UNIVERSIDAD POLITECNICA DE MADRID
FACULTAD DE INFORMATICA
DPTO. ARQUITECTURA Y TECNOLOGIA DE SISTEMAS INFORMATICOS

TESIS DOCTORAL

Off — Un Nuevo Enfoque en la Construcción de Sistemas Operativos Distribuidos

Autor: Francisco J. Ballesteros Cámara
Director: Sergio Arévalo Viñuales
Fecha: 20 de Febrero de 1998

Tribunal nombrado por el Mgfco. y Exmo. Sr. Rector de la Universidad Politécnica de Madrid.

Presidente

Vocal

Vocal

Vocal

Secretario

Realizado el acto de defensa y lectura de Tesis el día 20 de Febrero de 1998 en Madrid.

Calificación

El presidente   Los vocales

El secretario

A Esther,
mi mujer y mejor amiga.

Prefacio

El trabajo aquí expuesto ha sido desarrollado por el autor en el Grupo de Sistemas y Comunicaciones de la Universidad de Carlos III de Madrid.

El presente documento es el correspondiente a la memoria entregada para su defensa y la obtención del grado de doctor en informática.

Esta versión del manuscrito fue editada con *GNU Emacs* y tipografiada con LaTeX en un sistema *GNU/Linux* en Madrid el 9 de febrero de 1998.

Agradecimientos

Quiero agradecer a *Sergio Arévalo* que aceptase ser mi director de tesis, tarea que consumió no pocas de sus mañanas con discusiones y revisiones del presente manuscrito. Sergio es una de las pocas personas de este país que comprendieron mi trabajo y contribuyeron con ideas al mismo.

Agradezco a los miembros del Grupo de Sistemas y Comunicaciones (GSyC) el ambiente de trabajo que me han ofrecido y los comentarios sobre mi trabajo. Por orden alfabético: *José Centeno, Jesús González Barahona, Pedro de las Heras Quirós, Vicente Matellán Olivera y Juan Jesús Muñoz Esteban*. Han sido también de gran utilidad las reuniones mantenidas en el seno del Grupo de Sistemas Operativos Distribuidos (GSOD) de la UPM.

Dada la escasez de recursos que se dedican a tareas de investigación en mi país y dado el escaso número de personas que trabajan en el campo de los SSOO, el intercambio de opiniones con otras personas fue un recurso extremadamente valioso.

Así mismo, *Fabio Kon*, del System Software Research Group de la Universidad de Illinois, tuvo la paciencia de leer extensas partes del manuscrito en un idioma que no era el suyo.

También estoy en deuda con otras personas que no han estado directamente relacionadas con el presente trabajo. Con *Eric Jul* por pedirme el título de la tesis y sugerirme que imprimiese la primera hoja de la misma y con *Richard M. Stallman* por los comentarios que me hizo cuando visitó mi grupo.

Lo que uno es, es producto de su historia. Por tanto, me considero en deuda con los becarios con que compartí el laboratorio de lenguajes declarativos del departamento de LSIIS de la FIM de la UPM y también con Manuel V. Hermenegildo y demás miembros del laboratorio CLIP de la FIM de la UPM.

Muy especialmente, agradezco a Esther, mi compañera de viaje, la vida que me da día a día y que yo empleo en tareas de poca importancia como esta tesis.

Por último, debo agradecer a quienes se apropiaron indebidamente de algunas ideas y desarrollos originados en el transcurso del presente trabajo, que sus acciones me hiciesen pensar que tal vez, después de todo, el trabajo que sigue merecía la pena.

Resumen

La extensión alcanzada por las redes de ordenadores ha provocado la aparición y extensión de Sistemas Operativos Distribuidos (SSOODD) [158], como alternativa de futuro a los Sistemas Operativos centralizados.

El enfoque más común para la construcción de Sistemas Operativos Distribuidos [158] consiste en la realización de servicios *distribuidos* sobre un μkernel *centralizado* encargado de operar en cada uno de los nodos de la red (μkernel + servicios distribuidos). Esta estrategia entorpece la distribución del sistema y conduce a sistemas no adaptables debido a que los μkernels empleados suministran abstracciones alejadas del hardware.

Por otro lado, los Sistemas Operativos Distribuidos todavía no han alcanzando niveles de transparencia, flexibilidad, adaptabilidad y aprovechamiento de recursos comparables a los existentes en Sistemas Centralizados [160]. Conviene pues explorar otras alternativas, en la construcción de dichos sistemas, que permitan paliar esta situación.

Esta tesis propone un enfoque radicalmente diferente en la construcción de Sistemas Operativos Distribuidos: distribuir el sistema justo desde el nivel inferior haciendo uso de un μkernel distribuido soportando abstracciones próximas al hardware. Nuestro enfoque podría resumirse con la frase

> *Construyamos Sistemas Operativos basados en un μkernel distribuido en lugar de construir Sistemas Operativos Distribuidos basados en un μkernel.*

Afirmamos que con el enfoque propuesto (μkernel distribuido adaptable + servicios) se podrían conseguir importantes ventajas [148] con respecto al enfoque habitual (μkernel + servicios distribuidos): más transparencia, mejor aprovechamiento de los recursos, sistemas más flexibles y mayores cotas de adaptabilidad; sistemas más eficientes, fiables y escalables.

Palabras Clave: Sistemas Operativos, Microkernel, Adaptabilidad, Sistemas Distribuidos.

Índice General

1 Introducción 1
1.1 El Problema 2
1.2 Las Consecuencias 4
1.2.1 Poca flexibilidad 5
1.2.2 Aumento de la complejidad 5
1.2.3 Pérdida de rendimiento 6
1.2.4 Deterioro de la fiabilidad 6
1.2.5 Disminución del ámbito de aplicabilidad 7
1.2.6 Empeoramiento de la mantenibilidad 7
1.2.7 Pérdida de escalabilidad 7
1.2.8 Dificultades en la distribución 7
1.3 La Solución 8
1.3.1 Shuttles: contextos hardware extensibles 9
1.3.2 Portales: interrupciones distribuidas 9
1.3.3 DTLBs: TLBs software distribuidas 10
1.3.4 ¿Otros servicios? 11
1.4 Beneficios obtenidos 11
1.4.1 Alta flexibilidad y adaptabilidad 11
1.4.2 Simplicidad 11
1.4.3 Aumento del rendimiento 11
1.4.4 Mejora de la fiabilidad 12
1.4.5 Mejora en la distribución 12
1.5 Metodología y plan de trabajo 12
1.6 Nuevos problemas y preguntas 13
1.6.1 Ineficiencia por aumento de llamadas al sistema 13
1.6.2 Obstrucción a la gestión global de recursos 14
1.6.3 Complejidad de uso debida al bajo nivel 14
1.6.4 Distribución del μkernel 14
1.6.5 ¿Eliminación del SO? 14
1.7 Organización del manuscrito 14

2 μkernels Distribuidos Adaptables 17
2.1 Consideraciones generales 18
2.1.1 Directrices generales 20
2.1.2 Nombrado 21
2.1.3 Protección 25
2.1.4 Asignación de recursos 28
2.1.5 Revocación de recursos 28
2.1.6 Distribución de recursos 31

2.1.7 Localización de recursos . . . 32
2.2 Las abstracciones del sistema . . . 34
2.3 Manejo de dispositivos . . . 37
2.4 Migración y persistencia . . . 38
2.4.1 Autenticación y seguridad de objetos móviles . . . 39
2.4.2 Heterogeneidad . . . 40
2.5 Máquinas virtuales anidadas . . . 40

3 La realización de *Off* **43**
3.1 La arquitectura del μkernel . . . 43
3.1.1 Operación del sistema . . . 44
3.1.2 Dominios de protección . . . 45
3.1.3 Almacenamiento de shuttles, portales y DTLBs . . . 46
3.2 Recursos físicos . . . 47
3.2.1 Recursos físicos en otros sistemas . . . 48
3.2.2 El gestor de memoria física . . . 49
3.3 Gestión de Procesos en Off: Shuttles . . . 51
3.3.1 Otros enfoques en gestión de procesos . . . 51
3.3.2 Los shuttles de *Off* . . . 55
3.3.3 Propiedades . . . 57
3.3.4 Implementación en el Intel x86 . . . 59
3.3.5 Planificación . . . 62
3.3.6 Quanta vs Interrupciones . . . 64
3.4 Portales: soporte para IPC en *Off* . . . 65
3.4.1 Otros enfoques en IPC y tratamiento de eventos . . . 66
3.4.2 Los portales de *Off* . . . 68
3.4.3 Implementación en el Intel x86 . . . 77
3.5 Gestión de Memoria Virtual en Off: DTLBs . . . 81
3.5.1 Otros enfoques en gestión de memoria . . . 82
3.5.2 Las DTLBs de *Off* . . . 83
3.5.3 Uso de memoria remota . . . 85
3.5.4 Implementación en el Intel x86 . . . 87

4 Otros trabajos relacionados **91**
4.1 Distribución, adaptabilidad y DAMNs . . . 91
4.2 ¿Sistemas distribuidos adaptables? . . . 92
4.3 Adaptabilidad en SSOO distribuidos . . . 93
4.3.1 Sistemas monolíticos . . . 93
4.3.2 Sistemas basados en μkernel . . . 94
4.4 Distribución en SSOO adaptables . . . 99
4.4.1 Sistemas que descargan código en el μkernel . . . 99
4.4.2 Sistemas con μkernel extensible . . . 101
4.4.3 Sistemas que adaptan código dinámicamente . . . 104
4.4.4 Sistemas que emplean reflexión . . . 105
4.5 Sistemas de soporte a la ejecución . . . 106

5 Conclusiones y Trabajo Futuro **107**
5.1 Conclusiones 107
5.1.1 El prototipo de DAMN 107
5.1.2 El modelo de SO basado en DAMN 109
5.2 Contribuciones 109
5.3 Trabajo Futuro 109
5.3.1 Un SO distribuido auto-adaptable 109
5.3.2 Sistemas de tiempo real 110
5.3.3 Ordenadores de Red 110

Capítulo 1

Introducción

"More µkernel fantasies..."

—Comentario de un revisor anónimo

El Sistema Operativo (SO en adelante) cumple unas funciones claramente definidas en su entorno computacional [159, 147, 68, 20]. Habitualmente consiste en software íntimamente relacionado con el hardware utilizado, que permite la operación y uso del sistema suministrando abstracciones *más adecuadas* para las aplicaciones y usuarios.

Los Sistemas Operativos Distribuidos desempeñan estas mismas funciones [158, 160] en entornos distribuidos. Su trabajo consiste en facilitar el acceso y la gestión de los recursos distribuidos en la red utilizada a los usuarios del mismo. Para ello es preciso que las abstracciones suministradas por el sistema mantengan la *transparencia* [144]: el sistema debe ofrecer la ilusión de que se utiliza *un* entorno determinado y no un conjunto de recursos dispersos, inconexos y heterogéneos.

Los Sistemas Operativos Distribuidos existentes en la actualidad también pretenden suministrar *abstracciones adecuadas* y alejadas del nivel de abstracción del hardware para *todas* las aplicaciones. El problema es que, por un lado, *no hay* abstracciones que sean eficientes y apropiadas para *todas* las aplicaciones [60]; por otro lado, si adoptamos como solución la eliminación de las abstracciones del SO, el hardware no ofrece niveles de transparencia adecuados para la distribución del sistema [79, 80]:

- Si el SO suministra abstracciones de alto nivel puede mantener niveles de transparencia adecuados para el uso de recursos de modo distribuido. Al mismo tiempo, dichas abstracciones siempre serán inadecuadas para muchas aplicaciones que no encajen en el modelo de distribución adoptado.

- Si el SO no ofrece abstracciones, o suministra abstracciones que reflejen el hardware, cada aplicación puede implementar las que considere más adecuadas. En este caso hemos perdido la transparencia y la distribución del sistema conlleva la reimplementación de los servicios del sistema.

En resumen, parece imposible que se puedan suministrar abstracciones adecuadas a *todas* las aplicaciones manteniendo una buena y eficiente distribución del sistema. La construcción de SSOO distribuidos sigue sin resolverse satisfactoriamente: siempre hay gran cantidad de aplicaciones para las que las abstracciones suministradas son inadecuadas y a menudo es imposible incorporar nuevos enfoques en la distribución del sistema [5, 17, 60, 86].

Queremos proponer en esta tesis un enfoque radicalmente nuevo para la construcción de Sistemas Operativos Distribuidos que permite utilizar abstracciones adecuadas para cualquier aplicación y distribuir el sistema de un modo más conveniente.

En este sentido, la aportación fundamental de esta tesis será el suministro de abstracciones que modelen el hardware disponible en la red manteniendo la transparencia desde el principio y multiplexando dicho hardware de forma segura entre las aplicaciones existentes. El empleo de abstracciones sólo donde la distribución del hardware lo requiere y no para suministrar bloques de construcción a emplear en las aplicaciones permite que, al mismo tiempo, cada aplicación pueda emplear sus propias abstracciones y aprovecharse de la distribución de recursos ofrecida por el SO.

En lo que sigue utilizaremos definiciones de "Sistema Operativo", "Aplicación", "Usuario" y "Adaptabilidad" con los siguientes matices:

Sistema Operativo El conjunto de software en un sistema informático que, sin privilegios, no se puede cambiar o reemplazar. Esto es, todo aquello con lo que los usuarios y aplicaciones tienen que vivir día a día. El núcleo, manejadores de dispositivos ejecutando dentro y/o fuera del núcleo, librerías y utilidades de uso obligado están incluidos en esta acepción de SO.

Aplicación o Usuario El humano o conjunto de software que utiliza los servicios del SO (entendido este según la definición anterior). Esto incluye tanto a todas aquellas partes del sistema que son opcionales y/o pueden reemplazarse como a los seres humanos que utilizan el sistema. Compiladores, editores, sistemas de ficheros[1] reemplazables, librerías opcionales, y usuarios sentados ante un terminal están incluidos en esta categoría.

Adaptabilidad La capacidad de un sistema de aceptar cambios, incorporar extensiones y nuevos servicios, permitir especializaciones de los ya existentes de tal modo que no se comprometa la integridad, seguridad o eficiencia del sistema en aquellas aplicaciones que no los requieran.

1.1 El Problema

La definición habitual de Sistema Operativo es "*el software que abstrae y multiplexa de forma segura los recursos físicos del sistema*" [159]. Esto no quiere decir que dichos recursos deban estar confinados en un único nodo. ¿Por qué entonces construimos Sistemas Operativos Distribuidos alrededor de núcleos que abstraen y multiplexan sólo recursos locales?

Es cierto que dichos servicios pueden luego distribuirse si utilizamos un μkernel centralizado que implemente mecanismos y abstracciones básicas, como viene siendo habitual desde la aparición de sistemas como el kernel V [35, 38]. Incluso es posible hacerlo si utilizamos un sistema monolítico [129]. Pero esto no resuelve el problema de que el sistema no está distribuido en realidad y no está multiplexando transparentemente los recursos locales y remotos.

Una desventaja fundamental de los Sistemas Operativos Distribuidos actuales es su falta de adaptabilidad debida al suministro de abstracciones alejadas del hardware. Esto provoca una gestión inadecuada e ineficiente de los recursos [99] dado que dicha gestión se efectúa siempre de acuerdo con abstracciones diseñadas para aplicaciones "típicas", considerando el comportamiento normal (en el caso medio) de las mismas [85, 117]. Dichas aplicaciones no existen en la realidad. Baste citar como ejemplo el comportamiento de alguna de estas aplicaciones "típicas" en Entrada/Salida (E/S):

- Editores, compiladores, sistemas de ventanas, procesadores de texto y correctores ortográficos tienen comportamientos muy distintos entre sí: lecturas y escrituras aleatorias, poco frecuentes

[1] Nótese que los sistemas de ficheros se consideran tradicionalmente como parte del SO. Nosotros los consideraremos en cambio como "software de aplicación", dado que nuestro enfoque permite que cada aplicación utilice el suyo propio.

y de escasa entidad; grandes lecturas y escrituras aleatorias de forma esporádica; lecturas secuenciales y escrituras aleatoria.

- Incluso considerando sólo editores:
  - El editor *Emacs* tiende a leer y escribir ficheros completos, absorbiendo gran parte de la capacidad del sistema.
  - El editor *Vi* tiende a utilizar con mesura los recursos del sistema y lee sólo lo que considera estrictamente necesario.
  - El editor *Textedit* realiza grandes cantidades de E/S para implementar su interfaz gráfico comportándose justo al contrario en el acceso a los ficheros editados.

Claramente, no existe ninguna abstracción que pueda satisfacer de forma óptima tan dispares necesidades.

Además, dado que un SO tradicional está abstrayendo y ocultando los recursos disponibles, el problema no es solucionable: las aplicaciones no pueden emplear abstracciones diseñadas "a medida" de tal modo que la gestión de *sus* recursos sea la mejor posible [152]. Para que pudiesen hacerlo deberían tener disponibles los recursos que el Sistema Operativo les oculta.

Aunque hay diversos enfoques que permiten la construcción de sistemas adaptables [59, 34, 18, 71, 117], esencialmente, todos tratan de permitir que los usuarios puedan utilizar *directamente* el hardware disponible. En sistemas distribuidos esto provocaría la pérdida de la transparencia, la duplicación de trabajo y problemas de gestión de recursos en la distribución del sistema [52]. Esta es la principal causa de que no existan Sistemas Operativos Distribuidos Adaptables (SODAs).

La *mala* gestión de recursos en los sistemas distribuidos, debida al intento de suministrar abstracciones de propósito general, se ve claramente en la distinción entre:

- Sistemas distribuidos, que pretenden alcanzar en eficacia a los sistemas centralizados. Su objetivo es tan solo permitir el uso de recursos distribuidos con la simplicidad de los sistemas centralizados y *la misma* eficiencia.

- Sistemas paralelos, que utilizan la distribución de los recursos para superar en eficiencia (mediante el empleo de paralelismo) a los sistemas centralizados.

Puede apreciarse que el énfasis de los primeros en el suministro de abstracciones de propósito general obstaculiza, la mayoría de las veces, a aquellas aplicaciones que desean extraer el paralelismo existente en la red. Las aplicaciones paralelas habitualmente utilizan abstracciones "a medida" diseñadas para el aprovechamiento del paralelismo y estas abstracciones no tienen cabida en sistemas distribuidos que prentenden, sencillamente, ofrecer el equivalente a un sistema centralizado que operase en un conjunto de nodos.

Lo que es más, el uso de núcleos centralizados para realizar Sistemas Operativos Distribuidos hace que las abstracciones suministradas (al margen de su eficiencia) no sean adecuadas para las aplicaciones distribuidas; véanse por ejemplo [105, 113, 65, 169]. Otra prueba de esta observación es el uso de enfoques similares en la construcción de aplicaciones distribuidas tanto en SSOO distribuidos basados en μkernel como en sistemas monolíticos centralizados [124, 146, 79].

Los emuladores de Sistemas Operativos que ejecutan sobre μkernels centralizados [51] —los cuales han sido diseñados supuestamente para distribuir servicios— no son una excepción y son incapaces de utilizar más de una máquina a no ser que estén programados explícitamente para tal fin. A modo de ejemplo basta citar el emulador del SO Linux desarrollado por la OSF (Open Software Foundation) para ejecutar sobre el μkernel Mach. Dicho μkernel se considera a menudo un sistema

distribuido, incorpora gestión de memoria distribuida y es capaz de operar "transparentemente" sobre una red de nodos. Sorprendentemente, el emulador de Linux es incapaz de aprovechar la distribución de recursos ofrecida por Mach. Algo debe fallar en el modelo de distribución, ya clásico, empleado por Mach: un μkernel que gestiona recursos locales sobre el que realizar servicios distribuidos.

El problema expuesto, cuya raíz está en el uso de núcleos centralizados [20] y no adaptables para la realización de un sistema distribuido, podría concretarse aún mas en estos tres puntos:

- **La realización de abstracciones distribuidas sobre abstracciones centralizadas** (habitualmente) proporcionadas por el núcleo. Esto conduce a la re-implementación parcial o total de los servicios ya suministrados por el sistema para conseguir la distribución de los mismos. Así, es típico implementar un modelo distribuido de procesos usando otro centralizado suministrado por el núcleo [19]. Si éste último estuviese distribuido dicha repetición no sería necesaria, con el consiguiente incremento en eficiencia y simplicidad. La consecuencia es que las aplicaciones que utilizan la distribución de un modo diferente al impuesto por el sistema se ven dañadas. ¿Cuantos Sistemas Operativos *Distribuidos* pueden utilizarse como Sistemas *Paralelos*?

- **Las abstracciones del núcleo son pesadas**, ocultan y simplifican demasiado el hardware y están alejadas del mismo. Esto, que *aparentemente* parece ventajoso, empeora la fiabilidad, la flexibilidad y el rendimiento del sistema tal y como se dice en [59]. El uso de otro nivel más de abstracción (para distribuir servicios) sólo empeora la situación. En muchos casos, abstracciones de poco nivel de abstracción se implementan sobre otras de más alto nivel. Los sistemas de memoria distribuía compartida (DSMs) [98] son un claro ejemplo de ello. A este fenómeno lo denominamos *inversión de abstracciones*. No se trata de que las aplicaciones no utilicen abstracciones, se trata de que el SO no debe imponer las suyas ¡sean adecuadas o no!

- **El compromiso entre adaptabilidad y transparencia en la distribución** que se introduce, de modo artificial al utilizar un núcleo centralizado. Dado que el nivel inferior (el núcleo) no mantiene la transparencia en la distribución (es centralizado) y que la transparencia de distribución debe ofrecerse en un nivel superior, si el usuario adapta el sistema (obviando con ello el nivel superior) puede perder parte o toda la transparencia.

  Lo opuesto también sucede, para mantener la transparencia en la distribución del sistema es preciso sacrificar su adaptabilidad en gran medida, dejando inflexibles aquellas partes esenciales para la distribución del mismo (imponiendo los protocolos y algoritmos empleados para distribuir los servicios). Así pues aparece un compromiso, artificialmente, entre ambos aspectos.

  Basta considerar que la transparencia y el suministro de servicios distribuidos lo suministran componentes externos al núcleo: si éstos se pueden reemplazar se puede perder la transparencia y la distribución del sistema, si no se pueden reemplazar se pierde en adaptabilidad (hay más servicios "impuestos").

1.2 Las Consecuencias

Los problemas descritos anteriormente acarrean toda una serie de consecuencias indeseables. A continuación mencionaremos las más importantes.

1.2.1 Poca flexibilidad

El alto nivel de abstracción de los servicios suministrados por el sistema hace que los usuarios no puedan utilizar las abstracciones del sistema de un modo no previsto por los diseñadores del mismo. El sistema se torna pues inflexible.

En el mejor de los casos, las abstracciones suministradas permiten que las aplicaciones emulen abstracciones de más bajo nivel para después implementar sobre ellas las suyas propias [63]. En este caso incurrimos en una inversión de abstracciones con la consiguiente pérdida de eficiencia y aumento de complejidad.

La ocultación de información por parte del sistema también lo hace más rígido. Por ejemplo, es complicada la realización de aplicaciones que requieran conocer la instrucción y la dirección involucradas en alguno de sus fallos de página [157], como ocurre con emuladores hardware, librerías de *checkpointing*, etc.

La incapacidad de los Sistemas Operativos tradicionales para incorporar cambios es una muestra de su inflexibilidad y de la necesidad de introducir adaptabilidad en el sistema. De otro modo…¿Qué sistemas han incluido activaciones del planificador [4], varios dominios de protección en un único espacio de direcciones [33] o primitivas adecuadas para una gestión flexible y eficiente de memoria virtual [5] o *first class threads* a nivel usuario o…?

Desgraciadamente, la lista de los avances en Sistemas Operativos no incorporados a los ya existentes debido a su falta de flexibilidad es demasiado larga como para incluirla aquí.

En sistemas adaptables la capacidad de adaptación hace que el sistema sea altamente flexible dado que cualquier rigidez existente puede paliarse, adaptando o extendiendo el sistema de modo que ésta quede soslayada [148].

Ninguna aplicación que no utilice las nuevas mejoras se vería afectada en caso de error y ésto, si bien es importante en sistemas centralizados, lo es más en sistemas distribuidos —donde la reticencia a cambios es mayor. ¿Cuántas veces se ha argumentado como razón para *no* utilizar un sistema distribuido el deseo de no verse afectado por cambios en nodos remotos?

En el sistema que proponemos, al igual que ocurre en [60], estos avances podrían incorporarse en las aplicaciones del sistema dado que el software involucrado ejecuta en área de usuario. La flexibilidad está garantizada.

1.2.2 Aumento de la complejidad

Otra consecuencia es el suministro de abstracciones alejadas del hardware es el aumento en complejidad del sistema. Habitualmente todo, gran parte o la parte más compleja del software necesario para implementar los servicios del sistema está incluido dentro del mismo.

El código del sistema suele ser ya de por sí complejo y además concurrente, lo que aumenta aún más esta complejidad. Introducir nuevos servicios es difícil (lo que empeora la falta de flexibilidad) y a menudo afecta a los servicios existentes con anterioridad [163, 100].

En la arquitectura propuesta, puesto que *todos* los servicios tradicionales están realizados como extensiones (salvo por la multiplexación del hardware), el sistema es extremadamente simple. Basta tener en cuenta que los servicios suministrados son extremadamente básicos y requieren pocas líneas de código. Por la misma razón, las abstracciones suministradas son más simples y fáciles de entender.

Es cierto que la complejidad se desplaza hacia las aplicaciones y el espacio de usuario. A pesar de ello el nivel de complejidad del núcleo es extremadamente menor y las aplicaciones que lo deseen pueden emplear abstracciones más simples que las suministradas por los sistemas actuales. Por otro lado, la modularidad introducida por el empleo de un μkernel de bajo nivel hace más sencilla la realización de abstracciones, tradicionalmente suministradas por el núcleo, en área de usuario.

1.2.3 Pérdida de rendimiento

Dado que, en los SSOO distribuidos actuales, las abstracciones suministradas y los mecanismos empleados en la distribución son de propósito general y puesto que todas las aplicaciones están forzadas a utilizarlos, todas pagan por lo que no necesitan [4, 117, 126]. Basta ver la necesidad de incremento de recursos a medida que salen nuevas versiones del sistema incluso cuando seguimos utilizando el ordenador para hacer lo mismo. La razón hay que buscarla en que, con cada nueva *habilidad* del sistema, perjudicamos también a quienes *no* la utilizan.

La implementación de abstracciones por parte del sistema consume recursos que podrían utilizar las aplicaciones y la consideración sólo de recursos locales por parte del núcleo hace necesaria la realización de trabajo adicional para distribuir el sistema [19, 154, 83].

Basta tomar como ejemplo a cualquier sistema de ficheros distribuido (SFD). Actualmente *todas* las aplicaciones comparten la misma implementación del SFD. No obstante, no son las mismas las necesidades de distribución (aunque sí los algoritmos empleados actualmente para soportarlas) en los siguientes casos:

- Un sistema de ficheros (SF) que contiene ejecutables del sistema tiene transferencias masivas y pocas modificaciones —salvo por eventuales actualizaciones masivas—, admite como posibilidad el uso de versiones antiguas y no precisa contemplar la pérdida de datos (éstos pueden reinstalarse fácilmente). Por otra parte los ficheros viven durante largos intervalos de tiempo.
- En un SF con datos temporales rara vez se comparten datos. Sus transferencias suelen ser pequeñas y no admite el uso de datos antiguos. Además, sus ficheros son de poca duración.
- Un SF con directorios conteniendo ficheros de usuarios no admite el uso de versiones antiguas para el dueño de los ficheros (aunque sí para los demás), tiene transferencias pequeñas y precisa de cierta tolerancia a fallos (actualmente suministrada mediante "back-ups"). Sus ficheros suelen vivir aún más que en los mencionados en el primer punto.

La imposición de mecanismos y políticas en la distribución del sistema impiden una solución optimizada para cada caso concreto y suministran una pobre solución "media".

El uso de lenguajes de programación distribuidos y de alto nivel empobrece aún más el rendimiento debido a que sus soportes de tiempo de ejecución reimplementan gran parte de las abstracciones del sistema, duplicando trabajo ya hecho [106, 9, 65, 105, 114, 145].

Por el contrario, en un μkernel distribuido adaptable, como el que proponemos, la simplicidad de las abstracciones hace que su implementación sea extremadamente eficiente —este hecho se ha comprobado ya en sistemas centralizados [60]. Como cada aplicación puede adaptarlas o extenderlas de forma óptima en función del uso que haga de las mismas, la eficiencia en los servicios tradicionales o de alto nivel puede aumentar también en gran medida [46]. Por último, la consideración de recursos físicos remotos evita la duplicación de trabajo en la distribución del sistema [12, 11].

1.2.4 Deterioro de la fiabilidad

La disminución de la fiabilidad del sistema se debe principalmente a la complejidad del mismo. En los sistemas actuales el número de errores es elevado (éste está siempre en proporción directa al tamaño del mismo) y la introducción de cambios se hace compleja y en ocasiones inviable. Ésto perjudica la flexibilidad e impide introducir nuevas mejoras, entorpeciendo la incorporación de optimizaciones que corresponden con el estado del arte [51].

El tamaño y complejidad del sistema que proponemos son muy reducidos. Por lo tanto la fiabilidad del mismo es elevada.

1.2.5 Disminución del ámbito de aplicabilidad

Como consecuencia directa de la poca flexibilidad y del mal rendimiento, el número de usos y aplicaciones para los que los SSOO distribuidos actuales no es adecuado es considerable [25, 171]. No es de extrañar que no se considere a los Sistemas Operativos Distribuidos de propósito general como aptos para aplicaciones de tiempo real o tolerancia a fallos [64, 171].

En aquellos casos en que no fuese conveniente utilizar el sistema que proponemos, bastaría con adaptarlo o extenderlo, haciendo uso de su flexibilidad, para eliminar las razones alegadas.

1.2.6 Empeoramiento de la mantenibilidad

Los Sistemas Operativos actuales se mantienen con dificultad (véase la cantidad de dinero que engrosa las arcas de empresas de soporte de software de sistemas). Las causas son la gran complejidad de los mismos y su escasa fiabilidad. Se trata de un círculo vicioso puesto que no pueden hacerse más fiables debido a su complejidad.

El aumento en simplicidad y fiabilidad del sistema propuesto hace que su mantenimiento sea menos pesado con respecto al de un sistema no adaptable.

1.2.7 Pérdida de escalabilidad

El grado de *escalabilidad*[2] del sistema, en la actualidad, disminuye ante la imposibilidad de reemplazar aquellos servicios que marquen los límites ded mismo por otros que los amplíen.

Aunque la situación resultante es incluso peor, si el sistema se hace de modo que sea escalable todas las aplicaciones están pagando el precio de la escalabilidad incluso cuando se utiliza un único nodo. Si no lo es, el precio se paga cuando el sistema se escala.

En nuestro sistema cada aplicación puede escoger sus abstracciones dependiendo del grado de escalabilidad deseado, y reemplazar la implementación de las mismas cada vez que se cambie de opinión.

1.2.8 Dificultades en la distribución

En los Sistemas Operativos Distribuidos convencionales todas las aplicaciones deben emplear el modelo de distribución impuesta. En todos ellos es difícil o imposible conseguir que aplicaciones centralizadas aprovechen la distribución y utilicen recursos remotos de modo transparente.

Lo que es más, dado que no existe un único modelo de distribución de servicios que funcione de forma óptima (incluso, de forma tolerable) en todos los casos, es extremadamente difícil la construcción de SSOO distribuidos de propósito general. No es de extrañar que los sistemas "distribuidos" más extendidos comercialmente sean SSOO centralizados operando en red.

El control que otorgan éstos a los usuarios sobre la distribución es casi completo (dado que la distribución no está incorporada en el sistema). Los SSOO distribuidos, al contrario, no permiten a sus usuarios controlar la distribución de los recursos. Baste citar como ejemplo alguna de las quejas más típicas sobre SSOO distribuidos:

- Sabemos que estamos en un sistema distribuido cuando el fallo de una máquina que no conoces hace que la tuya deje de funcionar.
- Los sistemas de memoria distribuida compartida son malvados, la caída de una sola máquina puede destrozar todo el sistema.

[2] Término comúnmente adoptado del inglés *escalability*.

Estas quejas no expresan sólo el empeoramiento de la fiabilidad en los SSOO actuales, sino la frustración que provoca la falta de control sobre qué recursos utilizamos, cuáles no estamos dispuestos a utilizar y cuáles son los que preferimos.

Si el SO permitiese a los usuarios expresar sus preferencias y ejercer su libertad de control sobre la gestión de los recursos de que disponen, muchas de estas quejas desaparecerían. Sería posible que quienes lo prefiriesen sólo dependiesen del funcionamiento de aquellos nodos en los que se confía. Quien estuviese preocupado de la fiabilidad podría además sustituir los servicios del sistema por otros con más tolerancia ante fallos.

1.3 La Solución

Consideremos los siguientes pasos por los que ha pasado la construcción de Sistemas Operativos Distribuidos:

1. Núcleos monolíticos centralizados [139, 119] y servicios centralizados.
2. Servicios distribuidos sobre núcleos monolíticos centralizados [127, 129, 138, 3]
3. Servicios distribuidos sobre μkernels centralizados [84, 52, 140]

Por otro lado, en la construcción de Sistemas Operativos Adaptables la evolución ha sido:

1. Sistemas no adaptables basados en núcleos monolíticos centralizados [139].
2. Sistemas mucho más flexibles (aunque no adaptables) basados en μkernels centralizados [2, 140]
3. Sistemas adaptables basados en μkernels extensibles [34, 17] o exokernels [59].

Esta tesis propone como siguiente paso, uniendo la evolución en cuanto a la distribución y a la capacidad de adaptación:

4 *Sistemas distribuidos adaptables basados en μkernels distribuidos adaptables*.

Pero, ¿qué características y servicios deben estar presentes en un μkernel distribuido adaptable? y ¿dónde difiere éste de un μkernel centralizado tradicional? Para responder a estas preguntas proponemos el desarrollo de un prototipo de μkernel distribuido adaptable (o DAMN[3]) llamado *Off* [15].

La (doble) contribución de esta tesis consiste en buscar respuesta a éstas preguntas (ver tabla 1.1):

- para mantener la adaptabilidad, cualquier abstracción suministrada debe ser extremadamente básica y próxima al hardware, de tal modo que el único trabajo desempeñado por el sistema sea la multiplexación segura del hardware;
- y para ayudar a la distribución del sistema dichas abstracciones deben modelar el hardware *de la red*, no el de un simple nodo.

Cada recurso físico se divide en unidades elementales (la memoria en marcos de página, el procesador en ranuras de tiempo, el hardware de traducción de direcciones en “traducciones” (o conjuntos de traducciones), el espacio de E/S en puertos, las interrupciones en líneas[4] de interrupción) y se reparte bajo demanda a las aplicaciones que lo soliciten (como en [60]). No obstante, dichas unidades

[3] Distributed Adaptable Micro-Nucleus.

[4] Esto es, el valor numérico que provoca por software dicha interrupción

Arquitectura	**¿Adaptable?**	**¿Transparente en la distribución?**	**Ejemplo**
monolítica	no	si	Plan 9
μkernel tradicional	no	si	Mach
μkernel adaptable	si	no	Spin
exokernel	si	no	Aegis
DAMN	si	si	*Off*

Tabla 1.1: Adaptabilidad y distribución en SSOO

elementales pueden pertenecer a *cualquier* nodo de la red que disponga del recurso de que se trate. Así por ejemplo, la memoria se divide en marcos de página y el sistema de traducción de direcciones no hace distinción entre marcos locales y remotos [12].

La distribución del μkernel se consigue pues mediante la distribución de las abstracciones que suministra. Veamos cuáles son éstas en el caso de *Off*.

1.3.1 Shuttles: contextos hardware extensibles

En primer lugar es preciso suministrar algún mecanismo de tal modo que puedan crearse flujos de control e identificar los recursos necesarios para la ejecución de dichos flujos. Procesos, tareas, threads y otras abstracciones similares podrían implementarse sobre este mecanismo. Básicamente se trata de multiplexar los procesadores de un modo seguro.

Los servicios de gestión de procesos son sencillos en *Off*: *no hay procesos*. La única abstracción suministrada a tal efecto es el *Shuttle*. Un Shuttle es un contexto hardware extensible. Inicialmente consiste tan sólo en un contador de programa y un puntero de pila, aunque puede extenderse posteriormente para incluir otros fragmentos de contexto, tales como: registros de propósito general, espacios de direcciones, niveles de privilegio de entrada/salida, etc.

En otros sistemas el subsistema de gestión de procesos es rígido y debe reimplementarse para aceptar modificaciones [165, 73]. En *Off* basta con extender el Shuttle con un nuevo fragmento de contexto. A estos fragmentos los llamamos *propiedades*. No todos los shuttles han de tener las mismas propiedades, de este modo tenemos la posibilidad de utilizar diferentes abstracciones de “proceso” en el mismo sistema al mismo tiempo.

El tiempo de procesador se divide en *quanta* que se asignan a Shuttles bajo demanda [60] con lo que es posible implementar diversos algoritmos de planificación [174] en área de usuario. Además, los Shuttles, como el resto de las abstracciones suministradas por *Off* no están atados a un procesador o nodo determinado. Al considerar el sistema toda la red y no nodos aislados, cualquier procesador disponible será capaz de ejecutar un Shuttle, incluso si éste ejecutaba anteriormente en un nodo distinto, es decir, los Shuttles pueden migrar.

1.3.2 Portales: interrupciones distribuidas

En un sistema distribuido es necesario tratar tanto *traps*, como interrupciones y excepciones y suministrar algún mecanismo básico para implementar un sistema de intercomunicación de procesos. Todos estos servicios están accesibles a las aplicaciones mediante el mismo mecanismo.

El mecanismo básico de tratamiento de interrupciones, traps e intercomunicación de procesos en *Off* es el *Portal*. Un Portal puede verse como una *línea de interrupción distribuida*: los portales pueden invocarse transparentemente desde cualquier nodo para desencadenar la acción de un manejador. En este sentido, pueden utilizarse también como mensajes activos [172].

El sistema permite la invocación *automática* de portales ante la ocurrencia de interrupciones o traps. De este modo es posible gestionar dichos elementos mediante portales.

Flexibilidad y adaptabilidad están garantizadas puesto que los manejadores de Portales pueden cambiarse y residir en cualquier nodo. El Portal se puede mover de un nodo a otro bien moviendo su manejador al nodo deseado, bien mediante petición explícita.

Utilizando esta habilidad, los servicios del sistema pueden delegarse y redefinirse (como ocurre en [170]) mediante cambios de manejador del portal que los sirve.

Los Portales no almacenan mensajes y no fuerzan a elegir entre mensajes síncronos, asíncronos y llamadas a procedimiento remoto (RPC). Todos estos modelos pueden implementarse sobre portales. Los mecanismos y políticas empleados para localizar e invocar portales de otros nodos a través de la red los suministra el usuario y, de este modo, cada aplicación puede utilizar los protocolos que más convengan a cada caso.

Otros sistemas de intercomunicación de procesos (IPC) como los de Kea [170], Mach [2], Spring [84] y otros muchos se pueden implementar utilizando Portales. Aunque los portales de *Off* son más "pesados" que el mecanismo suministrado por Aegis [59] para IPC, los portales son a la vez un mecanismo adaptable y transparentemente distribuido de IPC. Como ocurre con los Shuttles, los Portales mantienen tanto la adaptabilidad como la distribución del sistema.

1.3.3 DTLBs: TLBs software distribuidas

También es preciso suministrar mecanismos para utilizar los servicios de memoria proporcionados por el hardware.

Off implementa en software una tabla de traducciones de direcciones distribuidas. Dicha abstracción suministra la misma funcionalidad que suministra el hardware de traducción de direcciones. Sus usuarios pueden establecer traducciones de direcciones virtuales a direcciones físicas y el sistema multiplexa de forma segura el hardware de traducción de direcciones entre las aplicaciones involucradas. La diferencia crucial con los servicios suministrados por el hardware es que en *Off* las direcciones físicas pueden referirse no sólo a marcos de página local sino también a marcos de página de otros nodos (y ello sin imponer un único espacio de direcciones como se hace en [175, 123]).

Dado que dicha abstracción ha de ser implementable tanto en sistemas con MMUs (Memory management unit) como en sistemas que sólo poseen TLB (translation lookaside buffer), dicha tabla se comporta como una cache de traducciones. Por dicha razón recibe el nombre de TLB software distribuida (DTLB). La DTLB permite implementar sistemas de memoria virtual (distribuidos y centralizados) en área de usuario. Puede considerarse como una versión distribuida del mecanismo presentado en [61],

El modo en que se use la memoria local para hacer cache de la remota y otras políticas involucradas están definidas por las aplicaciones; *Off* no incorpora ninguna política de gestión de memoria, sólo los mecanismos básicos suministrados por el *hardware distribuido* en la red [12].

Vemos entonces que la memoria física está dividida en marcos que se asignan bajo demanda. La consideración de la memoria existente en la red, y no sólo en un nodo, permite emplear políticas de asignación y revocación locales o globales según se prefiera.

En consecuencia, pueden coexistir simultáneamente en *Off* diferentes sistemas de memoria distribuida compartida (DSMs —de *distributed shared memory*) [63] y de memoria virtual distribuida [180] sin reimplementar una sóla línea del trabajo hecho por *Off*. Estos sistemas pueden ahora adaptarse (un buen sistema sería utilizar diseños similares a [92]), puesto que ejecutan a nivel de usuario, y pueden reemplazarse cuando sean inapropiados. Incluso cuando se reemplacen, los servicios de memoria serán esencialmente distribuidos puesto que la abstracción básica (la DTLB) lo permite.

Así, una aplicación[5] centralizada puede ahora utilizar transparentemente memoria remota mediante la DTLB (como ejemplo extremo de dicha aplicación podría considerarse un gestor centralizado de memoria virtual que, gracias a la DTLB, gestionase sin saberlo memoria procedente de otros nodos) y esa transparencia no impide el uso de sus propias políticas y mecanismos.

1.3.4 ¿Otros servicios?

En principio no hay ningún otro servicio suministrado por *Off*. Naturalmente siempre queda la posibilidad de utilizar portales para incluir nuevos servicios en el sistema como pudieran ser los de acceso al reloj y temporizadores de tiempo real, entrada/salida programada, etc.

1.4 Beneficios obtenidos

Los dos beneficios primarios son la *adaptabilidad* del sistema resultante y la *distribución inherente* de los recursos que permite hacer uso de aquella sin pérdida de transparencia. A continuación expondremos brevemente los beneficios adicionales más importantes.

1.4.1 Alta flexibilidad y adaptabilidad

Al mantener cerca del hardware las abstracciones suministradas (contextos hardware, interrupciones distribuidas y TLBs software distribuidas) todo el trabajo que queda por hacer es tarea de las aplicaciones. Consiguientemente, el sistema es extremadamente flexible y adaptable: abstracciones muy diferentes pueden coexistir y utilizar los recursos disponibles en red.

Cualquier usuario puede optar por utilizar librerías disponibles que suministren servicios de más alto nivel o por emplear las suyas propias.

Estas librerías contienen ahora no sólo funciones de propósito general como en [139] o servicios de alto nivel del SO como en [69]. Todo lo que no se refiera a multiplexar de forma segura el hardware puede estar incluido en ellas y ser adaptado o cambiado. El único sistema distribuido que otorga más flexibilidad es el hardware o una emulación completa del mismo [45] —aunque esto último acarrea graves ineficiencias.

1.4.2 Simplicidad

El prototipo de μkernel distribuido adaptable, *Off*, tiene un total de 8.000 líneas de código — incluyendo comentarios[6] — y ocupa unos 30 Kilobytes una vez cargado en memoria. Basta comparar estas cifras con las de otros sistemas existentes.

Por otro lado el sistema sólo incorpora *una* llamada al sistema (la que invoca un Portal: *prtl send*) y *tres* abstracciones sencillas. Es fácil incluir modificaciones y entenderlo por completo en poco tiempo.

1.4.3 Aumento del rendimiento

La simplicidad de los servicios suministrados hace que la implementación tenga poco trabajo que hacer y, por tanto, pueda concentrarse en hacer su *único trabajo* bien: multiplexar el hardware dis-

[5]En el sentido de la definición presentada anteriormente (incluyendo por tanto a sistemas de ficheros, DSMs, DVMs, editores, compiladores, etc.)

[6]Nos referimos a los comentarios escritos en *C*, puesto que el código fuente está redactado de forma literaria [103]

tribuido. No se pierde tiempo ni demultiplexando llamadas al sistema ni en tareas de registro y mantenimiento[7] que pueden no llegar nunca a ser necesarias [117].

La adaptabilidad permite emplear algoritmos optimizados para aplicaciones específicas. Mediante técnicas similares se ha conseguido incrementar la eficiencia de diversas aplicaciones hasta en dos *ordenes de magnitud* [59]. Ahora, gracias al trabajo desarrollado en esta tesis, estas técnicas se pueden utilizar también en Sistemas Operativos Distribuidos.

1.4.4 Mejora de la fiabilidad

La extremada simplicidad del sistema hace que disminuya la posibilidad de errores y efectos laterales y aumente la robustez del sistema.

Áreas antes vedadas a sistemas de propósito general [64, 171] quedan abiertas ahora. Nada impide utilizar el sistema con las mismas garantías que ofrecería un sistema de propósito específico.

1.4.5 Mejora en la distribución

En *Off* son los recursos físicos los que se consideran distribuidos. Naturalmente es posible gestionarlos localmente, pero esto no impide que aplicaciones centralizadas utilicen sin saberlo[8] recursos remotos. Aquellas aplicaciones que lo deseen pueden emplear políticas y mecanismos propios de gestión distribuida de recursos.

Podemos tener abstracciones de naturaleza distribuida y alto nivel comparables a las suministradas por otros Sistemas Operativos Distribuidos y también podemos distribuir, *casi* sobre el hardware, sistemas de tiempo de ejecución utilizados por lenguajes de programación distribuidos sin incurrir en pérdida de eficiencia ni en duplicación de código entre el sistema operativo y el lenguaje de programación. Como puede verse en la figura adjunta, ya no es preciso utilizar *middleware* que reimplemente los servicios del sistema dado que ahora es posible utilizar abstracciones "a medida".

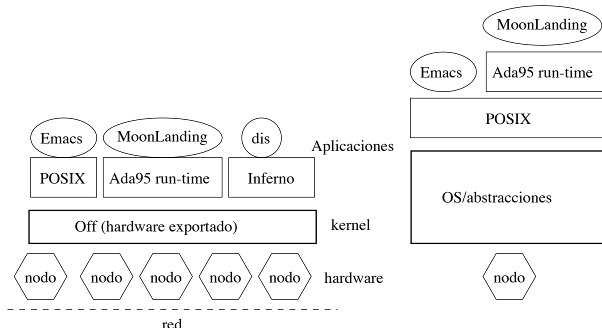

Figura 1.1: Diferentes aplicaciones utilizando las abstracciones que prefieren.

1.5 Metodología y plan de trabajo

Para la realización del prototipo propuesto basta con los siguientes elementos:

[7] esto es, *bookkeeping*.

[8] Naturalmente, sólo en el caso en que no les importe esta circunstancia. Siempre existe la posibilidad de que dichas aplicaciones controlen explícitamente el uso de recursos remotos.

- Para composición de documentos, incluyendo este y futuros manuscritos y el propio código fuente, se utilizará LaTeX, utilizando directamente TeX sólo cuando sea estrictamente necesario.

- El lenguaje de programación escogido ha sido *C*. La razón principal es la de disponibilidad de compilador en cualquier plataforma, la sencillez del soporte de tiempo de ejecución de este lenguaje y la disponibilidad de un kit de acceso al hardware de la arquitectura Intel/PC para dicho lenguaje. El compilador disponible es el "GNU *C* Compiler", GCC. Tras la conclusión de un primer prototipo, es probable que, a medida que se vayan introduciendo cambios, el código se reescriba en C++, dado que el diseño del sistema se ajusta bien a un modelo de programación orientada a objetos.

- El código de *Off* se redactará de forma literaria [103] con ayuda de *noweb*. Mediante esta herramienta se genera el código y la documentación a partir de los mismos ficheros fuente.

- Como entorno de desarrollo, y dado que *Off* tardará un tiempo en ser utilizable como tal, hemos optado por GNU/Linux.

- Para ayudar en la implementación sobre Intel utilizaremos OSKit [70], un *kit* para construcción de SSOO que posee una estructura altamente modular.

El código del prototipo implementado se dejará disponible bajo licencia GPL en [10].

La metodología empleada en la construcción del prototipo será la habitual en el caso de la construcción de SSOO: la implementación incremental de servicios del núcleo utilizando depuración remota mediante puerto serie.

Por último, el plan de trabajo consiste en la elaboración de la arquitectura basada en DAMN, el diseño de *Off* y su implementación.

1.6 Nuevos problemas y preguntas

El uso de la arquitectura propuesta también plantea diversas preguntas y problemas a los que trataremos de argumentar en este apartado.

1.6.1 Ineficiencia por aumento de llamadas al sistema

Es cierto que se producen más llamadas al sistema si el interfaz empleado es de más bajo nivel. No obstante, esto no es un problema grave y existen soluciones [89] aportadas por otros sistemas que también utilizan interfaces próximos al hardware [60].

Además hay que considerar la disminución del coste de llamada al sistema, la posibilidad de efectuar intercomunicación de procesos (IPCs —de *inter-process communication*) entre distintos dominios de protección sin cambiar de contexto (ver capítulo 3.4), el incremento de eficiencia en la manipulación de datos de usuario por parte de (lo que antes era) el sistema operativo (que ahora puede estar contenido en una librería) y la posibilidad de aumentar la eficiencia en IPC sobre la red mediante uso de protocolos específicos. Estas consideraciones contrarrestan el posible incremento de llamadas al sistema como se ha visto en sistemas centralizados adaptables [60] y en sistemas distribuidos que utilizan protocolos específicos [121, 104].

A esto hay que añadir que resultados anteriores muestran como los cambios de contexto en arquitecturas μkernel no son realmente un problema conceptual, sino que son cuestión a resolver por una implementación cuidadosamente diseñada [111].

1.6.2 Obstrucción a la gestión global de recursos

La eliminación de casi la totalidad de las políticas involucradas en la gestión de recursos origina una gestión distribuida de recursos. En el sistema propuesto el núcleo no decide qué recursos asignar y cuales liberar sino que las aplicaciones toman parte en dichas decisiones.

No obstante, como se ha visto en algunos sistemas adaptables centralizados como [60], el único cambio es la transición a un modelo de gestión distribuida de recursos.

Lo que es más, dado que la arquitectura propuesta se centra en la construcción de SSOO distribuidos, parece más adecuado optar por una gestión distribuida de recursos desde un principio.

Por otro lado, no es completamente cierto que se impida la gestión global de recursos en aquellos casos en que se desee utilizarla. El μkernel que proponemos expone aquellos datos (ocultos en otros sistemas) que pueden ser de interés para las aplicaciones siempre que no comprometan la seguridad del sistema. Así por ejemplo, en cuanto a gestión de memoria se refiere, los bits proporcionados por el hardware (marco referenciado, escrito, etc.) y la lista de marcos libres quedan expuestos de tal modo que las aplicaciones pueden leer dicha información. Sería posible entonces implementar una gestión global de memoria física (si es lo que se desea).

1.6.3 Complejidad de uso debida al bajo nivel

No es cierto que el sistema sea más complejo de utilizar. Es posible utilizar servicios tradicionales de más alto nivel implementados en librerías o servidores [94].

Por otro lado, el uso de sólo tres abstracciones básicas simplifica el uso del sistema. La complejidad añadida por el manejo de dispositivos de E/S puede eliminarse del mismo modo que nosotros hacemos con el procesador, las excepciones y la memoria; basta considerar enfoques que ya lo hacen para algunos dispositivos (por ejemplo, discos magnéticos [49]).

1.6.4 Distribución del μkernel

La pregunta podría ser: si el μkernel está distribuido ¿Es posible gestionar localmente los recursos?

Off está distribuido apoyándose en abstracciones que permiten utilizar hardware de *cualquier* nodo. Esto no quiere decir que *Off* realice gestión global de recursos. En concreto, la gestión de recursos cae por completo (salvo la multiplexación de hardware) en el nivel de aplicación. Es posible pues utilizar una gestión local de recursos, una global o una aproximación mixta.

1.6.5 ¿Eliminación del SO?

Por último, también se argumenta que el enfoque propuesto es equivalente a la *eliminación* el sistema operativo, o al uso de una librería de soporte a la ejecución que multiplexe el hardware.

Lo que proponemos en cambio es multiplexar el hardware de un modo *seguro*, considerando tanto recursos locales como remotos. Esto es muy diferente de lo realizado en sistemas donde el SO está incluido en una librería *por completo* [44], como los utilizados en ciertas aplicaciones empotradas. Dichos sistemas son incapaces de proteger a unas aplicaciones de otras.

1.7 Organización del manuscrito

En el capítulo 2 veremos en más detalle la problemática del enfoque propuesto (con sus ventajas y desventajas) y expondremos el diseño de *Off*, un prototipo implementado para corroborar las hipótesis efectuadas. El capítulo 3 detallará el diseño y la implementación de las tres abstracciones básicas suministradas por *Off*. El capítulo 4 comparará nuestra solución con las propuestas en

otros trabajos relacionados. Por último, en el capítulo 5 expondremos las conclusiones obtenidas y el trabajo a realizar en el futuro.

Capítulo 2

μkernels Distribuidos Adaptables

Mach was the greatest intellectual fraud in the last ten years.
What about X?
I said ‘intellectual´.

—;login, 9/1990

En este capítulo revisaremos las decisiones de diseño adoptadas, que han conducido a la implementación de *Off*, el prototipo de μkernel distribuido adaptable (DAMN[1], en adelante) que presentaremos en siguientes capítulos. Dichas decisiones vienen derivadas en gran medida de lo ya dicho en el apartado 1.3. En lo que sigue, expondremos conjuntamente las características que debe poseer un DAMN así como las decisiones adoptadas en el diseño e implementación del prototipo *Off*.

Utilizamos como hipótesis de trabajo que la distribución del sistema debe realizarse desde el nivel inferior (dentro del μkernel) manteniendo el interfaz suministrado tan próximo al hardware como sea posible sin comprometer la seguridad en la multiplexación de los recursos físicos distribuidos.

Como veremos en el capítulo 5, esto promete paliar en gran medida los problemas mencionados en el capítulo 1. Se trata, por un lado, de eliminar la barrera entre gestión local (en un nodo) y gestión global (en la red) de recursos, impuesta artificialmente por los μkernels centralizados que empleamos actualmente y, por otro lado, de garantizar la adaptabilidad del sistema.

En el diseño del sistema se han tratado de encontrar abstracciones básicas que modelen los recursos *físicos* disponibles de tal modo que estas abstracciones sean capaces de operar de forma transparente en distintos nodos de la red. Los servicios del sistema podrían de ese modo distribuirse en el μkernel y, considerando el bajo nivel de las abstracciones suministradas, el sistema podría mantener un adecuado grado de adaptabilidad.

Así, el objetivo de un DAMN —y el de *Off*— puede concretarse en los siguientes puntos:

- suministro de *abstracciones básicas de muy bajo nivel* que modelen el hardware disponible.
- suministro de abstracciones capaces de *operar transparentemente en los diferentes nodos* de la red.

Es aquí donde encontramos la diferencia fundamental entre el DAMN y otras arquitecturas basadas en μkernels adaptables: en un DAMN no sólo se extiende a la red la abstracción empleada

[1] Distributed Adaptable Micro-Nucleus.

para intercomunicación de procesos sino que también se distribuyen el resto de los servicios. Del mismo modo que en los μkernels actuales se emplean protocolos de localización y transporte para permitir comunicación remota, dichos protocolos se emplean también en *Off* para utilizar memoria, procesadores y dispositivos remotos.

Esta diferencia es fundamental en aplicaciones diseñadas convencionalmente para sistemas centralizados. Si tan sólo se distribuyesen los mecanismos de IPC, estas aplicaciones serían incapaces de utilizar la distribución en el resto de los servicios.

La migración y replicación de objetos en el sistema también se vería obstaculizada dado que sería preciso reproducir *manualmente* cualquier elemento que no fuese transparentemente accesible mediante IPC (esto es la tónica habitual en los sistemas actuales de migración y replicación que se ven obligados a utilizar complejos sistemas de soporte para dotar a las entidades migradas del entorno necesario). Adicionalmente, los sistemas distribuidos actuales obstaculizan aún más la migración al ocultar a las aplicaciones la parte de su estado que reside en el interior del núcleo.

Para las aplicaciones distribuidas es también importante la distinción entre un μkernel con IPC capaz de operar en red y un DAMN. En un DAMN, cada aplicación puede distribuir el sistema del modo más conveniente, utilizando sus propios modelos, algoritmos y protocolos, soslayando las desventajas que presente el modelo de distribución suministrado inicialmente por el sistema.

En lo que resta del presente capítulo discutiremos el diseño y las características de la arquitectura propuesta: un SO basado en DAMN. Así mismo, iremos introduciendo las decisiones de diseño y compromisos generales adoptados en el prototipo de DAMN implementado, *Off*.

2.1 Consideraciones generales

El diseño de un DAMN se basa en la apreciación de que las funciones realizadas por un SO distribuido[2] son:

- el suministro de abstracciones y
- la gestión distribuida de recursos, que incluye además
- la multiplexación del hardware presente en la red de un modo seguro.

Como podemos ver en la figura 2.1, los enfoques tradicionales consideran a todas estas funciones como responsabilidad del SO. En el modelo propuesto, el sistema se centra únicamente en la multiplexación del hardware. El suministro de abstracciones aparece sólo como herramienta para permitir el uso de hardware de un modo distribuido, quedando este aspecto reducido al mínimo. En este sentido seguimos lo dicho en [157], donde se emplea un nano-kernel para modelar el hardware, aunque nosotros creemos que ésta debiera ser la única parte privilegiada del sistema y no sólo consideramos un nodo.

Podemos ver esto mismo de otro modo encuadrando el sistema propuesto en relación a otros enfoques (ver también el capítulo 4). En la figura 2.2 aparecen las distintas arquitecturas utilizadas hasta el momento para la construcción de SSOO distribuidos:

a. *Arquitecturas monolíticas*. No permiten la adaptación del sistema y utilizan un único núcleo como mecanismo para alcanzar la distribución de servicios.

b. *Arquitecturas basadas en μkernel centralizado*. Permiten sólo la adaptación de servicios de alto nivel (Un ejemplo son los paginadores externos en Mach [24]). La distribución del sistema

[2]Puede apreciarse, no obstante, el paralelismo con las que realiza un SO centralizado.

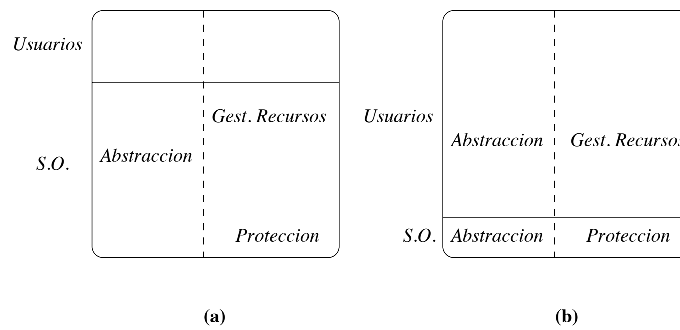

Figura 2.1: Funciones de un SO tradicional (a) y de un DAMN (b)

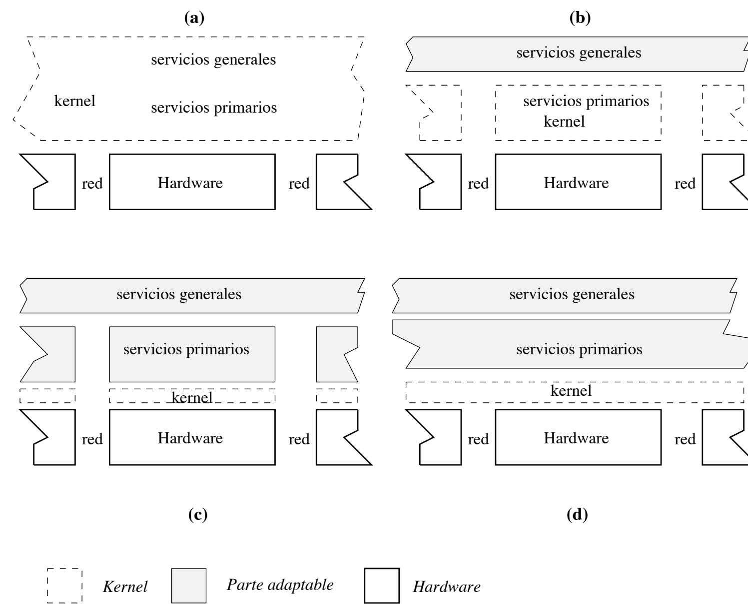

Figura 2.2: Modelos de construcción de SSOO

se efectúa, o bien dentro del μkernel (con lo que no es posible que las aplicaciones utilicen sus propios mecanismos y políticas de distribución), o bien fuera del núcleo (con lo que por un lado la realización de adaptaciones puede suponer la pérdida de la distribución y por otro, el sistema no está realmente distribuido al ser imposible el uso de servicios del μkernel desde otro nodo). La distribución suele verse reducida tan sólo a la posibilidad de utilizar el mecanismo de IPC a través de la red.

c. *Arquitecturas basadas en exokernel* [62]. Son altamente adaptables pero consideran sólo recursos locales. La distribución del sistema se realiza *sobre* el núcleo del sistema con lo que es difícil realizar adaptaciones y beneficiarse, al mismo tiempo, de la distribución del sistema. Por otro lado, las aplicaciones centralizadas tendrán dificultad en beneficiarse de la distribución del sistema. Esto no es de extrañar si tenemos en cuenta que este enfoque obvia por completo la construcción de sistemas distribuidos.

d. *Arquitecturas basadas en DAMNs*. El modelo que proponemos en esta tesis facilita la construcción de sistemas distribuidos adaptables.

2.1.1 Directrices generales

En el diseño del DAMN seguimos una serie de directrices que conviene mostrar:

1. *Los recursos deben poder utilizarse con independencia de su ubicación.*

   Esto es necesario para facilitar el uso del sistema a aquellas aplicaciones que no empleen sus propios mecanismos de distribución. Por otro lado, la ausencia de esta característica destruiría la distribución del sistema dado que es el rasgo distintivo de los sistemas distribuidos.

2. *Los recursos deben corresponder con los suministrados por el hardware.*

   Si garantizamos un bajo nivel de abstracción evitaremos la realización de trabajo que puede no ser útil e incluso estorbar a algunas de las aplicaciones. En este punto, coincidimos con el modelo de sistemas basados en Exokernel [60].

3. *Los nombres empleados por el sistema deben ser válidos en toda la red y a la vez corresponder con los empleados por el hardware.*

   Para poder utilizar recursos remotos es preciso que sus nombres tengan validez fuera de su nodo. La existencia de una correspondencia entre los mismos y los empleados por el hardware permite que las aplicaciones puedan conocer la información que estos últimos contienen: ubicación, en el caso de direcciones de red; habilidades, en el caso de las direcciones de memoria física en arquitecturas como el Intel/PC[3]; interferencias causadas por su uso, como la línea de cache utilizada en el caso de las direcciones de memoria física, etc.

4. *El estado de los recursos debe ser visible por las aplicaciones siempre que esto no comprometa la seguridad del sistema.*

   Si las aplicaciones conocen cuales son los recursos disponibles pueden emplear mejores políticas de asignación de recursos.

5. *La asignación de recursos debe realizarse en unidades tan pequeñas como sea posible sin comprometer la seguridad ni la distribución del sistema.*

[3] En ellas sólo el primer Mbyte de la memoria es capaz de recibir proyecciones de dispositivos de E/S y sólo los primeros 16 Mbytes son capaces de operar con acceso directo a memoria (DMA).

Las aplicaciones siempre pueden utilizar agregación si necesitan unidades mayores, pero lo contrario no siempre es cierto.

6. *Debe preservarse la diversidad.*

   La transparencia en la distribución ofrecida por las abstracciones de un DAMN no debe ocultar la heterogeneidad presente en la red. Esto puede parecer una contradicción, aunque no lo es. Las abstracciones deben poder *nombrarse* y *utilizarse* de forma transparente, no obstante, las diferencias derivadas de las distintas arquitecturas empleadas deben mantenerse. Siempre es posible añadir una máquina virtual o aumentar el nivel de abstracción para suministrar un entorno homogéneo a partir de uno heterogéneo (lo recíproco no siempre es cierto).

7. *Los objetos suministrados por el sistema deben soportar la migracón y la persistencia.*

   Dado que un DAMN considera la red como el hardware disponible, los objetos suministrados por el sistema deben ser capaces de migrar. De otro modo sería imposible cambiar de nodo flujos de ejecución al contrario de lo que ocurre en un multiprocesador y la barrera alrededor del nodo local obstruiría el uso de recursos remotos. La persistencia es en realidad una consecuencia de la capacidad de migración, ya que aquella puede considerarse una instancia del problema constituido por ésta (la persistencia puede implementarse como una migración a un almacenamiento "no-volátil").

8. *La gestión de recursos hecha por el sistema no debería tener en cuenta aspectos relacionados con la "escalabilidad"; aunque no debe impedirla.*

   Esto puede parecer perjudicial pero, al contrario, creemos que es beneficioso para el sistema resultante. El diseño de abstracciones escalables hace que la gestión de las mismas sea más complicada y, en muchos casos, ineficiente. Dado que en un DAMN los servicios los suministran realmente las aplicaciones, es razonable pensar que deben ser las aplicaciones que deseen escalabilidad las que la paguen. Por otro lado, si el número de nodos gestionado es demasiado alto siempre resulta conveniente para su administración partir el sistema en varios enlazados entre sí mediante pasarelas (esto es práctica común en los sistemas de comunicaciones actuales).

Para construir un DAMN basta con seguir estas directrices que conducen a un sistema que considera la distribución desde el principio y se sitúa en un bajo nivel de abstracción, manteniendo la adaptabilidad.

Veamos ahora lo que estos puntos suponen en relación a aspectos concretos tales como el nombrado, protección, asignación, revocación y distribución de recursos.

2.1.2 Nombrado

Existe división de opiniones en cuanto a cómo deben ser los espacios de nombres y los nombres en un SO.

Por un lado, hay sistemas [121, 2] que utilizan un único espacio de nombres para todos los distintos tipos de abstracciones (ej. Amoeba utiliza capabilities que se emplean para denotar unívocamente cualquier tipo de recurso) y por otro lado, tenemos sistemas [59, 139] que utilizan distintos espacios de nombres para distintas abstracciones. En sistemas de espacio de nombres único un nombre siempre representa al mismo objeto. En sistemas con múltiples espacios de nombres un nombre puede denotar objetos distintos en espacios de nombres distintos.

El primer enfoque (espacio de nombres único) conduce a sistemas de nombrado más simples de utilizar, complejos de implementar y de más alto nivel de abstracción. El segundo conduce a sistemas de nombrado más complejos de utilizar, simples de implementar y de menor nivel de abstracción.

El enfoque adoptado en un DAMN es el de *múltiples espacios de nombres* (uno por cada abstracción o recurso suministrado por el sistema) en correspondencia con la multiplicidad de espacios de nombres existente en el hardware (en una arquitectura dada el hardware hace que un *entero* pueda identificar a un marco de página, a una línea de interrupción, a un trap, etc. dependiendo del espacio de nombres empleado). La adopción de múltiples espacios de nombres facilita la *extensión* al conjunto de la red de los nombres físicos (los utilizados por el hardware) del modo más adecuado en cada caso. Seguimos pues las directrices 6 y 3 mencionadas en el apartado 2.1.1, según las cuales debe preservarse la diversidad y los nombres deben ser válidos en la red y corresponder con los empleados por el hardware.

Considerando ahora la estructura de los nombres en sí, algunos autores sostienen la conveniencia de utilizar nombres homogéneos[4], únicos, opacos, de alto nivel que mantengan la transparencia de ubicación [121, 2]. También encontramos otros autores que sostienen la utilidad y conveniencia de utilizar nombres heterogéneos y de bajo nivel como los suministrados por el hardware [59].

Los primeros son los más convenientes para la distribución del sistema dado que pueden utilizarse desde cualquier punto de la red con independencia de su posición. No obstante, también degradan la adaptabilidad y el rendimiento del sistema puesto que la información que contienen es inaccesible a las aplicaciones y habitualmente requieren de niveles intermedios de traducción hacia los correspondientes nombres físicos.

Los segundos son buenos en cuanto a adaptabilidad y rendimiento se refiere, pero no son adecuados para un sistema distribuido.

En un DAMN los nombres deberían *aproximarse* tanto como sea posible a los nombres físicos empleados por el hardware. Esto conduce al empleo de nombres heterogéneos y de relativamente bajo nivel de abstracción.

En *Off* tenemos tres tipos de recursos que hemos de nombrar:

- Nombres para abstracciones suministradas por el sistema que no corresponden directamente con recursos físicos (como Shuttles, Portales y DTLBs). Nótese que estos recursos pueden moverse libremente en la red (ej. una DTLB puede emplearse en distintos procesadores situados en distintos nodos del sistema).

- Nombres para aquellos elementos físicos (procesadores, bancos de memoria, bancos de puertos de E/S, etc.) presentes en el sistema que que son la mínima unidad reemplazable. Dichos elementos pueden reemplazarse por otros, posiblemente con distinta ubicación (por ejemplo, un banco de memoria m de un nodo p podría reemplazarse por otro banco en un nodo q). Sus nombres deben mantener la validez durante y después del reemplazamiento (en el ejemplo, el banco de memoria seguirá siendo siempre m), de tal modo que éste sea transparente para los usuarios del elemento físico

  Estos elementos pueden subdividirse en unidades elementales, constituyendo éstas las mínimas unidades asignables.

- Nombres para unidades elementales de recursos físicos multiplexados por el sistema (como marcos de página, líneas de interrupción, etc.). Estos recursos o *unidades* elementales son las mínimas unidades asignables, no son capaces de moverse y mantienen una posición fija dentro del elemento en que están contenidos (ej. un marco de página está *siempre* en una posición concreta—determinada por el número de marco—dentro de un banco de memoria.)

[4]Esto es, que dados dos nombres cualesquiera ambos deban presentar siempre la misma estructura.

Dado que en cada uno de los puntos anteriores las necesidades son diferentes *Off* trata de adoptar en cada caso los nombres más adecuados sin intentar imponer el mismo modelo de nombrado a *todos* los recursos existentes. De hecho, el énfasis en el uso de un *mismo* modelo de nombrado para *todos* los recursos es algo que tiene que ver con el suministro de abstracciones y no con la conveniencia y eficacia del sistema de nombrado.

De haber impuesto el mismo tipo de nombres a *todos* los recursos habríamos pagado el precio (en eficiencia y rendimiento) que pagan otros sistemas que lo hacen y habría sido necesaria la introducción de niveles intermedios de traducción de nombres.

Recursos abstractos móviles

En el primer caso (recursos lógicos móviles), dado que tanto shuttles como portales y DTLBs no corresponden realmente a recursos físicos; y dado que dichas entidades son capaces de *moverse* a lo largo de la red, resulta más conveniente el empleo de nombres que faciliten la localización de dichos elementos (shuttles portales y DTLBs). El empleo de nombres de "alto nivel de abstracción" en este caso no es un inconveniente dado que dichos recursos carecen de nombres "físicos", con lo que tenemos completa libertad en la diseño de sus nombres (no violamos la directriz 3, según la cual los nombres deben corresponder con los empleados por el hardware, al nombrarlos como deseemos hacerlo).

Estos recursos están nombrados con identificadores que presentan unicidad en la red y contienen la posición original del objeto nombrado (el nombre del nodo en que se creó dicho objeto) para ayudar en su localización. Para simplificar la implementación, cada tipo de recurso posee un espacio de nombres propio. Así, un mismo identificador puede hacer referencia a un Shuttle, un Portal o una DTLB dependiendo del contexto en que se emplee.

Estos identificadores están compuestos de una parte pública y una privada (ver figura 2.3-a):

- La *parte pública* es única en todo el sistema e identifica unívocamente un objeto dentro de un espacio de nombres distribuido. Así, podría haber un shuttle y un portal con la misma parte pública (dado que los espacios de nombres son distintos) pero nunca podrá encontrarse a dos portales con idéntica parte pública en la misma red.

- La *parte privada*, redundante y sólo útil para mejorar el rendimiento, es un conjunto de bits que sólo tiene sentido para el servidor que implemente el objeto al que se refiere el nombre. Este conjunto de bits puede considerarse como información del implementador del objeto que puede viajar con el nombre del objeto. En el apartado 2.1.7 veremos su utilidad.

Aunque la estructura de estos identificadores no es en realidad muy diferente de la adoptada en otros sistemas como Amoeba [121] o Aegis [60]. La diferencia estriba en que *Off* permite que la parte privada de un identificador cambie a discreción. Esto resulta útil cuando el objeto nombrado se puede mover de un servidor a otro (i.e. de un contenedor a otro) como veremos más adelante en el apartado 2.1.7 dedicada discutir la localización de objetos.

Centrándonos ahora en la parte pública de un identificador, ésta está a su vez dividida en dos partes:

- El campo *nodo*, que identifica el nodo donde se creó el objeto nombrado.Éste campo refleja el lugar donde se creó el objeto y, aunque que no tiene por qué coincidir con la ubicación actual del objeto, puede utilizarse como pista a la hora de localizarlo.

- El campo *secuencia*, que hace que el identificador sea único. Para ello se utiliza un contador almacenado en memoria no volátil[5]

[5]Ello no supone pérdida de eficiencia puesto que no es preciso acceder a memoria no volátil siempre que se incrementa

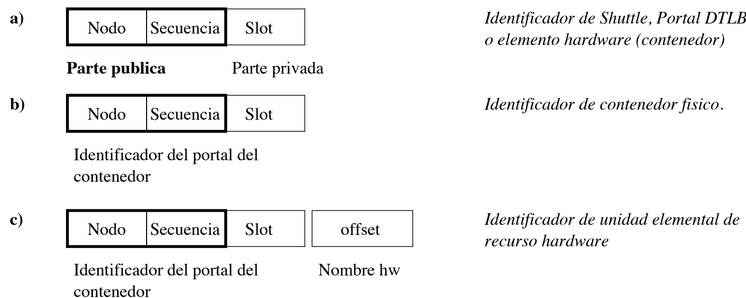

Figura 2.3: Identificadores en *Off*

Es conveniente recordar que dado que la parte privada puede cambiar, las comprobaciones de igualdad de identificadores deben restringirse a la parte pública de los mismos.

Recursos físicos inmóviles

En el segundo caso, para nombrar "contenedores" o elementos físicos que contienen unidades más elementales (ej. bancos de memoria que contienen marcos, etc.) empleamos como nombres identificadores de portales. Dichos portales son los puntos de acceso a los gestores de los elementos considerados. Así, dado un elemento físico, su nombre es en realidad el identificador del portal empleado para contactar con su gestor (ver figura 2.3-b).

Por ejemplo, si el portal al que hay que dirigir las peticiones relativas a un banco de memoria tiene como identificador `p`, entonces `p` es también el nombre de dicho banco de memoria y como veremos en el siguiente apartado, los marcos de página contenidos en dicho banco tendrán nombres de la forma `p:o`.

El empleo de un identificador de portal permite mantener el identificador de un elemento incluso cuando éste se reemplaza por otro situado en un nodo distinto (ej. Un banco de memoria que se reemplace por otro mantendrá su identificador).

Unidades elementales de recursos físicos

En el tercer caso (unidades elementales de recursos físicos inmóviles) hemos optado por *extender* el nombre físico de tal modo que el nombre ofrezca la información presente en el nombre físico y, al mismo tiempo, presente la unicidad deseable para preservar su significado en toda la red utilizada (ver figura 2.3-c).

Así, toda unidad elemental de un recurso físico (ej. un marco de página) estará identificada de la forma `p:o`, siendo `p` es el identificador del elemento en que está contenido (ej. el banco de memoria donde se encuentra) y `o` su nombre físico (ej. el número de marco o, en inglés, PFN).

De este modo podemos obtener a un bajo coste (aplicando una máscara) la información que suministran los correspondientes nombres físicos y además es posible localizar eficientemente el recurso cada vez que sea necesario (extrayendo de su identificador el de su contenedor, que corresponde al portal del gestor del recurso como ya vimos en el apartado anterior).

dicho contador. El contador podría incrementarse *n* veces sin acceder a memoria no volátil siempre que en tiempo de arranque se incremente *n* veces el contador.

Servicios de directorio

Hay que tener en cuenta que los nombres que hemos discutido en los apartados anteriores se utilizan para nombrar recursos del μkernel, y no recursos de mayor nivel de abstracción tales como *ficheros*, *procesos*, etc.

En este sentido, es posible afirmar que *Off* no incorpora realmente un *servicio de nombres* como tal. O, dicho con más precisión, el μkernel carece de servicios de directorio, cada aplicación puede utilizar el esquema de nombrado (estático, dinámico, centralizado, distribuido, etc.) que más le convenga y utilizar los nombres de los recursos suministrados por *Off* como si de nombres físicos se tratase.

Con ello respondemos a la necesidad de distinguir entre denotación (qué objeto es para nosotros el nombrado por un nombre) e identificación (de qué objeto estamos hablando realmente) de los recursos utilizados. Esta distinción es común en la vida cotidiana donde, por ejemplo, cada persona está identificada unívocamente por un identificador (por ejemplo, su D.N.I) y a pesar de ello se emplean distintos nombres para referirse a la misma (por ejemplo, "mi hijo", "el profesor", "Paco", etc.)

La disponibilidad de sistemas de nombrado de nivel de aplicación permite la realización de entornos en los que diferentes aplicaciones pueden percibir el mismo objeto como objetos diferentes y, del mismo modo, objetos diferentes como si fuesen el mismo. La relatividad de la identificación de objetos puede ahora modelarse dependiendo del contexto.

Por último, otra razón por la que no se suministran servicios de directorio es que otros sistemas han utilizado ya con éxito espacios de nombres en el nivel de aplicación—que no se imponen globalmente por parte del sistema— comprobándose su alta flexibilidad y su conveniencia [130, 168].

2.1.3 Protección

Antes de discutir la protección de recursos del μkernel en *Off* conviene explicitar que hablamos de la protección de recursos *físicos*. Los recursos lógicos del SO (ficheros, procesos, etc.) implementados ahora por las aplicaciones pueden emplear el modelo de protección que se desee.

Se han empleado diversos mecanismos para suministrar protección en los SSOO existentes en la actualidad. Podríamos agruparlos en tres grandes familias:

Capabilities Son identificadores que permiten nombrar y proteger al mismo tiempo [108, 162]. La posesión de una capability permite al usuario la invocación de una o más operaciones en un recurso del sistema.

Su principal ventaja es que no requieren de una gestión centralizada y poseen un buen nivel de transparencia. Esto las hace convenientes para SSOO distribuidos. Su principal desventaja es la dificultad en la revocación de permisos y la necesidad de almacenar al menos una por cada cliente con derecho a utilizar el recurso afectado.

ACLs (Listas de control de acceso). Son listas que se asocian con los recursos a proteger e indican selectivamente qué clientes pueden realizar qué operaciones en un recurso determinado. . Su principal ventaja es que permiten una revocación sencilla.

Guardas Existentes en algunos sistemas adaptables. Son funciones booleanas asociadas al propietario del recurso que se protege que se evalúan para autorizar o prohibir cada acceso en el instante en que se produce. Una guarda recibe información sobre derechos de acceso del cliente y devuelve como resultado un valor booleano indicando la concesión o no del acceso. El funcionamiento de las guardas es simple (ver figura 2.4-b): el cliente realiza una petición de

acceso a un recurso; el implementador del recurso invoca a la guarda suministrada por el propietario del recurso; si la guarda devuelve *true* se concede el acceso, en otro caso se deniega.

La desventaja fundamental es la ineficiencia que resulta de la evaluación (la cual involucra *up-calls* en algunos sistemas). La ventaja es la flexibilidad, ya que permiten implementar capabilities y ACLs de forma selectiva.

El esquema de protección empleado por un DAMN debería permitir la implementación de *cualquier* sistema de protección en el nivel de aplicación, manteniendo dentro del μkernel sólo aquello indispensable para implementar otros mecanismos de protección en área de usuario. El sistema de protección empleado debe así mismo ser utilizable en una red, no sólo dentro de un nodo. También hay que reseñar que hablamos de protección de recursos físicos, el esquema de protección empleado para recursos lógicos puede ser radicalmente diferente. Este enfoque consistente en la protección de recursos físicos y la delegación de la protección de recursos lógicos a las aplicaciones resulta útil cuando se desean obtener sistemas altamente seguros, como se dice en [118]

En *Off* utilizamos un esquema intermedio entre capabilities y guardas que hemos denominado ***guardabilities*** (ver figura 2.4). Para cada unidad asignada de un recurso dado, *Off* mantiene una *guardability* que (dependiendo del valor que tenga) puede comportarse como una guarda o cómo una capability. La guardability está asociada al recurso que protege.

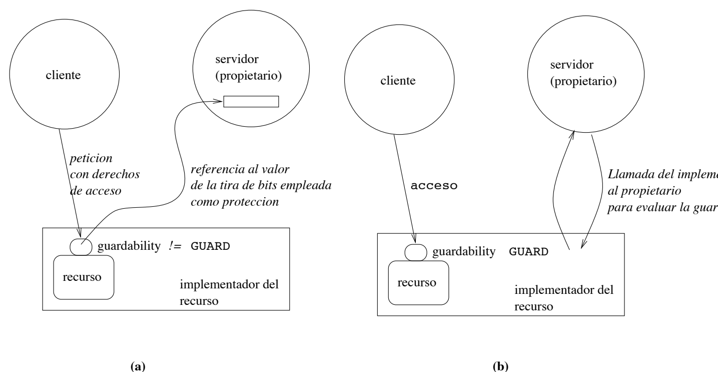

Figura 2.4: Protección en *Off*

Una *guardability* es un objeto capaz de almacenar alguno de los siguientes valores:

1. *GRANT*. Un valor especial que hace que en todos los casos, para una determinada operación, se apruebe el acceso del cliente que solicita el acceso al recurso.

2. *DENY*. Un valor especial que hace que se deniegue el acceso del cliente al recurso en todos los casos. Su utilidad radica en aquellos casos en que deseamos recursos que *sólo de usan una vez*. Por ejemplo, los marcos de página[6] empleados por el μkernel sólo se emplean cuando el μkernel instala traducciones hacia ellos. Una vez instaladas éstas, no volvemos a realizar operaciones sobre dichos marcos (aunque naturalmente sí que empleamos la memoria de éstos). Estos marcos debieran tener su protección a *DENY*.

3. *GUARD*. Un valor especial que hace que se invoque una guarda suministrada por el propietario del recurso. Del resultado de la invocación depende la aprobación o rechazo del acceso. Cuando el implementador del recurso invoca dicha guarda, le suministra la información (esto

[6] Concretamente, aquellos que sostienen páginas no expulsables.

es, los parámetros) que el cliente suministró en la petición. En base a dicha información (cuyo formato está preestablecido por el propietario del recurso) la guarda decide sobre la concesión del acceso.

4. Una referencia a una posición de memoria virtual interpretada en el espacio de direcciones del gestor del recurso. En dicha posición de memoria se encontrará una palabra seguida de una tira de bits de longitud variable. En este caso se compara la tira de bits (empleando la palabra precedente como longitud de dicha tira) del propietario del recurso con la suministrada por el cliente o peticionario. Si ambas tiras de bits coinciden en longitud y valor se concede el acceso.

El primer y el segundo caso permiten implementar eficientemente todas aquellas situaciones en las que los accesos se conceden o deniegan globalmente: recursos de dominio público —tales como las listas de asignación de recursos que mantiene el kernel— y recursos utilizados sólo una vez —como los marcos de página empleados por el kernel en el ejemplo anterior.

El tercer y cuarto caso permiten el empleo de guardas (y por tanto, ACLs) y capabilities allí donde se considere conveniente.

Hay dos elementos que conviene explicar, uno es el concepto de *propietario* de un recurso. Este concepto se utiliza cuando un recurso tiene una guardability con valor *GUARD*. Para *Off*, cada unidad de recurso del sistema asignada tiene un propietario identificado por un portal. Debemos mencionar en este punto que la implementación de esta abstracción "propietario" es tarea del SO o de las aplicaciones que ejecutan sobre *Off*. En *Off* no hay "procesos" ni otras abstracciones que puedan considerarse como "propietarios" de los recursos asignados. *Off* tan sólo necesita un punto de acceso (un portal) al que enviar las excepciones y notificaciones relacionadas con la unidad de recurso de que se trate.

El segundo elemento es el empleo de una referencia a memoria de usuario para almacenar la tira de bits que permite validar un acceso. El mantenimiento de esta tira en área de usuario permite una revocación rápida de recursos y el empleo de métodos eficientes en espacio para almacenarlas (por ejemplo, el empleo de una única tira de bits para más de una unidad de recurso). Evidentemente, cuando dicha memoria es remota puede producirse un trasiego de páginas entre distintos nodos, así que es conveniente que las aplicaciones mantengan compactadas todas sus guardabilities.

Los derechos de acceso suministrados por los clientes en sus peticiones no identifican a ningún recurso del mismo modo que los identificadores no se emplean como mecanismo de protección en *Off*.

Para optimizar todos aquellos casos en los que determinadas operaciones están siempre prohibidas *Off* almacena una máscara de bits junto con cada guardability. Cada bit corresponde a una posible operación sobre la unidad de recurso a la que protege. Si el bit asociado a una operación es cero, dicha operación se prohibe y cualquier acceso que la invoque será inmediatamente rechazado.

La separación entre identificadores, o nombres, y derechos de acceso hace más simple la implementación y permite que las aplicaciones tomen el relevo al kernel en la protección del sistema (el μkernel sólo necesita identificadores para manipular recursos, el modelo de protección puede pues alterarse sin modificar el μkernel). Creemos que el modelo de protección debería estar implementado en el nivel de aplicación, eliminando la responsabilidad del kernel tanto como sea posible. Haciendo que el sistema utilice nombres al margen del mecanismo de protección empleado abrimos la posibilidad de experimentación con distintos modelos de protección a nivel de aplicación.

Creemos que el modelo de protección adoptado en *Off* es tanto simple como potente y, en cualquier caso, cualquier otro modelo puede incorporarse en el futuro mediante la adaptación del sistema.

2.1.4 Asignación de recursos

La asignación de recursos (y también la revocación, como veremos en el apartado 2.1.5) se realiza de forma distribuida en un DAMN. Aunque tal vez puede pensarse que esto es una desventaja con respecto a la asignación de forma centralizada veremos que no es así.

La asignación óptima de recursos, o la aproximación a una solución óptima parece ser más fácil en sistemas de gestión de recursos centralizados, al disponerse de toda la información pertinente. No obstante, en un sistema distribuido no es conveniente centralizar dicha gestión puesto que estaríamos complicando el sistema y creando un punto único de fallo, además de empeorar el rendimiento dado que todos los nodos requerirían los servicios del único gestor.

Lo que es más, en general no es cierto que la implementación de una solución óptima (o una aproximación a la misma) requiera de un gestor centralizado. En la práctica basta con disponer de la información adecuada para tomar buenas decisiones. El problema queda reducido al suministro de información a aquellas partes que participan en la asignación de recursos.

Consideremos ahora el hecho de que estamos implementando un sistema distribuido. Como K. Li y P. Hudak afirman [109], es difícil encontrar un modo de distribuir la responsabilidad de la gestión (y localización) de recursos de un modo que convenga a todas las aplicaciones. La distribución de la gestión de recursos de forma dinámica en *Off* permite que distintos tipos de aplicación la implementen a su conveniencia. La ayuda ofrecida por el sistema a la gestión de recursos (la exportación a las aplicaciones del estado de los mismos y participación de las mismas en la revocación de éstos) y, al mismo tiempo, la libertad ofrecida a las aplicaciones en la elección de algoritmos empleados a tal efecto permiten el uso de modelos de asignación de recursos a medida de las aplicaciones.

Por lo que sabemos, sólo existe otro sistema —aunque este es centralizado— en el que son las aplicaciones las que gestionan sus propios recursos, Aegis [60]. Salvo en lo referente a la distribución física del sistema, lo que aquí decimos es también aplicable a dicho sistema, del mismo modo que (si obviamos la distribución) lo aprendido en el desarrollo de Aegis es aplicable a un DAMN.

Para que dicha gestión se pueda realizar de forma distribuida (entre las aplicaciones y el DAMN) es necesario que el sistema exponga la información necesaria para implementarla. Dicha información, al contrario que en Aegis [60], no está limitada a un nodo. Si una aplicación utiliza recursos procedentes de varios nodos de la red, la información relevante sobre todos ellos estará disponible con independencia de la ubicación concreta que tenga. La información expuesta por el sistema —como ocurre también en Aegis— incluye el estado de los recursos que posee la aplicación y la lista de recursos libres.

Consiguientemente las aplicaciones pueden emplear los algoritmos que deseen para escoger los recursos que prefieran. Las interfaz de asignación de recursos de un DAMN debe pues ser capaz de expresar el deseo de utilizar o no una unidad concreta de un recurso determinado (por ejemplo, un manejador de disco de un Intel puede expresar su deseo de utilizar marcos de página locales de los primeros 16M capaces de soportar DMA).

Veremos ejemplos concretos de asignación de recursos en los próximos capítulos, cuando detallemos la realización y el funcionamiento de las abstracciones del sistema.

Si los recursos fuesen infinitos, con lo ya dicho en este apartado bastaría. Dado que los recursos son finitos, es preciso revocar recursos eventualmente.

2.1.5 Revocación de recursos

La revocación de recursos es más compleja que la asignación dado que habitualmente acarrea agravios para las aplicaciones cuyos recursos se revocan.

En sistemas convencionales es el SO el que decide de forma inapelable qué recursos y a quién deben revocarse. En la mayoría de los casos las aplicaciones ni siquiera pueden percatarse de la

revocación y toman sus decisiones ignorantes de este hecho. La ignorancia de las aplicaciones ante la revocación es una causa de problemas derivados de falsas suposiciones sobre la misma.

Consiguientemente, en *Off* hemos optado por el uso de revocación explícita, en la que el μkernel permite que las aplicaciones conozcan cuando y qué recursos se les van a revocar. Hay también algunos SO adaptables que se comportan de este modo, aunque otros no permiten a las aplicaciones tener conocimiento de revocaciones. Sólo unos pocos SO adaptables permiten a las aplicaciones participar en la decisión de revocación.

Las preguntas que hay que contestar para diseñar un mecanismo de revocación de recursos son (ver tabla 2.1):

	¿Cuando?	**¿Quién?**	**¿Ámbito?**	**¿Cómo?**
SO monolítico	siempre	kernel	sistema	implícita
μkernel	siempre	μkernel	sistema	implícita
exokernel	en demanda	kernel+aplicación	aplicación	explícita
DAMN	en demanda	aplicación	aplicación	explícita

Tabla 2.1: Revocación de recursos en SSOO

- *¿Cuándo se revocan los recursos?*
  Habitualmente es el SO el que decide el momento apropiado en que se han de revocar recursos. El mecanismo empleado suele ser el uso de umbrales de revocación, de tal modo que si la cantidad usada de un tipo de recurso sobrepasa un umbral determinado el sistema revoca unidades de dicho recurso para aumentar su disponibilidad.

  No obstante, en sistemas monolíticos y μkernels no adaptables la situación suele ser aún más dictatorial ya que en ciertas ocasiones el sistema reclama recursos incluso cuando no es necesario hacerlo.

  En *Off* el μkernel dispara un suceso de revocación cuando hay peticiones de asignación que no pueden satisfacerse, esto es, cuando no hay más unidades libres del recurso que consideremos. En aquellos casos en que desee mantenerse un porcentaje libre de un tipo de recurso dado deberían ser las propias aplicaciones las que iniciasen la revocación.

- *¿Quién decide qué recursos hay que revocar?*
  El ejecutor del algoritmo de revocación suele ser también el SO. En SSOO monolíticos y en aquellos que utilizan μkernels no adaptables la implementación está centralizada en el núcleo del sistema.

  En μkernels adaptables, exokernels y DAMNs las aplicaciones suelen participar en la elección de recursos a revocar. En *Off* el μkernel sólo dispara un suceso de revocación como vimos en el punto anterior. La decisión está distribuida por completo entre las aplicaciones involucradas, como veremos al final de este apartado.

- *¿Qué ámbito tienen los recursos que se revocan?*

  Tradicionalmente el SO revoca los recursos que él controla. En SSOO distribuidos basados en μkernel éste revoca recursos restringidos a un nodo, con independencia de la aplicación que los utiliza. Esta revocación puede ser adecuada en relación a la disponibilidad de recursos locales aunque, en muchas ocasiones, puede no ser la óptima considerando todos los nodos involucrados.

En exokernels los recursos se revocan también localmente, dentro del nodo considerado. En este caso las aplicaciones pueden decidir qué unidades liberar del recurso revocado.

En DAMNs el μkernel no revoca recursos, tan sólo dispara sucesos de revocación. Las aplicaciones deberán recurrir a algún mecanismo de arbitraje que permita seleccionar los recursos a revocar. En este caso es posible que dicho arbitraje actúe de forma diferente en función de las circunstancias, operando localmente en unos casos y globalmente en otros.

Considerando ahora el caso de un DAMN, la revocación se basa en los siguientes puntos:

- El μkernel no fuerza la revocación, sólo notifica la necesidad de revocar recursos para satisfacer peticiones de asignación.
- El sistema de protección empleado debe permitir a un árbitro forzar dicha revocación en caso necesario.
- Debe permitirse la existencia de múltiples árbitros de tal modo que cada conjunto de aplicaciones puede emplear un mecanismo diferente de arbitraje.

Como puede verse, *Off* sólo dispara un evento de revocación para un recurso determinado cuando alguna petición de asignación no puede satisfacerse. El μkernel ha concluido con eso su trabajo.

En el caso de aplicaciones que compiten por un recurso será el árbitro el que establezca un reparto adecuado del mismo entre ellas. En general, la revocación de los recursos en un nodo podría proceder como sigue:

1. En un momento dado un porcentaje de los recursos locales podrían ofrecerse al resto de la red. Así un servidor compartido ofrecería el 100%, una estación de trabajo podría ofrecer el 100% cuando el propietario no la está utilizando y bajar a un 30% o un 60% cuando el propietario la use.
2. Un árbitro por nodo puede mantener dichos porcentajes.
3. Mientras estos porcentajes estén satisfechos los árbitros pueden revocar los recursos en función de otros criterios mas tradicionales (conjunto de trabajo, uso reciente, etc.)
4. Una vez elegida una aplicación por un árbitro, esta puede intentar convencer a otras para que liberen recursos en su lugar.
5. La aplicación resultante puede escoger las unidades que menos necesita.

Si las aplicaciones funcionan correctamente este sistema no causa problemas. Si las aplicaciones se comportan con malicia o error la responsabilidad de revocar los recursos cae ahora en el árbitro. Este árbitro puede incorporar un esquema de liberación mixto similar al empleado en Aegis:

- Primero se pide con revocación explícita la liberación de recursos a una aplicación.
- Si pasado cierto tiempo no aumenta la disponibilidad en dicho recurso, se procede a una revocación implícita.

2.1.6 Distribución de recursos

Cada aplicación considera el nodo local como una cache de los recursos disponibles en todo el sistema distribuido. En el caso de aplicaciones centralizadas, éstas se limitan a utilizar dicha cache ignorando la ubicación de los recursos (pensando que son locales). En cambio, las distribuidas pueden solicitar la asignación de recursos en las ubicaciones que deseen y controlar la revocación de tal modo que se mantengan en el nodo local (en la cache) los recursos convenientes (revocando primero aquellos recursos que sea más barato traer al nodo local, y no aquellos que sea costoso volver a obtener debido a su ubicación u otros factores). En este sentido es crucial que el μkernel permita a las aplicaciones escoger las unidades de recurso que han de revocarse, de otro modo el sistema escogería él mismo las unidades a revocar y ello sin tener una idea exacta de para qué se emplea cada una de ellas.

Sorprendentemente, en un DAMN, no hay un único modelo de distribución de recursos. El μkernel permite que peticiones locales al sistema puedan operar con recursos remotos, eso es todo lo que hace.

Por un lado, una aplicación centralizada se puede distribuir "automáticamente" interponiendo entre ella y el sistema un algoritmo distribuido de asignación y revocación de recursos. De este modo la distribución será como sigue:

- Ante una petición de recursos, el algoritmo de asignación puede solicitar recursos remotos (o locales) al μkernel.

- La aplicación realizará peticiones al sistema empleando dichos recursos de manera transparente. Sean éstos locales o remotos, el μkernel atenderá las peticiones.

- Ante una eventual revocación, el algoritmo de revocación empleado puede optar por eliminar primero los recursos que sean mas "baratos" en términos de posición y uso.

Por otro lado, una aplicación distribuida puede emplear algoritmos específicos de asignación y revocación sin necesidad de conformarse con un algoritmo general que funcione bien en el caso medio.

Un posible modelo para implementar estos algoritmos de asignación y revocación podría ser el *campo computacional* [164], donde las relaciones entre distintos objetos se tienen en cuenta para crear, destruir y migrar objetos.

Por último, conviene dejar claro que en un DAMN no sólo se distribuyen las IPCs, esto es, no sólo se permiten interacciones entre elementos en distintos nodos. Para cualquier servicio del sistema, la operación se procesa en el nodo local tanto como sea posible. Cuando el sistema ve que el recurso es remoto es la propia implementación del servicio la que contacta con el nodo remoto usando protocolos específicos de cada aplicación (esto es, realizando una *up-call*). Esto no es lo mismo que emplear una IPC distribuida que alcanza un núcleo remoto sin que el local se entere de ello. Si se distribuyen sólo las IPCs podemos tener problemas en el uso de referencias a memoria de usuario en las llamadas al sistema (una referencia local no es válida en el nodo remoto). Estas pueden ocasionar mensajes extra en la red o el envió de datos innecesarios.

A modo de ejemplo, la figura 2.5 muestra (a) cómo las aplicaciones utilizan la distribución del sistema en μkernels centralizados convencionales con IPC distribuida (como en el caso de Mach con un *netmsgserver* que extiende la IPC de Mach a la red) y (b) cómo pueden emplear su propia distribución en un DAMN.

En la figura se aprecian varios procesos de usuario (círculos) que efectúan llamadas a servicios del sistema (línea continua) y a servicios de un proceso remoto (línea discontinua). En el caso de un μkernel tradicional todos los procesos de usuario se ven obligados a utilizar un *mismo* protocolo

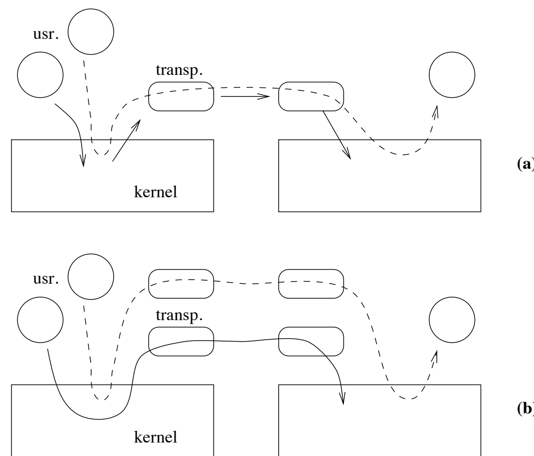

Figura 2.5: Distribución del sistema en μkernels (a) y DAMNs (b).

de transporte. En un DAMN cada aplicación puede utilizar su propio transporte (un servicio de datagramas, uno orientado a conexión, con o sin cifrado, etc.).

Lo que es más, en el caso de la llamada al sistema, un μkernel tradicional no permite llamadas al sistema desde otros nodos. Es el propio μkernel el que, *imponiendo* su modelo de distribución, utiliza el sistema de transporte para permitir que sus abstracciones utilicen recursos remotos (como denota la separación en varios tramos del camino de la llamada al sistema en la figura). En un DAMN el sistema se limita a encaminar aquellas peticiones dirigidas a recursos remotos al servidor especificado por la aplicación involucrada. Esto afecta tanto a las llamadas a servicios del sistema como a los mensajes de aplicación dirigidos a otros procesos de usuario.

2.1.7 Localización de recursos

Los recursos físicos mantienen siempre su posición y, por tanto, no son móviles. No obstante, hay dos casos que aparecen al considerar un DAMN que requieren de ciertas técnicas relacionadas con la "movilidad de recursos":

- Un elemento físico puede ser reemplazado por otro elemento similar en una posición (i.e. nodo) diferente. Dicha operación, cuando se realiza *en caliente* equivale a *mover* el elemento físico considerado.

- Las abstracciones lógicas empleadas por un DAMN para multiplexar el hardware no están sujetas a la restricción de inmovilidad a que se ven sometidos los elementos físicos. Así, en el caso de *Off*, tanto Shuttles, como portales y DTLBs podrían moverse de un nodo a otro.

En ambos casos el sistema debe hacer frente a cambios en la posición de recursos del sistema. Elementos físicos—p.ej. bancos de memoria—que contienen unidades elementales de recursos—p.ej. marcos de página—, en el primer caso; abstracciones del sistema—p.ej. Portales en *Off*— en el segundo caso.

Como ya se describió anteriormente, *Off* utiliza identificadores que ayudan a localizar recursos que son susceptibles de cambios de posición. De este modo, con ayuda del estado exportado por el kernel, es posible tolerar el cambio de posición de dichos recursos. Por ejemplo, podemos capturar el estado de un recurso determinado y emplear dicho estado con posterioridad para "recrearlo" en una ubicación distinta; el identificador de dicho recurso seguirá siendo válido.

En el caso de movimiento de elementos físicos, al estar estos identificados con el portal del gestor, basta cambiar el manejador de dicho portal para que el gestor del elemento opere en la nueva posición. Los nombres de las unidades elementales contenidas en el elemento mantienen su validez al ser relativos al identificador del gestor.

A modo de ejemplo, los nombres de los marcos de página de un banco de memoria siguen siendo válidos si el banco de memoria se reemplaza por otro en un nodo distinto: los marcos están nombrados por identificadores `p:o`, donde `p` identifica el banco de memoria y por tanto es el portal del gestor de dicho banco de memoria (ver apartado 2.1.2). Si el banco cambia de nodo, pero el gestor mantiene el portal `p`, las peticiones destinadas hacia los marcos seguirán funcionando correctamente. Los marcos se pueden replicar en el nodo destino (antes de desmantelar el banco de memoria origen) dado que el kernel no oculta ni el estado ni el contenido de los marcos. Alguien con privilegios suficientes puede sencillamente leer dicho estado (qué marcos están asignados, a quién, etc.) y replicarlo en el banco de memoria destino. Para replicarlo basta con copiar la memoria que lo contiene, no es preciso hacer traducción alguna puesto que los identificadores (referencias) contenidas en dicho estado tienen validez en todo el sistema distribuido.

El caso más delicado es el movimiento de abstracciones y consiguientemente lo trataremos en más detalle. En el caso de portales, shuttles y DTLBs sus identificadores están diseñados para facilitar su cambio de posición. Concretamente, dos de los tres componentes de los identificadores se emplean en la localización de los objetos nombrados con ellos:

nodo Indica la localización (el nodo) que se intentará en primer lugar para objetos que no son locales. Si dichos objetos no han migrado estarán aún en el nodo en que se crearon, que es precisamente el indicado en este campo.

secuencia No se utiliza para localizar recursos.

slot (o parte privada del identificador). Permite a un servidor (o implementador) de objetos indexar éstos. Este campo contiene típicamente un valor (esto es, un índice) que permite al implementador de un recurso (p.ej. al servidor de portales) localizarlo eficientemente (p.ej. con una operación de indexación).

En realidad, podemos pensar en la parte privada de los identificadores como en información privada del implementador del objeto que viaja junto con el nombre del mismo. Esta información puede utilizarse del mismo modo que se utilizan tablas *hash* en otros sistemas para localizar la implementación de un objeto a partir de su identificador [168, 2].

Para comprender mejor lo expuesto anteriormente, consideremos lo que ocurre cuando se crea un portal y también qué sucede cuando dicho portal se mueve de un nodo a otro.

Al crearse el portal se construye su identificador. El campo de nodo y secuencia se asignan de tal modo que el primero refleje el nodo donde se creó dicho portal y que ambos sean únicos (conjuntamente) a todo lo largo de la vida del sistema. El servidor de portales (el implementador de los portales) ha de asignar una estructura de datos nueva para mantener los datos del portal creado. Las estructuras de datos o descriptores de portales están situadas en un vector dentro del servidor de portales. Como puede suponerse en este punto, el índice en dicho vector para el portal creado se almacena en *slot* dentro del identificador de dicho portal.

Cuando un usuario del sistema referencia el portal, el servidor de portales extrae *slot* del identificador e indexa en su vector de portales para obtener el descriptor. Si el portal no ha migrado se ha localizado con una sóla indexación. No ha sido preciso el empleo de tablas hash, estructuras arborescentes u otro mecanismo más costoso.

Supongamos ahora que dicho portal cambia su posición. Cualquier cliente de dicho portal enviará una petición (como hacía siempre) al nodo de creación del mismo. Ahora bien, el servidor de portales

de dicho nodo indexará (utilizando de nuevo el campo *slot*) en su vector de portales y encontrará o una entrada libre, o un portal distinto[7] (los campos nodo y secuencia serán distintos).

En este punto se eleva una excepción que deberá tratar la aplicación. El propósito de dicha excepción es permitir a distintas aplicaciones el uso de distintos protocolos de localización. Aquellas aplicaciones que lo deseen, pueden hacer que un servidor—implementando un protocolo de localización de propósito general—trate esta excepción.

El manejador de la excepción localiza el portal (como veremos más adelante) y notifica al cliente de la nueva localización (en efecto, hace cache del nombre). De aquí en adelante el cliente enviará cualquier futura petición al nodo adecuado. Dado que el identificador de un portal es en realidad el valor de su parte pública (campos nodo y secuencia), en la notificación de relocalización recibida puede recibirse un nuevo valor para el campo *slot* del nombre del portal.

En pocas palabras, sólo la primera llamada a un portal tras la migración del mismo ocasiona más trabajo que una invocación antes de dicha migración. Posteriores llamadas presentan el mismo coste en tiempo que para aquellos portales que no han cambiado de posición. Es de reseñar que el μkernel no incorpora ningún protocolo concreto y tan sólo mantiene la última localización conocida, de este modo diferentes aplicaciones pueden optar por diversos esquemas a la hora de tolerar migración de recursos.

Nótese que la libertad en la implementación del protocolo de localización es casi absoluta. En aquellas ocasiones donde las migraciones son predecibles (p.ej. si se definen en tiempo de compilación) el protocolo de localización puede ser una búsqueda en una tabla. En redes de área local se pueden emplear algoritmos que aprovechen facilidades de difusión (*broadcast*), si las hay. También es posible implementar cualquier estructura de cache jerárquica de ubicaciones que sea imaginable para evitar ejecuciones del algoritmo de localización.

Para permitir la implementación de estos protocolos de localización, el μkernel permite la lectura de la memoria que contiene el vector de portales directamente desde nivel de usuario.

Por último, hay que tener presente que los usuarios realizan peticiones al servidor de portales de su nodo (de forma transparente), siendo éste el que se encarga de reenviar la petición (si procede, y mediante protocolos suministrados por las aplicaciones) al nodo apropiado de acuerdo con la información presente en la parte pública del identificador de portal.

Hablaremos más de la movilidad de recursos abstractos del sistema en el apartado 2.4.

2.2 Las abstracciones del sistema

En general, y antes de discutir el diseño de cada abstracción en particular, hemos seguido los siguientes pasos en el diseño de las abstracciones del sistema:

- Partir de la "abstracción" implementada por el hardware.
- Abstraerla lo necesario para que pueda utilizarse (referenciarse e invocarse) desde otros nodos de la red.
- Mantenerla en ese nivel de abstracción.

Al contrario que Engler [60], nosotros no creemos que si una abstracción es "portable" su nivel de abstracción es (probablemente) demasiado alto. En este sentido, la filosofía de *Off* está de acuerdo con lo sostenido por J. Liedtke [111]:

[7] Esto implica que todo elemento contenido en dicho vector deberá contener su identificador, para que sea posible efectuar una comparación entre identificadores y averiguar qué objeto está contenido en cada entrada.

> "Es posible alcanzar μkernels con buen rendimiento mediante implementaciones específicas [para una arquitectura] de abstracciones independientes [de la arquitectura]."

La diferencia estriba en que, en el caso de *Off*, estas *abstracciones* son tan sólo las suministradas por el hardware. Cualquier otra abstracción de más alto nivel debiera ser suministrada por las aplicaciones. Un buen esquema para ello podría ser el uso de la orientación a objetos tal y como pusieron de manifiesto (ver alguno de [28, 32, 142, 115, 29]) sistemas como Choices [31] y μChoices. [].

Retomando el compromiso entre adaptabilidad y transparencia en la distribución, mencionado en el apartado 1.1, éste no parece deberse a ninguna propiedad inherente a los sistemas distribuidos. Parece deberse en cambio al nivel en que situamos la barrera entre el μkernel y los servicios realizados en área de usuario: si la situamos en niveles superiores perdemos adaptabilidad y ganamos transparencia, si la situamos en niveles inferiores perdemos transparencia y ganamos adaptabilidad.

En *Off* hemos tratado de "bajar" esa barrera tanto como sea posible y nos hemos centrado en el diseño de tres abstracciones elementales, utilizadas como base para que las aplicaciones puedan implementar servicios de más alto nivel sobre ellas tales como gestión distribuida[8] de procesos y memoria junto con mecanismos de IPC.

La elección de cuántas y qué abstracciones debe suministrar el sistema es siempre complicada y, como ya dijimos en el capítulo anterior, siempre suele resolverse insatisfactoriamente (al menos para algunas aplicaciones). Otro agravante es la contradicción existente en el proceso de elección [39]:

> La facilidad de reconfiguración requiere abstracciones únicas e inamovibles; la flexibilidad se mejora con el uso de múltiples abstracciones; y, la eficiencia mejora cuando no hay abstracciones.

En el prototipo de DAMN hemos adoptado como abstracciones aquellas derivadas (mediante su extensión a la red) de las suministradas por el hardware y algunas otras (como shuttles y portales) que son necesarias para implementar en área de usuario servicios tradicionalmente suministrados por el sistema operativo (gestión de procesos e intercomunicación de procesos).

Hemos partido de las siguientes "abstracciones" suministradas por el hardware disponible en la red:

- Contextos de procesador. Constituidos por un conjunto de valores para los registros disponibles, incluyendo también aquellos registros que definen parte del contexto como registros base de tablas de páginas, etc.

- Interrupciones y excepciones. Constituidas por un número asociado a la línea de interrupción o excepción considerada y por determinados valores que identifican el manejador del evento considerado.

- TLBs y Tablas de páginas. En realidad, un conjunto de traducciones (incluyendo protecciones) de direcciones virtuales de memoria a direcciones físicas de memoria.

- Marcos de página. Fragmentos de memoria que pueden ser el destino de una traducción en una TLB o tabla de páginas.

- Puertos de E/S. Direcciones de un nodo que admiten operaciones de E/S.

Estos elementos implementados directamente en hardware se han abstraído hasta el punto en que pueden referenciarse y utilizarse de forma remota. Este proceso de abstracción ha permitido dotarlos de la capacidad de migrar y persistir y, debido al escaso nivel de abstracción empleado, mantener al mismo tiempo la adaptabilidad.

[8] También puede implementarse de forma centralizada, al igual que ocurre con los dos puntos siguientes.

- **Shuttles**. Contextos hardware extensibles reemplazan en *Off* a los contextos de procesador. Pueden instalarse en un procesador para su ejecución y son el soporte para la ejecución de *tareas*, *procesos* y *threads* o flujos de control —abstracciones estas que debe implementar el software de aplicación que ejecute sobre *Off*.

  Los Shuttles pueden cambiar de procesador al igual que en sistemas multiprocesador, con la salvedad de que en *Off* los procesadores origen y destino pueden residir en distintos nodos. La estructura que presentan internamente (esto es, los valores de los registros del procesador y de las extensiones que presenten) es accesible al usuario lo mismo que ocurre con los contextos hardware.

  No soportan migración entre sistemas heterogéneos ya que no hay un único y mejor modo de implementar esta habilidad. No obstante, se le suministran al usuario los mecanismos necesarios para implementar diversos sistemas de migración heterogénea.

- **Portales**. Abstraen las interrupciones hardware así como los traps y excepciones. Igual que ocurre con los Shuttles, no se fuerza la distinción entre portales locales y remotos. Una interrupción puede tratarse en otro nodo si se considera oportuno —aunque en general esto es una mala práctica.

  Permiten implementar diversos mecanismos de intercomunicación de procesos incluyendo RPCs, transferencias protegidas de control y envío de mensajes asíncronos. Además, como suele ocurrir con sistemas de IPC en otros μkernels, se utilizan como puntos de acceso al sistema de forma local o remota, de tal modo que en realidad sólo se utilizan llamadas al sistema para utilizar portales; el resto de los servicios están disponibles mediante portales.

- **DTLBs**. Abstraen el hardware de traducción de direcciones, permitiendo el uso de traducciones hacia marcos de página de otros nodos. Igual que ocurre en algunos multiprocesadores, la coherencia en el manejo de la memoria compartida se deja bajo el control de los usuarios del sistema.

Otros elementos manejados por el sistema son los siguientes:

- **Nodos**. Elementos que contienen procesadores, bancos de memoria, y puertos de entrada/salida.
- **Bancos de memoria**. Formados por un conjunto de marcos de página de un tamaño determinado.
- **Bancos de E/S**. Formados por un conjunto de puertos de E/S.
- **Marcos de página**. Instalados en un banco de memoria perteneciente a un nodo.
- **Puertos de E/S**. Instalados en un nodo. Incluimos en esta categoría los relojes existentes en la red.
- **Procesadores**. Capaces de ejecutar código, recibir interrupciones y traps.
- **Interrupciones**. Diferentes en cada procesador.
- **Traps**. Excepciones provocadas por la ejecución de código en un procesador.

Estos han necesitado pocas modificaciones con respecto a los suministrados directamente por el hardware. Por este motivo no son tan centrales en el diseño de *Off* como lo son las tres abstracciones básicas que mencionamos primero.

Dado que dedicamos un capítulo a las tres abstracciones básicas, discutiremos el resto de los servicios del sistema junto con la abstracción básica con la que se más se relacionen: veremos la gestión de procesadores al describir los Shuttles; interrupciones y traps al describir los portales; y los marcos de página y puertos de E/S al describir las DTLBs.

Lo que resta de capítulo lo dedicaremos a ciertos aspectos que, aún teniendo que ver con las abstracciones y servicios suministrados por el sistema, afectan al sistema en su conjunto.

2.3 Manejo de dispositivos

Off no multiplexa directamente los dispositivos secundarios. Tan sólo se ocupa de multiplexar puertos de E/S, de tan modo que cada puerto sólo puede pertenecer a una aplicación. Esta aplicación puede otorgar acceso al puerto a otras mediante el uso de *guardabilities* tal y como se indicó en el apartado 2.1.3 al discutir la protección incorporada por el sistema.

Aunque hay algunos intentos en multiplexación de dispositivos periféricos evitando la implementación de abstracciones en la medida de lo posible, el presente trabajo no trata de aportar soluciones nuevas en este sentido: la multiplexación de puertos e interrupciones realizada por *Off* permite implementar diversas soluciones ya conocidas para la multiplexación de dispositivos periféricos en área de usuario [49, 168, 62].

Un esquema simple (ver figura 2.6) que permite multiplexar dispositivos secundarios consiste en arrancar un “multiplexor” del dispositivo considerado (un disco o una impresora, por ejemplo) que retenga los derechos sobre los puertos de E/S y las interrupciones elevadas por el controlador. Dicho multiplexor puede multiplexar de forma segura el hardware disponible en el periférico (asignando atómicamente bloques o grupos de bloques y páginas o grupos de páginas en los ejemplos anteriores).

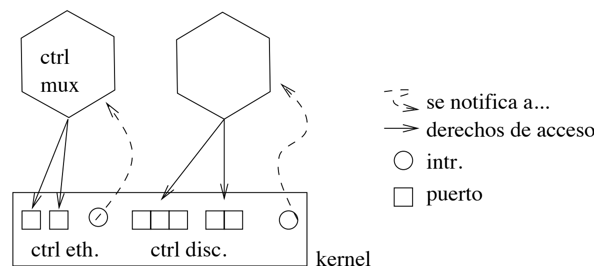

Figura 2.6: Multiplexores de dispositivos

Existirán casos en que un multiplexor deba operar durante toda la vida del sistema (como un multiplexor de red en el caso de una estación sin disco) y casos en que sea factible utilizar distintos multiplexores en distintos períodos de tiempo (como en el caso de una línea serie o una impresora). En los primeros, el multiplexor puede retener los derechos de acceso sobre los puertos e interrupciones del dispositivo y en los segundos el multiplexor puede bien compartir dichos puertos e interrupciones con otros multiplexores “cooperativos”, bien liberar los puertos e interrupciones en el momento en que se desee que otro multiplexor entre en funcionamiento.

Podemos pensar en varios esquemas de compartición de controladores entre varios multiplexores tal y como puede verse en la figura 2.7, de izquierda a derecha:

a. Un multiplexor puede otorgar derechos sobre parte de los puertos del controlador a otros multiplexores y sincronizarse con ellos para realizar operaciones (por ejemplo, un multiplexor de un controlador multipuerto que da derechos a otro multiplexor para utilizar alguno de los puertos).

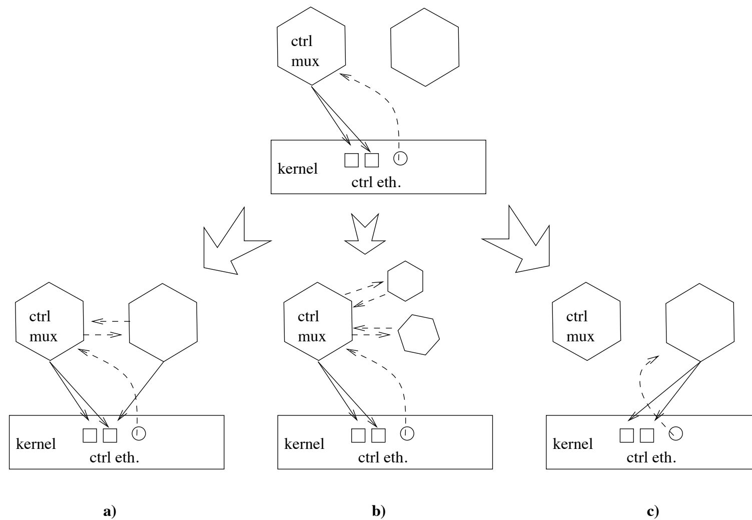

Figura 2.7: Multiplexores que compiten por un dispositivo.

b. Un multiplexor puede a su vez repartir el controlador de forma segura entre varios multiplexores (por ejemplo, un controlador de disco repartido en "discos" a su vez repartidos en "bloques" o un multiplexor de bloques que cede los derechos de algunos bloques a otro multiplexor de bloques).

c. Por último un multiplexor puede *retener* a otros que intentan multiplexar el mismo dispositivo hasta que él considere oportuno (tal vez, nunca) y en dicho momento cederle los derechos de acceso a los puertos y a la línea de interrupción.

2.4 Migración y persistencia

Hay recursos del sistema cuya migración no se produce nunca (marcos de página, procesadores, etc.). Hay otros donde esta migración resulta natural, como puede comprobarse por analogía con un sistema multiprocesador (Shuttles, DTLBs, etc.)

En un DAMN, todos aquellos recursos de naturaleza dinámica que son susceptibles de cambiar de nodo deben poder hacerlo. Así, en *Off*, tanto shuttles, como portales y DTLBs puede "moverse" o migrar de un nodo a otro.

Hay dos modelos de migración que deben tolerarse un DAMN: la implícita y la explícita. En una migración explícita es el usuario el que toma la iniciativa ejecutando alguna primitiva que causa la migración del objeto considerado. En una migración implícita el objeto migra sin que el usuario tenga necesidad de invocar primitiva alguna.

La migración implícita es importante en un DAMN puesto que permite que los usuarios que así lo deseen utilicen de manera transparente los recursos disponibles (tales como procesadores) con independencia de su posición en la red. Lo que es más, la migración implícita permite tolerar interdependencia entre varios recursos (p.ej. un shuttle necesita una DTLB para ejecutar) puesto que en todos aquellos casos en que se requiere de un recurso (p.ej. una DTLB) para emplear otro (p.ej. para ejecutar un shuttle en un procesador) el μkernel tomará la iniciativa efectuando una migración implícita del recurso correspondiente (p.ej. la DTLB empleada por el shuttle). La explícita permite a aquellas aplicaciones más sofisticadas tomar control de la migración implementando sus propias políticas.

En *Off*, las tres abstracciones básicas soportan las primitivas *freeze* (que congela el estado del objeto) y *melt* (que lo reanima, potencialmente en un nodo distinto). Estas dos operaciones permiten la implementación de diversos sistemas de migración en área de usuario.

La implementación de *freeze* y *melt* se apoya en la posibilidad de leer el estado del kernel desde área de usuario [166]. En realidad *freeze* se limita a mantener la validez del identificador del recurso a congelar y a bloquear todas aquellas peticiones dirigidas a dicho recurso. Cuando un recurso está congelado se opera igual que el caso de recursos remotos. Cualquier petición dirigida al mismo origina entonces una excepción de "objeto ausente" que el kernel envía al portal especificado por la aplicación. El manejador de dicho portal puede descongelar el recurso al vuelo, abortar la petición causando un error en el proceso de la llamada original o tomar cualquier otra acción pertinente. La primitiva *melt* se ocupa principalmente de recrear el objeto congelado recuperándose su identificador, protección y estado; esta primitiva podría usarse en el manejador mencionado anteriormente.

Cuando una petición al sistema requiere la presencia de un objeto de otro nodo (como ocurre cuando se instala para ejecución un Shuttle que anteriormente ejecutaba en un nodo distinto), *Off* inicia implícitamente la migración del objeto involucrado, elevando una excepción que tratará la aplicación. Esta utilizará las mismas primitivas que en el caso anterior y un protocolo de transporte de su elección.

Como puede suponerse, la persistencia es un efecto lateral de la capacidad de migración de los objetos del sistema; basta migrar el objeto hacia memoria no volátil[9].

Existen dos aspectos que conviene detallar en este punto: la protección y autenticación de los objetos *congelados* y la posible heterogeneidad de los sistemas origen y destino de una migración.

Como expondremos en los dos apartados que siguen, en *Off* se resuelven dichos problemas con el uso de aplicaciones escogidas por el usuario al arrancar el sistema y la definición de una excepción de autenticación y otra de traducción (tratadas estas por las mencionadas aplicaciones).

2.4.1 Autenticación y seguridad de objetos móviles

Cuando el usuario obtiene una copia congelada de un objeto del sistema existe la posibilidad de que éste realice cambios maliciosos destinados a la obtención de privilegios de modo ilegal. Sería fácil congelar un espacio de direcciones y en el proceso de copia hacia otro nodo instalar más traducciones, dado que la reanimación del objeto congelado requiere sólo de privilegios sobre el objeto a reanimar.

Tradicionalmente, este problema se evita *prohibiendo* al usuario la obtención de dicho estado. Sólo excepcionalmente, se permite su obtención prohibiéndose entonces la reanimación.

Otro aspecto a considerar es la confianza que el sistema destino puede o no tener respecto al sistema del que procede el recurso a reanimar.

Este punto no está resuelto en aquellos (escasos) sistemas que permiten el movimiento o uso de recursos entre distintos nodos, incluyendo las versiones en-cluster de SSOO tradicionales. La solución provisional adoptada pasa por *otorgar* crédulamente derechos especiales a aquellas peticiones procedentes de ciertos nodos o ciertos puertos.

En un DAMN son las aplicaciones las que tienen la ultima palabra en cuanto a autenticación y seguridad del sistema se refiere. Así, en *Off* la solución ha consistido en la definición de un portal de autenticación. Dicho portal está servido por una aplicación elegida por el usuario del nodo y permite aceptar o rechazar peticiones y reanimaciones de objetos remotos.

Cuando el núcleo congela un objeto suministra una firma junto con la imagen del mismo. Dicha firma puede comprobarse en cualquier otro nodo para corroborar la integridad del objeto. Esto puede

[9]El gestor de memoria distribuida (DMM) incorporado en *Off* también permite la implementación de persistencia de un modo casi trivial, como ya veremos.

combinarse con un algoritmo situado en el manejador del portal de autenticación para permitir que cada nodo pueda adoptar una política concreta y segura sobre uso de recursos remotos.

2.4.2 Heterogeneidad

La heterogeneidad es importante en un sistema distribuido. Aún más si se permite que los recursos migren entre distintos nodos.

En principio, el modelo de DAMN no impone ninguna de las soluciones conocidas para resolver los problemas presentados por entornos heterogéneos. No obstante, *Off* propone una solución poco común: La eliminación de representaciones independientes de arquitectura.

Es cierto que las representaciones neutrales, de red o independientes de arquitectura son esenciales para interconectar nodos heterogéneos en la red, pero sólo si:

- existe un gran número de arquitecturas que no pueden predecirse con antelación y/o
- los distintos sistemas ocultan su representación interna de la información.

Ninguno de estos dos puntos es cierto en los sistemas objeto de estudio. Consiguientemente, *Off* exporta o congela los objetos de un modo dependiente de arquitectura. Dado que junto con la firma del objeto se suministra un identificador de arquitectura, nada impide al usuario traducir la representación de una arquitectura a otra.

Nada, ¡salvo la integridad de la firma!

Un portal similar al de autenticación, el de *traducción*, permite a un núcleo de *Off* la obtención de una versión congelada válida la arquitectura local. Cuando un usuario intenta descongelar un objeto etiquetado como `i586PC` en un nodo con arquitectura `PA-RISC`, *Off* confía a una aplicación de usuario (en la que se confía) la traducción de dicho objeto congelado.

Dado que el núcleo ha comprobado ya la integridad de la imagen congelada, la traducción puede hacerse de forma segura en área de usuario.

2.5 Máquinas virtuales anidadas

Los servicios de *Off* están disponibles a los usuarios mediante portales, cada servicio del sistema utiliza un portal que ha de invocarse para utilizarlo. Esto trae consigo ciertos beneficios para aquellas aplicaciones que deseen utilizar máquinas virtuales anidadas (ej.: una máquina virtual de Java que opera sobre un emulador de Linux que a su vez opera sobre *Off*).

Dado que es posible interponer código (bien de un modo similar a como se hace en Kea [170], bien empleando los “Interposition Agents” descritos en [93]) entre el usuario y el servidor de un portal y dado que tanto el nombrado como la protección de recursos lógicos (ej.: abstracciones del SO tales como ficheros en Linux) lo implementan las aplicaciones, es posible cumplir las siguientes propiedades:

- Relatividad de los servicios del SO. Es decir, evitar puntos de acceso a servicio *absolutos*. Cada entidad puede (potencialmente) utilizar servicios distintos incluso cuando los identificadores de éstos son los mismos. Esto se podría conseguir con algo de código en la librería de usuario que hace de interfaz con el sistema de portales (bastaría con suministrar “descriptores” de portal similares a los descriptores de fichero en UNIX, de tal modo que pueda alterarse el identificador de un portal manteniendo el descriptor del mismo).
- Los procesos padre (ej. el emulador de Linux en el ejemplo anterior) pueden controlar directamente a los procesos hijos (ej. el emulador de Java) definiendo siempre “proceso” en términos

del SO que ejecuta sobre *Off*. No es difícil teniendo en consideración que *Off sólo* exporta recursos físicos y *no* fuerza la pertenencia de los recursos a una entidad concreta; esto es, no hay tareas, dominios de protección, contenedores de recursos ni otra abstracción similar.

- Todo el estado del kernel es legible por los procesos que lo usan. Concretamente, todo aquel que posea derechos de acceso sobre un objeto suministrado por el sistema puede leer directamente su estado directamente de la memoria física donde está almacenado. Por supuesto, estos derechos de lectura pueden pasarse de unas aplicaciones a otras a voluntad. Adicionalmente, cualquier información mantenida por el kernel que puede exportarse, sin poner en peligro ni la seguridad ni la privacidad, se exporta; esto incluye el estado de todos los portales y todas las listas de asignación de recursos (para interrupciones y excepciones, marcos de memoria, puertos, quanta de procesadores, portales, shuttles, DTLBs).

Con estas tres propiedades es posible implementar —tal y como dicen los autores de Fluke [71, 107]— un sistema de máquinas virtuales anidadas, donde la sobrecarga debida al anidamiento *no* aumenta exponencialmente. Esto abre nuevas posibilidades en implementaciones *eficientes* de sistemas globales (distribuidos en una red de área extensa) *altamente seguros* que utilizan un SO local sólo como medio para acceder a aquellos recursos físicos necesarios. Aplicaciones en Java [80, 79] y Limbo [52] son un ejemplo de estos sistemas.

Capítulo 3

La realización de *Off*

. . . Deep Hack Mode—that mysterious and frightening state of consciousness where Mortal Users fear to tread.

—Matt Welsh

En el capítulo anterior vimos las características de un DAMN y las líneas principales que determinan el diseño de *Off*. Ahora que ya debería estar claro el modelo de construcción de sistemas operativos distribuidos que estamos proponiendo, dedicaremos el presente capítulo a discutir la realización de *Off*, el prototipo de DAMN del que hemos estado hablando.

Además de apartados dedicados a los diversos aspectos de diseño e implementación del μkernel dedicaremos algunos apartados, a lo largo de la exposición, a la discusión de trabajos relacionados. Aunque bien podría pensarse que éstas estarían mejor encuadradas en el capítulo siguiente (dedicado a discutir otros sistemas que abordan los problemas que nos ocupan), nuestro propósito es establecer comparaciones puntuales, en aspectos concretos, entre *Off* y otros sistemas. De este modo, en el apartado dedicada a Shuttles encontramos párrafos donde se comparan éstos con otras abstracciones empleadas en gestión de procesos. Igualmente, encontraremos en el presente capítulo comparaciones puntuales de la IPC en *Off* con otros mecanismos de IPC, de la planificación en *Off* con otros esquemas de planificación, etc. Creemos que haciéndolo así se facilita la lectura y la comprensión del trabajo realizado. El capítulo siguiente estará entonces dedicado a comparaciones con otros sistemas con carácter general o global, sin entrar en aspectos concretos.

Hemos de mencionar que el código que aparece en el texto corresponde a la implementación actual del μkernel, omitiendo detalles del mismo que no son relevantes para su comprensión. En algunos casos hemos empleado incluso pseudo-código. Cualquier lector interesado puede dirigirse a [10] para obtener el código completo en formato electrónico o consultar [14] para obtener una descripción genérica de la arquitectura del sistema.

En lo que sigue, veremos primero aquellos aspectos generales de diseño e implementación que afectan al μkernel del sistema en su conjunto y, seguidamente, el diseño y realización de cada uno de los subsistemas que forman el μkernel.

3.1 La arquitectura del μkernel

El μkernel *Off* está formado por tres servidores básicos (ver figura 3.1): el servidor de shuttles, el servidor de portales y el gestor distribuido de memoria o DMM (“Distributed Memory Manager”). Dichos servidores implementan los shuttles, portales y DTLBs respectivamente.

Aunque estos tres determinan la arquitectura básica del sistema, el kernel incorpora otros servidores para implementar la asignación de recursos físicos: bancos de memoria física que asignan marcos de página, bancos de E/S que asignan puertos de E/S, bancos de DMA[1] que asignan líneas de DMA, etc.

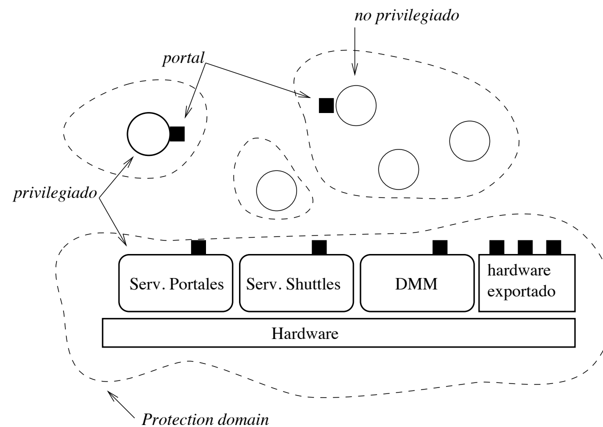

Figura 3.1: Arquitectura del sistema.

Estos otros servidores, empleados para asignar recursos físicos, son tan extremadamente simples que el presente capítulo versa casi en su totalidad sobre los tres servidores más complejos mencionados anteriormente.

Aplazaremos, no obstante, la discusión de los distintos servidores del μkernel para abordar primero algunos aspectos que afectan a todo el μkernel por igual. A tal fin dedicamos el resto del presente apartado.

3.1.1 Operación del sistema

El modelo adoptado por *Off* en cuanto a operación del sistema presenta dos características que modelan el comportamiento del sistema y la relación de éste con sus usuarios:

1. El μkernel se limita a suministrar recursos físicos distribuidos y tres abstracciones próximas al hardware distribuido en la red. La mayoría de los servicios suministrados a aplicaciones "tradicionales" (tales como "editores") por el SO sólo requieren llamadas entre entidades externas al μkernel (ej.: en el caso 1 de la figura 3.2 un editor llama a una biblioteca que implementa ciertos servicios del SO). Algunas de las llamadas de la aplicación al SO requerirán de llamadas al μkernel (ej.: en la figura 3.2, los casos 2 y 3).

2. En todos aquellos casos en que la distribución de recursos requiere de alguna política, protocolo o decisión de compromiso (localización de recursos, protocolos de coherencia, etc.), el μkernel se limita a efectuar llamadas a usuario (upcalls) para permitirle a éste implementar dichos protocolos y tomar sus propias decisiones (como en el caso 3 de la figura 3.2). La mayoría de las veces (caso 2 de la figura 3.2) las llamadas al μkernel del SO podrán ejecutarse inmediatamente. No obstante, en algunas ocasiones dichas llamadas no podrán completarse sin ayuda del usuario (el SO) y requerirán de llamadas adicionales (del μkernel al SO que lo usa).

[1] No incorporados en la implementación actual.

En realidad, no es que todos los "usuarios" deban implementar todas las operaciones invocadas por el μkernel mediante "upcalls", sino que cada usuario está dividido en dos partes: por un lado tenemos servicios del SO que operan directamente sobre el μkernel e implementan dichas llamadas; por otro lado tenemos aplicaciones típicas (editores, etc.) que emplearán los servicios del SO como en sistemas más tradicionales.

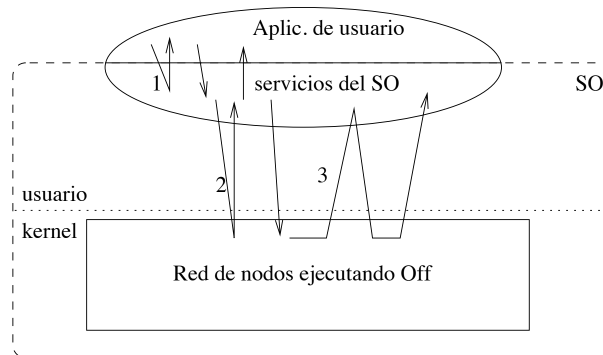

Figura 3.2: Modelo de operación del sistema en *Off*.

Off intenta suministrar un modelo de operación similar al que suministran las arquitecturas (hardware) que podemos encontrar en la actualidad. En este sentido, *Off* suministra algunas funciones a las aplicaciones (disponibles mediante llamadas al sistema) y en ciertas ocasiones envía eventos a dichas aplicaciones.

Al mencionar "evento" en el párrafo anterior nos referimos tanto a interrupciones como a excepciones y "upcalls". El SO implementado sobre *Off* ve una máquina virtual que dispone de las instrucciones suministradas por el hardware y de algunas otras implementadas por el μkernel. En ciertas ocasiones la ejecución normal se ve interrumpida por eventos (algunos procedentes de hardware, otros procedentes del μkernel). Como veremos en la sección 3.4.3 (donde discutimos el tratamiento de eventos en *Off* sobre Intel x86), *Off* sólo distingue entre dos tipos de eventos:

- Eventos síncronos, causados (o relacionados) con la ejecución de una determinada instrucción o llamada al sistema. Incluimos en éstos tanto traps, como excepciones y llamadas del μkernel al usuario puesto que la consideración de todos ellos como un solo tipo de evento simplifica el uso del sistema. Abusando del lenguaje, llamaremos a estos eventos *excepciones*.

- Eventos asíncronos, que no tienen relación directa con la ejecución del código del usuario. Incluimos en éstos interrupciones causadas por el hardware y mensajes asíncronos enviados por el μkernel o por los usuarios del sistema.

En ambos casos, los eventos se notifican mediante portales sean estos excepciones (traps, excepciones hardware y upcalls) o eventos asíncronos.

3.1.2 Dominios de protección

Los servidores del sistema están implementados como módulos de código que ejecutan con el procesador en modo kernel. Ésto impide que los usuarios accedan directamente a la implementación de los servicios del sistema (aunque si puedan leer su estado). Sólo mediante invocación de portales pueden las aplicaciones ejecutar código del μkernel.

En la implementación actual, los servidores del sistema están co-ubicados en un único dominio de protección, el dominio del μkernel, aunque nada impediría cargarlos en distintos dominios de protección (por ejemplo para probar nuevas implementaciones y encapsular aquellos servidores en cuya implementación no confiamos).

Como acabamos de mencionar, el acceso a los servicios del sistema se realiza mediante portales (ver figura 3.3). El servidor de portales es, por lo tanto, el único que incorpora realmente llamadas al sistema, implementadas éstas de forma tradicional mediante *traps*. Consecuentemente, cada servidor posee un portal (como puede verse en la figura 3.1) como único punto de acceso. Esto, que ya es práctica usual en los SSOO (por ej., en Mach [2]), permite que los usuarios ignoren el dominio de protección en que se encuentran realmente los servicios del sistema puesto que la invocación de portales independiza al usuario de los dominios de protección involucrados en una llamada. Como beneficio directo, en aquellas ocasiones en que todo un nodo se destina a un único fin (como en sistemas de tiempo real, o en nodos dedicados a ser únicamente servidores de Web, etc.) es posible emplear un único dominio de protección para aumentar el rendimiento. Incluso en tal caso, los usuarios verían los servicios del sistema como una serie de portales que pueden invocar.

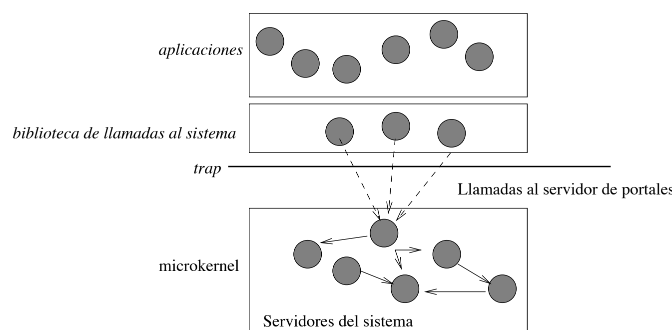

Figura 3.3: Llamadas al sistema.

Otro beneficio del empleo de portales es la habilidad para realizar interposiciones o redefiniciones de los servicios del sistema. Para acceder a un servicio del sistema cualquier usuario emplea el identificador de portal del servidor que los implementa. Estos identificadores los suministrará habitualmente el servicio de nombres implementado por el SO[2]. Si dicho servicio de nombres engaña a una aplicación interesada en el portal de determinado servicio del sistema y le da un identificador de portal diferente, ésta utilizará una implementación diferente de dicho servicio.Ésto se puede emplear para simular polimorfismo (emplear distintas implementaciones para suministrar el mismo servicio) en los servicios del sistema.

3.1.3 Almacenamiento de shuttles, portales y DTLBS

El servidor de Shuttles, el de portales y el DMM emplean versiones simplificadas del *Slab allocator* de Solaris [22] para almacenar las estructuras de datos correspondientes a Shuttles, Portales y DTLBs.

Un Slab en *Off* es tan sólo un vector capaz de crecer dinámicamente que sabe agrupar en una serie páginas de memoria virtual del μkernel estructuras de datos del mismo tamaño. A diferencia de un simple vector dinámico, un Slab sabe empaquetar los objetos por él asignados en páginas de memoria, evitando que un objeto atraviese un límite entre dos páginas, manteniendo en cada página objetos del mismo tamaño, etc. A pesar de que los Slabs empleados en la implementación actual son

[2]Ejecutando éste fuera del μkernel.

capaces de preinicializar[3] los objetos que contienen (véase [22]), la simplicidad de las estructuras de datos contenidas en ellos hicieron innecesario el empleo de esta habilidad.

Los servidores combinan el empleo de Slabs y la posibilidad de incluir alguna información extra en el campo `slot` de los identificadores de los objetos del sistema para optimizar la localización de los objetos que implementan. Cuando se crea un objeto y se obtiene almacenamiento en el Slab del servidor correspondiente, el campo *slot* de su identificador (ver apartado 2.1.7) se inicializa con el índice de dicho objeto en el Slab. De este modo, cuando se efectúa una petición al servidor que implementa el objeto, el servidor extrae del identificador del objeto el campo *slot* y lo utiliza como índice en el Slab. En todos aquellos casos en que el objeto considerado no ha migrado, la obtención de la estructura de datos a partir del identificador habrá sido extremadamente rápida: una indexación y una comparación.

Concretamente, los servidores del sistema que implementan recursos lógicos emplean la función `needs` mostrada en la figura 3.4. Dicha función tiene como propósito obtener la estructura de datos de un recurso lógico a partir de su identificador. Primero se indexa en el Slab como se mencionó en el párrafo anterior. Seguidamente se comprueba que el identificador del objeto almacenado en el Slab corresponde con el del objeto deseado. En caso afirmativo tenemos el identificador del objeto. En caso negativo deberemos emplear una tabla hash de relocalización (que básicamente permite encontrar aquellos objetos que, aún estando en el Slab, no se encuentran en la posición indicada por el campo `slot` de su identificador) y posiblemente sea preciso enviar excepciones a la aplicación, tal y como se ve en el código de la función.

Por último, hay un detalle que conviene considerar. Para evitar que una aplicación solicitando un excesivo número de shuttles, portales o DTLBs agote los recursos del sistema, los Slabs empleados emplean memoria virtual del μkernel que es posible paginar[4]. Cuando una aplicación crea un número excesivo de recursos lógicos sencillamente aumentará la probabilidad de que la escojan para expulsión de algunas de las páginas que emplea (debido a que también se consideran suyas las páginas que utiliza el μkernel para almacenar las estructuras de datos de los recursos suministrados por el μkernel a la aplicación). Dicho de otro modo, a pesar de que shuttles, portales y DTLBs están dentro del kernel y no son modificables directamente por las aplicaciones, la memoria que ocupan estos recursos está registrada a nombre de las aplicaciones que los crearon (y no está asignada al kernel, aunque sea éste el único capaz de escribir dicha memoria). En la figura 3.5 puede verse un ejemplo donde la parte del Slab empleado para portales está ausente debido al número excesivo de portales empleado por varias aplicaciones.

3.2 Recursos físicos

Cada recurso físico (bancos de memoria, bancos de puertos de E/S, bancos de líneas de DMA, etc.) se divide en unidades elementales (marcos de página, puertos de E/S, líneas de DMA, etc.) que son las unidades mínimas asignables a las aplicaciones. Estos “bancos” o recursos físicos constituyen las mínimas unidades reemplazables, como se mencionó en el apartado 2.1.2.

Para cada uno de ellos el μkernel incorpora un pequeño servidor, exportado a las aplicaciones mediante un portal, cuya utilidad es el procesamiento de peticiones de asignación o liberación de las unidades elementales que contienen.

[3] Una de las características del Slab incorporado en Solaris consiste en la delegación de la inicialización de los objetos contenidos en el Slab, de tal modo que no es preciso inicializarlos explícitamente. El Slab realiza la inicialización de dichos objetos sólo cuando es estrictamente necesario y libera al resto del kernel de tal responsabilidad.

[4] Esto no se hace por razones de simplicidad en la implementación actual.

```
slab_element_t *off_needs(slab_t *slab, hash_t *relocation_tbl, id_t id,
                          prtl_id_t ex)
{
   slot=id_slot(id);                      /* extract slot field from id */
   obj=(slab_so_nd_t*)slab_nd(slab,slot); /* (1)  obj=slab[id.slot]  */
   if (id_eq(id,obj->id))                 /* (2)  */
     return obj;                          /* object found */
   else {
       if (id found in relocation_tbl)
         send relocation exception for id to ex;
         slot=id_slot(id);
         obj=(slab_so_nd_t*)slab_nd(slab,slot);
         if (id_eq(id,obj->id))
               return obj;
         else  return NULL;
       else
         send missing exception for id to ex;
         slot=id_slot(id);
         obj=(slab_so_nd_t*)slab_nd(slab,slot);
         if (id_eq(id,obj->id))
               return obj;
         else  return NULL;
   }
}
```

Figura 3.4: Obtención del descriptor de un objeto a partir de su identificador.

Dado que los recursos físicos son conocidos en tiempo de arranque del sistema[5], la implementación de estos pequeños servidores es muy simple. Por cada recurso hay un vector creado en tiempo de arranque cuya dimensión corresponde al número de unidades elementales del recurso considerado. Una lista de unidades libres se emplea para localizar rápidamente alguna unidad libre susceptible de ser asignada.

La simplicidad que presentan estos servidores de recursos físicos en comparación con los tres servidores del sistema que implementan shuttles, portales y DTLBs no debería llevar a engaño: la actividad principal del kernel es la de asignar estos recursos físicos a las aplicaciones, aunque el porcentaje de código del μkernel dedicado a ello sea reducido.

El nombrado y protección de los recursos y unidades elementales que contienen corresponde a lo expuesto en el capítulo anterior, por lo que no repetiremos aquí lo ya dicho en este sentido.

Otra característica de estos servidores es que todos ellos presentan una estructura muy similar. Por lo tanto, detallaremos tan sólo la realización de uno de ellos (el gestor de memoria física), cuya función es la asignación de marcos de página a las aplicaciones.

3.2.1 Recursos físicos en otros sistemas

En realidad, salvo en el caso del exokernel [62] (y tal vez del Cache Kernel [34], si no somos demasiado estrictos), los SSOO no suministran recursos físicos directamente a las aplicaciones.

[5]Una vez concluido el trabajo de la presente tesis y el presente manuscrito, hemos visto que esta aseveración es un error. En el apartado 5.1.1 se hablará de este problema.

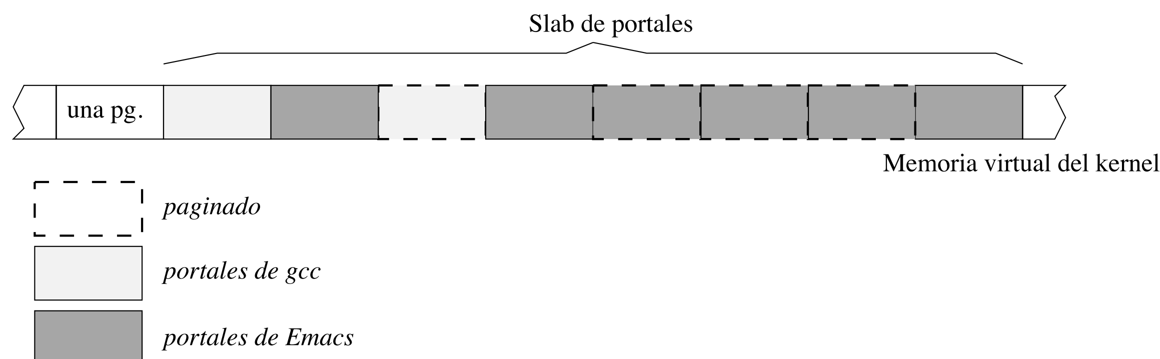

Figura 3.5: Los Slabs pueden paginarse.

En el caso del exokernel los recursos se exportan a las aplicaciones del mismo modo que en *Off*. No obstante, en dicho sistema, estos recursos pierden validez fuera del nodo en que se encuentran, cosa que no ocurre en *Off*.

Una diferencia radical entre la gestión de recursos físicos en *Off* y en sistemas tradicionales estriba en que dichos recursos están asignados al núcleo en su totalidad en éstos últimos. En *Off* el núcleo no tiene más derecho sobre los recursos físicos que una cualquiera de las aplicaciones.

3.2.2 El gestor de memoria física

Los puntos de entrada del servidor que exporta memoria física (el PMM[6]) tienen como función principal la asignación de marcos de página (que pueden emplearse posteriormente en la instalación de traducciones de memoria) como puede verse en la figura 3.6.

```
int pmm_pg_alloc(prtl_id_t who, paddr_t maddr[], int n,
                 pmm_flags_t flags, off_guty_t *guty);

int pmm_pg_free(prtl_id_t who, paddr_t maddr[], int n,
                off_guty_t *guty);

int pmm_pg_set_rights(paddr_t maddr[], int n, off_guty_t *to,
                      off_guty_t *guty, u_char ops);
```

Figura 3.6: Puntos de entrada del gestor de memoria fisica.

Podemos ver cómo hay dos funciones que tienen nombres terminados en `alloc` y `free`. Dichas funciones son las encargadas de asignar y liberar las unidades elementales de memoria física—marcos de página. Funciones similares aparecen en todos los servidores que exportan recursos físicos (nótese que los procesadores y el hardware de traducción de direcciones están encapsulados en el servidor de shuttles y el DMM respectivamente, por lo que su tratamiento es diferente y no deberían considerarse en la presente discusión).

En las funciones de asignación y liberación de recursos podemos apreciar cómo se permite la agrupación de llamadas al sistema. Concretamente, ambas funciones trabajan con vectores de marcos

[6] Physical Memory Manager

de página (parámetros maddr y n). Con ello se trata de reducir el número de llamadas al sistema ya que las aplicaciones operan típicamente con conjuntos de unidades elementales.

Centrándonos ahora en la asignación, la llamada que implementa ésta tiene como parámetro de entrada un portal que identifica al responsable del recurso hasta el momento de su liberación, who. Cualquier excepción relacionada con dicho recurso (ej.: la notificación de que el recurso no se está utilizando) se enviará a dicho portal. La función de liberación recibe también un parámetro con el mismo propósito.

La protección del recurso asignado viene determinada por la información de protección establecida por el usuario en el instante de la asignación mediante un parámetro adicional, guty (*guardability*).

Dado que todas las unidades de recurso están protegidas, cada servidor de recursos físicos incorpora una función para cambiar la protección de las mismas, como puede verse también en la figura 3.6.

Una vez asignados, los recursos físicos pueden ser utilizados por cualquiera que posea los derechos de acceso adecuados. El caso más delicado (y por ello el expuesto como modelo de servidor de recursos físicos) es el de la memoria física. La protección de un marco en un instante determinado no hace referencia a la posibilidad de leerlo o escribirlo en dicho instante. Hace referencia a la posibilidad de *instalar* traducciones de lectura o escritura en dicho instante. Recordemos que hablamos de recursos físicos y no de abstracciones; las protecciones se refieren, pues, a las operaciones realizables sobre dichos recursos y no a las operaciones de alto nivel que imaginan las aplicaciones.

Así, para revocar derechos de acceso sobre un marco de página con carácter *retroactivo* hay que invalidar también cualquier traducción existente hacia dicho marco. La información de asignación que exporta el kernel permite averiguar dónde están dichas traducciones.

Volviendo al código de la figura 3.6, puede apreciarse como los marcos de página están nombrados empleando el tipo de datos paddr_t. Es importante mencionar que dicho tipo corresponde a lo descrito en el capítulo anterior sobre nombrado de unidades elementales de recursos físicos en un DAMN. En la figura 3.7 puede verse como el nombre de un marco extiende el nombre físico del mismo con un identificador de portal (que corresponde al portal del servidor que gestiona el banco de memoria física, en este caso).

```
typedef vm_offset_t vaddr_t;            /* Virtual memory address  */

/* Physical memory address */
typedef struct paddr_st {
     prtl_id_t p;                       /* portal */
     vm_offset_t offset;                /* offset in that pmm */
} paddr_t;
```

Figura 3.7: Nombre de un marco de página

En cualquier operación referida a un marco de página, el μkernel redirigirá (elevando excepciones) las peticiones destinadas a marcos de página remotos (ya que descubre que los portales de sus identificadores son remotos). De este modo, los usuarios pueden emplear marcos cuya ubicación sea remota. En el apartado 3.5 veremos que también es posible emplear marcos de página remotos en las traducciones de memoria virtual a memoria física empleadas por las aplicaciones.

3.3 Gestión de Procesos en Off: Shuttles

Una de las actividades principales de un SO es la gestión de procesos. Consiste ésta en el conjunto de tareas y utilidades necesarias para controlar la actividad y la ejecución de las distintas aplicaciones, así como también —utilizando "gestión de procesos" en un sentido más amplio— para asignar los recursos del sistema a las mismas y contabilizar dicha asignación.

Hasta el momento, dos han sido típicamente las principales abstracciones empleadas (como veremos en el siguiente apartado) para suministrar servicios de gestión de procesos en SSOO tradicionales: las *tareas* o procesos y los *threads* o hilos de control. Las tareas contienen los recursos asignados y uno o más hilos de control.

Estas abstracciones no son ni extensibles ni adaptables, lo que da lugar siempre que es necesario incluir nuevas características o modificar el comportamiento de dichas abstracciones, a la reimplementación del sistema de gestión de procesos (con la consiguiente ineficiencia y el consiguiente perjuicio a la fiabilidad).

Por otro lado, en SSOO distribuidos la gestión de tareas y threads o bien no se distribuye en absoluto (como en Plan 9, donde un proceso está sujeto al nodo en que nació y su gestión es muy similar a la de un sistema centralizado), o bien se distribuye *sólo* del modo en que el arquitecto del sistema estimó conveniente.

En *Off* se utiliza el *Shuttle* como base para posibles implementaciones de abstracciones similares a "procesos" o "tareas" tradicionales en área de usuario—aunque bien pueden utilizarse directamente los shuttles sin necesidad de implementar abstracción alguna. La abstracción *Shuttle* trata de evitar las desventajas mencionadas constituyendo una abstracción de relativamente bajo nivel y contemplando la distribución del sistema.

En pocas palabras podríamos decir que el *servidor de Shuttles* de *Off* es un sistema adaptable y distribuido de gestión de procesos que permite a cada aplicación modelar la distribución de los procesos del sistema del modo más conveniente, a la vez que ofrece servicios básicos de distribución a aquellas aplicaciones que no deseen controlarla explícitamente.

En el presente apartado nos centraremos en el diseño e implementación del servidor de Shuttles, que implementa dicha abstracción. Antes de ello veremos brevemente qué alternativas existen en otros sistemas.

3.3.1 Otros enfoques en gestión de procesos

Procesos y tareas

Aunque las abstracciones más empleadas en gestión de procesos son las *tareas* (o procesos) y los *threads*, existen en realidad multitud de abstracciones empleadas con mayor o menor éxito en los SSOO actuales para el mismo fin. En cualquier caso, todas ellas siguen girando alrededor de los conceptos de tarea o proceso (en adelante utilizaremos sólo tarea, aunque puede leerse también "proceso") y de thread:

- Una *tarea* [158] representa una aplicación activa y está constituida por un dominio de protección y uno o varios flujos de control ejecutando en el mismo. La tarea es también el *contenedor* de los recursos utilizados por dichos flujos de control. Tanto las *tareas* de Mach [2] como los *procesos* de las diferentes versiones UNIX [78] son ejemplos de la misma.

- Un *thread* [158] es un flujo de control que ejecuta en un procesador *dentro* de un dominio de protección dado. Típicamente tanto el dominio de protección como los demás recursos necesarios para la ejecución del thread son suministrados por la tarea en la que se ubica el

mismo. Ejemplos son los threads de kernel en versiones modernas de UNIX [81] y los threads de Mach [2].

Como hemos dicho anteriormente, uno de los problemas que presentan dichas abstracciones es su falta de adaptabilidad. Cada vez que una aplicación necesita algo diferente, estas abstracciones deben reimplementarse y es necesario efectuar cambios importantes en el núcleo del SO. Esto ha ocurrido con los threads móviles en Mach [73] y con muchos otros servicios añadidos a la gestión de procesos de SSOO contemporáneos. En muchos casos ha sido necesario implementar un SO completamente nuevo debido a la incapacidad de soportar nuevas habilidades de las abstracciones existentes, como ocurrió con GrassHopper [50]. Por último, esta falta de adaptabilidad y la complejidad que requiere la implementación de tareas y threads dentro de un kernel conducen a ineficiencias y faltas de fiabilidad como se indicó en los capítulos anteriores.

Cuando el sistema implementa múltiples threads dentro de las tareas, la implementación se hace pesada y compleja. Se introducen con toda seguridad más errores en el sistema (puesto que el número de errores es proporcional a la complejidad y tamaño del código), perjudicando a aquellas aplicaciones que no utilizan los servicios añadidos (a las que sólo usan un thread, a las que sólo usan un dominio de protección, a las que emplearían un esquema de planificación estático de estar éste disponible, etc).

Por otro lado el sistema debe optar bien por planificar de forma simétrica los threads, bien por realizar una doble planificación (primero tareas y luego threads), bien por un esquema mixto. Sea cual sea el modelo empleado se aplicará a *todas* las aplicaciones incluso si la planificación resultante es peor.

Consideremos ahora la distribución y la migración de procesos en sistemas tradicionales. Cuando las tareas contienen a los threads no es factible migrar un thread aislado sin migrar también a la tarea que lo contiene (por no mencionar que habitualmente tanto tareas como threads son incapaces de abandonar el nodo en que se crearon). Esto incluye *todos* los recursos de la tarea, dado que los threads tienen noción sólo de la tarea que los contiene y no de los recursos que necesitan.

Esto es una consecuencia de la vinculación estrecha de los recursos a las tareas, que fuerza la compartición de recursos dentro de la misma tarea. ¿Acaso no tiene utilidad la posibilidad de tener distintos threads con distintos identificadores de usuario ejecutando dentro de un servidor, cada uno de ellos sirviendo a un usuario distinto y, aun así, compartiendo ciertos recursos del servidor? ¿No tiene utilidad que distintos threads dentro de la misma tarea mantengan diversos derechos de acceso, niveles de privilegio, protecciones de memoria, tablas de ficheros abiertos, etc.? La lista de aplicaciones es casi ilimitada.

Para solventar los mencionados problemas, tres han sido las contribuciones más relevantes en cuanto a abstracciones dedicadas a gestión de procesos en los últimos tiempos:

1. La implementación de threads que cambian de tarea o "*threads móviles*[7]" [73, 84]. Dicho de otro modo, la distinción entre estado (los recursos necesarios para la ejecución del thread y el contexto del mismo) y flujo de control. Hablaremos de esto en el apartado 3.3.1.

2. La implementación de *continuaciones* [56]. Son éstas abstracciones que resumen el estado de un thread de tal modo que no es preciso mantener más estado sobre un thread que la *continuación* para poder reanudar la actividad del thread en cualquier momento. Típicamente están formadas por una función (que representa lo que resta de ejecutar en un thread) y una pequeña zona de almacenamiento (que representa el estado en que se encuentra el thread).

3. El diseño de abstracciones de muy bajo nivel correspondientes a *contextos hardware* [59, 134].

[7] En inglés, "Migrating threads".

Lo que nosotros proponemos como alternativa es el uso de una única abstracción consistente en un contexto de procesador básico que puede extenderse bajo demanda para incluir nuevos elementos. No obstante, cualquier abstracción cercana al hardware (al contexto del procesador en este caso) que pueda emplearse en un conjunto de nodos distribuidos en una red puede considerarse también como candidata para el sistema de gestión de procesos de un DAMN.

Estado y flujo de control

Siempre pensamos en un thread como una entidad que posee los siguientes elementos:

- Un estado que esencialmente consiste en (ver figura 3.8)
  - Un contexto de ejecución que incluye el conjunto de valores para los registros, considerando también el contador de programa y la pila empleada.
  - La capacidad de planificar dicho contexto o de iniciar y detener la ejecución de dicho contexto en un procesador.

  En la figura, el "estado" del thread correspondería a la pila y juego de registros empleados en un instante determinado.
- Un flujo de control que corresponde con una serie de registros de activación en una (o varias) pila(s) y una sucesión de valores para el contador de programa. En la figura, el flujo de control está formado por la sucesión de instrucciones ejecutadas y llamadas a procedimiento realizadas que representamos con líneas discontinuas.

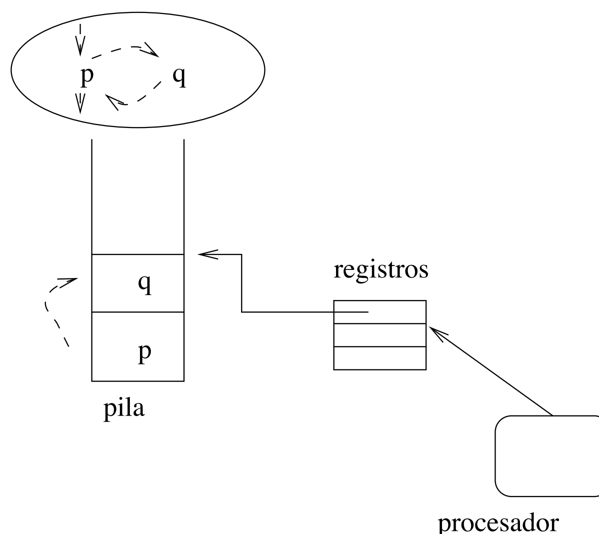

Figura 3.8: Threads: estado y flujo de control.

Sistemas como Spring han separado el concepto de estado del thread del de flujo de control (ver figura 3.9):

- Por un lado, Spring emplea una abstracción denominada *Shuttle* que representa el estado de la ejecución de un thread y es una entidad planificable (el planificador de Spring multiplexa los procesadores entre los Shuttles existentes). Podemos pensar en los shuttles como en "procesadores virtuales" que empleamos para ejecutar los flujos de control de las aplicaciones.

- Por otro lado, Spring incluye otra abstracción denominada *thread* para representar el flujo de control que involucra un conjunto de registros de activación correspondiente a la secuencia de llamadas a procedimiento efectuados. Este flujo de control es lo que los usuarios del sistema perciben como un thread, de ahí el nombre.

Nosotros hemos empleado la terminología de Spring por considerarla más clara, aunque otros sistemas que emplean nombres distintos para estas abstracciones (notablemente, una implementación de threads móviles para Mach [73] denomina *thread* al equivalente a un Shuttle en Spring y *activación* al equivalente a un thread en Spring).

En *Off*, los *shuttles* se conciben como contextos de procesador extensibles. Un shuttle en *Off* no es otra cosa que el estado necesario para ejecutar un thread. A diferencia de lo que sucede en Spring, los shuttles de *Off* pueden extenderse dinámicamente para incluir nuevos elementos en el "estado de ejecución" (hablaremos de esto en el apartado 3.3.2). El kernel se limita pues a crear, destruir y planificar shuttles. Como vemos en la figura 3.9, el procesador se multiplexa entre un conjunto de Shuttles que se emplean como abstracción básica para ejecutar threads.

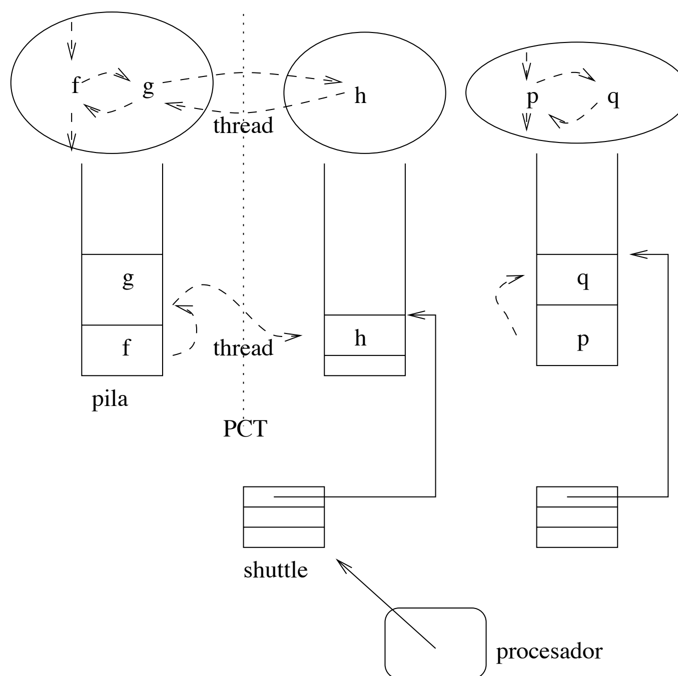

Figura 3.9: Flujos de control (threads) y estados (shuttles)

La distinción entre shuttles y threads permite que el sistema opere con contextos de procesador (Shuttles) a nivel de planificación y gestión de procesos y, por otro lado, los usuarios perciban una abstracción distinta correspondiente a un flujo de control (thread).

Lamentablemente, a pesar de la introducción de shuttles en Spring, los threads de Spring están dentro del núcleo y son de hecho la única abstracción exportada a los usuarios (los shuttles son una abstracción interna del núcleo que no es manipulable directamente por los usuarios del sistema). Los shuttles de *Off*, al contrario, dejan abierta la posibilidad de emplear otras abstracciones a nivel de usuario. El μkernel no los oculta en absoluto.

Como consecuencia de la ocultación, la planificación de los procesadores y otros aspectos relacionados con la implementación de los Shuttles en Spring están completamente determinados por el μkernel. En cambio *Off* deja estas tareas en la medida de lo posible en manos de las aplicaciones.

Trasferencias protegidas de control

Una consecuencia práctica de la disponibilidad de shuttles es la posibilidad de utilizar el shuttle del cliente, en una llamada a procedimiento remoto (o RPC[8]), para ejecutar sobre él el thread del servidor. Ésto no es otra cosa que un "modelo de objetos pasivo", donde clientes y servidores son un conjunto de datos a través de los cuales navegan flujos de control (shuttles en este caso). Al mecanismo en virtud del cual es posible *transferir* el shuttle del cliente al servidor en el transcurso de una RPC se le denomina *transferencia protegida de control* (o PCT[9]).

En la figura 3.9, el shuttle de la izquierda realiza una PCT para transferir el flujo de control del cliente al servidor y viceversa (en la RPC que *g* realiza a *h*). Como puede apreciarse en dicha figura, se emplea un único shuttle aunque estemos efectuando una RPC.

El beneficio de las PCTs radica en el ahorro que supone no cambiar de Shuttle durante una RPC: no es necesario replanificar y la latencia de la RPC (tal y como la percibe el usuario) disminuye debido a que también eliminamos operaciones de sincronización entre el cliente y el servidor.

Contextos hardware

Sistemas más avanzados, como el Cache Kernel [34] y el exokernel [62] optan en cambio por suministrar sólo los servicios suministrados por el hardware: contextos de procesador hardware. Esto permite solventar la falta de adaptabilidad que padecen las abstracciones mencionadas en apartados anteriores.

El problema que presentan los contextos hardware consiste en que no se contempla la distribución de los procesadores en la red (cosa natural puesto que tanto el Cache Kernel como el exokernel son núcleos centralizados).

En *Off* hemos optado por *extender* el "contexto hardware" hasta un punto en que es posible utilizar los contextos hardware de forma distribuida. El método empleado ha consistido en abstraer el concepto de contexto hardware, como veremos en el siguiente apartado.

3.3.2 Los shuttles de *Off*

En *Off*, un shuttle es un contexto hardware extensible. Inicialmente contiene tan sólo un conjunto de registros de propósito general (incluyendo un contador de programa y un puntero a pila). Posteriormente puede extenderse de forma segura para incluir otras partes del contexto no presentes inicialmente: identificadores de espacios de direcciones, niveles de privilegio, mapas de permisos de E/S, etc. A estas *extensiones* las denominamos *propiedades*.

Por ejemplo, en la figura 3.10 tendríamos las propiedades "A", "B" y "C", donde "A" y "B" corresponden con tipos de recurso que implementan ciertos módulos dentro del μkernel y "C" corresponde con un tipo de recurso implementado o suministrado por una entidad de nivel de aplicación. Como puede verse en dicha figura, la asociación de shuttles a recursos se produce únicamente a través del nombre de éstos, manteniendo la modularidad del sistema en cuanto a la interrelación de Shuttles y gestores de recursos se refiere: los shuttles sólo saben qué recursos necesitan (por ejemplo, el de la parte izquierda de la figura necesita "i" de tipo "A", "j" de tipo "B" y "k" de tipo "C"), y su implementación está completamente aislada de la implementación de estos tipos de recurso. Discutiremos más sobre las propiedades en el apartado 3.3.3.

Off no implementa tareas ni otras abstracciones equivalentes.

Naturalmente, un Shuttle necesita en la mayoría de los casos—aunque no siempre—algunos recursos además de los registros para poder ejecutar. Ahora bien, dado que estos recursos no necesitan

[8] Del inglés Remote Procedure Call.
[9] Del inglés, Protected Control Transfer.

estar contenidos en ninguna abstracción que pudiera actuar de *contenedor* de recursos los shuttles no necesitan estar contenidos en otra abstracción, al contrario que en otros sistemas donde los "threads de kernel" se consideran siempre incluidos en una tarea que incorpora los recursos necesarios.

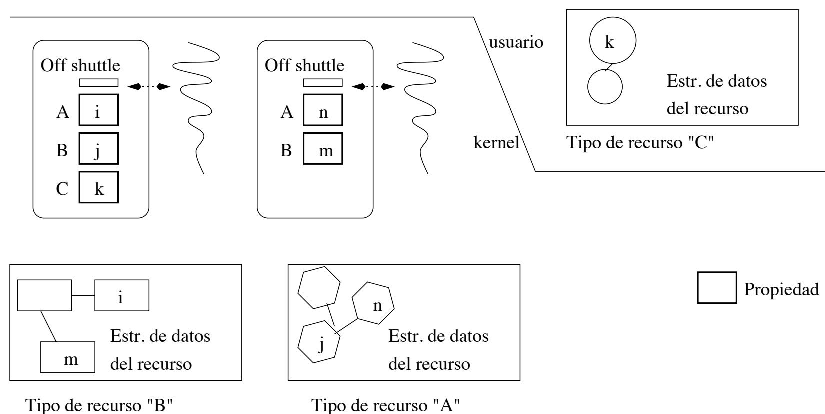

Figura 3.10: Los Shuttles de *Off* y los recursos utilizados.

Esta es una de las diferencias entre los shuttles de *Off* (figura 3.10) y los threads de otros sistemas (figura 3.11). En *Off*, los Shuttles sólo *saben* qué recursos necesitan para ejecutar y no se ocupan en absoluto del mantenimiento del estado de los mismos; tampoco de la implementación de sus operaciones (de ésto se encarga el implementador o gestor de dicho tipo de recurso). Complementariamente, los recursos empleados por los shuttles no están asignados o contenidos en ningún Shuttle. Al contrario, en sistemas como Mach los threads están contenidos en tareas que a su vez contienen los recursos necesarios para la ejecución de aquellos. Así, en la figura 3.11 vemos cómo las líneas que unen los threads con las implementaciones de los recursos utilizados representan la interrelación entre ambos, con lo que tenemos menos modularidad en la implementación del μkernel. Una situación como la que representa la figura 3.11 en la que distintos threads dentro de una misma tarea de Mach tienen distintos conjuntos de recursos (el de la izquierda de la figura utiliza tres, y el de la derecha sólo dos) no es realmente implementable dado que todos los threads de la misma tarea deben compartir todos los recursos disponibles dentro de la tarea.

Hay una importante diferencia entre la *asociación* de un Shuttle a un recurso y el esquema utilizado por otros sistemas para el mismo propósito: los shuttles *no* conocen ni cuantos recursos diferentes hay ni como están estos implementados (lo que también supone que el conjunto de recursos a emplear por un shuttle no está preestablecido y puede alterarse a voluntad).

La asociación de recursos a las abstracciones empleadas para modelar la "computación activa" o el flujo de control de las aplicaciones ya existía en cierta medida en algunos sistemas desde hace algún tiempo. Así, Ra [7] permite que sus "threads" (llamados Isibas en Ra) se asocien a espacios de direcciones de tal modo que un Isiba puede cambiar de dominio de protección a lo largo de su vida. Sprite [128] emplea un conjunto de módulos dentro del kernel de tal modo que el estado de uno de sus procesos está compartimentado: cada módulo (p.ej.: el sistema de ficheros) mantiene una parte del estado del proceso con independencia del estado mantenido por otros módulos. Esta arquitectura modular facilita la construcción de sistemas de migración de procesos al modularizar[10] el empaquetamiento del estado de dichos procesos [54].

En *Off* se ha empleado este modelo de *asociación a recursos* de forma intensiva, generalizándose éste mediante la introducción de propiedades que permiten a los shuttles asociarse a *todos* y cada uno

[10] Basta con que cada módulo empaquete su parte del estado.

de los recursos necesitados.

De este modo, cada gestor de recursos puede implementar los recursos que suministra del modo que desee y los cambios o adaptaciones en dicha implementación no afectan en absoluto al sistema de Shuttles de *Off*. Ésto no ocurre con tareas y threads, donde ambas conocen en mayor o menor medida qué gestores están implementando qué servicios e incluso, muchas veces, cómo los están implementando.

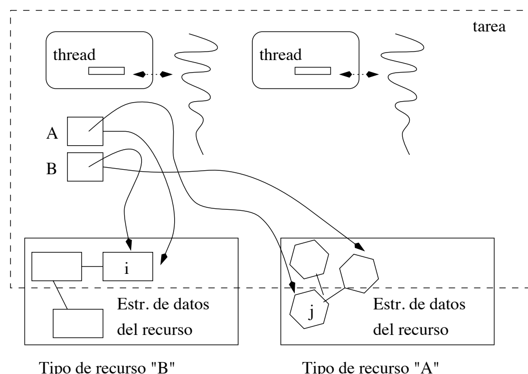

Figura 3.11: Tareas, threads y recursos utilizados en sistemas tradicionales.

Esta simple abstracción, el Shuttle, también permite a la aplicación implementar la migración y replicación de las abstracciones construidas sobre los shuttles de un modo sencillo mediante las funciones *freeze* y *melt*, como se indicó en el capítulo anterior. Si los recursos necesarios para la ejecución de los shuttles mantienen su disponibilidad en otros nodos y/o son capaces de migrar y replicarse no hay ningún obstáculo a la distribución y migración de abstracciones construidas sobre los Shuttles. Nótese que los recursos del sistema están disponibles en toda la red, por lo que la única limitación la impondrán los recursos lógicos implementados por aplicaciones sobre el μkernel.

Por último, un breve comentario sobre sistemas heterogéneos. Los shuttles de *Off* no tratan de solucionar el problema de la heterogeneidad. En este sentido no incorporan técnica alguna para su operación en sistemas heterogéneos. Si se desea que un mismo shuttle pueda ejecutar en distintas arquitecturas basta con emplear una máquina virtual o alguna otra técnica conocida.

Como vimos en el apartado 2.4.2, los sistemas heterogéneos están contemplados en el diseño de *Off* aunque es cierto que la solución al problema de la operación distribuida en dichos sistemas la aportan principalmente las aplicaciones del sistema.

3.3.3 Propiedades

Los distintos tipos de recursos que pueden asociarse a un Shuttle se denominan *propiedades* de shuttles en *Off*. Candidatos a propiedades son, pues, espacios de direcciones, niveles de privilegio, etc. Las propiedades tienen valores de tal modo que el valor de una propiedad identifica un recurso (o valor) de un tipo de recurso (o propiedad) necesario para la ejecución de un Shuttle. Así, el espacio de direcciones en que ejecuta un Shuttle viene identificado por el valor de la propiedad "virtual address space". En lo que respecta a la implementación de los shuttles, una propiedad es, en efecto, el conjunto de valores posibles que puede adoptar. Podría pensarse que las propiedades son *registros virtuales* que extienden el estado del procesador.

En tiempo de arranque, existen algunas propiedades que siempre define el sistema. Otras sólo se definen opcionalmente, también en tiempo de arranque. En cualquier caso los usuarios pueden definir aún más una vez completo el arranque del sistema.

Las propiedades definidas por el sistema son:

- *Contexto hardware*. Siempre se define y representa el estado del procesador en una arquitectura determinada.

- *Estado de ejecución*. Se define siempre que *Off* se compila para soportar multiproceso[11]. Indica si el shuttle está preparado para ejecutar o, por el contrario no debe hacerlo (debido a razones conocidas por el usuario que utiliza dicho shuttle como, por ejemplo, debido a que el usuario está alterando el contenido de los registros de procesador del shuttle,...). También especifica si la ejecución del shuttle debería efectuarse en modo de traza. Nótese que cuando hablamos de estado de ejecución nos referimos únicamente a cómo deben los procesadores considerar el shuttle de cara a su posible ejecución; no hablamos de estados de planificación (listo, bloqueado, ejecutando, expulsado de memoria, etc.) que corresponderían a las abstracciones implementadas por el SO que ejecuta sobre *Off*.

- *Portal de excepción*. Esta propiedad (definida siempre) indica a qué portal deben enviarse las excepciones sin manejar que cause la ejecución del shuttle.

- *Espacio de direcciones*. En aquellos nodos con soporte para espacios de direcciones (ésto es, todos en la implementación actual) esta propiedad identifica la DTLB en la que opera el shuttle.

- *Espacio de E/S*. En aquellos nodos con soporte para espacios de E/S[12] esta propiedad identifica el espacio de E/S (conjunto de puertos en que puede hacerse E/S) en el que opera el shuttle.

Los usuarios son libres de definir dinámicamente, en cualquier momento de la vida del sistema, otras propiedades cualesquiera como podrían ser:

- Nivel de privilegio. Identificando el nivel de privilegio al que debe quedar el procesador para ejecutar el shuttle afectado.

- Dominio de protección. Identificando un conjunto de derechos de acceso a otros elementos (por ejemplo, un conjunto de capabilities).

- Identidad de usuario y/o grupo(s). Determinando la identidad del responsable de la computación efectuada.

- Contexto virtualizado (en caso de usar una máquina virtual para resolver problemas de heterogeneidad). Identificando un estado de máquina virtual correspondiente al estado existente en nativo (el contexto hardware podría en este caso corresponder al contexto de ejecución del intérprete de la máquina virtual).

Para definir una propiedad, tanto si la define el sistema como si la define un usuario, es preciso suministrar dos elementos:

1. Una función de conmutación, que será invocada siempre que, por un cambio de contexto, sea preciso alterar el valor de la misma en el procesador.

2. Un valor por defecto. Que se empleará como valor en todos aquellos shuttles que no presenten dicha propiedad.

[11] Siempre, en la implementación actual.
[12] Todos, en la implementación actual.

El servidor de Shuttles de *Off* invocará a la función de conmutación siempre que sea necesario preparar para su uso el recurso identificado por el valor de la propiedad (a modo de ejemplo, la función de conmutación de la propiedad "espacio de direcciones" instalará la correspondiente DTLB en el hardware).

Consecuentemente, cuando una conmutación de shuttles tiene lugar en un procesador (ver apartado 3.3.5) se ejecuta un algoritmo como el que sigue.

- Para cada propiedad definida:
  - Si el valor de dicha propiedad en el shuttle que ejecutaba anteriormente difiere del valor para el shuttle que va a ejecutar a continuación y dicha propiedad tiene definida una función de conmutación:
    - Se invoca a la función de conmutación indicando como argumentos la pareja de valores[13] (antiguo,nuevo).

En los casos en que no es necesario preparar un recurso para su uso la función de conmutación puede quedar sin definir. Por supuesto, la información de los valores de la propiedad puede obtenerse siempre que sea necesario.

3.3.4 Implementación en el Intel x86

En este apartado veremos cómo están implementados los Shuttles en el prototipo realizado. Primero daremos un brevísimo repaso a las facilidades suministradas por la arquitectura utilizada y posteriormente describiremos tanto la implementación de los Shuttles como la de sus propiedades.

Nótese que en lo que sigue empleamos el término "tarea" para referirnos a las entidades *ejecutables* implementadas por el Intel x86 y no para referirnos a la abstracción *tarea* que corresponde con la noción de proceso.

Tareas en el Intel x86

En el Intel x86 el procesador tiene una serie de registros entre los que se encuentran los siguientes:

- **eip** Contador de programa
- **esp** Puntero de pila
- **eflags** Flags de estado del procesador.
- **tr** Registro de tarea.
- Registros generales como eax, ebx, etc.

Cada tarea que ejecuta en el procesador tiene a su vez un segmento de estado de tarea (TSS, de *task state segment*) que contiene, entre otros, los siguientes elementos:

- **Espacio de salvaguarda de registros** destinado al almacenamiento del estado del procesador cuando se deja de ejecutar la tarea (debido a una conmutación de tarea). También se emplea para recuperar dicho estado cuando es preciso reanudar la tarea.

[13]En realidad bastaría con suministrar el nuevo valor, pero conceptualmente se está reemplazando un valor antiguo con otro nuevo y el suministro de ambos valores en la invocación de la función de conmutación evita en ciertos casos el mantenimiento de estado en el implementador del tipo de recurso considerado.

- **Mapa de bits de E/S** que contiene un bit por cada puerto indicando si se permite ejecución de una operación de E/S en el mismo o bien se requiere que el procesador ejecute en modo privilegiado.

- **Punteros de pila** para los distintos niveles de privilegio (de 0 a 3 en el Intel). En una conmutación de tarea el hardware sitúa el puntero de pila (*esp*) con un valor almacenado en el TSS, dependiendo del nivel de privilegio con que se ejecuta dicha tarea (*esp0* para el nivel 0, *esp1* para nivel 1, etc.).

El procesador contiene un registro denominado "registro de tarea" (o *tr*) que identifica el TSS de la tarea en curso. Cuando se conmute hacia otra tarea el hardware salvará el estado del procesador en el TSS apuntado por *tr* y cargará en *tr* el valor para el nuevo TSS, recuperando el estado del procesador de la salvaguarda de registros mantenida en el TSS de la tarea entrante.

Así pues, el esquema completo quedaría como puede apreciarse en la figura 3.12. Hay otros muchos elementos que no hemos mencionado por razones de claridad y brevedad.

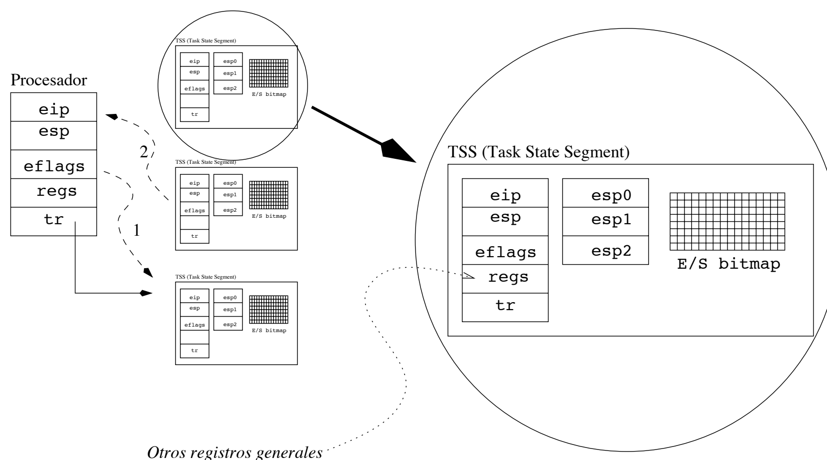

Figura 3.12: Tareas en el Intel x86. En un cambio de contexto el hardware (1) salva y (2) recupera el estado.

Veamos ahora cómo están implementadas las propiedades y los shuttles.

Implementación de las Propiedades

La implementación de las propiedades es bastante simple, basta con emplear un vector de definición de propiedades y extender cada uno de los shuttles con un vector de valores para cada una de las propiedades definidas. Cada shuttle mantiene (en un campo denominado *pmask*) una referencia al vector de valores para las propiedades definidas.

Dado que estas propiedades corresponden a elementos del contexto gestionado por el μkernel—el resto de elementos del contexto correspondientes a abstracciones de usuario estarán gestionados por completo en área de usuario—, no es necesario tener grandes vectores de propiedades. Al contrario, en nuestra corta experiencia con el prototipo implementado, la flexibilidad del modelo de shuttles ha hecho innecesaria la definición de nuevas propiedades salvo en lo referente a identidades de usuario y dominios de protección (contenedores de recursos en área de usuario).

Las información sobre las propiedades definidas se mantiene en el *vector de definición de propiedades* que contiene una posición para cada propiedad definida. En cada entrada se mantienen los siguientes elementos:

- El valor por defecto, *dflt*, a emplear en aquellos shuttles que no especifican valor alguno para ella (esto es, que ya existían antes de definirse la propiedad o que la ignoran).

- La función de conmutación de la propiedad, *p*, empleada para instalar un nuevo valor.

- Un campo de flags que permite distinguir entre propiedades definidas y no definidas.

Nótese como este modelo *aisla* por completo al sistema de shuttles de *Off* de las implementaciones particulares empleadas para suministrar recursos tradicionalmente mezclados con la gestión de procesos, tales como espacios de direcciones, espacios de E/S, identidades de usuario, privilegios, etc.

En la implementación actual, cada propiedad definida tiene un valor (posiblemente, el valor por defecto) para cada shuttle del sistema. Esto implica que la lista de valores de propiedades que cada shuttle posee corresponde a la lista de propiedades definidas en un momento dado. Una implementación más realista debiera permitir que cada shuttle sólo tuviese entradas para aquellas propiedades que utiliza. Esto es fácilmente implementable haciendo que en cada entrada del vector de valores de propiedades (*pmask*) esté contenido el identificador de propiedad. De este modo se aumentaría la escalabilidad en cuanto al número de propiedades se refiere.

Por último, y aunque en el diseño de *Off* las funciones de conmutación están invocadas mediante portales, en la implementación actual hemos utilizado llamadas a procedimiento en todos aquellos casos dónde el código de dichas funciones estaba contenido dentro del kernel. Creemos que esta implementación puede ser suficiente en la mayoría de los casos y esperamos cierta evolución del sistema en este aspecto.

Implementación del Shuttle

En la implementación actual cada shuttle está definido por una estructura de tipo *shuttle_t*. Estas estructuras están almacenadas en Slabs dentro del sistema, aunque su información es legible desde área de usuario disponiendo de los correspondientes derechos de acceso.

En la figura 3.13 vemos como *shuttle_t* contiene el identificador del shuttle, *id*, y el estado del procesador. Este último está compuesto principalmente por el TSS utilizado para dicho shuttle (*tss*), que contiene el estado del procesador; su descriptor (*descnb*) o índice en una tabla de TSSs, empleada por el hardware en la conmutación de tareas; y el estado del coprocesador matemático (*x87*).

```
typedef struct shuttle_st {
   shtl_id_t          id;            /* shuttle identifier */
   struct x86_tss     tss;           /* task state segment */
   int                descnb;        /* TSS descriptor # */
   struct i386_fp_save x87;          /* co-processor state */
   vm_offset_t       *ksp;           /* kernel stack */
   struct trap_state *uregs;         /* old user state */
   u_int             *kregs;         /* portal-specific shtl kern jmpbuf */
   shtl_pmask_t      *pmask;         /* shuttle properties  */
   u_long             qticks;        /* # of ticks on a CPU */
   u_long             quanta;        /* # of quanta expired (#preempted) */
} shuttle_t;
```

Figura 3.13: Implementación del Shuttle de *Off*

El tratamiento de excepciones, realizado en el contexto de la tarea interrumpida y la existencia de up-calls, hacen necesarios otros campos que permitan obtener de forma rápida el contenido de los registros de usuario, *uregs* y de los registros del kernel, *kregs*. También es necesario almacenar la posición de comienzo de la pila utilizada para el kernel, *ksp* por motivos de liberación de memoria. Describiremos el tratamiento de excepciones de forma detallada en el apartado 3.4.3, por lo que no diremos nada más al respecto por el momento.

El vector de propiedades utilizado por el shuttle está identificado por el campo *pmask*.

Por último, *qticks* y *quanta* son meros estadísticos disponibles para los servicios de gestión de procesos implementados en área de usuario; hablaremos de ellos en el apartado 3.3.5, dedicada a la planificación del procesador.

De todo lo dicho se concluye que un shuttle contiene dos elementos principalmente:

- El estado del procesador (*tss*, *x87* y *descnb*).
- El vector de propiedades (*pmask*).

El resto de la información se debe más a detalles de la implementación efectuada que a necesidades derivadas del diseño del sistema.

El estado del procesador viene representado por una propiedad predefinida por el sistema (*PPROCCTX*, o contexto del procesador). La función de conmutación de dicha propiedad (*x86shtl sw*) la suministra el propio servidor de shuttles y es la encargada de efectuar el cambio de contexto del procesador.

Para cambiar de contexto (ver figura 3.14) se recorre el vector de propiedades, (ignorando las predefinidas por el gestor de Shuttles) y se invoca a la función de conmutación (situada en *propCB[prop]*) cuando una propiedad cambie de valor. Después de esto, *siempre* se realiza la parte del cambio de contexto correspondiente a las propiedades predefinidas. En particular, se invoca a *x86shtl sw*, que es la función de conmutación que realiza el cambio de contexto de procesador.

3.3.5 Planificación

Los servicios de planificación incorporados en *Off* permiten la realización en área de usuario de diversos algoritmos de planificación (al igual que ocurre en otros sistemas como Aegis [60] y Fluke [71]). A diferencia de lo que ocurre en estos otros sistemas, es posible tratar la red entera como si de un multiprocesador se tratase.

Esquemas novedosos como los utilizados en Scout [85] donde la comunicación dicta la planificación (cómo en sistemas multimedia que requieren de transmisión de vídeo y audio y, por tanto, deben satisfacer ciertas restricciones temporales) son posibles ahora sin necesidad de ser aplicados a la totalidad de las aplicaciones. Téngase en cuenta que los procesadores en *Off* se limitan a repartir quanta entre las aplicaciones, con lo que es factible que distintas aplicaciones empleen "planificadores" distintos mediante el empleo de un sistema de planificación jerárquica (el μkernel reparte quanta, que asigna un planificador de primer nivel entre diversos planificadores de segundo nivel, éstos a su vez reparten los quanta que tienen asignados, etc.) Esto abre nuevos caminos y la posibilidad de experimentar con técnicas de planificación destinadas a la consecución de SSOO distribuidos con soporte para multimedia [120].

En *Off*, cada procesador tiene asociado un vector de identificadores de shuttle. Dicho vector puede ser (re)instalado en cualquier momento y representa los quanta de procesador del procesador en que está ubicado. Cada uno de los elementos del vector (quantum) es susceptible de asignación ante peticiones procedentes de las aplicaciones. El esquema completo puede verse en la figura 3.15.

En el ejemplo que aparece en la figura 3.15 hay tres procesadores *P1*, *P2*, *P3* y siete shuttles (1...7). Cada procesador ejecutará cíclicamente los elementos no vacíos del vector asociado *runq*. Los elementos vacíos pueden asignarse en demanda y, naturalmente, una vez lleno el vector futuras

```
shtl_switch(shtl_id_t current_shuttle, shtl_id_t next_shuttle)
{
   /* Obtain Shuttles for the given ids */
   shuttle_t *nxt    = locate_shuttle_from_id(next_shuttle);
   shuttle_t *current= locate_shuttle_from_id(current_shuttle);
   /* Switch properties */
   for(i=first user defined property; i<last property; i++){
         shtl_pval_t oldp,nxtp;

         /* Get old property value (deafault if not defined for current)*/
         shtl_pval_t oldp=(i<current->pmask->len)?
                             current->pmask->p[i] : propCB[i].dflt;

         /* Get new property value (deafault if not defined for next)*/
         shtl_pval_t nxtp=(i< nxt->pmask->len)   ?
                             nxt->pmask->p[i]     : propCB[i].dflt;

         /* Call switch function if they differ */
         if (oldp != nxtp && (cb=propCB[i].p))
             (*cb)(oldp,nxtp,&(nxt_shuttle->hw.x86))
         }
   }
   /* Switch processor context */
   x86shtl_sw(current,nxt); /* does a 'ljmp' to the nxt shuttle TSS */

}
```

Figura 3.14: Algoritmo de cambio de contexto entre Shuttles

peticiones de asignación darán lugar a un evento de revocación disparado por el sistema (tal y como se dijo en el capítulo anterior).

Como puede apreciarse, no hay inconveniente en situar un mismo shuttle (en la figura 3.15, el shuttle 3) en el vector de ejecución de varios procesadores. Ahora bien, dado que un shuttle representa el estado de un procesador (aunque virtualizado con extensiones), no es razonable que ejecute *simultáneamente* en más de un procesador. Para evitarlo, cada shuttle tiene un flag que se activa cuando un procesador lo carga para ejecución y se desactiva cuando el shuttle abandona dicho procesador. Los shuttles que poseen dicho flag activado se ignoran a la hora de su consideración para ejecución. De este modo es factible incluir dicho shuttle en más de un procesador y si, por error, se requiriese su ejecución simultánea en distintos procesadores, ésta puede evitarse.

Este modelo de planificación —el reparto de intervalos de tiempo de procesador— es un modelo simple que permite la implementación en área de usuario del esquema de planificación escogido. Como prueba basta considerar que el exokernel Aegis (que tiene una planificación semejante, para un sistema centralizado) soporta los esquemas de planificación empleados tradicionalmente (y otros experimentales). Dado que el modelo de planificación que hemos presentado es similar al de Aegis salvo por la distribución del sistema, cabe pensar que también soporta dichos esquemas de planificación.

Otra característica del modelo de planificación es que se incentiva la preplanificación. Ésto es, la instalación en la cola de ejecución (por parte del planificador que ejecuta sobre el μkernel) de más de un shuttle. Si no sucede ningún suceso imprevisto por el planificador es factible efectuar varios

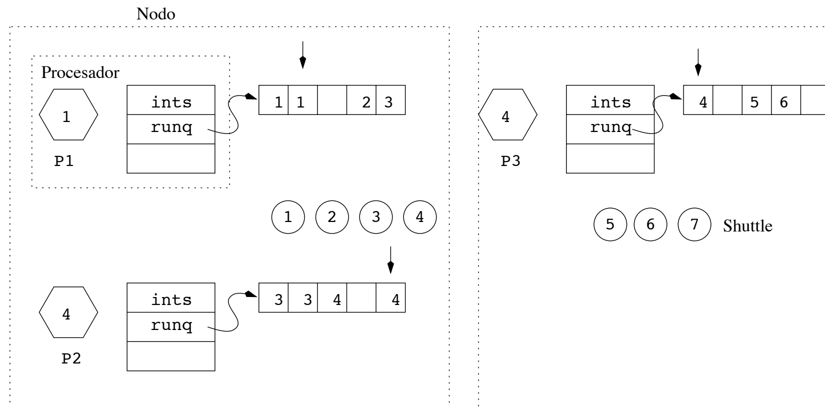

Figura 3.15: Vectores de ejecución en los procesadores de Off.

cambios de contexto (agotar varias ranuras de procesador) sin que sea preciso emplear llamadas entre el μkernel y el usuario. De no emplear preplanificación la sobrecarga introducida por las llamadas entre μkernel y planificador sería intolerable en cuyo caso sería necesaria la inclusión del planificador dentro del μkernel.

A modo de ejemplo, indicamos cómo pueden implementarse algunos esquemas de planificación bien conocidos:

- *Ejecutivos cíclicos*. Instalando un vector con un único shuttle se consigue evitar la expulsión del procesador. Dicho shuttle, que no ha de coincidir con los threads o tareas percibidas por los usuarios (como ocurre en sistemas como Solaris [81, 101] o Spring [84]) puede emplear un esquema de corrutinas para implementar el ejecutivo cíclico.

- *Planificación estática*. Basta con asignar a priori los elementos del vector. El quantum puede establecerse como el máximo común divisor de los tiempos de ejecución que aparecen en la tabla de planificación.

- *Planificación con prioridades*. Si se desea dar el control del procesador a un planificador que aplique un sistema de prioridades, basta con preasignar todos los quanta a dicho planificador. Posteriormente el planificador instalará en el vector aquellos identificadores de los shuttles que deban ejecutar.

  Alternativamente puede instalarse en el procesador el shuttle del planificador y hacer que el planificador por prioridades ceda su shuttle a las aplicaciones que deban ejecutar.

  Esquemas mixtos que combinen las dos implementaciones mencionadas también son posibles.

3.3.6 Quanta vs Interrupciones

La planificación del procesador se ve alterada por la ocurrencia de eventos asíncronos procedentes del hardware o interrupciones.

En este sentido, es preciso llegar a un compromiso entre el deseo del shuttle que está ejecutando de continuar en el procesador y el deseo de un gestor de interrupción de ejecutar cuanto antes.

La solución adoptada por *Off* consiste en el empleo de prioridades de ejecución (no confundir con la planificación por prioridades). Éstas indican qué prioridad tiene un shuttle que está ejecutando ante la ocurrencia de una interrupción. Cada interrupción lleva así mismo asociado un nivel de prioridad.

Sólo aquellas interrupciones que superan el nivel de prioridad del shuttle en ejecución son atendidas. El resto aguardan a que el nivel de prioridad del shuttle en ejecución sea reducido[14]. En este sentido, los niveles de prioridad de ejecución de los shuttles se corresponden a los niveles de privilegio de ejecución presentes en algunos procesadores.

Los eventos síncronos (o excepciones) no plantean problemas en cuanto a planificación se refiere: el tratamiento de los mismos no se considera de modo especial. Cuando ocurre una excepción el μkernel la recoge inicialmente, posiblemente para reenviarla a la aplicación para su tratamiento. Dicho tratamiento lo realiza el shuttle que sufrió la excepción, por lo que sigue rigiendo la planificación expulsiva por quanta que se aplica tal y como ya hemos mencionado.

3.4 Portales: soporte para IPC en *Off*

El mecanismo de intercomunicación de procesos[15] utilizado en *Off* es el *portal* y su invocación. Un portal es esencialmente un puerto al que pueden enviarse mensajes. Ahora bien, como veremos en este apartado, ahí es donde acaba la similitud con los "puertos" utilizados tradicionalmente en comunicaciones. Al margen de los portales no hay ningún otro mecanismo de intercomunicación de procesos en *Off*, obviando naturalemente el posible uso de memoria compartida.

Los mecanismos de IPC empleados por otros sistemas tratan de ser autosuficientes en el sentido de bastarse para el propósito de permitir la comunicación entre procesos. Es típico que dichos mecanismos de IPC contemplen (y o bien ignoren o bien solucionen) los problemas relacionados con la sincronización, transferencia de datos y heterogeneidad.

En *Off*, por un lado, hemos tratado de dar un servicio mínimo para que sea factible y efectiva la realización de diversos mecanismos de IPC. No hemos tratado, no obstante, de suministrar un servicio completo de IPC. Los portales de *Off* incorporan aquella parte de la IPC que no es factible (por razones de seguridad, o eficiencia) dejar en manos de las aplicaciones. Ni los protocolos empleados cuando la IPC tiene lugar a través de una red de comunicaciones, ni los mecanismos para transferencia masiva de datos, ni las transformaciones de datos para soportar heterogeneidad están incluidas en el mecanismo suministrado por los portales.

Por otro lado, los portales tienen características (nombres únicos, posibilidad de cambiar de ubicación, etc.) que los hacen más flexibles que otros mecanismos de IPC que típicamente están anclados a un nodo e incluso pierden su semántica fuera del nodo en que existen.

En resumen, los portales de *Off* son un sistema adaptable y distribuido de IPC que permite a cada aplicación adaptar la intercomunicación de procesos a sus necesidades, a la vez que mantiene ciertos servicios básicos de distribución de la IPC en la red en que se opere.

Este apartado muestra el diseño e implementación del servidor de Portales, que implementa la abstracción Portal. Pero antes de mostrarla daremos un breve repaso a otras alternativas de IPC en sistemas más tradicionales.

---

[14]Para implementar ésto basta anotar la ocurrencia de dicha interrupción en una máscara de interrupciones "pendientes". Como el nivel de prioridad se implementa como una propiedad de Shuttle, la función de conmutación de dicha propiedad puede mirar el conjunto de interrupciones pendientes siempre que sea invocada (siempre que varíe el nivel de prioridad en el transcurso de un cambio de contexto) y contemplar los niveles de prioridad de las mismas de tal modo que si el nuevo valor de prioridad (del shuttle) a instalar permite el envío de interrupciones "suspendidas" éstas se envían

[15]Aunque debemos recordar que *Off* no tiene ninguna abstracción que corresponda a un "proceso"

3.4.1 Otros enfoques en IPC y tratamiento de eventos

Señales

Tradicionalmente, el SO ha ocultado la ocurrencia de eventos e interrupciones a las aplicaciones. No obstante, algunos sistemas como UNIX, sólo en ciertos casos, notifican la ocurrencia de un evento (por ejemplo, el intento de ejecución de una instrucción ilegal) mediante una *señal*.

En efecto, las señales de UNIX se emplean para tales propósitos, aunque también se utilizan para notificación de expiración de temporizadores, sincronización de la E/S de las aplicaciones respecto del terminal en que éstas operan, etc. Este mecanismo es en realidad un mecanismo de IPC puesto que es factible que unos procesos envíen señales a otros, pero presenta varios inconvenientes como vemos a continuación.

Por un lado, la imposibilidad de enviar información conjuntamente con la señal (corregida sólo en los últimos años [90]), y la pesada carga semántica del mecanismo de señales (que tiene implicaciones hasta en la planificación de las aplicaciones señaladas) las hacen inadecuadas como mecanismo general de intercomunicación de procesos. Por otro lado, las señales están restringidas a un único nodo y no es factible enviar señales de un nodo a otro.

Finalmente, existe cierta similitud entre la señales y los portales, puesto que éstos permiten la notificación asíncrona de sucesos igual que ocurre con aquellas.

Mensajes y Puertos

Casi la totalidad de los μkernels existentes utilizan algún tipo de buzón de mensajes o *puerto* como abstracción principal para IPC.

Los puertos son sencillamente buzones a los que las aplicaciones pueden enviar mensajes [55, 74]. Naturalmente también es posible recibir mensajes de estos buzones.

En sistemas como Mach [55, 2] y otros muchos, los puertos tienen capacidad de almacenamiento y retienen los mensajes enviados que no han podido entregarse. Cuando el almacenamiento se agota se aplica algún mecanismo de control de flujo. Aquellas aplicaciones que no necesitan almacenamiento de mensajes deben soportar la sobrecarga añadida por esta facilidad, puesto que no es opcional. Las aplicaciones que desean almacenamiento de mensajes deben adoptar la implementación suministrada por el μkernel, dado que ésta está contenida y cableada dentro del núcleo.

Hay otro inconveniente que presentan los puertos empleados comúnmente.Éstos están fijos en un determinado nodo del sistema y no es posible reubicarlos. La *distribución* de puertos suministrada en modernos μkernels suele consistir tan sólo en la posibilidad de invocar transparentemente a puertos remotos. Como consecuencia, es típicamente imposible “mover” una aplicación y los puertos por ella servidos de un nodo a otro, con lo que es necesario emplear mecanismos adicionales (que introducen otro nivel de sobrecarga en el sistema de IPC) para “reconectar” los posibles clientes a la aplicación siempre que ésta sufre un cambio de ubicación.

Lo que es más, en algunos casos los puertos no sólo están anclados a un nodo sino también a una aplicación concreta y no es posible que una aplicación *ceda* algunos puertos a otra para delegar parte de su carga de trabajo. Afortunadamente existen sistemas flexibles como Kea [170] y otros permiten la cesión o delegación de puertos—aunque, típicamente, no toleran la “reubicación” de los mismos.

Por último, es habitual que los μkernels que incorporan puertos (salvo honrosas excepciones como L4 [111, 110] y otros) sólo soporten un modelo de objetos activo. Consiste éste en múltiples aplicaciones (u objetos) que poseen uno o más flujos de control (están activos) y se comunican entre sí mediante envío de mensajes. Esto introduce problemas de latencia y *jitter*s en las implementaciones de modelos cliente/servidor. La causa es la variabilidad que introduce la ejecución del planificador entre las ejecuciones del cliente y del servidor.

Como sostienen algunos autores [72], los μkernels deberían soportar modelos de objetos pasivos. En el modelo de objetos pasivos los clientes ceden su flujo de control al servidor durante la ejecución de éste. No es necesario replanificar en cada interacción cliente/servidor y la latencia es menor y más regular.

Los portales de *Off* tratan de superar los problemas arriba mencionados, como veremos a lo largo de este apartado.

Doors

Un avance introducido en los últimos años (aunque es más bien una recuperación de una vieja idea— los "*gates*" de MULTICS [125]) ha sido la incorporación de transferencias protegidas de control a los sistemas de IPC.

En esencia, éstas son similares a las puertas de MULTICS: un cliente invoca una puerta que produce un cambio de dominio de protección y la ejecución de una rutina en el domino del servidor, pero empleando el flujo de control del cliente.

Uno de los sistemas que incorpora este modelo es Spring [84]. Spring denomina "door" a esta abstracción.

A pesar de las ventajas en eficiencia y simplicidad (véase [72, 84]) que suponen para Spring la existencia de *doors*, éstas abstracciones también están ancladas en un nodo y no son reubicables, aunque puedan invocarse transparentemente de forma remota. Lo mismo sucede con abstracciones equivalentes en otros sistemas como Aegis [62, 59] L4 [111, 110], Amoeba [121] y Nanos [76].

Es de reseñar que no todos los sistemas que incorporan abstracciones similares a las *doors* de Spring tratan de suministrar "puertas de acceso" tan sencillas como sea posible. Por ejemplo, en L4 los mensajes viajan a través de dominios anidados (*clanes*) de tal modo que cuando un mensaje atraviesa varios dominios sufre un reenvío extra. La razón es que cada dominio posee una única entidad (*chief*) capaz de alcanzar dominios exteriores. Si un miembro de un clan distinto al *chief* desea alcanzar otro dominio el mensaje ha de pasar al menos por los *chiefs* de ambos clanes (ver figura 3.16). Como consecuencia, aunque L4 posee la IPC más rápida según las medidas publicadas [111], en el caso mencionado la sobrecarga deja de ser despreciable. La lección que hemos tratado de aprender en la realización de los portales de *Off* es la de no imponer un sistema completo, transparente y completamente funcional de IPC; al contrario, tratamos de dar el mínimo indispensable como dijimos anteriormente.

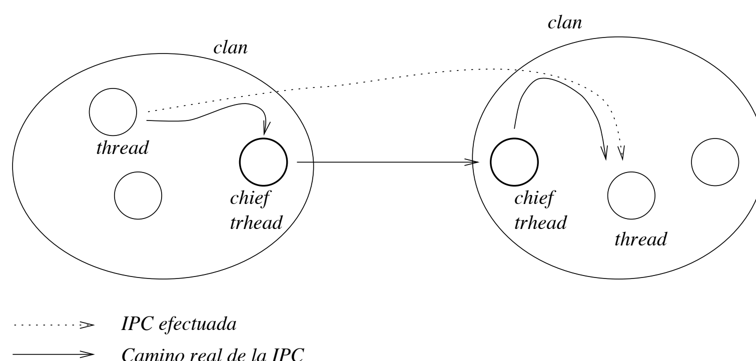

Figura 3.16: IPC en L4. Los *chiefs* son los puntos de acceso a un clan.

Volviendo a los *doors*, la incapacidad de ésta abstracción (también de aquellas mencionadas en los apartados anteriores) de moverse de un nodo a otro no hace aconsejable su uso para representar

contenedores de recursos físicos. Ya vimos anteriormente cómo éstos podían reemplazarse en *Off* y cambiar de ubicación durante el reemplazamiento. Si los “puertos” empleados no pueden cambiar de posición estamos impidiendo el reemplazamiento en caliente de contenedores de recursos físicos.

3.4.2 Los portales de *Off*

En *Off* hemos implementado una nueva abstracción de IPC denominada Portal que soporta tanto transferencias protegidas de control como envío de mensajes asíncronos y síncronos, invocación distribuida (empleando bien protocolos genéricos, bien protocolos específicos para cada aplicación) y reubicación entre distintos nodos. Además, los portales permiten múltiples invocaciones simultaneas a través del mismo portal. En pocas palabras, cada portal en *Off* representa un *punto de acceso direccionable desde cualquier nodo* en la red.

Una vez creado, un portal lleva asociado un manejador que puede reemplazarse posteriormente. El manejador viene definido principalmente por dos elementos:

- Un contador de programa que indica el comienzo de la rutina que habrá de ejecutar cuando dicho portal se invoque.

- La información necesaria para obtener un shuttle y una pila sobre la que dicha rutina deberá ejecutar.

El dominio de protección en que dicho contador de programa y dicha pila adquieren sentido podría considerarse también como parte esencial de la definición del manejador. No obstante, el diseño del sistema permite la operación sin empleo de dominios de protección; por esta razón podríamos considerarlo como una componente “opcional” en la definición del manejador.

Los portales implementan dos mecanismos básicos que constituyen dos formas de invocación distintas y pueden emplearse para la realización de distintos modelos de IPC.

1. Envío de mensajes asíncronos. Un cliente envía asíncronamente un mensaje compuesto por una serie de palabras máquina (posiblemente ninguna) a un portal. Como resultado de la invocación del portal, un manejador preespecificado comenzará su ejecución asíncronamente[16] en un servidor. La ejecución del cliente se bloquea durante el tiempo mínimo necesario para iniciar el envío del mensaje (hasta que el estado del servidor ha sido alterado para que él mismo invoque al manejador del portal en el caso de envíos locales y hasta que el servidor que implementa el protocolo de transporte en red ha sido invocado en el caso de envíos remotos). Podríamos considerar que la invocación es no-bloqueante puesto que el envío puede proseguir mientras el cliente está ya ejecutando el código presente tras la invocación del portal.

2. Transferencias de flujo de control de un cliente a un servidor. Un cliente envía síncronamente un mensaje compuesto por una serie de palabras máquina (posiblemente ninguna) a un portal. Como resultado de la invocación, el flujo de control (el shuttle) del cliente se “dona” temporalmente para que ejecute un manejador preespecificado en el servidor.

En ambos casos la invocación del portal transfiere un número dado de palabras máquina (el mensaje) y activa la ejecución de un manejador cuya finalidad es procesar el mensaje transferido.

Nótese que estamos empleando “mensaje” para referirnos a la información transferida en la invocación. Los mensajes que realmente transfieran los usuarios del sistema de portales no tienen por que ir contenidos en estos “mensajes” de los que hablamos continuamente. Por ejemplo, el “mensaje” que transfiere el portal podría ser una referencia a un buffer de memoria compartida que contuviese

[16] Con respecto a la ejecución de código en el servidor

el mensaje transferido por el usuario, cosa que podría hacerse para implementar protocolos de comunicaciones con cero-copias (sin copiar repetidamente los mensajes en su curso entre el hardware de comunicaciones y las aplicaciones) como los ya existentes para TCP en algunos sistemas como Solaris [41].

Los portales permiten la realizacion de esquemas de objetos pasivos [73] gracias al segundo mecanismo. No obstante hay que tener en cuenta que también pueden utilizarse para implementar un modelo de objetos activos (mediante cualquiera de los mecanismos). Ambos mecanismos se emplean también para enviar excepciones, interrupciones y otros eventos a las aplicaciones. Aunque veremos más adelante ambos casos en más detalle, se trata sencillamente de convertir la ocurrencia de uno de estos sucesos en invocación de portales (para ello basta que el μkernel invoque el portal adecuado tras el suceso considerado).

Salvo por detalles de sincronización entre el cliente y el servidor, ambos mecanismos se diferencian principalmente en el proceso de elección del shuttle sobre el que ejecutará el manejador del servidor como resultado de la invocación del portal. En el primer caso (envío asíncrono), el sistema activa el manejador de portal asíncronamente en un shuttle pre-especificado; este shuttle podría estar ejecutando tareas de "mantenimiento"en el servidor antes de la invocación del portal, podría también estar bloqueado esperando la invocación del portal. En el segundo caso (envío síncrono), el shuttle origen se emplea para ejecutar el código del manejador; no es necesario disponer de un shuttle preexistente en el servidor esta vez (aunque si se requiere una pila disponible para que el shuttle que realiza la invocación pueda ejecutar en el servidor).

Para comprender como operan los portales es conveniente ver la arquitectura del sistema de portales en su conjunto. En la figura 3.17 podemos ver todos los elementos involucrados:

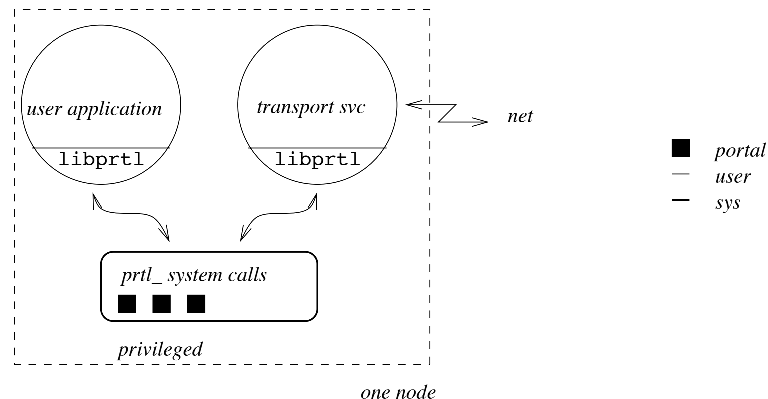

Figura 3.17: Elementos involucrados en el sistema de portales

- La librería de portales suministra un interfaz adecuado y permite realizar en área de usuario todas aquellas primitivas que pueden estar fuera del núcleo.

- Las únicas llamadas al sistema son en realidad *sys_prtl_send* y *sys_prtl_ret*. Éstas están implementadas por el servidor de portales del núcleo y permiten que se efectúen llamadas entre distintos dominios de protección de una forma segura y transparente para la aplicación, de forma similar a como ocurre en Kea [170].

- Un servicio de transporte (posiblemente en otro dominio de protección, aunque no necesariamente) permite que de forma transparente los mensajes sean capaces de llegar a portales ubicados en otros nodos. Cuando se invoca un portal remoto, el μkernel llama al servicio de

transporte identificado por el *portal de transporte* designado por la aplicación. Este portal no es otra cosa que el designado por la aplicación para gestión de las excepciones enviadas por el μkernel cuando es preciso redirigir una llamada al sistema a un nodo remoto. Podemos imaginar al servicio de transporte como un intermediario (especificado por la aplicación) entre el μkernel local y el remoto en aquellas invocaciones de portal destinadas a otros nodos.

En realidad, este servicio de transporte deberá incluir también un protocolo de localización de portales. El servidor deberá pues esperar peticiones de localización de portales y ayudar a procesarlas como vimos en el apartado 2.1.7.

Shuttles y portales

Cuando un cliente invoca un portal (ya sea mediante un envío asíncrono o mediante uno síncrono) es conveniente *ajustar* las propiedades del shuttle que ejecutará el manejador del portal en función del estado del cliente y del servidor. Intuitivamente, si el servidor opera en un espacio de direcciones dado y el cliente posee una prioridad de ejecución frente a eventos dada (ver apartado 3.3.6), lo más adecuado podría ser ejecutar el manejador que sirve al cliente con el espacio de direcciones del servidor y con el nivel de prioridad del cliente. De este modo la ejecución de un servicio se puede *adaptar* en función del cliente que lo solicita.

El mecanismo empleado es una *máscara de propiedades*. Cada portal posee un vector que actúa como máscara de propiedades. La misión del mismo es modificar los valores de las propiedades del shuttle que ejecuta el manejador del portal para *adaptar* el contexto en que ejecutará dicho shuttle a la ejecución del manejador. Dicho vector posee la misma estructura que los vectores de valores de propiedades presentes en los shuttles. Cada componente del vector, determinará el valor que adoptará una propiedad durante la ejecución del manejador:

- el valor que presentaba dicha propiedad el shuttle cliente antes de efectuar la invocación (por ej: fijar la propiedad “identificador de usuario” al valor presente en el shuttle que realiza la llamada),
- un valor preestablecido que se indica en la máscara de propiedades (por ej: fijar la propiedad “espacio de direcciones” a un valor que corresponde con los privilegios mínimos necesarios para la ejecución del servicio invocado),
- el valor que presenta dicha propiedad en el servidor (por ej: fijar la propiedad “prioridad de ejecución ante interrupciones” al valor presente en el servidor).

Un empleo típico es el ajuste del nivel de privilegio, prioridad de ejecución, derechos de acceso, etc. en función del shuttle que realiza la invocación del portal.

Upcalls

La parte más interesante del envío mediante portales es la realización de *upcalls* [42]. Consisten éstas en llamadas del núcleo a las aplicaciones, o intuitivamente, la inversa de una llamada al sistema. Este mecanismo primario se emplea no sólo para enviar excepciones e interrupciones a las aplicaciones sino también para implementar los envíos de mensajes a portales tal y como se muestra en la figura 3.18. En dicha figura podemos ver a rasgos generales cómo procede el envío de interrupciones, excepciones, mensajes y PCTs emplando upcalls en algunos casos.

En la figura 3.18-a vemos como un upcall es sencillamente una “llamada al sistema a la inversa”. En 3.18-b vemos que la notificación de eventos síncronos, con respecto a la ejecución de instrucciones de la aplicación que los sufre, es simplemente un upcall originado por la ocurrencia de un suceso

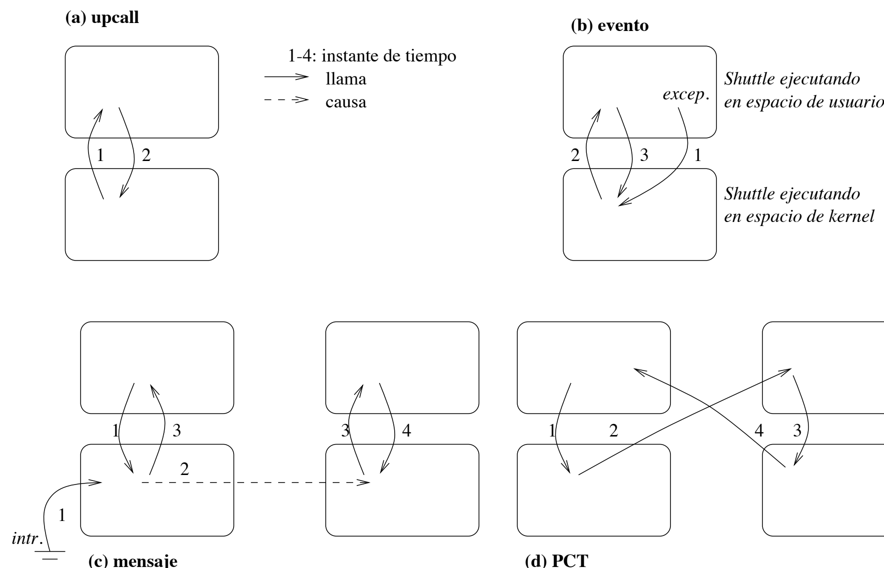

Figura 3.18: Uso de upcalls como base para eventos e invocaciones de portales.

(naturalmente, éste upcall lo causa en envío de un mensaje a un portal). En 3.18-c tenemos un envío de mensaje asíncrono (que podría originarse dentro del μkernel ante la ocurrencia de una interrupción—como en el paso 1 de la figura), la entrega del mismo se efectúa de forma asíncrona mediante un upcall, como veremos en el apartado siguiente. Por último, en 3.18-d vemos una transferencia protegida de control cuyo funcionamiento detallaremos en el apartado 3.4.2.

Como cabe suponer, una llamada al sistema puede dar lugar a un upcall, ésta a una nueva llamada al sistema, etc.

Para implementar upcalls (ver figura 3.19) aprovechamos el mecanismo (suministrado por el hardware) de traps. Para realizar la primera parte de un upcall (la invocación de la rutina de usuario a la que se desea llamar) construimos en la pila del kernel una estructura similar a la que salva el procesador cuando se produce una llamada al sistema. Dado que el hardware emplea esta estructura para recuperar, en el retorno de una llamada al sistema, el estado del usuario antes de que se produjese esta llamada, ejecutamos una instrucción `iret` (como si de un retorno se tratase). El procesador toma de la pila del kernel el "falso" estado que hemos dejado preparado y lo recupera. Por supuesto, antes de invocar a la rutina salvamos el estado del procesador para poder recuperarlo cuando se retorne del upcall, de tal modo que el μkernel siga ejecutando la instrucción posterior a la llamada a rutina de usuario.

Con este simple mecanismo conseguimos "saltar" a la rutina que deseamos llamar en área de usuario (denominada "func. usr." en la figura). Paradójicamente, mediante una instrucción de retorno y no mediante una de llamada.

La parte más complicada es retornar de nuevo al núcleo. Para conseguirlo (ver figura 3.20) hemos de conseguir que se produzca una llamada al sistema justo tras el retorno de la rutina de usuario que hemos invocado. Esto se consigue alterando la dirección de retorno de la rutina invocada, en la pila del usuario ("ret. falso", en la figura), antes de invocarla. Esta dirección de retorno (que determinará la dirección en la que se seguirá ejecutando instrucciones cuando la rutina termine) se hace apuntar a una instrucción (almacenada directamente por el μkernel en la pila del usuario) que produce la llamada al sistema deseada.

Esta llamada al sistema (`sys_prtl_ret`), toma el estado del núcleo que salvaguardamos ante-

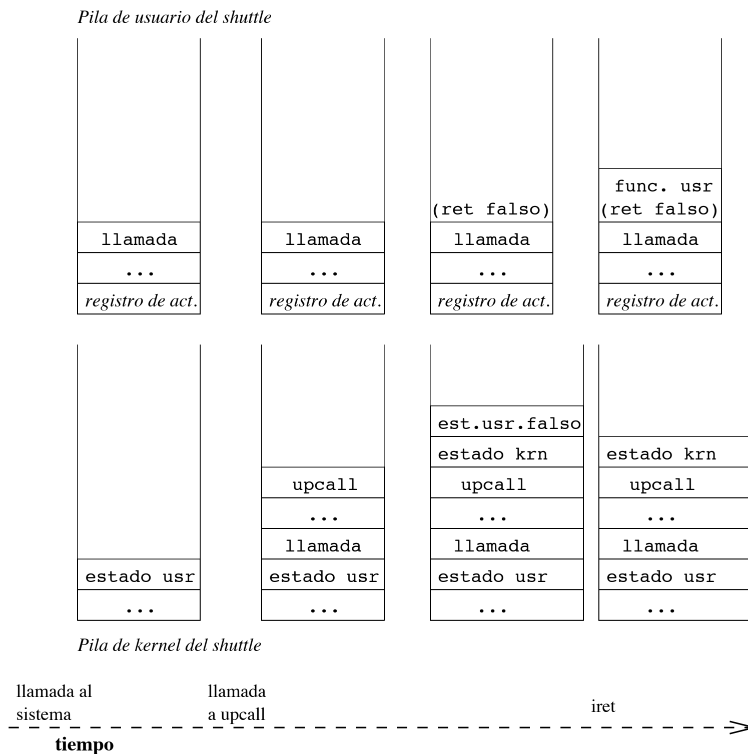

Figura 3.19: Funcionamiento de un upcall: llamada.

riormente y lo restaura. Como efecto lateral de la restauración, la pila del kernel vuelve a su posición original (antes de efectuar el upcall). El upcall se ha completado en este punto y, a partir de ese momento, se continúa con la llamada al sistema que originó el upcall. Cuando ésta retorne encontrará el estado de usuario correspondiente a dicha llamada al sistema y el siguiente `iret` (que finaliza la llamada al sistema) restaurará el estado del shuttle para que este continúe justo después de su llamada al sistema.

Hay un detalle que es preciso considerar en este punto. Como es natural, el servidor de Shuttles ofrece puntos de entrada para que sea posible obtener (o modificar) el contenido de los registros (de procesador) de un shuttle cualquiera[17], siempre que se presenten los correspondientes derechos de acceso. La pregunta que surge es ¿De qué registros estamos hablando? Las llamadas al sistema pueden dar lugar a llamadas a rutinas de usuario y éstas de nuevo a llamadas al sistema con lo que puede haber varias llamadas al sistema en curso (anidadas) en un mismo shuttle. ¿A qué "registros" nos referimos pues?, ¿A los correspondientes a la primera llamada al sistema o a la más externa? ¿A los correspondientes a la última o más interna?. Es decir, dado que pueden existir múltiples estados de ejecución anidados unos dentro de otros (como sucede con los estados de la ejecución de un programa cuando tenemos llamadas anidadas a funciones), puede existir más de un juego de registros (tal y como los ve el usuario) para un mismo shuttle. Naturalmente, sólo uno de estos juegos corresponderá al estado actual del shuttle.

En realidad, dado que las llamadas anidadas no corresponden sólo a upcalls, sino también a llamadas al sistema, tenemos una sucesión de juegos de registros correspondientes al estado del usuario y otra sucesión correspondiente al estado del núcleo. Los primeros corresponden a los instantes en que se produjo una llamada al sistema (y se salvaguardó el estado del usuario) y los segundos a los

[17] Además hay otros puntos de entrada que permiten consultar y alterar el valor de las propiedades de un shuttle.

Figura 3.20: Funcionamiento de un upcall: retorno.

instantes en que se produjo un upcall (y se salvaguardó el estado del núcleo).

En la implementación actual sólo es posible acceder al juego de registros más externo (el más reciente, o más superficial en la pila) de cada shuttle. Más concretamente, es posible acceder al juego más externo correspondiente a modo usuario y al juego más externo correspondiente a modo kernel.

Para conseguir esto, los contextos salvaguardados se enlazan entre sí en dos listas, una para contextos de usuario y otra para contextos de kernel. El mantenimiento de éstas listas se reduce a unas pocas operaciones en la entrada al núcleo y otras pocas en la salida del mismo. La existencia de éstas listas permite que con escasas modificaciones al código actual sea factible acceder a cualquier contexto de usuario o núcleo de un shuttle dado, no sólo al más externo como en la actualidad.

Una alternativa a la implementación efectuada habría sido una implementación tradicional en la que no es necesario mantener éstas listas. Consistiría en implementar un modelo similar al de los procesos UNIX, donde el núcleo, que ejecuta en el contexto de un determinado proceso, no realiza tareas sólo para el proceso en cuyo contexto ejecuta, sino que también realiza tareas en favor de otros procesos.

Nosotros en cambio hemos tratado de *compartimentar* la ejecución del núcleo de modo que éste trabaje sólo en favor de aquella aplicación en cuyo contexto ejecuta. Esto es, un shuttle que ejecuta código de usuario, ejecuta periódicamente código del núcleo en modo núcleo para realizar actividades (de núcleo) en favor de sí mismo. O dicho de otro modo, cada shuttle se hace su propio trabajo esté en área de usuario o en área de núcleo. Con el modelo implementado es factible expulsar al núcleo de forma indefinida de tal modo que sólo se perjudique al shuttle en que éste ejecutaba. Con el modelo implementado, *el núcleo es una entidad pasiva* en la que cada shuttle entra periódicamente

mediante PCTs[18].

La simplicidad de las estructuras de datos mantenidas por núcleo (téngase en cuenta que la practica totalidad de ellas no requieren de memoria dinámica) junto con el modelo de un núcleo pasivo en el que se mantiene en la medida de lo posible la independencia entre shuttles (cada ejecución en el núcleo actúa en un shuttle y en favor de dicho shuttle) permite que el fallo de un shuttle dentro del núcleo no afecte inmediatamente al resto del sistema. Esto ha resultado ser muy conveniente para el desarrollo y depuración del μkernel, tanto que no ha sido preciso emplear depurador alguno. La independencia entre shuttles facilita así mismo la movilidad de los recursos del sistema, dado que puede obviarse el hecho de que un shuttle esté ejecutando en modo usuario o kernel a la hora de capturar su estado.

Envío de mensajes

Cuando se emplea el portal como mecanismo para enviar un mensaje de un shuttle a otro, la invocación del portal causa la ejecución (de modo asíncrono) de un manejador en el contexto del servidor.

Para permitir este uso de los portales se ha incluido junto con el manejador (contador de programa e información para obtención de pila) un identificador de shuttle que identifica el shuttle en que deberá ejecutar dicho manejador. El núcleo salvará el estado de dicho shuttle e instalará el contexto necesario para la ejecución asíncrona del manejador. Si un portal no tiene instalado este identificador de shuttle no será factible su uso para envío asíncrono de mensajes.

El algoritmo de envío de mensajes en este caso es sencillo:

1. Se realiza la llamada al sistema `sys_portal_send`.

2. Se localiza el portal, si es remoto se enviará una excepción[19] (mediante un upcall) de tal modo que se reenvíe la petición al nodo en que se encuentra el portal. Si es local se continúa.

3. Se localiza el shuttle destino mediante la función `needs`:

   - Si éste es remoto se eleva una excepción (mediante un upcall) para que se localice el shuttle destino y se reenvíe la petición al nodo en que éste se encuentre. El kernel entrega las palabras que constituyen el mensaje como parámetro de la rutina llamada mediante el upcall. Se realiza una migración (implícita) del portal al nodo destino.
   - Si es local se continúa. En este punto el portal comparte nodo con el shuttle que lo maneja, esto facilita el resto de la operación.

4. Se bloquea al shuttle destino y se salva su estado. Esto se realiza tanto si el shuttle ejecutaba en modo kernel (estaba a mitad de una llamada al sistema) como si ejecutaba normalmente código de la aplicación. Este estado se salva en la pila de kernel del shuttle destino.

5. Se instala un registro de activación para `upcall` (la rutina que realiza upcalls) en la pila de kernel del shuttle destino y se fija su estado de tal modo que cuando éste ejecute realice inmediatamente un upcall. El kernel copia las palabras que constituyen el mensaje de tal modo que la rutina `upcall` (en el kernel del shuttle destino) pueda transferirlas hasta el manejador del portal (empleando sus parámetros).

[18] Aunque el mecanismo empleado sea una llamada a un portal, implementada mediante traps, la semántica corresponde a PCTs entre el usuario y el núcleo.

[19] Las “excepciones” a que nos referimos en este apartado son en realidad upcalls que realiza el kernel para requerir la ayuda de la aplicación. No obstante, la percepción del usuario es que ha ocurrido una excepción que debe tratarse para poder proseguir con la ejecución normal. Debido a ello nos referiremos a dichos upcalls como “excepciones” (percibidas por el usuario y “creadas” por el kernel).

Los parámetros de `upcall` situados en el registro de activación creado para ella están dispuestos de tal modo que `upcall` realizará un upcall al manejador del portal tan pronto como el shuttle destino ejecute.

La dirección de retorno del registro de activación preparado para `upcall` está dispuesta de tal modo que se salte a una rutina que restaurará el estado del shuttle destino a la situación en que este se encontrase antes de quedar bloqueado (sea ésta la ejecución dentro del kernel o fuera del mismo).

6. Se modifican las propiedades del shuttle destino de acuerdo con la máscara de propiedades presente en el portal.

7. Se desbloquea al shuttle destino.

8. Se retorna de la llamada al sistema `sys_portal_send`. El resto del envío procede en el contexto del shuttle destino, de tal modo que el origen puede continuar su tarea.

9. Un trampolín[20] instalado en el punto anterior, que ejecuta cuando el manejador retorna devolverá el control al μkernel. En este punto se restaura el estado (y las propiedades alteradas por la invocación del portal) salvado anteriormente. Así pues este punto corresponde en realidad al retorno del upcall que entrega el mensaje.

El shuttle que invoca el portal recupera el control en el punto 8 y puede proseguir su ejecución. La ejecución del manejador aguardará hasta el momento en que el shuttle destino disponga de un quantum para ejecutar.

Si el mensaje precisase de respuesta el remitente puede incluir un identificador de portal como parte del mensaje y bloquearse justo tras el envío. El destinatario empleará dicho identificar para enviar una respuesta al cliente; como efecto lateral de la recepción éste quedará desbloqueado.

Por último, conviene recordar que todas las operaciones efectuadas sobre shuttles se efectúan empleando los servicios del servidor de shuttles. Así por ejemplo, para bloquear al shuttle destino o cederle el resto del quantum de procesador de que disponemos[21] empleamos las rutinas `shtl_switch` y `shtl_setProp`, suministradas por el servidor de shuttles.

Transferencias protegidas de control

Podría parecer que el mecanismo básico de envío de mensajes que mencionamos anteriormente debería bastar para soportar cualquier modelo de IPC. No obstante, la realidad es que es conveniente soportar transferencias protegidas de control dado que esto supone un ahorro (principalmente en latencia) para las aplicaciones. Intuitivamente, tenemos las mismas razones que tienen los arquitectos de procesadores para suministrar “gates”: dar un mecanismo barato para que una aplicación pueda ejecutar una rutina en otro dominio de protección.

En este caso el usuario del sistema que envía el mensaje quedará bloqueado hasta que el manejador asociado al portal complete su ejecución tras la recepción del mismo. Dicho de otro modo, el shuttle del cliente se emplea para ejecutar el manejador en el servidor.

Para que un portal soporte PCTs es preciso que el servidor del portal (esto es, la aplicación donde se encuentra el manejador del portal) o el usuario que lo creó disponga de un conjunto de pilas donde puedan ejecutar las distintas invocaciones que pueda sufrir dicho portal. Nótese que las pilas deben

[20] código ejecutable instalado directamente en la pila

[21] En efecto, aunque no lo hemos mencionado, es factible ceder el resto de quantum en que ejecutamos al shuttle que ejecutará el manejador.

estar en el dominio de protección del servidor, aunque las emplee el shuttle que realiza la invocación durante el transcurso de la llamada.

Para ello la información presente en el portal contiene:

- una dirección de memoria válida como base de una pila y
- una máscara de propiedades en que al menos[22] debe incluirse el nuevo espacio de direcciones (la nueva de DTLB).

Si esta información no está presente el portal no se puede emplear para PCTs.

En cada invocación, dicha dirección se actualizará automáticamente en base a valores presentes en la base de la pila anterior. De este modo, las pilas disponibles están enlazadas y sucesivas invocaciones las van agotando una tras otra. El manejador del portal puede restaurarlas a voluntad puesto que los enlaces están mantenidos en su mismo espacio de direcciones.

La implementación de las PCTs se basa en que es factible alterar el contexto de usuario, salvado en la entrada al núcleo durante una llamada al sistema, para que el retorno se produzca *en un dominio diferente*.

El algoritmo empleado para efectuar una PCT es como sigue:

1. Se realiza la llamada al sistema `sys_portal_send`. Ésta es, naturalmente, en modo bloqueante y supondremos que el portal está en el nodo local. Si el portal fuese remoto se produciría una excepción (enviada mediante un upcall) que el usuario debería tratar. Habitualmente la llamada degenerará en el uso del mecanismo de envío de mensajes detallado anteriormente.

2. Se prepara la pila de ejecución (del manejador, en espacio de usuario) y se instala en ella un registro de activación para el manejador, incluyendo éste el mensaje en curso. Si no hubiese pila disponible la llamada se aborta.

3. Se modifican las propiedades del shuttle de acuerdo con la máscara de propiedades presente en el portal y se altera el estado del shuttle para que tras retornar de la llamada al sistema se ejecute el manejador.

4. Se modifica el contexto de usuario del shuttle de tal modo que ejecute el manejador en la pila seleccionada previamente. El contexto anterior a la modificación se salvaguarda en la pila de kernel del shuttle considerado.

5. Se retorna de la llamada al sistema en curso de tal modo que ejecute el manejador cuya instalación se ha preparado.

6. Un trampolín[23] instalado en el punto anterior, que ejecuta cuando el manejador retorna devolverá el control al μkernel. En este punto se restaura el estado (y las propiedades alteradas por la invocación del portal) salvado anteriormente. El procedimiento es el mismo que se emplea en la llamada, salvo que ahora, en lugar de crear un estado nuevo según la información presente en el portal se restaura el estado salvado anteriormente.

Cuando la modificación de propiedades del shuttle (ej.: espacio de direcciones) supone emplear recursos remotos la ejecución del manejador supondrá envío de excepciones (mediante upcalls) por parte del μkernel. Dichas excepciones se tratan como en cualquier otro caso, dando lugar a la transferencia de recursos (DTLBs, etc.) de un nodo a otro.

---

[22] Cuando el μkernel está compilado con soporte para espacios de direcciones.

[23] código ejecutable instalado directamente en la pila

Aunque la implementación actual no lo tolera, el diseño permite que el manejador de dichas excepciones aborte la PCT e inicie otra acción equivalente (transferir el shuttle al nodo donde están los recursos, emplear envío de mensajes asíncronos para simular la PCT entre varios nodos, etc.). La elección de la solución adecuada depende de la aplicación considerada y de la cantidad de información a transferir por la red, por lo que (de nuevo) dejamos en manos de la aplicación esta tarea.

Eventos hardware

Tal y como afirma J. Liedtke en [111], las IPCs pueden implementarse para que sean suficientemente rápidas como para permitir el tratamiento de interrupciones directamente con ellas. Es decir, interponer una IPC entre la ocurrencia de la interrupción y un manejador presente en área de usuario. Sistemas como Aegis [62], L4 [111], Mach [21] y VSTa [168] así lo hacen. *Off* sigue este modelo.

Cada servidor de shuttles incorpora vectores que hacen corresponder un portal con cada evento procedente del hardware. El servidor de portales convierte estos eventos en invocación de portales.

Aunque detallaremos la gestión de estos eventos cuando describamos la implementación de los portales para un Intel x86 en el apartado 3.4.3 hemos de mencionar que el envío del evento se efectúa en el contexto del manejador del evento y no en el del shuttle interrumpido. En otras palabras, la ejecución del shuttle que se ve interrumpido no se ve detenida durante toda la invocación del portal, tan sólo el tiempo necesario para que el shuttle destino sea capaz (él mismo, en modo kernel) de enviarse el evento a sí mismo (cuando le toque su turno de ejecución).

Portales remotos

Para enviar mensajes a portales en otros nodos es preciso utilizar un servidor de transporte. Dicho protocolo de transporte debe ser reemplazable por aquellas aplicaciones que lo deseen.

La implementación se basa en la distinción entre envíos locales y envíos remotos. Para envíos locales el envío procede como se describió anteriormente. Para envíos remotos se envía el mensaje junto con el identificador del portal destino y el identificador del nodo destino al portal del servidor de transporte (mediante una excepción). Intuitivamente, un envío remoto se efectúa mediante un envío local (que en realidad eleva una excepción) a un servidor de transporte.

Una vez recibido dicho mensaje, el shuttle del servidor de transporte envía el identificador del portal y el mensaje al nodo destino a través de la red. Una vez recibido el mensaje en el nodo remoto, el shuttle de transporte en dicho nodo enviá el mensaje al portal al que fuese destinado.

Los envíos desde el remitente al servidor de transporte en el nodo origen y del servidor de transporte remoto al portal remoto son locales por lo que el envío procede sin mayores complicaciones. El envío a portales remotos se realiza pues, sencillamente, mediante la interposición de un puerto de comunicaciones entre dos extremos (origen y destino) de un portal. Como cada aplicación puede indicar el portal de transporte que desee la adaptabilidad no se ve perjudicada.

3.4.3 Implementación en el Intel x86

La estructura de datos que soporta un portal (ver figura 3.21) es relativamente simple. Podemos identificar los siguientes elementos:

- El identificador del portal, `id`.
- El identificador del shuttle que ejecutará el manejador en el caso de llamadas asíncronas, `handler`.
- El punto de entrada del manejador, `pc`.

```
typedef struct portal_st {
   prtl_id_t      id;               /* portal identifier */
   shtl_id_t      handler;          /* shuttle for the portal handler  */
   vm_offset_t    pc;               /* handler entry point */
   vm_offset_t    sp;               /* handler stack for PCTs */
   prtl_pmask_t   pmask;            /* property changes */
   u_char flags;                       /* portal flags */
} portal_t;
```

Figura 3.21: Estructura de un portal

- La máscara de propiedades, `pmask`.
- El valor base para la pila en PCTs, `sp`. En este caso el shuttle a emplear es el que invoca el portal.

La estructura es bastante simple. En los siguientes apartados iremos viendo como se implementan los envíos.

Eventos: traps e interrupciones

Tanto traps como interrupciones se exportan a las aplicaciones mediante portales. Esto quiere decir que nadie fuera del núcleo recibe directamente eventos enviados por el hardware. Al contrario, estos eventos se transforman en invocaciones de portales dentro del núcleo.

Podemos clasificar los eventos procedentes del hardware (o por el enviados inicialmente) en dos categorías:

- *Eventos síncronos* o causados síncronamente respecto de la ejecución del shuttle que los sufre (que ejecuta en el instante en que sucede el evento). Los más conocidos tal vez sean los traps. Llamadas a sistema y excepciones (de fallo de página, instrucción ilegal, etc.) están incluidos en este grupo.

  Su característica más interesante, al menos para la implementación del sistema, es que el shuttle que los sufre no es capaz de continuar su ejecución hasta el punto en que el evento se ha tratado. De hecho estos eventos suelen indicar la necesidad de efectuar algún tipo de tarea antes de poder proseguir con la ejecución del shuttle que los sufrió. No sólo es factible, sino incluso aconsejable, emplear tanto el shuttle que sufrió el evento como el(los) quantums de procesador que dicho shuttle tiene asignados para procesar el evento.

- *Eventos asíncronos* o causados asíncronamente respecto de la ejecución del shuttle que los sufre. Típicamente interrupciones.

  En este caso el shuttle que los sufre puede perfectamente proseguir su ejecución con independencia del tratamiento del evento. De hecho, lo habitual es que el evento no esté dirigido al shuttle que lo sufre sino a otro cualquiera. Ahora, al contrario que en el caso anterior, no es necesario (de hecho, no suele ser conveniente) emplear el shuttle interrumpido para gestionar el evento.

Para ambos casos, el μkernel emplea una tabla para mantener los portales que corresponden a los eventos. Las entradas en esta tabla son susceptibles de asignación de tal modo que el usuario puede pedir la asignación de una línea de interrupción (indicando el portal que debe servirla) o de una excepción. Salvo por la interrupción de reloj (empleada por el servidor de shuttles para implementar

quantums, y reexportada a los usuarios mediante temporizadores) y la excepción de fallo de página (empleada por el DMM para implementar las DTLBs), el resto de eventos están disponibles para las aplicaciones.

El tratamiento de eventos síncronos es como sigue (ver figura 3.22:

1. El procesador genera el evento, lo que conduce a la entrada en modo kernel y a la ejecución de una pequeña rutina que llama a `shtl_trap_handler`, el manejador de excepciones del μkernel.

2. El manejador comprueba si la excepción corresponde a una llamada al sistema.

   (a) En caso afirmativo se llama directamente a `shtl_system_call`, que ejecutará la llamada al sistema correspondiente.

   (b) En caso negativo se consulta en la tabla de traps y se invoca el manejador correspondiente. Nótese que aunque estamos dentro del μkernel estamos ejecutando sobre el shuttle que sufrió la excepción y sujetos a la asignación de procesador determinada por la cola de ejecución que dicho procesador tenga (el μkernel es expulsable).

```
int x86shtl_trap_handler(struct trap_state *ts)
{
  switch(trap number){
  case X86SHTL_TSYSCALL:
    x86shtl_system_call(ts);
    return 0;
  /*...*/
  default:
      if (x86cpu->trap_f[trapno].flags){ /* if handled by a function */
        (*x86cpu->trap_f[trapno].h.f)(ts); /* call it */
      }
      else { /* deliver the trap -- blocking mode */
        prtl_send(x86cpu->trap_f[trapno].h.p, PRTL_BLCK, ts);
      }
    return 0;
  }
}
```

Figura 3.22: Manejador de traps del μkernel

Cuando el manejador de la excepción retorne se retornará de `shtl_trap_handler`, lo que conducirá a la reanudación de la ejecución del shuttle interrumpido en el punto en que éste se encontraba antes de la ocurrencia del trap.

El tratamiento de interrupciones es en realidad muy similar, la diferencia principal estriba en que el envío se debe realizar:

- de modo bloqueante (mediante PCTs) si el shuttle que ejecuta tiene menor prioridad que la interrupción considerada (con ello se hace ejecutar el manejador de la interrupción aunque ésto suponga parar temporalmente al shuttle interrumpido).

- de modo no bloqueante si el shuttle que ejecuta tiene mayor prioridad que la interrupción considerada (con ello el manejador de la interrupción debe aguardar hasta que el procesador esté ejecutando un shuttle con menor prioridad que la interrupción).

Aunque también se podría haber preparado la llamada asíncrona en el shuttle servidor y esperar a que le llegue su turno de ejecución esto supondría emplear más quantum del remitente, razón por la que no se ha hecho así.

Otro pequeño detalle es que la rutina real de tratamiento de interrupciones que invoca a `shtl_intr_handler` efectúa el *acknowledge* de la interrupción y la enmascara. Si el manejador de la interrupción desea permitir que ocurran nuevas interrupciones deberá desenmascararla.

Upcalls, mensajes y PCTs

Dado que no se emplea ninguna peculiaridad de la arquitectura Intel x86, la implementación de upcalls, envío de mensajes y PCTs se describió casi por completo en los apartados que exponían el funcionamiento detallado de las mismas de forma genérica. Quedan no obstante algunos detalles curiosos que precisan de código para su exposición. Hemos preferido aplazarlos para este apartado, dado que el lector que esté leyéndola estará seguramente interesado en ellos—aunque no sean precisos para la comprensión de la implementación.

En primer lugar, tenemos la instalación de registros de activación en el shuttle destino durante el envío de mensajes. Esto se hace consultando el ultimo juego de registros de kernel de dicho shuttle (mediante `shtl_kregref`). Empleándolo podemos salvarlo y alterarlo. El fragmento de código que realiza tal operación es como podemos ver en la figura 3.23.

```
kregs=(u_int*)shtl_kregref(p->handler);  /*get prt. to kernel state*/
/* save user kernel-level registers... */
assert(kregs);
PUSH(ksp,regs->r.x86.es);
PUSH(ksp,regs->r.x86.cs);
PUSH(ksp,regs->r.x86.ss);
PUSH(ksp,regs->r.x86.ds);
...
PUSH(ksp,*kregs);
*kregs=ksp;                          /* record the new kern jmpbuf */
```

Figura 3.23: Salvando el kernel de otro shuttle

En realidad no estamos haciendo otra cosa que lo que hace la rutina `ksetjmp` durante un upcall para salvar el estado del kernel. La diferencia, es que no estamos salvando *nuestro* estado, por lo que no nos sirve de nada la mencionada rutina.

Prosiguiendo con el envío de mensajes, otro punto delicado es la instalación del registro de activación para `upcall` (la rutina que implementa un upcall) en la pila de kernel del shuttle destino una vez que hemos salvado su estado. De nuevo, lo construimos a mano tal y como se ve en la figura 3.24. Puede verse como éste es el punto en que el mensaje se transfiere del remitente al destinatario. Una vez construido, se altera el estado del shuttle destino para que ejecute el manejador.

La parte crítica de la realización de un upcall es la llamada a área de usuario.Ésta se realiza como puede verse en la figura 3.25, la realiza la rutina `prtl_upcall_up` una vez que `upcall` ha salvado el estado mediante `ksetjmp`.

```
/* xfer message to the callee */
PUSH(ksp,flags);                    /* msg mflags */
PUSHLONGLONG(ksp,sender);           /* sender shuttle */
PUSHLONGLONG(ksp,p->id);            /* portal id */

/* upcall activation frame -- message (m), # of words (n), hndlr, saved eip
 */
{ int i;
  for( i=0, m=(((int*)m)+n-1); i<n; m = ((int*)m)-1, i++)
     PUSH(ksp,*((int*)m));
  PUSH(ksp,n);
}
PUSH(ksp,p->pc);                   /* handler returns to ret_from_portal*/
                                   /* to issue a sys_portal_ret */
/* Here we go */
regs->r.x86.esp=regs->r.x86.ebp=(u_int)ksp;
regs->r.x86.eip=(u_int)prtl_upcall;
```

Figura 3.24: Construcción del registro de activación para upcall.

3.5 Gestión de Memoria Virtual en Off: DTLBS

En los sistemas distribuidos existentes, los gestores de memoria virtual no permiten que a la vez que se comparten los recursos de memoria disponibles se mantenga la adaptabilidad y extensibilidad [61, 86].

Aunque se han producido avances en gestores de memoria distribuida [36, 37, 24, 98, 97], los gestores existentes en la actualidad aún presentan una arquitectura rígida. Salvo en unos pocos gestores de memoria *centralizados* (como AVM [61]), no es posible que las aplicaciones tomen el control y adapten la gestión de memoria del sistema para satisfacer sus necesidades. Aún más, cuando la gestión de memoria suministrada por el sistema resulta del todo inadecuada, ésta no puede obviarse ni ser reemplazada.

En *Off* se utiliza la DTLB como base para implementaciones en área de usuario de gestores de memoria virtual distribuida (DVM[24]) y memoria distribuida compartida (DSM[25]). La DTLB constituye un mecanismo básico que permite implementar en área de usuario distintos modelos de gestión de memoria, suministrando una visión distribuida del hardware existente (y ello sin imponer un único espacio de direcciones para todo el sistema, al contrario que en sistemas como Angel [175, 176]).

Sobre las DTLBs de *Off* se ha implementado una arquitectura para construir gestores de memoria virtual distribuida, llamada Advice [12]. Este gestor de memoria aprovecha la flexibilidad de *Off* para permitir la implementación de sistemas de objetos distribuidos de un modo más simple que en sistemas más tradicionales [11].

En este apartado detallaremos el diseño y realización de la DTLB suministrada por *Off*, o más precisamente, del gestor de memoria distribuida (o DMM[26]) que es el servidor del sistema encargado de implementarla. Antes de ello veremos algunos otros enfoques en gestión de memoria para sistemas distribuidos.

---

[24] Distributed Virtual Memory
[25] Distributed Shared Memory
[26] Distributed Memory Manager

```
    .globl prtl_upcall_up
prtl_upcall_up:
    addl  $4,%esp                     # pop saved eip
                                      #   stack has now: eip,cs,eflags,esp,ds
                                      #   which looks like an interrupt frame
    xorl %eax,%eax
    movw $0x4b,%cx                    # reset user DS and ES
    movl %cx,%ds
    movl %cx,%es
    cld                               # clear direction flag (GCC's happy w/)
    iret                              # restore such an "interrupt" frame.
                                      # this should also reenable interrupts
```

Figura 3.25: Llamada de kernel a usuario.

3.5.1 Otros enfoques en gestión de memoria

Espacios de direcciones en sistemas tradicionales

En sistemas tradicionales como puede ser UNIX, el SO suministra como abstracción principal el *espacio de direcciones virtuales*. Éste es sencillamente lo que una aplicación percibe como dominio de protección. Una vez suministrada esta abstracción, el sistema permite entonces realizar *proyecciones* sobre la misma de tal modo que es posible establecer correspondencias entre el contenido de ciertos “objetos” (por ej.: ficheros) y regiones de memoria dentro de dicho espacio.

Estos objetos que constituyen la fuente de la memoria que puede proyectarse sobre un espacio de direcciones se denominan *objetos de memoria* y, en general, pueden ser ficheros, segmentos de memoria o bien espacio en área de swap.

Algunos sistemas distribuidos [96, 180] han optado por distribuir éste modelo, de tal modo que sea posible establecer proyecciones entre objetos de memoria y espacios de direcciones con independencia del nodo en que se ubiquen.

Naturalmente, la semántica de la distribución de la memoria (principalmente, la semántica de la compartición de memoria) está *cableada* en el sistema, por lo que no es adecuada para *todas* las aplicaciones.

Espacios de direcciones únicos

Con el advenimiento de arquitecturas de 64 bits, capaces de soportar espacios de direcciones de tamaño considerable, algunos SO han optado por distribuir la gestión de memoria de un modo peculiar: empleando un único espacio de direcciones que abarca la memoria presente en *todos* los nodos del sistema.

La principal ventaja consiste en la validez de las referencias a memoria en todo el sistema, con independencia del nodo en que estén. No obstante, se introducen nuevos problemas como la necesidad de introducir en el sistema nuevas técnicas de protección para aislar a unas aplicaciones de otras (dado que todas ellas comparten el espacio de direcciones ahora) y problemas relacionados con la carga de aplicaciones en memoria (ahora no es factible hacer que la memoria virtual resuelva el problema de la reubicación). Por otro lado, la *inflexibilidad* del sistema es absoluta en cuanto a gestión de memoria se refiere. Hay que atenerse al modelo de único espacio de direcciones o no es posible trabajar en el sistema.

Paginadores externos

Un avance en el sentido de adaptabilidad en gestión de memoria lo constituyen los *paginadores externos*. Éstos son procesos en área de usuario que coordinan los aspectos de sincronización y transferencia de datos entre el espacio de direcciones (implementado por el SO) y el objeto de memoria del que proceden los datos.

La ventaja de los paginadores externos radica en que habitualmente es posible establecer la semántica de compartición mediante dichos elementos. El SO consulta habitualmente al paginador antes de efectuar ciertas operaciones y ésta es la ocasión para ejecutar código que implemente una semántica de compartición dada. Por ejemplo, Spring [97] y Mach [24, 180] hacen uso de éste mecanismo.

A pesar de sus ventajas, en los sistemas que disponen de paginadores externos, la abstracción "espacio de direcciones" reside todavía dentro del núcleo. Las implicaciones son obvias: la gestión de la memoria física y parte importante de las políticas involucradas en gestión de memoria (más notablemente, la de reemplazamiento de páginas) está contenida en el μkernel; él decide por tanto *qué* páginas hay que expulsar y cuándo deben expulsarse[27]. Él decide las estructuras de datos y las operaciones que deben constituir un "espacio de direcciones".

Memoria virtual de aplicación

Paradójicamente, el sistema de memoria virtual existente que más se parece al que suministra *Off* es el de un sistema centralizado. Hablamos de AVM [61], el sistema de gestión de memoria del exokernel.

En AVM el SO tan sólo suministra lo necesario para que las aplicaciones establezcan sus propias traducciones de memoria virtual. Éstas deben solicitar memoria física y emplear las facilidades del sistema para instalar sus traducciones. En pocas palabras, las abstracciones empleadas están completamente en área de usuario.

El problema ahora es que, dado que el sistema es centralizado, no disponemos de ninguna facilidad para operar en un sistema distribuido. En *Off* hemos adaptado este modelo a un sistema distribuido. El resultado es una abstracción, la DTLB, muy próxima en su nivel de abstracción al hardware que permite el empleo de recursos de memoria presentes en la red para implementar en área de usuario DVMs y DSMs.

3.5.2 Las DTLBs de *Off*

La única tarea que realiza *Off* es exportar el hardware presente en la red, y ésto también se aplica en gestión de memoria. El DMM de *Off* implementa una TLB software distribuida o DTLB. Una DTLB es una caché de traducciones de páginas de memoria virtual a marcos de memoria física distribuida. Su estructura es similar a la de AVM [61]; el usuario puede establecer traducciones de memoria virtual a memoria física. La diferencia radica en que las direcciones de memoria física (como ya vimos al comienzo del capítulo) no sólo pueden referirse a memoria local sino también a memoria remota (ver figura 3.26). Esta abstracción es la base para la implementación de cualquier sistema de memoria virtual distribuido que emplee el SO que ejecute sobre el μkernel.

Una DTLB (ver figura 3.26) contiene un conjunto de traducciones de la forma $v \rightarrow p{:}o$, donde v es una dirección virtual y $p{:}o$ es un identificador de marco de página (que como vimos puede corresponder a un marco local o remoto).

[27] Aunque existen algunos derivados de Mach que permiten que el paginador escoja qué páginas hay que expulsar, el problema persiste: la implementación de los espacios de direcciones está prefijada dentro del núcleo y no es posible alterarla cuando es necesario.

La DTLB emplea el hardware de traducción de direcciones como base para su implementación. Las traducciones de la DTLB que tienen como destino un marco de página local (por ejemplo, *v1* $\rightarrow$ *p:01* en la figura 3.26) tendrán un reflejo en la tabla de páginas empleada por el hardware (*v1* $\rightarrow$ *m1* en la figura). Las traducciones hacia marcos remotos requieren además de operaciones implementadas en software para su correcto funcionamiento. Aquellas traducciones hacia marcos remotos que estén utilizándose en un instante determinado deberán emplear, naturalmente, marcos locales como cache de los remotos (en la figura, el marco local *m3* actúa como cache de *q:03*, de tal modo que la traducción *v3* $\rightarrow$ *q:03* de la DTLB tiene como reflejo la traducción *v3* $\rightarrow$ *m3* en la tabla de páginas).

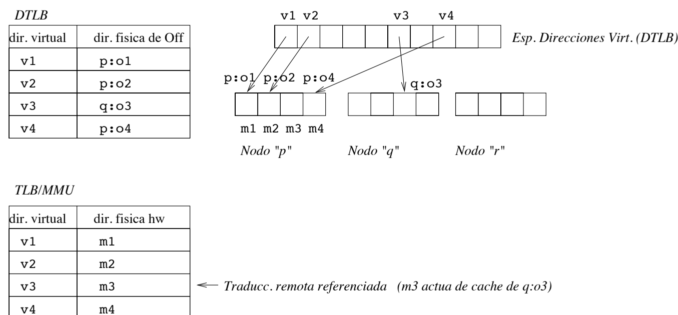

Figura 3.26: La DTLB. Las traducciones permiten emplear memoria remota.

Dado que las traducciones de que consta la DTLB no emplean identificadores que pudieran perder su significado fuera del nodo local (las direcciones de los marcos son válidas en toda la red) es factible emplear una DTLB dada desde cualquier nodo del sistema. Esto facilitaría la migración de espacios de direcciones construidos sobre las DTLBs de *Off*.

Las aplicaciones suministran los algoritmos que implementan los protocolos de coherencia y transporte de datos (necesarios para emplear memoria distribuida). En efecto, el DMM se limita a instalar las traducciones y elevar excepciones (mediante upcalls) para requerir la ayuda de la aplicación.

El interfaz suministrado por el DMM es en realidad extremadamente simple, como puede verse en la figura 3.27 (donde hemos omitido los puntos de entrada correspondientes a `freeze` y `melt`). En tiempo de creación de la DTLB se establece el portal al que deben redirigirse las excepciones (upcalls) de la misma, `prtl`. Una vez creada, las traducciones se instalan con `dmm_tlb_install` y las protecciones se pueden cambiar con `dmm_tlb_prot`.

```
dtlb_id_t dmm_tlb_creat(prtl_id_t prtl, off_guty_t *dtlb_guty);
int dmm_tlb_destroy( dtlb_id_t dtlb, off_guty_t *dtlb_guty);
int dmm_tlb_install(dtlb_id_t dtlb,
                    vaddr_t vaddr[], paddr_t paddr[],
                    prot_t prot, int n,
                    off_guty_t *dtlb_guty,off_guty_t*paddr_guty);
int dmm_tlb_invalidate(dtlb_id_t dtlb, vaddr_t vaddr[], int n,
                       off_guty_t *dtlb_guty);
int dmm_tlb_prot(dtlb_id_t dtlb, vaddr_t vaddr[], prot_t prot[], int n,
                 off_guty_t *dtlb_guty,off_guty_t *paddr_guty);
```

Figura 3.27: Principales puntos de entrada del DMM

A pesar de que la DTLB se inspira en la TLB software de Aegis [60], la implementación ha tomado un camino diferente. En Aegis, una única TLB software (de 1024 entradas) se multiplexa entre las aplicaciones, de tal modo que éstas asignan y liberan entradas de dicha tabla. Esta TLB software emplea la totalidad del hardware de traducción de direcciones, con lo que éste no se multiplexa directamente entre las aplicaciones. El DMM de *Off* implementa múltiples DTLBs (no una única) que se multiplexan sobre el único hardware de traducción de direcciones disponible. Consiguientemente, una DTLB no se multiplexa en realidad; cada aplicación posee la suya propia.

Las razón principal que nos llevó a optar por múltiples DTLBs multiplexadas sobre el hardware (y no a una única multiplexada entre las aplicaciones) fue la consideración de arquitecturas con tablas de páginas. En Aegis la implementación se efectuó sobre el procesador MIPS, que sólo dispone de TLB (su hardware no utiliza tablas de páginas, siendo éstas gestionadas mediante software). Esta TLB emplea identificadores de contexto de tal modo que es posible seleccionar fácilmente las entradas que corresponden a una aplicación determinada, resultando trivial su multiplexación.

Si consideramos ahora arquitecturas con tablas de páginas, resulta más simple emplear una tabla de páginas para cada aplicación que multiplexar una única tabla de páginas entre distintas aplicaciones. Ésto ha determinado el empleo de múltiples DTLBs.

3.5.3 Uso de memoria remota

Dado que el DMM opera con direcciones de marcos que no tienen por qué ser locales, podemos emplear memoria remota en nuestras traducciones. En este sentido, hay algunas implicaciones que resultan importantes para la implementación de cualquier sistema que emplee la DTLB.

En primer lugar, cualquier entidad que implemente el interfaz de un gestor de memoria física puede suministrar marcos de memoria que pueden luego emplearse como destino de las traducciones de una DTLB. Téngase en cuenta que el usuario del sistema solicita la asignación de marcos y emplea los identificadores de éstos para la instalación de traducciones en su DTLB. Todo lo que el DMM necesita para instalar una traducción en una DTLB es la dirección de la página que se traduce (ej.: *v3*) y el identificador del marco al que se traduce (ej.: *q:o3*). Dado que el identificador de un marco de página contiene el identificador de portal del gestor de memoria física donde está ubicado (ej.: *q*), el DMM tiene toda la información necesaria para gestionar la traducción.

Cuando se instala una traducción de una página a un marco remoto y se referencia dicha página, el DMM eleva una excepción (mediante un upcall al portal de excepciones indicado por la DTLB). El propósito de dicha excepción es requerir del usuario la copia del marco remoto en el nodo local. El DMM indica el origen y destino de la traducción considerada y se espera que el usuario indique cuál es el marco local que debe emplearse como cache del remoto. Las aplicaciones pueden escoger entre emplear gestores de memoria virtual distribuida de propósito general (estableciendo el portal de alguno de ellos como portal de excepción de la DTLB) o emplear sus propios gestores de memoria virtual distribuida.

En segundo lugar, *Off* no emplea revocación implícita (de hecho *Off* no revoca recursos como vimos en el capítulo anterior), con lo que las aplicaciones tienen más oportunidades para controlar la asignación y revocación de sus marcos. Concretamente, *Off* no revoca derechos sobre marcos de página a las aplicaciones. Ésto es tarea del SO implementado sobre *Off*. Las aplicaciones pueden escoger los marcos que deben liberar (bajo la supervisión de un árbitro como vimos en el apartado 2.1.5).

Por todo lo dicho, es posible operar con memoria local, memoria remota, memoria procedente de disco y memoria cuyo contenido se construye en demanda de un modo homogéneo, con lo que se simplifica la realización de sistemas de memoria virtual en área de aplicación. En los apartados que siguen veremos algunos ejemplos.

Memoria en disco y memoria sintetizada

Un servidor que haga uso de un disco (de un manejador de disco, en realidad) para suministrar el interfaz del gestor de memoria física del sistema puede emplearse para automatizar o simplificar la paginación a disco y la implementación de sistemas persistentes. Llamaremos a este servidor un gestor de memoria física de disco.

Si una aplicación desea que parte de su espacio de direcciones tenga como respaldo determinada área en disco, basta con emplear marcos de página que procedan de un gestor de memoria de disco. Como ejemplo, en la figura 3.28 podemos ver una aplicación de usuario que podría realizar peticiones de asignación de marcos a distintos gestores de memoria (uno local, uno remoto y uno de disco). Una vez obtenidos dichos marcos podría instalar traducciones hacia ellos en su DTLB con independencia de su lugar de procedencia.

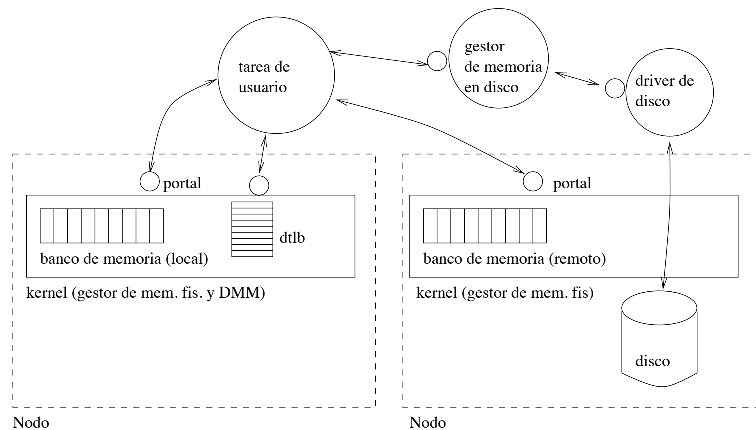

Figura 3.28: Gestor de memoria de disco

En realidad, dado que los protocolos empleados para obtener caches de marcos remotos también los suministra la aplicación, no sólo es posible emplear "marcos" de página procedentes de disco sino también de cualquier otra entidad con tal de que ésta implemente el interfaz del gestor de memoria física. Un uso típico es la implementación de un gestor de memoria física que sintetice el contenido de los marcos. Por ejemplo, podríamos imaginar un servidor que presenta un interfaz similar al del gestor de memoria física y (en lugar de mantener almacenado en algún soporte el contenido de los marcos que suministra) suministra marcos rellenos con ceros cada vez que se le requiere un marco. Este (falso) gestor de memoria física se puede emplear para inicializar el contenido de otros marcos. Otro uso posible sería el empleo de un gestor de memoria que emplease algún algoritmo de compresión y/o algún algoritmo de encriptación para reducir las necesidades de almacenamiento de respaldo y/o para proteger el área de almacenamiento de respaldo.

Por último, hay que considerar que la "aplicación" que trata las excepciones de la DTLB e implementa por tanto las políticas empleadas por la misma no tiene por qué ser la misma aplicación que utiliza la DTLB para implementar su espacio de direcciones. Es factible la implementación de servidores que realicen estas tareas y, debido al empleo de portales, también es factible la inclusión de estos servidores dentro del núcleo. Tenemos, pues, un amplio abanico de posibilidades a la hora de diseñar DVMs y DSMs.

3.5.4 Implementación en el Intel x86

La DTLB de *Off* es en realidad sólo una cache de traducciones. Aunque en arquitecturas con tablas de páginas todas las traducciones locales estarán almacenadas en dichas tablas, el resto de las traducciones está almacenado en una tabla de traducciones remotas, y está sujeto a revocación. Al igual que sucede en sistemas tradicionales, los marcos de página empleados para almacenar las tablas de páginas están asignados a las aplicaciones que los utilizan (no son marcos utilizados por el kernel *para el kernel*, son marcos empleados por las aplicaciones aunque sea el kernel el único que puede acceder a ellos). Dicho de otro modo, son responsabilidad de la misma entidad que es responsable de la DTLB (cuya tabla de páginas contienen).

La tabla de traducciones remotas se emplea para dos propósitos:

1. almacenar aquellas traducciones que no pueden estar situadas en la tabla de páginas[28] (traducciones remotas) y

2. mantener una correspondencia entre marcos remotos y marcos locales que están haciendo caching de ellos (para evitar un envío innecesario de excepciones, ya que si existe un marco local que sea cache de un marco remoto éste se puede emplear como destino de más de una traducción).

Así pues, las traducciones instaladas en la DTLB pueden estar en uno de tres estados (ver figura 3.29):

- *Traducciones locales*. Están presentes sólo en las estructuras (tablas de páginas, TLBs) que emplea el hardware, para evitar duplicidad de información (por ejemplo, en la figura vemos como la traducción de *w* a *p:n*, siendo *p* el portal del gestor local de memoria física sólo mantiene una entrada en la tabla de páginas). Esto es así tanto si son válidas y tienen un marco asociado como si no.

  Cuando se instala, desinstala o cambia la protección de una traducción hacia un marco local en la DTLB, se emplea el offset (*n* en el ejemplo) que forma parte del identificador del marco involucrado (*p:n* en el ejemplo) como dirección de memoria física. Funciones dependientes de arquitectura actualizan entonces la tabla de páginas en función de la dirección virtual suministrada por el usuario, la dirección física y la operación realizada.

- *Traducciones remotas no referenciadas*. Están presentes en la tabla de traducciones remotas, y tienen la entrada correspondiente de la tabla de páginas invalidada (por ejemplo, la traducción de *z* a *s* en la figura). En este caso, el campo de la entrada de la tabla de páginas que se emplea para identificar el marco de página (PFN[29]) de la traducción se utiliza para indexar en la tabla de traducciones remotas (índice *i* en la figura). De este modo, ante un fallo de página es posible encontrar con rapidez la entrada correspondiente en dicha tabla.

  La instalación de estas traducciones (remotas) consiste pues en:

  1. Asignar una entrada en la tabla de traducciones remotas para el marco de página remoto, destino de la traducción. Si ya hubiese una entrada para dicho marco remoto, se reutiliza ésta (se emplea un contador de referencias para poder reutilizarla).

  2. Si el marco remoto no ha sido referenciado no tendrá copia en el nodo local, por lo que en la entrada asignada en el punto anterior no habrá ninguna dirección de marco local

[28] De aquí en adelante usaremos "tabla de páginas" para referirnos a la estructura de datos empleada por el hardware, con independencia de que ésta sea una auténtica tabla de páginas, una TLB o una combinación de ambas.

[29] Page Frame Number.

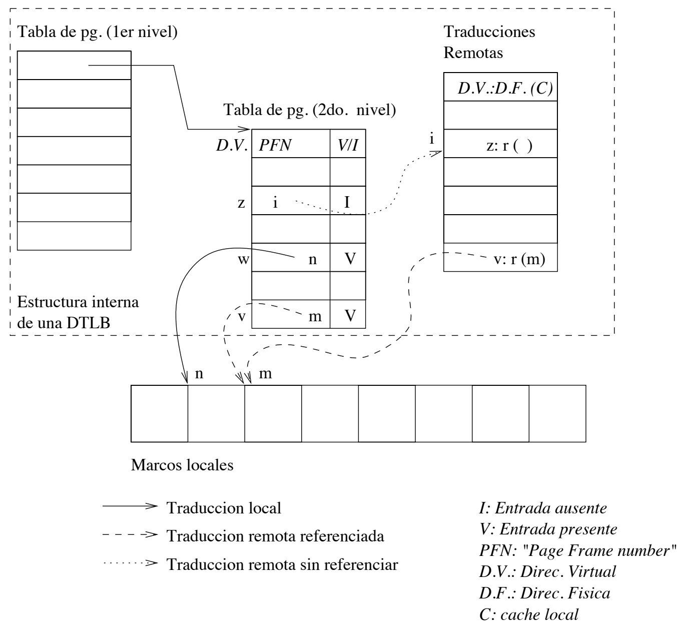

Figura 3.29: Estructura de la DTLB

(cache) asignado. En este caso se rellena la entrada correspondiente a la dirección virtual a traducir en la tabla de páginas para que su PFN corresponda con el índice de la entrada asignada en el punto anterior (*i* en la figura). Naturalmente, la entrada de la tabla de páginas se mantiene como inválida.

- *Traducciones remotas referenciadas*. Éstas están presentes en la tabla de traducciones remotas y tienen un marco de página local asignado de tal modo que éste hace cache del marco de página remoto destino de la traducción. La entrada correspondiente en la tabla de páginas está validada y contiene el número de marco de página local que se está empleando. (Por ejemplo, en la figura vemos cómo la traducción de v a r tiene en la tabla de traducciones remotas una indicación de que el marco local m se emplea como cache de r; la tabla de páginas tiene también una traducción válida de v a m).

  Cuando se referencia un marco remoto por primera vez, se eleva una excepción (mediante un upcall) que deberá tratar la aplicación. El propósito de la misma es asignar un marco de página local y realizar una copia del marco remoto. Una vez que la copia está hecha, la aplicación devuelve el control al μkernel. En este punto se anota en la entrada de la tabla de traducciones remotas la dirección del marco local que se emplea como caché del marco remoto y se valida la entrada de la tabla de páginas haciendo que ésta apunte al marco local.

  Cuando se instalen otras traducciones hacia el marco remoto (ya referenciado) se rellenarán otras entradas (correspondientes a dichas traducciones) en la tabla de páginas cuyo destino será el marco local que ya estaba empleándose como cache.

Fallos de página

Cuando sucede un fallo de página, la rutina `shtl_trap_handler` lo recibe justo después de su ocurrencia. Ésta se limita a redirigir el trap hacia el portal especificado por la aplicación para su

tratamiento. En este caso, el DMM habrá instalado un manejador para el suceso de fallo de página de tal modo que la rutina dmm_pg_fault recibirá el fallo de página siempre que éste ocurra.

Esta rutina extrae la información útil suministrada por el hardware (dirección virtual del fallo y causa del mismo) y ejecuta un algoritmo como el que se puede ver en la figura 3.30.

```
static int dmm_pg_fault(struct trap_state *ts)
{
  int faulting_maddr=faulting address from ts;
  int reason=error code from ts;
  if(error code says it is a kernel page fault)
    return 1;                          /* panic -- not implemented */
  if(absent page with PFN (i.e. invalid remote entry present)){
     get remote translation  table entry using the PFN as an index;
     if (!cache present){
       upcall to the user to:
          allocate page;
          start page copy;
     }
     reinstall cache on page table;
     return 0;
  }
  else { /* A real page fault --- Invalid translation/protection */
     forward page fault to the user (dtlb exception portal) with an upcall;
     return 0;
  }
}
```

Figura 3.30: Fallo de página

Como puede verse, en cualquiera de los casos que requieren tratamiento (entradas remotas no referenciadas, entradas inválidas y violación de protecciones) se eleva un upcall al usuario. Es éste el que toma el control. Nótese que el upcall se envía a un portal que podría estar servido por cualquier aplicación.

Puesto que el DMM no se encarga en modo alguno ni del control de concurrencia ni del suministro de protocolos de coherencia de memoria, es obvio que es la aplicación la que debe contemplar estos aspectos. Habitualmente, el usuario utilizará abstracciones de más alto nivel que deberán solventar estos problemas del modo más adecuado para la aplicación.

Como ejemplo de lo que acabamos de decir, un servicio de memoria distribuida compartida (implementado a nivel de aplicación) que suministrase una semántica de compartición estricta podría restringir las protecciones de las traducciones instaladas (en las DTLBs involucradas) para suministrar una semántica de un-escritor/múltiples lectores (bastaría con asegurar que o bien todas las traducciones son de sólo de lectura o bien existe una traducción de lectura/escritura y el resto están invalidadas).

Otro ejemplo sería el de aplicaciones que trabajasen cooperativamente (confiasen las unas en las otras) de tal modo que cada una *avisase* al gestor de memoria distribuida compartida antes de leer o escribir en una zona de memoria determinada. En éste caso no sería necesario alterar las traducciones y bastaría con copiar en demanda aquellas zonas de memoria que son escritas, de suerte que el resto de las aplicaciones puedan leer los datos escritos tras avisar al gestor de memoria de la lectura que van a efectuar.

Capítulo 4

Otros trabajos relacionados

Both models are identical in performance, functional operation, and interface circuit details. The two models, however, are not compatible on the same communications line connection.

—*Bell System Technical Reference*

A lo largo de la exposición realizada en los capítulos anteriores hemos ido mencionando aquellos aspectos de otros sistemas que tenían relación con el tema tratado en cada caso. Hemos tratado de eliminar en la medida de lo posible un capítulo "tradicional" de exposición del estado del arte por creer que resulta más útil al lector una sucesión de comparaciones puntuales (ya efectuadas en los capítulos anteriores) que el establecimiento de una comparativa separada por completo de la exposición del trabajo realizado entre éste y otros trabajos enmarcados en el área de investigación abordada. No obstante, ya expuesto el modelo de construcción de SSOO basados en DAMN y el diseño e implementación del DAMN *Off*, es conveniente comparar otros trabajos (considerados en su conjunto) con el aquí expuesto. Ahora es posible hacerlo sin oscurecer la comparación con detalles puntuales, puesto que éstos ya han sido tratados anteriormente.

4.1 Distribución, adaptabilidad y DAMNs

La diferencia fundamental entre *Off*[1] y otros sistemas radica en la combinación entre distribución y adaptabilidad. Como puede verse en la figura 4.1, la investigación en SSOO se ha centrado bien en adaptabilidad, bien en distribución. Esto se debe a que los sistemas existentes centran su atención en *uno* de los dos aspectos. Aunque recientemente han aparecido algunos sistemas que exploran ambas [135, 169, 170, 175], *Off* permite niveles de adaptabilidad comparables a los del exokernel [62] manteniendo una transparencia en la distribución comparable a la de sistemas distribuidos como Amoeba [121, 161]. En realidad, *Off* suministra facilidades de cara a la distribución del sistema que no están presentes en sistemas distribuidos como el que hemos mencionado—principalmente la posibilidad de emplear físicos remotos. Podemos decir que *Off* es un μkernel pionero en este sentido.

El presente capítulo se basa en la figura 4.1 para mencionar otros sistemas y compararlos con el propuesto. En primer lugar, en el apartado 4.2 veremos algunos sistemas que podríamos considerar

[1] En lo que sigue utilizaremos *Off* y DAMN indistintamente, puesto que nos estaremos siempre refiriendo al modelo del sistema propuesto.

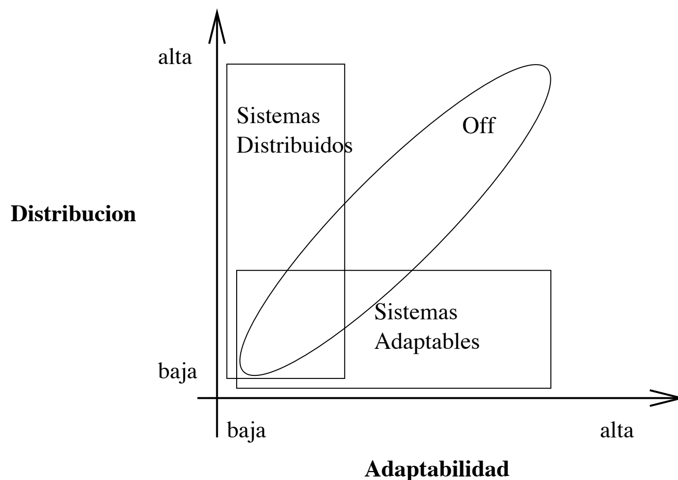

Figura 4.1: Distribución y Adaptabilidad en SSOO

como distribuidos y adaptables y los compararemos con *Off*. Seguidamente, en el apartado 4.3 veremos qué adaptabilidad presentan algunos sistemas distribuidos. Por último, el apartado 4.4 toma el enfoque opuesto y considera la distribución en algunos sistemas adaptables.

4.2 ¿Sistemas distribuidos adaptables?

Existen dos formas de suministrar adaptabilidad en SSOO [39]:

1. Implementando en el μkernel (o exokernel) sólo los mecanismos básicos imprescindibles, de tal modo que la adaptación del SO se realiza en realidad *extendiendo* los servicios del núcleo.

2. Descargando código (o modificando dinámicamente código) dentro del μkernel, de tal modo que sea factible adaptar el funcionamiento del sistema.

Aunque existen algunos sistemas que podrían calificarse como distribuidos y adaptables [169, 7, 140, 175, 167], ninguno de ellos se aproxima a las cotas de adaptabilidad encontradas en sistemas como SPACE [135], el Cache Kernel [34] o el Exokernel [59, 60], que adoptan el segundo modelo de los mencionados arriba. Baste un ejemplo: en ninguno de ellos es factible la adaptación de la revocación de recursos de tal modo que las aplicaciones tengan un control *completo* sobre la misma; todos ellos incluyen la abstracción "espacio de direcciones", con lo que no es factible que las aplicaciones que lo deseen construyan la suya propia.

Como ya dijimos en el capítulo 1, la razón principal estriba en que dichos sistemas no se limitan a suministrar los mecanismos mínimos indispensables para que sea posible la realización de aplicaciones distribuidas de un modo seguro y eficiente. En cambio, la mayoría de estos sistemas "distribuidos y adaptables" están basados en realidad en μkernels cuyos servicios presentan un nivel de abstracción relativamente elevado. Por ejemplo, incluyen espacios de direcciones y/o procesos y/o manejadores de dispositivo.

Dado que todos estos sistemas son en realidad sistemas distribuidos basados en μkernels centralizados que, a nuestro juicio, más que de "adaptables" podrían calificarse de "extremadamente flexibles", los discutiremos en el apartado 4.3.2 junto a otros sistemas distribuidos basados en μkernel.

4.3 Adaptabilidad en SSOO distribuidos

En este apartado discutiremos la adaptabilidad de SSOO que, claramente, son SSOO distribuidos.

4.3.1 Sistemas monolíticos

Sorprendentemente, algunos de los SSOO distribuidos más flexibles están basados en núcleos monolíticos. Destacan sistemas como *Plan 9* [129, 133] e Inferno [52]. Ambos sistemas siguen la línea de construcción de SSOO simples y elegantes de que han hecho gala algunos de sus autores desde la aparición de UNIX.

Existen otros muchos sistemas distribuidos monolíticos que podríamos haber mencionado en este apartado sin que por ello se introdujesen nuevos elementos de juicio en la discusión. En todos los sistemas de este tipo que podemos encontrar en [23] (Clouds [150], LOCUS [131], MOSIX [1] y otros muchos) hallamos ventajas e inconvenientes similares a los que mencionamos a continuación. Hemos tratado, en cualquier caso, de seleccionar los más representativos e ilustrativos para el trabajo discutido.

Sprite

Sprite [128] es un sistema distribuido que presenta una imagen de sistema único (tipo UNIX) a los procesos que en él operan. La distribución queda *contenida* dentro del núcleo: los usuarios de Sprite creen usar una variante más de UNIX. Aunque el núcleo es monolítico y es prácticamente imposible reemplazar o adaptar aquellas partes del sistema en que fuese deseable hacerlo, Sprite incorpora algunas ideas que han inspirado parte del diseño de *Off*.

Considerando Sprite en su conjunto, este sistema hace uso intensivo de su sistema de ficheros distribuido para suministrar una imagen de sistema único (por ejemplo, para distribuir la gestión de procesos se emplean ficheros—distribuidos[2]—de respaldo que contienen el estado de los mismos). Al contrario, en *Off* no tenemos ninguna *abstracción* central sobre la que se distribuya el resto del sistema.

Un aspecto interesante de Sprite es que el estado de los procesos está repartido entre los distintos módulos que integran el núcleo [54, 53]. El propósito de tal enfoque fue evitar que la modificación o inclusión de facilidades del sistema rompiese inadvertidamente la capacidad del mismo para migrar procesos entre nodos. La *modularización* del estado de los procesos hace que la inclusión de nuevas facilidades pueda (hasta cierto punto) independizarse de las ya existentes. Aunque en *Off* las propiedades de los Shuttles no tienen como único fin facilitar la migración de los procesos construidos sobre éstos, lo que persigue la introducción de propiedades es esencialmente lo mismo que perseguía la modularización del estado de los procesos en Sprite: independizar entre si la implementación de los recursos empleados por los procesos del sistema.

Plan9 e Inferno

La idea básica de ambos sistemas [129, 133, 52] es el suministro de una única abstracción: el fichero. En ellos *todos* los recursos se modelan como si de ficheros se tratase. El empleo de un protocolo de comunicaciones que permite emplear ficheros de forma remota hace posible la distribución del sistema.

El problema que encontramos radica en el hecho de que éstos sistemas están forzando el uso de la única abstracción suministrada. Por otro lado, aspectos como la gestión de procesos y la gestión

[2] Más correctamente, ficheros pertenecientes al sistema de ficheros distribuido que permiten interacciones simultaneas desde distintos nodos.

de memoria están completamente cableados dentro del núcleo. Por consiguiente, no es factible su adaptación. Si tenemos en cuenta que precisamente son estos dos aspectos los que permiten alcanzar mejores incrementos en rendimiento mediante la adaptación de los mismos [61], puede entenderse como ventajoso el empleo de un SO que sea realmente adaptable.

Solaris-MC

Solaris MC [96] es una extensión del núcleo de Solaris para operar en cluster.Éste sistema incorpora ideas procedentes de Spring [84] (sistema que discutiremos en el apartado 4.3.2), aunque mantiene la estructura de núcleo monolítico no adaptable que presenta UNIX.

4.3.2 Sistemas basados en μkernel

Como se mencionó al principio del presente trabajo, en el capítulo 1, el uso de μkernels nos acerca a sistemas operativos distribuidos y adaptables [158]. En este apartado discutiremos sobre algunos SSOO distribuidos que están construidos a partir de un μkernel. Todos ellos son sistemas flexibles, incluso algunos podrían calificarse de “adaptables” al menos en cierto sentido, dado que ya con el empleo de un μkernel se consigue que gran parte de los servicios del sistema queden implementados fuera del núcleo, con lo que la adaptación de estos es casi siempre[3] factible y, cuando lo es, más simple.

Además de los aquí mencionados, existen multitud de SSOO basados en μkernels cuya discusión hemos omitido puesto que una discusión seria de *todos* los existentes requeriría no uno, sino varios libros, dedicados sólo a tal fin. Algunos de estos sistemas se especializan en operación en tiempo real, como Maruti [48], Spring[4] [151], RT-Mach [165], ARTS [82], QNX [88], etc. Otros tratan de suministrar servicios básicos para sistemas de objetos distribuidos, como Guide [40, 16], Tigger [26], Nanos [76] o Apertos [177, 156]. Otros como Taos [132] introducen técnicas novedosas como la compilación en demanda para tolerar sistemas heterogéneos. Lamentablemente, en mayor o menor medida, estos sistemas incorporan abstracciones dentro del núcleo que o no son adaptables o no pueden reemplazarse. Todos ellos presentan una estructura similar a alguno de los sistemas que discutimos a continuación.

Mach

Consideraremos en primer lugar aquellos sistemas que están construidos sobre μkernels que, por su tamaño, podrían dejar de considerarse como “micro”, lo que no resta importancia a la contribución que estos sistemas supusieron.

Probablemente sea Mach [2, 55, 24], el paradigma de μkernel pesado, aunque también habría que considerar como tal a Windows NT [47]. Hay que decir que a pesar de ser un μkernel bastante pesado y monolítico en su concepción, Mach es un pionero de la técnica de construcción de SSOO mediante μkernels y supuso una innovación en el campo.

En Mach el μkernel incluye abstracciones que otros μkernels más recientes han expulsado del núcleo: tareas (similares en realidad a los procesos de algunas versiones modernas de UNIX) y espacios de direcciones. El μkernel emplea puertos [55, 21] como mecanismo básico de IPC.

[3] En algunos sistemas se obliga a las aplicaciones a emplear determinados componentes de tal suerte que, aunque éstos estén implementados fuera del μkernel, dichos componentes no pueden adaptarse en base a la aplicación que los utiliza. Tales componentes son pues tan inadaptables como si estuviesen cableados dentro del núcleo (aunque sea más simple reemplazarlos).

[4] Un sistema de tiempo real que no hay que confundir con el sistema distribuido del mismo nombre desarrollado por Sun.

Ya comentamos en el capítulo anterior que Mach introdujo y extendió el uso de paginadores externos como técnica común para obtener más flexibilidad (y adaptabilidad) en gestión de memoria [24, 180]. Un paginador es una tarea que se responsabiliza del respaldo de determinada área de memoria en uno o varios espacios de direcciones. Su actividad principal es la ejecución de actividades de paginación entre la memoria controlada por el núcleo y los dispositivos de respaldo. Aunque en versiones iniciales el uso de paginadores no permitía a las aplicaciones participar en la política de reemplazamiento de páginas, ésto ha sido subsanado en versiones recientes de Mach.

Dado que importantes abstracciones están implementadas dentro del núcleo y que no se previó la adaptación de éstas, no es factible modificar el comportamiento de dichas abstracciones. En consecuencia, no podemos afirmar que estemos ante un sistema adaptable.

Existen otros sistemas que son en realidad modificaciones, extensiones o derivados de Mach tales como Masix [66], que no discutiremos aquí dado que lo ya dicho para Mach también es aplicable a ellos.

Chorus

Citando a su autores [140]:

> "El núcleo basado en paso de mensajes de Chorus-V3 es comparable al del kernel V [...], su DVM y threads son comparables a los de Mach [...], el direccionamiento en red toma ideas de Amoeba [...]"

En μkernel de Chorus suministra servicios básicos para implementar distintos SSOO. En particular, se emplea en la implementación de un emulador de UNIX denominado Chorus/MiX. El μkernel incluye servicios de gestión de memoria, un ejecutivo de tiempo real y servicios de IPC que incluyen paso de mensajes asíncronos y RPCs. Las abstracciones que suministra son el *Actor* que es un contenedor de recursos (similar a las tareas de Mach), Threads, Mensajes y Regiones (áreas de memoria que configuran el espacio de direcciones de un Actor).

El SO distribuido Chorus presenta, pues, problemas similares a los de Mach en cuanto a adaptabilidad se refiere. En particular, emplea contenedores de recursos (con lo que caemos en los problemas mencionados en el capítulo 1 y en el apartado 3.3) y no es factible adaptar el funcionamiento de la gestión de memoria (contenida en el núcleo) o de los threads que suministra. Por último, los puertos empleados para IPC están *anclados* en las aplicaciones que los sirven, por lo que no pueden emplearse como una *indirección* (a pesar de estar implementados como buzones que ya constituyen una indirección) entre clientes y servidores para delegar servicios, tarea fácil si un servidor pudiese *ceder* un puerto a otro servidor para que le substituya.

Spring

Otro sistema distribuido basado en μkernel es Spring [84]. Este sistema introduce nuevas técnicas basadas en el modelo de Orientación a Objetos (OO) de tal modo que los servicios del SO están compuestos por un conjunto de objetos que cooperan para implementarlos y es factible modificar los mecanismos empleados en la interacción entre dichos objetos [75]. Anteriormente, Choices [27] ya empleó la OO como base para alcanzar un SO flexible y reconfigurable.

Spring introduce innovaciones como el modelo de *subcontrato* [84] que permiten adaptar en cierta medida la intercomunicación entre los distintos objetos que constituyen el sistema (por ejemplo, hacer caching o replicación de forma transparente). A grandes rasgos, el modelo de subcontrato consiste en delegar la invocación de métodos remotos a un objeto que implementa el "subcontrato" que tiene el cliente con el servidor. Modificando éste se puede alterar la interconexión entre el cliente y el servidor.

En cuanto a los servicios suministrados por el sistema, Spring incorpora Shuttles, Threads, Espacios de Direcciones y Doors como abstracciones fundamentales del sistema, imponiéndose abstracciones *pesadas* a las aplicaciones. Discutimos éstas abstracciones a continuación.

El sistema de memoria virtual de Spring [97], que implementa los espacios de direcciones, ejecuta "fuera" del μkernel. No obstante, ejecuta con todos los privilegios del núcleo y además impone su abstracción de Espacio de Direcciones a las aplicaciones. A efectos de adaptabilidad en el sistema, la gestión de memoria puede considerarse dentro del núcleo salvo por la existencia de paginadores externos como ocurría en Mach.

El sistema de gestión de procesos, que emplea Shuttles (similares a los de *Off*, salvo por el hecho de que no poseen *propiedades* y no es posible extenderlos), oculta éstos a las aplicaciones. Éstas perciben *Threads* como abstracción básica, y estos threads están multiplexados por el núcleo sobre los shuttles existentes [84]. No es preciso decir pues que tampoco es posible reemplazar o adaptar el comportamiento y/o la implementación de estas abstracciones en Spring.

Finalmente, el mecanismo básico de IPC en Spring (las *doors*) es una adaptación del modelo de *gates* de MULTICS. Un Shuttle puede emplear una *door* para efectuar una transferencia protegida de control cambiando (posiblemente) su dominio de protección. Esta abstracción está contenida en el núcleo y se emplean *descriptores*[5] (también contenidos y gestionados por el núcleo) para exportarla a las aplicaciones. La implicación obvia es que el núcleo debe intervenir en todos aquellos casos en que una aplicación obtiene (o pierde) derecho a invocar una *door*. Como consecuencia, la implementación de la transferencia de datos entre un cliente y un servidor está predeterminada en gran medida por el núcleo (dado que éste debe intervenir y cuidar las transferencias que involucran transferencias de derechos de invocación de *doors*. La ventaja de emplear descriptores y hacer que la transferencia de éstos la realice el núcleo es que resulta muy simple emplear mecanismos de cuenta de referencias para liberar recursos que ya no se utilizan.

En pocas palabras, como hemos visto en los párrafos anteriores, Spring es un sistema distribuido que no puede considerarse como un sistema adaptable en realidad, aunque sea elegante y extremadamente flexible.

Amoeba

Amoeba [121] es un SO distribuido simple y flexible. En dicho sistema el μkernel se limita a suministrar ciertos servicios básicos y el resto de funcionalidad está implementado mediante servidores que ejecutan como tareas de usuario. Los servicios suministrados por el μkernel incluyen threads, segmentos de memoria, mecanismos de IPC (RPCs y mensajes) y E/S [161]. Podemos decir que fue un sistema innovador y, en muchos sentidos, un adelantado a su tiempo.

Amoeba distribuye no sólo los servicios "tradicionales" del sistema operativo, sino también servicios de más bajo nivel de abstracción tales como el acceso a los bloques de memoria en disco y los procesadores. En *Off* exploramos la distribución del sistema a un nivel de abstracción inferior.

En cuanto a IPC, dos de las contribuciones más interesantes desprendidas del trabajo efectuado por los arquitectos de Amoeba son (1) la combinación de un modelo de nombrado basado en capabilities con un sistema de IPC basado en RPCs que permite distribuir los servicios del sistema con grandes niveles de transparencia y (2) la sugerencia de que los mecanismos básicos de IPC empleados en SSOO distribuidos deben facilitar tanto la implementación de RPCs como la de envío de mensajes [95]. A nuestro juicio, Spring ha dado un paso atrás al considerar que *toda* la IPC entre objetos está realizada mediante RPCs. En todos aquellos casos en que no se espera respuesta del servidor estamos consumiendo recursos innecesariamente.

[5] De igual modo que en UNIX se emplean descriptores de fichero y en Mach se emplean derechos de acceso a puertos.

Lamentablemente, en la época en que se diseñó Amoeba (que fue un sistema adelantado a su tiempo, como ya pasó con MULTICS) no se valoraba demasiado la adaptabilidad como parámetro deseable en la construcción de SSOO, en parte porque había problemas más acuciantes por resolver. Así, elementos como la gestión básica de memoria (la implementación de los "segmentos") están contenidos por completo en el núcleo del sistema en Amoeba. No es factible adaptar el funcionamiento de los mismos, lo cual hace que la flexibilidad del sistema de gestión de memoria no sea muy superior a la de Spring o Mach.

En realidad el problema de Amoeba no es la falta de adaptabilidad per-se. El problema principal radica en que Amoeba se centra en *ocultar* a las aplicaciones la distribución del sistema suministrando una imagen de sistema único (por ejemplo, la asignación de procesadores se realiza de modo centralizado, eliminado cualquier posibilidad de adaptar la política de asignación de procesos a procesadores). En *Off* en cambio, aunque el sistema considera un conjunto de recursos distribuidos, las aplicaciones pueden controlar la asignación y uso de recursos para evitar la pérdida de eficiencia que ocasiona el suministro de una transparencia *total* en cuanto a distribución de recursos. Por ejemplo, en *Off* una aplicación puede asegurarse de que aquellos recursos utilizados intensivamente se encuentran siempre en el nodo local. En Amoeba ésto es difícil puesto que todo el sistema se esfuerza en ocultar la distribución a las aplicaciones, aunque sea con el loable propósito de hacerles el trabajo más fácil.

Aunque habrá que esperar a tener un SO completo y en producción operando sobre *Off* para corroborarlo, esperamos que la distribución de recursos a un nivel de abstracción inferior y la cesión de la práctica totalidad de políticas de asignación y revocación de recursos a las aplicaciones, evite esta fuente de ineficiencia en nuestro sistema.

Finalmente, el modelo de hardware contemplado por Amoeba hace que la gestión de recursos no se realice de forma *simétrica*. Amoeba especializa la operación de los nodos en el sistema de tal modo que unos están dedicados a servir como *pool* de procesadores, otros como terminales de usuario y otros como servidores de bloques de datos para sistemas de ficheros. Tratar de emplear un nodo para una tarea que no es su tarea primaria es luchar contra la concepción de Amoeba. Esquemas más radicales de aprovechamiento de recursos que contemplen todos los recursos presentes en la red por igual se ven dificultados en cierta medida por el diseño de los servicios del sistema, aunque es cierto que ello se debe principalmente a la implementación de los servidores fuera del núcleo de Amoeba y no a la arquitectura del núcleo en sí.

Paramecium

Continuando con la línea de trabajo en SSOO distribuidos, algunos de los autores de Amoeba han seguido trabajando en la implementación de otros sistemas como Paramecium [169], un sistema distribuido basado en μkernel. Paramecium emplea un μkernel que ofrece servicios de soporte para sistemas de objetos distribuidos que implementan los servicios del sistema. A diferencia de *Off*, Paramecium explora la distribución del sistema a un mayor nivel de abstracción, con lo que se pierden oportunidades para adaptar el SO a las necesidades de las aplicaciones.

En Paramecium es factible extender dinámicamente el núcleo. El mecanismo empleado es permitir la inclusión en el sistema de nuevos objetos que implementan las extensiones del mismo. Se permite que algunas de estas extensiones ejecuten en modo privilegiado como si del núcleo se tratase. Lo que recuerda a la implementación del módulo de gestión de memoria virtual en Spring.

Este enfoque no es patrimonio de Paramecium, de tal suerte que otros sistemas han adoptado la misma línea de diseño que apreciamos en Paramecium, siguiendo el modelo de sistemas distribuidos de objetos. Entre estos sistemas encontramos a Spring [84], Nanos [76], Ra [7] e incluso Clouds [150]. A diferencia de éstos, Paramecium trata de suministrar sólo lo estrictamente indispensable,

por lo que podría considerarse como un sistemas más próximo a *Off* que estos otros.

El problema en el caso de Paramecium radica en que está centrado en mecanismos de alto nivel de abstracción (p.ej.: se considera primariamente el soporte necesario para el lenguaje Orca [8]). Así, incluye soporte para replicación y otros mecanismos que no es posible adaptar. También incluye una arquitectura de nombrado y localización de objetos que deben seguir las aplicaciones que deseen utilizar el sistema.

La esencia del problema es que *todos* los mecanismos básicos de soporte para la implementación de sistemas de objetos distribuidos están incorporados en el sistema. De hecho, el sistema incorpora como abstracción básica el *objeto distribuido*: un objeto cuyo estado está particionado entre un conjunto de nodos determinado. En todos aquellos casos en que tengamos aplicaciones que no están orientadas a objetos es muy posible que estemos empleando facilidades innecesarias.

Ra

El μkernel Ra [7] es el sucesor de Clouds [150] y presenta características presentes también en Spring y Paramecium, aunque Ra fue desarrollado con anterioridad a dichos sistemas y puede considerarse un precursor de los mismos. Clouds era un sistema monolítico y su sucesor, Ra, es ahora un μkernel extensible orientado a objetos. La extensibilidad se soporta (al igual que en Paramecium) permitiendo que ciertos objetos puedan "descargarse" en el μkernel de tal modo que su código pueda ejecutar en modo privilegiado.

Ra soporta un modelo de objetos pasivo (igual que Spring) de tal modo que los Isibas (la abstracción que modela computaciones activas o flujos de control) de Ra son capaces de atravesar diferentes dominios de protección a lo largo de su vida. Tanto Ra como Spring (y otros sistemas que permiten el uso modelos de objetos pasivos) *asocian* los espacios de direcciones a los Threads del mismo modo que *Off* asocia DTLBs a Shuttles. Al contrario que en estos sistemas, en *Off* se aplica el mismo tipo de *asociación* para el resto de los recursos que necesitan los Shuttles (niveles de privilegio de ejecución, niveles de privilegio de E/S, etc.) mediante la introducción de las *Propiedades*.

Además de Isibas, Ra introduce también "espacios de direcciones" y "segmentos" como sus abstracciones principales. Los espacios de direcciones modelan los dominios de protección y los segmentos se emplean (entre otros usos) para caracterizar los distintos bancos de memoria física presentes en el sistema. Lamentablemente, la gestión de estos "segmentos" y "espacios de direcciones" está contenida en el núcleo y no es factible alterarla (aunque la OO permita escoger distintas implementaciones para dichas abstracciones en tiempo de compilación).

Angel y Mungi

A diferencia de los sistemas que hemos mencionado anteriormente, Angel [123, 122, 175] y Mungi [87, 58] aprovechan el advenimiento de arquitecturas de 64 bits para implementar un SO cuya distribución se basa en el empleo de un DSM que cubre los nodos existentes. En Angel se intenta suministrar simultáneamente un sistema distribuido y un sistema paralelo. El enfoque adoptado es el empleo de un único espacio de direcciones virtual, de tal modo que distintas regiones de dicho espacio cubren la memoria de distintos nodos. Lamentablemente, no es factible emplear modelos más convencionales (no por ello menos útiles) de gestión de memoria en este sistema ni lo es alterar el modelo de gestión de procesos.

Taos

Taos [132] introdujo una innovación en la construcción de SSOO paralelos consistente en el uso de una máquina abstracta de muy bajo nivel de tal modo que los binarios del sistema se compilaban en

demanda durante el proceso de carga desde disco. Consiguientemente, Taos es capaz de operar en sistemas heterogéneos sin incurrir en la ineficiencia que el uso de una máquina abstracta conlleva—como ocurre en el caso de Java.

Taos combina el enlazado de código con la traducción a nativo en demanda de tal modo que todo el sistema está compuesto por una serie de *nodos* (la abstracción básica en Taos) que básicamente son paquetes de datos de tamaño variable susceptibles de enlazarse entre si. Estos nodos se compilan en demanda al procesador nativo que se utilice (el código fuente en sistemas Taos se compila para un procesador virtual).

La debilidad de Taos es precisamente el punto que le da un carácter innovador: el uso de nodos y la compilación en demanda. Toda la implementación del sistema está estructurada en listas enlazadas de nodos. La gestión de memoria necesaria para el enlazado de estos nodos está contenida en el núcleo y las primitivas de IPC entre distintos objetos también están completamente contenidas en el núcleo. A pesar de toda su flexibilidad, en Taos no es posible adaptar el funcionamiento de tan cruciales abstracciones. Aun así la flexibilidad de Taos es sorprendente si tenemos en cuenta que el sistema no se diseñó con la "adaptabilidad" como meta principal.

Recientemente, Tao Systems ha anunciado el sistema Elate [155] como sucesor de Taos, aunque no parece que éste introduzca nuevos avances desde el punto de vista académico con respecto a Taos.

4.4 Distribución en SSOO adaptables

Nos centraremos ahora en Sistemas Operativos que claramente son adaptables y trataremos de ver qué facilidades suministran para la construcción de SSOO distribuidos.

Aunque pueden reducirse a dos (extensión del sistema y descarga de código en el sistema), podemos distinguir cuatro tendencias diferenciadas en la construcción de SSOO Adaptables:

- Permitir que las aplicaciones descarguen código dentro del núcleo del sistema para adaptarlo y extenderlo (ej.: Spin [18]).
- Reducir el núcleo del sistema a lo indispensable de tal modo que cada aplicación pueda reimplementar los servicios del sistema (dado que estos ya no están en el núcleo) si es necesario (ej.: Aegis [59, 60, 34]).
- Adaptar el código existente de forma dinámica en función del uso que realicen las aplicaciones del sistemas, como en Synthesis [117].
- Emplear reflexión o meta-abstracciones para modificar el comportamiento del sistema en tiempo de ejecución. Apertos [177] es posiblemente el más conocido.

4.4.1 Sistemas que descargan código en el μkernel

El primer enfoque permite adaptar el sistema sólo en el modo en que se haya previsto en tiempo de su diseño [18, 17]. La descarga de código de aplicación dentro del Sistema Operativo sólo puede efectuarse con éxito allí donde no interfiera con los servicios que *ya está* suministrando el SO. Lo que es más, las técnicas existentes para asegurar la confiabilidad del código introducido no están aún bien desarrolladas y presentan serios inconvenientes [148].

Sería factible modificar este tipo de sistemas para incorporar el modelo de sistema que proponemos. Dado que en ellos tan sólo es posible modificar aquellas componentes del SO para las que el arquitecto del mismo ha permitido la descarga de código de usuario, la incorporación del modelo de DAMN requeriría importantes cambios en la implementación de estos sistemas. Alternativamente, podría modificarse la asignación/revocación de recursos físicos (realizada dentro del μkernel en

éstos sistemas) para que el código de aplicación tuviese acceso a dichos recursos y los nombres y protecciones empleados tuviesen validez en la red.

Spin

Spin [17] permite a las aplicaciones adaptar el funcionamiento del sistema mediante el uso de *spindles*, fragmentos de código escritos en un lenguaje con tipado estricto de datos. La idea de partida radica en que para cada política empleada dentro del μkernel es posible establecer un *spindle* como implementador de la misma. Permitiendo a las aplicaciones el establecimiento de spindles dentro del μkernel es factible entonces que éstas puedan emplear sus propias políticas.

El problema de este enfoque radica en que las partes del sistema que es posible adaptar están prefijadas de antemano. Sólo es posible adaptar el funcionamiento de aquellos elementos del sistema que el arquitecto del mismo ha dispuesto como *spindles*. Los sistemas que adoptan un esquema de especialización dinámica de código (ver apartado 4.4.3) presentan problemas similares.

Cuando una aplicación ejecuta una llamada al sistema se instalan sus spindles de tal modo que, por ejemplo, un fallo de página, una operación bloqueante de E/S, etc. den lugar a la ejecución del spindle *injertado* por la aplicación en el μkernel. En realidad, se emplea un esquema de despacho dinámico[6] para escoger la implementación deseada. Éste trabaja conjuntamente con técnicas de co-ubicación (carga dinámica de código en el μkernel) para permitir la inserción de spindles en tiempo de ejecución. El despacho dinámico introduce algo de sobrecarga en la ejecución de todas y cada una de las llamadas al sistema, aunque esta sobrecarga es mínima.

La descarga de código de aplicación dentro del μkernel introduce además problemas de seguridad en el sistema. El μkernel no puede *confiar* en el código de aplicación, por lo que se requiere de técnicas orientadas a garantizar la seguridad del código que las aplicaciones introducen en el μkernel. Así se fuerza el uso de un compilador (en el que el sistema ha de confiar, con los consiguientes problemas de seguridad que ello supone) con tipado estricto de datos y modularidad; Spin usa Modula 3 [57]). Se introduce así mismo el uso de dominios de protección lógicos consistentes en espacios de nombres empleados por el μkernel de tal modo que las extensiones efectuadas por las aplicaciones tengan acotado el conjunto de recursos del μkernel que pueden referenciar.

El sistema Spin es un sistema centralizado, por lo que no trata de abordar la combinación de adaptabilidad y distribución del sistema. En principio sería factible emplear la técnica de descarga de código dentro del μkernel, como hace Spin, para adaptar el funcionamiento de un sistema distribuido, aunque incurriríamos de nuevo en el problema mencionado anteriormente.

Vino

Vino [149] adopta un enfoque similar al de Spin, dado que permite insertar código (*grafts* o injertos en Vino) en el kernel como mecanismo básico de adaptación del sistema. Consiguientemente presenta los problemas (mencionados en el apartado anterior) derivados de esta estrategia. Vino se plantea extensiones que alteren la política, el rendimiento y la funcionalidad del sistema aunque no es factible reemplazar aquellas abstracciones que resulten inadecuadas y ya estén presentes en el sistema.

Vino suministra un interfaz común que se emplea para cualquier recurso presente en el sistema. Dicho interfaz es independiente de la representación adoptada por cada recurso. Los recursos en Vino son entidades que poseen un conjunto de operaciones y un conjunto de propiedades (algo no muy diferente de la estructura de los recursos en sistemas basados en objetos). En cuanto al nivel de abstracción de dichos recursos, debemos decir que éstos incluyen abstracciones tales como *ficheros*.

[6] dynamic binding

Estos recursos están tipados y, aunque pueden reemplazarse (o su implementación extenderse) en tiempo de ejecución, la jerarquía de tipos de dichos recursos en Vino ha de indicarse *por completo* en tiempo de compilación del sistema. Sorprendentemente, esta limitación es explícitamente introducida por los autores del sistema [149] de tal modo que se beneficie el rendimiento y la robustez como consecuencia de la información adicional (tipos) suministrada. Como compensación, se permite introducir nuevos *subtipos* en tiempo de ejecución (un subtipo ha de mantener al menos la semántica del tipo de que deriva).

Pensando ahora en la implementación de los recursos, y no en el interfaz que éstos presentan, la implementación por defecto que suministra el sistema para los distintos recursos puede alterarse mediante la introducción de *grafts* en el núcleo del mismo. Como vemos, Vino resulta entonces muy similar a Spin. Al contrario que Spin, Vino emplea C y C++ como lenguaje para escribir *grafts*. El compilador que se emplea para compilarlos introduce comprobaciones adicionales en todos los accesos a memoria y en todas las llamadas a función. En nuestra opinión, resulta en este caso más efectivo el empleo de un lenguaje con tipado estricto de datos como Modula 3, de tal modo que gran parte de estas comprobaciones puedan obviarse, aunque es cierto que sin dichas comprobaciones un usuario malicioso puede emplear un compilador modificado[7] para sortear el sistema de tipado estricto.

Una de las contribuciones de los autores de Vino es la clasificación de los distintos modelos de extensiones que pueden realizarse [148]. Dichos autores distinguen entre extensiones de priorización, de streaming y de caja negra. El primer modelo se basa en asignar prioridades a distintos elementos de entre los que se supone hay que escoger alguno(s). Un caso típico es la elección de una página a reemplazar. Así, anidando distintas políticas es factible partir de un conjunto original de elementos e ir aplicando dichas políticas de forma sucesiva hasta obtener el conjunto final de elementos que interesa. El segundo modelo, streaming, se basa en el procesamiento mediante pipe-lines de tal modo que sucesivos componentes van filtrando los datos de que se parte hasta obtener un conjunto de datos ya procesado. El tercer modelo (caja negra) es el más general y consiste en un elemento software que tiene algunas entradas, un proceso que no conocemos (y sobre el que no cabe hacer suposiciones) y presenta una única salida. La política de "read-ahead" en un disco es un ejemplo de este último modelo.

4.4.2 Sistemas con μkernel extensible

El segundo enfoque se centra en una reducción de servicios del núcleo [148], eliminando aquellos que pueden ser implementados posteriormente fuera del núcleo —posiblemente de forma que se obtenga un sistema distribuido. Como sugiere el hecho de que la mayoría de los sistemas adaptables opten por él [170, 34, 35, 160, 132, 141], parece más adecuado el segundo enfoque que el primero para conseguir sistemas extremadamente adaptables.

En este tipo de sistemas se puede implementar el modelo de SO basado en DAMN de un modo más simple que con el tipo anterior (sistemas que descargan código) de sistemas adaptables. No obstante, sería preciso implementar una capa de software sobre el μkernel de tal modo que se pudiesen utilizar recursos remotos. En la mayoría de los casos incurriríamos en una reimplementación de servicios dado que ya existe código dentro del μkernel que está haciendo tareas muy similares. Alternativamente, podría modificarse el núcleo del sistema para que considerase recursos físicos remotos. Esta modificación no sería demasiado complicada en sistemas como el Exokernel [62] o SPACE [135].

[7] Siempre se puede *forzar* el uso de un compilador en el que sí confiamos si forzamos un sistema de compilación bajo demanda en el momento de la inserción del código de usuario dentro del núcleo.

GLUnix

GLUnix [6, 167, 6, 3] ya sugería emplear librerías protegidas (aunque empleaba técnicas de aislamiento de fallos mediante software [173]) para implementar los servicios del SO, siendo éste el mecanismo principal para conseguir adaptabilidad en los servicios del sistema. GLUnix intentaba así eliminar la ineficiencia de los dominios de protección adicionales introducidos por los μkernels. Posteriormente el μkernel L4 [110, 111] parece mostrar que en realidad no es tan drástica la sobrecarga introducida por estos dominios adicionales.

Dado que éste modelo puede considerarse un precursor del de Exokernel, no lo discutiremos aquí. Lo dicho en el apartado 4.4.2 dedicado al Exokernel es válido también para la arquitectura de GLUnix, a pesar de las diferencias entre la arquitectura de ambos sistemas.

PEACE

PEACE [143] sigue la idea de Choices [31] de construir el SO como una *familia* de sistemas de tal modo que el sistema se construya ensamblando distintas piezas. En Choices se empleaban técnicas de orientación a objetos (polimorfismo) para conseguirlo. En PEACE se adopta un enfoque basado principalmente en la composición de módulos (aunque también se estructure el sistema como un conjunto de objetos).

La diferencia más notable entre éste y otros sistemas es que (al contrario que otros sistemas distribuidos) PEACE contempla el caso en que no es necesario multiplexar el hardware entre distintas aplicaciones (como es el caso en multitud de aplicaciones paralelas que adaptan la distribución de sus algoritmos a la topología de la red). Inferno [52] es otro sistema que también contempla esta posibilidad. (La posibilidad en *Off* de *no* incluir múltiples espacios de direcciones deriva de lo aprendido por estos sistemas).

PEACE está compuesto por dos elementos que ejecutan en modo privilegiado (el núcleo y el kernel) sobre los que se sitúan extensiones del sistema (POSE[8]) empleadas por las aplicaciones. POSE es dinámicamente reconfigurable (adaptable), de tal modo que para cada aplicación estén presentes sólo los servicios necesarios. Los distintos servicios presentes en POSE se pueden cargar en demanda durante tiempo de ejecución. El núcleo de PEACE es el componente encargado de suministrar IPCs y threads. El kernel es un conjunto de extensiones que emplean múltiples threads e implementan extensiones mínimas del núcleo. Los mecanismos de IPC son básicamente traps y RPCs (denominados “local object invocation” y “remote object invocation” en PEACE).

PEACE explora principalmente la reconfiguración de elementos del sistema que forman parte de librerías que ejecutan (empleando los flujos de control de las aplicaciones) en área de usuario, pensando en entornos que emplean un único dominio de protección (ej. para construcción de sistemas empotrados) y también en entornos de ejecución sobre un SO nativo que actúa como plataforma base. *Off* en cambio trata de *bastarse* para cubrir distintos ámbitos de aplicación.

En PEACE no es posible reemplazar dinámicamente las componentes del kernel que implementan extensiones del núcleo. Dado que son estas extensiones las que típicamente implementan las abstracciones (de bajo nivel de abstracción) básicas del sistema, no es factible realizar la adaptación en este punto.

Las ideas introducidas por *Off* podrían no obstante incorporarse como extensiones básicas en PEACE, aunque difícilmente podríamos seguir tolerando la adaptabilidad que presentan los Shuttles, Portales y DTLBs dado que habría que emplear componentes no adaptables que forman parte del núcleo y el kernel de PEACE.

[8] Parallel Operating System Extension

Kea

Kea [170] es un SO cuya principal novedad es la posibilidad de redefinir selectivamente los puntos de entrada del sistema.

Aunque las abstracciones suministradas por Kea no son en realidad novedosas (dominios o espacios de direcciones, threads y portales), su novedad radica en que permite que las aplicaciones redefinan selectivamente los manejadores de los portales que emplean.

El mecanismo empleado consiste fundamentalmente en emplear un nivel de indirección entre la aplicación y los portales, de tal modo que una aplicación puede *alterar* dicha indirección para uno o varios portales y desde ese momento las invocaciones de dichos portales alcanzarán otros portales diferentes de forma transparente.

Aunque en *Off* hemos optado por dejar para las aplicaciones la implementación de dicho nivel de indirección, los portales de Kea tienen una fuerte influencia en el diseño de los portales en *Off*.

Fluke

Fluke es el kernel de Flux [71], un sistema que se centra en la implementación de una arquitectura que permite la implementación eficiente de un modelo anidado de procesos.

Las propiedades más interesantes de Fluke son la *encapsulacӧn de estado* de los objetos del sistema de tal modo que el estado de procesos hijos está encapsulado en el de los padres, y el *control de borde* mediante el cual los padres pueden interceptar toda la comunicación de los hijos con el resto del sistema. En esencia, Fluke permite una redefinición selectiva de los servicios del sistema con el fin de que el padre de un proceso hijo pueda redefinir selectivamente algunos de los servicios suministrados por el “abuelo”. El resto de los servicios ejecutan sin necesidad de que el padre los intercepte.

Fluke constituye un “objeto” inicial dentro del cual se anida el resto de objetos del sistema. Las abstracciones por él suministradas son los espacios de direcciones (similares a los de L4) y los Threads. Éstas son gestionadas completamente por el μkernel (a un nivel de abstracción comparable al de L4, Grasshopper o Spring).

Podríamos decir pues que aunque Fluke es un μkernel extremadamente flexible que resulta ventajoso a la hora de implementar modelos de procesos anidados, no es un sistema adaptable si lo comparamos con el Exokernel y, por añadidura, es un sistema centralizado.

El Cache Kernel

El cache kernel [34] es un sistema que propone que el kernel del sistema sea básicamente una cache de los recursos empleados por las aplicaciones. De este modo, son las aplicaciones las que poseen los recursos físicos (aunque de modo virtualizado) y dentro del núcleo se mantienen sólo aquellos recursos más empleados. En la práctica, el cache kernel es un sistema similar al exokernel que discutimos a continuación.

Los objetos de los que hace caching el Cache Kernel son espacios de direcciones y threads. El mecanismo de intercomunicación de procesos empleado se basa en la implementación de los espacios de direcciones. Básicamente se extiende el concepto de *traduccӧn de direcciones* para que incluya opcionalmente un thread que deberá manejar mensajes relacionados con dicha dirección. Los clientes pueden *señalar* dicha dirección y el thread que lo sirve será notificado. La transferencia de datos entre clientes y servidores se implementa con ayuda del gestor de memoria virtual.

El Exokernel

El exokernel [62] es un nuevo modelo de construcción de SSOO consistente en reducir la función del kernel del sistema a la multiplexación segura de los recursos físicos. Un exokernel suministra a las aplicaciones el mismo interfaz que suministra el hardware, no se realiza ninguna *virtualización* ni *abstracción* del mismo. Consiguientemente, los exokernels son por naturaleza sistemas centralizados.

A pesar de su reducida funcionalidad, la asignación y revocación de recursos está implementada dentro del núcleo, aunque las aplicaciones pueden participar en las políticas de asignación y revocación (se emplea asignación y revocación explícita).

En *Off* hemos optado por dejar por completo fuera del núcleo la revocación de recursos, tal y como sucede en el sistema que discutimos a continuación.

SPACE

SPACE [134, 135] es un SO que difiere en las abstracciones básicas que implementa del resto de los SSOO tradicionales. Tradicionalmente se toman procesos (a veces threads), memoria virtual e IPC como las abstracciones básicas que hay que implementar. En SPACE se opta en cambio por implementar *procesadores*, *espacios de direcciones* y un mecanismo de *excepciones*. SPACE trata de exponer el hardware disponible de tal modo que sea factible la coexistencia de abstracciones convencionales de propósito general y abstracciones desarrolladas a medida de las aplicaciones, igual que hacemos en *Off*.

Las abstracciones más significativas que implementa SPACE son los *espacios* (de los que toma el nombre), los *dominios* y los *portales*. Los espacios suministran las traducciones para las direcciones empleadas por el procesador. El resultado de la traducción puede ser una dirección de memoria o un portal. En realidad, un portal es tan sólo una dirección que no tiene traducción; el fallo de página es gestionado por SPACE que lo traduce en una invocación a portal. Los dominios de SPACE son una generalización de los bits de *válido* y *escribible* empleados por el hardware para cada traducción. Un dominio en SPACE es un mapa de bits que restringe las direcciones que es posible emplear (esto incluye los portales que es posible direccionar).

Como hemos dicho, el mecanismo empleado por SPACE para enviar eventos a las aplicaciones también recibe el nombre de Portal. No obstante, SPACE incorpora distintos tipos de portales con distinta semántica (RPCs, interrupciones, etc.).

En pocas palabras, SPACE lleva al extremo el modelo de Exokernel. En SPACE se extrae del μkernel incluso la asignación y la revocación de recursos. Igual que sucede con el Exokernel, SPACE es un sistema centralizado, por lo que no se considera la distribución en el modelo propuesto.

4.4.3 Sistemas que adaptan código dinámicamente

Otro enfoque empleado en la construcción de sistemas adaptables es la generación dinámica de código o la especialización de código en tiempo de ejecución. Se trata de *alterar* el código ya presente en el SO para que su ejecución sea más eficiente para las aplicaciones concretas que utilizan realmente el sistema.

En este caso, las posibilidades de adaptación que nos ofrece el sistema están sumamente restringidas dado que han de estar previstas por el arquitecto del sistema. Estos sistemas suelen implementar abstracciones de mediano o elevado nivel de abstracción (como procesos, espacios de direcciones y en algunos casos incluso ficheros).

Synthesis

Synthesis [117, 136] emplea una técnica conocida como especialización incremental [137, 116] para especializar el código del SO (en [137] se discute el caso de la especialización de UNIX) en función del uso que realizan las aplicaciones. La especialización incremental es en realidad una generalización del concepto de evaluación parcial [43]. Se trata de construir el sistema de tal modo que el código puede especializarse dinámicamente (a medida que el sistema conoce más acerca del uso que la aplicación está haciendo de un recurso determinado) para evitar sobrecargas innecesarias.

El problema principal que vemos en este modelo (especialización dinámica de código) es que todo lo que se puede hacer con el sistema es especializar su código para usos concretos. No es factible alterar sustancialmente la implementación de los servicios suministrados ni es factible emplear abstracciones diferentes de las suministradas por el sistema.

Synthesis no contempla la distribución de los recursos en un sistema distribuido y ha centrado su estudio en sistemas centralizados. Aunque parece factible (y útil) emplear la especialización dinámica de código para optimizar la ejecución de protocolos de comunicaciones, protocolos de localización de objetos y protocolos de consistencia de caches en sistemas distribuidos, Synthesis no estudia estos aspectos.

4.4.4 Sistemas que emplean reflexión

El cuarto enfoque empleado para construir sistemas adaptables consiste en emplear *meta-abstracciones* que definan la semántica de los mecanismos básicos empleados por los objetos o entidades que constituyen el sistema. Típicamente tenemos operaciones (que podemos redefinir) que implementan mecanismos tales como la invocación de una operación en un objeto, la localización de un objeto, etc.

Este enfoque presenta problemas derivados de la sobrecarga que genera el nivel de meta-abstracciones. En sistemas que son ya de por si ineficientes, como Java, es posible emplear esta técnica (la reflexión) para optimizar el funcionamiento del sistema (véase por ejemplo MetaJava [77, 102], una extensión de la máquina virtual de Java [112])

Apertos

Apertos [178] es un sistema distribuido orientado a objetos que emplea intensivamente la reflexión, recogiendo lo aprendido anteriormente en la realización del sistema operativo Muse [179] (desarrollado por el mismo grupo).

En Apertos el sistema se estructura como una serie de objetos que interaccionan entre sí con ayuda de meta-objetos. Dichos meta-objetos suministran reflexión al sistema y permiten redefinir la semántica de la práctica totalidad de los mecanismos empleados por el sistema.

La orientación a objetos de Apertos es extremadamente fuerte, llegando alguno de sus autores a proponer un universo de objetos regidos por leyes quasi-físicas que se mueven libremente en función de las interacciones que se dan entre ellos. A dicho modelo se le denominó "campo computacional" [164].

Sería factible incorporar las técnicas de reflexión exploradas por Apertos en *Off*, aunque dado el escaso nivel de abstracción de los servicios suministrados por éste, resulta cuando menos discutible si la sobrecarga introducida por el nivel de meta-objetos compensa la flexibilidad obtenida.

4.5 Sistemas de soporte a la ejecución

Aunque no son estrictamente Sistemas Operativos, o al menos no son sistemas concebidos como SSOO de propósito general, debemos mencionar que existen sistemas de soporte para tiempo de ejecución cuya finalidad es suministrar servicios básicos para lenguajes de programación que permiten la realización de sistemas distribuidos. En cierto modo, su espíritu está cerca de la idea de partida de *Off*: suministrar mecanismos básicos para que el usuario pueda programar sus propios servicios (o emplear bibliotecas/servidores que lo hagan).

Lamentablemente, los sistemas de soporte en tiempo de ejecución para distintos lenguajes que permiten la implementación de sistemas distribuidos están centrados en el suministro de abstracciones para el lenguaje que soportan. Otro inconveniente es que estos sistemas de soporte rara vez suministran los servicios de protección básicos que necesita un SO, dado que su interés suele ser el uso de una única aplicación (con un único dominio de protección) en distintos nodos de una red.

Finalmente, la mayoría de las veces estos sistemas están implementados sobre un SO nativo que actúa de anfitrión (ej. como sucede con PVM [153]).

Capítulo 5

Conclusiones y Trabajo Futuro

Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?

— Epigrams in Programming, ACM SIGPLAN Sept. 1982

El principal propósito de este capítulo es el de extraer algunas conclusiones del trabajo efectuado, ya descrito en capítulos anteriores. A tal efecto dedicamos el primer apartado. Posteriormente, el apartado 5.2 enumerará las contribuciones realizadas y por último, en el apartado 5.3 describiremos someramente algunas líneas por las que puede discurrir el trabajo futuro.

5.1 Conclusiones

5.1.1 El prototipo de DAMN

El prototipo de DAMN, *Off*, se implementó para arquitecturas basadas en Intel x86. Está formado por unas 9.000 líneas de código fuente escrito en lenguaje C, incluyendo comentarios. Aunque es un prototipo algo inestable[1] y a pesar de que no toda la funcionalidad prevista inicialmente está incorporada (como evidencian las notas "no realizado por la implementación actual" que aparecen en los capítulos anteriores), está disponible públicamente bajo licencia GPL en [10]. Todo el código fuente, herramientas empleadas (incluyendo el OSKit [70]) y publicaciones relacionadas pueden encontrase ahí.

Las medidas de rendimiento de esta versión de desarrollo, que no se implementó con la *eficiencia* como objetivo principal e incluye múltiples asertos y comprobaciones de depuración, muestran que una transferencia protegida de control (PCT), consistente en una invocación de un portal que atraviesa un dominio de protección, precisa de 2.100 ciclos de procesador (tal y como muestra el contador de ciclos hardware del procesador). Considerando un procesador a 100 MHz, una PCT supone $21\mu s$, un orden de magnitud más rápido que Mach, en el mismo orden de magnitud que el sistema Spring de Sun y un orden de magnitud más lento que L4 [111].

[1] Algunas aplicaciones de prueba hacen que el sistema deje de funcionar tras dos semanas de uso continuado.

El tiempo de una PCT es el factor crítico que determina el rendimiento del sistema en su conjunto, dado que los servicios del sistema se hacen accesibles mediante PCTs.

Mediciones más significativas que consideren aplicaciones típicas (editores, compiladores, etc.) ejecutando sobre un DAMN deberán aguardar a la disponibilidad de una versión optimizada y estable de *Off*[2].

Con respecto al empleo de memoria, *Off* consume unos 50K incluyendo el soporte para DTLBs (el DMM) en el núcleo. La mayor parte de esta cantidad se emplea en el almacenamiento de las estructuras de datos que representan el estado de los marcos de memoria física y es proporcional a la cantidad de memoria física instalada en el sistema.

Por lo que hemos podido comprobar, la arquitectura implementada realiza trabajo extra (comparándola con una arquitectura similar para un μkernel centralizado) cuando los recursos son remotos. Parece pues que un DAMN debiese ser capaz de alcanzar niveles de rendimiento comparables a los de los μkernels más eficientes que se conocen, como L4 [111]. Cuando los recursos son locales, la sobrecarga en tiempo de ejecución de un DAMN con respecto a un sistema centralizado de similar nivel de abstracción se debe principalmente a operaciones de "indirección y comparación", añadidas en cada referencia a recursos del sistema cuyo objeto es comprobar la disponibilidad de recursos en el nodo local. Dicha sobrecarga es despreciable respecto al tiempo que se emplea en la implementación de una llamada al sistema típica.

Creemos entonces que resulta factible incorporar el modelo de *Off* a μkernels que emplean la extensibilidad como medio para alcanzar la adaptabilidad del mismo. Para hacerlo podría pagarse una "indirección y comparación" cada vez que se referencia un recurso del sistema. El beneficio de esta pequeña sobrecarga sería la posibilidad de operar transparentemente con recursos físicos remotos siguiendo los dictados de las aplicaciones.

Lecciones aprendidas

Como resultado de la realización del prototipo hemos podido aprender que:

- El modelo de DAMN resulta fácilmente implementable dando lugar a μkernels extremadamente simples.
- El lenguaje de programación *C* no es un buen lenguaje para implementar SSOO. Gran parte del tiempo de desarrollo se empleó en eliminar errores que podría haber detectado el compilador.
- El uso de orientación a objetos habría simplificado enormemente la realización del núcleo (ver apartado 5.3.1).
- No merece la pena multiplexar un portal entre *todos* los puntos de entrada de cada uno de los servidores del sistema. Los portales son suficientemente baratos como para emplear uno por cada punto de entrada.
- La suposición que realizamos de que los recursos físicos se conocen en tiempo de arranque es falsa. Últimamente han aparecido módulos de memoria que es posible insertar en caliente en algunas arquitecturas.
- El modelo de Shuttles de *Off* podría incorporarse a sistemas tradicionales para suministrar interfaces comparables a VFS entre el sistema de gestión de procesos y el resto del sistema.
- El empleo de Propiedades (de Shuttles) resulta muy conveniente para desarrollar un sistema de gestión de procesos, ya que permite depurarlo con independencia del resto de subsistemas.

[2] Cuya implementación ya ha comenzado como parte del trabajo futuro.

5.1.2 El modelo de SO basado en DAMN

Hemos visto que un μkernel distribuido adaptable que exporte el hardware presente en la red es factible y puede ofrecer altos niveles de adaptabilidad y, al mismo tiempo, facilidades de cara a la distribución del sistema. De este modo cada aspecto se beneficia del otro. Para cerciorarnos, hemos implementado un prototipo para realizar una *prueba de concepto* y también hemos publicado las ideas y resultados que se presentan en la presente tesis [15, 14, 13, 11, 12, 67, 30].

Con el sistema propuesto, las aplicaciones de los usuarios (incluyendo en éstas a servicios de SSOO distribuidos) pueden implementar abstracciones (distribuidas) a medida en lugar de reinventar las que ya existen en el sistema. El sistema presenta beneficios mencionados en otras publicaciones [13] que facilitan la realización de equilibrado de carga, replicación y migración en sistemas de objetos distribuidos.

Aunque la comparación entre una implementación completa de un SO basado en DAMN y un SO tradicional deberá aguardar a que completemos la reimplementación de Off, Off++ (ver apartado 5.3), todo parece indicar que el modelo podría emplearse en condiciones competitivas con respecto a los sistemas existentes en la actualidad.

Lecciones aprendidas

En relación al modelo de SO basado en DAMN hemos aprendido que:

- No basta con un DAMN para obtener un SO distribuido adaptable. Es necesario que los servicios implementados sobre el DAMN estén diseñados con el mismo espíritu. Una implementación de SO basada en un único emulador eliminaría por completo la adaptabilidad del sistema.

- La característica crucial de un DAMN es la habilidad de *referenciar* recursos remotos. Ésto puede incorporarse a sistemas existentes.

- Es importante que los SSOO expongan a los usuarios los interfaces que ellos emplean internamente.

5.2 Contribuciones

La presente tesis ha aportado las siguientes contribuciones al estado del arte en cuanto a la construcción de sistemas operativos distribuidos adaptables se refiere:

1. La percepción de que el sistema debe estar distribuido desde el principio: la manipulación de los recursos físicos con independencia de su ubicación.

2. La percepción de que un buen modo de distribuir un SO y mantener altas cotas de adaptabilidad es permitir que cada aplicación realice la distribución ella misma apoyándose en un DAMN.

3. La realización de un prototipo de DAMN, *Off*.

5.3 Trabajo Futuro

5.3.1 Un SO distribuido auto-adaptable

En realidad, el trabajo futuro se ha comenzado a desarrollar ya. Como puede verse en [30], estamos trabajando conjuntamente con el System Software Research Group de la Universidad de Illinois en

Urbana-Champaign en el desarrollo de un nuevo sistema operativo, *2K*, que entre otras novedades emplea como núcleo una reimplementación de *Off* denominada *Off*++ [91].

Off++ es en realidad la primera implementación de *Off* que se realiza con la eficiencia como objetivo primario, dado que pensamos competir con SSOO comerciales. El prototipo actual fue en realidad una prueba de concepto.

Como parte de este trabajo, implementaremos un SSOO distribuido auto-adaptable que sea consciente de su propia arquitectura (empleando técnicas de reflexión). Con la construcción de dicho sistema podremos comprobar los beneficios materiales (en términos de rendimiento de las aplicaciones) que un DAMN puede ofrecer en la práctica.

5.3.2 Sistemas de tiempo real

Dado que *Off* se limita a multiplexar el hardware, resulta interesante explorar su aplicación en el ámbito de los sistemas distribuidos de tiempo real.

Otra de las líneas de trabajo que tenemos intención de explorar es la aplicación de *Off*[3] a la construcción de sistemas distribuidos de tiempo real.

Como punto de partida, el Grupo de Sistemas y Comunicaciones (al que pertenece el autor) está proponiendo proyectos de investigación en cuyo seno se abordará el transporte del sistema de soporte de tiempo de ejecución del lenguaje de programación Ada 95 sobre *Off*.

Esperamos que el hecho de emplear un DAMN facilitará enormemente implementaciones sencillas y eficientes del anexo de sistemas distribuidos de dicho lenguaje.

5.3.3 Ordenadores de Red

Finalmente, sería útil explorar el empleo de DAMNs en la construcción de “Network Computers”, sistemas de recursos reducidos que descargan todos los servicios del sistema empleando una red de comunicaciones.

El reducido tamaño de un DAMN lo hace idóneo para su empleo en este tipo de sistemas, dado que podría arrancarse desde memoria flash y cargar el resto de sistema (ejecutando en área de usuario) desde la red.

[3] En realidad pensamos utilizar *Off*++.

Bibliografía

[1] Barak A. and La'adan O. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, April 1998. (to appear).

[2] Mike Accetta, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for Unix development. In *USENIX Summer Conference*, Jul 1986.

[3] T.E. Anderson, Culler D.E., and Patterson D.A. A case for NOW (networks of workstations). *IEEE Micro*, 1994.

[4] Thomas E. Anderson, B.N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 95–109. Association for Computing Machinery SIGOPS, October 1991.

[5] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference ASPLOS*, pages 95–109, October 1991.

[6] Remzi H. Arpaci, Andrea Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. Technical report, UC Berkeley, 1994.

[7] Jose M. Bernabeu Auban, Phillip W. Hutto, and M. Yousef A. Khalidi. The Architecture of the Ra Kernel. Technical report, Georgia Institute of Technology, 1987.

[8] H. E. Bal and A. S. Tanenbaum. Orca: A Language for Distributed Object-Based Programming. *SIGPLAN Notices*, 25(5):17–24, may 1990.

[9] Henri E. Bal, M. Frans Kaashoek, and A.S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[10] Francisco J. Ballesteros. Off web site. http://www.gsyc.inf.uc3m.es/off, 1996.

[11] Francisco J. Ballesteros and Luis L. Fernandez. An adaptable and extensible framework for distributed object management. In *Special Issues in Object Oriented Programming. Workshop reader of the 10th European Conference on Object-Oriented Programming, ECOOP'96.*, Linz (Au), july 1996.

[12] Francisco J. Ballesteros and Luis L. Fernández. Advice: An Adaptable and Extensible Distributed Virtual Memory Architecture. In *Parallel and Distributed Computing Systems*, 1996.

[13] Francisco J. Ballesteros and Luis L. Fernández. An Adaptable and Extensible Framework for Distributed Object Managment. In *Proceedings in the II Workshop on Mobility and Replication (ECOOP'96)*, Linz,Austria, July 1996.

[14] Francisco J. Ballesteros and Luis L. Fernandez. The inherently distributed adaptable Off microkernel. In *Actas de las Jornadas de Concurrencia*, Vigo, 1997.

[15] Francisco J. Ballesteros and Luis L. Fernández. The Network Hardware is the Operating System. In *Proceedings of the 6th Hot Topics on Operating Systems (HotOS-VI).*, Cape Cod, MA (USA), May 1997.

[16] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, P. Ledot, M. Meysembourg, H.Ñguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandôme. Architecture and implementation of guide, an object-oriented distributed system. *Computing Systems*, 4(1), Winter 1991.

[17] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN — An extensible microkernel for application specific operating system services. Technical report, University of Washington, February 1994.

[18] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[19] Krishnba Bharat and Luca Cardelli. Migratory Applications. In *ACM Symposium on User Interface Software and Technology'95*, Pittsburg, PA, Nov 1995. ACM.

[20] Lubomir Bic and Alan Shaw. *The Logical Design of Operating Systems*. Prentice-Hall, 1988.

[21] David L. Black, David B. Golub, Karl Hauth, Avadis Tevanian, and Richar Sanzi. The Mach Exception Handling Facility. Technical Report CMU-CS-88-129, Carnegie Mellon University, 1988.

[22] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 87–98, Berkeley, CA, USA, Summer 1994. USENIX.

[23] Patrick Bridges. Operating system projects. http://www.cs.arizona.edu/people/bridges/os/full.html.

[24] Bill Bryant, Steve Sears, David Black, and Alan Langerman. An Introduction to Mach 3.0's XMM subystem. In *Operating System Collected Papers*, volume 2. OSF Research Institute, Oct 1993.

[25] A. Burns. Distributed hard real-time systems: What restrictions are necessary? In H. Zedan, editor, *Proceedings of the 1989 Real-Time Symposium*, 1989.

[26] Vinny Cahill. An overview of the Tigger object-support operating system framework. In Keith G. Jeffery, Jaroslav Kral, and Miroslav Bartosek, editors, *SOFSEM '96: Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 34–55, Berlin/Heidelberg, November 1996. Springer-Verlag.

[27] R. H. Campbell and P. W. Madany. Considerations of persistence and security in choices, an object-oriented operating system. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen (Federal Republic of Germany), May 1990.

[28] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings and additional papers, C++ workshop*, Berkeley, CA (USA), November 1987. USENIX.

[29] Roy Campbell, Vincent Russo, and Gary Johnston. A class hierarchical, object-oriented approach to virtual memory management in multiprocessor operating systems. Technical Report R-88-1459, Department of Computer Science, Urbana, Illinois (USA), September 1988.

[30] Roy H. Campbell, Francisco J. Ballesteros, Fabio Kon, Ashish Singhai, Dulcineia Carvalho, and Robert Moore. 2k: A distributed adaptable operating system. http://choices.cs.uiuc.edu/2k, August 1997.

[31] Roy H. Campbell and Nayeem Islam. Choices: A parallel object-oriented operating system. In Peter Wegner Gul Agha and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[32] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report R-89-1510, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.

[33] Jeffrey S. Chase, Henry M. Levy, Michael Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR-92-03-02, University of Washington, 1992.

[34] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, November 1994.

[35] D. R. Cheriton. The V Kernel. *IEEE Software*, (1(2)19–42), 1984.

[36] D. R. Cheriton. Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems. *ACM Operating Systems Review*, 19(4):26–33, October 1985.

[37] D. R. Cheriton. Problem Oriented Shared Memory Revisited. In *Proc. of the 5th ACM SIGOPS European Workshop*, September 1992.

[38] David Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[39] W.H. Cheung and Anthony H.S. Loong. Exploring Issues of Operating Systems Structuring: from Microkernel to Extensible Systems. University of Hong-Kong csd.hku.hk.

[40] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte, and X. Rousset de Pina. Is the microkernel technology well suited for the support of object-oriented operating systems: the guide experience. In *Microkernel and Other Kernel Architectures*. USENIX, 1993.

[41] Hsiao-Keng Jerry Chu. Zero-copy TCP in Solaris. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 253–264, Berkeley, CA, USA, January 1996. USENIX.

[42] D.D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, December 1985.

[43] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity, and portability in operating systems. In *Proceedings of the 1993 Conference on Partial Evaluation and Program Manipulation*, Copenhagen, 1993. Also in http://www.cse.ogi.edu/DISC/projects/synthetix/publications.html.

[44] On-Line Applications Research Corp. RTEMS — Real time executive for multiprocessor systems. http://lancelot.gcs.redstone.army.mil/rtems.html, 1994.

[45] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. Research and development*, 25(5):483–490, September 1981.

[46] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[47] Helen Custer. *Inside Windows NT*. Microsoft Press, 1992.

[48] James da Silva et al. The maruti 3.0 project. http://www.cs.umd.edu/projects/maruti/, 1996.

[49] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1993.

[50] Alan Dearle. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 3(7):289–312, 1994.

[51] François Barbou des Places, M.Stephen, and Franklin D. Reynolds. Linux on the OSF Mach3 Microkernel. In *In Proceedings of Freely Distributable Software*. Free Software Foundation, 1996.

[52] Sean Dorward, Rob Pike, Dave Presotto, Howard Trickey, and Phil Winterbottom. Inferno: La commedia Interattiva. In *Proceeding of the OSDI'96 (WIP)*. USENIX, 1996. Also at http://www.usenix.org/publications/library/proceedings/osdi96/wip.html.

[53] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[54] F. Douglis and J. Ousterhout. Process migration in Sprite: A status report. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1), winter 1989.

[55] R. Draves. Mach. In *Proceedings of the USENIX Micro-Kernel and Other Kernels Architectures*, pages 27–28, Seattle, Washington, Apr 1992. USENIX.

[56] Richard P. Draves, Brian Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.

[57] Nelson G. (editor). *System Programming in Modula-3*. Prentice-Hall, 1991.

[58] Kevin Elphinstone. Address space management issues in the mungi operating system. Technical report, School of Computer Science and Engineering, The University of New South Wales, November 1993.

[59] D. Engler, M. F. Kaashoek, and J. O'Toole. The Operating System Kernel as a Secure Programmable Machine. In *Proc. of the 6th SIGOPS European Workshop*, pages 62–67, Wadern, Germany, Sept 1994. ACM SIGOPS.

[60] Dawson R. Engler. *The Design and Implementation of a Prototype Exokernel Operating System*. PhD thesis, Massachusetts Institute of Technology, January 1995.

[61] Dawson R. Engler, Sandeep K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Hot Topics in Operating Systems (HOTOS-V)*, 1995.

[62] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the Fifteenth Symposium on Operating System Principles (SOSP)*, December 1995.

[63] M. Rasit Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. Technical report, University of New Orleans, 1995.

[64] Esprit. *Predictably Dependable Computing Systems*. Basic Research Series. Springer, 1995.

[65] Miguel Castro et. al. Mike: A distributed object oriented programming platform on top of the Mach micro-kernel. In *Proceedings of the USENIX MACH III Symposium*, Santa Fe, New Mexico, Apr 1993.

[66] Mevel F. and Simon J. Distributed communication services in the masix system. In *15th International Conference on Computers and Communications*, Phoenix, 1996. IEEE.

[67] Luis L. Fernandez and Francisco J. Ballesteros. Advice: An adaptable and extensible distributed virtual memory architecture. Technical Report UC3M-TR-CS-1997-02, Carlos III University of Madrid, 1997.

[68] Raphael Finkel. *An Operating Systems Vade Mecum*. Prentice-Hall, 2 edition, 1988.

[69] Trent Fisher. Towards a New Strategy of OS Design. GNU's bulletin, January 1994. http://www.gnu.prep.ai.mit.edu/.

[70] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux os toolkit: A substrate for kernel and language research. In *Proceedings of the 16th SOSP*, Saint-Malo, France, October 1997. ACM.

[71] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. OSDI*, October 1996.

[72] Bryan Ford and Jay Lepreau. Microkernels should support passive objects. In *Proc. of I-WOOOS*, December 1993.

[73] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Winter Usenix Conference*, January 1994.

[74] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *The Second USENIX Mach Symposium Conference Proceedings, November 20–22, 1991. Monterey, CA*, pages 163–176, Berkeley, CA, USA, November 1991. USENIX.

[75] Hamilton G., M.L.Powell, and J.G.Mitchel. Subcontract: A flexible base for distributed programming. *14th ACM Symposium on Operating System Principles*, Dec 1993. Asheville,NC.

[76] P. Galdámez, F. Muñoz, and J. M. Bernabeu. Extending ORB to support high availability. In *Actas de las V Jornadas de Concurrencia*, Vigo, Spain, Junio 1997.

[77] Michael Golm and Jurgen Kleinoder. Metajava—a platform for adaptable operating-system mechanisms. In *ECOOP'97 workshop Object-Orientation and Operating Systems*, 1997. Also as Friedrich Alexander University CS TR-I4-97-10.

[78] GoodHeart and Cox. *The Magic Garden Explained*. Prentice-Hall, 1994.

[79] James Gosling and Henry McGilton. *The Java Language Environment – A White Paper*. Sun Microsystems, 2550 Garcia Av. Mountain View, CA 94043, May 1995.

[80] James Gosling and Hery McGilton. *The Java Language – A White Paper*. Sun Microsystems, 2550 Garcia Av. Mountain View, CA 94043, 1994.

[81] John R. Graham. *Solaris 2.x: internals and architecture*. McGraw-Hill, New York, NY, USA, 1995.

[82] C.W. Mercer H. Tokuda. Arts: A distributed teal-time kernel. *ACM Operating Systems Review*, 23(3), July 1989.

[83] Anna Hac. Distributed file systems - A survey. *Operating Systems Review*, 10(1):15–18, January 1985.

[84] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer USENIX Conference*. USENIX, Jun 1993.

[85] J.H. Hartman, A.B. Montz, David Mosberger, S.W. O'Malley, L.L. Peterson, and T.A. Proebsting. Scout: A communication-oriented operating system. Technical report, University of Arizona, June 1994.

[86] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on ASPLOS*, pages 187–199, October 1992.

[87] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Graham R. Hellestrand. A distributed single address-space operating system supporting persistence. Technical report, School of Computer Science and Engineering, The University of New South Wales, March 1993.

[88] D. Hildebrand. Qnx: microkernel technology for open systems handheld computing. In *Proceedings of Pen and Portable Computing Conference Exposition*, May, 1994. Available via ftp from ftp.qnx.com:pub/papers/qnx-pen.ps.Z.

[89] W.C. Hsieh, M.F. kaashoek, and W.E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *4th Workshop on Workstation Operating Systems*, pages 186–190, October 1993.

[90] IEEE, 345 East 47th St. NY 10017. *IEE Standard P1003.4 (Real-Time extensions to POSIX)*, 1991.

[91] Franscisco J.Ballesteros, Fabio Kon, and Roy H. Campbell. A Detailed Description of Off++, a Distributed Adaptable Microkernel. Technical Report UIUCDCS-R-97-2035, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1997.

[92] G. M. Johnston and R. H. Campbell. An object-oriented implementation of distributed virtual memory. In Eugene Spafford, editor, *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale FL (USA), October 1989. USENIX.

[93] M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, pages 80–93, Dec 1993.

[94] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. In *SIGOPS European Workshop*, September 9-11 1996.

[95] M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum. Flip: an internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, Feb 1993.

[96] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi computer OS. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 191–203, Berkeley, CA, USA, January 1996. USENIX.

[97] Yousef A. Khalidi and MichaelÑ. Nelson. The Spring Virtual Memory System. Technical report, Sun Microsystems Laboratories Inc., 2550 Garcia Avenue – Mountain View, CA 94043, Feb 1993.

[98] Dilip R. Khandekar. *Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems*. PhD thesis, Computer Science, University of Utah, March 1996.

[99] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. The need for customizable operating systems. In *Fourth Workshop on Workstation Operating Systems*, pages 165–170, October 1993.

[100] Takuro Kitayama, T.Nakajima, and Hideyuki Tokuda. RT-IPC: An IPC Extension for Real-Time Mach. In *Proceedings of the 2nd Microkernel and Other Kernel Architectures*. USENIX, 1993.

[101] Steve Kleiman, Bart Smaalders, Dan Stein, and Devang Shah. Writing multithreaded code in solaris. In *Proceedings of the 1992 IEEE International Conference, COMPCON*, pages 187–192, 1992.

[102] Jurgen Kleinoder and Michael Golm. Metajava: An efficient run-time meta architecture for java. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems – IWOOOS*, Seattle, Washington, October 1996. IEEE.

[103] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford University, 1992.

[104] H. Kopetz. *The Time-Triggered Approach to Real-Time System Design*, chapter II-C, pages 53–65. Esprit Basic Research Series. Addison-Wesley, 1995.

[105] R. Lea, C. Jacquemmot, and E. Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–46, Sept. 1993.

[106] Jean-Pierre LeNarzul and Marc Shapiro. Un service de nommage pour un système à objets répartis. In *Actes Convention Unix 89*, pages 73–82, Paris, March 1989. AFUU.

[107] Jay Lepreau, Bryan Ford, and Mike Hibler. The persistent relevance of the local operating system to global applications. In *Proc. of the 7th ACM SIGOPS European Workshop*. ACM, September 1996.

[108] H.M. Levy. *Capability based Computer systems*. Digital Press, 1984.

[109] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[110] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Ashville (NC), 1993.

[111] Jochen Liedtke. On μKernel construction. In *Proceedings of the 15th SOSP*. ACM, 1995.

[112] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Weley, 1996. Java Series.

[113] M.C. Little and D.L. McCue. The Replica Management System: a Scheme for Flexible and Dynamic Replication. In *Proceedings of the 2nd Workshop on Configurable Distributed Systems*, Pittsburgh, Mar 1994.

[114] S. Lucco and D. Anderson. Tarmac: A language system substrate based on mobile memory. In *10th Int. Conf. on Distributed Computing Systems*, pages 46–51, 1990.

[115] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ class hierarchy for building Unix-like file systems. Technical Report R-88-1462, Department of Computer Science, Urbana, Illinois (USA), September 1988.

[116] H. Massalin and C. Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating System Principles*, pages 191–201, Arizona, December 1989.

[117] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[118] David Mazieres and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA. USA., May 1997. IEEE.

[119] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[120] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating system support for distributed multimedia. In *Proceedings of the 1994 Summer Usenix Conference*, Boston, Ma., June 1994.

[121] S.J. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, and H. Van Staveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 14:365–368, May 1990.

[122] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P.E. Osmon. Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel. In *2nd USENIX Symposium on Microkernels and other Kernel Architectures*, 1993.

[123] K. Murray, T. Stiemerling, T. Wilkinson, and P. Kelly. Angel: Resource Unification in a 64-bit Micro-Kernel. In *Proceedings of 27th Hawaii International Conference on Systems Science*, 1993.

[124] OMG - Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification (CORBA)*, 1995.

[125] E. Organick. *The Multics System: an examination of its structure*. MIT Press, 1972.

[126] J.K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer Usenix*, pages 247–256, June 1990.

[127] John K. Ousterhout, Andrew R. Cherenton, Frederick Douglis, M.N.Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, February 1988.

[128] John. K. Ousterhout et al. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[129] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *NKUUG Proceedings of the Summer 1990 Conference*, London (England), July 1990.

[130] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The Use of Name Spaces in Plan 9. *ACM Operating System Review*, pages 72–76, April 1993. Reprint from Proceedings of the 5th ACM SIGOPS European Workshop.

[131] G. J. Popek and B. J. Walker (editors). *The LOCUS Distributed System Architecture*. The MIT Press, 1985.

[132] D. Pountain. Taos: An innovation in operating systems. *Byte*, 16(1), 1991.

[133] D. Presotto. Plan 9 from Bell Labs — The Network. In *EUUG Proceedings of the Spring 1988 Conference*, London, England, April 1988.

[134] D. Probert, J.L. Bruno, and M. Karaoman. Space: A new approach to operating system abstraction. Technical Report CUCS-039-92, Dept. of Computer Science, University of California at Santa Barbara, 1992.

[135] Dave Probert and John Bruno. Building fundamentally extensible application-specific operating systems in space. Technical report, UCSB Computer Science, March 1995.

[136] C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Computing Systems*, 1(1), Winter 1988.

[137] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, , and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, 1995.

[138] D. M. Ritchie. The Evolution of the UNIX Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 63(8):1577–1593, October 1984.

[139] D.M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 7(7):365–375, July 1974.

[140] M. Rozier. Overview of the CHORUS Distributed Operating Systems. Technical Report CS-TR-90-25, Chorus Systemes, 1990.

[141] M. Rozier. Chorus. In *USENIX Micro-kernels and Other Kernel Architectures*, pages 27–28, Seattle, Washington, Apr 1992. USENIX Association.

[142] Vincent F. Russo and Roy H. Campbell. Virtual memory and backing storage management in multiprocessor operating systems using object-oriented design techniques. Technical Report R-89-1509, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.

[143] Wolfgang Schroder-Preikschat. Design principles of parallel operating systems—a peace case study—. Technical Report TR-93-020, International Computer Science Institute, Berkeley, California, 1993.

[144] Michael D. Schroeder. A State-of-the-Art Distributed System: Computing with BOB. In Sape Mullender, editor, *Distributed Systems*, chapter 1. Addison-Wesley, 2 edition, 1993.

[145] S. K. Shrivastava and S. M. Wheater. Implemeting fault-tolerant distributed applications using objects and multi-coloured actions. In *10th Int. Conf. on Distributed Computing Systems*, pages 203–210, 1990.

[146] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, Jan 1991.

[147] Abraham Silberschatz and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, 1994.

[148] C. Small and M. Seltzer. A Comparation of OS Extension Technologies. In *Proceedings of the 1996 USENIX technical conference*, San Diego, CA, January 1996.

[149] Christofer Small and Margo Seltezer. Vino: An integrated platform for operating system and database research. Technical report, Hardvard Computer Science Laboratory, Hardvard University, Cambridge, MA 02138, 1994.

[150] E. H. Spanfford. *Kernel Structures for a Distributed Operating System*. PhD thesis, Georgia Institute of Technology, Atlanta, Ga (USA), 1986.

[151] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for Real Time Systems. *IEEE Software*, pages 62–72, May 1991.

[152] M. Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, July 1981.

[153] V. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[154] L. Svobodova. File servers for network based distributed systems. In *ACM Computing Surveys*, volume 16. ACM, Dec 1984.

[155] Tao Systems. Elate. http://www.tao.oc.uk/, 1997.

[156] Tenma T., Y.Yokote, and M. Tokoro. Implementing perisistent objects in the apertos operating system. In *2nd IEEE Intl. Workshop on Object Orientationin Operating Systems*, pages 66–79, Sept 1992. Dourdan, France.

[157] See-Mong Tan, David Raila, and Roy H. Campbell. An object-oriented nano-kernel for operating system hardware support. In *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society.

[158] Andrew Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[159] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[160] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[161] A.S. Tanenbaum, M. F. Kaashoek, R. van Renesse, and H. BalH. The Amoeba distributed operating system—A status report. *Computer Communications*, July/August 1991.

[162] A.S. Tanenbaum, S.J. Mullender, and R. Van Renesse. Using sparce capabilities in a distributed operating system. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 558–563. IEEE, 1986.

[163] T.Nakajima, T. Kitayama, and H. Tokuda. Integrated management of priority inversion in real-time mach. Technical report, ART group, CMU, 1993.

[164] Mario Tokoro. Computational field model: Toward a new computing model/methodology for open distributed environment. In *Proceedings of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*. IEEE, Sep 1990.

[165] H. Tokuda, T.Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of the 1st USENIX Mach Workshop*. USENIX, Oct 1990.

[166] P. Tullman, J. Lepreau, B. Ford, and M. Hibler. User-level checkpointing through exportable kernel state. In *Proc. 5th International Workshop on Object Orientation in Operating Systems*, Seattle, WA, Oct 1996. IEEE.

[167] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson. Efficient, portable, and robust extension of operating system functionalit. Technical report, University of California at Berkeley, 1994.

[168] Andy Valencia. An overview of the VSTa microkernel. http://www.igcom.net/ jeske/VSTa/.

[169] L. van Doorn, P. Homburg, and A.S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proceedings of the 5th Hot Topics in Operating Systems (HotOS) Workshop*, pages 86–89, Orcas Island, May 1995.

[170] Alistair C. Veitch and Norman C. Hutchinson. Kea – A Dynamically Extensible and Configurable Operating System Kernel. In *Proceedings of the Third Conference on Configurable Distributed Systems (ICCDS'96)*, 1996.

[171] Paulo Verissimo and Hermann Kopetz. Design of distributed real-time systems. In Shape Mullender, editor, *Distributed Systems*, chapter 19. Addison-Wesley, 2 edition, 1995.

[172] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–267, May 1992.

[173] R. Wahbe, Lucco S., and Anderson T. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Dec 1993.

[174] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 14th ACM SOSP*, pages 203–216, 1993.

[175] T. Wilkinson and K. Murray. Extensible, flexible and secure services in Angel, a single address space operating system. In *1st International Conference on Parallel Architectures and Algorithms*, 1995.

[176] Tim Wilkinson and Kevin Murray. Evaluation of a Distributed Single Address Space Operating System. In *Proceedings of the 16th ICDCS*. IEEE, 1996.

[177] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Nov 1987.

[178] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *OOPSLA'92 Proceedings*. ACM, October 1992.

[179] Yasuhiko Yokote and Fumio Teraoka. The design and implementation of the muse object-oriented distibuted operating system. In *Proceedings of First Conference on Technology of Object-Oriented Languages and Systems*, October 1989.

[180] Stephan Zeisset, Stefan Tritscher, and Martin Mairandres. A New Approach to Distributed Memory Management in the Mach Microkernel. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, Jan 1996. USENIX.