Departamento de Informática y Electrónica

UNIVERSITAT DE VALÈNCIA

Aportaciones a los Entornos de Desarrollo de Aplicaciones Tolerantes a Fallos

Tesis Doctoral

Presentada por:

D. Carlos Pérez Conde

Dirigida por:

D. Gregorio Martín Quetglás

D. Germán Fabregat Llueca

UNIVERSITAT DE VALÈNCIA

FACULTAT DE FISICA
DEPARTAMENT D'INFORMATICA I ELECTRONICA
C/ Doctor Moliner, 50
46100 - BURJASSOT (València)

D. Gregorio Martín Quetglás, Catedrático de Ciencias de la Computación del Departamento de Informática y Electrónica de la Universidad de Valencia y **D. Germán Fabregat Llueca**, Profesor Titular Interino de Escuela Universitaria del Departamento de Informática de la Universidad Jaume I de Castellón,

HACEN CONSTAR: Que el licenciado en Ciencias Físicas **D. Carlos Pérez Conde** ha realizado bajo nuestra dirección, en los laboratorios del Departamento de Informática y Electrónica, el trabajo titulado "**Aportaciones a los Entornos de Desarrollo de Aplicaciones Tolerantes a Fallos**.", que se presenta para optar al grado de Doctor en Ingeniería Informática.

Y para que así conste, firman el presente certificado en Valencia a treinta y uno de julio de mil novecientos noventa y siete.

Fdo.: Gregorio Martín Quetglás

Fdo.: Germán Fabregat Llueca

A Pepa, por lo que no hemos podido compartir,
a mis padres, por la educación que me han dado,
y a David, por su comprensión y su ayuda.

Agradecimientos

Son muchas las personas que, de una u otra forma, han contribuido a que haya podido llevar a cabo el trabajo de investigación que se expone en la presente memoria, y a todas ellas quiero agradecer sinceramente su ayuda. Desde mi familia y amigos, que me han proporcionado ánimos de forma continua, hasta mis compañeros de departamento, que se han encargado por completo de tareas en las que yo hubiera debido colaborar, permitiéndome así concentrarme en la investigación.

Me gustaría agradecer a mis compañeros de FASST su colaboración y las cosas que aprendí de ellos en el transcurso del proyecto. A Vicente Cerverón le debo el haberme iniciado en el mundo de los sistemas operativos y a Rafael Martínez el haberme ayudado a poner en práctica una buena parte de los conocimientos adquiridos sobre arquitectura de computadores.

De forma especial, deseo agradecer a mis directores, Gregorio Martín y Germán Fabregat, la confianza que han depositado en mí desde el principio, su apoyo y sus consejos, sin los cuales nunca hubiera podido completar este trabajo. En particular, a Gregorio le quiero agradecer, además de la oportunidad de trabajar en el proyecto FASST, su implicación personal en mi Tesis Doctoral. A Germán le agradezco profundamente el haber participado en el día a día de este trabajo de investigación, ayudándome a poner orden tras mis tormentas de ideas, acompañándome en viajes y, según sus propias palabras, disfrutando con todo ello.

También quiero agradecer a Pedro de Miguel y a Antonio Pérez, de la Universidad Politécnica de Madrid, la orientación a la hora de establecer los objetivos iniciales de la investigación, así como las revisiones y las recomendaciones posteriores, que han constituido un estímulo constante a la hora de llevarla a cabo.

Carlos Pérez Conde

Índice general

Introducción 1

Antecedentes del trabajo de investigación 3
Planteamiento 4
Objetivos 5
Estructura de la memoria 5

Capítulo 1
Tolerancia a fallos: situación actual 7

1.1 Revisión de las técnicas de tolerancia a fallos 8
1.1.1 Detección y confinamiento de errores 8
1.1.2 Recuperación hacia atrás 9
1.1.3 Redundancia modular 16
1.2 Análisis de la estructura de los sistemas tolerantes a fallos 21
1.2.1 Descripción de la estructura 21
1.2.2 Nivel de enmascaramiento de fallos 23
1.2.3 Distribución de técnicas por niveles 25
1.3 Ejemplos de sistemas relacionados 29
1.3.1 Tandem's Integrity S2 29
1.3.2 Sistema Avanzado de Automatización 30
1.3.3 Manetho 31
1.3.4 Delta-4 32
1.3.5 Tolerancia a fallos mediante el soporte lógico en AT&T 34
1.3.6 Mecanismo Sentry 35
1.4 Conclusiones 36

Capítulo 2
Propuesta y justificación de la arquitectura 39

2.1 Elección del nivel de enmascaramiento de fallos 39
2.2 Distribución de técnicas por niveles 40
2.3 Arquitectura del soporte físico 42
2.3.1 Descripción de la arquitectura 43
2.3.2 Justificación 43
2.4 Arquitectura del soporte lógico 46
2.4.1 Descripción de la arquitectura 46
2.4.2 Justificación 48
Conclusiones 50

Capítulo 3
Función del micro-núcleo en aplicaciones tolerantes a fallos: necesidades **51**

3.1 Los puntos de recuperación 52
3.2 Los registros 53
3.3 La redundancia modular 55
3.4 La detección y confinamiento de errores 58
3.5 Conclusiones 61

Capítulo 4
Especificación del micro-núcleo **63**

4.1 Las operaciones incrementales 64
4.1.1 Mensajes incrementales 65
4.1.2 Mensajes incrementales empaquetados 66
4.1.3 Mensaje incremental negativo 68
4.1.4 Recepción de la cabecera 69
4.1.5 Gestión de varios destinos 72
4.1.6 Copia incremental 72
4.2 La interfaz del micro-núcleo 73
4.2.1 Gestión de procesos 73
4.2.2 Paso de mensajes 77
4.2.3 Gestión de memoria 80
4.3 Aspectos adicionales sobre los servicios 84
4.3.1 Gestión de puntos de recuperación 84
4.3.2 Gestión de registros 87
4.3.3 Detección y confinamiento de errores 88
4.3.4 Transferencia de mensajes incrementales entre nodos 89
4.4 Conclusiones 90

Capítulo 5
Evaluación de la especificación del micro-núcleo **93**

5.1 Servicios tolerantes a fallos mediante grupos de servidores débilmente sincronizados 94
5.2 Puntos de recuperación desde una librería a nivel de usuario 96
5.3 Implementación de Manetho con los servicios del micro-núcleo 99
5.4 Servicios transaccionales 106
5.5 Revisión de otras técnicas y sistemas tolerantes a fallos 113
5.6 Conclusiones 114

Capítulo 6
Primeros resultados y evaluación **117**

6.1 Mensajes incrementales 118

6.1.1 Estudio básico de las prestaciones 119
6.1.2 Estudio del establecimiento de puntos de recuperación 123
6.1.3 Análisis de resultados 125
6.2 Servicio transaccional de ficheros 126
6.2.1 El sistema de ficheros 127
6.2.2 Estructura del servicio 129
6.2.3 Algoritmos 134
6.2.4 Estudio de las prestaciones 138
6.2.5 Análisis de resultados 140
6.3 Conclusiones 141

Capítulo 7
Discusión y conclusiones 143

7.1 Discusión de resultados 143
7.2 Resumen de aportaciones y trabajo futuro 147

Bibliografía 151

Anexo I
Tolerancia a fallos: conceptos básicos y metodología de diseño 157

A.I.1 Concepto de sistema 157
A.I.2 Garantía de funcionamiento 158
A.I.3 Tolerancia a fallos 161
A.I.4 Metodología de diseño 163

Anexo II
Interfaz detallada del paso de mensajes incrementales 167

A.II.1 Mensajes incrementales en el micro-núcleo 168
A.II.2 La estructura mensaje en el emisor 169
A.II.3 Algoritmo enviar 170
A.II.4 La estructura mensaje en el receptor 171
A.II.5 Algoritmo recibir 172
A.II.6 Gestión de la cola auxiliar del puerto 174
A.II.7 Ejemplos 174

Anexo III
Descripción de Manetho 181

A.III.1 Protocolo de comunicaciones 182
A.III.2 Protocolo de establecimiento de puntos de recuperación 184
A.III.3 Protocolo de difusión 185

A.III.4 Protocolo de recuperación 185

Anexo IV
Implementación de las operaciones incrementales 189

A.IV.1 Descripción de la interfaz 189
A.IV.1.1 Correspondencia entre servicios 190
A.IV.1.2 Recepción de la cabecera 190
A.IV.1.3 MIs negativos 191
A.IV.1.4 Interfaz para cliente/servidor 191
A.IV.1.5 Consistencia entre tipos de mensajes 191
A.IV.2 Gestión de los datos en el micro-núcleo 190
A.IV.2.1 Descripción de la estructura de datos 193
A.IV.2.2 Generación de MIEs 194
A.IV.2.3 Consumo de MIEs 195
A.IV.2.4 Gestión de la cola auxiliar 196
A.IV.2.5 Discusión 197
A.IV.3 Gestión del bit de modificado 197
A.IV.3.1 Descripción del problema 197
A.IV.3.2 Descripción de la solución 198
A.IV.3.3 Comprobación 199

Anexo V
Estudio complementario de las prestaciones de las operaciones incrementales 201

A.V.1 Medidas sin robo de páginas 201
A.V.2 Medidas con robo de páginas 204

Índice de ilustraciones

1-1 Recuperación hacia atrás utilizando un checkpoint. 10
1-2 Recuperación hacia atrás utilizando un registro. 10
1-3 Recuperación hacia atrás utilizando checkpoints y registros. 11
1-4 Ejemplo de sistema que pasa a un estado inconsistente tras la vuelta atrás de uno de sus componentes. 12
1-5 Bloque de recuperación con tres módulos lógicos. 15
1-6 Redundancia modular. 16
1-7 Ejemplo de redundancia activa, repuesto en espera (standby sparing). 17
1-8 Niveles del sistema. 22
1-9 Arquitectura del soporte físico en Delta-4. 33
1-10 Arquitectura del soporte lógico en Delta-4. 33
1-11 Flujo de ejecución de una llamada al sistema en Mach 3.0/UX. 35
1-12 Flujo de ejecución de una llamada al sistema con el mecanismo Sentry activado. 36

2-1 Separación entre mecanismos y políticas para el desarrollo de aplicaciones tolerantes a fallos. 42
2-2 Arquitectura del soporte físico. 43
2-3 Arquitectura del soporte lógico. 47

3-1 Necesidades comunes relativas al estado de los procesos. 52
3-2 Necesidades comunes relativas al registro de mensajes. 53
3-3 Necesidades comunes relativas al paso intermedio de mensajes. 56
3-4 Resumen de las necesidades comunes a nivel del micro-núcleo. 62

4-1 Mensajes incrementales. 65
4-2 Mensaje incremental empaquetado (MIE). 66
4-3 Transferencia de mensajes incrementales empaquetados. 67
4-4 Mensajes incrementales negativos. 69
4-5 Reordenación de mensajes mediante código de librería. 70
4-6 Operaciones para la gestión de la cola auxiliar. 71
4-7 Copia incremental. 73
4-8 Servicios relativos a la gestión de procesos. (I) 76
4-9 Servicios relativos a la gestión de procesos. (II) 77
4-10 Servicios relativos al intercambio de mensajes. (I) 79
4-11 Servicios relativos al intercambio de mensajes. (II) (Gestión de la cola auxiliar de los puertos). 80
4-12 Servicios relativos a la gestión de memoria virtual. 82
4-13 Servicios requeridos por el micro-núcleo a los servidores de puntos de recuperación. 85

4-14 Establecimiento de un punto de recuperación del proceso P_1, solicitado por el proceso P_2. 86
4-15 Resumen de la dependencia que presenta el micro-núcleo de los servicios de gestión ofrecidos por el servidor de puntos de recuperación. 86
4-16 Servicios requeridos por el micro-núcleo a los servidores de registros. 87
4-17 Ejemplo de utilización de recibir con cabecera como medio para la detección y confinamiento de errores. 89
4-18 Transmisión de mensajes incrementales entre nodos utilizando mensajes incrementales empaquetados. 90

5-1 Tolerancia a fallos mediante grupos de servidores débilmente sincronizados. 95
5-2 Utilización de los MIEs para mantener en un fichero una réplica actualizada de una zona de memoria virtual. 97
5-3 Algoritmo para actualizar el fichero de respaldo a partir de un mensaje incremental empaquetado (MIE). 98
5-4 Ejemplo de distribución de RUs en la propuesta para la implementación de Manetho. 99
5-5 Estructura del soporte lógico para la implementación de los algoritmos de Manetho. 100
5-6 Estructura para la difusión de mensajes en unidades de recuperación replicadas. 102
5-7 Estructura del servicio transaccional básico. 107
5-8 Interfaz del servicio transaccional básico. 107
5-9 Estructura del servicio transaccional tolerante a fallos. 108
5-10 Interfaz ofrecida por el proceso R. 109
5-11 Estructura del servicio transaccional para dos transacciones concurrentes. 111

6-1 Variación del coste de los mensajes incrementales en función del tamaño de la región de memoria virtual con 10 páginas modificadas. 120
6-2 Variación del coste de los mensajes incrementales en función del número de páginas incluidas en el mismo, para regiones de memoria virtual de 100 y de 500 páginas respectivamente. 121
6-3 Coste temporal de los mensajes incrementales en función del tamaño de la región de memoria virtual, para 10, 100, 300, 500, 750 y 1000 páginas modificadas. 122
6-4 Estructura del sistema de ficheros cfs. 129
6-5 Estructura del servicio transaccional de ficheros. 130
6-6 Interfaz ofrecida por el proceso ST a los procesos T_i. 131
6-7 Algoritmo para comenzar una transacción en los procesos T_i. 134
6-8 Algoritmo para comenzar una transacción en el proceso ST. 134
6-9 Algoritmo para terminar una transacción en los procesos T_i. 135
6-10 Algoritmo para terminar una transacción en el proceso ST. 136
6-11 Algoritmo para abortar una transacción en los procesos T_i. 137
6-12 Algoritmo para abortar una transacción en el proceso ST. 137

A.I-1 Correspondencia entre el origen de los fallos y su duración. 160
A.I-2 Medios para la obtención de la garantía de funcionamiento. 161
A.I-3 Componente tolerante a fallos idealizado. 165

A.II-1 La estructura de un puerto para el paso de MIs. 168
A.II-2 Algoritmo para el envío de mensajes. 170
A.II-3 Algoritmo para la recepción de mensajes. 173
A.II.4 Algoritmo para el proceso P_1. 175
A.II.5 Algoritmo para el proceso P_2. 175
A.II.6 Algoritmo para el proceso SC_1. 176
A.II.7 Algoritmo para el proceso SC_2. 176
A.II.8 Algoritmo para el proceso P_1. 177
A.II.9 Algoritmo para el proceso P_2. 177
A.II-10 Estructura de las réplicas. 178
A.II-11 Algoritmo para el hilo receptor. 179
A.II-12 Función de librería recibirL, versión para el líder. 179
A.II-13 Función de librería recibirL, versión para los seguidores. 180

A.III-1 Algoritmo para enviar y recibir mensajes de otras RUs. 183
A.III-2 Algoritmos enviar y recibir información del mundo exterior (ME). 183
A.III-3 Algoritmos para establecer puntos de recuperación coordinados. El algoritmo para la RU coordinadora se muestra en la parte de la izquierda y el del resto de RUs en la parte derecha. 184
A.III-4 Algoritmos para coordinar el procesamiento de mensajes en una RU replicada. 185
A.III-5 Protocolo de recuperación. 186

A.IV-1 Representación de los mensajes incrementales en el micro-núcleo. 193
A.IV-2 Copia virtual con dos destinos independientes. 199

A.V-1 Valor medio del número de páginas incluidas en cada mensaje incremental en función del tamaño de la región de memoria virtual en el caso aleatorio. 202
A.V-2 Variación del coste de los mensajes incrementales negativos en función del tamaño de la región de memoria virtual, con 10 páginas modificadas. 203
A.V-3 Coste de mensajes incrementales sucesivos para una región de memoria virtual de 5000 páginas. 204
A.V-4 Coste temporal de los mensajes incrementales en función del número de páginas modificadas para regiones de 500, 1000, 1500, 2000, 3000, 4000 y 5000 páginas. 206

Índice de tablas

1-1 Resumen conjunto de las ventajas e inconvenientes de enmascarar los fallos y de incorporar las técnicas de tolerancia a fallos en cada uno de los niveles del sistema. 29

2-1 Resumen de las características y las ventajas de la arquitectura propuesta para el soporte físico. 44

2-2 Resumen de las actividades desarrolladas en los niveles del núcleo del sistema operativo y de utilidades. 48

4-1 Posibilidades en la transferencia de mensajes incrementales y mensajes incrementales empaquetados. 68

4-2 Revisión de la utilidad de los servicios en base a las necesidades a nivel del núcleo. 83

5-1 Distribución de los elementos básicos de la estructura de datos de Manetho entre los diferentes componentes lógicos. 101

5-2 Resumen de la función de los diferentes componentes lógicos en cada uno de los cuatro algoritmos básicos de Manetho. 104

6-1 Coste de establecer puntos de recuperación utilizando operaciones incrementales. 124

6-2 Comparación del coste de las operaciones de control del servicio transaccional de ficheros (ctfs) con cfs, tmpfs y FAT. 139

6-3 Comparación del coste de las operaciones de lectura y escritura del servicio transaccional de ficheros (ctfs) con cfs, tmpfs y FAT. 139

6-4 Coste de confirmar y cancelar una transacción tras haber modificado (escribir) diferentes cantidades de datos. 140

A.II-1 Descripción de los indicadores que puede utilizar el emisor. 169

A.II-2 Descripción de los indicadores que puede utilizar el receptor. 172

A.III-1 Necesidades de acceso a las estructuras de datos desglosadas por actividades y protocolos. 187

A.IV-1 Correspondencia entre enviar/recibir y msg_send/msg_receive/msg_reply en función del indicador M_READ. 190

A.IV-2 Descripción de los identificadores que pueden utilizar el cliente y el servidor para el intercambio de MIs. 191

A.IV-3 Consistencia entre tipos de mensajes. 192

A.IV-4 Técnicas utilizadas para la generación de los MIEs en función de los indicadores utilizados por el emisor. 195

A.IV-5 Información obtenida por el receptor en función del contenido del MIE y de los indicadores que utiliza. _____ 196

Introducción

En la sociedad actual, los sistemas informáticos desempeñan un elevado número de funciones relativas a aspectos tan diferentes como el control de procesos en cadenas automatizadas de producción o la realización de transacciones bancarias. Este hecho pone de manifiesto la confianza que se tiene en los equipos informáticos, ya que de su correcto funcionamiento dependen una gran cantidad de actividades cotidianas.

Precisamente, la necesidad de disponer de sistemas informáticos en los que se pueda depositar justificadamente esta confianza se puede expresar mediante el concepto de garantía de funcionamiento.[1]

Los medios para alcanzar la garantía de funcionamiento comprenden todas las fases de la vida de los sistemas informáticos, distinguiéndose entre prevención de fallos y tolerancia a fallos. La finalidad de la prevención de fallos es la producción de un sistema libre de ellos, mientras que la tolerancia a fallos asume que el sistema en funcionamiento contendrá fallos que no habrán sido eliminados o que se habrán generado con el uso.

[1] En el anexo I se definen de forma precisa los conceptos básicos utilizados en la presente memoria, para permitir una descripción clara de las técnicas de tolerancia a fallos, de los sistemas y del trabajo realizado.

Para que los computadores sean capaces de tolerar los fallos, es decir, continuar llevando a cabo su función aunque se averíen algunos de sus componentes, es habitual introducir más componentes de los estrictamente necesarios para desempeñar la función para la que han sido diseñados. No obstante, la utilización de este tipo de estrategias conlleva una serie de costes adicionales que han evitado su utilización generalizada. Durante años, la tolerancia a fallos se ha utilizado casi exclusivamente en computadores aplicados a campos especialmente críticos —por ejemplo, en el control de vuelo de los aviones—.

El abaratamiento de los componentes y de la producción de computadores digitales ha permitido, y sigue permitiendo, por un lado, la incorporación de los equipos informáticos a nuevos ámbitos, como el de las pequeñas y medianas empresas o el doméstico, y por otro, que desempeñen un mayor número de funciones.

Como consecuencia, cada vez existe una mayor dependencia del correcto funcionamiento de los sistemas informáticos. Por ejemplo, cada vez más empresas dependen de sus sistemas informáticos para realizar actividades tan esenciales para su actividad diaria como el control de existencias o la facturación, de forma que un fallo del equipo informático provoca el cese de la actividad, traduciéndose inmediatamente en pérdidas económicas.

El avance de la tecnología, el desarrollo de nuevas herramientas —la mayoría basadas a su vez en los computadores actuales—, y la utilización de metodologías más refinadas para el diseño y la implementación de las nuevas generaciones de computadores digitales, favorecen la prevención de los fallos. No obstante, la enorme complejidad de los sistemas, no sólo en cuanto al soporte físico (o *hardware*), sino también en cuanto al soporte lógico (o *software*); junto con la fuerte competencia en el sector informático —que obliga a los diseñadores de sistemas a conseguir las máximas prestaciones en el menor tiempo y al mejor precio posibles—, hacen que los sistemas en funcionamiento continúen conteniendo fallos, y justificando, por lo tanto, la utilización de técnicas de tolerancia a fallos para ofrecer una mayor garantía de funcionamiento.

Una de las razones que han influido notablemente en el abaratamiento de los sistemas informáticos ha sido la aparición de plataformas de propósito general, es decir, sistemas que no han sido diseñados para resolver un problema específico, sino para desarrollar y ejecutar sobre ellos aplicaciones que puedan resolver problemas muy diferentes entre sí. Un ejemplo bien conocido sería una estación de trabajo, o un conjunto de ellas conectadas por una red de área local, ejecutando el sistema operativo UNIX, y sobre el que los usuarios desarrollan y utilizan aplicaciones como compiladores, programas para la preparación de documentos, bases de datos, programas para el control de existencias, etc.

Por todo ello, cada vez está más justificada la incorporación de técnicas de tolerancia a fallos a las plataformas de propósito general, puesto que los recursos necesarios (aumento de componentes, reducción de prestaciones) para la implantación de estas técnicas se ven compensados económicamente al aumentar la garantía de funcionamiento del sistema. Sin embargo, todavía hoy es difícil encontrar este tipo de técnicas en las plataformas de propósito general. Entre las razones por las que no se

utilizan técnicas de tolerancia a fallos en estos entornos cabría destacar las dos siguientes.

- Los programadores que utilizan estas plataformas para desarrollar sus propias aplicaciones no disponen, normalmente, de la formación o del tiempo necesario para incluirlas ellos mismos. Por lo tanto, sería necesario que las plataformas de propósito general incluyesen técnicas que fueran sencillas de utilizar por sus usuarios.
- La inclusión de técnicas de tolerancia a fallos supone un aumento del coste y/o una disminución de las prestaciones, en función de las técnicas empleadas, lo cual hace disminuir la competitividad del sistema en el mercado. Para evitarlo, desde el punto de vista del desarrollo posterior de aplicaciones tolerantes a fallos, sería necesario que los sistemas incluyesen técnicas eficientes y selectivas, en el sentido que la disminución de las prestaciones fuera reducida, afectando solamente a aquellas aplicaciones que realmente utilizan las técnicas de tolerancia a fallos y que, por lo tanto, obtienen una contrapartida en forma de un aumento de la garantía de funcionamiento.

A partir de estas reflexiones, surge la motivación principal que ha marcado el curso del trabajo de investigación que se describe en esta memoria, y que no es otra que facilitar la incorporación de técnicas de tolerancia a fallos a las plataformas de propósito general para así aumentar su garantía de funcionamiento.

Antecedentes del trabajo de investigación

El punto de partida de este trabajo de investigación es, precisamente, la finalización del proyecto ESPRIT P5212, FASST (*Fault-Tolerant Architecture with Stable Storage Technology*), que había comenzado en 1991 con la finalidad de diseñar, construir y evaluar un sistema tolerante a fallos de propósito general basado en el almacenamiento estable de la información [FASS90]. Durante el proyecto se propuso la implementación de un sistema multiprocesador tolerante a fallos basado en una memoria central estable [BMR+91]. La tarea a desarrollar por la Universidad de Valencia dentro del proyecto tenía dos objetivos básicos: colaborar con la empresa August Systems en el diseño e implementación del módulo procesador [MPF+95] y colaborar en la modificación del micro-núcleo de MACH para utilizarlo como base del sistema operativo de la plataforma FASST [JSM+93].

Al finalizar el proyecto, aunque no se habían podido culminar los objetivos iniciales del mismo, sí que se habían conseguido resultados satisfactorios, tanto a nivel industrial como académico. Dentro del trabajo desarrollado por la Universidad de Valencia, cabe resaltar la lectura de tres tesis doctorales [Fab96][Cer97][Mar97]. En [Cer97] se estudian algoritmos de comunicación tolerantes a fallos y se aplican al problema de la comunicación punto a punto, en [Mar97] se completa la especificación, la implementación y la evaluación del módulo de proceso iniciados durante el proyecto, y en [Fab96] se aborda el problema de los modelos de comportamiento de los sistemas multiprocesador y la ubicación en este ámbito de mecanismos de recuperación para obtener sistemas tolerantes a fallos.

Planteamiento

Teniendo en cuenta la experiencia adquirida en el proyecto FASST, para abordar el problema de las plataformas de propósito general tolerante a fallos,[2] se consideró oportuno enfocar la investigación hacia plataformas que perteneciesen al segmento representado por los multiprocesadores con memoria compartida basados en bus. Este enfoque sugiere un modelo de sistema con un número reducido de nodos (se puede considerar que actualmente 10 nodos es un límite adecuado para este tipo de plataformas), que abarca tanto a los multicomputadores como a los multiprocesadores con memoria compartida. Este enfoque establece una diferencia significativa respecto a los trabajos cuyo modelo de sistema son los sistemas distribuidos, ya que en este caso las técnicas y algoritmos utilizados deben resultar válidos para plataformas con un número mucho mayor de nodos. Además, en los sistemas distribuidos, la latencia y el ancho de banda del medio de comunicación pueden variar, incluso en varios órdenes de magnitud, en función de los nodos involucrados en la comunicación y de la carga del sistema.

Tras consultar la bibliografía sobre plataformas de propósito general tolerantes a fallos, se ha observado una tendencia a incluir las técnicas de tolerancia a fallos en los sistemas siguiendo una de las dos alternativas que se describen a continuación:

1. Diseñar un soporte físico especializado que incluya la mayor parte de las técnicas de tolerancia a fallos, con la finalidad de que el sistema operativo, apoyándose en estas técnicas, pueda ocultar a los procesos la gestión de los errores. El trabajo desarrollado durante el proyecto FASST, el que se describe en [Fab96] y el sistema comercial Integrity S2 de Tandem Computers, que se analiza en el apartado 1.3.1, son ejemplos de esta forma de abordar el problema.
2. Proporcionar a los procesos un entorno de ejecución que, como en el caso anterior, oculte a los procesos las actividades relacionadas con la tolerancia a fallos. La mayor parte de estos trabajos están orientados hacia los sistemas distribuidos y la mayoría de las actividades relativas a la tolerancia a fallos las realizan procesos y librerías ejecutándose sobre un sistema operativo estándar. Este es el caso, por ejemplo, del trabajo descrito en [HK93] (ver el apartado 1.3.5). En cambio, en Manetho (ver el apartado 1.3.3) se ha modificado el núcleo del sistema operativo para que ofrezca servicios específicos que hagan más eficiente la implementación de las técnicas de tolerancia a fallos que se necesitaba aplicar.

Ninguna de estas dos alternativas, sin embargo, contempla la especificación de los servicios que deben incluirse en el sistema operativo para desarrollar aplicaciones tolerantes a fallos. Resulta sorprendente para el autor que, pese a la larga tradición de la investigación en tolerancia a fallos —este año se celebrará la vigesimoctava edición del FTCS (*Fault-Tolerant Computer Symposium*)— no exista un estándar que establezca los servicios que deben ser incluidos en el sistema operativo para el desarrollo de aplicaciones tolerantes a fallos. Especialmente cuando esta especificación

[2] Así hemos denominado a las plataformas de propósito general que incluyen técnicas de tolerancia a fallos.

sí existe para otra área igualmente especializada como es la de las aplicaciones de tiempo real [IEEE92].

Objetivos

Por todo ello, la finalidad básica del trabajo de investigación que se describe en esta memoria es la de especificar una plataforma de propósito general tolerante a fallos, utilizando como modelo de sistema en el que enmarcar el trabajo un multiprocesador con un número reducido de nodos. Para abordar el problema, y teniendo en cuenta las consideraciones realizadas, se ha considerado adecuado marcarse dos objetivos más concretos, que son:

1. Proponer la arquitectura más adecuada para una plataforma de propósito general tolerante a fallos. Para ello es necesario estimar previamente la garantía de funcionamiento que es recomendable ofrecer en este tipo de plataformas, las técnicas de tolerancia a fallos que deben incluirse en el sistema para conseguirla, y los niveles de abstracción en los que deben ser incluidas estas medidas.
2. Especificar los servicios que debe ofrecer el sistema operativo para permitir el desarrollo de aplicaciones tolerantes a fallos en plataformas con la arquitectura escogida. Para especificar estos servicios es necesario analizar las necesidades de las aplicaciones tolerantes a fallos, y distribuir las medidas de tolerancia a fallos entre el sistema operativo y el resto de los niveles del sistema, de forma que se obtenga un compromiso entre el coste, las prestaciones y la garantía de funcionamiento del sistema.

Estructura de la memoria

A partir de estos objetivos básicos, se inició un trabajo de investigación cuyos resultados se describen en la presente memoria, que se ha estructurado en los siguientes capítulos:

- **Capítulo 1**. En él se analizan, con el objetivo de enfocar adecuadamente el trabajo de investigación, las técnicas de tolerancia a fallos, las formas de estructurar los sistemas tolerantes a fallos, y los sistemas más significativos que se han utilizado como referencia durante este trabajo.
- **Capítulo 2**. En él se propone la arquitectura para el sistema. La propuesta incluye la descripción de la arquitectura tanto para el soporte físico como para el soporte lógico, incluyendo el sistema operativo.
- **Capítulo 3**. Se describe la función a desempeñar por el micro-núcleo en aplicaciones tolerantes a fallos, identificándose las necesidades básicas para el desarrollo posterior de aplicaciones tolerantes a fallos.

- **Capítulo 4**. Contiene la especificación de los servicios del núcleo del sistema operativo, incluyendo la descripción de un nuevo tipo de servicios: las operaciones incrementales. Además, se especifica cómo debe interaccionar el núcleo con otros componentes lógicos para realizar tareas básicas relacionadas con la gestión de errores del sistema.
- **Capítulo 5**. En él se evalúan los servicios propuestos para el núcleo del sistema operativo en base a las necesidades identificadas en el capítulo 3.
- **Capítulo 6**. Se describe la implementación de un servidor de ficheros transaccional que utiliza los servicios del núcleo del sistema operativo, especificados en el capítulo 4, para tolerar fallos del sistema. El capítulo contiene, asimismo, los primeros resultados obtenidos tras la implementación de las operaciones incrementales y del servidor ficheros, que son analizados.
- **Capítulo 7**. En él se extraen las conclusiones y se muestran las líneas de trabajo futuro.

Capítulo 1

Tolerancia a fallos: situación actual

El objetivo del estudio que se describe en este capítulo es el de permitir enfocar adecuadamente el problema objeto de la investigación. Para ello se ha realizado una revisión de las diferentes técnicas actuales de tolerancia a fallos, considerando cómo se aplican tanto a nivel del soporte físico como a nivel del soporte lógico y resaltando las similitudes que existen entre ellas. Una vez consideradas las diferentes técnicas, se ha realizado un análisis sobre la forma de estructurar los sistemas tolerantes a fallos, haciendo hincapié en las ventajas e inconvenientes que supone incorporar estas técnicas a los diferentes niveles del sistema.

Para mostrar los resultados del estudio sobre la situación actual de los sistemas tolerantes a fallos, el capítulo se ha estructurado en tres partes bien definidas: la revisión de las técnicas de tolerancia a fallos actuales, que se recoge en el apartado 1.1; el análisis sobre la forma de estructurar los sistemas tolerantes a fallos del apartado 1.2; y la descripción de los sistemas que se han usado como referencia durante el diseño del sistema, que se recoge en el apartado 1.3.

1.1 Revisión de las técnicas de tolerancia a fallos

En este apartado se describen en primer lugar aquellas técnicas cuyo propósito específico es permitir la detección y el confinamiento de los errores. Posteriormente se aborda la revisión del resto de las técnicas de tolerancia a fallos, que es posible clasificar en dos grandes grupos: las técnicas basadas en recuperación hacia atrás y las técnicas basadas en redundancia modular. Cada uno de estos grupos se apoya en los mismos principios básicos, aunque la forma de aplicarlos es diferente en función de la técnica empleada y, especialmente, del nivel al que se aplique ésta. El estudio de cada uno de estos dos grupos sigue la misma estructura: comienza con la descripción de los principios básicos y después muestra las diferentes formas de aplicarlos en función de la técnica específica que se utilice y del nivel al que se aplique.

1.1.1 Detección y confinamiento de errores

No solamente los sistemas tolerantes a fallos incluyen técnicas para la detección y el confinamiento de errores; al contrario, cualquier sistema informático actual incluye numerosas medidas a diferentes niveles para detectar errores y limitar, dentro de lo posible el alcance de los mismos. A nivel del soporte físico, la memoria suele incluir redundancia en forma de códigos detectores y correctores de errores, así como los buses de interconexión, que, además, pueden incluir otras técnicas como el reintento automático de transacciones de bus o temporizadores que evitan que un componente averiado pueda detener completamente el sistema al no liberar el bus. Otros medios de interconexión, como las redes de área local, por ejemplo, también incorporan numerosas medidas, tanto en el soporte físico como en el lógico, para la detección de errores producidos durante la transmisión de los datos. Por su parte, los procesadores también incluyen medidas de detección y confinamiento de errores, que permiten la detección de numerosas situaciones excepcionales durante la ejecución de código. El mecanismo de traducción de direcciones, por ejemplo, comprueba las direcciones generadas por los procesos antes de que se lleve a cabo la operación de lectura o escritura en memoria, proporcionando un mecanismo efectivo para el confinamiento de errores en los procesos.

Existen numerosas clasificaciones de las técnicas de detección de errores en la bibliografía consultada. De entre ellas, cabe destacar la distinción que se hace en [LA90] entre las técnicas que permiten a un subsistema detectar sus propios errores y que denomina medidas (*measures*), y las que permiten detectar los de otros subsistemas a través de la interfaz, denominadas mecanismos (*mechanisms*).

Las medidas son de índole muy diversa, aunque, desde el punto de vista de las técnicas de recuperación, se pueden distinguir dos grupos. El primero comprende las técnicas basadas en la comparación de los resultados producidos por dos o más componentes replicados, y que será discutida en el apartado 1.1.3. El resto de las técnicas constituyen un conjunto de comprobaciones que serán utilizadas por las técnicas de recuperación hacia atrás, tal y como se describe en el apartado 1.1.2. En él se encuentran comprobaciones de temporización (*timing checks*), que permiten detectar componentes que no completan una petición de servicio en un tiempo

determinado, establecido por su diseño y su implementación. Asimismo, se encuentran diversas técnicas para detectar errores en los datos, basadas en la introducción de redundancia en la representación de los objetos y estructuras de datos del componente, o técnicas que pueden ser aplicadas a los resultados generados por el componente, como las comprobaciones de razonabilidad (*reasonableness checks*) o las comprobaciones inversas (*reversal checks*).

Los mecanismos, tal y como se describe en el anexo I, detectan errores en las peticiones de servicio recibidas por un componente, lo cuál supone un funcionamiento incorrecto del sistema que lo solicita. Las técnicas incorporadas en los procesadores actuales, que han sido nombradas al inicio de este apartado, serían un ejemplo a nivel del soporte físico de estos mecanismos. A nivel del soporte lógico es habitual, por ejemplo, la comprobación intensiva de las peticiones de servicio que recibe el núcleo del sistema operativo, con la finalidad de detectar errores de diseño en los procesos y evitar que puedan afectar al resto del sistema.

1.1.2 Recuperación hacia atrás

Los mecanismos de recuperación hacia atrás devuelven el sistema a un estado anterior libre de errores. Por lo tanto, es necesario que el sistema pueda distinguir entre estados libres de errores y estados erróneos. Se considera como estado libre de errores "cualquier estado al que el sistema haya llegado sin haber detectado ningún error" [Fab96, p.22]. La capacidad del sistema para identificar los estados libres de errores dependerá de la cobertura de los mecanismos de detección utilizados; pero, tal y como se recoge en la cita anterior, "es el único criterio que se puede aplicar". Debido a que estos mecanismos suponen la aceptación o rechazo del estado actual del sistema como un estado libre de errores, se les ha denominado **tests de aceptación** [LA94].

Existen dos formas de almacenar la información necesaria para la vuelta atrás del sistema [LA90, p. 151]: la primera consiste en almacenar una copia completa del estado del sistema, en cuyo caso el punto de recuperación se denomina *checkpoint*; [1] en cambio, en la segunda, se mantiene un registro (*log* o *audit trail*) de las operaciones que va realizando el sistema.

La vuelta atrás en el primer caso resulta sencilla, simplemente se desecha el estado actual del sistema y se reemplaza por el estado almacenado en el punto de recuperación. (Ver Ilustración 1-1). De esta forma la recuperación no depende en absoluto del estado erróneo del sistema; sin embargo, establecer un punto de recuperación es una operación costosa, ya que es necesario garantizar que existe una copia del estado completo del sistema.

[1] En este trabajo se utilizará punto de recuperación como traducción de *recovery point* y de *checkpoint*, indicándose el término original en inglés cuando sea necesario distinguir entre ambos.

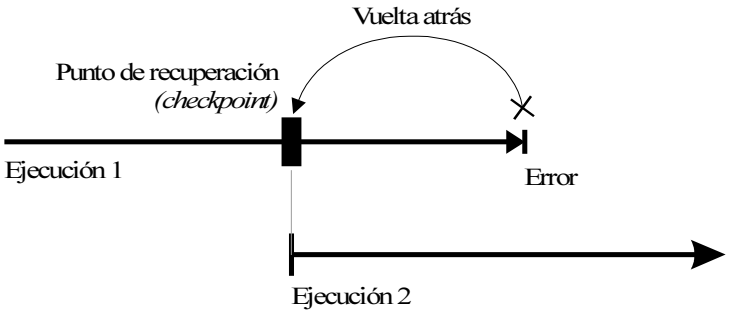

Ilustración 1-1: Recuperación hacia atrás utilizando un checkpoint.

Si se utiliza un registro, la vuelta atrás se consigue procesando el registro en orden inverso, deshaciendo los cambios producidos por cada operación. (Ver Ilustración 1-2). En este caso la recuperación depende del estado erróneo y es necesario que los daños producidos por el error estén confinados en los componentes cuyo estado puede ser recuperado a partir del registro. El registro permite una recuperación más rápida cuando su longitud es pequeña; mientras que la recuperación utilizando un punto de recuperación (*checkpoint*) presenta un coste constante, que llegará a ser menor que el necesario para procesar el registro cuando la longitud de éste sea lo suficientemente grande.

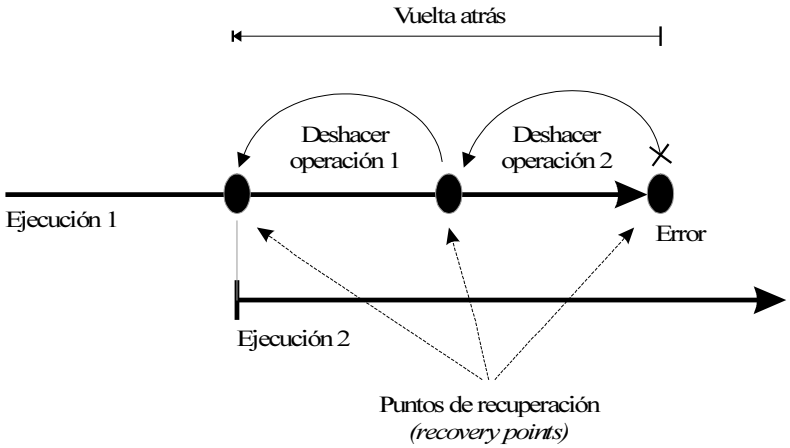

Ilustración 1-2: Recuperación hacia atrás utilizando un registro.

Es posible combinar ambas técnicas estableciendo puntos de recuperación (*checkpoints*) a intervalos relativamente largos y manteniendo simultáneamente un registro de operaciones. En este caso, la recuperación puede realizarse recuperando el estado almacenado en un *checkpoint* anterior al punto de recuperación al que se desea volver y procesando en orden directo las entradas necesarias del registro, tal y como se

muestra en la Ilustración 1-3.[2] Alternativamente, es posible recuperar el estado de un *checkpoint* posterior al punto de recuperación al que se desea volver y procesar en orden inverso las entradas necesarias del registro.

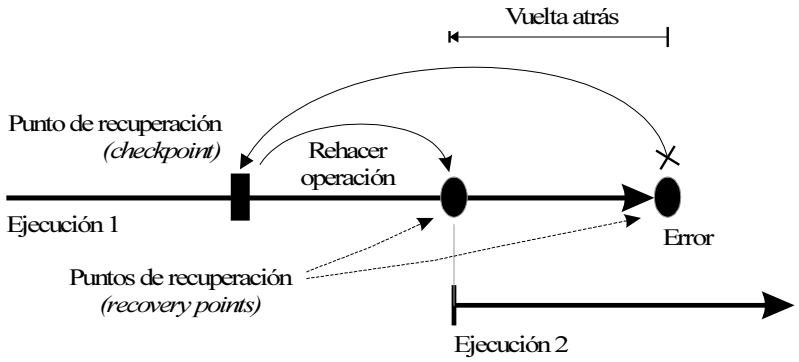

Ilustración 1-3: Recuperación hacia atrás utilizando checkpoints *y registros.*

Habitualmente, los sistemas están formados por varios componentes que intercambian información entre sí y progresan en paralelo (procesadores a nivel físico o procesos a nivel lógico). En general, cada uno de los componentes puede establecer un punto de recuperación[3] que contenga su estado, y puede hacerlo independientemente del resto de los componentes. En estos sistemas, los protocolos para el establecimiento de puntos de recuperación y la recuperación del sistema deben considerar el flujo de información entre componentes para garantizar que el estado tras la vuelta atrás sea uno de los estados alcanzables por el sistema en ausencia de errores, aunque no haya ocurrido previamente. La Ilustración 1-4 muestra un caso en el que el estado final no es correcto, ya que el componente C_2 se encuentra en un estado al que sólo podría haber llegado si hubiese recibido la información de C_1; pero éste ha vuelto atrás, a un estado anterior al envío de dicha información. A nivel del soporte físico, los multiprocesadores con memoria compartida basados en bus constituyen un buen ejemplo; los componentes serían procesadores y el flujo de información de C_1 a C_2 se produciría al escribir el primer procesador en una dirección de memoria compartida que posteriormente es leída por el segundo. Del mismo modo, a nivel del soporte lógico, los componentes serían procesos que podrían intercambiar información utilizando los servicios del sistema operativo.

[2] A partir del punto de recuperación (*checkpoint*), se utiliza el contenido del registro para llevar el sistema al *recovery point*. La ejecución del sistema se reanuda, sin embargo, en el *recovery point*, tal y como se describe en el apartado 1.1.2.2.

[3] De ahora en adelante, a menos que se indique lo contrario, punto de recuperación equivaldrá a *checkpoint*.

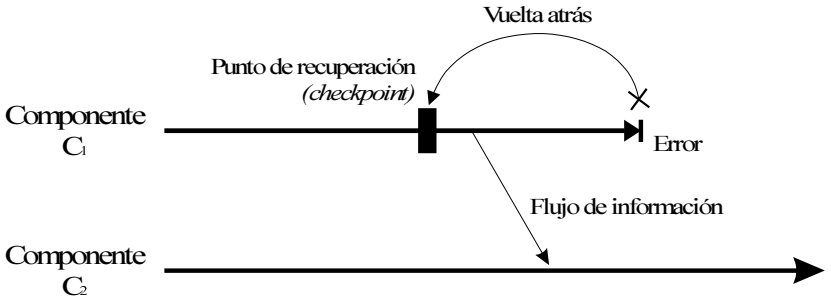

Ilustración 1-4: Ejemplo de sistema que pasa a un estado inconsistente tras la vuelta atrás de uno de sus componentes.

1.1.2.1 Recuperación hacia atrás en el soporte físico

La solución a este problema depende del nivel al que se trabaje. A nivel del soporte físico la información que debe ser copiada para establecer un punto de recuperación consiste, típicamente, en el contenido de los registros del procesador[4] y las modificaciones realizadas sobre una copia privada de la memoria del sistema (generalmente mantenida en una memoria cache en el módulo de proceso). Las soluciones propuestas pueden ser clasificadas en tres clases [Fab96, pp. 110-114]. La primera, que es una extensión directa del caso de un monoprocesador, consiste en el establecimiento de puntos de recuperación y vueltas atrás globales, participando siempre todos los procesadores del sistema [AFM90].

La segunda trata de reducir el número de procesadores que deben participar en el establecimiento de puntos de recuperación y en la eventual vuelta atrás anotando ciertas interacciones entre procesadores que han recibido el nombre de dependencias.[5] Cuando un procesador establece un punto de recuperación también lo hacen todos los procesadores de los que éste depende. Y cuando un procesador realiza una vuelta atrás también lo hacen todos los procesadores que dependen de él. [MF92][Ors93][Fab96]

En cambio, en la tercera se deshacen las dependencias haciendo que los procesadores establezcan puntos de recuperación tras ciertas operaciones de comunicación entre sí. De esta forma, la vuelta atrás de un procesador no requiere la vuelta atrás de ningún otro. [AFM90][JF93][Fab96]

La interacción de estos sistemas con el mundo exterior suele requerir el establecimiento de puntos de recuperación adicionales para garantizar la consistencia del sistema, evitándose la vuelta atrás a un estado anterior a la interacción.

---

[4] En [BJ90] y [BMR+91] se describe un enfoque diferente, en el que se consideran puntos de recuperación por proceso, en lugar de por procesador.

[5] En [Fab96, p. 104] aparece la siguiente definición de dependencia, que se considera acertada: "Entre dos procesadores que interaccionan se establece una dependencia cuando cualquier acción del mecanismo de recuperación (establecimiento de un punto recuperación, o realización de una vuelta atrás) efectuada por uno de ellos obliga a realizar una acción del mismo tipo en el otro."

1.1.2.2 Recuperación hacia atrás en el soporte lógico

A nivel del soporte lógico, las soluciones generales al problema de garantizar que el estado final de una vuelta atrás es consistente también pueden ser clasificadas en tres grupos [DVC+93]: puntos de recuperación coordinados, registro de mensajes y técnicas híbridas.

Puntos de recuperación coordinados

El las del primer grupo, todos los procesos, o un conjunto de ellos que ha interaccionado entre sí, se coordinan para guardar su estado conjuntamente. Estas técnicas permiten un proceso de vuelta atrás sencillo; pero requieren que los procesos permanezcan bloqueados mientras se establece el punto de recuperación, afectando a las prestaciones del sistema. Finalmente, la comunicación con el mundo exterior exige el establecimiento de un punto de recuperación para evitar inconsistencias.

Registro de mensajes

En el segundo grupo se encuentran las técnicas que mantienen un registro de los mensajes intercambiados entre los procesos. En este caso, los procesos guardan su estado de forma independiente y, en caso de vuelta atrás, se utiliza el contenido del registro de mensajes para llevar al proceso a un estado consistente con el resto del sistema, tal y como se muestra en la Ilustración 1-3. Por lo tanto, los procesos deben ser deterministas, es decir, a partir del mismo estado y con la misma entrada, deben producir la misma salida. Puesto que los procesos guardan su estado de forma independiente, la carga sobre el medio de comunicación es menor; aunque se produce una sobrecarga continua (*continuous overhead*) debido al mantenimiento del registro. Este mantenimiento puede ser pesimista u optimista. [DVC+93, p. 29][Eln93, p. 90][Rus94, p. 102]

- **Pesimista**. Los mensajes se guardan síncronamente en el registro, es decir, el proceso que envía un mensaje permanece bloqueado hasta que éste ha sido almacenado en el registro. Las ventajas de este tipo de estrategias son: solamente es necesario mantener un punto de recuperación por proceso, solamente los procesos afectados por errores deben volver atrás y, con respecto a la comunicación con el mundo exterior, un proceso puede enviar y recibir mensajes sin necesidad de coordinarse con otro procesos. El inconveniente principal es el tiempo que los procesos permanecen bloqueados como consecuencia de las escrituras síncronas en el registro.
- **Optimista**. Los procesos continúan con su ejecución mientras se guardan los mensajes en el registro, asumiendo (de ahí el nombre de optimista) que todos los mensajes habrán sido guardados antes de que se produzca un error. La gran ventaja de estas técnicas es la pequeña sobrecarga que añaden al sistema cuando no se producen errores, ya que no se necesita acceso síncrono al registro. Sin embargo, existen varios inconvenientes: es necesario mantener varios puntos de recuperación por proceso, los fallos no están confinados en los procesos que fallan, es necesario mantener más cantidad de información en el registro para

saber qué cálculos dependen de cada mensaje y qué mensajes han sido efectivamente guardados en el registro.

Técnicas híbridas

El tercer grupo comprende un conjunto de técnicas que están basadas en puntos de recuperación coordinados; pero que evitan el bloqueo de los procesos manteniendo un registro de mensajes. Estas técnicas combinan las ventajas de los grupos anteriores, aunque presentan el inconveniente de utilizar algoritmos más complejos.

1.1.2.3 Transacciones

Una transacción está formada por un conjunto de operaciones que deben ser ejecutadas respetando cuatro propiedades básicas: atomicidad, consistencia, aislamiento y durabilidad [Tan95][TV91]. A diferencia de las técnicas anteriores, en este caso sí se realizan suposiciones sobre la forma de organizar la ejecución de las actividades, ya que éstas deben estar organizadas en transacciones. A continuación se describen las cuatro propiedades básicas:

- **Atomicidad** (*Atomicity*). Para el mundo exterior, la transacción se lleva a cabo de forma indivisible, es decir, o bien se llevan a cabo todas las operaciones que la forman, o bien no se lleva a cabo ninguna.
- **Consistencia** (*Consistency*). La transacción es una transformación que conserva la integridad de los datos del sistema.
- **Aislamiento** (*Isolation*). Las transacciones no interfieren entre sí. Aunque pueden ejecutarse concurrentemente, el resultado final debe ser como si todas las transacciones concurrentes se hubieran ejecutado secuencialmente en algún orden establecido por el sistema.
- **Durabilidad** (*Durability*). Una vez ha finalizada la transacción, los cambios realizados por ella son permanentes.

Una transacción comienza con la operación *BEGIN*, contiene un conjunto de operaciones que transforman los datos del sistema manteniendo su integridad (propiedad de consistencia), y finaliza con la operación *END* (o *COMMIT*), que convierte en permanentes los efectos de la transacción (propiedad de durabilidad). Alternativamente, los efectos de la transacción pueden ser desechados utilizando la operación *ABORT*.

La implementación de las transacciones puede llevarse a cabo de dos formas muy diferentes [Tan95, pp. 150-153]. La primera consiste en asignar a cada transacción una copia privada de los datos sobre los que trabaja. Si la transacción termina con éxito, los cambios se transfieren a la copia original. En cambio, si aborta, simplemente se desecha la copia con los cambios. En la segunda, las modificaciones se realizan sobre los datos originales. Para garantizar las propiedades de las transacciones se mantiene un registro al que se añade una entrada por cada modificación antes de realizarla. Cuando la transacción finaliza con éxito, se añade una nueva entrada al registro que lo refleje; en cambio, si aborta, es necesario utilizar la información del registro para deshacer los cambios realizados hasta ese momento.

Los mecanismos para el control de concurrencia pueden clasificarse en pesimistas y optimistas. Los algoritmos pesimistas asumen que los conflictos entre transacciones son bastante frecuentes y comprueban la validez de cada operación antes de que se éste se produzca. En cambio, los algoritmos optimistas permiten que cada transacción realice las operaciones sin trabas. Solamente al final, antes de convertir los cambios en permanentes, comprueban que dichos cambios no interfieran con el trabajo realizado por otras transacciones. Si interfieren, la transacción debe comenzar de nuevo desde el principio, con el consiguiente coste adicional; pero, si los conflictos son poco frecuentes, el rendimiento del sistema es mayor que con los algoritmos pesimistas, ya que los optimistas permiten mayor paralelismo en la ejecución de transacciones.

1.1.2.4 Bloques de recuperación

Los bloques de recuperación proporcionan una forma de estructurar los programas para tolerar fallos en el soporte lógico. Cada bloque de recuperación está formado por un conjunto de módulos lógicos ordenados y un test de aceptación del resultado que producen. La Ilustración 1-5 muestra el flujo de control en un bloque de recuperación. En primer lugar, se establece un punto de recuperación. Después se ejecuta el módulo principal —el primero de los módulos ordenados. Entonces se comprueban los resultados con el test de aceptación. Si son correctos, se sale el bloque; si no lo son, se vuelve al punto de recuperación y se ejecuta el segundo módulo. El proceso se repite hasta que un módulo produce resultados aceptables para el test o hasta agotar todos los módulos, en cuyo caso el bloque se avería.

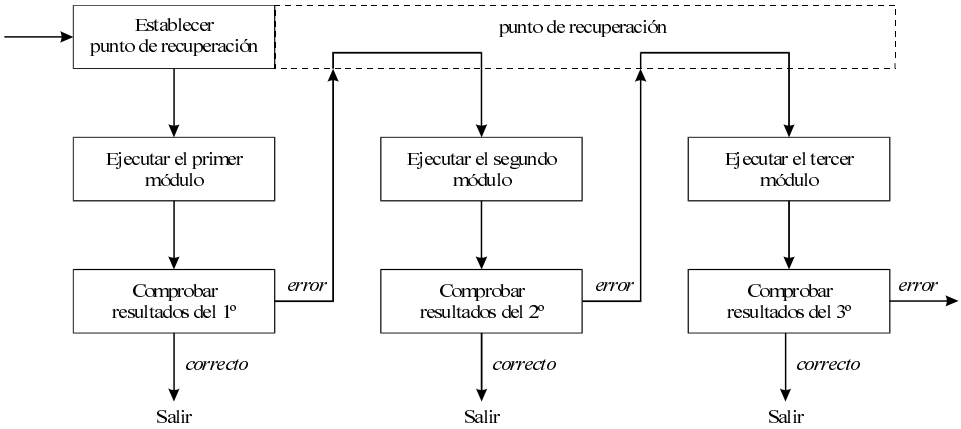

Ilustración 1-5: Bloque de recuperación con tres módulos lógicos.[6]

Los bloques de recuperación pueden anidarse, dando lugar a una jerarquía en la que el bloque de recuperación de nivel más alto es el sistema completo. Así, una avería en

[6] Esta ilustración está basada en la ilustración 11.1 de [LA94, p. 255].

un bloque de recuperación inferior puede ser gestionada por el bloque de recuperación superior como un error del módulo correspondiente.

1.1.3 Redundancia modular

La **redundancia modular** consiste en implementar las funciones de un componente C (módulo, o subsistema) utilizando un componente C' que incluya varias instancias de C, junto con un componente X encargado de procesar las salidas de dichas instancias y generar la salida de C', tal y como se muestra en la Ilustración 1-6.

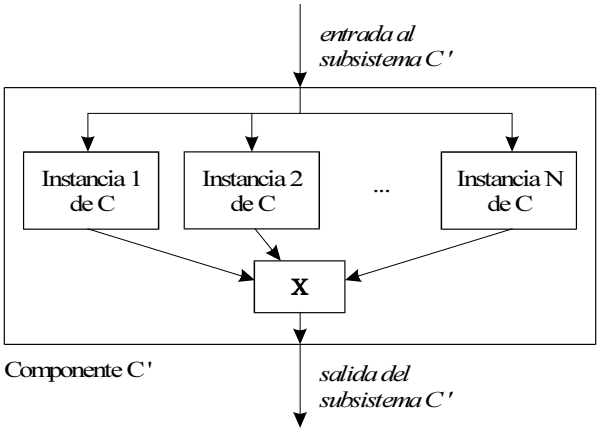

Ilustración 1-6: Redundancia modular.

Dependiendo del objetivo con que se diseñe el sistema modular redundante, el componente X realiza unas tareas u otras. Por ejemplo, una configuración triple modular redundante (TMR) contiene tres instancias del componente C, y el componente X se encarga de realizar una votación de las diferentes salidas. El sistema genera la salida más votada, evitando que un fallo en cualquiera de las instancias de C averíe el sistema. En cambio, una configuración más sencilla, con dos instancias del componente C y un comparador como componente X puede detectar un fallo de alguna de las instancias de C. Para conseguir un comportamiento de parada tras la avería, X debería aislar al sistema, evitando cualquier tipo de salida, que podría ser errónea. Evidentemente, el componente X es la parte más débil del diseño, y debe ofrecer una garantía de funcionamiento adecuada.

Además, para poder ofrecer una garantía de funcionamiento del sistema modular elevada, es necesario evitar la posibilidad de que un mismo fallo afecte a la mayoría de las instancias de C. Para lograrlo, se emplea una técnica conocida como **diversificación funcional** (*design diversity*), cuya definición aparece en [LAB+95]: "producción de dos o más sistemas que ofrezcan el mismo servicio a través de diseños

y realizaciones diferentes".[7] Esta técnica puede aplicarse tanto al soporte físico como al soporte lógico, y permite disminuir la probabilidad de que existan fallos comunes a las diferentes instancias de C, aumentando la garantía de funcionamiento.

1.1.3.1 Redundancia modular en el soporte físico

Las técnicas de redundancia modular en el soporte físico han sido clasificadas [Joh89] en tres grupos: redundancia pasiva, redundancia activa y redundancia híbrida.

Los sistemas con **redundancia pasiva** no detectan los fallos ni llevan a cabo acciones específicas para tratarlos. En lugar de ser detectados, los fallos son enmascarados, impidiendo que los errores se propaguen fuera del sistema, que siempre produce salidas correctas. La configuración triple modular redundante, discutida anteriormente, es la que se implementa con mayor frecuencia, debido al elevado coste que supone la replicación de componentes físicos.

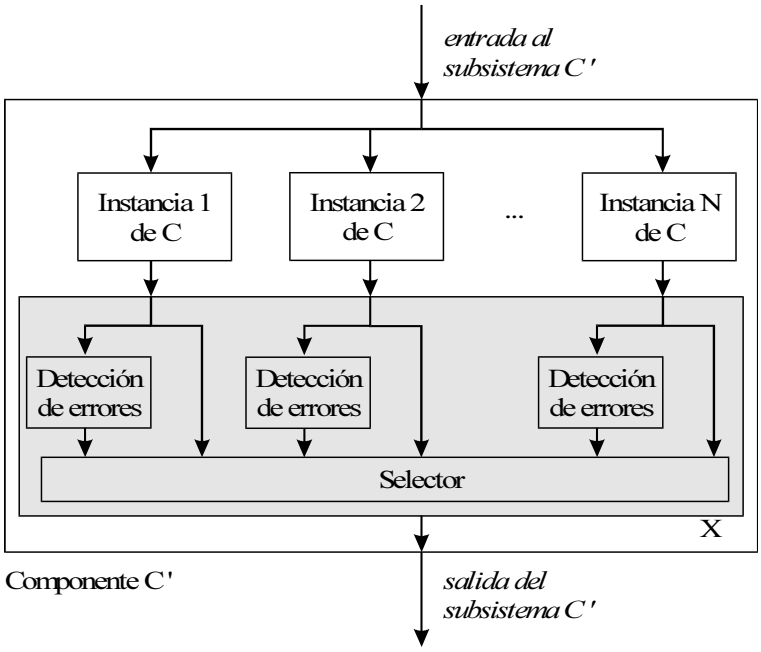

*Ilustración 1-7: Ejemplo de redundancia activa, repuesto en espera (*standby sparing*).*[8]

La **redundancia activa** (o dinámica) consiste en la inclusión en el sistema modular de técnicas para la detección de fallos y la posterior reconfiguración del sistema, evitando que éste se averíe. Un ejemplo de esta técnica se muestra en la Ilustración

[7] Del inglés: "***Design diversity*** *may be defined as the* ***production of two or more systems aimed at delivering the same service through separate designs and realizations***."

[8] Esta ilustración está basada en la ilustración 3.14 de [Joh89, p. 66].

1-7. El componente X se encarga de seleccionar como salida del sistema la de una de las instancias de C libre de errores. Las técnicas de detección de errores incluidas en el componente X permiten distinguir entre las instancias averiadas y las que están libres de errores. En este caso, la reconfiguración del sistema puede suponer la desactivación de las instancias defectuosas o, simplemente, que sean desestimadas por el selector a la hora de elegir la instancia que proporciona la salida del sistema. La posibilidad de sustituir las instancias averiadas para evitar la pérdida progresiva de la garantía de funcionamiento dependerá del tipo de componente que sea C. Por ejemplo, los circuitos integrados no pueden ser sustituidos sin detener el sistema; mientras que las tarjetas conectadas a un bus que admita la retirada e inserción de módulos con el sistema en funcionamiento, como Futurebus+ [Fut91], sí podrían ser reemplazadas.

Finalmente, la **redundancia híbrida** intenta combinar las ventajas de las técnicas anteriores, evitando sus problemas. Para ello, incluye el enmascaramiento de fallos de los sistemas pasivos y añade las técnicas de detección y tratamiento de fallos de los sistemas pasivos. De esta forma se evita la degradación en la garantía de funcionamiento que se produce en los sistemas con redundancia pasiva a medida que se van averiando componentes. Por otro lado, también se evita el tiempo necesario para tratar el fallo, típico de los sistemas activos, manteniendo las prestaciones de los sistemas pasivos. El inconveniente es el coste extra que suponen los componentes adicionales para la implementación de los sistemas híbridos.

1.1.3.2 Redundancia modular en el soporte lógico

Al implementar un componente lógico utilizando redundancia modular, es necesario garantizar que los errores causados por un fallo en cualquiera de las instancias del componente C no afectan a las demás. La forma más usual de conseguirlo es la que se propone en [LA94, p. 273], que consiste en implementar cada instancia de C como un proceso separado y hacer explícitas las comunicaciones entre ellos (usualmente mediante paso de mensajes), evitando el uso de variables compartidas.

Por otra parte, el soporte lógico presenta la deseable propiedad de no degradarse. Por lo tanto, los fallos en los componentes lógicos son permanentes, ya que son fallos de diseño que no se modifican a lo largo del tiempo. Por todo ello, en general, si todas las instancias de C compartiesen el mismo diseño e implementación, la redundancia modular no permitiría tolerar fallos del soporte lógico, puesto que todas las instancias se verían afectadas por ellos. Por esta razón, la programación de N versiones y la programación autocomprobante utilizan la diversificación funcional al nivel del soporte lógico, con el objetivo de tolerar fallos de diseño en el mismo. [9]

No obstante, debido al éxito de las técnicas modernas de ingeniería del *software*, que permiten reducir enormemente el número de fallos que contiene el soporte lógico de los computadores, la mayor parte de los fallos residuales tras el proceso de desarrollo y depuración del soporte lógico son sutiles y suelen manifestarse en condiciones poco usuales de interacción entre los diferentes niveles lógicos del sistema

[9] La diversificación funcional no es una técnica exclusiva de los sistemas con redundancia modular. Los bloques de recuperación, descritos en el apartado 1.1.2.4 (pág. 15), también la utilizan.

(aplicaciones, utilidades y sistema operativo) o entre los diferentes procesos que componen las aplicaciones complejas [Bor94] [CDD90] [Gra86[10]]. Ello hace que la redundancia modular pueda ser una técnica efectiva para tratar este tipo de fallos, puesto que la probabilidad de que se manifiesten simultáneamente en varias instancias de C es suficientemente pequeña.

Réplicas pasivas

Cuando no se utiliza diversidad funcional, todas las instancias del componente comparten el mismo diseño e implementación, por lo que a las instancias se les denomina réplicas. Al hablar de réplicas pasivas, se hace referencia a un componente modular redundante en el que sólo una de las réplicas (instancias) procesa los datos de entrada (permanece activa), mientras que el resto constituyen elementos de repuesto que pueden activarse en caso de necesidad. En este caso, el componente X debe detectar la avería de la réplica activa y provocar la activación de otra de las réplicas. Para evitar que el trabajo realizado por el componente (por la réplica activa del componente, en concreto) se pierda, es necesario que actualice periódicamente el estado de las réplicas pasivas. Por lo tanto, esta técnica, aunque tiene un origen diferente, es realmente una técnica de recuperación hacia atrás, en la que el punto de recuperación se almacena en las réplicas pasivas y, en caso de fallo, la vuelta atrás no se produce en el mismo proceso, sino mediante la activación de otra de las réplicas. Por lo tanto, para tolerar N fallos, son necesarias N+1 réplicas (1 activa y N pasivas).

Al igual que ocurría con los puntos de recuperación, la comunicación con el mundo exterior de un componente que utilice replicación pasiva provoca problemas de consistencia. En general, será necesario establecer puntos de recuperación (actualizar el estado de las réplicas pasivas) siempre que se produzca la comunicación con el mundo exterior, para deshacer las dependencias.

En cambio, si las réplicas son deterministas,[11] es posible reducir la frecuencia de los puntos de recuperación manteniendo un registro de los mensajes que envía y recibe la réplica principal. En caso de avería de la réplica activa, la réplica pasiva puede utilizar su estado y este registro para alcanzar un estado que sea consistente con el mundo exterior y, a partir de él, continuar como réplica activa. En este caso, es necesario sopesar el coste extra durante la ejecución libre de errores que supone la actualización del estado de las réplicas pasivas frente al tiempo necesario para la recuperación en caso de fallo, puesto que, cuanto mayor sea la longitud del registro, mayor será el tiempo necesario para activar una réplica pasiva.

[10] Jim Gray, en este artículo, define los Heisenbugs como aquéllos que se manifiestan en circunstancias tan excepcionales que las técnicas para detectar su existencia pueden perturbar la situación del sistema lo suficiente como para que desaparezcan, a semejanza del Principio de Incertidumbre de Heisemberg en Física.

[11] La siguiente definición se ha tomado de [CPR+92, p. 29]: "Se dice que una réplica (de un componente lógico dado) es determinista si, en ausencia de fallos, cualquier ejecución de la réplica que parta del mismo estado inicial y consuma el mismo conjunto ordenado de mensajes produce el mismo conjunto ordenado de mensajes." Es decir, produce los mismos mensajes y en el mismo orden.

Réplicas activas

En este caso, todas las réplicas que forman parte del componente son réplicas activas. Ahora, en caso de avería de cualquiera de ellas, no es necesaria una vuelta atrás, sino que el componente modular redundante continúa produciendo salidas correctas sin interrumpir el servicio.

La función del componente X depende del comportamiento de las instancias de C tras la avería. Si garantizan el silencio tras la avería, entonces X puede seleccionar como salida del sistema C' la de cualquiera de las réplicas que no se hayan averiado. En este caso, como ocurría con las réplicas pasivas, es posible tolerar N fallos con N+1 réplicas.

En cambio, si las réplicas pueden producir resultados erróneos, es necesario que el componente X realice una votación con los resultados. El resultado mayoritario es elegido como salida de C', y los resultados discrepantes indican averías de las instancias que los han producido. En esta situación, en cambio, son necesarias 2N+1 réplicas para tolerar N fallos [CDD90, p.8][CPR+92, p.32], ya que es necesario garantizar que la mayoría de las réplicas producirá resultados correctos.

En ambos casos, tanto para garantizar que el componente X puede llevar a cabo la votación, como para garantizar que la salida de C' no dependerá de la réplica cuya salida haya sido seleccionada por X, las réplicas deben ser deterministas.

Programación de N versiones

Al contrario que ocurría con la replicación de procesos, la programación de N versiones utiliza el principio de diversidad funcional para permitir que el componente tolere fallos en el soporte lógico. Como en el caso anterior, todas las instancias del componente modular redundante son activas; pero, en este caso, cada una de ellas tiene un diseño y una implementación lo más independientes posible.

En este caso, el componente X realiza una votación de los resultados para generar la salida del componente modular redundante. En [LA90, pp. 227-230], se destacan dos aspectos de la votación que, aunque serían aplicables también al caso de las réplicas activas, tienen en este caso una mayor importancia. El primero es la sincronización entre instancias para realizar la votación, ya que las instancias ejecutan algoritmos diferentes y esto hace que puedan producirse diferencias mayores en los tiempos de ejecución. Y el segundo es la necesidad de algoritmos de votación que tengan en cuenta resultados inexactos (es decir, resultados parecidos e igualmente válidos, aunque no exactamente iguales).

Programación de N versiones autocomprobantes

La programación autocomprobante consiste en añadir al programa suficiente redundancia como para que pueda comprobar su correcto funcionamiento en tiempo de ejecución. El resultado son programas que, o bien producen resultados correctos, o bien no producen ninguno (garantía de parada tras la avería).

La programación de N versiones autocomprobantes sigue el mismo principio de diversidad funcional que la programación de N versiones; pero, en este caso, cada una

de las instancias de C es un componente autocomprobante. Por lo tanto, el componente X puede generar la salida del sistema modular redundante utilizando la salida de cualquiera de las instancias de C. En caso de avería de la instancia elegida, el componente X simplemente tiene que seleccionar una nueva instancia, tal y como ocurría en el caso de las réplicas activas con comportamiento de parada tras la avería. A diferencia de aquél, en este caso cada instancia de C ha sido diseñada e implementada de forma diferente.

Esta técnica presenta una gran similitud con el caso de réplicas activas con comportamiento de parada tras la avería, aunque en este caso cada instancia de C habrá sido diseñada e implementada de forma diferente.

En [LAB+95, p. 105] se identifica esta técnica, cuyo nombre en inglés es *N self-checking programming* (NSCP), y se compara con la programación de N versiones (NVP) y los bloques de recuperación (RB). A diferencia de NVP, donde la decisión de si un resultado era aceptable o no dependía de los resultados de las demás instancias de C, en NSCP esta decisión es responsabilidad de cada una de estas instancias. Por otra parte, RB es una técnica de recuperación hacia atrás, aunque la ejecución en paralelo de las diferentes variantes del bloque de recuperación podría dar lugar a un esquema bastante similar a NSCP.

1.2 Análisis de la estructura de los sistemas tolerantes a fallos

La revisión de las técnicas de tolerancia a fallos ha puesto de manifiesto que existen numerosas similitudes entre ellas y que, en algunos casos, una misma técnica puede ser aplicada tanto en el soporte físico como en el soporte lógico. A continuación se estudia cómo se suelen estructurar los sistemas tolerantes a fallos, analizando las ventajas e inconvenientes de cada forma de enfocar el problema. El apartado comienza con la descripción de una estructura que es suficientemente detallada como para poder expresar los resultados de este análisis, y suficientemente general como para englobar a los sistemas estudiados. El resto del apartado contiene la descripción del análisis, que finaliza con un resumen de sus resultados.

1.2.1 Descripción de la estructura

La división natural entre soporte físico y soporte lógico proporciona una división del sistema en dos capas, y la gran diversidad de sistemas informáticos actuales hace difícil identificar una estructura común para todos ellos que permita ir más allá de esta estructura en dos capas. No obstante, con objeto de analizar los sistemas tolerantes a fallos desde el punto de vista de su estructura, se considerará que el soporte lógico de los sistemas informáticos está constituido a su vez por una estructura jerárquica de tres

niveles: sistema operativo, utilidades y aplicaciones;[12] tal y como se muestra en la Ilustración 1-8. Siguiendo la metodología de diseño descrita en el anexo I, cada uno de los niveles se corresponde con un componente (o subsistema) y el conjunto de todos ellos constituye el sistema informático.

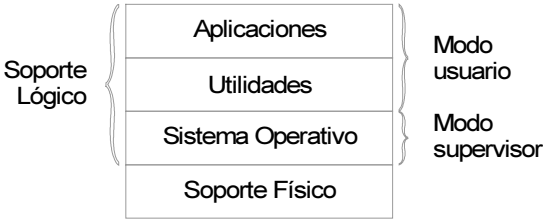

Ilustración 1-8: Niveles del sistema.

- Las **aplicaciones** son los subsistemas encargados de resolver problemas concretos. Por ejemplo, un procesador de textos es una aplicación que permite la edición de textos, ofreciendo una interfaz bien definida al usuario final del sistema. Así mismo, también es una aplicación el sistema encargado de manejar un conjunto de sensores y actuadores para el control automático de un proceso industrial o un sistema de gestión de bases de datos.
- El **sistema operativo** es el encargado de permitir y controlar la ejecución de las aplicaciones, y de actuar de interfaz entre éstas y el soporte físico del computador, teniendo como finalidades básicas facilitar el desarrollo de aplicaciones y aumentar la eficiencia del sistema. Durante la presente exposición se considerará que éste nivel se corresponde con el núcleo del sistema operativo, considerando como utilidades todos los servicios que sean implementados por código ejecutándose en modo usuario (por ejemplo, un servidor de ficheros en un micro-núcleo).
- Aunque las aplicaciones pueden utilizar directamente la interfaz del sistema operativo, la mayor parte de las herramientas de desarrollo y entornos de ejecución de aplicaciones ofrecen una interfaz más elaborada que facilita aún más su desarrollo. Dependiendo de la interfaz del sistema operativo y de la herramienta de desarrollo y el entorno de ejecución elegidos el nivel de **utilidades** será más o menos complejo. Por ejemplo, en el caso de un compilador C sobre Unix este nivel es bastante reducido, incluyendo librerías de funciones para la gestión de memoria dinámica o de E/S. En cambio, en un sistema operativo como Mach el nivel de utilidades incluye, además de las librerías, un conjunto de procesos como servidores de ficheros o manejadores de dispositivos de E/S.

[12] Esta división en cuatro niveles coincide con la de [Sta97, pp. 48-49]. En otras referencias bibliográficas [HK93][CDD90] se consideran únicamente tres niveles: soporte físico, sistema operativo y aplicaciones. No obstante, la distinción entre sistema operativo y utilidades resulta necesaria para la presente exposición.

1.2.2 Nivel de enmascaramiento de fallos

Tal y como se ha visto, el objetivo de las técnicas de tolerancia a fallos es disminuir la tasa de averías de los sistemas en los que se incluyen. Los objetivos del sistema establecen la importancia relativa de la garantía de funcionamiento frente al rendimiento (*performance*) y el coste, permitiendo que se tomen decisiones adecuadas sobre los fallos que deben ser tolerados por el sistema para que éste ofrezca la garantía de funcionamiento necesaria.[13] Con la intención de simplificar el análisis siguiente, se considerará a partir de ahora que un sistema tolerante a fallos es aquél que presenta una interfaz totalmente libre de averías; lo cuál quiere decir que ningún fallo considerado durante la fase de diseño causa la interrupción de los servicios ofrecidos por el sistema.

Un sistema es capaz de enmascarar un fallo si es capaz de evitar que los errores causados por dicho fallo se traduzcan en averías. Siguiendo con la estructura en niveles descrita anteriormente se considerará **como nivel de enmascaramiento de fallos** al primer nivel del sistema que presente a los niveles superiores una interfaz libre de averías. El nivel de enmascaramiento de fallos establece el último nivel en el que se pueden incluir herramientas de tolerancia a fallos; sin embargo, las técnicas de tolerancia a fallos también pueden ser incluidas (y suelen serlo) en los niveles inferiores, tal y como se discute en el apartado 1.2.3. Los sistemas encontrados en la bibliografía han escogido diferentes niveles de enmascaramiento de fallos. Cada elección implica ventajas e inconvenientes, que deben ser analizados a la luz de los objetivos del sistema. Se han identificado dos características básicas que deben tenerse en cuenta a la hora de elegir el nivel más adecuado: los fallos que pueden ser tolerados y la transparencia. A continuación se recogen las ventajas e inconvenientes que presenta cada nivel respecto a cada una de estas características.

Fallos que pueden ser tolerados

En [HK93], citando a [SRC84], se afirma que las técnicas de tolerancia a fallos incluidas en el sistema operativo o en el soporte físico para detectar y recuperarse de errores en el soporte lógico son necesariamente incompletas. Como consecuencia, si se elige como nivel de enmascaramiento de fallos el del soporte físico o el del sistema operativo se limita el tipo de fallos que pueden ser tolerados. En concreto, no se pueden tolerar fallos en los niveles de utilidades o aplicaciones.

Por otra parte, el estudio que se recoge en [Gra86], realizado sobre sistemas reales, revela que las averías se deben, en un porcentaje claramente significativo, a fallos de diseño del soporte lógico. Estos datos indican, tal y como se defiende en [Lee94] [Loc94][Chi94], que la disminución de la tasa de averías mediante tolerancia a fallos requiere, cada vez más, que los fallos del soporte lógico sean incluidos en el conjunto de los fallos que deben ser tolerados por el sistema.

Por lo tanto, en general, los niveles del soporte físico y del sistema operativo no resultan adecuados como niveles de enmascaramiento de fallos. El motivo es que,

[13] Es necesario tener presente que la tolerancia a fallos no es el único método para reducir la tasa de averías del sistema, tal y como se recoge en en anexo I.

generalmente, ni el soporte físico ni el sistema operativo constituyen soluciones a problemas concretos, sino que constituyen plataformas sobre las que implementar dichas soluciones mediante componentes lógicos. Si sobre un soporte físico que enmascara sus propios fallos se utiliza un sistema operativo y sobre él se ejecutan aplicaciones que no lo hacen, el sistema no podrá ofrecer una interfaz libre de averías al usuario final, ya que los fallos incluidos en el sistema operativo o en las aplicaciones podrían causar errores que hicieran que el sistema dejase de comportarse con arreglo a sus especificaciones. Del mismo modo, si el nivel de enmascaramiento es el del sistema operativo, todavía podrían aparecer averías debidas a fallos en las aplicaciones.

No obstante, este razonamiento descansa en la suposición de que no se puede asegurar que el soporte lógico se halle libre de fallos y esta suposición no es cierta en todos los casos. Existen aplicaciones en las que el soporte físico realiza la mayor parte del trabajo y el soporte lógico es suficientemente sencillo como para que pueda garantizarse que está totalmente libre de errores. Además, se ha hecho —y se sigue haciendo— un gran esfuerzo por desarrollar técnicas que permitan generar componentes lógicos libre de errores: desde las técnicas de ingeniería del *software* hasta los modelos matemáticos para la verificación formal de programas, pasando por entornos de programación que generan automáticamente el código correspondiente a una descripción formal o esquemática de los programas, etc. Todo este esfuerzo puede utilizarse para garantizar que un subsistema lógico está libre de fallos y que, por lo tanto, al ser ejecutado en un sistema tolerante a fallos ofrece una interfaz libre de averías.

Transparencia

La gran ventaja del soporte físico como nivel de enmascaramiento de fallos es que todos los niveles superiores del sistema se benefician inmediatamente de la reducción en la tasa de averías, con lo que la tolerancia a fallos resulta transparente tanto para el diseñador de aplicaciones como para el de utilidades. En los sistemas descritos en la bibliografía el sistema operativo colabora, en mayor o menor medida, en la tolerancia a fallos, por lo que la transparencia no es completa a este nivel. No obstante, en algunos casos las modificaciones a realizar en el sistema operativo pueden ser reducidas a un pequeño número de módulos de bajo nivel, tal y como se describe en [Jew91] para el sistema Integrity S2 de Tandem Computers. En otros casos, el sistema operativo desempeña un papel decisivo en la tolerancia a fallos, como ocurre con el sistema operativo Guardian [LI93], también de Tandem Computers.

Existen numerosas herramientas y entornos de ejecución de aplicaciones que facilitan el desarrollo de aplicaciones tolerantes a fallos y que consiguen diferente grado de transparencia para las aplicaciones que los utilizan. Por ejemplo, en [HK93] se describe la librería *libft*, una librería de funciones en C que se pueden utilizar en aplicaciones para establecer puntos de recuperación, recuperar los datos incluidos en el punto de recuperación, registrar sucesos, localizar y volver a establecer la conexión con un servidor, manejar excepciones, realizar programación de N versiones y utilizar bloques de recuperación. En este caso se ha dado mayor importancia a la velocidad y portabilidad (que se discuten en el apartado 1.2.3) que a la transparencia, por lo que las aplicaciones deben utilizar explícitamente las funciones de la librería para tolerar

fallos. En cambio, Manetho [Eln93] ofrece recuperación mediante vuelta atrás (*rollback-recovery*) totalmente transparente para las aplicaciones, que no necesitan ser modificadas.

1.2.3 Distribución de técnicas por niveles

Como se ha dicho al definir del nivel de enmascaramiento de fallos, las técnicas de tolerancia a fallos se pueden incluir en dicho nivel o en cualquiera de los niveles inferiores. Las tareas asignadas a cada nivel definen la estructura para tolerancia a fallos, dando lugar a diferentes tipos de sistemas. A la hora de diseñar un sistema tolerante a fallos es necesario distribuir dichas tareas de la forma más adecuada para obtener el compromiso entre garantía de funcionamiento, rendimiento y coste que establezcan los objetivos. Las características recogidas en la bibliografía y que resultan más significativas a la hora de tomar las decisiones relacionadas con la distribución de tareas por niveles son: la flexibilidad, la eficiencia, el coste económico, la fiabilidad y la aplicabilidad. A continuación se comparan las ventajas e inconvenientes de la inclusión de técnicas de tolerancia a fallos en cada uno de los niveles del sistema con respecto a cada una de estas características.

Flexibilidad

La inclusión de técnicas de tolerancia a fallos en el soporte físico da lugar a sistemas muy poco flexibles, pues las técnicas de tolerancia a fallos vienen determinadas por el soporte físico, siendo necesario un nuevo ciclo de diseño para cambiarlas. También es necesario un nuevo ciclo de diseño para permitir que el sistema evolucione y pueda aprovechar los nuevos componentes (procesadores, módulos de memoria, etc.) que vayan apareciendo en el mercado. Por otra parte, los componentes físicos que se utilicen deben ser lo más fiables posible, lo que desaconseja utilizar los componentes de última generación, que pueden contener fallos de diseño o construcción que no se hayan manifestado todavía.

Aunque sea el sistema operativo el que implemente técnicas de tolerancia a fallos sobre un soporte físico estándar, las técnicas incluidas en él no pueden ser modificadas con facilidad, ya que el código fuente del sistema operativo no suele estar disponible —y aunque lo esté, son pocas las aplicaciones que justifican la modificación del sistema operativo. Sin embargo, siguiendo el razonamiento de [Rus94, pp. 2-3] las necesidades de tolerancia a fallos de una estación de trabajo y de un servidor de ficheros son completamente diferentes, aunque ambos computadores utilicen el mismo sistema operativo.

En cambio, cuando las técnicas de tolerancia a fallos se dejan a los niveles superiores es posible ofrecer, en un mismo sistema, múltiples estrategias de tolerancia a fallos a las aplicaciones, de forma que cada aplicación utilice la más adecuada a sus necesidades. Por ejemplo, en Manetho [Eln93] se ofrecen dos técnicas complementarias de tolerancia a fallos: una técnica basada en procesos replicados (modelo líder-cohortes) que proporciona alta disponibilidad a los servidores y otra basada en la recuperación mediante vuelta atrás para el resto de aplicaciones.

Eficiencia

Las técnicas de tolerancia a fallos también se benefician de la velocidad del soporte físico cuando se las incluye en este nivel. Por ello, se ha dedicado un considerable esfuerzo al desarrollo de sistemas con técnicas de tolerancia a fallos incluidas total o parcialmente en él. Un buen ejemplo de este tipo de sistemas lo constituyen los sistemas multiprocesador tolerantes a fallos basados en puntos de recuperación como los propuestos en [Ber88], [BJ90], [BMR+91], [Ors93] o [Fab96]. No obstante, existe el peligro de que, debido a la falta de flexibilidad, estos sistemas puedan ofrecer unas prestaciones inferiores a otros sistemas con técnicas de tolerancia a fallos incluidas en niveles superiores y que utilicen componentes físicos más actuales y con mayores prestaciones.

A nivel del sistema operativo es conveniente distinguir entre dos tipos de sistemas: aquéllos en los que el sistema operativo ha sido especialmente diseñado (o adaptado) para aprovechar ciertas características específicas del soporte físico a la hora de implementar las técnicas de tolerancia a fallos y aquellos otros en los que el sistema operativo implementa estas técnicas apoyándose únicamente en un soporte físico estándar. El primer grupo se beneficia directamente del apoyo a la tolerancia a fallos que incluye el soporte físico [Bar78], [CDR+95]. En cambio, en el segundo, la mayor eficiencia proviene del control absoluto sobre el soporte físico del que dispone el sistema operativo. Por ejemplo, el sistema operativo puede hacer más eficiente el establecimiento de puntos de recuperación de los procesos estableciendo puntos incrementales y concurrentes, tal y como se describe en [Eln93, p. 56] y [Pla93, pp. 50-63].

Ya se ha descrito por qué el sistema operativo permite mayor eficiencia en la implementación de técnicas de tolerancia a fallos; pero la falta de eficiencia es, precisamente, una de las críticas que se realizan a este tipo de sistemas. En [HK93, p. 3] los autores afirman:

> *"Nosotros mantenemos que tales métodos (los métodos de tolerancia a fallos incluidos en el soporte físico y el sistema operativo), es decir, servicios en un nivel inferior para detectar y recuperarse de averías en un nivel superior, son también*[14] *ineficientes."*

La razón aducida es que el sistema operativo no dispone de la información suficiente para aprovechar las características específicas de la aplicación. En el artículo, los autores ponen como ejemplo el caso de los puntos de recuperación: cuando se implementan en el sistema operativo para que resulten transparentes para las aplicaciones, los puntos de recuperación incluyen toda la información mantenida en memoria por la aplicación; mientras que cuando la aplicación colabora en su establecimiento (y en una eventual vuelta atrás) se guardan únicamente los datos estrictamente necesarios para la aplicación. En cambio, el diseñador de aplicaciones

[14] Los autores hacen referencia al artículo de Saltzer y otros [SRC84], en el que se mantiene que las técnicas incluidas en el soporte físico y el sistema operativo para detectar y recuperarse de errores en el soporte lógico son necesariamente incompletas porque, puesto que no disponen de la información suficiente sobre la aplicación. Por lo tanto, con la palabra *también*, los autores reflejan que dichas técnicas son incompletas y, *también,* ineficientes.

dispone de la información adecuada para seleccionar la técnica de tolerancia a fallos más adecuada para cada aplicación, así como de la información necesaria para adaptar la técnica para que se implemente de la forma más eficiente posible. Así, la ventaja de la librería *libft*, nombrada anteriormente, es que permite que sea la aplicación la que establezca cuál es el conjunto de datos crítico que debe ser almacenado en cada punto de recuperación, disminuyendo la cantidad de datos a ser almacenada y aumentando, por tanto, el rendimiento.

No obstante, las técnicas implementadas en los niveles de utilidades y aplicaciones serán, en general, menos eficientes que las implementadas en niveles inferiores del sistema. Además, algunas técnicas de tolerancia a fallos no pueden implementarse de forma eficiente sin la colaboración de los niveles inferiores. Por esta razón, algunos sistemas, como Manetho o Delta-4 [Pow94], utilizan el soporte de las capas inferiores. En Manetho, por razones de eficiencia, el establecimiento de puntos de recuperación y el registro de mensajes se incluyen en el núcleo del sistema operativo. En Delta-4, cuya arquitectura está formada por una serie de nodos conectados por una red de área local, se utiliza una tarjeta de red autocomprobante para garantizar el silencio tras la avería de los nodos (comportamiento *fail-silent*).

Coste económico

La tolerancia a fallos en el soporte físico supone un coste elevado, puesto que es necesario realizar un diseño específico para implementar, por ejemplo, los puntos de recuperación de los sistemas anteriormente citados o configuraciones triple modular redundante como en el caso del sistema Integrity S2. Estos diseños incorporan componentes adicionales para realizar tareas como la votación o la gestión de dependencias entre procesadores, además de replicar, en muchos de ellos, componentes como los módulos de memoria o los procesadores. Por último, el número de unidades producidas por cada diseño será necesariamente menor en estos sistemas que en los que no incluyen medidas de tolerancia a fallos, que son la mayoría, encareciendo el proceso de producción. Este es precisamente uno de los motivos que se recogen en [Lev90] para defender que las técnicas de tolerancia a fallos sean trasladadas del soporte físico al soporte lógico siempre que sea posible.

Entre los diferentes niveles del soporte lógico no existen diferencias tan importantes, respecto al coste económico, como existen entre el soporte físico y el resto de los niveles. No obstante, diseñar o modificar el núcleo del sistema operativo es una tarea compleja, que no suele abordarse con frecuencia porque requiere la disponibilidad del personal adecuado, así como, en el caso de una modificación, el acceso al código fuente. La diferencia entre aplicaciones y utilidades está mucho menos marcada. De hecho, puede resultar más rentable modificar el nivel de utilidades para facilitar el desarrollo posterior de un conjunto específico de aplicaciones, que desarrollar las aplicaciones sobre una plataforma o que adquirir un entorno de desarrollo de terceros.

Fiabilidad

Cuando el diseñador de aplicaciones asume la responsabilidad de implementar las técnicas de tolerancia a fallos la fiabilidad del sistema puede verse seriamente

afectada. En [Rus94, p. 3], al hablar del nivel de utilidades, destaca como un defecto que las aplicaciones tengan que ser modificadas para utilizar dichas técnicas, o para adaptarse a las restricciones del modelo de ejecución (por ejemplo, que la aplicación sea determinista). El autor destaca la necesidad de una persona suficientemente cualificada para utilizar estas herramientas. Si el diseñador de aplicaciones no tiene los conocimientos, la experiencia, o el tiempo (a causa de la presión que ejerce el mercado) necesarios, entonces la probabilidad de introducir nuevos fallos en el sistema aumenta sustancialmente.

No obstante, tal y como se afirma en [HK93], si las herramientas (o aplicaciones) son desarrolladas por personal cualificado y son comprobadas suficientemente, es posible producir herramientas de calidad, cuya fiabilidad sea comparable a la de herramientas incluidas en niveles inferiores del sistema.

Con respecto al soporte físico, su elevado coste hace que no se considere siquiera la inclusión de las técnicas de tolerancia a fallos en este nivel a menos que se disponga de los medios humanos, de diseño y de implementación adecuados.

En los párrafos anteriores se ha mostrado cómo la forma de estructurar los sistemas tolerantes a fallos tiene una influencia decisiva sobre características tan importantes del sistema como su fiabilidad, su coste económico o su rendimiento. Se ha tratado por separado la elección del nivel de enmascaramiento de fallos y la distribución de técnicas por niveles. Y dentro de cada apartado se han descrito las ventajas e inconvenientes de cada nivel en función de los parámetros destacados. Se ha seguido este orden para facilitar la exposición; sin embargo, la Tabla 1-1 recoge, de forma conjunta, las ventajas e inconvenientes de cada nivel del sistema, tanto para el enmascaramiento de fallos como para la inclusión de técnicas de tolerancia a fallos.

Aunque la tabla puede parecer contradictoria a simple vista, puesto que hay propiedades que aparecen en un mismo nivel como ventajas y como desventajas, en realidad no lo es. La gran variedad de sistemas tolerantes a fallos, cada uno utilizando diferentes técnicas de tolerancia a fallos y cada uno con unos objetivos diferentes, hace que lo que resulta adecuado en un caso no lo resulte en otros, tal y como se ha descrito en el presente apartado.

Nivel	Ventajas	Desventajas
Aplicación	• elección de la técnica más adecuada • mayor eficiencia	• falta de transparencia • menor fiabilidad • poca eficiencia
Utilidades	• transparencia • flexibilidad • múltiples estrategias de TaF	• falta de transparencia • poca eficiencia
Sistema operativo	• transparencia • mayor eficiencia	• falta de flexibilidad • poca eficiencia
Soporte físico	• transparencia • mayor eficiencia	• coste elevado • falta de flexibilidad • la evolución exige rediseño • necesidad de utilizar componentes probados

Tabla 1-1: Resumen conjunto de las ventajas e inconvenientes de enmascarar los fallos y de incorporar las técnicas de tolerancia a fallos en cada uno de los niveles del sistema.

1.3 Ejemplos de sistemas relacionados

Los seis ejemplos que se describen a continuación han sido seleccionados por constituir un conjunto de sistemas que son representativos, por una parte, de las técnicas de tolerancia a fallos que pueden ser utilizadas por las aplicaciones que se desarrollen sobre la plataforma que se propone en este trabajo y, por otra, de las diferentes formas de distribuir estas técnicas entre los diferentes niveles del sistema. Los dos primeros son aplicaciones tolerantes a fallos, mientras que los cuatro restantes son propuestas de plataformas para el desarrollo de aplicaciones tolerantes a fallos.

1.3.1 Tandem's Integrity S2

El sistema Integrity S2 [Jew91][SS92, pp. 618-622] de Tandem Computers ofrece una plataforma tolerante a fallos en la que el nivel de enmascaramiento de fallos es el del sistema operativo, ofreciendo al usuario una interfaz de sistema operativo estándar, la de Unix. A nivel del soporte físico, el sistema utiliza una configuración triple modular redundante (TMR) para los módulos de proceso, con memoria local y sincronizados por *hardware*. Además, el sistema dispone de una memoria global duplicada en dos módulos, cada uno de ellos con configuración TMR y que se encargan de la votación de las transacciones realizadas por los módulos de proceso. Así mismo, estos módulos se encargan de gestionar las operaciones de E/S, comunicándose con dos módulos autocomprobantes con parada rápida tras la avería (*fail-fast*), cada uno conectado a un bus de E/S. Todos los dispositivos se hallan conectados a ambos buses, para evitar un punto único de fallo.

Sobre este soporte físico, especialmente diseñado para acoplarse a las versiones existentes de Unix, se ejecuta una versión modificada del sistema operativo. La modificación más importante consiste en la adaptación del sistema de memoria virtual para que gestione la memoria local del módulo de proceso como memoria principal y la memoria global del sistema como respaldo de la local, añadiendo un nivel más a la jerarquía de memoria, ya que el disco se emplea como respaldo de la memoria global del sistema. Así mismo, el almacenamiento intermedio para las operaciones de E/S se realiza en la memoria global, en lugar de en la memoria local de los procesadores. Aunque el soporte físico garantiza que es posible continuar con la ejecución de instrucciones del sistema operativo incluso ante fallos en los módulos de proceso o en la memoria global, el sistema operativo debe gestionar las excepciones debidas a fallos del soporte físico identificando los componentes físicos que hayan fallado y llevando a cabo las tareas necesarias para la recuperación del sistema. El sistema operativo también ha sido modificado para permitir la inserción de nuevos componentes físicos que sustituyan a los averiados, sin que sea necesario parar el sistema. Finalmente, el sistema operativo utiliza recuperación hacia adelante (*roll-forward recovery*) para tolerar sus propios fallos. La detección de errores se realiza mediante aserciones y la recuperación se lleva a cabo por rutinas específicas asociadas a las aserciones.

1.3.2 Sistema Avanzado de Automatización

El Sistema Avanzado de Automatización (AAS, *Advanced Automation System*) es un sistema distribuido de tiempo real desarrollado por IBM para la Administración Federal de Aviación de los EEUU [CDD90]. El objetivo del sistema es proporcionar servicios para el control de tráfico aéreo a los controladores aéreos, que serán los usuarios finales del sistema. Los requerimientos para los sistemas críticos exigen una falta de disponibilidad menor de 3 segundos por año; mientras que para otros servicios el límite está en 156 segundos por año.

El modelo de sistema que utilizan es similar al descrito en el apartado A.I.4, en el que cada subsistema recibe la denominación de servidor, que ofrece servicios (a través de su interfaz) a otros subsistemas, que actúan como clientes de éste. A su vez, estos clientes ofrecen nuevos servicios, actuando como servidores de subsistemas de un nivel de abstracción mayor. El proceso se repite hasta el nivel de abstracción más alto, en el que los servicios se ofrecen directamente al usuario a través de la interfaz del sistema. La implementación de servicios tolerantes a fallos se realiza mediante la utilización de grupos de servidores redundantes y físicamente independientes, para que si falla uno de los miembros del grupo el resto pueda continuar ofreciendo el servicio, enmascarando el fallo. El nivel elegido para el enmascaramiento de fallos es el más alto posible, el de la aplicación, con el objetivo de enmascarar fallos tanto del soporte lógico como del soporte físico. Se considera que los servidores con soporte lógico pueden producir resultados fuera de tiempo o no producirlos en absoluto; pero que no pueden producir resultados erróneos (*performance failures*). En el caso de los servidores *hardware* se considera que únicamente pueden producir resultados correctos o no producirlos; pero que no pueden producir resultados erróneos ni fuera de tiempo (*omission failures*). En [CDD90, p. 9] se describen los motivos de esta decisión, y han sido recogidos en el presente trabajo en el apartado 1.1.3.2.

A nivel del soporte físico decidieron utilizar soporte físico estándar en lugar de utilizar una configuración con dos procesadores funcionando en *lock-step*. La razón aducida es que resultaba demasiado cara y redundante con las comprobaciones en tiempo de ejecución que llevan a cabo los procesadores. Además, al ser los procesadores un componente más dentro del sistema no resultaba económicamente justificable duplicar un sólo eslabón de la cadena.

A nivel de utilidades y aplicación, los grupos de servidores utilizan diferentes políticas de sincronización en función de los tiempos de respuesta que es necesario conseguir. Las políticas de sincronización incluyen categorías similares a las réplicas pasivas y las réplicas activas de Delta-4. Del mismo modo, los niveles de comunicación a través de la red han sido modificados para incluir aspectos como la difusión o la reconfiguración de la red.

1.3.3 Manetho

Manetho es un sistema desarrollado por Elmootazbellah Nabil Elnozahy para su Tesis Doctoral [Eln93]. El sistema proporciona, de forma transparente, tolerancia a fallos del soporte físico para aplicaciones paralelas ejecutándose en un *cluster* de estaciones de trabajo. Cada aplicación está constituida por varias Unidades de Recuperación (RU, *Recover Units*) comunicándose entre sí únicamente mediante intercambio de mensajes sobre una red asíncrona. El modelo de sistema distingue entre dos componentes lógicos básicos: programas de aplicación y servidores de propósito general (por ejemplo, de ficheros o de nombres). Los programas de aplicación actúan como clientes de los servidores y sus necesidades de tolerancia a fallos pasan por la transparencia y la eficiencia durante la ejecución libre de errores. Por su parte, los servidores requieren, además, alta disponibilidad, puesto que el fallo de cualquiera de ellos afecta a todo el sistema. Manetho ofrece recuperación mediante vuelta atrás (*Rollback-Recovery*) para los clientes, y replicación de procesos para los servidores. Los algoritmos de Manetho permiten que formen parte de la misma aplicación procesos que utilicen ambas técnicas de tolerancia a fallos.

Ambas técnicas se apoyan en una estructura de datos, el Grafo de Antecedentes (AG, *Antecedence Graph*), que describe los eventos no deterministas que tienen lugar en el sistema. El AG es mantenido como resultado del protocolo de comunicación entre RUs y de éstas con el mundo exterior. Este protocolo es una extensión de la técnica de registro de mensajes basado en el emisor (*Sender-Based Message Logging*) que combina el alto rendimiento de las técnicas optimistas de registro de mensajes (*Optimistic Message Logging*) con las ventajas de las técnicas pesimistas (*Pessimistic Message Logging*). Todas las RUs siguen este protocolo, tanto si utilizan vuelta atrás como si utilizan replicación de procesos. En caso de fallo, el protocolo de recuperación (también común para todas las RUs), se encarga de devolver al sistema a un estado consistente con el estado de las RUs que no han fallado y con el estado del sistema tal y como es percibido por el mundo exterior.

En cambio, cada tipo de RUs utiliza una estrategia diferente para limitar el tiempo necesario para recuperar al sistema en caso de fallo. Las RUs basadas en vuelta atrás establecen puntos de recuperación coordinados; mientras que las basadas en procesos

replicados utilizan un algoritmo de difusión basado en el modelo líder-cohortes (*Leader-Cohorts*) especialmente diseñado para procesos replicados, que utiliza el AG y el registro de mensajes. En este caso, la sincronización entre réplicas es el equivalente del punto de recuperación coordinado.

Manetho es un ejemplo de sistema en el que las tareas relacionadas con la tolerancia a fallos han sido repartidas entre los diferentes niveles que lo componen con el objeto de aumentar su rendimiento. Así, el establecimiento de puntos de recuperación se ha incluido en el núcleo del sistema operativo, utilizando puntos de recuperación concurrentes y coordinados. También se ha incluido la rutina para registrar los mensajes enviados por una RU, utilizando un *buffer* circular de 512 Kb en el núcleo. La RU se bloquea cuando este *buffer* se llena; no obstante, para disminuir la probabilidad de que se bloquee, cuando se llena el *buffer* se protegen contra escritura los datos en el espacio de direcciones de la RU, con lo que ésta se bloquea únicamente si intenta modificarlos. Posteriormente, cuando queda suficiente espacio libre en el *buffer*, se copian los datos desde el espacio de direcciones del emisor.

La implementación del protocolo de difusión para la coordinación de réplicas se ha incluido en un nuevo nivel sobre el de enlace de datos; mientras que el mantenimiento del AG se realiza al nivel de transporte de datos para conseguir mayor eficiencia. El protocolo de recuperación ha sido implementado utilizando un servidor de recuperación y una serie de rutinas incluidas en el núcleo. Finalmente, el almacenamiento estable ha sido implementado utilizando un servidor replicado, optimizado para almacenar puntos de recuperación.

1.3.4 Delta-4

Delta-4 es el nombre de un proyecto ESPRIT cuyo objetivo era: "Definir y diseñar una arquitectura abierta, distribuida y con garantía de funcionamiento, es decir, una plataforma tolerante a fallos de múltiples vendedores para aplicaciones distribuidas."[15] Delta-4 ofrece un entorno de ejecución, Deltase, que oculta las diferencias entre los sistemas operativos de los nodos y que permite la ejecución de componentes lógicos replicados en distintos nodos de una red de computadores; donde un componente lógico es una parte de una aplicación distribuida que se comunica con otros componentes exclusivamente mediante paso de mensajes.

Delta-4 ofrece tres formas de replicar los componentes lógicos: replicado pasivo [SB89], replicado activo [CPR+92] y replicado mediante el modelo líder/seguidor [BHB+90]. Aunque el replicado activo permite ofrecer garantías de funcionamiento aún cuando los nodos del sistema presentan un comportamiento incontrolado tras la avería (*fail-uncontrolled*), los algoritmos son necesariamente complejos y poco eficientes. Del mismo modo, la interconexión entre nodos debe ser más compleja que si los nodos presentasen un comportamiento de parada tras la avería (*fail-stop*).

[15] Traducido literalmente de [Pow94, p. 37]: *"Define and design an open dependable distributed architecture, that is, a multivendor fault-tolerant platform for distributed applications."*

Por todo ello, la arquitectura de Delta-4, tal y como aparece en la Ilustración 1-9, divide el soporte físico del nodo en dos subsistemas. La computación se lleva a cabo sobre un soporte físico estándar, el *host*, que no ofrece garantías sobre su comportamiento tras la avería; mientras que el protocolo de comunicaciones se ejecuta sobre un soporte físico autocomprobante (*self-checking*): el NAC (*Network Attachement Controller*), que ofrece garantía de parada tras la avería.

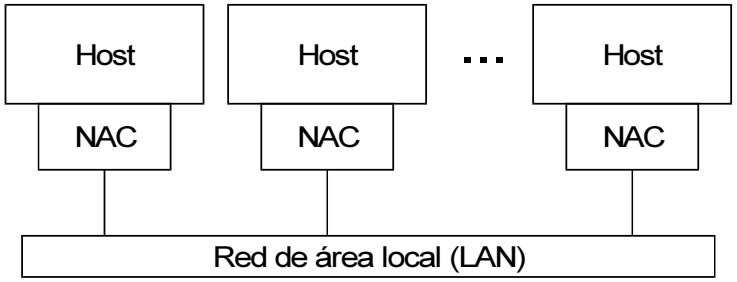

Ilustración 1-9: Arquitectura del soporte físico en Delta-4.[16]

El soporte lógico de Delta-4 está totalmente incluido en el nivel de utilidades, y se ejecuta sobre el sistema operativo del *host* o sobre el núcleo de tiempo real del NAC. Su estructura se muestra en la Ilustración 1-10, donde se muestran los tres subsistemas principales en que se divide el soporte lógico: Deltase, el subsistema de administración y los protocolos de comunicaciones multipunto (*multipoint communication protocol stack*).

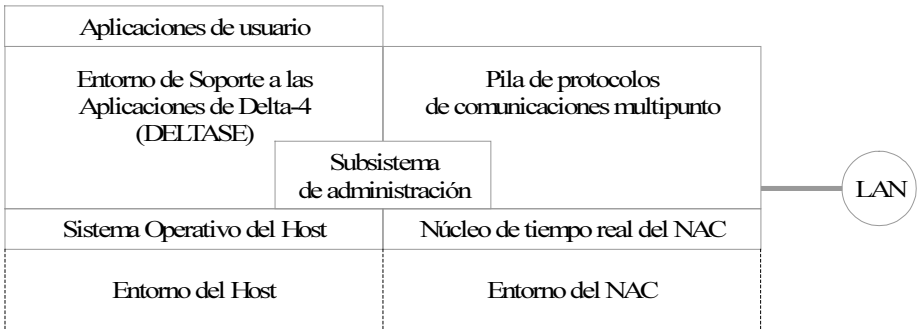

Ilustración 1-10: Arquitectura del soporte lógico en Delta-4.[17]

Deltase se ejecuta completamente sobre el sistema operativo del *host*, y ofrece la infraestructura necesaria para la ejecución de aplicaciones distribuidas, permitiendo la

[16] Esta ilustración está basada en la ilustración 1 de [Pow94, p. 38]

[17] Esta ilustración está basada en la ilustración 2 de [Pow94, p. 41]

creación la interacción entre los componentes lógicos de las aplicaciones. El subsistema de administración se ejecuta en parte sobre el *host* y en parte sobre el NAC, y se encarga de la configuración del sistema (integración de redundancia y distribución), de la gestión del rendimiento y de la gestión de fallos, incluyendo el tratamiento automático y el soporte para el mantenimiento del sistema. Finalmente, el subsistema de comunicación ofrece facilidades para la comunicación entre componentes replicados, ofreciendo diferentes modelos de difusión entre réplicas (atómico, fiable, ...) para acomodarse a las necesidades de las diferentes formas de replicado. Este subsistema incluye tareas de tolerancia a fallos como la detección de errores o la detección y compensación de errores mediante la comparación de mensajes replicados.

1.3.5 Tolerancia a fallos mediante el soporte lógico en AT&T

La propuesta de Y. Huang y C. Kintala en [HK93] ofrece un conjunto de componentes lógicos (servidores y librerías) que pueden ser utilizados para reducir el tiempo y el esfuerzo necesarios para incorporar estrategias de tolerancia a fallos en aplicaciones. Los fallos tolerados son los que conducen a que el proceso (o los procesos) afectados o bien se queden parados (no realicen progreso alguno desde el punto de vista del usuario) o bien desaparezcan del sistema (mueran). El nivel elegido para enmascarar los fallos es el de la aplicación, mientras que las herramientas propuestas han sido implementadas completamente en el nivel de utilidades, para facilitar su portabilidad y reducir el tiempo de desarrollo. El modelo para las aplicaciones consiste en un conjunto de servidores y clientes ejecutándose un sistema distribuido formado por un conjunto de computadores conectados por una LAN o una WAN, en el que cada nodo funciona como nodo de respaldo del siguiente en una configuración de anillo lógico.

El primer componente es *Watchd*, un proceso demonio que vigila que los procesos de la aplicación continúan progresando. Cuando detecta que un proceso ha muerto o se ha parado, lleva a cabo la recuperación a partir del estado almacenado en el último punto de recuperación o del estado inicial del proceso . *Watchd* también se encarga de detectar fallos de los nodos y de reconfigurar la red, así como de vigilarse a sí mismo y de recuperarse si le es posible.

Libft es una librería de funciones de C que pueden utilizarse en las aplicaciones para: especificar puntos de recuperación, establecerlos, recuperar la información guardada, mantener un registro de eventos, localizar y reconectarse con un servidor, manejar excepciones, programar N versiones y usar bloques de recuperación.

Finalmente, *nDFS* es un sistema de ficheros construido sobre los sistemas de ficheros de UNIX y que ofrece la posibilidad de replicar en tiempo real ficheros críticos especificados por los usuarios. Puesto que *nDFS* intercepta las llamadas al sistema de ficheros y las propaga automáticamente a los sistemas de ficheros de respaldo, no es necesario modificar el núcleo del sistema operativo.

1.3.6 Mecanismo Sentry

El mecanismo Sentry [RSS93][RS94], propuesto por Mark E. Russinovich, busca proporcionar tolerancia a fallos transparente para las aplicaciones y constituir un mecanismo de gestión de fallos de propósito general, accesible al usuario y configurable por él. El nivel elegido para el enmascaramiento de fallos es el de utilidades, incluyendo las técnicas de tolerancia a fallos en la interfaz entre el sistema operativo y las aplicaciones.[18]

El mecanismo Sentry ofrece al usuario la posibilidad de activar y desactivar de forma selectiva las diferentes estrategias de detección y recuperación de errores (políticas) disponibles, especificando la política, el proceso y la llamada al sistema.[19] Las ilustraciones siguientes muestran el flujo de control para las llamadas al sistema en la implementación sobre Mach 3.0/UX (UNIX).

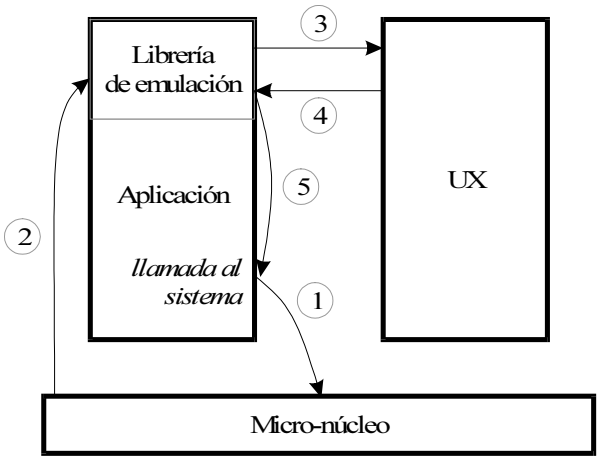

Ilustración 1-11: Flujo de ejecución de una llamada al sistema en Mach 3.0/UX.[20]

En (1) la aplicación realiza una llamada al sistema operativo mediante un trap que es capturado por el micro-núcleo y redirigido (2) a la librería de emulación. Ésta genera un mensaje[21] *con la petición (3) a UX, el servidor de Unix, que la procesa y contesta (4). Finalmente, la librería de emulación devuelve el control a la aplicación (5).*

[18] Mark E. Russinovich utiliza un único nivel, al que denomina sistema operativo, y que comprende los niveles del núcleo del sistema operativo y utilidades. De hecho, en la implementación que ha realizado sobre el micro-núcleo Mach 3.0, el mecanismo se ha incluido en la interfaz que ofrece el proceso servidor de Unix (UX) a sus clientes (procesos Unix).

[19] Llamada al servidor el sistema operativo, UX, no al micro-núcleo.

[20] Ilustración obtenida de [Rus94, p. 14], ilustración 4.

[21] Aunque, para simplificar, en la ilustración no se muestre, la transferencia de mensajes requiere la participación del micro-núcleo.

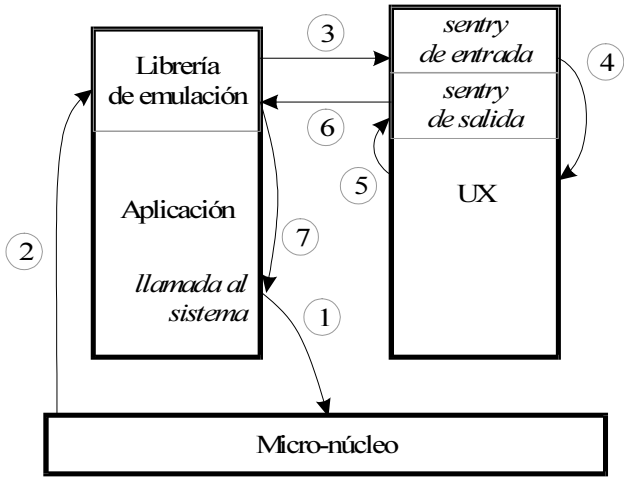

Ilustración 1-12: Flujo de ejecución de una llamada al sistema con el mecanismo Sentry activado.[22]

La diferencia con el caso anterior es el código adicional que se ejecuta en UX antes y después de procesar la solicitud de la aplicación. Al recibir el mensaje con la petición (3) se ejecuta el código Sentry de entrada, después se procesa la petición del usuario (4) y, antes de generar el mensaje con la respuesta (6), se ejecuta el código Sentry de salida.

1.4 Conclusiones

Durante la revisión de las técnicas de tolerancia a fallos, hemos puesto de manifiesto la similitud existente entre muchas de ellas, incluso cuando se aplican en diferentes niveles del sistema. A pesar de la gran variedad de técnicas existentes en la actualidad, tanto las técnicas basadas en recuperación hacia atrás como las técnicas basadas en redundancia modular presentan, dentro de su grupo, principios básicos similares. La identificación de estos aspectos comunes nos permite evaluar de forma conjunta las necesidades de estas técnicas, como paso previo a la especificación del núcleo del sistema operativo.

Tras el análisis de la estructura de los sistemas tolerantes a fallos actuales que se recoge en este capítulo hemos llegado a la conclusión de que existen dos aspectos básicos —el nivel de enmascaramiento de fallos y la distribución de técnicas por niveles— que definen las características generales del sistema desde el punto de vista de la tolerancia a fallos.

El primero establece qué fallos pueden ser tolerados por el sistema y también el nivel a partir del cual las actividades relativas a la tolerancia a fallos resultan totalmente transparentes. El segundo, la forma en que podríamos distribuir las técnicas

[22] Ilustración obtenida de [Rus94, p. 14], ilustración 5.

de tolerancia a fallos entre los diferentes niveles del sistema, influye decisivamente sobre la flexibilidad, la eficiencia, el coste económico y la fiabilidad del mismo.

Como resultado de este análisis, en el presente trabajo de investigación, antes de especificar la arquitectura del sistema estableceremos el nivel de enmascaramiento de fallos y los principios básicos para la distribución de técnicas de tolerancia a fallos entre los diferentes niveles del sistema.

Capítulo 2

Propuesta y justificación de la arquitectura

La revisión de las técnicas de tolerancia a fallos y su aplicación en sistemas reales que se describe en el capítulo anterior ha puesto de manifiesto que las características del sistema dependen, sobre todo, de dos decisiones básicas: el nivel de enmascaramiento de fallos y la forma de distribuir las técnicas de tolerancia a fallos entre los diferentes niveles del sistema.

A continuación, en los apartados 2.1 y 2.2 se describen y se justifican las decisiones relativas al nivel de enmascaramiento de fallos y la distribución de técnicas por niveles, respectivamente. Estas decisiones sirven de base para la propuesta de la arquitectura del sistema, que se recoge en los apartados 2.3, "Arquitectura del soporte físico" y 2.4, "Arquitectura del soporte lógico".

2.1 Elección del nivel de enmascaramiento de fallos

Tradicionalmente, se ha considerado que las plataformas tolerantes a fallos de propósito general debían ocultar las medidas de tolerancia a fallos a las aplicaciones, de forma que los usuarios tuvieran la impresión de trabajar sobre una plataforma libre

de averías. En algunos casos, el nivel de enmascaramiento de fallos ha sido el del soporte físico o el del sistema operativo, con la intención de que todas las aplicaciones del sistema se beneficiaran inmediatamente del aumento en la garantía de funcionamiento (por ejemplo, el sistema de Tandem Computers descrito en el apartado 1.3.1). En otros, el nivel elegido ha sido el de utilidades, permitiendo que las aplicaciones puedan beneficiarse de un aumento en la garantía de funcionamiento al utilizar un determinado soporte en tiempo de ejecución, tal y como se describe en el apartado 1.3.3, por ejemplo.

No obstante, tal y como se analiza en el apartado 1.2.2, la elección de un nivel para el enmascaramiento de fallos por debajo del de aplicación supone una limitación de los fallos que pueden ser tolerados. Aunque para numerosas aplicaciones limitar los fallos que pueden ser tolerados a los del soporte físico, o a los del soporte físico y el sistema operativo, pueda resultar perfectamente aceptable, el objetivo de conseguir una plataforma de propósito general lleva a rechazar estos dos primeros niveles como idóneos para el enmascaramiento de fallos. En general, el nivel de utilidades tampoco resulta satisfactorio por la misma razón, siendo el nivel de aplicación el más recomendable para este tipo de plataformas.

2.2 Distribución de técnicas por niveles

La necesidad de obtener una plataforma versátil y sencilla de utilizar (desde el punto de vista de la tolerancia a fallos) genera una disyuntiva entre transparencia y configurabilidad. En un extremo se encuentra el diseñador de aplicaciones que necesitan técnicas de tolerancia a fallos específicas. Este diseñador detestará aquellos sistemas en los que la tolerancia a fallos sea transparente para las aplicaciones y resulte inadecuada en su caso particular —aunque resulte muy adecuada para un elevado número de aplicaciones—. En el otro extremo se encuentra el diseñador de aplicaciones cuyas necesidades de tolerancia a fallos no son lo suficientemente importantes como para justificar una gran inversión. En este caso, si los mecanismos de tolerancia a fallos son difíciles de utilizar, no los empleará —aunque sean totalmente configurables y puedan ser adaptados perfectamente a sus aplicaciones—.

Esta disyuntiva no es nueva para los diseñadores de sistemas operativos, es la misma que aparece al tratar la relación entre las aplicaciones y el sistema operativo respecto a los recursos del sistema. Si el sistema operativo deja a las aplicaciones acceder directamente al soporte físico, escribir cualquier aplicación general requiere un esfuerzo desproporcionado. Por otro lado, si el sistema operativo intenta lograr que sea sumamente sencillo escribir aplicaciones generales, difícilmente será posible utilizarlo en casos particulares en los que el acceso directo a los recursos se hace necesario.[1] La solución propuesta por las nuevas arquitecturas de sistemas operativos que han surgido a la luz de los micro-núcleos es la separación entre mecanismos y políticas [BGJ+92]

[1] Un ejemplo de esta situación aparece cuando la estrategia de almacenamiento intermedio del sistema operativo no resulta adecuada para un sistema de gestión de bases de datos, mermando su eficiencia.

[RAA+92][PPT+92][Eng95][BF97]. El micro-núcleo se encarga de implementar los mecanismos básicos sobre los que es posible construir sistemas operativos completos que permiten una programación sencilla del sistema y sobre los que también es posible construir aplicaciones complejas que necesiten un acceso directo a ciertos recursos del sistema —como un disco, en el caso de una base de datos con necesidades específicas de almacenamiento intermedio—, ya que el micro-núcleo permite el acceso directo y controlado de los procesos de usuario a algunos de los recursos del sistema.

Pues bien, este mismo principio, que ha probado su utilidad en el caso anterior, aparece como una buena solución al problema de la tolerancia a fallos: deben añadirse unos pocos métodos básicos al sistema operativo que sean lo suficientemente generales como para que con ellos se pueda implementar un amplio rango de estrategias de tolerancia a fallos. Desde estrategias generales, que puedan ser usadas de forma transparente por muchas aplicaciones, hasta estrategias específicas para una aplicación determinada, cuando las generales no resulten adecuadas.

De esta forma, tal y como se muestra en la Ilustración 2-1, el diseñador de aplicaciones encuentra un conjunto de herramientas adecuado para el desarrollo de aplicaciones tolerantes a fallos. Para este conjunto de herramientas se han identificado cuatro características básicas: sencillez de utilización, versatilidad, eficiencia y completitud. A continuación se describen estas cuatro características.

- Las herramientas deben ser **versátiles**, para que con un conjunto reducido de herramientas sea posible implementar un número suficiente de técnicas de tolerancia a fallos, de forma que cada aplicación pueda adoptar la técnica que mejor se adecue a sus necesidades. En este sentido, el conjunto de herramientas debe ser **completo**: debe permitir implementar las técnicas actuales de tolerancia a fallos.

- También deben ser lo más **eficientes** posible, de forma que la disminución en prestaciones sufrida por las aplicaciones que las utilicen se vea compensada por el aumento de la garantía de funcionamiento.

  En este mismo sentido, las herramientas incluidas en el núcleo del sistema operativo no deben suponer sobrecarga alguna para las aplicaciones que no las utilicen y que, por lo tanto, no obtendrán ninguna contraprestación en términos de garantía de funcionamiento. Por ejemplo, un sistema operativo que estableciese puntos de recuperación periódicos de todos los procesos del sistema resultaría inadecuado para una aplicación que utilizase la programación de N versiones como técnica de tolerancia a fallos.

- Para permitir que la tolerancia a fallos sea utilizada en un mayor número de aplicaciones, incluso en aquellas cuyo diseñador no dispone de la experiencia en tolerancia a fallos o del tiempo necesario para implementar sus propias técnicas, las herramientas deben ser **sencillas de utilizar** en las aplicaciones (transparencia).

Ya se ha visto que es difícil combinar la transparencia con la versatilidad y la eficiencia. Por eso, se propone que el conjunto de herramientas se separe en dos clases, tal y como aparece en la Ilustración 2-1. Una primera clase que incluya herramientas básicas, que combinen versatilidad y eficiencia para formar un conjunto completo con

el que poder construir las técnicas de tolerancia a fallos actuales. Y una segunda clase formada por herramientas basadas en las anteriores que permitan construir aplicaciones tolerantes a fallos con un esfuerzo asequible.

En general, en el desarrollo de aplicaciones tolerantes a fallos se utilizará esta segunda clase, que permite un ciclo de desarrollo más corto. Solamente cuando el equipo que vaya a desarrollar la aplicación cuente con el tiempo y los conocimientos necesarios, y la aplicación así lo justifique, se deberán utilizar directamente las herramientas básicas.

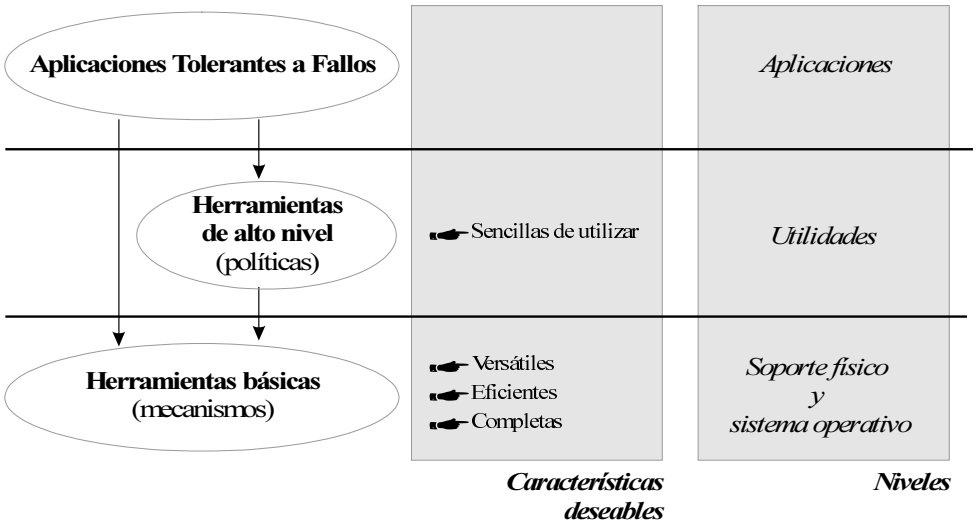

Ilustración 2-1: Separación entre mecanismos y políticas para el desarrollo de aplicaciones tolerantes a fallos.

2.3 Arquitectura del soporte físico

Las decisiones tomadas en los dos apartados anteriores deben guiar la elección de la arquitectura más apropiada para el sistema. Por otra parte, los términos generales en los que se ha definido la arquitectura se corresponden con el nivel de abstracción del presente trabajo de investigación.

A la hora de elegir la arquitectura, únicamente se han considerado arquitecturas con varios módulos de proceso. La razón para ello es que la tolerancia a fallos está basada en la redundancia y, si se utilizase una arquitectura con un único módulo procesador, una avería de este módulo provocaría, irremediablemente, una avería del sistema. A continuación, en el subapartado 2.3.1 se describe la arquitectura elegida; mientras que el subapartado 2.3.2 se justifica el porqué de esta elección.

2.3.1 Descripción de la arquitectura

La arquitectura que se considera más adecuada para una plataforma de propósito general tolerante a fallos, objeto del presente trabajo, es un multicomputador con las características que se describen a continuación, y que se muestran en la Ilustración 2-2.

La memoria del sistema debe estar físicamente distribuida, de manera que el espacio de direcciones de un procesador incluya únicamente los módulos de memoria locales a su nodo. Es decir, el acceso a la memoria local resultará transparente para el soporte lógico; pero no así el acceso a la memoria de otros nodos, que requerirá la colaboración, como mínimo, del núcleo del sistema operativo. Aunque en la ilustración, por simplicidad, se han representado nodos con un único procesador, el trabajo realizado también es válido si uno o varios de los nodos del sistema son multiprocesadores.

Cada uno de los nodos del sistema ejecutará su propia copia del núcleo del sistema operativo, que permitirá la comunicación entre procesos mediante paso de mensajes.

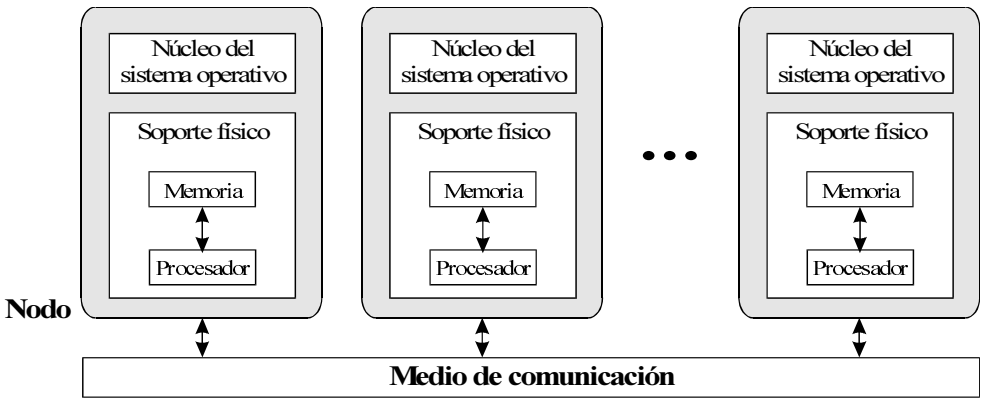

Ilustración 2-2: Arquitectura del soporte físico.

Esta arquitectura incluye a un amplio rango de sistemas que van desde estaciones de trabajo conectadas mediante una red local hasta sistemas formados por placas alojadas en un mismo *rack* y con un medio de comunicación específico. El único requerimiento sobre dicho medio de comunicación es que debe ser tan fiable como las necesidades de las aplicaciones a ejecutar lo requieran. Por ejemplo, si se desea evitar un único punto de fallo en el sistema, deberían utilizarse al menos dos redes de área local.

2.3.2 Justificación

La elección de la arquitectura descrita en apartado anterior se justifica en base a las ventajas que aportan sus características, y que se resumen en la Tabla 2-1. Por ello, la justificación se ha organizado de la siguiente forma. Primero se justifican las tres primeras características, que son típicas de un multicomputador y aportan, frente a los

multiprocesadores con memoria compartida, las tres primeras ventajas de la tabla. Y después se justifica la cuarta de las características, que exige la presencia de una copia independiente del núcleo del sistema operativo en cada nodo, y que aporta una mayor independencia de los modos de fallo.

Características	Ventajas
• Memoria físicamente distribuida • Espacios de direcciones de los procesadores separados • Sincronización y comunicación por paso de mensajes • Cada nodo ejecuta su propia copia del núcleo	• Es posible utilizar tanto soporte físico estándar como tolerante a fallos. • Permite adoptar diferentes configuraciones • Permite implementar diferentes técnicas de tolerancia a fallos • Mayor independencia de los modos de fallo

Tabla 2-1: Resumen de las características y las ventajas de la arquitectura propuesta para el soporte físico.

Multicomputadores frente a multiprocesadores con memoria compartida

Si los módulos de proceso comparten la memoria del sistema, entonces un fallo en cualquier módulo de proceso o en el soporte lógico que se está ejecutando en dicho módulo puede producir un error cuyos efectos se pueden propagar al resto del sistema a través de la memoria compartida. Además, un fallo en la memoria compartida puede afectar a cualquiera de los módulos de proceso. Por este motivo, los numerosos trabajos que han buscado tolerancia a fallos utilizando como soporte físico un multiprocesador han utilizado un soporte físico especialmente diseñado, y han utilizado puntos de recuperación y vuelta atrás como técnica para tolerar los fallos en los componentes físicos del sistema, tal y como se describe en el apartado 1.1.2.2. Dentro de este tipo de sistemas, los que utilizan módulos físicos especialmente diseñados que registran las dependencias entre los diferentes módulos de proceso son los que mejores prestaciones obtienen, ya que, de esta forma, únicamente aquellos módulos de proceso cuya ejecución depende de la del módulo averiado deben volver atrás.

En cambio, en un multicomputador es posible utilizar un soporte físico basado en componentes estándar y mantener copias de los datos en nodos diferentes para tolerar fallos en cualquiera de los nodos, considerando como unidad de recuperación el nodo. La garantía de funcionamiento así obtenida ha sido considerada como suficiente tanto en proyectos de investigación [Eln93] como en aplicaciones comerciales [CDD90][HK93]. No obstante, es necesario recalcar que las técnicas empleadas se basan en la suposición de que los nodos observarán un comportamiento de silencio tras la avería, lo cual resultará justificable o no en función de la garantía de funcionamiento requerida. La técnica utilizada en Delta-4, de utilizar un adaptador de red

autocomprobante (ver el apartado 1.3.4) para garantizar la parada tras la avería podría ser utilizada perfectamente en este caso.

La necesidad de utilizar un soporte físico especializado conlleva una serie de desventajas muy importantes, que han sido discutidas en el capítulo anterior, relativas al coste del sistema y la dificultad para que éste evolucione utilizando componentes físicos más potentes a medida que van apareciendo en el mercado. Además, las técnicas de tolerancia a fallos que pueden implementarse en el soporte físico no son adecuadas para tolerar fallos de los niveles superiores del sistema. Debido a estas dos razones, no es aconsejable que la arquitectura del sistema requiera la utilización de un soporte físico especializado, siendo preferible utilizar un multicomputador en vez de un multiprocesador.

Por otra parte, la arquitectura propuesta permite la implementación de las numerosas técnicas de tolerancia a fallos que existen para el soporte lógico, y que permiten tolerar fallos en cualquiera de los niveles del sistema. En cambio, en un multiprocesador, el diseño del soporte físico establece la técnica de tolerancia a fallos. En los sistemas propuestos en la bibliografía, el objetivo era bien la transparencia al nivel del sistema operativo, bien la tolerancia a fallos eficiente para una aplicación concreta (sistemas transaccionales de bases de datos o sistemas críticos de control, sobre todo).

Una copia por nodo del núcleo del sistema operativo

Aunque las técnicas modernas de ingeniería del *software* permiten reducir enormemente el número de fallos que contiene el soporte lógico de los computadores, la complejidad creciente de los sistemas actuales, junto con la presión que impone el mercado sobre el ciclo de desarrollo —por la necesidad de hacerlo lo más corto posible— hace que los fallos del soporte lógico sean más frecuentes que los fallos del soporte físico [Gra90]. Ésta ha sido, precisamente, una de las razones que ha llevado a la elección del nivel de aplicación como nivel de enmascaramiento de fallos, en el apartado 2.1.

No obstante, tal y como se explicó en el apartado 1.1.3.2, las técnicas para el desarrollo del soporte lógico permiten eliminar la mayor parte de los fallos que se manifiestan cada vez que utiliza un módulo lógico. A causa de esto, la mayor parte de los fallos contenidos por los componentes lógicos tras el desarrollo y la depuración se manifiestan en situaciones excepcionales provocadas por la interacción entre los diferentes componentes lógicos.

Por lo tanto, la arquitectura propuesta aporta una mayor independencia de los modos de fallo cuando se ejecutan varias réplicas del mismo proceso en nodos diferentes, ya que cada una de ellas interacciona con un núcleo de sistema operativo diferente, disminuyendo la probabilidad de que los fallos se manifiesten en varias réplicas simultáneamente.

2.4 Arquitectura del soporte lógico

La elección de la arquitectura del soporte lógico que se describe a continuación no es independiente de las decisiones tomadas en los apartados 2.1 y 2.2, aunque se describe por separado porque se considera que de esta forma queda mejor estructurada la presente memoria. De hecho, fue la existencia de los micro-núcleos la que sugirió la posibilidad de estructurar las técnicas de tolerancia a fallos en la forma descrita en aquél apartado.

2.4.1 Descripción de la arquitectura

La arquitectura propuesta para el soporte lógico (ver Ilustración 2-3) incluye un sistema operativo estructurado como un núcleo básico, el micro-núcleo, que incorpora herramientas básicas a las que se denominan mecanismos. Sobre este micro-núcleo se construyen los servicios de los sistemas operativos tradicionales, utilizando para ello servidores y librerías.

El micro-núcleo es la parte del sistema operativo que se ejecuta en modo supervisor y que interacciona directamente con el soporte físico. Basándose en él, ofrece una serie de servicios básicos que pueden ser utilizados por los niveles superiores del soporte lógico para construir servicios más elaborados. Usualmente, el micro-núcleo gestiona los procesadores, la memoria física y el controlador de interrupciones. En cambio, la mayor parte de los dispositivos del sistema son gestionados por módulos lógicos de niveles superiores. En estos casos, el micro-núcleo incluye, simplemente, una serie de servicios para permitir el acceso directo a los dispositivos desde niveles superiores (habilitar el acceso desde un proceso a un rango de direcciones de entrada salida, por ejemplo) y para que puedan sincronizarse con la llegada de interrupciones.

Los servidores son procesos que actúan como gestores de recursos de un determinado tipo. En unos casos, los recursos tienen una correspondencia directa con dispositivos físicos, como en el caso de un servidor de disco, que gestionará uno o varios discos ofreciendo servicios para leer y escribir bloques del mismo. En otros, los recursos son más abstractos, en el sentido de que no tienen una correspondencia directa con un soporte físico determinado, como en el caso de un servidor de ficheros. Otros procesos, los clientes, pueden utilizar estos recursos enviando un mensaje al servidor correspondiente. El mensaje contiene el tipo de operación que se desea realizar y los datos necesarios para realizarla. El servidor, si la petición es correcta, la lleva a cabo y devuelve, en otro mensaje, el resultado de la operación al cliente. Naturalmente, los servidores pueden apoyarse unos en otros, dando lugar a una estructura en que un servidor es, a su vez, cliente de otros servidores. Por ejemplo, el servidor de ficheros actuará, típicamente, como cliente de otros servidores que le proporcionen acceso a dispositivos de almacenamiento secundario, como servidores de disco.

Por su parte, las librerías se enlazan con los procesos, por lo que tienen acceso a su espacio de direcciones, y sirven de complemento a los servidores para implementar los servicios tradicionales ofrecidos por los sistemas operativos.

Por lo tanto, en los sistemas basados en micro-núcleo, son los servidores y las librerías los que ofrecen a las aplicaciones los servicios elaborados que éstas utilizan y es en ellos donde se incluyen la mayor parte de las decisiones sobre cómo se proporcionan estos servicios (cómo implementar el almacenamiento intermedio, por ejemplo). Así, se dice que los servidores y librerías incluyen las políticas mientras que el micro-núcleo, por su parte, provee los mecanismos necesarios para implementarlas.

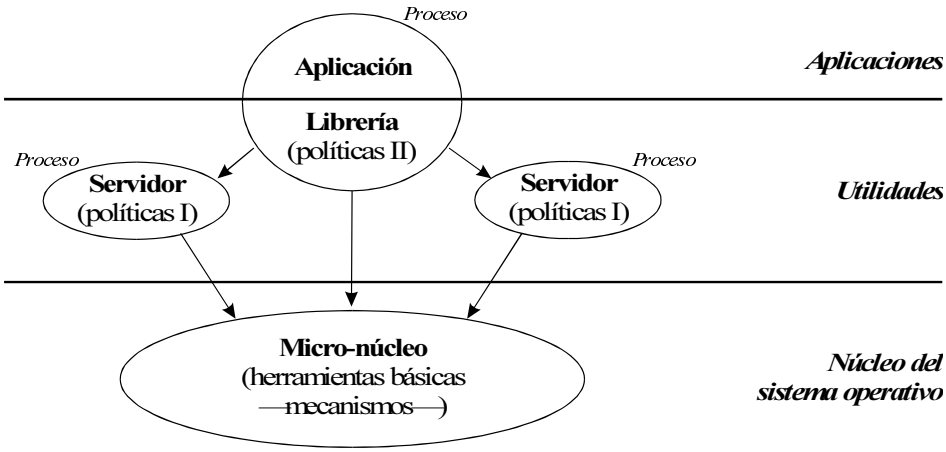

Ilustración 2-3: Arquitectura del soporte lógico.

Los diseñadores de sistemas operativos han llegado a soluciones diversas a la hora de establecer el nivel de abstracción de los servicios, es decir, sobre su correspondencia con el soporte físico. Por ejemplo, el micro-núcleo puede exportar servicios para la manipulación del TLB del procesador o puede proporcionar servicios de un mayor nivel de abstracción, como la creación o cambio de tamaño de una región de memoria virtual.

La elección del nivel de abstracción es importante, ya que el tipo de herramientas a incluir en el núcleo debe adaptarse al nivel de las abstracciones que éste proporcione. Por las razones que se exponen en el apartado siguiente, se ha escogido la distribución de funciones que se muestra en la Tabla 2-2. El núcleo de sistema operativo implementa completamente la abstracción de proceso y se encarga, así mismo, de la gestión de memoria, de la comunicación entre procesos y la gestión básica de interrupciones y excepciones. Por su parte, el nivel de utilidades se encargará de completar el resto de servicios del sistema operativo.

Núcleo del sistema operativo	Utilidades
• Gestión de procesos • Gestión de memoria • Comunicación entre procesos (paso de mensajes) • Gestión básica de interrupciones y excepciones	• Manejadores de dispositivos • Servidor de ficheros • Servidor de comunicación entre nodos • Otros servicios

Tabla 2-2: Resumen de las actividades desarrolladas en los niveles del núcleo del sistema operativo y de utilidades.

2.4.2 Justificación

Tal y como se ha discutido en el apartado 2.2, el diseñador de aplicaciones tolerantes a fallos debe encontrar un conjunto suficientemente completo de herramientas versátiles, eficientes y fáciles de utilizar. Utilizando la arquitectura propuesta, es posible construir este doble conjunto de herramientas: uno básico, en el que primen la eficiencia y la versatilidad, y otro más elaborado, en el que prime la facilidad de uso.

Siguiendo la misma filosofía que ha inspirado esta arquitectura, en el micro-núcleo debe estar el primer nivel de herramientas, que pueden ser identificadas con los mecanismos. Los servicios incluidos en el micro-núcleo deben evitar incluir decisiones (políticas), limitándose a incluir los mecanismos para que los niveles superiores del sistema sean quienes tomen estas decisiones. Por ejemplo, el micro-núcleo puede incluir la posibilidad de establecer un punto de recuperación; pero el contenido y la frecuencia con que debe ser establecido son decisiones que deben ser dejadas a los niveles superiores del sistema: librerías y servidores. Siguiendo con el ejemplo, el micro-núcleo puede ofrecer un servicio para establecer el punto de recuperación de un proceso y servicios adicionales para establecer qué datos deben ser incluidos o excluidos de él. Cuando la aplicación lo estima oportuno, utiliza el servicio para obtener del micro-núcleo la información necesaria, que es enviada por el micro-núcleo a un proceso de usuario, que será el responsable de gestionarla, por lo que se le puede denominar servidor de puntos de recuperación.

La flexibilidad de esta arquitectura permite que coexistan en el mismo sistema diferentes políticas (técnicas) de tolerancia a fallos, existiendo diferentes tipos de herramientas. Pueden existir herramientas que resulten transparentes para el diseñador de aplicaciones (máxima facilidad de uso), y otras que, aún siendo relativamente sencillas de utilizar, le permitan al diseñador de aplicaciones adaptarlas a sus necesidades. Finalmente, el diseñador experto, que se enfrente a una aplicación compleja que, por sus características, pueda beneficiarse de una técnica muy específica (o no implementada), siempre podrá desarrollarla a partir de las herramientas básicas contenidas en el micro-núcleo.

A la hora de diseñar un micro-núcleo es posible, tal y como se ha dicho en el apartado anterior, elegir diferentes niveles de abstracción para los servicios ofrecidos

por el núcleo, dando lugar a propuestas muy diferentes. En concreto, en el momento de decidir la arquitectura del sistema operativo (o de decidir el nivel de abstracción de los servicios del núcleo) se consideraron varias propuestas, entre las que destacan las arquitecturas de Exo-kernel [Eng95] y de Cache-kernel [CD94].

En la primera de ellas se distribuyen las funciones del sistema operativo en dos módulos. Un núcleo mínimo (el *exokernel*) se encarga de ofrecer los servicios del soporte físico añadiendo únicamente una multiplexación segura de su uso. Todo el resto de servicios del sistema operativo se implementa en el nivel de utilidades, mediante librerías que se enlazan directamente con las aplicaciones. El objetivo básico de esta propuesta es separar la protección de los recursos de su gestión y, para ello, utiliza tres técnicas: la asignación segura de recursos (*secure binding*) a cada conjunto aplicación-S.O., la revocación explícita de recursos (*visible resource revocation*) y la revocación forzosa de las asignaciones (*abort protocol*), que se utiliza cuando la aplicación no coopera en la revocación explícita para garantizar que ninguna aplicación monopoliza los recursos físicos del sistema.

En la segunda arquitectura también se reparten los recursos del sistema entre diferentes módulos que se ejecutan en modo usuario y que, en este caso, se denominan núcleos de aplicación (*application kernels*). El código que se ejecuta en modo supervisor (el *cache-kernel*) implementa versiones reducidas de hilos (simplemente el contexto del procesador) y de espacios de direcciones (simplemente un conjunto de correspondencias entre direcciones virtuales y físicas, junto con los derechos de acceso). El nombre se debe a la interacción entre el *cache-kernel* y los núcleos de aplicación. Cada núcleo de aplicación es responsable del almacenamiento de los hilos y espacios de direcciones que cree, mientras que el *cache-kernel* almacena únicamente una copia temporal del objeto, volcándola de nuevo sobre el espacio del núcleo de aplicación cuando necesita liberar el espacio correspondiente para alojar otros objetos.

Como puede observarse, el nivel de abstracción de los servicios del núcleo en estas propuestas, así como de los objetos que gestionan son muy diferentes a los de la opción elegida, cuya descripción se recoge en la Tabla 2-2, y que son comparables a los de los sistemas operativos Mach [BGJ+92], Chorus [RAA+92], Amoeba[2] [RT92] y VSTa [Val]. Las razones que han motivado la elección de este nivel de abstracción son los siguientes:[3]

- Está muy consolidado, habiendo sido probado con éxito en el diseño e implementación de muchos sistemas que iban desde los fuertemente acoplados hasta sistemas distribuidos.
- Desde hace tiempo, existen múltiples micro-núcleos que pueden ser tomados como ejemplo o como base para el trabajo de investigación.
- Por último, una interfaz bien diseñada que sea válida para arquitecturas micro-núcleo puede ser aplicada a un mayor número de sistemas, pues las

[2] Amoeba incluye la comunicación entre nodos como parte del núcleo.

[3] En el momento de escribir la presente memoria, todavía son válidas todas estas razones; aunque existen nuevas propuestas que serán consideradas al hablar de las líneas de investigación abiertas por el presente trabajo.

arquitecturas alternativas no están consolidadas y pueden desaparecer o pasar mucho tiempo antes de que lleguen a un nivel de aceptación parecido al que tienen los micro-núcleos.

2.5 Conclusiones

En este capítulo, teniendo en cuenta las conclusiones obtenidas tras el análisis del capítulo 1, se ha comenzado por especificar el nivel de aplicación como nivel de enmascaramiento de fallos. Después, se ha abordado el problema de la distribución de técnicas por niveles y se ha decidido que el sistema ofrezca al diseñador de aplicaciones tolerantes a fallos un conjunto completo de herramientas versátiles, eficientes y sencillas de utilizar. Con la finalidad de combinar estas características, se ha decidido distribuir este conjunto entre el nivel del núcleo del sistema operativo y el de utilidades. Mientras el núcleo proporcionará un conjunto completo de herramientas versátiles y eficientes; en el nivel de utilidades se utilizarán este conjunto básico de herramientas para implementar otras más elaboradas que sean sencillas de utilizar por los diseñadores de aplicaciones.

A continuación, se ha propuesto para el sistema una arquitectura multicomputador con sistema operativo con estructura de micro-núcleo, y se han descrito las características básicas de la arquitectura y las funciones a desempeñar por el micro-núcleo. La propuesta se ha justificado en base a la revisión del capítulo anterior, comprobando que, efectivamente, es la más aconsejable teniendo en cuenta el nivel de enmascaramiento de fallos que se ha establecido y la distribución de técnicas por niveles que debe realizarse. La arquitectura del sistema permite utilizar tanto soporte físico estándar como tolerante a fallos, y también implementar diferentes técnicas de tolerancia a fallos, aportando la flexibilidad necesaria para obtener el compromiso adecuado entre coste económico y prestaciones. Además, al ejecutar cada nodo una copia del núcleo del sistema operativo, se obtiene mayor independencia de los modos de fallo. Por otra parte, la estructura de micro-núcleo resulta adecuada, por su flexibilidad, para incluir un conjunto de mecanismos básicos en el núcleo que permitan la implementación, mediante servidores y librerías, de técnicas de tolerancia a fallos más elaboradas que puedan incorporarse de forma sencilla a las aplicaciones que lo necesiten.

Capítulo 3

Función del micro-núcleo en aplicaciones tolerantes a fallos: necesidades

Una vez establecida la arquitectura del sistema y siguiendo con los objetivos de la investigación, es necesario identificar los mecanismos que deben ser incluidos en el micro-núcleo para formar el primer conjunto de herramientas. Las características básicas de estas herramientas deben ser, tal y como se justificó en el apartado 2.2, la eficiencia, la versatilidad y la completitud.

Para que este conjunto sea reducido y, sin embargo, completo, es necesario identificar las necesidades comunes de las diferentes técnicas de tolerancia a fallos empleadas habitualmente, y que, tal y como se ha resaltado en el apartado 1.1, presentan muchos puntos en común, incluso cuando se implementen en diferentes niveles del sistema. No obstante, a la hora de implementarlas, aparecen diferencias tan importantes que pueden provocar que, pese a su raíz común, no deban ser consideradas como comunes al nivel del micro-núcleo. La evaluación de las necesidades de las aplicaciones reales, representadas por los casos de ejemplo descritos en el apartado 1.3, ha sido esencial para identificar las necesidades verdaderamente comunes a nivel del micro-núcleo, tal y como se recoge en los apartados siguientes.

3.1 Los puntos de recuperación

Todas las técnicas de recuperación hacia atrás que se describen en el apartado 1.1.2 comparten la necesidad de almacenar el estado del sistema para disponer de un estado libre de errores al que devolver el sistema en caso de que se detecte un error.

No obstante, tal y como se muestra en la Ilustración 3-1, la información que es necesario almacenar varía enormemente de una técnica a otra. Por ejemplo, algunas técnicas basadas en puntos de recuperación buscan la transparencia para la aplicación y almacenan el estado completo de los procesos; mientras que otras técnicas, como los bloques de recuperación o las réplicas pasivas, almacenan únicamente una parte concreta del espacio de direcciones del proceso, que contiene los datos que deben ser recuperados en caso de error. De los casos de ejemplo, Manetho ha seguido la primera alternativa, incluyendo las rutinas para almacenar y recuperar el estado en el núcleo del sistema operativo y aprovechando este hecho para introducir optimizaciones como establecer puntos de recuperación incrementales. En cambio, en el sistema de AT&T descrito en los casos de ejemplo y en Delta-4 se ha optado por la segunda alternativa.

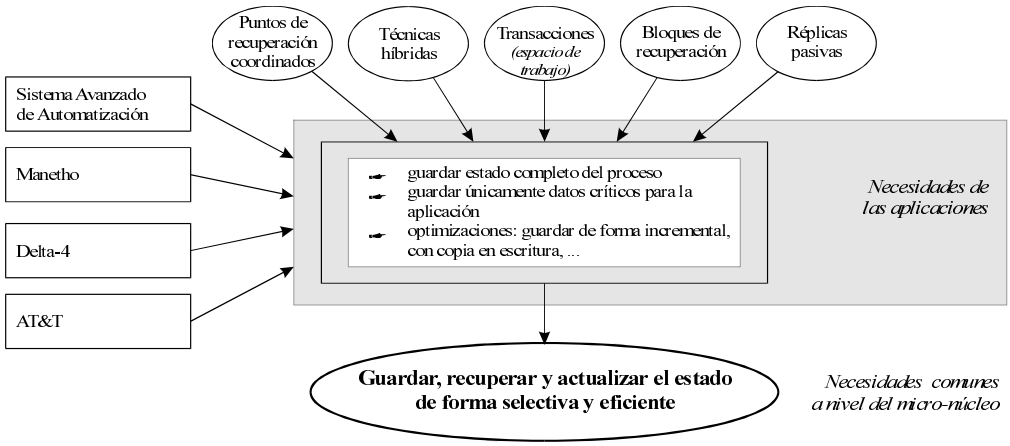

Ilustración 3-1: Necesidades comunes relativas al estado de los procesos.

Además, en el caso de las transacciones, cuando se le asigna una copia privada de los datos a cada una, aparece la necesidad de actualizar los datos originales cuando una transacción termina con éxito. Esta necesidad es común a otras técnicas de recuperación hacia atrás, como las basadas en puntos de recuperación. De hecho, un punto de recuperación incremental no es más que la información necesaria para

[1] Clave para la interpretación de esta ilustración y de las dos siguientes: La parte superior de la ilustración muestra las diferentes técnicas de tolerancia a fallos que comparten esta necesidad de almacenar y recuperar el estado de los procesos. La parte central sombreada recoge las diferentes formas en que las diferentes técnicas necesitan llevar a cabo estas operaciones. La parte de la izquierda muestra los sistemas de ejemplo que emplean estas técnicas y, finalmente, la parte inferior muestra las necesidades comunes al nivel del micro-núcleo.

actualizar la copia del estado del proceso que se guardó al establecer el punto de recuperación anterior. La misma necesidad aparece en el caso de la redundancia modular con réplicas pasivas, donde la única instancia activa suele actualizar periódicamente el estado del resto de las réplicas.

Por lo tanto, las herramientas incluidas en el micro-núcleo deben permitir que los procesos guarden y recuperen de forma eficiente aquella parte de su estado que sea necesaria, que puede abarcar desde el estado completo del proceso hasta una región concreta de su espacio de direcciones, en función de la aplicación y de la técnica de recuperación hacia atrás que esté utilizando. Tal y como se resume en la parte inferior de la Ilustración 3-1, la necesidad común a nivel del micro-núcleo consiste en permitir **guardar y recuperar** de forma **selectiva** y **eficiente** el estado de los procesos, así como permitir la **actualización** eficiente de las copias de su estado.

3.2 Los registros

La Ilustración 3-2 muestra, resumidas, las diferentes necesidades de los sistemas y las técnicas de tolerancia a fallos que utilizan registros. Estas necesidades se han separado en dos grupos: las relacionadas con la gestión del registro, que se recogen en el de la izquierda, y las relacionadas con la generación de la información, que se recogen en el bloque de la derecha.

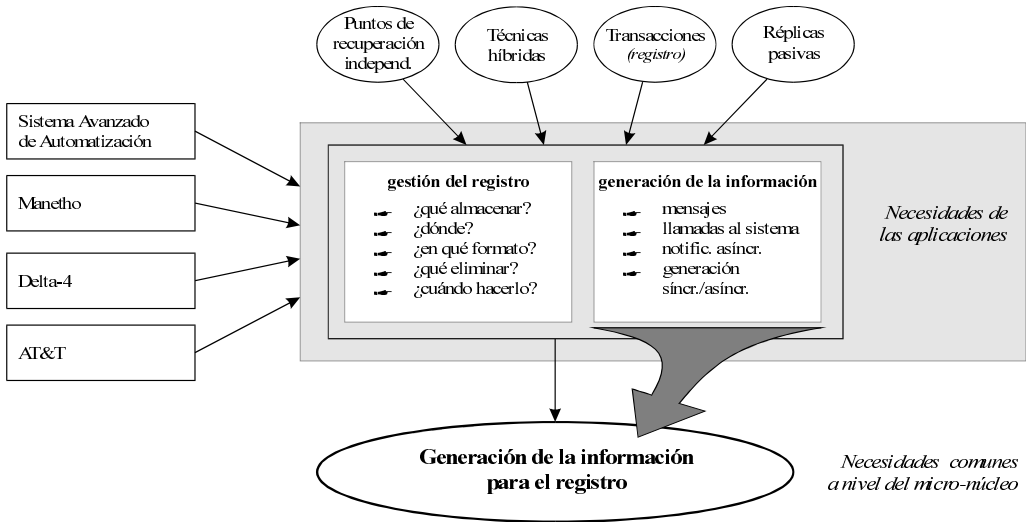

Ilustración 3-2: Necesidades comunes relativas al registro de mensajes.

Generación de la información para el registro

Al describir la recuperación hacia atrás, se ha visto que es posible utilizar registros para permitir que los procesos puedan establecer puntos de recuperación de forma

independiente o para disminuir la frecuencia con que deben ser establecidos de forma coordinada. La información que debe poder almacenarse en el registro incluye, tal y como se justifica en los dos párrafos siguientes: los mensajes enviados y recibidos por los procesos, las llamadas al núcleo del sistema operativo realizadas por los procesos y la notificación de eventos por parte del núcleo del sistema operativo.

El registro debe contener la información necesaria para que los procesos, en caso de vuelta atrás, puedan alcanzar un estado consistente a partir del punto de recuperación (*checkpoint*). Por ello, todas las técnicas basadas en registros recogen información sobre los mensajes y exigen que los procesos sean deterministas.

El modelo de ejecución de los procesos queda, de esta manera, fuertemente limitado. Por ejemplo, en general, la sincronización entre dos hilos de un proceso puede hacer que éste no sea determinista. No obstante, si se incluye información en el registro sobre el orden en el que los hilos realizan las operaciones de sincronización, es posible, durante la recuperación, obligar a que se respete el orden de la ejecución original, garantizando que el proceso alcanza finalmente un estado consistente con el resto del sistema [Eln92]. Por eso, en Manetho se recoge información sobre las llamadas al sistema,[2] permitiendo que los hilos de un proceso se sincronicen mediante llamadas al sistema. En cambio, no se permiten otras formas de indeterminismo, como las señales asíncronas, que, sin embargo, sí se permiten en [Rus94, pp.37-40][3].

En función de la técnica empleada, aparecen dos diferencias importantes en cuanto a la generación de la información para el registro. Por un lado, puede ser necesario almacenar el contenido del mensaje o solamente una indicación del orden en el que se procesan los mensajes y, por otro, en función de si la estrategia empleada es optimista o pesimista, los mensajes se guardarán síncronamente o se permitirá la ejecución del proceso mientras se almacena la información en el registro, respectivamente.

Si el mecanismo básico para obtener la información que debe ser almacenada en los registros se incluye en el núcleo, el código y los datos para la generación del registro están protegidos de accesos no deseados en caso de error de los procesos cuya actividad está siendo registrada, consiguiéndose mayor inmunidad frente al fallo de los procesos monitorizados.[4] Además, de esta forma se reduce el número de llamadas al sistema necesarias para mantener los registros. Por ejemplo, en el caso de los mensajes, el proceso realizaría una única llamada al sistema para solicitar el envío o la

[2] No se recoge información sobre todas las llamadas al sistema, sino sólo sobre aquéllas que no pueden ser repetidas o sobre aquéllas que deben ser repetidas y deben devolver el mismo resultado. [Eln92, p. 54]

[3] En [Rus94] se permite la recepción de señales asíncronas por parte del proceso; pero sólo cuando éste está volviendo de una llamada al sistema.

[4] Si el código para generar la información para el registro forma parte de la librería del proceso, un fallo en el proceso podría afectar a la información que se envía al registro, dando lugar a datos incorrectos en el registro, desde el punto de vista de que no se corresponden con la actividad que realmente ha llevado a cabo el proceso. Por ejemplo, la alteración de la librería podría hacer que se registrase un tipo de llamada al sistema diferente al que realmente se ha realizado. O podría ocurrir que, por error, se produjese un cambio en el flujo de ejecución del proceso que hiciera registrar un evento que nunca se ha producido realmente.

recepción del mensaje, generándose automáticamente la información necesaria para el registro.

Dado que este grupo contiene un conjunto de características reducido, que puede ser utilizado para satisfacer las necesidades de aplicaciones muy diferentes entre sí, y puesto que, tal y como se ha discutido anteriormente, su inclusión en el micro-núcleo aporta suficientes ventajas, se ha identificado como necesidad común a nivel del micro-núcleo la **generación de la información para el registro**.

Gestión del registro

El segundo grupo de necesidades, relativas a la gestión del registro, se muestran en forma interrogativa en la Ilustración 3-2 para resaltar las diferencias que existen entre los diferentes tipos de gestión.

Por ejemplo, en [JZ87] se almacenan en almacenamiento volátil del nodo del emisor los mensajes enviados; mientras que en las estrategias pesimistas es usual guardar en almacenamiento estable los mensajes recibidos. En Manetho, sin embargo, se almacena temporalmente la información en la memoria del núcleo del nodo del emisor, de donde va siendo enviada posteriormente al almacenamiento estable. Además, en un sistema de propósito general sería natural que se ejecutasen diferentes aplicaciones que utilizasen técnicas distintas de tolerancia fallos, dando lugar a que existiesen simultáneamente en el mismo sistema varios registros con diferente tipo de información y almacenada de diferente forma.

Otro ejemplo que muestra las diferencias entre sistemas es la eliminación de la información obsoleta del registro. En el caso de los puntos de recuperación coordinados en técnicas híbridas, la información del registro puede ser eliminada cuando se establece un punto de recuperación, puesto que se deshacen las dependencias. En cambio, con los puntos de recuperación independientes, el establecimiento de un punto de recuperación únicamente permite la eliminación de la información sobre los mensajes recibidos por el proceso que lo establece.

Estas preguntas son respondidas de forma diferente según el tipo de técnica e, incluso, según el algoritmo concreto utilizado; y ello ha motivado que se desestime la inclusión de la gestión del registro como parte del micro-núcleo.

3.3 La redundancia modular

Las técnicas de tolerancia a fallos basadas en la redundancia modular, cuando se implementan a nivel del soporte lógico, presentan dos aspectos básicos en común. El primero de ellos es el **procesamiento de los resultados** producidos por las diferentes instancias. En función de la técnica empleada (ver apartado 1.1.3.2) será necesario comparar los resultados, realizar una votación o, simplemente, seleccionar una de las salidas.

El segundo aspecto se refiere a la gestión de peticiones y resultados de los componentes lógicos implementados utilizando redundancia modular. Tal y como se describe en el capítulo anterior, es habitual implementar cada componente lógico

modular redundante mediante un grupo de procesos que se comunican entre sí y con el resto del sistema mediante paso de mensajes. Esta estructura necesita de un tipo de comunicación entre procesos conocida como **comunicación de grupo** [Tan95].

No obstante, antes de identificarlos como necesidades comunes al nivel del micro-núcleo, es necesario considerar la forma en que se manifiestan estos aspectos en sistemas reales. En la parte sombreada de la Ilustración 3-3,[5] en negrilla, se muestran estos aspectos comunes; mientras que en letra normal se recogen los problemas principales a la hora de incluirlos en el micro-núcleo, que se discuten a continuación.

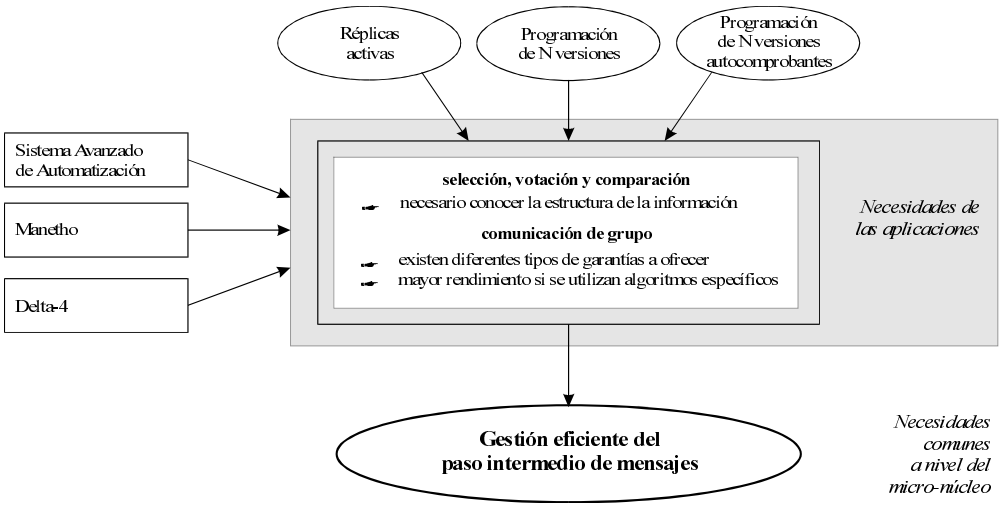

Ilustración 3-3: Necesidades comunes relativas al paso intermedio de mensajes.

Procesamiento de los resultados de las instancias

Aunque la selección es más sencilla, la forma adecuada de realizar la comparación y la votación puede ser diferente según los casos. Por un lado, en función de la garantía de funcionamiento que se desee alcanzar, se realizará una comprobación más o menos exhaustiva. Mientras en algunos casos puede ser necesario comparar completamente los resultados entre sí, en otros bastará con realizar una comparación parcial que permita mayores prestaciones. Por ejemplo, en Delta-4, al implementar réplicas activas, se ha utilizado la comparación de una firma (*signature*) para aumentar las prestaciones del sistema [CPR+92, p.32].

Por otro lado, existe la posibilidad de que los resultados no sean exactamente iguales (por ejemplo, debido a diferencias en el redondeo al realizar operaciones en coma flotante), especialmente si se emplea diversidad funcional [LA90, pp. 227-230].

[5] La implementación de réplicas pasivas, cuando éstas mantienen un registro de mensajes, también participa de la necesidad de la comunicación de grupo. No obstante, se ha preferido no incluirla como técnica en la ilustración porque no comparte la necesidad de procesar la salida de las instancias mediante selección, comparación o votación.

En este caso, la identificación de resultados diferentes, aunque igualmente válidos, debe hacerse siguiendo los criterios específicos de la aplicación.

Por todo ello, se ha desestimado incluir el procesamiento de los resultados proporcionados por las diferentes instancias como parte del núcleo, ya que no se dispondría de la información adecuada para procesar los resultados de forma eficiente.

Comunicación de grupo

Los diferentes sistemas recogidos en los casos de ejemplo (apartado 1.3) han seguido enfoques muy diferentes a la hora de implementar la comunicación de grupo, en función de sus requisitos específicos.

Por ejemplo, en Delta-4 se han incluido tres tipos diferentes de replicación de componentes, que presentan diferentes requisitos sobre la comunicación de grupo. Así, las réplicas pasivas requieren únicamente que el envío de mensajes sea fiable (en el sentido de que, si el emisor está libre de errores, el mensaje debe ser recibido por todos los receptores libres de errores) y que los mensajes sean recibidos por todas las réplicas en el mismo orden en el que los envía el emisor (no es necesaria ninguna garantía de orden en la transmisión de mensajes de emisores diferentes) [SB89, p.186]. En cambio, las réplicas activas sí necesitan que se garantice que todas las instancias recibirán los mensajes en el mismo orden, independientemente de si provienen del mismo emisor o de emisores diferentes [CPR+92, p.31]. Finalmente, las réplicas semi-activas [Pow94, p. 43], basadas en el modelo líder-seguidor, tampoco necesitan garantía de orden, aunque sí necesitan que las réplicas seguidoras consuman los mensajes en el orden preciso dictado por réplica líder.

De forma similar, el Sistema Avanzado de Automatización, descrito en 1.3.2 implementa diferentes tipos de réplicas y utiliza diferentes protocolos para la comunicación de grupo, con el objetivo de aumentar las prestaciones del sistema.

Pero, de entre los sistemas de ejemplo, Manetho destaca por incluir un nuevo algoritmo especialmente diseñado para la coordinación de procesos replicados, que consigue mejores prestaciones al aprovechar al máximo las particularidades del sistema, comparándose favorablemente en productividad (*throughput*) y latencia [Eln93, p. 76-81] con Amoeba [Tan95, p. 405], un sistema operativo distribuido basado en micro-núcleo que incorpora un algoritmo de difusión fiable y con garantía de orden.

Como se ha visto, las necesidades reales son muy diversas. Si se incluyese el soporte para la difusión fiable y con garantía de orden en el micro-núcleo, entonces, aquellas técnicas que no necesitan estas garantías (como las réplicas pasivas de Delta-4) se verían obligadas a soportar una sobrecarga que no les aporta ningún beneficio a cambio. Y, aunque se incluyese un conjunto reducido de protocolos con diferentes garantías, Manetho demuestra que sería posible desarrollar algoritmos específicos que podrían obtener mejores prestaciones.

A raíz de la explicación anterior, se ha decidido no incluir la comunicación de grupo como parte del núcleo, tal y como se hizo con el procesamiento de los resultados proporcionados por las diferentes instancias. Por lo tanto, estas actividades deberán ser realizadas por niveles superiores del sistema, bien utilizando procesos intermedios, bien incluyéndolas en el código de librería de los procesos de la aplicación. En ambos casos, aparece la necesidad común de **gestionar eficientemente el paso intermedio de mensajes**, puesto que los mensajes deberán pasar por agentes intermedios (sean estos procesos o librerías) para la selección, comparación o votación. Por lo tanto, los servicios para el paso de mensajes que se incluyan en el micro-núcleo deben facilitar, dentro de lo posible, este tipo de gestión.

3.4 La detección y confinamiento de errores

A la hora de evaluar las necesidades a nivel del micro-núcleo para la detección y confinamiento de errores, es necesario tener en cuenta aspectos bastante diferentes entre sí. Por una parte, las modificaciones necesarias para utilizar los mecanismos de detección incluidos en el soporte físico. Por otra, si es conveniente incluir nuevos servicios en el micro-núcleo para facilitar la implementación de las técnicas de detección de errores, o para detectar errores de las aplicaciones, del propio micro-núcleo y del soporte físico. Y, finalmente, la posibilidad de garantizar el comportamiento de silencio tras la avería de los nodos del sistema.

Técnicas al nivel del soporte físico

En el apartado 1.1.1 se han resumido las numerosas medidas que suelen incluirse en el soporte físico de todos los sistemas —no sólo de los tolerantes a fallos— para detectar y confinar los errores. Todos estos mecanismos tienen en común que, o bien enmascaran los fallos, o bien generan excepciones o interrupciones, que son manejadas por el núcleo del sistema operativo. Por lo tanto, para aprovechar la capacidad de detección de errores de estas medidas, no es necesario añadir nuevos servicios al micro-núcleo.

Tampoco resulta necesario cuando se utilizan técnicas de detección de errores mediante redundancia modular a nivel del soporte físico, o si se utilizan dispositivos sencillos para la detección de errores, como temporizadores de guardia (*watchdogs*).

No obstante, los procesadores de guardia basados en el comportamiento (*behavior based*) [MS92][MS94] sí necesitarían de este soporte por parte del núcleo del sistema operativo para poder ser utilizados en un entorno multitarea. No obstante, no se han encontrado ejemplos de utilización de estos dispositivos en este tipo de entornos y, aunque los resultados que se recogen en las referencias anteriores muestran que puede ser una forma efectiva de mejorar la detección de errores, la forma de implementar estas técnicas varían enormemente en cada caso, por lo que se ha desestimado la inclusión de soporte alguno en el micro-núcleo.

Además, los resultados obtenidos en [MS94] destacan la capacidad de los mecanismos de detección de errores incluidos en los procesadores para detectar un

porcentaje significativo de los errores del sistema (en el caso de estudio, con un procesador 68000, se mantenía el modelo de silencio tras la avería en un 90% de los casos). Especialmente efectivos resultan las comprobaciones realizadas en las unidades de gestión de memoria (*memory management units*) durante la traducción de direcciones [MS94][SCM+96].

Técnicas en los niveles de utilidades y aplicaciones

Por otra parte, las aplicaciones tolerantes a fallos, en función de las técnicas empleadas, incluirán sus propias medidas de detección de errores, tal y como se ha descrito en el capítulo anterior. En el caso de la redundancia modular, los errores se detectan al comparar los resultados producidos por las diferentes instancias. Esta comparación, como ya se ha discutido, para que sea eficiente, debe ser llevada a cabo por la propia aplicación, en función de la garantía de funcionamiento requerida y de la naturaleza de los resultados.

En el caso de la recuperación hacia atrás, las medidas para detectar errores al nivel de la aplicación, adoptan formas muy diversas, tal y como se recoge en el apartado 1.1.1. Las medidas para detectar errores en los datos dependerán de nuevo de la naturaleza de los mismos, por lo que también deberán incluirse como parte de la aplicación. Las comprobaciones de temporización, por otro lado, pueden ser realizadas por componentes externos a la aplicación; pero resulta más adecuado realizarlas al nivel de las utilidades, ya que incluirlas en el del núcleo del sistema operativo no aporta ninguna ventaja adicional.

Medidas de detección en el micro-núcleo

El micro-núcleo, como cualquier otro componente del sistema, debe incluir los mecanismos adecuados para la detección de errores en el resto de componentes con los que interacciona. Esta técnica es necesaria en todos los sistemas, no sólo los tolerantes a fallos, ya que el núcleo del sistema operativo debe protegerse de los intentos de acceso incorrecto por parte de sus usuarios. En sistemas tolerantes a fallos, el micro-núcleo podría incluir servicios para permitir que los procesos pudieran aumentar la detección de errores llevada a cabo por el micro-núcleo. No obstante, para evitar que la sobrecarga debida a estas comprobaciones adicionales afecte a todos los procesos del sistema, sin tener en cuenta la garantía de funcionamiento que necesitan, dicho aumento debería ser selectivo, es decir, aplicable únicamente al proceso que lo solicita.

Tal y como se ha destacado al hablar de la detección de errores en el soporte físico, las comprobaciones realizadas por el soporte físico durante la traducción de direcciones resultan sumamente efectivas para la detección de errores, que podrían verse aumentadas si se ofreciese al usuario la oportunidad de reducir las operaciones que puede realizar con cada zona de su espacio de direcciones (por ejemplo, la posibilidad de convertir una zona de su espacio de direcciones en zona de sólo lectura). Estos servicios tienen la ventaja de no suponer sobrecarga alguna para las aplicaciones (salvo durante el cambio de protección), por lo que resultan especialmente adecuados para ser incluidos en el micro-núcleo.

Además de detectar errores en sus usuarios, el micro-núcleo debe detectar sus propios errores y los de los componentes de los que depende —principalmente

componentes físicos (procesador, memoria, ...); pero también componentes lógicos como el servidor de comunicación entre nodos o el servidor de intercambio (*swap*)—. En la mayor parte de los sistemas, el núcleo del sistema operativo aprovecha la redundancia en sus estructuras de datos para detectar este tipo de errores; pero este tipo de comprobaciones se llevan a cabo independientemente de las aplicaciones que se estén ejecutando sobre el sistema. Si el incremento en la garantía de funcionamiento no es necesario para las aplicaciones en ejecución, éstas estarán soportando una sobrecarga que no les ofrece ninguna contrapartida, por lo que debe ser posible ajustar el coste de las comprobaciones en función de las aplicaciones. (En este caso, la sobrecarga es común para todas las aplicaciones, por lo que este tipo de comprobaciones no resultará adecuado para sistemas con varias aplicaciones con necesidades diferentes).

Finalmente, la capacidad del micro-núcleo para generar la información para los registros puede ser aprovechada también como herramienta para la detección de errores o la depuración, ya que su interacción con el micro-núcleo es una buena traza del comportamiento del proceso. Por ejemplo, en [Rus94, pp. 17-20] se describe cómo la política de supervisión para el mecanismo Sentry permitía la depuración y la perfiladura (*profiling*).

Comportamiento tras la avería

Tal y como se recoge en el apartado 2.3.2, una de las ventajas de la arquitectura propuesta es la posibilidad de confinar los errores en el nodo, de forma que éste garantice el silencio tras la avería. De esta forma, es posible utilizar técnicas de tolerancia a fallos más eficientes y sencillas. Por ejemplo, sería posible utilizar réplicas pasivas en lugar de réplicas activas, o utilizar técnicas basadas en puntos de recuperación.

Tradicionalmente se ha considerado que este modelo (el de silencio tras la avería) se viola cuando un nodo emite datos incorrectos; pero se puede relajar esta condición exigiendo que los datos erróneos no sean aceptados como válidos en ninguno de los otros nodos del sistema. Es más, sería suficiente con garantizar que los datos no son aceptados como correctos al nivel de la aplicación a la que van destinados, ni tampoco son aceptados por ninguna otra aplicación (en el caso de que el error hiciera que los datos fueran enviados a un destino incorrecto). Así, un mensaje entre núcleos de diferentes nodos debería ser rechazado (en caso de ser erróneo) a nivel de los micro-núcleos de destino. En cambio, un mensaje incorrecto generado por un proceso de una aplicación debería ser rechazado al nivel del proceso (o uno inferior) al que va destinado (o al que ha llegado por error). La razón de todo ello es que en el nivel en el que se generan los mensajes pueden introducirse errores que difícilmente podrán ser detectados por niveles inferiores del sistema, que no dispondrán, en general, de la información necesaria para detectarlos.

Por supuesto, si un fallo en uno de los nodos colapsa totalmente el medio de comunicación entre nodos, el sistema completo sufrirá una avería, por lo que esta eventualidad debe ser considerada.

Debido a que todos los niveles se hallan involucrados en la detección de errores, y dado que las diferentes técnicas se solapan unas con otras, serían necesarios estudios

empíricos mediante inyección de fallos sobre aplicaciones concretas, que mostraran hasta qué punto los mecanismos estándar de detección de errores resultan suficientes para garantizar el modelo de parada ante la avería.

En la bibliografía consultada no se han encontrado trabajos con estas suposiciones iniciales, salvo [SCM+96], en el que se estudian una serie de aplicaciones típicas de cálculo intensivo. El artículo recoge dos aspectos interesantes. El primero, que en un número significativo de casos (entre un 4.9 y un 12.7 %), las aplicaciones terminaban produciendo resultados incorrectos sin que se hubiera detectado ningún error. Y el segundo, que la efectividad de las técnicas de detección de errores depende enormemente de las aplicaciones.

Los resultados no son directamente extrapolables a aplicaciones tolerantes a fallos que, como ya se ha descrito, incorporan técnicas adicionales para la detección de errores y donde cabe esperar un porcentaje bastante menor de errores no detectados; pero refuerzan la hipótesis de que son necesarios estudios sobre aplicaciones concretas.

3.5 Conclusiones

En este capítulo se ha identificado la función que debe desempeñar el micro-núcleo en el desarrollo de aplicaciones tolerantes a fallos. Para ello, se han identificado las necesidades comunes de las técnicas de tolerancia a fallos revisadas en el apartado 1.1, donde se describen resaltando sus similitudes, así como las diferencias que aparecen a la hora de aplicarlas en los sistemas reales, representados por los sistemas descritos en ese mismo capítulo. La Ilustración 3-4 recoge, de forma resumida, las funciones que debe desempeñar el micro-núcleo. Además, el análisis realizado en este capítulo también permite extraer conclusiones sobre las funciones que deben ser desempeñadas por niveles superiores del sistema, y no por el micro-núcleo.

A continuación se extraen estas conclusiones para cada uno de los apartados que se muestran en la ilustración:

1. Respecto al estado de los procesos, el micro-núcleo debe ofrecer servicios para permitir que los procesos guarden, recuperen y actualicen su estado de forma selectiva y eficiente. No obstante, el micro-núcleo no debe establecer a priori qué parte del estado se va a guardar, dónde, ni cuándo.
2. El micro-núcleo debe generar, cuando los procesos lo soliciten, la información que debe ser almacenada en los registros, permitiendo que seleccionen el tipo de información, y también si debe ser generada de forma síncrona o asíncrona. En cambio, el micro-núcleo no debe incluir la gestión de los registros, que debe ser llevada a cabo en el nivel de utilidades.
3. La redundancia modular presenta dos tipos de necesidades: el procesamiento de los resultados de las instancias y la comunicación de grupo. Ninguna de las dos funciones debe ser desempeñada por el micro-núcleo, en cambio, éste debe ser diseñado de forma que el paso intermedio de mensajes sea eficiente —tanto a

través de procesos intermedios como a través del código de la librería—, aportando, si es necesario, servicios específicos para ello.

4. Respecto a la detección y el confinamiento de errores, el micro-núcleo debe detectar sus propios errores y los de aquellos componentes de los que depende, tanto físicos como lógicos. Además, debe detectar, en la interfaz, errores en el resto de los componentes con los que interacciona. Estos requisitos no son específicos de un micro-núcleo para el desarrollo de aplicaciones tolerantes a fallos; sin embargo, un micro-núcleo diseñado para este fin, debe proporcionar diferentes grados de detección. Debido al coste asociado a la detección, los procesos deben poder seleccionar, individualmente, el grado de detección que necesitan, para que se ajuste a sus requisitos de garantía de funcionamiento. En cambio, el micro-núcleo no debe gestionar dispositivos físicos específicos para la detección de errores, ni debe incluir servicios para la detección de errores en componentes lógicos de niveles superiores.[6]

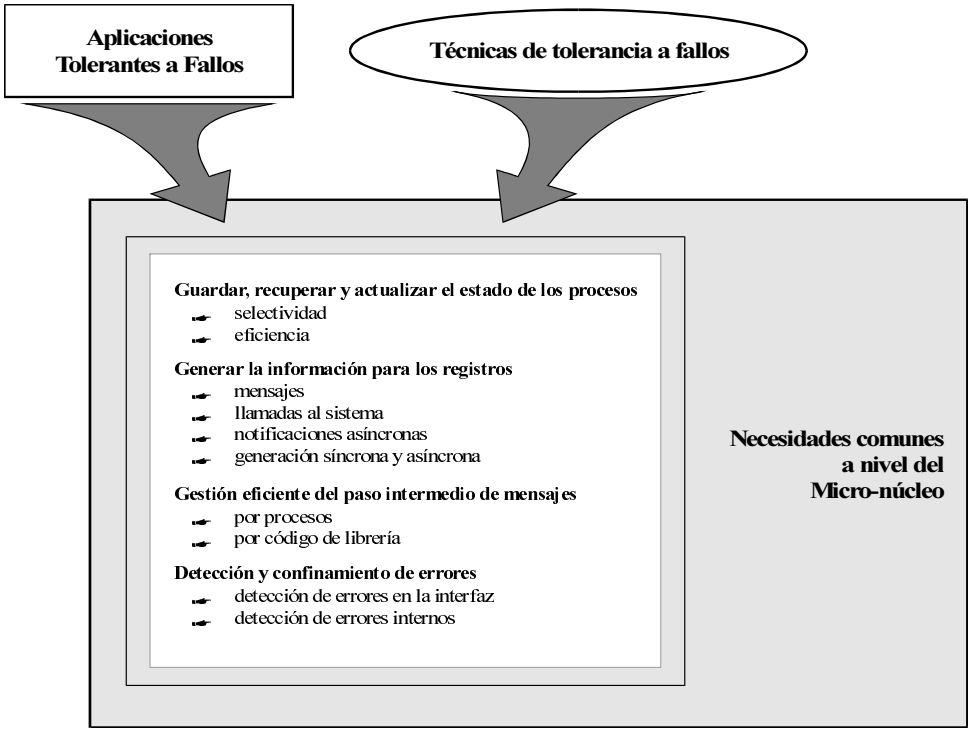

Ilustración 3-4: Resumen de las necesidades comunes a nivel del micro-núcleo.

[6] Nótese que la detección de errores a través de la interfaz sí debe ser desempeñada por el micro-núcleo. (Véase el punto "Técnicas en los niveles de utilidades y aplicaciones" en la pág. 59).

Capítulo 4

Especificación del micro-núcleo

En este capítulo se describe la propuesta de interfaz para el micro-núcleo, es decir, el conjunto de servicios que deben incluirse en él para formar un conjunto completo, eficiente y versátil de herramientas para el desarrollo de aplicaciones tolerantes a fallos. Por lo tanto, en este capítulo no se pretende especificar una interfaz completa, sino un conjunto de servicios que podría ser incluido en la interfaz de un micro-núcleo genérico, cuyas características básicas se han descrito en el apartado 2.4.

Para obtener el conjunto de servicios que se proponen en este apartado, se han seguido dos enfoques complementarios. En primer lugar, la presente propuesta recoge algunos servicios que existían previamente y es conveniente resaltar que, para seleccionar los servicios a incluir en el micro-núcleo, se ha tenido en cuenta que: 1) cumpliesen los requisitos descritos en el capítulo 2 para ser incluidos en el micro-núcleo (versatilidad y eficiencia) y 2) contribuyesen a satisfacer las necesidades que se desea cubrir. En cambio, no se ha tenido en cuenta qué estructura tiene el sistema operativo en el que el servicio estaba incluido. Por ejemplo, *mprotect()*, que se nombra a continuación, forma parte de la definición SVID3, publicada por AT&T y que especifica la funcionalidad que debe ofrecer un sistema Unix para que pueda ser calificado como SVR4. Estos servicios provienen de dos fuentes diferentes. Por una parte, se han considerado los servicios que han sido incluidos en otros núcleos de sistema operativo como soporte para la implementación de una determinada técnica de

tolerancia a fallos. Éste es el caso de Manetho (ver apartado 1.3.3), cuya implementación incluye modificaciones al núcleo del sistema operativo V como el soporte para el establecimiento de puntos de recuperación. Por otra parte, se han considerado servicios como *mprotect()*, que forman parte del núcleo de sistemas operativos no diseñados específicamente para aplicaciones tolerantes a fallos; pero que, sin embargo, resultan útiles para satisfacer algunas de las necesidades de las aplicaciones tolerantes a fallos.

En segundo lugar, como resultado de la investigación, se ha observado que los servicios existentes no cubrían de forma satisfactoria algunas de las necesidades de las aplicaciones, sobre todo en cuanto al almacenamiento, actualización y recuperación del estado de los procesos de forma selectiva y eficiente. En este sentido, se han propuesto una serie de nuevos servicios que ayudan a satisfacerlas, permitiendo que el conjunto de herramientas incluidas en el micro-núcleo sea, efectivamente, completo, tal y como se establece en el capítulo anterior.

El presente capítulo se ha estructurado como sigue: el apartado 4.1 introduce los nuevos servicios, destinados a completar la interfaz del micro-núcleo, que se describe en el apartado 4.2. Y, por último, el apartado 4.3 describe la interacción del micro-núcleo con los niveles superiores del sistema.

4.1 Las operaciones incrementales

Uno de los aspectos más relevantes del soporte para aplicaciones tolerantes a fallos consiste en facilitar la gestión eficiente de las copias de los datos críticos de los procesos que forman parte de estas aplicaciones. A nivel del micro-núcleo, el reto consiste en ofrecer servicios que incorporen la eficiencia de los puntos de recuperación incrementales y concurrentes, al mismo tiempo que se deja en manos de los niveles superiores del sistema las decisiones sobre qué datos deben ser incluidos o cuándo deben ser establecidos.

Los servicios existentes, o bien se apoyan en el núcleo del sistema operativo para aumentar la eficiencia utilizando las técnicas anteriores (puntos de recuperación incrementales y concurrentes) [Eln93], o bien se implementan como una librería que almacena únicamente la información crítica para la aplicación [HK93].[1] Ambos enfoques han sido discutidos al hablar de la eficiencia en el apartado 1.2.3.

Con el propósito de combinar las ventajas de ambos enfoques, se han diseñado las operaciones que se describen a continuación, y que han sido denominadas operaciones incrementales [PF97]. Mediante este conjunto de operaciones, se ofrece a los niveles superiores un mayor control sobre su espacio virtual de direcciones y también sobre el de otros procesos. Al mismo tiempo, al apoyarse en la unidad de gestión de memoria (MMU) del procesador, se facilita la gestión eficiente de información replicada. Por

[1] Una propuesta diferente aparece en [PBKL95]. La propuesta consiste en utilizar los servicios *fork()* y *mprotect()* para establecer puntos de recuperación incrementales y concurrentes desde el espacio de usuario (la librería *Libckpt*); pero la finalidad sigue siendo establecer puntos de recuperación que contengan el estado completo del proceso.

último, y no menos importante, al ser compatibles con el modelo de comunicación entre procesos mediante paso de mensajes, se mantiene la compatibilidad con las numerosas técnicas de tolerancia a fallos que existen para este tipo de sistemas.

4.1.1 Mensajes incrementales

Un **mensaje incremental** (MI) se diferencia de los mensajes tradicionales en dos aspectos básicos, que se describen a continuación. No obstante, como es usual en el paso de mensajes, contiene información que debe ser transferida desde el espacio de direcciones del emisor al espacio de direcciones del receptor.

- En primer lugar, no se transmite toda la información contenida en la zona de memoria virtual especificada por el emisor, sino que se transmiten, solamente, aquellas páginas que han sido modificadas desde el mensaje incremental anterior.
- En segundo lugar, el destino del mensaje es una zona del espacio de direcciones del receptor que contiene una copia de los datos del emisor.

En la Ilustración 4-1 se muestra el efecto que produce la transmisión de un mensaje incremental sobre los espacios de direcciones de los procesos emisor (P_1) y receptor (P_2).

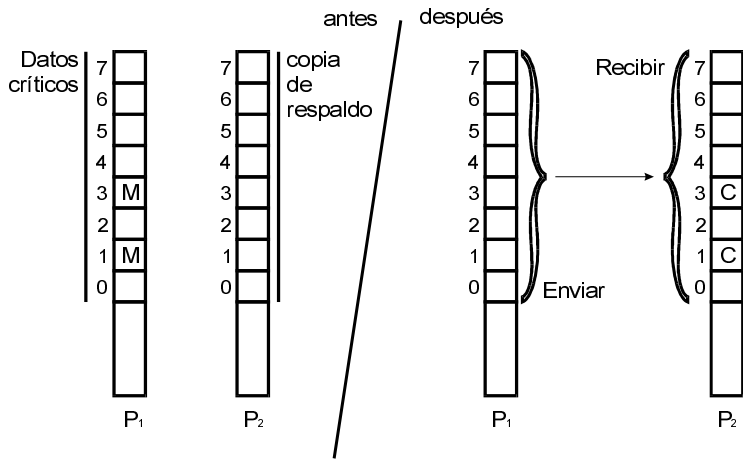

Ilustración 4-1: Mensajes incrementales.

La parte izquierda de la figura muestra la situación de ambos espacios de direcciones antes de que se transmita el mensaje incremental. P_2 mantiene una copia de respaldo de los datos críticos sobre los que está trabajando P_1 (las páginas numeradas de 0 a 7). Cuando P_1 considera que las modificaciones realizadas deben ser hechas permanentes, envía un mensaje incremental a P_2. En la ilustración se muestra un caso en el que P_1 ha modificado las páginas 1 y 3 desde que envió el mensaje incremental anterior. Al recibir P_2 el mensaje incremental, el efecto es el siguiente:

- Los datos de las páginas 1 y 3 son copiados desde el espacio de direcciones de P_1 al de P_2, reemplazando a los datos que ya existían (y que habían quedado desfasados).
- En el origen, las páginas 1 y 3 dejan de ser consideradas como modificadas a efectos de la transmisión de mensajes y no serán incluidas en mensajes incrementales sucesivos hasta que vuelvan a ser modificadas.

Como puede verse, la finalidad de los mensajes incrementales es la de permitir que se actualicen, de forma sencilla y eficiente, las copias de la información crítica para el sistema que se mantienen para aumentar la garantía de funcionamiento. La actualización es sencilla, porque simplemente requiere que el proceso con la información original envíe un mensaje incremental al proceso o procesos que mantengan las copias de la información. Y es eficiente, porque únicamente se transmiten las páginas que han sido modificadas desde la actualización anterior.

La implementación del paso de mensajes incrementales puede utilizar la copia en escritura como medio para mejorar las prestaciones. Si se utiliza, las operaciones incrementales aportan al usuario la ventaja de los puntos de recuperación concurrentes, es decir, un menor tiempo de latencia a la hora de actualizar las copias de la información. (Ver el estudio de las prestaciones que se recoge en el capítulo 6).

4.1.2 Mensajes incrementales empaquetados

Tanto los mensajes incrementales empaquetados, descritos en el presente apartado, como los mensajes incrementales negativos, que se describen en el apartado siguiente, son servicios que se basan en la misma idea básica de los mensajes incrementales y han sido diseñados para aumentar la versatilidad de tipo de mensajes.

Un **mensaje incremental empaquetado** (MIE) es una representación condensada de la información contenida en un mensaje incremental. Además de las páginas que deben ser transferidas, el MIE contiene una estructura de datos, a la que denominaremos **mapa**, que describe el contenido del mensaje. Puesto que un mensaje incremental contiene la información necesaria para actualizar una copia, en el mapa se indica el lugar que ocupaba cada una de las páginas en la zona de memoria original, descrita en términos de la dirección virtual de comienzo y su longitud.

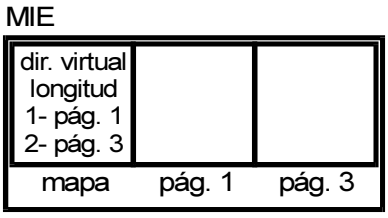

Ilustración 4-2: Mensaje incremental empaquetado (MIE).

La finalidad de los mensajes incrementales empaquetados es la de permitir que los mensajes incrementales sean procesados por procesos que no mantienen la copia de los datos del emisor en su espacio de direcciones, bien porque son meros agentes intermedios entre el emisor y el receptor (ver el apartado 4.3.2), bien porque la copia se mantiene en un dispositivo de almacenamiento secundario.

La conversión entre mensajes incrementales y mensajes incrementales empaquetados se lleva a cabo como parte del intercambio de mensajes entre procesos. Para controlarla, los procesos utilizan la opción 'empaquetado', que pueden incluir en las operaciones enviar y recibir. En la Ilustración 4-3 se muestran los efectos sobre los espacios de direcciones del emisor y del receptor para los tres casos en que intervienen mensajes incrementales empaquetados, y la Tabla 4-1 resume el significado de las cuatro posibles combinaciones de enviar y recibir con y sin la opción de 'empaquetado'.

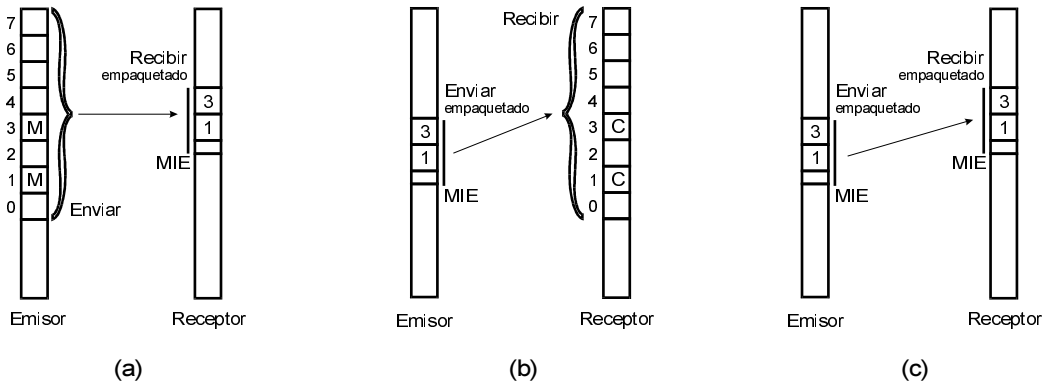

Ilustración 4-3: Transferencia de mensajes incrementales empaquetados.

La Ilustración 4-3 (a) muestra el caso en el que el emisor envía un mensaje incremental con las modificaciones que ha realizado sobre los datos críticos (páginas 1 y 3). El receptor, en este caso, utiliza la opción 'empaquetado' en recibir y ello hace que obtenga un mensaje incremental empaquetado. A diferencia de la recepción de un mensaje incremental, la recepción de un mensaje incremental empaquetado no modifica las páginas existentes en el espacio de direcciones del receptor, sino que el MIE se hace accesible al receptor mediante su proyección en una nueva zona de memoria virtual, que permite el acceso, utilizando direcciones virtuales consecutivas, tanto al mapa como a las páginas incluidas en el mensaje.

El caso que se recoge en la Ilustración 4-3 (b) es el inverso. El emisor utiliza la opción 'empaquetado' al enviar el mensaje incremental. Ello hace que el micro-núcleo interprete la zona de memoria especificada por el proceso emisor como un mensaje incremental empaquetado. Puesto que el receptor no utiliza la opción 'empaquetado' en recibir, se actualiza la copia de los datos que mantiene el receptor, utilizando la información del mapa y las páginas incluidas en el mensaje incremental empaquetado.

Finalmente, la Ilustración 4-3 (b) muestra el caso en el que tanto el emisor como el receptor utilizan la opción 'empaquetado'. En este caso, el micro-núcleo se limita a copiar el contenido del mensaje desde el espacio de direcciones del emisor al espacio de direcciones del receptor, como haría con un mensaje convencional.

Enviar	Recibir	Efecto
		Transferencia de un MI, tal y como se describe en el apartado 4.1.1
	empaquetado	El MI generado por enviar es almacenado en el espacio de direcciones del receptor en forma de MIE. (Ilustración 4-3, a)
empaquetado		El conjunto de datos incluidos en el mensaje es interpretado por el núcleo como un MI, que es transferido como tal al espacio de direcciones del receptor. (Ilustración 4-3, b)
empaquetado	empaquetado	Se transfiere un mensaje tradicional. Aunque el contenido es un MIE, el resultado es que una zona contigua de datos del emisor es transferida a una zona contigua de datos del receptor. (Ilustración 4-3, c)

Tabla 4-1: Posibilidades en la transferencia de mensajes incrementales y mensajes incrementales empaquetados.

4.1.3 Mensaje incremental negativo

Los mensajes incrementales negativos tienen como finalidad permitir que un proceso pueda desechar sencilla y eficientemente el trabajo realizado desde que envió el último mensaje incremental, para realizar una vuelta atrás, por ejemplo.

Para describir su funcionamiento, se va suponer que se parte de la misma situación inicial que se recogía en la Ilustración 4-1, es decir, que el proceso P_1 tiene la información original en su espacio de direcciones y que P_2 guarda una copia de respaldo de los datos críticos de P_1. En este caso, sin embargo, P_1 desea deshacer los cambios que ha realizado en los datos, y que han afectado a las páginas 1 y 3, tal y como se muestra en la Ilustración 4-4. Para desecharlos son necesario dos mensajes.

El primero lo debe enviar P_1 a P_2. P_1 incluye la opción 'negativo' en la llamada a enviar, para indicarle al micro-núcleo que el mensaje es negativo. Así, el núcleo no incluye las páginas modificadas en el mensaje, sino que incluye una descripción de dichas páginas. Cuando P_2 recibe el mensaje, identifica el mensaje como negativo e interpreta su contenido como un mapa de las páginas que P_1 necesita.

El segundo mensaje es la respuesta de P_2 a P_1. P_2 utiliza la opción 'mapa' en enviar, para indicar que el micro-núcleo debe transferir precisamente las páginas descritas en el mapa (y no las que P_2 pudiera haber modificado). Al recibir P_1 este segundo mensaje, las páginas 1 y 3, que habían sido modificadas son reemplazadas por una copia de las que mantenía P_2 en su espacio de direcciones, por lo que el efecto

conjunto de ambos mensajes ha sido, tal y como se pretendía, que los cambios realizados por P_1 han sido desechados.[2]

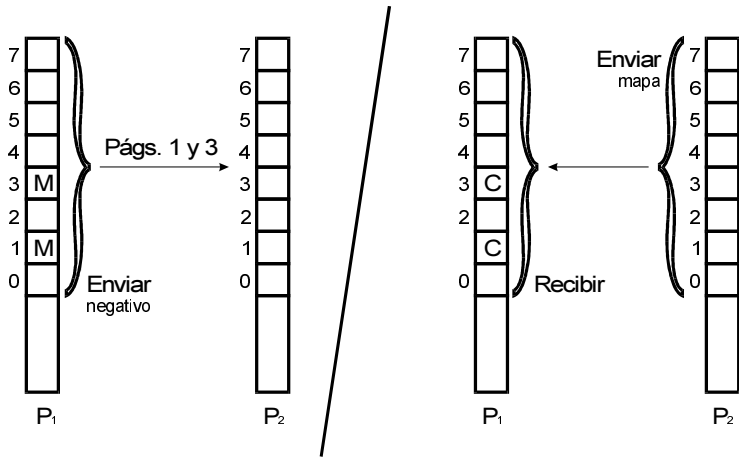

Ilustración 4-4: Mensajes incrementales negativos.

4.1.4 Recepción de la cabecera

Aunque los mensajes incrementales aparecen con la finalidad de facilitar la gestión de información replicada, no debe ignorarse el resto de necesidades de las aplicaciones tolerantes a fallos. En concreto, una de las necesidades identificadas en el capítulo anterior es la de gestionar eficientemente el paso intermedio de mensajes, tanto por procesos intermedios como por el código de librería, para permitir operaciones usuales al emplear redundancia modular, como la votación, la selección o la reordenación de mensajes.

El código de librería puede llevar a cabo, por ejemplo, la reordenación de los mensajes, manteniendo una cola privada de mensajes recibidos. Una vez que un mensaje está preparado para ser consumido por la aplicación (se ha acordado el orden con otras réplicas, por ejemplo), es trasladado a la cola desde donde son consumidos por el código de la aplicación (en el orden establecido por el código de librería, en el ejemplo). Esta situación se muestra en la Ilustración 4-5. En cambio, con los mensajes incrementales, esta estrategia no puede ser utilizada, ya que la copia de los datos resulta actualizada por el mero hecho de recibir el mensaje incremental (independientemente de que sea el código de la librería o de la aplicación quien invoque al servicio recibir, ya que ambos comparten el mismo espacio de direcciones)

[2] Aunque, tal y como se recoge en este ejemplo, en general son necesarios dos mensajes para desechar los cambios, cuando el micro-núcleo proporciona el servicio para enviar un mensaje al servidor y esperar su respuesta, es posible que un cliente deseche sus cambios mediante una sola interacción con el servidor, tal y como se describe en el anexo IV.

y es, precisamente, el orden de las actualizaciones (en este ejemplo) lo que el código de librería debería poder establecer.

Además, aunque se utilicen procesos intermedios, la copia de los datos al espacio de direcciones del receptor (incluso utilizando copia en escritura) es una operación costosa, que puede resultar inútil si el proceso no necesita consultar el contenido del mensaje. Por ejemplo, porque simplemente tiene que comprobar qué páginas han sido incluidas en el mensaje, o porque puede reordenar los mensajes basándose en algún tipo de información de cabecera, como un número de orden.

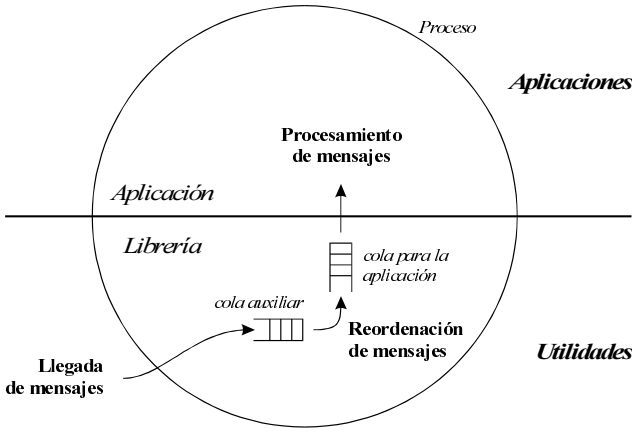

Ilustración 4-5: Reordenación de mensajes mediante código de librería.[3]

Para resolver este problema, y atender a la necesidad anteriormente reseñada, se ha incluido la posibilidad de que el receptor indique que solamente desea recibir la cabecera del mensaje. Esta cabecera puede ser diferente en cada implementación, pero debe contener la siguiente información:

- la información en línea del mensaje[4],
- la información sobre qué páginas contiene —se considera que esta información, que se corresponde con la que se incluye en el mapa de un mensaje incremental empaquetado, es información fuera de línea—.

No obstante, las páginas incluidas en el mensaje incremental no deben ser destruidas, sino que, por el contrario, el proceso debe poder indicar cuándo desea

[3] Esta ilustración está basada en la ilustración 11.9., pág. 322 de [CDK96].

[4] Los términos información (o datos) en línea (i*n-line data*) y fuera de línea (*out-of-line data*) se utilizan en el sistema operativo Mach [Dra90], y se utilizan aquí para expresar los mismos conceptos. La información en línea es la que se copia del espacio de direcciones del emisor al núcleo cuando envía el mensaje y de allí al espacio de direcciones del receptor cuando éste recibe el mensaje. En cambio, la información fuera de línea se transfiere utilizando técnicas de gestión de memoria, como copia en escritura. En este sentido, las páginas incluidas en el mensaje se consideran como información fuera de línea.

acceder a ellas. Para permitirlo, se asocia una cola a cada destino potencial de mensajes[5]: la **cola auxiliar de mensajes**. Así, cuando un proceso utiliza el servicio recibir con la opción 'cabecera', el proceso recibe la información en línea, la descripción de las páginas incluidas en el mensaje, y un identificador que puede utilizar para referirse a la información fuera de línea, que pasa a la cola auxiliar del puerto.

La Ilustración 4-6 muestra una representación esquemática de un puerto junto con su cola auxiliar, y también las operaciones disponibles para su gestión. Como puede observarse, un mensaje que es recibido con la opción 'cabecera' pasa a la cola auxiliar. Una vez allí, el proceso puede utilizar el identificador devuelto por recibir para procesar los mensajes en el orden que desee. El servicio proyectar permite completar la recepción del mensaje, el servicio reenviar permite enviar el mensaje a otro puerto sin necesidad completar la recepción y, por último, el servicio eliminar libera el espacio ocupado por el mensaje.

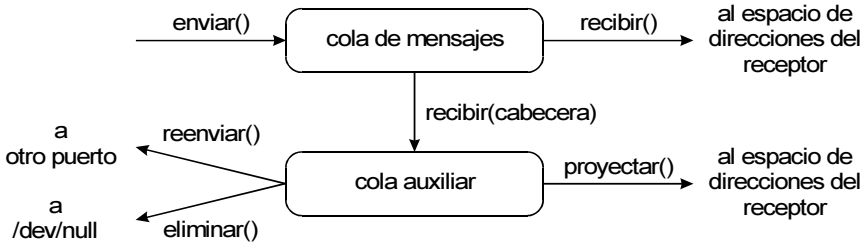

Ilustración 4-6: Operaciones para la gestión de la cola auxiliar.

Además de facilitar el paso intermedio de mensajes, estos servicios resultan adecuados como herramienta para la detección y confinamiento de errores. En efecto, el receptor de un mensaje incremental puede utilizar la opción 'cabecera' al recibir el mensaje para consultar qué modificaciones se van a realizar a su espacio de direcciones antes de que se lleven a cabo. Por lo tanto, es una medida efectiva para permitir confinar los errores del emisor, que pueden ser detectados antes de que se propaguen al receptor del mensaje incremental. (Ver el apartado 4.3.3).

Aunque la recepción de la cabecera se ha propuesto a partir de los mensajes incrementales, este servicio resulta de utilidad para cualquier tipo de mensajes con información fuera de línea. Por ejemplo, sea un conjunto de réplicas activas cuyo código de librería se encarga de la coordinación necesaria para que todas ellas procesen en el mismo orden los mensajes que reciben. En este caso, la recepción de la cabecera permite que la asignación de direcciones virtuales a los datos fuera de línea se produzca en el mismo orden en todas las réplicas. De la misma forma, si se utiliza un proceso intermedio, con la recepción de la cabecera se evita la proyección de los datos en su espacio de direcciones, con lo que se obtienen dos ventajas: mayor eficiencia y

[5] Sea éste un puerto, un buzón o un proceso. En la exposición se utilizará el término puerto; pero igualmente podría ser un buzón o un proceso.

protección frente a errores del proceso intermedio, que no podrá acceder a los datos fuera de línea de los mensajes.

4.1.5 Gestión de varios destinos

Hasta el momento no se han considerado casos en los que existiera más de una copia de respaldo de los datos críticos. Aunque proteger los datos críticos frente a un único punto de fallo es una situación muy común, puede ser necesario mantener más de una copia de los datos para conseguir una mayor garantía de funcionamiento. En este caso, puede ser conveniente que cada copia de los datos sea actualizada con diferente frecuencia, tal y como se propone en el apartado 5.1.

Para ello, es necesario que el micro-núcleo mantenga información sobre las páginas modificadas desde que se actualizó cada una de las copias de los datos, es decir, es necesario que mantenga un registro de páginas modificadas por cada destino potencial de mensajes incrementales. En el anexo IV se describe el algoritmo empleado para la gestión de los registros.

Aunque la gestión de los registros se realice en el micro-núcleo, la gestión de las copias de los datos es responsabilidad de los niveles superiores del sistema. Por eso, todas las operaciones incrementales incluyen un parámetro que permite indicar el registro de páginas modificadas que se debe utilizar, y que se ha denominado “destino incremental”. Además, se incluye un servicio (ver el apartado 4.2.3) para que se puedan consultar y modificar los diferentes registros de páginas modificadas mantenidos por el micro-núcleo.

4.1.6 Copia incremental

La copia incremental es un servicio que permite actualizar la copia de los datos críticos cuando la copia de trabajo y la de respaldo residen en el mismo espacio de direcciones. En la Ilustración 4-7 se muestra el efecto de la copia incremental sobre ambas copias de los datos. El proceso P_0 ha modificado las páginas 1 y 3 de la copia de trabajo cuando decide actualizar la copia de respaldo mediante una operación de copia incremental. Como en el caso del mensaje incremental, las páginas 1 y 3 de la copia de respaldo son actualizadas con una copia de las páginas 1 y 3 de la copia de trabajo, que pierden su condición de modificadas para sucesivas copias incrementales.

La copia incremental requiere un cambio de modo usuario a modo supervisor, la manipulación de las estructuras de datos de memoria virtual y la vuelta a modo usuario, con el consiguiente coste temporal. No obstante, este servicio permite actualizar eficientemente grandes cantidades de datos, al transferir únicamente las páginas modificadas. Como en el caso de los mensajes incrementales, la eficiencia puede aumentarse aún más utilizando la técnica de copia en escritura para transferirlas.

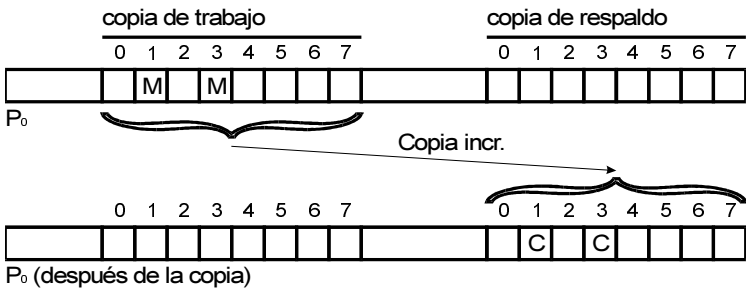

Ilustración 4-7: Copia incremental.

Al igual que se ha hecho con los mensajes incrementales, se ha definido la operación de copia incremental negativa, para permitir que el proceso pueda desechar de forma sencilla y eficiente el trabajo realizado desde la última copia incremental. La copia incremental negativa consiste, por lo tanto, en transferir desde la copia de respaldo a la copia de trabajo las páginas modificadas en la copia de trabajo. (Nótese que en la copia incremental se transfieren las páginas modificadas de la copia de trabajo a la copia de respaldo). Así, en el caso de la Ilustración 4-7, si P_0 realizase una copia incremental negativa en vez de una copia incremental, el resultado de la operación sería que las páginas 1 y 3 serían copiadas de la copia de respaldo a la de trabajo, dejando la copia de trabajo en el estado en el que estaba cuando se realizó la última copia incremental.

4.2 La interfaz del micro-núcleo

La descripción de los servicios del micro-núcleo se ha dividido en tres apartados: gestión de procesos, paso de mensajes y gestión de memoria virtual. Con ello se ha pretendido facilitar la exposición; pero no se quiere indicar, en modo alguno, que estos tres apartados resulten independientes entre sí. A modo de ejemplo, los servicios para establecer el punto de recuperación completo de un proceso, para enviar o recibir un mensaje incremental y para realizar una copia incremental comparten el mismo mecanismo básico —transferir al destino únicamente las páginas de memoria que han sido modificadas—, aunque hayan sido incluidos en la gestión de procesos, en el paso de mensajes y en la gestión de memoria virtual, respectivamente.

4.2.1 Gestión de procesos

La primera necesidad recogida en la Ilustración 3-4 es la de guardar y recuperar el estado de los procesos. Los servicios descritos en los apartados 4.2.2 y 4.2.3 actúan sobre zonas concretas del espacio de direcciones de cada proceso, mientras que los servicios que se describen en este apartado operan sobre el estado completo de un proceso, resultando complementarios para conseguir los objetivos de selectividad y

eficiencia. A continuación se describen estos servicios, que se resumen en la Ilustración 4-8 y en la Ilustración 4-9.

EstablecerDestinoPdR(proceso, puerto)

El primero de los servicios, *EstablecerDestinoPdR()*, permite especificar adónde deben ser enviados los datos relativos al estado del proceso cuando se establezca un punto de recuperación. El primer argumento, *proceso*, indica el proceso cuyo destino se especifica y permite que cualquier proceso autorizado pueda decidir el destino de los puntos de recuperación de otros procesos. De la misma forma, los servicios para establecer un punto de recuperación o para recuperar el estado del proceso pueden afectar a otro proceso diferente al que invoca dichos servicios. Así, es posible gestionar la coordinación de los puntos de recuperación de diversos procesos que formen parte de una aplicación mediante un único proceso responsable. (Esta solución se utiliza en Manetho [Eln93, p. 55]).

Y el segundo argumento, *puerto*, indica el destino al que deben ser enviados los puntos de recuperación del proceso. Para el micro-núcleo, el proceso al que pertenezca dicho puerto actuará como servidor de puntos de recuperación. Tal y como se describe con detalle en el apartado 4.3.1, este proceso será responsable de gestionar los puntos de recuperación generados por el micro-núcleo y de proporcionarle a éste una copia del estado del proceso en caso de una vuelta atrás.

EstablecerPdR(proceso, opciones) → identificador

El servicio *EstablecerPdR()* permite capturar el estado del proceso. El micro-núcleo se encarga de generar un mensaje que contenga la información sobre el estado del proceso; sin embargo, es responsabilidad del servidor de puntos de recuperación encargarse de procesar el mensaje y de garantizar que, en caso de fallo, existe al menos una copia disponible del estado completo del proceso. En función de las opciones indicadas (y de las que admita el servidor de puntos de recuperación), el micro-núcleo utilizará (o no) optimizaciones como la generación de puntos de recuperación concurrentes o incrementales.

RestablecerProceso(puerto, identificador)

En caso de fallo, el proceso debe volver al estado almacenado en un punto de recuperación. Si el proceso original ya no existe, lo que ocurrirá siempre que se averíe el nodo del sistema en el que se estaba ejecutando, entonces es necesario crear un nuevo proceso que continúe con la ejecución a partir del estado almacenado en el punto de recuperación. Para ello se utiliza el servicio *RestablecerProceso()*. Al ser invocado, el micro-núcleo creará un proceso a partir de la información que obtendrá a través del puerto, que será una copia del estado completo del proceso.

RestablecerEstado(proceso, identificador, opciones)

En cambio, si el proceso que debe volver atrás todavía existe, simplemente es necesario conseguir que vuelva al estado almacenado en el punto de recuperación, utilizando el servicio *RestablecerEstado()*. Si se selecciona la opción *punto de recuperación incremental negativo* y el servidor de puntos de recuperación lo permite, entonces se parte del estado actual del proceso y se deshacen los cambios realizados por éste desde el punto de recuperación indicado por *identificador*. (Ver el apartado 4.3.1).

La probabilidad de que un proceso todavía exista en el sistema cuando debe volver atrás es especialmente elevada cuando se utilizan puntos de recuperación coordinados o estrategias optimistas para la gestión del registro. En ambos casos, un fallo en un proceso puede implicar la vuelta atrás de otros libres de errores, para garantizar que el estado global del sistema es consistente. En estos sistemas, los procesos libres de errores pueden solicitar un punto de recuperación incremental negativo que les permita volver atrás más eficientemente, al requerir la copia de menor cantidad de información.

Aunque las técnicas que almacenen el estado completo de los procesos en los puntos de recuperación generalmente buscarán la transparencia para la aplicación, es conveniente ofrecer la posibilidad de que los hilos del proceso reciban una notificación asíncrona que les informe de que se ha llevado a cabo una vuelta atrás. Usualmente, esta notificación asíncrona será capturada por código de librería, que de esta forma ocultará a la aplicación los detalles de la recuperación.

Un caso en el que esta notificación resulta de utilidad es el de las técnicas de recuperación hacia atrás que utilizan puntos de recuperación y registros. Tras la vuelta al estado almacenado en el punto de recuperación, el proceso pasa por una fase en la que está procesando el registro de mensajes y llamadas al sistema. Así se consigue que el proceso alcance un estado consistente con el del resto del sistema antes de continuar con su ejecución normal. Otro caso en el que puede utilizarse esta notificación es en las aplicaciones donde se desea que, tras la reconfiguración, se cambie el modo de ejecución del proceso a un modo de ejecución degradado, en el que algunas actividades complementarias no se realicen para sobrecargar menos el sistema, cuyas prestaciones pueden haberse reducido.

EstablecerDestinoLlamadas(proceso, puerto, opciones)
EstablecerDestinoNotificaciones(proceso, puerto, opciones)

Los servicios *EstablecerDestinoLlamadas()* y *EstablecerDestinoNotificaciones()* permiten configurar el destino y el tipo de información que se desea recabar sobre la interacción del proceso con el micro-núcleo, que se encargará de generar la información y enviarla, en forma de mensajes, a los puertos indicados. La gestión de los registros, de esta forma, queda en manos de los procesos a los que pertenezcan dichos puertos, a los que denominaremos servidores de registro, y que se encargarán de garantizar que siempre exista una copia de la información del registro que gestionan para una eventual recuperación en caso de errores.

El parámetro *opciones* permite especificar el conjunto de llamadas al sistema (en el primero de los servicios) y de notificaciones asíncronas (en el segundo) que resultan de interés para la aplicación. En el caso de las llamadas al sistema, también permite elegir si el resultado de la llamada debe ser incluido en los mensajes generados por el micro-núcleo (o si, por el contrario, sólo debe notificarse el hecho). Por último, es posible especificar si se debe permitir que el hilo continúe con su ejecución o, por el contrario, se le debe bloquear hasta que el servidor de registros haya almacenado la información (generación asíncrona y síncrona, respectivamente).

EstablecerDestinoPdR(proceso, puerto)
Establece el puerto como destino para los puntos de recuperación del proceso indicado.
Errores: *puerto no es válido* *el servidor no ha aceptado la conexión* *se ha especificado otro proceso y no se tiene autorización*
EstablecerPdR(proceso, opciones) → identificador
Genera un mensaje con el estado del proceso indicado. Devuelve el identificador del nuevo punto de recuperación.
Opciones: *punto de recuperación concurrente* *punto de recuperación incremental*
Errores: *no se ha establecido destino para puntos de recuperación* *el servidor no ha aceptado el mensaje* *se ha especificado otro proceso y no se tiene autorización*
RestablecerProceso(puerto, identificador)
Crea un proceso a partir de la información que puede obtener del puerto, que será un mensaje con el estado completo del proceso.
Errores: *el servidor no ha atendido la petición*
RestablecerEstado(proceso, identificador, opciones)
Sustituye el estado del proceso por el almacenado en el punto de recuperación especificado por identificador.
Opciones: *punto de recuperación incremental negativo*
Errores: *no se ha establecido destino para puntos de recuperación* *el servidor no ha atendido la petición* *se ha especificado otro proceso y no se tiene autorización*

Ilustración 4-8: Servicios relativos a la gestión de procesos. (I)[6]

[6] El formato utilizado para describir los servicios de la interfaz del micro-núcleo es el siguiente. En negrita, en la primera línea, aparece el prototipo de la función con el nombre de los argumentos y el valor devuelto, en caso de que exista. A continuación, en un párrafo, una breve descripción de la finalidad de la función. Después, si el servicio dispone de ellas, se describen las opciones que se pueden especificar al invocar el servicio. Finalmente, se ha incluido la descripción de algunos de los errores que pueden producirse al utilizar los servicios. Estos errores son especialmente significativos porque destacan el hecho de que el micro-núcleo depende de la existencia y buen funcionamiento de otros componentes lógicos en niveles superiores del sistema (como el servidor de puntos de recuperación, por ejemplo) para ofrecer estos servicios. La interacción con estos componentes se describe con detalle en los apartados 4.3.1 y 4.3.2.

EstablecerDestinoLlamadas(proceso, puerto, opciones)	
Establece el puerto como destino para la información sobre llamadas al sistema.	
Opciones:	*qué llamadas deben ser registradas* *qué información debe ser incluida* *generación síncrona/asíncrona*
Errores:	*puerto no es válido* *el servidor no ha aceptado la conexión* *se ha especificado otro proceso y no se tiene autorización*
EstablecerDestinoNotificaciones(proceso, puerto, opciones)	
Establece el puerto como destino para la información sobre notificaciones asíncronas.	
Opciones:	*qué notificaciones deben ser registradas* *generación síncrona/asíncrona*
Errores:	*puerto no es válido* *el servidor no ha aceptado la conexión* *se ha especificado otro proceso y no se tiene autorización*

Ilustración 4-9: Servicios relativos a la gestión de procesos. (II)

4.2.2 Paso de mensajes

Los conceptos sobre los mensajes incrementales han sido introducidos en el apartado 4.1. En este apartado se recogen los servicios que debe ofrecer el micro-núcleo relativos al paso de mensajes y a la gestión de las colas auxiliares de los puertos, y que se resumen en la Ilustración 4-10 y en la Ilustración 4-11, respectivamente.

EstablecerDestinoRegistro(puerto_origen, puerto_registro, opciones)

El servicio *EstablecerDestinoRegistro()* es bastante similar al que, en el apartado anterior, permitía especificar el destino para la información sobre llamadas al sistema y notificaciones asíncronas. De la misma forma, el micro-núcleo se encarga de generar la información relativa al intercambio de mensajes y enviarla, mediante mensajes, al puerto indicado como destino, donde será gestionada por el servidor de registros.

Sin embargo, en este caso, cada puerto para el intercambio de mensajes puede tener asociado un puerto diferente como destino para el registro de la información; aunque la opción *todos los puertos* permite establecer el mismo destino para todos los puertos del proceso. La razón de ser de esta opción es que, usualmente, existirá un único servidor de registros que recopilará toda la información relativa al proceso.

También es posible seleccionar si se desea incluir en el registro los mensajes enviados o recibidos por el proceso, así como si deben ser enviados al registro de forma síncrona o asíncrona, para facilitar la implementación de los diferentes tipos de técnicas de tolerancia a fallos basadas en registros. Por ejemplo, en [JZ87] se propone una técnica (*Sender-based Message Logging*) en la que los mensajes se almacenan en el nodo del emisor. En otros casos, los mensajes se envían al registro cuando son

recibidos [JZ88], tanto de forma asíncrona (estrategias optimistas) como síncrona (pesimistas).

Finalmente, es posible seleccionar qué debe incluirse en la información generada por el micro-núcleo: la descripción del mensaje o la descripción y el contenido. En el caso de los mensajes incrementales, se puede seleccionar si la descripción del mensaje debe incluir o no la descripción de las páginas incluidas en el mensaje.

EnviarMensaje(puerto, mensaje, opciones)
RecibirMensaje(puerto, opciones) → mensaje (identificador)

Los servicios para enviar y recibir mensajes no necesitan de explicación adicional, pues se corresponden con los servicios usuales de cualquier micro-núcleo. No obstante, el mecanismo de paso de mensajes incorpora la posibilidad de enviar y recibir mensajes incrementales utilizando las opciones correspondientes. El significado de las diferentes opciones se ha descrito en el apartado 4.1; mientras que el anexo II contiene una descripción más formal de una interfaz genérica para el intercambio de mensajes incrementales. En ella se describen exhaustivamente los efectos de la utilización conjunta de las diferentes opciones disponibles. Además, es conveniente destacar que los mensajes incluyen un campo con el destino incremental, definido en el apartado 4.1.5.

La opción *sin copia* resulta útil para manejar los mensajes con información fuera de línea. (Ver apartado 4.1.4). Mediante esta opción, los datos no son transferidos del emisor al receptor mediante copia en escritura, sino que los datos son eliminados del espacio de direcciones del emisor y proyectados en el espacio de direcciones del receptor. Esta opción forma parte de la interfaz ofrecida por el sistema operativo Mach para el paso de mensajes con información fuera de línea [Loe91, p. 27], y se incluye en la especificación de la presente interfaz por su utilidad para hacer eficiente el paso intermedio de mensajes.

EstablecerDestinoRegistro(puerto_origen, puerto_registro, opciones)
Establece puerto_registro *como destino para la información sobre mensajes intercambiados a través de* puerto_origen.
Opciones: *todos los puertos* *qué mensajes deben ser registrados* *qué información debe ser incluida* *generación síncrona/asíncrona*
Errores: *puerto no es válido* *el servidor no ha aceptado la conexión* *se ha especificado otro proceso y no se tiene autorización*
EnviarMensaje(puerto, mensaje, opciones)
Envía un mensaje incremental al puerto.
Opciones: *sin copia*
Opciones incrementales: *negativo* *empaquetado* *mapa*
RecibirMensaje(puerto, opciones) → mensaje (identificador)
Recibe un mensaje incremental de un puerto.
Opciones incrementales: *empaquetado* *cabecera* *usar dirección de origen*

Ilustración 4-10: Servicios relativos al intercambio de mensajes. (I)

En la Ilustración 4-12 se muestran los servicios necesarios para gestionar la cola auxiliar de mensajes asociada a un puerto. Esta cola, tal y como se ha descrito en el apartado 4.1.4, contiene los mensajes recibidos con la opción *cabecera*, lo cuál proporciona al receptor información sobre el contenido del mensaje (el mapa de las páginas incluidas en el mensaje, en el caso de un mensaje incremental) y transfiere el mensaje a la cola auxiliar del puerto.

```
ProyectarMensaje(puerto, id_mensaje, opciones) → mensaje
ReenviarMensaje(puerto_origen, id_mensaje, puerto_destino,
                opciones)
EliminarMensaje(puerto, id_mensaje)
```

Los tres servicios se corresponden con las tres operaciones que pueden realizarse sobre los mensajes almacenados en la cola auxiliar de un puerto: obtener el contenido del mensaje (*ProyectarMensaje()*), enviarlo a otro puerto (*ReenviarMensaje()*) o, simplemente, eliminarlo de la cola auxiliar (*EliminarMensaje()*). De las tres operaciones, sólo *ProyectarMensaje()* hace accesible el contenido del mensaje al dueño del puerto. En los otros dos casos, el mensaje será reenviado o eliminado sin que su contenido haya sido proyectado en el espacio de direcciones de este proceso, con las ventajas que ello supone (ver el apartado 4.1). Además, los dos primeros

servicios permiten seleccionar, mediante la opción *eliminar de la cola*, si debe quedar una copia en la cola auxiliar o no.

ProyectarMensaje(puerto, id_mensaje, opciones) → mensaje
Proyecta el contenido del mensaje id_mensaje *sobre el espacio de direcciones del proceso. El identificador debe corresponderse con uno de los mensajes almacenados en la cola auxiliar del* puerto*.* *Opciones:* *empaquetado* *eliminar de la cola*
ReenviarMensaje(puerto_origen, id_mensaje, puerto_destino, opciones)
Envía el mensaje depositado en la cola auxiliar de puerto_*origen e identificado por* id_mensaje *a* puerto_destino*.* *Opciones:* *eliminar de la cola*
EliminarMensaje(puerto, id_mensaje)
Extrae de la cola auxiliar de puerto *el mensaje cuyo identificador es* id_mensaje*. Libera todos los recursos ocupados por el mensaje. El espacio de direcciones del proceso no se ve modificado.*

Ilustración 4-11: Servicios relativos al intercambio de mensajes. (II) (Gestión de la cola auxiliar de los puertos).

4.2.3 Gestión de memoria

Los dos primeros servicios de este apartado, que se describen a continuación, permiten que se establezcan más eficientemente los puntos de recuperación del proceso, al permitirle indicar qué zonas de su espacio de direcciones pueden ser excluidas de los puntos de recuperación que se tomen a continuación.

ExcluirRegiónPdR(dirección, longitud) **IncluirRegiónPdR(dirección, longitud)**

Aunque los servicios para el paso de mensajes incrementales resultan más flexibles y eficientes a la hora de guardar y recuperar el estado de forma selectiva, ni *ExcluirRegiónPdr()*, ni *IncluirRegiónPdR()*, suponen una complicación adicional excesiva del núcleo. Además, facilitan la implementación de herramientas existentes, como las librerías ***libft*** (ver apartado 1.3.5) o ***libckpt*** [PBKL95].

Los tres servicios siguientes, que se resumen junto con los dos anteriores en la Ilustración 4-12, completan el conjunto de servicios necesarios para utilizar las operaciones descritas en el apartado 4.1.

CopiarVirtualmente(origen, destino, longitud, opciones, destincr)

El primero de ellos, *CopiarVirtualmente()*, permite que un proceso pueda copiar de forma eficiente gran cantidad de datos entre dos zonas de su espacio de direcciones. Para ello, invoca al micro-núcleo, que se encarga de transferir los datos mediante copia en escritura. Evidentemente, la cantidad de datos a copiar debe ser suficientemente grande como para que compense la sobrecarga debida a: dos cambios del modo de ejecución del procesador (de usuario a supervisor, y la vuelta a usuario), el tiempo empleado por el micro-núcleo para manipular las estructuras de datos para la gestión de la memoria, copias adicionales debido a la modificación de las páginas.

No obstante, la inclusión de este servicio permite que un proceso guarde parte de su estado en su propio espacio de direcciones para una eventual vuelta atrás, y lo haga de forma eficiente (incremental y concurrentemente). Como ya se indicó al describir el punto de recuperación incremental negativo, que un proceso deba volver atrás no implica necesariamente que el proceso haya desaparecido, ni que se hayan perdido los datos almacenados en su espacio de direcciones.

ConsultarEstado(comienzo, longitud, opciones) → estado
EstablecerEstado(comienzo, longitud, opciones, estado)

Finalmente, los servicios *ConsultarEstado()* y *EstablecerEstado()* permiten al proceso controlar el derecho de acceso a una zona de memoria, mediante la opción *protección*, o el estado de dicha zona con respecto a los diferentes destinos incrementales (ver el apartado 4.1.5), mediante la opción *operaciones incrementales*.

El hecho de permitir al proceso que restrinja sus derechos de acceso sobre las diferentes zonas de su espacio de direcciones no es nuevo (por ejemplo, el servicio *mprotect()* forma parte de la definición SVID3). No obstante, se ha incluido como parte de la interfaz para el desarrollo de aplicaciones tolerantes a fallos para permitir que los procesos aumenten selectivamente las comprobaciones en tiempo de ejecución que realiza el mecanismo de traducción de direcciones del soporte físico. Tal y como se recoge en el apartado 3.4, estas comprobaciones se encuentran entre las herramientas de detección de errores más efectivas de las proporcionadas por los soportes físicos que no han sido diseñados especialmente para sistemas tolerantes a fallos.

La segunda opción tiene una doble finalidad. Por una parte, permite que el proceso especifique qué páginas deben ser incluidas en las siguientes operaciones incrementales, consiguiendo mayor rendimiento que con el comportamiento por defecto. Por ejemplo, si un proceso con una réplica pasiva en otro nodo crea una zona de memoria y la llena con el contenido de un fichero, en el siguiente mensaje incremental se incluirán todas las páginas de esta zona. Sin embargo, si la réplica también puede acceder a una copia del fichero almacenada en su propio nodo, esta transferencia de datos a través del medio de comunicación puede ser evitada. Para ello, la réplica activa cambiaría el estado de la región, para indicar que no debe enviarse ninguna de las páginas hasta que no sea modificada.

Y la segunda finalidad es, precisamente, facilitar la detección de errores. Un proceso puede, antes de hacer permanentes sus modificaciones, comprobar si las páginas que ha modificado se corresponden con las que debía modificar, como mecanismo para detectar errores en su ejecución. (Ver el apartado 4.3.3).

ExcluirRegiónPdR(dirección, longitud)
Excluye la región de memoria virtual indicada por dirección *y* longitud *de los puntos de recuperación del proceso.*
IncluirRegiónPdR(dirección, longitud)
Incluir la región de memoria virtual indicada por dirección *y* longitud *en los puntos de recuperación del proceso.*
CopiarVirtualmente(origen, destino, longitud, opciones, destincr)
Copiar, mediante técnicas de memoria virtual, los datos indicados. Si la copia es incremental se utiliza destincr *para saber a qué destino incremental corresponde la copia.* *Opciones:* *incremental* *incremental negativa*
ConsultarEstado(comienzo, longitud, opciones) → estado
*Consultar información sobre la zona de memoria virtual indicada por el par (*comienzo*,* longitud*).* *Opciones:* *protección* *operaciones incrementales*
EstablecerEstado(comienzo, longitud, opciones, estado)
*Modificar la información de la zona de memoria virtual indicada por el par (*comienzo*,* longitud*).* *Opciones:* *protección* *operaciones incrementales*

Ilustración 4-12: Servicios relativos a la gestión de memoria virtual.

Correspondencia entre servicios y necesidades.

Una vez especificada la interfaz, es necesario verificar hasta qué punto los servicios propuestos satisfacen las necesidades a nivel del núcleo. La Tabla 4-2 muestra la correspondencia entre los servicios del micro-núcleo (filas) y las necesidades (columnas). Tanto los puntos como las opciones que aparecen en las casillas indican que el servicio (o la opción del servicio) correspondiente a la fila está destinado a satisfacer la necesidad que aparece en la columna.

	Guardar y recuperar el estado	Generar inf. para los registros	Gestión eficiente del paso interm. de mensajes	Detec. y conf. de errores
EstablecerDestinoPdR	•			
EstablecerPdR	•			
RestablecerProceso	•			
RestablecerEstado	•			
EstablecerDestinoLlamadas		•		
EstablecerDestinoNotificaciones		•		
EstablecerDestinoRegistro		•		
EnviarMensaje	incr. incr. neg. incr. mapa		sin copia incr. neg. incr. emp.	
RecibirMensaje	incr. incr. emp.		cabecera incr. emp.	cabecera
ProyectarMensaje			•	
ReenviarMensaje			•	
EliminarMensaje			•	
ExcluirRegiónPdR	•			
IncluirRegiónPdR	•			
CopiarVirtualmente	•			
ConsultarEstado	incr.			•
EstablecerEstado	incr.			protección

Tabla 4-2: Revisión de la utilidad de los servicios en base a las necesidades a nivel del núcleo.

La mayor parte de los servicios que se han propuesto tienen como finalidad permitir guardar y recuperar el estado de los procesos de forma selectiva y eficiente. Los servicios relativos al establecimiento de puntos de recuperación, junto con las operaciones incrementales atienden esta necesidad. Aunque el número de servicios parece elevado, los mecanismos sobre los que se apoyan son reducidos: la copia en escritura y el aprovechamiento de la monitorización de las modificaciones que realiza la unidad de gestión de memoria de la mayoría de los procesadores modernos. No obstante, una vez justificada la necesidad de los mecanismos básicos, se ha diseñado el resto de la interfaz de forma que resulte lo más versátil y completa posible.

La generación de la información para el registro requiere únicamente que el proceso pueda establecer el destino al que el micro-núcleo debe enviar esta información. En cambio, para permitir la gestión eficiente del paso intermedio de mensajes, se ha añadido a los puertos la cola auxiliar de mensajes y los servicios necesarios para que los procesos tengan la capacidad de gestionarla atendiendo a sus propias necesidades.

Por otra parte, la detección y confinamiento de errores tiene una representación discreta en la interfaz. Esto no es extraño, ya que los aspectos básicos a cubrir eran la detección de errores en la interfaz y la detección de errores internos (ver apartado 3.5). Ambos aspectos requieren un diseño e implementación del micro-núcleo que realice las comprobaciones que se consideren adecuadas (ver apartado 3.4). No obstante, los servicios para consultar las páginas modificadas y establecer los derechos de acceso

sobre las regiones de memoria virtual permiten comprobaciones adicionales que suponen una sobrecarga únicamente para los procesos que los utilizan, tal y como se deseaba. Así mismo, la opción de recibir la cabecera de un mensaje incremental permite la detección y el confinamiento de errores, tal y como se describe en el apartado 4.3.3.

4.3 Aspectos adicionales sobre los servicios

En este apartado se completa la descripción de los servicios mostrando cómo interaccionan con otros módulos lógicos que han sido nombrados en el apartado anterior, como el servidor de puntos de recuperación o el servidor de registros (puntos 4.3.1 y 4.3.2). Así mismo, también se ilustra con un ejemplo cómo se pueden utilizar algunos de los servicios del micro-núcleo para aumentar la capacidad de detección y confinamiento de errores mediante comprobaciones adicionales en niveles superiores del sistema. Y, finalmente, se muestra cómo se pueden utilizar los mensajes incrementales empaquetados para intercambiar mensajes incrementales entre procesos residentes en nodos diferentes del sistema.

4.3.1 Gestión de puntos de recuperación

Tal y como se ha introducido en el apartado 4.2, el micro-núcleo no almacena los puntos de recuperación de los procesos, sino que se apoya en servidores externos, los servidores de puntos de recuperación.[7] A continuación se describe la interacción entre el micro-núcleo y estos servidores. Es importante destacar que la Ilustración 4-13 recoge los servicios requeridos por el micro-núcleo y que, por lo tanto, deben estar presentes en cualquier servidor de puntos de recuperación.[8] No obstante, la gestión de los puntos de recuperación puede requerir servicios adicionales para que las aplicaciones puedan utilizar técnicas como el establecimiento de puntos de recuperación coordinados. Por ejemplo, se pueden incluir servicios para permitir que los nuevos puntos de recuperación sean tentativos, en espera de ser confirmados o desechados posteriormente, para permitir protocolos de coordinación como el establecimiento en dos fases (*two-phase commit protocol*) [LA94].

[7] Aunque se ha utilizado el término servidor, esto no quiere decir que el servicio de gestión de puntos de recuperación sea llevado a cabo por un único proceso aislado. No obstante, como el micro-núcleo no contiene primitivas para la comunicación de grupo, en última instancia habrá un proceso encargado de atender las peticiones que el micro-núcleo realice a través del puerto especificado por *EstablecerDestinoPdR()*.

[8] Nótese que el micro-núcleo invocará los servicios para la gestión de puntos de recuperación mediante el envío de mensajes a través del puerto del servidor.

EstablecerConexión(proceso) → opciones
El servidor de puntos de recuperación se prepara para gestionar puntos de recuperación del proceso indicado. Devuelve al micro-núcleo las opciones que puede utilizar. *Opciones:* *incremental* *incremental negativo*
GuardarPdR(proceso, opciones, estado) → identificador
El servidor almacena la información sobre el estado del proceso contenida en el mensaje generado por el micro-núcleo. Devuelve un identificador único para este nuevo punto de recuperación. *Opciones:* *incremental*
RecuperarPdR(identificador, opciones [, estado]) → estado del proceso
El servidor proporciona la información correspondiente a identificador. Si se utiliza la opción incremental negativo*, solamente se proporcionan las páginas indicadas en* estado*.* *Opciones:* *incremental negativo*
TerminarConexión(proceso)
El servidor da por terminada la generación de información sobre el estado del proceso.

Ilustración 4-13: Servicios requeridos por el micro-núcleo a los servidores de puntos de recuperación.

La primera interacción se produce cuando se utiliza el servicio *EstablecerDestinoPdR()* del micro-núcleo. Entonces, el micro-núcleo, si el puerto especificado es válido, invocará el servicio *EstablecerConexión()* del servidor de puntos de recuperación, para obtener una descripción de las opciones que éste admite.

A partir de este momento, cada vez que se solicite al micro-núcleo el establecimiento de un nuevo punto de recuperación del proceso —mediante *EstablecerPdR()*—, el micro-núcleo preparará la información sobre el estado del proceso y la almacenará en el servidor utilizando el servicio *GuardarPdR()*. Tal y como se ha descrito en el apartado 4.2, es posible solicitar que el micro-núcleo establezca puntos de recuperación concurrentes e incrementales. No obstante, para establecer puntos de recuperación incrementales, es necesario que el servidor de puntos de recuperación sea capaz de gestionarlos. Por tanto, sólo si *incremental* está entre las opciones devueltas por *EstablecerConexión()*, el micro-núcleo establecerá el punto de recuperación de forma incremental. En la Ilustración 4-14 se muestra un ejemplo en el que el proceso P_2 solicita el establecimiento de un punto de recuperación concurrente e incremental del proceso P_1 (1). El la ilustración se muestra cómo el micro-núcleo envía los cambios en el estado de P_1 al servidor de puntos de recuperación (2), que responde al micro-núcleo con el identificador del punto de recuperación (3), que le es devuelto a P_2 (4).

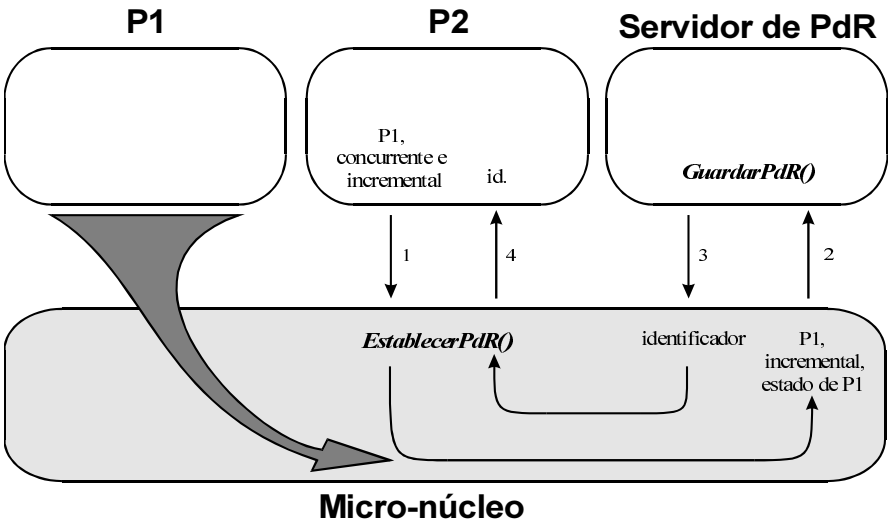

Ilustración 4-14: Establecimiento de un punto de recuperación del proceso P1, solicitado por el proceso P2.

Si se solicita una vuelta atrás del proceso —solicitando los servicios *RestablecerProceso()* o *RestablecerEstado()*—, el micro-núcleo enviará un mensaje al servidor solicitándole el estado del proceso —servicio *RecuperarPdR()*—. Si se ha utilizado la opción *punto de recuperación incremental negativo* al invocar el servicio del micro-núcleo *RestablecerEstado()*, y el servidor admite esta posibilidad, el micro-núcleo incluirá en el mensaje la información sobre las páginas que deben ser proporcionadas por el servidor y utilizará la opción *incremental negativo* al invocar *RecuperarPdR()*.

Finalmente, cuando termina la ejecución del proceso, el micro-núcleo se lo notifica al servidor utilizando el servicio *TerminarConexión()*.

Servicios ofrecidos por el micro-núcleo		Servicios del servidor de puntos de recuperación de los que depende
EstablecerDestinoPdR()		***EstablecerConexión()***
EstablecerPdR()		***GuardarPdR()***
RestablecerProceso()		***RecuperarPdR()***
RestablecerEstado()		
		TerminarConexión()

Ilustración 4-15: Resumen de la dependencia que presenta el micro-núcleo de los servicios de gestión ofrecidos por el servidor de puntos de recuperación.

4.3.2 Gestión de registros

Al igual que ocurre con los puntos de recuperación, el micro-núcleo no gestiona ninguno de los tipos de registro (de mensajes, de llamadas al sistema o de notificaciones asíncronas). Esta gestión debe ser realizada por los servidores de registros.[9] A diferencia de lo que ocurría en el caso de los puntos de recuperación, el micro-núcleo no tiene necesidad de recuperar información de los registros, sino que únicamente se encarga de añadir información al registro cada vez que se produce un suceso que debe ser registrado (envío o recepción de un mensaje, una llamada al sistema, o una notificación asíncrona al proceso).

Esto hace que la interfaz requerida por el micro-núcleo a cualquier servidor de registro conste únicamente de dos servicios. El primero, *EstablecerConexión()*, lo utiliza el micro-núcleo cuando se invoca a *EstablecerDestinoLlamadas(), EstablecerDestinoNotificaciones()* o a *EstablecerDestinoRegistro()*. El micro-núcleo especifica el tipo de sucesos que van a ser registrados. El servidor, si no existe previamente, prepara un nuevo registro para el proceso. Si el registro ya existía, lo prepara para que pueda contener, también, los nuevos tipos de sucesos.

Durante la ejecución del proceso, cada vez que se produzca un suceso que deba ser registrado, el micro-núcleo preparará un mensaje que describa el tipo del suceso que se ha producido y contenga la información que deba ser almacenada en el registro (por ejemplo, el contenido del mensaje) y, a continuación, invocará el servicio *GuardarSuceso()* del servidor.

Finalmente, cuando el proceso termine con su ejecución, el micro-núcleo se lo hará saber al servidor mediante el servicio *TerminarConexión()*.

EstablecerConexión(proceso, tipos)	
El servidor de registros se prepara para mantener un registro relativo a este proceso que contenga sucesos de los tipos indicados.	
Tipos:	*envío de mensaje* *recepción de mensaje* *llamada al sistema* *notificación asíncrona*
GuardarSuceso(proceso, tipo, datos)	
El servidor almacena los datos sobre el suceso del tipo indicado.	
Tipo:	*envío de mensaje* *recepción de mensaje* *llamada al sistema* *notificación asíncrona*
TerminarConexión(proceso)	
El servidor da por terminada la generación de información para el registro del proceso.	

Ilustración 4-16: Servicios requeridos por el micro-núcleo a los servidores de registros.

[9] Como con los servidores de recuperación, aunque se ha utilizado el término servidor, esto no quiere decir que el servicio de gestión de registros tenga que ser llevado a cabo por un único proceso.

4.3.3 Detección y confinamiento de errores

La evaluación de necesidades respecto a la detección y confinamiento de errores realizada en el apartado 3.4, junto con la evaluación de la interfaz que se realiza al final del apartado 4.2, han descrito suficientemente los mecanismos internos del micro-núcleo para la detección de errores propios y del soporte físico en el que se apoya, y los servicios que debían ser añadidos a la interfaz para permitir aumentar la eficacia del micro-núcleo a la hora de detectar errores en los componentes lógicos que se apoyan en él.

No obstante, algunos de los servicios incluidos en la interfaz para satisfacer otras necesidades pueden facilitar la detección de errores de los procesos mediante comprobaciones adicionales. A diferencia de las comprobaciones realizadas por el micro-núcleo, que afectan a todos los procesos del nodo, estas comprobaciones forman parte de los niveles superiores del sistema y pueden ser utilizadas selectivamente en aquellas aplicaciones cuyas necesidades —en términos de garantía de funcionamiento— lo justifiquen.

La posibilidad de recibir la cabecera de los mensajes incrementales, antes de que la información contenida en el mensaje modifique el espacio de direcciones del receptor facilita la detección de errores del emisor. En efecto, si el receptor dispone de su propia información sobre qué páginas debería haber modificado el emisor y qué páginas no debería haber modificado en ningún caso, es posible detectar aquellos errores que provoquen un cambio en el comportamiento del proceso. A continuación se muestra un ejemplo de esta posibilidad.

Sea un servicio que ha sido implementado utilizando dos procesos que trabajan sobre sendas copias del mismo conjunto de datos. Supóngase que se permite que cada uno realice modificaciones sobre los datos y que ambos utilizan un algoritmo distribuido de exclusión mutua para evitar accesos incompatibles a los datos (por ejemplo, que ambos modifiquen la misma página).

En esta situación, cada proceso tiene información sobre qué páginas pueden haber sido modificadas por el otro. En la Ilustración 4-17 se muestra un ejemplo, en el que P_1 y P_2 han acordado (mediante el algoritmo distribuido de exclusión mutua) que P_1 iba a modificar las páginas 1 y 7. Cuando P_1 vaya a actualizar la copia de los datos que mantiene P_2, le enviará un mensaje incremental que, en el ejemplo, contendrá las páginas 1 y 7.

Para detectar posibles errores de P_1, y evitar que los efectos se propaguen a la copia de los datos de P_2, éste puede recibir únicamente la cabecera del mensaje incremental y comprobar que, efectivamente, el mensaje contiene las páginas 1 y 7, antes de proyectarlo sobre su espacio de direcciones.

En general, P_1 y P_2 residirán en nodos diferentes y así, tanto si el error se debe a un fallo en P_1 —de diseño, pues es un componente lógico— como si se debe a una avería en otros componentes del nodo —tanto lógicos como físicos— cuyos servicios haya utilizado P_1, es posible confinar el daño en el nodo de P_1.

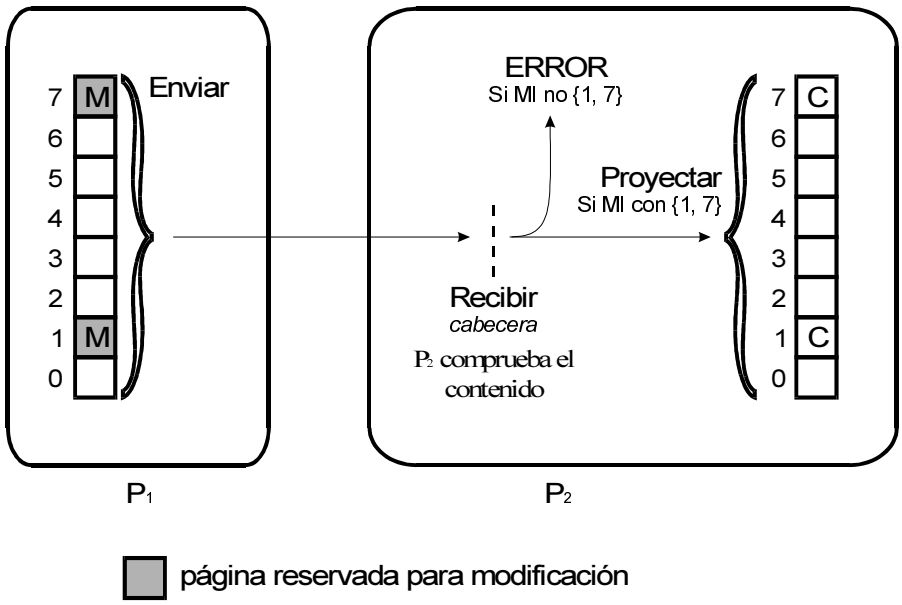

Ilustración 4-17: Ejemplo de utilización de recibir con cabecera como medio para la detección y confinamiento de errores.[CPC9]

Del mismo modo, la posibilidad que tiene un proceso de consultar el estado de las páginas mediante la función *ConsultarEstado()* le permite detectar sus propios errores. Por ejemplo, en el caso del ejemplo, P_1 podría utilizar este servicio para comprobar si las páginas que ha modificado se corresponden con las que tiene anotadas como reservadas para modificación.

4.3.4 Transferencia de mensajes incrementales entre nodos

En el apartado 2.4 se especifica que la comunicación con otros nodos no forma parte del núcleo y se propone que sea implementada mediante un proceso, al que se ha denominado servidor de comunicación entre nodos. Este proceso se encargará de transmitir, a través del medio de comunicación, los mensajes entre procesos que residen en diferentes nodos.

La transferencia de mensajes incrementales entre nodos a través de los servidores de comunicación puede realizarse utilizando los mensajes incrementales empaquetados de forma que resulte transparente tanto para el emisor como para el receptor el hecho de que ambos estén en nodos diferentes. En la Ilustración 4-18 se muestra cómo se puede transferir un mensaje incremental cuyo emisor sea el proceso P_1, residente en el nodo 1, y cuyo receptor sea el proceso P_2, residente en el nodo 2, a través de los servidores de comunicación (SC_1 y SC_2).

SC_1 hace de representante de los procesos de otros nodos en el nodo 1, para que los procesos de ese nodo puedan enviar los mensajes de la misma forma que lo harían si el proceso de destino residiese en ese mismo nodo. No obstante, SC_1 es un mero

intermediario cuya función es recoger el mensaje incremental y transferirlo a través del medio de comunicación, por lo que no mantiene una copia actualizada de los datos del emisor, sino que almacena temporalmente los datos hasta que puedan ser transferidos a través del medio de comunicación. Por eso utiliza la opción *empaquetado* cuando invoca *RecibirMensaje()*, con lo que los datos son almacenados en una zona contigua de su espacio de direcciones, en forma de mensaje incremental empaquetado.

A continuación, SC_1 transfiere el mensaje incremental empaquetado a SC_2 a través del medio de comunicación, y SC_2 se lo envía a su vez a P_2, utilizando la opción *empaquetado* al invocar *EnviarMensaje()*. Puesto que P_2 sí desea una actualización de su copia de los datos, el mensaje incremental empaquetado actualizará la copia de los datos de P_1 que mantiene P_2, ya que P_2 no utiliza la opción *empaquetado* al invocar *RecibirMensaje()*.

Es conveniente resaltar que el mensaje incremental empaquetado permite aprovechar toda la potencia de los controladores de DMA que permiten la reunión/dispersión (*scatter/gather*) de datos. Esto es posible porque este tipo de mensajes se almacena en una zona contigua del espacio de direcciones virtual, pero no necesariamente en una zona contigua de memoria física, facilitando la gestión de esta última.

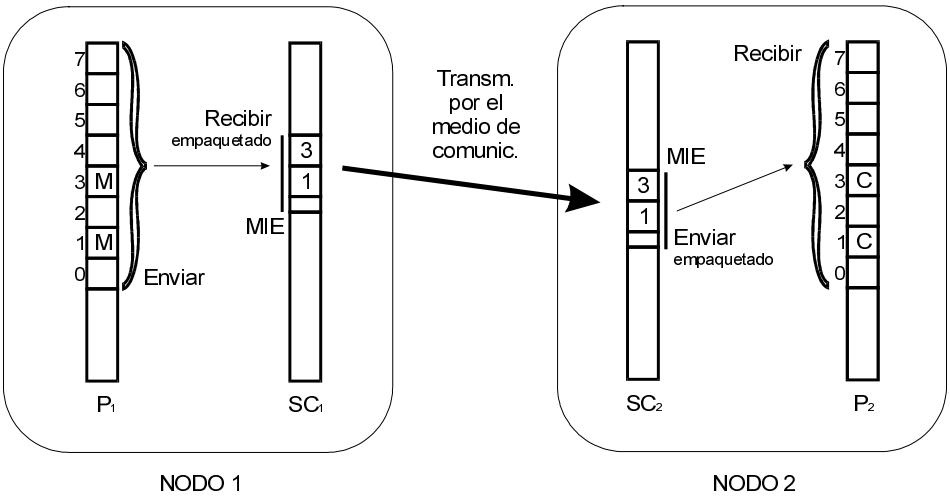

Ilustración 4-18: Transmisión de mensajes incrementales entre nodos utilizando mensajes incrementales empaquetados.

4.4 Conclusiones

En este capítulo se ha especificado el conjunto de servicios que deben ser incluidos en el micro-núcleo para que se satisfagan las necesidades de las aplicaciones a este nivel, y que han sido identificadas en el capítulo 3.

Se ha propuesto un conjunto de nuevos servicios, a los que se ha denominado operaciones incrementales, con la finalidad de permitir el almacenamiento, la recuperación, la actualización y la recuperación del estado de los procesos de forma selectiva y eficiente. La cualidad más importante de estos servicios es que proporcionan la eficiencia de los puntos de recuperación incrementales y concurrentes al mismo tiempo que permiten a los niveles superiores del sistema establecer qué datos deben ser transferidos y cuándo. Para completar la funcionalidad de estos servicios, permitiendo la gestión eficiente del paso intermedio de mensajes, se ha propuesto añadir a los puertos una cola auxiliar de mensajes, que es gestionada por los niveles superiores del sistema mediante funciones de la interfaz.

La especificación de la interfaz se ha completado con la descripción de las interacciones del micro-núcleo con otros componentes lógicos del sistema para proporcionar servicios como el establecimiento de puntos de recuperación o la gestión de registros.

Por último, se ha mostrado cómo se pueden utilizar los nuevos servicios para aumentar la capacidad de detección y confinamiento de errores en el sistema. Aunque el método propuesto requiere la realización de comprobaciones adicionales, éstas no son realizadas por el micro-núcleo, sino por niveles superiores del sistema. De esta forma, pueden ser utilizadas selectivamente por aquellas aplicaciones en las que se justifique por la necesidad de aumentar su garantía de funcionamiento.

Capítulo 5

Evaluación de la especificación del micro-núcleo

Para comprobar que los servicios incluidos en la interfaz forman un conjunto completo, versátil y eficiente de herramientas, se ha seleccionado una serie de casos que se consideran suficientemente representativos de las técnicas de tolerancia a fallos actuales.

En primer lugar, se han escogido dos aspectos concretos de sistemas que se apoyan en los servicios estándar de Unix. El primero, tomado del Sistema Avanzado de Automatización, consiste en la implementación de réplicas pasivas. El segundo, basado en el caso descrito en el apartado 1.3.5 sobre la tolerancia a fallos a nivel del soporte lógico en AT&T, consiste en la implementación de funciones para el establecimiento de puntos de recuperación desde una librería a nivel de usuario.

En segundo lugar se ha escogido Manetho porque incluye, combinadas, técnicas de tolerancia a fallos basadas tanto en recuperación hacia atrás como en redundancia modular. Además, la implementación original incluye modificaciones tanto en el nivel de utilidades como en el del núcleo del sistema operativo. Por lo tanto, su implementación permite poner a prueba los servicios propuestos en el capítulo anterior.

En tercer lugar se propone una estructura para el desarrollo de servicios transaccionales apoyándose en las operaciones incrementales. A diferencia de los casos anteriores, donde se ha respetado el diseño original con la intención de mostrar la versatilidad y completitud de los servicios incluidos en el micro-núcleo, en este caso la estructura es nueva, constituyendo una aportación del presente trabajo de investigación.

Finalmente, se revisa que, efectivamente, los casos considerados son representativos de las diferentes técnicas de tolerancia a fallos actuales y de los sistemas considerados en el capítulo 1.

5.1 Servicios tolerantes a fallos mediante grupos de servidores débilmente sincronizados

Para este ejemplo se va a seguir la descripción de [CDD90] sobre la implementación de servicios tolerantes a fallos mediante grupos de servidores y, en particular, los grupos débilmente sincronizados. Aunque el apartado 1.3.2 resume las características de este sistema —Sistema Avanzado de Automatización, ASS—, es conveniente describir con mayor detalle la estrategia de tolerancia a fallos que se desea implementar utilizando los servicios propuestos para el micro-núcleo.

En el AAS se define el estado del servicio (*service state*) como el resultado de aplicar, sobre el estado inicial, todas las operaciones solicitadas por los clientes desde la inicialización. Cada servicio es implementado por un conjunto de procesos servidores y uno de ellos, al que se denomina servidor principal (*primary server*) es el único que procesa las peticiones de servicio y responde a los clientes. El resto de los miembros del grupo reciben periódicamente una copia del estado del servidor principal y mantienen un registro de las peticiones de los clientes. De esta forma, en caso de fallo del servidor principal, es posible reconstruir el estado del servicio haciendo que uno de los otros miembros del grupo aplique las operaciones registradas desde la última actualización al estado que recibió del servidor principal. El servidor principal puede enviar su estado con diferente periodicidad a cada uno del resto de los miembros del grupo, dando lugar a una ordenación de éstos en función de lo cercano que es su estado al del servicio.

Los mensajes incrementales facilitan la implementación eficiente de esta técnica de tolerancia a fallos. En efecto, para actualizar el estado de cada una de las réplicas pasivas, el servidor principal simplemente tiene que enviar un mensaje incremental utilizando el servicio *EnviarMensaje()*. La réplica pasiva, tal y como se describe en el apartado 4.1.1, únicamente necesita utilizar el servicio *RecibirMensaje()* para que se actualice su copia de los datos con la información incluida en el mensaje incremental. Al utilizar mensajes incrementales, la cantidad de información que necesita ser transferida en cada mensaje se ve reducida, y además no depende directamente del tamaño de los datos críticos, sino de las modificaciones realizadas desde la última actualización de la copia. Por lo tanto, la cantidad de datos a transferir con cada copia será menor a medida que aumente la frecuencia de las actualizaciones. Así, es posible mantener el estado del resto de los miembros del grupo más próximo al del servicio,

reduciéndose el tiempo de recuperación en caso de fallo, ya que el registro contendrá menos peticiones que procesar.

El servidor puede actualizar el estado de cada réplica con la periodicidad más adecuada para cada aplicación. Para ello, simplemente tiene que incluir en los mensajes un valor diferente para el destino incremental (ver apartado 4.1.5) en función de la réplica a la que envía en mensaje, tal y como se describe en el ejemplo siguiente.

En la Ilustración 5-1 se muestra un ejemplo con un grupo de servidores débilmente sincronizados formado por el servidor principal y dos réplicas (0 y 1), para las que el servidor principal utiliza, respectivamente, los destinos incrementales 0 y 1. Dentro de las páginas del servidor se muestra el estado ('*M*', modificado; '-' sin modificar) de cada página respecto a los diferentes (en este caso dos) destinos incrementales. Como se ha especificado en el apartado 4.1, el micro-núcleo mantiene actualizado el estado de cada página respecto a cada destino incremental apoyándose en el mecanismo de traducción de direcciones del procesador, de forma transparente para los niveles de utilidades y aplicaciones.

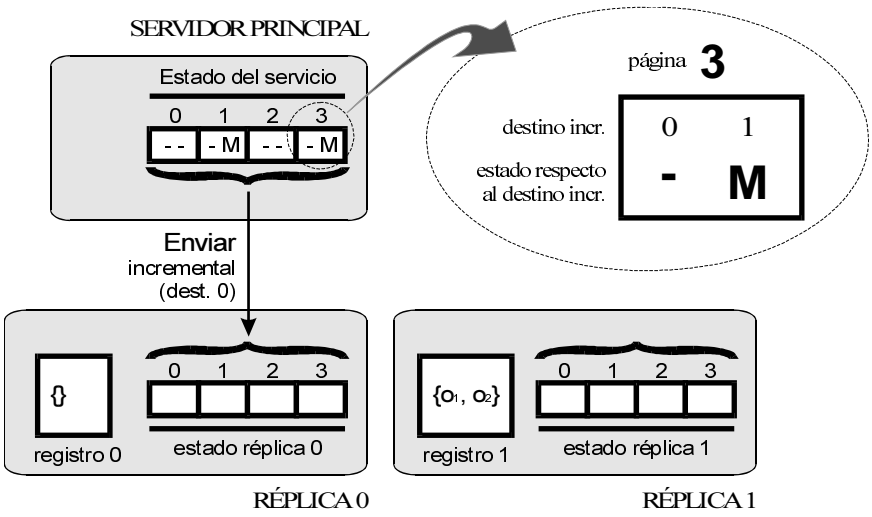

Ilustración 5-1: Tolerancia a fallos mediante grupos de servidores débilmente sincronizados.

El servidor principal acaba de enviar un mensaje incremental a la réplica 0 y, puesto que se supone que el servidor principal todavía no ha realizado ninguna operación desde el mensaje incremental anterior, ninguna de sus páginas está modificada respecto a este destino. Por lo tanto, el estado de la réplica 0 es igual al estado del servicio (mantenido en el servidor principal) y su registro de operaciones pendientes está vacío.

En la ilustración también se representa una segunda réplica, la 1, cuyo estado es diferente al estado del servicio. Para obtener el estado del servicio a partir del estado de la réplica 1 sería necesario aplicar las operaciones del registro 1 (O_1 y O_2) al estado de dicha réplica. En este caso se supone que la aplicación de estas operaciones supone

la modificación de las páginas 1 y 3. Por eso, ambas páginas están marcadas como modificadas respecto al destino incremental 1.

Si el servidor principal decidiese actualizar la copia que mantiene la réplica 1 en este momento, enviaría un mensaje incremental con destino 1. Entonces, el micro-núcleo incluiría en el mensaje las páginas 1 y 3. Al recibir el mensaje incremental se actualiza el estado de la réplica, por lo que su estado sería equivalente al del servicio. Por lo tanto, tras recibir el mensaje, la réplica vaciaría su registro.

Conclusiones

En este caso, la presencia de los servicios especificados para el micro-núcleo y, especialmente, de las operaciones incrementales, permite aumentar la eficiencia, de forma sencilla, de los grupos de servidores débilmente sincronizados.

Las operaciones incrementales permiten aumentar la eficiencia de la actualización del estado de las réplicas, ya que únicamente se copian las páginas modificadas desde la actualización anterior. Además, como el servidor principal puede especificar la región que contiene los datos críticos, no se añade la sobrecarga que añadirían servicios para el establecimiento de puntos de recuperación, que incluyen el estado completo del proceso.

Como el coste de la actualización depende, básicamente, del número de páginas modificadas y este número será menor, en general, cuanto más frecuentes sean las actualizaciones, es posible aumentar la frecuencia de éstas. De esta forma, es posible conseguir mejores tiempos de recuperación, ya que el estado de las réplicas será más cercano al del servidor.

Por otra parte, estas ventajas no suponen una mayor complejidad ni del servidor principal ni de las réplicas, puesto que, para actualizar el estado de una réplica, el servidor se limita a enviar un mensaje que debe recibir ésta (la gestión del registro en las réplicas continúa realizándose igual que se realizaría sin las operaciones incrementales). Además, el servidor puede actualizar cada réplica con la periodicidad más adecuada para la aplicación, simplemente utilizando un destino incremental diferente para cada una, pues el micro-núcleo mantiene el estado de las páginas respecto a los diferentes destinos incrementales.

5.2 Puntos de recuperación desde una librería a nivel de usuario

En el apartado 1.3.5 se describen las técnicas utilizadas en AT&T para conseguir tolerancia a fallos mediante el soporte lógico en un sistema distribuido. En el artículo de Y. Huang y C. Kintala [HK93] se destaca que las técnicas generales para el establecimiento de puntos de recuperación son necesariamente ineficientes, ya que son las aplicaciones las que disponen de la información necesaria para seleccionar los datos que deben ser guardados. En el artículo se describe una librería de funciones que implementa técnicas de tolerancia a fallos (*libft*). Entre otras facilidades, la librería

permite que el proceso establezca puntos de recuperación de los datos críticos (que él mismo puede definir).

En ese caso, el punto de recuperación se almacena en disco, utilizando el sistema de ficheros nDFS que replica los ficheros críticos tanto en el nodo principal como en el de respaldo. No obstante, cada operación de escritura supone la copia de toda la región que contiene los datos críticos de la aplicación. Estos datos deben ser transportados a través de la red al nodo de respaldo y, una vez allí, desde la memoria al disco.

En un sistema basado en micro-núcleo, el servicio de ficheros puede ser implementado utilizando un proceso servidor que tenga acceso a un disco. Si el micro-núcleo dispone de los servicios especificados en el capítulo anterior, entonces es posible implementar los servicios de *libft* para establecer puntos de recuperación apoyándose en las operaciones incrementales. En este caso, el servidor de ficheros debe ser modificado para ofrecer una operación de actualización —a la que denominaremos *ActualizarFichero()*—, de forma que el cliente pueda utilizar un fichero como copia de respaldo de sus datos críticos.

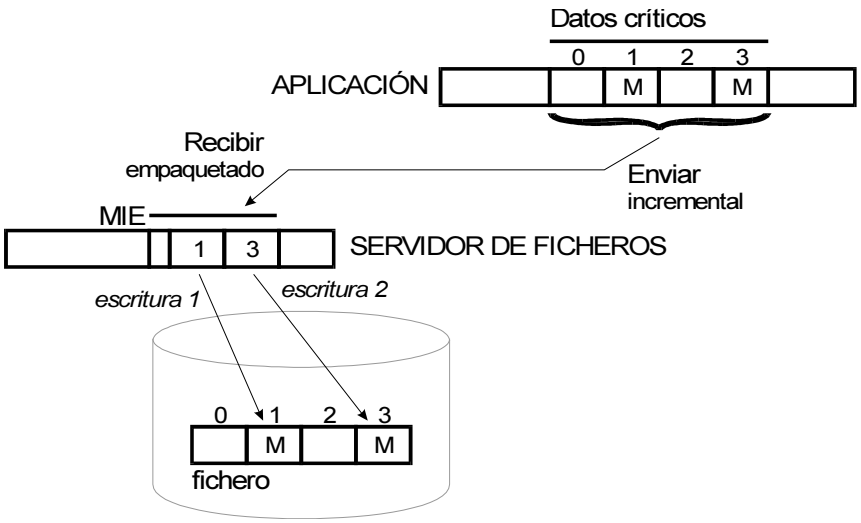

Ilustración 5-2: Utilización de los MIEs para mantener en un fichero una réplica actualizada de una zona de memoria virtual.

El cliente, durante su inicialización, comienza por crear un nuevo fichero y rellenarlo con una copia de los datos críticos. Para ello, invoca la operación *ActualizarFichero()* del servidor de ficheros, enviando un mensaje incremental que, por ser el primero, contendrá todas las páginas de la región en la que residen los datos. Los puntos de recuperación posteriores (actualizaciones sucesivas) se establecen de la misma forma enviando nuevos mensajes incrementales; pero, en este caso, los mensajes incluirán únicamente las páginas modificadas. (Ver apartado 4.1.1).

Por su parte, el servidor de ficheros recibe estos mensajes incrementales en su formato empaquetado, ya que la copia de los datos se encuentra almacenada en disco, y

no en una región de memoria en su espacio virtual de direcciones, tal y como muestra la Ilustración 5-2. Una vez proyectado el mensaje incremental empaquetado sobre su espacio de direcciones, el servidor interpreta el mapa incluido en el mensaje y lo utiliza para transferir cada página al lugar que debe ocupar en el fichero, tal y como recoge la Ilustración 5-3.

```
Servidor de ficheros, al recibir mensaje con operación ActualizarFichero:
{
    /* Se ha proyectado el contenido del mensaje en formato empaquetado */
    para (todas las páginas incluidas en el mapa del MIE) {
        despl ← nº de página * tamaño_de_página
        bloque ← traducir despl. en fichero a bloque de disco
        EscribirDisco(bloque, dirección_de_la_página, tamaño_de_página)
    }
}
```

Ilustración 5-3: Algoritmo para actualizar el fichero de respaldo a partir de un mensaje incremental empaquetado (MIE).

Nótese que en la Ilustración 5-2 se ha simplificado deliberadamente la comunicación del servidor de ficheros con el disco, ya que en los sistemas basados en micro-núcleo, generalmente existe un proceso que ofrece los servicios de acceso al disco. En la Ilustración 5-3 se ha considerado que la función *EscribirDisco()* contiene el código para enviar un mensaje a este servidor y así escribir en el bloque correspondiente del disco la información definida por el par (*dirección_de_la_página*, *tamaño_de_página*).

Además, si se desea obtener mayor fiabilidad, los mensajes incrementales con la actualización pueden ser enviados a un grupo de servidores de ficheros distribuidos por varios nodos utilizando servicios para comunicación de grupo que deben ser incluidos en el servidor de comunicaciones.

Conclusiones

En este apartado se ha mostrado cómo se pueden implementar, apoyándose en los mensajes incrementales negativos, funciones para el establecimiento de puntos de recuperación que pueden ser incluidas en una librería a nivel de usuario.

Estas funciones, al igual que las de la implementación original, permiten que el programa de aplicación especifique cuáles son los datos críticos y reducir así la información incluida en cada punto de recuperación. Además, los mensajes incrementales negativos permiten incluir, en cada punto de recuperación, únicamente las páginas que han sido modificadas desde el punto de recuperación anterior.

5.3 Implementación de Manetho con los servicios del micro-núcleo

Las técnicas de tolerancia a fallos utilizadas en Manetho, así como su distribución entre los diferentes niveles del sistema, se han introducido en el 1.3.3. En la implementación original sobre el sistema V [Che88], el núcleo incluía una buena parte de los algoritmos de Manetho. En cambio, la especificación del micro-núcleo del capítulo anterior incluye únicamente servicios básicos, que no incluyen estos algoritmos, y que, por lo tanto, deberán ser incluidos en el nivel de utilidades del sistema. En este apartado se describe la estructura del soporte lógico apropiada para implementar los algoritmos de Manetho (ver anexo III) utilizando estos servicios.

El modelo de sistema que se utiliza en Manetho considera que la unidad de recuperación es un proceso. No obstante, en la implementación original se supuso que en cada nodo (estación de trabajo) se ejecutaba un único proceso de la aplicación tolerante a fallos y eso influyó sobre la estructura de la implementación.

En el caso de las RUs replicadas, este supuesto es perfectamente asumible en la arquitectura propuesta. La razón es que la replicación se lleva a cabo para aumentar la garantía de funcionamiento y, por lo tanto, cada réplica debe ejecutarse en un nodo diferente.

No obstante, es conveniente resaltar que el sistema propuesto desempeñaría las funciones que podría llevar a cabo un multiprocesador con memoria compartida y que, por lo tanto, el número de nodos sería reducido. En esta situación, no es razonable suponer que podrá dedicarse un nodo a cada proceso (RU), sino que, en general, habrá más de una RU por nodo, tal y como se muestra en la Ilustración 5-4.

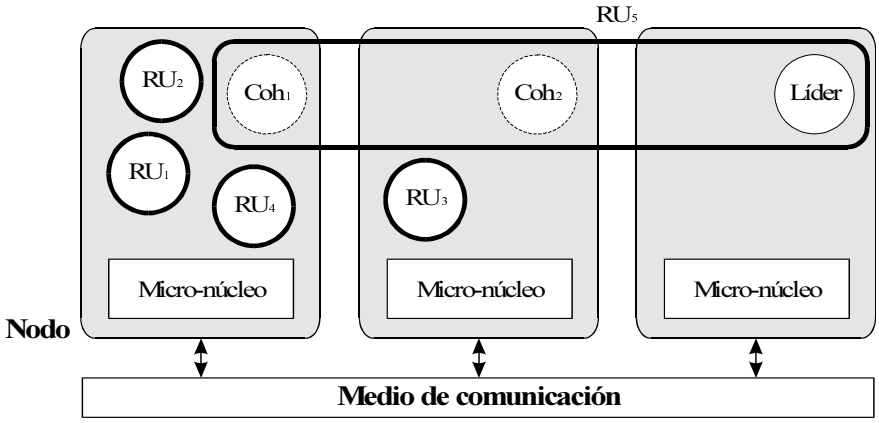

Ilustración 5-4: Ejemplo de distribución de RUs en la propuesta para la implementación de Manetho.

Teniendo en cuenta la estructura de datos de Manetho y cuándo se accede a cada elemento de la estructura, para implementar los algoritmos de Manetho se propone la estructura del soporte lógico que se muestra en la Ilustración 5-5. Los elementos

básicos de esta estructura aparecen con fondo blanco y son: el micro-núcleo, el servidor de comunicación entre nodos (SCom), el servidor de recuperación (SRec), el servidor de almacenamiento estable (SAE) y la librería incluida en el espacio de direcciones de cada proceso de la aplicación. Dentro de cada proceso habrá dos hilos dedicados a ejecutar código de la librería (H_{PdR} y H_{Rec}), cuya función se describe más adelante. Además, en la librería de las unidades de recuperación replicadas habrá un tercer hilo, H_{Msj}, que no es necesario en las RUs basadas en recuperación hacia atrás. El servidor de almacenamiento estable aparece fuera del área del nodo para resaltar que debe ser un componente modular redundante para aumentar su fiabilidad. En función de la implementación, tal y como se discute en el punto "Servidor de almacenamiento estable" (pág. 104), es posible que una de las instancias del componente se ejecute en el nodo, pero no es necesario.

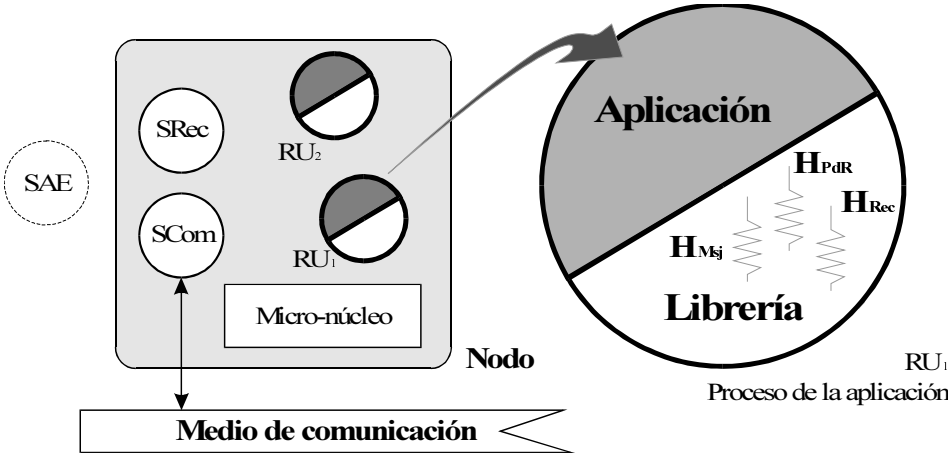

Ilustración 5-5: Estructura del soporte lógico para la implementación de los algoritmos de Manetho.

Es importante destacar que, a diferencia del servidor de recuperación, el servidor de comunicaciones no es un componente utilizado exclusivamente para la implementación de Manetho —al igual que tampoco lo es el micro-núcleo—.

La Tabla 5-1 recoge la distribución de los elementos básicos de la estructura de datos de Manetho entre los diferentes componentes lógicos que se acaban de introducir, y a continuación se describen las funciones desempeñadas por cada componente lógico en cada uno de los algoritmos básicos de Manetho.

Componente lógico	Elemento de la estructura de datos
Librería	*AG*
SRec	*registro de mensajes*
SAE	*P_AG* *registro de compromisos*

Tabla 5-1: Distribución de los elementos básicos de la estructura de datos de Manetho entre los diferentes componentes lógicos.

Protocolo de comunicaciones

Cuando se envía un mensaje, el código de la librería se encarga de añadirle una cabecera con el AG e invocar el servicio correspondiente del micro-núcleo. Por otra parte, durante la inicialización de la unidad de recuperación se habrá utilizado el servicio *EstablecerDestinoRegistro()* para que el micro-núcleo envíe una copia de los mensajes que transmita la RU. El servidor de recuperación es el encargado de mantener el registro de los mensajes enviados por todas las unidades de recuperación del nodo.

Al recibir un mensaje, el código de la librería lleva a cabo todas las acciones del algoritmo *Recibir* (ver anexo III) sin necesidad de interaccionar con ningún otro componente lógico, y lo mismo ocurre cuando se recibe información del mundo exterior. En cambio, cuando la aplicación envía información al mundo exterior, la librería interactúa con el servidor del almacenamiento estable. La librería le envía el identificador del mensaje para que sea almacenado en el registro de compromisos, y el grafo de antecedentes, para que sea añadido a la versión persistente.

Puntos de recuperación

El mecanismo básico para el establecimiento de puntos de recuperación ha sido descrito en el apartado 4.3.1. En este caso, el destino de los puntos de recuperación será el servidor de almacenamiento estable, que deberá gestionar la recepción de puntos de recuperación incrementales y admitir varias versiones de puntos de recuperación por cada proceso (al menos dos por proceso). Sin embargo, la gestión de puntos de recuperación incrementales negativos no resulta necesaria, ya que la recuperación de cualquier RU no fuerza la vuelta atrás de ninguna otra.

El protocolo de establecimiento de puntos de recuperación forma parte del código de la librería de cada unidad de recuperación. Para permitir la ejecución del código de la aplicación mientras se espera la confirmación del punto de recuperación, se utiliza un hilo dedicado, H_{PdR}, que se encarga de ejecutar este protocolo.

Protocolo de difusión

La difusión de los mensajes se lleva a cabo en el servidor de comunicaciones, de forma transparente para los emisores. Por otra parte, la librería del líder de un conjunto de unidades de recuperación replicadas se encarga de establecer el orden en el que la aplicación debe recibir los mensajes en todas las réplicas. La librería del resto de

unidades de recuperación de la *troupe* mantiene una cola privada de mensajes recibidos, de donde son consumidos por la aplicación en el orden indicado por el líder.

Puesto que la gestión de mensajes (tanto de aplicación como de orden) en la librería es asíncrona con respecto a la ejecución de la aplicación, es necesario utilizar un hilo dedicado a esta función, H_{Msj}. La Ilustración 5-6 muestra la estructura de la librería en una unidad de recuperación replicada.

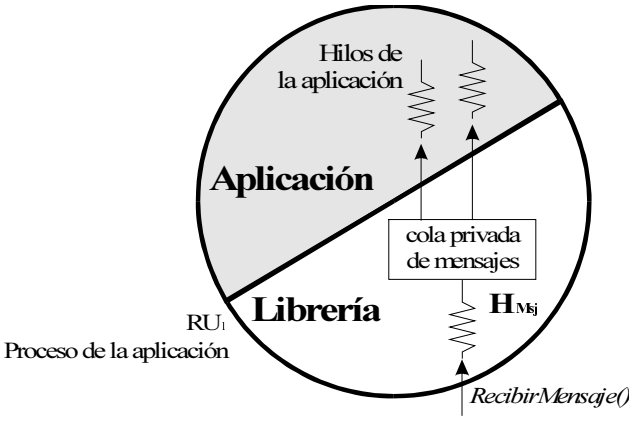

Ilustración 5-6: Estructura para la difusión de mensajes en unidades de recuperación replicadas.

Protocolo de recuperación

Cuando se produce un fallo en un nodo, todas las unidades de recuperación que estaban ejecutándose en él deben ser asignadas a otros nodos para continuar su ejecución. En el caso de las unidades basadas en vuelta atrás, la recuperación es iniciada por el servidor de recuperación del nodo que va a acoger a la unidad de recuperación, utilizando el servicio *RestablecerProceso()*. El micro-núcleo, tal y como se describe en 4.3.1, se apoya en el servidor de almacenamiento estable, que es quien le proporciona la información necesaria sobre el estado del proceso.

A continuación, el hilo H_{Rec}, que estará bloqueado esperando una notificación de vuelta atrás, ejecutará el resto del protocolo de recuperación (ver anexo III). El punto crítico de este algoritmo —desde el punto de vista del rendimiento— es la solicitud del grafo de antecedentes al resto de las unidades de recuperación del sistema, cuyo coste puede reducirse utilizando el servidor de recuperación como se describe a continuación.

Sean r_1 y r_2 dos RUs que, tras el fallo del nodo en el que residían, han sido destinadas al nodo A para continuar allí con su ejecución. Durante la recuperación, ambas deben obtener el grafo de antecedentes del resto de RUs del sistema. Sea r_3 una RU que reside en el nodo B. Si es la librería la que realiza directamente la petición del AG al resto de RUs, entonces r_3 recibirá dos solicitudes independientes y su AG será transmitido dos veces a través del medio de comunicación desde B hasta A. Para

evitarlo, el servidor de recuperación solicita los AGs a todas las RUs del sistema; y las librerías de las RUs que han vuelto atrás solicitan los grafos al servidor de recuperación de su nodo, evitando así el problema descrito. Nótese que la disminución del tráfico a través del medio de comunicación será tanto mayor cuanto mayor sea el número de RUs que son destinadas a un mismo nodo para recuperarse. Este número será elevado en sistemas con pocos nodos y un elevado número de unidades de recuperación, o en sistemas en los que se mantenga un nodo de reserva para, en caso de fallo de un nodo, poder asumir su trabajo.

La parte final del protocolo de recuperación, la repetición de la ejecución descrita en el grafo de antecedentes, es una parte compleja del algoritmo. Las operaciones relacionadas con el envío y recepción de mensajes, pueden ser llevadas a cabo por el código de la librería, que tiene acceso al grafo de antecedentes. En el caso de la recepción de mensajes, el hilo H_{Rec} se encarga de solicitar el reenvío de los mensajes a las RUs emisoras, para que estén disponibles cuando lo solicite la aplicación. Por supuesto, los hilos de la aplicación tendrán que bloquearse cuando intenten enviar o recibir información fuera de orden, para garantizar que la unidad de recuperación alcanza un estado consistente con el del resto del sistema. Toda esta sincronización adicional se realiza dentro de la librería, por lo que resulta transparente para los hilos de la aplicación.

No obstante, la solicitud de servicios del sistema presenta una complicación adicional relacionada principalmente con los identificadores de objetos (procesos o hilos, por ejemplo). Tal y como se discute en [Eln93, p. 54], existen tres tipos de llamadas al sistema.

- Las idempotentes (por ejemplo, obtener el identificador del hilo en ejecución), que pueden ser ejecutadas de nuevo durante la recuperación sin consideraciones adicionales.
- Aquéllas que no deben ser ejecutadas de nuevo, sino que se debe devolver el valor anterior al hilo de la aplicación (por ejemplo, obtener la hora del sistema). Este caso también es sencillo, basta con que se devuelva el valor almacenado en el grafo de antecedentes.
- Aquéllas que deben ser repetidas y deben devolver el mismo valor (por ejemplo, crear hilo debe devolver el mismo identificador de hilo). Esta funcionalidad puede ser ofrecida directamente por el micro-núcleo, como en el caso de la implementación original de Manetho, o se puede incluir código adicional en la librería que realice la traducción automática entre identificadores originales y los obtenidos durante la reejecución, como se hizo en [Rus94].

  No obstante, como este problema es común a todas las técnicas basadas en puntos de recuperación y simplifica considerablemente el proceso de recuperación, resulta adecuado incluirlo en el micro-núcleo. Para ello, el micro-núcleo tiene que garantizar que los identificadores no son reutilizados durante un periodo de tiempo suficientemente largo. Además de facilitar la recuperación, esta garantía también resulta ventajosa para el servicio de denominación de objetos y ha sido incluida en el sistema operativo Chorus, cuyo núcleo garantiza que un identificador existirá en un único nodo del sistema y que no será reutilizado [RAA+92].

Servidor de almacenamiento estable

En la implementación original de Manetho, el servicio de almacenamiento estable es ofrecido por un par de servidores dedicados, cada uno ejecutándose en una estación de trabajo con su propio disco.

Esta estructura es perfectamente asimilable en el sistema propuesto, y puede implementarse sin soporte adicional por parte del micro-núcleo. Para garantizar que ambos servidores mantienen copias idénticas de los datos es posible utilizar la técnica de redundancia modular, haciendo que ambos servidores procesen todas las operaciones en el mismo orden. En este caso, los servidores de comunicaciones de los diferentes nodos del sistema deberán coordinarse para proporcionar servicios adecuados para la comunicación de grupo, garantizando que las operaciones llegan a ambos servidores y que llegan en el mismo orden. Además, el servidor de comunicaciones deberá procesar las respuestas de los servidores, para que los clientes reciban la respuesta de uno sólo de ellos. Si ambos están funcionando, las respuestas son equivalentes y se puede utilizar la primera que esté disponible, ya que en Manetho se considera que los nodos garantizan silencio tras la avería. Por el contrario, si uno de ellos ha fallado, la respuesta del otro debe llegar al cliente.

Para finalizar la propuesta de implementación, la Tabla 5-2 resume las actividades que debe llevar a cabo cada uno de los componentes lógicos empleados para implementar los algoritmos de Manetho.

Protocolo	**Componente lógico**			
	Librería	**SRec**	**SAE**	**SCom**
Comunicaciones	*gestión del AG*	*gestión del registro de mensajes*	*gestión del P_AG* *gestión del registro de compromisos*	
Puntos de recuperación	*(H_{PdR})* *coordinación*		*gestión de los PdR*	
Difusión	*(H_{Msj})* *garantía de orden* *gestión de errores*			*difusión sin garantías*
Recuperación	*(H_{Rec})* *recuperar AG desde el SRec local* *repetir ejecución*	*restaurar el estado de las RUs* *recolección de AGs para RUs locales* *reenvío de mensajes*		

Tabla 5-2: Resumen de la función de los diferentes componentes lógicos en cada uno de los cuatro algoritmos básicos de Manetho.

Conclusiones

En el presente apartado se ha comprobado que los servicios del micro-núcleo especificado en el capítulo anterior permiten implementar los algoritmos de Manetho. A diferencia de los dos primeros apartados, en este apartado se ha considerado la implementación del sistema completo para comprobar que la especificación permite desarrollar aplicaciones que utilicen varias técnicas de tolerancia a fallos combinadas.

Otra diferencia significativa respecto a los casos anteriores es que la implementación original de Manetho incluye código en el núcleo del sistema operativo. Los servicios añadidos al núcleo incluyen una parte de los algoritmos de Manetho, por lo que resultan muy específicos y deberían ser modificados para implementar otras técnicas de tolerancia a fallos. Por ejemplo, la gestión del registro de mensajes se realiza en el núcleo del sistema operativo, y ni el nivel de utilidades ni el de aplicación pueden configurar cómo se gestiona este registro. Apoyándose en estos servicios específicos, la implementación original de Manetho consigue elevadas prestaciones, comparándose favorablemente respecto a otros sistemas.

En cambio, la implementación propuesta en este apartado se apoya en servicios básicos, que pueden emplearse para implementar diferentes técnicas de tolerancia a fallos. Frente a la implementación original de Manetho, la implementación sobre el micro-núcleo propuesto tiene la ventaja de que los algoritmos se implementan totalmente en el nivel de utilidades, sin necesidad de modificar el núcleo del sistema operativo, reduciéndose el tiempo y la complejidad de la implementación.

Es posible que la implementación propuesta en este apartado, si se implantara sobre un soporte físico de prestaciones similares a las del de la implementación original, presentara un rendimiento ligeramente inferior. No obstante, este hecho no descalifica a la interfaz propuesta, puesto que la mayor eficiencia de la implementación original se debería a la inclusión de servicios específicos en el núcleo del sistema operativo. Además, la diferencia en rendimiento debería ser pequeña, ya que los servicios incluidos en el micro-núcleo aportan las ventajas de dos de las tres optimizaciones más importantes de Manetho:

- El uso de copia en escritura para la generación del registro de mensajes. La generación de la información para el registro, puesto que se realiza en el micro-núcleo, puede implementarse utilizando esta técnica que, por otra parte, resulta también adecuada para la implementación de las operaciones incrementales, es muy conocida, y está muy extendida.

  Además, si se utiliza la opción para la generación asíncrona de la información, la interferencia con el proceso emisor será del mismo orden que en la implementación original, ya que el emisor únicamente se ve bloqueado durante la operación de copia en escritura.

- Establecimiento de puntos de recuperación incrementales y concurrentes. Ambas opciones están disponibles en la interfaz especificada.

La tercera optimización se refiere a la implementación del protocolo de difusión, que no forma parte de las funciones que debe desempeñar el micro-núcleo, sino que debe incluirse en niveles superiores del sistema. Precisamente, la existencia de protocolos de difusión específicos como el de Manetho ha sido una de las razones

principales para no incluir la comunicación de grupo como parte del micro-núcleo, tal y como se justifica en el capítulo 3.

5.4 Servicios transaccionales

En este apartado se describe la forma de estructurar aplicaciones que utilicen transacciones para ofrecer servicios tolerantes a fallos. El apartado comienza describiendo brevemente el concepto de servicio transaccional y continúa con la descripción de la estructura básica de este servicio. Después, se muestra cómo se puede completar esta estructura para que el servicio sea tolerante a fallos y admita transacciones concurrentes.

Concepto de servicio transaccional

Un servicio transaccional, siguiendo por ejemplo la definición de [CDK96, p. 361], consiste en la extensión de un servicio para proporcionar el acceso a sus recursos mediante transacciones. Por ejemplo, la realización de operaciones bancarias, el acceso a una base de datos o el acceso a un servicio de ficheros. Para construir las transacciones atómicas es necesaria la cooperación del cliente y del servicio. El cliente agrupa sus operaciones en transacciones, utilizando para ello las operaciones *BEGIN* y *COMMIT*, descritas en el apartado 1.1.2.3. Por su parte, el servicio transaccional garantiza las propiedades de atomicidad, consistencia, aislamiento y durabilidad de todo el grupo de operaciones que forman parte de la transacción.

Para garantizar la propiedad de aislamiento, el servicio de transacciones debe sincronizar las operaciones de las transacciones de forma que el resultado final de todas ellas sea equivalente al que se habría producido si se hubieran ejecutado de una en una en algún orden determinado. En este caso se dice que la ejecución de las transacciones es serializable. Aunque el objetivo de un servicio transaccional debe ser el de maximizar la concurrencia para optimizar las prestaciones, a continuación se comienza por considerar la implementación de un servicio transaccional que no permite concurrencia; pero que ilustra el modo básico de utilizar las operaciones incrementales. Posteriormente se extiende este modelo para permitir la ejecución de transacciones concurrentes.

Estructura del servicio transaccional básico

La estructura utilizada para implementar este servicio consta de un único proceso, tal y como se muestra en la Ilustración 5-7, donde las flechas simples indican la relación cliente-servidor con la punta señalando al servidor, y las flechas gruesas indican el flujo de datos entre copias de la información. El proceso S mantiene dos regiones de memoria en su espacio de direcciones; la primera, C_T, es la copia de trabajo, sobre la que se realizan las operaciones de la transacción en curso. La segunda, D, contiene la copia original de los datos.

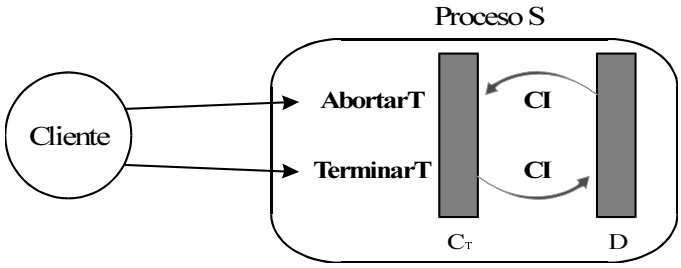

Ilustración 5-7: Estructura del servicio transaccional básico.

La interfaz ofrecida por el servidor S a los clientes del servicio transaccional se recoge en la Ilustración 5-8. Las operaciones *ComenzarTransacción(), TerminarTransacción()* y *AbortarTransacción()* son equivalentes a las operaciones *BEGIN*, *COMMIT* y *ABORT* descritas en el apartado 1.1.2.3. No obstante, ya que sólo puede haber una transacción en marcha, el aborto de la transacción solamente tendrá lugar a instancias del cliente o si el servidor falla (el modelo de fallos se discute más adelante, en este mismo apartado). Aunque no aparecen en la ilustración, S ofrecerá, además, las operaciones propias del servicio que implemente, como operaciones para la lectura y escritura en ficheros.

ComenzarTransacción()
Si no había otra transacción en marcha, anota que ha comenzado la transacción y devuelve éxito al cliente.
Errores: *transacción ya iniciada*
TerminarTransacción()
El servidor convierte en permanentes los cambios realizados a la copia de trabajo durante la transacción.
Errores: *transacción no iniciada*
AbortarTransacción()
El servidor desecha los cambios realizados durante la transacción en marcha.
Errores: *transacción no iniciada*

Ilustración 5-8: Interfaz del servicio transaccional básico.

Inicialmente, la copia de trabajo (C_T) y los datos (D) son iguales.[1] Además, respecto a las operaciones incrementales, todas las páginas de la copia de trabajo están marcadas como no-modificadas. Por lo tanto, cuando el cliente solicita que se inicie una nueva transacción (operación *ComenzarTransacción*), S simplemente comprueba que no haya una transacción previa en marcha. A partir de ese momento, el cliente

[1] Para ello, S creará las dos regiones, inicializará una de ellas y realizará una primera copia incremental sobre la segunda.

puede invocar cualquiera de las operaciones del servicio y los cambios se realizan sobre la copia de trabajo, en su espacio local de direcciones.

Posteriormente, cuando el cliente invoca la operación *TerminarTransacción()*, el proceso S realiza una copia incremental de C_T a D utilizando la llamada al micro-núcleo *CopiarVirtualmente()* con la opción *incremental*. Alternativamente, el cliente puede invocar la operación *AbortarTransacción()*. Entonces, S utiliza la opción *incremental negativa* en la llamada a *CopiarVirtualmente()*, para descartar los cambios hechos a C_T y cancelar así los efectos de la transacción en curso.

La gran ventaja que aportan las operaciones incrementales y la estructura propuesta es la gran sencillez con que puede ser transformado un servicio cualquiera en servicio transaccional. El único requerimiento básico es que C_t y D sean regiones del espacio virtual de direcciones. A partir de ahí, las operaciones incrementales —copia virtual incremental en este caso, y el intercambio de mensajes en las estructuras que se describirán a continuación— permiten una implementación de las tres operaciones básicas del servicio transaccional independiente del tipo de servicio ofrecido.

Estructura del servicio transaccional tolerante a fallos

Una vez propuesta la estructura básica, es posible modificarla para conseguir, de forma sencilla, un servicio transaccional tolerante a fallos. Para tolerar fallos que afecten a un único nodo del sistema, es suficiente con utilizar un segundo proceso, R en este caso, que mantenga una copia de respaldo de los datos originales (región D). Esta copia de respaldo puede mantenerse tanto en una región del espacio de direcciones de R como en un fichero. Esta última alternativa es similar a la descrita en el apartado 0, así que en este caso se considerará que se mantiene en memoria.

La estructura del servicio transaccional tolerante a fallos se muestra en la Ilustración 5-9, en la que se muestra el proceso S, que mantiene la estructura del servicio transaccional básico, y el nuevo proceso, R, que actúa como servidor de copias de respaldo para S. Tal y como se recoge en la ilustración, S y R deben residir en nodos diferentes con la finalidad de que el servicio transaccional pueda tolerar fallos que afecten al nodo A o al nodo B, siempre que no afecten a los dos simultáneamente. En cambio, el cliente se ha representado fuera de ambos nodos porque su localización no influye sobre el modelo básico.

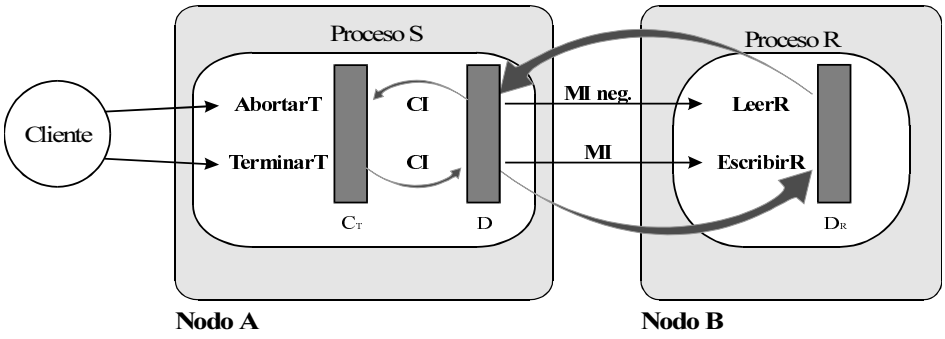

Ilustración 5-9: Estructura del servicio transaccional tolerante a fallos.

La interfaz ofrecida por S a los clientes se mantiene sin cambios, aunque ahora S utiliza las operaciones ofrecidas por R, y que se recogen en la Ilustración 5-10. Las operaciones *AbrirRegión()* y *EliminarRegión()* son invocadas mediante mensajes corrientes, mientras que los otros dos servicios son invocados mediante mensajes incrementales. Esta interfaz supone que cada proceso S tiene su propio proceso R; no obstante, resultaría sencillo extender la interfaz propuesta para que R gestionase más de una región de respaldo y así pudiera servir de respaldo a varios procesos simultáneamente.

AbrirRegión(origen, tamaño)
Crea una nueva región de memoria virtual del tamaño indicado y la llena con los datos especificados en origen.
EliminarRegión()
La región de memoria virtual es devuelta al núcleo del sistema operativo.
LeerRegión(identificador)
El mensaje recibido es un mensaje incremental negativo. El servidor de regiones de respaldo responde con otro mensaje incremental en el que se incluyen las páginas indicadas en el mensaje.
EscribirRegión(identificador)
El mensaje recibido es un mensaje incremental. La recepción incluye la proyección de los datos, por lo que el contenido de la región habrá sido actualizado.

Ilustración 5-10: Interfaz ofrecida por el proceso R.

La operación *AbrirRegión()* hace que R prepare una nueva región de memoria virtual del tamaño especificado, cuyo contenido inicial dependerá del parámetro *origen*. Por ejemplo, *origen* puede indicar que la región debe ser llenada con ceros, o puede indicar el fichero que debe ser utilizado para inicializarla. Cualquiera que sea su origen, al establecerse el servicio transaccional, C_t, D y D_R son iguales.[2] Además, respecto a las operaciones incrementales, todas las páginas de C_T y D están marcadas como no-modificadas. De esta forma, cuando el cliente solicita que se inicie una nueva transacción (operación *ComenzarTransacción*), S simplemente comprueba que no haya una transacción previa en marcha, tal y como hacía en el servicio básico. Igualmente, a partir de ese momento, el cliente puede invocar cualquiera de las operaciones de S y los cambios se realizan sobre la copia de trabajo (C_T) sin necesidad de comunicación con R.

Posteriormente, cuando el cliente invoca la operación *TerminarTransacción()*, el proceso S, antes de responder, envía un mensaje incremental a R para invocar la operación *EscribirRegión()*. El mensaje contiene todas las páginas modificadas y, cuando lo recibe R, como resultado de la proyección de las páginas incluidas en el mensaje, la copia de respaldo de los datos (D_R) queda actualizada. Entonces S

[2] Para ello, *Servidor* usará el parámetro *origen* de *AbrirRegión()* o, tras inicializar la copia de trabajo, enviará un primer mensajes incremental a *Respaldo*, tal y como se describe a continuación.

responde al cliente, confirmándole la operación. (Nótese que S continúa actualizando D, con lo que D y D_R son dos réplicas de los mismos datos).

En cambio, si el cliente invoca la operación *AbortarTransacción()*, S no necesita contactar con R, sino que continúa restaurando C_T a partir de D.

Esta estructura, a diferencia de la estructura básica, permite tolerar fallos tanto del proceso S como del proceso R, siempre que se garantice la parada tras la avería. Para ello, tal y como se ha expuesto anteriormente, es necesario que ambos procesos residan en nodos diferentes, sin ninguna dependencia entre ellos[3]. Así, si falla R, siempre es posible crear una nueva instancia del proceso, actualizar su estado utilizando la región D y continuar con la transacción en curso. En cambio, si el que falla es el proceso S, entonces las modificaciones realizadas durante la transacción en curso se perderán. No obstante, todos los cambios debidos a transacciones anteriores se conservan, ya que, tras crear una nueva instancia de S, éste puede solicitárselas a R para restaurar C_T y D, y continuar ofreciendo el servicio.

Estructura del servicio transaccional concurrente

Finalmente, para permitir la ejecución concurrente de transacciones, es necesario modificar la estructura del servicio tolerante a fallos. No obstante, los servicios ofrecidos por el servidor de respaldo (proceso R) permanecen sin cambios, así como su estructura. En cambio, tal y como se muestra en la Ilustración 5-11, se sustituye el proceso S por un conjunto de procesos de dos tipos, ST y T_i. Por una parte, el proceso ST mantiene la copia original de los datos (D) y se encarga de coordinar las transacciones. Por otra, cada proceso T_i mantiene una copia de trabajo (C_{Ti}) y ofrece los servicios a los clientes. Cada T_i gestiona una única transacción de principio a fin (de la transacción), cuyas modificaciones va aplicando sobre su copia privada de los datos, y utiliza los servicios ofrecidos por ST para: 1) coordinarse con otras transacciones concurrentes 2) confirmar o cancelar los efectos de su transacción.

Por lo tanto, el número máximo de transacciones concurrentes es igual al número de procesos T_i. La descripción siguiente considera que los procesos T_i existen antes de comenzar la transacción, que al finalizar ésta aguardan una nueva solicitud para iniciar una nueva, y que los clientes solicitan directamente a un T_i libre el inicio de la transacción. Alternativamente, los clientes podrían solicitar el inicio de una transacción a ST (o a otro proceso especialmente dedicado a esta función) y éste crear una instancia de T_i encargada de atender al cliente durante la nueva transacción (modelo del despachador y los trabajadores).

A pesar de las diferencias en la estructura, el esquema básico es análogo al del caso anterior, y el papel de las diferentes copias de los datos[4] (C_{Ti}, D y D_R) también es el mismo. Es decir, cada C_{Ti} contiene las modificaciones tentativas de la transacción atendida por T_i, y D junto con D_R contienen la copia permanente de los datos.

---

[3] Un ejemplo de dependencia consistiría en que el nodo A implementase memoria virtual y almacenase las páginas no residentes en el nodo B. O también, que el ejecutable de *Respaldo* no fuese accesible desde otro nodo diferente a B.

[4] D_R no se muestra en la ilustración; pero forma parte del espacio de direcciones del proceso R.

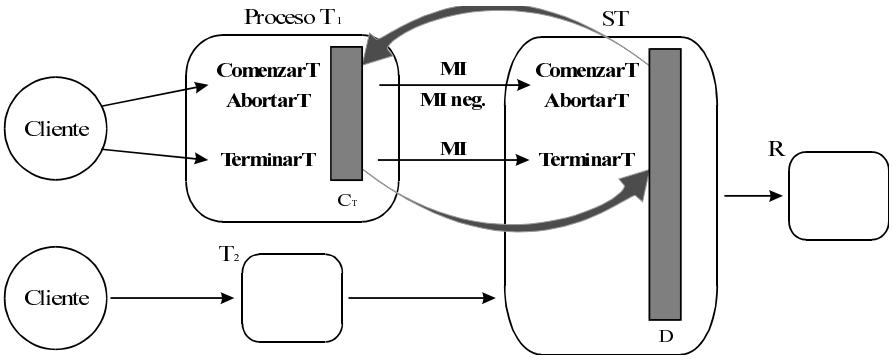

Ilustración 5-11: Estructura del servicio transaccional para dos transacciones concurrentes.

El funcionamiento básico del servicio es como sigue. Cuando comienza una transacción (a instancias del cliente), el proceso T_i invoca la operación *ComenzarTransacción()* del proceso ST. Éste anota que se va a iniciar una nueva transacción y responde a T_i con un mensaje incremental que contiene las páginas que han sido modificadas en D. Estas páginas contienen todas las modificaciones debidas a transacciones que han terminado desde que comenzó la transacción anterior de T_i, que es cuando éste actualizó su estado por última vez (salvo que hubiera sido abortada, tal y como se describe a continuación). Así T_i comienza la transacción con su copia de trabajo (C_{Ti}) actualizada respecto a D y D_R.

Cuando el cliente solicita abortar la transacción en curso, T_i invoca la operación *AbortarTransacción()* de ST mediante un mensaje incremental negativo. Éste anota el hecho y responde con las páginas solicitadas en el mensaje (que serán las que hayan sido modificadas como resultado de las operaciones llevadas a cabo durante la transacción).

En cambio, si el cliente decide terminar la transacción, T_i invoca la operación *TerminarTransacción()* de ST mediante un mensaje incremental, que contiene las páginas modificadas durante la transacción. La región D habrá sido actualizada como resultado de recibir el mensaje incremental, ST anota el hecho y, antes de responder a T_i, invoca la operación *EscribirRegión()* para actualizar la copia D_R en el proceso R. Entonces responde a T_i quien, a su vez, notifica al cliente que la transacción ha terminado con éxito.

Hasta este momento se ha evitado deliberadamente toda referencia al control de concurrencia. La razón es que la estructura propuesta y las operaciones incrementales constituyen un soporte adecuado tanto para las estrategias optimistas como para las pesimistas. En ambos casos, el proceso ST debe ser el responsable de este control, ya que es el único que dispone de información sobre las diferentes transacciones en marcha.

Si se opta por una estrategia optimista, cada T_i puede atender todas las operaciones del cliente sin contactar con ST. Si la transacción aborta a instancia del cliente, T_i descarta las modificaciones realizadas; pero, según se ha visto anteriormente, no es necesaria ninguna acción adicional ni por su parte ni por la de ST. En cambio, a la hora

de terminar la transacción, el proceso ST utiliza la opción *cabecera* en la llamada a *RecibirMensaje()*, para poder validar la transacción[5] antes de convertir los cambios en permanentes. Tras la validación, si la transacción puede terminar, ST utiliza *ProyectarMensaje()* para actualizar D y, tal y como se ha descrito anteriormente, envía un mensaje incremental a R para actualizar D_R antes de responder a T_i.

Si se opta por una estrategia pesimista, los procesos T_i deben obtener de ST el cerrojo correspondiente antes de acceder a la página (o al conjunto de páginas protegidas por el cerrojo). Éste es el esquema seguido en el servidor transaccional de ficheros que se describe en el capítulo siguiente, donde se describe con detalle el protocolo para el control pesimista de la concurrencia. En este caso, si la transacción aborta a instancia del cliente, ST recupera todos los cerrojos obtenidos por T_i en nombre de la transacción. Si el cliente decide terminar la transacción ST puede recibir directamente el mensaje sobre la región D, ya que los cerrojos garantizan que no existen conflictos entre transacciones concurrentes. No obstante, si es necesario aumentar la fiabilidad del servicio, ST puede utilizar la opción *cabecera* al recibir el mensaje, tal y como ocurría en el caso del control optimista. De esta forma, ST puede comprobar que, efectivamente, las páginas modificadas[6] por T_i se corresponden con las páginas protegidas por los cerrojos que posee.

Tal y como se desprende de la descripción realizada, el control de concurrencia añade un nuevo requisito sobre los servicios que utilicen esta estructura: la granularidad en el control de concurrencia es la página de memoria virtual. Esta restricción es importante; pero, en muchos casos es posible adaptar las estructuras de datos del servicio para que el nivel de concurrencia resulte adecuado. Un ejemplo de esta adaptación lo constituye el servicio de ficheros transaccional, donde la disposición de los datos en memoria se ha diseñado para permitir cerrojos con granularidad de página.

Conclusiones

En este apartado se especifica una nueva estructura para implementar servicios transaccionales, que es independiente del tipo de servicio ofrecido (de ficheros, operaciones bancarias, acceso a una base de datos, ...), disminuyendo el tiempo necesario para, una vez implementada la estructura especificada, convertir en transaccional un servicio determinado. Esta misma estructura también se adapta a diferentes protocolos de coordinación de transacciones, por lo que en cada caso se puede utilizar el que mejor se ajuste al servicio concreto que se vaya a implementar. Además, el módulo de respaldo puede ser utilizado en la implementación de otras técnicas de recuperación hacia atrás con sólo extender su interfaz para que sea capaz de gestionar varios clientes con varias regiones de respaldo.

---

[5]Para validar la transacción, ST necesita información sobre las páginas leídas y modificadas por T_i. Aunque el mensaje incremental no contiene información sobre las páginas leídas, T_i puede recoger esta información utilizando el servicio *ConsultarEstado()* y enviársela a ST.

[6] Sería posible igualmente comprobar que sólo ha accedido (tanto para lectura como para escritura) a páginas para las que ha obtenido el cerrojo. Para ello, T_i debería recoger la información sobre C_{Ti}, utilizando la función *ConsultarEstado()*, y enviársela a ST. O, alternativamente, podría realizar esta comprobación él mismo, antes de enviar el mensaje incremental con las modificaciones.

Esta estructura exige que las estructuras de datos del servicio se organicen de forma adecuada, ya que la unidad de transferencia y de control de concurrencia es la página. Estas características hacen que esta estructura no resulte adecuada para determinados servicios, como aquéllos en el que los objetos ocupen pocos *bytes* y donde la copia de una página completa representaría una sobrecarga excesiva.

No obstante, para evaluar la estructura propuesta es necesario implementar con ella diferentes servicios y así obtener información suficiente para realizar un análisis más detallado. Con este propósito se ha diseñado y se ha implementado un servicio transaccional de ficheros que muestra la factibilidad de utilizar esta estructura en aplicaciones reales. (El diseño, la implementación y los primeros resultados obtenidos se discuten en el capítulo 6).

5.5 Revisión de otras técnicas y sistemas tolerantes a fallos

Aunque tanto durante la propuesta de la arquitectura como durante la especificación de la interfaz del micro-núcleo se han tenido en cuenta las necesidades de todas las técnicas y sistemas descritos en el capítulo 1, se considera que los casos examinados son suficientemente representativos de todos ellos. A continuación se justifica esta afirmación para cada uno de ellos, comenzando por las técnicas de tolerancia a fallos.

Técnicas de tolerancia a fallos

- De las técnicas de recuperación hacia atrás, tanto los puntos de recuperación combinados como los registros han sido considerados en Manetho (apartado 5.3). Los puntos de recuperación independientes han sido abordados en el apartado 5.2, donde se implementaban desde una librería a nivel de usuario.

  Los bloques de recuperación, a pesar de no haber sido analizados en este capítulo, también pueden beneficiarse de los servicios especificados para el micro-núcleo. En concreto, podrían utilizarse operaciones de copia virtual (con la opción incremental), tanto para establecer el punto de recuperación antes de ejecutar un módulo, como para realizar la vuelta atrás en caso de fallar el test de aceptación. Alternativamente, la copia de los datos podría ser almacenada por un componente externo al proceso, como el proceso de respaldo descrito en el apartado 5.4.

- De las técnicas basadas en redundancia modular, las réplicas pasivas han sido consideradas en el apartado 5.1 y las réplicas activas en el 5.3. Por otra parte, desde el punto de vista de las necesidades al nivel del micro-núcleo, tanto la programación de N versiones como la programación de N versiones autocomprobantes son muy similares a las réplicas activas (ver apartado 1.1.3.2) y pueden implementarse, como ellas, apoyándose en los servicios de comunicación de grupo que deben ser incluidos en el servidor de comunicaciones.

Sistemas tolerantes a fallos

- En el caso del sistema Integrity S2, la dependencia del soporte físico es demasiado importante como para poder realizar una implementación equivalente apoyándose únicamente en los servicios del micro-núcleo. No obstante, la conveniencia de no exigir servicios específicos para tolerancia a fallos al soporte físico ha sido suficientemente justificada en los dos primeros apartados del capítulo 2.
- El mecanismo Sentry ofrece al usuario la posibilidad de activar y desactivar de forma selectiva diferentes políticas de detección de errores y de recuperación del sistema. Este mecanismo resulta especialmente adecuado en el caso de Mach, donde la emulación de Unix se realiza mediante un servidor monolítico; pero no resultaría extrapolable directamente al sistema propuesto en este trabajo, donde los servicios para la gestión de procesos y de memoria son ofrecidos directamente por el micro-núcleo, y no por un proceso de emulación en el nivel de utilidades.

  No obstante, la finalidad del mecanismo Sentry es proporcionar tolerancia a fallos transparente a las aplicaciones y de ahí la inclusión de políticas en el proceso de emulación de Unix. Las técnicas de tolerancia a fallos en que se basan estas políticas son los puntos de recuperación y los registros, que también se utilizan en Manetho y que sí han considerado en este capítulo.
- Delta-4, tal y como se describe en el apartado 1.3.4, permite la ejecución de componentes lógicos replicados en distintos nodos de una red de computadores. De las tres técnicas de replicación de componentes que se proponen, la replicación pasiva es muy similar a la utilizada en el Sistema Avanzado de Automatización, cuya implementación se discute en el apartado 5.1. La replicación semi-activa es una variante del modelo líder/seguidor, similar a la utilizada en Manetho, cuya implementación se considera en el apartado 5.3. Finalmente, la replicación activa se propone como técnica para implementar el almacenamiento estable de Manetho y, como allí se discute, se basa en servicios para la comunicación de grupo que deben ser incluidos en el servidor de comunicaciones.

5.6 Conclusiones

En este capítulo se ha comprobado que los servicios especificados en el capítulo 4 cumplen las condiciones exigidas de versatilidad, eficiencia y completitud.

Se considera que se ha conseguido un nivel de versatilidad y completitud adecuado, ya que ha sido posible especificar cómo implementar, utilizando estos servicios, diversas técnicas de tolerancia a fallos, que hemos considerado representativas de las descritas en el apartado 1.1. Se ha mostrado cómo implementar técnicas basadas tanto en vuelta atrás como en redundancia modular. De entre las primeras, se han incluido las transacciones y los puntos de recuperación, incluyendo un caso con puntos de recuperación independientes, y otro con una técnica híbrida, combinando puntos de

recuperación coordinados con registros de mensajes. De entre las basadas en redundacia modular, se ha incluido un caso con réplicas pasivas y otro con réplicas activas utilizando el modelo líder-cohortes.

Respecto a la eficiencia, este conjunto de servicios ofrece a los niveles superiores del sistema la posibilidad de beneficiarse de las técnicas aplicables en las implementaciones a nivel del núcleo, como son la copia en escritura, la actualización incremental de copias de la información, y los puntos de recuperación incrementales y concurrentes. Además, los servicios también permiten aprovechar las características específicas de las aplicaciones para aumentar la eficiencia del sistema, ya que los niveles de utilidades y aplicaciones incluyen las decisiones sobre qué datos se copian y cuándo.

Capítulo 6

Primeros resultados y evaluación

En este capítulo se describen los primeros resultados obtenidos tras la implementación de los mensajes incrementales y un servicio transaccional de ficheros. Se han escogido estos servicios por constituir una de las aportaciones más significativas del presente trabajo de investigación. Así, mientras que la factibilidad de implementar y de utilizar en aplicaciones reales otros servicios del micro-núcleo ya ha sido probada en otros sistemas, resulta necesario comprobar esta misma factibilidad para las operaciones incrementales. Siguiendo con este mismo razonamiento, de los sistemas cuya implementación se discute en el apartado anterior, se ha optado por implementar un servicio de ficheros que utilizase la estructura propuesta para un servicio transaccional concurrente y tolerante a fallos.

El capítulo está estructurado en dos partes bien diferenciadas. En el primer apartado se describe el estudio de las prestaciones de los mensajes incrementales. En el segundo, se comienza describiendo la implementación del servicio transaccional de ficheros, y se continúa con el estudio de sus prestaciones.

6.1 Mensajes incrementales

Los mensajes incrementales han sido implementados utilizando el sistema operativo VSTa [Val] como caso de ejemplo. VSTa es un sistema operativo basado en micro-núcleo que incorpora las funciones básicas descritas en el apartado 2.4, es decir, gestión de procesos y memoria, comunicación entre procesos mediante paso de mensajes y gestión básica de interrupciones y excepciones. Aunque en el anexo IV se describen con detalle los aspectos más relevantes de la implementación de los mensajes incrementales, a continuación se destacan, brevemente, algunos aspectos básicos que resultan necesarios para el análisis de resultados de este capítulo.

Para conseguir una implementación eficiente de los mensajes incrementales se ha utilizado la técnica de copia en escritura. Aunque VSTa utiliza esta técnica para copiar procesos (implementar la operación *fork*), el mecanismo en que se basa no resulta adecuado para las operaciones incrementales y, por lo tanto, ha sido necesario añadir un nuevo mecanismo de copia en escritura que, además, fuera compatible con la gestión de memoria virtual de VSTa.

Los servicios para el intercambio de mensajes están diseñados para la comunicación eficiente entre clientes y servidores, por lo que el servicio para enviar un mensaje bloquea al emisor (cliente) hasta que el receptor (servidor) responde. A la hora de implementar los mensajes incrementales negativos se ha aprovechado esta característica para permitir el intercambio de un mensaje incremental negativo mediante una única interacción entre cliente y servidor, tal y como se sugiere en el apartado 4.1.3.

Otro aspecto relevante es que VSTa ha sido diseñado para permitir multiprocesamiento simétrico. La sincronización de hilos en el micro-núcleo se realiza utilizando cerrojos y semáforos, y se permite la expulsión de hilos que estén ejecutando código del micro-núcleo, salvo que estén en posesión de un cerrojo. Como se verá a lo largo de este capítulo, los mecanismos de sincronización en el micro-núcleo tienen una influencia muy importante sobre las prestaciones de las operaciones incrementales.

Finalmente, la versión utilizada de VSTa (1.5.2) no ofrece servicios para transferir mensajes entre procesos que residan en nodos diferentes, por lo que se han estudiado casos en los que todos los procesos residen en el mismo nodo, lo cual resulta suficientemente adecuado para analizar el funcionamiento de las operaciones incrementales.

En este apartado se describen dos estudios diferentes. El primero contiene resultados detallados sobre las prestaciones de los mensajes incrementales, estudiándose la variación del coste en función del tamaño de las regiones de memoria virtual involucradas en el paso de mensajes y del número de páginas modificadas entre mensajes sucesivos. En el segundo se estudia la sobrecarga debida al establecimiento de puntos de recuperación dirigidos por el usuario utilizando mensajes incrementales.

6.1.1 Estudio básico de las prestaciones

Al realizar el estudio, las primeras medidas mostraron que la interrelación entre mensajes incrementales y el mecanismo de memoria virtual es tan estrecha que el factor que más influye sobre el coste de estos mensajes es la sincronización con el proceso encargado del robo de páginas. Por esta razón, con la finalidad de poder evaluar otros factores que también influyen sobre el coste de las operaciones incrementales, se han realizado dos estudios complementarios; el primero sobre datos obtenidos con el robo de páginas desactivado y el segundo sobre datos obtenidos con él activado.

En todos los casos se han tomado las medidas utilizando dos procesos residentes en el mismo nodo. El proceso principal mantiene la copia activa de los datos y realiza escrituras sobre un conjunto determinado de páginas. El segundo proceso es, precisamente, el proceso de respaldo que se ha descrito en el capítulo anterior y que se ha utilizado para implementar el servicio transaccional de ficheros que se describe en este mismo capítulo. Este proceso está continuamente esperando la llegada de mensajes incrementales (operación *EscribirR*) o de mensajes incrementales negativos (*LeerR*) y los atiende inmediatamente, sin ninguna otra actividad adicional. El proceso principal actúa como cliente del de respaldo, enviándole los mensajes correspondientes.

Las medidas han sido tomadas en un PC, con procesador 486DX2-66, 16 MB de memoria y una partición de intercambio de 216 MB, alojada en un disco duro conectado a una controladora IDE. El disco tiene un tiempo de búsqueda medio de 1,92 ms. y una velocidad de transferencia de 1210 KB/s.

Medidas sin robo de páginas

La Ilustración 6-1 recoge los resultados del primer experimento, que analiza la influencia del tamaño de la región de memoria virtual sobre el coste de los mensajes incrementales. Para cada tamaño de región se han enviado 50 mensajes consecutivos y se ha calculado el valor medio, que se muestra en la ilustración. Así, la desviación estándar sirve como medida de la variabilidad del valor obtenido

Entre cada dos mensajes consecutivos el proceso principal realiza un número determinado de modificaciones sobre páginas de la región seleccionadas al azar (serie de datos etiquetada como *aleatorio* en la ilustración) o sobre las primeras páginas de la región de memoria virtual. En la ilustración se muestra el caso de 10 páginas modificadas entre mensajes sucesivos.

Los resultados obtenidos para el caso secuencial muestran que el tamaño de la región de memoria virtual tiene poca influencia sobre el coste de la operación, que se mantiene entre los 90 y los 100 ms. para todos los tamaños. En cambio, los resultados para el caso aleatorio presentan una variación notable con el tamaño de la región de memoria virtual, produciéndose un descenso continuo en el coste de la operación a partir de 50 páginas, en contra de lo que cabría esperar, es decir, un ligero aumento del coste a medida que aumenta el tamaño de la región.

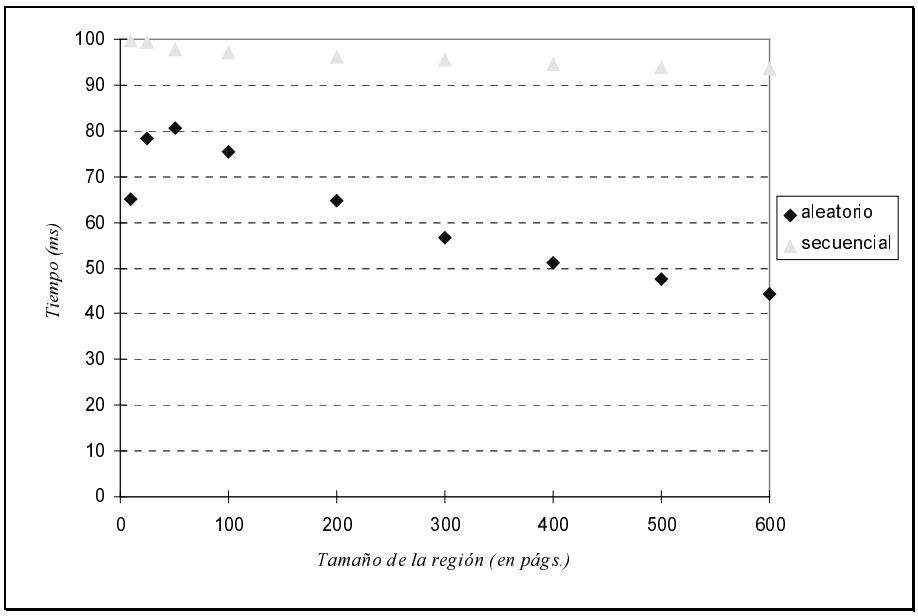

Ilustración 6-1: Variación del coste de los mensajes incrementales en función del tamaño de la región de memoria virtual con 10 páginas modificadas.

Para explicar este comportamiento es necesario tener en cuenta algunas decisiones de diseño del mecanismo de copia en escritura que se ha implementado sobre VSTa. El mecanismo se ha implementado para un tipo especial de región de memoria virtual, que se llena con ceros bajo demanda. Así, si una página no ha sido llenada todavía (porque no ha sido leída ni escrita), en el caso de que sea incluida en una operación incremental, no se copia en escritura, sino que se marca como vacía en el destino. El resultado es correcto, ya que, al ser accedidas, ambas copias de la página serán llenadas con el mismo valor (ceros). Por otra parte, cuando se recibe un mensaje incremental, se eliminan las referencias (a marcos de página o a bloques en el área de intercambio) que puedan existir en las estructuras de datos asociadas a páginas que deben ser sustituidas (por las que forman parte del mensaje).

Como resultado, si la página de destino está vacía[1], únicamente es necesario hacer que referencie a la página original mediante copia en escritura. En cambio, si la página de destino ya había sido incluida en un mensaje anterior, primero es necesario eliminar la referencia sobre el contenido de la página, antes de cambiarlo por el del mensaje. La probabilidad de que una página sea modificada —y, por lo tanto, incluida en un mensaje incremental— es inversamente proporcional al tamaño de la región. Por esto, cuanto menor es la región más probable es que una misma página sea incluida en varios mensajes sucesivos, y de ahí la disminución del coste a medida que aumenta el

[1] Nótese que, inicialmente, el proceso de respaldo prepara una región de memoria virtual cuyas páginas serán llenadas con cero bajo demanda. Puesto que el proceso de respaldo no lee ni modifica directamente esta región, la única forma de que una página de la región esté llena es que haya sido incluida en un mensaje incremental anterior.

tamaño. Evidentemente, esto no ocurre en el caso secuencial, en el que siempre se seleccionan las mismas páginas. Pero, precisamente por ello, el coste es mayor que en el aleatorio, ya que, salvo en el primer mensaje, todas las páginas de destino están llenas.

Finalmente, cuando la región es suficientemente pequeña —por debajo de 50 páginas, en este caso—, la probabilidad de que una página sea seleccionada más de una vez entre dos mensajes sucesivos es lo suficientemente grande como para conseguir que el coste disminuya, al transferirse un menor número de páginas. En el estudio realizado, el valor medio del número medio de páginas incluidas en cada mensaje incremental para una región de 50 páginas es de 9,04. Para tamaños inferiores a 50 se produce una disminución significativa, hasta llegar a 6,32 páginas de media para regiones de 10 páginas. En cambio, para regiones mayores de 50 páginas el valor aumenta sólo ligeramente (hasta llegar a 9,96 páginas en regiones de 600 páginas), por lo que deja de influir significativamente en los resultados. (Ver anexo V).

El anexo V recoge un estudio similar para los mensajes incrementales negativos que corrobora los resultados obtenidos. Además, el estudio demuestra que permitir el intercambio de un mensaje incremental negativo mediante una única interacción entre cliente y servidor resulta efectivo para reducir el coste de la operación y hacerlo comparable a la del mensaje incremental.

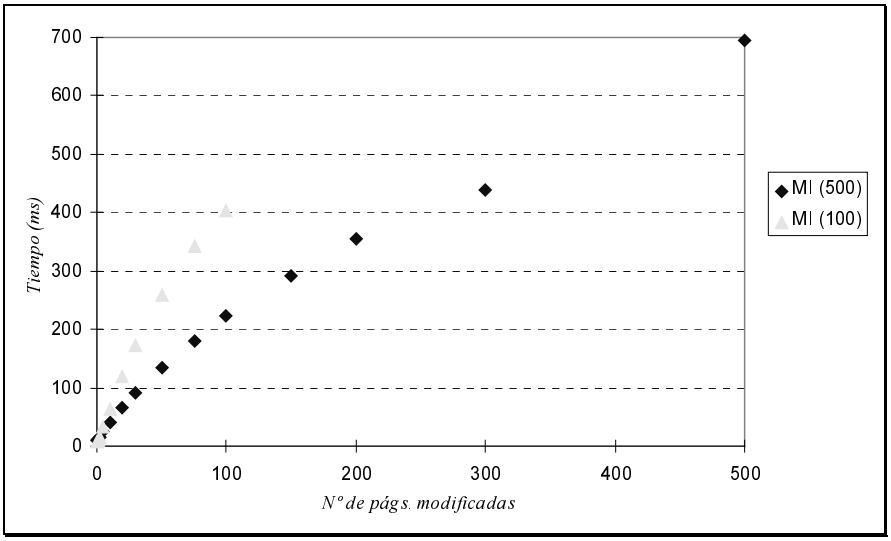

Ilustración 6-2: Variación del coste de los mensajes incrementales en función del número de páginas incluidas en el mismo, para regiones de memoria virtual de 100 y de 500 páginas respectivamente.

La Ilustración 6-2 recoge el coste de los mensajes incrementales en función del número de páginas modificadas para el caso de modificación aleatoria. Como puede observarse, el coste de la operación con el mismo número de páginas modificadas es menor para la región de 500 páginas que para la de 100, por los mismos motivos

expuestos anteriormente. La ilustración muestra que el coste de un mensaje incremental es prácticamente proporcional al número de páginas incluidas en él, como cabría esperar.

Medidas con robo de páginas

En la Ilustración 6-3 se recoge la dependencia del coste de los mensajes incrementales con respecto al tamaño de la región de memoria virtual para diferentes cantidades de páginas modificadas. El aspecto más destacado, como ocurría en el caso de la memoria virtual, es el decremento del tiempo a medida que aumenta el tamaño de la región. A pesar de la similitud, no es posible atribuir el efecto a la misma causa, ya que aquí el efecto es superior en dos órdenes de magnitud.

El motivo de esta disminución tan importante del coste, tanto más acusada cuanto mayor es el número de páginas modificadas, es la sincronización a nivel de página. En efecto, cuanto menor es el tamaño de la región de memoria, mayor es la frecuencia de los bloqueos por sincronización. Además, este problema es especialmente grave en el sistema estudiado, donde la rutina de robo de páginas únicamente selecciona para ser reemplazadas páginas de regiones de memoria virtual del tipo que permite las operaciones incrementales.[2] Finalmente, a partir de 3000 páginas de tamaño, se produce un incremento en el coste temporal, atribuible a que, al haber un mayor número de páginas en la región, se produce un mayor número de fallos de página.

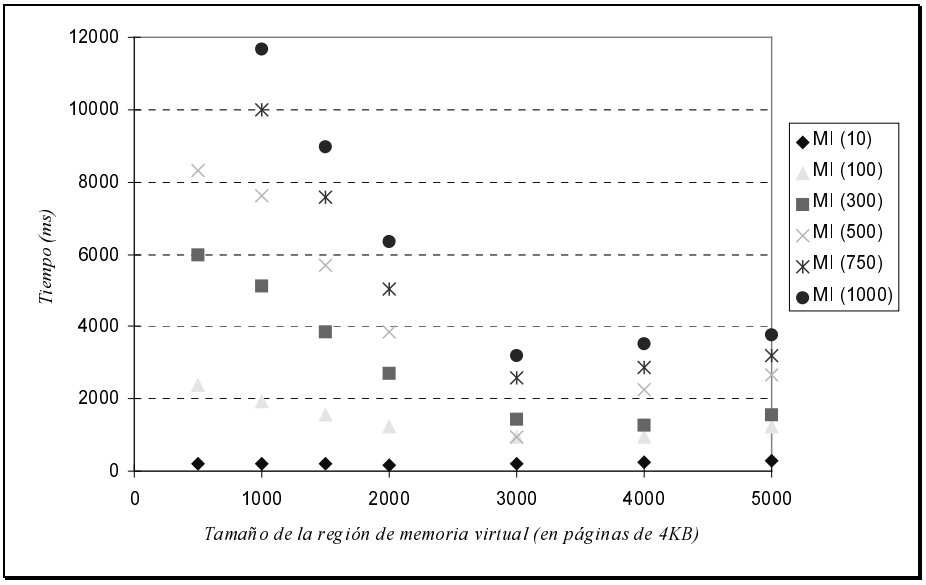

Ilustración 6-3: Coste temporal de los mensajes incrementales en función del tamaño de la región de memoria virtual, para 10, 100, 300, 500, 750 y 1000 páginas modificadas.

[2] La razón es que la rutina original de robo de páginas en la distribución de VSTa que se ha utilizado para la implementación (versión 1.5.2) contiene algunos fallos que provocan el pánico del núcleo.

Este estudio demuestra la importancia que tiene el diseño del sistema de memoria virtual y su interacción con la implementación de las operaciones incrementales. Nótese que en la implementación actual la operación incremental debe esperar a que se libere el acceso a la página tras finalizar la operación de lectura o escritura en disco. No obstante, en principio, sería posible modificar el diseño del mecanismo de sincronización del núcleo para el acceso a las páginas de forma que ambas actividades se realizasen de forma concurrente.

En el anexo V se recoge un estudio de la variación del coste de los mensajes incrementales en las sucesivas medidas tomadas para después calcular los valores medios que se muestran en este apartado. A pesar de que los mensajes sucesivos son enviados tras modificar el mismo número de páginas en la misma región virtual de origen, se produce un efecto transitorio[3] al arrancar el proceso que corrobora la importancia de los problemas de sincronización en el coste de estas operaciones.

6.1.2 Estudio del establecimiento de puntos de recuperación

En este apartado se estudia la sobrecarga introducida por el establecimiento de puntos de recuperación en una aplicación de cálculo intensivo. Como ejemplo de aplicación se ha utilizado un programa de multiplicación de matrices, que comienza su ejecución con dos matrices de 615x615 enteros almacenadas en disco. El programa las lee, las multiplica y, finalmente, escribe la matriz resultado en disco. Junto con este programa, se ha preparado una librería que permite establecer puntos de recuperación de seis formas diferentes:

1. Incremental en memoria. Para establecer el punto de recuperación, la librería envía un mensaje incremental a un proceso de respaldo como el descrito para el servicio transaccional. (Ver capítulo 5). Este proceso simplemente recibe el mensaje, que actualiza su copia de los datos, y responde inmediatamente.
2. Incremental en un fichero (síncrono). La librería envía un mensaje incremental a un proceso de respaldo. Pero, a diferencia del caso anterior, el proceso de respaldo recibe el mensaje empaquetado, interpreta el mapa y actualiza, mediante el servicio estándar *write*, la copia de los datos que mantiene en un fichero en disco.[4] A continuación responde al cliente (el proceso que realiza la multiplicación).
3. Incremental en un fichero (asíncrono). Es equivalente al caso anterior, salvo que el proceso de respaldo responde al cliente inmediatamente, antes de actualizar la copia.
4. Empaquetado en un fichero (síncrono). Es similar al almacenamiento incremental en un fichero, salvo que el proceso de respaldo no procesa el

[3] Para evitar este efecto transitorio se han eliminado las 15 primeras medidas de cada serie antes de calcular los valores medios que se recogen en este estudio con el robo de páginas activo.

[4] Esta configuración es equivalente a la descrita en el apartado 5.2. En este caso, el proceso de respaldo envía las actualizaciones al servidor de ficheros, quien a su vez las envía al servidor de disco.

mensaje incremental empaquetado, sino que lo guarda en un fichero nuevo cada vez.

5. Empaquetado en un fichero (asíncrono). Es equivalente al anterior, salvo en la respuesta del proceso de respaldo, que se produce antes de escribir los datos en disco.
6. No incremental en un fichero. En este caso, para establecer el primer punto de recuperación, la librería escribe las tres matrices. Para establecer el resto de puntos de recuperación escribe únicamente la matriz resultado, aprovechando que las matrices a multiplicar no son modificadas.

Todas las medidas que se muestran en la Tabla 6-1 han sido obtenidas en la misma plataforma descrita para el estudio de las operaciones incrementales. Durante las pruebas, los procesos de respaldo y los ficheros se han almacenado en el mismo nodo que el proceso que realiza la multiplicación, por lo que únicamente es posible tolerar fallos que no afecten a la imagen del sistema de ficheros en disco y, en caso de fallo permanente del nodo, el disco debería incorporarse a otro nodo para poder continuar con la ejecución de la aplicación. En todos los casos se han establecido 11 puntos de recuperación durante la ejecución de la aplicación, es decir, aproximadamente un punto de recuperación cada 50 s.

	Método	tiempo ejecución (s)	sobrecarga por PdR (s)	sobrecarga por PdR (%)	tiempo medio PdR (s)	tamaño medio PdR (MB)
0	Sin PdR	547,9				
1	Incr. en memoria	548,6	0,7	0,1	0,03	0,39
2	Incr. en fich. (síncr.)	566,9	19,1	3,5	1,71	0,39
3	Incr. en fich. (asíncr.)	566,6	18,8	3,4	0,05	0,39
4	Empaq. en fich. (síncr.)	557,3	9,4	1,7	0,73	0,39
5	Empaq. en fich. (asíncr.)	556,3	8,5	1,5	0,02	0,39
6	No incr. en fichero	572,3	24,4	4,4	2,04	1,71

Tabla 6-1: Coste de establecer puntos de recuperación utilizando operaciones incrementales.

Los resultados obtenidos para los puntos de recuperación con operaciones incrementales muestran que éstas permiten mejorar significativamente las prestaciones cuando el proceso modifica únicamente una parte de su espacio de direcciones entre dos puntos de recuperación, como ocurre en el caso considerado. Así, la sobrecarga debida a los puntos de recuperación se ve reducida de 24,4 s. cuando no se utilizan operaciones incrementales a 9,4 s. cuando se escriben únicamente las páginas modificadas. Aunque los valores absolutos serían diferentes si la copia de los datos se mantuviese en otro nodo diferente, cabe esperar una reducción similar en la sobrecarga, cuyo valor absoluto dependerá, principalmente, de la capacidad de transferencia del medio de conexión.

Como puede observarse, si se envía el mensaje incremental empaquetado (MIE) directamente a un nuevo fichero cada vez (casos 4 y 5), se obtiene una reducción del

50 % aproximadamente con respecto a los casos 2 y 3, en los que se mantiene la copia de respaldo en un fichero. Esta reducción se debe principalmente a las continuas interacciones entre el proceso de respaldo y el servidor de ficheros en los casos 2 y 3, ya que en estos casos este proceso interpreta el mapa del MIE y escribe las páginas en el fichero de una en una (aunque en ambos casos el tamaño de los puntos de recuperación es el mismo); y en cambio, en los casos 4 y 5 el proceso de respaldo realiza una única escritura en un fichero nuevo. No obstante, es necesario tener en cuenta que la estrategia utilizada en los casos 4 y 5 tiene un coste de recuperación tras el fallo más elevado, al ser necesario procesar todos los MIEs, y utiliza más espacio en disco, ya que puede haber varias copias de la misma página en los diferentes MIEs.

En los cuatro casos anteriores (2 a 5), se observa que apenas existe diferencia entre variantes síncrona y asíncrona de cada método. Este hecho se debe a que la prueba se ha realizado con todos los procesos en el mismo nodo. En cambio, si el proceso de respaldo residiese en un nodo diferente, cabría esperar diferencias significativas entre escritura síncrona y asíncrona. La razón es que en las variantes asíncronas el almacenamiento en disco no influiría sobre la aplicación por realizarse en otro nodo.

Por último, cabe destacar la poca sobrecarga debida a las operaciones incrementales, tal y como se observa en el caso 1, en el que el proceso de respaldo simplemente recibe el mensaje incremental y responde inmediatamente, sin realizar ninguna acción posterior. Igualmente, en las variantes asíncronas se puede observar que el tiempo medio durante el que el proceso principal está bloqueado, que se recoge en la columna etiquedada como “tiempo medio PdR”, es muy inferior al de la correspondiente variante síncrona.

6.1.3 Análisis de resultados

La implementación de los mensajes incrementales sobre VSTa ha permitido comprobar dos propiedades básicas de estas operaciones, que pueden deducirse de su definición:

- El coste de un mensaje incremental —y de un mensaje incremental negativo— es directamente proporcional al número de páginas que se incluye en el mensaje.
- Sin embargo, el coste no depende significativamente del tamaño de la región de memoria virtual, porque otros factores tienden a dominar sobre el coste de recorrer dicha región para localizar las páginas modificadas.

Además, se ha comprobado la efectividad de evitar la copia en escritura siempre que la semántica de la región de memoria virtual lo permita; es decir, de transferir únicamente las páginas llenas. En la implementación realizada, únicamente se han considerado regiones de llenado con ceros bajo demanda; pero sería necesario incluir en el diseño otros tipos de regiones, como las que se llenan bajo demanda desde un fichero.

Por otra parte, se ha puesto de manifiesto la importancia capital de un diseño cuidadoso de los mecanismos de sincronización para el acceso a las estructuras de datos del sistema de memoria virtual. En concreto, deben reducirse al mínimo las esperas de las rutinas que recorren una región de memoria virtual para identificar las

páginas modificadas, tanto para realizar operaciones incrementales como para consultar o establecer el estado de las páginas. Para ello es importante tener en cuenta que tanto la lectura de una página desde el disco como el volcado al mismo de una página modificada pueden realizarse al mismo tiempo que la consulta y modificación de las copias reservadas para las operaciones incrementales de los bits de modificado y accedido. Por lo tanto, es recomendable diseñar el mecanismo de sincronización de forma que este nivel de concurrencia potencial no se vea reducido por un exceso de sencillez —un único tipo de cerrojo por página, por ejemplo—. Nótese que las mejoras en este sentido pueden permitir un aumento considerable de las prestaciones de las operaciones incrementales, ya que los retrasos por sincronización son la causa principal de su coste temporal.

Es conveniente destacar que los retrasos por sincronización aumentan enormemente cuando una región consume una gran parte de los marcos de página del sistema. Por lo tanto, a menos que se reduzcan los retrasos debidos a la sincronización por otros medios (como se explica en el párrafo anterior), resulta recomendable dividir las estructuras de datos en varias regiones diferentes, para así reducir la probabilidad de interferencias entre las operaciones incrementales y el robo de páginas.

Finalmente, el estudio del establecimiento de puntos de recuperación utilizando mensajes incrementales ha mostrado, en un caso práctico, la viabilidad de utilizar estos servicios para establecer puntos de recuperación dirigidos por el usuario. Además, la gestión del registro de páginas modificadas no supone una carga adicional sobre la ejecución de los procesos cuando no se utilizan las operaciones incrementales, ya que este registro es mantenido directamente por el soporte físico, tal y como se describe en el anexo V.

6.2 Servicio transaccional de ficheros

En este apartado se describe el diseño y la implementación de una aplicación tolerante a fallos: un servicio transaccional de ficheros. La finalidad de esta aplicación es la de demostrar la factibilidad de construir servicios transaccionales de forma sencilla y prácticamente independiente de la funcionalidad apoyándose en los servicios especificados para el micro-núcleo y utilizando la estructura descrita en el capítulo anterior.

A la hora de diseñar el servicio se ha supuesto que el módulo cliente permite que el usuario final del sistema informático especifique cuándo comienza una transacción y si esta debe terminar o abortar. Para ello, la interfaz de usuario (en este caso el intérprete de órdenes) incorpora las operaciones *ComenzarT*, *TerminarT* y *AbortarT*. Únicamente cuando el usuario lo solicita, el cliente invoca la operación correspondiente en el servicio transaccional, por lo que el usuario tiene un control total sobre las transacciones.

Así, el usuario interacciona con el sistema de ficheros mediante utilidades convencionales como editores, compiladores o programas para copiar, crear o mover ficheros y directorios. Además, en este caso, el usuario dispone de la posibilidad de confirmar o de cancelar las actividades realizadas hasta un cierto instante. Por ejemplo, supóngase que un usuario intenta compilar un programa que acaba de preparar en un fichero

denominado “util.c”. No es un usuario experto y confunde el orden de los argumentos para el compilador, introduciendo la orden “cc -o util.c util”. Para su desespero, descubre que acaba de perder irremisiblemente varias horas de trabajo. Sin embargo, en este caso, podría recuperarlo simplemente abortando la transacción en marcha, mediante una orden del tipo: “abortarT idt”, donde ‘abortarT’ es un programa que invoca la operación *AbortarT* del servicio de ficheros e ‘idt’ es el identificador de la transacción.

Incluso para usuarios más avanzados esta facilidad transaccional ofrece múltiples ventajas. Por ejemplo, supóngase que un administrador de sistemas necesita realizar varias modificaciones en diferentes ficheros de configuración para poner en marcha una nueva utilidad del sistema. Con un servicio de ficheros transaccional podría realizar las modificaciones como parte de una misma transacción, comprobar si todo funciona correctamente y, sólo entonces, confirmar la transacción. En cambio, si descubre que ha cometido un error, simplemente tiene que abortar la transacción en curso para restaurar el estado original de todos los ficheros de configuración.

Además, el usuario se beneficia de la garantía de funcionamiento que permite ofrecer el servicio transaccional, que puede garantizar la persistencia de las transacciones completadas incluso ante la presencia de fallos. En caso de fallo, el usuario tiene la garantía de que el estado del sistema de ficheros sigue siendo consistente, y exactamente igual al que tenía en el momento en que comenzó la transacción.

Finalmente, y no por ello menos importante, la opción elegida ha permitido emplear las utilidades estándar para comprobar el correcto funcionamiento del servicio transaccional sin necesidad de modificar (ni siquiera recompilar) dichas utilidades.

Tal y como se describe en este apartado, este enfoque sobre la utilización del servicio transaccional ha influido notablemente sobre algunas decisiones importantes, como la técnica para el control de concurrencia. No obstante, en el subapartado 6.2.4 se discute qué otras posibilidades se han considerado y cómo hubieran influido sobre el diseño, mostrando que la estructura propuesta hubiera resultado igualmente adecuada para implementarlas.

6.2.1 El sistema de ficheros

La estructura utilizada para el servicio transaccional es la propuesta en el apartado (Cap. 4, servicio transaccional concurrente), que permite la ejecución de transacciones concurrentes y tolerar fallos que afecten a un único nodo del sistema. Para poder utilizar esta estructura, el servicio de ficheros debe cumplir dos condiciones básicas:

1. El servicio debe utilizar una región de memoria virtual para almacenar los datos.
2. Los datos alojados en cada página deben tener una relación lógica entre sí (por ejemplo, formar parte del mismo fichero), ya que la unidad mínima para el control de concurrencia es la página.

Desafortunadamente, los servidores de ficheros disponibles en VSTa —y que podrían haber sido transformados en transaccionales con poco esfuerzo— no satisfacen las condiciones anteriores. Por ejemplo, el sistema de ficheros temporal,

tmpfs, utiliza el espacio de direcciones del proceso servidor como espacio de almacenamiento; pero a) la información sobre el fichero puede ocupar varias regiones de memoria virtual debido a la librería de gestión de memoria dinámica, y b) la información contenida en una página puede formar parte de varios ficheros diferentes. Por otra parte, la organización de la información en los discos de otros sistemas de ficheros presentan inconvenientes parecidos. Por ejemplo, los nodos-I suelen agruparse en una zona del disco y cada uno de ellos ocupa unos pocos bytes (64 en el sistema de ficheros de Unix System V), con lo que habría un número importante de ellos en la misma página (64 con páginas de 4K). Esto significaría que los 64 ficheros correspondientes tendrían que compartir el mismo cerrojo, con la consiguiente disminución de la concurrencia.

Por lo tanto, basándose en los diseños existentes de sistemas de ficheros (especialmente los de VSTa, para así aprovechar al máximo los módulos de soporte lógico disponibles) se ha desarrollado un sistema de ficheros adecuado para la estructura del servicio transaccional, al que se ha denominado *cfs*. El diseño de *cfs* tiene dos aspectos básicos:

- Para simplificar al máximo las operaciones transaccionales, se ha diseñado la estructura del sistema de ficheros de forma que tanto los datos como los metadatos puedan ser actualizados con un único mensaje incremental.
- Para maximizar la concurrencia, cada nodo-I reside en una página distinta, junto con algunos de los datos del fichero —todos, si es suficientemente pequeño—.

La estructura de la página inicial del fichero, que es la que contiene el nodo-I, se muestra en la Ilustración 6-4. En ella se puede observar, a la izquierda, la estructura del sistema de ficheros en la región de memoria —que sería el equivalente a una partición de disco—. El aspecto más destacable es que la estructura de datos para la gestión de la lista de páginas libres, un mapa de recursos, tiene un tamaño fijo. Para minimizar el número páginas que resultan modificadas como resultado de la gestión de la lista, se ha preferido evitar la técnica habitual de enlazar las páginas (bloques, en el caso de un sistema de ficheros en disco) utilizando las propias páginas libres. En efecto, si cada página libre apuntase a la siguiente, cualquier modificación en el puntero que contendría una página causaría que la operación incremental la copiase entera, con el consiguiente coste adicional.

A la derecha de la ilustración se muestra el contenido de una página con un nodo-I. Como se ha dicho, cada nodo-I reside en una página diferente para maximizar la concurrencia. No obstante, para aprovechar el espacio en la región de memoria, el resto de la página contiene los primeros 3992 bytes del fichero, tal y como se recoge en la ilustración. Esta organización tiene la ventaja de que, para ficheros pequeños, es posible modificar los atributos y los datos afectando sólo a una página, que es la unidad de transferencia de las operaciones incrementales.

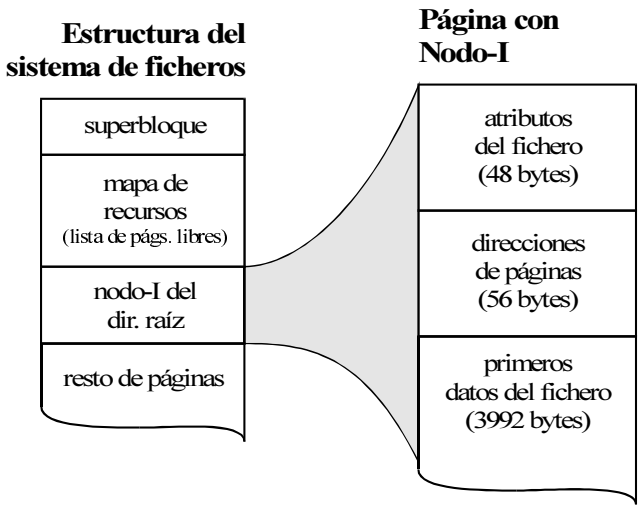

Ilustración 6-4: Estructura del sistema de ficheros cfs.

6.2.2 Estructura del servicio

En la Ilustración 6-5 se muestra la estructura del servicio transaccional de ficheros, que se corresponde con la estructura descrita en el capítulo anterior para el servicio transaccional concurrente tolerante a fallos.

En la ilustración se ha utilizado efecto de sombra para los datos almacenados de forma estable. En efecto, el sistema de ficheros es resistente a fallos que afecten a un nodo, ya que en todo momento se mantienen dos copias del estado del sistema: una en el proceso ST (D) y otra en el proceso R (D_R). Tal y como se resaltó en el apartado anterior, para poder tolerar los fallos es necesario que ambos procesos residan en nodos diferentes.

No obstante, aunque los tres tipos de proceso mantienen copia del sistema de ficheros,[5] tal y como se describe a continuación cada uno de ellos lleva a cabo funciones diferentes, por lo que no todos ellos interpretan estos datos. Para resaltar qué parte del sistema de ficheros es utilizada por cada uno de estos procesos se ha representado ésta sobre fondo blanco. En cambio, las partes que no son interpretadas por un proceso aparecen sobre fondo gris. Otra diferencia entre los distintos tipos de procesos es la estructura de datos adicional que utiliza cada uno de ellos, y que se incluye en la parte inferior de la figura. A continuación se describen las funciones y estructuras de datos de cada uno de los tipos de procesos.

[5] Cada copia se mantiene en una región de memoria virtual en el espacio de direcciones de cada uno de estos procesos.

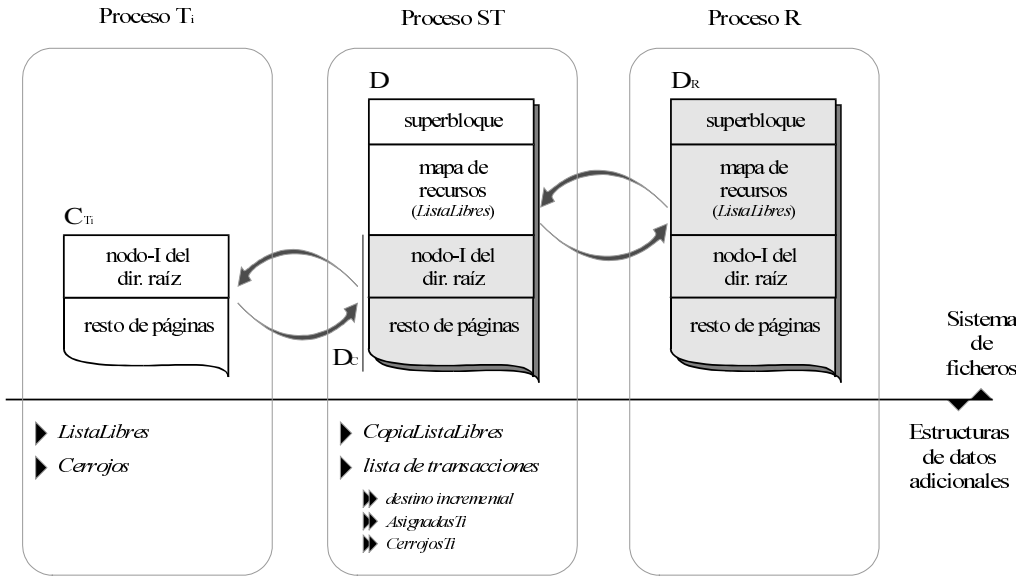

Ilustración 6-5: Estructura del servicio transaccional de ficheros.

El proceso R

El proceso R simplemente ofrece servicios para el almacenamiento y la recuperación incrementales de la región de memoria virtual, por lo que no incluye funciones para interpretar los datos de la región. Esto hace que su código sea totalmente general, aplicable a cualquier servicio transaccional. El relleno gris en la Ilustración 6-5 sobre la región de memoria en el espacio de direcciones de R destaca este hecho, indicando que no consulta ni modifica esta información, sino que se limita a permitir actualizaciones o recuperaciones de la información mediante operaciones incrementales.

El proceso ST

El proceso ST es, fundamentalmente, el coordinador de las transacciones en curso. Esta característica básica se refleja en su interfaz, que se recoge en la Ilustración 6-6. Con respecto a la interfaz básica, descrita en el capítulo anterior, ST ofrece tres nuevos servicios —*Abrir()*, *ObtenerI()* y *Obtener()*— que, junto con los servicios transaccionales básicos permiten la coordinación de transacciones concurrentes.

Como puede observarse en la Ilustración 6-5, ST utiliza el superbloque y el mapa de recursos del sistema de ficheros; pero, al igual que R, tampoco interpreta el resto de los datos de la región. En cambio, mantiene una estructura de datos adicional, formada básicamente por dos listas: la de páginas libres (*CopiaListaLibres*) y la de transacciones en marcha.

La necesidad de *CopiaListaLibres*, que replica el mapa de recursos del sistema de ficheros (*ListaLibres*), surge de la siguiente decisión de diseño: D contiene únicamente las modificaciones debidas a transacciones completas (y no a transacciones en curso).

Por eso resulta necesaria la lista auxiliar; ya que a medida que una transacción avanza puede solicitar nuevas páginas que ST debe reservarle —mediante la operación *Obtener()*— y, sin embargo, las modificaciones a la lista de páginas libres deben deshacerse en caso de un aborto de la transacción. De esta forma, en caso de aborto no es necesaria la comunicación entre ST y R, al no haberse modificado la región D. Nótese, además, que D y D_R deben contener en todo momento la misma información, pues de lo contrario el almacenamiento del sistema de ficheros no sería estable.

Por cada transacción en marcha ST mantiene dos estructuras de datos: el conjunto de cerrojos obtenidos por la transacción y el conjunto de páginas que le han sido asignadas temporalmente —como resultado de las operaciones *ObtenerI()* y *Obtener()*—. Como puede verse en la Ilustración 6-6, estas estructuras son redundantes con las que mantiene cada T_i; pero, tal y como se verá más adelante, esta información permite a ST tolerar fallos de los procesos T_i.

Abrir(T_i) → superbloque
Cada proceso T_i la utiliza para registrarse en el proceso ST. La función devuelve a T_i una copia del superbloque.
ObtenerI(nodoI, tipo) → resultado
Solicitud de cerrojo del tipo indicado para el fichero cuyo nodo de índices es nodoI. *ST devuelve el resultado de la operación, que no es bloqueante en ningún caso.*
Obtener(n) → comienzo
Solicitud de n *bloques (páginas) libres. Si es posible asignarlas, ST devuelve la dirección de comienzo (relativa al comienzo del sistema de ficheros) de la primera página asignada.*
ComenzarT(), TerminarT(), AbortarT()
Funciones del servicio transaccional. (La información proporcionada por T_i al invocar estos servicios y los resultados devueltos por ST se describen con detalle junto con los algoritmos, en este mismo apartado).

Ilustración 6-6: Interfaz ofrecida por el proceso ST a los procesos T_i.

Los procesos T_i

Los procesos T_i son los encargados de ofrecer el servicio de ficheros a los clientes. Para ello, contienen las funciones habituales en un servidor de ficheros que, en este caso, interpretan la estructura de *cfs*. Tal y como se recoge en la Ilustración 6-5, los procesos T_i únicamente trabajan con las páginas del sistema de ficheros y no con el superbloque o el mapa de recursos. La razón es que no pueden decidir por sí mismos qué bloques libres utilizar, sino que deben consultarlo con ST, que realiza una gestión centralizada de esta estructura.

Del mismo modo, las páginas liberadas por una transacción en curso (por ejemplo, porque se haya eliminado un fichero) tampoco pueden ser utilizadas por otra transacción, ya que, en caso de aborto, esas páginas deben volver a formar parte de los ficheros originales. Por eso, cada T_i mantiene una lista de páginas libres local

(*ListaLibres*), que contiene, además de las páginas solicitadas a ST[6] y no utilizadas todavía, las que se hayan liberado como resultado de las operaciones de la transacción en curso.

Control de concurrencia

Uno de los aspectos más delicados del servidor transaccional de ficheros es el control de concurrencia que lleva a cabo el proceso ST con la colaboración de los procesos T_i. En esta implementación se ha elegido una estrategia pesimista basada en cerrojos de lectura y de lectura/escritura con granularidad de fichero.

La razón básica es que el presente servicio, tal y como se ha descrito, ha sido diseñado para que los servicios transaccionales sean utilizados directamente por el usuario del sistema informático. En esta situación, resultaría inaceptable exigirle al usuario que repitiese el último cuarto de hora de trabajo (por ejemplo, de edición de un fichero) porque la transacción que estaba llevando a cabo ha entrado en conflicto con alguna de otro usuario. Las estrategias pesimistas son las únicas que permiten evitar el aborto por conflictos entre transacciones y por ello han sido elegidas en este caso aunque las estrategias optimistas permitan mayor grado de concurrencia y tiendan a ser más eficientes cuando el número de conflictos es reducido.

Uno de los problemas más graves de los cerrojos como medio de control de concurrencia es la posibilidad de interbloqueo entre transacciones y la solución escogida ha sido prevenirlos. Para ello, se ha eliminado una de las condiciones necesarias para que se produzcan, al no permitir que las transacciones esperen para obtener un cerrojo. En vez de esperar, cuando una transacción intenta obtener un cerrojo y no puede, ST devuelve un error a T_i quien, a su vez, lo devuelve al usuario. Esta solución resulta adecuada, ya que el usuario es quien posee la información necesaria para tomar la decisión sobre cómo tratar el conflicto.[7]

Modificaciones al servicio de ficheros

Los cambios realizados al código del servicio básico de ficheros *cfs* para transformarlo en un servicio transaccional han sido mínimos. La razón fundamental es la modularidad impuesta por la propia estructura del servicio. Tal y como se ha descrito, el único tipo de proceso que gestiona las páginas de datos, de atributos y de índices del sistema de ficheros es el T_i. Así pues, este proceso incluye, con pequeñas modificaciones, el código completo del servicio de ficheros original.

No obstante, existen dos diferencias importantes; aunque sencillas de llevar a la práctica. La primera se debe al hecho de que T_i no accede directamente al mapa de recursos del sistema de ficheros. En cambio, para obtener páginas libres debe solicitarlas a ST. Para disminuir el número de interacciones con ST, T_i solicita más de

[6] Para disminuir el número de mensajes entre los T_i y ST, existe un número mínimo de páginas libres que se solicitan a ST cada vez que se agotan las de la lista local.

[7] Nótese que una transacción típica podría incluir la edición de dos o más ficheros (por ejemplo, con un procesador de textos) y entonces el conflicto querrá decir que hay otro usuario trabajando con alguno de esos ficheros.

las que necesita y deposita el resto en la lista local de páginas libres. La segunda diferencia se refiere a la apertura de ficheros y directorios. Antes de permitir que un cliente abra un fichero (o directorio, fichero de ahora en adelante), T_i comprueba, en *Cerrojos* (lista local de cerrojos obtenidos), si se dispone del cerrojo adecuado sobre el fichero. Si no está en esta lista, o no es del tipo apropiado, lo solicita a ST y sólo entonces se permite la apertura del fichero. Un aspecto importante de la obtención de cerrojos es que el fichero puede haber sido modificado previamente por una transacción que ya haya finalizado, por lo que la obtención de un cerrojo requiere la actualización de C_{Ti}.

Además de las modificaciones reseñadas, ha sido necesario añadir a T_i un módulo para la gestión de transacciones, que se describe en el apartado siguiente. Igualmente, el código de ST ha tenido que ser desarrollado desde el principio. A pesar de ello, la transformación del servicio básico de ficheros en el servicio transaccional requirió únicamente en torno a una semana de trabajo. Teniendo en cuenta que el autor no tenía experiencia previa en el desarrollo de servicios transaccionales, este reducido esfuerzo resalta la versatilidad y facilidad de uso de las operaciones incrementales.

6.2.3 Algoritmos

A continuación se describen los algoritmos utilizados para implementar los servicios transaccionales. Para esta descripción se han utilizado los siguientes campos de los mensajes:

op	operación a realizar por el servidor
destincr	destino incremental, para que sea interpretado por el micro-núcleo
datos	región (o parte de una región) de memoria virtual y que puede ser el origen o el destino de los datos en función de la operación que se realice
mapa	lista de páginas que deben ser incluidas en el mensaje incremental

Antes de la invocación a *EnviarMensaje()* o a *RecibirMensaje()* únicamente se indica el contenido de aquellos campos que van a ser utilizados, para simplificar la descripción y, en el caso de los mensajes que devuelven el resultado, éste se incluye directamente como parte de la invocación. Además, se han destacado en negrita los mensajes incrementales, que producen la actualización de las diferentes copias de los datos C_{Ti}, D y D_R.

Comienzo de transacción

Para comenzar una transacción a instancias del cliente, un proceso T_i envía un mensaje a ST con *ComenzarT* en el campo *op*, tal y como se muestra en la Ilustración 6-7, y espera la contestación de ST.[8] Por su parte, ST, tal y como se muestra en la Ilustración

[8] Nótese que, por generalidad, se ha considerado que la función *EnviarMensaje()* es asíncrona. Pero, dado que la relación entre los procesos es de cliente y servidor, si el núcleo del sistema operativo ofrece

6-8, al recibir un mensaje de un T_i, concede un cerrojo de sólo lectura sobre el directorio raíz a T_i[9] y anota el principio de la transacción. A continuación, responde a T_i con un mensaje incremental en el que incluye las páginas modificadas desde el mensaje incremental anterior. Para ello, cada T_i tiene asociado un destino incremental diferente en ST. Finalmente, T_i recibe el mensaje sobre C_{Ti} (que resulta actualizada cuando se proyectan las páginas incluidas en el mensaje), añade el cerrojo de lectura al conjunto de cerrojos obtenidos y confirma la operación al cliente.

```
Ti, al recibir mensaje con operación ComenzarT:
{
      no debe haber transacciones en marcha

      m.op ← ComenzarT
      EnviarMensaje(ST, m)
      m.datos ← CTi
      RecibirMensaje(ST, m, incremental)

      Cerrojos ← {[directorio raíz, LECTURA]}
      marcar transacción en marcha

      EnviarMensaje(Cliente, OK)
}
```

Ilustración 6-7: Algoritmo para comenzar una transacción en los procesos T_i.

```
ST, al recibir mensaje con operación ComenzarT:
{
      Ti no debe tener en marcha ninguna transacción
      CerrojosTi ← {[directorio raíz, LECTURA]}
      marcar Ti con transacción en marcha

      m.destincr ← destino incr. asociado a Ti
      m.datos ← DC
      EnviarMensaje(Ti, m, incremental)
}
```

Ilustración 6-8: Algoritmo para comenzar una transacción en el proceso ST.

Terminar transacción

El algoritmo para terminar una transacción requiere mayor intercambio de información entre T_i y ST. En efecto, tal y como recoge la Ilustración 6-9, T_i comienza enviando un mensaje con la operación *TerminarT* y en él incluye la lista de páginas liberadas

esta posibilidad, resulta más adecuado utilizar un servicio *enviar* que se bloquee hasta la respuesta del servidor (como se ha hecho realmente en la implementación sobre VSTa).

[9] Este cerrojo de sólo lectura es el mínimo necesario para que una transacción pueda realizar operaciones sobre el sistema de ficheros, ya que atravesar un directorio o abrir cualquiera de los objetos contenidos en él requiere este tipo de cerrojo.

durante la transacción (*ListaLibres*). Seguidamente envía un mensaje incremental con las páginas de C_{Ti} que han sido modificadas. A diferencia de ST, que debe considerar varios destinos incrementales, los T_i solamente consideran uno, el correspondiente a ST.

Cuando ST recibe la petición de terminar una transacción copia la lista de páginas a liberar en la estructura de datos asociada a la transacción (*LiberadosTi*) y espera a recibir el siguiente mensaje de T_i, que actualizará $D_{págs}$, es decir, la zona del sistema de ficheros que contiene las páginas correspondientes a ficheros y directorios (incluyendo tanto las páginas de datos como las páginas de índices). (Ver Ilustración 6-10).

```
Ti, al recibir mensaje con operación TerminarT:
{
      debe haber una transacción en marcha

      m.op ← TerminarT
      m.datos ← ListaLibres
      EnviarMensaje(ST, m)

      m.op ← TerminarT
      m.desticr ← destino incr. asociado a ST
      m.datos ← CTi
      EnviarMensaje(ST, m, incremental)

      RecibirMensaje(ST, OK)

      ListaLibres ← ∅
      Cerrojos ← ∅
      marcar que no hay transacción en marcha
      EnviarMensaje(Cliente, OK)
}
```

Ilustración 6-9: Algoritmo para terminar una transacción en los procesos T_i.

Una vez actualizada $D_{págs}$, se retiran del mapa de recursos del sistema de ficheros (*ListaLibres*) las páginas que habían sido reservadas para esta transacción (en *CopiaListaLibres*). Por lo tanto, la transacción deja de tener páginas asignadas temporalmente (ahora son definitivas). A continuación, las páginas liberadas por la transacción se añaden tanto al mapa de recursos del sistema de ficheros (*ListaLibres*) como a la copia (*CopiaListaLibres*), y también se liberan los cerrojos obtenidos por la transacción.

Finalmente, antes de confirmar la transacción a T_i, ST envía síncronamente las modificaciones al proceso de respaldo, R, para garantizar la durabilidad de la transacción. Cuando T_i recibe la confirmación de ST, anota que no dispone de cerrojos ni de páginas libres en las listas locales y confirma a su vez la transacción al cliente.

```
ST, al recibir mensaje de Ti con operación TerminarT:
{
    Ti debe tener en marcha una transacción
    LiberadosTi ← bloques que libera la transacción

    m.datos ← D_págs
    RecibirMensaje(Ti, m, incremental)

    ListaLibres ← ListaLibres - AsignadasTi
    AsignadosTi ← ∅

    ListaLibres ← ListaLibres ∪ LiberadosTi
    CopiaListaLibres ← CopiaListaLibres ∪ LiberadosTi
    LiberadosTi ← ∅

    mientras ( CerrojosTi != ∅ ) {
        liberar cerrojo
        CerrojosTi ← CerrojosTi - cerrojo
    }
    marcar Ti sin transacción en marcha

    m.op ← EscribirR
    m.destincr ← destino incr. de proceso R
    m.datos ← D
    EnviarMensaje(R, m, incremental)
    RecibirMensaje(R, OK)

    EnviarMensaje(Ti, OK)
}
```

Ilustración 6-10: Algoritmo para terminar una transacción en el proceso ST.

Abortar transacción

A diferencia de cuando se termina una transacción, cuando se aborta, T_i no necesita enviar a ST información sobre cerrojos ni sobre páginas, ya que las páginas que se hayan liberado durante la transacción continuarán ocupadas al desecharse los cambios y ST mantiene información sobre las páginas asignadas temporalmente a la transacción, así como sobre los cerrojos que ésta ha obtenido. Este aspecto es importante, ya que, en caso de fallo de T_i (por ejemplo, generación de una dirección inválida), ST puede recuperar todos los recursos asociados a la transacción sin necesidad de colaboración por parte de T_i.[10]

Si el cliente solicita a T_i que aborte la transacción en curso, T_i envía un mensaje incremental a ST con la operación *AbortarT* y con la lista de páginas que ha modificado, para recibir una nueva copia. (Ver Ilustración 6-11). Para ello, utiliza la opción *negativo* en el mensaje incremental, con lo que el micro-núcleo genera esta lista y la recibe ST. A continuación espera la respuesta de ST, que contiene

[10] En VSTa, donde se utiliza comunicación orientada a conexión, el micro-núcleo genera un mensaje de desconexión cuando un cliente muere, de forma que el servidor (en este caso ST) puede llevar a cabo las acciones apropiadas, que en este caso se reducen a liberar los recursos asignados temporalmente a la transacción, ya que ninguna modificación sobre el sistema de ficheros se propaga a ST a menos que la transacción sea confirmada.

precisamente estas páginas, ya que ST utiliza la opción *mapa* e incluye en el mensaje incremental de respuesta la lista de páginas solicitadas por T_i, tal y como se recoge en la Ilustración 6-12.

```
Ti, al recibir mensaje con operación AbortarT:
{
      debe haber una transacción en marcha

      m.op ← AbortarT
      m.destincr ← destino incremental asociado a ST
      m.datos ← CTi
      EnviarMensaje(ST, m, incremental y negativo)

      m.datos ← CTi
      RecibirMensaje(ST, m, incremental)

      ListaLibres ← ∅
      Cerrojos ← ∅
      marcar que no hay transacción en marcha
      EnviarMensaje(Cliente, OK)
}
```

Ilustración 6-11: Algoritmo para abortar una transacción en los procesos T_i.

Las actividades desarrolladas por ST antes de responder a T_i consisten, precisamente, en liberar los recursos asignados temporalmente a T_i. En concreto, se devuelven a *CopiaListaLibres* las páginas reservadas por T_i y también se liberan los cerrojos que había obtenido. Como se ha destacado anteriormente, estas actividades también se realizan en caso de fallo de T_i, puesto que ST dispone de la información necesaria para hacerlo; aunque, evidentemente, sólo si la transacción aborta debido a una petición de T_i se enviará el mensaje incremental de respuesta.

```
ST, al recibir mensaje con operación AbortarT:
{
      Ti debe tener una transacción en marcha

      CopiaListaLibres ← CopiaListaLibres ∪ AsignadosTi
      AsignadosTi ← ∅

      mientras ( CerrojosTi != ∅ ) {
            liberar cerrojo
            CerrojosTi ← CerrojosTi - cerrojo
      }
      marcar Ti sin transacción en marcha

      m.destincr ← destino incr. asociado a Ti
      m.datos ← D
      m.mapa ← lista de págs. solicitadas por Ti
      EnviarMensaje(Ti, m, incremental y mapa)
}
```

Ilustración 6-12: Algoritmo para abortar una transacción en el proceso ST.

6.2.4 Estudio de las prestaciones

Las prestaciones de un servicio transaccional como el que se acaba de describir dependen de factores como la estructura de los datos del servicio (en este caso, de la estructura del sistema de ficheros), la distribución de procesos entre nodos, el ancho de banda y la latencia del medio de comunicación, y del coste de las operaciones incrementales.

El servicio de ficheros transaccional ha sido diseñado como ejemplo de aplicación transaccional que utiliza las operaciones incrementales, prestando especial atención a la implementación de la estructura del servicio transaccional y aprovechando al máximo el código disponible en la distribución de VSTa; aunque haya sido necesario, tal y como se ha explicado anteriormente, diseñar una nueva estructura para el sistema de ficheros. En estas circunstancias, las prestaciones del servicio no deben tomarse sino como una prueba de la factibilidad de utilizar esta estructura, tal y como veremos durante el análisis.

Para evaluar las prestaciones se ha comparado el coste de las operaciones del servicio transaccional de ficheros (*ctfs*), con las del sistema de ficheros que ha sido necesario diseñar (*cfs*), con las del servidor temporal de ficheros de VSTa (*tmpfs*) y con las del sistema de ficheros *FAT* (MS-DOS) de VSTa. Nótese que *cfs* y *tmpfs* son sistemas de ficheros residentes en memoria virtual, mientras que *FAT* es un sistema de ficheros alojado en disco. La comparación de *ctfs* con *cfs* y *tmpfs* se debe a que todos ellos son sistemas de ficheros residentes en memoria virtual. No obstante, *ctfs* es un sistema de ficheros permanente, pues tolera fallos tanto del nodo donde reside ST (y los T_i) como del nodo donde reside el proceso de respaldo[11]; y esta es la razón por la que sus resultados se comparan con los de un sistema de ficheros basado en disco, como es *FAT*.

Los experimentos se han desarrollado sobre la misma plataforma utilizada para el análisis de prestaciones de los mensajes incrementales, utilizando esa misma implementación de los servicios incrementales. Para *ctfs* se ha utilizado una configuración con 5000 páginas de 4 KB (unos 20 MB) y con todos los procesos residentes en el mismo nodo. Nótese que el coste de las operaciones sobre el sistema de ficheros no variaría si el proceso de respaldo estuviese en otro nodo, ya que el proceso ST no necesita intercambiar mensajes con él salvo para confirmar una transacción (operación *TerminarT*). Durante las pruebas se ha utilizado el mismo proceso cliente para invocar las operaciones de todos los servicios de ficheros y, en el caso de *ctfs*, se tenía un sólo proceso T_i en marcha (para atender a este cliente).

La Tabla 6-2 recoge una comparación de las operaciones de control en *ctfs* con el resto de servicios de ficheros. Como puede observarse, el coste de las operaciones es menor que para el resto de sistemas de ficheros siempre que el proceso T_i no necesite contactar con el proceso ST para atender la petición, es decir, cuando tiene los cerrojos adecuados y dispone de suficientes páginas libres en su lista local como para atenderla.

[11] Como se ha reseñado anteriormente, las pruebas para este análisis se han hecho con ST y el proceso de respaldo en el mismo nodo. La razón es que VSTa no permite el paso de mensajes a través de la red y todavía no se ha implementado esta facilidad; pero en su configuración final ST y *respaldo* deben residir en nodos diferentes.

Por lo tanto, solamente cuando el cliente accede por primera vez a un fichero, la operación correspondiente (crear, abrir o eliminar) tiene un coste elevado:[12]

Operación	ctfs		cfs	tmpfs	FAT
	Solicita a ST	No solicita			
Crear	71,3	2,1	4,1	3,0	3,7
Abrir	50,0	2,0	4,5	4,3	3,0
Cerrar	—	0,18	0,13	0,13	0,14
Eliminar	77,7	2,18	4,6	4,4	9,3

*Tabla 6-2: Comparación del coste de las operaciones de control del servicio transaccional de ficheros (*ctfs*) con* cfs*,* tmpfs *y* FAT*. (Tiempos expresados en ms.).*

La Tabla 6-3 recoge la comparación del coste de las operaciones de lectura y escritura para diferentes tamaños. A modo de ejemplo, se han considerado únicamente lecturas y escrituras secuenciales, de los tamaños especificados. En todos los casos, el proceso cliente ha utilizado un *buffer* de 8 KB como origen y destino de las operaciones. Como puede observarse, *ctfs* se compara favorablemente con el resto de servicios de ficheros, lo cual no es extraño en el caso de FAT, que es un sistema de ficheros basado en disco. En cambio, es sorprendente, en principio, que sea más eficiente que *cfs* o que *tmpfs* (especialmente que *cfs*, pues la organización del sistema de ficheros es idéntica). Esto se debe a que la gestión de bloques libres es más sencilla en T_i, que solicita a ST grupos de 16 páginas y que no tiene que gestionar él mismo la lista de bloques libres del sistema de ficheros completo.

Operación	Tamaño	ctfs	cfs	tmpfs	FAT
Leer	512 Bytes	1,1	1,6	1,6	26,8
	64 KB	48	60,6	59,4	105
	1 MB	736	967	949	1977
Escribir	512 Bytes	0,89	2,0	2,1	28,5
	64 KB	38,8	64,2	60,3	57,7
	1 MB	616	986	912	1853

*Tabla 6-3: Comparación del coste de las operaciones de lectura y escritura del servicio transaccional de ficheros (*ctfs*) con* cfs*,* tmpfs *y* FAT*. (Tiempos expresados en ms.).*

Por otra parte, el almacenamiento en *ctfs* es estable, a diferencia de los otros servicios de ficheros. No obstante, es necesario considerar el coste de confirmar una

[12] Nótese que al obtener un cerrojo, tal y como se ha descrito, T_i necesita recibir un mensaje incremental de ST con las modificaciones de transacciones terminadas recientemente y de ahí el elevado coste de estas operaciones. Además, el coste variará en función del número de páginas que deban ser copiadas de ST a T_i, y este número depende del número de páginas que hayan modificado las transacciones que han terminado desde que comenzó la transacción en curso en T_i.

transacción o de abortarla (operaciones *TerminarT* y *AbortarT*). La Tabla 6-4 recoge, precisamente, el coste de estas operaciones tras haber escrito diferentes cantidades de datos en un fichero creado durante una transacción anterior. Como se ha visto al analizar las prestaciones de las operaciones incrementales, uno de los factores más importantes del coste de los mensajes incrementales (que consumen la mayor parte del tiempo asociado a las operaciones transaccionales) es la sincronización con el robo de páginas. Nótese que este coste no depende tanto del tamaño de la región de memoria virtual como de la cantidad de páginas que se modifiquen y que los resultados obtenidos no son, por lo tanto, directamente generalizables. No obstante, el caso estudiado es un ejemplo de una situación en la que la utilización de este servicio de ficheros transaccional es factible.

Operación	Escribir		
Trans.	512 B	64 K	1 M
TerminarT	405	397	514
AbortarT	207	219	600

Tabla 6-4: Coste de confirmar y cancelar una transacción tras haber modificado (escribir) diferentes cantidades de datos. (Tiempos expresados en ms.).

Finalmente, es necesario resaltar que el coste de *TerminarT* depende de la ubicación de los procesos que conforman el servicio transaccional. En concreto, si el proceso de respaldo está en un nodo diferente a ST y los procesos T_i —y así poder garantizar la durabilidad de las transacciones—, será necesario considerar el coste de transferir a través del medio de comunicación las modificaciones debidas a la transacción, y añadirlo al coste recogido en la tabla.

6.2.5 Análisis de resultados

El servicio transaccional que se acaba de presentar prueba la factibilidad de desarrollar servicios transaccionales utilizando la estructura general descrita en el apartado anterior. Tal y como se ha destacado durante la exposición, la ventaja más importante de esta estructura es la facilidad con la que es posible transformar un servicio en transaccional.

Esta aplicación muestra la utilidad de los servicios propuestos para el núcleo. Dado que el objetivo de esta implementación era precisamente mostrar esta utilidad, en el diseño de la aplicación no se han aplicado otras técnicas que serían necesarias para completar la tolerancia a fallos, como procesos de guardia que detectasen el fallo de los procesos de la aplicación y se encargasen de regenerarlos automáticamente, ya que estas medidas no necesitan servicios especiales del micro-núcleo. No obstante, se ha comprobado que, tal y como se describe en el apartado 5.4, es posible recuperar el estado del sistema de ficheros tras la avería de cualquiera de los procesos que forman la aplicación, sin que se pierda ninguna modificación realizada por transacciones que hubieran sido completadas anteriormente.

Así mismo, merece la pena destacar que el servicio transaccional podría ser utilizado a otro nivel diferente al de usuario. En concreto, es perfectamente posible desarrollar una aplicación que utilice la interfaz extendida del servicio de ficheros de forma totalmente transparente para los usuarios. Por ejemplo, una aplicación que emplee ficheros índice para acelerar las operaciones con los ficheros de datos puede incluir en la misma transacción las operaciones necesarias para actualizar ambos tipos de fichero.

6.3 Conclusiones

El trabajo de implementación que se describe en este capítulo ha mostrado la factibilidad de utilizar los mensajes incrementales para implementar diferentes técnicas de tolerancia a fallos. Para ello, se han estudiado las prestaciones de los mensajes incrementales en función del tamaño de las regiones de memoria virtual involucradas y del número de páginas modificadas entre mensajes consecutivos. Este estudio se completa con los realizados para el caso de los puntos de recuperación dirigidos por el usuario y para el del servicio de ficheros transaccional.

De este estudio, es importante destacar que la utilización de copia en escritura ha sido un factor determinante en la eficiencia de la implementación, que resulta efectivo considerar casos particulares dependientes del tipo de región involucrado (el caso de las páginas que todavía no han sido llenadas con ceros) y que la influencia de los bloqueos por sincronización es suficientemente importante como para que sea recomendable considerar las operaciones incrementales al diseñar los mecanismos de sincronización.

La implementación del servicio transaccional de ficheros ha puesto de manifiesto la facilidad con que es posible convertir un servicio en transaccional basándose en las operaciones incrementales y utilizando la estructura propuesta en el capítulo anterior. El estudio de prestaciones realizado permite concluir que existen situaciones (como la descrita en este capítulo), en las que resulta práctico implementar servicios transaccionales como el de ficheros de esta forma. No obstante, es necesario tener presente que los resultados obtenidos no son directamente extrapolables para otras configuraciones. Aspectos como la relación entre el tamaño de las regiones de memoria y la memoria física disponible y, por lo tanto, de la actividad debida al intercambio, deben ser tenidos en cuenta, tal y como demuestra el estudio sobre los mensajes incrementales.

Capítulo 7

Discusión y conclusiones

Como resultado del trabajo de investigación que se ha presentado, se ha propuesto una arquitectura adecuada para la plataforma de propósito general tolerante a fallos, tanto para el soporte físico como para el soporte lógico; y también se han especificado los servicios relativos a la tolerancia a fallos que deben incluirse en el núcleo del sistema operativo, cubriendo así los dos objetivos iniciales. Además se ha comprobado, mediante el diseño de casos de ejemplo y la implementación —en el caso de las operaciones incrementales y el servidor transaccional de ficheros—, que sobre esta plataforma es posible desarrollar de forma sencilla y eficiente aplicaciones que utilicen las diferentes técnicas actuales de tolerancia a fallos.

7.1 Discusión de resultados

Considerando la finalidad básica de este trabajo de especificar una plataforma de propósito general tolerante a fallos, a continuación se discuten los resultados obtenidos durante la investigación, para evaluar las aportaciones realizadas a los entornos de desarrollo de aplicaciones tolerantes a fallos.

Garantía de funcionamiento y técnicas de tolerancia a fallos

Al introducir este trabajo de investigación, se ha resaltado que la finalidad de incluir técnicas de tolerancia a fallos en las plataformas de propósito general era, precisamente, la de aumentar la garantía de funcionamiento de este tipo de sistemas.

Para ello, tal y como se recoge en los objetivos de este trabajo, resulta necesario estimar la garantía de funcionamiento que se debe ofrecer y las técnicas de tolerancia a fallos que se deben incluir en el sistema para obtenerla. Sin embargo, el análisis realizado de la situación previa del desarrollo de aplicaciones tolerantes a fallos, ha puesto de manifiesto que no es recomendable dar una respuesta genérica a estas preguntas, sino que es más conveniente dejar abierta la posibilidad de que cada aplicación utilice las técnicas que le permitan obtener el mejor compromiso entre coste, prestaciones y garantía de funcionamiento. La razón básica es que son precisamente las aplicaciones las que deben ofrecer la garantía de funcionamiento, y no en el soporte físico o en el sistema operativo.

Además, tal y como se recoge en la bibliografía [Lee94][Loc94][Chi94], los fallos del soporte lógico deben ser incluidos en el conjunto de fallos que deben ser tolerados por el sistema.

Por todo ello, frente a propuestas que han buscado la transparencia como característica básica de una plataforma de propósito general tolerante a fallos, en este trabajo se defiende que el diseñador de aplicaciones tolerantes a fallos debe poder elegir la técnica de tolerancia a fallos que mejor se adapte a las necesidades de su aplicación.

Para permitirlo, el nivel de enmascaramiento de fallos debe ser el de aplicación, y la plataforma para el desarrollo debe ofrecer un conjunto completo de herramientas versátiles, eficientes y sencillas de utilizar.

Dada la dificultad para combinar estas cuatro características, se propone como solución distribuir las técnicas de tolerancia a fallos entre los niveles de utilidades y del núcleo del sistema operativo. En el núcleo se debe incluir un conjunto completo de servicios que sean versátiles y eficientes. Con estos servicios se pueden implementar las diferentes técnicas actuales en el nivel de utilidades, de forma que puedan ser incorporadas de forma sencilla a las aplicaciones en función de la garantía de funcionamiento que deban ofrecer. Al mismo tiempo, la existencia de estos servicios en el núcleo del sistema operativo permite el desarrollo de aplicaciones específicas que utilicen sus propias técnicas de tolerancia a fallos.

La arquitectura del sistema

Una vez establecido el nivel de enmascaramiento de fallos y cómo se deben distribuir las técnicas de tolerancia a fallos entre los diferentes niveles del sistema, se ha propuesto como arquitectura más adecuada para construir una plataforma de propósito general tolerante a fallos una arquitectura multicomputador en la que cada nodo ejecuta su propia copia del núcleo del sistema operativo, que tiene estructura de micro-núcleo.

Esta propuesta ha sido convenientemente justificada en términos de transparencia, flexibilidad, eficiencia, coste económico y fiabilidad; y también en base a su capacidad para permitir la distribución de las técnicas de tolerancia a fallos en dos conjuntos diferentes, tal y como se había establecido previamente.

La arquitectura propuesta incluye a un amplio rango de sistemas que van desde estaciones de trabajo conectadas mediante una red de área local hasta sistemas formados por placas alojadas en un mismo *rack* y con un medio de comunicación específico. Como consecuencia, los resultados obtenidos pueden ser aplicados a un amplio abanico de sistemas dentro del segmento seleccionado como objeto de la investigación.

La interfaz del micro-núcleo

Tras la propuesta de arquitectura y apoyándose en la revisión inicial de las técnicas actuales de tolerancia a fallos, se han analizado las necesidades de las aplicaciones tolerantes a fallos a nivel del micro-núcleo.

Como resultado de este análisis, se ha establecido la función que debe desempeñar el micro-núcleo en el desarrollo de aplicaciones tolerantes a fallos, que consiste en: permitir a los procesos guardar, recuperar y actualizar su estado; generar la información necesaria para mantener registros de los mensajes intercambiados por los procesos y de las interacciones entre los procesos y el micro-núcleo; facilitar la gestión eficiente del paso intermedio de mensajes; y contribuir a la detección y el confinamiento de errores.

También como resultado de este análisis, se ha establecido que el micro-núcleo no debe establecer a priori qué parte del estado de los procesos se va a guardar, dónde, ni cuando; que no debe encargarse de gestionar los registros; y que no debe incluir servicios de comunicación de grupo, ni encargarse de procesar los resultados de las instancias de procesos que utilicen técnicas de tolerancia a fallos basadas en redundancia modular. En cambio, se ha establecido que estas funciones deben ser desempeñadas por los niveles superiores del sistema.

Una vez definidas qué funciones debe desempeñar cada nivel, se ha especificado la interfaz que debe ofrecer el micro-núcleo para el desarrollo de aplicaciones tolerantes a fallos. Con la finalidad de que esta interfaz cubriese todas las funciones asignadas al micro-núcleo relativas a tolerancia a fallos, se ha propuesto un nuevo tipo de servicios: las operaciones incrementales. Estos servicios permiten que las aplicaciones puedan guardar, recuperar y actualizar su estado beneficiándose de la eficiencia de técnicas propias del núcleo del sistema operativo, como llevar a cabo actualizaciones incrementales o la copia en escritura. Al mismo tiempo, son muy versátiles, permitiendo que las aplicaciones establezcan qué datos se actualizan y cuándo.

Desarrollo de aplicaciones tolerantes a fallos

Como método para evaluar la interfaz propuesta para el micro-núcleo, se ha mostrado cómo implementar un subconjunto representativo de las técnicas de tolerancia a fallos. Se ha comprobado, de esta manera, que es posible, efectivamente, utilizar los servicios

del micro-núcleo para implementar de forma eficiente estas técnicas en el nivel de utilidades, mediante servidores y librerías.

Al mismo tiempo que ha permitido evaluar la interfaz, el trabajo realizado al diseñar cómo implementar diversas técnicas de tolerancia a fallos supone una primera aportación sobre cómo podría organizarse el nivel de utilidades de una plataforma tolerante a fallos de propósito general. De entre las estructuras propuestas, destaca la del servicio transaccional tolerante a fallos, que ha servido de base para el posterior diseño e implementación de un servidor de ficheros transaccional tolerante a fallos. Los primeros resultados obtenidos tras la implementación prueban la factibilidad, por una parte, de las operaciones incrementales y, por otra, de implementar servicios transaccionales utilizando la estructura propuesta.

Plataformas de propósito general tolerantes a fallos

Tras el trabajo realizado, se puede concluir que el complejo problema de construir plataformas de propósito general tolerantes a fallos puede ser dividido, de forma efectiva, entre los niveles del soporte físico y el núcleo del sistema operativo por una parte, y el de utilidades por otra. La clave de esta división es la interfaz propuesta para el núcleo del sistema operativo, que permite realizar de forma independiente el trabajo en los diferentes niveles del sistema.

Respecto al diseño del micro-núcleo, los primeros resultados obtenidos tras la implementación de los mensajes incrementales han puesto de manifiesto la importancia de un diseño adecuado del sistema de memoria virtual que permita una implementación eficiente de los servicios especificados. Por lo tanto, es conveniente resaltar que, aunque es posible incorporar los servicios propuestos a un micro-núcleo existente —como lo prueba la implantación en VSTa—, es conveniente considerar estos servicios al diseñarlo, para así permitir una implementación más eficiente.

Aunque no se ha considerado adecuado basar el diseño en características específicas del soporte físico, la existencia de un conjunto bien definido de servicios que deben ser ofrecidos por el micro-núcleo permite plantearse qué podría aportar el soporte físico para hacer más eficiente o, simplemente, facilitar su implementación. Se considera, sin embargo, que el soporte físico de una plataforma de propósito general no debe incluir técnicas elaboradas de tolerancia a fallos,[1] sino mecanismos básicos para su implementación en niveles superiores del sistema.

Respecto al nivel de utilidades, el trabajo realizado muestra la posibilidad de realizar diseños modulares que conduzcan a la existencia de componentes lógicos que puedan ser combinados entre sí o incorporados directamente a las aplicaciones de forma sencilla. El servidor de respaldo es un ejemplo de componente que ha sido reutilizado en diferentes diseños durante esta investigación; pero el servidor de puntos

[1] No se está discutiendo la conveniencia de utilizar técnicas relativamente sencillas como la utilización de códigos detectores (o detectores y correctores) de errores, sino la conveniencia de incluir técnicas complejas como el establecimiento automático de puntos de recuperación que contengan el estado global del sistema.

de recuperación o los servidores de registros aparecen igualmente como buenos candidatos para la reutilización.[2]

Teniendo en cuenta estas consideraciones, se considera que el presente trabajo realiza aportaciones significativas a los entornos de desarrollo de aplicaciones tolerantes a fallos. Estas aportaciones, junto con el trabajo a desarrollar en el futuro, se describen en el siguiente apartado.

7.2 Resumen de aportaciones y trabajo futuro

Para finalizar esta exposición, en este apartado se enumeran las principales aportaciones del trabajo de investigación que se ha descrito en esta memoria y se discuten las líneas de trabajo que han quedado abiertas. Las aportaciones más significativas son:

- Se han identificado el nivel de enmascaramiento de fallos y la forma de distribuir las técnicas de tolerancia a fallos entre los diferentes niveles del sistema como dos aspectos básicos del diseño de sistemas tolerantes a fallos.
- Se ha propuesto ofrecer al diseñador de aplicaciones tolerantes a fallos un conjunto completo de herramientas versátiles, eficientes y sencillas de utilizar; y distribuir estas herramientas entre los niveles de utilidades y del núcleo del sistema operativo.
- Se ha propuesto, como la más apropiada para las plataformas tolerantes a fallos de propósito general, una arquitectura multicomputador en la que cada nodo ejecuta su propia copia del núcleo del sistema operativo, que tiene estructura de micro-núcleo.
- Se han identificado las funciones que debe desempeñar el micro-núcleo y se han separado de las que deben ser incluidas en el nivel de utilidades.
- Se ha especificado una interfaz para el micro-núcleo que permita cubrir las necesidades para el desarrollo de aplicaciones tolerantes a fallos, proponiéndose un nuevo tipo de servicios: las operaciones incrementales.
- Se ha revisado esta interfaz, mostrándose cómo pueden implementarse un subconjunto representativo de las técnicas de tolerancia a fallos y proponiéndose, además, una nueva estructura para la implementación de servicios transaccionales tolerantes a fallos.
- Se han implementado los mensajes incrementales y un servidor transaccional de ficheros, y se han obtenido los primeros resultados experimentales, que prueban

[2] No se está intentando atribuirse el mérito de ser el primero en proponer los diseños modulares en el nivel de utilidades (existe mucho trabajo realizado en este campo, del que los sistemas descritos en los apartados 1.3.4 y 1.3.5 constituyen buenos ejemplos), sino destacar que el trabajo realizado por el doctorando se integra perfectamente en este tipo de diseños.

la factibilidad de utilizar tanto las operaciones incrementales como la estructura propuesta para los servicios transaccionales tolerantes a fallos.

La investigación expuesta en la presente memoria y que se acaba de resumir, deja abiertas varias líneas de trabajo que suponen una continuación del trabajo de investigación que se ha llevado a cabo. Las más significativas se resumen a continuación:

- En primer lugar, es necesario completar el trabajo de implementación de los servicios especificados para el micro-núcleo, para tener una plataforma sobre la que construir el siguiente nivel de herramientas para el desarrollo de aplicaciones tolerantes a fallos. La implementación de las operaciones incrementales ha puesto de manifiesto la importancia de los bloques por sincronización en el micro-núcleo. Por lo tanto, otro trabajo a realizar es el diseño de un sistema de memoria virtual que resulte adecuado para la implementación de estos servicios.

- El siguiente paso sería la investigación sobre cómo estructurar el nivel de utilidades, con el objetivo de obtener un conjunto de herramientas sencillas de utilizar que faciliten la incorporación de las técnicas de tolerancia a fallos a un mayor número de aplicaciones. En esta línea, tiene una especial importancia cubrir las necesidades relativas a la selección, votación y comparación de resultados en la implementación de técnicas basadas en redundancia modular, así como incluir servicios para la comunicación de grupo.

  Este trabajo ha sido emprendido por Francisco Pastor, en el Departamento de Informática y Electrónica de esta Universidad, para la elaboración de su Tesis Doctoral. Así mismo, se ha comenzado un segundo proyecto de Tesis Doctoral que se está llevando a cabo en el Departamento de Informática de la Universidad Jaume I de Castellón, con el objetivo de evaluar la posibilidad de utilizar la interfaz del micro-núcleo para desarrollar sistemas distribuidos de control en tiempo real y tolerantes a fallos.

- Dentro del nivel de utilidades, la estructura propuesta para el servicio transaccional requiere un estudio más profundo, que aborde problemas como la gestión del espacio dedicado al intercambio de páginas, algoritmos de reemplazo de páginas, o la forma de estructurar el servicio repartiendo sus datos en varias particiones para permitir así mayor concurrencia y una gestión más eficiente de la memoria.

- Respecto a la comunicación entre nodos, sería necesario comprobar la posibilidad de utilizar los mensajes sin copia y los mensajes incrementales empaquetados para conseguir transferir grandes cantidades de información entre espacios de direcciones de procesos en nodos diferentes sin llevar a cabo copias intermedias de los datos, y compararlos con otras soluciones a este problema, como [Bar91], [DP93] o [FAB+96].

- En el capítulo 3 se identifica la necesidad de integrar los diferentes métodos de detección y confinamiento de errores y realizar un estudio de la violación del comportamiento *fail-stop* o *fail-silent* a nivel de las aplicaciones. Este estudio debe confirmar o negar la necesidad de utilizar técnicas adicionales para la detección y el confinamiento de errores.

- Finalmente, la arquitectura de micro-núcleo no es la única que puede utilizarse para diseñar plataformas de desarrollo de aplicaciones tolerantes a fallos. Arquitecturas como Cache-kernel [CD94] o Exo-kernel [Eng95], junto a otras más recientes como Off [BF97], resultan alternativas interesantes que permitirían la implementación de los servicios propuestos en este trabajo y, tal vez, sugerirían otros nuevos, más adecuados a su nivel de abstracción.

Bibliografía

[AFM90] R. E. Ahmed, R.C. Frazier, P. N. Marinos. *Cache-Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multi-processors.* Proceedings of the 21st International Symposium on Fault-Tolerant Computing. Junio 1990.

[Bar91] J. S. Barrera III. *A Fast Mach Network IPC Implementation.* Proceedings of the Usenix Mach Symposium. November 1991.

[Ber88] P. A. Bernstein. *Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing.* IEEE Computer. February 1988.

[BF97] F. J. Ballesteros, L. L. Fernandez. *The Network Hardware is the Operating System.* Proceedings of the 6th Workshop on Hot Topics in Operating Systems. May 1997.

[BGJ+92] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid et altres. *Microkernel Operating System Architecture and Mach.* USENIX Micro-kernels and Other Kernel Architectures Workshop Proceedings. April 1992.

[BHB+90] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verissimo, L. Rodrigues, N. A. Speirs. *The Delta-4 Extra Performance Architectura (XPA).* Proceedings of the 20th Fault-Tolerant Computer Symposium, pp. 481-488. June 1990.

[BJ90] M. Banâtre, P. Joubert. *Cache Management in a Tightly Coupled Fault-Tolerant Multiprocessor.* Proceedings of the 20th Fault-Tolerant Computing Symposium. June 1990.

[BMR+91] M. Banâtre, G. Muller, B. Rochat y P. Sánchez. *Design Decisions for the FTM: A General Purpose Fault-Tolerant Machine.* Proceedings of the 21st Fault-Tolerant Computing Symposium. June 1991.

[Bor94] A. Borr. *Highly-Available Data Services for UNIX Client-Server Networks: Why Fault-Tolerant Hardware Isn't the Answer.* Hardware and Software Architectures for Fault Tolerance, pp. 117-121, Lecture Notes in Computer Science nº 774, Springer-Verlag. 1994.

[CD94] D. R. Cheriton, K. J. Duda. *A Caching Model of Operating System Kernel Functionality.* Proceedings of the First Symposium on Operating Systems Design and Implementation, pp. 179-193. November 1994.

[CDD90] F. Cristian, B. Dancey, J. Dehn. *Fault-Tolerance in the Advanced Automation System.* Proceedings of the 20th Fault-Tolerant Computing Symposium. June 1990.

[CDK96] G. Coulouris, J. Dollimore, T. Kindberg. *Distributed Systems. Concepts and Design.* Addison-Wesley, 2nd edition. 1996.

[Cer97] V. Cerverón. *Algoritmos de Soporte para Tolerancia a Fallos. Aplicación a la Comunicación Punto a Punto.* Tesis Doctoral, Dpto. de Informática y Electrónica, Universidad de Valencia. 1997.

[Che88] D. R. Cheriton. *The V distributed system.* Communications of the ACM, 31(3):314-333. March 1988.

[Chi94] R. Chillarege. *Top Five Challenges Facing the Practice of Fault Tolerance.* Hardware and Software Architectures for Fault Tolerance, pp. 3-12, Lecture Notes in Computer Science nº 774, Springer-Verlag. 1994.

[CPR+92] M. Chérèque, D. Powell, P. Reynier, J. L. Richier, J. Voiron. *Active Replication in Delta-4.* In Proceedings of FTCS-22, pp. 28-37. July 1992.

[CRD+95] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. 15th ACM Symposium on Operating Systems Principles, pp. 12-25. December 1995.

[DP93] P. Druschel, L. L. Peterson. *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility.* Proceedings of 14th ACM SOPS. December 1993.

[Dra90] R. Draves. *A Revised IPC Interface*. USENIX Mach Workshop Proceedings, pp. 101-122. October 1990.

[DVC+93] G. Deconinck, J. Vounckx, R. Cuyvers, R. Lauwereins. *Survey of Checkpointing and Rollback Techniques.* Technical Report of ESPRIT project 6731 (FTMPS). June 1993.

[Eln93] Elmootazbellah Nabil Elnozahy. *Manetho: Fault-Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication.* PhD Thesis, Rice University. October 1993.

[Eng95] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* 15th ACM Symposium on Operating Systems Principles, pp. 251-266. December 1995.

[FAB+96] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, K. Li. *Early Experience with Message-Passing on the SHRIMP Multicomputer.* Proceedings of ISCA '23. May 1996.

[Fab94] G. Fabregat. *Análisis de los Mecanismos de Establecimiento de Puntos de Recuperación y Vuelta Atrás en Sistemas Multiprocesadores con Memoria Compartida Basados en Bus.* Informe Técnico Universidad Jaume I, UPI 01-09/94. Septiembre 1994.

[Fab96] G. Fabregat. *Diseño de un Módulo de Proceso Tolerante a Fallos - Inclusión en una Arquitectura Multiprocesador*. Tesis Doctoral, Dpto. de Informática y Electrónica, Universidad de Valencia. Febrero 1996.

[Fut91] *Physical Layer and Profile Specifications.* IEEE P896.2. 1991.

[Gil96] P. J. Gil. *Garantía de funcionamiento: conceptos básicos y terminología.* Mayo 1996.

[Gra86] J. Gray. *Why Do Computers Stop and What Can Be Done About It?*. Proc. Fifth Symp. Reliability in distributed Software and Database Systems, IEEE Computer Society Press, Los Alamitos, California, pp. 3-12. 1986.

[HK93] Y. Huang, C. Kintala. *Software Implemented Fault Tolerance: Technologies and Experience.* Proceedings of the 23rd Fault-Tolerant Computing Symposium, pp. 2-9. June 1993.

[IEEE92] IEEE. *POSIX 1003.4: Real-Time Extensions for Portable Operating Systems*. IEEE, first edition. 1992.

[Int90] Intel Corporation. *i486 Microproccessor - Programmer's Reference Manual*. Intel Corporation. 1990.

[Jew91] D. Jewett. *Integrity S2: A Fault-Tolerant Unix Platform.* Proceedings of the 21st Fault-Tolerant Computing Symposium. June 1994.

[JF93] B. Janssens, W. K. Fuchs. *Relaxing Consistency in Recoverable Distributed Shared Memory.* Proceedings of the 23rd International Symposium on Fault-Tolerant Computing. June 1993.

[JSM+93] M. Jöhnk, W. Schwartz, C. Morin, M. Clatin, A. Thomas. *FASST Microkernel - Design and Implementation*. FASST Project (ESPRIT P5212) Internal Report. April 1993.

[JZ87] D. B. Johnson, W. Zwaenepoel. *Sender-Based Message Logging.* Proceedings of 17th Fault-Tolerant Computing Symposium, pp. 14-19. July 1987.

[JZ88] D. B. Johnson, W. Zwaenepoel. *Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing.* Proceedings of 7th Annual Symposium on Principles of Distributed Computing, pp. 171-181. August 1988.

[LA94] S. Levi, A. K. Agrawala. *Fault Tolerant System Design.* McGraw-Hill. 1994.

[LAB+95] J.-C. Laprie, J. Arlat, C. Béounes, K. Kanoun. *Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures.* Predictably Dependable Computing Systems, pp. 103-122, Basic Research Series, Springer. 1995.

[Lam78] L. Lamport. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, 21(7):558-565. July 1978.

[Lap92] J.-C. Laprie (ed.). *Dependability: Basic Concepts and Terminology.*. Dependable Computing and Fault-Tolerance (5), Springer-Verlag. 1992.

[Lee94] P. A. Lee. *Sofware-Faults: The Remaining Problem in Fault Tolerant Systems?* Hardware and Software Architectures for Fault Tolerance, pp. 171-182, Lecture Notes in Computer Science nº 774, Springer-Verlag. 1994.

[Lev94] Y. Levendel. *Fault Tolerance Cost Effectiveness.* Hardware and Software Architectures for Fault Tolerance, pp. 13-20, Lecture Notes in Computer Science nº 774, Springer-Verlag. 1994.

[LI93] I. Lee, R. K. Iyer. *Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System.* Proceedings of the 23rd Fault-Tolerant Computer Symposium, pp. 20-29. June 1993.

[Loc94] C. Douglass Locke. *Fault Tolerant Applications Systems; A Requirements Perspective.* Hardware and Software Architectures for Fault Tolerance, pp. 21-25, Lecture Notes in Computer Science nº 774, Springer-Verlag. 1994.

[Loe91] K. Loepere. *MACH 3 Kernel Principles.* OSF Mach 3 series. March 15, 1991.

[Mar97] R. J. Martínez. *Validación Experimental por Inyección Física de Fallos de la Garantía de Funcionamiento de un Sistema Multiprocesador Tolerante a Fallos*. Tesis Doctoral, Dpto. de Informática y Electrónica, Universidad de Valencia. 1997.

[MF92] C. Morin, C. Frankenfeld. *FASST Recovery Protocol and Stable Memory Specification*. FASST Project (ESPRIT P5212) Internal Report. September 1992.

[MPF+95] R. J. Martínez, C. Pérez, G. Fabregat, J. A. Boluda, F. Pardo. *DPU: a FB+ based fault-tolerant system*. Proceedings of the OBS'95. October 1995.

[MS92] H. Madeira, J. G. Silva. *On-Line Signature Learning and Checking*. Dependable Computing for Critical Applications. Vol. 6 in the series: Dependable Computing and Fault-Tolerance. 1992.

[MS94] H. Madeira, J. G. Silva. *Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking*. Proceedings of the 24th Fault-Tolerant Computing Symposium, pp. 350-359. June 1994.

[Ors93] R. Ors. *Sistema Multiprocesador Tolerante a Fallos Basado en Puntos de Recuperación*. Tesis Doctoral, DISCA, Universidad Politécnica de Valencia. Septiembre 1993.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, K. Li. *Libckpt: Transparent Checkpointing under Unix*. USENIX Winter 1995 Technical Conference, pp. 213-223. January 1995.

[PF+97] C. Pérez, G. Fabregat. *Los mensajes incrementales: una herramienta para el desarrollo de aplicaciones tolerantes a fallos*. Actas de las VIII Jornadas de Paralelismo. Septiembre 1997.

[Pla93] J.S. Plank. *Efficient checkpointing on MIMD architectures*. PhD Thesis, Department of CS, University of Princeton. Junio 1993.

[Pow94] D. Powell. *Distributed Fault Tolerance: Lessons from Delta-4*. IEEE Micro, pp. 36-47. February 1994.

[PPT+92] D. Presotto, R. Pike, K. Thompson, H. Trickey. *Plan 9, A Distributed System*. Micro-kernels and Other Kernel Architectures Workshop. April 1992.

[RAA+92] M. Rozier, V. Abrossimov, F. Armand et altres. *Overview of the Chorus Distributed Operating System*. Micro-kernels and Other Kernel Architectures Workshop. April 1992.

[RS94] M. Russinovich, Z. Segall. *Application-Transparent Checkpointing in Mach 3.0/UX*. Proceedings of the 28th Hawaii International Conference on System Sciences. January 1994.

[RSS93] M. Russinovich, Z. Segall, D. Siewiorek. *Application Tansparent Fault Management in Fault Tolerant Mach*. Proceedings of the 23rd Fault-Tolerant Computer Symposium, pp. 10-19. June 1993.

[RT92] R. van Renesse, A. S. Tanenbaum. *Short Overview of Amoeba*. Microkernels and Other Kernel Architectures Workshop. April 1992.

[Rus94] M. E. Russinovich. *Application-Transparent Fault Management*. PhD Thesis, Carnegie Mellon University. August 1994.

[SB89] N. A. Speirs, P. A. Barrett. *Using Passive Replicates in Delta-4 to provide Dependable Distributed Computing*. Proceedings of the 20th Fault-Tolerant Computer Symposium, pp. 184-190. June 1989.

[SCM+96] J. G. Silva, J. Carreira, H. Madeira, D. Costa, F. Moreira. *Experimental Assessment of Parallel Systems*. Proceedings of the 26th Fault-Tolerant Computing Sysmposium. June 1996.

[SRC84] J. H. Saltzer, D. P. Reed, D. D. Clark. *End-To-End Arguments in System Design*. ACM Transactions on Computer Systems, Vol. 2, Nº 4, pp. 277-288. November 1984.

[SS92] D. P. Siewiorek, R. S. Swarz. *Reliable Computer Systems. Design and Evaluation*. Digital Press, 2ª Edición. 1992.

[Sta92] R. Stainov. *Intelligent Ports Supporting Fault-Tolerance*. In Proceedings IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp. 44-51. 1992.

[Sta97] W. Stallings. *Sistemas Operativos*. Prentice Hall, 2ª Edición. 1997.

[Tan95] A. S. Tanenbaum. *Distributed Operating Systems*. Ed. Prentice-Hall. 1995.

[TV91] M. Tamer Oszu, P. Valduriez. *Principles of distributed database systems*. Prentice-Hall. 1991.

[Vah96] U. Vahalia. *UNIX Internals - The New Frontiers*. Prentice Hall. 1996.

[Val] A. Valencia. *An Overview of the VSTa Microkernel.* http://bodhi.zendo.com/vsta/vsta_intro.html

Anexo I

Tolerancia a fallos: conceptos básicos y metodología de diseño

En este anexo se describen de forma precisa los conceptos básicos utilizados en la presente memoria, con la finalidad de permitir una descripción clara de las técnicas de tolerancia a fallos, de los sistemas tolerantes a fallos, y del trabjo realizado.

A.I.1 Concepto de sistema

Sin duda alguna, cualquier definición de los conceptos relacionados con la tolerancia a fallos necesita basarse en el concepto de sistema. Para describir este concepto, es usual realizar una definición desde dos puntos de vista. Desde el punto de vista externo al sistema, éste se puede definir de la siguiente forma:

> *Un **sistema** es cualquier mecanismo identificable que mantiene un patrón de comportamiento observable en la interfaz con su entorno. La interfaz es, simplemente, el lugar de interacción entre distintos sistemas. [LA90, p. 14][LA94, p. 6]*

Desde este punto de vista, el sistema es considerado como una caja negra por sus usuarios, siendo un **usuario** otro sistema (humano o físico) que interacciona —a través de la interfaz—con el sistema considerado. Al conjunto de usuarios de un sistema se le denomina su **entorno** y, finalmente, el comportamiento que perciben los usuarios de un sistema se denomina **servicio**.

La definición anterior describe un sistema en términos de lo que hace; pero además, es necesario completar el concepto de sistema con una descripción de lo que el sistema es:

> *Un **sistema** se define como un conjunto de componentes que interaccionan bajo el control de un diseño. (Tanto los componentes como el diseño de un sistema son, a su vez, sistemas). [LA90, p. 14]*

Esta definición, claramente recursiva, muestra la relatividad de la noción misma de sistema según el punto de vista adoptado [Gil96, p. 5][Lap92]. El diseñador de un sistema lo verá tal y como recoge la segunda de las definiciones, como un conjunto de componentes que él relacionará mediante el diseño. A su vez, es posible que varios de esos componentes hayan sido desarrollados en una etapa previa por el mismo u otro diseñador, estando compuestos a su vez de otra serie de componentes (sistemas). De esta forma es posible construir un sistema comenzando por la construcción de componentes de bajo nivel (circuitos integrados, por ejemplo) que se combinan mediante un diseño para dar lugar a componentes que ofrecen otros servicios de mayor nivel en la jerarquía del sistema (un módulo procesador o una memoria, por ejemplo), hasta llegar al sistema final, que será percibido por sus usuarios como una caja negra (un ordenador personal, por ejemplo). De igual forma, es posible realizar el proceso de construcción inverso, en el que primero se considera el diseño de un sistema en base a unos componentes que son construidos posteriormente, yendo del nivel más abstracto al más concreto.

En cualquier caso, el nivel en el que se detiene la recursión también depende del punto de vista adoptado. A este nivel los componentes del sistema se consideran **sistemas atómicos**, indicando que su estructura interna ya no puede ser distinguida, o no es de interés y puede ser ignorada. [LA90, p. 15]

A.I.2 Garantía de funcionamiento

Tanto en la descripción de este concepto como de sus atributos y, en general, a lo largo de este trabajo, se utilizan los términos empleados en [Lap92], donde se recogen las contribuciones a este campo de numerosos autores. Para la redacción en castellano de este trabajo se ha seguido la traducción de la referencia anterior, [Gil96].

> *La **garantía de funcionamiento** (*dependability*) de un sistema informático es la propiedad que permite a sus usuarios depositar una confianza justificada en el servicio que les entrega. [Gil96, p. 1]*

Esta propiedad es totalmente genérica y expresa la necesidad, cada vez mayor en nuestros días, de disponer de sistemas informáticos en los que se pueda depositar justificadamente una confianza que se torna rápidamente en dependencia, al no poder

prescindir de los sistemas informáticos una vez comienzan a desempeñar una determinada función. Dependiendo del tipo aplicación, esta garantía de funcionamiento conllevará implicaciones diferentes, que han sido denominadas genéricamente **atributos de la garantía de funcionamiento**:

- la **disponibilidad** (*availability*) es la garantía de que un sistema informático estará preparado para el servicio,
- la **fiabilidad** (*fiability*) es la garantía de que el sistema ofrecerá continuidad en el servicio,
- la **seguridad-inocuidad** es la garantía de que no se producirán averías catastróficas, y
- la **seguridad-confidencialidad** es la garantía de que el sistema evitará la manipulación no autorizada de información.

Cualquier sistema informático debe enfrentarse a circunstancias no deseadas —pero no siempre inesperadas— que serán la causa de un mal comportamiento del sistema, con lo que no se justificará la confianza en el servicio del mismo. Estas circunstancias son los fallos, los errores y las averías; y se denominan conjuntamente **impedimentos** de la garantía de funcionamiento.

- Se considera que se ha producido una **avería** (*failure*) cuando el servicio entregado por un sistema no se corresponde con la especificación, que debe ser una descripción adecuada del comportamiento que debería mostrar el sistema.
- Se denomina **error** (*error*) a la parte del estado del sistema que lo diferencia de un estado[1] válido [LA90, p. 41]. Por ejemplo, la existencia en la memoria de un valor que no se corresponde con el que se ha escrito previamente, o la ejecución por el procesador de una instrucción que no forma parte del programa. Es conveniente resaltar que, aunque las averías están causadas por errores, la existencia de errores en el sistema no implica necesariamente que se vaya a producir una avería. (Por ejemplo, el valor erróneo almacenado en la memoria puede ser sobrescrito antes de ser utilizado).
- Un **fallo** (*fault*) es cualquier defecto o imperfección, físico o de diseño, en algún elemento del sistema [Fab96, p. 7]. Por ejemplo, una perturbación electromagnética es un fallo que puede alterar el contenido de una celda de memoria, produciendo un error. Por lo tanto, los errores pueden considerarse como la manifestación en el sistema de un fallo.

Cualquier medida dirigida a obtener la garantía de funcionamiento requerida debe partir de un análisis detallado de sus impedimentos y, concretamente, de los fallos; ya que son la causa última tanto de errores como de averías, tal y como se ha visto.

[1] El concepto de estado del sistema no se ha introducido previamente por dos razones: por una parte, porque las definiciones recogidas en la bibliografía requieren una explicación que quedaría fuera del alcance de la presente introducción; y, por otra, porque la definición de estado usual —en su acepción: "Situación de un sistema físico (o lógico) determinada por los valores del conjunto de variables necesarias para su descripción." —resulta adecuada para la definición de error y el resto de la exposición.

En [SS92, pp. 22, 23] se lleva a cabo una clasificación de los fallos en permanentes, intermitentes y transitorios; atendiendo a su distribución en el tiempo. (No obstante, las definiciones se han tomado de [Fab96, p. 8]). Los fallos **permanentes** (o **duros**) son aquéllos cuya causa no desaparece a menos que sean reparados, y, por tanto, sus efectos están siempre presentes. Los **transitorios** (o **suaves**) son aquéllos cuya presencia está limitada en el tiempo, y que desaparecen sin necesidad de reparación. Finalmente, los **intermitentes** son aquéllos cuyas causas requieren reparación para ser eliminados del sistema, pero sus efectos no son permanentes, manifestándose sólo cada cierto tiempo, normalmente en determinados estados del sistema.

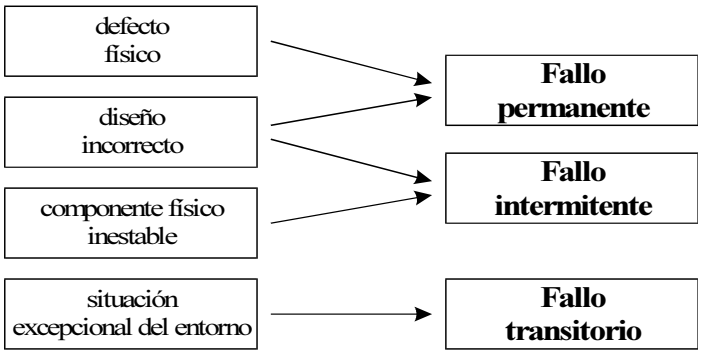

Ilustración A.I-1: Correspondencia entre el origen de los fallos y su duración.

En la Ilustración A.I-1 se muestra la correspondencia entre el origen de los fallos y su duración según la clasificación anterior. Como puede observarse, los fallos permanentes se producen como consecuencia de un defecto físico o de un diseño incorrecto del sistema (bien sea a nivel del soporte lógico o del soporte físico). Los fallos intermitentes pueden ser producidos por un diseño incorrecto o también por componentes físicos que se hayan degradado hasta llegar a un punto cercano a la avería. Finalmente, los fallos transitorios se producen como resultado de situaciones excepcionales del entorno, que desaparecen por sí solas.

Obtención de la garantía de funcionamiento

Los medios para alcanzar la garantía de funcionamiento requerida comprenden todas las fases de la vida de un sistema informático, tanto durante su desarrollo como durante su utilización. Tal y como puede verse en la Ilustración A.I-2, durante la fase de desarrollo del sistema se aplica la **prevención de fallos** (*fault-prevention*), cuya finalidad es la producción de un sistema libre de ellos.

La prevención de fallos comienza en la fase de diseño e implementación, durante la que se utilizan medios para la **evitación de fallos** (*fault-avoidance*) —por ejemplo, a nivel del soporte lógico, se emplearían técnicas de ingeniería del software—. Debido a que la evitación de fallos es imperfecta, es necesario un proceso de **eliminación de fallos** durante la fase de verificación del sistema, antes de que éste comience a ser utilizado.

No obstante, en muchos casos —especialmente en aquellos sistemas con necesidades especialmente elevadas de garantía de funcionamiento— es necesario considerar que el sistema en funcionamiento contendrá fallos que no habrán sido eliminados o que se habrán generado con el uso. La **tolerancia a fallos** se encarga, precisamente, de evitar que los fallos de existentes en un sistema en funcionamiento se traduzcan en averías.

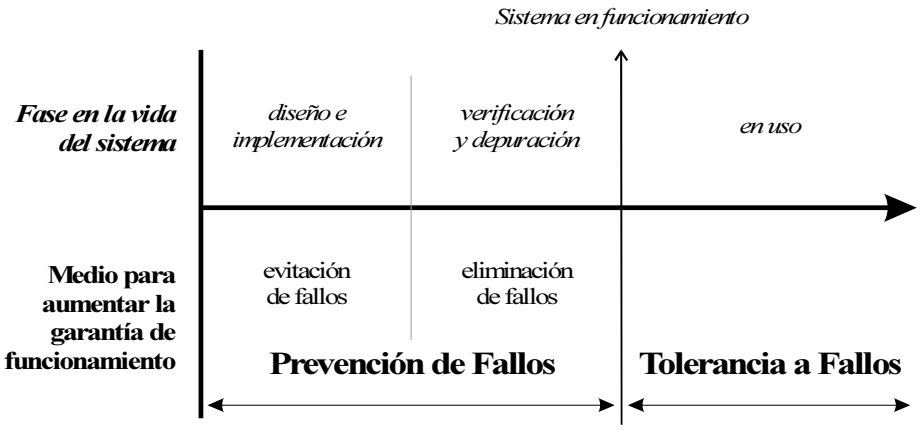

Ilustración A.I-2: Medios para la obtención de la garantía de funcionamiento.[2]

A.I.3 Tolerancia a fallos

La tolerancia a fallos supone dos actividades complementarias [Gil96, p. 16]. La primera, el **procesamiento de errores**, consiste en la eliminación de los errores del sistema, si es posible antes de que se produzca una avería. La segunda, **el tratamiento de los fallos**, tiene como finalidad impedir que se activen de nuevo los fallos, y así evitar la aparición de nuevos errores.

En algunos casos, es posible procesar los errores mediante **compensación**, dotando al sistema de redundancia suficiente como para que a partir de un estado erróneo sea posible continuar ofreciendo un servicio libre de errores. Por ejemplo, utilizando códigos correctores de errores para almacenar la información en memoria es posible mantener un servicio correcto incluso cuando existan errores internos, compensando el efecto de los errores producidos por un transistor defectuoso en una celda de la memoria. Si la compensación de errores se realiza de forma sistemática, sin realizar un intento explícito de detectar los errores (como en el caso anterior, por ejemplo), entonces se habla de **enmascaramiento de fallos**. No obstante, el enmascaramiento de fallos sin detección de errores puede acarrear una disminución no percibida de la

[2] Ilustración tomada de [Fab96, p. 4, ilustración 1.1], basada a su vez en [Lee90, p. 3, ilustración 1.1]

redundancia, por lo que es usual que las realizaciones prácticas de la técnica de enmascaramiento incluyan la detección de errores. [Gil96, p. 17]

La alternativa a la compensación de errores es la **recuperación**, que consiste en la sustitución del estado erróneo del sistema por otro libre de errores. En este caso, las actividades relacionadas con la tolerancia a fallos suelen clasificarse en una serie de etapas, atendiendo al orden en que deben ser ejecutadas. El número de etapas, así como las tareas que se llevan a cabo en cada etapa, difieren según el autor consultado. A continuación se describe la distribución de las tareas por etapas que se ha utilizado en el presente trabajo, y que es similar a las recogidas en [LA90, pp. 52-54], [LA94, pp. 235-238] y [Fab96, pp. 11-12].

Detección de errores

En la mayoría de los casos, la tolerancia a fallos requiere que los errores producidos por los fallos del sistema sean detectados explícitamente, como medio para distinguir entre estados correctos —o libres de errores— e incorrectos del sistema. Tal y como se afirma en [LA94], la detección de errores se puede realizar en paralelo con el funcionamiento normal del sistema y, al detectarse un error, iniciarse las medidas establecidas para evitar la avería.

Confinamiento y estimación de daños; diagnóstico de fallos

Desde que se produce un error hasta que se detecta transcurre un tiempo durante el cual es posible que la información errónea se propague a otras partes del sistema. Esta latencia en la detección de los errores hace que sea necesario establecer qué partes del sistema se han visto afectadas (han sido dañadas). La estimación de los daños producidos por un error puede realizarse utilizando técnicas estáticas, establecidas al diseñar el sistema, que se apoyen en su estructura; o técnicas dinámicas, que realicen comprobaciones en el sistema para refinar la estimación inicial (generalmente estática). Un ejemplo de técnica estática es utilizar una configuración con dos componentes cuyas salidas sean comparadas antes de que la información se propague al resto del sistema. En caso de error, los daños están confinados en este módulo. Por otra parte, comprobar la consistencia de una determinada estructura de datos (una lista enlazada, por ejemplo), es un ejemplo de técnica dinámica, que permite estimar si dicha estructura ha sido dañada.

El diagnóstico de los fallos consiste en la determinación del origen de los errores, para evitar que el mismo fallo vuelva a activarse de nuevo. Para ello, el sistema debe distinguir los errores propagados de los generados, y, en el caso de los generados, identificar los componentes físicos o lógicos que contienen los fallos que han generado dichos errores.

Recuperación del sistema

La recuperación del sistema supone la sustitución del estado erróneo en el que se encuentra el sistema debido a la presencia de errores por otro estado libre de ellos que sea consistente. La definición de estado consistente de [LA94] resulta adecuada para el presente trabajo:

> *"Un estado consistente debe ser uno de los estados alcanzables por el sistema en ausencia de errores (no necesariamente uno que haya ocurrido previamente) y debe ajustarse a la historia de eventos tal y como ha quedado reflejada en el comportamiento del sistema."*[3]

La primera parte de la definición garantiza que la evolución del sistema en presencia de errores es tan válida como la que se hubiera producido en ausencia de aquéllos. La segunda parte de la definición permite que los sistemas que interaccionen con éste no se vean afectados por una eventual vuelta atrás del mismo. En muchos casos no es posible modificar el estado del mundo exterior[4], especialmente cuando éste no es un sistema informático. Por ejemplo, si el sistema gestiona un cajero automático y ha proporcionado dinero en efectivo al usuario, no es posible deshacer esta acción; en cambio, el sistema debe garantizar que el estado final de la recuperación es compatible con el hecho (irremediable) de que se ha entregado este dinero.

Existen dos técnicas para realizar esta sustitución del estado erróneo. La primera, denominada **recuperación hacia atrás**, consiste en llevar el sistema a un estado anterior, que se sabe que está libre de errores y que se denomina **punto de recuperación** (*recovery point*). La segunda consiste en llevar al sistema a un nuevo estado realizando una serie de acciones y se denomina **recuperación hacia adelante**.

Antes de restablecer el servicio es necesario asegurarse de que el fallo que ha causado el error no vuelva a activarse inmediatamente. Si el fallo es permanente, será necesario **reconfigurar** el sistema para eliminar del mismo los componentes (lógicos o físicos) que contengan los fallos diagnosticados previamente.

Reparación y mantenimiento

Para evitar que las prestaciones, funcionalidad o fiabilidad del sistema vayan degradándose como consecuencia de sucesivas reconfiguraciones, es necesaria una política adecuada de reparación y mantenimiento. Como mínimo, los componentes averiados deben ser reemplazados. Y también es posible mantener una política preventiva de reemplazo de componentes para evitar fallos debidos al envejecimiento o desgaste de los mismos.

A.I.4 Metodología de diseño

Tal y como se recoge en [LA90], al añadir redundancia a un sistema su tamaño y su complejidad aumentan, y ello hace que la probabilidad de que se produzcan fallos tienda a aumentar. Puesto que la redundancia es necesaria para tolerar los fallos, es necesario tomar medidas adecuadas para conseguir que aumente la garantía de

[3] Del original en inglés [LA94, p. 237]: "*A* consistent *state of a system conforms with: the system correctly reachable states (the state has not necessarily been visited before); the events history as reflected in the system behavior (its interface).*

[4] Es común denominar conjuntamente **mundo exterior** de un sistema a todos aquellos sistemas que pueden interaccionar con él [Eln93, p. 6].

funcionamiento, en lugar de disminuir. Basándose en el modelo de sistema expuesto en el apartado A.I.1, en [Lee90] se recoge una metodología de diseño para sistemas tolerantes a fallos, que se ha seguido en el presente trabajo y que se expone a continuación.

Los sistemas, tal y como se ha descrito, tienen una estructura interna formada por un conjunto de componentes relacionados por un diseño. Cada uno de estos componentes puede ser considerado así mismo como un sistema con su propia estructura interna, dando lugar a una estructura jerárquica. En ausencia de impedimentos, el componente lleva a cabo su actividad normal atendiendo peticiones de sus usuarios. Para ello utiliza los servicios de los subsistemas que lo componen. Como puede observarse en la Ilustración A.I-3, existen tres tipos de situaciones excepcionales que pueden interrumpir su funcionamiento normal y hacerle pasar a la gestión de errores:

- Recibir una petición incorrecta de servicio. En este caso, el componente genera una excepción de la interfaz, ya que no le es posible satisfacer la petición recibida. El usuario del servicio es el responsable del tratamiento de dicha excepción.
- La detección de un error interno. En este caso, dependiendo de los mecanismos de recuperación incluidos en el componente, la gestión de errores puede devolver el componente a su funcionamiento normal o, si no puede recuperarse del error, generará una excepción por fallo, que indicará que este componente se ha averiado.
- Uno de sus subsistemas componentes genera una excepción, bien por fallo, bien debido a una excepción de la interfaz. Igual que en la situación anterior, el componente inicia la gestión del error, intentando volver al funcionamiento normal. Si no es capaz, generará una excepción por fallo que deberá ser manejada por el componente de nivel superior.

Para evitar que la mayor complejidad del sistema lleve a un mayor número de averías y poder ofrecer la garantía de funcionamiento requerida, resulta necesario evitar que las excepciones se propaguen hasta el usuario final del sistema, eligiendo adecuadamente las medidas de gestión de errores en cada uno de los componentes a lo largo de la jerarquía del sistema.

No obstante, las medidas de recuperación necesarias en un componente para tolerar el fallo de uno sus subsistemas dependerán del comportamiento del subsistema al averiarse. Por ello, tal y como se recomienda en [CDD90], es necesario completar la especificación de los componentes con su comportamiento en caso de avería (*failure semantics*). Por ejemplo, un servidor de comunicación a través de una red de área local puede garantizar que los datos recibidos son siempre correctos y que, en caso de avería, es posible que haya datos que no lleguen. Evidentemente, cuanto mayores sean las garantías que ofrezca un subsistema, más sencillo será la recuperación en el componente; pero más difícil y costoso será implementar dicho subsistema. Por eso, citando textualmente [CDD90, p. 8], "un aspecto clave del diseño de sistema tolerantes a fallos es cómo equilibrar los mecanismos de detección de errores, de recuperación y de enmascaramiento empleados en los diversos niveles de abstracción del sistema,

para conseguir el mejor compromiso posible entre coste, rendimiento y garantía de funcionamiento."

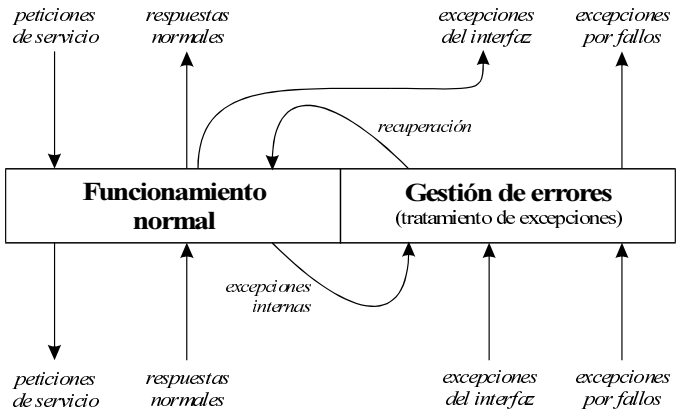

Ilustración A.I-3: Componente tolerante a fallos idealizado.[5]

[5] Esta ilustración ha sido tomada de la ilustración 1.2. de [Fab96, p. 10].

Anexo II

Interfaz detallada del paso de mensajes incrementales

El objetivo de este anexo es proporcionar una descripción detallada, e idependiente de la implementación, de una interfaz para el intercambio de mensajes incrementales (MIs) cuando éste se realiza utilizando los servicios *enviar* y *recibir*. En el anexo IV se describe la interfaz para un modelo de intercambio de mensajes más específico, el de cliente/servidor, aplicado al caso concreto de la implementación realizada sobre VSTa. Por sencillez, para esta descripción se supondrá que el origen y el destino de los mensajes son puertos y que, como es usual, sólo el proceso que crea un puerto puede leer de él.

La secuencia básica del paso de MIs se puede resumir como sigue: El emisor inicia la transmisión de un MI utilizando el servicio *enviar(puerto, msg)*. El núcleo utiliza la información de *msg* para construir una representación interna del mensaje, que depositará en el puerto. Finalmente, el receptor utiliza *recibir(puerto, msg2)*[1] y el núcleo utiliza la información de *msg2* para finalizar la transferencia de datos.

[1] La operación *recibir* del receptor puede iniciarse antes de que el emisor invoque el servicio *enviar*; pero en cualquier caso tendrá que esperar a que el mensaje esté disponible.

Tal y como se introduce en el apartado 4.1.4, al objeto de permitir la recepción de la cabecera de los MIs, es necesario dotar al puerto de una segunda cola de mensajes (a la que se ha denominado cola auxiliar), que contendrá los mensajes que hayan sido consumidos (mediante *recibir*); pero que no hayan sido proyectados en el espacio de direcciones del receptor. La estructura general de los puertos se muestra en la Ilustración A.II-1. El movimiento de mensajes entre la cola de mensajes y la cola auxiliar se describe en el apartado A.II.A.II.5 y las operaciones que puede realizar el emisor con los mensajes almacenados en la cola auxiliar se describen el apartado A.II.A.II.6.

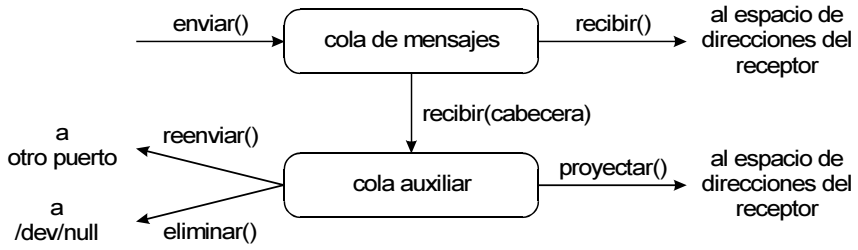

Ilustración A.II-1: La estructura de un puerto para el paso de MIs.[2]

A.II.1 Mensajes incrementales en el micro-núcleo

Con el doble objetivo de describir exhaustivamente las diferentes posibilidades de intercambio de MIs y de realizar una descripción independiente de la implementación, se considerará que el micro-núcleo utiliza para los MIs la representación interna que se describe en este apartado.

El MI consta de tres secciones: cabecera (*hdr*), mapa (*map*) y páginas (*pgs*).

- La **cabecera** contiene información dependiente de la implementación, que el núcleo puede transferir al receptor (por ejemplo, la identificación del emisor).
- El **mapa** contiene cuatro campos: *type*, *va*, *len* y *lst*.

  type Si *type* tiene el indicador NEG, que se describe en el apartado A.II.A.II.2, esto significa que el MI no contiene páginas (campo *pgs*) y que el MI está siendo utilizado para transferir únicamente la lista de páginas (campo *lst*). Además, *type* identifica el formato de la lista de páginas, que puede cambiar de una implementación a otra.

  va, *len* Dirección virtual de comienzo y longitud de la región de memoria virtual del emisor incluida en el MI.

[2] Se ha utilizado de nuevo la ilustración 4-6 para evitar, en la medida de lo posible, las referencias al capítulo 4.

lst Lista de las páginas de la región definida por *va* y *len* que se han incluido en el MI. (Ver campo *type*).

- Las sección **páginas** contiene las páginas virtuales descritas en el campo *lst* del mapa. Si el campo *type* tiene el indicador NEG esta sección está vacía, aunque *lst* no sea una lista vacía.

A.II.2 La estructura mensaje en el emisor

El emisor especifica el contenido del MI mediante el parámetro *mensaje* del servicio *enviar*. La estructura mensaje consta de los siguientes campos:[3]

- **Indicadores (flags)**. Mediante los indicadores incluidos en este campo el emisor puede determinar la interpretación del resto de los campos. (La Tabla A.II-1 describe brevemente cada uno de los indicadores).
- **Destino incremental (destincr)**. El destino incremental que debe utilizarse para establecer el conjunto del páginas a enviar. (Ver apartado 4.1.5).
- **Dirección virtual (va)**. Es la dirección de comienzo, en el espacio de direcciones del emisor, de la región a procesar.
- **Longitud (len)**. Tamaño de la región a procesar.
- **Máscara (lst)**. Este campo contiene la descripción de un subconjunto de las páginas que forman la región de memoria descrita por el par (dirección virtual de origen, longitud). Su interpretación depende de los indicadores. (Consultar la descripción de MASK y MAP en la Tabla A.II-1).

MASK[4]	La máscara indica el subconjunto de páginas que pueden haber sido modificadas. Si alguna página modificada no pertenece a este subconjunto el núcleo debe indicar al emisor esta circunstancia y no enviar el mensaje.
MAP	La máscara indica el subconjunto de páginas que deben ser enviadas, independientemente de que hayan sido modificadas o no (se ignora el campo destino y el estado de las páginas, que tampoco es modificado).
NEG	Sólo generar el mapa, no incluir ninguna página en los datos. El mapa se construye como si fuesen a enviarse las páginas, atendiendo al campo destino y al estado de las páginas, aunque éste no se modifica (la página no deja de estar modificada).
PACKED	Indica que la región de memoria virtual contiene un MIE. Si también se incluye el modificador NEG, el MIE no contiene realmente las páginas, sino solamente el mapa.

Tabla A.II-1: Descripción de los indicadores que puede utilizar el emisor.[5]

[3] Opcionalmente, la estructura mensaje puede contener otros campos o indicadores que dependan de la implementación; pero que no afectarán al protocolo para el intercambio de MIs.

[4] Los indicadores MASK y MAP son mutamente excluyentes.

A.II.3 Algoritmo enviar

En la Ilustración A.II-2 se describe el contenido del MI en función de las diferentes combinaciones de indicadores. Si el mensaje del usuario (*m*) incluye el indicador PACKED entonces el micro-núcleo interpreta la región de memoria virtual indicada por los campos *va* y *len* de *m* como un mensaje en formato empaquetado (MIE), utilizando directamente el mapa y las páginas del MIE para generar el MI interno. En este caso, el resto de indicadores no son aplicables y el usuario debe abstenerse de utilizarlos.

```
algoritmo enviar /* depositar mensaje en puerto */
entrada:
        puerto de destino
        estructura mensaje (m)
salida: resultado
{
        MI mi;

        si ( m.flags & PACKED ) {
                MIE *mie;

                no deben estar: MAP, MASK, NEG;
                mie = m.va;
                mi.map = mie->map;
                mi.pgs = mie->pgs;
        } si no {
                si ( mi.flags & MAP ) {
                        no debe estar: MASK;
                        mi.map.lst = m.lst;
                } si no {
                        si ( mi.flags & MASK )
                                páginas modificadas deben estar incluidas en m.lst
                        mi.map.lst = páginas modificadas;
                }
                mi.map.va = m.va;
                mi.map.len = m.len;
                si ( !(m.flags & NEG) )
                        mi.pgs = páginas en mi.map;
        }
        preparar mi.hdr
        incluir mi en la lista de mensajes del puerto de destino
        avisar al receptor si está esperando
        devolver correcto
}
```

Ilustración A.II-2: Algoritmo para el envío de mensajes.

[5] Los indicadores MAP, NEG y PACKED se corresponden, respectivamente, con las opciones *mapa*, *negativo* y *empaquetado* del apartado 4.2.2. El indicador MASK no ha sido introducido en el capítulo 4 porque las comprobaciones que permite pueden realizarse recibiendo la cabecera del MI y comprobando el contenido del mensaje antes de proyectarlo.

Cuando no se utiliza PACKED, si se utiliza el indicador MAP, se interpreta el campo *lst* de *m* como el conjunto de páginas que deben incluirse en el mensaje y se almacena en el campo *lst* del mapa del MI (*mi*). Si no se utiliza MAP, entonces *mi.map.lst* contendrá las páginas modificadas de la región definida por (*m.va*, *m.len*). Además, si se ha incluido el indicador MASK, se comprueba que las páginas modificadas estén incluidas en el conjunto especificado en *m.lst*. A continuación se rellenan los campos *va* y *len* del mapa a partir de sus homónimos en *m*. Finalmente, si no se ha incluido el indicador NEG, se rellena la sección páginas del MI. Si se incluye NEG quiere decir que el emisor está solicitando el conjunto de páginas indicado por *mi.map*, y las páginas no deben ser incluidas.

A.II.4 La estructura mensaje en el receptor

El receptor especifica la forma en que desea recibir el MI mediante el parámetro *mensaje* del servicio *recibir*. La estructura mensaje consta de los siguientes campos:

- **Indicadores (flags)**. Mediante los indicadores incluidos en este campo el receptor puede establecer el modo de recepción. (La Tabla A.II-2 describe brevemente cada uno de estos indicadores).
- **Dirección virtual (va)**. Tiene varios significados dependiendo de los indicadores. La puede utilizar el receptor, para indicar la región de destino de los datos, o el micro-núcleo, para indicarle al receptor dónde se han añadido los datos a su espacio de direcciones. (Ver el apartado A.II.A.II.5).
- **Longitud (len)**. Igual que el anterior.
- **Máscara (lst)**. Este campo indica las páginas que pueden ser actualizadas. Si el mensaje contiene alguna página no incluida en la máscara, *recibir* devuelve un error y no se modifica el espacio de direcciones del receptor.
- **Identificador de mensaje (msgid)**. Cuando el receptor utiliza el indicador HEADER, el núcleo deposita en este campo un número que identifica unívocamente a este mensaje en la cola auxiliar del puerto.

MASK	La máscara indica el subconjunto de páginas que pueden haber sido modificadas. Si alguna página modificada no pertenece a este subconjunto el núcleo debe indicar al emisor esta circunstancia y no enviar el mensaje.
SRCVA	Utilizar la dirección virtual incluida en el mapa del MI como dirección virtual de destino de los datos en el espacio de direcciones del receptor.
PACKED	El MI debe proyectarse empaquetado (MIE).
HEADER	Se consume el mensaje; pero se proyectan únicamente la cabecera y el mapa. El mensaje es transferido a la cola auxiliar del puerto. El identificador del mensaje en la cola es depositado en el campo *msgid*.

Tabla A.II-2: Descripción de los indicadores que puede utilizar el receptor.[6]

A.II.5 Algoritmo recibir

En la Ilustración A.II-3 se describe el comportamiento del servicio *recibir* en función de los indicadores incluidos en la estructura mensaje (*m*). En primer lugar, *recibir* siempre consume el mensaje, por lo que éste es retirado de la cola de mensajes del puerto (ver Ilustración A.II-1). A continuación se distinguen varios casos:

- **NEG**. El receptor obtiene la dirección de una nueva región de memoria virtual que contiene la sección *mapa* del MI. Además, el campo *flags* de *m* contiene el identificador NEG para indicarle al receptor que no se ha proyectado página alguna en su espacio de direcciones y que dispone del mapa en la región descrita por *m.va* y *m.len*.
- **HEADER**. El receptor obtiene, como en el caso anterior, el mapa del MI. No obstante, el campo flags no contendrá el indicador NEG. Además, tanto el mapa como las páginas del MI pasan a la cola auxiliar del puerto (el apartado A.II.A.II.6 describe las operaciones que el receptor puede realizar con los mensajes depositados en esta cola).
- **PACKED**. El receptor obtiene una nueva región de memoria virtual que contendrá, en formato empaquetado, el mapa y las páginas del MI. El receptor puede acceder a ella a través de *m.va* y *m.len*.
- **Resto de casos**. El MI contiene un conjunto de páginas que van a ser proyectadas en el espacio de direcciones del receptor. Si se incluye el indicador MASK se comprueba que las páginas descritas en *mi.map.lst* es un subconjunto de *mi.lst*. La región virtual de destino se obtiene de la siguiente forma: si se incluye el indicador SRCVA, se utiliza como destino la región indicada por *mi.va* y *mi.len*; si no se incluye y *m.va* es nula, se crea una nueva región de memoria virtual que es utilizada como destino; en otro caso, el destino es el indicado por *m.va* y *m.len*.

[6] Los indicadores SRCVA, PACKED y HEADER se corresponden, respectivamente, con las opciones *usar dirección de origen*, *empaquetado* y *cabecera*, descritas en el apartado 4.2.2.

```
algoritmo recibir /* obtener mensaje de puerto */
entrada:
        puerto
        estructura mensaje (m)
salida: resultado
{
        MI mi;

        esperar a que llegue un MI
        mi = sacar MI de la cola del puerto

        si ( mi.map.type & NEG ) {
                m.va = nueva región con mi.map
                m.len = longitud de la nueva región
                m.flags = NEG
                devolver correcto
        }
        si ( m.flags & HEADER ) {
                m.va = nueva región con mi.map
                m.len = longitud de la nueva región
                poner a mi en la cola auxiliar del puerto
                m.msgid = id. del mensaje en la cola auxiliar
                devolver correcto
        }
        si ( m.flags & PACKED ) {
                m.va =  nueva región con mi.map y mi.pgs
                m.len = longitud de la nueva región
                devolver correcto
        }

        /* se van a proyectar las páginas */
        si ( m.flags & MASK )
                mi.map.lst debe estar incluida en m.lst
        si ( m.flags & SRCVA ) {
                m.len = mi.map.len
                m.va = mi.map.va
                la región de destino debe ser válida
        } si no, si ( m.va == NULL ) {
                m.len = mi.map.len
                m.va = nueva región zero-fill-on-demand
        } si no {
                m.len debe ser mayor o igual que mi.map.len
        }
        proyectar mi.pgs a partir de m.va
        devolver correcto
}
```

Ilustración A.II-3: Algoritmo para la recepción de mensajes.

A.II.6 Gestión de la cola auxiliar del puerto

Los mensajes son transferidos desde la cola de mensajes del puerto a la cola auxiliar cuando el receptor utiliza el indicador HEADER en la invocación del servicio *recibir*. Una vez depositados en esta cola auxiliar, el receptor puede realizar las siguientes operaciones sobre ellos: proyectar las páginas de un mensaje (*proyectar*), eliminar un mensaje de la cola (*eliminar*), e incluir el mapa y las páginas en un nuevo mensaje (*reenviar*). Puesto que el receptor puede acceder a cualquier elemento de esta cola auxiliar, puede establecer el orden en que son proyectados los mensajes incrementales en su espacio de direcciones, o puede reenviarlos a otro proceso sin que las páginas hayan sido proyectadas en su espacio de direcciones. (Ver Ilustración A.II-1).

A continación se describen estas operaciones:

```
proyectar(puerto, id, va, len, lst, flags)
```

Completa la recepción del mensaje identificado por *id* que está almacenado en la cola auxiliar del *puerto*. Las páginas incluidas en el MI se proyectan en el espacio de direcciones del proceso. Los campos *va*, *len*, *lst* y *flags* tienen el mismo significado que sus homónimos en la estructura mensaje. No obstante, el indicador HEADER no está permitido en esta operación por carecer de significado.

```
eliminar(puerto, id)
```

Libera el espacio ocupado por el MI. Las páginas no son proyectadas en el espacio de direcciones del proceso.

```
reenviar(puerto, id, pdst, [, msg])
```

Extrae el MI *id* de la cola auxiliar de *puerto* y lo deposita en la cola del puerto *pdst*. Ninguno de los campos descritos al hablar de la estructura mensaje en el emisor, en el apartado A.II.A.II.2, tiene sentido en este caso, ya que el contenido del MI ya está definido. No obstante, todavía puede ser necesario que el proceso que reenvía el mensaje especifique parámetros adicionales que dependan de la implementación, por lo que se incluye en la interfaz, de forma opcional, el parámetro *msg*.

A.II.7 Ejemplos

En este apartado se describen, con mayor detalle, algunos de los ejemplos propuestos en el capítulo 4 al describir los mensajes incrementales y su utilización. Se repiten estos mismos ejemplos para clarificar, en términos de los parámetros e indicadores que

se acaba de describir, cómo se utilizan, a nivel de programación, los mensajes incrementales.

Actualización de copias de los datos críticos para la aplicación

En el caso que se describe en el apartado 4.1.1, en el que un proceso (P_1) desea actualizar la copia de los datos que mantiene otro (P_2), el algoritmo que deben seguir estos procesos se describe en las dos ilustraciones siguientes.

```
En P1:
{
        struct mensaje m;

        m.va = &datos;
        m.len = SIZE;
        m.flags = 0;
        m.lst = 0;
        m.destincr = destino de P2;
        enviar(puerto de P2, &m);
}
```

Ilustración A.II.4: Algoritmo para el proceso P_1.

```
En P2:
{
        struct mensaje m;

        m.va = &copia;
        m.len = SIZE;
        m.flags = 0;
        m.lst = 0;
        recibir(puerto de P1, &m);
}
```

Ilustración A.II.5: Algoritmo para el proceso P_2.

Actualización de copias a través del medio de conexión

P_1 y P_2 no deben necesitar modificar su comportamiento aunque no residan en el mismo nodo. La única diferencia es que el servidor de comunicación del nodo de P_1, al que denominaremos SC_1, debe recibir el mensaje y transferirlo a SC_2 (servidor de comunicación en el nodo de P_2) para que lo envíe a P_2. A continuación se muestran los algoritmos para SC_1 y SC_2. (Ver apartado 4.3.4).

```
En SC1:
{
        struct mensaje m;

        m.va = 0;
        m.len = 0;
        m.flags = PACKED;
        m.lst = 0;
        recibir(puerto imagen de P2,[7] &m);
        transmitir a través del medio[8]
}
```

Ilustración A.II.6: Algoritmo para el proceso SC_1.

```
En SC2:
{
        struct mensaje m;

        recibir del medio y almacenar en &buf
        m.va = &buf;
        m.len = len;
        m.flags = PACKED;
        m.lst = 0;
        m.destincr = 0;
        enviar (puerto de P2, &m);
}
```

Ilustración A.II.7: Algoritmo para el proceso SC_2.

Vuelta atrás

En el caso (ver apartado 4.1.3) de un proceso (P_1) que desea desechar los cambios realizados sobre su copia de los datos desde que envió el MI anterior a P_2, que mantiene otra copia, los procesos deberían incluir el algoritmo que se muestra a continuación.

[7] Con el fin de evitar dependencias de la implementación del servidor de comunicación, se ha utilizado aquí el término puerto imagen, queriendo indicar que el micro-núcleo depositará en este puerto los mensajes destinados al puerto de P_2, que reside en el nodo 2.

[8] SC_1 puede consultar la dirección de los datos recibidos en *m.va* y su longitud en *m.len*. En este ejemplo los datos habrían sido proyectados en una nueva región de su espacio de direcciones, en formato empaquetado.

```
En P1:
{
        struct mensaje m, m2;

        m.va = &datos;
        m.len = SIZE;
        m.flags = NEG;
        m.lst = 0;
        m.destincr = destino de P2;
        enviar (puerto de P2, &m);

        m2.va = &datos;
        m2.len = SIZE;
        m2.flags = 0;
        m2.lst = 0;
        recibir(puerto de P1, &m2);
}
```

Ilustración A.II.8: Algoritmo para el proceso P_1.

```
En P2:
{
        struct mensaje m, m2;

        m.va = &copia;
        m.len = SIZE;
        m.flags = 0;
        m.lst = 0;
        recibir(puerto de P2, &m);

        detecta que es una solicitud de las páginas incluidas en m.lst, y contesta:

        m2.va = &copia;
        m2.len = SIZE;
        m2.flags = MAP;
        m2.lst = m.lst;
        m2.destincr = 0;
        enviar(puerto de P1, &m2);
}
```

Ilustración A.II.9: Algoritmo para el proceso P_2.

Reordenación de mensajes en el receptor

En el apartado 4.1.4 se justifica la necesidad de llevar a cabo la reordenación de mensajes en la librería del proceso. Para este ejemplo, se va a suponer que existe un conjunto de réplicas que sigue el modelo líder/seguidor, tal y como se describe en [Bar+90] para la arquitectura XPA del proyecto Delta-4. Una réplica distinguida, el líder, es la responsable de tomar todas las decisiones que afectan al determinismo de la ejecución. En concreto, para la recepción de mensajes, establece el orden en el que los mensajes deben ser consumidos por todas las réplicas aplicando la regla de

precedencia adecuada (como prioridades o según el orden de llegada, por ejemplo) sobre el conjunto de mensajes pendientes. Después genera un mensaje de sincronización que indica a los seguidores el orden en el que deben consumir los mensajes.

Todas las réplicas, tanto el líder como los seguidores, tendrán la organización interna que se muestra en la Ilustración A.II-10. Cada réplica estará contenida en un proceso en el que que los hilos de la aplicación[9] obtienen los mensajes mediante la función de la librería para recibir un mensaje, *recibirL*. El proceso contiene otro hilo, el hilo receptor, que se encarga exclusivamente de ir recibiendo las cabeceras de los mensajes mediante la llamada al micro-núcleo *recibir*, e irlas añadiendo al conjunto de mensajes disponibles, de donde serán consumidos por los hilos de la aplicación. (Ilustración A.II-11).

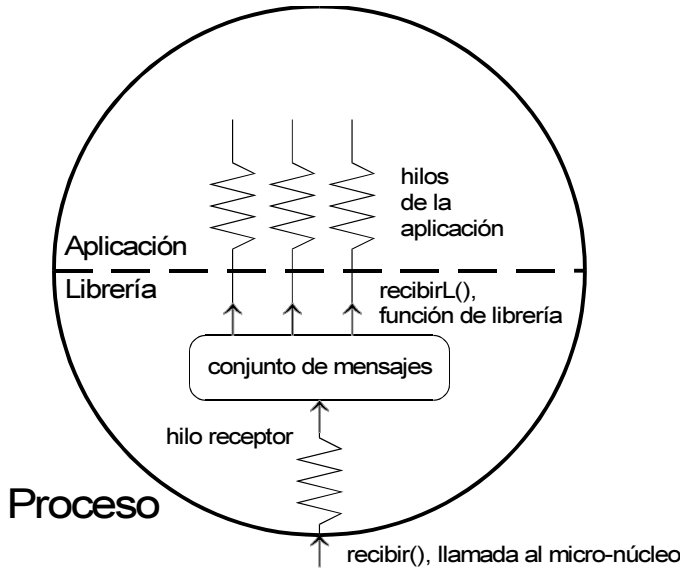

Ilustración A.II-10: Estructura de las réplicas.
(Es la misma en el líder y en los seguidores.)

[9] En XPA se exige que las aplicaciones sean deterministas. Aunque en la figura se muestren varios hilos de la aplicación, la utilización de varios hilos puede llevar a un comportamiento no determinista si el orden o el contenido de los mensajes generados dependen de la sincronización entre los hilos.

```
hilo receptor()
{
      mientras ( CIERTO ) {
            struct mensaje m = malloc(...);

            m.va = NULL;
            m.len = 0;
            m.flags = HEADER;
            m.lst = 0;
            recibir(puerto del líder, &m);
            añadir m al conjunto de mensajes pendientes
      }
}
```

Ilustración A.II-11: Algoritmo para el hilo receptor.
(Es el mismo en el líder y en los seguidores).

La función *recibirL* es diferente en el líder y en los seguidores. En el líder, se selecciona uno de los mensajes disponibles en el conjunto utilizando la regla de precedencia adecuada. Después, se notifica la decisión a las réplicas utilizando un mensaje ordinario (no incremental) y, finalmente, se hace accesible el mensaje a la aplicación, mediante la llamada al micro-núcleo *proyectar*, que proyecta los datos contenidos en el mensaje incremental elegido sobre el espacio de direcciones del receptor. (Ilustración A.II-12).

```
recibirL()
{
      struct mensaje mm;

      esperar a que el cjto. de mens. != vacío
      mm = extraer mensaje del conjunto aplicando el criterio de precedencia
      enviar mensaje ordinario a todas las répicas notificando el mens. elegido
      /* hacer accesible mm a la aplicación */
      proyectar(..., mm.id, ...)
}
```

Ilustración A.II-12: Función de librería recibirL*, versión para el líder.*

En los seguidores, la función *recibirL* espera a recibir del líder la identidad del siguiente mensaje a ser consumido, después espera a que el mensaje esté disponible (si no lo está ya) y, finalmente, hace accesible el mensaje a la aplicación de la misma forma que en el líder. (Ilustración A.II-13).

```
recibirL()
{
        struct mensaje mm;

        recibir mensaje ordinario
        esperar a que el mensaje indicado por el servidor esté en el cjto. de mens.
        mm = extraer mensaje del conjunto
        /* hacer accesible mm a la aplicación */
        proyectar(..., mm.id, ...)
}
```

Ilustración A.II-13: Función de librería recibirL*, versión para los seguidores.*

De esta forma, todas las réplicas ven modificado su espacio de direcciones en el mismo orden, el determinado por el líder. La posibilidad de separar la recepción de la cabecera del MI de la proyección de las páginas es la que permite que sea el código de librería el que se encargue de coordinar las réplicas entre sí. De lo contrario, sería necesario un proceso intermedio por cada réplica que recibiera los MIs en su forma empaquetada. Estos procesos llevarían a cabo la reordenación de los mensajes sincronizándose entre sí y, finalmente, los irían enviando a sus réplicas respectivas en el orden acordado.

Anexo III

Descripción de Manetho

Tal y como describe su autor, Manetho consiste básicamente en cuatro protocolos que deben ser añadidos a un sistema distribuido para proporcionar tolerancia a fallos [Eln93, p. 9]. Estos protocolos mantienen, durante la ejecución libre de errores, un conjunto de estructuras de datos por cada Unidad de Recuperación (*Recovery Unit*, RU) del sistema. En caso de fallo, durante la recuperación se utiliza esta información para restaurar el estado del sistema. Aunque existen otros elementos auxiliares, las estructuras de datos básicas se describen a continuación:

- El Grafo de Antecedentes (AG). Por cada RU existe un AG que contiene, en todo momento, un resumen de la ejecución del sistema que ha llevado a la RU al estado actual [Eln93, p. 12]. Cada nodo del grafo representa un evento y los arcos expresan la relación "ocurrió antes" [Lam78] entre estos eventos. La información almacenada en el nodo depende del tipo de evento; pero es importante resaltar que no contiene una copia de los mensajes intercambiados entre RUs y que, en cambio, sí contiene una copia de los mensajes recibidos desde el mundo exterior.

  Del AG existen dos versiones, una volátil, almacenada exclusivamente en el nodo de la RU, y otra persistente, guardada en almacenamiento estable, a la que se denomina P_AG.

- El registro de mensajes. Cada RU tiene asociado un registro de mensajes, en el que se almacena una copia de los mensajes que ésta envía. Es una estructura de datos volátil, de la que no existe versión persistente.
- Puntos de recuperación. Por cada RU basada en recuperación hacia atrás existe en todo momento un punto de recuperación permanente. Durante el establecimiento de un nuevo punto de recuperación cada RU establece un segundo punto de recuperación tentativo (porque se convierte en permanente solamente si el protocolo finaliza con éxito, en otro caso es desechado).

  Los puntos de recuperación se mantienen en almacenamiento estable, y no hay versión volátil de los mismos.
- El registro de compromisos (*commit log*). Contiene el identificador de los mensajes que han sido enviados al mundo exterior. Se mantiene exclusivamente en almacenamiento estable.

La implementación de los protocolos requiere que se añadan algunas actividades en determinados puntos de ejecución de las aplicaciones, de forma que se mantengan debidamente actualizadas las estructuras de datos que se acaban de describir.[1] A continuación se describen estas actividades para cada uno de los protocolos. Nótese que la descripción que sigue no pretende ser lo suficientemente detallada como para que el lector pueda comprobar la validez de los protocolos, para ello se le remite al trabajo original de [Eln93]. En cambio, en la descripción se detalla únicamente el tipo de operaciones que se llevan a cabo sobre cada una de las estructuras de datos básicas. Esta información será necesaria para justificar la propuesta de implementación que se describe en el apartado 5.3.

A.III.1 Protocolo de comunicaciones

Durante la ejecución libre de errores, todas las RUs utilizan el mismo protocolo de comunicaciones, que controla la comunicación entre RUs de la aplicación, así como la interacción de la aplicación con el mundo exterior.

Cuando una RU envía un mensaje a otra, se invoca el procedimiento *Enviar*. Tal y como se muestra en la Ilustración A.III-1, antes de que el mensaje sea realmente transmitido, se le añade una cabecera que contiene el AG. Esta cabecera será procesada por el código para la recepción de mensajes, antes de pasárselo a la aplicación, de forma que el protocolo es transparente para las aplicaciones. También es necesario guardar una copia del contenido del mensaje en el registro de mensajes de la RU

[1] La finalidad de Manetho, tal y como se introdujo en el apartado 1.3.3, es proporcionar tolerancia a fallos transparente para las aplicaciones. Por lo tanto, estas actividades deben ser añadidas a los niveles de utilidades y del núcleo del sistema operativo. En este apartado, el término *aplicación* se refiere al código en el nivel de aplicaciones (que desarrollarán los programadores de la aplicación específica); en cambio, el término *RU* se refiere al código de las librerías de Manetho, que son independientes de la aplicación.

(volátil); pero la copia se realiza después de transmitir el mensaje hacia la RU de destino, disminuyendo la sobrecarga en la operación de enviar.

Cuando el mensaje llega a la RU de destino, antes de pasárselo a la aplicación, se extrae el AG incluido en el mensaje y se añade al AG del receptor. La recepción de un mensaje es en sí misma un evento, por lo que también se añade un nuevo nodo que lo describa al AG del receptor.

Enviar:
incluir AG en el mensaje
transmitir mensaje
guardar copia en su registro de mensajes

Recibir:
AG ← AG ∪ AG del mensaje
añadir nuevo nodo al AG
pasar mensaje a la aplicación

Ilustración A.III-1: Algoritmo para enviar y recibir mensajes de otras RUs.

La comunicación con el mundo exterior requiere un comportamiento diferente, tal y como se recoge en la Ilustración A.III-2. Cuando se envía un mensaje al mundo exterior es necesario adquirir el compromiso de que, en caso de vuelta atrás, el estado de la RU reflejará el haber enviado esta información. Para ello se anota el identificador del mensaje en el registro de compromisos (*commit log*) y se vuelca el contenido del AG sobre su versión persistente, con lo que el AG queda vacío. Todas estas acciones se deben llevar a cabo de forma atómica, para garantizar la consistencia del registro de compromisos y del AG con el estado del mundo exterior.

De igual forma, cuando se recibe información del exterior, se debe garantizar que no va a ser necesario solicitarla de nuevo. Por ello, en el nodo que se añade al AG, se incluye una copia de la información (al contrario de lo que ocurre con los mensajes recibidos de otras RUs, a las que sí se les puede solicitar el reenvío del mensaje). No obstante, el algoritmo de entrada desde el mundo exterior, tal y como se destaca en [Eln93, p. 23] se apoya en la suposición de que el mensaje recibido desde el mundo exterior será respondido (para reconocerlo) de forma inmediata, con lo que se pasará el contenido del AG a almacenamiento estable.

Salida al ME: (Atómicamente)
añadir id. al Registro de Compromisos
volcar AG sobre P_AG
transmitir mensaje al ME

Entrada desde el ME:
añadir nuevo nodo al AG
pasar mensaje a la aplicación

Ilustración A.III-2: Algoritmos enviar y recibir información del mundo exterior (ME).

A.III.2 Protocolo de establecimiento de puntos de recuperación

Las RUs que utilizan recuperación hacia atrás utilizan un protocolo para el establecimiento de puntos de recuperación coordinados. Este protocolo permite garantizar que, en todo momento, existe un conjunto consistente de copias del estado de estas RUs en almacenamiento estable.

Respecto a este protocolo, existen dos tipos de RU. Una de las RUs actúa de coordinadora, mientras que el resto sigue sus instrucciones. En el caso de la coordinadora, tal y como se muestra en la Ilustración A.III-3, el algoritmo comienza enviando una notificación a las demás y estableciendo un punto de recuperación tentativo. Después, espera la respuesta del resto de las RUs. Si todas han establecido con éxito el PdR tentativo, todas ellas lo convierten en permanente al recibir la notificación de éxito de la RU coordinadora. En otro caso, todas ellas lo desechan al recibir el correspondiente mensaje de cancelación.

PdR en la RU coordinadora:
enviar mens. notificación al resto de RUs
establecer PdR tentativo
recoger reconocimientos
si (todas han tenido éxito)
enviar mens. éxito
convertir en perm. el PdR
desechar el PdR anterior
si no
enviar mens. cancelación

Resto de RUs:

Recibe mensaje de notificación:[2]
establecer PdR tentativo
enviar reconocimiento al coordinador
Recibe mensaje de éxito:
convertir en permanente el PdR
desechar el PdR anterior
Recibe mensaje de cancelación:
desechar el PdR tentativo

Ilustración A.III-3: Algoritmos para establecer puntos de recuperación coordinados. El algoritmo para la RU coordinadora se muestra en la parte de la izquierda y el del resto de RUs en la parte derecha.

Es importante resaltar que todas las RUs participan en el establecimiento de un punto de recuperación, tanto las basadas en recuperación hacia atrás como las basadas en la replicación de procesos. El protocolo en todas ellas es el que se acaba de describir, aunque en el caso de las RUs replicadas el punto de recuperación se establece mediante la sincronización de sus estados y solamente el líder del conjunto de réplicas responde a la RU coordinadora.

[2] El resto de RUs también pueden detectar que se está intentando establecer un punto de recuperación coordinado mediante una variable auxiliar, el contador de puntos de recuperación (CCN) que mantiene cada una de ellas y que también incluyen en cada mensaje que envían. Así, cuando una RU recibe un mensaje con un CCN mayor que el suyo, inicia también este algoritmo. Por supuesto, este contador se incrementa cada vez que la RU establece con éxito un punto de recuperación.

A.III.3 Protocolo de difusión

Las unidades de recuperación basadas en procesos replicados utilizan un protolo de difusión que garantiza que todas las réplicas del proceso (nivel de aplicación) procesan el mismo conjunto de mensajes y en el mismo orden. Así, es posible garantizar que, si las réplicas son deterministas, sus ejecuciones serán idénticas.

Por lo tanto, los mensajes dirigidos a un conjunto de RUs replicadas, denominado *troupe*, deben ser difundidos para que lleguen a todas las réplicas. Para determinar el orden en que se consumen los mensajes en las réplicas se utiliza el modelo líder/cohorte. Según este modelo, solamente una de las réplicas, la que actúa como líder, responde a los mensajes que recibe la *troupe*. Además, tal y como se muestra en la Ilustración A.III-4, el líder determina el orden en que se pasarán los mensajes al nivel de aplicación en todas las réplicas. El resto de las réplicas, las cohortes, antes de pasar un mensaje a la aplicación, esperan a recibir el mensaje del líder que les indique el orden en que debe ser consumido.

Recepción de un mensaje de aplicación en el líder:
establecer orden de consumo
pasar mensaje a la aplicación
enviar mens. con orden a las cohortes

Recepción de un mensaje de aplicación en una cohorte:
recibir mens. del líder con el orden
pasar mensaje a la aplicación

Ilustración A.III-4: Algoritmos para coordinar el procesamiento de mensajes en una RU replicada.

Además del protocolo básico, que se acaba de describir, el líder y las cohortes intercambian mensajes adicionales para tolerar pérdidas de mensajes (tanto de aplicación como de notificación de orden), y evitar que el estado de las réplicas diverja.

A.III.4 Protocolo de recuperación

Por último, el protocolo de recuperación permite que, en caso de fallo, las RUs afectadas puedan alcanzar un estado que sea consistente con el del resto del sistema sin afectar al mundo exterior. Al igual que en el caso del protocolo de comunicaciones, el protocolo de recuperación es común para todas las RUs del sistema, tanto si utilizan recuperación hacia atrás como si están basadas en la replicación de procesos.

En la Ilustración A.III-5 se muestra el protocolo de recuperación. Como puede verse, el protocolo comienza con la recuperación del estado de la RU que ha fallado. En el caso de RUs replicadas, el fallo de una cohorte simplemente requiere que se cree una nueva réplica a partir del estado del líder. Si ha fallado el líder, es necesario elegir a un nuevo líder y ejecutar el resto del algoritmo de recuperación, para garantizar que

la *troupe* alcanza un estado consistente con el resto del sistema.[3] En cambio, en el caso de una RU con recuperación hacia atrás, existe un punto de recuperación en almacenamiento estable, a partir del cual es posible restaurar el estado de la RU.[4]

Recuperación de una RU:

recuperar el estado de la RU
    (elegir nuevo líder u obtener el PdR de almac. estable)
recuperar AG a partir de P_AG
para (todas las otras RUs del sistema)
    pedirle su AG
    añadirlo al AG local
sincronizar las cohortes, si es una troupe
copiar AG en P_AG
hasta (alcanzar estado consistente con el resto del sistema)
    repetir la ejecución descrita en el AG

Ilustración A.III-5: Protocolo de recuperación.

A continuación se regenera el AG de la RU que ha fallado, solicitándoles sus copias al resto de RUs, de forma que la RU obtenga un grafo que represente el estado actual del sistema. A continuación, utilizando la información del AG recién obtenido, la RU repite su ejecución hasta alcanzar un estado consistente con el del resto del sistema (reflejado en el AG).

Durante la repetición, los eventos se reproducen en el orden indicado por el AG reconstruido. La RU solicita los mensajes ordinarios a los emisores (los emisores los tienen en su registro de mensajes volátil y el AG contiene el identificador del mensaje y el del emisor). En cambio, los mensajes emitidos por la RU no son reenviados, aunque sí se añaden al registro de mensajes, para reconstruirlo.

Por otra parte, no es necesario solicitar la información recibida del mundo exterior antes del fallo, pues está almacenada en el propio AG. De igual forma, tampoco se reenvían mensajes al mundo exterior durante la reejecución, pues los mensajes que ya habían sido enviados están anotados en el registro de compromisos.

[3] Nótese que, puesto que el líder le pasa los mensajes a la aplicación sin esperar ningún tipo de reconocimiento por parte de las cohortes, el estado de las cohortes no tiene por qué ser consistente con el del resto del sistema.

[4] Como en el caso del fallo del líder, este estado no es necesariamente consistente con el del resto del sistema, ya que la RU puede haber intercambiado mensajes con otras RUs o con el mundo exterior desde el último punto de recuperación.

Finalmente, la Tabla A.III-1 resume las necesidades de acceso a las diferentes estructuras de datos que presenta cada uno de los algoritmos expuestos. Son estas necesidades las que sugieran qué componentes lógicos es conveniente utilizar y qué actividades debe realizar cada uno, tal y como se describe en el apartado 5.3.

Protocolos	**Actividades**	AG	**P_AG**	RdM	**PdR**	**RdC**
Comunicaciones	*Enviar*	•		•		
	Atender Reenvío			•		
	Salida ME	•	•			•
	Recibir	•				
	Entrada ME	•				
Puntos de recuperación	*Establecer PdR*				•	
Recuperación	*Recuperar estado*	•	•		•	
	Recolectar AGs	•	•			
	Atender recolección	•	•			
	Repetir	•	•	•		•

Tabla A.III-1: Necesidades de acceso a las estructuras de datos desglosadas por actividades y protocolos. (AG=Grafo de Antecedentes, P_AG=AG en almacenamiento estable, RdM=Registro de Mensajes, PdR=Punto de Recuperación, RdC=Registro de compromisos)[5]

[5] P_AG, PdR y RdC aparecen en negrita para indicar que se guardan en almacenamiento estable.

Anexo IV

Implementación de las operaciones incrementales

El objetivo de este anexo es describir la implementación en el micro-núcleo de VSTa de los mensajes incrementales. Para ello, el anexo se ha estructurado en tres apartados que describen los aspectos más relevantes de la implementación. El primer apartado describe la adaptación de la interfaz descrita en el anexo II al protocolo de paso de mensajes de VSTa, el segundo explica cómo se gestionan los datos incrementales en el micro-núcleo y el tercero describe la gestión del bit de modificado.

A.IV.1 Descripción de la interfaz

VSTa presenta una interfaz para el intercambio de mensajes orientada al modelo cliente/servidor con semántica de llamada a procedimiento remoto. El servicio para enviar un mensaje, *msg_send()*, es bloqueante. El emisor (cliente) queda bloqueado hasta que el receptor (servidor) recoge el mensaje utilizando *msg_receive()* y responde mediante *msg_reply()* o *msg_error()*. El sentido del flujo de datos lo define el cliente mediante el indicador M_READ, que se incluye en el campo *m_op* del mensaje. Así, si el cliente utiliza M_READ, el mensaje pasado a *msg_send* no contiene datos, sino una

petición de los mismos. En este caso, los datos son transferidos desde el servidor al cliente cuando este último responde con *msg_reply*. Si no lo utiliza, entonces los datos son transferidos del cliente al servidor cuando éste invoca el servicio *msg_receive*.

A.IV.1.1 Correspondencia entre servicios

Para respetar en lo posible la interfaz descrita, se ha establecido una correspondencia entre los servicios *enviar* y *recibir* y los servicios *msg_send*, *msg_receive* y *msg_reply*, en función de la presencia del indicador M_READ en *msg_send*. Dicha correspondencia aparece en la Tabla A.IV-1. (No se ha considerado el caso de *msg_error* porque es un caso particular de *msg_reply*, que utiliza el servidor para notificar al cliente que su petición es incorrecta o no puede ser atendida).

	! M_READ	M_READ
msg_send	enviar()	recibir()
msg_receive	recibir()	
msg_reply		enviar()

Tabla A.IV-1: Correspondencia entre enviar/recibir *y* msg_send/msg_receive/msg_reply *en función del indicador M_READ.*

A.IV.1.2 Recepción de la cabecera

En principio, puesto que tanto el cliente como el servidor pueden ser el destino de MIs, parece necesario permitir la recepción del mapa tanto en un caso como en otro. No obstante, como se muestra a continuación, es suficiente con permitir que el servidor pueda recibir la cabecera.

Para comprobarlo, es necesario recordar los motivos por los que se introdujo la capacidad de recibir únicamente el mapa de un MI. Son fundamentalmente dos:

1. Permitir que el código de librería del receptor pueda reordenar los mensajes y proyectarlos en el orden adecuado (por ejemplo, para que todas las réplicas de proceso puedan procesar los mensajes en el mismo orden).
2. Aumentar la eficiencia y la seguridad en la gestión intermedia de MIs. En ocasiones, un proceso es un mero filtro que se encarga de esperar a que lleguen una serie de mensajes y realiza operaciones para las que no necesita acceder a los datos (como reordenar los mensajes) antes de enviárselos al proceso de destino. En estos casos se eliminan operaciones de proyección innecesarias, al mismo tiempo que se evita que el filtro pueda modificar inadvertidamente el contenido de los mensajes.

En ambos casos, el proceso debe actuar como servidor, utilizando *msg_receive*. Para un cliente, la recepción de mensajes sin proyección tiene menor interés, ya que es

él mismo el que inicia la comunicación mediante *msg_send* y, por lo tanto, puede sincronizarse con otras réplicas antes de utilizar este servicio.

A.IV.1.3 MIs negativos

En el apartado 4.1.3 y en el anexo II se ha descrito cómo transmitir MIs negativos utilizando dos mensajes: el primero para solicitar las páginas y el segundo, de respuesta, con las páginas solicitadas. En VSTa es posible aprovechar el protocolo de paso de mensajes para permitir que el cliente deseche los cambios realizados en su copia de los datos mediante una única interacción con el servidor. Para ello, el cliente utiliza el indicador NEG en *msg_send*, el servidor recibe la información sobre las páginas solicitadas mediante *msg_receive* y, finalmente devuelve dichas páginas utilizando el indicador MAP en *msg_reply*. Por lo tanto, será el proceso que desea desechar sus cambios el que inicie la comunicación, actuando como cliente.

A.IV.1.4 Interfaz para cliente/servidor

Teniendo en cuenta la correspondencia de la Tabla A.IV-1 entre las operaciones *enviar/recibir* y *msg_send/msg_receive/msg_reply*, y la discusión del apartado A.IV.1. A.IV.1 sobre la recepción de la cabecera, la interfaz para cliente/servidor se puede resumir mediante la Tabla A.IV-2. El efecto de cada uno de los indicadores es el mismo que en la interfaz para *enviar/recibir*, aunque es necesario tener en cuenta si el proceso está actuando como emisor o como receptor, ya que tanto cliente como servidor pueden desempeñar ambos papeles. En la tabla se ha utilizado texto en negrita para resaltar las entradas en las que el cliente ha utilizado el indicador M_READ y actúa, por lo tanto, como receptor de datos.

	Emisor	Receptor
msg_send	MASK, MAP, NEG, PACKED	**MASK, SRCVA, PACKED**
msg_receive		MASK, SRCVA, PACKED, HEADER
msg_reply	**MASK, MAP, NEG, PACKED**	

Tabla A.IV-2: Descripción de los identificadores que pueden utilizar el cliente y el servidor para el intercambio de MIs. Las entradas con texto en negrita indican que se ha utilizado el indicador M_READ.

A.IV.1.5 Consistencia entre tipos de mensajes

Los MIs se han implementando sobre VSTa utilizando los puertos y referencias a puertos que ya existían en el diseño original, así como las mismas funciones para gestionarlos; aunque, evidentemente, han sido necesarias algunas modificaciones. Para

permitir que se puedan enviar y recibir mensajes incrementales a través de todos los puertos del sistema, se han tomado las siguientes decisiones:

- Si los mensajes son reservados, es decir, generados por el micro-núcleo (por ejemplo, el de conexión de un cliente), son recibidos y procesados de la forma usual, independientemente de las opciones utilizadas por el servidor en *msg_receive*.
- La respuesta de un servidor al cliente debe ser del mismo tipo que su petición (los tipos en *msg_send* y *msg_reply* deben coincidir). Si no coinciden, no se transfieren datos.
- Cuando un cliente envía un mensaje con *msg_send*, se consideran las opciones expresadas por el servidor en la llamada a *msg_receive*. El efecto producido por las diferentes combinaciones posibles se describe en la Tabla A.IV-3.

M_INCR en *msg_send*	M_READ en *msg_send*	M_INCR en *msg_receive*	Efecto
sí	sí o no	sí	Se transmite sin más
sí	sí	no	Se recibe como si hubiera M_INCR en *msg_receive* (recibe el mapa si M_NEG en *msg_send* y nada en otro caso)
sí	no	no	Se recibe como M_INCR | M_PACKED en *msg_receive*
no	sí o no	sí	Se recibe como no incremental (se proyectan temporalmente los segmentos, si los hay)
no	sí o no	no	Se transmite sin más

Tabla A.IV-3: Consistencia entre tipos de mensajes.

A.IV.2 Gestión de los datos en el micro-núcleo

Uno de los aspectos más relevantes de la implementación de los mensajes incrementales que se ha llevado a cabo, es haber utilizado mensajes incrementales empaquetados para gestionar los datos en tránsito entre dos espacios de direcciones. Como se mostrará a continuación, esta estructura de datos conduce a una implementación muy eficiente de algunos casos especialmente significativos, y facilita la implementación al permitir la descomposición del problema en otros más sencillos. En particular, de esta forma es posible separar la generación del MIE a partir de los datos en el espacio de direcciones del emisor, del consumo del mensaje por el receptor.

A.IV.2.1 Descripción de la estructura de datos

La Ilustración A.IV-1 muestra los elementos básicos de esta estructura. Como puede observarse, el mensaje en el micro-núcleo consta de las secciones propuestas en el anexo II, es decir, una cabecera que contiene la información propia del mecanismo de paso de mensajes original del núcleo (en este caso VSTa) y que apunta al mensaje incremental, representado en formato empaquetado (MIE).

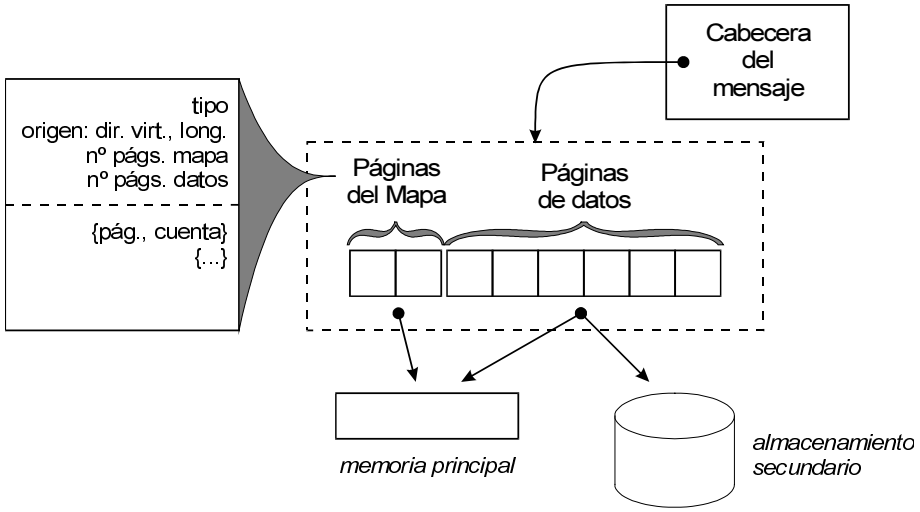

Ilustración A.IV-1: Representación de los mensajes incrementales en el micro-núcleo.

Todas las páginas del MIE, tanto las del mapa como las de datos, forman parte de la misma región de memoria virtual, tal y como se indica en la figura. Las páginas del mapa están ancladas en memoria principal, mientras que las páginas de datos pueden residir tanto en la memoria principal como en el almacenamiento secundario.

El mapa contiene los campos descritos en el anexo II (*type*, *va*, *len* y *lst*) y dos campos adicionales: el número de páginas que forman parte del mapa y el número de páginas de datos incluidas en el mensaje. La lista de páginas (campo *lst*) está formada por un vector acabado en un elemento nulo. Cada elemento del vector describe un conjunto de páginas consecutivas en la forma:

- número de la primera página del grupo —relativo a la región de origen—
- número de páginas consecutivas incluidas en el MIE (*cuenta*, en la Ilustración A.IV-1)

En la implementación, cada elemento del vector ocupa 32 bits, de los cuales se utilizan los 20 bits más significativos para indicar el número de la primera página y los 12 menos significativos para indicar la cuenta. Esta notación resulta adecuada, ya que en los procesadores x86 de Intel [Int90] el tamaño de página es de 4 KB, por lo que con 20 bits es posible representar cualquier página de la región de origen —incluso aunque ésta tenga el tamaño máximo para procesadores de 32 bits, es decir, 4 GB—.

A.IV.2.2 Generación de MIEs

Cuando el emisor envía un mensaje —mediante *msg_send*, si es el cliente, o mediante *msg_reply*, si es el servidor—, independientemente de si el receptor lo está esperando o no, el micro-núcleo utiliza la información incluida en la estructura mensaje que le proporciona el emisor para construir el MIE.

Los diferentes identificadores que puede utilizar el emisor y el efecto que producen han sido descritos en el anexo II, por lo que en este apartado se describe únicamente la técnica empleada para realizar la copia. La Tabla A.IV-4 muestra las técnicas que utiliza el micro-núcleo para generar las secciones *mapa* y *páginas* del MIE en función de los indicadores utilizados por el emisor. Asimismo, se describe la información que debe proporcionar el emisor al micro-núcleo en cada caso. A continuación se describe cada uno de los casos:

- **INCR**. El emisor indica al núcleo qué zona de su espacio de direcciones debe ser incluida en el mensaje. En la implementación actual esta zona debe estar contenida en una única región de memoria virtual.[1]

  El micro-núcleo construye el mapa a partir del registro de páginas modificadas correspondiente al destino incremental indicado por el emisor. Además, si el emisor no ha utilizado el indicador NEG, el micro-núcleo copia, utilizando la técnica de copia en escritura, las páginas que deben ser incluidas en el MI de la región de origen a la región creada para alojar el MIE (y que no forma parte todavía del espacio de direcciones de ningún proceso).

- **INCR, MAP**. El emisor indica explícitamente el conjunto de páginas que debe ser incluido en el MI. Para ello, prepara un mapa con el mismo formato que utiliza el micro-núcleo. Además, si no utiliza el indicador NEG, indica la zona de origen para las páginas descritas en el mapa (el micro-núcleo ignora los campos *va* y *len* del mapa que le proporciona el proceso).

  El micro-núcleo copia directamente el mapa proporcionado por el usuario y, si el emisor no ha utilizado el indicador NEG, copia en escritura las páginas de datos.

- **INCR, PACKED**. El emisor proporciona el MI en formato empaquetado, que debe ocupar completamente una sola región de memoria virtual.[2]

  En este caso, tanto el mapa como las páginas de datos son copiadas mediante copia en escritura.[3]

---

[1] Por zona se entiende la porción del espacio lógico de direcciones representada por una dirección virtual de origen y por el número de bytes a incluir en el mensaje. En cambio, las regiones son objetos del núcleo.

[2] Usualmente, el emisor ha recibido anteriormente un MI utilizando el indicador PACKED. Como se describe más adelante, esto provoca la proyección de una nueva región de memoria virtual sobre su espacio de direcciones, por lo que puede, posteriormente, enviar este mensaje a otro proceso, tal y como se describe en este caso.

[3] La implementación actual todavía no incluye la posibilidad de enviar mensajes sin copia (ver apartado 4.2.2). Cuando se incorpore esta opción, al enviar un mensaje empaquetado sin copia, la región

	Emisor	Micro-núcleo	
Indicadores	Información que proporciona al núcleo	*Mapa*	*Páginas*
INCR [NEG]	destino incremental datos	construye	copia en escritura (si no NEG)
INCR, MAP [NEG]	mapa datos (si no NEG)	copia	copia en escritura (si no NEG)
INCR, PACKED	MIE	copia en escritura	copia en escritura

Tabla A.IV-4: Técnicas utilizadas para la generación de los MIEs en función de los indicadores utilizados por el emisor.

A.IV.2.3 Consumo de MIEs

Cuando el receptor solicita un mensaje —mediante *msg_send*, si es el cliente y utiliza M_READ, o mediante *msg_receive*, si es el servidor—, el proceso queda bloqueado hasta que el mensaje esté disponible. Para entonces, el micro-núcleo habrá generado un MIE formado por el mapa y las páginas. No obstante, la sección *páginas* estará vacía si el emisor ha utilizado el indicador NEG, tal y como se acaba de describir. La información que obtiene el receptor depende, por una parte, de la información contenida en el MIE y, por otra, de los indicadores que haya utilizado el receptor, tal y como se muestra en la Tabla A.IV-5. A continuación se describe la información que recibe en cada caso:

- **INCR**. El receptor debe indicar la zona de su espacio de direcciones sobre la que debe llevarse a cabo la actualización contenida en el MIE. Si utiliza el indicador SRCVA, entonces se utilizan la dirección virtual y la longitud incluidas en el mapa del MIE. En otro caso, el receptor proporciona él mismo estos valores. Como ocurre para enviar un MI, la zona a actualizar debe estar incluida completamente en una región de memoria virtual.
- **INCR, NEG (en el emisor)**. En este caso, la sección *páginas* del MIE está vacía y el receptor obtiene el mapa, que es proyectado sobre una zona vacía de su espacio de direcciones.
- **INCR, PACKED**. La región que contiene el MIE es proyectada directamente sobre una zona vacía del espacio de direcciones del receptor.

será utilizada directamente como MIE en el micro-núcleo tras eliminarla del espacio de direcciones del emisor, sin necesidad de ningún tipo de copia.

Micro-núcleo	Receptor	
Información contenida en el MIE	Indicadores que utiliza	Información que recibe
Mapa,	INCR	actualización
Datos (si no NEG en emisor)	INCR, NEG (en emisor)	Mapa (nueva región)
	INCR, PACKED	MIE (nueva región)

Tabla A.IV-5: Información obtenida por el receptor en función del contenido del MIE y de los indicadores que utiliza.

Es importante destacar que la recepción del mensaje no requiere copia de datos (tampoco copia en escritura) en ningún caso. Cuando se produce una actualización, se copian las referencias que contienen las páginas del MIE a la región correspondiente del espacio de direcciones del receptor, y, a continuación, se libera la memoria ocupada por las estructuras de datos asociadas a la región que contenía el MIE. En los otros dos casos, la región que contiene el MIE no es destruida, sino que es proyectada sobre el espacio de direcciones del receptor.

A.IV.2.4 Gestión de la cola auxiliar

La implementación de la cola auxiliar de los puertos y de las operaciones para su gestión ha sido considerada desde el principio, cuando se optó por utilizar MIEs en el núcleo. A continuación se discute, para cada operación relacionada con la cola auxiliar, cómo se ha previsto implementarla.

- Cuando el servidor utiliza el indicador HEADER en la llamada a *msg_receive*, es necesario transferir el mensaje a la cola auxiliar y proporcionarle al proceso acceso al mapa del mensaje, para que pueda comprobar su contenido antes de que sea proyectado. (Ver apartado 4.1.4). El MIE puede ser transferido a la cola auxiliar directamente, sin más que añadir a la cola un nuevo elemento que apunte a la región de memoria virtual que lo contiene. Para proporcionarle el mapa al proceso, simplemente es necesario proyectarlo sobre una zona vacía de su espacio de direcciones.
- La proyección del mensaje sobre el espacio de direcciones del servidor puede realizarse de la misma forma que se ha descrito para la recepción de mensajes. (Ver Tabla A.IV-5).
- La eliminación supone liberar los recursos asignados a la región de memoria que contiene MIE, eliminado también la proyección del mapa sobre el espacio de direcciones del servidor, si todavía existe.
- El reenvío, al igual que ocurre con la recepción de la cabecera, no requiere copiar los datos contenidos en el MIE, sino que es suficiente con extraer el MIE de la cola auxiliar y depositarlo en la cola del puerto de destino.

A.IV.2.5 Discusión

La decisión de utilizar una representación intermedia en forma de MIE mientras el mensaje está en tránsito por el micro-núcleo ha permitido implementar las tres operaciones de paso de mensajes (*msg_send*, *msg_receive* y *msg_reply*) apoyándose, como se muestra en la Tabla A.IV-1, en dos funciones que implementan los algoritmos *enviar* y *recibir* descritos en el anexo II.

La utilización del MIE intermedio en el núcleo es aconsejable, desde el punto de vista del rendimiento del sistema, por varias razones:

- No se realizan copias adicionales de los datos, ya que la copia en escritura se realiza una sola vez, al generar el MIE. Como se ha descrito, la recepción no supone copia en escritura en ninguno de los casos y, cuando se permita enviar mensajes sin copia, será posible transferir un MIE sin que se realice ninguna copia, sino sólo reproyección.
- La construcción del mapa del MIE (ver Tabla A.IV-4) sólo es necesaria cuando el emisor no utiliza ni el indicador MAP ni el indicador PACKED. Además, si el receptor utiliza las opción PACKED o si el emisor ha utilizado la opción NEG, el receptor obtiene el mapa, por lo que hubiera sido necesario generarlo de todas formas en el momento de la recepción.
- La recepción del MI cuando el receptor utiliza la opción PACKED requiere únicamente la proyección de la región de memoria virtual que contiene el MIE en el núcleo, por lo que resulta muy eficiente. Nótese que este caso es especialmente significativo, ya que el envío de MIs entre nodos, tal y como se describe en el apartado 4.3.4, requiere la recepción del mensaje empaquetado por parte del servidor de comunicación. Considérese, además, que los mensajes incrementales han sido diseñados, principalmente, para facilitar la implementación de técnicas de tolerancia a fallos, y que, por tanto, cabe esperar que un porcentaje significativo de los MIs tendrán como destino procesos residentes en otros nodos.

A.IV.3 Gestión del bit de modificado

A.IV.3.1 Descripción del problema

En el apartado 4.1.5 se ha justificado la necesidad de mantener varios registros de páginas modificadas independientes, uno para cada posible destino incremental, y en el apartado 5.1 se muestra un ejemplo de aplicación que utiliza esta capacidad para mantener varias copias de los datos que actualiza mediante mensajes incrementales.

Además, es necesario compatibilizar la gestión de estos registros con la gestión del bit de modificado que lleva a cabo el módulo de memoria virtual en los núcleos que ofrecen esta facilidad.

Consideremos, a modo de ejemplo, el siguiente caso:

- Un proceso mantiene dos copias (C_0 y C_1) de un conjunto de datos (D) y en el instante t_0 realiza una copia incremental de D a C_0 y otra de D a C_1. (Para simplificar, se supone que entre ambas copias no se modifica ninguna página de D).
- En algún instante posterior, el proceso modifica la página 1 de D (D^1). Por lo tanto, D^1 debe ser incluida en posteriores copias de D a C_0 y C_1.
- En el instante t_1 se copia D a C_0. Ahora D^1 ya no debe incluirse en copias posteriores a C_0 (a menos que vuelva a ser modificada); pero todavía debe ser incluida en copias de D a C_1.

A.IV.3.2 Descripción de la solución

Para poder discernir si una página debe ser incluida en la siguiente operación incremental, el núcleo mantiene un vector de bits V en el que cada bit representa un destino potencial para operaciones incrementales. (El bit V_i representa al destino i). Para indicar si el contenido de una página virtual residente es diferente del contenido almacenado en el dispositivo de intercambio, el núcleo mantiene un bit M'.

Cuando una página es el **origen de una operación incremental**, el núcleo:

1. Comprueba el valor del bit de modificado M y, si es 1, pone a 1 todos los bits de V y el bit M'. El bit M es puesto a 0.
2. Comprueba el bit correspondiente al destino i (V_i) y, si es 1, incluye la página en la operación. Tras la operación incremental, el núcleo pone a 0 el bit V_i.

Cuando una página es el **destino de una operación incremental**, el núcleo pone a 1 todos los bits de V y a cero el bit M, ya que el contenido de esta página será diferente tras la operación y, por lo tanto, deberá ser incluida en operaciones incrementales posteriores. El bit M' no debe ser puesto a 1 necesariamente, sino que debe ser actualizado teniendo en cuenta que debe valer 0 si el contenido de la página se corresponde con su imagen en el dispositivo de intercambio y 1 en otro caso.[4]

Para permitir al usuario que gestione los destinos, tal y como se describe en el capítulo 4, en todas las llamadas al núcleo relacionadas con operaciones incrementales se incluye un argumento que el núcleo utiliza directamente como identificador i del bit V_i.

Cuando el núcleo desea **liberar el marco de página** asignado a una página virtual residente:

1. Comprueba el bit M. Si M es 1 pone a 1 todos los bits de V y el bit M'. Entonces pone M a 0.
2. Comprueba M'. Si M' es 1 se escribe la página. Entonces pone M' a 0.

[4] Una discusión más detallada requeriría profundizar en la implementación realizada; pero se ha considerado más interesante hacer una descripción independiente de la implementación.

A.IV.3.3 Comprobación

Veamos que el algoritmo anterior es correcto:

- Una página debe incluirse en una operación incremental si ha sido modificada desde la última operación incremental.

  Se cumple porque inmediatamente antes de realizar la operación incremental se comprueba el bit M y porque siempre que se pone a 0 el bit M se anota que estaba a 1 en todos los bits de V (tanto durante el robo de páginas como durante las operaciones incrementales).

- Una página no debe incluirse en una operación incremental a menos que haya sido modificada desde la última operación incremental.

  También se cumple, porque el bit V_i se pone a cero como resultado de una operación incremental con destino i y no se pone a 1 a menos que se encuentre el bit M a 1 durante una comprobación o sea modificada la página por ser el destino de una operación incremental.

El vector V se inicializa con todos los bits a 1 en el momento de la creación de la página. De esta forma todas las páginas se incluyen en la primera operación incremental y no es necesario modificar los algoritmos para que manejen correctamente la primera operación.

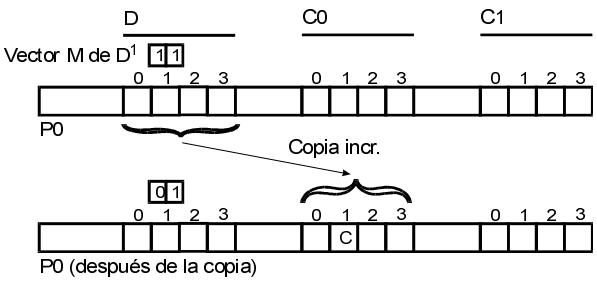

Ilustración A.IV-2: Copia virtual con dos destinos independientes.

En la Ilustración A.IV-2 se muestra el funcionamiento del algoritmo en el problema planteado en el apartado anterior. En la parte superior se muestra el estado de V asociado a la página D^1 tras el paso 1 del algoritmo. Puesto que la página había sido modificada después de la última operación incremental todos los bits de V están a 1. En la parte inferior se muestra cómo, tras la copia, el primer elemento del vector ha pasado a 0, lo que indica que la aplicación considera que este bit representa al destino C_0.

Anexo V

Estudio complementario de las prestaciones de las operaciones incrementales

En este anexo se aportan datos complementarios sobre el estudio básico de las prestaciones de los mensajes incrementales que se describe en el apartado 6.1.1, obtenidos utilizando la misma plataforma y la misma estructura de procesos que allí se describe.

A.V.1 Medidas sin robo de páginas

Estudio de la variación de número de páginas incluidas en mensajes incrementales sucesivos en función del tamaño de la región de memoria virtual

En la Ilustración 6-1 se puede observar, en el caso aleatorio, una variación significativa del coste de los mensajes incrementales en función del tamaño de la región de memoria virtual. Una de las causas de esta variación es el número de páginas incluidas

en el mensaje, que depende, como se muestra a continuación, del tamaño de la región de memoria virtual.

La Ilustración A.V-1 muestra el valor medio del número de páginas incluidas en cada mensaje incremental para el caso aleatorio. Como puede observarse, a partir de 50 páginas de tamaño, la media es superior a 9, con un crecimiento asintótico hacia 10 páginas de media. En cambio, para tamaños menores de 50 se produce una disminución significativa hasta llegar a las 6,32 páginas de media para regiones de 10 páginas.

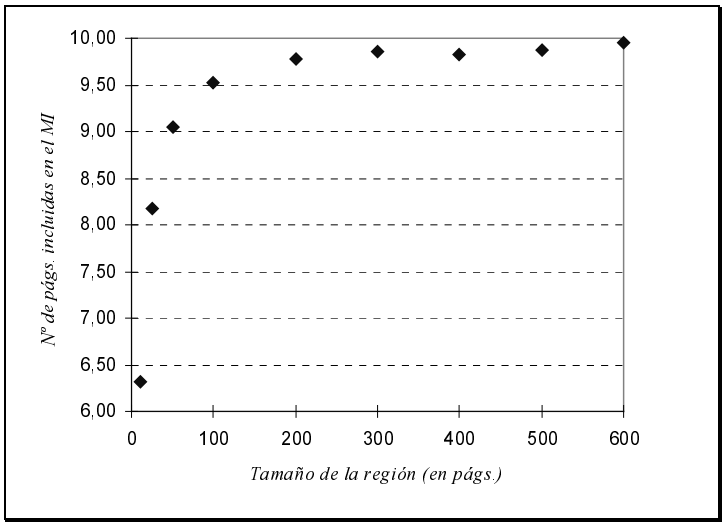

Ilustración A.V-1: Valor medio del número de páginas incluidas en cada mensaje incremental en función del tamaño de la región de memoria virtual en el caso aleatorio.

Estudio del coste de los mensajes incrementales negativos

La Ilustración A.V-2 recoge el coste medio de los mensajes incrementales negativos en un experimento análogo al descrito en el apartado 6.1.1 para mensajes incrementales. En este caso, la página de destino siempre está llena, ya que el proceso principal solicita al proceso de respaldo precisamente las páginas que acaba de modificar. No obstante, se puede observar que el coste del mensaje incremental negativo (en torno a 45 ms.) es menor que el del mensaje incremental (90 ms. para el secuencial, entre 45 y 80 ms. para el aleatorio), en contra de lo que cabría esperar, puesto que un mensaje incremental negativo requiere el mensaje de petición y el de respuesta.

Existen dos motivos para que se produzcan estos resultados. En primer lugar, el paso de mensajes en VSTa está optimizado para la comunicación entre cliente y servidor, por lo que el cliente queda bloqueado al enviar el mensaje hasta que el servidor responde. La implementación de los mensajes incrementales aprovecha este hecho, permitiendo que el mensaje incremental negativo requiera una única interacción entre cliente y servidor (igual que un mensaje incremental). No obstante, el mapa con

las páginas modificadas debe ser generado por el micro-núcleo y proyectado sobre el servidor, y posteriormente, cuando responde el servidor, debe ser copiado e interpretado de nuevo por el micro-núcleo, por lo que el coste del mensaje incremental negativo debería ser superior al del mensaje incremental.

El segundo motivo se debe, precisamente, a la forma en que se ha realizado el experimento: el proceso principal realiza una serie de modificaciones e, inmediatamente, las desecha mediante un mensaje incremental negativo. Así, las páginas de la región de respaldo permanecen continuamente vacías, y por lo tanto no es necesario (como se explica en el apartado 6.1.1) utilizar copia en escritura cuando responde el servidor. Este ahorro es importante, ya que la copia en escritura requiere, entre otras operaciones, la creación de una página de sombra[1] utilizando almacenamiento dinámico en el núcleo.

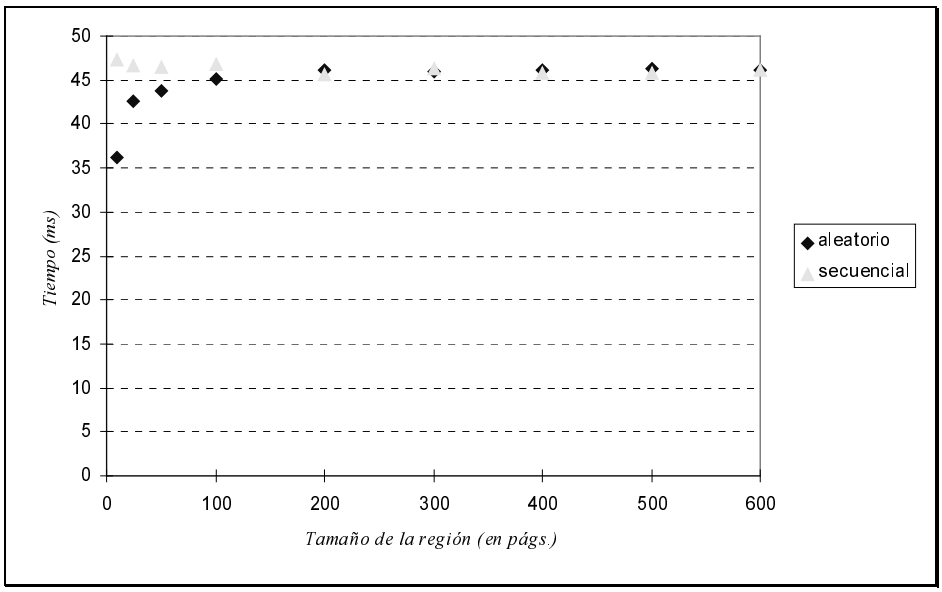

Ilustración A.V-2: Variación del coste de los mensajes incrementales negativos en función del tamaño de la región de memoria virtual, con 10 páginas modificadas.

Al igual que ocurre con el mensaje incremental, y por el mismo motivo, el número de páginas incluidas en el mensaje incremental negativo en el caso aleatorio es significativamente menor cuando el tamaño de la región es inferior a 50 páginas, y de ahí que el coste de los mensajes incrementales negativos sea significativamente menor para regiones con menos de 50 páginas.

[1] Para implementar la copia en escritura en VSTa se han utilizado páginas de sombra, que están basadas en el concepto de los objetos de sombra (*shadow objects*) utilizados en el sistema operativo Mach. [Vah96]

A.V.2 Medidas con robo de páginas

Estudio de la variación del coste de mensajes incrementales consecutivos

Tal y como se recoge en el apartado 6.1.1, una de las conclusiones más relevantes que se obtienen del análisis del coste de las operaciones incrementales es la gran importancia que tienen los problemas de sincronización de estas operaciones con el mecanismo de robo de páginas. Aunque el análisis del coste de las operaciones en función del tamaño de la región y del número de páginas modificadas también lo muestra, en la Ilustración A.V-3 se recoge la evolución del coste de los sucesivos mensajes incrementales que posteriormente se han utilizado para hacer la media en los análisis que se recogen en dicho apartado. La ilustración muestra el coste de cada uno de estos mensajes para una región de memoria virtual de 5000 páginas en tres casos diferentes, con 100, 500, y 1000 páginas al azar modificadas entre cada dos mensajes incrementales sucesivos.

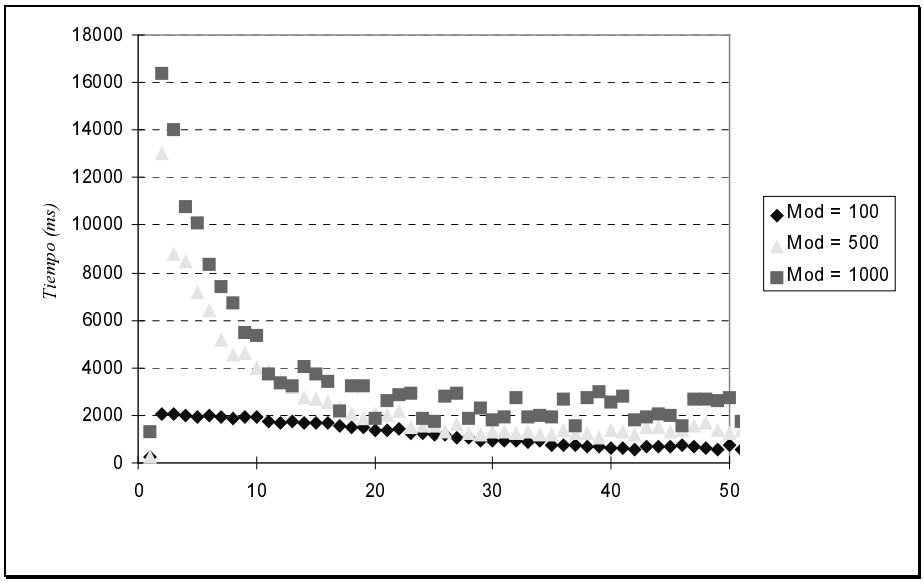

Ilustración A.V-3: Coste de mensajes incrementales sucesivos para una región de memoria virtual de 5000 páginas. (Todos los mensajes son enviados tras haber modificado aleatoriamente el mismo número de páginas: 100, 500 o 1000).

Con 100 páginas modificadas entre mensajes sucesivos el comportamiento es el que cabría esperar, con un coste similar en el envío de los diferentes mensajes de la serie. En cambio, con 500 y 1000 páginas modificadas, se produce un pico en el coste de la operación que decrece rápidamente hasta llegar a un valor estacionario a partir del mensaje 15, aproximadamente. Nótese lo significativa que es la variación, que supone una diferencia de prácticamente un orden de magnitud entre el peor caso y el valor estacionario.

Este comportamiento se debe a la forma en que se han recogido las medidas. Durante las pruebas, en el sistema existían únicamente dos regiones de memoria virtual de las que pueden robadas páginas cuando queda poca memoria principal libre, la del proceso principal y la del proceso de respaldo.[2] Inicialmente ambas están vacías y, hasta que se intercambian los primeros mensajes incrementales, la mayor parte de las páginas residentes en memoria están en la copia del proceso principal. Puesto que la rutina[3] de robo de páginas recorre la lista de marcos de página del sistema, la probabilidad de que esté accediendo a una página de la región del proceso principal es sumamente elevada. Por lo tanto, el hilo que está ejecutando la operación incremental, que tiene que acceder a todas las páginas de la región de origen (para determinar si han sido modificadas), encontrará bloqueadas las estructuras de datos asociadas a las páginas. Además, si la página está siendo volcada a disco, el tiempo de espera será del orden del tiempo de acceso al mismo. No obstante, tras las primeras interacciones, ambas regiones contendrán un número similar de páginas válidas que serán objeto de inspección por la rutina de robo de páginas, disminuyendo la frecuencia de las esperas por sincronización.

Este ejemplo muestra la importancia que tiene el diseño del sistema virtual y su interacción con la implementación de las operaciones incrementales. Nótese que en la implementación actual la operación incremental debe esperar a que se libere el acceso a la página tras finalizar la operación de lectura o escritura en disco. No obstante, sería posible modificar el diseño del mecanismo de sincronización del núcleo para el acceso a las páginas de forma que ambas actividades se realizasen de forma concurrente, tal y como se sugiere en el apartado 6.1.3.

Estudio de la variación del coste de los mensajes incrementales en función del número de páginas modificadas

En la Ilustración A.V-4 se halla representado el coste temporal de los mensajes incrementales en función del número de páginas modificadas para diferentes tamaños de la región de memoria virtual. Como puede observarse, existe un fuerte incremento del coste de los mensajes incrementales a medida que aumenta el número de páginas modificadas. La mayor parte de este incremento se debe a los bloqueos por sincronización que se han discutido anteriormente. Nótese que el coste más elevado para un número de páginas determinado corresponde a la región más pequeña (de 500 páginas). No obstante, a partir de 3000 páginas la tendencia se invierte, tal y como se describe en el apartado 6.1.1.

[2] Tal y como se explica en el apartado 6.1, en la implementación realizada sobre VSTa la rutina de robo de páginas únicamente selecciona para ser reemplazadas páginas de las regiones de memoria virtual del tipo que permite las operaciones incrementales. La razón es que la rutina original de robo de páginas en la distribución de VSTa que se ha utilizado para la implementación (versión 1.5.2) contiene algunos fallos que provocan el pánico del núcleo.

[3] En VSTa, el robo de páginas corre a cargo de un proceso de usuario, que utiliza una llamada especial del sistema que realiza un barrido continuo de las estructuras de memoria virtual para liberar marcos de página, utilizando una variante del conocido algoritmo del reloj.

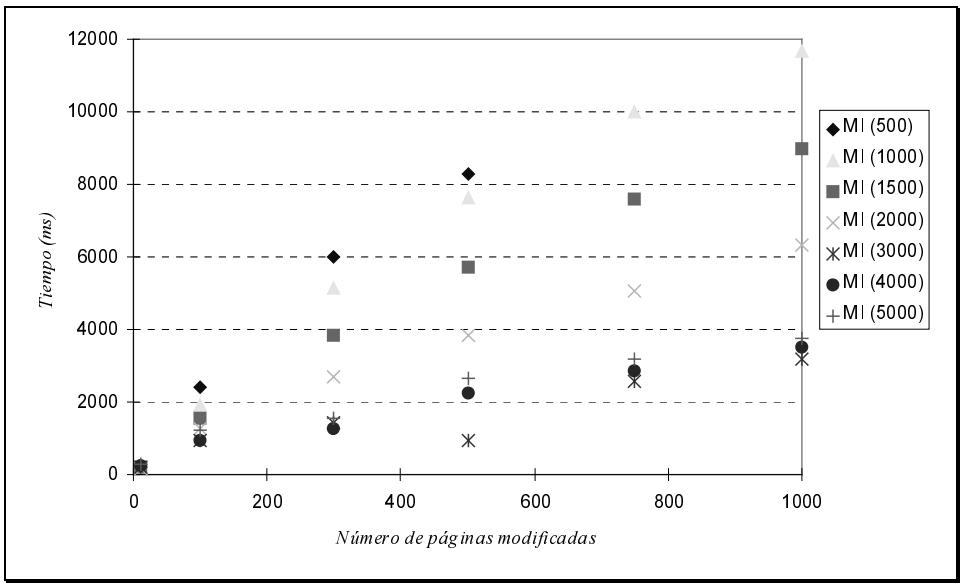

Ilustración A.V-4: Coste temporal de los mensajes incrementales en función del número de páginas modificadas para regiones de 500, 1000, 1500, 2000, 3000, 4000 y 5000 páginas.