza cada uno y se lo intenta abrir con la función `tryOpen()` (línea 8). Si el descriptor del fichero cambia en la función `tryOpen()`, el método retorna (líneas 9 y 10).

```
1  CrashRecoveryManager::CrashRecoveryManager(string processTag) :
2  processTag(processTag){
3         locateRecoveryFile();
4         if (recoveryFile != -1) {
5               loadRecoveryData();
6               recoveredProcess = true;
7         }
8         else {
9               initializeRecoveryFile();
10              recoveredProcess = false;
11        }}
```

Código 2.17. Método CrashRecoveryManager()

```
1  void CrashRecoveryManager::locateRecoveryFile(){
2         regex filenameFilter(".*\\.rec$");
3         recoveryFile = -1;
4         for (directory_iterator idir(STABLE_DIR_PATH); idir !=
5               directory_iterator(); idir++) {
6               if (is_regular_file(idir->status()) && regex_match(idir
7                     ->path().filename().string(), filenameFilter)) {
8                           tryOpen(idir->path().generic_string());
9                           if (recoveryFile != -1)
10                                return;
11              }}}
```

Código 2.18. Método locateRecoveryFile()

Una parte del método `tryOpen()` se muestra en el Código 2.19. En las líneas 1 a 7 se intenta abrir el fichero en modo exclusivo (ningún otro proceso puede hacer uso de él). Si la cabecera del fichero que consta de `MAGIC_STRING` y `processTag` es correcta,

entonces se asigna el nombre del fichero a `recoveryFilename` (línea 10). De lo contrario, se cierra el fichero y el descriptor con valor -1. Si el fichero no pudo abrirse en modo exclusivo (línea 17), entonces se asigna el valor de -1 al descriptor y un código de error (líneas 17-24). Parte del método `loadRecoveryData()` se muestra en el Código 2.20. En la línea 1 carga el nombre de la variable desde el descriptor `vdesc`. Este nombre de variable no debe terminar en cero. De lo contrario, asigna otro nombre más largo (línea 4). Luego, se asigna el tamaño de los datos, el identificador de aplicación, los datos en sí, y se añade a la lista de variables (líneas 5-8).

```
1  if ((recoveryFile = open(filename.c_str(), O_RDWR)) == -1)
2          throw new CrashRecoveryException((string("tryOpen: ") +
3          strerror(*__error())).c_str());
4  if (flock(recoveryFile, LOCK_EX | LOCK_NB) == 0) {
5          if (read(recoveryFile, &header, sizeof(header)) != sizeof(header))
6                  throw new CrashRecoveryException((string("tryOpen: ") +
7                  strerror(*__error())).c_str());
8          if (memcmp(header.magic, MAGIC_STRING, sizeof(header.magic)) == 0 &&
9          strncmp(header.tag, processTag.c_str(), processTag.size()) == 0) {
10                 recoveryFilename = filename;
11         }
12         else {
13                 close(recoveryFile);
14                 recoveryFile = -1;
15         }
16 }
17 else {
18         int errorCode = *__error();
19         close(recoveryFile);
20         recoveryFile = -1;
21         if (errorCode != EWOULDBLOCK)
22                 throw new CrashRecoveryException((string("tryOpen: ") +
23                 strerror(errorCode)).c_str());
24 }
```

Código 2.19. Fragmento del método tryOpen()

```
1  if (vdesc.name[sizeof(vdesc.name) - 1] == '\0')
2         variable->name = string(vdesc.name);
3  else
4         variable->name=string(vdesc.name, vdesc.name + sizeof(vdesc.name)-1);
5  variable->size = vdesc.size;
6  variable->appVariable = 0;
7  variable->data = new char[variable->size];
8  variables.push_back(variable);
```

Código 2.20. Fragmento del método loadRecoveryData()

```
1  char filename[sizeof(STABLE_DIR_PATH) + sizeof(STABLE_FILENAME_PATTERN)];
2  if ((recoveryFile = mkstemps(filename, 4)) == -1 ||
3         flock(recoveryFile, LOCK_EX) == -1) {
4         if (recoveryFile != -1) {
5                close(recoveryFile);
6                remove(filename);
7         }
8  if (write(recoveryFile, &header, sizeof(header)) == -1 ||
9         fsync(recoveryFile) == -1) {
10        close(recoveryFile);
11        remove(filename);
12        throw new CrashRecoveryException(reason);
13 }
14
```

Código 2.21. Fragmento del método initializeRecoveryFile()

El método responsable de la inicialización de un fichero, en el caso que no exista, es `initializeRecoveryFile`(). En el Código 2.21 se muestra un fragmento de este método. En las líneas 2 y 3 se crea un nombre de fichero aleatorio con la función `mkstemp()` [31], y se intenta obtener el descriptor del fichero. En la línea 8 se escribe el fichero con el nombre generado en el paso anterior. Si no es posible la creación del fichero, cierra y borra lo que se haya podido crear y lanza una excepción (líneas 8-13). El método `recoverVariable()` permite recuperar el valor de una variable para una aplicación mediante `appVariable`. El Código 2.22 muestra las instrucciones de este método. La línea 2 itera sobre todas las variables recuperadas. Una vez que se

encuentra la deseada, se copia su valor a `appVariable` y se devuelve `true` (líneas 3-6); de lo contrario, devuelve `false` (línea 9).

```
1  bool CrashRecoveryManager::recoverVariable(string name, void* appVariable){
2          for (auto variable : variables) {
3                  if (variable->name == name) {
4                          variable->appVariable = appVariable;
5                          memcpy(appVariable, variable->data, variable->size);
6                          return true;
7                  }
8          }
9          return false;
10 }
```

Código 2.22. Método recoverVariable()

```
1  if (write(recoveryFile, appVariable, variable->size) != variable->size ||
2          fsync(recoveryFile) == -1)
3          throw new CrashRecoveryException((string("synchronize: ") +
4          strerror(*__error())).c_str());
5  vdesc.valid = true;
6  if (lseek(recoveryFile, vdescFilepos - sizeof(vdesc), SEEK_SET) == -1 ||
7          write(recoveryFile, &vdesc, sizeof(vdesc)) != sizeof(vdesc) ||
8          fsync(recoveryFile) == -1)
9          throw new CrashRecoveryException((string("synchronize: ") +
10         strerror(*__error())).c_str());
```

Código 2.23. Fragmento del método synchronize()

El método `synchronize()` permite copiar el valor de las variables en el disco duro. Esta acción se realiza en dos pasos. El primero consiste en escribir la cabecera y el contenido de la variable, pero su cabecera es marcada como invalida. En el segundo paso se marca la cabecera como válida. De esta forma se asegura una acción atómica sobre el fichero.

En el Código 2.23, en las líneas 1 a 4, se muestra la implementación del primer paso, mientras que en las líneas 5 a 8 el segundo paso.

```
1  FailureDetector::FailureDetector() :
2  stopTask1Thread(false),
3  stopTask2Thread(false)
4  {
5          messageFactory.registerType(FDHeartbeatMessage::type_id,
           (UnpackerFunction) new FDHeartbeatMessage());
6          messageFactory.registerType(FDHeartbeatAckMessage::type_id,
           (UnpackerFunction) new FDHeartbeatAckMessage());
7          communicator = new
           EthernetAnonymousCommunicator(FAILURE_DETECTOR_PROTOCOL_ID,
           messageFactory, "", incomingMessages);
8          timeout = 0;
9          recover();
10         next_ack = 1;
11         quantity = 0;
12         initialize_leader_status();
13         task2Thread = boost::thread(&FailureDetector::task2, this);
14         task1Thread = boost::thread(&FailureDetector::task1, this);
15 }
```

Código 2.24. Método FailureDetector()

3.3.4. DETECTOR DE FALLOS AΩ' CON CAÍDA–RECUPERACIÓN

En este apartado se destacan los métodos más importantes del Detector de Fallos. El Código 2.24 muestra la implementación del constructor `FailureDetector()`.

Primero inicializa las comunicaciones con la creación de la factoría de mensajes y registro de los mensajes que se van a utilizar en el comunicador anónimo (líneas 5-7). En la línea 8 se inicializa el valor de la variable $TIMEOUT$. Llama a la función `recover()` para recuperar el valor de las variables hasta antes del fallo, o crea un nuevo fichero que contendrá los datos de recuperación (línea 9). Las variables $next\_ack$ y $quantity$ inician con valor cero (líneas 10-11). La llamada al método `initialize_leader_status()` y la creación e inicio de los hilos `task2Thread` y `task1Tread` corresponden a las líneas 3 a 10 del pseudocódigo de la Figura 2.3.

El método `~FailureDetector()` que se muestra en el Código 2.25 detiene la tarea 1 (línea 2 - 3) y la tarea 2 (línea 4 - 6).

```
1  FailureDetector::~FailureDetector(){
2          stopTask1Thread = true;
3          task1Thread.join();
4          stopTask2Thread = true;
5          incomingMessages.stopAll();
6          task2Thread.join();
7  }
```

Código 2.25. Método ~FailureDetector()

```
1  if (leader) {
2          seq++;
3          remove_old_HB_ACK();
4  #ifdef DEBUG_FD
5          if (globalParameters.debugFD > 100)
6          std::cerr << "[FD] Broadcast HB(" << seq << ")" << std::endl;
7  #endif {
8          communicator->broadcast(new FDHeartbeatMessage(seq));
9          }
10 }
```

Código 2.26. Fragmento de código del método task1()

El fragmento de código de la función `task1()`, que implementa las líneas 12 a 15 del pseudocódigo de la Figura 2.3, se muestra en el Código 2.26. Como se puede observar en la línea 8, se implementa la difusión del mensaje HB. Las líneas 4 a 7 permiten realizar un seguimiento de la ejecución del programa si se define la variable `DEBUG_FD` con un valor mayor a 100.

El Código 2.27 implementa las instrucciones de las líneas 17 a 19 del pseudocódigo de la Figura 2.3. Este segmento de código lo ejecuta un proceso líder, en donde, recorre la lista de mensajes ACK_HB y revisa si se cumple la condición $s \leq seq_i \leq s'$ (líneas 8 a 17). Se utiliza una variable `quantity_aux` para verificar si cambia con respecto al valor anterior (línea 22) y se establece una bandera que permite notificar a los observadores. Las líneas de código entre `#ifdef DEBUG_FD` y `#endif` son salidas por consola con la finalidad de realizar depuración de una ejecución.

```
1  if (leader) {
2         FDHeartbeatAckMessageList::iterator i;
3  #ifdef DEBUG_FD
4         if (globalParameters.debugFD > 90)
5         std::cerr << seq << ": ";
6  #endif
7         int quantity_aux = 0;
8         for (i = receivedHBAckMessages.begin(); i !=
          receivedHBAckMessages.end(); i++) {
9               FDHeartbeatAckMessage *message = *i;
10              if (message->getNextAck() <= seq && seq <= message->getSeq()) {
11 #ifdef DEBUG_FD
12              if (globalParameters.debugFD > 90)
13                    std::cerr << " (" << message->getNextAck() << "," <<
                            message->getSeq() << ")";
14 #endif
15              quantity_aux++;
16              }
17        }
18 #ifdef DEBUG_FD
19        if (globalParameters.debugFD > 90)
20              std::cerr << std::endl;
21 #endif
22        if (quantity_aux != quantity)
23              notifyListeners = true;
24        quantity = quantity_aux;
25 }
```

Código 2.27. Fragmento de código del método task1()

Las instrucciones que ejecutará un proceso no líder en la Tarea 1 se muestran en el Código 2.28. Verifica el tamaño de la lista de los nuevos mensajes HB_ACK recibidos (línea 2). Si el tamaño es igual a cero quiere decir que no hay ningún líder en el sistema (línea 11).

Entonces el proceso asume que no hay líderes en el sistema y se declara a sí mismo como un líder (línea 11). Antes de cambiar el valor de la variable `leader` en la línea

11, primero notifica a los observadores del cambio. Los segmentos de código entre `#ifdef DEBUG_FD` y `#endif` se utilizan para la depuración de la ejecución.

```
1  else {
2        if (newlyreceivedHBAckMessages.size() == 0) {
3  #ifdef DEBUG_FD
4              if (globalParameters.debugFD > 95) {
5                    for (auto message : receivedHBAckMessages)
6                          if (message->getSeq() > seq)
7                                seq = message->getSeq();
8              }
9  #endif
10             notifyListeners = true;
11             leader = true;
12       }
13       newlyreceivedHBAckMessages.clear();
14 }
```

Código 2.28. Fragmento de código del método task1()

Los fragmentos de código, que se muestran en el Código 2.29 y en el Código 2.30, indican las instrucciones que la Tarea 2 debe ejecutar. El Código 2.29 es la implementación de las líneas 25 a 29 del pseudocódigo de la Figura 2.3. Si se recibe un mensaje HB (línea 1), si el proceso es líder y si además el número de secuencia del mensaje es mayor a `next_ack` (línea 7), entonces realiza la difusión de un mensaje ACK_HB con los valores de `next_ack` y s_k (correspondiente al valor obtenido de `hbMessage->getSeq()`) (línea 12). Seguidamente, y por último, se actualiza el valor de la variable `next_ack` (línea 13).

Las líneas 30 y 31 del pseudocódigo de la Figura 2.3 son implementadas en el Código 2.30. Si el mensaje recibido es de tipo ACK_HB (línea 1), tal que `hbAckMessage->getNextAck()< seq` y además el proceso es líder (línea 11), entonces actualiza el valor de la variable `timeout` (línea 12) y se la guarda en el disco duro (línea 13). Las líneas 6 a 10 realizan control de concurrencia entre la Tarea 1 y la Tarea 2, evitando que ambas tareas accedan a `receivedHBAckMessages` al mismo tiempo.

```
1  case FDHeartbeatMessage::type_id:{
2          FDHeartbeatMessage* const hbMessage = (FDHeartbeatMessage* const)
           message;
3  #ifdef DEBUG_FD
4          if (globalParameters.debugFD > 100)
5                  std::cerr << "[FD] Receive HB(" << hbMessage->getSeq() << ")"
                   << std::endl;
6  #endif
7          if (hbMessage->getSeq() >= next_ack && leader) {
8  #ifdef DEBUG_FD
9          if (globalParameters.debugFD > 100)
10                 std::cerr << "[FD] Broadcast HBACK(" << next_ack << "," <<
                         hbMessage->getSeq() << ")" << std::endl;
11 #endif
12                 communicator->broadcast(new FDHeartbeatAckMessage(next_ack,
                   hbMessage->getSeq()));
13                 next_ack = hbMessage->getSeq() + 1;
14         }
15 }
16 break;
```

Código 2.29. Fragmento de código del método task2()

El método `recover()` se implementa en el Código 2.31. Este método recupera el valor de la variable `timeout`, haciendo uso de las funciones `isRecoveredProcess()` y `recoverVariable()` del Administrador de Caída-recuperación (línea 3). Si la variable no se recupera exitosamente, se inicializa `timeout` y se registra la variable con el Administrador Caída-recuperación (líneas 4 - 6).

El método `initialize_leader_status()` se implementa en el Código 2.32. Este método específica las instrucciones correspondientes a las líneas 3 a 7 del pseudocódigo de la Figura 2.3.

Si existen mensajes en la cola `incomingMessages` (línea 14), esto significa que hay al menos un líder en el sistema, por lo que no se considera a sí mismo líder (línea 15). El fragmento de código de las líneas 5 a 13 se introdujo con la finalidad de acelerar la

convergencia del sistema. En las pruebas preliminares se notó que tardaba mucho el Detector de Fallos en estabilizar la variable $quantity$. Es por ello que se introdujo un tiempo de espera con un periodo de tiempo mayor hasta recibir mensajes de los líderes, y así converger con mayor rapidez.

```
1  case FDHeartbeatAckMessage::type_id: {
2          FDHeartbeatAckMessage* const hbAckMessage = (FDHeartbeatAckMessage*
           const) message;
3  #ifdef DEBUG_FD
4          if (globalParameters.debugFD > 100)
5                  std::cerr << "[FD] Receive HBACK(" << hbAckMessage-
                   >getNextAck() << "," << hbAckMessage->getSeq() << ")" <<
                   std::endl;
6          {
7          boost::lock_guard<boost::mutex>  m(receivedHBAckMessagesMutex);
8          receivedHBAckMessages.push_back(hbAckMessage);
9          newlyreceivedHBAckMessages.push_back(hbAckMessage);
10         }
11         if (hbAckMessage->getNextAck() < seq && leader) {
12                 timeout++;
13                 crManager->synchronize(&timeout);
14         }
15 }
16 break;
```

Código 2.30. Fragmento de código del método task2()

```
1  void FailureDetector::recover(){
2          crManager = new CrashRecoveryManager("FAILURE_DETECTOR");
3          if (! crManager->isRecoveredProcess() || ! crManager
           >recoverVariable("TIMEOUT", &timeout)) {
4                  timeout = 1;
5                  crManager->registerVariable("TIMEOUT", false, &timeout,
                   sizeof(timeout));
6                  crManager->synchronize(&timeout);
7          }
8  }
```

Código 2.31. Método recover()

```
1  void FailureDetector::initialize_leader_status(){
2         timeout++;
3         crManager->synchronize(&timeout);
4         usleep(timeout * TIME_SCALE);
5  #ifdef DEBUG_FD
6         if (globalParameters.debugFD > 95) {
7               int extra_delay = EXTRA_DELAY;
8               while (incomingMessages.size() == 0 && extra_delay > 0) {
9                     usleep(timeout * TIME_SCALE);
10                    extra_delay--;
11              }
12        }
13 #endif
14        if (incomingMessages.size() > 0)
15              leader = false;
16        else
17              leader = true;
18 }
```

Código 2.32. Función initialize_leader_status()

CAPÍTULO 3
ALGORITMO DE CONSENSO

3.1. DISEÑO

3.1.1. INTRODUCCIÓN

Los procesos que hagan uso del algoritmo de Consenso pueden llamar al método `propose()` para proponer un valor. Al final, esta función retorna el valor acordado por todos los procesos del sistema. Como se puede observar en la Figura 3.1 el algoritmo de Consenso utiliza el Detector de Fallos para conocer si es o no líder por medio del método `amILeader()` y el número de líderes que el Detector de Fallos estima que hay en el sistema mediante el método `numLeaders()`.

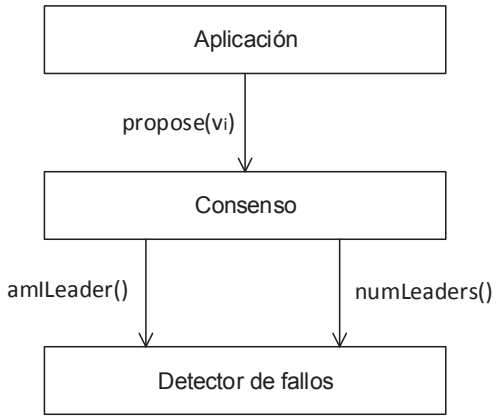

Figura 3.1. Interacción entre Aplicación, Consenso y Detector de Fallos

3.1.2. ALGORITMO

El algoritmo de Consenso a implementar se muestra en el pseudocódigo de la Figura 3.2. La función $propose$ tiene un parámetro de entrada v_i que corresponde al valor que el proceso p_i propone. Este valor puede ser de cualquier tipo, ya sea entero, cadena de caracteres, booleano, entre otros. En las líneas 1 a 3 se inicializa el número de ronda, se asigna el valor estimado de decisión al valor propuesto por el proceso p_i, y se inicia la Tarea 1 (*Task 1*). Una ronda es una iteración del bucle de las líneas 4 a 27 de la Tarea 1, en otras palabras, cada iteración del bucle aumenta en una unidad a r_i (línea 5). Cada ronda tiene tres fases: PH0, PH1 y PH2 que se detallan a continuación.

```
function propose(v_i):
Init:
1    r_i ← 0; est_i ← v_i;
2    decided_i ← false;
3    start Task 1.

Task 1:
4    repeat
5         r_i ← r_i + 1;
     % phase PH0
6         l_i ← D.leader_i;
7         if (l_i) then broadcast(PH0, r_i, est_i) end if;
8         wait until
8a             ((l_i ≠ D.leader_i)
                    ∨
8b             ((l_i) ∧ (D.quantity_i (PHO, r_i, −) received)
                    ∨
8c             ((PH1, r_i, −) received));
9         let rec_PH0_i = {est: (PH0, r_i, est) received};
10        let rec_PH1_i = {est: (PH1, r_i, est) received};
11        if (|rec_PH0_i ∪ rec_PH1_i | > 0) then
12             est_i ← min(rec_PH0_i ∪ rec_PH1_i)
13        end if
     % phase PH1
14        broadcast(PH1, r_i, est_i);
15        wait until |{(PH1, r_i, −) received}| > n/2;
16        if (est_i = est ∀ (PH1, r_i, est) received) then
17             agree_i ← true else agree_i ← false
18        end if;
     % phase PH2
19        broadcast(PH2, r_i, est_i, agree_i);
20        wait until |{(PH2, r_i, −) received}| > n/2;
21        if(∃(PH2, r_i, est, agree) received : agree = true) then
22             est_i ← est;
23        end if;
24        if(agree = true ∀ (PH2, r_i, −, agree) received) then
25             decided_i ← true; start Task 2
26        end if;
27   until decided_i;
28   return(est_i)

Task 2:
29   repeat each η units of time
30   broadcast(DEC, est_i);
31   end repeat

when a process p_i recovers:
32   wait until ((DEC, est) received);
33   return(est)
```

Figura 3.2. Algoritmo de Consenso

- Fase PH0: Esta fase utiliza el Detector de Fallos para consultar si el proceso es o no líder y guarda su valor en l_i (línea 6). En caso de ser un proceso líder, se difunde un mensaje PH0 con el número de ronda y el valor estimado de decisión a todos los demás procesos (línea 7). En la línea 8 el proceso espera hasta: un cambio de estado del Detector de Fallos (línea 8a); o, si el proceso es líder, esperar hasta recibir todos los mensajes PH0 de los demás procesos líder del sistema (línea 8b). El proceso conoce la cantidad de líderes mediante la variable $quantity_i$ que el Detector de Fallos provee. La línea 8c es para los procesos que no son líder y esperan a recibir un mensaje PH1 de un proceso líder (línea 8c). Los mensajes recibidos de la fase PH0 se guardan en el arreglo rec_PH0_i. Así mismo, los mensajes de la fase PH1 se guardan en rec_PH1_i (líneas 9 y 10). En la línea 11 se comprueba que los arreglos no estén vacíos, y en la línea 12 se asigna como nuevo valor estimado de decisión al menor de todos los valores de los dos arreglos. En la línea 12 se especifica escoger el mínimo de todos los valores. Sin embargo, este valor también puede ser, por ejemplo, el máximo. La única condición que debe cumplir es que debe ser un valor determinista (es decir, que siempre dé el mismo resultado en todos los procesos a partir de los mismos valores iniciales).

- Fase PH1: Inicia en la línea 14 con los procesos líderes y no líderes difundiendo un mensaje con el valor estimado de decisión adquirido en la fase PH0 y el número de ronda. Después, el proceso espera un mensaje PH1 de la mayoría de los procesos correctos del sistema (línea 15). Luego, verifica si en todos los mensajes PH1 recibidos se encuentra el mismo valor estimado de decisión (línea 16). Si es así, marca su variable $agree_i = true$. De lo contrario $agree_i = false$ (línea 17). De esta manera se informa en la fase PH2 si se ha recibido de todos los procesos el mismo valor estimado de decisión.

- Fase PH2: Los procesos líderes y no líderes difunden un mensaje PH2 con el número de ronda, el valor estimado de decisión y el valor de la variable $agree_i$ (línea 19). En la línea 20 el proceso espera los mensajes PH2 de la mayoría de los procesos. Si el valor de la variable $agree_i$ es $true$ en todos los mensajes PH2 recibidos, entonces se termina el bucle tomando la decisión, inicia la

Tarea 2 (*Task 2*) y la función retorna con el valor decidido (líneas 24 a 28). En el caso que no todas las variables $agree_i$ sean $true$, se toma como nuevo valor estimado de decisión el valor recibido de un proceso líder y se inicia una nueva ronda (líneas 21 a 23).

La función de la Tarea 2 de la Figura 3.2 es intentar acortar la duración de la decisión difundiendo un mensaje DEC, en el que informa la decisión tomada (líneas 29 a 31). De esta forma los procesos que se caen y se recuperan pueden integrarse rápidamente al sistema enterándose de los valores antes decididos (líneas 32 y 33).

3.1.3. DIAGRAMAS DE ACTIVIDAD

El diagrama de actividad que describe la función $propose$ descrita en el pseudocódigo de la Figura 3.2 se muestra en la Figura 3.3. Como se puede observar, esta función solo inicializa las variables para luego lanzar la Tarea 1 en un hilo que corre paralelamente al hilo principal.

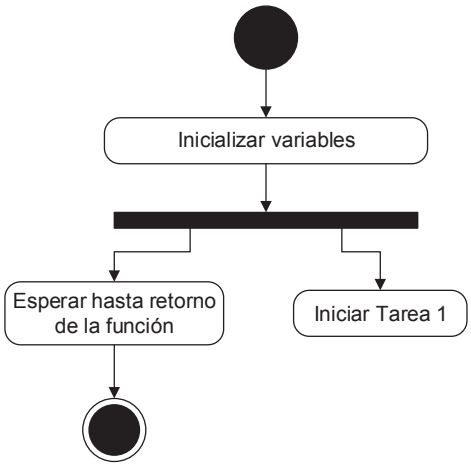

Figura 3.3. Diagrama de actividad de la función propose()

La Figura 3.4 muestra el diagrama de actividad de la Tarea 1 que modela el pseudocódigo de la Figura 3.2 de las líneas 4 a 28. Aumentar ronda implica sumar una unidad al número de ronda r_i. La fase PH0 corresponde a las líneas 6 y 7 y la evaluación de su condición se realiza en las líneas 8 a 13. De manera similar, la fase PH1 inicia en la línea 14 y su condición es evaluada en las líneas 15 a 18. Finalmente, la fase PH2 comienza con la difusión de un mensaje PH2 en la línea 19 y su condición

se realiza en las líneas 20 a 27. Si se ha decidido un valor se retorna la función de lo contrario se inicia de nuevo aumentando en una unidad el número de ronda r_i.

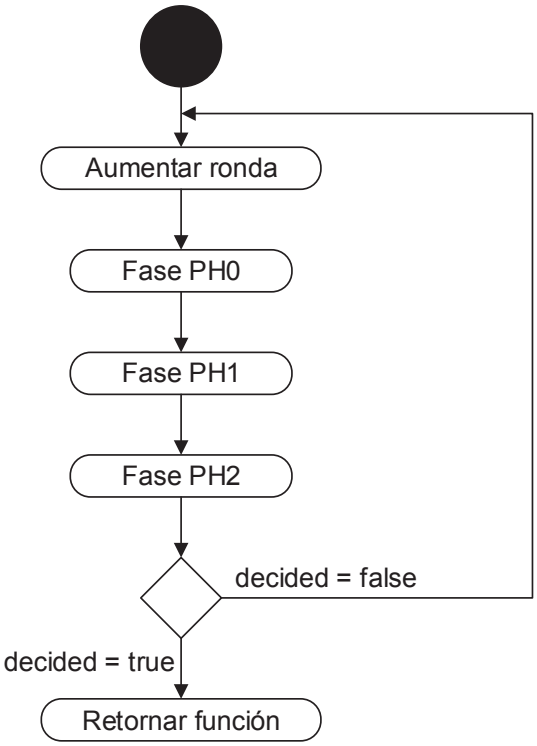

Figura 3.4. Diagrama de actividad de la Tarea 1

El diagrama de actividad que describe el comportamiento de los procesos que se recuperan se muestra en la Figura 3.6. Los procesos solo esperan por un mensaje DEC para tomar la decisión, tal y como se define en las líneas 32 y 33 del pseudocódigo de la Figura 3.2.

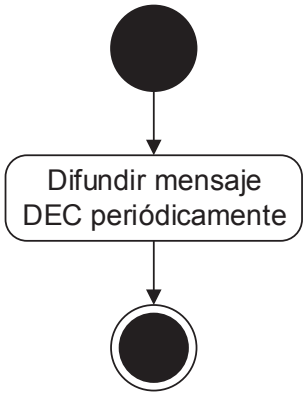

Figura 3.5. Diagrama de actividad de la Tarea 2

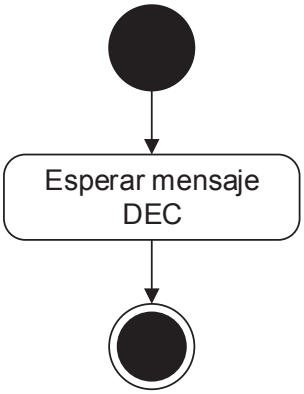

Figura 3.6. Diagrama de actividad de los procesos que se recuperan

El diagrama de actividad de la Tarea 2 definida en las líneas 29 a 31, se muestra en la Figura 3.5. Como se puede observar, el proceso difunde mensajes DEC periódicamente.

3.1.4. DIAGRAMA DE CLASES

La Figura 3.7 muestra el diagrama de clases correspondiente al algoritmo de Consenso. La clase `Consensus` implementa el algoritmo de la Figura 3.2, mientras que `Consensuable` es una clase que define una interfaz para todas las aplicaciones que requieran integrar Consenso a su programación.

A manera de ejemplo, la interfaz se implementa en la clase `TestType` que más adelante en el Capítulo 4 se utiliza en la aplicación para la demostración del funcionamiento de los protocolos. En el diseño se ha introducido la variable `runId`, la cual permite identificar la ejecución de Consenso. Es decir, una aplicación puede llamar varias veces al método `propose()` y para ello es necesario añadir un identificador, para conocer a que llamada corresponde el valor decidido.

Para que el algoritmo de Consenso funcione adecuadamente, se hace uso de algunas clases que se definieron para el Detector de Fallos. La clase `CrashRecoveryManager` se utiliza para almacenar y recuperar del disco duro los valores de las variables antes consensuadas. Las clases `FailureDetector` y `FailureDetectorStatusListener` se emplean para acceder al servicio que el Detector de Fallos provee y enterarse de los cambios que pueden ocurrir en este. Por

último, para registrar los mensajes que maneja Consenso, se hace uso de la factoría de mensajes mediante la clase `MessageFactory`.

3.1.4.1. Clase Consensus

Esta clase define las operaciones que Consenso debe ejecutar (Figura 3.7).

Atributos privados:

- `amLeader`: Variable *booleana* empleada para determinar si es o no un proceso líder. Este valor se obtiene desde el Detector de Fallos.
- `communicator`: Instancia del comunicador anónimo.
- `consensuableUnpackerFunction`: Objeto de tipo `Consensuable` para acceder a los métodos de serialización y deserialización de valores.
- `consensusAchieved`: Variable booleana utilizada para determinar si el algoritmo ha alcanzado una decisión.
- `crManager`: Administrador de Caída–recuperación.
- `decidedValue`: Variable que contiene el valor decidido.
- `decissionInformerThread`: Hilo que ejecuta la Tarea 2.
- `decissionMessageList`: Arreglo con todos los mensajes DEC.
- `decissionMessageListMutex`: Control de concurrencia para acceder a la lista `decissionMessageList`.
- `estimatedValue`: Variable que contiene el valor estimado de decisión.
- `fd`: Instancia del Detector de Fallos.
- `incomingMessages`: Arreglo delos mensajes que provienen del comunicador.
- `messageFactory`: Factoría de mensajes.
- `messageReceiverThread`: Hilo de recepción de mensajes.
- `numProcesses`: Número de procesos en el sistema.
- `phase0MessageList`: Arreglo con los mensajes de la fase PH0.
- `phase1MessageList`: Arreglo con los mensajes de la fase PH1.
- `phase2MessageList`: Arreglo con los mensajes de la fase PH2.
- `phase0Mutex`: Cerrojo para bloquear en la fase PH0.

- `phase1Mutex`: Cerrojo para bloquear en la fase PH1.
- `phase2Mutex`: Cerrojo para bloquear en la fase PH2.
- `phase0WaitCondition`: Bloqueo y desbloqueo de hilos en la fase PH0.
- `phase1WaitCondition`: Bloqueo y desbloqueo de hilos en la fase PH1.
- `phase2WaitCondition`: Bloqueo y desbloqueo de hilos en la fase PH2.
- `round`: Número de ronda.
- `runId`: Identificador de ejecución.
- `stopDecissionInformerThread`: Variable *booleana* empleada para detener el hilo `decissionInformerThread`.
- `stopMessageReciverThread`: Variable *booleana* empleada para detener el hilo `messageReceiverThread`.
- `task1Thread`: Hilo de la Tarea 1.

Métodos públicos:

- `Consensus()`: Constructor de la clase.
- `~Consensus()`: Destructor de la clase.
- `getCurrentRunId()`: Obtiene el identificador de ejecución actual.
- `propose()`: Implementa el algoritmo de la Figura 3.2.
- `statusChanged()`: Detecta un cambio en el Detector de Fallos.

Métodos privados:

- `decissionInformer()`: Método que contiene las instrucciones de la Tarea 2 de Consenso.
- `messageReceiver()`: Recibe los mensajes desde el comunicador.
- `recover()`: Método ejecutado cuando un proceso se recupera de una falla.
- `storeDecidedValue()`: Almacena el valor decidido con el Administrador de Caída-recuperación.

3.1.4.2. Clase Consensuable

Las aplicaciones que requieran integrar Consenso a su programación deben implementar esta interfaz (Figura 3.7).

Métodos públicos:

- `compareTo()`: Método que permite la comparación entre valores, con la finalidad de escoger en este caso el mínimo.
- `pack()`: Serializa los valores.
- `unpack()`: Deserializa los valores.
- `toString()`: Convierte a cadena de caracteres el valor.

Las clases que definen los mensajes para el algoritmo de Consenso son: `ConsensusPhase0Message`, `ConsensusPhase1Message`, `ConsensusPhase2Message` que corresponden a las fases PH0, PH1 y PH2, respectivamente. El tipo de mensaje `ConsensusDecissionMessage` se utiliza para realizar la difusión de los valores decididos (Figura 3.8). Como se puede observar, todos estos mensajes heredan de la clase `Message`.

3.1.4.3. Clase ConsensusPhase0Message

Define el mensaje de la fase PH0 (Figura 3.8).

Atributos privados:

- `estimatedValue`: Valor estimado de decisión.
- `round`: Número de ronda.
- `runId`: Identificador de ejecución.

Atributo público:

- `type_id`: Tipo de mensaje.

Métodos públicos:

- `ConsensusPhase0Message()`: Constructor de la clase.
- `~ConsensusPhase0Message()`: Destructor de la clase.
- `getEstimatedValue()`: Obtiene el valor estimado de decisión.
- `getRound()`: Obtiene el número de ronda.
- `getRunId()`: Obtiene el identificador de ejecución.
- `pack()`: Serializar un valor.

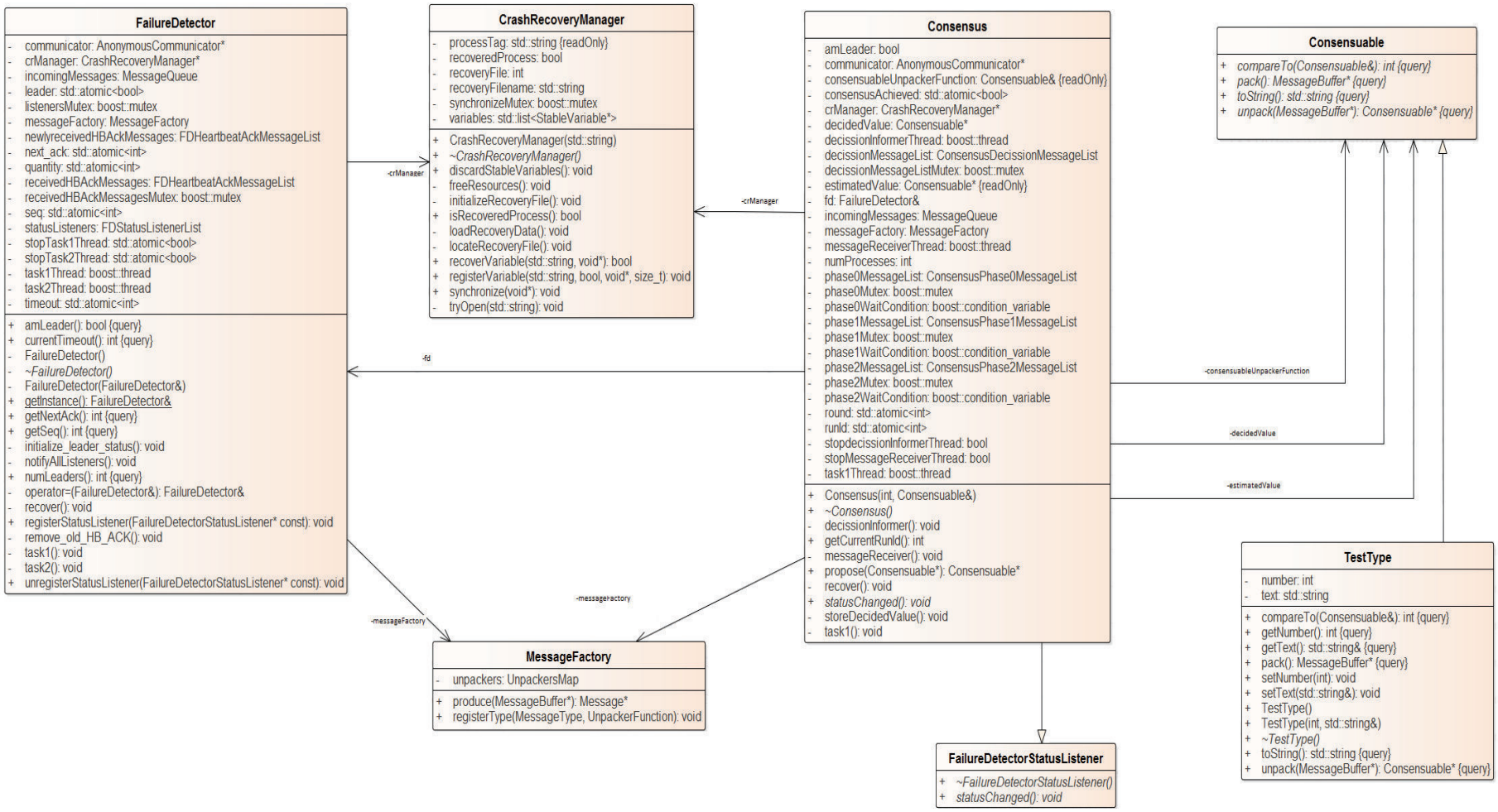

Figura 3.7. Diagrama de clases de Consenso

- `unpack()`: Deserializa un valor.

3.1.4.4. Clase ConsensusPhase1Message

Define el mensaje de la fase PH1 (Figura 3.8).

Atributos privados:

- `estimatedValue`: Valor estimado de decisión.
- `round`: Número de ronda.
- `runId`: Identificador de ejecución.

Atributo público:

- `type_id`: Tipo de mensaje.

Métodos públicos:

- `ConsensusPhase1Message()`: Constructor de la clase.
- `~ConsensusPhase1Message()`: Destructor de la clase.
- `getEstimatedValue()`: Obtiene el valor estimado de decisión.
- `getRound()`: Obtiene el número de ronda.
- `getRunId()`: Obtiene el identificador de ejecución.
- `pack()`: Serializar un valor.
- `unpack()`: Deserializa un valor.

3.1.4.5. Clase ConsensusPhase2Message

Define el mensaje de la fase PH2 (Figura 3.8).

Atributos privados:

- `estimatedValue`: Valor estimado de decisión.
- `round`: Número de ronda.
- `runId`: Identificador de ejecución.
- `agreed`: Da a conocer si se ha alcanzado Consenso en la fase PH2.

Atributo público:

- `type_id`: Tipo de mensaje.

Métodos públicos:

- `ConsensusPhase2Message()`: Constructor de la clase.
- `~ConsensusPhase2Message()`: Destructor de la clase.
- `getEstimatedValue()`: Obtiene el valor estimado de decisión.
- `getRound()`: Obtiene el número de ronda.
- `getRunId()`: Obtiene el identificador de ejecución.
- `isAgreed()`: Obtiene al valor de la variable `agreed`.
- `pack()`: Serializar un valor.
- `unpack()`: Deserializa un valor.

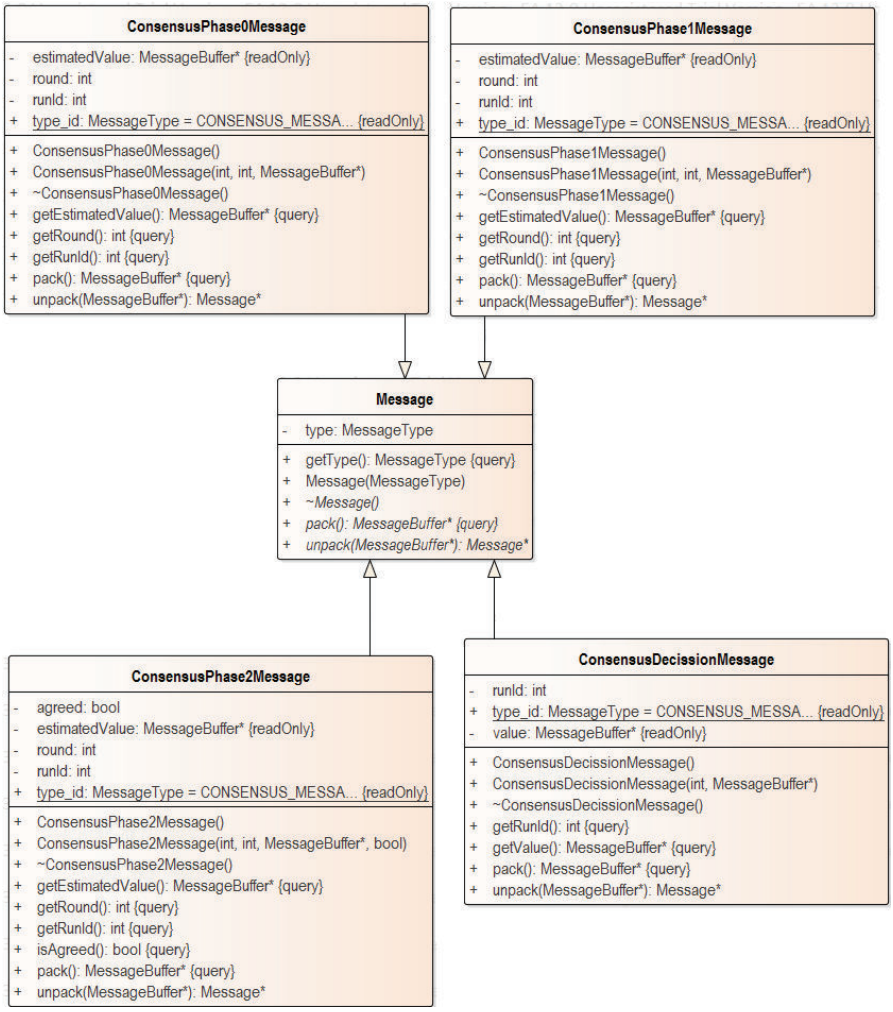

Figura 3.8. Diagrama de clases para tipos de mensajes de Consenso

3.1.4.6. Clase ConsensusDecissionMessage

Define el mensaje que informa de los valores decididos (Figura 3.8).

Atributos privados:

- `value`: Valor decidido.
- `runId`: Identificador de ejecución.

Atributo público:

- `type_id`: Tipo de mensaje.

Métodos públicos:

- `ConsensusDecissionMessage()`: Constructor de la clase.
- `~ConsensusDecissionMessage()`: Destructor de la clase.
- `getValue()`: Obtiene el valor decidido.
- `getRunId()`: Obtiene el identificador de ejecución.
- `pack()`: Serializar un valor.
- `unpack()`: Deserializa un valor.

3.1.5. DIAGRAMA DE SECUENCIA

El diagrama de secuencia de la Figura 3.9 muestra las operaciones realizadas al inicializar un objeto de la clase `Consensus`, mientras que en la Figura 3.10 se resumen las acciones que se llevan a cabo en segundo plano permanentemente. Por último, en la Figura 3.11 se destacan las operaciones realizadas al llamar al método `propose()`.

1. **Inicialización** (Figura 3.9)

    1.1. Se instancia un objeto de la clase `Consensus`.
    1.2. Se obtiene la instancia del Detector de Fallos mediante el método `getInstance()`.
    1.3. Se registra con el Detector de Fallos para conocer los posibles cambios que este puede tener.
    1.4. Instancia las listas que van a contener los mensajes de las diferentes fases.

1.5. Instancia el arreglo que va a contener los mensajes provenientes del comunicador.
1.6. Instancia el comunicador.
1.7. Crea y lanza el hilo que va a recibir los mensajes desde el comunicador.
1.8. Instancia el arreglo que va a contener los valores decididos.
1.9. Crea y lanza el hilo Informador que implementa la Tarea 2 de Consenso.

2. **Segundo plano permanente** (Figura 3.10)

2.1. Cuando el comunicador recibe un mensaje, este lo inserta en el arreglo de los mensajes entrantes.
2.2. El hilo de recepción extrae el mensaje desde el arreglo de mensajes entrantes.
2.3. El hilo de recepción guarda el mensaje extraído en el arreglo correspondiente. Por ejemplo, si es un mensaje PH0, lo guarda en el arreglo que contiene los mensajes de la fase PH0.
2.4. Si ocurre un cambio en el Detector de Fallos la función `statusChanged()`, se informa al hilo de aplicación del cambio.
2.5. El hilo de aplicación procesa los mensajes recibidos o el cambio del Detector de Fallos.
2.6. El hilo informador obtiene del arreglo que contiene la lista de decisiones los valores decididos.
2.7. El hilo informador difunde los valores decididos en mensajes DEC.

3. **Método** `propose()` (Figura 3.11)

3.1. Realiza una llamada al método `propose()`.
3.2. Difunde un mensaje PH0, PH1 o PH2 dependiendo en qué fase se encuentre. Luego espera para evaluar la condición que cada fase requiere.
3.3. Después de recibir un mensaje desde el comunicador, se lo guarda en el arreglo de mensajes entrantes.
3.4. El hilo de recepción obtiene el mensaje desde el arreglo de mensajes entrantes.
3.5. Se ubica el mensaje recibido en el arreglo correspondiente.

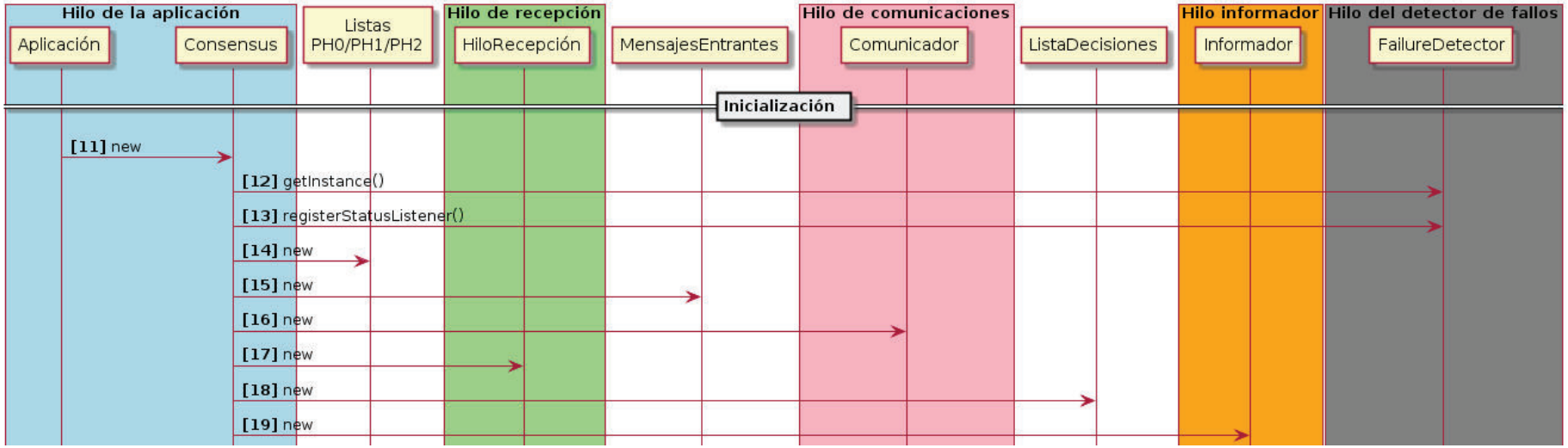

Figura 3.9. Diagrama de secuencia al inicializar

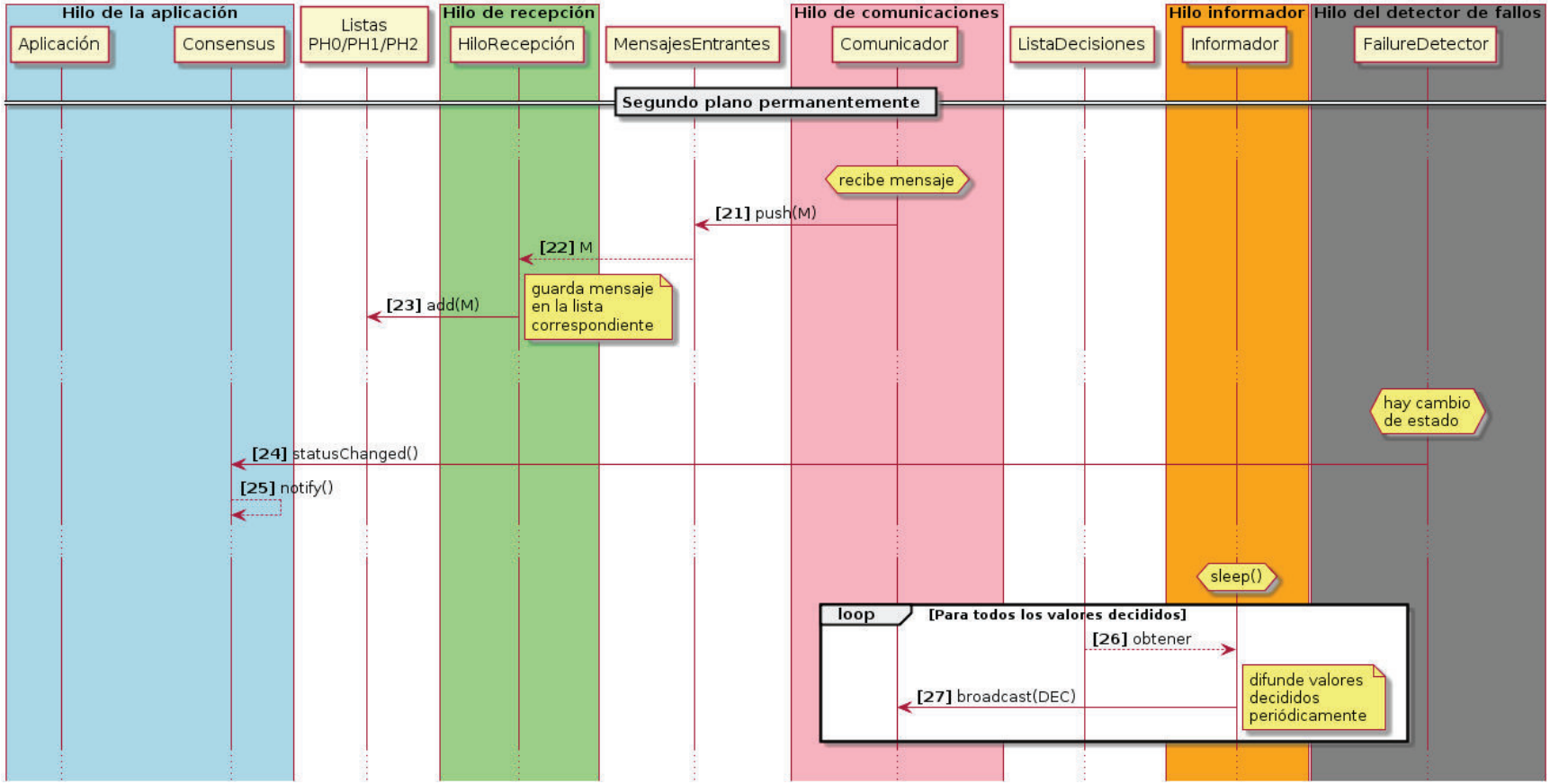

Figura 3.10. Diagrama de secuencia del segundo plano permanente

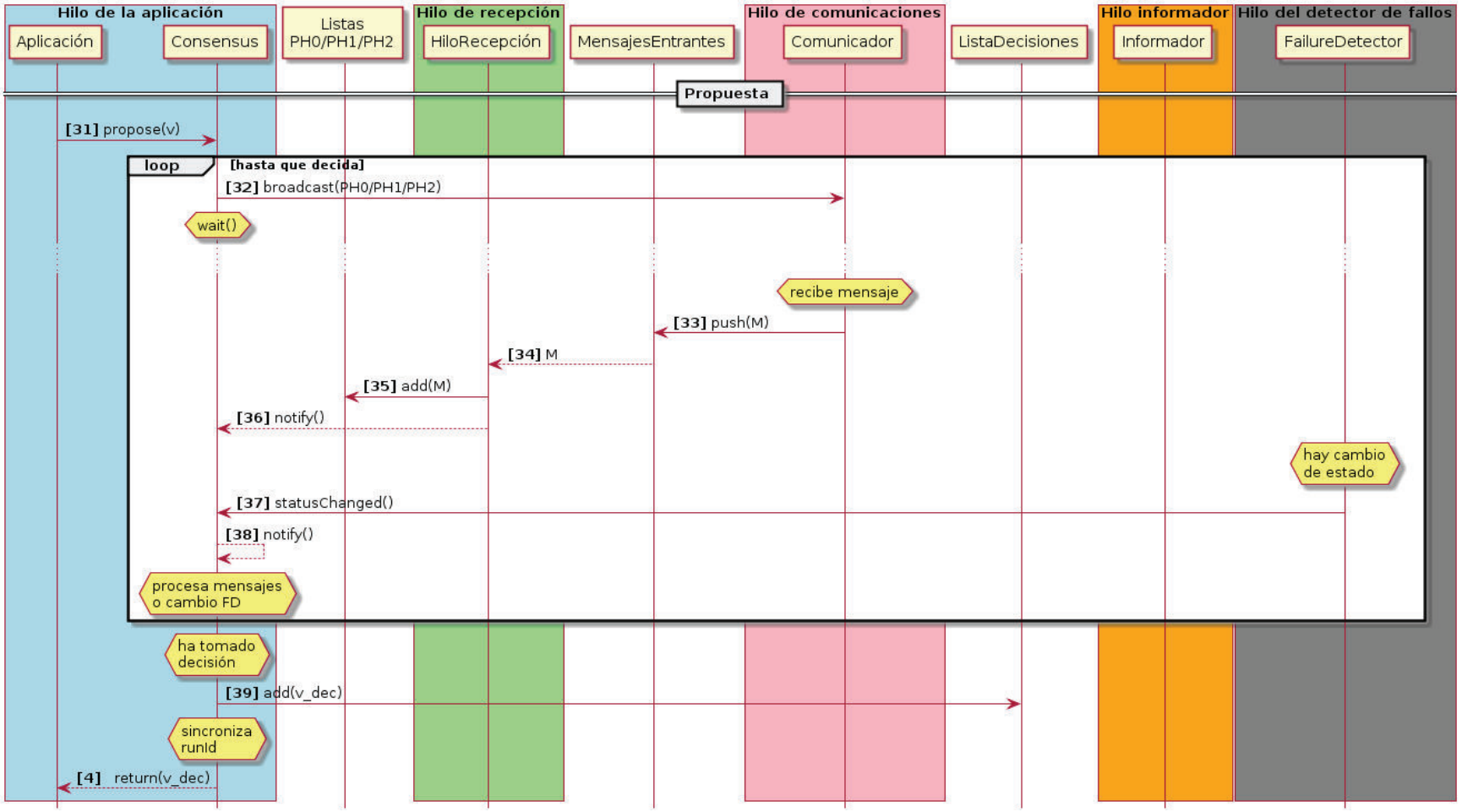

Figura 3.11. Diagrama de secuencia del método propose()

3.6. Notifica de un mensaje recibido.
3.7. Si ocurre un cambio en el Detector de Fallos, el método `statusChanged()` informa al hilo de aplicación del cambio.
3.8. El hilo de aplicación procesa los mensajes recibidos o el cambio del Detector de Fallos.
3.9. Al terminar Consenso y tomar una decisión, se guarda el valor en el arreglo que contiene la lista de decisiones.

4. Retorna el método `propose()` con el valor decidido (Figura 3.11).

3.2. IMPLEMENTACIÓN

En este apartado se detalla el código más importante del algoritmo de Consenso implementado en la clase `Consensus`. En el Código 3.1 se muestra el constructor de la clase `Consensus`. Este constructor recibe como parámetros de entrada `numProcesses`, correspondiente al número de procesos que conforman el sistema, y el objeto `consensuableUnpakerFunction`, mediante el cual es posible acceder a las funciones de serialización y deserialización de valores (línea 1).

En el constructor se inicializa el paso por referencia de: a) el Detector de Fallos `fd`, b) el número de procesos `numProcesses`, c) las variables *booleanas* `consensusAchieved`, `stopdecissionInformerThread`, d) `stopMessageReceiverThread`, y e) el objeto `consensuableUnpakerFunction` (líneas 2-6).

Los diferentes tipos de mensajes que Consenso maneja se registran con la factoría de mensajes (líneas 9-12), para luego instanciar el comunicador anónimo (línea 13). En la línea 14 se llama al método `recover()` para obtener los datos antes almacenados, en el caso de que un proceso que se recupere de un fallo. Si es un proceso que se recupera, no se corre el hilo informador con la finalidad de limitar el tráfico en la red (líneas 15-16). En la línea 17 se corre el hilo de recepción de mensajes, y en la línea 18 se registra con el Detector de Fallos para enterarse de los cambios que este puede tener. Las líneas 19 a 21 se introducen para evitar pérdida de mensajes, debido a que los demás procesos todavía no inician.

```
1  Consensus::Consensus(int numProcesses, const Consensuable&
   consensuableUnpackerFunction) :
2  fd(FailureDetector::getInstance()),
3  numProcesses(numProcesses),
4  consensusAchieved(false),
5  stopdecissionInformerThread(false),
6  stopMessageReceiverThread(false),
7  consensuableUnpackerFunction(consensuableUnpackerFunction)
8  {
9         messageFactory.registerType(ConsensusPhase0Message::type_id,
   (UnpackerFunction) new ConsensusPhase0Message());
10        messageFactory.registerType(ConsensusPhase1Message::type_id,
   (UnpackerFunction) new ConsensusPhase1Message());
11        messageFactory.registerType(ConsensusPhase2Message::type_id,
   (UnpackerFunction) new ConsensusPhase2Message());
12        messageFactory.registerType(ConsensusDecissionMessage::type_id,
   (UnpackerFunction) new ConsensusDecissionMessage());
          communicator = new
13        EthernetAnonymousCommunicator(CONSENSUS_PROTOCOL_ID, messageFactory,
          "", incomingMessages);
14        recover();
15        if (! crManager->isRecoveredProcess() )
16              decissionInformerThread =
                boost::thread(&Consensus::decissionInformer, this);
17        messageReceiverThread = boost::thread(&Consensus::messageReceiver,
          this);
18        fd.registerStatusListener(this);
19        if (globalParameters.initialSleep > 0) {
20              std::cerr << "Consensus sleeping for " <<
                globalParameters.initialSleep << " seconds..." << std::endl;
21              sleep(globalParameters.initialSleep);
22        }
23 }
```

Código 3.1. Constructor de Consensus()

El Código 3.2 presenta al destructor `~Consensus()`. La línea 2 de este código quita el registro del Detector de Fallos, mientras que en las líneas 3 y 4 se utilizan las variables `stopdecissionInformerThread` y `stopMessageReceiverThread` para terminar la ejecución del hilo informador y el hilo de recepción. Finalmente se

detiene el hilo de recepción de mensajes, y espera que los hilos terminen para finalmente borrar el comunicador (líneas 5-9).

```
1  Consensus::~Consensus(){
2          fd.unregisterStatusListener(this);
3          stopdecissionInformerThread = true;
4          stopMessageReceiverThread = true;
5          incomingMessages.stopAll();
6          if (! crManager->isRecoveredProcess() )
7                 decissionInformerThread.join();
8          messageReceiverThread.join();
9          delete communicator;
10 }
```

Código 3.2. Destructor ~Consensus()

```
1  void Consensus::recover(){
2          crManager = new CrashRecoveryManager("CONSENSUS");
3          if (! crManager->isRecoveredProcess() ||
4                 ! crManager->recoverVariable("RUNID", &runId)) {
5          runId = 0;
6          crManager->registerVariable("RUNID", false, &runId, sizeof(runId));
7          crManager->synchronize(&runId);
8          }
9  }
```

Código 3.3. Método recover()

El método `recover()` se detalla en el Código 3.3. Este método es similar el método `recover()` del Detector de Fallos del Código 2.31. Si es un proceso que se recupera, utiliza el Administrador de Caída-recuperación para recuperar el valor de la variable `runId` (líneas 3-4). Si es un proceso que se ejecuta por primera vez, inicializa `runId` y en ambos casos se registra la variable para que sea almacenada en el disco duro (líneas 5-7). El método que implementa la Tarea 1 de Consenso es `task1()`. El detalle del código desarrollado para este método se compara con las líneas del algoritmo de la Figura 3.2.

```
1  Consensuable* Consensus::propose(const Consensuable* proposedValue){
2  {
3          round = 0;
4          estimatedValue = proposedValue;
5          runId++;
6          task1Thread = boost::thread(&Consensus::task1, this);
7          task1Thread.join();
8          crManager->synchronize(&runId);
9          return decidedValue;
10 }
```

Código 3.4. Método propose()

La parte del método `task1()` que implementa las líneas 5 a 7 del algoritmo de la Figura 3.2 se muestra en el Código 3.5. En la línea 1 se aumenta en una unidad el número de ronda, seguidamente se conoce si se es o no un proceso líder (línea 2). En el caso de ser un proceso líder, difunde un mensaje PH0 con el identificador de ejecución, número de ronda y el valor estimado de decisión (líneas 3-5).

```
1  round++;
2  amLeader = fd.amLeader();
3  if (amLeader) {
4          communicator->broadcast(new ConsensusPhase0Message(runId, round,
           estimatedValue->pack()));
5  }
```

Código 3.5. Implementación de las líneas 5 a 7 del pseudocódigo del algoritmo de Consenso

La función $propose(v_i)$ del pseudocódigo de la Figura 3.2 se implementa en el método `propose()`. La línea 3 del Código 3.4 muestra la inicialización del número de ronda, seguidamente en la línea 4 el valor propuesto se asigna al valor estimado de decisión.

El identificador de ejecución se aumenta en una unidad (línea 5), para luego crear y correr el hilo de la Tarea 1 (línea 6). En la línea 7 el proceso espera a que el hilo de la Tarea 1 termine para luego guardar el valor del identificador de ejecución (línea 8), y finalmente retornar el método con el valor consensuado (línea 9).

```
1  await = true;
2  do {
3         if (consensusAchieved)
4                goto consensusAchieved;
5         if (amLeader != fd.amLeader()) {
6                await = false;
7         }
```

Código 3.6. Implementación de la línea 8a del pseudocódigo del algoritmo de Consenso

```
1  else if (await && amLeader) {
2         ConsensusPhase0MessageList::iterator i;
3         int numMessagesCurrentRound = 0;
4         int numLeaders = fd.numLeaders();
5         for (i = phase0MessageList.begin(); i != phase0MessageList.end() &&
          numMessagesCurrentRound != numLeaders; i++) {
6                ConsensusPhase0Message* msg = *i;
7                if (msg->getRunId() == runId && msg->getRound() == round)
8                       numMessagesCurrentRound++;
9         }
10        if (numMessagesCurrentRound == numLeaders) {
11               await = false;
12        }
13 }
```

Código 3.7. Implementación de la línea 8b del pseudocódigo del algoritmo de Consenso

La línea 8a del algoritmo de la Figura 3.2 se implementa en el Código 3.6. Las líneas 3 y 4 permiten salir del bucle de la Tarea 1 en caso de recibir un mensaje DEC. La línea 5 evalúa un posible cambio en el Detector de Fallos, si ocurre dicho cambio la variable `await` permite que las siguientes condiciones no se evalúen.

El Código 3.7 implementa la condición de espera de la línea 8b del pseudocódigo de la Figura 3.2. En la línea 4 se obtiene el número de líderes que el Detector de Fallos estima, para luego contar la cantidad de mensajes en el arreglo `phase0MessageList` (líneas 5-10). Si la cantidad de mensajes es igual a la cantidad de líderes entonces el hilo deja de esperar.

```
1  if (await) {
2          ConsensusPhase1MessageList::iterator i;
3          bool advanceMessageReceived = false;
4          boost::unique_lock<boost::mutex> m1(phase1Mutex);
5          for (i = phase1MessageList.begin(); i != phase1MessageList.end() && !
           advanceMessageReceived; i++) {
6          ConsensusPhase1Message* msg = *i;
7                 if (msg->getRunId() == runId && msg->getRound() == round) {
8                        advanceMessageReceived = true;
9                        await = false;
10                }
11         }
12 }
13 if (await)
14 phase0WaitCondition.wait(m0);
12 } while (await);
```

Código 3.8. Implementación de la línea 8c del pseudocódigo del algoritmo de Consenso

La condición de espera de la línea 8c del pseudocódigo de la Figura 3.2, se implementa en el Código 3.8. En las líneas 5 a 11 se verifica si existe un mensaje en el arreglo `phase1MessageList` que contenga un mensaje PH1 de la actual ejecución y de esta manera salir del bucle de espera que inicio en la línea 2 del Código 3.6 (línea 12). Si la variable `await` permanece con el valor `false`, el hilo se bloquea hasta recibir un mensaje PH0 o PH1 (línea 14).

El Código 3.9 y el Código 3.10 implementan las líneas 9 a 13 del pseudocódigo de la Figura 3.2. El arreglo `phase0MessageList` se recorre en el Código 3.9 para encontrar el menor valor de todos mediante la función `compareTo()`. Seguidamente, de manera similar en el Código 3.10, se recorre el arreglo `phase1MessageList` para escoger el menor valor de todos después de haber determinado el menor valor del arreglo `phase0MessageList`. Tanto para el Código 3.9, como para el Código 3.10, se usan variables auxiliares, como `minInitialized`, para iniciar la comparación con los demás valores del arreglo. Para acceder a los métodos `compareTo()` y `unpack()` se hace uso del objeto `consensuableUnpackerFunction`.

```
1  for (i0 = phase0MessageList.begin(); i0 != phase0MessageList.end(); i0++) {
2        ConsensusPhase0Message* msg = *i0;
3        if (msg->getRunId() == runId && msg->getRound() == round) {
4              if (! minInitialized) {
5                    estimatedValue = consensuableUnpackerFunction.unpack(msg-
                     >getEstimatedValue());
6                    minInitialized = true;
7              }
8              else {
                     Consensuable* msgEstVal =
9                    consensuableUnpackerFunction.unpack(msg-
                     >getEstimatedValue());
10                   if ( msgEstVal->compareTo ( *estimatedValue ) < 0 )
11                         estimatedValue = msgEstVal;
12             }
13       }}
```

Código 3.9. Implementación de las líneas 9 a 13 del pseudocódigo del algoritmo de Consenso (parte 1)

La línea 14 del pseudocódigo de la Figura 3.2 difunde un mensaje PH0 con el identificador de ejecución, el número de ronda y el valor estimado de decisión. Esto se implementa en el Código 3.11.

La condición de espera de la fase PH1 corresponde a la línea 15 del pseudocódigo de la Figura 3.2. Esta se implementa en el Código 3.12. La variable `await` permite mantenerse en el bucle mientras no se satisfaga la condición de la línea 11. Como se puede observar, en el lazo de las líneas 6 a 9 se cuenta la cantidad de mensajes PH1 recibidos en la lista `phase1MessageList` correspondientes a la actual ejecución y número de ronda.

Las líneas 2 y 3 permiten salir de los bucles de la Tarea 1 si se recibe un mensaje DEC. Si la variable `await` permanece con el valor `true`, en la línea 14 se bloque el hilo hasta recibir un mensaje PH1. El Código 3.13 detalla las instrucciones llevadas a cabo para implementar las líneas 16 a 18 del pseudocódigo de la Figura 3.2. Este código recorre el arreglo de mensajes de la fase PH1 y verifica si todos tienen el

mismo valor estimado de decisión (líneas 3-7). En caso de encontrar un valor distinto (línea 5), se asigna la variable `phase1Agreed` como `false` (línea 6).

```
1  for (i1 = phase1MessageList.begin(); i1 != phase1MessageList.end(); i1++) {
2        ConsensusPhase1Message* msg = *i1;
3        if (msg->getRunId() == runId && msg->getRound() == round) {
4              if (! minInitialized) {
5                    estimatedValue = consensuableUnpackerFunction.unpack(msg-
                     >getEstimatedValue());
6                    minInitialized = true;
7              }
8              else {
9                    Consensuable* msgEstVal =
                     consensuableUnpackerFunction.unpack(msg-
                     >getEstimatedValue());
10                   if ( msgEstVal->compareTo ( *estimatedValue ) < 0 )
11                         estimatedValue = msgEstVal;
12             }
13       }}
```

Código 3.10. Implementación de las líneas 9 a 13 del pseudocódigo del algoritmo de Consenso (parte 2)

```
1  communicator->broadcast(new ConsensusPhase1Message(runId, round,
   estimatedValue->pack()));
```

Código 3.11. Implementación de la línea 14 del pseudocódigo del algoritmo de Consenso

La variable `phase1Agrred` sirve posteriormente para construir el campo `agreed` del mensaje PH2.

El Código 3.14 difunde un mensaje PH2. De esta manera, se satisface la línea 19 del pseudocódigo de la Figura 3.2.

La condición de espera de la fase PH2 del algoritmo de Consenso se implementa en el Código 3.15. En este fragmento de código se cuentan los mensajes PH2 contenidos en el arreglo `phase2MessageList`, correspondientes a la actual ejecución y número de ronda (líneas 7-11).

```
1  do {
2          if (consensusAchieved)
3                 goto consensusAchieved;
4          ConsensusPhase1MessageList::iterator i;
5          int numMessagesCurrentRound = 0;
6          for (i = phase1MessageList.begin(); i != phase1MessageList.end() &&
           numMessagesCurrentRound <= numProcesses / 2; i++) {
7          ConsensusPhase1Message* msg = *i;
8                 if (msg->getRunId() == runId && msg->getRound() == round)
9                 numMessagesCurrentRound++;
10         }
11         if (numMessagesCurrentRound > numProcesses / 2)
12         await = false;
13         if (await)
14         phase1WaitCondition.wait(m1);
15 } while (await);
```

Código 3.12. Implementación de la línea 15 del pseudocódigo del algoritmo de Consenso

```
1  ConsensusPhase1MessageList::iterator i;
2  phase1Agreed = true;
3  for (i = phase1MessageList.begin(); i != phase1MessageList.end() &&
   phase1Agreed; i++) {
4          ConsensusPhase1Message* msg = *i;
5          if (msg->getRunId() == runId && msg->getRound() == round &&
           consensuableUnpackerFunction.unpack(msg->getEstimatedValue())-
           >compareTo(*estimatedValue)!=0)
6                 phase1Agreed = false;
7  }
```

Código 3.13. Implementación de las líneas 16 a 18 del pseudocódigo del algoritmo de Consenso

Si el número de mensajes PH2 es mayor a la mitad del número de procesos, entonces deja de esperar y continua su ejecución (líneas 12-13). Si la variable `await` permanece con el valor `true`, el hilo se bloquea hasta recibir un mensaje de la fase PH2. El Código 3.16 muestra la implementación de las líneas 21 a 23 del pseudocódigo de la Figura 3.2. Si existe un mensaje PH2 en la lista

phase2MessageList (línea 3) que contenga el campo agree con el valor true (línea 5), entonces se toma el valor de este mensaje como nuevo valor estimado de decisión (línea 6).

```
1  communicator->broadcast(new ConsensusPhase2Message(runId, round,
   estimatedValue->pack(), phase1Agreed));
```

Código 3.14. Implementación de la línea 19 del pseudocódigo del algoritmo de Consenso

```
1  await = true;
2  do {
3          if (consensusAchieved)
4                  goto consensusAchieved;
5          ConsensusPhase2MessageList::iterator i;
6          int numMessagesCurrentRound = 0;
7          for (i = phase2MessageList.begin(); i != phase2MessageList.end() &&
           numMessagesCurrentRound <= numProcesses / 2; i++) {
8          ConsensusPhase2Message* msg = *i;
9                  if (msg->getRunId() == runId && msg->getRound() == round)
10                         numMessagesCurrentRound++;
11         }
12         if (numMessagesCurrentRound > numProcesses / 2)
13                 await = false;
14         if (await)
15                 phase2WaitCondition.wait(m2);
16 } while (await);
```

Código 3.15. Implementación de la línea 20 del pseudocódigo del algoritmo de Consenso

Las líneas 24 a 26 del pseudocódigo de la Figura 3.2 se implementan en el Código 3.17. Si alguno de los mensajes PH2 tienen el campo agree con el valor false (línea 4), entonces es que todavía no se ha consensuado un valor y será necesaria una nueva ronda (línea 10). Si todos los mensajes PH2 tienen el campo agree con el valor true, significa que se ha consensuado un valor y el valor estimado de decisión será el valor decidido decidedValue (línea 8). Solo los mensajes que estén permanentemente ejecutándose pueden enviar mensajes DEC. Para ello, se guarda el valor decidido en el arreglo de decisiones decissionMessageList mediante el

método `storeDecidedValue()` (líneas 11-12). De este modo el método `decissionInformer()` puede acceder al arreglo y enviar los mensajes DEC. El método que implementa las acciones de la Tarea 2 de Consenso es `decissionInformer()`.

```
1  ConsensusPhase2MessageList::iterator i;
2  phase2AgreedReceived = false;
3  for (i = phase2MessageList.begin(); i != phase2MessageList.end() && !
   phase2AgreedReceived; i++) {
4         ConsensusPhase2Message* msg = *i;
5         if (msg->getRunId() == runId && msg->getRound() == round && msg-
          >isAgreed()) {
6               estimatedValue = consensuableUnpackerFunction.unpack(msg-
                >getEstimatedValue());
7               phase2AgreedReceived = true;
8         }
9  }
```

Código 3.16. Implementación de las líneas 21 a 23 del pseudocódigo del algoritmo de Consenso

```
1  consensusAchieved = true;
2  for (i = phase2MessageList.begin(); i != phase2MessageList.end() &&
   consensusAchieved; i++) {
3         ConsensusPhase2Message* msg = *i;
4         if (msg->getRunId() == runId && msg->getRound() == round && (! msg-
          >isAgreed() || consensuableUnpackerFunction.unpack(msg-
          >getEstimatedValue())->compareTo(*estimatedValue)!=0))
5               consensusAchieved = false;
6  }
7  if (consensusAchieved) {
8  decidedValue = (Consensuable*) estimatedValue;
9  }
10 } while (! consensusAchieved);
11 if (! crManager->isRecoveredProcess())
12        storeDecidedValue();
```

Código 3.17. Implementación de las líneas 24 a 26 del pseudocódigo del algoritmo de Consenso

En las líneas 2 a 10 del Código 3.18 se muestra que cada segundo se envía una decisión (línea 9). Se accede al arreglo de decisiones `decissionMessageList` en exclusión mutua, a fin de que otro hilo no use el arreglo mientras este método hace uso de él.

```
1  void Consensus::decissionInformer(){
2          do
3          {
4                  {
5                  boost::unique_lock<boost::mutex>  m(decissionMessageListMutex);
6                  for ( ConsensusDecissionMessage* decMessage :
                           decissionMessageList )
7                          communicator->broadcast(decMessage);
8                  }
9                  sleep(1);
10         } while (stopdecissionInformerThread );
11 }
```

Código 3.18. Método decissionInformer()

```
1  case ConsensusPhase0Message::type_id: {
2          ConsensusPhase0Message* const phase0Message = (ConsensusPhase0Message*
           const) message;
3          boost::unique_lock<boost::mutex>  m0(phase0Mutex);
4          phase0MessageList.push_back(phase0Message);
5          if (phase0Message->getRunId() == runId)
6                  phase0WaitCondition.notify_one();
7  }
8  break;
```

Código 3.19. Clasificación de los mensajes PH0

El método `messageReceiver()` recibe los mensajes que provienen del comunicador y los clasifica en la lista correspondiente. El Código 3.19 clasifica a los mensajes de la fase PH0, mientras que el Código 3.20, el Código 3.21 y el Código 3.22 clasifican los mensajes de la fase PH1, PH2 y DEC, respectivamente. En cada uno de estos Códigos se accede a la lista correspondiente en exclusión mutua,

utilizando `phase0Mutex`, `phase1Mutex` y `phase2Mutex` para el control de concurrencia con los demás hilos. En cada caso, se notifica a los bloqueos `phase0WaitCondition`, `phase1WaitCondition` y `phase2WaitCondition` la llegada de un mensaje utilizando `notify_one()`.

```
1  case ConsensusPhase1Message::type_id: {
2          ConsensusPhase1Message* const phase1Message = (ConsensusPhase1Message*
           const) message;
3          boost::unique_lock<boost::mutex>  m0(phase0Mutex);
4          boost::unique_lock<boost::mutex>  m1(phase1Mutex);
5          phase1MessageList.push_back(phase1Message);
6          if (phase1Message->getRunId() == runId) {
7                 phase0WaitCondition.notify_one();
8                 phase1WaitCondition.notify_one();
9          }
10 }
11 break;
```

Código 3.20. Clasificación de los mensajes PH1

```
1  case ConsensusPhase2Message::type_id:{
2          ConsensusPhase2Message* const phase2Message = (ConsensusPhase2Message*
           const) message;
3          boost::unique_lock<boost::mutex>  m2(phase2Mutex);
4          phase2MessageList.push_back(phase2Message);
5          if (phase2Message->getRunId() == runId)
6                 phase2WaitCondition.notify_one();
7  }
8  break;
```

Código 3.21. Clasificación de los mensajes PH2

El Código 3.20 en las líneas 7 y 8 notifica a la fase PH0 y a la fase PH1. Esto se debe a que de acuerdo con la línea 8c del pseudocódigo de la Figura 3.2 se requiere notificar en caso de recibir un mensaje PH1 si todavía se encuentra en la fase PH0. El Código 3.22 describe las instrucciones a ejecutar al recibir un mensaje DEC. Primero, bloquea todas las fases (líneas 4-6), y, después, se indica que se ha

consensuado un valor mediante la variable `consensusAchieved` (línea 7). Seguidamente toma como valor decido el valor del mensaje DEC (línea 8), y, finalmente, se informa al hilo que ejecuta el método `task1()` que se ha consensuado un valor y no es necesario seguir esperando en cualquier fase que se encuentre (líneas 9-11).

```
1  case ConsensusDecissionMessage::type_id:{
2         ConsensusDecissionMessage* const decissionMessage =
          (ConsensusDecissionMessage* const) message;
3         if (decissionMessage->getRunId() == runId && ! consensusAchieved){
4               boost::unique_lock<boost::mutex>  m0(phase0Mutex);
5               boost::unique_lock<boost::mutex>  m1(phase1Mutex);
6               boost::unique_lock<boost::mutex>  m2(phase2Mutex);
7               consensusAchieved = true;
8               decidedValue =
                consensuableUnpackerFunction.unpack(decissionMessage-
                            >getValue());
9               phase0WaitCondition.notify_one();
10              phase1WaitCondition.notify_one();
11              phase2WaitCondition.notify_one();
12        }
13 }
14 break;
```

Código 3.22. Clasificación de los mensajes DEC

CAPÍTULO 4
APLICACIÓN DE MENSAJERÍA

4.1. DISEÑO

4.1.1. INTRODUCCIÓN

El presente capítulo muestra una aplicación de prueba para demostrar el funcionamiento y la posible aplicación de los algoritmos de Detector de Fallos y Consenso. Para ello, se describe el problema que presentan las bases de datos al momento de replicarlas. Normalmente, es deseable tener varias copias de una base de datos y distribuirlas en diferentes lugares ya sea física o virtualmente. Con esto, se ganan algunos beneficios como: incrementar la accesibilidad a los datos y aumentar la disponibilidad.

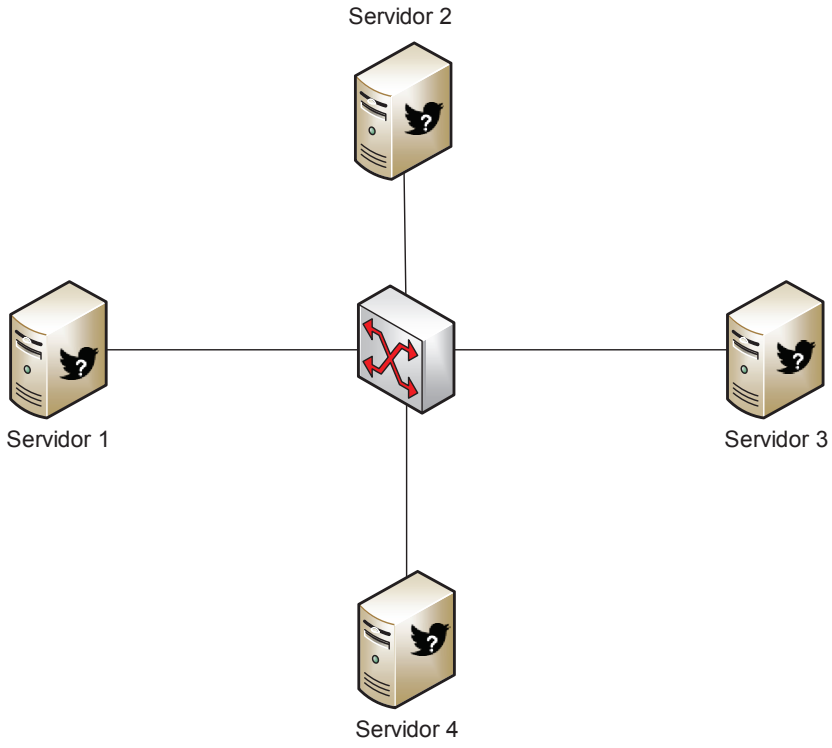

Figura 4.1. Escenario de prueba de la aplicación

La contraparte de estos beneficios es mantener la sincronización entre todas las copias. Es decir, que todas las copias sean idénticas [32]. Por diferentes motivos, como retardo, perdida de paquetes, entre otros, las copias pueden recibir las mismas

instrucciones de actualización de datos, pero en desorden o incompletas. Por esta razón, los servidores deben consensuar el orden de las operaciones para que una vez ejecutadas, las bases de datos se mantengan con la misma información. Un ejemplo práctico son las aplicaciones de mensajería o de noticias en donde se consensua el orden en el que los mensajes o las publicaciones deben aparecer.

4.1.2. CONFIGURACIÓN

En la Figura 4.1 se muestra el escenario de la aplicación de prueba. Como se puede observar, el escenario consiste de cuatro servidores conectados en red. Cada uno, cuenta con un mismo conjunto de mensajes que se asume que fueron recibidos y almacenados temporalmente.

Posteriormente el orden de estos mensajes se cambia aleatoriamente para emular un orden de recepción distinto en cada servidor. Luego de ejecutar los algoritmos del Detector de Fallos y Consenso en cada uno, se obtiene una lista consensuada de mensajes en el mismo orden en todos los servidores.

4.2. IMPLEMENTACIÓN

El Código 4.1 muestra el conjunto de mensajes que tienen inicialmente todos los servidores, para posteriormente ser desordenados aleatoriamente.

```
1  string tweets [10] = {
2          "Hello everyone", "I like this app", "I share my music",
3          "This track is the best", "I want to travel to Ecuador",
4          "Hello World", "Good morning", "Good Afternoon",
5          "I can't speak spanish", "This is so funny"};
```

Código 4.1. Lista de mensajes

En el Código 4.2 se muestran los arreglos que se utilizan en la aplicación. El arreglo `tweets_consensus` va a contener los mensajes en un orden consensuado. Los valores enteros que cada servidor propone se encuentran en la lista `proposeValues`. El arreglo permite desordenar el conjunto de mensaje mediante la generación posterior de números aleatorios.

```
1  std::list<string> tweets_consensus;
2  std::list<int> proposeValues;
3  std::vector<int> random_numbers(10);
```

Código 4.2. Arreglos

```
1  int myrandom (int i){
2  return random() % i; }
```

Código 4.3. Método de generación de números aleatorios

El método que permite la generación de un orden aleatorio es el que se muestra en el Código 4.3. Al iniciar la aplicación se ejecuta el Código 4.4. Primero se crea un objeto `Consensus`, para posteriormente en las líneas 2 a 4 llenar el arreglo `random_numbers` con números del uno al diez en orden. Luego, en la línea 5, se altera ese orden aleatoriamente utilizando la función `random_shuffle`, que es parte de la librería estándar de C++ y el método `myrandom`. Seguidamente se muestra en pantalla el orden que los mensajes tienen a partir de los números generados en la línea 9. El arreglo `proposeValues` se llena con los números de la lista `random_numbers` (línea 10).

```
1  Consensus consensus(globalParameters.numProcesses, unpacker);
2  for(int i = 0 ; i<= 9 ; i++){
3        random_numbers[i] = i;
4  }
5  std::random_shuffle(random_numbers.begin(), random_numbers.end(), myrandom);
6  cout << "Message order before Consensus" << endl;
7  cout << "" << endl;
8  for(int i = 0 ; i<= 9 ; i++){
9        cout << "\t" << tweets[random_numbers[i]] << endl;
10       proposeValues.push_back(random_numbers[i]);
11 }
```

Código 4.4. Inicio de la aplicación

La aplicación consta de un bucle principal el cual se ejecuta hasta que no haya más valores que proponer. El Código 4.5, en las líneas 2 a 4, crea el texto descriptivo de

la propuesta de la clase `TestType` (Figura 4.7). El método `front()` permite acceder al primer elemento del arreglo. Luego, en la línea 5 se crea un objeto de la clase `TestType` con el valor a consensuar junto con un texto descriptivo (Figura 4.7). En línea 6 se obtiene el retorno del método `propose()` que corresponde el valor consensuado. Por pantalla se muestra el valor propuesto y el valor consensuado junto con el identificador de ejecución (líneas 7-8). Las líneas 9 y 10 convierten el valor consensuado a entero. Se inserta el mensaje en el arreglo `tweets_consensus` correspondiente al valor consensuado (línea 11). El arreglo `proposeValues` contiene todos los valores a proponer, para evitar que un valor consensuado se proponga de nuevo. En la línea 12 se elimina este valor del arreglo. Finalmente, se borra `agreed_value` para liberar memoria y seguidamente se duerme el hilo durante un tiempo aleatorio para que los servidores propongan valores en diferentes tiempos.

```
1  do{
2          cout << "" << endl;
3          std::stringstream s;
4          s << "My proposal is message" << proposeValues.front() << ".";
5          TestType proposed_value(proposeValues.front() , s.str());
6          TestType* agreed_value = (TestType*)
           consensus.propose(&proposed_value);
7          cout << "Proposed value    : " << proposed_value.toString() << endl;
8          cout << "Agreed value (R " << consensus.getCurrentRunId() << "): " <<
           agreed_value->toString() << endl;
9          string value_s = agreed_value->toString();
10         int value = atoi(value_s.c_str());
11         tweets_consensus.push_back(tweets[value]);
12         proposeValues.remove(value);
13         delete agreed_value;
14         cout << "" << endl;
15         sleep((random() % 7) + 3);
16 }while(!proposeValues.empty());
```

Código 4.5. Bucle principal de la aplicación

CAPÍTULO 5

PRUEBAS

5.1. ALGORITMO DE DETECCIÓN DE FALLOS

El Código 5.1 muestra el programa de prueba para el Detector de Fallos. En las líneas 1 y 2 se obtiene la instancia del Detector de Fallos, para luego registrarse con el mismo. En el bucle, primero se imprime el identificador de proceso PID (*Process ID*) (línea 6), seguidamente se despliega si es o no un proceso líder (líneas 7-11).

```
1  FailureDetector& fd = FailureDetector::getInstance();
2  fd.registerStatusListener(new DemoFDListener);
3  int i = 0;
4  while (1) {
5         if ((i++ % 10) == 0) {
6                std::cout << "PID " << getpid() << " probed: ";
7                if (fd.amLeader())
8                       std::cout << "leader=TRUE, quantity=" << fd.numLeaders();
9                else
10                      std::cout << "leader=FALSE, quantity=NP";
11               std::cout << std::endl;
12        }
13        sleep(1);
14 }
```

Código 5.1. Programa de prueba para el algoritmo del Detector de Fallos

5.1.1. PRUEBA 1

El formato del mensaje de la Figura 2.10 se prueba utilizando el *sniffer* Wireshark para capturar las tramas. Se implementan tres máquinas, y el *sniffer* instalado en Windows. Se conecta todo mediante un *switch* virtual estableciéndose la topología de la Figura 5.1.

Las tramas capturadas con Wireshark se muestran en la Figura 5.2. Como se puede observar, la dirección MAC origen es `04:04:04:04:04:04` y la dirección MAC

destino es la dirección de *broadcast* `FF:FF:FF:FF:FF:FF`. También es posible comprobar que el tipo de trama es `0x88b5`, definida como trama experimental [30], tal y como se definió en el código.

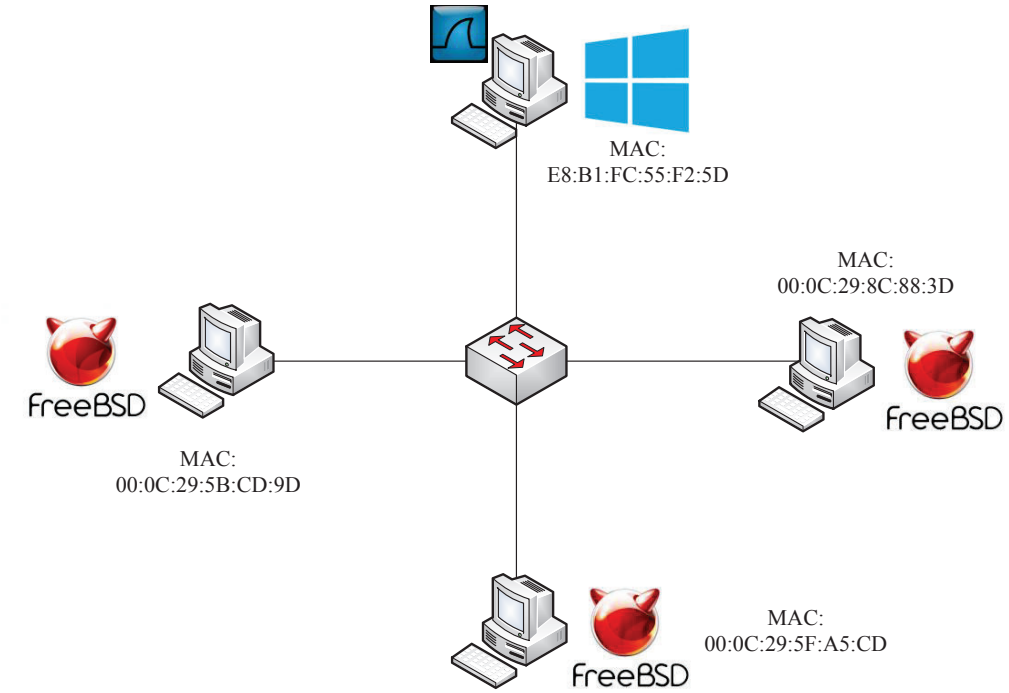

Figura 5.1. Topología implementada para pruebas

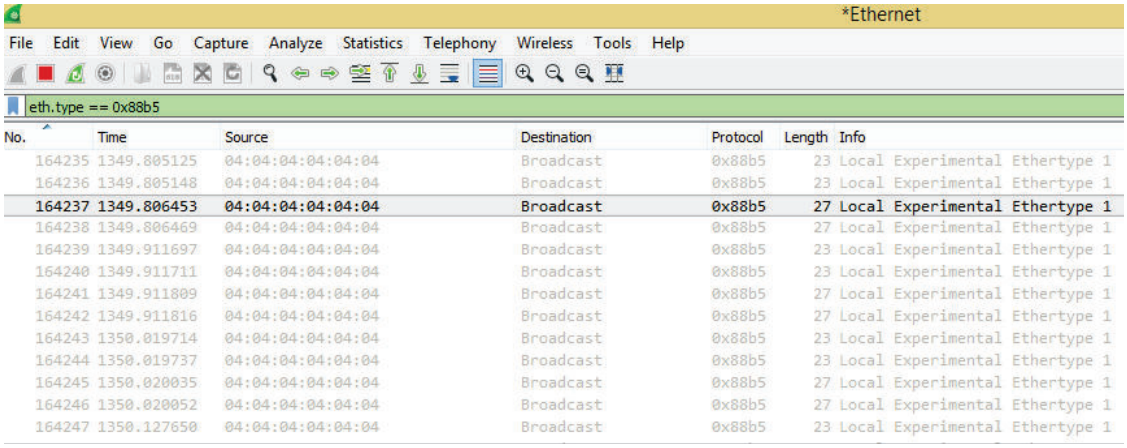

Figura 5.2. Tramas capturadas

En el campo de datos de la trama comienza con la etiqueta de protocolo, para el caso del Detector de Fallos, `FAILURE_DETECTOR_PROTOCOL_ID` y tipo mensaje `FD_MESSAGE_ID_BASE + type_id`. Como se indica en la Figura 5.3.

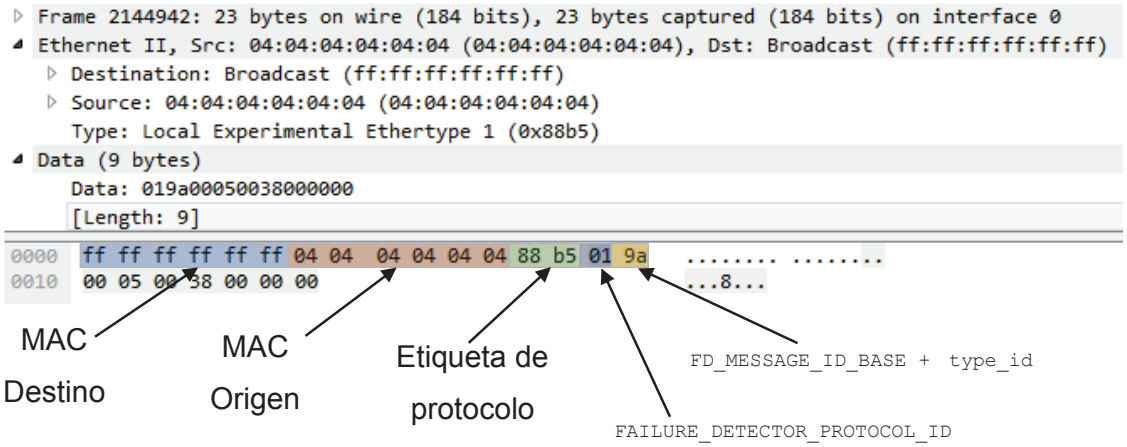

Figura 5.3. Formato de tramas capturadas

5.1.2. PRUEBA 2

Consiste en comprobar la creación del fichero `.rec` en el directorio `/tmp/`. En la Figura 5.4 se muestra como el programa reporta la creación del fichero `/tmp/obGAnAzX7.rec`.

```
[CRM] CrashRecoveryManager(FAILURE_DETECTOR)
[CRM] initializeRecoveryFile(/tmp/obGAnAzX7.rec)
[CRM] registerVariable(TIMEOUT,size=4)
[CRM] synchronize(TIMEOUT,1)
```

Figura 5.4. Creación del fichero .rec

Se lista el contenido del directorio `/tmp`, y efectivamente se visualiza el fichero creado en la ruta programada (Figura 5.5).

```
[desarrollador@albatros ~]$ ls -l /tmp
total 36
-rw-------  1 desarrollador  wheel     1 Feb  2 16:06 0oSAuC
srwx------  1 desarrollador  wheel     0 Feb 11 16:38 CODEBLOCKSdesarrollador.socket
-rw-------  1 desarrollador  wheel     4 Feb 11 16:38 Code::Blocks-desarrollador
drwx------  2 desarrollador  wheel   512 Feb 11 16:37 akonadi-desarrollador.d0ngnN
srwxrwxrwx  1 desarrollador  wheel     0 Feb 11 16:37 dbus-CK2wRBkQu6
srwxrwxrwx  1 desarrollador  wheel     0 Feb 11 15:21 dbus-ZqESFXr1ol
srwxrwxrwx  1 desarrollador  wheel     0 Oct 10 11:58 dbus-kcdUqQp7e8
srwxrwxrwx  1 desarrollador  wheel     0 Nov 30 16:21 dbus-pDfRYIXmvd
drwx------  2 desarrollador  wheel   512 Feb 11 16:37 fam-desarrollador
drwx------  2 root           wheel   512 Feb 11 14:24 fam-root
drwx------  3 desarrollador  wheel  1024 Feb 11 16:37 kde-desarrollador
drwx------  2 desarrollador  wheel  3072 Feb 11 16:37 ksocket-desarrollador
-rw-------  1 desarrollador  wheel     1 Feb 11 16:38 lN5zX2
-rw-------  1 root           wheel    84 Feb 11 20:39 obGAnAzX7.rec
```

Figura 5.5. Fichero .rec creado en el directorio /tmp

5.1.3. PRUEBA 3

Consiste en comprobar que un proceso que se recupera tome los valores del fichero `.rec`. Se ejecuta el algoritmo y se lo detiene a propósito. Posteriormente, se vuelve a iniciar y se muestra en pantalla que el fichero ha sido encontrado y las variables recuperadas (Figura 5.6). El proceso de recuperación del fichero consiste primero en localizarlo, si lo encuentra lo recupera, si no, crea uno nuevo. Este proceso se lleva a cabo en el método `locateRecoveryFile()` de `CrashRecoveryManager`.

```
[CRM] CrashRecoveryManager(FAILURE_DETECTOR)
[CRM] tryOpen() found /tmp/obGAnAzX7.rec
[CRM] loadRecoveryData() loaded TIMEOUT = 1
[CRM] recoverVariable(TIMEOUT) = 1
```

Figura 5.6. Fichero .rec recuperado

5.1.4. PRUEBA 4

Se pone a prueba la estructura del fichero `.rec`. En la Figura 5.7 se observa el contenido del fichero .rec. Como se puede observar, la primera cadena de caracteres es `CRASH-RECOVERY01` que corresponde al `MAGIC_STRING` de `CrashRecoveryManager`. La cadena `FAILURE_DETECTOR` es el descriptor del proceso que utiliza el fichero. `TIMEOUT` es el nombre de la variable y a continuación se encuentra su valor en hexadecimal.

```
00000000   43 52 41 53  48 2D 52 45  43 4F 56 45  52 59 30 31  46 41 49 4C  CRASH-RECOVERY01FAIL
00000014   55 52 45 5F  44 45 54 45  43 54 4F 52  54 49 4D 45  4F 55 54 00  URE_DETECTORTIMEOUT.
00000028   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
0000003C   00 00 00 00  04 00 00 00  00 00 00 00  01 FF FF FF  FF FF FF FF  ....................
00000050   01 00 00 00                                                      ....
```

Figura 5.7. Contenido del fichero .rec

5.1.5. PRUEBA 5

Consiste en comprobar el cambio de estado a líder de un proceso cuando no existen procesos líderes en el sistema. Se ejecutan tres procesos, dos de ellos no son líderes (Figura 5.8). Si se detiene al proceso líder, los demás detectan esa ausencia y se declaran a sí mismos como líderes (Figura 5.9). Nótese que luego del cambio de líderes la variable *quantity* se desestabiliza y muestra tres líderes en el sistema cuando solo hay dos. Tiempo después esta variable da el resultado correcto (Figura 5.9).

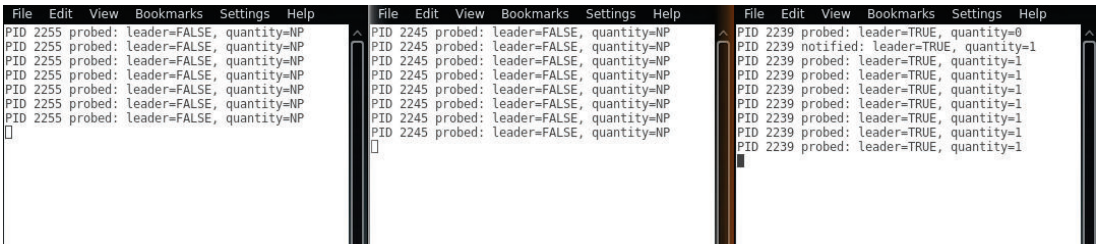

Figura 5.8. Un líder en el sistema

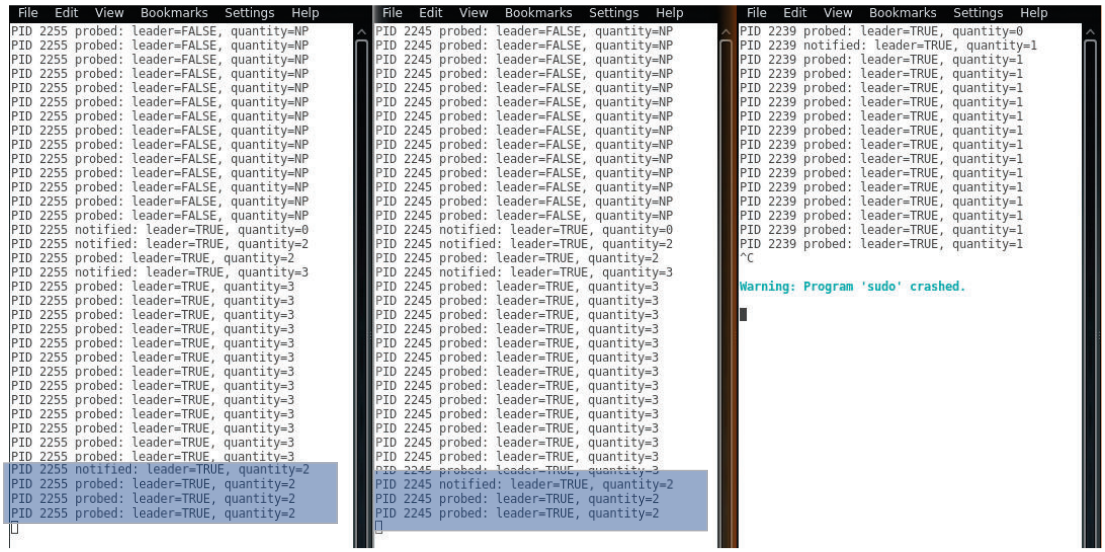

Figura 5.9. Eventos producidos cuando un líder falla

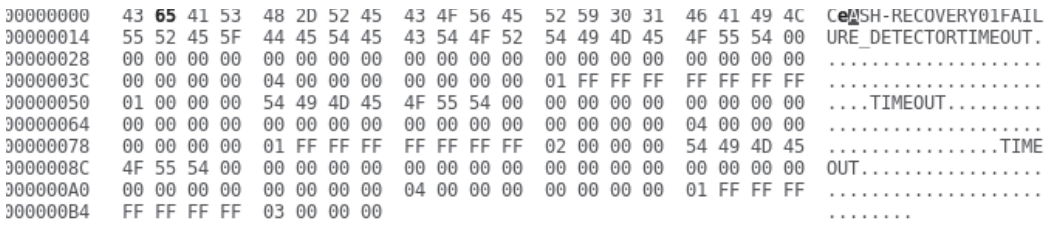

Figura 5.10. Modificación del fichero .rec

5.1.6. PRUEBA 6

Probar que sucede si el fichero `.rec` es modificado o alterado. Para ello se busca el fichero en el directorio `/tmp` y se lo modifica manualmente. En la Figura 5.10 se muestra que se le modifica un carácter del fichero. Se inserta una salida por consola en el método `locateRecoveryFile()` y `tryOpen()` para verificar que el proceso ignora el fichero por no ser válido. Al reiniciarlo, el proceso simplemente reconoce el fichero como no válido y crea uno nuevo, tal y como se muestra en la Figura 5.11.

```
[CRM] CrashRecoveryManager(FAILURE_DETECTOR)
File Found
Try to open
Invalid File
[CRM] initializeRecoveryFile(/tmp/qBcprJT4j.rec)
[CRM] registerVariable(TIMEOUT,size=4)
[CRM] synchronize(TIMEOUT,1)
[CRM] synchronize(TIMEOUT,2)
PID 2322 notified: leader=1, quantity=0, timeout=2
PID 2322 notified: leader=1, quantity=1, timeout=2
^C
```

Figura 5.11. Modificación del fichero .rec

5.2. ALGORITMO DE CONSENSO

El Código 5.2 muestra el programa utilizado para probar el algoritmo de Consenso. Este programa genera un número aleatorio (línea 3) y lo propone utilizando un objeto de la clase `TestType` (línea 6) definida en la Figura 3.7.

5.2.1. PRUEBA 1

Se corre el programa descrito en el Código 5.2 para verificar que efectivamente los procesos proponen valores y llegan a un acuerdo. Como se puede observar en la Figura 5.12, Figura 5.13 y Figura 5.14, los procesos consensuan sus valores como se espera que lo hagan.

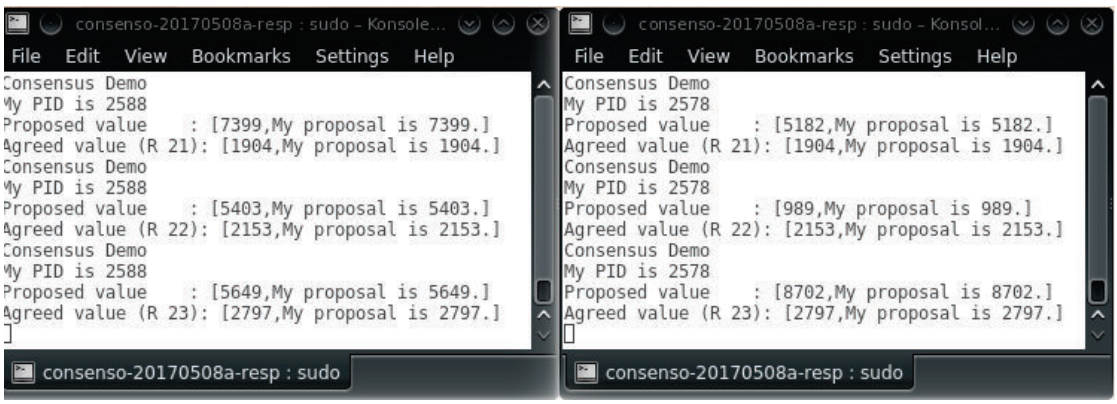

Figura 5.12. Prueba del algoritmo de Consenso (parte 1)

5.2.2. PRUEBA 2

Se detienen dos procesos de la Figura 5.14 con la finalidad de comprobar si el algoritmo de Consenso sigue trabajando con la mayoría de procesos. El resultado de la Figura 5.15, Figura 5.16 y Figura 5.17 muestran que el algoritmo sigue

consensuando valores con la mayoría de procesos que continúan ejecutándose. Esto se puede comprobar en el número de identificación de ejecución que continua en aumento (R 47), mientras que en los procesos detenidos el valor es (R 36) y (R 37).

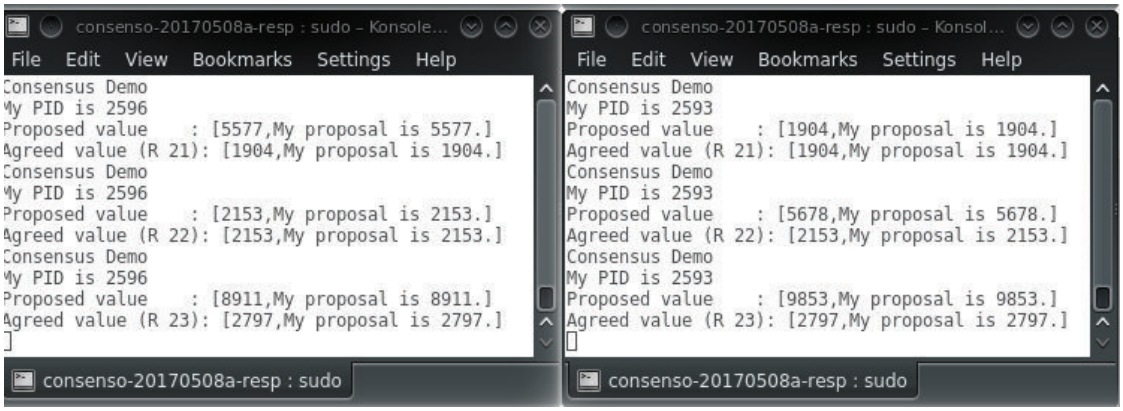

Figura 5.13. Prueba del algoritmo de Consenso (parte 2)

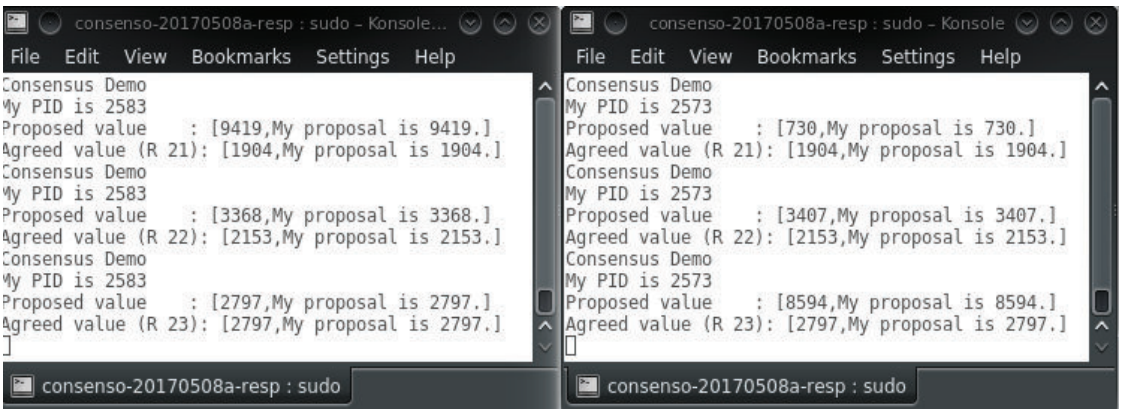

Figura 5.14. Prueba del algoritmo de Consenso (parte 3)

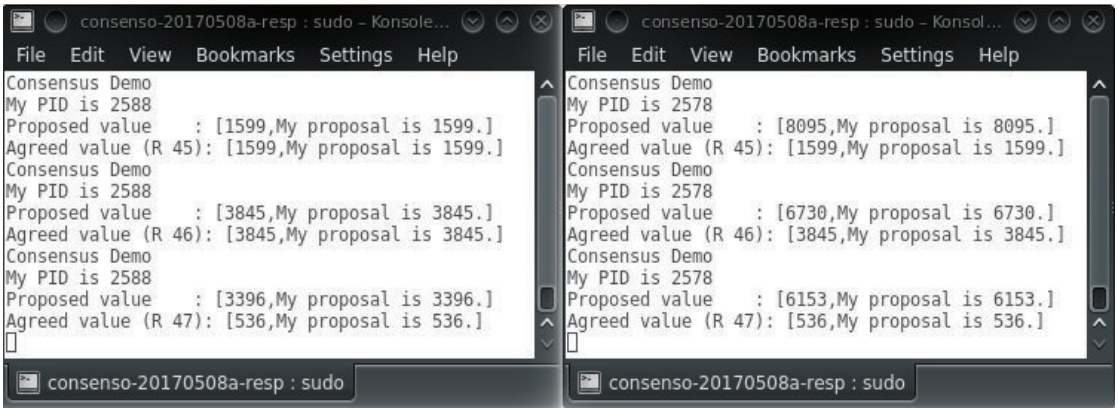

Figura 5.15. Consenso de valores con la mayoría de procesos (parte 1)

```
1   do {
2           std::stringstream s;
3           int number = random() % 10000;
4           s << "My proposal is " << number << ".";
5           TestType proposed_value(number, s.str());
6           TestType* agreed_value = (TestType*)
            consensus.propose(&proposed_value);
7           cout << "Consensus Demo" << endl << "My PID is " << my_pid << endl;
8           cout << "Proposed value    : " << proposed_value.toString() <<
    endl;
9           cout << "Agreed value (R " << consensus.getCurrentRunId() << "): "
    <<      agreed_value->toString() << endl;
10          delete agreed_value;
11          sleep((random() % 7) + 3);
12  } while (true);
```

Código 5.2. Programa de prueba para el algoritmo de Consenso

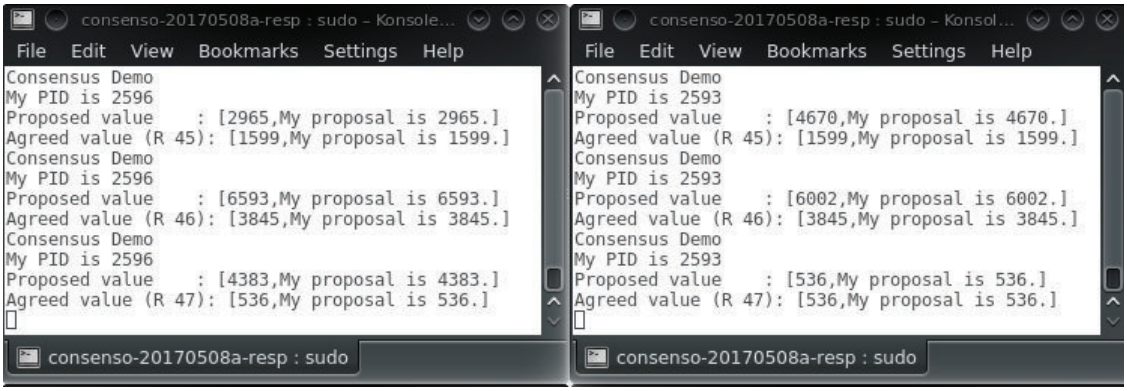

Figura 5.16. Consenso de valores con la mayoría de procesos (parte 2)

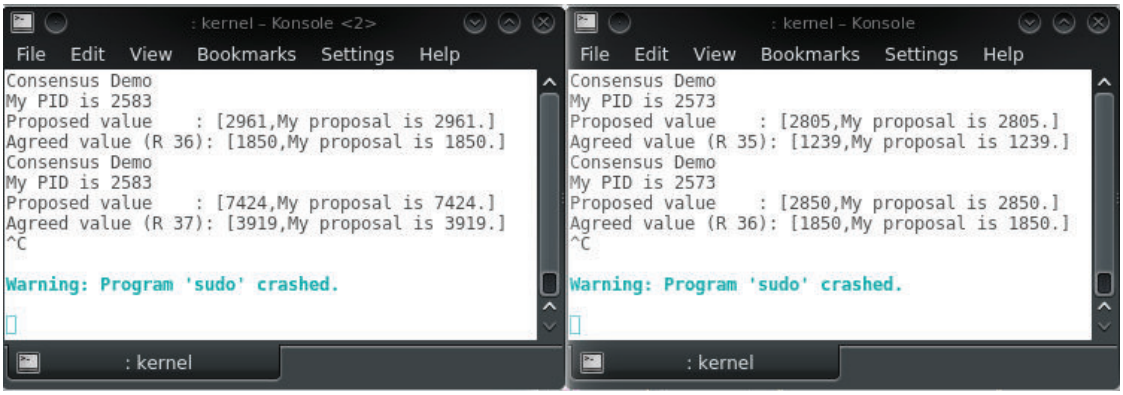

Figura 5.17. Consenso de valores con la mayoría de procesos (parte 3)

5.2.3. PRUEBA 3

Utilizando los procesos de la prueba anterior. Si se provoca un fallo en un proceso más, el algoritmo de Consenso debería dejar de funcionar. Efectivamente en la Figura 5.18, Figura 5.19 y Figura 5.20 se muestra que el algoritmo se queda bloqueado y no continúa consensuando valores. Los procesos que continuaron consensuando valores en la prueba anterior quedaron en (R 53) luego de provocar un fallo en la mayoría de procesos.

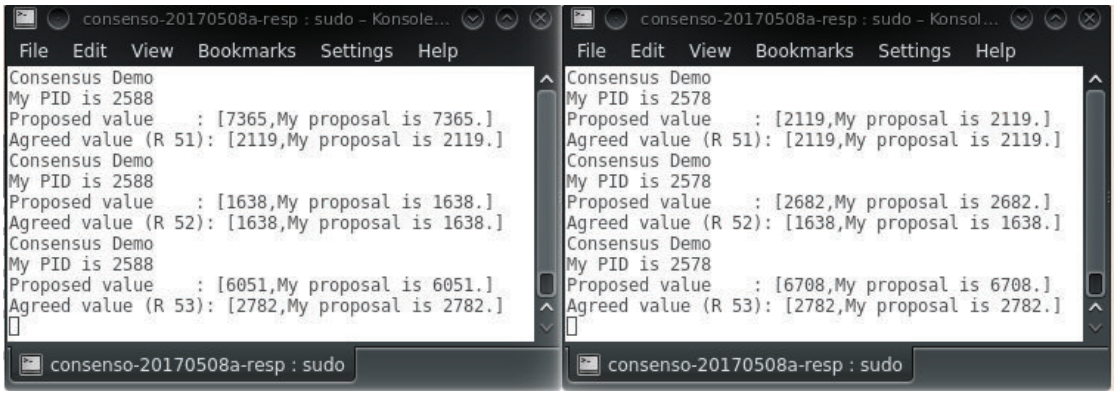

Figura 5.18. Consenso de valores con la minoría de procesos (parte 1)

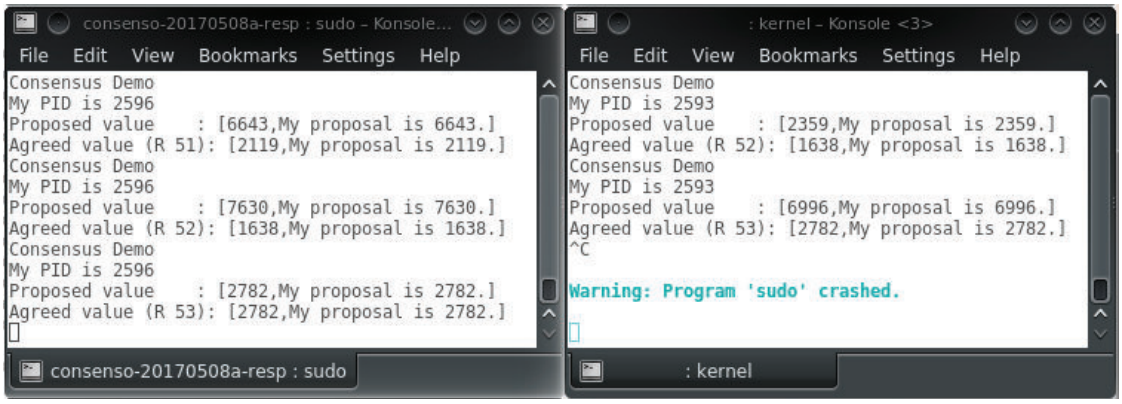

Figura 5.19. Consenso de valores con la minoría de procesos (parte 2)

5.3. APLICACIÓN DE MENSAJERÍA

5.3.1. PRUEBA 1

En la Figura 5.21 muestra como inician los procesos (cada proceso represena a un servidor de la topología de la Figura 4.1), cada uno con diferente orden de mensajes.

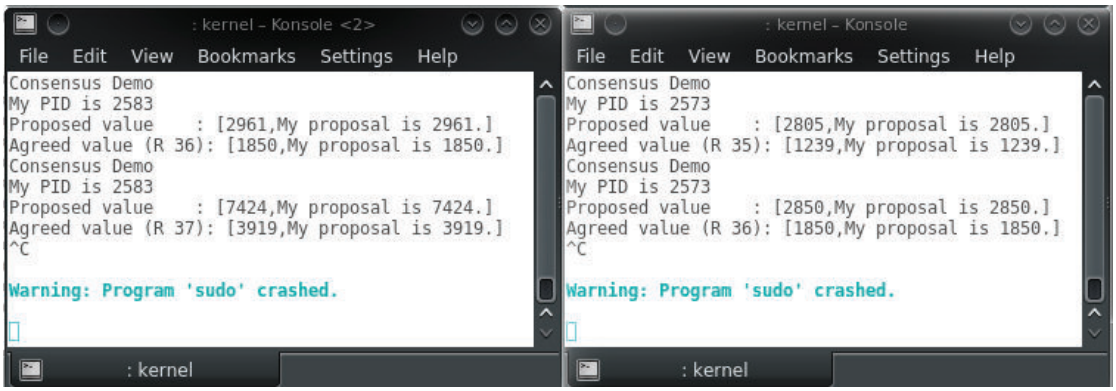

Figura 5.20. Consenso de valores con la minoría de procesos (parte 3)

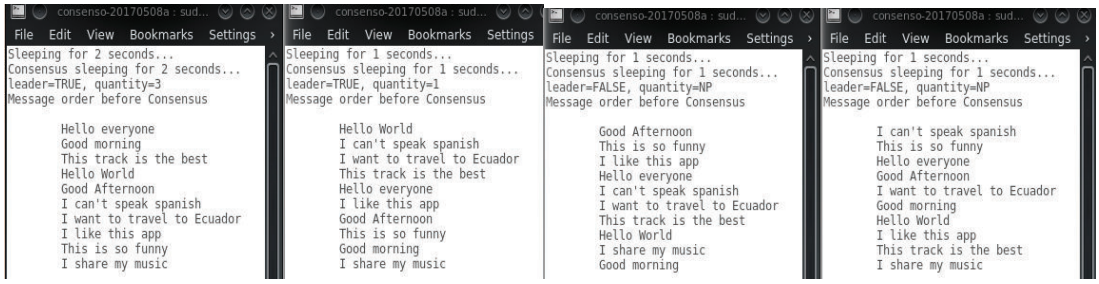

Figura 5.21. Resultados de una ejecución (parte 1)

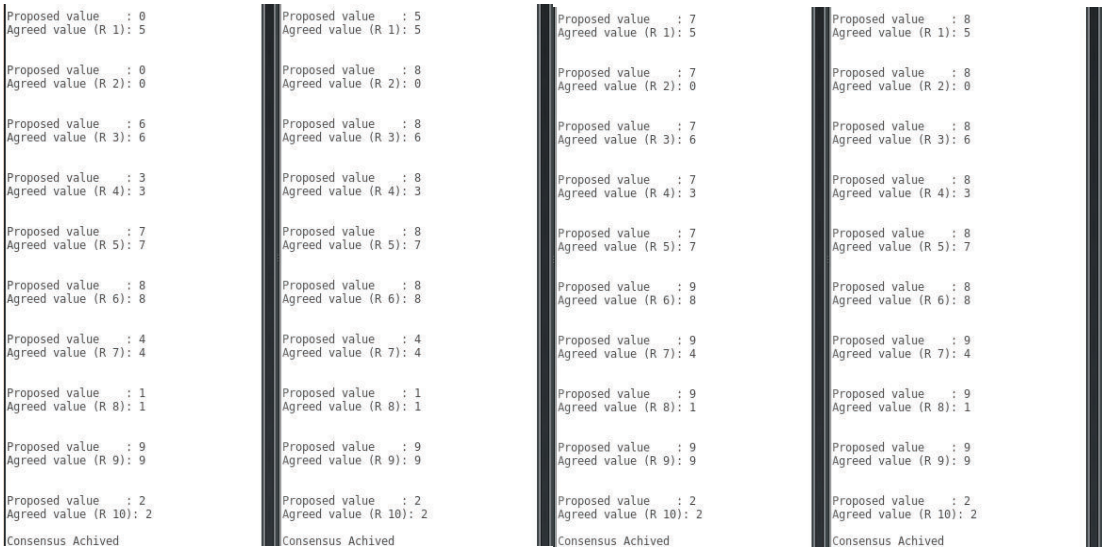

Figura 5.22. Resultados de una ejecución (parte 2)

Luego, en la Figura 5.22, cada proceso propone un valor y se llega a un acuerdo en el orden los mensajes. El resultado de la ejecución se muestra en la Figura 5.23, todos los procesos tienen el mismo orden de los mensajes. Para comprobar que en cada

ejecución el orden en el que inician los mensajes es diferente, se vuelve a ejecutar los cuatro procesos. La Figura 5.24 muestra el orden de los mensajes con el que parte cada proceso. Al comparar con la Figura 5.21 se puede fácilmente notar que el orden es diferente. En esta ejecución, al igual que la anterior los procesos consensuan los valores (Figura 5.25) y llegan a un orden en común (Figura 5.26).

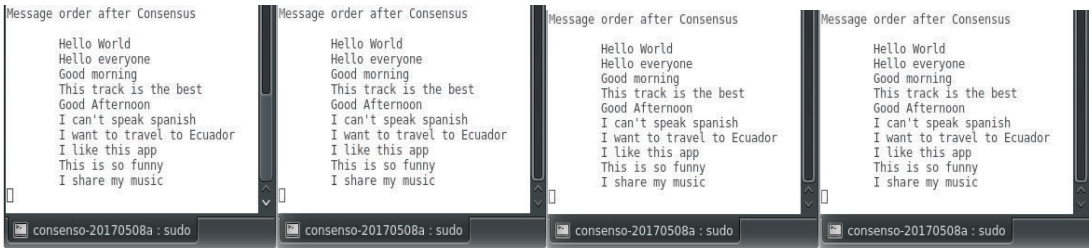

Figura 5.23. Resultados de una ejecución (parte 3)

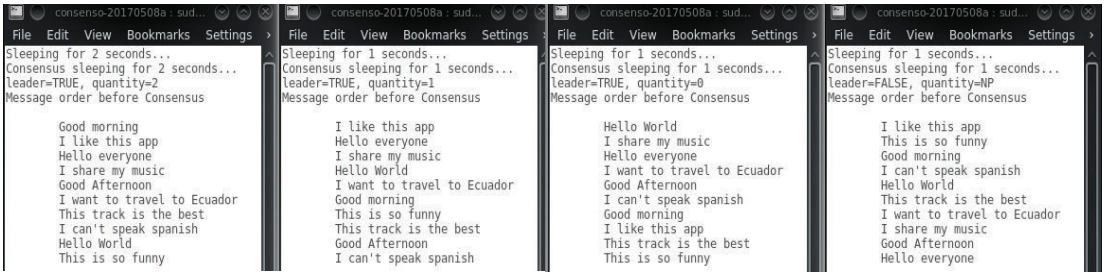

Figura 5.24. Resultados de la segunda ejecución (parte 1)

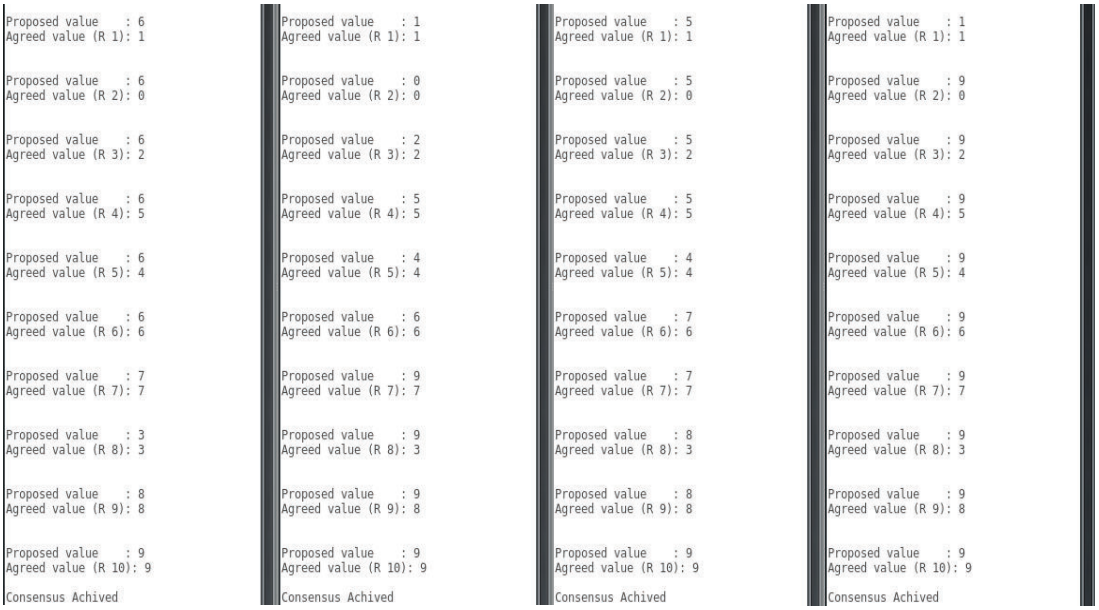

Figura 5.25. Resultados de la segunda ejecución (parte 2)

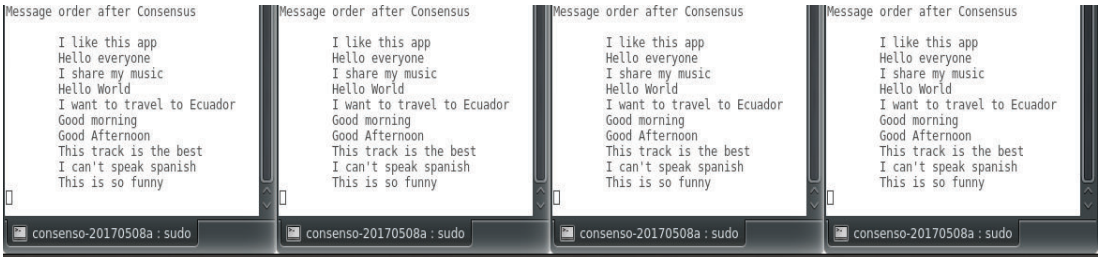

Figura 5.26. Resultados de la segunda ejecución (parte 3)

CAPÍTULO 6
CONCLUSIONES Y RECOMENDACIONES

6.1. CONCLUSIONES

- Los sistemas distribuidos asíncronos presentan varias ventajas frente a los sistemas distribuidos síncronos. Una de ellas es que no restringe límites de tiempo para la transmisión de mensajes o ejecución de procesos. Esto permite que el sistema que soporta la aplicación no deba satisfacer requerimientos estrictos de tiempo. Sin embargo, al combinar un sistema asíncrono y procesos que pueden fallar, se obtiene un escenario en donde se hace imposible detectar si un proceso ha fallado o simplemente su ejecución es lenta. Para solucionar este problema se hizo uso de un Detector de Fallos que proporcione información de procesos fallidos.
- El algoritmo Detector de Fallos y el algoritmo de Consenso implementados en el presente trabajo no hacen uso de identificadores de red únicos para su funcionamiento. Las direcciones MAC origen y destino de la trama Ethernet se configuraron como 04:04:04:04:04:04 y FF:FF:FF:FF:FF:FF respectivamente. Se destaca que es posible inventar métodos como *sniffers* en un puerto de un *switch* para rastrear el origen de los mensajes. Sin embargo, identificar al originador del mensaje de forma explícita es muy difícil, por lo que la identidad de los involucrados en la comunicación es anónima.
- A medida que el número de líderes en el sistema aumenta. El algoritmo de Consenso tardará más en consensuar un valor. Esto se debe a que los procesos líderes son quienes proponen valores, si existen más valores propuestos diferentes, es probable que se requiera un mayor número de rondas para consensuar un valor.
- Un Detector de Fallos no se define en términos de una implementación en particular, es decir, que es posible descomponer un protocolo en varias partes y una de estas sea el Detector de Fallos. De este modo, se asume una clase particular que puede resolver el problema y realizar su correspondiente implementación y esta sea independiente de las demás partes de la aplicación.

- El Detector de Fallos implementado permite la adición de nuevos procesos o equipos al sistema una vez puesto en marcha sin necesidad de reiniciarlo. Esta característica se conoce como dinamismo, o introducción en caliente, y es importante debido a que no es necesario conocer a priori la cantidad de procesos que van participar en el sistema.
- La recuperación de fallos combinado con dinamismo permite que los procesos que fallan se vuelvan a integrar al sistema sin necesidad de parar y reiniciar todo.
- El uso del valor de las variables que se recuperan de almacenamiento estable fueron optimizadas, debido a que el acceso de lectura y escritura del disco duro es lento a comparación de la memoria volátil.
- La variable `runId` fue introducida en la implementación del algoritmo de Consenso, para identificar el Consenso que se realiza. Es decir, una aplicación puede llamar al método `propose()` varias veces, para identificar esa llamada se utiliza la variable `runId`.
- La no fiabilidad del Detector de Fallos en la implementación radica en que la variable `quantity`, que indica el número de líderes en el sistema tarda en estabilizarse en el valor correcto. Sin embargo, esta variable tarde o temprano toma el valor correcto.
- El acceso a los arreglos que contienen los mensajes de los algoritmos se realiza utilizando control de concurrencia. Al ser algoritmos multi-hilo, se evita que mientras un hilo haga uso de un arreglo otro intente acceder al mismo recurso causando conflicto.

6.2. RECOMENDACIONES

- La implementación de ambos algoritmos se realizó en lenguaje C++, por lo que puede ser simulado en OMNeT++. Mediante esta herramienta es posible visualizar detalladamente el intercambio de mensajes y el funcionamiento de los algoritmos.
- Es posible extender el uso del código a otros sistemas operativos, tomando en cuenta que la librería PCAP tiene algunas particularidades dependiendo del sistema operativo sobre el cual se ejecuta, sin embargo, el procedimiento para capturar y generar paquetes es muy similar al presentado en este trabajo.

- Implementar diferentes cálculos deterministas para decidir un valor en el algoritmo de Consenso. Junto a ello la función `compareTo()` correspondiente. Por ejemplo: una aplicación que consensue variables de tipo: caracteres, imágenes, cadenas de caracteres, entre otros.

REFERENCIAS BIBLIOGRÁFICAS

[1] Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (1989). *Distributed systems*. Pearson Education A basic understanding of distributed systems as it is offered, 20-65.

[2] Chandra, T. D., & Toueg, S. (1996, Mar). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 225-267.

[3] Cachin, C., Guerraoui, R., & Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

[4] Stoller, S. D. (2000). Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, 49(3), 283-284.

[5] Pedone, F., & Schiper, A. (1999, September). Generic broadcast. In *International Symposium on Distributed Computing* (pp. 94-106). Springer Berlin Heidelberg.

[6] Guerraoui, R., & Schiper, A. (1997). Genuine atomic multicast. *Distributed Algorithms*, 141-154.

[7] Chandra, T. D., Hadzilacos, V., & Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4), 685-722.

[8] Fischer, M. J., Lynch, N. A., & Paterson, M. S. (1985, Abr). Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2), 374-382.

[9] Chandra, T. D., Hadzilacos, V., & Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4), 685-722.

[10] Larrea, M., Fernández, A., & Arévalo, S. (2000). Optimal implementation of the weakest failure detector for solving consensus. In *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on* (pp. 52-59). IEEE.

[11] Jiménez, E., Arévalo, S., & Fernández, A. (2006). Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, 100(2), 60-63.

[12] Anta, A. F., Jiménez, E., & Raynal, M. (2010). Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *Journal of Computer Science and Technology*, 25(6), 1267-1281.

[13] Aguilera, M. K., Chen, W., & Toueg, S. (2000). Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2), 99-125.

[14] Joshi, J. (2008). *Network security: know it all*. Morgan Kaufmann.

[15] Dingledine, R., Mathewson, N., & Syverson, P. (2004). *Tor: The second-generation onion router*. Naval Research Lab Washington DC.

[16] Onwuzurike, L., & De Cristofaro, E. (2015). Experimental Analysis of Popular Smartphone Apps Offering Anonymity, Ephemerality, and End-to-End Encryption. arXiv preprint arXiv:1510.04083.

[17] Wang, G., Wang, B., Wang, T., Nika, A., Zheng, H., & Zhao, B. Y. (2014, November). Whispers in the dark: analysis of an anonymous social network. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (pp. 137-150). ACM.

[18] Nemelka, C. L., Ballard, C. L., Liu, K., Xue, M., & Ross, K. W. (2015, November). You can yak but you can't hide. In *Proceedings of the 2015 ACM on Conference on Online Social Networks* (pp. 99-99). ACM.

[19] Bonnet, F., & Raynal, M. (2010, September). Anonymous asynchronous systems: the case of failure detectors. In *International Symposium on Distributed Computing* (pp. 206-220). Springer Berlin Heidelberg.

[20] Bonnet, F., & Raynal, M. (2011). The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. *ACM Transactions on Autonomous and Adaptive Systems* (TAAS), 6(4), 23.

[21] Jiménez, E., Arévalo, S., Herrera, C., & Tang, J. (2015). Eventual election of multiple leaders for solving consensus in anonymous systems. *The Journal of Supercomputing*, 71(10), 3726-3743.

[22] Chu, F. C. (1998). Reducing Omega to Diamond W. Inf. Process. Lett., 67(6), 289-293.

[23] Guerraoui, R. (2000, July). Indulgent algorithms (preliminary version). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (pp. 289-297). ACM.

[24] Hurfin, M., & Raynal, M. (1999). A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4), 209-223.

[25] Schiper, A. (1997). Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3), 149-157.

[26] Hurfin, M., Mostéfaoui, A., & Raynal, M. (2002). A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Transactions on Computers*, 51(4), 395-408.

[27] Mostéfaoui, A., & Raynal, M. (1999, September). Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *International Symposium on Distributed Computing* (pp. 49-63). Springer Berlin Heidelberg.

[28] Librería PCAP, [En línea] http://www.tcpdump.org/

[29] McKusick, M. K., Neville-Neil, G. V., & Watson, R. N. (2014). *The design and implementation of the FreeBSD operating system*. Pearson Education.

[30] Eastlake, D. (2008). IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters.

[31] Linux man page mkstemp, [En línea] https://linux.die.net/man/3/mkstemp.

ANEXOS

Anexo A: Código fuente C++ del algoritmo del Detector de Fallos.

Anexo B: Código fuente C++ del algoritmo de Consenso.

Anexo C: Código fuente C++ de la aplicación de prueba de mensajería.

El código desarrollado se encuentra en el CD adjunto a este documento.